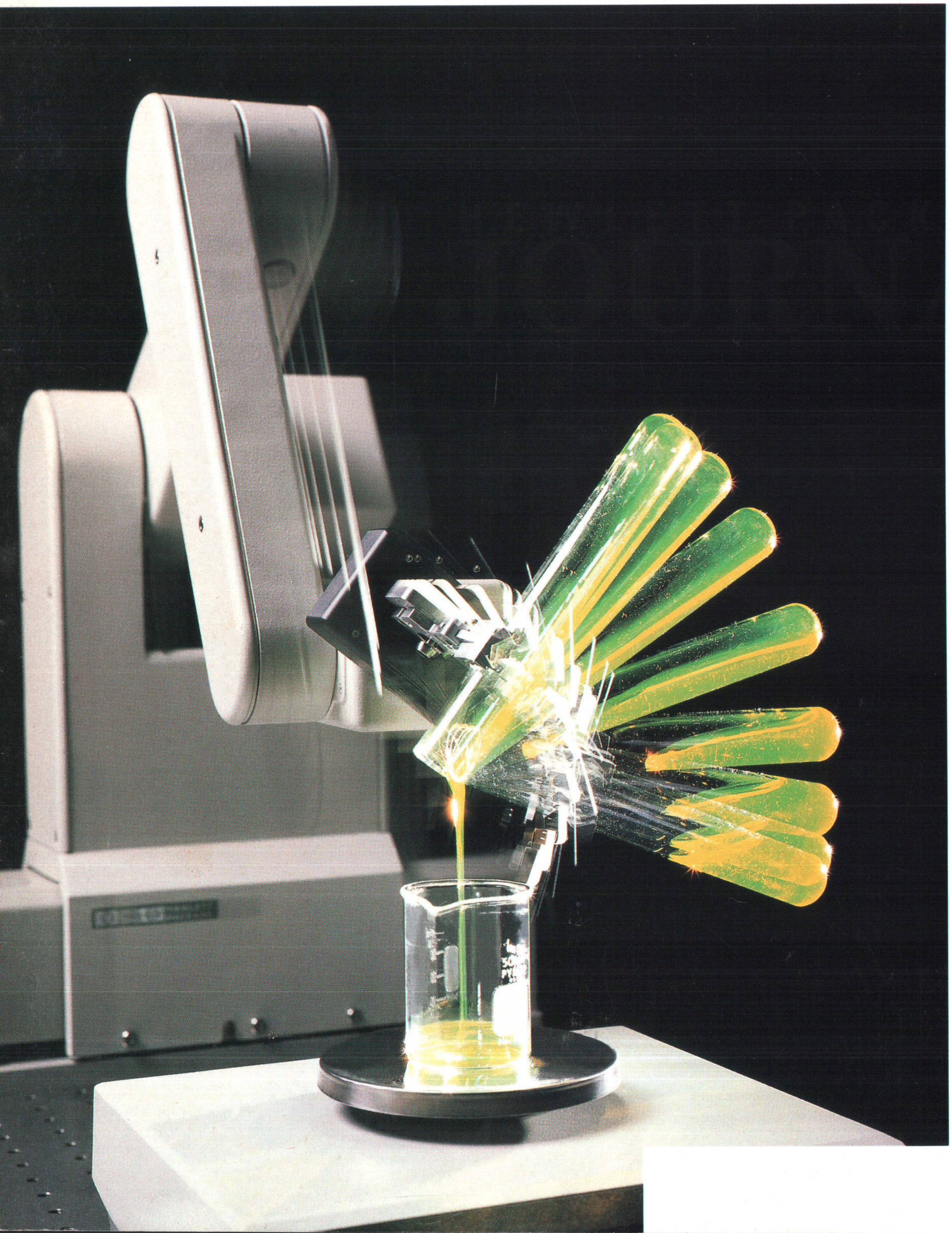
HEWLETT-PACKARD JOURNAL

June 1993



HEWLETT PACKARD

HEWLETT-PACKARD
JOURNAL

June 1993 Volume 44 • Number 3

Articles

Editor, Richard P. Dolan • **Associate Editor**, Charles L. Leath • **Publication Production Manager**, Susan E. Wright • **Illustration**, Renée D. Pighini
Typography/Layout, Cindy Rubin • **Test and Measurement Organization Liaison,** Sydney C. Avey

The Hewlett-Packard Journal is printed on recycled paper.

Departments

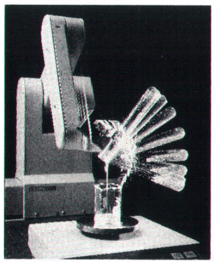
The Hewlett-Packard Journal is published bimonthly by the Hewlett-Packard Company to recognize technical contributions made by Hewlett-Packard (HP) personnel. While the information found in this publication is believed to be accurate, the Hewlett-Packard Company disclaims all warranties of merchantability and fitness for a particular purpose and all obligations and liabilities for damages, including but not limited to indirect, special, or consequential damages, attorney's and expert's fees, and court costs, arising out of or in connection with this publication.

Subscriptions: The Hewlett-Packard Journal is distributed free of charge to HP research, design and manufacturing engineering personnel, as well as to qualified non-HP individuals, libraries, and educational institutions. Please address subscription or change of address requests on printed letterhead (or include a business card) to the HP headquarters office in your country or to the HP address on the back cover. When submitting a change of address, please include your zip or postal code and a copy of your old label. Free subscriptions may not be available in all countries.

Submissions: Although articles in the Hewlett-Packard Journal are primarily authored by HP employees, articles from non-HP authors dealing with HP-related research or solutions to technical problems made possible by using HP equipment are also considered for publication. Please contact the Editor before submitting such articles. Also, the Hewlett-Packard Journal encourages technical discussions of the topics presented in recent articles and may publish letters expected to be of interest to readers. Letters should be brief, and are subject to editing by HP.

Please address inquiries, submissions, and requests to: Editor, Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304 U.S.A.

In this Issue



The HP ORCA (Optimized Robot for Chemical Analysis) system is a different kind of robot. In the words of Gary Gordon, one of the robot's designers and coauthor of the article on page 6, "Why would a company choose to optimize its first robot for initially doing chemistry instead of, say, circuit board or instrument testing? The answer is that there was a pressing customer need. HP is a major supplier of analytical instrumentation such as gas chromatographs. Such instruments help ensure the cleanliness of the food we eat and the water we drink by detecting harmful contaminants such as pesticides and industrial wastes. The first step in detection is sample preparation. It is tedious, time-consuming, and error-prone—in short, a ripe candidate for automation. Sample preparation entails reducing a matrix such as apples, pills, or blood serum to a clear concentrated fluid suitable for injection into the chromatograph. It involves such wet chemistry operations as crushing, weighing, centrifuging, extracting, and filtering. These are not enriching tasks for most people, yet tens of thousands of chemists are locked into the tedium of their repetition. What scale of sample-prep automation is appropriate? The variations from one procedure to the next rule out dedicated instruments. Instead a robotics approach fits best, interfaced with common laboratory apparatus such as centrifuges and balances. A search for commercial robots showed that they were typically too big and heavy, were optimized for precision assembly such as installing machine screws, and only accessed small work volumes. The HP ORCA robot system is quite different. It manipulates small (subkilogram) objects such as test tubes and probes in and out of tight places within a huge several-cubic-meter work volume at lively speeds. Much of the contribution of the HP ORCA system lies in the intuitive software interface and a gravity-sensing teach pendant that simplifies teaching the robot new tasks. The HP ORCA system can be easily integrated with other applications to create sophisticated, turnkey, automated systems."

In object-oriented programming technology, an object consists of some data and the methods or functions that can be used to access or operate on the data. Object-oriented programming is gaining wide acceptance for the development of large software systems because it makes developers more productive and makes software more maintainable and reusable. Central to many large commercial applications is a database management system, which allows efficient storage and flexible retrieval of large amounts of data. HP OpenODB (page 20) is an object-oriented database management system designed to support complex commercial applications. It combines a fast relational database management system with a specially developed object manager and has a client/server architecture. HP OpenODB provides tools that allow developers to use object-oriented modeling techniques to build a database. For data access, it has a procedural language called OSQL, which is based on the industry-standard SQL (Structured Query Language). It also offers run-time performance features such as late binding and schema modification and features to control access to data and ensure data integrity. The OpenODB model differs from other object models (there is no standard object model); how and why are explained in the article.

The HP Ultra VGA board is a video accessory card for the HP Vectra family of personal computers. (The same functionality is embedded in HP Vectra 486/U PCs.) Using hardware accelerators, the Ultra VGA board enhances video performance for graphics-intensive applications. It offers display resolutions up to 1024 by 768 pixels, refresh rates up to 72 times per second, and up to 256 colors. Software display drivers allow applications to take advantage of the performance enhancements. The article on page 31 traces the ancestry of the Ultra VGA board—from CGA to HGC to EGA to VGA—and discusses its design, including the hardware/software trade-offs, the use of a custom integrated circuit and video RAM memory devices, and the driver implementation.

POSIX stands for *Portable Operating System Interface,* a standard of the Institute of Electrical and Electronics Engineers. It defines a standard operating system interface and environment that guarantee that any POSIX application will run under any operating system that supports the POSIX interface, which is similar to the UNIX* operating system. As explained in the article on page 41, the MPE/iX operating system, which runs on HP 3000 Series 900 computer systems, does just that. In MPE/iX, the functions, services, and application program interface specified in the POSIX standard are integrated with HP's MPE XL operating system. MPE XL applications can access POSIX files and POSIX applications can access MPE XL files. Existing MPE XL applications are not affected. The integration team had no trouble visualizing MPE XL and POSIX as one operating system, but found challenges in the areas of directory structure, file naming, and security. The article describes their solutions.

Six papers in this issue are from the 1992 HP Software Engineering Productivity Conference. ▶ HP's Medical Systems Unit has been researching and experimenting with methods of preventing software failures in safety-critical medical applications. The paper on page 47 describes their software hazard avoidance process, a combination of testing for hazards and analysis aimed at prevention. ▶ If software can be reused, so can software tests. With this in mind, two HP printer divisions are implementing a software test library management system to make it easier to locate existing tests, determine their suitability, and combine them into test suites (see page 53). ▶ While the value of software inspections as a part of software development is well-accepted, in a busy R&D lab it's not always easy to get an inspection program started, maintain it once started, and meaningfully measure its success. The article on page 60 discusses one HP division's successful effort. ▶ Total Quality Control, or TQC, is a process improvement technique used extensively within Hewlett-Packard and by other companies. In the article on page 64, software engineers at HP's Imaging Systems Division tell how they successfully applied it to reduce the time required to localize, or translate, software text used in medical diagnostic ultrasound systems. ▶ A substantial number of engineering hours are spent developing system administration applications for the HP-UX* operating system, resulting in a major challenge in achieving user interface consistency. The article on page 71 describes the design of a special application program interface that enforces consistency and shields developers from the underlying user interface technology. ▶ Typically, error-handling code is dispersed throughout a program. In the article on page 80, Bruce Rafnel argues that this makes programs hard to write, read, debug, enhance, and reuse. He suggests handling errors in programs as they are handled in a database transaction recovery mechanism: the entire transaction is canceled as if it had never occurred if an error is detected anywhere in its processing.

R.P. Dolan
Editor

Cover

This photograph of the HP ORCA analytical robot in action was taken by author Gary Gordon with project manager Greg Murphy controlling the robot and artist Nicola Gordon providing art direction.

What's Ahead

In the August issue we'll have articles on HP's new line of high-brightness AlInGaP LEDs, the HP Tsutsuji logic synthesis system, the HP ScanJet IIc color scanner, the HP-RT (real-time) operating system design, the mechanical design for the HP 9000 Series 700i industrial workstations, the computation task distribution tool HP Task Broker, and three papers from HP's 1992 Software Productivity Conference—one on a defect management system, one on productivity and C++, and one on a modeling tool for real-time software systems. We'll also have a research report on surgical laser control.

HP-UX is based on and is compatible with UNIX System Laboratories' UNIX* operating system. It also complies with X/Open's* XPG3, POSIX 1003.1 and SVID2 interface specifications.

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

X/Open is a trademark of X/Open Company Limited in the UK and other countries.

ORCA: Optimized Robot for Chemical Analysis

This analytical PC peripheral is a congenial assistant, a sophisticated robotic teaching environment, and an interesting study of robotic architecture. Although optimized for the analytical laboratory, it also has applications in electronic test, quality assurance, and the clinical laboratory, where heavy commercial assembly robots are unsuitable.

by Gary B. Gordon, Joseph C. Roark, and Arthur Schleifer

Analytical chemists currently spend approximately two thirds of their time preparing samples to be placed into instruments for analysis. This sample preparation is not only tedious, but also subject to errors introduced by human interaction and human variation. At the same time, an ever-increasing number of samples to be analyzed each year coupled with a declining pool of skilled chemists has resulted in a pressing need for automation to improve the productivity of the analytical laboratory.

Samples arrive in the laboratory in liquid, solid, and gas form. Quantities range from the microgram or microliter size to tank cars filled with tons of material. The instruments that are used to analyze these samples, such as gas and liquid chromatographs, usually require that the samples be cleaned up to remove almost all of the components of the material except for the chemical compounds of interest. Sample preparation involves many steps, including weighing, grinding, liquid pipetting and dispensing, concentration, separation, and chemical reaction or derivitization. In most cases this work is done by hand, although instruments are available that perform particular preparation operations. To provide a system that performs a majority of the operations required for sample preparation requires a great deal of flexibility and versatility.

A robotic system seemed like the appropriate solution. But what type of robot? Robots designed for manufacturing and assembly are not well-suited for the analytical laboratory. The requirements for a laboratory robot go beyond the traditional characteristics associated with manufacturing systems. Since today's laboratory and instruments are designed for people, automation is difficult because not all the pieces of the laboratory are designed to work with robots. In contrast, assembly lines redesign the environment, process, and products to work easily with robots. It will be a long time before the chemical laboratory retrofits for automation.

Manufacturing robots in general are optimized for a different problem: very accurate positioning of often heavy payloads, in seldom reconfigured environments. These robots perform a small number of tasks very precisely in a small work volume. In contrast, an analytical robot is required to perform a wide range and number of tasks over an existing laboratory workbench and interact with existing laboratory containers and instruments. The same might be said of a robot for many other applications, such as electronic test, quality assurance, or clinical laboratory analysis (see "The HP ORCA System Outside the Analytical Laboratory" on page 9). To fit into existing laboratory environments, a robot must be installable without modification to the laboratory furniture. This will allow both rapid installation and easy relocation of the robot within the facility. The robot's work volume must allow the robot to reach the entire bench area and access existing analytical instruments. There must also be sufficient area for a stockroom of supplies for unattended operation.

The laboratory robot can be involved in three types of tasks during an analytical experiment. The first is sample introduction. Samples arrive in a variety of containers. It is time-consuming and a potential source of error for the operator to transfer the sample from the original container to one that is acceptable for automation. The robot, however, can be trained to accept a number of different sample trays, racks, and containers, and introduce them into the system.

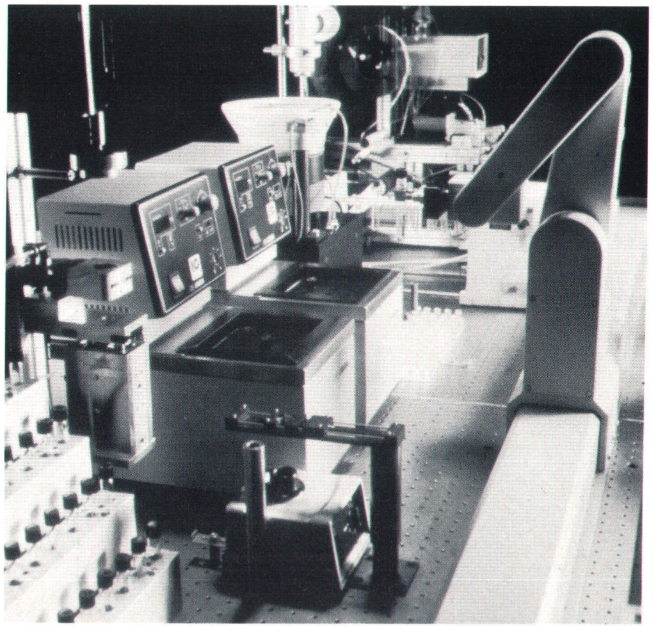


Fig. 1. Typical analytical laboratory work volume. (Photo reproduced with permission of Scitec SA, Lausanne, Switzerland.)

The second set of tasks for the robot is to transport the samples between individual dedicated automated stations for chemical preparation and instrumental analysis. Samples must be scheduled and moved between these stations as necessary to complete the analysis. The third set of tasks for a robot is where flexible automation provides new capability to the analytical laboratory. There will always be new chemical samples that require analysis steps that have never been automated. To prototype the automation of such steps, the robot must be programmed to emulate the human operator or work with various devices. This last use may require considerable dexterity for a robot. All of these types of operations are required for an effective laboratory robot.

Additional considerations for a laboratory robot are that it be the size of a human arm and have the dexterity needed for interaction with current chemical instrumentation. Interchangeable end effectors (robot fingers) are required to allow the robot to work with the wide range of existing containers and consumables used in sample preparation. The robot should provide a simple and clean work area with no external wires to catch on glassware or instruments.

After evaluating a number of existing robots for this application, it was finally concluded that a robot could be designed that was optimized for chemical sample preparation and other applications that have similar requirements, as mentioned above. The results of this analysis are the concept and design described in this article. The new HP analytical robot is called ORCA, which stands for Optimized Robot for Chemical Analysis. Fig. 1 shows it installed on a lab bench.

An anthropomorphic arm mounted on a rail was chosen as the optimum configuration for the analytical laboratory. The rail can be located at the front or back of a workbench, or placed in the middle of a table when access to both sides of the rail is required. Simple software commands permit moving the arm from one side of the rail to the other while maintaining the wrist position (to transfer open containers) or locking the wrist angle (to transfer objects in virtually any orientation). The rectilinear geometry, in contrast to the cylindrical geometry used by many robots, permits more accessories to be placed within the robot workspace and provides an excellent match to the laboratory bench. Movement of all joints is coordinated through software, which simplifies the use of the robot by representing the robot positions and movements in the more familiar Cartesian coordinate space.

The physical and performance specifications of the HP ORCA system are shown in Table I.

A robot alone does not satisfy the needs of laboratory automation. Besides the physical aspects of the robot, the system must be able to work with other devices, computers, and software. The other major development of the ORCA project was the control software, which is called Methods Development Software 2.0, or MDS. MDS runs on the HP Vectra and other PC-compatible computers under the Microsoft® Windows operating environment. It is designed to allow instruments to be added easily to the system. By the use of industry-standard communication interfaces, MDS can configure devices, and procedures can be developed to control and collect data from external devices. It is designed to be extensible; new modules can be added to the system at run time. MDS is also designed to be remotely controlled by other programs. This allows the laboratory robot system to be a server in a hierarchical automation system.

Table I
ORCA Robot Arm Hardware Specifications

Arm	Articulated, rail-mounted
Degrees of freedom	Six
Reach	±54 cm
Height	78 cm
Rail	1 and 2 m
Weight	8.0 kg
Precision	±0.25 mm
Finger travel	40 mm
Gripper rotation	±77 revolutions
Teach pendant	Joy stick with emergency stop
Cycle time	4 s (move 1 inch up, 12 inches across, 1 inch down, and back)
Maximum speed	75 cm/s
Dwell time	50 ms typical (for moves within a motion)
Payload	0.5 kg continuous, 2.5 kg transient (with restrictions)
Vertical deflection	< 1.5 mm at continuous payload
Cross-sectional work envelope	1 m^2
Power requirements	100V, 120V, 220V, or 240V (+5%, –10%), 350 VA, 47.5 to 66 Hz
Operating environment	5°C to 38°C at 0 to 90% RH (noncondensing)

Most previous robots were programmed in coordinate space with computer-like languages. The HP ORCA system, on the other hand, is taught by first demonstrating to the robot a move, using another new development, the gravity-sensing teach pendant (see "Gravity-Sensing Joy Stick," page 12). The move is then given an intuitive name, for example get_testtube. Later, this or any other move can be called out by name and built into higher-level procedures. Simple to grasp yet powerful, this concept is easily expanded to control an entire benchtop of laboratory equipment. The user is freed to think about the greater task at hand rather than specific software details.

Having these components leads to our vision of the automated laboratory bench. For the first time, it is possible to provide automation for the analytical laboratory all the way from sample introduction to the final report without human intervention. The HP ORCA system provides the physical glue to tie together the individual chemical instruments as well as the information or data bus, so that the system can request, acquire, and distribute information to all components of the workbench.

ORCA System Overview

The HP ORCA system is shown pictorially in Fig. 2, which shows the major functional blocks. In the broadest sense, the input to the HP ORCA system is a high-level command, such as to analyze a drug (such as aspirin) for purity. Near the output end, the ORCA electronics deliver strings of millions of individual millisecond-by-millisecond commands to the six robot servos. To keep this transformation task from becoming overwhelming, it is broken down into manageable hierarchical levels.

Input to the system begins with a user interacting with the PC, which is running MDS. MDS provides experiment control through its own programming language. In conjunction with other applications on the PC, such as HP ChemStation software, MDS can also be used for chromatograph control. MDS consists of a core system, which is used to build and run reusable tasks called procedures, and one or more modules, such as the robot module.

The MDS robot module accepts MDS procedure commands for controlling the robot and parses the commands into endpoints for each of the segments of the robot moves. A typical robot move might be from a starting point taught to the robot and named OverBalance to an ending point named OnBalance. These endpoints define straight-line segments. The output of the MDS robot module, and in fact the output of the PC, is a string of taught Cartesian waypoints representing the robot moves, sent every few seconds over an HP-IB link (IEEE 488, IEC 625) to the kinematics processor.

The kinematics processor consists of software running on a dedicated 68000 microcomputer. One of its tasks is coordination of the six axes of the robot to provide smooth motion and arrival at the Cartesian endpoints. This is accomplished in part by interpolation. The kinematics processor also handles tool offsets, so that, for example, a test tube can be rotated about its lip instead of its point of support. The final function of the kinematics processor is to compute the acceleration and deceleration profiles and the speed of each move so that motion is smooth and the axes' speed limits are not exceeded.

Physically, the kinematics processor shares a cabinet with the servo power supply. This cabinet has no controls and can be located out of the way under the bench. The output of the kinematics processor is coordinated position commands in joint space for motion along the path, sent over an RS-422 bus to the robot at a frame rate of 25 Hz.

The last functional block of the system is the joint interpolators, which are distributed throughout the robot. The 25-Hz commands from the kinematics processor are coarse enough that they would cause the robot to vibrate and growl, so instead they are linearly interpolated into 1.25-millisecond demand signals, which are sent to the digital servomechanisms.

(continued on page 10)

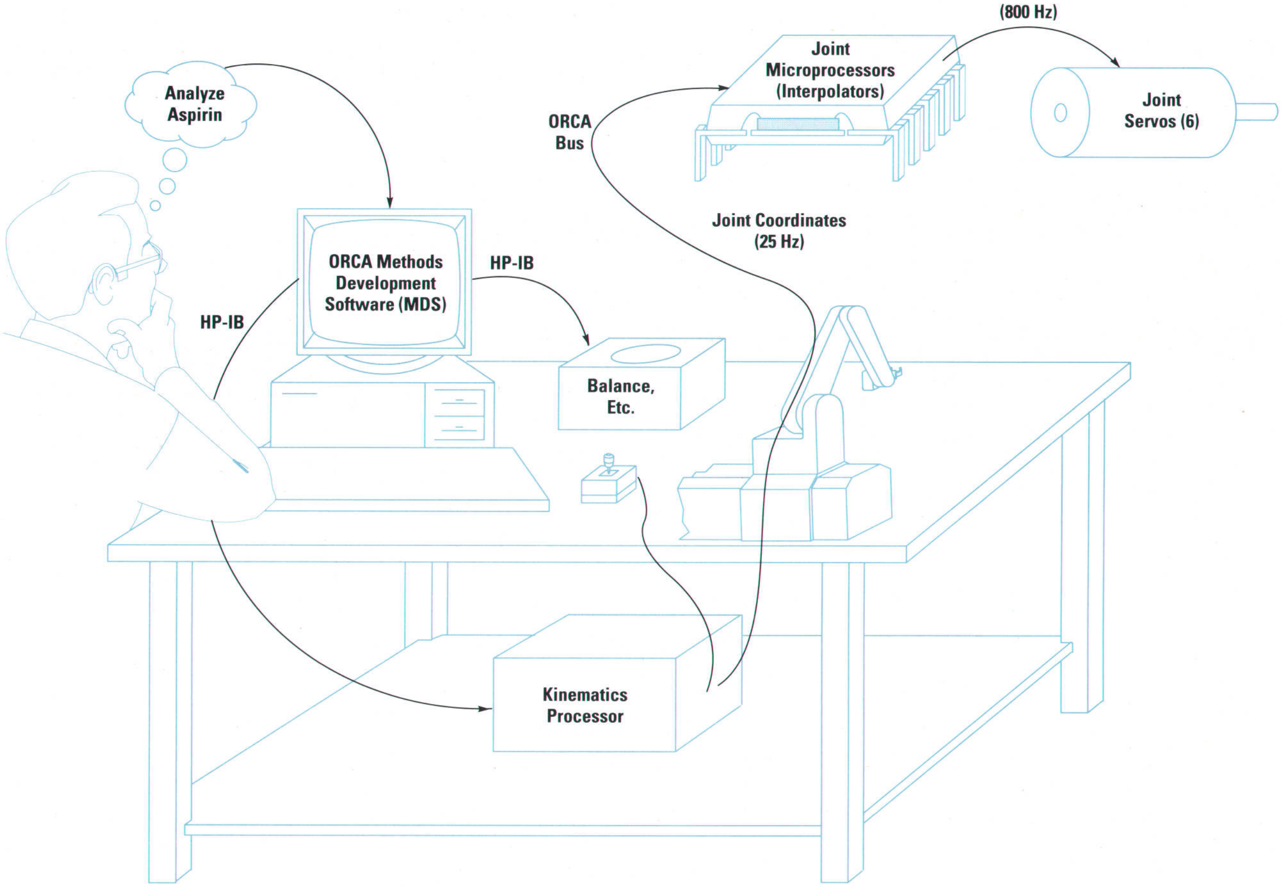

Analyze
Aspirin
ORCA Bus
Joint Microprocessors (Interpolators)
(800 Hz)
Joint Servos (6)
Joint Coordinates (25 Hz)
ORCA Methods Development Software (MDS)
HP-IB
HP-IB
Balance, Etc.
Kinematics Processor

Fig. 2. ORCA robot system diagram.

The HP ORCA System Outside the Analytical Laboratory

There are many applications in industry where the precision of a manufacturing robot is not required. Often, in fact, such robots are genuine misfits. They are costly, bulky, heavy, and too complex to program for these simpler jobs. The HP ORCA system, on the other hand, is much better suited for these lighter tasks. It weighs 1/10 as much as its industrial relatives, yet is just as reliable, and it is far easier to use because it behaves like an appliance or a PC peripheral. The robot can be connected to a PC, shown a task to do, and put to work without much fuss.

There are numerous small applications in manufacturing where one might not normally think of using robots. They include many of the repetitive tasks people now perform in assembly, test, and quality assurance.

At HP, two areas being looked at are instrument front-panel circuit board test and instrument final test. Front-panel controls are not very accessible electronically, and typically require human intervention to verify the operation of knobs, switches, and displays. These panels are quite amenable to robotic automation. Robotic fingers can manipulate controls, and carried sensors can monitor displays. Such robots fulfill a niche in medium-scale assembly, where the number of products to test is too many to do by hand, yet not enough to justify designing, building, and keeping track of hard-tooled test fixtures.

Other manufacturing uses lie in assembly tasks for which the HP ORCA system's 0.25-mm precision is sufficient. This rules out board loading and fastener inserting. On the other hand, as an assembly operation is studied, it often becomes apparent that there are many tasks in which dramatic savings and quality improvement can be had through robotics. Two examples are pick-and-place assembly and adhesives application.

The ORCA group at HP was encouraged when other HP groups came to us and wanted robots to try out. Four early users were in two areas: gas chromatograph manufacturing at HP's Little Falls site in Wilmington, Delaware, and column manufacturing at the Scientific Instruments Division in Palo Alto, California. The following story relates how the ORCA/NC lathe project came about at Little Falls.

The ORCA/NC Lathe Project

The ORCA/NC lathe project, conceived in a typical HP aisleway conversation, was initiated to address both current business needs and visions for the future. A shop supervisor at HP's Avondale, Pennsylvania Division, where many HP ORCA system components were fabricated and assembled, was able to bait an R&D manager with the dream of a robot building itself. The manager promptly found a prototype unit to donate to the shop, and an investigation of potential applications began. Five months later, the latest-model HP ORCA system was operating an unattended NC lathe.

Potential shop applications for a small flexible robot included the transfer of parts between hydraulic presses in a sequence of staking operations, the loading of components on a pneumatic manifold, and the loading and unloading of parts on a machine tool, such as a lathe or mill. The application of the lathe loader was chosen because it was the simplest in concept. It contributed to profitability by using outdated equipment (a ten-year-old lathe) and by reducing the shop cost driver rate, an important metric determined in part by the total of unattended machining hours.

The experience gained in the lathe loader project was expected to provide a knowledge base for future projects. It would help in establishing guidelines, planning resources, and scheduling more complex applications. Also, an understanding of the HP ORCA system's capabilities with respect to the fabrication business was needed. The experience objective was made explicit because other lathe autoloaders exist that are more accurate and simpler in design than the HP ORCA system.

The project team consisted of a fabrication process engineer, a tooling designer, and a journeyman machinist. The first challenge was positioning ORCA on the lathe. We wanted to access the full width of the lathe so we chose to bolt ORCA's rail onto the machine bed and operate the robot completely within the lathe shields. Although the ORCA rail ends are outside the shields, this decision meant that the robot would be operating in an environment with coolant and metal chips. Since the lathe is still used 80% of the time for bar-fed jobs, the robot is protected when not in use by being parked behind the turret under a plastic bag. ORCA-loaded jobs are run without coolant and the robot is isolated from chips by a telescoping cover that extends over the rail cover.

The ORCA/NC lathe system includes an ORCA robot, a Hardinge HNC lathe, an HP Vectra QS/20 PC, and an optointerface board. The custom hardware consists of four part staging magazines, special grippers for the robot, a V-block, and a secondary rail cover. The vertical magazines are mounted across the lathe bed behind the lathe turret and hold 75 parts each. The grippers are oriented such that the axis of the gripped part is perpendicular to ORCA's twist axis. The V-block is self-centering and spring-loaded, and is mounted on the lathe turret. The secondary telescoping rail cover is attached to ORCA's torso casting.

In one cycle of the ORCA/NC process, running 50 seconds, the robot removes a part from the magazine, then pushes the part into the V-block and moves to a safe position. The lathe turret moves so that the V-block stuffs the part into the collet. The collet closes, the part is machined, the spindle stops, and the collet opens. ORCA removes the part and drops it down a slide leading to a box of completed parts.

The HP ORCA system and the specialized tooling were set up in a lab for development and moved onto the lathe two weeks before startup. In the lab, parts of the overall concept were simultaneously prototyped, then tested together. For example, the custom gripper fingers were revised five times. The purpose of two of the revisions was to increase the maximum gripping force transmitted to the part. ORCA's grip had to exceed the holding force of the magazine and V-block by enough force to pick and place parts reliably.

Critical requirements for success were robustness over time and the ability to run for one shift without operator intervention. Robustness is defined as the ability to run day after day without a crash or robot recalibration or repair. The application ran in the lab for thousands of cycles.

Overall system control is vested in an ORCA MDS program that calls robot subroutines and starts, pauses, and stops the lathe program via the interface board. To start the application cycle, the lathe program is loaded and started, and then the MDS program is started. The robot positions within the move subroutine programs were rough-taught with ORCA's joy-stick teach pendant and refined by keyboard input. Accuracy and repeatability of the movements were further enhanced by unidirectional programming.

The ORCA/NC system is currently used to machine two brass and aluminum valve stems. Only one part was used in the system's development. The other part was implemented by the machinist two months after release of the application. In six months of operation, about 16 hours per week, there have been no problems. This is partially because of the robustness of the system and very much because of the ease of use of the system software.

The ORCA/NC lathe project met every initial objective, is a good example of teamwork, and has become a shop showpiece. The machine shops at Avondale were bought by two former managers in November of 1992, when HP moved to Little Falls. The HP ORCA system continues to run 16 hours per week at the new company, American Manufacturing Technology.

Nancy Adams
Manufacturing Process Engineer
Little Falls Operation

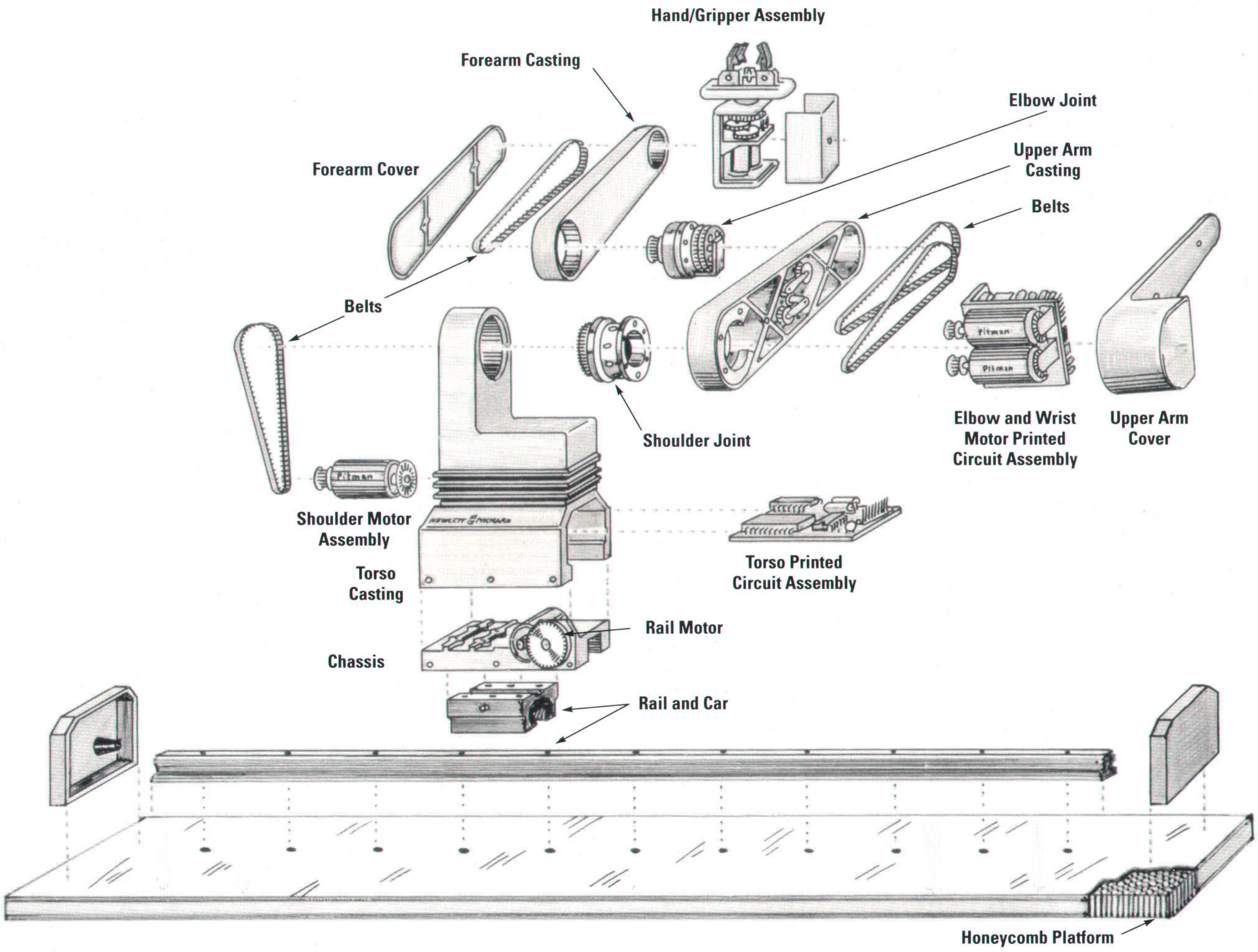


Fig. 3. Exploded view of the robot arm.

Mechanical Design

Robot design benefits greatly from a careful optimization of lightness, stiffness, speed, and payload. Airplanes seem almost simpler in comparison, with some being able to carry 100% of their empty weight, compared to 10% to 20% for robots.

Playing off against the performance goals were economic considerations, so exotic materials such as titanium and carbon fiber were not seriously considered. Further, with the continuing shrinking of electronics, a decision was made to imbed much of it in the structure itself. This diminished the payload slightly, but greatly simplified the wiring and improved the reliability.

The ORCA industrial design achieves other less-tangible goals with little or no additional expense. It provides a smooth structure for easy cleaning, accessibility for maintenance, a compact design for dexterity in tight places, and an attempt to achieve pleasing lines.

ORCA, unlike any other commercial robot, is an anthropomorphic shell or exoskeleton, with its chambers tightly packed with six axes of servo control electronics. Fig. 3 shows an exploded view. The shells are of aluminum, chosen over plastics for thermal conductivity. They draw out what little heat is generated by the pulse-width-modulated servos, and spread it over the surface of the robot, where it is dissipated by convection. The shells are ribbed internally for torsional rigidity so that the robot is mechanically sound with the covers removed, allowing easy service.

ORCA is scaled roughly to the proportions of humans. The similarity continues with its "muscles"—its motors—which are physically displaced toward the torso; the elbow and wrist motors are situated at the shoulder. This reduces the static moment loads the robot must carry because of its own weight.

Particular effort went into refining the hand to cut weight and bulk to a minimum. One interesting feature is that the axes that pinch and rotate the fingers are coupled mechanically and in software. The fingers are mounted to parallel gear racks, which are opened and closed by spinning a pinion gear that engages them both. Without coupling, whenever the finger bar was rotated, it would wind the finger racks around the center pinion gear, open the fingers, and drop the object. The easy solution is to feed a proportional correction signal to the pinch servo any time the finger bar is commanded to rotate; software is cheaper than mechanisms. Fig. 4 shows the interior of the hand.

Because of the immense advantages of keeping the hand light and small, gear technology was pushed near its limits.

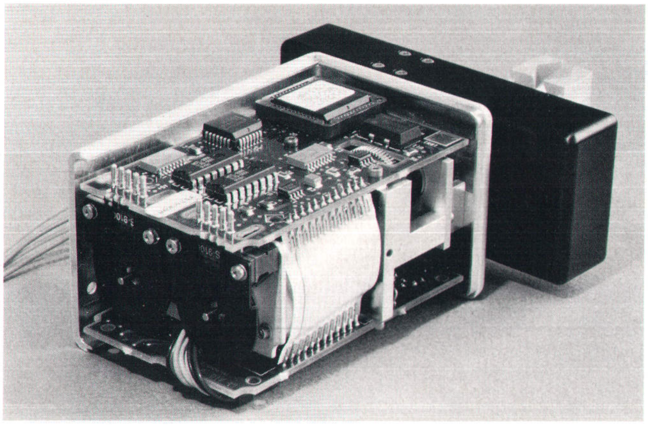


Fig. 4. Interior of robot hand.

Hardened ground gears and tight machining tolerances are employed to maintain the five-year minimum lifetime goal for all mechanisms.

ORCA is supplied mounted on an optical-bench-style tabletop, which comes in several sizes. This surface forms a stable platform, and its grid of threaded holes provides convenient attachment points as well as reference locations for instruments and accessories. A linear rail assembly is fitted to either the rear, the front, or the centerline of the surface. The chassis contains the rail motor, which propels the torso by engaging a steel-cable-reinforced plastic chain stretched tightly alongside the rail. A simple flat cable similar to that used in printers also folds into the rail cavity and carries dc power and the serial ORCA bus, which affords bidirectional communications with the six robot joints.

The most interesting feature of the shoulder and elbow joints is their harmonic drive reduction units, shown in Fig. 5. These drives use a ring gear with internal teeth, which engages a flexible inner gear shell. The flexible inner gear has slightly fewer teeth than the outer rigid ring gear. For example, in the shoulder reducer, the ring gear has 102 teeth, while the flexible gear has 100 teeth. Lobes inside the inner flexible gear shell force it into an egg shape and into contact with the rigid outer ring gear. One rotation of the lobe assembly causes the two gears to displace relative to one another by two teeth, or 2% of one revolution of the flexible gear. Thus the reduction ratio of one stage is 50:1, or far larger than that obtainable with a single stage of pinion gears. Furthermore, since many teeth are engaged, the torque transmitted can be substantial. Because the engagement is compliant, harmonic drives exhibit little or no backlash. They are more expensive, but are common in robots and other high-performance applications because they perform better and reduce the total number of parts required.

Proceeding outward towards the hand, the moment increases, and saving weight becomes more and more important. Every gram saved becomes a gram of payload. Saving bulk is of equal importance. The hand, shown in Fig. 4, packs two servos for rotating the finger bar and changing the grip, all into a cozy 20 cubic inches. Since the drive train for the grip extends through the rotating finger bar, commands to these two servos must be coordinated; otherwise, rotating the finger bar would drastically change the grip on the object held, as explained previously.

The Kinematics Processor

The kinematics processor is one of the most interesting blocks of a robot, and gives an insight into how robots work. Its input is position data received from the PC over the HP-IB every few seconds, directing the robot to move in a coordinated manner from the last Cartesian position to the next. For example, a command string might direct the robot to move from coordinates over a test tube rack to coordinates over a balance a meter away. Altogether it looks like a complicated task, but when broken down into individual steps it is easy to grasp.

Robot Joint Space and Cartesian Space. One important simplification in understanding robots is to understand the difference between two different coordinate spaces: the Cartesian space in which the task is defined, and the robot joint coordinate space in which ORCA operates.

It takes six coordinates to specify the position and orientation of an object in space. In our familiar Cartesian system, these coordinates are x, y, z, yaw, pitch, and roll. ORCA is fixed in yaw, so we restrict our interest to the other five degrees of freedom. A sixth degree is added, however, and that is pinch, to control the fingers. In friendlier terms, we will refer to these six degrees of freedom as rail (x), reach (y), height (z), bend, twist, and grip.

Robot joint space is defined as the joint positions of the robot that must be established to place an object in space with a specified position and orientation. ORCA has six movable joints, each controlled by a servo. Setting each of these correctly will position an object in Cartesian space. The six joints represent the six degrees of freedom in robot joint space. They are rail, shoulder, elbow, wrist, twist, and grip. Note that only rail, twist, and grip are the same in both coordinate systems.

Part of the reason for belaboring these differences in coordinate systems is that as the tasks of the robot are divided up, some are distinctly easier to perform in Cartesian space and others in robot joint space. In any case, the transformation must eventually be made to the joint space coordinate system to control the robot.

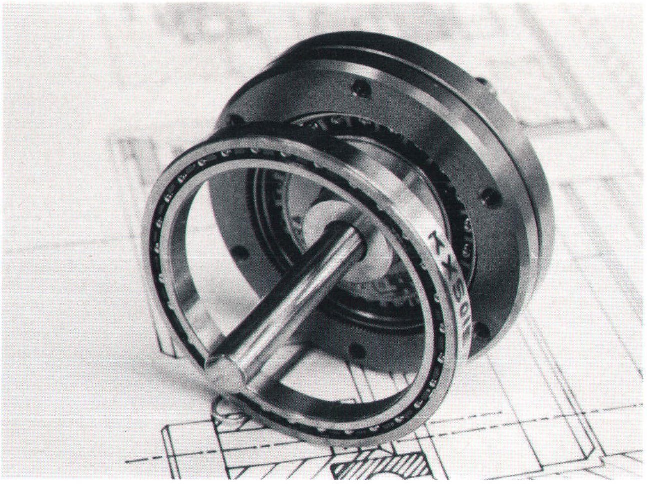


Fig. 5. Harmonic drive reduction unit.

Gravity-Sensing Joy Stick

What is the most intuitive way to teach moves to a robot? For the HP ORCA system, joy sticks rapidly became front runners. They are portable, allow control of many degrees of freedom, and combine delicate movements with the capability of high speeds.

The problem is that six degrees of freedom need to be taught: rail (x), reach (y), height (z), bend, twist, and pinch. Two degrees of freedom are easy to master. Everyone who picks up the joy stick can intuitively fly the robot in traverse and reach (x and y). The question was, "What would be intuitive and affordable for commanding the other axes?" Shift keys on the stick housing are common but they rarely become habitual, and relearning which one to press distracts one's attention away from teaching the robot.

The HP solution is to add gravity sensors to the stick box that sense its orientation and dynamically reassign the axes, transforming the joy stick into a teach pendant. Fig. 1 shows the joy stick and the sensors. The three mutually perpendicular tilt switches required to sense the six possible orientations are mounted along the (1,–1,1), (–1,0,1), and (1,1,1) vectors.

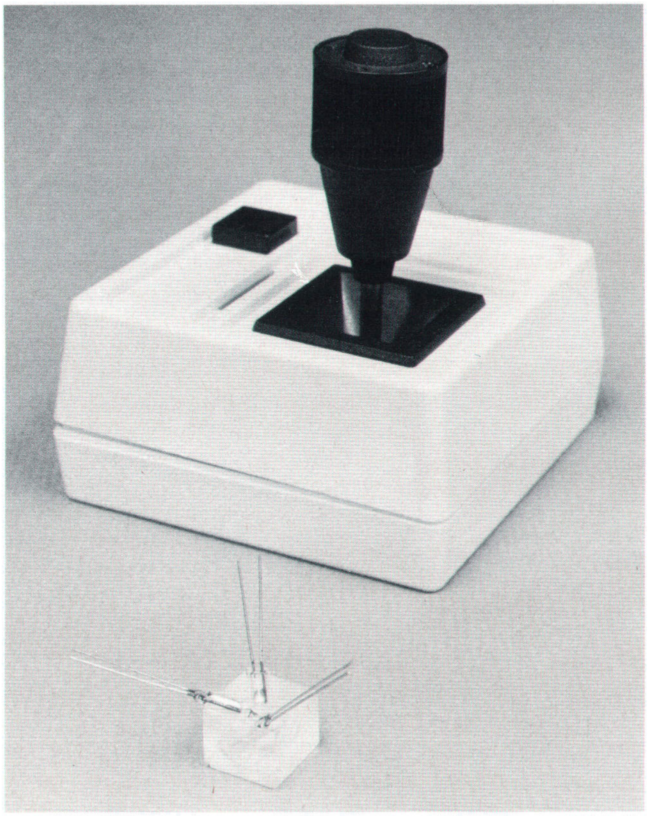


Fig. 1. ORCA robot joy stick and orientation-sensing tilt switches.

In use, tilting the joy stick to its right changes the stick from controlling traverse and reach to controlling height and reach. In other words, if the joy stick won't at first bend or move in the desired direction, reorient the box so that it will. Pointing the stick right, left, towards the user, or even down all have the expected result. The only exceptions are that wrist twist and bend are controlled by pointing the stick away from the operator, and twisting the knob atop the stick always controls finger grip. It's a teaching system one never forgets. The user moves the robot in Cartesian space, not robot joint space, which is a tremendous simplification for the user.

Controlling the Robot. The first step in robot control is to define the straight line along which the coordinated motion will occur. This is a matter of straightforward interpolation in Cartesian space. For example, when the robot is 40% of the way to its destination, the six individual coordinated degrees of freedom (x, y, z, bend, twist, grip) will each be 40% of the way to their final values. If the final twist of a pouring operation starting at 50 degrees is to be 100 degrees, for example, then at the 40% point, the instantaneous twist command will be 70 degrees.

The second step is to compute the velocity and acceleration profiles so that the robot will accelerate up to speed, traverse a distance, and decelerate and smoothly stop at the end of the move. Here the task is twofold. One is to allow the MDS software to control the speed of the move. The second is not to exceed the hardware performance limits of any robot axis. This situation arises because an articulated robot like ORCA is capable of much faster speeds in some directions and portions of the working space than in others. For example, if the arm is fully outstretched it can move vertically very rapidly, but if commanded to move inward towards the torso, its speed is limited for a moment as the elbow tries to accelerate vertically to infinite velocity. The HP ORCA system avoids such situations by limiting the velocity and acceleration to the lower of two numbers: the command from the PC or the limits of the joint servos.

The kinematics processor code is mathematically intensive and requires a fairly powerful 16-bit microprocessor. The processor has quite a number of tasks to perform in addition to computing waypoints along the straight-line trajectory between the robot's initial and final positions. It takes the processor 40 milliseconds per waypoint to complete all of its computations and tasks. Thus the determination of where the robot should be is not continuous at this point but periodic, and the period is relatively long. However, this is just an intermediate step in the control process. The information is still in the wrong coordinate system and is far too coarse for smooth motion control of the robot.

Immediately after computing each coarse Cartesian waypoint (every 40 ms), the kinematics processor converts the point into robot joint space coordinates. The x coordinate is easy to transform since it is the same in both spaces; the robot merely moves a certain number of millimeters down the rail to the new x coordinate. The same is true with grip commands to the gripper. The y (in and out) and z (height) coordinates are slightly more complicated to transform, and use trigonometry to compute the shoulder and elbow joint angles in robot space. The wrist joint is not a factor if it is kept horizontal; its angle bears a simple relation to the shoulder and elbow joints.

In addition to transforming the waypoints from Cartesian space to robot joint space, the kinematics processor also applies the tool offset parameters, if there are any, so that a test tube can be rotated about its lip instead of its point of support, for example. The kinematics processor outputs a joint-space vector every 40 ms and sends it to the six joint servos for further processing.

Joint Servos. Two wires of the four-conductor ORCA bus that snakes through the robot carry the serial RS-485 joint commands to the joint servos, which are embedded in the robot shell. The other two conductors carry unregulated 32V power. The data structure sent over the ORCA bus is shown in Fig. 6. Each pair of joints is serviced by a microprocessor which strips off its command from the bus at 40-ms intervals. After each joint's two-byte position command is sent, an idle space is provided for the joint to send back its position and status.

The fourth step in the robot control process takes place in the joint microprocessors, which further interpolate the 40-ms interval down to 1.25-ms position demands for the joint servo. What is the purpose of all this interpolating? If the 40-ms points were sent directly to the servos, the robot motion would be jerky. Yet it is uneconomical to generate them faster than 40 per second; that would take an unnecessarily fast kinematics processor and would take up too much bus communication time. There is an easier way to get the fine increments to send to the joint servos to ensure smooth motion. This interpolation step is the task of the joint microcomputers, which divide the motion down into 1.25-ms intervals. This interpolation produces smooth, nonjerky robot motion by keeping the step noise at a frequency well beyond the passband of the robot servos.

This is a particularly easy interpolation to accomplish, since the number of steps chosen, 32, is a power of two. Interpolation then consists of subtracting successive positions, dividing the difference by 32 (a right-shift of a binary number by 5 bits), and successively adding that quotient into an accumulator initialized to the starting servo demand position.

A consequence of this design expedient is that the robot actually moves in slight scallops, 40 ms long, since a straight line in robot joint space is curved in Cartesian space. However, these deviations are on the order of thousandths of an inch and are insignificant.

The remaining task performed by the joint microprocessors is to close the digital servos at each joint. These servos use incremental encoders, dc motors, and pulse-width-modulated amplifiers—technology borrowed from HP plotters.[1] Briefly, each joint demand position is first subtracted from the actual position of that joint to generate a position error value. The servo motor is then commanded to move at a velocity proportional to that position error, with the velocity and position of the joint servo motor being derived from the incremental encoder. Since each joint position is fed back to the PC, the control software knows if the robot has bumped into anything, and can also employ integral control to correct small errors such as sag of the arm caused by the influence of gravity.

A new HP technology introduced in the HP ORCA system is digital absolute position encoding (see "Absolute Digital Encoder," page 14) at each of the major joints. Its purpose is to allow the mechanism to ascertain its position when first powered up.

Application Development Environment

Although clearly the most conspicuous element, the robot is but a piece of a total automation system that also involves controlling and collecting data from analytical instruments and common laboratory devices, such as pH meters and balances. The HP Methods Development Software (MDS), written to address this need, provides a development environment for creating automation systems with laboratory robotics. MDS runs under Microsoft Windows on an HP Vectra or other PC-compatible computer. This choice was based on users' preferences for a PC-based system, compatibility with HP ChemStations, and the features that Microsoft Windows provides for a multitasking graphical user interface.

The targeted customer for MDS is a laboratory robotics application developer, typically an analytical chemist with instrumentation and BASIC programming experience. These developers create applications that a technician runs and monitors. Robotics programming has to be presented in a conceptually simple format that makes it easy for the chemist to create tasks, which can then be combined to form an application.

Again, the differences between the use of a robot in the laboratory and the manufacturing environment were considered. Whereas a manufacturing robot is typically programmed to repeat a small set of tasks in a world that can often be

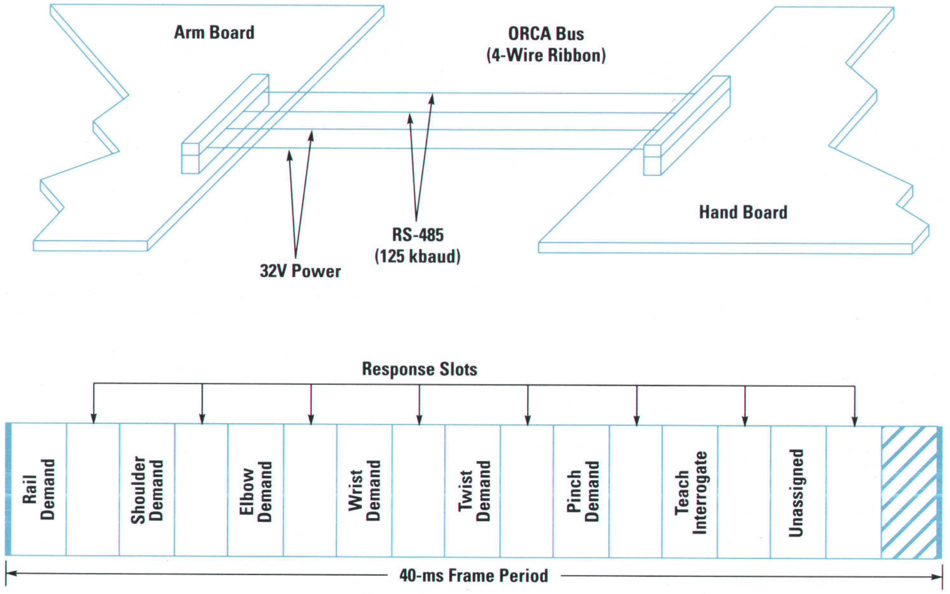


Fig. 6. ORCA bus and data structure.

defined with information from CAD drawings, a laboratory robot is used to perform a wide variety of tasks in a world where very little predefined knowledge is available. The laboratory robot must be taught how to interact with test tubes, vials, racks, balances, and other instruments.

Teaching a robot all of the individual positions and trajectories it must follow for every step in a laboratory application would be very time-consuming and tedious. The teaching process can be greatly simplified by providing a mechanism for teaching small manipulations instead of individual positions. These small manipulations can be used (and, most important, reused) as building blocks. This idea led to the concept of a robot *motion*. A motion in its simplest form is a sequence of robot positions. The motion is taught interactively using a special motion editor and the robot teach pendant. The motion is given a descriptive name, which is used in a program to have the robot execute, or move through,

Absolute Digital Encoder

The HP ORCA robot uses digital servos with incremental encoders, which need to be initialized when the robot is first turned on. Many digital servo products, such as plotters and impact printers, can initialize themselves by traversing to the ends of each axis. For many other applications, such as, for example, car seats with memory, robots, or machine tools, this expedient is either impractical or risky. One solution is to add a potentiometer, but this carries a cost, complicates the wiring, and is incompatible with leadscrew drives.

The encoder developed for the HP ORCA system is a small package less than 1 cm thick, which fits at the rear of the motor ahead of the incremental encoder. It uses a system of permuting gears, whose phases are measured to ascertain motor revolution number. Fig. 1 shows the encoder mounted to a servo motor, with its housing cut away to show the gears.

In practice, a center gear and a transfer gear with 23 teeth combine to drive plastic satellite gears with 24 and 25 teeth. In operation, as the motor turns, the satellite gears gradually fall farther and farther behind the drive gear. Thus, as the motor continues to spin, the gears go through a lengthy list of combinations of relative angles. This effect is shown in Fig. 2 for the first two rotations, an arbitrarily large number of rotations, and rotation number 599 where the cycle is nearing completion. If the gears have numbers of teeth that do not have common multiples, then the cycle is unique and does not repeat until 24×25 = 600 revolutions have occurred. Since this is more revolutions than ORCA requires to traverse any axis, any gear orientation corresponds to one unique revolution number and therefore one unique robot axis position. The gear phases are measured by shining light from LEDs through slits in the gears and onto optosensors.

In use, the signals from both the absolute and the incremental encoders are routed via a ribbon cable to each corresponding joint microcontroller, where a simple algorithm based on modulo arithmetic is used to convert the phase measurements into a revolution number. When each servo wakes up, its joint motor rotates one revolution and stops. This produces a slight motion, and in milliseconds the microcontroller knows the absolute position. HP is interested in exploring commercial applications for this or binary versions of this component and technology.

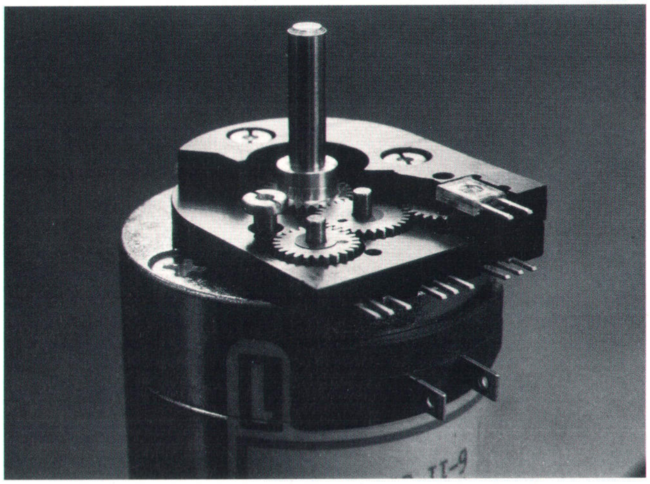
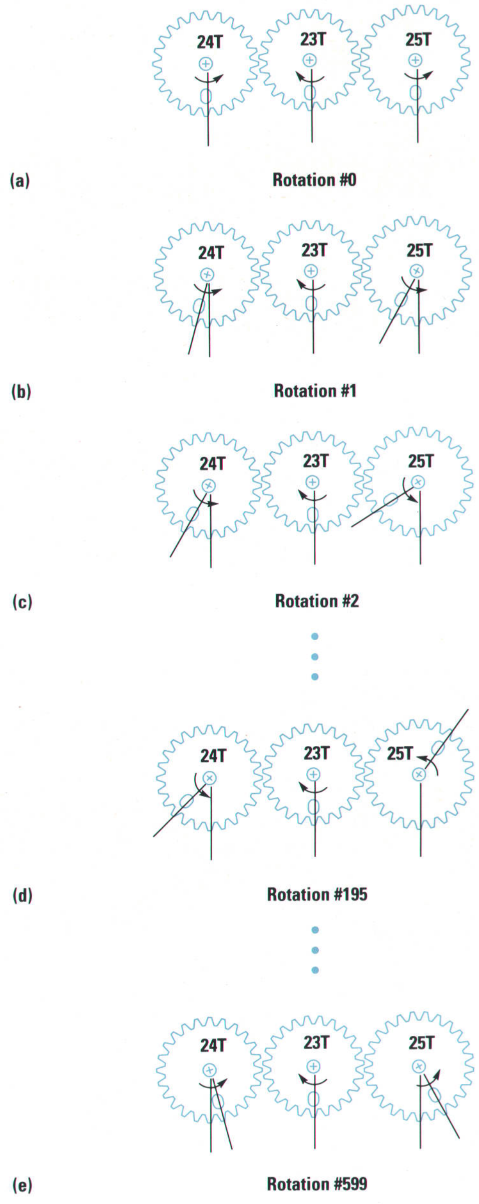


Fig. 1. Robot position encoder mounted on a servo motor.



(a) Rotation #0

(b) Rotation #1

(c) Rotation #2

(d) Rotation #195

(e) Rotation #599

Fig. 2. Absolute digital encoder operation.

the sequence of positions. In MDS, the program is called a procedure. Procedures are used as building blocks to connect robot actions with control of other devices and instruments into higher-level tasks. Procedures can call other procedures, much like a subroutine call in BASIC.

The concept of a motion was generalized to be an abstract execution object, which led us to consider other types of objects that could be provided to simplify robotics programming. These types include:

- Tool. Defines the endpoint of the robot arm.
- Frame. Defines a frame of reference for a motion.
- Motion. A sequence of robot positions.
- Rack. A rectilinear array of positions (similar to a pallet).
- Syrconfig. A configuration for a syringe pump dispenser.
- Procedure. The basic programming unit.
- Device. A configuration for an RS-232 or HP-IB device.

The metaphor used to create and store these objects is that of entries in a dictionary. Motions, racks, tools, frames, syrconfigs, devices, and procedures are all types of entries that can be created. An entry is an execution object, and a dictionary is a file that holds the entry objects. Users create and name entries, and save them in a dictionary. Each entry type has its own special editor, or form, for defining or teaching the entry. Entries can be used as commands (motions and racks), or as modifiers of entry commands (tools and frames).

For example, the following procedure statement will execute a motion PickUpDispenserNozzle using a tool offset defined by the tool NozzleGripper and referenced to a frame NozzleStand:

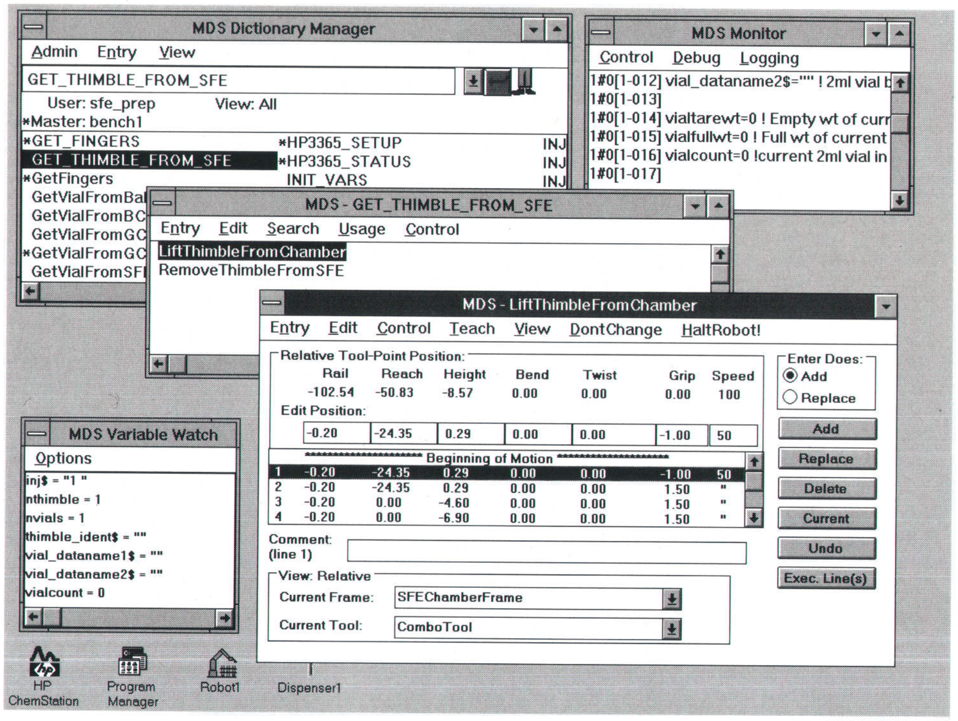
```
PickUpDispenserNozzle WITH NozzleGripper AT NozzleStand
```

The frame and tool that a motion uses can also be attached to the motion from within the motion editor, and may provide defaults for the motion to use when it is executed. The use of long (31-character) names and the command modifiers WITH and AT provide a very natural-language-like look to procedure statements for robot control and help the procedure code to be self-documenting. The use of longer names is simplified and encouraged by providing a variety of selection, copy, and paste features in the user interface, which reduces typing and programming errors that arise from typing mistakes.

MDS allows two dictionaries for editing and execution of entries: a user dictionary and a master dictionary. When an entry is referenced in a procedure statement, the user dictionary is searched first, and allows redefining entries that are in the master dictionary. Although developers are free to choose their own guidelines, the master dictionary is recommended for saving entries that are to be used across multiple applications and the user dictionary is generally application-specific.

The user interface for selecting entries to edit and for general browsing of the dictionaries is the MDS dictionary manager window (see Fig. 7). This window is the main user interface to MDS, and provides access to administration utilities, dictionary and entry manipulation, and selection of various views for the entry listings. Double clicking on an entry name presents an entry information dialog box, which in turn allows access to editing, execution, or printing of the entry. Keystroke accelerators are provided for quick access to common functions, such as editing and execution.

The dictionary manager also provides a command line, with history, for execution of commands. MDS supports the concept of a live command line, from which a user can execute anything at any time. The new execution preempts current execution. This feature is used most often to access or change the value of a variable quickly, and to execute procedures to correct problems when the application is paused.



GET_THIMBLE_FROM_SFE

User: sfe_prep View: All
*Master: bench1

```
*GET_FINGERS              *HP3365_SETUP        INJ
 GET_THIMBLE_FROM_SFE     *HP3365_STATUS       INJ
*GetFingers                INIT_VARS           INJ
 GetVialFromBal
 GetVialFromBC
 GetVialFromGC
*GetVialFromGC
 GetVialFromSFI
```

```
1#0[1-012] vial_dataname2$="" ! 2ml vial b
1#0[1-013]
1#0[1-014] vialtarewt=0 ! Empty wt of curr
1#0[1-015] vialfullwt=0 ! Full wt of current
1#0[1-016] vialcount=0 !current 2ml vial in
1#0[1-017]
```

```
LiftThimbleFromChamber
RemoveThimbleFromSFE
```

Relative Tool-Point Position:

	Rail	Reach	Height	Bend	Twist	Grip	Speed
	-102.54	-50.83	-8.57	0.00	0.00	0.00	100
Edit Position:	-0.20	-24.35	0.29	0.00	0.00	-1.00	50

Enter Does: ● Add ○ Replace

```
******************** Beginning of Motion ********************
1  -0.20  -24.35   0.29  0.00  0.00  -1.00  50
2  -0.20  -24.35   0.29  0.00  0.00   1.50  "
3  -0.20    0.00  -4.60  0.00  0.00   1.50  "
4  -0.20    0.00  -6.90  0.00  0.00   1.50  "
```

Comment: (line 1)

View: Relative
Current Frame: SFEChamberFrame
Current Tool: ComboTool

```
inj$ = "1 "
nthimble = 1
nvials = 1
thimble_ident$ = ""
vial_dataname1$ = ""
vial_dataname2$ = ""
vialcount = 0
```

Fig. 7. Methods Development Software (MDS) development environment.

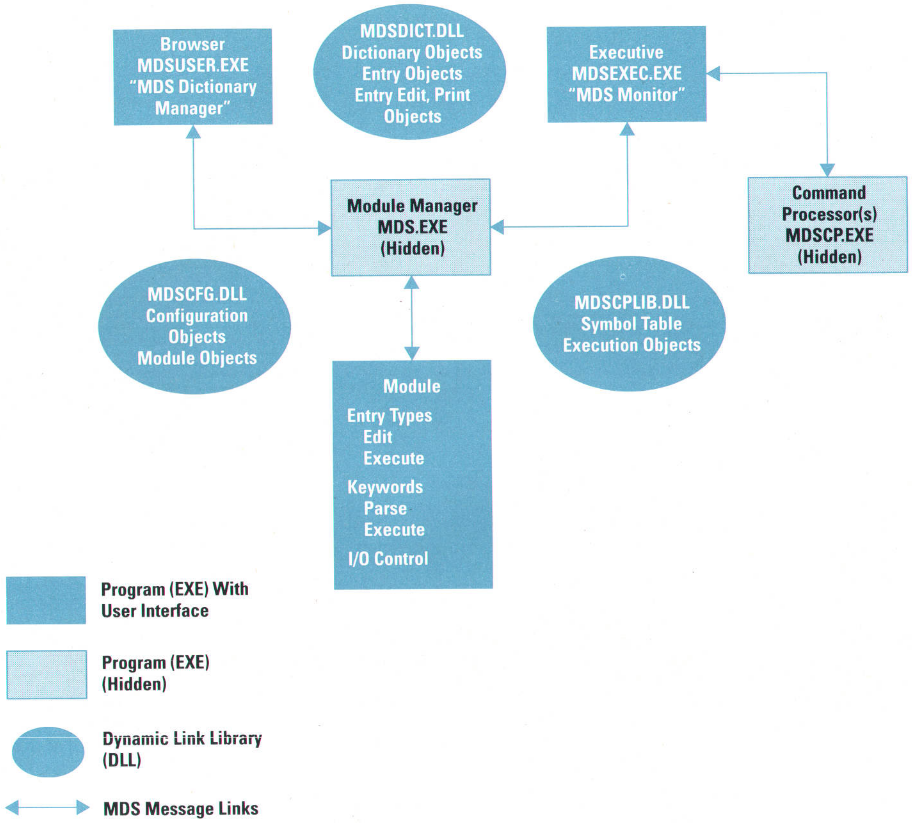


Fig. 8. MDS architecture.

Variables that are defined exist until they are explicitly removed. These features give MDS an interactive feel, and allow the creation and testing of an application in terms of small units.

The other main user interface window is the MDS monitor (see Fig. 7), which shows display output from procedure statements and provides execution control and debugging facilities. Debugging facilities include execution stepping, tracing, and logging display output and errors to log files. A variable-watch window, which can be used to monitor the values of variables as they change, is also provided via the MDS monitor menu.

The procedure editor is the other most commonly used element for developing an application. Each procedure is edited within its own window. Multiple edit sessions are possible, and text can be copied and pasted between them. The procedure editor also allows execution of the entire procedure or any highlighted text. This feature allows quick and simple testing of statements and procedures. The procedure editor also provides direct access to other types of entry editors, including other procedures. For example, the user need only double click with the mouse to highlight the name of a procedure, or other entry, and press **Ctrl+E** to access the editor for that entry. The procedure editor's features encourage the use of small procedures that can be easily tested to become building blocks for higher-level procedures and enhance the interactive feel of MDS.

MDS Architecture

In addition to supporting the features described in the previous section, MDS is designed for extensibility. Because of the wide-ranging nature of laboratory robotics, and because it is a developing field, the types of instruments and objects with which the robot must interface cannot be predefined in the software. Thus the software has to be both configurable and piecewise upgradable. The software also has to support multitasking of execution and simultaneous editing and programming during execution. These requirements suggest a modular design, with several programs that interact with a defined protocol. Fig. 8 shows the MDS architecture.

The design of MDS is based on a core system that can be enhanced by the addition of software modules. It is implemented as several Windows programs that communicate with a set of MDS messages, and a set of Windows dynamic link libraries that provide the basic "glue" for the architecture. In Windows, the use of dynamic link libraries allows sharing of common code and sharing of data between programs. MDS takes advantage of dynamic link libraries for both of these purposes. Since Windows itself is implemented as several dynamic link libraries, the various programs that make up MDS do not have to include code for the windowing interface. Run-time memory requirements are also minimized by taking advantage of Windows memory management facilities that allow code segments and data to be marked as load-on-call and discardable.

An important design rule for MDS was that no data structure definitions could be shared between the programs and dynamic link libraries that make up MDS. This rule allows MDS to be truly modularized so that parts of MDS can be modified without affecting or requiring changes to other parts. A direct benefit was that it enabled the core system to be developed in Palo Alto while the robot and dispenser modules were developed at the Avondale site, 3000 miles away.

Instead of data structures being shared, a set of data objects were defined and supported with calls to access their properties. These data objects are supported within dynamic link libraries, which provide the function call access, and which "own" the data and allow it to be shared. For example, the MDSDICT dynamic link library supports the dictionary and entry objects, the MDSCFG dynamic link library supports the configuration and module objects, and the MDSCPLIB dynamic link library supports objects used for execution. The use of the dynamic link library's local heap for allocating the objects compensates for the performance penalty of the overhead of the calls to access the object data. Handles to the objects are passed among the MDS programs using MDS messages, which specify an action to take with the object.

Certain objects and their corresponding calls and messages are considered "core-only" property. Modules can only access information in these objects using an intermediary object that can be properly shared with modules. For example, the entry and dictionary objects are core-only, so an entry edit block object is used to create and edit an entry object and is accessible by modules. Even in this case, though, not all of the object's properties—its corresponding entry and dictionary for example—are accessible by modules. These properties can only be set by a part of the core (the module manager or browser in this case).

MDS modules extend the functionality of MDS by providing support for new entry types and commands. Currently, there are three modules available: the MDS system module, the ORCA robot module, and a dispenser module that supports the HP 1243A syringe pump dispenser. The system module is different from the other modules in that it is an integral part of the MDS core, while the other modules can be optionally configured to run as part of MDS. Modules are responsible for the control of their respective hardware and entry editors, and for execution of the commands and functions that they register with MDS. Although the current modules all support hardware, modules can be written simply to add commands or other functionality to MDS, such as interfacing with another software package.

MDS Core. The MDS core system consists of the module manager MDS.EXE, the browser MDSUSER.EXE, and the executive MDSEXEC.EXE (see Fig. 8). The executive in turn supports the MDS command processor MDSCP.EXE as a separate program that it manages. Three dynamic link libraries, MDSCFG.DLL, MDSDICT.DLL, and MDSCPLIB.DLL, complete the MDS core.

The module manager is the main MDS program. Its window appears only momentarily to show booting information, and is then hidden. The module manager acts as the main gateway for all MDS messages. It is responsible for maintaining the configuration of MDS modules (via MDSCFG.DLL) and dictionaries and entries (via MDSDICT.DLL). When MDS boots, the module manager reads configuration information from the MDS.INI file, and executes the module programs that are to be activated. Modules dynamically register their entry type and keyword information with the module manager at boot time. The module manager also supports dialogs for modifying the configuration, and for creating and saving entries.

The browser is the main user interface for MDS. Its window title is MDS Dictionary Manager, because that is how it appears to function to the user. Internally, however, it is not the dictionary manager; it serves only as the user interface to the module manager, which is responsible for maintaining dictionaries. This distinction between how a user views MDS and how MDS is implemented internally was important for maintaining a consistent internal design. The browser window provides the command line and a listing of entries defined in the selected dictionaries. The browser also supports the server portion of Windows dynamic data exchange (DDE) for MDS.

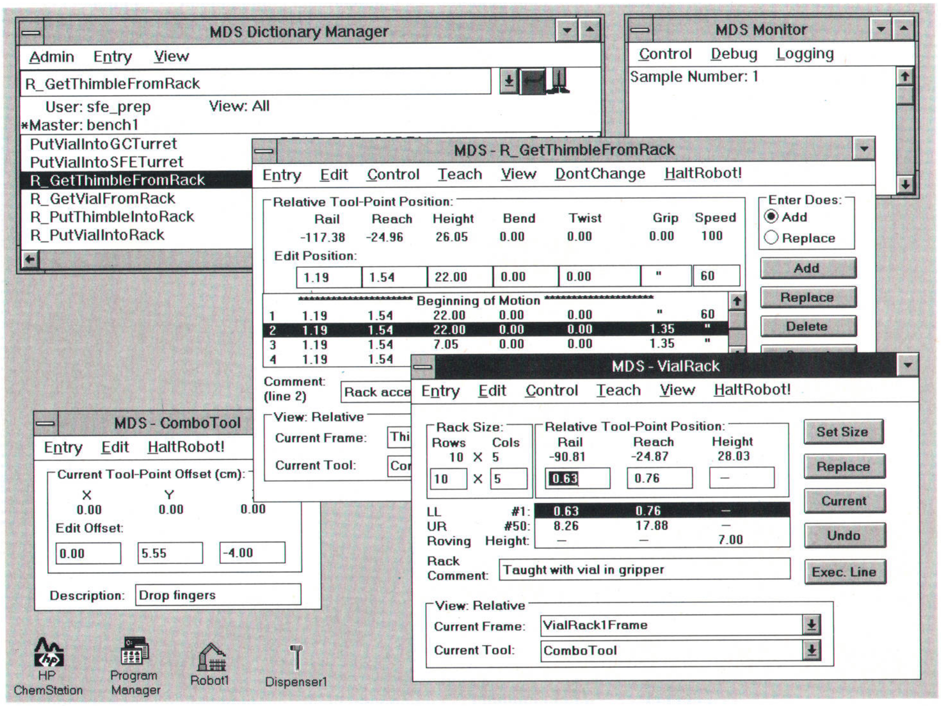
The executive provides the other window into MDS, the MDS Monitor window, which displays output from procedure PRINT statements. The executive also manages the MDS command processor and provides the user interface for execution control and debugging facilities.

The MDS command processor is responsible for procedure and text execution. All execution begins as text execution (from the command line, a procedure editor, or remote DDE execution), which is parsed and executed the same as a procedure. The syntax for the MDS procedure language is based on the HP ChemStation macro language (which is BASIC-like) with a number of enhancements.

Among the enhancements are support for the MDS entry concept and a PARALLEL command. The PARALLEL command allows procedures to be executed in parallel, sharing the same global symbol table and dictionaries. A set of commands for synchronization of parallel execution is also provided. This multitasking feature is used to increase the overall throughput of an application. For example, a procedure that tares a balance can be done in parallel with another procedure that uses the robot to get a beaker for weighing. When a PARALLEL command is executed, a new instance of the MDS command processor is run. Because Windows shares code segments for multiple instances of the same program, the command executes quickly, and the demands on memory are limited to the additional data segment for the new instance.

The MDS command processor parses and executes each line of a procedure at run time. An execution block object is used to pass execution information between the command processor and the modules during execution. Parameters to module commands are preevaluated for the module and passed on a shared stack object, whose handle is part of the execution block object.

An important part of automation is the ability to detect and recover from error conditions. MDS supports the ON ERROR statement, which specifies an error handling procedure to call. Through the use of a RESUME statement, the error handler can propagate the error back (RESUME EXIT), fix and retry the statement (RESUME RETRY), skip the statement (RESUME NEXT), or have the user decide (RESUME ALERT). The automatic error handling can be disabled, so that an error dialog box is always presented to allow the user to decide what action to take. The user can execute new procedures to fix the problem and return the application to a continuable state. This feature is particularly helpful during development of the application, and reduces the need to abort and restart an application when something goes wrong.



R_GetThimbleFromRack

User: sfe_prep View: All

*Master: bench1

- PutVialIntoGCTurret
- PutVialIntoSFETurret
- R_GetThimbleFromRack
- R_GetVialFromRack
- R_PutThimbleIntoRack
- R_PutVialIntoRack

Sample Number: 1

Relative Tool-Point Position:

Rail	Reach	Height	Bend	Twist	Grip	Speed
-117.38	-24.96	26.05	0.00	0.00	0.00	100

Edit Position: 1.19 1.54 22.00 0.00 0.00 " 60

Enter Does: ● Add ○ Replace

```
********************** Beginning of Motion **********************
1   1.19   1.54   22.00   0.00   0.00   "      60
2   1.19   1.54   22.00   0.00   0.00   1.35   "
3   1.19   1.54   7.05    0.00   0.00   1.35   "
4   1.19   1.54
```

Comment: (line 2) Rack acce

View: Relative

Current Frame: Thi

Current Tool: Coı

Rack Size: Rows Cols 10 × 5 [10] × [5]

Relative Tool-Point Position:

Rail	Reach	Height
-90.81	-24.87	28.03
0.63	0.76	—

		Rail	Reach	Height
LL	#1:	0.63	0.76	—
UR	#50:	8.26	17.88	—
Roving	Height:	—	—	7.00

Rack Comment: Taught with vial in gripper

View: Relative

Current Frame: VialRack1Frame

Current Tool: ComboTool

Current Tool-Point Offset (cm):

X	Y	
0.00	0.00	0.00

Edit Offset: 0.00 5.55 -4.00

Description: Drop fingers

Fig. 9. Robot motion, rack, and tool editors.

MDS System Module. The MDS system module provides support for procedures and devices. It is a "core-smart" module in that it uses certain calls and messages that are considered core-only. For this reason, it is usually thought of as part of the core system. Also, it is not a true module, in that it only provides for the creation and editing of procedures and devices. Procedure execution is handled by the MDS command processor, which is maintained by the MDS executive. Device entries are used in procedures, so their execution is also handled by the MDS command processor.

Dispenser Module. Liquid handling is important in many laboratory robotics applications. Solutions must be prepared, filtered, and extracted. The use of liquid dispensing in combination with a robot and a balance allows the gravimetric preparation of solutions, eliminating errors that often occur with the more traditional volumetric methods.

The dispenser module supports the syrconfig entry type, and control of the HP G1243A syringe pump dispenser. A syrconfig specifies which syringe to use, the syringe size, and the dispense speeds. An AutoFill feature allows the user to set levels at which a syringe will automatically refill. When enabled, this feature eliminates the need for the application to keep track of syringe levels, thus reducing procedure coding. The dispenser module also registers a set of commands that are used with the syrconfig entry to dispense and fill liquids. The dispenser module is implemented as a single program.

For example, the following statement will dispense 10 ml of liquid using the syringe specified within the syrconfig Buffer:

```
DISPENSE 10 ML Buffer
```

With AutoFill enabled within the Buffer syrconfig entry, the syringe will fill and empty until 10 ml are dispensed, using the volume setting in the entry as a guide. During the fill and empty cycles, a valve is automatically switched so that filling is done from a reservoir and emptying is done out through a nozzle.

Robot Module. The robot module supports the HP G1203A ORCA robot. The robot module provides the tool, frame, motion, and rack entry types, and a set of commands and functions for explicit control of the robot arm. The main robot window presents current robot position and status information, and provides access to calibration and entry edits (see Fig. 9). Multiple editors can be opened at any time, and positions can be copied and pasted.

All of the robot entry editors except for the tool editor are used interactively with the robot and its teach pendant. The user can also manually enter position information. For example, the motion editor presents a form for recording a sequence of positions. Pressing the remote teach pendant **Enter** key automatically records the current robot location as the next position in the motion. The motion editor allows setting force (grip) and torque (twist), as well as speed for individual steps in the motion. These parameters, along with a reference frame and tool offset, are called motion attributes. Another attribute, Don't Change, can be applied to individual axis values, and indicates that the axis value does not change for that position, no matter what value might have been taught. Once taught, positions in a motion can be rearranged or copied between motions. A motion that is used to pick up an object can easily be reversed and saved as a motion that replaces the object.

A rack is defined by teaching two corner locations and an access location, and by entering the number of rows and columns for the rack in the rack editor. Once defined, the desired rack location, or index, is specified as a parameter to the rack name in the command to move the robot to that rack access location. By teaching a rack with respect to a three-point frame, the rack can be accessed in virtually any orientation, with the robot bend and twist axes changing to reflect the new access angle. A hand-reference mode of teaching, by which the bend is locked at an angle perpendicular (or parallel) to the rack surface, greatly simplifies teaching access to tilted racks and centrifuges.

The robot coordinate system is presented as a Cartesian system with additional bend, twist, and grip axes. The robot module provides the frame and tool offset transformations for motion and rack positions. The conversion to joint values and the straight-line trajectories are all computed in the robot kinematics processor. When executing a motion, for example, the robot module applies the frame and tool offsets to each position in the motions, converting these values into absolute positions to send to the robot kinematics processor.

Use of Other Laboratory Equipment

A critical requirement for the MDS software is that it be able to support common laboratory devices through standard interfaces. MDS supports control of HP-IB (IEEE 488) and RS-232 (COM) interfaces using the device entry type and procedures written to use the devices. The device entry type provides a simple form for assigning the address, COM port parameters, and buffer sizes. The MDS procedure language supports using the device entry name in place of a file name in the BASIC-like OPEN statement. This allows the BASIC examples included with most manufacturers' instruments to be easily incorporated into MDS.

Dynamic Data Exchange (DDE). Early in the project we envisioned MDS as being the master controller of the robotics bench. As the project progressed, it rapidly became evident that MDS must also be capable of being controlled by other applications and must be able to exchange data with other applications. High on the list of other applications were the HP family of ChemStation products and software provided by other manufacturers for instruments that HP does not provide.

Windows dynamic data exchange (DDE) was chosen as the mechanism for the control and exchange of data. DDE allows Windows applications to control and pass data using a client/server model. MDS supports DDE in both client and server modes. The DDE client support is handled by a set of commands that allow developers to add DDE to their application at a very high level. MDS handles all of the low-level details of the protocol. The DDE server support is handled by the MDS browser and command processor, and allows remote execution of any block of text that follows MDS command syntax. All MDS variables are accessible via DDE, both for setting and requesting values. In addition, MDS variables can be put on advise, or "hot-linked," which means that the client application is notified whenever the variable's value is changed.

By supporting DDE, MDS is able to interact with a wide variety of software that runs under Windows. Features that MDS lacks, such as database management and report processing, can be provided using software designed for that purpose, using DDE as the connection with MDS. Another example is the use of HP ChemStation software with MDS. Using DDE, MDS is able to instruct the ChemStation to load and run methods for samples that the robot has prepared and placed in the chromatograph's injector. The use of DDE to integrate MDS with other Windows applications provides a new level of systems automation for the analytical laboratory.

Conclusion

The HP ORCA hardware and software provide a robotics system that is easily adapted to the needs and requirements of the analytical laboratory. The use of a gravity-sensing teach pendant, in conjunction with a graphical user interface, provides an intuitive and simple means for programming the robot. Supporting both client and server dynamic data exchange, the HP ORCA system can be fully integrated into the information flow as well as the sample flow of the analytical laboratory. Applications outside the analytical laboratory are also easily found (see "The HP ORCA System Outside the Analytical Laboratory," page 9).

Acknowledgments

The ORCA project involved an extraordinary amount of cooperation between groups at HP Laboratories and the automated chemical systems program at the HP Avondale Division (now at HP's Little Falls operation in Wilmington, Delaware), along with valuable input and support from HP Analytical Products Group management and the field engineering force. The authors enthusiastically acknowledge the numerous outstanding contributions made along the way by many individuals as the HP ORCA system emerged from a feasibility study and grew into a product. The HP Labs contributors included John Michnowicz (department manager), Miles Spellman (kinematics software), Jim Young (torso and product design), Stu Lerner (arm and finger mechanics), Carl Myerholtz (bus and joint servos), Andrew Stefanski (joint refinement and kinematics), Hans Neumann (arm and hand mechanisms), and Bob Widmayer (digital servos). Many valuable discussions were had with individuals who had worked on HP plotter technology.[1] The Avondale R&D team, led by Greg Murphy, included Eric Park (mechanical design), Gil Siegal (kinematics software), Phil Fuhrman (servo electronics), Dave Clouser (robot module), and Jeff Griffith (dispenser module). The manufacturing engineering team included Andrea Morris (hand), Dan Liszewski, and Bill Boyd. Marketing people who were key in shaping the HP ORCA system's feature set are Mark Shuman, Bill Berry, John Rollheiser, and John Poole. Lastly, special thanks to upper management for their support of the ORCA project.

Reference

1. W.D. Baron, et al, "Development of a High-Performance, Low-Mass, Low-Inertia Plotting Technology," *Hewlett-Packard Journal*, Vol. 32, no. 10, October 1981, pp. 3-9.

HP OpenODB: An Object-Oriented Database Management System for Commercial Applications

The functionality of object-oriented technology and basic relational database features such as access control, recovery, and a query language are provided in HP OpenODB.

by Rafiul Ahad and Tu-Ting Cheng

HP OpenODB is an advanced object-oriented database management system (ODBMS) that is designed to support complex commercial applications. Commercial applications require support for large numbers of concurrent users, many short transactions, security and authorization procedures, high availability of information access to other databases, and high integrity. HP OpenODB is a hybrid ODBMS that combines several years of research and development on a database object manager with a decade of investment in relational database technology. This powerful combination brings the two pieces that make up an object together in an ODBMS. It also enables a smooth evolution from, and coexistence with, a relational database management system (RDBMS).

In the current release, all of HP OpenODB's stored data is managed by ALLBASE/SQL which is HP's ANSI standard relational database and is tuned to be the fastest RDBMS on HP platforms. The HP OpenODB object model is implemented by an object manager which provides unlimited user-defined types of information. HP OpenODB is designed to port easily to other RDBMSs.

HP OpenODB is well-suited for applications with one or more of the following characteristics:

- Complex and varied data structures
- Various data formats
- Access to data stored in different systems
- Constantly changing environment
- Multimedia storage and manipulation
- Multiuser access to information.

This article will describe the features and the software architecture of OpenODB and the object model provided in OpenODB.

Product Features

OpenODB's object-oriented features help reduce development and maintenance costs by allowing the developer to model business problems more intuitively. These features can be divided into the following categories:

- Tools that allow developers to use object-oriented modeling techniques to build a database
- Programmatic capabilities that allow storing code in the database and interfacing to external functions that support access to external data and preexisting applications
- Run-time performance features such as late binding and schema modification
- Access control and data integrity features.

Object-Oriented Modeling

The features in this category allow users to use OpenODB objects, types, and functions to model the attributes, relationships, and behavior of things in the real world.

Object Identity. Each object stored in OpenODB has a system-provided unique handle called an object identifier (OID). OIDs reduce duplication of information and relieve the developer of the need to create unique keys to identify stored information in the database.

Complex Objects. Complex objects can be constructed from simpler objects. Complex objects relieve application code of the need to manage the relationships between simple objects.

Referential Integrity. Since OpenODB knows about the relationships between objects, it can manage referential integrity. With this feature, if objects referenced by other objects are deleted, the system removes all dependencies. The user can specify whether or not to cascade changes or just to delete the immediate dependency. For instance, if manager Al supervises employees John, Mary, and Joe, and employee Joe is deleted, function call Name(Manages(:Al)) will return just John and Mary. The result is a simplified database schema and simplified application code that can be developed more quickly since the user does not need to manage referential integrity explicitly.

User-Defined Data Types. Users can construct user-defined data types in OpenODB rather than having to do it in the application code.

Type Hierarchy. Types can be organized in a hierarchy. This hierarchy of types and related functions allows users to minimize the translation from a business model to an OpenODB schema. The hierarchy also enables a type to inherit functions defined on parents, eliminating duplication of functions.

Multiple Inheritance. Functions defined on a type can be inherited by one or more subtypes. By inheriting rather than redefining functions, developers can easily extend the functionality of an application by reusing existing functions.

Overloaded Functions. Several functions can have the same name with different implementations. In an application, all that is needed to do is to call a function (e.g., Salary). OpenODB will determine which code (salary for employee or salary for manager) to execute based upon the parameter passed at run time. As a result, application code is simplified since the logic for determining which function to execute is in OpenODB.

Dynamic Typing. An object's types can be dynamically changed without having to destroy and recreate the object. This is possible because an object can belong to more than one type.

Encapsulation. OpenODB supports the combination of data and user-defined functions. Since OpenODB only allows access to data through these functions, an application is protected from changes to the function implementation and the user has control over how to access information in OpenODB. Encapsulation allows modification of the function body without changing application code.

Programmatic Features

Procedural Language. OpenODB provides the language OSQL (Object-Oriented SQL), which is based on SQL. OSQL includes programming flow statements, including IF/THEN/ELSE, FOR, and WHILE. This procedural language allows OpenODB functions to be quite complex, simplifying application code. Also, application code can be moved into the database, allowing applications to share code and get all of the benefits of sharing data. The code is tightly coupled with an object type, and OpenODB manages the integrity of the code and its associated type. This is one of the features that distinguishes OpenODB from more mature database architectures in which all of the application code is located in the application (see Fig. 1).

External Functions. Using external functions, distributed data and code stored outside of OpenODB can be accessed, regardless of data format or location. This simplified view of an enterprise allows programmers to develop complex applications that integrate existing data and applications more easily. For instance, a programmer can develop an OpenODB application that accesses data stored in other databases (e.g., ALLBASE/SQL, TurboImage, or DB2) as well as data in flat files. OpenODB acts as an integrator so that an application just needs to know OSQL. OSQL statements can call functions that access data and encapsulate code stored outside of OpenODB.

Run-time Features

Late Binding. OpenODB supports functions that are resolved at run time. Late binding allows more flexibility in application development and gives the full power of overloaded functions as described above. Late binding also shields user applications from changes to functions since these changes can be made online and the new function definition resolved at run time.

Dynamic Schema Modification. New functions and types in OpenODB can be created at run time. Users can also change the implementation of functions without having to recompile applications.

Performance. To improve run-time performance, functions can be compiled ahead of time. Also, related functions can be stored close to each other (clustered) to optimize performance.

Access Control and Data Integrity

High Availability. OpenODB maximizes the availability of information by providing:

- Dual logging to ensure the integrity of the log file
- Database replication on other systems so that more users can effectively access the same information and applications can quickly switch over to another system in case of an unscheduled shutdown
- Automatic switch to a second log file if the original log file is damaged or becomes full
- Dynamic file expansion to expand the size of the OpenODB file system if it becomes full
- Online backup of the database, which backs up the database while it is being accessed.

Multiuser Concurrency Control. OpenODB is designed to support hundreds of users accessing the same information while guaranteeing the integrity of that information.

Access Methods on Stored Data. Indexes are automatically defined on object identifiers (OIDs) when types and functions are created. These indexes help provide quick access to

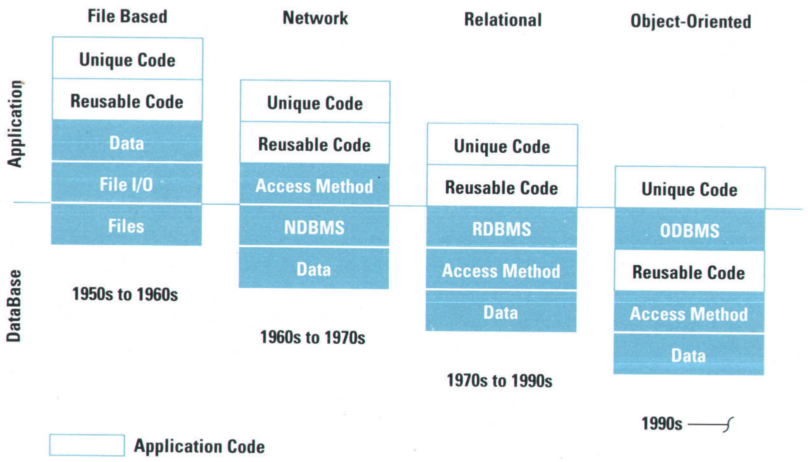


Fig. 1. With each new database architecture more and more major components have been moved from the application level to the database level. The years show approximately when the architecture was introduced and the peak years of use. By the way, all of these architectures are still in use.

information stored in the OpenODB database management system. Users can also define indexes.

Authorization. Access to OpenODB is controlled at the database and function levels and is based on authorization level (individual or group). Authorization statements provide a flexible way to control access to types and functions in OpenODB.

Persistent Data and Code. OpenODB allows the storage of data as well as code between application sessions.

Recovery. OpenODB uses the robust logging and recovery facilities of the ALLBASE RDBMS. In case of a failure, OpenODB can handle rollback or rollforward recovery to a particular time, using the log file to recreate saved work.

Transaction Management. OpenODB ensures the logical and physical integrity of the database by giving the user complete control over the unit of work to be performed within a single transaction. With this control, a transaction can be saved or rolled back (temporary work thrown away) to any point in the transaction.

Multimedia. OpenODB allows the storage and integration of large, unformatted data in binary format. Some examples include graphics, images, or voice data. Users can also define functions in OpenODB to manipulate this multimedia information. For example, the user can store a picture as well as the function to display the picture.

Product Structure

OpenODB uses a client/server architecture, enabling the user to use available computer power effectively. The OSQL interface and a graphical browser are located on the client side, and an object manager with a relational data storage engine and an external function interface are located on the server (see Fig. 2).

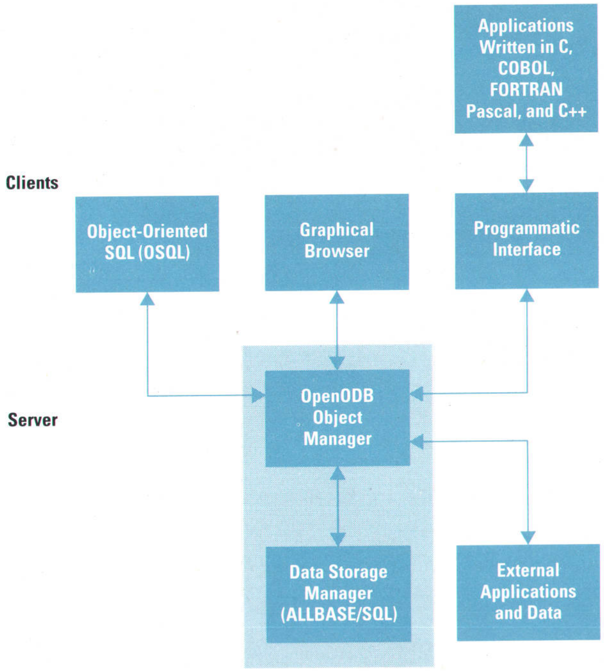

Applications Written in C, COBOL, FORTRAN Pascal, and C++
Clients
Object-Oriented SQL (OSQL)
Graphical Browser
Programmatic Interface
Server
OpenODB Object Manager
Data Storage Manager (ALLBASE/SQL)
External Applications and Data

Fig. 2. HP OpenODB client/server components.

The user can write OpenODB applications using any language that links with the C programming language including COBOL, FORTRAN, Pascal, Ada, and C++. Any fourth-generation language (4GL) or CASE tool that generates C code will interact with OpenODB. Application development can also be made easier by using tools that generate user interface code (e.g., OSF/Motif) in an X-Windows environment.

A software developer uses OSQL as an object definition, manipulation, and ad hoc query language that interfaces with the OpenODB server. OSQL can be used either interactively or from a program. We chose an evolutionary approach for developing OSQL by using ANSI standard SQL commands where possible and adding the full power of objects.

The graphical browser tool allows a developer to explore the structure (schema) of the database and its contents. The graphical browser is designed to increase the speed of application development by making it easier to reuse code stored in OpenODB.

The object manager supports all of the object-oriented features. This includes complex objects (objects that contain objects), dynamic schema modification, dynamic typing and multiple inheritance, encapsulation, late binding, object identity, overloaded functions, type hierarchy, and unlimited user-defined types.

The HP ALLBASE /SQL relational database provides the data storage and dynamic schema manipulation capabilities. These include authorization (security), declarative queries, high availability, multimedia support, multiuser concurrency support (e.g., graphics, images, voice), recovery, referential integrity, and transaction management. HP ALLBASE/SQL is transparent to the application developer and end user.

In keeping with an evolutionary approach, we recognized the need to access data and applications that already exist in a company's network. It is important to provide a developer access to this information regardless of which vendor's system it is stored on and regardless of its format. A gateway approach was considered but rejected because a significant amount of valuable data is stored in legacy (in-house) databases that are either custom-designed within a company or are spread across many different vendors' databases, and HP would never have the resources to build all the gateways needed to support commercial applications. Instead, we created an external function interface that acts like an exit subroutine in application code. Through this interface, database developers can access any code or data that resides within a company's network. Once external data is brought into the object manager, it can be integrated with data stored inside OpenODB and put into a format that is appropriate for the object-oriented application and the end user.

System Environment

The OpenODB server and all clients are available on HP-UX* 8.0 or later versions for the HP 9000 Series 700/800 systems, and on MPE XL 4.0 or later versions for the HP 3000 Series 900 systems.

OpenODB requires TCP/IP transport and ARPA Berkeley Services. The OpenODB graphical browser requires the X Window System.

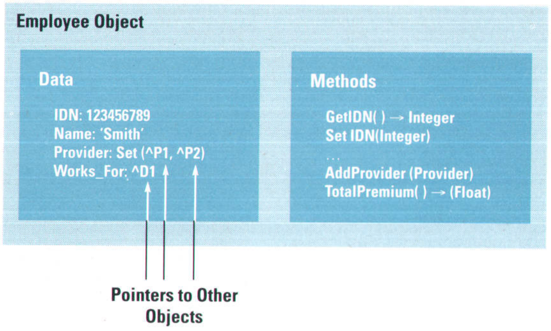


Fig. 3. The structure of a typical object. Notice that the data part contains pointers to the data (or other objects).

Object Models

There is no standard object model. The popular models in the programming language community are the C++ model[1] and the Smalltalk model.[2] These two models and others share many common features.

The commonly accepted definition of an object is something that contains data and code (called methods†) that operates on the data. This feature is known as encapsulation. External users retrieve or manipulate the data stored in objects by sending messages to the object. These messages are handled by the methods in the object.

From a database standpoint, the data stored in an object is the object's attributes or its relationship to other objects. The relationship is typically represented as a pointer to another object. The methods are the possible operations on the object.

For example, an employee object may have an employee identifier and name attributes (see Fig. 3). The object may also have information about the employee's health provider and who the employee works for. Notice in Fig. 3 that the relationship information is represented by pointers. Operations on data in an object may include setting or retrieving data values in the object, or doing some computation on the data. For example, in the employee object there is an operation to add a provider to the data and another operation for computing the total premium for the employee.

A data abstraction technique called classification abstraction[3] is commonly used in object models. In this technique objects of similar characteristics are modeled using classes. A class is used to define the structure of the data part of the object and the code that operates on the data structure. Each object that belongs to the class is called an instance of the class, and it has a copy of the data structure of the class and can be operated on by the methods specified for the class (see Fig. 4). For example, to define employee objects, we would define the class Employee. The class definition of Employee would state the data structure and the methods in some syntax. Once the class has been defined, any number of instances (employee objects) can be created by sending an appropriate message to the class.

Another abstraction technique called the generalization/specialization abstraction[4] is also used for object models. In this technique, a superclass may be created to model the common characteristics of two or more classes. Conversely, a class's characteristics may be refined by creating subclasses. For example, we may define a class Manager as a subclass of Employee. In this case every method and data structure definition applicable to the employee object is also applicable to the manager object. Fig. 5 shows that the classes Manager and Staff inherit the methods and data structures from class Employee and also have their own methods and data structures.

The OpenODB Model

The OpenODB model is based upon three concepts: objects, types, and functions. OpenODB objects are still modeled as data and methods but data is no longer automatically private to the object. Types are used to define functions, and functions are used to define the attributes, relationships, and operations on objects.

† The terms methods, functions, and operations refer to the code in an object.

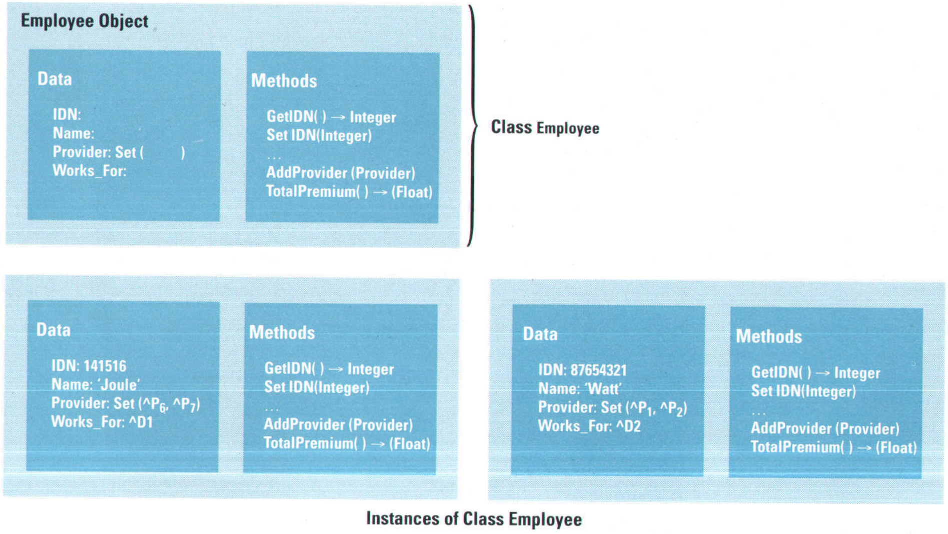


Fig. 4. An illustration of the data abstraction technique called classification abstraction. With this technique each object that belongs to a class is called an instance of that class. Although each instance contains data unique to that instance, the data structure is the same as that defined for the class the instance belongs to. Note that the same methods defined for the class are used for all instances of the particular class.

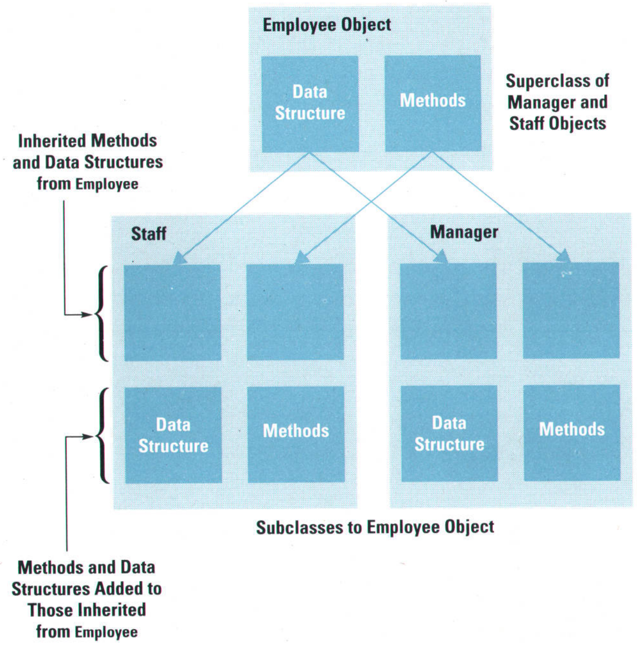

Employee Object
Data Structure
Methods
Superclass of Manager and Staff Objects
Inherited Methods and Data Structures from Employee
Staff
Manager
Data Structure
Methods
Data Structure
Methods
Subclasses to Employee Object
Methods and Data Structures Added to Those Inherited from Employee

Fig. 5. An illustration of the generalization/specialization data abstraction technique. Here two objects Manager and Staff inherit characteristics from the superclass Employee. They become subclasses when the methods and data structure inherited from Employee is augmented with new methods and data structures in each object.

Object

An object is a model (computer representation) of a thing or concept in the real world. Unlike data, which models the static aspects of the thing such as its attributes and relationships, an object also models the dynamic behavior in terms of the methods applicable to the object.

Conceptually, an OpenODB object is an encapsulation of data and operations. However, there are two important differences between OpenODB objects and those defined for object-oriented programming languages (OOPL). First, every data item in OpenODB is created as a function and is accessible and modifiable only through functions. This allows a uniform view of an object since there is no distinction between accessing an object attribute and invoking a method that returns values. For example, suppose the employee identification number data item in the employee object is defined as the function IDN with an integer return type. To access the employee identifier, we would evaluate the function IDN(e), which is equivalent to using a method in an OOPL model to access the employee's identification number in an employee object e. The difference is that OpenODB does not support strictly local data items as does the OOPL model, in which local data items are directly accessible only by the methods defined for the object. With OpenODB, since every data item is a function, any user who has the proper access privilege can evaluate the function and modify the data associated with the object.

It would seem that this model defeats the purpose of private data in the object-oriented methodology. However, OpenODB provides mechanisms for maintaining the concept of private data. Access control mechanisms provided in OpenODB are discussed later in this article.

The second difference between an OpenODB object model and an OOPL object model is the way in which an object's data is organized. In OOPL, the object's data is stored in some contiguous area of main memory and the object's reference is the address of the first byte of this memory area. For example, a reference to an employee object is an address in memory in which an instance of the employee data structure is stored. In OpenODB, the object's data may be stored (either clustered or dispersed) anywhere in main memory or on disk. The reference to the object is independent of the data associated with the object. This allows OpenODB objects to evolve gracefully without requiring the schema to be recompiled every time the object gains or loses data items.

OpenODB supports three kinds of objects: surrogate, literal, and aggregate (see Fig. 6).† Surrogate objects represent things or concepts in the application domain. The characteristics of the thing or concept that the surrogate represents are obtained by applying relevant functions to the surrogate object. In OpenODB, a surrogate object is identified by a unique object identifier (OID) that is totally separate from any data associated with the object. Examples of things that could be modeled as surrogate objects include persons, employees, parts, and manufacturing plants. Surrogate objects may also represent entities used by the OpenODB system to manage its own data such as the types and functions that keep track of OpenODB objects. Surrogates must be explicitly created and deleted either by the system (for system-defined objects) or by the user (for user-defined objects).

Literal objects are self-identifying in that they have external (printable) representations that correspond to well-known concepts. For example the number 123 and the string 'abc' are literal objects.

An aggregate object is a set, bag, list, or tuple of other objects. Table I lists the characteristics and some examples of aggregate objects.

The difference between a list object and a tuple object is that list objects and tuple objects have different constraints on the object types that make up their collections. These differences are described in more detail later. Sets are subtypes of bags.

In OOPL aggregate objects are supported by built-in constructs such as arrays, structures, and a library of predefined classes of aggregate objects.

† Note that literal and surrogate objects and types are sometimes collectively referred to as *atomic* types.

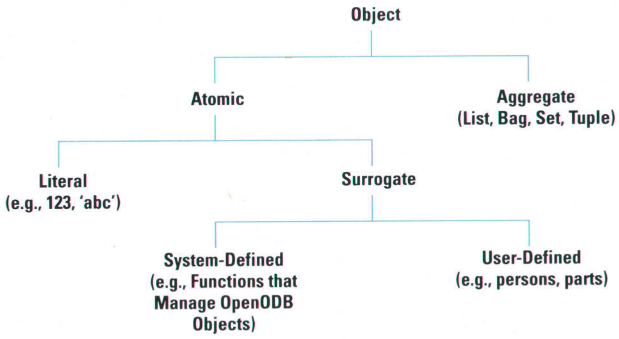

Object
Atomic
Aggregate (List, Bag, Set, Tuple)
Literal (e.g., 123, 'abc')
Surrogate
System-Defined (e.g., Functions that Manage OpenODB Objects)
User-Defined (e.g., persons, parts)

Fig. 6. The taxonomy of OpenODB objects.

Table I
Aggregate Objects

Type	Characteristic	Example
Set	Contains no duplicate elements	Set(123,‘abc’)
Bag	May contain duplicate elements	Bag(123,123,‘abc’)
List	Ordered collection of objects that may contain duplicates	List(‘ab’,‘ba’,‘ab’)
Tuple	Ordered, fixed-sized collection of objects	Tuple(1,‘abc’,2)

Type

A type is an object that implements the classification abstraction technique described earlier. OpenODB types are similar to OOPL classes. However, there are two major differences between an OpenODB type and an OOPL class.

First, an OOPL class must have a declaration of the data structure for the data part of its object instance (if there is any data part for a particular object). Since an OpenODB object is independent of its data part, OpenODB types do not have any data structure declaration. The data for an OpenODB object is defined by means of functions on the type, and such functions may be defined at any time, not only when the type is created. This difference can have a profound effect on the evolution of the application. Consider the following fragment of a data schema (written in hypothetical syntax for OOPL) for an application. Assume that the classes or types Company and Department have already been defined.

OOPL Declaration

```
Class Employee:
  Data
    IDN integer;
    Name char(20);
    Providers Set(Company);
    WorksIn Department;
  Methods
    ...
```

OpenODB Declaration†

```
Create type Employee;
Create function IDN(Employee)
  -> integer unique as stored;
Create function Name(Employee)
  -> char(20) as stored;
Create function Providers(Employee)
  -> SetType(Company) as stored;
Create Function WorksIn(Employee)
  -> Department as stored;
```

The OpenODB declaration could also be written as:

```
Create Type Employee Functions (
IDN integer unique,
Name char(20),
Providers SetType(Company),
WorksIn Department)
as stored;
```

Assume that sometime after the application has been in use, we decide to include date-of-birth information for employees. With OOPL, we would have to declare the date-of-birth field in the data structure for the class Employee, define relevant methods for it, and then recompile the entire application.

With OpenODB we simply define a new function DOB (date-of-birth) on object Employee to return a date (which is a predefined type). Furthermore, the definition of a DOB function can happen even while the application is running.

The second difference between an OpenODB type and an OOPL class is that an OpenODB type has an extension, which is a set of instances of that type. This feature facilitates queries over classes of objects. For example, in OpenODB, if we want to obtain the name of an employee whose IDN is 123456789, we simply pose a query as follows:

```
Select name(e) for each Employee e where IDN(e) = 123456789;
```

OpenODB is able to find the correct employee because for this example it keeps track of all the objects of type Employee.

The taxonomy of types is topologically identical to the taxonomy of objects because types are also divided into three groups: literal, surrogate, and aggregate. Literal types include integers and characters with extensions that are predefined and infinite. An extension of a type is the set of instances that belong to the type. For example, 1, 2, 3... are predefined extensions of the literal type integer.

Surrogate types include system and user-defined types. When a surrogate type is first created, the extension is empty. However, the extension for a surrogate type expands or contracts as objects are explicitly added to or deleted from the type. For example, the function Create Person would create a Person object and store it in the extension of the type Person.

The aggregate types supported are bag, set, tuple, and list types. They are referred to by type expressions consisting of constructors BagType, SetType, TupleType, and ListType respectively. Their extensions are automatically maintained and the user cannot explicitly insert objects or delete objects from extensions of aggregate types. For example, the type SetType(Person) refers to a type whose instances are sets of persons. If p is a person, then the object Set(p) is an instance of the type SetType(Person).

The difference between a list type and a tuple type is that a list type permits the specification of only one type for its members while the tuple type permits the specification of multiple types for its members. For example, an instance of the type ListType(Person) is an ordered collection of any number of objects of type Person or its subtypes. On the other hand an instance of the type TupleType(Person, integer, char) is an ordered collection of three objects, the first is of type Person, the second is of type integer, and the third is of type char.

OpenODB supports subtype/supertype relationships among types. The subtype/supertype relationship implements the specialization and generalization abstractions described earlier. This relationship induces a directed acyclic graph on the types known as the OpenODB type hierarchy (see Fig. 7). The type hierarchy is rooted in the type Object. If an object is an instance of a type, it is also an instance of all the type's supertypes. Thus the functions defined on the supertypes can be evaluated with instances of subtypes as arguments. Conceptually this is the same as subclass inheritance in an OOPL. For example, we can define the type Manager as a

† The OpenODB program fragments used here and in the rest of the document are based on OSQL (object-oriented SQL), which is OpenODB's database programming language.

subtype of the type Employee. Since by definition members of subtypes are also members of supertypes, every manager is an employee. Therefore, all the functions defined on Employee can be applied to any manager object. We can define additional functions on Manager. For example, the function Supervises(m) which takes a manager as an argument and returns the set of employees supervised by the manager, can be defined on type Manager. In general, this function cannot be evaluated with an employee object as an argument because not all employees are managers and the function is not defined for those who are not managers. Finally, since OpenODB allows a given type to have multiple supertypes, the inheritance is actually multiple inheritance (see TeachingAssistant in Fig. 7).

Functions

A function is an object that represents an attribute, a relationship, or an operation on an object. A function maps objects from one type to objects in another (not necessarily distinct) type. A function has one argument type, one result type, a result constraint, and an extension consisting of a set of tuples.

The result constraint can be specified for functions stored in the database (stored functions) only. If the stored function's result type is an atomic type or a tuple type of atomic types, the result constraint can be specified as unique or nonunique (default). A unique constraint states that the function will never produce the same results for two different arguments. For example, the function IDN will never return the same integer for two different employees. More important, a unique constraint will not permit setting the same number as the result of two different arguments. For example, two employees cannot be updated with the same IDN, otherwise an error will occur. For a function whose result type is SetType, the result constraint can be specified as disjoint or nondisjoint (default). The disjoint constraint states that the results of two different arguments will never overlap. For example, the function Worksfor(manager) –> SetType(Employee) returns a set of employees that work for a particular manager. Thus, Worksfor(john) returns {mary, jack, jill}, and Worksfor(henry) returns {cathy, abe, tom}. Each call results in a distinct set of names being returned. If the result constraint is nondisjoint and jack works for john and henry, then jack would appear in both sets.

Table II summarizes the types allowed to be assigned to function parameters and the relationships between the instances of atomic types and a tuple of atomic types that are modeled by the different parameter settings.

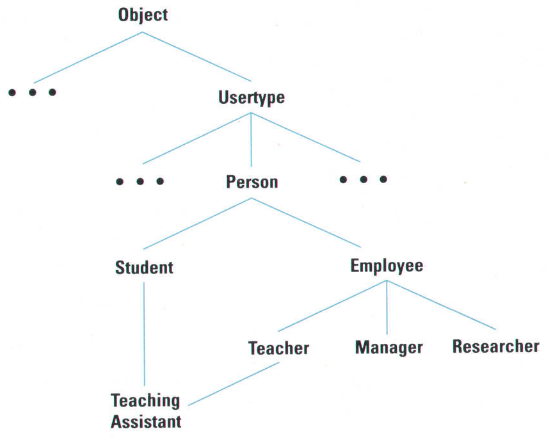


Fig. 7. An example of OpenODB subtype and supertype relationships.

Table II
Function Parameters and Their Relationships

Argument Type	Result Type	Result Constraint	Relationship Modeled
A	R	Unique	One-to-one (e.g., IDN to employee)
A	R	Nonunique	Many-to-one (e.g., employee to manager)
A	Set of R	Disjoint	One-to-many (e.g., manager to employees)
A	Set, Bag, or List of R	Nondisjoint	Many-to-many (e.g., skills to employees)

A = R = atomic types or tuple of atomic types
Atomic = literal or surrogate type

The specification of the function name, its argument and result types, and the result constraint constitutes the declaration of a function.† Like types, functions also have extensions that represent the attributes of and relationships between objects. For example, consider the function Name which is defined on type Person. This function returns a string of characters whenever it is called with the appropriate parameter. A set of object identifier and string pairs makes up the extension of the Name function. For example:

```
{<OID1, John> <OID2, Harry> <OID3, Helen> ...}
```

where OIDn represents the unique object identifiers and the names represent the strings of type Person. The declaration and implementation of a function may take place at different times in the database schema creation. The implementation of the function can be changed without recompiling the applications (although the database functions may have to be recompiled).

A function can be implemented as a stored function, a derived function, a computed function, or an external function. Derived and computed functions are collectively known as OSQL-based functions.

Stored Function. In this case the extension of a function is stored in an SQL table. For example, the functions IDN, Name, Providers, and WorksIn are stored functions. Their result values can be assigned and modified by the user. The following script creates an employee and assigns an employee identifier and a name.

```
Create Employee e;
IDN(e) := 123456789;
Name(e) := 'Smith';
```

† The argument type or the result type could be VOID, in which case the function has no argument or no result respectively. Functions with no results are typically used to model database update operations.

Derived Function. In this case the extension is described by a query using other functions. This is analogous to views in the relational data model. For example, assume we have an overloaded function called Name and we want to create a function EmployeeInfo that returns the identifier, employee name, and department name for a given employee. We could use the following script:

```
Create function EmployeeInfo(Employee e)
  –> TupleType(Integer,Char,Char) as osql
    select single IDN(e),Name(e),Name(WorksIn(e));
```

In this example, EmployeeInfo is not stored, but derived from the functions IDN, Name on Employee, and Name on Department and WorksIn. Unlike stored functions, not all derived functions are updatable.

Computed Function. In this case the extension is described by a program written in OpenODB's database programming language OSQL. For example, if we want to convert all managers who have less than a specified number of employees to programmers (assume that Programmer is a subtype of Employee), we would write:

```
Create function MgrToEng (Integer minemps) –> Boolean
  as osql
  begin
    declare SetType(Manager) mgrset;
    declare Manager m;
    /* get all the managers in mgrset */
    mgrset := select distinct atomic m for each Manager m;
    for m in mgrset do
      if (count(Supervises(m)) < minemps)
      then
    /* convert a manager to a programmer */
        begin
          add type Programmer to m;
          remove type Manager from m;
        end
      endif;
  end;
```

External Function. For an external function OpenODB uses code outside of itself to obtain the extension of the function. The user specifies how to invoke this code. External code can be specified to be invoked in the following three ways:

- Specify the implementation to be SQL and provide an SQL statement to be executed to obtain the extension. This is used to access SQL databases. For example, assume that there is a table EMPHISTORY(IDN, DEPT, JOBDESC) in an Admin database that describes the employment history of employees. We could create a function in OpenODB to access this information as follows:

```
Create function EmpHistory(Employee e)
  –> bagtype(tupletype(Char,Char)) as external
  SQL(‘Admin’,‘JSmith’,‘htimSJ’,
    ‘Select DEPT,JOBDESC
    from EMPHISTORY
    where IDN = :x’,List(SSN(e)));
```

Here the first argument to SQL is the database name. It could be specified as NULL in which case the current database would be used. To access remote SQL databases, ALLBASE/Net must be installed, and the database must be registered in AliasDB.[5] The second and third arguments are the user name and the user password respectively. If the second argument is specified as NULL, the third must also be NULL, and in this case the default user is used to access the SQL table. The fourth argument is the SQL statement to be executed. It may contain references to host variables such as :x. The fifth parameter is a list of values to be substituted for the host variables. In this example, IDN(e) will be evaluated and the resulting string will be substituted for :x. A connection will be made to the current database (if one does not exist) using the user name and the password. Then the query will be executed. The result will be converted to OpenODB format and returned to the caller.

- Specify the implementation to be OsCommand (an OSQL function tied to operating system commands) and provide the operating system with the command to be executed to obtain the extension of the function. For example, we could define a function Dir to look at a listing of the current directory in HP-UX with the following OpenODB function.

```
Create function Dir(Char d) –> bagtype(char) as
  external OsCommand (‘ls $d’);
```

- Specify the implementation to be GeneralExtFun or SimpleExtFun and provide the names of three routines (for GeneralExtFun) or one routine (for SimpleExtFun) that are linked with the user application. For GeneralExtFun the first routine is used to open a scan on the result of the function for a given argument. The second routine is used to read the result objects. The third routine is used to close the scan. SimpleExtFun can only be used for functions whose result type is atomic (literal or surrogate) or tuple type of atomic. When called, the specified routine must return the result for a given argument.

To show how SimpleExtFun is used to implement an external function suppose we have a C program called CreditRating that computes the credit rating (an integer number) of the person with the given identifier (assume a 9-character string). We could make the C program the body of an OpenODB function called CrRating as follows:

```
Create function CrRating(char p) –> char as external
  SimpleExtFun(‘CreditRating’);
```

SimpleExtFun is appropriate here since the C program will return a single integer number for each argument.

Let us now consider an example that uses the general external function to implement an external function. Suppose we have an OpenODB function called JobHist that returns the job history of a person as a array of character strings. We could define an OpenODB function that uses the following C code:

```
Create function JobHist(char p) –> BagType(char) as external
  GeneralExtFun(‘JobHistOpen’,‘JobHistNext’,‘JobHistClose’);
```

Here, JobHistOpen, JobHistNext, and JobHistClose are three routines written in C. When JobHistOpen is called with a person's identifier, it should create a scan data structure and return a pointer to the data structure as an integer number. One possible way to implement JobHistOpen is to have the routine allocate storage for the returned array. JobHistOpen can then return a pointer to a structure that contains an integer number and a pointer to the array. The integer number (initialized to –1) keeps track of the the next element of the array to be returned by JobHistNext. Each time JobHistNext is called it is given the pointer obtained from JobHistOpen. JobHistNext increases the integer number of the structure, accesses the array element at that location and returns the string. When JobHistClose is called, it is given the pointer obtained from JobHistOpen. JobHistClose frees the storage that was allocated by JobHistOpen.

The extensions of all stored functions and some derived functions can be changed by authorized users. Such a change is accomplished by update functions associated with the original function. These update functions are automatically generated by the system for updatable functions. For example, for the stored function Name on Employee, the system might create an update function called Updname, which takes the OID for an employee object and a string object and creates an association between the two. Thus, after using the Updname function with an Employee object that has the identifier OID123 and a string labeled 'Smith' as parameters, references to OID123 will be associated with 'Smith' and vice versa.

Function Name Overloading

OpenODB supports function name overloading. Many functions can have the same name but different argument types. For example, in the functions Weight(Part) and Weight(Unit), the function Weight(Part) returns the weight of an individual part and the function Weight(Unit) returns the weight of all the parts on a particular unit.

The functions that are explicitly created by the user are known as *specific* functions. For a set of specific functions with the same name, OpenODB creates a function called the *generic* function, whose name is the same as those in the set. However, the generic function has no implementation. The name of a function by itself refers to the generic function and is called a generic function reference. However, if the name is followed by a type reference separated by a period, then it refers to a specific function, and such a reference is called a specific function reference. For example, if function Weight is defined on type Part, the generic function reference is Weight and the specific function reference is Weight.Part.

Since the generic function has no implementation, it cannot be directly evaluated. It must first be bound to a specific function. OpenODB supports late binding of a generic function to a specific function. If a generic function is used in an evaluation, then the specific function to be used is determined based on the actual argument at run time. However, if a specific function is used in an evaluation, then that specific function is used regardless of the argument (the argument must be an instance of the argument type). Thus specific function reference supports early binding.

For example, assume that we have the generic function Salary on Employee that returns the salary of a given employee. Suppose we want a manager's salary to include the bonus. We could overload Salary on Manager to return the sum of the manager's salary and bonus as the following example illustrates.

```
Create function Salary(Employee e) –> Float as stored;
Create function Salary(Manager m) –> Float as osql
  salary.employee(m) + bonus(m);
```

If e is an employee but not a manager, then Salary(e) will return the salary of employee e. However, if e is also a manager, Salary(e) will return the sum of the salary of e as an employee plus the bonus of e as a manager.

In some cases, mapping from a generic function to a specific function is ambiguous. This happens because objects may belong to multiple types and there may be more than one specific function defined on those types. OpenODB does not prescribe default semantics for choosing a specific function in case of ambiguities. Instead it reports an error.

Access Control in HP OpenODB

One of the functions of a database management system (DBMS) is access control. A DBMS must prevent unauthorized access to data stored in the database. Although commercially available DBMSs do have security subsystems that support access control, support for authorization in object-oriented and functional database systems has not yet been fully addressed. In such systems, authorization features must interact with advanced model and language features such as user-defined operations, encapsulation, multiple inheritance, generic operations, and late binding.

Object-oriented languages inherently provide some form of access control in the way abstract data types encapsulate private state information.[6] Although abstract data types provide support for access control at the implementation level, there still remains the need to protect data at the organizational level.

The authorization model in OpenODB is based on the notion of controlling function evaluation. Only a single privilege, that is, the privilege to call functions, is necessary to support an authorization model that is more powerful and has finer levels of access granularity than the traditional authorization model of relational databases.[7]

The Authorization Model of Relational Systems

The basic authorization model of standard SQL can be characterized by a matrix captured by a relation schema (S,O,M). S is the domain of all subjects that access the database system (users, processes), O is the domain of objects (base relations and views), and M is the domain of access operations (SELECT, INSERT, DELETE, UPDATE, and ALTER).[8]

Most RDBMS implementations support the notion of ownership. Each database schema has a designated system administrator, database administrator, or database creator who is the owner of all objects created in that schema. Furthermore, the creator of an object becomes the owner of the object and typically holds all privileges on that object.

An object owner can grant access to the object to other users. Commercial systems implement mechanisms that allow the owner to grant SELECT, INSERT, DELETE, UPDATE, and ALTER privileges on objects to other users selectively. Furthermore, an object owner can permit a GRANT privilege on the object, thereby enabling other users to further grant privileges on the object.

Authorization in OpenODB

Many of the concepts developed for relational database authorization are fundamental and are applicable to other technologies such as object-oriented database management systems. These fundamental concepts include the notion of ownership, users and groups, privileges, and granting and revoking privileges. The authorization model of OpenODB uses many of these fundamental concepts.

Users, Groups, and Owners. Users are modeled by the OpenODB type User, which is characterized by a set of functions that create and destroy user objects, return valid user names, and match passwords. For example the strings:

```
Create User 'Smith' Password 'x012' in tester;
Create User 'Jones' Password 'y2' in engineer;
```

create the users Smith and Jones with their respective passwords and assign Smith to the group tester and Jones to the group engineer.

Users can be classified by groups. Privileges granted to a group apply to each user in the group. Typically, users are classified based on their roles.[9,10,11,12] A user can belong to multiple groups thereby accumulating the privileges from each individual group. Furthermore, groups may be nested. A nested group inherits the privileges of the nesting groups similar to the way functions are inherited by a subtype from a supertype.

A group is an object in OpenODB that is modeled by the type Group. A group has attributes such as name and members and is made up of either individual users or other groups. Typical functions defined on the type Group include functions for creating and deleting groups, for adding and removing subjects, for returning group members, and for granting and revoking call privileges. The following statements create the groups specified for the users Smith and Jones in the example above.

```
Create group developer;
Create group engineer subgroup of developer;
Create group tester subgroup of developer;
```

The access-control hierarchy created for these functions is shown in Fig. 8.

The user who creates a given function is said to be the owner of the function. The owner of a function automatically has a call privilege on the function. Furthermore, the owner can grant call privileges to other users using the Grant function which is described in the next section.

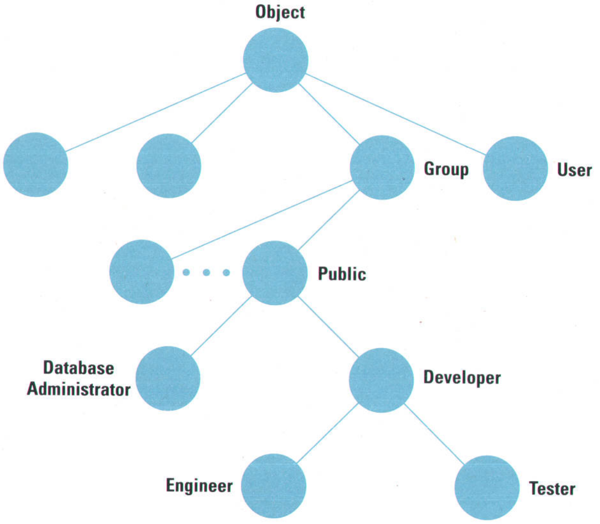


Fig. 8. An example of an access-control hierarchy in OpenODB.

The group hierarchy is rooted in the group Public which has call authority on common system functions such as Connect and Select. Every authenticated user by default belongs in Public.

Database administrators (DBAs) are special users with more privileges than ordinary users. For example, DBAs have all the privileges implied by function ownership.

Granting and Revoking Privileges. Call privileges can be granted and revoked on a per-group basis. Privileges can be granted unconditionally using the Grant statement or conditionally using the Grant statement with an IF option. With unconditional privilege granting, an authorized group member can call a function for all possible argument values. For example, if the user Jones from the example above wants to grant a call privilege on the salary of employees to the group tester, the OSQL statement would be:

```
Grant call on salary to tester;
```

For granting a conditional privilege, a predicate (Boolean-valued function) is associated with the group and the function. The predicate has the same argument type as the function. When the function is called, the argument is used to evaluate the predicate. If the predicate returns true, the user can call the function with that argument. For example, suppose user Jones wants to grant the privilege to tester to be able to modify the salary for part-time employees. First the function to filter part-time employees is created:

```
Create function SalaryGuard(employee e, float sal) ->
  Boolean as OSQL
begin
  if (status(e) = 'part-time') return TRUE;
  return FALSE;
end;
```

To grant the privilege to tester:

```
:f := funassign(function salary.employee);
Grant call on :f to tester if SalaryGuard;
```

The OpenODB access control model is based on the single concept of controlling function invocation by allowing only authorized groups access to a function. The OpenODB access control mechanism does not impact the performance of the owner of the function, the database administrator, or other common OpenODB services.

Acknowledgments

Many people contributed to the design and implementation of OpenODB. We would like to acknowledge the contributions of managers and researchers of the database technology department of HP Laboratories, and the development, support, marketing, learning products, and management personnel of the database laboratory of HP's Commercial Systems Division.

References

1. B. Stroustrup, *The C++ Programming Language*, Addison Wesley Publishing Company, 1987.

2. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison Wesley Publishing Company, 1980.

3. R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company.

4. J. M. Smith and D.C. Smith, "Database Abstractions: Aggregation and Generalization," *ACM Transactions on Database Systems*, Vol. 2, no. 2, June 1977.

5. *ALLBASE/Net User's Guide*, Hewlett-Packard Company, Part Number 36216-90031.

6. J. H. Morris, "Protection in Programming Languages," *Communications of the ACM*, Vol. 16 no. 1, January 1973.

7. P. P. Griffiths and B. W. Wade, "An Authorization Mechanism for Relational Database Systems," *ACM Transactions on Database Systems*, Vol. 1, no. 3, September 1976.

8. C. J. Date, *A Guide to the SQL Standard*, Addison-Wesley Publishing Company, 1987.

9. E. B. Fernandez, et al, "An Authorization Model for a Shared Database," *Proceedings of the ACM SIGMOD International Conference*, 1975.

10. R. Gigliardi, et al, *A Flexible and Efficient Database Authorization Facility*, IBM Research Report 6826 (65360), November 1989.

11. J. A. Larson, "Granting and Revoking Discretionary Authority," *Information Systems Journal*, Vol. 8, no. 4, 1983.

12. R. Ahad, et al, "Supporting Access Control in an Object-Oriented Database Language," *Proceedings of the International Conference on Extending Database Technology*, 1992.

The HP Ultra VGA Graphics Board

By increasing the display memory to 1M byte and providing some local graphics processing, the HP Ultra VGA board is able to increase VGA resolution to 1024 by 768 pixels with 256 colors at all resolutions.

by Myron R. Tuttle, Kenneth M. Wilson, Samuel H. Chau, and Yong Deng

The HP D2325A Ultra VGA board, which represents the latest in the evolution of HP personal computer video systems, is a video accessory card for the HP Vectra line of personal computers. This board offers exceptional video performance for graphics-intensive applications such as Microsoft® Windows and AutoCAD™. It enhances overall system performance by using hardware accelerators to relieve the CPU of common video processing functions. For high-resolution and flicker-free operation, the Ultra VGA board offers display resolutions up to 1024 by 768 noninterlaced and refresh rates up to 72 Hz. Finally, the board is upgradable to 1M bytes of video memory to give 256 colors in 800 by 600 and 1024 by 768 resolutions.

In this article we present a brief history of the evolution of PC video systems. We will then discuss the benefits of adding acceleration to video hardware and the hardware and software partitioning trade-offs that must be made. Finally, the implementation of the HP Ultra VGA board is described, both as a plug-in accessory and as an embedded feature as it is in the HP Vectra 486/U family of personal computers.

Evolution of PC Video

1981 marked a dramatic change in the world of computing because that was the year the IBM personal computer was introduced. The first IBM PC came equipped with 64K bytes of RAM and an alphanumeric Monochrome Display Adapter (MDA). The MDA contained 4K bytes of memory, enough to store one screen (25 lines of 80 characters each) of alphanumeric information. The PC with one of these adapters functioned like most terminals available at the time. It had very clean alphanumerics but lacked any graphical capabilities. Until the introduction of the MDA, virtually all PCs or "home computers" such as the Apple II, Commodore PET, and the Tandy Radio Shack TRS-80 used a television monitor or a modified television monitor as a display, grossly limiting the resolution.

Also available from IBM in 1981 was the Color Graphics Adapter (CGA). This adapter contained 16K bytes of memory, enough to hold four alphanumeric pages and provide limited-resolution graphics. The graphics capabilities of the CGA allowed it to display 320 by 200 pixels in four colors, or 640 by 200 pixels in two colors. The price of memory was still a limiting factor in display resolution. The 200-line vertical resolution severely impacted the CGA's alphanumeric capabilities because all characters were displayed in an eight-by-eight cell and were difficult to read. Several companies, including Hewlett-Packard, introduced their own extensions to the CGA, allowing greater resolution. The HP 45981A multimode adapter increased the resolution to 400 lines but kept the same horizontal and color resolutions and increased the memory to 32K bytes. The CGA became the lowest common denominator for graphics-based programs and, in fact, is still supported today by many applications—especially games.

In 1982 the Hercules Company introduced the Hercules Graphics Card (HGC). This adapter fully supported the alphanumeric capabilities of the Monochrome Display Adapter as well as providing 720-by-348-pixel monochrome graphics. Because of its modest cost and industry support, the HGC became very popular.

The next big breakthrough in PC video came in 1985 when IBM introduced the Enhanced Graphics Adapter (EGA). This was the first affordable PC video adapter to enter the "high-resolution" arena. It supported a resolution of 640 by 350 pixels with up to 16 colors simultaneously displayed from a palette of 64 colors. The memory required was 128K bytes. The EGA was fully backward compatible with the CGA (and with the monochrome monitor, the MDA). In 1987 the IBM PS/2 line of PCs was introduced and with it the Video Graphics Array (VGA) video adapter. The VGA has become the de facto standard of the PC industry today. The original VGA contained 256K bytes of video memory and supported resolutions up to 640 by 480 pixels with up to 16 colors simultaneously displayed from a palette of 262,144 colors. As memory prices have continued to decrease, the VGA has been enhanced. The first enhancement was the Super VGA (SVGA) which increased the resolution to 800 by 600 (or 1024 by 768) pixels. The color depth increased to allow up to 256 colors to be simultaneously displayed. Display memory was increased to 512K bytes. The VGA is fully backward compatible with the CGA, EGA, and HGC video adapters.

The video adapters mentioned above map the display memory into the system processor's memory space. All video and graphics operations are handled directly by the system processor. Newer display adapters, such as the HP Ultra VGA board, have taken the next step by increasing the VGA resolution to 1024 by 768 pixels with 256 colors available at all resolutions (increasing the display memory to 1M bytes) and providing some local graphics processing in the video display system. This frees the system processor from much of the work of updating the display and accelerates display operations.

HP Ultra VGA Board Implementation

In any design there are always trade-offs to improve performance, save board space, add more features, and so on. The Ultra VGA board implementation was confronted with some of these same trade-offs as well as the need to adopt some new technologies to provide a high-resolution, flicker-free graphics system.

Software versus Acceleration Trade-offs

In almost all nonscientific programs, video processing is the performance bottleneck. By taking some of the graphics burden off the applications, a good video solution is able to improve overall system performance dramatically. This is especially true as graphics-oriented user interfaces become more popular.

Performing the high-level graphics functions like area fill and line drawing inside the hardware has only become popular in the last few years. The MDA and CGA video solutions used video routines located in the main system BIOS (basic input/output system) of the computer and offered a few low-resolution modes. The EGA and VGA were logical extensions to MDA and CGA. They offered more modes, higher resolutions, more colors, and had their own video BIOS. There was no support for any high-level graphics functions, though a few simple graphical mixing functions like XOR were available. It wasn't until the IBM 8514 that high-level graphic functions were performed in the hardware. The 8514 is an accelerated display adapter that contains a graphics engine implemented in hardware. The problem with the 8514 is that it is not backward compatible with VGA.

Since the advent of the 8514, other video manufacturers have started to put high-level graphics functions and support for VGA on the same card. Some manufacturers add the capabilities of high-level graphics functions directly to the VGA modes while others add special video modes and VGA modes that have the extra capabilities the 8514 made popular.

Trade-offs must be made to determine where the high-level graphics functions like line drawing are implemented. In the past, an application would use the CPU to calculate all the points in a line and then write each separate dot in the line to video memory. This works very well if the CPU has bandwidth to spare and all video solutions behave the same. Since all video solutions do not work the same, each application has to either pander to the lowest common denominator or not work on all machines. Two main solutions to this problem have been implemented: video BIOS interrupt calls and application drivers.

The HP Ultra VGA video BIOS contains an industry-standard set of interrupt calls that change the configuration of the video adapter, get information about the video solution, and access all of the functions needed to work with text. All applications that know about the VGA standard can use these interrupt calls to perform the video BIOS functions. This is great for text, but there is almost nothing in the VGA BIOS that helps with drawing objects in graphics modes. Since graphical user interfaces are now becoming very popular, graphics support is very important.

Display Driver. The display driver fills the graphics support gap. A display driver is responsible for providing the means to translate application graphics commands to hardware commands and simulating capabilities not directly provided in the hardware. Each display driver is tailored to a specific application. Every application designer decides on the graphics commands needed and how they will be implemented. In the same way, each video chip maker chooses the graphics commands to implement in hardware. The application must supply a driver for every different video adapter it must run on (or the hardware manufacturer must supply a driver for every application it wants to support). This allows video manufacturers to produce software that allows specific applications to run at peak performance on their hardware.

The combination of display drivers, BIOS, and hardware provides the excellent video performance seen with the HP Ultra VGA video solution. The HP Ultra VGA board not only supports all of the modes that VGA contains, but also has a set of enhanced high-resolution modes that use a graphics engine to accelerate the most commonly used graphics functions. The enhanced modes are not standard VGA modes, but applications can get access to them via the display drivers. The drivers increase application performance by taking a high-level graphics operation like rectangle fill and performing the operation as fast as the video hardware can do it.

Applications send the display driver all graphics-related operations and the driver decides the best way to perform those operations. For example, to switch from mode three (text) to mode 201 (enhanced graphics) an application will send the driver the command to make the mode change. The driver then has to decide whether to make the mode change itself or send the command to the video BIOS. Typically, the driver will call the video BIOS for commands like mode changes. However, for commands like line drawing, the driver will usually communicate directly with the hardware to draw the line.

The implementation of display drivers is described later in this article.

Graphics Engine. The graphics engine is a state machine inside the video ASIC on the HP Ultra VGA board (see Fig. 1). Its purpose is to perform the high-level graphics functions in hardware so that the CPU is free to do other tasks. An eight-word FIFO buffer is provided so that the CPU can send all of the commands needed for at least one operation (the number of commands varies between operations). The FIFO reduces the amount of time the CPU has to wait when the graphics engine is still performing its last command.

The high-level graphics functions supported by the graphics engine include rectangle fills, line drawing, short stroke vectors, hardware cursor, bit-block transfers (BitBlt), and image transfers. Rectangle fill is the ability to fill a rectangular area of video RAM with some specific color and at the same time change what is already at that specified location in video RAM. For example, filling an area with all ones and the XOR operation will invert all colors (pixels) within the rectangle specified.

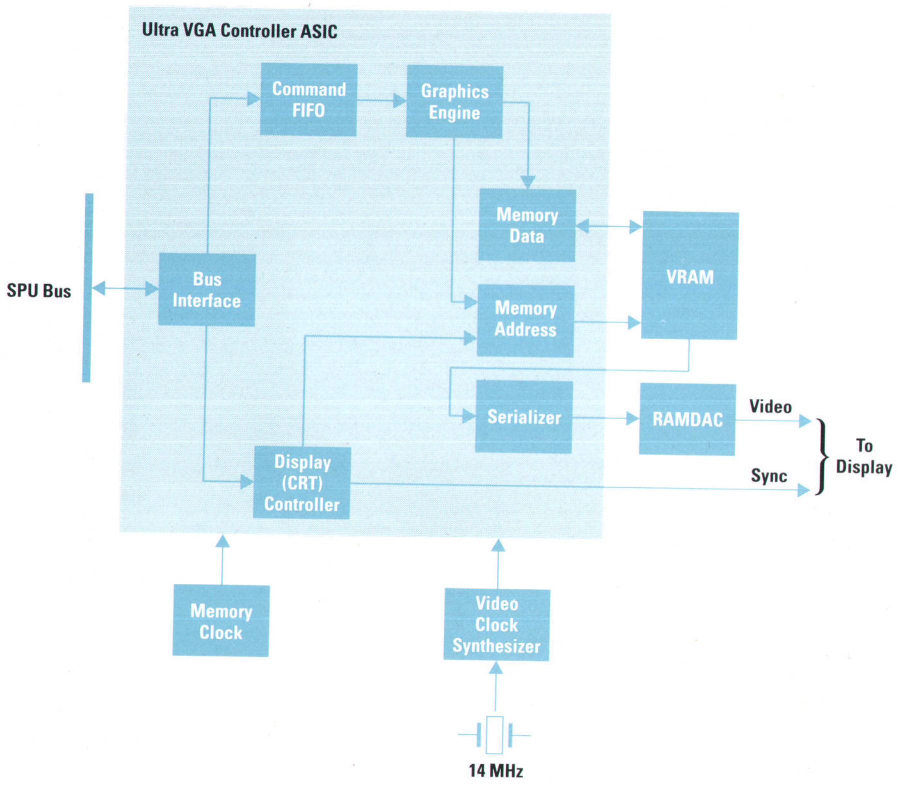


Fig. 1. A simplified block diagram of the Ultra VGA board.

Line drawing is the ability to draw a line between any two points. Short stroke vectors are a special case of line drawing in which short lines of sixteen dots or less are drawn at any forty-five degree angle (see Fig. 2). Each short stroke vector takes only one byte to specify, and the graphics engine can accept two bytes at a time. This is very handy for shapes that use many short lines, like characters, and since only one byte is sent per line, this operation is very fast.

Bit block and image transfers are for moving rectangular images around in video memory. A BitBlt is the fast transfer of a rectangular image from one location in video memory to another. Since the CPU only has to specify the source and destination coordinates and rectangle size, it can do other operations while the graphics engine does all of the work. The BitBlt operation is performed by the CPU first writing the source, destination, and rectangle size to the FIFO buffer if the FIFO has enough empty entries (the CPU requires four empty entries for this operation). Then the CPU writes a command word to the FIFO that tells the graphics engine what operation to perform. As soon as the command word propagates through the FIFO and the graphics engine receives it, the operation begins immediately. The graphics engine will read pixels from the memory within the source rectangle and the corresponding pixels from the destination rectangle. The graphics engine mixes the two sets of data in any of 256 different combinations and then writes the resulting pixels to the destination rectangle's video memory. This will continue until all the pixels in the destination rectangle are written. As soon as the graphics engine has completed this command, it will go back to the FIFO for the next command.

Image transfer performs the same function as a BitBlt except that it transfers an image from the CPU memory to video memory and vice versa. For this operation, the CPU only has to compute the number of bytes to be transferred and then write (or read) that number of bytes to or from a dedicated I/O register.

Since the graphic functions named above are constantly used with any graphical user interface, a large performance gain will be seen on any benchmark that tests video using these graphic operations. Since video is often the performance bottleneck, application-level benchmarks tend to improve as well.

After the CPU offloads operations to the graphics engine, it checks the FIFO to make sure that there is enough room before sending the next command. This means that the CPU might still have to wait (when the FIFO if full) if all it is doing is sending high-level video commands. For benchmarks that do a large number of one type of operation, the results may not indicate real performance. Because of this fact, benchmarks today are moving towards using real applications performing normal operations that can be completely automated.

Today the most useful benchmarks are applications that use a graphical user interface. Because of this, many PC benchmarks measure the performance of graphically intensive applications such as CAD programs. Two industry benchmarks, one running on Microsoft Windows and the other on AutoCAD, showed that HP Vectra 486/U machines using the HP Ultra VGA card significantly outperformed other PCs running the same benchmarks.

Host Bus versus the ISA Bus

Peripheral adapter cards are generally connected to a PC via the ISA (Industry Standard Architecture) bus, which runs at approximately 8 MHz (BCLK). BCLK is the ISA clock, which is obtained by dividing the CPU clock by either three or four depending on whether the processor clock is 25 MHz

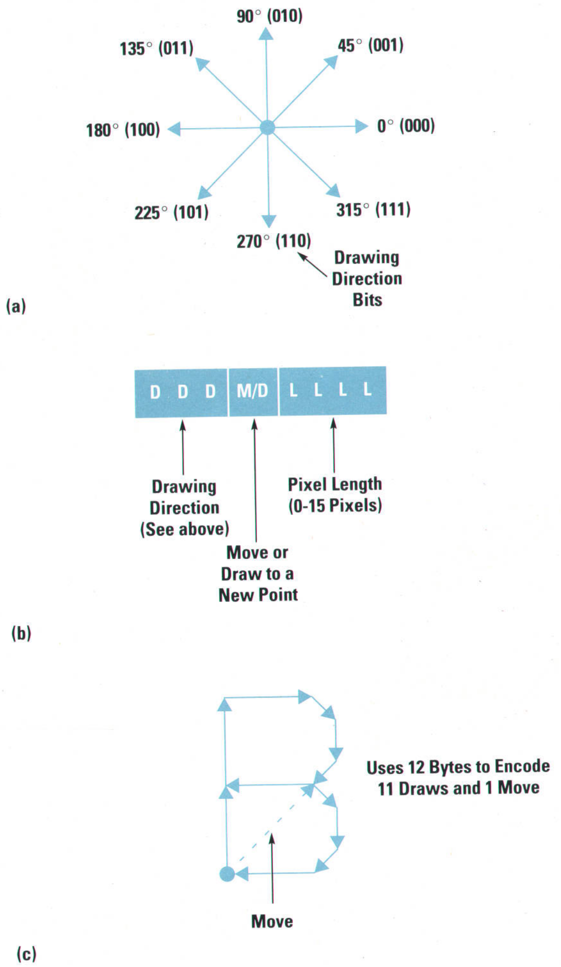


Fig. 2. (a) Short stroke vector directions. (b) Byte encoding of a short stroke vector. (c) Example of a character drawn using short stroke vectors. This character would require 48 bytes if stored as normal long vectors—12 X and 12 Y values each two bytes long.

or 33 MHz. One-byte accesses to an accessory card require a minimum of three BCLKs or approximately 375 ns. Two-byte accesses require a minimum of six BCLKs or approximately 750 ns. If the accessory board is not ready to start a cycle when it is accessed, or if the access takes longer than the minimum, additional wait states of one BCLK each are inserted. In Fig. 3 the signal BALE (bus address latch enable) signifies the start of a processor cycle and that the address on the bus is valid. Read/write (R/W)) and memory-I/O (M-I/O) are control signals indicating the direction and source or destination of the data on the bus. The read and write data signals indicate when the data must be stable for either type of transfer.

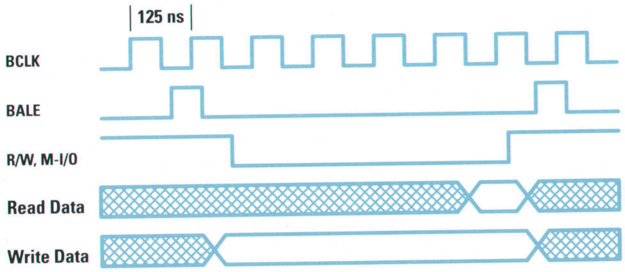


Fig. 3. A typical six-BCLK ISA cycle.

The HP Ultra VGA board is implemented as a 16-bit ISA accessory board. Because of this, its performance is limited by the bus bandwidth. For example, transferring a word from one location to another takes a minimum of six BCLKs (750 ns) even though the memory is capable of 80-ns access time. The HP Vectra 486 and 486/33T computers have an EISA bus, but the additional cost of implementing the Ultra VGA board as an EISA peripheral was not justified since it could not take advantage of the advanced features of the EISA bus such as bus mastering. Also, being an ISA board allows it to be used in other HP computers such as the Vectra RS 25/C, which doesn't have an EISA bus.

In the Vectra 486/U, the video subsystem is not a plug-in accessory board but is embedded on the processor PC board. It is connected directly to the host bus and thus can take advantage of the 32-bit bus width and the fast clock speed. The chipset used in the HP Vectra 486/U allows for four separate buses: the local bus, the host bus, the EISA/ISA bus, and the peripheral bus (see Fig. 4). The Intel486 processor and the secondary cache memory directly interface with the local bus. This bus architecture is unique to the Vectra 486/U. In most Intel486 designs, what we refer to as the host bus is the local bus and the cache shares bus bandwidth with other elements of the system. The HP 486/U gains performance by separating the local bus from the host bus since most high-speed critical operations are processor accesses to the cache. This also allows simultaneous access to main memory or mass storage by intelligent peripherals without interfering with the CPU.

The host bus is a 32-bit bus, operating at the Intel486 clock speed, which connects the main subsystems of the processor. As shown in Fig. 4, these subsystems include the memory controller, the EISA/ISA bus controller, the peripheral controller, and the video controller. The EISA/ISA bus is the backplane bus used for plug-in accessory cards. The peripheral bus connects many of the onboard subsystems in an ISA style protocol.

Devices on the host bus receive the signal HADS (host address data strobe) to begin a cycle (see Fig. 5). This causes an address decode to take place in the device. If the device recognizes the address as its own, it responds by asserting the signal HLAC (host local access). This causes other devices on the host bus to remain quiescent for the duration of the bus cycle. If no device responds, the address is propagated onto the EISA/ISA and peripheral buses. When the responding host bus device completes its operation, it responds by asserting the signal HRDY (host ready) which ends the cycle.

By comparing the timing diagrams shown in Figs. 3 and 5, it can be seen that the host bus implementation can speed up individual I/O or memory cycles by a factor of four to six. The decrease in CPU-to-peripheral transfer time and the accelerator built into the chip (described below) contribute to the superior performance of the embedded Ultra VGA subsystem.

VRAM versus DRAM

Most PC video adapters use standard dynamic RAM (DRAM) for display memory. While this is a cost-effective solution, it leads to performance penalties. Because a DRAM has a single data port, accesses from the CPU and the display controller must be time-multiplexed. The display controller must have

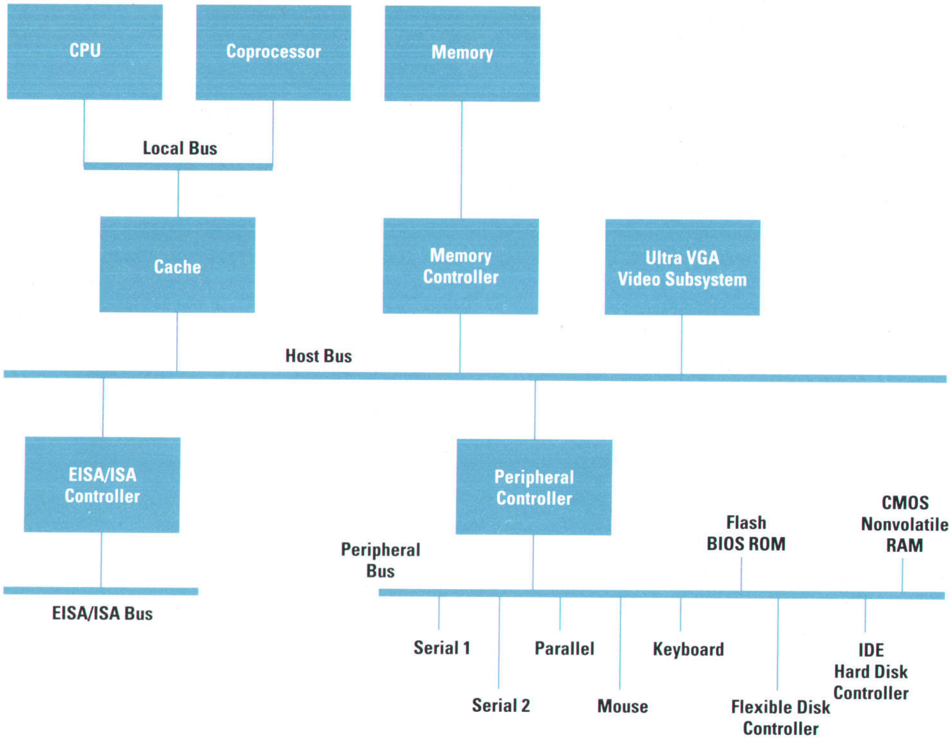


Fig. 4. Basic block diagram of the HP Vectra 486/U.

a high priority because any missing data would show up as noise or snow on the display. In the original CGA, the only time the CPU is allowed to access display memory is during the retrace intervals at the end of each scan line. This means that that the CPU can get only two or three clean accesses every 63 μs (see Fig. 6a).

The EGA and VGA architectures also use DRAM, but by using four-bit wide chips, the CPU/display interleave is brought down to 1:2 (see Fig. 6b). This allows a CPU access every 450 ns. This still necessitates slowing the CPU down, since if the access just misses, the processor has to wait 450 ns until the next access window opens.

Another RAM architecture made especially for video use is the video RAM (VRAM). VRAM is a dual-ported device that allows the CPU almost unlimited access to the display memory while still maintaining a noise-free display. Fig. 7 shows a simplified drawing of a 256-bit VRAM. The RAM array in this case is 16 rows of cells by 16 columns. In an actual VRAM the array would be 64K bytes in a 256-by-256 array.

In normal DRAM accesses, a row is selected by the row address and read in to the sense amplifiers. The column address is used to select one of the column sense amplifiers to read or write a single bit of the array.

In a VRAM, access to a row is also selected by the row address. However, instead of only one column being selected, all of the columns are simultaneously read into a serial shift register. The data is then shifted out of the shift register as it is needed by the display. In this way the display controller need only lock out the CPU for one cycle out of every 256 (or less depending on the width of the VRAMs) to present a clean display.

The Ultra VGA board uses the VRAM mode when it is operating in its enhanced mode. In the standard VGA mode of

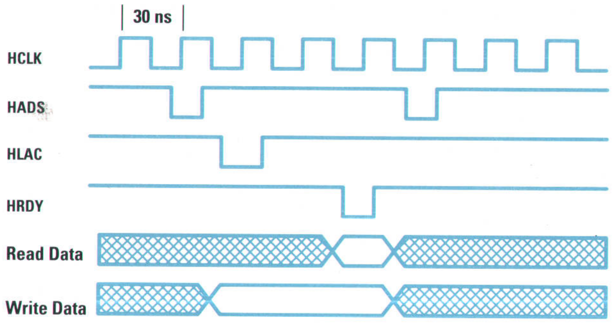


Fig. 5. Typical host bus cycle.

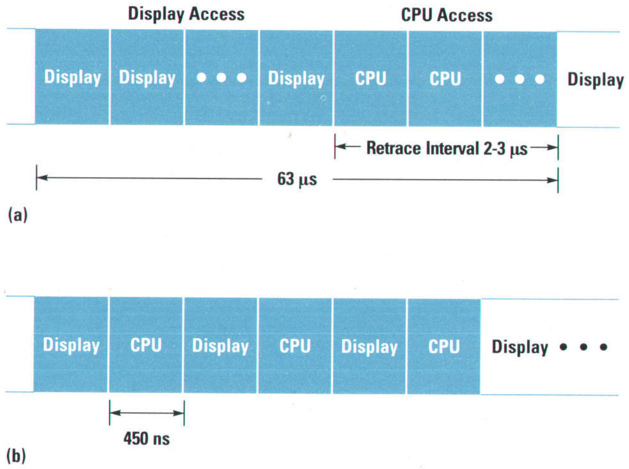


Fig. 6. CPU/display memory accesses. (a) In the original CGA implementation the CPU gets access only during retrace intervals. (b) In the EGA and VGA architectures the CPU gets access to memory one out of every two memory cycles.

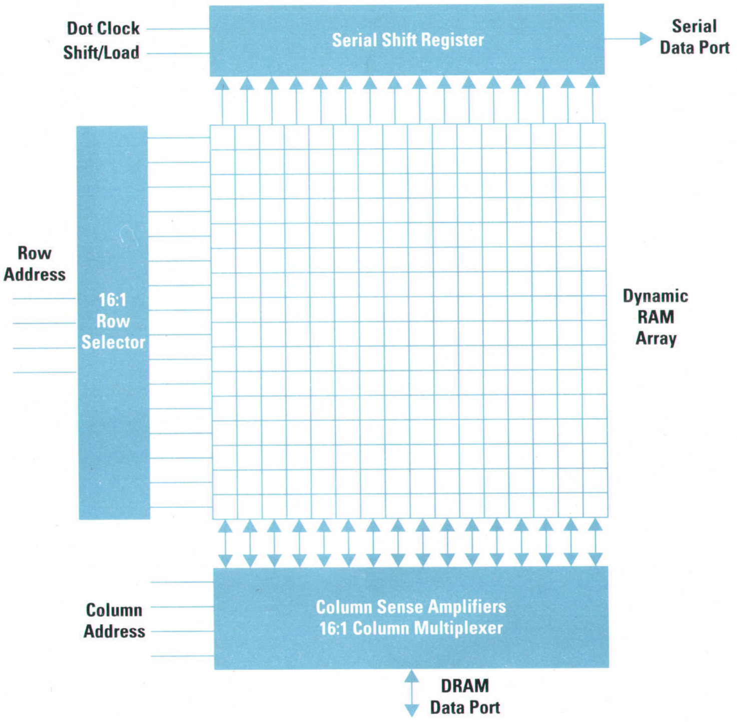


Fig. 7. Simplified diagram of a VRAM.

operation the Ultra VGA board accesses video memory as if it were DRAM.

Clock Synthesizer

Because of the many different display resolutions and monitor characteristics associated with the Ultra VGA board, up to 16 different video dot clock frequencies are needed. The board space needed would be prohibitive if these clocks were generated with discrete crystals or oscillators. Instead we use a clock synthesizer IC. This relatively new chip combines analog and digital circuitry on the same chip. It contains an oscillator, a phase-locked loop, and digital dividers that drive the phase-locked loop. Except for the reference frequency crystal (14.31818 MHz) and an RC filter, all of the necessary components are contained on the chip. This gives enormous capability in very little board space.

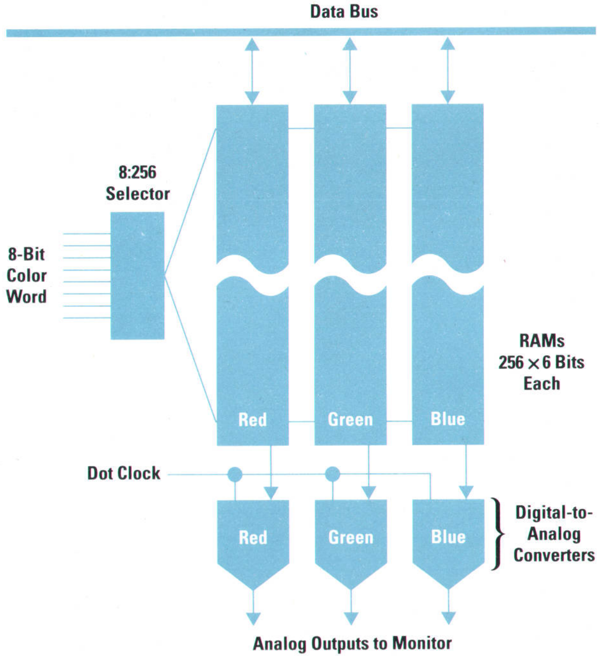


Fig. 8. Simplified diagram of a RAMDAC.

RAMDACs

The CGA runs its monitor with four digital signal lines: red, green, blue, and intensity. This allows a maximum of 16 colors to be displayed. In graphics modes the colors are fixed by the hardware and selected from two palettes of four colors each.

The EGA extends this by providing six digital signals: red, red′, green, green′, blue, and blue′. This allows a maximum of 64 different colors to be displayed. The digital-to-analog converters (DACs) are built into the monitor and the 64 shades are fixed by the manufacturer. The EGA board has a palette consisting of 16 six-bit entries, and each palette entry can be programmed to select one out of 64 shades.

The VGA doesn't drive the monitor with digital signals, but uses analog signals instead, one for each primary color (red, green, and blue). By varying these signals, an almost infinite range of colors can be displayed. The standard VGA uses a RAMDAC with 256 eighteen-bit entries. Each of the entries has six bits each for red, blue, and green (see Fig. 8). The maximum number of colors that can be generated is 2^{18} or 262,144. Of these, any 256 can be displayed simultaneously.

Ergonomics in PC Graphics

Higher Resolutions and Higher Refresh Rates

Since the establishment of the IBM VGA as a PC graphics standard, there has been steady progress in the development of higher screen resolutions. The IBM VGA offers a maximum resolution of 640 by 480 pixels with 16 colors. Recent super-VGA boards from various manufacturers support higher resolutions of 800 by 600 and 1024 by 768 pixels, along with 256 colors.

The most direct benefit of higher screen resolution is a larger display area for the user. This translates to advantages such as the ability to display more rows and columns of a spreadsheet, or larger sections of a word processor document.

The display refresh rate has also been steadily improved to address the problem of screen flicker. Flicker is perceived by the user as a direct result of the monitor screen not being refreshed at an adequate rate. Since all PC monitors are based on cathode ray tube (CRT) technology, the contents of the screen are not static but are constantly being swept onto the screen phosphor on a line-by-line basis. If the graphics system does not support an adequate screen refresh rate, pixel intensity will have time to decrease between successive refresh cycles, resulting in the perception of rapid screen flashing, or flicker. Viewing a monitor screen with significant flicker, especially for long periods of time, can result in eyestrain and other health hazards. The recent improvement in screen refresh rates has been largely successful in reducing the problems associated with screen flicker.

The standard VGA implements a screen refresh rate of 70 Hz for all text and graphics modes, except for 640 by 480 graphics modes, which offer a 60-Hz rate. The Video Electronics Standards Association (VESA) provides standards for refresh rates at higher resolutions including 72 Hz for 800 by 600 resolution and 70 Hz for 1024 by 768 resolution.

Monitors

Increasing the screen resolution or the refresh rate will directly increase the graphics output horizontal scan rate (Hsync), a measure of the time between successive horizontal display lines on the screen. The standard VGA uses a fixed Hsync rate of 31.5 kHz for all text and graphics modes. Combinations of higher resolutions and higher refresh rates can yield an Hsync rate ranging from 31.5 kHz to beyond 64 kHz.

All standard VGA-only analog monitors on the market can support only the standard 31.5-kHz Hsync rate. To properly support modes with higher Hsync rates, dual-sync or multisync monitors are required. Dual-sync monitors, such as the HP Super VGA Display (HP D1194A) and the HP Ergonomic Super VGA Display (HP D1195A), can support Hsync rates other than 31.5 KHz. The capabilities of these two monitors and others are listed in Fig. 9.

Multisync monitors are typically capable of synchronizing to a continuous range of Hsync frequencies, allowing them to support standard VGA modes as well as higher resolutions. The HP Ultra VGA Display (HP D1193A) is an example of a multisync monitor.

The HPUVGA Utility

With the choice of multiple refresh rates and monitors with different resolutions, the user needs to configure the graphics system to select the correct refresh rates for the resolutions supported by a given monitor. The HP Ultra VGA board is shipped with a configuration utility called HPUVGA.EXE, which allows the user to customize the Ultra VGA board output to any HP PC graphics monitor.

Embedded within the HPUVGA utility is information pertaining to the synchronization capabilities of all of HP's PC graphics monitors. By correctly selecting the monitor in use, the user is able to view the refresh rates supported by the monitor at graphical resolutions of 640 by 480, 800 by 600, and 1024 by 768 pixels. In cases in which the monitor can support two or more refresh rates for a given resolution, the user is given a choice. All refresh settings are saved in an HP CMOS video byte, which is described later.

Horizontal Resolution	Vertical Resolution	Colors	Mode Type	Memory Required	Hsync (kHz)	Vsync (Hz)	D1192A	D1187A D1193A	D1194A	D1195A
32 columns	25 rows	16	Text	512K Bytes	31.5	70	✓	✓	✓	✓
640	480	256	Graphics	512K Bytes	31.5	60	64 Shades	✓	✓	✓
640	480	256	Graphics	512K Bytes	37.9	72		✓	✓	
800	600	16	Graphics	512K Bytes	37.9	60		✓	✓	
800	600	16	Graphics	512K Bytes	48.1	72		✓		✓
800	600	256	Graphics	1M Byte	37.9	60		✓	✓	
800	600	256	Graphics	1M Byte	48.1	72		✓		✓
1024	768	16	Graphics	512K Bytes	48.4	60		✓		
1024	768	16	Graphics	512K Bytes	56.5	70		✓		
1024	768	256	Graphics	1M Byte	48.4	60		✓		
1024	768	256	Graphics	1M Byte	56.5	70		✓		

D1192A—HP Monochrome Display

D1187A—HP 20-Inch High-Resolution Display

D1193A—HP Ultra VGA 17-Inch Display

D1194A—HP Super VGA Display

D1195A—HP Ergonomic Super VGA Display

Fig. 9. Summary of HP PC monitor capabilities. In addition to the capabilities listed, all of the monitors provide standard VGA modes.

The HPUVGA utility also supports emulation modes for the MDA, HGC, and CGA PC graphics standards. Two additional 132-column text modes, with 25 and 43 rows respectively, can also be set via the HPUVGA utility.

HP CMOS Video Byte

Refresh rate settings for graphics resolutions of 640 by 480, 800 by 600, and 1024 by 768 pixels are saved in the HP CMOS video byte. The assignments for each bit in this byte are:

Bit 7: Alternate I/O port select
Bit 6: Reserved
Bits 4 and 5: 1024 by 768 refresh timing
Bits 2 and 3: 800 by 600 refresh timing
Bits 0 and 1: 640 by 480 refresh timing

The monitor timings for all supported video modes are stored in table format in the video BIOS, with one table entry per video mode. When an application calls the Int 10h set-mode function of the video BIOS to enter a specific accelerated graphics mode, the video BIOS accesses the HP CMOS video byte to determine the refresh rate currently selected, then uses the corresponding timing table to get the correct refresh rate. This scheme allows the refresh rate control to be application independent.

Since HP CMOS memory is a nonvolatile system resource, the refresh rate settings are preserved in the same way as other standard system configuration information. This scheme is capable of supporting operating systems besides MS-DOS®. Alternatives to HP CMOS memory for saving refresh rate settings have been carefully considered. Adding EEPROM hardware to the HP Ultra VGA board to store the refresh rates had the disadvantages of high cost and increased design complexity. Using a TSR program (memory resident software) to preserve the refresh rates would have worked only for MS-DOS, and other systems such as the OS/2 and UNIX* operating systems would also require specific memory resident software. Memory resident software would occupy valuable system memory and reduce ease of use.

Display Drivers

A display driver is a distinct program module that is made up of a group of display functions that provide a standard interface between an application and a particular type of video display hardware.

The HP Ultra VGA accessory card provides many features, such as hardware line drawing, bit-block image transfer (BitBlt), rectangle fill, and hardware clipping. However, these features can only be accessed through some special video enhanced modes which are unique to the graphics processor in the HP Ultra VGA card. In most cases, application programs, such as Microsoft Windows and AutoCAD, do not know (and do not want to know) how to enter these enhanced modes. It is the manufacturer-specific display driver that lets the application program take full advantage of the graphics processor.

For example, to make the HP Ultra VGA card work in 256-color enhanced mode with 1024-by-768-pixel resolution, a display driver has to call the BIOS interrupt 10h with registers AX=0x4F02 and BX=0x205. In general, to set the display in one of the HP Ultra VGA enhanced video modes, the driver calls BIOS interrupt 10h with the AX and BX registers set to values that represent different resolutions and colors.

To access hardware line drawing, BitBlt, and rectangle fill features of the HP Ultra VGA hardware, the display driver sets the drawing command register at I/O address 9AE8h. Fig. 10 shows a definition of each bit in this register.

For example, when an application wants to draw a line on the screen, the display driver sets the following bits in the drawing command register at I/O address 9AE8h:

Bits	Setting	Meaning
13-15	001	Draw Line Command
04	1	Draw = Yes
00	1	Write

The driver also has to find out the drawing direction to fill in bits 5 to 7.

Another important feature of the HP Ultra VGA card is the short stroke vector drawing ability. Using short vectors for displaying text in the graphics mode improves video performance. When an application program requests to display text on a high-resolution graphics screen, the display driver sets the short stroke vector command register at the I/O address 9EE8h. Fig. 2b shows the bit definitions for the short stroke vector register.

Typically, an application program uses a standard interface to the display so it doesn't have to be concerned with the type of hardware installed on the machine in which it is running. This isolates the application program from the display hardware. For example, most Windows applications are written without regard to the type of video adapter used. Instead, the programs are written to interface with Microsoft Windows.

Bit	Meaning
15, 14, 13	Command Type 001: Draw Line 010: Rectangle Fill 110: BitBlt
12	Byte Swap (1 = Yes 0 = No)
11	0
10	(Reserved)
09	Bus Size (1 = 16 bit 0 = 8 bit)
08	Wait (1 = Yes 0 = No)
07, 06, 05	Drawing Direction in Degrees 000: 0-45 001: 45-90 010: 90-135 011: 135-180 100: 180-225 101: 225-270 110: 270-315 111: 315-0
04	Draw (1 = Yes 0 = No)
03	Direction Type (1 = Radial 0 = x-y)
02	Last Pixel (1 = Off 0 = On)
01	Pixel Mode (1 = Multiple 0 = Single)
00	Read/Write (1 = W 0 = R)

Fig. 10. Definitions of each bit in the drawing command register.

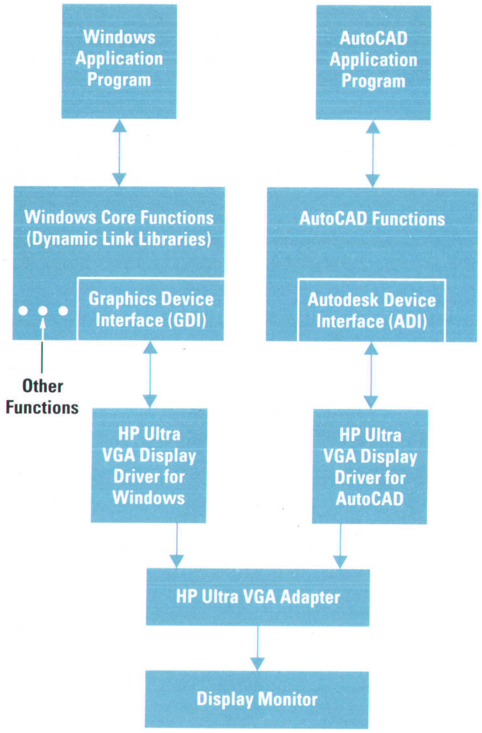

Windows Application Program
AutoCAD Application Program
Windows Core Functions (Dynamic Link Libraries)
Graphics Device Interface (GDI)
AutoCAD Functions
Autodesk Device Interface (ADI)
Other Functions
HP Ultra VGA Display Driver for Windows
HP Ultra VGA Display Driver for AutoCAD
HP Ultra VGA Adapter
Display Monitor

Fig. 11. Software hierarchy from the application to the display driver.

The video adapter's display driver takes care of writing to the display hardware. The Windows display driver works with any Windows program. By going through a standard interface, the display driver developer and the application program developer are isolated from each other (see Fig. 11).

The Windows Display Driver

The display driver for Windows is a dynamic link library that consists of a set of graphics functions for the HP Ultra VGA display card. These functions translate device independent graphics commands from the Windows graphical device interface (GDI) into the commands (such as the drawing command described above) and actions the Ultra VGA graphics engine needs to draw graphics on the screen. These functions also give information to Windows and Windows applications about color resolution, screen size and resolution, graphics capabilities, and other advanced features, such as BitBlt, line-drawing, polygon fill, and hardware cursor support. Applications use this information to create the desired screen output.

The HP Ultra VGA Windows display driver is based on the sample driver for the IBM 8514 graphics adapter. The source code for the 8514 driver is available from Microsoft's Driver Development Kit. Like most Windows display drivers, the Ultra VGA driver provides the following basic functions:

- Output. Draws various shapes.
- Enable. Starts or resumes display activity.
- Disable. Stops display activity.
- RealizeObject. Creates physical objects (e.g., pens, brushes, and device fonts) for exclusive use by the display driver. This is where the translation between device independent (or logical) and device-optimal (or physical) objects takes place.
- ColorInfo. Translates between logical colors, which are passed by Windows as double word RGB values, and physical colors recognized and used by the Ultra VGA display drivers.
- BitBlt. Supports bit-block transfers by copying a rectangular block of bits from bitmap to bitmap while applying some specified logical operations to the source and destination bits. A bitmap is a matrix of memory bits that defines the color and pattern of a corresponding matrix of pixels on the device's display screen. Bitmaps provide the ability to prepare an image in memory and then quickly copy it to the display.
- ExtTextOut. Draws a string of characters at a specified location on the screen and clips any portion of a character that extends beyond a bounding box of the string.
- StrBlt. Supports text drawing for the earlier versions of Windows. (It just makes a call to ExtTextOut.)
- Control. Passes special control information to or receives special information from the Ultra VGA display driver.

Besides the functions listed above, the following features have been added to take full advantage of the graphics engine in the Ultra VGA.

- Different Resolutions. Separate display drivers are provided to support resolutions of 1024 by 768, 800 by 600, and 640 by 480 pixels with 256 colors.
- Hardware Cursor. An onboard hardware cursor (64 by 64 pixels) for fast cursor movement in the enhanced mode.
- Fast Polyline Draw. Onboard hardware is used to draw solid polylines at a very fast speed.
- Polygonal Capabilities. An onboard drawing command register and hardware are used for quick rectangle fill and scanline drawing.
- Fast Rectangular Clipping. Rectangular clipping is provided via a clipping window boundary register and hardware that discards points that are outside of a specified rectangle or region drawn on the screen.
- High-Speed BitBlt. Onboard hardware is used for high-performance bitmap image transfer operations.
- FastBorder Function. A function that draws borders for windows and dialog boxes very quickly.
- Save Screen Bitmap. The SaveScreenBitmap function allows Windows to save bitmaps temporarily in offscreen video memory. This function speeds the drawing operations that require restoring a portion of the screen that was previously overwritten.
- Support for Large Fonts. Support for large fonts is provided in which the font and glyph information can exceed 64K bytes.
- DIBs Support. This function converts device independent bitmaps (DIBs) to physical format for direct transfer to the display without applying a raster operation. Note that a DIB is a color bitmap in a format that eliminates the problems that occur when transferring device dependent bitmaps to devices having difference bitmap formats.
- Support Device Bitmap. A device bitmap is any bitmap whose bitmap bits are stored in device memory (such as RAM on a display adapter) instead of main memory. Device bitmaps can significantly increase the performance of a graphics driver and free system memory for other uses.

- Font Caching. Font caching is temporarily saving the most recently used fonts in offscreen video memory. This function speeds up text-redisplaying operations.
- Small and Large Fonts. The Ultra VGA display driver provides both small and large fonts in the 1024-by-768 high-resolution mode.
- Vector Fonts. The Ultra VGA display driver supports vector fonts. A vector font is a set of glyph definitions, each containing a sequence of points representing the start and end points of the line segments that define the appearance of a character in a particular font.

Working Together

The functions in the Ultra VGA display driver and the Windows graphical device interface (GDI) work together to make efficient use of the features provided in the HP Ultra VGA board. The rest of this section describes how these two entities work together to initialize the display and perform some simple graphical operations.

When the user types WIN to start Windows, a small program WIN.COM determines the mode in which Windows is to run. If it determines that it can run in the enhanced mode, Windows runs KRNL386.EXE (via WIN386.EXE). While initializing, Windows checks the DISPLAY.DRV setting in the SYSTEM.INI file to determine the file name of the display driver to load. The HPUxxx.DRV driver modules are the display drivers for different resolutions and video memory configurations. The Windows graphical device interface (GDI) then calls the selected display driver's initialization routine.

During initialization, the Windows GDI makes two calls to the Enable function in the Ultra VGA display driver. After the first call, the Enable function then returns to the GDI the GDIINFO data structure, which describes the general physical characteristics and capabilities of the HP Ultra VGA graphics engine. The GDI uses this information to determine what the Ultra VGA display driver can do and what the GDI must simulate.

The second time the GDI makes a call to the Enable function, the display driver does three things. First, it initializes the Ultra VGA graphics engine to be ready to run Windows. This includes saving the current mode, using the video BIOS function 10h calls to set the enhanced display mode and colors, load the palette, and so on. Next, the Enable function calls the hook_int_2Fh function in the display driver so that each call to interrupt 2Fh will be checked to detect any screen Switch function calls. This is because in a preemptive multitasking environment such as 386-enhanced-mode Windows, the display driver has to save and restore the display hardware settings, such as video mode and register data, whenever Windows is changed between a Windows application and a non-Windows application.

The last thing the Enable function does is to initialize and copy the PDEVICE data structure. The PDEVICE data structure defines physical objects rather than bitmaps. Physical objects define the attributes (such as color, width, and style) of lines, patterns, and characters drawn by the display driver. Physical objects correspond to the logical pens (used to draw polylines and borders around objects drawn by the Output function), brushes (used to fill figures drawn by the Output function and to fill rectangular areas created by the BitBlt function), and fonts (used by the ExtTextOut function to draw text). Physical objects also contain Ultra VGA hardware device dependent information that a display driver needs to generate output. These physical objects are created by the RealizeObject function.

After the RealizeObject function is finished creating the default pens and brushes for Windows and the brushes needed to draw the desktop and fill the Program Manager window, the BitBlt and ExtTextOut functions are called to do all the drawing on the screen. First, the BitBlt function draws a rectangle on the screen with the background color by using a pattern copy operation. Next, the BitBlt function is called to draw some borders and rectangles. Finally, to complete display initialization all the icons, text, and pictures are put on the Windows screen by using functions such as BitBlt, ExtTextOut, and the brushes created by the RealizeObject function.

When a Windows application requests to draw a line on the screen, the GDI checks the dpLines entry in the GDIINFO structure, which was filled by the display driver during initialization, to see if the display driver supports line drawing. Since the Ultra VGA driver supports hardware line drawing, the GDI calls the Output function to draw a line on the screen. Otherwise, the GDI has to simulate line drawing by combining scan lines and polylines.

If a Windows application asks to display text on the screen, the GDI calls the ExtTextOut function in the display driver. The ExtTextOut function receives a string of character values, a count of the characters in the string, a starting position, a physical font, and a DRAWMODE structure. These values are used to create the individual glyph images on the screen.

Finally, when a user asks to quit Windows, the GDI calls the Disable function in the display driver. This function frees any resources associated with the physical device and restores the Ultra VGA hardware to the state it was in before Windows started. After the display driver returns from the Disable function, the GDI frees the memory it allocated for the PDEVICE structure and frees the display driver, removing any driver code and data from memory.

Acknowledgments

The authors would like to thank the following individuals who participated in the development of the HP Ultra VGA products: Mark Linsky, Mark Hanlon, Jean-Claude Roy, Larry Durandette, Kevin Owen, Kiran Mundkur, Dave Wilkins, and Mike Milani. We would also like to thank all the members of the HP Vectra 486/U hardware, BIOS, and utilities teams for their assistance and cooperation.

POSIX Interface for MPE/iX

Differences in directory structure, file naming conventions, and security were among the areas in which mechanisms had to be developed to enable the POSIX and MPE XL interfaces to coexist on one operating system.

By Rajesh Lalwani

The IEEE standard for a Portable Operating System Interface, or POSIX, defines a standard operating system interface and environment to support source level application portability. POSIX specifies the functions and services an operating system must support and the application programming interface to these services and functions.

POSIX 1003.1, which defines a standard set of programmatic interfaces for basic operating system facilities, and POSIX 1003.2, which specifies an interactive interface that provides a shell and utilities similar to those provided by the UNIX* operating system (see Fig. 1), are integrated in the MPE XL operating system to form the MPE/iX operating system, which runs on the HP 3000 Series 900 system.

The programmatic interface and directory structure of POSIX 1003.1 allow MPE/iX users to use POSIX functionality without any impact on existing HP 3000 applications. Moreover, MPE† applications are able to access POSIX files, and POSIX applications are able to access MPE files. Thus, MPE/iX provides interoperability and integration between MPE and POSIX applications and data.

† From here on MPE will be used to refer to the MPE XL version of the MPE operating system.

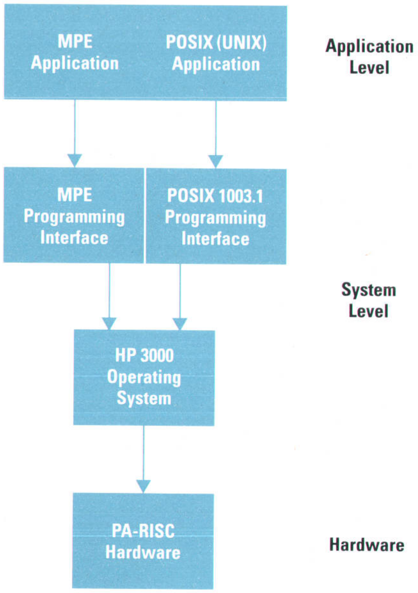


Fig. 1. MPE and POSIX applications coexist on the same system.

POSIX is significantly different from MPE in a number of technical areas such as directory structure, file system, security, user identification, file naming, process management, and signals. In the summer of 1990, a POSIX architecture team was formed to look at these differences and to architect a way in which MPE and POSIX could coexist on HP 3000 Series 900 systems.

Despite the differences between MPE and POSIX concepts, the team had no problem visualizing MPE and POSIX as one operating system. To achieve a successful integration of POSIX and MPE, several stumbling blocks had to be overcome. This paper describes the problems encountered in merging POSIX and MPE in three major technical areas: directory structure, file naming, and security. The paper also describes the alternatives rejected by the POSIX architecture team.

Directory Structure

MPE has a fixed, three-level directory structure. In this model, the directory tree consists of one or more accounts. Each account contains one or more groups and each group has zero or more files in it (see Fig. 2). On the other hand, POSIX supports the notion of a hierarchical directory structure. Fig. 3 shows a typical POSIX directory. Some of the features that POSIX requires in a directory structure include:

- Support for the . and .. entries upon creation of a directory
- Support for a true hierarchical directory with file names of at least 14 characters and path names of at least 255 characters
- Support for the POSIX rule for directory deletion, that is, the directory must not contain any entries other than . and .. for the unlink of a directory to succeed.

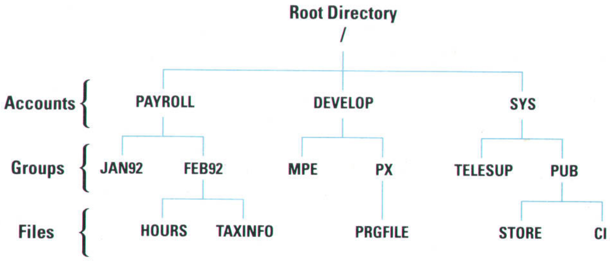


Fig. 2. A typical MPE directory.

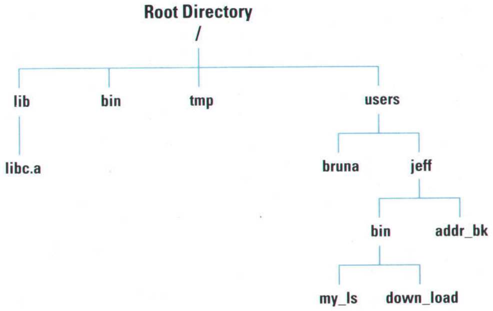


Fig. 3. A typical POSIX directory.

MPE groups and accounts are different from the POSIX directories because of the special information contained in them. To integrate the POSIX and MPE directory structures successfully, we had to consider removing this special information from MPE groups and accounts to accommodate the new directory structure. The following lists some of this special information.

- On MPE, accounts are used to manage collections of users for file sharing. Each MPE account entry contains a pointer to a list of users that are members of that account.
- MPE groups and accounts keep track of three resources: disk space, connect time, and CPU time.
- MPE groups and accounts contain security information that is used to evaluate permission to access certain files.

One of the challenges for the architecture team in integrating the hierarchical directory of POSIX with the MPE directory structure was to make the directory generic enough so that future standards could be supported and the accounting and user identification functions could be removed from the directory in the long term. These efforts resulted in the directory shown in Fig. 4.

Conceptually, the new directory structure is just a tree structure (a directed acyclic graph, to be more precise). The root of this tree is designated by slash (/). The root may have files and directories under it. Each directory may have files and directories under it. There is no architectural limitation on the depth of the directory tree.

The only restriction on the directory structure has to do with the MPE groups and accounts. MPE accounts can only exist as immediate descendants of the root directory and MPE groups can only exist as immediate descendants of MPE accounts. This means that although files and directories can be placed anywhere, MPE directory objects (groups and accounts) are restricted to their conventional locations.

Accounts and groups are distinguished from POSIX files and directories based upon how they are created. An account is a directory that is created by the :NEWACCT command and purged via the :PURGEACCT command. Similarly, a group is a directory that is created by the :NEWGROUP command and purged with the :PURGEGROUP or :PURGEACCT commands.

In the new directory structure all users are registered in the UID (user identifier) database required by POSIX, and the combination of the UID and GID (group identifier) databases replaces most of the information formerly held in the MPE account and user directory nodes.

Rejected Directory Designs

In the beginning the architecture team considered the idea of a dual-root directory structure. Fig. 5 shows an example of this idea. In this scheme, a directory called MPEXL would exist directly below the root of the hierarchical directory. This directory would be the root of the MPE directory containing all the accounts, groups, and files. Underlying this proposal was the assumption that the MPE environment would not be changed. Factors motivating this proposal were the desire to maintain compatibility with existing applications, to eliminate the need to modify existing directory services, and to isolate the POSIX hierarchical directory services from having to interact with MPE directory structures. This idea was rejected by the architecture team for several reasons. First, this would make the hierarchical directory nonuniform. The MPEXL directory (shown in Fig. 5) and all its descendants would not follow the normal hierarchical directory rules, and applications would have to make exceptions for them. Second, this would prevent users from incorporating the hierarchical directory into their current environment. Finally, this proposal was not in line with the long-term goal of making all the directory objects in the system one type.

Another idea was to use the same dual-root structure but not externalize the fact. With this idea, the directory services would determine which of the two roots to use when searching for a particular name, and duplicate entries would not be allowed in the two roots. Thus, for the directory shown in Fig. 5, a name like CI.PUB.SYS would only refer to CI.PUB.SYS in the MPE directory, and a POSIX interface would not be able to create a directory /SYS. While this solution solved some of the problems with the previous alternative, it still had some problems. For example, the directory services would have to do an extra search just to determine which root to use, thus affecting performance. More important, MPE and POSIX directories would not have been integrated.

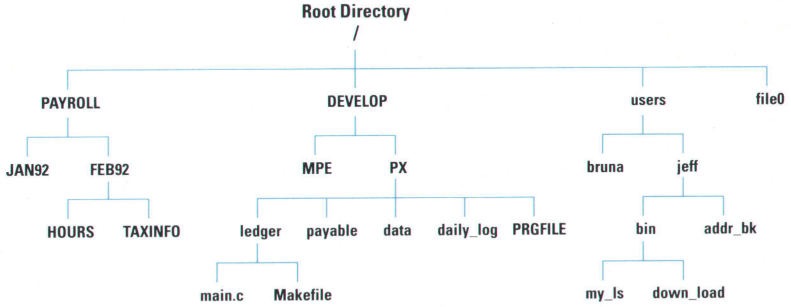


Fig. 4. A typical MPE/iX directory.

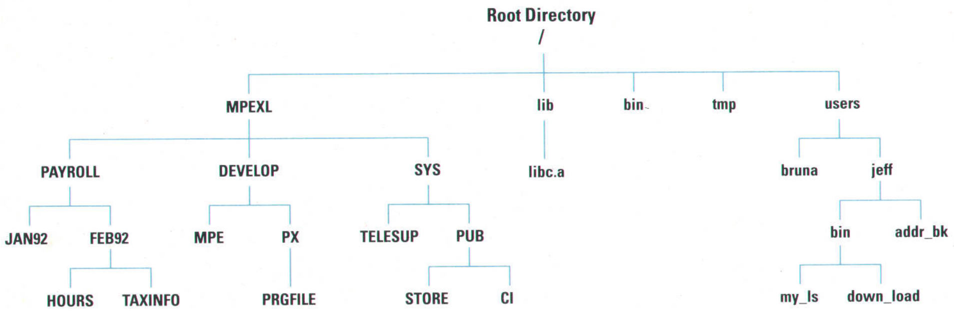


Fig. 5. A directory structure that was rejected for MPE/iX.

File Naming

In designing the file naming rules, the main objective was that the existing MPE interfaces such as MPE intrinsics and command interpreter (CI) commands should be able to refer to all objects in the same way they did before MPE/iX. The familiar MPE objects can be referred to by using the syntax for file names file.group.account, file.group, or file. The entire file name is first upshifted by the MPE name server. For example, if the file name is prgfile.px.develop, the MPE name server refers to the file PRGFILE in group PX in account DEVELOP. Similarly, if the file name is, say, prgfile.px, it refers to the file PRGFILE in group PX in the logon account. Finally, if the file name is fully unqualified (no group or account), say, prgfile, it refers to the file PRGFILE in the current working directory (CWD). When a user logs on, the CWD is the same as the user's logon group. So unless the user has explicitly changed the CWD, a fully unqualified file name such as prgfile continues to refer to the same object as before MPE/iX (e.g., file PRGFILE in logon group PX.DEVELOP).

The file named down_load in Fig. 4 cannot be referenced through the MPE name server because the file is not in an MPE group and it doesn't have a valid MPE file name. The file name must somehow escape being processed by the MPE name server so that it can be processed by the POSIX name server. This is done by using the characters . or / at the beginning of a file name. For file names beginning with . or /, the MPE name server does not upshift the name, but passes it to the POSIX name server. Therefore, the file down_load in Fig. 4 can be referred to by using the name /users/jeff/bin/down_load (the complete path name). Alternately, if the CWD has been changed to /users/jeff, the file can also be referred to by the name ./bin/down_load. Fig. 6 illustrates the MPE name server rules for MPE and POSIX file names.

	File Name Passed to MPE Name Server	Actual File Name Used
MPE File Names	a.b.c	A.B.C (Group = B, Account = C)
	a.b	A.B.logon account
	a	CWD/A (Not CWD/a)†
POSIX File Names	./a	CWD/a
	./a.b	CWD/a.b
	/c/b/a	/c/b/a (Not /C/B/A)†
	a_b	Error!
	./a_b	CWD/a_b

† Remember that MPE upshifts file names and POSIX does not.

Fig. 6. MPE name server rules for MPE and POSIX file names.

The name server used by POSIX 1003.1 functions and POSIX 1003.2 commands is the POSIX name server. However, MPE interfaces such as intrinsics and CI commands escape to the POSIX name server when the file name begins with the . or / escape characters (see Fig. 7).

The POSIX name server can also refer to all the objects in the directory. For naming purposes, MPE accounts and MPE groups can be treated as directories. Hence, the file PRGFILE.PX.DEVELOP can be referred to as /DEVELOP/PX/PRGFILE. Since the POSIX name server does not upshift names, the name /develop/px/prgfile cannot be used to refer to this file. Since file names are always in POSIX syntax in POSIX applications, they don't have to begin with . or /. Thus, if the CWD is /DEVELOP/PX/ledger (see Fig. 4), the file name main.c will not refer to the file MAIN in group C of the logon account, but to the file named main.c in the CWD. The . is a valid POSIX character.

A problem with the approach just described is that some interfaces might have to do special processing to accommodate POSIX file names. For example, the MPE :listfile command might have to query the MPE name server to

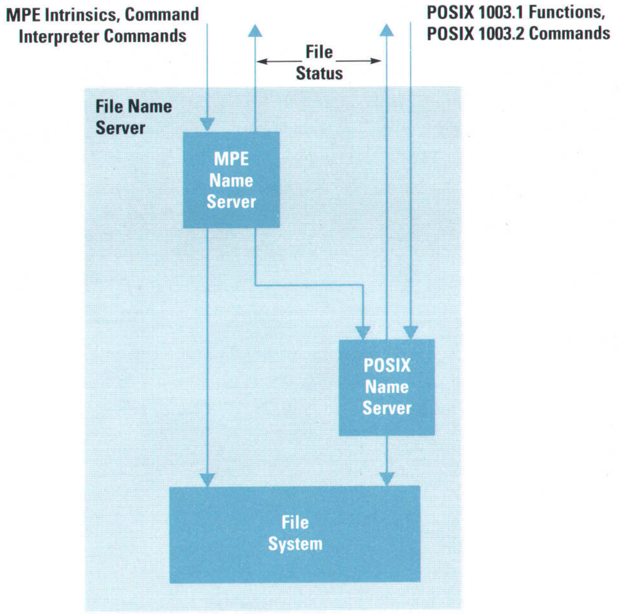

MPE Intrinsics, Command Interpreter Commands
POSIX 1003.1 Functions, POSIX 1003.2 Commands
File Status
File Name Server
MPE Name Server
POSIX Name Server
File System

Fig. 7. The POSIX and MPE name servers. The MPE name server escapes to the POSIX name server if the file name begins with . or /.

determine the type of file name it is dealing with (POSIX or MPE) before accessing the file system. However, this problem is minimized because there is one central file name server for parsing both MPE and POSIX file names.

Rejected File Naming Alternatives

The architecture team considered and rejected the following alternatives for file names using the MPE command interpreter.

PXUTIL. This program would have been an MPE system utility used for supporting POSIX file names. For example, in PXUTIL a user could have purged a POSIX file as follows:

```
pxutil> purge /users/jeff/addr_bk.
```

PXUTIL would have also supported the POSIX chdir() function, allowing the above file to be purged as follows:

```
pxutil> chdir /users/jeff/
pxutil> purge addr_bk
```

Among the few functions provided by PXUTIL would have been a mechanism (using : as the first character) for escaping to the MPE command interpreter so that the user could use MPE commands from within PXUTIL. This would have created problems when the CWD was different from the logon group. Consider the following scenario:

```
:hello jeff.develop,px          (logon group px.develop)
:pxutil                         (run pxutil)
pxutil> chdir /users/jeff/      (change working directory)
pxutil> :listfile @             (escape back to MPE)
```

Ideally, in this example the MPE :listfile command should display the files in the directory /users/jeff/. What happens instead is that the :listfile command transfers control back to the MPE command interface into the environment in which PX.DEVELOP is the logon group and there is no concept of current working directory. Hence, :listfile would display the files in the group PX.DEVELOP.

This example illustrates that using PXUTIL and the escape feature to the MPE command interface would have required enhancements to the CI commands so they could have understood POSIX names and the current working directory. We concluded that if all the CI commands were enhanced in such a manner there would have been no need for PXUTIL.

CI_POSIX Toggle. This alternative defined a command interpreter variable CI_POSIX, which, when set, would enable some of the MPE command interpreter commands to accept POSIX-named objects directly. The following sequence illustrates this idea:

```
:hello jeff.develop,px
:print daily_log        (try to use a POSIX file name directly
      ^                 in the MPE CI command PRINT)
Invalid Character In File Name. (CIERR 583)
:setvar CI_POSIX true   (tell the CI command to accept the
                        POSIX file names)
:print daily_log
```

The main problem with this idea was that with the variable CI_POSIX set to true, commands that were not enhanced would not recognize POSIX named objects. Thus, in this scheme if the :listfile command was not enhanced, the user would run into the same problem encountered with the PXUTIL utility. What is worse is that if the :listfile command were enhanced in the future to support POSIX names, the user would be surprised to see different results. This means that the same commands would have behaved differently when they were enhanced to recognize a variable like CI_POSIX.

It would have been nice to have an "accept POSIX names" switch on a per-command basis. In fact, the chosen design in which the leading characters . or / are used to escape to the POSIX name server does precisely that.

New Command Interpreter Commands. The last rejected idea was to create a new set of commands that would have directly accepted POSIX names. Consider the following scenario:

```
:hello jeff.develop,px
:chdir /users/jeff/
:pxlistf @
```

Presumably, pxlistf would be a new command that behaved like listf and :listfile and understood POSIX names and the CWD. This command would have displayed all the objects in the CWD (/users/jeff/) as opposed to listf and :listfile which would have displayed all the files in the logon group PX.DEVELOP. The biggest concern with this idea was that multiple commands would have been doing the same task.

File Access and Security

In the POSIX file access model a process can access a file in the following ways: read, write, and execute/search. Search access applies to a directory and execute access applies to an executable file like a program file or a shell script file. There is a subtle difference between read (as applied to a directory) and search access. If a process wants to open a directory and read the entries in it, the process needs read permission for that directory. But if a process wants to access a file, say, /users/jeff/addr_bk, the process needs search permission for all the directories in the path, namely, /, users, and jeff in this particular case.

The standard file access control mechanism of POSIX uses file permission bits, and every file in the POSIX directory structure has file permission bits associated with it. All directories in POSIX are just files of directory type. File permission bits contain the information about a file that is used, along with other information, to determine whether a process has read, write, or execute/search permission for that file. The bits are divided into three classes: owner, group, and other. In addition to the file permission bits there is a user identifier (UID) and a group identifier (GID) associated with every file. File permission bits are set at file creation time and can be changed by the chmod() function. These bits can be read by using the stat() or fstat() functions.

For access control, processes are classified as belonging to one of three access classes: file owner class, file group class, and file other class. A process is in the file owner class of a file if the effective UID of the process matches the UID of the file. A process is in the file group class of a file if the process is not in the file owner class and if the effective GID or one of the supplementary GIDs of the process matches the GID associated with the file. A specific implementation of POSIX may define additional members of the file group class. Lastly, a process is in the file other class if the process is not in the file owner class or file group class.

Implementations of POSIX may also provide additional or alternate file access control mechanisms. An additional access control mechanism can only further restrict the access permissions defined by the file permission bits. An alternate access control mechanism, if enabled, is used instead of the standard mechanism. The alternate access control mechanism has some constraints, the chief being that it must be enabled only by explicit user action on a per-file basis. Lastly, POSIX also allows privilege-based security in which access may be granted to a process if it has the appropriate privilege. Each POSIX implementation can define what constitutes an appropriate privilege.

The MPE access control scheme is based on several mechanisms such as a file access matrix, lockwords, and access control definitions (ACDs). Implementing POSIX 1003.1 on MPE required a mechanism that conformed to the POSIX 1003.1 standard. The existing MPE access control mechanisms did not satisfy the requirements specified in the standard. MPE file user types defined for the file access matrix are not exclusive categories and MPE XL 3.0 ACDs cannot express access permissions for a file's owner or group.

To support the POSIX 1003.1 access control mechanism, the architecture team considered either extending an existing MPE mechanism or developing a new mechanism. It is generally preferable to extend an existing MPE mechanism since this approach often minimizes customers' training costs and HP's development costs. These benefits are maximized when the modification is a logical feature extension.

The architecture team noticed the close similarity between the evaluation of ACDs and the POSIX 1003.1 file permission bits. In both implementations access control evaluation progresses from the most specific entry or class† to the most general entry or class. A process can match only a single entry or class because entries and classes are exclusive. The two access control schemes are also similar in the way they store access permissions locally with the file object. These observations led to the design decision to implement POSIX security using MPE ACDs.

In MPE/iX, POSIX 1003.1 file permission bits have not been implemented as a separate access control mechanism. Instead, POSIX 1003.1 functions support the file permission bits via the MPE ACD mechanism. ACDs themselves have been enhanced to enable the ACD mechanism to operate as a POSIX 1003.1 additional access control mechanism and to provide directory access control. Fig. 8 illustrates these access control relationships. The translation block performs the conversion from file permission bits to ACD format and vice versa.

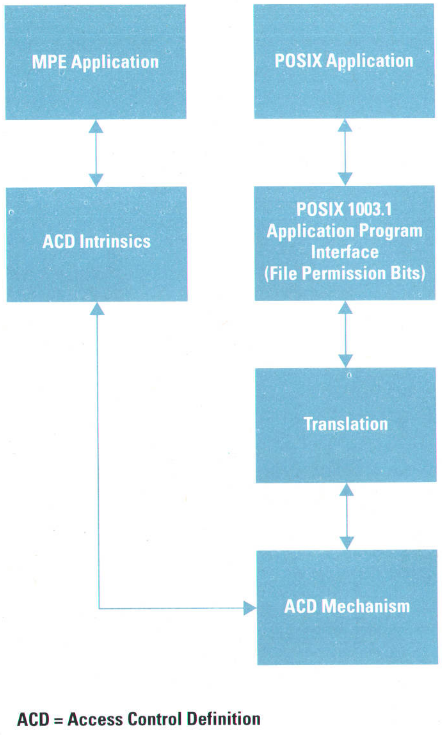

MPE Application
POSIX Application
ACD Intrinsics
POSIX 1003.1 Application Program Interface (File Permission Bits)
Translation
ACD Mechanism
ACD = Access Control Definition

Fig. 8. Access control implementation in MPE/iX. Note that the ACD mechanism is the foundation for both the MPE ACD intrinsics and the POSIX file permission bits.

POSIX 1003.1 applications will continue to use POSIX file permission bits to specify access permissions and will be unaware that file permission bits are implemented on top of the ACD mechanism. On the other hand, MPE applications will never deal with POSIX file permission bits; they will deal with ACDs, the file access matrix, and lockwords. When a POSIX 1003.1 interface such as open() creates a file, an ACD will be assigned to the file as part of the file creation operation. When the POSIX 1003.1 function chmod() is invoked to set access permissions for a file or directory, ACD information will be manipulated. Similarly, the stat() and fstat() functions will evaluate an ACD and map the access permissions granted by the ACD into file permission bits using this mapping in reverse. Fig. 9 illustrates the mapping between file permission bits and MPE ACD entries.

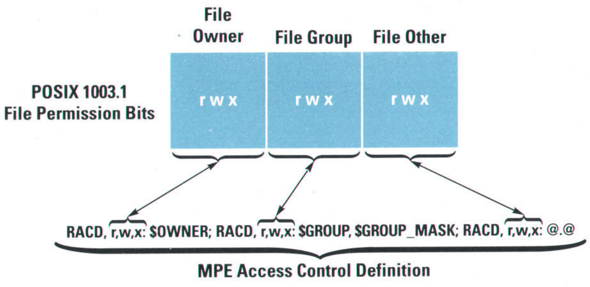

File Owner
File Group
File Other
POSIX 1003.1 File Permission Bits
r w x
r w x
r w x
RACD, r,w,x: $OWNER; RACD, r,w,x: $GROUP, $GROUP_MASK; RACD, r,w,x: @.@
MPE Access Control Definition

Fig. 9. Mapping between the POSIX 1003.1 file permission bits and the underlying ACD mechanism.

† Entry refers to an entry in the ACD such as r, w, x:@.@ in Fig. 9, and class is one of the three classes a process can be in: file owner class, file group class, or file other class.

Conclusion

When the MPE XL operating system was being designed for the new PA-RISC machines, backward compatibility with HP 3000 machines running the MPE V operating system was a major goal. This resulted in the design and implementation of the compatibility mode on the new HP 3000 Series 900 machines. Implementing POSIX on the HP 3000 Series 900 machines presented at least as great a challenge to the architecture team. Maintaining backward compatibility while seamlessly integrating POSIX and MPE concepts was one of the chief objectives of the architecture team. This paper has shown how this objective was achieved in the technical areas of directory structure, file naming, and file access and security.

Acknowledgments

I would like to thank the members of the architecture team and many other engineers who worked on implementing POSIX on the HP 3000 Series 900 machines. The core members of the architecture team were Steve Elmer, Craig Fairchild, Brian O'Connor, Mike Paivinen, and Jeff Vance. I would also like to thank my managers Bruna Byrne and Ann Stein for their support despite a very busy schedule during the POSIX implementation. Finally, I would like to thank Craig Fairchild, Mike Paivinen and Jeff Vance for reviewing this paper and providing valuable feedback.

HP-UX is based on and is compatible with UNIX System Laboratories' UNIX* operating system. It also complies with X/Open's* XPG3, POSIX 1003.1 and SVID2 interface specifications.

A Process for Preventing Software Hazards

Preventing software hazards in safety-critical medical instrumentation requires a process that identifies potential hazards early and tracks them throughout the entire development process.

by Brian Connolly

Since the occurrence of several patient injuries related to software failures of medical instrumentation,[1] much effort has been put into finding ways to prevent these software hazards in systems designed for medical use. Software hazards are a special category of software defects. If they occur during the operation of a system they may cause grave danger to human life and property. Software by itself does not harm anyone, but the instruments it controls and the information it collects from those instruments can cause damage. Therefore, since accidents in complex computer-controlled systems involve hardware, software, and human failures, software procedures to avoid hazards must be considered as part of overall system safety.

Many methods of analysis, prevention, and verification have been proposed to handle software hazards. HP's Medical Systems (MSY) Unit has researched and experimented with some of these methods and processes. This paper describes how we combined the most appropriate elements of these methods to develop a software hazard avoidance process for our organization. We will also show how the process was applied to one product.

MSY develops and manufactures instruments that provide clinical practitioners with patient information at bedside, a central nursing station, a hospital information system, a doctor's office, or anywhere the data is needed. Fig. 1 shows the layout for a typical high-level medical information and monitoring system. The information comes from transducers connected to a patient. Physiologic electromechanical activity is converted to analog electric signals. These signals are routed to data acquisition subsystems or modules of a complete patient monitor. The patient monitor electrically isolates the patient and digitizes the signals. Up to 12 modules can be connected to the patient. The digital data is moved to the monitor software subsystems where it is formatted, prioritized, and queued for display on the local patient monitor screen. In addition to local display, there is a proprietary local network, called SDN (serial data network), which is the pathway and protocol for displaying selected data from up to 24 bedside monitors on central reporting stations.

The patient data, whether at a bedside monitor or a central station, is used by the clinician as one element in patient treatment. Therefore, in analyzing the hazards in a medical instrument, consideration must be given to a direct hazard such as a software condition that produces an unsafe situation for the patient, and the possibility of patient mistreatment as a result of a monitor or central station indication

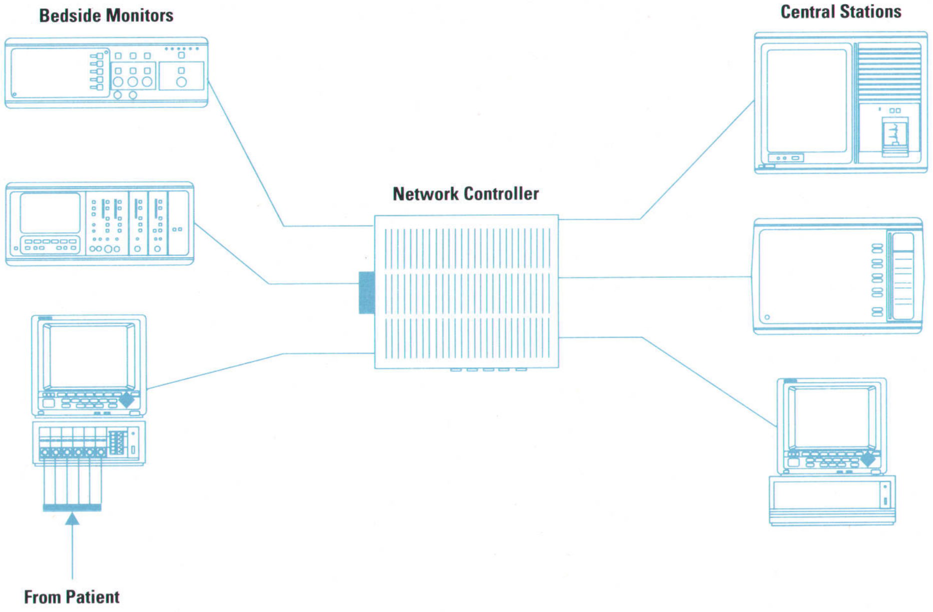


Fig. 1. A typical high-level medical information and monitoring system.

providing inaccurate trend data or failing to call a patient alarm. Preventing these problems was the motivating factor for the software quality engineering group at MSY to investigate software hazards and their avoidance in medical monitoring and reporting systems.

The Hazard Avoidance Process

Our hazard avoidance process is a combination of refinements to our existing verification methodology, which focuses on testing for hazards, and hazard avoidance analysis, which focuses on prevention.

Verification

When our investigation of software hazards began, the primary focus of the group was testing and verification of integrated embedded microprocessor-based patient monitoring systems and central reporting stations. Initially we exploited our experience in testing for hazards to develop a verification methodology.[2] This methodology was integrated into the normal product test development and test execution phases of product development.

The approach, as it is currently used, involves documentation of the test strategy and the creation of a software hazard avoidance fault tree, which shows the step-by-step verification of safety-critical subsystems. A portion of a software hazard avoidance fault tree is shown in Fig. 2.

Generation of a software hazard avoidance fault tree starts with identification of the most critical system hazards. There are many methods for identifying these particular hazards. The method we use involves investigating data on existing products and discussions with internal and external experts. The data collected consists of HP customer complaints, reports from the government regulatory agencies, journal articles, and internal HP defect data. External experts typically include clinicians such as nurses, technicians, doctors, and biomedical engineers. Internal experts include MSY engineers and marketing product managers. The experts use the data and high-level descriptions of the proposed product to determine the highest-level hazards to be avoided in the new system implementation.

The fault tree format provides a hierarchical decomposition of the areas of concern. At the lowest level of the tree, the "leaves" are tests that must be passed to establish safety for the lowest subsystem level. For example, tests 1.1.1.2.1 through 1.1.1.2.3 in Fig. 2 must be passed to ensure that the patient monitor is able to detect a failure in updating patient data in a specified time period. (Note that a failure in any one of these lower-level tests is analogous to a logical one to the next-higher level in the fault tree.) If any test fails, it is a simple exercise to evaluate the effect on the system. This is a valuable feature when considering releasing a system for clinical or beta testing and in making the trade-offs during development about whether to remove or correct an offending subsystem. The software hazard avoidance fault tree provides a summary of the plan for verification of hazard avoidance of the system and a way to ensure that test cases exist for verifying that the systems software safety goals are addressed.

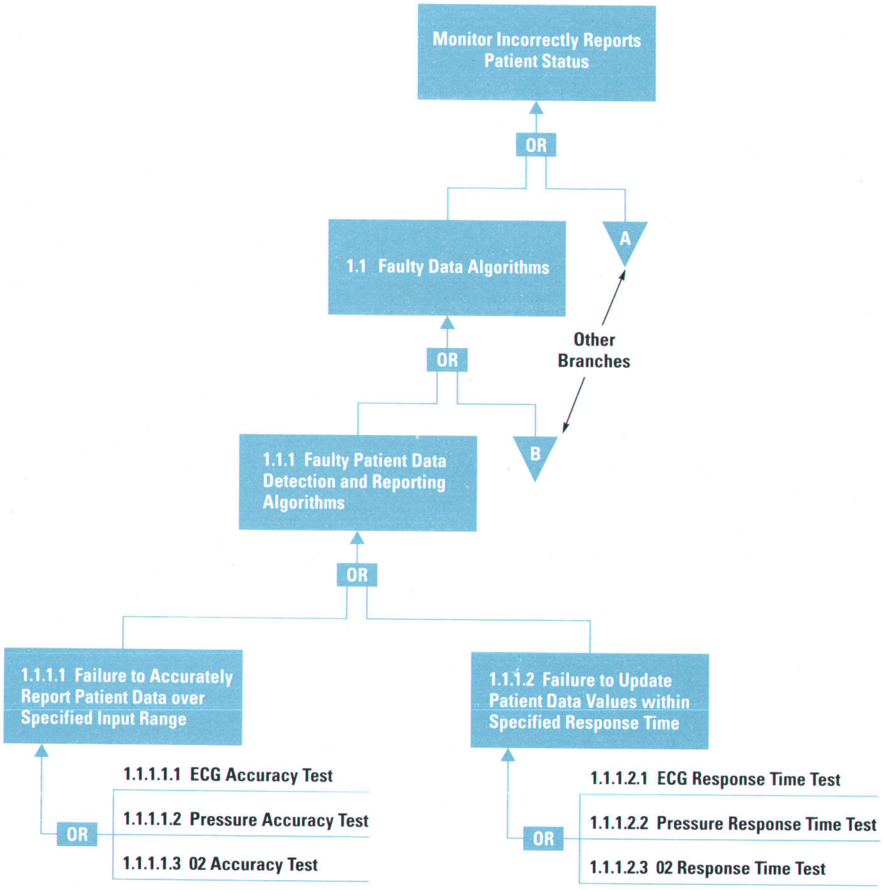

Monitor Incorrectly Reports Patient Status
OR
A
1.1 Faulty Data Algorithms
Other Branches
OR
B
1.1.1 Faulty Patient Data Detection and Reporting Algorithms
OR
1.1.1.1 Failure to Accurately Report Patient Data over Specified Input Range
1.1.1.2 Failure to Update Patient Data Values within Specified Response Time
1.1.1.1.1 ECG Accuracy Test
1.1.1.1.2 Pressure Accuracy Test
1.1.1.1.3 02 Accuracy Test
OR
1.1.1.2.1 ECG Response Time Test
1.1.1.2.2 Pressure Response Time Test
1.1.1.2.3 02 Response Time Test
OR

Fig. 2. A portion of a software hazard fault tree for a patient monitor.

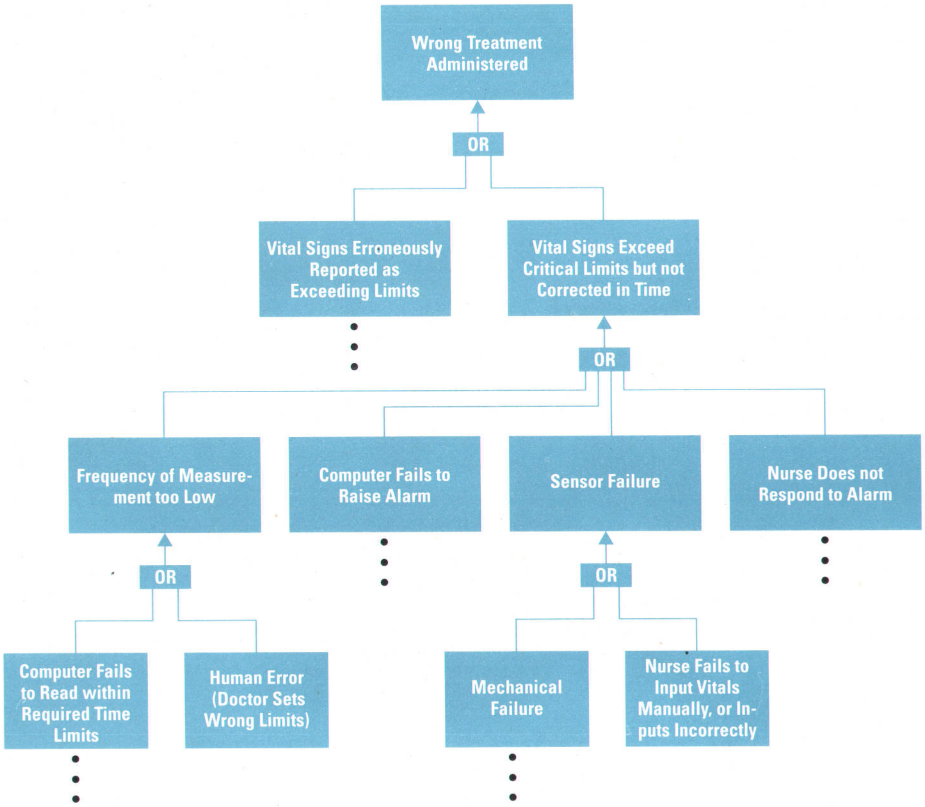

Wrong Treatment Administered
OR
Vital Signs Erroneously Reported as Exceeding Limits
Vital Signs Exceed Critical Limits but not Corrected in Time
OR
Frequency of Measurement too Low
Computer Fails to Raise Alarm
Sensor Failure
Nurse Does not Respond to Alarm
OR
OR
Computer Fails to Read within Required Time Limits
Human Error (Doctor Sets Wrong Limits)
Mechanical Failure
Nurse Fails to Input Vitals Manually, or Inputs Incorrectly

Fig. 3. An example of fault tree analysis.

At MSY, hazard avoidance tests are used to verify the safe operation of the system because when the system is partially implemented, it is used in a limited basis in clinical trials. Clinical trials provide valuable feedback from our customers in the early phases of implementation when we can easily make changes.

Since the hazard avoidance fault tree exists in a graphical representation it provides developers and government regulators with a clear map of the verification strategy for a product and its subsystems.

Avoidance

The software hazard avoidance fault tree and its associated tests provide a basis for understanding the compliance of a system with high-level safety objectives. As is the case for typical software defects, verification activities during the development process focus on how to either find defects earlier or ensure that they are not included in the first place.

For finding defects, a whole industry provides tools for low-level, or white box testing. Instead of manual testing at a system level, commercially available tools can enable the developer to test individual software components in an automated fashion. This is more comprehensive than finding defects in the system test cycle, which occurs at the end of the development process. The focus is still on defect removal, rather than prevention. In the prevention case, tools such as formal design reviews are used on design and specification documents, and formal inspections are used on code. These prevention techniques are used from the earliest user requirements phase through the code phase. Since hazardous defects are a subset of all software defects, prevention techniques are needed to complement the verification process described above.

For our hazard prevention program, we investigated processes and analysis techniques that would fit into our product development process. The IEEE draft document that defines a standard for software safety plans[3] recommends the following four documents for safety-critical software:

- Preliminary hazard analysis report. This report should document the results of looking for potential hazards from the initial system design documentation.
- Software safety requirement analysis report. This report should document:
  - The list of hazards, their criticality level, and relevant associated software requirements
  - Software safety design requirements and guidelines
  - Safety-related test requirements.
- Software safety design analysis report. This report may be divided into two parts. The first part addresses the safety analysis of the system's preliminary design, and the second part addresses the safety analysis of the system's detailed design.
- Software safety code analysis report. This report should document:
  - The rationale for and types of analyses performed on each module

Area of Concern	Failure Mode	Failure Cause	Failure Effect on System	Risk	Recommendations
Alarm Priority	High-Priority Alarm Masked by Instrument Status Message	Alarm Priority Scheme Nonstandard	Incorrect Alarm Sounding	High	Follow Scheme Outlined in XYZ Specification

Fig. 4. A part of a table used to document the results from a failure modes and effects analysis.

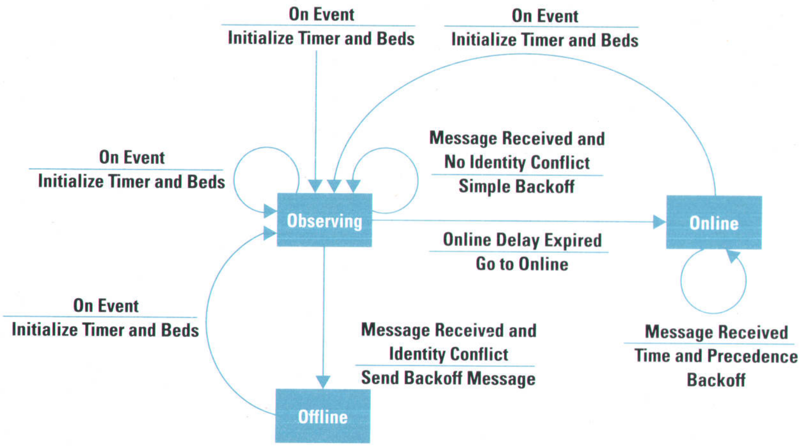


Fig. 5. An example of event tree analysis using a state machine approach. A backoff message tells the instrument to ignore any further messages or alarms.

- ○ Recommendations for design and coding changes
- ○ Detailed test recommendations
- ○ An evaluation of safety requirements.

The draft also lists several techniques for hazard analysis and avoidance. Some of these techniques include:

- Formal inspections. A formal method of peer review of target documentation (design documents, code, test plans, etc.) that culminates in a review meeting. Each member of the review team has a specific role in the review meeting. The objective of the inspection or review process meeting is to identify issues and defects, which are documented and addressed later by the reviewed document's author.[4]
- Fault tree analysis. A logic diagram of expected event sequences. It can be used to show how hardware, software, mechanical, and human interactions can cause a hazard (see Fig. 3).
- Petri nets. A diagramming technique that enables timing information to be incorporated into the safety analysis.[5] This technique helps to identify software events that are either too early or too late, thereby leading to hazard conditions. While fault tree analysis accents the effect of unexpected events in a logical linkage, the Petri net focuses concern on correct events occurring at possibly incorrect times.
- Failure modes and effects analysis. This involves an examination of all the failure modes in a system with a description of the cause and effect associated with the failure mode. Failure modes and effects analysis results can be documented in a table (see Fig. 4).
- Event tree analysis. This analysis moves events logically through the system to determine the consequences of these events. An event tree analysis is different from a fault tree analysis in that a fault tree traces an undesired event to its causes. Fig. 5 shows an event tree analysis using a state machine approach.
- Formal specifications. A rigorous mathematical method of defining a system, with rules and definitions that provide the basis of proof.[6,7]

Other techniques proposed in the IEEE standard include performance analysis, sneak circuit analysis, criticality analysis, and fault tolerant testing.

The form we chose for our hazard avoidance plan includes modified versions of the documents defined in the IEEE standard, some of the analysis techniques described above, and the hazard verification process described in the previous section. Like the IEEE standard, our plan allows the analysis format to be adjusted to fit the problem. The adjustments are made according to the experience of the the software quality engineer and the type of application being considered. For instance, most of our analysis is in the form of formal inspections, but an engineer may decide that instead of using a state machine to model some function, Petri net analysis is more applicable. Also like the IEEE standard, the emphasis is on analysis and documentation during the software development phases.

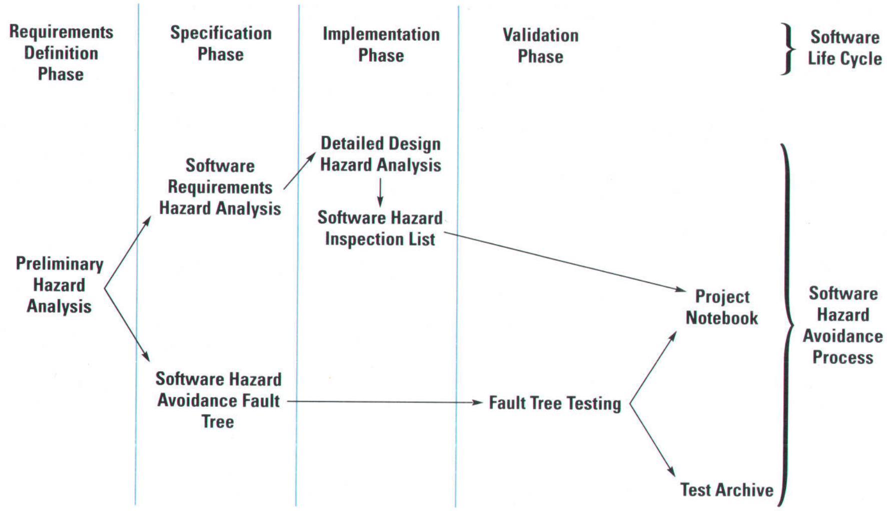


Fig. 6. Software hazard avoidance process flow in the software life cycle.

Inputs	Hazard Avoidance Process Step	Deliverables
Software Architecture and Other Evaluation Documents	Preliminary Hazard Analysis	List of Potentially Hazardous Areas
Preliminary Hazard List, High-Level Software Design	Software Requirements Hazard Analysis	Recommendations for Design Changes in Areas Defined by Preliminary Hazard Analysis
Preliminary Hazard List, Low-Level Software Design	Detailed Design Hazard Analysis	Recommendations for Detailed Design Changes in Areas Defined by Preliminary Hazard Analysis
Preliminary Hazard List and Code	Software Hazard Inspection List	List of Code Functions Requiring Formal Inspection in Hazardous Areas
External Specifications	Software Hazard Avoidance Fault Tree	Validation Tests for a "Safe" System Documented in Fault-Tree Format

Fig. 7. Inputs and deliverables associated with each of the steps in the hazard avoidance process.

The hazard avoidance activities that take place during the software development phases are shown in Fig. 6. The inputs and deliverables to the hazard avoidance activities are summarized in Fig. 7. Note that although the names of the documents may differ slightly from the IEEE-recommended documents, the contents remain the same. The one exception is the software hazard inspection list which corresponds to the IEEE-specified software safety code report. We provide the same contents as the IEEE report in terms of results, but we are more specific in mentioning the type of analysis used for hazard avoidance. The result is that no matter which form of analysis is chosen in any development phase, the process documentation contains the choice along with the results for each phase. The overall benefit is the capability to follow the hazard and its treatment through each development phase.

Analysis and Verification Together

As shown in Fig. 6, hazard analysis begins in the requirements phase of a project. When a project establishes user requirements and has identified a high-level architecture for a product to fulfill the requirements, work begins on the *preliminary hazard analysis.*

With the early documents completed, data from similar previously released products such as enhancement requests, defects, and government regulators' reported complaints are combined together for analysis. An evaluation team consisting of users, quality engineers, developers, internal regulatory personnel, and marketing engineers reviews the collected data and proposed product architecture, and with each member's experience base, decides on the levels of concern for hazards identified in the new product.

The hazards are rated minor, moderate, or major based on the consensus of the group. Each hazard is evaluated to determine its impact on patient care if the hazard were not removed. These hazardous areas provide the focus for the quality engineer's future analysis during development and verification. To complete the preliminary hazard analysis, each hazard has associated causes and methods of verification listed. Each cause indicates how a problem might occur in the given architecture, and the verification list describes the methods for making sure the hazard isn't included in the product. Fig. 8 shows a portion of a preliminary hazard report containing the information mentioned above.

During the specification phase of product development, the *software requirements hazard analysis*, which shows results of analyzing the software requirements, is produced. The requirements are provided in many forms such as data flow diagrams, verbal descriptions, entity relationship diagrams, and formal specifications depending on the form of software being developed. The areas identified as hazards in the requirements are analyzed according to whatever method was defined in the preliminary hazard analysis.

In addition to performing a hazard analysis of the requirements during the specification phase, a *software hazard avoidance fault tree* is also generated from the data provided in the preliminary hazard analysis phase.

In the implementation phase, a *detailed design hazard analysis* is performed to examine the same areas studied in the previous phase, except this time the details of the design and implementation are examined. Following the detailed design hazard analysis, the *software hazard inspection list* is created. This list is a collection of the software functions requiring inspection in the hazardous areas of the design. We require inspections of all software functions that have a cyclomatic complexity of 10 or greater.

The portion of a hazard analysis matrix shown in Fig. 9 provides an overview of the hazard avoidance history for two hazards identified in the preliminary analysis phase.

Once all the analyses and tests have been completed, the documentation is kept in the project notebook and the software test archives for the product. The notebook and archives are kept as evidence of process adherence required by industry and government regulatory agencies.

Software/Hardware Hazard 3: Error Reporting Failure

Cause(s): **Design Implementation Failure**

Verification: **Inspection with participation of SQE to verify software design for hazard avoidance. Use software hazard avoidance fault tree tests to verify design and implementation of error handling and reporting to correctly identify errors and their sources.**

Level of Concern: **Minor. Severe errors cause failsafe inoperability of the system, posing inconvenience to staff users.**

Software/Hardware Hazard 4: SDN Backoff Failure

Cause(s): **Design/Implementation Failure**

Verification: **Inspection with participation of SQE to verify software design for hazard avoidance. Use software hazard avoidance fault tree tests to verify design and implementation to avoid conflicts and incorrect behavior in backoff determination.**

Level of Concern: **Moderate. A confusion could occur between central stations as to "ownership" of bedside information. This could result in misdirection of patient information, alarms, and recordings if the backoff algorithm is incorrectly implemented.**

***SQE = Software Quality Engineering**

Fig. 8. A portion of a preliminary hazard analysis report.

Preliminary Hazards	Software Requirements Hazard Analysis	Recommendations	Fixed	Detail Design Hazard Analysis Findings	Recommendations	Fixed	Software Hazard Inspection List (Function Names)	Software Hazard Avoidance Fault Tree Reference (Test Names)
1. Date and Time Failure	1.a. Time Sources Unclear	1.a. Clarify	y	1.a. Section 2.3 – Consider what happens when no time table. b. Section 3.3.1 – When rtc_sync_count reaches 0, an error should be logged, sourcing time is not valid.	1.a. Date and time should source time when no time available. b. Log error, do not source time because rtc_sync_count=0 could be a system module error.	y y	No code inspections required.	Date/time Do all tests in SDN system subset (see fault tree).
2. SDN Backoff Failure	2.a. 5.4.1.1, 5, "... internal beds in consulting," avoids backoff question. How do beds get into consulting?	2.a. Clarify		2.a. See the document "SDN Backoff Mechanism A Hazard Analysis using HP-SL," attached to this report.	2.a. See the document "SDN Backoff Mechanism A Hazard Analysis using HP-SL," attached to this report.	y	init1 initHdrTables init_genHdr	Backoff Do all tests in SDN system subset (see fault tree).

Fig. 9. Hazard avoidance history for two software hazards as they are analyzed and verified during the hazard avoidance process.

Results

The software hazard avoidance fault tree portion of the hazard avoidance process has been used in eight MSY products and product enhancements since 1989. No safety defects requiring instrument recalls have been reported for these products. In one product line, hazard avoidance testing discovered 32% (as compared to 15% in previous projects that used traditional development and testing processes) of all serious and critical defects that would have had an effect on patient welfare.

As of writing of this article, one product has used and completed the full hazard avoidance process described in this paper. Several other projects are in different stages of using the process. In the project that has completed the process, two forms of analysis were used during the preliminary hazard avoidance phase: formal inspections and formal methods. The bulk of the hazards were found using formal inspections. A total of 23 inspections were performed on product documents, finding 12% of all hazardous defects.

Using HP-SL[7] (HP Specification Language) two areas of potential hazards were specified and examined. Four possible hazardous defects were identified with formal methods.

Conclusion

The MSY development and implementation of the hazard avoidance process has helped provide products with fewer recalls, providing a higher level of customer satisfaction. The results achieved so far with this process have been excellent. The emphasis on defect prevention will continue to pervade our development effort, with the hazard avoidance process as the cornerstone.

Defect prevention and analysis are not enough, and this process provides a verification trace back to the requirements phase of development. Also this process provides a system view of the objectives set in the early phases of a project to ensure their correct implementation.

From our experience so far, the cost of using this process depends on the form of analysis chosen. The form of analysis has to be carefully considered when a potential hazard is discovered. The types of analysis and the applicability to the problem must be investigated. The cost/benefit analysis may have to consider such questions as the cost of a potential recall, the possible reuse of functional elements in another product, and always, the effect on the patient and user.

No matter what form of analysis is chosen, the process steps have been standardized and documented and are available for government regulators, other developers, and any other interested parties to examine. This process provides the evidence that we have done all we can do to prevent software hazards before building the instrument.

References

1. E. J. Joyce, "Software Bugs: A Matter of Life and Liability," *Datamation*, May 1987.
2. B. Connolly, "Hazard Avoidance in Patient Monitors," *HP Software Engineering Productivity Conference Proceedings*, August 1990.
3. *Standard for Software Safety Plans*, Technical Committee on Software Engineering of the IEEE Computer Society, Preliminary Draft, March 1991.
4. M. E. Fagan, "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, Vol. SE-12, no. 7, July 1986, pp. 744-751.
5. N. Leveson and J. Stolzy, "Safety Analysis Using Petri Nets," *IEEE Transactions on Software Engineering*, Vol. SE-13, no. 3, March 1987, pp. 386-397.
6. T. Ferguson and T. Rush, *SDN Backoff Mechanism: A Hazard Analysis Using HP-SL*, HP Technical Memo, June 1991.
7. S. Bear and T. Rush, "Rigorous Software Engineering: A Method of Preventing Software Defects," *Hewlett-Packard Journal*, Vol. 42, no. 5, December 1991, pp. 24-31.

Configuration Management for Software Tests

To support software test reuse and to make it easier to ensure that the correct software versions are used to test printer products, a software test management system has been put in place.

by Leonard T. Schroath

Many software development organizations have begun to formalize software reuse as a way to improve productivity and increase quality. However, most of the effort is put into reusable components that are used for creating software products. Methodologies and processes can exist for the test development effort as well. If software components can be reused effectively, test components can be reused also. For any reuse program to be successful, a formal process and support tools are essential.

The software quality department that serves HP's Boise Printer Division and Network Printer Division maintains a vast printer test library, which includes performance tests, conformance tests, other black-box tests, test procedures, test results, test documentation, and known-good output for comparison. The test library requires extensive scripts to extract and execute tests for various projects. Many new tests are leveraged from old tests, but there is only a primitive browsing mechanism to aid in locating them.

Fig. 1 shows the steps and databases involved in submitting, reviewing, and executing tests in our existing test process. Except for test execution, the processes shown in Fig. 1 are scripts that are primarily responsible for moving test data* from one database to another. Each database represents a different state in the test development process.

The process begins with test data being moved from the lab to the submit database. Next the test data is reviewed to see if it is complete. For example, the test program is checked to make sure it compiles. If anything is wrong with the test data, it is sent back to the lab for correction, and if everything is okay the test data is moved to the trialrun database. During the trial run phase, the test program is executed to flush out any problems. If there are problems, the test program is sent back to the lab for repair. If all goes well, the test program is moved to the testlib database. Once a test is

* Besides the test program, test data may also include items such as test documentation and include files.

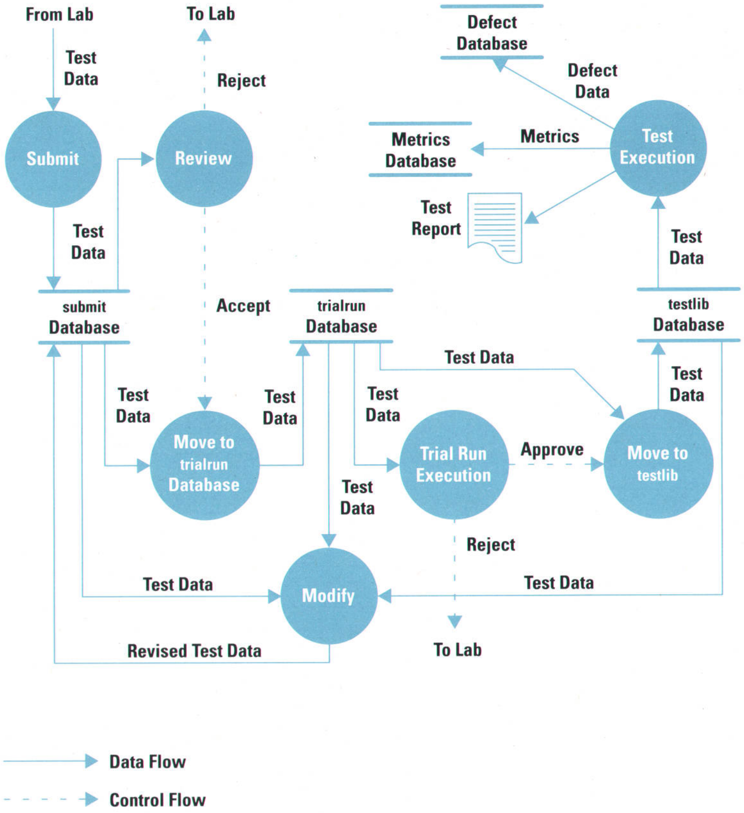


Fig. 1. The existing test process without a test management system.

in the testlib database, it is ready for formal test execution. Test execution involves not only testing the product, but also capturing data for various databases and generating a test report. At any point in this process modifications can be made to the test data, resulting in moving data from the database (state) it is in back to the submit database.

To improve this process with tools that manage test selection and assist in test development for the various software and firmware projects at our divisions, a test library management system, or TLMS, has been developed. This new system is gradually being phased in to enhance and replace parts of our existing test system. The rest of this paper describes the development, features, and test life cycle of the test library management system.

Goals

Several goals were established early in the TLMS development process. First, there was no doubt that a need existed to track all tests under a version control system so that previous versions of tests could be recovered and executed if the firmware or software was revised to fix defects or add enhancements. It was also desirable to be able to track test versions with firmware or software versions so that customized test suites could be created to test specific features of a particular version of firmware or software. TLMS also needed to ensure clear ownership for each version of a test so that there was only one individual who was responsible for any and all changes made to a particular version. The entire maintenance process needed to be more formally defined and automated as much as possible. Another important goal was to facilitate test reuse through a test case classification scheme and a test location mechanism. This scheme would aid in test case selection and test suite development.

Configuration Management Tool

TLMS is based on an object-oriented configuration management tool called CaseWare/CM,[1] which provides a structured development environment with a turnkey model that can be used for most software development projects. This configuration management tool also provides both a graphical OSF/Motif-based user interface and a command-line user interface, and it runs on HP 9000 Series 300 and 700 workstations.

CaseWare/CM's built-in flexibility allowed us to create a customized model to fit our test development life cycle. As user needs change, the model can be modified without impacting the objects stored in the database.

To help with our reuse efforts, CaseWare/CM records the development history of the test library as tests are created, modified, or imported. The test library configuration is standardized across projects so that test developers or test consultants who are browsing or searching for reusable tests can easily find them. Tests and test suites are treated as objects that can easily be included in other test suites.

Features of TLMS

Components and Attributes

Most of the operations of TLMS center around components. A component is a related set of attributes that describe an object such as a test program. Each component has a number of attributes including its name, type, status, source, description, author, subsystem, creation date, and date of last modification. A component also has a version attribute. Specific versions or instances of objects are referred to as *component versions.*

Different types of components exist for different types of objects. Objects in our case are test programs, include files, spreadsheets, documents, shell scripts, C source files, and font files. Some component types are built into the configuration tool, and others can be created and added to the model as needed. Specific behavior is given to each component type, such as how the source attribute is compiled or edited. Table I lists some of the built-in types and some of the types that have been created specifically for TLMS.

Table I
Component Types

Built-in Types	Description
csrc	C source
incl	C include
lsrc	lex source
ysrc	yacc source
shsrc	Shell source
C++	C++ source
ascii	ASCII text
swasm	Software assembly
TLMS Customized Types	
pmac	Printer macro language source (main test program)
p_incl	pmac include file
post	PostScript® source
msword	Microsoft® Word document
lotus	Lotus® 1-2-3® spreadsheet
excel	Microsoft Excel spreadsheet
font	Soft font file
testasm	Individual test assembly
suite	Test suite assembly

The naming convention used to designate a component version has four parts, which helps to avoid ambiguity. Each part is separated by a slash (/). This slash serves only to delineate the four parts of the component version name and has nothing to do with a file system hierarchy. The four parts are: subsystem, type, name, and version. Thus, for a test with a subsystem of fonts, a type of pmac, a test name of fontpri4, and a version of 2 the four-part name is fonts/pmac/fontpri4/2. Each of the four parts, along with the state of the component version, is represented graphically in Fig. 2.

A special type of component version called an assembly is used to group other component versions together. In the context of TLMS, a test assembly consists of all component versions needed to run a test. A minimum test assembly must include a test program, a test procedure, and some expected

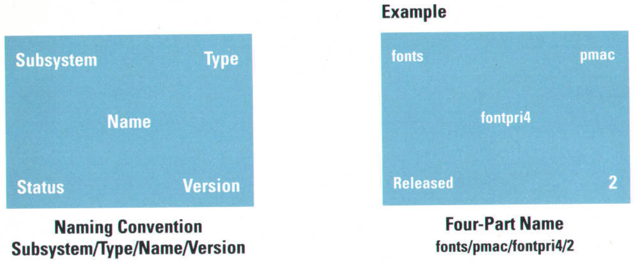


Fig. 2. A graphical representation of a component version.

results. An assembly is somewhat analogous to a directory in a file system. The graphical representation of an assembly displays an extra box around the name to distinguish it easily from regular component versions (see Fig. 3).

An assembly also has its own collection of attributes, including name, status, version, owner, and subsystem. An assembly may also have its own customized attributes. An example would be an installation directory attribute, which would serve as a definition of where to install source files in the file system once they have been extracted from TLMS.

TLMS uses two special assembly component versions which are customized for the test library application. The first is of type testasm (test assembly), which is the basic building block of test suites (see Fig. 3). There is no limit to the number of component versions that can be included in a test assembly. However, by TLMS convention there should be only one main test program per testasm, and as many include files as needed for the test. Each test assembly should also have some test design documentation describing what is being tested.

The other special assembly component version is of type suite. This assembly is used for grouping test assemblies into test suites. A test suite can consist of one or more testasms and one or more test suites. Although there is no requirement that a test suite consist of only suites and testasms, most other component version types should be part of a testasm and not a suite.

Component versions can easily be placed into an assembly by creating a list of components to be bound into the assembly. The operation reconfigure is performed, which automatically selects the specified component versions and binds them into the assembly. The list of components only needs to have the first three parts of the four-part name for each component, which causes the latest version of the component to be bound into the assembly. If a specific version of a component is desired, it may also be specified as the fourth part of the name in the list of components.

The reuse program being developed for tests in TLMS will make use of customized attributes found on each testasm. These attributes make up a faceted classification scheme,[2] in which the attributes form a tuple that describes a test. Queries to TLMS use the classifications to extract a set of tests requested by the tester. For example, a query might be to find all released tests for a specific printer that will exercise the font selection capability of the printer.

Test Suite Hierarchy

Components representing tests and test suites are grouped in a hierarchical form to facilitate test location. A catalog is maintained of reusable testasms (test assemblies) and suites, any of which can be easily bound into any test suite for any project. In this manner project-specific test suites can quickly be developed by combining new and existing tests.

The test library has at its highest level an assembly of type suite named testlib (see Fig. 4). This assembly consists of other assemblies of type suite. One is named catalog, which contains all of the tests that are used in more than one project. This is the basis of the reusable library. The catalog is hierarchically arranged into logical subassemblies, which makes it easy to find tests. Printer conformance tests and other black-box tests that can be used by more than one project are found in the catalog.

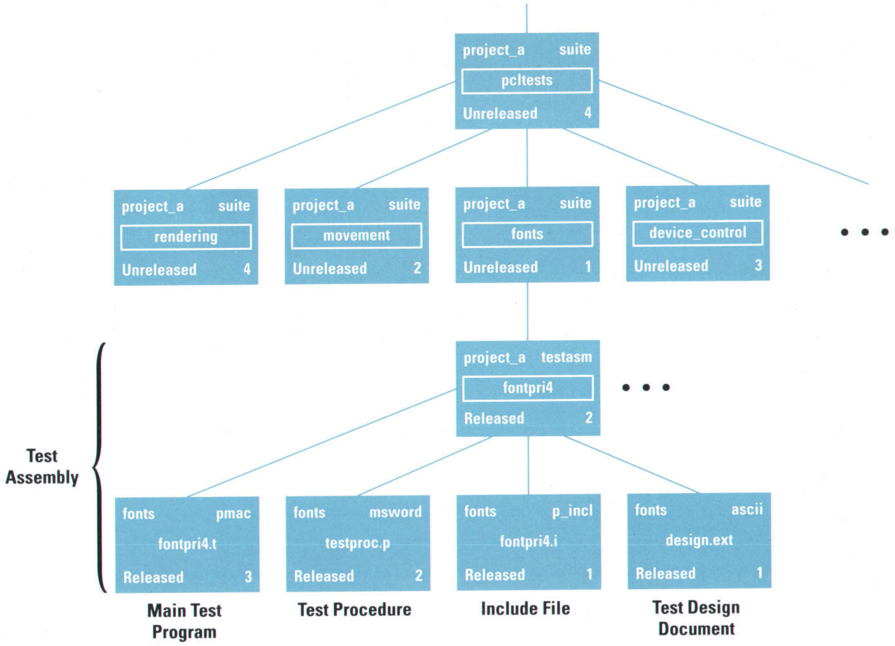


Fig. 3. A portion of a typical TLMS test assembly.

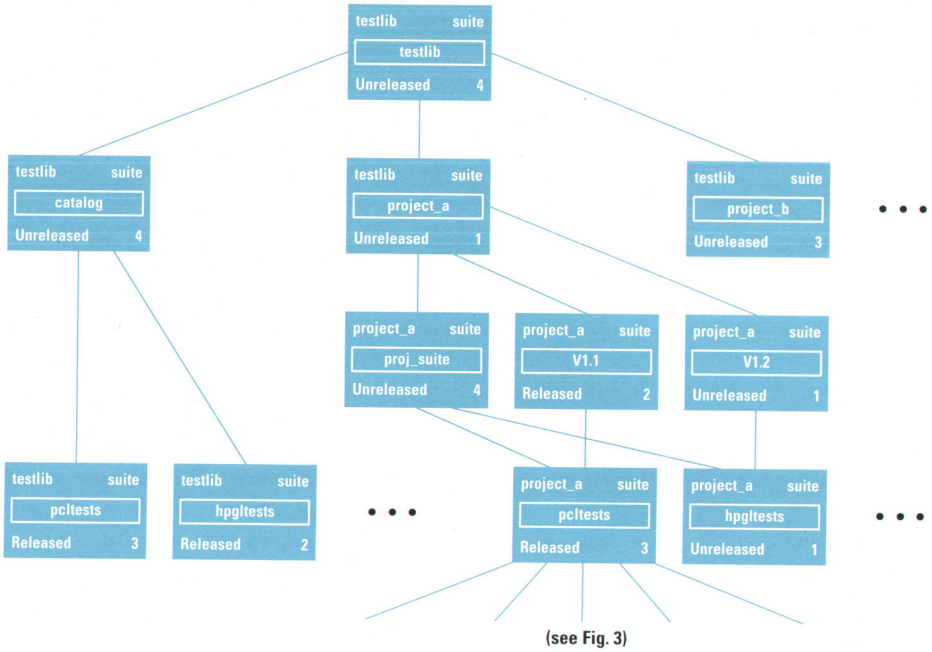

testlib suite testlib Unreleased 4
testlib suite catalog Unreleased 4
testlib suite project_a Unreleased 1
testlib suite project_b Unreleased 3
project_a suite proj_suite Unreleased 4
project_a suite V1.1 Released 2
project_a suite V1.2 Unreleased 1
testlib suite pcltests Released 3
testlib suite hpgltests Released 2
project_a suite pcltests Released 3
project_a suite hpgltests Unreleased 1
(see Fig. 3)

Fig. 4. A portion of a TLMS test suite hierarchy.

Other test suite assemblies bound to the testlib are specific to projects and are named for the project (e.g., project_a or project_b). Each project assembly consists of a test suite named proj_suite, which contains all of the suites and testasms that make up the entire test suite for that project. A proj_suite may contain links to testasms or suites that are in the catalog, or may have its own special component versions tailored for the specific project. A project assembly may also contain one or more testasms or suites with a name that corresponds to a version of software or firmware that is being tested. For example, a suite may be developed to test certain features of version 1.2 of some firmware, and hence could be named V1.2.

The subsystem name for all assemblies for a specific project should match the name of the project, unless the entire suite or testasm is being used as it came from the catalog. This allows extraction or reporting on all component versions that are specific to the project. For example, in Fig. 3 the component versions with the subsystem name fonts are actually from the catalog.

Roles

Much of the activity in TLMS is allowed or disallowed based on the role of the person interacting with TLMS. Users are assigned one or more roles. Each role allows browsing through the test library, along with certain other privileges briefly described below.

Author. An author is allowed to create a new component version, derive (check out) a new component version from an existing component version, and install tests on the file system for execution. An author who has created or derived a component version becomes the owner of the component version until it is reviewed, approved, and released.

Test Librarian. The test librarian is responsible for the content and structure of the overall test library. The test librarian is the only one able to create or manipulate component versions of type suite. The test librarian also makes sure that each test is executed to verify that it works correctly before putting it into the test library. Finally, the test librarian is the only person who can officially release component versions into the test library.

Test Consultant. The test consultant is responsible for reviewing test component versions before submitting them to the test librarian for a trial test run. Test consultants are only able to modify component versions that belong to projects to which they have been assigned. Test consultants, in cooperation with R&D, define the component versions that belong in a test suite, but they are not allowed to create them.

Test Technician. The only activity that a test technician is able to perform in TLMS is a test-suite build, which, in the context of TLMS, extracts all test sources, procedures, masters, and anything else that is required to execute a test and places them in the file system. Tests can then be executed.

TLMS Administrator. The TLMS administrator is the superuser of the system and is responsible for adding new users, changing their roles, and making changes to the TLMS model installed in the test library database. The TLMS administrator is also able to modify any component version and generally circumvent the built-in security provided by the rules set up in the TLMS model. This role must be used with caution.

Software Development Engineer. This role is similar to the test technician role. An individual with this role has permission to browse through the test library and is able to perform a test-suite build to extract tests and place them in the file system for execution.

Guest. A guest only has permission to browse through TLMS. A user with this role cannot modify any component version, extract tests via a build operation, or perform any other operation. This role is provided for users to look at tests currently in the test library.

TLMS Life Cycle

The main life cycle for tests is governed by a series of states and transitions that affect any assembly of type testasm (test assembly) and the component versions that are bound to it.

A component version progresses through several states from the time it is checked out or created until it is released into the test library. Most transitions are made directly to component versions attached to a test assembly, and all component versions bound to the testasm make the transition automatically. Transitions can also be made directly to an individual component version. Fig. 5 shows the states and transitions for the TLMS main life cycle.

The initial state is called private. Most newly created test assemblies begin the life cycle with this state as their default status. A component version in this state can only be modified by the author (owner) who created it. A private component version can only be bound into testasms created by the same owner. A new version cannot be derived from a component version in this state because it is considered to be unstable.

The only transition allowed from the private state is when the testasm is ready to be published. Publishing a test assembly moves it from the private state to the working state. The tests are made public for the first time which means that someone else can bind these component versions into another testasm, and the librarian can bind a testasm into a suite. This transition is initiated by the owner of the test assembly.

The working state assumes that the tests bound into a testasm are in working condition. In other words, the test can be executed. It may not be completely correct, but at least it runs. The owner is still the only one who can modify a component version. New versions can be derived from any component version in the working state (or any subsequent state), but when this happens, the source attribute of the original component version is frozen. The test assembly containing a modified component version can continue to proceed along the life cycle, but all source modifications must be made to the new component version to avoid a double maintenance problem.

The only transition allowed from the working state is when the test assembly is ready to be reviewed by the test consultant. This transition moves the test assembly from the working state to the finalreview state. This transition is also initiated by the owner of the test assembly. Electronic mail is sent to notify the authorized test consultant that a testasm has been submitted for final review.

The finalreview state allows the test consultant to review the test for completeness. All tests should be complete and must contain all of the necessary documentation. Once a test assembly is in the finalreview state, it can no longer be modified by the owner. Only the proper test consultant has modification privileges.

Two transitions are allowed from the finalreview state. The first is the transition that moves the test assembly back to the working state for rework by the owner. A report of the problems encountered is sent to the owner by electronic mail. This report is required before the transition is made. In the graphical user interface, a window opens that allows the test consultant to enter the report, and in the command-line interface, the test consultant can attach an ASCII file containing the report.

The second transition allowed from the finalreview state moves component versions to the trialrun state. As test assemblies make this transition, electronic mail is sent to the test librarian indicating that a test is ready for a trial run. Electronic mail is also sent to the author acknowledging that the test has passed final review and is ready for a trial run. Both transitions from the finalreview state are initiated by the test consultant.

The trialrun state allows the test librarian to review a test for completeness and ensure that it executes properly. The test librarian also makes sure that the testasm has a unique installation directory so that the test can be installed onto the file system without overwriting other tests. The test librarian is the only individual who can modify any component version in the trialrun state. This is to prevent any changes to the component version without the test librarian knowing about them.

Two transitions are allowed from the trialrun state. The first transition moves component versions from the trialrun state back to the finalreview state if the test assembly is rejected. This transition is used for those tests that do not meet applicable standards or do not pass the trial run. A report of why the test was rejected is sent via electronic mail to both the test consultant and the owner (similar to the report sent when the test assembly is sent back to the working state for rework).

The second transition allowed from the trialrun state moves the test assembly to the released state. This transition is made only after a testasm has successfully completed a trial run and it meets all of the applicable standards. An acknowledgment is sent via electronic mail to the author, the test consultant, and others who are interested in knowing when a test is released. Both transitions from the trialrun state are initiated by the test librarian.

The final state is the released state. Any component version in this state must have been approved by the test librarian and by definition, be part of the official test library. A testasm cannot be released unless all of its member component versions are released. Component versions in this state are no

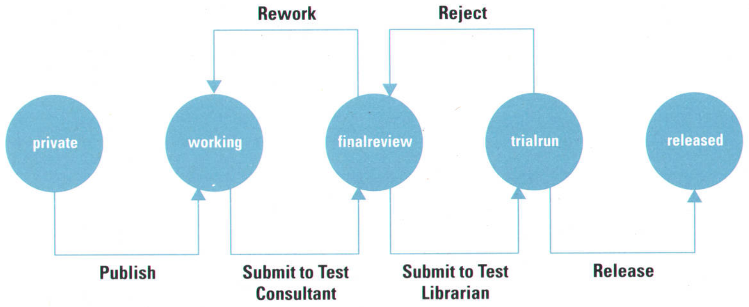


Fig. 5. The TLMS main life cycle. Test assemblies or individual component versions move through this life cycle.

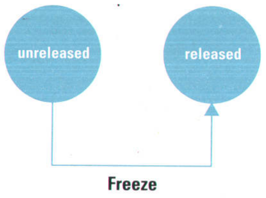


Fig. 6. The TLMS test suite life cycle.

longer modifiable by anyone. They are static, including all attributes. The only exception to this is a comment attribute, which can be updated at any time by the librarian if more information needs to be stored with a test.

A separate, shorter life cycle for component versions of type suite consists of two states: unreleased and released (see Fig. 6). This life cycle exists to facilitate the creation of large test suites without the overhead of having to move component versions of type suite through all five states of the main life cycle. Remember that component versions of type suite are typically made up of many test assemblies, and each test assembly is typically made up of a test program and many other files.

The initial state is called unreleased. Component versions of type suite in the unreleased state can be modified as often as necessary, but only by the test librarian. Transition from the unreleased state to the released state is allowed only when the test suite is determined to be frozen (no more changes). This transition can only be initiated by the test librarian. In the case of suites, all component versions bound to it must either be released already or released at the same time. This means that all children (test assemblies or suites) not already released must be in the trialrun state or the unreleased state (the only two states that allow transitions into the released state), or the transition will fail. The released state of this short life cycle is the same as the released state in the main life cycle.

Comparison

The TLMS main life cycle is intended to improve the test process described earlier and shown in Fig. 1. Overlaying the TLMS life cycle states shown in Fig. 5 with the processes shown in Fig. 1 results in the process diagram shown in Fig. 7. This diagram shows the following benefits to the test process with TLMS:

- Tests are maintained in one database, and there is no need to move files from one database to another to change state.
- Attribute information such as status, test owner, and creation date is easily stored and queried.
- Test development history is automatically recorded.
- The test maintenance process is easier to control.
- The test process is simpler.

Security and Access

Security rules for each model can be written and enforced to maintain the integrity of the database. These rules and privileges allow or disallow component version creation, modification, transition, and deletion based on the state of the component version or the role of the user. Security rules in TLMS enforce the privileges and operations described in the life cycle section.

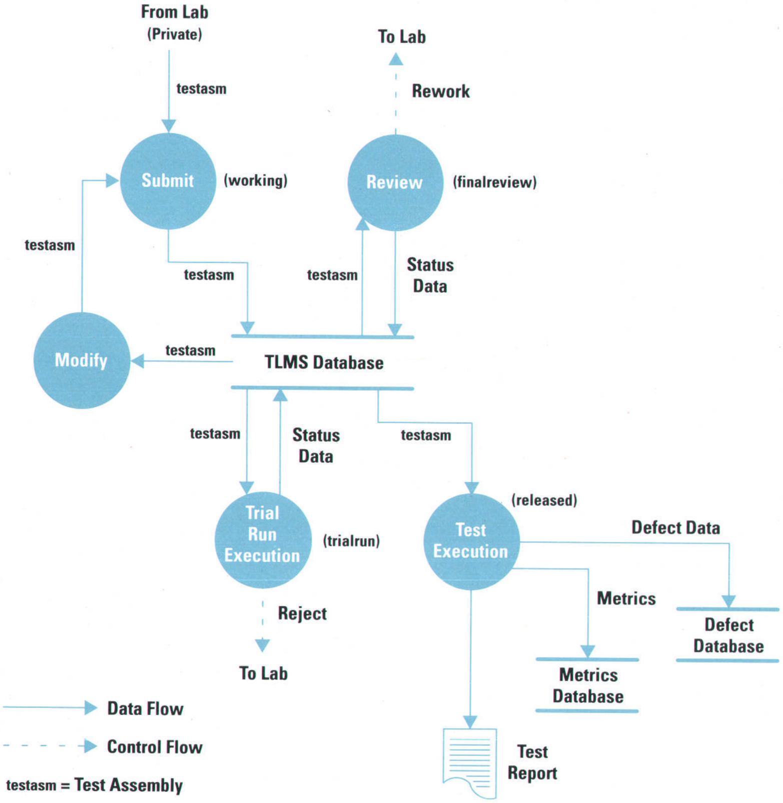


Fig. 7. The test process with TLMS in place. The items in parentheses are the states in the main TLMS life cycle shown in Fig. 5.

Subsystem	Type	Name	Version	Owner	Status	Last Mod
garbage	testasm	fontpri4	1	lts	trialrun	Thu Dec 3 22:07:50 1992
garbage	testasm	dfltfont	1	lts	finalreview	Fri Dec 4 13:16:38 1992
pt	testasm	pt	1	pault	working	Fri Dec 4 13:16:38 1992
tests	testasm	applelwp	1	amplify	trialrun	Thu Dec 3 16:49:12 1992
garbage	testasm	applelwp	1	lts	trialrun	Thu Dec 3 17:05:03 1992
catalog	testasm	applelwp	1	amplify	trialrun	Thu Dec 3 17:24:08 1992
catalog	testasm	fontpri4	1	amplify	private	Thu Dec 3 22:35:11 1992
catalog	testasm	fontpri4	2	amplify	private	Thu Dec 3 22:30:56 1992

Fig. 8. A portion of a TLMS test assembly report.

User Interfaces

Two user interfaces exist for TLMS. One is a graphical OSF/Motif-based interface that runs on the X Window System. Users can select actions from pop-up menus and navigate through the various windows using a mouse. The other interface is a command-line interface. Although a graphical interface is helpful in giving the user a visual picture of the elements in the test library, a command-line interface is also available to perform many simple commands. Among these are commands to create a component version, move an assembly of component versions from one state to another, modify the source attribute of a component version, install the test suite onto the file system for execution, and import an externally developed test into TLMS.

Reports

Reports of useful information are obtained from TLMS through some command-line queries. Some examples of reports that are easily obtainable are a list of all component versions in a test suite, the status of all component versions in an assembly, the authors of a list of component versions, a list of all projects using a given component version, and a list of all component versions currently in a given state. Fig. 8 shows part of a report of all testasms.

Benefits

TLMS provides an easy, consistent method of maintaining and accessing all tests. Those who need to interact with the test library can find what they want without having to ask an expert. Also, the entire process of test development and maintenance can be structured and controlled. The development history of tests in the test library is captured automatically. Tests can be identified and selected for a given test run, and a customized test suite can quickly be created. Tests that are not yet certified and released by the test librarian can still be executed if needed.

The software test center is able to increase testing efficiency by executing a test suite customized for a test run instead of the myriad of redundant tests that are present in the old test library. It is also much easier to verify that the proper versions of tests are executed for any version of firmware or software.

As with all programs, new enhancements and changes will probably be requested from the user community. TLMS features can be modified and extended without impact on the objects stored in the database. Finally, a more formal test reuse program can be established, which will allow test developers to take advantage of the work of others. This will in turn decrease the total test development time, a significant portion of a project's schedule.

Conclusion

The testing process is an important part of the software life cycle. Without the proper tools and structure, it can become inefficient and difficult to manage. With large volumes of tests for any given project, manual processes yield errors that can be avoided by applying configuration management practices to the test development and maintenance processes. Customizable tools such as CaseWare/CM are available to help control these processes. TLMS allows us to improve and automate test development and maintenance, as well as establish a formal test reuse program.

Acknowledgments

Special thanks to Fran McKain and Brian Hoffmann for their ideas and vision of how to manage tests more efficiently.

References

1. *CaseWare User's Guide*, CaseWare, Inc., 1992.
2. R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, Vol. 4, no. 1, January 1987, pp. 6-16.

Implementing and Sustaining a Software Inspection Program in an R&D Environment

Although software inspections have become a common practice in the software development process, introducing the inspection process and sustaining and measuring its success are still challenges.

by Jean M. MacLeod

There is not much disagreement in the industry about the value and benefits of software inspections. However, there's more to implementing a software inspection program than training moderators and creating forms. This paper discusses how the software inspection program was implemented at HP's Patient Care Monitoring Systems Division, with emphasis on how the program is sustained and how its success is measured.

One thing we learned while implementing and sustaining an inspection program is that it must be managed with a clear organizational owner and a champion (chief moderator). The process must be flexible enough to withstand changes and improvements without compromising those things that define formal inspections such as preparation, inspection, rework, and so on. The implementation is really an evolution that needs tailoring to the culture and environment while keeping the fundamental process intact.

We have conducted over 85 inspections at our division. Data has been collected and maintained in a database from the very beginning so we could analyze how well the process is working. Besides data about the process itself, we keep data about the rework performed after each inspection, including the time to fix defects and the cause of each defect. The cause data helps us look at the software development process and identify areas for further investigation and improvement.

Background

A formal software inspection process was introduced to our division about two years ago to increase the efficiency of the defect detection process and ultimately shorten the time to market for key products in the division. From our historical data on software projects, we know that it takes an average of 20 hours to find and fix a defect detected during the testing phase of a project. We also know that finding and fixing defects earlier in the development process takes less time. The problem we had was to introduce software inspections into the software development process as painlessly as possible to gain acceptance of the process and start collecting data that would prove the value of the process.

We modelled our inspection process after a method that uses clearly defined process steps: kickoff, preparation, defect logging meeting, causal analysis, rework, and follow-up.[1]

Kickoff. This step is used to hand out the materials for the inspection, assign roles to inspectors, and ensure that everyone understands the purpose of the inspection as well as when and where the defect logging meeting will be held. This step can be done in a meeting or by electronic mail.

Preparation. The preparation step is critical to the success of every inspection. This is when inspectors review and check the document* on their own, noting any defects to be logged. The objective is for all inspectors to work independently to find defects so that they will come to the defect logging meeting prepared to report their defects.

Defect Logging Meeting. This is the time when all inspectors come together to review the document and report any defects that were found. During this meeting a defect log is maintained by the moderator. The meeting is facilitated by the moderator and should last no longer than two hours.

Causal Analysis. The purpose of this step is to review three to five major defects that were found during the defect logging meeting to determine the most likely causes for the defects. The causal analysis is done at the end of the defect logging meeting and involves brainstorming causes of the defects and possible actions that could be taken to prevent those defects from occurring again.

Rework. Rework is performed by the owner of the document that was inspected. The owner is required to fix all defects and take appropriate action to ensure that every item logged during the defect logging meeting is addressed. The owner also assigns a defect cause code to every defect.

Follow-Up. In this step the moderator checks with the document owner to determine if all defects logged during the defect logging meeting were addressed. The moderator also collects and reports all appropriate inspection metrics during this step.

Our chief moderator was specially trained and then charged with implementing the software inspection process in the R&D lab. Although the process was modeled after the steps mentioned above, we have made modifications to the process that we felt were necessary to facilitate acceptance of the process in our organization. Our first goal was to start using inspections with one project, and then leverage that success

* This includes architecture documents, specifications, design documents, and code listings.

to help gain acceptance of inspections as a way of doing business in R&D. We also constantly looked for ways to improve the inspection process to increase the effectiveness and efficiency of finding defects.

Implementing the Software Inspection Program

Gaining Support. It is important to gain the proper support and acceptance whenever a new program is introduced. In the case of software inspections, the impact is felt across the entire R&D organization, and therefore it is very important to involve as many people as possible when trying to gain support and acceptance for the program.

Initially, the inspection process was accepted by one project team, including the project manager. They were ready, able, and willing to use inspections in all phases of the project, beginning with the architecture documents and continuing through specification, design, and code. It was important to gain acceptance from the project manager so that time could be allocated in the project schedule for inspections. Because the entire team had accepted the idea, we found that most inspections were very successful and showed a very large return on the investment for the time spent by the team in preparing for and participating in inspections.

While one team was targeted initially as the primary user of inspections, other teams were also encouraged to conduct inspections by their software quality engineering team leaders. The selling process was much less formal and relied primarily on the software quality engineering team leader to gain acceptance from the project team to do inspections. We found that this process was not as successful in gaining the acceptance required to make inspections a success throughout the R&D organization.

In one instance, we had data that showed that one project team was not getting a very high return on investment from the inspections they were holding. When we analyzed what was going on, we found that the team had never been given a presentation on inspections and didn't understand the purpose of inspections. We found that reviews were being held before the actual defect logging meeting to clean up the document before the inspection took place. As a result, defect logging meetings were held that found no critical, serious, or medium defects; only very low-level defects were found. Once we determined what seemed to be the root cause of the problem, an overview session was held with the team to explain the purpose and benefits of inspections. Inspections held after the presentation showed some improvement but more work needs to be done to determine other causes for the low return on investment.

All project teams must be given the same message about the purpose of inspections and they must have acceptance from project managers so that the proper time will be allowed for preparation and participation in defect logging meetings.

Chief Moderator. The chief moderator is the most important person when it comes to implementing and sustaining a software inspection program. It is the moderator's responsibility to make sure that the process is being followed correctly and to watch for variations in the data that would indicate a problem or a need for improvement. The importance of this role cannot be overstated when it comes to implementing and sustaining the inspection program.

Initially, the chief moderator acted as a champion for the inspection process, gaining acceptance from management and project teams for the process and putting the pieces in place to ensure that the process was followed in the same way by all participants. This involved developing forms customized for the site and writing a guide that could be read by anyone to help understand the inspection process. The next job of the chief moderator was to help other moderators learn how to moderate inspections by observing them as they conducted defect logging meetings and acting as a coach. It is important that the process be followed as consistently as possible. Having the chief moderator coach other moderators helped maintain the consistency of the inspection process.

The beginning stages of implementing the inspection process required constant attention by the chief moderator. Once the process was in place and the other moderators had been trained, the role of chief moderator was reduced to one of custodian, that is, analyzing the the data to find patterns that might indicate a problem. Although the role of the chief moderator has changed since we introduced the inspection process, our experience shows that without the constant watchfulness of the chief moderator the process will get out of control and the benefits of inspections will suffer.

Getting Started. Our initial inspections were performed on test scripts written by software quality engineers in the division. This allowed moderators to practice moderating and inspectors to see the process in action before it was used with project teams. Some of the bugs in the inspection forms were worked out and minor improvements to the process were made during this time.

Some shortcuts were taken to get the program started. We decided to go forward with inspections in spite of not having standards and checklists in place for every type of document. We recognized that creating standards and having them accepted by engineers as standards for writing documents and code would take too long. Although we have some standards and checklists in place, we still don't have a complete set. It remains a goal to develop standards for all of the different types of documents.

We also decided not to implement the causal analysis step of the inspection process during the initial phase of introducing inspections. Asking inspectors to stay on at the end of a two-hour meeting to do a causal analysis would increase resistance to participating in inspections. Now that the process is in place, we have recently started performing a causal analysis of three to five major defects at the end of every inspection. These limited causal analysis meetings are possible because inspectors are now familiar with the process and the defect logging meetings tend to be shorter. As with any change, however, it has been slow to take hold and it will take time to recognize the benefits of the causal analysis meetings.

This has proved to be a successful model for implementing inspections. It was very useful to practice doing inspections and make minor improvements before taking the program to the project teams. When the task of creating standards and checklists seemed too large and threatened to prevent the program from getting off the ground, we found it more useful to keep moving forward in spite of not having all the tools in place. We also felt that eliminating the causal analysis step

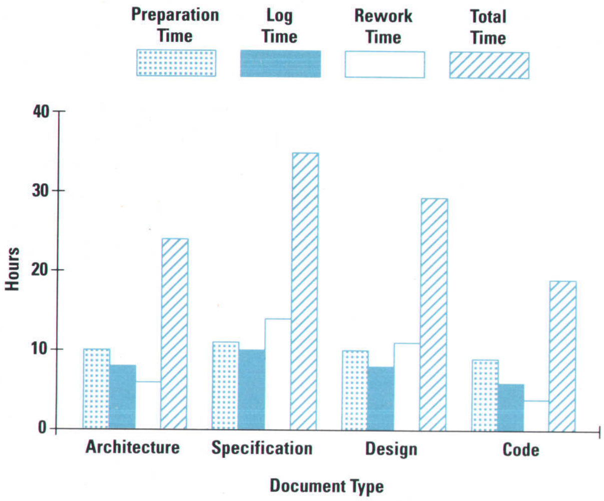


Fig. 1. Average inspection times by document type.

helped lower resistance to the new process and facilitated its acceptance by project teams. All of these shortcuts helped us get the program off the ground and have inspections start to take on a life of their own.

Sustaining and Improving the Process

Collecting Metrics. Metrics have been collected from the very beginning of our inspection program to enable us to measure the process and quantify the benefits of inspections. A Microsoft® Excel database was created to maintain the data. Data is collected from every inspection and includes the type of document inspected, the number of pages inspected, the number of defects found, defect severities, and the amount of time spent preparing for and participating in the inspection (see Fig. 1). This data helps us analyze how the inspection process is working. Average inspection effectiveness (defects per page), inspection efficiency (hours per defect), and return on investment are monitored to help us determine the overall usefulness and value of inspections (see Fig. 2).

Every defect found during an inspection must be resolved by the author of the document (or the engineer doing the rework). The reworker must assign a severity level to each defect and, using standard cause codes, assign a defect cause code for every defect that is fixed. The amount of time spent doing all of the rework is also noted. This data is maintained in the inspection database and analyzed to determine the most frequent causes of defects found during inspections. We have found that design defects are the most common types of defects discovered with inspections (see Fig. 3).* A further breakdown of the types of design defects is shown in Fig. 4. A more thorough analysis of this data in conjunction with the root-cause analysis data being collected at the end of each inspection should help us determine opportunities for improvement in the software development process.

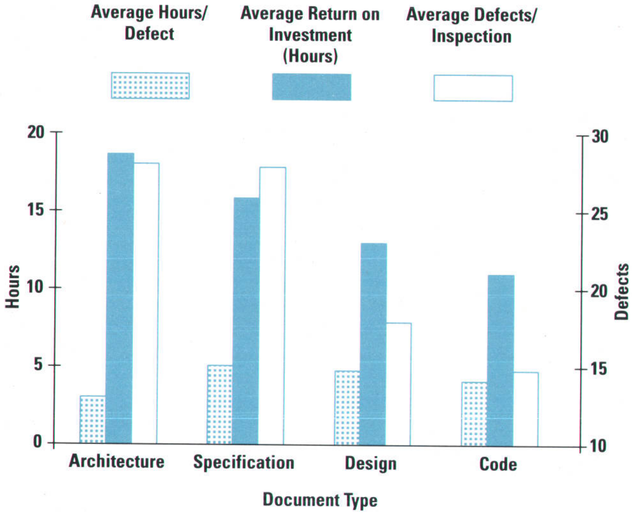


Fig. 2. Inspection data by document type. This data is intended to help determine the effectiveness of the inspection process.

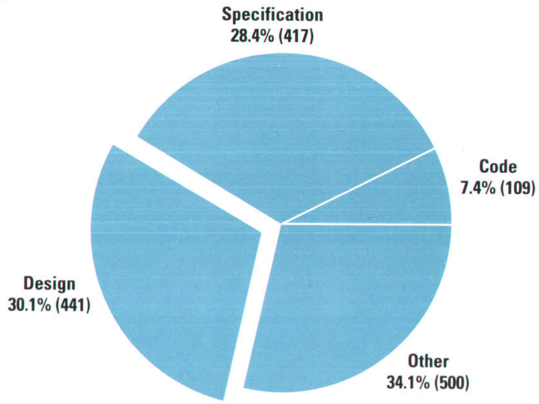


Fig. 3. Inspection defect causes.

Inspection metrics are also used to help sell the process to new project teams. We are able to show the results of every inspection held in the last two years and prove the usefulness of the process to the most skeptical engineers. We now have enough data to help other engineers and project managers estimate the amount of time to allocate for inspections based on the type and size of documents (see Fig. 5).

We continue to collect data and look for new ways to analyze it to help us determine where to make improvements in the process.

* Although we did not initially do the causal analysis step of the inspection process, we did require each reworker to identify as best they could the cause of every defect they fixed. This is the data shown in Fig. 3.

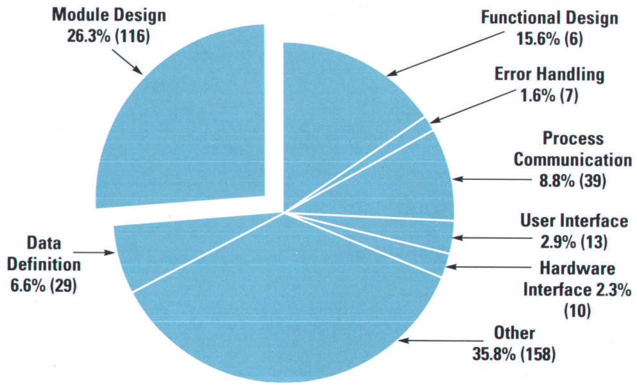


Fig. 4. Design defects by subcategory.

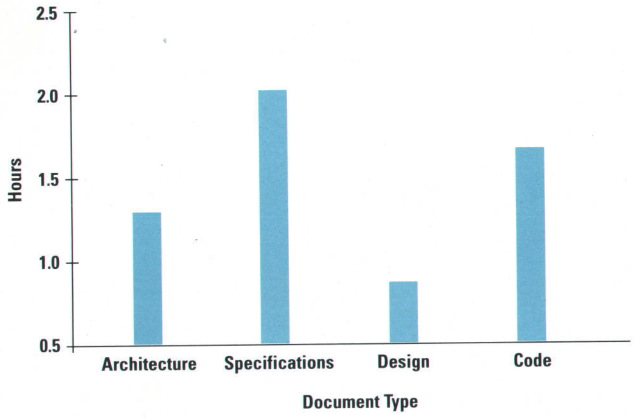


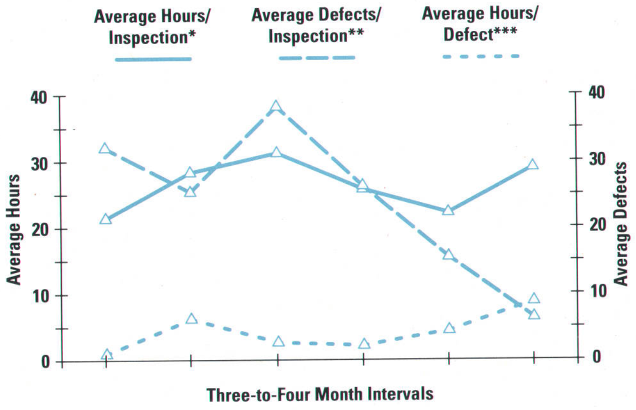
Fig. 5. Average hours per page by document. Each of these times includes preparation, defect logging, and rework time.

Intangible Benefits. We can't talk about the benefits of inspections without mentioning the intangible benefits, which are benefits that are hard to measure and quantify. For example, we have changed one of the rules for conducting the defect logging meeting by allowing limited discussions about defects when necessary. These discussions are allowed if they lead to a better understanding of a defect or explain how something works. However, discussions about style or how to fix a defect are not permitted. We have found that the increased communication, the transfer of knowledge, and the improved teamwork are some of the intangible benefits from these discussions. Inspectors leave the inspection feeling that they have gained something as well as given something.

Managing Process Improvements. All of the data collected is used to measure the inspection process and look for ways to improve it. It is important for the chief moderator and other moderators to be vigilant about the data from inspections and to look for opportunities for improvement. All of our moderators meet periodically to talk about what is working and what is not working. It is easy for the process to break down and lose some of its effectiveness.

We have found that analyzing the data over time is useful to show us whether our process is getting better or worse. For example, when we looked at the effectiveness and efficiency data points mentioned above at three-to-four-month intervals we saw that inspections were not as effective at finding defects as they were when we first started doing inspections (see Fig. 6). As a result, we have recently launched an inspection improvement project aimed at determining why our effectiveness and efficiency are dropping and what we can do to reverse the downward trend.

Defect Prevention. As mentioned earlier, when inspections were originally introduced, we decided not to include the causal analysis step of the process to facilitate acceptance of the process. Ultimately, however, the goal of software inspections is to help prevent defects from occurring again in the process. We are now trying to collect data on how some of the defects found during inspections could have been prevented by doing a causal analysis at the end of each inspection. We select three to five of the major defects that were reported during the defect logging meeting and try to get to the root cause of the defect by asking why it occurred. Over time we hope that we will accumulate enough data to help us pinpoint areas in the development process that need improvement.



$$* \frac{\sum_{i=1}^{n} h_i}{n} \qquad ** \frac{\sum_{i=1}^{n} d_i}{n} \qquad *** \frac{\sum_{i=1}^{n} \frac{h_i}{d_i}}{n}$$

h_i = Hours for Each Inspection

d_i = Defects Found During Each Inspection

n = Number of Inspections Conducted During a Particular Interval

Inspection Hours = Total Moderator Hours + (Kickoff Meeting Hours × Number of Participants) + Total Preparation Hours + (Defect Logging Meeting Hours × Number of Participants) + Rework Hours

Defects = Critical, Serious, and Medium Defects

Fig. 6. Software inspection trends. The calculation of each of these data points is based on the total number of defects, inspection hours, and inspections that occur within a particular interval.

Conclusion

The software inspections program at HP's Patient Care Monitoring Systems Division has been very successful. We know from our data that software inspections are a much more efficient way of finding defects than testing software at the end of the development process. We also know that we are getting a better return on investment for our inspection hours by investing them in inspecting the architecture, specification, and design documents for a product, rather than the traditional approach of inspecting only code.

Acknowledgments

Patsy Nicolazzo is the chief moderator and primary implementor of the software inspection program in the R&D organization at our division. It is because of her efforts that the program has been so successful.

Reference

1. T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley Publishing Company, 1988.

The Use of Total Quality Control Techniques to Improve the Software Localization Process

By implementing a few inexpensive process improvement steps, the time involved in doing translations for text used in HP's medical products has been significantly reduced.

by John W. Goodnow, Cindie A. Hammond, William A. Koppes, John J. Krieger, D. Kris Rovell-Rixx, and Sandra J. Warner

Text is a major element in the human interface of diagnostic ultrasound imaging products manufactured at HP's Imaging Systems Division (ISY). The ultrasound systems, which display real-time two-dimensional images of the anatomy, also display measurement and calculation labels, operator prompt messages, help screens, and anatomical annotation text (Fig. 1).

Regulatory requirements of certain countries stipulate that medical equipment sold in these countries must be localized with respect to software and hardware text and related user documentation.

Even where regulatory requirements do not stipulate language as a requirement, it can be a competitive advantage to have a localized product available.

ISY introduces new products or revisions of existing products at international medical conventions, where potential customers from all around the world see English-language product demonstrations. However, there is usually a large time delay between English-language and localized product availability. This can result in lost sales opportunities. In addition, some international sales contracts specify a financial penalty if a product is not delivered on schedule.

Localizing the software text in a product requires a significant engineering effort. Engineers must design the English product, provide documentation for the translators, implement the translated text, and assist with the language verification and software validation testing. It is prudent to make the time spent doing this localization work as efficient as possible.

This paper describes how Total Quality Control (TQC) was applied to the software localization at our division to reduce the time required to localize embedded software text used in

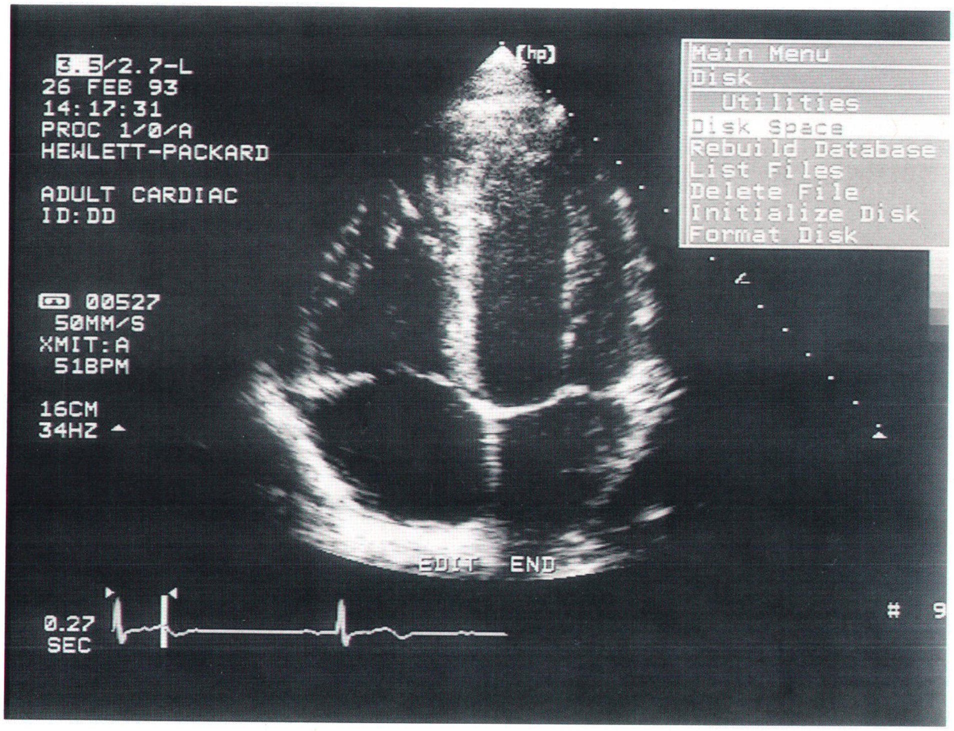


Fig. 1. A screen from an HP SONOS phased array imaging system.

1. Issue

Reduce the time required to localize the system software.

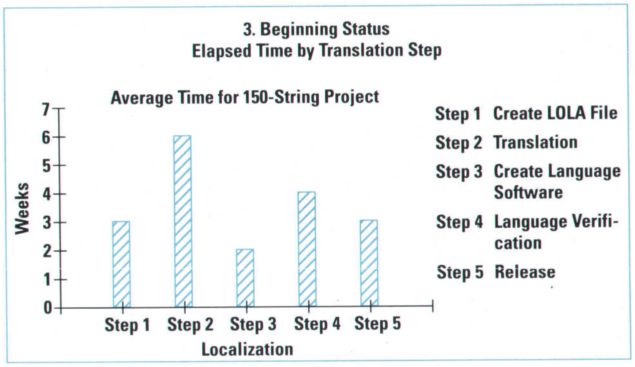
(a)

2. Why Selected

Decreases the time to market for all local language releases, which increases customer satisfaction and reduces the costs associated with transition plans outside of the U.S.

Reduces engineering resources required for localization.

(b)



(c)

Fig. 2. Panels for the first part of the TQC storyboard for the software translation enhancement project.

our medical diagnostic ultrasound systems. At first we thought that much of the delay in the localization process occurred in the translation step. By using TQC we discovered that we could reduce delays in areas originally thought to be beyond the control of the R&D lab.

The TQC Technique

The nine major stages of the TQC process improvement model are:

1. Issue
2. Why Selected
3. Beginning Status
4. Analysis
5. Actions
6. Results
7. Problems Remaining
8. Standardization
9. Future Plans.

We applied these nine steps to the issue of software localization.

The Issue. Our first step was to choose an issue and formulate a concise issue statement. A TQC issue statement must:

- Indicate a change or direction
- Have an indicator of quality in a product or service
- Declare the process or operation involved.

The issue statement for our project is shown in Fig. 2a.

Why Selected. This stage should state why the issue was selected. It should show that benefits can be gained by making improvements and conversely, that undesirable results will occur if the process continues unmodified. Fig. 2b shows the output from this stage for our project.

Beginning Status. Before beginning any changes to a process, the status of the current process must be understood. We first listed all the details involved in the current localization process. From these numerous details, we abstracted five major steps from the current process (see Fig. 2c).

The actions performed during this process include the following steps:

- Step 1: Create a LOLA file. The localization engineer extracts the English text from the source file and sends it to the translator in the form of a LOLA file. LOLA, which stands for local language, is an HP Vectra PC-based internal software tool for translation text entry. In LOLA, the translator sees the software text as isolated text strings.
- Step 2: Translation. The translator translates the text strings to the local language using LOLA. The translated text is sent back to the originating division for implementation.
- Step 3: Create language software. The translated text is encoded into software and memory chips (ROMs). The prototype system ROMs are then delivered to the translator.
- Step 4: Language verification. The translator verifies the translated text on a prototype product. During this step, the translator sees the text in its proper context and can verify that the wording is appropriate for the context. The translator sends corrections back to the localization engineer at the originating division.
- Step 5: Release. The localization engineer corrects the language software text, quality assurance performs final software validation, and manufacturing releases the localized products for shipment.

Because time was the indicator in our issue statement, we classified the following three different time process performance measures (PPMs):

PPM-1: The calendar time required for each of the five steps in the localization process
PPM-2: The calendar time from the English language release to the local language releases
PPM-3: The number of person days of R&D effort consumed during each step in the localization process.

To determine the time required for our current localization process, we evaluated several recent localization projects based on these PPMs. The average size of these projects was 150 software text strings (see Fig. 2c).

Analysis. During this stage we used brainstorming techniques to analyze previous projects to determine where time was spent. To promote an open and nonintimidating atmosphere, we followed these typical brainstorming guidelines:

- All ideas are permitted with one person speaking at a time
- Evaluation and discussion of ideas is postponed until later
- Questions are asked only to clarify ideas
- After all ideas are presented, the items on the list are clarified to be sure they are understood by all.

4. Analysis

Text is frozen very late, delaying the start of the translation process.

Local language releases have low visibility and are poorly managed.

Local language releases are often given low priority.

Not enough communication with the translators.

Translation tools used by R&D are poor (significant manual effort required).

R&D doesn't develop text with translation in mind.

(a)

5. Actions

Stabilize text and start the translation process earlier in the development cycle.

Increase the visibility of translation activities by providing regular status updates.

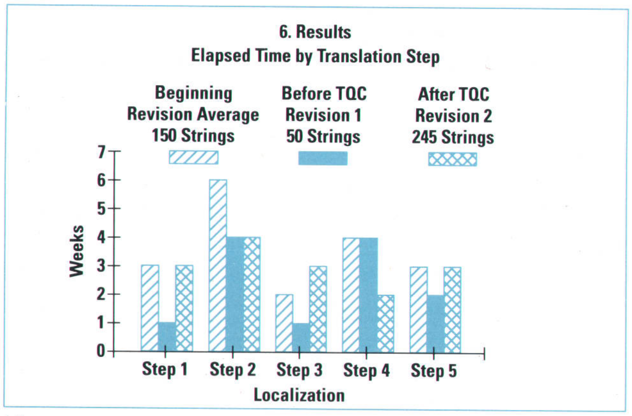
Incorporate translation activities into the project management documents.

Work to establish a better "team" relationship with the translators.

Develop better translation tools.

Provide software development guidelines for text.

(b)



(c)

Fig. 3. Panels for the second part of the TQC storyboard for the software translation enhancement project.

Several major items pertaining to ISY procedures were found from these brainstorming sessions:

- Preparing the original English LOLA file required many hours of manual effort to cut and paste text from the source code into the LOLA file. This process is tedious and error prone.
- The product development schedule did not provide enough time for localization activities. As soon as English was released engineering resources were dedicated to higher-priority projects. Localization was not a formal item in the product's release protocol.
- Most engineers at our division had an English-focused viewpoint and did not realize that certain design considerations are needed to make the software text localizable. Also, since no one was given the responsibility for localization, it was seen as someone else's job.
- Messages in languages other than English typically require between 30% and 50% more text characters. Often, the design of the original English text did not take this into account.
- Syntax in languages other than English is typically different. Sometimes engineers would construct text messages from individual translated words assuming that the syntax was universal. This resulted in nonsense localized messages.
- No attempt was made to stabilize software text before English release. Engineers continued to make text changes until final testing had begun. The translators either had to wait until the English product was released to begin, or they began earlier but had to revise the text numerous times.

In addition to the major items above, we also identified many smaller process items that involved our interactions with the translators. Fig. 3a summarizes all the issues found in the brainstorming session.

To better evaluate why these items occurred, we created several cause and effect diagrams, also called "fishbone" diagrams because of their visual similarity to a fish skeleton. We found four major categories in which time could be lost: process, communications, people, and tools. The categories are drawn as branches off the backbone of the cause and effect diagram shown in Fig. 4.

Each of these categories was branched further until root causes were identified. Each fishbone diagram highlights the causes for the step being addressed, which in this case is step 2 (translation).

Initially we created fishbone diagrams for Step 1 (create LOLA file) and Step 3 (create language software) because each required a large amount of engineering time and they were steps over which we had control. We believed that R&D had little control over Step 2 (translation), but since it took the longest time, we diagrammed it.

The fishbone diagram shown in Fig. 4 revealed some important issues. Since we do not have exclusive use of scarce translation resources, we must be on time in delivering translation materials to the translators. If we are late in sending our translation materials, the translators work on tasks from other HP divisions. We can keep our place in the work queue by giving more accurate dates to the translators.

Low estimates of the number of text strings to be translated were another cause of delay. The translators schedule their time based on the estimate we give them. If we underestimate the number of strings, our text cannot be translated in the time allotted. The translators then spend a lot of time juggling their projects to find time to translate the extra strings.

Our estimates of the number of strings were low because when the scope of the project increased, we did not update the estimate.

Poor communication between the translators and our division led to many small time delays, which together had a large impact on the turnaround time.

After creating the fishbone diagrams, we tied the root causes to the PPMs mentioned above. This gave us the insight needed for the next stage: developing the appropriate action plan.

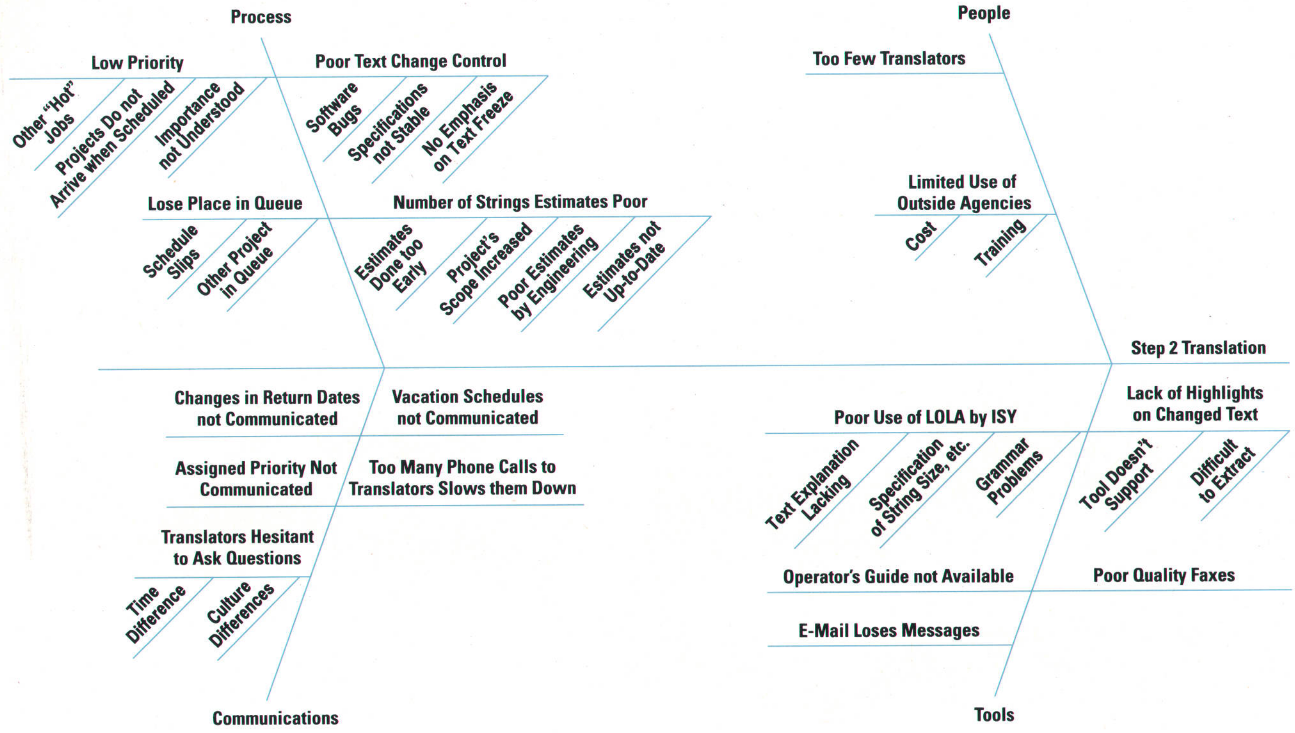

Process
Low Priority
Other "Hot" Jobs
Projects Do not Arrive when Scheduled
Importance not Understood
Poor Text Change Control
Software Bugs
Specifications not Stable
No Emphasis on Text Freeze
Lose Place in Queue
Schedule Slips
Other Project in Queue
Number of Strings Estimates Poor
Estimates Done too Early
Project's Scope Increased
Poor Estimates by Engineering
Estimates not Up-to-Date
People
Too Few Translators
Limited Use of Outside Agencies
Cost
Training
Step 2 Translation
Changes in Return Dates not Communicated
Vacation Schedules not Communicated
Assigned Priority Not Communicated
Too Many Phone Calls to Translators Slows them Down
Translators Hesitant to Ask Questions
Time Difference
Culture Differences
Communications
Poor Use of LOLA by ISY
Text Explanation Lacking
Specification of String Size, etc.
Grammar Problems
Lack of Highlights on Changed Text
Tool Doesn't Support
Difficult to Extract
Operator's Guide not Available
Poor Quality Faxes
E-Mail Loses Messages
Tools

Fig. 4. Cause and effect diagram for step 2 (translation) of the original translation process.

Actions. We held another brainstorming session to gather ideas to address the root causes of the delays. This resulted in a list of thirteen specific action items which were evaluated for impact, effort, and the amount of control we had over the action item. We also identified owners for the action items. Fig. 3b summarizes the main themes of the action items.

The following list ties the main themes of the action items to the problem they solve and the process performance measures (PPMs) that are impacted:

Action: Stabilize text and start the translation process earlier.
Problem: Text is frozen very late, delaying the start of the translation process.
PPM: 2—English-to-language release time

Action: Include translation activities in project management documents.
Problem: We often give local language releases a low priority and thus they are poorly managed and have low visibility.
PPMs: 1, 2, and 3—elapsed time for each step, English-to-language release time, and the number of R&D person days consumed

Action: Establish a better team relationship with the translators.
Problem: Communication
PPMs: 1, 2, and 3

Action: Investigate the development of our translation tools.
Problem: Significant manual effort is required to convert between LOLA and the source code.
PPMs: 1, 2, and 3

Action: Investigate software development guidelines for text string design.
Problem: English text must be reworked to function with translations.
PPMs: 1, 2, and 3

Results. After we generated the action plan, we began addressing the action items. This resulted in the following major changes to our localization process:

- In striving to release localized products earlier, stabilization of text has become a priority for software developers.
- The visibility of the translation activities has increased.
- Communication with the translators has improved, as has our credibility with them. By working as a team with the translators, we have improved communication and reduced turnaround time.
- Software development guidelines for handling localization have been established, published, and distributed internally.

Next, we evaluated the effect these actions had on our original goals by examining two localized product releases that occurred during the course of our activities. Fig. 3c shows PPM-1 (elapsed time between process steps) for the two product releases compared with the beginning average data from Fig. 2c. Revision 1 was before taking the actions listed above, while revision 2 shows the effects of the actions.

As we collected the data to generate the graph, we found that our PPMs were flawed. We did not have a good mechanism for scaling the results based on the number of text strings. Obviously, revision 1, which contained approximately 50 strings, would take a shorter period of time to localize than revision 2, which contained approximately 245 strings. Despite this problem, it is still apparent that our efforts have significantly reduced the times for steps 2 and 4. Because

(continued on page 69)

Tools for the Language Translation Process

The software translation enhancement project described in the accompanying article identified the need to reduce engineering time required to prepare text for translation and integrate translated text into the product. Partially automating the preparation and integration of local language text through the use of new development tools reduces inconsistencies between languages and frees valuable engineering resources.

Current Process

The current translation process is based around LOLA, or local language software tool, which is a PC-based application used by HP's medical products divisions for language translations. LOLA requires that files be in a special format, consisting of general information (such as revision, language, context), format specifications, and translatable text. The engineer responsible for localization manually converts a source file containing text to the required LOLA format. Converted files are moved from a workstation to the PC and sent to the translators. Translators using LOLA translate the files to the local language and return the translations to the originating division. The engineer then moves the files from the PC to a workstation and manually converts the translations to source code.

Requirements

The requirements identified by the TQC team for automatic handling of local language translations were:

- A unique text file format must be specified that will hold master English text and associated local language translations. The text format must be C-like in its construction and must contain all the information in a C source file as well as the information in a standard LOLA file.
- English and local language text files must be placed under revision control.
- The tools must automatically generate:
  - English LOLA files from master English text files
  - English C source files from master English text files
  - Local language text files from local language LOLA files
  - Local language LOLA files from local language text files
  - Local language C files from local language text files
  - A file reporting differences between two different versions of a text file.

Text Format

With the above requirements in mind, a file format was specified. Formats and global information take the form of functions. The name of the function indicates the format to be set, and the function argument indicates the value of the format (e.g., justify(CENTER);). Global formats, such as character sets and fonts, are usually software platform dependent, and are contained in the construct global{ }. Text and formats to be interpreted as translatable are contained in the construct string{ }.

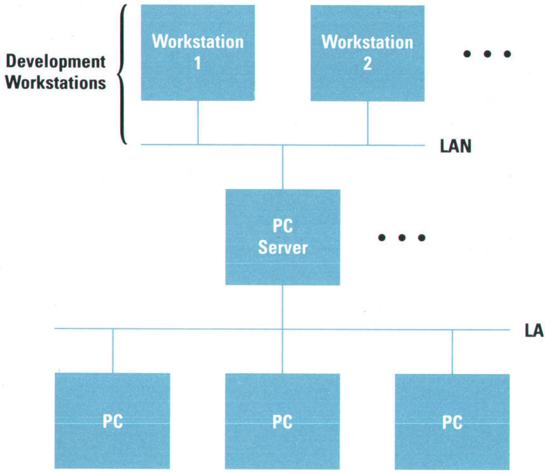


Fig. 1. The run-time environment for the translation tools.

C source code that logically belongs in the file can be included outside these additional constructs and will not be interpreted by the tools. A typical string definition is as follows:

```
string {
   size(4, 50); /* max size – 4 lines, 50 chars each */
   capitalize(LINE); /* capitalize first word of line */
   justify(LEFT); /* left justify text */
   const char *const User_def_msg= { /*user message that*/
      "Enter a comment to describe the", /*appears on display*/
      "LOOP (up to 16 characters):", 0};
   description(Dialog box prompt that appears when the user
      selects 'Manual Entry' from the store dialog box. The
      user is able to type in any 16-character string to
      describe the file to be stored. OKAY and CANCEL buttons
      will appear below the message. The user selects OKAY
      when satisfied with the comment or CANCEL
      to select a comment from the list instead.);
}
```

Localization Tools

The following localization tools were created to perform the required file conversions, calculate differences between revisions, and transfer files to and from a translation database. Fig. 1 shows the environment in which these tools run.

- **newtextfile**. This program generates header and global text information for the particular software environment.
- **text2hex**, **hex2text**, and **hex2c**. These tools perform the file conversions specified in the requirements.
- **textdiffs**. This is an interactive X Windows application. When an English file is changed, the engineer uses this program to generate the LOLA database files.
- **readyloclang**. This program transfers any files in the source tree to the LOLA database and checks the consistency of that database.
- **transferloclang**. This program moves LOLA files from the LOLA database to the PC server. It also puts the files in DOS format.
- **receiveloclang**. This program moves LOLA files from the PC server to the development workstation.
- **lolarc**. This program runs on the PC, and depending on the parameters sent to it, transfers LOLA files between the PC server and the PC.

The following scenario in conjunction with Fig. 2 shows the typical life of a text file using the localization tools.

1. A new English text file, which has been placed under revision control, is converted to hexadecimal and placed in a LOLA file.

2. The LOLA file is added to the translation database automatically by **readyloclang** when the database is ready for translation. **Readyloclang** will inform the user if **textdiffs** needs to be run on the file.

3. The LOLA files to be translated are transferred to the PC and archived.

4. The archived file is sent to the translators via e-mail for translation into the local language.

5. When the translators are done, the file is returned (via e-mail), unarchived, transferred to the PC server workstation, and then moved to a development workstation. This results in a local language LOLA file containing a reference to the English revision of the text file from which it was derived.

6. The original English revision of the text file and the LOLA local language file are used to create the local language text file.

7. The local language text file is checked in and C source code is automatically generated.

When engineers revise an English text file, the process is slightly more complicated. The engineer uses **textdiffs** to create the necessary LOLA files for the database. **Textdiffs** allows the engineer to match strings interactively that are the same

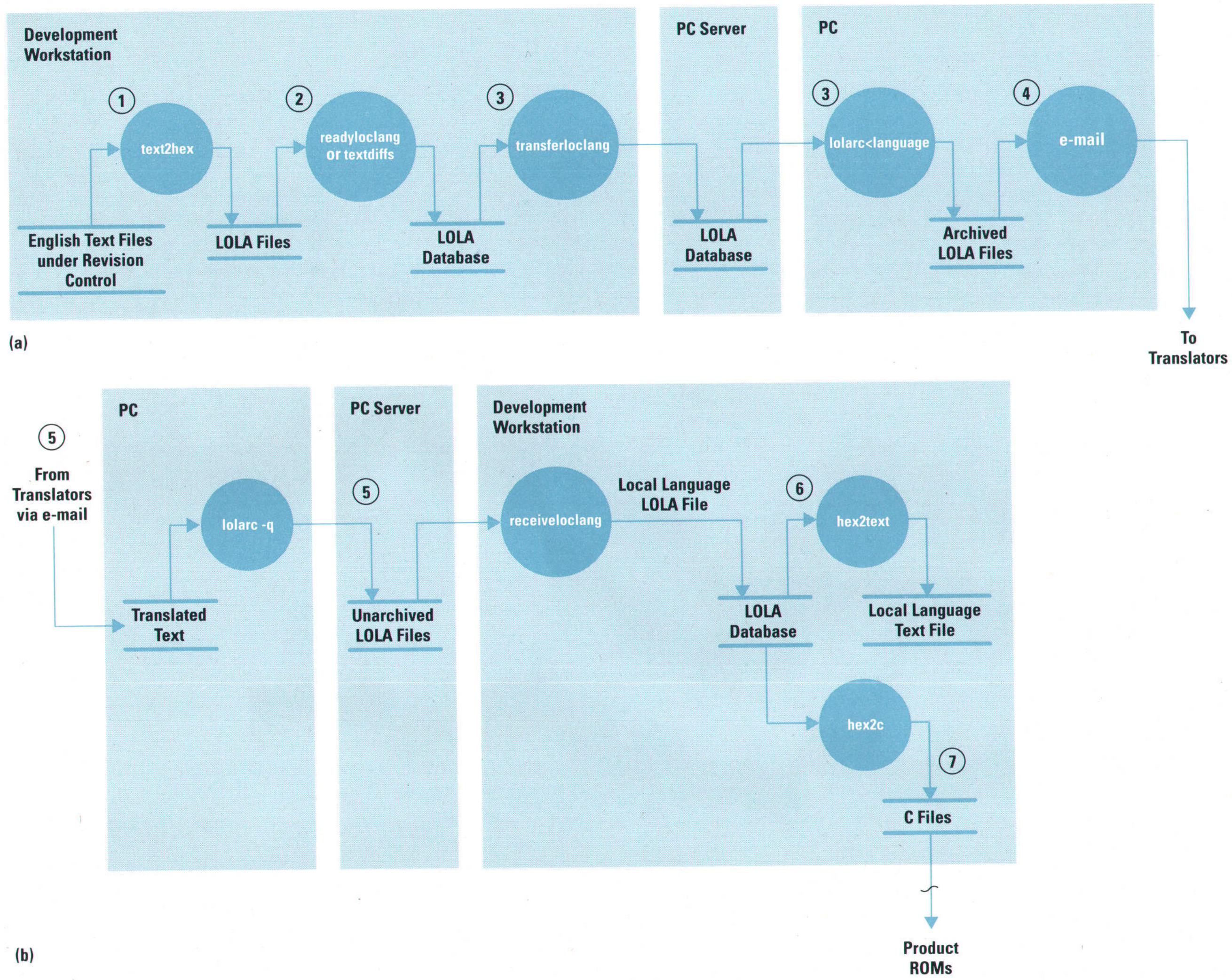


Fig. 2. The typical life cycle of a text file that is translated with the localization tools. (a) The flow from originating division to translators. (b) The return flow from translators to product ROMs.

between two revisions of the file. When the translator first looks at a matched line of text, the translated text from the previous revision will be shown.

Conclusion

The scheme outlined above helps to reduce the chance of inconsistencies between languages. Since all translated files refer back to English, comments, text formats, and code (other than text external to the additional constructs) are identical in all languages. Having a single file format to maintain ensures that changes to an English text file are carried through to the English LOLA file and the local language text file immediately after a change. Software testers use LOLA reports showing all the text to help them verify the operation of the system. These reports can be made available soon after any changes, reducing redundant defect reporting.

Embedding the format and comments in the text file centralizes documentation. Previously, the documentation was disseminated through the C file and the LOLA file. Having only a single file format to keep under revision control provides easier tracking of changes.

Ultimately, the new tools and processes will reduce the engineering resources required for translating to local languages and improve the quality of translated text.

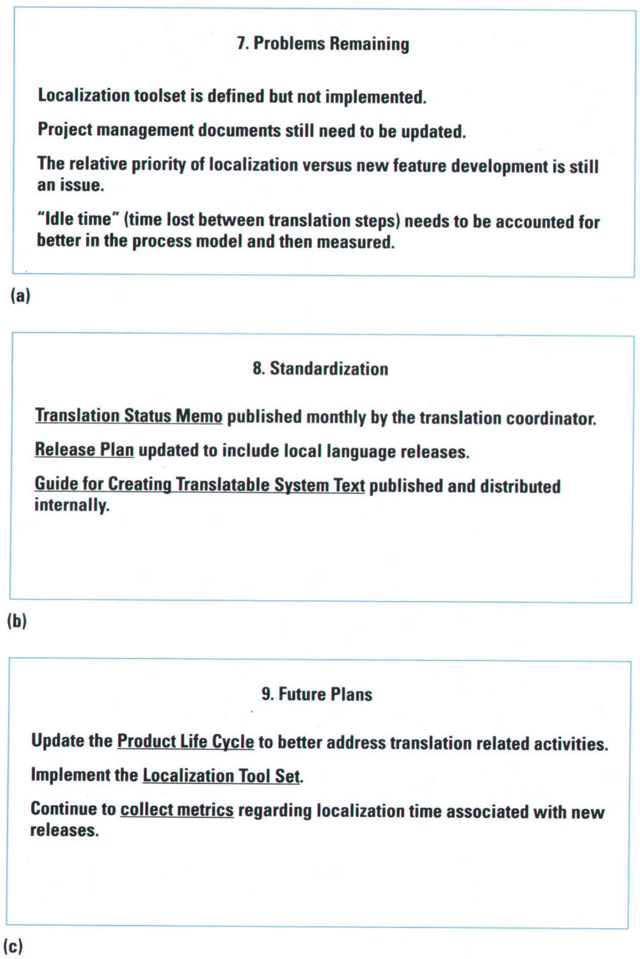
George Rom
Software Design Engineer
HP Imaging Systems Division

our action plan did not specifically address the R&D-intensive cutting and pasting process, steps 1, 3, and 5 did not improve significantly.

Problems Remaining. Although we were able to achieve a significant degree of success, we did not solve all of the localization issues we discovered during analysis (see Fig. 5a).

Standardization. The TQC process should lead to the development of standards. Standards help to maintain a process and provide a platform from which to continue to build (see Fig. 5b).

We developed three standard documents. The first of these documents, the Translation Status Memo, is published monthly. This document helps maintain high visibility for translations and clearly communicates changes to the schedule. The second document, the Division Release Plan, details the features and products scheduled for release in upcoming months. This document has been updated to include both local and English language release dates. Lastly, the Guide for Creating Translatable System Text has been published and distributed to the software development staff, and is a living document in our software environment.



7. Problems Remaining

Localization toolset is defined but not implemented.

Project management documents still need to be updated.

The relative priority of localization versus new feature development is still an issue.

"Idle time" (time lost between translation steps) needs to be accounted for better in the process model and then measured.

(a)

8. Standardization

Translation Status Memo published monthly by the translation coordinator.

Release Plan updated to include local language releases.

Guide for Creating Translatable System Text published and distributed internally.

(b)

9. Future Plans

Update the Product Life Cycle to better address translation related activities.

Implement the Localization Tool Set.

Continue to collect metrics regarding localization time associated with new releases.

(c)

Fig. 5. Panels for the third part of the TQC storyboard for the software translation enhancement project.

Future Plans. The final stage in the TQC process is generating plans for the future (see Fig. 5c). We plan to update our product life cycle process, and continue to collect metrics on local language releases.

In addition, we plan to collect metrics on a new localization tool set, which is design to improve our efficiency in handling English and translated text (see "Tools for the Language Translation Process" on page 68). This toolset will obviate the need to cut and paste, and will automate many of the other steps in the localization process. Without our TQC efforts, we would not have been able to justify the high cost of implementing these tools.

Conclusion

Our group, although untrained in TQC methodologies, successfully applied TQC principles to a real problem. Even though we focused on the results rather than the process, we learned a lot about the process. We were able to scale the use of TQC so that the process did not overshadow the problem at hand. In addition, we identified causes of problems that may not have been uncovered with an unstructured, ad hoc approach.

Given the positive experience we had with TQC methods during this project, we will enthusiastically and confidently use it again.

Acknowledgments

The authors would like to thank Paul Kelly for his encouragement and advice in the writing of this article.

A Transaction Approach to Error Handling

The transaction-based recovery concept used in databases can be applied to commercial applications to help provide more reusable and maintainable programs.

by Bruce A. Rafnel

Commercial programs contain two major paths: a forward path that does the work and a reverse path that rolls back the work when errors are detected. Typically, these paths are so tightly bound together that both paths are difficult to read. Code that is difficult to read results in code that is difficult to write, debug, enhance, and reuse.

For example, in the object-oriented programming methodology, one reason why objects are not as reusable as they should be is that they are tightly bound together at the error-handling level. Many times error codes even give clues about how an object is implemented.

The solution is to handle errors in programs as they are handled in a database transaction* recovery mechanism. In a database transaction, the transaction either executes in its entirety or, if an error is detected in any of its operations, it is totally canceled as if it had never executed. If an error is found, all work is automatically rolled back to the beginning of the transaction.

Error Handling

Software developers have sometimes been dismayed by how difficult commercial programs are to maintain and design, compared to programs they developed in school. Someone typically points out that programs developed in school were "toys," which assumed perfect inputs and hardware with unlimited memory and disk space. In addition, most software engineers have very little formal training in error-handling methods. Typically, software developers learned error handling by example or by trial and error, and they use the traditional error-handling model: check for an error, find an error, and return an error code.

Many formal design processes, such as structured analysis and structured design, recommend that errors be ignored during design because they are an implementation detail. It seems that this implementation detail can take up to one third of the code in commercial programs. This is not just code added around algorithms, but code placed directly in the middle of the algorithms. The resulting programs are difficult to read, debug, and reuse.

Exception handling, or error handling, has a large academic base and many of the ideas given in this paper are probably not new. However, most of the ideas presented here are based on 15 years of observations and experiences with a lot of good feedback from experienced programmers. This paper will describe a programming style that separates most of the error-handling process from the main algorithms.

Mixed Forward and Reverse Path Problem

The two major paths in commercial programs are shown in Fig. 1. The forward path is the path doing the work that the program is designed for. The reverse path is the error-handling code needed to keep the forward path working correctly. It does this by detecting problems, fixing them, and rolling back partially completed work to a point where the algorithm can continue forward again.

Tramp Error Problem

Often an intermediate function in a program has to stop what it is doing in the middle of the algorithm because a function it called cannot complete its designed task. This can lead to

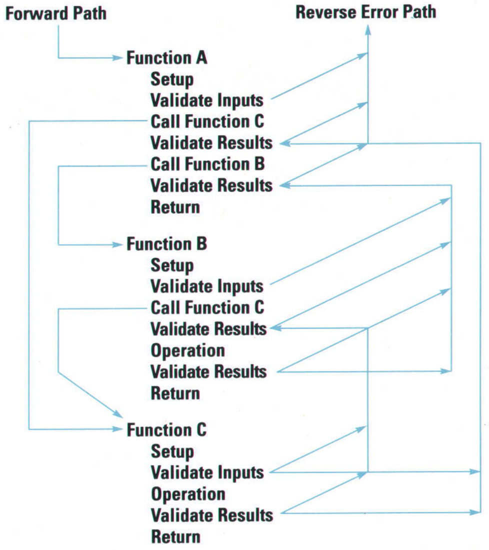


Fig. 1. Traditional error-handling program flow. The forward path does the work of the program and the reverse path does the error handling. Notice that error-handling code is dispersed throughout the algorithm.

* A database transaction is a unit of work that involves one or more operations on a database. For example, the operation of inserting data in the database could be a transaction if it's the only operation performed. If the insert is combined with an update, both operations would be considered one transaction.

Error Definition

In the accompanying article errors are not defects. Errors are exceptions that a particular algorithm is not designed to handle. Defects are errors that are out of the design limits of a whole application or system.

For example, many algorithms are designed with the assumption that there is unlimited memory. When there is not enough memory for the algorithm to complete successfully, this is an error. A whole application must be designed to handle these out-of-memory errors. If an application does not handle these errors and the program halts or the program behaves in an undocumented way, this is a defect. In other words, errors are relative in that they depend on what level of the software hierarchy is being observed.

Error handling consists of four main parts: detection, correction, recovery, and reporting. Error recovery is the main focus of the accompanying article.

"tramp errors."* Tramp errors are errors in functions that are not directly related to the current function.

Tramp errors are the result of a real error occurring in a lower-level function. For example, function A() calls function B(). Function B() needs some memory, so it calls the malloc() memory allocation function. The malloc() function returns an out-of-memory error. This is a real error for the malloc() function. Function B() does not know how to get more memory, so it has to stop and pass the error back to function A(). From the perspective of function B() and probably function A(), an out-of-memory error is a tramp error.

Tramp errors prevent functions from being the black boxes they were designed to be. In the above example, notice that function A() now knows something about how function B() is implemented.

Tramp errors are really part of error recovery and not part of error detection because if the real errors could be corrected immediately, tramp errors would not occur.

Unreadable Code and Poor Reuse

Mixed forward and reverse paths and tramp errors combine to obscure the main forward path of the program, which is doing the real work. The correction and recovery parts of error handling are the main areas that obscure the code. Most of the detection and reporting code can be put in separate functions.

Because of tramp errors, almost every function has to handle errors generated by all lower-level functions called. This can cause tight data coupling which makes code reuse more difficult.

Transaction Error-Handling Solution

To solve the above problems, two things need to be done: separate the forward processing path from the reverse error-processing path and use context independent error codes. This method of error handling is very similar to the way databases handle error recovery. Transactions are used to control the rollback process when a group of database operations cannot be completed successfully.

Separate the Paths

The traditional defensive way of programming is to assume that a function may have failed to complete its designed task, resulting in a lot of error-handling code to check for the errors and to roll back partially completed work. This is what we have in Fig. 1.

Reverse this assumption and assume that returning functions have completed their designed tasks successfully. If the function or any of the functions it calls has errors, it will pass processing control to a recovery point defined by the programmer. In other words, transaction points are defined so that if there are any problems, the work will be rolled back to those points and the processing will proceed forward again.

With this approach, there is no need to check for errors after each function call, and the forward path is not cluttered with tramp error-detection code. Only error-detection code for real errors remains, and most of the error-correction and recovery code is clustered around the beginning and end of the transactions (see Fig. 2).

Because errors are processed separately from where they are detected, the error codes need to hold the context of the error.

Context Independent Error Codes

Error codes that provide more information than just an error number are context independent error codes. Information such as what function generated the error, the state that caused the error, the recommended correction, and the error severity must be encoded in the error code so that it can be corrected in a location separate from the forward processing path.

Usually contexts of errors are encoded for error-reporting functions. For example, the names of the program, function, error type, and error code are saved and reported later. However, sophisticated encoding schemes are rarely used because with traditional error handling, the context of the

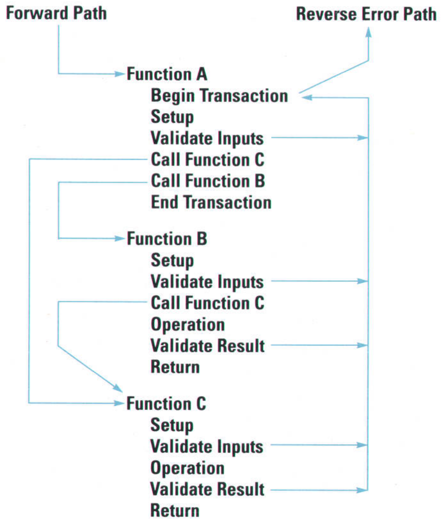


Fig. 2. Transaction error-handling program flow.

* The term tramp error is used because it is very similar to the tramp data term used in structured analysis and structured design.[1]

error is already known because checking is done right after a call to the offending function.

With transaction error handling, the recovery process is separated from the forward processing path so context independent error codes are required. This may involve the creation of unique error codes across a whole application or system (with the codes bound at compile time). An alternative would be to assign code ranges or other unique identifiers to functions at run time.

Code Readability and Reuse

The transaction approach makes programs easier to read because the reverse error process paths are visually separated from the forward process paths.

The transaction error-handling style makes it possible to create some general error-recovery interfaces so that functions (modules or objects) will only be loosely connected at the error-handling level. This is possible because the number of tramp errors used to control the recovery process is reduced and only the real errors need to be handled.

Implementation

The following are some ideas about how to start developing a transaction error-handling library. The list is not exhaustive and there are some problem areas, but it does offer some concrete ideas for building a transaction error-handling mechanism.

Transaction Control Management

Some language support is needed to implement the mechanism that controls error recovery. Languages like HP's Pascal-MODCAL have a try/recover feature that can be used to support a transaction error-handling style. The try/recover statement defines error-recovery code to be executed if an execution error is detected within a particular area of a program. Fig. 3 shows the flow of control for a try/recover statement.

For other languages a feature usually called a "global goto" must be used. This feature allows a lower-level function and all other functions above it to exit to a point defined in a higher-level function without passing error-code flags through all the other functions. In C this is done with the setjmp and longjmp library routines. The setjmp function saves its environment stack when it is called, and longjmp restores the environment saved by setjmp. The examples given later in this article are written in C and show how these functions are used.

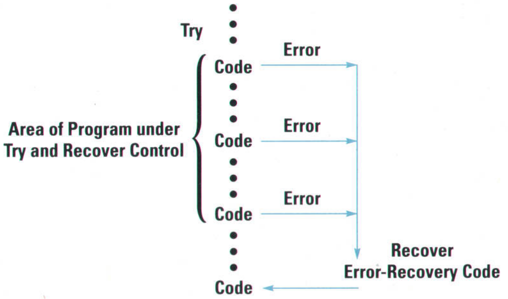


Fig. 3. The control flow for a try/recover statement.

The new C++ exception-handling feature[2] provides an excellent foundation for a transaction-based error handler. Reference 3 also describes how to add C++ error-handling functions to regular C programs. However, overuse of the C++ exception-handling feature could lead to code that is just as cluttered as the traditional error-handling style. Transaction boundaries for objects must be designed with the same care that goes into the design of an object's interface.

If the language is missing a global goto (or multithreaded) feature, macros or other wrapper* functions can be used to build recovery processes that are mostly invisible. Wrapper functions are described in more detail later.

Some of the features that might be considered for a transaction error-handling package include:

- Allowing nested transactions by keeping the transaction begin points on a stack
- Allowing functions to share a common transaction stack
- Allowing functions to define their own transactions with a common transaction stack or allowing functions to define their own transaction stack for special cases
- Defining special transaction points to handle errors in common categories (For example, abort the whole program, restart the whole program, close all files and restart, close current file and restart, and release all memory not needed and restart.)
- Making provisions for the transaction error handling to be turned on and off (When it is off, a function returns error codes in the traditional way.)
- Defining expected errors for some functions by masking out the errors needed. (This feature can be simulated by turning off transaction error handling, but then unexpected errors will also have to be managed.)

Transaction Data Management

Recovery involves more than just rolling back functions because there may be some intermediate work that needs to be undone. This may involve releasing unneeded memory or changing global variables back to the values they had at the beginning of the transaction.

Memory. Memory is best managed with a mechanism similar to the mark/release memory feature provided in some implementations of the Pascal programming language. The mark/release procedures allow dynamic allocation and deallocation of memory in an executing Pascal program.

The C language functions malloc() and free(), in conjunction with a stack of pointers to keep track of the memory allocated, provide the best features for allocating and freeing memory. With these features, a mark function can be called just before the program transaction's start point to mark the current stack point. If a longjmp() goes to this recovery point, a release function is called to free any memory allocated after the mark point.

A commit function, which indicates the successful completion of a transaction in the database context, is needed at the end of a program transaction to remove pointers from the mark/release stack. Nested transactions, however, need to be considered. A simple solution would be to have each transaction keep its own mark/release stack.

* Wrapper functions (or macros) are used to add functionality to existing functions that cannot be changed (e.g., library functions).

Globals. Global variables (and other static variables) can be rolled back with a strategy similar to the memory management problem. Just before a transaction's begin point the states of all the globals that might be changed in a transaction are saved on a stack. This allows transactions to be nested.

Context Independent Error Codes. The traditional error-handling style of checking error codes after each function call automatically gives errors a context. The transaction error-handling style needs to provide this context information in another way.

The biggest challenge here is that error codes alone are not very useful. For example, 97 could be the letter "a" (ASCII code), the digits "6" and "1" (BCD format), index 97 into a message array, the 97th error, an out-of-memory error, a disk-full error, a divide-by-zero error, and so forth.

To decode an error code the source of the error must be known. Some information that may need to be saved when an error occurs includes the machine name, program name, process number, module name, function name, and of course, the error code. This information needs to be sent only when it is necessary to roll back a transaction.

The amount of information that has to be saved is dependent on the location of the transaction recovery point and the run-time environment. For example, a client-server application may need more information than a simple PC application. Each recovery point can usually find higher-level context information fairly easily. For example, the names of the machine, program, module, and function can easily be passed down to a lower-level recovery point. However, lower-level context information cannot be collected because the function that had the error would no longer be active.

Implementation Summary

The following are some points to consider when implementing a transaction error-handling scheme:

- Put the rollback points (if any) at the beginning of functions
- Put error detection and default substitution at the beginning of functions
- Put some error-detection code in the middle of functions to check intermediate values
- Put error-detection code at the end of functions to validate the results
- Do not put error-handling code for managing rollbacks in the middle of a function.

Examples

Traditional Error-Handling Style. The following example program, which reads a binary formatted file, is coded with a common error-handling style. The code would have been more cluttered without the aExitErr() and aRetErr() macros to manage the error reporting and recovery. This example uses the simple error-recovery process: detect error, report error, and exit. However, notice how much error-handling code is mixed in with the algorithm.

```
/*  read.c – Read a binary formatted file                            */
/*  This program reads and prints a binary file that has the         */
/*  following structure:                                             */
/*                                                                   */
/*  Record type code (The last record has a value of 0)              */
/*  Size   Number of characters in Msg                               */
/*  Msg  0 to 2048 characters                                        */
/*  Record type code                                                 */
/*  Size                                                             */
/*  Msg                                                              */
/*  .                                                                */
/*  .                                                                */
/*  .                                                                */

#define aExitErr(pMsg, pErr)           puts(pMsg); exit(pErr)
#define aRetErr(pMsg, pErr)            puts(pMsg); return(pErr)

typedef struct {
    long     Type;
    int      Size;
}   aFileHead;

/*                                                                   */
/*  Forward Algorithm:                                               */
/*                                                                   */
/*     Main                                                          */
/*          1. Open the file.                                        */
/*          2. Call the Read process.                                */
/*          3. Close the file.                                       */
/*                                                                   */

main() {
    int       Err;
    FILE *    InFile;

    if ((InFile = fopen(“file.bin”, “rb”)) == NULL) {
               aExitErr(“Error: Could not open: file.bin”,1);
    }
    if ((Err = aRead(InFile)) != 0) {
               aExitErr(“Error: While reading: file.bin”, 2);
    }
    if (fclose(InFile) {
               aExitErr(“Error: Closing: file.bin”, 9);
    }
} /* main() */

/*  Forward Algorithm continued:                                     */
/*                                                                   */
/*     Read Process                                                  */
/*          1. Read the Type and Size values.                        */
/*          2. If Type = 0, exit.                                    */
/*          3. Read Size number of characters into the               */
/*             Msg variable.                                         */
/*          4. Print the Msg.                                        */
/*          5. Go to step 1.                                         */
/*                                                                   */

int aRead(pHandle)
    FILE *  pHandle;
{
    int               Err, N;
    char *            Msg;
    long              RecNum;
    aFileHead         RecHead;

    if ((Msg = (char *) malloc(2048)) == NULL) {
               aRetErr(“Error: Out of memory”, 3);
    }

    RecNum = 0L;
    while (1) {
         if (fseek(pHandle, RecNum, SEEK_SET) < 0) {
                    aRetErr(“Error: in fseek”, 4);
         }
         N = fread((char *) &RecHead, sizeof(aFileHead), 1, pHandle);
         if (N < 0) {
                    aRetErr(“Error: in fread”, 5);
         } else if (N != 1) {
                    aRetErr(“Error: short fread”, 6);
         }
```

```
        if (RecHead.Type == 0L) {
                return(0);              /* EOF */
        }
        if (RecHead.Size) {
                if ((N = fread(Msg, RecHead.Size, 1,
                    pHandle)) < 0) {
                        aRetErr("Error: in fread", 7);
                } else if (N != 1) {
                        aRetErr("Error: short fread",8);
                }
                if ((Err = aPrint(Msg,
                    RecHead.Size)) != 0) {
                        aRetErr("Error: in aPrint", Err);
                }
        }
        RecNum = RecNum + RecHead.Size + sizeof(aFileHead);
    }
} /* aRead() */
```

Transaction Error-Handling Method. The following listings show an implementation of the transaction error-handling style. The first listing shows the program (transaction) read.c rewritten to incorporate the transaction error-handling style. The other listings show the support functions for the transaction error-handling method.

Notice in the main body of the algorithm that the code following the recovery sections is clearer than the traditional error-handling example and there is no error-handling or recovery code mixed in with the algorithm.

There are some obvious shortcomings in the support modules. For example, most of the macros should be functions and the vEnv values should be saved in a linked list.

A number of engineers have pointed out that the transaction implementation of read.c is not really shorter than the traditional implementation of read.c because the error-handling code was simply moved out of read.c and put in the support functions. But that is exactly the goal: to remove the error-handling code from most functions and encapsulate the error-handling in common shared code.

The Main Program. This program performs the same function as the read.c program given above. However, it has been recoded to use the transaction style of error handling. The functions erSet, erUnset, and erRollBack provide the error handling and are defined in the include file erpub.h, which is described below.

The include file epub.h contains wrapper macros which are defined so that the appropriate transaction error-handling functions are called in place of the standard library function. For example, when the standard function fclose is invoked, the function eClose is actually called.

```
/*  read.c – Read a binary formatted file                           */
/*  This program reads and prints a binary file that has the         */
/*  following structure:                                             */
/*                                                                   */
/*  Record type code (The last record has a value of 0)              */
/*  Size  Number of characters in Msg                                */
/*  Msg  0 to 2048 characters                                        */
/*  Record type code                                                 */
/*  Size                                                             */
/*  Msg                                                              */
/*                                                                   */

#include    "erpub.h"
#include    "epub.h"
```

```
typedef struct {
    long    Type;
    int     Size;
} aFileHead;

/*                                                                   */
/*  Forward Algorithm:                                               */
/*                                                                   */
/*    Main                                                           */
/*        1. Open the file.                                          */
/*        2. Call the Read process.                                  */
/*        3. Close the file.                                         */
/*                                                                   */

main() {
    FILE *  InFile;

    erRecOn = 1;
    if (erSet()) { /* Transaction rollback point */
            printf("Error: %d in function: %s\n", erErr,
                    erFun);
        erUnset();
        exit(1);
    } /* End Recovery section */

    InFile = fopen("file.bin", "rb");
    aRead(InFile);
    fclose(InFile);
    erUnset();
} /* main() */

/*  Forward Algorithm continued:                                     */
/*                                                                   */
/*    Read Process                                                   */
/*        1. Read the Type and Size values.                          */
/*        2. If Type = 0, exit.                                      */
/*        3. Read Size number of characters into the                 */
/*           Msg variable.                                           */
/*        4. Print the Msg.                                          */
/*        5. Go to step 1.                                           */
/*                                                                   */

int aRead(pHandle)
    FILE * pHandle;
{
    char *          Msg;
    long            RecNum;
    aFileHead       RecHead;

    Msg = (char *) malloc(caMsgLen);

    RecNum = 0L;
    while (1) {
        fseek(pHandle, RecNum, SEEK_SET);
        fread((char *) &RecHead, sizeof(aFileHead), 1, pHandle);
        if (RecHead.Type == 0L) {
                return; /* EOF */
        }
        if (RecHead.Size) {
                fread(Msg, RecHead.Size, 1,pHandle);
                aPrint(Msg, RecHead.Size);
        }
        RecNum = RecNum + RecHead.Size + sizeof(aFileHead);
    }
} /* aRead() */
```

File erpub.h. The macros and global data structures defined in this file form a crude error transaction manager. The following operations are performed by these macros:

- erSet. This macro adds a rollback point to the vEnv (environment) stack.
- erUnset. This macro removes the top rollback point from the vEnv stack.

- erRollBack. This macro saves the function name and error code in a global area (erFun and erErr), and if the erRecOn flag is true, control is passed to the rollback point defined on the top of the vEnv stack. If erRecOn is false, erRollBack will simply return the usual error code.

Remember that these macros are for illustration only. Thus, there are no internal checks for problems, and the global data structures should be defined as static values in a library module or collected into a structure that is created and passed to each of the transaction error-handling functions.

```
/* erpub.h – Error Recovery Public Include file */
#include <setjmp.h>

/* Private Variables */
#define  vMaxEnv       5
jmp_buf  vEnv[vMaxEnv];
int      vLevel = –1;

/* Public Variables */
#define  cerFunNameLen 32
#define  erSet()       setjmp(vEnv[++vLevel])
#define  erUnset()     ––vLevel
#define  erRollBack(pFun, pErr, pRet)      \
         strncpy(erFun, pFun, cerFunNameLen); \
         erFun[cerFunNameLen-1] = \0; \
         erErr = pErr; \
         if (erRecOn && vLevel >= 0) { \
               longjmp(vEnv[vLevel], pErr); \
         } else { \
               return(pRet); \
         }
int      erErr = 0;
char     erFun[cerFunNameLen];
int      erRecOn = 0;
```

File epub.h. This file contains wrapper macros that cause the functions defined in the file e.c to be called in place of the standard library functions. The functions in e.c will behave the same as the standard library functions, but if the error transaction manager is on (erRecOn is true in erpub.h), control will be passed to the last defined rollback point, rather than just returning the same error code as the associated standard library function.

Using these wrapper macros makes it easier to add transaction error handling to old programs, but if it is desired to make the error-handling process more visible, the functions defined in e.c would be called directly instead of the standard library functions.

This file is also a good place to define context independent error codes.

```
/* epub.h – Error Library Wrapper Macros (only a few are
         shown here) */
#define  ceEOF          1
#define  ceOutOfMem     2
#define  ceReadErr      3
#define  ceReadShort    4

#ifndef  vInE
#define  fclose(pStream)                  eClose(pStream)
#define  fopen(pFileName, pType)          eOpen(pFileName, pType)
#define  fread(pPtr, pSize, pNItem, pStream) \
         eRead(pPtr, pSize, pNItem, pStream)
#define  fseek(pStream, pOffset, pPrtName) \
         eSeek(pStream, pOffset, pPrtName)
#define  malloc(pSize)                    eMalloc(pSize)
#endif
```

File e.c. This file contains the implementations of the wrapper macros defined in epub.h. Only two of the functions are shown in the following listing. Notice that these functions behave exactly like the standard library functions with the same name because they call the standard library functions.

For more flexibility, a real error transaction manager might allow the user to define the error codes that determine whether or not a rollback occurs.

```
/* e.c – Error Library Wrapper Functions (only a few are
         shown here) */
#define  vInE
#include        "epub.h"

void *   eMalloc(pSize)
         size_t  pSize;
{
         void *  Mem;
         if ((Mem = malloc(pSize)) == NULL) {
                 erRollBack("malloc", ceOutOfMem, Mem);
         }
         return(Mem);
} /* eMalloc */

size_t   eRead(pPtr, pSize, pNItem, pStream)
         char *  pPtr;
         size_t  pSize, pNItem;
         FILE *  pStream;
{
         size_t  Num;
         Num = fread(pPtr, pSize, pNItem, pStream);
         if (feof(pStream)) {
                 erRollBack("fread", ceEOF, Num);
         } else if (Num <= 0) {
                 erRollBack("fread", ceReadErr, Num);
         } else if (Num < pNItem) {
                 erRollBack("fread", ceReadShort, Num);
         }
         return(Num);
} /* eRead */
```

Conclusion

When transaction error handling was introduced on a project, engineers initially resisted removing IF statements after calls to functions using transaction error handling. After seeing how much easier the code was to write and read, the resistance faded and engineers started setting up their own transaction recovery points.

It would seem that debugging code using the transaction error-handling style would be difficult. However, experience has shown the debugging time to be a little better than the debugging time for a traditional error-handling style. This decrease in time can probably be attributed to less embedded error-handling code, causing defects to stand out more. Also when error-handling code is added, it is added in a structured way that disturbs very little of an already debugged program. This supports the way most engineers traditionally like to code.

So far this error-handling style has not been used on any large projects, but it has been used on small programs and functions written for enhancing old programs. One of the nicest features of this style is that it can be used independently of other error-handling styles.

In summary, a transaction error-handling style can lead to the following benefits:

- More reuse because error handling can be separated from the algorithm so that the coupling between functions is looser
- Improved code supportability because it is easier to read the algorithm and see what happens with errors
- Better code quality because there are fewer error-handling statements in the main algorithm, so the code is easier to read and the defects stand out.

Just as the functional parts of algorithms are being separated from user interfaces (client/server models), error handling can also be separated from the functional algorithm.

Acknowledgments

The author would like to thank a number of coworkers who helped review and refine the ideas presented in this paper, in particular: Andra Marynowski and Kevin Wentzel who were there at the birth of this coding style. Also thanks to King Wah Moberg for a number of reviews.

References

1. M. Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press, 1980, p. 104.

2. B. Stroustrup and M. Ellis, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, 1990.

3. C. Vidal, "Exception Handling," *The C Users Journal*, September 1992.

Authors

June 1993

6 ORCA Laboratory Robot

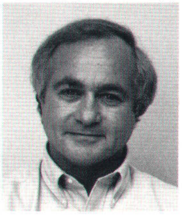
Gary B. Gordon

Gary Gordon was the project manager for the HP Laboratories phase of the ORCA robot project. He joined HP full-time in 1966 as a digital designer on the computing counter project, HP's first arithmetic unit to employ digital ICs. Later he became project manager and then section manager for HP's first laser interferometer. Gary is perhaps best known for pioneering HP's entry into instrumentation for digital designers with such instruments as the logic probe, clip, and pulser and the logic analyzer and signature analyzer. In 1979 he transferred to HP's central research laboratories, where he has been a project manager for a series of instruments for the HP Analytical Products Group including the just-introduced high-sensitivity capillary electrophoresis detector. Presently he heads a micromachining effort. Gary received a BSEE degree in RF communications from the University of California at Berkeley in 1962 and an MSEE degree in computer design from Stanford University in 1970. He has authored a dozen articles, has had products featured on seven magazine covers, is named as an inventor in 23 patents, and is listed in Who's Who in California. He served four years as a U.S. naval officer and is an associate professor at California State University at San Jose. His hobbies include large-format photography (he shot this issue's cover), flying, and designing and building modern furniture and houses.

Joseph C. Roark

Software design engineer Joe Roark was born in Springfield, Ohio and studied chemistry at Denison University (BS 1974) and Duke University (PhD 1980). He joined HP Laboratories in 1980, where he worked on prototypes for the HP 1046A fluorescence detector and the HP MicroAssay System. More recently, he moved to the Scientific Instruments Division and contributed to the architecture and the method development software for the ORCA robot project. He's presently working on networking and automation for HP ChemLAN products. A member of the American Chemical Society, Joe is named as a coinventor in a patent related to robot motion. Outside work, he plays jazz piano and soccer and coaches youth soccer and baseball teams. He is married and has two children.

Arthur Schleifer



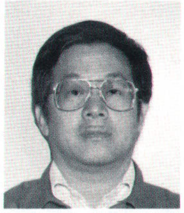
A New York City native, Artie Schleifer has held several technical and management positions since joining HP's Avondale Division in 1974. He contributed to the development of the HP 8450/51 diode array spectrophotometers and was project manager for robotic systems at HP Genenchem, HP's joint venture with Genentech, Inc. Now at the Scientific Instruments Division, he was project manager for the Analytical Products Group for the ORCA project and currently works on hyphenated instrument control and DOS and Windows systems software. Artie received a BS degree in chemistry from the City University of New York in 1971 and worked at Wyeth Laboratories before coming to HP. He is the author of six papers and conference presentations on chemistry, instrument software, and automation and is named as an inventor in three patents on software algorithms and robotics. Artie coaches soccer and baseball, has two sons, and enjoys sailing, boardsailing, tennis, softball, golf, woodworking, gardening, and skiing.

20 HP OpenODB

Rafiul Ahad

As architect of the HP OpenODB program at HP's Commercial Systems Division, Rafiul Ahad is responsible for software design and development and consultation with customers. Born in Rangoon, Burma, he studied physics and computer science at Rangoon University. His degrees (BSc in physics and MSc in computer science) were awarded in 1973 and 1975. He continued his studies at the Asian Institute of Technology in Bangkok, Thailand, from which he received an MSc degree in computer applications in 1980. After coming to the United States, he completed work for a PhD degree in computer science at the University of Southern California in 1985. Before coming to HP in 1989, he was an assistant professor at the University of Maryland. Rafiul is the author of four technical articles in the area of database systems and has presented papers at numerous conferences. He is a member of the ACM and the IEEE. He is married, has two children, and enjoys tennis and volleyball.

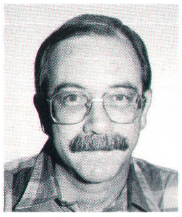
Tu-Ting Cheng



R&D section manager Tu-Ting Cheng was born in Bangkok, Thailand and attended National Taiwan University, from which he received a BSEE degree in 1969. Later, he completed work for an MSCS degree from the University of Wisconsin at Madison (1971) and for MS and PhD degrees in computer science from Ohio State University (1975 and 1976). With HP since 1976, most of his work has been in the database area, and he is now responsible for the HP OpenODB program. Tu-Ting and his wife have one child, and he likes ballroom dancing.

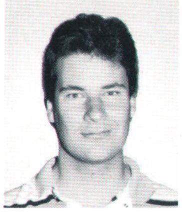
31 HP Ultra VGA Graphics Board

Myron R. Tuttle



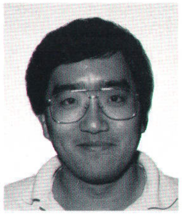
A development engineer at the California PC Division, Myron Tuttle studied electrical engineering at the University of California at Berkeley (BSEE 1973 and MSEE 1974). With HP since 1974, he worked on the HP 2625/28 terminals and the original multimode video board for the HP Vectra. He contributed to the development of the video subsystem for the HP Ultra VGA board and is now involved in video and graphics development. Myron is named as the inventor for a patent on automated software testing and is coauthor of an earlier HP Journal article as well as a paper for an HP software conference. He also served in the U.S. Navy as an electronic technician. His hobbies include computer programming, home improvement projects, and classical music.

Kenneth M. Wilson



With HP's California PC Division since 1989, Ken Wilson has worked on a series of HP Vectra products, including the Vectra 486/25T and 33T, the Vectra 486s/20, and the HP Super VGA board. Most recently, he contributed to the development of the HP Ultra VGA board. He completed work for a BSEE degree from California State Polytechnic College at San Luis Obispo in 1988 and expects to receive his MSEE degree from Stanford University in 1993. His professional specialty is computer architecture, and when he takes a break from work, he enjoys boardsailing and relaxing in his hot tub.

Samuel H. Chau



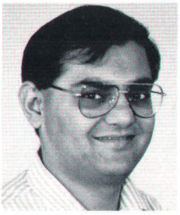
R&D engineer Sam Chau was born in Hong Kong and attended the University of California at Berkeley. He received his BA degree in computer science in 1984 and came to HP's Santa Clara Division in 1985. Now at the California Personal Computer Division, he contributed to the development of the HP Super VGA board and worked on the hardware and display timings for the HP Ultra VGA board and the HP Vectra 486U embedded Ultra VGA+. Sam's outside interests include personal computers, audio and video technologies, photography, piano, classical music, and badminton.

Yong Deng

Born in Shanghai, China, software design engineer Yong Deng joined HP's California Personal Computer Division in 1989. He studied for his bachelor's degree in computer science at the University of California at Santa Cruz and graduated in 1986. In the past, he was responsible for software drivers and utilities for HP's intelligent graphics controllers. He developed a new display redraw method that improved CAD display list performance. He also ported Texas Instruments Graphics Language (TIGA) 2.05 and 2.20 to HP's intelligent graphics controllers. For the HP Ultra VGA graphics project, he was responsible for the AutoCAD and Windows high-resolution display drivers and video BIOS. His other professional experience includes software development at National Semiconductor and Autodesk Inc. His professional interests include high-resolution display drivers and application development for Windows and CAD tools. Yong is married and has a young daughter.

41 POSIX Interface for MPE/iX

Rajesh Lalwani



A software engineer at the Commercial Systems Division, Rajesh Lalwani joined HP in 1988. He was born in Mandsaur in the Madhya Pradesh state of India. He received a master of technology degree in computer science from the Indian Institute of Technology, New Delhi in 1986 and an MSCS degree from Pennsylvania State University in 1988. In the past, he enhanced and maintained command interpreter software and components of the MPE operating system kernel. More recently, he developed a procedure for parsing MPE and POSIX file names and a directory traversal routine. He's currently working on symbolic links functionality and device files for MPE/iX. Rajesh is the author of several POSIX articles, has presented a number of conference papers on the same topic, and is working on a book on POSIX.1. His outside activities include tennis, watching classic movies, and staying in touch with his wife, who is finishing her medical degree in India.

47 Preventing Software Hazards

Brian Connolly

Brian Connolly is a project manager for software quality engineering in HP's Patient Monitoring Systems Division and has been with the company since 1984. Previously, he developed real-time software systems for Raytheon Corporation and Westinghouse Corporation. Since joining HP, he has worked on real-time software development for a bedside monitoring application, object-oriented software development in a clinical information system, and software quality engineering for several bedside monitor and central reporting station products. He has written several papers related to hazard avoidance and software quality and testing for internal HP publication. He's also a member of the IEEE. His educational background include a BS degree in physics and engineering awarded by Loyola College in 1977, and an MES degree (1983) in digital systems, also from Loyola. Brian is married and has two children. His leisure activities include running, swimming, woodworking, and coaching youth soccer.

53 Configuration Management for Tests

Leonard T. Schroath

With HP since 1985, software quality engineer Len Schroath worked at the Logic Systems Division and the Colorado Springs Division before moving to his current position at the Boise Printer Division. He's the author or coauthor of six papers for HP conferences related to software quality, testing, and reuse. Len was born in Detroit, Michigan and attended Brigham Young University, from which he received a BS degree in computer science in 1985. He is married, has three small children, and is a coach at his local YMCA. He also enjoys music and sports and officiates at basketball games.

60 Software Inspections

Jean M. MacLeod

A native of Arlington, Massachusetts, Jean MacLeod studied elementary education and sociology at Emmanuel College in Boston, Massachusetts, and received a BA degree in 1971. She has worked in the software quality field since 1974, initially in several small to mid-sized companies before joining Apollo in 1987. At Apollo, she was responsible for initiating a software inspection program in an R&D lab. She joined HP's Corporate Engineering Software Initiative in 1991, and contributed to the improvement of software inspections for the Patient Care Monitoring Systems Division at Waltham. She's now working on software process improvement with other divisions in the Northeast. Jean is a member of the Society of Women Engineers. She has two teenage children and enjoys golf, racquetball, and reading.

64 TQC for Software Localization

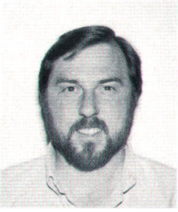
John W. Goodnow

A section manager at HP's Imaging Systems Division, John Goodnow joined HP in 1983 shortly after receiving a BS degree in electrical engineering from the University of Pennsylvania. He earned an MS degree in electrical engineering in 1988 from Stanford University through the Honors Coop program. His first HP projects included work on HP PageWriter electrocardiographs and ultrasound system software, and he continued work on ultrasound software development as a project manager and now as a section manager. John's professional interests include computer and system architecture, operating systems, and image processing. Born in York, Pennsylvania, he is married and enjoys boardsailing, skiing, woodworking, and vacationing on Martha's Vineyard.

Cindie A. Hammond

Cindie Hammond has been with HP's Imaging Systems Division since 1989. Born in Nassau, the Bahamas, she studied computer science at the University of Utah, from which she received a BS degree the same year she started at HP. An R&D software development engineer, her professional interests include ultrasound imaging and image processing. She's a member of the ACM and tutors mathematics at a local elementary school. Her outside activities include softball, boardsailing, drawing and painting, and renovating her home.

William A. Koppes

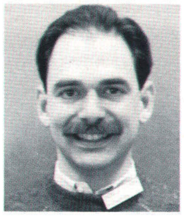


Bill Koppes was born in Morristown, New Jersey and studied electrical engineering at the University of Washington at Seattle (BSEE 1976) and at the University of California at Berkeley (MSEE 1978). At Berkeley's Donner Laboratory he also developed positron emission tomography and reconstruction algorithms. He joined HP's Imaging Systems Division in 1978, where he first was a hardware development engineer and then a software development engineer, project manager, and section manager. Now principal engineer at the Advanced Imaging Systems group, his professional interests include medical imaging and clinical diagnosis, software development, and R&D management. He is coauthor of a paper related to digital signal and image processing and has actively participated in several professional conferences. Bill is married, has a son and daughter, and enjoys composing and performing music.

John J. Krieger

John Krieger joined HP's Waltham Division in 1974, where he worked on the digital hardware and software design of the HP 47210A capnometer. After moving to the Imaging Systems Division, he contributed to the software design and development for the HP SONOS 100 and 1000 cardiovascular imaging systems. He's now specializing in diagnostic ultrasound imaging. He's the author of two previous HP Journal articles related to the HP 47210A capnometer, and has presented papers at two HP software engineering productivity conferences. He is also the inventor of a patent related to a help facility for the HP SONOS 100. Born in Santa Monica, California, John received a combined bachelor's and master's degree in electrical and biomedical engineering from the University of California at Los Angeles in 1973. He is married and has two daughters. Active in his church, he enjoys acting in community theater and is renovating his home, an 1850s vintage New England farm house.

Daniel Kris Rovell-Rixx



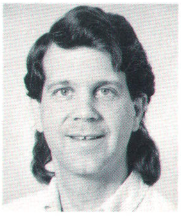
Kris Rovell-Rixx has been with HP since 1990 and is a software development engineer at the Imaging Systems Division. Previously, he designed and implemented real-time software for automated test equipment and fuel controls for jet aircraft at the Hamilton-Standard Division of United Technologies. He also worked for Ashton-Tate, and for a small manufacturer of IBM PC peripherals. Born in Miami, Florida, he completed work for a BS degree in engineering (computer science emphasis) in 1979 from the University of Florida and an MS degree in engineering management in 1987 from Western New England College. He's a member of the ACM and the IEEE. Kris is a sailing enthusiast. He and his wife have sailed in the Virgin Islands and Windward Islands, and in 1992 he was a volunteer for Sail Boston '92, a parade of tall ships. He's also an amateur radio operator (call sign WX1Z) and enjoys all types of music.

Sandra J. Warner

Sandy Warner joined HP in 1984 as a clinical applications specialist in the Midwest sales organization and is now a globalization specialist in the Imaging Systems Division. Her professional interests include market research on customer needs, foreign language translations, and ISO 9000 coordination. She was born in Rochester, New York and has a BS degree in zoology from The Ohio State University (1976). Before joining HP she managed a mobile cardiovascular testing service for a diagnostic service organization and before that she managed a diagnostic lab for a medical center. Outside of work, she teaches astronomy for an HP-sponsored school science program, and likes skiing, carpentry, landscaping, and backyard barbecue extravaganzas.

71 Transaction Error Handling

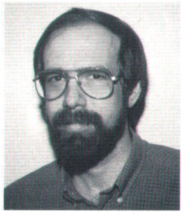
Bruce A. Rafnel



Software development engineer Bruce Rafnel has worked in the R&D labs at eight HP divisions since joining the company's General Systems Division in 1981. His contributions include the development of software for the HP 150 and HP Vectra personal computers and working on the dictionary team for the HP 3000 and HP 9000 computers. He's now in the Professional Services Division. A graduate of California Polytechnic State University at San Luis Obispo, he received a BS degree in computer science in 1981. He's a member of the IEEE and the C User's Group, and includes document management systems, computer graphics and image processing, object-oriented programming, and neural networks as professional interests. Bruce is married and has a young daughter. His hobbies include home automation with voice control.

80 HP-UX System Administration

Mark H. Notess



Mark Notess was born in Buffalo, New York and studied English and teaching English as a second language at Virginia Polytechnic Institute and State University. He received a BA degree in English in 1979 and an MA degree in education in 1981. He was an instructor at the University of Alabama and later was a programmer and instructional designer at the Virginia Cooperative Extension Service before completing an MS degree in computer science, again from Virginia Polytechnic Institute, in 1988. Since joining what is now HP's Open Systems Software Division the same year, he has designed, implemented, and tested software for several projects, including user interfaces for HP-UX system administration. His work on the object action manager for HP-UX has resulted in a patent application. He is a coauthor of two articles and has presented several papers at technical conferences. He's also a member of the ACM and SIGCHI. Mark is married and has three children. His leisure interests include reading medieval history and literature, hiking, and playing acoustic guitar.

A User Interface Management System for HP-UX System Administration Applications

Developing applications to simplify HP-UX system administration has been made easier by the creation of a tool that addresses the needs of the developer.

by Mark H. Notess

The HP-UX* system administration manager (SAM) provides basic system administration functionality for standalone HP-UX systems and diskless clusters. The SAM tool simplifies HP-UX system administration so that the administrator does not have to be a technical expert to manage an HP-UX system. Typical HP-UX system administration functions such as adding a peripheral, setting up the spooler, and adding or deleting users are provided in SAM. See "SAM versus Manual Administration" on page 81 for an example of the simplification provided with SAM.

By any measure, SAM is a large, complex interactive application. Of the approximately 150,000 lines of noncomment source code, almost half of it is directly related to the user interface. The SAM user interface consists of over 270 distinct screens, excluding help screens and messages. A substantial number of software, human factors, and learning products engineering hours have been spent working on SAM.

Any interactive application the size of SAM faces a major challenge in achieving a consistent user interface. User interface consistency includes many topics of concern such as:

- Interaction paradigm
- Selection and navigation methods
- Position, labeling, and behavior of common elements
- Relative layout of elements on the screen
- Colors, fonts, cursor shapes
- Message types
- Error handling.

With many developers coding portions of the SAM user interface, it is impossible to achieve consistency without some mechanisms in place to assist in the process. Style guides have been useful consistency mechanisms, but rarely suffice. We adopted a "no rules without tools" approach and decided to create a user interface tool that would enforce, where possible, consistency among developers. Where a semantic understanding of the interface was necessary, we developed a style guide, but the goal was to let the tool handle as much as possible. The other motivation for our tool was rapid development. Many general-purpose user interface toolkits are not easy to learn, are hard to use, and take a long time to master. Even user interface management systems that are easier to work with are still general-purpose and contain a lot of features that are irrelevant for a given application or class of applications. We wanted SAM developers to be as productive as possible, so we wanted to minimize the following:

- Time to learn the user interface tool
- Time to prototype
- Time to write working code.

To achieve rapid development, we needed to hide the underlying user interface technology† and provide an application program interface (API) that was at our developers' level of interest.

The result of our concern for user interface consistency and rapid development is the object action manager (ObAM), an application-class-specific API for SAM and SAM-like applications. The remainder of this paper describes the design of ObAM and reviews our preliminary results.

Design and Development

The old SAM user interface architecture did not shield developers from the underlying user interface technology and as a result they had to learn more about the technology than they wanted. While we did have a library of convenience functions such as message handlers, developers still found the API difficult to learn and a distraction from their primary focus—putting functionality in SAM. Providing SAM developers with a user interface toolkit they could use meant that we had to study the requirements of user interfaces for system administration and then factor out common elements and high-level constructs that would be most useful. Examples of common elements used by every application include displaying messages to the user and validating user input. The following sections list some of the higher-lever constructs we factored out.

List Management. Much of system administration is a matter of adding, deleting, or modifying items in a list. For example, system administrators frequently select an object such as a device file, print job, or disk from a list to perform some operation on. List manipulations such as filtering or sorting items can be useful for large lists and require the same type of interaction independent of what type of item is in the list.

† User interface technology includes components such as window managers, display handlers, and graphics routines.

SAM versus Manual Administration

To illustrate the simplification provided by SAM consider the following example, which shows what is involved in adding a user called samt to the group users without SAM and with SAM.

Without SAM. The following keyboard operations are required to add a user to the system.

1. Add the new user to the group users by editing the file /etc/group. After the edit we have:

```
users::20:john,harry,sue,klv,samt
```

2. Add samt to the /etc/passwd file. First make a backup copy of the file:

```
cp /etc/passwd /etc/passwd.old
```

3. Edit the /etc/passwd file to add a line for the the new user. The new line for samt might look like:

```
samt:,..:120:300:sam thomas,x2007:/users/samt:/bin/ksh
```

4. Create a home directory for the user:

```
cd /users               (go to users directory)
mkdir samt              (create a directory called samt)
chown samt samt                        (set ownership)
chgrp users samt                       (set group ownership)
chmod samt              (set permissions for samt)
```

5. Create a login environment for the new user by copying the default profile file from /etc to the new users directory:

```
cp /etc/d.profile /users/samt/.profile
```

With SAM. After selecting the Add... item from the SAM Users and Groups Actions menu, the administrator fills in a form. If the default values on the form are adequate, all the administrator has to do is:

1. Type in a login name and select OK.

2. Enter a password (twice) and again select OK.

SAM does the rest. This approach avoids error-prone procedures such as editing files and typing command strings.

Task Orientation. System administration is task-oriented. A typical task sequence consists of selecting an object to operate on, choosing the action to perform, supplying the necessary task parameters, launching the task, and verifying the success of the task.

Selectors. Supplying task parameters can be done with a small set of user interface elements we call *selectors.* Examples of selectors are a field for text entry, a list to choose from, or an option to toggle.

Procedures. Some tasks consist of multiple, possibly interdependent or sequential steps. While doing the steps, users want to be able to figure out where they are in the process, redo or revisit earlier steps, and so on.

The new SAM user interface architecture (Fig. 1) completely shields developers from the underlying user interface technology. Developers have only one API to learn, which is tailored to their needs. With this architecture developers retain direct control over the items most important to them while the ObAM controls the common elements. For example, element names, object attributes, and messages are controlled by the developer, and fonts, window positioning, labeling, and control button layout are under ObAM control.

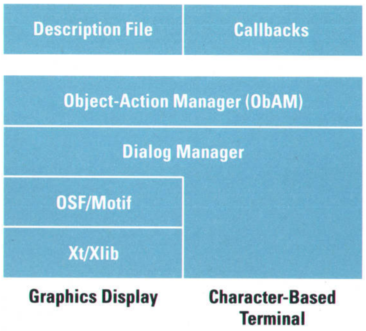


Fig. 1. The SAM user interface software architecture.

Choosing the Underlying Technology

Most user interface tools only support one display technology. With SAM, however, we wanted to provide an OSF/Motif graphical user interface while continuing to support a character-based terminal user interface. After an extensive evaluation, we selected a third-party tool called Dialog Manager from ISA† to provide the platform on which to build our user interface management system. Dialog Manager provides multiplatform support (OSF/Motif, terminals, Microsoft® Windows, MPE/iX, Presentation Manager, etc.), 16-bit internationalization, and run-time binding of the user interface.

ObAM

The major components of the ObAM are shown in Fig. 2. The description file defines the various screens used by a particular application and declares the functions to use in callbacks. These functions, which are associated with items on the screen via definitions in the description file, perform operations associated with those items. For example, a menu item for mounting a disk might eventually result in the execution of a C function containing a series of HP-UX system calls to accomplish the task. At run time the description file is read in and parsed, and a data structure is created with the appropriate linkages to the developers' callback functions. The object-list executor is responsible for listing objects on the screen (see Fig. 3) and facilitating user operation of the display, and the dialog box builder is responsible for creating dialog boxes on the screen.

Description File

The description file allows developers to define the type of objects they are managing, the attributes of those objects, the actions that can be applied to the objects, and the inputs that are necessary for those actions to proceed. ObAM interprets the contents of the description file at run time to create all the SAM screens.

The description file shown below is for a simple file manager application. The ObAM description file language is not a full-featured programming language; it contains only definitions and variables. No sequencing, branching, or looping constructs are defined in the language.

† Informationssysteme für Computerintegrierte Automatisierung GmbH.

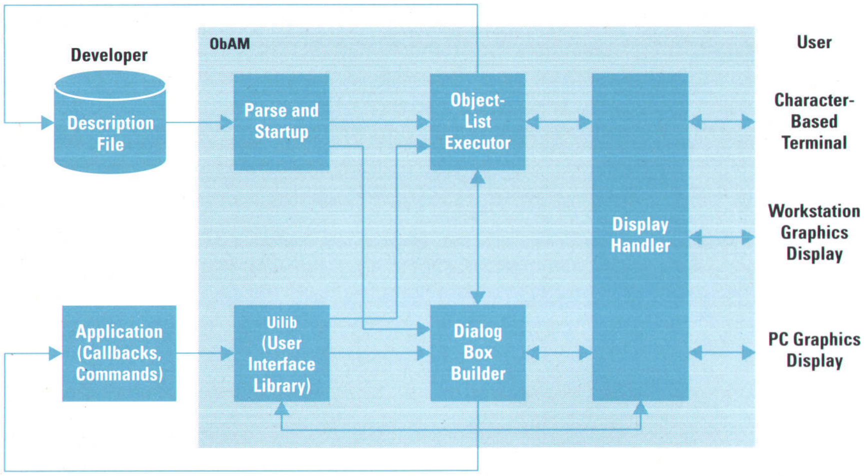
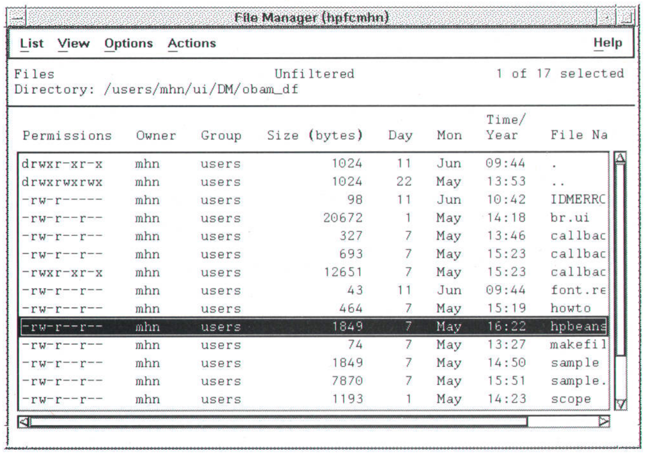


Fig. 2. The main components in and associated with ObAM.

```
library "callbacks.sl" /* points to shared library */

object_list_screen sample {
  label "File Manager"
  status_item fi_path
  label "Directory:"
  subarea files { /* multiple subareas may be defined */
    label "Files"
    entry callback fi_pwd() /*get path for current directory*/

  /*define the format and labels to show on the screen */
    table {
      init "/bin/ll –a 'pwd' | /bin/grep –v
        '^total' | awk '{print$1; print $2; print
        $3; print $4; print $5; print $6; print $7;
        print $8; print $9 }'"
      attr fi_perm { label "Permissions" column 1 }
      attr fi_links { label "Links" type numeric
                justify right }
      attr fi_owner { label "Owner" column 2 }
      attr fi_group { label "Group" column 3 }
      attr fi_size { label "Size (bytes)" column 4
                width 12 type numeric justify right}
      attr fi_month { label "Month" column 6 width 3}
      attr fi_day { label "Day" column 5 type
                numeric justify right }
      attr fi_time { label "Time/ \nYear" column 7 }
      attr fi_name { key label "File Name" column 8 )

    }
  /*Actions associated with the Actions menu item */
    action fi_remove {
      label "Remove"
      mnemonic "R"      /* Keyboard input selector */
      do "rm $(fi_name)"
      gray when no selections
    }
    action fi_cd {
      label "Change Directory"
      mnemonic "C"
      do fi_changedir
    }
    action fi_cd_immed {
      label "Change To"
      mnemonic "T"
      gray when no or multiple selections
      do fi_cd_to()
    }
  }
}

  /* define dialog box for the change directory action */

task_dialog fi_changedir {
  label "Change Working Directory"
  /* when the OK button is selected execute fi_cd_doit () */
  ok callback fi_cd_doit()
  text_edit fi_cd_path {
  label "New Directory:"
  width 30 }
}
```



```
Files                          Unfiltered            1 of 17 selected
Directory: /users/mhn/ui/DM/obam_df

                                                      Time/
Permissions  Owner  Group  Size (bytes)  Day  Mon    Year    File Na

drwxr-xr-x   mhn    users         1024   11   Jun    09:44   .
drwxrwxrwx   mhn    users         1024   22   May    13:53   ..
-rw-r-----   mhn    users           98   11   Jun    10:42   IDMERRC
-rw-r--r--   mhn    users        20672    1   May    14:18   br.ui
-rw-r--r--   mhn    users          327    7   May    13:46   callbac
-rw-r--r--   mhn    users          693    7   May    15:23   callbac
-rwxr-xr-x   mhn    users        12651    7   May    15:23   callbac
-rw-r--r--   mhn    users           43   11   Jun    09:44   font.re
-rw-r--r--   mhn    users          464    7   May    15:19   howto
-rw-r--r--   mhn    users         1849    7   May    16:22   hpbeans
-rw-r--r--   mhn    users           74    7   May    13:27   makefil
-rw-r--r--   mhn    users         1849    7   May    14:50   sample
-rw-r--r--   mhn    users         7870    7   May    15:51   sample.
-rw-r--r--   mhn    users         1193    1   May    14:23   scope
```

Fig. 3. The SAM screen that appears when the description file associated with our simple file manager example is parsed and displayed.

Fig. 3 shows the object-list screen that ObAM creates for this application. The List, View, Options, Actions, and Help menu items are put on the screen automatically. This is the same for other standard screen items. Object-list screens provide list manipulation with the List menu item. The Actions menu item contains a pull-down menu with actions that can be performed on the selected object. The View pull-down menu allows users to customize the list presentation through specifying the attributes (columns) to display, the order to display the columns, the objects to filter out, and how to sort the data.

Fig. 4 shows the dialog box defined by the task dialog fi_changedir in the listing above. When the action item Change Directory is selected from the Actions pull-down menu, ObAM accesses the task fi_changedir to determine what to do. According to the definitions in fi_changedir, when the user enters the desired directory in the selector and pushes the OK button, the callback function fi_cd_doit is executed. The callback

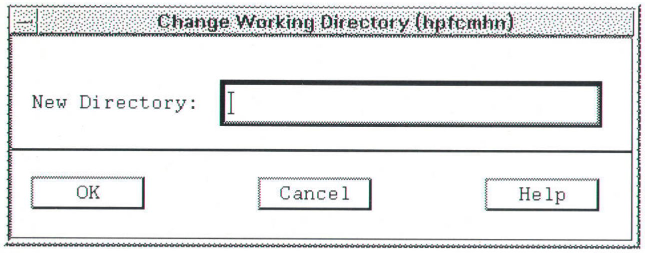


Fig. 4. The dialog box that appears when the Change Directory action item is selected from the Actions menu.

function executes the chdir command to change to the new directory.

For dialog boxes we wanted to lay out selectors (text_edit in this example) automatically rather than requiring developers to worry about positioning objects on the screen. As we designed our dialog-box builder, we realized it was unlikely that we could do all the layout automatically because too much semantic knowledge of the task is required to make the appropriate layout decisions. Our solution was to lay out selectors in the order in which they appear in the description file. In addition, if the developer defines more selectors than will fit on the screen, we add scroll bars to the screen automatically. ObAM also provides some simple layout constructs that allow specification of columns, skipping lines, indenting, and grouping. In trading off control for flexibility, we chose to control only where we had enough information to control appropriately.

Application functionality can be connected to the screens in two ways: callback functions, which can be used to get data from the screens and perform actions with that data, and a shell interface, which provides direct access to HP-UX commands from ObAM. Direct calls to commands can include variables as arguments so that screen data can be used in command execution. The Remove action in the sample description file uses this capability to remove one or more files.

User Interface Library

Within the developer's callbacks, the user interface library (uilib) functions provide access to the data on the screens and allow developers to control a limited number of screen characteristics such as visibility and graying. The code below shows the C functions associated (via a shared library) with the simple file manager example defined above.

```
int fi_cd_doit()
{
  char buf[1025];

  ui_get_data("fi_cd_path",buf);
  chdir(buf);
  return(0);
}/* fi_cd_doit */

int fi_cd_to()
{
  char buf[1025];

  ui_get_object_field("fi_name",buf);
  chdir(buf);
  return(0);
}/* fi_cd_doit */

int fi_pwd()
{
  char buf[1025];

  ui_set_status(1,getcwd(buf,1025));
  return(0);
}/* fi_pwd */
```

Evaluation and Discussion

Since ObAM has been in use for SAM development, our initial evaluation suggests that we have been successful in achieving our goals.

Developer Learning. Learning the new user interface tools is an order of magnitude faster and easier than the old tool. Users have reported learning times of two or three days for the new system as opposed to two or three weeks for the old system.

Prototyping. An entire functional area of SAM can now be prototyped in a day or two. Because ObAM supports using HP-UX commands directly as well as C callbacks, a completely functional prototype can be built without the developer having to write and compile any C code. Turning an ObAM prototype into an ObAM product is evolutionary. The screens can be constructed rapidly, and the functionality to support the screens can be added incrementally.

Development. Developer satisfaction is much higher with the new tools. Our old development tools required us to centralize screen creation responsibilities. If we had allowed developers to create their own screens with our old system, we would have paid a heavy price for having everyone learn the cumbersome screen creation tool we had available, and it would have been much more difficult to enforce consistency across all the areas of SAM. Consequently, in the old system, most of the SAM screens were created by one engineer. Requests for new screens or changes had to be funneled through that one person, even if the need was as trivial as changing the name of a callback or lengthening a field. In contrast, ObAM puts developers in control of the parts of the interface that are most important to them and reduces time-consuming dependencies between engineers.

Consistency. Our consistency issues can be divided into two categories: semantic and syntactic. Semantic consistency is achieved by mapping different sets of functionality onto ObAM capabilities using the same set of rules. This mapping has to be done by hand because ObAM is not intelligent enough to determine the attributes of a printer object, or to figure out what steps are needed to add a disk. We have produced a SAM user interface style guide to help developers with these decisions. Syntactic consistency is achieved by ensuring that similar user interface elements look and feel similar. ObAM has made our syntactic consistency effort much easier. ObAM code knows, for example, that a pushbutton label should be followed by "..." if pressing it leads to another screen; the developer doesn't have to think about this decision.

An unstated goal of our ObAM work was to create something that would be useful beyond SAM. SAM itself is not so much a single application as it is a collection of related applications such as a file system manager, a user accounts manager, a cluster configuration tool, and so on. We know that other projects have similar needs for rapid development of consistent user interfaces. Over the years, the SAM team has received many inquiries about how to create SAM-like user interfaces. This interest suggests that we have achieved an application-category user interface management system, not just a SAM-specific user interface management system.

Areas for Improvement

ObAM is still in its early stages, but we have already found areas for improvements and new features. Some of these areas include:

- More factoring is needed. Some messaging and error handling is repetitive enough that we could support it in the description-file, further reducing the amount of application developer code.
- Performance could be improved by precompiling screen descriptions. Our current architecture requires two separate parsing cycles for task dialog descriptions.
- Better error messages would aid learning and prototyping. Currently, ObAM identifies the point of failure when parsing a description file, but it could provide more helpful error messages.

Conclusion

Our success in creating an application-specific user interface management system suggests that other types of interactive applications could speed development and improve user interface consistency if an appropriate system existed that was tuned to the requirements of each type of application. Application-specific user interface management systems are likely to be beneficial when the target user interface has the following characteristics:

- Large size (many screens)
- Factorability (many similarities between interactions)
- Multiplatform (more than one user interface display technology must be supported)
- Developers do not have to be user interface designers.

Acknowledgments

ObAM was a team effort. Significant R&D contributions came from Mike Conca, Tammy Heiserman, Mike Kingdom, and Scott Warren, with Paula Curtis and Dan Castle providing human factors input. Management support came from Aland Adams the SAM project manager.

HP-UX is based on and is compatible with UNIX System Laboratories' UNIX* operating system. It also complies with X/Open's* XPG3, POSIX 1003.1 and SVID2 interface specifications.

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

X/Open is a trademark of X/Open Company Limited in the UK and other countries.

Microsoft is a U.S. registered trademark of Microsoft Corp.

Hewlett-Packard Company, P.O. Box 51827
Palo Alto, CA 94303-0724

ADDRESS CORRECTION REQUESTED

HEWLETT-PACKARD
JOURNAL

June 1993 Volume 44 • Number 3

Technical Information from the Laboratories of Hewlett-Packard Company

Hewlett-Packard Company, P.O. Box 51827
Palo Alto, California, 94303-0724 U.S.A.

Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan

CHANGE OF ADDRESS: To subscribe, change your address, or delete your name from our mailing list, send your request to Hewlett-Packard Journal, P.O. Box 51827, Palo Alto, CA 94303-0724 U.S.A. Include your old address label, if any. Allow 60 days.