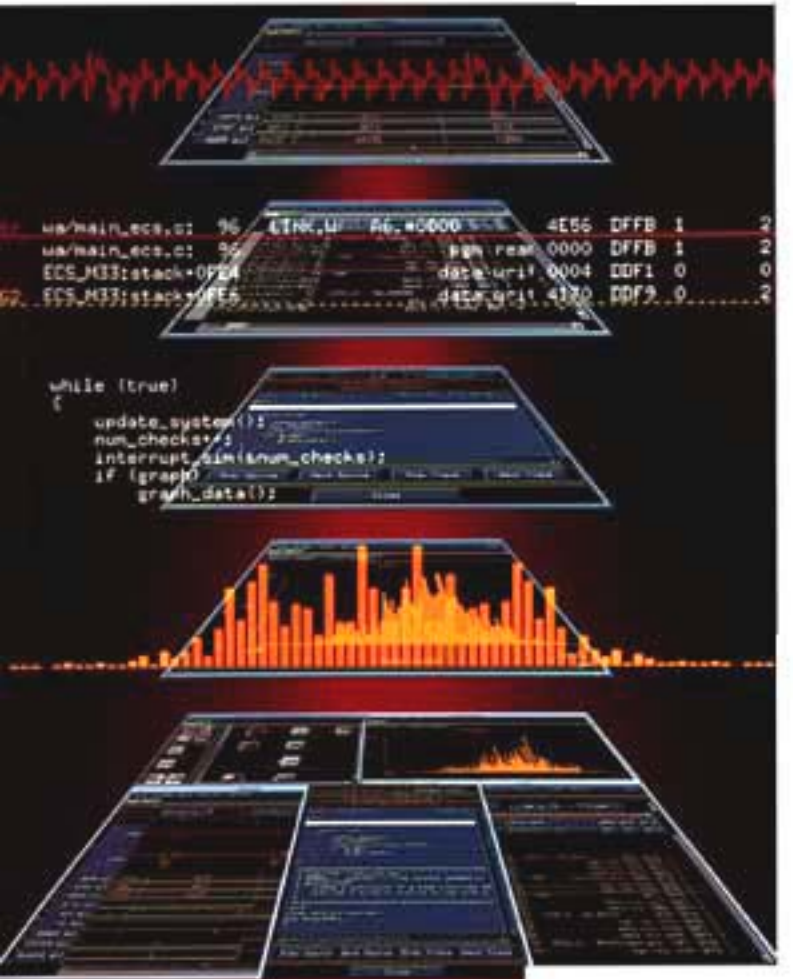
HEWLETT-PACKARD
JOURNAL

June 1996



HEWLETT PACKARD

table of contents

June 1996,
Volume 47, Issue 3

Articles

Reducing Time to Insight in Digital System Integration

Digital design teams are facing exponentially growing complexities and need processes and tools that reduce the time needed to gain insight into difficult system integration problems. This article describes modern digital systems in terms of the problems they create in the system integration phase. The debug cycle is described with special emphasis on the "insight loop," the most time-consuming phase of system integration. A case study from an HP workstation design effort is used to illustrate the principles. A new digital analysis tool, the HP 16505A prototype analyzer, is introduced as a means of solving these vexing problems more quickly by reducing time to insight.

by Patrick J. Byrne

The digital revolution is upon us in every form. Computer performance doubles every 18 months. Networks of high-performance servers are replacing mainframes at a dizzying pace. Personal communication systems are pervasive, from remote sales tools to medical information systems to networked workgroup tools.

What is behind this revolution? The answer is hardworking teams of engineers focused relentlessly on reducing time to market with the latest silicon and embedded systems. These teams of engineers are taking the latest silicon technology in ASICs (application-specific integrated circuits) and memory, exploiting them while adding value in the form of special application-focused architectures and implementations, and bringing the end products to market at the right time, price, and performance. These efforts are often made up of highly integrated subteams of four to eight engineers—hardware development specialists with expertise in ASIC design, printed circuit board system design, hardware architectural trade-offs, and the latest CAE (computer-aided engineering) tool methodologies. These teams specialize in designing high-performance, efficient, embedded software systems using the latest software design tools and languages. It is not unusual for design teams to use C++ and to have a carefully crafted multilayer software system with de facto APIs (application programming interfaces) between layers. These abstractions are used to manage the growing complexities of real-time embedded systems. These teams have to keep the customer in clear view as they make complex trade-offs between software and hardware implementation, between time to market and feature sets, between material cost (hence customer price) and component sourcing risks.

Concurrent Design and the Integration Phase

Figs. 1 and 2 illustrate the nature of the modern digital design process. Fig. 1 shows how the many competencies are brought together by the design team into the end product. The modern digital design process is highly concurrent. To make progress, each engineer makes assumptions about the other engineers' work. Sometimes these assumptions are well-documented, but often they are not because of the relentless schedule pressures. The risk accumulates throughout the design process because the tools used by different members of the design team do not explicitly articulate the interdependencies between subsystems. These design interdependencies are often found in the integration phase, when the whole design comes together for the first time. This is the well-known finger-pointing phase. "It's a hardware problem," says the software engineer. "It's a software problem," says the hardware engineer. The team members then sort out the problems from their respective points of view, engaging in the iterative and unstructured debug process.

Fig. 2 shows this development process in terms of cycle times. Twenty to thirty percent of the entire development process time is used in this unstructured integration phase. This phase of the design process is unstructured because there are so few well-designed tools and processes for engineering teams to use to work through the key integration tasks. In the integration phase, hardware meets hardware in the form of ASICs interacting with the circuit board. Interhardware design and modeling flaws are found here. Hardware meets software in the form of driver code running ASIC subsystems. There are also problems from application programs causing software integration problems. Application code provides an incredibly rich and complex stimulation system to the hardware.

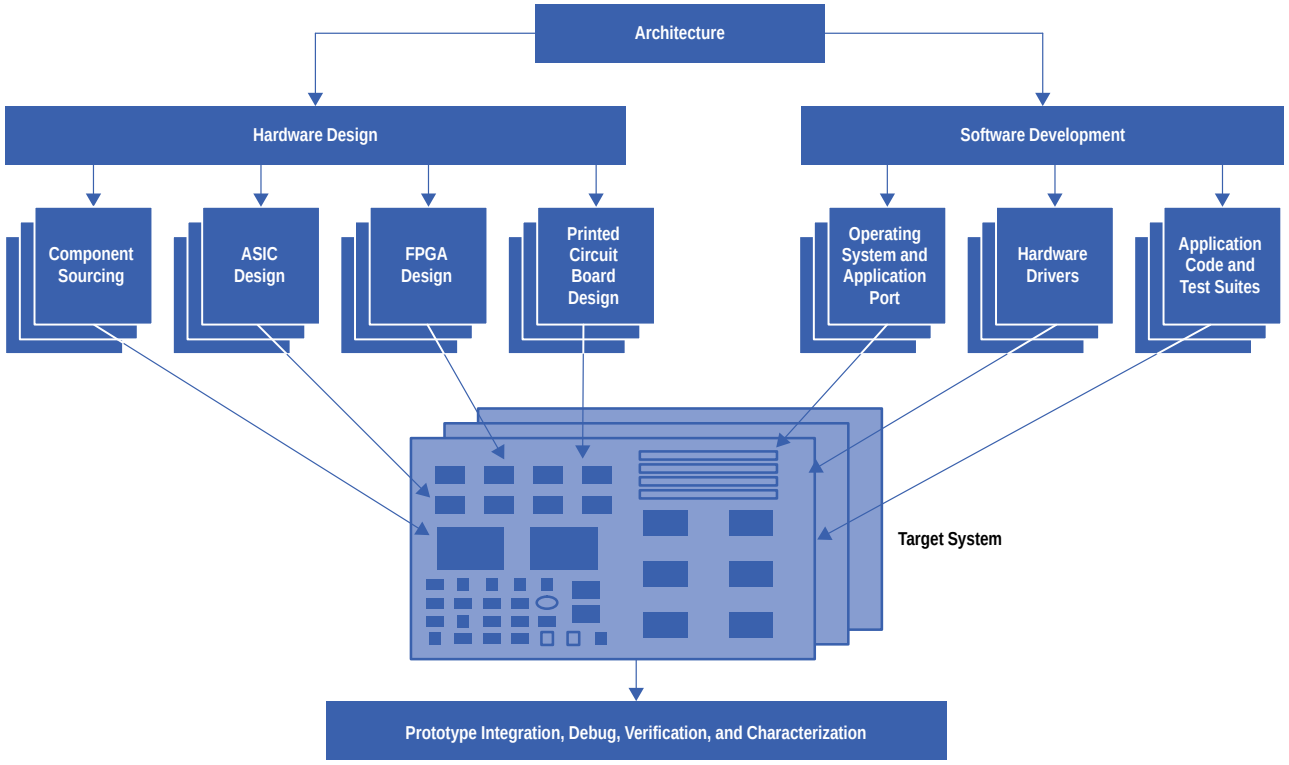


Fig. 1. *Digital system design process.*

Hardware and software modeling tools (simulators, compilers, logic synthesis, etc.) falter in their ability to model all these complex combinations, sequences, and data and code interdependencies. Fig. 3 shows the simulation execution speed of a typical 40-MHz system. Most digital circuit simulators execute at 10 to 30 cycles per second, leading to total system simulation times of days and sometimes weeks. This is too long for designers who are used to the iterative design and debug process. This shows that simulators are just too slow to expose real-world problems within the mental time constants designers need.

Hard Problems in the Real World

This rich and complex environment is the real-world environment indicated in Fig. 2. The modeled world in Fig. 2 is where designers work in isolation to predict real-world behavior. But it is very hard to predict every corner case to which the actual applications will take complex combinations of hardware and software. The real-world environment is the domain of the "hard problems," also called the "interesting cases" by some engineers. The software team has done their best to model the

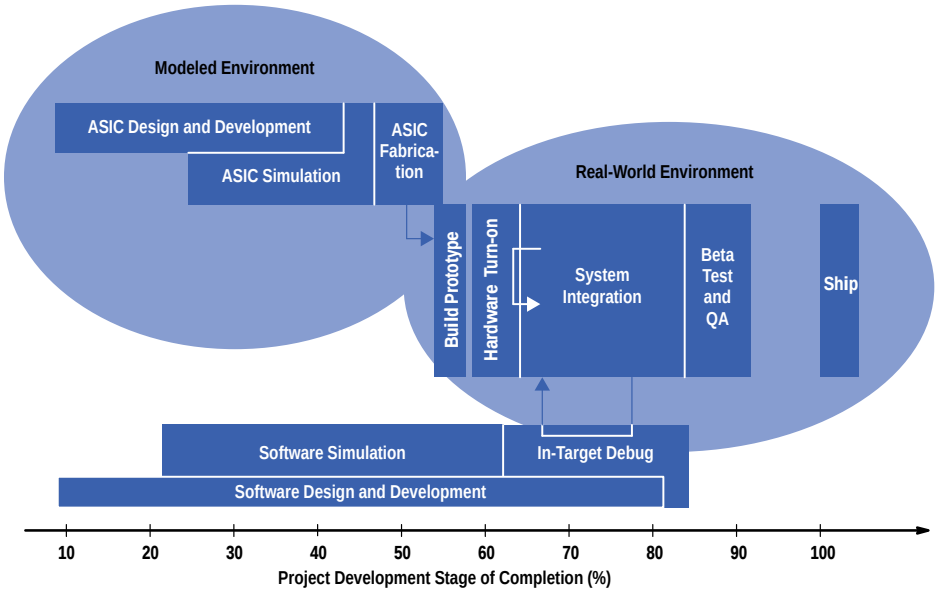


Fig. 2. *Stages in the development process for ASIC-based designs.*

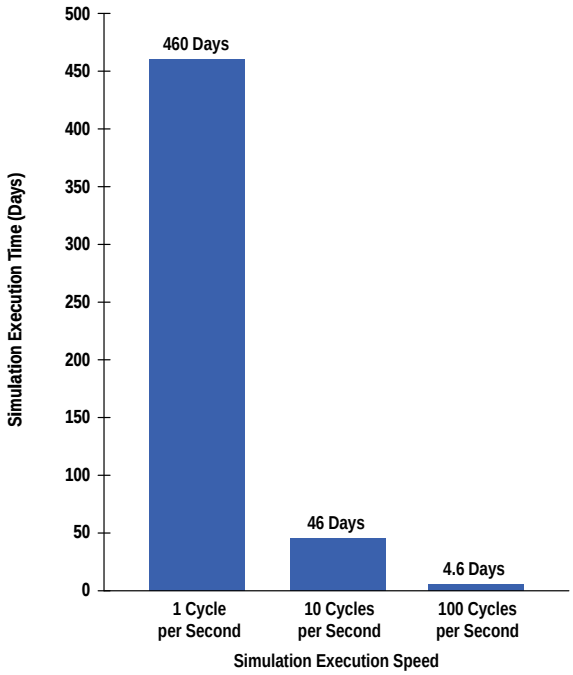


Fig. 3. *Time required to simulate one second of operation in a 40-MHz system.*

hardware. They have read the hardware documentation and written their driver and application code consistent with these models. The hardware engineers have done their best to model the software. Often, hardware engineers will standardize hardware interfaces specifically to avoid unmodeled software behavior. However, when the team puts the system together, they find unmodeled or undermodeled cases—the interesting cases. They find that their documentation doesn't cover some sequence of behavior by which the application code runs the ASIC into a state machine or switching condition that causes a glitch that delays response, causing the software to time out at the fourth layer of the protocol. It is very common to have the problems in the real world be separated, symptom from root cause, both temporally and spatially. For example, an ASIC on a peripheral bus has a state machine sequence that causes an error condition, but the errored packet is not sent back to the CPU until it fits into the priority stack. Or even more simply, the bad data is written to memory a long time before it is read and found to be faulty. The problem is detected later in time and at a different point in the system from where and when it was caused. Multiple bus architectures, such as Peripheral Component Interconnect (PCI), using intelligent queueing techniques, make these kinds of latencies commonplace.

Another common real-world problem is the infrequent problem that only happens at odd, nonrepeatable times because complex hardware and software systems have deep memory capacities and latencies in registers and state machines in addition to complex logical extent. The hard problems are those that the system stimulates through complex interactions between big application systems and deeply sequenced complex hardware systems. Odd pairings of hardware and software states cause infrequent behavior. Every engineer knows the problem of solving an infrequent system crash.

Another common real-world problem is the hidden dependency. This is caused by underspecified subsystems working together at blinding speeds. It is just not possible to specify every hardware or software component's ultimate use. Environmental conditions like temperature or application loading stress the system to expose these hidden dependencies. The case study described later in this article will show how simple and "perfectly specified" TTL parts can be the root cause of a very nasty operating system crash. This is typical of the hard problems.

Complexity: Growing Exponentially

The root causes of these kinds of vexing problems are hard to anticipate and hard to find. They delay projects and defy schedule prediction for managers. The reason they are so common is the exponentially growing complexity of today's systems. Silicon allows systems with 50,000 to 100,000 gates to be combined with 32-bit CPU architectures with out-of-order execution and large on-chip caches, controlled by 3M-to-10M-byte embedded software systems. These software systems are designed carefully with multiple layers of functionality to promote reuse and maintainability. These software layers, combined with poor code execution visibility because of caches and queues, can create indirection and poor observability during the integration phase.

Fig. 4 illustrates the exponentially growing complexities of today's subsystems. ASIC integration allows entire subsystems to be placed beyond the eyes of probes. Modern CPUs have 10 million transistors and multiple execution units. Embedded systems grow as well. All these technologies are brought to bear at varying levels of risk to be competitive today. Teams must master not only the individual technologies but their inevitable integration. The complexity simply overwhelms. It is

very difficult for engineering teams to predict all the interdependencies between such complex subsystems. Few engineers on a team understand enough of the details of each technology to predict the conditions that will cause a failure during integration. Engineers have to know their design tools: what they do right and what they do wrong. They have to know the system functionality and how it reflects down to the details of their design and the design of the next engineer. They have to know how to unravel the odd sequence of events that causes the infrequent system failures. Most hard problems must be solved by teams of engineers because the complexities of the interactions cross engineering disciplines.

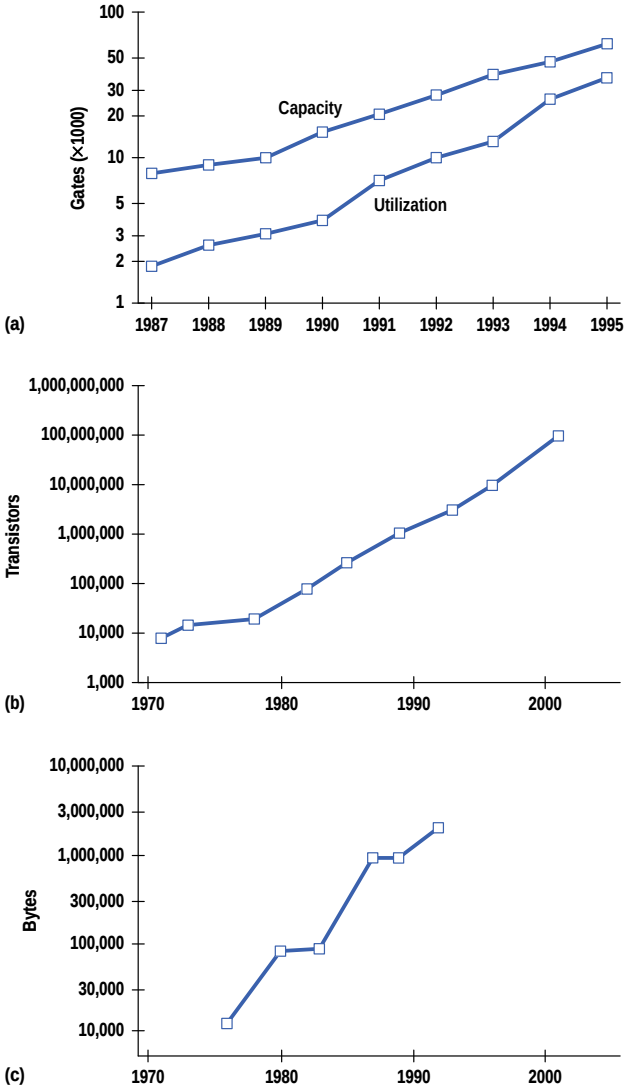


Fig. 4. *Increasing capability and complexity in digital system design. (a) In ASIC and FPGA hardware value added. (b) In microprocessors. (c) In software value added.*

Iterative Debug Loop

The process used by engineers to solve these complex integration problems that have eluded their design tools is illustrated in Fig. 5. The engineers start in the upper right corner, identifying the problem first by clarifying the symptoms.

The next step in the process, isolating problems correctly, is often tricky. Rumors and "old problems—old solutions" confuse. It is very common to jump to conclusions based on symptoms because the engineer once saw a problem like this before and solved it in a certain way. The correct approach is to perform what doctors call a differential diagnosis —describing in detail and with accuracy what has changed from the correctly working condition that could cause a problem, thereby isolating the unique operational sequence. This takes discipline and careful analysis.

The next phase is to discover the hardware or software dependencies that consistently describe the unique configurations and sequences to describe the problem correctly. Sometimes a binary search technique is used: code is sliced apart and patched back together to stimulate the hardware to the unique and small code segment that reliably repeats the problem. Experience and intuition are very important here because hard problems seem to have classic root causes but a variety of

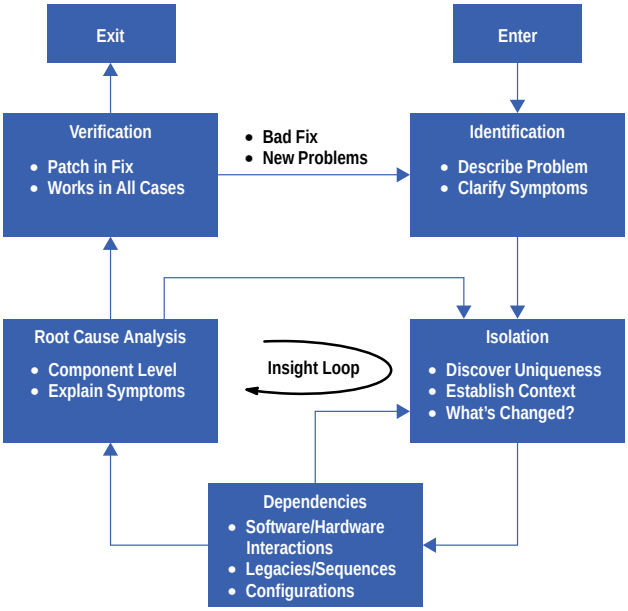


Fig. 5. *Iterative debug loop, showing the insight loop.*

symptoms, discovered over and over by each generation of engineers. Yet the process must be data-driven so that the team is not led to false conclusions.

It is often the case that only one in twenty engineers can correctly isolate the dependencies and sequences. This is because few people are chartered with the task and have the experience to understand the total picture of how a system works in enough pervasive detail to lead structured detective work. The next phase in the debug cycle is finding the true root cause. Infrequent problems, large unobservable ASIC systems, complex software systems, and time pressures make this task difficult. But it is very important to find the true root cause or a simple fix will unravel when some hidden dependency is invoked. A new solution causes another problem. A small change in temperature, voltage, or application loading will cause the fix to stress to the breaking point. Margins will be broken. This is the "insight loop": the structured detective work needed to synthesize the true root cause from confusing and overwhelming data sets caused by vast complexities. The ability to reach the true root cause rapidly and explain the symptoms, dependencies, legacies, and sensitivities is an art form and is deeply insight-based.

The root cause phase leads to the verification phase, in which the suspected fix is patched in and tested to see if it works in all cases. Often design members will go back to their design tools (simulators, etc.) to verify a fix because design tools provide a much richer method for controlling stimulation variables than the real world. It is hard to control a train wreck! The design team works hard to recreate exactly the parameters that caused the problem, and then they look for other related problems and symptoms. Unfortunately, a patched fix often breaks the system in a new way, causing new problems to occur. Or worse, the root cause was wrong and the fix doesn't work. Then the team starts over, having "wasted" time solving the wrong problem or only partially solving the right problem.

Required Tools and Processes

The time it takes to solve these complex problems correctly is highly variable and highly sequential. Therefore, it is very hard to predict or reduce this time systematically. The variability comes from the inner loop, called the insight loop. This is where the engineers labor over the vast complexities and try to connect detailed hardware and software component-level understanding with rich and complex sequences and interactions. The engineering team must traverse from operating system to switching sequences to solve the problems. They need tools that traverse these levels of design hierarchy and the attendant complexity in a way that preserves context but reveals details. The engineer must retain detailed understanding but in the context that describes the problem, always performing the deconvolutions that slice the problem into the subcontext. This is the great challenge of the insight loop.

The other reason for the long integration time is that problems are uncovered sequentially. One problem is solved and then another is uncovered. An operating system must boot before a critical performance bottleneck can be discovered. A clock distribution problem must be fixed before a bus timing problem is tackled, and so on.

To be competitive, design teams must use the latest technologies to get to the price/performance or sheer performance they need. The underlying technology pace causes the complexity, which causes the long integration phase. The design teams that master the integration phase are the first to market, since few companies have a sustainable advantage in underlying technologies. The winning design teams are able to master this phase consistently over several product generations while bringing unique architectures and implementations to important customer needs. Saving 20% of a design cycle may not seem

like much, but doing this four times in a row places the team fully one generation of products ahead of its competition, a crushing advantage in today's electronics markets. This is illustrated in Fig. 6. With product life cycles in the 6-to-18-month time frame, victors and vanquished are determined in just a few short years using this logic. The complexity is the blessing and the curse, the enabler and the nemesis for competitive market entry.

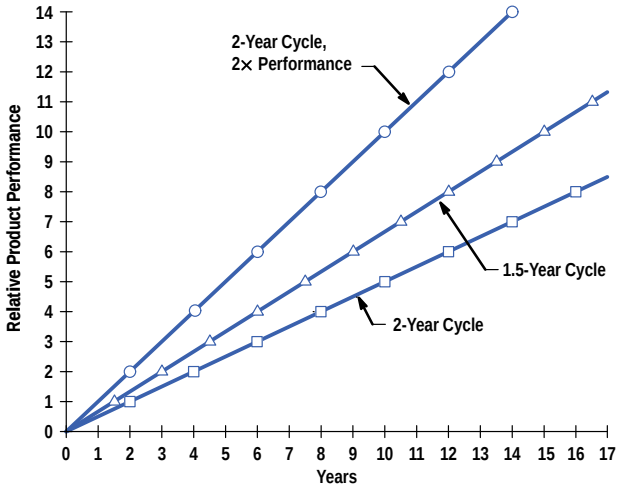


Fig. 6. *The cumulative effect of reducing design cycle times from two years (bottom) to 1.5 years (middle) is equivalent to increasing the performance of each product generation, as exemplified by the top curve.*

Design teams need processes and tools to manage this complexity during the integration phase so that they can, with confidence, take one step more in the use of modern technologies without going too far. Reducing the time to insight in the insight loop is how teams can master complexity rather than have it curse their design cycles with long and unpredictable times. The masters of the integration phase will win over the long term. These are the winning design teams. The tools and processes they need simultaneously provide insight into problem dependencies, reducing the time required to isolate the context, and solve multiple problems at once. Therefore, these tools must retain context while revealing details, and must not hide one problem with the symptoms of another. Without the right tools and processes, teams will suffer through multiple insight loops in sequence.

The Hard Problems: Summary

To summarize the nature of the hard problems and the debug process so that they remain in focus during the case study to follow, the hard problems are listed here in order of common occurrence and degree of pain to the digital design team:

- Hardware-software integration. Different tools used by different engineers. Very complex interactions. Data dependent.
- Displaced locality. Symptom is separated from root cause in location and time.
- Infrequent. Hard to repeat because of deep legacies in hardware and software subsystems.
- Hidden dependencies. Underspecified components find the corner cases.

The tools and processes needed in the insight loop provide the following:

- Design hierarchy traversal. Details are shown in the unique context that causes them.
- Correct isolation. The true and unique context of a problem is found using sequential exploratory logic. Symptoms are explained.
- Correct root cause analysis. The component-level source of the problem is found.
- Verification of proposed fixes, making it possible to move on to the next problem with confidence.
- Accuracy. One problem is not hidden with the symptoms of another.

HP Workstation Case Study

Following is a case study from an HP development to illustrate the above principles. Hewlett-Packard is a heavy user of advanced technologies and tools to bring high-performance products to market. The product developments are typically organized in integrated design teams that effectively design complete products. Therefore, HP is a good place to look for the challenges and best practices associated with the hard problems of complex systems. The case study presented here is described in significant detail in **reference 1**. Here it will be used to illustrate the nature of the hard problems and the iterative debug process (the insight loop) and the need to reduce the time to insight.

The case study is from the HP workstation group. The target system is one of the HP 9000 Series 700 high-performance HP-UX* workstations. The block diagram is shown in Fig. 7. It has a multiple-bus architecture with several custom ASICs,

and was designed specifically for high performance at a low cost. During the final product qualification phase, an operating system crash was found by a test engineer who was verifying various hardware and software configurations. The problem was infrequent and hard to duplicate. The system just crashed at odd times. The test engineer called together a team of engineers with the ability to traverse the design hierarchy with acumen.

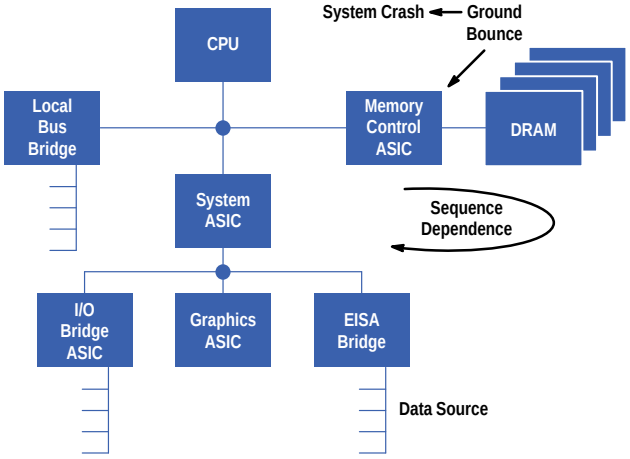


Fig. 7. *Block diagram of an HP workstation showing the location of the ground bounce causing the problem described in the case study.*

The first step was to identify the problem correctly. The team found that the operating system crashed only when the system was performing DMA (direct memory access) transfers from a SCSI drive on the EISA bus. The next step was duplicating and isolating the problem. It could have happened anywhere from the data source on the EISA card to the DMA cycle termination at the CPU bus. To isolate the problem, the data transfers had to be reduced in size and made repeatable. This is where the software team got involved, since they could assist in slicing the data sets into pieces that might recreate the problem. A binary search ensued, cutting and trying different data sequences until repeatability was achieved with a minimum failing sequence (MFS). Even this arduous trial-and-error process yielded only 90% repeatability. The 10% nonrecurrence was ultimately tracked to a DRAM refresh cycle that changed the timing of the sequence that caused the failure (all symptoms must be explained to ensure that a true root cause has been found. Often, teams will neglect inconvenient facts.) The team was now deeply in the insight loop, trying to find the sequences and context that would isolate the problem, leading them to the root cause.

The next step was to probe all data transfers from the EISA card into and out of main memory. For this task, they needed a logic analyzer and multiple, time-correlated, probing points. They looked through many large (1M-byte) data sets to find the failing sequences. Comparing data across the bus bridges is time-consuming, since the latencies are variable as a result of the intelligent queuing techniques employed in the bus bridge ASICs. Fig. 8 shows how the four bus transfers were observed to isolate the failing transfer. Finally, it was found that the MFS was isolated to memory writes and reads across the bus transceivers in the main memory system. The suspect part became a TTL bus transceiver, a common part “well-specified” by multiple vendors. The root cause of the problem was found with a high-speed oscilloscope time-correlated to the logic analyzer trigger looking at the error detecting logic in the CPU, as shown in Fig. 9.

The root cause was ground bounce inside the TTL part. A particular sequence of events placed the TTL part in a situation where a write enable to memory was asserted and directly followed by the simultaneous switching data transfer of the entire byte (Fig. 10). While this seems normal and is indeed within specifications, the closeness of the switching events, combined with the initial state conditions and the weak ASIC drive circuit nuances and memory system capacitive parasitics, caused a 2-to-3V ground bounce inside the TTL part. This caused the part to open up both bidirectional buffers, which caused temporary oscillation, which caused data corruption. This corrupt data, when read back to the CPU many cycles later (several microseconds in a 60-MHz design), caused the operating system to crash.

In the end, all symptoms were explained. Redesigned TTL parts were used and the memory system layout and the ASIC buffer design were improved. The breadth of experience and measurements needed to solve the problem was significant and is a true illustration of the insight loop. Vast data dependencies and odd sequences had to be explored and sliced through to find the MFS. Multiple points in the system had to be probed in time correlation. Data transfers had to be compared across bus bridges. In the end, the entire design hierarchy had to be traversed, from the operating system crash to the ground bounce triggered by data dependent signal integrity. To solve the problem, the entire design team had to be involved: the board designer, the ASIC designers, the software engineers, and the component engineers.

The Prototype Analyzer

The HP Model 16505A prototype analyzer, shown in Fig. 11, is a prototyping tool designed specifically to reduce the time to insight into the common and complex problems plaguing complex digital systems. The prototype analyzer architecture is

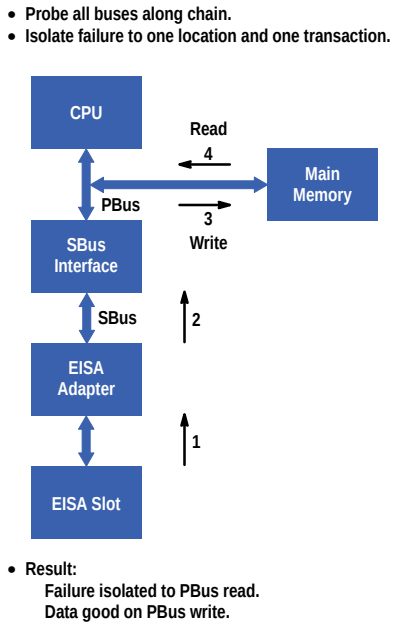


Fig. 8. *Four bus transfers had to be observed to isolate the failing transfer.*

described in ***Article 2***. It is based on a *measurement server architecture*, described in ***Article 3***. It is further based on the key concepts of smart data exploration, through a logically rich postcapture filtering capability, as illustrated in the case study and in ***Article 2***. The key software technology that allows the product to present multiple time-correlated views across the design and measurement hierarchy is called *normalized data* and is discussed in detail in ***Article 4***.

The key concepts of the prototype analyzer product are as follows:

- Insight into problem sources comes from the time sequence dependencies and is facilitated by sequential ordered exploration tools (filters). Therefore, time correlation, using global time markers, across measurements and across measurement domains is critical. Use of color and flexible user-definable data organization is required to aid the deep insights required to solve the hard problems.
- Most important problems are rare in occurrence. Therefore, the tool must quickly and intelligently filter out and eliminate large data sets that don't relate to the problem at hand. Furthermore, since the hard problems are rare, the tool must be able to keep data around to postprocess in every conceivable way to extract the maximum use from the valuable captured data (the cause is in there someplace!). Avoiding the rerun penalty with its associated time delay keeps problem-solving times short, the way engineers need them.
- Hard problems are solved by design teams, not by single members only. Therefore, the measurement server architecture needs to reside on the network and allow offline analysis by multiple members. Fig. 12 shows a common configuration of the prototype analyzer on a network, accessible to every design member. The easy

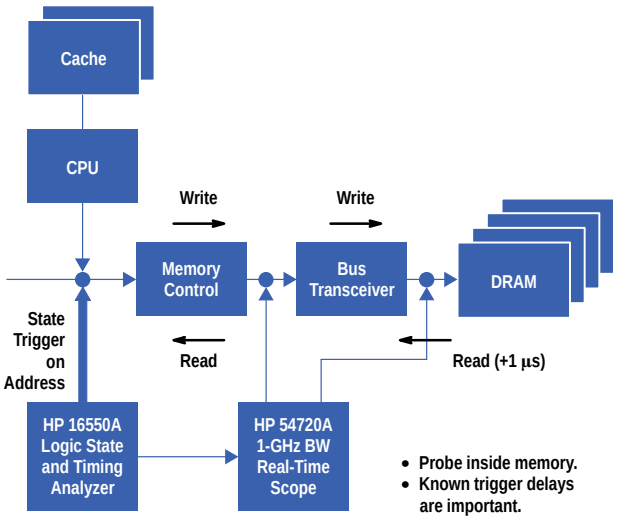


Fig. 9. *The root cause of the problem was found with a high-speed oscilloscope time-correlated to the logic analyzer trigger looking at the error detecting logic in the CPU.*

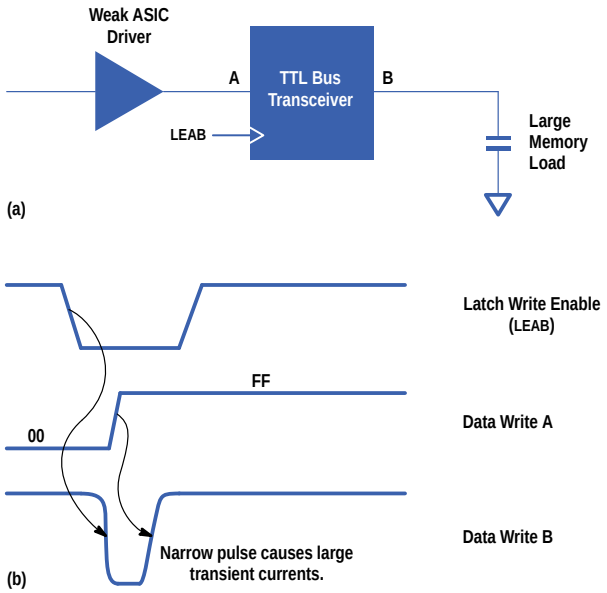


Fig. 10. *The root cause was ground bounce inside the TTL bus transceiver.*

communication of information across a local area network using standard file and server protocols facilitates team solving processes. Furthermore, the tool needs to provide team-member-centric views appropriate to each person and problem. For this reason, the HP 16505A prototype analyzer provides source-to-signals correlation, allowing correlated views from C source code to analog signals, all based on time-correlated measurements. The 16500B logic analysis system supports all these measurement modalities.

- Designers have vast intellectual property vaults in their design databases. These form the basic raw material for an engineer's analysis needs. They must be viewable and comparable to the measured data to accelerate time to insight. A file in, file out capability facilitates the easy movement of the critical data sets to where they can be manipulated with the best postprocessing tools available. Engineers have their favorite tools and the prototype analyzer accesses them all through the X11 Window interface.
- Large data sets are difficult to use to find the elusive root cause of a problem even if the problem has been captured in the data set. This is because the user is overwhelmed by the magnitude of the data and has to wallow around in it. Overview tools are needed that transform the low-level digital data and software code to a higher level where users can see the data that lies outside the norm. The human eye easily distinguishes such

Fig. 11. *HP 16505A prototype analyzer.*

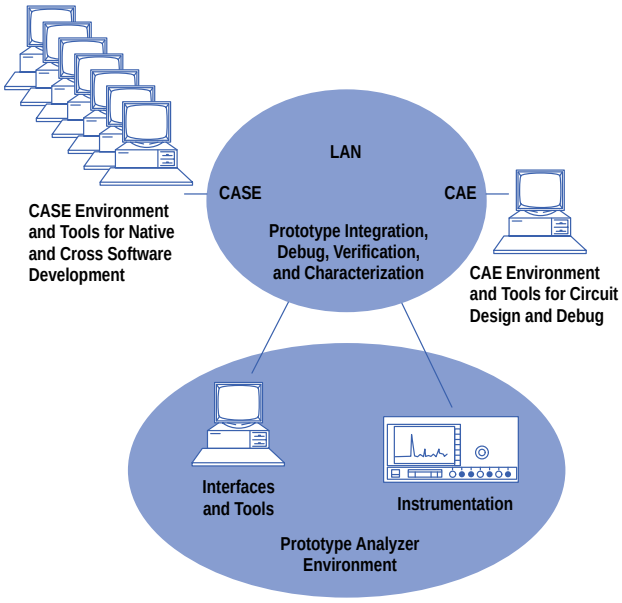


Fig. 12. *Team profile and design environment for digital system design.*

signals. The user wants to see changes in the data rather than just the raw data. This higher level is called the *modulation domain* and is a specialty of the prototype analyzer with its ability to create new labels from combinations of raw incoming data. An example of this is shown in Fig. 13. The analyzer is used to extract setup times on a bus from the raw incoming data and clock signals. A new label is created from the difference between the data and the clock. This label can then be displayed as it changes over large time constants using the chart tool. A user can easily view a million setup times over several milliseconds. Dramatic changes of setup time at periodic intervals tell the engineer that the system timing margins are being stressed on software time constants, perhaps as a result of a DRAM refresh or the servicing of a periodic interrupt. This display of the modulation domain is unique to the prototype analyzer among digital analysis tools and illustrates the power of postprocessing large data sets in overview displays to guide an engineer towards the elusive integration problem.

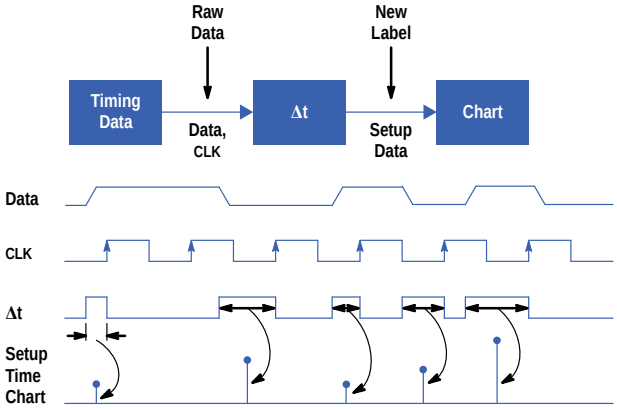


Fig. 13. *In the modulation domain, changes in the data, rather than just raw data, can be observed. In this example, the prototype analyzer extracts setup times on a bus from the raw incoming data and clock signals and then displays changes in setup time using the chart tool.*

Prototype Analyzer Solution

Now we revisit the HP workstation case study to see how the prototype analyzer can solve this problem much faster. The problem in the target system only occurred when simultaneous switching occurred on the data lines with write enable followed quickly by data write changes (refer again to Fig. 10). Fig. 14 shows how the prototype analyzer can be used with postprocessing filters to isolate the case much faster. The prototype analyzer is able to filter out states that follow specific sequences. In this case, Filter 1 in Fig. 14 could be set to view only the data for cases in which all states changed from ones to zeros (FF to 00) or vice versa. The design team suspected simultaneous switching noise in their design and this technique would have quickly given them the opportunity to explore this possible root cause without rerunning the target system. Any logic analyzer can find the first case of FF followed by 00, but only the prototype analyzer can scan an entire

one-megasample record, find *all* the cases where this is true, and *display them all* with the original data, time correlated with other views, such as analog data. Showing all the data, rather then just triggering on the first occurrence of this transition, allows the design team to find the one in a million times that this transition causes the failure, rather than having to keep retriggering the logic analyzer, hoping to find this case. The next step is to filter again using a timing filter (Filter 2 in Fig. 14). In this case, the design team is testing the theory that close timing between LEAB and data switching on data line A causes a narrow pulse, which, when driving the high capacitive load of the memory system, causes high ground currents. Only the prototype analyzer can find *all* of the occurrences of these conditions *and* only those that meet the Filter 1 criteria. This consecutive filtering allows engineers to build isolation techniques into their analysis to test their theories, rather than having to recapture the data and incur the retrigger and recapture time penalties. Using the oscilloscope built into the HP 16500B logic analyzer, they can capture with time correlation the lines that are being corrupted by the simultaneous switching noise and ground bounce. The time correlation allows engineers to retain the state and switching context that created the failing conditions. The waveform and lister tools merge the analog and code contexts to form a new measurement domain.

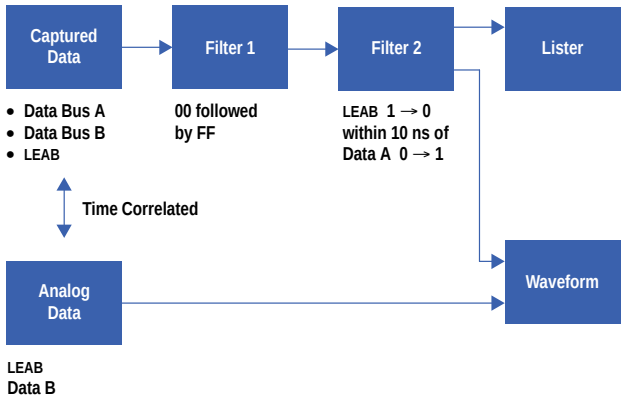


Fig. 14. *Solving the case study problem with the prototype analyzer.*

In summary, the prototype analyzer can be used to isolate the root cause of this problem much more quickly because it has sequential postprocessing filters, allowing analysis of entire data records at once. These kinds of problems are the cause of many delayed developments in advanced digital systems. The prototype analyzer tool is a way to solve them more quickly, minimizing time to insight for the design team.

Conclusion

Fig. 15 shows the use model for the prototype analyzer, redrawing Fig. 1 to show how the risk in the concurrent engineering process can be managed with a single digital analysis tool platform to unravel the complexities of modern digital systems and deliver to design teams the tools and processes needed to master the integration phase to competitive advantage.

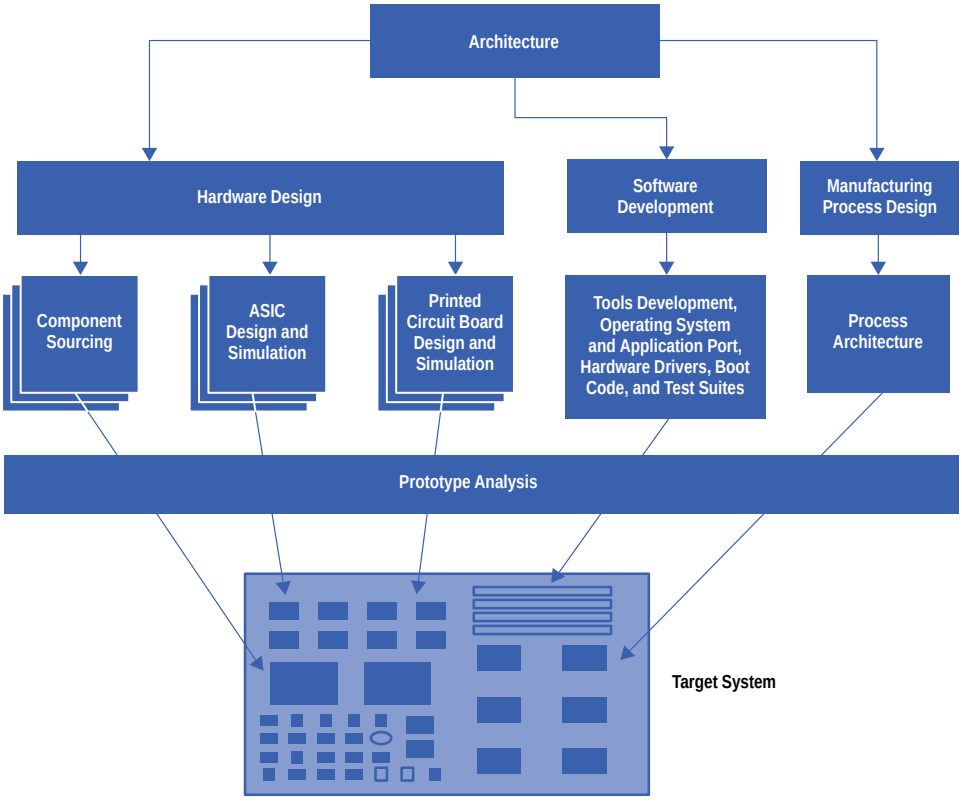


Fig. 15. *Updated system design process using the prototype analyzer.*

Acknowledgments

Pursuing the hard problems and delivering superior time to insight to digital design teams was the work of many people at the Colorado Springs Division. Greg Peters, Steve Shepard, Jeff Haeffele, Chuck Small, Jeff Roeca, and James Kahkoska were the leaders. James is the creator of the prototype analyzer concept. His unique ability to understand customer needs and translate these insights into compelling products and architectures was the magic behind this product. Fred Rampey, Dave Richey, and Tom Saponas sponsored the market and product initiative. Frank Lettang and Rob Horning of the Computer Organization were the case study agents and their easy access was critical to deep understanding.

Reference

1. *Hewlett-Packard 1993 High-Speed Digital Symposium Digest of Technical Papers*, pp. 10-1 to 10-20.

Prototype Analyzer Architecture

The HP 16505A architecture allows multiple concurrent views of acquired logic analysis data. Markers on all views are correlated. The user only needs to place the marker on one view and the markers on the other views automatically relocate. Thus a stack anomaly in one view can be immediately correlated with the software routine causing the violation.

by Jeffrey E. Roeca

The HP 16505A prototype analyzer is the next-generation user interface for logic analysis. It is an X11/Motif application running on an HP 9000 Model 712 PA-RISC workstation directly tied to the HP 16500 logic analysis mainframe. This system is a managed, or closed, system. From boot until power-down this system is solely dedicated to extending the prototype debug paradigm from the nine-inch touchscreen of the HP 16500 to a multiple-window, high-resolution interface.

The design uses the X11/Motif graphical user interface (GUI). This was done to accomplish several goals. First, using an existing state-of-the-art GUI allowed us to focus on our own contributions. Second, with the distributed X11 interface, we can make the application available over a network to any X-compliant computer, including PCs. The software was written in C++ for the Model 712 workstation platform. This gave our application affordable MIPS, which were needed because debugging one-megasample logic analyzer traces across hundreds of channels will tax any computer.

Architecture

Fig. 1 shows a simple HP 16505A setup. There are three tools on the workspace. Two tools are analyzer machines, and one tool is a display tool. This simple configuration reveals the essential functions. Tools can be placed into a data flow and run. The analyzers are tools that probe target hardware. They sample real-world data and the display tools independently view that data.

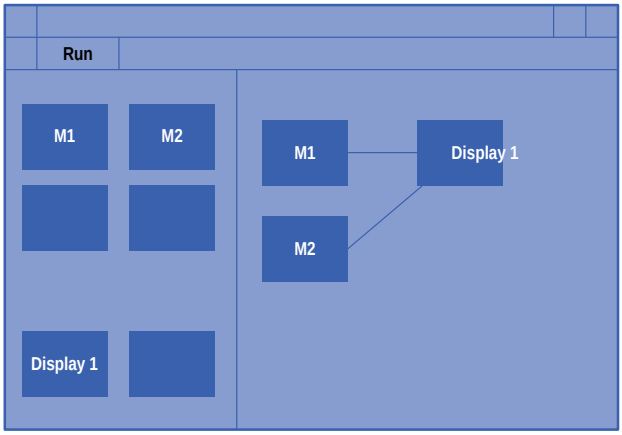


Fig. 1. *A simple HP 16505A prototype analyzer setup. Available tools are selected from the toolbox at left and dropped onto the workspace, placed into a data flow, and run. Here there are three tools on the workspace: two analyzer machines (M1, M2) and one display tool.*

To deliver this functionality we have partitioned the architecture into the following blocks:

- Tools. Tools are the atomic unit of value in the HP 16505A. All meaningful user interaction is performed within a tool definition.
- Acquisition Tools. The acquisition tool class is derived from the tool class. As such, it plays in the data flow workspace. The reason for making this a derived class lies in the way we connect or relate acquisition tools to the HP 16500 mainframe.
- Data. All tools export and digest a common unified data format.
- Workspace and Graph. The workspace is the visual programming GUI. It allows tools, represented by icons, to be connected to a data flow, represented by lines. The graph is the ordered list of tools in the flow.

- Run Management. This is a state machine that pumps data through the data flow by executing the graph.
- General Support. This is basically control or “glue” logic.

Data Flow

Let’s look at an example data flow through the architecture.

When the user powers up the HP 16505A we begin an HP-UX* operating system boot procedure. Most of the boot code is provided by the Model 712 ROM, but we adjust the daemons and hardware configuration for this application. Once the operating system is minimally configured we call the session manager routine, which never returns. The session manager handles software updates and spawning of the main application. This is also where the workstation is powered down. (The Model 712 workstation has a front-panel power switch, which forces the HP-UX file system to be written to the SCSI drive before actually cutting power. This prevents the user from crashing the file system. Unplugging the workstation can be problematic for the file system, however, so the session manager has a power-down button to remind the user not to simply unplug the product.)

The real application begins as main.c on HP-UX. We begin by creating an application context under the X Window System with a call to XtVaAppInitialize(). The entire application runs as an X11/Motif application. From the GUI mouse/keyboard callbacks through the acquisition control of the HP 16500, all software is run as an X11 application callback.

The HP 16505A system software offers a handful of services to tools. These services are collectively gathered beneath a structure called the frame. The frame is one of the few globally available pointers. Tools can access these system services by referencing frame->service. The frame services are:

- ResourceMgr. Colors, fonts, cursors, path names.
- RunMgr. State machine for stepping the tools through runs.
- FileMgr. Storing and loading of configurations.
- HelpMgr. Help screens.
- GeometryMgr. Pixel drawing management for the workspace.
- Symbols. Definition and display.
- FrameMarkerMgr. Global marker synchronization.
- FrameMessageExchange. Mail service between nodes on the graph.
- TbcMgr. Manager of toolbox icons.
- InstrumentMgr. Manages a list of instruments.
- AdminMgr. Networking and other HP-UX management.
- PrintMgr. Print services.

Next we perform a directory search for instruments and tools. All tools are compiled into shared libraries. These libraries exist beneath a well-known directory. We traverse the directories beneath this, searching for valid tools and instruments. A valid directory consists of a shared library, a resource file, and a pixmap. The load algorithm is generic and does not need to be rewritten when a new tool is created. Simply creating a directory with these three files is sufficient for this algorithm to attempt to load. Tools are noted, but are loaded only when placed onto the graph. This can be observed as a slight time delay in the creation of the first tool of a given class. The benefit of demand loading tools is that the memory is kept free for deep trace analysis.

Unlike tools, instruments are loaded when they are found. An instrument is an HP 16500. Within the HP 16500 there can be up to ten modules, but there are not usually so many and it is efficient and simple to load them at power-up.

Next the workspace is created. All of the loaded and noted tools are represented by icons in the toolbox.

Next we start the file system, remote procedure call mechanism, messaging services, and some other details. Finally, we submit all processing to the X Window System by calling XtAppMainLoop(). The application goes dormant until either a remote command or a GUI event activates a software callback. Even the remote interface becomes an X event as we receive remote procedure calls and simply pass them along to the X Window System. When XtAppMainLoop returns, the HP 16505A returns to the session manager.

Tool Design

Tools and their design are a major factor giving this platform room to grow. A tool is a C++ class with a small set of pure virtual functions. These functions are:

- getRev. What release of software (e.g., 1.20)?
- about. Returns general tool information in ASCII format.
- save. Saves the tools configuration to a file.
- load. Loads the configuration from a file.
- addToPopup. Call from the workspace frame to add to a menu.

- setName. Call from the workspace to change this tool's name.
- raiseAll. Open up or raise the windows.
- preExecute. Validates the internal state as legal to execute.
- validateConfig. Validates the configuration before loading.
- execute. Receives a new data group, returns a data group.

These tool functions are mostly mundane calls to get, set, save, or load static information within the tool definition. All real tools must supply these functions, and the system glue is written with calls through these functions from tool pointers kept in the graph.

Only one function deals with the primary objective of a tool—passing data—and that is the execute function. This function is declared as follows:

```
virtual DataGroup * execute (const DataGroup&) = 0;
```

A tool receives a call and is passed a pointer to the data and labels. The tool can then interpret, filter, or create data as needed. All that is required is that the tool return a pointer to the output data group. This is very simple and powerful. Compliance with the execution protocol does not require any coupling between tools.

From this definition all real tools are derived. Fig. 2 shows the current tool classes in the HP 16505A.

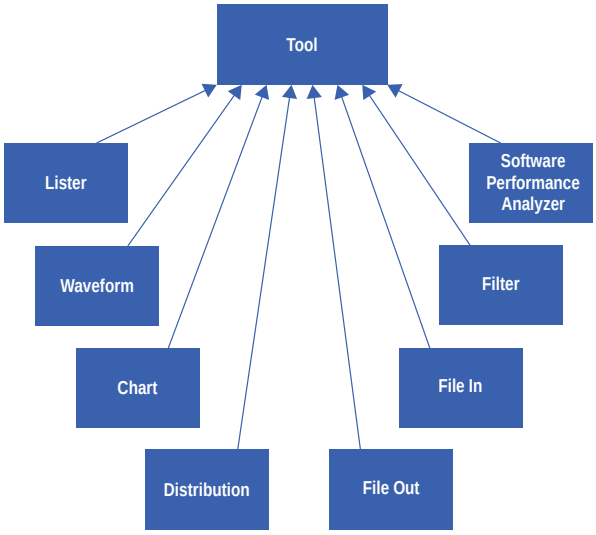


Fig. 2. *Tool class hierarchy.*

Acquisition Tool Design

The workspace manages an ordered list of tools. Consequently, a logic analyzer must be represented as a tool to be placed on the workspace. According to HP 16500 legacy, the fundamental working unit of a logic analyzer is a *machine*. A single HP 16500 card, or module, can have two machines. In the HP 16505A, each of the module machines is represented as an individual tool in the toolbox. The abstract C++ class for these machines is called a probed source. Probed sources must address multiple complications. One is communication with the HP 16500 module. Another is handling a workspace presentation that represents probed sources as independent tools, while behind the scenes both machines of a single HP 16500 module are joined at the card. Another complication is coordinating the remote acquisition of data. Fig. 3 presents the acquisition tool hierarchy, showing the basic composition of probed sources and their relationship to their module (card) and the HP 16500 (enterprise).

The class that is placed onto the workspace is one of the machine classes. Because these classes are derived from the probed source class, which is derived from the tool class, they can be inserted into the workspace. As previously stated, a logic analyzer card can have multiple analyzers, or machines. Accordingly, each machine has a relationship with its parent card class, which is derived from the abstract card class. Cards in turn are owned by the acquisition enterprise. The enterprise class is derived from the abstract instrument class. Within the instrument class exists the transport knowledge, which happens to be SCSI.

Encapsulating the SCSI transport in the instrument class allows us to port the transport layer with a minimal impact on the rest of the software. This was a benefit in the initial stages of development when the SCSI interface did not exist and we programmed the HP 16500 over the HP-IB (IEEE 488, IEC 625).

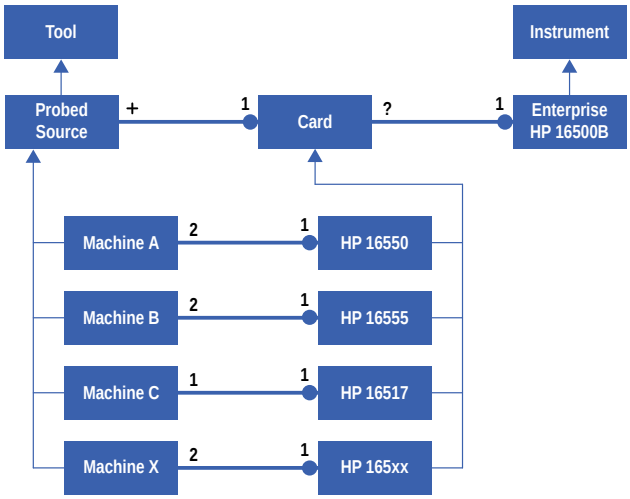


Fig. 3. *Acquisition tool class hierarchy. The dots and numbers are Booch notation for N-to-1 relationships. For example, an HP 16550 card has two machines.*

By using the existing ASCII programming strings of the HP 16500 we are able to acquire data in an existing stable platform. By using the SCSI transport we minimize the latency of transfer from the HP 16500 to the HP 16505A.

Data

Data is different for each analyzer card in the HP 16500. Historically this slowed development by giving rise to custom, hardware-specific algorithms to render that data. Behind similar HP 165xx interfaces may be completely different software. Fig. 4 shows the transform that we apply in the HP 16505A prototype analyzer. As data is uploaded from the unique hardware, it is transformed into a common, or normalized format. This normalized data is what all tools downstream from the analyzer source operate upon (see ***Article 4***).

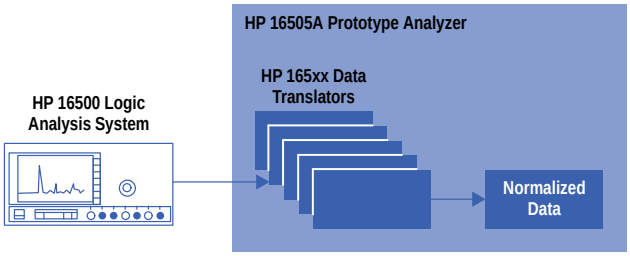


Fig. 4. *Data is different for each analyzer card in the HP 16500 logic analysis system. As data is uploaded into the HP 16505A prototype analyzer from the unique hardware, it is transformed into a common, or normalized format.*

Workspace and Graph

The primary role of the workspace is to construct the data flow relationship that is executed during runs. The data flow is constructed with tools and data. The GUI provides a toolbox that contains icons representing all available tools. A user drags an icon representing an available tool from the toolbox and drops it into the workspace. Data flow is represented by lines connecting the icons or tools. Each tool that is placed onto the workspace is entered into the graph. The graph is an ordered list of containers. A container is a structure that holds information about real instantiated tools. Any software function can execute the graph, which results in an ordered delivery of containers to the requesting function. The function can then call the appropriate tool function through the container.

The graph has a simple set of rules determining the order of execution of the list of tools:

- Only tools that are on the workspace are executed.
- Orphan tools—tools that have no data connections—are executed first.
- Top-level tools such as logic analyzers and oscilloscopes are executed next.

- The graph does not transition from one level to the next until all tools at a given level are complete. This ensures that any tool that receives data from multiple sources will have the data available when the tool is executed, since all parent tools must have completed before the tool is called.

Fig. 5 shows a workspace representation with three analyzer machines, two filter tools, a lister, and a waveform tool. M1 and M2 are merging their independently acquired data together into a single data group and then passing that merged data group through the Filter1 tool to the lister. M3 is an independent data group that passes through Filter2 into the waveform tool. There are highlighted boxes labeled level 1, level 2, and level 3. These boxes indicate the order of execution of the graph. Level 1 tools are the data sources, or analyzer tools. This level is executed first. No tool in level 2 or level 3 will execute until all level 1 tools are complete. Once level 1 is complete then the level 2 tools will execute. No tools in level 3 will execute until level 2 is finished. This is a general rule: level N–1 must finish before level N executes.

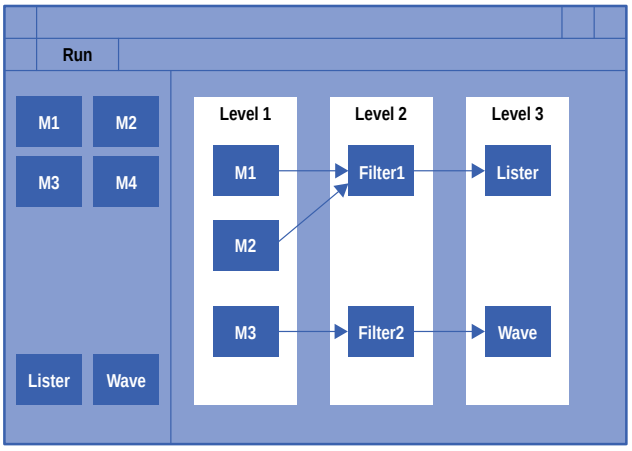


Fig. 5. *A workspace representation with three analyzer machines, two filter tools, a lister, and a waveform tool.*

Run Management

The run manager is the software that pumps data through the graph. It is a state machine that executes the graph. Remember that executing the graph is an ordered, level-by-level event. This ordering allows for all probed sources to be configured, started, and uploaded as a coordinated group.

Fig. 6 shows the states and their execution cycle, together with the entry points into the state machine. The states are:

- PreRunDownload. The HP 16505A probed source software maintains internal configurations. The GUI modifies the configurations and the changes are stored in the HP 16505A without remotely updating the HP 16500. When Run is pressed, a probed source is requested to download to the HP 16500 the current HP 16505A probed source configuration.

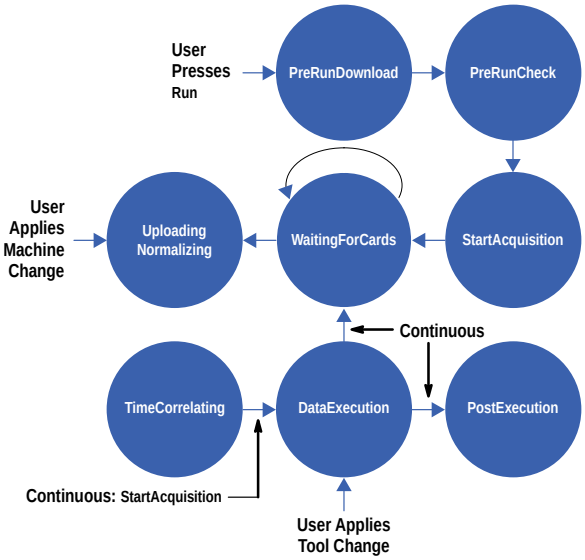


Fig. 6. *Run manager state diagram.*

- PreRunCheck. A given HP 16500 card may have an illegal setting that prevents a run from proceeding. This state allows the HP 16500 status to abort a run.
- StartAcquisition. The HP 16500 is started. The current release of the HP 16505A only performs group runs. A group run is one that time-correlates the trigger events from participating modules.
- WaitingForCards. The HP 16505A polls the HP 16500 run status waiting for the run completion.
- UploadingNormalizing. Each card is called serially to upload and normalize the acquisition data.
- TimeCorrelating. After each card is normalized the system acquires the timing correlation values and updates the normalized data.
- DataExecution. The normalized data is now passed through the graph to the configured tools. Each tool called will render the data and update its display.
- PostExecution. The HP 16505A does some cleanup.

The graph is executed during each state at least once. A given state may only choose to communicate with a tool if the tool is a probed source. A state may execute the graph multiple times. For example, WaitingForCards is a state that can take a significant amount of time, depending upon the repeatability and cycle time of the trigger specification. The run manager could be stuck in this state forever. The logic within this state executes the graph. If the cards are not complete then the run manager sets an X timeout and exits.

The setting of the X timeout is significant. It is essential that the GUI is allowed to run to service potential stop events. At the first release there is no remote control of the HP 16505A, and a GUI event on a stop button is the only way to abort a run.

Example Control Flow through the Software

Fig. 7 shows a simplified control flow for acquiring status during an acquisition of data. A user has previously pressed the Run button, which starts the run manager state machine. In this example the run manager has been called (from an X timeout) and is processing the WaitingForCards state. This state, like all run manager states, calls the graph execute function. This causes the graph to cycle through all of the tools on the workspace. The run manager calls tools that are probed sources requesting their status. Now is where the mapping of analyzer machines, or probed sources, to cards becomes important. The HP 16500 remote control is ordered by modules. Within a module, communication is with an analyzer machine. The HP 16505A has requested the run status of an executing machine. This request is passed to the parent card class. The card knows which HP 16500 slot to select. The card also knows what messages to apply to obtain the status. These ASCII HP 16500 command strings are then passed to the enterprise, or HP 16500B class. This is the class that inherits from the abstract instrument class. The SCSI transport sends the ASCII programming instructions down to the HP 16500B and there they are parsed and interpreted.

Fig. 8 shows a similar simplified block diagram with the normalized data, tools that are not probed sources, and the tool support blocks added. Ignoring the fine details of the myriad of support tools, correlated markers, and so on, one can obtain a good understanding of the HP 16505A architecture from this figure. The frame object has a run manager object which controls the state machine for acquiring and pumping data through the tools. The workspace has an ordered list of instantiated tools and controls the execution order. Tools are either independent tools or probed sources. If they are probed sources they communicate through their parent card through their instrument through the transport layer. Probed sources generate normalized data, and all other tools can use, modify, and generate normalized data.

Examples of Visualization

The primary value of the HP 16505A lies in its ability to help the user visualize data. The architecture was defined to allow multiple concurrent views of the acquired data. By slicing and dicing data in different ways, fault isolation becomes easier. Fig. 9 shows an HP 16505A setup demonstrating how visualization in multiple views promotes rapid time to insight. Software stack corruption is a difficult problem to solve. Usually a corrupt stack does not directly reveal itself. More often, the product crashes far away from the actual write violation. In this example a 68332 microprocessor running software has been sampled. This data has been displayed both unfiltered and filtered. The filtered data is shown in ASCII format, as a chart, and as a distribution. The filtered data in chart mode shows the stack as a function of time. The filter is a simple range between the low and high boundaries of the stack. Visually one can see an anomaly in the stack space. In this example the write taking place beneath the G1 marker in the Stk vs. Time window is in error. By looking at the ASCII listing of the unfiltered data in the Raw Trace window we can see exactly what software was executing at the time that this erroneous write to the stack space took place.

The HP 16505A allows these multiple views—the lister and the filtered chart—to be displayed concurrently. Within the HP 16505 display tools we have correlated markers. The default behavior of a display tool is to track the placement of a marker automatically. The user only needs to place the marker on the visual stack anomaly, and the lister automatically relocates itself to reveal the software routine in violation.

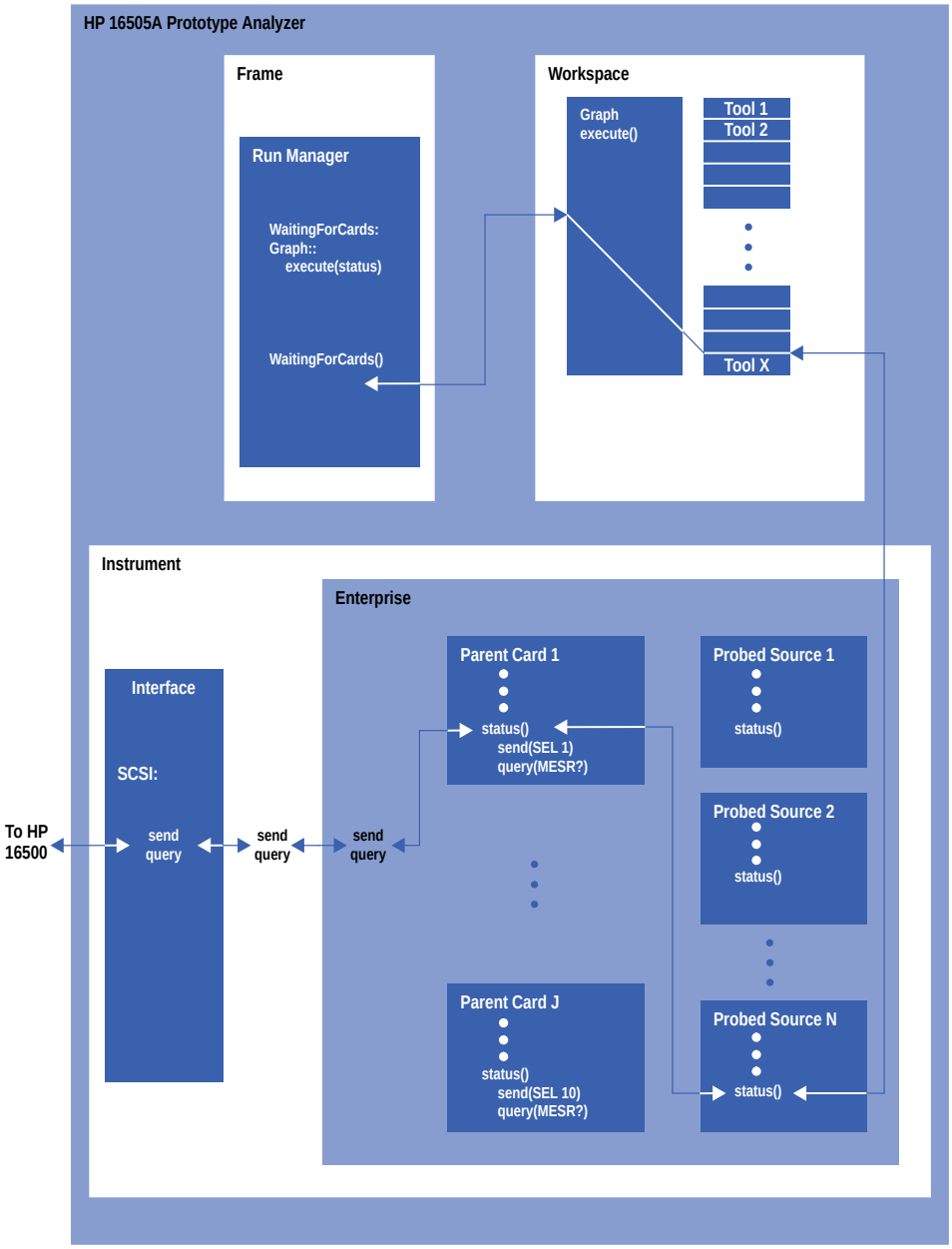


Fig. 7. *A simplified control flow for acquiring status during an acquisition of data. The run manager state machine has been called and is processing the* WaitingForCards *state. This calls the graph execute function, causing the graph to cycle through all of the tools on the workspace.*

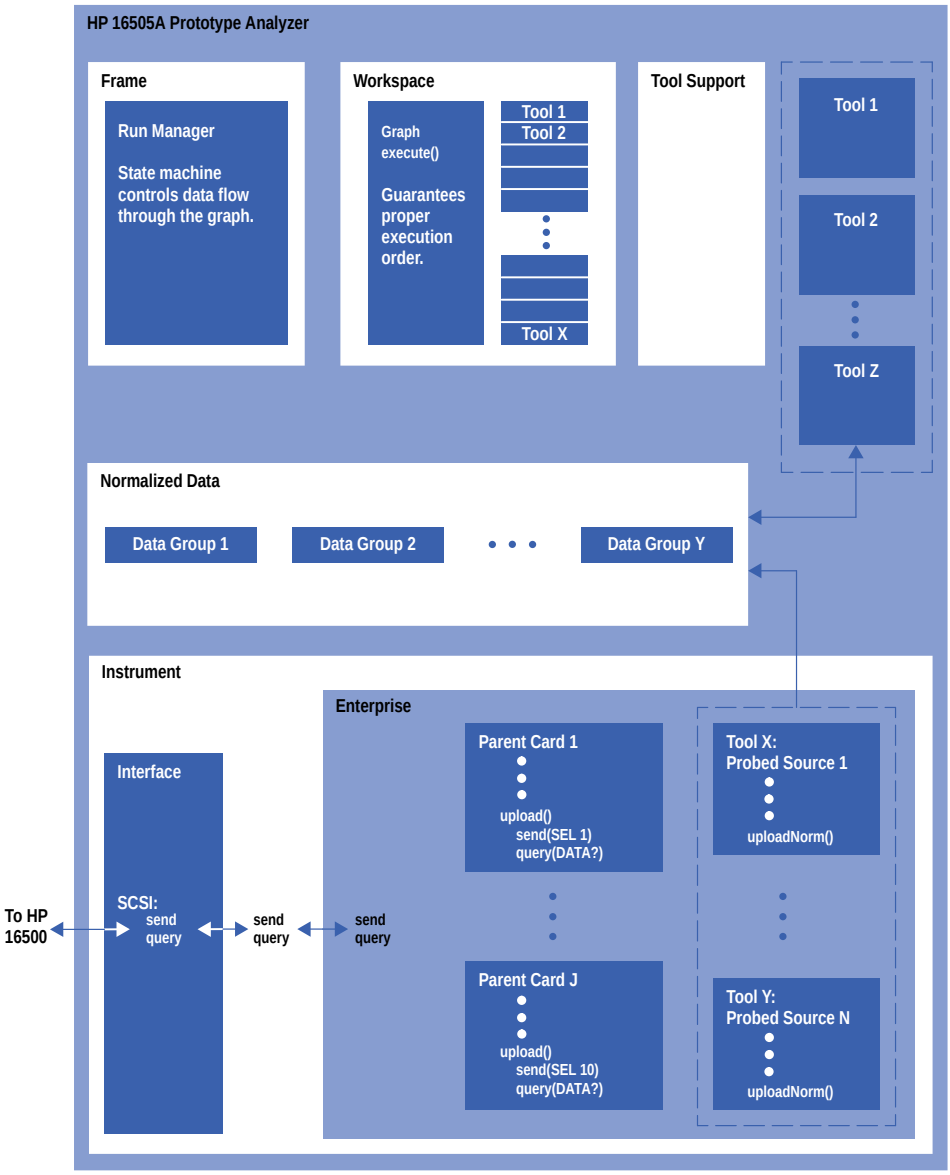


Fig. 8. *Simplified block diagram similar to Fig. 7 with normalized data, tools that are not probed sources, and tool support blocks added.*

Fig. 9. *An HP 16505A setup demonstrating the diagnosis of a software stack corruption problem. The write taking place beneath the G1 marker in the Stk vs. Time window is in error. By looking at the ASCII listing of the unfiltered data in the Raw Trace window we can see exactly what software was executing at the time that this erroneous write to the stack space took place. The user only needs to place the marker on the visual stack anomaly, and the lister automatically relocates itself to reveal the software routine in violation.*

Acknowledgments

The author would like to acknowledge James Kahkoska for defining, designing, and driving the product idea into reality. I would also like to thank Mark Schnaible for helping distill the HP 16505A architecture down to a few slides.

Determining a Best-Fit Measurement Server Implementation for Digital Design Team Solutions

Prototype analyzer customers wanted fast throughput, quick answers, a turnkey solution, an affordable base system price, connection to diverse open-systems networks and platforms, and interfaces to a wide variety of tools. An encapsulated measurement server architecture based on a dedicated workstation and a SCSI II interface best fit the requirements.

by Gregory J. Peters

Implementing an analysis and display architecture for the next generation of HP logic analyzers presented opportunities and challenges for the cross-functional design team assigned to the task. Freed from the boundaries of commercial real-time instrument operating systems and needing to offer a flexible and high-performance window-based interface, the project team took a long look at both technologies and the total envelope of costs before settling on an encapsulated measurement server architecture based on an HP 9000 Series 700 workstation.

The goal of the HP 16505A prototype analyzer team was to develop a breakthrough analysis and display environment for digital design teams involved in the integration and debug of prototype systems. The resulting product is a combination of the best of traditional instrument and system benefits. The turnkey prototyping system provides the performance and flexibility demanded by leading-edge digital design teams but its low total system cost appeals to the smaller teams as well. The tough decisions made very early in the product definition phase have paid off. The product meets market needs and has a low cost of sales and support envelope.

Genesis

The genesis of the prototype analyzer occurred with the realization that digital hardware developers, who are traditional logic analyzer users, were spending an ever higher percentage of time using electronic design automation (EDA) tools, such as schematic capture and simulation software packages. The use of workstation or PC-based EDA tools came at the expense of time spent in the lab making measurements on prototype hardware.

The workstation or PC was fast becoming home base for designers. Traditional test instrumentation was in danger of becoming less relevant to the design team, in large part because it didn't interface with EDA tools. The question the team pondered was how future real-time measurement systems would interface with the designer's workstation home base.

This concern generated an investigation into workstation connectivity, and the prototype analyzer project was born. The development team at first conducted research with hundreds of customers worldwide, in different industries and with different measurement needs. The results of the investigation drove the definition of the prototype analyzer measurement server implementation.

Measurement Servers

Measurement servers are defined as a class of solution that combines physical measurements with client/server computing. Real-world measurement data is captured, analyzed and shared with users and other tools over the network using a variety of open-systems standards.

Measurement servers use widely accepted networking standards such as Ethernet, the X Window protocol, and the Network File System (NFS) to enable customers to control and view measurement instruments and get easy access to measurement data. Standard data formats are used to interconnect the measurement server with other applications. Specialized host-based software is not required. Measurement servers can provide anywhere, anytime access to prototype measurement data via today's ubiquitous networks.

Client/server computing enables each component of the system to specialize in what it can do best. The locations and definitions of the interfaces between components are determined by the nature of the application and the network topology. A well-recognized example of a client/server application can be found in the EDA industry: a simulation engine runs on a large compute server while the simulation interface runs on the engineer's desktop. Application developers must pick the appropriate points to split apart an application to maximize a product's performance and value.

Measurement server architectures vary depending on the scope of the solution. Some architectures are completely encapsulated inside a box, while others are split apart or distributed (see Fig. 1). Identifying the appropriate point to split the application apart is critical. Splitting an application at the wrong place can limit response times or cause nondeterministic behavior. Encapsulating the wrong functionality into a system can increase the cost of the system or the cost of sales without increasing revenue. Encapsulating too little functionality may result in trivial or redundant products.

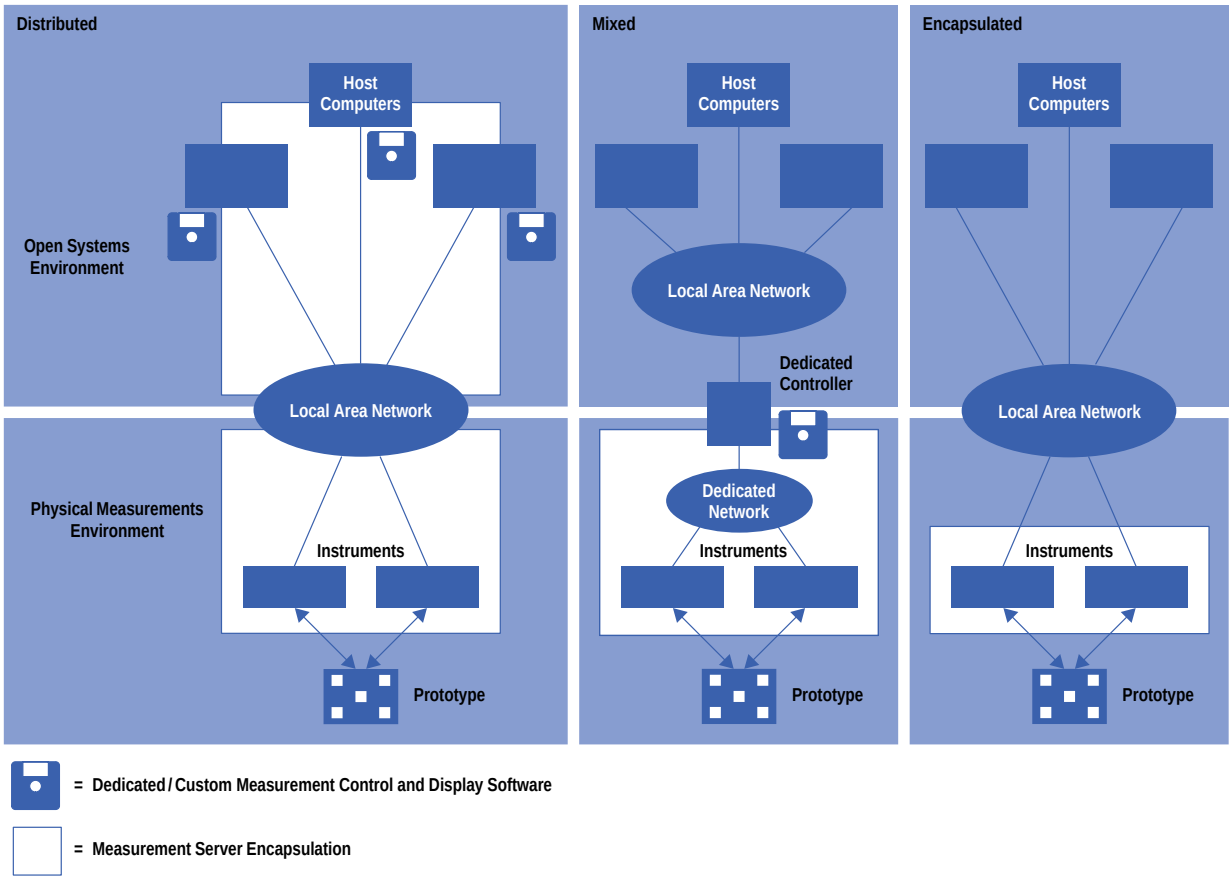


Fig. 1. *Three possible measurement server architectures: distributed, mixed, and encapsulated.*

The full use of the client/server paradigm is the differentiating aspect between traditional instruments and measurement servers. Traditional instruments can also be connected to computers via the HP-IB (IEEE 488), but this connection requires special interface[1] and control hardware and software on the host computer. Additional analysis and display software is generally custom, and can be time-consuming to develop and debug. The localized nature of the HP-IB limits its penetration into open computing environments and the need for specialized control and interface software puts HP-IB-controlled instrumentation at a distinct disadvantage in the networked design environment.

Architecture Selection

To obtain the best price and performance trade-offs, the proposed solution must be evaluated closely to determine the optimal measurement server architecture. The three examples given below highlight some of the issues that must be addressed. The prototype analyzer development team was able to gain valuable experience from reviewing each of these product forms.

An example of a distributed measurement server architecture can be found in the HP B3740A software analyzer. This software package provides software developers with a means of viewing real-time microprocessor traces referenced to source code. The HP B3740A runs on several workstations and on PCs.

The host-based nature of this application was derived from a need to provide source-viewing functionality to a large number of design team members who timeshare a single real-time instrument (the HP 16500B logic analysis system). In this case, the small amount of data transferred over an Ethernet link between the HP 16500B and the HP B3740A does not impact performance. Performance here is measured as update rate.

As a point application, this product does not require extensive support. If the same approach were taken for several applications, however, the sales and support burden could limit broad market penetration. In particular, maintaining

host-based software on a platform that has frequent operating system revisions is not only time-consuming but also results in a low return on investment to the development team.

HP VEE, a visual programming and test development environment,[2] best illustrates a mixed measurement server architecture. HP VEE runs as a host-based program on PCs and workstations. This product enables engineers to quickly develop test programs that control a wide variety of instruments over the HP-IB on a standard computing platform such as a workstation or PC. The key benefit of this approach is the flexibility afforded both solution providers and end users. A choice of supported platforms means customers can use a platform they are familiar with. An open-system development environment enables value-added engineering, upon which solution providers can build custom applications.

A mixed measurement server architecture has drawbacks, however. Maintaining a reliable interface between the host-based software and the measurement instruments through operating system revisions or changes in network protocols can be challenging. Setup and maintenance of a distributed system require significantly more effort than a turnkey instrument. A dedicated Ethernet network can impose bandwidth barriers that severely limit measurement throughput.

The HP 16500B logic analysis system with an HP 16500H,L interface module is an example of an encapsulated measurement server. This system can provide both local and remote control and viewing via the X Windows protocol. The turnkey nature of this product makes installation and setup simple. Since all data processing takes place in the instrument, only X Window calls are transmitted over the Ethernet. This helps keep the update rate fast over most network topologies.

While the support burden of an encapsulated measurement server is low, it comes at a price. User-defined measurement sequences and extended data analysis and reporting are not provided. Customers must make a connection to the HP 16500B and import data and screen images into a general-purpose computer to perform these tasks.

Considerations such as customer expertise in open systems, support costs, and the use model for the product must all be factored in when choosing an architecture for a measurement system. Trade-offs must be made to fit the needs of the market and target customer. Table I outlines the advantages and disadvantages of the three measurement server architectural approaches.

Customer Requirements

The workstation connectivity investigation generated an extensive set of customer requirements. These inputs spanned the entire realm of product definition, from technology needs through user disciplines and measurement tasks. The investigation revealed:

- Prototype measurements are essential for hardware/software integration and product verification.
- There is a wide variety of use models, from the benchtop standalone user to completely networked control and viewing of measurements from a remote site.
- An analyzer might see single-person use or shared use by teams of as many as 50 engineers.
- There is a strong desire for a turnkey system, not a collection of components that must be integrated by the customer.
- Equipment setup time must be a minimum. Customers want to be able to make a connection to the prototype and make measurements as soon as possible.
- There is a broad diversity of measurement needs, from signal integrity to software, and across a variety of probed points, from serial buses to RISC processors.
- Surprisingly, we found the potential to address a broader group of customers than envisioned before the investigation, including software developers, who traditionally avoided the lab and used host-based software development tools.
- We heard a strong request for snappy displays and quick measurement results (facilitation of the debug loop explained in ***Article 1***).
- The solution must interface with a wide variety of design tool chains, consisting of EDA software programs, test development programs, and others.
- There is a desire for connection to diverse open-systems networks consisting of nearly every possible combination of computer hardware and operating system software on the market.
- The base system must be affordable to reach most users and scalable to cover both performance and mainstream users.
- The Ethernet protocol was in use at most customer sites.

Taken together, these inputs overwhelmed the project team. Scaling the task to available time and resources became a critical project goal. Reducing the dimensionality required of the architecture would be the key to delivering an on-time product that met key user needs.

Table I
Measurement Server Architectures

Advantages

Distributed
- Fastest leverage of links to other host-based applications.
- Most efficient at using customer's available computing power (MIPS). Takes advantage of idle MIPS on customer's host computer.
- Low cost of goods sold (software only).

Mixed
- Provides best mix of flexibility and power for custom system development.
- Best for integrating different measurement systems.
- Deterministic performance can be achieved over a dedicated network (HP-IB).

Encapsulated
- Turnkey solution (ready to run).
- Very high measurement throughput.
- Deterministic performance.
- Low cost of sales.
- Low cost of support.
- Low R&D resources required to support and enhance the system.

Disadvantages

Distributed
- Low measurement throughput.
- Nondeterministic performance (Ethernet).
- Requires continuing R&D effort to maintain support on new operating system revisions.
- Requires extensive and time-consuming QA on supported host platforms.
- Requires some customer operating system and I/O knowledge.
- Higher support costs than encapsulated solution.

Mixed
- Requires special dedicated network hardware and software.
- Requires customer operating system, I/O, and networking knowledge.
- Throughput can be limited (depending on the performance of the dedicated network).
- Requires continuing R&D effort to maintain support on new operating system revisions.
- Higher support costs than encapsulated solution.

Encapsulated
- High cost of goods sold.
- Fixed functionality.
- Low efficiency of using customer's available MIPS.
- Requires some networking knowledge.

Demographic Factors

Digital design teams of all sizes are found in every industrialized country. Design teams as small as two engineers expressed many of the desires listed above. Because of the broad demographics of digital design teams, a key challenge was offering a low cost of sales and support product structure to fit a sales channel that could reach these customers.

A complex product structure greatly increases the cost of sales and support. A high cost of sales makes broad product adoption difficult. This is because complex product structures typically demand a specialized sales and support organization, which in turn limits coverage both geographically and at smaller accounts. Product specialists are expensive because of their training and expert knowledge, but also because it is difficult for them to cover the same geographic area as thoroughly as a group of generalists who make daily contact with a wide variety of customers, and who sell a wide variety of products.

Product generalists are far more likely to make contact with small customers. These customers are less inclined to purchase a system they view as so complex that a specialist is required to sell and support it. Creating a product that demands product expertise and specialization of a sales force directly contradicts the fact that design teams of all sizes and budgets have many of the needs listed above.

The resolution of this issue became a focal point of the prototype analyzer product structure. The outcome of the measurement server architecture decision would be the major contributor to the cost of sales. Reducing the dimensionality of the product would help lower overall costs and allow the product to be sold and supported by generalists, not specialists.

Issues

The four issues that most affected the choice of measurement server architecture were:

- Fast throughput and quick answers
- A desire for a turnkey solution
- An affordable base system price
- Connection to diverse open-systems networks and platforms and interfaces to a wide variety of tools.

The additional requirement that was used as a starting point for the design was that the product must work with existing HP 16500B systems and measurement modules. The design team was to create a product that added functionality to the HP 16500B system but did not require customers to make a significant new investment in real-time acquisition hardware.

Work on the core software architecture began independently of the measurement server decision. The four issues were used as a basis for discussion about the pros and cons of each measurement server architecture. In the end, an encapsulated architecture was chosen, as shown in Fig. 2.

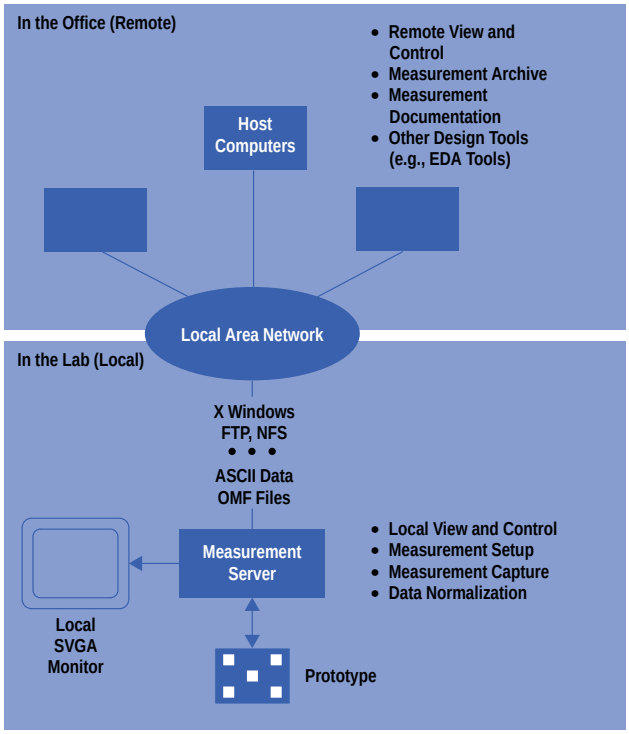


Fig. 2. *HP 16505A prototype analyzer measurement server implementation.*

Throughput

Throughput was a key element of the prototype analyzer design. A customer set a benchmark for the design team by stating the expectation that a full screen of HP 16550A data should be refreshed once per second. This became known as the "one update per second" goal and was used as the de facto throughput benchmark.

Table II outlines the amount of data that some HP 16500B logic analysis system measurement modules capture in one acquisition. The italic numbers indicate common configurations. These configurations were used in ad hoc tests by R&D to evaluate the update rate as the software architecture progressed.

A single acquisition covering several microseconds could generate as much as 30M bytes of data. Sending this amount of data over a customer's local area network would be impractical and would cause severe network performance problems. It

was obvious that some sort of dedicated network would be required to move data quickly from the HP 16500B system to a workstation or PC for data normalization and display.

Table II
Typical Measurement Data Sets

HP 16500 Measurement Module	1 Module	2 Modules	3 Modules
HP 16550A 500-MHz Timing, 100-MHz State, 4K Depth, 100 Channels	55K bytes	*110K bytes*	N/A
HP 16555A 500-MHz Timing, 110-MHz State, 1M Depth, 64 Channels	8M bytes	16M bytes	*24M bytes*
HP 16532A 1-GSa/s Digitizing Oscilloscope, 8K Deep, 2 Channels	*16K bytes*	32K bytes	48K bytes
HP 16517A/18A 4-GSa/s Timing, 1-GSa/s State, 64K Deep, 16 Channels	*128K bytes*	256K bytes	384K bytes

The real-time nature of the system implied a robust and deterministic interface. Handling real-time I/O across open networks without the benefit of a well-defined protocol is difficult, and would require special drivers on both ends of the network. Porting real-time code across platforms would add complexity to the design.

Leveraging analysis done by HP Laboratories and other divisions,[3] the design team was quickly able to evaluate interface performance characteristics. Table III provides a comparison of different interfaces and the estimated throughput of each. Data handling in the HP 16500B and normalization time in the HP 16505A is not included in these figures.

Table III
Interface Speeds and Estimated Throughputs for 110K Bytes of Data

Interface	Maximum Transfer Rate	Typical Transfer Rate	Transfer Time for 110K Bytes of Data	Transfer Time for 24M Bytes of Data
HP-IB	1 Mbyte/s	240 kbytes/s	0.47 s	100.4 s
Ethernet	1 Mbyte/s	300 kbytes/s	0.38 s	80.7 s
SCSI II	5 Mbytes/s	1.5 Mbytes/s	0.07 s	15.27 s

Although all three interfaces perform the 110K-byte transfer in less than 0.5 second, the SCSI II interface offered a substantial improvement in performance, which would be needed when transferring the 30M-byte files found in high-end HP logic analysis configurations.

SCSI II was not the first choice of the design team. The HP 16500B already had HP-IB and Ethernet ports. Adding a SCSI port would take more time and resources. Some team members argued that HP-IB performance could be improved. However, the HP-IB interface would then require a corresponding connection on the other end. Since workstations and PCs don't come standard with HP-IB interfaces, the use of this port would require special hardware for the host computer. No one relished the task of designing a computer-based HP-IB card or evaluating the commercially available cards.

Ethernet was a clear winner from a user perspective and an Ethernet port was available on the HP 16500B. The two strikes against Ethernet were its performance compared to SCSI II and its inherent nondeterministic behavior, a result of the collision detection and retransmission scheme used.

In the end, the SCSI II port won out. In retrospect, the use of the fastest interface available was an excellent choice, since HP 16500B data sets continue to grow in size and customer throughput expectations constantly increase.

As the design team developed the software architecture, it became apparent that there were many areas where code optimization could improve throughput. A problem with software optimization is that it is often dependent on the architecture of the underlying hardware. Although the team was using the HP 9000 Series 700 workstation as a development station, a platform choice had not yet been made. One factor that swayed the development team in favor of an encapsulated measurement server was their feeling that significant improvements to performance could be obtained by tuning the software architecture to one computer architecture. This insight proved fortuitous because the R&D team got the chance to optimize the architecture and gain a 10× performance improvement when the coding was complete.

The decision to use SCSI II meant that a distributed measurement server architecture would not be feasible, since SCSI is not a network protocol.

Turnkey System

With the interface issue settled, the design team began investigating dedicated controllers. Both workstations and PCs were evaluated. Each platform had advantages and disadvantages, as outlined in Table IV.

Table IV
Controller Comparison

Advantages

Workstations

- Highest single-processor performance available
- Standard SCSI interface
- Good networking
- Good development environment
- Can act as X client or server
- Excellent graphics support

PCs

- Good performance
- Generally lower cost than workstation
- Excellent development environment
- Customers familiar with Microsoft Windows®
- SCSI interfaces available

Disadvantages

Workstations

- Higher cost than PCs
- Many customers not familiar with HP-UX* operating system
- Potential for file system corruption
- Requires more base system memory

PCs

- X client software not readily available
- Acceptable but not robust networking

An important factor in the decision between a workstation and a PC was the ability of a workstation to operate as both an X client and an X server. Since customers demand both local and remote viewing and control of the prototype analyzer, X Windows was a necessity. In general, PC-based operating systems did not support this function or handled it as a special case.

However, a problem with the choice of a workstation was the HP-UX operating system. Many customers were not familiar with HP-UX and did not want to learn it. Seeing an HP-UX prompt on the screen would also create problems for HP's general-purpose sales channel, who were generally unfamiliar with the UNIX® operating system.

The team decided to create a complete turnkey system on top of the operating system. This meant that the system could not be used to run other HP-UX applications or as a general-purpose computer. While the implementation of this policy was not technically difficult, explaining the concept took considerable effort. The team found that since there were many existing models of mixed measurement server systems, the natural conclusion was that the prototype analyzer would also be open for customer development. The level of communication required to explain that the prototype analyzer would be turnkey was much higher than anticipated. In the end, the team found that demonstration was by far the most effective way of communicating the product's structure and its advantages.

An added benefit of offering a turnkey system was that the team did not have to worry about operating system revisions. Maintaining an open system would require the team to put ongoing effort into supporting three operating system revisions: the last, the current, and the future.

The prototype analyzer is built upon only one operating system revision. The entire system will be revised only when customer needs change and substantial new functionality is required. This frees the design team from chasing operating system revision-related product defects.

Meeting Price Goals

Research indicated that while customers wanted a powerful analysis and display environment, their perceptions of price were influenced by the availability of PC and workstation-based data analysis packages that sold in the U.S.$1000 to $5000 range. Customers also viewed the prototype analyzer as a kind of operating system. Operating systems aren't valued as highly as the applications that run on them.

This data implied that the total system price would need to be in the range of U.S.$5000. The design team had two areas where costs could be lowered. The workstation would represent a large portion of the prototype analyzer material cost. Selling and support expenses also contribute to the total system cost.

Concurrent with the prototype analyzer development, HP's Workstation Group was designing the HP 9000 Model 712 low-cost workstation. A review of the workstation's price and performance specifications indicated that it would be an ideal fit as an encapsulated measurement server.

The base system would consist of a 60-MHz CPU, 32M bytes of RAM and a 525-Mbyte hard drive. The system would be shipped from HP's workstation manufacturing operation to the Colorado Springs Division, where a new operating system and the application code would be loaded and tested. The completed product would be shipped to customers from Colorado Springs as a turnkey system. Minimizing the extra handling helped keep direct manufacturing expenses low. As with any new product, initial inventory was difficult to estimate because there was no order history. However, the ability to order and receive semiconfigured systems with fairly short lead times helped maintain a low inventory and thus contributed to a low manufacturing cost.

Cost of sales is defined here as the effort put into customer contact, demonstration, and objection handling during the sales process. Although precise numbers on these efforts are not available, the design team had sufficient practical experience to know what factors contributed to a complex product and a higher cost of sales.

Traditional instruments generally have a low cost of sales because they can be easily explained, demonstrated, and left with the customer for an extended evaluation. The instrument model was used as a goal for prototype analyzer structure, connection, and demonstration.

A learning products engineer was assigned to evaluate barriers to installation and first measurement. Two goals were adopted. The first was that customers should be able to get the system running from a shipping box in less than one hour. This was accomplished by examining in detail the steps a customer would take getting from the shipping box to power-up.

A second goal of getting from turning on the power to a measurement in less than 15 minutes resulted in significant changes to the initial user interface design. The initial interface used the standard window bar to access instrument, analysis, and display tools. After several days of usability testing at a customer site, the learning products engineer mocked up a toolbox on the left side of the workspace using masking tape on the monitor. The toolbox contained all available tools. These tools could be dragged and dropped onto the workspace (see Fig. 3) and would automatically connect themselves.

The efforts put into these goals paid off. Both sales engineers and customers found the product easy to set up and run. The toolbox proved to be a big hit during customer demonstrations and added significantly to the ease of use of the product.

Products with a substantial software content present support problems. The EDA industry generally addresses this issue with software maintenance or support contracts, which provide the customer with software updates and defect fixes over a specified period of time. Software maintenance contracts are generally priced at a percentage of the total system software cost.

The prototype analyzer team wanted to hold to the instrument model. Most instruments do not have software maintenance contracts. Instead, software upgrades and defect fixes are usually distributed through an ad hoc process of sales and system engineers personally distributing flexible disks or occasional mailers to customers who have expressed an interest in future software upgrades.

The prototype analyzer team decided to implement a software notification process. This process would reduce the burden on the HP sales or system engineer of distributing new software and defect fixes. Defects and minor revisions would be distributed free of charge. Customers would receive notification of the availability of major revisions and related new products.

The difficulty with this approach lay in the need for many new processes within HP. These included development of a call-handling center to interface with customers and get their names and shipping addresses for the free updates. Process improvements are ongoing, but customers have indicated their satisfaction with the approach.

Standard Interfaces

Somebody once commented that the great thing about standards is that there are so many to choose from. The prototype analyzer design team faced a bewildering array of networking, data, and tool chain standards. Narrowing the choices down to just a few supported standards would be required to meet the project schedule.

Networking standards were quickly defined as a result of the decision to go with the encapsulated architecture. This choice necessitated the use of the X Window protocol to support local and remote control and FTP/NFS to get data in and out of the

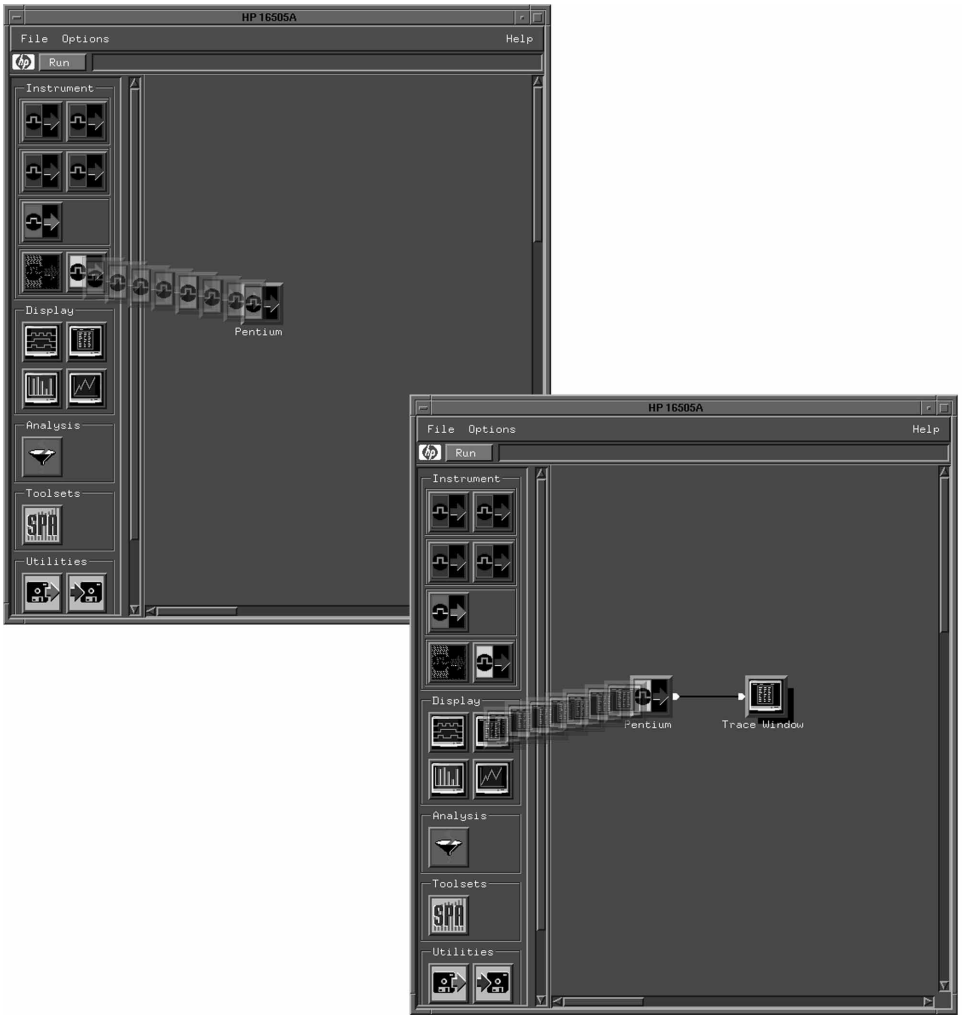


Fig. 3. *Tools can be dragged from the toolbox at the left side of the display and dropped onto the workspace of the HP 16505A prototype analyzer. They connect themselves automatically.*

system. Ethernet was the natural choice for a network connection. Fortunately all of the networking hardware and software was already present in the HP Series 712 workstation and only needed to be augmented with a graphical user interface.

Early prototype analyzer users found that the networking was sufficient but not ideal. In particular, customers asked for the ability to call up remote X Windows applications onto the prototype analyzer's local display. This feature was useful because customers could access a remote application such as a schematic or text editor from the lab bench. This capability was added in a subsequent release of the product.

Interfaces to hardware and software tool chains continue to evolve. The encapsulated measurement server approach enables the design team to maintain control over the type and manner of data exchanged with other design tools. Having control over this process provides additional robustness and stability which is critical to maintaining the low cost of sales discussed above.

The Results

The HP 16505A prototype analyzer has met its price and performance goals. The only true measure of a product's contribution, however, is customer acceptance. A wide range of customers, including all major computer manufacturers, have purchased prototype analyzers to aid in the development of their high-performance digital systems.

Measurement throughput continues to improve as the design team gains knowledge about the cause of performance bottlenecks. The ability to focus on a single platform and the simple software update process afforded by the encapsulated nature of the prototype analyzer make it easy for the design team to develop and distribute new software that contains performance improvements.

Adherence to network standards such as FTP, NFS, and the X Window System has lowered the support burden. However, the diversity of customer networking schemes does create a demand for factory-based network troubleshooting expertise. The

design team will continue to apply best networking practices to the system in an effort to reduce the occurrence of network setup problems. Usability testing will be used to gain insight into networking problems.

The ability of the architecture to support additional functionality makes it a cornerstone of HP's real-time measurement solution for digital design teams. The reduced cost of incremental development and the low cost of sales and support are important product attributes that outweigh the higher cost of goods sold compared to host-based architectures.

Conclusions

The development of the HP 16505A prototype analyzer presented unique challenges to the project team. The encapsulated measurement server approach taken by the team meant there were no guideposts to follow. The success of the project was in doubt until the product was introduced.

The decisions described in this article may appear obvious in hindsight. They were not. The design team wrestled with the pros and cons of the measurement server architecture decision throughout the product development cycle. As new data became available, the team eventually rallied around the encapsulated approach. As with any new endeavor, at the time the decisions must be made, there are no right answers, only informed judgments.

The design team learned several lessons during the development process. One clear lesson was to focus on solving customer needs. Issues such as internal architecture and form factor are important, but clearly secondary to the problems the product is trying to solve. An important market research lesson the team learned is to encourage customers to describe their problems, not their thoughts on instrument architecture.

Acknowledgments

The author would like to acknowledge several people for their significant contributions to the HP 16505A prototype analyzer. James Kahkoska distilled hundreds of customer needs into an innovative software architecture and provided the vision, skill, and tenacity needed to bring the product to market. Pat Byrne acted as mentor for the project through various internal checkpoints and helped craft and present the product's benefits to countless customers. The design team of Frank Simon, Mark Schnaible, Tracy Vandeventer, Doug Robison, Jeff Roeca, Richard Stern, John Friedman, Mason Samuels, Dave Malone, Mick Backsen, Bob Beamer, and Pat Desautels added their expertise, insights, and dedication to meet project deadlines. Doug Scott added his unbiased expertise on user task analysis. Steve Shepard and Jeff Haeffele (the "Binars") provided invaluable moral support and coaching. The extended project management team spanning all functional areas within the HP Colorado Springs Division, too numerous to mention here, also merit appreciation and gratitude for their process expertise and enthusiasm in support of this project.

References

1. N. Barnholt, "Future Test Systems Will Use Standard Networking Protocols," *Electronic Design*, January 10, 1994.
2. R. Helsel, *Cutting Your Test Development Time with HP VEE*, Hewlett-Packard Professional Books, Prentice Hall, 1994.
3. J. Burch and K. Moore, *Communication Architectures for Next Generation Measurement Systems*, HP Laboratories Technical Report HPL-92-137, November 1992.

A Normalized Data Library for Prototype Analysis

The goal was that each analysis and display tool to be included in the prototype analyzer should be designed and written only once. Therefore, the data library is designed to normalize the variety of basic logic analyzer data types and the variety of postacquisition data types generated by various analysis tools and to present this data to other analysis and display tools in a standard format.

by Mark P. Schnaible

Early in the design of the HP 16505A prototype analyzer we realized we needed a new library for storing and retrieving logic analyzer data if we were going to meet our project goals. We needed to duplicate the results of hundreds of software engineer-years of effort in the HP 16500A/B logic analysis system and measurement modules. We also had to lay the groundwork to meet the requirements of our prototype analyzer vision.

Understandably, we did not want to allocate enough people to meet the first challenge for the duration of our relatively short schedule. However, looking at where the original time was spent led to some insights. In the original HP 16500A/B system, effort had to be duplicated for each logic analyzer plug-in card introduced. That is, for each card, the lister, waveform, chart, and other software modules needed to be rewritten. Some code leveraging took place, but we did not come close to complete code reuse. The different ways in which the analyzer cards present data to the software (largely because of differing memory layouts) had much to do with this lack of reuse. Out of this observation came the design goal that each analysis and display tool to be included in the prototype analyzer should be designed and written only once. This meant that our library needed to handle the variety of basic logic analyzer data types as well as accommodate the variety of postacquisition data types that our envisioned analysis tools would generate, and present this data to other analysis and display tools in a normalized format. Fig. 1 shows the reduction in the dimensions of the coding effort we hoped to attain because of this library. It also shows that we would have automatic commonality of feature sets across acquisition modules. Previously, some features for a new logic analyzer card were not present at the initial release of that card.

Several other design goals emerged that would allow us to meet our challenges:

- Retrieval time for analysis and display tools to access the data should be minimized. In the HP 16500A/B environment, the acquired data is examined in typically one display per acquisition. In the prototype analyzer environment, the goal was to encourage data exploration and to view and analyze acquisitions with several tools. We wanted to permit users to examine data simultaneously with the same view in different locations and with different views in the same location. These use models led to multiple accesses of the same data, in contrast with the HP 16500A/B model of a single view of data in a single location. Fig. 2 shows the equivalent prototype analyzer graph of an HP 16500A/B use model while Fig. 3 shows a prototype analyzer graph with the multiple-view, multiple-location use model.
- Storage space for acquired data should be minimized. Given the potential size of some logic analyzer acquisitions ($>$40M bytes), undue expansion of the data was unacceptable. As always, however, there exists the trade-off between minimized retrieval time and minimized storage space.
- The storage mechanisms used by the library should be completely decoupled from application or client modules. That is, changing the way data is stored should not affect any lines of client code. Examples of client code include the tools that analyze and display the acquisition data such as the lister, waveform, chart, distribution, and pattern filter tools.
- The interface to the library should be a “natural” C++-style application programming interface. No new style of programming should be introduced that would be different from normal C++ syntax.
- The library should provide a layer of memory management over the acquired data. Again, the possibility of very large data sets makes it vitally important not to leak these huge memory chunks or provide stale pointers to nonexistent data. This goal is made more difficult by the potential complexity of graphs possible in the prototype analyzer. Fig. 3 shows that there may be many analysis and display tools examining and viewing the data simultaneously. All of these references to the data must be handled properly to prevent large memory leaks.
- Finally, time correlation between different sets of data or analyzer acquisitions must be possible. This requirement makes it possible for users to examine their systems on several different hierarchical levels

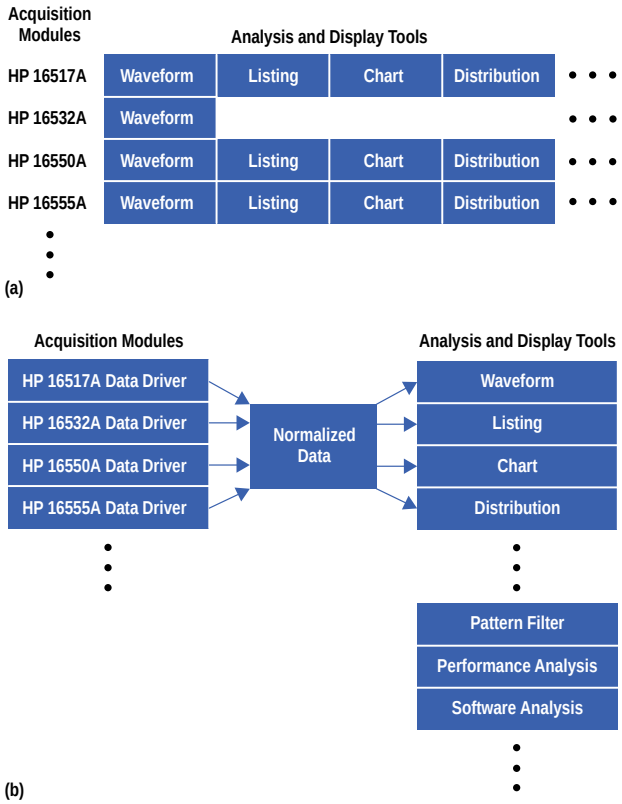


Fig. 1. *(a) In the HP 16500A/B logic analysis system coding model, tool software had to be rewritten for each plug-in card. (b) In the HP 16505A prototype analyzer coding model, data is normalized immediately after acquisition and is available to all tools in a standard format. Thus, the tools had to be coded only once.*

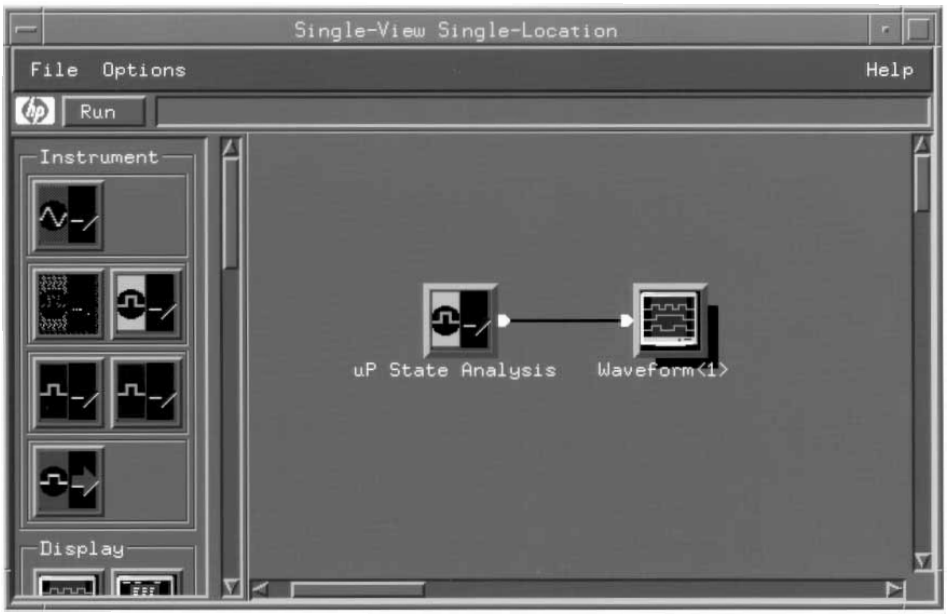


Fig. 2. *Prototype analyzer graph of an HP 16500A/B logic analysis system single-view, single-location use model.*

concurrently, from looking at the analog nature of a ground pin bouncing to looking at the high-level source code executing at that moment in time. Time correlation of measurements also makes it feasible to plot one derived measurement against another derived measurement. Previously nonobvious relationships between variables may appear as a result of these charts. Simply having the ability to move markers in one view of an acquisition and watch that location automatically tracked in a different view of the same acquisition is another benefit of time correlation.

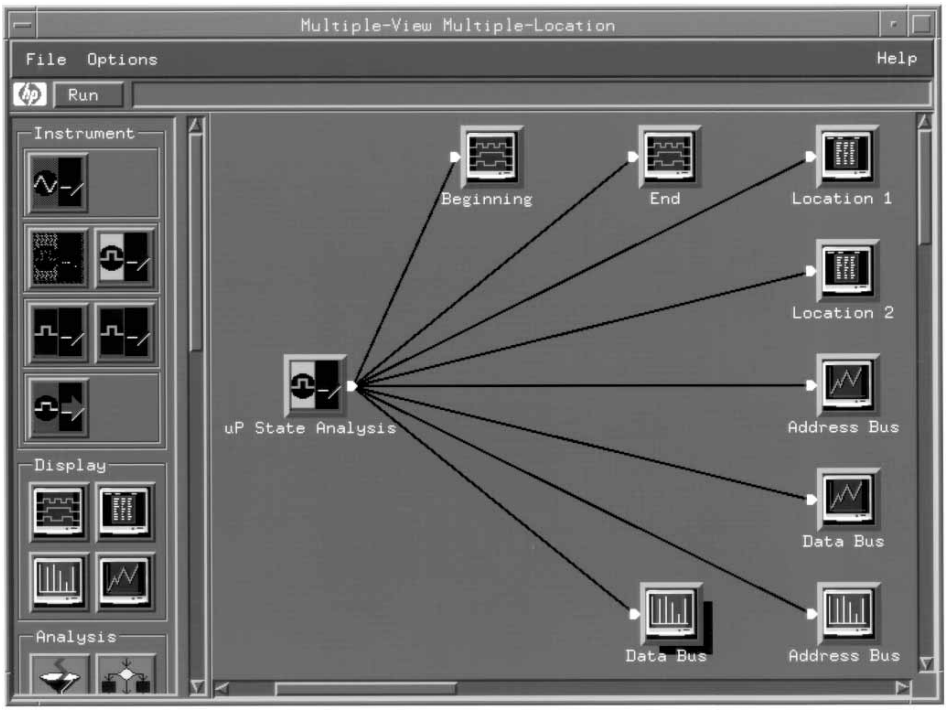


Fig. 3. *Prototype analyzer graph of an HP 16505A prototype analyzer multiple-view, multiple location use model.*

Starting with this list of requirements, we searched for any existing libraries that would meet our needs. We examined three HP internal libraries and one public-domain library. None of these met all of our requirements. Among the reasons we rejected them were data set size limitations, non-C++ APIs, file-based memory storage, inefficient storage models, inability to handle the variety of data types, and mismatches between application models.

Data Sets and Data Groups

The visual programming user interface of the HP 16505A prototype analyzer presents users with a left-to-right flow of data from sources to displays. Lines between the icons represent this flow. The first thing to decide was exactly what kind of objects move from icon to icon.

One of the first things users do in the HP 16500A/B environment is to define what the analyzer probes are connected to. This seemed like a natural place to start our definition of the normalized data library. A *label entry* represents one or more probes defining a logical value. Typical examples are the logic analyzer probes connected to an address, data, or status bus of a microprocessor or an oscilloscope probe connected to V_{cc}, V_{ss}, or a clock signal. The label entry has a name, polymorphic ordinate data, attributes, and perhaps a database of value-to-symbol mappings associated with it. Fig. 4 shows the class diagram of a label entry using the Booch notation.[1] A logic analyzer or oscilloscope commonly probes several of these label entries.

If we collect all label entries that share *exactly* the same sampling information, we have a *data set*. All of the label entries from a single logic analyzer acquisition share the same sampling information, so they can be collected into a data set. Likewise, all of the label entries from a single oscilloscope acquisition share the same sampling information. The data set

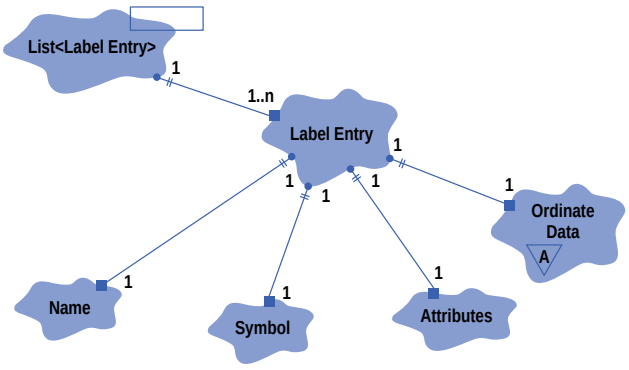


Fig. 4. *Class diagram of the label entry class using Booch notation.*[1]

contains a pointer to some polymorphic data representing this sampling or x-axis information. We call this the *abscissa data*. The actual abscissa data object varies with the type of acquisition. The data set also contains time correlation information indicating which other data sets a particular data set can correlate with and indicating their relative trigger times. Another form of information contained in a data set is called *tags*. Tags represent the availability of each sample row in a data set. Tools such as the pattern filter or the sequencing filter can remove lines from a data set based on some pattern of data or a sequence of data. The memory for these lines is not deleted; the lines are simply marked as not available by these tools.

When we collect one or more data sets for input to a tool, we have a *data group*. Data groups are the objects that "go along the wires" of the visual programming graph. A data group is simply a list of data sets and some information indicating the nature of correlation among the data sets. A data group may contain data sets that have no correlation, time correlation, state correlation, or both time and state correlation. Each tool can indicate to the system what kind of correlation it requires incoming data sets to have. Inputs that do not match this criterion cause a warning message to be shown on the display. Fig. 5 shows the class diagram for data sets and data groups.

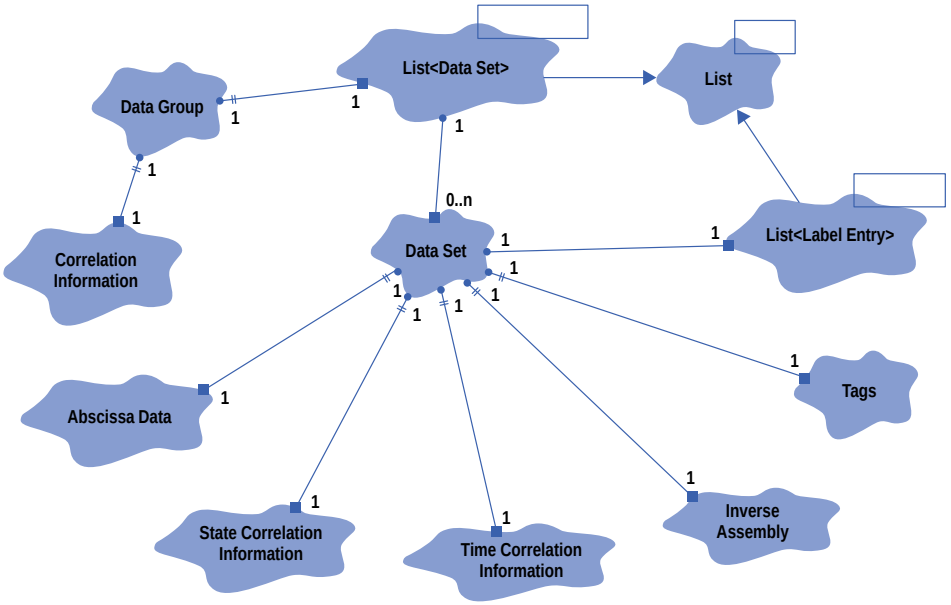


Fig. 5. *Class diagram for data sets and data groups.*

Abscissa Data

As mentioned above, the x-axis information (the abscissa data object contained within each data set) is a polymorphic type dependent on the type of acquisition or measurement of the data. Fig. 6 shows the class hierarchy of this information. The abscissa data base class represents a simple numbered sequence of states with one of those states being the trigger state. This concrete base class is the type used for generic logic analyzer state clocked acquisitions with no time information, that is, no time tags. Objects of type *periodic* are used for logic analyzer timing acquisitions as well as oscilloscope acquisitions. This class stores information about the sample period and exact time at trigger. The *time tags* class is used for logic analyzer state acquisitions with time tagging turned on. These tags indicate the exact time for each sample, which may or may not be periodic. This class is also used for calculated data sets derived from logic analyzer acquisitions. For instance, a series of setup and hold calculations derived from a timing acquisition will have a time tag for each pair of setup and hold values. We can save considerable space with this technique because the setup and hold times are sparse compared to the sample density of the original acquisition.

Ordinate Data

The probed data or calculated data contained in the label entries is also a polymorphic object. The base class of this hierarchy is an abstract base class called *ordinate data* which defines the interface for classes derived from ordinate data. Fig. 7 shows the class hierarchy rooted with ordinate data. Classes derived from ordinate data are *analog*, *states*, *glitch*, and *state count*. Other classes can be added to this hierarchy as needed. The analog class contains a polymorphic pointer to the quantized samples and an object specifying how to convert these quantized values to floating-point values. In the case of an oscilloscope acquisition, this header contains the formula to convert from quantized samples to voltage. The states class contains a pointer to some polymorphic data representing the binary values acquired at the logic analyzer probes. The glitch class contains the same information as the states class as well as a parallel data object indicating the presence of glitches for each sample. The HP 16550A logic analyzer card is capable of acquiring this type of data. The state count class holds the data representing the number of states that transpired between acquired or store-qualified states. The trigger systems of HP logic

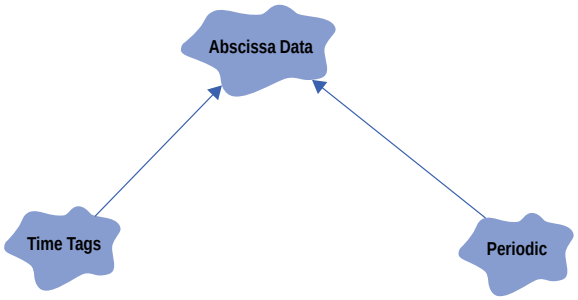


Fig. 6. *Class hierarchy for abscissa data.*

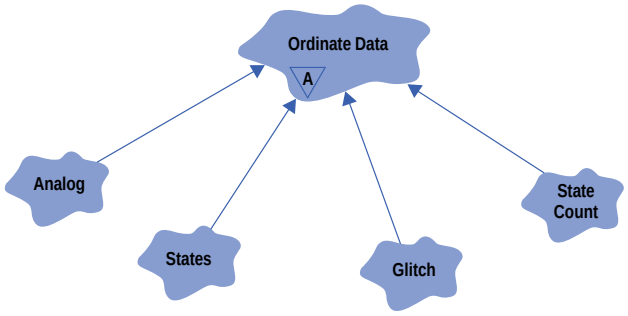


Fig. 7. *Class hierarchy for ordinate data.*

analyzers allow users to specify which states to store out of all the states the logic analyzer sees based on some sequence of events. The state count object holds values that represent the number of states that were *not* stored.

Analyzer Memory Layout

Our definition of a label entry shows that the data for each label entry is stored with that label and the data for no other label entry is stored with that label. This contrasts with the HP 16500A/B series of logic analyzer cards in which all of the acquisition data is stored in one block of RAM. Each time the data for a particular label needs to be retrieved, the bits must be extracted from this block and packed into a value representing a sample. As Figs. 8 to 10 show, the format of these blocks of memory differs from analyzer to analyzer.[2,3] The data formats in the acquisition cards are designed to be optimally space-efficient. There is a one-to-one mapping between bits of memory and probe tips so no memory waste occurs. There are two downsides to this design choice: (1) time must be spent every time a sample needs to be extracted from this block of memory and (2) the memory layouts are different for each analyzer. For the multiple-view, multiple-location use model of the prototype analyzer, access time for each sample of data is critical. For code reuse reasons, we need to hide the memory layouts from client code. Since these acquisition cards were already in production, we had little motivation to alter the way they presented data to the prototype analyzer. We wanted to minimize the software investment in these finished products.

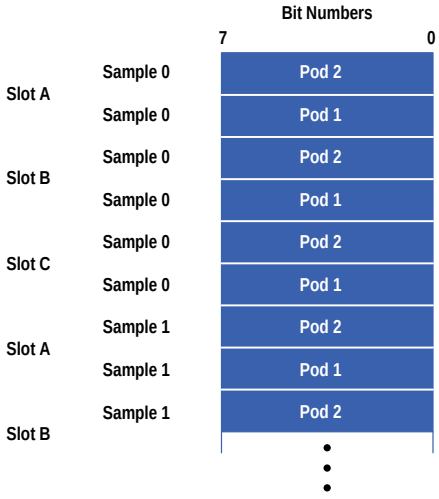


Fig. 8. *HP 16517A acquired data format (three-card analyzer, full-channel mode only).*

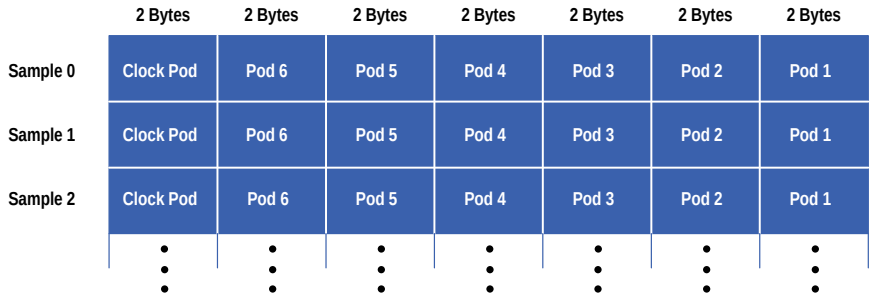


Fig. 9. *HP 16550A acquired data format (one-card analyzer only).*

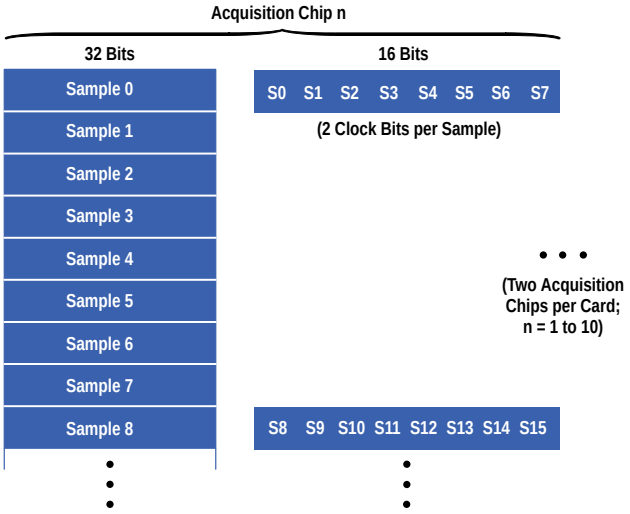


Fig. 10. *HP 16555A acquired data format (full-channel mode only).*

Format Specifications for Labels

Fig. 11 shows a partial format specification for the Intel P54C microprocessor. Customers use this dialog to specify which pins on various pods of logic analyzer probes are connected to different labels. Some labels, such as for an address bus or a data bus, tend to be probed with contiguous probe tips. However, other labels can be probed with discontinuous pins from different pods. The exception label (Excptn) in Fig. 11 is an example of this in the P54C specification. Depending on the complexity of the format specification, the extraction time for these labels can vary greatly. Even for contiguous labels with widths of 32 or 16 bits such as address or data, considerable time can be spent extracting these values from the monolithic block of memory.

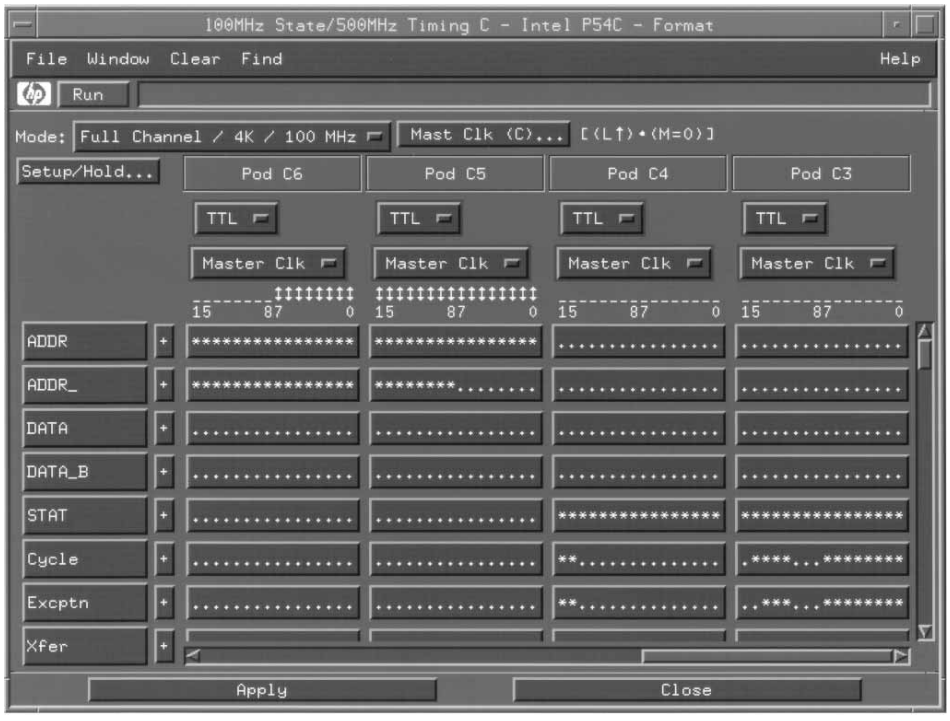


Fig. 11. *Partial format specification for the Intel P54C microprocessor.*

For the normalized data library, we decided to extract the data for each label once for each acquisition and store it in its own chunk of memory. Label entries whose bit widths match the native machine data type widths (8, 16, or 32 bits) are extracted and stored as arrays of C++-type chars, shorts, or ints. Once these samples have been extracted or *normalized*, subsequent retrievals are simply array lookups. All labels that have non-native widths are combined into a block of bytes. Before the

values are inserted into this block, however, any discontinuity in the ordering of the bits is removed to speed later retrievals. Some space inefficiency can result from this technique. The total possible bit wastage is:

$$\left[\left(\sum \text{non-native widths}\right) \bmod 8\right] \times \text{acquisition depth.}$$

We make this trade-off to provide faster access times to the data. Data drivers for each logic analyzer module supported are responsible for normalizing the data. These drivers can be arbitrary about choosing data types because they know the memory layouts of the cards and the specific widths of labels. Fig. 12 shows the class diagram of the ordinate data types with the various *integral data* types.

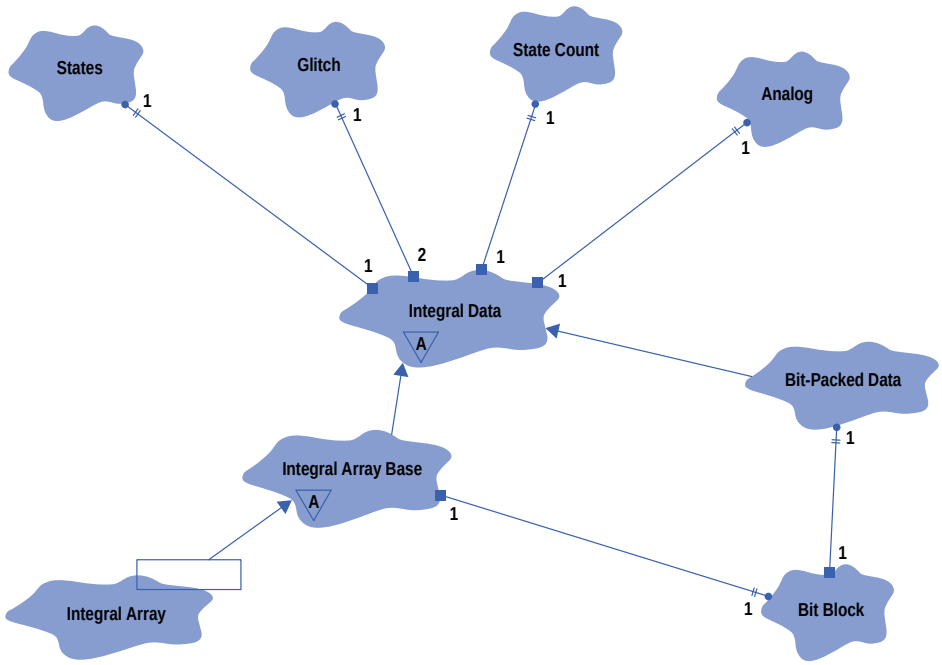


Fig. 12. *Class diagram for the ordinate data types (top) with the various integral data types.*

Accessing the Data

Once we have stored the data in a normalized format, we need methods to access the data. A study of existing code and an analysis of how future tools might need to access the data showed that retrieval tended to be very sequential in nature. If we picture an acquisition as a matrix of data, with rows representing samples or time and columns representing the various labels, two retrieval styles emerge: row major and column major. Tools that use the row major style want to examine all of the labels for a certain sample before they proceed to the next sample. The lister, pattern filter, and sequencing filter use this style. Tools that use the column major style look at all samples of a particular label or column before they look at the next label. Waveform drawing, charting, and histogramming tools access the data this way.

A programming idiom that supports this sequential nature of accessing data is called an *iterator*.[4] We defined classes for iterating over data contained in the abscissa data class hierarchy as well as data contained in the ordinate data class hierarchy. Fig. 13 shows the *abscissa-itor* class hierarchy and Fig. 14 shows the *ordinate-itor* class hierarchy.

Both diagrams show that there is a one-to-one correspondence between data types and iterators over those data types. Since we don't want client code to know how the data is stored, we need some way to hide this information from clients while still providing them with a way to get at the data. We use the letter/envelope programming idiom to accomplish this.[5] Clients construct an envelope object (*general abscissa-itor* and *general ordinate-itor* in the diagram), which can then ask the data to construct an iterator over itself. Clients never have to know what kind of data is being accessed. The envelope and letters are derived from a common base class which defines the interface for the iterators. The envelope serves as a forwarding object to the real iterator, the letter. The iterators have an orthogonal interface for data retrieval and iterator positioning. There are methods for looking at the next and previous data elements (which then alter the position of the iterator), methods for peeking at the next and previous data elements (do not alter the position), methods for querying and setting the position of the iterator, and methods for testing forward and backward iterator exhaustion. Multiple iterators can be constructed to look at the same data; this would be the case in a prototype analyzer graph that takes advantage of the multiple-view, multiple-location use model. Iterators are declared so that they can only access the data. We provided no iterator methods that would modify the data.

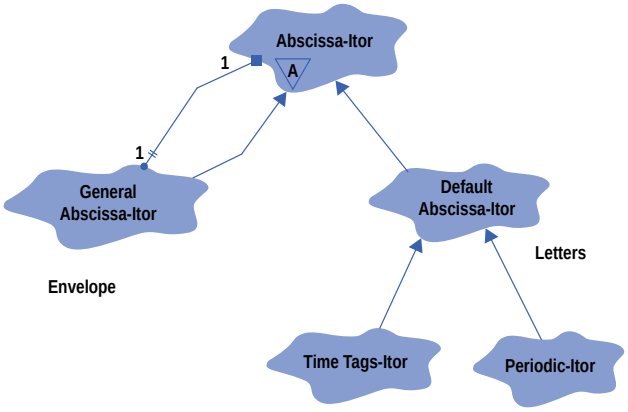


Fig. 13. *Abscissa-itor class hierarchy.*

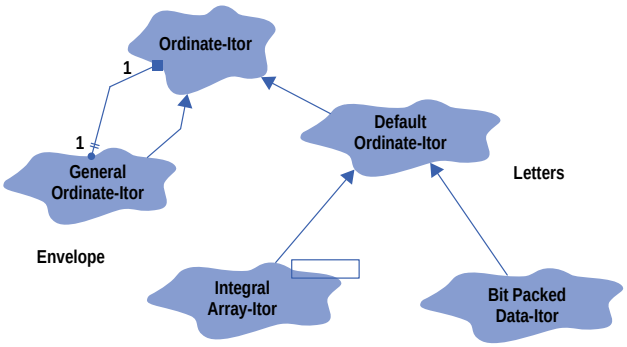


Fig. 14. *Ordinate-itor class hierarchy.*

We also defined an iterator that is a combination of an abscissa-itor and an ordinate-itor. The *data-itor* retrieves pairs of values representing the value of a label and the sample or time at which it occurred. In addition to the orthogonal interface described above, this iterator also provides methods to report the value and location of the next or previous change in the ordinate data. These methods are useful for waveform drawing algorithms.

Row major iteration over multiple correlated data sets is aided by group abscissa iterators. These iterators examine a list of abscissa-itors (one for each data set in the data group) and return the next or previous x-axis value and a list of identifiers that select the data sets from which that value comes. These iterators come in handy, for example, in visual programming graphs that have multiple analyzers fanning in to a single analysis tool (Fig. 15). For a lister display that has multiple data sets merged (Fig. 16), we need to show the data from the various analyzers interleaved in time as they were actually sampled.

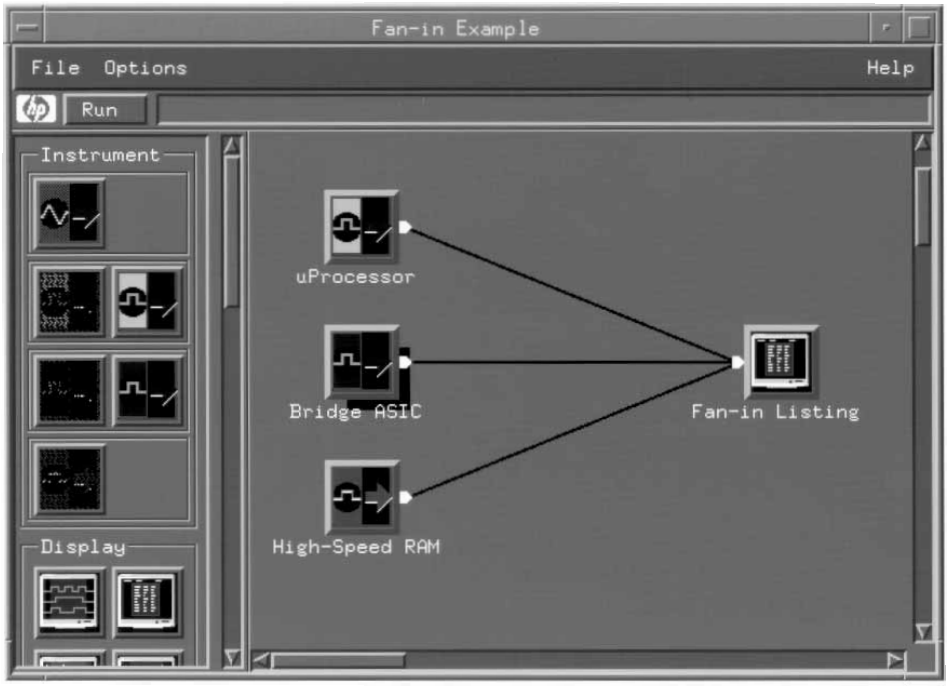


Fig. 15. *A visual programming graph with multiple analyzers fanning in to a single analysis tool.*

Memory Management

With the increasing acquisition depth and width of HP logic analyzers, data sets can have sizes greater than 40M bytes. In the prototype analyzer environment, users can be examining these data sets in several locations with several views. Having so many references to the data can easily lead to memory management problems, particularly nasty among them being memory leaks and pointers to stale memory. A memory leak occurs when a process no longer has any references to a chunk of memory that has been allocated from the heap of memory available to the process, and thus there is no way to return the memory back to the heap. As an application leaks more and more memory, less is available to that process and eventually

Fig. 16. *A lister display with multiple data sets fanned in. The data sets from the various analyzers are interleaved in time as they were actually acquired.*

the application will halt because of an out-of-memory error. Leaking very large blocks of memory will cause this condition to occur sooner. Pointers to stale memory result when the application does return a chunk of memory to the heap but keeps other pointers to that same memory, which the application no longer owns.

The normalized data library uses the reference counting idiom[5] to overcome these potential problems. Passing a data group from one tool to the next creates a copy of the data group. These copies result in incrementing reference counts to the individual blocks of data contained in the abscissa data and ordinate data object hierarchies. Defining an iterator over this data also constructs a copy of the data, which causes a reference count increase. Destroying the data (by deleting a tool) or deleting iterators causes these copies to be deleted, which then causes the reference counts to be decremented. When the reference counts reach zero it is safe to release the memory back to the heap. At that time we are sure no other references to that memory exist. The normal execution of a tool includes these steps:

- Delete the old output data group of the tool.
- Delete the old input data group of the tool.
- Construct a new input data group for the tool by merging all inputs.
- Execute the tool (delete old iterators and construct new ones).
- Construct a new output data group for the tool.

We defined our iterators such that they do not modify the underlying reference counted data. In cases where tools need to modify the reference counted data (such as pattern filters which modify the tags object in a data set), we use the copy-on-write optimization: operations that would change the data cause a completely new copy of the data to be created and cause reference counts to be adjusted accordingly.

Case Study

During the design of the normalized data library for the prototype analyzer we made a conscious decision to incur the overhead of normalizing the acquisition data immediately after transferring the data from the HP 16500B mainframe to the prototype analyzer. This happens once per acquisition and before any tools examine the data. We do this immediately to optimize the response time of postacquisition data exploration. For very large acquisitions, however, this normalization time can be considerable.

At the initial release of the prototype analyzer, one customer was making such an acquisition with 1M-byte-deep HP 16555A logic analyzer cards probing an Intel P6 microprocessor and a PCI bus. We found the normalization time to be too large in this case—the acquisition data exceeded 30M bytes. After studying where our code was spending time during the normalization process, we optimized certain sections of code as well as significantly changed the way in which the data driver for that module stored the data in the object hierarchies. After our optimizations, we reduced the normalization time by a factor of ten. More important, we did not need to change a single line of client code (lister, waveform, chart, etc.) to take advantage of this optimization.

Conclusions

At the introduction of the HP 16505A prototype analyzer we met most of our project goals. We duplicated almost all of the HP 16500B functionality for the HP 16550/4/5/6A, HP 16517A, and HP 16532/3/4A acquisition cards and added several significant features such as pattern filtering and multiple-view, multiple-location data exploration. The ability to write a single dynamically loaded shared library for each analysis and display tool that would work for any of the data retrieved from the acquisition cards was prominent among the reasons we met our goals. We continue to create new analysis and display tools that will help our customers solve their most difficult design problems.

References

1. G. Booch, *Object-Oriented Design with Applications*, Benjamin Cummings, 1991.
2. *HP 16517A/18A Programmer's Guide*, Hewlett-Packard Part Number 16517-97000.
3. *HP 16550A Programmer's Guide*, Hewlett-Packard Part Number 16550-90902.
4. E. Gamma, et al, *Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
5. J.O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

A Full-Featured Pentium® PCI-Based Notebook Computer

The HP OmniBook 5000 computer takes advantage of new technologies such as mobile Pentium, PCI, plug and play, lithium-ion batteries, and hot docking to give users the same capabilities as their desktop computers.

by Timothy F. Myers

The HP OmniBook 5000 computer (Fig. 1) marks a change in the direction of notebook computers designed by Hewlett-Packard. Earlier OmniBook products focused on being ancillary tools to the conventional desktop PC. The designs were biased towards small size, long battery life (or more usually, a smaller battery), and the ability to run most major programs. They brought to the customer new features such as instant-on, battery charging while operating, expandability via the PCMCIA standard (now known as PC Card Standard, or more simply PC Card), and simplified use. As customers became acquainted with their mobile computers they demanded that their portables have the same functionality as their desktops. Thus began the emergence of color displays, larger hard drives, faster processors, and external flexible disk drives in the HP OmniBook family. When the Corvallis Division became the Mobile Computing Division with the charter to design, produce, and market a full range of mobile computers, we realized that we needed to fill a gap in HP's computer line: the full-featured notebook computer.

Fig. 1. *HP OmniBook 5000 computer.*

Full-featured notebook computers today are characterized as having the same capabilities as desktop computers, albeit with some time lag for the highest performing models. Over the past few years, the spread in processing performance and features between notebooks and desktops has narrowed. At the end of 1995, the gap was only about three to six months. Today's customers not only demand that their notebook have the same capabilities as their desktop but also expect more in features that make mobile computing easy. In addition to being a desktop replacement the notebook computer should provide the same flexibility in prepurchase configurations and future expandability.

To facilitate HP's entry into the full-featured notebook market we chose to partner with a foreign notebook design and manufacturing company to quickly modify and produce a product for this segment. While the HP OmniBook 4000 was a very solid and feature-rich notebook product, our customers quickly requested features that were unique to our original OmniBooks, especially instant-on. Instant-on is a feature that allows users to put the machine into a low-power state and later resume working exactly where they had suspended their work. The main difference between HP's instant-on and our competitors' suspend feature is that the amount of time that our products can remain in the suspended state is measured in weeks, compared with hours or a couple of days for competitors. With the advent of new technologies such as mobile

Pentium, PCI, plug and play, lithium ion batteries, and hot docking, we felt that we could design a product that would set HP apart from many competitors offering Pentium notebooks. The HP OmniBook 5000 shows that there are still abundant areas for contribution, even in a highly competitive market.

Because of the multiple choices before us in chipset selection and peripheral IC technologies, we needed to have some major goals to guide our design process. While earlier HP OmniBooks focused primarily on size, power, compatibility, and performance, in that order, our criteria were different. Our target market was to be mainly corporate users, so our first goal was compatibility: "If we can't run it, they won't buy it." If our product makes it difficult to get some program or utility working, our customers will pick another product. The second major goal was performance. If our product is not near the top in benchmark performance, we will not appear to be technology leaders. The third goal was power management. If our product does not run very long on a battery charge it will not be very convenient. If we do not manage power wisely, the heat generated by the components could affect reliability, functionality, and customer satisfaction. Of course, there were other goals, such as convenience, quality, reliability, and functionality. However, the first three guided the decision process for the HP OmniBook 5000.

Architecture

The HP OmniBook 5000's architecture is based on a layered bus concept. Fig. 2 is a block diagram of the computer. This approach allows the devices that need higher data bandwidths to reside on an appropriate layer to maximize performance, power, number of pins, and functionality. The widest-bandwidth bus is the Pentium's 64-bit data bus. On this bus are the

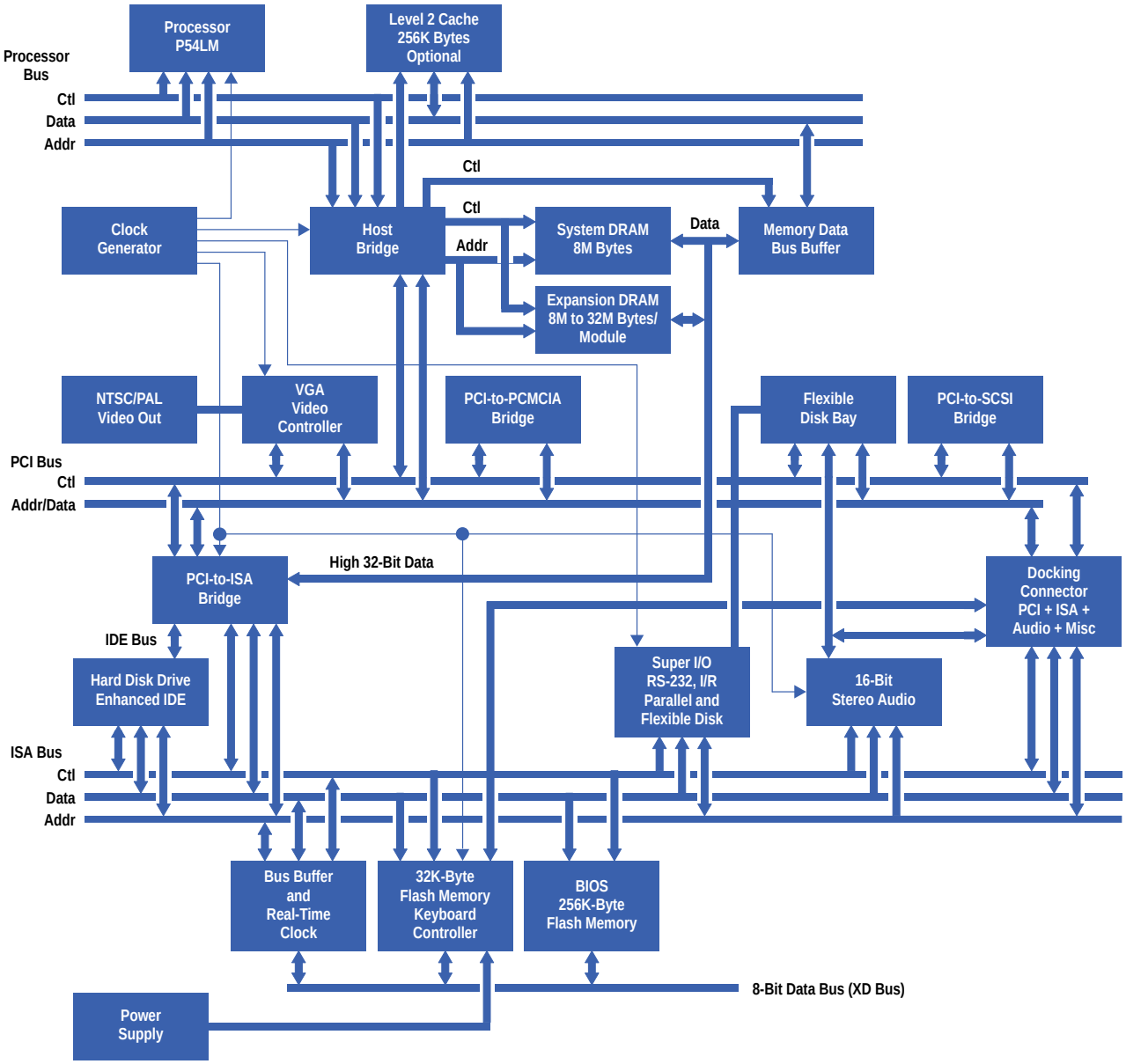


Fig. 2. *Block diagram of the HP OmniBook 5000 computer.*

system DRAM memory, the level-2 cache, the host bridge, and the Pentium CPU. The CPU bus can operate at 50, 60, or 66 MHz, depending on the CPU processing speed.

It is imperative that external memory data be delivered to the CPU as fast as possible. The Pentium processor has 16K bytes (8K data, 8K code) of internal level-1 cache, and the CPU module can support 256K bytes of external level-2 cache. The level-2 cache is implemented using burst synchronous static RAM with fast access speeds so there are no wait states other than the address lead-off cycle. Because the Pentium CPU and level-2 cache consume quite a bit of power even in a nonclocked state, the power to these devices is turned off during suspend periods and the control signals from the host bridge are put into a high-impedance state (tristated). The host bridge provides the DRAM and level-2 control and also serves as a gateway to the PCI bus.

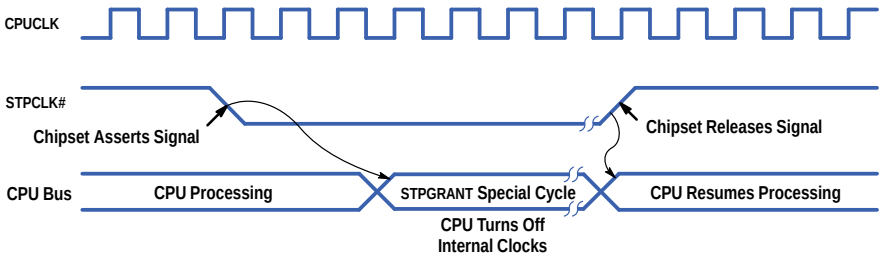
The system DRAM can be arranged into four logical banks in two physical slots. The user can select from 8M to 64M bytes of memory in granularity of 8, 16, 24, 32, 40, 48, and 64M bytes by combining 8M-byte, 16M-byte, and 32M-byte modules. The DRAM supports self-refreshing, so when the HP OmniBook 5000 is suspended, the DRAM retains its contents without clocks or signals from the host bridge. All DRAM memory is removable and accessible which makes it easier to replace and upgrade.

PCI-Based I/O Architecture

The HP OmniBook 5000 I/O bus architecture is designed around the Peripheral Components Interconnect (PCI) bus. The PCI bus in the HP OmniBook 5000 is designed to run at 33 MHz for all processor speeds. At 32 bits of data width, the theoretical bus transfer rate at burst speeds is 132 Mbytes per second. This overabundance of data transfer bandwidth allowed us to add many devices to the PCI bus to increase system performance. The video, SCSI, and 16-bit PC Card (formerly PCMCIA) controllers all reside on the PCI bus along with the CPU host bridge and the ISA (Industry Standard Architecture) bridge. The SCSI controller, the CPU host bridge, and the ISA bridge are capable of bus mastering, which allows them to assume control of the PCI bus.

The PCI-to-ISA bridge contains the power management unit that controls the CPU clocking and peripheral power states. The ISA bridge also contains the standard PC-compatible components such as the interrupt controller, the DMA controller, system timers, memory mappers, and the ISA bus interface. This IC translates the 32-bit PCI bus commands and data that are directed to the 16-bit ISA Bus.

The CPU clocks are managed by using a *clock throttling* scheme. This method was developed for the Pentium CPU since it has an internal fractional frequency multiplier to allow higher processor speeds while limiting the external bus speed. Because of this multiplier function, one cannot just slow down the CPU's frequency to reduce power consumption as was done in previous products. Instead, the power unit makes a request to the CPU for permission to stop the clock. The CPU responds by issuing a special bus cycle when it is finished with the current instruction and then stops its internal clock. When an external event such as an I/O or timer interrupt occurs, the power unit releases the request signal and the CPU restarts its internal clock and resumes operation. Fig. 3 shows the timing of the stop-clock function.



CPUCLK	CPU's external bus clock (50, 60, or 66 MHz, depending on model).
STPCLK#	Signal asserted by the chipset to alert the CPU to stop its internal clock.
CPU Bus	64-bit data bus of the Pentium CPU. A special cycle is asserted which states that the CPU acknowledges the STPCLK# signal.

Fig. 3. *Timing of the stop-clock function used for power conservation.*

The ISA bridge includes a 32-bit IDE controller, which allows faster disk accesses with newer operating systems. This IC redefines the ISA bus during idle cycles to handle the 16-bit IDE hard drive controller signals. Buffers provide isolation between the IDE and ISA buses and allow the hard disk to be powered off while the system continues to function.

Video Controller. The HP OmniBook 5000 video controller features LCD panel (DSTN or TFT) and external video interfaces with VGA or SVGA options. In addition, the video system provides NTSC/PAL video output (the video used in VCRs or camcorders) to allow presentations to be displayed on a standard TV monitor or recorded on a VCR for playback at a later date.

The video controller resides on the PCI bus so that data from the CPU is quickly written to the video memory. The video memory provides a 1M-byte video space for up to 64 million colors in 640 × 480 VGA mode. It also has a 0.5M-byte frame buffer to increase the performance of the dual-scan (DSTN) LCD. The memory on the video controller operates at 3.3V to conserve power and reduce heat. The video DRAM's self-refresh mode preserves the video screen data while the HP OmniBook 5000 is suspended and clocks to the video controller are stopped.

The video circuits automatically detect when a user connects either SVGA or NTSC/PAL video cable to the HP OmniBook 5000. The BIOS will detect the appropriate monitor type and enable the proper video settings based on the selections made in the system setup utility. This feature makes it easier for the user to set up presentations quickly.

16-Bit PC Card Controller. The PC Card controller resides on the PCI bus to allow for several features. By making this controller a PCI device, we are able to map it to a different I/O address should there be a conflict with a legacy ISA card in the docking station (described later). This design makes it possible to support new PC Card standards (e.g., Cardbus, based on 32-bit PCI) in the docking station by remapping or disabling the internal controller. We are also able to increase the system performance by having PC Card accesses bypass the bus translations that would be necessary with separate PCI, ISA, and PC Card buses as in the classical implementation. Performance is also increased because devices that support DMA (e.g., the sound card) do not slow down CPU-to-PC Card transfers (i.e., networks). The HP OmniBook 5000 design has two slots for either two type II cards or one type III (double-height) card. The interface software allows the cards to be shut off completely during suspend mode and to be restored when operation resumes. Device drivers are designed to support advanced power management calls (described later) to prevent disk corruption during suspend mode.

SCSI Controller. The SCSI controller option on the HP OmniBook 5000 is provided to give the user a simple way of connecting to external devices such as CD-ROM, external hard disks, backup tape drives, and scanners. This versatile interface provides a high data transfer rate (up to 10 Mbytes per second). The SCSI interface can be disabled when not in use to conserve power. A SCSI BIOS is provided so no drivers are required to operate a SCSI hard disk if it is connected when the system boots. This SCSI BIOS supports the SCAM protocol, which allows the user to set the SCSI IDs for SCAM-equipped devices from the HP OmniBook 5000 keyboard to simplify setup.

Special device drivers were written to support advanced power management calls to prevent read/write operations to the disk from being interrupted when the user tries to go into the suspend mode. This feature helps prevent corrupt disk images resulting from mobile use, which was not taken into account when the SCSI standard was developed.

Keyboard Controller

The keyboard controller in the HP OmniBook 5000 provides the personality of the notebook. It performs many of the notebook-related tasks so that the Pentium CPU can remain focused on compatibility and performance. Some of the tasks performed by the keyboard controller are:

- Keyboard scanning
- Support for three PS/2 ports (internal trackball, external mouse, external keyboard)
- Status panel control
- Battery charging and low-voltage monitoring
- Battery capacity gauge communication and tutoring
- Temperature sensing and thermal feedback control
- Interface to EEPROM for passwords, PC Tattoo, serial number, and other system tunable parameters
- Docking station control
- Power on/off control.

Because of the complexity of the tasks it has to perform, the keyboard controller is flash-memory-based so that it can be reprogrammed in the field along with the system BIOS and the EEPROM that contains user and system tunable parameters for battery charging, voltage detection, and so on. A special technique was implemented to perform the in-circuit programming of the flash memory in the keyboard controller. First, a bootstrap program is downloaded to the keyboard controller RAM space and executed. This program takes over the keyboard controller and handles the communication with the Pentium. The Pentium downloads the flash update program into the keyboard controller's RAM. The flash memory inside the keyboard controller is erased and then the new keyboard controller BIOS is transferred to the keyboard controller and programmed into the flash memory. Upon a hard reset, the keyboard controller begins functioning with the new BIOS.

The keyboard controller has eight 10-bit analog-to-digital converter (ADC) channels, which are used to monitor the battery temperature for each of the two battery slots, the CPU temperature, the ambient temperature, both battery slot voltages, and a 2.5V reference. The voltages read by the ADC channels are digitally filtered by the keyboard controller before being used by the system BIOS.

The keyboard controller also handles talking to various I/O ports such as three PS/2 channels, two Benchmarq smart battery channels, and an I^2C interface. Events that have higher priorities are serviced first by the keyboard controller. If the keyboard controller is servicing a lower-priority task and a higher-priority event occurs, it is processed first and the lower-priority task

is rescheduled. The highest-priority tasks are docking and low-battery detection, then keyboard and mouse-related tasks, followed by housekeeping chores of battery charging, battery capacity monitoring (see below), and status panel updating.

Smart Batteries with Tutor Assist

ICs in the HP OmniBook 5000 battery packs allow the battery packs to retain their charge state and status information. If the battery is removed from the HP OmniBook 5000 and used or charged in another device, the capacity of the battery is reread from the pack when it is reinserted. These "smart" batteries can measure their temperature and current flow and perform compensated updates of the battery capacity gauge so that, over time, the gauge contents do reflect the state of the battery.

Although there is quite a bit of intelligence designed into the battery capacity gauge circuits, there are events and boundary conditions that result in the gauge's not representing the capacity of the battery accurately. One example is the first time the battery pack is assembled. The capacity gauge requires that the battery pack be discharged and then charged back to a full state to calibrate and initialize the battery capacity reference. So that the user does not have to perform this function, the HP OmniBook 5000 will calibrate the pack under certain conditions when it knows the battery's charged state. At those times, the HP OmniBook 5000 will compare the pack's battery capacity reading and if it disagrees with the HP OmniBook 5000's own gauge by a fixed factor, the pack's gauge will be updated. This tutoring approach provides a closed-loop system to help correct for any inaccuracies caused by component tolerances, bad assumptions by the smart battery, current crest factor corrections, and charging inefficiencies. With this approach, the user does not have to be involved in the recalibration of the battery pack. The philosophy is that, even with machines, two heads are better than one.

Power Supply and Battery Charging

The power supply generates 3.3V, 5.0V, and 12.0V. It can charge either a NiMHy battery using a constant current source or a Li-Ion Battery using a constant current/constant voltage source. The ac adapter is the same as that used by the HP OmniBook 600 Series. It provides an output of 12V at 3.3A maximum current or a total power of about 40 watts. Half of the adapter wattage is used to operate the product and half is used to charge the batteries in the system. The battery voltages for both the NiMHy and Li-Ion packs can each be higher or lower than the +12V adapter voltage, depending on the charge state of the cells. A flyback configuration is used in the adapter since it can be designed to support this voltage span (see subarticle "***Flyback Charger Circuit***"). The power to the batteries during charging is limited by both current and voltage sensing. An important convenience goal of the HP OmniBook 5000 was to charge and operate at the same time. This feature allows the user to operate the HP OmniBook 5000 during the day and still have full battery power for ready use after disconnecting the computer from the ac adapter.

The 3.3V and 5.0V regulators use a synchronous switching topology that allows the power supply to maintain a high level of efficiency. The +12V output is derived using a transformer tap on the 3.3V inductor to generate about 14V, which is then linearly regulated back to +12V. This technique generates a very clean and stable +12V to program the system flash memory, the keyboard controller flash memory, and any PC Card memory cards. Sometimes the +12V is used for analog circuits on PC Cards, so we felt it necessary to provide a filtered signal.

One challenging design parameter of the power supply is that it must deliver about 15W during maximum operation and also maintain regulation while supplying less than 100 mW to the system in suspend mode.

Docking Strategy

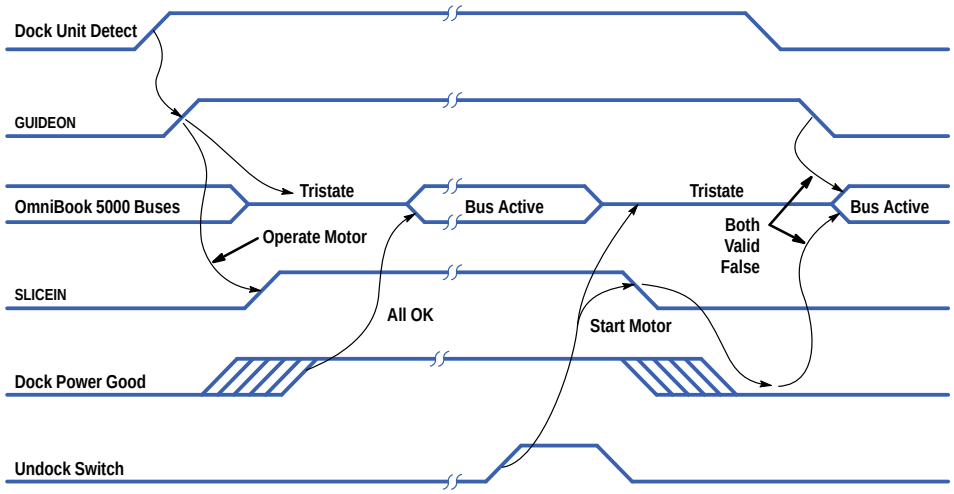
The HP OmniBook 5000 docking station (Fig. 4) provides the docked computer with one PCI slot and 2 ISA slots, giving the user access to the same options as desktop users. The docking station provides one-handed, power-assisted docking (VCR style). The I/O ports on the HP OmniBook 5000 are replicated on the docking station so that the user does not have to remake a lot of cable connections. Some ports, such as the sound system (line in, line out, and microphone), MIDI, and SVGA out ports, are passed straight through the docking station. Other interfaces such as the SCSI, RS-232, parallel, and game ports are replicated by using the same ICs as the HP OmniBook 5000's internal chips, which are disabled. By using the plug and play features of the BIOS, the docked HP OmniBook 5000 can have either the same configuration as the portable HP OmniBook 5000 or a different configuration.

Control of the docking sequence is handled by the keyboard controller using some control signals and the I^2C bus. The I^2C bus is used to read and write additional signals and to save and restore configuration information for the docking station. Fig. 5 shows the timing of the docking sequence.

The user initiates the docking event by placing the HP OmniBook 5000 on the docking station's receiving tray and providing a gentle shove. When the docking station's detection switch is activated, the motor engages and captures the HP OmniBook 5000 and draws it onto the connector. When the docking connector guide pins begin to mate into the HP OmniBook 5000 the keyboard controller is notified by a nonmaskable interrupt and it notifies the chipset that a hot-dock event is about to take place. The chipset then halts the Pentium processor and tristates the buses. When the two connectors mate, the keyboard controller is notified by a signal on the connector. The keyboard controller verifies that the docking station has a valid power good signal, and if it does, the keyboard controller signals the chipset to release the Pentium and begin driving the PCI and ISA buses. The BIOS first disables the internal I/O chips and configures the I/O chips on the docking station before allowing the user to resume work. Depending on the operating system on the HP OmniBook 5000, a reboot of the operating system may be necessary to load device drivers for SCSI peripherals or other cards that may be plugged into the slots.

Fig. 4. *HP OmniBook 5000 computer in docking station.*

The undock sequence begins when the user either presses the undock key on the docking station or initializes an undock sequence via the operating system (Windows® 95). When the keyboard controller receives the undock event it signals the chipset to halt the Pentium once again and releases control of the buses. It then signals the docking station via the I^2C bus to begin the undock sequence for the motor. The motor starts and begins to eject the HP OmniBook 5000 from the docking station. The keyboard controller detects that undocking has occurred by monitoring the docking detect signal. It then tells the chipset to release the Pentium and drive the buses again. The BIOS then reconfigures the I/O devices back to the mobile configuration. Again, depending on the operating system and selection of peripherals, it may be necessary to perform a reboot.



Dock Unit Detect	Sensor that detects that the OmniBook 5000 is present and ready for docking. Starts the dock motor for VCR-style docking. Moves the OmniBook 5000 into the dock connector.
GUIDON	Sensor that detects that connectors are about to mate. Alerts the keyboard controller to begin the docking sequence to tristate the system buses.
OmniBook 5000 Buses	PCI and ISA buses are tristated to prevent data corruption during docking and undocking sequences.
SLICEIN	Signal that the connectors have mated.
Dock Power Good	Signal read by the keyboard controller to verify that a good docking event has occurred and the system can resume. If the dock power good signal is not detected, the system is powered off.
Undock Switch	Button or software event to initiate the undock sequence.

Fig. 5. *Docking sequence timing diagram.*

CPU Thermal Management

The Pentium CPU has a maximum thermal envelope of about 7.5W. This power is dissipated as heat in the CPU. The HP OmniBook 5000 combines many techniques to remove the heat from the CPU to ensure both functionality and reliability of the product. The tape carrier package (TCP) version of the Pentium is used to allow the case temperature of the CPU to be higher than is allowed in the standard ceramic package. The TCP package is essentially a metal plate attached to the backside of the CPU die. The pads on the circuit side of the CPU connect to thin-film conductors on a piece of cellulose (which looks similar to 35-mm film) rather than using traditional bonding wires. The top side of the die and film are then covered with an epoxy coating to protect the bond connections and to seal the die from contaminants. The net effect of this package is that the thermal heat resistance from the CPU to the outside mounting pad of the package is minimal. This feature allows us to operate the case temperature at 95°C versus 75°C for the PGA package.

The TCP package has very little thermal mass, so we need to move the heat energy away from the package and out into the product. Aluminum cast heat sinks are attached to two sides of the CPU package, both to the package epoxy and to the backside of the package through the use of many via holes under the die. This approach moves the heat energy away from the CPU to a larger mass for further disposal out of the product.

To get the heat out of the product the cast heat sink is attached to an extruded aluminum heat sink, which attaches to the back aluminum I/O panel. This heat sink has the ridges found in typical heat sinks so that the heat can convect out of vents in the top case.

To remove additional heat from the cast CPU heat sink, a heat pipe technology is used (see Fig. 6). A heat pipe is basically a small Carnot engine that transfers heat from one end of the pipe to the other via a temperature differential, using a wicking action to recycle the fluid. The fluid in the pipe is just water that has been depressurized to allow it to boil at a lower temperature. When the CPU heat sink is heated to the boiling point of the water in the heat pipe, the water changes state from liquid to gas. This phase change extracts a large amount of thermal energy from the heat sink. As the water boils, the wick inside the heat pipe brings cooler water to the CPU end and the gas moves down the heat pipe to the condensing end. When the gas in the heat pipe cools below the boiling point, the energy taken from the CPU is released into the condenser. The condenser consists of a thin aluminum stamped sheet. This sheet is attached to the back of the metal plate of the keyboard and the heat is spread out over the keyboard surface area.

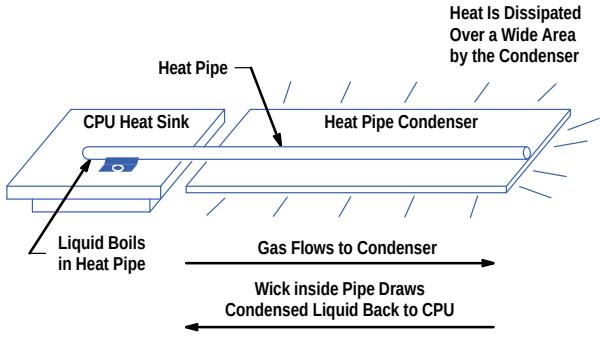


Fig. 6. *Heat pipe principle.*

To prevent the CPU from generating excess heat during typical operation, the chipset provides different methods of controlling the CPU activity with the use of *advanced power management* (APM). Software APM is the most effective. It consists of having a running program shut the CPU clock down when there are periods of inactivity in the program. Of course, this method requires the software to be APM compliant. If the user is operating software that is not APM compliant, the HP OmniBook 5000 can be set to use hardware APM as a default. Hardware APM is implemented by monitoring the interrupts and major I/O function accesses. When the program is performing any I/O activity or the I/O devices generate an interrupt, a timer is reset that keeps the CPU active from one half second to eight seconds, depending on the setting. If another event occurs before the timer times out, the timer will be set back to the full count. If the program is just executing out of the CPU memory and the timer expires, then the CPU clock is throttled back to 1/4 speed. This reduction lowers the power used by the CPU by about 75%. Access to an I/O event will again set the timer to the full count and enable the CPU to run at full speed again.

To prevent the CPU from reaching the critical case temperature, active thermal feedback is employed. This technique measures the temperature of the CPU module and the ambient air inside the unit. The temperature limits for each are monitored by the keyboard controller. If either limit is exceeded, the CPU speed is reduced by clock throttling. When the CPU and ambient temperatures cool below their restart thresholds, the keyboard controller will again allow the CPU to work at full speed. For the thermistor on the CPU module, the feedback is fairly rapid because of the mechanical heat removal methods, and the CPU tends to run at an average clock speed at which thermal equilibrium is reached for the given operating conditions. In the case of the ambient air sensor (used to protect other ICs in the notebook from overheating) the effect of slowing down the CPU is not very rapid because of the large thermal resistance between the sensor and the CPU

module. If this sensor trips the thermal feedback, the unit will run at a slower speed (1/2 clock rate) until the air temperature returns to a cooler level. This rate was chosen so that the user can still operate the machine even if it takes several minutes for the ambient air to cool.

Summary

The technologies developed for the HP OmniBook 5000 computer are designed to achieve the notebook features required for the future. The PCI bus will allow higher speed and functionality in video and communications than is possible today. The heat transfer and modular assembly technologies will permit incorporation of new faster processors as they become available. The Li-Ion batteries will continue to provide more energy for a given weight, thereby making possible lighter products or longer battery life. The instant-on feature allows the user the freedom to work whenever it is convenient, thus helping the customer adapt to a more mobile lifestyle. The PCI and ISA hot-docking technology allows the user to have full desktop functionality and performance when purchasing a notebook computer.

Acknowledgments

The design of the HP OmniBook 5000 posed many challenges related to the new technologies of PCI, Pentium, plug-and-play, and hot docking. Many of the product's most significant features would not have been possible without the efforts of many people both within and outside of Hewlett-Packard. Kevin Quan and Justin Maynard of Systemsoft dedicated themselves to the design and implementation of the system and keyboard BIOSs. Without the tireless efforts of Shen Wang, Donald Chen, and Gwo-Huang at Twinhead, the transition from a product concept to a producible notebook would have been much more difficult. Many HP team members devoted their time and energy to make the OmniBook 5000 the finest product possible. Dan Rudolph was instrumental on the video, SCSI, PCMCIA, and sound designs. Andy Van Brocklin worked ceaselessly on getting the product qualified for EMI submittal. Tracy Lang's enthusiasm and diligence on the mechanical design ensured a solid-fitting product. The HP software team headed by Eric Evett along with Stan Blascow, Dan Pinson, and Everett Kaiser defined how the OmniBook 5000 would function for the customer and ensured that the completed product was as close to the original specification as possible. Of course, there were many others that contributed to the product's development that cannot be named here. Special recognition should be given to Terry Bradley and Bill Wickes, who as project and section managers, respectively, were able to coordinate and lead a team of creative people spanning the world while focusing individual team member efforts with patience, diligence, and continual motivation.

Flyback Charger Circuit

The flyback transformer design of the HP OmniBook 5000 charger circuit can provide a wide range of output voltages because of the energy storage, coupling, and isolation of the transformer (see Fig. 1). During the first portion of a clock cycle, energy is stored in the primary side of the transformer. During the second portion of the clock cycle, the energy is transferred to the secondary winding via the core's coupling. Because the current in the primary is cut off abruptly, the secondary voltage can become higher than the input (dI/dt is high and V = LdI/dt).

The secondary output current charges a large output capacitance. Using a high-frequency clock, the output capacitor's voltage can be charged up to any desired level and then the converter's control circuitry can be shut down until the output voltage has decayed by some desired amount for regulation.

The isolation of the secondary winding allows for both voltage and current sensing to control the flyback operation. This permits both current and voltage limiting, which are required with Li-Ion batteries. For current sensing, the voltage across the sense resistor is filtered and compared to a set reference. When the output voltage has risen high enough to cause the desired current in the battery, the control circuit is turned off. Again, when the current is reduced by a desired amount to maintain regulation, the control circuit is reactivated.

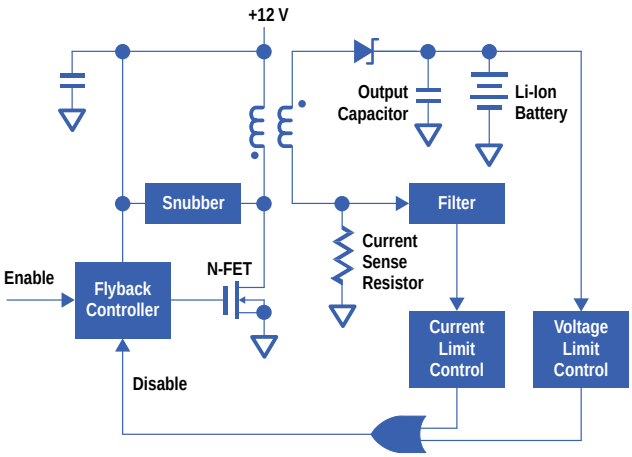


Fig. 1. *Flyback charger circuit.*

A Graphing Calculator for Mathematics and Science Classes

The HP 38G calculator allows teachers to direct students and keep them focused while they explore mathematical and scientific concepts. It features aplets, which are small applications that focus on a particular area of the curriculum and can be easily distributed from the teacher's calculator to the students'.

by Ted W. Beers, Diana K. Byrne, James A. Donnelly, Robert W. Jones, and Feng Yuan

The HP 38G calculator is a graphing calculator for students and teachers in mathematics and science classes. It features aplets, which are small applications that focus on a particular area of the curriculum and can be easily distributed from calculator to calculator. This allows the teacher to send an electronic story problem to each student in the class.

The HP 38G is built on the same software platform as the HP 48G family of graphing calculators,[1] but has a simpler user interface and feature set. Equations are entered using algebraic format rather than the reverse Polish notation (RPN) found in most HP calculators. The features of the HP 38G include:

- Graphical user interface
- Function, polar, parametric, stairstep, cobweb, histogram, scatter, and box and whisker plots
- Side-by-side split screen
- Tables
- Unlimited, scrollable history stack
- Symbolic equations
- HP Solve numeric root finder
- EquationWriter display
- Statistics functions
- Matrix operations
- User programming.

The hardware platform of the HP 38G is very similar to that of the HP 48G: they both have 32K bytes of RAM, 512K bytes of ROM, the same CPU and the same display (131 by 64 pixels). They both have a two-way infrared link for sending information to a printer and for transferring information between two calculators, and an RS-232 link for calculator-to-computer communications. An accessory that allows the calculator screen to be displayed with an overhead projector works with both the HP 48G and the HP 38G, but the HP 38G cable connector has been modified so that the overhead display accessory works with every HP 38G; no special handset is required.

The HP 48 case was redesigned to include a sliding plastic cover to make the HP 38G more durable for use by younger students. Also, two keys were removed to give visual emphasis to the navigation keys and to make the keyboard look less complicated.

Designed for and with Teachers

We set out to design a graphing calculator for precalculus students and teachers. To help us do this, we formed an Education Advisory Committee consisting of eight high school, community college, and university teachers. We met with the teachers every few months, and between meetings we kept in touch by email.

The teachers told us that they want to allow students to explore mathematical and scientific concepts, and at the same time they need to direct the students and keep them focused. In our first meeting with the teachers, we compared this to the idea of a child's sandbox: the child is given toys for playing and exploration, but within a protected, specialized environment. Thus, one of our main goals with the calculator software design was to encourage exploration by limiting choices. This led us to the concept of *aplets*: an aplet is a small application that focuses on a particular problem.

HP 38G Aplets and Views

We based the design of aplets on the National Council of Teachers of Mathematics "three views": graphic, symbolic, and numeric. Each problem can be explored using these different representations. For example, a mathematical function can be expressed as a graph (Fig. 1a), in symbolic form (Fig. 1b), or using a table of numbers (Fig. 1c).

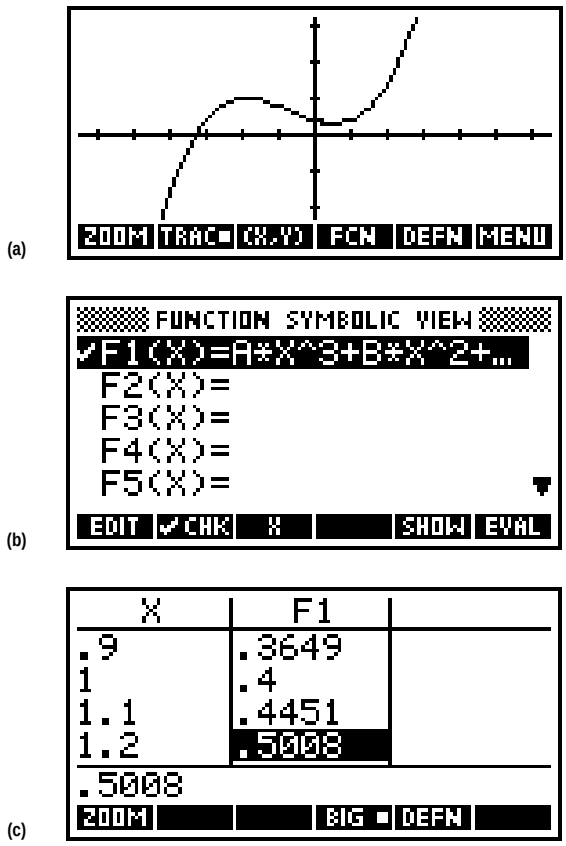


Fig. 1. *With the HP 38G calculator, a mathematical function can be expressed (a) as a graph, (b) in symbolic form, or (c) using a table of numbers.*

Aplets can be created by teachers (either directly on the HP 38G or with the assistance of a computer) and then "beamed" to the students' calculators using infrared transfer. This way, a whole classroom full of students can have their calculators programmed identically at the beginning of a lesson. Then, the students can explore within the aplet on their own calculators.

Each aplet packages the formulas, settings, and other information associated with a particular problem. If the user wants to switch from one aplet to another, this can be done without disturbing any individual aplet's settings, since they are compartmentalized.

Several aplets are built into the HP 38G. When the calculator is first turned on, these built-in aplets are empty. The user must add some information, such as equations, notes, or sketches to make these aplets come alive.

Using Aplets on the HP 38G

Six different types of aplets are built into the HP 38G: function, parametric, polar, sequence, statistics, and solve. Typically, a teacher will start with one of these aplet types, then customize it by adding particular functions that define a certain problem, together with settings, pictures, and text directions.

To start using a particular aplet that is already in the calculator, the student chooses it from the HP 38G library by pressing the **LIB** key. Continuing to explore the problem, the student may want to view the problem in different ways, and can do this by pressing the different view keys. **PLOT**, **SYMB**, and **NUM** display the graphic view, symbolic view, and numeric view, respectively. Additional views, such as split-screen views (Fig. 2), can be found by pressing the **VIEWS** key.

The student or teacher can customize the graphical, numeric, and symbolic views for a specific problem by using the setup screens. It is also possible to annotate the problem by typing some text into the note view or by adding a sketch to the sketch view.

The teacher and student can easily generate custom aplets that present new examples based on the built-in aplet types. An aplet is created by working with an aplet type and adding all of the customizing information that relates to a particular problem (see ***Article 7***).

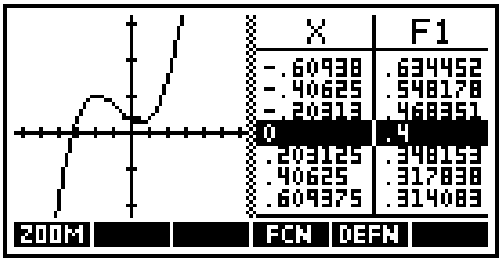


Fig. 2. *The plot-table view is a typical split-screen view.*

Main Components

The keyboard (Fig. 3) reflects the seven main components of the HP 38G's functionality (from top to bottom on the keyboard):

- Softkeys for accessing custom behavior defined by softkey labels along the bottom of the display
- View keys for moving among the possible representations of aplet data
- Library key for selecting the current topic of exploration with aplets
- Arrow keys for screen and menu navigation
- Home calculator keys for numeric entry and basic calculator operations
- Alpha keys for access to alphabetic characters
- Editing environments for creating, editing, and managing objects such as programs, matrices, lists, and notes.

Four of the key groupings are related as follows: first, a topic is chosen with the library key. Then, a way of looking at the problem is chosen with the view selection keys. Finally, the peculiarities of the problem view are manipulated with the softkeys and the arrow keys.

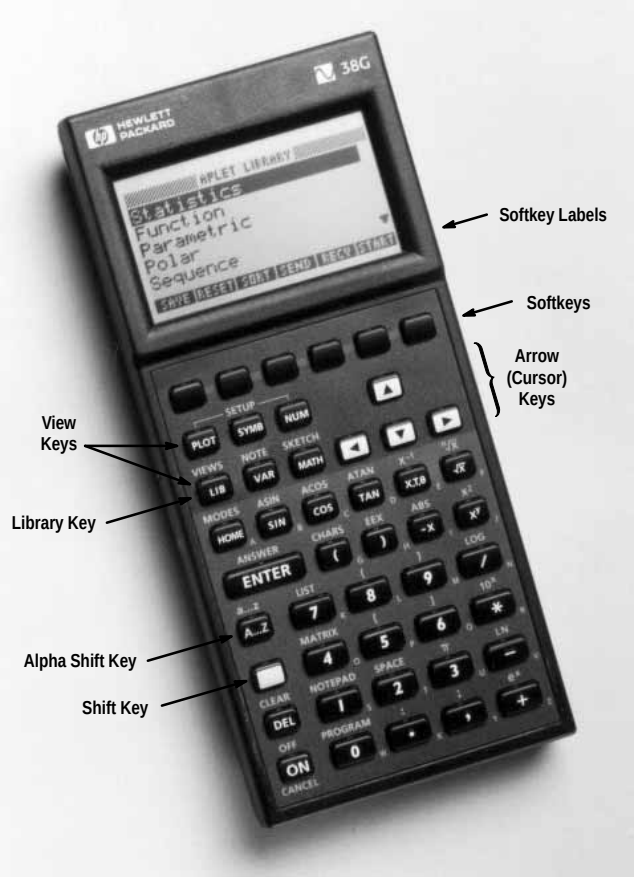


Fig. 3. *HP 38G calculator.*

Softkeys

The softkeys are the top row of unlabeled keys, which are associated with the dynamic menu labels displayed along the bottom of the screen. They give access to context-specific custom behavior. Unlike the HP 48G, there is no “next-row” key since all HP 38G softkey sets are limited to six items for the sake of simplicity.

View Keys

The view keys provide one-key access to the three National Council of Teachers of Mathematics representations of aplet data: graphical, numerical (tabular), and symbolic, plus annotation and sketch views for documenting a topic. Keys for setting up view parameters and managing split-screen views are also included.

- The plot view gives a conventional mathematical graphical representation. In most cases it looks like some flavor of plot output in the HP 48G.
- The numeric view is generally a table of sampled values. It is a derived form of the mathematical object (except for statistics, where it is the defining view).
- The symbolic view is generally an expression representing the mathematical object of interest. The variables are defined by the aplet. This view is the defining form of the mathematical object (except for statistics, where it is derived).
- The note view is a mini word processor that allows the user to create, edit, and view a text document associated with the aplet.
- The sketch view is a set of standard-sized GROBs (graphic objects in HP 48G terminology) that explain the “story” of the aplet. The user can create, edit, view, and animate pictures. Editing capabilities include lines, boxes, circles, text labels, and Etch-a-Sketch-style drawing.

Moving from view to view is the same as task switching, which means the user can always go on to the next task, but there is no concept of going back to somewhere, since tasks are not being nested.

The **LIB** key invokes the library, which is the aplet selection and management environment, in which different aplets (including those that are not built-in) can be started, exchanged with others, and created by saving the state.

The **VAR** key provides access to variables and the **MATH** key provides access to scientific functions and other operations and commands. These variables and operations are available whenever the user is using an edit line.

Invoking the **VAR** or **MATH** menu starts a subtask, which must be completed or cancelled, at which point the calculator is back where it was when the subtask was started.

The Library Environment

This environment gives high-level access to aplets. The user can select an aplet and manage the current collection of aplets. From within the library, the user can take a snapshot of a built-in aplet, giving it a name and a directory of its own. This is how aplets are generated for dissemination and how users show their work. Also, the user can import or export aplets from the library to another HP 38G or to a computer.

Arrow Keys

The arrow keys are used for all direction-oriented operations such as tracing a function plot, moving among fields in an input form, and selecting commands from a pop-up menu.

The shift key can be used as a modifier for the arrow keys that signals motion “all the way” in the direction indicated.

Home Calculator Keys

The home calculator environment gives a familiar tool with a nice graphical interface. The user invokes it by pressing the **HOME** key. Inputs are displayed on the left side of a line, with the results displayed on the right side of the next line.

The calculator keys are used to type numbers and to access basic scientific calculator functions. The **ENTER** key is used to terminate data entry, to select operations from menus, and in general, to make things happen. Some mathematical operations are available on the keyboard and other operations and commands are available through the **MATH** pop-up menu.

The **ANSWER** shifted key gives access to the variable called Ans, which always contains the last result.

Alpha Keys

The alpha shift key, **A...Z**, provides access to the alphabetic characters, which are labeled on the keyboard overlay. Pressing the **CHARS** shifted key invokes the character browser, which provides access to characters that are not on the keyboard.

Unlike the HP 48G, there is no alpha lock key to confuse the user. The user can either press and release the alpha shift key before pressing each alpha character key, or hold the alpha shift key down while pressing as many alpha character keys as desired. However, an alpha lock toggle softkey is provided in some editing environments.

Editing Environments

The HP 38G has specialized environments for managing programs, matrices, lists, and notes. When the user invokes one of the editing environments, a scrolling choose list of all the existing objects of that type appears, together with a softkey menu. From this environment, objects can be transferred between the HP 38G and a computer or another HP 38G.

- The program environment provides tools for creating, editing, storing, sending, receiving, and running programs. Variables can be accessed through the **VAR** pop-up menu. Mathematical operations can be accessed from the keyboard or through the **MATH** pop-up menu, and programming commands can be accessed through the commands section of the **MATH** pop-up menu.
- The matrix environment provides a two-dimensional matrix editor for creating, editing, viewing, sending, and receiving matrices.
- The list environment provides a list editor for creating, editing, viewing, sending, and receiving lists.
- The notepad environment provides an environment for creating, editing, viewing, saving, sending, and receiving text documents. The tools for editing notes in the notepad environment are identical to the editor for the note view. The documents created in the notepad environment are not bundled with an aplet as they are in the note view. The notepad environment can be used for tasks such as creating, storing, and viewing lists or notes.

User Interface

One of our goals for the HP 38G project was to leverage software from the HP 48G/GX calculator platform. For the HP 48G/GX we developed two primary general-purpose user interface tools: input forms and choose boxes.[1]

Input forms and choose boxes are interactive environments that are used to gather user input for a task. Input forms (see Fig. 4) are flexible, screen-sized dialog boxes similar to those in other graphical user interfaces. The appearance and use of input form fields resemble spreadsheet programs. Choose boxes (see Fig. 5) are either pop-up or screen-sized boxes optimized for selecting or working with one or more items from an arbitrarily long list. Input forms and choose boxes, along with some other user interface constructs, contribute to the improved ease of use that distinguishes the HP 48G/GX from its predecessor, the HP 48S/SX.

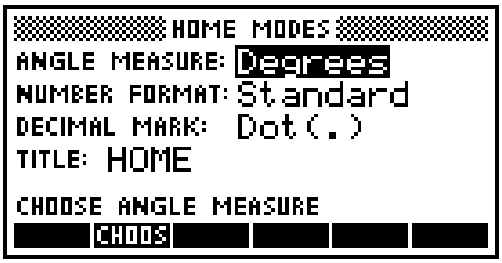


Fig. 4. *A typical input form.*

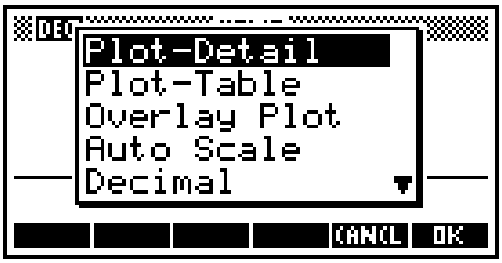


Fig. 5. *A typical choose box.*

In the HP 38G, we found that input forms and choose boxes were a good match for the needs of the utility environments and most of the nine views available for working with aplets.

Graphical User Interface Applications

Input forms are used in most aplet views for entering a mix of settings and values. They're also used for gathering input to complete a task. For example:

- The function aplet's plot setup view is an input form in which modes and parameters are specified (see Fig. 6).

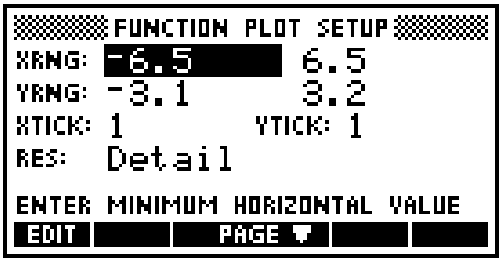


Fig. 6. *The function aplet's plot setup view input form.*

- The solve aplet's numeric view is an input form whose appearance varies according to the equation to be solved (see Fig. 7). The flexible layout and labeling of input form fields are employed to generate a custom input form each time the view is entered.

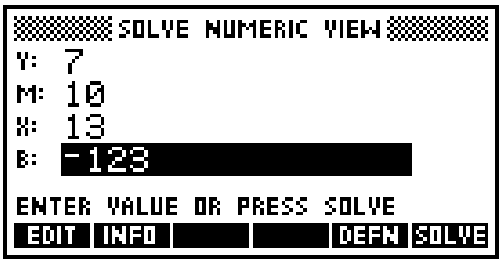


Fig. 7. *The solve aplet's numeric view, a dynamically generated input form.*

- When an aplet is to be saved, a simple input form is displayed to get the new aplet's name (see Fig. 8).

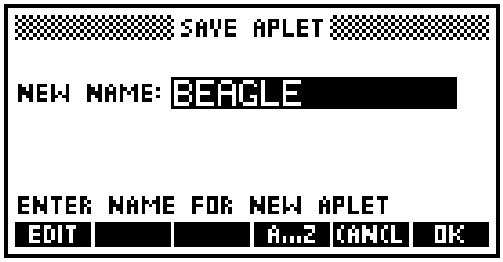


Fig. 8. *The aplet library's save-aplet input form.*

- The statistics symbolic view is a highly customized input form that scrolls to show more information than fits on one screen (see Fig. 9). Choose box elements, like the up and down arrows indicating more information is available offscreen, help suggest the operation of this hybrid view.

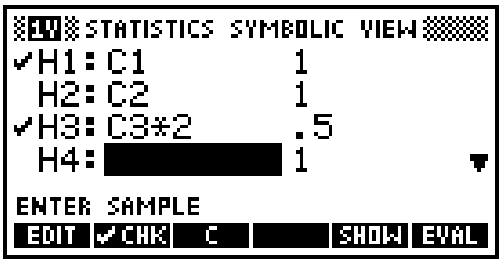


Fig. 9. *The statistics aplet's symbolic view, a custom scrolling input form.*

Choose boxes are used in a variety of views and utility environments wherever a list of related items needs to be managed. Here are a few examples of choose boxes in the HP 38G:

- The aplet library (see Fig. 10) is actually an elaborate choose box.
- Almost all symbolic views, such as the function aplet's symbolic view (see Fig. 11), are choose boxes.

Fig. 10. *The aplet library, an elaborate choose box.*

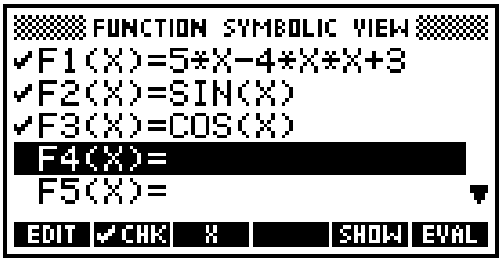


Fig. 11. *The function aplet's symbolic view.*

- Choose boxes pop up within input forms like the **MODES** input form (see Fig. 12).
- The **VAR** and **MATH** menus (see Fig. 13) are two-column choose boxes that show categories of items plus the items in the selected category.

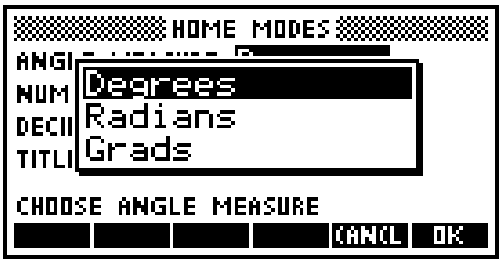


Fig. 12. *A pop-up choose box in the* **MODES** *input form.*

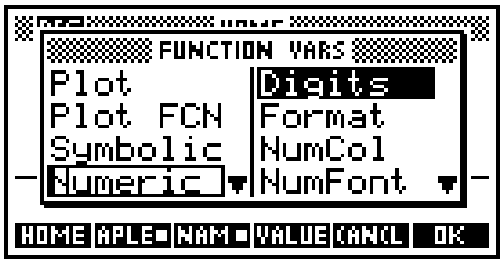


Fig. 13. *The* **VAR** *menu, a two-column choose box.*

As these examples illustrate, the look and feel of HP 48G/GX input forms and choose boxes remain largely unchanged in the HP 38G. However, substantial reengineering of the underpinnings of these tools was required to match other aspects of the HP 38G's use model, as discussed in the next section.

Topic Outer Loop

Many of the custom interfaces developed for the HP 48S/SX used an RPL-language tool we developed called the parameterized outer loop.[2] Parameterized outer loop applications depend on the parameterized outer loop for routine key and error handling and display management. The graphical user interface (GUI) elements introduced in the HP 48G/GX are also parameterized outer loop applications that automate routine matters of input entry, selection of options, and presentation of output.

In both the HP 48S/SX and the HP 48G/GX, the user interface was based on the notion of having a central environment (the user stack) from which other applications are launched and to which applications return when completed. All applications on these platforms, including parameterized outer loop applications like the GUI tools, are based on this "function call" model: they start, run for a while, then end, returning the flow of control to where they started. We call such applications *modal*.

New Model, New Tool

When we were investigating the use model for the HP 38G, it became apparent that the function call approach to application management would not suffice. The HP 38G is a tool for exploration, so we wanted to provide an environment that promotes wandering from one subject area to another, or in HP 38G terms, from one view or aplet to another.

To attain this goal we quickly determined that the modal nature of the parameterized outer loop and the applications based on it was too constraining, yet we weren't prepared to discard the wealth of useful tools and concepts we had built up from the parameterized outer loop foundation. Furthermore, we knew there were still plenty of times when the modal call-and-return interface would still apply, such as when pausing to get further input from the user before proceeding with a task. To accommodate all these needs, we developed the new *topic outer loop*.

Topic Outer Loop Overview

Where parameterized outer loop applications are designed to preserve the environment from which they're launched and later restore that environment, topic outer loop topics are optimized for rapidly setting up and switching from one topic to the next. Except with regard to the home environment from which the topic outer loop is originally launched and to which it ultimately returns—after running many topics, perhaps—the topic outer loop doesn't preserve or restore a previous user interface since there is none to go back to.

The most obvious examples of topic outer loop topics in the HP 38G are aplets, but many other environments with similar behavior are also topic outer loop topics—for example, the aplet library and the user program catalog (see Fig. 14).

Like the parameterized outer loop on which it's based, the topic outer loop is launched from the calculator's system outer loop and temporarily redefines the current user interface until some exit condition is met. By design, the topic outer loop operates very similarly to the parameterized outer loop, but differs from the parameterized outer loop in two fundamental ways:

- To better support the standard two-tiered structure of HP 38G topics, the topic outer loop manages two nested user interface levels. The parameterized outer loop manages just one.
- The topic outer loop fully supports organized and efficient switching from one topic to another. The parameterized outer loop is designed to shut down completely before launching another application.

The operation of the topic outer loop for starting a topic can be summarized as follows:

```
If a topic outer loop is already running {

  Evaluate the old view exit handler
  Evaluate the old topic exit handler
  Set the topic entry and exit handlers
  Evaluate the topic entry handler
  Set the view entry and exit handlers
  Evaluate the view entry handler
  Set the remainder of the view user interface

}

Else {

  Save the home user interface

  If error in {

    Set the topic entry and exit handlers
    Evaluate the topic entry handler
    Set the view entry and exit handlers
    Evaluate the view entry handler
    Set the remainder of the view user interface

    If error in {

      While not done with the topic outer loop {
        Evaluate the view display object
        Read a key and get its custom definition
        If error in
          Evaluate the key definition
        Then
          Evaluate the error handler object
        }

      Evaluate the view exit handler
      Evaluate the topic exit handler

      }
    Then {
      Evaluate the view exit handler
      Evaluate the topic exit handler
      Error

      }

    }

  Then {

    Restore the saved home user interface
    Error

    }

  Restore the saved home user interface

}
```

The code setting up the topic specifies the user interface and other environment elements unique to the topic, such as the topic entry handler and the view display object, when it runs the topic outer loop. This is how the behavior of the topic outer

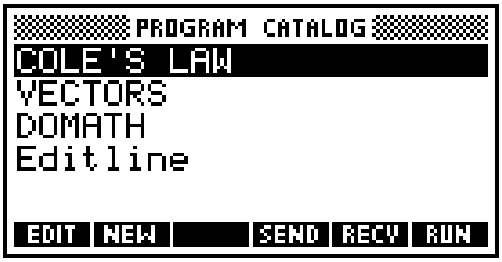


Fig. 14. *The user program catalog, a topic outer loop utility environment.*

loop is customized. The topic outer loop is responsible for the key-display loop, low-level error handling, and juggling the topic and view entry and exit handlers and the saved home user interface.

When an event occurs that calls for running the topic outer loop, the topic outer loop may or may not already be running. As the first section of the topic outer loop overview illustrates, if a topic outer loop is already running, switching to a new topic is quick yet still gives the exiting topic an opportunity to shut down in an orderly fashion. Since it's common with the HP 38G to move from topic to topic without first returning home, this efficiency translates to faster performance.

Switching among views within the same topic is also common, and involves a similarly efficient set of operations:

```
Evaluate the old view exit handler
Set the view entry and exit handlers
Evaluate the view entry handler
Set the remainder of the view user interface
```

Reengineering the GUI Tools

Although very similar in specifications and use to the standard modal input form and choose box environments, versions of these environments based on the topic outer loop, which we call *modeless* environments, differ in the following ways:

- OK and cancel keys are nonfunctional.
- The default softkey set does not include OK and cancel keys.
- Task-switching keys are processed normally to allow task switching.
- The results returned by the input form or choose box engine always consist of the confirmed exit values. No flag indicating canceled or normal exit is returned.

To support modeless views and utility environments based on HP 48G/GX GUI tools, we adapted the input form and choose box engines to use the topic outer loop. However, modal input forms and choose boxes are also employed by the HP 38G. Rather than simply switch the engines from using the parameterized outer loop to using the topic outer loop, we modularized the components of the engines to enable the use of either. We then repackaged the modal versions of the engines to ensure backwards compatibility for existing code using them. To make modeless input form and choose box programming as straightforward as possible for programmers familiar with the modal versions, and yet still meet the requirements of the topic outer loop, we developed tools to translate traditional modal input form and choose box arguments and results to and from the specifications required for topic outer loop applications. This greatly simplified the reengineering of existing user interface code to make use of new modeless input forms and choose boxes. The process was largely mechanical, requiring only that the developer follow a few well-documented steps.

Aplets and Views

One of the key ideas of aplets is that they provide different ways of looking at a problem. For example, when exploring a story problem about speed and distance, the student can look at a symbolic expression, a table of numbers, a graph of the distance function, or even a diagram showing a tortoise and a hare. These different ways of looking at an aplet are called *views*. The topic outer loop manages the transitions between the views of an aplet. The views are implemented using the graphical user interface tools plus aplet data. The data associated with an aplet is encapsulated in a directory structure inherited from the HP 48G/GX.[1]

Aplet Structure

Associated with each aplet is a standard set of information. The topic outer loop uses this aplet information for aplet directory checking, topic switching, resetting, and so on. It's also used by the **VAR** menu and the **VIEWS** choose box. The standard information is:

- Topic ID
- Initial view
- Topic name

- Special views data
- Aplet-specific variables
- Topic entry procedure pointer
- Topic exit procedure pointer
- Topic reset procedure pointer
- Aplet directory checker procedure pointer.

An HP 38G aplet is a collection of aplet data and aplet views. Aplet data includes the aplet name, which is used in the library, real variables like Xmin and Xmax, which are set via the plot setup view, symbolic expressions, note text, and sketches. An aplet usually defines at least eight views: plot, symbolic, numeric, note, sketch, plot setup, symbolic setup, and numeric setup. The generic aplet contains items such as:

- Aplet name
- Aplet topic
- Attached library
- Plot view procedure pointer
- Symbolic view procedure pointer
- Numeric view procedure pointer
- Plot setup view procedure pointer
- Symbolic setup view procedure pointer
- Numeric setup view procedure pointer
- Note view procedure pointer
- Sketch view procedure pointer
- Xmin variable
- Ymin variable
- Other shared variables
- Aplet-specific variables
- List of symbolic expressions
- Array of independent values for numeric view
- List of strings for custom views
- Note text
- List of graphics objects for the sketch view.

For aplets built into the HP 38G, pointers like the plot view procedure pointer refer to code built into the 512K-byte ROM. New aplet types can include a RAM-based support library containing new view procedures. All aplets must contain a basic set of variables, but specific aplet types may implement additional optional variables.

View Structure

Each aplet view is managed by the topic outer loop, typically with the help of graphical user interface (GUI) tools like input forms. To customize its behavior, a view (or its GUI helper) specifies objects that are used while the view is visible. These include the view entry and exit handlers, the display handler, and the key handler, among others:

- Initialization procedure pointer
- Exit procedure pointer
- Display procedure pointer
- Key handler procedure pointer
- Non-view-specific-key-allowed flag
- Softkey menu description
- Error handler.

Symbolic View

Normally, the symbolic view is the defining view for an aplet. When the user starts an aplet, its symbolic view will be shown so the user can enter symbolic expressions or equations to be used in the aplet.

The symbolic view is the generalization of the HP 48G/GX EQ list. The EQ variable on the HP 48G is a list of unnamed symbolic expressions, which are plotted and traced together. The HP 38G breaks this into individual named symbolic expressions that have independent check marks. Expressions for a parametric function are further broken into real and imaginary parts. The expression for a sequence is broken into two initial terms and a recursive relation. This simplifies

editing and selection of symbolic expressions. Giving expressions names in the symbolic view allows them to be reused in other expressions, home calculations, and programming.

Expressions entered in the symbolic view are checked for syntax errors and to a limited extent for semantic errors. Expressions defining a sequence are further classified and transformed into an internal form for cache-based iterative evaluation, saving both time and run-time RAM space. An EVAL menu key is provided in the symbolic view for constant expression evaluation, expression simplification, and function unfolding. Fig. 15 shows how the Fibonacci sequence can be defined and checked using ten keystrokes. (The EVAL menu key is shown when a command line is not active. It appears where OK appears in Fig. 15.)

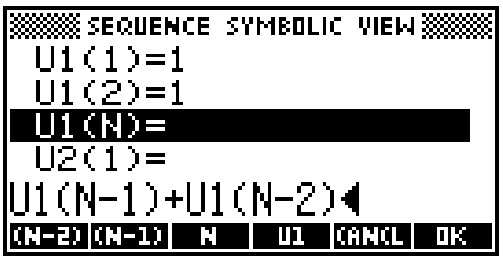


Fig. 15. *Using the symbolic view of the sequence aplet, the Fibonacci sequence can be defined and checked using ten keystrokes.*

The generic symbolic view is based on the choose box engine, which takes over display handling to maintain check marks on the left of the screen and takes over key mapping for dynamic menu changes and editing of expressions. Because of the different requirements for different aplets, the generic symbolic view is implemented as a derived instance of the symbolic view with several data fields and virtual routines to be overridden. The information for a symbolic view includes:

- Combine factor
- Total expression
- Group size
- Single-pick flag
- Softkey menu description
- Edit menu description
- Move focus procedure pointer
- Special initialization procedure pointer
- Edit terminator procedure pointer
- Expression checker
- Poststore procedure pointer.

For example, the sequence aplet combine factor is 3 (three terms define one sequence). The total expression is 30 (up to 10 sequences allowed). The group size is 3 (every three terms will share one check mark). The edit terminator procedure transforms definitions into internal form. The move focus procedure updates menu softkeys with the current sequence name. The expression checker rejects list or matrix expressions and initial terms that depend on the sequence independent variable n.

The symbolic view for the statistics aplet posed a new problem because we decided to show two expressions on one line, one for data and one for frequency, which is not supported by the choose box engine. The statistics symbolic view is implemented by customizing the input form to mimic a scrollable choose box with a check mark and two columns of data.

Setup Views

After expressions are set up, setup views are the natural places to go for setting the parameters for further explorations. The plot setup view is the main setup view, similar to the plot dialog box on the HP 48G, but enhanced with wider fields and a double-page design. The plot setup view, the symbolic setup view, and the numeric setup view are all based on the input form engine.

Plot View

The plot view is the most complicated of all aplet views. Students will spend most of their time here exploring the behavior of curves graphically and interactively.

The HP 38G takes the DRAW command in the HP 48G/GX plot window and improves it by implementing a “smart redraw.” When switching back from other views, the picture, cursor position, and display mode are restored to the same state instantly, except when the defining parameters are changed. Plotting can be stopped and resumed later. When the user tries

to move the cursor beyond the screen boundary, the graph shifts and redraws the scrolled-in portion. Zoom options are put into a choose box with more descriptive names. Instead of taking the current point as the first point, the box zoom prompts the user to select the first point. Tracing is improved and extended to statistics plots. Fig. 16 shows tracing on box and whisker plots.

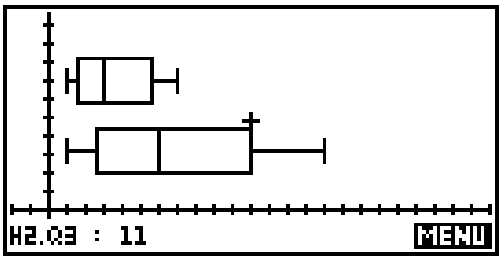


Fig. 16. *Box and whisker plot.*

For the function aplet, more areas of exploration are supported through the FCN menu key, which displays a choose box with choices of root, intersection, slope, area, and extremum. Fig. 17 shows the display for an area computation.

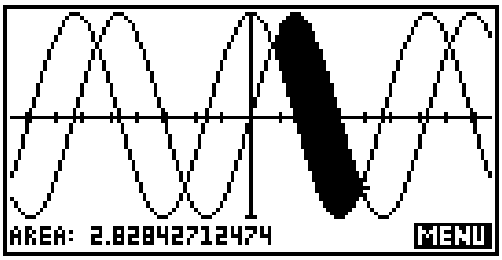


Fig. 17. *Area computation accessible via the FCN softkey.*

The plot view has overridable routines for curve drawing, curve tracing, FCN key handling, DEFN key handing, and other functions. For example, the function, parametric, and polar aplets share the same plot loop with different preprocessing, but the sequence aplet uses another plot loop for handling the discrete independent variable n.

For tracing, the sequence aplet implements four-way scrolling: the screen will scroll when the cursor is moved offscreen in all directions. The scatter plot overrides the FCN menu key to calculate and display a data-fitting curve. The information for a plot view includes:

- Draw procedure
- Independent variable ID
- Softkey menu description
- Display procedure pointer
- Key handler procedure pointer
- Pointer display procedure
- Draw axes flag
- Draw grid flag
- Axis labels
- Display coordinate procedure
- Tracing procedures
- Coordinate display procedures
- Equation display procedure.

Numeric View

The numeric view lets a student explore the functions defined in the symbolic view in numeric form. The leftmost column displays values of the independent variable (except for statistics) and adjacent columns represent function results. There are two basic forms of the numeric view: automatic (Fig. 18) and build-your-own (Fig. 19). The automatic view displays a series of independent values with a starting value and a step specified by either the numeric setup view or the variables NumStart and NumStep.

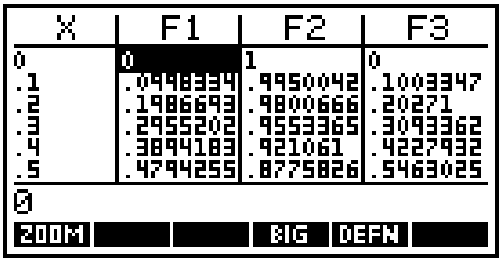


Fig. 18. *Automatic numeric view.*

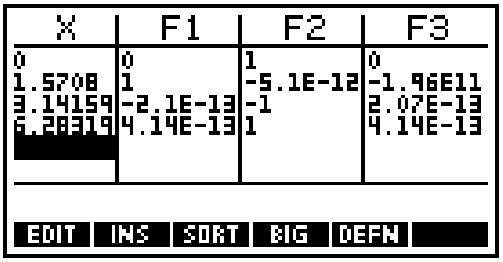


Fig. 19. *Build-your-own numeric view.*

The BIG menu key lets the student display the numbers using a larger font. The ZOOM key provides a series of options for changing the start and step values for the independent variable display.

When the cursor is moved to the upper or lower boundaries of the display the table scrolls to show adjacent values. The table can also be reset by entering a new value for the independent variable in the leftmost column.

The build-your-own numeric view is useful for creating a table of interesting values. The values for the independent variable column in the build-your-own numeric view are accessible via the NumIndep variable.

The information for a numeric view includes:

- Initialization procedure pointer
- Numeric zoom choices
- Softkey menu description
- Display procedure pointer
- Key handler procedure pointer
- Split plot-table configuration information.

Plot-Table View

The split plot-table view allows a student to combine the plot and numeric views (Fig. 20).

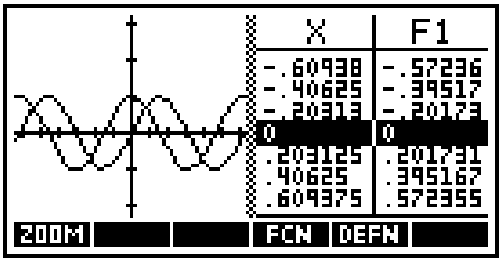


Fig. 20. *Plot-table view.*

This view is implemented primarily as a plot view, with the right side of the display being a small numeric view that updates to reflect the position of the plot cursor. As the student moves the cursor from one function to another, the right side of the table changes to reflect the function being traced. The DEFN menu key displays the current function at the bottom of the display. This lets the student display the symbolic, plot, and numeric views of a function all at once.

Note View

Besides main aplet views like plot view, symbolic view, and so on, auxiliary views like the note view and sketch view are provided to add textual and pictorial descriptions to an aplet. The note view (Fig. 21) can be used to edit and display a text string attached to an aplet. The note can provide information about the aplet's subject, a suggested sequence of exploration, or the supported keys. The note view is basically a word-wrapping text editor with a 6-line-by-22-character display.

Although the original HP 48G/GX edit line supports multiple-line editing, individual lines are handled independently. When more than 22 characters are inserted into a line, that line gets scrolled to the left without affecting the rest of the lines. The HP 38G note view is implemented independently from the edit line code. Direct display routines are coded to show a character or a string at a specified location without generating intermediate GROBs. System-level keyboard handling code is modified to implement a general blinking cursor display scheme. Besides the text string to be edited, the note editor maintains a linewidth array for word-wrapping bookkeeping.

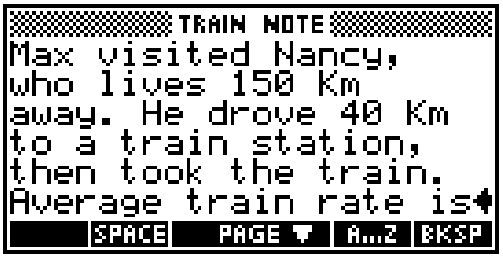


Fig. 21. *Sample note view screen.*

Sketch View

Some people believe that a picture is worth a thousand words, so the HP 38G generalizes the HP 48G/GX's bitmap editing features to form an aplet sketch view. With the sketch view, the user can edit and display a set of bitmaps attached to an aplet. Holding down the page-down or page-up key shows a prestored animation sequence (Fig. 22).

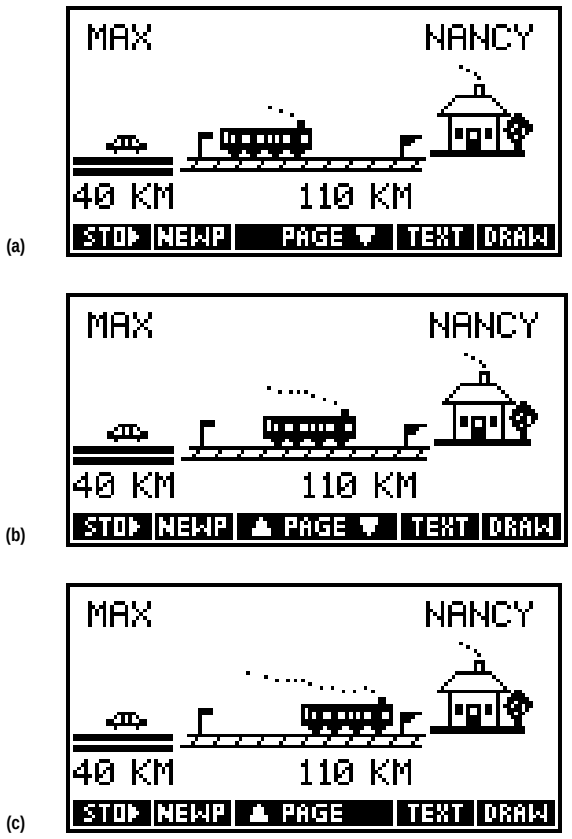


Fig. 22. *An animation sequence in sketch view.*

Compared with the HP 48G/GX, the HP 38G limits the bitmap editing features and improves the user interface and implementation. The HP 38G implements a rubber band algorithm when the user drags the second point to define a line, rectangle, or circle. The circle drawing code uses a fast integer-based iterative algorithm. The user can drag a text string in the small or medium font to any location. The HP 38G can store a selected portion of a screen into a GROB variable and recall it. When bitmaps are stored back into an aplet directory, they are first trimmed to the minimum enclosing rectangle to save RAM space.

Additional Views

Each aplet type defines additional views available in the **VIEWS** menu. For example, the function aplet offers some hybrid views and some preconfigured plot views. Fig. 23 shows the function **VIEWS** menu.

In addition, a user can define a new set of special views that provide high-level tools for an aplet. Such a custom interface makes the aplet easier to explore and hides details of the calculator's operation.

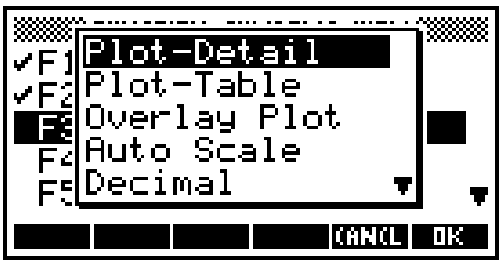


Fig. 23. *Function* **VIEWS** *menu.*

The Home Environment

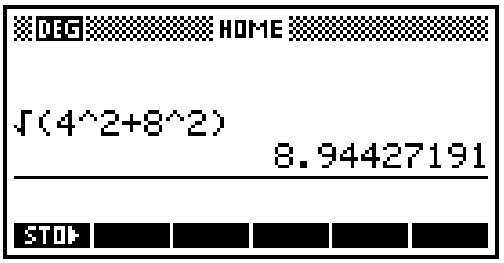
The freeform home environment fills the traditional calculator role of supporting quick calculations. The user enters expressions in algebraic form and the value of the expression (usually a number) is returned.

Unlike aplets, the home environment provides access to all calculator resources, including lists, matrices, graphics objects, and programs.

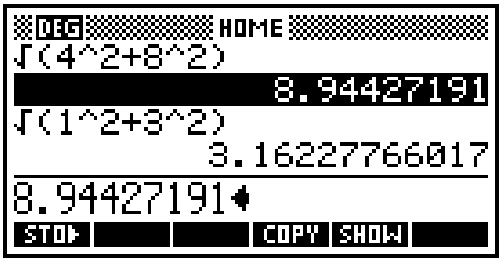
The History Stack

The text of the input and the result are stored on a history stack (Fig. 24a). The user can review the items on the history stack and reuse those items as parts of the current input (Fig. 24b). Expressions and equations on the history stack can be shown in two-dimensional mathematical format (Fig. 24c).

(a)



(b)



(c)

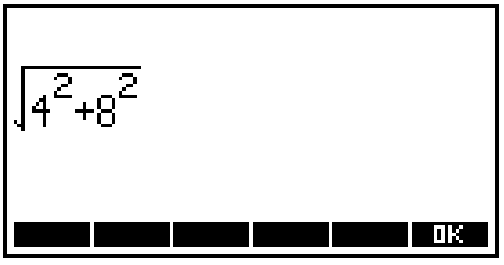


Fig. 24. *(a) History stack. (b) Previous result copied into the command line. (c) EquationWriter display of the same expression.*

The HP 38G includes special features to help beginners. To help students learn about fractions, the HP 38G has fraction number format, which uses continued fractions to convert results to rational form (Fig. 25). To help students unfamiliar with standard algebraic syntax, the HP 38G attempts to interpret ill-formed expressions as implied multiplication (Fig. 26).

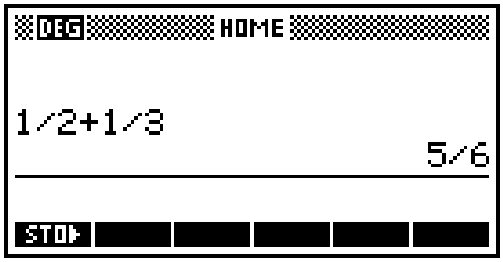
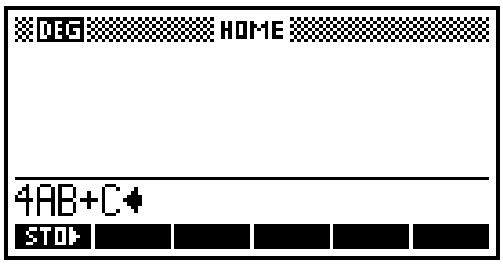


Fig. 25. *Fraction number format.*

(a)



(b)

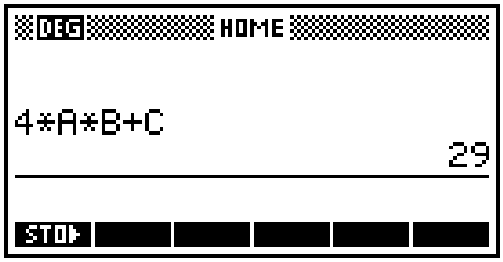


Fig. 26. *(a) Implied multiply input. (b) Result.*

The Variable Ans

Each time the user inputs an expression, the value of the expression is stored in a variable named Ans. This current value of Ans is placed on the history stack, and the name Ans can be used in the next calculation. Even when the user enters a command that performs some action but doesn't return a value, the current value of Ans is placed on the history stack.

If the user starts an input with an infix function such as **+**, **–**, *****, or ***/***, the calculator inserts the name Ans first. This saves keystrokes for tasks such as balancing a checkbook (Fig. 27). If the user presses **ENTER** without input, the previous input is repeated (Fig. 28). This saves keystrokes for iterative operations.

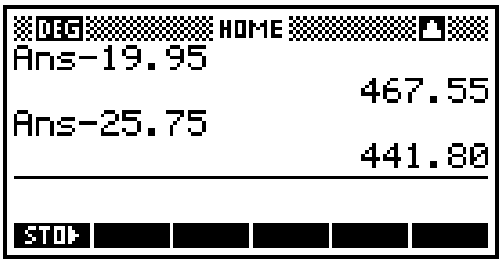


Fig. 27. *Checkbook calculations in the home environment.*

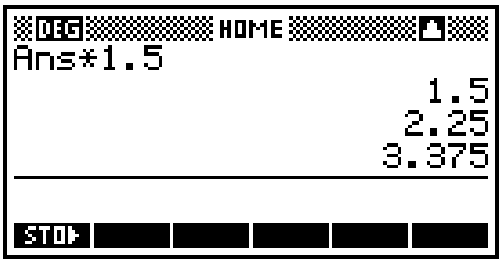


Fig. 28. *If the user presses* ***ENTER*** *without input, the previous input is repeated. This saves keystrokes for iterative operations.*

The VAR and MATH Menus

To organize its extensive resources, the HP 38G presents the most-used variables and functions on the keyboard. Additional resources are available in the **VAR** and **MATH** menus. These two-column menus offer specific items in the right column, categories of items in the left column, and broader choices on menu keys.

In the **VAR** menu (Fig. 29) the user can choose to examine variables either from the current aplet or from the shared home variables. The user also can choose either the name or the value of a variable.

In the HP 38G most variables are strongly typed, that is, for many variables the value must be a specific type. For example, the variable X must contain a real number, Z1 must contain a complex number, and M2 must contain a matrix. Several classes of variables contain exactly ten variables, such as the ten complex variables Z1, Z2, ..., Z9, Z0 (Fig. 30).

Variables are used not only for mathematical objects, but also for modes. For example, storing the constant Degrees (whose numerical value is 1) into the variable Angle selects degrees angle mode.

The classes of home variables include complex numbers (Z1 to Z0), graphics objects (G1 to G0), aplets (user-selected names), lists (L1 to L0), matrices (M1 to M0), modes (fixed descriptive names), notepad (user-selected names), programs (user-selected names), and real numbers (A to Z, θ).

Fig. 29. *VAR menu of the home environment.*

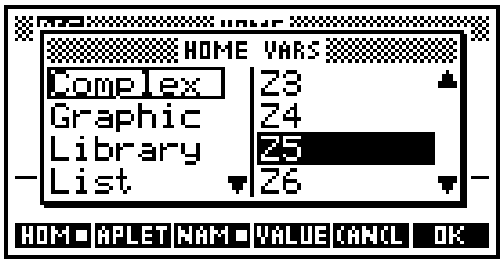


Fig. 30. *Most variable values must be a specific type and several classes of variables contain exactly ten variables. For example, Z1 to Z0 are complex variables.*

The **MATH** menu (Fig. 31) offers additional functions, commands, and constants not available on the keyboard. The categories of functions are: calculus functions, complex-number functions, constants, hyperbolic functions, list functions, loop functions, matrix functions, functions of polynomials, probability functions, real-number functions, statistics functions, functions for symbolic manipulation, tests, and trigonometric functions.

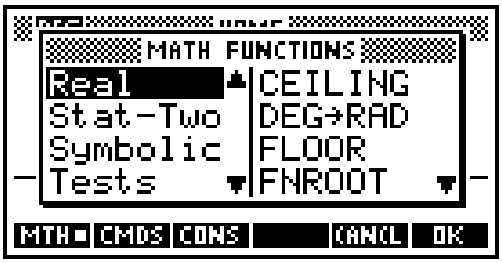


Fig. 31. *MATH menu.*

Lists and Matrices

For these composite variable types there are catalogs that report the variables' sizes, along with special editors to enter and modify the elements. The catalogs and editors are tasks, so the user can easily move among aplets, home, catalogs, and editors. The list editor (Fig. 32) is one-dimensional. The matrix editor is two-dimensional (Fig. 33).

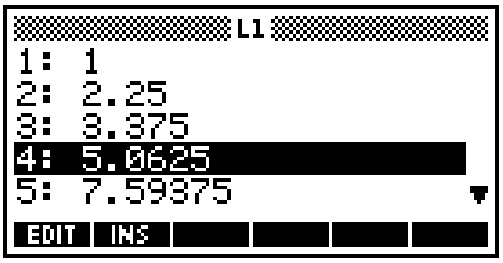


Fig. 32. *The list editor is one-dimensional.*

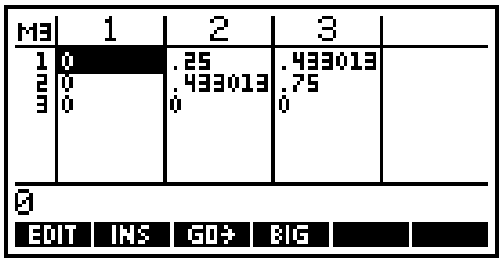


Fig. 33. *The matrix editor is two-dimensional.*

Lists and matrices can also be used as functions of an index value. For example, L1 is a list, while L1(2) is an expression whose value is the second element of L1.

Notes and Programs

For these text variable types there are catalogs that show the user-selected names and editors that allow freeform text input. The notepad holds simple text files such as phone lists (Fig. 34).

There is a program with the fixed name Editline, which holds the most recent input from the home environment. The user can choose to edit the most recent home input from within the program editor, or to execute the program Editline from the home environment by simply pressing **ENTER**.

Programs aren't parsed until the first time they're run. Because of this, and because the program editor is a task, the user can write a program a little at a time, leaving the program in an invalid state between editing sessions. Fig. 35 shows part of a program.

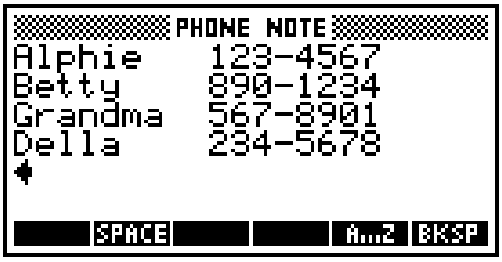


Fig. 34. *The notepad holds simple text files such as phone lists.*

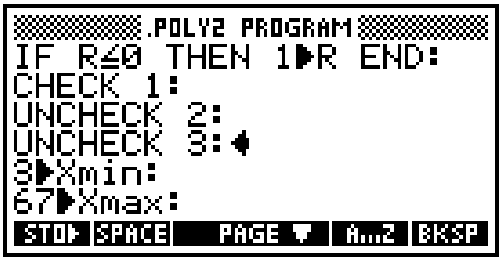


Fig. 35. *Part of a program.*

Commands that perform some action and return no result can appear only within programs (which includes Editline). The categories of commands are: commands to control aplets, branch commands, commands for scaled drawings, commands to manipulate graphics objects, loop commands, matrix commands, printing commands, prompt commands for input and output, and statistics commands.

Programs that Support Aplets

The most important purpose of programs in the HP 38G is to support the user-defined entries within the **VIEWS** menu (Fig. 36). Using the SETVIEWS command, the user specifies a number of special views that represent the tools for manipulating the aplet. These tools can be presented at a high level, using terms relevant to the particular aplet and hiding the actual methods of the calculator from the aplet user.

Fig. 36. ***VIEWS*** *menu with user-specified special views.*

The specification for each special view includes a prompt, which appears in the **VIEWS** menu. It also includes a program, which is run when the special view is selected, and a view specification, which determines the standard view (plot, symbolic, etc.) to be started when the program is done.

If an aplet has special views with start or reset prompts, pressing the menu keys START or RESET within the **LIB** catalog (Fig. 37) automatically selects the corresponding special view.

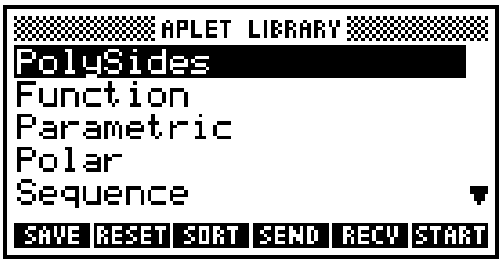


Fig. 37. ***LIB*** *catalog.*

While the prompt category of commands enables programmatic interaction with the user, the main goal of HP 38G programs is to modify the configuration variables of the current aplet. After the program runs, the standard view specified in the **VIEWS** menu starts, operating with the configuration left by the program. This approach has the advantage that the interface to the standard view is familiar to the user.

All the components needed for special views are automatically managed by the HP 38G. The menu of special views is stored as a part of the aplet, and the programs used by the special views are automatically transferred with the aplet.

This makes aplets simple to use: a single request gets the aplet and associated programs, and the **VIEWS** menu offers the high-level tools that allow the user to focus more on the content of the aplet and less on the calculator.

See the ***Sidebar*** for information about how the Distributed Software Team developed the HP 38G Calculator.

Acknowledgments

We would like to acknowledge the rest of the calculator software team: Helen Choy, Gabe Eisenstein, and Charles Patton. Youwen Wu worked closely with us to ensure quality. Diana Roy (learning products) and Kevin Myers and Cary McCallister (technical support) provided many valuable user interface design suggestions. Ron Brooks from our marketing department contributed input from educators. Our education advisory committee provided valuable day-to-day input on our design decisions and usability issues.

References

1. D.K. Byrne, et al, "An Advanced Scientific Graphing Calculator," *Hewlett-Packard Journal*, Vol. 45, no. 4, August 1994, pp. 6-22.
2. T.W. Beers, et al, "HP 48SX Interfaces and Applications," *Hewlett-Packard Journal*, Vol. 42, no. 3, June 1991, pp. 13-21

Distributed Software Team

The HP 38G calculator was the first product to be developed by Hewlett-Packard's Asia Pacific Personal Computer Division after the charter for handheld products was transferred there from Corvallis, Oregon. The software team was split between Oregon and Singapore, so we learned to make good use of communication technology, including Internet tools.

We had to figure out how to overcome a separation of over 9,000 miles and eight time zones. At the end of normal working hours in Oregon, the Singapore workday has just begun. Our specific communication needs included same-time, different-place meetings and documents that could be accessed anywhere, anytime.

Same-Time, Different-Place Meetings

Two of our Oregon team members were already experienced telecommuters, working at home a few days each week. They pioneered the idea of "virtual team meetings." Our first virtual meeting experiments were practiced with everyone in the office, sitting in our individual work areas with telephone headsets and a shared window running on all of our computers, so that we could all view and edit the same document at the same time.

We were able to work out the initial kinks in the process by standing up and shouting to each other if necessary. The next step was to link our two telecommuters from their Oregon homes. When we added our two Singapore engineers, we were already familiar with the procedures and etiquette for same-time, different-place meetings.

The team used HP 9000 workstations as well as computers at home that were connected to the workstations in the office using a LAN connection over 56-kbit/s frame relay lines. The applications used for sharing documents during meetings ran on our workstations and included Collage and HP SharedX. These applications allow a group of people to view and edit the same document simultaneously. However, our more typical way of sharing documents during meetings was by individually accessing our team's WorldWide Web pages.

Anywhere, Anytime Documents

Project documents are used to keep people informed of discussions and decisions and to provide product descriptions for the extended team members. Our challenge was to create and maintain project documents that would be accessible from many different locations. Our documentation process made extensive use of Internet technologies: hypertext documents, graphical browsers, and electronic mail.

Hypertext documents contain links to other documents. They can be easily created with any text editor using a special kind of formatting called Hypertext Markup Language (abbreviated HTML). Our team members were all able to get started with simple hypertext document creation after only a few minutes of study.

It became apparent to us that HTML was the best choice for developing and maintaining our project documentation, because:

- As a text-based source file format, HTML is compatible with our source code control system. This allowed us to maintain HTML documents with the same familiar tools that we used to maintain source code.
- A variety of HTML browsers such as Mosaic and Netscape are commonly available for multiple hardware platforms, which ensured that each team member could access our documentation.
- The hypertext nature of HTML documents provided a natural and extensible means to link together our rapidly growing documentation set.

We started with an HP 38G project home page. It had links to conventional software project documentation, such as our external reference specification (ERS) documents, which were also authored in HTML. But the power of linked documents to disseminate information also led us in new directions that supported the HP 38G project development process.

In past projects, we often regretted our inability to look back at topics discussed through less formal means like email, but with our HP 38G home page and tools to support HTML document generation, we finally had the enabling technologies to overcome this limitation. We began to archive our group email discussions and link them into our HP 38G project home page, indexed by date, author, and subject. On frequent occasions (sometimes as a group during a teleconference) we would refer back to these email archives to revisit a line of reasoning about design choices.

The success of the discussion archival technique led to the side benefit of a uniform, centrally maintained mailing list for software team members. Over time we expanded this concept to include multiple discussion groups, each with associated email archives and each specialized for a different audience.

Soon we were using the HP 38G home page to collect information that was previously scattered throughout paper documents or engineers' minds. For the first time, documents detailing the setup of our software debugger systems, EPROM burners, and more were readily available around the clock.

Other departments involved with the HP 38G began to seek out the resources we made available through the HP 38G home page, and it was quite rewarding for us to be able to direct them to our pages. The frustration of trying to coordinate delivery of information by hand was becoming a distant memory. Some departments, with our encouragement and support, began making information available through linked documents. These we gladly added to our project page, making it more valuable for all parties. For example, when our colleagues

in Singapore had industrial design drawings to share, these were made available to all through the HP 38G home page, and the QA department's software test plans were added so that engineers could review and comment on them to ensure complete test coverage.

Conclusion

Our use of linked HTML documents has increased steadily since the HP 38G project. We now have more email discussion groups, our own Web server to simplify access for remote connections, a team home page with links to all of our project home pages, a dynamic team calendar of events, a "what's new" service for quickly learning what's changed, and a searching service for finding specific topics amongst our wealth of interconnected documents. We expect exciting new developments to further increase our productivity as we expand our knowledge and use of the Web.

Creating HP 38G Aplets

This article explores a simple aplet and shows how to construct an aplet called PolySides.

by James A. Donnelly

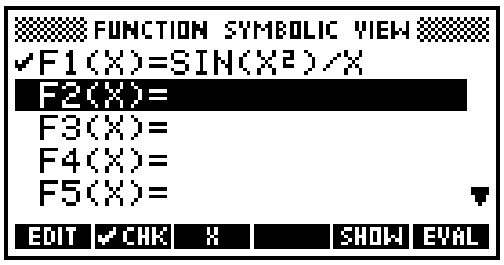
In designing an aplet for the HP 38G calculator, it's important to remember two guiding principles of the HP 38G design. First, the design recognizes that in a classroom setting there is (and should be) a large discrepancy between the amount of information entered into a graphing calculator and the amount of information produced by the calculator. We sought to minimize the calculator input required of the student and the teacher and maximize the returned information. Secondly, we sought to exclude irrelevant possibilities. By reducing irrelevant choices you focus the user's attention on the subject of interest and avoid distractions from things that don't matter. Many choices made during the design of the HP 38G were based on this goal.

Aplets contain information and have views. The information in an aplet consists of every major piece required to produce the views: the equations, setup information, mode information, sketch or text annotations, and attached libraries or programs. In this article we'll explore a simple aplet, then look at an aplet called PolySides and examine how it was constructed.

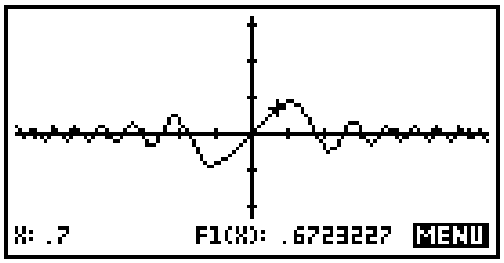
Using Built-in Aplets

When the HP 38G is first turned on, the built-in aplets are empty. There's no equation, note, or sketch. When the user adds this information, these aplets come alive. You can save an aplet at any time, so it's easy to start one project, change in midstream to another, then come back to the first. (Because the appearance of the screens depends on the information you enter, the screens on your calculator may look different from the example screens in this article.)

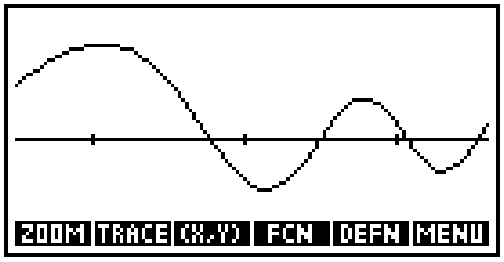
A simple way to illustrate the aplet concept is to explore the equation $\text{SIN}(X^2)/X$. Select the function aplet, press **SYMB**, and enter the equation.



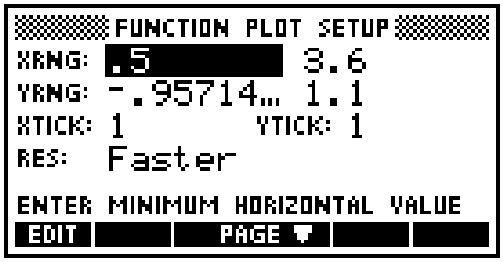
Now press **PLOT** to see the plot using the default plot parameters.



Use the box option under **ZOOM** to look at a smaller area of the plot.



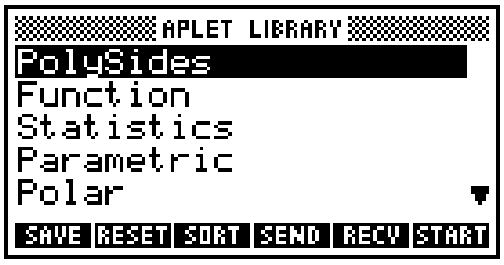
You can see the plot scale by pressing [shift]**PLOT** to see the plot setup view.



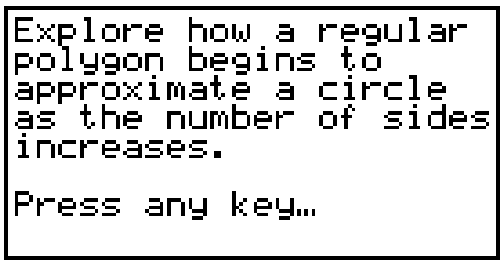
At this point, you have an aplet that's completely dedicated to your interest in the function $SIN(X^2)/X$, and the aplet can be saved under a unique name or transmitted to another HP 38G or a computer. When the aplet is restarted on the original or another HP 38G, all modes and scales are set just the way they were saved.

Exploring the PolySides Aplet

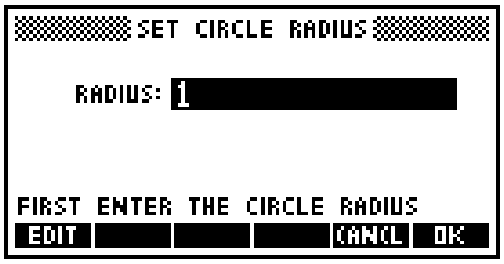
The PolySides aplet is designed to explore how a regular polygon can approximate a circle as the number of sides increases. PolySides can be loaded from a disk or another HP 38G in a single operation from the **LIB** catalog. Once loaded, PolySides appears in the aplet library.



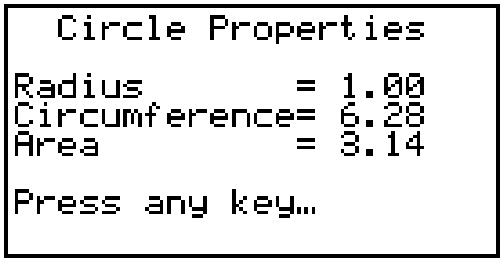
We are now just a single keystroke away from exploring the aplet. To begin the exploration, press START. The motivation for the lesson is displayed first.



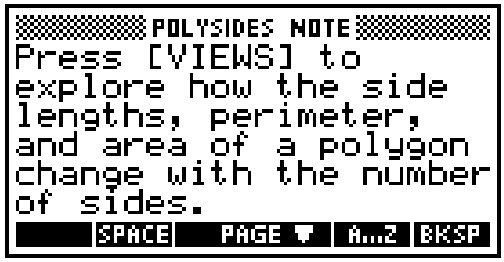
After the introduction has been read and a key pressed, the next step is to enter the radius of the circle to be approximated.

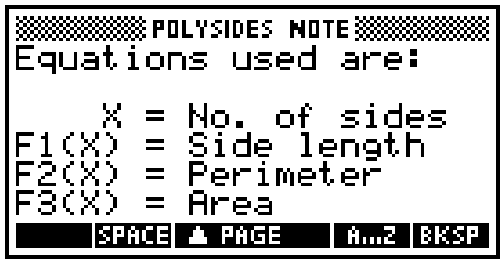


After the radius has been entered, the properties of the circle are displayed.

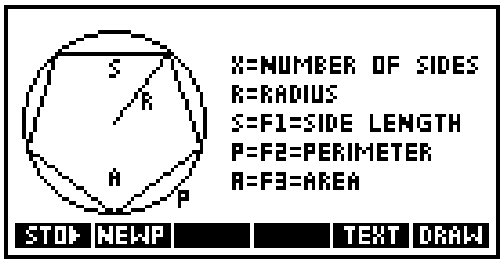


After the circle properties have been viewed, the note view of the aplet is displayed. The PAGE menu key switches between the pages of the note.





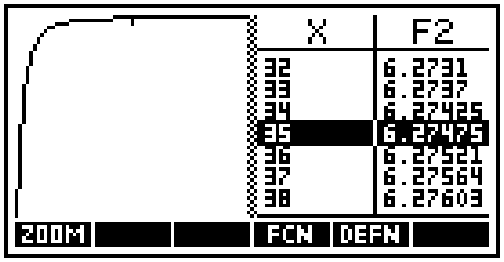
To see a sketch of the problem, press **SKETCH**.



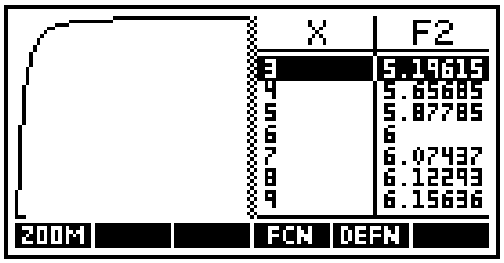
So far we have seen a pretty complete summary of the lesson with less than a dozen keystrokes. Now we can begin to explore how may sides are needed to approximate a circle.

Our central point of departure for PolySides is **VIEWS** (as mentioned in the note view).

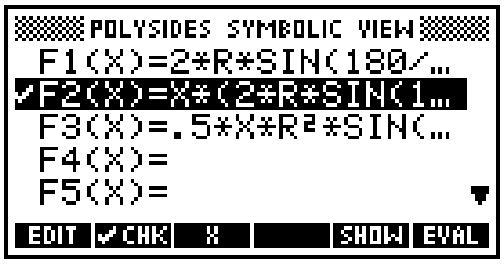
These choices let you determine what aspect of a polygon approximation to a circle you'd like to explore. To explore how the perimeter of the polygon approximates the circle, move the highlight down to Perimeter and press OK. The plot-table view is presented automatically.



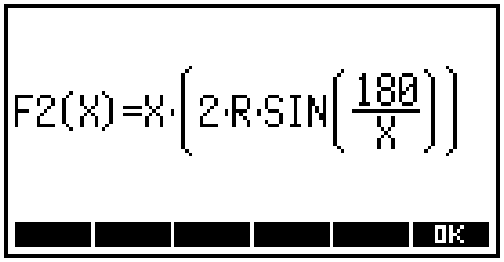
This is one of the HP 38G's split views, showing the plot and numeric views of the perimeter approximation at the same time. The variable X represents the number of sides, and the equation stored in F2 returns the perimeter as a function of X. Pressing the left or right arrow key moves the cursors for the plot and numeric views simultaneously. When the cursor is at the left edge of the plot, the table on the right shows the rapid change of the perimeter from a triangle to a nonagon.



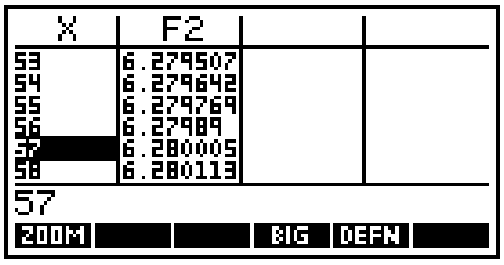
To see the equation, just press **SYMB**.



The symbolic view is where aplet equations are stored. The check mark beside F2 indicates that when a plot or numeric view is displayed, F2 will be used. Another view of the equation is available by pressing SHOW.

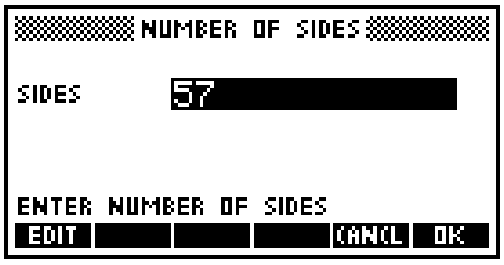


Notice that after the introduction there is no fixed order of events. You can change to any of the views at any time or press **HOME** at any time to do a short calculation. Part of the design philosophy of aplets is to let the student explore the lesson at will, without following any particular algorithm. Once you've explored the approximations to the perimeter and area, and perhaps explored the effect on side lengths, you may decide that a polygon with 57 sides yields a fair approximation to the circumference of a unit circle.

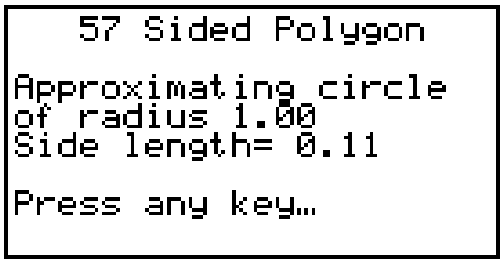


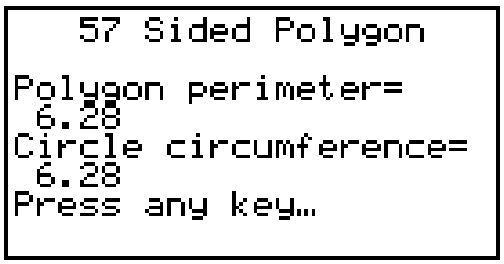
Now you can look at a summary of a 57-sided polygon by selecting Polygon Props (for polygon properties) from the **VIEWS** choices.

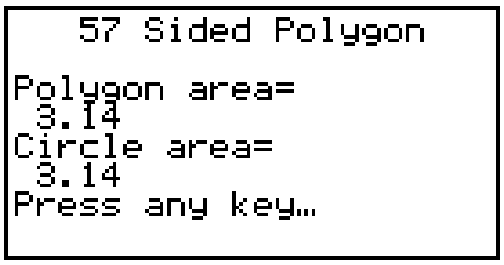
Press OK to select this choice. Since the cursor was last on X = 57, the number of sides defaults to 57.



After the user accepts (or alters) the number of sides, the polygon data is displayed.







Now we've observed that a 57-sided polygon does a fair job of approximating a unit circle. More explorations are possible. For instance, if the circle has a very large radius, how many more sides are required for a polygon to closely approximate the circle's area?

After the last polygon property screen has been acknowledged with a keystroke, the **VIEWS** menu is displayed.

This lets you start with another circle or go back to find a different approximation.

Notice that the keystrokes involved for the entire exploration have been oriented to the views. We haven't entered a single bit of setup or scaling information beyond the radius of the circle being approximated. This is the goal of a well-formed aplet; more attention is directed to the lesson and less attention is directed to the mechanics of the calculator. You can imagine a teacher distributing this aplet in a classroom for part of a day's lesson.

If you're familiar with the HP 48G/GX calculators, you'll notice that variables like EQ and PPAR have never been mentioned, even though we changed the equation and plot scaling parameters several times. Furthermore, we never went near an input form to set the scale manually (although we could have done so by pressing [shift] **PLOT** to display the plot setup view).

Building the PolySides Aplet

PolySides was constructed using the function aplet and five user programs attached with the SETVIEWS command.

Beginning with a reset function aplet, the equations, note, and sketch were entered directly on the calculator. The equations are:

- Side Length **F1(X)=2*R*SIN(180/X)**
- Perimeter **F2(X)=X*2*R*SIN(180/X)**
- Area **F3(X)=.5*X*R²*SIN(360/X)**

Then five programs were entered, one for each entry in **VIEWS**. After the programs were entered, they were attached to the aplet with the SETVIEWS command. For convenience during aplet development, a separate program, SetPolyViews, was entered that contains the SETVIEWS command. SETVIEWS allows you to customize the **VIEWS** key with your own entries and selected entries from the default **VIEWS** choices. SETVIEWS takes triplets of arguments. Each triplet contains the prompt that will appear in the **VIEWS** list, a program name, and a number corresponding to the view that should be displayed after the program is executed. The SETVIEWS command for PolySides looks like this:

```
SETVIEWS
"Start";".Poly1";8;
"Side Lengths";".Poly2";16;
"Perimeter";".Poly3";16;
"Area";".Poly4";16;
"Polygon Props";".Poly5";7;
```

When Side Lengths is selected, the program .Poly2 is executed, then view number 16 is displayed. View number 7 is the **VIEWS** list, view number 8 is the notes view, and view number 16 is the split plot-table view. A SETVIEWS convention is that if a prompt is named Start or Reset, it is associated with the START or RESET menu key in the **LIB** catalog. The PolySides aplet takes advantage of this feature so that pressing START in the **LIB** catalog gets a student underway in a single keystroke.

The programs are listed below. (Note that the translation codes used are the same as translate code 3 on the HP 48G/GX.)

.Poly1 (Start)

```
ERASE:                              Clear the display
DISP 1;“Explore how a regular”:     Display the
DISP 2;“polygon begins to”:         motivation for
DISP 3;“approximate a circle”:      the aplet
DISP 4;”as the number of sides”:
DISP 5;“increases.”:
DISP 7;“Press any key...”:
FREEZE:
IF R\<=0 THEN 1\|>R END:
INPUT R;“SET CIRCLE RADIUS”;        Prompt for the
“ RADIUS:”;                         circle radius
“FIRST ENTER THE CIRCLE RADIUS”;R:
ERASE:                              Clear the display
2\|>Digits: Fixed\|>Format:         Set FIX 2 display mode
DISP 1;“ Circle Properties”:        Display the screen title
DISP 3;“Radius = ” R:               Display each property
DISP 4;“Circumference = ” 2*\pi*R:
DISP 5;“Area = ” \pi*R^2:
DISP 7;“Press any key...”:
FREEZE:                             Wait for a key press
Standard\|>Format:ERASE             Restore standard display format,
                                    clear the display
```

Programs .Poly2 through .Poly4 do similar jobs. Each validates the value for R (the radius), selects the appropriate equations in the symbolic view, calculates the plot scale parameters, and sets the initial values for the numeric view. Note that the plot scale is calculated to fit above the menu. The menu occupies 1/8 the display, so an adjustment of 1/8 the calculated vertical range is made to the vertical scale.

.Poly2 (Side Lengths)

```
IF R\<=0 THEN 1\|>R END:            Ensure a reasonable value for R
CHECK 1:                            Select equation F1
UNCHECK 2:
UNCHECK 3:
3\|>Xmin:                           Set the plot scale
67\|>Xmax:
F1(Xmax)\|>Ymin:
F1(Xmin)\|>Ymax:
Ymin–.125*(Ymax–Ymin)\|>Ymin:       Fit the plot above the menu
3\|>NumStart:                       Set the numeric view to begin at 1
1\|>NumStep:                        and increase in steps of 1
```

.Poly3 (Perimeter)

```
IF R\<=0 THEN 1\|>R END:
UNCHECK 1:
CHECK 2:                            Select equation F2
UNCHECK 3:
3\|>Xmin:
67\|>Xmax:
F2(Xmin)\|>Ymin:
F2(Xmax)\|>Ymax:
Ymin–.125*(Ymax–Ymin)\|>Ymin:
3\|>NumStart:
1\|>NumStep:
```

.Poly4 (Area)

```
IF R\<=0 THEN 1\|>R END:
UNCHECK 1:
UNCHECK 2:
CHECK 3:                          Select equation F3
3\|>Xmin:
67\|>Xmax:
F3(Xmin)\|>Ymin:
F3(Xmax)\|>Ymax:
Ymin-.125*(Ymax-Ymin)\|>Ymin:
3\|>NumStart:
1\|>NumStep:
```

.Poly5 (Polygon Properties)

```
INPUT X;"NUMBER OF SIDES";"SIDES";"ENTER NUMBER
OF SIDES";X:                      Sides prompt
ERASE:                            Clear the display
Standard\|>Format:                Set standard format
DISP 1;" " X " Sided Polygon":    Display the screen title
2\|>Digits: Fixed\|>Format:       Set FIX 2 format
DISP 3;"Approximating circle":    Display circle radius
DISP 4;"of radius " R:            and polygon
DISP 5;"Side length= " F1(X):     side length
DISP 7;"Press any key...":
FREEZE:                           Wait for a key press
ERASE:                            Clear the display
Standard\|>Format:                Set standard format
DISP 1;" " X " Sided Polygon":    Display the screen title
Fixed\|>Format:                   Set FIX 2 format
DISP 3;"Polygon perimeter=":      Display the
DISP 4;" " F2(X):                 perimeter and
DISP 5;"Circle circumference=":   circumference
DISP 6;" " s*\pi*R:
DISP 7;"Press any key...":
FREEZE:                           Wait for a key press
ERASE:                            Clear the display
Standard\|>Format:                Set standard format
DISP 1;" " X " Sided Polygon":    Display the screen title
Fixed\|>Format:                   Set FIX 2 format
DISP 3;"Polygon area=":           Display the area
DISP 4;" " F3(X):
DISP 5;"Circle area=":
DISP 6;" " \pi*R^2:
DISP 7;"Press any key...":
FREEZE:                           Wait for a key press
Standard\|>Format: ERASE          Restore standard display format,
                                  clear the display
```

Building Your Own Aplets

Your own aplets can be as simple as the first example, somewhat more involved (like the PolySides aplet), or very involved, with more complex user programs. Creating a new base aplet that can be downloaded into the HP 38G requires the use of System RPL programming beyond the built-in user programming language.

Basic Steps. The basic steps for building an aplet are:

- Pick an appropriate built-in base aplet—function, polar, parametric, etc.
- In each of the views, enter the information that applies. For instance, you would put your equations in the symbolic view, plot parameters in the plot setup view, and so on.
- Put a sketch of the problem in the sketch view and a description of the problem in the note view.
- For a fancier aplet, use SETVIEWS to attach user programs to the **VIEWS** key. Remember that it may be handy to have a separate program that stores the SETVIEWS command while you're developing the aplet.

Flow of Control. In general, the user of the aplet should determine the flow of control, so that a problem can be explored freely. The PolySides aplet provides a little extra guidance by using the program .Poly1 attached to the start prompt in **VIEWS**. .Poly1 displays an initial screen, prompts for the circle radius, displays the circle's properties, and finally exits to the note view as directed by the SETVIEWS command. In general, the user should be able to navigate freely among the choices in **VIEWS** or the various views, or go home and come back.

Shared Variables. The global variables A to Z, Z1 to Z0, and so on are not stored within an aplet, so if an aplet depends on the value of a global variable it's a good idea to have the program associated with Start in **VIEWS** set these values.

HP PalmVue: A New Healthcare Information Product

The HP PalmVue system integrates personal computer, alphanumeric paging, and palmtop computer technology into an effective solution for delivering timely and high-quality patient data to mobile physicians.

by Edward H. Schmuhl, Allan P. Sherman, and Jon D. Waisnor

The HP PalmVue system (HP M1490A) is a new offering from HP's Medical Products Group that allows transmission of clinical patient information to HP palmtop computers via conventional alphanumeric paging systems. This system integrates several forms of current technology, including computer networks, palmtop computers, paging systems, and devices to deliver a powerful new capability to hospitals and physicians who use HP monitoring systems and cardiographs. This article will provide an overview of the operation of the PalmVue system and show how the system integrates several components to deliver this new service to clinical users.

The User Need

It's Saturday evening, and Dr. Ikeda is enjoying a rare dinner out with his spouse and another couple. As chief cardiologist of the Memorial Medical Center, he does not get much time to relax. This evening, he has largely been able to put out of his mind the unstable condition of Frank Nielsen, a patient under his care in the Memorial cardiac care unit (CCU) who is recovering from a recent heart attack. Suddenly, the waiter taps him on the shoulder and informs him of an important telephone call. He goes to the phone, to be told by Laura in the CCU that Frank has developed a very irregular heart rhythm and she is very concerned about his condition. Since he always carries his portable computer and modem card, he suggests that Laura send him a fax of Frank's ECG so that he can assess the clinical situation himself. He goes to get his computer, and finds his way to a telephone jack that the restaurant has allowed him to use. After several failed attempts, he finally gets the faxed ECG transmitted to his laptop computer. He is able to tell from the fax that Frank is in no immediate danger, but cannot read the ECG well enough to make a clear diagnosis. He tells Laura that he will quickly finish his dinner and stop by the CCU on his way home.

It happens that Dr. Washington, another local cardiologist, is also out with his family celebrating a birthday at the same restaurant. He has several patients in the CCU at St. Francis Hospital. As he is about to enjoy his appetizer, he hears the familiar sound of a paging alert from his pocket computer. He removes his HP palmtop computer from his pocket, waits a moment until the computer displays the HP PalmVue index screen, and presses a key to display the message. It is a message from Tony in the CCU. Dr. Washington's cardiac patient Olga Smetana has been having an increased number of premature ventricular contractions (PVCs), and they seem to him to have changed shape. Tony's message says he needs to know if a change in dosage, or perhaps a new medication, is needed. Dr. Washington pushes another key and views Olga's ECG waveform in a crisp and detailed display on the palmtop screen. He can easily see that the PVCs are not really clinically different from those he has been seeing in Olga's ECG for the past few days. He pulls out his pocket-sized cellular phone, calls Tony in the unit and tells him that everything is OK with Olga, and he will check on her in the morning. He then proceeds to enjoy the rest of his dinner.

HP PalmVue System Description

The HP PalmVue system integrates personal computer, alphanumeric paging, and palmtop computer technology into an effective solution for delivering timely and high-quality patient data to mobile physicians (see Fig. 1). The dispatch station PC handles acquisition of the clinical patient data. It also runs the user application to review and select the data, converts the clinical data to paging messages, and sends these messages to a commercial paging system through a modem connection. The HP PalmVue critical care system acquires patient data (such as patient waveforms and vital signs) from the HP CareNet monitoring network via the SDN interface card. For the HP PalmVue ECGstat application, the patient's 12-lead electrocardiographic signals taken by an HP cardiograph are transferred to the dispatch station PC via flexible disk.

The paging messages travel through the paging system in exactly the same form as a typical "call me at 301-457-8438" paging message, except that each message consists of a string of up to 230 meaningless characters. Most of the major paging providers (radio common carriers) send these pages from their paging system computers (known as switches) to a satellite uplink. The satellite retransmits the messages through a downlink to ground-based paging transmitters in the geographical area covered by the subscriber's paging service arrangement. This coverage territory may be a metropolitan area, a region, or an entire country. The transmitters broadcast the signals, which are then received by the pager.

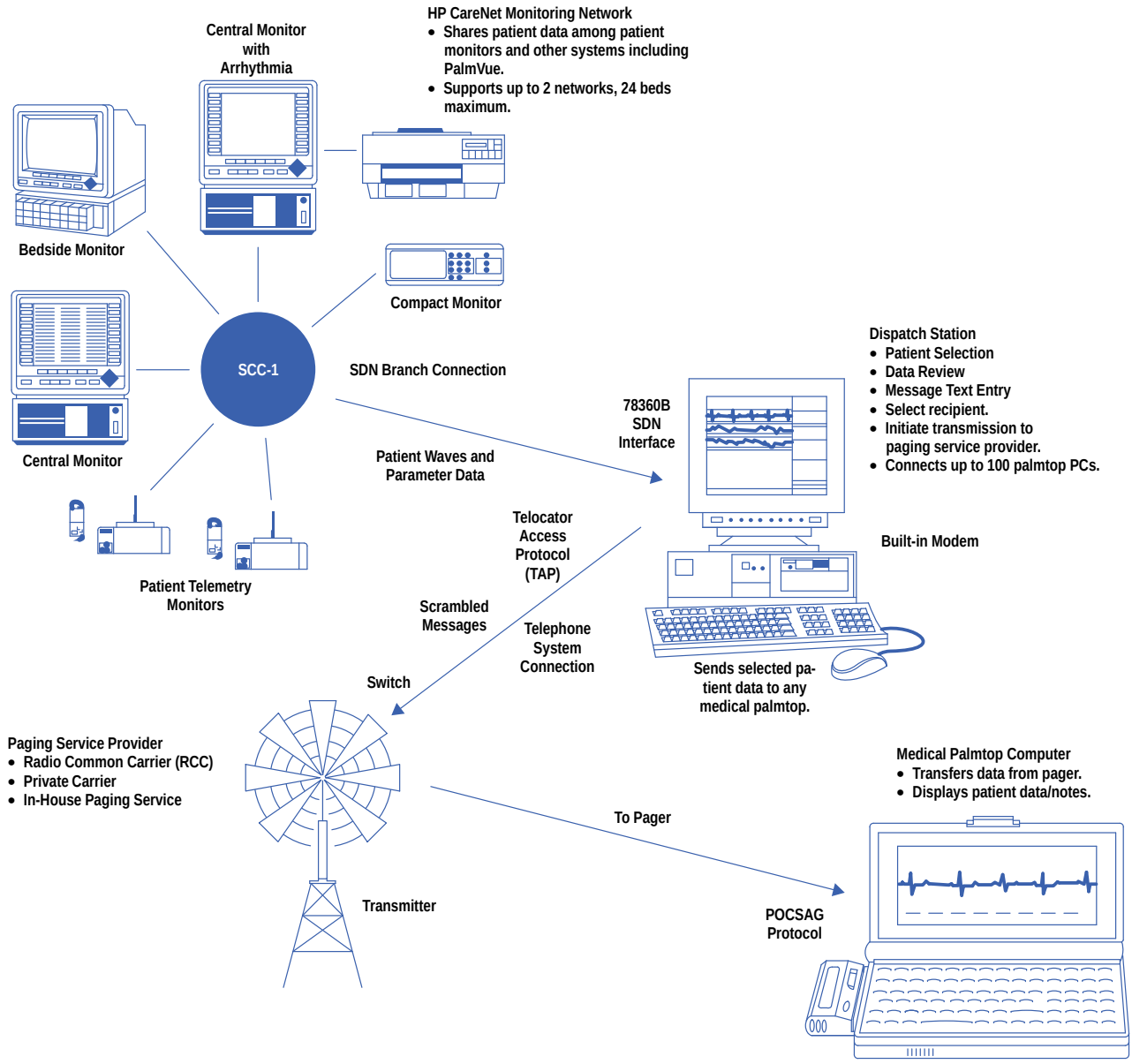


Fig. 1. *HP M1490A PalmVue system.*

The paging receiver used in the HP PalmVue system is a device called a NewsCard, provided by Motorola. This device combines the radio circuits from a standard pager with processing, memory, and a PCMCIA (Personal Computer Memory Card International Association) interface. The individual paging messages are transferred to the HP 200LX palmtop computer by a software program developed by HP for use in the HP StarLink paging service. If the NewsCard is plugged into the palmtop when an HP PalmVue message arrives, the palmtop turns on and the messages are automatically transferred into the memory of the palmtop. If the NewsCard is not plugged in, the messages are transferred later after the user plugs in the card and turns on the palmtop.

The heart of the HP PalmVue system is the software. The program in the dispatch station PC compresses the complex patient data, transforms it into a series of short, character-based messages, and sends them to the paging service. The palmtop software processes the paging messages to reconstruct the patient data, and handles the user interface and display of the patient data on the palmtop screen. Refer to subarticle "***Data Through Paging Technology***" for details of the packetizing and reconstruction process.

The HP PalmVue critical care application provides a snapshot of the current patient data as acquired from the HP patient monitor through the HP CareNet. This data may consist of a 15-second waveform snapshot (typically the ECG, used for assessing the patient's heart rhythm), up to three 5-second snapshots of other physiological waveforms (blood pressure waveforms or other measurements), and the full set of vital signs (heart rate, blood pressures, etc.). The user at the dispatch station selects the patient data, reviews it on the PC screen, and freezes a particular snapshot of information for transmission to the remote physician. The user can also enter a text note to explain particular concerns or provide

additional data to the physician. After choosing the recipient's name, the user initiates the transmission of the message. An example of the user screen on the dispatch station is shown in Fig. 2.

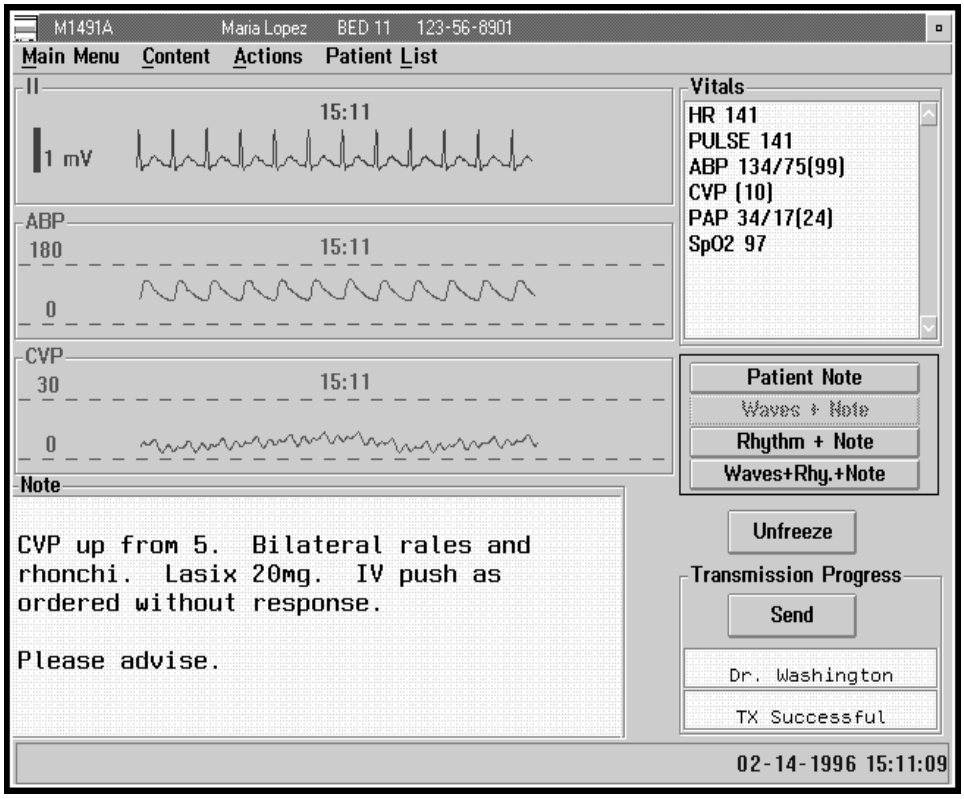


Fig. 2. *An example of the user screen on the dispatch station PC.*

The operation of the HP PalmVue ECGstat application is very similar, except that a previously acquired patient ECG is read in from a flexible disk (or it may have been previously stored in the dispatch station PC).

Operation of the palmtop application is designed to be as simple and intuitive as possible. If the NewsCard is plugged into the palmtop when a new HP PalmVue message arrives, the palmtop turns on, automatically transfers the paging messages into the palmtop, and executes the application program. This program reconstructs the patient data file and shows the index screen on the display with the new message highlighted (see Fig. 3). The user simply presses **Enter**, and the first screen of the new HP PalmVue message appears on the palmtop display (see Fig. 4). This screen provides patient identification, the time the data was taken, and the text note as entered by the sending clinician. The user can access the clinical data portion of the message by using the function keys. For example, pressing **f5** Vitals brings up the patient's vital signs (see Fig. 5). The 15-second ECG waveforms can be accessed by pressing **f6** Rhythm (see Fig. 6). Other portions of the patient data file, or alternate views such as expanded waveform time scales, are similarly accessed by using the function keys. Again, the palmtop application for display of a patient's 12-lead ECG operates in a very similar fashion, with display formats tailored to useful presentation of the 12-lead waveform and diagnostic information.

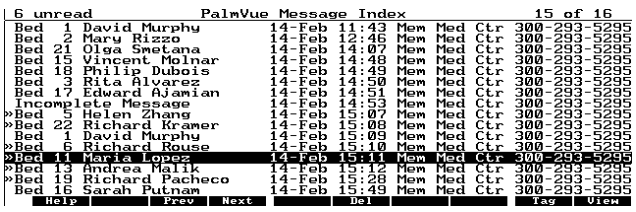


Fig. 3. *Index screen on the palmtop display with the new message highlighted.*

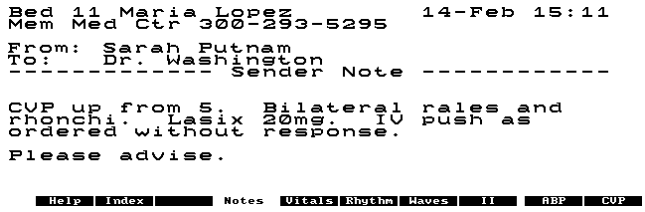


Fig. 4. *When the user presses **Enter** at the index screen, the first screen of the new HP PalmVue message appears on the palmtop display.*

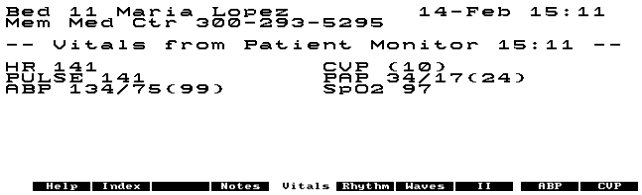


Fig. 5. *Pressing* **f5** *Vitals brings up the patient's vital signs.*

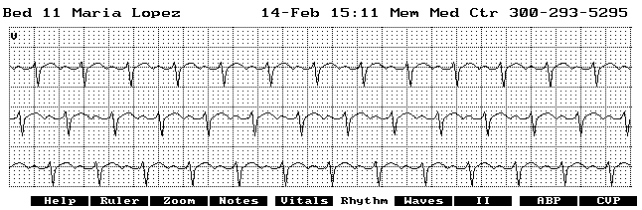


Fig. 6. *The 15-second ECG waveforms can be accessed by pressing* **f6** *Rhythm.*

HP PalmVue Architecture

Several key objectives drove the architectural design of the HP PalmVue system:

- Allow independent development and testing of the dispatch station and palmtop portions of the HP PalmVue software.
- Allocate processing tasks appropriately to the dispatch station and the palmtop.
- Leverage existing software for the critical care application.
- Isolate the transmission process as a subsystem to minimize regulatory concern about the specifics of the paging infrastructure.
- Develop the key software modules in the PC and palmtop to be independent of the paging technology to allow a smooth transition to alternate wireless communication technologies.

The resulting architecture provides a good solution that meets these goals and allowed for a smooth development process.

The HP PalmVue system architecture is shown in Fig. 7. The key to achieving many of the design objectives is the clinical message file. This file is a specially defined representation of the clinical patient data, together with appropriate identification and control information. It exists in the system as a standard DOS binary file. This allows the clinical message files to be created, edited, and transported using conventional PC tools. The clinical message file is provided as input to the transmission process within the dispatch station PC, and is reconstructed within the palmtop software.

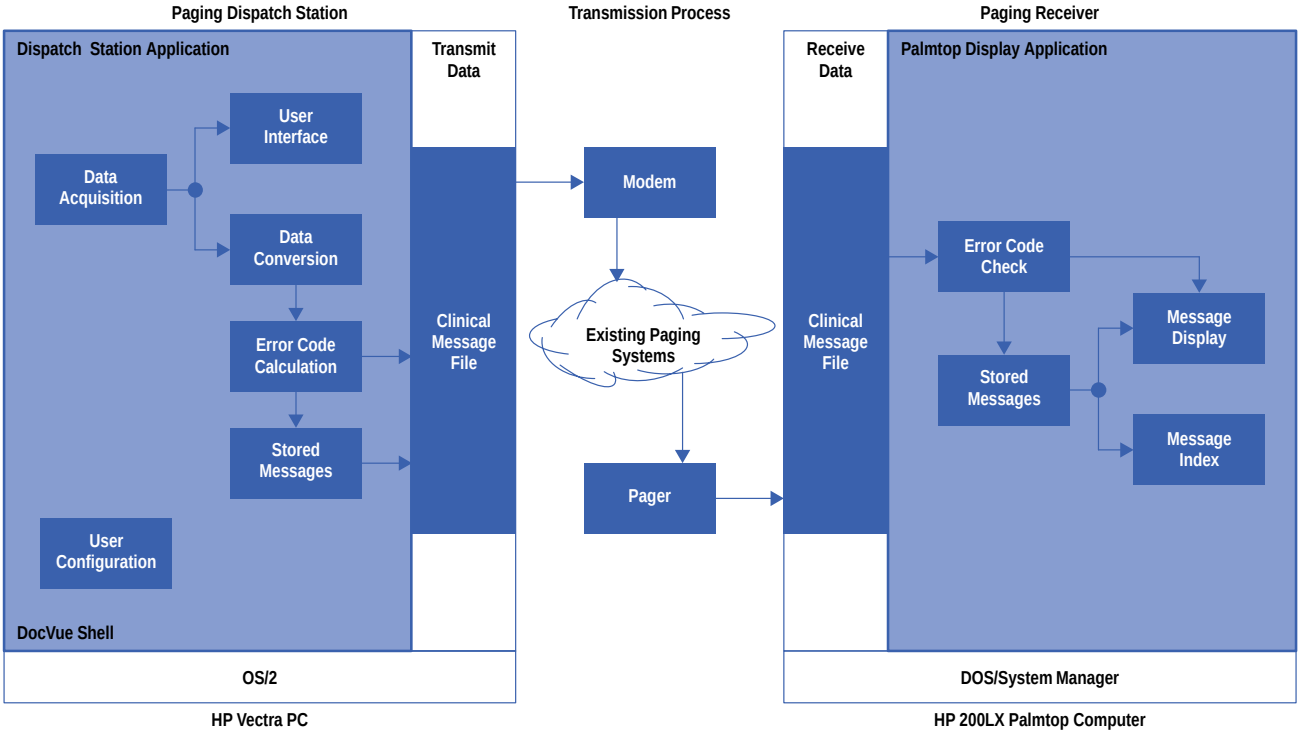


Fig. 7. *HP PalmVue architecture.*

This design greatly facilitates testing. Each application subsystem can be tested and verified independently by either evaluating the clinical message file as output from the PC, or by providing a valid clinical message file as input to the palmtop. The entire software system can be tested without any use of the wireless communication subsystem.

This design also resulted in optimized system performance. The bulk of the processing load, which is the transformation of waveforms into scaled display data, is handled in the dispatch station PC. The scaled waveform data that the palmtop receives in the clinical message file requires very little processing to display on the palmtop screen.

The inclusion of explicit error codes in the clinical message file allows the verification of data integrity with total independence from the transmission subsystem. This allowed the regulatory concerns for data integrity to be addressed without consideration of the specific performance characteristics of the paging systems. Both the clinical message file-based architecture and the error code implementation are capable of transparent migration to other wireless (or wireline) communication methods.

Background of the Product Concept

The availability of Palmtop computers, along with compatible paging devices and services, created the potential for wireless transmission of patient data to these tiny computers. The early product concept included the original Palmtop (the HP 95LX), and a paging device called the NewsStream, which connected to the serial port of the Palmtop using a cradle. The HP 95LX display allowed only coarse waveform presentation, and the combination of the Palmtop, cradle, and NewsStream was somewhat unwieldy. However, prototyping based on these concepts, using the 12-lead ECG, showed the feasibility of the product idea.

Dr. David Albert, a technology oriented physician with previous ties to the HP Diagnostic Cardiology Division, became a champion for commercialization of the product. Having formed a company (Data Critical Corporation, or DCC) to pursue building products based on this technology, he searched for partners to provide access to sources of clinical patient information as well as established marketing channels. He found an enthusiastic sponsor in Jim Cyrier, the division manager of HP's patient monitoring business. Jim had a well-developed vision of the future needs of medical professionals for access to patient data whenever and wherever they were, so he saw great promise in this product concept. Jim set in motion a process that resulted in a contract between HP and Dr. Albert's company for the joint development and marketing of the PalmVue system.

Underlying this product are two major concepts. One is the usefulness of providing complex patient data to physicians on a pocket-sized device, which is enabled by the advent of the HP palmtop, a tiny computer with a high-resolution graphics display. The other is the ability (seen by many in the paging industry as "not possible") to send large, binary information files through the standard and ubiquitous alphanumeric paging systems. Refer to "Data Through Paging Technology" above for the details of the data compression, translation, packetizing and reconstruction process used to implement this capability.

Product Development

The product definition for HP PalmVue was a joint effort of the engineering teams from the HP Patient Monitoring and Diagnostic Cardiology Divisions, working together with Data Critical Corporation. The primary design goals were to provide high quality and useful displays of patient data on the palmtop computer, to provide a very simple and intuitive user interface for the palmtop user, and to ensure the integrity of any patient data presented to the physician. The patient monitoring team developed a prototype of the palmtop displays, using Visual BASIC on a PC. These display screens were transferred to the palmtop, where they provided detailed screen formats and basic user interface controls. This proved to be a very effective method for quickly developing detailed prototype screen formats and control structures for evaluation by the development team and representative clinical users.

The development of the HP PalmVue critical care and ECGstat applications followed separate paths. The critical care product built on existing software products to provide a framework for the dispatch station application and a proven subsystem to acquire patient data from the HP CareNet patient monitoring network. The PC application to provide the HP PalmVue user interface was developed by the engineering team in the HP European Project Engineering Center by leveraging their DocVue product. Data Critical Corporation, under the development provisions of the contract, developed the transmission software for the dispatch station PC and the data reconstruction and user application software for the palmtop. The program management, system integration, testing, and product documentation were performed by the project team at the HP Patient Monitoring Division.

The development path for the HP PalmVue ECGstat product was far simpler. The HP Diagnostic Cardiology Division developed the initial product specification and Data Critical Corporation developed all of the PC and palmtop software. Substantial portions of the software (other than the PC user application) are common between the two products.

The joint development plan for these products made optimal use of the strengths of the partners. The HP patient monitoring and cardiology groups have substantial knowledge of their respective clinical application areas. They also have highly refined and formalized processes for product definition, architectural design, regulatory approval, and software quality assurance. In the case of the critical care product, existing product software from the HP European Project Engineering Center was also a key HP contribution. Data Critical Corporation provided their established Data Through Paging technology, current knowledge of paging systems and paging devices, and specific expertise in programming in the system manager environment of the HP 200LX palmtop. Substantial learning occurred by all of the partners. Data Critical Corporation was forced to conform to the rigorous testing and software QA methods required by HP's regulatory and ISO

9001 processes, while HP embraced the spirit of rapid product creation and aggressive attitudes that are the strengths of a small startup company.

The project teams needed to adapt quickly to evolving technology as the project progressed. While the early prototype was based on the HP 95LX, the HP 100LX was introduced during the early stages of the development work. By the time of product release, the HP 200LX was announced and became the platform for the initial HP PalmVue products. The paging device evolved from the bulky NewsStream to the PCMCIA-based NewsCard. The advent of the NewsCard provides the end user with a compact receiver setup that can be readily carried in a pocket or purse. A team in HP Corvallis and HP Singapore was concurrently developing interface software for the NewsCard, and they modified their software product to make HP PalmVue possible. Adapting to these changes required the development and test teams to be flexible and efficiently rework their software and test procedures to accommodate the evolving HP PalmVue product configuration.

Acknowledgments

Dr. David Albert and Aziz El Idrisi of Data Critical Corporation worked closely with our team on the development and introduction of HP PalmVue. Tim Beevers and Eric Meyers made key contributions to the formulation of early product definition, market exploration, and project planning. Thomas Kerker and Olaf Schnapper of the HP European Project Engineering Center provided the knowledge and technical capabilities for the successful development of the HP PalmVue Critical Care application based on the DocVue platform. Carla Mond, Steve Slaton, and Michelle Maietta provided the market introduction ideas and plans, and orchestrated the highly successful promotional activities. Steve deserves special mention as the originator of the idea for the user scenario at the beginning of the article. Our special thanks go to Jim Cyrier, Patient Monitoring Division general manager, for his visionary role in sponsoring the HP PalmVue project from its inception.

Data Through Paging Technology

The Data Through Paging software, which is licensed to HP by Data Critical Corporation, allows patient information in the form of a binary file (the clinical message file) to be transmitted through a conventional alphanumeric paging system and received on a palmtop computer equipped with a NewsCard paging receiver.

Transmission Data Structure

Two types of pages are generated and transmitted by the dispatch station application. The first type is the file identification packet. This page is sent redundantly as the first and last packets. It identifies the clinical message file and provides the clinical callback phone number and the name of the institution. This page is sent as readable text and is displayed when the clinical message is not received successfully.

The second type of page is the clinical message packet. The clinical message file is divided into clinical message packets using the processing step outlined below. The clinical message packets can be transmitted redundantly if that feature is selected in the dispatch station application.

Clinical Message File Compression

The clinical message file is compressed using a lossless compression algorithm. The approach, which is based on industry-standard concepts and algorithms, is referred to as LVSS. The algorithm employs dictionary-based and Huffman encoding processes. This combined algorithm is similar to the commercially available LHARC program and the widely used PKZIP. The compression is intended to decrease the number of pages sent through the paging service provider by about 50%. As part of the compression process, a 16-bit CRC error detection code is computed and included with the compressed clinical message file. The CRC serves both to check the integrity of the compression process and to detect errors that might have been introduced by the transmission process.

Encoding

The compressed clinical message file, which is in the form of 8-bit binary data, is translated into 7-bit printable ASCII characters. The binary-to-ASCII encoding step is necessary to use the paging system, which was originally intended for the transmission of only printable ASCII characters. The algorithm is similar to the uuencode utility used with many UNIX® email programs. This process tends to introduce an overhead of 30% in the clinical message file size. This encoding overhead is more than offset by the gains obtained through compression.

Packetization

The encoded clinical message file is divided into small blocks of ASCII characters. The size of a block is dictated by the maximum page length allowed by the specific paging service used. Each block contains header

information for later identification of the data block, which allows correct reconstruction of the clinical message file after reception in the palmtop. The header contains a clinical message file filename, a block sequence number, the total number of blocks in the clinical message file, and error detection codes. Each block is then packaged into a page with Telocator Access Protocol (TAP) control characters embedded in preparation for sending to the paging service by modem. Each page also contains information that is used later in the RF transmission stage to target specific receiving pagers.

Data Transmission

A telephone line connection is established between the dispatch station modem and the central paging switch modem. TAP is used to upload the packetized clinical message file to the paging switch. TAP establishes communication handshaking, performs forward-acting error correction using checksum calculation, and retransmits erroneous packets when requested. The uploaded pages are then RF-transmitted to the target pager device (or devices) by the central paging system. The order of transmission of the packets is not necessarily the same as the order of reception by the paging switch.

Receiving Pages

On the receiving end of the RF link, the process is reversed to recreate the original clinical message file. All pages in a clinical message file are stored by the NewsCard paging receiver in its local memory, where they can be downloaded into the receiving palmtop computer. The relevant clinical message file pages are then redirected to a temporary incoming clinical message file page file based on the header information supplied with each page. The last page sent contains a coded message that activates a palmtop macro to start processing.

Clinical Message File Reconstruction

The HP PalmVue application accesses the incoming clinical message file page file and, using the page header information, reorders and reassembles the received pages. The reassembled page message is then translated from 7-bit ASCII to 8-bit binary data to reverse the pretransmission encoding process. The decoded clinical message file page file is then decompressed to reconstruct the original clinical

message file, which is then stored. It is displayed by the palmtop application only if the CRC sent with the clinical message file matches the CRC computed by the palmtop.

Constructing An Application Server

In a dynamic networked environment in which there are several hundred workstations and servers there is a constant demand for new versions of software. In this environment software installation procedures must be quick, flexible, and tolerant of change.

by Jill E. Swenson

With the rapid advancements and constant changes associated with computer technology, new and better hardware and software capabilities can always be expected. Hardware comes and goes as needs fluctuate, and software popularity tends to bloom and fade as fast as software features evolve. All this change makes managing software installation and updates across more than 600 UNIX® systems a very challenging task. This paper describes a project to simplify software administration among a group of workstations at HP's Integrated Circuits Business Division.

At one time our division software was purchased by system owners and installed on individual workstations as requested. Users shared common applications by connecting their workstations in HP-UX* clusters in which one machine's disks were used by many diskless workstations (cnodes).[1] These clusters grew large, and access to software became a more important component in deciding system configuration than performance or networking issues. Some software was shared between cluster servers through NFS mounts, and this led to what we call "spaghetti mounts," which is many machines mounting portions of each others' disks to access shared software (Fig. 1). Additionally, software installations were not tracked, software versions were not synchronized, and network licensing was not effectively implemented.

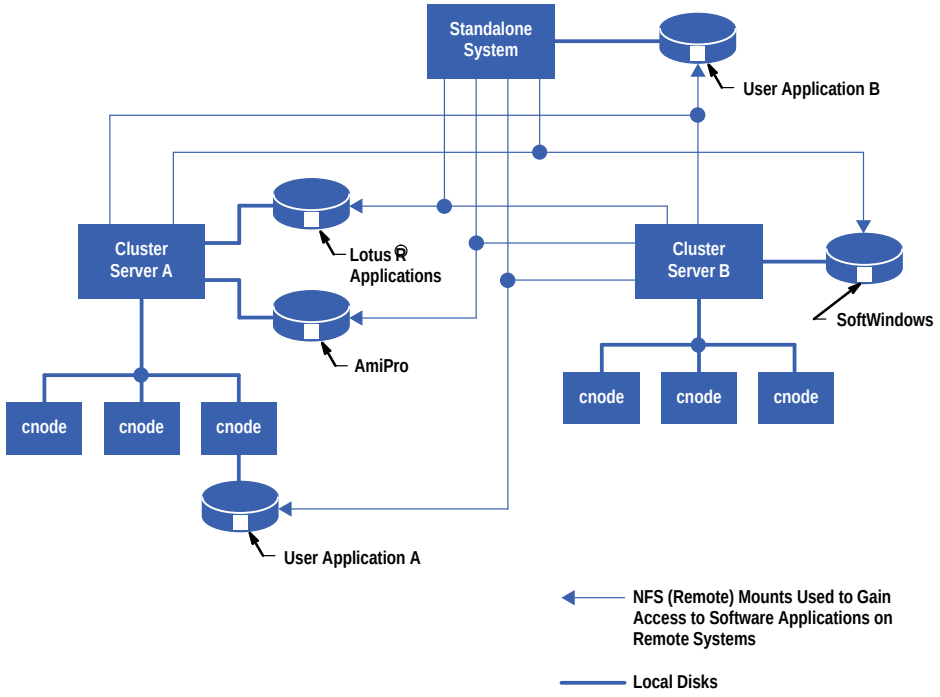


Fig. 1. *An example of "spaghetti mounts," which is many machines mounting portions of each others' disks to access shared software.*

My goal for this project was to find a way to reduce users' dependencies on cluster configurations, to untangle the mounts, and to improve the way we installed, updated, removed, and tracked software. To do this, I moved the application code files to a central server and connected all the user workstations to this server (see Fig. 2). File servers are not an unusual concept, but application servers create unique challenges, and this one needed to be as easy to use as possible.

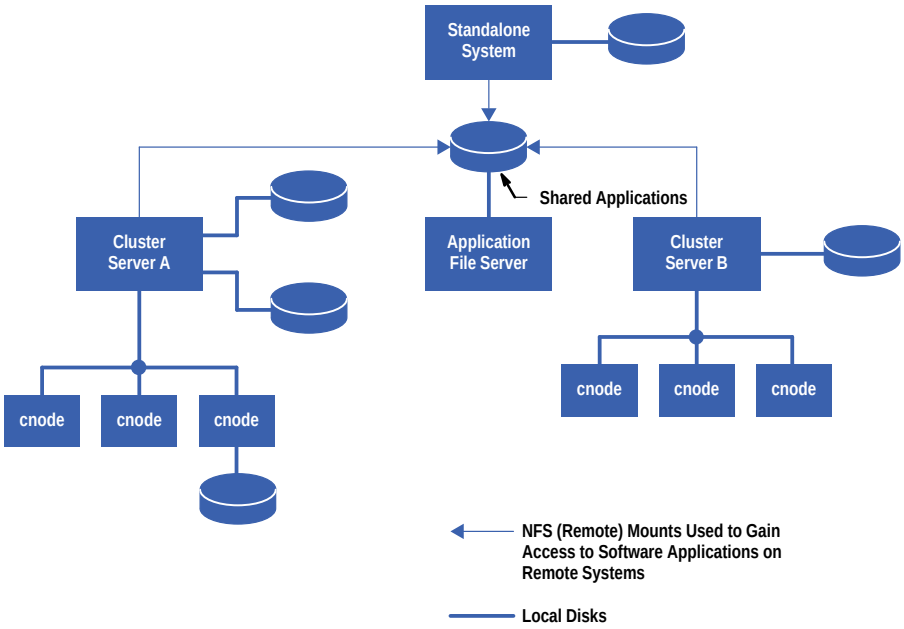


Fig. 2. *A network configuration in which many machines are mounted to an application server to gain access to shared software.*

Developing the Structure

Since we had some spare equipment and disk space, a central software server was a convenient choice for the application server. The basic concept was in place—add disks to the application server, install software on those disks, mount those disks to client machines, and enable the client machines to run the software. Two issues that had to be dealt with were how to mount disks so that symbolic links would be followed correctly on the application server and on the clients, and how to isolate each application and allow it to reside in its desired location on each system. Finally, since saving administrative effort was equally important, the entire process needed to be automated.

A search of the literature produced a helpful article describing how to manage local software on a network.[2] Even more conveniently, the article included the author's installation script. Although not entirely suitable for our environment, this article provided valuable ideas for establishing mount points, isolating applications, and creating configuration files for each application.

Mount Points

To identify mount points, I looked at documentation for the HP-internal UNIX Common Operating Environment (UX-COE) and the HP-UX 10.0 operating system. Both documents recommended using different locations for mounting local and remote (NFS-mounted) disks. This meant that disks physically connected to the application server would be mounted in one location on the server (e.g., /mnt/disk1) and in a different location on the server's clients (e.g., /nfs/servername/ disk1) (see Fig. 3). This approach makes logical sense, but it is limited because there is a route to a file on the server that the client doesn't know about. In other words, the server could refer to a file by the name /mnt/disk1/filename, but that name would not be valid on a client. The client must call that file /nfs/servername/disk1/filename.

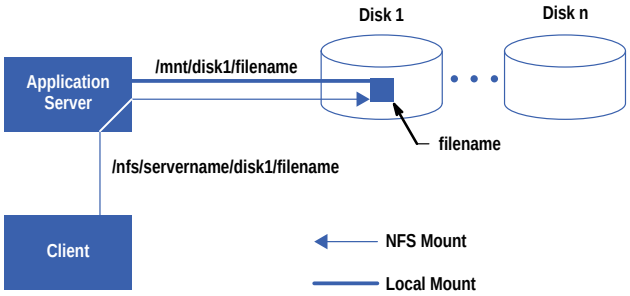


Fig. 3. *An example of using different locations to mount local and remote disks.*

Why Is this a problem?

Calling a file by multiple names typically does not cause problems. However, most application installation programs have a problem with multiple names because they act under the assumption that code is installed on the machine where it will be used and not on an application server. Many installation programs detect the directory into which code is installed and then plug that directory name into scripts, use it to create symbolic links, and so on. The resulting scripts and links work correctly on the application server, but fail on client systems.

To handle this, disks were mounted in the recommended locations and symbolic links were created on the application server to allow it to refer to its own disks under the names used by the clients. Thus, when software is installed it is always redirected, if possible, to the name used by clients.

In retrospect, this was not the best solution. Some particularly difficult installation processes resolve all symbolic links to their absolute paths and use these paths to create scripts and symbolic links. Thus, no matter what name was used for the installation directory, I would find some reference to /mnt/disk1 tucked away in a software configuration file or script. The only way to fool these programs is to either ignore recommended standards and mount the disks to /nfs/servername/disk1 on both the client and the server, or create "fake" symbolic links to the mount points on both machines.

After considering both options, I determined that the first option had the least number of disadvantages and was the simplest to implement.

Software Organization

The next question was how to organize the software on the shared disks. Opinions varied. One approach was to put all code into a common bin directory and set it up so that the administrator owned the bin directory on all machines. Although this would ensure consistency, it wasn't very flexible. Since all users don't have identical software needs (some may be testing a new version of software while others might be installing software packages that are not shared with others), it did not make sense to force all systems to have an identical bin directory.

The approach used in **reference 2** suggested isolating each application in its own directory tree and organizing the components of the application in explicit directories (documentation files under the doc subdirectory, binaries under bin, etc.). This idea of isolation was good, but it required spending time unnecessarily reorganizing the code components.

The software packages we were dealing with came from multiple developers, each of whom had a different idea about where an application's files should reside and the relative location of different components such as library, binary, and configuration files. There was some concern that many of these software packages would not react well to being moved around or being forced to run out of an unexpected directory. The best approach was to install each application into a separate directory on the shared disks, and to use symbolic links to allow the application's files to be logically located wherever they would normally expect to be installed.

To keep this neat, each software package is installed into its own directory as if that directory were the root directory on the system. In other words, if an application called editor came with a file called editor.dict, which it expected to reside in /usr/local/lib, a directory for the new application would be created:

```
/nfs/servername/disk1/editor
```

and the editor.dict(ionary) file would be installed in

```
/nfs/servername/disk1/editor/usr/local/lib
/editor.dict
```

When this software is finally hooked up to client systems, they will each have a symbolic link:

```
/usr/local/lib/editor.dict –>
/nfs/servername/disk1/editor/usr/local/lib
/editor.dict
```

The software can find its data file where it expects, and all the files associated with the editor program can be isolated in one directory tree which can be easily removed, replaced, or modified as needed.

Each application's directory tree mimics the root file system, making it easy to see where the software components will ultimately reside on client systems. Developers like the design because they can easily see the relative relationship of all their files in isolation.

There are two significant issues with this approach. First, it requires that a large number of symbolic links be maintained on multiple systems. Second, some applications might require configuration changes on the client or insist that certain files physically reside on the client's hard disk.

The solution to both of these problems is a configuration file and a process (script) to use it.

Configuration File

Every application on the application server has an associated configuration file. This is an ASCII file, created manually, that contains all the instructions necessary for installing the application on a client. For most applications, the configuration file simply lists a number of symbolic links to create. In the example above, one line in the editor program's configuration file would look like:

```
LINK:/nfs/servername/disk1/editor/usr/local/lib
/editor.dict:/usr/local/lib/editor.dict
```

Each line in the configuration file contains a tag field followed by several colon-separated parameters. The tag field is a word that has meaning to the configuration script. In this example, the LINK tag tells the script to create a symbolic link using the next two fields as arguments. The syntax for the LINK command is:

```
LINK:/link/to/directory:/link/from/directory
```

Additional tags cause the configuration script to copy files (COPY), unlink directories (UNLINK), or execute programs.

Creating a configuration file is the most time-consuming action required when installing an application. It requires a good knowledge of the directory structure on client machines and an understanding of the run-time environment expected by the application. The overall goal is to minimize the number of links while allowing multiple applications to be installed where they wish. However, with a little creativity, nearly any application can be dealt with, and once the configuration file is constructed, it remains unchanged and can be used on every client.

The configuration file is also used to uninstall software. An option on the configuration script causes it to deal with certain commands in reverse, removing links and files that had been copied to the client.

In accordance with HP-UX 10.0 file structure recommendations, the configuration file is created in /etc/opt/appname/config.fs. To make the configuration files readable from client machines, they are placed under each application's directory tree. Thus, the /nfs/servername/disk1/editor/etc/opt/editor/config.fs file would be the configuration file for the editor application mentioned above.

The Script

The configuration script is used to install an application located on a file server on a workstation (i.e., it sets up links to applications on the server). Because the configuration (and application) files reside on a remote server, a new client must mount the remote server's disks before installing software. The installation script establishes mounts (-m option) to the file server and creates the required links (-c option) using information from the configuration file. The installation script has an option (-u) to undo a mount and uninstall software. The command line to the configuration script to establish the mount points and set up the links for the editor application would be:

```
scriptname -m servername:/mnt/disk1:/nfs/
servername/disk1 -f -c /nfs/servername/disk1/editor/etc/opt/editor/config.fs
```

The –m option introduces the server and client involved in establishing the mount points, and the –f and –c options introduce the configuration file.

The configuration script is written in Perl (Practical Extraction Report Language) because this is the language that provides the easiest and most efficient techniques for reading through the configuration file once, storing the information, and acting on it later. Shell scripts require more coding to perform the same steps, and a C program would have to be compiled for every platform we support. Thus, Perl was a good choice. Fig. 4 shows the portions of the installation script responsible for mounting disks and reading a configuration file.

The configuration script reads and acts on instructions in the configuration file. It checks the first field of each line for a tag, which is a special word (e.g., LINK) describing a command. Unknown tags are ignored.

Because the configuration files reside on the server, clients must mount the server's disk before attempting to read any configuration files. Although the configuration script can be used to mount the server's disk, it does not currently modify a client's boot procedures or its /etc/checklist file. The system administrator still needs to perform these steps manually.

The configuration script uses the same configuration file for installing and uninstalling. It does an uninstall by removing all files or directories listed as LINK or COPY in the configuration file.

Remaining Issues

With any project there are always lingering issues. This section describes some of these issues.

Software Location and Fine-Tuning. Although it's convenient to locate software centrally, not everything belongs on a remote machine. Ideally, enough software should remain on local workstations so that the users are reasonably functional when the application server is down for maintenance. This means that key files such as operating system components, screen icons, and critical applications such as electronic mail readers should reside on local systems. Also, performance is impacted by accessing remote files so finding the right mix of local and remote files can have a significant role in application performance.

```
# Options:                                          #
#  -m Mountinfo: Revive/establish mount points      #
#  -c Configfile: Revive/establish links            #
#  -f:Force symlinks and mounts (removes            #
#     existing files and directories).              #
#  -u:Unmount or remove links, depending            #
#     on other options used                         #
#  -v:Verbose                                       #
#                                                   #
require 'getopts.pl';
do Getopts('m:c:fuv'); (A)

# Initialize variables
$numlinks=0;
$numunlinks=0;
.
.
.
################################################
# Read mount information.                      #
################################################
sub readmount {
 print "Starting readmount subroutine \n"
 if ($opt_v);

# The mount parameter consists of the machine
# to mount from, the directory on that machine
# to mount and the directory to mount to (on
# the client machine), all separated by ":"
# characters.
# Break line into its components.
     (B)        (C)        (D)
 ($machine, $mountfrom, $mountto)=split
 (/:/,$mountpoint);
 .
 .
 .
}# End subroutine readmount

################################################
# Perform mounts, creating directories, if     #
# needed.                                      #
################################################
sub makemounts {

 $dontdo=0
 if (substr($mountto,0,1)eq '/')
    {
    print "Making" .$mountto ."\n";
    if ((system("/usr/bin/bdf$mountto|/bin/
    grep $machine:$mountfrom"))==0)
     {
     print "Correct mount point already exists\n";
     $dontdo=1;
     }
    &makedir ("$mountto"); (E)
    print "mounting $machine:$mountfrom\n";
    if ((-d $mountto)&&(!$dontdo))
     {
     system ("/etc/mount -o soft $machine:
     $mountfrom $mountto"); (F)
     }
    }else
    {
    exit 97;
    }
} # End subroutine makemounts

################################################
# Read config file, filling up variables with  #
# contents.                                    #
################################################

sub readconfig {
  print "Starting readconfig subroutine \n"
  if ($opt_v);
  while ($_=<CONFIGFILE>){ (G)

# The configuration file consists of an initial
# tag and colon-separated parameters. Break each
# line into its components

   chop;
   ($tag, @parms)=split(/:/)
   .
   .
   .
    if ($tag eq "LINK") (H)
    {
    $linkto[$numlinks]=$parms[0];  (I)
    $linkfrom[$numlinks]=$parms[1];  (J)
    $numlinks++;
    }
   .
   .
   .
  }# while
} # End of subroutine readconfig

################################################
# Process links: Create all links specified in #
# config file (must run readconfig() first).   #
################################################
sub linkall {

 print "Starting linkall subroutine \n"
 if ($opt_v);

 for ($i=0; $i<($numlinks); $i++)
   {
   $linkfrompath=$linkfrom[$i];
   $linkfrompath=  s'[a-z_0-9.\-]*$";
     # Chop file/dir name off path

# If parent directory path doesn't exist,
# create it.
&makedir("$linkfrompath");

# Check for existing file/link (exit or
# remove it depending on -f option)
if (-e $linkfrom[$i]&&(!$opt_f))
     {
      print "Non-Link exists:" .$linkfrom[$i].
       "Terminating\n";
```

Fig. 4. *Portions of the configuration script for mounting disks, reading configuration files, and creating links.*

```
      exit 98:
      }

    elsif (-e $linkfrom[$i])&&($opt_f))
      {
# Force (-f) section – force the link (remove
# existing file or directory tree.
      if (-f $linkfrom[$i])
         {       ~
         print "Forcing removal of File"
         .$linkfrom[$i]. "\n";
         unlink ($linkfrom[$i];
         }
       elsif (-d $linkfrom[$i])
         {
         print "Forcing removal of directory"
         .$linkfrom[$i]. "\n";
         system ("/bin/rm -rf $linkfrom[$i]");
       }
       .
       .
       .
   }
for ($i=0; $i<($numlinks); $i++)
   {
   symlink ($linkto[$i], $linkfrom[$i]); (K)
   print "Linking from " .$linkfrom[$i].
   "to " .$linkto[$i]. "\n";
   }
}
```

(A) Input String: `-m servername:/mnt/disk1:/nfs/servername/disks -f -c /nfs/servername/disks/editor/etc/opt/editor/config.fs`

(B) Server's Name

(C) Server's Name for Disk1 `mnt/disk`

(D) Client's Mount Point `nfs/servername/disk1`

(E) Make Directory `/nfs/servername/disk1`

(F) Mount `nfs/servername/disk1 to mnt/disk1`

(G) `config.sys`

(H) Tag = LINK

(I) `/nfs/servername/disk1/editor/usr/local/lib/editor.dict`

(J) `/usr/local/lib/editor.dict`

(K) Create Symbolic Link: `/usr/local/lib/editor.dict -> nfs/servername/disk1/editor/usr/local/lib/editor.dict`

Fig. 4. (Continued)

Software Installation Programs and Headaches. Most installation programs expect the code to be installed on the machine from which it is run and don't embrace the concept of remote access. Good installation programs allow you to change the default installation location and log every step they take, and the friendliest programs place all files under a user-specified location. Bad installation programs silently tread in certain system areas by adding user accounts, modifying boot scripts, and changing the system configuration. Bypassing installation programs, tracking down all the changes they made, and rewriting processes to duplicate the installation on client machines is the most time-consuming part of installing new software.

Read-only Environment. Ideally, the application server's disks should be mounted read-only. Unfortunately, some applications require write access to shared files, so we use read-write NFS mounts. However, application directory permissions are kept tight so that users are not able to write to the server.

File Ownership Conflicts. Occasionally, two different application programs will both try to "own" the same file. I first encountered this with two Lotus® applications I was installing: AmiPro/UX and Lotus Notes/UX. Both versions came with an /opt/lotus/bin/lpconfig file and the files were not identical. Normally, the version a client receives depends on the order in which the two software packages are installed. If AmiPro is installed last, a client will run AmiPro's version of the lpconfig file. If Lotus Notes is installed last, the client will run the Notes version.

Inconsistency and Supportability. To ensure that all users, no matter which software package was installed, used the same application versions, some rearranging was necessary. I installed the Lotus applications mentioned above on the application server into their own separate directories. Next, I copied all duplicate files to a third directory, using the version of the files that came with AmiPro. Finally, I modified the configuration files for both AmiPro and Lotus Notes so that clients would link to these duplicate files in the common directory.

This fix allowed the complete AmiPro and Lotus Notes products to be installed and available. If a problem arises with any of the common files, it's easy to change the file in one place and make sure the fix reaches everyone.

Summary

Overall, the application installation process has been a great success. Users are running consistent versions of applications, sharing centrally managed licenses, and no longer choosing to be part of a cluster simply because of the software available on that cluster. Application providers are able to make modifications in one location accessed by everyone. The application server contains no user accounts, runs very few processes, and is a fairly well-behaved I/O-dedicated machine. Unlike a cluster, the server can suffer occasional downtime without bringing down client systems.

We still run clusters, and the application server works in that environment as well. But the clusters are smaller and they are being implemented for the right reasons—not because of application availability. Many users are choosing standalone workstations, where they are allowed greater flexibility to install experimental versions of code or to use as much swap and disk space as they like without impacting other users.

References

1. *Hewlett-Packard Journal*, Vol. 39, no. 5, October 1988, pp. 6-50.
2. J. Lees, "Sharing Local Software on a Network," *The Journal for UNIX System Administrators*, Vol. 3, no. 4, July/August 1994, pp. 48-69.

Interface Translation for Reuse of Assembly-Language Modules in a Two-Language Environment

A mixture of low-level and high-level implementation languages is likely when old modules are reused. In a two-language system, some interfaces must be expressed in both languages. This paper describes the design and implementation of a production-quality software tool that solves this problem for the C programming language.

by James R. Buffenbarger

The advantages of developing a software system in a high-level programming language rather than a low-level assembly language are well-known. Nevertheless, many embedded systems have been and continue to be developed in a low-level language. This implementation language paradox is typically justified in terms of the required execution efficiency or the lack of high-level software development tools.

Typically, a new system is built from a mixture of old and new modules. The old modules are often from a previous system. Depending on their age, they may be implemented in a low-level language. The new modules might be implemented in a low-level or high-level language.

This paper is concerned with the development and reuse of intermodule interfaces in a system requiring a mixture of low-level and high-level implementation languages. Such a mixture is likely when old modules are reused. New modules might also be implemented in a low-level language, but they are more likely to be implemented in a high-level language, since processors are getting faster, memories are getting larger, and high-level software-development tools are becoming available.

In this paper, a problem is identified and defined, several possible approaches are proposed, and the most promising approach is selected and pursued. The design and implementation of a production-quality software tool that solves the problem for the C programming language are described in detail and short-term industrial experiences with this tool are briefly described.

The Problem and Possible Solutions

Suppose a two-language system contains a module named f. Clearly, f should have only one implementation. However, f may need two equivalent interfaces: one for each language. There are several possible approaches:

- Manually develop and maintain a low-level interface only. This approach requires f and every user of its interface to be implemented in the low-level language. However, f or one of its users may be a new module, which suggests a high-level implementation.
- Manually develop and maintain a high-level interface only. This approach requires f and every user of its interface to be implemented in the high-level language. However, f or one of its users may be an old low-level module, which should be reused rather than reimplemented.
- Manually develop and maintain both interfaces. This approach allows f to be implemented in either language. However, the probability of inconsistency is very high, as it is for any instance of double maintenance.
- Develop and maintain the high-level interface manually and derive the low-level interface automatically, according to a well-defined transformation. This approach also allows f to be implemented in either language, but avoids double maintenance. Note that the reverse transformation is not possible.

Interface Translation

The last of these approaches seems to be the most reasonable, and is pursued here. Interface translation is flexible, accurate, and fast.

In many languages, an interface is contained in what is called an *include* file. Thus, deriving a low-level interface from a high-level interface means translating an include file written in a high-level language into an include file written in a low-level language. An include-file translator can be used in at least two ways.

Suppose a programmer is writing a module named g and needs to use a module named f. If the programmer is an assembly-language programmer, module f needs to consist of an assembly-language include file f.inc, which specifies the interface of f, and a binary object file f.obj for linking. On the other hand, if the g programmer is a C-language programmer, module f needs to consist of a C include file f.h, which specifies the interface of f, and a binary object file f.obj for linking. Therefore, the "user view" of module f is the three files: f.inc, f.h, and f.obj. These three files must be available so that both assembly-language and C-language programmers can use module f.

Module f may be written either in assembly language or in C. If it is written in assembly-language, there will be an assembly-language file f.a96 which implements the interface of f and includes f.inc. If f is written in C, there will be a C file f.c which implements the interface of f and includes f.h.

Therefore, if f is written in assembly language, the following files are necessary so that both assembly-language and C programmers can use module f: f.h, f.a96, f.inc, and f.obj. With an include-file translator, only f.h and f.a96 need to be maintained manually; f.inc can be translated from f.h, and f.obj is assembled from f.a96. The processes are shown in Fig. 1.

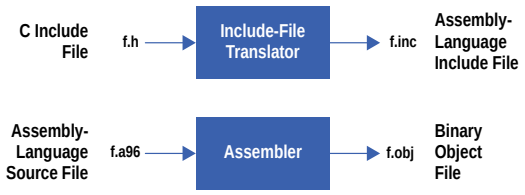


Fig. 1. *For a module f, written in assembly language, to be usable by both assembly-language and C programmers, the four files shown here are needed. With an include-file translator, only the files on the left need to be maintained manually.*

If f is written in C, the following files are necessary: f.h, f.c, f.inc, and f.obj. Again, with an include-file translator, only f.h and f.c need to be maintained manually; f.inc can be translated from f.h, and f.obj is compiled from f.c. The processes are illustrated in Fig. 2.

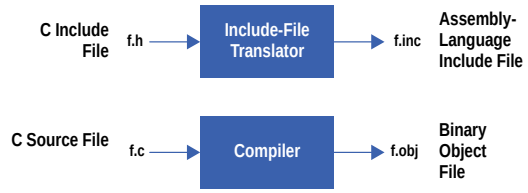


Fig. 2. *For a module f, written in C, to be usable by both assembly-language and C programmers, the four files shown here are needed. With an include-file translator, only the files on the left need to be maintained manually.*

Include-file translation is similar to compilation. In fact, one way of interpreting these ideas is as additional requirements for a compiler. Perhaps compiler/assembler vendors will eventually incorporate these features into their products.

The ideas presented here are programming language independent. However, for demonstration purposes, a particular assembly language and a particular high-level language (C) are discussed. Furthermore, just one implementation of what should be considered a class of translation tools is described.

Pretranslator Environment

We begin with a description of an environment that needs an include-file translator.

The software is developed in an evolutionary way, in a heterogeneous environment. The software is large and mature, residing in an centralized version-controlled repository. The *development platforms* are workstations based on the UNIX®, MS-DOS®, and OS2 operating systems. The *target platform* is an embedded system based on an Intel microcontroller. The software is developed in assembly language because good cross-compilers for the target platform do not exist, compiler-generated code is too slow, and compiler-generated code is too big.

Eventually, the software developers and their managers are ready for a change. New microcontrollers are much faster, and they can address much more memory. Management recognizes the potential benefits of assembler and processor independence, not the least of which is the freedom to buy development tools (e.g., emulators) from more than one vendor. Good cross-compilers are finally available. New developers would become productive more quickly if they did not have to learn an esoteric assembly language. All developers would become more productive if they could program in a high-level language when appropriate.

In this environment, the high-level language of choice is C, but its complete and immediate adoption has well-founded resistance. For speed and space efficiency, some parts of the software should never be rewritten in C. Furthermore, those parts of the software that should be rewritten in C represent an enormous corporate investment. An effort to rewrite all of them at once would have severe short-term productivity and quality costs. Instead, the unanimous understanding is that only some of the assembly-language modules should be replaced by C modules, and then only in a prioritized piecemeal fashion. Thus, the environment must support two-language development.

A Tool for Two-Language Development

Consider the consequences of rewriting an assembly-language module named f in C. The original module has two parts: an assembly-language file named f.inc, specifying the module's interface, and an assembly-language file named f.a96, implementing the module's interface.

Likewise, the new module has two parts: a C file named f.h, specifying the module's interface, and a C file named f.c, implementing the module's interface

However, other assembly language modules still need the assembly-language interface to f. That is, they need to include f.inc in one way or another. One solution is to maintain both f.h and f.inc manually. This solution is tedious and error-prone. A better solution is to maintain f.h manually and automatically translate it into f.inc with a tool. Such a tool needs to be able to translate the subset of C that occurs in a .h file into assembly language.

Two brief definitions are relevant. A *target assembler* is a cross-assembler from a development platform to the target platform. Analogously, a *target compiler* is a C cross-compiler from a development platform to the target platform. Note that the word size of the compiler with which the translator is developed typically differs from that of a target assembler/compiler pair.

The characteristics of C are fairly standardized, stable, and well-known; those of assembly language are not. Thus, the translator needs to be retargetable, primarily for different target assembler/compiler pairs. Several aspects of a particular target assembler/compiler pair are described below, not because they are interesting in themselves, but because they provide insight into the translation task and help explain the example presented later.

The target assembler has several relevant characteristics. It uses a preprocessor, similar to a C preprocessor, to effect file inclusion, macro definition, and conditional inclusion. It processes two kinds of equate directive, EQU and SET, whose difference is irrelevant here. It processes EXTRN directives, which are analogous to C extern declarations. Symbols and values have types such as: NULL, BYTE, WORD, POINTER, and ENTRY. During assembly, the output for each statement goes to one of several segments that are available (e.g., code, data, stack, or register).

The target compiler has only one relevant characteristic. Its algorithm for determining the distance from the beginning of a struct to a member requires alignment analysis of the types of all members of the struct.

Include-File Translation versus Compilation

Although the translator is much like a C-subset compiler, there are significant differences. For example, the translator generates code from a #define directive, but a compiler does not. On the other hand, a compiler generates code from statements included by a #include directive, but the translator does not.

A more interesting difference concerns error handling. A compiler can stop producing output (i.e., object code) as soon as it detects any nonwarning error. In contrast, the translator must recover from most errors, including syntax errors, and produce as much correct output (i.e., assembly code) as it can. In many cases, translation errors are quite acceptable, representing C declarations that are simply inaccessible to assembly-language programmers.

Translation Details

The include-file translator accepts as input a subset of the C programming language. This subset consists of C modules (also known as translation units[1]) that do not define functions or variables. Such modules are conventionally stored in include files, with a .h file name extension. Accordingly, the translator need not recognize the following reserved words: auto do return break else static case for switch continue goto while default if. However, the translator does understand the reserved words near and far as qualifiers for pointer types.

Like a conventional compiler,[2] the translator's operation can be understood as a sequence of phases (see Fig. 3):

1. Prior-inclusion preprocessing
2. Define directive preprocessing
3. C preprocessing
4. Scanning and parsing
5. Code generation.

As usual, phases 4 and 5 are interleaved. The phases are described below.

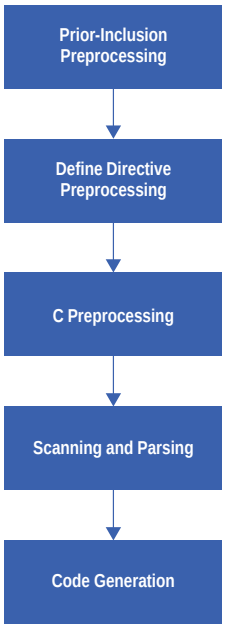


Fig. 3. *Include-file translator operation consists of a series of phases.*

Prior-Inclusion Preprocessor. An include file can reference an identifier declared in another include file, but the former does not necessarily include the latter. The prior-inclusion preprocessor solves this problem.

For example, suppose f.c contains the following:

```
#include "a.h"
#include "f.h"
```

and f.h references an identifier from a.h. To translate f.h, a.h must be processed first. However, declarations from a.h must not produce output.

The prior-inclusion preprocessor solves the problem by constructing a file containing #include directives, #line directives, and the content of the original input file (i.e., f.h). Later phases only produce output for declarations from the original input file.

The prior-inclusion preprocessor is implemented with Awk.[3]

Define Directive Preprocessor. In general, a #define directive in the input produces output. However, C preprocessors delete #define directives. The define directive preprocessor solves this problem.

For example, suppose the following line is line five in file f.h.

```
#define SIZE 100 /* size of something */
```

The define directive preprocessor translates this into the following lines.

```
#define SIZE 100
#line 5 "f.h"
%%% "SIZE" %%% SIZE %%%
```

Since normal C preprocessing has not yet been performed, the original #define directive must be retained.

The #line directive allows error messages from subsequent phases to accurately specify the location of an error. Here, subsequent phases must recognize that a line has been inserted during preprocessing.

The three-character token %%% is effectively a new reserved word. C preprocessing ignores all but the fourth token on this line, which it replaces with the definition of SIZE in the usual way. Phases after C preprocessing recognize this statement and produce output from it corresponding to the original #define directive.

Initially, an @ was used rather than %%% but some C preprocessors complained about it. The probability of a programmer accidently using three adjacent modulus operators seems low. The identifier is used as the fourth token, rather than its definition, to ensure correct operator precedence during evaluation in later phases. If the identifier has no definition, a 1 is used as the fourth token. The line ends with %%% to prevent the parser from skipping too many tokens during error recovery.

The define directive preprocessor is also implemented with Awk.

C Preprocessor. This preprocessor is an ordinary C preprocessor (e.g., cpp). The translator expects the value of an environment variable to specify a particular preprocessor.

Scanner/Parser. This part of the translator is essentially a front end for a normal C compiler. Several differences are:

- Only the subset of C described previously needs to be recognized.
- The new %%% statements are recognized.
- The definition part of a %%% statement is evaluated. Since it can be any C constant integer expression, its evaluation is not trivial. Furthermore, evaluation is simulated to occur according to the word size of the target compiler, which may be different from the word size of the translator, and simulated arithmetic overflow is detected and reported. The alternative to evaluating the expression during translation is to translate it into an equivalent assembly-language expression.
- Most errors, including syntax errors, do not preclude code generation.

The scanner/parser is implemented with Yacc,[4] Lex,[5] and GPerf.[6]

Code Generator. Since the translator only translates declarations, this part is simpler than the back end of a normal C compiler. Its primary task is to output assembly-language directives and declarations, based on the content of a typical symbol table, which is built by the scanner and parser.

To simplify retargeting, target compiler type sizes and type alignment rules are encoded in a tabular fashion. Likewise, target assembler mnemonics and type names are easily modifiable. Code is generated only for declarations from the original input file. Declarations from files it includes produce no output.

Translation Algorithm

The translation from C to assembly language is predominantly syntax-directed. In other words, each syntactic construct in the C subset is translated to a line of assembly language. However, as in a normal compiler, the symbol table provides context. The translated constructs are described in the following paragraphs.

Define Directives. A #define directive, without arguments and whose replacement text is a constant integer expression, is translated to a SET directive. The value is that of the expression, which is evaluated according to the word size of the target compiler. Simulated arithmetic overflow is detected and reported. For example,

```
#define TWO 1+1
```

is translated to:

```
TWO  SET  2:NULL
```

Other kinds of #define directives are not translated.

Enumeration Members. An enum declaration declares at least one member. Each such member is translated to an EQU directive. The value is that which would be assigned to the member by the target compiler. For example,

```
enum  numbers {
    zero,
    one,
    two,
    ten=5*2,
    eleven,
    twelve
};
```

is translated to:

```
zero   EQU  0:NULL
one    EQU  1:NULL
two    EQU  2:NULL
ten    EQU  10:NULL
eleven EQU  11:NULL
twelve EQU  12:NULL
```

Structure and Union Members. A struct or union declaration also declares at least one member. Again, each such member is translated to an EQU directive. The value is the offset, in bytes, from the beginning of the nearest enclosing structure or union, according to the target compiler. For example,

```
struct STR {
    double d;
    double *dp;
    int i;
```

```
    char c,*cp,b;
    float f;
};
```

is translated to:

```
d   EQU   0:NULL
dp  EQU   4:NULL
i   EQU   8:NULL
c   EQU   10:NULL
cp  EQU   12:NULL
b   EQU   16:NULL
f   EQU   18:NULL
```

Notice that alignment occurs. A command line option causes structure and union member names to be qualified by the nearest enclosing structure or union tag (e.g., the translation of the second member would equate the identifier STR_dp to four).

Nested structure and union declarations force an interesting decision. A member's offset can be computed as the distance from the beginning of its (1) nearest enclosing structure or union or (2) outermost structure or union. Both offsets are valuable to an assembly-language programmer. Both offsets require name qualification to differentiate identical member names in distinct structures or unions. However, offset 2 requires a qualification for each level of nesting, which can quickly exhaust the 31-character length limit for C identifiers. In addition, offset 1 allows a programmer to compute offset 2, as necessary. Therefore, offset 1 is considered the best choice.

Structure and Union Tags. In addition to its members, a struct or union declaration optionally declares a tag. Such a tag is translated to an EQU. The value is the size in bytes of the structure or union, according to the target compiler. For example, for the previous structure, the translation is:

```
STR    EQU    22:NULL
```

Variable Declarations. A variable declaration (i.e., a variable definition prefixed by extern) is translated to an EXTRN directive. For example,

```
extern int a[10];
extern int e;
```

is translated to:

```
EXTRN    a:POINTER
EXTRN    e:WORD
```

These directives are output in data segment space. If the C reserved word register is present, they are output in register segment space.

Function Declarations. A function declaration (i.e., a prototype) is also translated to an EXTRN directive. For example,

```
extern int f(int x, int y);
```

is translated to:

```
EXTRN   f:ENTRY
```

These directives are output in code segment space.

An Example

This section provides a concrete demonstration of the include-file translator described in earlier sections. Relevant aspects of the translator's interface are presented and an example translation is shown.

The translator recognizes a variety of command-line arguments and environment variables, but most of them are not very interesting. The important command-line arguments are:

- –i specifies the input file
- –o specifies the output file
- –q simulates prior inclusion of a #include "..." file
- –a simulates prior inclusion of a #include <...> file

The only important environment variable is cppcmd, which allows any C preprocessor to be used, rather than the default. Thus, the translator can be executed as follows:

```
c2as -i cars.h -o cars.inc
```

Suppose cars.h contains the following C code:

```
#define MAKELEN 9
#define CARS 3
typedef enum {black=10,red,blue} Color;
typedef char Make[MAKELEN];
typedef double Price;
typedef struct Car {
    Color color;
    Make make;
    Price price;
    struct Car *oldcars[CARS-1];
} Car;
extern Car *car;
extern Car FixCar(Car car);
```

Then, after translation, cars.inc would contain the following Intel i960 assembly-language code.

```
MAKELEN      SET     9:NULL
CARS       SET     3:NULL
black      EQU     10:NULL
red        EQU     11:NULL
blue       EQU     12:NULL
Car_color    EQU     0:NULL
Car_make   EQU     2:NULL
Car_price    EQU     12:NULL
Car_oldcarsEQU     16:NULL
Car        EQU     20:NULL
           DSEG
           EXTRN   car:POINTER
           CSEG
           EXTRN   FixCar:ENTRY
```

Integration with Build Process

From a build-process perspective, an include-file translator is no different than a compiler: it translates a source file into an intermediate form. Like a compiler, the translator's invocation should be automated by a build process. There are several details, which are described below. A build process based on Make[7] is assumed.

In general, only some of a system's C include files need to be translated. These .h files need to be identified. Typically, a Make variable is used for this purpose.

A single rule is sufficient to tell Make how to translate any of the previously identified .h files into a corresponding .inc assembly-language file. Some versions of Make can use *static pattern rules* for this, while others can use *dynamic prerequisite macros* to accomplish the same thing. If these Make features are unavailable, multiple rules are required.

Any additional dependencies of each resulting .inc file must be specified explicitly, in two ways. First, Make must be told about the dependencies, but this can be done with ordinary rules. Second, such a dependency may require a translator option specifying prior-inclusion preprocessing, as described above. This second kind of dependency cannot be computed automatically. Fortunately, an esoteric Make feature called *computed variable names* can customize translator options for different .h files. Again, if these Make features are unavailable, multiple rules are required.

Conclusion

Include-file translation supports the reuse of modules whose interfaces must be expressed in both a high-level language and an assembly language. These two-view interfaces allow a module to be implemented in either language and referenced by other modules implemented in either language. The basic approach is to maintain the high-level interface manually and automatically derive the low-level interface.

The C include-file translator described above has recently been implemented and incorporated into the software build process used by a group of about 20 firmware engineers at Hewlett-Packard's Disk Memory Division. The translator was implemented for the UNIX and MS-DOS operating systems by one person working half time for about six months. The firmware upon which the translator operates consists of about 275 files occupying over 13 megabytes. About one quarter of these files contain C source code, while the rest contain assembly-language source code. The translator is used on about 30 include files.

For the most part, the experiences and comments of those involved have been positive. Surprisingly, arithmetic-overflow checking caused problems, because the target assembler's word size depends on the complexity of an expression (operator-free expressions can be large). In addition, several users requested an output-file preamble of comments. These comments contain file names, version numbers, and time stamps.

The vast majority of difficulties have been with build process integration rather than translation. Users have had trouble understanding and correctly using the prior-inclusion preprocessor. Typically, a file containing a typedef is omitted, which causes the parser to produce a parse error message. In addition, a .inc file's dependence on its corresponding .h file is not properly recognized by the mechanism responsible for automatically checking out the .h file from the version control system.

Acknowledgments

I wish to acknowledge the contributions made by other members of HP's Disk Memory Division to this project: Michael Banther, especially for his work on the requirements document, Art Beale, Chris Grund, and Steve Folster, for their efforts in the inspections, Ken Hibbs, especially for his thoroughness and patience in the system-test phase, Don Peterson, Dan Martin, Steve King, and Jeff Aguilera, especially for their help in the build process integration phase, Roy Foote and Scott Dehart, my project managers, and of course, the firmware developers who use the tool.

References

1. B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice Hall, 1988.
2. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
3. D. Close, A. Robbins, P. Rubin, and R. Stallman, *The Gawk Manual*, anonymous ftp distribution, 1993.
4. C. Donnelly and R. Stallman, *Bison*, anonymous ftp distribution, 1992.
5. V. Paxson, *Using Flex*, anonymous ftp distribution, 1990.
6. D. Schmidt, *GPerf Manual*, anonymous ftp distribution, 1989.
7. R. Stallman and R. McGrath, *GNU Make: A program for directing recompilation*, anonymous ftp distribution, 1991.