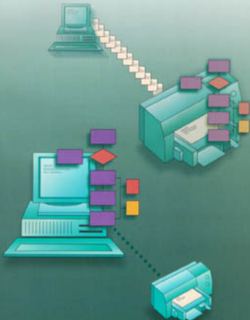
HEWLETT-PACKARD

JOURNAL

June 1997



HEWLETT PACKARD

table of contents

June 1997,
Volume 48, Issue 3

Articles

1 **A Lower-Cost Inkjet Printer Based on a New Printing Performance Architecture**
by David J. Shelley, James Majewski, Mark R. Thackray, and John L. McWilliams

2 **PPA Printer Software Driver Design**
by David M. Hall, Lee W. Jackson, Katrina Heiles, Karen E. Van der Veer, and Thomas J. Halpenny

3 **PPA Printer Firmware Design**
by Erik Kilk

4 **PPA Printer Controller ASIC Development**
by John L. McWilliams, Leann M. MacMillan, Bimal Pathak, and Harlan A. Talley

5 **Next-Generation Inkjet Printhead Drive Electronics**
by Huston W. Rice

6 **The PA 7300LC Microprocessor: A Highly Integrated System on a Chip**
by Terry W. Blanchard and Paul G. Tobin

7 **Functional Design of the PA 7300LC**
by Leith Johnson and Stephen R. Undy

8 **High-Performance Processor Design Guided by System Costs**
by David C. Kubicek, Thomas J. Sullivan, Amitabh Mehra, and John G. McBride

9 **Verifying the Correctness of the PA 7300LC Processor**
by Duncan Weir and Paul G. Tobin

10 **An Entry-Level Server with Multiple Performance Points**
by Lin A. Nease, Kirk M. Bresniker, Charles J. Zacky, Michael J. Greenside, and Alisa Sandoval

11 **A Low-Cost Workstation with Enhanced Performance and I/O Capabilities**
by Scott P. Allan, Bruce P. Bergmann, Ronald P. Dean, Diane Jiang, and Dennis L. Floyd

12 **Testing Safety-Critical Software**
by Evangelos Nikolaropoulos

13 **A High-Level Programming Language for Testing Complex Safety-Critical Systems**
by Andreas Pirrung

14 **An Automated Test Evaluation Tool**
by Jörg Schwering

15 **Effective Testing of Localized Software**
by Evangelos Nikolaropoulos, Jörg Schwering, and Andreas Pirrung

A Lower-Cost Inkjet Printer Based on a New Printing Performance Architecture

The HP DeskJet 820C printer is the first HP inkjet printer in an evolutionary product plan that takes advantage of computer and operating system trends to make inkjet printing affordable for more users. The printer's integrated software, firmware, and digital electronics architecture uses the computational resources in the PC instead of duplicating these resources in the printer.

by David J. Shelley, James T. Majewski, Mark R. Thackray, and John L. McWilliams

The two Hewlett-Packard divisions in Vancouver, Washington are responsible for establishing and maintaining HP color inkjet printers as market leading personal products in the home and office computing environments. These divisions have a ten-year history of successful products starting with the original HP DeskJet printer in 1986 and culminating most recently with the introduction of the new HP DeskJet 820C (Fig. 1) in the spring of 1996.

Fig. 1. *HP DeskJet 820C color inkjet printer.*

Our competitors, of course, have also been introducing products, some of which incorporate newly developed technologies that strongly challenge the performance, print quality, and cost-effectiveness of our own. It is clear that our competitors are here for the long term, so we must develop long-term strategies to compete with them.

Aside from competition, we also have before us an excellent opportunity to broaden our printing solutions to embrace the needs of the entire family, a step well beyond the traditional "take work home" professional who has been our mainstay home customer. These new customers have distinctly different needs that will require insightful understanding as well as timely incorporation of focused innovations in our products.

At the beginning of the HP DeskJet 820C project, it was clear that our ability to retain and grow our market leadership depended heavily upon our ability to deal with these two powerful market dynamics. We knew that we had to simultaneously stay ahead of the competition and satisfy the rapidly increasing breadth of home printing needs. The ingredients for long-term success in this endeavor were equally clear:

- Technologies that result in continuously improving print throughput and quality
- Designs that earn adequate profits at reduced customer prices
- Designs that appeal to home customers by virtue of small size, attractive industrial designs, very quiet operation, and unparalleled ease of use
- Designs capable of high-volume production at multiple international factory sites
- The ability to design products to hit narrow market windows.

We realized that no single product program could successfully satisfy all of these criteria, so we needed to develop a phased approach. We decided that each new product development effort should leverage previous capabilities while incorporating a small set of new and innovative capabilities focused on our customer needs. These new capabilities would then be leveraged forward into succeeding efforts. In this fashion we could ensure a timely series of product introductions, each building upon previous successes and incrementally providing new capabilities that would ultimately satisfy all of our strategic initiatives. In addition to the market timeliness gained by a phased approach, we also knew that this plan would use scarce development resources in the most efficient manner.

Design Objectives

The HP DeskJet 820C printer is the first product in this evolutionary product plan. In keeping with our overall strategy, the primary objectives of the development program were to:

- Leverage the speed and print quality afforded by the new writing system developed for the HP DeskJet 850C
- Leverage the printing mechanism of the HP DeskJet 850C
- Innovate by offering this printing capability at a greatly reduced price for home customers
- Introduce in the spring of 1996.

While reduced cost and a spring 1996 introduction were clearly the primary objectives for the HP DeskJet 820C effort, we also decided to begin our journey towards consumer design by making industrial design changes that fit within the constraints of a leveraged mechanism and package.

Since we had decided to leverage the writing system, print mechanism, and package, we needed to examine the electronic, firmware, and software driver subsystems to find cost reduction opportunities. Based on our initial investigation we set a program goal to reduce direct material cost by 30%.

Design Approaches

Our first design tactic recognized two trends. First, newer generations of personal computers have more than enough bandwidth to take on some of the computing load that has until now resided in the printer itself. Second, software applications are rapidly moving away from MS-DOS® and into the Microsoft Windows® environment. In view of these trends, we decided that the HP DeskJet 820C printer would not support printing from standalone DOS applications. This enabled us to develop an integrated software, firmware, and digital electronics architecture that uses the computational resources in the PC instead of duplicating these resources in the printer. We call the architecture ***Printing Performance Architecture***, or PPA. This architectural choice enabled us to achieve half of our 30% cost reduction goal by reducing RAM from 1M bytes to 128K bytes, ROM from 2M bytes to 64K bytes, and gate count in our largest ASIC by 25%. At the same time, higher-power PCs enabled us to maintain and in many cases improve system throughput.

A second critical design decision was to disallow simultaneous firing of the black and color print cartridges during a single print swath. While this strategy achieved an additional 20% of our overall goal, the obvious risk was a reduction in throughput for documents that contain juxtaposed black and color. However, we felt that our new system architecture would mitigate this risk and still allow us to meet our performance objectives. This single decision allowed us to simplify the drive electronics for the print cartridge to the point where they could be located on a small, carriage-mounted printed circuit assembly rather than on the main logic printed circuit assembly. This change, in turn, enabled two other very significant cost reductions. First, the interface between the logic and carriage printed circuit assemblies was dramatically simplified, allowing the use of standard and easily available cables and connectors rather than the custom designs that we had previously used. Second, using this new partitioning of analog functions, the design team was able to implement the required capability using two custom analog ASICs in contrast to the four that had been used in the DeskJet 850C.

An additional 10% of our cost goal was achieved by capitalizing on three cost saving opportunities in our power supply. First, the initial HP DeskJet 850 power supply was specified with significant margin to allow flexibility for the newly developed writing system in that product. However, the HP DeskJet 820C development team had the advantage of a stable writing system and therefore could specify power needs more precisely. Second, we modified the user interaction model with the printer's power functions and were able to eliminate some of the complex capabilities that were included in the HP DeskJet 850. Third, we specified our power supply at a very high level of abstraction to use the design expertise of our vendor base to deliver cost-optimal implementations.

Several sources contributed to the final 20% of our cost reduction goal. Our new system architecture and new partitioning of analog functionality allowed a significant reduction in the size of our printed circuit assemblies. Direct material cost savings were realized by elimination of the connectors and support components for interconnecting to Apple PCs. Focused design work to cost-optimize our EMI and ESD solutions eliminated many discrete electronic components.

As a result of our plan to leverage and our focus on limited but meaningful innovation, the HP DeskJet 820C was introduced to the market on schedule in the spring of 1996 following a development effort that exceeded objectives by achieving a 33% direct material cost reduction and actual performance nearly twice our initial expectations. The techniques responsible for this success have been carried forward and are already incorporated into the next products in our evolutionary process.

Printing Performance Architecture

The process of printing a document created on a computer involves several steps to transform and prepare the information. In the traditional Windows model used by inkjet printers, the printer driver software receives a description of the page from the application, transforms that description into a mechanism independent format that can be understood by the printer, and encodes it into a standard printer language. The encoded description is then transferred to the printer. The printer decodes the data and formats it for its particular printing mechanism. To encode the information for transfer to the printer, Hewlett-Packard developed a standard language called PCL (Printer Control Language). Because of the widespread use of HP printers, this language has become a de facto standard. PCL allows the computer to prepare an image for printing without detailed knowledge of the mechanical details of the printer.

For the Microsoft Windows environment, HP has always developed the software drivers for its inkjet printers. In the Windows model, the application sends a page description to the driver through the operating system. The description is in the form of drawing objects (lines, rectangles, text, etc.). The driver then rasterizes the description. Rasterization is the process of mapping the page description to an X-Y plane or bitmap. At this point, the data still must undergo several more transformations before it can be used to print. For example, the first bitmap may be 24-bit data at 300 dpi, whereas the inkjet mechanism may be 600 dpi and only able to put one of four colors at each pixel (black, cyan, magenta, yellow). Traditionally, the driver performed some of the needed transformations, but left many of the more compute-intensive ones to dedicated hardware and firmware in the printer.

After the driver has performed all of its computations, it encodes the information using the subset of PCL needed for bitmapped data. The printer in turn decodes the PCL and performs all of the necessary further computations to format the data for the printing mechanism. Manipulations the printer must perform include, but are not limited to, some color transformations, cutting the data into individual swaths, and separating the data into columns (inkjet cartridges are composed of two columns of staggered nozzles). This process is diagrammed in Fig. 2.

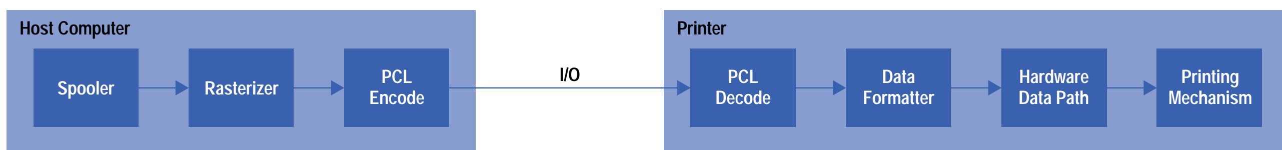


Fig. 2. *Traditional PCL printing model.*

The process in the MS-DOS environment is similar with two exceptions. One, the application must perform the rasterization for all graphics and PCL encoding, and two, the printer will accept nonrasterized text, alleviating the need for the application to do it. Because of this second difference, previous inkjet printers were required to have extensive memory-intensive fonts built into them. In addition, the printer had to contain firmware and hardware to rasterize the fonts. Since the data manipulations performed were extensive, they required a powerful microprocessor and significant amounts of dedicated hardware.

The concept of the new Printing Performance Architecture, or PPA, is to change this model by eliminating some of the steps. Because modern personal computers have powerful microprocessors and a large amount of system memory, the task of data formatting for the print mechanism is moved entirely to the host computer. Also, because the data is no longer in a PCL-compatible format, PCL is not used to transmit the data to the printer. Instead, a very simple proprietary protocol was developed. The protocol is simple enough that the hardware can automatically depacketize the data without help from the firmware. The data is then directly used to print the image on the page. This process is diagrammed in Fig. 3.

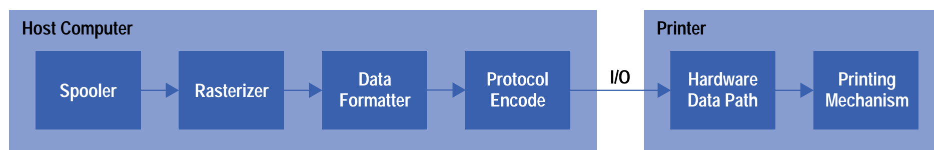


Fig. 3. *New Printing Performance Architecture (PPA) printing model.*

Advantages of PPA

The primary advantages of PPA are cost and performance. A PPA printer can deliver performance similar to a traditional non-PPA printer at a reduced cost. Alternatively, it can deliver higher levels of performance at a similar cost. The reasons for the cost advantage fall into two areas: less memory is required (both RAM and ROM), and a lower-performance microprocessor can be used in the printer because the microprocessor doesn't have to touch the data.

Memory costs are a significant portion of the material cost of a low-end printer. A PPA printer requires significantly less ROM and RAM. First, the PPA printer doesn't have to store any internal fonts. Traditional printers supported both the

Windows environment and the DOS environment. The printing model in the DOS environment requires the printer to store font information. A DOS application sends an ASCII code for the desired text character. To print that character, the printer needs a bitmap for that character in its ROM. In contrast, applications in the Windows environment send only bitmapped graphic information to the printer, never ASCII text. Because the PPA printer is designed exclusively for the Windows environment, it doesn't need to store the fonts in ROM.

Second, because the printer doesn't do any PCL decoding, swath cutting, or data formatting, the printer requires much less firmware, again saving ROM. The primary functions of the printer firmware are mechanism control, input/output, and the user interface. In the HP DeskJet 820C, the firmware is stored in only 64K bytes of ROM. Because there is so little, it was possible to integrate the ROM into the digital ASIC. Previous non-PPA printers of similar capability used 512K bytes or more of ROM.

Finally, because the processor doesn't touch the data and doesn't need to create any intermediate forms of the image data, the printer requires less RAM. The HP DeskJet 820C uses a 128K-byte DRAM. The previous generation, non-PPA printer used 512K to 1M bytes of RAM. Because there are fewer memory ICs, the memory cost for a PPA printer is much lower. The reduced number of memory devices also reduces the printed circuit board area, again saving cost.

The second factor in saving cost comes from the need for less microprocessor horsepower. In a PPA printer, the processor does not do swath cutting and formatting of the data. Its primary functions are mechanism control, input/output, and the user interface. This requires a less complex and consequently less expensive microprocessor. The HP DeskJet 820C uses a Motorola 68EC000. The 68EC000 can be configured with either an 8-bit or a 16-bit data bus. In the HP DeskJet 820C, the processor is used in 8-bit mode. This reduces the bus width in the digital custom ASIC, again saving area and hence cost.

Finally, because of the simplified data path in the printer (the data path is the path the data takes from the input/output port, through the ASIC, and out to the print cartridge), it was possible in the HP DeskJet 820C to design a data path in which the processor doesn't touch the image data. A dedicated hardware data path is always much faster, albeit less flexible, than a data path in which the processor must transform or handle the data. A full hardware data path is not limited to a PPA architecture, but is much easier to accomplish in a PPA printer because of the simplified data path.

Challenges of PPA

While PPA has some significant advantages, it also brings with it some challenges:

- DOS is supported only through Windows and not in a standalone environment.
- PPA hosts must be more powerful than hosts for an equivalent non-PPA printer.
- The printer driver requires detailed knowledge of the printing mechanism.
- PPA required a change in the development and manufacturing paradigm at the HP Vancouver Division.

The PPA architecture does not support printing in the traditional standalone DOS environment. In the Windows environment, all information sent to the printer is bitmapped graphics. The data is prepared under the control of a single, HP-designed and optimized printer driver. In the older DOS environment, application vendors write their own printer drivers. Applications send ASCII codes to the printer and expect the printer to use its own internal fonts to generate the bitmapped characters. The applications have no knowledge of the printing mechanism and hence are unable to do any swath cutting or data formatting.

The HP DeskJet 820C does support printing from a DOS application if the application is run under the Windows environment. Windows allows DOS-only applications to be run in a ***DOS box***. Printing in this environment uses the standard Windows printing mechanism and hence the HP driver.

PPA printers require a higher-powered host than non-PPA printers to achieve comparable levels of performance. Because the job of swath cutting and data formatting is now done by the PC, more computing power is required. On the HP DeskJet 820C, acceptable levels of performance are achieved with a 66-MHz Intel486-based machine with 8M bytes of RAM.

PPA required a shift in the HP Vancouver Division's development and manufacturing paradigm. Having designed and built PCL-based printers for over 15 years, all of the division's tools and processes were centered around this type of printer. For instance, over the years the manufacturing and customer assurance organizations had developed many tools based around PCL printers for doing production tests and exercising the printer in environmental tests. None of these tools work with a PPA printer. Similarly, the firmware test organization had to revise its tests completely. Because the HP DeskJet 820C printer has only 64K bytes of ROM, extensive demo pages and self-test pages could no longer be included in the printer.

Because of the high level of integration and because the architecture follows the paradigm that "the processor doesn't touch the dots," it is difficult to observe the flow of data through the machine. This made debugging problems during development quite challenging. This problem was solved in several steps. First, the ASIC design team did extensive simulations. Second, the team used a hardware emulator to emulate the digital ASIC. This emulator had a mechanism that provided ports to internal nodes so that they could be observed with a logic analyzer. Finally, simple patterns were devised and sent through the architecture that simplified problems and made debugging possible.

Finally, in the PPA environment, the driver must have knowledge of the printing hardware. This makes the driver less universal and the job of leveraging the driver to future products more difficult. The driver was carefully organized and

modularized so the hardware dependent pieces can be changed while the underlying driver features can be leveraged into future products.

Inside the Printer

Inkjet printing is a complicated process that involves tying together several electromechanical subsystems that work together to create the printed page. All inkjet printers consist of these major subsystems regardless of the particular implementation used for each one. Fig. 4 shows the HP DeskJet 820C printer with its top cover removed and the major subsystems labeled.

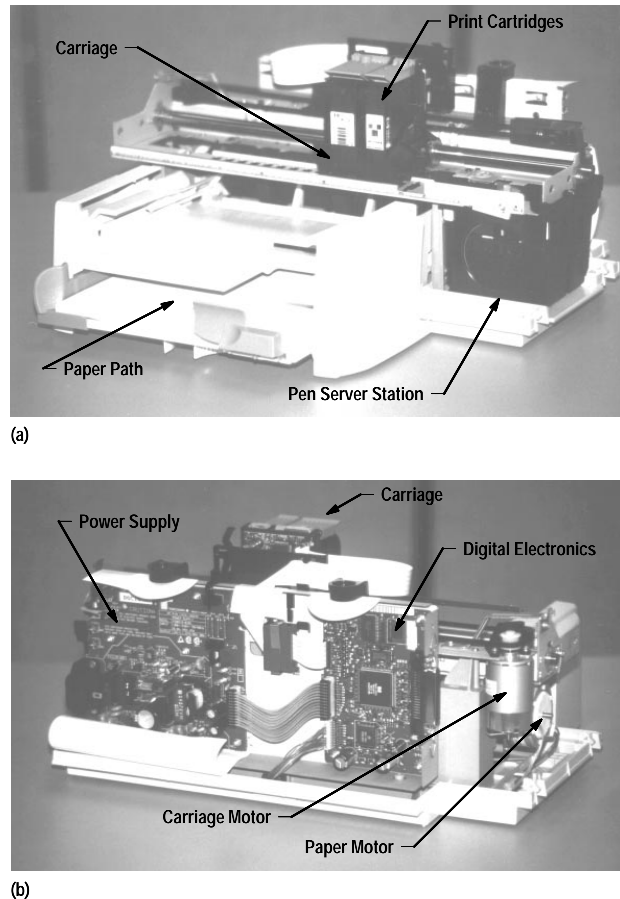


Fig. 4. *HP DeskJet 820C printer subsystems.*

Paper Path. The paper path is responsible for moving paper through the printer. The user inserts paper or envelopes into the input tray. At the appropriate time, a single piece of paper is picked from the stack and begins moving through the printer. Each time the carriage finishes a pass over the paper, the paper is advanced an appropriate amount to prepare for the next pass of the carriage. At the end of a page, the paper is "kicked," or deposited in the output tray, where the user can remove it. In the HP DeskJet 820C, a single electric motor is used to move the paper. Paper movement is open-loop—there is no feedback about the actual paper position. The paper path in the HP DeskJet 820C gracefully handles a variety of paper sizes and thicknesses as well as envelopes.

Carriage. The carriage holds the pens used in the printer. To print a swath of data, the printer moves the carriage across the page at a constant speed, firing the pens at appropriate times. A single motor is used to move the carriage. Carriage movement is a closed-loop process. The carriage's position is tracked using an LED, which shines on a photoreceptor and a strip of plastic made up of alternating light and dark regions placed between the LED and the photoreceptor. As the carriage moves across the page, logic recognizes when the LED is in front of a dark region and when it is in front of a transparent region. Using this information, it tracks the carriage's position on the page. In addition to holding the pens, the carriage holds a printed circuit board. On the board are parts that connect electrically to the pens and a portion of the electronics needed to drive the pens. In the HP DeskJet 820C, all electronics directly used to fire the pens are located on the carriage board (see ***Article 5***).

Print Cartridges. The print cartridges in the HP DeskJet 820C are user-replaceable cartridges that contain both the ink and the mechanism for placing the ink on the paper (thermal inkjet). They are often referred to simply as the "pens." The pens are the same as those used in the HP DeskJet 850 and 870 printers. There are two pens: a black pen and a color pen. The black pen has 300 nozzles spaced at 1/600 inch. The swath height for black is therefore 1/2 inch. The color pen holds three colors of ink: cyan, magenta, and yellow. Each color is printed with a series of 64 nozzles spaced at 1/300 inch. The swath height is therefore approximately 1/5 inch. Colors other than cyan, magenta, and yellow are created by placing dots of these three colors in close proximity in appropriate ratios. Since at a distance of more than a few inches the resolution of the eye is not great enough to discern the individual dots, they blend together visually, forming the desired colors.

Pen Service Station. To maximize the life of the pens and to maintain optimum print quality over that life, it is necessary to service the pens. Servicing includes but is not limited to such actions as capping the pens when not printing so that they do not dry out and wiping them on occasion to prevent ink buildup. The service station includes all the electrical and mechanical parts necessary to perform the servicing actions. In particular, it includes a motor that is used to actuate actions such as wiping and capping. The motor is controlled by an open-loop process.

Power Supply. A power supply is needed to provide energy to the printer. The power supply accepts an ac signal from a standard outlet and converts it to the dc voltages and currents used to power the printer. Because the HP DeskJet 820C will be sold worldwide, it is capable of running on all permutations of 50/60Hz and 110/220V inputs found around the world.

Digital Electronics. The digital electronics are responsible for controlling all of the other electromechanical parts. The digital electronics generally include at least one of each of the following: a microprocessor, a ROM, a DRAM or SRAM or both, a block of custom logic, and an EEPROM. The microprocessor controls all mechanism movements, I/O, the user interface, and print data manipulation if necessary. The ROM holds firmware, and in previous products but not in the HP DeskJet 820C, fonts. The volatile memory is used to hold firmware variables and print data and commands that arrive over the I/O port. The custom logic implements printer-specific functions that require hardware support. The EEPROM holds information that must be retained through a power cycle. In the HP DeskJet 820C, the microprocessor, the ROM, the custom logic, and an SRAM are all integrated into a single ASIC (see ***Article 3*** and ***Article 4***).

Case. The case is the part of the printer that the customer sees, so every effort is made to make it attractive. The case includes a small panel of LEDs and buttons by means of which the user interacts with the printer. The front panel of the HP DeskJet 820C is very simple, consisting of just two buttons and three LEDs. The case also has a door that can be lifted to gain access to the pens.

Driver. In addition to the physical part of the printer, all printer products require a software driver, which resides on the host computer. The driver allows applications software running on the PC to interact with the printer. In most modern operating systems, an application that wishes to print calls the printer driver through the operating system. This model allows the printer manufacturer to supply the driver, so application suppliers don't have to. The exception to this model is DOS, which requires the driver be integrated into the application. Because of the simplifications that can be made to the printer, the HP DeskJet 820C only works with Windows applications, or DOS applications running in a DOS box (see above and ***Article 2***).

HP DeskJet 820C Printing Sequence

To begin the printing sequence, the user chooses Print from the appropriate menu in the application. The application formats the page into the standard description format used by the Windows operating system. Using this format, the application passes a description of the page to the printer driver. The driver reformats the page into a form appropriate for sending to the printer. In the process of reformatting the image, the driver performs various transformations to map the image to the inkjet printing technology. In previous HP inkjet printers, the format used to send data to the printer was PCL, a page description language. In the HP DeskJet 820C, the format is a bitmapped image that can be used to fire the printheads with minimal further transformations.

Once the image is in the right format, data is sent to the printer over the I/O cable. Before the data can be printed, the driver must send commands to the printer that tell it to prepare to print a page. When the driver sends these commands, the printer first uncaps the pens and services them to prepare them for printing. Then it picks a piece of paper and advances it to the first spot where printing will occur.

After the printer is prepared and has enough data in its local memory to print an entire swath, it performs a print sweep by moving the carriage across the page. As it moves the carriage, it pulls data out of its local memory, performs some final formatting, and uses the data to fire the printheads at appropriate times. After the sweep has been completed, the printer advances the paper, waits for enough data to print the next swath to arrive over the I/O, and then, upon command from the driver, prints the data. The process repeats for the rest of the page. At end of the page, again upon command from the driver, the printer kicks the paper, depositing it in the output tray. Assuming that there are no further pages to be printed, the printer then parks the carriage over the service station, caps the pens, and performs other cleanup pen servicing. The printer then waits patiently until the next time it is called upon to print.

Summary

The advance of personal computer horsepower and the uniformity of the Windows printing environment in which HP has control of the printer driver have made it possible to change from the PCL printer model to a PPA printer model. The customer benefit is that PPA printers can provide equivalent levels of performance at a much lower cost.

Microsoft, Windows, and MS-DOS are U.S. registered trademarks of Microsoft Corporation.

Intel486 is a U.S. trademark of Intel Corporation.

PPA Printer Software Driver Design

The software driver for the HP DeskJet 820C printer performs many functions that were formerly performed in the printer, including swath cutting, data formatting, and communications. The driver also includes a PCL emulation module for DOS application support.

by David M. Hall, Lee W. Jackson, Katrina Heiles, Karen E. Van der Veer, and Thomas J. Halpenny

The software driver for the new HP DeskJet 820C printer includes many new functions that need to be performed on the host computer because of the printer's Printing Performance Architecture (PPA). In older PCL (Printer Control Language) printers, these functions were performed in the printer. Fig. 1 shows the differences. These functions include:

- Swath cutting
- Data formatting
- PPA communications
- PCL emulation for DOS application support.

This article provides an overview of the changes necessary for supporting PPA and then discusses each of the functions listed above in more detail.

Driver Overview

Under the Windows® operating system, printer drivers are responsible for supporting a specific API (application programming interface) known as the DDI (Device Driver Interface). This interface gives the driver fairly high-level drawing commands. It is up to the driver to take those commands and produce a bitmap that can be encapsulated in a language and sent to the printer.

Typically, within a Windows printer driver, a rendering engine takes the DDI commands and produces a rendered bitmap. A halftoning algorithm is performed on the rendered bitmap and a halftoned bitmap is produced. This halftoned bitmap is typically in a format that can be encapsulated in a language such as PCL and then given to the printer.

For the HP DeskJet 820C, this halftoned bitmap has to be put through additional processing as shown in Fig. 1 to create data that is ready to be printed by the printer's electronics directly. This additional processing includes swath cutting and sweep formatting.

Since the HP DeskJet 820C does not understand PCL (Printer Control Language), a PCL emulation module is necessary to provide support for DOS applications. The DOS application data stream is captured by a DOS redirector and passed to the PCL emulator, which produces a halftoned bitmap ready for swath cutting.

PCL versus PPA

Fig. 2 shows the printing model for PCL printers. For PCL printers, the process of encapsulating the halftoned bitmap is fairly straightforward. Raster data from the halftoned bitmap is compressed, PCL wrapped, and then sent to the I/O module. The reason that this is a simple process is that PCL printers are designed to receive data in the same format as the halftoned bitmap. PCL printers unwrap the data into an internal buffer and perform the necessary swath cutting and data formatting internally.

Fig. 3 shows the printing model for PPA printers. For the HP DeskJet 820C, the PCL encapsulator is replaced with an SCP data encapsulator. SCP (Sleek Command Protocol) is an HP-proprietary command language. This module contains swath cutting functionality, data formatting, SCP language encapsulation, and printer status management.

Raster data from the halftoned bitmap comes into the SCP data encapsulator, goes through the SCP manager, and eventually arrives at a raster block within the swath manager. The swath cutting state machine examines the data and determines the appropriate sweep to generate. A sweep is a collection of rasters appropriate for the printer mechanism to print while it sweeps the printhead over the paper.

Once the sweep is generated, it is given to the sweep formatter. The sweep formatter is responsible for taking the sweep data and putting it into the appropriate format for the HP DeskJet 820C internal hardware. Then the data is compressed, wrapped in SCP, and handed off to the I/O layer.

The I/O layer is responsible for communicating with the printer by wrapping the data stream in VLink and IEEE 1284 protocols. VLink is an HP-proprietary link-level protocol and IEEE 1284 is an industry-standard physical-layer protocol.

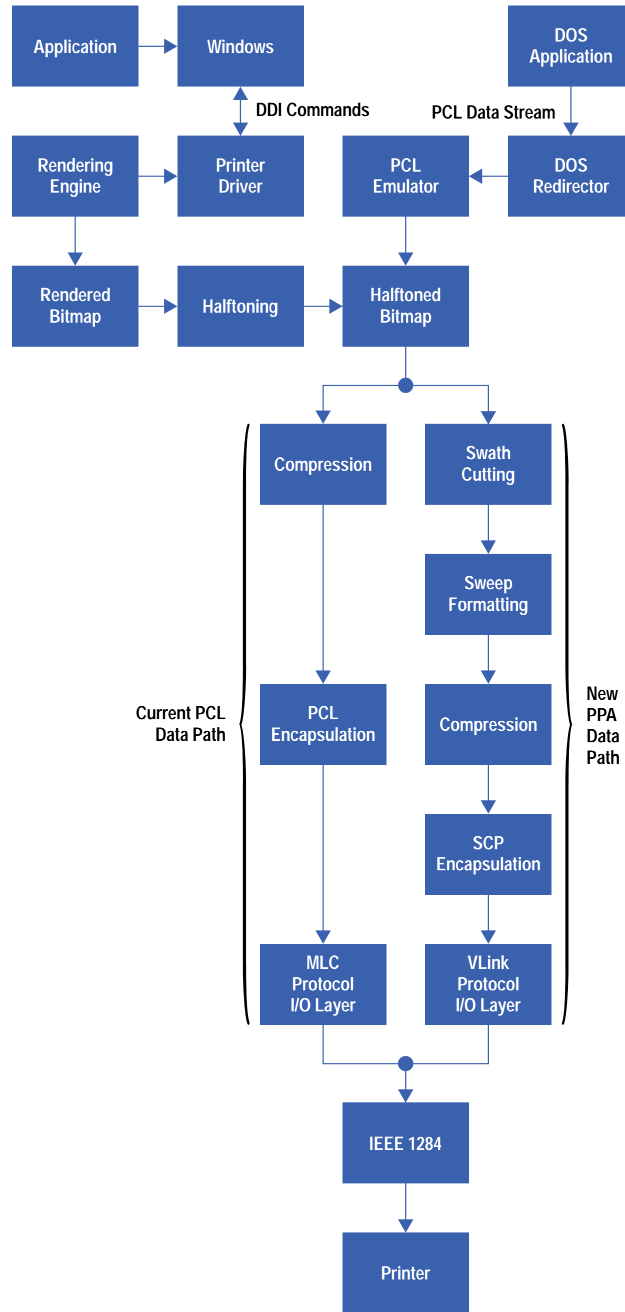


Fig. 1. *Printer driver functional block diagram, showing differences between PCL and PPA data paths.*

Performing Swath Cutting on the Host

Swath cutting is the process of taking a page of halftoned raster data and producing sweep data appropriate for the carriage electronics to print as the printhead is sweeping across the page. Swath cutting has historically been part of printer firmware, but in the HP DeskJet 820C printer, it is part of the software driver running on the host computer. Typically, a swath manager encapsulates a swath cutting engine and receives as input a bitmap representation of the page to be printed. The swath manager is responsible for determining how the pens and paper should be moved and when and how the pens should be fired to produce the printed page. The swath manager must balance the often conflicting goals of printing with the highest possible print quality and printing as fast as possible. The swath manager must be aware of certain printer-specific attributes such as printhead alignment and strategies to minimize line feed error. In PPA, swath management is performed on the host computer.

The process of swath cutting can be readily modeled using a state machine. Consider the example shown in Fig. 4. A state machine capable of processing this page would need to contain five states: Top of Page, Blank Skipping, Black Text Printing, Color Graphic Printing, and End of Page. Thus, we can create the state machine shown in Fig. 5. A particular instance of a state machine exists for each print mode the swath manager supports. For example, there could be a print mode for pages that only have

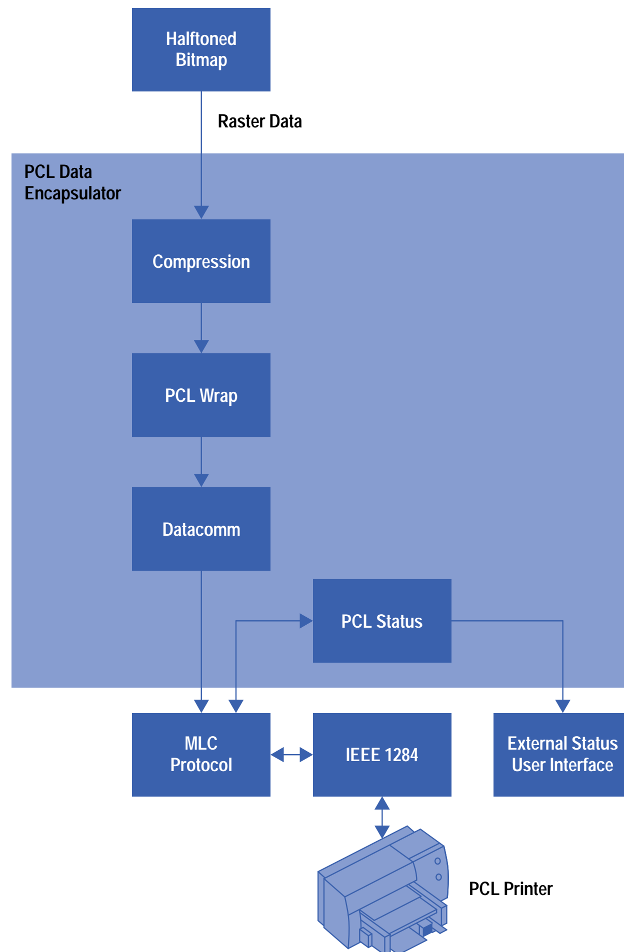


***Fig. 2.** PCL printing model.*

black text on them, another print mode for pages with black and color, and yet another print mode for pages with complex graphic images.

As the state machine begins to examine the data on the page, it starts in the Top of Page state. The first data it comes to is a series of blanks. This would cause it to move to the Blank Skipping state. During this transition the swath manager would typically load the page. While in the Blank Skipping state, the swath manager would advance the paper. Next, it would encounter a black text region and move to the Black Text Printing state. Depending upon the type of printing being done at that time, this transition may produce a sweep.

Assume that for this print mode, the data on the page is being printed by making two sweeps for each line. Thus, in making the transition from Blank Skipping to Black Text Printing the printer could print the first pass of the black text region with the bottom half of the printhead, advance the paper half a printhead height, and then enter the Black Text Printing state. During the next sweep generated, the Black Text Printing state would finish the lines that were printed during the transition and continue printing the black text region (see Fig. 6). The data on the page would continue to be consumed and transitions made between states until the End of Page state is reached.

Obviously, this example is a simple one. The number of states and the number of transitions to consume data for a real page can be quite large. Using PPA we have the opportunity to perform the resource-intensive task of swath cutting on the host. This allows greater flexibility in developing machines with unique print modes, which provides the opportunity for higher print quality and throughput as well as reduced mechanism costs.

PPA Data Formatting

The HP DeskJet 820's Printer Performance Architecture requires the host to perform the majority of the data manipulation. The data that is sent to the printer must be in a format that is very close to the final form used to fire the printheads. The main difficulty in formatting the data for the printhead lies in the fact that the data doesn't come out of one position on the carriage mechanism. Instead, there are two columns for each of the four pen colors. Each column is at a different vertical and horizontal offset from a relative zero carriage position. To minimize the cost and complexity of the electronics in the printer mechanism, the data sent from the host to the printer must be ordered so that it is ready to go directly into these offset printheads in the appropriate order so that it is fired at the correct locations on the page. This ordering is based on:

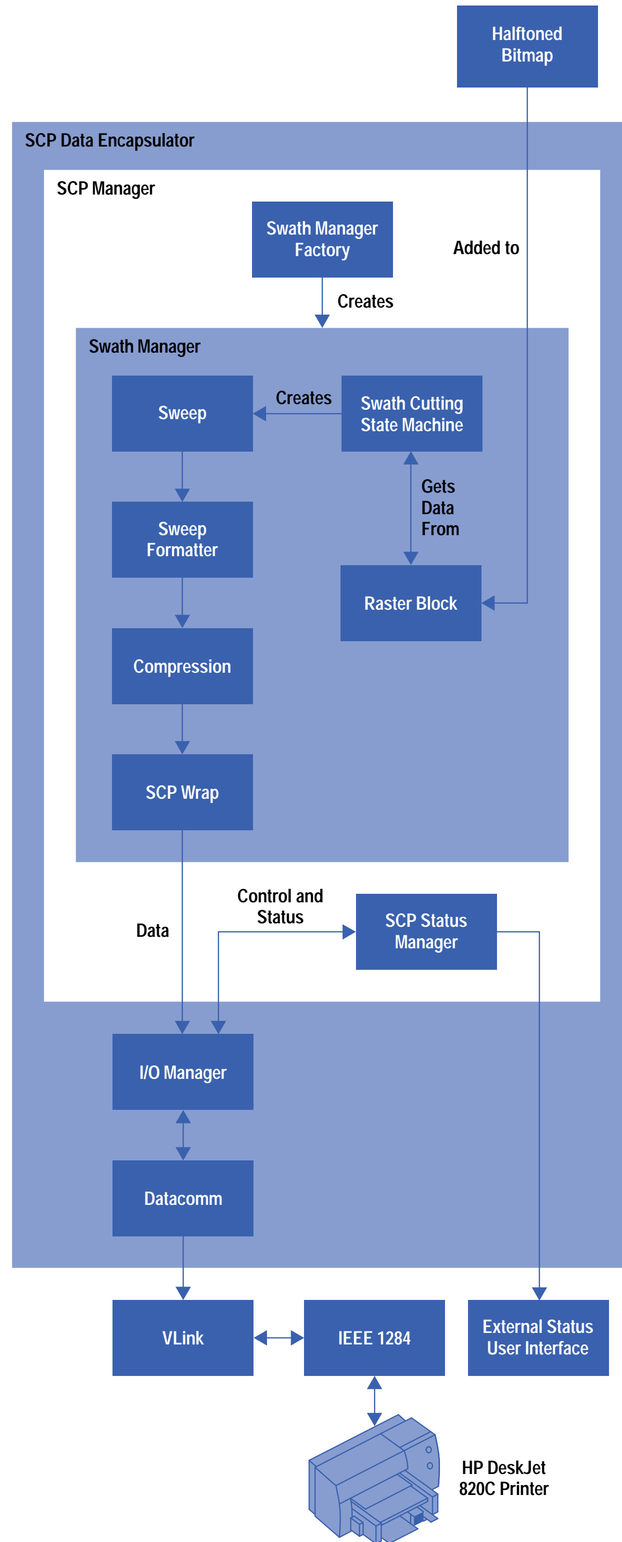


Fig. 3. *PPA printing model.*

- The starting page position of each color
- The *servant* architecture in the printer hardware (described later)
- The printhead (see Fig. 7).

To print a page, it is necessary for the carriage mechanism to move back and forth across the page, firing drops of ink as it moves. Each movement of the carriage across the page is called a print sweep. When the driver receives a page to print from some application, it renders the page into a halftoned bitmap. At this point, a PCL printer driver would send compressed and encapsulated PCL data directly to the printer. The PPA printer driver uses the swath cutting state machine to generate a

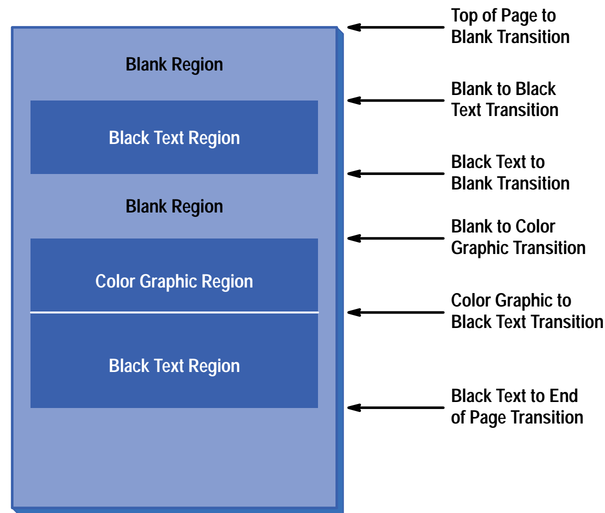


Fig. 4. *Swath cutting state machine transitions for a typical page.*

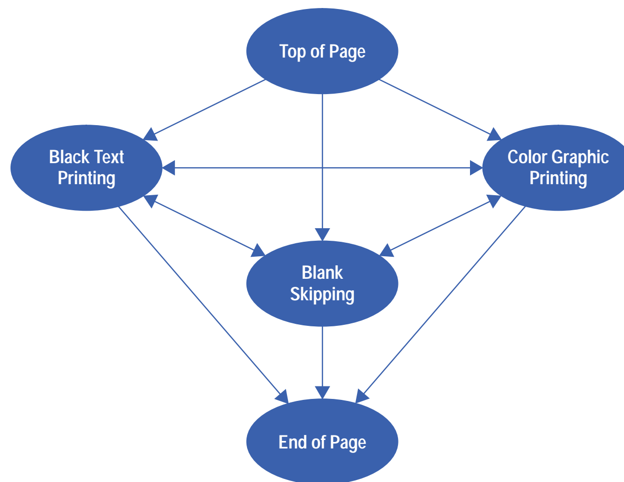


Fig. 5. *Swath cutting state machine.*

Fig. 6. *(a) In making the transition from* Blank Skipping *to* Black Text Printing, *the printer prints the first pass of the black text region with the bottom half of the printhead, advances the paper half a printhead height, and then enters the* Black Text Printing *state. (b) During the next sweep generated, the* Black Text Printing *state finishes the lines that were printed during the transition and continues printing the black text region.*

swath of data that can be printed by a single pass of the pen carriage. The resulting swath of data is passed on to the sweep formatter, which manipulates the data into a buffer that can be copied directly to the printheads. The print sweep formatter uses knowledge of the pen carriage, hardware, and firmware architecture to prepare and reformat the data into a print sweep.

The number of print sweeps required on a given page is dependent upon:

- The amount of data on the page (text or dense graphics)

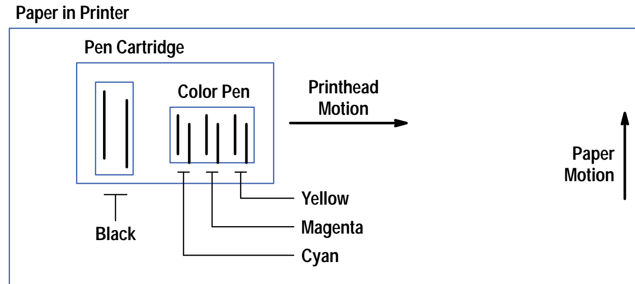


Fig. 7. *HP DeskJet 820C print cartridge layout. The lines correspond to nozzle columns and their general configuration on the printer carriage.*

- The print mode selected by the user (best, normal, or econofast)
- The paper type (plain, glossy, transparency, or special).

For each print sweep, the host sends two pieces of information to the printer. The first is the PRINT_SWEEP data, a buffer of image data sent before the PRINT_SWEEP command, which contains an entire sweep of swing buffer data blocks in the correct order. The second piece of information is the PRINT_SWEEP command, the mechanism by which the driver tells the printer where and how to place the print sweep data on the page. A PRINT_SWEEP command contains minimum and maximum positions for each pen column, the print direction, print speeds, and NEXT_PRINT_SWEEP information.

The PRINT_SWEEP command information is calculated by the printer driver based upon:

- Which pens are active (black, cyan, magenta, yellow)
- The starting and ending locations on the page for each pen color
- The direction of the print sweep
- The servant architecture:
  - The distances between pens
  - The distances between odd and even columns within a pen
  - The 0,0 position in relation to the pen columns.

Servant Architecture

The servant hardware (see ***Article 4***) is composed of a pair of buffers, called *swing buffers*, for each column of the printhead (two columns per color). To build a print sweep, the driver must:

- Separate the image into CMY planes, or primitive data blocks
- Separate the primitive data blocks into swing buffer data blocks
- Order the swing buffer data blocks into a servant image.

A primitive data block (a bitmap image of each plane for each color) is created by the driver. Each primitive data block needs to be split into two separate swing buffer data blocks: an odd block and an even block. This is necessary because of the pen design, which consists of two offset columns, as pictured in Fig. 8.

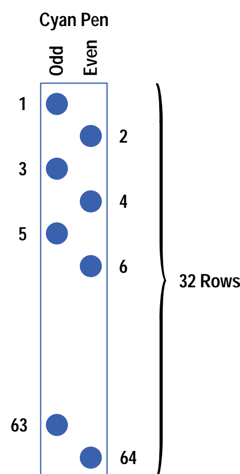


Fig. 8. *Each color pen has two offset columns of nozzles.*

Each column on the color pen has 32 nozzles. The color pen has a height of 64/300 inch. For any given column of data, rows 1, 3, 5, ..., 63 will be part of the odd column and rows 2, 4, 6, ..., 64 will be part of the even column.

The even and odd swing buffer data blocks are each 8 bits wide, the width of servant RAM, and each is the height of a printhead nozzle column. Swing buffer data blocks are cut for each primitive color and for either the even or odd nozzle column. Thus, each swing buffer data block contains every other row from the primitive data block.

Fig. 9 shows a simplified example of a primitive data block. Each byte is a buffer of data that is one byte (8 pixels) wide by N rows high, where N is the number of nozzles in a printhead column. For the example in Fig. 9, N is 6, while N is 32 for the HP DeskJet 820C color printheads.

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35
36	37	38	39	40	41
42	43	44	45	46	47
48	49	50	51	52	53
54	55	56	57	58	59
60	61	62	63	64	65
66	67	68	69	70	71
72	73	74	75	76	77
78	79	80	81	82	83
84	85	86	87	88	89
• • •	• • •	• • •	• • •	• • •	• • •
120	121	122	123	124	125

Fig. 9. *Primitive data block organization for a printhead that has two columns of six nozzles per color. Byte n (n= 0, 1, 2, 3, 4, 5) is a buffer of data 8 pixels wide by 6 rows (nozzles) high. The HP DeskJet 820C printheads have two 32-nozzle columns per color, as shown in Fig. 8.*

Each column of the primitive data block in Fig. 9 is divided into four swing buffer data blocks with bytes relocated to the positions shown in Fig. 10. Only the cyan pen is shown, and only two of the swing buffer data blocks for each column of Fig. 9 are shown. The drawing would be similar for the magenta and yellow pens.

Swing Buffer Data Blocks, Byte 0		Swing Buffer Data Blocks, Byte 1	
CO:0	CE:0	CO:1	CE:1
0	6	1	7
12	18	13	19
24	30	25	31
36	42	37	43
48	64	49	55
60	66	61	67

CO:x = Cyan Odd Printhead Column: Primitive Data Block #
CE:x = Cyan Even Printhead Column: Primitive Data Block #

Fig. 10. *Swing buffer data blocks for the example primitive data block shown in Fig. 9.*

Once the data is in the form of even and odd swing buffer data blocks, the blocks must be ordered and sent to the printer. This ordering is done with knowledge of the column spacing on the printhead and knowledge of the order in which the servant architecture will require the data. The printer driver controls the order in which the columns will trigger via fields in the PRINT_SWEEP command. The ordered swing buffer data blocks are then sent down as PRINT_SWEEP data ready to be loaded into the primitive swing buffers in the printhead.

Each primitive swing buffer consists of two 8-bit columns, separated by a *swing trigger point*. While the servant print process is unloading one side of the odd column swing buffer, the other side of the odd column swing buffer is being loaded by the servant load process. Once the byte is loaded, the servant print process fires one bit by 32 rows at a time for each pen column in the color pen. When the servant print process has unloaded all eight bits, it crosses a swing trigger point, and the

servant print process switches to the other swing buffer and triggers the servant load process to load the empty swing buffer. The pen fires one bit by 32 rows at a time for each pen column. The servant (printer) is responsible for any complexity involved below the byte level.

When all of the swing buffer data blocks have been consumed by the printhead, the carriage mechanism uses the NEXT_PRINT_SWEEP information to position itself for the start of the next print sweep.

Because the PPA printer relies upon the driver to format the data appropriately, the architecture does not require the printer firmware to have any knowledge of the operations just described. Thus, the cost and complexity of the electronics in the printer mechanism are significantly reduced.

PPA Communication

One of the goals of the HP DeskJet 820C printer is to provide continuous feedback to the user during any printing operation, and to guide the user during problem solving. To accomplish this, the driver requires a mechanism to ask the printer for information and to allow the printer to notify the driver whenever something happens (the printer is out of paper, the user opened the cover, etc.). The mechanism used by the PPA driver to communicate with the printer is called *status messaging*.

To notify the user to align the print cartridges when a print cartridge has been changed, that the top cover is open, or that something else needs attention, a bidirectional link with the printer is required. Two new HP-proprietary protocols allow the driver to communicate bidirectionally with the HP DeskJet 820C: VLink packet protocol and Sleek Command Protocol (SCP). Previous HP DeskJet printers used an I/O packetizing protocol called MLC (Multiple Logical Channel) and a proprietary HP printer command protocol. For PPA, VLink replaces MLC, and SCP replaces both PCL and the old printer command protocol.

While giving users error messages might seem to be a luxury they could do without, the real reason to have a protocol like VLink is that it is useful to figure out what is wrong when, for example, the printer's input buffer fills up, the printer stops accepting data, and the host is unable to send even one more byte. This often happens and is temporary, but in the days before bidirectional protocols, the driver would sometimes wait and wait to be allowed to send again, and it didn't know whether the delay was because the top cover had been opened, a print cartridge had failed, or a fatal error had occurred. It is helpful to know whether to abort the job or ask the user to insert a print cartridge or close the door. With a bidirectional protocol, the printer tells the driver exactly what the problem is, and the driver can decide what action to take next.

A bidirectional link is not required for printing or to have limited status feedback from the printer. However, unlike PCL printers, which can accept either PCL data wrapped in MLC or raw PCL data, PPA printers can only interpret data wrapped in VLink and SCP. Thus, while MLC is an option that can be added when a bidirectional link exists, VLink must handle printing with and without a bidirectional link as well as printing to a file.

Based on VLink's channelization features, there are two paths the data can take to the printer. One is for image data (the dots that will go on the page), and the other is for command data. Command data includes commands sent to the printer, such as "Print this sweep," requests for information, or queries, such as "What print cartridges are installed?", and status information, termed *autostatus*, such as "The top cover is open." Sending image data is easy from an I/O standpoint—if the printer has room in its buffer, the driver will send the data. Since command data must be sent and also received (autostatus may come in at any time), it is by nature a more complex affair.

As shown in Fig. 11, data that comes in from the front end of the driver goes through the data encapsulator, like PCL printer drivers, but from there it goes through several new objects. The SCP manager wraps the data in SCP and sends it to the I/O manager, which provides an interface to the datacomm objects. The VLink layer wraps the data in the VLink protocol and sends it to the IEEE 1284 layer and out to the printer.

Data that is sent by the printer, such as notifications that something is wrong, are put in the printer's output buffer. The driver spawns a hidden executable at the beginning of each print job called the *port sniffer*, which checks the port every half second to determine if the printer has sent any data. If so, the data is routed through the IEEE 1284 layer to the VLink layer, which then posts a message to the I/O manager's hidden status window.

The status window uses a callback to call into the SCP manager, which translates the status information, and if the message is something that should be displayed to the user, puts it on the event list. The event list prioritizes the messages on it so that the most important message gets sent to the HP Toolbox, which displays the dialog box to the user. If the message is an error, it may get resolved (for example, the user puts paper in the printer and presses the Resume button). The message is then routed up through the same path and deleted from the event list. The Toolbox takes the dialog box down and displays the next most important message, if there is one.

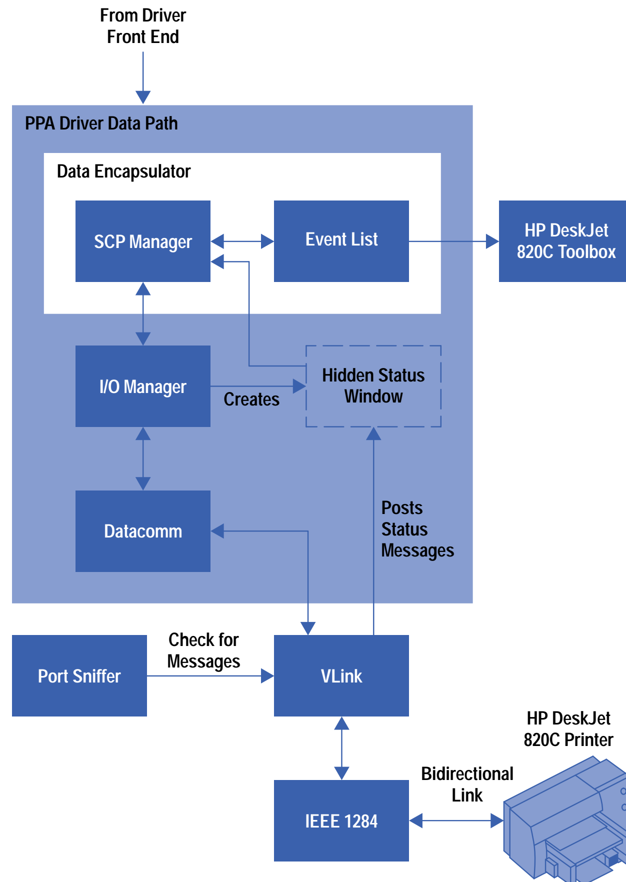


Fig. 11. *PPA status messaging architecture.*

Internal Objects in PPA Status Messaging

PPA status messaging involves several high-level modules and objects: the SCP (Sleek Command Protocol) manager, the I/O manager, the VLink module, and the event list (see Fig. 12).

SCP Translator. The function of the SCP translator object in the SCP manager is to encode data into the SCP format and decode messages received in the SCP format from the printer into query replies and event information. The SCP translator does not send SCP data directly to the I/O manager, since memory management for the data buffers is done in the SCP translator's clients, which are the swath manager and the status manager. The client of the SCP translator passes in a pointer to the data, an empty buffer, and the maximum data length. Once the data has been packaged, if the SCP translator finds that the data is larger than the buffer, it will return an error. Otherwise, it will pass back the actual SCP data length. The goal in designing the SCP translator was to encapsulate the Sleek Command Protocol so that changes in SCP in the firmware affect clients of this module as little as possible.

Commands in SCP use the format shown in Fig. 13. The command specifier field identifies the SCP command. The length field indicates the number of bytes in the data field. The data field does not exist for every command.

Priorities. Priorities allow the printer to execute commands in a different order than received. This may be necessary when a command cannot complete execution and it is desirable for the printer to process queries so the driver can find out what the problem is. Priority levels are defined in the SCP translator and the clients can set whatever priorities they like. Standard priority levels are defined as shown in Table I.

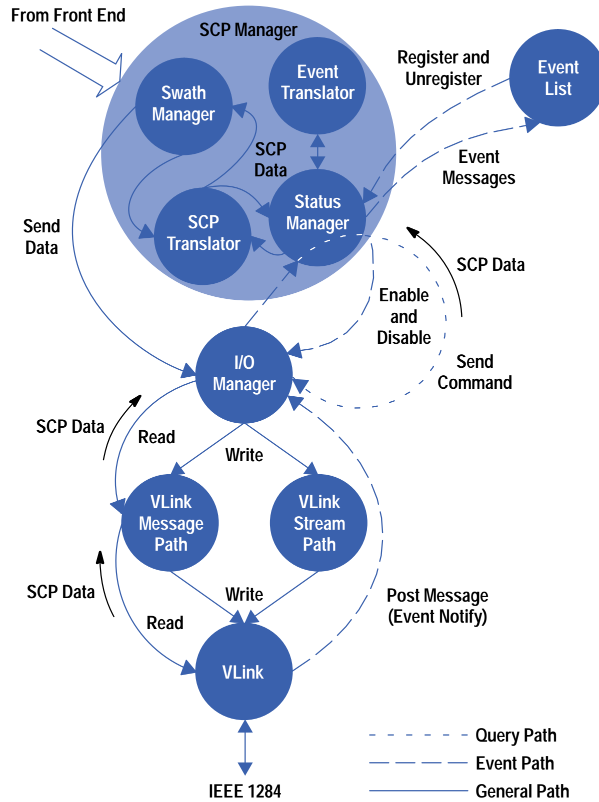


Fig. 12. *Calls between status messaging objects.*

Fig. 13. *SCP command format.*

Table I
Command Priorities

Command	Priority
Printing Commands	Low
Queries	Medium
Initializing and Deinitializing the I/O Link	High
Recovering from Errors	Recover
Canceling	Cancel
Restarting the Printer	Restart

It is assumed that the swath manager will send all of its printing commands (LOAD_MEDIA, PRINT_SWEEP, EJECT_MEDIA) at the lowest priority. Any queries it needs to make will call into the status manager. All queries should be at the same priority and higher than printing commands. It is up to the clients to set priorities.

Status Manager. The status manager manages messages to and from the printer. These messages can be broken into two categories: events and queries. Events are unsolicited notifications by the printer (i.e., autostatus) that something has occurred to change the state of the printer, such as “the door is open.” Queries are requests for information made by the driver to the printer, such as the pen IDs of the installed pens. The status manager tracks the state of the printer and creates

events when state changes occur. For example, when the Resume button is pressed, an internal state change occurs. This state change is recognized by the status manager and reported as an event to the event translator.

When the status manager receives notification of an event, it determines what has changed and whether the event is something the event translator has requested to know about. If it is, a callback in the event translator is called.

Upon starting a print job, the status manager queries the printer to get the current state of events. No event notification will be received until an event occurs in the printer.

Event Translator. This module exists between the event list, which is Windows-specific, and the status manager. The event translator translates the bit-field data, which is returned to the status manager by the printer in autostatus, into events. New events are added to the event list by the status manager, and events that are no longer valid (e.g., the door was open but the user shut it) are removed from the list. The event list orders the events reported to it according to their importance to the user, and tells the status monitor which dialog box to display. From most important (1) to least important (10), the following event priorities are used: (1) I/O errors, (2) paper jam, carriage stall, or maximum thermal limit, (3) pen failure, (4) wrong pen, (5) low or out of ink, (6) pen missing, (7) out of paper, (8) cover open, (9) dry timer, (10) new pen.

I/O Manager. This module is intended to glue the VLink module, which is Windows-specific, to the SCP manager, which is shared. Handling for events, queries, and buffer management must be performed by the I/O manager in addition to sending data to the printer as quickly as possible.

Events. The I/O manager creates a hidden window so that when the printer sends unsolicited event notification, Windows messages to that effect can be posted to this window by the VLink module. When the I/O manager processes this window message, it will read the SCP data buffered by VLink and call a callback in the status manager, passing in the SCP data.

Queries. To get replies to queries, the inquiring module calls VLink, specifying a buffer in which to place the reply. VLink checks this query reply buffer to see if anything has been returned in response to the query. If so, it immediately returns with the SCP data. If not, it polls the incoming channels for a specified timeout period to attempt to retrieve the reply. If a reply is received before the timeout period expires, the SCP data is passed through to the status manager.

Datacomm Paths. The image and command datacomm paths send data to the printer as long as there is space in the buffer. If space runs out, the command datacomm path waits until more space becomes available. The image data is handled differently. If space runs out while sending image data, the image datacomm path returns to the caller, allowing it to render more swaths until more space becomes free in the printer.

VLink. The VLink module must package data in a protocol the printer recognizes, and send only as much data as the printer can take, as quickly as possible. VLink must also unwrap data from the printer and route the messages to the appropriate clients.

The VLink protocol replaces MLC (Multiple Logical Channels) for the HP DeskJet 820C. Like MLC, VLink's intent is to provide a way for the host and the peripheral to exchange data. Unlike MLC, VLink is not optional. All data going to the printer must be wrapped in its protocol. In addition, VLink is streamlined or "sleek," and doesn't have many of MLC's features. MLC supported multiple logical channels, while VLink supports two outgoing and three incoming channels.

Outgoing Channels. The printer accepts data in either its input buffer or its command buffer. The VLink module specifies which type of data it is sending through a field in the VLink packet header. A template of a VLink packet is shown in Fig. 14.

Fig. 14. *VLink packet format.*

Image data is sent to the printer's input buffer on the image data output channel. Commands and queries are sent to the command buffer on the command data output channel.

Incoming Channels. Since a bidirectional link cannot be guaranteed, all incoming data is optional. This is necessary for file dumps and bad cables, and miscellaneous communication problems.

The printer periodically notifies the host how much buffer space is left in the printer. This is known as *credit*, and the printer sends notification for both the command and input buffers on the credit input channel. The VLink module will not send more data than the available credit.

VLink accepts two types of data packets from the printer in addition to credit packets: query replies, which are expected on the status input channel, and a collection of bundled items regarding printer status (such as out of paper), called autostatus messages. Autostatus messages ultimately map to events.

An autostatus message from the printer consists of a bit collection of several long words representing the current state of the printer. For example, when the door is opened, the door open bit in the collection is set to true. A report is generated on the autostatus input channel when any of these bits are toggled.

When the VLink layer receives some data, the data is identified as either credit, a query reply, or an autostatus message. Credit is interpreted and handled within the VLink module. A query reply or an autostatus message is buffered internally so that the clients can read it later.

If a received message is an autostatus message, the VLink layer posts a Windows message to the I/O manager indicating that an autostatus message is waiting to be read. When the I/O manager processes the Windows message, it reads the buffered autostatus message. Posting a message is necessary so that VLink can be free to poll the data lines for more incoming data from the printer.

Once the buffered message has been read, it is deleted. Only one query reply and one autostatus message can be buffered at a time. If a new message comes in before the original message can be read, the new message replaces the old one. It is for this reason that no additional printer queries should be made while waiting for a reply. No harm is done if a new autostatus message overwrites the old message because the same information is contained in each message and the newest message is the most relevant.

PCL Emulation for DOS Application Support

The development period of the HP DeskJet 820C coincided with most users rapidly transitioning away from DOS applications towards Windows applications. While we expected that most users would use the printer in its optimized design center, we recognized that we needed an adequate bridge to the few DOS applications that would continue to be used.

The HP DeskJet 550C printer was the final printer to be supported by most DOS applications, so the solution had to be functionally compatible with this printer and provide equally good print quality. We chose to provide compatibility with the HP DeskJet 660C printer, which was a contemporary printer that satisfied these requirements and provided an internal interface that enabled us to separate the PCL personality from the printer engine firmware. We planned to port the PCL personality functions to the HP DeskJet 820C printer driver, encapsulating them in a ***PCL emulator*** module. The required printer-engine functions would then be supplied by the rest of the HP DeskJet 820C driver. In this way, we could minimize design changes and maximize the chances of identical compatibility. If a DOS application is run from an MS-DOS prompt window, also referred to as a ***DOS box***, the printer driver can intercept the PCL data stream that the DOS application sends to the PC's parallel port and redirect the data stream to the PCL emulator.

The HP DeskJet 820C PCL emulator encapsulates the HP DeskJet 660C formatter and text engine code. The design of the HP DeskJet 660C firmware was such that all interfacing to the external mechanism was done through a well-defined API internally known as the ***Ed Interface*** (see Fig. 15).

The Ed Interface resides between the formatter and font manager and the rest of the firmware. It is a collection of function calls to the support code in the firmware. Since we reused the formatter and font manager code, we provided the equivalent firmware functionality by mapping the Ed Interface calls into HP DeskJet 820C support objects.

The functions of the formatter and text engine firmware code were written in C, and as such are functions in the PCL emulator application (Fig. 16). The PCL emulator application provides C++ objects that encapsulate the functionality expected by the Ed Interface.

The PCL emulator application is designed to receive a file name that contains the PCL data to operate on. Interfacing between the internal PCL emulator object and the external driver is provided through a PCL personality object.

The PCL emulator is implemented as an executable application because the original firmware code expects to be a separate task, and this implementation allows almost direct porting of the HP DeskJet 660C firmware code. The PCL personality provides the handler functions and the external interface for receiving the PCL file name.

To allow DOS applications to print to the HP DeskJet 820C, it is necessary to capture the data generated by the DOS applications. This process is referred to as ***DOS box redirection***. Essentially, it is necessary to capture the bytes intended for the parallel port and put them into a file so that the PCL emulator can properly interpret the data.

Under Windows 3.1, DOS box redirection is not part of the operating system, so it was necessary for us to provide a redirection solution. This functionality is provided by a redirector VxD (virtual device driver), a redirector DLL (dynamic link library), and a redirector EXE (executable), as shown in Fig. 17. These three pieces capture the data stream and put it into a temporary file. This file is then handed to the driver, and the driver hands it to the PCL emulator.

Under Windows 95 (Fig. 18), DOS box redirection is provided by the Windows printing system, so our redirector solution is not necessary for spooling to work under Windows 95. PCL printers essentially get DOS box redirection free. PPA printers need to intercept and perform PCL emulation on the DOS data stream. Microsoft provides a replaceable module called a language monitor where the data stream can be intercepted. The language monitor is a 32-bit DLL called directly by the spooling subsystem. The language monitor takes the incoming buffers, writes them to a temporary file, and passes the file name to the driver.

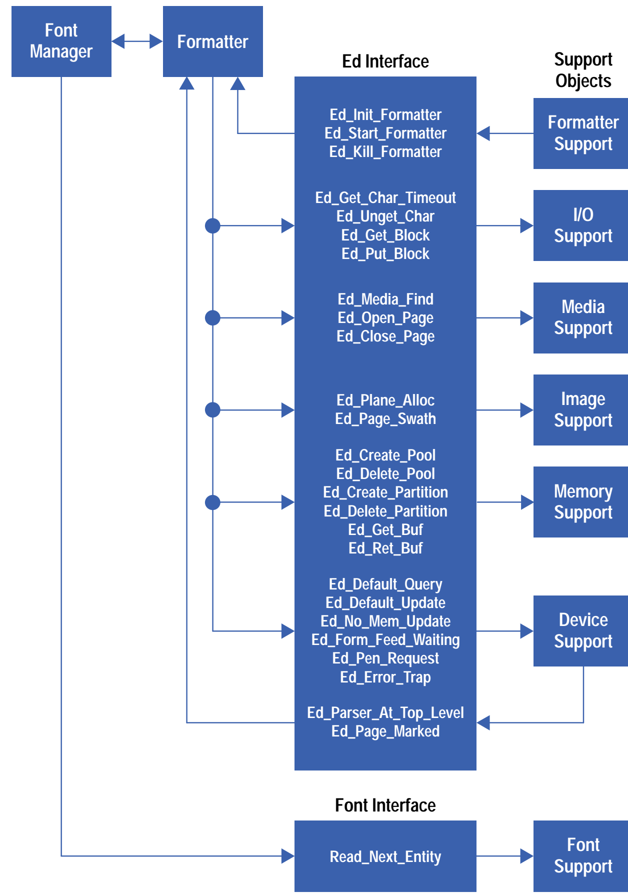


Fig. 15. *PCL emulation is provided in the HP DeskJet 820C printer by mapping the existing Ed Interface calls to DeskJet 820C support objects.*

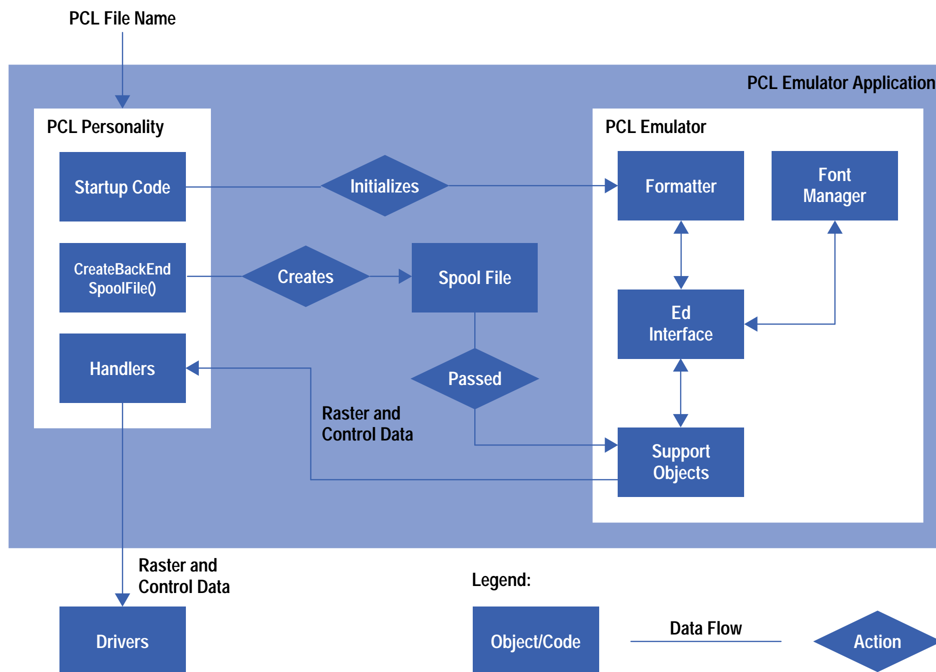


Fig. 16. *The PCL emulator application provides C++ objects that encapsulate the functionality expected by the Ed Interface.*

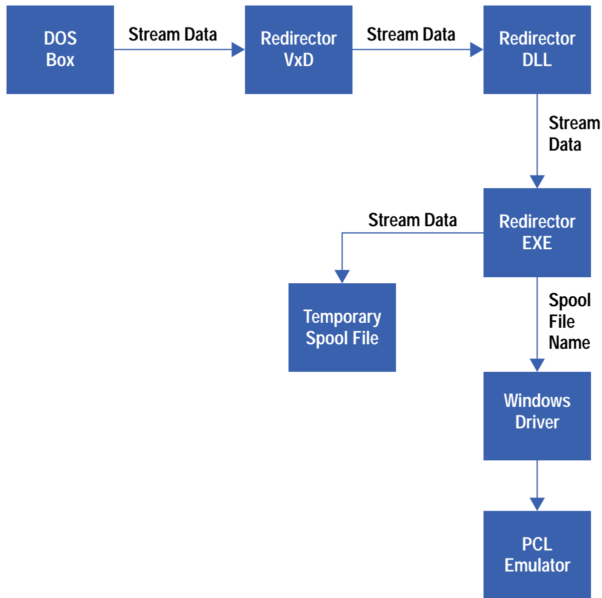


Fig. 17. DOS box redirection for Windows 3.1.

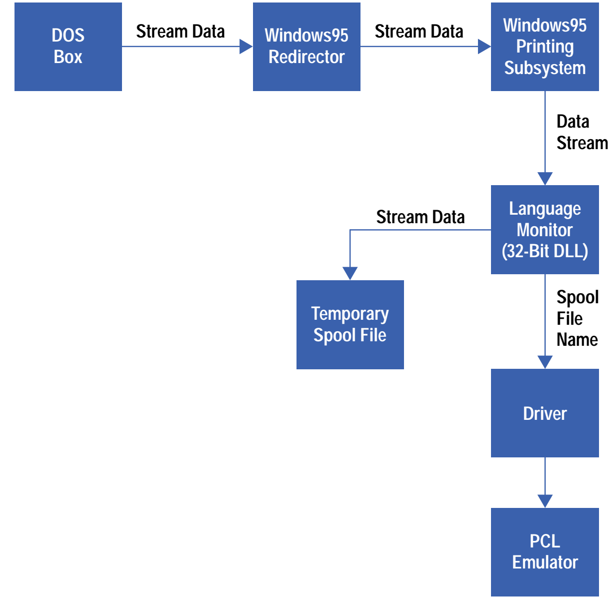


Fig. 18. DOS box redirection for Windows 95.

Porting the Firmware

The process of porting the C-language code from the HP DeskJet 660C presented several challenges. The original firmware was developed for a Motorola 68000 processor, while the printer driver runs on the Intel 80x86 processor in Windows 16-bit mode.

These two hardware platforms have conflicting ways of addressing memory for data types larger than a byte—the former is big endian (the most significant byte comes first) and the latter is little endian. As long as a data element is consistently accessed with the same data type, there is no problem. However, there are places in which a data type is written as several single bytes, then read as 2-byte or 4-byte quantities. We needed to identify and change the code in these places.

The original font data that described the glyph (shape) information for the text engine was a single block of 250K bytes of read-only data. This block was mapped to five blocks of resource data, since each block had to be less than 64K bytes for Windows 16-bit mode. These blocks are discardable, meaning that the operating system can load them when it needs to read some data, but to load other code or resource blocks when Windows has run out of memory, they can be replaced by other blocks.

The original firmware's text engine depended on a special hardware component that rotated font glyph data from horizontal to vertical orientation, could double the size of the data, and smoothed the edges of a glyph using several rules for HP Resolution Enhancement technology (REt). Since this hardware was not available to the printer driver, we were able to simulate the first and second of these functions in software. We determined that the print quality would still be better than the HP DeskJet 550C even if we did not simulate the REt rules. The resulting software simulation executes more slowly, but the orginal firmware design included a font cache, which minimizes the the number of times that we need to execute this function.

Some further syntax modifications were necessary. The printer driver is capable of supporting more than one of the same printer, for example, a printer on the LPT1 port and another on the LPT2 port, and these printers can be printing at the same time. For Windows to be able to execute multiple instances of the PCL emulator, the code must be compiled in the Windows ***medium-memory model***. This required that many C-language pointer variables be designated ***far pointers*** rather than the more efficient ***near pointers***. Also, some subtle syntax correction was necessary because an integer data type is 32 bits for the 68000, but 16 bits for the 80x86.

The PCL emulation implementation was accomplished in a staged development process. Two months before the first printer driver components to support the HP DeskJet 820C became available, we were able to build a DOS application that was totally decoupled from a printer driver. It would accept a test input stream of PCL data and map the input to an output file of raster data, which could be printed on the HP DeskJet 850C, which was mechanically identical to the target HP DeskJet 820C. Using our test center's extensive suite of input test files, we were able to stabilize the porting implementation, within the limits of the DOS application. For example, we noticed that the DOS memory allocation algorithm would fragment memory that was being continually allocated and freed, so that eventually a memory allocation request would fail. However, when we moved on to a subsequent stage in which we depended on the Windows memory manager, we found that this memory fragmentation no longer occurred. Once the DOS port was stabilized, we integrated the PCL personality into the printer driver, using the HP DeskJet 850C output target path, while still providing an input file of PCL. Next we introduced

and stabilized the DOS redirector input path. When the HP DeskJet 820C output target path finally became available, we were able to switch to it cleanly, and the PCL emulator became an effective tool to help stabilize the new output target path. Finally, we completed the target functionality, always building upon a stable base.

To summarize, by reusing original firmware code we were able to provide identical PCL functionality for PPA printers. Providing support for the Ed Interface API allowed the firmware code to be reused with little design modification.

Windows is a U.S. registered trademark of Microsoft Corporation.

PPA Printer Firmware Design

Hewlett-Packard's new Printing Performance Architecture (PPA) includes a significantly reduced set of printer firmware. "Don't touch the dots" was the firmware designer's golden rule. This means that the firmware and processor do only mechanism control, I/O, command parsing, status reporting, user interface, and general housekeeping functions.

by Erik Kilk

A significant factor in Hewlett-Packard's new Printing Performance Architecture (see ***Article 1***) is the reduction of the processing power embedded in the printer. Using the host PC for all image formatting leaves only motor, print cartridge, I/O, user interface, command, and status functions to be controlled by the firmware. This results in significant cost savings by reducing processor needs and by reducing ROM and RAM requirements. The goal, which was achieved, was to reduce the ROM requirements to 64K bytes.

Fig. 1 shows the traditional firmware architecture used in HP DeskJet printers. The firmware receives from the host PC a combination of text, text formatting commands, and raster graphics data. This is formatted according to the Hewlett-Packard PCL printer language specification. The information to print arrives at a page description level, which requires firmware to rasterize a bit image, generate and place fonts, and format and cut the image into swaths according to the requirements and format of the print cartridge.

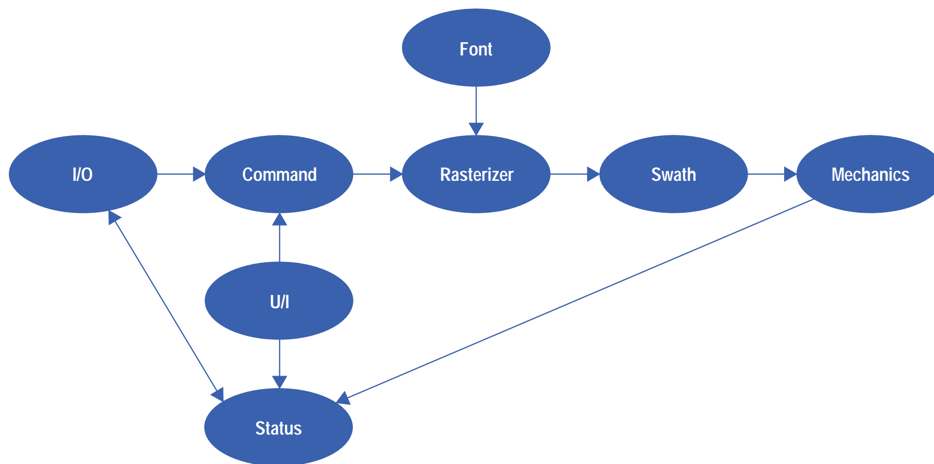


Fig. 1. *Traditional HP DeskJet printer firmware architecture.*

At the I/O layer, previous HP DeskJet printers make use of the Multiple Logical Channel packetizing layer (MLC, being proposed as IEEE standard 1284.4) to offer multiple connections between a host and a printer. PCL and an HP proprietary peripheral status language share the bidirectional parallel port.

The rasterizing step involves converting text and text formatting commands into a graphical bit image to be printed. Separate bit-image planes are created for each of the four ink colors: black, cyan, magenta, and yellow.

The swath cutting step involves cutting the bit image into print-cartridge-high swaths, performing image enhancements such as overlapping print sweeps, and adjusting the bit-image planes to the particular format required by the print cartridges used in the printers.

In general, not only does the traditional HP DeskJet firmware consist of more modules but the modules themselves are considerably more complex than with the new Printing Performance Architecture.

PPA Firmware Architecture Overview

The primary goal of the Printing Performance Architecture, or PPA, is to reduce the price of an HP DeskJet printer while maintaining or increasing print performance. The digital electronics portion of this savings is accomplished by reducing ROM, RAM, and microprocessor costs. ROM is reduced by moving the rasterization, font, and swath module functions onto the host's printer driver, and by using streamlined I/O and command language protocols. RAM is reduced by requiring only enough RAM for a worse-case print sweep plus spare RAM for firmware overhead. Microprocessor costs are held down by reducing the processing, in particular the data processing, required of the microprocessor. The "don't touch the dots" concept enabled the use of a low-cost 68000 processor.

Fig. 2 shows the firmware architecture of the HP DeskJet 820C. It consists of a small set of communicating modules. Each module is implemented with a few communicating processes and interrupt service routines. The processes communicate through the use of messages (see below).

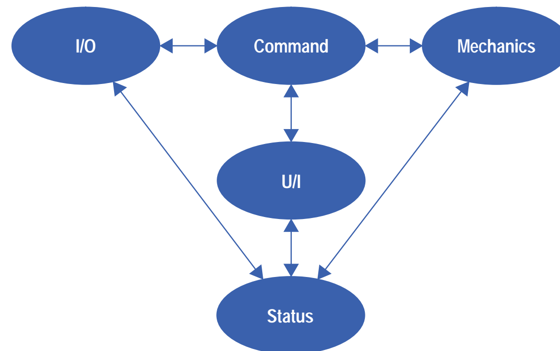


Fig. 2. *HP DeskJet 820C printer firmware architecture overview.*

The I/O module receives data and commands from the host PC, passing them on to the command module, and transmits responses and status information back to the PC. The command module parses and prioritizes the incoming commands and passes them on to the other modules, most often the mechanism module, for execution. The mechanism module receives paper load and eject, print sweep, and print cartridge servicing commands, performs the requested actions by controlling the motors and print cartridges, and passes the results back to the command module. The U/I (user interface) module handles the front-panel state machine, sending commands to the command module as necessary. The status module monitors the printer's status, communicates this status back to the PC via the I/O module, and keeps the rest of the modules informed of system status.

Processes, Messages, and Operating System

A small and efficient custom operating system manages the execution of multiple processes and the delivery of messages from one process to another. The operating system also provides support for interrupt service routines, delayed procedure calls, and binary semaphores.

Processes. Multiple, cooperating independent threads of execution called *processes* are used to provide priority, modularity, and parallelism within the PPA firmware architecture. Individual processes are instantiated with a function stack, a fixed priority of execution, and a specific set of broadcast classes. The highest priority ready process executes until either a higher-priority process becomes ready to execute, the current process is blocked waiting for a new message, or the process is blocked waiting for a semaphore to be unlocked. The process's broadcast classes indicate which set of broadcast messages the process wants to receive. Processes are static and never terminate.

A fundamental architectural concept is that there is a one-to-one correspondence between a process and a message queue. In other words, each and every process has its own queue for messages and no other queue. This concept is hardwired into the system. There are no facilities for the creation or use of any other message queues. When a process requests a message, its context defines which queue is selected.

The PPA firmware design is rather liberal with the use of processes to both modularize and parallelize functionality. Table I shows the eighteen processes used in the HP DeskJet 820C printer.

Table I
Firmware Processes in the HP DeskJet 820C Printer

Major Firmware Module					
I/O	**Command**	**Mechanism**	**Status**	**U/I**	**Other**
IO	Parser	Mechanism State Machine	Autostatus	UI	PState
IEEE 1284	Pacer	Walker/Dispatcher	Status Request		Configuration
VLink Pacing	Executer				NV RAM
					Execute Data
					Test Print
					Simple

Messages. Messages form the fundamental communication method between processes. Physically, messages are fixed-size, small blocks of memory. They contain both required and optional fields.

The typical life of a message is as follows. A process acquires an uninitialized message from the operating system. The process fills the necessary message fields. The message is posted to another process with a specific priority. The receiving process gets the message and performs the action implied by the message's identity. Depending on flags set within the message, a response message may be posted back to the originator or the message may be released back to the operating system for reuse.

The reception of messages can be gated by a priority or limited by a timeout or both. Messages can be posted to an individual process or broadcast to many processes. The posting of a message can be deferred for a specific time to provide for periodic actions. Interrupt service routines can only post messages, so arrangements must be made to acquire their messages outside of interrupt execution.

Table II shows the message structure. Messages include a token field, which gives the message an identity or specific meaning. For example, a command module process requests raw input data by posting to an I/O module process the RECV_REQUEST message (a message with its token set to RECV_REQUEST). A response field indicates which process is to be posted the result of the message. For example, when processing the RECV_REQUEST message, the I/O module process will post a response back to the process mentioned in the response field. A data pointer field, a size field, and a recover field associate a block of memory with a message. The recover field indicates which process is to be notified to recover the memory block when it is no longer needed. The use of associated data in this manner allows the firmware to pass data blocks from process to process and let the final process recover the data properly.

Semaphores. Semaphores provide a mechanism to restrict access to a shared resource (often global variables) to one process at a time. They are analogous to a lock on a door. Semaphores can be instantiated, locked, and unlocked. There are only a few critical uses of semaphores in the system. One is for the exclusive use of global configuration data. Another is for the exclusive use of the general-purpose memory pool.

Delayed Procedure Calls. Individual functions can be executed at a later time via the operating system. The operating system maintains a list of functions to be executed and at the appropriate time will execute the functions at a low interrupt level. Processes can take advantage of this feature to execute critical code at a higher-priority interrupt level. Interrupt service routines can take advantage of this feature to execute noncritical code at a lower interrupt level. Since a list of functions is maintained by the operating system, delayed procedure calls can be canceled. The user interface module uses deferred procedure calls to implement key debouncing. The deferred post feature of message posting is implemented by using deferred procedure calls.

Interrupt Service Routines. Interrupt routines are statically installed. In practice, interrupt routines often just post a message to wake up a process. For more sophisticated needs, interrupt routines can logically suspend until a subsequent interrupt. This facilitates designing serial and sequential interrupt state machines.

Memory Management. Memory management is strictly static with few exceptions. The operating system does not provide any sort of functionality to allocate or free memory. The reliability of the system was greatly enhanced by designing it for static memory use. The I/O module does provide for the use of its output ring buffer for general-purpose, restricted memory allocation with function calls such as Ring_Request() and Ring_Recover(). The restrictions were imposed for simplicity and because of the ring nature of the buffer: memory must be allocated in multiples of 4 bytes, memory must be held for a very short time or the efficiency of the output buffer will degrade, and although memory can be returned piecemeal, the pieces must start on 4-byte boundaries and be multiples of 4 bytes.

Table II
Message Structure

Message Field	Size	Description
Token	16 bits	Message identity. For example, RECV_REQUEST indicates this message is a request to receive data.
Sender	32 bits	Sending process's identity.
Response	32 bits	Identity of the process to receive the response to the message.
Data Pointer	32 bits	Pointer to an associated data block. For example, this could point to a block of input data for a RECV message.
Data Size	32 bits	Number of bytes of data associated with the message.
Recover	32 bits	Identity of the process to recover the associated data.
Flag	8 bits	Indicates whether the message must be responded to or data must be recovered. If a response message, indicates if failure, OK, or an unknown message type.
Misc 1	32 bits	Message-specific information.
Misc 2	32 bits	Message-specific information.

Firms (Soft Constants). A *firm* is a concept added to the firmware design to facilitate adjusting constants postrelease. Constants that may need adjustment after the printer has been released for manufacture are grouped together in lists. Access to these constants is via the Firm() function call. Firm() is called with a list and a constant identifier. Firm() looks up and returns the desired constant. Firm() also quickly scans a small constant replacement list. This replacement list includes the original list and constant identifier along with a new value for the constant. If a replacement exists, Firm() returns the replacement. The constant replacement list is stored in nonvolatile memory. Generally this would occur as a final step in the manufacturing process.

I/O Module

Fig. 3 shows how the I/O module is structured into physical, link, and application layers. The physical layer deals with the signaling on the parallel cable. The link layer deals with logically dividing a single cable into multiple logical channels. The application layer deals with the various data, command, status, and pacing applications necessary to implement the printer features.

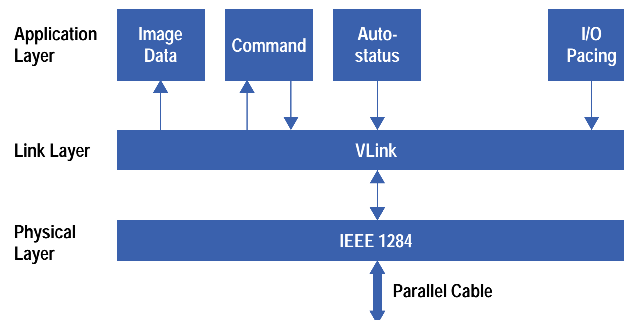


***Fig. 3.** Layered I/O structure.*

Physical Layer—IEEE 1284 Parallel Port. The connector on the back of the HP DeskJet 820C printer connects to the parallel printer port on the PC. The IEEE 1284 bidirectional parallel port specification is supported by dedicated hardware and firmware. Hardware performs the basic data transfer of bytes from the PC directly into RAM. Firmware supports the IEEE 1284 overhead required to put the port in the proper transfer modes and to transfer data back to the PC.

IEEE 1284 redefines the traditional parallel port lines BUSY, NFAULT, PERR, and so on to permit faster data transmission and to allow data to be sent back to the PC from the printer. Faster data rates are achieved by having the host only pulsing the NSTROBE line until the printer raises its BUSY line. Traditionally the NSTROBE line had to be held down for a set minimum time period (which was a relatively long time).

The IEEE 1284 overhead for mode switching is implemented as a separate firmware process in the system. To achieve the IEEE 1284-required 35-ms response time, the process runs at the highest priority in the system. The process monitors the parallel port lines and responds to changes by maneuvering through a constant state table. This state table includes information on what to watch for on the parallel lines, how to respond on the parallel lines, how to get and retrieve data at the appropriate times, and which states can be expected next.

Link Layer—VLink. The link layer provides a simple logical channel protocol. To prevent the printer's input buffer from completely filling up and preventing communication with the PC, image data and command data are separated into two logical channels. Each of these two logical channels is individually paced to prevent one from blocking the other. To separate the data and commands into two logical channels, the raw bytes are packetized so that a channel number can be assigned to each packet.

A new HP-proprietary link-level protocol, VLink, replaces the more sophisticated MLC protocol used in the other DeskJet and LaserJet models. VLink requires considerably less code, can be substantially implemented in hardware, and doesn't require a bidirectional link.

Fig. 4 shows the VLink packets. To packetize the data, VLink adds four additional header bytes to each block of data. First, a start-of-packet character, $, is sent. Second, one byte specifying the channel number is sent. Third, a word is sent indicating the number of data bytes to follow. A packet can contain up to 64K bytes of data. Custom I/O hardware strips off the four header bytes, uses the channel number to select a ring buffer in RAM in which to store the data, and subsequently tranfers the data into the ring buffer by DMA.

***Fig. 4.** VLink packet traveling on the physical cable.*

Table III shows how channels are allocated in the DeskJet 820C. Incoming packets arrive for either channel 0 or channel 1. Channel 0 is used for image data. Channel 1 is used for commands. Outgoing packets are transmitted using channels 1, 2, and 128. Outgoing channel 1 is used for responses to commands. Outgoing channel 2 is used for the periodic autostatus information. Outgoing channel 128 is used to supply pacing information to the host PC.

Ring Buffers. Two ring buffers store the two incoming data streams from the host PC. One stores the image data arriving on VLink channel 0. The other stores commands arriving on VLink channel 1. The ring buffers are implemented with a combination of custom hardware and firmware.

Table III
VLink Channel Uses

Use	Input Channel	Output Channel
Image Data	0	
Commands and Responses	1	1
Periodic Autostatus		2
Periodic Ring Buffer Pacing		128

Fig. 5 shows a diagram of a single ring buffer. The custom ASIC selects the ring buffer in which to deposit incoming data based upon the channel number in the VLink header. Incoming data bytes are placed into the byte pointed to by the ring's fill register, and the fill register is incremented. If the fill register passes the high wrap register, the fill register is set equal to the low wrap register. Once the fill register equals the recover register, no more input is permitted. Any further input for this ring buffer will cause the parallel port's BUSY line to be set high and remain high until the recover register is changed.

For the command ring buffer, the grant register (a firmware-only register not in the custom ASIC) is used to mark the data that has been granted to the parser. When the command associated with the data has completed executing, its data is recovered by advancing the recover register. This permits further data input.

For the image ring buffer, the ASIC advances the recover register as it pulls data out for the print cartridges. This occurs while the print sweep is taking place. This permits further input to occur on the image channel in case the buffer was previously full. The grant register does not exist for the image buffer.

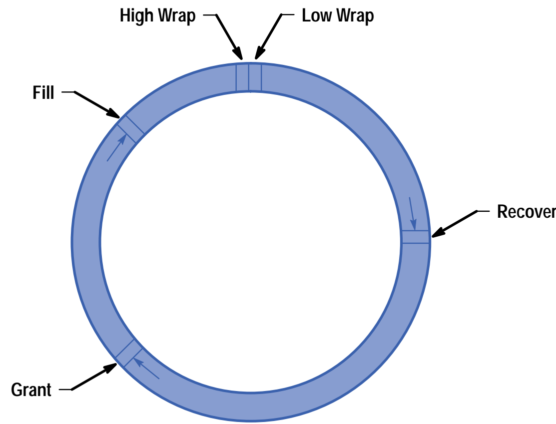


Fig. 5. *I/O ring buffer.*

A third ring buffer, one that is implemented entirely in firmware and has no custom ASIC registers, is used for an output buffer and occasionally general-purpose memory allocation. The same ring buffer design is used, thereby reusing the ring buffer utility functions. In this case, memory is granted to a process, advancing the grant register. Eventually the memory will be recovered, advancing the recover pointer. For this output buffer, the input register has no use.

An enhancement useful for both the command and general-purpose output ring buffers is the ability to recover blocks of memory out of order. This is facilitated by managing subrecovered blocks of memory between the grant pointer and the recover pointer. With the rule that all memory requests and recoveries must be restricted to multiples of 4 bytes, subrecovered blocks can be implemented using only the RAM contained within the recovered blocks themselves.

Output. For output, the main control I/O process receives SEND messages from the other processes within the system. Like input, output is formatted as VLink packets. Three VLink channels are used: channel 1 to transmit command responses back to the host, channel 2 to transmit periodic autostatus back to the host, and channel 128 to transmit I/O buffer pacing information back to the host.

The design handles cases when bidirectional I/O is not available. This can happen when the printer driver is busy and not communicating with the parallel port, when the driver is not running at all, when an external device using non-IEEE-1284-compliant cables is between the printer and the host, when the PC does not support IEEE 1284, or when there exist miscellaneous hardware and software conflicts with the parallel port.

In cases where bidirectional I/O is not available, output is prevented from accumulating inside the printer by buffering at most one packet per VLink channel. Any previous packets are automatically recovered back into the system and never transmitted. This priority scheme ensures that the host PC always receives the latest status. The only repercussion for bidirectional systems is that the driver cannot send multiple queries to the printer without waiting for each individual response.

Image Data. A key and early concept of PPA is that data arriving at the printer will already be formatted for the custom ASIC hardware controlling the print cartridges. In other words, the firmware and microprocessor in the printer do not process the data, nor do they move the data in RAM. The image data is transferred by DMA into the image ring buffer from the ASIC I/O block and from the ring buffer to the print control ASIC blocks.

Autostatus. To keep the host PC informed of the status of the printer, an autostatus process periodically formats a data block with the printer's current status. This data block is then given to the I/O module for transmission back to the host on VLink channel 2.

I/O Pacing. When one of the input ring buffers fills up completely and another byte arrives for this full ring buffer, the overflowing byte causes the parallel port's BUSY line to raise and hold off the host PC from transmitting any further data. Such a situation could prevent the host PC from querying the printer's status or canceling a print job, so the printer and host work together to prevent either of the input ring buffers from completely filling up, thus allowing the other ring buffer to continue to receive data.

The printer transmits back to the host periodic ring buffer status information on VLink channel 128. The data transmitted indicates both the instantaneous free space available in each buffer and the amount of data recovered from the ring buffers. The amount of data recovered from the ring buffers is cumulative. In other words, the printer reports the total number of bytes it has recovered from all of the input buffers.

This total number of recovered bytes permits the host PC to determine exactly how much space is available at any time in the printer's input buffers, as long as it keeps track of how many bytes it has itself transmitted. This mechanism is required because the printer's report of the free space available in the input buffer is only an instantaneous reading. It doesn't account for any data in transition and could thus give the host PC a false reading.

Command Module

The command module is responsible for parsing and executing SCP (Sleek Command Protocol) commands. SCP provides the command protocol for communication between a PPA printer and its host driver. SCP is a binary language (as opposed to the ASCII formatting of the traditional PCL command language). The general command syntax is shown in Table IV. Some SCP commands are shown in Table V.

Table IV
SCP Command Structure

Command Field	Field Size	Description
ID	2 bytes	Identifies the command
Reference	2 bytes	Reference number used to cancel commands
Priority	1 byte	Order in which the command is processed
Pad	1 byte	Unused
Length	2 bytes	Number of additional data bytes
Data	0 to n bytes	Depending on command, typically contains a number of subfields

Table V
Examples of SCP Commands

Command	Description
PRINT_SWEEP	Configure hardware to print a sweep of data
HANDLE_MEDIA	Load and eject
HANDLE_PRINT_CARTRIDGE	Print cartridge change, wipe, spit, etc.
CONFIGURE_PRINT_CARTRIDGE	Print cartridge temperatures
STATUS_REQUEST/REPORT	Synchronous status information
CONFIGURE_AUTOSTATUS	Asynchronous status information
CANCEL_COMMAND/DATA	Flush a command or image data
RESTART	Reboot printer
ECHO_DATA, PERFORM_TEST, SET_ALIGNMENT_INFORMATION	Miscellaneous functions
UI_STATE, UI_MONITOR	User interface set and read
ATOMIC_COMMAND	Low-level manufacturing and test command

Command Parsing. The Parser process requests raw data bytes from the I/O module by sending a message. Command boundaries are identified, blocked, and attached to an acquired message. Each SCP command is attached to one message. This message identifies and leads the command through the system for execution. The individual messages are at first posted to the Pacer process.

Command Pacing. The Pacer process receives messages pointing to raw SCP command bytes and sorts them according to priority. It continuously selects the highest-priority command and posts that command to the appropriate module for execution (which could be I/O, mechanism, etc.) The Pacer then waits for the command to complete by waiting for a response message from the selected executor.

Commands are sent to the Pacer not only by the Parser but also by other modules that may want a command executed. For example, when the printer door is opened, a HANDLE_ PRINT_CARTRIDGE: Change_Print_Cartridge command is given to the Pacer for execution. This command is issued by the U/I module.

Command Execution. A third command module process, the Executor, executes SCP commands designated for the Parser. Generally, SCP commands are delegated to their respective modules for execution. A few commands, such as the CANCEL_COMMAND command, are executed by the command module itself.

Mechanism Module

The mechanism module executes the mechanism-related SCP commands, maintains the system's mechanical state, handles periodic print cartridge servicing needs, handles all motor needs and functions, and prints sweeps of data.

The mechanism module consists of two processes and several interrupt service routines. The top-level process, the Mechanism State Machine, manages the high-level mechanism state (cover open, paper loaded, etc.). The low-level process, the Walker/Dispatcher, manages the execution of mechanism motion scripts called ***flows***.

Mechanism State Machine. The Mechanism State Machine is a process that maintains the current mechanical state of the system, takes the proper actions when state changes occur, and returns to previous states after asynchronous state changes occur. Fig. 6 shows a small portion of the mechanism state machine to give an example of its hierarchical nature.

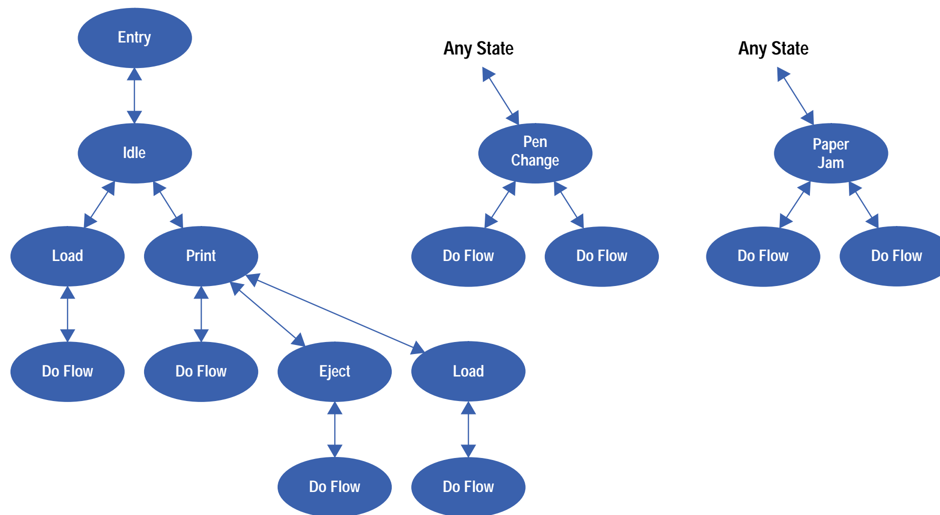


***Fig. 6.** A portion of the mechanism module's hierarchical state machine.*

The mechanism starts in the Entry state and after initialization proceeds to an Idle state. As a print job comes in, a HANDLE_MEDIA: Load_Paper command causes an entry into the Load state, and when paper is loaded, to a Ready to Print state. During these states and changes, mechanism flows (or scripts) are performed and the state machine responds to asynchronous events such as a print cartridge change or paper jam. When responding to asynchronous events, an asynchronous state change is made and the appropriate flows are performed. The state then reverts back to the state that existed when the asynchronous event occurred.

Mechanism Flows. Mechanism flows are small lists of individual mechanism instructions, typically motor moves, to complete a high-level mechanical task. For instance, when starting the Load Paper state, a load paper flow is executed. This flow contains a list of individual motor move commands to accomplish the multimotor task of loading paper.

Flows are written in a custom scripting language. The reason for the custom scripting language is to permit development of motor motion without recompiling and building the firmware set. A flow can be downloaded to the printer and executed, permitting an easy standalone mechanism development environment. This technique is also used during manufacturing to invoke custom manufacturing motor movements. When a particular mechanism flow has been developed, the flow can be incorporated into the firmware set and executed just as during development. Table VI lists a few of the available flow commands.

Walker/Dispatcher. The flow script executor is a process called the Walker/Dispatcher. This process receives messages with either addresses of flows to execute or addresses of completion routines to execute. When given an address of a flow to execute, the Walker/Dispatcher looks up each opcode and calls the corresponding function to perform the opcode. When given a completion routine to execute, the Walker/Dispatcher executes the completion routine and then retries any opcode that had to wait for a completion before continuing. This is similar to a microprocessor retrying an instruction after a page fault is corrected.

Table VI
Partial List of Flow Scripting Commands

Flow Opcode	Parameters
Carriage Motor Move	Speed, position
Paper Motor Move	Speed, distance
Wait Carriage Motor Done	
Wait Paper Motor Done	
Jump to Sub Flow	Flow ID
Goto Flow	Flow ID
Fan	On/off
Relative Branch	Condition, branch distance
Exit Flow	

Actors and Gaffers. The functions that implement the flow opcodes have been nicknamed *actors*. There is one actor function for each flow opcode. A function table is used to select which actor function to execute for each flow opcode encountered.

An actor function parses the flow opcode's parameters, verifies that the particular mechanism resource isn't in use (generally a motor), and makes the appropriate call to the motor control code to start the proper motor movement. If a resource is in use preventing the actor from continuing execution, execution of the actor terminates and is retried when a resource becomes free.

A completion routine is passed to the motor control code to be executed when the motor has completed motion. These completion routines have been nicknamed *gaffers*. They deal with errors during motion, do any final cleanup, and cause the script executor to retry an actor function that couldn't execute because of a resource limitation. Gaffers aren't executed by the motor control code, but rather are posted to the Walker/Dispatcher for execution.

Motor Control. Motor control is accomplished via a combination of process and interrupt threads of execution. Generally the execution that occurs in process space would include all initial motion and interrupt setup calculations. A transition is made to the interrupt space of a selected hardware interrupt with a call to Interrupt_Context(). Once in interrupt space, calls can be made to Wait_For_Interrupt() to effectively suspend the execution until the associated interrupt occurs again. Execution continues, including any additional suspensions for additional interrupts, until time to inform the Walker/Dispatcher flow executor of completion. A message is posted to the Walker/Dispatcher with the address of the appropriate completion routine, the gaffer.

An example of a motor control function using such a combination is CM_Move_And_Hold(), which moves the carriage motor to a specific location, holds there, and posts the given completion routine. CM_Move_And_Hold() is called with a motion acceleration profile, a final position, a few other motor adjustments, and a pointer to a completion message. The function does some preprocessing to account for previous motor motion errors, to calculate the direction and distance to travel, and to select acceleration and slew parameters. The transition to interrupt space is made. The function then goes through three loops: one for accelerating, one for slewing, and one for decelerating. Each loop calls Wait_For_Interrupt() and sets up the next incremental motion request. Finally, at the completion of the motion, the function posts the completion message to the Walker/Dispatcher process.

Configuration RAM

The HP DeskJet 820C printer has a block of nonvolatile RAM that is used for configuring the printer in ways that must survive shutdowns. A C-language structure is used to organize this configuration data. Two small processes read and write the data from the nonvolatile RAM and control when this must be done. Examples of fields stored in nonvolatile RAM are shown in Table VII.

A copy of the nonvolatile RAM is kept in normal RAM. This copy is made upon startup by the Configuration process. Any process can access the configuration data copy as long as its access is protected by locking a semaphore. After a process has made any change, it must send a SAVE_CONFIGURATION message to the Configuration process. Configuration schedules the nonvolatile RAM update by sending a message to the NV RAM Process.

A second process actually reads and writes the nonvolatile RAM. This is to avoid holding up the system, since the physical reading and writing of nonvolatile RAM takes time. The NV RAM process executes at a very low priority so that nonvolatile RAM is only updated when there is nothing else to do in the system.

Table VII
Partial List of Configuration RAM Contents

Configuration Field	Description
Startup Tests	List of startup tests to perform
Print Cartridge Calibration	Stored print cartridge calibration figures
Page Count	Count of how many pages have been printed
Firm Replacements	Set of constant replacements
Alignment	Dual print cartridge alignment adjustment factors
Mechanism State	Indication of whether the mechanism was properly stored before shutdown

Power-On/Shutdown Sequencing

A process known as Pstate is used to facilitate a controlled startup and shutdown procedure. This is important to ensure that dependencies are handled during startup and shutdown. To accomplish this, a phased startup or shutdown is used. During startup phase 1, processes cannot assume that any other process has had a chance to execute any code. Each process initializes only its own internal data structures. During startup phase 2, processes can assume that all of the other processes have completed their phase 1 code. There is no hard and fast rule governing what is to be done at each phase. It is simply known that within a given phase, a process can assume that all other processes have completed all previous phases. Similar procedures are used in shutdown.

The startup sequence proceeds as follows. At startup, Pstate broadcasts to each process desiring startup information the START message. Processes indicate they want the startup information by belonging to the Startup class. Included within the START message is a phase number. The first time START is broadcast, the phase field is set to 1. Once each process has responded to this first START message, another START message is broadcast, this time with the phase field set to 2, and again, each process will respond. Finally, once all phases have been completed, Pstate broadcasts the message START_SEQUENCE_DONE. At this point, all processes can assume that the system is operational.

Internal Test Print

A small process is used to perform the internal test print feature. The Test Print process waits until it is handed the DO_TEST message. It then temporarily disables I/O and takes over the image input buffer, filling it up with test print data. To print, this process builds its own HANDLE_MEDIA: Load, PRINT_SWEEP, and HANDLE_MEDIA: Eject commands and sends them to the command Pacer for execution. Finally I/O buffers are restored and I/O reenabled.

User Interface

The user interface module, U/I, is designed to respond to stimulus of various events happening in the system. A state table is used to map a stimulus to a particular action and subsequent state. Each state also includes a set of exit conditions. The process's main function is to respond to UI_EVENT messages which are posted when front-panel changes occur.

The printer cover door and buttons generate interrupts when they change. Each of these has an interrupt service routine that takes care of debouncing, using deferred procedure calls, and posts a message to the UI process indicating the event change. The UI process then marches through its state table to make the internal change to the printer and the visual change to the user.

Printer Status

Printer status is managed by the status module. This module receives update indications from the rest of the system, composes specific status responses back to the host PC, and composes periodic autostatus responses back to the host PC.

Autostatus. Table VIII shows a sample of the autostatus data. Autostatus is a fixed structure of bits and numeric fields that is transmitted back to the host on a periodic basis. The Autostatus process is responsible for building the transmitted data block and handing it over to the I/O module for transmission to the host.

Status Update. UPDATE_ITEM messages are posted to the status module to update specific fields in the autostatus block. At this time additional notification to the rest of the system can be made by the status module. For instance, the status module will notify the U/I module of paper misloads, cover door openings, missing print cartridges, and so on.

Table VIII
Examples of Autostatus Fields

Field	Description
Misload	Paper load failed—most likely out of paper
Door Open	Cover door open
Media Jam	Paper jam detected
Print Cartridge Unaligned	Dual print cartridges not properly aligned
Last Error Code	Last error encountered by the firmware

Status Responses. The host PC can also request specific information from the printer. The Status Request process receives status request commands from the Command Pacer module, formats the result, and again hands the data over to the I/O module for transmission to the host.

The Simple Process

There are small functions or commands that must be executed that don't really fit logically into any of the modules in the system. Logically for modularity reasons, they might each form their own module or process. But in an effort to conserve ROM and RAM, these functions have been combined into a single process. Table IX shows a partial listing of the functions handled by the Simple process.

Table IX
Examples of Simple Process Functions and Commands

Function	Description
SCP Cmd NV RESET	Reinitializes nonvolatile RAM to default values
SCP Cmd SET ALIGN INFO	Stores the print cartridge alignment information received from the host
SCP Cmd SET PAGE COUNT	Stores a new value for the page counter
SCP Cmd REPLACE FIRM	Stores a new value for the specified firm
Calibration Functionality	Performs periodic calibration functions

Flash Memory Support

To provide for firmware upgrades during development and the early stages of manufacturing, flash memory is temporarily substituted for ROM. The flash memory can be reprogrammed whenever a new firmware set is available.

The SCP command language provides a command, EXECUTE DATA, which causes the firmware to jump to data downloaded into the image buffer. Before making this jump, the printer shuts down all interrupts to guarantee that none of the existing firmware is still executing. To reprogram the flash memory the downloaded program contains both the code to reprogram the flash memory and the data to program into the flash memory. When this downloaded program has completed reprogramming the flash memory, it executes a 68000 reset instruction, effectively returning control back to the flash memory and beginning execution of the newly installed firmware.

Conclusion

The HP DeskJet 820C printer firmware architecture successfully met or exceeded all cost, quality, schedule, and throughput goals. This is particularly satisfying considering the firmware platform started completely from scratch, with design leverage only in the mechanism flow scripting arena.

Acknowledgments

Special acknowledgments are in order for the HP DeskJet 820C firmware team. David Neff and Eric Ahlvin managed the team at different times. Gene Welborn started on the architecture early and had a key role in leading the concept of using message communication processes and designing the operating system interface. Mark Garboden led the design and implementation of the Walker/Dispatcher along with its associated flow scripts for motion control. John Van Boxtel designed and implemented the operating system and the low-level motor control code. Rose Elley had the critical role of providing engineering support and tools to enable R&D, customer assurance, and manufacturing to move towards the new PPA architecture. Melinda Grant did user interface design. Carl Thompsen, Bob Callaway, and Hugh Rice did significant design, consulting, and coding in the areas of print cartridge, motor, and analog functions.

PPA Printer Controller ASIC Development

As the first Printing Performance Architecture printer, the HP DeskJet 820C needed a completely new digital controller ASIC design. The chip's architecture was optimized for the specific requirements of PPA. Concurrent development of hardware and firmware through the use of hardware emulators and attention to regulatory issues during the design helped the product meet all of its requirements on schedule.

by John L. McWilliams, Leann M. MacMillan, Bimal Pathak, and Harlan A. Talley

The Printing Performance Architecture (PPA) used in the HP DeskJet 820C printer is a significant step forward from any previous HP inkjet printer product in providing the consumer with a high-performance product at an excellent price point. Since PPA redistributes the printing tasks between the host and the printer, a complete redesign of the digital controller ASIC in the printer was required. This redesign effort took into account the overall product constraints of cost and time to market as well as all applicable government regulations. The result is a highly integrated ASIC that implements all digital functions performed by the HP DeskJet 820C on a single chip. This high level of integration significantly decreased the cost of the electronics in the HP DeskJet 820C compared to the previous-generation product while maintaining the printer's performance. This article describes the system considerations, engineering decision trade-offs, and development methodologies that played a role in the development of the digital controller ASIC for the HP DeskJet 820C.

The design of the controller ASIC had to be done under numerous constraints. As in any consumer-oriented product, the foremost consideration during design was the final cost to the buyer. The Performance Printer Architecture, as described in ***Article 1***, was developed to reduce the total cost of the printer. PPA allows several optimizations in the digital architecture. In today's competitive environment, time to market is nearly as critical a constraint as cost. Meeting the time-to-market constraint required concurrent development of hardware and firmware and a bug-free ASIC at netlist release. These needs were addressed by using hardware emulators during development. Finally, the printer had to meet or exceed all government regulations including those pertaining to EMI and ESD. Taking these needs into account during the ASIC design helped the product pass all requirements on schedule.

Digital Architecture

Regardless of their specific type, all printers require several pieces of digital hardware. These pieces include a microprocessor to control the printer, RAM for data, ROM for firmware, and custom digital logic for printer-specific functions. By optimizing each of these pieces, significant cost savings were realized in the digital ASIC.

PPA significantly reduces the cost of the printer by optimally partitioning the printing tasks between the software running on the host and the hardware and firmware running in the printer. The partitioning is done without sacrificing the printer's performance. All tasks that can be done on the host computer without severely affecting application performance are done in the driver. Tasks with real-time constraints are performed by the hardware and firmware in the printer. Because the host performs the majority of the data manipulation, data that is sent to the printer is in a format that is very close to the final form used to fire the printheads. Because of this, the digital architecture was designed with a guiding principle of "the processor does not touch the data." Once this principle was adopted, the ASIC team was able to make several important design decisions.

First, a relatively low-power processor is all that is needed, since the processor does not manipulate the data. After surveying the available microprocessors, the 16-MHz version of the Motorola 68EC000 was chosen as the best fit. Second, since the number of tasks the firmware performs is limited, the code size can be kept small enough that a ROM with all firmware can be integrated on the ASIC, eliminating the need for an external flash memory or ROM. Third, all data manipulations need to be done in hardware, which limits those manipulations to being relatively simple. Finally, the memory requirements are limited—a 1M-bit DRAM is sufficient for the data needs. The DRAM holds all firmware variables and stacks as well as all printing data. Even with the DRAM doing double duty, the memory bandwidth requirements of the architecture are fairly low, and the product is able to use a low-cost, 1M-bit, nibble-wide DRAM.

A block diagram of the HP DeskJet 820C's digital architecture is shown in Fig. 1. The digital electronics consists of three main components: the digital ASIC, a 1M-bit DRAM, and an optional external flash memory or ROM. The digital ASIC consists of a 68EC000 microprocessor, a 64K-byte ROM, a 55,000-gate standard cell block, and a 1K-byte SRAM used as a

data cache. In addition to the external memory components, the digital ASIC is connected to the I/O connector (IEEE 1284), the printer motor ASIC, the printhead ASIC, and an optical encoder which provides carriage position information.

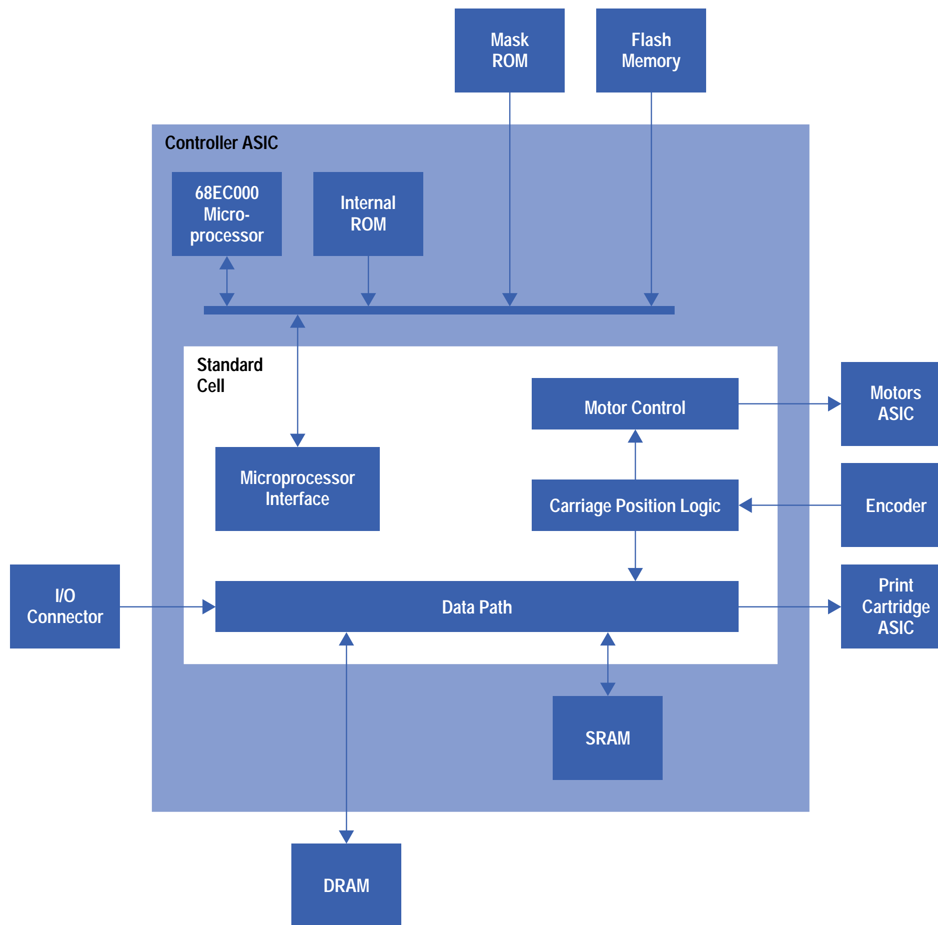


Fig. 1. *HP DeskJet 820C controller ASIC block diagram.*

The majority of the standard cell area is devoted to the data path, which is the path the data follows as it moves from the I/O connector, through the DRAM and SRAM, and up to the pen ASIC. The remaining logic is used for interfacing to the microprocessor, for controlling motors, and for keeping track of the current carriage position. All memories, including registers in the standard cell block, are memory mapped into the 68EC000's standard address space.

Flash or ROM

The ASIC is designed to be able to read code for the processor from one of three sources: a flash memory device, an external mask-programmable ROM (MROM), or the internal ROM. The reason for the three separate sources is to better meet time-to-market constraints. At the beginning of the manufacturing ramp, code was stored in flash memory. That way, final firmware did not need to be released until just before the start of the ramp. As soon as the firmware was stable, it was released to both the MROM vendor and the digital ASIC vendor for programming into the internal ROM. However, MROM lead times are much shorter than general-purpose ASIC lead times, so MROM parts were available much sooner than ASICs with properly programmed internal ROMs. Consequently, printers were built with MROMs for a period of time until ASICs with final firmware were available (MROMs are about half the cost of flash parts).

Motor Control

The HP DeskJet 820C has three motors: a dc motor for moving the carriage across the paper, a stepper motor for picking and advancing the paper, and a second stepper motor for controlling the pen service station. The stepper motors are controlled in an open-loop process by the firmware. The firmware controls a stepper motor move by writing appropriate phase and pulse width data to registers in the ASIC. Hardware then generates the appropriate signals for the motors. The phase and pulse width data determines the direction and speed of the moves.

The carriage motor is controlled by a firmware-based control loop that monitors the carriage position and adjusts the motor control signals appropriately. The carriage position is determined through the use of an optical encoder. The optical encoder consists of a light emitter-detector pair with a plastic encoder strip between them. As the carriage moves across the paper,

the light emitter-detector pair senses that it is moving along the plastic strip, and sends some signals to the ASIC. The hardware in the ASIC takes this information and uses it to keep track of the current carriage position. Using the carriage position, the firmware tracks the carriage's speed and acceleration and adjusts the motor energy appropriately.

PPA I/O Packet Format

The data from the host comes to the printer in a simple packetized format. As shown in Fig. 2, the packets are made up of two pieces: header information and data. The header information consists of a start-of-packet (SOP) byte, a channel byte, and a two-byte data-size field that reflects the number of bytes in the data field (0 to 65K).

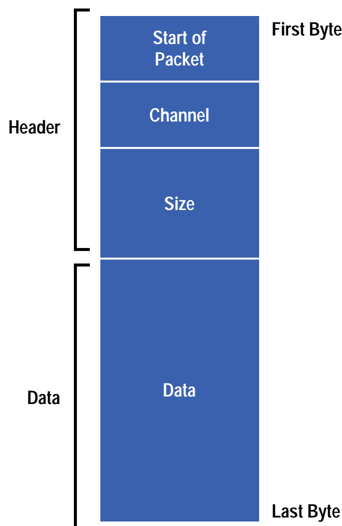


Fig. 2. *PPA data packet format.*

In the HP DeskJet 820C, packets from the host may contain one of two types of information: command data or image data. The channel byte determines which type of data is contained in the packet (hence, in the HP DeskJet 820C, the channel byte will be one of only two distinct values). Command data contains PPA printer control commands, while image data contains information that is to be printed on a page. The image data that is sent to the printer is in a form that resembles a bitmap of the image, and therefore requires a minimum amount of reformatting before being used to fire the printheads. To minimize the amount of data that must be sent over the I/O cable, image data is optionally compressed before being sent to the printer.

Data Path

A block diagram of the data path is shown in Fig. 3. Data enters the ASIC through the I/O cable. Hardware depacketizes it and separates it into the image and command channels. Image data is transferred to one buffer in the DRAM and command data to another, both by DMA. Command data is consumed by the firmware with no hardware interference. Image data is moved by the *servant* hardware from the DRAM to the SRAM. During the move, the image data is decompressed if necessary. From the SRAM, data is moved to a shift register from which it is serially shifted up to the carriage board and is used to fire the pens.

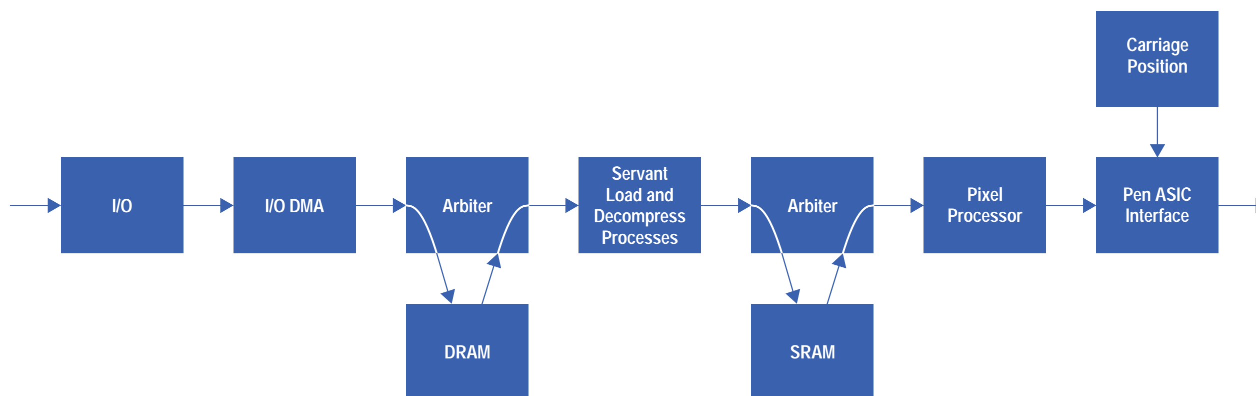


Fig. 3. *Data path.*

Input/Output

The standard cell I/O block implements the low-level hardware that takes in packets of information from the host via the parallel port. It contains hardware support for IEEE 1284 compatibility mode and extended capability port (ECP) in the forward direction from the host to the printer. The hardware also supports, with firmware assist, reverse-channel nibble mode for sending information back to the host computer. The I/O block also contains hardware that strips the data stream of its packet header information, separates the packets into command and image data, and sends them to the I/O DMA block. From the header information, the hardware checks the start-of-packet byte to make sure it is the correct value, uses the channel byte to select the appropriate DMA channel, and uses the size field to determine when to expect a new packet.

The I/O DMA block receives data via the I/O interface and stores it into either the command buffer or the image buffer in the DRAM. These buffers are designed as general-purpose circular buffers that can reside anywhere in the DRAM memory space. The command buffer is emptied by the processor as it executes the commands. Image data is consumed by the servant hardware. As data from the buffers is consumed, the host is notified, via the firmware, of the available buffer space, and more data is sent down. This architecture allows the printer to make optimal use of its limited memory resources.

DRAM Controller/Arbiter

The external DRAM is connected to its own nibble-wide bus. Hardware arbitrates accesses to the DRAM between the I/O DMA hardware, the servant hardware, and the microprocessor. The arbitration method is a combination of priority and round-robin schemes. Both the I/O and the servant hardware processes have real-time constraints that dictate the maximum length of time they can be blocked while waiting for access to the DRAM. Although the microprocessor is less tightly constrained, it is important that it not be completely locked out of the DRAM for extended periods of time. Hence, while each block has a priority for DRAM accesses, the hardware is designed so that no one unit can hold the DRAM bus continuously.

In addition to arbitration, the DRAM controller takes care of the low-level interface to the DRAM. It interfaces the 4-bit DRAM data bus to the 8-bit microprocessor data bus. Using fast page mode accesses, it retrieves two nibbles and concatenates them into a byte. It guarantees that the DRAM refreshes take place at appropriate times. Although only a 1M-bit part is used in the HP DeskJet 820C, the controller also supports 4M-bit DRAMs.

Servant

One of the key contributions of the PPA architecture is that it moves much of the pixel processing into the driver. The image data sent to the printer is in a format nearly ready to be used to fire the pens. The only significant operation that is not done by the driver is the operation of picking out the individual bits (corresponding to dots on the paper), and sending them to the pens in the correct order. The servant logic, so named because it serves the pen by providing it with pixel data, accomplishes this by loading the data into an on-chip cache (the SRAM), and subsequently pulling it out at the correct time and in the correct order. The cache is divided into sets of swing buffer pairs (one pair for each color) such that while data is being taken out of one swing buffer by the pixel processor logic, new data is being loaded into the other swing buffer by the servant load logic. When the pixel processor consumes all the data in one swing buffer and switches to the other swing buffer, the servant load process begins loading new data into the first buffer. This process is depicted in Fig. 4.

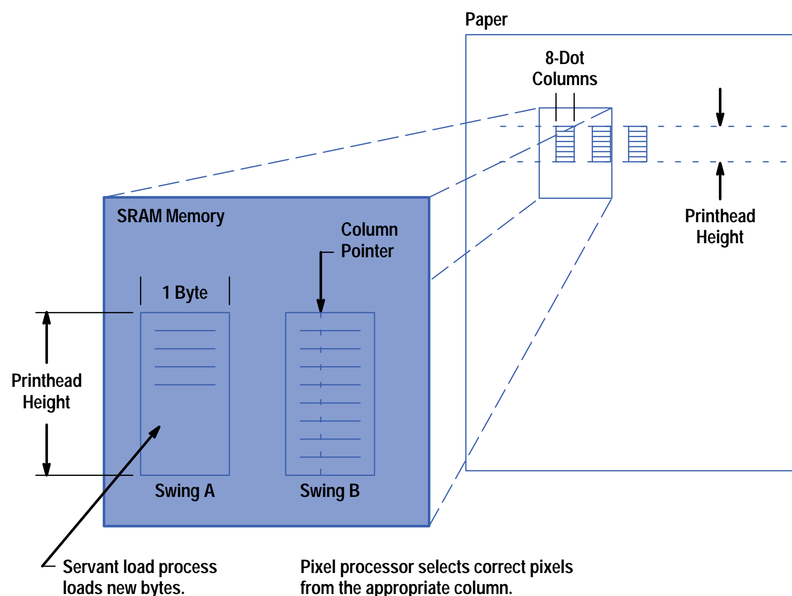


***Fig. 4.** Swing buffer operation.*

The PPA driver provides the pixel data in swing buffer loads, which are chunks of a bitmap eight pixels wide and the same height as the printer's pens. The driver provides the swing buffer loads in exactly the order required by the pens. The servant load process transfers the data by DMA from the DRAM to the SRAM as it is needed to fire the pens. During the process of moving data from the DRAM to the SRAM, the data is decompressed if it was sent over the I/O in a compressed format.

SRAM Arbiter

The SRAM arbiter arbitrates memory requests between the servant load process, the pixel processor, and the microprocessor. The arbiter implements a priority-based scheme. Since the microprocessor accesses the SRAM only infrequently, it is given the lowest priority. On the other hand, data for the pen must be immediately available on demand (the carriage cannot be paused for the pen to wait for data). Hence, the pixel processor is given the highest priority. The servant load process has the middle priority.

Pixel Processor

The pixel processor is responsible for placing the bits sent to the pens in exactly the order in which they are needed to fire the pens. Since the nozzles on the pen are staggered, the order in which the bits are needed is not entirely straightforward. As the correct bits are pulled out of SRAM, they are placed in a shift register from which they are serially shifted to the pen driver IC on the carriage board.

Each bit sent to the pens corresponds to a nozzle firing or not firing. Firing data sits in the SRAM in byte-wide chunks. Each byte corresponds to eight columns of dots on the printed page. All dots in a column are fired before beginning to fire the next column. Hence each byte in a swing buffer is accessed eight times, once for each column. After all eight columns are fired, the logic switches to the other swing buffer.

Pen Interface

This block communicates with the analog pen driver IC over a custom serial interface. The pen interface receives data from the pixel processing block and shifts it serially to the pen driver IC. It also generates the timing pulses that the pen driver IC uses to fire the pens and put ink dots on the page. In addition to sending pen firing data, the interface sends setup information to the pen driver IC to adjust various printing parameters that affect print quality. The serial interface is bidirectional, enabling the pen driver IC to send back information about the pens' status. For example, the pen drive IC is able to measure temperatures of the pens, which are important parameters in thermal inkjet printing. This information is sent to the digital IC and read by the firmware. Firmware then uses this information to adjust printing to ensure that the customer will receive optimum print quality.

Because of the staggering of the nozzles on the printhead (see Fig. 1 on page 11), for each column of dots on the page, the pen must be fired multiple times. The pens must be fired every time a set of vertically aligned nozzles is at the correct position on the page. If all nozzles in a dot column are not fired at the same physical position on the page, that column will appear jagged to the customer. Special logic in the chip ensures the proper alignment of the dots and therefore optimum print quality. This logic uses the carriage position as determined directly from the optical encoder, which is at a relatively low resolution, and interpolates it up to the resolution needed to fire the pens. The interpolation is done by phase-locked-loop-like logic that measures the time it takes the carriage to move 1/150 inch, and divides this time down to get the time it takes the carriage to move a distance equal to the nozzle stagger distance. By doing this, the logic is able to issue firing pulses to the pen at the correct time.

Development Methodology

The HP DeskJet 820C was developed under some very tight time-to-market constraints. These constraints dictated that the latest CAE tools be used to speed the development of the ASIC. Additionally, the project team wished to have concurrent design of the hardware and the firmware. This meant that the firmware team needed a platform on which to do development before the ASIC was finished. To meet this need, Aptix hardware emulators were used.

The HP DeskJet 820C ASIC was designed entirely in the Verilog Hardware Description Language (HDL). HDLs are computer languages used to describe digital circuits. They contain constructs that allow designers to describe the function of a circuit rather than the exact gates that are necessary to implement that function. Thus, HDLs allow designers to work at a higher level of abstraction than in the past. Once a designer has written the HDL for a circuit, a compiler program can synthesize the HDL into a gate-level design. By working at a higher level of abstraction, engineers can greatly increase their productivity. The time required to do the design of the HP DeskJet 820C ASIC was significantly decreased over past products.

Since designing an ASIC using an HDL is analogous to writing a piece of software, it is not surprising that many of the practices used by software engineers can be used successfully by hardware teams using HDLs. At the beginning of the project, coding conventions were established. Similar structures in different designers' modules were coded similarly. Designers were encouraged to comment their code liberally. Code reviews were held during the project to find errors and to improve designers' coding practices. These techniques allowed designers to look at each other's code and quickly understand it. In addition to having obvious benefits for the HP DeskJet 820C project, the good coding practices will allow the HP DeskJet 820C hardware to be easily leveraged into future products.

To synthesize the Verilog code into standard cell gates, Synopsys software was used. This software allows the designer to enter information about the design to help the software produce an optimum implementation. Synopsys-specific scripts were used to enter the required information. By using scripts, the designers were able to make changes to the code, and with minimum effort, synthesize the new implementation. Just as for the original Verilog code, conventions and templates for the scripts were developed. As a result of these techniques, in a few special cases engineers were able to modify code written by a different designer and synthesize new hardware very efficiently.

An important part of ASIC design is test development. The HP DeskJet 820C ASIC design team used an HP proprietary technology that allowed the engineers to write test vectors directly in Verilog. These test vectors were used to verify that the functionality of the synthesized design matched the original Verilog. The same vectors were then translated into a format the ASIC tester understood, and used to test the finished silicon. Using this technique, a single set of test vectors was used throughout the project. In addition to the functional test vectors, scan testing was used to achieve the desired fault coverage. Since the insertion of scan hardware and the creation of scan test vectors is done semiautomatically, scan testing was successfully added to the ASIC without incurring a schedule delay. The use of scan testing did increase the cost of the chip because the scan circuitry caused the chip size to grow, but this was deemed acceptable when traded off against the time it would have taken the designers to write functional test vectors with adequate coverage.

Hardware/Software Codesign

To meet the overall project goal of a low-cost product, it was necessary to make careful trade-offs between the hardware and firmware in the product. Functions that are realized in hardware cost money because silicon real estate is used. Functions realized in firmware cost money because they require bits in memory. Since the ROM that holds the firmware in the HP DeskJet 820C is integrated into the system ASIC, its size had a hard upper limit. Also, the HP DeskJet 820C uses a relatively low-power processor, so the processing bandwidth available to perform functions in real time is limited. With the standard cell logic, the processor, and the firmware ROM all integrated onto the same chip, optimal trade-offs between the three were especially important.

To make the correct trade-offs, the ASIC engineers responsible for particular hardware blocks coordinated closely with the firmware engineers responsible for the corresponding firmware blocks. This process allowed the hardware engineers to gain insight into how the firmware would use the block, and at the same time allowed the firmware engineers to have a good understanding of the hardware. This mutual understanding led to better trade-offs. The hardware was designed with just enough functionality to allow the firmware designer to implement the code within the product code size and processor bandwidth constraints, but without a lot of extra hardware thrown in “just in case.” A secondary benefit of this approach was that the firmware engineers were able to write the code for blocks designed this way with few problems. Code for blocks not designed using this process (primarily blocks leveraged from previous products) proved much more problematic to bring up.

ASIC Emulation

To meet the aggressive schedule, it was necessary for the firmware team to begin implementing the code well before ASICs were available to run the code. In fact, firmware implementation began before the ASIC design was even complete. To allow this activity to take place, it was necessary to set up an emulation environment. Traditional methods of doing such emulation include building a custom printed circuit board populated with one-time-programmable devices (anti-fuse devices, laser programmed parts, etc.) and software-based emulation using a previous product. Since the digital architecture for this ASIC was a significant departure from any previous product, emulation on a previous product would have been difficult at best and would have required a significant code port when the ASIC became available. One-time-programmable devices were not flexible enough to support emulation before the design was functionally complete. For these reasons, the product team chose to use full-chip IC emulators from Aptix to support firmware development before ASICs were available.

IC emulators are essentially a large array of SRAM-based FPGAs (field programmable gate arrays) with programmable interconnect between them. Software reads in a gate-level design for an IC, partitions it into the FPGAs, and then creates all the necessary files to program the FPGAs and the interconnect between them. The emulator then is functionally equivalent to the IC that will be fabricated. The emulators are highly flexible since they can be reprogrammed by simply downloading a new pattern into the SRAMs. The main drawback of the IC emulators is that they generally are not able to run at the same speed as the final silicon. For this product, the emulator ran at one fourth of the final clock speed.

Using this approach, the IC team was able to provide the firmware team with usable hardware approximately four months before silicon arrived. Since the ASIC design was not complete at that point, the first hardware provided was only a subset of the full standard cell logic. What was provided was enough for the firmware team to begin writing and testing the operating system, the code that needed to be written first. As more blocks in the IC were completed, they were incorporated into the emulation system.

In addition to providing early hardware to the firmware team for development purposes, the use of IC emulators allowed the hardware to be verified in the full printing system with actual firmware before being committed to silicon. Because the emulators ran at close to the system speed, several orders of magnitude more clocks cycles of verification occured on the emulators than with software simulation. Also, since it was real firmware running, the ASIC was put in states that would have been difficult or impossible to achieve in simulation because of the complexity of getting into that state. Finally, many

unanticipated hardware/firmware interactions were discovered. The team was eventually able to print with the IC emulators, giving very high confidence in the functional correctness of the ASIC.

Thanks to the emulators, two problems in the design were discovered and fixed before committing the design to silicon. Both problems were system interaction issues that would have been very difficult to discover through simulation alone. When silicon arrived, firmware was almost immediately bootable on it. The only things that needed to be changed in the firmware were things that were affected by the difference in clock speed, and these had been deliberately coded to be easy to change.

Regulatory Requirements

Because the HP DeskJet 820C printer is sold in the consumer marketplace it must meet all applicable consumer electronic regulations. Of particular interest to the electronic design of the product are the electromagnetic interference (EMI) and electrostatic discharge (ESD) requirements. EMI occurs when an electronic product creates an electric field and interferes with the correct operation of another electronic product. ESD occurs when an object (generally a human) that has built up a large static charge discharges to a second object (for example, an IC). In addition to government requirements on a product's level of EMI and sensitivity to ESD, HP maintains internal standards, which are generally tougher than the government standards. Meeting or exceeding these internal standards on every product is an important aspect of HP's reputation for high-quality, reliable products. Since the HP DeskJet 820C could not legally be released without meeting government regulations on EMI and ESD, failure to meet them was a significant schedule risk to the product. Therefore, both were addressed early during the design of the digital ASIC.

In general, EMI results from improperly controlled high-frequency signals that travel a long distance, particularly signals that travel over cables. The trick in designing for reduced EMI is to control the signals with high-frequency content to the greatest possible extent. All I/O pads in the HP DeskJet 820C make use of an HP proprietary technology that compensates for process, voltage, and temperature (PVT) variations in the operating environment of the chip. The compensation ensures that the pads have nearly the same slew rate regardless of the PVT environment. (Typically, parts in an environment that causes the chip to run fast have about twice the slew rate of parts in an environment that causes the chip to run slow). Of particular concern in this product were the digital signals that travel between the main logic board and the carriage board. These signals, which are in the megahertz range, travel approximately 19 inches along an unshielded and untwisted flex cable. This was deemed the most EMI-prone piece of the design. The I/O pads that drove these signals were designed to have as slow a slew rate as the signal speed would allow. As a backup system, phase-locked loop hardware and additional logic to dither the system clock or the signals on the flex cable was designed into the ASIC. The result of this design effort was successful passing of the EMI regulations on schedule.

The other big regulatory threat, ESD, was recognized early in the project by both the designers and the ASIC vendor (ICBD, a division of HP). Because ESD events can cause damage that will not immediately destroy the chip but instead will cause it to fail months or even years later, inadequate ESD protection can result in product reliability and customer satisfaction problems down the road. The chip needed to be able to withstand ESD events both before and after being put on a printed circuit board. When the chip is on a printed circuit board, external components can be placed around the chip to help protect it. However, each component adds cost to the product, so integrating protection into the chip results in a cost savings. Also, although the chips will be in a relatively controlled environment before being put onto the printed circuit board, ESD events can and will occur, so the chip needs to be able to withstand them.

When an ESD event occurs at the chip pins, a large current between the discharge point and ground is induced in the ASIC. Most structures internal to the ASIC cannot withstand such a large current without damage. Therefore, a chip designed to withstand ESD has structures that can withstand the high current, and routes the current to these structures rather than to the chip internals. Since an ASIC's only contact to the outside world is through its pads, ESD protection devices are generally located at the pads. During the design of the ASIC, the ASIC vendor assigned a dedicated engineer to evaluate the ESD design. A current-limiting resistor and a reverse-biased diode were used in each pad to limit the current that can reach the chip internals. Additionally, shunt structures between V_{dd} and ground were carefully designed and positioned in the IC. This early involvement by the vendor resulted in a solid design that meets HP ESD requirements.

Conclusion

By taking into account the HP DeskJet 820C's overall project goals of low cost and time to market from the start, a well-optimized digital controller for the printer was delivered on schedule. The chip is specifically designed for HP's Printing Performance Architecture. By integrating all digital functions in the printer on a single piece of silicon, the cost of the electronics in the product was greatly reduced over the previous-generation product while maintaining or increasing performance. The design team used the latest ASIC development tools to deliver a correctly functioning ASIC on a very tight schedule. Through the use of hardware emulators, the firmware team was able to begin coding before the final chip design had been released for manufacturing, further speeding the printer's design. All EMI and ESD requirements for the product were met on schedule. A lithograph of the final chip silicon is shown in Fig. 5. The chip area is approximately 81 mm^2.

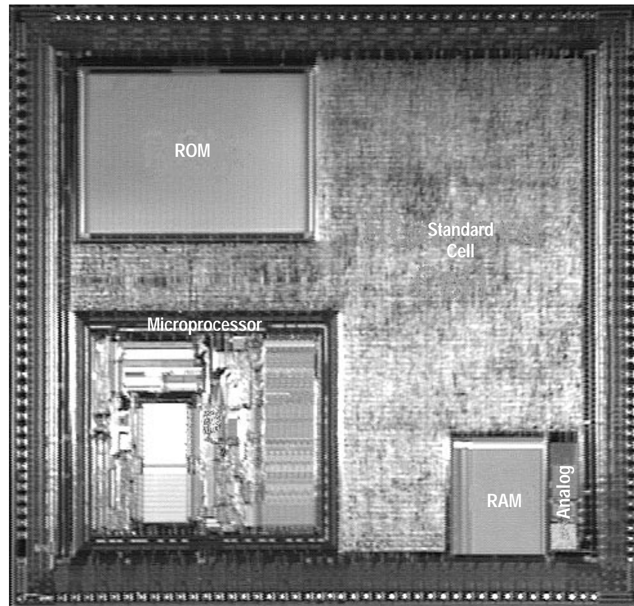


Fig. 5. *HP DeskJet 820C digital controller ASIC.*

Acknowledgments

The authors would like to acknowledge Tom Pritchard and Mark Thackray for their contributions to the ASIC. They would also like to thank the entire team at the HP Integrated Circuit Business Division for their design, development, and manufacturing contributions.

Next-Generation Inkjet Printhead Drive Electronics

By integrating the functions of four ICs into one new custom IC and then moving all the electronics related to the pens up to the carriage with the pens, significant savings were realized. A simple, low-contact-count, inexpensive flexible cable is used to connect the carriage to the main printed circuit assembly.

by Huston W. Rice

The project team for the HP DeskJet 850C printer developed the many elements of the printing system in parallel. In particular, the print cartridges (called *pens*) were new designs, along with the electronics that control them. As a result of the pens being new designs, their drive and control requirements were not completely defined, but were changing during the development program. The result of this was a system that worked well electrically, but was not fully optimized from a cost standpoint.

In particular, two aspects of the pen drive system presented opportunities for significant cost reduction. First, the flexible cable connecting the carriage and pens to the main printed circuit assembly was very elaborate and fairly expensive. Second, the electronics that control the pens were implemented in four different analog ICs, three of them custom ASICs.

With the advantage of being able to look back on the now well-defined system needs, a new approach was selected. By integrating the functions of the four ICs into one new custom IC and then moving all the electronics related to the pens up to the carriage with the pens, significant savings were realized. The new, highly integrated ASIC is less expensive to purchase and to assemble into the product. Since the signals are restricted to digital data and raw power, a simple, low-contact-count, inexpensive flexible cable is used to connect the carriage to the main printed circuit assembly.

For this design approach to be successful in the HP DeskJet 820C, several issues had to be overcome. For the greatest benefit, ***all*** of the electronics associated with the the pens had to be contained on the carriage printed circuit assembly. Would it all fit? Because of a tight schedule and limited mechanical engineering staffing, no mechanical changes could be made to the carriage assembly to make more room. An additional mechanical constraint was that no components could be placed on the bottom half of the printed circuit board, which was needed for the connectors for the pens. To get the circuits to fit, all the analog IC functions had to be integrated into a single ASIC. Could all the different functions—power control, digital I/O, sensitive analog-to-digital measurements, power drivers—be integrated into a single device? If all the analog functions from four ICs were integrated into one IC, would there be thermal overheating issues in the IC? Would there be problems with radiated electromagnetic emissions from the digital interface to the carriage over a simple unshielded flexible cable?

To provide an aspect of excitement to the program, once this approach was chosen, there was no easy alternative to fall back upon if the above issues could not be dealt with. If this design failed, the whole HP DeskJet 820C printer program would be put in jeopardy.

Carriage Electronics Implementation

A key architecture change was made in the pen drive and control electronics in the HP DeskJet 820C compared to the DeskJet 850C. The power supply for the pens was modified in two ways. First, two independent dc-to-dc converters are used to supply power to the black and color pens in the DeskJet 850C. In the DeskJet 820C, a single pen power supply is used to drive both the black and color pens. Second, the control topology of the dc-to-dc converter was changed, as explained later in this article. The DeskJet 850C design requires seven large capacitors, two inductors, two power FETs, two power diodes, and several small discrete resistors and capacitors. All of this was replaced with two capacitors, one inductor, one power FET, one power diode, and one power resistor. This eliminates not only the need for several square inches of printed circuit board space that was not available on the DeskJet 820C carriage printed circuit board, but also the cost of the unneeded components.

The two pens in the product (black and color) must be driven at different voltages, and the DeskJet 820C design now only has one power supply, which is shared between the pens. This forced a change in the way printing is done. In the DeskJet 850C, both pens can be driven at any time, allowing maximum flexibility in how the printed image can be formed, and therefore maximum speed. In the DeskJet 820C, printing with the black and color pens alternates. For instance, black may be printed from right to left and then color from left to right. This difference costs a little in print speed for some color documents, but was key in enabling all the electronics to fit on the carriage printed circuit assembly.

Several techniques were used to integrate all the pen electronics onto the carriage for the HP DeskJet 820C. Beyond the power supply changes, the next most important step was designing a mixed-signal analog/digital/power ASIC that integrates all the functions required to drive and control the pens. The general strategy was to integrate all the relatively small-signal electronic functions into one ASIC to minimize the the total component count. This both minimizes the cost and uses the minimum printed circuit board area on the very small carriage printed circuit assembly. However, to keep the ASIC silicon die area under control and to minimize the total power dissipated by the ASIC, several key components are not integrated. The power FET and diode for the dc-to-dc converter, both very large devices (from a silicon area point of view) are implemented as discrete devices externally. Two linear regulators are also implemented with off-the-shelf discrete devices to keep their power dissipation out of the ASIC package. Beyond these parts and some discrete capacitors and inductors that cannot be integrated, everything else is internal to the ASIC.

The process of developing the ASIC was the most difficult aspect of the carriage electronics design. Because of the high expected production volumes, at least two independent suppliers were needed. In the special mixed-signal/power IC industry, there is considerable process variation from one supplier to the next. However, only pin compatibility is required between sources. The two ICs do not have to be identical. Over a period of about six months, the analog ASIC was codeveloped by HP and the two suppliers. This allowed system design trade-offs to be made to keep both ASICs compatible. In addition, the overall program schedule demanded that the first pass of this full custom IC had to work, because there was only time for small revisions to the device before production started on the HP DeskJet 820C. As a result of excellent design teams at the suppliers and the careful codevelopment communication between HP and the suppliers, the first samples of the ASICs worked with only a few faults. Simple IC mask changes and the addition of a few small external components resulted in the system being completed on time.

In addition to the mixed-signal ASIC, three printed circuit board layout techniques were used to get all the components to fit. First, a 3D map of available space for components on the circuit board was generated. With this, small parts could be tucked under mechanical components, and the larger components could be carefully placed to avoid mechanical interferences. Second, the placement of the components was very carefully designed to minimize interconnect distances and the number of vias required. Third, the classical layout placement design rules that govern component spacings were pushed or outright violated. Breaking the rules was justified because the alternative was changing to a two-sided surface mount assembly process, which is a much more expensive and unattractive alternative.

In the end, all the parts were made to fit on the top side of the printed circuit board. An added benefit of the careful printed circuit board design is a very low-noise circuit. During the development process, we discovered many of the high-current switching circuits interfered with the sensitive measurement circuits. The compact layout provided significantly better performance than earlier prototype layouts.

As might be expected, the low-noise layout is also low-noise from an electromagnetic radiation point of view. Steps taken to control the slew rates of the digital signals on the flexible cable also proved effective in minimizing radiated emissions from the cable.

Finally, the steps taken to minimize the power dissipation in the mixed-signal ASIC were successful, to the point that the IC operates at junction temperatures less than 100°C under even the most extreme conditions.

Pen Operation

The black and color pens in the HP DeskJet 800 family of printers operate as a matched pair to deliver high-quality color documents. The pens themselves are in many senses the heart of the printer, and all of the electrical and mechanical systems are designed to support and optimize their performance. The electrical systems that drive and control the pens accomplish two major tasks: they maintain the temperature of the printheads to optimize the print quality, and they drive the correct inkjet nozzles at the right times to print the desired image on the paper.

The viscosity of the ink in the pens is sensitive to temperature, and the size of the drops ejected by the pens is sensitive to ink viscosity. By controlling the temperature of the pens, the viscosity and therefore the drop size can be controlled. Consistently sized drops provide the best print quality. Integrated into the pens is a temperature sensor, which can be used to measure and control the temperature of the pens, and therefore the ink viscosity and drop volume.

In previous generations of inkjet pens, the task of driving the pen nozzles has been fairly straightforward. The nozzles were arranged into columns on the pens, and every nozzle (or firing resistor) was controlled and driven directly. The pens in the HP DeskJet 820C printer have 300 nozzles (black pen) or 192 nozzles (color pen, 64 nozzles for each color). This high nozzle count makes it impossible to drive each nozzle directly with a dedicated signal and interconnection. For these high-nozzle-count pens, a matrix drive technique is used. The method is the same for the black and color pens. The matrix drive has two benefits. First, the number of connections to the pens is now much lower: 22 addresses and 16 columns can select $22 \times 14 = 308$ nozzles. The connection count is $22 + 14 + 14 = 50$. (The second 14 connections are for the column ground return currents.) Second, for power reasons, all the nozzles cannot be fired at one time. For each nozzle, a 2.5-μs firing pulse is applied to the nozzle resistor that boils the ink and ejects the drop. The pulse voltage is about 10V and the current is 250 mA. If all 300 nozzles were driven at the same time, a total current of 75A at 10V would have to be available! This is impractical. For power reasons alone, all the nozzles cannot be fired at once, and the matrix drive provides a convenient way to distribute firing the nozzles in time.

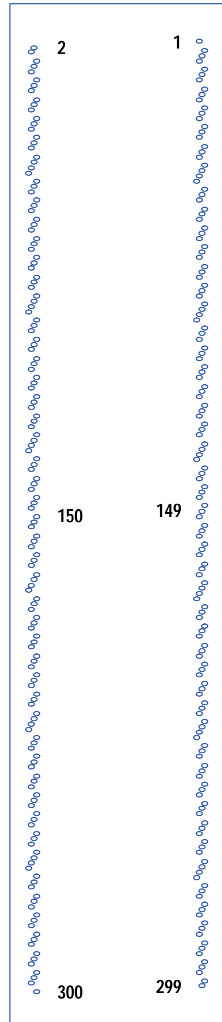


Fig. 1. *Black pen nozzle placement.*

The pen is electrically constructed as a series of 22 address inputs driving FETs in the rows of the matrix and 14 primitive inputs driving the firing resistors in the columns of the matrix. Inside the pen are selection FETs for each nozzle resistor; these can enable or disable a given nozzle.

The pen is driven one address row at a time. First, address one is driven, turning on the selection FETs for the top row of 14 nozzle firing resistors. Any of the 14 primitives are then driven (all, a few, or none) with the previously mentioned 10V, 250-mA pulse to fire the desired nozzles for each column associated with row 1. Address one is then turned off and address two is driven, selecting the next row of FETs and nozzle resistors. The desired primitives are again driven, but this time firing nozzles associated with row 2. This process is continued through address 22 and then repeated. By sequencing through all 22 addresses, every one of the 300 nozzles can be selected and driven. (Note: 22 addresses times 14 primitives yields 308 potential nozzles. Since the pen only has 300 nozzles, eight of the combinations do not have a selection FET or nozzle resistor.)

Mechanically, the pen nozzles are arranged in a pattern to generate proper images, even though all the nozzles are not fired at the same time. The black pen in the HP DeskJet 820C is capable of 600-dpi printing. The print swath (the band of ink printed in one pass) is 1/2 inch high, and the columns of dots in the swath are 1/600 inch apart.

A simple example will illustrate how the nozzles are arranged. Suppose we want to print a vertical line, 1/600 inch (one dot) wide. If the pen were constructed to fire all the nozzles at the same time to print a vertical column, the nozzles would be arranged in a vertical line on the pen. For power reasons, the nozzles are fired in 22 different groups of 14 (the 22 addresses and 14 primitives), and these are not all driven at the same time. Since the pen is continuously moving while the nozzles are fired, the desired vertical line would come out jagged or sloped if the nozzles were arranged in a straight line on the pen. To get a straight line on the page, the nozzles are staggered to compensate for the timing differences of the firing (see Fig. 1).

There is one additional complicating factor: The 600-dpi black pen has the nozzles arranged in two groups, odd nozzles in one column (with some stagger to compensate for the firing timing), and even nozzles in another column about 4 mm away.

Two nozzle columns allow 1/300-inch spacing between nozzles in a given column rather than 1/600-inch, making the pen easier to manufacture.

Now, to print a vertical, one-dot-wide line on the page, the odd nozzles are first driven, one address group at a time. Some time later, after the pen/carriage assembly has moved 4 mm and the even nozzles are over the same location on the paper, the even nozzles are driven, one address group at a time.

The implication is that the data sent to the pens must be sequenced properly to compensate for the nozzles being fired in 22 different address groups and also for the 4-mm odd/even nozzle spacing on the pen.

The color pen is constructed in a similar manner. Just like the black pen, the nozzles are fired one address at a time and are staggered on the pen to compensate for this. The first 16 of the 22 black address lines are shared between the black and color pens, while the color pen has its own unique 12 primitive lines. The 16 addresses times 12 primitives are sufficient to drive the 192 color nozzles, 64 per color. Like the black pen, the nozzles of the three colors are placed in dual odd and even columns. Between all the colors and the odd and even columns there are six different color nozzle placement columns, all with small-scale stagger. Therefore, for a color document, the sequencing of data is even more complex than for black printing. The data has to be timed for the address and row sequential firing, separated for odd and even nozzle columns, and timed to compensate for the displacement of the three colors with respect to each other.

The task of sequencing the data for the pen nozzles was traditionally handled by the printer digital electronics. In the HP DeskJet 820C, the Printing Performance Architecture (PPA) implementation moves the data sequencing task to the host PC and driver, where a lot of data processing was already being done. This relieves the printer of the burden of this data sequencing task, and allows it to simply drive the nozzles selected by the data coming into the printer.

Pen Drive Electronics Functions

The pen control and drive electronics have two key tasks. They provide the driving signals to eject the ink from the pens, and they provide a temperature control system to maintain a constant temperature in the active area of the pen. To accomplish this, the overall carriage electronics system has the following functions, most of which are integrated into the mixed-signal ASIC (see Fig. 2):

- Two-way digital interface between the pen drive electronics and the main digital controller ASIC in the printer (see article, page 1). Data is sent to the carriage to control which pen nozzles to fire during printing and to give control commands for the analog-to-digital converter (ADC), pen power supply, and other circuits. Pen measurements made by the mixed-signal ASIC are sent to the digital ASIC.
- 22 address drivers to provide the 12V signals to turn on the FETs inside the pens and select the correct row to be driven. These drivers are shared by the black and color pens.
- 26 column drivers to provide the high-current drive that fires the ink out of the pens. 14 drivers are used by the black pen and 12 by the color pen.
- A 30W, programmable dc-to-dc converter to provide precision power for the column drive signals.
- Two temperature control systems, one for each pen. This consists of an ADC to make calibration measurements on the pen temperature sensor, DACs to set target temperatures for the pens, and control comparators and logic to implement the temperature control system.
- Electronics to measure the pen firing resistors, to determine if the pen is damaged.
- Circuitry to provide thermal protection to the analog ASIC and resetting functions.

The overall system provides all of the means necessary for the digital ASIC and firmware to control the pens, both to maintain their target temperature, and to drive specific nozzles to print the images desired by customers.

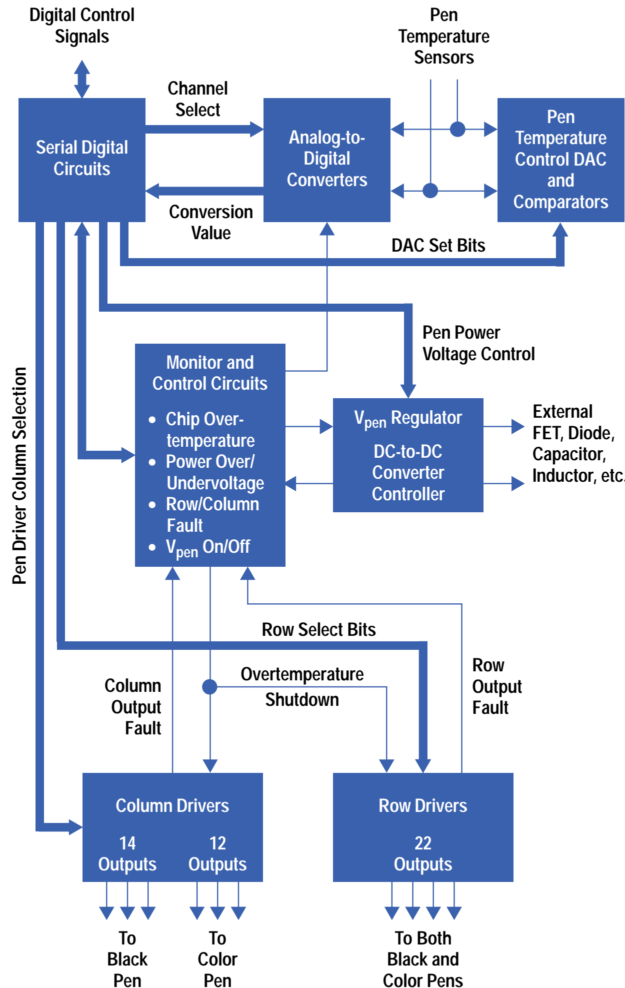


Fig. 2. *Block diagram of the mixed-signal ASIC.*

DC-to-DC Converter Design

The dc-to-dc converter that provides regulated power to drive the pens uses a new digital control technique. The feedback control of the regulator is a simple, purely digital system. If the voltage is too low, it turns on the regulator to full power and charges the bulk-storage filter capacitor to the target voltage as fast as possible. If the voltage is high, it turns off the regulator completely. Fig. 3 is a block diagram of the converter.

This control technique has several advantages:

- The control system is very simple and is easily integrated.
- The control system is inherently very stable and does not require additional compensation components or controlled values for the inductor and capacitor.
- The effective bandwidth of the regulator is very high, so it responds to changes in the load very quickly. Since the load placed on the regulator by the pens can change from 0 to 3A in less than 100 ns, fast response time is useful.
- As a result of the inherent stability and fast bandwidth, the size of the bulk storage capacitor could be reduced. Only one capacitor is needed in the HP DeskJet 820C design where three identical parts were used in the DeskJet 850C design. This was a key contributor to the goal of getting everything to fit on the carriage printed circuit board.
- Additional simple digital control functions, such as overcurrent and undervoltage shutdowns, were easily integrated.

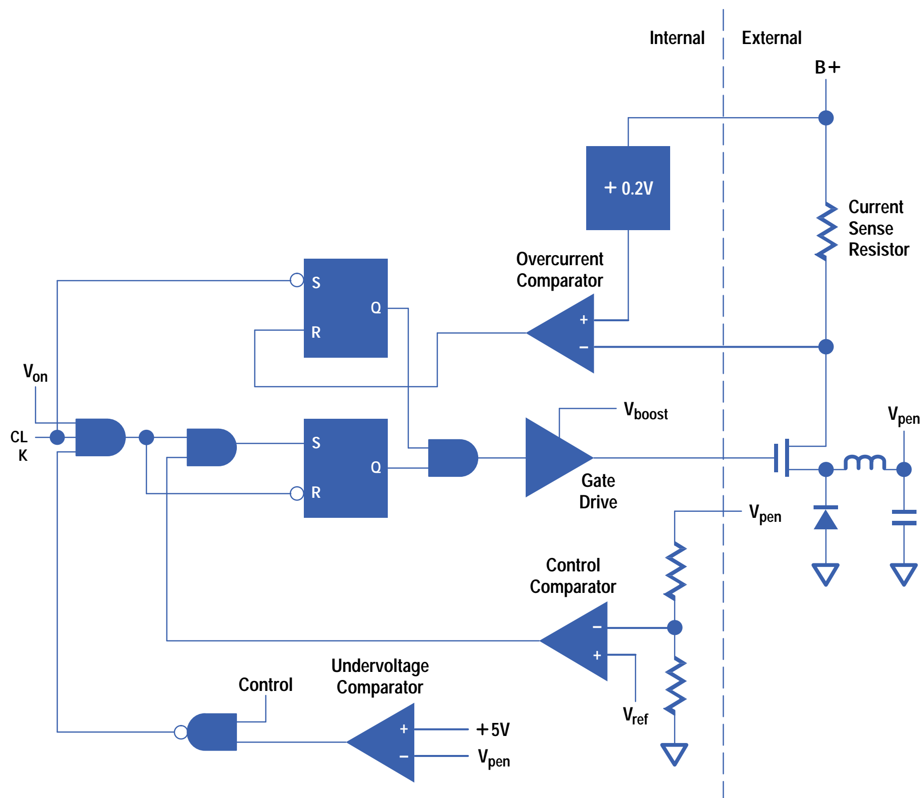


Fig. 3. *Block diagram of the dc-to-dc converter.*

Beyond the savings in components, the biggest benefit that this control topology presented was design flexibility. The definition of the mixed-signal ASIC, which contains the controller for the regulator, had to be finalized months before any testing could be started. By externally generating the clock for the regulator in the main digital ASIC under firmware control, changes could be made to the regulator ***in software***, up to the day printer production began. Two of the key parameters in the design of any switching power supply are the switching frequency and the maximum duty cycle. By moving the generation of the clock frequency and duty cycle to the firmware and digital hardware, the final decision on the clock parameters could be delayed until the system was carefully tested and analyzed. For instance, as changes were made to the printed circuit board layout, the clock was fine-tuned to compensate for the differences in performance that were seen. The clock can even be dynamically modified to provide regulator behavior to match the printer operation mode at any given time.

The net result is very successful. The programmable clock was used to tune the regulator performance to match the system need after many prototypes had been built and characterized. The regulator switches 30W of power from a poorly regulated 18V supply to a very well-regulated, programmable voltage appropriate for either the black pen or the color pen. The regulator only uses about about 6 cm^2 ($\sim$1 in^2) of printed circuit board area.

Acknowledgments

John Widder and George Barbehenn, both of the HP DeskJet 850C development team, provided vital consulting on the design of the pen drive and control systems. Mark Thackray and Leann MacMillian worked on the development and implementation of the serial digital interface for the mixed-signal ASIC. Mark Garboden and Carl Thompson wrote the firmware that controls all of the carriage electronics and integrated the assorted electronics components into a complete pen control system. Steve Stemple, the manufacturing support engineer for the HP DeskJet 820C, provided endless testing and design verification throughout the prototyping phases of the project.

The PA 7300LC Microprocessor: A Highly Integrated System on a Chip

A collection of design objectives targeted for low-end systems and the legacy of an earlier microprocessor, which was designed for high-volume cost-sensitive products, guided the development of the PA 7300LC processor.

by Terry W. Blanchard and Paul G. Tobin

In the process of developing a microprocessor, key decisions or guiding principles must be established to set the boundaries for all design decisions. These guiding principles are developed through analysis of marketing, business, and technical requirements.

Several years ago, we determined that we could best meet the needs of higher-volume and more cost-sensitive products by developing a different set of CPUs tuned to the special requirements of these low-end, midrange systems. The PA 7100LC was the first processor in this line, which continues with the PA 7300LC.

This article will review the guiding principles used during the development of the PA 7300LC microprocessor. A brief overview of the chip will also be given. The other PA 7300LC articles included in this issue will describe the technical contributions of the PA 7300LC in detail.

Design Objectives

Although the PA 7300LC was targeted for low-end systems, cost, performance, power, and other design objectives were all given high priority. With the design objectives for the PA 7300LC we wanted to:

- Optimize for entry-level through midrange high-volume systems (workstations and servers)
- Provide exceptional system price and performance
- Roughly double the performance of the PA 7100LC
- Provide a high level of integration and ease of system design
- Provide a highly configurable and scalable system for a broad range of system configurations
- Tune for real-world applications and needs, not just benchmarks
- Emphasize quality, reliability, and manufacturability
- Provide powerful, low-cost graphics capabilities for technical workstations
- Use the mature HP CMOS14C 3.3-volt 0.5-µm process
- Use mainstream, high-volume, and low-cost technologies while still providing the necessary performance increases
- Emphasize time to market through the appropriate leverage of features from previous CPUs.

Meeting Design Goals

We began by leveraging the superscalar processor core found in the PA 7100LC processor. First we investigated the value of high integration. Next we added a very large embedded primary cache, now feasible with the 0.5-µm technology. Then we enhanced the CPU core to take advantage of the new on-chip cache by reducing pipeline stalls. We also ensured high manufacturing yields by adding cache redundancy.

We found that integration supported our design goals in many positive ways. Because the primary cache, the secondary cache controller, and the DRAM controller could be on the same chip (see Fig. 1), we had an opportunity to design and optimize them together as a single subsystem. This was a large factor in allowing us to achieve such an aggressive system price and performance point. The high-integration approach also yielded much simpler system design options for our system partners. To further support these partners, we designed the integrated DRAM, level-2 cache, and I/O bus controller with extensive configurability (see ***Subarticle 6a*** “Configurability of the PA 7300LC”). This configurability enabled a wide variety of system options ranging from compact and low-cost systems to much more expandable, industrial-strength systems.

We were careful not to take a cost-first approach to this design. We believe that performance is just as important for customers of HP’s lower-cost systems. We took a total system approach in optimizing performance while emphasizing application performance over benchmarks in making design trade-offs. The highly optimized memory hierarchy shows dramatic improvement for the memory-intensive programs found in technical and commercial markets.

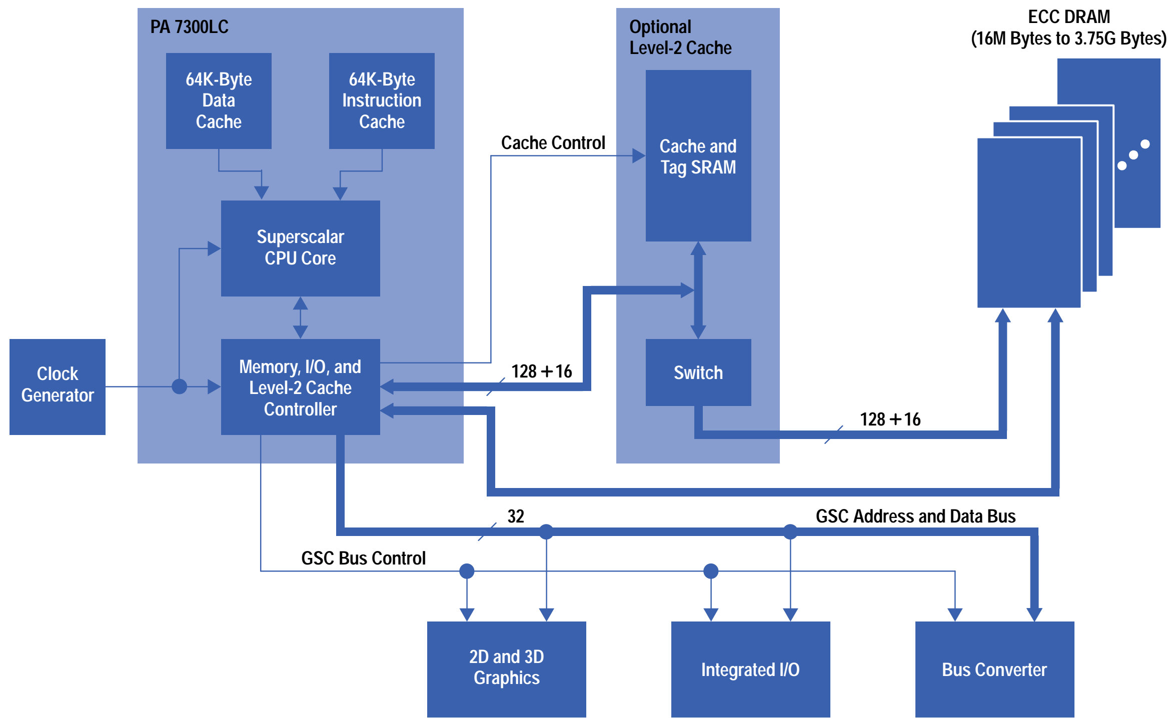


Fig. 1. *PA 7300LC system design.*

Another way of meeting our performance goals was to push the frequency while increasing the level of integration. We focused early on the layout and floor plan of the chip to enable higher-frequency operation. Through this effort, all critical paths were optimized. We tracked and optimized 62,000 individual timing paths during the design phase.

Despite leveraging the design from an existing CPU, the PA 7300LC design team still evaluated a large array of technical features and alternatives to meet our performance goals. Fundamentally, our approach was to build a robust CPU using a simple, efficient microarchitecture. Such a design ran less risk of functional bugs and allowed physical designers more leeway to push their circuits for higher performance.

On-Chip Primary Cache Decisions

It was clear from the beginning that the CMOS14C process would allow an on-chip cache of reasonable size, so a significant investigation was done to determine an optimal cache size and configuration. HP's System Performance Lab in Cupertino, California assisted us by repeatedly running benchmarks and code traces with different cache topologies and memory latencies.

Optimal Cache Size. Finding a balance between instruction-cache and data-cache sizes was difficult. The PA 7300LC was intended for use in both technical markets, where larger data caches are desired, and commercial markets, where programs favor large instruction caches. The standard industry benchmarks can easily fool designers into using smaller instruction caches, trading the space for more data cache or simply keeping the caches small to increase the chip's frequency. HP has always designed computer systems to perform well on large customer applications, so we included them in our analysis. Ultimately, we found that equally sized caches scaled extremely well with larger code and data sets. The typical performance degradation found when a program begins missing cache was mitigated by large cache sizes and our extremely fast memory system.

We could physically fit 128K bytes of cache on the die, so it was split into 64K bytes for the instruction cache and 64K bytes for the data cache. Not only would this provide impressive performance, but we noted that it would be the largest on-chip cache of any microprocessor when it began shipping.

Cache Associativity. Cache associativity was another issue. Recent PA-RISC implementations have used very large directly mapped (off-chip) caches. Associativity would reduce the potential for thrashing in the relatively small 64K-byte caches, but we were worried about adding a critical timing path to the physical design—selecting the right *way** of associativity and multiplexing data to the cache outputs. Increasing the ways of associativity would further reduce the thrashing, but make the timing even worse. The Systems Performance Lab included associativity in their performance simulations, helping us arrive

* Way, or N-way associativity, is a technique used to view a single physical cache as N equally sized logical subcaches. The PA 7300LC caches are two-way associative, so each 64K-byte cache has two ways of 32K bytes each. This provides two possible locations for any cached memory data, reducing the thrashing that can occur in a direct-mapped cache when two memory references are vying for the same location.

at our decision to implement two-way caches. To reduce the impact on timing, we eliminated cache address hashing, which had been used to reduce thrashing in directly mapped cache designs. Once we added associativity, hashing was no longer necessary.

Associative cache designs also need an algorithm for determining which way to update on a cache fill. Again, there are many alternatives, but our simulations showed the easiest approach to be the best. A pointer simply toggles on each fill, so that the ways alternate.

Other Cache Decisions

Many other cache decisions fell out of the same types of analysis. The data cache uses a copy-back rather than a write-through design** and a 256-bit path to the memory controller was included for single-cycle writes of copyout lines as shown in Fig. 2.

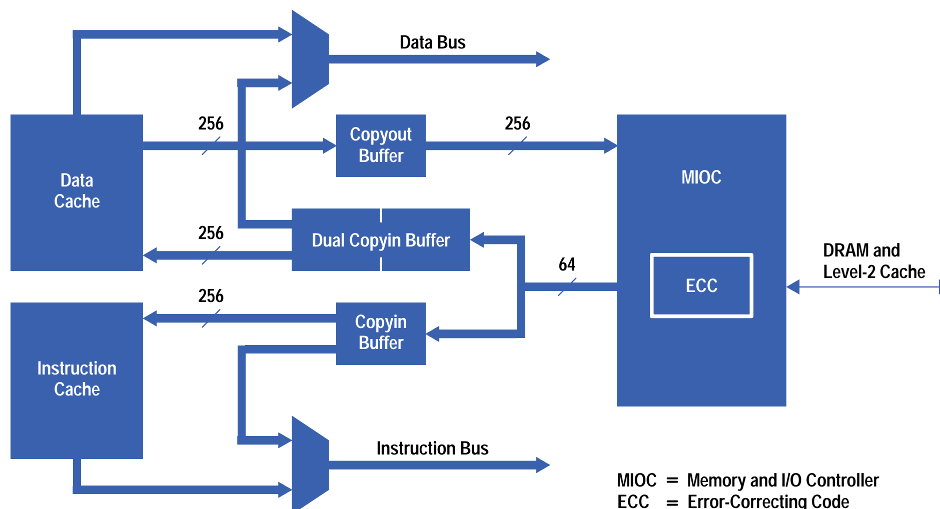


Fig. 2. *Primary cache system on the PA 7300LC chip.*

Moving the caches onto the chip also simplified changing the CPU pipeline to remove the “store-tail” penalty, in which stores on consecutive cycles cause a hang. This made it easier for compilers to optimize code.

Adding Spare Columns to the Cache Arrays. Manufacturability is a big concern for large VLSI memory structures like the PA 7300LC’s caches. Dense, regular structures like cache RAM cells are very susceptible to the smallest manufacturing defects, and just one failing bit out of 1,200,992 can make a part useless. To compensate, the cache design team added spare columns to the cache arrays. During the initial wafer test of a CPU die, an internal built-in self-test (BIST) routine runs to check for errors. If a bad RAM cell is found, the BIST signature indicates which column should be swapped out, and a laser is used to blow a special metal fuse on the chip. The bad column is replaced with the spare, fully restoring the chip’s functionality. The article on page 1 describes this feature in detail.

Integrated Memory and I/O Controller Decisions. Incorporating the memory and I/O controller (MIOC) onto the PA 7100LC chip was an important performance win, and we worked to make it even better on the PA 7300LC. Simply having the MIOC and CPU on the same die is extremely efficient. An off-chip MIOC would require a chip crossing for each data request and data return. Chip crossings are time-consuming, costing many chip cycles at 160 MHz. Since the CPU stalls on a critical request, chip crossings directly degrade performance.

Chip crossings also require additional pins on packages, driving up the cost. As a result, designers strive to keep external data paths narrow. With the MIOC on-chip, we were able to use wider data paths liberally for faster transfers. We placed some of the MIOC’s buffers inside the cache and used wider data paths to create a bus that is one cache line wide for blasting cache copyouts to the MIOC in one cycle.

Cost and Performance Decisions. Despite all the performance enhancements, the increased CPU frequency placed a burden on the MIOC to minimize memory latencies and pipeline stalls because of filled request queues. Blocking for an off-chip resource costs more CPU cycles at higher frequencies, so it was paramount that the PA 7300LC MIOC be fast and efficient. The challenge was in achieving this without significantly increasing the system cost.

** In a write-through cache design, data is written to both the cache and main memory on a write. In a copy-back cache design, data is written to the cache only, and is written to main memory only when necessary.

Doubling the external memory data path to 128 bits was a clear performance advantage, but it also increased system cost. Adding 72 (64 data + 8 error correction) pins to the CPU die and package came at a price. We were concerned that system designers would also be forced to create more expensive memory designs. Configurability was the best solution. The increased performance warranted adding pins to the CPU, but the MIOC was designed to support a 64-bit mode for less expensive memory designs in low-cost systems.

Off-Chip Second-Level Cache Performance. In addition to the primary cache, one of the PA 7300LC's most intriguing features is its second-level cache (see Fig. 3). Even with the MIOC's very fast memory accesses, it takes at least 14 CPU cycles for cache miss data to be returned. While this is excellent by industry standards, we had the opportunity to make it even faster by implementing an off-chip second-level cache.

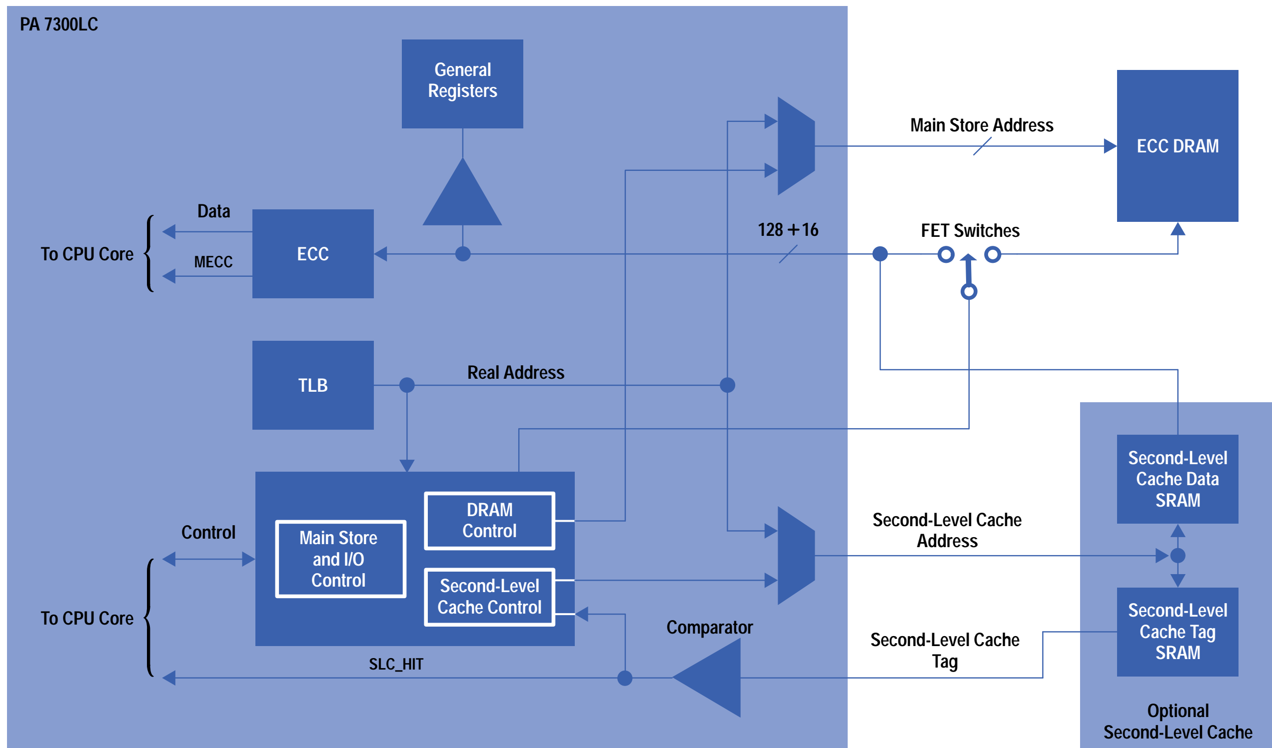


***Fig. 3.** Memory subsystem for the PA 7300LC.*

In many cases, the CPU is stalled during the entire memory access. A typical second-level external cache design could drastically reduce the number of stall cycles, but would be expensive. The engineering pros and cons were debated, and a very interesting solution was found. Address pins for a second-level cache were added to the CPU, but the second-level cache and DRAMs share the memory data lines (either 64 or 128). Very fast FET switches are used to shield second-level cache accesses from the heavy DRAM line loads until it is determined that the second-level cache will miss. While adding one cycle to memory accesses, this technique reduces access time to only six cycles on a second-level cache hit. The second-level cache is optional for low-cost systems or for those applications where a second-level cache is not beneficial.

MIOC Design Enhancements. Internally, the MIOC design was enhanced in many areas in the PA 7100LC MIOC. The internal pipeline was split into independent queues for memory and I/O, preventing memory stalls during long I/O operations. Reads can be promoted ahead of memory writes to satisfy CPU requests rapidly, and graphics writes are accelerated ahead of other transactions to increase graphics bandwidth. Finally, the GSC* (general system connect) interface was enhanced to improve graphics bandwidth by well over 200% over the PA 7100LC and to support a broader range of CPU:GSC operating ratios.

CPU Core Decisions

Removing the Phase-Locked Loop. Because of its higher operating frequency, the original PA 7300LC design contained a phase-locked loop circuit to synthesize both CPU and system clocks. Designing a phase-locked loop in a digital CMOS process is challenging and historically has affected yield and robustness in VLSI designs. When an inexpensive external clock part was found, we decided to recover the phase-locked loop circuit area and reduce technical risk by removing it.

* The GSC is the local bus that is designed to provide maximum bandwidth for memory-to-graphics transfers.

Integer and Data Cache Controller Enhancements. The on-chip caches caused both the integer and data cache controllers to be redesigned, and significant enhancements were included in both. The data cache controller added a deeper store buffer, and by also modifying the instruction pipeline, we were able to eliminate completely the store-tail problem mentioned earlier. Also, memory data is bypassed directly to execution units before error correction, with later notification in the rare event of a memory bit error.

The instruction cache controller expanded the instruction lookaside buffer (ILAB) from one entry to four, and improved the performance of bypassing instructions directly from the MIOC to the execution units. Both are very tightly coupled to the MIOC so that memory transfers to and from the caches are extremely fast.

Summary

We developed a set of guiding principles based upon marketing, business, and technical requirements for this system. The guiding principles enabled the design of an exceptional microprocessor targeted to the volume and price/performance requirements of the workstation and server market. A large part of the overall success of this design comes from the well-engineered cache and memory hierarchy. The technology we chose allowed us to develop a high-capacity primary cache and a rich set of performance-improving features.

The PA 7300LC design met its schedule and exceeded its performance goals. Customers are receiving PA 7300LC- based systems today.

Acknowledgments

We wish to thank the rest of the PA 7300LC design team in HP's Engineering Systems Lab in Fort Collins, Colorado for their superb technical contributions. We also express appreciation to our partners in the General Systems Lab and Fort Collins Systems Lab, as well as various HP marketing organizations for providing customer input and requirements. Finally, we express appreciation to the Systems Performance Lab for their efforts in running performance simulations on our behalf. A special recognition is made to Tian Wang (developer of the PA 7300LC performance simulator) who passed away during this development effort. We extend our sympathies to his family.

Configurability of the PA 7300LC

Rather than choosing a single, inflexible memory and level-2 cache configuration, we architected the PA 7300LC so that system designers can make price and performance trade-offs themselves. Most of the choices available to designers are in the memory system.

Bus Frequencies

The PA 7300LC GSC (general system connect) bus interface supports several CPU:GSC frequency ratios. GSC frequencies at or near the bus maximum frequency of 40 MHz can be maintained even when the CPU is running at noninteger multiples of the bus frequency (e.g., 132 MHz).

Memory Interface

The memory interface can be designed with either 64-bit or 128-bit (72-bit or 144-bit with error correction) data paths. A maximum of 16 memory banks is supported, and each bank can hold from 8M bytes to 512M bytes of DRAM. The DRAM technology can be either FPM (fast page mode) or EDO (extended data out), with chip sizes from 4M bits to 256M bits. A broad range of DRAM speeds is allowed, as DRAM timing can be software programmed using a nine-element MIOC (memory and I/O controller) timing vector.

Memory error correction is optional. Single-bit correct and double-bit and four-bit burst error detection schemes are available, all with sufficient error logging for system diagnosis and program data protection.

Level-2 Cache

The level-2 cache is completely optional. Three types of SRAM are supported: register-to-register, flow-through, and asynchronous. Depending on the SRAM speed and CPU frequency, level-2 cache latencies of two, three, or four CPU cycles can be programmed into the MIOC. Parity error protection on the SRAM data is also optional.

Functional Design of the HP PA 7300LC Processor

Microarchitecture design, with attention to optimizing specific areas of the CPU and memory and I/O subsystems, is key to meeting the cost and performance goals of a processor targeted for midrange and low-end computer systems.

by Leith Johnson and Stephen R. Undy

The PA 7300LC microprocessor is the latest in a series of 32-bit PA-RISC processors designed by Hewlett-Packard. Like its predecessor, the PA 7100LC,[1,2] the PA 7300LC design focused on optimizing price and performance. We worked toward achieving the best performance possible within the cost structures consistent with midrange and low-end systems. This paper describes the microarchitecture of the two main components of the PA 7300LC: the CPU core and the memory and I/O controller (MIOC).

CPU Core Microarchitecture Design

Approximately one-half of the engineering effort on the PA 7300LC processor was dedicated to the design of the CPU core. The CPU core includes integer execution units, floating-point execution units, register files, a translation lookaside buffer (TLB), and instruction and data caches. Fig. 1 shows a block diagram of the CPU core.

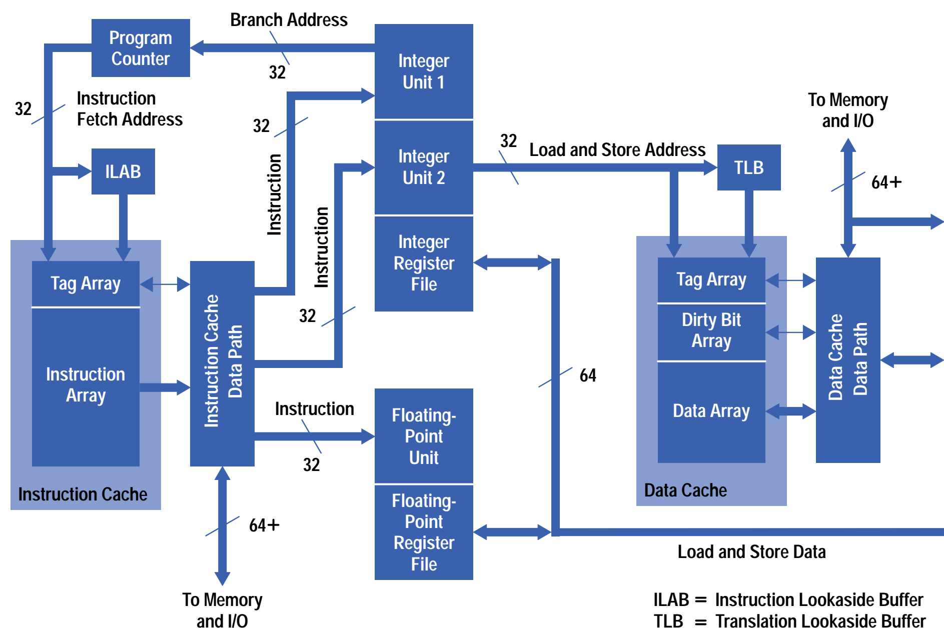


Fig. 1. *The PA 7300LC CPU core block diagram.*

Core Design Objectives

The design objectives for the PA 7300LC processor were to provide the best possible performance while choosing the proper set of features that would enable a system cost appropriate for entry-level and high-volume workstation products. To reach this goal, we integrated large primary caches on the processor chip and developed a tight coupling between the CPU core and the memory and I/O subsystems. The design objectives for the PA 7300LC are discussed in detail in ***Article 6***.

CPU Core Differences

The PA 7300LC CPU core is derived from the PA 7100LC CPU design.[1,2] Although the PA 7300LC has many similarities with its predecessor, there are some key differences in the design that allowed us to meet our performance objectives. The first difference is that the PA 7300LC runs at 160 MHz compared to only 100 MHz for the PA 7100LC. The most obvious difference is the large primary instruction and data caches integrated directly onto the PA 7300LC chip. The PA 7100LC only has a small (1K-byte) instruction cache on the chip. Also, the organization of the caches was changed to avoid many of the stall cycles that occur on the PA 7100LC. The cache organization is discussed later in this article. The PA 7300LC has a 96-entry TLB, compared to 64 entries on the PA 7100LC. Finally, the PA 7300LC has a four-entry instruction lookaside buffer (ILAB) while the PA 7100LC's ILAB contains one entry.

Pipeline and Execution Units

Like all high-performance microprocessors, the PA 7300LC is pipelined. What is notable about the PA 7300LC pipeline is that it is relatively short at six stages, while running at 160 MHz.

Fig. 2 shows a diagram of the PA 7300LC pipeline. It does not differ greatly from the pipelines used in the PA 7200, PA 7100LC, or PA 7100 processors.[1,2,3,4] The following operations are performed in each stage of the pipeline shown in Fig. 2:

1. Instruction addresses are generated in the P stage of the pipeline.

2. The instruction cache is accessed during the F stage.

3. The instructions fetched are distributed to the execution units during the first half of the I stage. During the second half of the I stage, the instructions are decoded and the appropriate general registers are read.

4. The integer units generate their results on the first half of the B stage. Memory references, such as load and store instructions, also generate their target address during the first half of the B stage.

5. Load and store data is transferred between the execution units and the data cache on the second half of the A stage.

6. The general registers are set on the second half of the R stage.

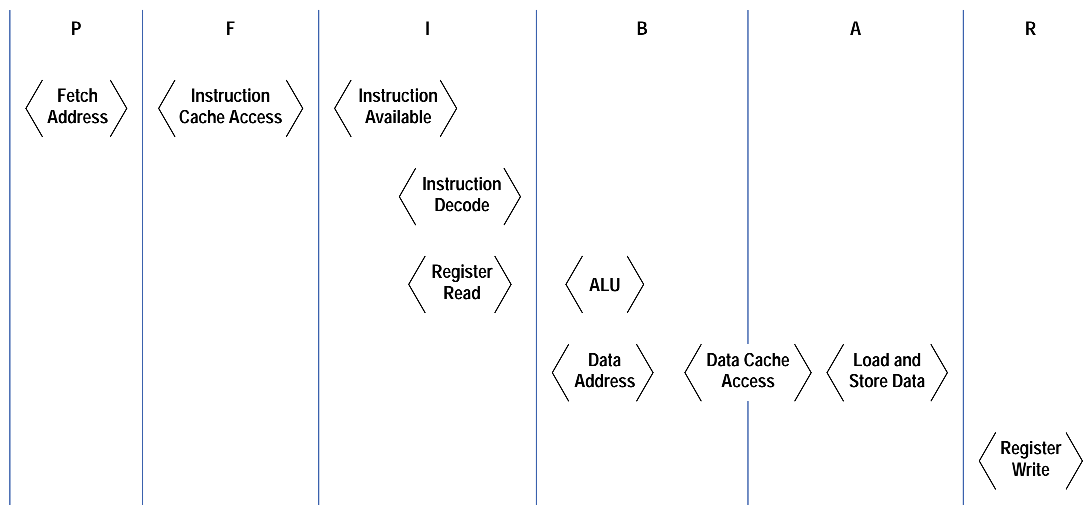


Fig. 2. *The PA 7300LC pipeline diagram.*

Superscalar Processor. The PA 7300LC is a superscalar processor, capable of executing two instructions per pipeline stage. This allows it, at 160 MHz, to execute at a maximum rate of 320 million instructions per second. This, however, is a peak rate that is rarely achieved on real applications. The actual average value varies with the application run. The theoretical maximum assumes the proper mix of instructions, but not every pair of instructions can be bundled together for execution in a single cycle. Fig. 3 shows which pairs of instructions can be bundled for execution in a single pipeline stage.

Delayed Branching. The PA-RISC architecture includes delayed branching.[5] That is, a branch instruction will not cause the program counter to change to the branch address until after the following instruction is fetched. Because of this, branches predicted correctly with a simple branch prediction scheme execute without any pipeline stalls. The majority of the remaining branches execute with only a single stall (see Fig. 4).

Two Integer Execution Units. The PA 7300LC contains two integer execution units. Each contains an ALU (arithmetic logic unit) that handles adds, subtracts, and bitwise logic operations. Only one unit, however, contains a shifter for handling the bit extract and deposit instructions defined in the PA-RISC architecture. Since only one adder is used to calculate branch targets, only one execution unit can process branch instructions. This same unit also contains the logic necessary to

First Instruction \ Second Instruction	flop	lwd	stw	ldst	flex	mm	nul	br
flop		X	X	X	X	X	X	X
lwd	X	*			X	X	X	X
stw	X		*		X	X	X	X
ldst	X				X	X	X	X
flex	X	X	X	X	X	X	X	X
mm	X	X	X	X	X			
nul	X							
br								

flop : Floating-Point Computation
ldw : Simple Load Word
stw : Simple Store Word
ldst : All Other Loads and Stores
flex : Integer ALU
mm : Shifts
nul : Can Cause Nullification
br : Branches

* Instructions will combine if they reference two different words in the same double word.

X Valid Superscalar Combinations

Fig. 3. *Valid superscalar instruction combinations for PA 7300LC.*

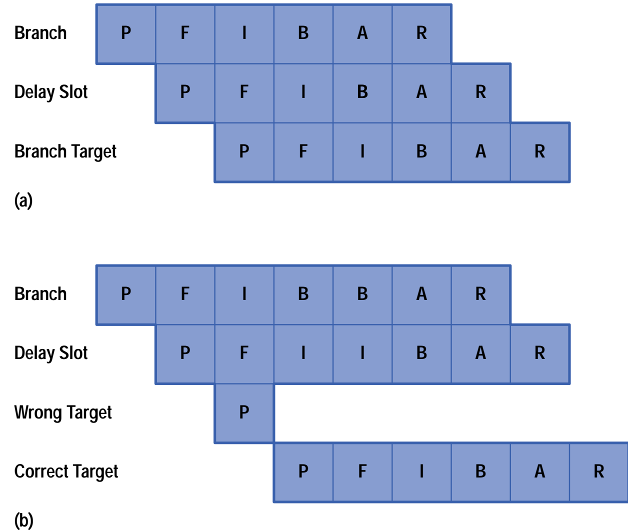


Fig. 4. *Branch behavior. (a) Correctly predicted branch. (b) Incorrectly predicted branch.*

calculate nullification conditions.* By limiting execution to only one branch or nullifying instruction per pipeline stage, we avoided a great deal of functional complexity. Finally, only one unit contains the logic to generate memory addresses. Since the data cache is single-ported, there is no need to have two memory addresses generated per cycle. In special cases, however, two integer load or store instructions may be bundled together, provided they use the same double-word address. As mentioned before, these asymmetries between the integer units prevent any two arbitrary integer instructions from bundling together. However, even with this limitation, compilers are able to take advantage of the integer superscalar capabilities of the PA 7300LC.

Multimedia Instructions. The PA 7300LC integer units implement a set of instructions first introduced on the PA 7100LC that accelerate multimedia applications.[1,2,6] These instructions allow each integer unit to perform two 16-bit adds, subtracts, averages, or shift-and-adds each cycle. Because of superscalar execution, the PA 7300LC can execute four of these operations per cycle for a peak rate of 640 million operations per second.

Floating-Point Unit. The PA 7300LC contains one floating-point unit. Contained in this unit is a floating-point adder and a floating-point multiplier. The adder takes two cycles to calculate a single- or double-precision result. It is pipelined so that it can begin a new add every cycle. The multiplier takes two cycles to produce a single-precision result and three cycles for a double-precision result. It can begin a new single-precision multiply every cycle and a new double-precision multiply every other cycle. Divides and square roots stall the CPU until a result is produced. It takes eight cycles for single-precision and 15 cycles for double-precision operations.

Instruction Cache and ILAB

Integrating a large primary instruction cache onto the processor chip broke new ground for PA-RISC microprocessors. In the past, our processor designs relied on large external primary caches. With the PA 7300LC, we felt that we could finally integrate enough cache memory on the processor chip to allow fast execution of real-world applications. Indeed, we have integrated twice as much cache on-chip as the PA 7100LC used externally in the HP 9000 Model 712/60 workstation (i.e., 128K bytes versus 64K bytes). The integrated cache not only improves performance but also reduces system cost, since an external cache is no longer mandatory.

See Subarticle 7a: ***Timing Flexibility.***

* The PA-RISC architecture enables certain instructions to conditionally nullify or cancel the operation of the following instruction based on the results of the current calculation or comparison.

Primary Instruction Cache. The PA 7300LC primary instruction cache holds 64K bytes of data and has a ***two-way set associative*** organization. A set associative cache configuration is difficult to achieve with an external cache, but much more practical with an integrated cache. When compared to a similarly sized directly mapped cache, it performs better because of higher use and fewer collisions. We chose a two-way associative cache over other ways to save overhead caused by the replication of comparators and to reduce the propagation delay through the way multiplexer.

The primary instruction cache is virtually indexed and physically tagged. Because the PA-RISC architecture restricts aliasing** to 1M-byte boundaries, we could use a portion of the virtual address (in this case, three bits) to form the index used to address the cache. To avoid using virtual address bits would have required us either to place the virtual-to-physical translation in series with cache access (increasing the cache latency) or to implement a large number of ways of associativity (in the case of a 64K-byte cache, this would have required a 16-way set associative organization).

Data Array Requirements. The instruction cache is composed of a tag array and a data array, each containing addresses and instructions. Without using more wires or sense amplifiers than those found in a conventional cache organization, we organized the data arrays in an unusual fashion in the primary caches on the PA 7300LC to meet two requirements.

The first requirement is for the instruction cache to supply two instructions per cycle to the execution units. Because the cache is two-way set associative, each location, or ***set***, contains instructions corresponding to two distinct physical addresses. Thus, for any given set (determined by the instruction fetch address), there are two possible choices for the instructions being read. Each of these choices is called a ***group*** (see Fig. 5a). For speed reasons, both groups are read from the instruction data arrays simultaneously. Logic that compares the physical addresses in the tag arrays (one per group) with the physical address being fetched from the data array determines which group is selected and sent to the rest of the CPU. Since this is the normal instruction fetch operation, it must be completed in a single processor cycle.

The second requirement is to be able to write eight instructions simultaneously, all to the same group. Because a write occurs as part of the cache miss sequence, it is important that the write take only a single cycle to interrupt instruction fetches as little as possible.

Fig. 5a shows the conventional method of addressing data arrays. Because of electrical and layout considerations, the upper four instructions of each eight-instruction-long cache line are kept in a separate array from the lower four instructions. Both the upper and lower arrays are addressed and read concurrently. There are four arrays in total: group 0 upper, group 0 lower, group 1 upper, and group 1 lower. The instruction fetch address sent to the instruction cache, Address[0:11], contains twelve bits. One address bit, Address[10], selects between the upper and lower arrays. The rest of the address bits, Address[0:9] and Address[11], go to all four arrays and determine which set (Set[x]) is read out of the arrays. This is accomplished with the 11-to-2048 decoders. In reality, four decoders, one for each array, would be needed, but they all connect to the same address. As discussed above, there are two possible pairs of instructions to choose from with a given address. A signal from logic called ***hit compare*** selects between the two possibilities. In the example shown in Fig. 5a, instructions 0 and 1 from group 0 are selected from the instruction cache.

This conventional approach meets our first requirement. However, it does not meet our second requirement. It cannot access all eight instructions as a single group simultaneously. This is because a cache line is located in two adjacent sets and only half of the line can be read (or more important, written) at any one time. For example, if the group 0 upper array is supplying instructions 0 and 1, it obviously cannot supply 2 and 3. The only way to solve this problem with the conventional approach is to split each array into two halves. This, however, would require twice as many wires and possibly sense amplifiers producing a sizable increase in area cost. By making a slight modification to the way the data arrays are organized and addressed, we found we could avoid this pitfall and meet both of our requirements.

Our addressing approach on the PA 7300LC is called ***checkerboarding***. Fig. 5b shows how instructions are fetched from the instruction cache on the PA 7300LC. There are, again, four arrays: left upper, left lower, right upper, and right lower. The most significant address lines, Address[0:9], go to all four arrays, while Left[11] goes only to the two left arrays and Right[11] goes only to the the two right arrays. A single address bit, Address[10], selects between the upper and lower arrays, as before.

When an instruction is fetched, both Left[11] and Right[11] are set to the value of Address[11]. Because of this, the operation is virtually identical to the conventional approach described above, except for one key difference: a cache line for a given group is spread across all four arrays, rather than just two. This can be seen in Fig. 5b, where the instructions corresponding to group 1 have been shaded. Each array contains pieces of cache lines from both groups in a checkerboard fashion.

Fig. 5c illustrates how checkerboarding allows simultaneous access to an entire cache line. By setting Left[11] to the group desired and Right[11] to the opposite value, all eight instructions from one group can be read or written. In the example shown in the figure, an entire cache line from group 1 is read out. Left[11] is set high, while Right[11] is set low. Address[0:9] selects which pair of sets, Set[x] and Set[x+1], are accessed. Fig. 6 lists the results of addressing the arrays with the various combinations of values on Left[11] and Right[11].

** Aliasing refers to intentionally allowing two different virtual addresses to map to the same physical address. The PA-RISC architecture restricts the number and location of bits that may differ between two virtual addresses.

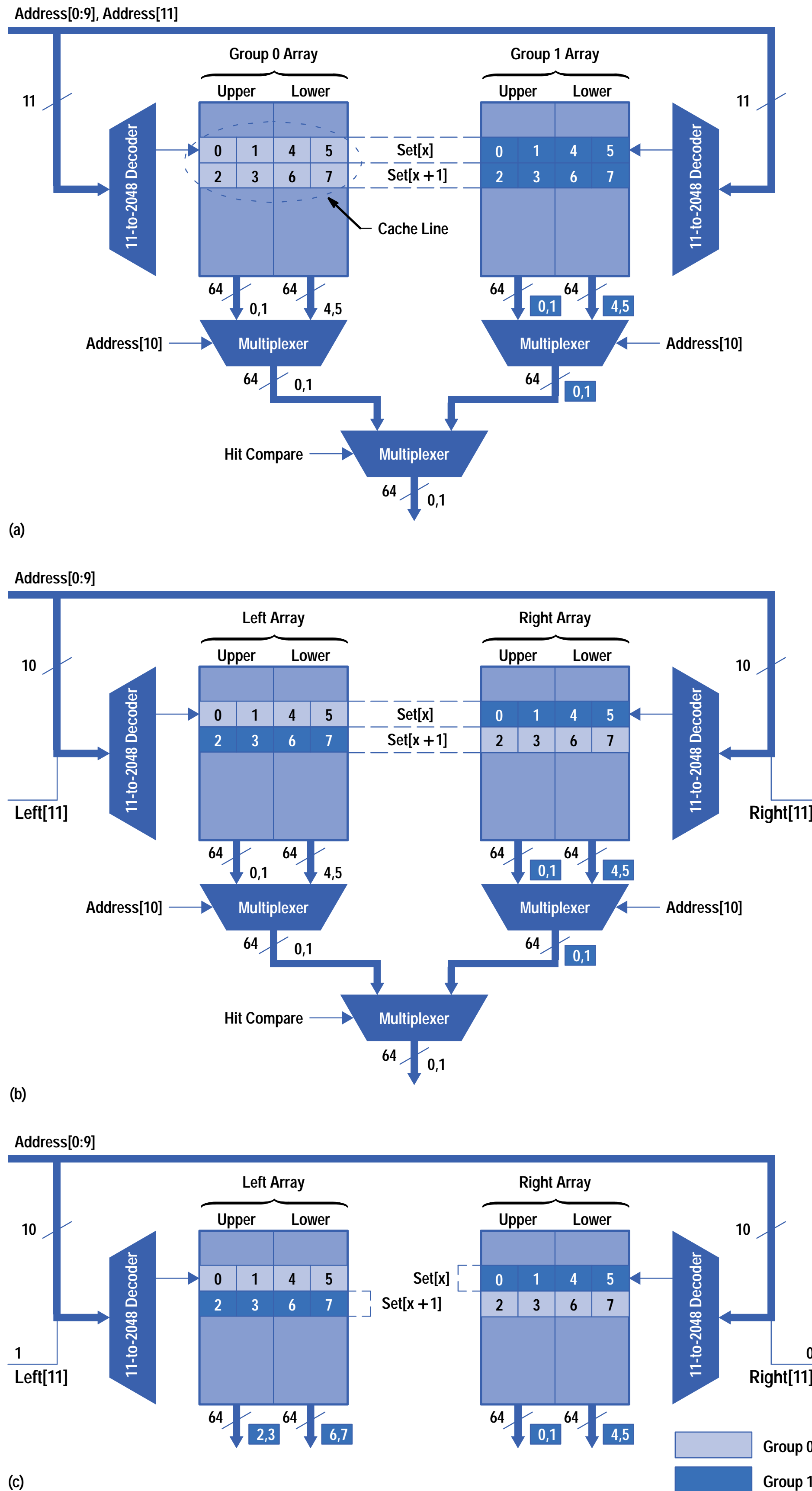


***Fig. 5.** (a) Conventional two-cache organization. (b) Checkerboard instruction fetch. (c) Checkerboard full line access.*

Left[11]	Right[11]	Left Upper Array Output	Left Lower Array Output	Right Upper Array Output	Right Lower Array Output
0	0	Set[x] Group 0 Instructions 0,1	Set[x] Group 0 Instructions 4,5	Set[x] Group 1 Instructions 0,1	Set[x] Group 1 Instructions 4,5
0	1	Set[x] Group 0 Instructions 0,1	Set[x] Group 0 Instructions 4,5	Set[x +1] Group 0 Instructions 2,3	Set[x +1] Group 0 Instructions 6,7
1	0	Set[x +1] Group 1 Instructions 2,3	Set[x +1] Group 1 Instructions 6,7	Set[x] Group 1 Instructions 0,1	Set[x] Group 1 Instructions 4,5
1	1	Set[x +1] Group 1 Instructions 2,3	Set[x +1] Group 1 Instructions 6,7	Set[x +1] Group 0 Instructions 2,3	Set[x +1] Group 0 Instructions 6,7

Note: x is determined by Address[0:9]

Fig. 6. ***The meaning of checkerboard address lines*** *Left[11]* ***and*** *Right[11].*

Instruction Cache Hit Stages. The CPU core will attempt to fetch a pair of instructions from the instruction cache every cycle during which it is not stalled. For example:

- The instruction fetch address arrives at the instruction cache at the end of the P stage of the pipeline.
- On the first half of the F stage, the word line decoders fire one word line to each array.
- On the second half of the F stage, the array is read, driving its value onto the bit lines to the sense amplifiers. The way multiplexer then selects the proper pair of instructions from the sense amplifier outputs.
- On the first half of the I stage, the instructions are driven to the execution units for decoding and execution.

Instruction Cache Miss Stages. In the case of an instruction cache miss, which is known by the end of the F stage of the pipeline, the entire pipeline will stall. A read request for an entire cache line will then be sent to the memory controller. This request is called a *copyin request*. A 64-bit data path between the memory controller and the instruction cache requires a minimum of four cycles to transfer the entire cache line to the instruction cache. Four cycles are required because the memory controller can only deliver 64 bits per cycle and a cache line contains 256 bits. The memory controller will return the pair of instructions originally intended to be fetched first, regardless of the pair's position within the cache line. As each pair of instructions is returned from memory, it is written into a write buffer. The instructions can be fetched directly from this buffer before they are written to the cache, with the first pair's arrival causing the pipeline to resume execution. This capability is commonly referred to as *streaming*. In effect, the write buffer forms a third way of associativity. After the last pair of instructions arrive from memory, the write buffer contents are written to the cache in one cycle.

Unified Translation Lookup Table. Since the instruction fetch address is a virtual address, it must be mapped into a corresponding physical address at the same time the instruction cache arrays are being accessed. Normally, a full instruction translation lookaside buffer, or ITLB, is used to perform this function. On the PA 7300LC, as on all recent PA-RISC processors, we felt that the performance improvements achieved with a separate ITLB and DTLB (for data accesses) did not warrant the increased chip area costs. Instead, we opted for a unified TLB that performs both instruction and data translations.

Instruction Lookaside Buffer (ILAB). Because both an instruction and a data translation are required on many cycles, a smaller structure called an instruction lookaside buffer, or ILAB, is used to translate instruction addresses, while the larger unified TLB is free to translate data addresses. The four-entry ILAB is a subset of the unified TLB and contains the most recently used translations. This strategy is quite effective because instruction addresses, unlike data addresses, tend to be highly correlated in space in that they generally access the same page, a previous page, or the next page.

When an instruction address does miss the ILAB, normally because of a branch, the pipeline will stall to transfer the desired translation from the unified TLB to the ILAB. We designed in two features to mitigate these ILAB stalls. On branch instructions that are not bundled with a memory access instruction (such as a load or store), the unified TLB will be accessed in parallel with the ILAB, in anticipation of an ILAB miss. If the ILAB misses, the normal ILAB stall penalty will be reduced. The second feature we added was ILAB prefetching. Every time the CPU begins executing on a new instruction page, the TLB will take the opportunity to transfer the translation for the next page into the ILAB. This can completely avoid the ILAB misses associated with sequential code execution.

Data Cache and TLB

We designed the data cache array to be very similar to the instruction cache arrays. Like the instruction cache, the data cache is two-way set associative, virtually indexed, and physically tagged. It is composed of three arrays:

- A data array, which has the same checkerboard organization as the instruction cache data array
- A tag array, which is almost identical to its instruction cache counterpart

- A dirty bit array, which has no counterpart in the instruction cache. This array keeps track of whether a data cache line has been modified by the instruction stream.

Although organized in a way similar to the instruction cache, the data cache's internal operation and effect on the CPU pipeline are quite different. The data cache and TLB operate in the A and B stages of the pipeline. A load instruction causes a data address to be generated in the first half of the B stage. The data cache word line decoders operate on the second half of the B stage. On the first half of the A stage, the arrays drive their values out. Based on the comparison between the physical address and the output of the tag arrays, the way multiplexer then selects the proper data value. This word or double-word value is then driven to the integer and floating-point units during the second half of the A stage.

A store instruction generates a data address in the same manner as a load instruction. That address is used to read from the tag array as described above. Instead of using the store address to read from the data array, however, the address from the head of a two-entry store queue (Fig. 7) is used to index into the data array on the second half of the B stage. The data from the head of the store queue is written into the data array on the first half of the A stage. The data from the store instruction is driven from the integer or floating-point units to the data cache on the second half of the A stage where it is written into the tail of the store queue.

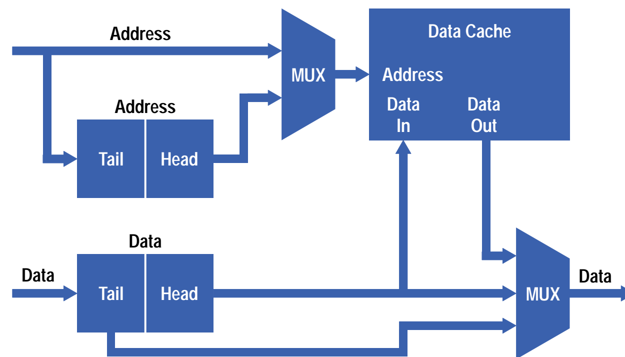


Fig. 7. The store queue.

Store Queue. A load can retrieve data directly out of the store queue if it is to the same address as the store. The necessity of the store queue is twofold:

- The floating-point unit cannot drive store data in time to write the data array during the proper pipeline stage. The store queue, therefore, provides the time to transfer the data from the execution units to the data cache.
- Memory cannot be modified until it is known that the store instruction will properly finish execution. If the store instruction is going to trap, say, because of a TLB fault, any architected state, such as memory, must not be changed.

The disposition of the store instruction is not known until the R stage of the pipeline, well after the data array is to be written. The store queue serves as a temporary buffer to hold pending store data. If a store that writes into the store queue subsequently traps, that store queue entry is merely invalidated. Also, by using a store queue, we are able to use a single bidirectional bus to transfer data between the execution units and the data cache. The store queue allows data to be transferred on the second half of the A pipeline stage for both load and store instructions, preventing conflicts between adjacent loads and stores in the instruction stream.

Semaphore Instructions. The data cache performs other memory operations besides load and store instructions. It handles semaphore instructions, which in the PA-RISC architecture require a memory location to be read while that location is simultaneously zeroed. In operation, a semaphore is quite similar to a store instruction with zeroed data, except that the semaphore read data is transferred on the second half of the A stage. In cases in which the semaphore is not present or modified in the data cache, the load and clear operation must be performed by the memory controller.

Flush and Purge Instructions. We must also execute flush instructions, which cause a given memory location to be cast out of the data cache. Related is the purge instruction, which at the most privileged level causes a memory location to be invalidated in the data cache with no cast out, even if the line is modified.

Reducing Miss Latency. Data cache misses are detected on the first half of the A stage of the pipeline. To reduce miss latency, the physical address being read from the data cache is forwarded to the memory controller before the data cache hit-or-miss disposition is known. This address is driven to the memory controller on the first half of the A stage. A "use address" signal is driven to the memory controller on the first half of the R stage if a cache miss occurs.

Copyin Transaction. A number of transaction types are supported between the CPU core and the memory controller. The most common type is a copyin transaction.

After receiving a copyin request, the memory controller returns the requested cache line, one double word at a time. As with instruction misses, the memory controller returns the data double word that was originally intended to be fetched first.

On load misses, when the critical double word arrives, it is sent directly to the execution units for posting into the register files. On integer load misses, the critical data is bypassed before error correction to reduce latency even further.

In the extremely rare event that the data contains an error, the CPU is prevented from using the bad data and forced to wait for corrected data. As each double word arrives from the memory controller, it is placed into a copyin buffer.

When all the data has arrived, the contents of the copyin buffer are written to the data cache data array in a single cycle. There are actually two copyin buffers to ensure that two data cache misses can be handled simultaneously.

Fig. 8 shows a block diagram of the copyin and copyout buffers.

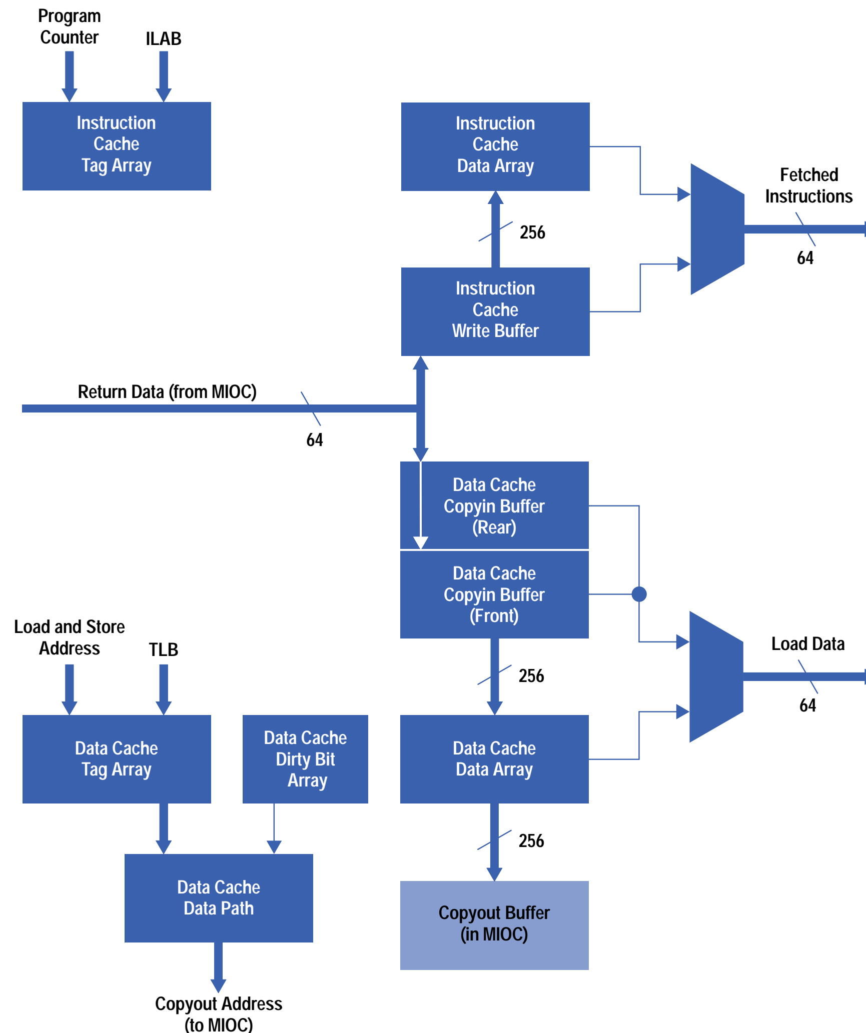


Fig. 8. *A block diagram of the copyin and copyout buffers.*

Copyout Transaction. A data cache line can contain modified data requiring posting or writing back to memory when cast out. To this end, another transaction type is implemented—a copyout transaction. A copyout is necessary under two circumstances. The first case is when a data cache miss is detected and the existing cache line selected for replacement has been modified. This is the most common case.

The second case is when a flush instruction is executed and hits a modified line in the data cache. The data cache supplies both a physical address and 32 bytes of data on a copyout. The data cache uses the checkerboard organization, so the full cache line read for the copyout takes only one cycle.

Reducing Cache Miss Penalties. In the PA 7300LC, we have taken a number of steps to reduce the penalty caused by cache misses. As mentioned above, we have reduced cache miss latencies. We have also continued to adopt a “stall-on-use” load miss policy pioneered on earlier PA-RISC designs.[4] In this policy a load miss stalls the CPU pipeline only long enough to issue the copyin transaction and possibly a copyout transaction. In many cases, the delay lasts for only one cycle. The CPU

will then only stall when the target register of the load instruction is subsequently referenced. If the critical data returns from memory fast enough, the pipeline will not stall at all.

Because memory data is not needed by the CPU on a store miss, the CPU only stalls once, again for only one cycle in many cases, to issue the copyin and copyout transactions.

A scoreboard keeps track of which words have been stored so that the copyin write will not overwrite more recent data. Since high-bandwidth writes to I/O space can be critical to graphics performance, under most circumstances the PA 7300LC will not stall on a store to I/O space. This optimization is possible because an I/O space access is guaranteed to miss the data cache, so there is no need to stall the CPU to perform a copyout read.

Cache Hints. The PA-RISC architecture defines cache hints to allow the programmer or compiler to communicate information that can be used by the hardware to improve performance.[5] We have implemented two of these hints on the PA 7300LC:

- Block copy store. Hints are used to indicate that software intends to write an entire cache line. In this case, there is no need to perform a main memory read on a cache miss. With this hint specified, upon detecting a store miss, the PA 7300LC simply zeros out a copyin buffer and continues without issuing a copyin transaction.
- Coherent operation semaphore hint. This optimization improves semaphore performance by not forcing the load and clear operation to the memory unit if the data is present in the cache.

TLB Access. All memory reference instructions are guaranteed access to the unified TLB containing both instruction and data translations, during the B and A stages of the pipeline. The TLB is fully associative and contains 96 page translations. The TLB receives a virtual data address on the first half of the B stage and drives a translated physical address on the first half of the A stage. This physical address goes to the data cache to perform hit comparison and to the memory controller in anticipation of a data cache miss.

In addition to containing 96 page entries, each of which maps to a 4K-byte page, the TLB also contains eight block entries used to map larger memory ranges. These block entries are managed by the operating system.

CPU Summary

Although the CPU core of the PA 7300LC is not dramatically different from its predecessors, several noteworthy features that improve performance and allow more cost-effective system designs include:

- A simple pipeline and a capable superscalar core that increased our operating frequency to 160 MHz.
- Substantial primary caches integrated directly onto the processor chip
- Most important, cache controllers that take advantage of integrated caches, resulting in features designed into the CPU core to increase the competitiveness of PA 7300LC-based systems.

Memory and I/O Controller Design

The memory and I/O controller (MIOC) is responsible for interfacing the CPU core to the memory and I/O subsystems. Integrating the MIOC on the same chip as the CPU core provides a tight coupling that results in outstanding memory and I/O performance. The memory controller includes a main memory controller and a controller for an optional second-level cache. The I/O controller interfaces the CPU core to HP's general system connect (GSC) I/O bus and handles direct memory access (DMA) requests from I/O devices.

CPU to MIOC Interface

The CPU core transmits four basic types of request to the MIOC:

- Copyins. These requests occur during first-level cache misses and are used by the CPU core to read a cache line from the memory subsystem.
- Copyouts. A copyout is a cache line from the CPU core that must be written to the memory subsystem because it was modified in the first-level cache by a store instruction. Copyouts are only issued when a modified cache line is replaced or flushed from the first-level cache.
- Uncached loads and stores. An uncached load or store request is a read or write to either memory or I/O for an amount of data that is less than a cache line.
- Load-and-clears. This request is an indivisible request to read a location and then clear it. This operation is needed to implement PA-RISC's semaphore mechanism. Requests that have addresses located in memory address space are processed by the memory controller, and all others are sent to the I/O controller.

The PA 7300LC has a four-entry copyout buffer. Copyouts are posted to memory as a background operation, allowing copyins to be processed before copyouts. New copyin requests are checked for conflict within the copyout buffer. If there is no conflict, the copyin is processed before all copyouts to help minimize load use stalls.

Second-Level Cache Control

Even though first-level caches on the PA 7300LC are relatively large for integrated caches, many applications have data sets that are too big to fit into them. The second-level cache (SLC) implemented for the PA 7300LC helps solve this problem. Logically, the SLC appears as a high-speed memory buffer; other than its performance improvement, it is transparent to software. The SLC is physically indexed, is write-through, has unified instructions and data, and is direct mapped.

The SLC becomes active after an access misses the first-level cache. The first-level cache miss indication becomes available after the TLB delivers the real address. As a result, there is little advantage to virtually indexing the SLC and real indexing avoids the aliasing problems associated with virtual caches.

Multiway Associative Cache Comparison. Multiway associative caches enjoy better hit rates because of fewer collisions. However, multiway caches are slower because of way selection, and for a given cache size, are much more expensive to implement with industry-standard components. For most applications, it is more advantageous to trade off size for ways of associativity.

Write-Back Cache Comparison. Write-back caches* generally have better performance than write-through caches. However, sharing the data bus with main memory alters this situation. If the SLC were write-back, lines copied out of the SLC would have to be read into the PA 7300LC, the error-correcting code (ECC) would have to be computed, and the line would have to be written back to main memory. This operation would be quite expensive in terms of bus bandwidth. Instead, dirty lines cast out by the first-level cache are written to the SLC and to main memory simultaneously.

Any valid line in the SLC always has the same data as its corresponding location in main memory. Writing simultaneously to main memory and to the SLC is slightly slower than simply writing to the SLC SRAM, but produces a good performance and complexity trade-off when compared to a write-back design.

DMA Interface. DMA reads and writes from I/O devices are typically sequential and do not exhibit the access locality patterns typical of CPU traffic. Entering DMA traffic into the SLC tends to pollute the SLC with ineffective entries. Instead, buffering and prefetching inside the DMA interface are better ways of improving DMA performance. To maintain consistency, an SLC check cycle is run for DMA writes, and if it hits, the line is marked invalid. DMA write data is always written to main memory and DMA reads are always satisfied from main memory. Because of the write-through design of the SLC described above, data in the SLC never becomes stale.

SRAM Components. The PA 7300LC is optimized for both price and performance. Relatively early in the design process, it became necessary to select the static random access memory (SRAM) components used to build the SLC. SRAM components are frequently used in cache construction because they offer high speed with moderate cost and capacities. Given the relatively long design cycles necessary to produce a complex microprocessor and the uncertainties of the semiconductor marketplace, it was impossible to predict which components would be most attractive from a price and performance perspective when the PA 7300LC entered full production. Instead of selecting a single component, the decision was made to support a broad range of SRAM types. This allowed component selection to be made late in the development cycle and even upgraded at some point during the production phase.

Second-Level Cache Size. Most popular computer benchmark programs have relatively small working sets and are not particularly sensitive to the performance of the memory system beyond the first-level cache. On the other hand, application programs have widely variable working set sizes. Some are small and fit well in the first-level cache and some exhibit little reference locality and don't fit in any reasonably sized cache. Hence, no single SLC size is appropriate. The PA7300LC SLC controller supports cache sizes ranging from 256K bytes up to 64M bytes. Although a 64M-byte SLC is expensive, it might be cost-effective for some applications.

The SLC data array width can be programmed to either 64 or 128 bits plus optional ECC. However, the width must match the width of main memory.

Memory Arrays. The SLC consists of two memory arrays: the data array and the tag array. The data array shares the data bus with main memory. As an option, ECC bits can be added to the data array, and the full single-bit correct and double-bit detect error control invoked for SLC reads. The tag array includes a single optional parity bit. If parity is enabled and bad parity is detected on a tag access, an SLC miss is signaled, the failing tag and address are logged, and a machine check is signaled.

Main Memory Control

DRAMs. Dynamic random access memory (DRAM) technology is used to construct main memory because of its high density, low cost, and reasonable performance levels. The main memory controller supports industry-standard DRAMs from 4M-bit to 256M-bit capacities. Systems can have up to 16 slots and total memory can be up to 3.75G bytes, the maximum possible with the PA-RISC 1.1 architecture.

Data Bus Width. Data bus width can be either 64 or 128 bits plus optional ECC. The 128-bit data bus width significantly improves memory performance. The 64-bit option supports lower-cost systems.

* In a write-back cache design (also called copy-back), data is written only to the cache on a write, and is not written to main memory until the cache line is invalidated. In a write-through cache design, data is written to both the cache and main memory on a write.

Main Memory Controller. The PA 7300LC main memory controller is very flexible and is able to support most types of asynchronous DRAMs. The controller is intentionally not SIMM/DIMM (single or double inline memory module) specific. This allows use of the PA 7300LC in a wide variety of system configurations. The main memory can support extended data out (EDO) DRAMs, which are similar to other DRAMs but use a slightly modified protocol that pipelines the column access.

Fig. 9 shows the timing diagrams of read accesses, emphasizing the improved data bandwidth of EDO DRAMs compared to standard page-mode DRAMs.

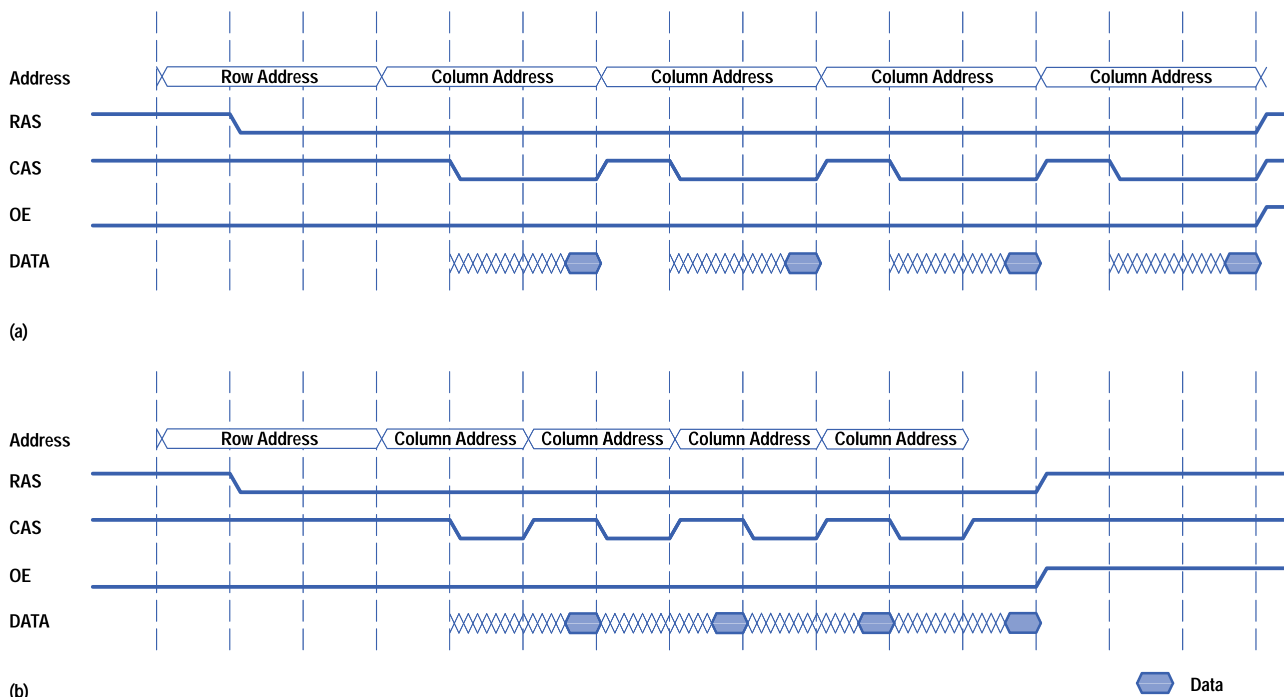


Fig. 9. *DRAM timing diagrams. (a) Page mode read. (b) Extended data out (EDO) mode read.*

Error-Correcting Code. The state of DRAM memory cells is susceptible to corruption from incident energetic atomic particles. Because of this, the PA 7300LC main memory controller optionally generates and checks an error-correcting code. The code is generated over a 64-bit data word. Any single-bit error within the 64-bit data word can be corrected. All double-bit errors and all three- or four-bit errors within an aligned nibble can be detected. The aligned nibble capability is useful since memory systems are typically built with four-bit-wide DRAMs. The nibble mode capability allows detection of the catastrophic failure of a single four-bit-wide DRAM. Whenever an error is detected, data and address logging registers are activated to support efficient fault isolation and rapid field repair.

Shared SLC and Main Memory Data Bus

From a cost perspective, it was desirable to share the large data buses needed for the SLC and main memory, thereby lowering the pin count of the PA 7300LC. However, sharing the large load from main memory DRAM cards would have significantly impacted the speed of SLC operations. The solution to this problem resulted in using an FET switch to isolate the main memory load from the SLC bus when the SLC is driving the bus, but to allow the bus to be shared when main memory is being accessed (see Fig. 10). The FET switch is a relatively inexpensive industry-standard part, which has a propagation delay of less than 1 ns when in the on state.

FET Switch. The FET switch also enabled us to connect the PA 7300LC to legacy 5-volt DRAM cards. The PA 7300LC operates at 3.3 volts and is not tolerant of the 5-volt logic swing of many existing DRAM cards. Biasing the gate of the FET switch to a voltage lower than 5 volts effectively limits the voltage swing from DRAM cards to 3.3 volts when seen by the PA 7300LC.

Chip Layout Challenges

Although the MIOC is a small part of the PA 7300LC, it controls nearly all of the I/O pins. Because the pins are located at the chip perimeter, long signal routes from the MIOC to some pins are unavoidable. Separating the MIOC into several blocks that could be placed near the chip perimeter and controlled remotely helped manage this problem. In particular, the data flow across the shared SLC and main memory data bus is completely predictable (because there are no slave handshakes from the memories), making the memory data interface the ideal block to be controlled from the other side of the chip.

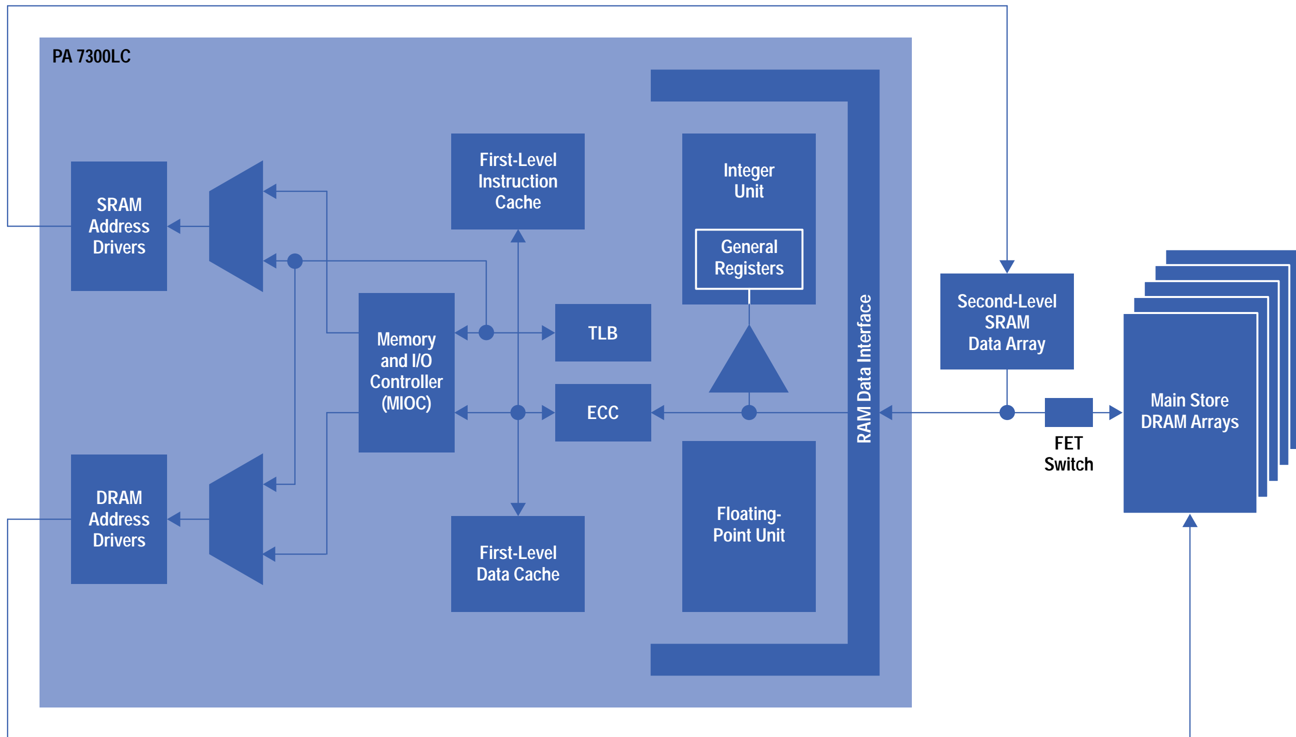


Fig. 10. *PA 7300LC block diagram showing the position of the FET switch.*

Cache Miss Data Flow

The MIOC is highly optimized for satisfying CPU cache misses. Although DMA transaction processing is handled efficiently, system performance is more sensitive to CPU cache miss performance than DMA performance.

When idle, the SLC and main memory controllers pass through physical addresses that are coming directly from the TLB and going to the SLC and main memory address pads. On the cycle following each address, the CPU core indicates whether that address resulted in a miss in the first-level cache. If a miss occurred, then an access is initiated and a cycle is saved by having passed along the physical addresses to the SLC and main memory.

For copyins, the SLC begins an access. The tag and data array are accessed in parallel. If there is an SLC hit, then data is returned to the processor.

On an SLC miss, the SLC data array data drivers are disabled, the FET switch is closed, and control is transferred to the main memory controller.

When a transaction is received by the main memory controller, it endeavors to activate the correct DRAM page. This may be as simple as issuing a row address strobe (RAS) with the proper row address, or may require deasserting RAS, precharge, and a new RAS. The memory controller sequences up to the point at which it is ready to issue a column address strobe (CAS) command, waits there until the SLC misses, and switches control over to complete the CAS command. However, if the SLC hits, it will wait for the next transaction and start the cycle again. Performance is improved by starting the DRAM access in parallel with the SLC access.

In the case of an SLC miss, once the main memory controller has control, it issues the proper number of CAS cycles to read the data. As the data passes the SLC, it is latched into the SLC data array. At the end of the cycle, the FET switch is opened, the SLC drivers are enabled, and the next transaction is processed.

Reducing Low-Miss Latencies. Much of the work described above concerns reducing miss latencies. This is important because even though the PA 7300LC CPU core has a non-blocking cache, load use stalls still develop quickly for many instruction sequences. Low-miss latencies minimize the impact of these stalls, which results in better overall performance. At CPU clock rates of 160 MHz, the PA 7300LC, as seen by the CPU pipeline, is capable of SLC hit latencies of three cycles with industry-standard 6-ns asynchronous SRAM. Main memory latencies can be as low as 13 cycles with 50-ns DRAM. Many single-cycle latency reductions have been implemented in the PA 7300LC; each by itself would not have much impact on overall memory access latency, but taken together, they make a significant difference.

I/O Interface

The PA 7300LC contains interface logic that allows direct connection to HP's high-speed GSC I/O bus. This interface processes I/O requests from the CPU core and DMA requests from GSC I/O bus devices.

Programmed I/O. Programmed I/O allows load and store instructions from the CPU core to communicate with the I/O subsystem. From a performance perspective, programmed I/O writes to graphics devices are important for many workstation applications. The improvements made for graphics performance in the PA 7300LC are described later in this article.

DMA Interface Controller. The DMA interface controller is designed to minimize main memory controller traffic and to reduce DMA read latency. The DMA interface controller employs three 32-byte line buffers. When servicing any DMA read, the controller requests 32 bytes from main memory and puts the data into one of the buffers. DMA requests on the GSC bus may be 4, 8, 16, or 32 bytes long. Since most DMA requests are to sequential addresses, requests less than 32 bytes can probably be satisfied from data contained in the buffer without issuing another request to the main memory controller. The DMA controller is also able to prefetch the next sequential line of information to increase the chances that DMA read requests are serviced from the DMA buffers.

GSC Write Requests. Writes are collected by the DMA hardware and passed on to the main memory controller. GSC write requests of 32 bytes are sent directly to the controller, but when possible, smaller-sized writes are collected into 32-byte chunks by the DMA controller to allow the main memory controller to access memory more efficiently.

Improvements for Graphics Applications

Graphics performance depends on many aspects of the system design. In addition, graphics workloads are sensitive to the system architecture. For the PA 7300LC, we chose to optimize the design for engineering graphics, where the typical workload involves rendering an object to the display device.

From a high-level point of view, the process of rendering an object can be divided into three steps:

1. Traversing the display list that describes the object
2. Clipping, scaling, and rotating the object with the current viewpoint
3. Transforming the object from primitive elements, such as polygons, into pixels on the screen.

This process can be partitioned in different ways. With today's powerful CPUs, the most cost-effective method is to store the display list in the computer system's main memory. The host CPU performs the display list traversal and the clipping, scaling, and rotation steps, and then passes primitives to dedicated graphics hardware for conversion into onscreen pixels.

Graphics Requirements. Several different models, including specialized CPU instructions and DMA engines, have been used to extract data to be rendered from main memory. While these approaches work, they incur the undesirable cost of specialized driver software that doesn't port well between processor generations. Starting with the PA 7100LC, the philosophy has been to support the graphics requirements within the existing architecture as much as possible. For example, the PA-RISC architecture defines a set of 32-bit integer unit general registers and another set of 64-bit floating-point unit general registers. Loads and stores from either set can be made to memory space, but only integer register loads and stores were architecturally defined to I/O space.

Starting with the PA 7100LC, floating-point register loads and stores to I/O space have been implemented. This has yielded improved performance because a single load or store can now move 64 bits and because more registers are available for operations that communicate with I/O space.

In contrast with specialized operations, extensions within the architecture are generally applicable and carry forward into future generations. These optimizations can also be used to benefit workloads other than graphics.

Graphics Optimizations. Several of the optimizations made in the PA 7300LC to further improve graphics performance include:

- A large I/O store buffer
- A relaxation of the load and store ordering rules
- The elimination of a CPU hang cycle previously needed for I/O stores
- Improvements to the GSC I/O bus protocol.

The structure of industry-standard graphics libraries leads to bursty graphics I/O traffic. The bursts are of many different sizes, but the most common burst is a write of 26 words. The PA 7300LC CPU core-to-I/O interface implements a large write buffer and can accept up to 19 double-word writes without stalling the CPU pipeline. This allows the CPU core to burst up to 19 double-word writes to the I/O subsystem, and then continue with its next task while the I/O interface is sending this data out to the graphics hardware.

Graphics Ordering. PA-RISC is a strongly ordered architecture. Strongly ordered means that all elements of the system must have a consistent view of system operations. In the case of graphics performance, this means that all buffered I/O stores must be observed by the graphics device before the CPU can access a subsequent piece of data in main memory. Hence, an I/O

store and a following memory read are serialized. A loophole to the ordering requirement was created for graphics. I/O stores within a programmable address range are allowed to be out-of-order with respect to the memory accesses. The graphics software takes responsibility for ordering when necessary.

Hang Cycle. Previous PA-RISC processors always incurred a minimum of one hang cycle for an I/O store. Extra logic was added to the data cache controller on the CPU core to eliminate this hang cycle.

Graphics Transfer Size. HP's high-speed GSC bus is used to connect graphics adapters to the PA 7300LC. The CPU sends data to the graphics device with I/O stores. In the PA-RISC architecture, I/O stores are 64 bits or less. The GSC is a 32-bit multiplexed address and data bus. Stores of 64 bits turn into an address cycle followed by two data cycles. At best the payload can be only two thirds of the maximum bus bandwidth. As mentioned above, the average transfer size to graphics is 26 words. Since these transfers are sequential, sending an address with every two words is unnecessary. Some form of address suppression or clustering of sequential writes was desired. Thus, the write-variable transaction was created.

Write-Variable Transactions. A new write-variable transaction type was created for the GSC bus. Write-variable transactions consist of an address and from one to eight data cycles. Since the PA 7300LC must be compatible with existing cards that do not implement the write-variable cycle type, the PA 7300LC only generates them in configurable address spaces.

With this protocol, the I/O controller blindly issues write-variable transactions for enabled I/O address regions. Starting with the initial write, as each write is retired from the I/O write queue, the I/O controller performs a sequentiality check on the next transaction in the queue. The process repeats for up to eight GSC data cycles. Maximum performance is achieved by allowing the I/O controller to begin issuing the write when the first piece of data becomes available.

The length of the transaction is limited to eight data cycles. Choosing eight data cycles is a good compromise between flow control issues and amortizing address cycle overhead with payload. The write-variable enhancement increased maximum CPU-to-graphics bandwidth from two thirds of the GSC raw bandwidth to 8/9 of the raw bandwidth. The PA 7300LC can easily saturate the GSC bus at 142 Mbytes per second compared with the 50 Mbytes per second achieved by the PA 7100LC with careful coding.

MIOC Summary. The MIOC implemented a number of features that improve system performance while keeping costs low, including:

- The second-level cache and main memory controllers are optimized to reduce the latency of copyin requests from the CPU core.
- The I/O controller improves graphics bandwidth and supports efficient DMA accesses through the use of buffers and prefetching.
- The MIOC is designed to be flexible, supporting a range of second-level cache sizes, a variety of industry-standard memory components, two different memory widths, and an optional error correction scheme.

Conclusion

The PA 7300LC design builds on the success of past processor designs and offers significant improvements in key areas. It features a superscalar CPU core, a large, efficient on-chip cache organization, tightly coupled second-level cache and main memory controllers, and bandwidth improvements for graphics. These features combined with frequency increases, extensive configurability, and high chip quality make the PA 7300LC attractive for a wide range of computer systems.

Acknowledgments

A large number of people were responsible for the successful design of the PA 7300LC, especially the design teams from the Engineering Systems Lab in Fort Collins and from the Integrated Circuits Business Division's Fort Collins Design Center. Many important contributions were also made by individuals from the Fort Collins Systems Lab, the Systems Performance Lab in Cupertino, the Computer Technology Lab in Cupertino, and other organizations within HP.

References

1. P. Knebel, et al, "HP's PA 7100LC: A Low-Cost Superscalar PA-RISC Processor," *Proceedings of IEEE Compcon*, February 1993, pp. 441-447.
2. S. Undy, et al, "A VLSI Chipset for Graphics and Multimedia Workstations," *IEEE Micro*, Vol. 14, no. 2, April 1994, pp. 10-22.
3. G. Kurpanek, et al, "PA 7200: A PA-RISC Processor with Integrated High-Performance MP Bus Interface," *Proceedings of IEEE Compcon*, February 1994, pp. 375-382.
4. E. DeLano, et al, "A High-Speed Superscalar PA-RISC Processor," *Proceedings of IEEE Compcon*, February 1992, pp. 116-121.
5. R. Lee, "Precision Architecture," *IEEE Computer*, Vol. 22, no. 1, January 1989, pp 78-91.

6. R. Lee, J. Beck, L. Lamb, and K. Severson, "Real-Time Software MPEG Video Decoder on Multimedia-Enhanced PA 7100LC Processors," ***Hewlett-Packard Journal***, Vol. 46, no. 2, April 1995, pp. 60-68.
7. M. Bass, T. Blanchard, D. Josephson, D. Weir, and D. Halperin, "Design Methodologies for the PA 7100LC Microprocessor," ***Hewlett-Packard Journal***, Vol. 46, no. 2, April 1995, pp. 23-35.

Timing Flexibility

Microprocessor design is a time-consuming and expensive process. Ideally, a design should scale through several fabrication process generations with low-investment algorithmic artwork shrunk to help amortize the cost of the original design.

Although it is relatively straightforward to increase the processor frequency, the frequency of interconnect to the rest of the system is more or less fixed. Typically the base processor design has the capability for a range of core-processor-frequency-to-interconnect-frequency ratios.

The PA 7300LC has three interfaces that are tolerant of increases in the processor frequency: the I/O bus interface, the main memory interface, and the second-level cache interface.

The cycle time of the general system connect (GSC) I/O bus can be configured to some multiple of the processor's cycle time. The I/O controller supports ratios from three to nine. The second-level cache controller can be configured to support a variable number of CPU cycles per second-level cache cycle. The controller supports two, three, or four CPU cycles per cache cycle. Similarly, the main memory controller can configure the setup and hold times of the DRAMs to be two, three, or four CPU cycles. Additionally, seven key DRAM timing parameters can be individually programmed.

As the processor gets faster, performance may improve but only as a sublinear function of processor frequency since memory and I/O performance remain constant. The large first-level caches on the PA 7300LC help insulate the processor from the effects of the relatively slow memory accesses, allowing the performance to scale well with increasing core processor frequency. The initial frequency target for the PA 7300LC was 132 MHz, but design ratios support core processor frequencies up to 360 MHz.

Two additional benefits are derived from the timing flexibility of the PA 7300LC. The increasing availability of higher-speed DRAMs and SRAMs makes it a simple matter to configure the timing generators to take advantage of these new components. Also, timing flexibility decouples the design effort from uncertainties that develop as RAM component vendors traverse their own development cycles.

High-Performance Processor Design Guided by System Costs

To minimize time to market and keep costs low, the PA 7300LC design was leveraged from a previous CPU, the chip area was reduced, cache RAM arrays with redundancy were added, and high-speed, high-coverage scan testing was added to reduce manufacturing costs.

by David C. Kubicek, Thomas J. Sullivan, Amitabh Mehra, and John G. McBride

While designing the PA 7300LC processor, the CPU team had to make design trade-offs between time to market, performance, and manufacturing costs. Occasionally these seemingly contradictory goals worked together to drive the team to a decision. More often, however, the team had to make hard decisions, weighing the benefits of each of the design goals.

This paper discusses the strategies used by the PA 7300LC physical design team to implement the design goals for the PA 7300LC.

Design Goals

One of the factors driving the design process was the desire to bring the product to market as fast as possible. To accomplish this goal, we employed three major strategies:

- Leverage as much as possible from the previous HP processors, including hardware, software, and methodologies for design and test
- Design quality into phase one, or the presilicon design stage, so that there would be fewer iterations of the design during phase two, after the first tape release
- Monitor project progress, avoiding any obstacles that might seriously impact or threaten our schedule.

Keeping the cost of the system as low as possible was another important goal of the project. Systems based on the PA 7300LC are meant to position HP in the low-to-midrange workstation market where prices are set by competition, not system cost. Therefore, savings in the system cost have a big impact on profit. To meet these aims, the team decided to:

- Integrate the first-level cache, a major system cost, into the processor, which had never been done before in an HP microprocessor
- Integrate the memory and I/O controller (MIOC), creating a system on a chip
- Reduce chip area to lower cost
- Add redundancy to the SRAM arrays on the chip, allowing some process defects to be repaired, thereby saving chips that would otherwise be thrown out
- Provide high-coverage, high-speed scan testing to lower the manufacturing cost of the processor.

Designs Leveraged to Minimize Time to Market

To reduce the time to market for the PA 7300LC, the CPU physical design team decided to leverage as many circuits as possible from the PA 7100LC. Except for the process shrink from CMOS26 to CMOS14, much of the superscalar integer data path on the PA 7300LC was leveraged from the PA 7100LC unchanged. Also, many of the cells used in the integer data path were used in other data path blocks on the chip. Although some of the circuits were reworked for speed improvements, the floating-point unit was also highly leveraged from the PA 7100LC. Furthermore, the floating-point unit was used in the geometry accelerator chip for the Visualize 48XL graphics product. This leverage strategy not only helped reduce time to market, but also split the design costs associated with the circuit between the ASIC and the CPU.

Control Blocks

While all of the control blocks leveraged from the PA 7100LC required some changes, much of the original control logic remained intact or was at least similar to the original code. This provided the opportunity to start the physical design early, providing the designers with the chance to work the bugs out of the tool flow, begin composition, and provide early feedback on difficult timing paths to the control designers.

For physical circuit layout, the control physical team initially used data scaled from the PA 7100LC in the CMOS26 process to the CMOS14 process. In several cases, the final artwork was almost entirely based upon the floor plan scaled from the

PA 7100LC. In other cases, the control equations were either vastly different (memory I/O control) or entirely new (the cache controllers), so we were unable to take advantage of earlier work.

In the case of the three main integer control blocks, the timing information and a significant portion of the control equations were usable. However, a study of interconnect between the three blocks indicated that they could be combined into a single block to simplify the design from a timing standpoint and to use global routing resources efficiently. By moving several hundred signals away from the center of the die into a more localized area near the integer data path, we also saved significant area.

Core Logic Library. While much of the logical design of the PA 7300LC was leveraged from the PA 7100LC, most of the standard cell libraries were borrowed from the PA 8000 project. The PA 8000 was fabricated using the same IC process technology as the PA 7300LC, but was farther along in the design cycle. The PA 7300LC team was able to use almost the entire PA 8000 core logic library unchanged. Unfortunately, a different clocking strategy meant that the driver library needed significant rework.

Standard Cell-Based Control Block Design. The use of a standard cell-based design for the control blocks, which was leveraged from the PA 7100LC, allowed great flexibility when fixing functional bugs, both in phase one (presilicon) and in phase two (postsilicon). During phase one, the standard cell approach permitted fairly quick turnarounds of a control block for rather complex changes. Often all that was required of the physical designer was to rerun the synthesis and routing tools, apply a few hand changes, and verify the design.

Use of Spare Gates. During the very late stages of phase one and all of phase two, the use of spare gates in the standard cell blocks allowed the physical designers to make logical changes by changing only the metal layers. One very late bug fix was made between the time the lower-level masks (e.g., diffusion, well, polysilicon) and the higher-level metal masks were released to the mask shop. Additionally, when phase two bugs were found, we were able to use the spare gates for metal-only changes. Because a number of wafers were held in the fabrication shop before M1 (the lowest level of metal) was placed, metal-only changes were run through the fabrication shop very quickly since the lower layers were already processed.

FIB Process. Another advantage of the metal-only changes was exploited during phase two. As control bugs were uncovered, we were able to rewire the logic using spare gates and the FIB (focused ion beam) process. The FIB process uses an ion beam to cut and expose various metal lines on a functional chip and to deposit platinum, reconnecting the gates into a new logic structure. A typical FIB repair is illustrated in Fig. 1. Use of the FIB process allowed the design team to verify bug fixes in a system that often ran at full operating speed. This resulted in a more complete functional verification, since tests run much faster in real silicon than in simulation.

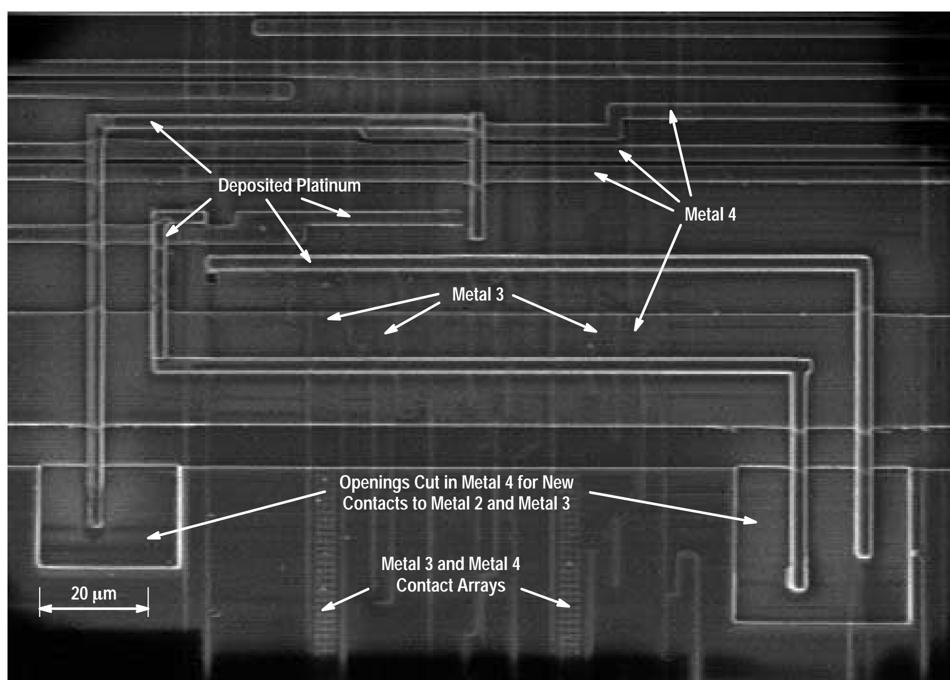


Fig. 1. *The photomicrograph shows a typical FIB (focused ion beam) repair. For this FIB repair, portions of the circuit of interest were located under a metal four power bus. Therefore, openings had to be cut through the power bus to access the circuits below. Notice how the platinum deposited by the FIB runs over top of the metal four, separated by passivation.*

New Tools. While the synthesis tool (Synopsys) and routing tool (Cell3 from Cadence Systems) were the same as on the PA 7100LC project, newer versions of these tools with additional features and problems were employed. The ability to work with the tools at an early stage allowed the physical control design team the chance to learn the strengths and weaknesses of the tools, so that they could be exploited or compensated for once full functionality was reached in the control equations. Even though new versions of these tools presented a few new problems, the basic method of operation was the same as for the PA 7100LC. Thus, use of these tools helped reduce our time to market by leveraging our previous experience with them.

Phase One Quality Equals Reduced Phase Two Debug Time

In addition to leveraging designs and methodologies, correct balance between the time and resources spent ensuring phase one quality and the time and resources spent finding functional problems in phase two can also reduce time to market. In this project, we gave great weight to ensuring phase one quality, since this would make debugging much easier in phase two. In return for our investment, the PA 7300LC had one of the shortest and smoothest phase two periods of all CPUs designed by HP.

Debugging Trade-offs

In phase one of the design cycle, simulation, emulation, and hand analysis are the key tools of the designer. With these tools, the designer can examine every detail of the design at any chip state and under any conditions.

In phase two, tests can be run much faster on a real chip than in simulation, accelerating bug detection. However, root-cause analysis of problems is a slow and difficult process because virtually all signals are hidden from the scrutiny of the designer. In addition, the signals that are available for the designer to examine are either chip I/Os, or are only indirectly available through scan paths. Hence, electrical phenomena such as glitches and power supply droops are not easily observed. Therefore, phase one debugging is a much simpler process because of the availability of detailed data about the internals of the chip.

Cross-Checking Designs to Improve Phase One Quality

To ensure phase one quality, each design on the PA 7300LC was subjected to a series of computerized automatic design checks, and then a set of manual checks was performed by designers who were not involved with the original design. These checks looked at:

- Circuit topologies
- FET size and connections
- Wire size
- Power routing
- Clock signal routing
- Signal coupling
- Signal types and timing.

Automated and Human Checks

For automated tests, the computer applied the same rules to each node on the chip quickly, without the bias that a human may have had. However, the computerized checks often generated spurious error messages and required significant human intervention to identify the real problems. Also, many rules were beyond the scope of computer algorithms and required human checks.

An example of a simple computerized check used on the PA 7300LC is a signal edge rate check. Every signal on the chip was checked against a set of criteria that depended on the signal context. The computer program blindly reported any signal that violated the specification set for that type of signal. It was the job of the designers to determine which errors flagged by the computerized check were real problems. The designers then fixed real errors and waived all others. Obviously, with this and all other automatic checks, a certain amount of skill and experience is needed to judge what constitutes a potential problem and what does not.

Because some quality checks do not lend themselves easily to computerized checking, each cell, subblock, and major block of the CPU had to be examined by an experienced engineer who was not the designer of the block. The cross-checking engineer had a list of guidelines to follow for checking each design, and any variance from these guidelines was discussed with the designer. This checklist was broken into categories so that the cross-checking engineer could focus on one particular area at a time, such as schematics, artwork, test, and so on.

An example of a check that is not automated is an artwork check, which ensures that all circuits have very solid power and ground networks. The subjective nature of this check makes it very difficult to implement with a computer check. Also, because of the subjective nature, the checking engineer must be very diligent about what constitutes a solid supply net and what does not.

Circuit Timing

When operating with a clock period of only a few nanoseconds, timing is of utmost importance as a phase one quality issue. Several different tools were used to this end, most notably Cadence's Veritime, EPIC's Pathmill, and HP SPICE (see Fig. 2).

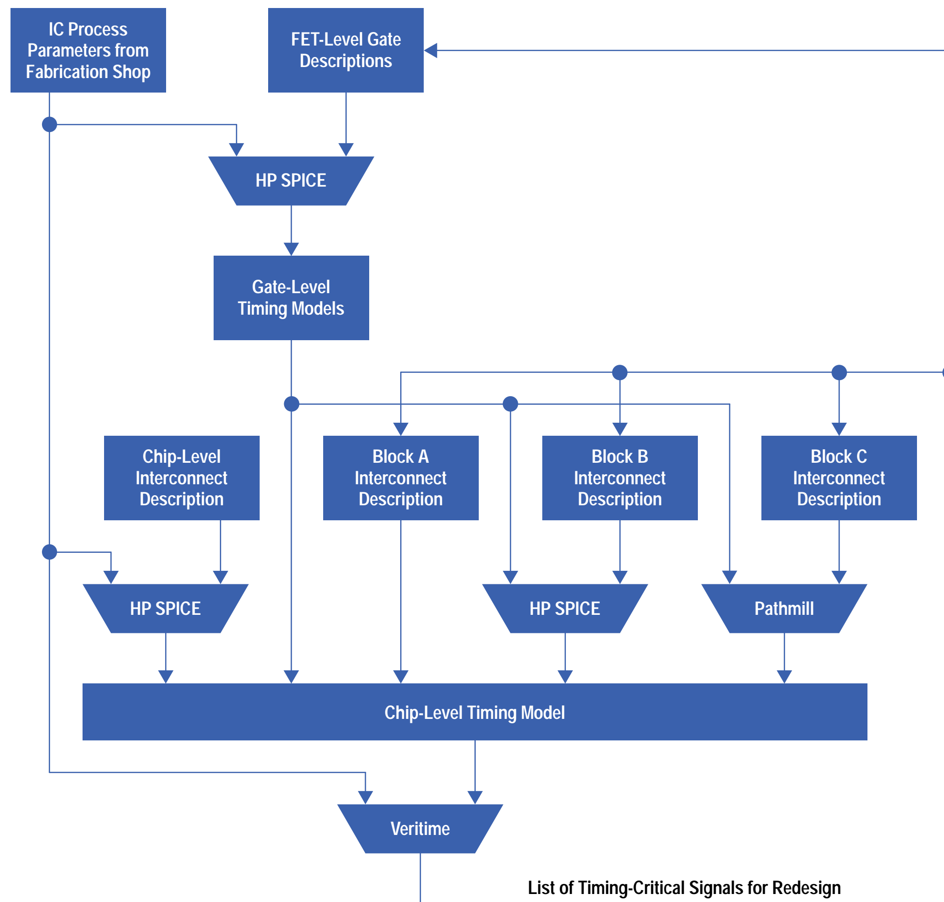


Fig. 2. *The chip modeling tools used for the timing simulation.*

Chip Model Tools. A Veritime model was generated and maintained for the top level of the chip. This model included either gate-level descriptions of blocks (generally for the standard cell blocks) or black box descriptions of blocks (for the custom data path blocks), as well as models for the delay due to the interconnects between blocks. On a regular basis, the timing team updated the model and performed timing analysis. The results were then given to the various block owners, who redesigned slow portions of critical timing paths.

HP SPICE and EPIC's Pathmill were used by a number of the custom data path designers to generate black box models of their blocks for the global Veritime model. Also, some designers analyzed larger standard cell blocks with Veritime. Additionally, a tool was developed that estimated the delays of all signal routes, which could then be hand-checked for anomalies.

Finally, HP SPICE was used extensively to simulate the timing of all major buses, many top-level routes, and other timing-critical paths. All elements of the standard cell libraries were also characterized with HP SPICE, using conservative parameters. While this approach caused a few more phase one headaches for the control designers, we uncovered no timing issues for standard cell blocks during phase two characterization.

Chip Composition Focused on Minimizing Cost

One of the ways that we were able to drive down the cost of systems that incorporate the PA 7300LC was to reduce the die size, thereby allowing more die per wafer in fabrication and improving yield. This was a key focus of the physical design group, and resources were dedicated to monitoring the impact of all changes on the manufacturing cost of the chip. We took several steps during phase one to ensure that the PA 7300LC was as small as we could reasonably expect.

Global Floor Planning. We started global floor planning and routing early in the design phase. Our initial floor plans, although they bear little resemblance to the final chip floor plan, provided the groundwork for early estimates on die size and feasibility. One of the early decisions was whether we would use three layers of metal, as on the PA 7100LC, or add a fourth metal layer. After extensive analysis, we concluded that, with only three metal layers, our stepper size would limit us to a

very small first-level cache, which would not meet our performance targets. So, we added the fourth metal layer. As it turned out, the fourth metal layer was essential to the success of the project for many other reasons, even though the decision was made over a year before tape release.

As the design matured, the floor plan and routes kept up with the changes and provided feedback on die size and potential timing problems. Fig. 3 shows the final floor plan. We made several major compositional changes early solely to remove congestion on the top metal layer and to compact the die area. These compositional changes included movement of a block in the data cache and changing the aspect ratio of our pad-ring bitslice.

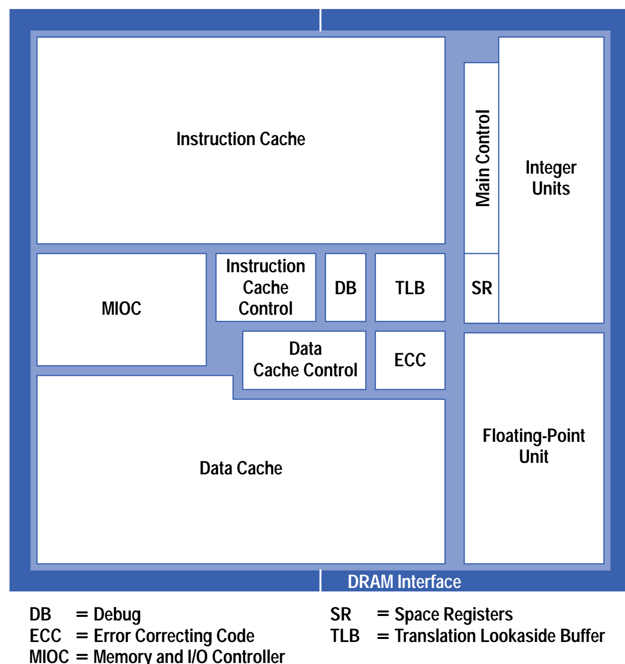


Fig. 3. *The final PA 7300LC floor plan.*

Using the "Dirty Trick." One of the opportunities we saw was in the composition of the data cache. Originally the data cache was designed to be completely symmetrical, with a right side, a left side, and a data path in the middle to merge the data from the two sides. The design essentially had three blocks on each side stacked from bottom to top: the data RAM array, the tag RAM array, and the dirty block array.

As we started routing the chip in its early phases, we saw that we had much more routing congestion in the channels above the right side of the data cache than above the left side. The channels above the right side led to the integer and floating-point units, while those above the left ran towards the memory and I/O controller (MIOC) (see Fig. 4a). The congestion on the right side of the data cache would have increased the height, while leaving unused area above the left side.

To deal with this congestion problem, we employed what we called the "dirty trick." The dirty bit block in the data cache is used to store one bit of information for each line in the cache. This bit tells the processor whether the information contained in that cache line has been modified by the CPU and is therefore dirty. In our original conception, each side of the cache had its own dirty block, which consisted of one bit of information per cache line and an address-to-cache-line decoder, the latter being ten times larger.

By putting both dirty data bits on the left side of the data cache and sharing one address decoder, the left side of the data cache grew by one bit of information, but the right side shrank by one bit of information and one address decoder (see Fig. 4b). This was a big win as it allowed the die size to be shrunk by the height of the dirty block. If we had not floor planned and routed early in the design phase, we would not have seen this opportunity in time to act on it and reduce the die size.

Outer Ring of Pads Limited. We were not in the clear yet, however. After all of this work on driving down the die size, we entered an interesting situation. Even though we were able to shrink the core dimensions, we were now limited by the outer ring of pads that connect the die to its package. This was not an issue earlier, since we had fewer pads and a larger core, but as the project progressed we added pads and shrank the core until we reached this predicament. However, the I/O ring team elongated our bit slice in the ring slightly and narrowed the width considerably, allowing us to reduce our die size until we were once again limited by the size of the core. Again, this trade-off on the bit slice dimensions was not readily apparent at the outset of the project, but was obviously a big win when we analyzed the situation.

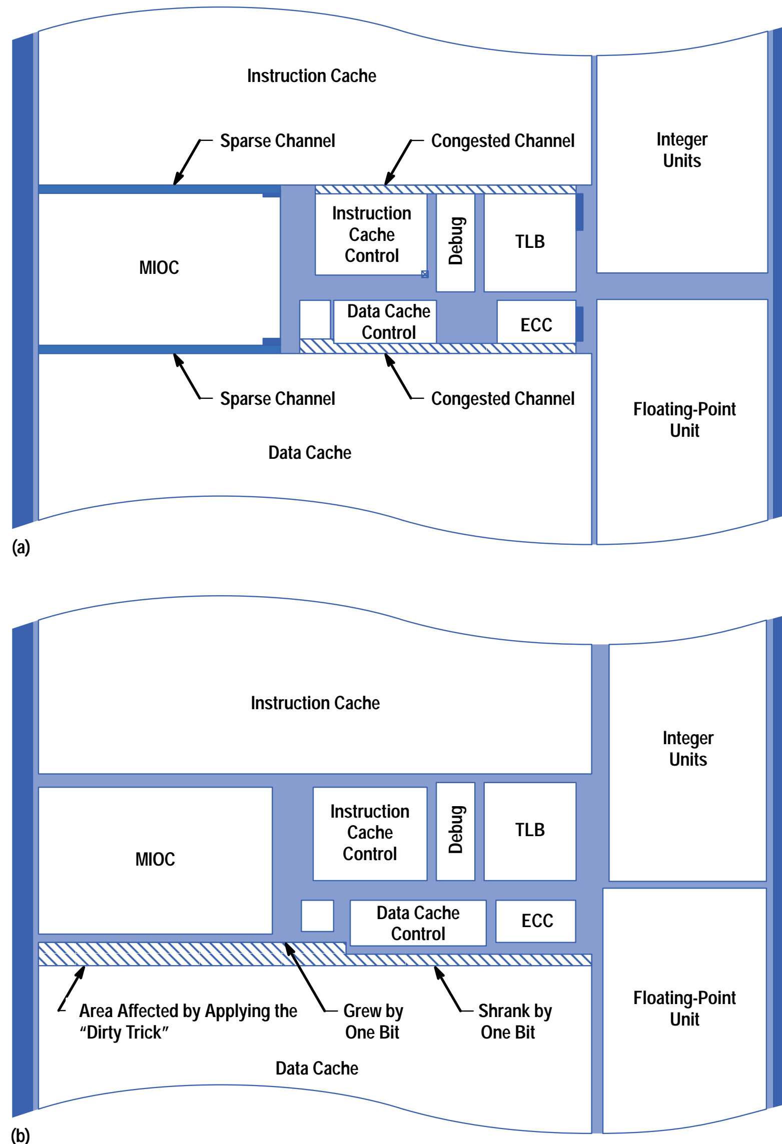


Fig. 4. *(a) The floor plan before applying the dirty trick. (b) The floorplan after applying the dirty trick.*

Metal-Four Routing. The last obstacle came after we finished automated signal routing. The router we used, HARP,* was designed for the three-metal-layer process used on the PA 7100LC and so it was unable to automate the fourth metal layer. It was a channel-based router, which allowed the block designers to use all three metal layers within their block boundaries, but required us to leave areas free between the top-level blocks for the interconnect. We used HARP to connect signals between top-level blocks, but we left major buses, power connections, clocks, and speed-critical signals for the hand-routed fourth metal layer. This meant, however, that any layer-four metal used within the blocks could interfere with the global metal four, which we planned to run over the blocks on the chip, not merely in the channels. Therefore, from the outset of the project, metal four was under the ownership and control of the composition team.

The cache array designers were given full control of layer-four metal in their areas, but all other block designers proceeded as if they only had three metal layers. As the global metal-four floor plan matured, metal four was released to the block owners to reduce area in places that did not conflict with the global route. In all other cases, the block owners were constrained to use the lower three metal layers instead of placing obstructions to the global metal-four route, even if it meant growing their blocks.

This stingy allocation of metal four became very important as new buses and timing-critical signals were promoted up to the metal-four "superhighway." Near the end of the project, the connection of the metal-four power buses to the top-level blocks became more and more challenging, and would have been impossible if not for the freedom retained by keeping metal four clear of obstacles.

* HARP (Hewlett-Packard Automatic Routing Program) is an internal routing tool that was leveraged from the PA 7100LC toolset.

Leaving the flexibility to make last minute changes was critical to meeting our die size commitment. Since, at that point in the project, our packages had been ordered with a specified die cavity, changing would have had serious financial and schedule implications.

Practice Runs

The PA 7300LC team used several techniques to ensure that the project would proceed as smoothly as possible. These techniques included building an SRAM test chip and doing a mock tape release.

Using a Prototype Chip. Before the PA 7300LC, HP had never produced a CPU with any significant amount of on-chip memory. How could we ensure that the cache would work in first silicon? Without a working cache, running test code in an actual system would not be practical. To help ensure a working cache in first silicon, we designed and built an experimental memory chip, featuring various RAM cell topologies. This test chip provided a large amount of visibility into the workings of the RAM cells. It also proved to be an excellent tool for analyzing the workings of the on-chip cache. Because the RAM design was effectively "phase two verified" during phase one of the CPU design cycle, the PA 7300LC on-chip cache worked in first silicon, greatly easing the time and resources required for phase two debugging of the rest of the CPU.

Mock Tape Release. Tape release of a CPU is quite a complicated process, involving several steps of database copying, verification, translation, and data transmission. Also, it does not lend itself to leveraging. Any one of these steps could cause a delay of several days in fabricating the chip. Therefore, we performed a mock tape release in which each step was executed as if it were part of an actual tape release. The only exception was that the data used was not the final, fully designed CPU. When the time came to do the actual tape release, the process went very quickly and smoothly.

SRAM Redundancy Improves Yield

With about eight million of the nine million transistors on the PA 7300LC, the cache is the most likely block on the chip to fall victim to a fabrication defect. Therefore, we added redundant blocks of four columns each in the SRAM, so that a block that contains a fabrication defect can be replaced with a functional block via a set of multiplexers. Fig. 5 illustrates the shifted-block method used to replace the defective block with a redundant block.

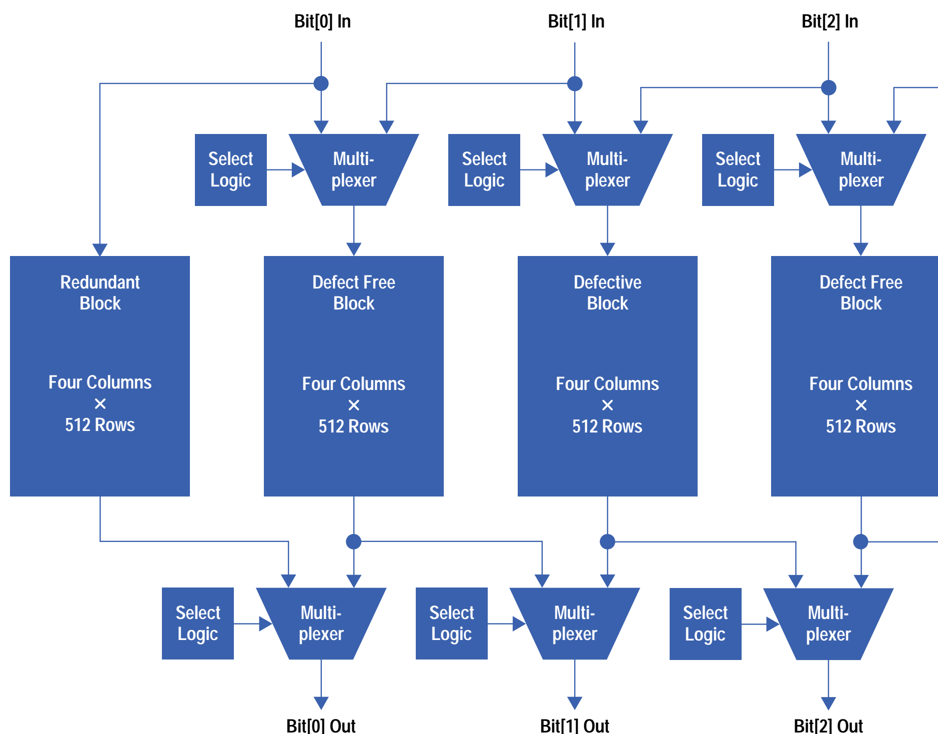


***Fig. 5.** The shifted-block column replacement method.*

The select logic on the multiplexers shown in Fig. 5 is controlled by a fuse that can be blown with a laser to deselect the failing block of columns and select an adjacent block. The adjacent block's multiplexer must also be programmed to select the next block. This ripple continues until one of the redundant blocks has been substituted into the RAM array. By substituting an adjacent block rather than immediately substituting the redundant block for the defective block, timing changes are minimized.

Adding a Serial Number

One of the new features incorporated on the PA 7300LC is a serial number individually programmed onto each die by the same laser that programs the redundancy selection multiplexers for the on-chip cache SRAM. As wafers are put on the laser for cache redundancy programming, we are able to blow the serial number fuses at the same time. The serial number feature was added to the production flow with very little overhead. Fig. 6 shows a set of serial number fuses. The serial number was added to the PA 7300LC because:

- It provides the ability to track any given die back to its original lot, wafer, or die designation, so we can analyze information gathered on the die at wafer test and at initial package test. On previous microprocessors, we were unable to track backwards in this fashion.
- It allows the design team to select specific dice off a wafer without having to remove the whole wafer from the production flow. This makes it much easier to grab interesting parts for further characterization.
- It provides a convenient way to refer to and classify production parts. The serial numbers became an invaluable part of the phase two debug effort, because we were able to know the history of the part we were debugging.

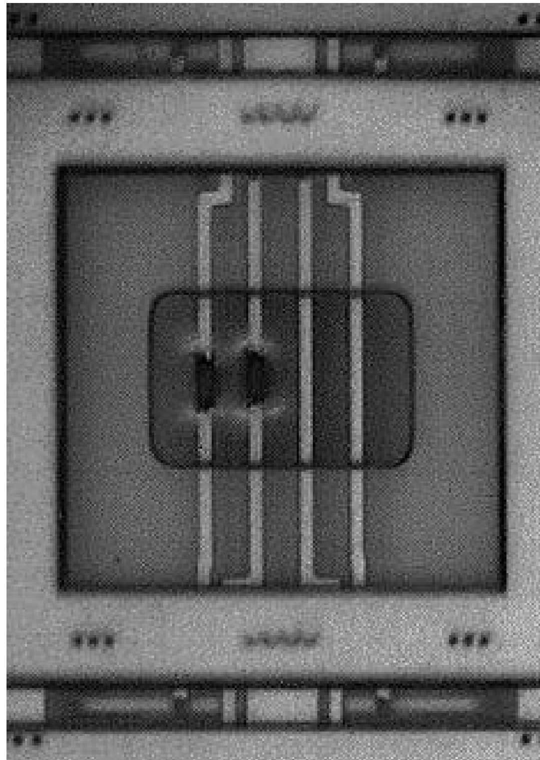


Fig. 6. *Two of the serial number fuses in the photomicrograph above have been blown by the laser. The other two are left intact.*

High-Speed, High-Coverage Testing Reduced Manufacturing Cost

Both broadside parallel and serial scan tests were used to test the PA 7100LC. Many of these tests were leveraged for use with the PA 7300LC. Some tests were simply copied from the PA 7100LC test suite and reformatted for use with the PA 7300LC. These tests included legal PA RISC assembly code for parallel vectors and serial scan tests of highly leveraged blocks, such as the integer data path.

Other tests required small changes. For instance, TLB tests on the PA 7100LC involved writing and then reading a variety of values for each TLB entry. Then the test simply looped through this process for each of the 64 entries in the TLB. Thus, to test the PA 7300LC's 96-entry TLB, we merely changed the loop value from 64 to 96 entries and reformatted the test.

Automated Test Generation. While many of the highly leveraged custom data path blocks could use scan tests leveraged from the PA 7100LC, this was not the case for the logic-synthesized standard cell blocks because any logic change rendered the old tests useless. Fortunately, the use of an automated test generation tool allowed the PA 7300LC team to have a significant portion of the serial tests written before we received first silicon. Shortly thereafter, we completed the rest of the serial tests, with high fault coverage. The control block test efforts were also helped by widespread use of state memory latches which were controllable and observable via serial scan testing.

Manual Test Generation. For custom data path blocks that were not leveraged, such as those in the MIOC and cache controller blocks, block designers wrote tests by hand, ensuring that each transistor in their design would be tested. Often, this daunting task was aided through the use of Perl* scripts to help generate the test vectors. Thus, many circuit designers found themselves becoming part-time software developers until their block tests were written.

Verifying Block Tests. As block designers began generating serial tests, the ability to verify these tests became an issue. Simulating a single block test on a model of the chip would take anywhere from a few minutes to several hours. However, a real chip could run even the largest test in just a few seconds. Therefore, a way to verify the block tests on an real chip could save a lot of simulation time without compromising test quality.

However, the testers used to test these chips in manufacturing were not readily available. Furthermore, they were too expensive to use for this purpose. Since we were running serial block tests, we only needed to control the chip's serial test port pins. The other chip I/O pins could be tied to ground.

Fortunately, we had decided to make our serial test port comply with the JTAG (IEEE 1149.1) industry standard. This meant that a relatively inexpensive test port interface was readily available. We purchased a JTAG Industries PM 3720 and built what we called a "bench tester" around one of these interfaces. Fig. 7 shows a block diagram of the bench tester.

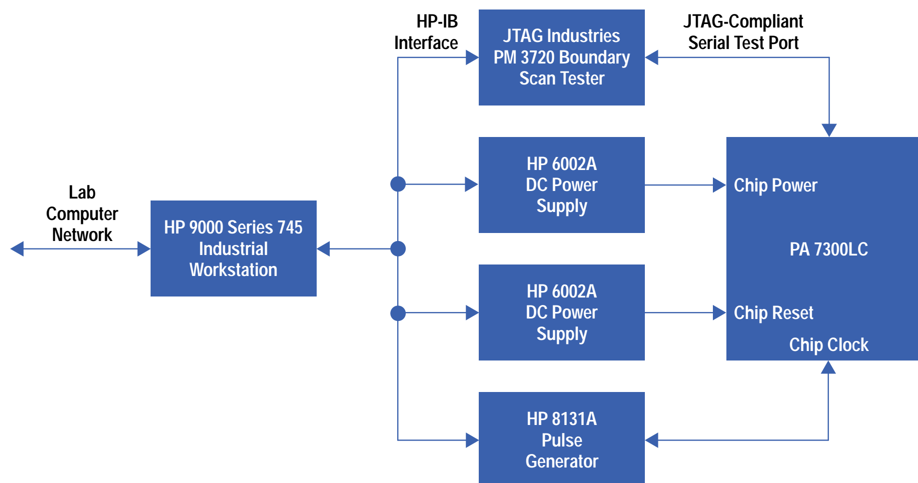


Fig. 7. The JTAG serial port bench tester.

We fed the chip power and controlled the reset pins with a couple of old HP 6002A dc power supplies. The system clock was provided by an HP 8131A pulse generator. Finally, all of these components were controlled via the HP-IB and connected to the lab's computer network through an HP 9000 Model 745 industrial workstation. The network connection allowed designers to run tests and monitor results from their desks, or even from home.

Many block designers pushed the bench testers well beyond their original intended use. By the end of the project, they could be used to create voltage-versus-frequency shmoo plots as the tests were executed over a range of power supply and clock frequency values. We even engineered a way to execute a loop of code in the instruction cache with no other system support logic, proving that the PA 7300LC is truly a system on a chip.

* Perl (Practical Extraction Report Language) is designed to handle a variety of UNIX® system administrative functions.

Summary

In conclusion, the PA 7300LC design team owes much of its success to previous project teams. Our aggressive time-to-market goals were met not only because of circuit leverage, but also because of methodologies from previous projects. Also, an early focus on quality prevented a lot of rework at the end of the project. Excellent performance from this highly integrated processor gives HP a competitive advantage in the cost-sensitive, performance-hungry market for which it was designed.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

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

Verifying the Correctness of the PA 7300LC Processor

Functional verification was divided into presilicon and postsilicon phases. Software models were used in the presilicon phase, and fabricated chips and real systems were used in the postsilicon phase. In both phases the goals were the same—to find design bugs and ensure that customers get the highest quality part possible.

by Duncan Weir and Paul G. Tobin

Ensuring the correctness of the complex PA 7300LC design required an extensive verification effort. We wanted to ensure that no customer would ever encounter a design bug. To reach this goal, we set out to exercise the design more extensively than is done with user software. Previous HP processors have maintained a well-earned reputation for quality, and we wanted the PA 7300LC to meet or exceed the quality of its predecessors.

This paper discusses the methodology used to verify the correctness of the PA 7300LC and the diagnostic hardware incorporated into the design to support debugging.

Functional Verification

The functional verification effort was divided into presilicon and postsilicon phases. The presilicon phase involved creating a software model of the chip and an environment in which the model could be thoroughly tested and debugged. The modeling environment provided many features to aid verification including the ability to initialize the machine state, inject stimuli, and see into all portions of the design for debugging. One major drawback of the modeling environment was the slow simulation speed.

Complementing the presilicon effort, an extensive postsilicon verification program was completed that took advantage of the test throughput available when running on an actual computer.

Extensive testing of the physical circuit design of the PA 7300LC was done in presilicon and postsilicon environments to ensure that the circuits would meet frequency, voltage, and temperature targets. This topic is covered in ***Article 8***.

Presilicon Verification

For better efficiency, we chose to divide the design of the PA 7300LC into two components: the CPU core and the memory and I/O controller (MIOC). These two portions of the design were logically separated by a well-documented interface that enabled us to verify each component independently. Verifying the two components independently provided several benefits:

- Smaller and faster models
- Precise control over the stimuli at the CPU-MIOC interface
- Simpler model management (because less coordination was needed)
- Reduced debugging time (since it was known which portion of the design contained the bug).

As the design neared completion and both the CPU and MIOC had been extensively verified, we created a single merged model that included both components. This provided a thorough check of the interface between the components and was a double check of the independent verification work. In addition, the MIOC was incorporated into a model with external I/O devices to ensure that the PA 7300LC design would work with the components needed for a complete computer system.

The presilicon verification environment consists of three parts: modeling environment (model), test case environment (stimuli), and checking environment (checks).

Modeling Environment

We modeled the PA 7300LC design using the Verilog hardware description language. The design was primarily modeled at the logic gate level with connectivity extracted from the physical design. Some key portions of the design like the caches, TLBs, and floating-point execution units were modeled at a higher level to improve the size and speed of the model.

Fig. 1 shows the CPU and MIOC modeling environments. Software emulators were connected to the model interfaces to provide input and respond to output from the model. The programmable nature of the emulators allowed test cases to exercise the interfaces fully.

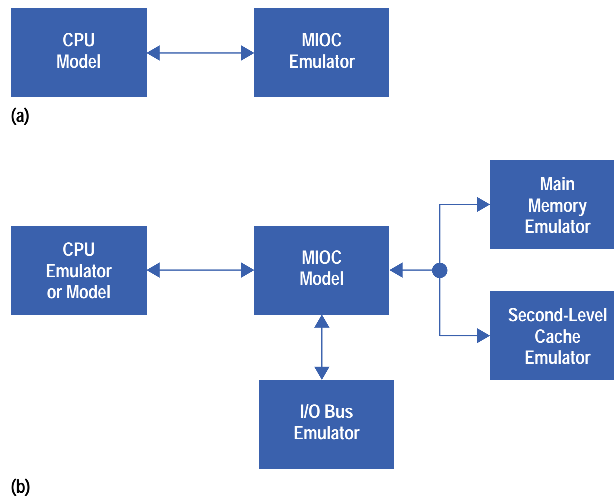


Fig. 1. *Presilicon verification modeling environments. (a) CPU modeling environment. (b) Memory and I/O controller (MIOC) modeling environment.*

New Modeling Process. Managing the modeling environment of a large design is a time-consuming task requiring coordination among all team members. Problems with a model build could lead to downtime that would stall the verification effort. To minimize these problems, a new model building process was implemented for the PA 7300LC design. All blocks of the modeling environment were placed under revision control. Any changes had to be included in a process change order that documented the purpose of the change, the blocks affected, the dependencies existing between this and other process change orders, and the testing needed to verify the change. In addition, an automated model build procedure was put in place to allow designers to integrate their changes into a private copy of the model and verify them in isolation before submitting a process change order. Finally, before a model was released to the verification team, it would undergo regression testing to eliminate blatant errors. Using the new system resulted in a consistently stable model that accelerated the verification effort.

Test Case Environment

Test cases control the stimuli applied to a model, thereby providing the event interactions that stress the design. Having an efficient way for test cases to stress the entire design is an important factor for improving quality. The strategy used for the PA 7300LC was largely leveraged from the successful PA 7100LC effort.[1] It provided a simple way to initialize machine-state resources like registers, caches, TLBs, and memory. It also allowed high-level coordination of instructions executed by the CPU along with transactions occurring at the model interfaces.

Test cases for the PA 7300LC came from three sources: cases leveraged from the PA 7100LC, new cases focused on the PA 7300LC, and randomly generated cases.

Thousands of cases that were written to cover the PA 7100LC design were leveraged to run on the PA 7300LC. Most cases needed no modifications to be effective because of similarities in the designs of the two chips. For the portions of the PA 7300LC design that were different, new cases were produced. Some of these cases were written to focus on particular aspects of the design such as instruction-cache misses, the CPU-MIOC interface, and the second-level cache. Other cases were produced using random code generators that were designed to stress the PA 7300LC.

Random code generators are mainly employed for postsilicon verification, but the PA 7300LC team also emphasized their use for presilicon testing. Although challenges were encountered, the results were positive. Many subtle bugs that might not have been found until postsilicon testing were discovered early in the design process. Random code generators also provided an efficient way of achieving broad coverage with fewer engineers than other testing methods. See ***Subarticle 9a*** “Random Code Generation” for more on this topic.

Checking Environment

A modeling environment and interesting stimuli are only two pieces of the verification puzzle. The other critical piece is verifying the model’s response to stimulation. On a complex design like the PA 7300LC, with many designers and tens of thousands of test cases, it would have been impossible to verify correct model behavior without aids to automate the process. As a result, a significant part of the PA 7300LC verification effort was spent creating software modules that automatically verified the model’s response to events created by test cases.

Modules were compiled into the model to check that the MIOC followed the proper I/O bus protocol and to ensure that both the CPU and the MIOC followed the protocol at the CPU-MIOC interface. Checkers were also written to ensure that the memory controller obeyed proper timing protocol on the main memory and second-level cache buses.

CPU Testing. For the CPU core, we linked a PA-RISC architectural simulator to run synchronously with the model to ensure that instructions were executed as the architecture requires. When an instruction finished executing, the results were compared between the model and the simulator. A special module called a *depiper* was written to translate internal CPU signals into architectural events that could be checked by the simulator. After a test case finished, the model's final machine state was compared against the simulator's final machine state.

New Transaction Checker. Logically, the MIOC converts inbound transactions on one interface to outbound transactions on a different interface. For example, the CPU core might initiate a cache line copyin that the MIOC converts to a read on the memory port. When the memory supplies the data, the MIOC returns the cache line to the CPU. A special transaction checker, called the *metachecker*, was written to verify that proper transaction conversions occurred. The metachecker matched inbound transactions with their associated outbound transactions. Mismatched transactions were reported as failures.

New Cache Checker. The cache controllers for the PA 7300LC are among the most complex portions of the design. As a result, a checker was written to verify their operation. It monitored the instruction pipeline, the cache read and write ports, and the CPU-MIOC interface. Any incorrect behavior was detected and reported.

Ad Hoc Checks. Finally, a collection of small, ad hoc checks were included in our presilicon testing to cover things that might otherwise be missed. Some were signal-level checks (for example, checking that a set of signals were mutually exclusive), others were special checks required by test cases. Some checked that performance features such as the superscalar pipeline were operating correctly.

Together, the checkers formed a seamless net to ensure that incorrect model behavior would be detected. There was some overlap between the checkers. Many times a design flaw would get flagged by several of the checkers, but dividing the work between multiple checkers was an effective way to reduce the risk of a design flaw escaping detection, while allowing verification engineers to work in parallel.

Model Matches Physical Design. Once the physical design and the verification effort stabilized, we verified that the Verilog model matched the physical design. This was done by deriving a switch-level model from the actual chip artwork and running thousands of tests on both it and the Verilog model, comparing key signals on every clock phase of simulation.

Postsilicon Verification

Once the design is fabricated, the nature of the verification effort changes completely. The goals are still the same—to find the design bugs and ensure that customers get the highest-quality part possible, but the tools and the approach are different.

Test Systems. The environment for testing the design shifted from software models to real computer systems that included PA 7300LC chips. We set up a number of test systems, each of which could be controlled remotely from a host workstation using remote debugger software. The remote debugger provided us with the ability to load and run programs on the test system and to examine portions of the machine state. It also gave us complete control over the machine without any operating system layers obstructing our access to system resources.

Because the PA 7300LC is designed to work in a number of different system configurations, we set up systems that had different clock frequencies, cache configurations, and memory timing. To ensure that the design would work with a variety of different I/O cards, exercisers for the GSC I/O bus were created that could change their behavior to mimic any type of I/O card.

Random Code Generation. Random code generators are an efficient way to take advantage of the speed of postsilicon testing. With a small amount of human control, these programs can create millions of unique tests to exercise every nuance of a complex design. We used random code generation extensively on the PA 7300LC by employing six different generators. One targeted the floating-point design, one was directed at the MIOC, and four covered the entire chip operation.

Extensive Suite of Tests. We supplemented the random testing with an extensive suite of tests using I/O exercisers to stress the MIOC design. Many tests were leveraged from postsilicon testing of the PA 7100LC and were modified for the PA 7300LC. Additional tests were written to provide better coverage, especially for areas where the PA 7300LC design differed from the PA 7100LC.

Self-Checking Tests. The elaborate checking methodology from presilicon verification was of no use in postsilicon testing because it was not possible for the checking software to observe the design now embedded on a VLSI chip running in a system. To compensate, all of the postsilicon tests were self-checking. The generators that created the random tests also ensured that the chip responded properly to them.

System Test. A final element of the postsilicon testing was verifying that operating systems and application programs ran properly on computer systems built around the PA 7300LC. A large amount of testing was done by several different organizations within HP and included operating system reliability tests, benchmark programs, and key user applications.

Verification Results

The PA 7300LC verification work was a success. Presilicon testing eliminated over 800 design bugs, and more than 1200 process change orders were added to the model in one year. The quality of the first revision of the chip was very high. Only eight functional bugs were found in postsilicon testing. Of these, only one affected our design partners, and it had a simple workaround. The HP-UX* operating system was booted shortly after first revision parts arrived. Our postsilicon testing was far more extensive than what we had previously done with the PA 7100LC or its predecessors. The verification effort ensured that the PA 7300LC will maintain HP's reputation for quality processors.

Debug Support

The high level of integration on the PA 7300LC reduces the visibility into chip operation that aids in debugging prototype silicon. In particular, moving the primary caches onto the chip removed a valuable source of debug data while also introducing a new source of potential functional and electrical problems.

Since the MIOC, floating-point coprocessor, and TLB are also contained on the same die, the only external pads visible to debuggers were for the I/O bus and the memory interface. At the same time, the PA 7300LC had new challenges such as a large primary on-chip cache, a new IC process, higher operating frequencies, and a second-level cache. Debug support was important to improve the signal visibility and to reduce the risks associated with the new technology.

Debug Mechanisms

Signal visibility is of primary importance when debugging a failure, so several techniques were used to make internal signals accessible.

- Idle cycles on the GSC I/O bus were used to drive debug information.
- Seventeen special chip pads are dedicated to driving real-time debug information. To reduce cost, these pads are not bonded in production parts.
- Thorough implementation of IEEE 1149.1 and sample-on-the-fly (a scan technique invented for the PA 7100LC)[1,2] allowed a very broad, but only one-cycle-deep, snapshot of the chip state to be reported. Custom data capture hardware was designed to gather the debug traces and present them to a logic analyzer.

New Pattern Mapping Failure Isolation Technique

Traces captured from the debug ports can be overwhelming in size, making it difficult to isolate the failure. The PA 7300LC addressed this problem by implementing circuits to recognize internal chip state patterns. The patterns are programmed from software using special instructions implemented on the PA 7300LC, and the capture of debug traces can be predicated on a state pattern match. Debug traces are thus shortened to an interesting region. It is also possible to alter the program flow upon a pattern match, allowing a branch to diagnostic software to probe for a failure. By providing a flexible scheme for programming repeatable patterns, the task of isolating a failure and performing experiments to determine its root cause was greatly simplified.

Target Applications For Debug

Functional and electrical verification were the primary applications for which the debug circuitry was designed, but the debug features were general enough that they could be used to diagnose processor problems encountered during bringing up the operating system, firmware development, and benchmarking.

Electrical verification relies more extensively on debug hardware because failures cannot be reproduced in our software model of the CPU. Engineers working to verify a chip's thermal and electrical margins use debug features to investigate and understand failures occurring at extreme operating points.

Debug Features

The PA 7300LC debug features are intended to work in any environment used to test the CPU—wafer test, package test, and system test. The debug features are operable and portable across these environments. In addition, debug circuits were designed to tighter specifications than the rest of the PA 7300LC. This ensured that they functioned properly well into the operating regions where the CPU core is expected to fail. We achieved this through the use of simple logic and conservative timing budgets.

Although no major problems were found during qualification of the PA 7300LC, debug features were relied upon to help fix the problems that arose, helping us to achieve quick time to market for PA 7300LC-based systems.

Conclusion

The extensive verification of the PA 7300LC design was based on the successful strategy used for the PA 7100LC. Improvements were made in the model building process and in the extensive use of random code generation in the presilicon and postsilicon phases. Many features were added to the PA 7300LC design to allow efficient debugging of postsilicon failures. Together, these efforts ensure that customers get the highest quality part possible.

Acknowledgments

Many people contributed to the verification effort of the PA 7300LC including members of engineering systems laboratory in Fort Collins and the Integrated Circuits Business Division at the Fort Collins design center. Key contributions were also made by individuals from the Fort Collins systems laboratory, the computer technology laboratory in Cupertino, and the UNIX® development laboratory in Fort Collins.

References

1. M. Bass, T. Blanchard, D. Josephson, D. Weir, and D. Halperin, "Design Methodologies for the PA 7100LC Microprocessor," *Hewlett-Packard Journal*, Vol. 46, no. 2, April 1995, pp. 23-35.
2. *IEEE Standard 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture,* IEEE Standards Board, May 1990.

HP–UX 9.* and 10.0 for HP 9000 Series 700 and 800 computers are X/Open Company UNIX 93 branded products.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

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

Random Code Generation

The complexity of processor designs has increased dramatically in an effort to improve performance, reduce system cost, and allow processors to be used in more system configurations. The increasing complexity makes it almost impossible to identify the specific event cross-products that need to be tested to ensure that a design is correct. Random code generation is an effective method for testing a design without having to identify exactly what needs to be tested. A random code generator creates legal, random sequences of machine states and instructions that exercise a design more thoroughly than application software.

The term random is somewhat misleading—generating completely random machine states and instructions would result in uninteresting tests as far as stressing the design is concerned. Instead, generators focus on key aspects of the design while preserving an element of randomness. Accelerating rare events, hitting boundary conditions, and concentrating on instructions that exercise complex parts of the design are among the ways to focus a generator. The probabilistic distribution of random numbers creates interesting combinations of these focused events.

Although random code generation has higher coverage in postsilicon testing where the design can be tested at high speeds, it can also be effective in presilicon testing. When running on relatively slow presilicon models, the effectiveness can be improved by adding more elaborate checking strategies and focusing the generators on smaller portions of the design.

Some elements of a quality random code generator include:

- Coverage of the entire design
- Focus on complex portions of the design
- Low fault latency (i.e., a failure gets noticed soon after it occurs)
- Reproducible test cases
- Aids for debugging failing tests.

Random testing techniques can also be applied to designs other than microprocessors. Memory or I/O controllers can use these techniques to randomly generate machine state and transactions that will stress the controllers. Designing special-purpose bus exercisers that are controlled by random test generators can extend such testing into the postsilicon environment.

An Entry-Level Server with Multiple Performance Points

To address the very intense, high-volume environment of departmental and branch computing, the system design for the D-class server was made flexible enough to offer many price and performance features at its introduction and still allow new features and upgrades to be added quickly.

by Lin A. Nease, Kirk M. Bresniker, Charles J. Zacky, Michael J. Greenside, and Alisa Sandoval

As the computer industry continues to mature, system suppliers will continue to find more creative ways to meet growing customer expectations. The HP 9000 Series 800 D-class server, a new low-end system platform from HP, represents a radically different approach to system design than any of its predecessors (see Fig. 1).

Fig. 1. *The D-class server system cabinet. The dimensions of this cabinet are 10.2 in (26 cm) wide, 23.8 in (60.4 cm) high, and 22.2 in (56.4 cm) deep.*

The Series 800 D-class server comes at a time when server systems priced below U.S. $20,000 are at a crossroads. Commodity technologies are upsizing, enterprise customers are enjoying choices of product families that offer thousands of applications, open computing and networking have blurred the distinctions between competitors' offerings, and finally, indirect marketing and integration channels can offer compelling value bundles that were once the exclusive domain of big, direct-marketing computer system suppliers. These trends have created an environment of very intense, high-volume competition for control of departmental and branch computing.

To address this environment, the system design for the D-class server had to be flexible enough to offer many price and performance features at its introduction and still allow the addition of new features and upgrades to the system quickly. The server's competitive space, being broad and heterogeneous, also demanded that the system be able to accommodate technologies originally designed for other products, including technologies from systems that cost more than the D-class server.

System Partitioning Design

In designing the D-class entry-level servers, one of the primary goals was to create a new family of servers that could be introduced with multiple performance points without any investment in new VLSI ASICs. The servers also had to be capable of supporting new advances in processors and memory and I/O subsystems with minimal system reengineering.

The server family would cover a span of performance points that had previously been covered by several classes of servers which, while they were all binary compatible with the PA-RISC architecture, had very different physical implementations. The lower-performance-point designs would be drawn from uniprocessor-only PA 7100LC-based E-class systems. The upper-performance-point designs would be drawn from the one-to-four-way multiprocessor PA 7200- based K-class systems. The physical designs of these systems varied widely in many aspects. Issues such as 5.0V versus 3.3V logic, a single system clock versus separate clocks for I/O and processors, and whether I/O devices would be located on the processor memory bus or on a separate I/O bus had to be resolved before the existing designs could be repartitioned into compatible physical and logical subsystems. Tables I and II list the key performance points for the HP 9000 E-class, K-class, and D-class servers.

Table I

Key Performance Points for HP 9000 K- and E-Class Servers

Model	Processors	Clock Speed (MHz)	Multiprocessor Configuration	Memory (M Bytes)	Cache (I/D K Bytes)	HP-PB I/O Slots	HP-HSC I/O Slots	Disk Capacity (G Bytes)
K2x0	PA 7200	120	1-to-4-Way	64 to 2048	64 to 256	4	1	3800
	PA 8000	160	1-to-4-Way	128 to 2048	256	4	1	3800
K4x0	PA 7200	120	1-to-4-Way	128 to 3840	256 to 1024	8	8	8300
	PA 8000	160 to 180	1-to-4-Way	128 to 4096	1000	5	5	8300
Ex5	PA 7100LC	48 to 96	1-Way	16 to 512	64 to 1024	4	4	144

Table II

Key Performance Points for HP 9000 D-Class Servers*

Model	Processors	Clock Speed (MHz)	Multiprocessor Configuration	Memory (M Bytes)	Cache (I/D K Bytes)	HP-HSC I/O Slots	EISA I/O Slots
D2x0	PA 7100LC	75 to 100	1-Way	32 to 512	256	4	4
	PA 7300LC	132 to 160	1-Way	32 to 1024	64 **	4	4
	PA 7200	100 to 120	1-to-2-Way	32 to 1536	256 to 1024	4	4
	PA 8000	160	1-to-2-Way	64 to 1536	512	4	4
D3x0	PA 7100LC	100	1-Way	32 to 512	256	4	7
	PA 7300LC	132 to 160	1-Way	32 to 1024	64 to 256 **	4	7
	PA 7200	100 to 120	1-to-2-Way	32 to 1536	256 to 1024	5	7
	PA 8000	160	1-to-2-Way	64 to 1536	512	5	7

* HP-PB slots = 0 and disk capacity = 5 Tbytes.

** 1M bytes of second-level cache.

Additional constraints on the design were a direct result of competitive pressures. As the presence of Industry Standard Architecture-based systems has grown in the entry-level server space, the features they offer became D-class requirements. These requirements include support for EISA (Extended Industry Standard Architecture) I/O cards and an increase in the standard warranty period to one year. Both of these requirements were new to the Series 800. Also new to the Series 800 was the desire to design a system enabled for distribution through the same type of independent distribution channels used by other server vendors. Add to these constraints the cost sensitivity of products in this price range, and we have a system that uses as many industry-standard components as possible, is extremely reliable, and is capable of being assembled by distributors, all without compromising any performance benefits of current or future PA-RISC processors.

Feature List. The first step in the process of partitioning the system was to detail all possible features that might be desired in an entry-level server. This list was compiled by pulling features from our development partners' requirements analysis and from knowledge of our competitors' systems. Once this feature list was developed, each feature was evaluated against all of our design goals (see Fig. 2). Each feature was then ranked in terms of its relative need (must, high want, want) and technical difficulty (high, medium, low). Determining the possible feature list was the first goal of the partitioning process; the list was continually updated during the entire process.

Feature	Need	Difficulty
Front panel display	Must	Low
HP 100LX Palmtop front-panel display	Want	High
Two-line LCD display	High Want	Medium
16-character display	Must	Low
Backlit display	Want	Medium
Hexadecimal status information	Must	Low
English status information	High Want	Medium
Localized language status information	Want	High
Display suspected failure causes after faults	Must	Medium
Include power-on LED indicator	Must	Low
Include disk-access LED indicator	Want	Medium
Include blinking "Boot in progress" indicator	Want	Medium
Reset switch	High Want	Medium
Reset switch with key and lock	Want	High

Fig. 2. *The feature list for the D-class server's front-panel display.*

Once the initial feature list was created, a small design team consisting of a mechanical engineer, an electrical engineer, a firmware engineer, a system architect, and a system manager began analyzing the list to see how each feature would affect the physical partitioning of the system. The goal of this process was to generate a fully partitioned mock-up of the physical system. Successive passes through the feature list led to successive generations of possible designs. With each generation, the list was reevaluated to determine which features could be achieved and which features could not.

Physical Partition. After the first few generations it became clear that a few critical features would drive the overall physical partitioning. The physical dimensions would be determined by the dimensions of a few key subsystems: the disk array and removable media, the integrated I/O connectors, the I/O card cage, and the power supply (see Fig. 3). All of these components were highly leveraged from existing designs, like the hot-swap disk array module developed by HP's Disk Memory Division, and the industry-standard form-factor power supply. The first major design conflict arose when we realized that these components could not be integrated in a package short enough to fit under a standard office desk, and yet narrow enough to allow two units to be racked next to each other in a standard rack. Numerous attempts to resolve these two conflicting demands only succeeded in creating a system that would violate our cost goals or require more new invention than our schedule would allow.

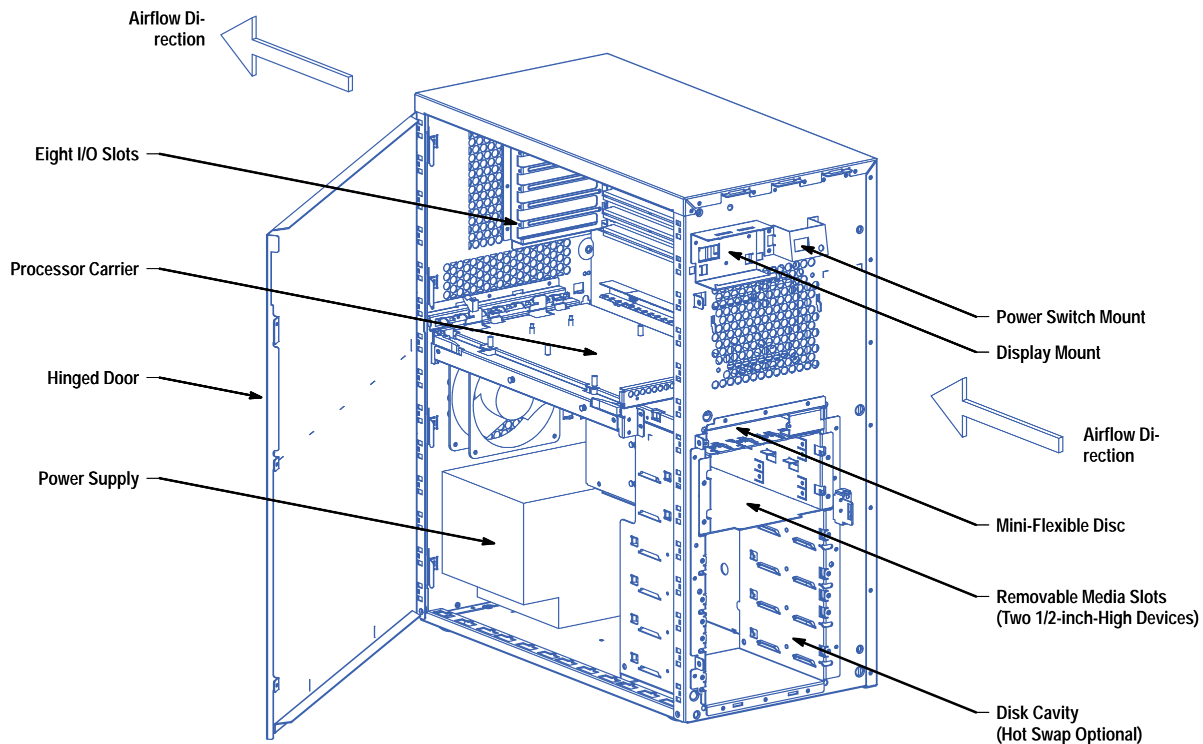


Fig. 3. *The chassis for the Series 800 D-class server. The features that determined the overall size of this chassis were the disk cavity, the removable media slots, the power supply, I/O slots, and processors.*

In the end, it was determined that the desk-height requirement and cost goals were more important than the rack requirement, so the package was shortened and widened to accommodate all the critical components in a package that would fit under a standard office desk. Once this was decided, the system mock-up came together quickly, and the second goal was reached. The system partitioning shown in Fig. 3 provides several benefits necessary to achieve our goals. The standard PC-type power supply helped us to achieve new lows in cost per watt of power. The division of the box into the lower core I/O and disk array volume and the upper expansion I/O slot and processor area helped to simplify the design of the forced-air cooling system since it separates the large interior volume of the box into two more manageable regions.

Printed Circuit Assemblies. The next goal in the process was to repartition and integrate the disparate design sources* into logically and physically compatible printed circuit assemblies, while maintaining all of our design constraints on cost, expandability, and design for distribution. Again, a single crucial design decision helped to quickly partition the system: the D-class would not use any high-speed, impedance-controlled connectors. This decision was made as a direct result of the K-class development process and the success of the Series 800 G/H/I-class Model 60 and 70 systems. The K-class development process showed that although high-speed impedance-controlled connectors can add excellent flexibility and expandability to midrange systems, they require a great deal of mechanical and manufacturing infrastructure.

The processor modules for the G/H/I-class Models 60 and 70 are the uniprocessor and dual-processor versions of the same board (see Table III). The same printed circuit board is loaded with either one or two processors at the time of manufacture. To increase the number of processors in a system, the entire processor module must be replaced with a new printed circuit assembly. Other systems, like the K-class servers, allow for the incremental increase of the number of processors in the system with just the addition of new processor modules. Even though the board swap is a less desirable upgrade path than an incremental upgrade, the success of the Models 60 and 70 systems led us to believe that it was quite acceptable to our customers.

Table III

Key Performance Points for the HP 9000 Series 800 G/H/I-class Model 60 and 70 Systems*

Model	Multiprocessor Configuration	Memory (M Bytes)	HP-PB I/O Slots	Disk Capacity (G Bytes)
G60	1-Way	32 to 768	4	156
G70	2-Way	32 to 768	4	156
H60	1-Way	32 to 768	8	300
H70	2-Way	32 to 768	8	300
I60	1-Way	64 to 768	12	330
I70	2-Way	64 to 768	12	330

* Processor = PA 7000, clock speed = 96 MHz, Cache per CPU (Instruction/Data) = 1M byte/1M byte, and HP-HSC slots = 0.

This decision simplified repartitioning the design sources, since it meant that the high-speed processor and memory clock domains and their data paths could remain on a single printed circuit assembly, while the moderate-speed I/O domain and its data paths could cross multiple printed circuit assemblies. It was determined that both the dual I/O bus architecture of the K-class and the single I/O bus of the E-class would be supported in the system. To do this, the connector technology used in the D-class is modular, allowing the designs to load only those portions of the connector that are supported. This lowers both the material and assembly costs. To further lower the material cost, the PA 7100LC-based processor and memory module is fabricated on a smaller printed circuit board than the PA 7200-based processor module. The difference in the size of the modules is accommodated by attaching them to a sheet-metal carrier that adapts the modules to a common set of card guides. Not only is the sheet metal cheaper than the corresponding printed circuit material would have been, but it is also stronger and easier to insert and remove.

A secondary benefit of this strategy is that it allows new investment to be made as needed. Historically, I/O subsystems and technology are much longer-lived than processor and memory technologies. The partitioning strategy we used helped to decouple the I/O subsystem from the processor and memory. As long as they remain consistent with the defined interface, processor modules are free to exploit any technology or adapt any design desirable. This also enables D-class servers to excel in meeting a new and growing requirement—design for reuse. A customer is able to upgrade through many performance points simply by changing processor modules. As some countries are investigating forcing manufacturers to accept and recycle old equipment, keeping the return stream as small as possible is highly desirable.

* The design sources were parts from E-class and K-class servers and the J-class workstations that were combined to form the D-class servers.

Once the printed circuit assembly board outlines were complete, the process of adapting the various design sources to the new partitioning was time-consuming, but relatively straightforward. Fig. 4 shows the the various design sources that were pulled together to form the PA 7200-based processor module. As portions of designs were merged, altered, and recombined, the possibility of transcription errors grew. The original designs were executed by three different labs and many different design teams. All designs were fully functional as designed, but we were extending designs as well as integrating them. In an effort to minimize the possibility of errors being introduced during the adaptation process, the schematic interconnect list was extracted and translated into a simulation model. This model was then added to the models used to verify the original designs to ensure that no new errors had been introduced.

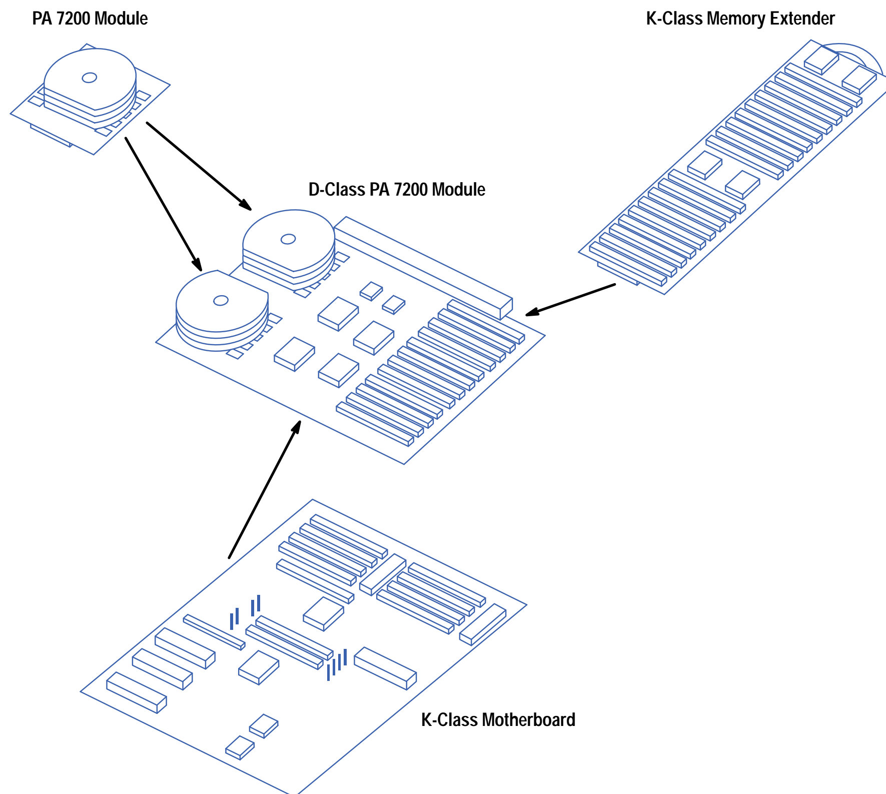


Fig. 4. *The various design sources that were pulled together to form the PA 7200 D-class module.*

System Partitioning and Firmware Design

Because of the partitioning scheme used for the D-class entry-level servers, the firmware design was a critical factor in achieving the overall program objective of low cost. The firmware design addressed cost issues in support of the manufacturing process, field support, and the upgrade strategy. In addition, although the underlying hardware is dramatically different depending upon which processor module is installed in the system, from the customer's perspective, the external behavior of each performance point should be the same. For the firmware design team, that meant that regardless of the underlying hardware, the entire D-class had to have the look and feel of a single product.

D-Class Subsystems. From a firmware perspective, the D-class is partitioned into two subsystems: the system board and the processor modules (see Fig. 5). The system board contains all of the I/O residing on or hanging off the HSC (high-speed system connect) bus. This includes optional I/O modules that plug into the HP-HSC slots, such as the fast-wide SCSI and graphics cards. It includes core I/O built into the system board, which provides serial interfaces, single-ended SCSI, Ethernet LAN, a parallel interface, a mouse and keyboard interface, and a flexible-disk interface. The EISA bus, which is connected at one end to the HSC bus, is also found on the system board. The Access Port/MUX card,* which contains its own HSC-to-HP-PB I/O bus converter, also plugs into an HSC slot. In addition to these I/O buses and devices, there is an EEPROM and two hardware dependent registers that hold I/O configuration information. This nonvolatile memory and the configuration registers are critical to the partitioning and upgrade strategy for the D-class server.

The core of each processor module houses the CPU, instruction and data caches, and memory subsystems. Also on the processor board is an EEPROM and two more hardware dependent registers. The PA 7100LC uses the HSC as its native bus, so its connection to the system board is relatively straightforward. However, the PA 7200 requires a bus converter between

* HP Access Port is a tool for providing remote support for HP servers.

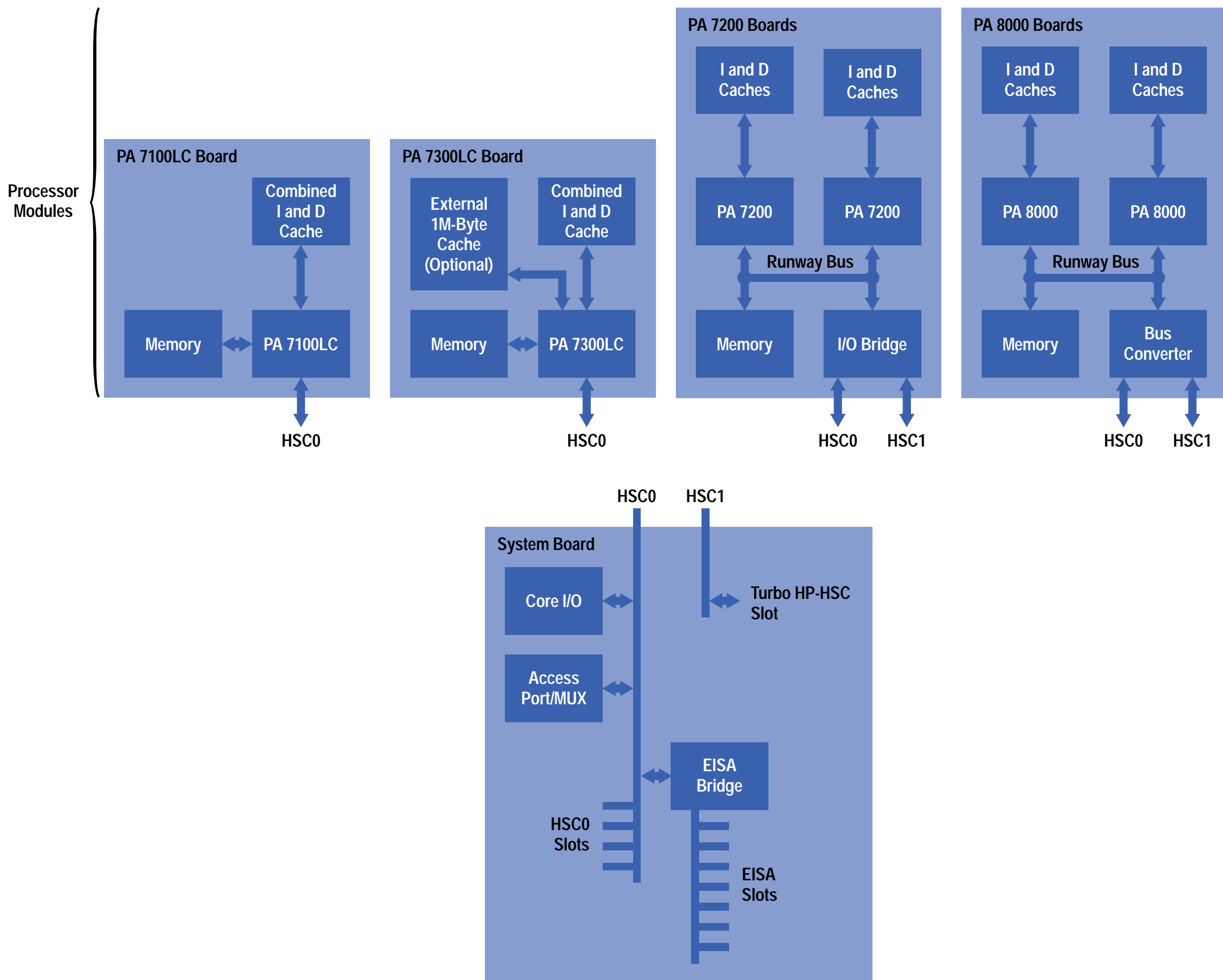


Fig. 5. *A firmware perspective of the D-class subsystems.*

its native bus and the HP-HSC bus. Thus, to have one system board common between the two processor modules, the PA 7200 processor board was burdened with carrying the bus converter circuitry. One other significant difference exists between the two processor modules: scratch RAM.** The inclusion of 32K bytes of static RAM on the PA 7200 module meant that system variables and a stack could be set up very early in the boot process. The lack of this scratch RAM on the PA 7100LC limited the amount of code that could be common between the two platforms.

Consistent Look and Feel. The goal of having the same look and feel for the entire product line could be met by having one common code base for all performance points, but because of the significant hardware differences mentioned above, this was not possible. The primary differences lie in the processors themselves. There is no commonality in the control and status registers of the two processors, caches are accessed differently, and the memory subsystems are too different to share code. These differences, along with other hardware incompatibilities, meant that each processor module needed to have its own separate and distinct code base. However, because the primary differences between the two PA 7100LC processor versions (75 MHz or 100 MHz) and the two PA 7200 processor modules are processor speed and cache size, all performance points that use a common CPU can be supported by one common code base.

The three areas needed to give the product line a consistent look and feel included a common feature set, similar strategies for handling and reporting errors, and a common user interface. To ensure consistency in this regard, one engineer was given responsibility for the same functional area of each platform. For example, the engineer who worked on memory code for the PA 7100LC also had responsibility for the memory code on the PA 7200 platform. Taking advantage of this synergy paid off especially well in the design and implementation of the user interface, where differences between platforms could easily lead to confusion.

System Configuration. Partitioning the EEPROMs between the processor board and the system board is a key enabler of the upgrade strategy. Since an upgrade consists of replacing the processor module, I/O configuration information must remain with the system board. The EISA configuration, graphics monitor types, and LAN MAC address are stored on the system

** The PA 7300LC and the PA 8000 also include scratch RAM.

board. Additional control information is used to check for consistency between the two EEPROMs. The firmware expects the format of the system board EEPROM to be the same, regardless of which processor module is installed. With all I/O configuration information and control variables in the same location and sharing the same set of values, processor modules can be freely swapped without changing the I/O configuration.

Dynamic configuration of the system is used to support the upgrade strategy, the manufacturing process, and field support. When a D-class server is powered on, state variables in the processor module's and system board's nonvolatile memories are tested for a value that indicates whether or not they have been initialized and configured. If they fail this test (which is always the case for initial turn-on during the manufacturing process) the system's hardware configuration is analyzed and the corresponding state and control variables are set. Much of this information is available via the hardware dependent registers located on each board. The processor frequency, system board type, and other details concerning the I/O configuration can be read from these registers.

The state variables, which are set as a result of examining the hardware configuration, include the system's model identifier (e.g., HP9000/S811/D310), hardware version (HVERSION), and paths to the boot and console devices. The boot path can be either the built-in single-ended SCSI device or a hot-swappable fast-wide SCSI device. The firmware checks for the presence of a hot-swappable device which, if present, becomes the default boot path. Otherwise a single-ended SCSI device is configured as the default boot path. The actual hardware configuration is also examined to select an appropriate console path. The default console path can be either the built-in serial port, the HSC Bus Access Port/MUX card, or a graphics console. Depending upon the presence of these devices and their configurations (e.g., a graphics device must also have a keyboard attached), a console path is selected according to rules worked out in cooperation with the manufacturing and support organizations.

The same sequence of events occurs when upgrading or replacing a processor module. In this case, the system board is already initialized and only the processor module requires configuration. On every boot, information such as the model identifier is checked against the actual hardware configuration and any mismatch will invoke the appropriate configuration actions. Likewise, because some information is kept redundantly between the processor module and the system board, they can be checked for a mismatch. This redundancy means that the system board can also be replaced in the field with a minimal amount of manual reconfiguration. Because a D-class server can consist of any combination of two system boards and several different processor modules, and because further enhancements will double the number of processor modules and include two new CPUs, dynamic configuration has obviated the expense of developing external configuration tools, reduced the complexity of the manufacturing process, and simplified field repairs and upgrades.

System Packaging

Mechanical packaging is one of the key variables in maintaining a competitive edge in the server market. The challenges involved in the system package design for the HP 9000 D-class server included industrial design, manufacturability, EMI containment, thermal cooling, and acoustics, while having the design focus on low cost.

The D-class low-cost model was based on the high-volume personal computer market. However, unlike personal computers, server products must support multiple configurations with an easy upgrade path and high availability. This meant that the D-class package design had to be a highly versatile, vertical tower with the ability to be rack-mounted in a standard EIA rack (see Fig. 6). It allows for multiple processors and power supplies, and can support up to eight I/O slots for EISA and GSC cards. The design also supports up to five hot-swappable or two single-ended disk drives and two single-ended removable media devices with one IDE (Integrated Device Electronics) mini-flexible disk (Fig. 7). This diversification provides the entry-level customer with a wide range of configurations at various price/performance points.

Manufacturing and Field Engineering Support. Concurrent engineering was a key contributor to the design for assembly (DFA) and design for manufacturing (DFM) successes of the D-class server. Since we are very customer focused, we take the disassembly and repair of the unit just as seriously as the manufacturability of the product. The D-class mechanical team worked closely with key partners throughout the program to ensure the following assembly and manufacturing features:

- A single-build orientation (common assemblies)
- Multiple snap-in features
- Slotted T-15 Torx fasteners (Torx fasteners are used for HP manufacturing, and the slot is for customers and field engineers.)
- System board that slides into the chassis
- Quick access to all components
- Manufacturing line for high-volume production.

EMI. Design for EMI containment was a considerable challenge for the D-class server program. The package goal was to contain clock rates up to 200 MHz. This required a robust system design and two new designs for the EISA bulkheads and core I/O gaskets.

The system EMI design is based on a riveted sheet-metal chassis using a slot-and-tab methodology for optimum manufacturability. A cosmetic outer cover with a hinged door completes the EMI structure. EMI is contained using continuous seams and EMI gaskets with small hole patterns for airflow.

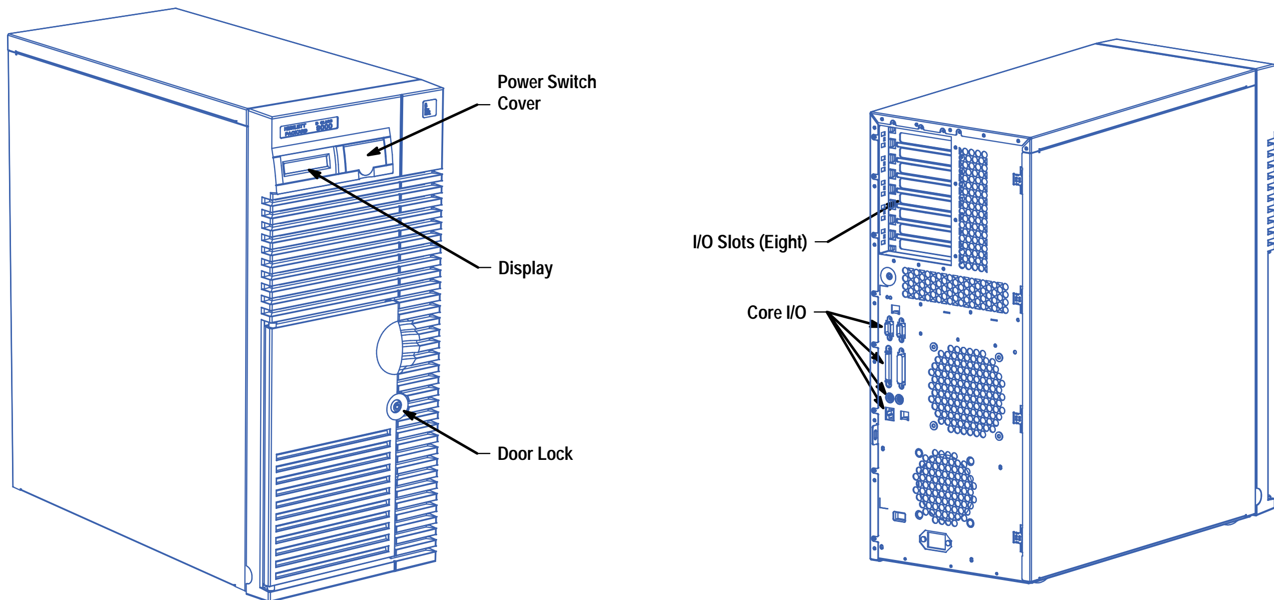


Fig. 6. *A front and back view of the D-class server.*

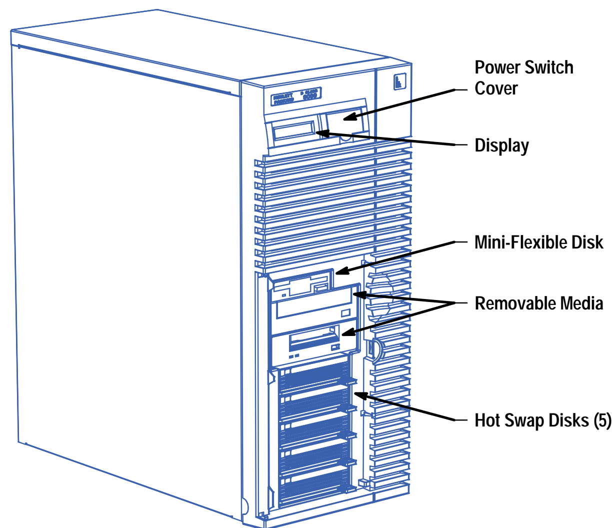


Fig. 7. *The D-class cabinet with the door removed.*

The EISA bulkhead gaskets required a new EMI design. The new design is a slotted pyramid that forms a lateral spring element with a low deflection force but a high contact force (see Fig. 8). The new design for the I/O gasket includes a foam core wrapped with a nickel-over-copper fabric, which provides a 360-degree contact around each connector (see Fig. 9). These new designs produced excellent results.

Thermal Design and Airflow. The thermal design for the D-class server also had some interesting challenges. The design strategy had to encompass multiple configurations and multiple processor chips and boards. Some of these options were in development, but most were future plans. A thermal analysis program, Flotherm, was used to develop the thermal solution for the system.

The Flotherm models and tests resulted in the package being separated into two main compartments. The top half, which includes the I/O and the processors (see Fig. 3), is cooled by a 12-mm tubeaxial fan. The processor chips are located side-by-side directly behind the front fan, giving an approximate air velocity to the processors of about 2.5 m/s. Heat sinks are used for processor chips that consume under 25 watts. For chips over 25 watts, a fan mounted in a spiral heat sink is used.

The bottom half of the package includes the peripheral bay and power supply and components. It is cooled using a 120-mm tubeaxial fan. However, when the hot-swap disks are in use, a separate cooling system is installed. The hot-swap bay is a

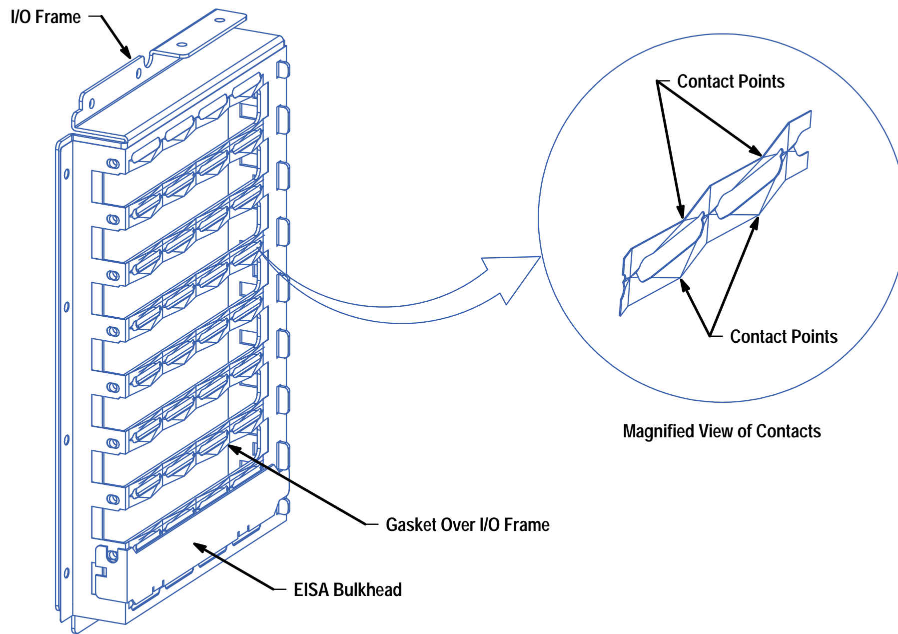


Fig. 8. *The EISA bulkhead gaskets for the D-class server.*

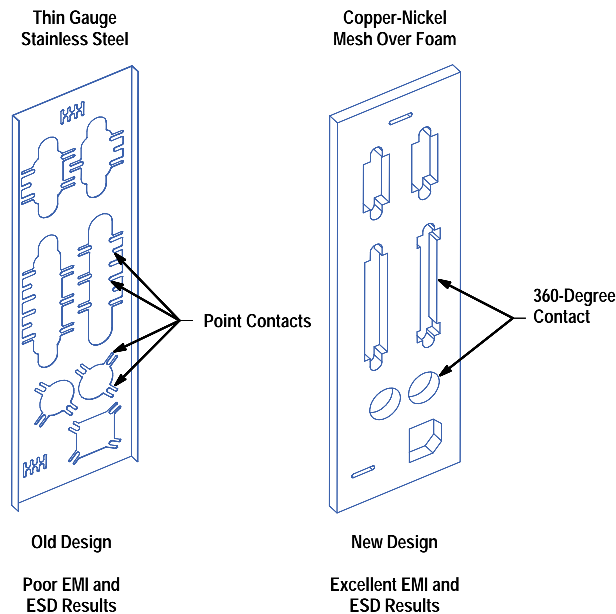


Fig. 9. *A comparison between the new and old core I/O gasket designs. The new design provides a 360-degree contact around each connector.*

sealed subsystem that uses a small blower to pull air through the disks. Any disk can be pulled out and the airflow to the other disks remains relatively unchanged. The power supply has its own 92-mm fan.

Acoustics. The acoustical goal for the D-class server was designed to be 5.4 bels at the low end, which was the same as for earlier server products. This package has a higher power density than previous products, more versatility, higher-speed discs, and an off-the-shelf power supply rather than a custom one. The fan in the power supply ended up being the loudest component of the system. Still, the system came in at 5.4 bels at the low end by custom tuning sheet-metal parts, baffles, and fan speeds.

High Availability and Ease of Use

Hot-Plug Internal Disks. An important feature that the D-class server has brought to the Series 800 product line is hot-pluggable internal disk drives. While commodity servers had once provided forms of internal hot-pluggable disks, these solutions were deemed too likely to cause data corruption for use in Series 800 systems. For example, the commodity

hot-plug solutions evaluated for use in Series 800 systems had the issue of "windows of vulnerability" in which sliding contacts from a swapping disk on a SCSI bus could cause data bits to actually change after parity had already been driven, causing undetected data errors. With a low probability of corruption, this approach may have made sense for a workgroup file or print server. However, Series 800 servers are expected to operate in mission-critical environments. Therefore, a more robust approach to hot-swapping disk drives had to be developed for these products.

The approach used for the D-class server was to provide a hot-swap solution using logical volume management (LVM)* and mirroring middleware facilities, and to offer a disk-drive carrier common to the standalone enclosure disk carrier being separately developed for the rest of the Series 800 product line. The common carrier approach would allow the field to learn only one solution and guarantee a higher volume of parts. In addition, solutions for the data corruption problem, the use of sequenced SCSI resets, and an automated swap script could be shared by both the enclosure team and the D-class server team.

Mission-Critical ServiceGuard and EISA I/O. HP's MC/ServiceGuard product (portions of which were previously called Switchover/UX) has been an unofficial standard in the application-server industry for several years now. This middleware product allows Series 800 systems to operate as standby servers for one another, transferring mission-critical workloads from a failed system to its standby. This feature requires that a system and its standby host transactions on the same mass storage buses, enabling the standby system to have access to all of the primary system's data. Multiple hosting (known as multi-initiator) on mass-storage interconnects requires significant design attention. In addition, the LANs that allow these systems to communicate with one another must offer special capabilities in the areas of error handling and reporting.

A significant challenge for the D-class design was merging the workstation, PC, and I/O infrastructures with the Series 800 infrastructure. This requires supporting MC/ServiceGuard capabilities, higher slot counts, more extensive errorhandling, and a remote support infrastructure. Before the D-class server, all Series 800 systems that supported MC/ServiceGuard were HP Precision Bus (HP-PB) based.

The D-class server's use of EISA and HP-HSC (with the EISA form factor) required the team to implement, debug and verify MC/ServiceGuard functionality on these new I/O buses. In the process, various I/O implementations had to be modified, such as special EISA bus operating modes, SCSI adapter functionality, periodic EISA cleanups, guaranteed arbitration for core LAN, queueing transactions on HP-HSC, extra buffering of data signals, and slot configurations. These modifications made the HP-HSC and EISA I/O infrastructure more robust in a highly available departmental branch server.

In addition, HP's Access Port product, which is used for remote support of HP servers, only existed on the HP-PB. Without HP-PB slots, the D-class server would have had to either forego remote support or develop a new card. The answer was not to develop a new card, but to leverage existing logic by supplying a buried HP-PB on the new Access Port card. To the user, the D-class server's Access Port is an HP-HSC card. However, to the operating system and response center engineer, the D-class server's Access Port looks like the familiar, compatible HP-PB card found on all the other server products.

Pushbutton Graceful Power Shutdown

The D-class server is the first Series 800 system to offer pushbutton graceful power shutdown. Basically, when a D-class system is up and running, the power button is equivalent to the command `reboot -h`, which causes the system to synchronize its buffer cache and gracefully shut down. This feature is most useful in a branch office or department where the server is minimally managed by local personnel. Single-user HP workstations had introduced this feature to PA-RISC systems.

Built-in Remote Management

With an emphasis on remote server management, the D-class server team decided to offer the same robust, worldwide-usable internal modem as the K-series products. This modem offers support for transfer rates well beyond those used by today's HP response centers and is integrated with the remote support assembly in D-class systems. The product also offers a special serial port for controlling optional uninterruptible power supplies (UPS), as well as pinout definitions for future direct control of internal power-supply signals.

The D-class server team also accommodated "consoleless" systems, whereby a D-class server can be completely managed remotely without a local console at all. In addition, graphics console customers can still take advantage of remote console mirroring (formerly reserved strictly for RS-232 consoles) by merely flipping a switch on the product.

* LVM is software that allows the number of file systems to be relatively independent of the number of physical devices. One file system can be spread over many devices, or one device can have many file systems.

Conclusion

The system partitioning design for the first release of the D-class servers helped to achieve all of our introduction goals. We were able to introduce both PA 7100LC-based and PA 7200-based processor modules, integrate the industry standard EISA I/O bus into the Series 800 hardware for the first time, and achieve our cost and schedule goals without any investment in new VLSI ASICs.

In the end, the D-class design had leveraged from all current entry-level and midrange Series 800 servers and many Series 700 workstations. Because of the care taken during the adaptation process, performance enhancements made to the original design sources were made available in the latest D-class module quickly and with very little investment. As an example, only two weeks after a larger-cache, higher-speed PA 7200 K-series processor module was released, the corresponding D-class PA 7200 processor module had been modified and released for prototype. This module provides four times the cache and a 20% increase in frequency over the initial D-class PA 7200 module.

Table IV summarizes the leverage sources for the various subsystems that make up the D-class servers.

Table IV
Leverage Sources for the D-Class Subsystems

D-Class Subsystem	Leverage Source
EISA I/O	HP 9000 Series 700 Workstations
HP-HSC I/O	K-class server and J-class and HP 9000 Series 700 Workstations
Clocking	E-class and K-class Servers
PA 7100LC Processor Modules	E-class Server
PA 7200 and PA 8000 Processor Modules	K-class Server
PA 7300LC Processor Modules	C-class Server
Power Supply	Industry-Standard Suppliers
Hot Swap Disk Array	HP's Disk Memory Division

A Low-Cost Workstation with Enhanced Performance and I/O Capabilities

Various entities involved in product development came together at different times to solve a design problem, evaluate costs, and make adjustments to their own projects to accommodate the cost and performance goals of the low-cost HP 9000 B-class workstation.

by Scott P. Allan, Bruce P. Bergmann, Ronald P. Dean, Dianne Jiang, and Dennis L. Floyd

The design and development of the HP 9000 B-class workstation is a good example of cooperative engineering. In cooperative engineering, the various entities involved in product development come together at different times to solve a problem or make adjustments to their own projects to accommodate a common need. Examples of this cooperation for the B-class workstation include coordination between system designers and firmware developers, the addition of new functionality without impacting the development schedule, close ties with manufacturing, evaluation of implementation based on detailed cost models, and simplification of the PA 7300LC design by moving clocking functions onto a small chip on the system board.

Design Objectives

The design objectives for the B-class workstation were low cost, quick time to market, performance, functionality, longevity, and modularity. In addition to these objectives, the development team's main goal was to produce a workstation based on the PA 7300LC processor that would be comparably priced to the HP 9000 Model 715 workstation, but with superior performance and I/O capabilities. This goal and the design objectives remained the same throughout the project.

With low cost as the primary objective, any feature that was perceived as too costly or of limited value to our customer base was not included. Leveraged subsystems were reviewed in search of creative ways to reduce cost. This led to reductions in the cost of the clock circuitry and firmware interface and elimination of some legacy I/O interfaces. From a cost/performance perspective we were able to justify the addition of a PCI (Peripheral Component Interconnect) bus, a higher-speed memory technology, a second-level cache, and a higher-performance processor and graphics subsystem. Fig. 1 shows the B-class workstation.

Fig. 1. *The B-class workstation.*

Features and Capabilities

Based on the objectives for the B-class workstation, the following features are included in the product:

- PA 7300LC high-performance, low-cost microprocessor with two on-chip associative caches with 64K bytes for data and 64K bytes for instructions
- 1M bytes of ECC (error-correcting code) directly mapped second-level cache for additional performance
- HP VISUALIZE graphics technology from HP VISUALIZE-EG (entry-level graphics)
- HP VISUALIZE-IVX graphics on the B132 workstation (optional)
- Six memory slots that support up to 768M bytes of ECC memory, including fast-page mode (FPM) and extended-data-out (EDO) DRAM dual inline memory modules (DIMMs)
- General system connect (GSC) bus for high-speed I/O bandwidth
- Flexible I/O that includes two I/O slots, which can be configured as:
  - Two PCI slots
  - Two GSC slots
  - One EISA slot
- Optional fast-wide SCSI (20-Mbyte/s) card that supports internal and external disks without using an I/O slot.

In addition to these features, the B-class workstation's modular design provides simple installation, flexibility of use, and easy servicing. This is accomplished through design features such as:

- Simple tray design
- Built-in expandability
- Plug-in memory modules.

Fig. 2 shows a block diagram of the components that make up the B-class workstation.

Processor and System Design

Since the processor chip used in the B-class products is the PA 7300LC, one of the main areas of cooperation was between the PA 7300LC processor design team and the B-class system design team.

The previous-generation processor used in HP workstations of a comparable price was the PA 7100LC. The PA 7100LC was an extremely versatile processor, and many of its best points were leveraged into the PA 7300LC design (see ***Article 6***, ***Article 7***, ***Article 8***, and ***Article 9***). However, the PA 7100LC was not without its challenges, such as the difficulty in synchronizing the processor clock with the GSC (general system connect) bus.

Clock Frequency

The GSC bus is a general-purpose synchronous bus used to communicate between the processor and I/O. Its phase is determined in relation to a nonexistent GSC clock. This imaginary clock runs at half the frequency of the clock sync signals driven to each GSC device. Its rising edge is defined by the rising edge of reset during initialization, and each GSC device is responsible for keeping track of the current phase of the GSC clock starting from initialization.

On the PA 7100LC, the GSC bus was only permitted to operate at fixed ratios of the processor clock frequency, including some odd clock ratios such as 1.5:1 (see Fig. 3). All of the clock syncs and the resets used to initialize the GSC clock were external to the chip. Designing circuitry to maintain these ratios and timing margins with minimal clock skew and noise immunity became increasingly problematic. In addition, every frequency point of operation required a special clock design to ensure maximum performance. This limited our ability to select the frequency of operation based upon yield at a later point in the design process. For the PA 7300LC, the situation became more critical because the final processor frequency was still uncertain, and the final ratio between the processor frequency and the GSC clock was also undecided.

The first approach investigated was to bring the entire clocking solution into the PA 7300LC. It would be much easier to adjust the delays and control the skew within an ASIC rather than in discrete circuits. The proposal was to incorporate a phase-locked loop circuit within the PA 7300LC to generate the processor clocks from a low-frequency external crystal.

The GSC syncs could then be created by dividing the phase-locked loop output internally in the PA 7300LC. The PA 7300LC would also drive out the reset used to initialize the GSC phase. Upon further investigation, the PA 7300LC design team became concerned about the risk associated with the phase-locked loop. The phase-locked loop was considered a major component of the PA 7300LC design. This was significant because all post-fabrication verification and debugging of the chip would be dependent upon a functional phase-locked loop.

At this point, the B-class system designers and the PA 7300LC design team began to look at a mixed solution. The phase-locked loop was scrapped to avoid risk, and its die area recovered for other uses. The PA 7300LC would continue to drive the primary synchronizing reset to eliminate the need to synchronize the asynchronous power-on reset to the GSC's syncs. The generation of the syncs and the maintenance of their skew requirements would be moved to an external ASIC.

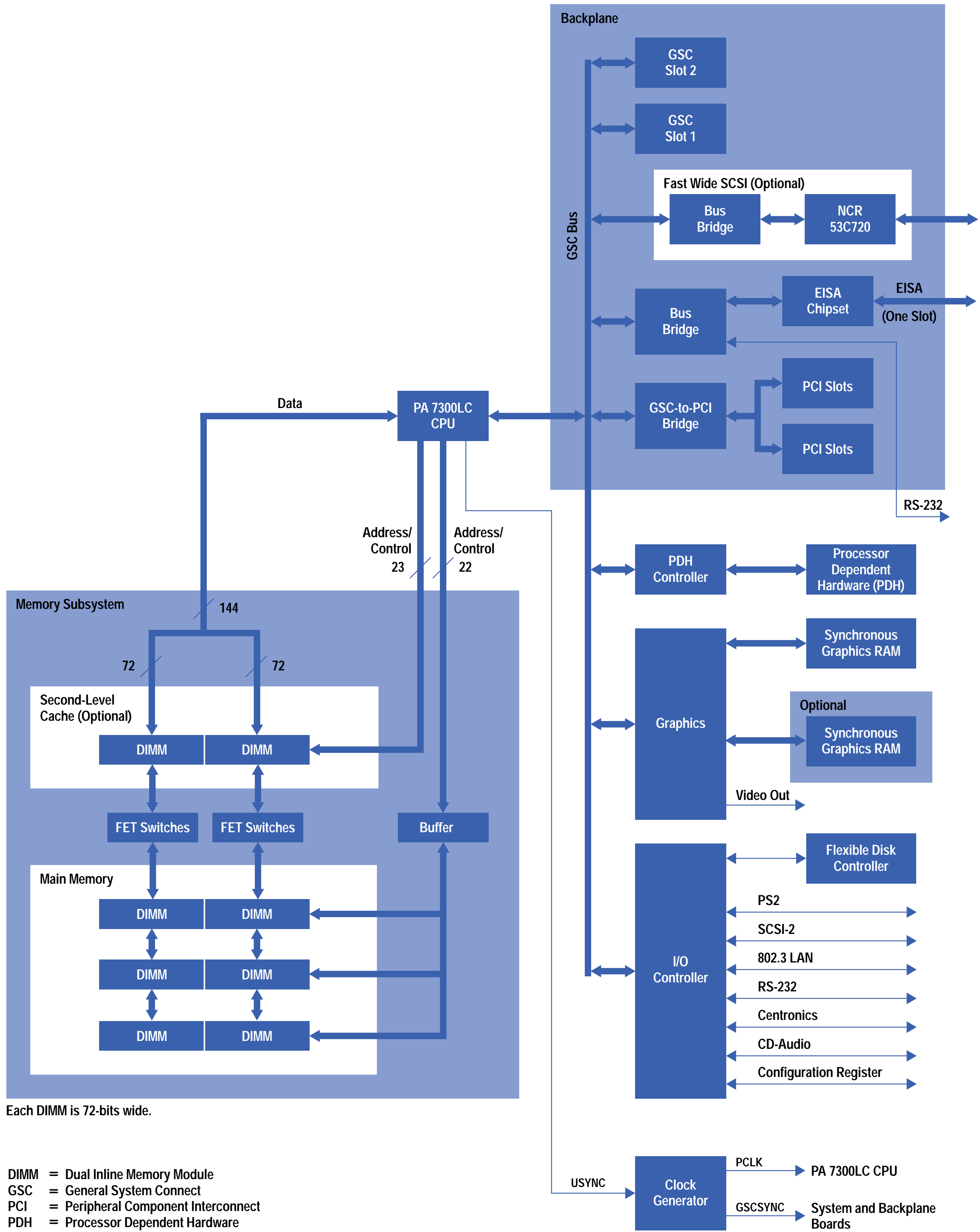


Fig. 2. *The system block diagram for the B-class workstation.*

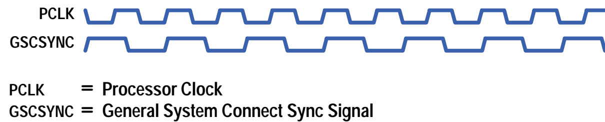


Fig. 3. *The PA 7100LC clocking scheme.*

Any necessary turns to a small ASIC would be quicker and less expensive. In addition, the clocking solution could be completely bypassed to allow continued verification and debugging of the PA 7300LC if necessary.

Working with Motorola, the PA 7300LC design team, and our materials organization, the system designers specified the device that became the Motorola MPC992 (the clock generator in Fig. 2). This device uses a phase-locked loop and an external low-frequency crystal to generate differential clocks that provide clocking to the processor and the other GSC devices. As an added benefit, its cost is relatively low in relation to the external clock oscillator and ECL devices used in previous products. The USYNC signal, which comes from the PA 7300LC processor, is the synchronizing signal that is responsible for aligning the GSCSYNC with the processor clock signal.

Memory and I/O Controller

The proximity and working relationship between the PA 7300LC and B-class system design teams allowed us to communicate design specifications with relative ease. This working environment allowed us to view the product as a whole rather than designing the system around an existing chip.

The design of the memory and I/O controller (MIOC) was the first area affected by this arrangement. The PA 7300LC is designed to support optional second-level caches of different technologies and sizes. When the PA 7300LC chip design team began investigating each of these second-level cache options, the B-class system designers were able to check the appropriateness of their solution with the design. One of the first decisions under this arrangement was to make the second-level cache optional and locate it on a DIMM (dual inline memory module) on a separate board. This provides the B-class workstation with several benefits:

- Lower-performance systems are not burdened with the cost of the second-level cache.
- Systems with and without a second-level cache can share system boards, reducing development and verification time.
- The exact configuration of the second-level cache can be altered at a later date if market conditions warranted.
- Less space is required on the board, permitting a lower-cost system board.

It was important for the PA 7300LC design team to know that a DIMM solution was being considered since it would have a big impact on the I/O pad design of the PA 7300LC.

Another area of concern within the MIOC involved the impact of the expanded data bus on the PA 7300LC (144 bits) compared to the PA 7100LC (72 bits). This would require additional pins and incur additional packaging costs. The PA 7300LC design team wanted to share the memory data bus with the second-level cache data bus to reduce the number of external I/O pins. However, the additional load associated with the memory would degrade the response of the second-level cache. The PA 7300LC design team suggested FET switches, which could be dynamically opened and closed to isolate the second-level cache from main memory.

The B-class system designers were able to verify using FET switches in a system environment. However, the only devices available that met the enable/disable speed requirements were 8-bit devices. This was viewed as an unwieldy and expensive solution in the B-class system. Working with our materials organization and Texas Instruments, the B-class system designers were able to make minor specification changes to an existing 24-bit Texas Instruments part to improve this speed parameter and cut the quantity and cost of the FET switches significantly. The B-class system designers verified the signal quality of the memory data and second-level cache data of these devices in a system environment.

As the configuration of the second-level cache solidified, the B-class system designers were able to provide the PA 7300LC design team with specific information concerning the electrical environment in which the PA 7300LC would be operating. With this information they were able to run simulations of their I/O pad drivers operating within the actual system. This led to some changes in their pad designs, eliminating potential problems later.

Memory

As with most projects, the PA 7300LC design team and the B-class system designers had their share of resource shortages. One such issue involved the memory family. The PA 7300LC is designed to support both fast-page mode and extended-data-out DRAMs. In fast-page mode, sequential data is driven from the DRAMs on successive column addresses following a single row address, rather than requiring both the row and column address to be driven on each data access. Extended-data-out DRAMs are an enhancement to fast-page mode DRAMs in which the data remains valid until the column address changes or a new column address strobe occurs, rather than becoming invalid when the column address strobe disappears. This allows a longer time period over which to latch incoming data and saves processor states in memory accesses.

Unfortunately, resource conflicts and schedule constraints made it impossible for the PA 7300LC design team to verify functionality of the chip for both memory technologies. The PA 7300LC design team wanted to qualify the extended-data-out DRAM technology because it would provide a higher-performance memory technology. The B-class system design team wanted the fast-page mode DRAM technology qualified to be compatible across the workstation family, rather than having a unique memory component for the B-class systems. The compromise solution was to have the PA 7300LC design team qualify the fast-page mode DRAM technology for first release. At a later point in the design phase, the B-class system designers would qualify the operation of extended-data-out mode DRAMs to be introduced as a performance enhancement.

Data Capture

Resource balancing was also evident in the development of a data capture board for the PA 7300LC. A data capture board is a device that is attached to a system board and is used to observe the high-frequency signals between the processor, second-level cache, and memory for debugging purposes. Since the B-class system designers were more familiar with board design tools and the board design environment, the B-class system design team developed the data capture board for debugging the PA 7300LC.

Hardware and Firmware Trade-offs

Design teams frequently look at trade-offs between optimizing resources and meeting the goals of the team. For the B-class workstation, the hardware and firmware teams fostered a close working relationship, allowing trade-offs to be made on a broader scale.

A most unusual but significant outcome of this close working relationship was the development of an unplanned ASIC for interfacing to the processor dependent hardware (PDH). The PDH consists of components such as the boot ROM, nonvolatile memory, and configuration registers. Although there was already a way to connect to the PDH functionality through part of the core I/O logic being leveraged from previous lower-end workstations, this interface did not provide the level of functionality that was implemented in the higher-end workstations. The firmware team could save significant resources by leveraging portions of code from the C-class workstation and the higher-end members of the D-class server family. Many of the basic I/O and graphics functions were similar between these platforms. However, the code leverage was predicated on having certain PDH functionality that could not be provided with the low-end solution. In addition, the high-end solution provided superior debug capabilities. These better debug capabilities were very attractive to help ensure a speedy startup of the new PA 7300LC processor, and hence help meet our time-to-market goals.

The key capability missing from the PDH interface used in previous lower-end workstations was the ability to perform word-wide write accesses to PDH devices. The PDH interface was optimized for reads, with only byte-write capabilities provided. The new PDH ASIC added the word-write capability to support a scratch RAM. This seemingly innocent scratch RAM was key, because in high-end workstation code it is used as a stack in the early stages of the boot process before main memory is initialized. The scratch RAM is also used for global information such as tables of I/O and graphics configuration information. It would have been very difficult to leverage code with the word-write capability to a platform without this capability.

The new PDH ASIC also provided additional address decode and the appropriate flexibility in timing to allow the direct connection of a serial port into the PDH hardware. This direct connection to a serial port, in conjunction with the capabilities offered with the scratch RAM, allowed a debugger to be operational even with hardware that was minimally functional. This serial port aided code and hardware debugging by allowing the hardware status to be monitored and the hardware configuration to be modified early in the boot process.

The risk for the new PDH ASIC was minimized by incorporating it into system simulation efforts and by keeping the design focused on the needed functionality and disallowing any unnecessary features.

Product Definition

The B-Class system was originally defined alongside the C-class workstations. The B-class system is essentially a smaller version of the C-class workstation. Our original intention for the B-class implementation was to use the same modular philosophy of separate I/O, CPU, disk interface, and human interface subsystems used in the C-class machines. However, when the time came to implement the B-class product, cost goals had become more important. When preliminary costs were evaluated, it became clear that we were not meeting the cost objectives with the existing product definition.

Many alternatives were generated and evaluated against product objectives. Finance and R&D reviewed their cost models to see where costs could be saved. Manufacturing reviewed the design alternatives for manufacturability and analyzed the supply chain for issues associated with parts procurement, assembly, material, and structure. Service was consulted to review serviceability and warranty implications of the various options, as well as issues with potential future upgrade products. The result of this analysis was a single-board integrated computer (see Fig. 4). The design, which was initially spread out over four separate boards in the C-class system for the sake of modularity, was now integrated onto one system board.

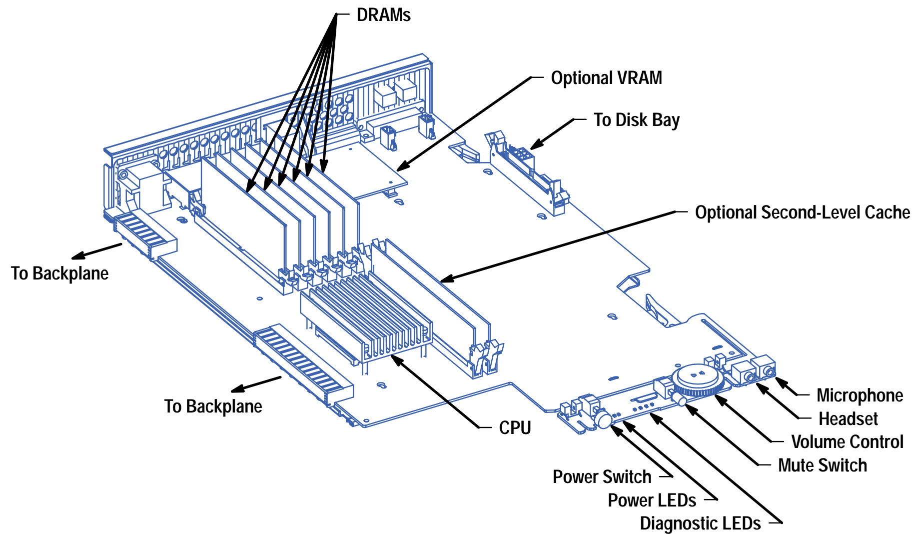


Fig. 4. *The B-class system board and its components.*

Single-Tray Concept

Like the C-class workstation, the B-class workstation uses a tray concept. However, instead of two trays (one for the disks and one for the boards), there is one tray that holds everything (see Fig. 5). For this reason, during the design phase it was important to consider keeping the weight down. Holes were added in the tray wherever possible to reduce the overall product weight.

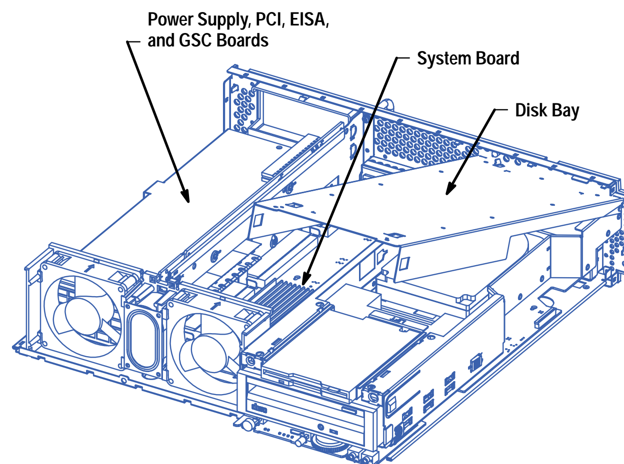


Fig. 5. *The main tray assembly.*

EMI

The tray assembly slides into a metal can. With this approach, the EMI (electromagnetic interference) interface is limited to the perimeter of the rear panel. Once the tray is removed, there is easy access to the option boards, memory modules, second-level cache modules, optional fast-wide SCSI interface board, power supply, disk drives, speaker, fans, and the CPU chip. The system board is accessible by removing the disk bay, which is secured by only one screw and a few cables.

Disk drives can be accessed without removing the disk bay from the main tray simply by removing the snap-on cover. The disks are mounted using plastic brackets so that they can be changed without tools (Fig. 6). A fan was added to the bottom of the disk bay to provide enhanced disk cooling since successive generations of disks consume more power. Removing the backplane is slightly more difficult, requiring all modules to be removed first.

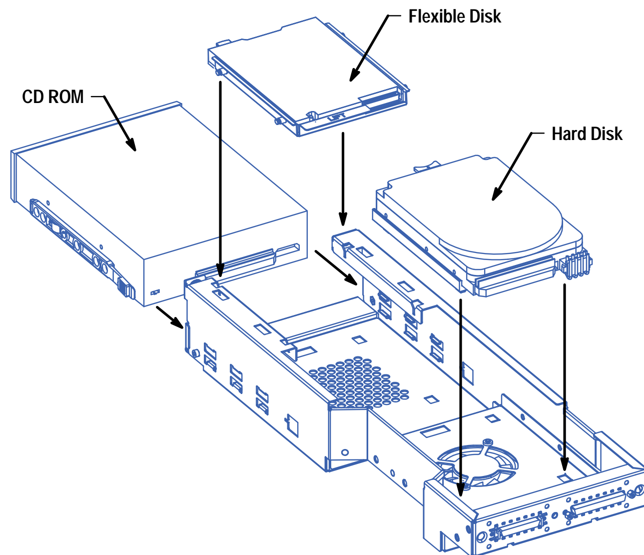


Fig. 6. *The disk bay.*

Manufacturing

Working with manufacturing included performing a supply chain analysis[1] as part of the total system cost analysis. Design efforts produced detailed material lists that were used to determine an overall system cost. Several design scenarios were developed with mechanically exploded models and material lists. The cost model for the B-class workstation was not limited just to the material content of the product, but also included system interconnect costs, parts procurement costs, part placement costs, printed circuit board electrical and system functional testing costs, and system support costs.

Manufacturing and field-support representatives were involved in defining the system for manufacturability, inventory control, and configurability to reduce the system cost. Design scenarios were then evaluated against each design objective.

Initial prototypes were assembled and disassembled by manufacturing personnel to provide a hands-on critique of the designs. These inputs were fed back into the design in the early stages of development.

Serviceability

One of the challenges of this single-board solution was to make the system board accessible for service. We wanted to have the board slide in and out, but there were connectors and switches on both edges of the board. In addition, the connectors had to be accessible through the rear panel. To allow the board to slide into the package we added a small tray to the bottom of the system board that could slide along card guides. One of the design requests from service representatives was to be able to service the system board without removing the rear panel. To accomplish this, the rear-panel connectors were recessed and a separate small bulkhead attached with a sliding EMI interface to the rear panel. This bulkhead remains attached to the board in a board replacement.

Another serviceability concern was the alignment of the power button, mute button, volume knob, audio jacks, and LEDs located on the front of the product. In the chassis, the main tray engages alignment pins, which serve to lock the tray to the chassis during vibration and shock. Because of the tolerance stackup from the front panel through the can, tray, backplane, system board, and all the connectors and buttons, we were concerned that the cosmetics at the front would be unacceptable. To improve alignment, we mounted these connectors on a long, thin section of the printed circuit board that would flex and be supported by a metal brace so that the front section could move relative to the rest of the board. We added aligning forks to the front panel to position just that section of board. With this method, we were able to locate these connectors accurately.

Manufacturing also assisted in improving the design through their participation in design reviews. One suggestion lead us to abandon the captive rear-panel fasteners that we had been planning to use. If the captive screws are not properly aligned, they can be cross-threaded and stripped, or the captive nut on the chassis may be damaged. Consequently, the whole tray would need to be replaced just for a simple nut or screw. Instead, we designed custom thumbscrews with an unthreaded nose section to align the screw before the threads engage. This minimizes cross-threading. To save labor costs we also used a coarse thread to reduce the number of rotations necessary to remove and install them.

Another goal was to reduce the number of screw types. We tried to standardize on a single screw used in our earlier option boards because this was the one screw that we could not change. We used it to attach the power supply and the disk bay. To reduce screw count, the main fans and speaker snap in place. The backplane slides in place with keyhole standoffs and forks in the main tray. When the power supply is installed, two pins from the power supply trap the backplane in place. The power supply is supported with two screws and the two pins that are routed through the backplane into the backplane support. The

power supply has floating connectors so that stresses from vibration and shock are not transmitted between the backplane and the power supply via the connectors.

One of the primary objectives was upgradability. Upgrades can be easily accomplished by a simple swap of the system board. Since everything is on one board, there are no issues with incompatibilities between different versions of the I/O and CPU. The small I/O bulkhead stays with the board so the main tray assembly need not change. Sufficient extra height remains where the CPU and memory are located so that future high-power CPUs have room for larger heat sinks or even small daughter boards if more board real estate is needed.

Processor and System Verification

The verification effort for the PA 7300LC and the B-class and C-class products was also a joint effort. Shmoo tests were conducted simultaneously on both the B-class and C-class workstations.

A shmoo test is designed to verify the product under voltage, temperature, and frequency extremes. Its intention is to electrically stress the system under test to within and beyond its operating limits. This process is part of our electrical characterization of the processor and system. A shmoo test is an important part of our product development cycle. By pushing the system to its electrical extremes, we hope to reveal any design weaknesses that could affect the operation and performance of the system under extreme operating conditions. It often uncovers weaknesses in both chip and board designs. These might include signal cross talk, chip-drive capability, slow-speed paths at high temperatures, or board-level clocking problems.

To achieve superior product quality, both processor and system shmoo tests were performed on B-class systems. The processor shmoo test focused on the core processor, caches, memory, and GSC bus. The system shmoo test emphasized peripherals and I/O, including the expansion I/O on the GSC, EISA, and PCI buses.

Since the PA 7300LC was designed to work in both B-class and C-class systems, it was tested in both systems. Processor characterization was performed in the C-class systems by the PA 7300LC design team. Simultaneously, the B-class system design team completed the processor shmoos in a B-class system. Both the B-class and C-class system design teams completed system shmoos with the PA 7300LC in their respective environments.

The parallel verifications of the PA 7300LC in the B-class and C-class systems complemented each other, providing opportunities for leveraging and making the debug process go smoother. One of the issues discovered during the processor shmoo test was the limited operating frequency of the GSC bus. This was caused by the length and load on the bus and a threshold problem on the PA 7300LC. The combined efforts of the PA 7300LC processor and B-class system design teams extended the operating frequency of the GSC bus in our systems and provided the desired performance. The PA 7300LC design team corrected the threshold problem and the B-class team shortened the GSC bus, which slightly changed its characteristic impedance and helped to alleviate the problem.

Processor-level electrical verification has three main goals: uncover electrical (nonfunctional) bugs in the system, find critical speed paths that limit the maximum frequency of the processor, and provide correlation between the IC tester frequency and the eventual system frequency. The third goal had the biggest impact on costs. As development progressed, it became obvious to the PA 7300LC and B-class teams that the frequency mix (132 MHz to 160 MHz) between the IC tester and the system was not meeting marketing requirements. The correlation effort between the teams uncovered ways to enhance the system electrical and thermal environments to bring the yield mix and market demand together. The close cooperation between the two teams enabled the quick identification of a solution to the problem. We made alterations to the system's thermal cooling environment, allowing us to run the PA 7300LC at a higher frequency, something we could not do in the original cooling environment.

Over the years, many efforts have been made to address and improve the shmoo test process at both the processor and system level. While processor shmoo testing reveals many system level problems, its primary focus is still the processor, cache, and memory subsystems, rather than the I/O subsystems. As I/O bus speed and peripheral interface IC complexity has increased, it has become more important to address the I/O subsystems in shmoo testing. The PA 7300LC was designed to make complete system shmoos more practical for this reason. The clock circuitry for the PA 7300LC was designed to permit overriding the nominal clock frequency while maintaining the correct synchronous relationship between the processor and I/O clocks. This allowed us to vary the frequency of operation more easily over a larger range of operation than in past products.

One of the challenges for system shmoo testing in B-class systems was the range of new system components that had to function correctly together during testing. As with the processor shmoo, system testing attempted to stress the electrical design of the new components by operating them under extremes of temperature, voltage, and frequency. In addition to the core I/O components, various expansion I/O cards were selected to verify complete system functionality.

The extensive system shmoo testing of the B-class system led to the optimization of several circuits and resulted in a higher-performing, more robust system. We have come to believe that shmoo tests are an indispensable part of our product development. Besides helping to catch potential problems before introduction, shmoo tests also make post-product support and maintenance easier.

Conclusion

Cooperative efforts between many functional areas such as manufacturing, service and support, marketing, firmware development, and the PA 7300LC chip development team together with the electrical and mechanical system designers have produced the B-class workstations. The closely coupled system design approach has yielded a workstation that provides significant value to our customers.

References

1. G. A. Kruger, "The Supply Chain Approach to Planning and Procurement Management," *Hewlett-Packard Journal,* Vol. 48, no. 1, February 1997, pp. 28-38.

Testing Safety-Critical Software

Testing safety-critical software differs from conventional testing in that the test design approach must consider the defined and implied safety of the software at a level as high as the functionality to be tested, and the test software has to be developed and validated using the same quality assurance processes as the software itself.

by Evangelos Nikolaropoulos

Test technology is crucial for successful product development. Inappropriate or late tests, underestimated testing effort, or wrong test technology choices have often led projects to crisis and frustration. This software crisis results from neglecting the imbalance between constructive software engineering and analytic quality assurance. In this article we explain the testing concepts, the testing techniques, and the test technology approach applied to the patient monitors of the HP OmniCare family.

Patient monitors are electronic medical devices for observing critically ill patients by monitoring their physiological parameters (ECG, heart rate, blood pressure, respiratory gases, oxygen saturation, and so on) in real time. A monitor can alert medical personnel when a physiological value exceeds preset limits and can report the patient's status on a variety of external devices such as recorders, printers, and computers, or simply send the data to a network. The monitor maintains a database of the physiological values to show the trends of the patient's status and enable a variety of calculations of drug dosage or ventilation and hemodynamic parameters.

Patient monitors are used in hospitals in operating rooms, emergency rooms, and intensive care units and can be configured for every patient category (adult, pediatric, or neonate). Very often the patient attached to a monitor is unconscious and is sustained by other medical devices such as ventilators, anesthesia machines, fluid and drug pumps, and so on. These life-sustaining devices are interfaced with the patient monitor but not controlled from it.

Safety and reliability requirements for medical devices are set very high by industry and regulatory authorities. There is a variety of international and national standards setting the rules for the development, marketing, and use of medical devices. The legal requirements for electronic medical devices are, as far as these concern safety, comparable to those for nuclear plants and aircraft.

In the past, the safety requirements covered mainly the hardware aspects of a device, such as electromagnetic compatibility, radio interference, electronic parts failure, and so on. The concern for software safety, accentuated by some widely known software failures leading to patient injury or death, is increasing in the industry and the regulatory bodies. This concern is addressed in many new standards or directives such as the Medical Device Directive of the European Union or the U.S. Food and Drug Administration. These legal requirements go beyond a simple validation of the product; they require the manufacturer to provide all evidence of good engineering practices during development and validation, as well the proof that all possible hazards from the use of the medical instrument were addressed, resolved, and validated during the development phases.

The development of the HP OmniCare family of patient monitors started in the mid-1980s. Concern for the testing of the complex safety-critical software to validate the patient monitors led to the definition of an appropriate testing process based on the ANSI/IEEE software engineering standards published in the same time frame. The testing process is an integral part of our quality system and is continuously improved.

The Testing Process

During the specifications phase of a product, extended discussions are held by the crossfunctional team (especially the R&D and software quality engineering teams) to assess the testing needs. These discussions lead to a first estimation of the test technology needed in all phases of the development (test technology is understood as the set of all test environments and test tools). In the case of HP patient monitors the discussion started as early as 1988 and continues with every new revision of the patient monitor family, refining and in some cases redefining the test technology. Thus, the test environment with all its components and the tools for the functional, integration, system, and localization testing evolved over a period of seven years. Fig. 1 illustrates the testing process and the use of the tools.

The test process starts with the test plan, a document describing the scope, approach, resources, and schedule of the intended test activities. The test plan states the needs for test technology (patient simulators, signal generators, test tools, etc.). This initiates subprocesses to develop or buy the necessary tools.

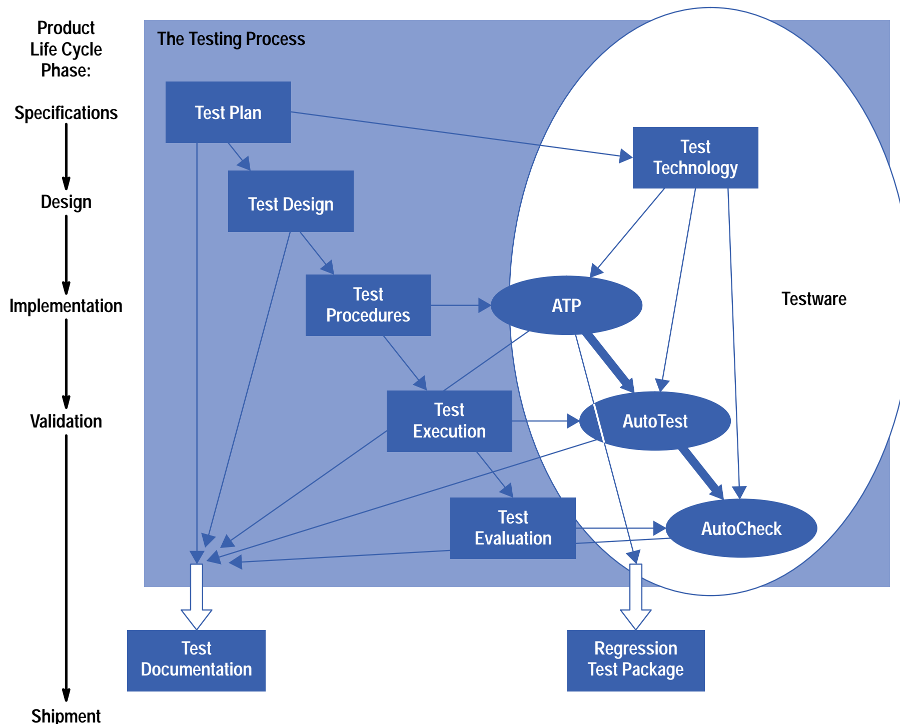


***Fig. 1.** The software testing process for HP OmniCare patient monitors.*

Test design is the documentation specifying the details of the test approach and identifying the associated tests. We follow three major categories of test design for the generation of test cases (one can consider them as the main directions of the testing approach): white box, black box, and risk and hazard analysis.

The white box test design method is for design test, unit test, and integration tests. This test design is totally logic-driven and aims mainly at path and decision coverage. Input for the test cases comes from external and internal specifications (design documents). The test design for algorithm validation (proof of physiological measurement algorithms) follows the white box method, although sometimes this is very difficult, especially for purchased algorithms.

The black box test design method is for functional and system test. This test design is data-driven and aims at the discovery of functionality flaws by using exhaustive input testing. Input for the test cases comes from the external specifications (as perceived by the customer) and the intended use in a medical environment.

Risk and hazard analysis is actually a gray box method that tries to identify the possible hazards from the intended and unintended use of the product that may be potential sources of harm for the patient or the user, and to suggest safeguards to avoid such hazards. Consider, for instance, a noninvasive blood pressure measurement device that may overpump. Hazard analysis is applied to both hardware (electronic and mechanical) and software, which interoperate and influence each other. The analysis of events and conditions leading to a potential hazard (the method used is the fault tree, a cause-and-effect graph) goes through all possible states of the hardware and software. The risk level is estimated (the risk spectrum goes from catastrophic to negligible) by combining the probability of occurrence and the impact on health. For all states with a risk level higher than negligible, appropriate safeguards are designed. The safeguards can be hard or soft (or in most cases, a combination of both). The test cases derived from a hazard analysis aim to test the effectiveness of the safeguards or to prove that a hazardous event cannot occur.

Test cases consist of descriptions of actions and situations, input data, and the expected output from the object under test according to its specifications.

Test procedures are the detailed instructions for the setup, execution, and evaluation of results for one or more test cases. Inputs for their development are the test cases (which are always environment independent) and the test environment as defined and designed in the previous phases. One can compare the generation and testing of the test procedures to the implementation phase of code development.

Testing or test execution consists of operating a system or component under specified conditions and recording the results. The notion of testing is actually much broader and can start very early in the product development life cycle with specification inspections, design reviews, and so on. For this paper we limit the notion of testing to the testing of code.

Test evaluation is the reporting of the contents and results of testing and incidents that occurred during testing that need further investigation and debugging (defect tracking).

While test design and the derivation of test procedures are done only once (with some feedback and rework from the testing in early phases, which is also a test of the test), testing and test evaluation are repeatable steps usually carried out several times until the software reaches its release criteria.

Various steps of the testing process also produce test documentation, which describes all the plans for and results of testing. Test or validation results are very important for documenting the quality of medical products and are required by regulatory authorities all over the world before the product can be marketed.

The regression test package is a collection of test procedures that can be used for selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified product and legal requirements.

From the Test Plan to the Testware

Ten or fifteen years ago it was perhaps enough to give a medical instrument to some experts for clinical trials, algorithm validation, and test. The instrument had a simple display, information placement was not configurable, and the human interface was restricted to a few buttons and knobs. All the attention was on the technical realization of the medical solution (such as ECG monitoring), and software, mainly written by the electrical engineers who designed the hardware, was limited to a few hundred statements of low-level languages.

Today the medical instruments are highly sophisticated open-architecture systems, with many hundreds of thousands lines of code. They are equipped with complex interfaces to other instruments and the world (imagine monitoring a patient over the Internet—a nightmare and a challenge at the same time). They are networked and can be remotely operated. This complexity and connectivity requires totally new testing approaches, which in many cases, are not feasible without the appropriate tooling, that is, the ***testware***.

Discussion of the test plan starts relatively early in the product life cycle and is an exit criterion for the specifications phase. One of the major tasks of the testing approach is the assessment of the testing technology needed. The term technology is used here in its narrow meaning of process plus hardware and software tools.

The testing technology is refined in the next phases (design and implementation) and grows and matures as the product under development takes shape. On the other hand, the testing tools must be in place before the product meets its implementation criteria. This means that they should be implemented and validated before the product (or subproduct) is submitted for validation. This requirement illustrates why the test technology discussion should start very early in the product life cycle, and why the testware has a "phase shift to the left" with respect to the product validation phase (see Fig. 2).

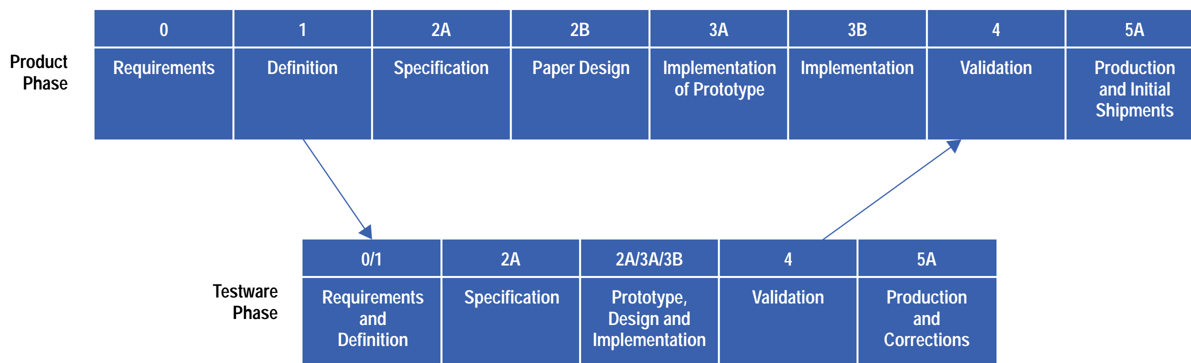


Fig. 2. *Relationship of product to testware development phases.*

Test Tool (Testware) Development

Testware development follows the same product life cycle as the product under development. The phases are:

- Requirements and Definition Phase. The test needs are explained according to the test plan and the high-level test design. Alternatives are discussed and one is selected by the software quality team.
- Specifications Phase. The tool is described in as much detail as possible to enable the testers to start work with their test cases and test procedures as if the tool were already available. These specifications are reviewed (or formally inspected) by the product development and test teams, approved, and put under revision control.
- Design and Implementation Phase. Emphasis is on the rapid development of engineering prototypes of the tools, which again are the object of review. These prototypes are used by the test team for low-level test design and first test trials.

- Validation Phase. The test tool is validated against its specifications. The most up-to-date revision of the patient monitor under development is used as a test bed for the tool's validation. Notice the inversion of roles: the product is used to test the test tool! Our experience shows that this is the most fruitful period for finding bugs in both the tool and the product. A regression package for future changes is created. First hardware construction is started if hardware is involved.
- Production Phase. The tool is officially released, hardware is produced (or bought), and the tool is used in the test environment. After some period of time, when the tool's maturity for the test purposes has been assessed, the tool is made public for use by other test departments, by marketing for demos, by support, and so on.

Fig. 2 demonstrates the main difficulty of testware development: the test tool specifications can be created after the product specifications, but from this point on, all of the testware development phases should be earlier than the product development phases if the product is to be validated in a timely manner.

Besides the shift of the development phases, there is also the testware dilemma: as the progress of the product's design and the test design leads to new perceptions of how the product can be tested, new opportunities or limitations appear that were previously unknown, and influence the scope of the testware. The resulting changes in the testware must then be made very quickly, more quickly than the changes in the product. Only the application of good hardware and software engineering processes (the tester is also a developer) can avoid having the wrong test tool for the product.

AutoTest

The test technology assessment for the patient monitors led us to the development of a number of tools that could not be found on the market. This make instead of buy decision was based mainly upon the nature of the patient monitors, which have many CPUs, proprietary operating systems and networks, proprietary human interfaces, true real-time behavior, a lot of firmware, and a low-level, close-to-the-machine programming style. Testing should not be allowed to influence the internal timing of the product, and invasive testing (having the tests and the objects under test on the same computer) had to be avoided.

The first tool developed was AutoTest,[1] which addressed the need for a tool able to (1) simulate the patient's situation by driving a number of programmable patient simulators, (2) simulate user interactions by driving a programmable keypusher, and (3) log the reaction of the instrument under test (alarms, physiological values, waves, recordings, etc.) by taking, on demand, snapshots of the information to send to the medical network in a structured manner.

AutoTest was further developed to accept more simulators of various parameters and external non-HP devices such as ventilators and special measurement devices attached to the HP patient monitor. AutoTest now can access all information traveling in the internal bus of the instrument (over a serial port with the medical computer interface) or additional information sent to external applications (see ***Article 14***).

AutoTest is now able to:

- Read a test procedure and interpret the instructions to special electronic devices or PCs simulating physiological signals
- Allow user input for up to 12 patient monitors simultaneously over different keypushers (12 is the maximum number of RS-232 interfaces in a PC)
- Allow user input with context-sensitive keypushing (first search for the existence and position of an item in a menu selection and then activate it)
- Maintain defined delays and time dependencies between various actions and simulate power failure conditions
- Read the reaction of the device under test (alarms, physiological values and waves with all their attributes, window contents, data packages sent to the network, overall status of the device, etc.)
- Drive from one PC simultaneously the tests of up to four patient monitors that interact with each other and exchange measurement modules physically (over a switch box)
- Execute batch files with any combination of test procedures
- Write to protocol files all actions (user), simulator commands for physiological signals (patient), and results (device under test) with the appropriate time stamps (with one-second resolution).

AutoCheck

The success of AutoTest and the huge amount of data produced as a result of testing quickly led to the demand for an automated evaluation tool. The first thoughts and desires were for an expert system that (1) would represent explicitly the specifications of the instrument under test and the rules of the test evaluation, and (2) would have an adaptive knowledge base. This solution was abandoned early for a more versatile procedural solution named AutoCheck (see ***Article 14***). By using existing compiler-building knowledge we built a tool that:

- Enables the definition of the expected results of a test case in a formal manner using a high-level language. These formalized expected results are part of the test procedure and document at the same time the pass-fail criteria.
- Reads the output of AutoTest containing the expected and actual results of a test.

- Compares the expected with the actual results.
- Classifies and reports the differences according to given criteria and conditions in error files similar to compiler error files.

AutoCheck has created totally new and remarkable possibilities for the evaluation of tests. Huge amounts of test data in protocol files (as much as 100 megabytes per day) can be evaluated in minutes where previously many engineering hours were spent. The danger of overlooking something in lengthy protocols full of numbers and messages is eliminated. AutoCheck provides a much more productive approach for regression and local language tests. For local language tests, it even enables automatic translation of the formalized pass-fail criteria during run time before comparison with the localized test results (see ***Article 15***).

ATP

The next step was the development of a sort of test generator that would:

- Be able to write complex test procedures by keeping the test design at the highest possible level of abstraction
- Enable greater test coverage by being able to alter the entry conditions for each test case
- Eliminate the debugging effort for new test procedures by using a library of validated test primitives and functions
- Take account of the particularities and configurations of the monitors under test by automatic selection of the test primitives for each configuration
- Produce (at the same time as the test setup and the entry conditions) the necessary instructions for automated evaluation by AutoCheck.

The resulting tool is called ATP (Automated Test Processor, see ***Article 13***). Like AutoCheck, ATP was developed by using compiler-building technology.

Results

Good test design can produce good and reliable manual tests. The industry has had very good experience with sound manual tests in the hands of experienced testers. However, there is no chance for manual testing in certain areas of functionality such as interprocess communication, network communication, CPU load, system load, and so on, which can only be tested with the help of tools. Our process now leaves the most tedious, repetitive, and resource-intensive parts of the testing process for the automated testware:

- ATP for the generation of test procedures in a variety of configurations and settings based on a high-level test design
- AutoTest for test execution, 24 hours a day, 7 days a week with unlimited combinations of tests and repetitions
- AutoCheck for the automated evaluation of huge amounts of test protocol data.

One of the most interesting facets is the ability of these tools to self-document their output with comments, time stamps, and so on. Their output can be used without any filtering to document the test generation with pass-fail criteria, test execution with all execution details (test log), and test evaluation with a classification of the discrepancies (warnings, errors, range violations, validity errors, etc.).

Automated testware provides us with reliable, efficient, and economical testing of different configurations or different localized versions of a product using the ***same*** test environment and the ***same*** test procedures. By following the two directions of (1) automated testware for functional, system, and regression tests (for better test coverage), and (2) inspection of all design, test design, and critical code (as identified by the hazard analysis), we have achieved some remarkable results, as shown in Fig. 3.

Through the years the patient monitor software has become more and more complex as new measurements and interfaces were added to meet increased customer needs for better and more efficient healthcare. Although the software size has grown by a factor of three in six years (and with it the testing needs), the testing effort, expressed as the number of test cycles times the test cycle duration, has dropped dramatically. The number of test cycles has dropped or remained stable from release to release.

The predictability of the release date, or the length of the validation phase, has improved significantly. There has been no slippage of the shipment release date with the last four releases.

The ratio of automated to manual tests is constantly improving. A single exception confirms the rule: for one revision, lack of automated testware for the new functionality—a module to transfer a patient database from one monitor to another—forced us to do all tests for this function manually.

The test coverage and the coverage of the regression testing has improved over the years even though the percentage of regression testing in the total testing effort has constantly increased.

See Subarticle 12a: ***Processor and System Verification***.

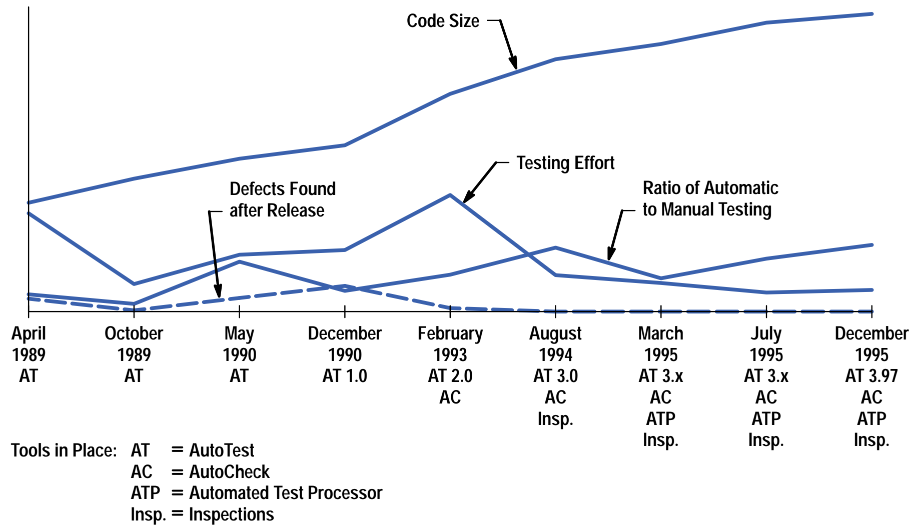


Fig. 3. *Trends of testing metrics for patient monitors.*

Conclusion

Software quality does not start and surely does not end with testing. Because testing, as the term is used in this article, is applied to the final products of a development phase, defect discovery through testing always happens too late in each phase of product development. All the experience gained shows that defect prevention activities (by applying the appropriate constructive software engineering methods during product development in all phases) is more productive than any analytic quality assurance at the end of the development process.

Nevertheless, testing is the ultimate sentinel of a quality assurance system before a product reaches the next phase in its life cycle. Nothing can replace good, effective testing in the validation phase before the product leaves R&D to go to manufacturing (and to our customers). Even if this is the only and unique test cycle in this phase (if the defect prevention activities produced an error-free product, which is still a vision), it has to be prepared very carefully and be well documented. This is especially true for safety-critical software, for which, in addition to functionality, the effectiveness of all safeguards under all possible failure conditions has to be tested.

In this effort, automated testware is crucial to ensuring reliability (the testware is correct, validated, and does not produce false negative results), efficiency (no test reruns because of testware problems), and economy (optimization of resources, especially human resources).

Reference

1. D. Göring, "An Automated Test Environment for a Medical Patient Monitoring System," ***Hewlett-Packard Journal***, Vol. 42, no. 4, October 1991, pp. 49-52.

A High-Level Programming Language for Testing Complex Safety-Critical Systems

Dealing with an enormous amount of data is characteristic of validating complex and safety-critical software systems. ATP, a high-level programming language, supports the validation process. In a patient monitor test environment it has shown its usefulness and power by enabling a dramatic increase in productivity. Its universal character allows it to migrate validation scenarios to different products based on other architectural paradigms.

by Andreas Pirrung

This article concentrates on the specific problem of transforming a test design into concrete automatic test procedures. For a systematic overview and context the reader is referred to ***Article 12***. As described in that article, the test design identifies and documents the test set for a given product. It is derived from external and internal specifications, software quality engineer expertise, and risk and hazard analysis results. A test design is normally informal and describes test cases and test data on a high, abstract level, independent of the test environment. On the other hand, an automatic test procedure has to deal with all the details of the test environment and reflects the abstraction capabilities of the existing tools.

In our software quality engineering department the automatic test environment is based upon two major tools: AutoTest[1] and AutoCheck. The first is a test execution tool and the latter is responsible for test evaluation (see ***Article 14***). AutoTest is very close to the devices it controls and requires detailed commands on a low abstraction level. AutoCheck has to cope with the detailed low-level information produced by AutoTest and therefore also requires input on a detailed, low abstraction level (see Fig. 1). The strengths of the low-level interfaces are their flexibility and adaptability to various different test situations.

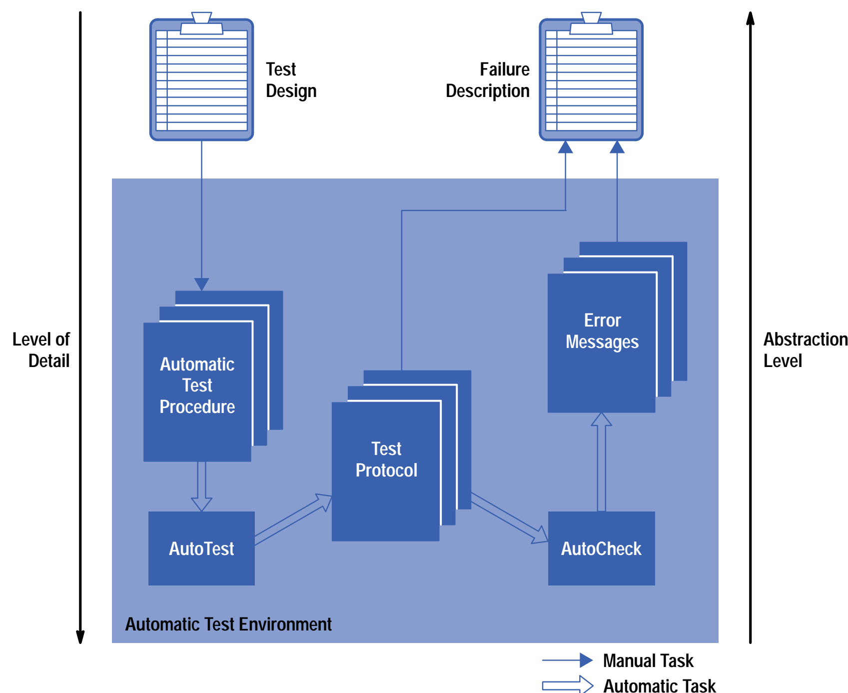


Fig. 1. *Patient monitor test process.*

There are some difficulties with the process shown in Fig. 1. The test engineer spends a lot of time transforming test designs into automatic test procedures. There is a large gap in abstraction level between the test design and the test procedure. The detail level is low in the test design, but very high in the test procedure. It is an error-prone, time-consuming task to bridge

this gap manually. Because resources are always restricted, the software quality engineer has less time for a more intensive test design.

Because the test procedures have a high explicit redundancy, it is difficult to maintain and evolve test procedures. The explicit redundancy is high because AutoTest and AutoCheck do not support data and functional abstraction, nor do they offer control flow elements. A piece of code may exist in many copies scattered over the test procedures. If the test requires a change in the code pattern, for instance because of changes in the timing behavior of the system under test (in our case a patient monitor), the test engineer has to update numerous copies of this code pattern. The risk of forgetting one pattern or introducing an error in a test procedure increases with the number of update steps. It is very resource-consuming to adapt test procedures to a change in system behavior.

The test procedure describes a static test scenario. Therefore, the test engineer has to document the test setup completely. Every parameter that influences the test environment and consequently the test execution must be carefully controlled before starting the automatic test. Our test environment consists of so many simulators, forcing devices, and sensing devices that sometimes tests need to be repeated because the initial conditions are wrong. The problem is that the automatic test procedure describes only one specific test situation. It is not possible to use parameters for the test procedure and to feed in the actual start parameters at the beginning of the test execution to get more general and robust test procedures. Even a slight change in the start condition may require an adapted or nearly new test procedure.

The test coverage is limited because the test data is coded within the test procedure. The repetition of a test case with other test data requires a modified duplicate of the test procedure. Again, it would help if a test case were able to profit from data abstraction and parameters, enabling the test engineer to formulate more general test procedures.

AutoTest and AutoCheck do not support the statistical structural testing approach. It is therefore not possible to select test data randomly (see ***Subarticle 13a*** "Structural Testing, Random Testing, and Statistical Structural Testing").

The following section illustrates the above problems by presenting a practical example to demonstrate the transformation of a high-level test design to an automatic test procedure.

A Practical Example

Patient monitors are electronic medical devices used to monitor physiological parameters of critically ill patients in intensive care units or operating rooms. They alert the medical staff when physiological parameters exceed preconfigured limits. In this example, we will concentrate on a well-known physiological parameter, the heart rate. The nurses and doctors want to get an immediate alarm when the heart rate falls below a given lower limit or exceeds a given higher limit. A malfunction of the monitor may result in the death of a patient, so this functionality is safety-critical and must be validated very carefully by the vendor of the patient monitor. The example illustrates a test design for heart rate alarm testing and the transformation process to the appropriate automatic test procedure.

Fig. 2 shows the upper and lower alarm limits for the heart rate parameter. The data space can be divided into three subdomains (equivalence classes):

- The normal heart rate domain—the interval between the alarm limits. The monitor should not alarm for data points taken from this area.
- The lower alarm domain. All data here produces a low limit alarm.
- The upper alarm domain. All data here produces a high limit alarm.

A classic method of testing the alarm behavior is to select representatives from each of the three areas and check that the monitor reacts as expected. Fig. 2 shows the selected data points and their order in time.

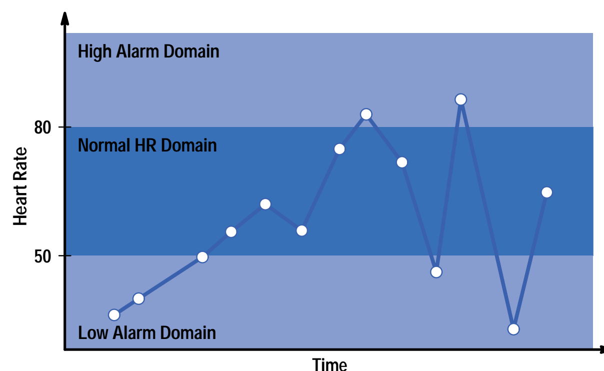


Fig. 2. *Heart rate (HR) alarm test principle.*

This graphical representation of the heart rate (HR) alarm test leads to the following test design:

- Test Case 1:
  - Action(s): Configure HR alarm limits to 50/80. Apply signal HR 45.
  - Expected: Low limit alarm with text "**HR 45 < 50".
- Test Case 2:
  - Action(s): Apply HR signal 49.
  - Expected: Low limit alarm remains with text "**HR 49 < 50".
- Test Case 3:
  - Action(s): Apply HR signal 50.
  - Expected: Low limit alarm disappears.
- Test Case 4:
  - Action(s): Select 5 different HR values between 50 and 80 and apply them.
  - Expected: No HR alarm for each of the selected values.

These few test cases are enough to demonstrate the principles of test design. The appropriate automatic test procedure description for test case 1 on the AutoTest and AutoCheck level then looks like:

```
.....
//--------------- Test Case 1: -----------------
// Adjust alarm limits to 50-80.
//
merlin param
merlin "HR"
merlin f2
merlin f7
merlin f3 -n48
merlin f6 -n12
merlin f4 -n34
merlin f5 -n5
//
// Apply HR signal 45 (normal sinus beat).
//
siml NSB45
//
// Delay 10 s : after that time the alarm must
// be announced.
wait 10
//
// Check if low limit alarm "** HR 45 < 50" is
// present.
^ Verify begin
^ Alarm "HR" < al_min;
// Low alarm active.
^ sound is c_yellow;
// Limit alarm sound audible.
^ Verify end
mecif tune HR 10
// Tune 10 the HR numeric of the patient
// monitor.
.....
```

This example demonstrates the difference in abstraction between a test design (high abstraction) and an automatic test procedure (low abstraction) and gives an impression of the difficulties noted above. An automatic test description language designed to alleviate these difficulties should offer abstraction capabilities to hide details and to compose complex functions from simpler functions. Like every high-level programming language, it should bridge the abstraction gap automatically. In the following section a solution is presented that meets these needs.

The ATP Programming Language

Often specific problems need *basic processors* like AutoTest or AutoCheck to perform some operation such as pushing keys, simulating patient signals, simulating powerfail conditions, and so on. A straightforward solution might extend the command interfaces of AutoTest and AutoCheck to support data and functional abstraction, provide control flow elements like

conditions and loops, and allow further probabilistic data generation. This would probably eliminate the difficulties mentioned above. However, redundant effort would have to be spent implementing an abstract command interface again and again.

Our solution is ATP (Automated Test Procedure), a high-level programming language that offers the abstraction facilities and makes it possible to integrate basic processors smoothly. ATP allows the integration of many different basic processors, so the coordination of the basic processors is much easier than with separate control.

The following is a typical ATP routine representing the automatic test procedure for the heart rate alarm test:

```
DEFINE AlarmTest ( IN PatientSize CHECK IN {
              "ADULT", "PEDIATRIC", "NEONATE"
                  },
              IN Category CHECK IN {"OR", "ICU"}
                 )

DESCRIPTION
  PURPOSE :
    This routine demonstrates some of the
    ATP features. It is an automatic test
    procedure testing the HR alarm
    capabilities.

  SIGNATURE :
    AlarmTest ( <PatientSize>, <Category> )
END DESCRIPTION

LOCAL   HRValue,/* selected HR Value          */
        AL,     /* HR low alarm limit         */
        AH,     /* HR high alarm limit        */
        walk    /* repetition counter         */

/*-------- Initialize the Repository ---------*/

LINK Repository <- "$PatientMonitorRepository"
Repository:Init (PatientSize, Category)
              /* Declare the use of a function
                 repository and initialize the
                 repository link. This gives
                 context-specific access to all
                 available functions for the
                 given PatientSize and Category.
                                              */

/*--------------- Test Case 1 -----------------
  Action(s): Configure HR Alarm Limits to 50/90.
             Apply Signal HR 45.
  Expected:  Low limit alarm with text
             "**HR 45<50".
  ----------------------------------------------*/
HR:SetAlarmLimits (50, 90)
HR:SimulateValue (45)
HR:CheckAlarm (10, "** HR 45 < 50")
              /* Check for limit alarm after
                 delay of 10 s.               */

/*--------------- Test Case 2 -----------------
  Action(s):  Apply Signal HR 49
  Expected:   Low limit alarm remains with text
              "**HR 49<50" (alarm string is
              updated without delay).
----------------------------------------------*/
HR:SimulateValue (49)
HR:CheckAlarm (0, "** HR 49 < 50")
```

```
/*--------------- Test Case 3 -----------------
  Action(s):  Apply Signal HR 50.
  Expected:   Low alarm limit disappears.
-----------------------------------------------*/
HR:SimulateValue (50)
HR:CheckNoAlarm (5)
            /* No HR alarm present after 5 s. */

/*--------------- Test Case 4 -----------------
  Action(s):  Select 5 different HR values
              between 50 and 80 and apply them.
  Expected:   No HR Alarm for each of the
              selected values.
 ----------------------------------------------*/

walk <- 1
              /* Randomly choose some HR values
                 in the range 50/80, i.e., no
                 alarm condition exists and
                 therefore no HR alarm must
                 be visible and audible.      */
WHILE walk <= 5 DO
  HRValue <- RANDOM (50, 80, 1)
  HR:SimulateValue (HRValue)
  HR:CheckNoAlarm (0)
  walk  <- walk + 1
ENDWHILE

.....

END AlarmTest
```

An automatic test procedure for a random heart rate alarm test in ATP might look like the following:

```
DEFINE RandomAlarmTest ( IN repetitions )

DESCRIPTION
  PURPOSE :
    Random HR Alarm Test

  SIGNATURE :
    RandomAlarmTest ( <repetitions> )
END DESCRIPTION

LOCAL   AL,     /* low alarm limit            */
        AH,     /* high alarm limit           */
        walk,
        HRValue

/*-------- Initialize the Repository ---------*/

LINK Repository <- "$PatientMonitorRepository"
Repository:Init (CHOOSE ({"ADULT","PEDIATRIC",
                          "NEONATE"}),
                 CHOOSE ({"OR", "ICU"})
                 )
               /* Declare the use of a function
                  repository and initialize the
                  repository link. Choose
                  patient size and category
                  randomly. This gives context-
                  specific access to all avail-
                  able functions for the given
```

```
                     PatientSize and Category.   */


/*--------------------------------------------*/
                /* Randomly select valid HR
                   alarm limits, then randomly
                   select an HR value and check
                   if the monitor reacts as
                   expected.                */
walk <- 1
WHILE walk <= repetitions DO
  HR:RandomSelectAlarmLimits (AL, AH)
                /* randomly select valid alarm
                   limits                   */
  HR:SetAlarmLimits (AL, AH)
  HRValue <- RANDOM (20, 180, 1)
                /* select HR values between 20
                   and 180 with step width 1. */
  HR:SimulateValue (HRValue)
  IF HRValue < AL THEN
    HR:CheckAlarm (10, "** HR " + HRValue + "<"
                   + AL)
  ELSIF HRValue > AH THEN
    HR:CheckAlarm (10, "** HR " + HRValue + ">"
                   + AH)
  ELSE
    HR:CheckNoAlarm (5)
  ENDIF
  walk    <- walk + 1
ENDWHILE

END RandomAlarmTest
```

Even without familiarity with the syntax and semantics of the ATP language, it can be recognized that the abstraction level is higher than with plain code for the basic processors (in our case, AutoTest and AutoCheck). It is also worth noting that the automatic test procedures are not restricted to a specific patient monitor. They describe in a general and abstract way a heart rate alarm test for any patient monitor with a limit alarm concept. The differences between specific patient monitors are in the basic processor interfaces and in the primitive functions.

The ATP Concept

ATP consists of two major functional elements (see Fig. 3): a set of tools to maintain a function repository and an ATP language interpreter.

The tools give the user adequate access to the function repository, which contains well-documented, well-tested ATP functions that can be reused. This effectively reduces redundancy and increases productivity (see the next section, "Working with the Function Repository"). The tools can be grouped into:

- File-oriented tools. Check functions into and out of the function repository, compare two versions of a function, etc.
- Repository query functions. Obtain information about available functions, about an interface of a function, etc.
- Repository administration functions. Administer and maintain the structure of the repository. Allow archival and retrieval of the repository. These functions are only accessible by the repository administrator.

With these tools and a text editor, a programmer writes ATP functions by reusing existing functions from the repository. When the function repository is well-structured and offers reliable functions on an adequate abstraction level, it is easy even for an inexperienced programmer to write functions, as shown in the example above.

Interpreter

The core of the system is the ATP language interpreter. The interpreter requires an input file and an output file. The input file is an ATP function. In contrast to other common high-level languages, ATP requires one file per function. One advantage of this approach is that each ATP function is executable. There is no explicit syntactical distinction between a main routine and a subroutine. Any function can be the execution starting point and can call any other function. There is no explicit function hierarchy. The hierarchical structure is provided independently by the repository structure.

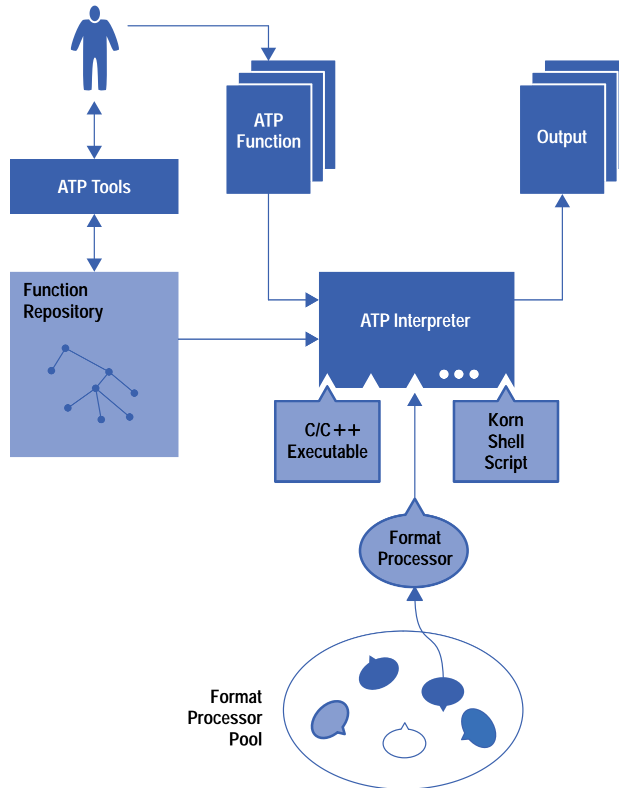


Fig. 3. *ATP concept overview.*

Another advantage is that, because each file contains only one function, it is much easier to administer the functions in the function repository. To fulfill structural requirements the functions can be grouped by any criteria. The heart rate example above uses functions that all belong to the logical functional group HR.

The interpreter output data can be written into an output file. A powerful feature of the language is its ability to integrate *format processors*. A format processor is a problem-domain-specific process (a basic processor) that can be integrated into the interpretation process. In the patient monitor testing example, AutoTest and AutoCheck are format processors. The ATP language offers syntactical elements to establish a communications channel to a format processor so that within the ATP code the user can send any information to the processor. The format processor can send back information to the ATP interpreter, which then can be sent to another format processor or logged to the output file. The creation of this ATP adapter interface is an easy task, thanks to an API that enables a programmer to implement this communication interface with ATP. If an integrated format processor is general-purpose, it can be offered to all ATP programmers. A good example is KSH, a format processor that enables ATP programmers to integrate Korn shell commands within ATP code. The format processor concept and the abstraction facilities of the ATP language offer the programmer the means to model the problem domain in an adequate and flexible way.

Working with the Function Repository

The following short tour illustrates some of the tools that are available to handle function repositories. Suppose that a programmer wants to know which functions in the repository are available for dealing with heart rate operations. Typing:

```
LibIndex
Group: "HR"
```

will, for example, produce the output:

```
CheckAlarm
CheckNoAlarm
SimulateValue
SetAlarmLimits
RandomSelectAlarmLimits
.....
```

To get information about the interface of the SetAlarmLimits function, the programmer can type:

```
TellMe
Group: "HR"
Functions: "SetAlarmLimits"
```

and will get:

```
************** HR:SetAlarmLimits **************
*
* purpose: configure the Heart Rate lower and
* upper alarm limits.
*
* signature: SetAlarmLimits(<lower alarm limit>,
*                    <higher alarm limit>)
*
* .....
```

The heart rate test example above uses this and other functions. At the beginning of the function a LINK statement declares an access path to a given function repository. Then the function repository is initialized. This initialization function introduces all available functional groups given a specific system context. At that point the programmer is able to call the functions, for example HR:SetAlarmLimits, without knowing implementation details or physical locations. It is possible to check out a function for enhancement or maintenance purposes. It is also possible to check a function into the repository so that all test engineers can use the new function.

Format Processors

Fig. 4 presents the possibilities and the flexibility of format processors. Each format processor consists of two parts: a basic processor and an ATP adapter. The basic processor is a proprietary part, that is, any executable code written by a programmer. Typically the basic processors are on a low abstraction level. The ATP adapter is the interface to ATP that allows data to be sent to ATP and received from it. This functionality is encapsulated and offered as an API.

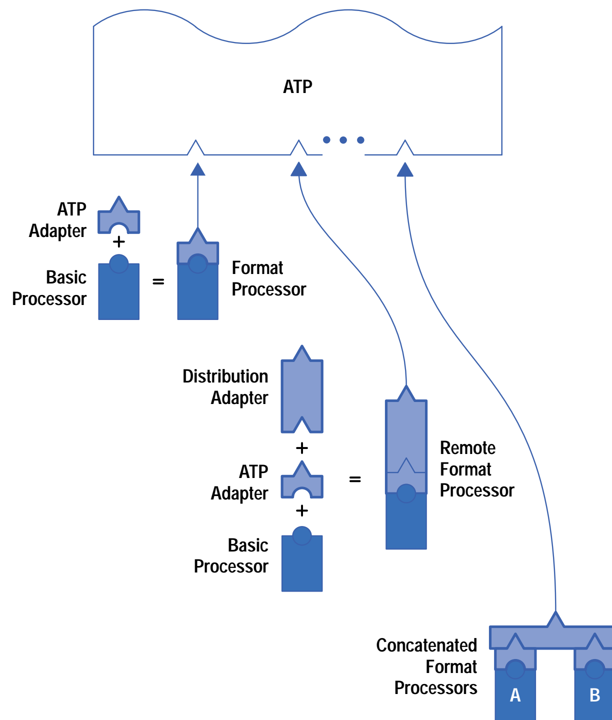


Fig. 4. Format processor functional blocks.

A format processor can be used within ATP in the following way:

```
.....
FORMAT MyFormatProcessor <- "$MY_FP_EXECUTABLE"
/* Declare the use of a format processor. */
.....
BEGIN [ MyFormatProcessor ]
.....
.....
END [ MyFormatProcessor ]
/* Use the format processor, sending and receiving information. */
.....
```

First, a specific syntactical construct introduced with the keyword FORMAT is used to declare the use of a format processor. It is then the task of ATP to control and to communicate with the format processor. All information enclosed in the syntactical bracket BEGIN [MyFormatProcessor] and END [MyFormatProcessor] represents a code template for the named format processor. ATP generates the actual code block from this code template by substituting the actual parameter values for the code template parameters. Then this code block is sent to the format processor for immediate execution. The format processor receives the code by calling API functions provided by the ATP adapter. The proprietary part of the format processor processes the received information. The format processor can send back information to ATP. ATP receives this information and logs it to the output file or redirects it to another format processor.

Remote Format Processor. If a programmer needs to integrate a format processor on another machine, for example on a PC running Windows® NT, this can be specified in the FORMAT declaration. No additional effort is required for the programmer to establish a remote format processor. The adaptation for remote control is done by ATP automatically by adding a distribution adapter.

Concatenated Format Processor. Another feature of the ATP language is the ability to concatenate existing format processors.

```
.....
FORMAT X <- .....
FORMAT Y <- .....
FORMAT Z <- X | Y
/* Concatenate format processors X and Y to Z.
This is similar to UNIX pipes. */
.....
BEGIN [ Z ]
.....
END [ Z ]
```

The code block between BEGIN [Z] and END [Z] is first sent to format processor X. The output of format processor X is sent to format processor Y. For format processor Y it makes no difference where the information comes from, that is, the concatenation is mediated by ATP automatically. Format processor Y sends its output back to ATP.

ATP in the Patient Monitor Test Environment

The concept behind ATP eased its integration into the patient monitor test environment. The impetus to develop this concept came from our experience with the test environment, as described at the beginning of this article. But the concept is more general. It is not resticted to the patient monitor test environment. ATP can be used to attack many different problems.

The integration of such a tool into an existing environment is a challenging task. ATP, like every tool, requires some effort to build up the necessary infrastructure, to support the tool, and to learn the new language. A step-by-step, three-phase integration of ATP in the test process was planned.

Phase I. Develop a patient-monitor-relevant repository structure that is easy to use and maintain. In parallel, implement a set of primitive functions to fill the repository. Then test the structure on new patient monitor functionality. In this phase ATP does not specify any format processor executable, that is, AutoTest and AutoCheck are not integrated as format processors. ATP writes the code block immediately to the output file. The generated code is then processed in a postprocessing step.

Phase II. Enhance AutoTest and AutoCheck with ATP adapters so that they can be integrated as format processors. Then the same functions used in phase I can be executed immediately without the postprocessing steps needed in phase I. The test tools are invoked by ATP.

Phase III. Complete the function repository and migrate, step by step, the existing test package to ATP functions. Then the testing package can be used again for new patient monitor products simply by replacing the format processors by new ones and by substituting some primitive functions.

Currently the phase I integration is completed (see Fig. 5). A repository structure has been proposed and evaluated in some projects. In these projects test engineers use ATP to automate the tests. ATP generates AutoTest and AutoCheck code, which

is then passed to AutoTest and AutoCheck for execution. For phase II integration, only the declaration of the AutoTest and AutoCheck format processors will change. They will then specify integratable AutoTest and AutoCheck format processors. This phase is currently in progress. Phase III has been started.

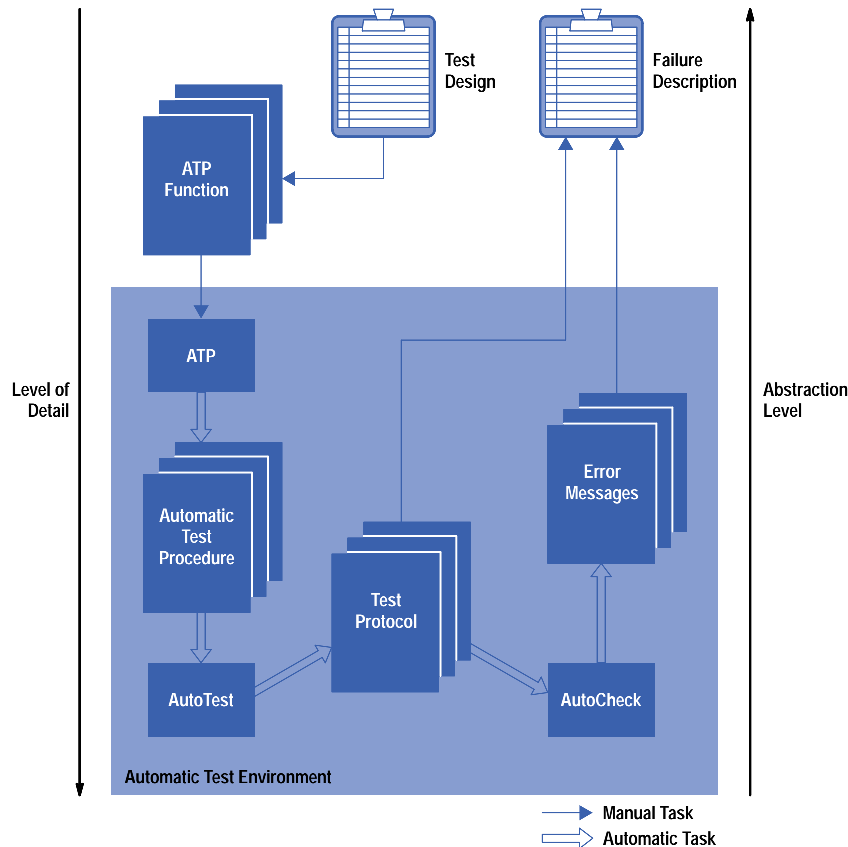


Fig. 5. *Current ATP integration in the patient monitor test environment (phase I).*

ATP Integration in Phase I: An Example

The following ATP function illustrates how ATP generates AutoCheck code. This function is the CheckAlarm function called in the heart rate alarm test used in the example presented earlier.

```
DEFINE CheckAlarm ( IN AlarmDelay TYPE IN
                      {"REAL"},
                    IN AlarmString TYPE IN
                      {"STRING"}
                  )

DESCRIPTION
  PURPOSE :
    Check if alarm is present after a specified
    delay.

  SIGNATURE :
    CheckAlarm ( <AlarmDelay>, <AlarmString> )
END DESCRIPTION

FORMAT  AutoTest  <- " "
FORMAT  AutoCheck <- " "
       /* At the moment AutoTest and AutoCheck
          are not really format processors. The
          declaration of AutoTest and AutoCheck
          does not specify any format processor
          executable. In this case ATP writes
          the code block immediatly to the
```

```
          output channel, i.e. ATP generates
          AutoTest/AutoCheck code.            */

LOCAL sound

IF "***" == AlarmString [1, 3] THEN
       /* Is it a red alarm?                  */
  sound <- "red"
ELSE
       /* NO ==> yellow alarm                 */
  sound <- "c_yellow"
ENDIF
       /* Very critical alarms are announced as
          red alarms whereas less critical
          alarms are announced as yellow alarms.
          Parallel to the visible colored alarm
          string a corresponding sound is
          audible, i.e. for red alarms a red
          alarm sound is audible and for yellow
          alarms a c_yellow alarm sound is
          audible.                           */

BEGIN [ AutoCheck ]
  ^ Verify begin
  ^    Alarm @(1,AlarmString) within
             (@(1,AlarmDelay),NaN);
  ^    sound is @(1,sound);
  ^ Verify end
END [ AutoCheck ]
       /* AutoCheck code generation.
          The code template includes ATP
          variables, which will be evaluated
          at run time.                       */

END CheckAlarm
```

Discussion

Although the current ATP integration is only the first phase, ATP has proved to be a powerful tool for attacking and solving complex testing problems that otherwise would not have been solved in the same time frame. Like every new tool, at the beginning some effort is required to learn the language. Also, the test engineers have to implement a set of primitive functions to build a powerful function repository. Nevertheless, our experience has shown that productivity increased significantly and that ATP helped to ensure the predictability of product releases.

After a few days of use the test engineers felt comfortable enough to develop their first automatic tests with ATP and were able to use the function repository.

Tests are much more sophisticated and effective than before. The same tests written directly in Autotest/AutoCheck code would have probably required three times more development time without reaching the same degree of reliability, flexibility, and maintainability. The test engineers using ATP used the increased productivity to think about better test designs.

Failures have been found earlier because of higher test coverage, especially from the use of random test data generation. These failures would not have been detected in the validation phase with the existing static test. The risk of missing a failure is therefore reduced by ATP.

The redundancy of the tests is much lower. Test engineers are now able to adapt their test procedures rapidly to changed system behavior. In most cases they just have to update some constants.

The higher abstraction level of the test procedures enables the test engineer to use the same test procedures to test new patient monitor products. The adaptation requires the substitution of some low-level primitive functions and the format processors.

Implementation

The ATP interpreter is implemented in C on a workstation running the HP-UX* operating system. Most of the repository tools have been written in the ATP language itself. This illustrates that ATP is not only a language for formulating test procedures.

The architecture follows the classical compiler architecture. The front end with lexical analysis, syntactical analysis, and semantic analysis is similar to other compilers for high-level formal languages. The back end consists of the code generation module and the communication module, which manages the format processor communication and other functions.

Conclusion

The new ATP language bridges the gap between high-level test design and low-level automatic test procedures. The integration of ATP into the test environment has increased productivity and reduced redundancy. More important, the quality of the testing process has increased with the use of this abstract high-level programming language. Migration of the test procedure set to new products is now much easier because most of the code can be reused.

Reference

1. D. Göring, "An Automated Test Environment for a Medical Patient Monitoring System," ***Hewlett-Packard Journal***, Vol. 42, no. 4, October 1991, pp. 49-52.

Windows is a U.S. registered trademark of Microsoft Corporation.

HP-UX 9.* and 10.* for HP 9000 Series 700 and 800 computers are X/Open Company UNIX 93 branded products.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

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

Structural Testing, Random Testing, and Statistical Structural Testing

Random testing is one of the more common test strategies. It does not assume any knowledge of the system under test, its specifications, or its internal design. This technique is insufficient for validating complex, safety-critical or mission-critical software.

The structural testing approach systematically derives the test procedures from the external and internal specifications. Therefore, the term test design best describes the mental activity behind this method. The structural testing approach divides the input data space into subdomains. The criteria for this partitioning are given by the external specification of the system. Each subdomain is an equivalence class which is tested by choosing some representatives. But what if the subdomain is heterogeneous, has unknown side effects, or includes errors if executed in a particular order? (A heterogeneous subdomain includes both good and bad data points. Good means that the system works as specified, whereas bad data leads to system failure. For example, in the heart rate alarm test described in the accompanying article, the high alarm limit domain may contain data points that, when applied to the patient monitor, produce no high limit alarm. Other data points may behave as expected). Unfortunately, the subdomains are seldom homogenous or disjoint.

Waeselynck & Thévenot-Fosse[1] showed that a statistical component has to be included to provide a sufficient test data set for a subdomain. This approach is known as statistical structural testing. Our experience has shown that this strategy leads to the best results.

Reference

1. H. Waeselynck and P. Thévenot-Fosse, "An Experimentation with Statistical Testing," *Proceedings of the 2nd European International Conference on Software Testing Analysis & Review*, 1994.

An Automated Test Evaluation Tool

The AutoCheck program fully automates the evaluation of test protocol files for medical patient monitors. The AutoCheck output documents that the evaluation has been carried out and presents the results of the evaluation.

by Jörg Schwering

The AutoCheck program extends the automated test environment described in this journal in 1991.[1] It fully automates the evaluation of the test protocol files generated by the AutoTest program.

Fig. 1 is a brief summary of the 1991 system. The main part of the figure shows the main elements of the AutoTest program and its environment. The AutoTest program reads and interprets commands line-by-line out of a test script (a test procedure). Basically there are three types of commands:

- Commands to simulate user input on the monitor under test (keypusher)
- Commands to control the signal simulators, which play the role of a critically ill patient
- Commands to request and log output data from the monitor under test (reactions to keypresses and signals applied).

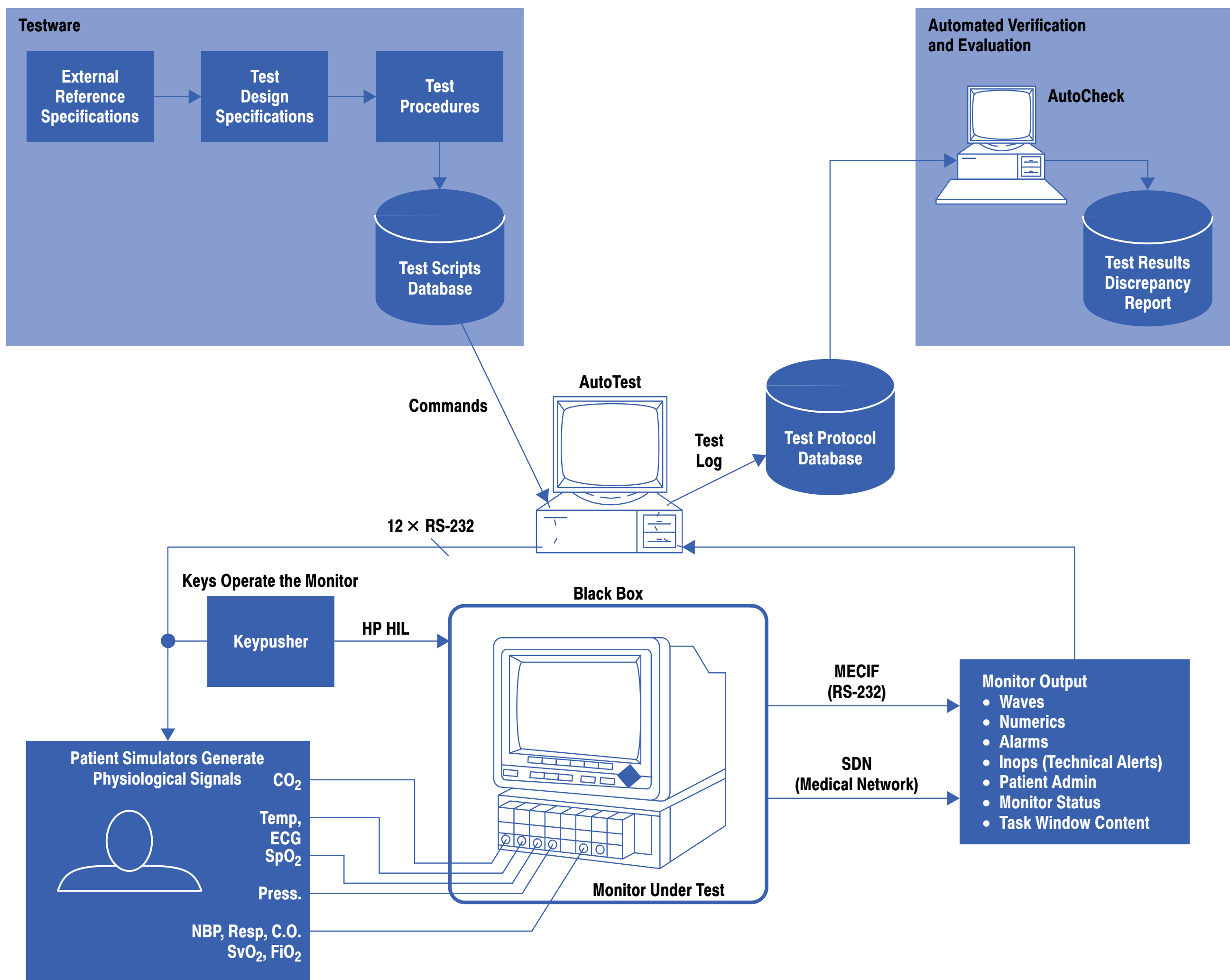


Fig. 1. *Automated test environment for medical patient monitoring systems. AutoCheck, the automatic verification and evaluation tool shown in the upper right corner, is a recent addition.*

The execution of all commands and the data received from the monitor are logged into a protocol file. Fig. 2 is an example of a test protocol file (one test case only).

```
08:57:35 // ------------------------------------------------------------------------
08:57:35 // Test case 6)
08:57:35 // Adjust HR alarm limits to 50-80
08:57:35 Merlin param
08:57:37 Merlin f2
08:57:38 Merlin f7
08:57:39 Merlin f3 -n48
08:58:33 Merlin f6 -n48
08:59:28 Merlin f4 -n7
08:59:36 Merlin f5 -n34
09:00:15 // Set HR to 120
09:00:15 sim1                                  NSB120
09:00:15 // delay
09:00:15 wait 20
09:00:35 // ???????????????????????????????????????????????????????????????
09:00:35 // Verify that:
09:00:35 // HR = 120;
09:00:35 // alarm "HR 120 > 80" ;

09:00:35 // ???????????????????????????????????????????????????????????????
09:00:35 mecif tune HR ECG-CH1 3

Tuned : HR -NU, ECG-CH1 -WS,
@
09:00:36 A "** HR 120 > 80    " O R   33   ECG-CH1 -WS p=---ovY-    40     4
09:00:37 (W1) II       16/            2752   1492    mV         p=---o--- ECG-CH1
09:00:37 HF      120               50    /80      30   /250      p=---ovY- /min HR
ADAT1: "Extreme Brady:  " 0  0   20
@
09:00:37 A "** HR 120 > 80    " O R   33   ECG-CH1 -WS p=---ovY-    40     4
09:00:38 (W1) II       16/            2752   1492    mV         p=---o--- ECG-CH1
09:00:38 HF      120               50    /80      30   /250      p=---ovY- /min HR
ADAT1: "Extreme Brady:  " 0  0   20
@
09:00:38 A "** HR 120 > 80    " O R   33   ECG-CH1 -WS p=---ovY-    40     4
09:00:39 (W1) II       16/            2752   1492    mV         p=---o--- ECG-CH1
09:00:39 HF      120               50    /80      30   /250      p=---ovY- /min HR
ADAT1: "Extreme Brady:  " 0  0   20
09:00:40 EOT
~
09:00:40 // ----------------------------------------------------------------
```

Comment

Time Stamp

Keypusher commands to set alarm limits to 50-80

Simulator command to set heartrate to 120

Additional AutoTest command (wait)

Expected output (for human evaluator)

Command to gather data

Numeric value (heart rate)

Alarm string

Data blocks read

Besides the alarm string and the HR numeric, much related data such as units or the alarm limits is received. Most of these attributes should also be checked.

Fig. 2. *An example of a test protocol file for one test case.*

The upper left block in Fig. 1 indicates how the test scripts are derived from the product specifications. More information on the testing process can be found in ***Article 12***. A test script consists of a startup configuration block, which configures the monitor to a defined startup condition, and the test cases. Each test case consists of three parts:

- The actions (keypusher and simulator commands)
- The description of the expected results
- The AutoTest data request commands.

The upper right corner in Fig. 1 shows the automatic evaluation tool, which is the subject of this article.

Manual Evaluation

In the test environment of Fig. 1, the test engineer had a tool that ran sequences of test scripts in batch mode, completely unattended, overnight or on weekends. With the introduction of AutoTest, the main effort for the test engineer shifted from test execution to test evaluation, which was done with the help of an editor. The protocol files generated by AutoTest (see Fig. 2) are ASCII text files that are very often larger than one megabyte (some large tests have reached a size of more than 100 megabytes).

The evaluation task was not only tedious and time-consuming but also error-prone and dependent on the knowledge, experience, and even the mental alertness of the evaluator. As a consequence, the manual checks were restricted to a minimum for each test case, which typically meant that unexpected attributes were not detected. Furthermore, for tests

that needed to be repeated, the evaluation was normally restricted to a few (one to three) selected repetitions. Statistical tests, such as adjusting an alarm limit randomly and checking the alarm string, generate particularly large protocol files that are difficult to evaluate manually, leading the test engineer to reduce the number of test cases to a minimum.

Goals for an Automatic Test Evaluation Tool

Because of these problems, an investigation was started on a tool that could replace the human evaluator. The goals for this automatic evaluation tool were:

- To relieve the test engineer of tedious, time-consuming manual evaluation and thereby increase efficiency
- To avoid overlooking discrepancies
- To get the test results faster by quicker evaluation
- To increase test coverage through side-effect checks
- To make evaluation more objective (not tester dependent)
- To allow conditional checks (flexibility)
- To automate local language regression tests (see ***Article 15***).

Use Model

The basic use model (see Fig. 3) is the replacement of manual evaluation with the automatic evaluation tool. The evaluation of the protocol file runs after the test has finished. The test files already contain the expected results coded in a formal language readable by AutoCheck.

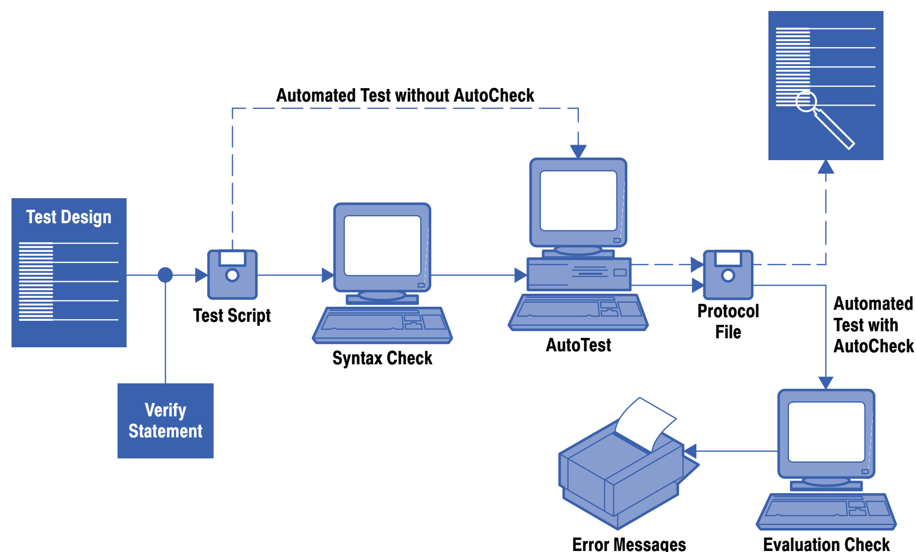


Fig. 3. *The use model for AutoCheck replaces manual evaluation with the automatic evaluation tool.*

Test execution and evaluation now consists of the following steps:

1. Write the AutoTest test script including the expected results in AutoCheck format. The basic test script layout as described above stays the same. The only differences are that some AutoCheck definitions such as tolerances (see "AutoCheck Features and Syntax" below) are added to the startup block and that the description of the expected results has to follow the AutoCheck format.
2. Run this test script through the AutoCheck syntax check to avoid useless AutoTest runs.
3. Execute the test script with AutoTest as usual. The expected results (AutoCheck statements) are treated by AutoTest as comments, which means that they are only copied into the protocol file together with a time stamp.
4. Run the protocol file through the AutoCheck evaluation check, which includes a syntax check. AutoCheck generates a diff file reporting the deviations from the expected results (errors) and warnings for everything that couldn't be evaluated or is suspicious in some other way (for details see "AutoCheck Output" below).
5. If and only if AutoCheck reports errors or warnings, check the protocol file to find out whether the deviation is caused by a flaw in the test script or a bug in the patient monitor under test.

Architecture

We first conducted a feasibility study, which investigated different architectural approaches and implementation tools. The first approach was in the area of artificial intelligence, namely expert systems and language recognition (this would be expected for an investigation started in 1991). It soon became apparent that protocol file evaluation is basically a compiler problem. The languages and tools investigated were Prolog/Lisp, sed/UNIX® shell, lex/yacc, C, and a C-style macro language for a programmable editor. We came to the conclusion that a combination of lex/yacc and C would lead to the easiest and most flexible solution.

Fig. 4 shows the AutoCheck architecture. The protocol file is first run through a preprocessor, which removes all lines irrelevant to AutoCheck, identifies the different AutoTest interfaces, and performs the local language translations. Thereafter it is analyzed by a combination of a scanner and a parser. We implemented specialized scanner/parsers for the AutoCheck metalanguage and the data provided by the different patient monitor interfaces. The AutoCheck statements and the AutoTest data are written into separate data structures. A third data structure holds some control parameters such as the accepted tolerances (see "AutoCheck Features and Syntax" below). After each data package, which is the answer to one AutoTest data request command, the compare function is started. The compare function writes all deviations into the error file.

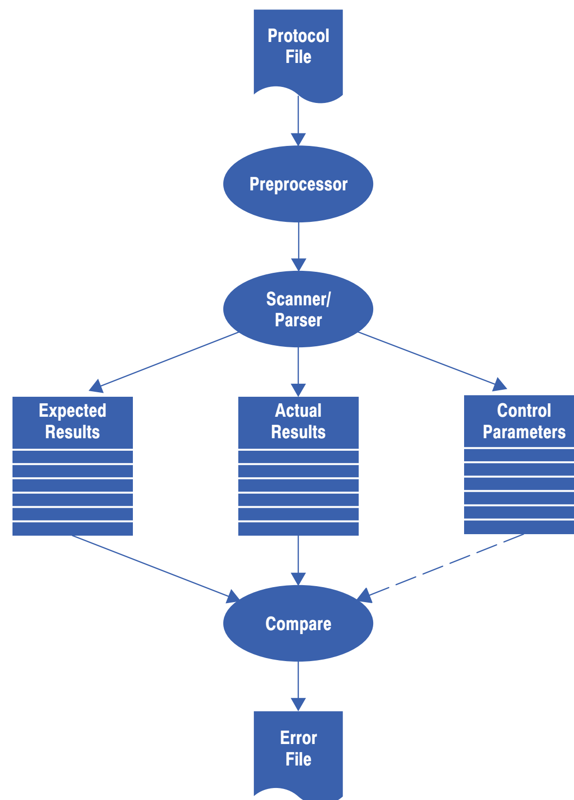


Fig. 4. *AutoCheck architecture.*

Basically, AutoTest and AutoCheck recognize two types of data requests: ***single tunes***, which respond with exactly one data set for each requested message, and ***continuous tunes***, which gather data over a defined time interval.

In the monitor family under test all important data has an update frequency of 1024 ms. AutoTest groups all data messages received within a 1024-ms frame into one data block and AutoCheck treats each data block of a continuous tune like a data package of a single tune.

All AutoCheck statements are then verified against each data block. The AutoCheck statements remain valid and are applied to the next data block until they are explicitly cleared or overwritten by a new AutoCheck block.

AutoCheck Features and Syntax

The AutoCheck features and syntax are best described using the example shown in Fig. 5. The numbers below correspond to the numbers in the left column of Fig. 5.

1. All AutoCheck statements are preceded by a caret (^) and are treated as comments by AutoTest. As mentioned above under "Architecture," AutoCheck statements are grouped into blocks. Each block is enclosed by the two statements Verify Begin and Verify End.

```
(4)     ^ Tolerance Definition
(4)     ^    "Temp1" : 1%;
(4)     ^ End Tolerance Definition

(1)     ^ Verify Begin
(2)(4)  ^    "Temp1"->value = 37.0;
(2)     ^    "Temp1"->unit = C;
(3)     ^    not alarm for "Press1"
(3)     ^      within (5,NaN);
(5)     ^    alarm "HR" > al_max;
(6)(7)  ^    if Neonate
(6)     ^    then
(6)     ^      "HR"-> al_min = 30;
(6)     ^    endif;
(6)(7)  ^    if value of "Pat.Size" is "Adult"
(6)     ^    then
(6)     ^      "HR"-> al_min = 15;
(6)     ^    endif;
(8)     ^    write "Check user input";
(1)     ^ Verify End)
```

Fig. 5. *An example of expected results written in the AutoCheck language. The numbers at the left refer to the paragraphs in the article that describe these statements.*

2. There is a set of AutoCheck statements that enables the user to verify all data that can be read by AutoTest (numerics, alerts, sound, wave data, task window texts, etc.). An example of a numerical value is temperature, including all of its accompanying attributes such as units (°C) and alarm limits. In this example the value of the temperature numeric is expected to be 37.0°C.

3. Verify statements can be combined with:

- A negation, for example to check the absence of an alarm
- Timing conditions, for example to verify that an alarm delay is within its specified range.

   In this example it is expected that in the time interval from 5 seconds to infinity (NaN) there is no alarm for blood pressure. This is a typical test case in which there was an alarm and the simulated measurement has been reset between the alarm limits, the object being to check that the alarm disappears within a defined time.

4. For all numerical values (measurements), including those in the alarm string, a tolerance can be defined to compensate for simulator tolerances. The tolerances are defined outside the Verify block in an additional block. Although the user can change the tolerances as often as desired, they are typically defined once at the beginning of a test procedure and then used for the whole test procedure. In this example, all values in the range from 1% below 37.0°C to 1% above 37.0°C (36.7 to 37.3°C) would be accepted as correct for the Temp1 parameter.

5. There are special combinations, such as a numeric value and an alarm string. For instance, in the monitor family under test an alarm message typically indicates that the alarm limit has been exceeded. The alarm limit is also included in a numeric message along with its attributes. The command alarm "HR" > al_max allows the tester to compare the alarm limit in the alarm message with the alarm limit in the numeric message (as opposed to checking both messages against a fixed limit). This feature is mainly useful for statistical tests.

6. Simple control structures (if, and, or) can be used to define different expected results for conditions that are either not controllable by the test environment or are deliberately not explicitly set to expand the test coverage. In the monitor family under test some settings are dependent on the configuration (e.g., patient size). The simple control structures allow configuration-dependent evaluation

7. As a condition in an if statement, either flags, which have to be defined earlier in the test procedure, or an ordinary AutoCheck statement can be used.

8. AutoCheck provides a command to write a comment into the output file. This can be used to instruct the user to check something manually (e.g., a user input string).

AutoCheck Output

Fig. 6 is an example of AutoCheck output. AutoCheck generates seven different output types:

- Evaluation Error. The expected data and the received data don't match.

- Evaluation Warning. AutoCheck couldn't determine whether the data is correct (e.g., data missing).
- AutoTest Error. Errors reported by AutoTest are mirrored in the output file to make them visible to the test engineer, who only looks at the protocol file in case of reported errors.
- Syntax Error. The interpretation of the AutoCheck syntax failed.
- Syntax Warning. The AutoCheck syntax could be interpreted, but is suspected to be incomplete or wrong.
- Data Error. The AutoTest data couldn't be interpreted correctly. This indicates either a corrupted protocol file or an incompatibility of AutoCheck and AutoTest versions.
- Write. This is a user-defined output. It enables the user to mark data that should be checked manually, such as user input at a pause statement.

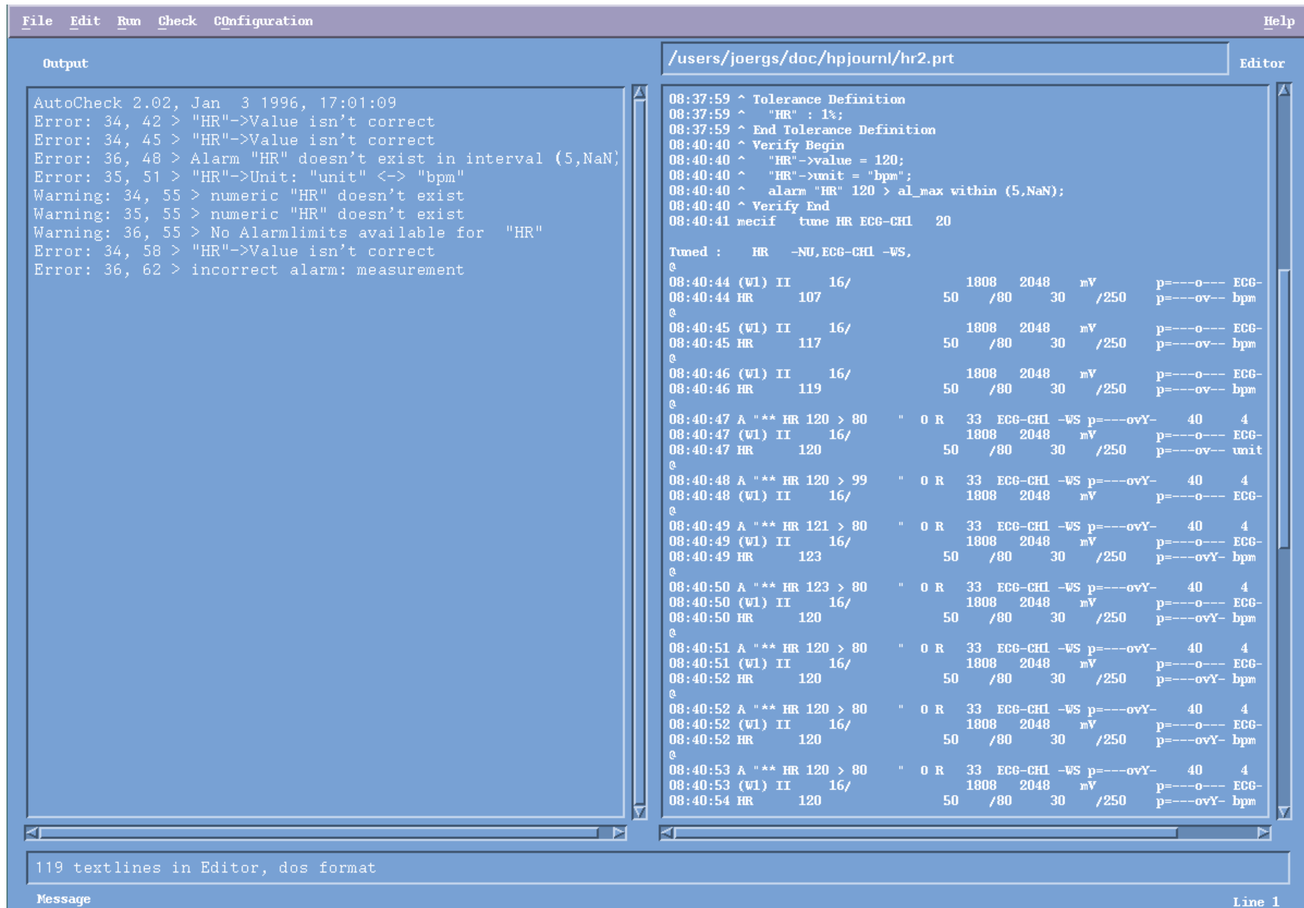


***Fig. 6.** An example of AutoCheck output.*

The user can choose between four different warning levels for the syntax and evaluation warnings and can switch individual warnings on or off.

The output generated by AutoCheck has the following format:

```
ErrorType: statementline, dataline >
  descriptive_text
```

Thus, both the line containing the AutoCheck statement and the line containing the data are indicated.

If the output is written into a file, each line is preceded by the filename(statementline). This is the same format as used by many compilers, and therefore the built-in macros of many programming editors can be used in combination with AutoCheck. This means that errors can be reviewed in much the same way that a source file is debugged after compilation using an editor pointing to the source code errors.

At the end of the evaluation, AutoCheck gives the test engineer a quick overview of the result by providing a table showing how many output messages of each type have been generated. Whereas the evaluation errors indicate bugs either in the product or in the test script, the other output messages indicate potential problems in the test execution or evaluation process.

The AutoCheck output documents both that the evaluation has been carried out and the result of the evaluation, which for medical products are important for regulatory approvals and audits.

Platforms

AutoCheck and Autotest run on different platforms. AutoTest runs on a DOS®-based PC, which is very appropriate as a test controller because of the inexpensive hardware, an operating system that doesn't require dealing with tasking conflicts, and the availability of interface cards (the interface to the medical network is available as a PC card only). AutoCheck runs on a UNIX-based workstation because of the availability of lex/yacc and the greater resources (memory and processing power). However, both tools work on the same file system (a UNIX-based LAN server). The user doesn't have to worry about the different file formats, because AutoCheck automatically takes care of the format conversions. It accepts both DOS and UNIX formats and generates the output according to the detected protocol file format. Having different machines for execution and evaluation has also not proved to be a disadvantage for the test engineer.

Expandability

The basic architecture of AutoCheck has proven to be flexible for enhancements over time. Since the first release of AutoCheck we have implemented many enhancements because of new product features and because AutoTest provides additional data structures.

Validation

The risk of the AutoCheck approach is that, if AutoCheck overlooks an error (false negative output), the tester won't find the error. An automatic evaluation tool is only useful if the tester can rely on it, since otherwise, even if no errors were reported, the tester would still have to look at the protocol file. Therefore, the validation of an automatic evaluation tool is crucial to the success of such a tool. For this reason a thorough test of the tool was designed and every new revision is regression tested. Changes and enhancements undergo a formal process similar to that used for customer products.

Results

The manual evaluation time for an overnight test of around one to two hours has been reduced by the use of AutoCheck to less than a minute. This means that the additional effort for the test engineer for writing the expected results in the AutoCheck syntax is compensated after three to five test runs. This depends on the experience of the test engineer with AutoCheck (the normal learning curve) and the nature of the test.

A positive side effect is that it is much easier for another test engineer to evaluate the test.

AutoCheck also leads to bigger tests with an increased number of checks for each test case, such as checks for side effects. Such an automatic evaluation tool is also a prerequisite for statistical testing. It would take too much time to evaluate all these test cases manually. In other words, AutoCheck leads to higher test coverage with lower effort for the test engineer.

Once relieved of a great deal of the more mechanical test execution and evaluation activities, the test engineer has time to work on new and better test approaches or possibilities for an increased automation level. Over time this has led to enhancements of both AutoTest and AutoCheck and to new tools like ATP (see ***Article 13***).

Acknowledgments

I wish to thank Pascal Lorang for the creation of the prototype and all students involved in the development of AutoCheck.

Reference

1. D. Göring, "An Automated Test Environment for a Medical Patient Monitoring System," ***Hewlett-Packard Journal***, Vol. 42, no. 4, October 1991, pp. 49-52.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

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

MS-DOS is a U.S. registered trademark of Microsoft Corporation.

Effective Testing of Localized Software

Testing localized software is a complex and time-consuming task. With the help of the testing tools developed for HP patient monitors, local language validation for these products is fully automated.

by Evangelos Nikolaropoulos, Jörg Schwering, and Andreas Pirrung

Localization plays a very important role in the successful marketing of software all over the world. For medical devices there are legal requirements to provide instruments and accompanying documentation in the language of the healthcare personnel who use them (as is the case in the European Union). It is often forgotten that localized software is ***different*** software from the original (most probably in English) that was used for system integration and final validation. Localized software undergoes a proper integration cycle (integration of software and translated strings) and must be validated separately. The complexity of this validation is obvious if one considers the efforts required to check all error conditions and the corresponding error messages (and to understand them) for software in every language where the product is marketed.

The most common errors in localized software, assuming that the translation is done by a professional translator for this language and is correct, are:

- Missing strings (empty messages, parts of screen text missing, menu selection items missing)
- Strings with wrong attributes (maximum length exceeded— a possible crash cause) or strange characters filling up the remainder of a field
- Wrong strings (not reflecting the intentions of the author for this particular context)
- Various misspellings or violations of grammar rules applied to the language produced through the combination of translated strings by the software
- Strings not properly cleared in a text field before a new string is displayed.

Local language testing in our laboratory is composed of two steps: the verification of the translation and the validation (regression testing) of the localized software.

To verify the translation, a translator goes over all possible screens, messages, help texts, printouts, and so on to check for translation errors. The difficulty here is that in most cases the translator is not a frequent user of the device under test, and needs assistance in operating the medical instrument and generating all possible string combinations.

The aim of validation (regression testing) of the localized software is to prove that the localization has not negatively affected the functionality and performance of the instrument. Additional attention must be paid to typical localization errors (overflows or garbage generation).

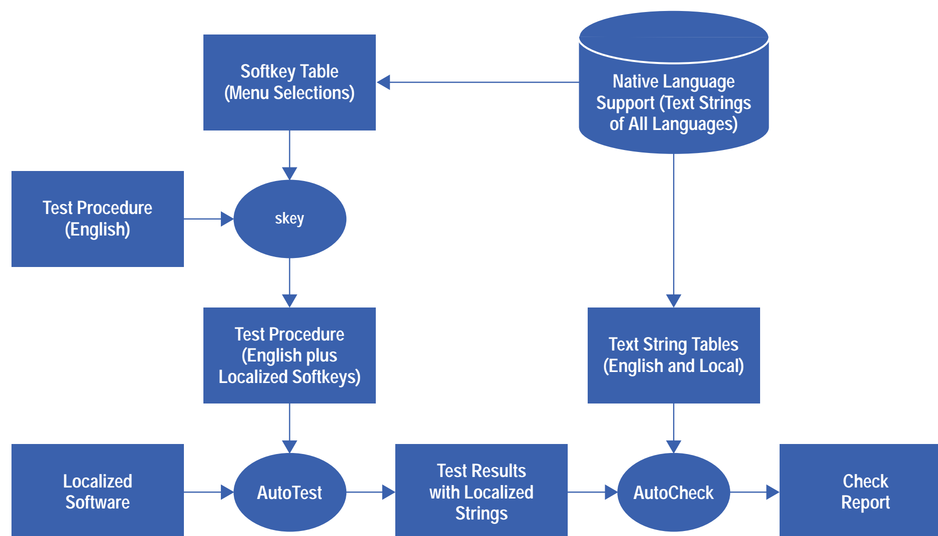


Fig 1. *Local language validation process for patient monitors.*

(a) Extract of a Test Procedure for Blood Pressure as it Is Used for Tests of English Software

```
                    .....
* alarm suspended -> alarms not suspended
merlin mainscrn
mecif skey -kalarmvol
merlin "SwitchOnAlarms"
* =====================================================
^ Verify begin
^  INOP "NBP EQUIP MALF" ;
^  Sound is hard inop ;
^ Verify end
* =====================================================
                    .....
* =====================================================
^ Verify begin
^  twprompt is "Problems with the pneumatic are detected" ;
^ Verify end
* =====================================================
                    .....
```

Press a hardkey
Make a selection from a menu
Verify statement (AutoCheck)
Verify statement (AutoCheck)

(b) The Procedure after Translation of Softkeys (Menu Selections) to Finnish

```
                    .....
* alarm suspended -> alarms not suspended
merlin mainscrn
mecif skey -kalarmvol
merlin "HälytPäälle"
* =====================================================
^ Verify begin
^  INOP "NBP EQUIP MALF" ;
^  Sound is hard inop ;
^ Verify end
* =====================================================
                    .....
* =====================================================
^ Verify begin
^  twprompt is "Problems with the pneumatic are detected" ;
^ Verify end
* =====================================================
```

Press a hardkey (no translation)
Make a selection from a menu (translated)
Verify statement (AutoCheck)
Verify statement (AutoCheck)

Fig 2. *Example of the steps of the local language test process.*

(c) Protocol File after Test Run with Software in Finnish

```
                .....
23:35:14 * alarm suspended -> alarms not suspended
23:35:14 merlin mainscrn
23:35:17 mecif skey -kalarmvol
23:35:21 merlin "HälytPäälle"
                .....
23:36:11 * ======================================================
23:36:11 ^ Verify begin
23:36:11 ^  INOP "NBP EQUIP MALF" ;
23:36:11 ^  Sound is hard inop ;
23:36:11 ^ Verify end
23:36:11 * ======================================================
Filter: NBP -NU, @
23:36:13 (hard inop sound ARec: CS)
23:36:13 I "NBP LAITEVIRHE " O - 1 NBP -NU p=---o--H
                .....
23:36:23 * ======================================================
23:36:23 ^ Verify begin
23:36:23 ^  twprompt is "Problems with the pneumatic are detected" ;
23:36:23 ^ Verify end
23:36:23 * ====================================================== 0
Tuned : TWPROMPT,
23:36:26 F " NBP " 8, 15
23:36:26 P "NBP last calibration done 16 KES 94 15:32 " W 3 T @
23:36:29 P "Pneumatiikassa on havaittu ongelmia " W 3 T
23:36:31 EOT
                .....
```

String translated

Combined string only partly translated

String translated

Fig 2. *(Cont.)*

Automated Local Language Validation

With the help of the testing tools developed for our patient monitors (see Fig. 1 and the accompanying articles in this issue), local language validation is a fully automated process:

1. Translation tables called ***local language tables*** are prepared from the native language support database containing all the English strings and their corresponding translations.
2. A test package, a subset of the test procedures designed for the regression testing of the original English version, is compiled.
3. A copy of each test procedure out of this package is translated into the local language by a tool developed by software quality engineering called skey. The intention here is to replace in the test procedures the selection menu items (softkeys) with the corresponding localized terms. Of course, a test can be executed by passing the position of a selection item (e.g., "press the second selection on the third menu") to the test execution tool, but this approach has proved to be ineffective. The issue here is not to test that the "second selection of the third menu" works, because this was already tested in English, but to prove that the "second selection of the third menu" is translated correctly, and if selected, produces exactly the same behavior as its English counterpart. By calling the selections by their values and not by their positions we achieve higher test coverage and we test functionality and translation at the same time. Another argument for this approach is that the "second selection on the third menu" may be configuration dependent (even local configuration dependent) and therefore not accessible by a position dependent test (e.g., it is still on the third menu but in the sixth place). Thus, calls by value make test procedures more robust.
4. The translated test procedure is passed to AutoTest[1] and is run on the localized software. The results are saved in protocol files containing English verify statements (the expected results), localized softkeys (selections), and localized actual results (see the example in Fig. 2).
5. The protocol files are submitted to AutoCheck (see ***Article 14***). First, the AutoCheck preprocessor takes over the task of translating the verify statements. It uses the local language tables to replace the English text in the verify clauses with the localized text. On the second pass these translated expected results are compared with the localized actual results. Discrepancies are reported in the normal way but with localized content.

A special solution is also provided for Asian languages (simplified and traditional Chinese and Japanese), which use 16-bit codes. For these languages the hexadecimal equivalent for each character is used in the test procedures instead of the "drawn" character. This enables us to keep such characters in ASCII files (like the test procedures and the protocol files) and use them with the test execution and evaluation tools.

The automated local language validation has dramatically improved the process of localized software release. It has reduced the effort for local language testing for a new patient monitor release from twelve to four weeks and has significantly increased the test coverage compared with the traditional manual testing approach.

Acknowledgments

The authors wish to thank Gerhard Tivig for his support during the development of the local language test tools.

Reference

1. D. Göring, "An Automated test Environment for a Medical Patient Monitoring System," *Hewlett-Packard Journal*, Vol. 42, no. 4, October 1991, pp. 49-52.