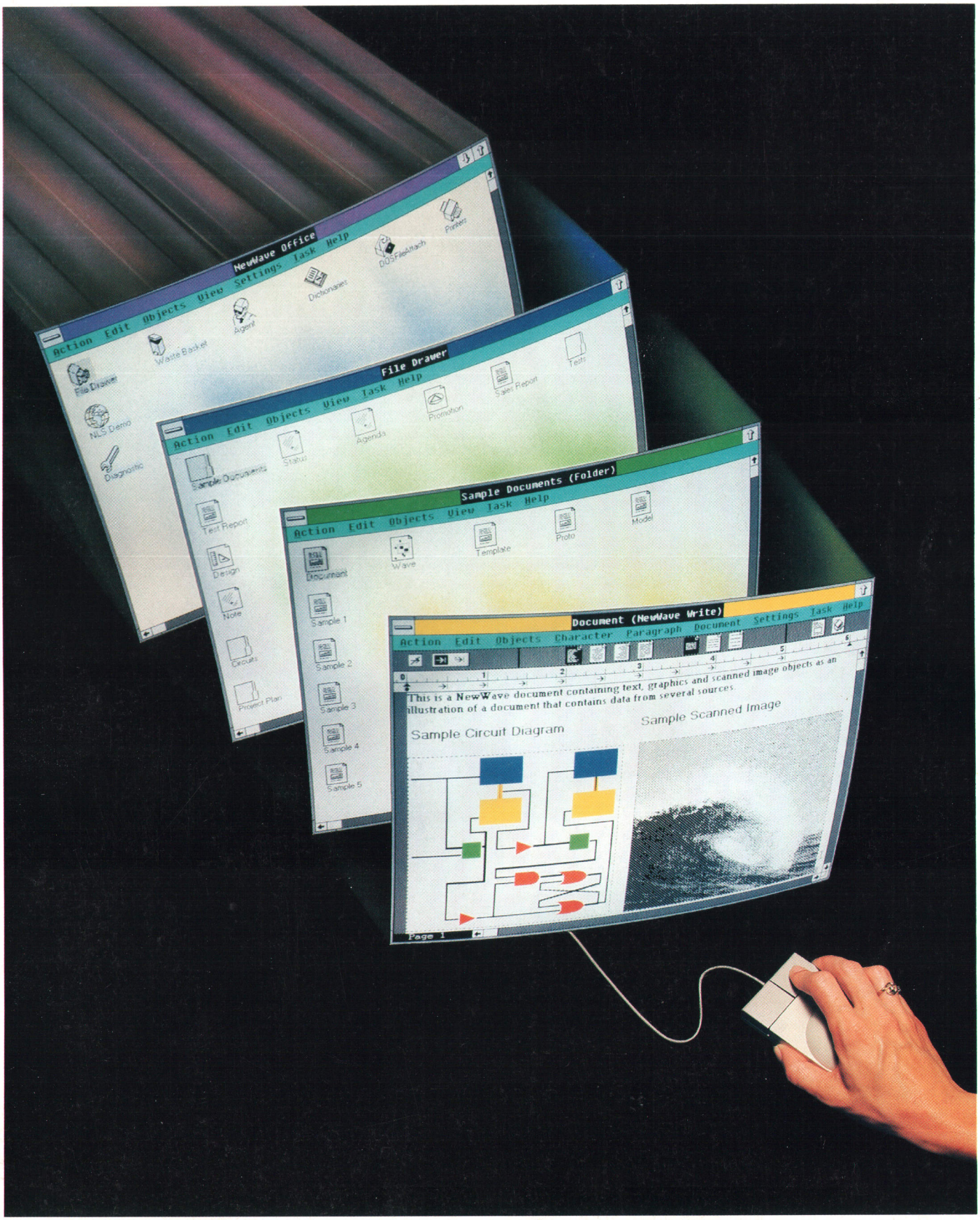
HEWLETT-PACKARD
JOURNAL

AUGUST 1989



HEWLETT
PACKARD

HEWLETT-PACKARD JOURNAL

August 1989 Volume 40 • Number 4

Articles

6 **An Overview of the HP NewWave Environment,** *by Ian J. Fuller*

9 **An Object-Based User Interface for the HP NewWave Environment,** *by Peter S. Showman*

17 **The NewWave Object Management Facility,** *by John A. Dysart*

23 **The NewWave Office,** *by Beatrice Lam, Scott A. Hanson, and Anthony J. Day*

32 **Agents and the HP NewWave Application Program Interface,** *by Glenn R. Stearns*

35 **AI Principles in the Design of the NewWave Agent and API**

38 **An Extensible Agent Task Language,** *by Barbara B. Packard and Charles H. Whelan*

40 **A NewWave Task Language Example**

43 **The HP NewWave Environment Help Facility,** *by Vicky Spilman and Eugene J. Wong*

48 **NewWave Computer-Based Training Development Facility,** *by Lawrence A. Lynch-Freshner, R. Thomas Watson, Brian B. Egan, and John J. Jencek*

57 **Encapsulation of Applications in the NewWave Environment,** *by William M. Crow*

67 **Mechanical Design of a New Quarter-Inch Cartridge Tape Drive,** *by Andrew D. Topham*

74 **Reliability Assessment of a Quarter-Inch Cartridge Tape Drive,** *by David Gills*

82 **Use of Structured Methods for Real-Time Peripheral Firmware,** *by Paul F. Bartlett, Paul F. Robinson, Tracey A. Hains, and Mark J. Simms*

87 **Product Development Using Object-Oriented Software Technology,** *by Thomas F. Kraemer*

95 **Objective-C Coding Example**
98 **Object-Oriented Life Cycles**

Departments

4 **In this Issue**
5 **Cover**
5 **What's Ahead**
31 **Correction**
31 **Trademark Acknowledgments**
64 **Authors**

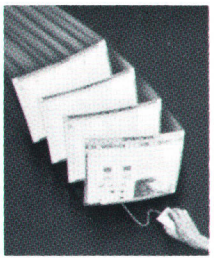
The **Hewlett-Packard Journal** is published bimonthly by the Hewlett-Packard Company to recognize technical contributions made by Hewlett-Packard (HP) personnel. While the information found in this publication is believed to be accurate, the Hewlett-Packard Company makes no warranties, express or implied, as to the accuracy or reliability of such information. The Hewlett-Packard Company disclaims all warranties of merchantability and fitness for a particular purpose and all obligations and liabilities for damages, including but not limited to indirect, special, or consequential damages, attorney's and expert's fees, and court costs, arising out of or in connection with this publication.

Subscriptions: The Hewlett-Packard Journal is distributed free of charge to HP research, design, and manufacturing engineering personnel, as well as to qualified non-HP individuals, libraries, and educational institutions. Please address subscription or change of address requests on printed letterhead (or include a business card) to the HP address on the back cover that is closest to you. When submitting a change of address, please include your zip or postal code and a copy of your old label.

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

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

In this Issue



Can the majority of people ever be as comfortable using a personal computer as they are driving a car? Most people would say that the computer industry has a long way to go to achieve this goal. However, the subject of the first nine articles in this issue—the HP NewWave environment—is a truly giant step in that direction. Instead of seeing the computer as a pile of hardware supporting multiple application programs that can be used to accomplish complex tasks if the user is knowledgeable enough, the NewWave user sees the computer as a single tool able to perform multiple complex tasks. The NewWave user interface is modeled on an ordinary office, offering a filing cabinet, a wastebasket, a printer, and other common functions, along with an uncommon staff member, a software robot called an "agent," which acts as a personal assistant for the user. In the NewWave Office, icons representing tasks performed by application programs are displayed along with icons representing basic office functions. Supporting this user interface but unseen by the user is an advanced architecture that specifies how applications are designed for the NewWave environment and how they are managed within it. Applications, tasks, and basic functions are treated as objects, and a transparent but powerful object management facility orchestrates the objects to accomplish what the user wants, switching between applications as necessary. The agent acts somewhat like the macro facilities offered by some applications, but it is much more powerful because it can use many applications and data sources, interacting with the object management facility through a NewWave architectural component called the application program interface. Another NewWave component is an advanced help facility. When the user wants help, it isn't necessary to determine which application to ask; the user simply asks the computer. An overview of the HP NewWave environment is presented in the article on page 6. For the conceptual and object models behind the object-based user interface, see page 9. Concepts and features of the object management facility and the NewWave object-based file system can be found in the article on page 17. For the design of the NewWave Office, see page 23. The technical details of the agent, the application program interface, and the agent command language can be found in the articles on pages 32 and 38, and the help facility is described in the article on page 43.

For the NewWave environment to provide the hoped-for benefits to the user, application programs must be designed to work within it. Existing applications, particularly those written for the Microsoft Windows environment upon which the NewWave environment is based, can be "encapsulated" and gain some of the NewWave benefits, but not all. How encapsulation works is detailed in the article on page 57. NewWave designers have also addressed the question of how the NewWave architecture can aid in the development and delivery of computer-based training. This is the subject of the article on page 48.

The NewWave environment is an example of object-oriented software. Object-oriented software technology, which includes both programming languages and development methods, is becoming more widely accepted for software development. It has proved to be both productive and powerful, and it offers useful new approaches to the problems of code reuse and software maintainability. In the paper on page 87, Tom Kraemer gives us an introduction to object-oriented software technology and an example of its use in the development of the HP VISTA software for the HP 3565S Signal Processing System.

We first encountered the design story of the HP 9145A Quarter-Inch Cartridge Tape Drive in a paper presented at the 1988 HP Software Engineering Productivity Conference. The paper described the use of structured software design methods for the tape drive's real-time firmware. The article on page 79 is based on that paper. The hardware side of the design story is told in the article on page 67. Achieving the objectives—doubling the speed and cartridge capacity of an existing product—required a new tape cartridge with new media, tighter component and assembly tolerances, and new manufacturing processes. An extensive reliability assessment program verified the new design and continues to ensure the drive's reliability (page 74).

R.P. Dolan
Editor

Cover

The cover shows the displays that would appear when a NewWave Office user opens a file drawer, selects a folder, and chooses a document to edit—operations performed in a typical office environment. The example shows a complex document, that is, it contains text and graphics. The NewWave user does not have to select an application and then the document, but only the document. The NewWave environment handles associating the document with the application.

What's Ahead

The October 1989 issue marks the *Hewlett-Packard Journal*'s fortieth anniversary. The first issue was in September 1949. This is also HP's fiftieth anniversary year. To celebrate this double anniversary, we'll have a special article recapping some of the highlights of those years for both the company and the publication. We'll also present six papers from the 1988 HP Technical Women's Conference, the first conference of its kind at HP. The papers address a variety of hardware, software, and process management topics. Eight articles will cover the hardware and firmware design of the Performance Signal Generator family, consisting of the HP 8644A, 8645A, and 8665A Synthesized Signal Generators. A distinguishing feature of these instruments is a single phase-locked loop design, in contrast to the multiple-loop designs formerly used.

An Overview of the HP NewWave Environment

The NewWave environment allows users to concentrate on the task and not the computer system. For developers of new applications, it provides the facilities to integrate applications into the the NewWave environment.

by Ian J. Fuller

THE NEWWAVE ENVIRONMENT is a comprehensive system developed by HP to provide a new level of flexibility and ease of use in our business systems. This article presents the history, the motivation, and an overview of the features and major components of the NewWave environment. The other NewWave articles in this issue discuss the components of the NewWave architecture in detail.

To understand the NewWave environment it is helpful to review the background for HP's decision to invest in this product. HP has been producing office systems software since the late 1970s. Some early products include the Design Systems Graphics package (DSG) and the HP Word word processing package, both of which run on HP 3000 Computers. In personal computer software, the CP/M®-based HP 125 Business Assistant, which was released in 1982, had a number of business software solutions, including word processing, a spreadsheet application, and a presentation graphics package.

The release of HP DeskManager for HP 3000 Computers in 1982 marked HP's first product that addressed the need for integrating these office systems software products. HP DeskManager was originally designed as a simple electronic mail product,[1] allowing users to send and receive messages easily across a network of HP 3000 Computers. It was very clear that our customers needed more than electronic mail functions from HP DeskManager. They needed to be able to compose arbitrary packages of information into messages and to file those messages for later retrieval. The introduction of new software on the HP 3000 and on our personal computers meant that we needed to integrate the products mentioned above with HP DeskManager. At the very least, users should be able to create and read an HP Word document within the HP Deskmanager environment without leaving the product. Over the years HP has invested very heavily in HP DeskManager, evolving it into a true integrated office system. It has a wide range of features and acts as the center of our HP 3000-based office solutions with close links to the personal computer.

To enable users to connect their personal computers to the HP 3000, HP developed the Personal Productivity Center range of products. For the HP 3000 Computer these products include HP DeskManager, HP File/Library, HP Schedule, and Resource Sharing. For PC users these products include software packages for electronic mail, terminal emulation, word processing, data management, spreadsheets, and graphics.

There are a number of architectural limitations in the current products that prevented us from creating the truly integrated and powerful system our customers need. The products that we have available were designed in different HP divisions and by outside companies to meet the needs of their particular markets. This resulted in software applications that did not have the same basic architecture.

An investigation of potential solutions to the problem of integrating our products began with the knowledge that the solution developed must have the following fundamental characteristics:

- It must be practical. The solution must work well on existing hardware and add value to existing software so that customer investments are protected.
- It must be flexible and extendable. The software field is expanding rapidly and opportunities will exist tomorrow that could not have been dreamed of when the system was designed. This includes portability across different hardware platforms.
- It must allow for the efficient development of systems. Software components must be available for reuse according to the maxim that if a function is used several times in a system it should be provided as a system service for use by all components.

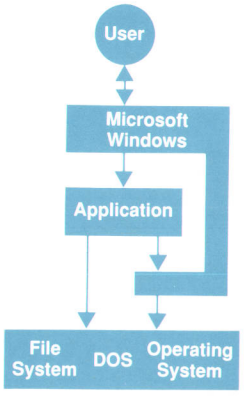


Fig. 1. *Microsoft Windows environment. MS Windows enables users to isolate applications from the details of the hardware and provides a graphical user interface and multitasking under the MS-DOS operating system.*

- It must have a sufficient advantage over existing systems that customers and software developers will make the investment necessary to use the new system.
- It must allow us to build easy-to-learn and easy-to-use software. Users do not want to spend weeks or months learning to use software, and once they are proficient with the software, they do not want to be hampered by an inflexible system.
- It must be an open system. HP could not hope to offer proprietary solutions in all areas. Customers would want to take advantage of our architecture. Designing an open system from the start would allow many more solutions than we alone could provide.
- It must run on existing hardware such as the HP Vectra and the IBM PC/AT.

This is a formidable list of objectives. The NewWave environment is designed to meet these objectives, and exceed them wherever possible.

Open and Extendable Architecture

The NewWave environment was originally designed as a proprietary environment for the development of better integrated office systems. However, as prototypes of the system became available and were demonstrated, it became clear that the system had great appeal to a wider audience than the internal HP community. Major accounts and independent software vendors (ISVs) all wanted to learn how to write NewWave applications. They wanted to take advantage of our object-based file system and applications, while addressing areas of the market that were their specialty.

We decided to open up the NewWave architecture. This entailed:

- Ensuring that our interfaces were made general and robust
- Providing documentation, training, and support suitable for software developers
- Doing extensive testing of the programmatic interfaces in different software and hardware environments.

The NewWave environment will be used by a wide variety of customers, all with different needs, and we know that our customers developing applications to run in the NewWave environment will expect us to keep up with new technologies.

Based on the Microsoft® Windows interface (see Fig. 1), the NewWave environment is well positioned for IBM's OS/2 operating system and the Presentation Manager system from IBM and Microsoft. We are also working on implementing the NewWave environment under the HP-UX operating system. Thus, the NewWave environment provides software developers with the advantage of a system that is portable across a range of platforms.

The NewWave environment is extendable to incorporate new software techniques. For example, the agent facility, which is a major component in the NewWave architecture, is designed to include the functionality and power of artificial intelligence. Future applications, such as natural language systems, can be integrated with NewWave applications that use the agent facility without any impact on the application.

The System Approach

From the beginning the NewWave environment has been designed as an integrated software system. This is in contrast to previous systems, which were designed piecemeal, with integration added later. The major components of the NewWave architecture are shown in Fig. 2. Taken together, these components offer software developers and users a number of benefits, including:

- Consistency. NewWave applications have a consistent user interface, based upon Microsoft Windows. HP provides user interface rules that are an extension of those provided by Microsoft to ensure that NewWave applications have a common look and feel. Common data formats adopted by NewWave applications ensure that information generated in one area can be understood in another, even if it is on a different machine in another location.
- Automation and Training. The agent facility provides task automation across all applications in the NewWave environment. The agent can be thought of as a personal assistant that can perform tasks on behalf of the user. Computer-based training (CBT) is implemented in the NewWave environment using the facilities provided by the agent. The NewWave environment provides the tools for developers to use the agent and to build CBT into

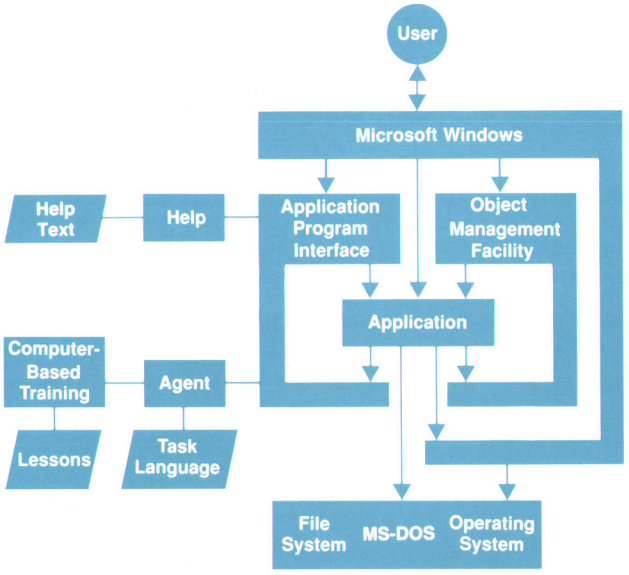


Fig. 2. *The NewWave environment. The NewWave environment is built on the industry-standard IBM PC/AT compatible platform. It uses MS Windows to obtain a graphical user interface consistent with the OS/2 operating system and the forthcoming Presentation Manager system from IBM and Microsoft. The object management facility (OMF) provides the object-oriented capability of the NewWave environment, and allows any kind of data to be treated as an object and represented on the screen as an icon (e.g., text, graphics, spreadsheet, image, voice, etc.). OMF also provides application and data binding, information links, and instant integration. The application program interface (API) provides a set of systemwide services for applications in the NewWave environment. These services include task automation, context sensitive help, and computer-based training (CBT).*

their products. The agent is described in the articles: "An Extensible Agent Task Language" and "Agents and the HP NewWave Application Program Interface" on pages 38 and 32 respectively, and computer-based training is discussed in the article "NewWave Computer-Based Training Development Facility" on page 48.

- Developer Productivity. The NewWave environment handles many of the tasks that developers would have to design and implement individually if they were writing applications outside of the NewWave environment. An example of this is the NewWave help facility. NewWave help is available to all NewWave applications and provides a powerful context sensitive help capability in all areas of the system. The help facility is described in the article "The HP NewWave Environment Help Facility" on page 43.
- Integration. The NewWave object management facility (OMF) provides standardized data object management for all NewWave applications. The OMF includes facilities to define and install specific classes of objects, such as text, graphics, image, or voice, and the links between them. For example, a text object can incorporate an arbitrary number of other objects to produce a compound document. The OMF manages the links so that the object can be manipulated as a whole when it is copied, deleted, or mailed over the network. The OMF also supports data-passing links so that, for example, a spreadsheet object can obtain data from a data base and in turn link that data into a document or a line chart. These facilities are all supported at the system level and thus are available to all NewWave applications. The OMF is described in the article "The NewWave Object Management Facility" on page 17.
- Ease of Learning. A major objective of the NewWave environment is to ensure that if users learn a technique in one area, they will be able to apply the same techniques in another area, even if the application is not supplied by HP. The computer-based training tools are available to all developers so that they can build these powerful learning aids into their products.
- Ease of Use. The NewWave environment is designed to be substantially easier to use than previous systems. We selected an interface based upon direct manipulation of icons that represent NewWave objects (e.g., file drawers, folders, printers, etc.). From the user's perspective the NewWave Office window provides an easy-to-use facility for organizing, filing, retrieving, and deleting objects as necessary. The NewWave Office is the primary interface between the user and the NewWave environment, and it is the first display seen by the user when the environment is loaded. The NewWave Office allows users to focus on the tasks that they need to perform rather than the tools they use. See the article "The NewWave Office" on page 23 for more details.
- Integration with MS-DOS Applications. The NewWave environment offers a set of facilities that enable MS-DOS applications to be used within the system. At the simplest level users can access non-NewWave applications from the NewWave Office window and return to the NewWave Office when they have completed their task. The data from non-NewWave programs can also appear as objects in the OMF. HP provides facilities to encapsulate non-NewWave applications within a shell that allows them to share some of the NewWave environment advantages such as data linking and background printing. Integration with MS-DOS applications is described in the article "Encapsulation of Applications in the NewWave Environment" on page 57.

Supported Hardware Platforms

The initial release of the HP NewWave environment is designed to run on any Intel 80286 or 80386-based personal computer that supports Microsoft Windows. The primary platform is the HP Vectra range of personal computers. However, the NewWave environment is also supported on the IBM PC/AT and IBM OS/2 series and HP Vectra PC compatibles. The NewWave environment supports all hardware peripherals supported by Microsoft Windows, including HP LaserJet printers, HP plotters, and the HP ScanJet image scanner.

HP has developed an expanded memory card[2] for the HP Vectra ES Personal Computer that delivers substantial performance improvements without compromising the industry-standard compatibility of the software using it. One of the objectives of this card is to enhance the performance of applications using Microsoft Windows, such as the NewWave environment.

The NewWave environment connects to networks using the HP AdvanceNet data communications software and hardware. Users have the choice of serial or HP ThinLAN or StarLAN connections to a local area network that may include other personal computers and an HP 3000 Computer system. Software developers can build distributed applications using the networking tools that HP provides, such as NetIPC interprocess communication software, the Cooperative Services library, and Resource Sharing.

Conclusion

The HP NewWave environment is an important step for HP in its strategy of giving computer users a powerful software environment that allows them easier access to information wherever it is stored on the computer network. For the first time we are able to produce a system built on a common architecture, rather than a collection of individual products. The NewWave environment that is available today is the beginning of a trend in software system design that will eventually give the benefits of NewWave to the entire spectrum of information workers.

Reference

1. I. J. Fuller, "Electronic Mail for the Interactive Office," *Hewlett Packard Journal*, Vol. 34, no. 2, February 1983.

2. G.W. Lum, et al, "Expanded Memory for the HP Vectra ES Personal Computer," *Hewlett-Packard Journal*, December 1988, pp. 57-63.

An Object-Based User Interface for the HP NewWave Environment

The NewWave environment is designed to allow users to focus on their tasks and not the tools. To accomplish this, the NewWave environment presents users with a conceptual model based on an office metaphor that is built on an object-based architecture.

by Peter S. Showman

A KEY ELEMENT OF THE HP NEWWAVE ENVIRONMENT is the combination of a system conceptual model, which defines the user's perception of how the system works, and an object model, which defines the architecture of the system. This article describes the NewWave conceptual model and object model by presenting examples based on an office metaphor.

Matching Computers to People

As the use of personal computers in business has become more and more widespread, two conflicting trends have emerged. The complexity of the systems and applications has increased dramatically. Computers are being applied to more complex and more strongly interrelated tasks, often requiring the use of several interacting application programs and data from several sources. At the same time, the need for simple operation is critical. The typical PC user is no longer a computer hobbyist. Most users are busy enough maintaining expertise in their own areas without having to be computer experts as well. Software must be easy to learn and easy to use for first-time or occasional users without sacrificing flexibility and effectiveness for more experienced users. The conflict between the need for simple operation and the increasing functional complexity leads not only to less user satisfaction, but also to decreased productivity and increased training costs.

A simple and consistent user interface has been a long-standing goal of many HP products, including such commercial and office products as the HP Touchscreen Computer,[1] HP DeskManager,[2] and the HP 250 Computer,[3] along with numerous earlier systems (e.g., reference 4). Application of human factors principles and ever-improving hardware have allowed designers to build interactive and responsive application programs. Such software keeps the user informed about the state of the software and the data, and reduces the prior knowledge required to operate it. Techniques that were once seen only on expensive workstations are now commonplace in personal computers. However, in spite of clear progress, a number of obstacles remain for the users.

Where We Are Today

Today's office automation software is almost exclusively function-oriented. Most of the user's work is done in terms of the functions incorporated in the application programs the system provides. The first decision the user must make

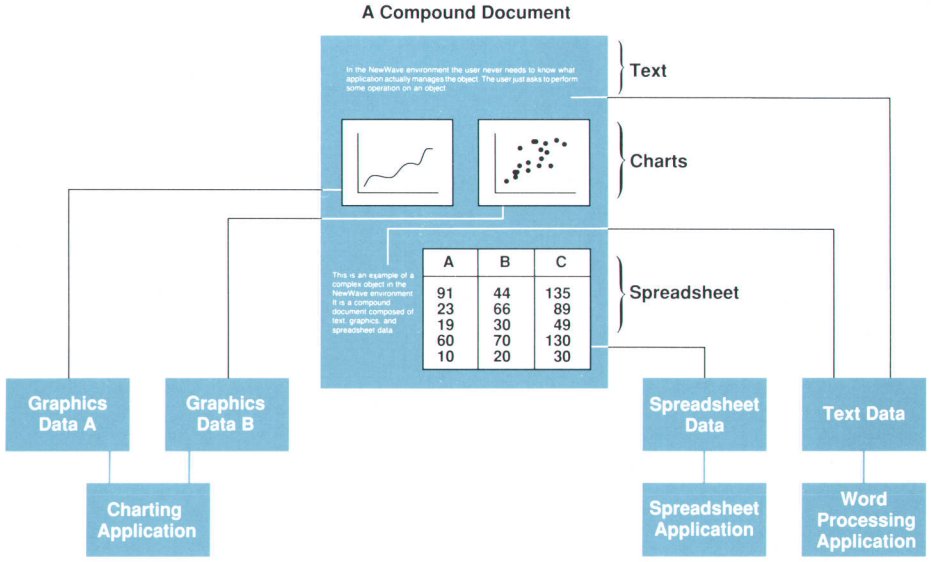


Fig. 1. *A complex object and its components. Each of the component objects has associated with it an application and a data segment containing the data it is currently managing.*

is, "Which application should I run?" Only after the application has been selected can the user indicate which data should be used, typically by remembering and typing in rather cryptic file names. If that data is not of the correct type for the application, the user is given an error message.

Transferring data between systems or among the applications within each system is a significant and yet difficult part of many common tasks. Often such data transfers must be done manually on a trial-and-error basis—even for repetitive transfers. Where applications allow several data files to be tied together, the result still must be maintained manually (e.g., when copying compound items such as text and graphics, or sending them via electronic mail). Unless the user is very careful and understands the system well, it is easy for a mailed document to lose its figures accidentally, or for a spreadsheet that consolidates the results of other spreadsheets to become separated from its components.

There are other annoyances as well. Many users must juggle several tasks at once, often changing from one to another as interruptions occur in the workplace. This is often difficult because MS-DOS®, the most common operating system for personal computer systems, restricts the user to operating one application program at a time. Exiting one program to run another is time-consuming, and the problem of remembering and restoring the context on returning is typically left to the user. Although many application programs have independently developed workable user interfaces, until recently there has not been much standardization across applications within this environment. Thus, users must learn and remember how to direct each application to perform its various functions, and often operations common to several applications are handled differently by each.

The net effect is that in the course of accomplishing the real task at hand, the user must also solve some additional problems introduced by the system: determining which of the available programs is appropriate for each step, remembering the names of the files that contain the data, and managing the movement of data between the programs. These problems have little to do with what the user really wants to accomplish—they are just extra things to worry about.

Tasks, not Tools

The NewWave environment is designed to address many of these problems. At the visual and operational level, it strongly encourages the basics of good software design, such as a consistent and logical user interface, the use of WYSIWYG (what you see is what you get) displays, and direct-manipulation interaction. Many of these characteristics are based on the features of Microsoft Windows, on which the current NewWave environment is built. The NewWave environment also inherits two very important features of Microsoft Windows: multitasking to allow more than one application program to be active simultaneously, and window management facilities so the user can switch among applications.

The NewWave environment takes a step beyond the features of Microsoft Windows by providing an architecture for managing data and applications and a consistent conceptual model to help the user understand the system. Whereas most systems today are function-oriented, the NewWave environment is information-oriented. With the NewWave environment the user can operate in terms of the information stored in the system, and instead of deciding which application to run, the user's first decision becomes "Which information do I want to work with?" The system will automatically pick the application that is appropriate for working with that kind of information.

A phrase we have used to capture the overall goal of the NewWave evironment is "tasks, not tools." By this we mean that users should be able to focus primarily on what they want to accomplish (their tasks), and not have to spend so much mental energy on the mechanics of how to do it (the software tools). Put another way, the computer should be part of the solution, not part of the problem.

The underlying NewWave architecture provides other benefits as well. Some of these, such as the task automation provided by the agent facility and the comprehensive help facility, are also directly beneficial to the user. Other characteristics, such as the ability of one application to use features of another, are primarily visible to application designers. However, they often benefit the user in the end. Several of these aspects of the NewWave environment are the subjects of other articles in this issue. Here, we will focus on how data and application programs are managed in the NewWave environment, and how these and other features of the system are presented to and managed by the user. In sum, these NewWave characteristics constitute a significant step forward in reducing the complexity from the user's viewpoint while simultaneously providing:

- An information-oriented user interface that makes the

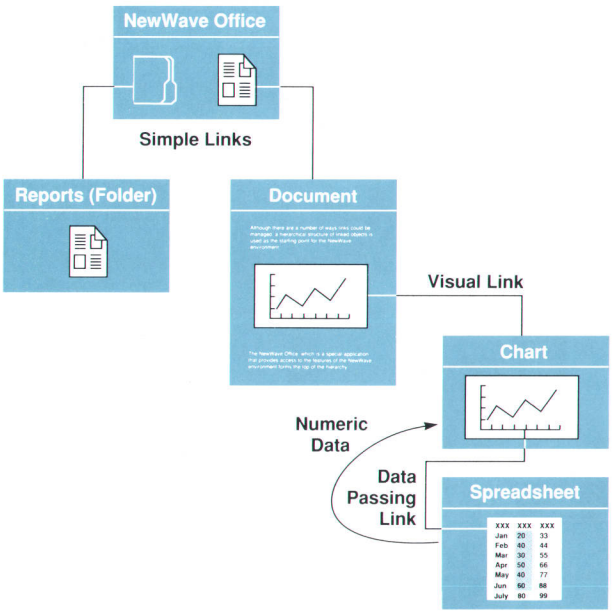


Fig. 2. *Simple links are parent-child relationships that attach the child to the parent with minimal obligations on the child. Visual links are used when a parent object requires the child object to display itself within the parent's window and print itself as the parent is printed. Data passing links require data transfer between the linked objects. Visual and data passing links are also called views in OMF terminology.*

process of running an application transparent.

- Information storage and retrieval that allow the workstation user to label, store, find, and manage information conveniently.
- Automated data integration among independent applications, including multimedia compound documents (e.g., documents that contain text and other data types such as graphics).
- Simple data interchange between systems, including exchange of integrated data items between work groups connected by networks and electronic mail.

Conceptual Models

A key consideration throughout the NewWave development was how to present the system features so they would be easy to understand and remember. As users explore a system, they develop mental images or models of how the system works. As a simple example, a new user will quickly observe that when the mouse is moved on the table, a cursor on the screen moves correspondingly. Consistent, logical behavior helps the user build the correct model and then reinforces it, making it easy for the user to predict what will happen in other similar circumstances. However, even the slightest inconsistent behavior of the system tends to break down the models the user has constructed, often leading to frustration and confusion.

To achieve this consistency, there should be a cohesive conceptual model of how things work. This model must be understood by the developers and presented clearly and consistently to the users. If there is a close enough match, patterning portions of a system's behavior after the real world by using visual and operational metaphors can help the behavior seem logical, or at least familiar—and hence more rememberable.

Also, the apparent complexity of the system is directly related to the number of different concepts and rules needed to describe how things work. Thus, consistency is important not only in doing the same things the same way within the user interface, but also in treating similar things similarly in the fundamental behavior of the components of the system. As will be seen, an object management architecture, together with an object-oriented user interface, was chosen as the best way to provide a consistent approach to information management.

The Object Model

To show how an object model addresses these problems, we should first describe a few key characteristics of software objects. In general, a software object is a set of data, plus the software required to manage the data and provide access to it. An object can be small, such as a word or a number, in which case the associated software is relatively simple, or it can be more complex, such as an entire document (see Fig. 1). NewWave objects are typically complex, corresponding to the kinds of data ordinarily associated with traditional office application programs: complete documents, spreadsheets, charts, drawings, and so on. In essence, the software associated with a NewWave object is an application program corresponding to typical office applications. However, there are some key differences between the way ordinary application programs and objects behave.

The most noticeable characteristic of an object is that a user never needs to know what program actually manages the object. The user just asks to perform some operation on an object, such as to see it, edit it, or print it, by using standard commands. The object management facility (OMF), which keeps track of all NewWave objects, knows which application software is appropriate, and runs it automatically behind the scenes to do the work. This provides the information orientation needed to remove the extra step of selecting and running the correct application. The OMF is described in the article "The NewWave Object Management Facility" on page 17.

Another key characteristic, not visible to the user, is that processes behind each object are designed to communicate with each other by sending and receiving messages. A standard message protocol allows one object to negotiate with another object to find out what operations the other object supports (e.g., displaying, printing, or user editing), and also to request the object to perform any of the operations that it supports. This combination of negotiation plus action lets objects interact flexibly with each other in meaningful ways, providing an underlying mechanism for automated data integration in the NewWave environment. In particular, it lets one object send information to or perform services for another object even though the objects may have been designed without knowledge of each other. For example, a chart object can get the data to be plotted from a spreadsheet object, or a document can ask a graphics object to display or print an illustration on its behalf. As will be seen, the user does see this communication flexibility indirectly, in that objects can easily be connected in various combinations.

Linking the Objects

As noted above, the ability of objects to communicate with each other, requesting and providing services and information, provides a key form of data integration in the NewWave environment. But to communicate, the objects must somehow be connected to each other. To take advantage of the flexibility of the object model, the user should be able to manage these connections easily among objects that are ordinarily independent. In the NewWave environment, these connections are accomplished by establishing persistent links between objects. The links are persistent in that they remain until explicitly broken, and like the object-to-application linkages, they also are maintained by the OMF.

NewWave Objects				
Tools		User Objects		
Simple	Containers	Containers	Data Objects	
			Compound	Simple
Terminal Printer Dictionary	File Drawer Wastebasket In Tray	Folders Envelopes	Documents Spreadsheets Charts Drawings	Images Text Notes Voice Notes

Fig. 3. *Categories of objects in the NewWave environment, with examples.*

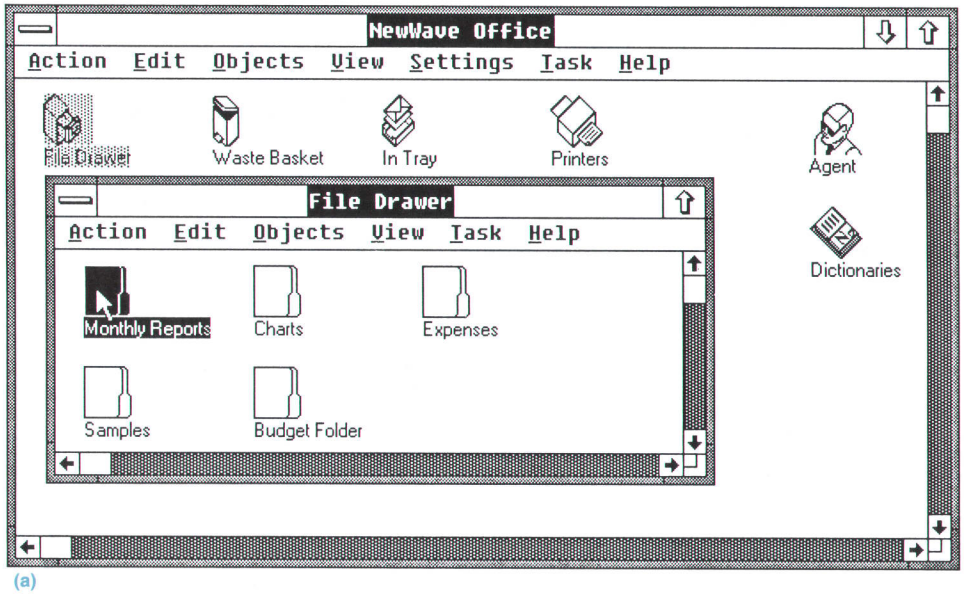
Although there are a number of ways links could be managed, a hierarchical structure of linked objects is used as the starting point for the NewWave environment. The NewWave Office, which is a special application that provides access to the features of the NewWave environment, forms the top of the hierarchy, with other objects descending from it (see Fig. 2). Within a pair of linked objects, the one that is closer to the top of the hierarchy is called the *parent* and the lower one is the *child*. In general, the set of all the objects below a given object might be called its descendants. The only structural restriction imposed by the OMF is that there can be no loops, in the sense that no object can be its own descendant.

Links between objects may be used simply to keep the objects together; these are called *simple links*. Links can also be used to let the child object provide data or services to the parent object; these are called *data links*. Because a data link in effect allows the parent to view a portion of the child object, data links are also sometimes called *views*. These provide for automatic updating of one object by another, a facility sometimes referred to as a hot connect.

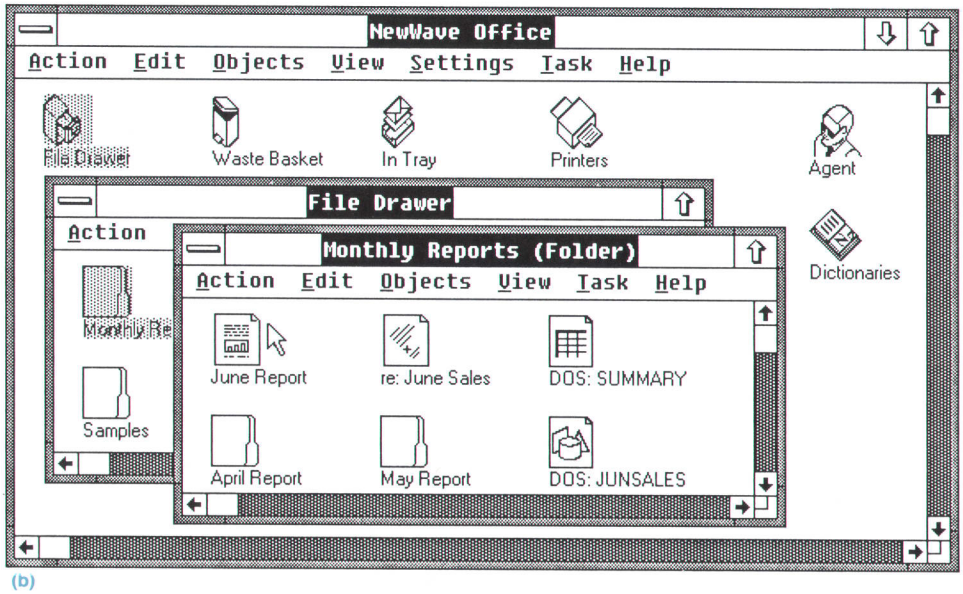
The Office Metaphor

Rather than requiring users to manipulate links directly, which would have required us to display hierarchy diagrams such as that shown in Fig. 2, the NewWave Office provides an office metaphor for managing and using objects. This metaphor is built around a containment model. In this model, an object is connected to its parent, and thus placed in the hierarchy, by putting it inside the parent object. To exist, an object must be attached to a parent—or in terms of the containment model, must be inside something. The visual presentation reinforces this containment model. For example, a folder window contains and displays icons that represent the objects contained in the folder, and a WYSIWYG document contains visual representations of the contained figures.

Objects in the NewWave Office fall into two primary categories: tools and user objects (see Fig. 3). Sometimes the latter are simply called objects. Note that the tools in



(a)



(b)

Fig. 4. *File system in the NewWave environment. (a) The file drawer contains the folder "Monthly Reports," whose contents can be displayed by opening it. (b) The folder "Monthly Reports" contains the report of interest. Note that icons are used to represent objects within the file system.*

Fig. 3 are singular because they are permanent parts of the system that cannot be duplicated, mailed, or destroyed by the user. However, they can be added or removed as part of system installation and maintenance, and it is sometimes possible to install two similar tools (e.g., two printers). User objects, on the other hand, can be freely created, copied, mailed and destroyed. The subcategories shown in Fig. 3 are less well-defined than the major categories, but still prove useful for classifying similar objects' characteristics.

Simple Tools and Simple Data Objects. Objects that nominally cannot contain other objects fall into these categories. However, some of the tools shown as simple can in fact hold objects temporarily while processing them (e.g., the printer object temporarily holds objects being formatted for printing).

Container Objects. The primary purpose of containers is to hold objects without actively using them. Since containers make no demands on their contents, they usually can contain any sort of user object. For example, the file drawer can contain folders, and folders can contain documents, but neither container does anything with its respective contents. But again there are exceptions; the in tray can only contain envelopes, and the wastebasket acts on its contents by destroying them when the user empties it.

Compound Data Objects. These are user objects that generally have data of their own, but can also contain other objects for specific purposes, usually to supply data or provide display services. Because they do place particular demands on their contained objects, only certain combinations are meaningful and are allowed (e.g., a document containing a spreadsheet is a valid combination, but a spreadsheet containing a document is probably invalid). Because the objects can negotiate with each other, the user can be told immediately if an improper combination is proposed.

Examples

Let's consider three examples that illustrate the implementation of the hierarchical object model and the containment model in the NewWave environment. The examples illustrate information storage and two kinds of data integration. Although in other systems these situations have typically each been handled differently, they all fit well into the hierarchical object model by using links in three different ways. This common underlying implementation makes it easy to provide a consistent user interface and conceptual model.

Example 1

In a single-user workstation a hierarchical filing system is a direct and natural way for the user to store things. The obvious metaphor is a file drawer containing several hanging folders, each possibly containing other folders and various data items. This can be represented directly by the hierarchical object structure, in which data objects are linked to (contained by) folder objects, which are linked (possibly through other folders) to the file drawer. Because no information passes between these objects, they can be connected by simple links. And because the folder displays no data from the contained objects, an object within a folder can be represented simply as an icon or a line of text, as in the NewWave Office (See Fig. 4).

From any level, the user can manipulate a lower-level child object as a whole entity by manipulating the icon. Items can be moved from one folder to another, or out of the file drawer entirely, by moving the corresponding icon from one window to another. The user simply points at the icon using the mouse, presses a mouse button to pick

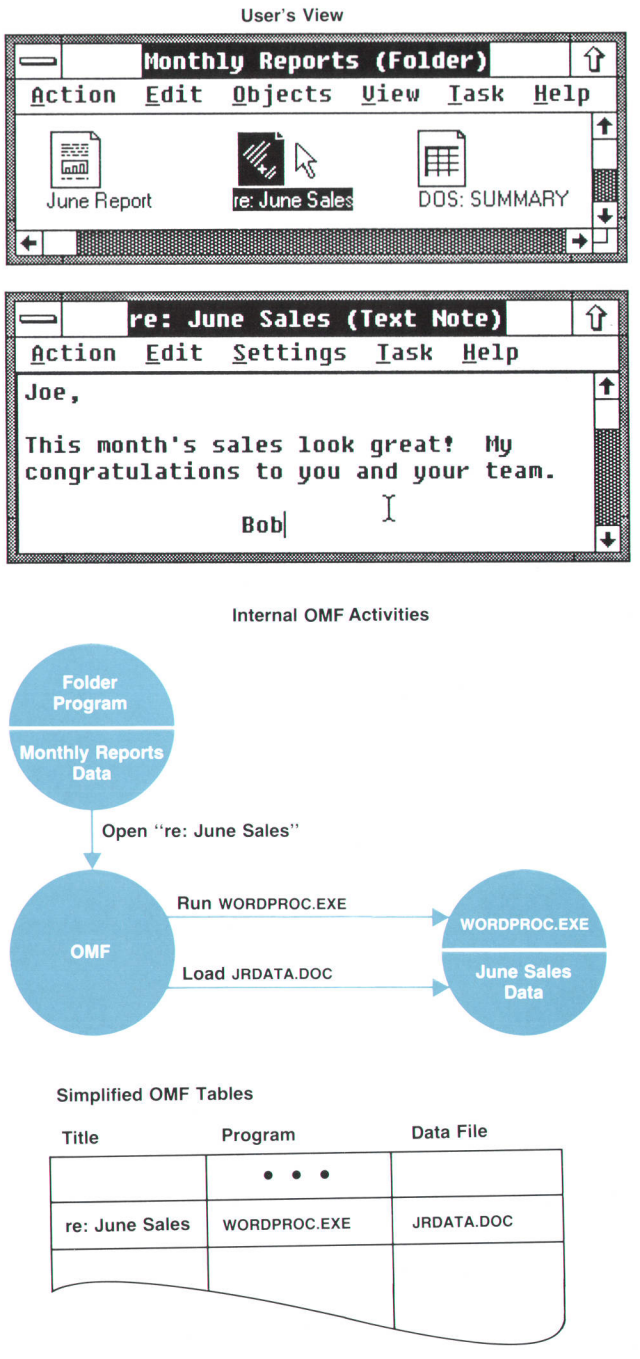


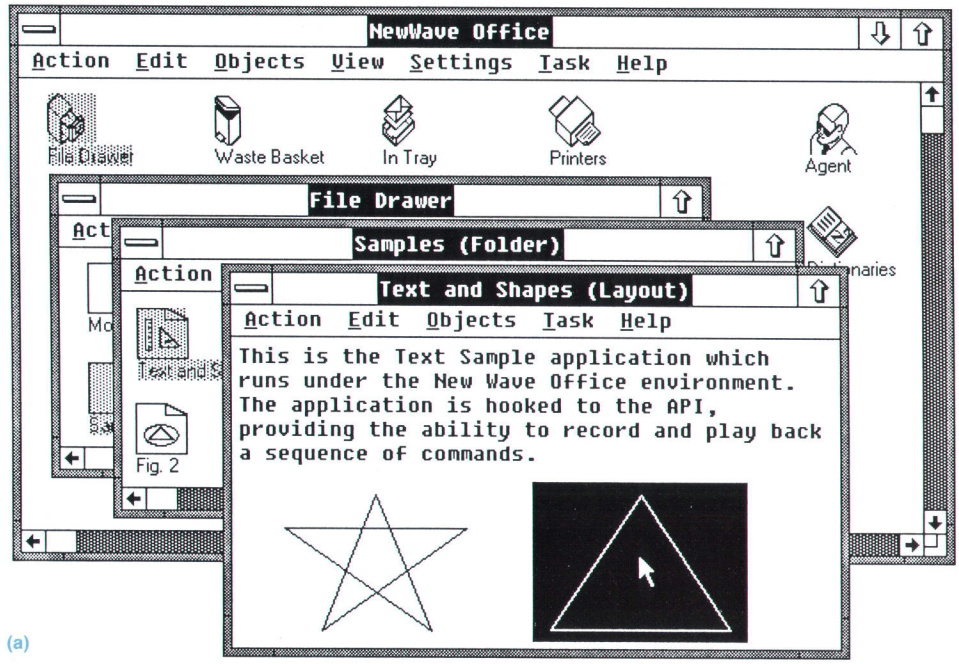
Fig. 5. *The OMF provides the linkage between the information the user wants to update (text file* JRDATA.DOC *in this example) and the application (*WORDPROC.EXE*) associated with the data. Double-clicking on the icon within the folder "Monthly Reports" would cause the folder application to request the OMF to open the corresponding object.*

up the icon, drags it to the new location, and releases the mouse button to let go. The OMF maintains the relationships between the file drawer, folders, and other objects contained in them based on changes made by the user. All the user needs to worry about is where the icons that represent the objects are displayed.

The user can also choose to open a contained object to operate on it in detail, by pointing at the icon and double-clicking (pressing the mouse button twice) as shown in Fig. 4. The OMF instructs the application program associated with the object to run, and provides the name of the data file. The application program then creates a new window and displays the contents of the open object for the user to manipulate as desired (see Fig. 5). The representation in the parent object is greyed to show it is open. When the user has finished, the window is closed, and the object is once again shown only in its parent folder.

Example 2

A compound document also provides a natural hierarchy in which the objects corresponding to the illustrations and tables contained by the document are all attached to it by a type of data link (view) called a *visual link*. These contained child objects are in fact responsible for displaying and printing the information contained in the corresponding illustrations and tables. Therefore, as in the first example, because the child object is linked to its own software, the user can double-click on the illustration to open the



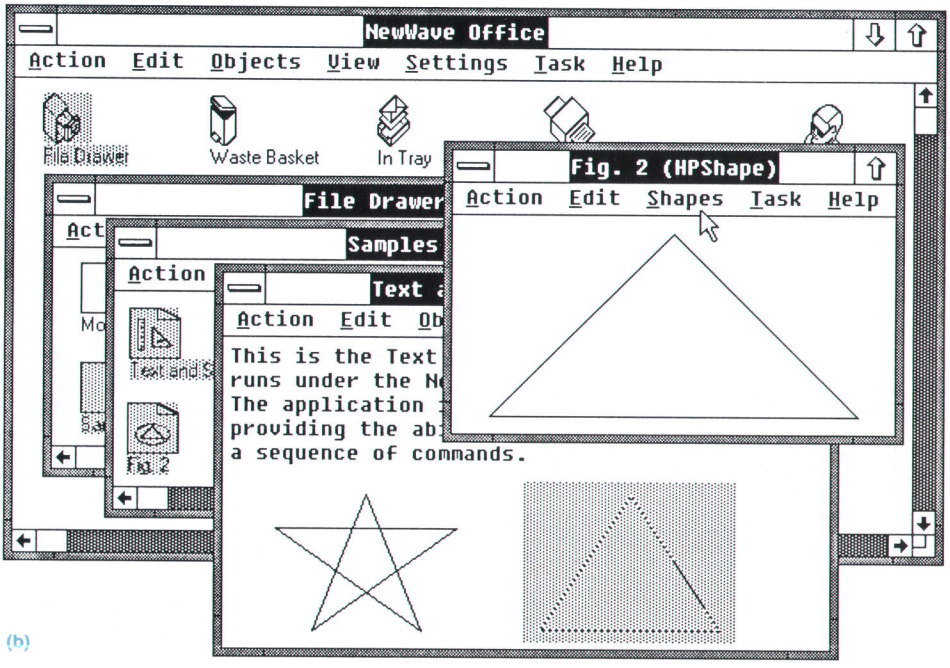
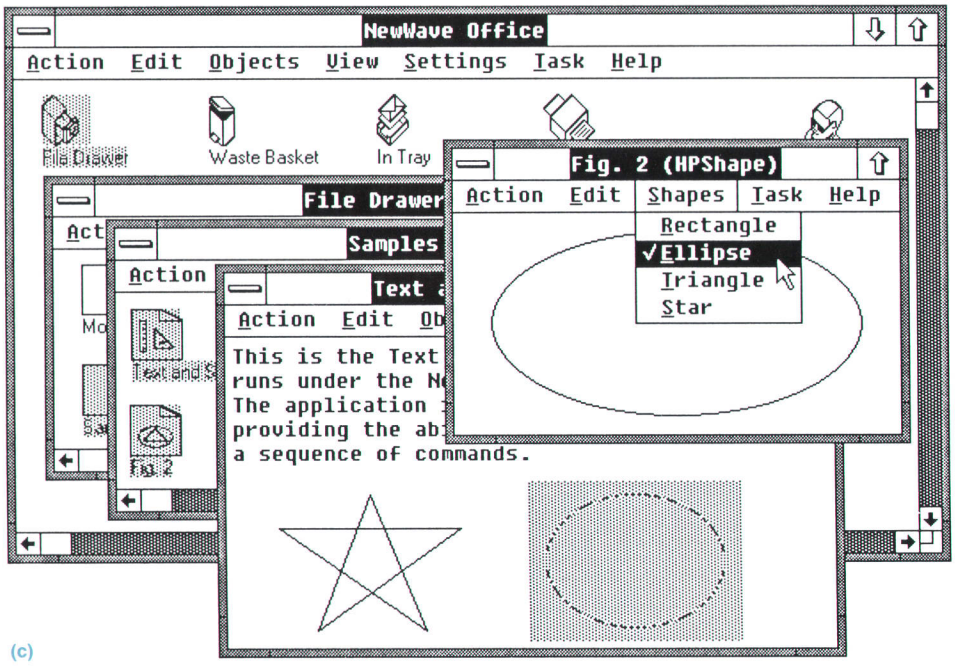


Fig. 6. *The process of opening a figure object within a document and editing it is illustrated using some sample applications provided to NewWave developers. (a) Each child object is attached to the parent (Layout object called Text and Shapes) by a visual link. (b) The child object's application (HP Shape) is responsible for displaying the object and providing the user interface. (c) and (d) See page 15.*

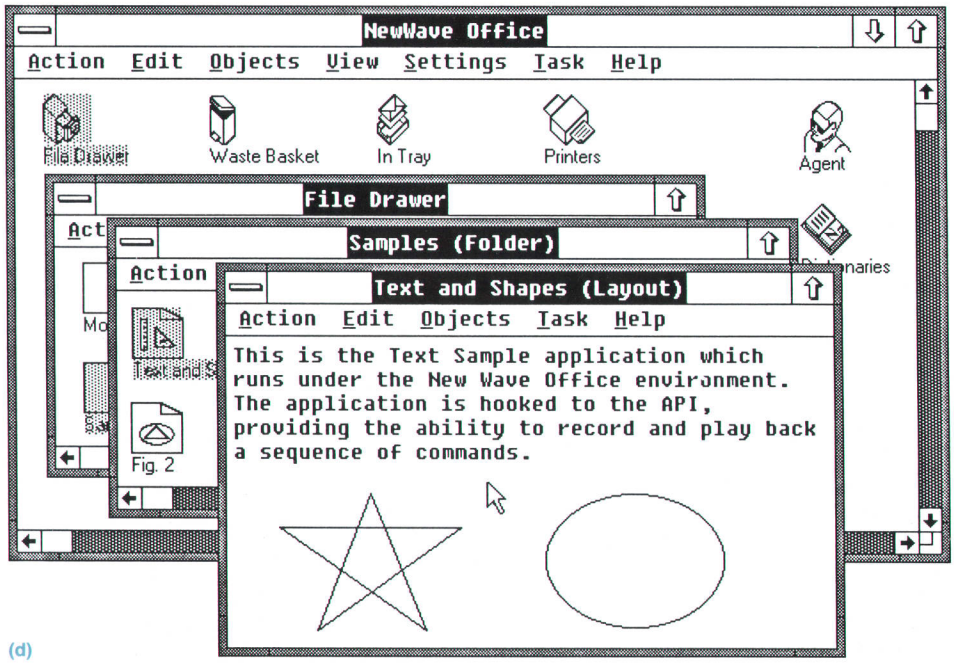
corresponding child object, thus running the associated application, and make changes to the illustration using the application's user interface. This is illustrated in Fig. 6 using two of the sample applications provided to NewWave developers, Layout and HP Shape. When the user closes the child object, the application associated with the child first updates the representation in the parent document automatically.

Example 3

In addition to being used to provide a service as in example 2, a data link can also allow data to be passed from one object to another for further processing. We call this a *data passing link*. A typical data passing scenario might have two spreadsheets linked together and passing data to a third, the consolidation sheet (see Fig. 7). This third spreadsheet combines the data, and is in turn linked into a chart, which displays some values from the consolidation spreadsheet graphically. Again, this can be described as a hierarchical set of linked objects. Here, in contrast to the document situation described in example 2, the child objects actually pass data to the parent objects for further processing through a data passing link. The parent may use the data, which consists of numerical values in this example, in calculations or for plotting the chart. And the same access benefits apply here as well—the user can open the child from the parent, make changes to it, and have the changes reflected automatically back in the parent. Be-



(c)



(d)

Fig. 6. *(a) and (b) See page 14. (c) The drawing is changed using the editing facilities provided by HP Shape. (d) The changed drawing back in the parent.*

cause the user starts from the initial representation of the data in the parent, it may not even seem like data transfer at all, but just the natural consequence of changing something and seeing the result.

In all these cases, objects can activate their applications automatically and perform standard services. Thus, the application (object) receiving the information—for example, the charting application—does not need to know anything specific about the other application program, or the file format it uses, or even the name of the file where the data is stored. Instead, it simply uses the appropriate standardized data interchange protocol. The immediate benefit to the user is that many more combinations of objects can be linked this way than would be possible in a traditional file-based scheme.

Because the links are persistent, the user never needs to worry about whether the components of the compound objects will be there. The OMF ensures that each one will continue to exist and be accessible as long as some parent object has a link to it. And because the links are managed centrally by the OMF, the user can copy the entire structure or mail it to another workstation simply by copying or mailing the top-level object, which contains all the others. The OMF, together with the objects themselves, ensures that each component is copied and placed in its proper place in the new object.

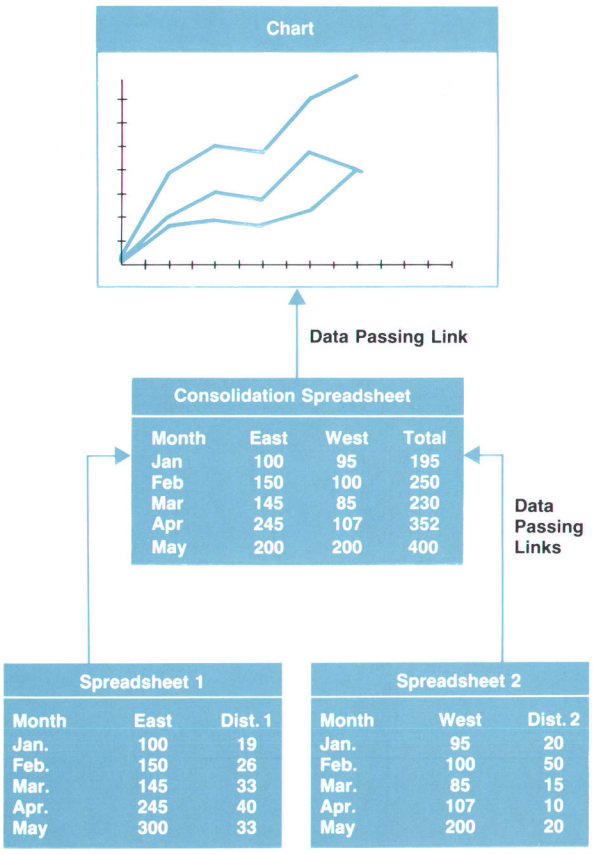


Fig. 7. *Two spreadsheets are passing data to a third, and the third, a consolidation sheet, is passing data to a charting program that displays the data graphically. These objects are linked together by the OMF using data passing links.*

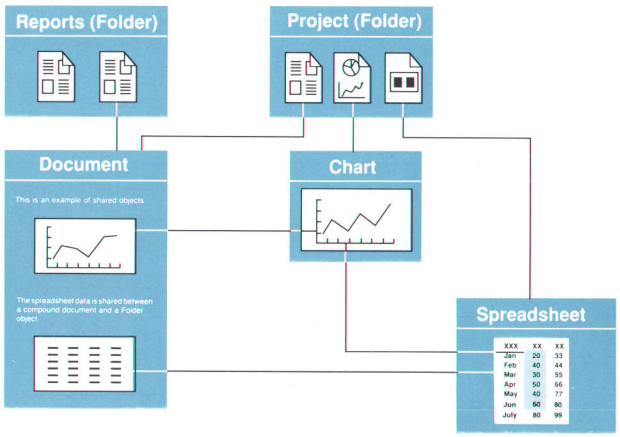


Fig. 8. *The spreadsheet is shared between the document and the chart, ensuring that both show consistent data. All the objects are also shared into the folder "Project" to allow direct access to each.*

Sharing Objects

In one important respect the NewWave environment extends beyond the capabilities implied by the containment model. A single object in the real world can only be in one place at a time. If a user wishes to use the same figure in two related documents, or to file a document under multiple categories in a file drawer, multiple copies of the item must be used. In the NewWave environment, an object can be linked to more than one parent object—or in the NewWave containment model, it can be contained in more than one place at a time. The user can control this by using the NewWave share command. On the surface share works like the Microsoft Windows copy command, in that the user sees the same data in the new location as in the old. The share command differs in that there is actually only one copy of the data. When changes are made to a shared object, the changes show up in all the locations that share the object. For example, the user may have a spreadsheet whose data is used to create a chart, and which is also displayed directly in a document along with the chart (document in Fig. 8). The user can also share the items into a folder, for example to allow direct access to the spreadsheet without going through the chart (folder in Fig. 8). Any change to the spreadsheet, no matter how it is accessed, will automatically be reflected in the chart and the document. This greatly enhances the user's flexibility in managing the objects and their connections.

User Interface Specifications

As noted earlier, the NewWave environment is designed to work within the Microsoft Windows environment. In addition to the architectural support features already mentioned, Microsoft Windows defines a user interface style, described in an application designer's style guide. This style guide sets the standards for using many of Microsoft Windows' features, such as menus, dialog boxes, and the data transfer clipboard.

The NewWave user interface specifications, described

for software developers in the *NewWave User Interface Design Rules* document, were developed as consistent extensions to this basic definition. The share command described above is an example because it has many characteristics in common with the existing copy command, and it is presented to the user in the same way. Other areas in which the Microsoft Windows style guide was extended include a more fully specified set of editing and cursor-motion commands, a more comprehensive specification for dialog box use, and the requirement that applications provide user help in a consistent way.

In a few areas there are conflicts, and the NewWave and Microsoft Windows standards differ. The use of an object model rather than the traditional applications and files model led to most of these differences. But the result is a standard that recognizes that users may need to use non-NewWave applications under Microsoft Windows concurrently with using the NewWave environment. The standard attempts to minimize the differences.

Conclusion

The NewWave environment's object technology provides a powerful, flexible and extensible data integration and management tool, allowing new applications to become full participants in the NewWave family without requiring changes to any other NewWave applications. This information orientation provided the opportunity to simplify many aspects of the system's behavior, and to make the user interface more regular. Together with a consistent set of user interface design rules, these features truly allow NewWave users to focus on their own tasks, rather than being distracted by system-imposed stumbling blocks.

Acknowledgments

Much of the original work that led to the NewWave environment, including the initial recommendation to use an object model, came from an earlier project at the Office Productivity Division in Pinewood, England, and particularly from Peter Williams. Many hours were also contributed by members of the various design review committees at the Personal Software Division in Santa Clara, California, developing and refining the object model and the specific design rules. These included Bill Crow, Andy Dysart, Paul Mernyk, Jeannine Sartori, Lynn Rosener, Ross Roesner, Wanda Shearer, Doug Smith, Lisa Towell, Bob Vallone, and Jon Weiner.

References

1. *Hewlett-Packard Journal*, Vol. 35, no. 8, August 1984, entire issue.
2. I. J. Fuller, "Electronic Mail for the Interactive Office," *Hewlett-Packard Journal*, Vol. 34, no. 2, February 1983.
3. A. P. Hamilton, "A Human-Engineered Small Business Computer," *Hewlett-Packard Journal*, Vol. 30, no. 4, April 1979.
4. K. A. Fox, M. P. Pasturel, and P. S. Showman, "A Human Interface for Automatic Measurement Systems," *Hewlett-Packard Journal*, Vol. 23, no. 8, April 1972.

The NewWave Object Management Facility

An object-based file system is the foundation of the New Wave environment. This paper describes the concepts and features of this system.

by John A. Dysart

THE NEWWAVE OBJECT MANAGEMENT FACILITY (OMF) provides the HP NewWave environment with a sophisticated object-based file system. The objectives for OMF can be translated into the following features:

- Focus on tasks. Our initial designs of a user interface for the NewWave environment clearly indicated that an object model could help solve many problems. An object model allows a user to spend more time thinking about the task to be done and less about how to get the computer to do it. The OMF helps by providing a file system for storing objects.
- Compound multimedia objects. The OMF is needed to keep track of all the relationships between objects. This knowledge can be used to manage compound objects as an integrated whole.
- Data sharing. OMF supports automatic data transfer so that new data can be entered in one place and then propagated to all the other places it is used.
- Code sharing. OMF helps the application designer by making it easier to write and use reusable code. The NewWave environment uses this ability to provide many system services that can be plugged into applications.
- Evolution versus revolution. Use of the OMF does not require developing applications in a completely different way. Current development languages and tools can be used. OMF also runs on the current generation of hardware and software platforms. Most important, the basic OMF features scale very well into the more sophisticated development environments that may exist in the future.

OMF Concepts

Objects and Classes

A NewWave user uses objects for the storage of data. Examples of objects include folders, documents, spreadsheets, and charts. Each object is a set of data files joined with an application program that is capable of processing those files. Many different objects with different data files can be bound to the same application program. Objects that share an application program in this way are of the same class (see Fig. 1).

New classes are installed into the OMF with an installation script that is written by the application developer. This script contains all the information the OMF needs to create and use objects of the new class. A user only needs to know the name of the script file, and the rest of the process is automatic.

There are two basic kinds of objects: global and user. Global objects are used to represent fixed entities in the system, such as a printer or wastebasket. Global objects are installed with the system and cannot be copied or destroyed by the user. Global objects can be referenced by any object in the system, and are often used to provide some common service to the other objects. Global objects that are visible to the user are called tools. User objects are objects created by the user to organize and contain data. User objects can be copied, moved, and destroyed by the user.

Some object-based systems allow objects to be used at a very fine level of granularity—for example, a separate object for each character of a document. In the OMF design, we took a more pragmatic approach and decided that objects should be used to represent larger entities, such as whole charts and reports. This approach allows applications to be programmed more conventionally, and is less demanding of hardware and software resources. OMF provides a very rich integration between these larger objects.

The data files associated with an object are just ordinary files in the MS-DOS® file system. The names of these files are generated automatically by the OMF and are not entered by or displayed to the user at any time. To optimize performance, the OMF automatically distributes the data files among a number of internal subdirectories that are created and maintained for this purpose.

All NewWave files, data and executables, are kept in MS-DOS directories separate from the user's other files. This allows the same storage volume to be used for other purposes, and makes it easier to archive and restore the NewWave portion of the file system.

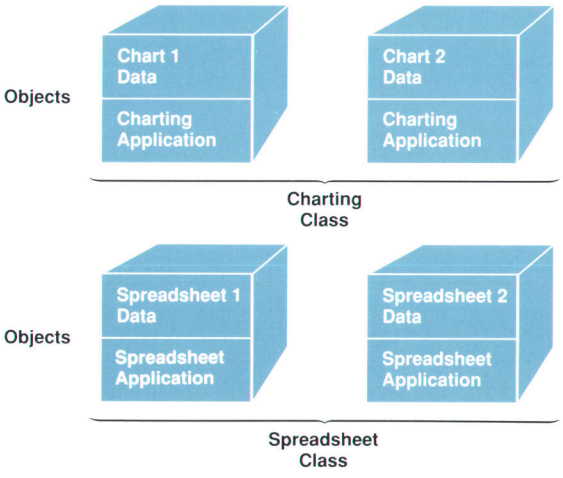


Fig. 1. *Objects and classes.*

Properties

Properties are chunks of data used to store descriptive information about objects and classes—for instance, the name of a class of objects, or the last time an object was modified. Fig. 2 illustrates some typical object and class properties.

Each object has a list of properties associated with it. An application can read or write the data associated with a property by specifying the object and the name of the property. There is also a property list associated with each installed class. The properties in a class property list are common to all objects of that class.

Property names can either be strings, such as MyProperty, or numbers. Numeric names offer the advantage of being more efficient to access and store.

A number of properties, such as the object title, have been standardized so that they can be used by all applications. Some of these standard properties are listed below. The OMF also allows application developers to define their own private properties.

PROP_TITLE	PROP_LASTWRITER
PROP_COMMENTS	PROP_MODIFIED
PROP_CREATOR	PROP_CLASSNAME
PROP_CREATED	PROP_TEXTID

Links

A key feature of the OMF is the ability to link objects together into compound objects. A link is a directional relationship from one object (called the parent) to another object (called the child). All objects have at least one parent, and can have any number of children. The one restriction is that the OMF will not allow an object to become its own descendant.

There are two kinds of links: simple and data. Simple links are usually used to support containment. For example, a folder object has a simple link to each object that it contains. Simple links are used to implement a hierarchical filing system, with the additional capability of supporting shared objects. Thus a single object can be contained in any number of folders, and be equally accessible from each. For example, a spreadsheet named "Western Division Sales Data" could be filed in a folder named "Western Data" and

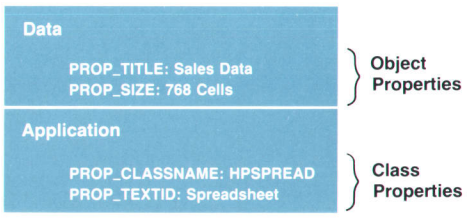


Fig. 2. *Properties.*

also in another folder named "Sales Data."

Data links, which are also called views, are a more sophisticated kind of link. Data links are just like simple links except that the child object can automatically transfer data to the parent object each time new data is available. If a view is simply displayed as an illustration in the parent object, the view is a visual view. For example, figures in a word processing object would be included using visual views. In a visual view, the parent does not get data from the child and then execute its own code to display the object, but simply tells the child to display the requested portion of the display. Alternatively, if data values are passed through the view, it is called a data passing view. For example, when a spreadsheet object gets data from another spreadsheet it is using a data passing view. Views are described in more detail later in this article.

Since the OMF knows about all the links in the system, it can manipulate a compound object as a whole when it is copied, mailed, or destroyed. Users do not need to remember and manipulate each component of the object separately.

Each link has a reference name that is a number assigned by the parent object to the link to identify a particular child. Parent objects use these reference names in their own data files to refer to their children. The reason for this is apparent when you consider what happens when a compound object is copied. By default, when a parent object is copied, the descendants of the parent object are copied as well (see Fig. 3). The copy of the parent object will expect to use the same reference name to refer to the copy of the child object that the original parent used to refer to the original child. Reference names provide the indirection necessary to make this possible. It is possible to override this default behavior when parent objects are copied. If a child object has a property named PROP_PUBLIC, when one of its parents is copied, the copy of the parent will be given a link to the public child, rather than to a copy of the child (see Fig. 4).

Messages and Methods

The OMF allows objects to communicate with each other using messages. For example, parent objects use messages to cause their children to display or provide data. Most messages are defined to work with any kind of object, so applications are isolated from having to know exactly the kind of object receiving the message. This allows new kinds of objects to be integrated into the system without having to modify any existing applications.

The code that an object executes when it receives a message is called a method. Saying that an object supports the X method simply means that it has code to process the X message. Sending a message to an object is similar to directly calling the method code. The distinction is that the message is generic; it can be sent to any kind of object without changing the code in the sending application. But the actual code executed when the message is sent is different depending on what kind of object receives the message. For example, when a message to print is sent to a word processing object the word processor code for printing a document is executed. And if that very same message is sent to a spreadsheet object, the spreadsheet application code for printing a spreadsheet is executed.

Applications send messages by calling the function OMF_Send. The sender specifies the reference name of the destination object, the message type, and any additional parameters the message may have. The OMF directs the message to the appropriate object, which receives the message as a Microsoft® Windows message. The receiver performs the requested function and returns a status value. The OMF then returns this status value as the return value of OMF_Send. This process makes messages synchronous, meaning that the sender does not proceed until the destination has processed the request (see Fig. 5).

Life Cycle of An Object

There are six stages in the life of an object. Fig. 6 shows how an object moves between these stages in its life cycle. The stages are:

- Creation. Objects are often created by copying a template object of the desired type. Objects are also created when they are received as data in an electronic mail message, or when a temporary object is created to perform some task. The object that calls the OMF to create the new object becomes that object's parent.
- Activation. An object is activated whenever some other object tells the OMF that it wants to send a message to it. The OMF starts a new process running the object's application, and passes to it the names of the object's data files. While active, an object generally is in a loop, receiving and processing messages sent from Microsoft Windows, the OMF, other objects, and other sources.
- Opening. Activation of an object is invisible to a user. However, when a user wants to edit an object, that object needs to present an interactive interface to the user. This

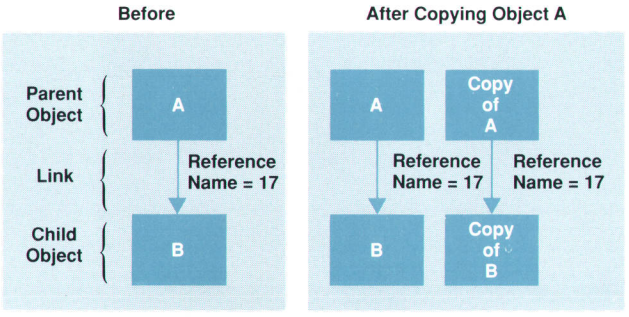


Fig. 3. *Copying a compound object.*

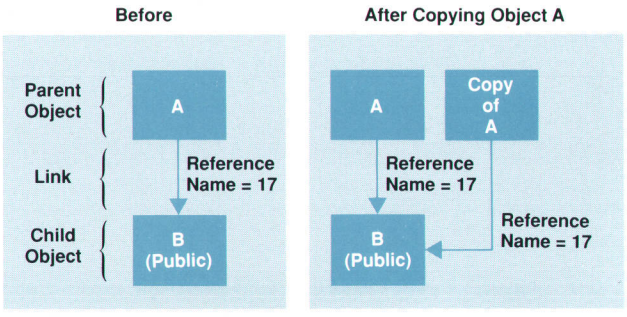


Fig. 4. *Copying a public object.*

is called opening. Generally, a parent object is responsible for providing a command in its user interface to open each of its children. When the user gives this command, the parent tells the OMF to activate the child, and then sends an Open message to it. When the child receives the Open message it presents its user interface on the display screen, and becomes interactive with the user.

- Closing. When the user finishes editing an object, the object's user interface is given a Close command. The object tells the OMF that it is closing, and then removes its user interface from the display screen.
- Termination. An object remains active as long as it is open or is being held active by some other object. The OMF terminates an object as soon as both of these conditions become false. OMF sends the object a Terminate message which lets the object save to disc any state information it has in RAM. Then the OMF terminates the process that was started when the object was activated.
- Destruction. When an object only has one parent and that parent deletes its link to the object, the OMF destroys the object. Destruction reclaims all the disc space that was allocated to the object's data files and properties.

OMF Views

Perhaps the single most important feature of the OMF is the support of automatic data transfer through the use of views. A view is a special kind of link for data transfer from the child object to the parent. Each time the child is changed, any parent objects that depend on the linked data are automatically updated. Each view has associated with it a destination specification, a source specification, a data ID, a view class, and an optional snapshot object.

Destination Specification

The destination specification is a data structure that is maintained by the parent object to keep track of how it is using the linked data from the child. For example, for each figure in a word processing object there is a destination specification that provides information such as where in the document the figure appears and how large it is. Usually, the parent also keeps the reference name of the view in the destination specification so that it can map from the reference name to information about how the view is used. The OMF does nothing to maintain destination specifications.

Source Specification

The source specification is a data structure that is maintained by the child object to keep track of a part of itself that is being transferred through a view. For example, when two spreadsheets are linked together, the child maintains a source specification that tells it what range of its cells are linked to a parent. Like destination specifications, the OMF does nothing to maintain source specifications.

Data ID

The data ID is a number that the child object gives the OMF to identify a particular range of linked data. The child object must be able to map from a data ID to a source specification, and vice versa. A number of different views from different parents may all share the same data ID and source specification if they all use the same range of linked data.

The OMF keeps track of which data ID is associated with each view in a data structure called the OMF view specification. This allows the parent and child object to converse in the terms that are most natural for them, with the OMF providing the translation between them. For example, when any linked data is changed by the user, the child object tells the OMF the data ID of the changed data. The OMF then finds all of the parents with views of the child associated with the data ID of the changed data. The OMF notifies each of these parents of the change by sending them a DATA_CHANGE message containing the reference name of the view as a parameter. Fig. 7 illustrates the relationship between destination specifications, source specifications, and data IDs.

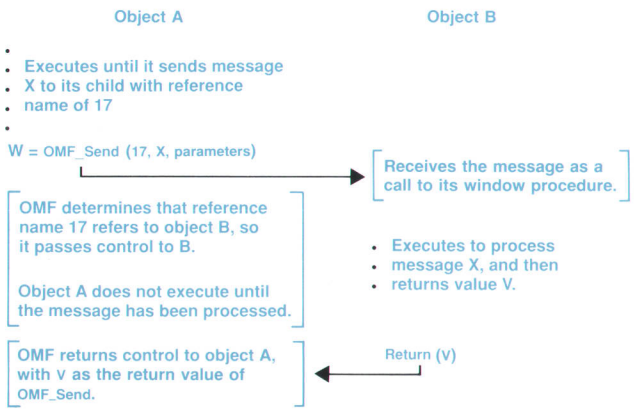


Fig. 5. *Control flow of* OMF_Send.

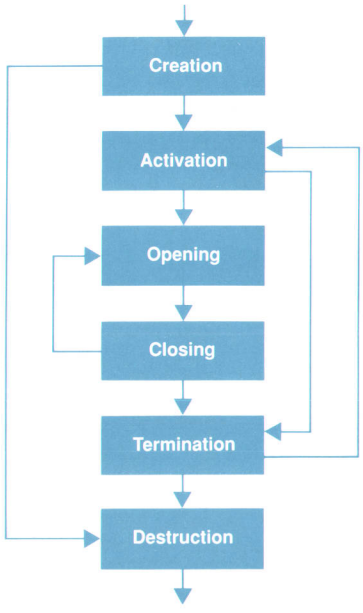


Fig. 6. *Life cycle of an object.*

View Classes, Methods, and Messages

Each view has a view class associated with it which is specified by the child object when the view is initialized. The view class determines the kinds of data transfer operations the parent object can request the view to perform. Each of these operations is called a view method and the parent requests them by sending a view message to the view.

If a child object supports the same set of view methods for all ranges of linked data, it only needs to use a single view class. An example of an object that needs more than one view class is a data analysis tool that can show both tabular and graphical representations of data. When a range of tabular data in this object is linked, a view class that supports transfer of data as text strings is used. However when graphical data is linked, the object uses a view class that supports transfer of data as a vector list.

View classes, view methods, and view messages are analogous to the classes, methods, and messages that were described earlier in this article, but are not exactly the same. The parent object uses different OMF functions to activate and send messages to a view than to activate and send messages to the child object of the view. The messages may ultimately be routed to the child, but they may instead be routed to a different object called a snapshot. Snapshots are described below. The indirection provided by view messages allows the presence or absence of a snapshot to be hidden from the parent object.

Snapshots

A snapshot is an object that serves as an intelligent buffer that allows the linked data from a child object to be accessed without activating the child. This results in better performance and use of resources. When a view is initialized, the child object tells the OMF whether or not to create a snapshot for the view, and if so, what kind of snapshot is desired. The child object then sends messages containing the linked data information to the snapshot. When the parent object sends a view message to the view, the OMF routes the message to the snapshot. The child object remains inactive. Fig. 8 illustrates how a snapshot is associated with a view.

One reason why snapshots can improve performance is that they are implemented differently from normal objects. Normal objects have applications associated with them, but snapshots have dynamic libraries associated with them. Like an application, a dynamic library is a separate executable file. It can be loaded and linked to while the system is running and unloaded when it is no longer needed. When a dynamic library is used to implement a snapshot, one procedure in the library is designated the message procedure. Each time a message is sent to the snapshot, the OMF calls the snapshot's message procedure. The message procedure determines the type of the message and processes it accordingly.

Unlike an application, there is no process or task associated with a dynamic library; its code only executes when it is called from an application. Dynamic libraries do not require that a stack be allocated for them because they always use the stack of the application calling them. In addition, since sending a message results in a call to the message procedure in the library, the overhead of a task switch through the operating system is avoided.

A second reason why snapshots can improve performance is that they only need to manage the data that is linked through the view. This data may be a subset of the child object's full data set (e.g., a small range of cells from a large spreadsheet), or it can be in a simpler form (e.g., just values instead of formulas).

There is some additional overhead associated with a

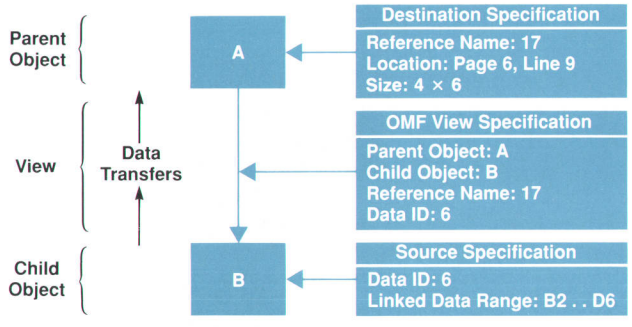


Fig. 7. *Destination, source, and view specifications.*

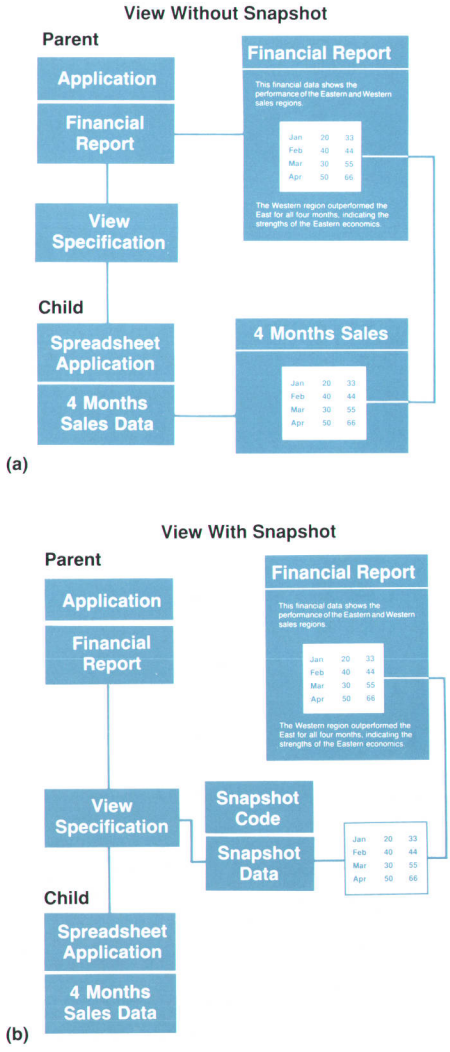


Fig. 8. *View with and without a Snapshot.*

snapshot. Each time the linked data is changed in the child object, the child must pass the new data to the snapshot before the snapshot can supply that data to a parent. Snapshots also take up additional disc space. Each application designer must carefully consider whether using snapshots will improve the performance of a particular application.

Routing View Messages

When a message is sent to a view, the OMF considers a number of factors in determining where the message should be routed. These include:

- Is there a snapshot?
- Can the snapshot handle this message?
- Is the snapshot's data up-to-date?
- Has the child object elected to process all messages?

The flowchart in Fig. 9 illustrates how the OMF considers these factors when routing a view message.

Other OMF Functions

The OMF provides a number of other functions to application developers, including event notification, clipboard support, serialization, and start-up and shutdown control.

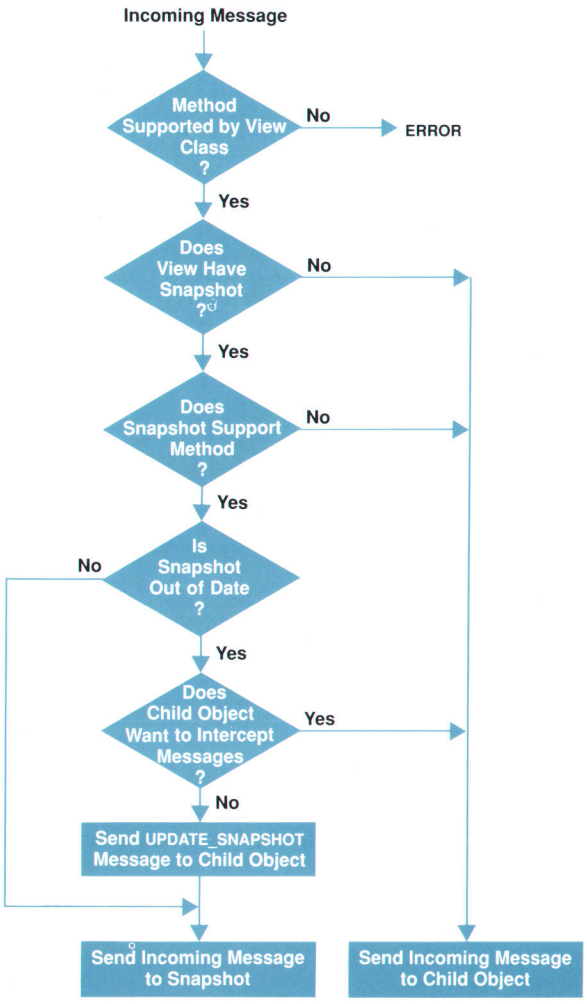


Fig. 9. *How view messages are routed.*

Event Notification

Objects can tell the OMF that they wish to be notified when some specific event occurs. OMF provides this notification by sending a message to the object. The following notifications are possible:

- A property change message is sent when an object's own properties or the properties of one of its children are changed. For example, it is necessary to notify all open folders containing a shared object when the title of the shared object is changed.
- A child opening or closing message is sent when a child of an object is opened or closed. It provides visual feedback concerning which object is being opened or closed.
- A copy or destroy message is used when an object is copied or destroyed.
- A configuration change message is sent when some system-wide configuration value is changed.
- A shutdown message is sent whenever the user tries to exit the NewWave environment.

Clipboard Support

Microsoft Windows provides a clipboard for data transfer which can hold any data that is in memory. The OMF enhances this clipboard by providing an invisible global object called the OMF clipboard. This global object serves as a temporary parent for any objects that are placed on the Windows clipboard. A temporary parent is needed so the object on the clipboard will have at least one parent and will not be destroyed while it is on the clipboard.

The OMF also provides a facility for storing large amounts of data, that is, more than can fit into memory, in temporary files that are known to belong to the clipboard. OMF can then take care of deleting these temporary files when the clipboard is cleared or used for some other purpose.

OMF provides functions that applications can call to put objects on the clipboard, remove them from the clipboard, and empty the clipboard.

Serialization

The OMF provides the ability to serialize compound objects. Serialization takes an object that may have many data files, plus its descendents and their data files, and packages all this data into a monolithic stream of data called a serial file. This serial file can be easily copied onto a flexible disc or transmitted through the mail network. On another NewWave system, the serial file can be deserialized. This process unpackages all of the objects and data in the file and produces a compound object that is a copy of the original object.

In general, the OMF can handle the serialization and deserialization of objects without any help from the objects themselves. However, in certain cases, objects may want to transform their data in some way when it is being copied to a serial file. If an object being serialized supports a method called SERIALIZE, the OMF will activate the object and send it a SERIALIZE message, rather than serialize the object's data files itself. The object then copies its transformed data into the serial file using an OMF function. When the resulting serial file is deserialized, the OMF creates a new object of the appropriate class, activates it,

and sends it a DESERIALIZE message. The object reads the transformed data from the serial file using an OMF function, and creates its normal data files.

Start-up and Shutdown

The OMF is the first process started when the user runs the NewWave environment. The OMF locates and opens the system files, which are a data base of all the classes, objects, links, properties, and so on. Once it has initialized itself, the NewWave environment activates and opens a special global object called the NewWave Office. The NewWave Office provides a user interface for much of the OMF's functionality. See the following article, "The NewWave Office," for more detail. The OMF itself is never visible to a user of the NewWave environment.

When the user closes the NewWave Office, a function in the OMF is called to shut the system down. Any objects that are active when the system is shut down can request that the OMF restart them when the system is restarted. The next time the OMF is started, it will activate and open those objects after activating and opening the NewWave Office. The benefit of this to the user is that the NewWave environment can be exited and reentered without losing track of current status.

Conclusion

The NewWave OMF provides the foundation of the NewWave environment. It implements a sophisticated object-based file system that can be accessed by any NewWave application. The OMF supports a powerful mechanism for building compound, multimedia objects with automatic transfer of data when changes are made. Although it does not have any user interface itself, it is in some ways the most important part of the NewWave user interface.

Acknowledgments

The first prototype of the OMF was developed at the Office Productivity Division in Pinewood, England by Brad Murdoch, John Senior, and Brian McBride. Much of their initial work still stands. Chuck Whelan deserves special acknowledgement for his key role in the design and implementation of many parts of the OMF. Thanks also to Ian Fuller and Bill Crow for their effective management of the OMF's development.

The NewWave Office

The NewWave Office is the user interface for the NewWave environment. It provides the tools and methods to perform tasks found in a regular office environment.

by Beatrice Lam, Scott A. Hanson, and Anthony J. Day

THE NEWWAVE OFFICE IS THE FOCAL POINT for the user's interaction with the NewWave environment, and it is the first NewWave object the user sees when the NewWave environment is initialized. It remains active throughout the entire session until the user terminates the NewWave environment. It incorporates many special features to reinforce the office concept in the minds of users. These features include iconic representation of tools found in a real office, such as a file drawer, a wastepaper basket, a printer, and so on (see Fig. 1). The diagnostic tool shown in Fig. 1 is not a typical tool found in an office, but is a tool that enables NewWave application developers to interface to the NewWave object management facility (OMF).

It is easy to work with these tools using a mouse to manipulate the icons that represent the tools. The tools are NewWave objects, as explained in the article on page 9. The user can either open the tool and move other NewWave objects directly into the opened tool window, or drop other objects on the icon representing the tool. Incoming objects are handled by each tool according to the function of the tool. The file drawer and the wastebasket accept the incoming object and display its representation in their windows. The printer, on the other hand, asks the object to print itself on the selected printer device. The diagnostic tool accepts an incoming object into its window and displays the OMF information pertaining to that object. In addition to viewing objects as icons, the user has the option to switch into the list view. In the list view, the title, type, and modified date are displayed, and the objects are sorted by one of these parameters (see Fig. 2).

This article describes the main features of the NewWave Office and shows how these features interact with the other NewWave components shown in Fig. 3.

NewWave Windows

The window that represents the NewWave Office is designed to remain as a background window while NewWave objects are open as pop-up windows over it. Activating an object window brings it to the top, overlapping other windows on the display (see Fig. 4). Since the office tools are always available, the user can easily manipulate an object from its window onto a tool at any time (e.g., moving a document to the wastebasket from a folder's window).

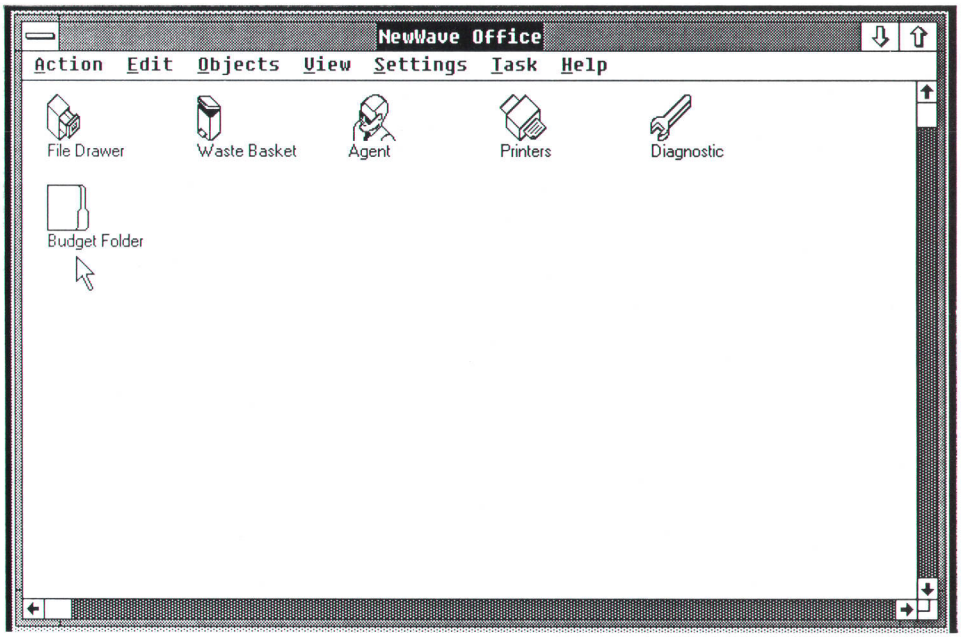


Fig. 1. *NewWave Office window.*

Another distinct feature of the Office window is its maximized window. When the NewWave Office window is maximized, all object windows remain in the same positions while the Office window occupies the full screen in the background. When an object window is maximized, the object is brought to full screen and all other windows fall behind. The Office window can also be minimized; this hides all object windows and turns the whole NewWave Office into an icon.

To achieve these special display features, the NewWave Office uses functions from the library HPNWLIB.EXE. These functions are:

- NW_CreateWindow. This function creates an object's main window and binds the object window as a child to the NewWave Office window.
- NW_Minimize and NW_Maximize. These functions perform the window maximize and minimize operations described above.
- NW_Restore. This function returns a window from a maximized or minimized state to its original size.

The default windows for the file drawer, the wastebasket, and folders are positioned in the NewWave Office window as shown in Fig. 4. The first window is positioned about halfway down the screen and partway in from the left, and the rest are staggered down and to the right of the other default windows. By using a defined parameter to the NW_CreateWindow function, NewWave applications can create default windows for themselves.

Container Objects

The NewWave Office is used as a temporary work space to store objects the user is currently working on. The file drawer and folders provide the NewWave user with a convenient hierarchical filing and retrieval system to manage information. The wastebasket is used for discarding objects the user no longer needs, and a user-defined maximum number of objects can be set to trigger automatic emptying of the wastebasket when it opens. These four objects are called container objects because they are capable of containing other objects (e.g., the file drawer contains folders and folders contain documents, and so on). These objects display and manage their child objects in identical ways. The menu for each of these object types is customized to the particular needs of that object. For instance, the wastebasket's Edit menu has the Delete command, whereas all other object types use the Throw Away command.

The NewWave Office, the file drawer, the wastebasket, and folders share the same executable code. Every invocation of the same code is an instance, a feature provided by the Microsoft® Windows environment. For efficiency, only a single copy of the code is in memory, even though there are multiple instances running. The code is made up of multiple code segments, and only those segments in use are kept in memory. As other segments are needed, they are read in from disc. Each instance has its own data segment, which includes global variables, the stack, and a local heap.

When the first instance of the NewWave Office starts up, it computes certain environment variables and stores the results into some global variables. Also, several text fonts, paintbrushes, cursors, and other items are created and stored. Each succeeding instance needs to use these same items, and so copies them to its own data segment at start-up time instead of recomputing and recreating them itself.

When the user runs another instance from the NewWave Office, such as the file drawer or folder, that object must open up to the same state it was in when last closed. This includes the object's window size and position, whether it was in iconic view or list view, where the window was scrolled to, and more. To do this, most of the needed information is kept in a data file associated with the object. The information is read from the data file and placed in the appropriate global variables for use when creating and displaying the instance's window.

The container must also determine the list of children contained within it (e.g., the list of folders in the file

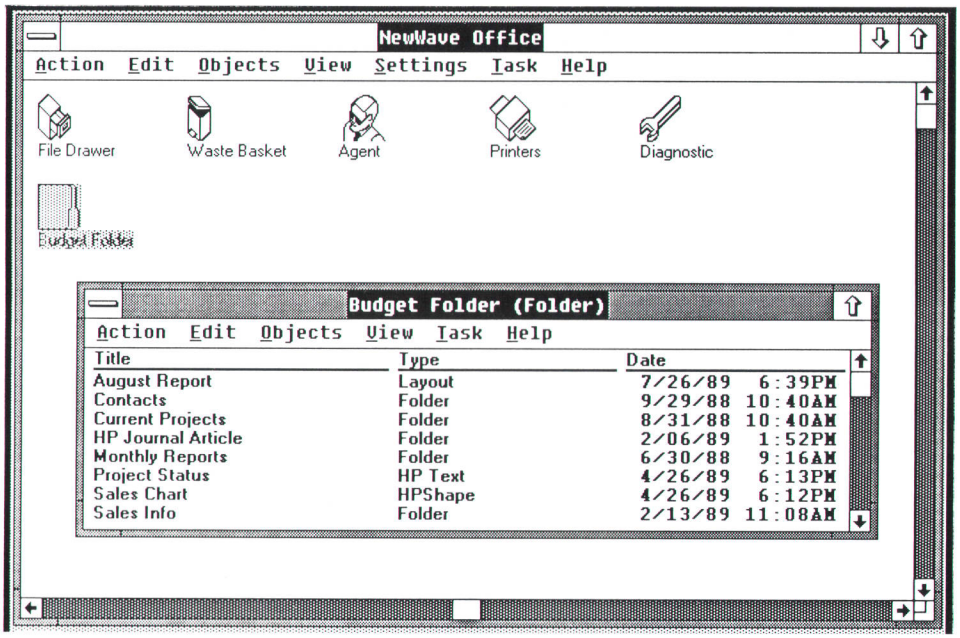


Fig. 2. *A list view representation of a folder.*

drawer). This information is also kept in the data file and is read in and placed in data structures in globally allocated memory. The children information, which is kept in the data file, is not always up to date. While the container was closed, new children may have been added, titles may have changed, children may have been opened or closed, and so on. The first situation might have occurred as a result of the OMF_AddChildTo function (discussed on page 30), and the latter two would have occurred because the container is shared with other objects. To ensure that the data in the data file is correct when the child object is opened, the latest list of children that belong to the container, and information about them, are obtained from the OMF. The information given by the OMF always overrides the data file, and when a container is opened a check is made with the OMF before finalizing the data structure. The OMF function OMF_EnumChildren is used to enumerate a container's children and to obtain the information to update its data structure. During enumeration, the OMF sends the container ENUM_OBJECT messages for each child belonging to it. In processing the message, if the child is not in the data file, it is added to the data structure. For new child objects, the information obtained from the OMF includes the object's title, its icon handle,* its active state, and the date and time when the object was last opened. All of this information is added to the data structure.

To optimize enumeration, an object property called PROP_FASTPROPS allows retrieval of numerous pieces of information about an object in only one reading. PROP_FASTPROPS includes the last-modified date, the last writer name, tool display information (PROP_SYSTEM described below), and the object's title string.

The NewWave Office must also enumerate all of the tools. This is accomplished through the OMF function OMF_EnumGlobalObjects. As with the data objects, the tools that are visible in the NewWave Office are added into the data structure using the information in PROP_FASTPROPS. Using the Manage Tools dialog box, which is available from the Settings menu item, the user can select which tools to display in the NewWave Office window and which are to be hidden. This status is stored in the object's PROP_SYSTEM property: 0 means never display it (e.g., the OMF clipboard), 1 means don't display it now, and 2 means display it. If the tool is never to be displayed and is in the data file, it is removed from the data structure. There is no need to waste space storing information on an object that is not displayed.

The data structure that holds all this information is composed of two parts. The first part is an array of structures, one for each child in the container. This child array holds the OMF object name for each child, its location in the container, a flags word that indicates whether the object is selected or opened and other status, the pixel width of its title on the screen, the handle to the object's icon, the date and time of its last change, and indexes to its title and class strings. The second part of the data structure is the string lists. They hold the title and class strings for each object. The strings are stored end-to-end and each is preceded by a length word and terminated by a NULL byte. This minimizes the overhead needed per string. These strings can be accessed quickly because each child's structure in the child array contains offsets into these string lists for its title and class strings. One optimization performed is not to store duplicate strings in the class string list. If a newly added object's class string is already in the list, the object's child array structure references the one already in the list. This works out well because a container will usually contain many similar objects, that is, objects of the same class.

*A handle is a pointer into a table of pointers. The pointers in the table point to data or code segments scattered throughout memory that are allocated to the object that owns the handle. Thus, the object's icon handle points to the location in the table (handle table) that points to the memory location where the object's icon is stored.

Office Functions

The NewWave Office not only provides the central user interface for object management, but also works in conjunc-

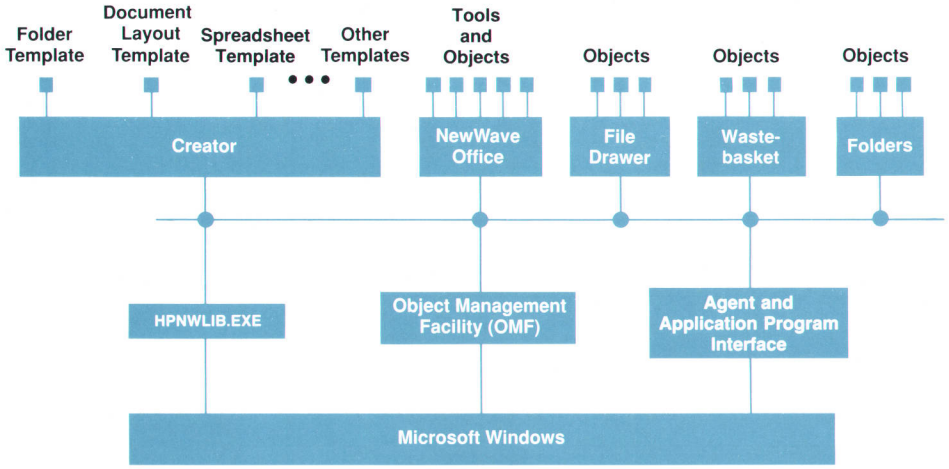


Fig. 3. *The architecture of the NewWave Office and the other components it uses in the NewWave environment.*

tion with the OMF to provide essential architectural components for the interaction with NewWave applications. The close cooperation between the NewWave Office and the OMF is achieved with the use of various OMF function calls, and the interaction with NewWave objects is done using predefined OMF methods. A few examples of these special interactions will be given in the following sections.

The Create Process

The user will frequently want to create a new object, which may be in the form of a new document, a new spreadsheet, a new pie chart, a new data base query result table, or some other typical office object. There were two requirements for the user interface that displays the object types available to be created.

- The interface had to allow the user to invoke object creation facilities from the NewWave Office window and from tools such as the file drawer, from containers such as folders, and from compound data objects such as a compound document.
- The interface had to show the user only those objects that can be created in the domain in which the user is working. For example, when the create process is invoked inside a compound document object, a folder object cannot be created.

The OMF maintains a list of all types of objects in the user's NewWave environment. This list represents a set of empty template objects and class information such as which methods a class of objects supports. Object templates are objects that are frequently used (e.g., form letters, expense reports, distribution lists, etc.). To manage these template objects from within the NewWave Office and other objects, and to provide the features mentioned above, a tool called the creator was developed. The creator is a global NewWave object of type tool. Its object creation facilities are made available to the user by selection of the Create a New... command in the Objects menu.

The creator has the only reference to all the template data objects installed in a NewWave environment. The creator's global OMF reference name is known throughout

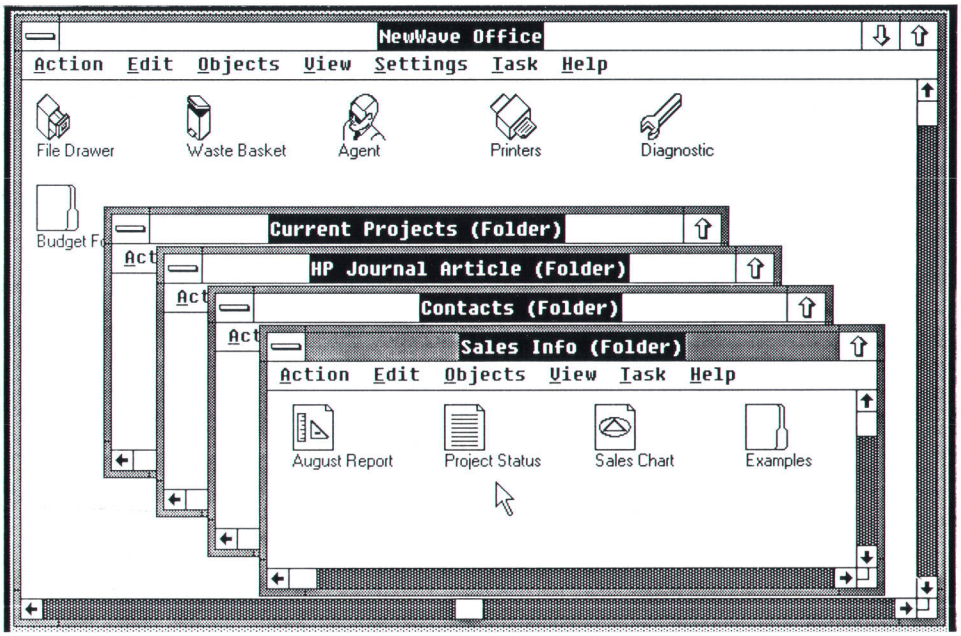


Fig. 4. *Object windows overlapping one another.*

the system and any application can establish communication with it by simply specifying that reference name. Developing the creator as a distinct NewWave object provides all of the benefits of objects in the NewWave environment. Its protocol is defined and established so that other objects can communicate with it. The process of creating a new object is simply a matter of copying the template object and ensuring that the parent object, from which the user created the new object, establishes an OMF reference to it.

CREATE_A_NEW Message. Communication with the creator to create a new object is accomplished with the CREATE_A_NEW message. All container type NewWave objects are required to provide the menu item Create a New..., which is used to set into motion the process of creating a new object (see Fig. 5). When the user selects Create a New... the object invokes the creator using the OMF_GetOMFObject call and sends it a CREATE_A_NEW message. It then terminates conversation with the creator by calling OMF_FreeOMFObject. Sometime after terminating conversation with the creator the object may receive an OMF_NEW_OBJECT message containing the newly created object. If the user cancels dialog with the creator, this message may never be received. To prevent problems with reentrancy while waiting for the new object, the creator disables the caller's window. This prevents the user from doing anything else in that window until business with the creator is finished.

The CREATE_A_NEW message may be accompanied by a memory handle to some global shared memory containing a list of methods the calling object requires its children to support. When the creator receives the CREATE_A_NEW message, it displays to the user a dialog box containing the icons and titles of all template objects that support the list of methods sent (see Fig. 6). If a methods list is not sent, it is assumed that all template objects are suitable and they are all displayed. The user is then able to choose an object to create and give it a title.

When the user has chosen an object to create, given it a title, and hit the OK button, the creator calls an OMF routine to make a copy of the template object. The creator has a temporary reference to this new object. It uses this temporary reference to write the user-supplied title to the new object's PROP_TITLE property. It also writes other properties of the new object such as PROP_CREATED, which is the time and date of creation, and PROP_CREATOR, which is the user's logon name. The creator then sends the calling object (the parent) an OMF_NEW_OBJECT message that has a reference to the new object as a parameter. The calling object is expected to absorb the new object into its data structure and establish its own permanent OMF reference name to it. When control returns to the creator from this message, the creator deletes its reference to the new object, severing any further connection with it. If, for some reason, the calling object does not establish a permanent reference to the new object, the OMF detects that the new object has no references to it from any object and destroys it.

Installing Objects. Since the creator is used to create new objects in the NewWave Office, it is also part of the process of installing new applications into the NewWave environment. To install an object (i.e., an application) into the NewWave environment the user must provide the following files:

- An appropriate .EXE program file
- Default data files
- A help file
- An icon file in standard MS Windows format
- Any other necessary files, such as configuration information
- An installation file.

The installation file, which normally has the extension .IN$, is a command file that specifies everything the system needs to know about the object being installed. Entries include the class name, which defines the type of object, paths to the files on the installation disc, instructions defining where those files should be placed in the system, methods supported by the object, and other installation data. The installation file format is very flexible—for exam-

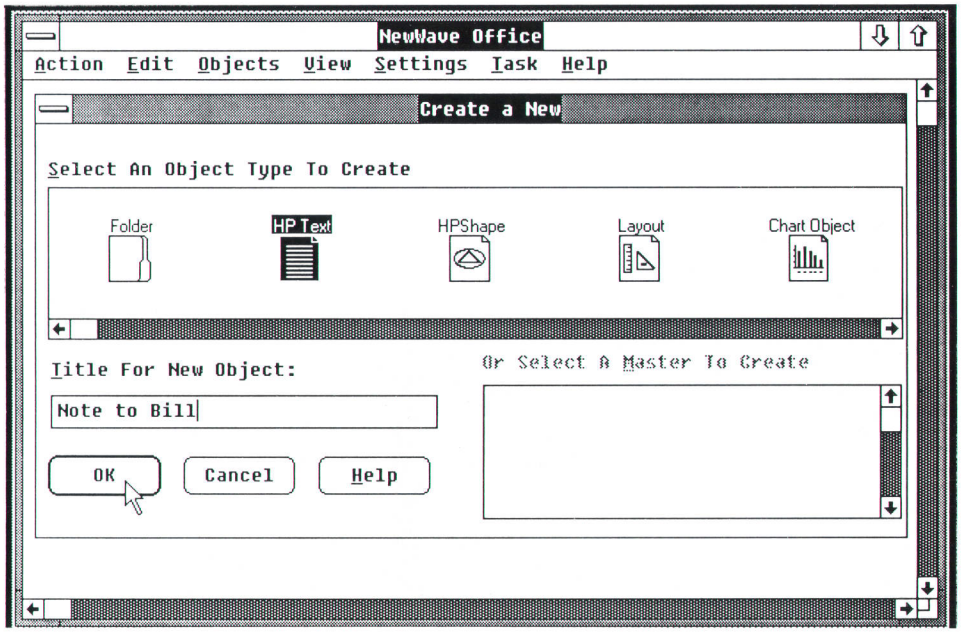


Fig. 5. *Display after selecting the* Create a New ... *menu item.*

ple, it is possible to install compound objects and provide updates for installed applications with or without altering existing objects of that class. It is also possible to install global tool objects such as the wastebasket, the printer, and the creator itself. Tools cannot be created by the user; therefore, they can only be added to the system by installation.

The installation process is designed to execute with minimal participation on the part of the user. It is possible for a user to install a complete NewWave environment merely by inserting the NewWave disc and running the NewWave install utility. The install utility creates a file called HPINSTAL.IN$ in the NewWave system directory on the hard disc. This file contains a list of paths to one or more .IN$ files on the installation disc. After copying some system files onto the NewWave system directory, the install utility starts up the NewWave environment by running the OMF.

The OMF is the first process to run at the start of a NewWave session. After performing some housekeeping chores, the OMF starts the NewWave Office. During its initialization phase, the NewWave Office looks for the HPINSTAL.IN$ file. If it finds one, it invokes the creator, passing the paths of the .IN$ files to it one at a time. The creator calls certain OMF routines to install the application and establish a reference to each new template object as it is built. At the end of this process, the creator, having successfully added these new children to its list, is terminated and the NewWave Office opens and presents its interface to the user for the start of the NewWave session.

MS-DOS® Objects. A major benefit of the NewWave environment is its ability to make objects from the data files of standard MS-DOS applications that were not written to run under the NewWave environment. This capability is known as encapsulation of MS-DOS applications (see article on page 57). Once an MS-DOS application is encapsulated, the user can create NewWave objects that represent data files pertaining to that application. These objects are known as MS-DOS objects. They can be displayed in the NewWave Office window the same as other objects such as the file drawer or folders. They appear as icons and can be manipulated by the user in the same way as true NewWave objects.When the user opens one of these objects, a shell is run, which then runs the appropriate MS-DOS application and associates it with the appropriate encapsulated data files.

Encapsulated MS-DOS applications are installed in much the same manner as regular NewWave objects. Besides the information provided in a typical NewWave installation file, an MS-DOS installation file contains information such as keystroke sequences used by an MS-DOS application to load the encapsulated data files.

MS-DOS objects that support the calling object's required methods appear in the creator's dialog box along with regular NewWave objects. If the user chooses to create an MS-DOS object, the MS-DOS filename associated with the new object must be provided. The MS-DOS object is then created and special object properties are written to the object's data files. When the user first opens this MS-DOS object, the MS-DOS application shell reads these properties to find out what data files to load with the application.

Masters

Once the decision was made to make the creator a NewWave object with the capability of managing other NewWave objects, a very useful feature quickly presented itself. This is the ability of the user to create template objects from existing NewWave objects. These user-created templates, or masters, can easily be sent to the creator which displays them and allows the user to create a new master in addition to new empty template objects. For example, the user can create a master from a form letter or spreadsheet using existing templates of the letter or spreadsheet.

Adding Masters. The Save As Master... command, which is used to save master templates, is currently implemented in the NewWave Office, the file drawer, and in folder objects. To save a master, the user selects an object icon in one of these window's objects and then chooses the Save

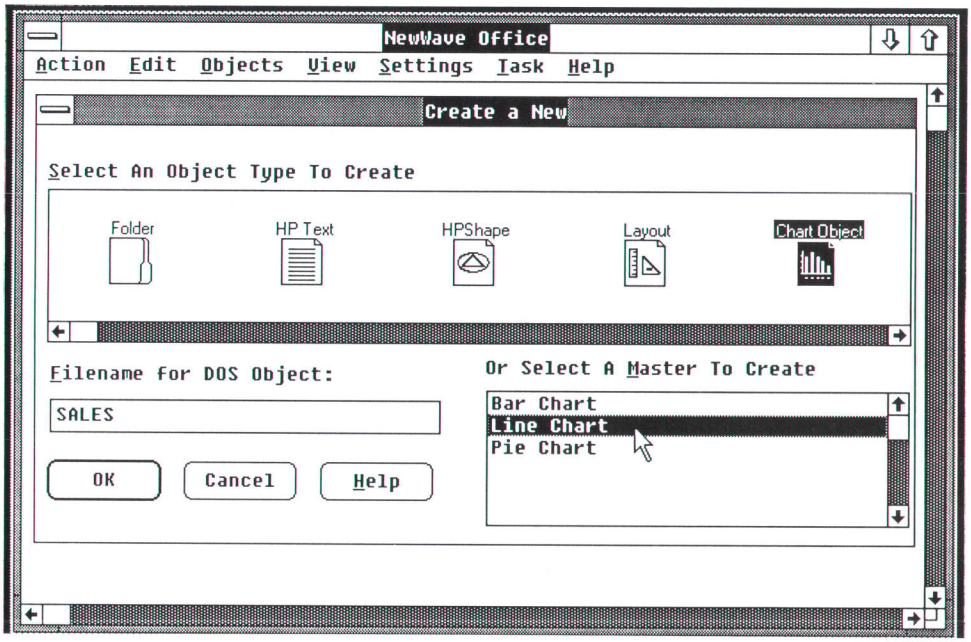


Fig. 6. Create a New ... *dialog box showing a list of template objects.*

As Master... command from the menu. The container object then makes a copy of the selected object, invokes the creator, as described above, and sends it an OMF_INSERT message with a reference to the copied object as a parameter. When the creator acknowledges the OMF_INSERT message, the container deletes its reference to the copied object. The original selected object is unaffected by this transaction. Objects saved as masters can be as complex as the user wishes.

Upon receiving the OMF_INSERT message, the creator verifies that the inserted object is an installed data object. It does this by comparing the object's class name with the class name of each of its empty template children. If the inserted object does not match any of the creator's children an error is signaled to the user. This can occur, for example, if the user tries to save as a master an object that was received through the mail, but is not currently installed anywhere on the system.

When the master is accepted by the creator, an OMF reference name is established for it and it is placed in the creator's data structure. Whenever the user chooses the Create a New... command and selects a template object icon in the creator's dialog box, all masters associated with that template object are displayed in a Microsoft Windows listbox within the creator's dialog box. The user can then choose to create an empty template or a customized master.

Managing the Masters. With the user given the ability to install template objects and save customized masters numbering in the thousands, it became apparent that the user must be provided with a way to manage all of these objects. For example, the user should be provided with a means to put the most frequently created template objects at the front of the creator's display. This would eliminate the need to scroll the window to find the objects. In addition, having saved customized masters as templates, the user might decide to remove these masters from the creator's display. To solve these problems, a menu choice called Manage Masters was added to the NewWave Office window. When this menu item is selected, the NewWave Office invokes the creator and sends it a MANAGE_MASTERS message. The creator then displays the appropriate dialog box and waits for the user to decide what to do with the template and masters displayed. This dialog box allows the user to select a template object and change its position within the display. It also allows the user to select customized masters in the listbox and press a Delete button which causes the creator to remove its OMF reference to that object.

Opening an Object

When the user opens an object, the NewWave Office must first find out whether that object class supports the open method by calling the OMF function OMF_GetMethod. If the method is supported, then the object is activated through an OMF call and the OPEN message is sent. Once the object is activated, other messages can also be sent. The telescoping effect is one example of the cooperation between the NewWave Office and the object. The telescoping effect is the drawing of a series of rectangle corners to simulate the enlargement of an object from a small icon to a full window. While processing the OPEN message, the object decides where its window should be opened, and sends back that coordinate through the OMF function OMF_Opening. When the NewWave Office receives the information from the OMF, it uses the position of the object's icon in its window as the origin, and performs the telescoping effect using the OMF's library function call NW_TelescopeEffect.

After the telescoping effect is performed, the object's iconic representation in the Office window is grayed out to indicate to the user that the icon is temporarily inactive,

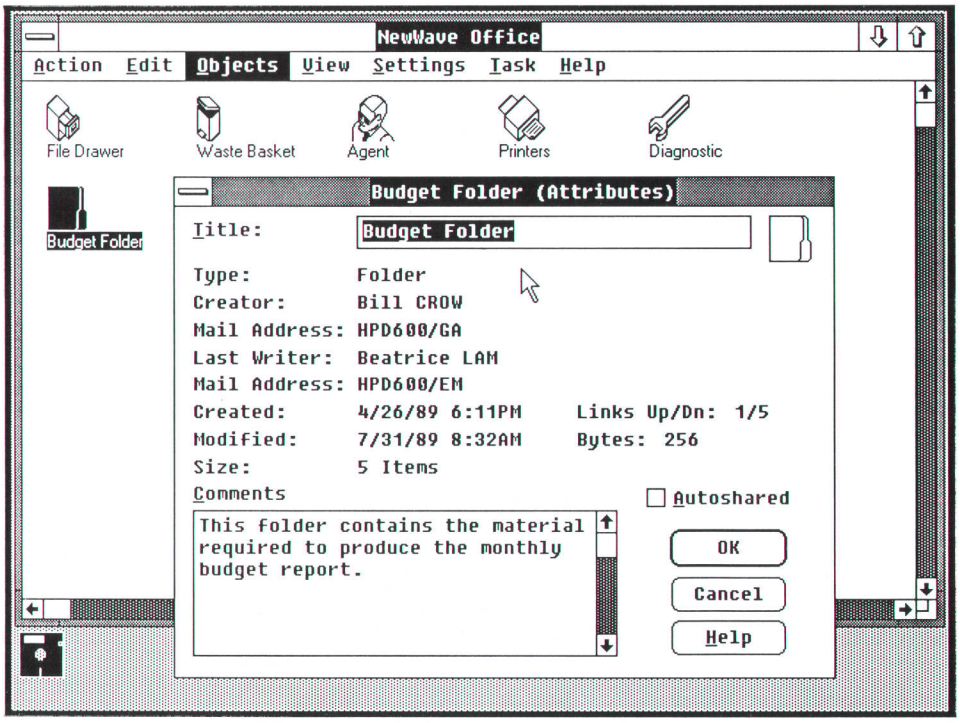


Fig. 7. *Attributes dialog box.*

and that all interaction with that object should be directed to its opened window.

Attributes Dialog Box

The attributes dialog box displays the generic properties of the object selected (see Fig. 7). It provides the user with more details about the object than are provided by the iconic view or the list view. The contents of the attributes box may be useful while the user is working within an object as well. The attributes dialog box is available to NewWave applications through the functions NW_DisplayAttributes and NW_ChangeAttributes in HPNWLIB.EXE.

If the object's properties are changed while the dialog box is displayed, the information will be updated in the dialog box. This is achieved through the OMF message PROP_CHANGE. If an object has requested that property change information be sent to it by setting FLAG_PROPNOTIFICATION with the OMF_ObjectFlag call, it will receive the PROP_CHANGE message from OMF when this happens. To notify the attributes box, the object must then pass the PROP_CHANGE information to a library function, which extracts the useful data from the message and updates the property in the attributes dialog box.

Moving Objects

Objects in the NewWave Office are easily manipulated with the mouse. The user can move the mouse to the icon representing the object, and hold the mouse button down to select the object. By holding the button and moving the mouse at the same time, the user initiates the dragging of the object. As the mouse is being moved, the cursor changes to the shape of a rectangular frame with an arrow in the middle. The original icon representing the object is now grayed, giving the user direct feedback that the object is now in a transient state. Once the user decides where the object should be placed by lifting the mouse button, a whole series of operations and messages will be invoked to decide on the final placement of the object.

Depending on the position where the mouse button goes up, the object can be placed according to various placement algorithms. The final destination may be an empty area in the same window, an area already occupied by an icon, or a different window. Moving to an empty area in the same window simply involves updating the data structure with the new x,y coordinates and drawing the object in the new position. Two other cases—moving on top of an icon, or containment as it is sometimes called, and moving to a different window—are more involved operations.

Containment Move. When the object destination coincides with an existing icon, the containment move operation is initiated. Without waking up the icon object, inquiries are sent to the OMF to find out if the icon object has the appropriate method to accept a new child. The method can be either ADD_CHILD or OMF_INSERT. The ADD_CHILD method allows the insertion of a new object without activating the object itself, and the OMF_INSERT method involves activating the receiving object and sending it the message.

If ADD_CHILD is supported, then the child can be added by OMF using the function OMF_AddChildTo. The file drawer, the wastebasket, and folders are examples of objects that support the ADD_CHILD method. When this method is present, the OMF can examine the property PROP_ADDCHILD to determine if the new child can be added. This property is a structure consisting of the reference name and the number of objects that can still be added. After the new child is added, the structure is updated by the OMF with the appropriate values and is ready for the next OMF_AddChildTo call.

When the container object is opened again, it will discover all the new children added to it while it was inactive. As described earlier, during initialization the object enumerates all of its children through the OMF_EnumChildren call, and the child objects that were added while it was sleeping will then be discovered and added to the object's data file. If new children arrive while the object is opened, the ADD_CHILD message will be sent to the parent directly, and the updating of PROP_ADDCHILD will be maintained by the object itself.

If the ADD_CHILD method is not supported the container object is activated, and an OMF_INSERT message is sent instead. For example, when an object is moved to the printer icon, the process involves waking up the printer and sending it the OMF_INSERT message.

The result of either the OMF_AddChildTo call or the OMF_INSERT message is interpreted by the sender in the same manner. If the return value is TRUE, then the container has successfully accepted the new child, and the sender can now delete its OMF reference to the object and erase it from the screen. If the result is FALSE, then the object is not deleted from the sender's list, and its icon will be repainted again in its position before the move. The return of FALSE does not necessarily indicate a refusal to accept the child. In some cases, the container may accept the object being inserted, and return FALSE to the OMF_INSERT message to restore the object back to the sender. The printer, after adding the object as its child, uses this method to return the object to its original position.

Moving to an Opened Window. When the object is moved outside of its own window, checks are made to ensure that the destination window is a NewWave object. To check whether the destination window will accept the object being moved, the HAS_METHOD message is sent to determine if the OMF_INSERT method is supported. If the return is an OMF value METHOD_PRESENT or NO_METHOD, then the destination is indeed a NewWave object. Other values may indicate that the destination is either a non-NewWave window (e.g., a Microsoft Windows program) or the child window of a NewWave object. The Microsoft Windows function GetParent is used to find the handle of the parent if one exists. This handle is again used in the HAS_METHOD inquiry, and the search continues until the GetParent call returns a NULL handle indicating that there is no parent.

Once a window handle is found that supports the OMF_INSERT method, the object is then sent to the destination window with the screen coordinates of the destination point included in the message. When processing the insert, the receiver of the OMF message decides whether the object should be accepted and then returns a TRUE value if it is accepted and FALSE otherwise. The receiver may also decide that the destination point is occupied by another object, and in that case, will pass the received object to the container as described in the above section.

Moving Multiple Objects. Multiple objects can be selected

by holding down the Shift key while selecting. Dragging any one of the selected objects changes the mouse cursor to a multiple-box frame with an arrow in the middle. The original icons representing the moved objects are grayed to indicate that these objects are now in a transient state. When the cursor has reached the destination, lifting the mouse button brings the objects to the new location. These objects are now positioned as a group at the destination location and staggered down and to the right from each other.

Once again, the OMF_INSERT message is used when moving a group of objects from one window to a different window. Each object from the group is sent to the receiver window using the OMF_INSERT message. The sender is responsible for setting the x,y coordinates in the data structure that accompanies the message for the very first object of the group. The receiver calculates the coordinate position for each succeeding object and writes these values into the insert data structure. These coordinate values are used to position each object in the group.

In all of the cases described above, the sender of the OMF message knows very little about the receiver and vice versa. These two NewWave objects determine whether certain messages should be sent to each other by consulting with the OMF first. If the method in question is supported, then a message is sent to the receiver with the relevant data stored as predefined parameters or data structures. Once the message is received, the receiver can interpret the data or part of it according to its own established protocol. For example, the receiver of an OMF_INSERT message could be a folder in the list view, and in that case the x,y coordinates in an OMF_INSERT message are ignored with no ill effects. The sender, on the other hand, is only interested in the return value from the message to determine whether the message was successfully received. The NewWave object model gives applications a high degree of flexibility to integrate with different object types without the need to understand the particular requirements of these data types.

Conclusions

A few examples have been chosen to illustrate the use of the object model in the implementation of the NewWave Office. By sharing the common code among a set of tools such as the file drawer, the wastebasket, and container objects such as folders, the user is presented with a consistent and familiar user interface. The reusability of some NewWave Office functions in NewWave applications, such as the creator and the attributes dialog box, further enhances the integration of the entire NewWave environment. The NewWave Office not only plays a central role as the first NewWave tool the user encounters, but is also a close collaborator with the OMF to supply essential architectural support to all NewWave applications.

Acknowledgments

The NewWave Office project was first conceived in the Office Productivity Division in Pinewood, England. Paul Fletcher, the project engineer from OPD, laid the foundation for many of the ideas that became the final product, and provided our team with the first prototype of a working Office. We would like to thank Paul for all his efforts and his continual support, and to remind him that his "mojo" is still very much alive and working.

CORRECTION

On page 75 of the April 1989 issue, the equation for the cumulative number of defects found by time t should be:

$$m(t) = a(1 - e^{-(k/a)t}).$$

Trademark Acknowledgments for this Issue

AutoCAD is a U.S. trademark of Autodesk, Inc.
CP/M is a U.S. registered trademark of Digital Research, Inc.
dBase III is a U.S. registered trademark of Ashton-Tate Corp.
Lotus and 1-2-3 are U.S. registered trademarks of Lotus Development Corporation.
Microsoft is a U.S. registered trademark of Microsoft Corp.
MS-DOS is a U.S. registered trademark of Microsoft Corp.
UNIX is a registered trademark of AT&T in the U.S.A. and other countries.
The X Window System is a trademark of the Massachusetts Institute of Technology

Agents and the HP NewWave Application Program Interface

In the NewWave environment, an agent is a software robot that acts as a personal assistant for the user. The agent interacts with the applications through the application program interface.

by Glenn R. Stearns

TO IMPROVE THE PRODUCTIVITY and ease of use of workstation applications, products such as macro processors, script facilities, and integrated intelligent front-end processors are being incorporated into application programs. These allow the machine to do more of the work in performing a task. If these facilities are integrated into each application designed for a software environment, they can be accessed from the integrating environment and operate across all the applications.

One of these facilities, known as an agent, performs tasks on behalf of the user within and across applications. The agent is a software paradigm, like objects (see article, page 9). The agent is added to the system to increase its intelligence. Objects provide the capabilities the agent has at its disposal. The agent uses the objects in an intelligent way to perform work on behalf of the user.

Categories of Agent-Like Products

When current agent-like products are grouped together, three categories emerge, along with an overall pattern (see Fig. 1). The categories range from keystroke processors to integrated intelligence across applications.

The first category is the macro processors. These store keystrokes with record, playback, and edit facilities. Within this category there are macros built into applications, and there are macros that span applications.

The second category is the script processors. These provide a procedural language with structures and verbs based on the products they operate on. For example, communication packages have script facilities to allow automatic logon and data downloading. The script verbs could include Logon or Connect. Within the second category there are scripts built into products, and there are also script facilities across applications.

The third category is the intelligent processors. These have knowledge about the products they operate on. For example, a product may provide a natural language interface for user interaction with an application such as a data base. Within the third category there is integrated intelligence within applications and there is integrated intelligence across applications. For example, the same intelligence may be applied to a data base and a graphics package.

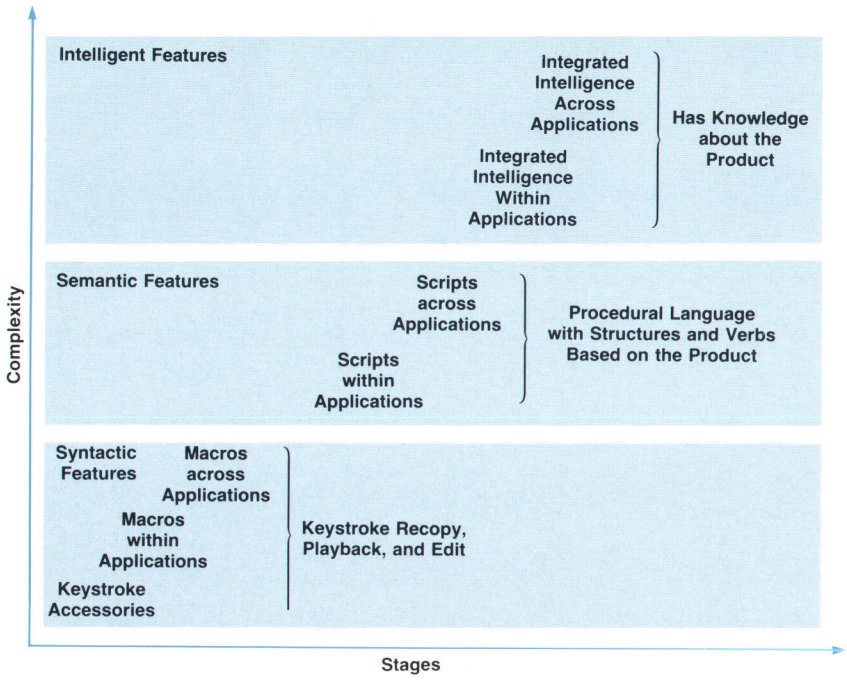


Fig. 1. *Stages of complexity in agent-like products.*

The NewWave agent is designed so that features from all of the above categories can be offered as technology permits.

NewWave Agent Design Criteria

Within the office, users view the computer as a means to accomplish a task. Their goal is not "doing data base" or "doing spreadsheet," but instead, "doing sales analysis" or "doing cost forecasting." Applications, such as the data base and the spreadsheet, are the tools the user employs in the tasks of sales analysis and cost forecasting. A NewWave object is the combination of an application and the related data. Thus, the user interacts with objects to carry out tasks.

The NewWave agent carries out repetitive tasks that the user would otherwise have to perform. It duplicates the user's operation of the system in a repeatable way.

What should the agent do? An agent should improve productivity by being an automated office assistant, and should provide automated testing, demonstrations, and training. The agent should perform user-defined tasks, allow the user to schedule when those tasks get done, do the tasks unattended, and learn from the user. The agent technology should support simulation of user actions, commands, testing of user responses, and the integration of intelligent processes.

Users must tell their agents what to do, of course. If users are to construct automated tasks or modify tasks that are provided for them, the operations must be presented to them in an understandable way. Users should not have to interpret what they have done or what they want their agents to do with the system in terms different from those in which they understand the system. In other words, the agent shouldn't require the user to work with a language that looks like {\F2}+"mydoc"+{CR} instead of SAVE "mydoc".

To allow the agent to interact with the application at the command level, the application must participate in the interaction with the agent. The design requirements to do this must not restrict the application designers' ability to design and structure their applications as they wish.

In designing the agent, we wanted not only to develop the necessary technology, but also to put in place an effective process to bring theoretical work into our design, so we could build a platform for the future as well as deliver a product to our customers today. This process needs to be maintained over an extended period of time to ensure a smooth evolution of the agent facility. For example, the design of the agent and the application program interface needs to support the addition of artificial intelligence (AI) technologies in the future (see "AI Principles in the Design of the NewWave Agent and API" on page 35).

Agent Capabilities

The NewWave agent can be thought of as a personal assistant or software robot. Each workstation has only one agent.

The agent is autonomous in that it can carry out instructions without user intervention. The agent can also make decisions based on the criteria a user gives it.

A user does many repetitive things on a workstation that can be turned over to the agent. These range from sending out reminder notices for meetings to looking up data and making decisions. The kinds of things you would have an agent do are the kinds of things you would expect a personal assistant to do for you with a workstation. You should not expect the agent to do things a personal assistant would not do. For example, the agent is not all-seeing or everywhere at the same time. The agent will only do one task at a time or look at one thing at a time.

The Agent Metaphor

The goal of the agent metaphor in the NewWave environment is to draw upon the conceptual model of a personal assistant that users bring with them when using the system.

Keeping the personal assistant metaphor in mind, the user locates the agent on the desktop in the NewWave

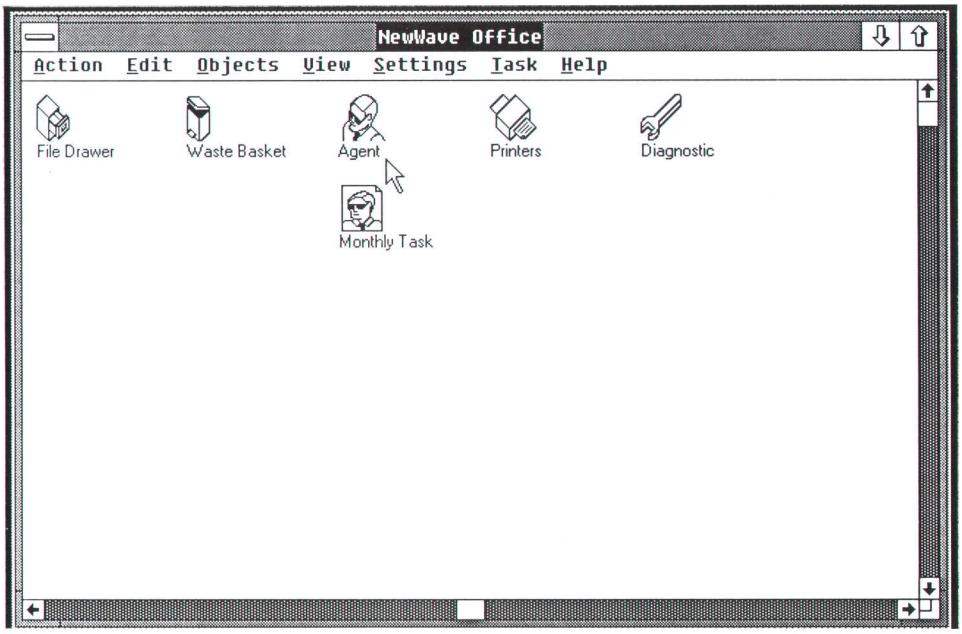


Fig. 2. *The NewWave Office desktop display, showing the icons for the agent and the agent task.*

Office by recognizing the iconic face (Fig. 2). By "tapping the agent on the shoulder" with a double click of the mouse, the user activates the agent and opens a window revealing the agent calendar. The agent calendar is used to define when the agent is to perform a given task.

A task is a series of instructions for the agent to perform and is represented in the NewWave Office as an icon labeled Monthly Task (Fig. 2). The task can be opened by double clicking on it to examine and edit the instructions.

If the user drags the task over to the agent, the agent will perform the instructions included in the task.

Agent Task Templates

When spreadsheets were first introduced, the user needed to learn quite a bit to build one. More complex spreadsheets required sophisticated work and debugging. When template spreadsheets were introduced, the power available to a first-time user was increased. By using a template and making some modifications the user got the desired results with much less work. The template provided the first 80% of the effort and the user worked out the last 20% according to the specific needs of the job.

Agent tasks can be used in this way. Application developers can provide task templates to users of the NewWave environment. Users can then customize the template tasks to fit their specific needs. Users will also be able to make useful template tasks and distribute them within their organizations.

The Task Automation Spectrum

The agent falls within a spectrum of methods of automating tasks, and there is a point where it may be appropriate to develop a specific software application dedicated to a task. For example, if the task is time card entry, the user can perform it manually by filling out a paper time card, bundling it with all of the others, and sending the package via interoffice mail to accounting.

This might be automated with the agent by writing an agent task that brings up a spreadsheet to collect the time card information and add it to a local data base. When all the time cards are entered that agent task would send the data base to accounting for merging with their dedicated payroll data processing system. Here the agent functions as an end-user programming environment, addressing the problem close to the source, and avoiding a data processing systems development effort to write low-level code for the computer.

At some point, however, this solution may not be sufficient, and the user or others may deem that a specific software application needs to be developed.

The Application Program Interface

The key element in the implementation of the NewWave agent is the application program interface (API), which is the interface between the agent and the application. The API provides the necessary and sufficient facilites to provide task automation today and allow the addition of intelligence in the future. It is through this interface that the NewWave help, agent, and computer-based training (CBT) facilities interact with the application.

The API is both an interface and an application architecture. The interface is made up of message definitions, application modes, code macros, and function calls in the C language. The application architecture is the organization of the application to support this interface.

Fig. 2 on page 7 shows where the API fits in the NewWave architecture.

Applications and the API

To support the API, an application must support five categories of interaction with it. These are recording, playback, interrogation, monitoring, and error handling.

Recording support provides the API with the command the application just executed for playback at a later time. The format of the command is binary, is specific to that application, and is not expected to be viewed by a user, although like any machine or intermediate language, it can be examined and understood by the application designer.

Playback support provides the application with commands to be executed as part of an agent task. The commands come from the agent by way of the API.

Interrogation support provides help, agent, CBT, and other applications with the information they need about the application. For example, when the help facility is being used, the application will be interrogated to provide the specific help number for the area the user is pointing to within the application.

During recording and playback, the source application may need to interrogate the destination application that is receiving an object from the source. This is because the source may only know the screen coordinates of the operations, but needs to record or playback a meaningful command. This allows the recording of a command like MOVE_TO FOLDER "Sam" WITHIN FOLDER "Mary" instead of MOVE_TO 123,346.* It is the destination application object that knows that 123,346 is FOLDER "Sam".

Monitor support allows CBT to intercept an application command before it is executed, and provide corrective tuto-

*123,346 is a screen coordinate pair.

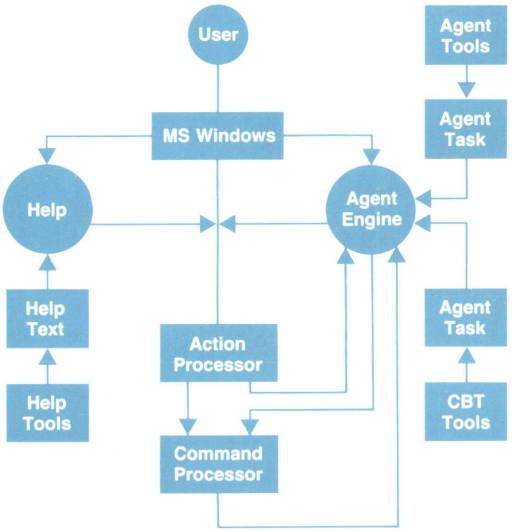


Fig. 3. *The main feature of the application architecture is the splitting of the application code into an action processor and a command processor.*

rials. The CBT lesson can elect either to let the command go through to the application for execution, or to ignore the command so the application does not execute it. See the article on page 48 for more information about CBT.

Error handling support provides the agent, via the API, with any error conditions that occur during execution of a command. These error conditions can be trapped within an agent task so error recovery can be performed.

Application Architecture

To support these interactions, the application must be designed to do so. This design we call the application architecture. The major design concept is the splitting of the application between the portion that interacts with the syntactic user actions (keystrokes and mouse moves) and the portion that interacts with the semantic commands that result from one or more user actions. The two portions of code are called the *action processor* and the *command processor* (see Fig. 3). The agent, the help facility, and the CBT interface with these in a predefined way.

The agent must be able to record commands after they are generated and play them back to the command processor, as well as monitor the commands before they are executed. Therefore, the agent interfaces with the application between the action processor and the command processor. For support of user action monitoring and playback by CBT, the agent also interfaces before the action processor through the application program interface. Help facility interrogation of the application also occurs before the action processor, also through the application program interface. The agent engine provides the execution of agent tasks for task automation and CBT.

On a more detailed architectural level (Fig. 4), the application can be thought of as organized into processors, which the application writer designs and constructs, and components, which are supplied as code fragments to the developer to be placed within the application. It is the relationship of these processors and components that allows the API to interface to the application.

Extensible Task Language

The results of the application architecture can be seen by looking at an example of recording a command, viewing it, and playing it back (see Fig. 5). For more information on the NewWave task language, see the article on page 38.

An agent command takes on several forms as it moves through this process. There is the form in which the user sees it—for example, SELECT "XYZ". This is the *task language form*. There is the form in which the command is stored for playback via the agent. This is the *P-code form*. There is the form of the command when it is transferred from the agent to the application. This is the *external form*. There is the form the application uses during execution within the application. This is the *internal form*.

Assume that the application is in record mode and the user operates on it with several user actions. These are analyzed by the action processor and a command is generated.

The command may be the selection of an object on the screen. Because the objects are kept in a list within the application's data structure, the command's internal form

AI Principles in the Design of the NewWave Agent and API

The field of artificial intelligenge, or AI, is made up of many areas of concentration, including expert systems, natural language, planning, cognitive psychology, and robotics. In the design of the agent and the application program interface (API) for the HP NewWave environment, the area of robotics was drawn upon, with the agent modeled as a "software robot."

The basic agent architecture as depicted in AI research, is made up of three components: the agent, the world, and the knowledge base containing the agent's knowledge about the world.[1] There are two main relationships: the relationship between the agent and its knowledge base and the relationship between the agent and the world. Much work has been done on the relationship of the agent and its knowledge base, but less has been done on the relationship of the agent and the world. In developing the NewWave environment we have concentrated on the relationship of the agent to the world of NewWave objects.

If the agent is a software robot, then by analogy, it should interface to the software world much like a hardware robot interfaces to the physical world. What is necessary and sufficient for a hardware robot to interface to the physical world? First of all, a hardware robot needs effectors. These are the arms that cause change in the physical world. It also needs feedback to determine what its effectors are doing and to detect that an attempted movement has encountered an obstruction. Finally, the robot must have passive sensors to view the physical world and provide the needed feedback.

The software equivalent of effectors is software commands to applications. Feedback is the return of application error conditions and status to the agent. For passive sensors, the agent interrogates the application's data. These are the robotics principles designed into the NewWave agent and API. Commands, error conditions, and interrogation are the necessary and sufficient faculties for a software robot to interface to its software world.

By providing the design guidelines and environment to support these faculties, the NewWave environment facilitates the building of software to support a software robot. The NewWave environment does not presently supply the robot—the agent—with artificial intelligence, that is, a knowledge base and inference capability. Hence the agent needs to be programmed just as industrial robots are. This can be done by leading the robot through the motions of the task, and in the case of the NewWave agent this is the record mode. Industrial robots can also be programmed using an appropriate procedural language, and in the NewWave environment this is the agent task language.

However, like industrial robots, the NewWave agent can and undoubtedly will become more intelligent. The design of the agent and the API makes this straightforward by allowing intelligence to be added to the agent without changing the interface of the agent to the applications. Intelligence can be added to the agent in a plug-compatible way, so that different development organizations and companies can add their specific types of intelligence to the agent to meet their needs.

Therefore, while the NewWave agent presently has no artificial intelligence, the use of AI principles in the design of the agent and the API will make it easy to add intelligence as technology permits.

Reference

1. M. Genesereth, *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Publishers, Inc., 1987.

may specify that element 1 be selected from the list. SELECT would be represented by a number. The command would read 6, 37, ptr[x], where the number 6 represents the length of the command and 37 the command number. Element 1 could be represented by a pointer to an element x. The command processor executes the command, and because the application is in record mode, the command is passed to the processor that translates it to the external form. This processor is written by the application designer.

The recording of the selection of element 1 would not be of value to the application, because the object may be moved around on the screen and become element 7, while element 1 may be replaced with another object. The internal form is what best serves the application designer's needs.

Translated to eternal form, 6, 37, ptr[x] may become 8, 37, XYZ. In this form, the command can be preserved for later playback.

The application transfers the external form of the command to the agent via the API and the agent attaches a P-code to it, resulting in the command being in the P-code form. The agent has many P-codes for arithmetic and logical operations and for flow control that are executed by the agent without interaction with the application. One of the P-codes identifies that the parameters of the P-code form are to be sent to the application in external form.

With the command P-code prefix, the command is sent to the agent task for recording. The agent task passes it to the class independent recorder. There are two categories of commands: *class independent* commands like IF, WHILE, and so on, and *class dependent* commands, which are specific to an application, like the SELECT command. If this were a class independent command, the class independent recorder would produce the user-viewable task language form. However, in this case it is a class dependent command, and it is passed off to the class dependent recorder for the specific application the agent is interacting with for this command. The class dependent recorder produces the user viewable command SELECT "XYZ". This command is stored as a line of ASCII task language within the agent task.

The user could be watching the recording as it is going on and would see the line of text displayed within the agent task's open window. The user can modify any task language that is viewable. When recording is completed the resulting task is compiled to be performed at a later time.

If the user modifies the agent task language, the process is reversed. The task language form is compiled through the class independent parser for commands like IF, WHILE, and so on, and through the class dependent parser written for the application for application-specific commands. The results of compilation are stored within the agent task in P-code form.

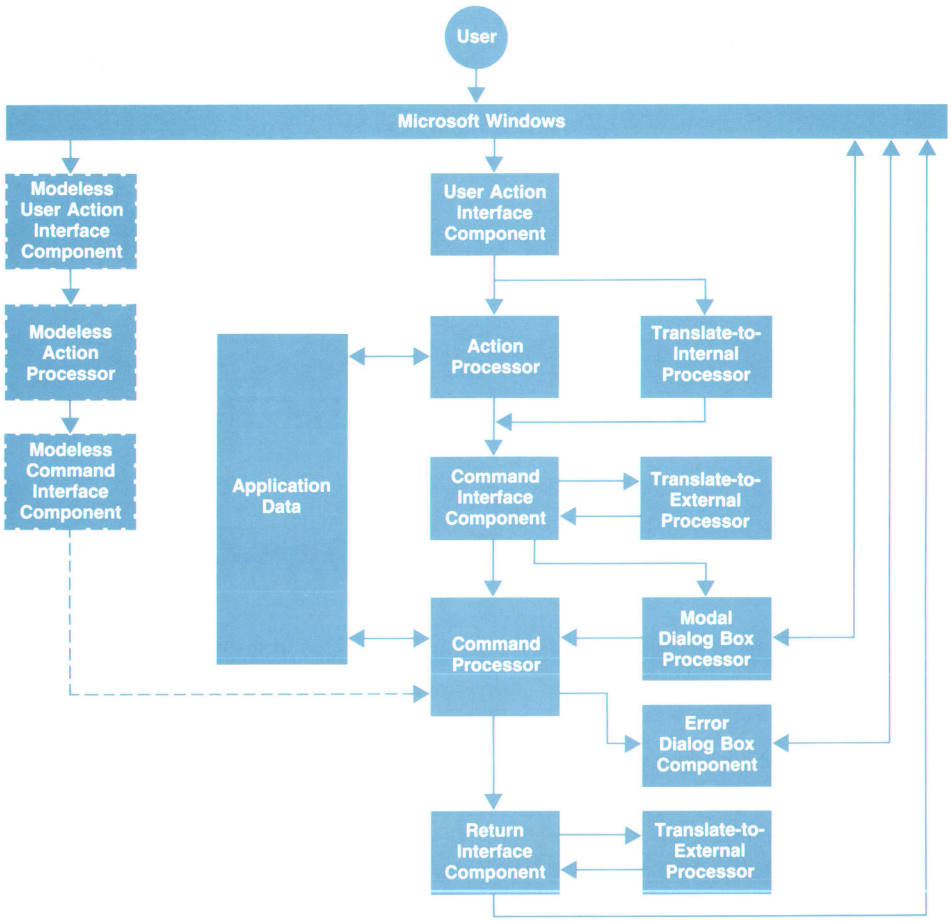


Fig. 4. *Detailed block diagram of the architecture of a NewWave application.*

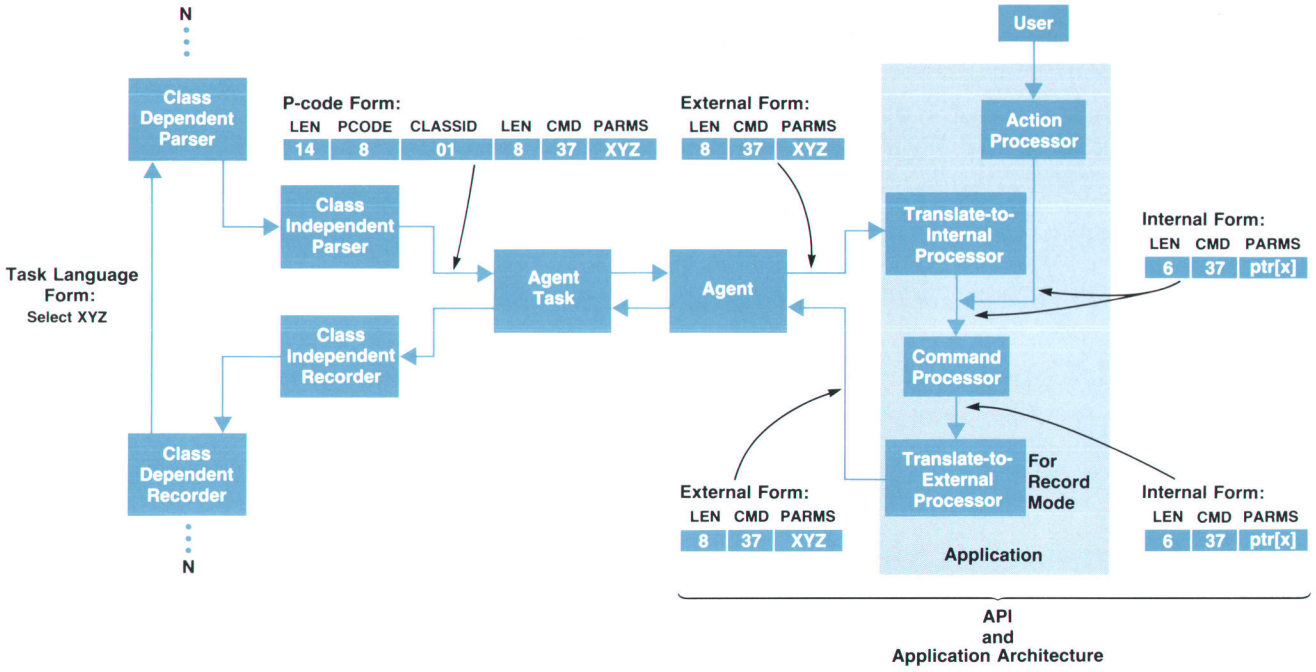


Fig. 5. *The task language, the application, and the application program interface use various command forms to support the agent.*

During playback the agent reads the P-codes from the agent task object and passes the external form of the commands to the application. P-codes, like IF, WHILE, and so on, are executed within the agent engine and are not passed to the application.

When the application receives the command in external form via the API, the application passes it to the processor that translates it to internal form. This processor is written by the application designer and produces the internal form of the command for execution by the application's command processor.

The cycle is now complete. Combinations of record, modify, edit, and playback can be performed and the agent will reliably maintain the proper command forms.

Acknowledgments

The agent concept would not have been investigated for the NewWave environment without the sponsorship of Ian Fuller and Larry Lorren. Their continuing support contributes to the ongoing development of the agent concept. I would also like to recognize John Alburger, Tom Anderson, Barbara Baill, Mark Barbary, Barbara Packard, Lynn Rosener, Phil Sakakihara, Pete Showman, Tom Watson, and Eugene Wong for their support during the time that the agent concept was being investigated and proposed as a NewWave project. This was a critical time that required endorsement of an idea that, at the time, was challenging the way we built software. The agent concept would not be part of the delivered product without the the contributions of the agent team: Martin Chaney, Paula Dieli, Ania Dilmaghani, David Fogelsong, Brian Harrison, Tony Martin, Bob Mayer, Barbara Packard, Vicky Spilman, Tom Watson, Karen Wales, Jonathan Weiner, Chuck Whelan, and Gary Visser. New architecture ideas are not developed in a vacuum, and special recognition is due the NewWave Office team of Bill Crow, Tony Day, Andy Dysart, Scott Hanson, Bea Lam, and Yitzchak Ehrlich for working with us on implementing the API in its earliest forms.

An Extensible Agent Task Language

With this language, users of the HP NewWave environment can create scripts to direct their NewWave agent to perform tasks for them. The language is designed for both novice and knowledgeable users.

by Barbara B. Packard and Charles H. Whelan

THE AGENT TASK LANGUAGE of the HP NewWave environment is a set of procedural commands that provide users access to the task automation functions of the NewWave environment. Scripts can be written to create, delete, modify, and otherwise manipulate NewWave objects. The scripts are processed by an interpretive engine, which is part of the agent object. More information on the interaction of the agent, the task language, and the application can be found in the article on page 32.

In the NewWave environment, each task is a separate object with associated data files. Tasks function across and within object classes and are supported by all NewWave applications. Upon opening a task, the user sees the contents of the file containing the task language commands, available for editing and compilation. When the user drags a task to the agent icon, the associated binary P-code file is executed. Task language commands have a verb/object syntax:

⟨command keyword⟩ [parameter]. . .

The parameter of a command may be either a keyword or a literal. Commands are line-oriented, but a continuation character is available to extend a command across the line boundary. A primary concern in the language definition was the mapping of the interactive user interface to the task language commands. To make the scripts as readable as possible, we wanted to have the command keywords reflect user actions. For example, if an action is accomplished interactively by clicking on a menu item such as CUT, the corresponding task language command will contain that menu item as its command keyword verb. The parameter type is command dependent, but numeric, string, and keyword command parameters can be used.

User Requirements

Agent tasks will be created and executed by users whose expertise varies widely. The casual NewWave user will record a few actions within an object and save them as a task, which is executed to repeat the actions. The power user will construct complicated automated tasks, frequently for other users, that execute for a considerable time without user intervention.

The novice, using the task language as a macro recorder, quite possibly may never look at the task language form of the task. This user will require ease of use and high performance. The power user will demand a language with at least the power of command languages in existing applications such as Excel or DBase III Plus.®

Our chosen model for the task language is the power user. The language is appropriate for constructing large automated tasks involving several applications. We have provided a conversational window facility, designed by the task writer and controlled by the task language script, which enables the task to receive user input and display information. Other features include variables, functions, task procedures, control statements such as conditionals and loops, and numeric, string, and logical expressions. A command parameter defined as a literal may also be an expression of the same type.

We expect that in time many casual users will move toward the power user model. The language should be designed to facilitate this. Toward this end, the agent task recorder facility has a built-in watch feature. The user can see the task language command that was recorded as a result of an action. Recorded tasks do not contain the advanced programming features listed above, but the relationship of the user's interactive actions to the task language commands will be apparent from the syntax of the command. In particular, the syntax of task language commands is meaningful enough to serve as a learning aid to users who wish to explore the more advanced features of NewWave agent tasks. This is another reason for the close mapping of the command keywords to the interactive user actions.

System Requirements

The system requirements for automating a task that spans applications have a somewhat different perspective. A task language statement is either a control statement (examples include variable assignment, procedure call, loops) or an action command (such as CLOSE, CUT, PASTE) to a particular object. Control statements are independent of the current active object and can be executed by the agent interpretive engine, but action commands are sent to an object of a particular application class and executed by it. Commands are not identical across applications; many are class-specific. For example, most applications support some form of

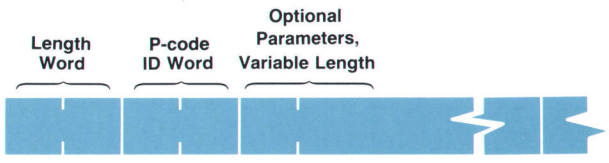


Fig. 1. *Binary P-code record format.*

SELECT. But the object of the selection, which translates to the parameter of the task language command, will vary widely depending on the object class. In a document one would SELECT a range of text, in a spreadsheet a cell or range of cells. However, in the NewWave Office window the selection is an icon, that is, another object with a class and title. The NewWave open architecture specification mandates the dynamic installation and removal of application classes and leads to a different configuration on each system. Task language commands for a NewWave application written by an independent software vendor must also be supported.

It is impossible for the agent engine to keep track of the command set and syntax supported by each application currently active on the system. The agent engine should not interpret the contents of a command it sends to an application in playback. It is equally impractical to have each application class parse its commands at execution time, returning similar syntax error messages, or handling variables or expressions as parameters.

The solution is a task language parser module and recorder template for each application class. The parser converts ASCII task language commands to the external command form recognized by the application. The recorder template converts the external command to ASCII task language commands during the task recording. These are installed into the task automation process when the application is installed into the NewWave environment. As applications are added to and removed from the system, the set of task language commands accepted by the compiler and created by the recorder is customized accordingly.

Application Developer Assistance

Because developers of NewWave applications need to provide parser and recorder modules for task automation, we supply tools and guidelines to make their job as simple as possible. We separate out the components that are common to all applications and provide code for these in libraries, and we provide source templates for typical semantic routines. Since we wish to have the task language commands appear as one programming language to the user, we provide guidelines for commands and examples of appropriate syntax that are the same or similar across applications.

The Task Language Compiler

Our first design decision was to compile task language scripts to a binary P-code format for execution rather than interpreting the ASCII commands at run time. There were several reasons for this:

- The binary format is more compact, particularly for long tasks.
- A standardized binary format is more suitable for execution by applications in the MS Windows environment.
- Syntax and other obvious errors can be flagged and fixed at compile time.
- Nonsequential instructions such as loops and procedure calls can be handled efficiently.
- Functions, variables, and expressions can be preprocessed and handled in a standard manner.

As a result, the task language compiler is a two-pass compiler. The first pass follows the general compiler model of scanner, parser, and semantics. It receives as input the ASCII task language script, parses it, and generates binary P-code records which are written to a temporary file. The second pass fixes instructions that reference addresses that were unknown when the P-code was initially generated.

Object File Format

Successful compilation of a task creates a binary object file. An object file consists of two main parts: a fixed-length header record and the binary P-code records which will be executed by the agent interpretive engine.

The header record contains the version ID of the compiler as well as information such as the number of variables, conversational windows, and pages of code in the task.

The code section of the object file consists of the variable-length, binary P-code records which are executed at run time by the agent engine. Many P-code instructions are similar to high-level assembly language. Pointers to locations in the code are maintained as addresses. Addresses consist of a page number and an offset into that page, thus identifying the start of an instruction. Page size is fixed. P-code instructions do not cross page boundaries; however, a continuation P-code is available.

The P-Code Record

The agent interpretive engine performs a task by fetching and executing the P-code instructions. The generic record format is shown in Fig. 1. The length field contains the number of bytes in the record, including the length word. A record with no parameters will have a length of 4. The P-code ID is the numeric opcode of the instruction. The parameters are any parameters the instruction requires. The type and length are instruction dependent. Parameters of type string are null-terminated, which is indicated by the string \0.

The Command P-Code

As mentioned earlier, most P-code instructions result in an action command sent to a particular application object. Fig. 2 illustrates the P-code format for a command. The parameters of the P-code, except for the integer class ID word, make up the external command form, which is sent to the application. The class ID parameter is an integer indicating the class of object that recognizes this command. It is task dependent. The command length is an integer containing the total length of the length word, the command ID word, and the parameters. The command ID is set by the application. The parameters are of variable length and type, and are command dependent.

At run time, the agent engine strips the first three words

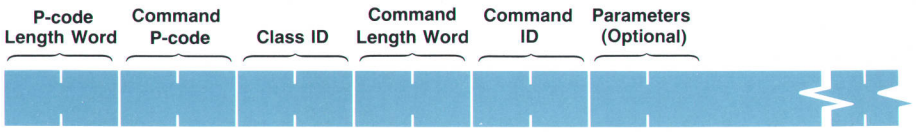


Fig. 2. *The P-code format for a command.*

A NewWave Task Language Example

Mary Smith starts the following task every Friday evening as she leaves work. The task cleans up her NewWave Office window by tidying up the past week's work. It then makes preparations for jobs that Mary will begin when she arrives on Monday morning.

```
TASK
FOCUS ON OFFICE
```

Start by moving all data objects into a folder for the week and putting that folder in the file drawer.

```
CREATE_A_NEW
FOCUS CREATOR
CREATE FOLDER TITLE "Week Ended 5/29"
FOCUS OFFICE
SELECT_ALL
DISJOINT_SELECT FOLDER "Week Ended 5/29"
MOVE_TO FOLDER "Week Ended 5/29"        'Moves all selected data
                                        'objects into the folder
SELECT FOLDER "Week Ended 5/29"
MOVE_TO FILE_DRAWER
```

Next, the task empties the waste basket.

```
SELECT WASTE_BASKET
OPEN
FOCUS ON WASTE_BASKET
EMPTY
CLOSE
FOCUS ON NEWWAVE_OFFICE
```

Now, the task creates a spreadsheet, which Mary will use on Monday morning to prepare this month's profit and loss statement. The spreadsheet will be a copy of a customized master complete with prepared formulas and formats.

```
CREATE_A_NEW
FOCUS CREATOR
CREATE SPREADSHEET FROM MASTER CALLED "Monthly Profit And
Loss" TITLED "May Profit & Loss"
```

Next, the task will mail some memos that Mary wrote this afternoon. Since it is now late on Friday evening, the mail system will be more responsive.

```
SELECT FILE_DRAWER
OPEN
FOCUS ON FILE_DRAWER
SELECT FOLDER "memos"
OPEN
SELECT DOCUMENT "January Sales analysis"
DISJOINT_SELECT TEXT_NOTE "Quick note about meeting Tuesday"
DISJOINT_SELECT DOCUMENT "Next month's forecasts"
MOVE_TO ENVELOPE "outgoing"
SELECT ENVELOPE "outgoing"
SEND_TO_MAIL_ROOM
CLOSE

FOCUS ON NEWWAVE_OFFICE
```

Finally, the task will lock Mary's display. Her NewWave system will be locked until she unlocks it on Monday by giving her password.

```
LOCK_DISPLAY

END
ENDTASK
```

and sends the remainder, the external command, to the application. The agent engine requires the length word; the remainder of the structure is designed by the application. However, applications are strongly urged to use the format illustrated.

Run-Time Environment

The agent interpretive engine is implemented as a simple stack machine. Variable assignments, function and procedure calls, and expression evaluations are all stack operations. When a task starts up, the agent initializes its data structures using information in the task header record. It then makes an MS Windows intrinsic call to receive an MS Windows TIMER message at regular intervals. Each TIMER triggers a P-code fetch and execution. The agent relinquishes control between instructions, thus allowing tasks to conform to the same execution guidelines as other NewWave objects. P-codes are fetched from the current page, which is retained in memory. Pages are procured as needed. If the P-code is a command, the agent checks the class ID to determine if the instruction class matches the class of the object that currently has the focus. If so, it posts an API_PLAYBACK_MSG to the object with the command as a parameter. No more P-code instructions are executed until the agent receives an API_RETURN_MSG.

The Task Language Parsers

To facilitate the modularization and customization of the task language, we designed a system of multiple parsers. The main compiler contains two parsers: the top level or *class independent* parser, and the *expression* parser, which handles functions and expressions of numeric, string, and logical type. Each application also has a parser module, which parses its *class dependent* task language commands. This module also includes semantic routines which convert the parsed command to the external command form. The parser modules are in the form of MS Windows dynamic libraries and are accessed from the class independent parser through the MS Windows LoadLibrary intrinsic. The application's installation file identifies the library file, the application class name, and the names of its parse routines by adding them to its OMF property list as a property PROP_AGENTTASKINFO. The task language compiler enumerates all applications with this property. It is then aware of all available classes of task language commands. Again, this can be different for each system configuration.

However, the compiler loads only the libraries of the classes requested by the task script.

Parser Components

The following sections describe briefly the various components of the parsers. Fig. 3 shows a data flow diagram of their interaction.

Parser Routines. The current parser modules have been created using the yacc parser generator. yacc was developed at AT&T Bell Laboratories. There is nothing to preclude a developer's substituting a customized parse routine for a yacc-generated one.

Keyword File. The recommended class dependent parser model stores its command keywords in a file which is read into a table during the initialization process. The token number of each keyword depends on its position in the file. This permits a certain amount of command localization without reconstructing the parser.

Scanner Routine. The scanner was developed in-house at HP and provides the only access to the task language source file. All parser modules must call it for token input. The scanner returns token types and indexes to appropriate tables where the values are stored. If a parser module uses a different token representation, it can modify its scanner to retrieve the value at this point and continue processing.

Expression Parser. The expression parser is available to the class independent and class dependent parsers. It is activated by a semantic call during the parsing of a command. It processes tokens until it finds one that is not part of the expression. The associated semantic routines generate P-codes to evaluate the expression and place the result on the engine run-time stack. The expression parser then sets the state of the scanner so that the next token request (by a command parser) will return a token type expression, which will satisfy the conditions of the parse. There is no requirement that class dependent parsers use the expression parser.

Semantic Routines. Since the structure of the external command form is known only to the relevant application, the semantic routines must be the responsibility of the application developer. However, we have provided a library of routines to perform functions such as initialization, buffer management, and cleanup. Also, there are routines that handle the semantic processing of expressions when they occur as command parameters. Use of this library will greatly simplify the implementation of the semantics. The output of the semantic routines is returned to the compiler in a buffer and then written to the object file.

The FOCUS Command

The compiler directs the source processing to the appropriate parser through the FOCUS command. This command needs additional discussion since it results in both compile-time and run-time actions. The syntax is

FOCUS [[ON] ⟨classname⟩ "⟨title string⟩"]
[OFF]

where classname is the name of the class of object (for example, DOCUMENT or FOLDER) as recognized by the task language parsers, and title string is the title of the specific object referenced. At installation time this classname is added to the OMF PROP_AGENTTASKINFO property of the class. When a task is compiled, the compiler adds the classnames to its list of keywords recognized by the scanner, and through the scanner by the class dependent parsers as well.

When a task is executed, the majority of the commands will result in the agent's sending a message to the object that currently has the focus. The parameters of this message make up a command that will direct the object to change its state. At run time, the FOCUS command tells the agent which object is the target of subsequent command messages. At compile time it has another role. It controls selection of the class parser that will parse class dependent commands and generate the external command. Commands are compiled sequentially in the order received. However, the order in which commands are executed at run time will seldom, if ever, be completely sequential. The inclusion of conditional execution (IF, WHILE), jumps (GOTO), procedure execution (DO), or user variables in a task virtually guarantees that there is no way to make a determination at compile time which object will have the focus at run time. The FOCUS command sets a compile-time focus. In effect, it determines which class dependent parser will parse the commands following it. The command

```
FOCUS DOCUMENT "Orders Report"
```

will cause all class dependent commands to be parsed by the document parser until another FOCUS command is encountered. If the OFF parameter is used, only class independent commands will be accepted by the parsers until another FOCUS statement is encountered. The main effect of this command is to reduce compilation time, since a syntax error returned by a class dependent parser will cause the command to be reprocessed by the class independent parser.

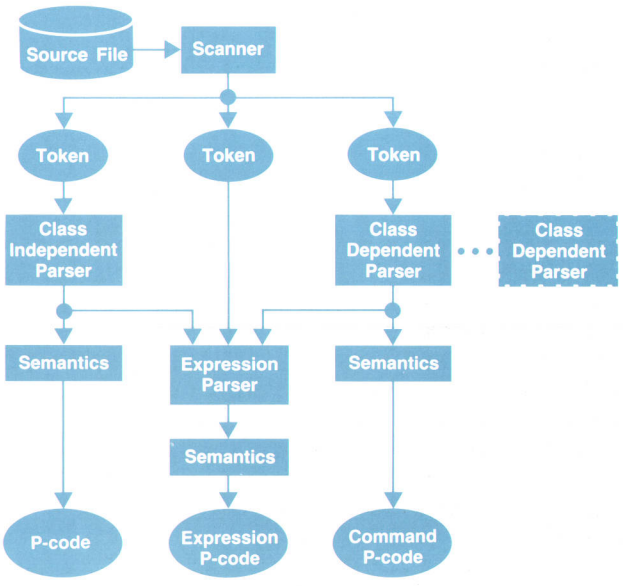


Fig. 3. *Parser data flow.*

The Task Language Recorders

Task recording provides the ability to monitor run-time events and recompose them to produce a reusable task in the ASCII task language format. The focal point of the recording process is the class independent recorder. This module is an MS Windows dynamic library which is loaded only during a recording session. It receives all external commands from the agent while recording is active.

The recorder first determines if the received command is specific to a class. If it is not, the command is immediately converted to its ASCII task language form. If the command is class-specific, the library will either provide default dependent recording or will pass the external command to a separate class dependent recorder module. In either case, the completed task language text is used to build a source file of ASCII task commands.

Default Dependent Recording

The majority of class-specific external commands are handled wholly within the class independent recorder. This module uses ASCII recorder template files to provide the necessary information to do default dependent recording. These files provide formatting information so that a multiplicity of external commands can be recomposed into task language without the need of invoking class-specific executable code.

Recorder template formatting strings are patterned after the C programming language's print control strings. They describe how a given external command's parameters are to be interpreted and formatted into compilable task language text. They can include information on the order of parameters in the external command, the size of each parameter, the data type of the parameter, and how the parameter is to be formatted in the task language line. Templates also support arrays of parameters, enumerated parameters, optional parameters, and optional fields in the task language form. Comment fields, ignored by the recorder but useful for documentation, may be included as well.

Template file information for a particular class is read into memory when a FOCUS command for that class is first received during a recording session. As external commands are passed to the recorder at run time, they are then formatted into task language.

The example below illustrates a template and a command definition from the NewWave Office recorder template file.

Template Formatting String:

```
"%v %1d COPIES TO DEVICE %2s"   ; 13th template.
```

Command Definition:

```
103      13 "LIST"
```

The command definition specifies that when external command 103 is received (and NewWave Office has the focus) the 13th template in the file is to be used with the verb LIST. The template definition specifies that the first parameter in the external command is a decimal integer and the second parameter is a string.

The external command form of the example is shown in Fig. 4. From the external command shown, default dependent recording will produce

```
LIST 2 COPIES TO DEVICE "LaserJet"
```

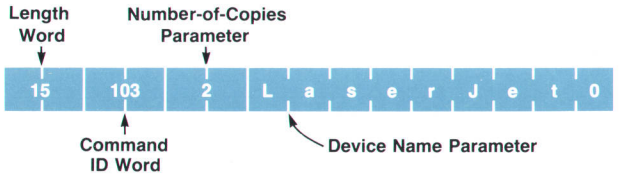


Fig. 4. LIST *external command format.*

Class Dependent Recorders

If there are cases that cannot be handled by template files, an application can provide its own class dependent recorder. The class independent recorder will pass the external commands that it cannot handle by default recording to the class dependent recorder for the class that currently has the focus.

Extensibility

All recorders including the class independent recorder are written as MS Windows dynamic libraries that are loaded only when needed with the MS Windows intrinsic LoadLibrary. All recorders must also support a common programmatic interface, so that the interactions between the class independent recorder and any class dependent recorder are identical.

The developer of a new application can implement recording by producing the ASCII recorder template file and, if necessary, by developing the application's own separately linked dynamic library. The file names are declared in the PROP_AGENTTASK_INFO property in the application's installation file. Recording is activated when the running application gets the focus during a recording session.

Acknowledgments

The agent task language project is indebted to many individuals. However, some should be specially acknowledged. Glenn Stearns was the creator of the agent project and has been the driving force behind it ever since. His enthusiasm, persistence, and constant flow of ideas clearly "made it happen." Tom Watson designed and implemented the prototype for the class independent parser and wrote the initial definition of the task language commands. Many of his ideas found their way into the final product. Tony Day was a major contributor to the definition of the command set of the NewWave Office class dependent parser. The success of this first parser helped establish viability. Many thanks to the agent team and the task language review committee for the hours spent reviewing task language specifications, and to project manager Ian Fuller and section manager Larry Lorren for their continual support of this and other NewWave projects.

The HP NewWave Environment Help Facility

The NewWave environment provides a common, context sensitive, intuitive, unobtrusive help facility for NewWave applications.

by Vicky Spilman and Eugene J. Wong

DURING THE INVESTIGATION AND DESIGN phases of the help facility for the HP NewWave environment, the development team followed objectives passed down from the system level. Among these were ease of use, emphasis on the tasks instead of the tools, and consistent user interfaces. The decision had already been made to include a help facility in all components of the NewWave environment. The decision was easy because customers expect some form of on-line help and there was a general desire to reduce the need for users to refer to the manuals. Other objectives were also added specifically for the help facility, but all of these objectives can be summarized by four descriptors: common facility, context sensitive, intuitive user interface, and unobtrusive.

Common Facility

Since every NewWave component has to supply help, it makes sense for help to be implemented as one facility that can be called by each component. Help runs as a separate program in the Microsoft Windows environment, like the other NewWave architectural components. However, help does not depend upon the NewWave architecture since it has to provide services for the NewWave Office, NewWave formatter, NewWave agents, other architectural components, and NewWave applications. Since the help facility is concentrated in one program, it can afford to provide additional features that would have been prohibitive to implement in multiple components. The character display formats, related topics buttons, window movement, and other functions are examples of features that would have been duplicated.

The use of a shared program to provide help also guarantees that the help user interface is identical for all NewWave components. All displays look alike and all user interface features work identically across NewWave applications. A user can ask for help in the same way, regardless of whether the request is for a reminder of the purpose of an icon or for information on how to perform a specific operation on some data.

A common help facility also gives the NewWave developer several advantages. One obvious advantage is that the developer of an application does not have to develop the help facility. Also, maintenance of the application is simplified, since it uses the existing help facility. The help text can be written as a separate activity from the application code development, further reducing the workload on the developer. Help text maintenance is also easier since this text is separated from the other text materials in the application code.

Context Sensitive Help

For the help facility to be useful, it has to meet the expectations of users, which means that it should be context sensitive. When a user asks for help, the resulting information applies to the specific situation at that moment, instead of to a general situation, which might not provide any meaningful information to a user at all. For example, a user who needs to know how to use a text input box in the middle of a screen is not interested in finding out that the screen is used to communicate information and to get many

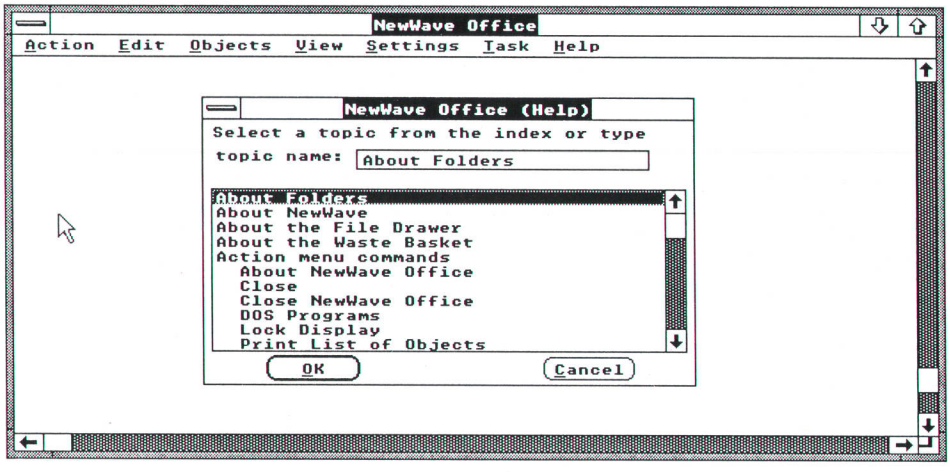


Fig. 1. *An example of the help index.*

types of input. The user would rather have—and gets—instructions on what a correct entry might look like or what choices are expected.

Intuitive User Interface

In most cases, a user can look at the screen and figure out how to use the help facility without a lot of training or searching through the manuals. Thus one can get help immediately when it is most needed, when a small hint or short explanation will allow the task to be completed without a long delay or distraction. Help can be started from a menu, from a function key, or from a pushbutton. Index items can be chosen by mouse scrolling, keyboard scrolling, or typing. The help user interface allows all common modes of operation so that its use will seem intuitive to as many users as possible.

Unobtrusive

Help has to be able to meet the needs of users without becoming another problem. Therefore, it allows flexibility and yet remains efficient in execution and use of memory. The help window is large enough so there is room for explanations, but users can still see their original tasks. If it is necessary for them to see more of the previous window, they can move the help window out of the way or make it disappear until they want to make it reappear. If they move the help window aside, they can continue working on the previous task and refer back to the information given in the help text window.

Starting Help

To start help, the user can select one of two menu items from the help pull-down menu. The help pull-down menu for NewWave objects contains two items: Help Index and Screen/Menu Help. The menu items activate the help index and screen/menu help mode, respectively. The user can make the selection by using the mouse or by using accelerators (keyboard interface). The accelerator for gaining access to the help index is **f1**.

The user can also start help by selecting the help pushbutton from a dialog box. When the help pushbutton is selected, the help text window containing information pertinent to the dialog box is displayed.

Screen/Menu Help

As mentioned above, one of the major objectives for help is to provide context sensitivity. Context sensitivity is best illustrated by screen/menu help mode, which is also called ? mode. The user selects screen/menu help from the help pull-down menu and the cursor changes to a question mark shape, indicating that the user is in a special help mode. The user can move the question mark cursor around on the screen and click the button on the mouse when the cursor is on an area of interest. The verbal equivalent of this action is to ask the question "What is this?" while pointing at the referenced area.

Screen/menu help allows the user to get help on anything in the application window, which might include pull-down menus, icons, and fields. When ? mode is activated, the Screen/Menu Help item in the help pull-down menu changes to Cancel Help, which allows the user to exit the mode without having to select a help topic.

When the user activates screen/menu help and selects an item, the help window displays information about that item instead of executing that selection. After the help window is displayed, the mouse cursor and the help pull-down menu are restored to the previous state. ? mode is active for one selection at a time.

Index

The index will be displayed when the user selects the Help Index item from the pull-down menu, or activates the Index pushbutton from the help text window. The main purpose of the index is to list all available help topics and allow the user to select a topic. The index facilitates a quick search for a particular topic or allows the user to read all topics. An example of the index is shown in Fig. 1.

All topics are listed in a standard listbox with a vertical scroll bar. With one exception, all topics are listed alphabetically. The one exception is the first index entry, which at the help text writer's discretion, may be a special topic that should be pointed out to the user. Within the listbox, the user can scroll through the entire index and select a topic, which is done by double clicking with the mouse on a topic, or by selecting the topic and then the OK pushbutton.

Another option for searching for a topic is to use the editbox, which appears above the listbox. This allows quick

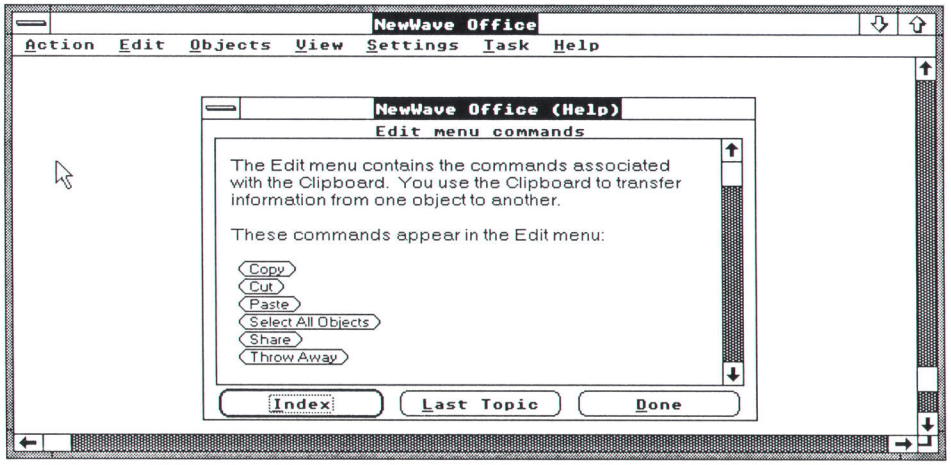


Fig. 2. *An example of the help text window.*

access to topics listed in the index. As single characters are typed into the editbox, a search begins immediately to find the first string in the listbox that matches the characters typed in the editbox. Once a topic is highlighted in the listbox (signifying a match), the user can press **Enter** to select that topic.

Help Window

The user can view the help text window and the index as one window, since the windows appear in the same place on the screen (on top of each other) and only one can be viewed at a time. The index and the help text window are separate windows and are programmatically handled differently. There is only one instance of help; the user would never see two help windows.

The help window appears to the user as a modeless dialog box belonging to an application. The help window can be moved, and disappears when the application closes or iconizes. When the help window is initially displayed, it is placed in the lower right corner of the screen. The user can move the help window if it is obstructing the application; this will allow the user to work in the application and also view the help text.

The help window displays the help topic title just below the help window caption bar. The help text is in 12-point Helvetica type, with options for bold, underline, and pushbutton (related topics) variations. The font variations are used at the discretion of the help text writer. If the topic text is longer than one screen (14 lines of text), a vertical scroll bar will be provided for single-line and page scrolling through the topic.

The Index pushbutton is available from the help topic window to let the user select another topic. The Last Topic pushbutton allows the user to back through previously displayed topics. A sample of the help text window is shown in Fig. 2.

Related Topics

Related topics are highlighted words or phrases within the help text that can be selected. The related topic highlight is a simulated pushbutton. When the user selects the related topic pushbutton, the help text pertaining to the related topic will be displayed.

Help Components

The NewWave help system is made up of three separate sections or pieces: the user interface, the help files, and the help file utility.

User Interface. The user interface refers to what the user sees and interacts with on the screen. The user interface can be further separated into two parts, which are separate executable modules.

The first part, or the front end, of the user interface code is contained within the application and performs independently of the back end. The front end is a dynamic library used by all NewWave applications, thereby giving the applications the common help interface. This benefits the user because access to help information is consistent between applications. The front end portion of the help system is contained within the NewWave dynamic library HPNWLIB.EXE, which is used by all NewWave applications.

The front end handles all of the help system functionality for the application. The front end initializes and maintains the help pull-down menu, maintains the question mark cursor, and monitors messages. The front end provides the routines that allow the application to communicate with the help system. In the NewWave environment, applications make API function calls, which in turn call help. The front end loads the back end into memory, begins execution of the back end, and then communicates with it.

The second part, or the back end of the user interface code, is a separate executable file, HPHELP.NWE, which is loaded by the front end. The back end does all of the help window maintenance (both the help text window and the index) and reading of the help files. The back end also maintains the connection to an application, since the connection changes when a second application asks for help. Most of the error handling is processed in the back end. Fig. 3 shows the pieces of the user interface.

Help Files. The help files are located on the disc in the same directory as HPHELP.NWE. For consistency, the help files are named with the application name and the .HLP file extension. There is one help file for each application and, with the current help system design, one help file can be used at a time.

The help file is a binary file that contains the structures, pointers, and help text required by the user interface. The format of the help file is shown in Fig. 4. The file header contains information about the entire file, such as pointers to the index table and context table and the lengths of the tables. There is one text block for each help topic in the file. Each text block has its own header describing the topic title and the number of related topics. The help text follows the text header, and the pointers for the related topics are

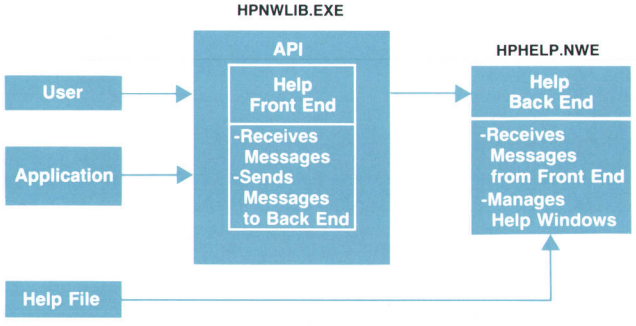


Fig. 3. *Architecture of the help user interface.*

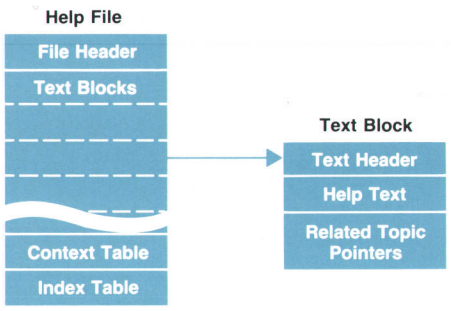


Fig. 4. *Format of the help file.*

placed after the text. Following all the text blocks is the context table containing the context numbers provided for ? mode. Last in the file is the index table which is used to display the help index.

Help File Utility. The function of the help file utility is to produce a file that is readable by the user interface code. The help system requires tables and pointers that are difficult to input manually. Therefore, the help file utility produces the desired results. The help file utility is basically a file converter, converting the text input by the help text writer to the desired format.

The utility, HPHELPFL.EXE (help files), is a DOS program that doesn't require MS Windows to run (however, it can be run with a PIF* file under MS Windows). HPHELPFL.EXE is not part of the NewWave environment, but rather a development tool that is used by NewWave application teams, help writers, and localizers.

How Help Works

As mentioned above, the help facility is an independent program. It is called by the application program interface (API) whenever the API detects a request for help while a NewWave program is executing.

The API is the primary interface between an application program and the rest of the NewWave environment. The API provides the functions that must be called by the application program code to start the program, stop the program, display API menus, and call other system services. These functions give the API the information to inform the help facility of the state or location of the application program whenever a user requests help.

For the NewWave programmer, there is no extra work involved to get the use of the help facility. If the programmer follows the standard NewWave guidelines, which include the API functionality, and provides a help file, the help system will automatically be provided for the application.

The following is a list of the API functions and the help facility counterparts. The functions are all part of the generic API template for NewWave applications and are included in the HPNWLIB module.

*A PIF file is a program information file that tells MS Windows how much memory to reserve, and whether the application is windowing, nonwindowing, nonswitching, or nontransferring.

```
/topic=Edit menu commands
/related=Copy
/related=Cut
/related=Paste
/related=Select All Objects
/related=Share
/related=Throw Away


The Edit menu contains the commands associated
with the Clipboard. You use the Clipboard to transfer
information from one item to another.


These commands appear in the Edit menu:

/R Copy/N
/R Cut/N
/R Paste/N
/R Select All Objects/N
/R Share/N
/R Throw Away/N
```

Fig. 5. *An excerpt from a help source file.*

- APIInit calls HELP_Initialise which initializes data structures, checks for the help file, and does general initialization.
- APIInitMenu calls HELP_InstallHelpMenu which attaches the help menu to the application's menu.
- APIUserActionInterface calls HELP_CheckMessage which is a message filter that executes help commands.
- APITerm calls HELP_Done which closes the help window, frees memory, and handles general termination.

To allow the user to get help from a dialog box, the programmer must provide a help pushbutton in the dialog box (in the resource file) and use the NewWave standard call APIDigUserActionInterface, which calls HELP_Topic to display the appropriate help topic.

Internal Functionality

All incoming help messages are processed by the front-end code. The heart of the front end and of the entire help facility is the message filter, consisting of APIUserActionInterface and HELP_CheckMessage. Help requests from the help pull-down menu come into the latter routine and all subsequent help functionality is generated from this routine. Because the API filters messages, the help facility receives only the messages that pertain to it. When the user selects an item from the help pull-down menu, messages are generated and are directed to the help message filter.

When the user selects ? mode, the help facility sends a message to the application, telling the application to set the mode flag to intercept on. The mode flag is a variable that is part of the NewWave architecture and is maintained by the application. Setting the mode flag causes all messages to be sent through the help message filter until the help facility tells the application to set intercept off (another message is sent). When intercept ? mode is invoked, the help facility maintains the ? cursor and interprets the help selection of the user. Once a selection is made with ? mode, intercept mode is turned off and a help message is generated to display the help topic.

When the request for a help topic or index is received (or generated), the front end processes the message by loading the back end and then sending it the appropriate messages.

The back end receives a message for displaying the help topic or index, then executes accordingly. The back end manages everything pertaining to the help text window and the index window, which includes reading the help file (text and tables), scrolling, window placement, font and related topics, and the last-topic stack.

For dialog boxes, the APIDlgUserActionInterface behaves similarly to APIUserActionInterface in that it is a message filter routine, checking all messages to see if the help pushbutton (or other API button) has been pressed. If the help pushbutton is selected by the user, then a help topic request is generated.

Context Numbers for ? Mode

The main task for NewWave programmers using the help system is defining and setting context numbers. A context number is a value that uniquely identifies a topic in the help file. All items that can be selected by using ? mode should have a context number in the help file. If there is no context number in the help file for an item that the user wants help on, the help system displays a message "No

help available for this selection," which is not very helpful. It is to the programmer's advantage to provide a complete set of context numbers.

In the case of menu commands, the context number is the command identification number set in the resource file.

When the user selects help in the client area, the front-end code sends a message to the application requesting a context number. The application responds to the message—API_INTERROGATE_MSG, case API_RENDER_HELP_FN—by returning to the help system a context value that pertains to the cursor position in the client area. The cursor position is given to the application within the API_INTERROGATE_MSG message. The application can return a context value based on the current state or cursor position within fields, thus producing context sensitivity.

Providing help for system menus and the nonclient area requires that the context values match those in windows.h (include file provided for MS Windows development). For example, to provide help for the caption bar, the value of HT_CAPTION is used in the help file. Nonclient area values are the HT_ values.

Creating Help Files

There is no help facility if there is no help file. Writing and developing the help text is a major task in providing help for an application.

The source files for the help text consist of control statements and help text. Control statements identify features pertaining to a help screen and the help file. Control statements are also directives to the help file utility for setting up structures and tables. Fig. 5 shows an excerpt from a help source file, and Fig. 2 is the resulting help screen.

Control statements are distinguished from the help text by the slash as the first character in a line of text. Here is a list of some basic control statements and their functions:

/context = #	Provides a context sensitive topic available through ? mode.
/topic = string	Declares string as a main topic that is listed in the index.
/end	End of the help screen.
/indent=string	Lists string in the index indented under the topic.
/related = string	Specifies where a related topic points to—i.e., string.

Any text (not control statements) between the /topic and /end statements is the help text that is shown in the help window. Text enhancements (bold, underline, and related topics) are specified within the text by marking the text to be enhanced. Besides the text enhancements, no other formatting is done. The text appears as it was typed within the source.

When the source file has the help text required for the application, it must be converted with the help file utility. The input to the help file utility is the source file, and the resulting file is the file used by the user interface code.

Stand-Alone Help

The help facility was first developed as a stand-alone facility to be used by MS Windows applications. During the development of the NewWave environment, several NewWave architectural component libraries were combined so that the applications only need to use one library to access these functions, rather than several libraries.

However, there was still a need for an MS Windows help system for other products that run in the MS Windows environment, so the help facility is also usable as a stand-alone version. The difference is that the application makes help calls directly instead of API calls. Other than the programmatic interface, there is little difference between the stand-alone and NewWave help systems. They both have the same user interface and operate in the same way.

Localizability

As in other NewWave and MS Windows applications, all help text strings are in a resource file. The strings can be translated into other languages and the resource file recompiled. There is no need to recompile and link the entire help system source code. This is a feature of application development under MS Windows.

For applications using the help facility, the help text is easily translated by changing the help source and then producing a new help file with the help file utility.

Acknowledgments

The help project had a lot of inputs from many people, but special thanks are due to Jonathan Weiner who did the design and implementation of the help front end and contributed much to the whole project. Thanks to Bill Thompson for his help during the investigation phase. The technical writers reviewed and commented on many of the features in the help facility that would be needed by help text writers. Thanks also to the engineers in the Scanning Gallery group for their feedback in the use of the stand-alone version of the help facility, and to section manager Larry Lorren for his support for this project.

NewWave Computer-Based Training Development Facility

Computer-based training in the NewWave environment allows users to learn how to use the system at their own pace, and provides facilities for users to create their own computer-based training courseware.

Lawrence A. Lynch-Freshner, R. Thomas Watson, Brian B. Egan, and John J. Jencek

FORMAL TRAINING IS OFTEN ASSOCIATED with a crowded lecture room. Despite a long and successful history, classroom training is becoming increasingly expensive without a corresponding increase in effectiveness. The growing influence of computers provides a possibility for improvement: let the computer do all or part of the instruction.

Computer-based training, or CBT, has been extensively used by the military for teaching everything from medicine to flying. Academia has also come to rely heavily on the patience of the computer, while bright colors, interesting music, and supplemental video all add appeal for a generation raised on television. Industry has been slower to adopt CBT. Available courses are limited, equipment and software are expensive, and people have felt threatened by the new technologies. Most important, many people are unconvinced that CBT is effective, often because of bad experiences with unimaginative or boring CBT.

Properly written CBT can cut costs while raising retention and motivation. Achieving this requires a partnership between the courseware and the CBT authoring software:

- Ideal CBT courseware is flexible enough to handle a variety of student experience levels, provides task-based instruction immediately applicable to the job, is available whenever needed, provides chunks of instruction relevant to the task at hand, and doesn't constrain the student because of its own limitations.
- Ideal CBT authoring software is simple to use with minimal programming experience, provides a realistic learning environment, costs very little, and allows courseware to be developed quickly and inexpensively in response to local needs.

In creating the HP NewWave CBT facility, we set out to reach these ideals. Earlier experiences with commercially available CBT authoring and delivery systems showed the potential of CBT, yet also pointed out the limitations of conventional technologies. It was time for original thinking.

NewWave CBT Facility Design

Throughout the project, there have been four design goals for the NewWave CBT facility: it must use the NewWave architecture, it must provide effective courseware, it must simplify and speed the development process, and the courseware must be adaptable to local cultures and languages with minimum effort.

No commercially available CBT authoring or delivery system was available for either the NewWave environment or Microsoft Windows. To take advantage of the power of the NewWave environment, a CBT system needs a graphic display, full mouse and keyboard input capability, the ability to span concurrently open application windows, and the ability to operate on what the students do and how they do it. CBT also must be started from within the NewWave environment, since requiring a return to the MS-DOS

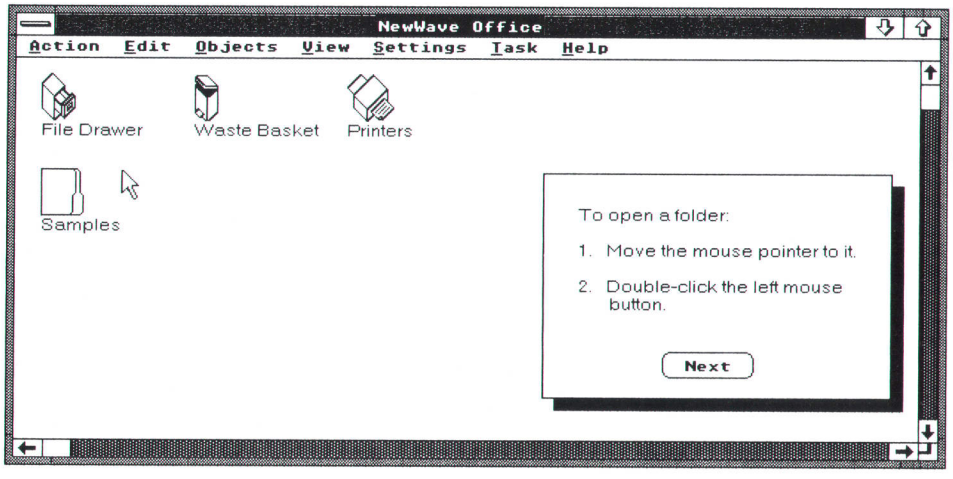


Fig. 1 *Sample lesson, frame 1.*

prompt for training would probably discourage people from using it. Finally, since the NewWave environment is capable of running existing MS-DOS and Microsoft Windows applications (without providing many of the NewWave environment features) the CBT must also provide some way of training on these applications, even if only by simulating them within the NewWave environment.

The second requirement for the NewWave CBT facility was that it must provide the capabilities for an unparalleled level of quality in the CBT courseware. CBT is an integral part of the learning product strategy for the NewWave environment. When properly designed and executed, CBT has proven to be successful and inexpensive. Our goal was to minimize the technical limitations placed on the lesson author by allowing for multimedia lessons (text, graphics, etc.), modularized courseware, and easy access from within the normal work environment.

The third requirement for the NewWave CBT facility was that it must reduce the long development times traditionally associated with CBT courses. A typical hour of CBT takes between 300 and 500 hours to design, construct, and test. Much of this time is spent in programming the CBT logic and creating the screen displays, rather than in the instructional design itself. By providing efficient, easy-to-use tools, and by eliminating as much programming as possible, the NewWave CBT facility can make expense less of a consideration when deciding whether CBT is an appropriate medium for a particular course.

Finally, the courseware created with the NewWave CBT facility had to comply with HP's guidelines for localizability. The primary requirement is that text should be maintained separately from program logic. This way, nontechnical translators can translate the lessons into local languages without having to delve into the source code of the course. Since translated text is often 30% larger than the original English version, the position, size, and proportions of the NewWave CBT text window had to be easily adjustable, with automatic text wrap and preservation of formatting, so that the localizers could ensure the legibility of the lesson without having to recode it. Finally, text within illustrations had to be accessible separately (i.e., no bit-mapped text) so that the illustrations would not have to be redrawn to translate them.

During its history, CBT has evolved into two families of technologies:

- Simulation. The CBT software is fully responsible for what the student sees, with all screen displays produced by the training software. Simulations have great flexibility, allowing training on any real or imagined subject, but require more development effort because an entire environment must be recreated.
- Concurrent. A CBT engine resides in memory and runs in conjunction with a real software application. The application provides all its screen displays and computations just as if it were being used normally. The CBT software sequences the lessons, supplies instructional text, and controls which keystrokes and commands are allowed to reach the application. Since the application supplies the bulk of the code, concurrent CBT is usually easier to produce, but few applications can interact with the CBT engine in a really meaningful way.

The NewWave CBT facility is designed to maximize the advantages of both methods, providing text, graphics, and animations for vivid simulations and intimate communication between the CBT lesson and the applications being taught.

A Sample Lesson

The best introduction to the NewWave CBT facility is a sample lesson. Fig. 1 shows the first of a series of screens that might appear during part of a CBT course about the NewWave Office. These screen displays, or frames, sequence in response to student actions.

Frame 1. The real NewWave Office (not a simulation) is running, with all of its normal tools and objects visible. Also showing is a real folder object, placed by the CBT facility specifically for this lesson. Overlaying the Office is an instructional window, which contains an explanation of how to open an object, and a pushbutton control. The student reads the text in the window and clicks the mouse pointer on the Next pushbutton to go on to the next frame.

Frame 2. The window now contains an illustration of a folder, along with an instruction to the student to try opening the real folder. The directions are repeated for reinforcement. At this point, there are two options. First, the student can try to open the folder called "Samples." Second, the student can click on the Show Me pushbutton, asking for the CBT to do it once as a demonstration. We'll assume the latter choice for now.

Frame 3. The instructional window now describes the first step in the open process. The mouse pointer, by itself, slowly moves from its previous position to the folder "Samples" and pauses.

Frame 4. The window now changes to display the second step. The screen reacts to the double-click performed by the CBT facility, and the folder begins opening. The mouse clicks are not simulated; instead, the real message is injected into the system by the CBT facility. Beeps are sounded by the computer's speaker to mimic the sound of the mouse buttons clicking. When the student clicks on the Continue pushbutton, the folder is closed automatically and the next text frame is displayed.

Frames 5-6. The student is now asked to open the folder unassisted, just as in Frame 1. If the open is unsuccessful, an appropriate remedial message is given, and the student is asked to try again, as in Frame 2.

Frame 7. If the open is successful, congratulatory text is now displayed, and a brief animated reward appears. Then, using pushbuttons, the student chooses the next step: continue to the next lesson, or return to the main course menu. In either case, the CBT facility closes the "Samples" folder and destroys it, so that it won't remain in the NewWave Office as an artifact of the training.

The NewWave CBT facility is capable of monitoring the student's actions to a very fine level. The choice of which conditions to watch is left to the instructional designer, and will probably vary throughout a lesson. In this lesson, for diagnosing the cause of the open failure, some possibilities might be:

- An open was attempted, but on the wrong object. In this case, to save time and distraction, the open operation can be prevented, with an appropriate message being

substituted.

- The mouse was double-clicked, but off the folder icon.
- The folder was selected (by a single click) but not opened. Here, a time-out would be used to assume the action had not been completed.
- And so on. The number of monitored possibilities is limited more by the designer's imagination and time constraints than by technology.

An Overview of CBT Components

The initial vehicles for CBT were the NewWave agent and the NewWave application program interface (API). As a task automation facility, the agent can sequence through a series of steps either automatically or in response to interactions with the computer user. The API provides a door into all consenting NewWave data objects and tools, allowing the agent to control them or determine their inner states. Together, the agent and the API can automate anything a user might do. Thus the basics for a powerful CBT toolset were present in the NewWave environment from the beginning.

At its simplest, a CBT lesson is just an agent task (see article on page 38). Generic agent commands can open conversational windows anywhere, display textual information, present pushbuttons for user control, monitor certain events within the system, and make sequencing decisions in response to user actions.

While these agent tasks are sufficient for some training, they are not optimal for large-scale, highly visual CBT. They require programming expertise to construct, and because of their size when used for CBT, are expensive in terms of development time and memory use. To construct superior training, we needed additional visual effects, full-screen graphics, the ability to simulate applications that didn't lend themselves to concurrent training, a more detailed knowledge of what the student was doing to the system, and a clean and easy method for starting and controlling a sequence of lessons. This required that the generic agent task language be supplemented by:

- Extensions to the generic agent task automation language that perform training-specific actions such as mouse position sensing and user input simulation.
- A CBT display object (with integral development editor), which displays the sequences of instruction windows, text, static graphics, and user controls that make up the student's view of a lesson.
- An animation object (and editor), which displays color or monochrome animated graphics.
- A CBT menu object (and editor), which displays the course's initial user interface and allows access to all instructional objects.
- Interrogation hooks coded deep within NewWave data objects and tools, which send information and perform actions in response to application-specific agent commands.

In its current form, the NewWave CBT facility allows lesson authors to create full-color graphical and textual objects without writing a single line of code. A short and straightforward logical structure written in the agent's task automation language provides flow control for the lesson.

Architecture for Application Training

The NewWave architecture has been designed to support an integrated application training approach. This approach has its roots in concurrent training technologies, but has been taken much farther in the NewWave environment.

To facilitate an integrated approach, NewWave objects are designed to communicate through the API. A typical application architecture includes a user action processor and a command processor (see Fig. 2). The user action processor collects user input (mouse movement, keyboard input, etc.). It takes no action until it detects that an input sequence has conformed to the syntax of a command. At this point, the user action processor sends the command through the API to the command processor, which processes the command and updates the object's state, data, and/or interface. Hence, the syntactic (element-by-element) user actions are translated to semantic (meaningful to the system) commands.

When several objects are open under the NewWave environment, all following this protocol, the agent has privileged access to monitor and examine commands that are sent by the objects through the API command interface. If desired, the agent may also filter a command from a user action processor, causing it to be ignored by preventing it from reaching its respective command processor. These

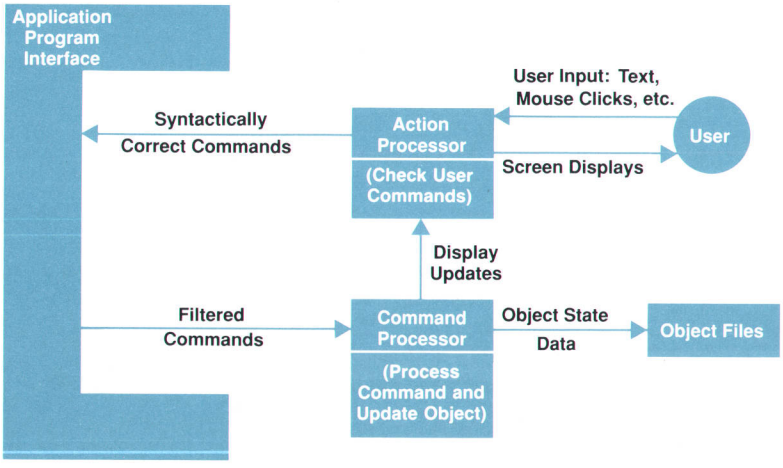


Fig. 2. *An architecture for application training.*

techniques, called *command monitoring* and *command filtering*, are employed by training programs that are based on the agent and can be used to guide the user through learning and using NewWave applications. The primary advantages over previous application training technologies are:

- Training programs do not need to simulate applications, since the real applications are used.
- Monitoring of application activities is at a semantic level, so the training program observes a command like OPEN FOLDER "Samples" instead of a sequence of mouse and keyboard inputs that must be interpreted.

It is common for a NewWave application to provide alternative ways to issue any given command. Typically there are at least two alternatives, one using the mouse and another using the keyboard. In either case, the same command is generated. This greatly simplifies the effort involved in developing training.

The Agent and Agent Tasks. Agents can be thought of as software robots. The NewWave agent follows instructions from the user. It can automate tasks by sending commands to applications and can record tasks by receiving and storing commands from applications. Additionally, the agent can monitor and filter commands, as previously mentioned. The sequence of instructions that the agent follows is called a task, and the language in which tasks are written is the agent task language.

Command Level Control. The easiest and most common use of the task language is to control NewWave applications. For example, part of a task might be:

```
FOCUS ON DOCUMENT "Letter"
TYPE "Dear Chris,"
```

This tells the agent to direct its commands at (FOCUS ON) a document object called "Letter," and then to TYPE some text in the letter. Before it can receive agent commands, the letter must be open. This would have been done earlier in the task by:

```
FOCUS ON OFFICE
SELECT DOCUMENT "Letter"
OPEN
```

Here, the agent is instructed to direct commands at the main NewWave Office window, select the document object called "Letter," and then open it. Notice how the task language is modeled after the semantic activities of the user. Since the user would follow an identical sequence to open an object, this type of agent interaction is called *command level control*.

Training tasks will typically control applications this way to initialize them for a lesson. For example, in a lesson on enhancing text within a document, a training task can automatically open the document and conduct other setup activities, rather than requiring these of the user.

Additionally, training tasks can use command level control to present instruction collected in a separate object. Consider the CBT display object, which is used by the training author to design a set of named instructional windows that can be randomly accessed. Within the lesson task, commands can be sent to the CBT display object to open instructional windows at appropriate times. At the beginning of such a task, the CBT display object is opened:

```
FOCUS ON OFFICE
SELECT HP_CBT_DISPLAY "Lesson1 Instruction"
OPEN_SHIFTED
```

Later in the task, commands are used to display specific instructional windows:

```
FOCUS ON HP_CBT_DISPLAY "Lesson1 Instruction"
GOTO_FRAME "How To Open"
```

The command GOTO_FRAME advances the instructional sequence to the CBT display object's How To Open frame.

This approach offers a significant benefit, in that training content in the CBT display object is conveniently separated from the training logic in the task. Hence, lesson content can be created, modified, and localized by nonprogrammers.

Class Independent and CBT Commands. During the control of objects from an agent task, most of the commands used are specific to a class of applications, that is, they are class dependent. For example, GOTO_FRAME is a command that is specific to CBT display objects. Such commands are executed by the object that received the FOCUS command, and not by the agent, which only delivers the commands.

To provide the rich syntax available in other high-level languages, the task language also has class independent commands such as IF..ELSE..ENDIF, PROCEDURE..ENDPROC, WHILE..ENDWHILE, and an assignment statement for variables. These commands are executed by the agent.

In addition to the generic agent commands used for all task automation, a set of commands specific to training development can be included by placing the CBT ON command at the beginning of a task. Likewise, if these special commands are not needed, the CBT OFF command can be used to speed the language translation process.

Command Level Monitoring. Many of the decisions made by the agent during a CBT lesson are based on the particular commands a student executes. To process the command activities of NewWave objects, two things are needed: a trap that recognizes that a command has occurred, and a procedure that interprets the nature of the command and acts on it. A typical implementation might be:

```
ON COMMAND DO TrapProcedure
SET COMMAND ON
WAIT
```

The ON COMMAND DO command is used to define which task procedure contains the interpretation and action steps. Once a monitoring procedure has been specified, monitoring must be turned on with SET COMMAND ON. This arms the trapping so that when any command is received through the API, the task jumps to the specified procedure. Typically, the third command in such a sequence is WAIT. The WAIT command directs the agent to stop processing task language commands, and to wait for a command to be generated in a NewWave object. Essentially, the agent is idle until this condition is met. When a command is de-

tected, the agent executes the monitoring procedure and then resumes execution of the task, beginning with the first command after the WAIT. It is also possible for the agent to execute other commands in a task while it is waiting for a command trap, but this scenario is more complex.

The monitoring procedure can contain any class independent commands. Its roles in a task are to examine commands that are trapped and to filter undesired commands. A monitoring procedure can use four class independent commands to acquire specific command information: the class and title of the object in which the command occurred, the type of command that occurred, and the parameters of the command. From the object's perspective, the monitoring procedure sits between the user action and the command processors, acting as a watchdog and a valve.

The agent can simultaneously monitor several objects for commands. For example, a single trap procedure can be written to act on a command either from the object being taught or from the CBT display object.

User Action Level Control. Although command level control and monitoring are quite efficient ways to work with objects in a task, there are instances where the user action level is more appropriate. For example, the training task may need to distinguish between two alternative syntaxes of the same command to ensure that the user has learned one of them. The user action level is also inherently part of all Microsoft Windows-based applications. Thus, training can extend into the domains of non-NewWave applications at the expense of additional task complexity.

One powerful use of user action level control in a training task is for demonstrations. In a demonstration, commands like POINT, DRAG, CLICK, DOUBLE-CLICK, and TYPE are used to manipulate objects with full visual feedback of mouse pointer movement and individual key entry. Command level control would not suffice for demonstrations, since it only shows the display changes that follow changes in object state, rather than the causes of those changes.

Interrogation functions complement user action control by locating specific display elements. Class independent interrogation functions locate elements of the windowing interface, such as window caption bars and system menu boxes. These functions also locate elements that are managed by specific objects, such as a folder object icon within the NewWave Office window. Class dependent interrogation functions are used to ask the object questions like:

- Where on the screen is a display element?
- What display element is at a given point on the screen?
- What is the status of some condition within an object?

Each of these questions deals with revealing information that is known only by the object. The first two questions map between display elements that are managed by the object and screen coordinates. For example, REGION_OF_OBJECT returns a region, which is a data type that specifies a rectangular screen region by its origin (upper left point), width, and height.

Together, user action level control and interrogation can be used to construct demonstrations that will always work, regardless of where elements of the demonstration have been moved. For example, to demonstrate how to open a folder object called "Samples":

```
FOCUS ON OFFICE
SAMPLES#=REGION_OF_OBJECT ("hpoffice folder","Samples")
POINT TO CENTER (SAMPLES#)
DOUBLE_CLICK
```

In these four commands, the office object is interrogated for the rectangular region occupied by the icon of the folder "Samples." The answer to the interrogation is assigned to the variable SAMPLES#. Finally, the mouse pointer is directed to move to the center of the icon, and then the effect of a left mouse button double click is produced.

The CBT Display Object

The CBT display object provides a fast, easy, and flexible way to create and display training-specific screens ranging from small and simple text windows to simulations of entire applications.

The basic building block of a CBT lesson is the frame, which contains everything that might appear on the screen at any one time. By sequencing between frames, various textual instructions and graphical illustrations can be presented to the student. Frames may be made up of any of the following:

- Windows, which can be full or part-screen. These can be any color, and can have various styles of borders. An elaborate window might look like a NewWave application window, with sizing controls, scroll bars, real menus, and so on. Windows are used as the framework to hold other elements, or can be used as backgrounds.
- Text, which comes in a variety of sizes, colors, typefaces, and enhancements.
- Controls, which can be pushbuttons, check boxes, or other forms.
- Color or monochrome bit maps, which can be input through the HP ScanJet scanner or MS Windows clipboard, or created using a built-in painter utility.
- NewWave environment icons, MS Windows stock icons, or icons loaded from a user-created file.
- Graphic primitives, such as lines, arcs, and circles, which can be drawn in various weights, colors, and fills.
- Animations, which are actually separate animation objects controlled by the frame object.
- "Hot regions," which are invisible areas sensitive to mouse button clicks.

A CBT display object contains one or more frames that

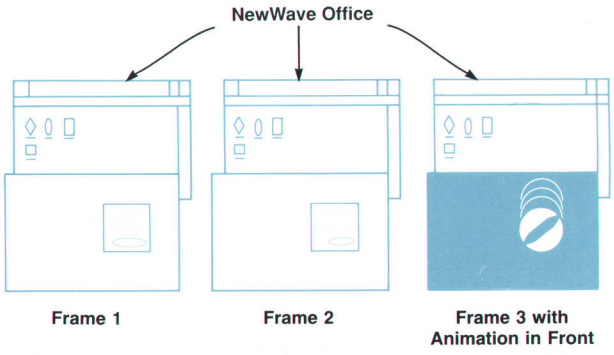


Fig. 3 *The foreground/background relationship.*

hold all of the displays needed for a complete lesson. Windows, menus, and controls are all real, but are connected to the intelligence of the agent rather than NewWave application code.

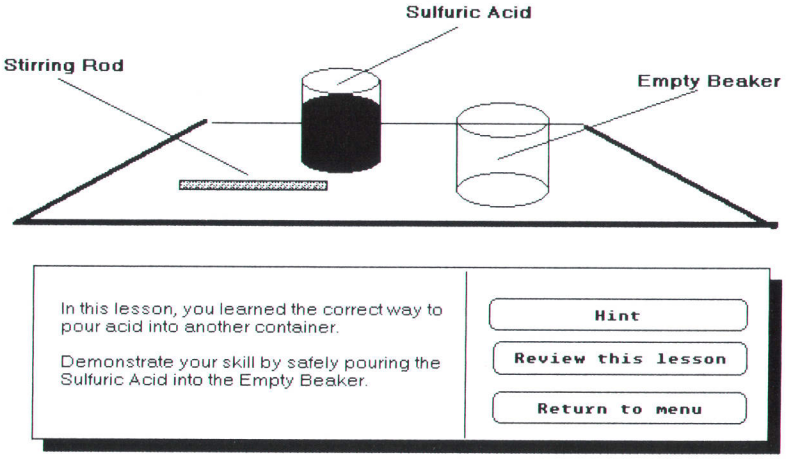
Fig. 3 shows a schematic view of the sample lesson discussed earlier. The frame sequencing for a lesson is handled by the agent. Simple statements command the CBT display object to display a particular frame. The student then performs an action such as opening an object, selecting from a menu, or typing in text or numbers. The task reacts to the action in a predetermined way by advancing to the next frame, requesting additional text, or presenting an error message.

Without the CBT display object, input/output and display control would have to be handled by the task language. With the CBT display object, the agent is used solely for decisions and sequencing. This greatly reduces the size of the task, minimizes the need for programming expertise, and speeds the lesson development process.

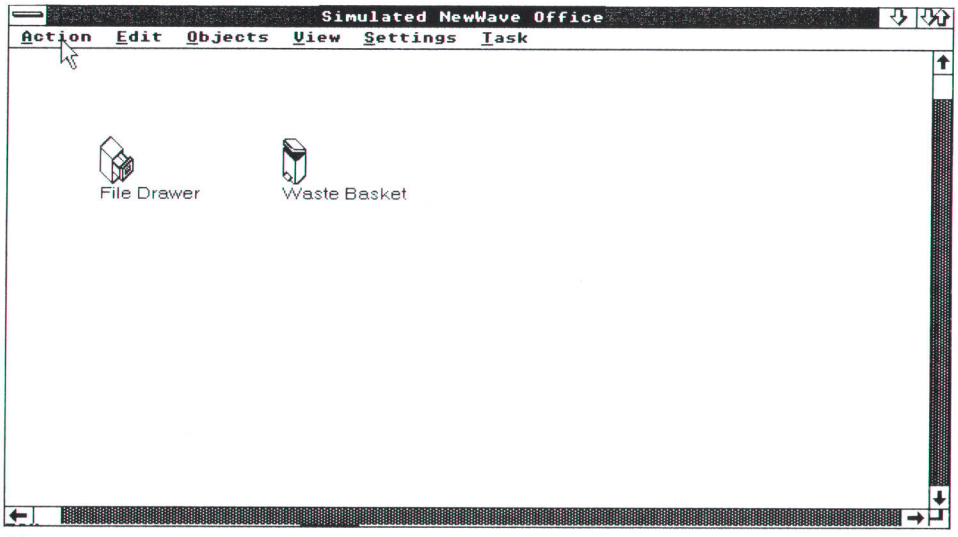
When the lesson's final animation is played, a full-screen background window covers the NewWave Office, simplifying the display. The agent is responsible for coordinating the CBT display object and animation object.

The frame editor can create very sophisticated screen displays. This allows the CBT to go far beyond simply displaying text on top of existing NewWave objects. It allows virtually any program to be simulated or prototyped, and allows courses to be developed on subjects far removed from software applications. Fig. 4 shows two such possibilities.

Program Structure. The CBT display object runs in two modes: development and run-time. The run-time version displays the frames inside the object, while the development version adds the editing facilities for text and graphics. If the object is opened by the agent, it assumes it is in run-time mode and displays the first frame. If the object is opened manually, it assumes it is in development mode and displays the initial editor interface, along with the first frame—if there is one yet. Since the CBT display object is hidden inside the CBT container object on run-time systems, it can only be opened manually by authors using a development system. Fig. 5 is a block diagram of the CBT display object. The painter (a bit-map editor) and the color selector are placed in dynamic libraries to maximize code reuse, since they appear in several places within both the CBT display object and the animation object.



(a)



(b)

Fig. 4. *(a) Nonsoftware simulation. (b) Simulated NewWave Office.*

The CBT display object will usually be used to develop training that will run concurrently with one or more NewWave objects. To simplify the synchronization of the CBT lessons with the applications, the development-mode frame object is designed to run unobtrusively on top of the object being taught about. This allows the lesson author to place windows and other frame elements optimally without having to guess what the final lesson will look like.

The primary user interface of the CBT display object editor consists of a small window which contains menus and a list of frames. A new frame is created by typing a name in the Title field and clicking the Add button. Any frame can be selected for editing by double-clicking on its title. The CBT display object editor window can be moved around as needed so that the lesson author has access to the entire screen display without interference.

Once a new frame exists, a window must be created to form a parent for all other elements in the frame. This window can be a featureless area used only to contain other elements, or it can be an integral part of the lesson. The opaque background or the simulated NewWave Office shown earlier in Fig. 3 are examples of these two variations.

Recall that a frame can contain windows, text, controls, bit maps, icons, graphic primitives, animations, and hot regions. Each of these elements is created through specialized menus which provide easy and fast selection of various features. Once created, elements can be locked to prevent inadvertent movement or editing. They can be reselected for editing at any time, and can be moved, cut, or copied within or between frames. Additionally, text and bit maps can be imported from any NewWave or Microsoft Windows application using the MS Windows clipboard. When displayed, elements are stacked on top of one another, and if elements overlap, the highest one shows. The author can pull a given element to the top or push it to the bottom to control whether it is obscured by other elements.

Internally, each type of element in a CBT display object is maintained in a single file, and when a frame is loaded into memory, all elements are fetched and readied for display. While the files are essentially invisible in an object-based system, they may be specially accessed for the purpose of translating text to a local language.

The CBT Display Object and the Agent. The CBT display object and all of its elements are designed for a high level of interactivity with the agent. Class dependent commands are used at run time to sequence between frames, hide and show various windows, launch animations, and sense when elements have been clicked on. All menus, submenus, and controls that appear in a displayed frame are real, but they are not connected to any application code. Instead, any frame element can be given a unique name. Class dependent agent commands are used to determine when a named object has been selected, and then the agent evaluates the choice and directs the frame object to display the frame that reflects the action.

An optional "go to frame n" feature can be specified for any control (pushbutton) in any frame. Clicking on that element will cause the frame object to display a specific frame automatically, without any interaction with the agent, providing a high-performance, self-contained, hypertext-like capability.

The Animation Object

Animated demonstrations and graphics are often more instructionally valuable than static graphics, and can play a major role in keeping students motivated. The animation object is designed to provide high-quality animation with minimal effort on the part of the lesson author.

The animation object is analogous to an ordinary animated cartoon. It consists of a series of pictures which, when sequenced rapidly, give the illusion of motion. Fig. 6 shows a typical animation object being created. The upper filmstrip display gives a low-resolution view of a bouncing ball in each of the four poses that make up its motion. The lower display is a detail view of a pose, which can be edited. When the sequence of poses is played, the ball appears to bounce in place. Moving the sequencing object horizontally as it plays makes the ball appear to bounce across the screen.

Depending on its purpose, an animation may take several forms. An animation might be a single image, such as an arrow, which is given a velocity in some particular direction, or it might consist of a sequence of poses that remain in the same place on the screen. Another form, such as a barnstorming airplane, might have a complex motion path and several poses.

Each pose or cell in an animation is a bit map. These bit maps can be created in several ways:

- The integral bit-map editor can draw them directly, in either color or monochrome.
- They can be imported from any NewWave object or Microsoft Windows application using the MS Windows clipboard.
- They can be hand-drawn on paper and scanned in using the HP ScanJet scanner.
- All or part of an image in one frame can be copied to another frame, where it can be used as is or modified.

If desired, a combination of these methods can be used, so that a hand-drawn image can be scanned in and com-

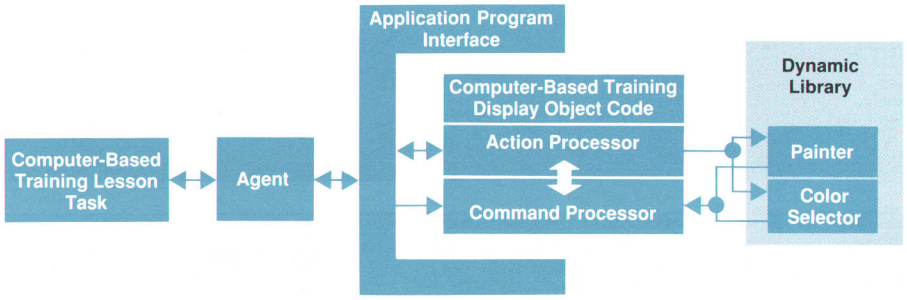


Fig. 5 *Block diagram of the CBT display object.*

bined with a bit map from another application, and the composite image colored using the editor. An image can be created on a black, white, or transparent background. An image can also have transparent areas painted on it, so that the actual screen display will show through the animation when it is played. The editor also provides several alignment features, such as a grid, to allow the images to be positioned precisely for smooth movement.

The initial position, velocity, and direction of an animated image can be set using a menu in the editor of the animation object. If a complex path or changes in velocity are desired, or if the lesson author wishes to freeze or hide an animation during run time, the animation object's comprehensive class dependent task language allows full agent control.

Operation of the Animation Object. Like the CBT display object, the animation object has both development and run-time personalities, the difference being the presence of the editing facilities. The run-time version is primarily controlled by the agent or a CBT display object. The development version opens ready for modification, and provides only limited play capability for testing the animation.

Animations are sequences of bit-map images, transferred one at a time to the display screen. Unlike ordinary cartoons, which own the entire screen of a television set, NewWave animations must coexist with the other objects that appear on the screen. The animation object contains a pair of buffers which save areas of the screen that will be overwritten by an animated image. Image bit maps are transferred to these buffers, rather than directly to the screen. Once the buffers contain the correct display, they are transferred to the display screen. When the animated image is about to move to a new point, the saved area is restored into the buffer to obliterate any trace of the image in its former position.

An author will often crop an animated image with an irregular shape by surrounding it with a transparent background. This allows the image to pass over other objects without a "halo" surrounding it. Transparent areas can also be drawn into an image so the screen background shows through. The technique used for this is analogous to the matte process used by filmmakers. An animation frame with transparent areas contains both its normal image and an automatically created mask, or outline, of the non-transparent part of the image bit map. When an image is written to the buffers inside the animation object, its mask is first combined with the buffer, removing all the colors from the area where the image will go. The mask is also combined with the image itself, removing the background and leaving just the picture part. The stripped image is then placed precisely over the "hole" left in the screen display. Since the two images are not actually combined, there is no interference between the screen and the image.

The Animation Object and the Agent. The animation object has a rich class dependent task language which allows powerful agent control of a running animation. Animations can be started, frozen, or hidden, the course and velocity can be changed at will, or a programmed complex course can be loaded into the object in one step. Subsets of the frames in an object can be specified for display so that several different animations can be performed without having to load a new object.

CBT Start-up and Menus

A CBT lesson consists of an agent task object and a frame object, and may also contain one or more animation objects. A full CBT course will have many of these objects. If all had to reside openly in the NewWave environment, they would clutter the Office and confuse the student. To prevent these problems, all of the objects in a CBT course are contained in a special CBT global container object. It has many of the properties of other container objects like folders, but remains invisible in a run-only system.

The CBT global container object serves several purposes:

- It contains all of the CBT objects, simplifying the NewWave Office display.
- It protects the CBT objects from accidental deletion or modification.

A separate CBT menu object keeps track of topics and lessons:

- It provides a start-up capability for the various agent tasks that drive the lessons.

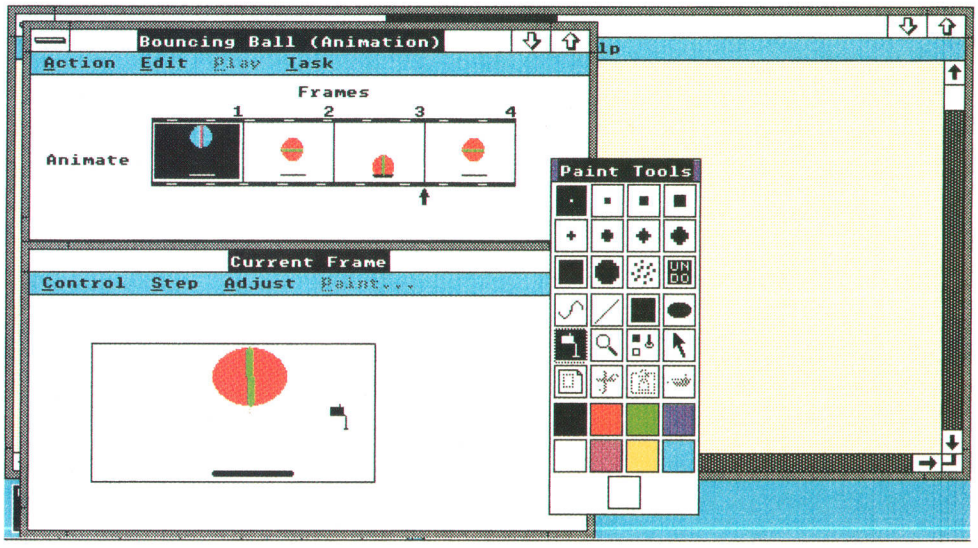


Fig. 6. *The animation object's editor.*

- It presents a menu that allows the student to choose a particular lesson within a course, and maintains a record of which lessons have been completed.
- It accepts additional CBT lessons from newly installed objects and integrates them into the standard menu.

To start CBT, the student pulls down the NewWave Office's Help menu and chooses Tutorial. This opens the CBT menu object and displays its initial menu. First-time users will probably not have acquired the mouse skills needed to use menus in the ordinary way. To get them going, the NewWave Office comes with an autostarting agent task, which gives the new student the option of running the CBT by pressing the Enter key. This autostarting task can be disabled after the student takes the lesson, removing the overhead of the question when it is no longer needed. The CBT menu object also allows a lesson to be specified as the default, and pressing the Enter key will initiate the default lesson. This provides a backup for those students who may be working on a system whose autostart training task has been removed.

CBT menus are nested in outline form. Students choose a broad topic, and a submenu is then brought up that specifies the actual lesson titles. After a lesson has been completed, a check mark is placed on the menu to help the students track their progress. When a lesson is chosen, the CBT menu object dispatches the appropriate task to the agent, which then runs the lesson. When the task is finished, the agent returns control to the CBT menu object so the next lesson can be run.

Developers of NewWave applications can write their own CBT lessons using the CBT development facility. As part of the installation process for the new application, its CBT is loaded into the CBT menu object and its menu structure is registered. This allows all lessons in the NewWave environment to be run from the same place, ensuring easy use and easy tracking of all lessons.

Conclusion

New hardware technologies such as voice, interactive videodisc, CD-ROM, and pen-and-paper input devices are all finding places in the training environment. The inherent flexibility and expandability of the NewWave architecture make it easy to incorporate new ideas, and the power and ease of use of the NewWave environment make it a natural vehicle for exploring new techniques.

The NewWave computer-based training facility is still in its infancy, yet it contains features not found in other CBT systems. It embodies the true potential of the entire HP NewWave environment: to give knowledge workers the time to let their imaginations work and the means to make their ideas real.

Acknowledgments

The NewWave Computer-based training facility was the goal of many people for several years. The continuing support of HP's Personal Software Division's (PSD) former learning products center manager Barbara Baill has been critical to the project's success. Tim Gustafson, Karen Tucker, Lisa Heckenmueller, and Bob North have made invaluable contributions to the development and refinement of the CBT facility through the courseware they have written for the NewWave environment. Bill Coleman and Tom Rideout of PSD's human factors group have made many improvements to the usability of the CBT development software. Glenn Stearns, Barbara Packard, and the entire NewWave agent project team have made the CBT possible through their foresight and continued support. Tony Day and Andy Dysart have been extremely understanding about enhancement requests and pleas for assistance. Finally, the team would like to thank Kim Aglietti, Joe Ercolani, Mary Page, and Deborah Plumley for their efforts during the early investigation and design phases.

Encapsulation of Applications in the NewWave Environment

To allow non-NewWave applications to run in the NewWave environment, the NewWave encapsulation facilities provide features for the partial or full integration of these applications into the NewWave environment.

by William M. Crow

THE HP NEWWAVE ENVIRONMENT provides powerful capabilities for applications that are written to take full advantage of the object management facility (OMF), the agent, and other NewWave features. However, there are thousands of personal computer software applications currently available that were not written for the HP NewWave environment, or for that matter, were not even written to operate under Microsoft Windows. In many cases, these are mission-critical applications that organizations depend on as part of their day-to-day operations. These applications may ultimately be replaced by better solutions that take full advantage of the HP NewWave environment. However, if we expect users to begin using the HP NewWave environment today, users must be able to continue to use the software currently at their disposal.

For an existing MS-DOS®-based application program to operate correctly in the HP NewWave environment, either the application must be modified, the HP NewWave environment must recognize and accommodate the MS-DOS application, or an additional program must provide an interface between the MS-DOS application and the HP NewWave environment. The HP NewWave encapsulation facility uses a combination of all these techniques to provide a wide range of support for applications not specifically written to operate in the HP NewWave environment.

In some cases, the encapsulation facility makes it possible to continue to access applications that were in use before installing the HP NewWave environment, but offers no enhancement to their features or operation. In other cases, encapsulation makes it possible for existing applications to take full advantage of the HP NewWave environment without the need for a complete rewrite. For most existing applications, the situation lies somewhere between these two ends of the spectrum. The basic levels of application program encapsulation in the HP NewWave environment are shown in Fig. 1.

Microsoft Windows Multitasking and Context Switching

Because the HP NewWave environment is based on the Microsoft Windows environment and runs on industry-standard workstations that support the MS-DOS operating system, it is always possible to run other applications by temporarily leaving the HP NewWave environment and the Microsoft Windows environment. While workable, this approach is not always practical, and certainly does not meet the objective of providing a complete environment for the HP NewWave user. Microsoft Windows improves on this by providing the necessary device and memory management facilities to allow multiple applications to run simultaneously. These applications may or may not be written for Microsoft Windows. A Microsoft Windows application will appear on the screen in its own window along with the windows of other Microsoft Windows applications. Multiple Microsoft Windows applications can be executing simultaneously, depending on available memory. The user interacts with them by using the mouse or keyboard to select the appropriate on-screen window and to enter information or select commands.

If the application is not written for the Microsoft Windows environment, it can still be accessed and operate in conjunction with other applications through additional facilities provided by Microsoft Windows. Such an application will be given control of the entire screen, presenting its own display and user interface. The user can switch contexts between the full-screen application and Microsoft Windows with a simple keystroke sequence. Multiple full-

DOS Programs

- Program Launch
- Context Switching
- Simple Cut and Paste

Generic Encapsulation

- Program Launch
- Context Switching
- Simple Cut and Paste
- Iconic Representation
- Direct Manipulation for Open, Copy, Move, Mail, and Discard
- Autoregistration for New Files
- Menu Overlays with Macros

Application-Specific Encapsulation

- Program Launch
- Context Switching
- Simple Cut and Paste
- Iconic Representation
- Direct Manipulation for Open, Copy, Move, Mail, and Discard
- Autoregistration for New Files
- Direct Manipulation for Print
- Browsing with Agents and Help
- Outgoing Views (Visual, Data)
- Many Additional Application Special Features

Fig. 1. *Basic levels of encapsulation.*

screen applications can be active simultaneously, depending on the amount of available memory in the system. Full-screen applications are not truly multitasked by Microsoft Windows because an application's operation is suspended when it is not displayed on the screen. The complete state is saved and the application continues operation when the user once again gives it access to the display.

The encapsulation facilities take advantage of this fundamental capability of Microsoft Windows, and provide ways to operate MS-DOS-based applications from within the HP NewWave environment.

The DOS Programs Service

In its simplest form, the encapsulation facility provides an easy-to-use method to access and run other applications from within the HP NewWave environment. A user configurable menu of available MS-DOS applications is accessed through the DOS Programs... command in the HP NewWave Office. Selecting an entry from this menu starts the application, using the facilities in Microsoft Windows discussed earlier. Fig. 2 shows the DOS programs service menu.

Adding, removing, or modifying the menu of available applications requires no special knowledge or programming skills. The user can directly access a simple configuration command to make the applications chosen available through the HP NewWave DOS programs service. For many popular commercial applications, it is as simple as selecting the desired program from a preconfigured list.

There is virtually no other relationship or integration between the HP NewWave environment and an application encapsulated using the DOS programs service other than the ability to start it up. The application still accesses the MS-DOS file system to store and retrieve information. The user must be aware of the proper methods to specify filenames and navigate MS-DOS directories. Accessing data from these applications within the HP NewWave environment requires an explicit process to import or convert the MS-DOS file to a form recognized by the HP NewWave object management facility (OMF).

Because the DOS programs service is itself an HP NewWave application, it is fully integrated with the agent facility. HP NewWave agent tasks can access the service and start the operation of an MS-DOS application. However, the agent cannot monitor or control the operation of the application once it is started because that application was not developed with the necessary agent interfaces. Because many existing applications offer their own internal facility to automate a sequence of tasks, it may still be possible to integrate them into an overall automated solution.

While the DOS programs service does not provide MS-DOS applications any integration with the HP NewWave environment, it is a useful tool for many users. In many cases, users need access to stand-alone applications that are critical to day-to-day business activity, but don't require integration with other applications or task automation services. In time, these applications can be replaced with HP NewWave solutions, but in the interim, the DOS programs service provides a useful solution.

Generic Encapsulation

The DOS programs service provides a method to begin encapsulation, but it still requires the user to perform all the necessary file management using the MS-DOS file system. An important contribution of the HP NewWave environment is the ability of users to access and manage information easily by using a mouse to manipulate iconic representations of the data. The HP NewWave environment provides a facility called generic encapsulation, which allows many MS-DOS applications to be easily configured and accessed in a similar manner.

Using generic encapsulation, an MS-DOS application is installed as a unique object class. Data files created or accessed by the application are treated as instances of this class, and can be represented by individual icons within the HP NewWave environment. The user can access and

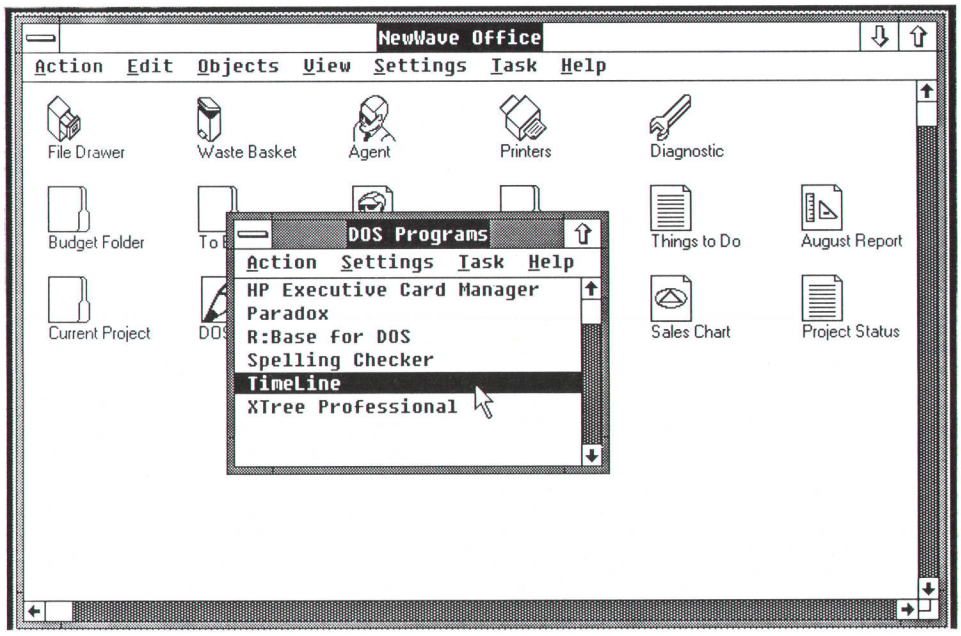
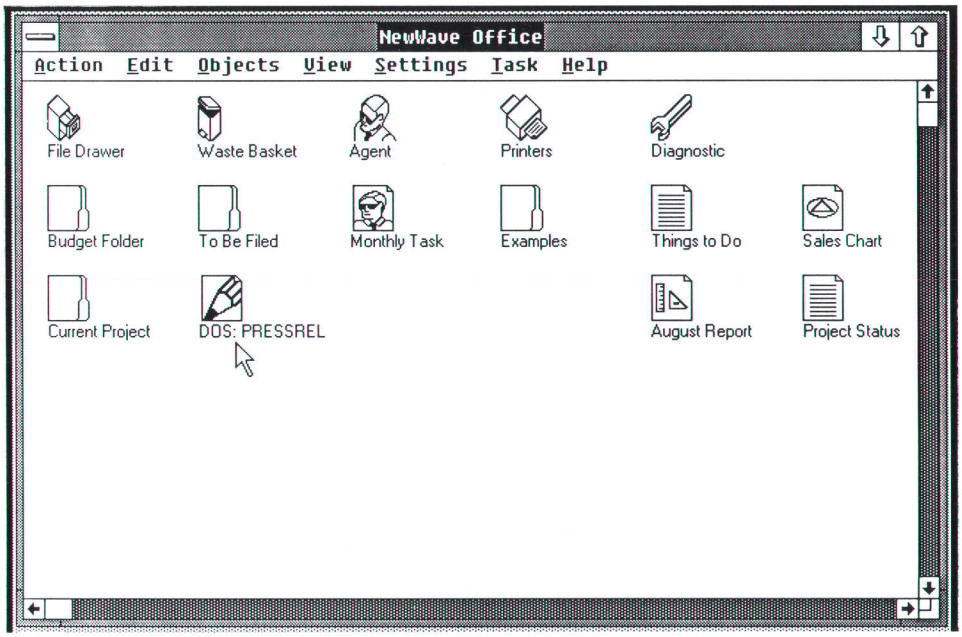


Fig. 2. *The DOS programs service menu.*

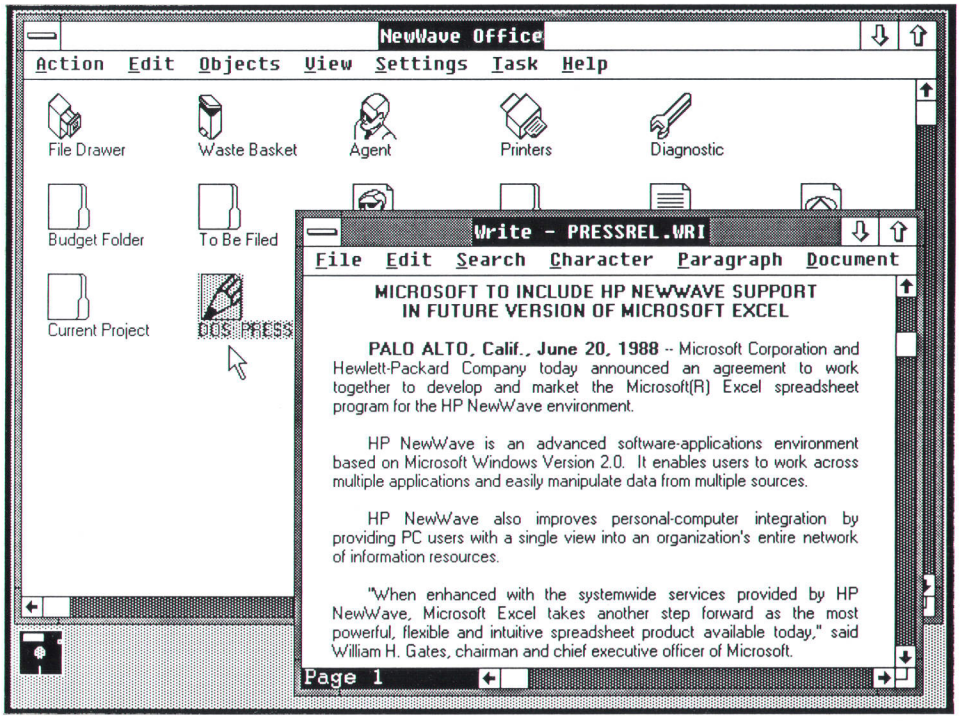
manipulate these data files like any HP NewWave object. Through direct manipulation with the mouse, the user can open an object and the generic encapsulation facility will start the associated encapsulated application and automatically load the data file associated with that instance. Fig. 3 illustrates direct manipulation of an encapsulated object. Direct manipulation can also be used to store, retrieve, move, copy, mail, or delete encapsulated application objects, just like other NewWave objects. These encapsulated objects can be organized and stored with other HP NewWave objects, allowing the user to manage all information in a similar fashion, and shielding the user from most details of the MS-DOS file system. The user can even create data file sharing at the object level, allowing the same object to be stored in multiple places at the same time.

Because these object-level features are implemented in a similar manner for a wide range of applications, the generic encapsulation facility is implemented as a single program that can be customized for different applications with a configuration file. The effort to encapsulate a new application is reduced from months of development time to a few minutes to set up the appropriate configuration information.

Generic encapsulation does not allow the user to estab-



(a)



(b)

Fig. 3. *The generic encapsulation facility allows the user to manipulate iconic representations of data. (a) An encapsulated MS-Write file* DOS:PRESSREL *and other objects in the NewWave Office. (b) Double clicking with the mouse on* DOS:PRESSREL *starts the application MS-Write and automatically loads the data file associated with* DOS:PRESSREL.

lish views to or from the application. Views are data links that establish the parent-child relationship of compound objects in the NewWave environment. Generic encapsulation also offers only limited support for the HP NewWave agent. These features are not possible without direct participation from the application itself, or a more sophisticated form of encapsulation that understands the specific data formats and operation of a particular application.

Objects versus Files. One of the significant contributions of the HP NewWave OMF is to manage access to the MS-DOS file system. While object data is stored in ordinary MS-DOS files, filenames are controlled by the OMF, not by the user. When the OMF starts an application program, it passes the MS-DOS file specification for the location of the application's data. The actual filenames are controlled by the OMF, and while known by the application, are never displayed to the user. This frees the user from the details of the MS-DOS file system, and allows the use of meaningful titles to identify individual objects. More important, it allows the user to create, copy, move, and mail compound objects, which are made up of multiple files, without worrying about creating individual names and resolving naming conflicts and collisions among the files.

For many applications it may require multiple files to store the information that makes up a specific object. For example, a data base application may require a form specification file, an index file, and the actual data base file to describe a data base completely. To accommodate this need, the OMF passes the application a fully qualified root filename (first eight characters with no extension) to specify the location of the OMF-managed data. Using the root filename, the application can create multiple files with different extensions, or even create nested subdirectories. The OMF will properly recognize all files in the nested structure when manipulating the object's data.

When encapsulating an unmodified MS-DOS application, it is often impossible to shield the user from all the details of the file system. For many applications, important functions, such as merging data, saving subsets, linking macros or scripts, translating files, or accessing individual files that make up the entire data set, depend on the user's specifying the MS-DOS filename. The encapsulation facility cannot hide these filenames from the user without severely limiting the capabilities of the application. Instead, the encapsulation facility must allow the user to specify the MS-DOS filenames used for each data instance and treat them as if they were the data files assigned and managed by the OMF. Resolving this dichotomy between the MS-DOS and OMF environments presents the biggest single challenge (and limitation) of encapsulation.

Since encapsulated applications rely on the MS-DOS file system to store and retrieve data, the user must specify a valid MS-DOS filename when creating a new encapsulated object. A complete MS-DOS file specification is made up of several parts (see Fig. 4). To simplify this process for the user, the encapsulation facility assigns a default subdirectory for each encapsulated application class. The user need only specify the eight-character root filename and the encapsulation facility provides default values for the drive and path of the assigned subdirectory as well as the predefined default file extension for the selected encapsulated application class. The appropriate template file (or files) is copied to the default subdirectory and the newly created object references this file instead of the file the OMF would automatically assign for a true NewWave application. An error will be indicated if the user does not specify a valid, unique filename.

Fig. 5 shows the directory organization for a typical NewWave system with encapsulated applications installed. There are three directories managed by the NewWave environment as well as other directories for application programs or user data unrelated to the NewWave environment. The OMF maintains the HPNWPROG directory for all NewWave executable programs and the HPNWDATA directory contains all the NewWave object data files as well as the OMF data base. To improve file access performance, the OMF automatically creates multiple subdirectories within HPNWDATA as required. There is rarely any reason for the user to directly access the files in the HPNWPROG or HPNWDATA directories. The encapsulation facility maintains the HPNWDOS directory which contains the data files for encapsulated applications. Each encapsulated application class is assigned its own subdirectory within the HPNWDOS directory and the filenames within these subdirectories are those assigned by the user when the encapsulated objects were created. By means of this well-defined directory structure, the user can locate and manipulate encapsulated data files even when the HP NewWave environment is not active. Since each encapsulated application may have its own specific requirements for how its executable program files are organized within a directory, the encapsulation facility does not attempt to manage these files directly. An encapsulated application program is installed on the system in its normal manner and the encapsulated application class definition provides the necessary information for the encapsulation facility to locate and access the application. This also allows the application to be accessed even if the HP NewWave environment is not active.

The encapsulation facility also supports references to files that already exist in the MS-DOS file system, files outside the default subdirectory, and an entire subdirectory of encapsulated data. The specific details of these facilities go beyond the scope of this article and are not necessary to understand the basic operation of encapsulation.

Once the MS-DOS file is encapsulated the user is free to move or share the object representation of the file anywhere within the OMF domain. It can reside in the NewWave Office, or be placed in any level of nested folders. Even though the object representation is moved within the OMF, the same MS-DOS file reference originally assigned by the user is still maintained. As with OMF data files, this

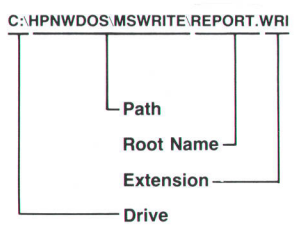


Fig. 4. *An MS-DOS file specification.*

filename may represent multiple files with different extensions, or a subdirectory structure of arbitrary complexity. The root filename is always displayed in place of the object's title. To minimize clutter on the screen, only the eight-character base portion of the filename is displayed by default. However, if the user selects this filename field for any specific object, the complete, fully qualified MS-DOS file specification is displayed.

Whenever the user copies or imports an object, a new file must be created. This happens transparently for OMF objects, since the OMF assigns the new filename. However, a new file cannot be created for an encapsulated object until the user provides a valid new filename. In many cases, this is simply not practical at the time the copy operation is performed. For example, when new mail arrives via an electronic mail system, objects and their associated files must be created to receive the incoming information. But at that time the user has no idea exactly what is being received, and therefore cannot intelligently decide what the filenames should be. Likewise, when copying a folder object that contains dozens of encapsulated objects in several levels of nested folders within it, it is not practical to expect the user to provide all the needed filenames. It should be equally obvious that it is simply not acceptable to prevent the user from copying, importing, or mailing encapsulated objects. The filename of the source object cannot be used without fear of collision with existing files.

The encapsulation facility solves this problem by recognizing that the data does not actually have to be in an MS-DOS file known to the user until the user is ready to open the object. Therefore, whenever an encapsulated object is copied, imported, or received via mail, the encapsulation facility allows the OMF to assign a filename for the copy. Rather than display this filename as the title (which would be meaningless to the user) it displays the string Copy of: followed by the eight-character root filename of the original object. The user can move, copy, share, delete, or mail this copied object, and the OMF will manage the associated data file as it does for any other object. The first time the user opens the object, the encapsulation facility will prompt the user for the required filename. Before starting the encapsulated application, the data is moved from the OMF-maintained file to the file identified by the user, and the object title is changed to reflect this new filename.

While this approach does have some limitations, it provides a practical method for encapsulated applications to behave like other HP NewWave objects, while still being able to access data using the MS-DOS file system.

Automatic Object Creation. A common feature of many applications is the ability to save data in a new file while the application is active. This allows the user to create multiple files, saving the current data in a new file and then continuing to make changes. In this case, the application itself is creating the data file, not the encapsulation facility or the OMF. But since the user expects the newly created file to appear as a new encapsulated object, the encapsulation facility monitors the creation of new files, and automatically creates an associated object with the required file reference.

Because of the difficulty of monitoring the internal operations of the MS-DOS file system, the encapsulation facility performs this automatic registration by comparing the con-

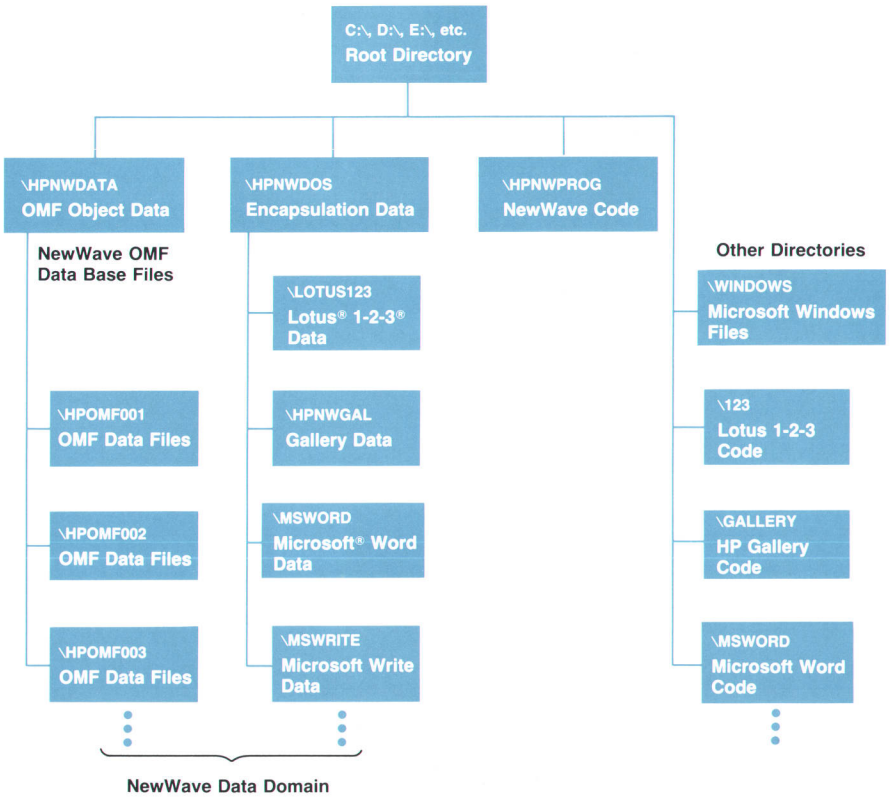


Fig. 5. *Organization of OMF and encapsulated file subdirectories.*

tents and date stamps of files in the default data directory before and after the encapsulated application is run. This search-and-compare operation is limited to the default data directory for the encapsulated object class, primarily for performance reasons.

Many applications are also able to create files compatible with other applications, for example, a spreadsheet that can store information in a form compatible with a data base program. In the case where both applications are encapsulated in the HP NewWave environment, the encapsulation facility can be configured to allow one application class to create objects of the other class, referencing the newly created files.

In some cases, applications specifically written for the HP NewWave environment provide a facility to convert or import information from MS-DOS files. The encapsulation facility can be configured to access this feature programmatically, providing a method to create the appropriate HP NewWave object automatically from information created by an encapsulated object.

Menus and Macros. While the encapsulation facility supports many automatic procedures, it may often require a less-than-intuitive sequence of steps within the encapsulated application to take advantage of it. For example, initiating an automatic conversion to an HP NewWave application requires saving the file in a predefined location, allowing the encapsulation facility to locate it easily. To make this feature easily available to the user, the encapsulation facility provides a method to create new pull-down menus for the encapsulated application. A keystroke macro is associated with each menu command and executed when the command is selected by the user. These macros can contain instance-specific, class-specific, or system-wide global variables.

For Microsoft Windows applications, these menu commands are added to the existing application menus. Microsoft Windows provides the programmatic facility to change menu configurations dynamically. By intercepting and filtering all Microsoft Windows messages received by the encapsulated application, the encapsulation facility can detect when one of its new menu commands is selected, and respond by sending a series of messages to the application that simulate the associated macro being entered via the keyboard.

For full-screen applications, the encapsulation facility monitors the interrupt service routines used to manage the keyboard. When the appropriate activation key is detected, the added menu commands overlay the full-screen application's display. The keyboard interrupt service routine is monitored to determine what menu command is selected, and the appropriate macro is sent to the application through the interrupt service routine, simulating keyboard input.

Configurable menus and keystroke macros offer a powerful mechanism for extending the user interface of encapsulated applications, and provide an easy way for the user to access the additional features provided by the encapsulation facility.

Agent Support. The encapsulation facility provides keystroke recording and playback to extend the functions of the agent facility to support encapsulated applications. This is implemented using the same windows message or keyboard service interrupt filtering used for the configurable menu and macro feature. While this solution is not without significant limitations, it does provide the basic capabilities required to automate system-wide tasks that also include encapsulated applications.

Configuration Parameters. Generic encapsulation supports a wide variety of applications, using a sequence of configuration parameters to provide the detailed information related to each program. Following are some of the parame-

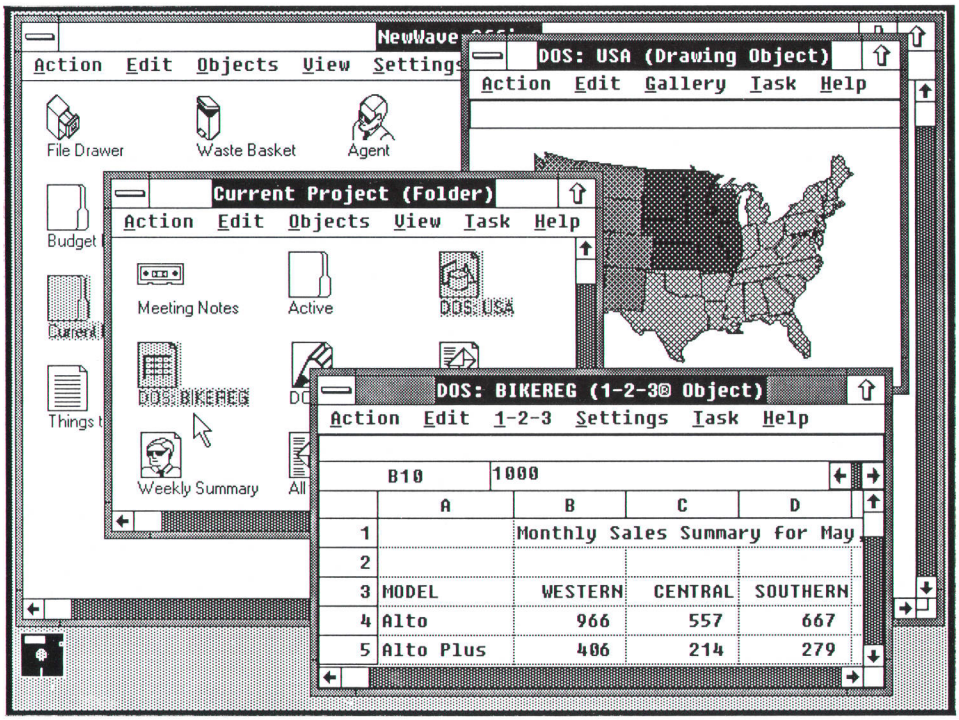


Fig. 6. *Application-specific encapsulation for HP Drawing Gallery and Lotus 1-2-3 provides NewWave browsers, allowing the application's data to be viewed and linked with other NewWave applications.*

ters that provide the generic encapsulation facility with the information it needs to know about the application being encapsulated.

- Application Type. Specifies whether the encapsulated application is a Microsoft Windows application or a full-screen application.
- File Access Method. Specifies whether data for this application will be stored in referenced MS-DOS files, or in OMF-controlled files. The latter are useful when the encapsulated application can be modified to remove dependencies on users to specify the filenames for data files.
- Application Name. The subdirectory location and name of the encapsulated application.
- Command Line. The command line parameters to be passed to the application when it is started. This is the most common method for providing the instance-specific filename to the application.
- Current Directory. The drive and subdirectory that should be selected as the current directory before the application is started. Some applications require that the directory containing the main program be the current directory. For others, making the default data directory the current directory eases access to additional data files.
- Menu File. The name of the optional file that defines the added menu commands and their associated keystroke macros.
- Window Class. For Microsoft Windows applications, this specifies the indentifier used by Microsoft Windows to identify the application's window type.
- Window Title. The text identifier maintained by Microsoft Windows that identifies a window and is displayed in the window's title bar. This field and the preceding one are used by the encapsulation shell to locate the application's window when it is initiated. The location is needed to configure menus and macros and filter the application's messages.
- Key Extension. The three-character filename extension (e.g., .GAL for Drawing Gallery files) that is searched for to determine if new files for the application have been created and need to be automatically registered. The key extension is also used for error and filename collision detection when the user provides new filenames.
- Required Extensions. Specifies the extensions of other files that must be present to make up a valid data instance. This is also used for error and filename collision detection.
- Data Directory. The default data directory for the application. This is where all files for this application class are typically stored.
- Export Classes. Specifies the classnames of other encapsulated applications that may be created by this application.
- Conversion Classes. Specifies the classnames of HP NewWave applications that support programmatic conversion and can be accessed by this application.

These and other configuration parameters are specified when the encapsulated application class is installed in the HP NewWave environment. They are stored as text in the class properties for the application, and can therefore be programmatically accessed by any other application that needs this information. Like the configurable keystroke macros, the configuration parameters can include a variety of instance-specific, class-specific, or global system variables, providing a tremendous degree of flexibility. Using variables, any text property of any other application class can be accessed. Because the OMF property system is extensible, developers can specify additional properties and access this information as configuration information or keystroke macros from multiple, cooperating applications.

Application-Specific Encapsulation

While generic encapsulation makes it possible for many applications to operate as part of the HP NewWave environment, it does not support one of the most important capabilities: views. For applications to share data cooperatively, the encapsulation facility must not only know how to communicate with other applications, but must also understand and be able to modify the application's data structure. It must also provide additional commands and dialogs to manage the links that are created. This invariably requires the development of a specialized encapsulation program that recognizes the specific features and data formats of the application it encapsulates. The solutions are as varied as the applications that can be encapsulated. With enough effort invested in the encapsulation shell, virtually all HP NewWave capabilities can be supported. But this process may entail more effort than simply rewriting the application for the HP NewWave environment.

To encapsulate Lotus® 1-2-3®, a spreadsheet program from Lotus Development Corporation, an HP NewWave browser application was developed (see Fig. 6). This program reads the Lotus 1-2-3 spreadsheet data file, displaying its contents in the Microsoft Windows environment. The encapsulation browser provides the necessary user commands and interfaces to the OMF to allow ranges of the Lotus 1-2-3 worksheet to be viewed in other HP NewWave applications. To minimize the browser's complexity, it is only capable of reading the Lotus 1-2-3 worksheet file; it cannot make changes to it. Instead, the browser provides a command to start the Lotus 1-2-3 program, automatically loading the data file being browsed. When the user exits from Lotus 1-2-3, the updated file is redisplayed by the browser. Because the browser does not alter the spreadsheet data, it will not accept data passing views* from other applications. A similar browser-type encapsulation shell was developed for HP Graphics Gallery.

Conclusion

HP NewWave's encapsulation services provide the bridge from today's MS-DOS-based applications to the next generation of applications developed for the HP NewWave environment. The DOS programs service and generic encapsulation provide the facilities for users to take advantage of HP NewWave immediately, while continuing to use their current suite of applications. Application-specific encapsulation provides an interim solution to allow developers to move their existing applications into the HP NewWave environment and take advantage of its advanced features without requiring a complete application rewrite. The HP

*A data passing view is a data link between objects that allows the child to pass data to the parent.

NewWave environment clearly defines the applications environment of the future, and the complete range of encapsulation services provides a clear, well-lighted path for today's personal computer users.

Acknowledgments

Yitzchak Ehrlich is responsible for much of the design and implementation for HP NewWave encapsulation technology and the generic encapsulation facility. Tony Day designed and implemented the DOS programs service. Scott Hanson designed and implemented the configuration utility used to install generic encapsulation applications. Doug Smith implemented a shared library of DOS file management routines used by all encapsulation programs. Andy Dysart and Chuck Whelan implemented new facilities in the OMF to support encapsulated applications. Several enhancements to support encapsulation features were made to MS Windows by Tom Battle's team at HP's Sunnyvale Personal Computer Operation. Many more engineers at HP's Personal Software Division have contributed to the design and implementation of the encapsulation services for the HP NewWave environment.

Authors

August 1989

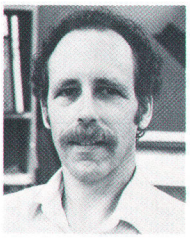
6 NewWave Overview

Ian J. Fuller

On the NewWave project, Ian Fuller served as project manager for the NewWave Office, OMF, and DOS application support. He has since become project manager for distributed object-based systems. Ian joined HP's Office Productivity Division in his native England in 1980. Work as an engineer and as a project manager on HPDeskManager and AdvanceLink were among his early assignments. Before coming to HP, he was an engineer on message switches and telephone exchanges for ITT Business Systems. Ian was born in Gosport, Hampshire, and attended Oxford Polytechnic, where he received his BSc degree in 1978. Describing himself as an "adopted Californian," he now resides in Santa Clara, California, and is recently married. As his leisure activities, he likes travel and photography.

9 NewWave User Interface

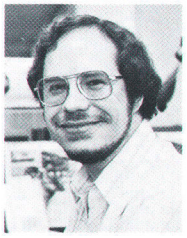
Peter S. Showman



With HP since 1967, Pete Showman managed the design committees responsible for the NewWave environment's architecture, user model, and user interface specifications. He has since moved on to projects involving the definition of future application environments and architectures. In past assignments, he has worked as a hardware engineer and project manager on such HP products as the HP 8542A Network Analyzer, the HP 8500 Graphics System console, the HP 2648 and HP 2700 graphics terminals, and the HP 150 Touchscreen PC. Pete is a member of the IEEE and the ACM. His BSEE degree is from Cornell University (1965) and his MSEE degree is from the Massachusetts Institute of Technology (1967). He has authored or coauthored two previous articles for the HP Journal. Pete was born in Madison, Wisconsin. He, his wife, and his two teenage sons live in Cupertino, California. Among his varied hobbies are woodworking, recreational computer programming, skiing, and playing old-time music on a fiddle and other instruments. He has also been studying Chinese for two years.

17 NewWave Object Management

John A. Dysart



Andy Dysart came to HP in 1982, after receiving his bachelor's degree in computer science from the New Mexico Institute of Mining and Technology. After initially serving on the NewWave user interface design committee, he worked on design of the object management facility. He continues to contribute to NewWave design. Andy is a coinventor on two pending patents for the NewWave OMF. Past responsibilities include design and implementation of HP ExecuDesk and HP FormsMaster software. He is a member of the ACM and considers software architectures, object-based systems, and productivity his professional specialties. Born in Temple, Texas, Andy is married and lives in Santa Clara, California. He likes the outdoors, reading science fiction, and what he calls "recreational computing."

23 NewWave Office

Scott A. Hanson

Scott Hanson worked on the screen display and data structures of the NewWave Office facility. As a development engineer, some aspects of the project continue to be his responsibility. In the past, he contributed to development of VisiCalc software for both HP 150 and HP 3000 Computers. Scott attended the University of California at Davis, where in 1983 he received his BS degree in mathematics and computer science. He joined HP the same year. Born in Sacramento, California, he lives in Sunnyvale, California. Scott spends his leisure time scuba diving, jogging, or reading science fiction.

Anthony J. Day

NewWave desktop and creator facilities and DOS programs have been Tony Day's focus of responsibility as a development engineer. He joined HP in 1984, shortly after receiving his BA degree in computer science from the University of California at Santa Cruz. Software reliability, reuse, and project control are his professional interests. In a previous career, he worked for the U.S. Navy, first as personnel manager in London, England, then as budget and accounting manager in Monterey, California. Tony was born in London and lives in San Jose, California. He's married and has a child. He spends his off-hours with recreational computer programming and gardening.

Beatrice Lam

The Office facility was Bea Lam's focal interest in the development of the NewWave environment, and she has since become project manager for NewWave architecture components. In the years since she joined HP in 1980, she has worked as development engineer on software projects such as HPWord and Executive MemoMaker for the HP Vectra PC. Bea earned her BSEE degree at Cornell University (1973) and her MSEE at Purdue University (1974). Previous professional experience includes firmware design at Control Data Corporation and a position as translator and coordinator for the National Council on U.S.-China Trade. Born in Canton, China, Bea is married and lives in Sunnyvale, California. Her diverse recreational interests include opera, sophisticated audio equipment, bridge, Chinese cooking, restaurant sampling and critique, and learning Japanese.

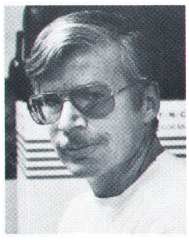
32 Application Program Interface

Glenn R. Stearns

Glenn Stearns' professional interests focus on autonomous systems, software architectures, and artificial intelligence. He was a software engineer on the NewWave project and became a project manager at the time of its release. Before joining HP in 1984, his professional activities involved mini- and microprocessors, multi-CPU applications, radio data communications, and PC environments. Glenn is the named inventor on one software patent and a coinventor on two others. Studying computer science, philosophy, and psychology, he attended California State University at Hayward for four years. He is a member of the ACM and the American Association for Artificial Intelligence. Born in New York, he is married, has a daughter, and lives in Scotts Valley, California. Motorcycles and philosophy provide his off-hours recreation.

38 Agent Task Language

Charles H. Whelan



Microcomputer systems software is Chuck Whelan's main professional interest, and his contributions to the NewWave project focused on the object management facility, agent recorder, and diagnostic utility. Since joining HP in 1973, his projects have included the development of RTE-L and RTE-VI operating systems for the HP 1000 Computer, DS/1000 networking software for the HP 1000, and BIOS for the HP 150 PC. Chuck's BA degree in mathematics is from Oregon State University (1964). Born in New York, he is married, has four children, and lives in Placerville, California.

Barbara B. Packard

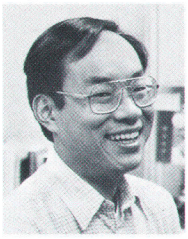
A development engineer at the Personal Software Division, Barbara Packard's NewWave assignments included NewWave agents and task language compilers. She came to HP in 1973 and spent seven years working on the COBOL compiler and COBOL toolset for the HP 3000 Computer, partly as project manager. She also served as project manager for a cross-Pascal compiler and MemoMaker software developments. Before coming to HP, Barbara was an aeronautical research engineer for the U.S. National Aeronautics and Space Administration. Her BS degree in mathematics is from Stanford University (1954), as are her two master's degrees, one in mathematics (1955) and one in computer science/computer engineering (1977). She is a member of the ACM and the American Association for Artificial Intelligence. Barbara was born in Orange, California, and lives in Los Altos Hills, California. She is married and has four children; one of her daughters is an application engineering manager for HP. Her hobbies include hiking, birdwatching, and traveling. Recent trips took her to the Himalayas in Nepal and on safari in Kenya and Tanzania.

43 NewWave Help Facility

Vicky Spilman

As a software engineer at the Personal Software Division, Vicky Spilman worked on the help system for the NewWave environment. She came to HP in 1982, soon after graduating from California State University at Chico with a BS degree in computer science. In the past, Vicky has been involved with graphics systems and independent-software-vendor products as a support engineer and has worked on software for the HP 150 Personal Computer. Vicky lives in Sunnyvale, California, and enjoys gardening, aerobics, and bicycling.

Eugene J. Wong



Eugene Wong was project manager for the NewWave help facility, formatter, builds, and system performance and installation. He also served as system manager for the NewWave developer system. In past assignments, he has served as engineer and as project manager for automatic test systems and real-time systems. He also managed the development of third-party software ports to the HP 150 Personal Computer. Eugene's BA degree in mathematics is from San Francisco State College; he came to HP after receiving his degree in 1970. He is a member of the ACM and the IEEE. The HP 1000 RTE-IV operating system is the subject of a previous article he has written for the HP Journal. Eugene was born in Stockton, California, and lives in Cupertino, California. He is married and has two daughters. In his off-hours, he likes to play go, read, and study medieval calligraphy.

48 Computer-Based Training

Lawrence A. Lynch-Freshner

A software engineer assigned to the NewWave project, Larry Lynch-Freshner focused his attention on design and implementation of the animation and computer-based training display functions. In projects before he joined the Personal Software Division, he supported the operating systems for HP 150 and HP Vectra PCs. He studied computer science at Oregon State University and came to HP in 1983. He is a member of the ACM, SIGPLAN, and SIGGRAPH, and his professional interests include computer languages, computer graphics, and windowing systems. Larry has served in the U.S. Air Force Reserve. He was born in Salem, Oregon, and makes his home in Mountain View, California. He is married and has a son. Among his many avocations are beer brewing, ancient history and ancients war-gaming, electronics, robotics, science fiction, and jewelry-making.

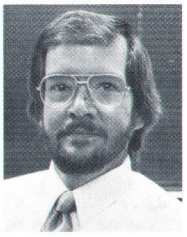
R. Thomas Watson

As software development engineer, Tom Watson was responsible for the agent facility for computer-based training, and he continues to be involved with NewWave development. His past professional experience includes positions as computer sales manager and as computer programmer. He came to HP in 1984. His BS degree in computer science is from Pennsylvania State University. He is a member of the ACM. Tom was born in Upper Darby, Pennsylvania, is married, and lives in San Jose, California. His after-hours interest is synthesized music.

John J. Jencek

The most recent addition to the NewWave CBT development team, John Jencek joined HP in June 1988 as a software engineer. The computer-based training software was his first assignment. He developed the parsers and the CBT menu object and continues to work on NewWave design objectives. His previous experience includes work as computer programmer at IBM Corporation and as instructor at Texas Instruments. John was born in Prague, Czechoslovakia. He attended the California State University at San Francisco, where he earned his BS degree in computer science (1988). He lives in San Francisco, California, and enjoys scuba diving in his off-hours.

Brian B. Egan

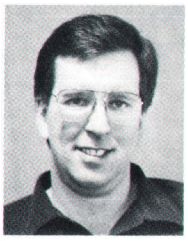


Brian Egan is product manager for interactive learning systems at HP's Personal Software Division. His role in the development of the NewWave environment was that of manager and editor of the computer-based training software and courseware. His past positions include publications manager, support manager, and customer engineer. His professional interests focus on computer-based and classroom instruction, user interfaces, and teaching. Brian's

BS degree in computer science engineering is from California State University at San Jose, earned after seventeen years of attending night school. He served seven years in the U.S. Air Force, where he was a technician and instructor in metrology. He was born in New York City, is married, and has two children. Brian lives in San Jose, California. As an avocation, he teaches writing classes for engineers at California State University. He also likes hiking and mountain bicycling, and plays bass guitar in a rock band.

57 Encapsulation of Applications

William M. Crow



As an R&D project manager on the NewWave project, Bill Crow was responsible for OMF and Office software and the generic encapsulation. He continues to serve as project manager on other NewWave assignments. He attended the University of Vermont where in 1974 he received his BS degree in mathematics. He joined HP's Personal Software Division in 1984, where his responsibilities included the development of graphics products for the HP 3000 Computer. Bill's past professional experience includes positions as director of computer systems at The Austin Company and as software designer for an aerospace company. He has authored numerous papers and articles about data communications, office automation, and personal computers. He is named coinventor in two patents relating to navigational systems and three pending patents on the NewWave environment. Bill was born in Newark, New Jersey, is married, and lives in San Jose, California. His major hobby is personal computers.

67 Tape Drive Design

Andrew D. Topham

Tape head and cartridge technology were Andy Topham's focal points as an R&D engineer on the HP 9145A project. He has been a manufacturing engineer on a similar product, the HP 9144A, and is now responsible for a new product involving a tape drive. Before he joined HP in 1985, he worked for Racal Research Ltd. as an R&D engineer for digital signal processing and for Research Machines Ltd. on the design of microcomputers. The tape head mounting system described in this issue of the HP Journal is the subject of a pending patent that names Andy as a coinventor. He obtained his degree in physics at the Imperial College in London in 1981 and is an associate member of the IEE. Born in Birmingham, he now resides in Dursley, Gloucestershire. He is married and has an infant son. His hobbies include boardsailing, gardening, running, and photography.

74 Tape Drive Reliability

David Gills

For over four years, Dave Gills has been a reliability engineer and has worked in both R&D and quality control departments in HP's Bristol facility. He was the project reliability engineer for the HP 9145A and has since moved to a new digital audio tape project. His past assignments include the HP 35401A Cartridge Tape Drive. In his earlier career, Dave was a range and flight trials engineer working on guided weapons for the British aerospace company. Some years before graduating from Coventry Polytechnic with a BSc degree in 1983, he served a four-year mechanical-engineering apprenticeship with ICI Fibres Ltd. He is an associate member of the Institute of Mechanical Engineers and a member of the Safety and Reliability Society. Dave was born in Cheltenham, is married, and has an infant son. He lives in Dursley, Gloucestershire. Golf is his favorite pastime.

82 Real-Time Peripheral Firmware

Tracey A. Hains

As an R&D engineer on the HP 9145A, Tracey Hains' responsibilities included analysis, design, and development of firmware for the device task and the operating system. She has since begun designing the digital data storage format for new products. Other assignments Tracey has worked on include the design of software for a networked backup product. She came to HP in 1985, after receiving her BSc degree in mathematics and computer science from the University of Bristol. A pending patent describes algorithms Tracey originated. She was born in Dorset, is married, and lives in Bristol.

Mark J. Simms

Design, test, and debugging of the buffer management software for the HP 9145A Cartridge Tape Drive was Mark Simms' responsibility. A software engineer at the Computer Peripherals Bristol facility, his cognizance now includes the overall analysis and architectural design for other tape drives and the design of buffer management software. In past assignments, he was responsible for the data spooling software used in earlier products. Two patents are based on Mark's ideas, one for remote backup software and another for a file system search method. He received his BSc degree in computer science/mathematics from Bristol University in 1984, the same year he joined HP. Born in Leeds, he is married and lives in Bristol.

Paul F. Bartlett

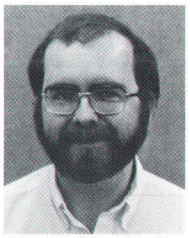
As an R&D software quality engineer, Paul Bartlett was responsible for the design, implementation and testing of the HP-IB interface handling process used with the HP 9145A. He is now working on the development of a process aimed at assuring quality and reliability of firmware design. Before coming to HP in 1985, Paul designed telephone switching systems software for C.E.C. Telecommunications and software for mobile radio applications for Pye Telecommunications. He is named inventor in a pending patent describing algorithms used in remote backup software. Paul received his BSc degree in mathematics from the Imperial College in 1977, and he is a member of the British Computer Society. He was born in Aldershot and lives in Bristol, the home of HP's computer peripherals facility. He's married and has three children, a boy and twin girls. His hobbies include bicycling and photography.

Paul F. Robinson

As one of the project managers for the HP 9145A Cartridge Tape Drive, Paul Robinson was responsible for firmware development. In a previous assignment, he worked as an R&D engineer on a cost reduction project. Before joining the HP Computer Peripherals Bristol Division in 1985, he designed CAD software for a number of electronics companies, among them Phillips and Racal Corporations. Paul studied computer science at the Loughborough University of Technology and is a member of the British Computer Society. His professional interests focus on software methods, metrics, and R&D processes.

87 Object-Oriented Software Technology

Thomas F. Kraemer



Tom Kraemer was the project manager of the HP Vista software development at HP's Lake Stevens facility. He has contributed to the development of many HP calculator, computer, and instrument products as an engineer, project manager, and section manager. Currently a section manager in HP's Logic Systems Division, he is responsible for R&D on HP Teamwork SA/SD and HP 64700 emulator products. Tom joined HP in 1978, after earning his MSEE degree from Oregon State University, where he also worked on a U.S. Navy research program. Before starting college, he was a professional animator and became interested in applying computer technology to animation. He is married and resides in Sunnyvale, California.

Mechanical Design of a New Quarter-Inch Cartridge Tape Drive

The design of the HP 9145A Tape Drive required doubling both the track density and the tape speed of the existing HP 9144A, thereby doubling the older drive's 67-Mbyte capacity and 2-Mbyte-per-minute transfer rate.

by Andrew D. Topham

THE EVER-INCREASING VOLUMES OF DATA being handled by computer systems make it mandatory for backup tape devices to continue to match the growing disc capacities being projected. Both data transfer rate and tape cartridge capacity must continually be improved.

The HP 9145A ¼-Inch Cartridge Tape Drive (Fig. 1) was developed in response to this need. Before the HP 9145A was developed, HP's entry level and midrange commercial computer systems and technical workstations were usually configured with either an HP 9144A Tape Drive or an HP 35401A Autochanger for backup, depending on disc capacity. The HP 9144A has a cartridge capacity of 67 Mbytes and a data transfer rate of 2 Mbytes per minute. The autochanger uses the same mechanism and has the same transfer rate, but achieves a capacity of 536 Mbytes by changing eight tape cartridges without operator attention.

The HP 9145A provides full compatibility with the HP 9144A and the HP 35401A while also providing twice the data transfer rate. This is achieved by doubling the tape speed from 60 to 120 inches per second. As a result, users can back up their systems in half the time.

The HP 9145A has twice the data capacity per cartridge of the HP 9144A. This is achieved by doubling the number of recording tracks from 16 to 32. The HP 9145A can read the older 16-track tapes, but the older drives cannot read the new 32-track tapes.

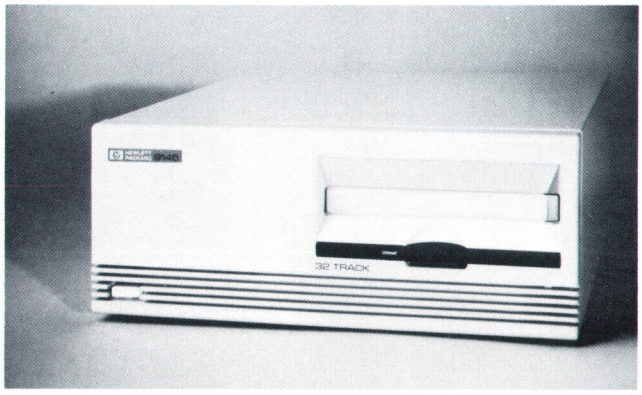


Fig. 1. *The HP 9145A ¼-Inch Tape Drive provides a storage capacity of 133 Mbytes per cartridge and a data transfer rate of 4 Mbytes per minute for backing up disc memory in entry level and midrange computer systems.*

Technical Challenges

When the development team started the task of designing the HP 9145A, there were several key areas where major design changes were required.

Mechanical Design. To achieve the increased capacity, the number of tracks across the tape width had to be doubled within the same ¼-inch tape width. To achieve the increased data transfer rate, the tape speed had to be doubled. The design had to accommodate variations in components, manufacturing processes, and operating environments and remain capable of accurately positioning the read/write

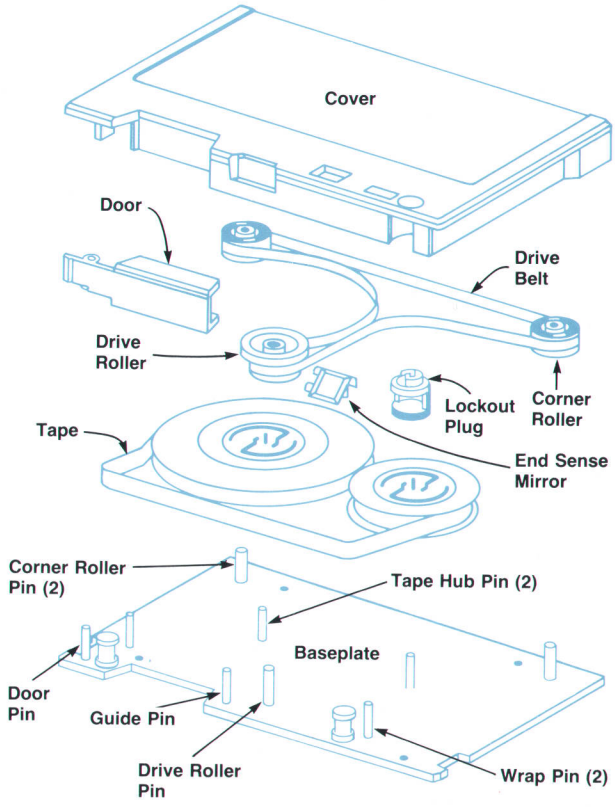


Fig. 2. *Exploded view of the new HP 92245L/S cartridge.*

head within the data tracks to guarantee data reliability.

New Cartridge. An improved cartridge design had to be introduced to support the increased tape speed and capacity requirements. This implied complete qualification and testing as well as the setup of formatting and certification lines by the cartridge manufacturers. At the same time, to maintain compatibility, the drive design had to guarantee that HP 9144A-written tapes could be read. The new cartridge features an improved mechanical design and new tape media. The tape offers higher reliability with a new oxide formulation, which reduces the signal decay that occurs each time the cartridge is used. The cartridge has a new belt and corner rollers to accommodate the increased tape speed, and an extra tape guide for better read/write accuracy.

Increased Reliability. The HP 9145A had to satisfy the user needs that had been identified. This required designing to much tighter tolerances and higher performance. At the same time, we had to ensure that the new product incorporated the lessons learned from the existing line and product range with regard to reliability and manufacturability. Reliability issues are discussed in the article on page 74.

Time to Market. To meet market needs, reliable prototypes of the HP 9145A had to be ready for testing with the target computer systems in under 12 months. Thus the design team had less than a year to design hardware and firmware from concept to reliable implementation.

HP 9144A Design

The HP 9144A Tape Drive's tape transport mechanism has design concepts common to all ¼-inch cartridge tape drives. The cartridge itself (Fig. 2) provides a reference plane in the form of the cartridge baseplate against which the tape path and servo interface are closely aligned. The drive takes advantage of this by clamping the baseplate against accurately defined stops within the mechanism. This ensures that the tape path aligns precisely with the tape head magnetic cores used to read data from and write to the tape, and that the servo motor puck aligns with the drive roller within the cartridge.

Sixteen tracks of data are written across the quarter inch of tape width. To read and write each of these tracks independently, the tape head in the drive is driven vertically by a stepper motor and leadscrew arrangement.

HP 9145A Improvements

Because the development cycle had to be short and the HP 9144A design offered a good starting point for many of the design requirements, it was decided to leverage off this product as much as possible. This approach was particularly pronounced in the mechanism area where, for example, the casting used to align all the mechanical components and the cartridge clamping assembly were left totally unchanged. Fig. 3 shows the HP 9145A drive mechanism with its associated electronics removed.

The development of an enhanced ¼-inch cartridge from the cartridge manufacturer, dubbed the HP 92245L/S, made possible the doubling of the track density. Evaluation of this cartridge was in itself a major task which was run in parallel with the drive development.

Tape Speed

The HP 9144A data transfer rate was identified as a priority area to be improved upon. With the HP 9145A providing double the cartridge capacity, keeping the data transfer rate constant would have resulted in a doubling of the time for

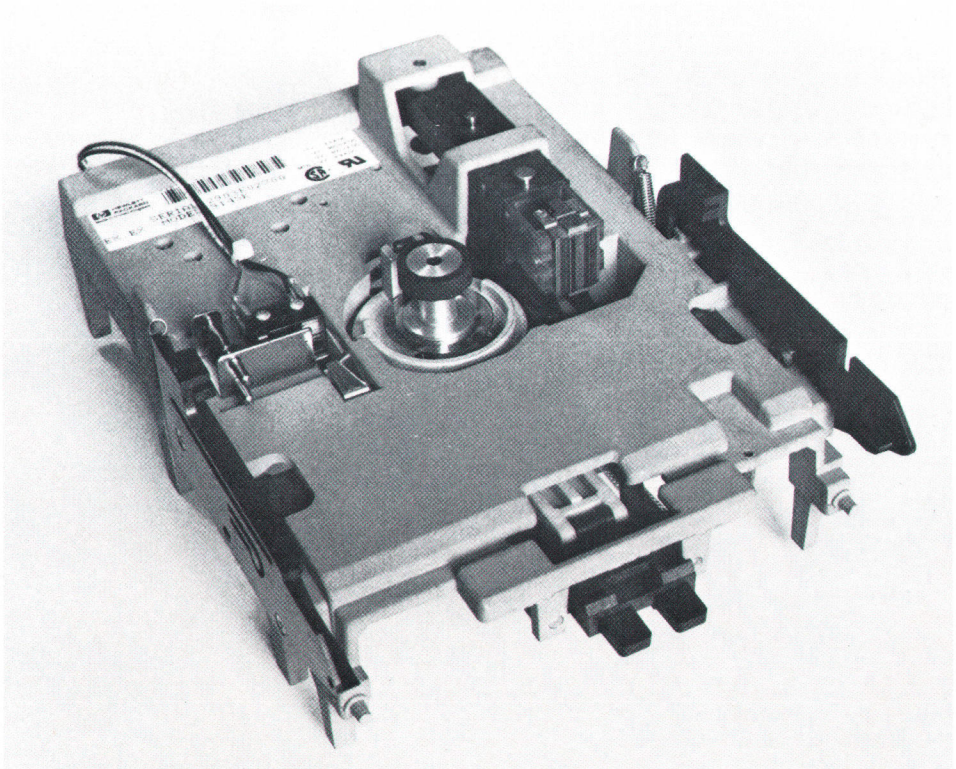


Fig. 3. *HP 9145A drive mechanism.*

reading or writing a cartridge.

One approach that could have been taken would have been to increase the linear transition density of the data on the media, resulting in a correspondingly higher data rate for a constant tape speed. However, this was rejected for two reasons. First, it would have led to complications in the read channel filtering and data recovery side of the drive, because of the need to continue to be able to read HP 9144A data with its lower transition density. Second, the media had not been proved able to perform sufficiently well at the higher transition density. The cartridge manufacturer was actively evaluating the media for this density, but there would inevitably have been an increased risk to the project.

The approach taken to improve the data transfer rate was to double the tape speed while keeping the transition density the same as in the HP 9144A drive. This led to a twofold transfer rate improvement, and took the drive from the 60 ips (inches per second) used by the HP 9144A to 120 ips.

Running the tape at this speed raised some technical concerns about the cartridge. Would the mechanics of the cartridge be able to handle this speed without either instabilities in the tape transport or degradation of cartridge operating lifetime? Would an air bearing form between the head and the tape, causing signal loss?

Cartridge Mechanics

The new HP 92245L/S cartridge was developed by the cartridge manufacturer with one of its major goals being continuous, reliable operation at a tape speed of 120 ips. During the testing of the HP 9145A drive the design team was able to provide valuable feedback to the cartridge manufacturer on the performance of the cartridge, with the result that several design modifications were made to boost the long-term reliability.

Critical parameters in the cartridge evaluation were the tape tension, the drive force, and acoustic noise. The tape tension had to be high enough to prevent the formation of an air bearing between the head and the tape, and yet low enough to prevent excessive head wear and hence short drive lifetimes. The drive force (the drag applied by the cartridge on the servo motor) had to be sufficiently low that the servo drive motor and associated control electronics that control the tape speed at 120 ips were not unduly stressed. Doubling the tape speed was found to have a substantial effect on the audible noise emitted from the cartridge. The HP 9144A and HP 9145A drives are bound by the HP specification for office environment operation and so have to satisfy a very low noise requirement. A joint development program, with the drive design team supplying test data to the manufacturer concerning the noise emissions from the cartridge in the HP 9145A drive, allowed the cartridge manufacturer to modify the cartridge to bring the noise level down to an acceptable level (Fig. 4).

The overall result of the work that went into solving all the above problems was that the HP 92245L/S cartridge has emerged as a substantial improvement over its predecessor. Many of the changes that have been implemented in the HP 92245L/S cartridge are now being adopted for the HP 9144A cartridge.

There was concern whether older cartridges used in HP 9144A drives could be read in the HP 9145A drive. These had only been rated at a maximum tape speed of 90 ips by the cartridge manufacturer. However, an extensive testing program during the development of the HP 9145A drive confirmed initial indications that these cartridges were very conservatively rated, and the majority performed well in the test program. In a very small number of cases there was some cause for concern over the longer-term use of such cartridges at 120 ips. An unacceptable increase in drive force could occur after running continuously for several days at the maximum rated operating temperature. This problem was avoided by specifying that the new drive would only be required to offload data from an HP 9144A cartridge once. This is backed up by instructions to this effect in the user manual.

Head-to-Tape Contact

Intimate contact between the tape head and the media is essential in any tape drive to provide maximum read signal and minimum distortion. Head-to-tape contact is dependent on three factors:

- Tape speed. A higher speed produces greater spacing.
- Tape tension. Higher tension pulls the tape closer to the head.

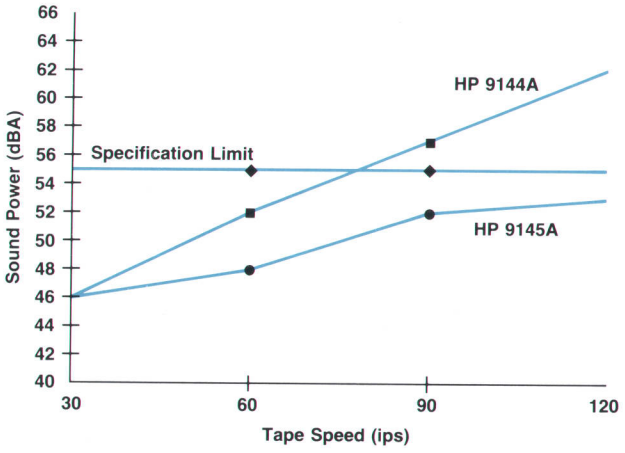


Fig. 4. *Noise level of the new HP 92245L/S cartridge compared with its predecessor.*

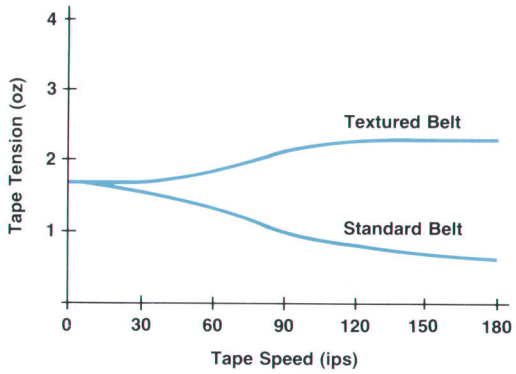


Fig. 5. *Steady-state tape tension of the HP 92245L/S data cartridge, comparing the improved textured drive belt with the standard belt used in older cartridges.*

- Head profile. The shape of the front face of the head in contact with the tape has a complex effect on the tape spacing.

The development program for the HP 92245L/S cartridge resulted in that cartridge having such an excellent tape tension characteristic that head-to-media separation is not a problem, even at 120 ips (Fig. 5).

The testing program showed that for the vast majority of the older HP 9144A cartridges, there was no problem in running at 120 ips. A very small number of exceptions to this rule were found. The problem cartridges were from a few batches that the cartridge manufacturer was able to trace back to a time when there had been minor problems in the cartridge production process. These cartridges exhibited very low tape tension so that, at 120 ips, there was a tendency for the tape to lift off the read/write head slightly. This led to reduced read signal amplitude (spacing loss) and so occasionally to read errors.

One attempt to keep the spacing loss to a minimum was through experimenting with the tape head profile. This profile must be accurately designed and machined to offer a surface that does not abrade the media, will withstand a lifetime's use, and maintains intimate contact with the media. The wear requirement and the intimate contact requirement tend to favor opposing profile shapes, so that any solution is inevitably a compromise between the two. Some experimentation with profiles that exhibited radii both sharper and more gentle than that used on the HP 9144A tape head showed that the existing profile, as shown in Fig. 6, was a good approximation to the ideal. The adoption of this profile removed another risk area in that this profile is already well-understood and in large-scale production.

The spacing loss problems were overcome by building into the drive a 90-ips read mode option. Dropping the speed causes the tape to drop closer to the head, thereby improving the read signal. This option is automatically invoked by the drive when it detects that errors are occurring because of the above phenomenon. Extensive testing has proved the capability of this approach.

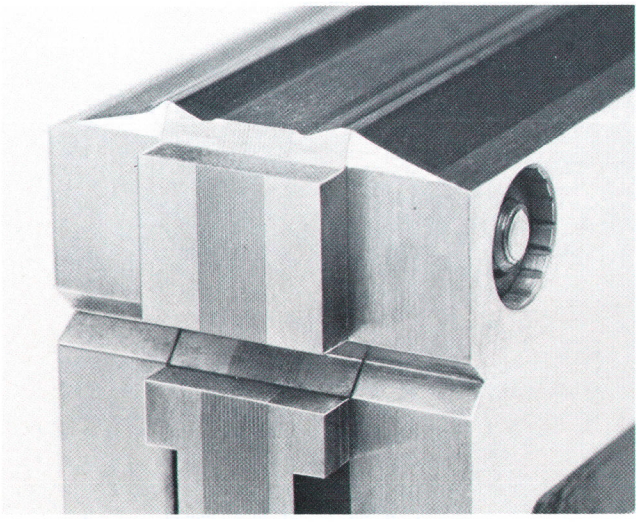


Fig. 6. *HP 9145A tape head profile.*

Track Density

The HP 9144A drive places 16 tracks across a ¼-inch media width. To double the capacity of the drive without increasing the linear bit density it was necessary to fit 32 tracks across the media. This was achieved by:

- Reduced written track width
- Improved head positioning accuracy
- Improved cartridge tracking specifications
- Improved cartridge media defect specifications
- A new track layout
- Track seeking
- Improved core alignment
- Improved tape head mounting.

Track Width. The track width of a tape drive is defined by the width of the magnetic core within the tape head. Both the HP 9144A and the HP 9145A use a system of wide write, narrow read, whereby the read core width is less than the width of the written track. This ensures that the read core will be over the written track even if there are positional errors between the core and the center of the track (Fig. 7). If the read core falls outside the written track, the signal amplitude from the track will be reduced and the core may also pick up the remains of previously written data. This would degrade the drive's signal-to-noise ratio and compromise its recovery capability.

The track width specified for the HP 9145A drive is, as far as we know, the narrowest used in the industry on ¼-inch cartridges, and is about half that in the HP 9144A drive. To achieve this we need to hold the tape head core width and core alignment tolerances tighter than in any comparable head. This was achieved by working very closely with the tape head vendor to refine their existing HP 9144A head manufacturing process until it was capable of producing HP 9145A heads with consistently good yields. In a year the vendor went from doubting that adequate yields could ever be achieved to producing heads that fully met the specification with good yields.

Head Positioning Accuracy. Head positioning to select between tracks in the HP 9144A drive is achieved by a stepper motor driving a lead screw. Riding up and down the lead screw is the head carrier assembly with the tape head mounted at one extreme.

This approach was maintained for the HP 9145A. However, the resolution of the stepper had to be at least doubled to place the head accurately over tracks that were half as far apart. Stepper motors with small angular stepping increments are now fairly common. However, the real challenges

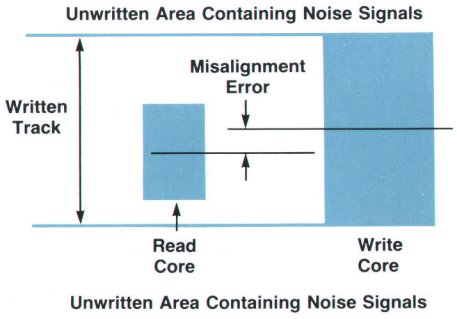


Fig. 7. *Misalignment error.*

here proved to be obtaining a leadscrew that maintains a constant thread pitch along its length and improving the head carrier movement to minimize the variation in linear displacement for each step of the stepper motor. This is essential if the track-to-track spacing is to be constant across the width of the tape.

After much evaluation of various leadscrews and modifications to the head carrier system, a head positioning system was developed that exhibits minimal errors that are highly predictable and repeatable across all mechanisms. Fig. 8 shows a typical final positioning accuracy plot.

Cartridge Tracking Specifications. The greatest single improvement of the HP 92245L/S cartridge over the HP 9144A cartridge is the replacement of a simple pin at the front of the cartridge with a guide roller. This part supports the tape on one side of the read/write head. This has allowed the cartridge to be respecified by the manufacturer so that the maximum vertical tape movement (tracking) is cut in half.

Clearly, vertical tape movement results in the tape head being slightly off the data track to be read. The improved cartridge specification is essential to be able to put 32 tracks on the tape and repeatably recover the data. Testing both at the cartridge manufacturer's laboratories and at HP showed that the new cartridge performs well within the new specification, as shown in Fig. 9.

Cartridge Media Defect Specifications. The HP 9145A drive is far more prone to data errors arising from media defects because of its narrow track size. Any drive can recover a read data signal until it goes below a certain threshold voltage that is set as a fraction of the peak read signal amplitude (typically 25 to 50%). The read signal level is proportional to the effective read track width. This effective track width is reduced by the presence of any defect. The drive is sensitive to media defects that occupy such a large proportion of the track width that the read signal falls below the threshold voltage (Fig. 10). This makes the HP 9145A drive, with its smaller track width, susceptible to smaller defects. In addition, the number of defects on a given piece of media dramatically increases as the defect size decreases. Fig. 11 shows the defect characteristics for the HP 92245L/S media.

The HP 9144A and HP 9145A drives have two main weapons with which to tackle these defects. First, all the HP-labeled cartridges that are supplied to customers are certified by the cartridge manufacturer. Certification takes the form of writing to the tape and then reading the signal back. Any errors are assumed to be because of media defects; these positions on the tape are marked and "spared out" so that they will not be used again. If any cartridge has an unusually high number of blocks spared it is rejected. In the case of the HP 92245L/S cartridges supplied to HP, this certification is performed by the cartridge manufacturer using unmodified HP 9145A drives. This immediately removes any concern about the unknown relationship between the certifying drive and a customer drive in reading data from the certified cartridge. Second, when writing data to a cartridge, both the HP 9144A and the HP 9145A drives immediately read back the written signal to verify its integrity using their read-after-write capability.[2] Any errors cause the drive to mark that area of tape as bad and then rewrite the affected data farther down the tape.

The HP 9145A drive's narrow track width makes it more prone to small-scale media errors than the HP 9144A. This is overcome by the higher-quality media in the HP 92245L/S cartridge, which has a lower proportion of defects at the HP 9145A read core dimension. To cope with the slightly inferior media in the HP 9144A cartridges, all HP 9144A-written data is read back with the write core of the HP 9145A head. This core is larger than the HP 9145A read core and so is less affected by the smaller defects. This write core approaches the size of the HP 9144A drive read core and so, when coupled with the HP 9145A's track seeking capability (discussed later), results in the HP 9145A's being able to recover a signal from an HP 9144A cartridge at least as well as an HP 9144A drive can.

Track Layout. The HP 9144A drive lays down data on tape in a serpentine fashion, that is, one track is written in one direction from one end of tape to the other, then the next track is written directly above in the opposite direction, and so on up the tape (see Fig. 12a).

A problem with this format is that the tape has a natural

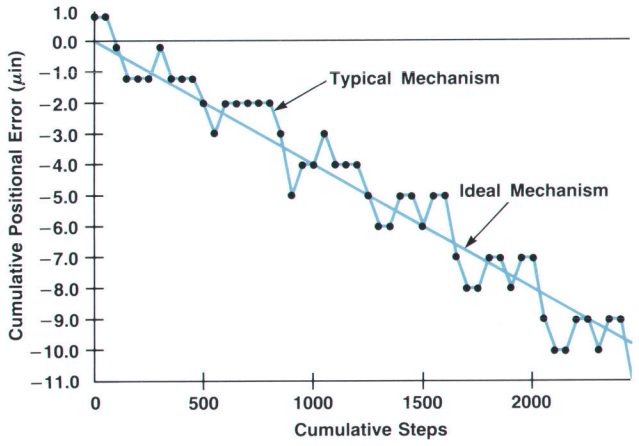


Fig. 8. *Typical head positioning error plot for an HP 9145A Tape Drive.*

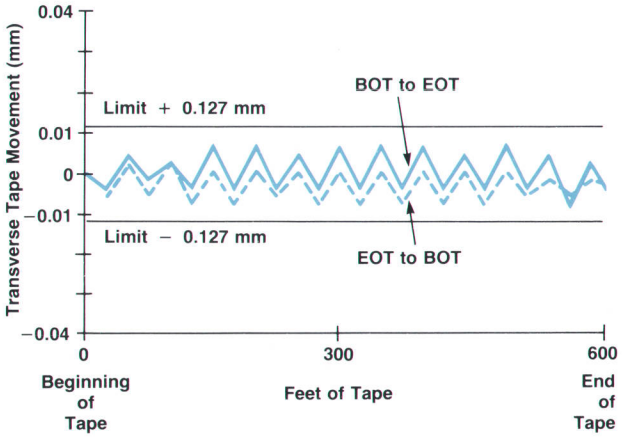


Fig. 9. *Transverse tape movement plot for an HP 92245L/S cartridge shows that vertical tape movement (tracking error) is well within specifications.*

tendency to step up or down as the direction is reversed because of the reversal of direction of tape pull. This leads to an error in the relative positions of two adjacent tracks that is at its maximum since these tracks are in opposite directions.

In the HP 9145A drive this problem was alleviated by writing all the tracks in the forward direction in the lower half of the tape, and all the tracks in the reverse direction in the upper half of the tape (Fig. 12b). Thus, adjacent tracks are generally written in the same direction and so do not suffer from this step error.

There is still a problem in the center of the tape where tracks 30 and 1 run alongside each other in opposite directions. This is overcome by allowing a slightly increased track spacing between these two tracks. This increased spacing does not impact the track density, since the allowance only has to be incurred once rather than 32 times.

Track Seeking. Both the HP 9144A and the HP 9145A drives locate the edge of the tape when each new cartridge is loaded. This edge is found by moving the tape head down until the read signal disappears. From this edge-of-tape position, the drive firmware is programmed with the number of stepper motor steps needed to reach each track.

This dead reckoning approach for locating any track from the located edge of tape has been perfectly adequate for the HP 9144A drive. In addition, it offers sufficient accuracy in positioning the head for the HP 9145A during a write operation. However, during reads, the HP 9145A may be attempting to read data that has been written by another drive. It is possible for the writing drive to have written the data to one extreme of the allowed tolerances, and then the reading HP 9145A to have positioned its read head to the opposite extreme. With the narrower data tracks used by the HP 9145A, this can lead to such poor alignment of the head over the written track that read data errors occur.

The HP 9145A overcomes this track misregistration by a technique known as track seeking. Initially, the track is located in the normal way as described above. If read errors occur, the drive attempts to find the track by stepping the head alternately above the nominal position and then below the nominal. The step size is progressively increased until the errors disappear. This new position is then considered to be the correct position for subsequent reads of the cartridge.

Core Alignment. To ensure that both the read and write cores of the appropriate channel in the tape head are always centered on the data track, the horizontal alignment between the write core and the read core must be held very tightly (Fig. 7). Although the write core is made slightly larger than the read core to minimize this problem, a limit is imposed by the need to fit 32 tracks across the tape. A tolerance analysis of the existing HP 9144A head manufacturing process showed that the write-to-read core alignment was already at the extremes of the process capability. For

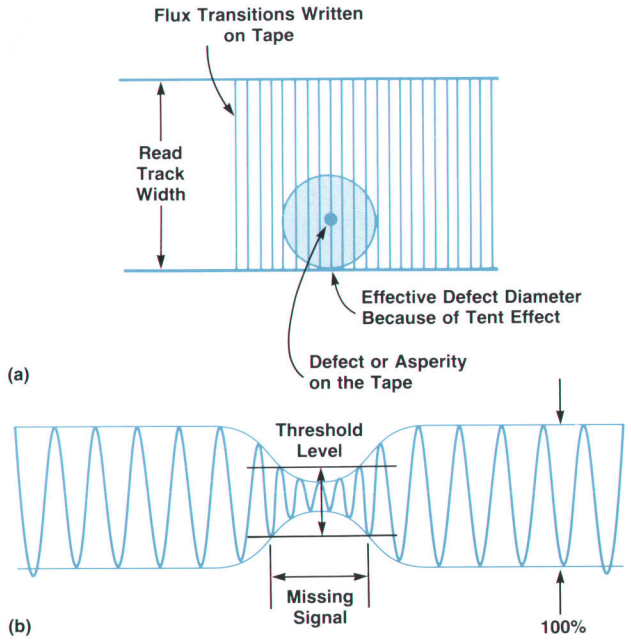
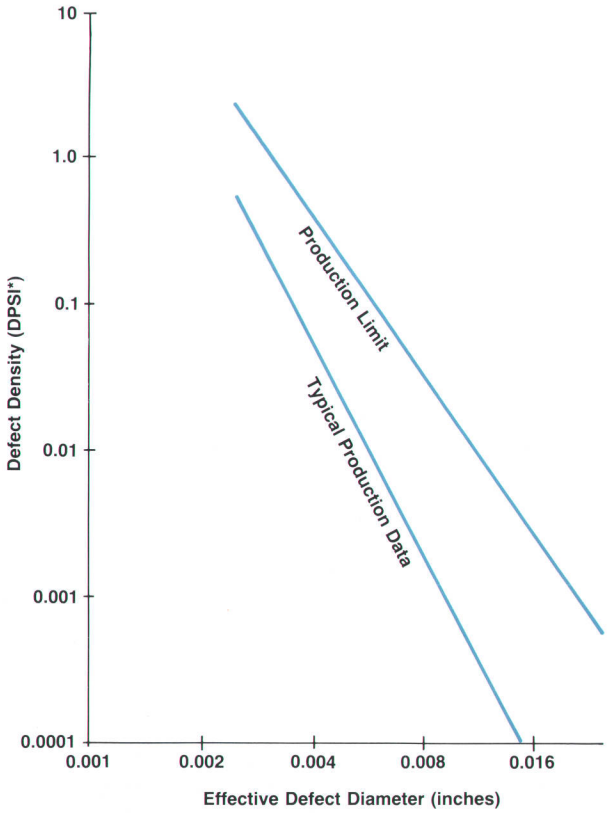


Fig. 10. *Effect of a media defect on the read signal. (a) The tent effect is where a defect on the media (effectively a lump) causes the media to be lifted away from the tape head surface. The resulting shape that the media takes (looking at a cross-section) as it is lifted in the middle and pulled back to the tape head surface looks like a wigwam, hence the name tent. (b) An analog display of the output of the read head with a defect on the media. A defect results in a reduction of the signal below the threshold level, causing lost read data.*



*DPSI = Dropouts Per Square Inch

Fig. 11. *Defect density characteristics for the HP 92245L/S cartridge. Defect density is the number of defects per square inch of readback area. The readback area is calculated from the read track width, the tape length, and the number of tracks.*

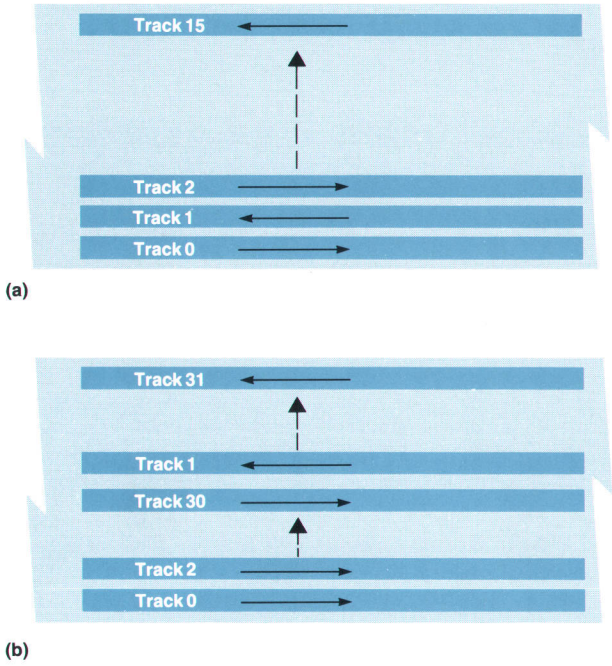


Fig. 12. *Track layouts. (a) HP 9144A. (b) HP 9145A.*

the HP 9145A, the tolerances had to be halved, necessitating a radical approach to the manufacturing process.

Several lengthy meetings with the head vendor resulted in an agreement to adopt a new manufacturing process that an exhaustive tolerance analysis showed should achieve the yields required. Subsequent manufacturing runs indicate that the original analyses were very accurate.

Tape Head Mounting. As previously mentioned, the tape head uses a read core that immediately follows a write core to provide read-after-write capability. To minimize pickup of the write signal through direct flux linkage into the read core it is desirable to maximize the separation between the read and write cores. This separation will cause the read core to be offset from the written track if the head is not mounted perfectly perpendicular to the tape motion (zero azimuth angle, see Fig. 13). This problem is twice as acute in the HP 9145A mechanism because its written track is half as wide as that of the HP 9144A drive.

In the manufacturing process for the HP 9144A the azimuth angle of the tape head is set by running a tape past the head and measuring the relative times at which the four cores see patterns prerecorded on the tape. While this process provides a good method for measuring the required accuracy, it takes a fairly long time to perform for each head and the adjustment of the head position to achieve zero azimuth is extremely difficult.

A design change to the head greatly simplifies this requirement while also speeding the head mounting process. The central section of the head was increased in size so that it protrudes above and below the outer sections. Since the interfaces between these sections define the core gaps and hence the effective positions of the read and write cores, a tool was designed that references off the exposed sides of the central head section to set the azimuth angle of the head accurately. This design change to the head also

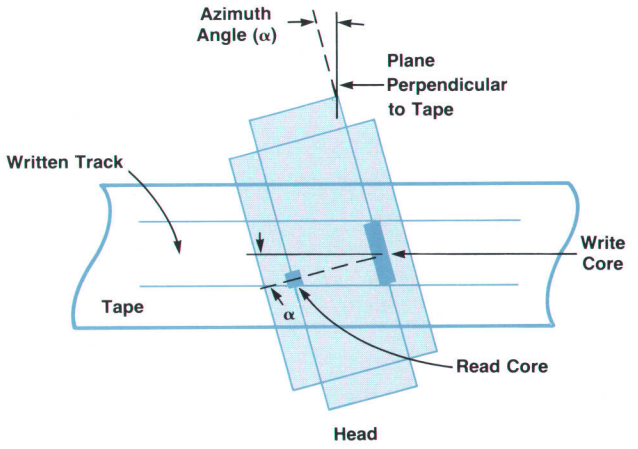


Fig. 13. *Azimuth angle. For clarity, only one of the two pairs of cores is shown.*

allows a simple mechanical means of verifying the accuracy of the azimuth angle of the mounted head.

Conclusion

The HP 9145A drive is now shipping to customers. Strict quality control measures are being used throughout the manufacturing process and further stressed-environment audit testing is being applied to the drives produced. So far these tests have verified the ability of the HP 9145A drive to achieve its original performance and reliability goals.

Acknowledgments

I would like to take this opportunity to recognize the considerable design achievements of the HP 9145A mechanism design team, specifically Jack McGinley, Simon Gittens, Henry Higgins, Alex Clark, and Steve Langford. In addition, I would like to thank the transition team of Geoff Mansbridge and Chris Martin, who devised the HP 9145A manufacturing process, which is key to being able to manufacture these mechanisms repeatably to the tight tolerances required.

References

1. W.L. Auyer, et al, "Controlling the Head/Tape Interface," *Hewlett-Packard Journal*, Vol. 36, no. 3, March 1985, pp. 44-47.
2. K.D. Gennetten, "Cartridge Tape Data Integrity Ensured at Five Levels," *Hewlett-Packard Journal*, Vol. 36, no. 3, March 1985, pp. 39-43.

Reliability Assessment of a Quarter-Inch Cartridge Tape Drive

Aggressive quality standards were verified by over 97,000 test hours before manufacturing release and are audited continually in production.

by David Gills

THE QUALITY GOALS FOR THE HP 9145A Tape Drive included a failure rate that was half that of the earlier HP 9144A, an error rate performance that was 10 times better than the HP 9144A's, the same useful life as the HP 9144A, and full backwards compatibility with all HP ¼-inch data cartridges.

The reliability test plan showed that to be able to halve the failure rate value within the development time of just over 1.5 years, then approximately 100 prototype units would be needed, resulting in an accumulation of 97,000 test hours before manufacturing release. Reliability growth was monitored using the Duane plot technique,[1] and there were interim goals at each of several checkpoints within the development program.

The reliability of this product is also being continuously assessed during manufacturing. For this purpose a detailed manufacturing reliability audit test schedule was developed. This will be discussed in more detail later in this article.

Tape Head

As described in the article on page 67, the tape head had to be totally redesigned because of the reduction in the track width and the increase in the tape speed. The effect of the tape head on the track placement accuracy is governed mostly by the mechanical tolerances of the core sizes and the positioning of the cores on the head. Shock, vibration, and temperature can lead to inaccuracies in the track placement. A full test program was carried out to explore and quantify all these effects on the performance of the drive.

The temperature margin above the storage specifications of the HP 9144A tape head before damage is incurred is well-understood from past test data. The elements of the manufacturing process that affect this margin are also well-understood.

The first mode of failure of the HP 9144A tape head when the temperature is increased outside the nonoperating temperature limits is a deformation of the profile of the head. This is caused by stress relieving of the plates that make up the structure of the head. It is a permanent failure, making the head unsuitable for further service. The manufacturing process has been radically changed to eliminate this mode of failure, which is now well-understood. By eliminating this mode of failure in the design of the new head, a much wider reliability margin has been achieved, making the head less sensitive to manufacturing variations. This was clearly demonstrated during the testing. No permanent damage was seen on any of the data heads following extensive strife (stress + life) testing.

Head Wear

Wear of the head is accommodated until the depth of the wear reaches the throat depth of the core. When this occurs, the head performance drops off dramatically and without warning.

Since the speed of the tape over the head has doubled from the existing HP 9144A, head wear characteristics have become an issue. As the tape speed increases, the frictional forces on the interface increase. However, aerodynamic compression of the air behind the tape at these higher speeds can have the opposite effect on the wear rate. This phenomenon is very difficult to describe theoretically, so the relationship had to be confirmed by a controlled experiment.

Testing showed that the rate of wear of the head took on the standard exponential shape when plotted as a function of time, as shown in Fig. 1. The measurements were taken using a Rank Taylor Hobson Talysurf 10 machine. A typical profile is shown in Fig. 2.

The throat depth of the core is nominally 40 μm, so it can be seen from Fig. 1 that there is considerable margin, even based on the small sample of drives tested. Therefore, head wear is unlikely to be the first mode of wearout failure. This was confirmed during testing. The capstan motor was found to be the first mode of wearout failure in the product.

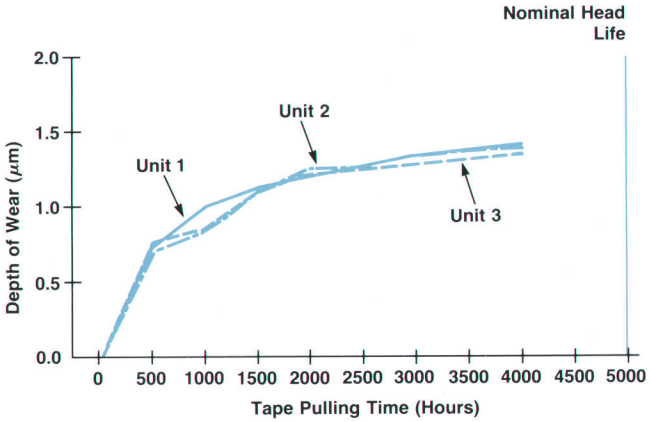


Fig. 1. *Head wear as a function of test hours for production HP 9145A heads.*

This will be discussed later in more detail.

Positioning Tolerance

Because of the increase in the requirement for track placement accuracy, the tolerances on the design and manufacture of the leadscrew that drives the head up and down across the tape had to be tightened significantly.

The leadscrew design for the HP 9145A is the same as for the HP 9144A. However, because of the precision required, the manufacturing tolerances were tightened significantly. On the HP 9144A, the thread pitch dimension has a ±5-μm tolerance, which is accumulated along the length of the thread. This will obviously give a large overall tolerance on the length of the leadscrew. On the HP 9145A, the thread pitch dimension also has a ±5-μm tolerance, except that it is not accumulated along the length of the leadscrew. This gives an overall tolerance for the length of the leadscrew of ±5 μm.

The manufacturing processes that influence this precision have to be controlled using statistical process control techniques to maintain the required accuracy. The data from the control charts is continually being monitored.

Repeatability of the track placement was critical to the success of the project, so a rigorous test program was adopted to assess its impact on the reliability of the drive. The leadscrew is machined and ground precisely from nonmagnetic stainless steel, but the nut that runs along it is made out of acetyl. The two main problems that arise are the accuracy of the leadscrew and the long-term accuracy of the nut. Since the nut is made of a much softer material than the leadscrew, we needed to ensure that there was no significant wear or load deformation. A key factor in the design of the head positioning system is that the head carrier is preloaded with a spring. This ensures that there is no mechanical hysteresis or backlash in the system, thereby driving the nut on one face of the thread only.

Shock and vibration testing was carried out to check for these issues, and it was found that this design has a very wide margin of safety over the quoted operating specifications before track placement becomes an issue.

Capstan Motor

Since the tape speed of the HP 9145A is twice that of the HP 9144A, that is, 120 ips compared with 60 ips, the

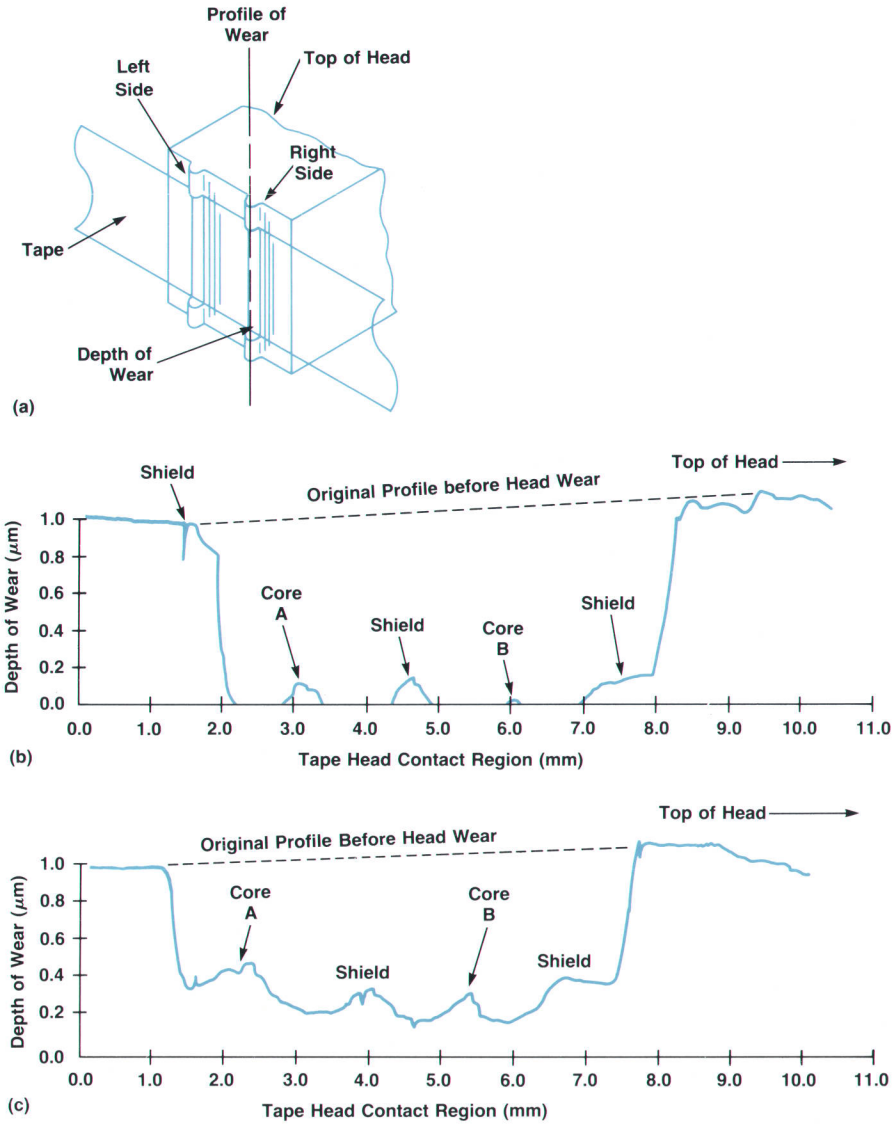


Fig. 2. *Profile of HP 9145A head wear after 2000 hours. (a) Area being profiled. (b) Left side of head. (c) Right side of head.*

capstan motor is required to run faster and have a higher torque. This allows the tape mechanism to start and stop more rapidly and accelerate to normal operating tape speed more quickly. These factors affect the overall life of the product, and since the goal for the useful life of the HP 9145A is the same as that of the HP 9144A, we needed to assess these parameters.

Working closely with our vendor, we were able to feed back test data from the development work being carried out in the laboratory from failures that were uncovered. The increase in speed and start-up torque showed that the current density at the brushes of the motor was too high, resulting in an unacceptable rate of brush wear. It was also found that the debris from the brushes was finding its way into the motor bearings, pointing to a need for shielded bearings.

The vendor was able to change the brush shape and material to extend the life to meet the goals. This was subsequently verified by extensive testing by the motor manufacturer and HP.

Printed Circuit Assemblies

Both the servo control printed circuit assembly and the main controller printed circuit assembly are newly designed boards, and as such had to be tested by a rigorous performance and stress test program. Again, working closely with our vendors, we were able to attack potential failure modes before the design was put into full production. An example of a typical component failure that was uncovered and eliminated is a crystal oscillator that failed during strife testing. Subsequent analysis showed that the failure had been caused by thermomechanical expansion of the terminals that support the crystal plate within the device. The movement of the terminals had resulted in a fracturing of the brittle crystal plate, rendering the component unserviceable. Since the vendor was unable to help in this instance, the component was second-sourced, resulting in much better quality.

Cartridge

The design of the HP 9145A relies very heavily on the quality of the media that it uses. With the performance of the drive increased so dramatically, the existing tape was inadequate. Although the older tape is compatible with the HP 9145A, its longer-term reliability was questionable. A reduction in defect size was critical to the reliability goals that were set for data integrity. The media defect size becomes far more critical as the width of the track decreases.

The project was discussed with the vendor that supplies the media, and jointly we agreed that a new tape needed to be introduced. This was a very extensive development program, carried out by the media manufacturer in parallel with the development of the drive at HP. Some of the problems associated with this development work have already been outlined in the article on page 67.

The cartridge mechanics were also redesigned to give better tape handling characteristics, resulting in better tape tension and drive force control.

The hubs that support the tape were redesigned from a new material, so that the cartridge is able to cope with the additional tape speed without wearing the hubs at an unacceptable rate.

The drive belt that runs on the tape (wound around the hubs), was also redesigned from a new material, and is being manufactured with a new process, giving more stable belt tension.

An additional tape guide was designed in to provide better tape placement accuracy. Since the tracks on the HP 9145A are half the width of the tracks on the HP 9144A, this was a critical area in the design of the cartridge.

The guide rollers were also redesigned, since the increase in tape speed caused the rollers to become a source of unacceptable acoustic noise. Resonances were built up from out-of-balance forces of the rollers running at high speed, and were transmitted through the case and baseplate of the cartridge.

Backwards Compatibility

Since the HP 9145A is intended as a natural upgrade path from the HP 9144A, it must be able to read existing tapes that have been written by the HP 9144A and other HP ¼-inch tape drives. Complexity is added by the variety of revisions of each product and of the ¼-inch cartridge. The HP 9145A has to be compatible with the entire ¼-inch tape product family over the operating temperature range, for any data pattern written by any other compatible drive. To prove the error rate performance over all possible combinations, the testing required would take over five years!

Some of the variables to be considered when concerned with interchange and backwards compatibility are temperature, humidity, data pattern, length of tape, data source, tape age, tape type, revision of unit, altitude, shock and vibration, 120V/240V, and age of drive. Using Graeco-Latin square (statistical design of experiments) techniques,[2] we were able to get this test program down from 260 combinations to a program of 16 representative combinations.

On each of the 16 runs, 10^{11} bits of data was handled by each drive (the error rate specification is 1 bit in error for 10^{11} bits of data handled). The tests were designed to find combinations that did not work at all or any trend or pattern that could indicate a combination that would potentially not meet specification.

The program was very extensive, and the testing was not able to find a combination that indicated that the specifications could not be met. The HP 9145A showed that it was able to read data from any combination that it was tested

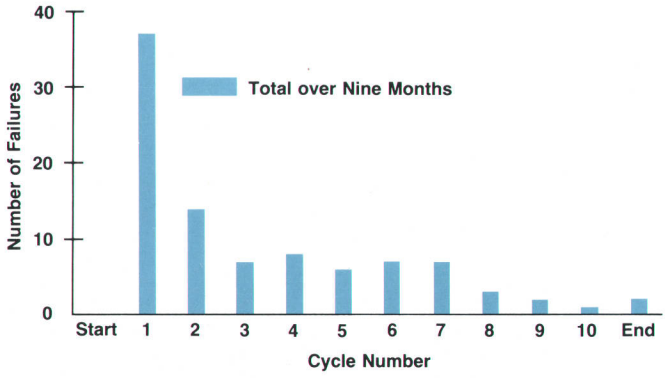


Fig. 3. *Typical distribution of faults in burn-in testing.*

against.

Test Program

At the completion of the test program, 97 prototype units had been tested, accumulating over 97,000 test hours. Over 500 tapes were used during the performance and interchange testing, and a total of over 150 failures were found. These failures can be broken down as follows:

Mechanism related failures: 70%
Controller related failures: 20%
Firmware related failures: 10%

Audit Testing

Although the tests described so far confirmed the reliability of a sample of prototype and early production drives, they in no way reflect the factors that will affect the reliability of the product in the long term, that is, factors related to the manufacturing process. Although the HP 9145A has been highly leveraged from the HP 9144A, with many improvements to the product and the process based on the wealth of information available, the reliability of the product in the long term cannot be measured or quantified unless real data is at hand. The warranty system, which records the numbers of failures in the field, provides data that is obviously too late. What is needed is a means of controlling the reliability of the drives with a closed-loop system that has a response that is almost immediate. The only way to do this is by continued unit testing on an audit basis.

An audit test strategy was devised for the HP 9145A that will enable manufacturing engineering to keep a tight control on process variations that affect the reliability of the product. A secondary objective of the audit testing is to ensure that the data being collected correlates closely with the data that is being continually collected from the warranty claims.

Much data is available from the prototype testing, and this was used as a basis for determining the general content and duration of each audit test. Also, a comparison was made with other divisions of Hewlett-Packard to appreciate some of the problems encountered in such testing.

The audit testing has three phases: burn-in, customer environment, and life tests.

Burn-In. This is performed on 100% of all units manufactured, and lasts for approximately 14 hours. This test is solely designed to catch the dead-on-arrival or infant mortality failures that somehow escape the manufacturing final test. Although the final test is considered to be adequately thorough in testing the total functionality of the unit, there are often intermittent faults or failures caused by weak materials that survive the final test. These intermittent faults often appear very early in the product's service life.

The format of the burn-in test is based on data from the prototype testing. It consists of ten cycles of power cycling and self-tests, loading tapes, performing read/write and read-only error rate tests, performing locate and read and/or write operations, comparing data with the host system, and unloading and unlocking the cartridge.

The results from the initial nine months of testing show that the faults are being uncovered very early in the test cycles (see Fig. 3). This obviously means that there is a real opportunity for shortening this test after more confidence is built up from a bigger test history.

Customer Environment. These tests are designed to simulate more closely the environment seen by the product during the warranty period. Hence, the failures found are intended to mirror the failures found in the warranty system. The duration of this test has been established from an assumed typical use of the product. The test is not performed on all units, but on a sample of approximately 5% of production. The testing simulates the time from when the unit leaves the end of the production line to the end of the warranty period. Therefore, the shipping of the product, the end-use handling, and the in-service operation of the product are simulated. The details of the customer environment are as follows:

- Book out unit from finished goods.
- Drop unit in packaging onto concrete from a height of 1.2 m.
- Make visual and functional inspection for cosmetic damage, accessories completeness, failure to power-up and pass self-test.
- Thermal cycle between the operating limits of the drive for 40 hours.
- Apply nonoperating vibration at 1.5 times specification for one hour.
- Compare data at ambient temperature between host and drive for 100 hours.
- Interchange data between drives in product family for 24 hours.

The test cycle lasts for one week, after which the units are returned to the production line for shipment. These units are considered to be some of the most reliable to leave the factory, since they have had all the infant mortality problems removed, and have proved to work reliably for a significant period of time.

Life. The life testing of the HP 9145A is currently being carried out by the suppliers of the media. This is because the media suppliers own many HP 9145As and use them at a very high duty cycle to certify all the tapes that are manufactured by them. This enables HP to use this information without cost. Obviously we need to be working very closely with these suppliers to ensure that the information that they supply to us is accurate and complete. The collection of this data will continue into the foreseeable

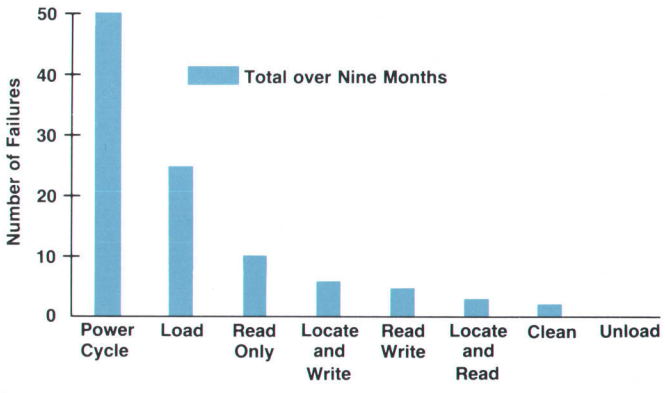


Fig. 4. *Distribution of faults among the various subtests of the burn-in test.*

future since the data is obtained free and will be needed to ensure that any changes in the manufacturing process do not affect the long-term reliability of the drives.

Results of Audit Testing

The first nine months of data has shown some interesting results. Problems are being uncovered in the burn-in test, as expected. This justifies its presence and quantifies the costs saved from warranty claims arising from very early failures, not to mention the hidden costs of customer dissatisfaction.

It can be seen from Fig. 3 that the majority of failures that have been uncovered so far have occurred very early in the test program. After the fourth cycle, which is 1.3 hours after the start of the test, most of the problems seem to have been found. This data indicates that a reduction in the test time will find the same level of faults, but will save substantial cost in manufacturing overhead (the cost of the tapes is probably the biggest factor here).

Fig. 4 shows the tests that are most effective in producing the faults. This data agrees with the test data from the earlier prototype testing, and is useful feedback for reliability planning on future projects.

The customer environment testing is currently showing a very similar trend in the results. One year after introduction over 8,000 units have been shipped to customers. The warranty data shows that the actual failure rate of the HP 9145A is better than the failure rate goal at introduction. As a result of the audit testing strategy, the warranty failure rate continues to fall to a point where today the warranty failure rate is half of that measured a year ago.

Acknowledgments

The successful development of the 9145A was a result of substantial contributions from many people in different functional areas. Recognition should go to the R&D mechanism design team, led by Jack McGinley. Simon Gittens, Andy Topham, Henry Higgins, and Alex Clark designed the new head, servo control with associated read/write electronics, stepper motor, and front panel. Thanks go to the controller design team, led by Ben Wilkinson, and the firmware design team, led by Paul Robinson. In particular, we would like to acknowledge the efforts of Tracey Hains and Kevin Jones, whose work in failure diagnosis helped to make this project run so smoothly. Particular thanks in the materials department go to Ian Russell for his work on the development and qualification of the new tape cartridge, and to Steve Daniels and Robin Longdin, whose hard work and dedication in running the test program, ensured that the project was successful. There are many more people whose contributions were invaluable to this project, but there are just too many to mention individually.

References

1. P.D. O'Connor, *Practical Reliability Engineering, 2nd Edition*, John Wiley & Sons, Inc., 1985, p. 285.
2. G.P. Box, et. al., *Statistics for Experiments*, John Wiley & Sons, Inc., 1978, pp. 253-261.

Use of Structured Methods for Real-Time Peripheral Firmware

HP's Computer Peripherals Bristol Division made some significant changes in their firmware development process to ensure that they met a demanding development schedule and still produced a quality product.

by Paul F. Bartlett, Paul F. Robinson, Tracey A. Hains, and Mark J. Simms

PRODUCTIVITY AND CONCERNS about quality may seem to be opposing concepts when product development time is short. However, with planning, the proper tools and a good development method, productivity and quality objectives can be achieved and still meet the time-to-market goals. In the development of the HP 9145A Cartridge Tape Drive at HP Computer Peripherals Bristol Division (CPB) the firmware was always on the critical path during the entire product development time. We had to produce reliable prototypes of the HP 9145A for testing with the target machines one year after the project start date. We realized at the beginning of the project that if we used the firmware development process we had at the time, we could not meet the schedule and still produce a quality product. Some of the problems we had in our development process at the time included:

- Total reliance on text for firmware specifications. There were very few graphical representations for the system architecture, data, and module organizations.
- Firmware testing was different for each project and the effectiveness of testing was not measured. Also, there was no overall test planning process.
- Except for the number of noncomment source statements (NCCS), no metrics were collected.
- Tool support consisted of emulation, source code control, and editing on HP 64000 Logic Development Systems. There were some tools for text documentation and structured design which existed on a variety of systems.

Improvements were made to our development process in the areas of planning, methods (analysis, design, and testing), and metrics (process measurement). The most significant changes involved the use of structured analysis, structured design, and structured testing. Structured design had been used on past projects for module design and the technique had worked well.

Each engineer on the project was equipped with an HP 9000 Series 300 workstation which was used for program development and emulation. A network of workstations was created with one workstation dedicated as a central data base for configuration management (i.e., keeping track of all versions of our documentation and code). To enable us to use the structured analysis and structured design (SA/SD) methods effectively, HP Teamwork/SA was installed on each workstation. This product allowed us to produce all of the real-time structured analysis and structured design documentation for the HP 9145A firmware, and assisted in ensuring analysis and design consistency between the members of the team. Other software tools that we used included a code-efficient cross compiler from C to 68000 assembly language and a 68000 emulator.

This paper describes our experiences with applying SA/SD techniques and tools to the development of the HP 9145A firmware.

Real-time Structured Analysis

Structured analysis is a method that enables designers to partition a system into manageable component processes. It helps to identify the system requirements and functionality so that consideration about implementation details, such as system architecture and module design, is delayed until necessary. This allows the designer to keep as many design options open as possible. Structured analysis[1] has been successfully applied to business and commercial systems where the emphasis is primarily on data flows and processes. In real-time systems, in addition to data flows and processes, control and timing are also major considerations. For the HP 9145A firmware development we used some parts of the structured analysis real-

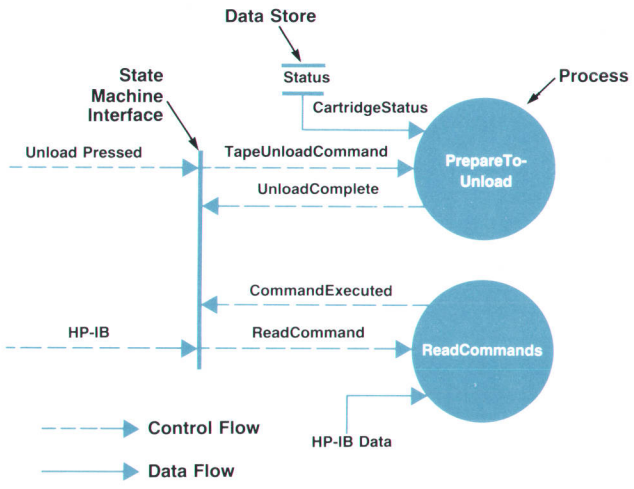


Fig. 1. *Modelling control flows and data flows in real-time structured analysis. The function of a process is to perform the operation implied by its name. The vertical bar represents the interface to a state machine.*

time extensions described in references 2 and 3.

Real-time systems have two features that nonreal-time structured analysis cannot model. One is the ability to distinguish between the flow of control signals such as interrupts, and simple data flow such as flags or values. In real-time structured analysis, information flow between processes is represented by control flows for control signals and events and data flows for plain data (see Fig. 1). The control flows shown in Fig. 1 send a signal to activate or deactivate a process. For example, when a user presses the **Unload** button on the front panel, the state machine sends the TapeUnloadCommand signal to activate the process PrepareToUnload. The data flows represent information a process must retrieve from elsewhere in the system (e.g., a data store or another process) to perform its operation. For example, in Fig. 1 the process PrepareToUnload retrieves data about the CartridgeStatus from the Status data store.

The other deficiency of ordinary structured analysis is in modeling sequences of real-time operations. These are situations where timing or the order of responding to events and actions is very important. Starting a servo motor and waiting until it is up to speed before proceeding, or enabling DMA transfer of data to tape, are examples where timing and sequence are critical. One method used in real-time structured analysis to model sequence control is the state transition diagram (STD). State transition diagrams are used to model state machine behavior and to show how different system states are influenced by control signals. Fig. 2 shows the state transition diagram for the model shown in Fig. 1. This state machine is designed to respond to events such as **Unload** button pressed, **Self Test** button pressed, cartridge inserted, and so on, and still read commands from the HP-IB.

Real-time structured analysis can be used to help partition the hardware and software functionality for a whole system. In our situation the division between the hardware and firmware functions had already been decided before we began using the method. Therefore, we concentrated on using the methods only on the firmware.

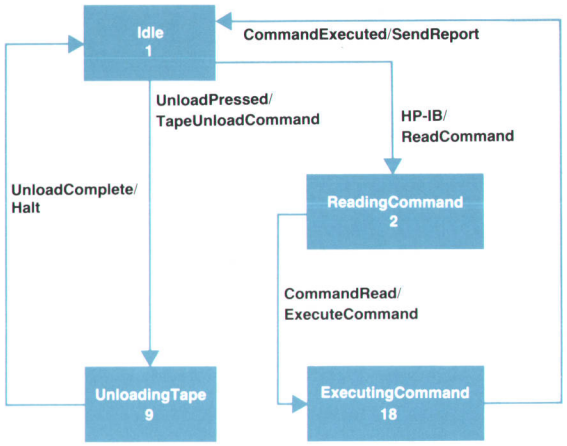


Fig. 2. *A portion of the state transition diagram for the model shown in Fig. 1. The number in each block is a state identifier. There are at least 18 states in this state machine, but for clarity, only four essential ones are shown.*

Context and Data Flow Diagrams

Our first task was to define a context diagram for the HP 9145A firmware. A context diagram enables the designer to identify all the external entities such as other systems, users, and peripherals, with which a system must communicate. The context diagram for the HP 9145A firmware is shown in Fig. 3. There are three components to a context diagram: terminators, data and control flows, and a single process. Terminators represent external entities that can be either sources or sinks depending on whether they transmit or receive data. The data and control flows represent the communication paths between the terminators and the single process. The single process defines the central role of the system being designed. In our case the firmware is used to control and monitor the HP 9145A tape drive.

From the context diagram we developed a top-level data flow diagram (DFD) which defines the main firmware tasks and the interfaces between them (see Fig. 4). The interfaces between the tasks consist of messages passed via an interprocess communication module in the operating system. The effort involved in developing the data flow diagram enabled us to understand how to divide the system into manageable pieces for development and further analysis. Our development plan was refined so that the analysis phase was divided into smaller stages in which functions within each task could be analyzed. This enabled us to plan reviews to occur whenever one of these stages was

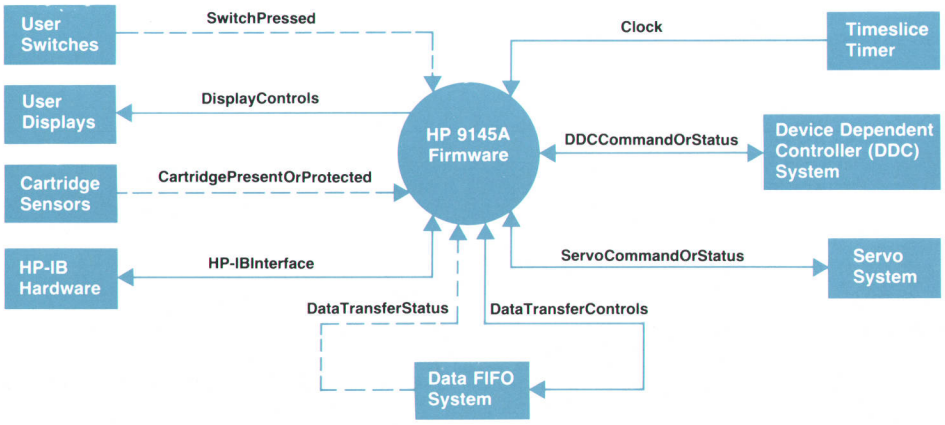


Fig. 3. *Context diagram for the HP 9145A firmware.*

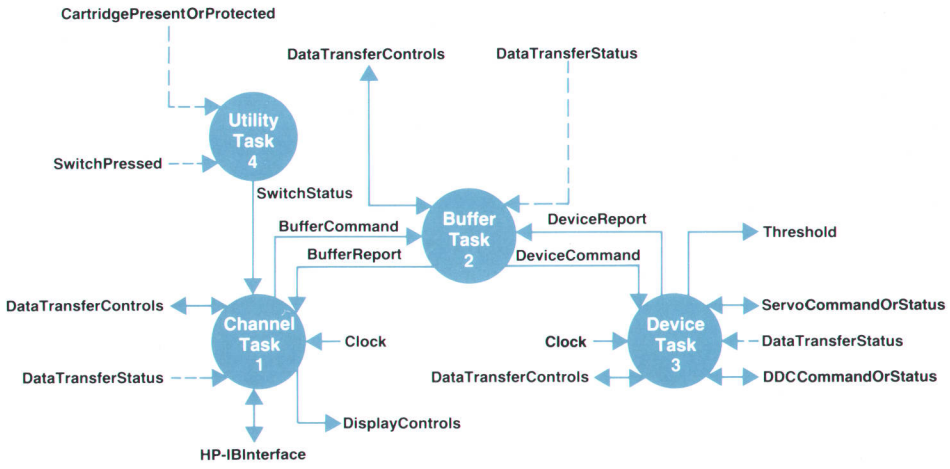


Fig. 4. *Data flow diagram for the HP 9145A firmware.*

completed. We could review a small amount of documentation every two or three weeks, instead of waiting for months to review a vast amount of information.

The tasks shown in Fig. 4 perform the following functions:

- Channel Task. The channel task controls the interface between the operator, the host computer system, and the HP 9145A firmware.
- Buffer Task. The buffer task controls the flow of data between the HP-IB interface, an internal data buffer, and the magnetic medium (tape).
- Device Task. The device task controls the read/write circuitry and the tape mechanism.
- Utility Task. The utility task contains functions used to perform switch and button debouncing and control the operation of the lights on the front panel of the drive.

The data and control flows in Fig. 4 represent the interprocess communication between the tasks. Interprocess communication in the HP 9145A firmware is implemented by a number of mailboxes used for holding messages or commands.

Each of the tasks has its own context diagram and its own set of external entities. Fig. 5 shows the context diagram for the device task. From these context diagrams detailed DFDs were generated for each task. Fig. 6 shows a portion of the DFD for the device task and Fig. 7 shows a portion of the DFD for the process Read/Write Operations which appears in Fig. 6. When the DFDs were leveled to primitive processes (processes that cannot be decomposed any further) process specifications were created like the one shown in Fig. 8 for the process ExecSingleShotRead which appears in Fig. 7. The number and complexity of the data and control flows increased as the design became more detailed. For example, the data flow diagram in Fig. 6 actually contains 24 control flows and 46 data flows between the processes. HP Teamwork/SA was used to create and maintain a central data dictionary data base for the whole firmware system. A data dictionary is a method for defining every data flow, control flow, and data store used in a system. The central data base allowed us to maintain data consistency between the various tasks. Because we were putting a great deal of effort into analysis, the data dictionaries became colossal. HP Teamwork/SA was really helpful here because it provides a checking facility that makes sure the data and control flows are consistent between levels of the system model. We ran the checking facility before each review so that the reviewers could concentrate on checking for correct functionality instead of spelling and consistency errors. Fig. 9 shows some of the data dictionary entries for the context diagram shown in Fig. 5.

A large proportion of the analysis of the firmware for the project involved the analysis of control. State transition diagrams served as a major part of this analysis. These diagrams allowed us to concentrate our control structures in a small number of places. To ensure manageability and readability most of our STDs consist of less than 20 states. An STD with more than 20 states becomes very confusing and hard to read. A portion of of an STD for the process ExecSingleShotRead from Fig. 7 is shown in Fig. 10.

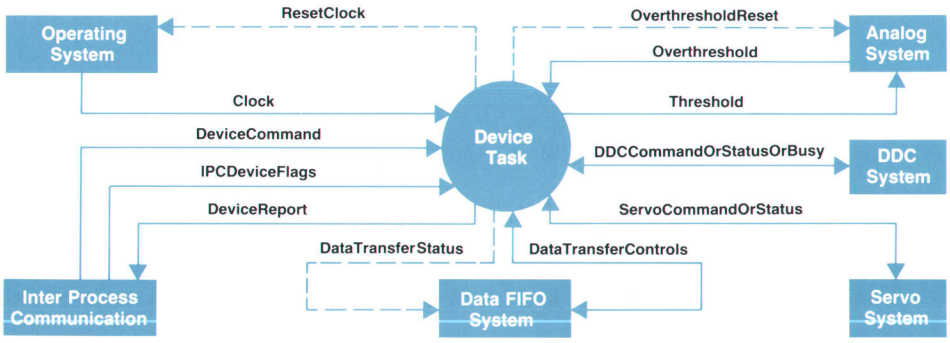


Fig. 5. *Device task context diagram.*

Lessons Learned

Five months had been allocated for the structured analysis portion of the firmware development and we managed to finish on time with two weeks to spare. Had we been more experienced with the methods and tools we would have finished sooner. Some of the observations and lessons we learned from using real-time structured analysis include:

- A lot of effort, maybe too much, was expended at the very top level of each task. One reason for this is that we had to do some informal lower level analysis to decide whether the top level solution produced was good enough. Having spent this effort early we found that when it came time to do formal lower level analysis the task was much easier.
- In some areas of the analysis we found that it was very difficult to produce a solution because of the amount of fan-in* to most processes associated with hardware dependent areas. We encountered some difficulty with using structured analysis for analyzing functionality associated with time-critical control of hardware. There are some techniques in the SA/SD real-time extensions[3] that can be used to analyze critical hardware/software timing situations. However, we did not get a chance to use these methods. In addition, we were trying to specify detailed algorithms using data flow diagrams, which is not the intention of the method.
- Too much effort was expended considering the implementation aspects of the system instead of defining the system functionality. This resulted in process specifications that tended to be trivial and not very useful.
- By the end of the structured analysis phase all of the engineers on the team thoroughly understood what their portion of the firmware was expected to do as well as what some of the rest of the firmware was doing.
- Because of the thorough analysis that had taken place a large number of anomalies were discovered and fixed in the original project specifications (external reference specifications).
- After structured analysis there were very few changes to the functionality of the product, except in areas where the characteristics of the mechanism or the tape were not fully understood.

*Fan-in is defined as a large number of processes all making calls to one common process.

Structured Design

In this phase of the development the data flow diagrams developed during the structured analysis phase were used to design the architecture and hierarchy of functions for the HP 9145A firmware. In most cases this process resulted in structure charts like the one shown in Fig. 11 for the process ExecSingleShotRead. In one task we found that there was no need to develop structure charts because the structured analysis produced such a flat structure, all based on

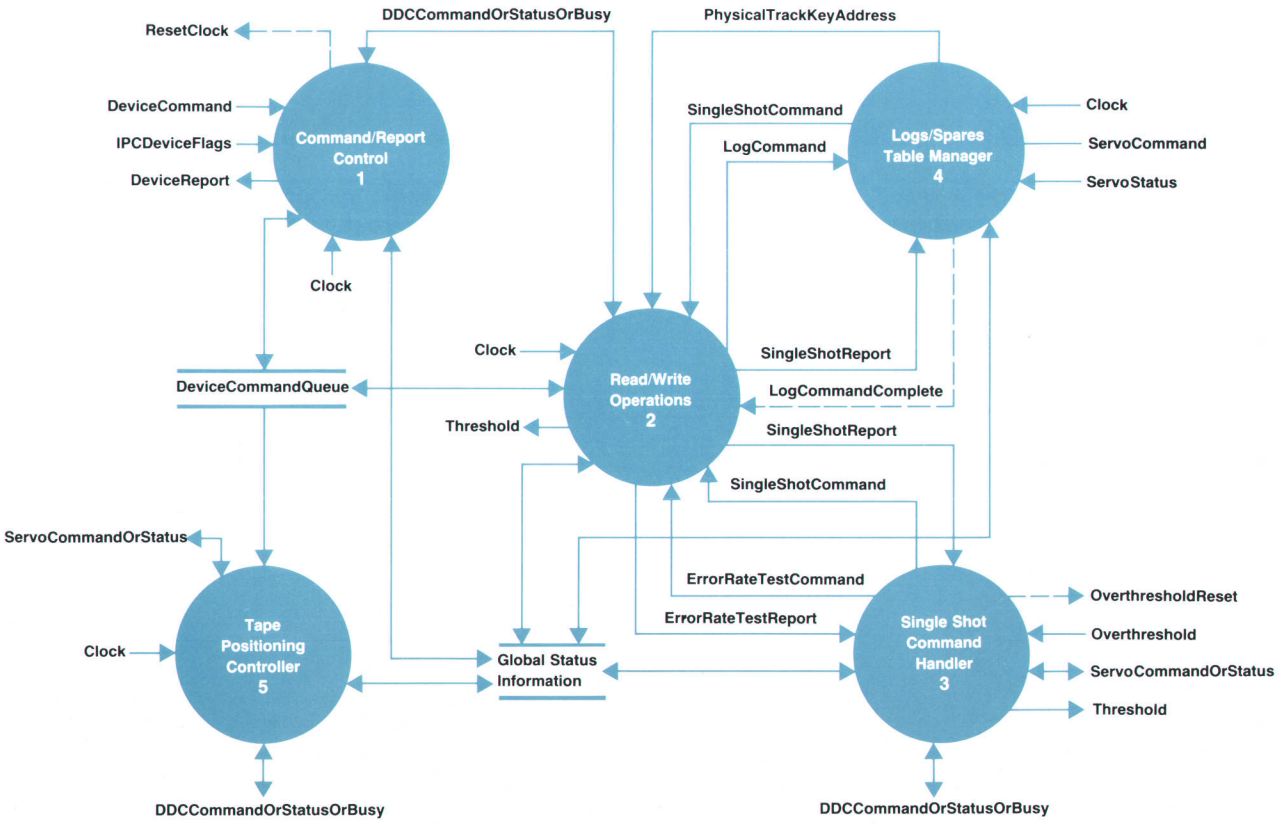


Fig. 6. *A portion of the detailed data flow diagram for the device task. There are actually 24 control flows and 46 data flows associated with this DFD. The number within each process bubble is used for identification and traceability.*

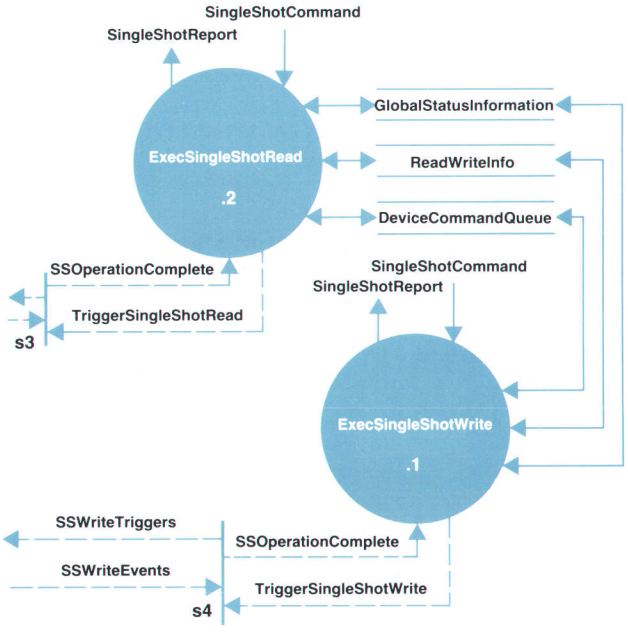


Fig. 7. *The data flow diagram for two of the processes associated with the process* Read/Write Operations *shown in Fig. 6.*

calls from a state machine, that we felt that the exercise would not yield any useful extra information.

The production of structure charts was limited only by the speed with which information could be put into HP Teamwork/SA. This was because there was so much detail from the structured analysis phase that the design came together with very little effort. Also, as in the structured analysis phase, the data dictionary proved to be invaluable and the HP Teamwork/SA checking facility helped to ensure that consistent designs were produced.

In parallel with producing structure charts, detailed module specifications were written for all the procedures. These module specifications were written so that they could be used as procedure headers for the code. Fig. 12 shows the module specification for the state machine SSReadInitialize shown in Fig. 10. In many instances part of the module specification was extracted directly from the process specifications written during the structured analysis phase. At this point of the project module specifications became the most important documentation. All changes that were made to the code were documented in the module specification for the affected function. Keeping the structured analysis documentation up to date was relatively tedious and time-consuming. However, towards the end of the testing phase this documentation was updated to match the final design of the firmware. This showed that even with an automated tool to enter a design, there must be a mechanism to update the design documentation automatically when changes are made during implementation.

Structured Testing

Structured testing encompasses the planning, design, documentation, and execution of tests. It is a method for managing the overall testing process and for providing traceability between the various types of test documenta-

```
Name: 2.2;2

Title: ExecSingleShotRead

Input/Output:

TriggerSingleShotRead  : control_out  *control flow out*
SSOperationComplete    : control_in   *control flow in*
DeviceCommandQueue     : data_inout   *bidirectional data flow*
SingleShotCommand      : data_in      *data flow in*
SingleShotReport       : data_out     *data flow out*
GlobalStatusInformation: data_inout
GoodBlocksRequired     : data_out
ReadWriteInfo          : data_inout

Body:

Translate parameters into the appropriate command queue settings
and other stored information.

Trigger the single shot read state machine.

Wait for completion.

Compile status return value derived from Comfail and abort
conditions.

Reset Comfail and any other abort conditions generated during the
test.

Get more status from ReadWriteInfo (MaintenanceTrackOverflow and
TapeNotWrittenTo).

Gather status.

Return result.
```

Fig. 8. *The process specification for the process* ExecSingleShotRead.

```
Clock (data flow, cel) =
   *A continuous data flow indicating the number of ticks of the
    clock. * .

ResetClock (control flow) =
   * Control from device tells the operating system *
   * to zero the millisecond timer.                  *

DDCCommandorStatusororBusy (data flow) =
   [ DDCCommand ¦ BusyStatusBit ¦ DDCReport ¦ ReportStatusBit]

DeviceCommand (data flow) =
   * The DeviceCommand splits into six groups of commands.     *
   * Each command in each group contains an opcode identifying *
   * the type of command, a subcode identifying the command    *
    * itself and a number (sometimes 0) of parameters.          *
    [ DiagnosticCommand ¦ TapeLoadCommand ¦ ReadBlockCommand ¦
      ResetDeviceCommand ¦ TapeUnloadCommand ¦ UtilityCommand ¦
      WriteBlockCommand ]

DeviceReport (data flow ) =
   *Report contains 10 16-bit words where the order of the words *
   *is significant. There is a seperate report for each class of *
   *command.                                                     *
    [ DiagnosticReport ¦ TapeLoadReport  ¦ ReadWriteReport ¦
      ResetReport  ¦ TapeUnloadReport  ¦ UtilityReport  ]

IPCDeviceFlags (data flow, del) =
   * Flag value - True or False.   *.

Threshhold (data flow, pel) =
   *Hardware *
   *8-bit numeric valus to adjust the effective gain of the read *
   *amplifier.                                                   *.

OverThresholdReset (control flow, del) =
   *Hardware *
   *Control to reset the overthreshold latch *

ServoCommand (data flow) =
    [NormalOperationCommand ¦ SpecialFunctionCommand ¦
     ServoTransparentCommand ¦ UtilityDiagnosticCommand]

Data Dictionary Notation

cel  : Continuous element. An attribute that indicates that the
item can take on a large number of values.

del  : Discrete element. An attribute that indicates that the item
can take on a finite number of values.

pel  : Primitive element. An item that cannot take on any other
values or be be further decomposed.

  =  : Is-Equivalent-To.

 [¦] : Either-Or selection.

  +  : And (sequence selection).

* * : Comments.
```

Fig. 9. *A portion of the data dictionary definitions for the data and control flows for the process* ExecSingleShotRead.

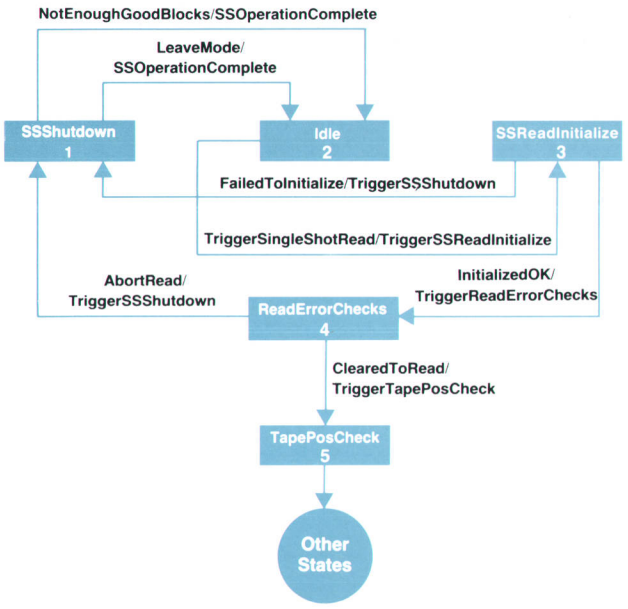


Fig. 10. *A portion of the state transition diagram for* ExecSingleShotRead.

tion. Structured testing tends to minimize the cost of product development by finding problems as early as possible before the cost of rework is high. An example of this might be when a test designer is writing a test and discovers that a specification is incomplete or ambiguous. If the specification defect is removed before coding takes place the cost of defect removal is low.

Our test strategy was based on traditional structural and functional test techniques[4] and well-coordinated test planning.[5] A hierarchy of test plans (Fig. 13) was produced for the whole product. Each sector of the system, mechanical, electronics, and firmware, had a similar hierarchy of test plans. These test plans were produced from the top down so that the overall firmware test plan was produced before the test plans of any individual tasks. Once the code had been written, the tests described in the test plans were executed starting from the bottom. By writing the test plans in parallel with designing the firmware we found a lot of problems that otherwise might have been overlooked.

To minimize the effort required during the testing phase an automatic test package was developed to run most of the tests. This test package accepted test scripts, exercised the product, checked for correct responses, and reported any anomalies to the test engineer. This enabled us to run tests during periods when there were no engineers available to monitor the tests.

All problems were recorded in a defect tracking system. This system was used to monitor the number of defects found, their severity, and the reason for each defect. It was also used to monitor the current status of each defect so that we could determine how many defects still had to be resolved. From this information we were able to monitor the progress of the project, and to tell whether the defect rate was under control.

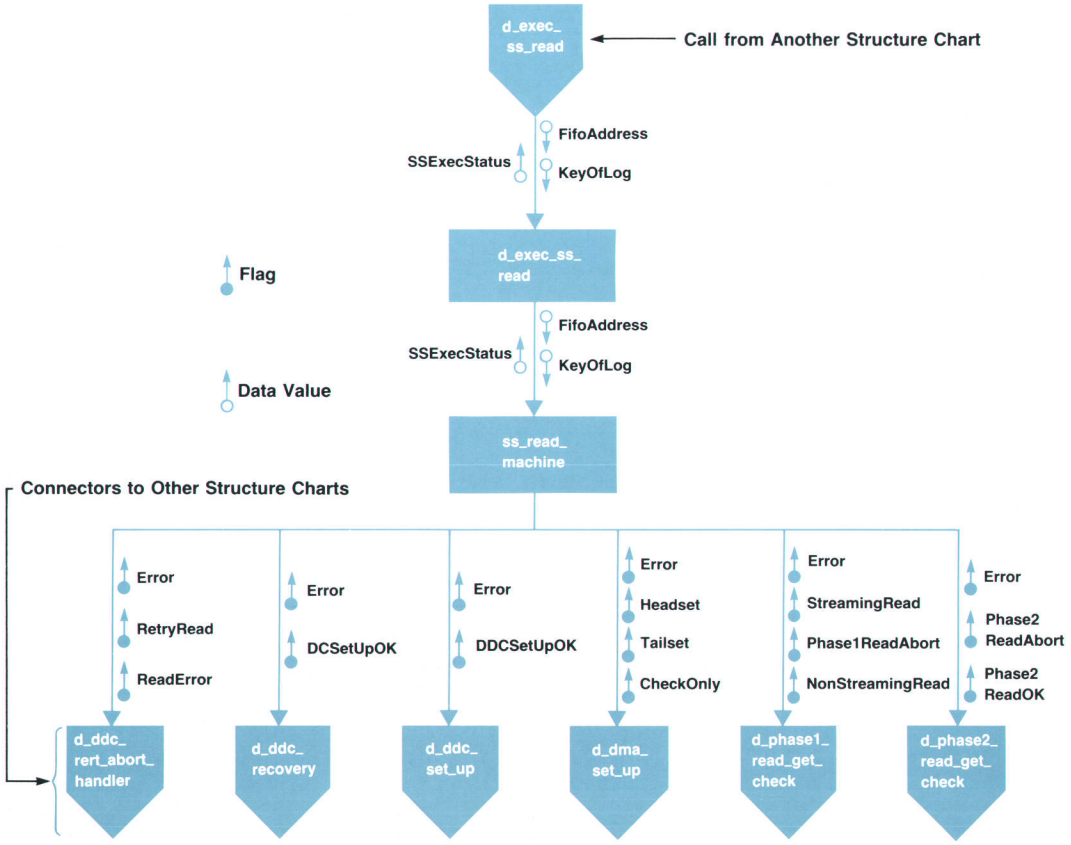


Fig. 11. *A portion of the structure chart for* ExecSingle ShotRead.

Results

As mentioned earlier, the firmware for this project was on the critical path from the beginning. As it turned out, our progress was so good that we were able to meet all major deadlines. Table I shows that our original estimates were very accurate and we achieved very respectable figures for our estimation quality factors (EQFs)* for each phase of the project.

Table I
Project Estimation Quality Factors (EQFs)

Activity	EQF	Estimated Elapsed Months	Actual Elapsed Months	Estimated Engineer Months	Actual Engineer Months
Analysis	9.0	4.5	5.0	20.0	22.5
Design	7.5	1.7	2.0	6.5	7.5
Coding	9.3	1.1	1.25	6.25	7.0
Functional Testing	11.5	4.2	4.6	21.0	23.0
System Integration	5.0	4.0	5.0	12.0	15.0
Total Project	8.1	15.5	17.85	65.75	75.0

During the analysis phase we defined over 420 processes on the data flow diagrams which resulted in about 570 procedures in the final product. The final code consisted of 24 KNCSS (thousand lines of noncomment source statements), and 123 kilobytes of object code.

Defects were tracked and a chart maintained to show the cumulative number of defects detected as a function of elapsed time. A code path monitor, which kept a count of the number of code statements tested, was run while the regression test package was being run. When the total test package was run, 85% of the statements were being exercised. The code path monitor enabled us to verify that 100% of the most critical areas of the code were being exercised. These figures do not take into account the extra code coverage that individual engineers achieved during their module testing activities. Our expectation at the outset of the project was that we would achieve 70% code coverage. Fig. 14 shows the cumulative number of defects found and the known code coverage, and compares the actual

*EQF gives the reciprocal of the average discrepancy between the estimated and the actual duration of a phase. An EQF of 8 is considered to be good for software estimates. See reference 6 for more information about EQF.

```
/*************************************************************/
/* Module  Name : d_ss_read_initialise        File : k_smc_init */
/* Function     : Set up the head conmand for a single shot   */
/*                read from tape. Check the DDC to make sure  */
/*                it is working.                              */
/*************************************************************/
/*                                                           */
/* Parameters   : None                                       */
/*                                                           */
/*************************************************************/
/*                                                           */
/* Globals :                                                 */
/*                                                           */
/* Global  Name             I/O Status           Type        */
/* -----------              ----------           ----        */
/*                                                           */
/* d_report_record             write         device_report_type */
/* d_headcom                   write         command_record  */
/*                                                           */
/*************************************************************/
/*                                                           */
/* Function Result:                                          */
/*                                                           */
/* 0 - SSReadInitOK                                          */
/* 1 - FailedToInitSSRead                                    */
/*                                                           */
/*************************************************************/
/*                                                           */
/* Process :                                                 */
/*                                                           */
/*  Use  d_ss_initialise to do most of the initialisation.   */
/*  If this succeeds,  then                                  */
/*     d_headcom->command_status.read_flag = TRUE            */
/*     d_headcom->comnand_status.write_flag = FALSE          */
/*     Set the mode for ddc head select according to tape type. */
/*     For old HC tapes, reads should use the write heads,   */
/*     otherwise use the  read heads.                        */
/*     RETURN  InitialisedOK.                                */
/*  otherwise                                                */
/*  set d_report_record.device_status.comfail = TRUE         */
/*  return FailedTo Initialise.                              */
/*                                                           */
/*************************************************************/
/*                                                           */
/*  Notes:                                                   */
/*                                                           */
/*                                                           */
/*************************************************************/
/*                                                           */
/*  Author : Kevin Jones                    Date : 5 June 1987 */
/*                                                           */
/*  Modification History                                     */
/*  --------------------                                     */
/*  Modifier              Version       Date      Reason     */
/*                                                           */
/*************************************************************/
```

Fig. 12. *One of the module specifications for the state transition diagram shown in Fig. 10.*

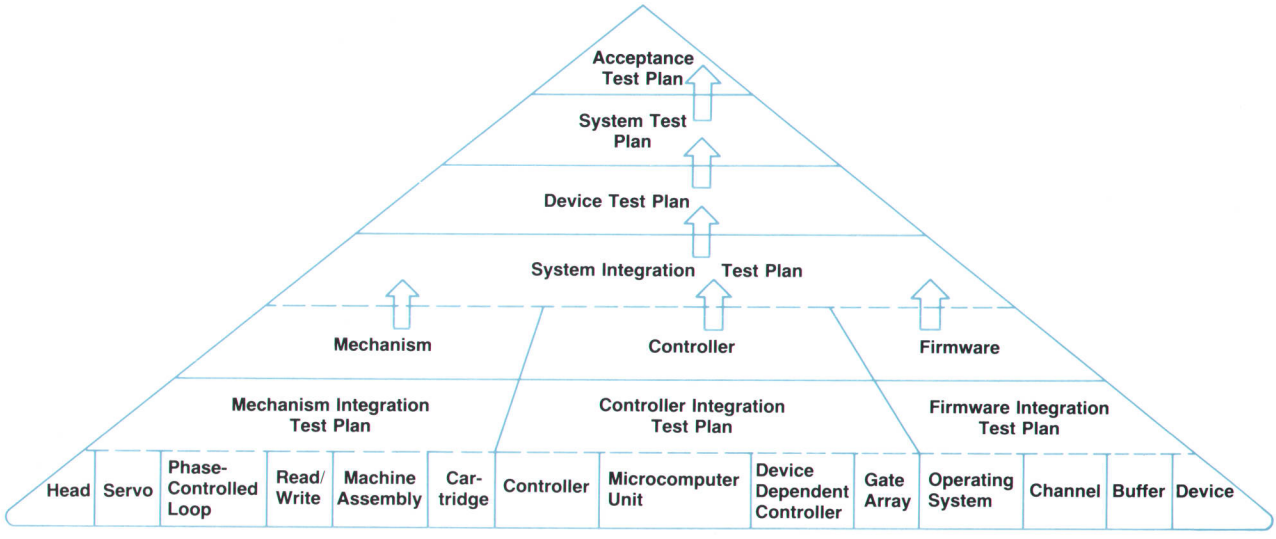


Fig. 13. *Hierarchy of test plans.*

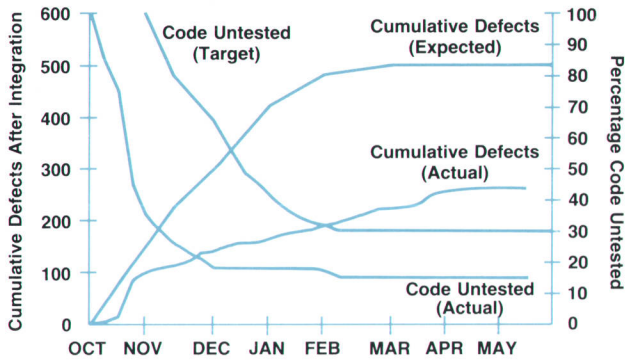


Fig. 14. *Cumulative defect and code coverage.*

figures against the expected figures. This data illustrates the completeness of testing performed.

Conclusion

Structured methods really are appropriate for the development of firmware on major projects. The mistakes that we made were mainly caused by our inexperience with the methods and tools. Access to an experienced practitioner as a consultant or as a member of the team would have eliminated many of our problems. We carried out these firmware development process changes to meet the needs of a particular product development. Our challenge now is to continue to improve our firmware development process and extend what we have learned to future projects.

Acknowledgments

The authors would like to acknowledge the contributions made by Kevin Jones, who worked on the development of the low-level device controller firmware and whose approach to firmware design enabled us to identify the problems involved in using structured methods on real-time firmware, and Ken O'Neill, who developed the automatic test package that enabled us to test the firmware comprehensively and quickly.

References

1. T. DeMarco, *Controlling Software Projects*, Yourdon Press, 1982.
2. P.T. Ward and S.J. Mellor, *Structured Development for Real-Time Systems*, Yourdon Press, 1986.
3. D.J. Hatley and I. A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing, 1987.
4. G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979.
5. W. C. Hetzel, *The Complete Guide To Software Testing*, QED Information Sciences, 1984.
6. R. M. Levitt, "Process Measures to Improve R&D Scheduling Accuracy," *Hewlett-Packard Journal*, Vol. 39, no. 2, April 1988, pp. 61-65.

Product Development Using Object-Oriented Software Technology

Object-oriented technology is rapidly becoming an accepted technology for designing and developing software systems. This paper provides a brief history, a tutorial, and a description of HP's Lake Stevens Instrument Division's experience using the technology for product development.

by Thomas F. Kraemer

OBJECT-ORIENTED SOFTWARE TECHNOLOGY is rapidly changing the way software systems are being designed and developed, and the way we think about and use computers. Over the last eight years HP has been active in the research and use of this technology. HP has completed several products that use an object-oriented approach, and the technology continues to be used in new product development. The essential idea in the object-oriented approach is that data and procedures are represented in a structure called an object, and the data is only accessible through the procedures contained in the object. Also, objects are the basic building blocks for any system designed using an object-oriented approach. The reason for such interest in this technology is that it provides a productive and powerful paradigm for software development, and it addresses such issues as code reuse and software maintainability.

Since object-oriented technology requires a new way of thinking about software development, some questions exist regarding its origin, its difference from established software development methodologies, and the advantages it offers the developer and the end user. This paper addresses these questions by presenting a brief history and some basics of object-oriented technology, and a description of the design and development of a product from HP's Lake Stevens Instrument Division that uses an object-oriented language.

History

Computer science researchers started to look at object-oriented concepts in the late 1960s. The concept of data hiding, in which data is accessed only through a well-defined interface and the data structure is unknown to the

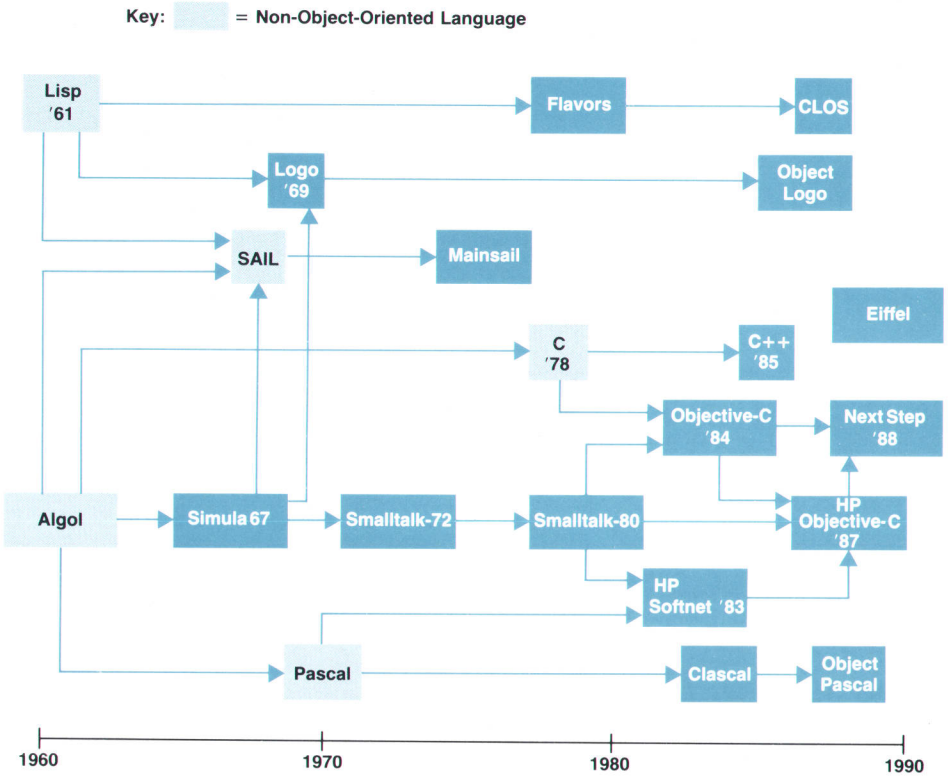


Fig. 1. *Chronology of object-oriented languages since 1960.*

accessing routines, formed the foundation of the object concept. This was followed by the development of abstract data type systems. Abstract data types implement the data hiding concept and include local procedures, which have exclusive knowledge of the data structures within the data type, and therefore perform all operations on the local data. One of the first languages to explore and implement object-oriented concepts was Simula67. Simula67 was developed in the 1960s in Norway for modeling and simulation programs. Fig. 1 shows a chronology of object-oriented languages since 1960. With the exception of Eiffel all of these languages had their origins in a non-object-oriented language. Also, because the technology is still developing there are several competing object-oriented languages and no single approach has become a standard.

Object-oriented programming gained popularity with the introduction of the the Smalltalk language.[1] Smalltalk was originally implemented as part of the research work done at Xerox Corporation's Palo Alto Research Center (PARC) in the 1970s and early 1980s. The language attempted to represent everything from individual bits to whole systems as objects. In 1981 HP Laboratories ported Smalltalk to an HP research computer as part of an investigation of personal computer environments.[2] The fine granularity of its implementation was adequate for research purposes, but because of poor performance Smalltalk was impractical on generally available computers. However, many of the ideas demonstrated by Smalltalk are still models for today's object-oriented systems, and Smalltalk is being commercially developed on today's more powerful workstations.

By 1983 several research and commercial organizations, including HP, were prototyping practical implementations of object-oriented languages. Most of these implementations use a preprocessor that translates object-oriented programs into a conventional language such as C or Pascal. This permits a hybrid approach in which the developer can choose to program in C, Pascal, or even assembly language when appropriate. The idea is to provide an evolutionary rather than revolutionary path toward object-oriented programming. The designer is able to fall back on established methods if the object-oriented approach causes problems.

The artificial intelligence community has produced several object-oriented languages. The language Flavors inspired a series of languages leading to the Common Lisp Object System (CLOS). CLOS includes object-oriented extensions to the Common Lisp system and is primarily an evolving platform used by researchers. It is being designed as part of the ANSI X3J13 Common Lisp standardization process. HP played a role in the invention and development of CLOS. The Stanford Artificial Intelligence Language (SAIL) was based primarily on Algol and evolved into a language called Mainsail which was used in the development of HP's Electronic Design System.

Pascal has been used as a foundation for several object-oriented languages. Additions made to Pascal to implement an object-oriented concept called classes resulted in the language Clascal and more recently Object Pascal. Clascal was used to develop the Lisa computer from Apple Computers. Pascal was used for an early implementation of HP's Softnet which influenced HP's version of Objective-C. HP's Objective-C modifications provided facilities to design systems in which customers could add new features without requiring any changes to the original system, including compiling and linking. HP's use of this modified version of Objective-C is covered later in this paper when the development of HP VISTA is described.

The C language has been used as the foundation for several object-oriented language implementations. The Objective-C language, developed by Stepstone, Inc.[3] is a C preprocessor that permits intermixing standard C code with statements similar to Smalltalk. C++[4] developed by AT&T Bell Laboratories is another preprocessor that generates C code. The latest object-oriented derivative from C is NextStep which is being codeveloped by Next Computers and Stepstone using the Objective-C language.

Eiffel is a relatively new language whose goal is to support software engineering more effectively than C-based languages can. Eiffel is a typed, object-oriented programming language from Interactive Software Engineering, Inc.. It is compiled into C and comes with a class library including graphics classes based on X11 (the X Window System,® Version 11). Eiffel is available on HP 9000 Series 300 HP-UX workstations.

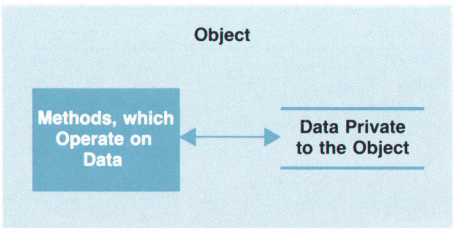


Fig. 2. *Object encapsulation. The object has an internal data structure, which is private to the object, and methods (procedures), which have sole access to the data.*

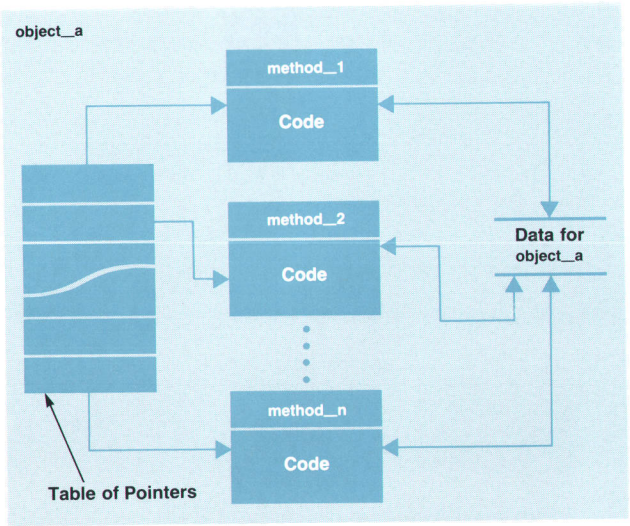


Fig. 3. *Mapping messages to code in a object. When a message is sent to an object the method name (selector name in Objective-C) is mapped to a table of pointers which point to the code segments for the methods.*

Object-Oriented Technology

Object-oriented technology includes object-oriented programming languages and object-oriented design methodologies and processes. Because object-oriented concepts are so very different from conventional procedure-based design techniques and programming languages like C and Fortran, developing an object-oriented design is initially difficult for software designers.

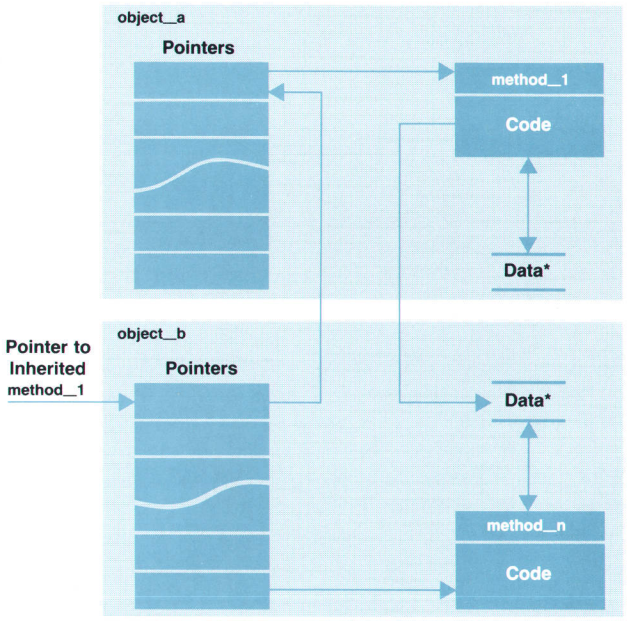
The following sections describe some fundamental concepts about object-oriented technology. Because object-oriented technology is rapidly evolving and researchers are exploring new approaches to object-oriented implementation, there are concepts presented that cannot be fully described in this article.

Encapsulation and Messaging

In procedure-based systems, software developers are concerned with creating global data structures and procedures and functions that operate on the data. In an object-oriented environment, developers are concerned with using and creating objects to build a system. Objects contain local data structures and local procedures to operate on the data. The technical term for this is encapsulation.* The current values of an object's internal data define the object's current state and the object's behavior is dependent on its current state. The concept of data abstraction makes the data inside an object private and accessible only through one of the procedures associated with the object. Procedures inside an object are called methods. Fig. 2 shows a representation of object encapsulation.

Restricting access to data in this manner may appear to be a serious restriction to programmers accustomed to accessing shared data structures directly from any procedure. However, there are some advantages to imposing this restriction:

*This is not the same as NewWave encapsulation described in the article "Encapsulation of Applications in the NewWave Environment," on page 57.



*Data structures are the same.

Fig. 5. *Simple inheritance.* object_a *inherits* method_1 *from* object_b.

- The software can be tested more thoroughly because the data is private to the object and is only modified by the object's methods. A common source of defects in conventional software is a situation in which a shared data structure is modified by one procedure unknown to a second procedure. The second procedure then uses the modified data and generates errors. In contrast the definition of the object's interface can be guaranteed to work when used by other objects because the data structure is private to the object.
- Because all data is accessed by methods associated with the object, the internal representation of the data structure can change without changing any of the software

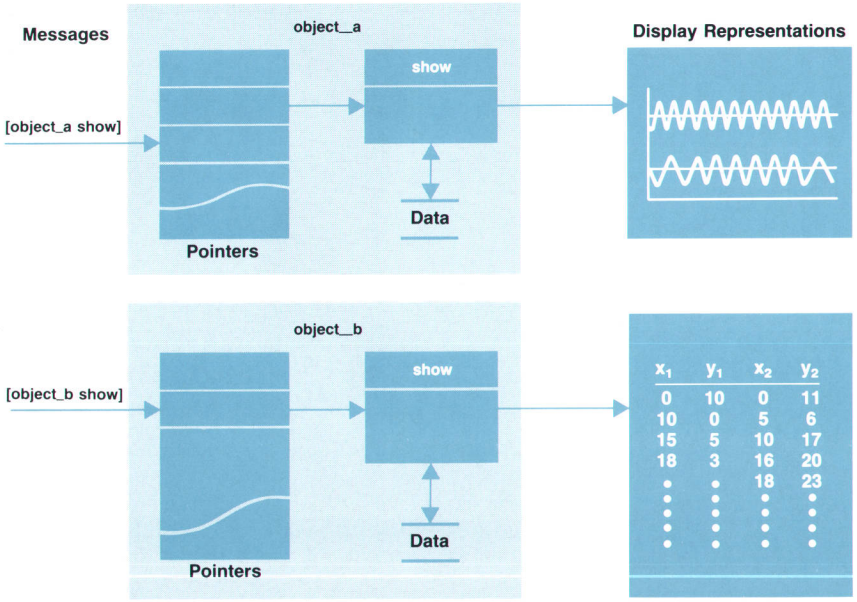


Fig. 4. *Polymorphism allows the same message to be sent to different objects regardless of their internal data types and code implementations.*

that uses the data. This is another common problem in conventional software; it can be difficult to modify a data structure because it requires finding and modifying all references to the data structure. In an object, the methods can be improved and become immediately available everywhere without having to change anything else.

Objects communicate with other objects by sending messages. An object's interface is defined by the messages that can be sent to it. The set of messages an object's interface will respond to is sometimes called an object's protocol. These messages cause methods to be invoked to perform various functions or to obtain data. A message typically consists of the name of the object to which the message is to be sent, the methods to be invoked, and any arguments required. For example, a print message sent to an output object would cause the object to invoke the method responsible for doing printing. See "Objective-C Coding Example," on page 95 for an example of sending messages.

This message scheme is one of the fundamental differences between object-oriented programs and procedural programs. In procedural languages the routine to perform any function is directly invoked by another, whereas in an object-oriented language methods are not directly invoked. Each message to an object is mapped to a table containing pointers to the code segments for each method. Therefore, a method in an object consists of its pointer and code. Fig. 3 illustrates this concept. This scheme enables each object to decide what specific code segment is run to fulfill an external request. The physical implementation of this concept varies between languages.

Polymorphism

Polymorphism in object-oriented methodology means that the same message can be sent to different objects without concern for the method implementations or the type of data structures in each object. Only the functional specifications and the message protocol for dealing with the object are important. The same message may yield a different response depending on the object receiving the message. For example, in Fig. 4 a message called show, which causes an object to display itself on the screen, is sent to two different objects. In object_a the data type represents graphic coordinates and the code performs graphics operations. In object_b the data type is simply an array of real numbers and the code performs formatted print operations. Note that the functional specification (i.e., cause an object to display itself) and the message protocol for show remain the same for both objects. From this example it can be seen that a good application of polymorphism is an iconic user interface, where many different objects, determined by the user, can appear on the screen.

To handle different types in a procedural language, the programmer must anticipate and provide for all the different types of data structures. For example, in Pascal to contend with different data types a programmer might use an IF-THEN-ELSE construct or CASE statement to execute different procedures depending on whether the data structure contains integers or real numbers. This may work initially, but it becomes a maintenance problem if a new data type is added, causing the programmer to modify the program statements to include the new type. Then the program must be recompiled and reloaded.

Inheritance and Ownership

Object definitions can be shared and reused among ob-

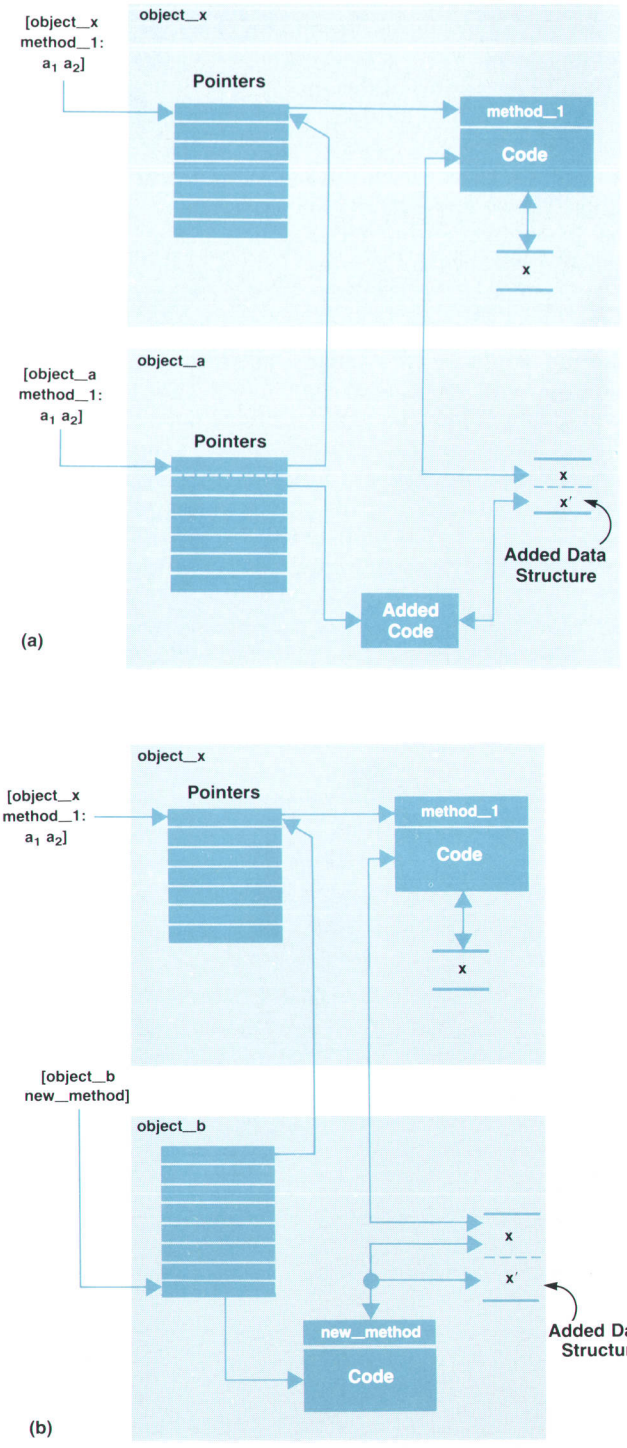


Fig. 6. *Additional approaches to inheritance. (a)* object_a *inherits* method_1 *from* object_x. *New code and data structures are added to* object_a *but the same interface to the object is maintained. (b)* object_b *inherits all the methods from* object_x *and a new method and additional data structures are added to* object_b.

jects without having to make duplicate copies. Also, the functionality of an object can be expanded without modifying the original object. This capability is referred to as inheritance—the object inherits functionality from another object which it can expand and build on by adding data and new methods. Fig. 5 illustrates inheritance: object_b inherits method_1 from object_a. The data structure in object_b that is used by method_1 from object_a must be in the same form as in object_a. However, the data values used by method_1 in object_b are still private to object_b. Method_1 gains access to the data structure in object_b through a pointer that is set up when object_b is created or by a pointer that is passed to it from object_b.

Inheritance can be implemented easily since messages are mapped via a table that points to what code should be executed. This table can just as easily point to another object's message table, thereby inheriting the other object's functionality. In some languages, like Smalltalk and Objective-C, a strict hierarchy is imposed. Inheritance is only allowed from one other class of objects. Other languages allow objects to inherit functionality from more than one class of objects. Advantages resulting from being able to inherit functionality include:

- Inheritance results in a significantly smaller amount of code for identical functionality. This is because objects can be defined incrementally in terms of other kinds of objects.
- Inheritance can provide a convenient way of organizing and describing the relationships between objects. For example, a class of objects called "employee record" contains all of the generic information about an employee such as name, rank, and serial number. If there are multiple types of employees such as managers, professionals, and nonprofessionals, a new type of object could be defined for each of these groups that would include data and operations specific to each group. For managers, this might be the department budget, number of employees, and other data that might not be relevant to the categories of professionals and nonprofessionals. Since each new object type inherits from the same employee object definition, the generic employee methods, such as a request to print out the employee's name, are executed in the same way. These new objects do not need to copy or reimplement the "print" employee method.

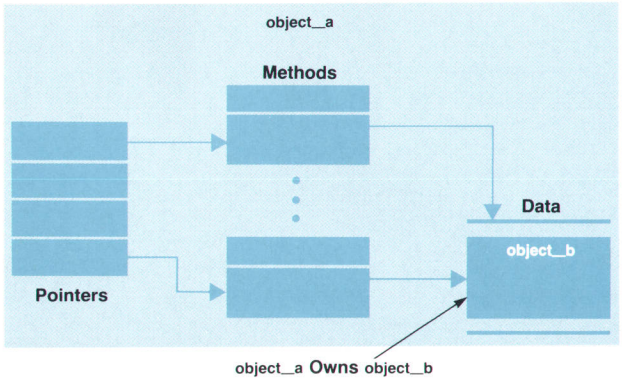


Fig. 7. *Ownership.* object_a *owns* object_b.

There are three typical approaches to inheritance. First, a method can be inherited from another object and used as is with no modifications as shown in Fig. 5. Second, the same method and the same interface to the method can be inherited and modifications to the method can be made. These modifications might include code to handle new data structures as well as the inherited data structure. The same interface means that the inherited and modified method will respond to the same messages it did before. This is illustrated in Fig. 6a where the same message that invokes method_1 in object_x can also be used to invoke method_1 in object_a. Finally, all the methods of object_x can be inherited and new methods and additional data structures can be added to the inheriting object. This is illustrated in Fig. 6b.

Inheritance is sometimes confused with ownership. An object might own another object because the other object is part of its internal data (see Fig. 7). Owned objects communicate with the owner with the same messages as before. However, the owned object is not directly accessible or visible to objects outside the owner.

Class

Many object-oriented languages allow the definition of a class of objects. A class is a template that can be used to create multiple instances. Each instance is an independent object with its own data and state but it shares identical methods with all other objects of the same class. Objective-C and Smalltalk support classes.

There is not a universal agreement about the importance of classes in an object-oriented language. Some object-oriented languages are classless. These include prototypi-

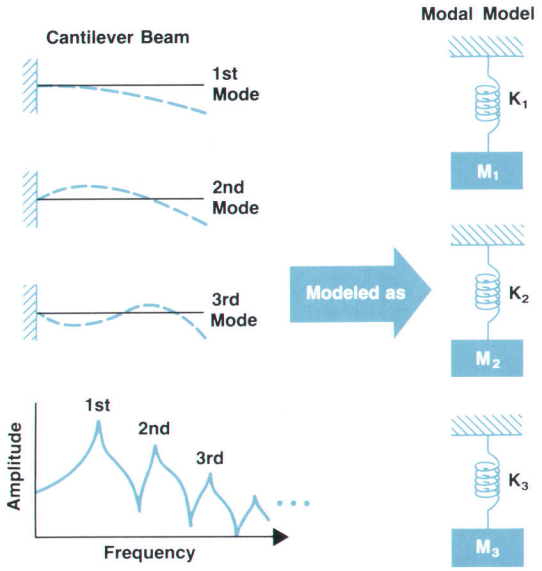


Fig. 8. *Modal analysis and testing. Every structure, such as a cantilever beam, naturally vibrates at several different frequencies. Such a complex structure can be modeled as an equivalent group of decoupled first-order systems whose modal parameters M_i and K_i, which represent the mass and stiffness of the beam, can be derived from a frequency response measurement.*

cal languages which support inheritance, and actor-like languages which support concurrency but not inheritance.

Desired Characteristics

Regardless of the implementation, there are some characteristics that object-oriented languages and methodologies should have to ensure that they continue to be useful as a software development technology. These characteristics include:

- Modifiability. It should be possible to "plug" new software objects into an existing system without having to change the original system. This allows the customer to enhance and expand a system just like plugging new hardware cards into a computer. This has been called a *software card cage* or *software IC*,[5] to use a hardware analogy. In the context of software engineering this means dynamically linking and loading a software module after the program has started running. Today, new functionality is normally added to software systems by recompiling and relinking the entire software system. This is a tedious and time-consuming process and usually impractical to do by end users. Dynamic linking allows a user to add a new object to a system more quickly than relinking and reloading the entire system. This kind of modularity allows software objects to be designed in relative isolation and at different times while remaining compatible.
- Portability. Objects should be able to move from system to system and be usable in a wide range of software products. This should be as easy as the user's popping up a window (possibly connected over the network to another system), selecting the desired object, and then selecting what to do with it.
- Reusability. It should be possible for newly created objects to have functionality that represents an incremental expansion of the functionality of existing objects. This allows reuse without having to waste extra time or memory space to duplicate previous work. There should be catalogs of these standard software objects available for general use.
- Usability. At the user's request most objects should be viewable. With more familiarity and experience the user should be able to view and manipulate objects that may not be prominently visible to a novice user. Each object should be storable and retrievable by the user. The object environment needs to follow modern user interface conventions and de facto standards. It is widely accepted that objects are an excellent way to implement modern human-to-computer interfaces.
- Reliability. Since objects are fundamental system building blocks, each object must be tested as if it were a complete software unit and all the standard structural and functional testing methods must be applied to it.[6] Also, process and defect monitors should be built into every system.
- Performance. Object-oriented systems should not prevent the encapsulation of high-performance or highly tuned software.
- Supportability. Regression test suites must exist to retest objects whenever they are changed to ensure that no new defects are introduced and their interaction with other objects remains the same. This implies that objects must have hooks included to ensure testability.
- Compatibility. Object-oriented systems should work with existing conventional software. This is important to preserve previous investments and allow compatibility with emerging software standards.

These characteristics have been implemented in some object-oriented implementations and some are still topics of research. Together they form a vision of what object-oriented systems are trying to achieve. No system today implements everything. Formal design methods can be used to express and communicate an object-oriented design. This is essential for building complex software systems since it allows a design to be analyzed and improved in a systematic way. Like any other design methodology the successful use of object-oriented techniques depends on good project management and tools, such as a defect tracking system and configuration management.

Recent HP Experience

Several products at Hewlett-Packard have used object-oriented approaches. They include the NewWave environment,[7] a chemical data editor, a dynamic signal measurement and analysis system, and an electronic engineering computer-aided design system.

NewWave objects typically correspond to the kinds of data ordinarily associated with traditional office application programs: complete documents, spreadsheets, charts, drawings, and so on. The most noticeable feature of a NewWave object is that a user never needs to know what program actually manages the object. The user simply asks to

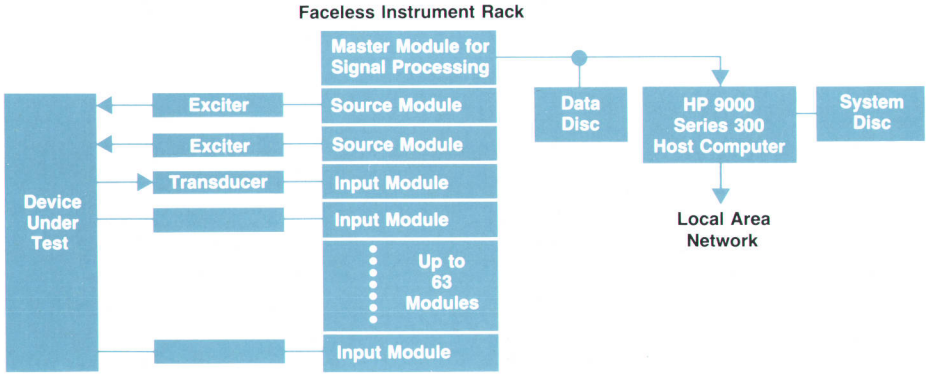


Fig. 9. *Hardware block diagram for the HP 3565S Signal Processing System.*

perform some operation on an object and the NewWave object management facility (OMF) automatically associates the object with the correct application.

The chemical data editor is part of an HP ChemStation used to perform chemical analysis. The editor portion is written in an object-oriented manner by using Pascal records as objects containing a pointer to a class dispatch table. Seven classes of objects represent the different data types corresponding to chemical spectra. The editor allows the user to manipulate the data. The key benefit of object-oriented programming to this editor is the ability to treat different styles of spectra (data types) in a similar way. This makes the manipulation of the test results much easier for the user.

HP's Electronic Design System is written in the Mainsail language with object-oriented extensions. This system allows software objects to be created incrementally and linked dynamically to the existing system. Polymorphism is used to help implement generic routines that handle a wide variety of computer-aided design data.

Recently, HP's Lake Stevens Instrument Division introduced the HP VISTA software product which is a dynamic signal measurement and analysis system implemented with the Objective-C programming language. HP VISTA is described in more detail in the next section.

More HP products are under development using object-oriented technologies. In many cases the use of objects is not directly observable by the end user. In other cases the user has explicit control over creating and using new objects.

Design Example

There are no cookbook procedures or formal methods that can be used to generate and analyze an object-oriented design and this is still an area of active research. The architecture documented here illustrates one way of applying object-oriented techniques for product development.

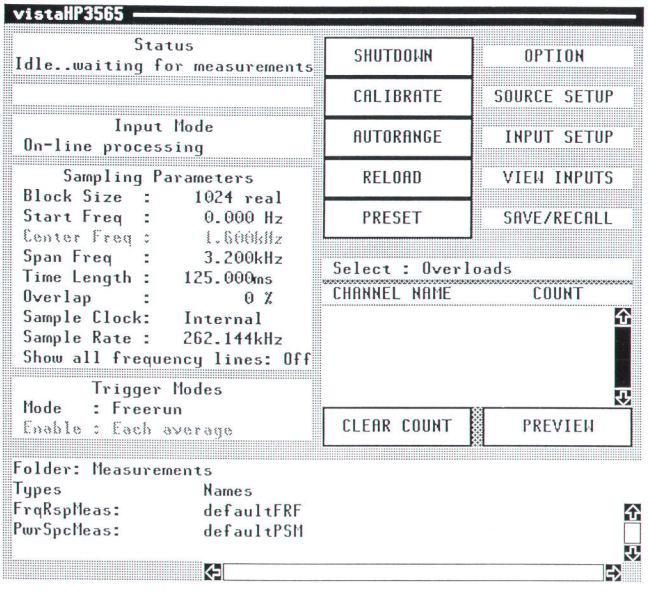


Fig. 10. *Typical HP VISTA display showing a virtual instrument screen.*

HP VISTA

HP VISTA is the software portion of the HP 3565S Signal Processing System. The system is used for dynamic signal analysis. Real and imaginary components of electrical signals are measured and digital signal processing techniques, like the fast Fourier transform, are used to calculate the frequency-domain components. The measured electrical signals can come from a wide range of sources and transducers. Knowledge gained from these measurements can be used to improve a design or detect faulty machinery.

One common application of a dynamic signal processing system is in the design and testing of mechanical structures (modal analysis). A mechanical structure, such as an airplane body, can be drawn on a computer-aided design system and analyzed with finite element methods. A prototype can be built and tested by measuring the outputs of accelerometers attached to different points on the structure while shaking the structure with a random input.

The random input has a broad bandwidth and will cause all of the modes of vibration of the structure to be excited. The frequency spectrum plot of each transducer will peak at a frequency associated with the natural vibration of the structure. A structure can be formally modeled by parameters that describe each mode of vibration (see Fig. 8). The resulting modal model is useful for simulating modifications to the structure.

The HP 3565S hardware (see Fig. 9) provides a rack that can hold up to sixty-three input modules. Each input module contains analog-to-digital converters for data acquisition, memory buffers for one or more input channels, and hardware dedicated to digital filtering and zoom analysis. The master module contains a dedicated 16-bit processor and additional digital signal processing hardware to fetch data from each input buffer for windowing, transforming, and uploading data to a host computer via an HP-IB (IEEE 488, IEC 625) interface.

The HP VISTA software resides on the host computer and provides the user interface and processes for interactive control of the measurement and analysis functions and stimulus-response testing. It is also responsible for downloading software to the master module in the signal processing hardware. The host computer is an HP 9000 Series 300 Computer running the HP-UX operating system. The multitasking capabilities of the HP-UX operating system allow HP VISTA to perform several measurements simultaneously, plot results, write a report, and receive electronic mail from a local area network. Measurement results can be used directly by the design and analysis tools. The HP-UX system also provides real-time extensions such as a preemptive kernel, shared memory, and memory lockable processes, which are required to handle instrument I/O.

Selection of an Object-Oriented Language

The developers of the HP VISTA product had previous experience with several large instrument firmware projects written in Pascal. The complexity and size of HP VISTA required a new software engineering approach. Since the range of applications for dynamic signal analysis is very large, the major goals for HP VISTA were to provide a system that enabled customers to configure their own software and add new hardware into the system. It was also

important to be able to add new features to the product incrementally.

Object-oriented software was able to satisfy these goals. The HP VISTA product was initially based on the Softnet technology developed by a group at HP's Ft. Collins Engineering Operation. These concepts were ported to C for compatibility with the HP-UX system. This C-based object-oriented system was enhanced to include encapsulation, messaging to support polymorphism, and inheritance.

During the early development stages of HP VISTA, a group at HP's Loveland Measurement System Operation was using and and extending the Objective-C programming language. The HP VISTA C code was ported to Objective-C halfway through the project to permit sharing of code and ideas with other HP development groups. Software engineering issues such as source code control and configuration management were worked out in parallel (see "Object-Oriented Life Cycles" on page 98).

Faceless Instruments

The use of a general-purpose HP workstation as the primary interface to instruments is a fairly new concept. Instruments traditionally have been contained in a dedicated box with a display and buttons on the front panel for control and observation. The development of the HP-IB interface permitted these stand-alone instruments to be controlled from a central computer, but the primary interface remained via the front panel.

A "faceless" instrument transfers the primary interface from the front panel of the instrument to the screen of a general-purpose computer. This makes it easier to interface with a group of instruments, and it is a lower-cost solution since it eliminates the redundant displays and front panels on each instrument. The HP PC Instruments[8] family was the first HP product line to build faceless instrument modules with the user interface displayed on a personal computer screen. These software displays mimic the buttons and displays seen on a normal instrument front panel.

HP VISTA software also controls a group of faceless instruments that can support hundreds of input channels. Because of the number of input channels, the HP VISTA computer display cannot duplicate a standard front panel on its smaller display. A method was devised for controlling and viewing measurement activity via a windowed user interface (see Fig. 10). HP VISTA does this by using the standard HP Windows/9000 system. Each object in HP VISTA is capable of displaying itself in a window on the screen. The user can also have other software applications displayed simultaneously in other windows. For example, the user can view a drawing of a device while performing measurements on it. It also allows the user to reconfigure the display as required since windows can be added, deleted, and moved to different regions on the screen.

The user interface required several iterations to achieve the goals of user configurability and ease of use. One version of the design considered use of the entire display filled with tiled* information. The display tiles are nonoverlapping and adjust in size if additional tiles are added to the screen. The advantage to this approach is that it provides the user with a consistent view of the system. The disadvantage is that it is difficult to add new objects as they are developed and the user may not want the exact display chosen by the program. Another design proposed to display every object in a directory from where it could be selected and displayed. The advantage here is total flexibility. The major disadvantage is confusion by new users on where to start work. Also, each window creates a new HP-UX process, and too many processes can reduce system performance. The final design for the user interface settled on limiting the display windows to a small number of objects and tiling objects within these windows in a controlled manner. Different-sized displays can be accommodated with this technique and still provide the user with the advantages of windows.

Object Model/View/Controller

The HP VISTA system is based on three types of objects: model objects, view objects, and controller objects (see Fig. 11). This organization is similar to that used in the Smalltalk language.

In HP VISTA, model objects contain data about the hardware, data collected from measurements, and data associated with computations. They also contain the methods to retrieve and manipulate the data. View objects own model objects and they contain the methods to draw the information contained in the model objects on a display.

*Tiled information refers to a window system in which windows are shaped and sized to fit together without overlapping each other.

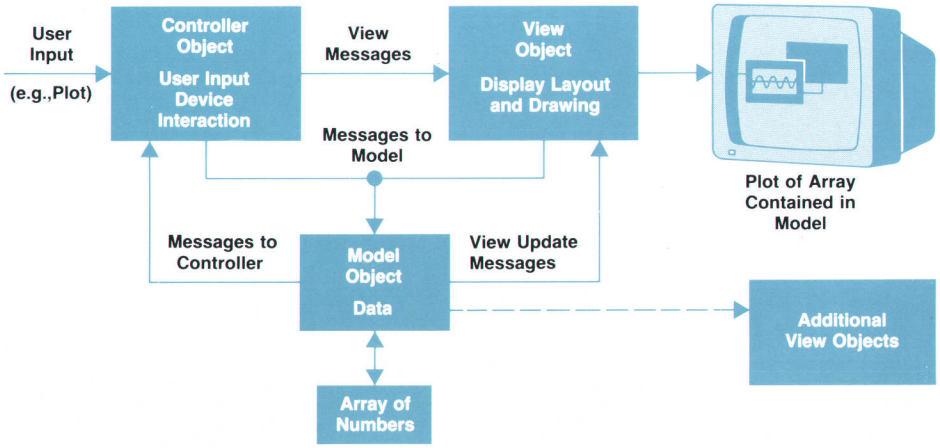


Fig. 11. *Relationship between model, view, and controller objects in HP VISTA.*

Objective-C Coding Example

The HP VISTA software is written in the Objective-C programming language. Standard C code can be freely intermixed with Objective-C code because a preprocessor translates Objective-C code into standard C. Message expressions are written as a pair of balanced square braces surrounding the receiver, selector, and arguments.

```
[receiver selector:argument]
```

The receiver is the name of the object that will receive the message. The selector is the particular method that should be performed by the receiver. Arguments are parameters which can be either simple data or other objects.

Message expressions are allowed anywhere function calls are allowed in C. For example, a message expression can appear as arguments in the standard C print function:

```
printf("The size of the set is %dn", [aSet size]);
```

The standard C function, printf, will print out the string, "The size of the set is N", where N is the number returned from the object aSet after the method size is executed.

Other objects can be sent along with the message as arguments with the following syntax:

```
[anObject do:arg1 with:arg2]
```

This statement says to select the method do:with: from the object anObject, and use arguments arg1 and arg2. For example, to sum two coordinate point objects, coordx and coordy, they could be sent to a summation object aSum with the message:

```
[aSum x: coordx y: coordy]
```

The Objective-C preprocessor converts this syntax into standard C function calls. The preprocessor also maps these messages into a more efficient form than if the function calls had been written directly using string values. Also, because the messages are symbolic and do not refer to any specific function, they can be dynamically bound to any particular function at run time. This permits the efficient implementation of inheritance, polymorphism, and the ability to link and load new objects dynamically.

View objects send messages to model objects to retrieve the information to be displayed.

With the model object separated from the view object each can specialize in what it does best. A single model object can be viewed in more than one way by creating multiple view objects that own the same model object. In Fig. 12 a single model object that contains the current value of the time of day is owned by two view objects. The view objects use the time-of-day value from the model to display two different representations of the time on the display. Likewise, a single view object can be used to display many different model objects. Because of the dynamic linking and loading capability of objects, new objects can be dynamically added to a view object, thus eliminating the need to provide for all combinations of models and views.

The controller object handles all of the user's input devices, such as the mouse and keyboard. The controller interprets the user inputs and maintains information about the state of the input devices. The inputs can even come from a computer-generated source such as an automatic test program. A standard controller object can be used by many different view objects.

Views and Subviews

HP VISTA has several general classes of view objects that are widely reused either through inheritance or simply by reusing the whole object. These view objects include text displays, scroll bars, pushbuttons, and so on. A window display can be constructed from several of these view objects in a hierarchical manner. The subviews of a view object are owned by and nested within the view when it is displayed (see Fig. 13). Subviews are usually added to a view when it is created and are set up to display the individual objects that make up the model object.

Each view has the ability to calculate its size automatically from the positions and sizes of its subviews. This permits subviews to change in size without requiring any changes in code. A subview can operate as it would by itself, or it can transfer responsibility to its superview. This allows general-purpose subviews to be created which can be selectively overridden by the superview.

Dependent View Objects

Each view object can automatically update the display whenever the model object changes. The model object maintains a list of all objects that want to be informed whenever a change takes place. This list is called a dependency list. An object can request to be added to this list by sending a message to the model object it wants to be notified by when something changes. HP VISTA added this capability to the Objective-C system.

Exception Handling

To prevent the system from crashing because of errors

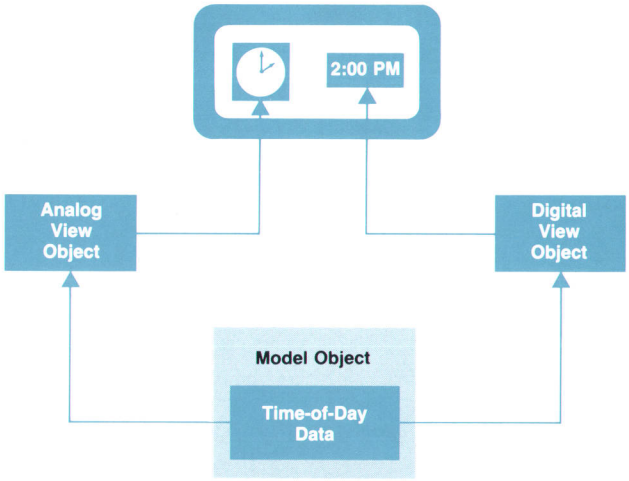


Fig. 12. *Two view objects own the same model object. The model object contains the current time of day. One view object displays the time in analog format and the other view object displays the time in digital format.*

in user input or measurement computations, Objective-C was extended to include a try/recover exception handling capability. Different sections of the code are declared a try region, where if anything disrupts the normal sequence of processing, such as division by zero or a hardware error, the recover block of code is executed either to patch things up and continue, or to inform the user that a problem exists. Every attempt was made to provide the user with helpful error messages so that any problem can be corrected and a successful measurement made.

A try/recover block was also placed around the entire HP VISTA system. The system is then protected against all undetected software defects. These defects are trapped and the user can continue to operate the system. Some of these undetected defects might cause the system to freeze up and require the user to restart the program. Despite rigorous quality assurance testing, it is well known that software defects can remain undetected for a long time. Object-oriented programming improves this situation but it will not guarantee zero defects.

HP VISTA Architecture

Between the model and view object categories, HP VISTA has more than 300 different types of objects, which are further categorized into three groups: virtual instrument objects, measurement objects, and result objects. Each group is made up of model and view objects, that is, there are virtual instrument model and view objects, measurement model and view objects, and result model and view objects. These categories and the relationship between the objects and the display windows are shown in Fig. 14.

Virtual instrument objects encapsulate the instrument hardware, which includes input modules, source modules, and the hardware configuration setup. All of the software in HP VISTA interfaces with the hardware via the virtual instrument objects.

The measurement objects perform generic measurement functions such as arm, pause, continue, abort, and averaging, as well as functions specific to a particular measurement type. Typical measurement types include frequency response functions and power spectrums.

The result objects store the computed measurement data. This data is corrected for calibration variances and is available for display by various view objects. The result objects can also be used for additional computation and processing by the user.

The measurement folder, which appears in the virtual instrument display window, lists all of the measurement objects accepting data from the virtual instrument (see the bottom window in Fig. 10). The user can request that multiple measurements be computed simultaneously from the data collected by the virtual instrument objects.

A standard interface is defined between the virtual instrument objects and measurement objects as well as between the measurement objects and the result objects. These interfaces permit one category to change its implementation without affecting the other two categories of objects. For example, as new instrument hardware is introduced, only the virtual instrument objects need to change. The measurement and result objects can remain the same.

Virtual Instrument Objects and Dependency

All instrument hardware handled by HP VISTA is encapsulated by virtual instrument objects. The term virtual instrument was chosen to represent the fact that the software representation is a virtual image of the instrumentation hardware. The letters of the word VISTA stand for Virtual Instrument System for Test and Analysis. The virtual instrument objects encapsulate the hardware setup state, the input module states, and source module states.

Virtual instrument objects read and update the state of the hardware. HP VISTA's object dependency feature described in a previous section is used to automate updating these objects whenever the state of the hardware changes or new modules are added. The use of dependency eliminates the need for each object to poll periodically for any changes.

Each hardware input module is represented by an input module object. Input module objects are displayed in the input setup window. When an input module object changes for any reason, it informs the appropriate virtual instrument view object that a change has occurred. The virtual instrument view object can then take appropriate action. A common use for this scheme is to indicate when a new block of data has been acquired. The input module loads its buffer memory and signals all of its dependent objects that it has new data. The objects signaled can then choose to use or ignore the new data.

A standard interface exists between the virtual instrument objects and measurement objects. The virtual instrument collects one block of data from each input module and creates an object containing an array of data block objects. This general-purpose object contains the data from the input module object and methods for performing measurement calculations (e.g., add two blocks of data, complex conjugate addition, etc.). This object can be used by all measurement objects. Measurement objects can also make requests to the virtual instrument objects to activate instrument source modules through a standard protocol of messages.

Concurrency

The process of collecting data and passing it on for further computation is effectively an endless loop. A deferred messaging system was created in HP VISTA to allow messages from the user to be executed while a measurement is taking place. For example, the user may wish to change the voltage

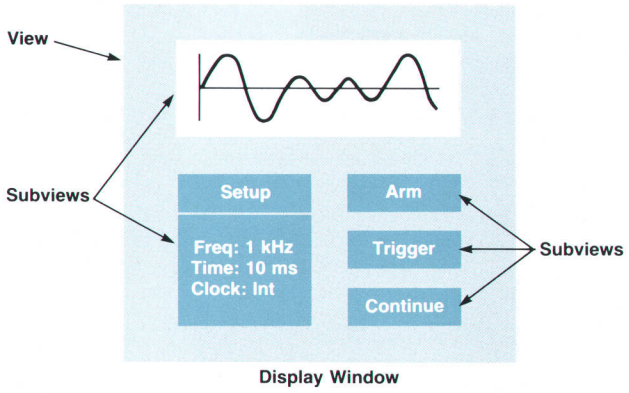


Fig. 13. *Display window constructed out of view and subview objects.*

setting of a source output while performing some measurement process. Instead of forcing each object to waste CPU time in a software loop that polls for new messages, a deferred message queue was created. This queue is a list of messages waiting to be executed.

All messages sent to objects on the dependency list are sent to this deferred message queue. Also, all messages that result from the user's pushing a button or selecting a menu item are sent to this deferred message queue. The main measurement loop is divided so that a deferred message is sent frequently enough to objects on the dependency list to allow a user's message to be executed in a timely fashion. This creates the illusion of two operations occurring at the same time even though there is only one program and one HP-UX process. Users can modify and display instrument settings interactively while watching on-line updates of the measurement in progress. The spawning of extra processes is not required and objects do not need to poll for input—they just need to respond to messages sent to them. Significant performance advantages are achieved by reducing the number of processes required.

Measurement Objects

Measurement objects perform specific measurement functions using the data collected by the virtual instrument objects. The concept of inheritance is used by all measurement objects to provide new measurement functions incrementally. All measurement objects inherit key functionality from a super class object which implements a state machine that handles all generic measurement functions, such as arm, pause, continue, abort, and averaging.

Several different measurement objects can make themselves dependent on the virtual instrument objects. Each time a new set of data has been collected, a virtual instrument object informs each object on its dependency list that new data is available. More than one measurement object can be on a dependency list to permit the simultaneous calculation of several measurement results from the same data.

One specific measurement performed by HP VISTA is a multiple-input, multiple-output (MIMO) frequency response function. There is a significant increase in measurement throughput and measurement accuracy associated with processing multiple simultaneous inputs. The calculation of this measurement requires a matrix computation combining data from multiple channels.[9] The MIMO frequency response measurement object uses the data collected by the virtual instrument objects to do these computations.

Other measurement objects perform 1/3-octave measurements, power spectrum measurements, and time-domain measurements. Depending on the application, one or more measurement objects might be using the virtual instrument output simultaneously. These measurement objects are set up and started in measurement windows on the display. The measurement windows sometimes contain views of the measured results.

(continued on page 99)

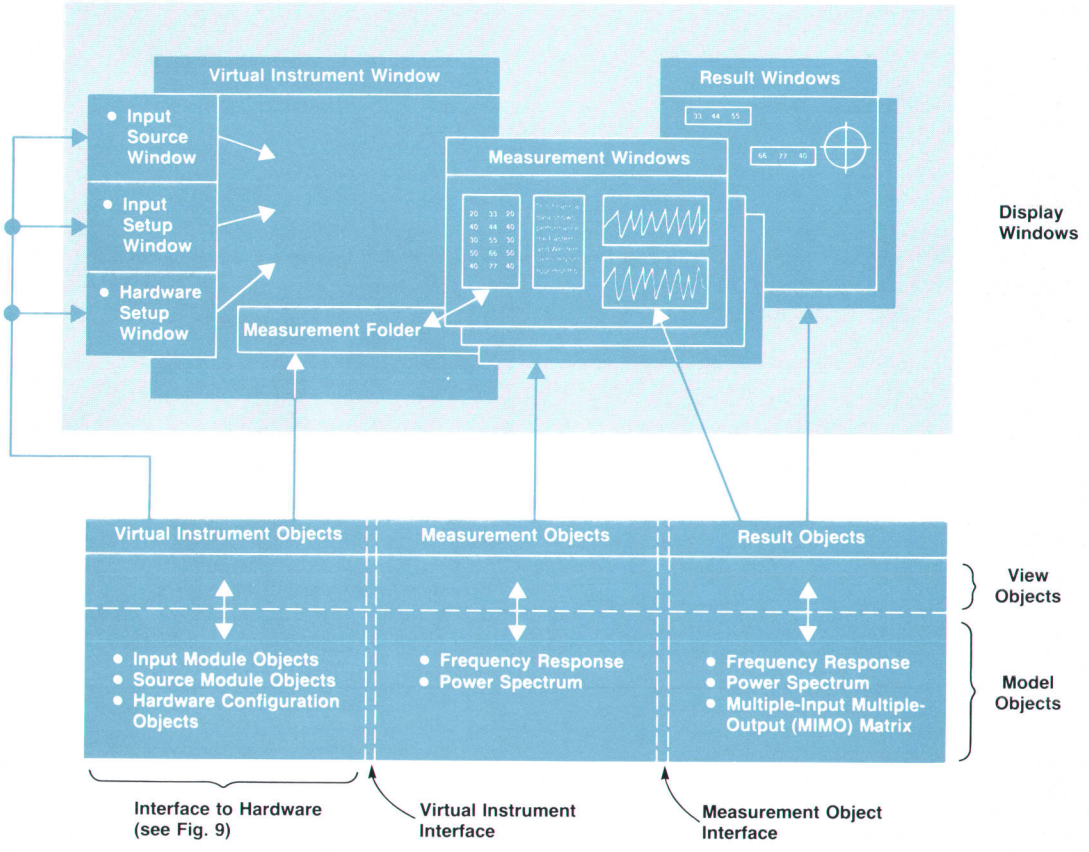


Fig. 14. *HP VISTA architecture.*

Object-Oriented Life Cycles

An independent metrics group was formed early in the HP VISTA development process to help measure and improve the software development process.[1] It was not clear whether an object-oriented development process would fit into a conventional software development life cycle. Some of the life cycle models considered included:

- A code and fix model—write some code and fix the defects found. Detailed design is not considered at the beginning.
- A stagewise and waterfall[2] model—separate the development process into a design phase, a coding phase, and a testing and integration phase.
- An evolutionary development model—incrementally build rapid prototypes to gain early user feedback for subsequent improvements and evolution. This approach can easily have all of the problems of the old code and fix model. This is a popular approach with fourth-generation languages and spreadsheet applications when the developer does not know what is required until the solution is shown.
- The transform model—develop a formal specification that is transformed into an optimized implementation. The objective is to have an implementation that can be proved correct and error-free. This method is primarily an area of research.

Previous project experience indicated that none of these approaches was a total solution. A common belief shared by the design team was that code should be quickly prototyped to gain operational understanding and experience and then thrown out. The code can then be rewritten and constructed in a cleaner and more organized fashion based on the quick prototype experience. The only problem with this approach is that because of the pressure to release a product quickly the prototypes might be shipped, resulting in maintenance problems because early prototypes typically consist of patched and poorly structured code.

In addition to creating and managing a development life cycle other goals of the metrics program were to develop methods to:

- Show the progress of a project throughout different phases in its life cycle.
- Indicate when a project had passed from one phase of the life cycle to the next.
- Monitor changes to the product's definition, internal design, and code.
- Improve resource and schedule planning to help coordinate project staffing, determine development costs, and forecast future schedules.
- Flag any new process problems that a project might have.

It was made clear to the design team that the metrics program was designed to measure projects and processes and not individual performance. The goal was to determine ways in which the process could be improved rather than find out who was writing the most code. This noncompetitive approach along with an independent metrics group was considered essential to acquiring good, nonbiased information to improve the software development process continually.

A waterfall model, modified to take into consideration the needs of object-oriented development, was used to define our development life cycle. The major phases included:

- Definition phase, in which the external specifications of the product functionality are defined.
- Design phase, in which the system design is partitioned into objects and the internal design and functionality of each object are defined.
- Coding phase, in which the objects and other modules are created, tested, and debugged.
- System testing phase, in which independent testing of the entire product is performed. During this phase the system is 100% functional with some bugs and performance problems remaining.
- Production release phase, in which the product is released with no remaining known defects and an acceptably low defect rate.[3]

For each phase metrics were defined and collected to determine if that phase was completed (see Table I). To encourage minimum development time, different parts of the project were allowed to be at different points in the life cycle. Graphs of the metrics collected were used by management for planning purposes and detecting process problems.

Table I
Software Metrics Set by Life Cycle Phase

Definition Phase:

- Cumulative Engineering Months

Design Phase:

- Above Set, plus
- Object Classes Planned, Designed
- Methods Planned, Designed
- Productivity: Engineering Months/Classes Planned and Designed
- Productivity: Engineering Months/Methods Planned

Coding Phase:

- Above Set, plus
- Object Classes Tested, Released
- Methods Coded, Tested
- Total KNCSS (Thousand Lines of Noncomment Source Statements)
- Link Success Rate
- Compile Success Rate
- HP-UX **make** Time
- Productivity: Engineering Months/Classes Released
- Productivity: Engineering Months/Methods Released
- Productivity: Engineering Months/KNCSS

System Testing Phase

- Above Set, plus
- Defects Found, Resolved
- Cumulative Defects versus Cumulative QA Hours
- Defect Finding Rate
- Estimated QA Hours Remaining
- QA Hours Per Week
- Unresolved Defects by Severity

Over All Phases:

- Cumulative Engineering Months
- Productivity: Engineering Months/Classes Released
- Productivity: Engineering Months/Methods Tested
- Productivity: Engineering Months/KNCSS

References

1. C. Stanford, "Managing Software Projects Through a Metrics Driven Lifecycle," *HP Software Engineering Productivity Conference*, May 1987, pp. 4-232 to 4-241.
2. B.W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, May 1988, pp. 61- 72.
3. G.A. Kruger, "Project Management Using Software Reliability Growth Models," *Hewlett-Packard Journal*, Vol. 39, no. 3, June 1988, pp. 30-35.

Result Objects

Results computed by the measurement objects are stored in special result objects that include special information about the measurement. This information includes data correction information, units, input range factors, pointers to where the measurement is documented, and so on. The result objects inherit functionality from general-purpose math objects. These results are displayed in result and trace view windows.

Result objects can be saved on disc by using an automatic I/O feature of HP's Objective-C. Every object in HP VISTA has the ability to save itself on request. The save procedures are nested so that a single message to an object will save the object and all of the objects it uses. Therefore, the user can save just the results, the virtual instrument setup objects, the measurement setup objects, or everything except the actual hardware.

Each result view object can be a dependent of a result model object. Separating the view function allows each measurement result to be displayed in more than one way by creating a new view object. Also, more than one view object can be simultaneously displaying the measurement results. Each time the measurement object computes a new result, it sends a message to a result model object informing it that there has been a change. The result model object then informs all of the result view objects on its dependency list that it has new results. The result view objects can then take the new results and update the trace plots in the display windows. For example, a user can be viewing the same measurement results both as a frequency response plot and as a Nyquist plot in two different windows.

The user of HP VISTA has a large amount of control over what measurement objects and results objects are created and displayed. This makes setting up a custom measurement display easy. The user can open the specific measurements and results views required and position these on the screen where desired using standard HP Windows/9000 menu commands. For users not requiring this flexibility, objects can be created that will manage and perform a specific measurement application. These applications require less knowledge about the windows and object system and are easier to use.

A user programming language is provided by HP VISTA to permit users to send messages to any object visible in the system. With this feature the user can create automatic scripts to set up and perform measurements without human interaction. The user can also create a user program to process the results further if required by the application.

Performance

Since HP VISTA measurements must be performed in real time and on-line, consideration was given to designing the system for performance. Common folklore says that object-oriented systems are slower than conventional systems because data can only be accessed by sending a large number of messages.

HP VISTA was designed with an object-oriented language that permits the inclusion of standard C code where required for performance reasons. Early in the design an estimate was made concerning what items should be objects and what items should be coded in C to increase performance. For example, direct pointing into an array of numbers was allowed in a controlled manner since millions of array computations are performed.

Most of the anticipated bottlenecks did not become problems. Optimization work on the Objective-C language by Stepstone Corporation and a group within HP resulted in message execution times only a fraction slower than a standard procedure call. Several major performance problems were solved during a tuning phase by modifying the offending algorithms and repartitioning some objects. Performance was also improved by measuring and determining where the bottlenecks were and then rewriting those sections of code. This included eliminating objects in some places and creating new objects elsewhere.

A common design issue related to object-oriented programming is the granularity of objects created. A fine-grain object is one that performs a very small function in the overall system. A large-grain object is one that encapsulates a large amount of functionality. A common perception says that a system with fine-grain objects will have poor performance because of the large number of messages required. On the other hand, a system with too many large-grain objects will not provide the user with many of the advantages of objects.

HP VISTA designers discovered that both large-grain and fine-grain objects must be carefully chosen and designed to balance system features with performance. Granularity is an important system design issue that is still not well-understood. For example, a very fine-grain integer object was created to aid in the display of integer numbers on the screen. This object turned out to be a good use of an object since it was able to fit into the standard view object structure and was updated automatically at a fairly low frequency. Only a couple of messages are sent to it and so it is not a performance bottleneck. However, this is a poor object to use in a numerically intensive application such as a fast Fourier transform which requires fast and frequent access to each integer value in an array. A faster design would create an object with a fast Fourier transform method coded using an array of integers rather than an array of integer objects. This array of integers would be classified as a coarser-grained object.

Additional performance tuning was done with HP VISTA's interface to the HP Windows/9000 system and the HP-UX operating system real-time extensions. An additional memory-locked process was created to store incoming blocks of data into an area of memory shared by the main HP VISTA program. This memory queue buffers the statistical time variations caused by hardware and operating system response time.

Memory Management

The creation and destruction of objects requires that system memory be allocated and deallocated. These actions can result in two problems: memory can be fragmented into small and unusable pieces, and memory can run out because memory space occupied by old objects is not reclaimed. The process of cleaning up memory is often referred to as garbage collection. The program must periodically reconfigure system memory space to avoid running out of memory for new objects.

The garbage collection process can take a significant amount of time and could interfere with a real-time measurement process. HP VISTA designers carefully designed the main real-time measurement loops so that memory management is not required during operation. This is done by recycling old objects every time through the measurement loop and avoiding the creation or destruction of new objects. Outside of the measurement loop, a simple memory garbage collection takes place in the background. The user can also request a more complete garbage collection when spare time is available.

Summary

Object-oriented technology represents a fundamentally new way of creating software. It has not yet been quantified into a textbook procedure. To be successful, object-oriented approaches must be applied in conjunction with good project management and tools, such as a defect tracking system and configuration management. There are no panaceas that will eliminate software engineering problems, but object-oriented approaches will help produce software that is far more tolerant of change. HP VISTA and other HP products are examples demonstrating these advantages.

Acknowledgments

I would like to acknowledge the vision, determination, and support shown by Howard Hilton to build an object-oriented software product. Howard served initially as section manager and later as lab manager for the HP VISTA product development. This product could not have been done without Howard's insights and understanding of object-oriented technology. The HP VISTA product was able to implement only a small portion of Howard's vision of how software should be done. Also, the Instrument System Laboratory under the leadership of David Palermo helped to develop and champion these ideas companywide. John Uebbing and Charles Young of HP Laboratories demonstrated what could be done with Objective-C and real-time measurement graphics. The HP VISTA design team, too numerous to list here, invented many innovative solutions while pioneering in no less than six major technologies: a new real-time HP-UX operating system, a new HP 9000 Series 300 workstation, new signal processing hardware, object-oriented programming, faceless instrument and windowed user interface, and multiple input/output frequency response measurements. The design team's solutions to technological problems in each of these areas helped in the successful completion of the HP VISTA product.

References

1. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
2. G. Krasner, *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 1983, pp. 79-112 and pp. 207-237.
3. B. Cox, *Object-oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.
4. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.
5. B. Cox and B. Hunt, "Objects, Icons, and Software ICs," *Byte*, August 1986, pp. 161-176.
6. S. P. Fiedler, "Object-Oriented Unit Testing," *Hewlett-Packard Journal*, Vol. 40, no. 2, April 1989, pp. 69-74.
7. *Hewlett-Packard Journal*, this issue, pp. 6-64.
8. *Hewlett-Packard Journal*, Vol. 37, no. 5, May 1986, pp. 4-32.
9. T. Kraemer, "Software Architecture for a Multiple Input/Output Dynamic Signal Analyzer," *4th International Modal Analysis Conference*, 1986.

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

ADDRESS CORRECTION REQUESTED

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

HEWLETT-PACKARD JOURNAL

August 1989 Volume 40 • Number 4

Technical Information from the Laboratories of Hewlett-Packard Company
Hewlett-Packard Company, 3200 Hillview Avenue
Palo Alto, California 94304 U.S.A.
Hewlett-Packard Marcom Operations Europe
P.O. Box 529
1180 AM Amstelveen, The Netherlands
Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan
Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

00199127 HPJ 8/89
GEORGE PONTIS
SUITE 409
1742 SAND HILL RD
PALO ALTO, CA 94304

CHANGE OF ADDRESS:

To subscribe, change your address, or delete your name from our mailing list, send your request to Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304 U.S.A. Include your old address label, if any. Allow 60 days.

5953-8576