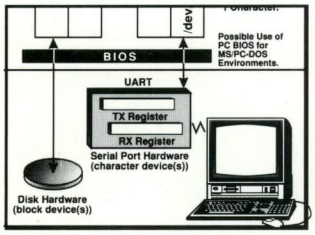
NOVEMBER 1989 • VOL4 NO6 • TEN DOLLARS

Microsoft®

SYSTEMS JOURNAL

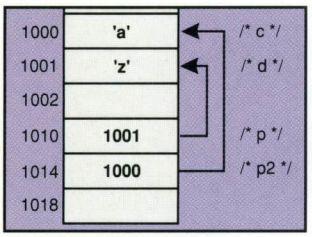
01 Examining NewWave, Hewlett-Packard's Graphical Object-Oriented Environment

Hewlett-Packard's NewWave is part of the new generation of graphical environments based on object-oriented programming. This article explores the concept of object-oriented programming and how it relates to the NewWave environment, then presents some sample applications and scripts.



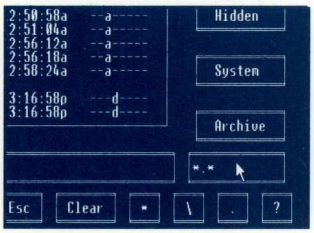
19 Emulating the UNIX® RS-232 General Serial I/O Interface Under DOS

RS-232 serial communication is a means of transferring information between DOS and UNIX applications and a terminal. UNIX serial I/O and a device driver that emulates it under DOS are examined, then techniques for writing applications that use RS-232 communications are discussed.



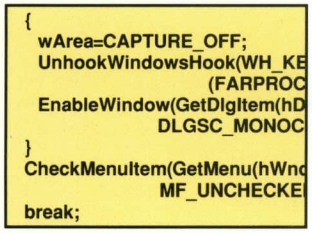
35 Simplifying Pointer Syntax for Clearer, More Accurate Programming

While pointers can simplify your C code, they can be difficult to use and often produce unwanted side effects. This article dissects some simple pointer examples, builds them into compact and efficient pieces of code, and examines the side effects they can produce.



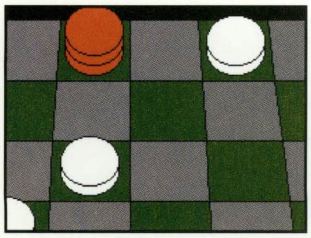
47 Integrating Subsystems and Interprocess Communication in an OS/2 Application

The final article in our series designed to introduce you to OS/2 programming presents an application that integrates the topics covered previously. It shows you how to implement an event-driven, message-based queue that you can use to create applications whose architecture is similar to that of Presentation Manager.



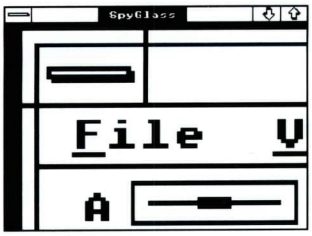
61 Exploring Dynamic-Link Libraries with a Simple Screen Capture Utility

Dynamic-link libraries form the backbone of the Windows™ environment. This article examines DLLs and keyboard hook functions as a means of capturing screen images and copying them to the Windows clipboard, and provides you with the complete source code for this useful utility.



69 Checkers Part I: Design Goals for Building a Complete Graphical Application

In this issue, *MSJ* presents the specification for a complex CHECKERS game for OS/2 Presentation Manager (PM). Charles Petzold will demonstrate PM programming techniques in the next few issues, covering such topics as graphics, child windows, dynamic-link libraries, and network-aware applications.



76

SpyGlass: A Utility for Fine Tuning the Pixels in a Graphics Application

This article presents a handy display-enlargement utility, which allows you to see easily how pixels are aligned on a high-resolution monitor in your Windows application. In so doing, the article demonstrates some graphic device interface (GDI) programming techniques.

EDITOR'S NOTE

Object-oriented programming (OOP) has become the favored programming seminar topic of the late 1980s. Many experts believe that it will be the programming paradigm of the 90s. OOP is a programming technique in which data, along with the code that operates on it, is encapsulated into a single entity known as an object. The object is designed to handle a set of operations that define what can be done with it. Objects have the ability to "inherit" the behavior of similar objects, which is one of the most important features of OOP. OOP is particularly valuable because it allows the dynamic collection of disparate objects in compound documents.

Smalltalk, the first object-oriented environment, was developed in 1967 at the Xerox Palo Alto Research Center. Since then, OOP has been the almost exclusive provence of academics and technical journals. The development of C++, a language combining object-oriented techniques with the popular C language, has helped to bring OOP into mainstream programming.

NewWave by Hewlett-Packard® is one of the first graphical environments to benefit from the renewed interest in OOP. In this issue, we explore NewWave from the development of applications to the use of scripts, taking a special look at its Object Management Facility.

Also in this issue, we examine the UNIX® RS-232 serial interface. This interface, independent from hardware, is very easy to work with; we explore a device driver that emulates it under DOS. And Charles Petzold begins a series, using a checkers program to demonstrate Presentation Manager programming techniques. You'll soon be able to play checkers under OS/2 Presentation Manager; at the same time, you'll become familiar with Graphics Programming Interface (GPI) techniques. —Ed.

Microsoft Systems Journal (ISSN# 0889-9932) is published bimonthly by Microsoft Corporation at 666 Third Avenue, New York, NY 10017. Single-copy price including first-class postage: $10.00. One-year subscription rates: U.S., $50. Canada/Mexico, $65. International rates available on request. Subscription inquiries and orders should be directed to the Circulation Department, Microsoft Systems Journal, P.O. Box 1903, Marion, OH 44305. Subscribers in the U.S. may call (800) 669-1002, all others (614) 382-3322 from 8:30 am to 4:30 pm, Mon—Fri. Second-class postage rates paid at New York, NY and additional mailing offices. POSTMASTER: Send address changes to Circulation Department, Microsoft Systems Journal, P.O. Box 1903, Marion, OH 44305.
MSJ is now available on microfilm and microfiche from University Microfilms Inc., 300 North Zeeb Road, Ann Arbor, MI 48106

Manuscript submissions and all other correspondence should be addressed to Microsoft Systems Journal, 16th Floor, 666 Third Avenue, New York, NY 10017.

Microsoft Systems Journal is a publication of Microsoft Corporation, 16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717. Officers: William H. Gates, III, Chairman of the Board and Chief Executive Officer; Jon Shirley, President and Chief Operating Officer; Francis J. Gaudette, Treasurer; William Neukom, Secretary.

Examining NewWave, Hewlett-Packard's Graphical Object-Oriented Environment

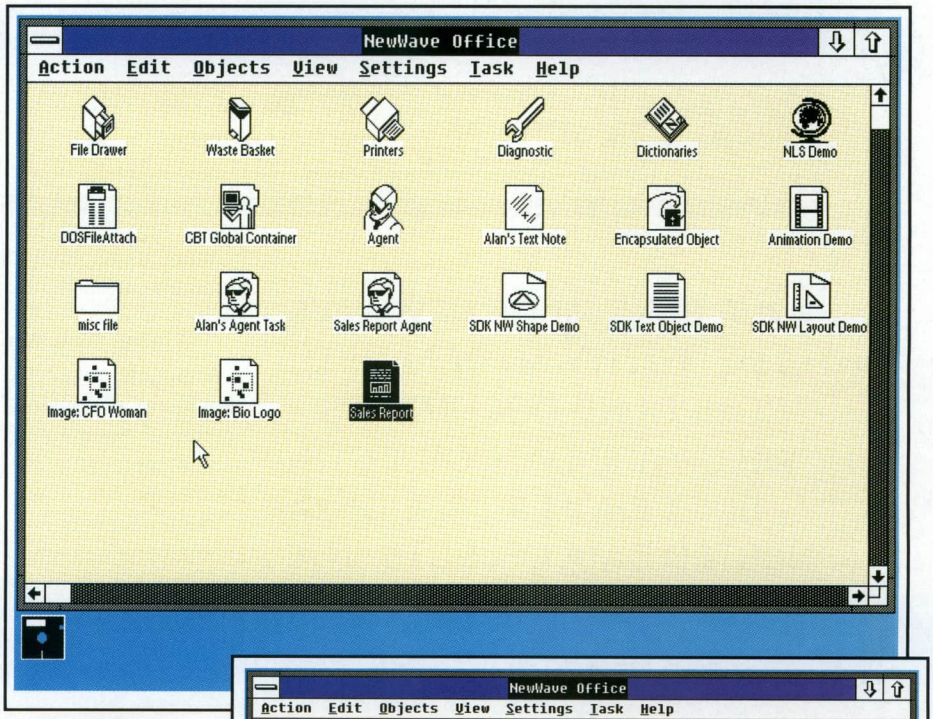
01

Alan Cobb and Jonathan Weiner

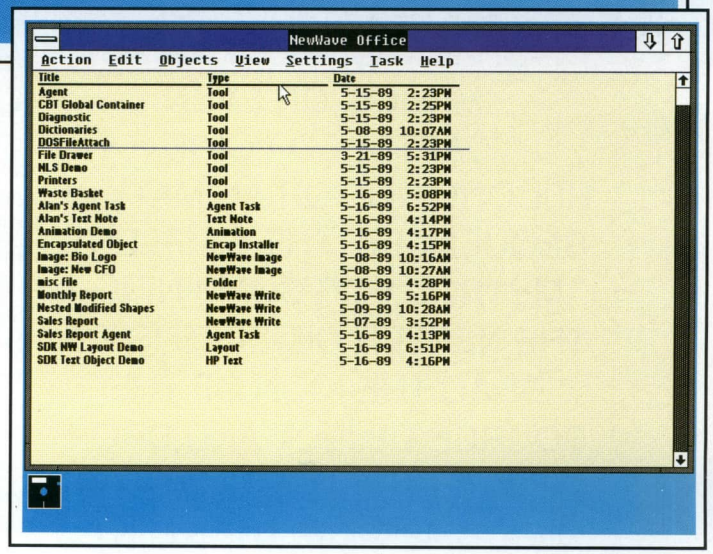
NewWave by Hewlett-Packard offers a wide range of advanced features for Microsoft® Windows™ Version 2.11 graphical environment-based applications. It is in effect an extra layer on top of the MS-DOS® operating system, enhancing and extending the services provided by MS-DOS[1] and Windows[2]. There are five areas in which NewWave surpasses Windows: control, communication, integration, abstraction, and ease of use. Because NewWave controls programs and provides new methods of communication between them, users can integrate several programs to do one task easily. Furthermore, NewWave enables the users to deal with the computer at a higher level of abstraction. That is, users can use simpler techniques to do broader, more complicated tasks—similar to the way programmers can do much more with one line of code in a high-level language than with one line of Assembler. Finally, NewWave is easy to use. Its advanced help and Computer-Based Training (CBT) systems make it simple for users to learn how to avail themselves of its services.

At the implementation level, NewWave is built upon multiple Windows programs and dynamic-link libraries (DLLs), as well as its data files. Windows developers can access these features by making calls to the NewWave functional interface and con-

Alan Cobb is a consultant and developer of Presentation Manager and Windows applications based in Redwood City, Ca. Jonathan Weiner developed various NewWave system components at Hewlett-Packard and is now the Technical Accounts Manager for NewWave ISVs.

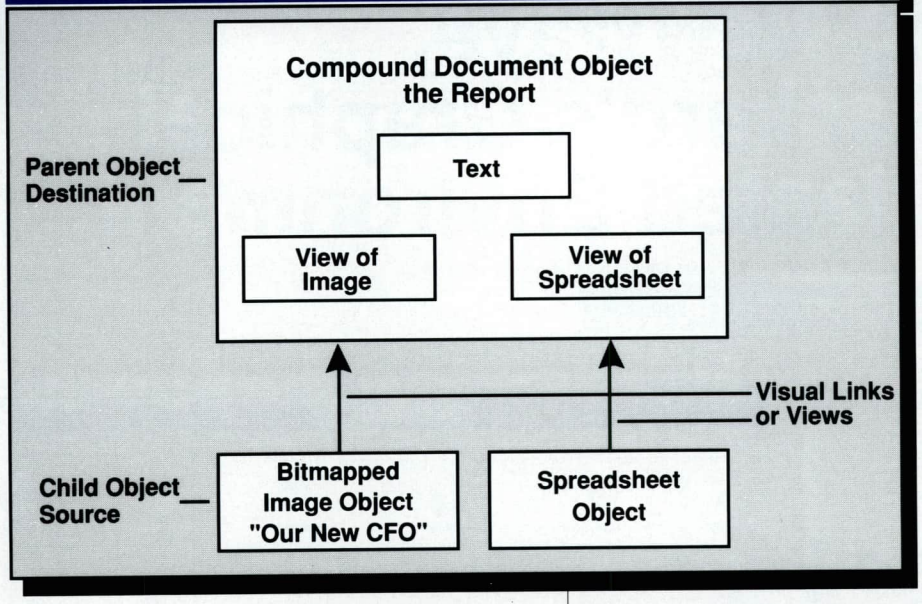


▲ **Figure 1**
The NewWave Office start-up screen.



▶ **Figure 2**
The start-up screen can also be shown in regular text view form.

Figure 3: Simple NewWave Document with Only Visual Links



ONE BENEFIT OF NEWWAVE IS ITS ABILITY TO DEAL WITH COMPOUND DOCUMENTS, WHICH ARE DATA OBJECTS BUILT FROM A TREE OF NESTED DATA OBJECTS. THE COMPONENT OBJECTS CAN BE TEXT, GRAPHS, SPREADSHEETS, TIFF IMAGES, OR EVEN CAPTURED VOICE RECORDINGS OR ANIMATION SEQUENCES. THE SIMPLEST METHOD FOR COMBINING THE NESTED OBJECTS IS TO DRAG THE ICON OF THE SOURCE OBJECT AND DROP IT INTO THE DESTINATION DOCUMENT AT THE POINT WHERE IT IS TO APPEAR.

forming to its interapplication communication protocols.

Using NewWave

Although under the surface NewWave is constructed from normal Windows programs, the user interacts with the program in quite a different way. Instead of beginning work in the MS-DOS Executive with its familiar directory listing, the first thing a NewWave user sees is the NewWave Office window, shown in **Figure 1**. Users can also access a conventional text listing of the icons, illustrated in **Figure 2**.

The Office can be resized like any other Windows program, but it usually covers the entire screen. On its surface is a set of icons representing familiar office tools such as the Printer, the Waste Basket, and the File Drawer; data objects such as the file folders, simple text documents and images; and compound documents built from smaller objects of text, graphs, and images. While the screen will show only one copy of a tool at a time, it can show multiple instances of data objects.

To work with a particular data object, just double click on its icon. NewWave has a record of which tool (EXE file) was used to create that object's data file. The appropriate program is automatically started and the data is read into it. Thus, the user is raised above the details of the file system. NewWave also has a one-step drag-and-drop technique used for operations like copying, deleting, and printing objects. For example, to delete a data object or folder of objects, the user just clicks on the object, drags the mouse to the Waste Basket icon, and drops it in.

Features

One of the principal benefits of NewWave is its superior ability to deal with compound documents. To get a better feel for NewWave's features, we will look at a simple demonstration system, consisting of a compound monthly report document that contains a nested bitmapped image and a nested spreadsheet. **Figure 3** shows the structure of this document.

Compound documents are data objects built from a tree of other nested data objects. The component objects can be pieces of text, graphs, spreadsheets, Tagged Image File Format (TIFF) images, or even captured voice recordings or animation sequences. In the example, the pieces are incorporated into a compound object and manipulated by a prototype NewWave-capable word processor called NewWave Write.

There are several methods that can be used to combine the nested objects into a larger compound document. The simplest is to drag the icon of the nested object (the source object) and drop it into the larger destination document at the point where it is to appear. This method moves the document completely inside the destination, so the separate source icon

will no longer be shown in the Office window. **Figure 4** shows the sample object on the screen after it has been constructed.

But what if you also want to show the source object, say the spreadsheet, in another compound document at the same time? In that case, you can simultaneously share the single source with multiple destination objects. To do this, highlight the source icon with a mouse click, then select Share from the Edit menu. A reference to the source object is now on the clipboard. Now you can go into one or more destination documents and use the clipboard's familiar Paste command to insert a nested view of the source document. After the pasting, the separate icon for the source will still appear in the NewWave Office window.

The full power of NewWave is demonstrated when you need to modify the compound document or one of its components. For example, suppose you decide to add another column to the nested spreadsheet. To access the spreadsheet, simply double click on the area where it is nested. Since the NewWave database of links or views knows which tool and data files were used to create the nested spreadsheet, it can automatically start the tool application and load the spreadsheet for modification. So you can see what is happening, NewWave explodes the nested window and grays the area where it is normally nested. The exploded window will be the main window of the application that was used to create the source object. In our spreadsheet example, the application might be Microsoft Excel. **Figure 5** shows the nested TIFF picture being modified.

When a conventional program needs to support a new data type, new code to handle that type must be added to the application. Another feature of

A NewWave Glossary

NewWave has its own definitions for a number of terms. Most of them are illustrated in **Figures 3, 4** and **5**.

NewWave Terms:

Objects All the icons in the NewWave Office window represent either tools or data objects.

Office Tools or System Objects Tool objects are tools such as the Waste Basket, File Drawer, and Printer. They are associated with some type of fixed system service. They cannot be copied or deleted by the user.

User Objects User objects are objects that can be freely copied, deleted, cut, pasted, or shared by the user.

Data Objects Data objects are the combination of an application that creates and manipulates data and a specific data file that was created with it. For example, a word processor and a memo created with it would form one object. Under NewWave the user doesn't work with applications but rather with these bundled pairs of application and data.

Compound Objects Compound objects are built from a combination of smaller objects. For example, a spreadsheet object could have several small text note objects attached to it to explain some of the calculations. The aggregate group can be copied, printed, and moved as a single entity via the clipboard.

Container Objects Containers are objects like the File Drawer, File Folders, and Waste Basket, that are used to hold objects in a group. They are represented by a single icon in the Office window. Containers can be opened to show the objects they hold.

Views or Links Objects can be connected to one another in several different ways called links or views.

Visual Views or Links In a visual link, one object projects a view of itself into another. The projected object is shown nested inside the destination object. For example, a graph object could be connected with a visual view to a text report in which it appears. The nested object does all the drawing of its own view.

Data Passing Links Data links or views actually pass pieces of binary data between objects. For example, a communications program could pass stock prices to a spreadsheet object for analysis.

Simple Links Container objects are connected by simple links to the objects they enclose. There is no passing of data or visual views between them.

Source and Destination Objects The source object is the one either sending binary data via a data link or else projecting a view of itself via a visual view into a destination object.

Child and Parent Objects A child object is either a source object or an object held within a container. A parent object is either the destination of a visual or data link or else a container holding other objects. One object can be simultaneously both a parent to objects below it and a child to objects above it.

Other Terms:

API Although API (Application Program Interface) is normally a generic term that refers to any functional interface to a subsystem, NewWave uses it to refer specifically to the interface to its Agent, CBT, and Help systems.

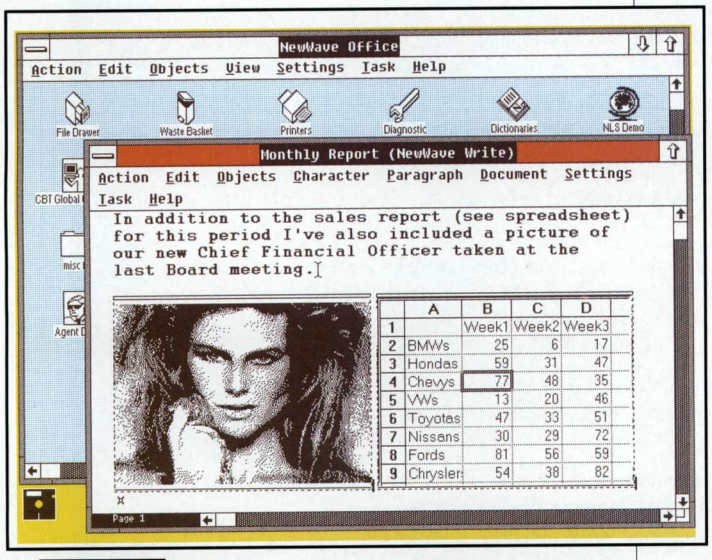
Methods Objects communicate among themselves and with NewWave by sending messages to each other. For example, the message DISPLAY VIEW is sent from a parent object to a child during the setup of a visual link. The code in the child that processes a particular message is referred to as a method.

Agent An Agent is like a batch or macro file for controlling data objects, tools and other programs in the NewWave environment. The Agent's commands are written in an Agent task script language. Individual scripts are shown in the NewWave Office as icons.

Share Although one child object can be moved or pasted completely into a single parent object, it can also be shared into the same object.A child can be shared into multiple objects at once. For example, one chart could appear in several reports at the same time. When the child chart is updated, it would be shown in its updated form in all the parent report objects.

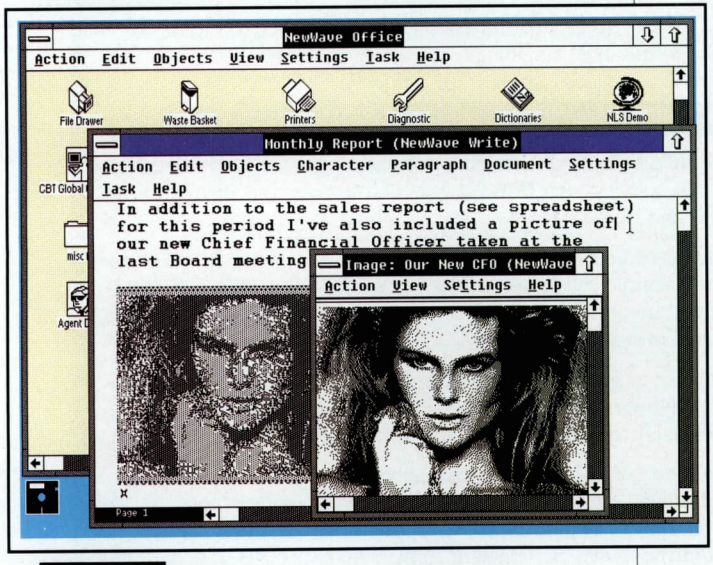
04

NEWWAVE AVOIDS THE NEED TO DUPLICATE CODE FOR NEW DATA TYPES IN APPLICATIONS BECAUSE IT LOGICALLY JOINS ALL APPLICATION DATA FILES TO THE PROGRAMS THAT CREATED AND EDITED THEM.



	A	B	C	D
1		Week1	Week2	Week3
2	BMWs	25	6	17
3	Hondas	59	31	47
4	Chevys	77	48	35
5	VWs	13	20	46
6	Toyotas	47	33	51
7	Nissans	30	29	72
8	Fords	81	56	59
9	Chryslers	54	38	82

▲ **Figure 4** Example of a compound document.



▲ **Figure 5** Editing the nested TIFF image.

NewWave is that it eliminates this duplication of code by logically joining all application data files to the programs that created and edited them. When it is time to view or manipulate the source data, the destination object can simply ask the source object to do it. This is the principal sense in which NewWave is object-oriented. In fact, a NewWave object is defined as this paired combination of data and the application required to manipulate it. NewWave is also object-oriented in that objects pass command messages back and forth to each other. The standard term "method" is used to describe the code used by an object to process one of its messages.

What if you start modifying the visually linked image separately while the larger compound document is closed? NewWave's database of interobject links—part of the Object Management Facility (OMF) discussed below—takes care of this also. When the larger report is later opened or printed, it will see a flag in the database telling it that the graph has changed. The report object can then ask the graph to rerender the projected view of itself.

NewWave allows you to paste the report into an even larger compound document. Nested groups of compound objects can be moved, copied, printed, erased, or mailed all at once, simply by selecting the overall object with the mouse and dragging it to the destination or by copying and pasting it through the clipboard.

Advanced Features

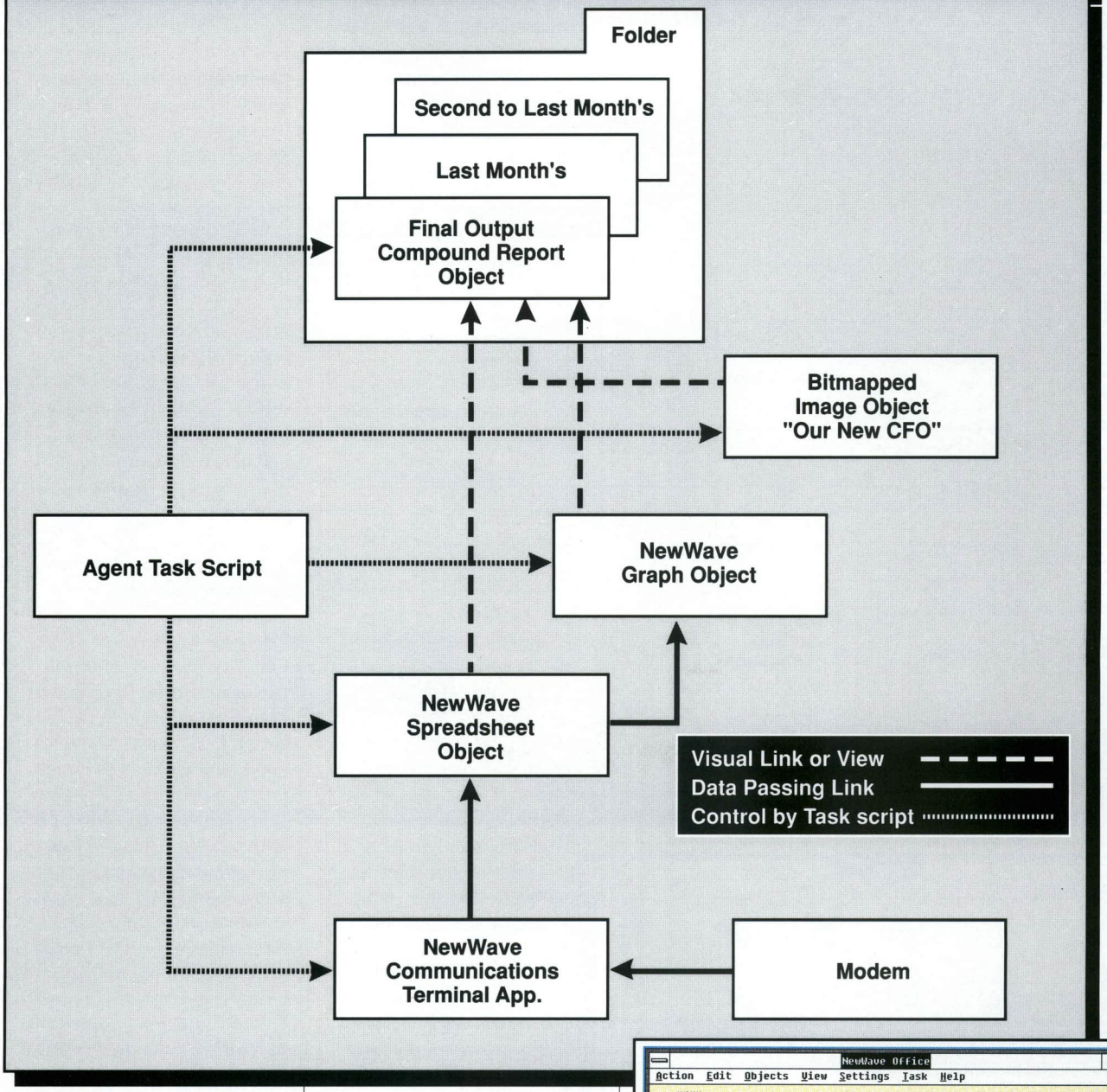
Now we will extend the sample program to include advanced NewWave features. In its present form the sample monthly report only uses NewWave's visual links (also called visual views). That is, the source objects are only connected to the larger destination visually; there is no actual passing of data between the applications. To pass binary data, such as an array of integers from a spreadsheet, NewWave uses its second main type of link, the data passing link.

Figure 6 shows how data passing links can be added to the sample program. We added two new hypothetical applications, a NewWave-capable terminal program and a NewWave-capable graphing program. A NewWave Agent task script, which is like an advanced batch file or macro for Windows programs, has also been added. The script is used to make all the applications work as a team to produce the monthly report.

To produce the report, the controlling Agent script first sends the appropriate commands to the NewWave-capable terminal, causing it to dial all of the company's regional offices. The Agent script automatically collects the monthly sales data from each office. Details of the quantities of products sold are passed via a NewWave data passing link to the spreadsheet. As in the first part of the example, the spreadsheet uses a visual link to project the spreadsheet grid into the report, but now it also uses a data passing link to pass the summarized spreadsheet data to a NewWave-capable graphing program. The graphing program then uses a visual link to display itself inside the report.

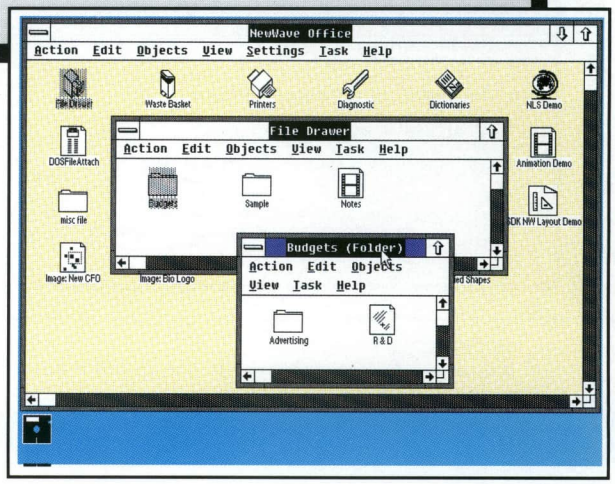
The abstraction of file folder icons to represent directories

Figure 6: NewWave Application System with Data Passing Links



and the files they contain is another advanced feature of NewWave. To illustrate this feature, the sample also includes the monthly report in a folder of all monthly reports for the year. The folders can be nested inside each other as well as inside the File Drawer icon. Folders are opened with a simple double click. **Figure 7** shows the open File Drawer; the Budgets folder within has also been opened.

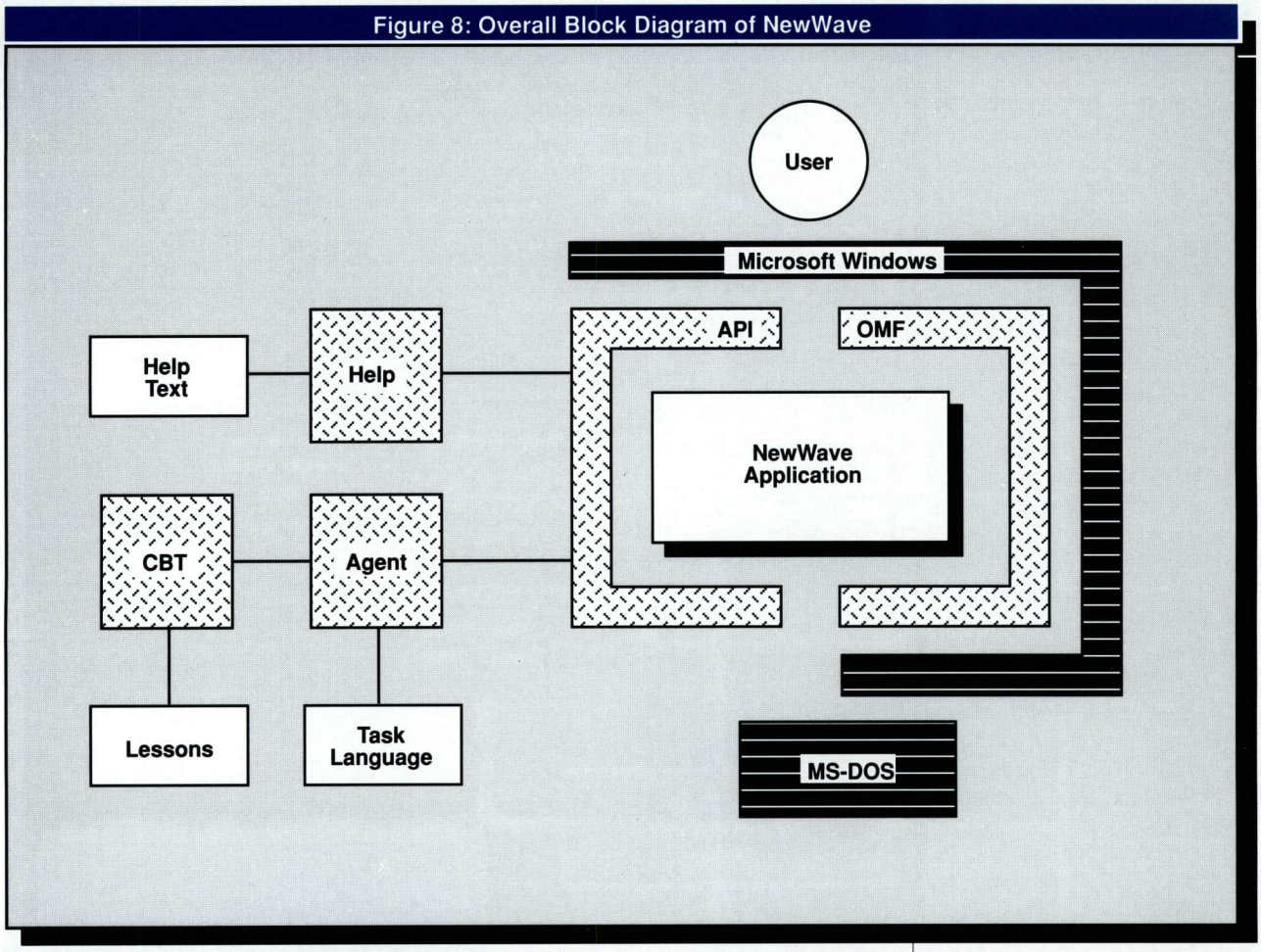
The connection between filed objects is accomplished with NewWave's third type of link, the simple link. Instead of passing views or data these links simply enclose one object inside a parent container object. Other objects can also be inserted into

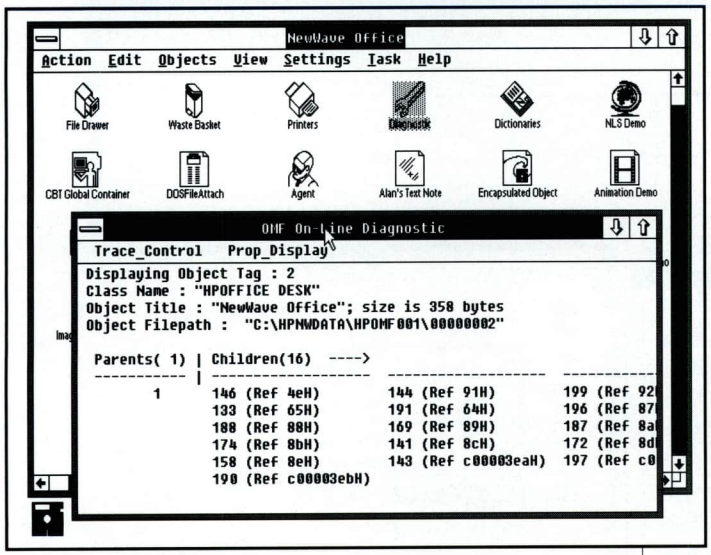


▶ **Figure 7** **The File Drawer and budgets folders have been opened.**

06

Figure 8: Overall Block Diagram of NewWave





▲ **Figure 9** **This diagnostic utility allows the user to traverse the object tree.**

or removed from a container object, such as the Waste Basket, with the drag-and-drop method.

OMF and API Systems

Now that you are familiar with the features of NewWave, we will describe its major components (some of which were referred to earlier). NewWave consists of two main systems, the OMF and the API (see **Figure 8**). The API in turn is composed of three smaller systems: the Agent task script system, the Help system, and the CBT. The OMF is used to record and supervise the visual and data links between objects. The API Agent script system handles the recording, playback, and editing of the task scripts used to control NewWave applications. The API, Help, and CBT systems provide a high-quality, prewritten, standard foundation for adding help, demos, and training to NewWave applications.

All the objects in a NewWave system are connected in a tree. Near the top of the tree is the NewWave Office. It is a parent to every object that appears in the Office window. For example, the File Drawer is a child of the NewWave Office, and it in turn has its own children in the form of folders. **Figure 9** shows one of the diagnostic utilities that comes with the system; this utility allows you to traverse the object tree. A child object is

enclosed in a container or acts as the source in a visual or data link. One child can have multiple parents (a graph object, for example, can have visual views projected into three parent reports at the same time), which makes the structure more than a simple tree.

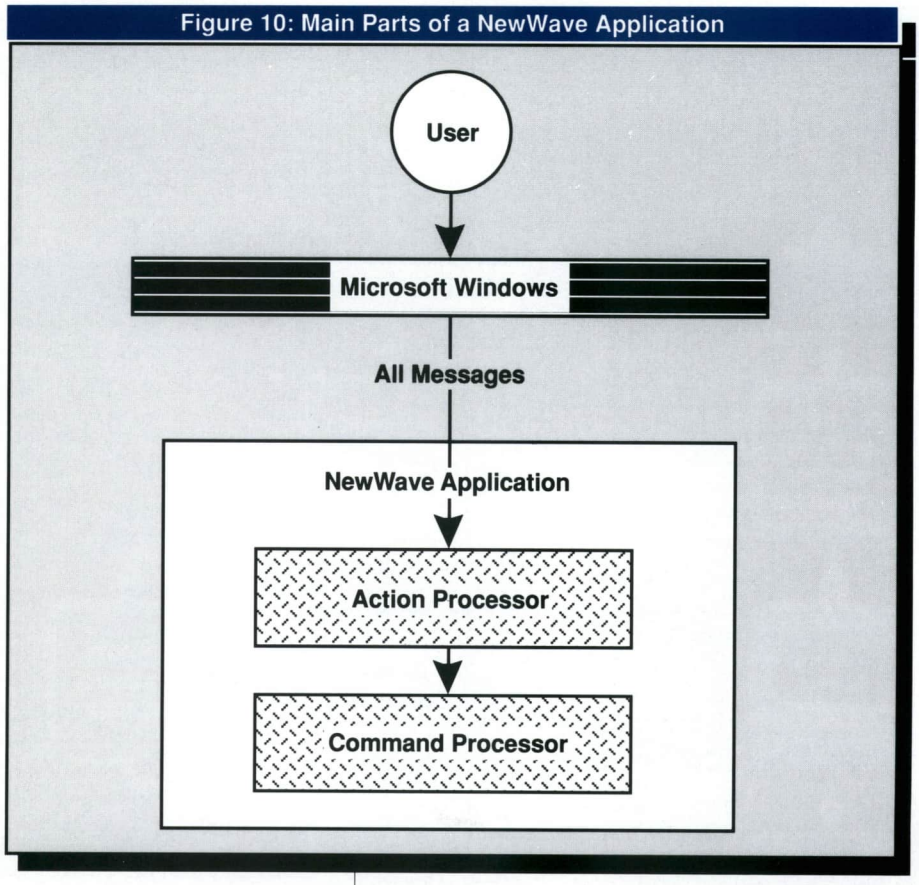
Processors

Every NewWave application contains two systems, the Action processor and the Command processor, as shown in **Figure 10.** The Action processor translates user actions, such as mouse clicks and menu selections, into one of the commands that the NewWave application can perform. Often, several different actions will be translated into the same command. For example, using an accelerator key sequence (such as Alt-F-x) can execute the same command as selecting a menu item with the mouse (such as clicking on File, then Exit).

The Command processor provides all the actual functionality of the application. It executes commands that are passed to it from several possible sources. For example, commands can come directly from the Action processor as a result of current user keystrokes and mouse movements or be played back from an Agent script. The Command processor must be able to handle the full range of verbs in the application's command language. Agent scripts are built from this set of commands. This is discussed in detail below.

Splitting the application's control system into Action and Command processors makes it much easier for NewWave to implement the Agent macro facility. That is, when NewWave records an Agent script, it need not concern itself with the details of the user's actions. It only has to store the series of commands that result from the Action processor's translation.

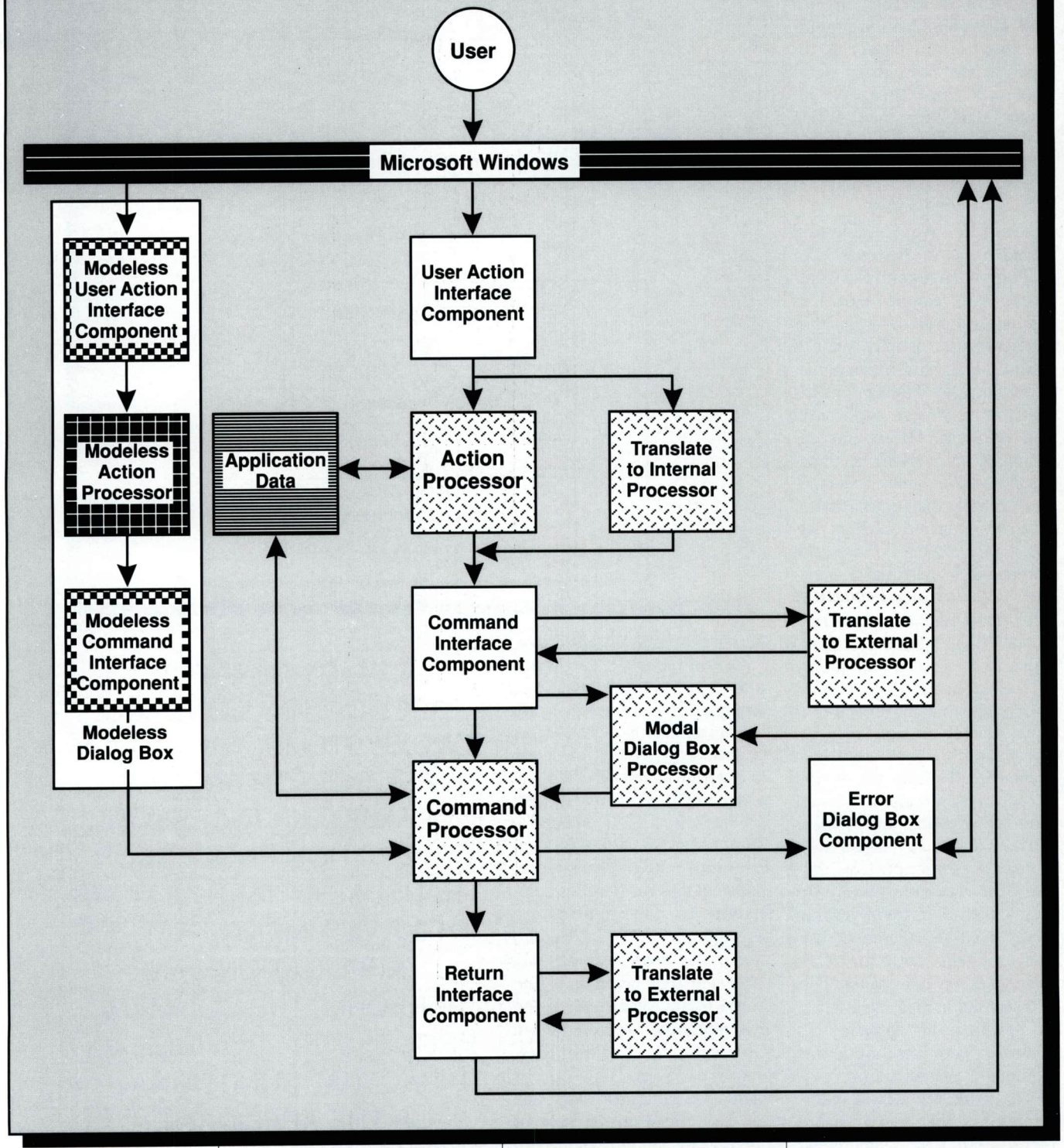
Figure 10: Main Parts of a NewWave Application



At a lower level, the Action and Command processors are supported by several other processors and components. **Figure 11** shows the system in detail. This design enables the application to run in one of five modes: Playback, Record, Intercept (for Help), Monitor (for CBT), or Error mode. In Playback mode, an Agent task script sends commands to the application through the API. In Record mode, the user's actions are translated into commands and stored in an Agent task script. Intercept mode is entered when the user asks for context-sensitive help. The cursor changes to a question mark, and the user clicks on the item of interest. In Monitor mode, all commands are passed to the CBT system before they are executed. This allows the CBT system to discard inappropriate commands during a lesson and

EVERY NEWWAVE APPLICATION CONTAINS TWO SYSTEMS, THE ACTION PROCESSOR AND THE COMMAND PROCESSOR. THE ACTION PROCESSOR TRANSLATES USER ACTIONS INTO A COMMAND THAT THE APPLICATION CAN PERFORM. THE COMMAND PROCESSOR PROVIDES ALL THE ACTUAL FUNCTIONALITY OF THE APPLICATION, EXECUTING COMMANDS THAT ARE PASSED TO IT.

Figure 11: Detail of Message Flow in a NewWave Application



guide the user in the right direction. During Error mode, any error notifications are rerouted to the controlling Agent instead of to the normal destination, which is an error message box for the user.

Although the system is complex, the hard part of initial design and debugging has been done for you. The NewWave Software Development Kit (SDK) gives detailed examples of how to build the pieces. The components (as opposed to pro-

Need a good reason for getting your own subscription to *Microsoft Systems Journal*?

Just look at what you've been missing:

Designing *a virtual memory manager*

Using *Presentation Manager's Dynamic Data Exchange Application Program Interface*

Writing *TSRs in C*

Developing *and debugging embedded systems applications*

OS/2 *kernel programming*

BASIC *as a professional programming language*

and *much much more!*

Need additional coaxing? Subscribe for 1 year and save 30% off the regular price. Subscribe for 2 years and save 40% (and protect yourself from price increases).

Just fill out a card, drop it in the mail, and the issues are on their way.

Microsoft Systems Journal

Introductory Savings Coupon

for new subscribers only.

Microsoft SYSTEMS JOURNAL

YES! *I want to subscribe to Microsoft Systems Journal. Enter my order for the rate and term I've checked below.*

○ 1 year $34.95 (Save 30%) ○ 2 years $59.95 (Save 40%)

❑ Payment enclosed ❑ I'll pay when invoiced

Name _____

Address _____

City _____ State _____ ZIP _____

Telephone _____ ❑ Business ❑ Home

Guarantee: If you are ever dissatisfied with MSJ, you're entitled to a full refund on the unmailed portion of your subscription.

Note: Offer limited to new subscribers only. Regular price is $50 per year. Foreign subscribers add $15 per year postage. Payment in U.S. funds. Please allow 6 to 8 weeks for delivery of first issue. Offer expires December 31, 1990.

EN9AB1-X

Microsoft SYSTEMS JOURNAL

YES! *I want to subscribe to Microsoft Systems Journal. Enter my order for the rate and term I've checked below.*

○ 1 year $34.95 (Save 30%) ○ 2 years $59.95 (Save 40%)

❑ Payment enclosed ❑ I'll pay when invoiced

Name _____

Address _____

City _____ State _____ ZIP _____

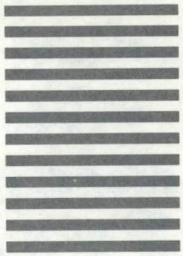
Telephone _____ ❑ Business ❑ Home

Guarantee: If you are ever dissatisfied with MSJ, you're entitled to a full refund on the unmailed portion of your subscription.

Note: Offer limited to new subscribers only. Regular price is $50 per year. Foreign subscribers add $15 per year postage. Payment in U.S. funds. Please allow 6 to 8 weeks for delivery of first issue. Offer expires December 31, 1990.

EN9AB1-X

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

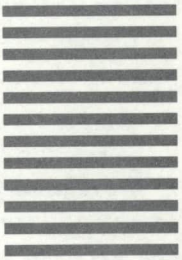


BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 603 MARION, OH USA

POSTAGE WILL BE PAID BY ADDRESSEE

MICROSOFT SYSTEMS JOURNAL
P.O. BOX 1903
MARION OHIO 43306

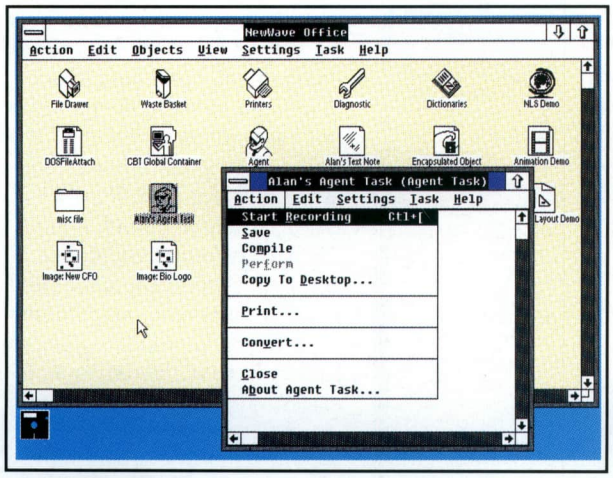
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



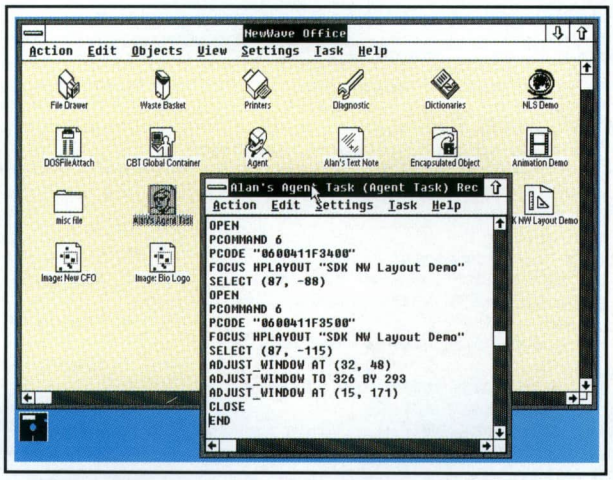
BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 603 MARION, OH USA

POSTAGE WILL BE PAID BY ADDRESSEE

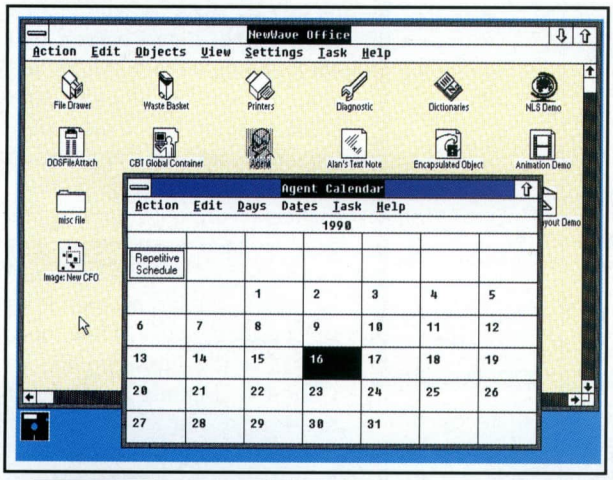
MICROSOFT SYSTEMS JOURNAL
P.O. BOX 1903
MARION OHIO 43306



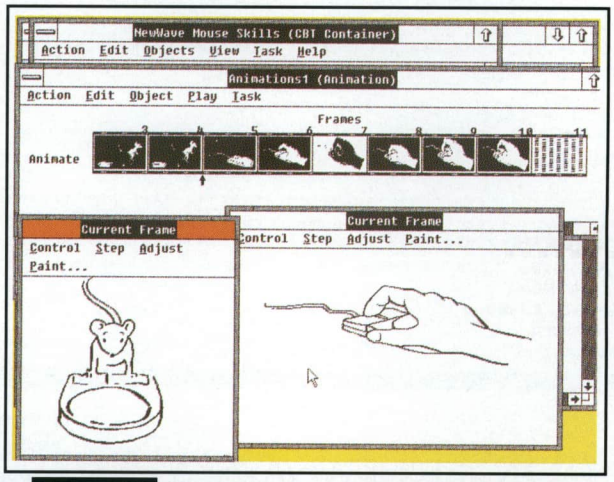
▲ **Figure 12** Create Agents task scripts by recording user commands.



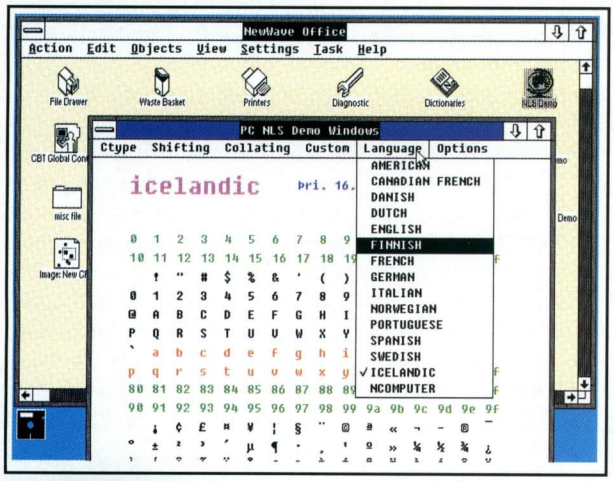
▲ **Figure 13** Viewing an Agent script.



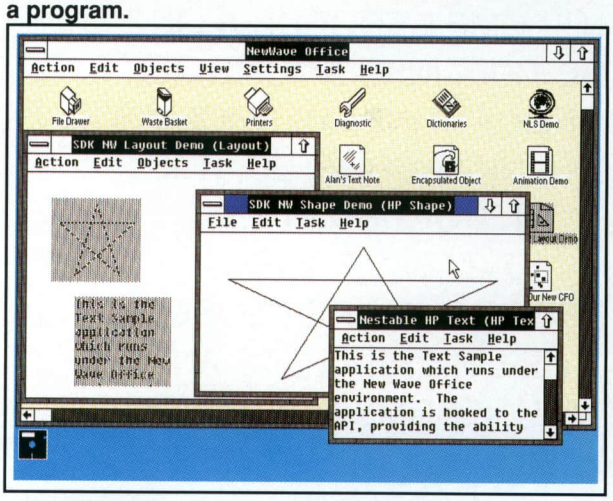
▲ **Figure 14** Agents can perform tasks at specified times (here, the sixteenth of the month.)



▲ **Figure 16** The animation system lets developers create documentation and help files while designing a program.



▲ **Figure 17** You can customize NewWave for different national languages without changing code.



▲ **Figure 18** NewWave SDK demo application.

Figure 15: Sample NewWave Task Script

```
'*****************************************************************
'* Simple NewWave task script example. Copies a bitmapped image  *
'* object called "Image: Our New CFO" to the clipboard, then     *
'* pastes this into a compound document called "Demo Report."    *
'* It repeats this append operation in a loop until the user     *
'* decides to exit. Author:  Alan Cobb                           *
'*****************************************************************

TASK
bContinue# = 1
WHILE bContinue# = 1
  FOCUS OFFICE "NewWave Office"              ' Set keyboard focus.
 SELECT NEWWAVE_WRITE "Demo Report"
  OPEN                                       ' Open report.
  SELECT NWImage "Image: Our New CFO"
  COPY                                       ' Copy to clipboard.
 FOCUS NEWWAVE_WRITE "Demo Report"
  DO PASTE_INTO_REPORT                       ' Paste image.
  DO CLOSE_REPORT                            ' Save result.
 MESSAGE bContinue# "Repeat process?" yesno ' Loop again?
ENDWHILE
MESSAGE bTemp# "Exiting." ok                 ' Pause before exit.
END
ENDTASK

'*****************************************************************
'*  Equivalent of PASTE command in pcode form.
'*****************************************************************
PROCEDURE PASTE_INTO_REPORT
  PCOMMAND 4
  PCODE "0400CD00"
  RETURN
ENDPROC

'*****************************************************************
'*  Equivalent of CLOSE command in pcode form.
'*****************************************************************
PROCEDURE CLOSE_REPORT
  PCOMMAND 8
  PCODE "0800020001000100"
  RETURN
ENDPROC
```

UNDER NEWWAVE, A SYSTEMWIDE DATABASE IN THE OMF RECORDS A LIST OF THE LINKS AMONG ALL APPLICATIONS. AS SOON AS A LINK IS CREATED, IT IS RECORDED IN THE DATABASE. WHEN PROGRAMS ARE CLOSED, THE LINK PERSISTS AND WILL RESUME OPERATION WHEN THE PROGRAMS ARE RESTARTED.

cessors) are supplied by Hewlett-Packard. The user need only change the variable names in these components to match his or her system. As shown in the figure, any modeless dialog boxes need to be supported with their own Action processor.

The Translate to Internal and Translate to External processors convert commands between Internal and External format. Commands are stored in External binary form by Agent tasks. This is the form passed to the application for execution. The application translates this to its own private Internal format, which can take whatever form is necessary.

Compared to Windows

NewWave has several advantages over Windows. One is the superiority of the forms of links NewWave has to the methods of interprocess communication in Windows. In Windows there are no standard equivalents of the visual views and simple container links found in NewWave. Nor are there Windows equivalents to the share capability or ability to move, print, copy, and paste compound documents NewWave possesses. These features make it significantly easier to organize and maintain complex objects in NewWave than in Windows. The Windows Dynamic Data Exchange (DDE) protocol does, however, provide some of the same features as the NewWave data passing links. DDE moves data between two applications via messages and shared memory blocks. Both DDE and NewWave data passing links allow one application to update a linked program automatically without requiring the user to take any action. For example, one conventional Windows program can read stock price data from Dow Jones and pass the latest prices via DDE to a graphing program for real-time display.

The primary advantage that NewWave data passing links have over DDE is that they provide several fairly complex services that DDE users would have to rewrite and debug from scratch. Not only would a DDE application need considerable added code, that code would have to be duplicated the same way in all the other programs with which the DDE application was going to communicate. NewWave moves much of that common code out of individual applications and into one centralized, standardized, operating-system-style service.

Another way in which the NewWave link services go beyond DDE is that they are persistent. When you shut down two programs using DDE, the

NewWave Command and Function Summary

The Class Independent Commands are executed by the Agent itself at run time, independent of any application object that may be open. Most of them either manipulate task conversational windows or handle flow control of the Agent task.

Command Name	Description
CLEARWINDOW	Clear a user conversational window
CLOSEWINDOW	Close a conversational window
DEFINEWINDOW	Define a user conversational window
DO	Execute a procedure
EDITBOX	Create an edit box in a conversational window
END	Terminate execution of a task
FOCUS	Change focus to a specified object
GOTO	Transfer control to a labeled statement
IF ELSE ENDIF	Conditional execution
INPUT	Create a window to prompt the user for input
JUSTIFY	Justify text in a conversational window
LABEL	Define a label
LOCATE	Position the cursor in a conversational window
MESSAGE	Create a message window (OK, RETRY, and so on)
ON ERROR DO	Trap on any error condition
ON ESCAPE DO	Trap on the escape key
ON TIMEOUT DO	Trap on a timeout
OPENWINDOW	Open a conversational window
OUTPUT	Output text to a conversational window
PAUSE	Halts execution temporarily
PROCEDURE ENDPROC	Define a procedure
PUSHBUTTON	Draw a pushbutton in a conversational window
RETURN	Exit a procedure
SCREEN	Map logical screen and window coordinates
SET ERROR	Set error trapping on or off
SET ESCAPE	Set escape trapping
SET RATE	Sets rate at which Agent executes commands
SET TIMEOUT	Set timeout trapping
TASK ENDTASK	Defines the main body of a task script
TITLEWINDOW	Set caption bar text of conversational window
WAIT	Suspend execution until an event is trapped
WHILE ENDWHILE	Looping of control

Class Independent Functions for Data Manipulation:

Function Name	Value Returned
EDITBOX	String of text in an edit box
TESTBUTTON	Tells if a given button was pushed
ABS	Absolute value of a numeric argument
ASC	Integer for a given string
CHR	String for a given integer
FIND	Location of substring in a string
INT	A long when passed a real
LEFT	String of leftmost characters
LEN	Number of characters in a string
MID	Extracted substring
MOD	Remainder for two integers
RIGHT	String of rightmost characters
STR	ASCII string given a numeric argument
SYS_ERROR	Error number of last run-time error
NUM	Tells if parameter is numeric
VAL	Numeric value of input string

Class Dependent Commands for the NewWave Office Window

Individual applications have their own unique set of commands that they will accept. These are called Class Dependent Commands. Below is a subset of the Class Dependent Commands for the NewWave Office.

Command Name	Description
ABOUT?	Display "About ..." dialog box
ACTIVATE	Change the currently active window
ADD_SELECTION	Select the specified object
ADJUST_WINDOW	Move or size the current window
ALIGN_BY_ROWS	Align the icons in the Office window
AUTO_ALIGNMENT	Set auto_alignment (snap to grid) mode
CHANGE_ATTRIBUTES	Change attributes of a selected object
CHANGE_TITLE	Change an object's title
CLOSE	Close the currently active window
CONTROL_PANEL	Execute the Windows Control EXE program
COPY	Copy an object to the clipboard
COPY_TO	Copy an object to a closed container
CREATE_A_NEW	Create a new object
CUT	Delete selected objects to the clipboard
EXPORT_TO_DISK_FILE	Serialize an object's data to a disk file
ICONIC_VIEW	Display objects in HP Office as icons
IMPORT_FROM_DISK_FILE	Deserialize an object from a disk file
LIST_VIEW	Display a container's objects as a list
LOCK_DISPLAY	Display dialog box to record password
MAKE_COPY	Copy all selected objects in the window
MANAGE_MASTERS	Remove objects from the workspace
MANAGE_TOOLS	Change the selection of tools in the Office
MAXIMIZE	Increase the current window's size
MOVE_TO	Move selected objects to a container
OPEN	Open all selected objects
OPEN_SELECTED_OBJECT	Open an object
PASTE	Paste objects from the clipboard
PERFORM	Perform an Agent task
PRINT	Drop selected object on printer icon
PRINT_LIST_OF_OBJECTS	Make a hard copy of container contents
RESTORE	Return a window to its default size
SAVE_AS_MASTER	Save a copy of the object as a template
SELECT	Select one object by class and title
SELECT_ALL_OBJECTS	Select all objects in the current window
DESELECT_ALL	Deselect all the objects in the window
DESELECT	Deselect one object by class and title
SELECT_OPENED	Select an open object by class and title
SEND_TO_MAILROOM	Insert selected objects into the mail room
SET_PASSWORD?	Display the "Password" dialog box
SET_USER_TIME_ZONE	Modify user name and time zone data
SHARE	Share selected objects to the clipboard
SHOW_DOS_PATH	Display path of an MS-DOS application
SHOW_LINKS	Show the links to an object
SHOW_OWN_LINKS	Show the links to an object's parents
OPEN_PARENT	Open a linked parent
STRAIGHTEN_UP	Snap objects to the nearest grid point
THROW_AWAY	Move selected objects to the Waste Basket
TRANSFER_MAIL	Initiate a send or receive mail transfer

Class Dependent Commands for the HPSHAPE Sample Application

This is the unique set of commands for the HPSHAPE sample program discussed in the text. The commands correspond to the items on HPSHAPE's menus. Notice that there is some overlap with the Office window's command set.

Command Name	Description
CLEAR	Clear the object's window
CLOSE	Close the currently active window
MAXIMIZE	Increase the current window's size
RESTORE	Return a window to its default size
ELLIPSE	Display an ellipse
RECTANGLE	Display a rectangle
STAR	Display a star
TRIANGLE	Display a triangle

Figure 19: Calltree for SHAPES.C Before NewWave Enhancements

```
WinMain                         < Initialization and main event loop.
|   ShapesInit                  < Initialize main window.

ShapesWndProc                   < Handle all messages sent to Shapes.
|   ShapesCommand               < Process menu selections.
|   ShapesPaint                 < Paint the requested shape.
|     |  ShapesSetupDC
|     |   DrawRect
|     |   DrawEllipse
|     |   DrawTriangle
|     |   DrawStar
```

Figure 20: Calltree for HPSHAPE.C After NewWave Enhancements

```
WinMain                         < Initialization and main event loop.
|   ShapeInit                   < Initializes HPSHAPE'S window
|  ShapeCreateWindow            < Creates a NewWave main window.

ShapeWndProc                    < Handle all messages sent to HPSHAPE.
| TranslateToInternalProcessor
|   |                           < Take the external command from the buffer.
|   |                           <  Change external command into the internal
|   |                           <  command format.
|  TranslateToExternalProcessor
|   |                           < Convert the internal command format
|   |                           <  into the external command format.

|  ActionProcessor              < Handle all of the user actions. Prepare
|   |  |                        <  them for Command Processor
|   |  InterrogateFromAPI
|                               < Handle context-sensitive help requests.
|   |  MessageFromOMF           < Process messages sent from the OMF.
|   |    |  ReadDataFile        < Get the data file for HPSHAPE.
|   |    |  HasMethod           < Reply to the HAS_METHOD message to
|                               <  indicate whether a method is supported by
|   |    |   |                  <  HPSHAPE.
|   |    |   InitView           < Establish a view definition for
|   |    |   |                  <  the data to be displayed in the
|   |    |   |                  <  requesting object.
|   |    |   GetSize            < State the size HPSHAPE needs to be in
|   |    |    |                 <  order to display itself as a view.
|   |    |   DisplayView        < Paint the view in the DC specified
|   |    |    |                 <  by the destination object.
|   |    |   CopySelf           < Allow the object to be copied when
|   |    |   |    |             <  contained in another object.
|   |    |   |  SaveDataFile
|   |    |   |    |             < Save the data file that is the last
|   |    |   |  |               <  shape selected.
|   |    |  SaveDataFile...
|   |    |  SaveWindowPosition
|                               < Saves the last window coordinates in a
|   |    |  |  |                <  property.
|   |    |  ShapePaint          < Paint the requested shape.

|  CommandProcessor             < Handle any commands passed from the
|   |  |                        <  ActionProcessor or API (for example, Agent).
|   |  SaveWindowPosition...
```

link will be broken. Under NewWave, a systemwide database in the OMF records a list of the links among all applications. As soon as a link is created, it is recorded in the database. When the NewWave programs are closed, the link persists and will resume operation when the programs are restarted. (DDE applications, on the other hand, would have to reestablish the link each time.)

For example, suppose you have linked some numbers from a spreadsheet to a report you are producing with an editor. The editor doesn't have to be on constantly, waiting to get possible changes from the spreadsheet. If the spreadsheet does change, NewWave will automatically set a flag in its link database, indicating that the report needs to be updated when it is reopened. When the editor starts again, NewWave can automatically restart the spreadsheet in the background to rerender the linked data.

If the editor were using conventional DDE, it would have to implement its own list of the server applications to which it was linked as well as the type of data passed over each of the links. NewWave removes this burden from individual applications by providing it as an environmental service.

Another interprocess communication service unique to NewWave is the snapshot. A snapshot is a special type of object used to reduce the time and memory overhead required for communication views between two objects. Usually when a destination object requests the rendering of a fresh view or more data from a source object, the entire source object must be loaded and run. This could mean loading an entire spreadsheet application and a large worksheet just to access three numbers somewhere in the worksheet. NewWave enables you to provide a small snapshot object that can render only a particular linked view. If a snapshot is present when a destination object requests the rerendering of a view, NewWave will reroute the request, instead of the full application, to the destination object.

A snapshot is implemented as a small DLL with an associated data file. It loads faster and requires less memory because it is a DLL and not a full process. A snapshot doesn't have to contain the user interface or any other code beyond what is required to render that one view. Conventional DDE, on the other hand, would require both complete applications to be present in memory in order to pass the new data.

Another way in which NewWave extends Windows is in its task script language, called Agents. Agents are comparable to extended BAT files that control Windows programs. Windows Version 2.1 currently has no built-in control script facility of this kind. Although under Windows it is possible to write journaling programs that record and play back series of keystrokes and mouse movements, the NewWave Agents function at a cleaner and more fundamental level. Unlike journaled keystrokes and mouse movements (which are just recorded user actions rather than commands), Agents will work regardless of how many applications are present in the Office and where they are positioned.

The task language has many standard statements, including opening, closing, minimizing, and maximizing applications, that correspond to commands shared by most NewWave programs. The most interesting feature of the Agent language, however, is its extensibility to support individual applications. Each NewWave program defines its own command language and implements a parser to translate it. A full NewWave application must define statements in its command language to support all its menu items and their parameters. The goal is for the Agent to be able to do anything the user can do directly. See the sidebar "NewWave Command and Function Summary," which has some commands from the Agent task language.

An Agent can control individual applications by communicating with them directly in their own language of commands. As a result, NewWave applications no longer need the extra code necessary to support their own nonstandard internal macro languages. Users can

Figure 21: HPSHAPE Main Procedure

```
/*************************************************************/
                    ShapeWndProc
  Main procedure to handle all messages sent to HPSHAPE
      COPYRIGHT HEWLETT-PACKARD COMPANY 1987, 1988
/*************************************************************/

long FAR PASCAL ShapeWndProc( hWnd, message, wParam, lParam )
    HWND            hWnd;
    unsigned        message;
    WORD            wParam;
    LONG            lParam;
{
    APIRTNTYPE      applRtn;  /* Used as a return value */
    APICMDSTRUCT    extCmd;   /* The external command structure for API */
    INTCMDSTRUCT    intCmd;   /* The internal command structure for API */


    /* Function called for windows that are created with          */
    /* NW_CreateWindow to specially handle some messages          */

   if( NW_MessageFilter(hWnd, message, wParam, lParam,
                        (LONG FAR *) &applRtn))
         {
        return (applRtn);
         }

    /* Is API to intercept messages or has an API menu item been    */
    /* selected such as the help or the task?                       */

   if( APIInterceptOn(gAPIModeFlags) || APIHaveMenu (message, wParam))
         {
        /* Is this a message for the API - may set certain flags?   */
      APIUserActionInterface(ghAPI, hWnd, (LPAPIUNSIGNED)&message,
                             wParam, lParam, API_NO_MODE);
         }

   applRtn = (APIERRTYPE)0L;
   /* Still a command to be handled? Or did APIUserAct take it?  */
   if( APIHaveMessage (message) )
         {
        intCmd wCmd = API_NO_CMD;

       /* Are you currently playing back a message or recorded task? */
       if (APIPlaybackMsg(message))
           /* Translate to the internal format of the application   */
         TranslateToInternalProcessor(message, wParam, lParam,
                                      &intCmd);
        else
           /* You are in record mode or a user interactive command  */
         ActionProcessor(hWnd, message, wParam, lParam, &intCmd,
                         &applRtn);

       /* Is there a command created in the action processor
          that needs to be executed? */
       if( APIHaveCommand(intCmd.wCmd) )
            {
           gapplErr = API_NO_ERR;

           /* Are you in  a CBT? */
           if( APIMonitorOn(gAPIModeFlags) )
               {
              /* Set to external language and pass the external
                  form of the command to the Agent                  */

             TranslateToExternalProcessor(&intCmd,&extCmd);
             APICommandInterface(ghAPI, (LPAPICMDSTRUCT)&extCmd,
                                 API_NO_MODE);
              /* The internal command must be canceled if the     */
                APICommandInterface or APIClgCommandInterface
                 has nullified the command   */
              if (extCmd.wCmd == API_NO_CMD)
                 intCmd.wCmd = API_NO_CMD;
               }

           /* Has the command been formed and executed? */
           if( APIHaveCommand(intCmd.wCmd) )
```

CONTINUED

Figure 21 CONTINUED

```
            /* Perform the command   */
            CommandProcessor(hWnd, message, wParam, lParam, &intCmd,
                             &applRtn);

          /* Is a command being played back or being recorded?    */
          if( APIPlaybackOn(gAPIModeFlags) ||
                          APIRecordOn (gAPIModeFlags))
              {
              if( APIRecordOn(gAPIModeFlags) )
                  {
                  /* Translate to ext  lang  and tell Agent that
                    command is complete and ready for next
                    command.                                    */
                  TranslateToExternalProcessor(&intCmd,&extCmd);
                  APIRecordInterface(ghAPI,(LPAPICMDSTRUCT)&extCmd,
                                     API_NO_MODE);
                  }
              /* Return control to the user   */
              APIReturnInterface( ghAPI, gapplErr, API_NO_MODE );
              }

          } /* EndIf of APIHaveCommand   */

        } /* EndIf of APIHaveMessage   */

    return(applRtn);

} /* End of ShapeWndProc   */
```

Figure 22: HPSHAPE Action Processor

```
/*********************************************************************/
                    ActionProcessor
  Handle all of the user actions - set up intCmd for Cmd Processor
       COPYRIGHT HEWLETT-PACKARD COMPANY 1987, 1988
/*********************************************************************/

void PASCAL ActionProcessor (hWnd, message, wParam, lParam, intCmd,
                             pRtn)
    HWND          hWnd;
    unsigned      message;
    WORD          wParam;
    LONG          lParam;
   PINTCMDSTRUCT intCmd;
    LONG         *pRtn;    /* The return value of the procedure */
{
    PAINTSTRUCT   ps ;         /* The paint structure */

   /* This routine actually builds a command */
   switch( message )
       {
       /* Was a paint message sent to update the screen? */
       case WM_PAINT:
           BeginPaint    (hWnd, (LPPAINTSTRUCT)&ps);
           ShapePaint    (hWnd, (LPPAINTSTRUCT)&ps);
           EndPaint      (hWnd, (LPPAINTSTRUCT)&ps);
           break;

       /* Was there a menu selection? */
       case WM_COMMAND:
           /* Was the close option chosen? */
           if (wParam == IDDCLOSE)
              intCmd->wCmd = API_CLOSE_WINDOW_CDCMD;
           else {
               /* Something was selected from the shape menu */
               intCmd->wCmd = NEW_SHAPE;
               intCmd->internal.ICmd = wParam;
               }
           break;

       /* The window being closed */
       case WM_CLOSE:
          intCmd->wCmd = API_CLOSE_WINDOW_CDCMD;
           break;
```

CONTINUED

control all their NewWave applications with a single task language.

Agent scripts can be generated by capturing a series of user commands (see **Figures 12** and **13**) or by creating them directly with an ordinary text editor. The language supports control structures such as loops, branching, and procedures; it also supports integer, float, and string variables. For performance, the scripts are compiled into a binary form before being run.

In the future, a built-in scheduling system will allow Agents to perform tasks at specified times, such as a single time or regularly every day or week (see **Figure 14**). An Agent could be told to wait for a trigger event before it starts running. For example, the receipt of a piece of E-mail could trigger the data being placed into a report.

Figure 15 shows a simple Agent task script. Note that because this article was written before the release version of NewWave was ready, the PASTE and CLOSE commands had to be used in intermediate pcode form.

NewWave also adds to Windows with its user support. NewWave includes prewritten CBT, Help, and Native Language Support (NLS) systems. NewWave is in a unique position to offer first-rate Help and CBT services because of the similarity of its Help, CBT, and Agent macro facilities. All three systems are concerned with monitoring and controlling the execution of individual applications. In fact, the CBT lessons are written in an extended form of the Agent macro language. As you have seen, all user actions must pass through a NewWave application's Action Processor before being translated into commands that are executed by its Command Processor. The CBT system can watch and

modify this traffic of commands to control the user's interaction with the system. The CBT can, for example, intercept, recognize, and point out correct and incorrect user responses during a lesson. NewWave provides tools for creating CBT and Help documents. Its built-in CBT animation development system, shown in **Figure 16**, helps developers design documentation and Help files at the same time they are writing the program.

Conventional CBT systems often attempt to simulate the behavior of specific parts of the application. Rather than adding new simulation code, NewWave CBT simply uses the application itself by sending its commands directly to the Command Processor.

Converting Applications

NewWave has built-in support to help convert an application to other languages and customs. First, a single generic version of the program is written. Later, nontechnical translation workers can use the tools NewWave provides to adapt the local character handling (see **Figure 17**) and customs support without changing any code.

In order to see how NewWave capabilities can be added to an existing Windows application, we will look at portions of the code for a Shapes program that comes with the Windows SDK. The NewWave version is called HPShape. Shapes does only one thing: it draws one of four geometric figures selected from its menu—a triangle, an ellipse, a rectangle, or a star. HPShape has been made into a typical source object that projects its visual view into a destination object.

The NewWave SDK includes a second program called HPLayout as a sample destination object. To nest a visual view of HPShape in HPLayout, mark off an area inside the HPLayout window with the mouse. The nested area is shown in reverse (white foreground on a black background). To project the HPShape view into this small area, drag HPShape's icon over it and drop it in. If you want to manipulate the nested HPShape object, double click on the nested area to bring up HPShape's main window. **Figure 18** shows the compound document, SDK NW Layout Demo, with its two nested children open. The areas where they normally appear in the docu-

Figure 22 CONTINUED

```
        /* Selection from system menu or minimize/maximize */
        case WM_SYSCOMMAND:
            switch (wParam)
                {
                case SC_MINIMIZE:
                    intCmd->wCmd = API_MINIMIZE_WINDOW_CDCMD;
                    break;

                case SC_MAXIMIZE:
                    intCmd->wCmd = API_MAXIMIZE_WINDOW_CDCMD;
                    break;

                /* Request for saving previous coordinates */
                case SC_RESTORE:
                    intCmd->wCmd = API_RESTORE_WINDOW_CDCMD;
                    break;

                default:
                    /* Let windows handle it */
                    *pRtn = DefWindowProc(hWnd, message, wParam,
                                                          lParam);
                    break;

                } /* End of case WM_SYSCOMMAND */
             break;

        case API_INTERROGATE_MSG:

            /* The message has come via the API */
            *pRtn = InterrogateFromAPI(wParam, lParam);
            break;

        case  API_SET_MODE_FLAGS_MSG:
              if (wParam == API_SET_MODE_ON_FLAG)
                gAPIModeFlags = gAPIModeFlags | lParam;
              else
                gAPIModeFlags = gAPIModeFlags & lParam;
             break;

       case WM_OMF:

          /* The message has come via the OMF */
         *pRtn = MessageFromOMF(hWnd, wParam, lParam);
          break;

       default:
         *pRtn = DefWindowProc(hWnd, message, wParam, lParam);
          break;
        }
} /* End of ActionProcessor */
```

AN AGENT CAN CONTROL INDIVIDUAL APPLICATIONS BY SPEAKING TO THEM IN THEIR OWN LANGUAGE OF COMMANDS. AGENT SCRIPTS ARE CREATED USING A TEXT EDITOR OR BY CAPTURING USER COMMANDS.

Figure 23: HPSHAPES Command Processor

```
/*********************************************************************/
                CommandProcessor
  Handle any of the commands passed from the ActionProcessor or API
       COPYRIGHT HEWLETT-PACKARD COMPANY 1987, 1988
/*********************************************************************/

void PASCAL  CommandProcessor (hWnd, message, wParam, lParam, intCmd,
                               pRtn)
    HWND          hWnd;
    unsigned      message;
    WORD          wParam;
    LONG          lParam;
   PINTCMDSTRUCT intCmd;
    LONG         *pRtn;          /* Application return error code */
{
    HMENU         hMenu;
    RECT          rcRect;

   switch( intCmd->wCmd )
       {
      case API_MINIMIZE_WINDOW_CDCMD:
           if (IsIconic(hWnd))
               NW_Restore(hWnd);
            else
                {
               if (!IsZoomed(hWnd))
                   GetWindowRect(hWnd, (LPRECT)&gWinPosn.rcRect);
               NW_Minimize(hWnd);
                }
            break;

      case API_MAXIMIZE_WINDOW_CDCMD:
           if (IsZoomed(hWnd))
               NW_Restore(hWnd);
            else
                {
               if (!IsIconic(hWnd))
                  GetWindowRect(hWnd, (LPRECT)&gWinPosn.rcRect);
               NW_Maximize(hWnd);
                }
            break;

      case API_RESTORE_WINDOW_CDCMD:
           NW_Restore(hWnd);
            break;

      case API_CLOSE_WINDOW_CDCMD:

           SaveWindowPosition(hWnd);
          GetWindowRect(hWnd, (LPRECT) &rcRect);
           ShowWindow(hWnd, SW_HIDE);
           UpdateWindow(hWnd);
          APINotReady(ghAPI, API_NO_MODE);
          if (!OMF_Closing(ghOMF, (LPRECT) &rcRect))
              NoteError();
           break;

       case NEW_SHAPE:
           /* Get the handle to the menu of the current window */
           hMenu = GetMenu(hWnd);

           /* Uncheck the old menu item */
          CheckMenuItem(hMenu, gnShape, MF_UNCHECKED);

           /* Check the new menu item */
          if ((gnShape = intCmd->internal.ICmd) != SHAPE_NONE)
              CheckMenuItem(hMenu, gnShape, MF_CHECKED);

           /* Send a paint message */
          InvalidateRect(hWnd, (LPRECT)NULL, TRUE);
          UpdateWindow(hWnd);  /* Force repaint for every shape,
                                  not just last shape, when playing
                                  back several NEW_SHAPE commands */

           /* Are there views of the shape program in any other    */
```

CONTINUED

ment are gray. The object called Nestable HP Text is another simple source object that comes with the NewWave SDK. It projects a visual view of text.

Getting NewWave's additional functionality comes at the price of a fair amount of extra code. Whereas the simple Shapes WinApp weighs in with only 7Kb of C source code, the HPShape and HPLayout applications require 55Kb and 157Kb of C source code, respectively. Of course, that added code gives you a help system, a programming language, and data and visual linking capability. The amount of added NewWave code is also relatively fixed in size. For a normal sized program it will be a smaller percentage of the total code.

Figures 19 and **20** are calltrees for the programs produced with Microsoft CALLTREE.EXE utility. Calltree listings show the hierarchy of function calls for a C program. The calls made from a function are indented beneath it. **Figure 19** shows the structure of the pre-NewWave Shapes; **Figure 20** shows the HPShape NewWave program. The full source code for three HPShape functions (ShapeWndProc, ActionProcessor, and CommandProcessor) is shown in **Figures 21, 22,** and **23.**

The data file referred to in the figures is used to record the specific shape that was being displayed in the HPShape window. This means that even after the NewWave application is closed, its state persists and when it is reopened this state is restored.

As with all Windows and Presentation Manager programs, a NewWave program is essentially a large message processor. The program spends its life responding to the spectrum of messages that enter its main window procedure. It separates the messages into general categories and passes them to more

specialized handlers. **Figure 24** contains the messages recognized by the pre-NewWave Shapes; **Figure 25** contains the considerably broader list to which HPShape must respond.

Conventional Applications

In order for applications to exploit NewWave fully, they must be specifically written to interact with its new interfaces. But while these fully NewWave-capable applications are being written it is important for NewWave to be able to interact with any conventional MS-DOS and Windows programs such as Lotus® 1-2-3®. NewWave provides several methods for encapsulating these existing applications to make them more functional in NewWave.

At its lowest level, NewWave allows existing programs to be run from a menu. This requires no encapsulation at all. Windows applications as well as character mode or graphic MS-DOS programs that take over the whole screen can be run this way. The user can context switch between the program and the rest of NewWave and use the clipboard to cut and paste between it and other applications. Integrating a new program to operate at this level takes only a few minutes.

Moving up the integration scale requires the encapsulation of the existing program into a shell created by an interactive installation tool provided by NewWave. The lowest level of encapsulation takes about an hour. It allows data files created by the program to be represented as an icon in the NewWave Office. These objects can be manipulated in most of the same ways that a full NewWave application can. They can be opened with a double click or moved, copied, filed, mailed, or discarded with the drag-and-drop technique. The disadvantages are that they allow neither data nor visual links to other objects nor can they use the full Agent task language, help, or training systems.

The highest level of encapsulation requires considerable programming, but doing that programming is significantly easier than completely rewriting the application as a native NewWave application. The new code primarily adds support for data and visual links. It consists of a browser program that understands the format of the application's data and can talk to the NewWave OMF. For example, when a destination object sends a message asking an encapsulated source object to display a view of itself in a given rectangle, the browser reads the object's data file and displays the data to the screen. If it is necessary to edit the data, the browser can invoke the necessary application to do so.

A full NewWave application goes further by providing complete support for the Agent task language, context-sensitive help, NewWave user interface, and Computer-Based Training.

Machine Requirements

The minimum hardware that HP recommends for NewWave users is a 286 PC/AT® or 100 percent compatible, 3Mb of LIM 4.0 EMS memory, a 20Mb hard disk, and an EGA display (a VGA is significantly better). Developers should increase this to 4Mb of EMS memory. A

Figure 23 CONTINUED

```
            /* object?  Set the new_data flag for all views of      */
             /* shapes */

            if (OMF_SetNewData(ghOMF, 0))
                 {
                /* Notify the destinations */
                if (!OMF_AnnounceNewData(ghOMF))
                     NoteError();
                 }

             break;

        default:
            NoteError();
            break;
         }

} /* End of CommandProcessor */
```

Figure 24: Messages Recognized Before NewWave Enhancements

Conventional Windows applications deal with two primary types of messages.

Messages resulting from user actions

These result from mouse clicks, mouse movements, menu selections, or keystrokes. They are acted on immediately.

WM_COMMAND	Results from menu selection
WM_SYSCOMMAND	User selected something from system menu or minimize/maximize

MS Windows housekeeping messages

WM_CREATE	Sent when CreateWindow is called
WM_PAINT	Sent to update the screen
WM_ERASEBKGND	Sent when window background needs clearing
WM_DESTROY	Terminate the message loop

Figure 25: Messages Recognized After NewWave Enhancements

NewWave applications must process five types of messages.

Messages resulting from user actions

These messages result from mouse clicks, mouse movements, menu selections, or keystrokes. They are not acted on directly, but rather are translated by the Action Processor into commands for the Command Processor in the application's own task language. (See message type 5.)

WM_COMMAND	Results from menu selections.
WM_SYSCOMMAND	User selected something from system menu or minimize/ maximize.

OMF Messages

These messages come from the OMF as WM_OMF messages. They are used to communicate either with the OMF itself or with other objects. They each correspond to an OMF method that this application implements. A method is the code that is executed in response to a given message: for example, the code needed to create, open, or terminate this object.

CREATE_OMF	Respond by reading data file to remember the "persisting" state of the object when it was last closed. Was it a triangle, and so on.
WARM_START	Indicates that the receiving object was shut down in a consistent state the last time it was run. The object can treat as valid any context or state data that was saved at shutdown.
OPEN	Respond by reading properties to find the last size and position of window. Move the window there.
HAS_METHOD	Respond to say if a particular OMF "method" is supported by HPShape.
INIT_VIEW	Respond by setting up a view specification for the data to be displayed in a requesting object.
GET_SIZE	Respond by saying what size HPShape needs to allow it to display itself as a view in a destination object.
DISPLAY_VIEW	Paints the visual view from HPShape in the DC specified by a destination object.
COPY_SELF	Allows this object to be copied when it iscontained in another object.
TERMINATE	Respond by saving current shape in data file.
DIE_PLEASE	Tells this object to terminate itself.
WINDOW_TO_TOP	Tells this object to bring its window to the front.

NewWave API messages

User request
Agent request
Interrogate messages
API_INTERROGATE_MSG
A request for information from the application.
API_SET_MODE_FLAGS_MSG
Used to change mode the application is in, such as Playback, Record, and Monitor.

MS Windows Housekeeping messages

WM_PAINT	Sent to update the screen.
WM_CLOSE	User closing window.

Internal Messages for the Command Processor

These messages originate inside the application. They may result from a translated user action or from an Agent script being played back. Each corresponds to a command in the application's task language.

API_MINIMIZE_WINDOW_CDCMD
Requests application to iconize itself

API_MAXIMIZE_WINDOW_CDCMD
Requests application to grow to take up the whole screen.

API_RESTORE_WINDOW_CDCMD
Requests application to return its window to the previous size.

API_CLOSE_WINDOW_CDCMD
Requests application to close its main window.

NEW_SHAPE	Rerender the current shape.

40Mb or larger hard disk is recommended. Developers will also need Microsoft Windows/286™ Version 2.11, and Microsoft C 5.1 or higher. As with most Windows or Presentation Manager work, a fast 386 machine significantly increases productivity.

NewWave enhancements to Windows programs come at the cost of increased code, although that cost can be lessened by choosing the level of encapsulation your program requires. Windows programmers who are interested in extending and enhancing their Windows programs must give serious consideration to programming for the NewWave environment. ❑

[1]For ease of reading, "MS-DOS" refers to the Microsoft MS-DOS operating system. "MS-DOS" refers only to this Microsoft product and is not intended to refer to such products generally.
[2]For ease of reading, "Windows" refers to the Microsoft Windows graphical environment. "Windows" refers only to this Microsoft product and is not intended to refer to such products generally.

Emulating the UNIX® RS-232 General Serial I/O Interface Under DOS

Michael J. Chase

One popular means of transferring information between DOS[1] or UNIX® applications and terminals or control devices is the use of RS-232 serial communication. This article focuses on writing applications that rely on RS-232 serial communication under both DOS and UNIX. We will explore the generalized serial I/O interface provided under UNIX and a device driver that emulates it under DOS.

UNIX programmers have a general interface for asynchronous serial devices that is independent of hardware; it has many useful features and, once understood, is very easy to work with. Programmers working in the DOS environment, however, usually can't use the COM1 or COM2 serial device interfaces because they are not interrupt driven and do not support buffering or XON/XOFF handshaking. Third-party communications libraries are often called upon to help, but most require the C programmer to learn at least 20 function calls to a proprietary interface—20 more than most people would prefer to have to learn. Moreover, third-party communication code usually will not port to UNIX-based systems or to another vendor's DOS communication libraries.

Documentation for the generalized UNIX serial device seems incredibly terse and hard to read if you are not familiar with the many details surrounding asynchronous communications. After some explanation, however, the serial device interface becomes easy and convenient to use. Usable features (buffering, XON/XOFF handshaking, watchdog timers, parity control, exception handling, line disciplines, and so on) of this general interface are presented so that portable device control software can be written for general and embedded applications under both UNIX and DOS. Trade-offs in communications software design (buffer sizes, communication attributes, error recovery, and so on) are also discussed.

DOCUMENTATION FOR THE GENERALIZED UNIX SERIAL DEVICE IS INCREDIBLY TERSE AND HARD TO READ IF YOU ARE NOT FAMILIAR WITH THE MANY DETAILS SURROUNDING ASYNCHRONOUS COMMUNICATIONS. AFTER SOME EXPLANATION, THE SERIAL DEVICE INTERFACE BECOMES EASY TO USE.

Communication Device Drivers

A major goal of all device drivers is to provide a logical software interface for applications that isolates them from physical hardware. The physical hardware can differ; however, the logical software interface to the hardware (device) remains the same. This is true for block-oriented devices (disk interfaces) as well as character-oriented devices (serial port interfaces).

Michael Chase is a principal of the Boulder Software Group, which provides contract programming services and instruction on C and UNIX to AT&T, DEC, and IBM. He is also a faculty member in the Univ. of Colo. M.S. Telecommunications program.

Asynchronous Communications Character-at-a-time transmission. Characters are randomly sent one at a time, far apart or close together. They are separated by start and stop bits. (See *Framing*.)

Baud Symbols per second. Each symbol usually contains analog information for 2^s:n: bits (n = 0, 1, 2, 3, ...). For example, a symbol that has four different voltage levels (n = 2) contains information for 2 bits; thus a baud rate of 600 will yield an effective bit rate of 1200 bps.

Blocked Process If a program (process) is waiting for an I/O completion, it may be put to sleep; that is, marked as no longer eligible to run. The process is unblocked (awakened) when the I/O request is complete; it is then eligible to run.

BPS Number of binary transitions per second. (See *Baud*.)

Break BREAK asserts the SPACE condition on the serial line for approximately 0.25 to 0.30 seconds. It may cause a framing or overrun error to occur. It is typically used to reset the serial link to some known state.

DCD Data Carrier Detect. An electrical signal stating the presence or absence of a modem carrier. The DCD is used to indicate when two or more modems have established a connection.

DTE/DCTE Data terminal equipment. Usually a terminal device capable of generating or displaying information. Data circuit terminating equipment. Usually a device capable of transmitting or receiving data over a chosen medium. Examples are a modem and a line driver.

DTR/DSR Data Terminal Ready/Data Set Ready. A DTE advises that it is ready to converse by raising DTR. The DCTE advises that it is ready to converse by raising DSR. DTR and/or DSR are usually active for the duration of a conversation.

Flow Control A method or protocol for governing the starting and stopping of transmission. It is used so that resources can be managed to accommodate incoming data; typically it advises a sending entity that a receiving buffer is nearly full (stop transmitting) or nearly empty (start transmitting).

Framing For asynchronous character transmission, framing identifies the meaning of bits that constitute a character. The number of start bits (1; SR), information bits (5, 6, 7, or 8; D0..D7), optional parity bit (0, 1; P), and stop bit(s) (1.0, 1.5, 2.0; SP) are identified.

Example:
| SR | D0 | D1 | D2 | D3 | D4 | D5 | D6 | D7 | P | SP |

Hardware Handshaking Flow control that is managed at the hardware level. The UART waits for permission to send characters, which is granted through the RS-232 RTS/CTS signals.

Level 1 Direct I/O The C programmer has the choice of two levels of I/O: level 1 or level 2. Level 1 I/O sends and receives characters directly from the operating system (the operating system may buffer characters).

Level 2 Stream I/O The C programmer has the choice of two levels of I/O, level 1 or level 2. Level 2 I/O sends and receives characters to and from stream buffers (typically 512 bytes long) maintained by the C standard library. Level 2 is written as a function of level 1. When write buffers become full, they are flushed out to the operating system; when read buffers become empty, more characters are requested from the operating system.

Line Discipline Conversion rules chosen for the c_lflag in the termio structure. A mapping or filter strategy for transferring characters from the raw input queue to the canonical input queue.

Mark An electrical signal that denotes a logical binary 1. Under RS-232, typically –3 to –25 VDC.

Overrun When too many data bits are received between the start and stop bit, an overrun condition has occurred.

Parity A bit designated for error-detection purposes. There are two basic types of parity—even and odd. If the parity bit is designated as odd (or even), its job is to take a value that makes the total number of 1s in an asynchronous character transmission odd (or even); for example, 00010101P; P = 0 odd parity (P = 1 even parity). Parity errors detect some, but not all, types of transmission errors.

Ring Buffering A circular buffer or queue of finite size that has no physical beginning or end. The logical beginning and logical end are typically tracked with pointers. The next character put in the queue is at the logical end of the buffer; the next character taken out of the queue is at the logical beginning. If the buffer is full and a character is put in the queue, the oldest character in the queue may be overwritten. A ring buffer of size N will always contain at most the last $N - 1$ characters in the queue.

RS-232 Recommended Standard 232. RS-232 is administered by the Electronic Industries Association (EIA). Electrical characteristics for transmission of information and signaling information is specified. RS-232 is also known as CCITT V.24. Physical characteristics of a connector are also specified; for example, the common DB-25.

RTS/CTS Request to Send/Clear To Send. Hardware signals used in flow control. A DTE can request permission to send characters by raising RTS and be granted permission with the activation of CTS by the DCTE.

SDLC Synchronous Data Link Communications. (See *Synchronous Communications*.)

Software Handshaking Flow control that is managed at the software level. Software typically sends an XOFF (ASCII DC3; 13H) character to request the suspension of character transmission (receive buffers are nearly full) and an XON (ASCII DC1; 11H) character to resume a suspended transmission (receive buffers are nearly empty).

Space An electrical signal that denotes a logical binary 0. Under RS-232, typically +3 to +25 VDC.

Synchronous Communications Block-at-a-time transmission. Groups of characters are sent in blocks of bits. There are no start or stop bits to separate characters; there are, however, bit patterns (flags) that separate blocks of bits.

UART Universal Asynchronous Receiver Transmitter. A hardware device whose primary function is to perform serial-to-parallel conversion of electrical signals.

XON/XOFF Characters that have been chosen for software flow control. An entity receiving an XON is advised to resume (begin) transmission. An entity receiving an XOFF is advised to suspend (stop) transmission. (See *Software Handshaking*.)

Both UNIX and DOS provide block and character device drivers that allow C programs to read and write bytes by using standard library calls through the file system. Familiar I/O function calls such as open, read, write, and close (level 1 functions) or stream-buffered functions such as fopen, fread, fscanf, fwrite, fprintf, and fclose (level 2 functions available in the standard library) can be used to communicate with a file or a serial device. These functions afford programmers writing serial I/O software in UNIX and DOS portability across many hardware platforms.

There is one problem though. The default asynchronous device driver shipped with DOS is not very powerful because it does not allow you to change speed, parity, buffer control, handshaking, and so on, independent of the hardware. To control a serial device from a C program, calls to the BIOS must be made—a serious impediment to portability. Fortunately there are three viable methods you can use to write serial I/O software for the DOS environment: you can build your own communications software, use linkable communications libraries provided by a third party, or use a device driver capable of emulating the popular UNIX serial I/O interface under DOS.

Building Software

Building a communications functions library from scratch has certain advantages. It gives you control over exactly what you need, it helps you learn about communication port hardware, and it gives you ownership of source code.

By and large, though, you will be reinventing the wheel. You will need a hardware debugger and/or a logic analyzer to catch the most subtle bugs, especially those found in interrupt logic. You will also need to invest a great deal of time in supporting, updating, and maintaining the software—in other words, you will become your own technical support staff.

Third-Party Libraries

Third-party add-on communication libraries may help you write serial I/O programs by providing a C language interface for controlling device attributes. Your C code, however, will be married to these typically nongeneral interfaces under DOS. Therefore, you must weigh both the advantages and disadvantages of this scheme.

An advantage of third-party libraries is their highly granular control over speed, parity, stop bits, XON/XOFF, buffer management, and so on. Another advantage is not having to install a device driver, since device drivers have to be loaded when a system is booted and then remain part of the operating system. I/O functions loaded with your programs, however, are not resident in the operating system, do not consume any space when your program exits, and do not incur operating system overhead to pass information through the file system.

One disadvantage is that the interface to a third-party library is not through the file system; another is that calls to fopen, fread, fscanf, fwrite, fprintf, and fclose will not work. You must devote the time to learn a new set of interface functions supported by a given vendor. Some interfaces are unnecessarily complex; one commercial vendor, for example, boasts more than 125 serial communication function calls. Another disadvantage is that simple command line (or batch file/shell script) redirection is not supported. For example, under DOS the simple command line

```
C> dir > \device\tty01
```

will not work. Further, the logical interface is vendor specific; C code that is portable between DOS and UNIX is nearly impossible to write. Finally, some third-party libraries are designed as terminate-and-stay resident (TSR) and consume memory.

If you only develop for DOS and portability is not an issue, or if you only need a serial device interface for DOS that emulates the widely accepted UNIX serial interface, the advantages probably outweigh the disadvantages. In that case, and with the bridging of the UNIX and DOS worlds, this option deserves consideration.

Emulating UNIX Serial I/O

A UNIX-compatible serial device driver under DOS provides a number of useful advantages. First, it gives you sufficient control over speed, parity, stop bits, XON/XOFF, buffer management, and so on. Second, C code that controls serial I/O is portable between DOS and UNIX. UNIX programmers need not learn a new serial I/O interface for DOS and vice versa. Third, the interface is through familiar file system calls (such as fopen and fprintf) so command line (and batch/shell) redirection is possible. In fact, the interface is so general that it allows any language-supporting file I/O (C, Assembler, Pascal, Clipper, and so on) to use the serial device. Fourth, the controlling interface is via one standard function call, namely IOCTL. It is the same under both UNIX and DOS. Fifth, standard DOS-critical error handlers can be used to trap run-time exceptions.

Emulation, however, presents its own set of problems. Since device drivers are only loaded once, at boot time, and DOS supports no explicit resource management, an existing appli-

Figure 1: Interface Specification for ioctl()

```
#include  <termio.h>

int ioctl( fd, command, argument);
int fd;
int command;

union
{
   int i_arg;
   struct  termio  *s_arg;
} argument;
```

ioctl()

Returns 0 if successful in performing the requested command; returns -1 otherwise and errno is set to reflect the reason for the error. EBADF, ENOTTY, EINTR, EFAULT, EINVAL, EIO, ENXIO and ENOLINK are possible values for errno; see the include file <errno.h>.

fd

A valid file descriptor (DOS file handle) obtained from a successful call to open() or fopen(). It must be possible to write to the file to set device attributes and it must be possible to read from the file to get device attributes.

command

There are seven commands that tell the device driver how to respond. Note that the value and type of the last parameter, argument, will depend on the command selected.

TCGETS

Get current attributes from the device (TCGETA on some UNIX systems). argument.s_arg points to the structure that will receive the existing attributes.

TCSETS

Immediately set the passed attributes in the device (TCSETA on some UNIX systems). argument.s_arg points to the structure that contains the attributes to be set.

TCSETAW

Set passed attributes after the output buffer has drained. argument.s_arg points to the structure that contains the attributes.

TCSETAF

Wait for the output to drain, flush the input queue, then set the new attributes. argument.s_arg points to the structure that contains the attributes.

TCSBRK

Send a break sequence for 250 milliseconds. argument.i_arg must be 0.

TCXONC

Start (XON) or stop (XOFF) the transmission of output from the device. If argument.i_arg is 0, output is suspended; if 1, suspended output is restarted.

TCFLSH

Flushes the input and/or output queues. If argument.i_arg is 0, flush the input queue; if 1, flush the output queue; if 2, flush both the input and output queues.

argument

The union argument contains one of two data types:

```
struct termio * s_arg
```

or

```
int i_arg
```

The type depends on the command being issued. s_arg is a pointer to a C structure that will be used for setting or getting device attributes. i_arg is an integer.

cation (a TSR, for example) could corrupt the device driver by competing for the same communication port hardware. (Add-on communication function libraries can also suffer the same dilemma.)

The device driver always consumes memory, even though it may not be used. The sample DOS serial device driver accompanying this article emulates the UNIX driver by consuming almost 8Kb, excluding buffers. (*Code for the device driver may be downloaded from any* MSJ *bulletin board—Ed.*) Furthermore, applications depending on the driver cannot load it before executing, although they can test for its absence and recover. Microsoft® documents no formal mechanism (that is, a DOS function call) for dynamic device driver loading, except for its mouse driver.

Finally, since DOS is not reentrant, and must be entered to gain access to the device driver, buffers cannot be resized at run time. They must be sized at boot time and remain fixed.

Of the three methods you can use to write serial I/O software in the DOS enviroment, the best one is emulation of the UNIX serial I/O device driver. It is the most general solution for a given need; which means it best meets the goals of providing transportable code, a standard interface, and minimization of the learning curve. The issues you will need to address when developing serial I/O devices are discussed next.

Blocking and Nonblocking I/O

Normally when a function call is made requesting information from a block device (a disk), the function call will return only when the request is satisfied or when an error occurs. For example, the following call to read will wait until 10 characters are transferred or an error occurs:

```
if(read(fd, buffer,10)!=10)
{
```

```
    /* process the error */
}
```

If the read is from the disk, the request is usually not satisfied until the disk access is complete. Under the DOS operating system, the application program waits for DOS to service the request completely. Under a preemptive multitasking system such as UNIX, the program (or process) is put to sleep (that is, the program blocks) until the request is satisfied; then it is awakened (or unblocked) and is eligible to run. Note that while the process is asleep, the operating system can serve other processes. DOS, however, is a single-tasking system, so a program must patiently block until DOS returns control—nothing else can run.

If you have a serial communications link with the same read calling for 10 characters, and no other characters are available, your process will block under both UNIX and DOS. If nothing shows up, the DOS program may block forever. The UNIX program also blocks, but other programs can run. Both DOS and UNIX, however, can be convinced not to block if nothing is available. Several issues related to blocking and nonblocking I/O must be addressed: The minimum amount of time and/or number of characters for which a read request waits to be assembled before being partly or fully satisfied; the number of *m* characters (if available) with *n* requested ($m<n$), that should be picked up; whether the request is free-form and just asks for a completed line (that is, whether all characters are up to a '\n' or some other line delimiter); whether or not line editing is available so that mistakes can be corrected; and how the EOF condition will be detected and handled. These issues are explored in later examples; as it turns out, they are all options that can be configured through a general interface.

Figure 2: Two Methods of Opening a Serial Device

```
#include <stdio.h>

#define   SERIAL_DEVICE   "/dev/tty01"

static   int     fd_tty;
static   FILE    *fp_tty;

/*****************************************************************/
/*    L E V E L   1   O P E N   M E T H O D      */

if( ( fd_tty = open( SERIAL_DEVICE, 2 ) )  !=  -1 )
{
   /*  fd_tty now a valid UNIX file descriptor */
   /*  fd_tty now a valid DOS file handle      */
}
else
{
   /*  The device can't be opened.  */
}
/*****************************************************************/
/*     L E V E L   2   F O P E N   M E T H O D      */

if( ( fp_tty = fopen( SERIAL_DEVICE, "rwb" ) )  !=  NULL )
{
   /*    fp_tty now a valid stream pointer          */
}
else
{
   /*    The device can't be opened.                */
}
```

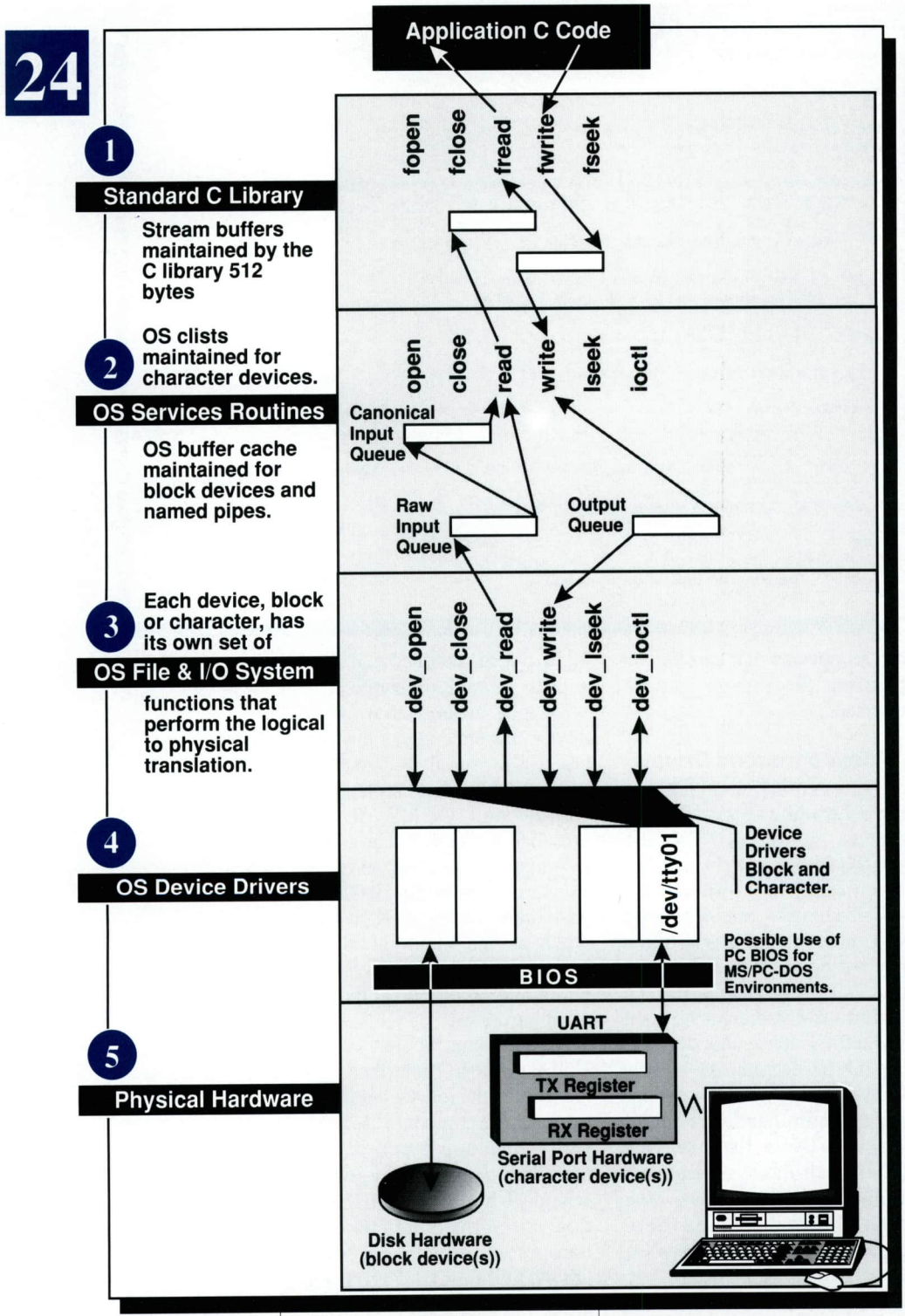
Asynchronous Driver

There is one general interface for controlling serial I/O parameters under UNIX—the IOCTL system call. Its interface specification is shown in **Figure 1**. If you have worked with DOS, you know that there is a similar, but not equivalent, DOS function call (44H) named IOCTL. Unfortunately, there is no documented evidence that lets you use IOCTL to control COM1 or COM2 attributes such as bps, parity, and hardware handshaking. (Note that the DOS IOCTL call allows several operations to be performed other than those discussed here, especially on block devices.) BIOS calls must be used.

Microsoft borrowed the UNIX device driver philosophy for DOS, but did not generalize the DOS asynchronous serial I/O device interface as they did the disk drive interface. As a consequence, you must use one piece of serial hardware, the 8250 UART, mapped at a fixed address through the BIOS.

To get around that, DOS device drivers that use the general UNIX serial I/O interface are available; they replace the COM1 and COM2 drivers provided with DOS. The UNIX terminal interface serves as an excellent model for DOS, as have many other UNIX features (such as the hierarchical file system, the I/O subsystem, redirection, pipes, and environment variables).

Before using the generalized UNIX serial I/O interface, you must know the following: the bit rate, character data length, parity, and stop bit(s) requirements for communications; the scheme to be used to manage the flow of information in both the transmit and receive directions (XON/XOFF, RTS/CTS, DTR/DSR); how the device handles exceptions such as loss of modem carrier, a break signal, or ^C; the kind of input or output post-processing, if any, that needs to be done (for example, CR–>NL, NL–>NL/CR, convert cases, expand tabs);



▲ Figure 3

I/O Paths: *The path a character will take when an application writes it to an output stream (or reads from an input stream).*

❶The application opens the device with a call to fopen(). The application then sets up a buffer, count, and so on, and calls fwrite(). A call to write() bypasses (2), the C standard library stream buffers.

❷fwrite() transfers bytes from the user buffer to a stream buffer maintained by the standard library. Stream buffers are usually 512 bytes in length (see the definition for BUFSIZ in the include file <stdio.h>). Writing or reading bytes to or from a stream buffer saves the overhead of making a context switch to the operating system. When the stream buffer becomes full, its contents are flushed out to the operating system with a call to write.

❸write() transfers bytes from the stream buffer to internal operating system buffers (known as clists under UNIX) if the destination is a character device. Under DOS, fixed buffers are most likely maintained by the device driver itself. If the destination is a block device, write() transfers bytes from the stream buffer to an operating system buffer (known as the buffer cache under UNIX).

❹When the device is capable of transmitting, a byte is transferred from the appropriate clist to a transmit register.

❺Finally, the byte held in the transmit register is shifted out to the serial line designated for transmission.

The following description also applies to the fread() call, with the exception that the sequence is reversed.

❺Binary information is shifted in from the serial line designated for reception and is held in a receive register.

❹When the device has received a byte, it is transferred from the receive register to the appropriate clist.

❸read() transfers bytes from the appropriate clist to a stream buffer maintained by the C standard library. Note that there are two types of input queues (clists)—the raw and canonical input queues. The raw input queue contains exactly what was received. The canonical input queue contains characters that have been transferred from the raw input queue and possibly filtered per a selected line discipline.

❷ fread() transfers bytes from a stream buffer maintained by the standard library to the buffer provided by the user. When the stream buffer becomes empty, its contents are replenished by a call to the operating system function read().

❶ fread() returns to the application program.

whether DOS programs should use binary or text mode when communicating with a device driver; and how parity, framing, and overrun errors should be detected and handled.

Getting Started

Just as you must open a disk file (with open or fopen) to modify it, you also must open a serial device, so that it can be written to, read from, or advised of new operation attributes. You can use either method shown in **Figure 2**; both level 1 and level 2 file I/O functions are provided in the standard C library. Note that in the call to fopen, "rwb" advises the I/O system not to insert CR/LF translations within the data and to pass the ^Z (1aH) character as binary data.

The advantages of level 1 are that there is no buffering overhead in the standard library and that information specific to the operating system such as locking and networking, can be communicated. Level 1 provides more control for character-at-a-time reading and writing. A disadvantage of level 1 is that only read and write are available for I/O. The process can require a costly context switch to the operating system kernel for each read or write function call.

Level 2 allows the use of fprintf, fscanf, fgets, fputs, and so on, for I/O. These are more portable between different operating system environments. Buffers can be flushed (reset) in the standard library. Context switching to the operating system for I/O is only necessary when output buffers are flushed or when input buffers are replenished, not for each call to fread or fwrite.

A problem with level 2 is that there may be too much buffering overhead. The device, operating system, and standard library all maintain buffers. Furthermore, stream buffer overhead for character-at-a-time I/O is more beneficial for block devices (disks) than character devices (serial I/O ports). See **Figure 3** for more information about I/O paths.

Despite the disadvantages of using level 2 file I/O, it is the best choice for the application discussed here because of the convenience provided by the printf and scanf functions. Also, the standard library can be convinced that the stream buffer it maintains for a file stream is of length 1 with a call to setbuf. Buffering overhead is reduced to approximately that of level 1 I/O, and fflush need not be called after each fwrite or fprintf transaction.

Once a device has been opened, it can immediately be read from or written to. For now, assume that either the device has been initialized or its default attributes will suffice. You can therefore communicate with the serial device, as in **Figure 4**.

Figure 4: Communicating with a Serial Device

```
auto   int     in_int;
auto   FILE    *fp_tty;

/* Trivial request and response from the terminal */

/* Assumes fp_tty is opened in "rwb" mode */

do
{
  fprintf( fp_tty, "\nHello Terminal. \nEnter an integer: " );

} while( fscanf( fp_tty, "%d", &in_int )  !=  1 );
```

Figure 5: The Termio Structure

```
#define  NCC  8

struct   termio
{
   unsigned  short  c_iflag;   /* Input modes.     */
   unsigned  short  c_oflag;   /* Output modes.    */
   unsigned  short  c_cflag;   /* Canonical modes. */
   unsigned  short  c_lflag;   /* Local modes.     */

   char      c_line;           /* Line discipline. */

   unsigned  char   c_cc[NCC]; /* Special chars.   */
};
```

Changing Parameters

Next, consider the mechanism for changing serial device attributes. As stated earlier, the IOCTL function is used to get or set device attributes passed in a C structure. To use the IOCTL function call, the device must be opened with the open function, and a valid file descriptor (a DOS file handle) must be obtained. If the standard library function fopen is used, the file descriptor can be derived from the file pointer with the fileno macro found in stdio.h. One C structure, termio, is used to set or get attributes for all asynchronous devices under UNIX; it is found in the include file termio.h. Members of the termio structure contain information on input modes, output modes, control modes, line disciplines, and an array of eight special control characters. The termio structure is shown in **Figure 5**.

Each flag in the termio structure is actually a collection of several flags that produce a bit pattern fitting into an unsigned short. The flags have the effects listed below.

c_iflag Instructs the device how to react to received input. Break, parity generation, parity

Figure 6: Possible Termio Flags

The c_iflag field describes the basic terminal input control.

IGNBRK	Ignore break condition
BRKINT	Signal interrupt on break
IGNPAR	Ignore characters with parity errors
PARMRK	Mark parity errors
INPCK	Enable input parity check
ISTRIP	Strip character
INLCR	Map NL to CR on input
IGNCR	Ignore CR
ICRNL	Map CR to NL on input
IUCLC	Map uppercase to lowercase on input
IXON	Enable START/STOP output control
IXANY	Enable any character to restart output
IXOFF	Enable START/STOP input control

The c_oflag field specifies the system's treatment of output.

OPOST	Post-process output	
OLCUC	Map lowercase to uppercase on output	
ONLCR	Map NL to CR-NL on output	
OCRNL	Map CR to NL on output	
ONOCR	No CR output at column 0	
ONLRET	NL performs CR function	
OFILL	Use fill characters for delay	
OFDEL	Fill character is DEL, else NUL	
NLDLY	Select New-Line delays:	
	NL0	New-Line character type 0 (no delay)
	NL1	New-Line character type 1 (0.10 second delay)
CRDLY	Select Carriage-Return delays:	
	CR0	Carriage-Return delay type 0 (no delay)
	CR1	Carriage-Return delay type 1 (column position dependent)
	CR2	Carriage-Return delay type 2 (0.10 second delay)
	CR3	Carriage-Return delay type 3 (0.15 second delay)
TABDLY	Select Horizontal-Tab delays:	
	TAB0	Horizontal-Tab delay type 0 (no delay)
	TAB1	Horizontal-Tab delay type 1 (column position dependent)
	TAB2	Horizontal-Tab delay type 2 (0.10 second delay)
	TAB3	Horizontal-Tab delay type 3 (expand tabs to spaces)
BSDLY	Select backspace delays:	
	BS0	Backspace delay type 0 (no delay)
	BS1	Backspace delay type 1 (0.05 second delay)
VTDLY	Select Vertical-Tab delays:	
	VT0	Vertical-Tab delay type 0 (no delay)
	VT1	Vertical-Tab delay type 1 (2.0 second delay)
FFDLY	Select Form-Feed delays:	
	FF0	Form-Feed delay type 0 (no delay)
	FF1	Form-Feed delay type 1 (2.0 second delay)

The c_cflag field describes the hardware control of the terminal.

CBAUD	Baud (Bit) rate:	
	B0	Hang up
	B50	50 bps (baud)
	B75	75 bps (baud)
	B110	110 bps (baud)
	B134	134.5 bps (baud)
	B150	150 bps (baud)
	B200	200 bps (baud)
	B300	300 bps (baud)
	B600	600 bps (baud)

CONTINUED

checking, bit stripping, carriage return to new line mapping, uppercase to lowercase mapping, and XON/XOFF software handshaking can be set.

c_oflag Instructs the device how to process output. Mapping of case, new line translation, fill characters, and delays for CR, HT, NL, and BS can be set.

c_cflag Allows the setting of speed (bps), character size, stop bits, parity options, and control of data terminal ready (DTR), request to send (RTS), and data carrier detect (DCD) RS-232 control signals.

c_lflag Enables exception handling, input-queue processing, echoing, line editing, buffer management, and mapping of special characters.

c_line Selects a line discipline; usually set to 0. The line discipline selects a mode of mapping (filtering) characters from a raw input queue to a canonical input queue (this will be explained later). This filter mechanism allows line editing, conversion of lowercase to uppercase, expansion of tabs, and so on. Most UNIX systems support only one well-established line discipline, but other disciplines have been implemented (for example, for handling synchronous data links supporting SDLC types of communications).

c_cc[NCC] An array of eight characters that can be assigned to represent the interrupt and quit signals (such as ^C, DEL, and BREAK), the erase character, EOF, and EOL. Some of the c_cc[] positions have dual meanings that depend on other flags, specifically the c_cc[4] (EOF, MIN) and c_cc[5] (EOL, TIME) positions.

All of the flags of the termio structure are set by using the bitwise AND(&) and OR(|) operators with the constants provided in the termio.h include

file. **Figure 6** describes many of the possible flags. They are fully documented in the manual pages under termio(M) (Xenix) and termio(7) UNIX System V and by vendors of equivalent drivers for DOS.

The fragment of code shown in Figure 7 demonstrates how to open and obtain the current attributes for the device /dev/tty01. Flags are assembled to change the speed to 9600 bps, to change the number of data bits to 7, and to enable reading.

Once the flags are configured, a call to IOCTL is made to set the attributes. Set all values of the termio structure to valid values so the driver will not configure randomly. Also check return codes from library calls (discussed later).

Data Acquisition

Computer-to-terminal communications is a simple, general, and easily tested form of data acquisition. The program in **Figure 8** will send a request message to an attached I/O device and expect a 10-byte response back within 2 seconds. If the expected response is received in time, data received back from the device is written to a log file: otherwise error recovery is started. Most, but not all, data acquisition devices request responses in a similar way. If an unsolicited response arrives, it will be buffered for the next read and is not lost. Selected attributes for the device, which are given below, are stored in the termio structure.

c_iflag IGNBRK advises the driver to ignore the break sequence completely if the break signal is received. The driver could otherwise cause an interrupt signal to be sent to the application. IGNPAR states that any received characters containing detected parity errors are ignored. IXON enables software flow control on the output

CONTINUED

Figure 6

	B1200	1200 bps (baud)
	B1800	1800 bps (baud)
	B2400	2400 bps (baud)
	B4800	4800 bps (baud)
	B9600	9600 bps (baud)
	B19200	19200 bps (baud)
	B38400	38400 bps (baud)
CSIZE	Select character size for both transmission and reception:	
	CS5	5 bits
	CS6	6 bits
	CS7	7 bits
	CS8	8 bits
CSTOPB	Send two stop bits, else one	
CREAD	Enable receiver	
PARENB	Parity enable	
PARODD	Odd parity, else even	
HUPCL	Hang up on last close	
CLOCAL	Local line, else dial-up (modem)	
LOBLK	Block layered output	

The c_lflag field of the argument structure is used by the line discipline to control terminal functions. The basic line discipline (0) provides the following:

ISIG	Enable signals
ICANON	Canonical input (erase and kill processing)
XCASE	Select canonical uppercase/lowercase presentations
ECHO	Enable echo
ECHOE	Echo erase character as BS-SP-BS
ECHOK	Echo NL after kill character
ECHONL	Echo NL
NOFLSH	Disable flush after interrupt or quit

The c_cc[NCC] array contains characters that are mapped to special actions. As characters are received by the device, they can be checked against the characters stored in the c_cc[] array. This checking is enabled by setting combinations of the ICANON, ISIG, and IGNBRK flags. If a match is found, a special action can be performed by the device driver. Characters representing backspace, EOF, EOL, MIN, TIME, and assorted signals are stored here.

c_cc[0]	(VINTR)	If matched, an interrupt signal is sent to the controlling process.
c_cc[1]	(VQUIT)	If matched, a quit signal is sent to the controlling process.
c_cc[2]	(VERASE)	If matched, the last enqueued character received is deleted.
c_cc[3]	(VKILL)	If matched, the current line buffer is flushed and the line is killed.
c_cc[4]	(VEOF/MIN)	If matched, the device driver assumes that the end of input (EOF) has been reached. Any characters pending in the input line buffer are immediately passed to the application. This also serves as the minimum number of characters to collect in raw input mode when ICANON flag is not set.
c_cc[5]	(VEOL/TIME)	If matched, the device driver assumes that an entire line has been assembled; the line is now available for reading by the application. This also serves as a timeout parameter if the ICANON flag is not set.
c_cc[6]	(Reserved)	Not used.
c_cc[7]	(SWTCH)	If matched, the current foreground process is moved to the background. This is the job control switch character (UNIX only).

Figure 7: Code Fragment to Obtain Attributes for a Device

```
#include <stdio.h>
#include <termio.h>

static FILE    *tty_fp;
static struct termio  tty_params;


   /* Open the device    */

  tty_fp = fopen("/dev/tty01", "rwb");

  /* Obtain its current attributes; stored at &tty_params */

 ioctl(fileno(tty_fp), TCGETS, &tty_params);

  /* Change desired attributes; stored at &tty_params     */

  tty_params.c_cflag = B9600 | CS7 | CREAD;
                       :
                       :
  /* Set new attributes */

 ioctl(fileno(tty_fp), TCSETS, &tty_params);
```

stream; IXOFF enables the same on the input stream.

c_oflag No flags are selected and all features represented by this flag are disabled.

c_cflag B4800 selects the bit rate of both the transmit and receive channels. CS8 advises that 8 data bits be sent and received. CREAD enables the read system call to transfer characters from the device driver's buffer. HUPCL drops the DTR signal on the last close of the device (the device can be opened more than once). CLOCAL advises the device not to examine the DCD signal as a qualifier when the driver opens the device.

c_lflag No flags are selected, and all features represented by this flag are disabled.

c_line The default line discipline is 0.

c_cc[0] (VINTR) Not used since the ICANON | ISIG flags are not set.

c_cc[1] (VQUIT) Not used since the ICANON | ISIG flags are not set.

c_cc[2] (VERASE) Not used since the ICANON | ISIG flags are not set.

c_cc[3] (VKILL) Not used since the ICANON | ISIG flags are not set.

c_cc[4] (VEOF/MIN) Since the ICANON flag is not set, c_cc[4] contains the minimum number of characters that must be collected by the driver before a read call is satisfied. Otherwise it would represent the EOF character. c_cc[4] can be used in tandem with c_cc[5].

c_cc[5] (VEOL/TIME) Since the ICANON flag is not set, the c_cc[5] attribute contains the number of 0.10 second increments the driver should wait for before returning from a read. Otherwise it would represent the EOL character. c_cc[5] is a watchdog that will wait until the requested (or minimum) number of characters is available or the timer has expired, whichever comes first. This feature allows application programmers to specify the amount of time to wait for an attached serial device to send something (for example, after a query has been sent to it). c_cc[4] and c_cc[5] let you specify the minimum number of characters that must be collected and/or the maximum amount of time to wait for the characters; no timing loops are required in the application.

c_cc[6] Reserved.

c_cc[7] (SWTCH) Not used.

Note that the stream buffers for the tty_stdin and tty_stdout have been set to length 1. This example can be adapted to work with almost any I/O device that is request/response driven. As characters arrive, they are buffered until the next read call is satisfied. tx_block and rx_block are simple character arrays; they could be structures containing more detailed information about what is sent and received.

Also note that the path name of the device is /dev/tty01. This is a standard pathname to a UNIX device driver residing in the /dev directory. Curiously enough, /dev/tty01 is also a valid path under DOS to the device tty01. The Microsoft C run-time library translates the forward slash (/) to the back slash (\ ) in file or device pathnames on calls to file system routines. However, \dev\tty01 must be used at the DOS command line (batch file) level.

Input and Output Queues

Once the serial device attributes have been selected and set and the device has been opened, I/O may begin. Three queues are maintained by the UNIX serial device driver—the output queue, which can be written to, and the raw and canonical queues, which can be read. (Do not confuse these queues with stdin, stdout, and stderr.)

Characters that are written to the device are copied to the output queue for transmission. Depending on flag settings, delays and/or translations may be performed as characters are transmitted. **Figure 9** shows the relationship of the raw and canonical input queues. An application can choose to read characters from either. The raw input queue is simple: anything received (excluding parity errors and break sequences) is buffered and passed directly to the application. When reading

from the raw queue, it is possible to configure the driver to implement a watchdog timer or specify a minimum number of characters that must be available before a read completes successfully. The size of the raw queue is limited by a shared pool of buffers; these buffers, called clists under UNIX, are dynamically allocated and released. There is no pool of clists under DOS; buffers are usually fixed, because the device driver is accessed through the DOS file system. Remember that DOS is not reentrant, so it is nearly impossible to perform dynamic memory allocation of clists inside any DOS device driver.

The canonical input queue obtains its characters from the raw queue via a line discipline. The line discipline is analogous to a filter; as characters are obtained from the raw queue, they may be translated to other sequences. For example, CR can translate to NL (ICRNL flag). The canonical input queue is enabled with the ICANON flag, which also advises the device driver to look for the special characters in the c_cc[] array and take appropriate action in accordance with the IBRK and ISIG flags.

Interactive Terminals

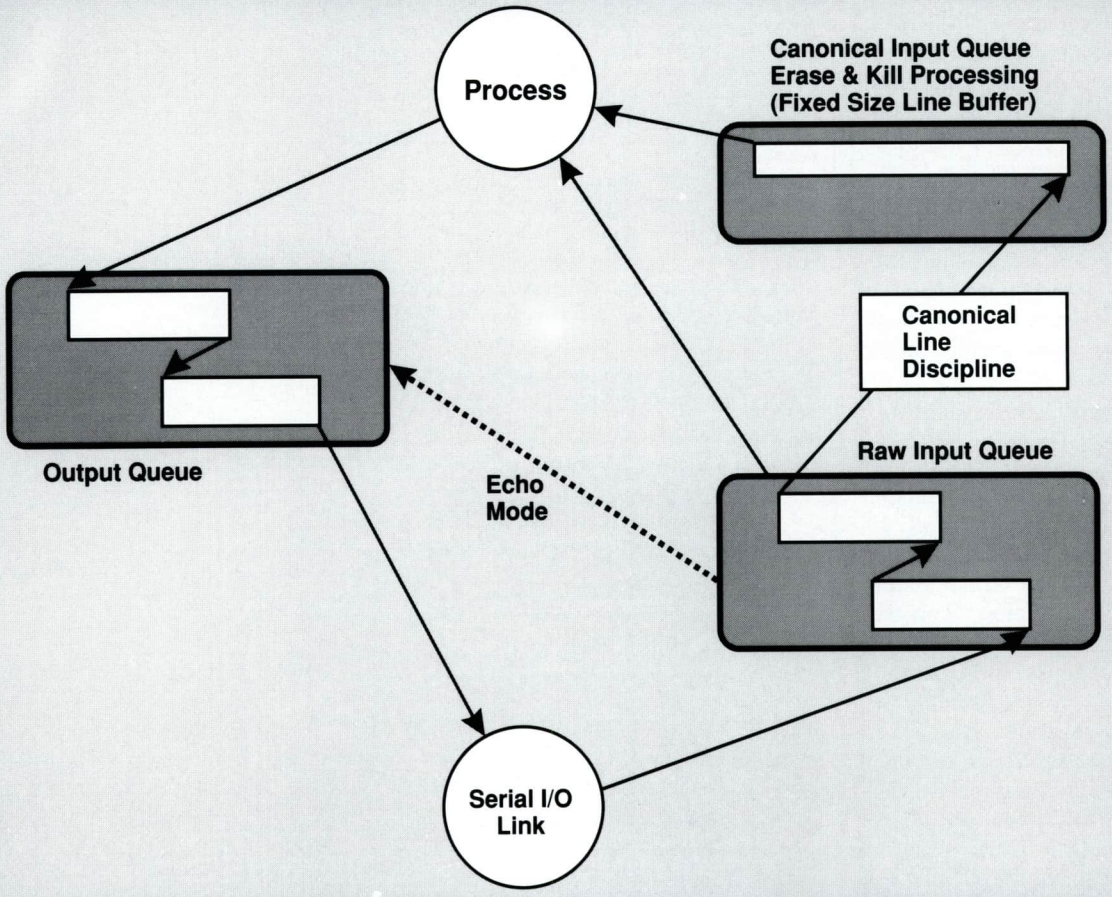
Interactive programming requires a special style of coding. People are much less predictable than machines, so special care must be taken when prompting and handling responses (valid or invalid). This example assumes that there is a dumb terminal or PC connected to the serial port, either directly or through a modem.

The program in **Figure 10** is trivial if input is from and output is to the DOS console device. During an interactive session at a remote terminal on a serial link, however, special precautions must be taken. When the

Figure 8: Program to Send a Request Message to an Attached I/O Device

```
#include  <stdio.h>
#include  <termio.h>  /* Installed with .h files */

#define   TTY_STDIN_FILE_NAME    "/dev/tty01"
#define   TTY_STDOUT_FILE_NAME   "/dev/tty01"
#define   LOG_FILE_NAME          "logfile"

static    char    tx_block[10] = "SEND.DATA";
static    char    rx_block[10];
extern    int     errno;


main()
{
   static    struct   termio termio_var;
   auto      FILE     *tty_stdin, *tty_stdout, *log_file;
   auto      int      error_flag, i;

  if((tty_stdin = fopen(TTY_STDIN_FILE_NAME, "rb")) ==
          (FILE *)NULL)
      exit(1);

  if((tty_stdout = fopen(TTY_STDOUT_FILE_NAME, "wb")) ==
          (FILE *)NULL)
      exit(2);

  if((log_file = fopen(LOG_FILE_NAME, "wb")) == (FILE *)NULL)
      exit(3);

  /* Length of tty_stdin & tty_stdout stream buffers to 1 */

  setbuf(tty_stdin, (char *)NULL);
  setbuf(tty_stdout, (char *) NULL);

  if(ioctl(fileno(tty_stdin), TCGETS, &termio_var))
      exit(4);

  /* Set up the terminal attributes */

    termio_var.c_iflag = IGNBRK | IGNPAR | IXON | IXOFF;
    termio_var.c_oflag = 0;
    termio_var.c_cflag = B4800 | CS8 | CREAD | HUPCL | CLOCAL;
    termio_var.c_lflag = 0;
    termio_var.c_line = 0;
    termio_var.c_cc[VEOF] = sizeof( rx_block ); /* MIN */
    termio_var.c_cc[VEOL] = 20;                 /* TIME */

  if(ioctl(fileno(tty_stdout), TCSETS, &termio_var))
      exit(5);
   else
   {
      for(i = 0; i < 100; i++)
      {
        if(fwrite(tx_block, sizeof(tx_block), 1, tty_stdout) != 1)
             fprintf(stderr,
                    "Can't write to I/O device on tty_stdout\n");

        if(fread(rx_block, sizeof(rx_block), 1, tty_stdin) == 1)
          {
             fprintf(stdout,
               "Valid response from device received; Logging...");

        if(fwrite(rx_block,sizeof(rx_block), 1, log_file) == 1)
             fprintf(stdout, "Passed.\n");
          else
             fprintf(stdout, "Failed.\n");
                       }
                      else
             fprintf(stderr,
                "No response from device; errno: %d\n", errno );
      }
     fclose(tty_stdin);
     fclose(tty_stdout);
     fclose(log_file);
   }
  return(0);
}
```

Figure 9: Raw and Canonical Input Queues



This figure shows the input and output queues maintained by a typical UNIX character device driver. As characters are written by the process, they are placed on the output queue and are eventually sent by the serial link hardware. As characters are received by the device driver, they are placed on the raw input queue and may be read directly, without modification, by the process. The process can also elect to read from the canonical input queue (the ICANON flag must be set). As characters are received and placed in the raw queue, they are moved to the canonical input queue via the selected line discipline. The line discipline acts as a filter; removing, replacing, or adding characters on the input stream. A common replacement is translating CR with CR-NL. If the canonical input queue is enabled, line editing is possible.

terminal attached to the serial link is prompted for an integer, there are three possible responses: correct, incorrect, or no response. The enclosed switch statement handles these three cases. This example sets up a terminal with many of the features that a shell might use for command line interpretation, notably line editing, character echoing, and abort sequences.

Selected attributes for the device, given below, are stored in the termio structure. Note that the buffers for the streams tty_stdin and tty_stdout have been set to length 1.

c_ifla IGNBRK advises the driver to ignore the break sequence completely if the break sequence is received; otherwise, the driver could cause an interrupt signal to be sent to the application. IGNPAR states that any received characters containing detected parity errors are ignored. IXON enables software flow control on the output stream. IXOFF enables software flow control on the input stream. CLOCAL disables modem control; DCD is ignored when opening the device. ICRNL specifies that the CR character is converted to the LF ('\n') character on input.

c_oflag OPOST enables the post-processing of output. ONLCR converts the '\n' (LF) character to the CR/LF characters on output. OCRNL converts the CR character to the LF ('\n') character on output. TAB3 expands the horizontal tab char-

acter to spaces up to the next tab stop. Tab stops usually occur every eight characters.

c_cflag B4800 selects the bit rate of both the transmit and receive channels. CS8 advises that eight data bits are sent and received. CREAD enables the read system call to transfer characters from the device driver's buffer. HUPCL drops the DTR signal when the device is closed for the last time. CLOCAL advises the device not to examine the DCD signal as a qualifier in opening the device.

c_lflag ICANON enables the processing of character filtering from the raw input queue to the canonical input queue. Line editing and character echoing is now performed by the device driver. The watchdog for a timer and/or a minimum number of characters seen in the last example is no longer active. c_cc[4] and c_cc[5] now contain characters representing EOF and EOL, respectively. ISIG tells the driver to check each input character against the special characters designated in the c_cc[] array for INTR, QUIT, and SWTCH. If any of these characters are found in input, the appropriate action is taken (that is, signals are sent). ECHO echoes all characters received to the output queue. ECHOE echoes the designed erase character as BS-SP-BS, thus erasing the character in error.

c_line Selects a line discipline; usually set to 0.

c_cc[0] (VINTR) If matched on input, the interrupt signal is sent to the controlling process. Under DOS, the current process is killed. Note that the ICANON | ISIG flags are set in c_cflag.

c_cc[1] (VQUIT) If matched on input, the quit signal is sent to the controlling process.

c_cc[2] (VERASE) If matched on input, the last valid character entered is erased.

c_cc[3] (VKILL) If matched on input, whatever has been assembled in the current line is erased.

c_cc[4] (VEOF/MIN) This is the designated EOF character, typically ^Z (DOS) or ^D (UNIX), although any character will work.

c_cc[5] (VEOL/TIME) represents the EOL character; typically '\n'; any character will work, however. This character is the anchor used by the line discipline routines to mark the end of a line. Typically lines are assembled and edited by the user. When the EOL character (stored in c_cc[5]) is received, the line is available for reading by the application.

c_cc[6] Reserved.

c_cc[7] (SWTCH) Not used.

Serial Link

The examples of data acquisition and request/response communications discussed in **Figures 8** and **10** demonstrated important and useful features when using the UNIX (or compatible) serial I/O device driver. All I/O was performed with familiar calls to the C standard library. Many useful features of the UNIX serial I/O device driver can save the C programmer many headaches—features such as watchdog timeouts, buffering in both the TX and RX directions, CR and LF translation on both input and output streams, line editing, tab expansion, and XON/XOFF software handshaking. One common interface, IOCTL, is used to adjust device parameters using a standard system call. Serial I/O software is portable between UNIX and DOS systems.

There are other modes of serial link communications and other programming contexts, but they are usually variations of one of the examples presented. Advanced techniques of serial communication programming might address any of the following topics.

Advanced Modem Control If the CLOCAL flag is cleared, the device will wait for the DCD line to become active for an open call to complete. This allows the application to block for a modem connection before the device driver opens. UNIX programmers can set the O_NDELAY flag when performing an open operation. The open function will then use errno to return immediately, indicating success (DCD present) or failure (DCD absent) in opening the device.

Reacting to Interrupt Signals Under UNIX, the reception of DEL or ^\ (changeable defaults in the c_cc[ ] array) sends the signals interrupt or quit, respectively. Reception of a BREAK sequence will also send the interrupt signal. If the flags ICANON | ISIG are set, a signal is sent to the controlling process. Under DOS, some device drivers provide an option to kill the currently executing process by issuing DOS interrupt 4BH.

Sending the BREAK Character Calls can be made to IOCTL to send the BREAK character. See the IOCTL command TCSBRK.

Forcing the Driver XON/XOFF State Calls can be made to IOCTL to suspend output and restart suspended output. See the IOCTL command TCXON.

Handling Parity Errors If parity checking is enabled by setting the PARENB and PARMRK flags and clearing the IGNPAR flag, a character (X) received with a parity error is passed as a special sequence ('\x7F', '\0', X) to identify the character in error. The character

Figure 10: Setting Up the Terminal with Shell Features

```
#include  <stdio.h>
#include  <termio.h>  /* Installed with .h files */

#define   TTY_STDIN_FILE_NAME    "/dev/tty01"
#define   TTY_STDOUT_FILE_NAME   "/dev/tty01"

extern    int  errno;


main()
{
   static  struct  termio  termio_var;
   auto    FILE    *tty_stdin, *tty_stdout;
   auto    int     error_flag, i_var, rc;

  if((tty_stdin = fopen(TTY_STDIN_FILE_NAME, "rb")) == (FILE *)NULL)
       exit(1);

  if((tty_stdout = fopen(TTY_STDOUT_FILE_NAME, "wb")) ==
                        (FILE *) NULL)
       exit(2);

   setbuf(tty_stdin, (char *) NULL);
   setbuf(tty_stdout, (char *) NULL);

  if(ioctl(fileno(tty_stdin), TCGETS, &termio_var))
       exit(3);

   termio_var.c_iflag =   IGNBRK | IGNPAR | IXON |
                          IXOFF | CLOCAL | ICRNL;
   termio_var.c_oflag =   OPOST | ONLCR | OCRNL | TAB3;
   termio_var.c_cflag =   B4800 | CS8 | CREAD | HUPCL | CLOCAL;
   termio_var.c_lflag =   ICANON | ISIG | ECHO | ECHOE;
   termio_var.c_cc[VINTR] = '\177';        /* DEL char         */
   termio_var.c_cc[VQUIT] = '\003';        /* ^^C char         */
   termio_var.c_cc[VERASE] = '\b';         /* backspace        */
   termio_var.c_cc[VKILL] = '@';           /* @ kills line     */
   termio_var.c_cc[VEOF] = '\004';         /* Designated EOF   */
   termio_var.c_cc[VEOL] = '\n';           /* Designated EOL   */

  if(ioctl(fileno(tty_stdout), TCSETS, &termio_var))
       exit(5);
   else
   {
      fprintf(tty_stdout, "\nEnter an Integer > ");

       while(1)
       {
          switch(rc = fscanf( tty_stdin, "%d", &i_var))
            {
                case EOF:
                    break;

                case 0:
                  fprintf(stderr, "\nInvalid response\n\n");
                  fflush(tty_stdin);
                  fprintf(tty_stdout, "\nInvalid response\n\n");
                  fprintf(tty_stdout, "\nEnter an Integer > ");
                  break;

                case 1:
                  fprintf(stdout, "From tty: %d \n\n", i_var);
                  fprintf(tty_stdout, "\nEnter an Integer > ");
                  break;

                default:
                  fprintf(stderr, "fscanf() : %d ?\n", rc);
                  break;
            }
       }
       fclose(tty_stdin);
       fclose(tty_stdout);
   }
   return(0);
}
```

'\x7F' is then read as '\x7F','\x7F'.

Recovering from DOS Critical Errors DOS will translate critical errors to interrupt vector 24H. If the application traps this interrupt, all critical errors encountered by a device driver should trap the user's exception handler with sufficient information (device name and type of critical error) to recover.

Command Line Interface

As mentioned earlier, an advantage to performing serial I/O with a device driver is having access through the file system. Command line redirection (and batch/shell files) can communicate with the device. For example, under DOS the command

```
C>dir > \dev\tty01
```

sends a directory listing to a serial device. Note that no mention has been made of how to change device attributes from the command line. This is accomplished with the UNIX command stty. The stty command is also capable of obtaining the current attributes of a device and displaying them. The same command is available with DOS serial device drivers; however, DOS requires the device path to be \dev\tty01. Refer to **Figure 11** for an example of stty under both DOS and UNIX.

In the examples in **Figure 11**, the stty program reopens its standard input, the serial device /dev/tty01, and sets its attributes to 9600 bps, even parity enabled, 7 data bits per character, XON/XOFF software handshaking enabled, and modem control enabled. This configuration allows communication with a popular laser printer over the serial link. In the two commands shown in **Figure 12**, file_1 is sent through a PostScript® preparation filter and finally to a printer attached to /dev/tty01.

Figure 11: Using stty under DOS and UNIX

Under UNIX

```
$stty 9600 parenb -parodd cs7 ixon ixoff -clocal hupcl < /dev/tty01
```

Under DOS

```
C>stty 9600 parenb -parodd cs7 ixon ixoff -clocal hupcl < \dev\tty01
```

Figure 12: Communicating with a Laser Printer over a Serial Link

Under UNIX

```
$cat file_1 | PS_filter > /dev/tty01
```

Under DOS

```
C>type file_1 | PS_filter > \dev\tty01
```

System Interface

There are some intrinsic features of the operating system that concern both UNIX and DOS programmers that you should be aware of when opening a serial device for I/O. The first concern is how the blocking (or process suspension) is handled. Under DOS, there is no facility to suspend a process on the assumption that it will resume running when some event has occurred (or has not occurred). Under UNIX, a terminal driver can cause the suspension of a process if there are no characters to read or a line has not yet been completely assembled via a selected canonical line discipline. Once characters are ready or a line has been assembled, a suspended (or blocked) process is awakened to run. Since DOS has no facility that suspends a process to wait for I/O, any request to a device driver should return immediately (after checking for set watchdog timers), whether characters (or lines) are ready or not. If nothing is available, EOF can be returned.

The second concern is that there are differences in the treatment of EOF indications. DOS supports two access modes for reading and writing character devices—ASCII text mode and binary mode. If the character 1aH (^Z) is read, the input from a file or device opened in text mode is terminated (shut down) and an EOF indication is returned. Even if there are characters pending, any and all read requests from this point forward are ignored and the EOF indication is always returned. If the file were opened in binary mode, the EOF character (^Z) would be passed to the application, and further requests for characters would be honored.

Text mode also insists on translating LF to CR/LF on output and CR/LF to LF on input, so I/O in text mode could add characters or delete them. Binary mode reads and writes exactly what is requested. These features are intrinsic to DOS.

DOS device drivers should return EOF if there is nothing to read (after checking for set watchdog timers). A device driver could block the process attempting the read; the application might then hang forever waiting for a character. If EOF is returned, the application can do something else and attempt another read later. Anything that arrives between reads will be buffered.

When the EOF character, designated by c_cc[4], is received by a UNIX character device driver, all characters waiting to be read are immediately passed to the application without waiting for a new line, and the EOF is discarded. Thus if no characters are waiting in the input queue—that is, if EOF occurred on the beginning of a line—no characters will be passed back to the application; this is a standard EOF indication under UNIX.

The third concern is the difference in the treatment of signals. The signal mechanism available under the UNIX operating system is not available under the DOS operating system, although most C run-time libraries, most notably Microsoft C 5.10, have mechanisms that perform similar functions.

The signal( ) function call allows the UNIX and DOS programmer to catch special messages sent to processes when exceptions such as floating point overflow, ^C, and modem disconnect occur. Async device drivers under UNIX are also able to send signals to related processes. These signals are typically INTR, QUIT, HANGUP, and BREAK. DOS does not provide the signal mechanism that UNIX does, although DOS drivers can be designed to have similar functions.

Conclusion

This article explored how to write portable serial I/O software, useful features of a general serial I/O device interface operating under both UNIX and DOS. UNIX and its many hybrids all share the same serial I/O interface. With AT&T, Digital Equipment, Hewlett-Packard, IBM, Microsoft, and other companies, embracing UNIX as a standard product offering, it simply cannot be ignored by the professional programmer. ❑

[1] As used herein, "DOS" refers to the MS-DOS and PC-DOS operating systems.

Suggested Readings

For a thorough discussion of PC serial port hardware: Greenberg, R.M., "Keeping Up With the Real World: Speedy Serial I/O Processing," *MSJ* (Vol. 2, No. 3), pp. 37–50.

For an in-depth discussion of the internals of an MS-DOS device driver: Greenberg, R.M., "A Strategy for Building and Debugging Your First MS-DOS Device Driver," *MSJ* (Vol. 2, No. 5), pp. 51-65.

For an excellent reference on UNIX device driver internals: Eagan, J. I., and Teixeira, T. J., *Writing a UNIX Device Driver* (John Wiley, 1988).

We think programmers need more drive.

Get the tools you need to enter the CD-ROM age. Microsoft Programmer's Library Software plus a Denon CD-ROM Drive—for under $1,000.

Call Microsoft direct at (800) 227-4679 now for complete details on the bundled bargain of the year: Microsoft *Programmer's Library* with 72 reference manuals and guides—over 278 MB of data on a single compact disc. Plus 3-user OPTI-NET™ software, including 2 additional user licenses (a $130 savings). And Denon's SCSI CD-ROM drive system, complete with interface card and cables... all for just $949.

> *"If you do much programming and your time is worth anything at all, get a CD-ROM reader and Microsoft's Programmer's Library. You won't regret it. Highly recommended."*
>
> —*Jerry Pournelle,* Byte Magazine, *Sept. 1989*

With accelerating hardware and software development by leading companies like IBM, Intel, Lotus and Microsoft, CD-ROM is fast becoming a standard peripheral of the 90's. Utilizing CD-ROM technology, Microsoft Programmer's Library helps you work faster...and smarter. Over 100,000 hypertext links streamline your search time, giving you instant access to all documentation on networks, hardware, OS/2, Windows™ and MS-DOS® environments, C, Macro Assembler, Pascal, FORTRAN and BASIC. And the latest versions of 72 essential reference texts, with regular updates available from Microsoft. You'll also have 26 MB of indexed code you can actually cut-and-paste right into your text editor.

The Denon CD-ROM drive—with its advanced servo-circuitry, high reliability, multimode flexibility and access time of 400 ms.—dramatically cuts manual search time and downtime. And when you really want to switch modes, the built-in speaker and jacks let you play your favorite audio CDs. So you can *listen* to a keyboard for a change. But if you just can't stop programming, Microsoft audio software turns your CD drive into a programmable audio CD player. Order now and this special music program (a $99 value) is *yours free*... while supplies last.

15-Day Money-Back Guarantee. Order direct from Microsoft today and get Microsoft Programmer's Library complete with the Denon CD-ROM Drive (IBM® AT® version) and Microsoft's audio program for just $949.

Call now and save $799!
(800) 227-4679

Microsoft®
Making it all make sense™

Simplifying Pointer Syntax for Clearer, More Accurate Programming

35

Greg Comeau

Pointers containing the address of functions, variables, or objects clarify and simplify your code. They are tricky, though, and confusing if carelessly written. The derivations of pointer syntax that are presented in this article should make pointer usage perfectly clear to you. While these derivations are simple, some of them may be new to you or may produce results you would not have predicted. For example, the output of the program in **Figure 1** may not be obvious, although it is a common and basic C construct.

An analysis of **Figures 2A** and **2B** will give us a common frame of reference for this article. In these figures, the value of c is 'a', and the value of p when p = &c executes is 1000. The value 1000 is the address of c (randomly chosen for this discussion), and the & signifies that we want to take the address of the accompanying object operand c.

Therefore p = &c implies that the data contained in p is actually the address of another variable, which also contains its own data. This seems logical, since c and p each have their own storage space and can hold independent data within that storage space. It is easy to forget this when your declarations and programs become more complex.

Pointer Uses

Basically, you can do three things with pointers: assign another value to the pointer, assign the pointer's value to another pointer explicitly by means of an assignment or implicitly as an argument to a function, or dereference the pointer.

Assigning another value to the pointer is as simple as coding the assignment (see **Figure 3A**). The memory map of the program as it executes will help you understand exactly how this works (see **Figure 3B**). Notice how the reassignment to p simply causes p to change values; this is the normal functioning of a variable and applies even though we are dealing with a pointer. The reassignment of a pointer, however, can have devastating effects on some programs. This is especially true with heap space when free subroutine calls are not issued between the assignments.

Using the pointer in an expression such as an rvalue shows once again that a pointer is just another variable. In **Figures 4A** and **4B**, the statement p = &c assigns the address of c to p. Then the statement p2 = p assigns to p2 the address of c by using the value contained in

Greg Comeau is a principal of Comeau Computing, an independent software development firm specializing in UNIX® and C productivity tools. He also does consulting and training for UNIX and C users.

BASICALLY, THERE ARE THREE THINGS YOU CAN DO WITH POINTERS: ASSIGN ANOTHER VALUE TO THE POINTER, ASSIGN THE POINTER'S VALUE TO ANOTHER POINTER EXPLICITLY BY MEANS OF AN ASSIGNMENT OR IMPLICITLY AS AN ARGUMENT TO A FUNCTION, OR DEREFERENCE THE POINTER.

Figure 1: Simple C Construct

```
#include <stdio.h>

main()
{
    char    *p = "abcdefg";

   printf("%c\n", *(p++));
}
```

p. This does not imply that p2 points at p. Instead the value of p gets copied into p2. This is no different from

```
int   i;
int    j;
i = 5;
j = i;
```

where we know that j will also contain its own copy of the value 5. To prove this, run the code from **Figure 4A** to see what the output will be. Note that it will not print out numbers, like 1000, used in the memory maps. Instead the %p format specifier of printf will encode the output in segment:offset format.

The last use of pointers involves the value that the pointer contains through the unary * operator. The unary * operator is one way to accomplish indirection and dereferencing of pointers. For the moment, let's take a look at the last line of **Figure 5A.** As you can see from the memory map, the result of this statement is to change the value of c from 'a' to 'A.' Since p2 becomes another name for c, we can code

```
*p2 = 'A';
```

instead of

```
c = 'A';
```

The indirection ability of pointers is very powerful. It allows us to access named symbolic variables and unnamed variables (identifiers) with the same ease—without requiring that you keep a lot of information about which objects you are pointing to, where they are located, and so on. This indirection can result in faster and smaller code (depending upon the compiler and hardware in question).

Unary * Operator

Near the end of **Figure 5A**, c is a char object whose value is 'a' and p2 is a pointer to a char object whose value is the address of c. *p2 represents what p2 points to, not the contents of p2 as it may appear to read. For example, using i and j from the previous example, we know that j = i says j is equal to the contents of i. There is no reason to expect pointer notation to function differently. For example, **Figure 5A** contains the statement

```
p2 = p;
```

which requests that the contents of p be placed into p2.

To explain further why *p2 does not mean "the contents of p2," it is worth noting that when we say j = i, it actually means j = *&i. Read *&i as "a pointer to the contents of the address of i." A memory map of **Figure 5A** is shown in **Figure 5B.**

Besides mapping the contents of what p2 points to, *p2 can represent the contents of the address of p2. Since in this case the content of p2 is 1000, this expression implicitly becomes * char *1000—meaning the contents of memory location 1000, taken to be a char, which is 'A'.

Derivations

Now that we've gotten some of the background out of the way, we can analyze some concrete pointer examples. PTRINC.C (see **Figure 6**) shows a small group of simple pointer operations. These pointer operations illustrate that there is more going on behind the scenes than is apparent at first glance. The following discussion is based on a debugging technique described in the sidebar "In-Line C Program Debugging." For further information, refer to "Pointers 101: Understanding and Using Pointers in the C Language," *MSJ* (Vol. 4, No. 4).

The debug.h file explained in the sidebar is included in the first line of PTRINC. Examining lines 9 and 10 you will find the declarations char *p and char *origp, which are two pointers to char objects.

On line 12, p is assigned to the address of a string literal—a group of characters next to each other in a source program, prefixed and suffixed by the double-quote character. Internally, this kind of construct is a static array of characters containing each character that is enclosed in the quotes, including a terminating null byte which is added by the compiler. This format is known as ASCIIZ.

A string literal is a good example of an unnamed area of storage. To use a string literal, we must either set a pointer to its address as PTRINC has done or use it as an argument to a function where its address becomes apparent. Most of us have used this context in our first C program by specifying a string literal as an argument to printf.

For a complete explanation of string literals, especially those involving multibyte characters, refer to the ANSI C draft (the latest version, published December 7, 1988, is available from the Computer and Business Equipment Manufacturers Association in Washington, D.C.) or a relatively new C text such as *The C Programming Language*, Second Edition, Kernighan and Ritchie (Prentice-Hall Inc., 1988).

Line 13 of PTRINC prints the characters 'a' through k in our

debug format. Line 14 assigns origp to array—the same place that p is pointing to. For a visual representation of the statements in lines 12 to 32, see **Figure 7.**

At line 17, the value printed for the contents of p is 1234, which is the same as &array[0]. Note that line 17 does not print the address of p. If we wanted that, we would have coded printp(&p) to produce an output of &p = 1200. The address of p is a given, in accordance with the top of **Figure 7.** An arbitrary location for it has been chosen since it will not affect this discussion (actual execution of the code will give another result). Therefore, like any occurrence of text which only mentions p, printp(p) produces the contents of p; origp = p functions similarly. The value printed by the execution of line 17 should point at the beginning of the array since we have not incremented or reassigned a value to p.

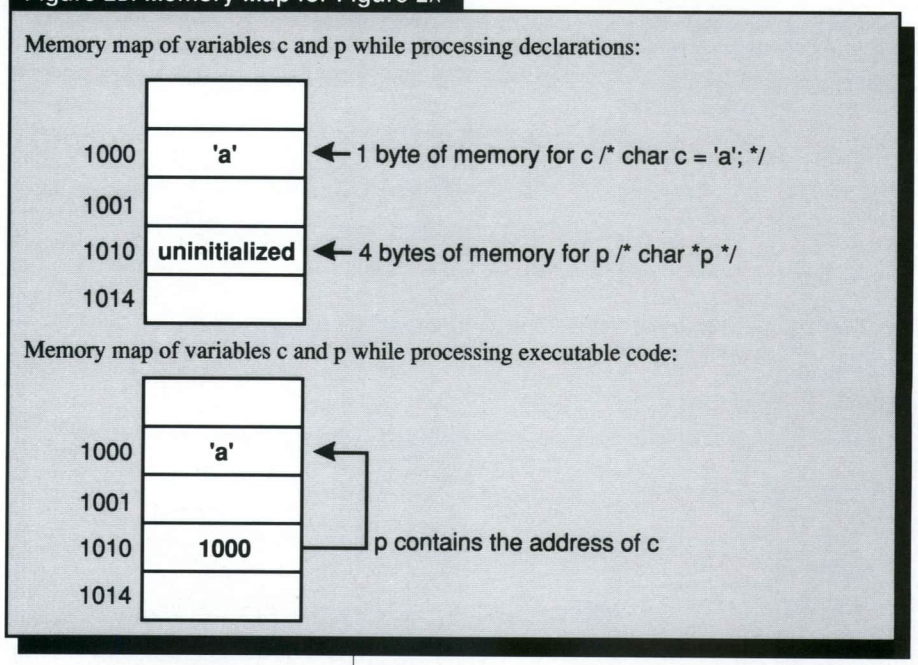
Line 18 increments p. As with all pointers, the increment value is contingent upon the derived or base type of the pointer. In PTRINC, p is a pointer to char, therefore the increment will be 1, the sizeof(char).

If PTRINC had been written so that p was a pointer to an integer (an int *), p++ would have a scaling factor of sizeof(int), which typically evaluates to 2 or 4 bytes depending upon the compiler. In C, scaling factors allow pointers to be manipulated without forcing the programmer to worry about adjusting the given pointer by hand. The pointers, however, must be of the proper type. This can cause a problem because the C language does not let you create pointers to an object with variable size. The objects may be created in the heap via the various alloc routines. This concept will be covered in a future article. Note that such objects do not need to be a linked list and that there are useful cases in which a method for obtaining the object's length after the allocation is unnecessary (dynamic copies of ASCIIZ strings, for example).

Figure 2A: A Variable Containing the Address of Another Variable

```
main()
{
    char    c = 'a';
    /* .... */
    char    *p;

    /* .... */
    p = &c;
    /* .... */
}
```

Figure 2B: Memory Map for Figure 2A



The output of line 19 reflects the increment of p by 1. Line 18 allowed p to point at the b within the array, as expected.

Before continuing our examination of PTRINC, we need to discuss side effects, sequence points, expression statements, and the comma operator—features of the C language that relate directly to line 20 of PTRINC.

Side Effects

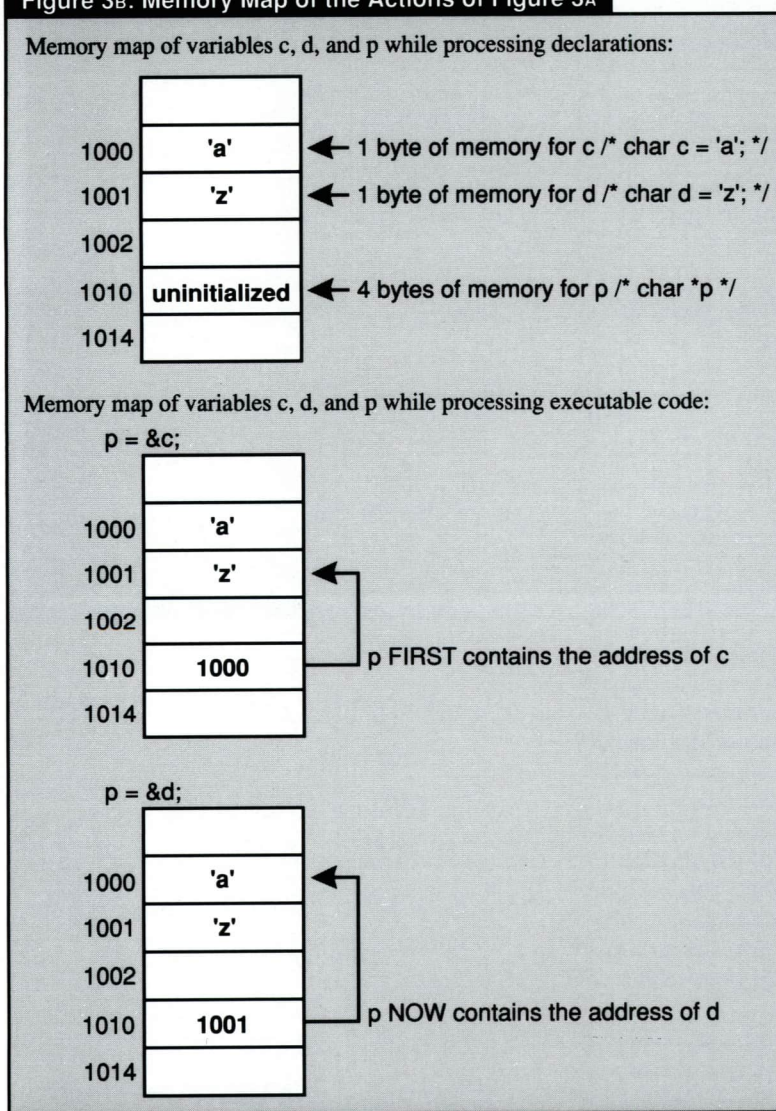
A side effect is a change in an object's value due to the evaluation of an expression. Side

A STRING LITERAL IS AN UNNAMED AREA OF STORAGE. TO USE ONE, EITHER SET A POINTER TO ITS ADDRESS OR USE IT AS AN ARGUMENT TO A FUNCTION WHERE ITS ADDRESS BECOMES APPARENT. MOST OF US HAVE USED A STRING LITERAL AS AN ARGUMENT TO PRINTF.

Figure 3A: Assigning Another Value to a Pointer

```
main()
{
    char    c = 'a';
    char    d = 'z';
    /* .... */
    char    *p;

    /* .... */
    p = &c;
    p = &d;
    /* .... */
}
```

Figure 3B: Memory Map of the Actions of Figure 3A



effects happen as a result of function calls, assignment expressions, auto-increment expressions, and auto-decrement expressions. Unless an expression is void, it produces a value; in the process, it also produces a side effect.

Side effects involving functions can modify the value of a variable in a way that might not be clear in a large or complex program. For example, in

```
int     i;
void f(void)
{
     i++;
}

main()
{
    f();
}
```

it may appear that the call to f is innocent, but actually it modifies the value of i. No real harm is done here, although the increment of i may not be apparent in a larger program or in a program with many statements occurring in f and main. If, however, we modify this code slightly by introducing another function named g

```
int g(void)
{
    i++;
    f();

    return (5);
}
```

and add

```
j = g() + i;
```

to main, the resulting value of j may not be clear. Either way we are dealing with a side effect.

Code such as this assignment statement

```
i = 5;
```

has the side effect of replacing the value of i with 5. As long as an expression modifies an object, a side effect occurs. Therefore a side effect can have ill effects.

The classic example of a side effect with an expression statement is:

```
a[i] = i++;
```

In this situation, the compiler has the choice of computing the value of the subscript before or after it computes the value on the

right-hand side of the equal sign. We would expect this unpredictability to be incorrect, but the compiler nevertheless makes this decision on its own.

Three more examples are represented by the code:

```
i = 5;
func(i++, i++);
```

In the first scenario, the compiler chooses to generate code to perform the post increment after both i's are evaluated. Both arguments will be 5; the value of i after the expression can be either 6 or 7.

The i's are post-incremented in the second scenario in the order shown; the first argument will be 5 and the second one will be 6. The value of i after the expression will be 7.

The i's are post-incremented in the last scenario in reverse of the order shown; the compiler is not constrained to evaluate the arguments of a function in left-to-right order. The second argument will be 5 and the first one will be 6. The value of i after the expression will be 7.

Generally speaking, the third scenario will usually be true, since most compilers put the arguments to functions on the stack in reverse order. This, however, does not necessarily imply that the arguments are evaluated as they are put on the stack. All side effects of function arguments are evaluated before calling the function, which makes them consistent and does not conflict with any of the scenarios given above.

As tricky as these examples are, we can accept them after giving them some thought. In fact, these examples appear much more sensible than the statement:

```
j = i++ + i++;
```

I will leave this one as an exercise. The possibilities it presents will be similar to those discussed above.

Figure 4A: Using the Pointer in an RValue Expression

```
main()
{
    char    c = 'a';
    char    d = 'z';
    /* .... */
    char    *p;
    char    *p2;

    /* .... */
    p = &c;
    p2 = p;
    p = &d;
    /* .... */
   printf("%p\n", p);
   printf("%p\n", p2);
}
```

Expression Statements

Most C programmers do not realize that C is more expression-based than any of the other popular general-purpose languages. For instance, even though we typically refer to a statement such as

```
i = 5;
```

as an assignment statement, it is actually an assignment expression. A statement in C can be one of the standard flow control statements such as for, goto, and break. In the *Microsoft® C Version 5.1 Optimizing Compiler* manual, the syntax summary on page 229 in Appendix B of the *User's Guide Language Reference* manual informs us that a statement may be of the form:

```
expression;
```

The assignment listed above is actually an expression statement containing an assignment.

We've already examined several situations in PTRINC that prove that this example is an assignment expression statement. For instance, line 18 of PTRINC contains p++. Even though this resembles p = p + 1, the syntax is clearly a construct that allows it to stand alone. Even p = p + 1 results in a value—this statement not only assigns p + 1 to p, it also returns p. The return value of the expression is often ignored; this is how statements such as j = i = 5 can occur without wreaking havoc.

Sequence Points

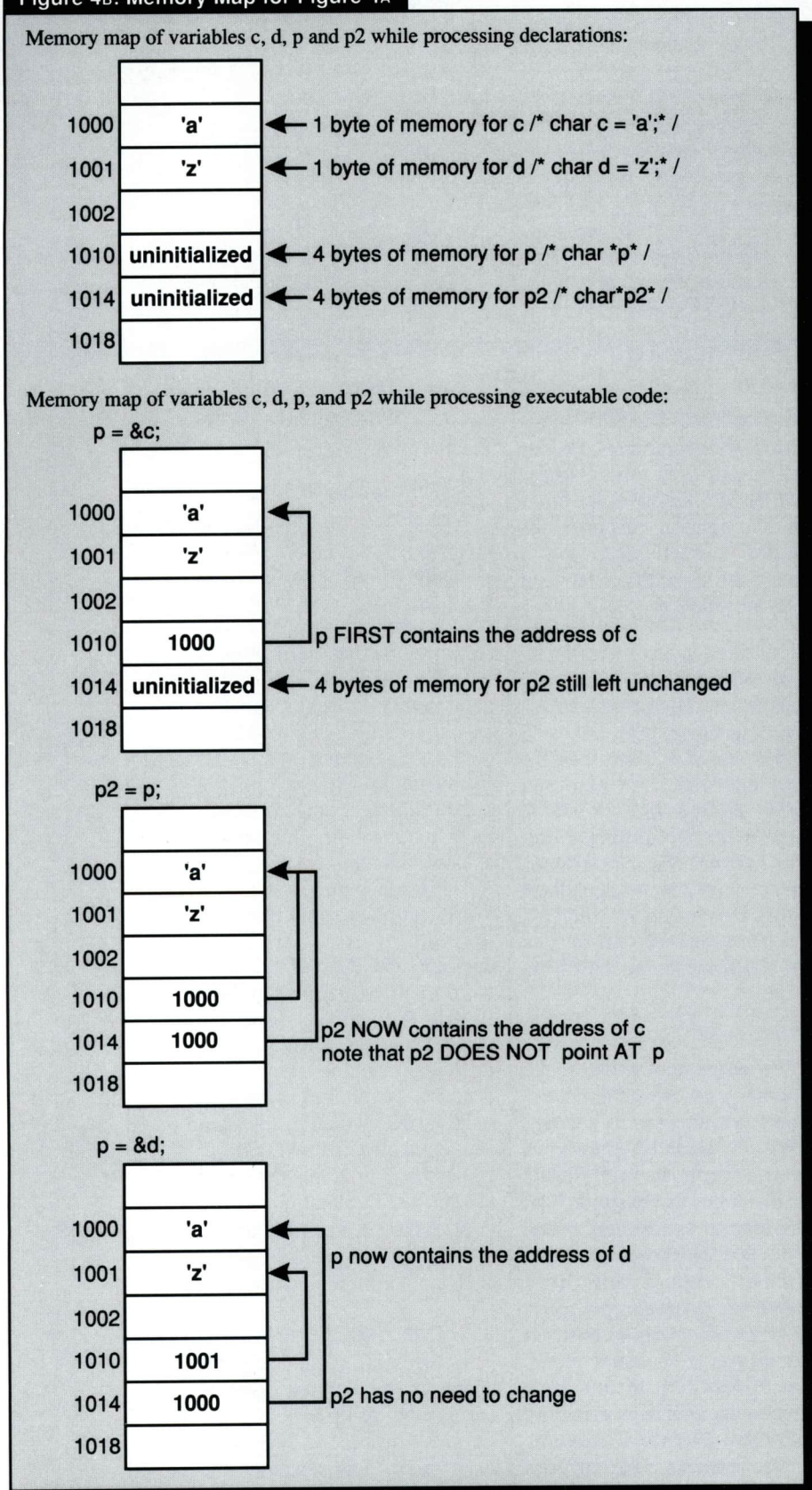
A sequence point invokes the concept of a hypothetical C machine. In such a machine, the compiler's code generator must completely follow your C code. A more precise definition is shown in **Figure 8.** Every statement in this C machine must be executed in the order in which it has been coded: statement order followed by flow of control order. This means that the hypothetical C machine allows few if any optimizations to take place.

This C machine exists only in theory; in practice no compiler will follow it. The comma operator and the volatile keyword (which many compilers still do not support) serve the same function as the hypothetical C machine. The comma operator is explained in the next section. The volatile keyword was explained in "A Guide to Understanding Even the Most Complex C Declarations," *MSJ* (Vol. 3, No. 5). Their purpose is to ensure that agreement points exist at appropriate places.

Comma Operator

As with all sequence point operators, the comma operator ensures that its operand expressions are evaluated in left-to-right order. The result of the comma operator is always the

Figure 4B: Memory Map for Figure 4A



value of the last expression in the expression list. Generally, the comma operator cannot force the evaluation of a given statement sequence, such as a for or block compound statement. The comma operation is only valid within expressions—it does not control statements, though it may be used within the controlling expression of a conditional statement.

The comma operator is different from a semicolon; it occurs in expressions, while the semicolon terminates statements. Also, note that the comma operator has nothing to do with the comma that is used to separate function arguments. In the latter case, the comma serves strictly as punctuation.

PTRINC.C

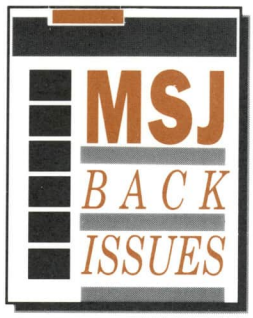
We've just discussed many features of C; I'll now explain how they pertain to line 20 of PTRINC. Line 20 of PTRINC contains the statement *p++. To determine what this means, look at the operator precedence table (see **Figure 7** for the output of line 20). A precedence table can be found on page 137 of the *Microsoft C 5.1 Optimizing Compiler Language Reference*.

The precedence table reveals whether line 20 increments the contents of p, increments p, changes the address of p, increments the contents of the object p points to, or performs some combination of these.

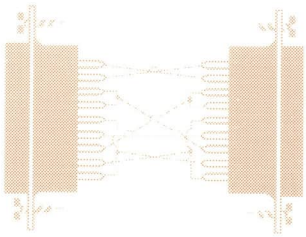
Some of these cases are obvious. The first two cases are the same for the reasons stated earlier. Case c cannot be true, since we do not have the ability to change the address of a variable, especially when that variable is an identifier.

We are left with the following possibilities: line 20 increments the contents of p, increments the contents of the object p points to, or performs some combination of the above.

Quick, before they run out...



Order your back issues of *Microsoft Systems Journal*. They're now available at discounted prices. Here's a complete listing of all back issues of *MSJ* in stock. Pick and choose the ones that interest you; or, if you prefer, get complete volumes (and save a few dollars). But don't delay. Supplies are limited; first come, first served.



October 1986: Vol. 1 No. 1

Article	Author	Page
◆ Advanced Reuter Terminal Gives Traders Windows on Financial World	MSJ Interview	1
◆ DDE: A Public Protocol for Advanced Application Linkages	Harvey Berger	7
◆ New Intel Graphics Coprocessor Makes Windows Sparkle	Joe Desposito	17
◆ TI's Programmable Graphics Processor Perks up PC Pixels	Joe Desposito	21
◆ Latest Dialog Editor Speeds Windows Application Development	Charles Petzold	25

December 1986: Vol. 1 No. 2

Article	Author	Page
◆ Aldus: Preparing PageMaker for the Move to Windows	Kevin Strehlo	1
◆ Moving Toward an Industry Standard for Developing TSRs	Nancy Andrews	7
◆ A Step-by-Step Guide to Building Your First Windows Application	Charles Petzold	13
◆ New XENIX Version Will Be First to Run On the Intel 80386	Joe Desposito	25
◆ A New Generation of Debugging Arrives with CodeView	Charles Petzold	29

March 1987: Vol. 2 No. 1

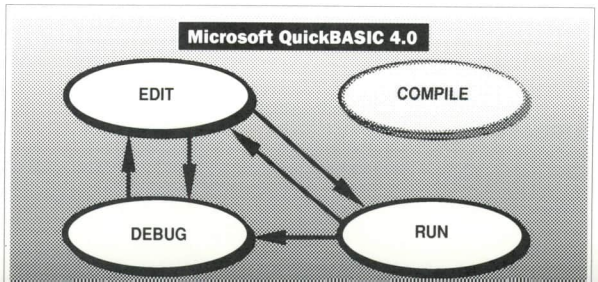
Article	Author	Page
◆ IRMA: A 3278 Terminal Emulator for Micro-to-Mainframe Communication	Frank Derfler & Edward Halbert	1
◆ Upgrading Applications for Multi-user Environments	Robert Cowart	11
◆ Expanded Memory: Writing Programs That Break the 640K Barrier	Marion Hansen & Bill Krueger & Nick Stuecklen	21
◆ Keep Track of Your Windows Memory with FREEMEM	Charles Petzold	33
◆ A Guide to Debugging With CodeView	David Norris & Michael J. O'Leary	41
◆ Page Description Languages: High-Level Languages for Printer Independence	Steve Rosenthal	49
◆ Dial 2.0 Provides Software Developers with Integrated Support System	Barbara Krasnoff	59
◆ Rich Text Format Standard Makes Transferring Text Easier	Nancy Andrews	63

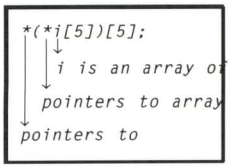
May 1987: Vol. 2 No. 2

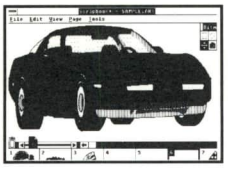
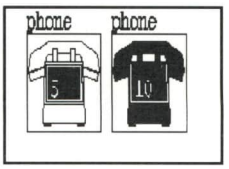
Article	Author	Page
◆ Microsoft Operating System/2: A Foundation for the Next Generation	Tony Rizzo	1
◆ OS/2 Windows Presentation Manager: Microsoft Windows on the Future	Manny Vellon	13
◆ OS/2 DOS Environment: Compatibility and Transition for MS-DOS Programs	Joel Gillman	19
◆ OS/2 Multitasking: Exploiting the Protected Mode of the 286	Ray Duncan	27
◆ OS/2 Inter-Process Communication: Semaphores, Pipes, and Queues	Ray Duncan	37
◆ A Compleat Guide to Writing Your First OS/2 Program	Charles Petzold	51
◆ Turn Off the Car to Change Gears: An Interview with Gordon Letwin	Lori Valigra	61
◆ A Simple Windows Application for Custom Color Mixing	Charles Petzold	67

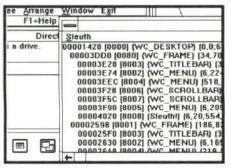
July 1987: Vol. 2 No. 3

Article	Author	Page
◆ PLEXUS Introduces Windows-based Tools for Building Image Databases	Kevin Strehlo	1
◆ Porting MS-DOS® Assembly Language Programs to the OS/2 Environment	Ray Duncan	9
◆ Microsoft Windows 2.0: Enhancements Offer Developers More Control	Michael Geary	19
◆ Keeping Up With the Real World: Speedy Serial I/O Processing	Ross M. Greenberg	37
◆ BLOWUP: A Windows Utility for Viewing and Manipulating Bitmaps	Charles Petzold	51
◆ Increase the Performance of Your Programs with a Math Coprocessor	Marion Hansen & Lori Sargent	59
◆ TIFF: An Emerging Standard for Exchanging Digital Graphic Imagines	Nancy Andrews & Stan Fry	71







September 1987: Vol. 2 No. 4

Article	Author	Page
◆ Microsoft® Windows/386: Creating a Virtual Machine Environment	Ray Duncan	1
◆ Programming in C the Fast and Easy Way with Microsoft® QuickC™	Augie Hansen	15
◆ Character-Oriented Display Services Using OS/2's VIO Subsystem	Ray Duncan	23
◆ Dynamic Allocation Techniques for Memory Management in C Programs	Steve Schustack	35
◆ CD ROM Technology Opens the Doors on a New Software Market	Tony Rizzo	47
◆ MS-DOS® CD ROM Extensions: A Standard PC Access Method	Tony Rizzo	54
◆ Microsoft® QuickBASIC: Everyone's First PC Language Gets Better	Dan Mick	63

November 1987: Vol. 2 No. 5

Article	Author	Page
◆ Microsoft® Excel for Windows: Meeting the Demands of a New Generation	Jared Taylor	1
◆ Interprogram Communication Using Windows' Dynamic Data Exchange	Kevin P. Welch	13
◆ Designing for Windows: An Interview with the Microsoft® Excel Developers	MSJ Interview	39
◆ A Strategy for Building and Debugging Your First MS-DOS® Device Driver	Ross M. Greenberg	51
◆ Microsoft C Optimizing Compiler 5.0 Offers Improved Speed and Code Size	Augie Hansen	67

January 1988: Vol. 3 No. 1

Article	Author	Page
◆ Preparing for Presentation Manager: Paradox Steps Up to Windows 2.0	Craig Stinson	1
◆ Converting Windows Applications for Microsoft® OS/2 Presentation Manager	Michael Geary	9
◆ Programming Considerations in Porting to Microsoft® XENIX ® System V/386	Martin Dunsmuir	31
◆ HEXCALC: An Instructive Pop-Up Calculator for Microsoft® Windows	Charles Petzold	39
◆ Effectively Using Far and Huge Data Pointers in Your Microsoft® C Programs	Kaare Christian	49
◆ EMS Support Improves Microsoft® Windows 2.0 Application Performance	Paul Yao	57
◆ LIM EMS 4.0: A Definition for the Next Generation of Expanded Memory	Marion Hansen & John Driscoll	67

March 1988: Vol. 3 No. 2

Article	Author	Page
◆ Microsoft® Windows Adapts to the Unique Needs of the Japanese Market	Tom Sato & Lin F. Shaw	1
◆ Utilizing OS/2 Multithread Techniques in Presentation Manager Applications	Charles Petzold	11
◆ OS/2 LAN Manager Provides a Platform for Server-based Network Applications	Alan Kessler	29
◆ Writing OS/2 Bimodal Device Drivers: An Examination of the DevHlp API	Ray Duncan	39
◆ Exploring the Structure and Contents of the MS-DOS® Object Module Format	Richard Wilton	56
◆ A Guide to Program Editors, the Developer's Most Important Tool	Tony Rizzo	63

```
     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F 0123456789ABCDEF
0000 43 4F 4D 53 50 45 43 3D 43 3A 5C 43 4F 4D 4D 41 COMSPEC=C:\COMMA
0010 4E 44 2E 43 4F 4D 00 50 52 4F 4D 50 54 3D 24 70 ND.COM.PROMPT=$p
0020 24 5F 24 64 20 20 20 24 74 24 68 24 68 24 68 24 $_$d   $t$h$h$h$
0030 68 24 68 24 68 20 24 71 24 71 24 67 00 50 41 54 h$h$h $q$q$g.PAT
0040 48 3D 43 3A 5C 53 59 53 54 45 4D 3B 43 3A 5C 41 H=C:\SYSTEM;C:\A
0050 53 4D 3B 43 3A 5C 57 53 3B 43 3A 5C 45 54 48 45 SM;C:\WS;C:\ETHE
0060 52 4E 45 54 3B 43 3A 5C 46 4F 52 54 48 5C 50 43 RNET;C:\FORTH\PC
0070 33 31 3B 00 00 01 00 43 3A 5C 46 4F 52 54 48 5C 31;....C:\FORTH\
0080 50 43 33 31 5C 46 4F 52 54 48 2E 43 4F 4D 00    PC31\FORTH.COM.
```

May 1988: Vol. 3 No. 3

Article	Author	Page
◆ SQLWindows Brings a Graphical User Interface to SQL Database Applications	Craig Stinson	1
◆ The Graphics Programming Interface: A Guide to OS/2 Presentation Spaces	Charles Petzold	9
◆ Using OS/2 Semaphores to Coordinate Concurrent Threads of Execution	Kevin Ruddell	19
◆ Design Concepts and Considerations in Building an OS/2 Dynamic-Link Library	Ross M. Greenberg	27
◆ New Compiler Technology Boosts Microsoft® QuickBASIC 4.0 Productivity	Augie Hansen	49
◆ Debug Microsoft® Windows Programs More Effectively with a Simple Utility	Kevin P. Welch	64
◆ An Examination of the Operating Principles of the Microsoft Object Linker	Richard Wilton	73

July 1988: Vol. 3 No. 4

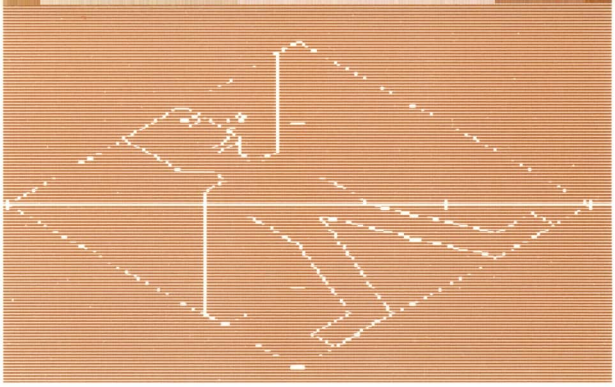
Article	Author	Page
◆ DARWIN: Merrill Lynch Develops a New Workstation Based on Windows 2.0	Tony Rizzo & Karen Strauss	1
◆ CodeView for Windows Provides an Interactive Debugging Environment	Paul Yao & David Durant	13
◆ OS/2 Graphics Programming Interface: An Introduction to Coordinate Spaces	Charles Petzold	23
◆ Microsoft® Macro Assembler Version 5.1 Simplifies Macros and Interfacing	Ross M. Greenberg	41
◆ Color Mixing Principles and How Color Works in the Raster Video Model	Kevin P. Welch	49
◆ Creating User-Defined Controls for Your Own Windows Applications	Kevin P. Welch	54
◆ SQL Server Brings Distributed DBMS Technology for OS/2 Via LAN Manager	Marc Adler	67

September 1988: Vol. 3 No. 5

Article	Author	Page
◆ Bridge/386™: A Tool for Integrating Applications Under Windows/386	Matt Trask	1
◆ A Guide to Understanding Even the Most Complex C Declarations	Greg Comeau	10
◆ Techniques for Debugging Multi-thread OS/2 Programs with CodeView® 2.2	Charles Petzold	21
◆ Exchanging Data Between Applications Using the Windows Clipboard	Kevin P. Welch	31
◆ Using Microsoft® C Version 5.1 to Write Terminate-and-Stay-Resident Programs	Kaare Christian	47
◆ Customizing the Features of the M Editor Using Macros and C Extensions	Leo N. Notenboom	59
◆ Dynamically Creating Dialog Boxes Using New Windows 2.*x* Functions	Don Hasson	73

November 1988: Vol. 3 No. 6

Article	Author	Page
◆ Building a Device-Independent Video Display I/O Library in Microsoft® C	Jeff Prosise	1
◆ Developing SQL Server Database Applications Through DB-Library	Marc Adler	13
◆ OS/2 Protected-Mode Programming with Forth, LISP, Modula-2, and BASIC	Andrew Schulman	25
◆ The High Memory Area: Addressing 64Kb More Memory in Real Mode	Chip Anderson	53
◆ Developing and Debugging Embedded Systems Applications	Y.P. Chien, Ph.D.	58
◆ C Scope and Linkage: The Keys to Understanding Indentifier Accessibility	Greg Comeau	65
◆ Extending the Functions of the Windows Clipboard with Scrapbook+	Kevin P. Welch	73



January 1989: Vol. 4 No. 1

Article	Author	Page
◆ Quotron® Uses Windows to Develop New Market Analysis Tools for Real-Time Data	Tony Rizzo & Karen Strauss	1
◆ Porting Apple® Macintosh® Applications to the Microsoft® Windows Environment	Andrew Schulman & Ray Valdes	11
◆ Developing Applications with Common Source Code for Multiple Environments	Michael Geary	41
◆ Using the OS/2 Environment to Develop DOS and OS/2 Applications	Richard Hale Shaw	77

March 1989: Vol. 4 No. 2

Article	Author	Page
◆ Exploring Vector Fonts with the OS/2 Graphics Programming Interface	Charles Petzold	1
◆ BASIC as a Professional Programming Language: An Interview with Ethan Winer	MSJ Interview	15
◆ Organizing Data in Your C Program with Structures, Unions, and Typedefs	Greg Comeau	23
◆ Whitewater's Actor®: An Introduction to Object-Oriented Programming Concepts	Zack Urlocker	33
◆ MDI: An Emerging Standard for Manipulating Document Windows	Kevin P. Welch	45
◆ Planning and Writing a Multithreaded OS/2 Program with Microsoft C	Richard Hale Shaw	63

May 1989: Vol. 4 No. 3

Article	Author	Page
◆ A Technical Study of Dynamic Data Exchange Under Presentation Manager	Susan Franklin & Tony Peters	1
◆ Creating a Virtual Memory Manager to Handle More Data in Your Applications	Marc Adler	17
◆ Using the OS/2 Video I/O Subsystem to Create Appealing Visual Interfaces	Richard Hale Shaw	25
◆ Investigating the Debugging Registers of the Intel 386 Microprocessor	Marion Hansen & Nick Stuecklen	39
◆ Strategies for Building and Using OS/2 Run-Time Dynamic-Link Libraries	Ross M.Greenberg	51
◆ How the 8259A Programmable Interrupt Controller Manages External I/O Devices	Jim Kyle & Chip Rabinowitz	59
◆ Advanced Techniques for Using Structures and Unions In Your C Code	Greg Comeau	69

List continues on next page ➤

Issue	Qty	Price	Total
October 1986: Vol. 1 No.1*			
December 1986: Vol. 1 No.2			
March 1987: Vol. 2 No.1*			
May 1987: Vol. 2 No.2			
July 1987: Vol. 2 No.3			
September 1987: Vol. 2 No.4			
November 1987: Vol. 2 No.5			
January 1988: Vol. 3 No.1*			
March 1988: Vol. 3 No.2			
May 1988: Vol. 3 No.3			
July 1988: Vol. 3 No.4			
September 1988: Vol. 3 No.5			
November 1988: Vol. 3 No.6			
January 1989: Vol. 4 No.1			
March 1989: Vol. 4 No.2			
May 1989: Vol. 4 No.3*			
July 1989: Vol. 4 No. 4			
September 1989: Vol. 4 No. 5			
Subtotal			
Shipping (**$1.50 U.S., $5.00 foreign/magazine**)			
Grand total			

Please fold in half with our return address on the outside, tape closed (PLEASE DO NOT STAPLE) and drop in the mail, or enclose in your own envelope.

*** Very limited supply.**

Your Name _____

Address _____

City _____ State_____ ZIP_____

Phone No. (___)_____

Shipping Address (if different)

Your Name _____

Address _____

City_____State_____ZIP_____

All orders are shipped via U.S. mail, unless otherwise requested.

ORDER FORM

Pricing: the more you buy, the less you pay for each.

Qty	Price each
1-5	$5.00
6-10	$4.50
11-15	$4.00
>15	$3.50

Complete volumes are available at $24 each

Payment Method

❑ Check enclosed
❑ VISA ❑ MasterCard
❑ American Express

Card # _____

Exp Date _____

Signature _____

Shipping Method

❑ US Post Office ❑ UPS

July 1989: Vol. 4 No. 4

September 1989: Vol. 4 No. 5

We know that the associativity of the ++ operator from the operator precedence table chart means "do not perform the increment until after the (simple) expression has been evaluated" because it is a post-increment operator. Since the ++ immediately follows p, it's p that's being incremented, not the contents of the object p points to.

The simple expression referred to in the preceding paragraph is *p. However, like the case presented in the previous sections, we don't assign or use the value of *p. It is discarded—*p goes nowhere and then p is incremented afterwards. This is equivalent to having coded the following:

```
*p, p++;
```

This syntax is a valid C expression and a valid C statement. It demonstrates that we wouldn't normally want to write code this way. For all practical purposes, line 20 increments only p; the implication is that it wasn't supposed to be written that way. We should be cautious of such situations, because it's too easy to code a construct such as *p++ without stopping for a second to see what is actually happening, especially since the compiler is not going to object.

Sometimes, however, it makes sense to write code that uses the value of this kind of expression. For example, the assignment to c in the following code is legitimate and powerful:

```
char  *p;
char  c;

p = ...;
c = *p++;
...;
```

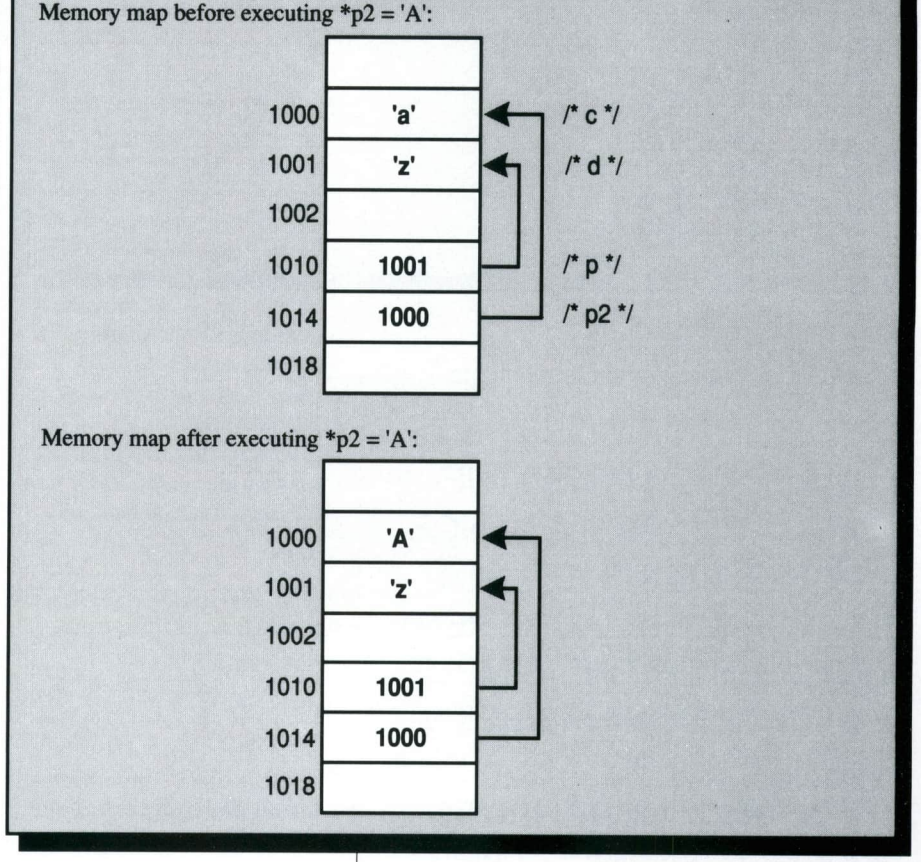
Statements like the assignment to c present a style issue because such coding practices can create hard-to-read code. Instead you could write the assignment on two lines:

```
c = *p;
p++;
```

Figure 5A: Using the Unary * Operator

```
main()
{
    char    c = 'a';
    char    d = 'z';
    /* .... */
    char    *p;
    char    *p2;

    /* .... */
    p = &c;
    p2 = p;
    p = &d;
    /* .... */
   printf("%c\n", *p);
    *p2 = 'A';
}
```

Figure 5B: Memory Map of Figure 5A



Such constructs, however, are idioms of C, so whether or not you should use them is simply a matter of style.

Next Case

In line 22, (*p)++ might increment the contents of p or increment the contents of the object to which p points.

Figure 6: PTRINC.C Source Code

```
 1  #include "debug.h"
 2
 3  /*
 4   * ptrinc.c
 5   */
 6
 7  main()
 8  {
 9      char    *p;
10      char    *origp;
11
12      p = "abcdefghijk";
13      prints(p);
14      origp = p;
15      prints(origp);
16
17      printp(p);
18      p++;
19      printp(p);
20      *p++;
21      printp(p);
22      (*p)++;
23      printp(p);
24      *++p;
25      printp(p);
26      printc(*p);
27      printc(*(p++));
28      printc(*p);
29      printp(p);
30
31      prints(p);
32      prints(origp);
33  }
```

WE KNOW THAT THE ASSOCIATIVITY OF THE ++ OPERATOR FROM THE OPERATOR PRECEDENCE TABLE MEANS "DO NOT PERFORM THE INCREMENT UNTIL AFTER THE (SIMPLE) EXPRESSION HAS BEEN EVALUATED" BECAUSE IT IS A POST-INCREMENT OPERATOR. SINCE THE ++ IMMEDIATELY FOLLOWS P, IT'S P THAT'S BEING INCREMENTED, NOT THE CONTENTS OF THE OBJECT P POINTS TO.

Since the ++ is outside the parentheses, it is doubtful that they modify p. Of course, (*p)++ is a legal expression that modifies the contents of p and ++ occurs outside the parentheses in that case. Also, if we envision a case such as

```
i++;
```

being at some point equivalent to

```
(i) = (i) + 1;
```

line 22 could be interpreted as:

```
(*p) = (*p) + 1;
```

Therefore line 22 increments the contents of the object to which p points. Case c wasn't worth considering since there is only one ++ operator in the statement. **Figure 7** shows that p has not changed when we print it on line 23. What has changed is that we've added 1 to c; under the ASCII character set this changes the c to a d. Line 22 could also have been written as either of the following:

```
++(*p)
++*p
```

since a standalone statement such as i++ could be rewritten as ++i. If the result of these statements were to be used, however, this would not be true since it makes a difference where ++ is placed, according to the precedence table. For example, the expression ++(*p) is evaluated and then incremented, but the expression ++*p is incremented first and then evaluated.

It may not be clear whether the ++ on line 24 applies to the * or the p. It can't apply to the * without the introduction of some other operand, like the ++*p above, which is mentioned as a syntax level reference since the two statements have different meanings. As with all these examples, the operator precedence table provides the key. Since the ++ binds with the p and is a pre-increment, the compiler will not let the dereference occur without first performing the increment. Therefore, line 24 could have been coded

```
p++, *p;
```

or:

```
++p, *p;
```

Much of what is true about line 20 is also true of this statement. The effect of 24 is that p is incremented. Line 25 confirms this. On line 26, we simply check to make sure we know where we are.

Turmoil strikes again on line 27 and this time we have two sets of problems. First we are going to use the value of the expression instead of ignoring it as we did in earlier statements. We are left with:

```
*(p++)
```

Clearly, p will be incremented, but what's not clear is when this will happen. At first glance, we would say that it's the same as

```
p++, *p
```

which is the same thing line 24 mapped into because expres-

sions in parentheses must be performed first. We'd be wrong. The statement actually maps to

```
*p, p++
```

which is the same as line 20. The explanation of this is that p++ constitutes a complete subexpression involving a long value no different from what

```
while ((c = getchar()) != EOF)
```

does in its assignments to c. In this case, the subexpression has a side effect involving a post operator. We know that the side effect will occur sometime during that statement and before the next statement in this situation. The operator precedence table clarifies the situation by showing that the post ++ operator and the unary * operator are always right-associative. This means that if we look at the reverse situation of *p++, it must be interpreted as *(p++) rather than (*p)++. Therefore, the parentheses on line 27 only order the p, not the p++, since the increment is performed afterwards.

Line 27 has a second problem. We can see that when line 27 is output and fed from the C preprocessor into the C parser it looks something like this:

```
fprintf((&_iob[2]),
"*(p++)=%d/%c\n", *(p++),
*(p++));
```

(*This code should all be on one line; it is broken here due to space considerations—Ed.*)

Notice that we have ended up with two *(p++) expressions—a statement containing a macro with a side effect. This is a very subtle problem. The decimal value of the character printed by the first *(p++) in this example may actually be different from the character value we expected because we still do not know when the p++'s will occur. On my UNIX machine, the p's were incremented in the reverse of the order shown, probably because

Figure 7: Map and "State Diagram" of PTRINC Code Starting with Line 12

Assume that p and origp are pointers each 4 bytes in length located at addresses 1200 and 1210, respectively.

```
Line No.  Address ⟶
          1  1  1  1  1  1  1  1  1  1  1  1
          2  2  2  2  2  2  2  2  2  2  2  2
          3  3  3  3  3  3  4  4  4  4  4  4
          4  5  6  7  8  9  0  1  2  3  4  5   output
12      'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' '\0' ⟶elements of
                                                          array
        p points at the beginning of array
13                                                  p="abcdefghijk"
          1  1  1  1  1  1  1  1  1  1  1  1
          2  2  2  2  2  2  2  2  2  2  2  2
          3  3  3  3  3  3  4  4  4  4  4  4
          4  5  6  7  8  9  0  1  2  3  4  5
14      'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' '\0'
        p points at the beginning of array
        origp points at the beginning of array
15
      origp="abcdefghijk"
17                                                  p=1234
          1  1  1  1  1  1  1  1  1  1  1  1
          2  2  2  2  2  2  2  2  2  2  2  2
          3  3  3  3  3  3  4  4  4  4  4  4
          4  5  6  7  8  9  0  1  2  3  4  5
18      'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' '\0'
       origp  p
19                                                  p=1235
          1  1  1  1  1  1  1  1  1  1  1  1
          2  2  2  2  2  2  2  2  2  2  2  2
          3  3  3  3  3  3  4  4  4  4  4  4
          4  5  6  7  8  9  0  1  2  3  4  5
20      'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' '\0'
       origp     p
21                                                  p=1236
          1  1  1  1  1  1  1  1  1  1  1  1
          2  2  2  2  2  2  2  2  2  2  2  2
          3  3  3  3  3  3  4  4  4  4  4  4
          4  5  6  7  8  9  0  1  2  3  4  5
22      'a' 'b' 'd' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' '\0'
       origp     p
23                                                  p=1236
          1  1  1  1  1  1  1  1  1  1  1  1
          2  2  2  2  2  2  2  2  2  2  2  2
          3  3  3  3  3  3  4  4  4  4  4  4
          4  5  6  7  8  9  0  1  2  3  4  5
24      'a' 'b' 'd' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' '\0'
       origp        p
25                                                  p=1237
26                                                  *p=100/d
          1  1  1  1  1  1  1  1  1  1  1  1
          2  2  2  2  2  2  2  2  2  2  2  2
          3  3  3  3  3  3  4  4  4  4  4  4
          4  5  6  7  8  9  0  1  2  3  4  5
27      'a' 'b' 'd' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' '\0'
       origp          p

                    or
```

CONTINUED

Figure 7 CONTINUED

```
        'a' 'b' 'd' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' '\0'
          ↑                   ↑
        origp                 p

27                                          *(p++)=101/d
28                                          *p=102/f
29                                          p=1239
31                                          p="fghijk"
32                                          origp="abddefghijk"
```

Figure 8: Sequence Points

Sequence Point The point at which all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.

Agreement Point The sequence point for some object or class of objects at which the value of the object(s) in the real implementation must agree with the value prescribed by the abstract machine.

The sequence points of a C program are:

- at the call to a function after the arguments to the function have been evaluated
- at the end of the first operand of the following operators:
  - logical and: &&
  - logical or: ||
  - conditional: ?
  - comma: ,
- at the end of a full expression such as:
  - an initializer
  - the expression in an expression statement
  - the controlling expression of a selection statement (if or switch)
  - the controlling expression of an iteration statement (while, do, or for)
  - the expression in a return statement.

THE OPERATOR PRECEDENCE TABLE SHOWS THAT THE POST ++ OPERATOR AND THE UNARY * OPERATOR ARE ALWAYS RIGHT-ASSOCIATIVE. IF WE LOOK AT THE REVERSE OF *P++, IT MUST BE INTERPRETED AS *(P++) RATHER THAN (*P)++. THEREFORE, THE PARENTHESES ORDER ONLY THE P, NOT THE P++ .

of the way the arguments on the stack were put. The value for the d is correct, but the decimal value 101 represents an e (see **Figure 7**).

Line 28 verifies that there were two increments on my machine. There might not have been since we cannot predict whether both increments start using the same value of p or whether one is based on the other. Lines 29, 31, and 32 are self-explanatory except for the third character of the array, which is now a d rather than a c. This change occurred in line 22.

Summary

This article has introduced and explained many pointer derivations and shown some common and often unexpected side effects of using pointers in your code. A solid understanding of how pointers work, and a willingness to examine even simple pointer constructions for hidden errors, will help you to write concise, fast code.

In-Line C Program Debugging

DEBUG.H is a header file that performs in-line program debugging. An example of DEBUG.H (see **Figure A**) is shown in TESTDEBUG.C (see **Figure B**). TESTDEBUG.C includes DEBUG.H in a source file and uses it and three macro variables to control the output. One macro controls the occurrence of any debug output, another controls the printing of subroutine entry and exit points, and the third controls the printing of subroutine names with the respective debugging information.

The output shown for TESTDEBUG.C demonstrates the advantage of using DEBUG.H, which is that the argument to the print* and DEBUG* macros allows the code to be less tedious and more easily read (assuming the debug information is to be left in). In this program, a statement such as

```
printd(somevariable)
```

produces

```
somevariable = the value of
               somevariable
```

as its output. This avoids having to code fprintf(stderr, "i=%d\n", i); or a similar statement whenever you want to see what value a variable holds. For instance, line 16 in TESTDEBUG.C produces i = 5. The expanded form can become a nuisance, especially with more complicated variables such as a member of an element of an array of structures. As in DEBUG.H, there are five categories of output statements controlled by the last character of the printx

Figure A: DEBUG.H

```
/* debug.h  */

#ifndef DEBUG_H
#define DEBUG_H

/* Depending upon which compiler you have, STRINGIZING should be set to
determine if #v syntax is allowed. For some compilers, this setting may not
be obvious since __STDC__ may not be supported or may be set to an
inappropriate value. */

#if __STDC__ == 1
#define STRINGIZING 0
#else
#define STRINGIZING 1 /* 1 means compiler can use #v syntax. You may need
                         to swap this line with the other one */
#endif

#include <stdio.h>

#ifndef SUBTRACE
#define SUBSTART(x)
#define SUBEND(x)
#else
#if STRINGIZING == 1
#define SUBSTART(func) fprintf(stderr, "%s()ing\n", #func)
#define SUBEND(func) fprintf(stderr, "%s()ed\n", #func)
#else
#define SUBSTART(func) fprintf(stderr, "%s()ing\n", "func")
#define SUBEND(func) fprintf(stderr, "%s()ed\n", "func")
#endif
#endif

#if STRINGIZING == 1
#define prints(str) fprintf(stderr, #str "='%s'\n", str)
#define printd(dec) fprintf(stderr, #dec "=%d\n", dec)
#define printld(ldec) fprintf(stderr, #ldec "=%ld\n", (long)ldec)
#define printp(p) fprintf(stderr, #p "=%p\n", p)
#define printc(ch) fprintf(stderr, #ch "=%d/%c\n", ch, ch)
/* printc without a side effect is properly written as:
  #define printc(ch) {char c = (ch);fprintf(stderr, #ch "=%d/%c\n",
   c, c); } */
#else
#define prints(str) fprintf(stderr, "str='%s'\n", str)
#define printd(dec) fprintf(stderr, "dec=%d\n", dec)
#define printld(ldec) fprintf(stderr, "ldec=%ld\n", (long)ldec)
#define printp(p) fprintf(stderr, "p=%p\n", p)
#define printc(ch) fprintf(stderr, "ch=%d/%c\n", ch, ch)
/* printc without a side effect is properly written as:
  #define printc(ch) {char c = (ch);fprintf(stderr, "ch=%d/%c\n",
   c, c); }*/
#endif

#define PRINTSUBfprintf(stderr, "%s:", __FILE__)

#ifndef DEBUGL1
#define DEBUG1(a1)
#define DEBUG2(a1, a2)
#define DEBUG3(a1, a2, a3)
#define DEBUG4(a1, a2, a3, a4)
#define DEBUG5(a1, a2, a3, a4, a5)
#define DEBUG6(a1, a2, a3, a4, a5, a6)
#define DEBUGS(str)
#define DEBUGD(d)
#define DEBUGLD(ld)
#define DEBUGC(c)
#define DEBUGP(p)
#define DEBUGCALL(func)
#else
#ifdef DEBUGF
#define DEBUG1(a1) PRINTSUB, fprintf(stderr,"%s",a1)
#define DEBUG2(a1,a2) PRINTSUB, fprintf(stderr,a1,a2)
#define DEBUG3(a1,a2,a3) PRINTSUB, fprintf(stderr,a1,a2,a3)
#define DEBUG4(a1,a2,a3,a4) PRINTSUB, fprintf(stderr,a1,a2,a3,a4)
#define DEBUG5(a1,a2,a3,a4,a5) PRINTSUB, fprintf(stderr,a1,a2,a3,a4,a5)
#define DEBUG6(a1,a2,a3,a4,a5,a6) PRINTSUB, fprintf(stderr,a1,a2,a3,a4,a5,a6)
#define DEBUGS(str)PRINTSUB, prints(str)
```

CONTINUED

Figure A CONTINUED

```
#define DEBUGD(dec)PRINTSUB, printd(dec)
#define DEBUGLD(ld)PRINTSUB, printld(ldec)
#define DEBUGC(c)PRINTSUB, printc(c)
#define DEBUGP(p)PRINTSUB, printp(p)
#define DEBUGCALL(func) PRINTSUB, func
#else
#define DEBUG1(a1) fprintf(stderr, "%s", a1)
#define DEBUG2(a1, a2) fprintf(stderr, a1, a2)
#define DEBUG3(a1, a2, a3) fprintf(stderr, a1, a2, a3)
#define DEBUG4(a1, a2, a3, a4) fprintf(stderr, a1, a2, a3, a4)
#define DEBUG5(a1, a2, a3, a4, a5) fprintf(stderr, a1, a2, a3, a4, a5)
#define DEBUG6(a1, a2, a3, a4, a5, a6) \
            fprintf(stderr, a1, a2, a3, a4, a5, a6)
#define DEBUGS(str)prints(str)
#define DEBUGD(dec)printd(dec)
#define DEBUGLD(ld)printld(ldec)
#define DEBUGC(c)printc(c);
#define DEBUGP(p)printp(p)
#define DEBUGCALL(func) func
#endif
#endif

#endif
```

Figure B: TESTDEBUG.C

```
#define DEBUGL1
#define DEBUGF
#define SUBTRACE
#include "debug.h"

/* TESTDEBUG.C */

main()
{
int i = 5;
int *pi = &i;
char *s = "string";

SUBSTART(main);

printd(i);
DEBUGD(i);
DEBUGS(s);
DEBUGP(&i);
DEBUGP(pi);

SUBEND(main);
}
```

When executed, this program prints:

```
main()ing
i = 5
testdebug.c:i = 5
testdebug.c:s = 'string'
testdebug.c:&i = 0005:0FB8
testdebug.c:pi = 0005:0FB8
main()ed
```

statement. (You can add more.) Use *s* to print the value of a string, *d* to print an int or short, *ld* to print a long, *p* to print a pointer, and *c* to print a character.

Also note that DEBUG.H contains a group of macros, the print*x* macros, which you can use to print unconditionally. This is in contrast to the DEBUG*x* macros which have an effect on your code only if you define DEBUGL1 before you include DEBUG.H. Therefore, TESTDEBUG.C will always print the value of i, but whether it prints any of the other output, including the subroutine trace, depends upon DEBUGL1 being an active macro.

Space does not permit us to elaborate upon the internal workings of DEBUG.H. Remember that for the purposes of this article, print?(var); allows you to produce var = value. TESTDEBUG.C should be sufficient to demonstrate the interface to DEBUG.H. ❑

Integrating Subsystems and Interprocess Communication in an OS/2 Application

Richard Hale Shaw

Programming in the OS/2 systems is always a challenging and interesting experience. One reason is that the OS/2 environment provides a platform for integrated application development that is richer than its rivals and outperforms them. Another is that the OS/2 application programming interface (API) is a robust set of functions that makes even the most complicated applications much easier to design, develop, test, and complete.

The intent of this series of OS/2 articles (*MSJ* Vol. 4, Nos. 1 through 6) has been twofold. First, its purpose was to introduce you to OS/2 programming, showing you how to write multithreaded applications, use the Vio, Kbd, and Mou subsystems, and allow multiple OS/2 applications to communicate via interprocess communication (IPC). This final article presents an application that integrates all of those aims. Second, the series intended to give you the OS/2 kernel programming knowledge needed to program for OS/2 Presentation Manager (hereafter PM). Good PM programming requires a thorough understanding of three elements: the OS/2 multithreaded application programming environment and application programming interface (API); PM windowing facilities (and their associated API functions); and PM event-driven, message-based architecture (a superset of the architecture of the Microsoft® Windows™ environment).

The first five articles in this series have provided an overview of the OS/2 API. This final article discusses how to use OS/2 IPC to implement an event-driven, message-based queue that you can use to construct applications whose architecture is similar to the architecture of PM.

Directory Information

The programs in this article prepare you for PM programming. They assume that you are familiar with how to work with multiple threads, the various subsystems, and IPC. The programs integrate many (if not all) of the concepts presented in the previous articles in the series. They are also practical; that is, they are either useful in their own right or can be easily modified to fill a particular need.

The directory information (DI) application presented in this article consists of three programs built on a client-server architecture. One, a directory server, gives disk and directory information to any client program that requests it, provided the client knows how to initiate the request and use the results. Each of the client

Richard Hale Shaw, a writer who contributes to various computer magazines, is the author of an upcoming book on OS/2 programming.

THE DI CLIENT-SERVER ARCHITECTURE IS IMPLEMENTED USING IPC. WHEN THE SERVER STARTS, IT OPENS A QUEUE THAT IS KNOWN TO THE DI FUNCTION IN A CLIENT PROGRAM. A CLIENT DI FUNCTION PASSES REQUESTS FOR DIRECTORY INFORMATION TO THE SERVER VIA THIS QUEUE, AND THE SERVER CREATES A THREAD TO SERVICE THE REQUEST.

Figure 1: DISIMPLE.C

```
#define INCL_DOS

#include<os2.h>
#include<stdio.h>
#include"di.h"


void main(int argc, char **argv);

void main(int argc, char **argv)
    {
   PVOID requesthdl = NULL, resulthdl;
   USHORT  i, numresults;
   char    filename[40];

   if(argc != 2)
        {
      printf("Usage: disimple filespec\n");
      DosExit(EXIT_PROCESS,0);
        }

  DiMakeRequest(&requesthdl,argv[1],0);
  DiSendRequest(requesthdl);
  DiGetResultHdl(requesthdl,0,&numresults,&resulthdl);

   for( i = 0; i < numresults; i++)
        {
        if(i)
          DiGetNextResult(resulthdl,filename);
        else
          DiGetFirstResult(resulthdl,filename);
      printf("\t%s\n",filename);
        }
  DiDestroyRequest(&requesthdl);
   }
```

applications uses a set of function calls to generate and process a request for directory information. The function calls, combined with the server, are a high-level interface to the OS/2 DosFindFirst function; this interface is simple and easy to use, it hides the complexities of finding files and managing memory, and it avoids having to make multiple calls to the DosFindNext function. The server program can also be modified to run on a network, where it could be enabled to display information about directories that a program running on a node might not be able to access. And, although I didn't take this step, the client functions can be placed in an OS/2 dynamic-link library (DLL), where they can be used by more than one program at a time even though they are only loaded once by OS/2[1].

Two kinds of client programs are presented here. One is a simple command-line directory utility that I ported to OS/2 and modified to use the directory information functions. The other is a directory program for end users that employs multiple threads to manage input and output and lets the user work with the keyboard and mouse.

Client-Server Architecture

The DI functions implement a classic client-server relationship. If, for example, the client asks "C:\OS2*.* ", the server will respond with a list of all the files that meet that specification. The client can request the expansion of multiple filespecs at a time. A filespec may consist of a full or partial path and a standard MS-DOS® operating system file specification, including wildcard characters. If a path is not included, the client's current directory is assumed; each filespec can have a different path. Each filespec can also have a unique matching attribute value. The server will return the results in a form that makes it easy for other DI functions to sort them, retrieve them, and make use of the information.

The DI client-server architecture is implemented using OS/2 interprocess communication. When you start the DI server, it opens a queue that is known to the DI functions bound into a client program. A client DI function can pass requests for directory information to the server via this queue, and the server will create a thread to service the request. The server can manage up to 20 threads at a time, making it easy for it to handle multiple demands for directory information. Each server thread receives a request packet that is sent via the queue from the client. (The queue actually carries a pointer to a shared memory block that contains the request.) This request packet includes all the information necessary for the server to process the request, as well as a pointer to a work area segment (also in shared memory) where the server thread must place the results. When the server thread has finished its task, it clears a semaphore in the request header, which signals the client that the thread's task is complete. Then the client can use other DI functions to retrieve the results of the request.

With DI functions, a client program can prepare a request by making one or more calls to the DiMakeRequest function. When the request has been prepared, the client can send it to the server using the DiSendRequest function, which will return when the request is complete. (You could put the call to this function in a separate thread, so that the application can continue while the server fulfills the request.) When the thread calling DiSendRequest returns,

the client can retrieve the request information by calling several different functions that are described later in the article. Finally, the request information should be released with DiDestroyRequest. An example of the code required to write such a client program is shown in the file DISIMPLE.C (see **Figure 1**).

DI Data Structures

DI functions use the two data structures shown in **Figures 2** and **3**. (*Full source code for the figures in this article can be downloaded from any* MSJ *bulletin board—Ed.*) **Figure 2**, REQUESTHEADER, is the primary structure and contains the information the server will need to process and complete the request. This information is stored in a segment that is known as the request header segment. This segment is allocated by the first call to DiMakeRequest (which also allocates the work segment). The information in the REQUESTHEADER structure includes the following:

- the client's selector to the work segment
- the server's selector to the request header segment and a server selector and pointer to the work segment
- the handle of the server's queue and the server's process ID (PID)
- the total size of the header and work segments
- the number of filespecs that make up the request
- the number of results found by the server
- a RAM semaphore that blocks the client thread calling DiSendRequest until the semaphore is cleared by the server
- other pointers used by either the client or server

The header also includes an array of one or more structures of type DIRINFORESULT, one for every filespec contained in the request (see **Figure 3**). One of these structures is automatically included in a header segment when the segment is allocated by DiMakeRequest. An application can subsequently call DiMakeRequest for each filespec to be included in the request. When a new filespec is included, DiMakeRequest resizes the header segment (using DosReallocSeg) to include space for an additional DIRINFORESULT structure.

Each DIRINFORESULT structure contains the following information about a particular filespec:

- the filespec for which to search
- the path to the filespec
- the attributes to use when searching for files
- a pointer to the first file that the server finds matching the filespec
- the number of files found by the server for the filespec
- a pointer used by the functions that retrieve the results from the work segment

Creating a DI Request

An application program should call DiMakeRequest to create a directory information request. The DiMakeRequest function is part of DI.C (see **Figure 4**). DiMakeRequest requires the address of a variable that will be a handle to the request, the complete path and filespec that the server will expand, and the attributes that the server should use when expanding the filespec. The handle is of type PVOID and should be initialized to NULL before the first call to DiMakeRequest, so that the function will create a new request. Otherwise, the function will assume that the handle refers to an existing request. You can add filespecs to the request by repeated calls to DiMakeRequest with the same

Figure 2: REQUESTHEADER Structure

```
typedef struct _requestheader
    {
    ULONG           RAMsem;          /* RAM semaphore for client     */
    SEL             rselector;       /* Client selector to results   */
    HQUEUE          qhandle;         /* Handle to server's queue     */
    PID             qowner;          /* PID of server (queue owner)  */
    VOID FAR        *resultptr;      /* Server pointer to work area  */
    SEL             serverhsel;      /* Server selector to header    */
    SEL             serverwsel;      /* Server selector to results   */
    PCH             currentdir;      /* Client's dir in work area    */
    PCH             requestspec;     /* Next part of work area       */
    USHORT          size;            /* Current size header segment  */
    USHORT          resultsize;      /* New size of result segment   */
    USHORT          totalresults;    /* Total results found          */
    USHORT          numRequests;     /* Number of requests being made */
   DIRINFORESULT    resultArray[1];  /* Request structures           */
   } REQUESTHEADER;
typedef REQUESTHEADER FAR *PREQUESTHEADER;
```

Figure 3: DIRINFORESULT Structure

```
typedef struct _dirinforesult
    {
   USHORT                  attributes;   /* Attributes this request   */
   PCH                     filespec;     /* File spec to use          */
   PCH                     currentdir;   /* Current dir this request  */
   PFILEFINDBUF            firstfile;    /* First result              */
   PFILEFINDBUF            nextfile;     /* Next result               */
   USHORT                  numfound;     /* Number of files found     */
   USHORT                  errorval;     /* Error value returned      */
  struct _dirinforesult    *next;        /* Related structure if found */
   } DIRINFORESULT;
typedef DIRINFORESULT FAR *PDIRINFORESULT;
```

Figure 4: DI.C Source Code

```
#define   INCL_DOS
#define   INCL_ERRORS

#include<os2.h>
#include<mt\stdio.h>
#include<mt\string.h>
#include<mt\stdlib.h>
#include<mt\ctype.h>
#include"errexit.h"

#define DICODE
#include"di.h"

#define lastchar(str)   (str[strlen(str)-1])

void    diInit(PID *qowner, HQUEUE *qhandle);
void    adddriveletter(PCH *filespecs,USHORT driveno);
void    makefpath(char *org,char *result,char *currentpath,
                  USHORT currentdrive);
void   getdriveinfo(USHORT *currentdrive,char *currentpath,
                    USHORT *psize);
USHORT  diallocseg(USHORT size, SEL *oursel,PID other,SEL *othersel);
void    convertptr(VOID **ptr,SEL newsel);

   /* DiMakeRequest
   This function creates or adds to an existing directory information
   request. A new request is created if the pointer (hptr) is set to
   NULL. Note that this is the pointer whose address is passed to the
   function. The filespec is a full path and file specification with
   optional wildcards. The attribute parameter will control the files
    that are found. */

void DiMakeRequest(PREQUESTHEADER *hptr, PCH filespec, USHORT att)
    {
    PREQUESTHEADER   header;
    PDIRINFORESULT   resultstru;
     SEL             hselector,serversel;
     void far        *results;
     USHORT          retval,size,psize = 0,setdir = FALSE;
     HQUEUE          qhandle;
     PID             qowner;
     PCH             requestspec;
     char            resultbuf[_MAX_DIR],*resultfspec;
     char            currentpath[_MAX_PATH];
     USHORT          currentdrive;

    if(!(header = *hptr))
      diInit(&qowner, &qhandle);

    /* Now gather the information to prepare the header:
      • Get the current disk drive
      • Get the length of the current path
      • Get the size of the request arguments (the requestspec)
      • Allocate the header segment
      • Make it available to the server process */

   getdriveinfo(&currentdrive,currentpath,&psize);

     if(!header)                              /* If no header is allocated */
       {
                                              /* Allocate header segment */
     if(retval = diallocseg(size = sizeof(REQUESTHEADER),&hselector,
                            qowner,&serversel))
       error_exit(retval,"diallocseg");

      header = MAKEP(hselector,0);            /* Make header pointer */
      *hptr = header;
      header->serverhsel = serversel;         /* Allocate work segment */
     if(retval = diallocseg(MAXSEGSIZE,&header->rselector,qowner,
                            &serversel))
       error_exit(retval,"diallocseg");

     results = MAKEP(header->rselector,0) /* Make pointer */

      header->RAMsem = 0L;                    /* Initialize semaphore */
     header->resultptr = MAKEP(serversel,0); /* Set pointer to work area */
```

CONTINUED

handle before you call DiSendRequest.

When you are ready to send a request to the server, your application should call DiSendRequest. Remember that this call will block until the server has finished processing the request, so you should place this call in a separate thread if you need to avoid blocking the main thread of a client application. You should either call DiDestroyRequest before reusing the handle or use a different handle for subsequent requests.

DiMakeRequest assumes that the server is running and has created its queue; otherwise, the function will wait until the server's queue has been created. Once the server has created its queue, DiMakeRequest will open it using DosOpenQueue, which returns a queue handle and the PID of the server. DiMakeRequest will use DosGiveSeg and the server's PID to enable the work and header segments to be shared with the server. DiMakeRequest will add each filespec to the request, expand it to include a full path, and create a new DIRINFORESULT structure for each filespec. The details of this process and how the DI functions work with the DosFindFirst OS/2 function are discussed in the sidebar "DosFindFirst and DI Memory Management."

Once an application has finished adding filespecs to a request, it can then use DiSendRequest to send the request to the server (see **Figure 2**). The DiSendRequest function does some housekeeping, writes the request to the queue, then waits until the server has completed its task and clears the header semaphore.

DISERVER Program

The DISERVER program

handles requests for directory information from client processes that pass these requests to the server by using its queue. DISERVER is also a thread server—it creates a new thread to process every request it receives. DISERVER.C, which contains the entire server program, consists of two functions: main, which waits for requests and then starts a new thread to service them, and server_thread, which contains the code that services the request.

The main thread of DISERVER is fairly simple. It creates its queue and blocks on a call to DosReadQueue to wait for requests from client processes. Once it receives a request, it finds an available thread and creates a specific thread to service the request. It uses an array of SERVERTHREAD structures that contains a semaphore, a thread ID, a request pointer, a thread number, and a stack for each thread. To find an available server thread, the main thread simply looks for an unused structure (identified by a thread ID set to zero) in this array. Then it sets the thread's semaphore and creates the new thread. The thread will immediately block on the semaphore, allowing the main thread to set the thread's ID, request pointer, and number. Then the main thread clears the thread's semaphore and returns to the DosReadQueue call to wait for the next request.

There is no need for DISERVER to provide any visual output in routine use, so you can run it in a PM window, a full-screen session, or in a background screen group. I occasionally found it useful (and fascinating) to watch its progress, however, so I left a number of messages embedded in the code. These will print if you start DISERVER with the /v (verbose) parameter.

Figure 4 CONTINUED

```
      header->serverwsel = serversel;      /* Selector to work area */
      header->numRequests = 0;             /* Set number of requests */

     requestspec = (PCH)results;           /* Set to work area */
      header->currentdir = requestspec;    /* Set pointer to it */
      *requestspec++ = (char)(currentdrive + 'A' - 1);
                                           /* Add drive letter */
      strcpy(requestspec,":\\");           /* And ":\" */
      requestspec += 2;                    /* Move pointer past them */

    DosQCurDir(currentdrive,requestspec,&psize);
                                          /* Add current directory */
     requestspec += (strlen(requestspec)+1); /* Move pointer past dir */
      header->qowner = qowner;
      header->qhandle = qhandle;
       }

     else
       {
      requestspec = header->requestspec;
      qowner = header->qowner;
      qhandle = header->qhandle;

     size = header->size + sizeof(DIRINFORESULT);
     hselector = SELECTOROF(header);

     if(retval = DosReallocSeg(size,hselector)) /* Resize segment */
       error_exit(retval,"DosReallocSeg");
       }

         /* Resultstru always points to the next available structure */
   resultstru = &header->resultArray[header->numRequests];
     strupr(filespec);                    /* Set arg to upper case */
   memset(resultstru,0,sizeof(DIRINFORESULT)); /* Clear structure */
     resultstru->attributes = att;        /* Set attributes */
                                          /* Get full path filespec */
   makefpath(filespec,resultbuf,currentpath,currentdrive);
    resultfspec = strrchr(resultbuf,'\\');   /* Find last backslash */

    if(strcmp(resultbuf,header->currentdir))  /* If not in currentdir */
       {
       strcpy(requestspec,resultbuf);     /* Copy path */
      resultstru->currentdir = requestspec;   /* Set pointer */
      requestspec += (strlen(requestspec)+1);
                                          /* Set pointer to next spot */
       }

     else                                 /* Use default directory */
      resultstru->currentdir = header->currentdir;  /* Set pointer */

    strcpy((char *)requestspec,resultfspec);  /* Copy the filespec */
     resultstru->filespec = requestspec;      /* Set a pointer to it */
    requestspec += (strlen(requestspec)+1);   /* Set to next position */

     header->size = size;
    header->numRequests++;
    header->requestspec = requestspec;
     }

void DiSendRequest(PREQUESTHEADER hdr)
     {
    PCH *resptr,*newptr;
    USHORT offset,retval,i;
   SEL newsel,serverwsel = SELECTOROF(hdr->resultptr);
    PBYTE sheader = MAKEP(hdr->serverhsel,0);  /* Make pointer */

         /* Adjust resultptr to point to available space */
     resptr = MAKEP(hdr->rselector,0);        /* Create ptr to result */
    offset = (hdr->requestspec - (PCH)resptr); /* Get offset to use */
   hdr->resultptr = MAKEP(serverwsel,offset); /* Reset pointer */

         /* Write request to the queue */
  if(retval = DosWriteQueue(hdr->qhandle,0,hdr->size,(PBYTE)sheader,0))
     error_exit(retval,"DosWriteQueue"); /* Wait for server to finish */

  DosSemSetWait(&hdr->RAMsem,SEM_INDEFINITE_WAIT); /* Get new segment */
```

CONTINUED

Figure 4 CONTINUED

```
  if(retval = DosAllocSeg(hdr->resultsize,&newsel,SEG_NONSHARED))
    error_exit(retval,"DosAllocSeg");

   newptr = MAKEP(newsel,0);                /* Make pointer to segment */
  memmove(newptr,resptr,hdr->resultsize); /* Copy from old segment */
  convertptr(&hdr->currentdir,newsel);   /* Convert pointers */
    convertptr(&hdr->requestspec,newsel);

   for( i = 0; i < hdr->numRequests; i++)
     {
   convertptr(&hdr->resultArray[i].filespec,newsel);
   if(SELECTOROF(hdr->resultArray[i].currentdir) == hdr->rselector)
   convertptr(&hdr->resultArray[i].currentdir,newsel);
   convertptr(&hdr->resultArray[i].firstfile,newsel);
   convertptr(&hdr->resultArray[i].nextfile,newsel);
      }

   DosFreeSeg(hdr->rselector);          /* Free old segment */
   hdr->rselector = newsel;             /* Set for new selector */
    }

void DiGetNumResults(PREQUESTHEADER header,USHORT *numresults,
                     USHORT *numrequests)
    {
  *numresults = header->totalresults;
  *numrequests = header->numRequests;
    }

void DiDestroyRequest(PREQUESTHEADER *header)
    {
   USHORT retval;
   PREQUESTHEADER hdr = *header;

   if(retval = DosFreeSeg(hdr->rselector))  /* Free work segment */
    error_exit(retval,"DosFreeSeg");
   if(retval = DosFreeSeg(SELECTOROF(hdr))) /* Free header segment */
    error_exit(retval,"DosFreeSeg");

    *header = NULL;                          /* Set pointer to NULL */
    }

char *DiGetResultFspec(PDIRINFORESULT result)
    {
    static char *p = "*.*";

   if(result->errorval == DIREXPANDED)
        return p;
   return result->filespec;
    }

char *DiGetResultDir(PDIRINFORESULT result)
    {
    static char dirbuf[80];

   if(result->errorval == DIREXPANDED)
        {
       strcpy(dirbuf,result->currentdir);
        strcat(dirbuf,"\\");
       strcat(dirbuf,result->filespec);
        return dirbuf;
        }

   return result->currentdir;
    }

void DiGetResultHdl(PREQUESTHEADER header,USHORT requestnum,USHORT *num,
PDIRINFORESULT *resulthdl)
    {
  *resulthdl = &header->resultArray[requestnum];
   *num = (*resulthdl)->numfound;
    }

void DiGetFirstResult(PDIRINFORESULT result, char *buffer)
    {
   result->nextfile = result->firstfile;
  DiGetNextResult(result,buffer);
    }
```

CONTINUED

Server Threads

Each thread started by the server will block on its semaphore until the semaphore is cleared by the server's main thread. Then the server thread will enter a loop, calling DosFindFirst for each filespec in the request. The server thread does not have to call DosFindNext, since the thread provides a large buffer in which DosFindFirst can place its results. DosFindFirst creates a FILEFINDBUF structure for each filename found; these structures are placed in a buffer area that is specified when the function is called.

If DosFindFirst indicates that no files are found or that the path doesn't exist, the returned error value is placed in the DIRINFORESULT structure for that filespec. If only one file is found and the file's attribute byte has its directory bit set, the thread will try the call to DosFindFirst again, but this time with "*.*" appended to the filename, allowing it to expand to a single matching directory. Otherwise, DosFindFirst will place the resulting filenames in the work segment area. A pointer to this area and a variable indicating the remaining space are adjusted with each pass through the loop, so that subsequent calls to DosFindFirst will not overwrite information from previous calls.

If a call to DosFindFirst is successful, the server thread will set the fields of that filespec's DIRINFORESULT structure for the first file found and the number of files found. The server thread will also adjust the total number of files found in the request header. Then the thread enters a loop to adjust the buffer pointer past the last file found (if verbose mode is on, it will also print each filename as it passes through the loop), sleep for one

second, and return to the top of the loop in order to process the next filespec.

Once all filespecs have been processed, the thread will use pointer arithmetic to calculate the total space used in the work segment. Then it frees the work segment, clears the header semaphore to notify the client that it is finished with the request, and frees the header segment. The server thread then sets its own thread ID to zero, thus notifying the main thread that it is terminating (and allowing the main thread to reassign the server thread's data area to a new thread). Finally, it will call DosExit to terminate itself.

Return to the Client

When the server clears the request header semaphore, the client thread that called DiSendRequest and blocked on the semaphore can proceed. The thread using DiSendRequest allocates a new work segment, copies the data and adjusts the pointers, and frees the old work segment. Then it returns to the calling thread.

At this point, the client can retrieve the request results. Even though the client has only made a call to DiMakeRequest for each filespec and a single call to DiSendRequest, the results are in; the server has actually done all the real work.

The client can use several functions to access the results of a request. In DISIMPLE.C, a call is made to DiGetResultHdl to get a handle to the results of filespec zero, since this was the first (and only) filespec in the request. An application can call DiGetNumResults, however, to find out how many filespecs were placed in a request (and, incidentally, how many filenames were found) and then call DiGetResultHdl for each of the filespecs. Once it has obtained a handle to a result, an application

Figure 4 CONTINUED

```
void DiGetNextResult(PDIRINFORESULT result, char *buffer)
    {
   if(!result->nextfile)
      {
      *buffer = '\0';
      return;
      }

  strcpy(buffer,result->nextfile->achName);
   result->nextfile =
  (PFILEFINDBUF)(&(result->nextfile->cchName)+
                  result->nextfile->cchName+2);
    }

void DiGetFirstResultPtr(PDIRINFORESULT result, PFILEFINDBUF *ptr)
    {
   result->nextfile = result->firstfile;
  DiGetNextResultPtr(result,ptr);
    }

void DiGetNextResultPtr(PDIRINFORESULT result, PFILEFINDBUF *ptr)
    {
   if(!result->nextfile)
      {
      *ptr = NULL;
      return;
      }

   *ptr = result->nextfile;
   result->nextfile =
  (PFILEFINDBUF)(&(result->nextfile->cchName)+
                  result->nextfile->cchName+2);
    }

void DiBuildResultTbl(PREQUESTHEADER header, PFILEFINDBUF **table)
    {
    SEL tablesel;
    USHORT retval, i;
    PFILEFINDBUF f,*temp;

  if(retval = DosAllocSeg((header->totalresults*sizeof(PFILEFINDBUF)),
                          &tablesel, SEG_NONSHARED))
    error_exit(retval,"DosAllocSeg");

   temp = MAKEP(tablesel,0);
   f = header->resultArray[0].firstfile;

   for( i = 0; i < header->totalresults; i++)
      {
      temp[i] = f;
     f = (PFILEFINDBUF)(&(f->cchName)+f->cchName+2);
      }

   *table = temp;
    }


void DiDestroyResultTbl(PFILEFINDBUF **table)
    {
    USHORT retval;
  if(retval = DosFreeSeg(SELECTOROF(*table)))
    error_exit(retval,"DosFreeSeg");
     *table = NULL;
    }

void diInit(PID *qowner, HQUEUE *qhandle)
    {
    USHORT    retval;

/* Try to open the queue to the directory server */
  if(!(retval = DosOpenQueue(qowner,qhandle,DIRINFOQNAME)))
     return;
   if(retval != ERROR_QUE_NAME_NOT_EXIST)
    error_exit(retval,"DosOpenQueue");
    else
```

CONTINUED

Figure 4 CONTINUED

```
        error_exit(retval,
                   "DosOpenQueue - Server probably hasn't opened queue");
    }

void convertptr(VOID **ptr,SEL newsel)
    {
   USHORT offset = OFFSETOF(*ptr);
   *ptr = MAKEP(newsel,offset);
    }

/* Assumes that globals, currentpath and currentdrive are properly
  initialized */
void makefpath(char *org,char *result,char *currentpath,
               USHORT currentdrive)
    {
  char drive[_MAX_DRIVE], dir[_MAX_DIR], fname[_MAX_FNAME],
               ext[_MAX_EXT];
   char currdir[_MAX_DIR], *backup, *outdir;
   USHORT driveno,cdirsize = _MAX_DIR-1, retval;
    strupr(org);                              /* Make the path uppercase */
   _splitpath(org,drive,dir,fname,ext);     /* Get path components */

   /* If we have a full path from the user at this point, we don't have
    to do anything more—whatever they ask for, they get. If we don't have
    a full path, we will need to get the full path to the current
    directory of that drive, and then reconcile the working directory or
    parent directory to end up with the "real" path to the file. However,
    if the path is on the same drive, we can use currentdir and save the
    time of a DosQCurDir call. */

    if(!(*drive))                             /* If no drive letter */
      {
     driveno = currentdrive;
     *drive = (char)(currentdrive+'A'-1);
     strcpy(&drive[1],":");
      }
    else
     driveno = (*drive-'A'+1);

    if(*dir != '\\')                          /* If not a full path */
      {
     if(driveno != currentdrive)
        {
       *currdir = '\\';
      if(retval = DosQCurDir(driveno,&currdir[1],&cdirsize))
         error_exit(retval,
                    "DosQCurDir - probably invalid drive or directory");
        }

      else
      strcpy(currdir,&currentpath[2]);

     if(lastchar(currdir) != '\\')
       strcat(currdir,"\\");

     strcat(currdir,dir);
     strcpy(dir,currdir);
      }

   /* dir now has full path to the filespec, reconcile and restore */
   while(backup = strstr(dir,"\\..\\"))    /* Remove any "\..\" */
      {
     for(outdir = backup-1; (*outdir != '\\') && (outdir > dir);
          outdir--);
                                          /* Now outdir is '\' dest */
      backup += 3;    /* now backup is source '\'*/
      strcpy(outdir,backup);
       }

   while(backup = strstr(dir,"\\.\\"))          /* Remove any "\.\" */
      {
     outdir = backup;
     backup += 2;
     strcpy(outdir,backup);
      }
```

CONTINUED

can either call DiGetFirstResult and DiGetNextResult to retrieve the name of each file found, or DiGetFirstResultPtr and DiGetNextResultPtr to retrieve a pointer to the FILEFINDBUF structure returned by OS/2 for that filename. Although the FILEFINDBUF structures are not in an array, a client can call DiBuildResultTbl to create an array of pointers to the results of a request. The use of DiBuildResultTbl will be demonstrated in one of the client programs discussed below.

An application should call DiDestroyRequest when it has finished accessing the results of a DiSendResult call. DiDestroyRequest will call DosFreeSeg to free the work and header segments and permit OS/2 to discard the segments. DiDestroyRequest will reset the request handle to NULL, too.

The next part of this article is a discussion of two client applications—the LS utility and the DIPOP program.

LS Utility

The LS utility has an unusual history. I first wrote LS as an add-on utility for a UNIX®-like DOS[2] shell. When I began programming for OS/2, one of the first assignments I gave myself was to port my favorite tools from DOS to OS/2. I had always found the UNIX LS program more useful and informative than the DOS DIR command, so my DOS-based imitation of the LS program was one of the tools I ported to OS/2.

Once the DI routines were complete, I found it a trivial task to modify LS to use them—and found myself cutting away a great deal of the existing code, now that the complexities of DosFindFirst and DosFindNext were absent. LS not only illustrates how easy it is to incorporate and use the DI routines in an application, it also demon-

strates the portability of C; after all, the program imitates a UNIX utility, was written for DOS, and was ported to OS/2.

LS, like its UNIX counterpart, takes a series of command-line filespecs, expands them, and prints the results of the expansion. One or more options can be specified that control the output of the program (sorting, printing in columns, paging, filtering out certain types of files, and so on).

LS is only slightly more complex than the DISIMPLE program. It uses the read_options function to read any command-line arguments, sets the options, and removes the options from the command line. Then it enters a loop and calls DiMakeRequest for each of the command-line filespecs. Finally, it calls DiSendRequest to issue the request to the server.

Upon returning from DiSendRequest, LS calculates the remaining space on the drive on which the first filespec is found. To do this, it calls DiGetResultHdl to get a handle to the first filespec and DiGetResultDir to access the full path to that filespec, which is another convenient use of the DI functions.

In order to access, sort, and print the results, LS calls DiBuildResultTbl to build a table of the results. This function will create an array of pointers to each of the FILEFINDBUF structures placed in the work segment by the server. DiBuildResultTbl returns a pointer to the results table that the calling thread can use to access these structures. LS passes the table to the C library qsort routine, which sorts the pointers in preparation for printing the file information. (The details of the sort comparisons are contained in the function qscmp, which can sort the pointers by the date-time stamp of the file or the filename.)

Figure 4 CONTINUED

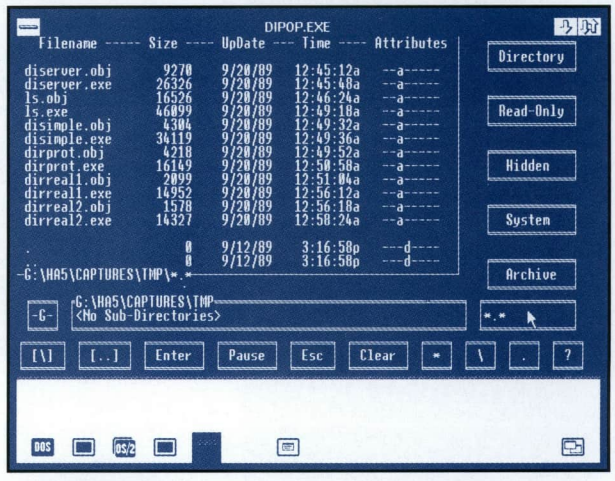
```
    _makepath(result,drive,dir,fname,ext);  /* Put it all back together */
     }

void getdriveinfo(USHORT *currentdrive,char *currentpath,USHORT *psize)
     {
     ULONG  drivemap;
     USHORT retval;

    DosQCurDisk(currentdrive,&drivemap);    /* Get current drive number */
    *currentpath = (char)(*currentdrive+'A'-1);
    strcpy(&currentpath[1],":\\");

     *psize = _MAX_PATH;
    if(retval = DosQCurDir(*currentdrive,&currentpath[3],psize))
      error_exit(retval,"DosQCurDir");

    DosQCurDisk(currentdrive,&drivemap);    /* Get current drive number */
     *psize = 0;
    DosQCurDir(*currentdrive,NULL,psize);   /* Get size of current path */
     }

USHORT diallocseg(USHORT size, SEL *oursel,PID other,SEL *othersel)
     {
     USHORT retval = 0;
                                            /* And shareable by server */
   if(!(retval = DosAllocSeg(size,oursel,SEG_GIVEABLE)))
     retval = DosGiveSeg(*oursel,other,othersel);

     return retval;
     }
```

LS calls the print_entries function in order to access the table and print the filenames. Print_entries can display the output in a long listing (the default), with name, date, time, size, and attributes, or as filenames only, in a single column or multicolumn listing. It can also wait for a keystroke at the end of each screenful of information and will automatically detect the screen size of a window. When it has finished displaying the results, LS calls DiDestroyResultTbl in order to free the table and DiDestroyRequest to free the header and work segments.

DIPOP Program

The DIPOP client program is more complex. It uses the DI functions to access the server, expand filespecs, and retrieve directory information; it has a user interface, and it uses multiple threads to manage input and output.

Like LS, DIPOP is a character-mode directory information program. But DIPOP is not a



▲ **Figure 5** **The user interface for DIPOP, a client program that uses the DI functions.**

USING MESSAGE QUEUES AND MOUSE BUTTONS MINIMIZES THE OVERHEAD OF THE MAIN THREAD AND EFFICIENTLY DELEGATES THE MANAGEMENT OF INPUT TO THE KEYBOARD AND MOUSE THREADS.

command-line utility; it has a real user interface and you can leave it running in a PM window and use it as needed. DIPOP handles only one filespec at a time, but you can use either the keyboard or the mouse to select the drive and directory and to create the filespec used to search for files. You can also select a combination of attributes to be used. To facilitate mouse input in a character-mode application, I've adapted the screen buttons that were used in an earlier article in this series ("Exploring the Key Functions of the OS/2 Keyboard and Mouse Subsystems," *MSJ* Vol. 4, No. 4). Finally, DIPOP implements a message queue scheme that is reminiscent of the one found in Presentation Manager (see the sidebar "Create Message Queues Without Using Presentation Manager").

DIPOP's architecture is inherently multithreaded. Separate threads monitor mouse and keyboard input and another thread displays the results of each filespec request in a portion of the screen. The main thread starts each of the other threads, reacts to information from the keyboard and mouse threads, creates and sends requests to the DI server, and notifies the print thread when the server has returned results.

Two key components are essential to the architecture of DIPOP. The first is the message queues, which the mouse and keyboard thread use to notify the main thread of input events and which the main thread uses to notify the print thread that output is available. The other is the use of screen buttons, which are kept in a table (an array of structures) in DIPOP.C. Several routines, found in BUTTON.C, are available to initialize, display, paint and repaint the buttons, and to notify a thread when a mouse click occurs inside them. Each button is assumed to be three screen rows high, use a double border on the top and bottom and a single border on the sides, and be painted as white foreground on red background when highlighted (this is controlled by a macro at the top of the source file). The structure for each button includes the text to be displayed in the button, the button's row and column coordinates, and a button attribute for repainting the button. A variable controls whether the button is a press button (turns briefly on and then off when clicked), a toggle button (turns off when clicked and on when clicked again), or an input button (does not change state when clicked). Finally, each button contains values that are returned when the left or right mouse button is clicked on it and an accelerator key is pressed (this allows the button to be accessed by using the keyboard).

Using the message queues and mouse buttons minimizes the overhead of the main thread and efficiently delegates the management of input to the keyboard and mouse threads. The buttons also allow the main thread to create an input screen easily, and simplify maintenance and modifications by the programmer. For example, I added a directory attribute button to the interface long after the rest of the code had been written, spending only a few moments and adding only a couple of lines of source code.

How DIPOP Works

The use of the screen buttons and the message queue simplifies the operation of DIPOP immensely. The main thread begins by calling the message queue function, MsgQCreate, to create its message queue. Next, the program clears a semaphore that it will use to control the scrolling by the print thread (signaled by pressing the Pause button). Then the main thread creates each of the three threads and sleeps briefly, to allow them to get started. The main thread then calls a set of customized functions that initialize the buttons and the screen and display the various buttons.

Finally, the main thread enters a loop that contains a large switch statement. It will block on a call to MsgQGet until it receives an event message and will use the switch statement as a table of messages and actions. Unidentified messages are passed, by default, to the buffermgr function, which will attempt to use the message to manipulate the filespec entry field. Thus, once the main thread of DIPOP receives an event message from its message queue, it can process the message swiftly and effectively and take any necessary action.

All mouse and keyboard input in DIPOP is detected by the appropriate thread. The mouse thread uses the ButtonPressed function to screen out unwanted mouse events and to include events for which a screen button was pressed. The keyboard thread uses the function AcceleratorPressed to distinguish button accelerators from other keyboard events. Both of these threads call the message queue function, MsgQSend, to notify the main thread of these events. Since all message queue messages are a single unsigned word, the application is

designed to use the high-order bits in the message to pack additional meaning into the message. This is how ordinary ASCII values, scan codes, and mouse events are encapsulated into messages (see the SCANCODE macro and EVENTCODE macro at the beginning of the program).

Most event messages pertain to building a request for directory information, but there are three exception messages. When the main thread receives an ENTER message, it calls SendRequest to create the request and send it to the DI server. After the server has serviced the request and before the function returns, it places the request on the print thread's message queue, notifying the print thread that there is work to be done. If the main thread receives a PAUSE_EVENT message, it will toggle the state of the semaphore that controls the processing by the print thread. This lets the main thread control the print thread instantly, allowing the user to control the scrolling of the display temporarily. If the main thread receives an ESCAPE_EVENT message, it will close its message queue, destroy any existing DI request, clear the screen, and exit.

User's View of DIPOP

Figure 5 shows what the user sees when using DIPOP. The screen is divided into two sections: half of the screen area is in a window that extends from the upper-left corner of the screen, and the other half is filled with buttons and runs along the right and bottom sides of the screen.

To build a directory request, users just type a filespec (at the cursor position in the entry field) and select the drive and directory in which the server will search for the filespec. To select the drive, the mouse is used to click the drive button. Clicking the left mouse button will decrement the drive letter choice; clicking the right button will increment it. As users change drives, the current directory for each appears along the upper border of the directory button. The path that appears is always the one that is used in a DI request. The directory button can be used to see into a list of subdirectories in the current directory. Users can click on this button to view the entries in the list. Clicking the right mouse button on the [..] button changes to a directory; clicking the left mouse button on the [..] button changes to the parent directory. When changing directories, the path on the upper border of the drive button will reflect the change. The [\] button can be used to jump to the root directory of any drive.

In addition to typing the characters that make up the filespec, users can use the mouse to click the " * ", " \ ", " . ", and " ? " buttons. When one of these buttons is clicked, the program inserts the character represented by the button at the current cursor position in the entry field. Users can control the scope of the search by clicking the attribute buttons that appear on the right side of the screen. They can access the attribute buttons via an accelerator key, which is Alt and the first letter of the button text; for example, Alt-H toggles the Hidden button.

When users are ready to initiate a request for directory information, they press the Enter key or click the Enter button. To pause the scrolling of the results, they click the Pause button (clicking the Pause button again resumes scrolling). Pressing Alt-C or clicking the Clear button clears the entry field and resets the attribute buttons. Users can press Esc or click the Esc button to terminate the program.

Improvements

After designing the DI programs presented in this article, I thought of several improvements that could be made to them. The server could be modified to use named pipes, which would allow it to be used transparently over a network. Moving the results from a memory block that belongs to the server into a memory block that belongs to the client would require some work—but it could be done. As I mentioned, the DI functions could be implemented as dynamic-link libraries.

IF THE MAIN THREAD RECEIVES A PAUSE_EVENT MESSAGE, IT WILL TOGGLE THE STATE OF THE SEMAPHORE THAT CONTROLS THE PROCESSING BY THE PRINT THREAD.

One improvement to DIPOP involves a bug in OS/2 that occurs when it is run in a full-screen window. If DIPOP is run in a full-screen window, the mouse pointer will leave traces of itself on the screen whenever a screen button is clicked. The code that repaints a button (in the ButtonPress routine) simply paints the button, sleeps momentarily, and repaints the button. Unfortunately, there is no coordination between VIO and the mouse driver, so VIO doesn't take into account that the mouse driver might need to know that the mouse pointer should be redisplayed in a new color—thus, traces of the mouse pointer are left on the screen. The only way around this problem is to store the current location of the mouse pointer before doing any screen update that might overwrite it, save what is stored under the mouse pointer, perform the screen update, and restore what was under the

mouse pointer. It's unfortunate that the user has to do this, since this is just what VIO and the mouse driver are supposed to do.

With the advent of OS/2 Version 1.2, there will be opportunities to make some minor modifications to the programs so that they will work transparently with the new High Performance File System (HPFS). These modifications should allow long, free-form filenames, few or no restrictions on path punctuation, new extended attributes, and new access control lists. The changes should also allow the use and display of the additional date-time stamps provided by HPFS.

THE FUNCTIONS IN MSGQ.C ALLOW ONE THREAD TO USE MSGQCREATE TO CREATE A MESSAGE QUEUE. THIS FUNCTION RETURNS A HANDLE THAT THE QUEUE OWNER PASSES TO MSGQGET.

The error checking and exception handling in the DI programs can be improved. The server could be modified to clear all pending semaphores before it exits and to allocate the thread stacks dynamically instead of keeping them in a table.

Other improvements to DIPOP might include better error and exception handling, syntax checking, and a wait indicator. Finally, the most important improvement would be a graphical user interface with real buttons, list boxes, and scroll bars. The next step for the program is obvious: a port to Presentation Manager.

Create Message Queues Without Using Presentation Manager

Early in the design of the DIPOP program, it became apparent that using semaphores to pass information and synchronize input from the mouse and keyboard threads would be awkward and cumbersome. If a semaphore were used, the mouse or keyboard thread might be blocked for too long while waiting for the main thread to acknowledge and act on the event. Plus, users could alternate between clicking mouse buttons and pressing keys; using semaphores to determine the order of events would be messy. It is also possible that multiple events could occur and accumulate quickly—which would be intolerable if the threads were not able to process input efficiently.

A facility was needed that would report keyboard and mouse events in the order in which they occurred. Such a facility would allow the event information to accumulate in that order until the main thread acted on them. In addition, the mechanism would be efficient, simple, and, with luck, elegant.

The solution was the handful of functions in MSGQ.C (see **Figure A**). These functions apply a software interface layer to OS/2 queues and create and manipulate a series of message queues. OS/2 queues are typically used to transfer a pointer to shared memory (this is how the DI client-server functions use them), but they can also transfer a user-defined code in the form of a 2-byte, unsigned word. You can find more information on OS/2 queues in the article on IPC in this series, "A Complete Guide to OS/2 Interprocess Communications and Device Monitors," *MSJ* Vol. 4, No. 5. Since the contents of this user-defined code are determined by the programmer, you can use it to create your own messages, which can be customized from application to application or used over again. In addition, queues can be an efficient way to pass the event code.

Thus, the functions in MSGQ.C allow one thread, the message queue owner, to use MsgQCreate to create a message queue. This function returns a handle that the queue owner passes to MsgQGet, where the thread will block until a message is received and returned. Other threads can send messages to the queue owner thread by opening the message queue with MsgQOpen, which also returns a handle. These threads can pass the handle to MsgQSend, along with a message that is placed in the queue. The message queue functions hide the details of manipulating and using OS/2 queues; a thread only needs the name of the queue to create or open it and it can use the returned handle when referencing the queue.

Message Queues and DIPOP

In DIPOP, message queues are essential to coordinating the activities of the different threads and passing information from one thread to another. Two message queues are created: one is owned by the main thread and is used by the mouse and keyboard threads; the other is the print thread, which uses a message queue to allow access to the results of DI requests from the main thread.

After the main thread has cre-

ated the other threads and completed its initialization, it calls MsgQGet and blocks until it receives an input event message from the keyboard or mouse thread. These threads block on keyboard input (by using KbdCharIn) or mouse input (using MouReadEventQueue). The keyboard thread can quickly identify a keystroke as an ASCII key, scan code, or accelerator key, place the key in the main thread's message queue, and wait for the next keystroke. The mouse thread waits for mouse events, screens out unwanted events (such as mouse movements when no buttons are pressed), identifies important mouse clicks, and places the corresponding messages in the message queue. Then it resumes waiting for the next mouse event. The main thread, meanwhile, remains blocked much of the time, waiting for event messages to appear in its queue and reacting to them when they do appear. The result is that the main thread's code is a large switch statement that resembles the kind of message-processing code found in Windows[3] and in PM programs.

This design worked so well that I decided to use it again when I wrote and tested the print thread code. It was easy to have the print thread create a message queue, block on the call to MsgQGet until the main thread posted a message, and use the message to display the results of a DI request. The messages generated by the keyboard and mouse threads are largely keyboard codes with the high-order bits set to indicate whether they are from the text buttons used in DIPOP. Since a message passed to the print thread is the selector of the request header segment, the main thread uses the SELECTOROF macro to extract the selector from a request header pointer or handle, and that selector is the message that is placed in the queue. The print thread uses the MAKEP macro to convert this selector back into a pointer or handle to the request.

Finally, a welcome side effect of using the message queues in DIPOP was the compactness of the code for the mouse and keyboard threads. With so little to do, these functions stayed small, tight, and efficient.

Advantages

Message queues let you create a message-based, event-driven architecture that simplifies the workings of your program and the organization of your application's code. They let you focus the program's code on reacting to events; for example, user input. This makes the job of writing the code much simpler. (Just make sure you have a way to handle any foreseeable input.) Message queues also make it easier to write more efficient code, since a thread will wait for something to do and then promptly do it.

In DIPOP, message queues are the means of coordinating input and output among multiple threads. You won't have to worry about flags or semaphores or the order of events—that's taken care of for you. Message queues allow each thread to monitor a source of input and independently report events to another thread, and for the events to be stored and retrieved in a simple, orderly fashion, without additional programming overhead.

Message queues also allow more flexibility in designing a program and simplify stepwise refinements to the code. If, for example, a new capability is needed, you simply define the input (such as a keystroke or mouse click), define the steps for processing that input (such as adding a case to the switch statement), and you're done.

A multithreaded program architecture allows for a proper division of the labor; that is, each thread is delegated the responsibility of handling a specific task. Message queues help ensure the smooth operation and flow of information from one thread to another, since they allow each thread to report important events and return to what the thread does best—waiting for the next event to occur.

DosFindFirst and DI Memory Management

The most important benefit of the DI functions and server is that they give applications a simpler, more effective, more efficient interface to the OS/2 file-searching functions. DosFindFirst and DosFindNext fit several application contexts flexibly, but their main purpose is to make it easy to port applications to OS/2 from DOS. Before exploring how to use them more effectively under OS/2, here's a brief look at their DOS predecessors.

MESSAGE QUEUES LET YOU CREATE A MESSAGE-BASED, EVENT-DRIVEN ARCHITECTURE THAT SIMPLIFIES THE WORKINGS OF YOUR PROGRAM AND THE ORGANIZATION OF YOUR APPLICATION'S CODE.

Finding Files under DOS

When a DOS application needs to expand a wildcard filespec, it uses two functions of Int 21h. Typically, the application gains access to the DOS disk transfer area (DTA) and passes the filespec to

function 4Eh of Int 21h. This function places the name, size, attributes, and date-time stamp of the first file found into the DTA. Then the application enters a loop and generates the interrupt again, this time with function 4Fh, and retrieves the information on each file until no more files are found.

This process is illustrated in two real-mode sample programs—DIRREAL1.C and DIRREAL2.C. The former illustrates how the entire process is accomplished in C. The latter shows how high-level additions to Microsoft's real-mode C run-time library simplified the process somewhat.

TO USE DOSFINDFIRST, AN APPLICATION MUST SUPPLY A POINTER TO A FILESPEC, A DIRECTORY HANDLE, AN ATTRIBUTE WORD, A BUFFER FOR THE RESULTS, THE BUFFER LENGTH, AND A VARIABLE SPECIFYING THE NUMBER OF FILES.

Under OS/2, you can create a protected-mode equivalent, like DIRPROT.C. That program demonstrates how the DosFindFirst and DosFindNext functions simplify a port of a DOS program to OS/2 by keeping the logic of the application intact. To port the code, just replace the Int 21h calls with DosFindFirst and DosFindNext, adjust the code to use the OS/2 FILEFINDBUF data structure, and you're done.

Although this approach to searching for files must be followed under DOS, it isn't particularly efficient for OS/2. First, the calling application has to loop through multiple calls to the DosFindNext function and there is no way to determine how many of these calls must be made ahead of time. The loop could take a long time to finish if the number of files is large. Second, the application can process information on only one file at a time, or it must manage memory in an attempt to store all the file information found. There are two accepted approaches to storing all of the information. One is to allocate space for each piece of file information dynamically. The other is to make an assumption about the number of files to be found, allocate the space to hold them, and add to that space when necessary. Either approach can slow down an application if a large number of resulting files is found.

DosFindFirst and DosFindNext

In some OS/2 applications, using a loop with DosFindFirst and DosFindNext is reasonably efficient, especially when you're searching for a single file or when time and memory management are not critical to the calling thread. To use DosFindFirst, an application must supply a pointer to a filespec, a directory handle, an attribute word in which at least one bit should match a bit in the attributes of the resulting files, a buffer in which to place the results, the length of the buffer, and a variable in which the application specifies the number of files to find. (OS/2 will replace the variable with the number of files found.) DosFindNext requires the same parameters, except for the filespec and attribute word.

DI Approach

Although the DOS-like approach to finding files makes it easy to port applications from DOS to OS/2, DosFindFirst can be used more efficiently. Indeed, DosFindFirst is quite capable of returning all the files found in a single call, eliminating the need to call DosFindNext at all. The trick is to provide a memory buffer big enough to hold all the resulting FILEFINDBUF structures.

The DI functions and the DI server make providing that memory buffer easy. They allocate a full 64Kb segment for use in the DosFindFirst call and place all of the results found in this segment. Then they essentially resize the segment, so that when DiSendRequest returns to the calling thread, the application owns only the memory that is necessary to hold the results.

The DI functions assume that all results found for a single request, including those returned by the expansion of multiple filespecs, will fit into a single 64Kb segment. What this means, for example, is that if an application made a request with 10 filespecs and each expanded to 100 files, in which every filename occupied a full 13 characters, the result would occupy 36,000 bytes. (A full 13-character filename with the terminating NULL makes a FILEFINDBUF structure that is 36 bytes long; thus the result is 10 x 100 x 36, or 36,000 bytes.). A more likely situation would be 10 filespecs expanded to 200 files each in which the average length of the filenames was eight characters—in this case, the results would need a total of 62,000 bytes (10 x 200 x 31). This would leave 3535 bytes to spare. Although for most applications 64Kb is a safe assumption, an adjustment should be made under OS/2 Version 1.2 to use multiple work segments and allow for a longer average filename.

Managing Memory

When an application calls DiMakeRequest to create a new request (with a handle initial-

CONTINUED ON PAGE 80

Exploring Dynamic-Link Libraries with a Simple Screen Capture Utility

61

Kevin P. Welch

A **new programming concept** that the Microsoft® Windows™ environment has introduced to personal computers is dynamic linking and dynamic-link libraries (DLLs). In Windows[1], dynamic linking refers to the manner in which a function call in one module is dynamically related to object code in another. DLLs contain a collection of functions that are linked dynamically.

Although DLLs are not directly executable and don't receive messages, they are the building block on which the entire Windows interface is based. However, despite this relationship, DLLs remain somewhat mysterious to many Windows programmers.

The PRTSC utility is a relatively simple application that uses a DLL to capture screen images and place them on the Windows clipboard. When screen capture is activated, the Alt-PrtSc key combination copies a bitmap representation of the client area of the active window, the window frame, or the entire display to the clipboard.

Besides serving as a useful documentation utility for your programs, PRTSC also demonstrates how to employ DLLs, keyboard hook functions, and the Windows clipboard. In fact, this utility is very similar to one in ClickArt® Scrapbook+, published by T/Maker Company; both are powerful and full-featured extensions of the Windows clipboard.

System Hooks

The section of the Windows Version 2.0 *Programmer's Reference* manual on the SetWindowsHook function is very intriguing. This function was not thoroughly documented in previous versions of Windows, and developers were advised to leave it alone until it became a formal part of the documented application program interface (API).

System hooks are particularly interesting because they enable you to intercept various messages before they are processed by Windows or dispatched to an application. With this capability you can check for interesting keystrokes (as PRTSC does), monitor application or system messages, and even record all system events for subsequent playback.

System hooks are necessarily a shared resource; when you install a hook you can affect all applications. Also, because of expanded

Kevin P. Welch is a computer scientist specializing in applied mathematics, robotics and artificial intelligence. President of Eikon Systems, Inc., and a doctoral candidate in applied mathematics, he has written numerous articles on a variety of technical subjects.

SYSTEM HOOKS ARE INTERESTING BECAUSE THEY ENABLE YOU TO INTERCEPT VARIOUS MESSAGES BEFORE THEY ARE PROCESSED BY WINDOWS OR DISPATCHED TO AN APPLICATION. WITH THIS CAPABILITY YOU CAN CHECK FOR INTERESTING KEYSTROKES, MONITOR APPLICATION OR SYSTEM MESSAGES, AND RECORD EVENTS FOR PLAYBACK.

Figure 1: Filter Functions

```
FAR PASCAL FilterFunction( nCode, wParam, lParam )
        int     nCode;
        WORD    wParam;
        LONG    lParam;
{
        if ( nCode == DO_SOMETHING ) {
                /*** DO YOUR PROCESSING HERE ***/
        } else
                DefHookProc( nCode, wParam, lParam, &lpfnOldHook );
}
```

Figure 2: System Hook Events

Keyboard Hook	Any keyboard event
Journal Record Hook	Any message retrieved from the event queue
Journal Playback Hook	Any message played back to the event queue
Application Message Hook	Messages to your application
System Message Hook	Messages to any application in the system
Window Procedure Hook	Messages to a window function (debug only)
Get Message Hook	Messages retrieved by GetMessage (debug only)

Figure 3: CPU Registers for _astart

BX	Requested stack size
CX	Requested heap size
DI	Handle to application instance
SI	Handle to previous application instance
ES	Program segment prefix

Figure 4: CPU Registers for LibInit

DI	Handle to library instance
DS	Library data segment
CX	Requested heap size
ES:SI	Pointer to command line

memory considerations under LIM 4.0, most hook functions must be in DLLs to ensure that they are never paged out and made inaccessible.

Associated with each system hook is a filter function, an application-supplied routine that is installed into a chain of functions and called whenever events of a specified type occur. Filter functions normally have the format shown in the code in **Figure 1**.

The nCode parameter typically specifies whether the filter function should process the information or call the DefHookProc function and pass it back to the system. The wParam parameter provides additional information on the event, usually defining the context in which the event occurred. Finally, the lParam parameter is used to transfer additional information that further clarifies the event in question. This could be a far pointer to a message data structure or simply a set of binary flags that describe the event. Using this filtering scheme, you can define system hooks that intercept and/or process the kinds of events shown in **Figure 2**.

Once defined, system hooks are inserted into the chain with the SetWindowsHook function. When you call this function you provide a code that specifies the type of hook you are installing, followed by a procedure-instance address of the appropriately defined filter function. Note that this function must be exported in the library's module definition file. Although libraries can use a function address directly since they can only have one data segment, applications must use MakeProcInstance to retrieve a procedure instance.

```
lpfnOldHook = SetWindowsHook
            (nHookType,
            (FARPROC) HookFn );
```

The value returned by the SetWindowsHook function is the procedure-instance address of the filter function previously installed in the chain, if any such filter exists. This value should be saved as needed when passing messages down the chain with the DefHookProc function.

Of these system hooks, all but the application message hook must be defined within the context of a DLL. The application message hook intercepts only task-specific messages and is not subject to the same limitations as the other system hooks.

While they are in use, system hooks can seriously degrade the performance of Windows. Because of this, you should restrict their use to special-purpose applications or development tools that help you to debug and test an application. When you have finished using a system hook, you should promptly remove the filter using the UnhookWindowsHook function :

```
UnhookWindowsHook
(nHookType, (FARPROC) HookFn );
```

PRTSC Library

The PRTSC utility consists of a single DLL that contains one assembly language and four conventional C functions. Of these five routines, the keyboard hook function is perhaps the most interesting and unusual.

Although DLLs like PRTSC form the basis of the Windows API, programmers who are new to Windows seldom understand them. This is partly due to the fact that they can be loaded only once and are in effect resources for use by other applications. Further complicating matters is the need to write at least one assembly language routine when developing a DLL.

When you link a conventional Windows application with LINK4, it defines _astart as the program entry point. This function is defined in the standard Windows object library you link to your program and is responsible for performing the required housekeeping chores prior to calling WinMain, the perceived entry point for the application. When _astart is called, the CPU registers are defined as shown in **Figure 3.**

Unfortunately, a DLL like PRTSC requires a different scheme. By design, PRTSC, like all other libraries, operates without a stack segment, using the caller's stack in place of its own. This difference is reflected in the use of LIBRARY instead of NAME for the program name in the module definition file PRTSC.DEF.

When you make this change, LINK4 is instructed to use LibInit as the program entry point in place of _astart and to define the CPU registers as shown in **Figure 4.** LibInit, defined in PRTSC1.ASM (see **Figure 5),** is then responsible for performing any required housekeeping, including initialization of the local heap. Unfortunately, this subtle variation and the subsequent assembly language programming have prevented some people from experimenting with DLLs.

Figure 5: PRTSC1.ASM Source Code

```
; WINDOWS SCREEN CAPTURE - DYNAMIC LIBRARY
;
; LANGUAGE : Microsoft Assembler 5.1
; SDK : Windows 2.03 SDK
; MODEL : small
; STATUS : operational
;
;

        Extrn   PrtScInit:Near

_TEXT   SEGMENT BYTE PUBLIC 'CODE'
        ASSUME  CS:_TEXT
        PUBLIC  LibInit

LibInit PROC    FAR

        Push    DI      ; hInstance
        Push    DS      ; Data Segment
        Push    CX      ; Heap Size
        Push    ES
        Push    SI      ; Command Line

        Call    PrtScInit

        Ret

LibInit ENDP
_TEXT   ENDS

End     LibInit
```

With some extra code, the Windows object library could include an entry point for DLLs that performs this initialization and calls something like LibInit with the library instance handle and a pointer to the command line. The programmer would then write LibInit in his or her favorite language and perform any desired initialization steps before returning control to the system. In the case of PRTSC, LibInit simply calls PrtScInit, which is defined in PRTSC2.C , where all library initialization is performed. See excerpts in **Figure 6.** (*The full source code can be downloaded from any of the* MSJ *bulletin boards—Ed.*)

During initialization PrtScInit checks the current display adapter type, activates the keyboard hook, and iteratively searches for the MS-DOS® Executive window. The search ends when it has found the window, which is identified by the Session class name. Note that

ONCE DEFINED, SYSTEM HOOKS ARE INSERTED INTO THE CHAIN WITH THE SETWINDOWSHOOK FUNCTION. WHEN YOU CALL THIS FUNCTION YOU PROVIDE A CODE THAT SPECIFIES THE TYPE OF HOOK YOU ARE INSTALLING, FOLLOWED BY A PROCEDURE-INSTANCE ADDRESS OF THE APPROPRIATELY DEFINED FILTER FUNCTION. NOTE THAT THIS FUNCTION MUST BE EXPORTED FROM THE LIBRARY'S MODULE DEFINITION FILE.

Figure 6: Excerpts from PRTSC2.C Source Code

```
#include <windows.h>
#include <string.h>
#include "prtsc.h"

/* global data */
BOOL     bMono;                  /* Convert to monochrome?  */
WORD     wArea;                  /* Current capture area    */
WORD     wColors;                /* Number of system colors */
HWND     hWndDOS;                /* Handle to DOS session   */
HANDLE   hInstance;              /* Library instance handle */
FARPROC  lpfnDOSWnd;             /* DOS session function    */
FARPROC  lpfnOldHook;            /* Old hook function       */

/* PrtScInit( hLibInst, wDataSegment, wHeapSize, lpszCmdLine ) : BOOL
   This function performs all the initialization necessary to use the
   screen capture dynamic-link library. It is assumed that no local heap is
   used; therefore there is no call to LocalInit. A nonzero value is
   returned if the initialization is successful. */

BOOL PASCAL PrtScInit( hLibInst, wDataSegment, wHeapSize, lpszCmdLine )
HANDLE   hLibInst;
WORD     wDataSegment;
WORD     wHeapSize;
LPSTR    lpszCmdLine;

{
      extern BOOL       bMono;
      extern WORD       wArea;
      extern WORD       wColors;
      extern HWND       hWndDOS;
      extern HANDLE     hInstance;
      extern FARPROC    lpfnDOSWnd;
      extern FARPROC    lpfnNewHook;

      HDC   hDC;                  /* Handle to temporary DC */
      HWND  hWndFocus;            /* Window that has focus  */
      char  szClassName[32];      /* Temporary class name   */

      /* Initialization - Alt = PrtSc active */
      bMono = FALSE;
      wArea = CAPTURE_WINDOW;
      hInstance = hLibInst;

      lpfnOldHook = SetWindowsHook( WH_KEYBOARD, (FARPROC)PrtScHook );

      hWndFocus = GetFocus();

      hDC = GetDC( hWndFocus );
      wColors = GetDeviceCaps( hDC, NUMCOLORS );
      ReleaseDC( hWndFocus, hDC );
                              :
}

/* PrtScFilterFn( hWnd, wMessage, wParam, lParam ) : LONG FAR PASCAL
  This window function processes all the messages received by the MS-DOS
  session window. When the user selects the Screen Capture... menu option
  this function displays the screen capture control panel. All other
  messages are passed on to the window without modification. */

LONG FAR PASCAL PrtScFilterFn( hWnd, wMessage, wParam, lParam )
HWND  hWnd;
WORD  wMessage;
WORD  wParam;
LONG  lParam;

{

      /* Trap appropriate messages */
      switch( wMessage )
                        {
      case WM_COMMAND : if ( wParam == CMD_CAPTURE )
                        {
                            DialogBox( hInstance, "PrtSc", hWndDOS,
                                       PrtScDlgFn );
```

CONTINUED

lstrcmp is used instead of strcmp because the szClassName variable is defined on the stack and is not directly accessible when using a near pointer (remember that SS != DS in DLLs).

Using the handle to the MS-DOS Executive window, PrtScInit appends a Screen Capture... menu option to the end of the Special pull-down menu and subclasses the entire window. This effectively enables the library to intercept any message sent to the Executive window, including the message generated by our newly appended option.

Although this may not be the most cooperative way to display the screen capture control panel, it demonstrates how one application, in this case a library, can subclass another. However, this is probably not an acceptable programming practice for a commercial application.

The next function in PRTSC2.C is PrtScFilterFn. This routine effectively filters all of the messages sent to the MS-DOS Executive window. The only two messages of interest are the ones generated when the user selects the Screen Capture... menu option or when the window is destroyed.

When subclassing an application with a function like PrtScFilterFn, there are four ways in which messages can be handled, as shown in **Figure 7**. The first message PrtScFilterFn intercepts is the one generated when the user selects the Screen Capture... menu option from the MS-DOS Executive window. In this case it displays a dialog box using the PrtSc dialog box template contained in the resource file appended to the library. Because the CMD_CAPTURE message is of interest only to the PRTSC library, it is not passed to the real window function. Also note that because the PRTSC library, like most other

DLLs, has only one data segment, the dialog box is displayed without creating a procedure instance (using MakeProcInstance) of the dialog box window function.

The second message that PrtScFilterFn intercepts is the WM_DESTROY message. In this case, screen capture is automatically turned off (assuming it is active) and the MS-DOS Executive window filter function is removed. This ensures that screen capture is not left active without a means of controlling it.

Most of the intercepted messages are ignored by PrtScFilterFn and passed to the real window function; otherwise, the window would not work and the system would probably stop.

Following the PrtScFilterFn in PRTSC2.C is the PrtScDlgFn function. This routine is responsible for processing all the messages relating to the screen capture control panel that is displayed when the Screen Capture... menu option is selected. In this function, the keyboard hook function is inserted or removed when the user selects a new screen capture mode. Again, note how the PrtScHook function is used directly, without calling MakeProcInstance to create a procedure-instance address.

The last function in PRTSC2.C is PrtScHook. This is the most complicated routine in the library; it is responsible for intercepting each keystroke and checking whether it is the Alt-PrtSc key combination.

The first thing this function does is examine the nCode parameter to see if some action is expected. If so, the virtual keycode and corresponding keyboard state are examined to see if Alt-PrtSc has been entered. When encountered, the top-level window handle is

Figure 6 CONTINUED

```
                        return( 0L );
                    }
                    break;
        case WM_DESTROY : /* Window being destroyed - unhook
                             everything */
                    if ( wArea )
                    {
                        wArea = CAPTURE_OFF;
                        UnhookWindowsHook( WH_KEYBOARD,
                                           (FARPROC)PrtScHook );
                    }
                    SetWindowLong( hWndDOS, GWL_WNDPROC,
                                   (LONG)lpfnDOSWnd );
                    break;
        default :
        break;
        }

        /* Pass message on to window */
        return(CallWindowProc(lpfnDOSWnd,
                              hWndDOS,wMessage,wParam,lParam) );

}

/* PrtScDlgFn( hDlg, wMessage, wParam, lParam ) : BOOL;
   This function processes all the messages that relate to the PrtSc
   dialog box. This function inserts or removes the keyboard hook
   function, depending on the user's selection. */

BOOL FAR PASCAL PrtScDlgFn( hDlg, wMessage, wParam, lParam )
HWND  hDlg;
WORD  wMessage;
WORD  wParam;
LONG  lParam;

{

        switch( wMessage )
        {
        case WM_INITDIALOG :      /* Initialize dialog box */
            CheckDlgButton( hDlg, DLGSC_MONOCHROME, bMono );
            EnableWindow( GetDlgItem(hDlg,DLGSC_MONOCHROME),
                          (wColors > 2) );
            CheckRadioButton( hDlg, DLGSC_OFF, DLGSC_SCREEN,
                              DLGSC_OFF + wArea );
            break;
        case WM_COMMAND : /* Window command */

            /* Process submessage */
            switch( wParam )
                {
            case DLGSC_OFF : /* Turn screen capture off */
                        if ( wArea )
                        {
                         wArea = CAPTURE_OFF;
                         UnhookWindowsHook( WH_KEYBOARD,
                                            (FARPROC)PrtScHook );
                         EnableWindow( GetDlgItem(hDlg,
                                       DLGSC_MONOCHROME),FALSE );
                        }
                        CheckMenuItem( GetMenu(hWndDOS), CMD_CAPTURE,
                                       MF_UNCHECKED );
                        break;

            case DLGSC_CLIENT : /* Capture client area of active window */
            case DLGSC_WINDOW : /* Capture active window */
            case DLGSC_SCREEN : /* Capture entire screen */
                        if ( !wArea )
                        {
                         lpfnOldHook = SetWindowsHook(WH_KEYBOARD,
                                       (FARPROC)PrtScHook );
                         EnableWindow( GetDlgItem(hDlg,
                                       DLGSC_MONOCHROME),
                                       (wColors > 2) );
                        }
```

CONTINUED

Figure 6 CONTINUED

```
                        wArea = wParam - DLGSC_OFF;
                        CheckMenuItem( GetMenu(hWndDOS), CMD_CAPTURE,
                                       MF_CHECKED );
                        break;
                        case DLGSC_MONOCHROME : /* Capture image in
                                                   monochrome */
                        bMono = !bMono;
                        break;

                    case IDOK :
                        EndDialog( hDlg, TRUE );
                        break;
                    default : /* ignore everything else */
                        break;
                    }

                    break;
            default : /* message not processed */
                    return( FALSE );
                    break;
            }

            /* normal return */
            return( TRUE );

}


/* PrtScHook( nCode, wParam, lParam ) : WORD
   This function is called whenever the user presses any key. The Alt = PrtSc
   key combination is trapped, and a bitmap copy of the desired portion of
   the screen is copied to the clipboard. The return value is FALSE if the
   message should be processed by Windows; the return value is TRUE if the
   message should be discarded. */

WORD FAR PASCAL PrtScHook( nCode, wParam, lParam )
int   nCode;
WORD  wParam;
LONG  lParam;


{
        extern BOOL     bMono;          /* Convert to monochrome? */
        extern WORD     wArea;          /* Area to capture */
        extern FARPROC  lpfnOldHook;    /* Old keyboard hook */
        WORD            uWidth;         /* Width of bitmap */
        WORD            uHeight;        /* Height of bitmap */
        WORD            wDiscard;       /* Return value */
        POINT           ptClient;       /* Client point */
        RECT            rcWindow;       /* Window rectangle */
        HDC             hScreenDC;      /* Handle to screen DC */
        HDC             hMemoryDC;      /* Handle to memory DC */
        HWND            hActiveWnd;     /* Handle to active window */
        HBITMAP         hOldBitmap;     /* Handle to old bitmap */
        HBITMAP         hMemoryBitmap;  /* Handle to memory bitmap */

         if (nCode == HC_ACTION)
            {
         /* This check traps the Alt = PrtSc key combination using bit
          29 for the Alt key and bit 31 for the key being released. */

          if ((wParam==VK_MULTIPLY) && ((lParam&0xA0000000)
                                        ==0xA0000000))

              {
                              :
                              :
             case CAPTURE_WINDOW :
          /* Retrieve active window dimensions */
                        GetWindowRect( hActiveWnd,&rcWindow );

                        break;

             case CAPTURE_SCREEN :
       /* Retrieve dimensions of entire screen */
                        rcWindow.top = 0;
                        rcWindow.left = 0;
```

CONTINUED

determined and the screen coordinates of the capture area calculated. In cases where window boundaries extend beyond the screen dimensions, appropriate clipping is performed to ensure a completely defined image.

The remainder of the function is devoted to copying the screen image and placing it on the Windows clipboard. When this is completed, the clipboard is closed, which causes a WM_DRAWCLIPBOARD message to be sent down the clipboard viewer chain. All associated resources are then removed. If any part of this process fails, a message box is displayed, providing some feedback to the user on the problem that was encountered.

Building PRTSC

Building a DLL is only slightly different from constructing a traditional Windows application. Before doing this, however, you will need to enter or download the following source files: PRTSC (make file), PRTSC.DEF (module definition file), PRTSC.H (header file), PRTSC.RC (resource file), PRTSC1.ASM (assembly language startup code), and PRTSC2.C (C source code).

In addition to these source files, you must install the following software tools on your system: Microsoft Macro Assembler Version 5.0 or later, Microsoft C Compiler Version 5.0 or later, and the Microsoft Windows Software Development Kit (SDK) Version 2.03 or later. When you have all the tools and source files in place, you can create the PRTSC library by entering the following command:

```
MAKE PRTSC
```

Using PRTSC

Once you have created the PRTSC library, try it out by running Windows and double

clicking PRTSC.EXE. When PRTSC is loaded, a small message box will be displayed in the center of your screen indicating that the Alt-PrtSc key combination is available for screen capture to the clipboard.

If you select the Screen Capture... option under the Special pull-down menu of the MS-DOS Executive window, you can view the current settings, turn screen capture off, or select another capture mode. The Screen Capture menu option will be checked whenever Alt-PrtSc is active.

If you run additional MS-DOS Executive windows after loading PRTSC, the Screen Capture... menu option will be listed only on the first instance. If you accidentally close this instance, screen capture will be turned off and you will no longer be able to load the library; only one instance of a library is allowed, and it has already been loaded. With a little effort, you can enhance the PRTSC library to prevent a lock-out by subclassing a second MS-DOS Executive instance when the first window is closed.

Another feature of the PRTSC library you can experiment with is the monochrome conversion option, assuming you have a color display system. As you can imagine, the capture of color screen images to the clipboard can consume large amounts of system memory. Also, many Windows applications handle such bitmaps incorrectly, especially when they are transported in files to computers with different display subsystems.

By choosing the Convert to Monochrome option you can automatically convert the screen images you capture to monochrome by using the conversion routines built into GDI. Although in certain cases this color mapping will produce unexpected results, like

Figure 6 CONTINUED

```
                rcWindow.right = SCREEN_WIDTH;
                rcWindow.bottom = SCREEN_HEIGHT;

                     break;
          }

               /*  Note that the window boundaries that need to be
                   adjusted as part of the window may not be visible on
                   the screen. This eliminates the problem of  a
                   partially defined bitmap. */

                /* Adjust boundaries to screen clipping region */
                     if (rcWindow.right > SCREEN_WIDTH)
                        rcWindow.right = SCREEN_WIDTH;

                     if (rcWindow.bottom > SCREEN_HEIGHT)
                        rcWindow.bottom = SCREEN_HEIGHT;

                     if (rcWindow.left < 0)
                         rcWindow.left = 0;

                     if (rcWindow.top < 0)
                         rcWindow.top = 0;

                /* Compute display size of window */
                     uWidth = rcWindow.right - rcWindow.left;
                     uHeight = rcWindow.bottom - rcWindow.top;

                /* Open clipboard */
                     if ( OpenClipboard(hActiveWnd) )
                     {

                /* Empty clipboard */
                     EmptyClipboard();

                /* Create screen DC & compatible memory DC's */
                     hScreenDC = CreateDC( "DISPLAY", NULL, NULL,
                                           NULL );
                     hMemoryDC = hScreenDC ?
                            CreateCompatibleDC((hScreenDC):NULL;

                /* Create color or monochrome bitmap */
                     if (hMemoryDC && hScreenDC)
                     {
                     hMemoryBitmap = CreateCompatibleBitmap(bMono ?
                                     hMemoryDC :hScreenDC,uWidth,
                                     uHeight);
                     } else
                           hMemoryBitmap = NULL;

                           if ( hMemoryBitmap )
                           {

                /* Select bitmap & copy bits */
                                hOldBitmap = SelectObject
                                             (hMemoryDC,
                                             hMemoryBitmap);
                                BitBlt(
/* Dest DC              */       hMemoryDC,
/* x upper left         */       0,
/* y upper left         */       0,
/* Dest width           */       uWidth,
/* dest height          */       uHeight,
/* Source DC            */       hScreenDC,
/* Source X upper left  */       rcWindow.left,
/* Source Y upper left  */       rcWindow.top,
/* ROP code             */       SRCCOPY);

                             SelectObject(hMemoryDC, hOldBitmap );
                             SetClipboardData(CF_BITMAP,
                                             hMemoryBitmap);

          } else
                    MSGBOX( hWndDOS,"Insufficient memory!");

                    /* Close clipboard */
```

CONTINUED

Figure 6 CONTINUED

```
CloseClipboard();

/* Delete DCs */
if ( hMemoryDC )
   DeleteDC(hMemoryDC);
if ( hScreenDC )
   DeleteDC(hScreenDC);

} else
        MSGBOX(hWndDOS,
             "Unable to open clipboard!" );

} else
        MSGBOX(hWndDOS,
             "Active window is iconic!" );

} else
        wDiscard = FALSE;

} else
        wDiscard = (WORD)DefHookProc(nCode,
                                     wParam,
                                     lParam,
                                     (FARPROCFAR *)
                                     &lpfnOldHook);

/* Return value */
return(wDiscard);

}
```

mapping yellow to black, it will perform acceptably with minor adjustments to your system color palette.

You should also try to capture menus using PRTSC library. If you hold down Alt-spacebar, the system menu of the active window will appear. By keeping Alt depressed and using the cursor movement keys you can bring up other pull-down menus. Still holding Alt down, if you press PrtSc you can copy the contents of the visible menu (perhaps with some other information) to the clipboard. You can then edit out the parts you don't want and include the menu with the documentation associated with your application. ❑

Figure 7: Message Handling

Ignore the message and block it
Ignore the message but pass it on
Handle the message and block it
Handle the message and pass it on

[1]For ease of reading, "Windows" refers to the Microsoft Windows graphical environment. "Windows" refers to this Microsoft product only and is not intended to refer to such products generally.

U.S. Postal Service
STATEMENT OF OWNERSHIP, MANAGEMENT AND CIRCULATION
(Required by 39 U.S.C. 3685)

1A. Title of Publication: Microsoft Systems Journal
1B. Publication No.: 08899932
2. Date of Filing: September 25, 1989
3. Frequency of Issue: Every other month
3A. Issues Published Annually: Six
3B. Annual Subscription Price: $50.00
4. Complete Mailing Address for Known Office of Publication:
666 Third Avenue, 16th Floor, New York, New York 10017
5. Complete Mailing Address of the Headquarters of General Business Offices of the Publisher:
Same
6. Full Names and Complete Mailing Address of Publisher, Editor, and Managing Editor:
Publisher and Editor:
Jonathan Lazarus, 666 Third Avenue, 16th Floor, New York, New York 10017
Managing Editor:
Joanne Steinhart, 666 Third Avenue, 16th Floor, New York, New York 10017
7. Owner:
Microsoft Corporation
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717
Stockholders having more than 1%:
To the company's knowledge, the only persons who beneficially own more than 1% of outstanding common stock are:
William H. Gates III, Paul G. Allen, Steven A. Ballmer, and Jon A. Shirley
8. Known Bondholders, Mortgagees, and other Security Holders Owning or Holding 1% or More of Total Amount of Bonds: None
9. Does not apply
10.

	Average No. Copies Each Issue During Preceding 12 Months	Actual No. Copies of Single Issue Published Nearest to Filing Date
A. Total No. Copies:	50,231	55,307
B. Paid and/or Requested Circulation		
1. Sales through dealers and carriers, street vendors, and counter sales:	0	0
2. Mail Subscription:	36,678	35,948
C. Total Paid and/or Request Circulation (Sum 10B1 and 10B2):	36,678	35,948
D. Free Distribution by Mail, Carrier or Other Means; Samples, Complimentary, and Other Free Copies:	6,168	6,799
E. Total Distribution (Sum of C and D):	42,846	42,747
F. Copies not Distributed		
1. Office use, left over, unaccounted, spoiled after printing:	7,385	12,560
2. Return from News Agents:	0	0
G. Total (Sum of E, F1 and 2—should equal net press run shown in A):	50,231	55,307

11. I certify that the statements made by me are correct and complete.

Steven Pippin, Circulation Director, September 25, 1989

CHECKERS PART I

69

Design Goals for Building a Complete Graphical Application

Charles Petzold

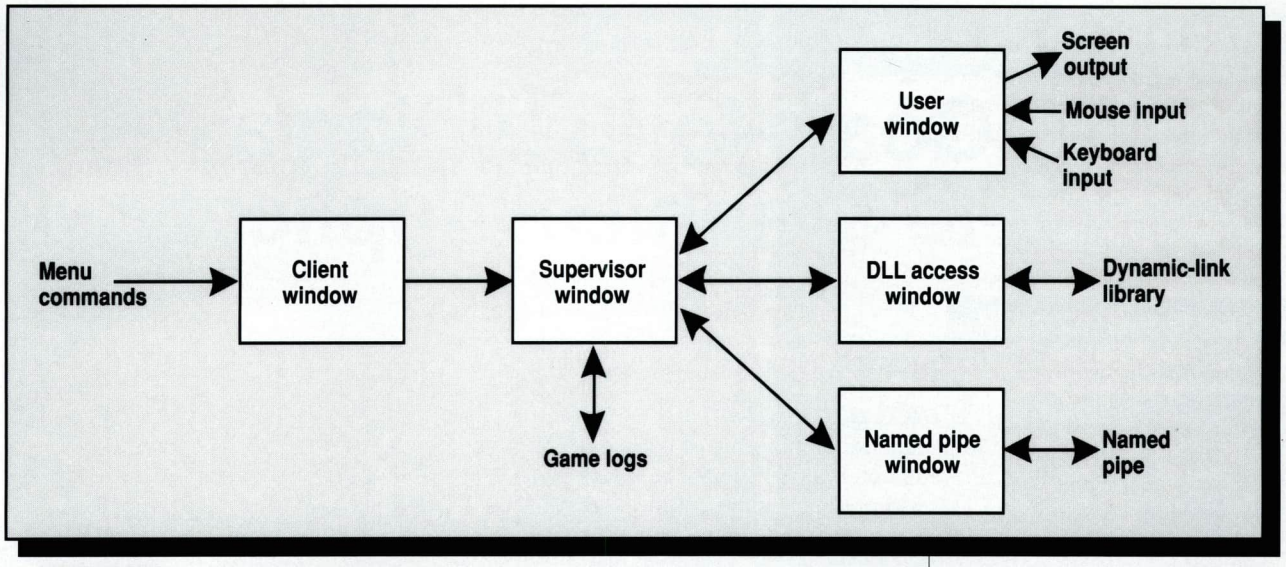
How about a nice game of checkers? Soon you will be able to play a game of checkers under OS/2 Presentation Manager. In the next few issues of *MSJ* we will present a complete checkers program for Presentation Manager (hereafter referred to as PM) called CHECKERS.EXE. You can play CHECKERS against yourself, the program, a person across a network, or an external dynamic-link library that implements a checkers-playing strategy.

Writing a game program is an excellent way to learn about a graphical user interface, because games make use of graphics and are often highly interactive. You can use this article and the following ones to create a fun game that demonstrates many aspects of OS/2 and Presentation Manager programming, including graphics, keyboard and mouse input, menus, dialog boxes, and child windows. All too often the programs published in books and magazines are very short. They show programming techniques in isolation. These articles, on the other hand, will illustrate structure and organization associated with real-world coding, including the use of dynamic-link libraries (for the checkers-playing strategy), named pipes (for playing the game over a network), and multiple threads. We'll look at algorithms for working with logical structures that are common in games, and perhaps even some object-oriented techniques that will help generalize several components of the program. Because of its detail, CHECKERS will be longer than any other program this magazine has published.

▲ **Figure 1** **A CHECKERS game in progress.**

This article describes what I intend to put into the program and some of the problems I expect to encounter. Code with the com-

Charles Petzold is the author of Programming Windows *(Microsoft Press, 1988) and* Programming the OS/2 Presentation Manager *(Microsoft Press, 1988). A copy of the latter is included in Microsoft's OS/2 Programmer's Toolkit.*



▲ **Figure 2** **General CHECKERS program structure.**

ponents of the program that draws the checkerboard and playing pieces, as shown in **Figure 1**, will be published in the next issue. Using the additional user interface code from a future issue, you'll be able to play a game against yourself.

Modes of Play

CHECKERS will feature several basic modes of play available from a menu option. You can:

- Play against yourself, by alternately playing the black and white pieces.
- Play against the computer (the default). The logic of the checkers-playing strategy will be implemented in a dynamic-link library.
- Play against an alternative dynamic-link library. If you would like to code your own checkers-playing strategy, you can create a dynamic-link library and hook into it from CHECKERS.
- Play one dynamic-link library against another. In this case, you just sit back and watch the game.
- Play against another person, running a copy of CHECKERS across a network. This facility will use named pipes.
- Re-create a previous game. CHECKERS will allow you to store a log of a game (using standard checkers notation) as an ASCII file with the extension CKR. The program will also allow you to load a CHR file, to re-create the moves of the game at a speed you define.

Program Structure

Of course, all these options require that some serious consideration be given to the program structure. The best approach seems to be controlling games with a "supervisor" that I intend to implement as a Presentation Manager object window. (Object windows are not visible on the screen, but they can receive and send messages like other windows. You can use an object window to implement object-oriented programming techniques through Presentation Manager architecture rather than through the syntax of your programming language.) The supervisor will be responsible for maintaining the current board layout and keeping a log of the game.

As is usual, menu commands will be processed by the program's client window. When you initiate a new game from the menu, the client window will inform the supervisor that a new game has been requested and who the players will be.

If the game is to be played from a log file, the supervisor will be responsible for re-creating the game. Otherwise, each of the possible players (the user, a dynamic-link library, or a named pipe connection) is known to the supervisor as a window handle. Thus, for each game, the supervisor has two window handles, one for the black player and one for the white player.

Alternating between black and white, the supervisor will use a message to notify a player window when it should make a move. Depending on who the players are, these player windows can get information about a move from the user, a dynamic-link library, or a named pipe.

The player windows will inform the supervisor when a move has been completed. The supervisor can then notify the other player window about the move and request that that win-

dow make a move. The supervisor will also be responsible for determining when a player has won a game, and perhaps even for determining when a game has ended in a draw.

The general program structure (as I conceive it now) is shown in **Figure 2**. The client window and the user window are normal Presentation Manager windows. The supervisor window, dynamic-link library access window, and named pipe window are object windows.

Suppose that you're playing black and that your opponent is a dynamic-link library playing white. In this case, the supervisor window has two window handles: black is the handle of the user window and white is the handle of the dynamic-link library window. The supervisor sends the user window a message telling it to make a move. The user window displays an appropriate mouse pointer and waits for you to make a move. When the move is completed, the user window informs the supervisor window of the move. The supervisor then sends a message to the dynamic-link library window requesting a move. When the dynamic-link library determines what the move should be, it sends a message back to the supervisor window. The supervisor informs the user window of this move so that the display can be updated. The supervisor then sends a message to the user window asking for the next move.

When one of the players is a dynamic-link library or another person over a named pipe, multiple threading will be required while the supervisor window is awaiting word of the next move. The dynamic-link library access window and the named pipe windows will be responsible for creating these threads.

Notation/Representation

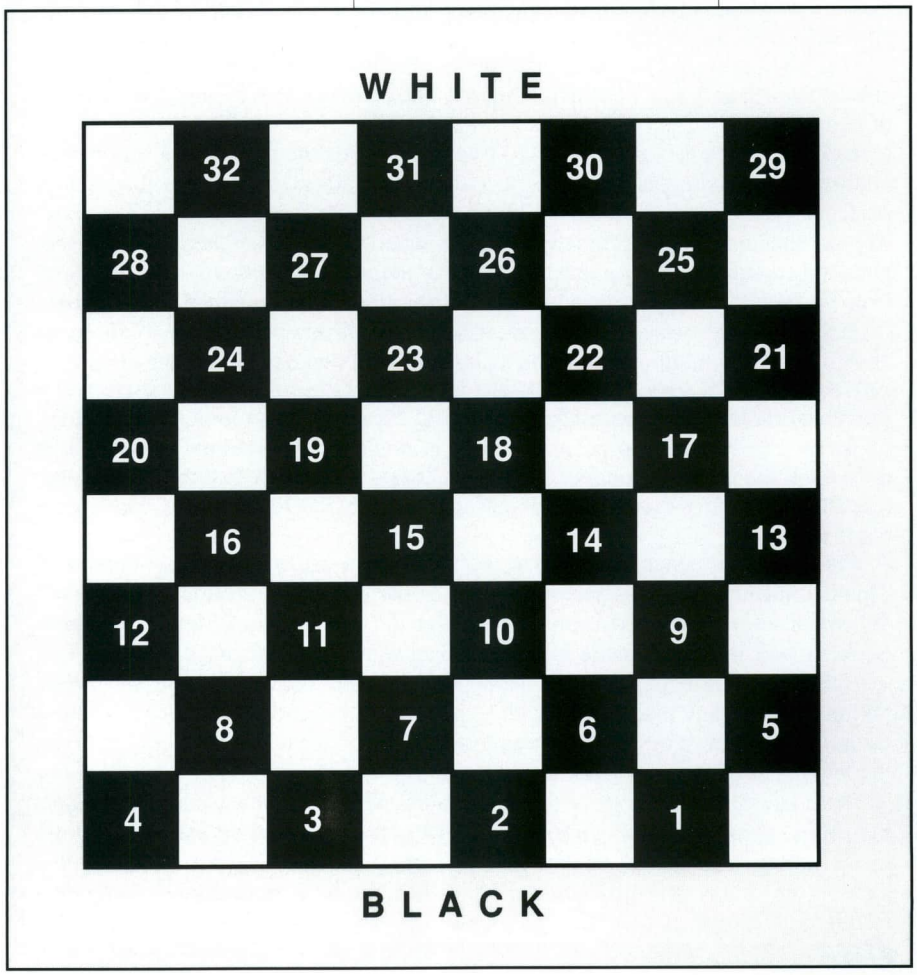
I mentioned earlier that CHECKERS will be capable of storing a log of the game in an ASCII file. This log will use standard checkers notation, which is shown in **Figure 3**. The black squares are simply numbered 1 through 32. The numbering makes more sense if you turn the board around so that black is on top, as is usually done when showing checkerboard layouts in books. (**Figure 3** is shown with black on the bottom to be consistent with **Figure 1**.)

A game can be represented by showing each move with the starting and ending square numbers separated by a dash. Here's the beginning of a game with two jumps in the first four moves:

Black: 10-15
White: 24-19
Black: 15-24
White: 28-19

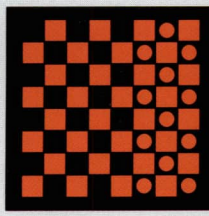
There are only 32 squares on which pieces can reside, making it very convenient to represent the board in a C program. The number of black squares corresponds to the minimum number of bits in a long integer, as required by the ANSI C standard. By simply subtracting 1 from each square number, the board positions can correspond to the bits of a long integer, where 0 is the least significant bit and 31 is the most significant bit.

Such a representation is discussed in Christopher S. Strachey's paper "Logical or

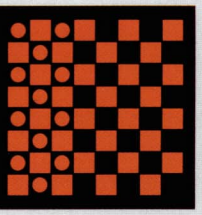


▶ **Figure 3**

Numbering of squares used in standard checkers notation.



The Rules of the Game



The game of checkers (called draughts in Great Britain) has obscure beginnings. It possibly originated in the 1500s as a merging of chess and the Spanish game Alquerque de doze, but it may have come from the Orient, related to Parcheesi and tic-tac-toe.

Checkers is played on a chess board, which is an 8-by-8 grid of alternating black and white squares. The squares are not always black and white—the black squares may be any dark color and the white squares any light color—but the squares are referred to as black and white regardless of their actual colors. The official colors in US tournaments are green and buff. I decided to use green and light gray for the default square colors, as shown in **Figure 1**.

The two players sit on opposite sides of the board. The board is oriented so that the edge of the board closest to each player has a black square on the left and a white square on the right.

The game is played with 12 black pieces and 12 white pieces. (Again, these are the colors used to refer to the pieces. The actual colors may be different. Some commercial games of checkers use black and red pieces, but in tournament play the pieces are usually red and white.) One player controls the black pieces and the other controls the white pieces.

At the beginning of the game, the pieces are arranged on the 12 black squares closest to each player, as shown in **Figure 1**. The players alternate turns, with black moving first.

Initially, pieces can move forward (that is, towards the opponent's side of the board) and to a diagonally adjacent, unoccupied black square. If a diagonally adjacent black square is occupied by an opponent's piece, and the black square beyond that one is unoccupied, a player must move his or her piece to the unoccupied square. The opponent's piece is jumped (or captured) and removed from the board. The move must continue with additional jumps if they are available.

Here's where the rules get controversial. Many players observe the tradition of "huffing." If a jump is available and a player does not take the jump, the opponent can require that the player take back the move, or that the piece that did not take the jump be forfeited and removed from the board. However, most contemporary rulebooks take a different approach: if a jump is available, a player must take it, and must continue jumping opponent's pieces until no longer able to do so. This is the rule I will impose in the Presentation Manager CHECKERS program. There will be no huffing.

When a piece reaches the opponent's edge of the board, it is kinged or crowned. This is indicated by placing another piece of the same color on top of the piece. When a piece is first kinged, its move ends even if it can continue jumping. In subsequent moves, kings can move forward or backward along the diagonals.

A player wins after capturing all the opponent's pieces or when the opponent can no longer move any piece. Many games between two good players end in draws. Usually a draw must be decided by a referee based on the inability of either player to gain any advantage after 40 or 50 moves.

There are some variations of the game. Some versions of checkers allow kings to make long jumps, passing unoccupied black squares on a diagonal and landing on the square beyond an opponent's piece. Sometimes (particularly in Russia), the game is played on a 10-by-10 board. The PM CHECKERS program will play the standard game only.

Nonmathematical Programs" originally published in *Proceedings of the Association for Computing Machinery Conference, Toronto* (1952, pp. 46-49) and reprinted in *Computer Games I*, edited by David N.L. Levy (Springer-Verlag, 1988). Although Strachey's notation is pretty obvious once you start working with it, it's probably not something I would have stumbled upon myself.

At any time during a game, the board can be represented using three 32-bit integers: white (W), black (B), and king (K). For example, at the beginning of a game, the three integers have the following values:

```
B = 00000FFFH
W = FFF00000H
K = 00000000H
```

There's a little redundancy in this notation because each square requires 3 bits (one in each of the integers) for a total of 8 different states. In reality, each square can be only one of 5 states (empty, black, white, black king, and white king), but the notation is so convenient that we can ignore the waste.

You can apply bitwise operations to these integers to derive some other characteristics of the board. For example, the squares on which black kings currently reside can be represented in C notation as:

```
B & K
```

You can determine all the empty squares (E) using:

```
E = ~B & ~W
```

As we'll see, this type of logic is very important in the CHECKERS program, both in determining whether a user is making a legal move and in determining all possible moves a piece can make.

For example, there are 9 positions on the board where an unkinged black piece can move from a square position *n* to a square position (*n*+3) without

If you're a professional software developer, *you should read* Microsoft Systems Journal *regularly. Every issue is full of creative technical and theoretical insights so you can write better programs.*

Now is an especially good time to subscribe because you can take advantage of our discounted international rates. Fill out the card below (note that the rate for 2 years offers a greater discount), drop it in the mail, and we'll start your subscription immediately.

SPECIAL FOR THE INTERNATIONAL COMMUNITY!

Microsoft Systems Journal

Please check the appropriate box

COUNTRY	1 YEAR RATE	2 YEAR RATE
Australia*	A$ 78	A$ 130
Austria*	Sch 673	Sch 1124
Belgium	BFr 2000	BFr 3340
Denmark	DKr 369	DKr 616
Eire	I£ 44	I£ 73
Finland	FMk 218	FMk 364
France	FFr 324	FFr 541
Israel*	IS 99	IS 165
Italy*	Lir 70000	Lir 116900
Japan*	¥ 6750	¥ 11273
Netherlands	F 125	F 209
Norway	NKr 352	NKr 588
Portugal	Esc 7884	Esc 13166
Spain	Ptas 6000	Ptas 10020
Sweden	SKr 327	SKr 546
Switzerland*	SFr 79	SFr 134
United Kingdom	£ 33	£ 55
West Germany*	DM 90	DM 150
Other European*	US$ 65	US$ 109
Other International*	US$ 70	US$ 117

Orders accepted at these rates until June 30, 1990

*Subscriber must apply return postage.

Add my name to your subscriber list. I've checked the rate and term I want on the chart at left.

Name _____

Address _____

_____

Postal Code _____

Country _____

☐ My payment is enclosed

☐ I'll pay when invoiced

Guarantee: *If you are ever dissatisfied with* MSJ, *you're entitled to a full refund on the unmailed portion of your subscription.*

Note: Offer limited to new subscribers only. Regular price is $50 per year, plus $15 per year postage. Please allow 8 to 12 weeks for delivery of first issue. Offer expires July 1, 1990.

FN9AC1-4

REPONSE PAYEE
PAYS-BAS

Microsoft Systems Journal
int. Antwoordnummer
C.C.R.I. Numero 454
2130 WB Hoofddorp, Holland

jumping an opponent's piece. You can represent these positions with a variable called M:

```
M = 00E00E00E0H
```

To calculate the current unkinged black pieces that might be able to move to squares with numbers that are 3 higher than their current position, use the following formula:

```
B & M
```

To calculate the destinations of these pieces you can simply shift left by 3 bits:

```
(B & M) << 3
```

But it's only possible to make these moves if the destination square is empty:

```
((B & M) << 3) & E
```

Shifting this expression to the right by 3 bits gives us a 32-bit integer that describes the position of all the current unkinged black pieces that can move to a square 3 higher in number:

```
(((B & M) << 3) & E) >> 3
```

White kings can make these same moves:

```
(((W & K & M) << 3) & E) >> 3
```

Of course, this gets more complex when you take into account the pieces that can move to squares that are 4 higher or 5 higher in number, and when considering the possible jumps. But the concepts remain the same. It is an extremely useful type of representation that avoids otherwise lengthy logic.

Visuals and Interaction

As you can see in **Figure 1**, CHECKERS will feature a checkerboard in its client window. The window that draws the board and receives keyboard and mouse input will be a child of the client window. The supervisor communicates with this child window and two others to request information about moves and to inform the window about moves made by the opponent. The board's colors may be changed using a menu option. The checkerboard looks three-dimensional and is sized to fit within the client window while maintaining the correct aspect ratio.

To move the pieces, you can use either the mouse or the keyboard. Using the mouse you'll be able to pick up a piece from a square and move it to another square. The easiest way to do this would be to have predefined mouse pointers that look like the pieces. But the pieces will not generally be the same size as a mouse pointer, so PM's normal pointer logic can't be used. Instead, you'll move bitmaps around the window. CHECKERS will also have a keyboard interface. You can move the mouse pointer to a square using the cursor movement keys, pick up a piece by pressing the space bar, move the piece to another square, and set it down using the space bar again.

CHECKERS will prevent a user from making an illegal move and require that the user make a jump when one is available and continue jumping until no more jumps are possible. This logic may also be implemented in a Presentation Manager object window.

Playing Strategy

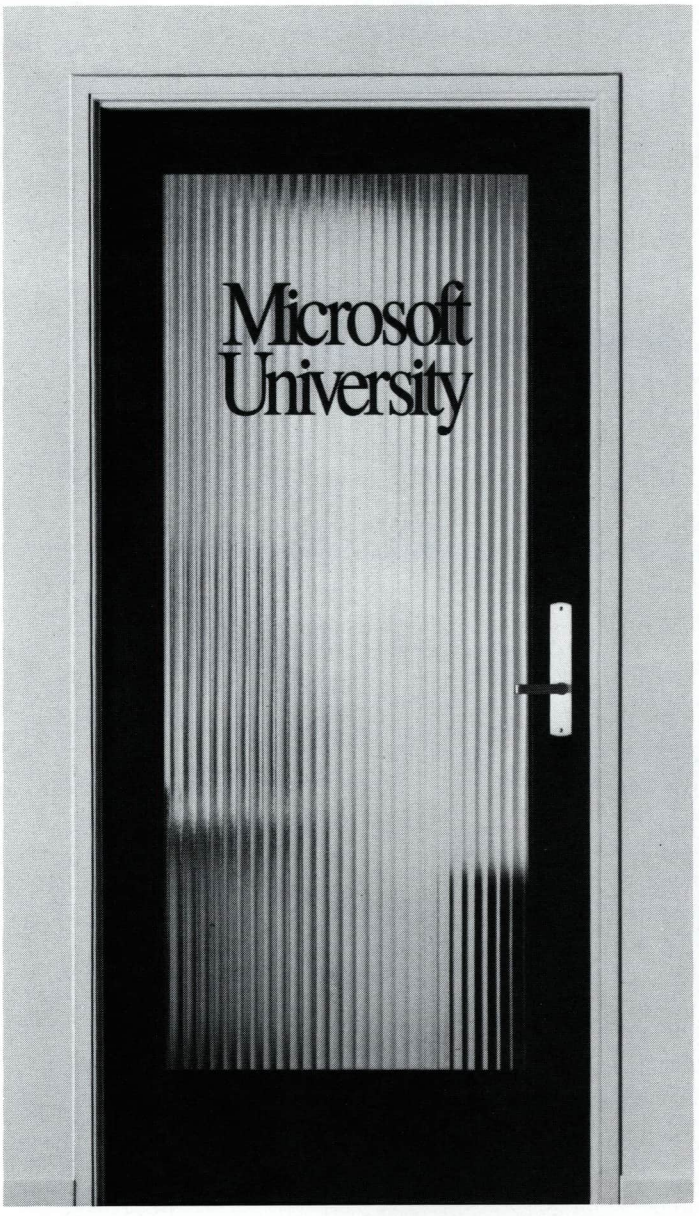
Playing a game against the program is the feature many users will find most appealing about CHECKERS. This feature requires that the program include a reasonable checkers-playing strategy. This strategy will be as simple as possible, because PM programming is difficult enough in itself.

The checkers-playing strategy will look ahead through all possible moves, counter-moves, counter-counter-moves, and so forth (to a level that I'll have to determine empirically), and determine the best move by calculating a simple score, where an unkinged piece is counted as 1 and a kinged piece is counted as a number somewhere between 1 and 2.

My standards are very low: I want the CHECKERS program to beat me (or come to a draw) most of the time, and that should not be difficult at all! I claim no skill in playing checkers. I am not interested in developing the best possible checkers-playing strategy right now. However, CHECKERS will provide a framework for others who are interested in this aspect of the program. Because the checkers-playing strategy will be implemented in a dynamic-link library, readers of *MSJ* can develop their own checkers-playing modules that CHECKERS can easily access. The interface to this dynamic-link library will be documented in a future issue. The game will also be able to play two dynamic-link libraries against each other, so we could conceivably we can stage checkers tournaments where the humans would watch while the programs played.

Now it's time for this programmer to stock up on necessary beverages, lock himself in his room, disconnect the phone, post a note on his door saying "E-mail only," and get to work. I'll emerge in time for the next issue to discuss the code that draws the checkerboard and the playing pieces. ❑

We don't have an athletic department, but our graduates get a jump on everyone in their field.



Microsoft University brings the institution of higher learning to an even higher plane. It's a place where programmers and developers learn about Microsoft systems software straight from the source: from instructors with firsthand knowledge of what drives Microsoft systems software.

Our instructors take you through intense, hands-on courses that concentrate on the development of Microsoft® OS/2 and Microsoft Windows applications.

You gain insight and expertise writing software in environments such as MS® OS/2, MS OS/2 LAN Manager and MS OS/2 Presentation Manager. As well as Microsoft Windows/286 and Windows/386.

Courses take place in a lab setting so you not only learn theory, you also gain practical experience by actually writing software. And courses are offered for different levels of expertise.

To receive more information and a copy of the Microsoft University catalog, call 206-882-8080†. And if you or your team cannot attend classes at one of our facilities, be sure to ask about our on-site customer training program or video training course that is available.

Either way, we're going to make sure you learn the nuts and bolts of Microsoft systems. So you can make great leaps in your field.

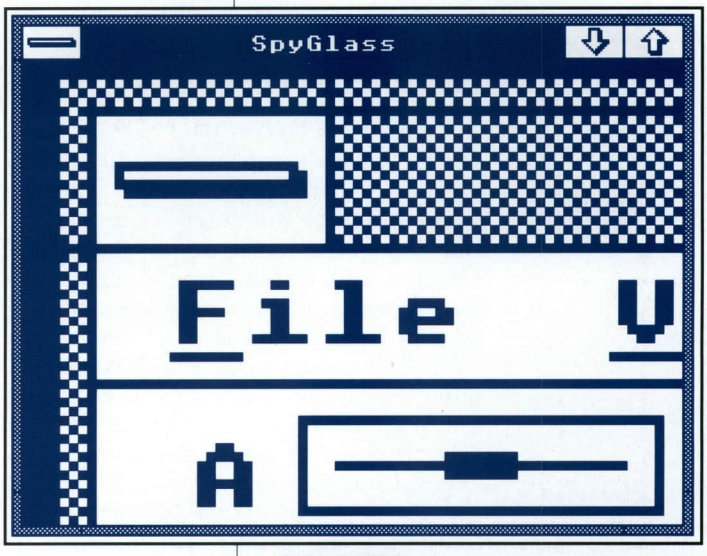
Microsoft University

SpyGlass: A Utility for Fine Tuning the Pixels in a Graphics Application

Kevin P. Welch

One of the most time-consuming aspects of programming in the Microsoft® Windows™ environment is getting things to look just right on the screen. Invariably (it seems) any user interface change creates alignment problems requiring several tweaks of one or two pixels before correct alignment is restored. Although such off-by-one errors are easy to see on a coarse, low-resolution display, they become a little more difficult when using a high-quality one with resolution in excess of 100 pixels per inch.

SpyGlass is a BLOWUP.EXE inspired utility (see "BLOWUP: A Windows Utility for Viewing and Manipulating Bitmaps," *MSJ* Vol. 2 No. 3) that enables you to enlarge selected portions of your display dynamically while maintaining a constant pixel-to-aspect ratio. In addition, it serves as a simple demonstration of several subtle graphics device interface (GDI) programming techniques that might be of use in your own applications. And although SpyGlass won't eliminate off-by-one errors, it will make them a little easier to find when tuning your application.



▲ **Figure 1** **SpyGlass enlarges a portion of the MS-DOS Executive.**

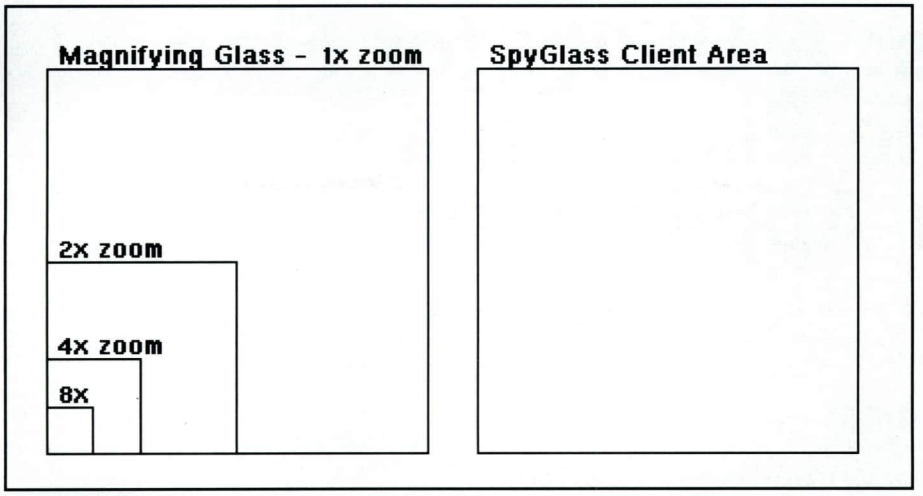
Using SpyGlass

To use SpyGlass, click inside the window client area with the left mouse button. A small rectangle (in proportion to the SpyGlass window) will appear in place of the mouse cursor. Then, while the left button is depressed and you drag the mouse around, the rectangle acts like a cursor. It becomes a magnifying glass, dynamically magnifying whatever portion of the display it covers, thereby enlarging the images in the SpyGlass window.

If you click the right button while dragging the mouse, the screen image enclosed by the magnifying glass will be enlarged inside the SpyGlass window. You can click the right button as many times as you like, taking pictures of various portions of the display (see **Figure 1**). If you hold down the right button while dragging the mouse, a continuous series of enlargements is produced inside the viewport. The enlargements may appear a little jerky on an 80286-based computer, but they look quite nice on an 80386 machine—especially one with a hardware graphics coprocessor.

Releasing the left button when you finish dragging the mouse will copy the final image from the SpyGlass window to the clipboard. If you want to

Kevin P. Welch is a computer scientist specializing in applied mathematics, robotics, and artificial intelligence. President of Eikon Systems, Inc., and a doctoral candidate in applied mathematics, he has written numerous articles on a variety of technical subjects.



▲ Figure 2 **Increasing the enlargement ratio makes the magnifying glass proportionately smaller.**

capture different sized images, you can change the magnifying glass proportions by adjusting the SpyGlass window dimensions or by selecting a different enlargement factor from the application's system pull-down menu.

Coordinate Systems

Each Windows application maintains its own coordinate system. SpyGlass is no exception. But unlike most other Windows applications, SpyGlass performs all its work in screen or display coordinates.

The origin for most application coordinate systems is defined as the upper-left corner of the client area. Although this is normally adequate when you work within your own window, it is insufficient for SpyGlass since the magnifying glass can roam all over the display. Therefore, both the magnifying glass and the viewport area are defined in terms of screen or display coordinates (DC). This enables SpyGlass to use the screen display context or DC when performing each enlargement.

As mentioned previously, the size of the magnifying glass is determined by the dimensions of the SpyGlass client area, adjusted by the currently selected magnification or enlargement factor. When the zoom factor is one, the interior portion of the magnifying glass is the same size as the SpyGlass client area. As you increase the enlargement ratio, the magnifying glass becomes proportionally smaller (see **Figure 2**).

Although the current implementation of SpyGlass supports only a small number of enlargement ratios, you could easily change the source code to include a wider range. If you experiment with other values, be aware that SpyGlass will be most efficient when using powers of two for enlargement factors. This is because the underlying StretchBlt function operates best when doubling or quadrupling each pixel size instead of performing many fractional enlargements. If you get even more adventurous, you could experiment with values less than one.

Understanding SpyGlass

To build SpyGlass, you will need the Microsoft Windows Version 2.1 Software Development Toolkit (SDK) and a Microsoft C Compiler (Version 5.0 or later). Before constructing the program with the MAKE SPYGLASS command, you will need the files listed in **Figure 3**. (*The files are available for downloading from any of the* MSJ *bulletin boards—Ed.*)

Structurally, SpyGlass is a relatively simple program, acting in most situations like a bitmap clipboard viewer. Internally, SpyGlass consists of only two functions—a main procedure that defines the SpyGlass window and retrieves or dispatches all application messages, and a function that processes all window-related messages.

Throughout the source code, note the use of the macros defined in SPYGLASS.H . The WIDTH and HEIGHT macros (see **Figure 4**), for example, compute the corresponding width and height of a given rectangle, respectively. Both use a feature of the C compiler preprocessor called token pasting. To see how token pasting works, note that each instance of the variable *x* is preceded by a double number sign (##). When the ## is interpreted by the preprocessor, it allows tokens to be used as actual arguments that can be concatenated to form other tokens. Without this replacement capability, you would not be able to access the individual structure elements of a token within a macro.

As is the case with most Windows applications, the SpyGlass window message processing function is the heart of the program. From an operational perspective, four major events are of interest; they are listed in **Figure 5** and discussed below.

SpyGlass is activated when the left mouse button is depressed inside the window client area. This causes a WM_LBUTTONDOWN mes-

Figure 3: Files Needed to Construct SpyGlass

SPYGLASS	Make file for application
SPYGLASS.DEF	Module definition file
SPYGLASS.H	Header file
SPYGLASS.RC	Application resource file
SPYGLASS.C	Source code
SPYGLASS.ICO	Icon referenced by resource file

Figure 4: WIDTH and HEIGHT Macros

```
#define   WIDTH(x)     (##x.right - ##x.left)
#define   HEIGHT(x)    (##x.bottom - ##x.top)
```

Figure 5: Response to Each Windows Message

Message	Response
WM_LBUTTONDOWN	Mouse capture activated
WM_RBUTTONDOWN	Selected portion of screen enlarged
WM_MOUSEMOVE	Magnifying glass moved
WM_LBUTTONUP	Mouse capture deactivated

sage to be sent to the application. Since it is possible to receive extraneous mouse messages, a check is made to see if the window is in an inactive state. If it is, mouse capture is enabled via a SetCapture function call. This causes the next important event: notifying the window of all subsequent mouse movements via a series of WM_MOUSEMOVE messages.

Mouse movements are captured by SpyglassWndFn. Since SpyGlassWndFn is by nature reentrant (or called before it returns), a Boolean flag is set to indicate the active window state. This allows the function to screen out unwanted messages, acting only on those of immediate interest.

Besides capturing mouse movements, SpyGlassWndFn defines both the viewport and the magnifying glass rectangles in screen coordinates. The viewport region is calculated by retrieving the window client area (in client coordinates) and converting them to screen coordinates. The magnifying glass region is also based on the size of the window client area, but it is adjusted using the selected enlargement factor and by aligning the origin with the current mouse position. The resulting rectangle is then used to replace the normal mouse cursor.

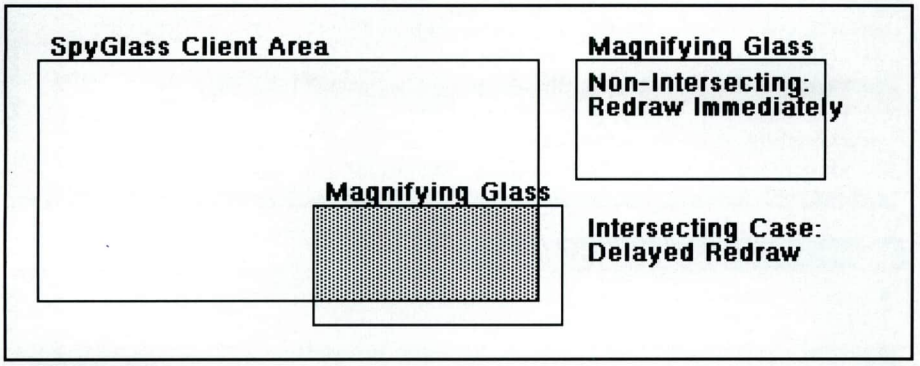
Note how the magnifying glass dimensions are enlarged by one pixel in each direction. This allows the program to work with the interior of the rectangle without erasing the inverted line around the border. The resulting visual effect is considerably smoother and eliminates unnecessary redrawing.

When the magnifying glass rectangle is tied to the mouse and WM_MOUSEMOVE messages are received, the rectangle is hidden and then redrawn in the new mouse position. Before processing the message, the function checks whether the capture flag is set. This check allows it to distinguish the message from those mouse messages received when capture is inactive.

Operationally, the mouse movement messages are handled in two different ways. If the magnifying glass intersects the SpyGlass window, it is redrawn after the viewport has been updated. This additional delay (associated with the overhead of transferring the screen image from the magnifying glass to the viewport) causes the magnifying glass to flicker and appear unresponsive. If the magnifying glass does not intersect the SpyGlass window, it is immediately repositioned, followed by the transfer of the screen image to the viewport (see **Figure 6**).

An earlier version of SpyGlass did not attempt to take advantage of those two situations. Instead, it only redrew the magnifying glass after updating the viewport. But several users commented on the unresponsive magnifying glass, prompting this algorithmic change and resulting additional complexity

THE ORIGIN FOR MOST APPLICATION COORDINATE SYSTEMS IS DEFINED AS THE UPPER-LEFT CORNER OF THE CLIENT AREA. THIS IS INSUFFICIENT FOR SPYGLASS SINCE THE MAGNIFYING GLASS CAN ROAM ALL OVER THE DISPLAY. THE MAGNIFYING GLASS AND THE VIEWPORT AREA ARE DEFINED IN TERMS OF SCREEN OR DISPLAY COORDINATES, WHICH ENABLES SPYGLASS TO USE THE SCREEN DISPLAY CONTEXT OR DC FOR EACH ENLARGEMENT.



▲ Figure 6 **If the magnifying glass intersects the SpyGlass client area, it is redrawn after a delay; if it does not intersect the window it is redrawn immediately.**

ONE CRITICAL STEP OF CLIPBOARD DATA MANAGEMENT SHOULD BE EMPHASIZED. THIS INVOLVES THE HANDLING OF THE MEMORY BITMAP CONTAINING THE VIEWPORT IMAGE. NOTE HOW THIS BITMAP IS UNSELECTED FROM THE MEMORY DISPLAY CONTEXT BEFORE BEING PLACED ON THE CLIPBOARD. BITMAPS THAT ARE TRANSFERRED TO THE CLIPBOARD WHILE SELECTED IN A DISPLAY CONTEXT ARE TYPICALLY INACCESSIBLE TO MEMBERS OF THE CLIPBOARD VIEWER CHAIN, POTENTIALLY CAUSING SOME UNUSUAL DATA MANAGEMENT PROBLEMS.

to the program. Unfortunately, this seems to be the case with many other user-interface issues: in theory they are simple, but they actually require considerably more tuning than one might expect.

One situation still ignored by SpyGlass is when the magnifying glass extends beyond the borders of the display. Currently the results produced are highly dependent on the characteristics of the active display driver. Some drivers automatically erase areas outside the screen, others leave them as they were, and still others leave them undefined. Typically, issues such as this only surface during the last few weeks of beta testing and sometimes they stay unresolved indefinitely.

Another event of interest occurs when the right mouse button is depressed. This sends a WM_RBUTTONDOWN message to SpyGlassWndFn. If the magnifying glass is active, the selected portion of the screen is enlarged in the viewport. As with the WM_MOUSEMOVE message, the magnifying glass is hidden if it intersects with the SpyGlass client area. Note how the BitBlt function is used in place of StretchBlt to transfer the screen image when the enlargement factor is one. This is because the BitBlt function is inherently faster than StretchBlt, especially when implemented by the device driver at the hardware level.

The final event of interest occurs when the left mouse button is released at the end of a drag operation. The resulting WM_LBUTTONUP message causes the mouse capture to be released, the magnifying glass to be erased, and the original cursor to be restored. In addition, the final viewport image is copied to a bitmap, which is subsequently transferred to the clipboard.

One critical step of clipboard data management should be emphasized. This involves the handling of the memory bitmap containing the viewport image. Note how this bitmap is unselected from the memory display context before being placed on the clipboard. Bitmaps that are transferred to the clipboard while they are selected in a display context are usually inaccessible to members of the clipboard viewer chain, potentially causing some unusual data management problems.

You should know that calling SetClipboardData causes the immediate transmission of a WM_DRAWCLIPBOARD message down the clipboard viewer chain. Since SpyGlass is a member of the viewer chain, the capture flag is not reset until SetClipboardData returns. This eliminates an extra update of the SpyGlass client area; an update that would be visible to the user.

Variations

Once you understand the inner workings of SpyGlass, you might want to try some experiments with the program. One interesting effect can be achieved when you set the enlargement factor to one and capture recursive images of SpyGlass itself. The resulting display is very much like point-

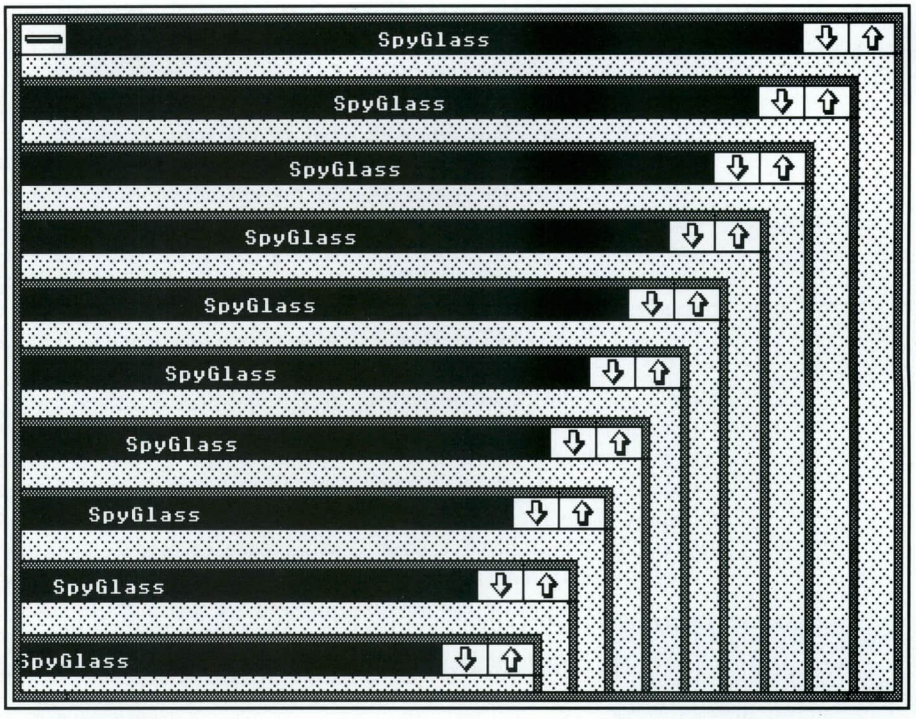
ing a high-quality video camera at its own output monitor (see **Figure 7**).

If you are a little more adventurous, you can change SpyGlass, disabling some of the subtleties discussed earlier with a few well-placed comments. If you try this, it will be clear that the additional complexity does make a difference and transforms a good implementation into a great one.

If you feel even more intrepid, you can try converting SpyGlass to run under OS/2 Presentation Manager (hereafter PM). In addition to the normal Windows to PM conversion tasks, you will have to pay attention to a few important details.

The first of these details involves the way in which PM handles menus. When you convert SpyGlass you will have to retrieve the handle to the submenu in the zero position of the system menu explicitly before you can attach and manipulate the various enlargement commands. Second, instead of creating a new display context for the DISPLAY device, you will be able to retrieve a handle to the screen presentation space directly. And when working with this presentation space, you will be able to use a single GpiBitBlt function call in place of the Windows BitBlt and StretchBlt equivalents.

Third, since PM doesn't support the clipboard viewer chain concept, you will have to remove all the code that relates to the clipboard viewer chain. OS/2 systems contain provisions for only one active clipboard viewer. Finally, when you try to place the final bitmap image on the clipboard, you will have to go through a complicated process of retrieving a memory display context (for further information, refer to the SNAP.EXE source code distributed with the Microsoft OS/2 Software Development Kit Version 1.1).



▲ **Figure 7** **The image after setting the enlargement factor to one and capturing repeated images of SpyGlass itself.**

The end result of your changes to SpyGlass will be a program that acts much the same in both Windows and PM. The same underlying design and tuning principles used with the Windows version apply equally well to PM. In addition, these enhancements can also be applied in your own applications. Although these subtle refinements are easy to perform with SpyGlass, performing such magic with a large application can be extremely time-consuming. But the results are well worth the extra effort. ❑

ONCE YOU UNDERSTAND THE INNER WORKINGS OF SPYGLASS, YOU MIGHT WANT TO TRY SOME EXPERIMENTS. ONE INTERESTING EFFECT CAN BE ACHIEVED WHEN YOU SET THE ENLARGEMENT FACTOR TO ONE AND CAPTURE RECURSIVE IMAGES OF SPYGLASS ITSELF. THE RESULT IS LIKE POINTING A HIGH-QUALITY VIDEO CAMERA AT ITS OWN OUTPUT MONITOR.

CONTINUED FROM PAGE 60
ized to NULL), the function allocates the header segment and a 65,535-byte work segment and makes them shareable with the server, which will use the work segment for calls to DosFindFirst. Later, when the server makes this call, it will ask DosFindFirst to return all files found in one call, eliminating the need to call DosFindNext.

When a filespec is added to a request via DiMakeRequest, the full path and filespec are stored in the work segment. The function also calls DosReallocSeg to enlarge the header and append a new DIRINFORESULT structure to it (pointers in this structure are set to the path and filespec). Thus, the DI functions can access these structures as a contiguous, variable-length array from an offset in the header, eliminating the complicated work with pointers that would be necessary if the structures and filespecs were intermixed. Since the actual number of structures is determined by the numRequests field of the header structure, the header can be extended indefinitely. In addition, this method allows filespecs and paths of varying length to be stored and accessed with little or no effort. As a footnote, I should mention that if a filespec's path is the same as the application's (stored in the work segment on the first call to DiMakeRequest), the path is not stored; its DIRINFORESULT pointer is instead set to the application's path.

THE SERVER ASKS DOSFINDFIRST TO RETURN ALL FILES THAT MATCH THE FILESPEC. AFTER EACH CALL, THE SERVER ADJUSTS THE WORKSPACE POINTER BEYOND THE MOST RECENT FILEFINDBUF RESULTS.

The additions to the beginning of the work segment do not have too great an impact on the remaining space. A large number of filespecs (perhaps 20) with long paths (say an average of 50 characters each) would occupy about 1000 bytes—leaving 64,535 bytes for the resulting filenames after expansion. Thus, the workspace available to the server when it first calls DosFindFirst is 65,535 minus the space occupied by the filespecs and paths.

Just before DiSendRequest places a request in the server's queue, it adjusts the server's workspace pointer to an offset beyond the paths and filespecs in the work segment. As I indicated earlier, the server asks DosFindFirst to return, in one call, all files found that match the filespec. After each call, the server adjusts the workspace pointer to point beyond the most recent FILEFINDBUF results. Thus, each call to DosFindFirst in a request uses a smaller buffer space that is found at offsets further and further into the work segment. When a server thread is finished with a request, it calculates the space occupied by the results in the work segment and stores that result in the request header for use by DiSendRequest. Then it clears the request header semaphore, frees the header and work segments, and terminates itself.

At this point, the client owns both segments; OS/2 will not discard shared segments until both processes have freed them. It would be ideal if the client could adjust the work segment down to the size occupied by the results of the request—there's no need for a 64Kb segment if only 4Kb, 16Kb, or even 20Kb are being used. Unfortunately, you cannot use DosReallocSeg to shrink a shared segment. Therefore, as soon as the server thread handling the request has freed the work segment and is about to free the header segment and terminate itself, it prepares a new work segment, which is the same size as the occupied portion of the old one. Then it copies the contents of the old segment to the new one and adjusts the appropriate pointers in the header to use the selector of the new work segment. Finally, DiSendRequest frees the work segment (which OS/2 will now discard) and returns. Note that the client can now call DiDestroyRequest in order to free the header and the new work segment.

The DI functions and server make as much memory available as possible in the calls to DosFindFirst, then shrink the memory to the required size when the results are in. The overhead of memory management is relatively small, and the DI functions conveniently handle multiple filespecs in a request, while hiding the workings of the code from the client application. Further, the time spent managing memory is made up by eliminating the calls to DosFindNext and packaging the results in a form that makes it easy for applications to employ functions like DiBuildResultsTbl to access them. You could use the DI functions in an application any time the application needs to use wildcards to search for more than one file, when time is critical, and when you want to preserve the simplicity of an application. ❑

[1] As used herein, "OS/2" refers to the OS/2 operating system jointly developed by Microsoft and IBM.
[2] As used herein, "DOS" refers to the MS-DOS and PC-DOS operating systems.
[3] For ease of reading, "Windows" refers to the Microsoft Windows graphical environment. "Windows" is intended to refer to this Microsoft product and not to such products generally.

MSJ Source Code Listings

All our source code listings can be found on Microsoft OnLine, CompuServe®, and two public access bulletin boards. On the East Coast, users can call (212) 889-6438 to join the RamNet bulletin board. On the West Coast, call (415) 284-9151 for the ComOne bulletin board. In either case, look for the MSJ directory. Communications parameters for public access bulletin boards: 2400 baud, word length 8, 1 stop bit, full duplex, no parity. ComOne is also accessible using a Hayes® 9600 baud modem.

Microsoft®

000-000-257