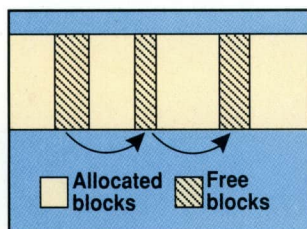


01

A Technical Study of Dynamic Data Exchange Under Presentation Manager

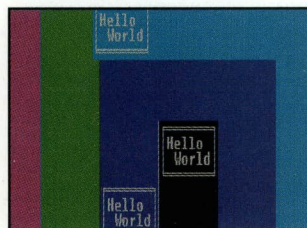
The features of the Presentation Manager DDE Application Program Interface (API) are explored in this article. In addition to an overview of the calls, messages, and structures that DDE uses, this article demonstrates DDE with an application that implements a dynamic graphical exchange between OS/2 processes.



17

Creating a Virtual Memory Manager to Handle More Data in Your Applications

Using a virtual memory manager (VMM) as a replacement for heap-based allocation gives an application more flexibility in dealing with huge amounts of data. This article presents a generalized VMM that can be used in any application that needs memory management beyond that of malloc.



25

Using the OS/2 Video I/O Subsystem to Create Appealing Visual Interfaces

The OS/2 video (VIO) subsystem offers the display services that are required by character-based applications. This article guides you through the VIO subsystem, examining screen virtualization, VIO data structures, pop-up programs, and how to create an appealing visual display in character mode.

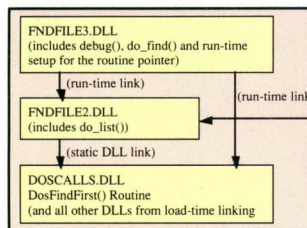
REGISTER SET			
0B	00001372	EAX	00003020
0A	000022ED	EBX	000083F3
09	00000000	ECX	00000000
08	00000000	EDX	00001A31
		EBP	000
		FS	000
		GS	000
		CR0	FFF

help F3=INT3 F10=exit F9=ap

39

Investigating the Debugging Registers of the Intel® 386™ Microprocessor

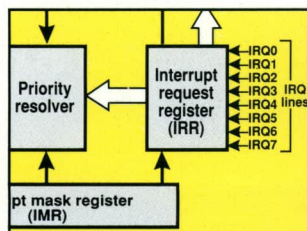
The Intel 386 microprocessor provides built-in debugging support through six special debugging registers. This article examines the debugging feature of the 386 and how debuggers work with them on your code. A debugger that supplements the setting of breakpoints by old-style debuggers is created.



51

Strategies for Building and Using OS/2 Run-Time Dynamic-Link Libraries

With dynamic-link libraries only one copy of any given function needs to be in memory where it is then shared by all sessions requiring it. Run-time linking lets you load functions as you need them and, when necessary, discard them just as easily. The five function calls necessary for run-time linking are discussed.



59

How the 8259A Programmable Interrupt Controller Manages External I/O Devices

Hardware interrupts occur in response to electrical signals received from peripheral devices such as serial ports or disk controllers and are given priority servicing by the CPU. This article focuses on maskable interrupts and how interrupt handler routines enhance program control of input/output devices.

```

1 typedef struct {
2     sl      *slptr;
3     char    sldata
4 } sl;
5
6 main()
7 {
8     sl      slinstan
9
10    /* ... */

```

69

Advanced Techniques for Using Structures and Unions in Your C Code

Structures are used to organize data in your code. When they contain references to themselves in the form of a pointer, structures can produce unlikely syntax errors. This article continues our discussion of structures as it leads you through methods of correctly using pointers to structures by analyzing sample code.

EDITOR'S NOTE

A

Articles about OS/2 systems, Presentation Manager, and Microsoft® Windows play a prominent role in *MSJ*, therefore some readers mistakenly assume we have forgotten that many of them are gainfully employed in the MS-DOS® world and will be for many years to come. We haven't.

In each issue we have tried to include articles for those readers who are continuing to work in the MS-DOS environment, and on topics of general concern such as Greg Comeau's articles on structures in C programming. The reality of DOS is that 640Kb of memory never seems to be enough. In this issue, Marc Adler presents a practical virtual memory manager that he developed for use with his ME editor. It offers *MSJ* readers a way to easily use more data in their programs by swapping memory as needed.

The Intel® 386™ microprocessor is becoming increasingly important in the marketplace and in particular with software developers. In keeping with our goal to provide the most up-to-date information, we asked two Intel engineers to take you on a guided tour of the 386 debugging registers with a view toward optimizing the efficiency of your development on 386 systems. They offer an interesting mini debugger that takes advantage of this little known 386 feature.

For OS/2 programmers, two IBM engineers discuss changes that were made to the Windows Dynamic Data Exchange (DDE) messaging protocol to make it run efficiently in the multithreaded world of Presentation Manager. They examine the technical aspects of DDE, give tips on writing an application that makes effective use of the protocol, and demonstrate DDE with a graphics exchange program.

In future issues we will have more articles for DOS developers, including a three-part series that explores the secrets of pointers in C programming, and for those for whom C is not the only programming language, a candid discussion by Ethan Winer about BASIC programming in the OS/2 environment.

—Ed.

JONATHAN D. LAZARUS
Editor and Publisher

EDITORIAL

TONY RIZZO
Technical Editor

KAREN STRAUSS
Assistant Editor

JOANNE STEINHART
Production Editor

KIM HOROWITZ
Editorial Assistant

ART

MICHAEL LONGACRE
Art Director

VALERIE MYERS
Associate Art Director

CIRCULATION

STEVEN PIPPIN
Circulation Director

L. PERRIN TOMICH
Assistant to the Publisher

JAANA NIEUWBOER
Administrative Assistant

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.

Copyright © 1989 Microsoft Corporation. All rights reserved; reproduction in part or in whole without permission is prohibited.

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.

A Technical Study of Dynamic Data Exchange Under Presentation Manager

01

Susan Franklin and Tony Peters

The IBM Corporation and Microsoft recently shipped an updated version of the OS/2 operating system, which contains some important enhancements over the initial release of the OS/2 systems. The major enhancement to the OS/2 Version 1.1 release is the inclusion of the Presentation Manager (referred to herein as PM) as a standard component. The OS/2 Presentation Manager is based on Microsoft® Windows and provides the same benefits Windows provided to DOS: a windowed, graphical user interface and support for a variety of input and output devices.

An important component of Microsoft Windows that has been implemented in OS/2 PM is the Dynamic Data Exchange (DDE) protocol. The Windows DDE version was described in "Inter-Program Communication Using Windows' Dynamic Data Exchange," *MSJ* (Vol. 3, No. 6). DDE is a published message protocol for the exchange of data between participating programs and has gained wide acceptance among Windows applications as the standard messaging protocol for data exchange. The evolution of DDE from DOS to the OS/2 environment has required some enhancements to address limitations associated with the original Windows DDE specification. This article describes that evolution and provides a graphical data exchange program as an example.

Protocol Modifications

In the Windows environment, DDE provided a consistent, flexible method for communication between applications. When moving DDE to the multitasking, protected memory environment of the OS/2 PM, however, certain changes to the protocol were needed. These changes had to address the new concepts introduced by the OS/2 environment without significantly changing the DDE model, which has proved successful in the Windows environment.

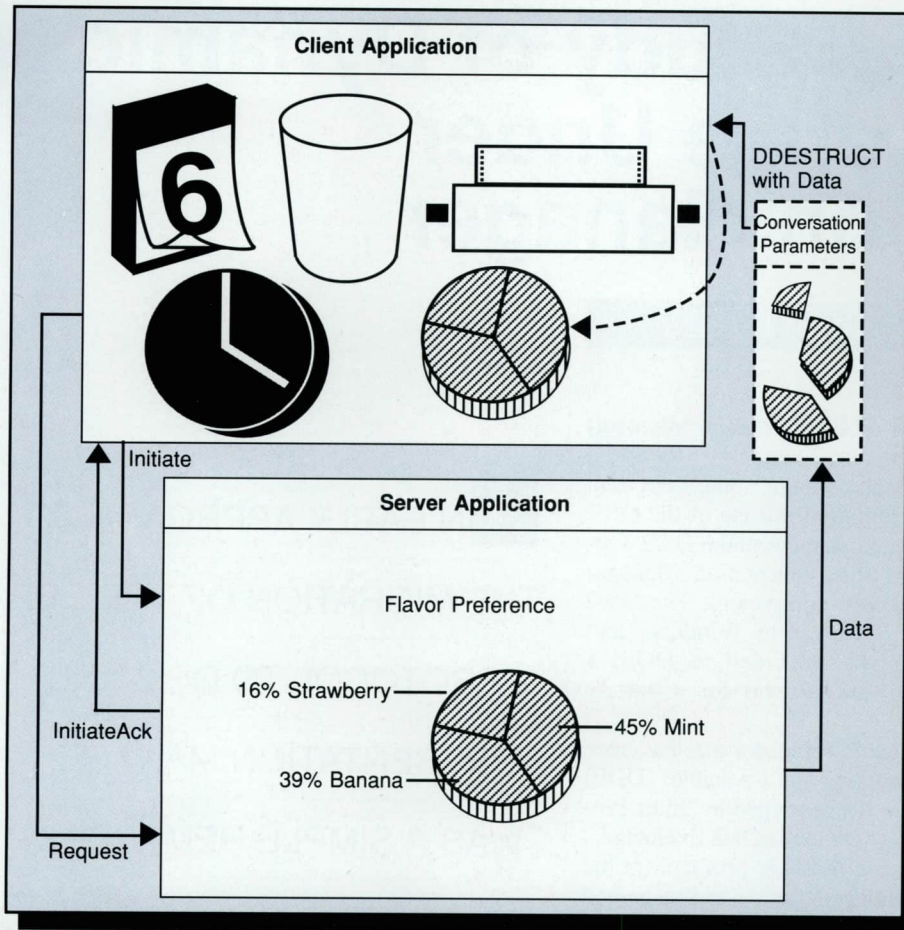
An early approach to the migration of the DDE protocol to OS/2 and Presentation Manager was a simple remapping of the message parameters. The primary change necessary was the parameter used when actually passing the data to another application, which requires crossing OS/2 process boundaries. Whereas a handle to the data is sufficient in the Windows environment, a memory selector is

Susan Franklin is a Software Engineer with the IBM Applications Systems Division in Fort Worth, Texas working with OS/2 Presentation Manager.

She has published various articles on user interface development.

Tony Peters is a Software Engineer with the IBM Applications Systems Division in Fort Worth, Texas working with OS/2 Presentation Manager and UNIX. He has published various articles on user interface development.

AN EARLY APPROACH TO THE MIGRATION OF THE DDE PROTOCOL TO OS/2 AND PRESENTATION MANAGER WAS A SIMPLE REMAPPING OF THE MESSAGE PARAMETERS. THE PRIMARY CHANGE WAS THE PARAMETER USED WHEN PASSING THE DATA TO ANOTHER APPLICATION, WHICH REQUIRES CROSSING OS/2 PROCESS BOUNDARIES.



▲ **Figure 1** In the DDE single client/single server model the client application initiates the conversation, the server acknowledges it, then the server sends data as the client requests it.

Figure 2: Initiation of a DDE Conversation

```
case WM_CREATE: /* broadcast the initiate message */
    WinDdeInitiate(hwnd, "App_Name", "Graphics_Exchange");
    break;
```

Figure 3: Responding to a DDE Conversation

```
case WM_DDE_INITIATE: /* respond if app and topic strings match */
    pDDEInit = (PDDEINIT)lParam2;
    if ((HWND)lParam1 != hwnd) {
        if (!strcmp("App_Name", pDDEInit->pszAppName)) &&
            (!strcmp("Graphics_Exchange", pDDEInit->pszTopic)) {
            WinDdeRespond(lParam1, hwnd, "Client",
                "Graphics_Exchange");
        }
    }
    DosFreeSeg(PDDEITOSEL(pDDEInit));
    break;
```

required to pass data between separate OS/2 processes. String data could still be passed in the global atom table, although

applications were necessary to explicitly request access to the atom table. This approach solved the problems introduced by protected memory into the DDE message set. But several other problems and limitations still existed. They could be solved by further modification of the protocol.

DDE suffered from the two-parameter limit inherent in PM messages. Following the Windows DDE model, the first parameter in any DDE message is the handle of the sending window. This left just one 32-bit parameter to pass all other conversation parameters and references to data. Although the necessary parameters and selectors could fit into the remaining long parameter, there was no room for any future expansion to the protocol.

Communicating with other machines on a local area network (LAN) or with other types of computers proved difficult given the parameter limits. This limitation was also a problem if and when the operating system's addressable space increased. Even in the current environment, any expansion to the DDE model or conversation parameters would not be possible given the lack of parameter space. Clearly, the protocol had to be modified such that it permitted expandability and application freedom to include additional parameters as necessary.

In order to expand the parameter space for DDE messages, the PM version of DDE uses the second DDE message parameter as a 32-bit pointer to one of two available DDE data structures. These structures contain all necessary DDE conversation parameters as well as the actual data when necessary. As the requirements for DDE change, this structure can be expanded without changing the parameters of the DDE messages. The

long address of the structure permits compatibility to future system software and other machines. By packaging all parameters into one structure, the message parameters become more consistent since the variant parameters are contained in the structure itself.

All parameters are packaged into a single structure, so the use of the atom table is no longer necessary. Adding string parameters to the atom table just introduces a second data access method that complicates the protocol. Instead string data is included in the DDE structures.

Since the majority of DDE messages will be sent or posted to windows in a separate process, the memory containing the DDE structure had to be made accessible to the receiving window. Rather than require applications using DDE to grant this access, the PM DDE protocol provides new Application Program Interfaces (APIs) for sending and posting DDE messages. Applications do not use WinPostMsg or WinSendMsg to transmit DDE messages. Instead, the message and parameters are passed to a system API, which grants the receiving window access to the DDE structure and sends or posts the message on behalf of the calling application. These APIs ensure that the access to the structure is granted consistently and correctly, while simplifying the programming efforts of an application implementing DDE.

These enhancements to DDE for OS/2 have provided a standard framework for applications to communicate without having to design a new protocol that would be suitable for DDE within a multitasking operating system. Additionally, using OS/2 DDE provides a concise method of implementing ever-increasingly complex graphical data exchange in an efficient manner.

Figure 4: Server Code for a One-Time Data Transfer

```
case WM_DDE_REQUEST: /* allocate DDESTRUCT and send data */
pDDEStruct = (PDDESTRUCT)lParam2;
strcpy(szTemp, "Text_Data");
if((pDDEStruct->usFormat == DDEFMT_TEXT) &&
(!strcmp(szTemp, DDES_PSZITEMNAME(pDDEStruct)))) {
nNumBytes = (strlen(szTemp) + 2 + strlen(szData));
pDDEStruct = st_DDE_Alloc((sizeof(DDESTRUCT) + nNumBytes),
"DDEFMT_TEXT");
pDDEStruct->cbData = strlen(szData) + 1;
pDDEStruct->offszItemName = (USHORT)sizeof(DDESTRUCT);
pDDEStruct->offabData = (USHORT)((sizeof(DDESTRUCT) +
strlen(szTemp)) + 1);
memcpy(DDES_PSZITEMNAME(pDDEStruct), szTemp, (strlen(szTemp)
+ 1));
memcpy(DDES_PABDATA(pDDEStruct), szData, (strlen(szData)
+ 1));
pDDEStruct->fsStatus |= DDE_FRESPONSE;
WinDdePostMsg((HWND)lParam1, hwnd, WM_DDE_DATA, pDDEStruct,
TRUE);
DosFreeSeg(PDDESTOSEL(lParam2));
}
else { /* send negative ACK using their DDESTRUCT */
pDDEStruct->fsStatus &= (~DDE_FACK);
WinDdePostMsg((HWND)lParam1, hwnd, WM_DDE_ACK, pDDEStruct,
TRUE);
}
break;

case WM_DDE_POKE: /* unsolicited data from client */
pDDEStruct = (PDDESTRUCT)lParam2;
strcpy(szTemp, "Text_Data");
if((pDDEStruct->usFormat == DDEFMT_TEXT) &&
(!strcmp(szTemp, DDES_PSZITEMNAME(pDDEStruct)))) {
strcpy(szData, DDES_PABDATA(pDDEStruct));
DosFreeSeg(PDDESTOSEL(lParam2));
}
else { /* send negative ACK using their DDESTRUCT */
pDDEStruct->fsStatus &= (~DDE_FACK);
WinDdePostMsg((HWND)lParam1, hwnd, WM_DDE_ACK, pDDEStruct,
TRUE);
}
break;
```

Figure 5: Server Code for a Permanent Data Link

```
case WM_DDE_ADVISE: /* set ADVISE bit in window data and ACK */
pDDEStruct = (PDDESTRUCT)lParam2;
if(pDDEStruct->usFormat == DDEFMT_TEXT) {
pWWVar->bAdvise = TRUE;
if((pDDEStruct->fsStatus & DDE_FACKREQ) == DDE_FACKREQ) {
pDDEStruct->fsStatus |= DDE_FACK;
WinDdePostMsg((HWND)lParam1, hwnd, WM_DDE_ACK, pDDEStruct,
TRUE);
}
else {
DosFreeSeg(PDDESTOSEL(lParam2));
}
}
else { /* Send a negative ACK using their DDESTRUCT */
pDDEStruct->fsStatus &= (~DDE_FACK);
WinDdePostMsg((HWND)lParam1, hwnd, WM_DDE_ACK, pDDEStruct, TRUE);
}
break;
```

Single Client/Server

In the simplest case, DDE is used when one application, called the client application, requires data from another independent application, called the server application. The classic example for this model is a

Figure 6: Server Code for Remote Execution of Commands

```

case WM_DDE_EXECUTE:          /* execute a command */
    pDDEStruct = (PDDESTRUCT)lParam2;
    strcpy(szCommand, DDES_PABDATA(pDDEStruct);
    if(!Dde_Cmd_Processor(szCommand)) { /* parse and execute
                                        the command */
        pDDEStruct->fsStatus &= (~DDE_FACK);
        WinDdePostMsg((HWND)lParam1, hwnd, WM_DDE_ACK,
                      pDDEStruct, TRUE);
    }
    else {
        DosFreeSeg(PDDESTOSEL(lParam2));
    }
}
break;

```

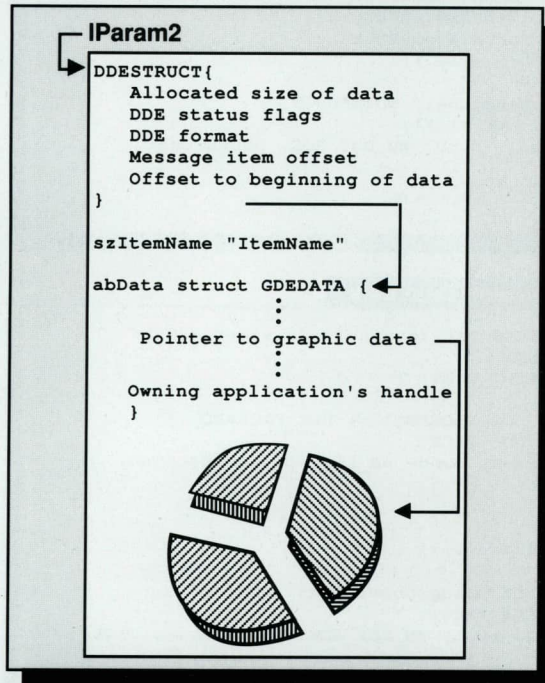
Figure 7: User DDE Data Format Registration

```

USHORT Register_DDEFMT(pszFormat)
PSZ pszFormat;
{
    HATOMTBL hAtomtbl;
    USHORT retn;

    hAtomtbl = WinQuerySystemAtomTable();
    if (retn = WinFindAtom(hAtomtbl, pszFormat))
        return retn;
    else
        return (WinAddAtom(hAtomtbl, pszFormat));
}

```



▲ Figure 8 The user format for the DDE graphics format used in the sample program is shown above. Arrows represent the offset to the data.

charting program receiving updates of data from a spreadsheet and reflecting those changes by redrawing the chart. In that case, the charting program is the client and the spreadsheet is the server.

We will begin by following the logic for the simplest of cases, the single client/single server model. In this example, the purpose of the DDE conversation is the exchange of graphical data between programs. The server application is any application wishing to display a picture in the client application's window. The graphical data is packaged and sent to the client application each time the server application wishes to change the appearance of its picture.

Figure 1 shows the general logic flow for the single client/single server type of application. The client application initiates the conversation. Once the server acknowledges the initiate, the client requests the data and the server sends it inside the appropriate structure.

A single client/single server DDE conversation begins when

the client application broadcasts a WM_DDE_INITIATE message to all other top-level windows in the system. The client specifies the string name of the application expected to reply, as well as a string name identifying the topic of the proposed conversation. Either of these string names may be NULL to indicate that the desired server application or the topic name is not specific and any application implementing DDE may participate. The WM_DDE_INITIATE message is not broadcast directly by the application. Instead, the client application uses the WinDdeInitiate call to send the message. Figure 2 illustrates the procedure for initiation.

When a server application gets the WM_DDE_INITIATE message, it checks the application and topic names to determine whether it will participate in the conversation. Sample code for initiate processing appears in Figure 3. Note that the application and string pointers that were input to the WinDdeInitiate call do not surface as explicit message parameters in the WM_DDE_INITIATE message. Instead, they are included in the DDEINIT structure, which is referenced by the second parameter. In all DDE messages, the first parameter contains the window handle of the window that originated the DDE message.

If the server decides to participate in the conversation, then the WinDdeRespond call is used in order to send the WM_DDE_INITIATEACK message back to the client application. Again, the strings referenced in the parameters of this call will be copied to the DDEINIT structure, and the pointer to the structure will be passed as the second parameter in WM_DDE_INITIATEACK.

Once the conversation link

Insider Information That Won't Land You In Jail!


**(But is guaranteed to
make you a better
programmer!)**

Get the technical insights on the new generation of software straight from the folks who wrote it. Written, *and read*, by the best in the industry, *Microsoft Systems Journal* is your entree to the inner circle of software development.

*Subscribe TODAY at this
Professional Programmer's Discount!*
**Save 40% Off the
Regular Price.**

OUR GUARANTEE: IF MSJ EVER LETS YOU DOWN, YOU'LL RECEIVE A PROMPT REFUND ON THE UNMAILED PORTION OF YOUR SUBSCRIPTION. **NO QUESTIONS ASKED.**

Microsoft SYSTEMS JOURNAL

 **YES!** I want to start writing better programs now! Begin my subscription today for 1 year (6 issues for \$29.95).

New subscription Renewal

Name _____

Organization _____

Address _____

City _____ State _____ Zip _____


Telephone _____ Business Home

Payment enclosed Bill me

Allow 6-8 weeks for delivery of first issue. Offer good in the 50 United States, the District of Columbia and Canada only. Canadian subscribers add \$10 per year. Full U.S. rate is \$50 per year. Offer ends Dec. 31, 1989.

EF059A-1

Microsoft SYSTEMS JOURNAL

 **YES!** I want to start writing better programs now! Begin my subscription today for 1 year (6 issues for \$29.95).

New subscription Renewal

Name _____

Organization _____

Address _____

City _____ State _____ Zip _____

Telephone _____ Business Home

Payment enclosed Bill me

Allow 6-8 weeks for delivery of first issue. Offer good in the 50 United States, the District of Columbia and Canada only. Canadian subscribers add \$10 per year. Full U.S. rate is \$50 per year. Offer ends Dec. 31, 1989.

EF059A-1



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

has been established, the actual data transfer may take place. This exchange may be a one-time data transfer, an ongoing data transfer, or a transfer of remote commands.

One-time data transfer is accomplished by using the WM_DDE_REQUEST message when data is flowing from server to client and the WM_DDE_POKE message when data is flowing from client to server. In either case, a DDESTRUCT is allocated and filled by the client application. The structure contains status flags describing the format of the data that is being requested (or poked) as well as the status and the size of the data. Once the structure is filled, the WM_DDE_REQUEST or WM_DDE_POKE message is posted to the server application using the WinDdePostMsg API. This is a call that is used for all messages except for WM_DDE_INITIATE and WM_DDE_INITIATEACK. The call ensures that the receiving window handle gets access to the memory allocated for the DDESTRUCT.

When the data is being poked, the WM_DDE_POKE message is the only one required to complete the transfer, since the transfer is unsolicited. When the data is requested, the server application responds with the WM_DDE_DATA message. If the server cannot supply the data in the requested format, it responds with a negative WM_DDE_ACK message. Figure 4 illustrates the code required in the server application to handle both one-time data transfer methods.

Perhaps the most common type of data transfer in DDE is the establishment of a permanent data link between applications. In that case, the client application posts a WM_DDE_ADVISE. The

Figure 9: Allocation of Shared Memory for DDE Message Communication

```
/* send a request for data */

/* allocate memory */
DDEstrptr = DDE_Alloc(sizeof(DDESTRUCT), IDS_GDE);
WinDdePostMsg((HWND)lParam1, DDEtoHWND, (ULONG)WM_DDE_REQUEST,
              DDEstrptr, TRUE);
:
PDDESTRUCT DDE_Alloc(size, format)
int size;
char *format;

/*****
 * 1. Allocate a block of size bytes
 * of giveable, shared memory for DDE call.
 * 2. Fill in DDE data format by
 * calling Register_DDEFMT((PSZ)format);.
 *****/
{
  SEL ddepsel;
  USHORT dasret;
  PDDESTRUCT DDEstrptr;
  if ((dasret = DosAllocSeg(size, &ddepsel, SEG_GIVEABLE)) == 0) {
    DDEstrptr = (PDDESTRUCT)SELTOPDDES(ddepsel);
    memset(DDEstrptr, (BYTE)NULL, size); /* set allocated memory
                                          to nulls */

    /* fill in DDE data format */
    DDEstrptr->usFormat = Register_DDEFMT((PSZ)format);

  } else { /* error */
    return((PDDESTRUCT)NULL);
  }
}
```

Figure 10: Freeing DDE Shared Memory

```
MRESULT APIENTRY MasterDDEWndProc(hwnd,message,lParam1,lParam2)
:
:
DDEstrptr = (PDDESTRUCT)lParam2;

case WM_DDE_DATA:
:
:
  DosFreeSeg(PDDESTOSEL(DDEstrptr));
```

DDESTRUCT is filled in the same manner as it is for the WM_DDE_REQUEST message. This message informs the server application that the client would like to receive WM_DDE_DATA messages as the data change. As in the case of the WM_DDE_REQUEST message, the server responds with a negative WM_DDE_ACK message if the data cannot be supplied in the requested format. If the data can be supplied, the client will continue to receive WM_DDE_DATA messages until either the conversation terminates or the permanent link is terminated. Figure 5 shows the server handling a request for a permanent data link.

A SINGLE CLIENT/SERVER DDE CONVERSATION BEGINS WHEN THE CLIENT APPLICATION BROADCASTS A WM_DDE_INITIATE MESSAGE TO ALL OTHER TOP-LEVEL WINDOWS IN THE SYSTEM.

Table 1: OS/2 DDE Messages

Each DDE message has two parameters. The first parameter, IParam1 (a long word), carries the handle of the sender's window; it is the same in all cases and is not shown in the table below. The second parameter, IParam2 (a long word), carries the far pointer to the DDESTRUCT structure for WinDDEPostMsg() or the DDEINIT structure for the WinDDEInitiate() calls.

DDE Message	Purpose	IParam2
WM_DDE_INITIATE	Requests initiation of a conversation.	DDEINIT far *
WM_DDE_INITIATEACK	Sent by a server application in response to a WM_DDE_INITIATE message for each topic the server wishes to support. The application uses WinDDEInitiate() to send this message.	DDEINIT far *
WM_DDE_TERMINATE	Posted by either application participating in a DDE conversation to terminate the conversation.	(reserved)
WM_DDE_REQUEST	Posted from a client to request data from a server application.	DDESTRUCT far*
WM_DDE_ACK	Notifies an application of the receipt and processing of a: WM_DDE_EXECUTE WM_DDE_DATA WM_DDE_ADVISE WM_DDE_UNADVISE WM_DDE_POKE WM_DDE_REQUEST (some cases)	DDESTRUCT far*
WM_DDE_DATA	Notifies a client application of the availability of data.	DDESTRUCT far*
WM_DDE_ADVISE	Requests the server application to supply and update for a data item whenever it changes.	DDESTRUCT far*
WM_DDE_UNADVISE	Request to a server application that a specified item should no longer be updated.	DDESTRUCT far*
WM_DDE_POKE	Request an application to accept an unsolicited data item.	DDESTRUCT far*
WM_DDE_EXECUTE	Sends a string to a server application to be processed as a series of commands.	DDESTRUCT far*

When the client terminates the permanent link, it posts the WM_DDE_UNADVISE message. This message does not end the DDE conversation; rather, WM_DDE_DATA messages will no longer be posted when data changes.

Another requirement during a DDE conversation is the execution of commands by a server application on behalf of the client. In this particular case,

DDESTRUCT contains a string of commands to be executed. A positive or a negative WM_DDE_ACK message, depending on the outcome of the execution, is posted to the client. A code fragment for remote execution of commands is illustrated in **Figure 6**.

Termination of a DDE conversation may be instigated by either the client or the server application. Typically, the

WM_DDE_TERMINATE message is posted when the user has requested to close the application, although this isn't always the case. Whatever the reason for the termination, posting of the WM_DDE_TERMINATE indicates that no further DDE messages will be sent. The application that is posting the WM_DDE_TERMINATE may not shut down until the other application has responded with

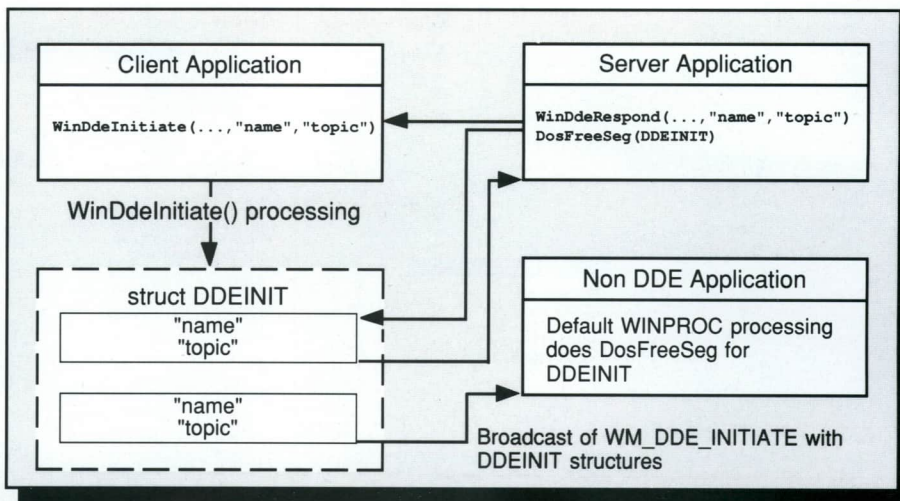
another WM_DDE_TERMINATE. There are no parameters used with the terminate message.

User-Defined Formats

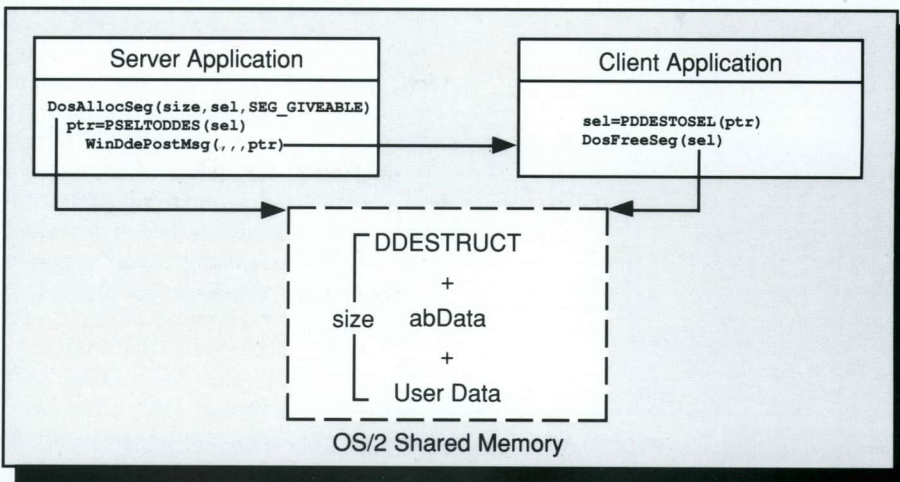
It is possible for an application to create its own DDE data format if an appropriate, system-provided data format is not available. The system provides one predefined format for interchanging text strings, DDEFMT_TEXT. Before an application uses an application-defined data format, however, it must establish a convention for obtaining a unique ID and registering that format so other applications can associate the format ID in the DDESTRUCT with their specialized data format. Although appearing similar to clipboard data formats, DDE formats are not to be confused with clipboard formats. Clipboard formats identify handle types, whereas DDE formats identify the actual layout of the data in the DDESTRUCT block. We have chosen to register DDE formats using the system atom table. The prefix of DDE formats is DDEFMT_. Using the atom manager to register DDE formats guarantees unique IDs among all applications that use this method to register DDE formats.

Figure 7 represents a function, Register_DDEFMT, which illustrates how to register a user-defined DDE data format with the system. Register_DDEFMT returns the DDE data format for either an existing or a newly created data format. The first time Register_DDEFMT is

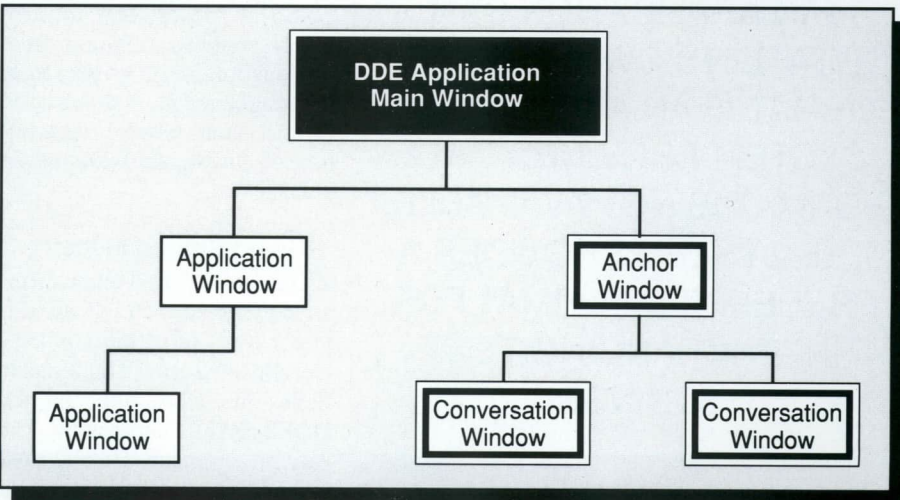
► **Figure 13**
The parent-child relationship of DDE windows in our applications is shown by this family tree. To facilitate DDE processing by our applications, all DDE windows are isolated under a single parent window.

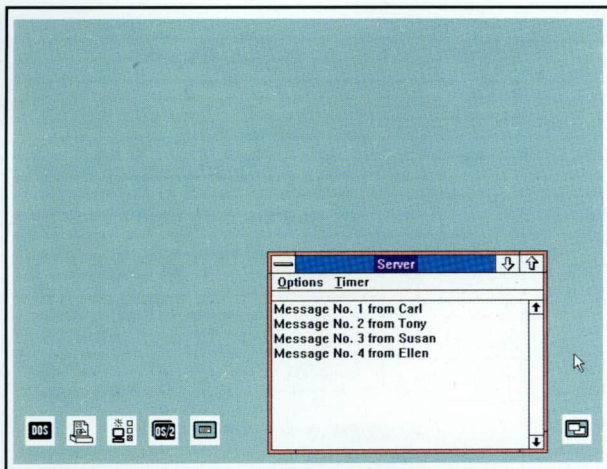


▲ **Figure 11** The system takes the strings passed in the WinDdeInitiate call and copies them into multiple DDEINIT blocks which are broadcast to all applications running on the system.

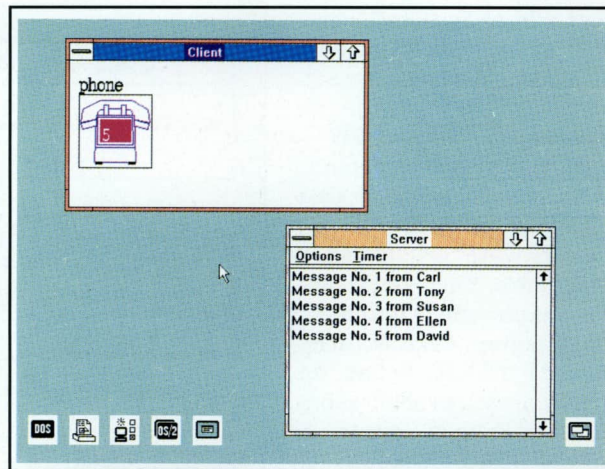


▲ **Figure 12** The DDESTRUCT is allocated by the server application, passed by the server to the client, and then released by the client application.

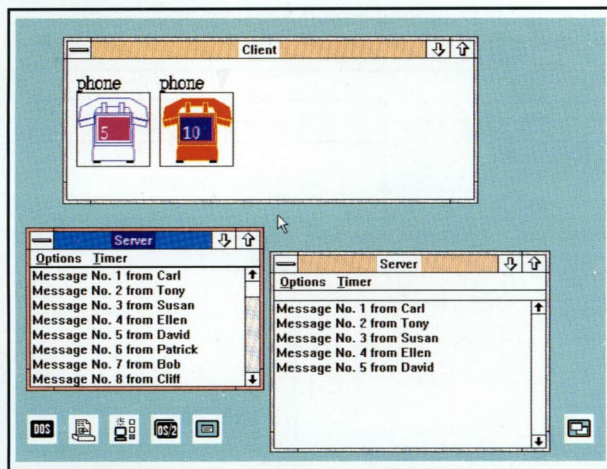




▲ **Figure 14** Invocation of one server.



▲ **Figure 15** Invocation of client and DDE initialization.



▲ **Figure 16** The second server is invoked and signals the client.

AN APPLICATION CAN CREATE ITS OWN DDE DATA FORMAT IF AN APPROPRIATE SYSTEM-PROVIDED DATA FORMAT IS NOT AVAILABLE. THE SYSTEM PROVIDES A PREDEFINED FORMAT FOR INTERCHANGING TEXT STRINGS.

invoked for a particular DDE data format, that format is registered in the system atom table. Every call that is sent to Register_DDEFMT for a particular DDE format string identifier results in the format being retrieved and returned to the caller. Looking ahead, **Figure 9** shows an example of a function call to Register_DDEFMT.

The user DDE data format for the DDE

graphics exchange sample program in this article is shown in **Figure 8**. The format consists of a graphics descriptor control block in the abData area. This graphics descriptor control block contains, among other information, an offset pointer to the graphics data that is passed in the same shared memory block following the GDE data in abData.

Using Shared Memory

DDE uses shared memory for all communications. Two different types of memory objects are allocated for DDE transactions—the DDEINIT and the DDESTRUCT structures. The way these objects are allocated may differ, but both result in

shared memory made available to the recipient. These shared memory objects need to be freed properly for OS/2 shared memory management to be effective. The data areas containing szAppName and szTopicName in the WinDdeInitiate and WinDdeRespond calls may be in private or shared memory. When the WinDdeInitiate and WinDdeRespond calls are executed, the system will copy those strings into DDEINIT and make DDEINIT available to the recipient.

The data area that contains the DDESTRUCT structure must be allocated using the SEG_GIVEABLE flag in the DosAllocSeg and DosAllocHuge calls. The WinDdePostMsg API makes the DDESTRUCT memory object available to the message recipient and frees the object from the sender.

Any pointers that are part of the abData field must point to shared memory. It is the application's responsibility to manage the allocation and sharing synchronization of these shared data areas. The application may use either base OS/2 memory management API calls or higher level memory subsetting common services to achieve this, as long as the

memory is managed properly.

The message recipient is responsible for freeing all memory objects after retrieving the information. `DosFreeSeg` is used to release the memory. `DDEINIT` and `DDESTRUCT` must both be released by the message recipient.

Figures 9 and 10 show the allocation and deallocation of the shared memory objects necessary for processing of the `WinDdePostMsg` API. The first part of Figure 9 is a call to a local function, which allocates and initializes the DDE shared memory object. The accompanying function, `DDE_Alloc()`, does the actual shared memory allocation by making a call to `DosAllocSeg()` and clears the entire shared memory object. The DDE shared memory block is initialized with the user-defined data DDE format by setting `usFormat` field in `DDESTRUCT`. This is accomplished by calling the function `Register_DDEFMT()`, which we previously discussed. By allocating and initializing the DDE shared memory objects in this manner, we can readily obtain ready-to-use shared memory objects for each of our DDE transactions.

Figure 10 shows how deallocation of the `DDESTRUCT`, in this case during the processing of a `WM_DDE_DATA` message, is accomplished by calling `DosFreeSeg` with the selector of the DDE memory object as its parameter.

DDEINIT and DDESTRUCT

Presentation Manager processes the data and allocates the memory for the `WinDdeInitiate` and `WinDdeRespond` API calls as shown in Figure 11. The system takes the strings passed in the `WinDdeInitiate` call and copies the strings into multiple `DDEINIT` blocks that are broadcast to all applications running

Figure 17: Initiation of a Graphics_Exchange DDE Conversation

```
case WM_CREATE: /* broadcast the initiate message */
    WinDdeInitiate(hwnd, "", "Graphics_Exchange");
    break;

case WM_DDE_INITIATE: /* reply with an initiate to specific server */
    if (hwnd != lParam1) {
        if ((!strcmp("Client", DDEInitPtr->pszAppName)) &&
            (!strcmp("Graphics_Exchange", DDEInitPtr->pszTopic))) {
            itoa((int)lParam1, itoa_buf, 10);
            WinDdeInitiate(hwnd, itoa_buf, "Graphics_Exchange");
        }
    }
    DosFreeSeg(PDDEITOSEL(DDEInitPtr));
    break;
```

Figure 18: Response to a Graphics_Exchange DDE Conversation

```
case WM_DDE_INITIATE: /* respond if app and topic strings match */
    pDDEInit = (PDDEINIT)lParam2;

    /* check for null strings or Graphics_Exchange */
    if ((HWND)lParam1 != hwnd) {
        itoa((int)hwnd, szTemp, 10);
        if(((!strlen(pDDEInit->pszAppName)) &&
            (!strcmp("Graphics_Exchange", pDDEInit->pszTopic))) ||
            ((!strcmp(szTemp, pDDEInit->pszAppName)) &&
            (!strcmp("Graphics_Exchange", pDDEInit->pszTopic))) ||
            ((!strlen(pDDEInit->pszTopic)) &&
            (!strlen(pDDEInit->pszAppName)))) {

            /* create conversation window and respond */
            hwndConv = WinCreateWindow(hwndDDE,
                (PSZ)"DDEConversation",
                (PSZ)NULL, WS_VISIBLE,
                0,0,0,0, hwnd, HWND_TOP,
                ++nConvID, (PVOID)NULL,
                (PVOID)NULL);

            pPWWVar->ConvCnt++;
            WinDdeRespond(lParam1, hwndConv, "Client",
                "Graphics_Exchange");
        }
    }
    DosFreeSeg(PDDEITOSEL(pDDEInit));
    break;
```

on the system. Participating DDE server applications process the `WinDdeInitiate` call by sending a `WM_DDE_INITIATEACK` message with the `WinDdeRespond` call to the client and freeing the `DDEINIT` memory object from the `WinDdeInitiate` call. Non-DDE applications don't respond to the `WM_DDE_INITIATE` message and the shared memory object is released by the system default winproc.

Figure 11 also illustrates how the `DDESTRUCT` is allocated, passed, and released by the client and server applications. Figure 12 is a diagrammatic view of `DDEINIT` processing.

Multiclient/Server Model

Earlier we discussed the

IT MAY BE POSSIBLE,
AND IN A MULTITASKING
ENVIRONMENT SUCH AS
OS/2 IT IS LIKELY,
THAT SERVER APPLICATIONS
MAY HAVE TO SUPPORT
MULTIPLE CLIENTS
AND MULTIPLE CLIENTS CAN
RECEIVE DATA
SIMULTANEOUSLY FROM
MULTIPLE SERVERS.

Table 2: Presentation Manager DDE Functions

BOOL WinDdeInitiate(hwndClient, pszAppName, pszTopicName)

Sends the WM_DDE_INITIATE message to the main window of all applications. This function creates a DDEINIT structure and passes the pointer to the structure in lParam2 of the Initiate message.

MRESULT WinDdeRespond(hwndClient, hwndServer, pszAppName, pszTopicName)

Sends the WM_DDE_INITIATEACK message back to the sender of the WM_DDE_INITIATE message. This function creates a DDEINIT structure and passes the pointer to the structure in lParam2 of the Initiate Acknowledgment message.

BOOL WinDdePostMsg(hwndTo, hwndFrom, ULONG wm, DDESTRUCT far *, BOOL fRetry)

Posts a DDE message to the hwndTo. wm contains the message, which must be within WM_DDE_FIRST and WM_DDE_LAST. If fRetry is FALSE, this call returns FALSE if the message could not successfully be posted. If fRetry is TRUE, this call retries the posting of the message in one-second intervals until the message is successfully posted.

Table 3: Presentation Manager DDE Data Structures

DDESTRUCT is used for all messages except WM_DDE_INITIATE and WM_DDE_INITIATEACK.

```
typedef struct _DDESTRUCT {
    ULONG      cbData;          /* allocated size of data */
    USHORT     fsStatus;       /* status flags */
    USHORT     usFormat;       /* DDE format */
    USHORT     offszItemName;  /* offset to item referred to
                               in this message */
    USHORT     offabData;      /* offset to beginning of data */
} DDESTRUCT;
```

The Item string name and the actual data passed are appended to the DDESTRUCT in the same memory object.

```
BYTE  szItemName[]; /* null terminated item name string*/
BYTE  abData[];    /* actual data */
```

DDEINIT is passed in the WM_DDE_INITIATE and WM_DDE_INITIATEACK messages.

```
typedef struct _DDEINIT {
    USHORT     cb;             /* size of memory object */
    PSZ        pszAppName;    /* pointer to application
                               name string in memory object */
    PSZ        pszTopic;      /* pointer to topic name
                               string in memory object */
} DDEINIT;
```

the client or server application is initiated first. In a multiclient/multi-server application relationship, multiple clients and server applications can be invoked in any conceivable order, with virtually any number of permutations. The real power of DDE to manage conversations efficiently becomes apparent in implementing multiclient/multiserver relationships.

In managing an N-way conversation, the client, the server, or both applications may be involved in a one-to-many conversation. That is, a single server may be supplying data to multiple client applications, a client application may be receiving data from several servers, or both of these things may be happening.

The DDE convention for managing one-to-many conversations is for the managing application to open a window—the conversational DDE window—for each conversation and to process the messages in a generic winproc for each conversation based on the conversational DDE window handle. There are several benefits associated with managing the conversations based on a window handle assigned to the conversation.

First, the individual conversation window handles serve as an ID for the conversation, which is guaranteed by the operating system to be unique. Second, conversations can easily be maintained and manipulated by using PM API calls (for example, WinEnumerateWindow)

single client/single server DDE conversation model. It may be possible, and in a multitasking environment such as OS/2 PM it is likely, that 1) server applications may have to support multiple clients and 2) multiple clients can receive data simultaneously from multiple servers. Likewise, there may be little or no distinction between whether

without having to develop specific data structures, such as linked lists, to keep track of conversations. Third, it is possible to easily store and retrieve conversational specific data in window words created with each conversational DDE window. We have created one additional window in each DDE application to facilitate DDE processing by our applications.

Each of our DDE applications creates a DDE anchor window, which imposes an artificial one-layer window hierarchy on the application window structure, isolating all the DDE windows under a single parent window. This lets us search through the DDE conversations directly using `WinEnumerateWindow` without having to search all application children windows for DDE conversations. The DDE conversation windows are normally traversed to locate information specific to a single conversation or during termination processing when all the links of a particular application are being terminated. **Figure 13** illustrates the parent/child relationship of DDE windows in our applications.

When either the server or the client can initiate the conversation, as can be done in a multi-client/multiserver application, a client/server relationship that is consistent with the single client/single server DDE conversation model should be maintained. An example of this situation is a newly invoked server application participating in an existing client/server conversation.

When a server is invoked during execution of a client, the server must signal the client that it is willing to participate in a conversation. For our signal, we have chosen to send the `WM_DDE_INITIATE` message with a predefined application name. The client responds to the server's initiate message

```

case WM_DDE_INITIATEACK: /* establish conversation with server */
  if( (!strcmp("Client", DDEInitPtr->pszAppName)) &&
      (!strcmp("Graphics_Exchange", DDEInitPtr->pszTopic)) ||
      (!strlen(DDEInitPtr->pszAppName) &&
       (!strcmp("Graphics_Exchange", DDEInitPtr->pszTopic))) ) {

    /* create a window for the conversation -
       child of DDEanchorHWND */

    DDEconversationHWND = WinCreateWindow(hwnd, (PSZ)"DDE_Win",
                                           (PSZ)NULL, WS_VISIBLE, 0, 0, 0, 0,
                                           WinQueryWindow(hwnd, QW_PARENT, FALSE),
                                           HWND_TOP, ++winDDEid, (PVOID)NULL,
                                           (PVOID)NULL);
    WinSetWindowULong(DDEconversationHWND, WW_CONV_HWND,
                     (ULONG)lParam1);
    WinSetWindowULong(WinQueryWindow(hwnd, QW_PARENT, FALSE),
                     WW_CONVCOUNT,
                     WinQueryWindowULong(
                       WinQueryWindow(
                         hwnd, QW_PARENT,
                         FALSE),
                       WW_CONVCOUNT)+1);

    /* send a request for initial data */
    DDEstrptr = st_DDE_Alloc(sizeof(DDESTRUCT) +
                            strlen("Graphics")+1, "DDEFMT_graphics_data");
    DDEstrptr->offsetszItemName = (USHORT)sizeof(DDESTRUCT);
    strcpy(DDES_PSZITEMNAME(DDEstrptr), "Graphics");
    WinDdePostMsg((HWND)lParam1, DDEconversationHWND,
                  (ULONG)WM_DDE_REQUEST, DDEstrptr, TRUE);

    /* send an advise to subscribe to
       receive future data updates */

    DDEstrptr = st_DDE_Alloc(sizeof(DDESTRUCT) +
                            strlen("Graphics")+1, "DDEFMT_graphics_data");
    DDEstrptr->offsetszItemName = (USHORT)sizeof(DDESTRUCT);
    strcpy(DDES_PSZITEMNAME(DDEstrptr), "Graphics");
    WinDdePostMsg((HWND)lParam1, DDEconversationHWND,
                  (ULONG)WM_DDE_ADVISE, DDEstrptr, TRUE);
  }
  /* free the memory */
  DosFreeSeg(PDDEITOSEL(DDEInitPtr));
  break;

```

▲ **Figure 19** Completion of a link establishment in a `Graphics_Exchange` DDE conversation.

with an initiate message of its own, causing the server to respond with a `WinDdeRespond` message, as would be done under the single client/single server model.

Graphics Exchange Program

DDE extensions for the multi-client/multiserver model are implemented in the sample graphics exchange program. For sample purposes, the client application exists merely to display graphical pictures representing the running server applications. Server applications may differ greatly in the func-

tion provided, but they all use DDE to transfer their graphics data to the client. In our model, the client always establishes a permanent data link with the server to receive updates as the state of the server application warrants a change in the picture's appearance. We have chosen `Graphics_Exchange` as the topic name for this example.

The server provided in the example is simulating a phone messaging service. As phone messages arrive, they are listed in the main window of the server application. Each time a message is added, the picture is updated so that it reflects the

Figure 20: WM_DDE_REQUEST Processing in the Graphics_Exchange Server

```

case WM_DDE_REQUEST: /* allocate DDESTRUCT and dump graphics data */
    pDDEStruct = (PDDESTRUCT)lParam2;
    strcpy(szTemp, "Graphics");
    if((pDDEStruct->usFormat == pWWVar->usFormat) &&
        (!strcmp(szTemp, DDES_PSZITEMNAME(pDDEStruct)))) {
        if(!pWWVar->bNoData) {
            nNumBytes = (strlen(szTemp) + 1 + sizeof(GDEDATA) +
                LOUSHORT(lPhFigCnt));
            pDDEStruct = st_DDE_Alloc((sizeof(DDESTRUCT) +
                nNumBytes), "DDEFMT_graphics_data");
            pDDEStruct->cbData = sizeof(GDEDATA) + lPhFigCnt;
            pDDEStruct->offsItemName = (USHORT)sizeof(DDESTRUCT);
            pDDEStruct->offabData = (USHORT)((sizeof(DDESTRUCT) +
                strlen(szTemp)) + 1);
            pGDEData = (PGDEDATA)DDES_PABDATA(pDDEStruct);
            st_Init_GDEData(pGDEData);
            pGDEData->cBytes = lPhFigCnt;
            strcpy(pGDEData->szItem, "phone");
            pGDEData->pGpi = (unsigned char far *)((LONG)pGDEData +
                sizeof(GDEDATA));
            GpiGetData(hpsGraphics, (LONG)IDSEG_PHONE,
                (PLONG)&lOffset, DFORM_NOCONV,
                (LONG)lPhFigCnt, (PBYTE)pGDEData->pGpi);
            memcpy(DDES_PSZITEMNAME(pDDEStruct), szTemp,
                (strlen(szTemp) + 1));
            if(lParam1 != WinQueryWindow(hwnd, QW_OWNER, FALSE)) {
                pDDEStruct->fsStatus |= DDE_FRESPONSE;
                pWWVar->hwndLink = (HWND)lParam1;
            }
            WinDdePostMsg(pWWVar->hwndLink, hwnd, WM_DDE_DATA,
                pDDEStruct, TRUE);
        }
        else {
            WinDdePostMsg(pWWVar->hwndLink, hwnd, WM_DDE_DATA,
                NULL, TRUE);
        }
        DosFreeSeg(PDDESTOSEL(lParam2));
    }
    else { /* post negative ACK using their DDESTRUCT */
        pDDEStruct->fsStatus &= (~DDE_FACK);
        WinDdePostMsg(lParam1, hwnd, WM_DDE_ACK, pDDEStruct, TRUE);
    }
    break;

```

Figure 21: Creating Multiple WM_DDE_REQUEST Messages During ADVISE

```

case WM_TIMER: /* insert phone message into listbox,
    update picture, and generate new
    REQUESTS for all ADVISING windows */

    /* listbox and picture have been updated */

    hEnum = WinBeginEnumWindows(hwndDDE);
    while((hwndEnum = WinGetNextWindow(hEnum))){
        pWWChild = (PWWVARS)WinQueryWindowULong(hwndEnum, QWL_USER);
        if(pWWChild->bAdvise){
            pDDEStruct = st_DDE_Alloc(sizeof(DDESTRUCT) +
                strlen("Graphics")+1,
                "DDEFMT_graphics_data");
            pDDEStruct->offsItemName = (USHORT)sizeof(DDESTRUCT);
            strcpy(DDES_PSZITEMNAME(pDDEStruct), "Graphics");
            WinDdePostMsg(hwndEnum, hwnd, WM_DDE_REQUEST,
                pDDEStruct, TRUE);
        }
        WinLockWindow(hwndEnum, FALSE);
    }
    WinEndEnumWindows(hEnum);
    break;

```

current number of phone messages, and the client is posted a WM_DDE_DATA message. In order to limit the complexity of

the server code presented, phone messages are generated through the use of the timer. At regular intervals, phone messages are

added or deleted from the list. This lets us focus our attention on the DDE implementation in the program rather than on the function behind the server application.

In Figure 14 the server application has been invoked and is generating phone messages. When the client application is invoked, it broadcasts the WM_DDE_INITIATE message and the server responds. Note that the application string is zero length to indicate that the client will talk to any application that will respond to the topic of Graphics_Exchange. Figure 15 shows the screen after the conversation has been linked and the first picture has been transferred to the client application.

Upon invocation of a second instance of the server application, the server must signal the client that it has begun execution. In our case, the server broadcasts the WM_DDE_INITIATE message using the same topic but specifying a target application of Client. This indicates that the initiate is a special case in which the server is signaling its invocation to any client applications that may want to subsequently initiate a conversation. Upon receiving this message, the client application broadcasts another WM_DDE_INITIATE message. If the application string name were zero length, however, the client would inadvertently establish a second link with the original server application. To prevent this occurrence, the client specifies an application name, which is simply the handle of the new server converted to a string name. The newly invoked server checks the application name and responds because the application name matches its handle. Figure 16 illustrates the message flow in establishing the second link.

This signaling convention establishes several rules for ini-

tiation of a Graphics_Exchange conversation and the subsequent response. The client application must broadcast a WM_DDE_INITIATE upon invocation with the null application string. It must also be prepared to receive a WM_DDE_INITIATE message as a signal of a newly invoked server. If the application name Client is present in this message, then the client must again broadcast a WM_DDE_INITIATE, this time with an application string containing the handle of the new participant. Figure 17 contains the initiate processing for the Graphics_Exchange client application.

The server must respond to a WM_DDE_INITIATE of topic Graphics_Exchange if and only if the application string name is zero length or if it contains the string representation of the server's window handle. Any other application string name should be ignored. If the server responds to the conversation, it creates a window to handle all future processing of the new link and responds to the client using this window handle. Note that a window word containing the conversation link count is incremented at this time. This counter will be used later by the server to determine when shutdown may occur. Figure 18 shows the response processing in the Graphics_Exchange server application.

Upon receipt of the WM_DDE_INITIATEACK message, the client application creates a window to process the link. Just as the server did, the client increments a conversation link count stored in its window word. This will be used during TERMINATE processing. The WM_DDE_REQUEST and WM_DDE_ADVISE messages are posted to the server on behalf of the newly created conversa-

Figure 22: WM_DDE_ADVISE and WM_DDE_UNADVISE Processing

```

case WM_DDE_ADVISE: /* set ADVISE bit in window data and ACK */
    pDDEStruct = (PDDESTRUCT)lParam2;
    if (pDDEStruct->usFormat == pWWVar->usFormat) {
        pWWVar->hwndLink = (HWND)lParam1;
        pWWVar->bAdvise = TRUE;
        if (pDDEStruct->fsStatus & DDE_FNODATA) {
            pWWVar->bNoData = TRUE;
        }
        if (pDDEStruct->fsStatus & DDE_FACKREQ) {
            pDDEStruct->fsStatus |= DDE_FACK;
            WinDdePostMsg(pWWVar->hwndLink, hwnd, WM_DDE_ACK,
                pDDEStruct, TRUE);
        }
        else {
            DosFreeSeg (PDDESTOSEL (lParam2));
        }
    }
    else { /* Send a negative ACK using their DDEStruct */
        pDDEStruct->fsStatus &= (~DDE_FACK);
        WinDdePostMsg (lParam1, hwnd, WM_DDE_ACK, pDDEStruct, TRUE);
    }
    break;

case WM_DDE_UNADVISE: /* turn off ADVISE bit in window data and ACK */
    pDDEStruct = (PDDESTRUCT)lParam2;
    if ((lParam1 == pWWVar->hwndLink) &&
        (pDDEStruct->usFormat == pWWVar->usFormat)) {
        pWWVar->bAdvise = FALSE;
        pWWVar->bNoData = TRUE;
        pDDEStruct->fsStatus |= DDE_FACK;
        WinDdePostMsg (lParam1, hwnd, WM_DDE_ACK, pDDEStruct, TRUE);
    }
    else {
        pDDEStruct->fsStatus &= (~DDE_FACK);
        WinDdePostMsg (lParam1, hwnd, WM_DDE_ACK, pDDEStruct, TRUE);
    }
    break;

```

Figure 23: WM_DDE_DATA Processing by Graphics_Exchange Client

```

DDEstrptr = (PDDESTRUCT)lParam2;

switch (message) {

    case WM_DDE_DATA: /* process incoming picture */
        gde_ptr = (PGDEDATA)DDES_PABDATA (DDEstrptr);
        gde_ptr->hwnd_idItem = LOUSHORT (hwnd);

        /* add if request */
        if (DDEstrptr->fsStatus && DDE_FRESPONSE) {
            WinSendMsg (WinWindowFromID (
                WinQueryWindow (hwnd, QW_OWNER, FALSE),
                ID_GRAPHICS1, IC_INSERTITEM,
                MPFROMSHORT (ICM_END), (MPARAM) gde_ptr);
        }
        else { /* replace if advise */
            WinSendMsg (WinWindowFromID (
                WinQueryWindow (hwnd, QW_OWNER, FALSE),
                ID_GRAPHICS1, IC_SETITEMSTRUCT,
                MPFROMSHORT (gde_ptr->hwnd_idItem),
                (MPARAM) gde_ptr);
        }

        if (DDEstrptr->fsStatus && DDE_FACKREQ) {
            DDEstrPtrAck = st_DDE_Alloc (sizeof (DDESTRUCT),
                "DDEFMT_graphics_data");
            WinDdePostMsg ((HWND)lParam1, hwnd, (ULONG)WM_DDE_ACK,
                DDEstrPtrAck, TRUE);
        }

        DosFreeSeg (PDDESTOSEL (DDEstrptr));

    break;

```

Table 4: Flags Contained in fsStatus Field of DDESTRUCT

Flag Name	Purpose
DDE_FACK	Set for a positive ACK
DDE_FBUSY	Set if application is busy
DDE_FNODATA	Application has no data to transfer for ADVISE
DDE_FACKREQ	Set if application wants ACKS
DDE_FRESPONSE	Set if message is a response to REQUEST
DDE_NOTPROCESSED	Message not supported by application
DDE_FRESERVED	Reserved
DDE_APPSTATUS	Application specific return

Table 5: DDE Macros for DDEINIT and DDESTRUCT Data Structures

Macro Name	Purpose
DDES_PSZITEMNAME(pddes)	Returns szItemName contained in the DDESTRUCT pddes.
DDES_PABDATA(pddes)	Returns a far pointer to the data area following the DDESTRUCT pddes.
SELTOPDDES(sel)	Converts a selector into a far pointer.
PDDESTOSEL(pddes)	Converts a far pointer to DDESTRUCT pddes to a selector. This is required when using DosFreeSeg to free the structure.
PDDEITOSEL(pddei)	Converts a far pointer to DDEINIT pddei to a selector. This is required when using DosFreeSeg to free the structure.

Figure 24: WM_CLOSE Processing and Posting of WM_DDE_TERMINATE

```

/* send WM_DDE_UNADVISE, shutdown all conversations
for this application, then quit */
case WM_CLOSE:
    if (WinQueryWindowULong(hwnd, WW_CONVCOUNT)) {
        WinSetWindowULong(hwnd, WW_CLOSE,
            WinQueryWindowULong(
                hwnd, WW_CLOSE) | WIN_CLOSING_FLAG);
        henum = WinBeginEnumWindows(DDEAnchorHWND);
        while (hwndenum = WinGetNextWindow(henum)) {
            WinSetWindowULong(hwndenum, WW_CONV_FLAGS,
                WinQueryWindowULong(hwndenum, WW_CONV_FLAGS) |
                WIN_TERM_FLAG);
            tohwnd = (HWND)WinQueryWindowULong(hwndenum, WW_CONV_HWND);
            DDEptr = st_DDE_Alloc(sizeof(DDESTRUCT) +
                strlen("Graphics")+1, "DDEFMT_graphics_data");
            DDEptr->offszItemName = (USHORT)sizeof(DDESTRUCT);
            strcpy(DDES_PSZITEMNAME(DDEptr), "Graphics");
            WinDdePostMsg(tohwnd, hwndenum, WM_DDE_UNADVISE,
                DDEptr, TRUE);
            WinDdePostMsg(tohwnd, hwndenum, WM_DDE_TERMINATE,
                NULL, TRUE);
            WinLockWindow(hwndenum, FALSE);
        }
        WinEndEnumWindows(henum);
    }
    else {
        WinPostMsg(hwnd, WM_QUIT, 0L, 0L); /* quit if no
            conversations open */
    }
    break;

```

tion window. **Figure 19** illustrates this processing.

The primary activity of the server window is the packaging of the data and posting of the WM_DDE_DATA message. In the sample program, this activity always occurs during WM_DDE_REQUEST processing. That is, even when the main server application determines that data should be transferred as a result of an ongoing ADVISE, the result is the posting of a WM_DDE_REQUEST by the main application to all of the conversation windows on behalf of the applications being advised. The format of the data message is a user-defined data format that was registered as discussed previously.

This data format is actually a data structure, GDEDATA, which is used as the input structure for a control that manipulates the graphics for the client application. Upon receiving a WM_DDE_REQUEST, the server allocates memory for the DDESTRUCT and the underlying data structure and fills in the necessary data fields required by the client and its control. The actual graphics may be deposited in either bitmap or GPI drawing orders format and included in the memory object. In the sample, they are always deposited as drawing orders. The whole data package is then transmitted via the WinDdePostMsg call. The complete processing of the WM_DDE_REQUEST message is shown in **Figure 20**.

Figure 21 shows how the main server application generates WM_DDE_REQUEST messages on behalf of all advised clients when the data changes. This is done by enumerating all child windows of the anchor window handle and posting the message to those windows that are advising a client application. The advise status of a server

application is stored in the window word of the conversation window procedure. **Figure 22** illustrates the setting of this status field upon receiving either the WM_DDE_ADVISE or the WM_DDE_UNADVISE message.

The data format for a Graphics_Exchange conversation is simply the input structure to a control, therefore the WM_DDE_DATA processing by the client is no more than an insertion or replacement of the data structure into the control. If the DDE_FRESPONSE bit is set, the client knows that the WM_DDE_DATA message is the result of the initial WM_DDE_REQUEST that was made after the conversation was linked. Since this is the first transmittal of data from the server, the picture must be inserted into the graphics control. If the DDE_FRESPONSE bit is not set, then the WM_DDE_DATA message is the result of an ongoing advise and the client must replace the picture in the control with the new data.

The control messages themselves are quite simple. The first parameter identifies the ID of the picture in question, and the second points to the structure describing and containing the picture. The internal details of the control are not relevant; the control manages the appropriate sizing and placement of the graphics data. The complete WM_DDE_DATA processing is shown in **Figure 23**.

During execution of the participating applications, the WM_DDE_DATA messages will be sent each time the timer triggers the delivery or removal of a phone message. The DDE conversation will continue until either application terminates the conversation. In our example, the conversation is only terminated when the user attempts to

Figure 25: WM_DDE_TERMINATE, and Posting of APPM_CONV_CLOSE

```

/* post terminate to server, tell client, and die */
case WM_DDE_TERMINATE:
    if (!WinQueryWindowUlong (WinQueryWindow (hwnd, QW_OWNER, FALSE),
        WW_CLOSE)) {
        WinDdePostMsg ((HWND) lParam1, hwnd, WM_DDE_TERMINATE,
            NULL, TRUE);
    }
    WinPostMsg (WinQueryWindow (hwnd, QW_OWNER, FALSE),
        APPM_CONV_CLOSE, MPFROMLONG (hwnd), (MPARAM) NULL);
    WinDestroyWindow (hwnd);

    break;

```

Figure 26: APPM_CONV_CLOSE Processing and Subsequent Shutdown

```

/* decrement conversation count and delete picture */
case APPM_CONV_CLOSE:
    WinSetWindowUlong (hwnd, WW_CONVCOUNT,
        WinQueryWindowUlong (hwnd, WW_CONVCOUNT) - 1);
    WinSendMsg (WinWindowFromID (hwnd, ID_GRAPHICS1), IC_DELETEITEM,
        (MPARAM) LOUSHORT (lParam1),
        (MPARAM) NULL);
    if (WinQueryWindowUlong (hwnd, WW_CLOSE) &&
        !WinQueryWindowUlong (hwnd, WW_CONVCOUNT)) {
        WinPostMsg (hwnd, WM_QUIT, 0L, 0L);
    }
    break;

```

close either application. The WM_CLOSE processing as well as the subsequent WM_DDE_TERMINATE processing follow the same algorithm in both the client and server applications.

When a WM_CLOSE message is received, the application enumerates all DDE conversation windows (by enumerating on the anchor window). For each conversation window, a window word is set to indicate that the user has requested a shutdown of the application. At the same time, another window word is queried to determine the handle of the conversation window with which the enumerated window is exchanging data. That window is posted a WM_DDE_TERMINATE message by the main application on behalf of the conversation window. No further shutdown processing will take place until all conversation links have terminated. **Figure 24** shows the WM_CLOSE processing by the client application. Note that the server code is very similar.

When it receives the WM_DDE_TERMINATE

THE MAIN SERVER APPLICATION GENERATES WM_DDE_REQUEST MESSAGES ON BEHALF OF ADVISED CLIENTS WHEN THE DATA CHANGES. THIS IS DONE BY ENUMERATING ALL CHILD WINDOWS OF THE ANCHOR WINDOW HANDLE AND POSTING THE MESSAGE TO THOSE WINDOWS THAT ARE ADVISING A CLIENT. THE ADVISE STATUS OF A SERVER IS STORED IN THE CONVERSATION WINDOW PROCEDURE.

Table 6: Graphics Control Messages

Message and parameters	Cause and processing
<p>Messages are sent to the control using WinSendMessage, where the message parameter is the specific message code such as IC_INSERTITEM. lParam1 and lParam2 are set as specified for the particular message.</p> <p>IC_INSERTITEM</p> <p>lParam1</p> <p>(SHORT) iPosition</p> <p>lParam2</p> <p>(PGDEDATA) pItemStruct</p> <p>Returns (SHORT) iItemActual</p>	<p>Inserts an item into the graphics control.</p> <p>This message inserts an item into a graphics control. iPosition is the 0-based position in the list where the item should be inserted.</p> <p>If iPosition is ICM_END, the item is added to the end of the control.</p> <p>pItemStruct is a pointer to the GDEDATA structure. iItemActual is the actual position where the item was inserted.</p> <p>If the control cannot allocate space to insert the item in the list, it will return ICM_MEMERROR.</p> <p>If the index selected is invalid, the control will return ICM_INVINDEX.</p>
<p>IC_DELETEITEM</p> <p>lParam1</p> <p>(SHORT) idItem</p> <p>lParam2 NULL (Reserved value).</p> <p>Returns (SHORT) cItemsLeft</p>	<p>Deletes an item from the graphics control.</p> <p>This message deletes an item from the graphics control. idItem is the unique id of the item to be deleted. cItemsLeft is the number of items remaining in the list after the item is deleted.</p>
<p>IC_SETITEMSTRUCT</p> <p>lParam1</p> <p>lParam2 (PGDEDATA) pItemStruct.</p> <p>Returns (BOOL) bSuccess.</p>	<p>Sets the pointer of the structure whose ID is specified by idItem to the structure defined by pItemStruct.</p> <p>Returns TRUE if successful; otherwise returns FALSE. (USHORT) idItem</p>

message, a conversation window checks its window word to determine if the close was initiated by its own application or by its conversation partner. If the close was initiated by the conversation partner, the window posts a corresponding WM_DDE_TERMINATE to the sender. If the close was not initiated by the conversation partner, the message is simply an acknowledgment by the partner that terminate processing

may continue. In either case, the conversation window may now destroy itself, since its conversation link is ending. At this time, it posts an application-defined message, called APPM_CONV_CLOSE, to indicate to the main window that the link has successfully terminated. Figure 25 shows the WM_DDE_TERMINATE processing by the client. Again, the algorithm is the same in the server application.

The APPM_CONV_CLOSE message serves as the signal that final shutdown may occur. As each APPM_CONV_CLOSE message is received, the link counter is decremented. When the link counter reaches zero, the main application checks its window word to determine whether a close was requested for the application. If close was requested and all links have successfully terminated, the application may complete its shutdown and terminate (see Figure 26).

Graphics exchange provides an extremely visual working example of the use of DDE to establish permanent links between separate PM applications. Support for multiple conversation management is added simply by defining one additional message and making extensive use of the window word and window enumeration facilities. Of course, these applications may be expanded to exchange other data in addition to the graphics representations exchanged in this program.

By using our example as a guide, you should be able to generalize the implementation presented in order to develop your own multitasking interprogram data exchange. We have found DDE to be a powerful and flexible element of Presentation Manager, providing an added dimension to developing interacting applications in the OS/2 multitasking environment. As the programming environment has evolved from Windows to the extensive capabilities of the OS/2 Presentation Manager, the changes made to the DDE protocol have kept pace with this new function. Moreover, it has provided a framework for further expansion as the environment continues to evolve. □

Creating a Virtual Memory Manager to Handle More Data in Your Applications

Marc Adler

The amount of memory that is available to a program under the Microsoft® OS/2 operating system is beginning to spoil many programmers. For example, when Magma's ME Programmer's Text Editor (not to be confused with the Microsoft Editor) was ported to OS/2, one of the advantages was the ability to easily edit files larger than the available memory. Going back to the DOS version of the editor, with its limited file size, became very difficult. To satisfy the desire to edit very large files under DOS, the DOS version of ME had to be enhanced. The logical way to do that was to design and build a virtual memory manager (VMM) that could handle the demand. Figure 1 lists the APIs for the VMM.

The motivation for writing a virtual memory manager was enhanced by a desire to overcome the shortcomings of malloc, the staple function of the C runtime library. Different compiler manufacturers have implemented the C memory allocation functions in different ways, each implementation being equally mysterious to the average programmer.

From a programming perspective, it is advantageous to take the mysteries out of malloc and to put the inner workings of a memory management function at programmers' fingertips, to be tinkered with in special situations and to be traced meaningfully with the Microsoft CodeView® debugger. Being able to do such tracing and tinkering might be very useful, for example, if one felt that a program had corrupted a chain of allocated blocks of memory.

Finally, there was an overriding desire to ensure that ME would never again be limited by the amount of free memory left in a given system. Unbelievably, there are still people who are using 256Kb machines. The amount of memory that DOS itself takes up combined with both the memory used by terminate-and-stay-resident (TSR) programs and the huge size of the EXE files that make up today's major applications, severely limits the amount of space that can be allocated, even with 640Kb of memory.

One specific goal in designing the virtual memory manager for the ME text editor was to keep it simple enough and general enough so that it could be used in

Figure 1: The Virtual Memory Manager API

VMMInit

Initializes the VMM. Must be called at the beginning of the application before any memory is requested.

VMTerminate

Shuts down the VMM. Call it at the end of your application.

char far *MemDeref(HANDLE h)

Dereferences the object pointed to by handle h and returns the memory address of that object.

HANDLE MyAlloc(unsigned size)

Allocates size bytes from the VMM and returns a handle to that block. The block is also filled with zeros.

MyFree(HANDLE h)

Frees the memory block pointed to by handle h.

SetVMPageSize(int kbytes)

Sets the default page size to kbytes kilobytes. For instance, SetVMPageSize(16) sets the default page size to 16Kb.

MakePageDirty(HANDLE h)

Sets the dirty bit of the page that contains the memory block referenced by handle h.

Marc Adler is the head of Magma Systems, a company that specializes in programmer's tools, OS/2 and Windows consulting, and workstation design. He is currently consulting for several Wall Street firms.

Figure 2: VMM Handles

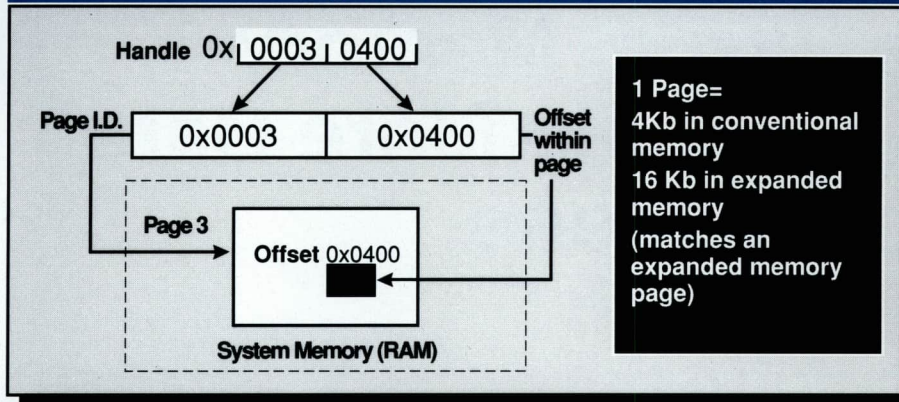


Figure 3: The Structure Serving as the Page Header

```
typedef struct page
{
    struct page *next;           /* chain to next free page in list */
    char far *memaddr;          /* memory address of the page block */
    unsigned long diskaddr;     /* disk address of the page block */
    PAGEID id;                  /* page identifier */
    unsigned long LRUcount;     /* least-recently-used count */
    unsigned pagesize;          /* how many bytes is this page */
    unsigned freebyte;          /* index of 1st free byte in page */
    unsigned bytesfree;         /* # of bytes free in this page */
    unsigned maxcontigfree;     /* max # of contiguous free bytes */

    unsigned flags;
#define PAGE_IN_MEM      0x0001
#define PAGE_ON_DISK    0x0002
#define IS_DIRTY        0x0004
#define SET_PAGE_DIRTY(p) ((p)->flags |= IS_DIRTY)
#define NON_SWAPPABLE   0x0008
#define PAGE_IN_EMM     0x0010
} PAGE;
```

FROM A PROGRAMMING PERSPECTIVE, IT IS ADVANTAGEOUS TO TAKE THE MYSTERIES OUT OF MALLOC AND TO PUT THE INNER WORKINGS OF A MEMORY MANAGEMENT FUNCTION AT PROGRAMMERS' FINGERTIPS, TO BE TINKERED WITH IN SPECIAL SITUATIONS AND TO BE TRACED MEANINGFULLY WITH CODEVIEW.

any other application that needed memory management beyond what malloc offers. Since many people have installed additional memory boards in their systems, it was also desirable to be able to exploit the capabilities of expanded memory in the virtual memory manager. (Even though ME's virtual memory manager supports EMS, its implementation won't be covered here. Such an implementation, however, would be useful as an exercise for the reader.)

The VMM must be compiled in large model. The reason is that we must remain consistent with the far pointer that we use and the pointers that some of the C library routines need (such as memset). A little work needs to be done if you would like to use the VMM under a different

memory model. (The actual VMM source code listings, VM.H and VM.C, are available on MSJ's bulletin boards and are not included here due to space constraints—Ed.)

Interspersed throughout the text will be comments about possible extensions you could implement to make the VMM more powerful. If you decide to implement any of the suggestions, I'd be interested in receiving changes, along with your comments about how they affected the performance of the VMM. They should be forwarded to the Technical Editor of *Microsoft Systems Journal*. (Interesting additions and comments may be published by MSJ in a future issue.—Ed.)

Initializing the VMM

The first thing every program that uses the virtual memory manager must do is to initialize it. This is done simply by calling the function VMInit. The main job of VMInit is to create the swap file that is used when blocks of memory must be paged out to disk. VMInit will first check to see if the user defined an environment variable called METEMP. METEMP should contain the path name of where the swap file should go. If the METEMP variable is not defined, the swap file will be created in the current directory. The use of a user-definable destination for the swap file allows the user to take advantage of any RAMdisks that might be available. Swapping to a RAMdisk, of course, is significantly faster than swapping to a hard disk.

You can set the METEMP variable with a line in your AUTOEXEC.BAT file that looks like this :

```
set METEMP=<swap path>
```

To create the name of the swap file, you use the mktemp function, which is part of the C run-

time library. This function takes a single parameter, a string representing a file name template, and creates a unique file name from that template. In this case, the template that is used is VMXXXXXX. The mktemp function will replace the uppercase Xs with characters that would make the file name unique in the current directory. For instance, mktemp might return the string VM065291 to VMInit, and we would use that as the name of our swap file.

At this point, I must confess that the virtual memory manager has one major limitation; the swap space is bounded by the available space on the swap disk (plus the amount of expanded memory that is free). A possible extension would be to allow the VMM to use multiple volumes when swapping, including swapping over a network to a totally different computer (such as a remote file server). If you do this, beware of the DOS limitation on the number of files that an application can have open simultaneously.

Terminating the VMM

Before terminating, an application must call VMTerminate to do some cleanup work. VMTerminate simply closes the swap file and deletes it. If you added routines to do performance analysis or to swap to multiple volumes, you might need to do some cleanup work at this point.

Obtaining Memory

The function that obtains memory from the VMM and returns it to the caller is MyAlloc. It replaces the standard call to malloc. Since the memory returned is zeroed out, MyAlloc can replace calloc as well. A single argument is passed to MyAlloc—the number of bytes needed—and a handle is returned.

The word handle is becoming

an increasingly popular term, mainly because of its frequent usage in Microsoft Windows. A handle is simply an identifier that is associated with a block of memory; it is used by the internal routines to identify an object. In the true spirit of data hiding, the value of a handle (also referred to as a magic cookie) will typically have no meaning to the application that uses the VMM.

In the case of our VMM, a handle is an unsigned long quantity comprised of two parts. The upper 16 bits is the page number in which the block of memory is found, and the lower 16 bits constitutes the zero-based offset from the beginning of that page. For example, a handle whose value is 00030400H signifies that the memory block is located 400H bytes away from the beginning of the block allocated for page 3. **Figure 2** illustrates this example.

This design allows us to have at most 64Kb pages, each page containing as many as 64Kb of memory. Thus our VMM can access a total of more than 4Gb, well over the maximum size of the largest hard disk.

The Structure of a Page

As I stated earlier, MyAlloc expects a single argument that represents the amount of memory we want to allocate from the VMM. If we cannot find a page with enough free memory, a new page will be allocated. At this point, let's see what is in the PAGE data structure.

You'll find the declaration of the PAGE data structure (shown in **Figure 3**) in the listing of VM.H. Since a block can reside anywhere in RAM, we need a field to hold its far address. A block can reside on disk, so we also need a field to hold the offset from the beginning of the swap file where the block is found. These two fields are called memaddr and diskaddr.

In addition to these two fields, we have fields that contain the ID of the page (a unique integer), the size of the page (in case we modify the VMM to deal with different sized pages), a bit mask to represent the status of the page (if it's in memory, on disk, or both, and whether the page is dirty), the offset to the first free byte in the page (used for implementing a linked list of free blocks within the page), and the clock for the least recently used (LRU) swapping algorithm. We also have fields for the number of free bytes within the page and the size of the largest free block of contiguous memory. These two fields can be used in conjunction, in the event we modify the VMM to do true compaction of free blocks.

The amount of space allocated for a page should be a power of 2. With careful experimentation, you might optimize the performance for your application. A routine known as SetVMPageSize is provided that allows the application to set the page size from within your application. It should be called directly after VMInit. If you decide to alter the default size of a page, then you must call SetVMPageSize only once within your application, and the call should be made before any pages are actually allocated. The reason for this is that the swapping algorithm thinks that each page is the same size. We use a default size of 4Kb for each page; however, in the version that uses expanded memory, the page size is increased to 16Kb to match the size of an expanded memory page.

Within each page, a linked list of the free blocks within that page is maintained. This list is defined by the FREEINFO structure. Each free block (and allocated block) has a header that records the number of bytes in the block and the offset to the

next free block in the page. A block that has the value of FFFFH in its offset field is the last block in the chain. This linked list is a simple one with no implicit ordering of blocks. We will talk about enhancing this list when we discuss the freeing of blocks. **Figure 4** shows an example of a typical chain of free blocks.

When MyAlloc is called, we search the page list for the first page with the necessary amount of contiguous free bytes. The function that handles this search is FindNContigBytesFree. Precedence is given to pages that are already in memory, but if there are no pages in memory that contain the needed bytes, we look for the first disk-based page that has the free space. If there are no pages either on disk or in memory that have a sufficient number of contiguous bytes free, we allocate a new page and return its address. The AllocPage routine is responsible for allocating space for a new page header and for the associated buffer.

Back in MyAlloc, we have a pointer to a page with the necessary number of bytes free, and we are assured that the page is in conventional memory. We then traverse the list of free blocks within that page and stop when we find the first block with the necessary free bytes. The block is zeroed out and the handle is returned to the calling routine.

Swapping Pages

In this version of the VMM, a call is made to the Microsoft C library routine, `_dos_allocmem`, to allocate memory for a page. This routine is really a front end for the main DOS memory allocation service (Int 21H, function 48H). Using the DOS memory functions lets us totally bypass the malloc family found in the C run-time library. (Actually, each page header is allocated using

malloc, but we can choose to use DOS memory for this if we want.) We continue to use `_dos_allocmem` to grab space for a page until we run out of memory. At this point, we have a page header allocated for the new page but no block of memory allocated for its data. What we need to do is borrow the memory used by a previously allocated page. But before that can be done, we must save that page's data somewhere. Then the new page can use that page's block of memory to store its own data in. This process is called swapping or paging.

How do we know just where in the swap file the old page's contents should be stored? The array `VMFile.slottable` contains a map of which sectors of the swap file are used by which pages. A NULL entry for a sector means that the sector is free. When we swap a page to disk for the first time, we search the slot table for the first NULL entry and then write the page to the corresponding sector.

The remaining question is, How do we determine which page to swap out to disk? If we have a bad algorithm for choosing the swappable page, we can run into a hideous phenomenon known as thrashing. If a VMM thrashes, it is spending an inordinate amount of time swapping pages between disk and memory. That can happen if we choose to swap out a frequently referenced page.

For our swapping algorithm, we use the old, time-tested least recently used algorithm. In order to implement the algorithm, we must keep a clock which is incremented every time a page is accessed. Each page has a variable which records the time when it was last accessed. To find the LRU page, we just scan the page list for the page with the minimum clock time and return a pointer to that page.

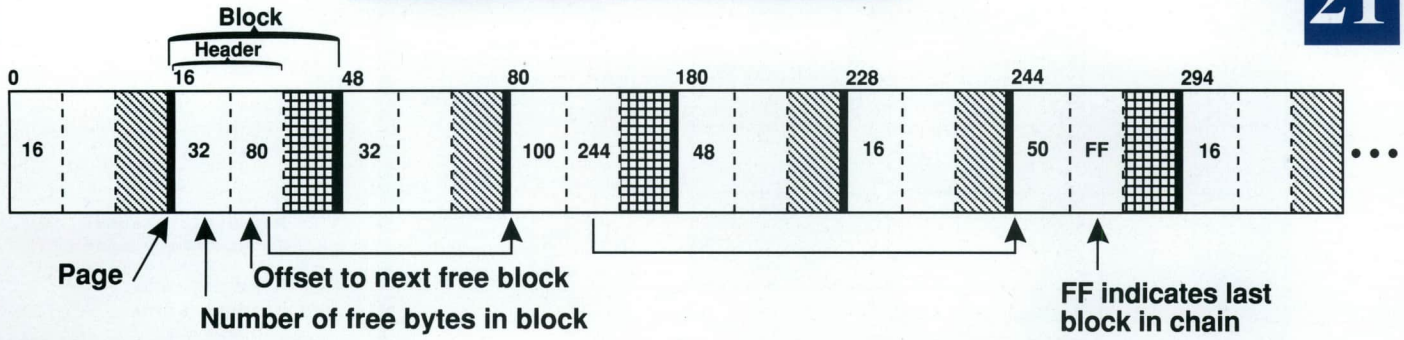
A possible enhancement to the VMM is to implement a means of making certain important pages nonswappable. For instance, instead of using malloc to allocate the headers for each page as we do now, we could allocate memory from the VMM for them. However, it would be disastrous if the page headers were swapped out to disk! In this case, we might use a nonswappable page to hold the headers and other important system information. The LRU algorithm would require a simple modification that would tell it not to consider pages marked as nonswappable.

Another possible enhancement is to keep the list of pages in a doubly linked list rather than a singly linked one and to maintain a pointer to the tail of the list. Since we move a page to the head of the page list whenever we access that page, we will be guaranteed that the least recently used page will be at the tail of the list. Using this method, we find the LRU page merely by looking at the tail; we don't need to implement a clock for the LRU algorithm, and we don't have to allocate a counter for each page in the system.

A Perfect Fit?

The method of allocation followed here is called the first fit approach. It's given that name because we stop our search at the first block that fits the criterion, that is, the first block that has enough contiguous bytes free. Another popular approach is called best fit; we traverse the entire free list in search of a block with the smallest size that satisfies our criterion. A third method is called next fit; we remember the position the last block that was allocated came from, and the next time we search for a free block, we start at that spot. The first fit method has the advantage of taking less

Figure 4: Typical Snapshot of a Memory Chain



time to find the proper memory block, and best fit has the advantage of reducing fragmentation (if you use one of the methods of reducing fragmentation discussed below).

Fragmentation of memory is a major concern when designing a memory manager. It is caused when you allocate part of a memory block that has more free bytes than you really need: part of that block will be allocated and the remainder will be put back onto the free list; however, if the remaining block is too small for any subsequent allocation request, it may never be allocated. For example, let's say that I need a block of 24 free bytes, and after searching the chain of free blocks, I come to a block that has 32 bytes free. I will allocate 24 bytes out of that block and leave a block of 8 bytes on the free list. This free block is probably too small to satisfy any future allocation request, so it will remain forever on the free list. That may not seem so bad, except that the memory manager will still have to examine the block whenever it searches the free list; multiply this example by the thousands of memory requests that a typical application might make and you'll see why fragmentation is a severe problem. **Figure 5** illustrates a graphic representation of fragmentation.

A simple way of solving fragmentation problems is to round

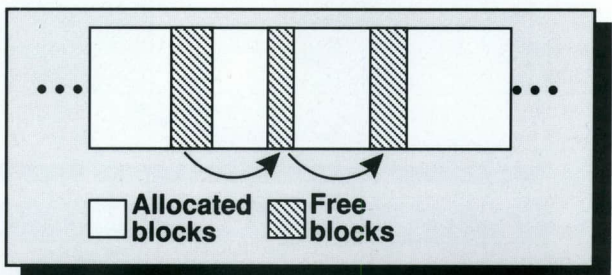
off an allocation request to a higher number; the excess bytes will be included in the free block that we return to the application. For the example above, I might use the simple heuristic of rounding off all allocations to the nearest power of 2. If I ask for 24 bytes, and I come upon a block of 32 bytes, then I will return the entire block to the application. Although 8 bytes may never be used, I will not have a small block floating about in the free list.

Another solution is to perform periodic garbage collection and periodic compaction on the memory blocks, an approach we will also consider.

Dereferencing Handles

You will recall that the handle to a memory block is merely an identifier that tells the VMM how to reference that block; the application does not really know what memory address the handle points to. Once an application obtains a handle to a memory object, it has to go through a dereferencing step in order to use the memory block that the handle refers to.

Remember, too, that a handle is an unsigned long value that comprises the page identifier and the offset within that page. The function MemDeref must be called whenever you need to transform a handle into a memory address. For example, if we wanted to copy the string



▲ **Figure 5** Fragmentation occurs when free blocks are too small to be allocated.

PERIODIC COMPACTION OF MEMORY BLOCKS WILL REDUCE FRAGMENTATION IN OUR VMM. BEFORE COMPACTION, THE FREE BLOCKS ARE INTERMINGLED WITH THE ALLOCATED BLOCKS, AND THE FREE LIST MUST BE TRAVERSED IN ORDER TO LOCATE A BLOCK WITH THE DESIRED AMOUNT OF CONTIGUOUS BYTES FREE.

Figure 6: Coalescing

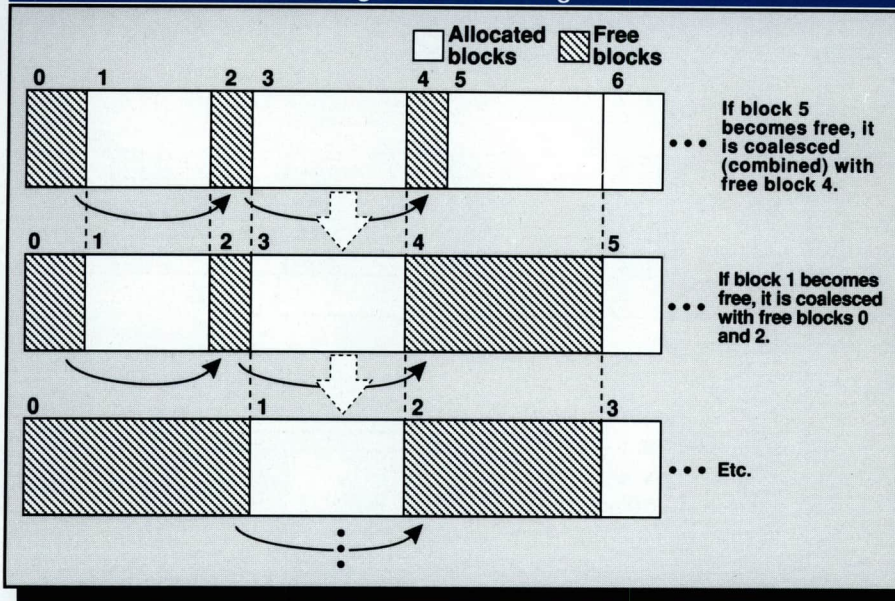


Figure 7: Double Indirection

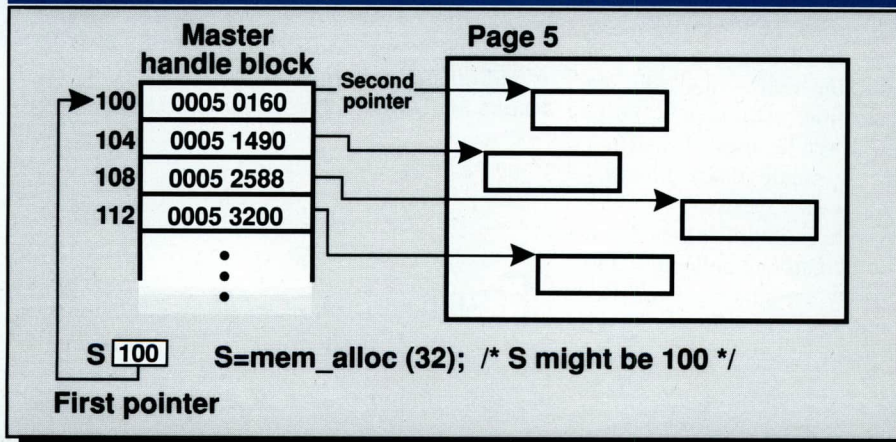
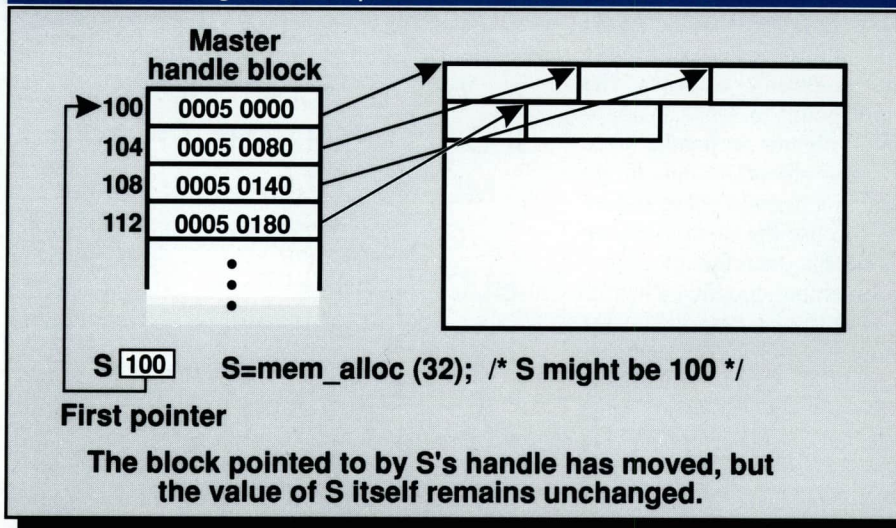


Figure 8: Compaction with Double Indirection



XYZ into an allocated memory block, we must write the following code:

```
#include "vm.h"
...
HANDLE h;
char far *s;
...
/* Note—this assumes
large memory model */
if ((h = MyAlloc(4)) ==
(HANDLE) NULL)
/* perform error
processing */
...
if ((s = MemDeref(h)) !=
NULL)
strcpy(s, "XYZ");
```

MemDeref simply breaks the handle into its constituent parts, searches for the page with the ID contained in the handle, swaps the page into memory if necessary, adds the base memory address of the page with the offset of the block that the handle specifies, and returns a pointer to the memory address. One nice thing that MemDeref also does is to move the referenced page to the front of the linked list of pages that are currently in use. The VMM assumes a locality of reference within an application—that is, once a page is referenced, it will continue to be referenced in the application's surrounding code. Moving the newly referenced page to the front of the page list reduces the time needed to traverse the page list for subsequent searches for that page.

Touching a Page

If you look carefully at the code for the WritePage function, you'll notice that a page will get written to disk only if it's currently in memory and if the page has been modified since it was allocated or last written to disk. If the same image of a page exists both in memory and in the swap file, there is no need to perform the actual write to disk.

The VMM includes a routine

that sets the dirty bit of the page that contains a referenced memory block. The `MakePageDirty` function takes a single argument, the handle of a memory block, and searches for the page corresponding to that block. When that page is located, its dirty bit is set. This scheme ensures that when we modify a certain portion of memory, the associated page will be swapped properly when memory begins to run low.

Referring to the example above, if we wanted to change the bytes pointed to by handle `h` to the string `ABC`, we would need to do the following :

```
s = MemDeref(h);
strcpy(s, "ABC");
MakePageDirty(h);
```

Freeing a Memory Block

When a memory block is no longer needed, we call the `MyFree` routine to release it back into the memory pool. `MyFree` takes a single argument—the handle to the memory block that we will release. To release a memory block, we simply place it on the page's linked list of free blocks and increase the number of free bytes within that page. What makes the operation a little more complicated is the fact that we would like to examine the blocks adjacent to the newly freed block, and if they are free, coalesce them with the newly freed block. When coalescing is done, our most important statistic, the number of contiguous free bytes, also increases.

To increase how quickly we traverse the linked list of free blocks and to help us examine adjacent blocks, we keep the list sorted by the blocks' offsets. When we free a block, we traverse the list until we find a free block whose offset is greater than that of the newly freed block. Then we insert the newly freed block in the list before this

block. **Figure 6** illustrates a typical coalescing action.

The astute reader will notice that although the user can theoretically request up to 64Kb of memory (the maximum value of the argument to `MyAlloc`), the amount of memory requested is limited to the size of the page block. To accommodate this limit, we can modify the VMM to dynamically modify the basic page block size if a request comes in that is too large.

The Double Indirection Method

Earlier, we alluded to the fact that periodic compaction of the memory blocks would reduce fragmentation in our VMM. Before compaction, the free blocks are intermingled with the allocated blocks, and the free list must be traversed in order to locate a block with the desired amount of contiguous bytes free. After compaction, all the allocated blocks are pushed to one side of the page, and a single large free block is created out of the remaining space. When we look for a free block to allocate, there is only one block to examine in the page.

Compaction presents one major problem, however. If we move an allocated block to another position in memory, we must modify all our application's variables that point to that block to point to the new place in memory. But how does the memory manager know which variables point to that memory block?

To solve this problem, we use a scheme known as double indirection. Double indirection has been made popular by the memory managers of both the Macintosh® and of Microsoft Windows. When an application requests a block of memory, we will return not a pointer to that block, but a pointer to a pointer

to that block. This is illustrated in **Figure 7**.

Figure 8 illustrates what happens when we compact a number of memory blocks. Even though the blocks shift in memory, the pointers to the blocks remain stationary. Since all our application knows about are the indirect pointers, and since the position of those pointers doesn't change, the memory manager does not have to alert the application when the compaction operation occurs. Everything is totally invisible to the application. (Actually, in Microsoft Windows Version 2.x, an application can choose to be alerted when allocated blocks are moved by specifying the `GMEM_NOTIFY` flag in the call to `GlobalAlloc`.)

As you can see from the diagrams in **Figures 7** and **8**, the double indirection method requires one extra memory reference in order to dereference a memory handle. However, with the speed of modern day CPUs increasing yearly, the extra memory reference is not as much of a problem as it used to be. If you consider the reduction of time involved in searching the free list, and also take into account the elimination of fragmentation, you'll see why the designers of Windows chose the doubly indirect way of doing these things.

The McBride Allocator

Let's take a brief look at another virtual memory management package, `VMEM` by Blake McBride. `VMEM` (see **Figure 9**), which also comes with full source code, uses the double indirection method to achieve high performance. Although it does not support expanded memory at this time, it compensates for this deficiency by allowing compaction of the swap file. It also uses one of the enhancements that was men-

Figure 9: The McBride VMEM Virtual Memory Manager API

char far *VM_addr(VMPTR_TYPE voh, int dirty, int fFreeze)
Dereferences the memory handle voh. The variable dirty should not be zero if you are going to change the contents of the memory block. fFreeze is not zero if the block's position in memory should be frozen.
VMPTR_TYPE VM_alloc(long size, int fClear)
Allocates size bytes from the memory pool. If fClear is not zero, the memory block will be cleared to zeros.
VM_dcmps
Initiates compression of the swap file. The compression method may be selected with VM_parm.
int VM_dump(char *filename)
Dumps the entire contents of the VM system to the disk file filename. Returns 0 if the dump was successful, nonzero if not.
void VM_end
Terminates VMEM. The swap file is deleted, but real memory is not released back to the operating system.
void VM_fcure
Same as VM_end, except that all real memory is returned to the operating system.
void VM_free(VMPTR_TYPE voh)
Frees the object referenced by handle voh.
int VM_rest(char *filename)
int VM_frest(char *filename)
Restores the VMM to a previous state that was saved by VM_dump. The main difference between VM_rest and VM_frest is that the VM_rest will read in the memory blocks only as they are needed and is therefore much faster than VM_frest.
int VM_init
Initializes VMEM.
void VM_parm(long rmmmax, long rmasize, double rmcompf, long dmmfree, int dmmfbkls, int dmctype)
Sets various VMEM parameters. Used to fine-tune the system. The arguments are the following:
rmmmax — maximum amount of real memory that VMEM will request
rmasize — minimum amount of real memory that VMEM will request
rmcompf — real memory compression factor
dmmfree — determines when automatic swap file compression occurs
dmmfbkls — another method used to determine when swap file compression occurs
dmmctype — type of swap file compression used
VMPTR_TYPE VM_realloc(VMPTR_TYPE voh, long newsize)
Reallocates the memory block pointed to by voh and returns a handle to the new block.
long *VM_stat
Used to obtain various statistics about the current state of VMEM.

tioned above—it maintains the virtual memory objects on a doubly linked list so that there is always a pointer to the least recently used object.

The user has a choice of two methods of swap file compression. Using the first method, all objects are moved to the beginning of the swap file, so

that one large hole remains at the end. The second method uses two swap files; all allocated objects are copied from the first swap file to a newly created second swap file, then the original swap file is deleted. Although both methods produce the same results, each method has an important advantage.

With the single file method, even though the allocated blocks have been moved, the size of the swap file will never decrease. With the dual-file method, you need twice the disk storage during the compression operation.

VMEM also has the ability to save and restore entire virtual memory images to and from disk. This capability is ideal if, for example, you'd like to save the entire state of the virtual memory system, release the memory back to DOS, shell out a large program (like a compiler), and restore the state of the system when the shelled program is completed.

A study of the VMEM API will reveal that you have a bit more control over the operating parameters with VMEM than you do with our simple memory allocator. The API lets you choose whether you want to clear the memory when you allocate, set the dirty bit of a block when you deference that block, and directly change some of the important operating parameters of VMEM.

Using a VMM as a replacement for heap-based allocation gives an application more flexibility in dealing with huge amounts of data. I believe that virtual memory is best done at the operating system level (for example, using Microsoft OS/2) and kept invisible from all applications in the system. But since a large part of the market is, and will remain for some time to come, firmly entrenched in the DOS world, virtual memory managers are still needed. In the future, all operating systems will have built-in virtual memory and will make use of the dedicated memory management chips produced by semiconductor vendors like Intel and Motorola. Let's hope that that day is not too far away. □

Using the OS/2 Video I/O Subsystem to Create Appealing Visual Interfaces

Richard Hale Shaw

The most noticeable attribute of an application is the way it visually interacts with the user. If the application looks snappy and smart, the user will gain confidence when running it for the first time. If the application appears slow and dull, however, the user will become apprehensive or, worse yet, bored. Thus, from the user's perspective, the screen is the most essential mechanism of the application. To an OS/2 application developer, this makes the video I/O (VIO) the most important of the three subsystems available to OS/2 character-based applications.

An Overview of VIO

The MS-DOS® operating environment provided such limited and inefficient video services, that application developers looked for other means to improve the video throughput of their programs. A great number of DOS applications took advantage of the PC ROM BIOS video services (Int 10h) or wrote to and directly manipulated the video hardware and display buffer. This approach hindered portability, but it was not a problem under the real mode of the DOS operating environment, since only one application at a time was able to access the video hardware.

In contrast, OS/2 systems are endowed with a robust highly efficient set of video services, which comprise the VIO subsystem. The subsystem consists of a set of character-oriented display services of the type that are generally required by the current generation of character-based applications. Think of VIO as a superset of the PC ROM BIOS services (Int 10h) found in real mode under DOS, with the difference being that VIO uses calls instead of an interrupt.

The efficiency and effectiveness of the VIO services are such that there is little need for an application to manipulate the video hardware directly. Perhaps the only reason an OS/2 application



HELLO1.C, a multithreaded version of Hello, world, demonstrates many of the features of the OS/2 video subsystem, including color support.

Richard Hale Shaw is a contributor to various computer magazines and a software engineer at Hilgraeve, Inc.

Figure 1: The OS/2 Logical Video Buffer

The LVB of OS/2 is organized similarly to the Text mode video buffer found on most IBM® PCs. Each screen character is represented by a 2-byte character-attribute pair or cell. Further, each character is accessed at an offset of $((\text{row} * \text{number_of_columns} * 2) + \text{column})$ in the buffer, and the character's attribute is accessed at the same offset plus 1.

For instance, suppose the video screen is in ordinary black and white, 80 × 25 text mode, and the upper-left-hand corner looks like the following:

```
Hello, World
from thread 3
```

Also, suppose a code fragment that looks like the following is given:

```
char far *lvb;
unsigned size;

VioGetBuf((PULONG)&lvb, &size, VIOHDL);
```

The LVB variable can access the material in the video buffer in the following way:

Code Location	Value	Type	Row	Column
lvb[0]	'H'	Character	0	0
lvb[1]	0x07	Attribute	0	0
lvb[2]	'e'	Character	0	1
lvb[3]	0x07	Attribute	0	1
lvb[4]	'l'	Character	0	2
lvb[5]	0x07	Attribute	0	2
lvb[6]	'l'	Character	0	3
lvb[7]	0x07	Attribute	0	3
lvb[8]	'o'	Character	0	4
lvb[9]	0x07	Attribute	0	4
lvb[160]	'f'	Character	1	0
lvb[161]	0x07	Attribute	1	0
lvb[162]	'r'	Character	1	1
lvb[163]	0x07	Attribute	1	1
lvb[164]	'o'	Character	1	2

The attribute values of 0x07 assume that the screen is in standard white-on-black mode. Color, highlighted, or blinking attributes will have different values.

THE PROTECTED MODE OF THE OS/2 OPERATING ENVIRONMENT REQUIRES THAT SYSTEM FACILITIES BE VIRTUALIZED AND SHARED AMONG PROCESSES, INCLUDING ACCESS TO THE VIDEO SCREEN.

would need to access the video hardware while using VIO would be to generate graphics images. VIO has only limited graphics support, but it does allow an application to move to and from graphics mode (this will be discussed in greater detail below). In the event that an application does need to access the video hardware, OS/2 can serialize such accesses and provide a means for applications to do so that's compatible with its protected mode environment.

Although it's essential that a

real operating system like OS/2 offer efficient video services (where DOS does not), there is another, more profound reason for the existence of VIO. The protected mode of the OS/2 operating environment requires that system facilities be virtualized and shared among processes, including access to the video screen and video hardware. An application should not be hindered from performing screen updates while running in the background, but a foreground application should not have its screen trashed by a program running in the background. By using VIO, the user is assured that the video output of one application will not interfere with that of another application. In addition, most VIO calls can be used in bound programs that must run under both OS/2 and DOS. Like the ROM BIOS calls or direct video hardware control code under DOS, VIO calls under OS/2 make a program less portable to the DOS environment and more OS/2 specific. Of course, this is true of all non-FAPI OS/2 calls.

VIO is implemented as a dynamic-link library (found in VIOCALLS.DLL), so a program's references to VIO services are bound at execution time, not at link time. Thus, you can replace VIO functions at any time without recompiling or relinking the client application. This is how the OS/2 Presentation Manager (PM) handles VIO calls. By providing its own version of VIOCALLS.DLL, PM can allow non-PM OS/2 programs to run under it—even in the PM screen group (that is, in a PM window).

VIO calls are efficient, but they do not offer the complex formatting facilities of the printf standard library function nor do they handle output redirection. A subsystem like VIO must be fast, so it's necessary that it

Figure 2: OS/2 VIDEO MODES

avoid the encumbrances of the file system and I/O redirection. In those instances in which redirection is a consideration, however, you can use DosRead and DosWrite. These two kernel services use VIO to handle the screen component of their output. Note that when STDOUT points to a screen device, OS/2 routes it through VIO.

If generic screen output is a consideration, you'll find it simpler and easier to use the standard C library routines, since a typical application will probably use a combination of these and VIO calls. The VIO services do, however, allow considerable flexibility in the way text is written to the screen, including control over color and cursor positioning.

The Logical Video Buffer

As mentioned earlier, OS/2 considers the screen a resource that is able to be simultaneously shared by more than one process. When a session is started, OS/2 creates a logical video buffer (LVB) for it. OS/2 retains control of the physical video buffer (PVB). If a program makes a VIO call, VIO will update the LVB of the program's session and notify OS/2 that the PVB must be updated. While a session is in the foreground, OS/2 will duplicate VIO updates of the LVB in the PVB. Thus, VIO calls always update the two buffers while a session is in the foreground. When you move the session into the background, OS/2 assumes that its screen contents are completely updated in the LVB. Therefore, it doesn't have to save the screen and can quickly update the PVB from the new foreground session's LVB. This makes screen switches very fast and allows OS/2 to virtualize screen access among processes.

While the session is in the background, VIO services will

Adapter	Type	Colors	Columns	Rows	HRes.	VRes.
Mono	Text	-	80	25	720	350
CGA	Text	16	40	25	320	200
CGA	Text	16	80	25	640	200
CGA	Graphic	4	40	25	320	200
CGA	Graphic	16	40	25	320	200
CGA	Graphic	2	80	25	640	200
EGA	Text	16	40	25	320	200
EGA	Text	16	40	25	320	350
EGA	Text	16	40	43	320	350
EGA	Text	16	80	25	640	200
EGA	Text	16	80	25	640	350
EGA	Text	16	80	43	640	350
EGA	Graphic	4	40	25	320	200
EGA	Graphic	16	40	25	320	200
EGA	Graphic	2	80	25	640	200
EGA	Graphic	4	80	25	640	200
EGA	Graphic	2	80	25	640	350
EGA	Graphic	2	80	25	640	200
EGA	Graphic	16	80	25	640	200
EGA	Graphic	16	80	25	640	350
VGA	Text	16	40	50	360	400
VGA	Text	16	80	50	720	400
VGA	Graphic	2	80	30	640	480
VGA	Graphic	16	80	30	640	480
VGA	Graphic	256	40	25	320	200

continue to write to the session's LVB. OS/2 ignores calls from VIO to update the PVB, since the foreground session is using the PVB. When the session is brought into the foreground again, OS/2 will copy the contents of the LVB to the PVB and resume updating the PVB from the LVB. Thus, as long as the application uses VIO services to do its video updates, the LVB will always accurately depict an application's visual state in character mode. This remains true whether the application is in the foreground or background, and it makes VIO a safe means of updating the video screen.

Each LVB is organized in 2-byte pairs, called cells (shown in Figure 1). The number of cells in the LVB will vary with the current video mode (for example, a 43-line mode will require more

cells than a 25-line mode). Each cell corresponds to one character on the screen, with the first byte of each cell containing the character itself and the second byte holding the attribute value (which controls foreground and background color, intensity, and blinking). You can use different VIO services to write characters, attributes, or cells to the video screen.

The VIO subsystem supports the full range of PC-based video adapters and displays, including monochrome, CGA, EGA, and VGA. The 24 different flavors of text and graphics modes that are available are shown in Figure 2. The smallest user-addressable unit of the screen is a character in text mode or a pixel or pel in graphics mode. PC display hardware is memory mapped, so whatever is currently stored in

Figure 3: OS/2 VIDEO ATTRIBUTE BITS

Meaning	Value	Bit No.
Black foreground (character)	0H00	0
Blue foreground (character)	0H01	0
Green foreground (character)	0H02	1
Red foreground (character)	0H04	2
High-intensity foreground	0H08	3
Blue background	0H10	4
Green background	0H20	5
Red background	0H40	6
Blinking character	0H80	7

Figure 4: OS/2 VIDEO COLOR VALUES

Color	Value
Black	0H01
Blue	0H02
Green	0H03
Cyan	0H04
Red	0H05
Magenta	0H06
Brown	0H07
White	0H08

UNDER OS/2, THE CHARACTERISTICS OF A TSR PROGRAM IN THE DOS ENVIRONMENT ARE NO LONGER NEEDED SINCE YOU CAN RUN AN APPLICATION IN ANOTHER SCREEN GROUP AND SWITCH THAT SCREEN GROUP INTO THE FOREGROUND WHENEVER AND FOR HOWEVER LONG YOU LIKE. FROM THE PERSPECTIVE OF ACCESSING A TSR PROGRAM, EVERY PROGRAM IS A TSR UNDER OS/2.

video memory (the PVB mentioned above) is displayed on the screen. Other than the description of the video buffers given above, however, it's not essential that you know how video memory is organized or where it's physically located, unless you're going to access the video hardware directly. Also, note that unlike DOS, OS/2 doesn't use mode numbers to specify a video mode. When getting or setting the video mode, you deal directly with the video characteristics themselves.

As previously mentioned, the attribute byte of each pair or cell controls the colors, intensity, and blinking characteristics of a character displayed on screen. Of the 8 bits in an attribute byte, the lower 3 bits control the foreground color, a value of 0-7. Bit 3 toggles intensity on or off, bits 4-6 control the background color, and bit 7 toggles the blinking characteristic. These are shown in **Figures 3 and 4**.

VIO and PM

As you are probably well aware, an OS/2 system offers a high-powered graphical user interface called the Presentation Manager. PM offers a graphical presentation space, windowing, scroll bars, icons, and interfaces for both a mouse and the keyboard. It runs in its own screen group under OS/2 and can run other programs in PM windows in the same screen group. These capabilities differ from those of VIO applications under OS/2 Version 1.0, where a background process remains in its own screen group and is not visible until a user brings it into the foreground screen group. PM can also run ill-behaved programs (that is, those programs that do not run in a PM window or that write directly to the PVB), but they will run in their own screen group like any other OS/2 program.

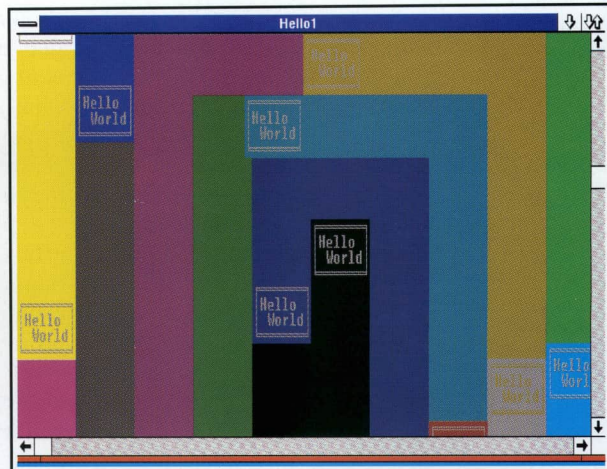
Although the Presentation Manager can do all of the things mentioned, there is still a need for VIO. First, PM applications are complex. They take more effort to write since simple visual ideas can be complex to express in PM code. Second, some applications (a command-line utility, for instance) do not require PM or wouldn't benefit very much from PM's graphical user interface. Other applications will have to be redesigned or completely rethought before they can take advantage of PM. These are problems that VIO can easily solve. In addition, VIO offers a path to OS/2 that requires minimal changes to existing DOS applications. Thus, it's considerably easier to adapt a DOS application to VIO than to PM.

VIO-based programs will run under a PM text window. That's because the versions of VIO (and, for that matter, KBD and MOU) offered with PM (OS/2 Version 1.1) are different from those supplied with OS/2 Version 1.0. The VIO DLL supplied with PM is windows aware and can map each character and its attributes into the appropriate pattern of pixels inside a PM window, while clipping the output of each VIO call to fit the application's window size. The entire process is invisible to both the programmer and the compiled program.

If you use VIO in your OS/2 applications today, you won't pay an extensive penalty later. As OS/2 is ported to other processor families, VIO calls will remain the same and should not require changes to those portions of an application that use VIO. Future releases of OS/2 for the 80286 family of processors won't require a recompile, since new versions of VIO will be supplied with each release. This is why VIO is implemented as a dynamic-link package.



Hello, world is running in a Presentation Manager window. Each frame has a different foreground and background color because an attribute byte for each frame has been added to the FRAME data type.



Chasing frames demonstrate an animated use of the VIO scrolling routines. Hello, world frames are launched from the upper-right-hand corner of the screen and traverse the screen perimeter counterclockwise.

As this series continues, I'll include practical tips that will make it easier for you to produce efficient, high-quality programs that take advantage of OS/2 facilities. If you refer back to the second article in the series, "Planning and Writing a Multithreaded OS/2 Program With Microsoft® C," *MSJ* (Vol. 4, No. 2), there is a discussion of some of the ways a multithreaded program can use up too many CPU cycles, making it difficult for other processes to run (particularly those in the background). You'll see another example of this in the program listing for this article. From the perspective of using OS/2's video system, however, keep in mind that a well-behaved OS/2 program is one that confines itself to the use of VIO (most standard library functions call VIO) and does not access the PVB.

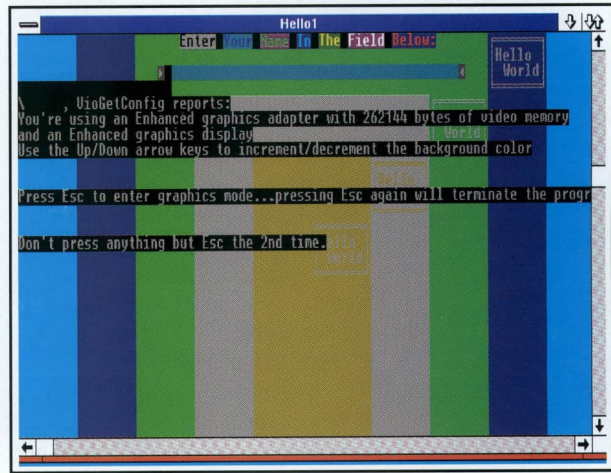
VIO Data Structures

Earlier in this series, I briefly discussed the new OS/2 header files, structures, and type definitions. Because of the variety of objects used to manipulate the screen, there are several data structures designed strictly for VIO programming. These are

displayed in **Figure 5**, which you can refer to as you read about the sample program listing.

The VIO data structures provide all of the information used by the old IBM® PC ROM BIOS calls, and more. But instead of passing information in registers (the convention in the DOS environment), OS/2 places the information in a structure whose address is passed to the appropriate function. Not only does this make retrieving and changing the video environment rather easy, it places a machine-independent interface on the entire mechanism. Moreover, in keeping with other OS/2 functions, it allows the function to return a single unsigned integer as an error return status.

For example, the program listing contains a function (Cursor) that can save or restore the cursor. Notice that it uses the VIOCURSORINFO object type, which includes the starting and ending cursor scan lines, the cursor width (in columns or pixels depending on the current screen mode), and the cursor character attribute. Information in the VIOCURSORINFO object is easily retrieved and/or modified via the VioGetCurType



HELLO1 retrieves the video configuration report and prints the information on the screen.

and VioSetCurType functions, as shown in the listing.

The VIO Pop-up Facility

VIO offers a means by which a background process can temporarily pop up in the foreground or a foreground process can temporarily switch to a blank text screen. If you note the stress on temporary, you'll begin to see that the need for pop-ups is not the same as with DOS. Under DOS, a terminate-and-stay-resident (TSR) application could take control of the processor and pop up over the

current application for an indefinite period of time. Since DOS is a single-tasking environment, this became an acceptable manner of communicating with a TSR program. And since the current application was frozen until the TSR gave up control of the processor, the TSR could stay popped up for as long as you wished.

Under OS/2, the characteristics of a TSR program in the DOS environment are no longer needed since you can run an application in another screen group and switch that screen group into the foreground whenever, as often, and for however long you like. From the perspective of accessing a TSR program, every program is a TSR under OS/2. It's always there, running in its own private screen group, although you can still issue the commands to the application for it to terminate. This is even more obvious under PM, since PM and well-behaved VIO programs that run in the PM screen group can simultaneously display and run in overlapping windows.

Remember that a process can be detached and run in a special screen group, where it does not have to have regular keyboard input or show screen displays. Detached processes are good candidates for pop-up programs under OS/2. You'll doubtless come up with other ideas.

Pop ups are exceptions to the way that OS/2 typically handles the console. Only one pop-up program can be activated at a time: if another process issues a pop-up call, it will be blocked until the first one relinquishes the screen. While a pop-up program is active, the user cannot switch to another process or screen group. In this sense, the pop-up program gains temporary ownership of the foreground screen.

The pop-up facility of OS/2

should only be used by a process that needs to gain temporary control of the physical console (that is, the screen, the keyboard, or the mouse). Since other processes cannot be switched into the foreground during a pop up, the pop-up program must be efficient, do its job, and end the pop-up session quickly. A good example of how the OS/2 pop-up facility is used is the OS/2 Hard Error handler. You can observe its use of the pop-up facility by issuing a DIR command on a disk drive while the drive door is open.

When a pop-up program begins, the screen is automatically placed in 80 by 25 text mode. The screen is restored to the previous mode when the pop-up program ends. The VIO service routines for activating a pop-up program are VioPopUp and VioEndPopUp. The function prototypes for these two routines are shown in **Figure 6**. Note that only a subset of VIO services are available during the pop-up call (shown in **Figure 7**). An example of code that activates a pop-up program can be found in the graphics function in the sample program listing.

Using the VIO API

Any discussion of the VIO Application Programming Interface (API) ought to begin with a reference to Ray Duncan's article, "Character-Oriented Display Services Using OS/2's VIO Subsystem," *MSJ* (Vol. 2, No. 4). It contains an excellent overview of the API and discusses most of the services that VIO provides. This article will now pick up where Duncan's article left off and, in doing so, will walk through the sample program listing presented here. The sample is a modified and enhanced version of HELLO0.C, which was presented in the preceding article in this series. This discussion will

provide insight into the practical application of VIO services.

A New HELLO.C

Recall that HELLO0.C was a multithreaded version of HELLO.C. It was designed to illustrate the use of threads and semaphores to control and serialize access to resources. Although technically sound, it was visually boring, so adding a variety of colors to it is a good idea. In addition, the current version of HELLO1.C (*The actual code listing is available on MSJ's bulletin boards and is not included here due to space constraints—Ed.*) has been restructured and modularized to allow different sample functions to be plugged in with ease.

Probably the first thing you'll notice in HELLO1.C is that the macro INCL_SUB has been defined at the beginning of the program listing. As such, the program automatically includes the function prototypes, macros, and type definitions that will be needed to use VIO functions via the system of header files and defines that were discussed in the earlier article.

Next, note that every VIO function requires a device handle as its last argument. In non-PM versions of OS/2, this handle is always zero. A special set of VIO calls under PM (called advanced VIO, or AVIO) use this argument. The argument is represented by a macro, VIOHDL, in the listings here, making it easy to search for and replace references to it later.

Finally, in fitting with the convention used by all OS/2 API functions, notice that VIO services that return an error code will return a nonzero unsigned value when an error occurs. You can always assume that a VIO service was successful when it returns a zero value. Also note that one of the most commonly used VIO routines, VioWrtTTY,

Direct Screen I/O and Graphics Under VIO

The OS/2 VIO subsystem does not offer any facilities for a process to access the physical screen directly. This can pose problems if you are porting DOS applications that perform direct screen writes or generate graphics or if you are writing an OS/2 application that generates graphics without the use of Presentation Manager (PM). Fortunately, OS/2 offers two indirect means for accessing the physical screen, depending on the needs of the application.

Direct Screen Writes in Text Mode

OS/2 systems provide a logical video buffer (LVB) for each screen group, and all VIO services update the LVB. When a screen group is in the background, screen updates can take place without disrupting the screen of the foreground process. When a screen group is in the foreground, OS/2 duplicates VIO updates of the LVB in the physical video buffer (PVB).

You can easily modify a DOS program that performs text mode direct screen writes to run under OS/2. To do this, however, it's essential that you change the code that updates the PC's video buffer to write to the LVB instead.

First, you must call `VioGetBuf` to retrieve the address of the LVB of your program's screen group. Since the format of the LVB is identical to a PC's text mode video buffer, the direct screen write code remains largely intact—only the destination changes. Note that there is no need to include code that checks for horizontal retrace or that manipulates the video hardware. OS/2 will take care of these details for you. When you are finished updating the LVB, your program should call `VioShowBuf` to update the PVB from the LVB. `VioShowBuf` can update all or a portion of the LVB to the PVB.

This indirect screen write process remains the same whether your program is in the foreground or background: OS/2 will only honor calls to `VioShowBuf` when the program is in the foreground and will ignore such calls when a program is in the background. OS/2 will, however, update the PVB from the LVB once the program is switched to the foreground again. Note that although your program only needs to call `VioGetBuf` once, when it begins, your program can (and should) call `VioShowBuf` as often as is necessary. `VioShowBuf` will do nothing while a process is in a background screen group. This is not a problem since OS/2 will update the PVB upon a screen switch.

I should mention that `VioShowBuf` is very fast. I modified the Magma Systems ME Editor to perform screen paging by writing to the LVB and occasionally calling `VioShowBuf`. This improved the speed of screen page updates and made the calls to `VioShowBuf` appear every bit as fast as direct screen writes under DOS.

VIO function calls in a foreground process automatically update both buffers, since they notify OS/2 that the PVB must be updated. You can see this for yourself by writing a piece of code that first writes some text directly to the LVB, then calls a VIO service routine to write additional text, which overwrites part of the text written in the first step. This updates the LVB and causes OS/2 to update the PVB. Finally, have the code call `VioShowBuf` or manually switch the program into the background and back again. Until the PVB is updated, only the output of step 2 will appear on the screen. The text written in step 2 that overlays the text written in step 1 will, however, survive the screen update after the call to `VioShowBuf`. This also illustrates that, in text mode, the LVB always contains the current visual state of an application.

Going Around VIO to Generate Graphics

Although it's relatively simple to perform direct screen writes in text mode, it's not as easy to work with graphics. Part of the problem is that you can't use the LVB, since it's only available for direct screen writes in text mode. Therefore, if your program is going to generate graphic images, you'll need to access the PVB directly. Fortunately, VIO has a mechanism that allows an application to step around it and manipulate the PVB directly. This makes it easier to port graphics applications from DOS and enables you to write OS/2 graphics applications without the complexity of PM programming.

There are a few caveats to be aware of when writing to the PVB. First, a routine can only access one adapter and video mode at a time. To support several adapters in a graphics routine, you'll have to write code that can address each, making the code somewhat hardware dependent.

Next, images drawn in the PVB by an application are lost when a screen switch takes place. As I said earlier, you can't use the LVB in graphics mode. Further, there is no built-in mechanism for saving the contents of the PVB when a screen switch takes place. Since it's not built in, you'll have to provide this mechanism yourself. If your program manipulates any of the video hardware, you'll also have to save that information when a screen switch takes place.

Finally, an application that accesses the physical screen must run in its own screen group, so it will never run in a PM window. When run in the PM environment, PM gives the program its own full-screen window. If you want it to run in a PM window, you'll have to rewrite the program to take advantage of the PM graphic facilities.

With these warnings in mind, there are two problems to solve when writing a program that generates graphics in a VIO context. How do you use the mechanism for obtaining

CONTINUED

Direct Screen I/O and Graphics Under VIO

access and writing to the physical screen? And how do you provide a facility to save and restore the screen before and after a screen switch?

The Direct Screen Write Process

First, consider how to access and write to the physical screen buffer. Your program should call `VioSetMode` to ensure that the display is in the correct graphics mode. Then you must obtain the address of the PVB by calling `VioGetPhysBuf`, which will return a selector that corresponds to the address of the physical buffer. When you're ready to write to the buffer, you must use `VioScrLock` to lock the screen so a background application will not switch screens during the update. Next, your program should perform the direct screen write itself. This should be fast, efficient code, since a screen lock temporarily hangs the system and prevents a screen switch from taking place. Finally, your program must call `VioScrUnLock` to unlock the screen.

Note that once you've retrieved a selector for the physical buffer address, the remaining steps must be taken every time your program is going to access the display hardware and write to the screen. These steps are essential to inhibit screen switches (which will disrupt the display) and to prevent other processes from writing to the display at the same time.

`VioScrLock` is also a fast, effective way to determine whether or not your program is in the foreground, since the caller can either choose to block if the screen is not available or to return immediately. If the screen is unavailable, the program can do other processing until it is. Probably the most effective way to use `VioScrLock` is to place the direct screen write code in a function that's executed by a separate thread. You can use a semaphore to tell the thread when to perform a screen update, and the thread can block on `VioScrLock` until the screen can be accessed. This leaves the main thread free to continue with other work and lets you structure the code so that a graphics-intensive I/O can be written quickly, in its entirety, without a great deal of overhead.

`VioScrLock` is so effective in locking the screen that nothing can switch the screen while it's locked. It protects the system from the application and protects the application from the system. Even pop ups (including the hard error handler) cannot be activated during a screen lock. As a safety valve, OS/2 will cancel the lock and perform a screen switch after 30 seconds if a program or the user has requested it. If your graphics application is still drawing an image when the screen switch takes place, the picture will be disrupted.

Saving/Restoring the Video Display and State

The second problem is to see how an application that manipulates the video hardware can save and restore the entire video state. The contents of the display buffer, the video mode, the palettes, the cursor, and so on, will all be lost if a screen switch takes place. VIO cannot perform this service for you, since a graphics bitmap might require a great deal of memory to save and restore the display. Further, some video hardware adapters have write-only registers that cannot be saved for later restoration—only your program will know whether or not it changed a video register.

Thus, OS/2 can only notify a process when a screen switch is about to take place. It becomes the application's job to save and restore the screen when so notified. This is accomplished by creating a thread whose job it is to save and restore the screen. You can register the thread function with OS/2; then, just before the screen switch takes place, OS/2 will activate the thread. This lets you construct a thread to do one job and do it well—to save or restore the screen and video mode at the appropriate time. Thus, the thread's structure, minus the details of saving/restoring the display and video mode, is quite simple. The thread enters a loop and immediately calls `VioSaveRedrawWait`. This function will cause the thread to block until OS/2 indicates that a screen switch is about to take place. When this happens OS/2 will cause `VioSaveRedrawWait` to return, at which time the thread must perform the screen save or restoration. Returning to the top of the loop, the thread again calls `VioSaveRedrawWait`, which notifies OS/2 that the screen switch can continue. The thread will continue to block until the next screen switch for this process occurs.

Note that upon its return, `VioSaveRedrawWait` will indicate whether a screen save or restore should transpire. And, as with `VioScrLock`, the system is vulnerable while the save/restore thread is active, so the thread function must be as efficient as possible. Finally, note that a similar function, `VioModeWait`, is provided if your program only needs to save the video mode and not the display. `VioModeWait` is used in the same fashion as `VioSaveRedrawWait`.

Please keep in mind that the discussion presented here pertains to graphics applications that need to circumvent VIO or PM. It's not an issue with ordinary text applications. There's nothing to stop you from using `VioGetPhysBuf` to write to the screen in text mode, if you choose. The efficiency you gain will, however, be offset by the code overhead needed to save and restore the screen. That's why the `VioGetBuf/VioShowBuf` combination is provided for text applications.

is now called VioWrtTTY in OS/2 Version 1.1, which has been released since this series was initiated.

The Startup Function

As you see from the listing, main has been simplified to include only a handful of functions. The first of these, startup, processes the optional command-line arguments. As in HELLO0.C, these arguments are the number of milliseconds used in calls to DosSleep, which allow you to vary the sleep time between thread events. If you set these values too low, the program will hog CPU time and radically slow down OS/2 and other programs. These arguments are discussed in more detail below.

Startup performs several other services for the program. It initiates a keyboard thread (discussed below). Then it calls the Screen function to save the current video display for restoration when the program terminates. Screen takes a single argument, which determines if the screen should be saved or restored. It assumes that the calling program's screen group is in text mode and calls VioGetBuf to retrieve the address of the screen group's LVB and the size of the buffer. Thus Screen can allocate storage to hold the LVB contents and copy the contents of the LVB to the new storage. To restore the display when the program terminates, Screen copies the contents of the storage buffer back to the LVB and calls VioShowBuf to update the physical display. This process is further discussed in the sidebar, "Direct Screen I/O and Graphics under VIO".

Startup also calls the Cursor function to hide the cursor and save the cursor position. Like Screen, the Cursor function takes a single argument, whose bits determine whether Cursor

Figure 5: OS/2 VIO Data Structures

Note that several of the structures shown below include a 'cb' member which designates the size of the structure in bytes. This field must be set to that size by the calling program before calling the appropriate VIO service routine. The Sizeof() macro in the example program performs this exact function.

```
typedef struct _VIOCURSORINFO
{
  USHORT yStart;          /* top cursor scan line */
  USHORT cEnd;            /* bottom cursor scan line */
  USHORT cx;              /* cursor width */
  USHORT attr;            /* cursor attribute character */
} VIOCURSORINFO;
```

- Note that cx represents the width of the cursor in columns for text mode or pixels when in graphics mode
- You can hide the cursor by setting attr to 0xffff.
- This structure is used by VioGetCurType and VioSetCurType.

```
typedef struct _VIOMODEINFO
{
  USHORT cb;              /* length of structure in bytes */
  UCHAR fbType;          /* screen mode */
  UCHAR color;           /* number of color bits */
  USHORT col;            /* number of text columns */
  USHORT row;            /* number of text rows */
  USHORT hres;           /* number of pixel columns */
  USHORT vres;           /* number of pixel rows */
  UCHAR fmt_ID;          /* reserved, must be 0 */
  UCHAR attrib;          /* reserved, must be 0 */
} VIOMODEINFO;
```

- Note that the fbType member contains one or more of the following values to designate the graphics mode:
 - 0 - monochrome
 - VGMT_OTHER - non mono enabled
 - VGMT_GRAPHICS - graphics mode enabled
 - VGMT_DISABLEBURST - color burst disabled

- The color member is the number of colors as a power of 2 (i.e., the number of color bits that define the color):
 - 1 2 colors
 - 2 4 colors
 - 4 16 colors

- This structure is used by VioGetMode, VioSetMode.

```
typedef struct _VIOPHYSBUF
{
  PBYTE pBuf;            /* video buffer address */
  ULONG cb;              /* video buffer length in bytes */
  SEL asel[1];           /* array of selectors */
} VIOPHYSBUF;
```

- The pbuf member is the 32-bit physical address of the first byte in the physical video buffer. This will vary with the video mode being selected, and must be in range from 0xa0000 to 0xbffff. You will, of course, have to know what address a particular video mode uses in order to set this member.
- The asel member is the beginning of an array of selectors used to address the physical buffer--1 selector for every 64k of buffer space. Thus, the number of selectors will vary with the size of the video buffer. You should be sure and allocate enough space for the structure and additional selectors, if needed.
- The cb field includes the total size of the structure plus the size of the additional selectors, if present.
- This structure is used by VioGetPhysBuf.
- The adapter member can have the following values:
 - 0x0000 mono
 - 0x0001 color graphics adapter
 - 0x0002 enhanced graphics adapter

CONTINUED

Figure 5 CONTINUED

0x0003 video graphics array or PS/2 adapter

- The display member can have the following values:

0x0000 monochrome
0x0001 color
0x0002 enhanced graphics display
0x0003 8503 monochrome
0x0004 8512, 8513 or 8514 color

- This structure is used by VioGetConfigInfo and VioSetConfigInfo.

```
typedef struct _VIOFONTINFO
{
    USHORT cb; /* length of structure in bytes */
    USHORT type; /* request type */
    USHORT cxCell; /* width of characters */
    USHORT cyCell; /* height of characters */
    PVOID pbData; /* buffer or font address */
    USHORT cbData; /* length of font in bytes */
} VIOFONTINFO;
```

- The type member can contain the following values:

VGFI_GETCURFONT - to retrieve the current font
VGFI_GETROMFONT - to retrieve ROM font

- The cxCell member is the width in pixels of each character cell in the font.
- The cyCell member is the height in pixels of each character cell in the font.
- The pbData member points to a buffer that receives the requested font table. Alternatively, you can set it to 0x00000000, and VioGetFont can supply an address. VioGetFont will then copy the address of the font to the pbData field.
- This structure is used by VioGetFont and VioSetFont.

```
typedef struct _VIOPALSTATE
{
    USHORT cb; /* length of structure in bytes */
    USHORT type; /* request type */
    USHORT iFirst; /* first register to retrieve */
    USHORT acolor[1]; /* array to receive color values */
} VIOPALSTATE;
```

- The cb field must include the length of the structure in bytes plus the size of the additional array members to hold the palette registers, if present.
- The type field must be 0 to retrieve the palette register state.
- The iFirst field designates the first register to be retrieved and can be a value from 0x0000 to 0x000f.
- This structure is used by VioGetState and VioSetState.

```
typedef struct _VIOOVERSCAN
{
    USHORT cb; /* length of structure in bytes */
    USHORT type; /* request type */
    USHORT color; /* color value */
} VIOOVERSCAN;
```

- The type field should be set to 0x0001 to retrieve the border color.
- This structure is used by VioGetState and VioSetState.

```
typedef struct _VIOINTENSITY
{
    USHORT cb; /* length of structure in bytes */
    USHORT type; /* request type */
    USHORT fs; /* foreground/background status */
} VIOINTENSITY;
```

- The type field should be set to 0x0002 to retrieve the blink/background intensity switch.
- The fs field designates the foreground/background color status: 0x0000 for blinking foreground, or 0x0001 for high-intensity background.
- This structure is used by VioGetState and VioSetState.

should hide, save, or restore the cursor (that is, position it and make it visible). It uses VioGetCurType to retrieve the cursor character attribute and VioSetCurType to change the attribute (and hide the cursor). It calls VioGetCurPos to save the current cursor position and VioSetCurPos to reposition the cursor. Note that the Screen and Cursor functions rely on the VIOCURSORINFO data type.

Startup also resets the screen rows to the highest number possible. It does this by calling VioGetMode to get the current video mode information (into a VIOMODEINFO object type). Then it sets the row parameter to either 50, 43, or 25 lines and calls VioSetMode until it is successful. This lets the program run at 50 lines on a VGA monitor, 43 lines on an EGA monitor, and 25 lines otherwise. The original row count is saved so the termination function can restore it before the program exits.

After startup, main calls the initialization function shown in the listing. This code is virtually identical to the code used by HELLO0.C in the preceding article and initializes the FRAME structures.

Flickering Frames

The flickering frames of HELLO0.C that greeted you with "Hello, World from thread..." are now encapsulated in the flicker_frames function. The code for this function is very similar to that in the original program. The function will create about two dozen threads; the exact number will vary with the number of screen rows. Each thread carries the responsibility of displaying or clearing its frame on cue from its semaphore, which is stored in its FRAME data type. And each thread shares the code found in the hello_thread function.

Hello_thread now calls

VioWrtCharStrAtt to do its screen I/O. This is a VIO service routine that writes a string at a specified row and column location with a specific attribute. Note that an attribute byte for each frame (and thus each frame thread) has been added to the FRAME data type and is set and manipulated by flicker_frames. Thus, whenever each frame appears it has a different foreground and background color. The flicker_frames function masks off the blinking bit in the attribute so that the frames do not blink. In addition, the function increments the FRAME attribute byte whenever the foreground and background colors match, thereby ensuring that the foreground and background contrast.

In order to accommodate the changes to the FRAME type, the program now includes a filler byte. This ensures that the threadstack member is aligned on an even-numbered address. The _beginthread library routine will not successfully create a thread whose stack falls on an odd-numbered address.

The operator initiates the termination of the flicker_frames function by pressing the Esc key. This activates the keyboard thread, which is contained in the keyboard_thread function mentioned above. This thread blocks on keyboard input and continues to block until the Esc key is pressed. Then it blocks on a semaphore, doneSem, until flicker_frames clears the semaphore. Clearing the semaphore allows keyboard_thread to begin the termination sequence and flicker_frames to postpone the sequence until it has completed a round of activating the frame threads.

Once the semaphore clears, keyboard_thread continues and sets the done flag, thereby notifying flicker_frames that it can terminate the frame threads.

Flicker_frames will cause each thread to run one final time, clear its box (if it is not already cleared), and terminate. As each frame thread terminates, it clears its own frame semaphore to notify flicker_frames that it is terminating. In the meantime, flicker_frames calls the WaitForThreadDeath function, which blocks on each thread's semaphore, and returns to the flicker_frames when all the threads have terminated. Then flicker_frames returns to main.

Chasing Frames

New to HELLO1.C are chasing frames, which demonstrate an animated use of the VIO scrolling routines. Encapsulated in the chasing_frames function, the chasing frames are a series of Hello, World frames that are launched from the upper-right-hand corner of the screen and traverse the screen perimeter counterclockwise. Each successive circuit takes place one frame height and width inside the last, so that the frames gradually make their way to the center of the screen. Another frame follows after a period of time (which is set by one of the command-line sleeptime arguments). The VIO scrolling functions, VioScrollLf, VioScrollDn, VioScrollRt, and VioScrollUp, not only cause the frames to chase each other, they also leave a trail of each frame's color behind. Each chasing frame is managed by its own thread, thereby giving a visual confirmation to the reality of independent threads of execution. The threads share the code found in the box_thread function.

Again the keyboard_thread function plays a role in terminating these threads. After servicing flicker_frames, the keyboard thread goes into a loop, blocking on keyboard input and ignoring all but the Esc key. When the Esc key is finally

pressed, the keyboard thread will again set the done flag and terminate. This in turn notifies chasing_frames not to start any additional frames. It also notifies those frame threads that have already started to terminate themselves. If a frame thread is in the middle of scrolling around the screen, its sleeptime will fall to zero and it will race to the upper-right-hand corner of its current circuit. Then it will clear its own semaphore and terminate. As flicker_frames did before it, chasing_frames calls WaitForThreadDeath to wait until all the frame threads have died; it then returns to main.

There is one catch to the way the keyboard thread works. The user must press the Esc key before the last chasing frame completes its final circuit. Otherwise, the keyboard thread will continue to block until the Esc key is pressed. The next section of the program assumes that the keyboard is available, so the keyboard thread must have terminated by that time. Unfortunately, there is no kernel routine for one thread to kill another. In any case, it would have made keyboard_thread even more complex to cause it to terminate some other way.

The Video Configuration

The next function called from main is displayconfig. This function calls get_user_name to prompt the user for his or her name. To do this, get_user_name displays a multicolored prompt, and below it, a highlighted input field delimited with two markers. It uses VioWrtCellStr, which writes a series of character-attribute cell combinations from those stored in the cellstr array to create the multicolored prompt. Then it calls VioWrtNCell, which can replicate a single cell, to create the field and markers. Finally, it restores the cursor,

Figure 6: VIO Functions Allowed While a Pop Up is Active

VioEndPopUp	VioGetAnsi	VioGetCurPos
VioSetCurPos	VioGetCurType	VioGetMode
VioScrollDn	VioScrollLf	VioScrollRt
VioScrollUp	VioWrtCellStr	VioWrtCharStr
VioWrtCharStrAtt	VioWrtNAttr	VioWrtNCell
VioWrtNChar	VioWrtTTY	

Note that these functions essentially allow only a subset of VIO activities: printing text to the screen, controlling text attributes, modifying the cursor shape and position, scrolling text, clearing the screen and setting the display mode.

sets it to the beginning of the field, and calls the KBD routine, `KbdStringIn`, to read the user's name. After hiding the cursor, it returns the number of characters read by `KbdStringIn`.

When `displayconfig` returns from `get_user_name`, it knows how long the user's name is, but it does not have the name itself. So it calls `VioReadCharStr` to read the characters directly from the screen. Next, it retrieves the video configuration with `VioGetConfig`, prints that information on the screen, and returns to main.

Background Colors

The `togglebackground` function illustrates the use of two other VIO services: `VioReadCellStr` and `VioWrtNAttr`. The former reads one or more cells from the screen. The `Background` function, which is called by `togglebackground`, uses it to retrieve the attribute of the character in the upper-left-hand corner of the screen. It then increments or decrements the attribute value (depending on the argument passed to it) and calls `VioWrtNAttr` to flood the screen with this attribute. Thus, `togglebackground` can prompt the user to use the arrow keys to increment or decrement the background color. Note that exactly which color it uses the first time depends on the color of the character in the upper-left-hand corner. This may have been set by one of the chasing frames. The Esc key is used to

terminate the `togglebackground` function and return to main. An additional message (discussed below) is printed, warning the user to press only the Esc key while in graphics mode.

Generating Graphics

The last facet of VIO that `HELLO1.C` illustrates is how to generate graphics and create a pop-up program under OS/2. The graphics function changes the video mode to CGA graphics with a 320 by 200 resolution and writes a graphical Hi in magenta on a white field. The function creates a separate thread of execution, which saves and restores the screen before and after a screen switch (see the sidebar, "Direct Screen I/O and Graphics under VIO"). Finally, it waits for the user to press the Esc key to terminate. If, however, the user presses any key (other than Alt-Esc or Ctrl-Esc), a pop up will be generated that informs the user to press the Esc key.

The graphics function works by first retrieving the current video mode information and changing the video mode parameters. It sets them to indicate a video mode of non-monochrome graphics, with four colors, 40 by 25 characters, and 320 by 200 pixels. Then it calls `VioSetMode` to change the video mode, followed by `VioGetPhysBuf` to retrieve a selector for the CGA video buffer. It then locks the screen with a call to `VioScrLock`.

The function continues by creating a new thread that exe-

cutes the `SaveRestoreScreen` function discussed below. It calls `clear_graphics_screen` to clear the screen and writes a large Hi in the middle of the screen. The screen write uses two loops to traverse a bitmap of character 1s and 0s (stored in the `hi_bitmap` array) and calls the `putpixel` function to write a magenta pixel whenever a 1 is encountered in the bitmap. You can get an idea of what this will look like by staring at the `hi_bitmap` array at the beginning of the listing and refocusing your eyes slightly.

Last of all, graphics calls `VioScrUnLock` to unlock the screen and goes into a loop. In the loop it calls `KbdCharIn` to block on keyboard input. If the Esc key is pressed, the function breaks out of the loop and returns to main. Otherwise, it calls `VioPopUp` to create a blank 80 by 25 text mode screen, prints a message, and waits for a single keystroke. Upon receiving the keystroke, the function calls `VioEndPopUp` to end the pop up and loop back to `KbdCharIn`.

The graphics function illustrates two important points: how to generate graphics under OS/2 and how to save and restore the screen in graphics mode. When you run the program, try some screen switches and press the wrong key to generate the pop up. You will find that the warning message (at the end of `togglebackground`) not to press anything but Esc during the graphics display is a taunt to trick the user into generating the pop up. The `SaveRestoreScreen` function is called by OS/2 each time the wrong key is pressed to save or restore the graphics image and mode.

`SaveRestoreScreen` starts by calling `VioGetMode` to save the current video mode information. Following that, it goes into a loop and immediately calls `VioSavRedrawWait`. This call

Figure 7: VIO Services Discussed in the Text

```
USHORT APIENTRY VioEndPopUp ( HVIO );
USHORT APIENTRY VioGetBuf ( PULONG, PUSHORT, HVIO );
USHORT APIENTRY VioGetConfig ( USHORT, PVIODEVICEINFO, HVIO);
USHORT APIENTRY VioGetCurPos ( PUSHORT, PUSHORT, HVIO );
USHORT APIENTRY VioGetCurType ( PVIODEVICEINFO, HVIO );
USHORT APIENTRY VioGetMode ( PVIODEVICEINFO, HVIO );
USHORT APIENTRY VioGetPhysBuf ( PVIODEVICEINFO, USHORT );
USHORT APIENTRY VioPopUp ( PUSHORT, HVIO );
USHORT APIENTRY VioReadCellStr ( PCH, PUSHORT, USHORT, USHORT, HVIO );
USHORT APIENTRY VioReadCharStr ( PCH, PUSHORT, USHORT, USHORT, HVIO );
USHORT APIENTRY VioSavRedrawWait ( USHORT, PUSHORT, USHORT);
USHORT APIENTRY VioScrLock ( USHORT, PBYTE, HVIO );
USHORT APIENTRY VioScrUnLock ( HVIO );
USHORT APIENTRY VioScrollDn ( USHORT, USHORT, USHORT, USHORT, USHORT, PBYTE, HVIO );
USHORT APIENTRY VioScrollLf ( USHORT, USHORT, USHORT, USHORT, USHORT, PBYTE, HVIO );
USHORT APIENTRY VioScrollRt ( USHORT, USHORT, USHORT, USHORT, USHORT, PBYTE, HVIO );
USHORT APIENTRY VioScrollUp ( USHORT, USHORT, USHORT, USHORT, USHORT, PBYTE, HVIO );
USHORT APIENTRY VioSetCurPos ( USHORT, USHORT, HVIO );
USHORT APIENTRY VioSetCurType ( PVIODEVICEINFO, HVIO );
USHORT APIENTRY VioSetMode ( PVIODEVICEINFO, HVIO );
USHORT APIENTRY VioShowBuf ( USHORT, USHORT, HVIO );
USHORT APIENTRY VioWrtCellStr ( PCH, USHORT, USHORT, USHORT, HVIO );
USHORT APIENTRY VioWrtCharStrAtt ( PCH, USHORT, USHORT, USHORT, PBYTE, HVIO );
USHORT APIENTRY VioWrtNAttr ( PBYTE, USHORT, USHORT, USHORT, HVIO );
USHORT APIENTRY VioWrtNCell ( PBYTE, USHORT, USHORT, USHORT, HVIO );
```

registers the thread with OS/2, causing the thread to block until a screen switch of some kind occurs. At this point, it doesn't matter whether the user switches sessions (using Alt-Esc), brings up the Session Manager (with Ctrl-Esc), or presses an inappropriate key that causes the graphics function to generate a pop up. If you want a screen switch to take place, OS/2 will activate the SaveRestoreScreen thread by returning from VioSavRedrawWait.

Once SaveRestoreScreen returns from VioSavRedrawWait, it quickly saves the screen by copying the contents of the video screen to a buffer. If a screen restore has to take place, the function restores the screen by resetting the video mode and copying the buffer contents back to the video screen.

The graphics function uses two support functions. The first, clear_graphics_screen, simply floods the pixels in the video buffer with FFH, turning them

white. The second, putpixel, converts pixel row-column coordinates into pixel offsets and stuffs the color bits passed into the appropriate part of a byte into the video buffer.

Terminating the Program

The termination function completes HELLO1.C. It resets the screen rows to the original number (restoring the original video mode at the same time), calls Screen to restore the screen to its original state, and calls Cursor to restore the cursor. Finally, it calls DosExit to terminate the program.

Here's a summary of how to run the program. The program goes through five stages: flickering frames, chasing frames, the name prompt, the configuration display, and graphics. The flickering frames will run indefinitely until Esc is pressed. The chasing frames will continue until the frames are exhausted, but for best results, terminate them before the last frame

HELLO1.C ILLUSTRATES HOW TO GENERATE GRAPHICS AND CREATE A POP-UP PROGRAM UNDER OS/2. THE GRAPHICS FUNCTION CHANGES THE VIDEO MODE TO CGA GRAPHICS WITH A 320 BY 200 RESOLUTION. THE FUNCTION CREATES A SEPARATE THREAD OF EXECUTION, WHICH SAVES AND RESTORES THE SCREEN BEFORE AND AFTER A SCREEN SWITCH.

Figure 8: Additional VIO Services Not Discussed in the Text

USHORT APIENTRY VioGetAnsi (PUSHORT, HVIO);

USHORT APIENTRY VioSetAnsi (USHORT, HVIO);

These two functions are used to retrieve or modify the ANSI flag, which controls the processing of ANSI escape sequences by VioWrtTTY.

USHORT APIENTRY VioGetCp (USHORT, PUSHORT, HVIO);

USHORT APIENTRY VioSetCp (USHORT, USHORT, HVIO);

These functions retrieve or set the code page (displayed character set) for the current screen group.

USHORT APIENTRY VioGetFont (PVIOFONTINFO, HVIO);

USHORT APIENTRY VioSetFont (PVIOFONTINFO, HVIO);

These functions retrieve or set the font used to display characters on the screen.

USHORT APIENTRY VioGetState (PVOID, HVIO);

USHORT APIENTRY VioSetState (PVOID, HVIO);

These functions retrieve or set the palette-register values, the border color, and the blink/background intensity switch.

USHORT APIENTRY VioModeUndo (USHORT, USHORT, USHORT);

USHORT APIENTRY VioModeWait (USHORT, PUSHORT, USHORT);

Like VioSavRedrawWait, VioModeWait blocks a thread until the video mode is about to change, and unblocks the thread to restore the video mode. VioModeUndo cancels a VioModeWait request.

USHORT APIENTRY VioPrtSc (HVIO);

USHORT APIENTRY VioPrtScToggle (HVIO);

VioPrtSc performs the same function as the PrintScrn key, by copying the contents of the screen to the printer. VioPrtScToggle enables or disables the printer echo feature.

USHORT APIENTRY VioRegister (PSZ, PSZ, ULONG, ULONG);

USHORT APIENTRY VioDeRegister (void);

These two functions allow you to register (or de-register) an alternate video subsystem within a screen group. You can replace up to two default video functions per call with VioRegister. VioDeRegister restores the default video sub-system.

USHORT APIENTRY VioSavRedrawUndo (USHORT, USHORT, USHORT);

This function, like VioModeUndo, cancels a previously issued request to be informed of a screen switch (via VioSavRedrawWait).

USHORT APIENTRY VioWrtCharStr (PCH, USHORT, USHORT, USHORT, HVIO);

This function writes a character string to the screen at the specified row-column coordinates.

USHORT APIENTRY VioWrtNChar (PCH, USHORT, USHORT, USHORT, HVIO);

This function writes a character to the screen a specified number of times.

USHORT APIENTRY VioWrtTTY (PCH, USHORT, HVIO);

This function writes a character string to the screen at the current cursor position. It advances the cursor as it writes each character and wraps to the next line if necessary (including screen scroll). It can process ANSI escape sequences (if the ANSI flag is ON—see VioSetAnsi), and can process carriage-return and line-feed characters like printf does.

makes its way to the center. Since a chasing frame normally appears before its predecessor moves off the top line, it should be easy to identify. The number of frames is set in the initialization function and will vary on different video hardware.

Once you have entered your name at the name prompt, you can use the Up and Down arrow keys to change the background color. You can continue indefinitely changing the background until you press Esc. When you press Esc, the graphics display

will terminate. If you press any other keys, it will generate the pop-up screen. Try recompiling the program with the SaveRestoreScreen thread commented out. The graphics screen will be trashed to some degree if you switch screens without this thread operating.

Finally, you may find it interesting to alter the sleeptime variables in the program. Three of these can be altered on the command line. The command line defaults are

```
HELLO1 1 1500 1
```

which places a minimum of 1 millisecond between the activation of each flickering frame, 1500 milliseconds (1.5 seconds) between each chasing frame, and a 1 millisecond pause when a chasing frame returns to the upper-right-hand corner of its circuit. The most interesting results occur when using 0 instead of 1 and a low number (like 100) instead of 1500. Keep in mind, however, that 0 will cause the program to steal a lot of CPU time from other processes. But it's fun to watch the results.

This concludes our examination of the OS/2 VIO subsystem. Additional functions and services, which are not essential to a typical OS/2 application but should be mentioned, are listed and briefly discussed in **Figure 8**.

You are now ready to tackle VIO. In fact, you can even write some sophisticated applications. With a good feeling for VIO, you'll find the rest of the OS/2 API no more difficult than gaining knowledge of how and when to use the different services. This will become even more apparent in the next article in this series, which switches from visual output to keyboard and mouse control, and will explore the KBD and MOU subsystems. □

Investigating the Debugging Registers of the Intel[®] 386[™] Microprocessor

Marion Hansen and Nick Stuecklen

The more complex a program and the microprocessor executing it, the bigger the task of debugging. In very complex systems, external debuggers simply cannot do the whole job. Fortunately, debugging sophisticated software created for the 386 hardware environment is made easier by the fact that the Intel[®] 386[™] microprocessor (hereafter referred to as μ p) itself provides a substantial set of internal debugging support features. In this article we will take a look at these 386 features and explore how today's state-of-the-art debuggers use them.

In the Past...

In the past, PC-based debuggers could only set code breakpoints, not data access breakpoints, and setting those code breakpoints was restricted to random access memory (RAM).

Data monitoring utilities could be used, but they merely attach themselves to the timer interrupt and periodically examine and display the specified region of memory. Typically, a debugger worked by overwriting the first byte of the specified instruction, which preempted the ability to set breakpoints in read only memory (ROM). If you wished to set a breakpoint at an instruction, an old-style debugger usually saved the first byte of that instruction and then overwrote the byte with a 0CCH opcode (INT 3). When the microprocessor executed the 0CCH opcode, it generated an INT 3, and the debugger, which was monitoring INT 3, was invoked. The debugger displayed the breakpointed instruction and the register state, then waited for the next command.

Marion Hansen is a senior technical writer with Intel's Personal Computer Enhancement Operation in Hillsboro, Oregon. Nick Stuecklen is a senior software engineer at Intel's PCEO. He has worked with both the Motorola[®] 68000 and Intel 386 microprocessors and has developed on minicomputer, MULTIBUS[®], and PC Bus architectures.

Figure 1: 386 Debug Registers

	31			23			15			7			0							
Control Register	LEN	R/W	LEN	R/W	LEN	R/W	LEN	R/W	**	*	***	G	L	G	L	G	L	G	L	DR7
	3	3	2	2	1	1	0	0				E	E	3	2	1	1	0	0	
Status Register	Reserved by Intel									B	B	B	Reserved by Intel			B	B	B	B	DR6
										T	S	D				3	2	1	0	
Address Registers	Reserved by Intel																			DR5
	Reserved by Intel																			DR4
	Breakpoint 3 Linear Address																			DR3
	Breakpoint 2 Linear Address																			DR2
	Breakpoint 1 Linear Address																			DR1
	Breakpoint 0 Linear Address																			DR0

*Reserved by Intel

WITH ITS SIX DEBUG REGISTERS, THE 386 MICROPROCESSOR PROVIDES BUILT-IN DEBUGGING SUPPORT TO LET YOU SET BREAKPOINT ADDRESSES AND DEFINE WHEN BREAKPOINTS WILL OCCUR.

Figure 2: Aligning Breakpoint Addresses

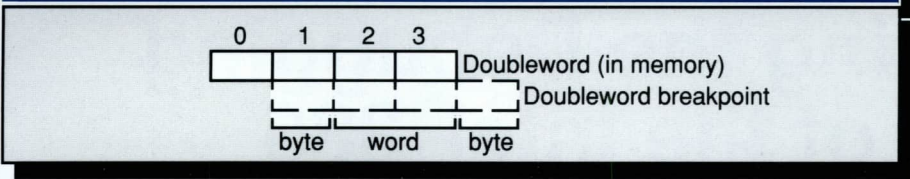


Figure 3: Example Breakpoint Addresses

	DR0 Address: 0A0001 (length 1)	DR1 Address: 0A0002 (length 1)	DR2 Address: 0B0002 (length 2)	DR3 Address: 0C0000 (length 4)
Addresses Trapped	0A0001 (length 1)	0A0002 (length 1)	0B0002 (length 2)	0C0000 (length 4)
	0A0001 (length 2)	0A0002 (length 2)	0B0001 (length 4)	0C0003 (length 1)
Addresses Not Trapped	0A0000 (length 1)	0A0003 (length 4)	0B0000 (length 2)	0C0004 (length 4)

Today...

Fortunately the introduction of the 386 μ p changed all that. With its six debug registers, the 386 μ p provides built-in debugging support to let you set breakpoint addresses and define when breakpoints will occur. The four debug address registers are individually programmable and individually enabled or disabled through the debug control register. The debug status register maintains debug status information.

In addition to supporting the usual *break on* instructions, the 386 μ p also supports data access breakpoints. Data breakpoints are a very useful debugging tool (besides being an exceptional capability for a microprocessor). A data breakpoint occurs at the exact moment that data residing at a particular address is read or written. Using data breakpoints, you can immediately locate the instruction responsible for overwriting a data structure. The 386 μ p lets you set breakpoint addresses in ROM as well as RAM. You can set a 386 debug exception when

any of the following conditions occur:

- an execution of the breakpoint instruction
- an execution of every instruction (single-stepping)
- an execution of any instruction at a given address
- a read or write of a byte, word, or doubleword at any specified address
- an attempt to change a debug register
- a task switch to a specific task (protected mode only)

Some Limitations

Even with all the above capabilities, the 386 debug registers can't always do the job of hardware-assisted debuggers. For example, while many in-circuit emulators (ICEs) can maintain breakpoints across processor resets, a 386 μ p reset clears the debug registers, effectively destroying any previously set breakpoints. Another limitation is that many ICEs can set breakpoints on I/O space accesses, but the 386 debug registers can't distinguish between memory and I/O space accesses. All data

breakpoints are associated with the memory space only. You need an ICE to trap I/O space accesses, or you can use an I/O permission bitmap in virtual 8086 mode.

You can use a hardware-assisted debugger such as an ICE to debug a debugger, but the only way to debug a 386-based debugger using the 386 debug registers is to ensure that the debugger uses only the INT 3 breakpoint. When triggered, the 386 debug registers generate an INT 1; therefore, a debugger relying only on INT 1 can debug a debugger relying only on INT 3. Further, because you need stable hardware to use the 386 debug registers, debugging in harsh, unknown system environments is easier with a full-scale ICE or other hardware-assisted debugger. System developers usually don't have the functional keyboard, display, disk, and operating system that is needed to load and run a debugger relying only on the 386 debug registers; they must rely on an ICE when debugging preliminary software and device drivers.

You do, however, need more than just the 386 debug registers to create an acceptable software debugger. Programmers expect today's debuggers to provide recognition of symbolics, fast disassemblies, and the ability to handle multiple object module formats (OMFs). Another desirable feature is a Boolean comparison layer above the debug exception handler.

Debuggers that take advantage of the 386 debug features are on the horizon. Combining the 386 debug features with what is available on today's debuggers will produce a powerful development tool. In the meantime, the example at the end of this article is part of a working program that exploits the 386 debug features to create a useful debugging tool.

Figure 4: The Seven Interrupt 1 Breakpoint Conditions

Status Register Flags	Condition
BS=1	Single-step trap
B0=1 AND (GE0=1 OR LE0=1)	Breakpoint DR0, LEN0, R/W0
B1=1 AND (GE1=1 OR LE1=1)	Breakpoint DR1, LEN1, R/W1
B2=1 AND (GE2=1 OR LE2=1)	Breakpoint DR2, LEN2, R/W2
B3=1 AND (GE3=1 OR LE3=1)	Breakpoint DR3, LEN3, R/W3
BD=1	Debug registers not available ICE-386 using debug registers (protected mode only)
BT=1	Task switch (protected mode only)

Built-in Features
The 386 μ p provides several built-in debugging features:

- **Four breakpoints.** You can specify four addresses that the CPU will automatically monitor.
- **Arm or disarm the breakpoints.** You can selectively enable and disable various debug conditions that are associated with the four debug addresses.
- **Data and instruction breakpoints.** You can set breakpoints on data accesses as well as on instruction executions.
- **Singlestepping.** You can step through the program one instruction at a time.

Let's look at how the 386 debug features are implemented. Along the way, we will develop some debugger software that will amplify these features.

Debug Registers

The 386 μ p has six registers to control debug features. **Figure 1** shows the format of the debug registers. You can access them by using variants of the MOV instruction. A debug register can be either the source or the destination operand. In protected mode, the MOV instructions that access the debug registers can only be executed at privilege level zero. Trying to read or write the debug registers from any other privilege level causes a general protection exception. Under a protected mode operating system (such as OS/2 or XENIX[®] systems), the debug registers are considered privileged resources, and only privileged tasks can access them.

Debug address registers (DR0-DR3). You can set a breakpoint address in each of the four debug address registers. Each register contains a linear address used to identify a break-

point. (In contrast, addresses are presented as a segment/offset pair in real mode.)

Debug control register (DR7). The debug control register lets you define, enable, and disable debug conditions. It specifies the conditions under which the 386 recognizes a breakpoint, the breakpoint type (local or global), and the breakpoint length field. The low order 8 bits of DR7 enable or disable the four breakpoints. Each breakpoint has 2 enable bits. The L bit enables local breakpoints, and the G bit enables global ones. The real mode debugging program shown later in this article sets and clears both bits.

For each address in register DR0 through DR3, the corresponding fields R/W0 through R/W3 (in register DR7) specify the type of action that can cause a breakpoint. The 386 interprets these bits as follows:

Bits	Meaning
00	break on instruction execution only
01	break on data writes only
10	undefined
11	break on data reads and writes but not on instruction fetches

The breakpoint length field (LEN_n) is associated primarily with data breakpoints. The

THE DEBUG CONTROL REGISTER LETS YOU DEFINE, ENABLE, AND DISABLE DEBUG CONDITIONS. IT SPECIFIES THE CONDITIONS UNDER WHICH THE 386 RECOGNIZES A BREAKPOINT, THE BREAKPOINT TYPE (LOCAL OR GLOBAL), AND THE BREAKPOINT LENGTH FIELD.

Figure 6: Debugging Assistant Common Code

```

NAME      ISR

EXTRN display_and_edit_regs:FAR

CODE SEGMENT PUBLIC 'CODE'
ASSUME CS:CODE

PUBLIC INT1_ISR, INT9_ISR, SYSREQ_ISR, ISR_common
PUBLIC orig_INT1_ISR_ptr, orig_INT3_ISR_ptr, orig_SYSREQ_ISR_ptr
PUBLIC orig_INT9_ISR_ptr

$EJECT
;-----
; This Assembly language module is composed of two
; interrupt service routines (ISRs) and another block of
; code shared between the two ISRs.
;
; The module is used to either awaken a register
; display/edit routine as a result of an 80386 debug
; exception or to awaken the display/edit routine
; as a result of a user request to "pop-it-up" (via the
; SYSREQ key).
;
; The first ISR, known as INT1_ISR, services the 80386 debug
; exception interrupt and then CALLs the common block of
; code, ISR_common.
;
; ISR_common copies the current 80386 register image into a
; large data structure and then CALLs the register
; display/edit routine (written in a higher-level language).
;
; The second ISR, known as SYSREQ_ISR, is chained into the
; BIOS INT 15h interrupt, and it merely awaits a SYSREQ key
; press (passing any other INT 15h requests to the original
; INT 15h handler) before calling ISR_common.
;
; The display/edit routine (not shown in this article, but
; written in PL/M-86) allows the user to examine and/or
; modify, by reading/altering the aforementioned register
; image data structure, the normal 80386 registers (in full
; 32-bit form), as well as the debug registers.
;-----
$EJECT
;-----
; Define structure/data area used to hold copies of the
; register images as they exist when the INT 1 ISR is
; entered.
;
; WARNING :
; The register ordering is FIXED!!! The
; display_and_edit_regs subroutine assumes a predefined
; ordering.
;
; ALSO :
; You probably noticed that each table entry is 32 bits
; long, even though some registers are actually only 16
; bits long (like CS). The uniform entry size makes
; indexing into the table much easier on the display
; routines. If a register is only 16 bits wide, the
; high order 16 bits of its register image from the
; following structure are wasted. This is not a problem
; since the program is not so large that such wastage is
; critical.
;-----
ISR_register_image LABELDWORD
ISR_DR0    DW 0 ; This register is 32 bits wide
           DW 0
ISR_DR1    DW 0 ; This register is 32 bits wide
           DW 0
ISR_DR2    DW 0 ; This register is 32 bits wide
           DW 0
ISR_DR3    DW 0 ; This register is 32 bits wide
           DW 0
ISR_DR6    DW 0 ; This register is 32 bits wide
           DW 0
ISR_DR7    DW 0 ; This register is 32 bits wide
           DW 0
ISR_CS     DW 0 ; This register is 16 bits wide
           DW ? ; (these 16 bits unused)
ISR_EIP    DW 0 ; This register is 32 bits wide
           DW 0
ISR_SS     DW 0 ; This register is 16 bits wide
           DW ? ; (these 16 bits unused)
ISR_ESP    DW 0 ; This register is 32 bits wide
           DW 0

```

CONTINUED

length of a data breakpoint can be a byte, a word, or a doubleword. Specifying only the starting address of a data item is not enough to match a breakpoint condition because data items can be of three different lengths (8, 16, and 32 bits). The length field adds flexibility by selecting a range of address accesses which can trigger a breakpoint. You can specify the following lengths:

Bits	Meaning
00	1-byte length
01	2-byte length (word)
10	undefined
11	4-byte length (doubleword)

Because instruction breakpoints should uniquely specify the byte-granular starting address of the intended instruction, instruction breakpoints always specify a length field of 1 byte. If *RWn* is 00—an instruction execution breakpoint—*LENn* should also be 00 (1 byte). Any other length is undefined.

When the LE (local) or GE (global) bit is set, the 386 μ p slows execution so data breakpoints can be reported on the precise instruction that caused them. Keep in mind that while an instruction execution breakpoint occurs *before* the specified instruction is executed, a data access (read or write) breakpoint occurs *after* the specified data is read or written.

Debug status register (DR6). The debugger uses the debug status register to determine what debug conditions have occurred. The 386 μ p sets status bits, and the debugger reads them. When the microprocessor detects a debug exception, it sets one or more of the status register's 4 low-order bits, B0 through B3, before entering the debug exception handler. Bn is set if the condition described in the address registers (*DRn*) and the control register (*LENn* and *R/Wn*) occurs.

Figure 6 CONTINUED

The BS bit, associated with the TF (trap flag) bit of the 386 EFLAGS register, is set if the debug handler is entered, since a single-step exception occurred. The single-step trap is the highest-priority debug exception. When BS is set, any of the other debug status bits may also have been set by the 386 μ p. This means that a single-step trap can occur at the same time an instruction or data breakpoint occurs. The BT and BS bits are used only when the microprocessor is in protected mode.

The BT bit is associated with the T bit (debug trap bit) of the task state segment, or TSS. The TSS is a data structure defined by the 386 system architecture; it is available only in protected mode. Each task has its own TSS, which holds the state of the task's virtual processor. The 386 μ p sets the BT bit before entering the debug handler if a task switch has occurred and if the T bit of the new TSS is set. There is no corresponding bit in DR7 that enables and disables this trap; the T bit of the TSS is the only enabling bit.

The ICE-386 is Intel's in-circuit emulator for the 386 μ p. When the ICE-386 is attached, it has priority over the 386 debugger. The BD bit is set if the next instruction will read or write one of the debug registers and ICE-386 is also using the debug registers.

The 386 μ p only clears the bits of DR6 when the microprocessor is reset. To avoid confusion in identifying the cause of the next debug exception, the debug handler should move zeros to DR6 immediately before returning.

Setting Breakpoints

Each of the four breakpoints is defined by its linear address (DR*n*) and its length (LEN*n*). The LEN field lets you specify a 1-, 2-, or 4-byte field. The 386

```

ISR_DS      DW      0      ; This register is 16 bits wide
            DW      ?      ; (these 16 bits unused)
ISR_ESI     DW      0      ; This register is 32 bits wide
ISR_ES      DW      0      ; This register is 16 bits wide
            DW      ?      ; (these 16 bits unused)
ISR_EDI     DW      0      ; This register is 32 bits wide
ISR_EAX     DW      0      ; This register is 32 bits wide
ISR_EBX     DW      0      ; This register is 32 bits wide
ISR_ECX     DW      0      ; This register is 32 bits wide
ISR_EDX     DW      0      ; This register is 32 bits wide
ISR_EBP     DW      0      ; This register is 32 bits wide
ISR_FS      DW      0      ; This register is 16 bits wide
            DW      ?      ; (these 16 bits unused)
ISR_GS      DW      0      ; This register is 16 bits wide
            DW      ?      ; (these 16 bits unused)
ISR_CRO     DW      0      ; This register is 32 bits wide
ISR_EFLAGS  DW      0
            DW      0

```

```

; The following space is allocated for the
; display_and_edit_regs routine but is not used by the
; ISRs.

```

```

dr0_segment DD      0
dr0_offset  DD      0
dr0_compare_value DD  0
dr0_compare_enable DD 0
dr1_segment DD      0
dr1_offset  DD      0
dr1_compare_value DD 0
dr1_compare_enable DD 0
dr2_segment DD      0
dr2_offset  DD      0
dr2_compare_value DD 0
dr2_compare_enable DD 0
dr3_segment DD      0
dr3_offset  DD      0
dr3_compare_value DD 0
dr3_compare_enable DD 0
dr0_boolean DD      0
dr1_boolean DD      0
dr2_boolean DD      0
dr3_boolean DD      0

```

```

$EJECT

```

```

; Allocate storage for INT 3 flag -- this flag is
; set/reset by the register display routine and
; indicates whether the ISR common code should trigger
; an INT3 shortly before RETing.

```

```

INT3_flag   DB      ?

```

```

; Allocate storage for the request flag -- we set this
; flag to indicate to the register display routine
; whether we're calling it from SYSREQ (pop-up request)
; or INT1 (debug exception has occurred).

```

```

request_flag DB      ?
INT1_request EQU     0
SYSREQ_request EQU    1

```

```

; Define the values necessary for processing SYSREQ key
; presses.

```

```

SYSREQ_key_pressed EQU     08500h
keyboard_status_port EQU    064h
input_buffer_full EQU     002h
enable_keyboard EQU       0AEh
PIC EQU              020h
EOI EQU              020h

```

CONTINUED

Figure 6 CONTINUED

```

;-----
; ISR local stack definition...
;-----
ISR_stack      DW      512 DUP      (0)
               LABEL WORD

;-----
; Where we store the original SS:SP before we install the
; local stack.
;-----
orig_ISR_stack_ptr  DW  ?
                  DW  ?

;-----
; Storage for copy of local code segment value.
;-----
local_data_seg     DW  CODE

;-----
; Define storage for the original interrupt service
; routine addresses for the interrupts onto which we'll
; be chaining or installing ourselves.
;-----
orig_INT1_ISR_ptr  DW  ?
                  DW  ?
orig_INT3_ISR_ptr  DW  ?
                  DW  ?
orig_INT9_ISR_ptr  DW  ?
                  DW  ?
orig_SYSREQ_ISR_ptr DW  ?
                  DW  ?

;-----
; The following instruction prefix is 80386-specific.
; It identifies the subsequent instruction as one that
; uses a 32 bit operand size. This prefix allows us to
; create 80386 instructions using an 80286 assembler.
;-----
operand_size_32_prefix EQU      066H

;-----
; Define stack frame that exists at the time the
; ISR_common is CALLED.
;-----
ISR_stack_parm     STRUC
ISR_RTNA           DW  ? ; Return address (to INT1_ISR
                    ; or SYSREQ_ISR)
old_IP             DW  ? ; CS:IP of code which was in
                    ; progress at time
old_CS             DW  ? ; the debug exception
                    ; occurred
old_FLAGS          DW  ? ; State of the FLAGS at time
                    ; of exception
ISR_stack_parm     ENDS

;-----
; Define flag that can be used to determine whether or
; not we're already servicing an interrupt request (in
; other words, are we being reentered ? )
;-----
ISR_in_progress    DB  FALSE
FALSE              EQU  0
TRUE               EQU  0FFh

$EJECT

;-----
; INT1 (80386 Debug Exception) Handler
;
; This interrupt service routine can be entered as a
; result of one of the following conditions:
;
; 1 - instruction execution breakpoint
; 2 - data access breakpoint

```

CONTINUED

up requires that 2-byte fields be aligned on word boundaries (addresses that are multiples of two), and 4-byte fields must be aligned on doubleword boundaries (addresses that are multiples of four). You will get unexpected results if code or data breakpoint addresses are not properly aligned.

You can set a data breakpoint for a misaligned field longer than 1 byte by using two or more debug address registers to hold the entire address. Each entry must be properly aligned, and the entries must span the length of the field. For example, when setting three breakpoints for a doubleword address that starts on an odd-byte boundary, the first address identifies the first byte of the doubleword, the second one identifies the next two bytes, and the third identifies the last byte. **Figure 2** shows you how to align breakpoint addresses for the address of a doubleword starting on an odd-byte boundary.

A memory access triggers a data read or write breakpoint when it occurs within a defined breakpoint field (as determined by a breakpoint address register and its corresponding LEN field). **Figure 3** contains examples showing when breakpoints will and will not occur.

Instruction breakpoint addresses are always specified as 1 byte (LEN=00). Other values for instruction breakpoint addresses are undefined. The 386 recognizes an instruction breakpoint only when the breakpoint address points to the first byte of an instruction. Therefore, if the instruction has prefixes, then the breakpoint address must point to the first prefix byte.

Debug Exceptions

Breakpoints set on instructions cause faults; all other debug conditions cause traps.

(Faults break before executing the instruction at the specified address. Traps report a data access breakpoint after executing the instruction that accesses the given memory item.) The debug exception can report faults and traps at the same time. The following describes the four classes of debug exceptions.

Instruction execution breakpoint. An instruction execution breakpoint is a fault, so the 386 μ p reports an instruction execution breakpoint before it executes the instruction at the given address.

Data access breakpoint. A data access breakpoint exception is a trap. That is, the processor reports a data access breakpoint after executing the instruction that accesses the given memory item.

When using data breakpoints, you should set the DR7's LE bit, GE bit, or both. If either LE or GE is set, any data breakpoint trap is reported exactly after completion of the instruction that accessed the specified memory item. This exact reporting is accomplished by forcing the 386 execution unit to wait for completion of data operand transfers before beginning execution of the next instruction. If neither GE nor LE is set, data breakpoints may not be reported until one instruction after the data is accessed or they may not be reported at all. This is because the 386 μ p normally overlaps instruction execution with memory transfers to such a degree that execution of the next instruction may begin before memory transfers for the previous instruction are completed.

If a debugger is creating a data write breakpoint, it should save the original data contents before setting the breakpoint. Because data breakpoints are traps, a write into a breakpoint location is completed before the trap condition is reported. The handler

Figure 6 CONTINUED

```

;
; 3 - general detect fault
; 4 - single step trap
; 5 - task switch breakpoint
;
; The ISR first ensures that it is not being reentered,
; and then CALLs ISR_common.
;-----
INT1_ISR PROC FAR
    JMP     INT1_start ; Jump around header field
    DB     'DAGGER'    ; This header is a "marker" used to
                    ; determine if the ISR is already
                    ; installed

INT1_start:
    PUSHF                    ; Save FLAGS
    CMP     CS:ISR_in_progress,TRUE ; Are we being
                    ; reentered?
    JNE     not_being_reentered ; NO
    POPF                     ; YES, recover FLAGS,
    STI                     ; put interrupts back
    IRET                    ; on, and leave

;-----
; Mark "in progress" flag so that we can't be reentered
;-----
not_being_reentered:
    MOV     CS:ISR_in_progress,TRUE ; Show "in progress"
    MOV     CS:request_flag,INT1_request ; Set flag i
                    ; indicating that
                    ; this INT occurred as a result of
                    ; an 80386 debug exception
    POPF                     ; Recover FLAGS

    CALL    ISR_common        ; CALL common ISR code
    IRET                    ; and leave

INT1_ISR ENDP

$EJECT

;-----
; SYSREQ (System Request Key) Handler
;
; This interrupt service routine can be entered
; as a result of one of the following conditions:
;
; SYSREQ key pressed OR some other BIOS INT 15
; request occurred
;
; We will only CALL ISR_common if the ISR was entered
; as a result of a userrequest to pop-up the register
; display/edit screen. Otherwise, we simply chain onto the
; old BIOS INT 15 ISR.
;-----
SYSREQ_ISR PROC FAR

    PUSHF                    ; Save FLAGS
    CMP     AX,SYSREQ_key_pressed ; Was the INT for
                    ; SYSREQ key ?
    JE     process_SYSREQ    ; SYSREQ key ?YES --
                    ; do local processing

chain_to_original_ISR:
    POPF                     ; NO, recover FLAGS
    STI                     ; Interrupts back on
    JMP     DWORD PTR CS:orig_SYSREQ_ISR_ptr ; Chain on to
                    ; original
                    ; SYSREQ ISR

process_SYSREQ:
    PUSH     AX                ; Save AX
    MOV     AL,EOI            ; Issue End-of-Interrupt to PIC
    OUT     PIC,AL

await_keybd_controller:
    IN     AL,keyboard_status_port ; Get keyboard
                    ; controller status
    TEST    AL,input_buffer_full ; Keyboard controller
                    ; ready to accept
                    ; command?
    JNZ     await_keybd_controller ; NO
    MOV     AL,enable_keyboard ; YES
    OUT     keyboard_status_port,AL ; Re-enable keyboard
    POP     AX                ; recover AX
    POPF                     ; recover FLAGS,

;-----
; IF we're being reentered, simply chain to

```

CONTINUED

Figure 6 CONTINUED

```

; original ISR.
;-----
PUSHF          ; Save flags
CMP    CS:ISR_in_progress,TRUE  ; Are we being
          ; re-entered?
JNE    SYSREQ_not_being_reentered ;NO
JMP    chain_to_original_ISR    ;Chain to original ISR

;-----
; ELSE: mark "in progress" flag so that we can't
; be reentered
;-----
SYSREQ_not_being_reentered:
MOV    CS:ISR_in_progress,TRUE  ;Show "in progress"
MOV    CS:request_flag,SYSREQ_request
          ;Set flag indicating that this
          ; INT occurred as a result of
          ; a user request to pop-up
          ; the display/edit routine
POPF          ;Recover FLAGS

CALL   ISR_common    ; CALL common ISR code,
JMP   DWORD PTR CS:orig_SYSREQ_ISR_ptr
          ; and then JMP to
          ; original INT 15h ISR

SYSREQ_ISR    ENDP

$EJECT

INT9_ISR     PROC    FAR

    JMP     DWORD PTR CS:orig_INT9_ISR_ptr

INT9_ISR     ENDP

$EJECT

ISR_common   PROC    NEAR

;-----
; Common interrupt service routine code (shared by
; INT1_ISR and SYSREQ_ISR)
;-----
; This common block of code simply copies the current
; register state into the large data structure described
; earlier. The address of the data structure is passed to
; the display/edit registers routine, which is a
; higher-level language subroutine that allows the user
; to edit the normal 80386 registers, as well as edit
; the debug registers.
;-----
STI          ;Interrupts back on
;-----
; Put EBP into data structure.
; Put EAX into data structure.
; Then get EFLAGS into data structure via EAX. FLAGS
; (low word of EFLAGS) were pushed onto the stack by the
; CPU when the INT occurred. We'll just OR the FLAGS
; (low 16 bits of EFLAGS) from the stack with the high
; word of current EFLAGS.
;-----
DB    operand_size_32_prefix
MOV   CS:ISR_EBP,BP    ; EBP into global structure

DB    operand_size_32_prefix
MOV   BP,SP           ; EBP = current ESP

DB    operand_size_32_prefix
MOV   CS:ISR_EAX,AX   ; EAX into global structure

DB    operand_size_32_prefix
PUSHF          ; PUSH EFLAGS
DB    operand_size_32_prefix
POP   AX         ; POP EAX (high word of EAX
          ; has high word of EFLAGS)
MOV   AX,[BP].old_FLAGS; Get FLAGS from stack into
          ; low word of EAX
DB    operand_size_32_prefix
MOV   CS:ISR_EFLAGS,AX; EFLAGS into global
          ; structure

$EJECT

;-----
; Get CS:IP from stack (placed there by 80386 when
; INT 1 occurred).
;-----

```

CONTINUED

can report the saved (original) value after the breakpoint has been triggered. The data in the debug registers can be used to address the new value written by the instruction that triggered the breakpoint.

Single-step trap. This debug condition occurs at the end of an instruction if the trap flag (TF) of the flag's register held the value 1 at the beginning of that instruction. Note that the exception does not occur at the end of an instruction that sets TF. For example, if POPF is used to set TF, a single-step trap does not occur until after the instruction that follows POPF.

The interrupt priorities in hardware guarantee that if an external interrupt occurs, single stepping stops. When an external and a single-step interrupt occur together, the single-step interrupt is processed first. This clears the TF bit. After saving the return address or the switch tasks, the external interrupt input is examined before the first instruction of the single-step handler executes. If the external interrupt is still pending, it is then serviced. The external interrupt handler is not single stepped. In order to single step an external interrupt handler, single step an INT n instruction that refers to the interrupt handler.

Task switch breakpoint. In protected mode, a breakpoint occurs after switching to a new task if the new TSS's T bit is set. The breakpoint occurs after control passes to the new task but before the first instruction is executed. The exception handler can detect a task switch by examining the BT bit of the debug status register (DR6).

Interrupts

Both the 386 and earlier microprocessors have two interrupt vectors dedicated to debugging. Interrupt 1 is reserved

for the single-step instruction trap, interrupt 3 for instruction breakpoints. In addition, the 386 μ p generates an interrupt 1 on any debug register trigger.

Interrupt 1. The handler for interrupt 1 is usually a debugger or part of a debugging system. **Figure 4** shows the seven breakpoint conditions that can cause an interrupt 1. The debugger can check flags in DR6 and DR7 to determine what condition caused the exception and what other conditions might have occurred.

Interrupt 3. This exception is caused by execution of the breakpoint instruction INT 3. Typically, a pre-386 debugger prepared a breakpoint by substituting the opcode of the 1-byte breakpoint instruction for the first opcode byte of the instruction to be trapped.

Prior to the 386 machine, microprocessors used the breakpoint exception extensively for trapping execution of specific instructions. The 386 μ p solves this need more conveniently by using the debug registers and interrupt 1. However, the breakpoint exception is still useful for debugging debuggers because the breakpoint exception can vector to an exception handler that is different from that used by the debugger. The breakpoint exception can also be useful when you need to set more breakpoints than allowed by the four debug registers.

Sample Debugger Program

You don't need to scrap your old-style debugger to take advantage of the 386's built-in debugging features. With a little help from a friendly debugging assistant (such as the working program fragment shown later in this article), you can continue to use your current debugger. By enabling or disabling data breakpoints with a pop-up rou-

Figure 6 CONTINUED

```

MOV     AX,[BP].old_CS      ; Get CS from stack
MOV     CS:ISR_CS,AX       ; CS into global structure
DB     operand_size_32_prefix
SUB     AX,AX              ; Clear high word of EAX
MOV     AX,[BP].old_IP     ; Get IP from stack into
                          ; low word of EAX
DB     operand_size_32_prefix
MOV     CS:ISR_EIP,AX      ; EIP into global structure

;-----
; Copy the rest of the registers into the data structure.
;-----
MOV     CS:ISR_DS,DS      ; DS
DB     operand_size_32_prefix
MOV     CS:ISR_ESI,SI     ; ESI
MOV     CS:ISR_ES,ES      ; ES
DB     operand_size_32_prefix
MOV     CS:ISR EDI,DI     ; EDI
DB     operand_size_32_prefix
MOV     CS:ISR_EBX,EBX    ; EBX
DB     operand_size_32_prefix
MOV     CS:ISR_ECX,CX     ; ECX
DB     operand_size_32_prefix
MOV     CS:ISR_EDX,DX     ; EDX
DB     08Ch,0E0h          ; FS
MOV     CS:ISR_FS,AX
DB     08Ch,0E8h          ; GS
MOV     CS:ISR_GS,AX
DB     00Fh,020h,0C0h     ; CR0
DB     operand_size_32_prefix
MOV     CS:ISR_CR0,AX
DB     0Fh,021h,0C0h     ; DR0
DB     operand_size_32_prefix
MOV     CS:ISR_DR0,AX
DB     0Fh,021h,0C8h     ; DR1
DB     operand_size_32_prefix
MOV     CS:ISR_DR1,AX
DB     0Fh,021h,0D0h     ; DR2
DB     operand_size_32_prefix
MOV     CS:ISR_DR2,AX
DB     0Fh,021h,0D8h     ; DR3
DB     operand_size_32_prefix
MOV     CS:ISR_DR3,AX
DB     0Fh,021h,0F0h     ; DR6
DB     operand_size_32_prefix
MOV     CS:ISR_DR6,AX
DB     0Fh,021h,0F8h     ; DR7
DB     operand_size_32_prefix
MOV     CS:ISR_DR7,AX

$EJECT

;-----
; Save original SS:SP into save area and also into the
; global structure and then create a new SS:SP so that
; we can CALL a stack-intensive P/LM-86 routine.
;-----
DB     operand_size_32_prefix
ADD     BP,SIZE_ISR_stack_parm
        ;Adjust EBP (which is a copy
        ; of original ESP) so that it
        ; reflects original state of
        ; ESP at the time INT occurred

DB     operand_size_32_prefix
MOV     CS:ISR_ESP,BP     ;ESP into global structure
MOV     CS:ISR_SS,SS     ;SS into global structure

MOV     CS:orig_ISR_stack_ptr,SP
        ;Save SP into local storage
MOV     CS:orig_ISR_stack_ptr + 2,SS
        ;Save SS into local storage

```

CONTINUED

Figure 6 CONTINUED

```

CLI ;Clear INTs while working on stack regs
MOV SS,CS:local_data_seg ;New SS
LEA SP,ISR_stack ;New SP
STI ;Restore INTs

;-----
; CALL the PL/M-86 procedure responsible for displaying
; and processing the information we've just put into the
; data structure. The PL/M routine will read and display
; the register state (as shown in the data structure),
; and will allow the user to indirectly modify the
; registers (including the debug registers) via edits
; to that same structure. When the display/edit routine
; RETURNS, we'll copy the register image back into the
; 80386 registers.
;-----
LEA AX,ISR_register_image ; Pass the address of
PUSH CS ; the register image
PUSH AX ; as pointer on stack
LEA AX,INT3_flag ; Pass the addr of INT3 flag
PUSH CS ; so that display/edit routine
PUSH AX ; can set/reset as user desires
MOV AL,CS:request_flag ; Pass request type flag so
PUSH AX ; that display/edit routine
; can determine whether it was
; called as result of an INT 1
; or SYSREQ
CALL display_and_edit_regs ;CALL P/IM-86 PROCEDURE

$EJECT

;-----
; Transfer the edited register images from the global
; structure back into the 80386 registers.
;-----
DB operand_size_32_prefix
MOV BX,CS:ISR_EBX ; EBX

DB operand_size_32_prefix
MOV CX,CS:ISR_ECX ; ECX

DB operand_size_32_prefix
MOV DX,CS:ISR_EDX ; EDX

DB operand_size_32_prefix
MOV SI,CS:ISR_ESI ; ESI

DB operand_size_32_prefix
MOV DI,CS:ISR_EDI ; EDI

DB operand_size_32_prefix
MOV BP,CS:ISR_EBP ; EBP

MOV ES,CS:ISR_ES ; ES
MOV DS,CS:ISR_DS ; DS

MOV AX,CS:ISR_FS
DB 08Eh,0E0h ; MOV FS,AX

MOV AX,CS:ISR_GS
DB 08Eh,0E8h ; MOV GS,AX

DB operand_size_32_prefix
MOV AX,CS:ISR_CR0
DB 00Fh,022h,0C0h ; MOV CR0,EAX

DB operand_size_32_prefix
MOV AX,CS:ISR_DR0
DB 00Fh,023h,0C0h ; MOV DR0,EAX

DB operand_size_32_prefix
MOV AX,CS:ISR_DR1
DB 00Fh,023h,0C8h ; MOV DR1,EAX

DB operand_size_32_prefix
MOV AX,CS:ISR_DR2
DB 00Fh,023h,0D0h ; MOV DR2,EAX

DB operand_size_32_prefix
MOV AX,CS:ISR_DR3
DB 00Fh,023h,0D8h ; MOV DR3,EAX

;-----
; Clear out DR6 -- all bits in that reg must be reset
; after each INT 1 (ignore whatever is currently sitting
; in the DR6 register image).
;-----

```

CONTINUED

time, the debugging assistant shown supplements the setting of instruction breakpoints by old-style debuggers. You still have all the benefits of your older debugger (symbolics and disassemblies, for example) plus the powerful built-in debugging capabilities of the 386 μ p.

The 386-based debugging assistant runs in the background as an adjunct to a traditional, pre-386 debugger. The debugging assistant is a terminate-and-stay-resident (TSR) program that pops up either when you press the SYSREQ key or when a debug exception occurs. The pop-up screen describes the 386 registers and identifies which of the four individually programmable breakpoints occurred.

Our debugging assistant has several features built into it. A human interface displays all the registers you're interested in and lets you easily enter breakpoint addresses, enable/disable breakpoints, and define breakpoint conditions. Figure 5 shows a model of this screen, which is divided into the following five option fields:

- BREAKPOINT contains the four 386 debug registers and their options.
- COMPARE shows the Boolean comparison options that are available.
- SPECIAL REGS contains the 386 EFLAGS register in its hexadecimal representation, the debug control register (DR7), and the debug status register (DR6).
- REGISTER SET displays the values in the most frequently referenced 386 registers.
- EFLAGS presents the EFLAGS register decoded into an English format.

You can change most of the values displayed on the screen

by either editing or toggling each respective field.

A Boolean comparison layer decides if the exception meets the criteria you specified. The debugging assistant has Boolean comparison logic that is armed and disarmed separately by toggling the switch (sw) field in COMPARE. You can also toggle the Boolean (bool) field in COMPARE between:

- < less than
- <= less than or equal to
- = equal to
- <> not equal to
- > greater than
- >= greater than or equal to

The value to be used in the comparison logic is specified by editing the value field in COMPARE.

An interrupt service routine handles exceptions. The debugging assistant program has two interrupt service routines (ISRs). One ISR handles the SYSREQ key and the other handles debug exception interrupts. Part of the code in Figure 6 contains an interrupt service routine.

Common code is called by the interrupt service routines. In the debugging assistant program, both ISRs call a common block of code that copies the 386 registers into a large data structure. The common code then passes the address of the data structure to the high-level language program that controls the human interface.

After the registers are modified (through the human interface), the high-level language program returns control to the common code. The common code then copies the edited register images back into the 386 registers and "goes to sleep" until the next debug exception or until the SYSREQ key is pressed. Figure 6 contains the common code, written in assembly language.

Figure 6 CONTINUED

```

DB      operand_size_32_prefix
SUB     AX,AX                ; SUB EAX,EAX
DB      00Fh,023h,0F0h      ; MOV DR6,EAX

DB      operand_size_32_prefix
MOV     AX,CS:ISR_DR7       ; MOV EAX,CS:ISR_DR7
DB      00Fh,023h,0F8h      ; MOV DR7,EAX

DB      operand_size_32_prefix
MOV     AX,CS:ISR_EFLAGS    ; MOV EAX,CS:ISR_EFLAGS
DB      operand_size_32_prefix
PUSH    AX                  ; PUSH EAX
DB      operand_size_32_prefix
POPF                        ; POP EFLAGS

DB      operand_size_32_prefix
MOV     AX,CS:ISR_EAX       ; MOV EAX,CS:ISR_EAX

$EJECT
;-----
; Clean up stack.
;-----
CLI     ; Clear INTs while working on stack
MOV     SS,CS:orig_ISR_stack_ptr + 2 ; Get original SS
MOV     SP,CS:orig_ISR_stack_ptr     ; Get original SP
MOV     CS:ISR_in_progress,FALSE     ; Show "no longer
; in progress"

;-----
; Issue an INT3 (old-style debugger interrupt) if user
; so directed. In that fashion, we can trigger a "real"
; debugger (presumably one that will allow us to
; examine/modify memory and display symbol information).
; We can merely get rid of the local caller's return
; address (SYSREQ_ISR or INT1_ISR) and then "JMP"
; directly to the original INT3_ISR. Since our code was
; entered as a result of an INT1, the stack will
; already be set up such that the INT3 routine has only
; to execute an IRET.
;-----
PUSHF   ; Did user want an INT3?
CMP     CS:INT3_flag,0          ; NO
JZ      ISR_common_RET         ; YES, recover flags,
POPF    ; Recover flags,
ADD     SP,2                   ; pop-off the local
; caller's return address
PUSH    BP                    ; Save reg
MOV     BP,SP                 ; Get stack ptr
INC     WORD PTR [BP + 2]      ; Adjust IP on stack such
; that the INT3 handler
; can DEC to adjust for
; the "INT3"

POP     BP                    ; Recover reg
STI     ; Put interrupts back on,
JMP     DWORD PTR CS:orig_INT3_ISR_ptr
; and then "JMP" to the
; original INT3_ISR

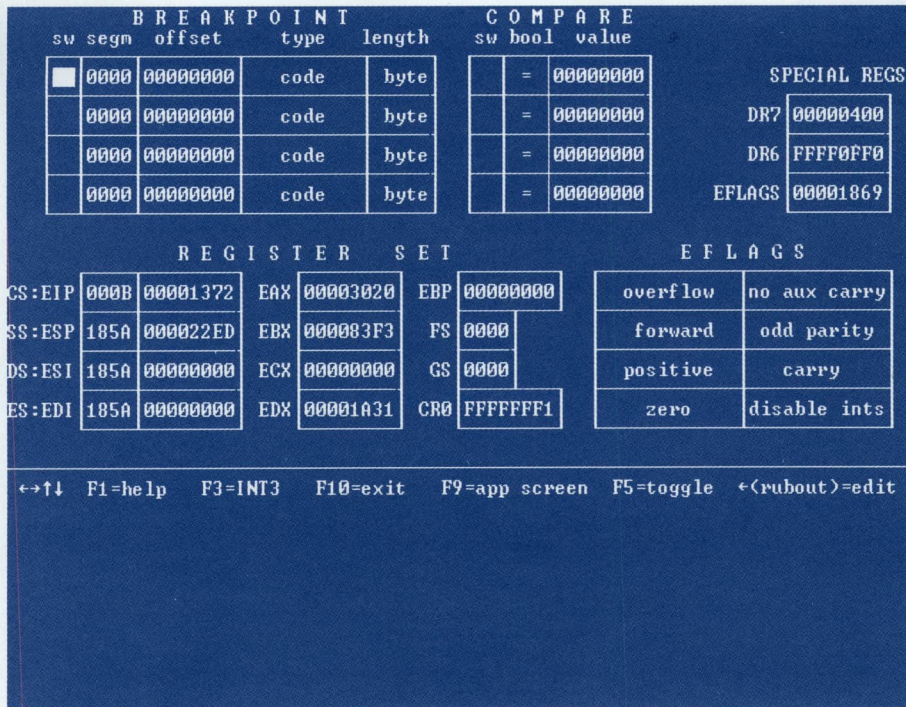
ISR_common_RET:
POPF    ; Recover flags
STI     ; Interrupts back on
RET

ISR_common ENDP

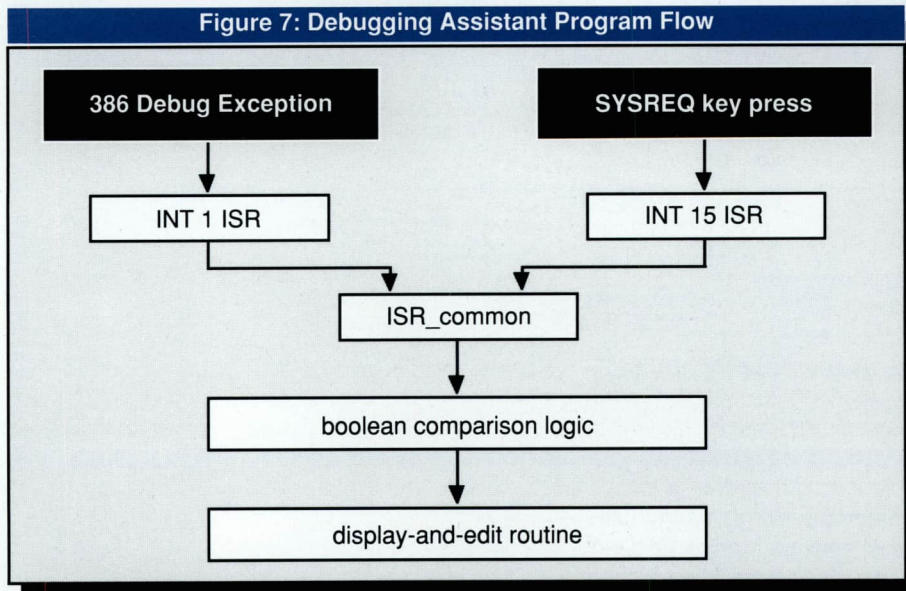
CODE    ENDS
END

```

Some of the assembly language mnemonics may be unfamiliar. Because of the absence of 386 assemblers, we hand-assembled certain 386-specific instructions. Most of them were relatively easy to code because they only needed a prefix byte. The prefix byte specifies that the following instruction will use 32-bit operands, rather than 16-



▲ **Figure 5** The debugging assistant screen displays the 386 microprocessor registers, with the four debug registers gathered in the top left quadrant of the screen, and allows you to set breakpoint addresses and define breakpoint conditions.



bit pre-386 instructions. For example, inserting the prefix byte 066h ahead of

```
mov ax, bx
```

forms an instruction which the 386 μ p interprets as:

```
mov eax, ebx
```

Figure 7 shows the flow of the debugging assistant program.

A thorough understanding of the 386 debug support features presented in this article will provide a good starting point for tackling difficult debugging chores. It makes it possible to build special debugging utilities that focus on specific problems that might not be handled by a generic debugger. We are not, of course, suggesting that the reader should actually build a debugger. But under certain circumstances a debugging assistant such as the one presented here will decrease debugging time plus give you the satisfaction of both fully exploiting the debugging capabilities and better understanding the internal workings of the most advanced microprocessor on the market. □

Strategies for Building and Using OS/2 Run-Time Dynamic-Link Libraries

Ross M. Greenberg

```

You are in an OS/2 maze of twisty little passages, all alike.
There is a dwarf here.
There is an ax here.

> TAKE AX
Ax taken. The dwarf growls and guards his cache of 256Kb chips.
> THROW AX
You have no ax. The dwarf smirks menacingly at your empty
memory expansion card.
> INVENTORY
You are carrying an ax and a memory expansion card.
> THROW AX
You have no ax. The dwarf chuckles and takes out a static
discharge gun.
> LOAD DEBUG DLL
  (loading new DLL ... DLL loaded)
The dwarf leers at you and takes aim.
> THROW AX
DBG:inv_cnt = 2
DBG:inv_tab[0] = 'ax' / strcmp != 0
DBG:inv_tab[1] = 'memory expansion card' / strcmp != 0
DBG:End of List - Failed
You have no ax. The dwarf times it well and fires while you are
caught thrashing as you swap to disk. _

```

Y

our process has died.

Later, when you run the BACKTRACE function in your debug DLL, you find that you TOOK the ax with a strcmp but tried to THROW with a strcmp. And with a quick correction the bug is history.

A Debug DLL

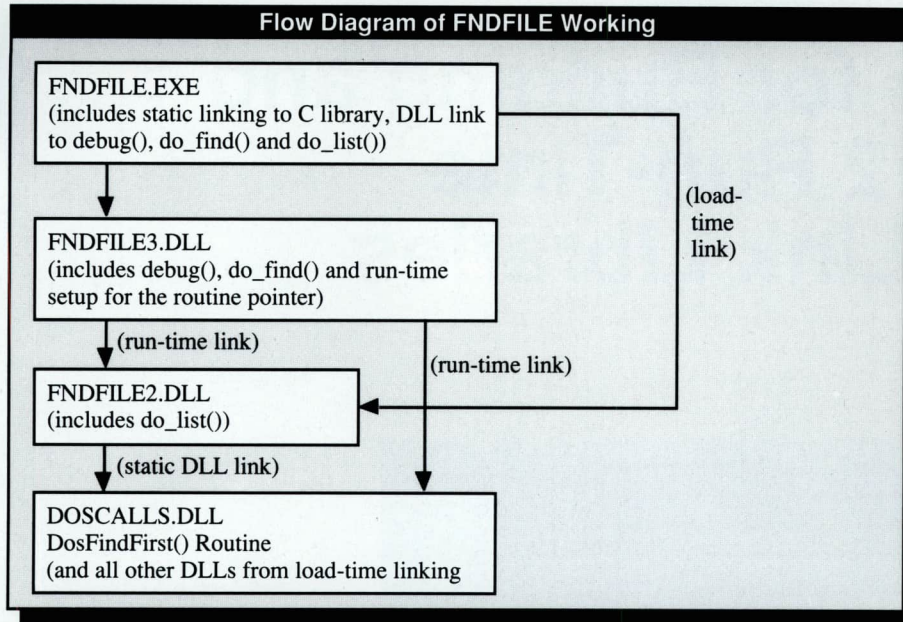
What is a debug DLL anyway, and how can you create one?

The Dynamic-Link Library (DLL) capabilities of OS/2 systems actually come in two "flavors." The first, called static linking, was described fully in "Design Concepts and Considerations in Building an OS/2 Dynamic-Link Library," *MSJ* (Vol. 3, No. 3). Briefly stated, the static linking implementation of DLLs lets an EXE function be called into memory from another file, a DLL.

When the EXE program loads, the necessary functions are loaded into memory from the DLL, linkage references to the functions are resolved, and the EXE starts to run. Only functions referenced at link time are loaded into memory, and the operating system can discard and reload them as needed. Importantly, only one copy of any given function needs to be in memory and it is then shared by all sessions requiring it.

There are various advantages to using the Dynamic-Link capabilities of OS/2 systems when developing code: smaller run times; the ability to truly share a single copy of your developed routines between programs (when properly configured, only a single copy of your routines will get loaded into memory regardless of how many processes

Ross M. Greenberg is a computer consultant and software author in NYC, specializing in communications. His most recent product is RamNet, a background communications program, BBS and E-mail program.



THERE ARE VARIOUS
ADVANTAGES TO USING
THE DYNAMIC-LINK
CAPABILITIES OF OS/2 WHEN
DEVELOPING CODE:
SMALLER RUN TIMES;
THE ABILITY TO TRULY
SHARE A SINGLE COPY OF
YOUR DEVELOPED
ROUTINES BETWEEN
PROGRAMS; THE ABILITY TO
DISTRIBUTE DIFFERENT
VERSIONS OF YOUR
PROGRAM PACKAGED WITH
DIFFERENT DLLS TO SERVE
DIFFERENT MARKETS;
AND SO ON.

use them); the ability to distribute different versions of your program packaged with different DLLs to serve different markets; and so on.

DLLs were such a good concept that a majority of OS/2 is, in reality, a collection of DLLs. Different OEMs, when porting OS/2 over to their own unique hardware environment, need only replace individual DLLs as they continue with their port effort.

Each software author will most likely find a different aspect of the DLL concept that fits their needs; DLLs are that flexible. Although a little difficult to use initially, I've found that most of my OS/2 code uses DLLs more and more—and, more and more, it uses routines already written as I create my own libraries, with a resolution granularity that's definable on a routine-by-routine basis.

However, loading a DLL like this doesn't let you load functions as you might wish—and all functions you reference in the link stage are loaded into memory. If you happen to have a very large, infrequently used function, it will still be read from disk, loaded into memory, and

later (perhaps) discarded by the operating system if there's a memory crunch.

Dynamic Linking

OS/2 provides a second flavor to its Dynamic-Link Library Package: the ability to have "run-time linking." In essence, this lets you load functions (called procedures in DLL parlance) as needed. And when necessary, you can (in essence) discard them just as easily. You can even mix the two different types of DLLs, so that the operating system takes care of the "usual" cases and you take care of the unusual ones.

Necessary Functions

There are only five new function calls you need to learn to take advantage of this feature. But there are several strategies you will need to consider as you create and build your DLLs. The five new functions are:

- DosLoadModule
- DosGetProcAddress
- DosFreeModule
- DosGetModHandle
- DosGetModName

In discussing these functions I'm going to begin using the C notation provided in the most recent set of manuals from the Microsoft® Software Development Kit (SDK); assembly language programmers, be forewarned. The new definitions and typedefs allow for more machine independent code, although they might be a bit difficult to understand at first. A bit of sound advice here: it is essential that the very first thing you do is to make a complete hard-copy list of all the OS/2 header files. Study them now and you will save yourself lots of digging for the information later. Future versions of OS/2 systems will most likely allow the use of different addressing schemes, so machine independen-

dence is actually a valuable concept, even at this stage.

To begin with we have DosLoadModule, defined (prototyped) in BSEDOS.H:

```
USHORT  DosLoadModule(
        bomb_ptr,
        bomb_ptr_len,
        mod_name_ptr,
        mod_handle_ptr)
PSZ     bomb_ptr;
USHORT  bomb_ptr_len;
PSZ     mod_name_ptr;
PMODULE mod_handle_ptr;
```

The bomb_ptr field is a character array, of length bomb_ptr_len. The operating system will fill in this array with the null-terminated name of the DLL file that caused the error in loading. Normally this null-terminated string would be equivalent to the actual module name, mod_name_ptr. But since a DLL can call other DLLs, there is a possibility that some DLL further down the "tree" might have caused the problem.

If the function returns a zero, it loaded successfully. If the module was already loaded by some other process, the reference count on the module is incremented. The module will remain loaded as long as the reference count is greater than zero. Only segments of the DLL marked as "preload" in the DLL's DEF file will be automatically loaded.

The module name itself must be the name of a DLL in your LIBPATH or the function will fail. The module handle must be preserved for all other calls of the run-time load package; think of it as a file handle if you wish. Once the module can be referenced by a handle, routines within the DLL can now be accessed—if their address is known.

```
USHORT  DosGetProcAddress(
        mod_handle,
        function_name,
        function_ptr)
HMODULE mod_handle;
PSZ     function_name;
PPFN    function_ptr;
```

Figure 1: Source Code for FNDFILE.C

[FNDFILE.C]

```
#define INCL_DOS
#define INCL_ERRORS

#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

USHORT far _loadds pascal debug(USHORT);
USHORT far _loadds pascal do_find(PSZ, PHDIR, USHORT,
    PFILEFINDBUF, USHORT, PUSHORT, ULONG);
USHORT far _loadds pascal do_list(PSZ, PHDIR, USHORT,
    PFILEFINDBUF, USHORT, PUSHORT, ULONG);

void do_usage(void);
void main(void);

void show(PFILEFINDBUF f_ptr, USHORT cnt)
{
    while (cnt--)
    {
        printf("Len is: %2d. Filename is:%s\n",
            (int)(f_ptr->cchName), f_ptr->achName );
        f_ptr = (PFILEFINDBUF) ((char far *)f_ptr + 24 +
            (int)(f_ptr->cchName));
    }
}

void do_usage()
{
    printf("\n\nexit - to exit\n");
    printf("file=<filename> - enter name of message file\n");
    printf("hand=<val> - 1 = default, ffff = create
new\handle\n");
    printf("attrb=<attrb> - Attributes: see pg 98 of Ref Manual\n");
    printf("len=<buf_len> - length of output buffer to\
use.Allocated\n");
    printf("cnt=<cnt> - Maximum number of files to return\n");
    printf("debug=on|off - turn debug mode on or off\n");
    printf("show - show buffer contents...\n");
    printf("list - show current parameter settings\n");
    printf("go - call DosFindFirst()\n");
}

void main()
{
    CHAR tmp_buf[256];
    CHAR name[64];
    USHORT max_len = 0;
    char *buf_ptr = (char *)NULL;
    HDIR handle = 1;
    USHORT attrb = 0;
    USHORT num_files = 0;

    if(!debug(FALSE))
    {
        printf("Error in initial call to debug\n");
        exit(1);
    }
    *name = (CHAR)NULL;
    while(TRUE)
    {
        printf(">");
        gets(tmp_buf);
        strlwr(tmp_buf);
        if(!strcmp(tmp_buf, "exit", 4))
            exit(1);
        else
            if(!strcmp(tmp_buf, "file=", 5))
```

CONTINUED

Figure 1 CONTINUED

```

        strcpy(name, tmp_buf + 5);
    else
    if(!strncmp(tmp_buf, "len=", 4))
    {
        if(buf_ptr)
            free(buf_ptr);
        max_len = atoi(tmp_buf + 4);
        buf_ptr = calloc(max_len, 1);
    }
    else
    if(!strncmp(tmp_buf, "cnt=", 4))
        num_files = atoi(tmp_buf + 4);
    else
    if(!strncmp(tmp_buf, "hand=", 5))
        sscanf(tmp_buf + 5, "%x", &handle);
    else
    if(!strncmp(tmp_buf, "attrb=", 6))
        sscanf(tmp_buf + 6, "%x", &attrb);
    else
    if(!strncmp(tmp_buf, "list", 4))
        do_list(name, (PHDIR)&handle, attrb,
            (PFILEFINDBUF)buf_ptr, max_len,
            (PUSHORT)&num_files, (ULONG)0);
    else
    if(!strncmp(tmp_buf, "debug=", 6))
    {
        if(!debug(!strncmp(tmp_buf + 6, "on", 2)))
        {
            printf("Error in subsequent call to debug\n");
            exit(1);
        }
    }
    else
    if(!strncmp(tmp_buf, "show", 4))
        show((PFILEFINDBUF)buf_ptr, num_files);
    else
    if(!strncmp(tmp_buf, "go", 2))
        do_find(name, (PHDIR)&handle, attrb,
            (PFILEFINDBUF)buf_ptr, max_len,
            (PUSHORT)&num_files, (ULONG)0);
    else
    if(*tmp_buf == '?')
        do_usage();
    else
        printf("?Huh?\n");
}
}

```

[FNDFILE.DEF]

```

IMPORTS FNDFILE3.do_find
IMPORTS FNDFILE3.debug
IMPORTS FNDFILE2.do_list

```

This function will return the address in the DLL (specified in `mod_handle`) of the named function into the function pointer, `function_addr`. Once returned, it may be called via dereferencing as with any other function pointer. Functions with the same name can be referenced in different DLLs since the handle to the DLL itself would be different.

The name of the function, however, can also be an ordinal number. Each function within a DLL can have a public name

associated with it. For efficiency, you can remove the function names from the DLLs and only allow reference through the ordinal number. As an example, `DOSCALLS.DLL` does not have the function names readily available, and functions contained therein must be done by ordinal number.

An ordinal number is of the form "#xyz"—it must start with a pound sign, and the remaining part of the string (that is, `xyz`) is the ASCII representation of the ordinal number of the function

you wish to reference.

OS/2 does not provide a means of discarding a function you no longer have any use for. But the operating system will discard the function if it needs the memory the function occupies. If you no longer reference the function, then it will not be reloaded into memory.

On the other hand, when you no longer need a DLL module, you may discard it.

```

USHORT    DosFreeModule(
           mod_handle)
HMODULE   mod_handle;

```

This will decrement the reference count of the module itself. If the reference count returns to zero, then all memory allocated to the module will be deallocated and discarded as required. Even if the reference count of the module is not zero, however, the local process's references to the DLL are no longer valid; calls to function addresses returned by `DosGetProcAddress` will fail with a protection violation.

When a program exits, it returns all of its resources to the operating system, so it is not necessary to call `DosFreeModule` when you exit your program—although it is good programming practice.

Two complementary functions exist, for determining whether the process has already loaded a module and for determining the name of a module based on its handle.

```

USHORT    DosGetModHandle(
           mod_ptr,
           mod_handle_ptr)
PSZ       mod_ptr;
PHMODULE  mod_handle_ptr;

```

This function will return the handle associated with a named DLL module. It's useful if you want to double-check whether a given module is already loaded. Since the handle itself is local to a given process, if you need to use this call, you should double-check your error processing—how could you forget a handle?

There are some specific reasons for using this function in a large program. Potentially, some other member of your programming team could have released the module, and then reloaded it later. This would usually cause it to get a different handle, which might cause you a problem if the handle is in a local, static variable. You can find a way around that problem in the debug routine in the sample program.

DosGetModName, which is the best function, is defined as follows:

```
USHORT DosGetModName (
    mod_handle,
    name_len,
    name_ptr)
HMODULE mod_handle;
USHORT name_len;
PCHAR name_ptr;
```

This function simply returns the DLL name associated with a given handle into the character array of name_ptr. If the name of the DLL can't fit into the character array, then an error condition is returned. This is useful in determining the name of a currently loaded DLL. Again, take a look at the debug function in the sample program for an idea of how to use it.

DLLs Call DLLs

One of the more interesting aspects of DLLs has largely been ignored until now; the ability of a DLL to utilize another DLL almost infinitely. I use this technique as an efficient way of dealing with two distinct DLLs utilized via the run-time load mechanisms.

There is a decent enough reason for it. You needn't change any code in an applications program to take advantage of things like extended commands and functions; it becomes transparent where the actual DLL "lives," and the total separation of the two DLLs—through a third, common DLL—allows easy modification of one

Figure 2: Source Code for FNDFILE2.C

```
[FNDFILE2.C]
#define INCL_BASE
#define INCL_ERRORS
#define INCL_DOS

#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void show(PFILEFINDBUF f_ptr, USHORT cnt);

USHORT far _loadds pascal do_list(PSZ name_ptr, PHDIR
    hptr, USHORT attrb,
    PFILEFINDBUF
    buf_ptr,
    USHORT buf_len,
    PUSHORT num_ptr,
    ULONG reserved)
{
    USHORT stat;
    HDIR save_handle = *hptr;
    USHORT save_num = *num_ptr;

    printf("File name: %s\n", name_ptr);
    printf("Buffer address is %lx\n", buf_ptr);
    printf("Buffer length: %d\n", buf_len);
    printf("File count: %d, address: %ld\n",
        *num_ptr, num_ptr);
    printf("Handle is: %d\n", *hptr);
    printf("Attribute: %d\n", attrb);

    stat = DosFindFirst(name_ptr, hptr, attrb, buf_ptr,
        buf_len, num_ptr, reserved);
    printf("Return status was:%d, cnt was:%d\n", stat,
        *num_ptr);
    *hptr = save_handle;
    *num_ptr = save_num;
    return(stat);
}
```

```
[FNDFILE2.DEF]
LIBRARY FNDFILE2
EXPORTS do_list
```

without having to recompile or maintain the other.

FNDFILE

The simple application I use to demonstrate the run-time loading capabilities of OS/2, FNDFILE (see Figures 1, 2, and 3), uses the OS/2 function call, DosFindFirst. This function lets you retrieve structures containing information about files with names matching a given pattern. You can retrieve information on as many files matching the pattern as you like, so long as you provide a buffer large enough to hold the returned information.

One interesting little quirk in DosFindFirst is that—although it is documented to fill a buffer

AN INTERESTING CONCEPT USED IN OS/2 IS THAT OF A "DIRECTORY-SEARCH HANDLE." THIS LETS YOU CONSIDER THE DOSFINDFIRST FUNCTION A RESOURCE WHICH CAN BE ALLOCATED AND DEALLOCATED LIKE ANY OTHER RESOURCE.

Figure 3: Source Code for FNDFILE3.C

```
[FNDFILE3.C]
#define INCL_BASE
#define INCL_ERRORS
#define INCL_DOS

#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define FAIL_BUF_LEN 128
PSZ mod_name[] = {"doscalls", "fndfile2"};
PSZ routine_name[] = {"#64", "DO_LIST"};

USHORT (pascal far *routine)(PSZ, PHDIR, USHORT, PFILEFINDBUF,
USHORT, PUSHORT, ULONG);

USHORT far _loads pascal do_find(PSZ name_ptr, PHDIR
                                hptr, USHORT attrb,
                                PFILEFINDBUF buf_ptr,
                                USHORT buf_len,
                                PUSHORT num_ptr,
                                ULONG reserved)
{
    USHORT stat;

    if(!(stat = routine(name_ptr, hptr, attrb, buf_ptr,
                        buf_len, num_ptr, reserved)))
    {
        printf("Good return. Files found = %d\n", *num_ptr);
    }
    else
    {
        switch(stat)
        {
            case ERROR_BUFFER_OVERFLOW:
                printf("Buffer Overflow - Increase Buffer
Size\n");
                break;

            case ERROR_DRIVE_LOCKED:
                printf("Drive Locked\n");
                break;

            case ERROR_FILE_NOT_FOUND:
                printf("File: %s not found\n", name_ptr);
                break;

            case ERROR_INVALID_HANDLE:
                printf("Invalid handle: %d\n", *hptr);
                break;

            case ERROR_INVALID_PARAMETER:
                printf("Invalid Parameter\n");
                break;

            case ERROR_NO_MORE_FILES:
                printf("Ran out of files\n");
                break;

            case ERROR_NO_MORE_SEARCH_HANDLES:
                printf("Can't allocate another Search Handle\n");
                break;

            case ERROR_NOT_DOS_DISK:
                printf("Not a DOS Disk\n");
                break;

            case ERROR_PATH_NOT_FOUND:
                printf("I can't locate that Path\n");
                break;

            default:
                printf("Unknown error in FindFirst: %d\n", stat);
                break;
        }
    }
}

```

CONTINUED

with the contents of a structure, called FILEFINDBUF—it cheats a little. Basically the structure is a bunch of dates and times, a couple of longs (to indicate filesize), a character count, and an array of 13 characters, which holds the filename. The character array contains only enough bytes to satisfy the need for a null-terminated string. The next structure starts immediately after the null byte, so a simple pointer-to-structure increment doesn't work, since that increments the pointer by exactly one sizeof(FILEFINDBUF). Therefore, when examining the code in the show routine, please forgive the ugliness of the increment to the pointer.

An interesting concept used in OS/2 is that of a "directory-search handle." This lets you consider the DosFindFirst function a resource which can be allocated and deallocated like any other resource. For example you could set up a number of initial requests to find a different matching filename pattern (say, the same pattern in different directories) and to then call functions, passing the directory-search handle as a parameter, which in turn call DosFindNext using those handles to find the next file on a pattern-by-pattern basis. Calling DosFindFirst with a directory-search handle already in use causes closing and then reuse of that handle.

Other parameters in the DosFindFirst function let you include matching "special" files in your request, including directories, hidden files, read-only files, etc. By examining the returned attribute for each of the files, you can determine what type of special file you're currently examining.

The DosFindFirst function is useful enough that most of us will have some need of it. Yet it's also sufficiently complex that we'll all make mistakes

when we first use it. Additionally it makes a perfect test case for a debug DLL.

The structure of the FNDFILE program is a bit convoluted, so some explanation is in order. The main routine lets you enter the parameters of your choice via a simple keyword parsing if-then-else clause. If you enter the "GO" command, you call the do_find function. Other "action" keywords let you turn debugging on or off, show your current parameter list, and show the current return buffer contents.

The do_find function is called with parameters passed exactly as DosFindFirst expects them; it makes it easier to do the call later. When the find program is first invoked, the debug function is called with the flag turned off, and debug itself (in FNDFILE3) loads the normal DosFindFirst function from the appropriate DLL, DOSCALLS.

You cannot, however, just load the DosFindFirst function as you'd wish. The function name isn't immediately accessible from the DOSCALLS library. Since the name is "private," I was forced to do a hex dump of the OS2.LIB file (or the DOSCALLS.LIB file, depending on your version of OS/2) and to find the so-called "ordinal number" of the DosFindFirst function.

An ordinal number is basically a simple numeric representation of the function number itself in the library, and is a more efficient way of calling important functions than forcing the kernel to do a strcmp on a function name.

The hex dump (see Figure 4) shows 64 as the ordinal number for DosFindFirst. To indicate that you're using an ordinal number, you must specify the number as an ASCII null-terminated string, with a '#' prefixed to it as the function name in the DosLoadProcAddr function.

Figure 3 CONTINUED

```

    }
    }
    return(stat);
}

far _loads pascal debug(USHORT debug_flag)
{
    USHORT    stat;
    CHAR      fail_buf[FAIL_BUF_LEN];
    static    HMODULE    handle = 0;
    HMODULE   tmp_handle;
    CHAR      tmp_buf[128];

    printf("Debug is: %s\n", debug_flag ? "On" : "Off");

    /* already a DLL loaded? */
    if(handle)
    {
        /* some DLL already loaded. Requested DLL? */
        stat = DosGetModHandle(mod_name[debug_flag],
                               &tmp_handle);

        /* if error, or a handle mismatch, then it isn't
         * the requested DLL */
        if(stat || tmp_handle != handle)
        {
            /* Get name of the DLL currently loaded */
            if(stat = DosGetModName(handle, 128,
                                    tmp_buf))
            {
                printf("Couldn't retrieve loaded DLL Name. Error
                    code is: %d\n", stat);
                return(FALSE);
            }
            else
                printf("Currently Loaded DLL is: %s\n",
                    tmp_buf);

            /* free the already loaded module, whatever
             * it is */
            DosFreeModule(handle);
        }
        else
        {
            /* current handle is for requested DLL.
             * Simply return */
            printf("DLL (%s) already loaded\n",
                mod_name[debug_flag]);
            return(TRUE);
        }
    }

    /* wrong DLL is now freed */
    /* try to load the requested DLL, and get the entry
     * points */
    if(stat = DosLoadModule(fail_buf, FAIL_BUF_LEN,
                           mod_name[debug_flag],
                           &handle))
    {
        printf("Couldn't load: %s (stat is :%x)\n",
            mod_name[debug_flag], stat);
        printf("DLL problem was in: %s\n", fail_buf);
        return(FALSE);
    }
    else
        printf("Module handle is: %d\n", handle);
}

```

CONTINUED

Figure 3 CONTINUED

```

/* Now get the entry point for the requested routine */
if (stat = DosGetProcAddr(handle,
    routine_name[debug_flag], &routine))
{
    printf("Couldn't get routine: %s (stat is :%d)\n",
        routine_name[debug_flag], stat);
    return(FALSE);
}
else
    printf("Routine address is: %lx\n", routine);

/* module loaded, entry point returned, so we return */
return(TRUE);
}

```

```

[FNDFILE3.DEF]
LIBRARY    FNDFILE3
EXPORTS    do_find
EXPORTS    debug
IMPORTS    FNDFILE2.do_list

```

Partial Dump of \PMSDK\LIB\OS2.LIB

0FC0:	00 00 8A 02 00 00 00 00 00 00 00 00 00 00 00 00
0FD0:	80 0F 00 0D 44 4F 53 43 41 4C 4C 53 30 30 30 36DOSCALLS0006
0FE0:	33 16 88 1D 00 00 A0 01 01 0C 44 4F 53 46 49 4E	3.....DOSFIN
0FF0:	44 46 49 52 53 54 08 44 4F 53 43 41 4C 4C 53 40	DFIRST.DOSCALLS@
1000:	00 00 8A 02 00 00 00 00 00 00 00 00 00 00 00 00
1010:	80 0F 00 0D 44 4F 53 43 41 4C 4C 53 30 30 30 36DOSCALLS0006
1020:	34 15 88 1C 00 00 A0 01 01 0B 44 4F 53 46 49 4E	4.....DOSFIN
1030:	44 4E 45 58 54 08 44 4F 53 43 41 4C 4C 53 41 00	DNEXT.DOSCALLSA.
1040:	00 8A 02 00 00 00 00 00 00 00 00 00 00 00 00 00

▲ Figure 4 Looking through the OS2.LIB file, you can find the correlation between an OS/2 function call by its name and its ordinal reference

Before you can call the `DosLoadProcAddr` function, the appropriate DLL must be opened up and a handle, returned by the operating system, preserved for future reference. Use of the `DosGetModHandle` checks whether the program has already loaded the DLL; if so, `DosLoadModule`, the function which returns the handle address, is not called. If, however, the DLL is not open, this implies that the other DLL is open (for all calls to debug but the first). Therefore a call is made to `DosGetModName`, simply to show the name of the DLL already loaded, and the open DLL is closed via the `DosFreeModule` call.

The `DosLoadProcAddr` function returns the actual address of the requested function, whether

called with a function name (which should be in uppercase) or an ASCII-based ordinal number. Simply by calling this function via the normal C mechanisms, you can call any one of a number of functions, as long as the parameters match.

One of the functions with matching parameters is the `do_list` function. This is in `FNDFILE2` (see **Figure 2**). This function simply prints out all of your parameters, saving the ones the call might change, then calls the real `DosFindFirst` function. This implies that the function is already loaded when you first invoke the `FNDFILE` program. If you want to avoid this "preloading," you can also have the `do_list` function call all the appropriate run-time load functions for its own copy of the

`DosFindFirst` pointer—but this is only demonstration code, in any case.

The call to `DosFindFirst` is done simply to get back the status code, which is expected and will be processed by the `do_find` function. The number of files to be found on subsequent calls is reset, as is the original handle.

The important thing here to realize is that the `do_find` function calls either `do_list` or `DosFindFirst` without really knowing which. So in a transparent manner, you can create your program with full run-time loading of your difficult functions, debug them completely, then let the actual programming effort continue.

An obvious extension of this technique would let you provide your end-users with a full debug DLL of all your functions and all the OS/2 system calls you use. Then when you get that inevitable tech support phone call, you can just have the user turn on debug mode and modem you a copy of the voluminous output the debug DLL produces.

Run-time loading is one of the treasures in OS/2 which, with enough exploration, you'll find extraordinarily valuable as you produce some of the larger programs OS/2 makes possible. In fact, you can prompt for a DLL to be loaded, and load it on the spot! The name doesn't have to be in the code. Even the simple debug facility demonstrated here dwarfs the capabilities of other debugging methods in other operating systems. □

How the 8259A Programmable Interrupt Controller Manages External I/O Devices

Jim Kyle and Chip Rabinowitz

Unlike software interrupts, which are service requests initiated by a program, hardware interrupts occur in response to electrical signals received from a peripheral device such as a serial port or a disk controller, or they are generated internally by the microprocessor itself. Hardware interrupts, whether external or internal to the microprocessor, are given prioritized servicing by the Intel® CPU architecture.

The 8086 family of microprocessors (which includes the 8088, 8086, 80186, 80286, and 80386) reserves the first 1024 bytes of memory (addresses 0000:0000H through 0000:03FFH) for a table of 256 interrupt vectors, each a 4-byte far pointer to a specific interrupt service routine (ISR) that is carried out when the corresponding interrupt is processed. The design of the 8086 family requires certain of these interrupt vectors to be used for specific functions (see **Figure 1**). Although Intel actually reserves the first 32 interrupts, IBM, in the original PC, redefined usage of Interrupts 05H to 1FH. Most, but not all, of these reserved vectors are used by software, rather than hardware, interrupts; the redefined IBM uses are listed in **Figure 2**.

Nestled in the middle of **Figure 2** are the eight hardware interrupt vectors (08-0FH) IBM implemented in the original PC design. These eight vectors provide the maskable interrupts

for the IBM® PC-family and close compatibles. Additional IRQ lines built into the IBM PC/AT® are discussed under The IRQ Levels below.

The conflicting uses of the interrupts listed in **Figures 1** and **2** have created compatibility

Figure 1: Intel Reserved Exception Interrupts

Interrupt Number	Definition
00H	Divide by zero
01H	Single step
02H	Nonmaskable interrupt (NMI)
03H	Breakpoint trap
04H	Overflow trap
05H	BOUND range exceeded (see note 1)
06H	Invalid opcode (see note 1)
07H	Coprocessor not available (see note 2)
08H	Double-fault exception (see note 2)
09H	Coprocessor segment overrun (see note 2)
0AH	Invalid task state segment (see note 2)
0BH	Segment not present (see note 2)
0CH	Stack exception (see note 2)
0DH	General protection exception (see note 2)
0EH	Page fault (see note 3)
0FH	(Reserved)
10H	Coprocessor error (see note 2)

Note 1: The 80186, 80286, and 80386 microprocessors only.
 Note 2: The 80286 and 80386 microprocessors only.
 Note 3: The 80386 microprocessor only.

This article is an excerpt from The MS-DOS® Encyclopedia (Microsoft Press, 1988)—Ed.

Figure 2: IBM Interrupt Usage

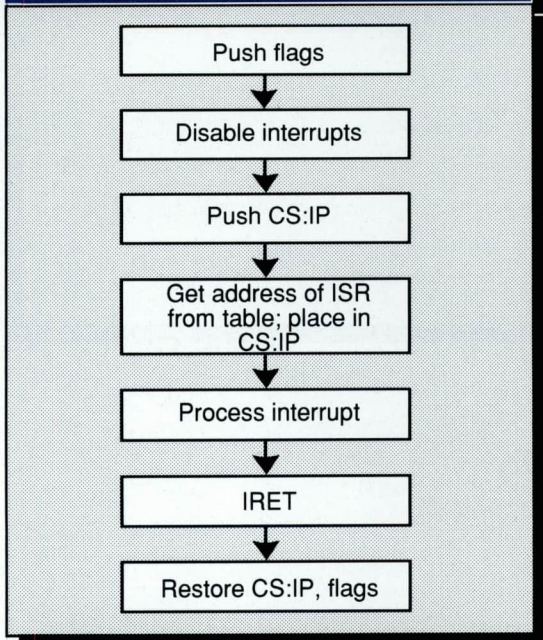
Interrupt Number	Definition
05H	Print screen
06H	Unused
07H	Unused
08H	Hardware IRQ0 (timer-tick) (see note 1)
09H	Hardware IRQ1 (keyboard)
0AH	Hardware IRQ2 (reserved) (see note 2)
0BH	Hardware IRQ3 (COM2)
0CH	Hardware IRQ4 (COM1)
0DH	Hardware IRQ5 (fixed disk)
0EH	Hardware IRQ6 (floppy disk)
0FH	Hardware IRQ7 (printer)
10H	Video service
11H	Equipment information
12H	Memory size
13H	Disk I/O service
14H	Serial-port service
15H	Cassette/network service
16H	Keyboard service
17H	Printer service
18H	ROM BASIC
19H	Restart system
1AH	Get/Set time/date
1BH	Control-Break (user defined)
1CH	Timer tick (user defined)
1DH	Video parameter pointer
1EH	Disk parameter pointer
1FH	Graphics character table

Note 1: IRQ = Interrupt request line.
Note 2: See figures 7 and 8.

microprocessor-generated exception interrupts (refer to **Figure 1**). Second is the non-maskable interrupt, or NMI (Interrupt 02H), which is generated when the NMI line (pin 17 on the 8088 and 8086, pin 59 on the 80286, pin B8 on the 80386) goes high (active). In the IBM PC family (except the PCjr and the Convertible), the nonmaskable interrupt is designated for memory parity errors. Third are the maskable interrupts, which are usually generated by external devices.

Maskable interrupts are sent to the main processor through a chip called the 8259A Programmable Interrupt Controller (PIC). When it receives an interrupt request, the PIC signals the microprocessor that an interrupt needs service by driving the interrupt request (INTR) line of the main processor to high voltage level. This article focuses on the maskable interrupts and the 8259A because it is through the PIC that external I/O devices (disk drives, serial communication ports, and so forth) gain access to the interrupt system.

Figure 3: General Interrupt Sequence



problems as the 8086 family of microprocessors has developed. For complete compatibility with IBM equipment, the IBM usage must be followed even when it conflicts with the chip design. For example, a BOUND error occurs if an array index exceeds the specified upper and lower limits (bounds) of the array, causing an Interrupt 05H to be generated. But the 80286 processor used in all AT-class computers will, if a BOUND error occurs, send the contents of the display to the printer, because IBM uses Interrupt 05H for the Print Screen function.

Hardware Interrupt Categories

The 8086 family of microprocessors can handle three types of hardware interrupts. First are the internal,

Interrupt Priorities

The Intel microprocessors have a built-in priority system for handling interrupts that occur simultaneously. Priority goes to the internal instruction exception interrupts, such as Divide by Zero and Invalid Opcode, because priority is determined by the interrupt number: Interrupt 00H takes priority over all others, whereas the last possible interrupt, 0FFH, would, if present, never be allowed to break in while another interrupt was being serviced. However, if interrupt service is enabled (the microprocessor's interrupt flag is set), any hardware interrupt takes priority over any software interrupt (INT instruction).

The priority sequencing by

Figure 4: Maskable Interrupt Service

interrupt number must not be confused with the priority resolution performed by hardware external to the microprocessor. The numeric priority discussed here applies only to interrupts generated within the 8086 family of microprocessor chips and is totally independent of system interrupt priorities established for components external to the microprocessor itself.

Interrupt Service Routines

For the most part, programmers need not write hardware-specific program routines to service the hardware interrupts. The IBM PC BIOS routines, together with MS-DOS services, are usually sufficient. In some cases, however, the MS-DOS operating system and the ROM BIOS do not provide enough assistance to ensure adequate performance of a program. Most notable in this category is communications software, for which programmers usually must access the 8259A and the 8250 Universal Asynchronous Receiver and Transmitter (UART) directly.

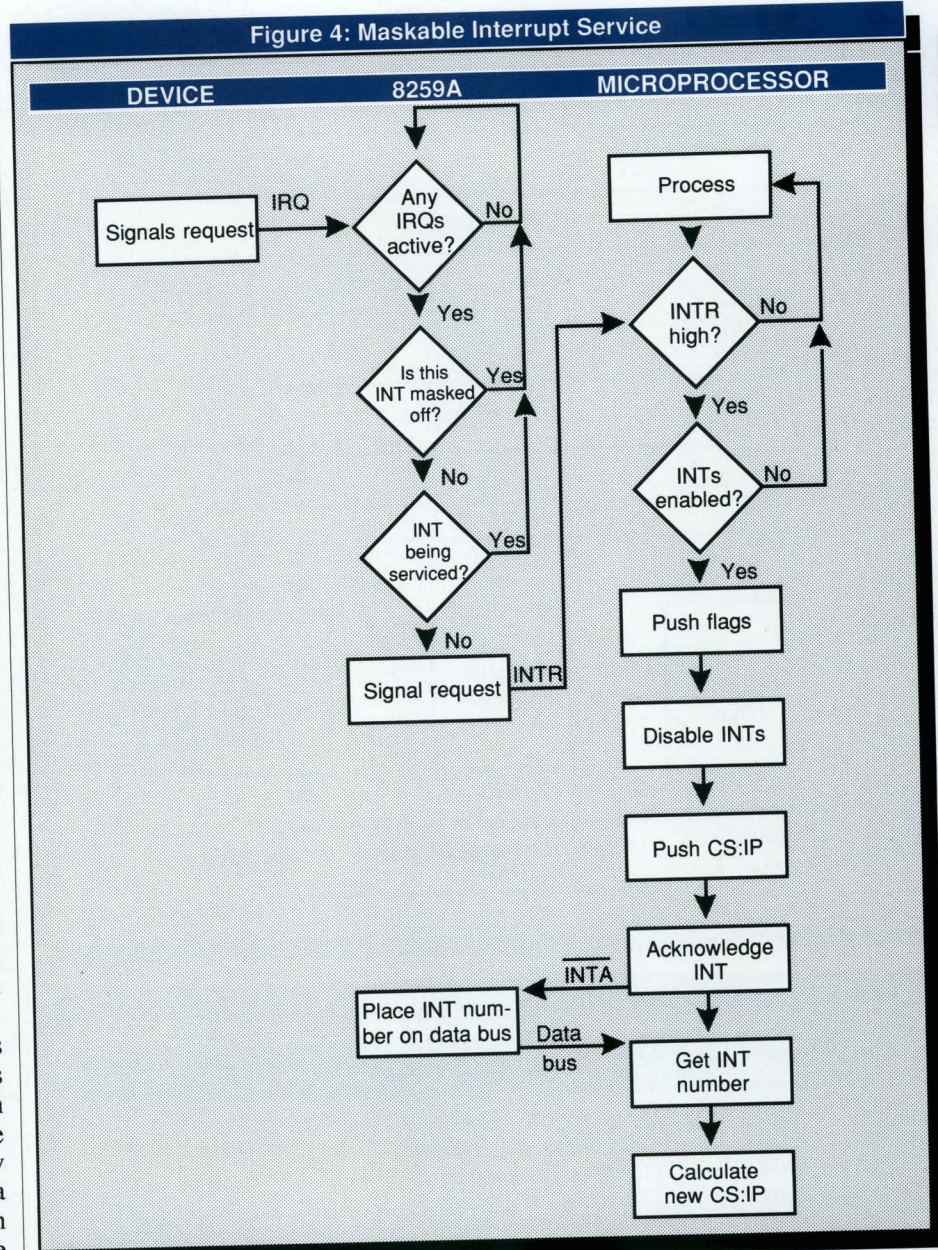
Two major characteristics distinguish maskable interrupts from all other events that can occur in the system: they are totally unpredictable, and they are highly volatile. In general, a hardware interrupt occurs when a peripheral device requires the complete attention of the system and data will be irretrievably lost unless the system responds rapidly.

All things are relative, however, and this is especially true of the speed required to service an interrupt request. For example, assume that two interrupt requests occur at essentially the same time. One is from a serial communications port receiving data at 300 bps; the other is from a serial port receiving data at 9600 bps. Data from the first serial port will not

change for at least 30 milliseconds, but the second serial port must be serviced within one millisecond to avoid data loss.

Unpredictability

Because maskable interrupts generally originate in response to external physical events, such as the receipt of a byte of data over a communications line, the exact time at which such an interrupt will occur cannot be predicted. Even the timer interrupt request, which by default



TWO MAJOR CHARACTERISTICS DISTINGUISH MASKABLE INTERRUPTS FROM ALL OTHER EVENTS THAT CAN OCCUR: THEY ARE TOTALLY UNPREDICTABLE AND HIGHLY VOLATILE.

Figure 5: Block Diagram of the 8259A Programmable Interrupt Controller

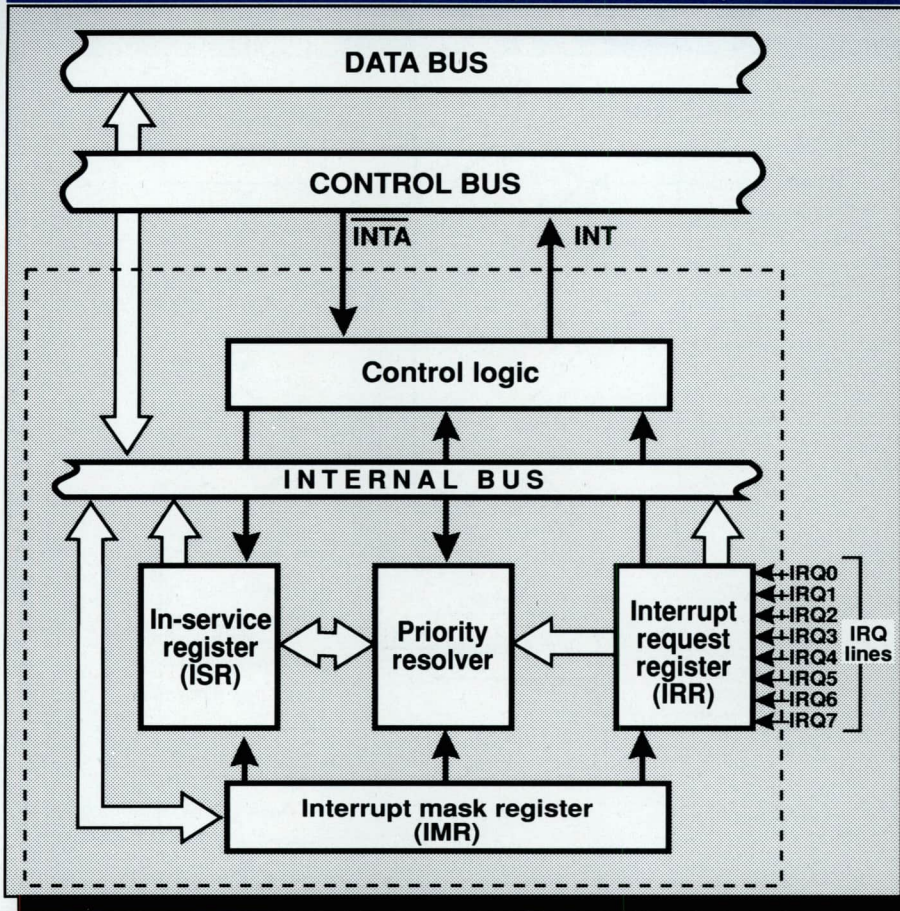


Figure 6: Eight-Level Interrupt Map

IRQ Line	Interrupt	Description
IRQ0	08H	Timer tick, 18.2 times per second
IRQ1	09H	Keyboard service required
IRQ2	0AH	I/O channel (unused on IBM PC/XT)
IRQ3	0BH	COM1 service required
IRQ4	0CH	COM2 service required
IRQ5	0DH	Fixed-disk service required
IRQ6	0EH	Floppy-disk service required
IRQ7	0FH	Data request from parallel printer (see note 1)

Note 1: This request cannot be reliably generated by older versions of the IBM Monochrome/Printer Adapter and compatibles. Printer drivers that depend on this signal for operation with these cards are subject to failure.

recognized, be prepared to service all maskable interrupt requests. Conversely, if interrupts cannot be serviced, they must all be disabled. The 8086 family of microprocessors provides the Set Interrupt Flag (STI) instruction to enable maskable interrupt response and the Clear Interrupt Flag (CLI) instruction to disable it. The interrupt flag is also cleared automatically when a hardware interrupt response begins; the interrupt handler should execute STI as quickly as possible to allow higher priority interrupts to be serviced.

Volatility

As noted earlier, a maskable interrupt request must normally be serviced immediately to prevent loss of data, but the concept of immediacy is relative to the data transfer rate of the device requesting the interrupt. The rule is that the currently available unit of data must be processed (at least to the point of being stored in a buffer) before the next such item can arrive. Except for such devices as disk drives, which always require immediate response, interrupts for devices that receive data are normally much more critical than interrupts for devices that transmit data.

The problems imposed by data volatility during hardware interrupt service are solved by establishing service priorities for interrupts generated outside the microprocessor chip itself. Devices with the slowest transfer rates are assigned lower interrupt service priorities, and the most time-critical devices are assigned the highest priority of interrupt service.

Maskable Interrupts

The microprocessor handles all interrupts (maskable, non-maskable, and software) by pushing the contents of the flags

occurs approximately 18.2 times per second, cannot be predicted by any program that happens to be executing when the interrupt request occurs.

Because of this unpredictability, the system must, if it allows any interrupts to be

register onto the stack, disabling the interrupt flag, and pushing the current contents of the CS:IP registers onto the stack.

The microprocessor then takes the interrupt number from the data bus, multiplies it by 4 (the size of each vector in bytes), and uses the result as an offset into the interrupt vector table located in the bottom 1Kb (segment 0000H) of system RAM. The 4-byte address at that location is then used as the new CS:IP value (see Figure 3).

External devices are assigned dedicated interrupt request lines (IRQs) associated with the 8259A. See the subsection titled "The IRQ Levels" below. When a device requires attention, it sends a signal to the PIC via its IRQ line. The PIC, which functions as an "executive secretary" for the external devices, operates as shown in Figure 4. It evaluates the service request and, if appropriate, causes the microprocessor's INTR line to go high. The microprocessor then checks whether interrupts are enabled, that is, whether the interrupt flag is set. If they are, the flags are pushed onto the stack, the interrupt flag is disabled, and CS:IP is pushed onto the stack.

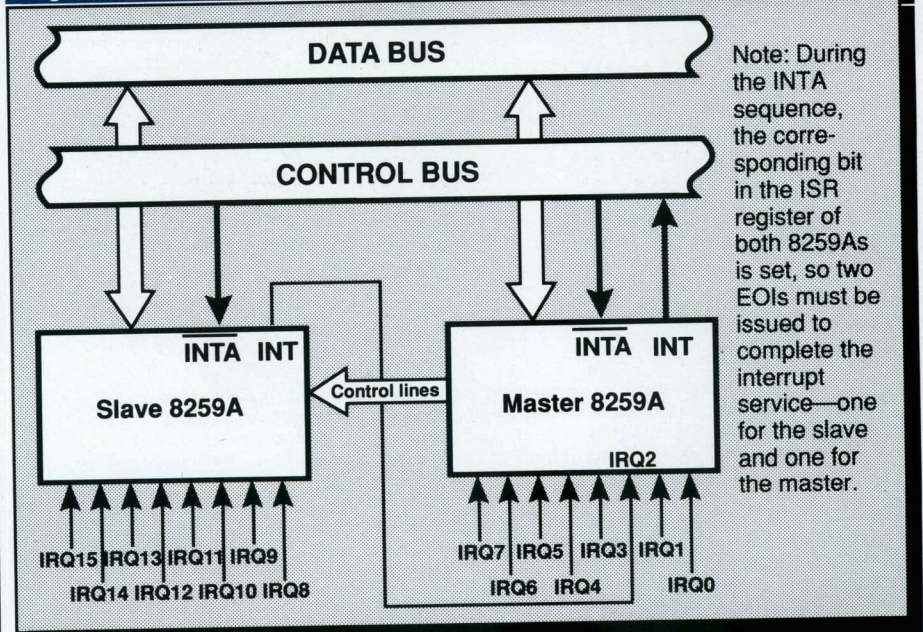
The microprocessor acknowledges the interrupt request by signaling the 8259A via the interrupt acknowledge (INTA) line. The 8259A then places the interrupt number on the data bus. The microprocessor gets the interrupt number from the data bus and services the interrupt. Before issuing the IRET instruction, the interrupt service routine must issue an end-of-interrupt (EOI) sequence to the 8259A so that other interrupts can be processed. This is done by sending 20H to port 20H. (The similarity of numbers is pure coincidence.)

The 8259A (see Figure 5) has a number of internal components,

Figure 7: Sixteen-Level Interrupt Map

IRQ Line	Interrupt	Description
IRQ0	08H	Timer tick, 18.2 times per second
IRQ1	09H	Keyboard service required
IRQ2	0AH	INT from slave 8259A:
IRQ8	70H	Real-time clock service
IRQ9	71H	Software redirected to IRQ2
IRQ10	72H	Reserved
IRQ11	73H	Reserved
IRQ12	74H	Reserved
IRQ13	75H	Numeric coprocessor
IRQ14	76H	Fixed-disk controller
IRQ15	77H	Reserved
IRQ3	0BH	COM2 service required
IRQ4	0CH	COM1 service required
IRQ5	0DH	Data request from LPT2
IRQ6	0EH	Floppy-disk service required
IRQ7	0FH	Data request from LPT1

Figure 8: A Graphic Representation of the Cascade Effect for IRQ Priorities



many of them under software control. Only the default settings for the IBM PC family are covered here.

Three registers influence the servicing of maskable interrupts: the interrupt request register (IRR), the in-service register (ISR), and the interrupt mask register (IMR).

The IRR is used to keep track of the devices requesting attention. When a device causes its

INTERRUPTS FOR DEVICES THAT RECEIVE DATA ARE NORMALLY MUCH MORE CRITICAL THAN INTERRUPTS FOR DEVICES THAT TRANSMIT DATA.

Figure 9 The Divide by Zero Replacement Handler

```

name          divzero
title         'DIVZERO - Interrupt 00H Handler'
;
;DIVZERO.ASM: Demonstration Interrupt 00H Handler
;This code is specific to 80286 and 80386 microprocessors.
;To assemble, link, and convert to a COM file:
;
;
;      MASM DIVZERO
;      LINK DIVZERO
;      EXE2BIN DIVZERO.EXE DIVZERO.COM
;      DEL DIVZERO.EXE
;
cr            equ            0dh            ; ASCII carriage return
lf            equ            0ah            ; ASCII linefeed
eos           equ            '$'           ; end of string marker

_TEXT        segment        word public 'CODE'
            assume         cs:_TEXT,ds:_TEXT,es:_TEXT,ss:_TEXT
            org            100h

entry:       jmp            start          ; skip over data area

intmsg       db             'Divide by Zero Occurred!',cr,lf,eos

divmsg       db             'Dividing '    ; message used by demo
par1         db             '0000h'        ; dividend goes here
            db             ' by '
par2         db             '00h'         ; divisor goes here
            db             ' equals '
par3         db             '00h'         ; quotient here
            db             'remainder'
par4         db             '00h'         ; and remainder here
            db             cr,lf,eos

oldint0      dd             ?              ; save old Int00H vector

intflag      db             0              ; nonzero if divide by
            ; zero interrupt occurred

oldip        dw             0              ; save old IP value

;
;The routine 'int0' is the actual divide by zero interrupt handler.
;It gains control whenever a divide by zero or overflow occurs. Its
;action is to set a flag and then increment the instruction pointer
;saved on the stack so that the failing divide will not be reexecuted
;after the IRET.
;
;In this particular case we can call MS-DOS to display a message during
;interrupt handling because the application triggers the interrupt
;intentionally. Thus, it is known that MS-DOS or other interrupt
;handlers are not in control at the point of interrupt.
;

int0:        pop            cs:oldip        ;capture instruction pointer

            push           ax
            push           bx
            push           cx
            push           dx
            push           di
            push           si
            push           ds
            push           es

            push           cs              ; set DS = CS
            pop            ds

            mov            ah,09h          ; print error message
            mov            dx,offset _TEXT:intmsg
            int             21h

```

IRQ line to go high, to signal the 8259A that it needs service, a bit is set in the IRR that corresponds to the interrupt level of the device.

The ISR specifies the interrupt levels that are currently being serviced; an ISR bit is set when an interrupt has been acknowledged by the CPU (via INTA) and the interrupt number has been placed on the data bus. The ISR bit associated with a particular IRQ remains set until an EOI sequence is received.

The IMR is a read/write register (at port 21H) that masks (disables) specific interrupts. When a bit is set in this register, the corresponding IRQ line is masked and no servicing for it is performed until the bit is cleared. Thus, a particular IRQ can be disabled while all others continue to be serviced.

The fourth major block in Figure 5, labeled Priority resolver, is a complex logical circuit that forms the heart of the 8259A. This component combines the statuses of the IMR, the ISR, and the IRR to determine which, if any, pending interrupt request should be serviced and then causes the microprocessor's INTR line to go high. The priority resolver can be programmed in a number of modes, although only the mode used in the IBM PC and close compatibles is described here.

The IRQ Levels

When two or more unserviced hardware interrupts are pending, the 8259A determines which should be serviced first. The standard mode of operation for the PIC is the fully nested mode, in which IRQ lines are prioritized in a fixed sequence. Only IRQ lines with higher priority than the one currently being serviced are permitted to generate new interrupts.

The highest priority is IRQ0, and the lowest is IRQ7. Thus, if

an Interrupt 09H (signaled by IRQ1) is being serviced, only an Interrupt 08H (signaled by IRQ0) can break in. All other interrupt requests are delayed until the Interrupt 09H service routine is completed and has issued an EOI sequence.

Eight-level Designs

The IBM PC and PC/XT (and port-compatible computers) have eight IRQ lines to the PIC chip—IRQ0 through IRQ7. These lines are mapped into interrupt vectors for Interrupts 08H through 0FH (that is, 8 + IRQ level). These eight IRQ lines and their associated interrupts are listed in Figure 6.

Sixteen-level Designs

In the IBM PC/AT, 8 more IRQ levels have been added by using a second 8259A PIC (the "slave") and a cascade effect, which gives 16 priority levels.

The cascade effect is accomplished by connecting the INT line of the slave to the IRQ2 line of the first, or master, 8259A instead of to the microprocessor. When a device connected to one of the slave's IRQ lines makes an interrupt request, the INT line of the slave goes high and causes the IRQ2 line of the master 8259A to go high, which, in turn, causes the INT line of the master to go high and thus interrupts the microprocessor.

The microprocessor, ignorant of the second 8259A's presence, simply generates an interrupt acknowledge signal on receipt of the interrupt from the master 8259A. This signal initializes both 8259A and also causes the master to turn control over to the slave. The slave then completes the interrupt request.

On the IBM PC/AT, the eight additional IRQ lines are mapped to Interrupts 70H through 77H (see Figure 7). Because the eight additional lines are effectively connected to the master

Figure 9 CONTINUED

```

add     oldip,2           ; bypass instruction causing
                        ; divide by zero error

mov     intflag,1        ; set divide by 0 flag

pop     es               ; restore all registers
pop     ds
pop     si
pop     di
pop     dx
pop     cx
pop     bx
pop     ax

push    cs:oldip        ; restore instruction pointer

iret                                ; return from interrupt

;
;The code beginning at 'start' is the application program. It alters
;the vector for Interrupt 00H to point to the new handler, carries
;out some divide operations (including one that will trigger an
;interrupt) for demonstration purposes, restores the original
;contents of the Interrupt 00H vector, and then terminates.
;
start:  mov     ax,3500h   ; get current contents
        int     21h      ; of Int 00H vector

                                ; save segment:offset
                                ; of previous Int 00H handler
        mov     word ptr oldint0,bx
        mov     word ptr oldint0+2,es

        mov     dx,offset int0 ; install new handler ...
                                ; DS:DX = handler address
        mov     ax,2500h   ; call MS-DOS to set
        int     21h      ; Int 00H vector

                                ; now our handler is active,
                                ; carry out some test divides.

        mov     ax,20h    ; test divide
        mov     bx,1      ; divide by 1
        call    divide

        mov     ax,1234h  ; test divide
        mov     bx,5eh    ; divide by 5EH
        call    divide

        mov     ax,5678h  ; test divide
        mov     bx,7fh    ; divide by 127
        call    divide

        mov     ax,20h    ; test divide
        mov     bx,0      ; divide by 0
        call    divide    ; (triggers interrupt)

                                ; demonstration complete,
                                ; restore old handler
        lds     dx,oldint0 ; DS:DX = handler address
        mov     ax,2500h   ; call MS-DOS to set
        int     21h      ; Int 00H vector

        mov     ax,4c00h  ; final exit to MS-DOS
        int     21h      ; with return code = 0

;
;The routine 'divide' carries out a trial division, displaying the
;arguments and the results. It is called with AX = dividend and
;BL = divisor.
;

```

CONTINUED

Figure 9 CONTINUED

```

divide proc near
    push ax ; save arguments
    push bx

    mov di,offset par1 ; convert dividend to
    call wtoa ; ASCII for display

    mov ax,bx ; convert divisor to
    mov di,offset par2 ; ASCII for display
    call btoa

    pop bx ; restore arguments
    pop ax

    div b1 ; perform the division
    cmp intflag,0 ; divide by zero detected?
    jne nodiv ; yes, skip display

    push ax ; no, convert quotient to
    mov di,offset par3 ; ASCII for display
    call btoa

    pop ax ; convert remainder to
    xchg ah,al ; ASCII for display
    mov di,offset par4
    call btoa

    mov ah,09h ; show arguments, results
    mov dx,offset divmsg
    int 21h

nodiv: mov intflag,0 ; clear divide by 0 flag
       ret ; and return to caller

divide endp

wtoa proc near ; convert word to hex ASCII
       ; call with AX = binary value
       ; DI = addr for string
       ; returns AX, CX, DI destroyed

    push ax ; save original value
    mov al,ah
    call btoa ; convert upper byte
    add di,2 ; increment output address
    pop ax
    call btoa ; convert lower byte
    ret ; return to caller

wtoa endp

btoa proc near ; convert byte to hex ASCII
       ; call with AL = binary value
       ; DI = addr to store string
       ; returns AX, CX destroyed

    mov ah,al ; save lower nibble
    mov cx,4 ; shift right 4 positions
    shr al,cl ; to get upper nibble
    call ascii ; convert 4 bits to ASCII
    mov [di],al ; store in output string
    mov al,ah ; get back lower nibble

    and al,0fh ; blank out upper one
    call ascii ; convert 4 bits to ASCII
    mov [di+1],al ; store in output string
    ret ; back to caller

btoa endp

```

CONTINUED

8259A's IRQ2 line, they take priority over the master's IRQ3 through IRQ7 events. The cascade effect is graphically represented in Figure 8.

Programming for the Hardware Interrupts

Any program that modifies an interrupt vector must restore the vector to its original condition before returning control to DOS (or to its parent process). Any program that totally replaces an existing hardware interrupt handler with one of its own must perform all the handshaking and terminating actions of the original: reenables interrupt service, signal EOI to the interrupt controller, and so forth. Failure to follow these rules has led to many hours of programmer frustration.

When an existing interrupt handler is completely replaced with a new, customized routine, the existing vector must be saved so it can be restored later. Although it is possible to modify the 4-byte vector by directly addressing the vector table in low RAM (and many published programs have followed this practice), any program that does so runs the risk of causing system failure when the program is used with multitasking or multiuser enhancements or with future versions of DOS. The only technique that can be recommended for either obtaining the existing vector values or changing them is to use the MS-DOS functions provided for this purpose: Interrupt 21H Functions 25H (Set Interrupt Vector) and 35H (Get Interrupt Vector).

After the existing vector has been saved, it can be replaced with a far pointer to the replacement routine. The new routine must end with an IRET instruction. It should also take care to preserve all microprocessor registers and conditions at entry and restore them before returning.

Replacement Handler

Suppose a program performs many mathematical calculations of random values. To prevent abnormal termination of the program by the default MS-DOS Interrupt 00H handler when a DIV or IDIV instruction is attempted and the divisor is zero, a programmer might want to replace the Interrupt 00H (Divide by Zero) routine with one that informs the user of what has happened and then continues operation without abnormal termination. The COM program DIVZERO.ASM (see Figure 9) does just that.

Supplementary Handlers

In many cases, a custom interrupt handler augments, rather than replaces, the existing routine. The added routine might process some data before passing the data to the existing routine, or it might do the processing afterward. These cases require slightly different coding for the handler.

If the added routine is to process data before the existing handler does, the routine need only jump to the original handler after completing its processing. This jump can be done indirectly, with the same pointer used to save the original content of the vector for restoration at exit. For example, a replacement Interrupt 08H handler that merely increments an internal flag at each timer tick can look something like the code in Figure 10.

The added handler must preserve all registers and machine conditions, except for those machine conditions that it will modify, such as the value of myflag in the example (and the flags register, which is saved by the interrupt action), and it must restore those registers and conditions before performing the jump to the original handler.

Figure 9

```

ascii proc near ; convert AL bits 0-3 to
                ; ASCII (0...9,A...F)
                ; and return digit in AL
    add     al,'0'
    cmp     al,'9'
    jle     ascii2
    add     al,'A'-'9'-1 ; "fudge factor" for A-F
    ret
ascii2 ret ; return to caller
ascii     endp
_TEXT    ends
        end     entry
    
```

Figure 10: Example of a Supplementary Handler

```

;
;
myflag dw ? ; variable to be incremented
           ; on each timer-tick interrupt

oldint8 dd ? ; contains address of previous
              ; timer-tick interrupt handler

;
; ; get the previous contents
; ; of the Interrupt 08H vector...
; ; AH = 35H (Get Interrupt Vector)
mov     ax,3508h ; AL = Interrupt number (08H)
int     21h
mov     word ptr oldint8,bx ; save the address of the
mov     word ptr oldint8+2,es ; previous Int 08H Handler
mov     dx,seg myint8 ; put address of the new
mov     ds,dx ; interrupt handler into DS:DX
mov     dx,offset myint8 ; and call MS-DOS to set vector
mov     ax,2508h ; AH = 25H (Set Interrupt Vector)
int     21h ; AL = Interrupt number (08H)
;
;
myint8: ; this is the new handler
         ; for Interrupt 08H

inc     cs:myflag ; increment variable on each
                ; timer-tick interrupt

jmp     dword ptr cs:[oldint8] ; then chain to the
                ; previous interrupt handler
    
```

A more complex situation arises when a replacement handler does some processing after the original routine executes, especially if the replacement handler is not reentrant. To allow for this processing, the replacement handler must prevent nested interrupts, so that even if the old handler (which is chained to the replacement handler by a CALL instruction) issues an EOI, the replacement handler will not be interrupted during postprocessing. For example, instead of using the preceding Interrupt 08H example routine, the programmer could use the code shown in

ANY PROGRAM THAT MODIFIES AN INTERRUPT VECTOR MUST RESTORE THE VECTOR TO ITS ORIGINAL CONDITION BEFORE RETURNING CONTROL TO DOS.

Figure 11: Replacement Handler Utilizing a Semaphore

```

myint8:                                ; this is the new handler
                                        ; for Interrupt 08H

    mov     ax,1                        ; test and set interrupt-
    xchg   cs:myflag,ax                 ; handling-in-progress
                                        ; semaphore

    push   ax                            ; save the semaphore

    pushf                                     ; simulate interrupt,
                                        ; allowing the previous
    call  dword ptr cs:oldint8          ; handler for the
                                        ; Interrupt 08H
                                        ; vector to run

    pop    ax                             ; get the semaphore back
    or     ax,ax                         ; is our interrupt handler
                                        ; already running?

    jnz    myint8x                       ; yes, skip this one

    .                                        ; now perform our interrupt
    .                                        ; processing here...
    .

    mov    cs:myflag,0                  ; clear the interrupt-
                                        ; handling-in-progress
                                        ; flag

myint8x:
    iret                                ; return from interrupt

```

A CUSTOM INTERRUPT HANDLER ROUTINE AUGMENTS, RATHER THAN REPLACES, THE EXISTING ROUTINE. THE ADDED ROUTINE MIGHT PROCESS SOME DATA BEFORE PASSING THE DATA TO THE EXISTING ROUTINE, OR IT MIGHT DO THE PROCESSING AFTERWARD. THESE CASES REQUIRE SLIGHTLY DIFFERENT CODING FOR THE HANDLER.

Figure 11 to implement myflag as a semaphore and use the XCHG instruction to test it.

Note that an interrupt handler of this type must simulate the original call to the interrupt routine by first doing a PUSHF, followed by a far CALL via the saved pointer to execute the original handler routine. The flags register pushed onto the stack is restored by the IRET of the original handler. Upon return from the original code, the new routine can preserve the machine state and do its own processing, finally returning to the caller by means of its own IRET.

The flags inside the new routine need not be preserved, as they are automatically restored by the IRET instruction. Because of the nature of interrupt servicing, the service routine should not depend on any information in the flags register, nor can it return any information in the flags register. Note also that the previous handler (invoked by the indirect CALL) will almost certainly have dismissed the interrupt by

sending an EOI to the 8259A PIC. Thus, the machine state is not the same as in the first myint8 example.

To remove the new vector and restore the original, the program simply replaces the new vector (in the vector table) with the saved copy. If the substituted routine is part of an application program, the original vector must be restored for every possible method of exiting from the program (including Control-Break, Control-C, and critical-error Abort exits). Failure to observe this requirement invariably results in system failure. Even though the system failure might be delayed for some time after the exit from the offending program, as soon as some subsequent program overlays the interrupt handler code the crash is imminent.

Summary

Hardware interrupt handler routines, although not strictly a part of DOS, form an integral part of many MS-DOS programs and are tightly constrained by MS-DOS requirements. Routines of this type play important roles in the functioning of the IBM personal computers, and, with proper design and programming, significantly enhance product reliability and performance. In some instances, no other practical method exists for meeting performance requirements. □

Advanced Techniques for Using Structures and Unions In Your C Code

Greg Comeau

Structures, unions, and typedefs, constructs essential to organizing data in your programs, are often misunderstood and poorly used. In "Organizing Data in Your C Programs with Structures, Unions, and Typedefs," *MSJ* (Vol. 4, No. 2), I attempted to clear up some of the mysteries surrounding the use of the constructs. In this article, I build on that discussion by examining pointers to structures and then move on to some of the finer points of multilevel structure access, memory allocation, and arrays.

The previous article may have convinced some of you to use typedef more often in your programs. However, when doing so, you should be aware of one particular problem; structures that contain references to themselves in the form of a pointer can begin to produce mysterious syntax errors. **Figure 1** shows an example of such a situation. The problem here is that the typedef for the identifier `s1` has not yet been completed on line 2; therefore the compiler cannot acknowledge the existence of the `s1` typedef or even the existence of any name for `s1` and must deduce that an invalid type is being used.

The case shown in **Figure 2** is a different situation but a similar problem. Two typedefs are set up; however, they each refer to the other (in circular fashion). In this instance, the first typedef always gives an error since the second one does not yet exist.

Is there a solution to these situations? There are a few; however, some are wrong and some are messy. Without a clear syntactic way to remedy this, many programmers are tempted to change the `s1 *` and/or the `s2 *` into `char *` (yes, either one since the typedefs may actually be found in `#include` files whose order of occurrence in your source file is not determinable) and then, when using or assigning to `s1ptr` or `s2ptr`, will cast it into `s1 *` or `s2 *`, respectively. This is a mess and a sure way to get into trouble with a different compiler or hardware. Casting into `s1 *` or `s2 *` is a sure bet that something will go wrong sooner or later since the cast will keep the compiler from saying anything (that is, a cast is a directive to tell the compiler that you've decided that you know what you're doing with a mismatched type. Suffice it to say for now that even though a cast implies portability, it in no way guarantees it).

The description of the problem having been stated, let's discuss possible ways to go about

STRUCTURES, UNIONS,
AND TYPEDEFS,
CONSTRUCTS ESSENTIAL TO
ORGANIZING DATA IN YOUR
PROGRAMS, ARE OFTEN
MISUNDERSTOOD AND
POORLY USED. IN THIS
ARTICLE, I EXAMINE
POINTERS TO STRUCTURES
AND THEN MOVE ON TO
THE FINER POINTS OF
MULTILEVEL STRUCTURE
ACCESS, MEMORY
ALLOCATION, AND ARRAYS.

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.

Figure 1: A Structure Containing a Reference to Itself

```

1  typedef struct {
2      s1      *s1ptr;
3      char    s1data[100];
4  } s1;
5
6  main()
7  {
8      s1      s1instance;
9
10     /* ... */
11 }

```

Figure 2: Two Typedefs Referring to Each Other in Circular Fashion

```

1  typedef struct {
2      s2      *s2ptr;
3      char    s1data[100];
4  } s1;
5
6  typedef struct {
7      s1      *s1ptr;
8      char    s2data[100];
9  } s2;
10
11 main()
12 {
13     s1      s1instance;
14     s2      s2instance;
15
16     /* ... */
17 }

```

THE PROBLEM OF INCOMPLETE TYPES IN C IS AN IMPORTANT ONE TO BE AWARE OF. IT ACTUALLY GOES WELL BEYOND THE DISCUSSION PRESENTED HERE AND IS NOT RELATED TO STRUCTURES ONLY. IN GENERAL, YOU CAN ALWAYS REFER TO AN INCOMPLETE TYPE BUT CAN NEVER USE IT AS A SINGLE ENTITY, THAT IS, AS A UNIT BY ITSELF, SINCE IT IS NOT COMPLETELY DEFINED.

solving it. We'll begin by breaking the problem down into various steps. As we move along I will introduce some other concepts that you might not be aware of along the way. Let me just say in passing that Pascal fans will be glad to hear that the solution is to use a forward reference. We will go through several variations of this technique.

Forward References

Let's make the problem simpler for a moment by removing the typedef from the examples (Figure 3). At this point it should be obvious that the ability to reference a pointer to s1 or s2 in the code sample does not exist. In other words, s1 and s2 are clearly not types. However, what if we were to add a structure tag to each declaration first so that the structure shape can be referenced instead (Figure 4)? The key here is that now s2ptr and s1ptr must not deal with a "type of;" they only serve as references to some structure_tag.

Note, however, that though a

forward reference through a pointer declaration is legitimate, an instance of the structure is not. If we look at Figure 5, the declaration of anotherptr is fine, since the pointer does not yet have to know what the shape of the structure it points to (the shape of the struct another tag) looks like. In C, this is one way that an incomplete type can exist. Of course, if you wish to use anotherptr beyond having the pointer assigned to it, you must first define the structure to which it points.

Similarly, the declaration of anotherinstance will fail, since I was attempting to include a whole instance of another tag within sometag. This can't happen because the compiler has no way to determine sometag's members and therefore it is size-less. The compiler can't delay this either since sometag would not have a valid size and shape until another tag had a size and shape, thus creating an error.

The problem of incomplete types in C is an important one to be aware of. It actually goes well beyond the discussion presented here and is not related to structures only. In general, you can always refer to an incomplete type but can never use it as a single entity, that is, as a unit by itself, since it is not completely defined. Another popular place where incomplete types are commonly used is with external arrays, for example, in a declaration such as extern int array[];. Here you can index and get the address of the array as normal; however, you cannot perform some operations such as sizeof(array) since the dimension of the array may not be in the same source file of such a declaration. This happens because array only serves as a declaration and not a definition.

Typedef/Structure Tags

Generally speaking, when

you are considering the use of `#define` and `typedef`, it is usually safe to conclude that `typedef` is the better choice. Here I have introduced yet another major source of confusion when using structs by using structure tags. However, do not get the mistaken impression that I'm saying tags are better than `typedefs`, since this is not necessarily the case. I've only used tags as a stepping stone.

For instance, now that forward references are clearer, let's turn back to the original problem and solve that using `typedefs`. There are two common ways to set this up (see **Figures 6 and 7**)—both are equivalent. In the first, two `typedefs` are set up that refer to structure tags (and since a `typedef` only serves as a synonym, we only care that it matches a structure tag that can be defined at a later time). This brings the two `typedefs` into scope and lets you use them from that point onward, simply using the names `s1` and `s2` as if they were types. This is true even when using them within the definitions of the structure tags that they were based upon, as shown in **Figure 6**.

The second method (**Figure 7**) allows us to produce the same result using a different method. Actually it's the reverse of the previous procedure: instead of `typedefing` the names and creating the structure tags last, we'll create the structure tags as we're `typedefing` by using structure tags internally.

Passing/Returning Structs

I'm assuming that the reader knows how to pass structures to functions. It should be sufficient to say that you should almost always pass a pointer to a structure or the address of a structure (`&struct_var`, which is mappable to a pointer to a structure) as the argument to a function instead of the actual structure

Figure 3: Defining Structs without Typedefs

```

1  struct {
2      s2      *s2ptr;
3      char    s1data[100];
4  } s1;
5
6  struct {
7      s1      *s1ptr;
8      char    s2data[100];
9  } s2;
10
11 main()
12 {
13     s1      s1instance;
14     s2      s2instance;
15
16     /* ... */
17 }
```

Figure 4: Adding a Structure Tag

```

1  struct s1_tag {
2      struct s2_tag *s2ptr;
3      char    s1data[100];
4  };
5
6  struct s2_tag {
7      struct s1_tag *s1ptr;
8      char    s2data[100];
9  };
10
11 main()
12 {
13     struct s1_tag s1instance;
14     struct s2_tag s2instance;
15
16     /* ... */
17 }
```

itself. In other words, pass the structure by reference, not by value. The latter case can require an unreasonable amount of stack space and, if you are not careful, can cause symptoms that include random crashes, lockups, invalid pointers, and other such problems.

Returning a structure should also be through a pointer, mostly for the sake of efficiency. This involves three areas: returning a complete structure from the function, returning a pointer to a structure, and the case where just some of the structure's members are accessed after function(s) return them.

Returning a Structure

When returning a complete structure from the function, consider what the compiler must go through. A typical compiler might copy the structure into a

PASS A STRUCTURE BY REFERENCE, NOT BY VALUE. THE LATTER CASE CAN REQUIRE AN UNREASONABLE AMOUNT OF STACK SPACE AND, IF YOU ARE NOT CAREFUL, CAN CAUSE SYMPTOMS THAT INCLUDE RANDOM CRASHES, LOCKUPS, INVALID POINTERS, AND OTHER SUCH PROBLEMS.

Figure 5: Nonlegal Formal Reference

```

1  struct atag {
2      struct another_tag *another_ptr;
3      /* reference to another tag before
4         it comes into scope is fine */
5  };
6
7  struct sometag {
8      struct another_tag another_instance;
9      /* This is an error since a
10         description of another_tag
11         is not in scope */
12  };
13
14  main()
15  {
16      /* ... */
17  }

```

Figure 6: Using Typedefs that Refer to Structure Tags

```

1  typedef struct s2_tag s2;
2  typedef struct s1_tag s1;
3
4  struct s1_tag {
5      s2 *s2_ptr;
6      char s1data[100];
7  };
8
9  struct s2_tag {
10     s1 *s1_ptr;
11     char s2data[100];
12 };
13
14 main()
15 {
16     struct s1_tag s1instance;
17     struct s2_tag s2instance;
18
19     /* ... */
20 }

```

YOU WILL ONLY BE ABLE TO OBTAIN AN IDENTIFIER'S ADDRESS WHILE IT IS IN SCOPE; BUT ONCE YOU HAVE THE ADDRESS OF AN OBJECT, DURING ITS LIFETIME IT'S YOURS TO DO WITH AS YOU PLEASE (WITHIN REASON OF COURSE).

static area set aside by the compiler and linker. This area can be thought of as a union of every structure in your program. This of course means that it must be large enough to hold any given structure. It also needs to be addressable without any idiosyncrasies, and must be static with suitable alignment qualities.

This copying alleviates much of the game playing the generated object code goes through since there is usually a standard function calling and return convention that the compiler writer will try to make the code adhere to. There are, however, problems. Consider the program in **Figure 8**, specifically the call to `func` on line 25. This should work as if `f1` and `f2` return base types, since you are, after all, allowed to return structures

from functions. Note the series of events that must occur—`f1` will return a struct `ps1` by copying it into a static area; next that static area might be copied onto the stack, which is accomplished on some systems by generating lots of “move long word” instructions instead of a loop. The same will be performed for `f2`'s return value.

The end result is reached by a slow and tedious process. But that's only half the problem. If you pass the structures returned from `f1` and `f2` by their addresses as shown in line 26 (you can do this since a structure is an aggregate and not a simple scalar), you may find that your compiler does not work as expected. Both `f1` and `f2` might return the same address (remember our friend provided by the linker), thus the addresses and strings printed out by `func2` could be the same.

Returning a Pointer

The second case is quite simple: don't return a pointer to a structure that's automatic. **Figure 9** shows an example of such misuse. Notice that `func` returns `&temp`, which is perfectly valid C but not valid logic. When `func` exits, the temp structure—not the temp tag, which has file scope in this example—will terminate since it is local to a function and nonstatic. Since `temp` has terminated, accessing it will result in undefined behavior and most likely a program crash.

Note that changing the declaration to static struct `temp temp`; will not cause any problems. Because `temp` is now static it will have a permanent existence during the life of your program. In this case it doesn't matter if the function that houses `temp` is currently executing or not; the fact is that `temp` is addressable since it's static.

Additionally, whether `temp` is in scope or not is irrelevant here. Scope relates only to identifiers.

It does not deal with concepts such as variables—for instance setting a pointer to an absolute position and accessing it—or things like addresses. Of course you will only be able to obtain an identifier's address while it is in scope; but once you have the address of an object, during its lifetime it's yours to do with as you please (within reason of course).

Member Access

The last scenario is one in which a function returns a structure, but you're only interested in accessing a few of its members. Think about this for a moment. Should you access the members by returning the structure and be hit with all the copying or should you take the easy route by returning a pointer to the structure?

In the case of an operating system call, you'll rarely want a pointer to one of the internal data structures of the operating system unless you're writing a device driver or a very specialized application that requires some special knowledge.

In another situation you might find you're calling a C library function instead of a system call, and are most likely accessing something in your "process's space." But the arguments to the particular library you are using are usually a given and neither can nor should be changed. Routines may accept an argument that's a pointer to a structure (one that you've properly allocated based on some #include file) and completely avoid a return value, which would have been a structure.

The routine itself will set the structure's members as might be appropriate. This will prevent the situation I've presented and is a viable way to code some of your programs. In addition, you might find routines of your own that use large structures, and

Figure 7: Typedefing Using Structure Tags Internally

```

1  typedef struct s1_tag {
2      struct s2_tag *s2ptr;
3      char    s1data[100];
4  };
5
6  typedef struct s2_tag {
7      struct s1_tag *s1ptr;
8      char    s2data[100];
9  };
10
11 main()
12 {
13     struct s1_tag s1instance;
14     struct s2_tag s2instance;
15
16     /* ... */
17 }
```

instead of passing whole entities back and forth, it is often worth creating another smaller structure which is a subset of the larger one and manipulating that instead.

Depending upon your logic and program design, you will also find it worthwhile to return a pointer to a structure even if only one of two members is going to be used. You only have to apply the arrow operator (->) to the returned pointer to access the member in question, rather than dealing with the whole structure. Remember that you may be calling a function containing an instance of the structure it's returning a pointer to as an internal static identifier. However, the consequence is that calling the function twice could destroy the previous value of the structure, so the calling function must account for this and copy all the data it needs before calling the previously called function again.

This presents another situation in which either the caller of the function or the function itself dynamically allocated the structure. You must therefore be careful to control the allocation of the structure and be just as careful to make sure that you free it properly.

Structure Access

Figure 10 contains both structures with pointers to other

structures and instances of the other structures. Chances are that you will never encounter this specific situation. Nevertheless I've included it to provide some further insights into structures in general.

Most of you can no doubt construct a simple structure member reference as in line 21 of Figure 10. Some of you can even create a multistructure or multilevel reference as in line 23—but without certainty that it is correct (it is). Anything beyond this, however (meaning those nasty creatures we call pointers), is unfamiliar territory.

To understand line 23, you must be aware that s3 contains an instance (that is, an occurrence) of an s2tag structure (an s2tag structure named s2inst), which in turn contains an instance to an s1tag structure (s1inst), which of course contains an instance of an integer identifier named s1var.

To use each of these instances, you simply create a structure access to the member, as shown in line 21. Since the operator precedence of the dot (.) operator presents no problem and its associativity is naturally oriented to be left to right, the setup is simply

```
structure1.<...>.structureN
```

which line 23 shows.

Remember that this all takes place within s3 because s3 con-

Figure 8: Returning a Complete Structure from a Function

```

1  struct ps1 {
2      char    *p;
3      char    array[1024 - sizeof(char *)];
4  } v1;
5
6  struct ps2 {
7      char    *p;
8      char    array[512 - sizeof(char *)];
9  } v2;
10
11 struct ps1 f1()
12 {
13     v1.p = "hi there";
14     return (v1);
15 }
16
17 struct ps2 f2()
18 {
19     v2.p = "HI THERE";
20     return (v2);
21 }
22
23 main()
24 {
25     func(f1(), f2());
26     func2(&f1(), &f2()); /* This doesn't mean the
27                          address of f1 and f2!
28                          It means the address
29                          of the structures they
30                          return—Remember
31                          precedence! */
32 }
33
34 func(struct ps1 v1, struct ps2 v2)
35 {
36
37     printf("%s\n", v1.p);
38     printf("%s\n", v2.p);
39 }
40
41 func2(struct ps1 *v1, struct ps2 *v2)
42 {
43
44     printf("%ld\n", v1); /* print out the adrrs
45                          that v1 and v2 point
46                          to, these */
47
48     printf("%ld\n", v2); /* may also be printed
49                          with a %p instead of
50                          %ld */
51
52     printf("%s\n", v1->p);
53     printf("%s\n", v2->p);
54 }

```

tains an s2tag and s2tag contains an s1tag. Under a different case, such as in lines 25–28, the code will use references to s3tag, s2tag, and s1tag by way of ps3, ps2, and ps1, which go beyond the limits of s3 to gain access to other variables such as s2 and s1.

On line 25, ps3 is set equal to the address of s3. Note the use of &s3 instead of s3 since we want to obtain the address of s3 and not an assignment to the pointer of the actual contents of the s3 structure. Once this is done, we can reference the members of

the structure (s3 in this case) that ps3 points to by using the -> operator. It happens that we want to assign to ps2, which is another structure pointer (a pointer to an s2tag). This assignment will take place just as smoothly as the ps3 assignment.

The evaluation of the arrow operator will take place in exactly the same order of precedence as the dot operator. Again, this will occur with left to right associativity. Our knowledge of line 23 coupled with this discussion should make the inter-

pretation of lines 27 and 28 very easy. Briefly then, line 27 uses the fact that ps3->ps2 references an s2tag, which contains a reference to an s1tag. This allows ps3->ps2->ps1 to be assigned to an identifier such as s1, which has an s1tag size and shape. Line 28 uses ps3->ps2->ps1, which is a pointer to an s1tag and therefore a reference to a member such as s1var is possible as well.

Every pointer used in lines 25–27 needed to be initialized. You could not simply have coded the access to s1var in line 28 without the other assignments. That would not be valid logic since every pointer in this example must access a memory location in order to be used.

Be careful here since this constraint is something that you as the programmer must take care to enforce. The compiler doesn't care, for instance, that you may have coded line 28 without lines 25 through 27. You may actually have performed the structure pointer assignments in those lines in another part of the program based on if/else logic and not necessarily right before line 28. In general the compiler has no easy way of determining this.

The moral is to make sure all your assignments and pointers are set up properly since the compiler is not going to give you a warning or error message for failing to initialize them correctly. Problems of this nature will typically begin to crop up during the execution of your program; more often than not they will be sporadic and very hard to debug. If you provide the extra ounce of prevention during coding, you will avoid many of these situations.

Line 29 really means less than you might think it does. If you take a closer look at the code, you will see that the execution of line 27 allowed line 28 to gain access to s1.s1var. However, this may work properly even if

ps3 and/or ps3->ps2 are not valid pointers. Or I should say it appears to work properly—can you see the problem here?

If we work under the assumption that line 25 was accidentally deleted from the source file, would the program continue to work properly? Perhaps. If it did continue to work, should it have? No. As explained above, it would most certainly compile so that's not the concern here. It would most likely execute under DOS as though nothing were wrong. However, most versions of the XENIX® operating system, as well as OS/2 systems, would produce a general protection fault because of the invalid memory access.

This would occur because ps3 is an external (that is, an external defined in the same source file with no initializer) variable and would therefore be implicitly initialized to zero. If this is the case, then line 26 would be indexing ps2 off a pointer that points at memory location zero. That assignment would then be assigning something to a memory location of 0 + sizeof(int), which may map into memory location 2 and then be treated as the location of ps3->ps2, which we all know is wrong. However, if an access to that location at that moment in time does not stomp on something it shouldn't (and in many cases this is not as clear as the scenario that I'm describing), execution will continue with no apparent damage.

Microsoft® C Optimizing Compiler versions 5.0 and later usually protect against the case above with their infamous R6001 run-time error message upon program termination; however, this is not something you should particularly depend upon, and it shouldn't be relied on to help solve your problems. Furthermore, the R6001 error functions only when your program writes in low core.

Figure 9: Returning a Pointer to an Automatic Structure

```

1  struct temp {
2      int  members;
3      /* ... */
4  };
5
6  struct temp *
7  func()
8  {
9      struct temp temp; /* yes, structure tags *and*
10                          structures can have the
11                          same name */
12
13     return (&temp);
14 }
15
16 main()
17 {
18     struct temp *temp;
19
20     temp = func();
21     /* <expressions using temp->???> */
22 }
23

```

Anything beyond that certain magic number and you're on your own. Therefore, make sure that your pointers are always legitimate regardless of whether or not your program executes properly.

Clearly looks can be deceiving in this case. An excellent example of this scenario is when you merge together different stubs to your program and suddenly something seems to be going wild. Most likely this is due to invalid pointers pointing to locations that cannot be touched without causing a problem. This would occur because your program modules' variables and functions will most likely be in different memory locations as well as in a larger program. When this happens, errors that didn't show up in the stubs will begin to appear.

Given the preceding information, we're now ready to tackle some of the other derivations in **Figure 10**. For instance, if we wanted to access s1inst indirectly through the ps2 pointer, we might use code similar to line 31. If we read this left to right (since both -> and . have equal precedence and left to right associativity), you'll notice that the part referring to ps2.s1inst is in error. Since ps2 is a pointer,

constructing anything with ps2. (note the dot) is not going to work because the dot operator requires that the left operand have a structure type. This structure type must be either an identifier reflecting a structure name, or a dereference of a parenthesized pointer to structure type as in *ps=s; ... (*ps).m...*.

The proper way to do this is shown on line 32, which is the same as line 27, only instead of referencing another pointer (ps1), we're accessing a real structure (s1inst).

You may see another solution to this problem since you should know from your knowledge of C that a structure reference such as *pointer->member* is translatable into *(*pointer).member*. However, how do we implement it in this case? Line 33 seems like a good possibility to resolve this problem (let's not even consider the syntax of line 34) but it does not go far enough. Unfortunately, compiling this will not clue you in any better since most compilers will simply report a syntax error. This isn't immediately intuitive—at least not until you see exactly what's going on behind the scenes.

You need to ask yourself: What exactly is ps2? We know it's a pointer, and we think we

Figure 10: Multilevel Structure Access

```

1  struct s1tag {
2      int    s1var;
3  } s1;
4
5  struct s2tag {
6      int    s2var;
7      struct s1tag *ps1;
8      struct s1tag s1inst;
9  } s2;
10
11 struct s3tag {
12     int    s3var;
13     struct s2tag *ps2;
14     struct s2tag s2inst;
15 } s3;
16
17 struct s3tag *ps3;
18
19 main()
20 {
21     s1.s1var = 99;
22
23     s3.s2inst.s1inst.s1var = 5;
24
25     ps3 = &s3;
26     ps3->ps2 = &s2;
27     ps3->ps2->ps1 = &s1;
28     ps3->ps2->ps1->s1var = -99;
29     printf("s1var=%d\n", s1.s1var);
30
31     /* ps3->ps2.s1inst.s1var = 11; */
32     ps3->ps2->s1inst.s1var = 22;
33     /* ps3->(*ps2).s1inst.s1var = 33; */
34     /* ps3->*ps2.s1inst.s1var = 44; */
35     /* ps2 = 0; */
36     /* might as well have called this 'abccba' */
37     (*ps3->ps2).s1inst.s1var = 55;
38     /* *ps3->ps2.s1inst.s1var = 66; */
39     printf("s3.s2inst.s1inst.s1var=%d\n",
40           s3.s2inst.s1inst.s1var);
41
42     ps3->ps2 = &ps3->s2inst;
43     ps3->ps2->s1inst.s1var = 22;
44     printf("s3.s2inst.s1inst.s1var=%d\n",
45           s3.s2inst.s1inst.s1var);
46 }

```

DEPENDING ON YOUR LOGIC AND PROGRAM DESIGN, YOU WILL ALSO FIND IT WORTHWHILE TO RETURN A POINTER TO A STRUCTURE EVEN IF ONLY ONE OF THE TWO MEMBERS IS GOING TO BE USED.

know its name but in fact its name is not ps2. If it were, you would be able to say something like ps2 = 0; and that is clearly unreasonable (compile it to prove it), since there is no identifier with a name consisting solely of ps2. Note that there is one named s3.ps2 and another that can reference it through pointer notation, as in ps3->ps2. Therefore, these are two names we must refer to in this particular program when accessing ps2.

Tying this back to the (*pointer).member notation discussion above, the correct syntax for line 33 is shown in line 37 since ps3->ps2 is the proper pointer reference to ps2. This in no way infers line 38 is

equivalent to line 37. As discussed in the previous article, this line would produce an error because of the precedence of the * operator.

As a final note, make sure that you understand that lines 32 and 37 do not access the same s1var as in line 38. Again, recall that an s3tag contains a pointer to an s2tag and an instance of an s2tag—they are not the same thing. You could point the s2tag pointer (ps2) to the s2tag instance (s2inst), but if you had wanted to do that, you'd need to code lines 39–42.

Multilevel Structure Access Involving Functions

There are many situations in which you may call functions that return structures, or pointers to structures and the use of temporary variables becomes a nuisance. Consider the function example1 in Figure 11. Is it clear that all of the lines from 27 through 30 will print out 12345?

Line 27 ought to be self-explanatory as a reference to s1.s1var, which has been statically initialized to the value 12345. Line 28 involves a structure analysis exactly like any other, with the same precedence and associativity of operators involved. Do you care that there is a () involved? No, because you memorized the top line of the Operator Precedence/Associativity chart. Therefore since s1returner returns an s1tag type and explicitly returns s1, why not just treat it like line 27? It's simply a structure.member reference.

Line 29 presents a function that returns a pointer to an s1tag, but nothing is that different here even though it does involve a function. If it returns a pointer, access it as a pointer->member reference. Of course line 30 is just a familiar pointer->member case being mapped into (*pointer).member.

Special INTERNATIONAL RATES

Direct international subscriptions are now available for **Microsoft Systems Journal**, the unique publication created especially for the professional software developer. Stay up to date with the newest developments in hardware and software technology.

Microsoft SYSTEMS JOURNAL		
Please check the appropriate boxes.		
COUNTRY	1 YEAR RATE	2 YEAR RATE
Australia *	A\$75	A\$137
Austria *	Sch661	Sch1124
Belgium	BFr1969	BFr3346
Denmark	DKr360	DKr613
Eire	£ 35	£ 60
Finland	FMk223	FMk378
France	FFr319	FFr542
Israel*	IS99	IS181
Italy *	Lir69735	Lir118550
Japan *	¥ 7995	¥ 14657
Netherlands	F106	F180
Norway	NKr345	NKr586
Portugal	Esc7655	Esc13013
Spain	Ptas6165	Ptas10481
Sweden	SKr323	SKr549
Switzerland *	SFr79	SFr134
United Kingdom	£ 29	£ 50
West Germany *	DM94	DM160
Other European *	US\$50	US\$85
Other International *	US\$60	US\$110

Orders accepted at these rates until August 30, 1989

* Return Postage Required

**ENTER MY
SUBSCRIPTION
AT THESE
SPECIAL
RATES!**



Name _____

Title _____

Organization _____

Address _____

Post Code _____

Country _____

Payment enclosed Bill me

Charge my:

Access Visa American Express

Card Number _____

Expiry Date _____

Signature _____

Date _____



NE PAS AFFRANCHIR



NIET FRANKEREN

**REPONSE PAYEE
PAYS-BAS**

Microsoft

c/o KLM Caging/Cashiering Agency
P.O. BOX 10026
2130 CA Hoofddorp
Holland

These variable references are all rather natural. The alternative is to code something along the lines of the statements shown in the example2 function. As you can see, this makes things longer and more tedious than necessary and in this program needn't be used. This point hits home when you realize that you may have a second structure involved, such as s2, and as in the previous section, there are pointers and references to all sorts of variables. This would result in code such as that found in example3 and example4 (or even more complex situations).

Structures and Malloc

There is one other important point to consider. Typically, one must deal with packets of information. In other words, you may find that you're being fed some group of data that is prefixed with control or status information and is then followed by the actual data being described. This would normally occur when dealing with communications, but it needn't occur only there. The problem this presents to the C programmer is that the information part of the packet is finite and predictable, whereas the data portion may be variable in length. For instance, you may find that the following structure is sufficient for describing the control information:

```
struct apacket {
    int    packethead;
    int    packetcontrol1;
    int    packetcontrol2;
    char   data[???];
};
```

But how large do you make the dimension of data[]?

If you make it too small, you could lose data. If you make it too large, you could waste critical space in your program or machine state. Make it reasonably large, and you may not know if that size will be too small for the future—then what do you do? You could add

another field that encodes the size of the data length, then derives some unions and associated code to perform something like the codedrecord example in the previous article. I'm sure we can all agree, however, that that would be a big mess.

The best answer is to avoid the constraints altogether and work with what you have been given. In other words, since you know what the members of the structure that represent the control information are, why not use that to your advantage? This, along with the ability to allocate memory dynamically through standard library routines, such as alloc, calloc, and malloc, is all you need.

A first attempt at a solution might result in a structure template and sample code such as:

```
struct apacket {
    int    packethead;
    int    packetcontrol1;
    int    packetcontrol2;
    int    packetsize;
    char   data[0];
};
< other code >
struct apacket apacket;
struct apacket *ppacket;
ppacket = (struct
    apacket *)malloc(
    sizeof(struct apacket) +
    apacket.packetsize);
```

Unfortunately, C does not allow for zero-length data items to occur, even in structure tags. (Note that some compilers allow this, but they are clearly in error. This is not a portable construct and should be completely avoided.)

Many of you will be quick to point out that making data into a 1-dimensional array with a bound of 1 (for example, char data[1]) should work without a problem (which is true). Note that the 1-byte length will need to be subtracted from the value that's being passed to malloc. This length can be presented as sizeof(char), sizeof(char [1]), or simply the constant 1 since in this case they all represent and refer to the same thing. By the

way, because sizeof can accept a derived type as its argument, the second sizeof says, "give me the size of an array of x characters, where x is 1."

An equal yet slightly less messy approach is to make use of the offsetof macro. Under this circumstance, the structure tag would still contain a char data[1], however instead of

```
ppacket = (struct
    apacket *)malloc(
    sizeof(struct apacket) -
    sizeof(char[1]) +
    apacket.packetsize);
```

you would code:

```
ppacket = (struct
    apacket *)malloc(
    offsetof(apacket, data) +
    apacket.packetsize);
```

since the offset of data within apacket would represent the length of the previous fields (in our case all the fields) in apacket. I feel this is a superior method. With offsetof it's perfectly clear: get the size of the structure and add it to the size desired for the data.

Once you have a pointer of the correct length, you may copy the structure members as appropriate. However, before leaving this subject, let me once again refer you to the March article and encourage you to reread the sections dealing with structure holes and structure packing since you may find that you will need to set up your structures in the same "manner" as the one(s) which you are copying it from.

This scheme of creating dynamic structure images is, as most things, not without problems and requires some careful thought before you use it. For instance, it should be clear that if you use this scheme, every reference to the packet's members will be through a pointer.

This is due to the variable length data that must remain as a single unit. It's important to note that if you do not need a variable length structure, you should

Figure 11: Multilevel Structure Access Involving Functions

```

1  struct s1tag {
2     int   s1var;
3  } s1 = { 12345 };
4
5  struct s2tag {
6     int   s2var;
7     struct s1tag *ps1;
8     struct s1tag s1inst;
9  } s2;
10
11 struct s2tag *ps2;
12
13 struct s1tag
14 s1returner()
15 {
16     return (s1);
17 }
18
19 struct s1tag *
20 ps1returner()
21 {
22     return (&s1);
23 }
24
25 void example1()
26 {
27     printf("%d\n", s1.s1var);
28     printf("%d\n", s1returner().s1var);
29     printf("%d\n", ps1returner()->s1var);
30     printf("%d\n", (*ps1returner()).s1var);
31     printf("%d\n", (ps1returner()->s1var);
32     /* printf("%d\n", (ps1returner()).s1var); */
33 }
34
35 void example2()
36 {
37     struct s1tag s1holder;
38     struct s1tag *ps1holder;
39
40     printf("%d\n", s1.s1var);
41     s1holder = s1returner();
42     printf("%d\n", s1holder.s1var);
43     ps1holder = ps1returner();
44     printf("%d\n", ps1holder->s1var);
45     printf("%d\n", (*ps1holder).s1var);
46 }
47
48 struct s2tag
49 s2returner()
50 {
51     return (s2);
52 }
53
54 struct s2tag *
55 ps2returner()
56 {
57     return (&s2);
58 }
59
60 void example3()
61 {
62     printf("%d\n", s2.s1inst.s1var);
63     printf("%d\n", s2returner().s1inst.s1var);
64     printf("%d\n", ps2returner()->s1inst.s1var);
65 }
66
67 void example4()
68 {
69     printf("%d\n", ps2returner()->ps1->s1var);
70 }
71
72 main()
73 {
74     example1();
75     example2();
76
77     s2.s1inst.s1var = 54321;
78     example3();
79
80     s2.ps1 = &s1;
81     example4();
82 }

```

think about organizing your data in a less elaborate way. For example, you could change the apacket structure tag layout so that data is not an array but a character pointer.

Allowing for this circumstance will usually be far more natural, since apacket instances must be set up (using a char *data; construct) and then referenced. Note that now all the structure members can be accessed directly with the dot operator. The close relationship between arrays and pointers allows data to be referenced in array notation, if desired, as follows:

```

struct apacket somepacket;
somepacket.data =
    (char *) malloc(
        somepacket.packetsize);
<...>
somepacket.data[i] = <...>;
<...>
*somepacket.data = <...>;
/* Note precedence with
   no parens! */
<...>

```

The only ramification of this example is that you will need to free the memory block associated with somepacket.data when you no longer need the data (for dynamically allocated data, the programmer controls the lifetime of the memory block).

Arrays

Since we are on the subject of dynamic memory, it's worth dwelling a bit on arrays and structures. First, look at line 21 of Figure 12. By now I would hope that you wouldn't have much of a problem interpreting it. Nor should you have a problem deciphering lines 24-26, where a pointer to a structure is set equal to an element of an array (which is therefore using only one structure).

The use of arrays (and pointers) implies that their usage and idiosyncrasies remain the same regardless of whether they are used inside structures or as structures (for instance, the syn-

Figure 12: Multilevel Structure Access Involving Arrays

```

1  #define HBOUND(array) (sizeof(array) /
2     sizeof(array[0]))
3
4  struct s1tag {
5     char   chararray[20];
6  };
7
8  struct s1tag s1[10];
9  struct s1tag *ps1;
10
11 main()
12 {
13     int     i;
14     int     j;
15     char   *cp;
16
17     printf("%d/%d=%d\n", sizeof(s1),
18           sizeof(s1[0]), HBOUND(s1));
19
20     for (i = 0; i < HBOUND(s1); i++)
21     for (j = 0; j < sizeof(s1[0].chararray); j++)
22         s1[i].chararray[j] = '\0';
23
24     ps1 = &s1[5];
25     ps1->chararray[2] = 5; /* Note that this is ASCII 5 */
26     (*ps1).chararray[2] = 5; /* not '5' */
27
28     printf("%d\n", sizeof(ps1->chararray));
29     for (ps1 = &s1[0]; ps1 < &s1[HBOUND(s1)];
30         ps1++) {
31         cp = ps1->chararray;
32         while (cp < &ps1->chararray[sizeof(
33             ps1->chararray)])
34             *cp++ = '\0';
35     }
36 }

```

tax and semantics of passing an array to a function does not change because the array might consist of structures rather than a base type like int).

I'd also like to direct your attention to the use of HBOUND that occurs on line 20. It's actually very simple (as lines 17-18 demonstrate) and is very handy when dealing with a good many array situations.

Finally, if you needed to use and access s1 strictly via pointers, an alternate to lines 20-22 can be found in lines 29-35. The latter lines are surely more cryptic; however, they do remove the indexing notation, which would be advantageous if you were to reference other elements of the structure or even repeated occurrences of the same element.

Returning to HBOUND, you should study the constructs on lines 29 and 32-33. The code is checking to see if the pointer has gone beyond the end of the array by checking it against an array dimension that is one greater than the array. In other words, ps1 will be checked against &s1[10], which is one greater than nine (remember that in C arrays start at element 0). Similarly, cp will be checked against &ps1->chararray[20]. Note in particular how all of this occurs without having to explicitly use any constants in either of the two nested loops that occur in this program. You should strive for this type of construction in your own programs, whether you are dealing with structures or not.

Summary

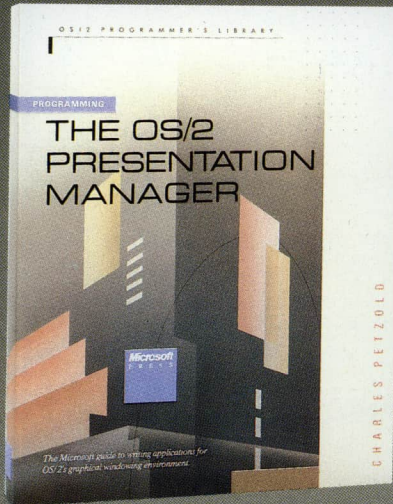
Although this article has focused specifically on structures and the different aspects of having pointers to structures, accessing struct members, avoiding memory conflicts with arrays, and memory allocation, most of these concepts apply to unions as well. With the information presented here, added to

the insights presented in the previous article, you now have a solid base of knowledge concerning the use of structures. When you add to this the insights into unions, typedefs, and C declarations that you have been storing up from the past several issues, you are ready for some serious C programming. □

THE USE OF ARRAYS
(AND POINTERS) IMPLIES
THAT THEIR USAGE AND
IDIOSYNCRASIES REMAIN
THE SAME REGARDLESS
OF WHETHER THEY ARE
USED INSIDE STRUCTURES
OR AS STRUCTURES.

OS/2 and Presentation Manager...

When You Need Reliable Information, Go to the Source.



PROGRAMMING THE OS/2 PRESENTATION MANAGER

Charles Petzold

Here is the first full discussion of the features and operation of the OS/2 1.1 Presentation Manager. If you're developing OS/2 applications, this book will guide you through Presentation Manager's network of windows, messages, and function calls. Petzold includes scores of valuable C programs and utilities. Endorsed by the Microsoft Systems Software group, this book is unparalleled for its clarity, detail, and comprehensiveness. Petzold covers: managing windows ■ handling input and output ■ controlling child windows ■ using bitmaps, icons, pointers, and strings ■ accessing the menu and keyboard accelerators ■ working with dialog boxes ■ understanding dynamic linking ■ and more. An incomparable resource. **\$29.95**

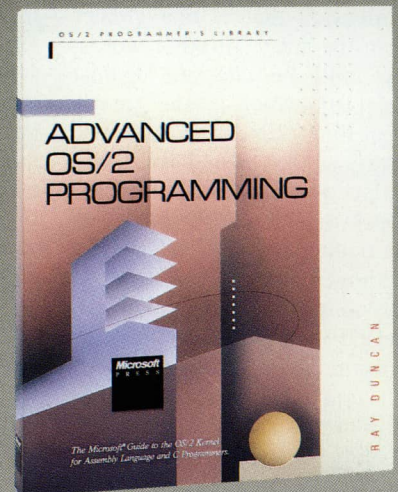
Order Code 86-96791

ADVANCED OS/2 PROGRAMMING

Ray Duncan

Authoritative information, expert advice, and great examples in assembly language and C make this comprehensive overview of OS/2 indispensable to anyone programming the OS/2 kernel. Duncan addresses a broad range of significant OS/2 issues: programming the user interface; mass storage; memory management; multitasking; interprocess communications; customizing filters, device drivers, and monitors; and using OS/2 dynamic link libraries. A valuable reference section includes detailed information on each of the more than 250 system service calls in version 1.1 of the OS/2 kernel. **\$24.95**

Order Code 86-96106



Microsoft® Press
Hardcore Computer Books

Available wherever books and software are sold.

Or place your credit card order by calling 1-800-638-3030 (8AM to 4PM EST). In MD call 824-7300.



MSJ Source Code Listings

All our source code listings can be found on Microsoft OnLine, CompuServe®, GEnie™, and three public access bulletin boards. On the East Coast: RamNet (212) 889-6438 1200/2400 bps. Channel1 (617) 354-8873 1200/2400/9600 bps. On the West Coast: ComOne (415) 284-9151 1200 bps. Look for the MSJ directory. Communications parameters for public access bulletin boards: word length 8, 1 stop bit, full duplex, no parity.

Back copies of *MSJ* can be found in the Information Access Company Computer Database Plus on CompuServe.

Microsoft Corporation assumes no liability for any damages resulting from the use of the information contained herein.

Microsoft, the Microsoft logo, CodeView, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation. UNIX is a registered trademark of American Telephone & Telegraph Company. Macintosh is a registered trademark of Apple Computer, Inc. CompuServe is a registered trademark of CompuServe, Inc. GEnie is a trademark of General Electric Corporation. Intel and MULTIBUS are registered trademarks and 386 is a trademark of Intel Corporation. IBM and AT are registered trademarks of International Business Machines Corporation. Motorola is a registered trademark of Motorola, Inc.



