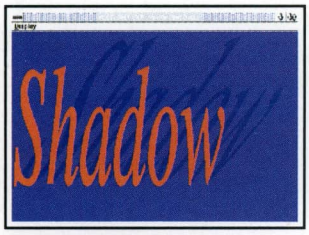
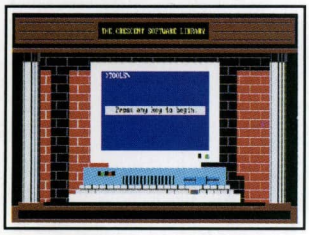
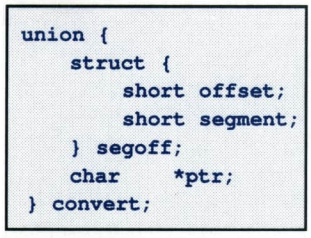
MARCH 1989 • VOL4 NO2 • TEN DOLLARS

Microsoft®

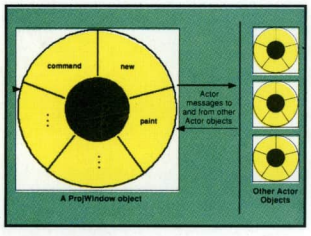
SYSTEMS JOURNAL

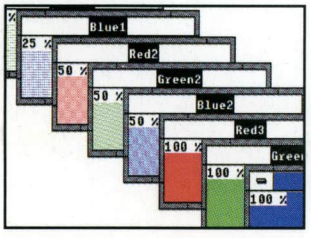






```
union {
    struct {
        short offset;
        short segment;
    } segoff;
    char    *ptr;
} convert;
```



EDITOR'S NOTE

In the September issue of *MSJ*, we challenged you to decipher a convoluted C declaration. See "A Guide to Understanding Even the Most Complex C Declarations," *MSJ* (Vol. 3, No. 5). Recently, we received a call from a company in Cambridge, Mass. asking for the answer to this declaration. It seems the company was sponsoring an in-house programming contest to see who could solve correctly the convoluted example from this article. The winner would be treated to lunch in a cafe within the office building.

Before you go digging for your September issue, or try to figure out which colleague borrowed it and didn't return it, here it is again:

```
unsigned long(far * (far * const(far * far const V[2])[4])())[6];
```

Are we ready to reveal the answer yet? No, we'd rather leave it as a challenge to you, with this bit of advice: in the future we will have a three-part series on C pointers, which you might find helpful.

In this issue, our series on advanced C programming continues with a look at the use of structures, unions, and typedefs to organize data in your code. Charles Petzold's article explores the OS/2 Presentation Manager's (referred to here as PM) vector font capabilities, an integral part of building graphical applications under PM. Vector fonts, which are defined as a series of lines and curves, can be stretched or compressed to any size as well as rotated to any angle. They are device-independent, and therefore not tied to a particular output device resolution. Font controls that were previously restricted to PostScript® printers are now available for other laser printers, output devices (for example plotters), and many different types of video displays.

One concern when building graphical applications under Windows and Presentation Manager is the management of child windows. Kevin Welch provides a thorough discussion of Windows' Multiple Document Interface (MDI), a protocol for managing child windows. The concept of MDI is relatively simple, however, its implementation can be quite difficult. Welch provides the reader with an MDI-based library of function calls, guaranteed to take valuable weeks off the MDI learning curve.

Finally, because serious OS/2 and PM applications cannot be written without a solid understanding of the OS/2 multithreaded programming model and the related use of OS/2 semaphores, we offer an exploration of the special requirements of multithreaded programming. —Ed.

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

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

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

Exploring Vector Fonts with the OS/2 Graphics Programming Interface

01

Charles Petzold

Let's begin with a question: What graphics programming language stores fonts as lines and curves (rather than bitmaps) and thus allows fonts to be arbitrarily stretched, rotated, outlined, filled with different patterns, or even used as clipping areas? One answer is obviously PostScript®, Adobe Systems' page composition language implemented on many high-end laser printers (beginning with the Apple® LaserWriter®) and Allied Corporation's Linotronic® phototypesetters. Over the past few years, PostScript has become the language of choice for computer manipulation of fonts and text.

An equally valid answer is GPI—the Graphics Programming Interface (referred to herein as GPI) component of the OS/2 Presentation Manager. This article shows you how to work with vector fonts in GPI and demonstrates many PostScript-like techniques. As we'll see, GPI has facilities to do virtually everything with fonts that you can do with PostScript. However, GPI does have a deficiency that I will discuss at the end of this article.

Dynamic-Link Library File	Image Fonts	Vector Fonts
COURIER.FON	"Courier" (8, 10, and 12 points for CGA, EGA, VGA, and IBM Proprinter)	"Courier" "Courier Bold" "Courier Italic" "Courier Bold Italic"
HELV.FON	"Helv" (8, 10, 12, 14, 18, and 24 points for CGA, EGA, VGA, and IBM Proprinter)	"Helv" "Helv Bold" "Helv Italic" "Helv Bold Italic"
TIMES.FON	"Tms Rmn" (8, 10, 12, 14, 18, and 24 points for CGA, EGA, VGA, and IBM Proprinter)	"Tms Rmn" "Tms Rmn Bold" "Tms Rmn Italic" "Tms Rmn Bold Italic"

▲ **Figure 1 The OS/2 1.1 dynamic-link library files that contain fonts. The font face names are shown in quotation marks.**

The Trouble with Text

The display of text is always the most problematic part of a graphics programming system. Unlike lines and polygons (which are merely mathematical constructs), text is rooted in a long tradition of aesthetic typography. In any computer graphics system, the goal must always be to display text that is as pleasing and as easy to read as a well-printed book. Yet, most computer output devices (such as video displays and printers) are digital media. The subtly shaped and rounded characters that comprise traditional fonts must be broken down into discrete pixels for storage and then reassembled on the output device. This often causes distortions in the appearance of the text.

One major advantage of using a computer for this job is versatility. We can use a wide variety of fonts in various sizes and characteristics and modify these fonts for display. The extent to which we can modify fonts depends on the way in which the fonts are stored in memory.

Images and Vectors

A font is generally stored in computer (or printer) memory in one of two very different ways. First, a font can be stored as an image or bitmap. Each character of the font is simply a rectangular array of bits. The 0 bits generally correspond to the background around the character and the 1 bits correspond to the character itself. Second, a font can be stored in a vector or outline format

Charles Petzold is the author of Programming Windows (Microsoft Press, 1988) and Programming the OS/2 Presentation Manager (Microsoft Press, 1988). A copy of the latter is included in Microsoft's OS/2 Software Development Kit.

02 in which each character is defined as a series of lines and curves that enclose areas. The character is displayed by drawing the outline on the output device and filling in the enclosed areas.

Image and vector fonts have distinct advantages and disadvantages. Image fonts are always created for specific font sizes and specific device resolutions. The size of a particular image font cannot easily be changed. (For example, enlarging an image font by doubling the rows and columns of pixels often emphasizes the jaggedness of the characters.) Also, image fonts cannot be rotated except possibly by 90 degree increments.

Vector fonts are much more malleable. Because they are defined as a series of lines and curves, vector fonts can be stretched or compressed to any size and can be rotated to any angle. Vector fonts are not tied to a particular output device resolution.

In general, however, image fonts are more legible than vector fonts. Various techniques are used to design image fonts so they fool the eye into thinking the characters are smoother than they actually are. Vector fonts—particularly when displayed on low-resolution devices and scaled to small font sizes—can be adjusted only by mathematical algorithms, which currently are less capable than human font designers. Another advantage of image fonts is performance since vector fonts usually require much more processing time to draw each character.

Most conventional laser printers store fonts as images, either within the printer or in font cartridges. The printer is restricted to specific font sizes and the characters cannot be arbitrarily rotated. Much more versatile are the fonts stored in PostScript-based printers. These fonts are stored as vectors. PostScript fonts can be stretched or compressed to any size, they can be arbitrarily rotated, filled with various patterns, and used for clipping.

Figure 2: Code Fragment from VECTFONT.C

```
                                    :
case WM_CREATE:
     hdc = WinOpenWindowDC (hwnd) ;

                              // Create PS use Twips page units
     sizl.cx = 0 ;
     sizl.cy = 0 ;
     hps = GpiCreatePS (hab, hdc, &sizl,
                              PU_TWIPS      | GPIF_DEFAULT |
                              GPIT_MICRO    | GPIA_ASSOC) ;

                              // Adjust Page Viewport for points

     GpiQueryPageViewport (hps, &rcl) ;
     rcl.xRight *= 20 ;
     rcl.yTop   *= 20 ;
     GpiSetPageViewport (hps, &rcl) ;

     hwndMenu = WinWindowFromID (
                         WinQueryWindow (hwnd, QW_PARENT,
                                         FALSE), FID_MENU) ;
     return 0 ;

case WM_SIZE:
     ptlClient.x = SHORT1FROMMP (mp2) ;       // client width
     ptlClient.y = SHORT2FROMMP (mp2) ;       // client height

     GpiConvert (hps, CVTC_DEVICE, CVTC_PAGE, 1L, &ptlClient);
     return 0 ;
                                    :
```

The GPI Fonts

GPI can, of course, take advantage of fonts that are stored in and supported by output devices such as laser printers. But it also includes its own support of both image and vector fonts. The image fonts are expected because they are particularly suited for low-resolution video displays and dot matrix printers. Image fonts are an important part of most graphics programming systems (such as Microsoft Windows GDI).

The addition of vector fonts in GPI is a real treat. GPI can use these vector fonts with any output device. Thus, various font techniques that previously have been restricted to PostScript printers are now possible with other laser printers and even the video display.

OS/2 Version 1.1 is shipped with three resource-only dynamic-link libraries with FON extensions. These are font files and the contents are shown in **Figure 1**. In addition, the video display (DISPLAY.DLL) and printer device drivers may also contain fonts designed specifically for the device. For example, the default System Proportional font is stored in DISPLAY.DLL.

If you want to use any of the fonts in the FON files, you must install the fonts from the Presentation Manager Control Panel program. It is only necessary to install one font from each of the three files, and you only need do this once.

Each font has a face name, which is shown in quotation marks in **Figure 1**. Each of the image fonts is available in several point sizes and for several output devices: the CGA, EGA, VGA (and 8514/A), and IBM® Proprinter. GPI can synthesize variations of these fonts, such as italic or boldfaced versions. Vector fonts, however, need not be designed for a particular output device and point size because they can be arbitrary scaled. You'll note that italic and boldface versions of the vector fonts are also included.

The vector fonts in GPI are similar in style to the Courier, Helvetica®, and Times® fonts included in most PostScript printers.

A little exploration of the font files will reveal that vector fonts are encoded as a series of GPI drawing orders. When drawing text with these fonts, GPI translates the drawing orders into GPI functions, usually GpiPolyLine to draw straight lines and GpiPolyFilletSharp to draw curves.

The VECTFONT Program

The VECTFONT program demonstrates the use of GPI vector fonts. For purposes of clarity, I've divided the program into several modules. The files that comprise the basic shell of the

```
/*-------------------
   VF01.C — Display 24-point vector fonts
  -------------------*/

#define INCL_GPI
#include <os2.h>
#include <string.h>
#include "vectfont.h"

VOID Display_24Point (HPS hps, LONG cxClient, LONG cyClient)
     {
     static CHAR *szFacename[] = {
                                  "Courier",       "Courier Italic",
                                  "Courier Bold",  "Courier Bold Italic",
                                  "Tms Rmn",       "Tms Rmn Italic",
                                  "Tms Rmn Bold",  "Tms Rmn Bold Italic",
                                  "Helv",          "Helv Italic",
                                  "Helv Bold",     "Helv Bold Italic"
                                  } ;
     static INT  iNumFonts = sizeof szFacename / sizeof szFacename[0] ;
     FONTMETRICS fm ;
     INT         iFont ;
     POINTL      ptl ;

     ptl.x = cxClient / 8 ;
     ptl.y = cyClient ;

     for (iFont = 0 ; iFont < iNumFonts ; iFont++)
          {
                                  // Create font, select it and scale

          CreateVectorFont (hps, LCID_MYFONT, szFacename[iFont]) ;
          GpiSetCharSet (hps, LCID_MYFONT) ;
          ScaleVectorFont (hps, 240, 240) ;

                                  // Get font metrics for scaled font

          GpiQueryFontMetrics (hps, (LONG) sizeof (FONTMETRICS), &fm) ;
          ptl.y -= fm.lMaxBaselineExt ;

                                  // Display the font facename

          GpiCharStringAt (hps, &ptl, (LONG) strlen (szFacename[iFont]),
                           szFacename[iFont]) ;

          GpiCharString (hps, 10L, " - abcdefg") ;

          GpiSetCharSet (hps, LCID_DEFAULT) ;      // Clean up
          GpiDeleteSetId (hps, LCID_MYFONT) ;
          }
     }
```

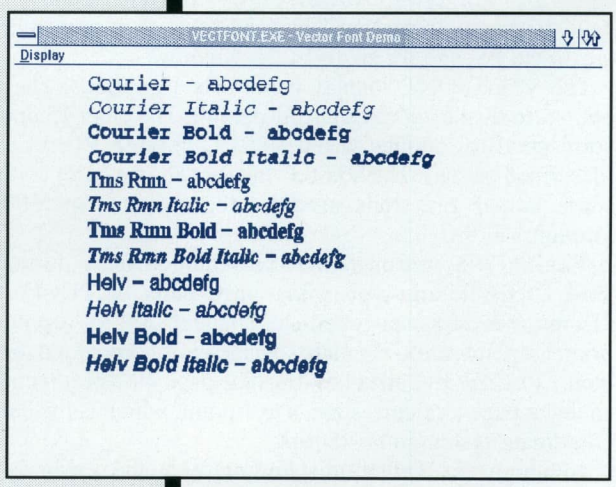


Figure 3 The GPI vector fonts in 24-point size, displayed by selecting 24 Point Fonts from the VECTFONT menu.

Figure 4 A vector font strectched to fill the client window, displayed by selecting Stretched Font from the VECTFONT menu.

```
/*-----------------------------------------------------------
   VF02.C -- Display vector font stretched to client window
  -----------------------------------------------------------*/

#define INCL_GPI
#include <os2.h>
#include "vectfont.h"

VOID Display_Stretch (HPS hps, LONG cxClient, LONG cyClient)
     {
     static CHAR szText[] = "Hello!" ;
     static LONG cbText = sizeof szText - 1 ;
     POINTL      ptl ;

                              // Create font, select, and scale

     CreateVectorFont (hps, LCID_MYFONT, "Tms Rmn Italic") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;
     ScaleFontToBox (hps, cbText, szText, cxClient, cyClient) ;
     QueryStartPointInTextBox (hps, cbText, szText, &ptl) ;

     GpiCharStringAt (hps, &ptl, cbText, szText) ;      // Display text

     GpiSetCharSet (hps, LCID_DEFAULT) ;                // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;
     }
```

program are VECTFONT, VECTFONT.LNK, VECTFONT.DEF, VECTFONT.H, and VECTFONT.C. (*These files are not shown here, but can be downloaded from any of our bulletin boards—Ed.*) **Figure 2** shows the code fragment from VECTFONT.C that is responsible for creating a PS, and for sizing client windows.

The VECTFONT Display menu lists 16 options. The first option (to display nothing) is the default. The other 15 options correspond to routines in the VF01.C through VF15.C files (described below). The VF00.C file (not shown here) contains some helper functions used by the routines in VF01.C through VF15.C.

VECTFONT creates a micro presentation space during the WM_CREATE message using page units of PU_TWIPS. (Twips is a fabricated word standing for 20th of a point. A printer's point size is $^1/_{72}$ inch, so page units correspond to $^1/_{1440}$ inch.) VECTFONT then modifies the page viewport rectangle so that a page unit corresponds to 1 point, which is the default coordinate system in PostScript.

Although VECTFONT displays output to the screen, vector fonts are obviously better suited for laser printers. As you will see, the appearance of the fonts—even at 24 point sizes—is not nearly as good as the image fonts.

You will also notice that several of the demonstration routines in VECTFONT require a few seconds to run. For anything other than a demonstration program, you would probably want to use a second thread of execution to avoid holding up message processing.

Selecting an Outline Font

To use an outline font in a Presentation Manager program, you must first create a logical font and then select the font into your presentation space. The GpiCreateLogFont function creates a logical font and associates the font with a local ID (a number between 1L and 254L that you select). This function requires a pointer to a structure of type FATTRS (font attributes) that specifies the attributes of the font you want.

To create a vector font, most of the fields in this structure can be set to zero. The most important fields are szFacename (which is set to the face name of the font, one of the names in the last column of **Figure 1**) and fsFontUse, which is set to the constant identifiers FATTR_FONTUSE_OUTLINE and FATTR_FONTUSE_TRANSFORMABLE, combined with the C bitwise OR operator.

You may prefer using the CreateVectorFont function in VF00.C to create a vector font. This function requires only the presentation space handle, the local ID, and the face name:

```
CreateVectorFont(hps, lcid, szFacename);
```

After you create a logical font (using either GpiCreateLogFont or CreateVectorFont), you can select the font into the presentation space:

```
GpiSetCharSet(hps, lcid);
```

The lcid parameter is the local ID for the font. After the font is selected in the presentation space, you can alter the attributes of the font with various functions described below, obtain information about the font by calling GpiQueryFontMetrics, GpiQueryWidthTable, and GpiQueryTextBox; and use the font for text output with one of the text functions such as GpiCharStringAt.

```
/*--------------------------------------------------
   VF03.C -- Display four strings in mirror reflections
  --------------------------------------------------*/

#define INCL_GPI
#include <os2.h>
#include "vectfont.h"

VOID Display_Mirror (HPS hps, LONG cxClient, LONG cyClient)
     {
     static CHAR szText[] = "Mirror" ;
     static LONG cbText = sizeof szText - 1 ;
     INT         i ;
     POINTL      ptl ;
     SIZEF       sizfx ;
                                 // Create font, select and scale

     CreateVectorFont (hps, LCID_MYFONT, "Tms Rmn Italic") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;
     ScaleFontToBox (hps, cbText, szText, cxClient / 2, cyClient / 2) ;

     ptl.x = cxClient / 2 ;          // Center of client window
     ptl.y = cyClient / 2 ;

     for (i = 0 ; i < 4 ; i++)
          {
          GpiQueryCharBox (hps, &sizfx) ;

          if (i == 1 || i == 3)
               sizfx.cx *= -1 ;
                                 // Negate char box dimensions
          if (i == 2)
               sizfx.cy *= -1 ;

          GpiSetCharBox (hps, &sizfx) ;

          GpiCharStringAt (hps, &ptl, cbText, szText) ;
          }
     GpiSetCharSet (hps, LCID_DEFAULT) ;          // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;

     }
```

Figure 5 Vector font with positive and negative character box values, displayed by selecting Mirrored Font from the VECTFONT menu.

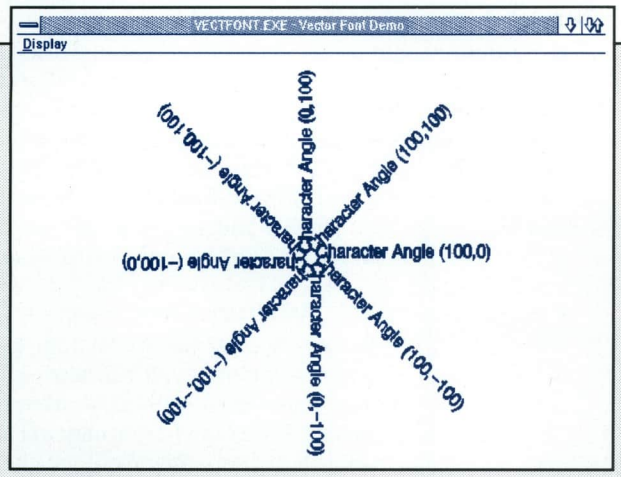


Figure 6
Vector fonts at eight character angles, displayed by selecting Character Angles from the VECTFONT menu.

```
/*-----------------------------------------
   VF04.C -- Display eight character angles
  -----------------------------------------*/

#define INCL_GPI
#include <os2.h>
#include <stdio.h>
#include "vectfont.h"

VOID Display_CharAngle (HPS hps, LONG cxClient, LONG cyClient)
     {
     static GRADIENTL agradl[8] = { 100,    0,  100,  100,
                                      0,  100, -100,  100,
                                   -100,    0, -100, -100,
                                      0, -100,  100, -100 } ;
     CHAR             szBuffer[40] ;
     INT              iIndex ;
     POINTL           ptl ;

                           // Create Helvetica font

     CreateVectorFont (hps, LCID_MYFONT, "Helv") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;
     ScaleVectorFont (hps, 200, 200) ;

     ptl.x = cxClient / 2 ;   // Center of client window
     ptl.y = cyClient / 2 ;

     for (iIndex = 0 ; iIndex < 8 ; iIndex++)
          {
          GpiSetCharAngle (hps, agradl + iIndex) ;      // Char angle

          GpiCharStringAt (hps, &ptl,
               (LONG) sprintf (szBuffer, " Character Angle (%ld,%ld)",
                               agradl[iIndex].x, agradl[iIndex].y),
               szBuffer) ;
          }
     GpiSetCharSet (hps, LCID_DEFAULT) ;               // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;
     }
```

When you no longer need the font, you reselect the default font into the presentation space:

```
GpiSetCharSet(hps, LCID_DEFAULT);
```

and then delete the local ID associated with the font:

```
GpiDeleteSetId(hps, lcid);
```

In VECTFONT I always use the identifier LCID_MYFONT for the local ID. This is defined in VECTFONT.H as 1L.

Scaling to a Point Size

When you call GpiSetCharSet to select a vector font into the presentation space, the initial width and height of the font are based on the GPI character box, which defines a character width and height in page units. The default character box is based on the size of the default system font. You can change the character box size by calling GpiSetCharBox.

An important point to note is that to get a correctly proportioned vector font, you must change the character box size. Do not use the default. Generally you set the height of the character box to the desired height of the font. If you want a vector font to have a normal width, set the width of the character box equal to the height. For a skinnier font, set the width of the character box less than the height; for a fatter font, set the width greater than the height.

If you're working in page units of PU_PELS, you must also adjust the character box dimensions to account for differences in horizontal and vertical resolution of the output device. For this reason, it's much easier to work with vector fonts if you use one of the metric page units (PU_LOENGLISH, PU_HIENGLISH, PU_LOMETRIC, PU_HIMETRIC, or PU_TWIPS). With these page units, horizontal and vertical page units are the same. For example, suppose you're using page units of PU_TWIPS. This means that one page unit is equal to $^1/_{20}$ of a point or $^1/_{1440}$ inch. After selecting a vector font into the presentation space, you want to scale the font to 24 points. You first define a structure of type SIZEF:

```
SIZEF sizfx;
```

The two fields of this structure, named cx and cy, are interpreted as 32-bit FIXED numbers; that is, the high 16 bits are interpreted as an integer, and the low 16 bits are interpreted as a fraction.

If you want to scale the vector font to a 24-point height, you can use the MAKEFIXED macro to set to the fields of the structure like this:

```
sizfx.cx = MAKEFIXED(24 * 20, 0);
sizfx.cy = MAKEFIXED(24 * 20, 0);
```

Multiplying by 20 is necessary to convert the point size you want to twips. Then call GpiSetCharBox:

```
GpiSetCharBox(hps, &sizfx);
```

After setting the character box, any character or text dimensions you obtain from GpiQueryFontMetrics, GpiQueryTextBox, and GpiQueryWidthTable will reflect the new font size.

The ScaleVectorFont routine in VF00.C can help in scaling

06

a vector font to a desired point size. The function will work with any page units, even PU_PELS. The second and third parameters to ScaleVectorFont specify a point size in units of 0.1 points. (For example, use 240 for a 24-point size.) If you want a normally proportioned vector font, set the third parameter to ScaleVectorFont equal to the second parameter.

The VF01.C file in **Figure 3** shows how the functions discussed so far can be used to display all the available vector fonts in 24-point size. You can run the function in VF01.C by selecting the 24 Point Fonts option from the VECTFONT Display menu. The source code and its results are shown in **Figure 3**.

```
/*-----------------------------------------
   VF05.C -- Display "Hello, World" in circle
  -----------------------------------------*/

#define INCL_GPI
#include <os2.h>
#include <math.h>
#include <stdlib.h>
#include "vectfont.h"

VOID Display_Rotate (HPS hps, LONG cxClient, LONG cyClient)
     {
     static CHAR szText[] = "Hello, world! " ;
     static LONG cbText = sizeof szText - 1L ;
     static LONG alWidthTable[256] ;
     double      ang, angCharWidth, angChar ;
     FONTMETRICS fm ;
     GRADIENTL   gradl ;
     INT         iChar ;
     LONG        lCircum, lRadius, lTotWidth, lCharRadius, cyChar ;
     POINTL      ptl ;

                         // Create the font and get font metrics

     CreateVectorFont (hps, LCID_MYFONT, "Tms Rmn") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;

     GpiQueryFontMetrics (hps, (LONG) sizeof fm, &fm) ;

                         // Find circle dimensions and scale font

     lRadius = min (cxClient / 4, cyClient / 4) ;
     lCircum = (LONG) (2 * PI * lRadius) ;
     cyChar  = fm.lMaxBaselineExt * lRadius / fm.lMaxAscender ;

     ScaleFontToBox (hps, cbText, szText, lCircum, cyChar) ;

                         // Obtain width table and total width

     GpiQueryWidthTable (hps, 0L, 256L, alWidthTable) ;

     for (lTotWidth = 0, iChar = 0 ; iChar < (INT) cbText ; iChar ++)
          lTotWidth += alWidthTable [szText [iChar]] ;

     ang = PI / 2 ;      // Initial angle for first character

     for (iChar = 0 ; iChar < (INT) cbText ; iChar++)
          {
                         // Set character angle

          angCharWidth = 2 * PI * alWidthTable [szText [iChar]] / lTotWidth ;

          gradl.x = (LONG) (lRadius * cos (ang - angCharWidth / 2 - PI / 2)) ;
          gradl.y = (LONG) (lRadius * sin (ang - angCharWidth / 2 - PI / 2)) ;

          GpiSetCharAngle (hps, &gradl) ;

                         // Find position for character and display it

          angChar = atan2 ((double) alWidthTable [szText [iChar]] / 2,
                           (double) lRadius) ;

          lCharRadius = (LONG) (lRadius / cos (angChar)) ;
          angChar += ang - angCharWidth / 2 ;

          ptl.x = (LONG) (cxClient / 2 + lCharRadius * cos (angChar)) ;
          ptl.y = (LONG) (cyClient / 2 + lCharRadius * sin (angChar)) ;

          GpiCharStringAt (hps, &ptl, 1L, szText + iChar) ;

          ang -= angCharWidth ;
          }
     GpiSetCharSet (hps, LCID_DEFAULT) ;          // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;
     }
```

Arbitrary Stretching

Besides scaling the vector font to a specific point size, you can also scale vector fonts to fit within an arbitrary rectangle. For example, you may want to scale a short text string to fill the client window.

The function shown in VF02.C (see **Figure 4**) shows how this is done. You can run this function by selecting Stretched Font from VECTFONT's menu. The function in VF02.C displays the word "Hello!" in the Tms Rmn Italic font stretched to the size of the client window.

The ScaleFontToBox function in VF00.C helps out with this job. This function first calls GpiQueryTextBox to obtain the coordinates of a parallelogram that encompasses the text string. The character box is then scaled based on the size of this text box and the rectangle to which the font must be stretched. The QueryStartPointInTextBox function in VF00.C determines the starting point of the text string (that is, the point at the baseline of the left side of the first character) within this rectangle.

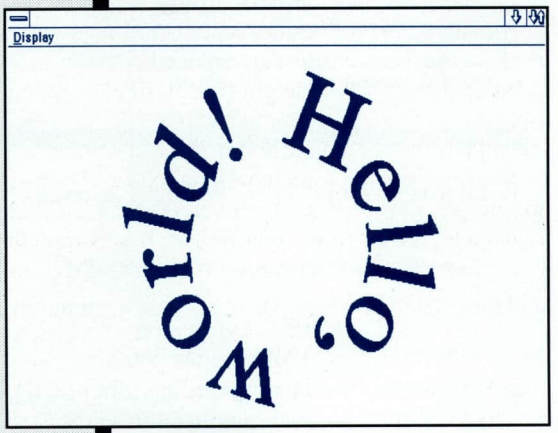
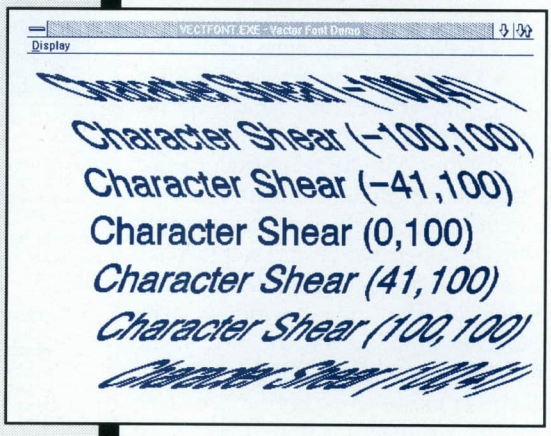


Figure 7 Character string displayed around the perimeter of a circle, displayed by selecting Rotated Font from the VECTFONT menu.

```
/*-------------------------------------------------------
   VF06.C -- Display seven different character shear angles
  -------------------------------------------------------*/

#define INCL_GPI
#include <os2.h>
#include <stdio.h>
#include "vectfont.h"

VOID Display_CharShear (HPS hps, LONG cxClient, LONG cyClient)
     {
     static POINTL aptlShear[7] = { -100,  41, -100, 100,
                                    -41, 100,    0, 100,
                                     41, 100,  100, 100,
                                    100,  41 } ;
     CHAR          szBuffer[40] ;
     FONTMETRICS   fm ;
     INT           iIndex ;
     POINTL        ptl ;

                    // Create and scale Helvetica font

     CreateVectorFont (hps, LCID_MYFONT, "Helv") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;
     ScaleVectorFont (hps, 480, 480) ;

                    // Get font metrics for scaled font

     GpiQueryFontMetrics (hps, (LONG) sizeof (FONTMETRICS), &fm) ;

     ptl.x = cxClient / 8 ;
     ptl.y = cyClient ;

     for (iIndex = 0 ; iIndex < 7 ; iIndex++)
          {
          GpiSetCharShear (hps, aptlShear + iIndex) ;  // Char shear

          ptl.y -= fm.lMaxBaselineExt ;

          GpiCharStringAt (hps, &ptl,
               (LONG) sprintf (szBuffer, "Character Shear (%ld,%ld)",
                         aptlShear[iIndex].x, aptlShear[iIndex].y),
               szBuffer) ;
          }
     GpiSetCharSet (hps, LCID_DEFAULT) ;              // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;
     }
```

Figure 8 There is nothing wrong with your monitor. These examples of character shear are displayed by selecting Character Shear from the VECTFONT menu.



This point is used in the GpiCharStringAt function to display the text.

Mirror Images

The character box, besides scaling the font to a particular size, can also flip the characters around the horizontal or vertical axis.

If the character box height (the cy field of the SIZEF structure) is negative, the characters will be flipped around the baseline and displayed upside down. If the character box width (the cx field) is negative, the individual characters will be flipped around the vertical axis. Moreover, GpiCharStringAt will draw a character string from right to left. It's as though the whole character string is flipped around the vertical line at the left side of the first character.

The function in VF03.C (see **Figure 5**) displays the same string four times, using all possible combinations of negative and positive character box dimensions. You can run this function by selecting Mirrored Font from the VECTFONT menu.

Transformations

Unlike image fonts, vector fonts can be scaled to any size and sheared and rotated to any angle. The matrix transforms described in my article "OS/2 Graphics Programming Interface: An Introduction to Coordinate Spaces," *MSJ* (Vol. 3, No. 4), affect vector fonts in the same way they affect the display of other graphics primitives.

In addition, GPI also supports several functions specifically for performing transforms on vector fonts. We've already seen how the GpiSetCharBox function allows font characters to be scaled. The GpiSetCharAngle rotates the font characters and the GpiSetCharShear performs *x* shear.

Character Angle and Rotation

By default, the baseline of the vector font characters is parallel to the *x* axis in world coordinates. You can change this by calling GpiSetCharAngle. This rotates the vector font's characters.

The character angle is specified using a GRADIENTL structure, which has two fields named *x* and *y* of type LONG. Imagine a line connecting the point (0,0) to the point (*x*,*y*) in world coordinates. The baseline of each character is parallel to this line. The direction of the text is the same as the direction from (0,0) to (*x*,*y*).

You can also think of this in trigonometric terms. The baseline of the text is parallel to a line at angle α measured counterclockwise from the *x* axis, where:

$$\alpha = \arctan(y/x)$$

and *y* and *x* are the two fields of the GRADIENTL structure.

The absolute magnitudes of *y* and *x* are not important. What's important are the relative magnitudes and signs. The signs of *x* and *y* determine the direction of the text string as indicated in the

following table:

x	y	Direction
+	+	To upper right
-	+	To upper left
-	-	To lower left
+	-	To lower right

The function in VF04.C (see **Figure 6**) uses the GpiSetCharAngle function to display eight text strings at 45 degree increments around the center of the client window. Each string displays the fields of the GRADIENTL structure that is used in the GpiSetCharAngle call.

In this example, the text strings begin with a blank character so as not to make a mess of overlapping characters in the center of the client window. The character angle does not affect the interpretation of the starting position of the string specified in the GpiCharStringAt function. If you move your head so that a particular string is seen as running left to right, the starting position still refers to the point at the baseline of the left side of the first character.

You can make the characters in a text string follow a curved path by individually calculating the starting position angle of each character and displaying the characters one at a time. This is what is done in VF05.C (see **Figure 7**) to display "Hello, World!" around the perimeter of a circle.

The text string is scaled based on the circumference of a circle that is positioned in the center of the window and has a diameter half the width or height (whichever is less) of the window. The GpiQueryWidthTable function is used to obtain the width of individual characters and then space them around the circle.

Figure 9 Character shear used to draw a shadow of a character string, displayed by selecting Font with Shadow from the VECTFONT menu.

```
/*------------------------------------------------
   VF07.C -- Display characters with sheared shadow
  ------------------------------------------------*/

#define INCL_GPI
#include <os2.h>
#include "vectfont.h"

VOID Display_Shadow (HPS hps, LONG cxClient, LONG cyClient)
     {
     static CHAR szText[] = "Shadow" ;
     static LONG cbText = sizeof szText - 1 ;
     POINTL      ptl, ptlShear ;
     SIZEF       sizfx ;

     CreateVectorFont (hps, LCID_MYFONT, "Tms Rmn Italic") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;
     ScaleFontToBox (hps, cbText, szText, 3 * cxClient / 4, cyClient) ;
     QueryStartPointInTextBox (hps, cbText, szText, &ptl) ;

     ColorClient (hps, cxClient, cyClient, CLR_BLUE) ;

     GpiSavePS (hps) ;

     ptlShear.x = 200 ;                              // Set char shear
     ptlShear.y = 100 ;
     GpiSetCharShear (hps, &ptlShear) ;

     GpiQueryCharBox (hps, &sizfx) ;
     sizfx.cy += sizfx.cy / 4 ;                      // Set char box
     GpiSetCharBox (hps, &sizfx) ;

     GpiSetColor (hps, CLR_DARKBLUE) ;
     GpiCharStringAt (hps, &ptl, cbText, szText) ;  // Display shadow

     GpiRestorePS (hps, -1L) ;

     GpiSetColor (hps, CLR_RED) ;
     GpiCharStringAt (hps, &ptl, cbText, szText) ;  // Display text

     GpiSetCharSet (hps, LCID_DEFAULT) ;             // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;
     }
```

Character Shear

It is easy to confuse the character angle with the character shear. Let's look at the difference. The character angle refers to the orientation of the baseline. As you can see in **Figures 6** and **7**, text displayed with various character angles is rotated but otherwise not distorted in any way.

The character shear affects the appearance of the characters themselves apart from any rotation. Character shear by itself bends characters to the left or right, but the bottom of each character remains parallel to the *x* axis. You can use character shear to create oblique (sometimes mistakenly called italic) versions of a font.

To set character shear you call the GpiSetCharShear function. This function requires a pointer to a structure of type POINTL, which has two fields named *x* and *y*. Imagine a line drawn from (0,0) to the point (*x*,*y*) in world coordinates. The left and right sides of each character are parallel to this line.

The function shown in in VF06.C (see **Figure 8**) displays seven text strings using different character shears. You can run this function by selecting Character Shear from the VECTFONT menu. Each string displays the *x* and *y* values used in the POINTL structure to set the character shear.

The character shear is governed by the relative magnitudes and signs of the *x* and *y* fields of the POINTL structure. If the signs of both fields are the same, then the characters tilt to the right; if they are different, the characters tilt to the left. The character shear does not flip the characters upside down. For

example, character shear using the point (100,100) has the same effect as (-100,-100).

The angle of the left and right sides of the characters from the *y* axis is sometimes called the shear angle. In theory, the shear angle can range to just above -180 degrees (infinite left shear) to just under +180 degrees (infinite right shear) and is equal to

```
α = arctan (x/y)
```

where *x* and *y* are the two fields of the POINTL structure.

When you set a nondefault character shear, the GpiQueryTextBox function returns an array of points that define a parallelogram rather than a rectangle. However, the top and bottom sides of this text box are the same width as for a nonsheared text string, and the distance between the top and bottom sides also remains the same.

You can use character shear to draw an oblique shadow of a text string. The function in VF07.C (see **Figure 9**) colors the background of the client window blue and displays the text string Shadow twice.

The first call to GpiCharStringAt displays the shadow. This is drawn in dark blue using a positive character shear. The second call to GpiCharStringAt draws the characters upright in red with a slightly smaller character box height. You can run this function by selecting Font with Shadow from the VECTFONT menu.

A Primer on Paths

To explore more capabilities of vector fonts, it's necessary to become familiar with the GPI path, which is similar to a PostScript path. In GPI, you create a path by calling line drawing functions between calls to the GpiBeginPath and GpiEndPath functions:

```
GpiBeginPath(hps, idPath);
    <... call line drawing functions ... >
GpiEndPath (hps) ;
```

This is called a path bracket. In OS/2 1.1, idPath must be set equal to 1L. The functions that are valid within a path bracket are listed in the documentation of the Presentation Manager functions.

The functions you call within the path bracket do not draw anything. Instead, the lines that make up the path are retained by the system. Often the lines you draw in a path will enclose areas, but they don't have to.

After the GpiEndPath call, you can do one of three things with the path you've created:

- Call GpiStrokePath to draw the lines that comprise the path. These lines are drawn using the geometric line width, joins, and ends (discussed shortly).
- Call GpiFillPath to fill enclosed areas defined by the path. Any open areas are automatically closed. The area is filled with the current pattern.
- Call GpiSetClipPath to make the enclosed areas of the path a clipping area. Any open areas are automatically closed. Subsequent GPI calls will only display output within the enclosed area defined by the path.

Each of these three functions causes the path to be deleted. Prior to calling any of these three functions, you can call GpiModifyPath, which I'll describe toward the end of this article.

Normally, GpiCharStringAt and the other text output functions are not valid in a path bracket. The exception is when a vector font is selected in the presentation space. When called from within a path bracket, GpiCharStringAt does not draw the text string. Instead, the outlines of the characters become part of the path.

Paths opens up a whole collection of PostScript-like stylistic techniques that you can use with vector fonts.

Hollow Characters

Let's begin by calling GpiCharStringAt in a path bracket and then use GpiStrokePath to draw the lines of the path. GpiStrokePath has the following syntax:

```
GpiStrokePath(hps, idPath, 0L);
```

In the initial version of the Presentation Manager, the last parameter of the function must be set to 0L. When used to stroke a path created by calling GpiCharStringAt, only the outline is drawn and not the interiors. This creates hollow characters.

The function in VF08.C (see **Figure 10**) uses this technique to display the outline of the characters in the text string Hollow. You can run this function by selecting Hollow Font from the VECTFONT menu.

You may want to display characters in one color with an outline of another color. In this case, you must call GpiCharStringAt twice, first to draw the interior in a specific color and second, in a path bracket followed by a GpiStrokePath call to draw the outline. The function in VF09.C (see **Figure 11**) does something like this to draw text with a drop shadow.

The shadow is drawn first using a normal GpiCharStringAt call. Another GpiCharStringAt function draws the text again in the current window background color at a 1/6-inch offset to the first string. This is surrounded by an outline created by a third GpiCharStringAt call in a path bracket followed by GpiStrokePath.

Quite similar to this is the creation of characters that look like solid blocks, as shown in the function in VF10.C (see **Figure12**). Eighteen character strings are drawn at 1-point offsets using CLR_DARKGREEN. This is capped by the character string drawn again in CLR_GREEN and the border in CLR_DARKGREEN.

Filling the Path

When first introducing paths, I mentioned that you can do one of three things with a path: stroke it, fill it, or use it for clipping. Let's move on to the second; filling the path.

To fill a path, you call:

```
GpiFillPath(hps, idPath, lOption);
```

This fills the path with the current pattern. Any open areas of the path are automatically closed before being filled. The lOption parameter can be either FPATH_ALTERNATE or FPATH_WINDING to fill the path using alternate or winding modes. For vector fonts, FPATH_WINDING causes the

10 interiors of some letters (such as O) to be filled. You'll probably want to use FPATH_ALTERNATE instead.

If the current area filling pattern is PATSYM_SOLID, the code

```
GpiBeginPath(hps, idPath);
GpiCharStringAt(hps, &ptl,
                cch, szText);
GpiEndPath(hps);
GpiFillPath(hps,idPath,
            FPATH_ALTERNATE);
```

does roughly the same thing with a vector font as does a GpiCharStringAt by itself. When using GpiFillPath you will want to set a pattern other than PATSYM_SOLID. (A bug in OS/2 1.1 causes the current pattern to be reset to PATSYM_SOLID during a path bracket in which GpiCharStringAt is called. You can get around this bug by calling GpiSetPattern after you end the path.)

The function in VF11.C (see **Figure 13**) uses GpiFillPath to display the text string Fade eight times filled with the eight GPI shading patterns (PATSYM_DENSE1 through PATSYM_DENSE8) and then finishes by calling GpiCharStringAt outside of a path bracket. You can run this function by selecting Fading Font from the VECTFONT menu.

Geometrically Thick Lines

At first, it may seem as though there is no difference between drawing a line

Figure 10 Hollow characters, displayed by selecting Hollow Font from the VECTFONT menu.

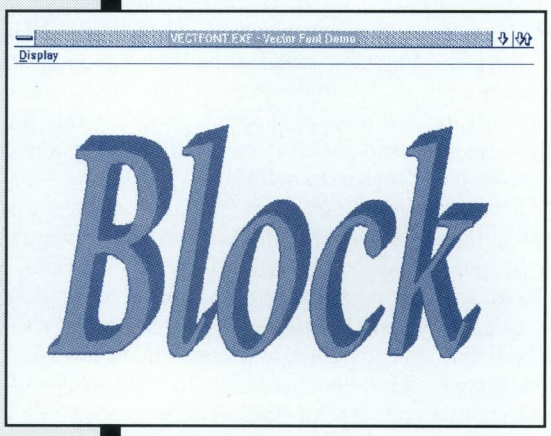
```
/*------------------------
   VF08.C -- Hollow font
   ------------------------*/

#define INCL_GPI
#include <os2.h>
#include "vectfont.h"

VOID Display_Hollow (HPS hps, LONG cxClient, LONG cyClient)
     {
     static CHAR szText[] = "Hollow" ;
     static LONG cbText = sizeof szText - 1 ;
     POINTL      ptl ;

     CreateVectorFont (hps, LCID_MYFONT, "Tms Rmn Italic") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;
     ScaleFontToBox (hps, cbText, szText, cxClient, cyClient) ;
     QueryStartPointInTextBox (hps, cbText, szText, &ptl) ;

     GpiBeginPath (hps, ID_PATH) ;
     GpiCharStringAt (hps, &ptl, cbText, szText) ;      // Text in path
     GpiEndPath (hps) ;

     GpiStrokePath (hps, ID_PATH, 0L) ;                 // Stroke path

     GpiSetCharSet (hps, LCID_DEFAULT) ;                // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;
```

Figure 11 Characters with a drop shadow, displayed by selecting Drop Shadow Font from the VECTFONT menu.

```
*---------------------------------
   VF09.C -- Font with Drop Shadow
   ---------------------------------*/

#define INCL_GPI
#include <os2.h>
#include "vectfont.h"

VOID Display_DropShadow (HPS hps, LONG cxClient, LONG cyClient)
     {
     static CHAR szText[] = "Hello!" ;
     static LONG cbText = sizeof szText - 1 ;
     POINTL      ptl ;

     CreateVectorFont (hps, LCID_MYFONT, "Tms Rmn Italic") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;
     ScaleFontToBox (hps, cbText, szText, cxClient, cyClient) ;
     QueryStartPointInTextBox (hps, cbText, szText, &ptl) ;

     GpiCharStringAt (hps, &ptl, cbText, szText) ;      // Shadow

     ptl.x -= 12 ;        // 1/6 inch
     ptl.y += 12 ;
     GpiSetColor (hps, CLR_BACKGROUND) ;
     GpiCharStringAt (hps, &ptl, cbText, szText) ;      // Text string

     GpiBeginPath (hps, ID_PATH) ;
     GpiCharStringAt (hps, &ptl, cbText, szText) ;      // Outline
     GpiEndPath (hps) ;

     GpiSetColor (hps, CLR_NEUTRAL) ;
     GpiStrokePath (hps, ID_PATH, 0L) ;

     GpiSetCharSet (hps, LCID_DEFAULT) ;                // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;
     }
```

```
/*----------------------------
   VF10.C -- Solid block font
  ----------------------------*/

#define INCL_GPI
#include <os2.h>
#include "vectfont.h"

VOID Display_Block (HPS hps, LONG cxClient, LONG cyClient)
     {
     static CHAR szText[] = " Block " ;
     static LONG cbText = sizeof szText - 1 ;
     INT         i ;
     POINTL      ptl ;

     CreateVectorFont (hps, LCID_MYFONT, "Tms Rmn Italic") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;
     ScaleFontToBox (hps, cbText, szText, cxClient, cyClient) ;
     QueryStartPointInTextBox (hps, cbText, szText, &ptl) ;

     ColorClient (hps, cxClient, cyClient, CLR_WHITE) ;
     GpiSetColor (hps, CLR_DARKGREEN) ;

     for (i = 0 ; i < 18 ; i++)
          {
          GpiCharStringAt (hps, &ptl, cbText, szText) ;     // Block

          ptl.x -= 1 ;
          ptl.y -= 1 ;
          }

     GpiSetColor (hps, CLR_GREEN) ;
     GpiCharStringAt (hps, &ptl, cbText, szText) ;     // Text string

     GpiBeginPath (hps, ID_PATH) ;
     GpiCharStringAt (hps, &ptl, cbText, szText) ;     // Outline
     GpiEndPath (hps) ;

     GpiSetColor (hps, CLR_DARKGREEN) ;
     GpiStrokePath (hps, ID_PATH, 0L) ;

     GpiSetCharSet (hps, LCID_DEFAULT) ;               // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;
     }
```

Figure 12 Solid block characters, displayed by selecting Block Font from the VECTFONT menu.




```
/*-------------------------------------------------------
   VF11.C -- Fading font with various pattern densities
  -------------------------------------------------------*/

#define INCL_GPI
#include <os2.h>
#include "vectfont.h"

VOID Display_Fade (HPS hps, LONG cxClient, LONG cyClient)
     {
     static CHAR szText[] = "Fade" ;
     static LONG cbText = sizeof szText - 1 ;
     LONG        lPattern ;
     POINTL      ptl ;

     CreateVectorFont (hps, LCID_MYFONT, "Tms Rmn Italic") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;
     ScaleFontToBox (hps, cbText, szText, cxClient, cyClient) ;
     QueryStartPointInTextBox (hps, cbText, szText, &ptl) ;

     GpiSetBackMix (hps, BM_OVERPAINT) ;

     for (lPattern = 8 ; lPattern >= 1 ; lPattern--)
          {
          GpiBeginPath (hps, ID_PATH) ;
          GpiCharStringAt (hps, &ptl, cbText, szText) ;  // Text out
          GpiEndPath (hps) ;

          GpiSetPattern (hps, lPattern) ;
          GpiFillPath (hps, ID_PATH, FPATH_ALTERNATE) ;  // Fill path

          ptl.x += 2 ;
          ptl.y -= 2 ;
          }

     GpiSetPattern (hps, PATSYM_SOLID) ;
     GpiSetBackMix (hps, BM_LEAVEALONE) ;
     GpiCharStringAt (hps, &ptl, cbText, szText) ;        // Solid

     GpiSetCharSet (hps, LCID_DEFAULT) ;                  // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;
     }
```

normally, like this:

```
GpiMove(hps, &ptlBeg);
GpiLine(hps, &ptlEnd);
```

and calling these same two functions within a path bracket and then stroking the path, like this:

```
GpiBeginPath(hps, idPath);
GpiMove(hps, &ptlBeg);
GpiLine(hps, &ptlEnd);
GpiEndPath(hps);
GpiStrokePath(hps, idPath, 0);
```

There are, in fact, some very significant differences.

First, the line drawn with the normal GpiLine function has what is called a cosmetic line width. The default width of the line is based on the resolution of the output device. It is a device-

Figure 13 Characters filled with different patterns, displayed by selecting Fading Font from the VECTFONT menu.

12 dependent width for a normal line. The width of the line does not change when you use matrix transforms to set different scaling factors in the coordinate space. Although GPI provides a function called GpiSetLineWidth to change the cosmetic line width, this function is not implemented in MS® OS/2 Version 1.1.

But a line drawn by stroking a path has a geometric line width. This is a line width (in world coordinates) that you set with the GpiSetLineWidthGeom function. Because this line width is specified in world coordinates, it is affected by any scaling that you set using matrix transforms.

Second, a line drawn with GpiLine can have different line types that you set with the GpiSetLineType function. These line types are various combinations of dots and dashes. The line is drawn with the current line color and the current line mix.

But a line drawn by stroking a path does not use the line type. The line is instead treated as an area that follows the path of the line but which has a geometric width. This area is filled with the pattern that you set with GpiSetPattern, and is colored with the current pattern foreground and background color, and the current pattern foreground and background mix.

Third, a line drawn by stroking the path can have various types of line joins and ends. By calling GpiSetLineJoin you can specify that lines meet with a rounded, square, or miter join. GpiSetLineEnd lets you specify rounded, square, or flat ends to the lines.

The function in the VF12.C file (see **Figure 14**) demonstrates the use of geometrically thick lines filled with patterns to give the letters a kind of neon look.

You can run this function by selecting Neon Effect from the VECTFONT menu. The function strokes the path using various geometric line widths filled with a PATSYM_HALFTONE pattern and several colors. The outline of the font is white, but it is surrounded with a halo of red.

A better effect could be achieved on devices capable of more than 16 colors. In this case, you can use a solid pattern but color each stroke with a different shade of red.

Clipping to the Text Characters

The third option after creating a path is to call GpiSetClipPath:

```
GpiSetClipPath(hps, idPath, lOption);
```

The lOption parameter can be either SCP_RESET (which equals 0L, so it's the default) or SCP_AND. The SCP_RESET option causes the clipping path to be reset so that no clipping occurs. The SCP_AND option sets the new clipping path to the intersection of the old clipping path and the path you've just defined in a path bracket. Any open areas in the path are automatically closed.

You can combine the SCP_AND option with either SCP_ALTERNATE (the default) or SCP_WINDING. As with GpiFillPath, you'll probably want to use alternate mode when working with paths created from vector fonts.

The function in VF13.C (see **Figure 15**) calls GpiCharStringAt with the text string WOW within a path bracket. This is followed by a call to GpiSetClipPath. The clipping path is now the interior of

```
/*-------------------------------------------------
   VF12.C -- Neon font using geometricall thick lines
  -------------------------------------------------*/

#define INCL_GPI
#include <os2.h>
#include "vectfont.h"

VOID Display_Neon (HPS hps, LONG cxClient, LONG cyClient)
     {
     static CHAR szText[] = " Neon " ;
     static LONG cbText = sizeof szText - 1 ;
     static LONG lForeColor[] = { CLR_DARKRED, CLR_DARKRED, CLR_RED,
                                  CLR_RED,     CLR_WHITE,   CLR_WHITE };
     static LONG lBackColor[] = { CLR_BLACK,   CLR_DARKRED, CLR_DARKRED,
                                  CLR_RED,     CLR_RED,     CLR_WHITE };
     static LONG lWidth[] = { 34, 28, 22, 16, 10, 4 } ;

     INT         iIndex ;
     POINTL      ptl ;

     CreateVectorFont (hps, LCID_MYFONT, "Tms Rmn Italic") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;
     ScaleFontToBox (hps, cbText, szText, cxClient, cyClient) ;
     QueryStartPointInTextBox (hps, cbText, szText, &ptl) ;

     ColorClient (hps, cxClient, cyClient, CLR_BLACK) ;

     for (iIndex = 0 ; iIndex < 6 ; iIndex++)
          {
          GpiBeginPath (hps, ID_PATH) ;
          GpiCharStringAt (hps, &ptl, cbText, szText) ;   // Text out
          GpiEndPath (hps) ;

          GpiSetColor (hps, lForeColor[iIndex]) ;
          GpiSetBackColor (hps, lBackColor[iIndex]) ;
          GpiSetBackMix (hps, BM_OVERPAINT) ;
          GpiSetPattern (hps, PATSYM_HALFTONE) ;
          GpiSetLineWidthGeom (hps, lWidth[iIndex]) ;

          GpiStrokePath (hps, ID_PATH, 0L) ;          // Stroke path
          }
     GpiSetCharSet (hps, LCID_DEFAULT) ;              // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;
     }
```

Figure 14 Neon characters using geometrically thick lines, displayed by selecting Neon Effect from the VECTFONT menu.

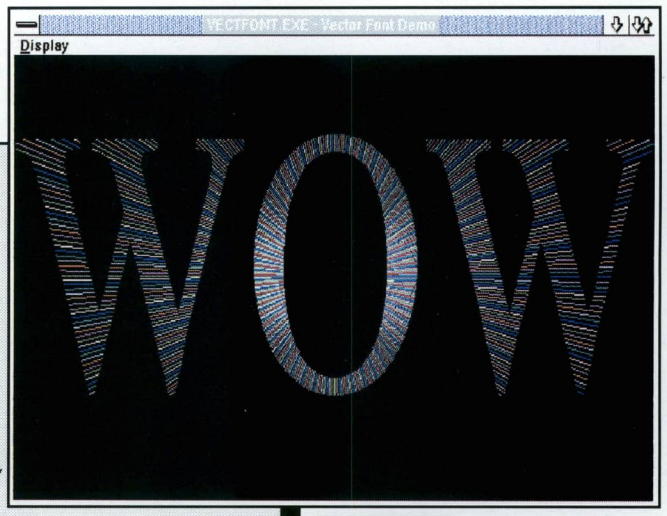


Figure 15 Color lines clipped to the interior of a text string, displayed by selecting Clipped Spokes from the VECTFONT menu.

```
/*--------------------------
   VF13.C -- Clipped Spokes
  --------------------------*/

#define INCL_GPI
#include <os2.h>
#include <math.h>
#include "vectfont.h"

VOID Display_Spokes (HPS hps, LONG cxClient, LONG cyClient)
     {
     static CHAR szText[] = "WOW" ;
     static LONG cbText = sizeof szText - 1 ;
     static LONG lColors[] = { CLR_BLUE, CLR_GREEN, CLR_CYAN,
                               CLR_RED,  CLR_PINK,  CLR_YELLOW,
                               CLR_WHITE } ;
     double      dMaxRadius ;
     INT         i, iNumColors = sizeof lColors / sizeof lColors[0] ;
     POINTL      ptl ;

     CreateVectorFont (hps, LCID_MYFONT, "Tms Rmn") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;
     ScaleFontToBox (hps, cbText, szText, cxClient, cyClient) ;
     QueryStartPointInTextBox (hps, cbText, szText, &ptl) ;

     ColorClient (hps, cxClient, cyClient, CLR_BLACK) ;

     GpiBeginPath (hps, ID_PATH) ;
     GpiCharStringAt (hps, &ptl, cbText, szText) ;     // Text string
     GpiEndPath (hps) ;

     GpiSetClipPath (hps, ID_PATH, SCP_AND | SCP_ALTERNATE) ;

     dMaxRadius = sqrt (pow (cxClient / 2.0, 2.0) +
                        pow (cyClient / 2.0, 2.0)) ;
                                                  // Draw spokes
     for (i = 0 ; i < 360 ; i++)
          {
          GpiSetColor (hps, lColors[i % iNumColors]) ;

          ptl.x = cxClient / 2 ;
          ptl.y = cyClient / 2 ;
          GpiMove (hps, &ptl) ;

          ptl.x += (LONG) (dMaxRadius * cos (i * 6.28 / 360)) ;
          ptl.y += (LONG) (dMaxRadius * sin (i * 6.28 / 360)) ;
          GpiLine (hps, &ptl) ;
          }
     GpiSetCharSet (hps, LCID_DEFAULT) ;               // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;
     }
```

the letters. The function draws a series of colored lines emanating from the center of the client window.

The function in VF14.C (not shown here) uses a similar technique but draws a series of areas defined by splines.

Modifying the Path

Between the call to GpiEndPath to end the path and the call to GpiStrokePath, GpiFillPath, or GpiSetClipPath, you can call GpiModifyPath. This function uses the current geometric line width, join, and end to convert every line in the path to a new line that encloses an area around the old line. For example, suppose that the path contained a single straight line. After GpiModifyPath the path would contain a closed line in the shape of a hot dog. The width of this hot dog is the geometric line width. The ends of the hot dog could be round, square, or flat, depending on the current line end attribute.

Following the creation of a path, these two functions in succession:

```
GpiModifyPath(hps, ID_PATH, MPATH_STROKE);
GpiFillPath(hps, ID_PATH,
            FPATH_WINDING) ;
```

are usually equivalent to:

```
GpiStrokePath(hps, ID_PATH, 0L);
```

GpiModifyPath and GpiStrokePath are the only two functions that use the geometric line width, joins, and ends.

In theory, you can call GpiStrokePath after GpiModifyPath, like this:

```
GpiModifyPath(hps, ID_PATH,
              MPATH_STROKE);
GpiStrokePath(hps, ID_PATH,
              0L);
```

This should do something, and it should be rather interesting, but GPI usually reports that it can't create the path because it's too complex.

Instead, let's look at GpiModifyPath followed by GpiSetClipPath. The function in VF15.C (see **Figure 16**) is almost the same as the one in VF13.C (see **Figure 15**) except that it sets the geometric line width to 6 ($^1/_{12}$ inch) and calls GpiModifyPath before calling GpiSetClipPath.

Note that the colored lines are clipped not to the interior of the characters but to their original outlines. By the use of GpiModifyPath, the outlines of the characters have themselves been made into a path that is $^1/_{12}$ inch wide. This is the path that is used for clipping.

Is It Enough?

I think it's clear that the facilities provided by GPI for working with vector fonts equal—and sometimes exceed—those in PostScript. The GPI interface is very powerful and very versatile.

Is that enough? No, it's not. The implementation of vector fonts in GPI has a structural flaw that still leaves PostScript the king.

Take a close and careful look at **Figure 3** and the display of the Helv font. You'll notice that the two legs of the H are different in width by one pixel when they should be the same width. This

14

is undoubtedly caused by a rounding error. It's obviously more noticeable on a low-resolution video display than it would be on a 300 dpi laser printer, but even on a laser printer such errors will affect the legibility of the text.

Errors such as this do not occur with PostScript fonts. PostScript fonts are true algorithms that are able to recognize and correct any anomalies in the rendition of the characters. In contrast, GPI fonts (which are encoded as a simple series of polylines and polyfillets) are drawn blindly without any feedback or correction.

So, while we can rejoice in what we have in GPI, there is still the need for some improvement. ❑

```
/*--------------------------
   VF15.C -- Clipped Spokes
  --------------------------*/

#define INCL_GPI
#include <os2.h>
#include <math.h>
#include "vectfont.h"

VOID Display_ModSpokes (HPS hps, LONG cxClient, LONG cyClient)
     {
     static CHAR szText[] = "WOW" ;
     static LONG cbText = sizeof szText - 1 ;
     static LONG lColors[] = { CLR_BLUE, CLR_GREEN, CLR_CYAN,
                               CLR_RED,  CLR_PINK,  CLR_YELLOW,
                               CLR_WHITE } ;
     double      dMaxRadius ;
     INT         i, iNumColors = sizeof lColors / sizeof lColors[0] ;
     POINTL      ptl ;

     CreateVectorFont (hps, LCID_MYFONT, "Tms Rmn") ;
     GpiSetCharSet (hps, LCID_MYFONT) ;
     ScaleFontToBox (hps, cbText, szText, cxClient, cyClient) ;
     QueryStartPointInTextBox (hps, cbText, szText, &ptl) ;

     ColorClient (hps, cxClient, cyClient, CLR_BLACK) ;

     GpiBeginPath (hps, ID_PATH) ;
     GpiCharStringAt (hps, &ptl, cbText, szText) ;      // Text string
     GpiEndPath (hps) ;

     GpiSetLineWidthGeom (hps, 6L) ;                    // 1/12 inch
     GpiModifyPath (hps, ID_PATH, MPATH_STROKE) ;
     GpiSetClipPath (hps, ID_PATH, SCP_AND | SCP_ALTERNATE) ;

     dMaxRadius = sqrt (pow (cxClient / 2.0, 2.0) +
                        pow (cyClient / 2.0, 2.0)) ;
                                                        // Draw spokes
     for (i = 0 ; i < 360 ; i++)
          {
          GpiSetColor (hps, lColors[i % iNumColors]) ;

          ptl.x = cxClient / 2 ;
          ptl.y = cyClient / 2 ;
          GpiMove (hps, &ptl) ;

          ptl.x += (LONG) (dMaxRadius * cos (i * 6.28 / 360)) ;
          ptl.y += (LONG) (dMaxRadius * sin (i * 6.28 / 360)) ;
          GpiLine (hps, &ptl) ;
          }
     GpiSetCharSet (hps, LCID_DEFAULT) ;                // Clean up
     GpiDeleteSetId (hps, LCID_MYFONT) ;
     }
```

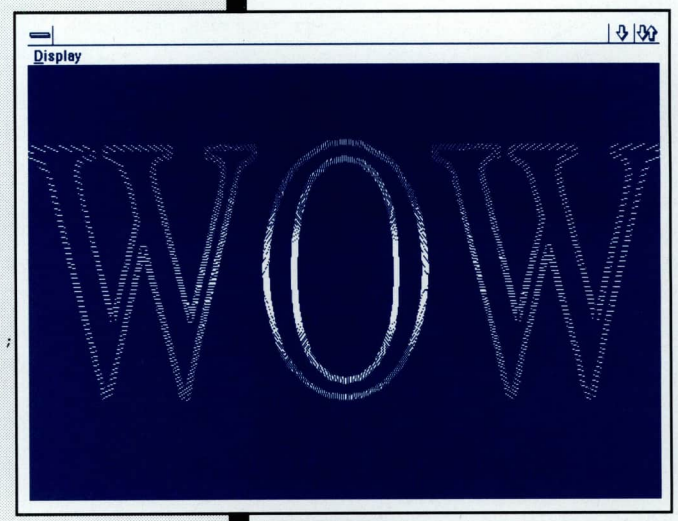


Figure 16 Colored lines clipped to the outline of text string characters, displayed by selecting Mod-Clipped Spokes from the VECTFONT menu.

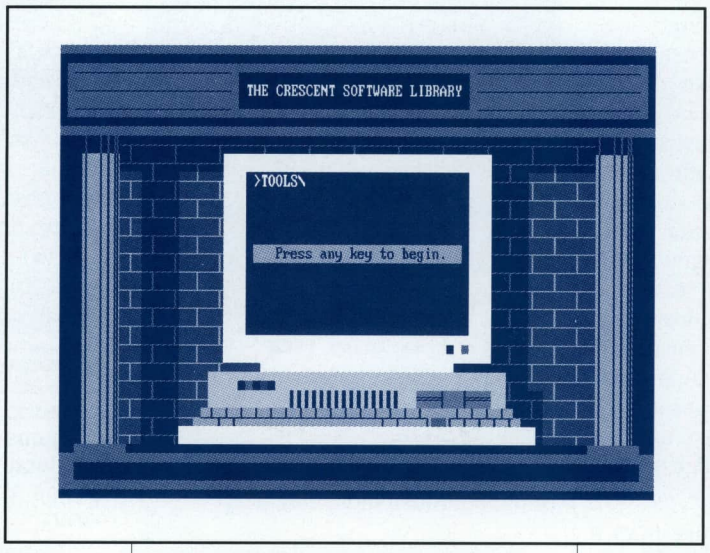
Basic as a Professional Programming Language

C and Macro Assembler (MASM) are the languages of choice for most programmers working in the personal computing environment. Occasionally one hears of a programmer using BASIC, but generally what one hears from professional programmers is that BASIC is a toy language—nice for learning, but hardly a language for serious program development.

That attitude is beginning to change. In fact, BASIC has grown up quite a bit over the past year. The Microsoft® QuickBASIC Version 4 user interface provides an effective working environment for both the beginner and the advanced user, and the BASIC language itself, via the Microsoft BASIC Compiler Version 6.0, now has the professional language features programmers need.

One believer in the ability of BASIC to handle the needs of programmers is Ethan Winer, founder of Crescent Software—a firm that specializes in software development in BASIC—as well as the developer of QuickPak Professional, a library of advanced routines designed especially for the BASIC compiler environment. Winer has also written numerous articles on BASIC for many industry publications.

Recently we talked to Winer, who shared with us the reasons BASIC is his language of choice, his reasons for developing QuickPak Professional, and some insights into the technical issues involved in BASIC programming.



BASIC offers the programmer extensive graphics capabilities.

MSJ: *Why has BASIC become the language of choice for you and Crescent Software?*

EW: Unlike C and Pascal, Microsoft BASIC has always included a wealth of built-in commands and features. It can produce sophisticated graphics on an assortment of monitors, open and manipulate files and devices, play music, and access memory and hardware ports directly. The release of Microsoft QuickBASIC 4 and the BASIC 6.0 compiler, has added even more commands and features.

BASIC has always been the easiest of the high-level languages to use, and the latest version is as powerful and capable as either C or Pascal. Furthermore, Microsoft has made clear they intend to keep improving both their BASIC 6.0 Professional Compiler and Microsoft QuickBASIC.

Of the features that raise BASIC to the level of other professional languages, which do you find particularly useful?

The overwhelming advantage of BASIC is that it's extremely easy to learn and use; the commands and functions have sensible names and their purpose is usually obvious. Unlike C or assembly language, it is almost impossible to overwrite the operating system or corrupt memory unless you really try, but this doesn't limit the control a program has over the system resources. For example, the PEEK and POKE commands let you read or write to any memory address, and INP and OUT will directly access hardware ports.

Another important advantage of BASIC is that it handles type conversion automatically. That is, you can freely multiply an integer times a double precision variable, and BASIC will automatically handle the math. String handling is equally powerful; there are functions to assign or extract characters anywhere in a string. Graphics is yet another powerful feature. BASIC has built-in commands to draw circles, boxes, filled boxes, and so forth, on all of the popular monitor types.

Why did Crescent Software focus on developing a library of programmer's tools for BASIC?

There is no disputing the fact that programming languages have come a long way in the past few years. Yet no matter how complete or well designed a

language is, programmers will always find they need some additional capability or feature. Third party libraries—or toolboxes—have traditionally filled that gap. A language like C requires external library support, due to its inherent limitations. For example, "pure" C can neither clear the screen nor locate the cursor, so C programmers must use toolbox routines, often viewing them as a necessary fact of life. Likewise, Pascal has its own limitations, and toolboxes are now equally common for that language as well.

But perhaps the most important reason to use a third-party library is to reduce the effort needed to achieve a completed program. No doubt knowledgeable programmers could design a pull-down menu system or a full-featured text editor with mouse support, but that requires an enormous amount of time, time better spent designing the rest of the program. Furthermore, end users are used to snappy screen displays, pop-up windows, and all the other hallmarks of a sophisticated user interface. A good toolbox product can provide "canned" solutions to such programming problems, as well as enhance and extend a language.

Did the library simply emerge from a collection of utilities you built up over time, or did you set out specifically to create the library?

Designing a set of tools requires much more effort than, say, merely writing an interface to the various DOS and BIOS services. We would often start designing a program only to discover that one or more special support routines were also needed. In many cases our needs led to the development of a custom function or subroutine that turned out to be useful in a broader sense.

One example is a set of functions that returns the minimum and maximum of two values. These are used extensively in many of the BASIC programs provided in QuickPak Professional and they eliminate what would otherwise be many IF/THEN statements. The MinInt assembler function returns the minimum of two integer values,

BASIC HAS ALWAYS BEEN THE EASIEST OF THE HIGH-LEVEL LANGUAGES TO USE, AND THE LATEST VERSION IS AS POWERFUL AND CAPABLE AS EITHER C OR PASCAL.

eliminating such code as:

```
IF X > MaxAllowed THEN
Value = MaxAllowed
ELSE
Value = X
END IF
```

Instead a single statement does the same thing, but much more elegantly:

```
Value = MinInt(X, MaxAllowed)
```

We made many other custom functions to assist the QEdit editor that also resulted in meaningful additions to the package.

In designing QuickPak Professional we had several very clear goals in mind. We wanted to provide ready-made solutions to common programming problems. These fall into two general categories: routines that are difficult for most programmers to create themselves, and services BASIC just cannot do directly.

An example of the first category would be routines written in assembly language, perhaps to search or sort an array very quickly. Other difficult programs would be a text editor, a spreadsheet, and a delay timer with microsecond resolution.

The second category is comprised of DOS and BIOS interrupts, because BASIC cannot easily access them. You would need these to list the files on a disk, change their date and time, or read and write a disk's volume label.

Second, we wanted these services to be very easy to use. For example, all of the routines that search and process strings would be in both case sensitive and insensitive versions. It was also important to limit the number of passed parameters to the absolute minimum, and implement the routines as functions where appropriate. Using functions rather than called subroutines is the most natural way to extend BASIC.

Third, in some cases BASIC's error handling abilities are not powerful enough. C or Pascal let you try opening a file and then check whether an error occurred, but BASIC requires a special error handling subroutine set up in advance. If an error occurs, you end up in the error handler—often with no idea of how you got there or where in the program to return to. So some means to test a disk drive or printer or even bypass BASIC's file handling altogether would be a useful addition.

From the looks of all the source code and tutorials you've provided, it would appear that you have very strong convictions about teaching BASIC.

Yes. What programmers often need is not just more language features, but an understanding of how to use those capabilities already present. The single most useful tool a programmer can acquire is knowledge.

Most accomplished programmers will tell you the best way to become proficient is by studying other people's programs. Indeed, many programmers are self-taught, learning solely from books and articles in popular computer magazines. By using someone else's program, studying the source code and perhaps even modifying it, a much deeper understanding results.

We want people to understand how these routines work, and be able to learn from them. This means not only providing all of the BASIC and assembly source, but also writing a series of tutorials explaining the underlying concepts.

The manuals that come with Microsoft QuickBASIC 4 are excellent as far as they go, but they gloss over a number of important topics; for example passing TYPE variables and arrays to subprograms and functions. Instead, the examples that use a TYPE array skirt the issue by declaring the array as SHARED. Likewise, the manual makes no mention of saving and loading arrays and EGA screen images to disk.

Programmers who would like to become more proficient in BASIC must understand these concepts. QuickPak Professional includes much of this information, along with tutorials on accessing files, storing string and numeric data in a program's code segment, and a comparison of the various ways procedures are designed.

One of the nice things about BASIC is that it relieves the programmer of the need to know about the nuts and bolts of the operating system. Unfortunately this also results in some limitations. Can you provide an example where you've removed some of these limitations while keeping BASIC's ease of use?

BASIC has many built-in commands, but it is admittedly lacking in the ability to access DOS and BIOS interrupts. Microsoft QuickBASIC 2 added a form of the CALL INTERRUPT feature, though only as an add-on available by linking with a special object module. Further, CALL INTERRUPT is clumsy to implement, and its use requires a knowledge of DOS services that many BASIC programmers do not possess.

Rather than simply provide a "watered down" replacement for CALL INTERRUPT, we felt it was important that the BASIC programmer not have to understand how DOS and BIOS services are accessed.

Let me give you an example. Any DOS function that accepts a file name expects that name to be in an ASCIIZ format; this is how C strings are stored. Unlike C, BASIC strings do not contain a zero byte to mark the end; instead, a string descriptor is maintained for each string or string array element. A string descriptor is a 4-byte table containing the length of the string and the address of the actual data.

Using CALL INTERRUPT required BASIC programmers to add the zero byte manually whenever a file name has to be passed to a DOS file service. Since one of our highest priorities was ease of use, we instead chose to have the file name copied to a temporary holding area; then the routine adds the terminating zero byte before calling DOS.

An important DOS service most programs need is the ability to get a list of file names from disk. No application worth its salt will force the operator to remember the names of files to be loaded. Instead, a menu should list all the files present, with some sort of "point and shoot" method provided for selecting one.

DOS provides no direct way to get a complete list of file names in one operation. Even if it did, the BASIC program would need to know beforehand the number of names present so sufficient string memory could be reserved for them. The solution we devised was to create a function that returns the number of file names matching a given search specification. Once this is known, a BASIC string array may be reserved for them, and a second subroutine retrieves all of the names from DOS at once.

This brings up an interesting point. Microsoft BASIC provides two ways to create a subroutine: subprograms accessed with the CALL keyword, and functions invoked from a BASIC expression. You seem to rely heavily on functions.

Functions are indeed useful and a major and welcome addition. One can develop a BASIC function in either BASIC or assembly language. The instructions for implementing an assembler function in QuickBASIC can be found in the Mixed-Language Programming Guide that comes with Microsoft Macro Assembler Version 5 (MASM).

One major advantage of a function is the elimination of a passed parameter, in situations where it is appropriate. If the routine that returns the number of matching files were set up as a program it would be called like this:

```
CALL FCount("*.*", Count)
PRINT Count; "files were found"
```

But designing the same routine as a function makes it both easier to use and understand:

```
PRINT FCount("*.*");_
 "files were found"
```

The output of a BASIC function may be used directly within a PRINT statement or, in the case of setting aside sufficient string elements, as part of the DIM command:

```
DIM Array$(FCount("*.*"))
```

Using a function like this is the most natural way to extend the BASIC language, and since one less parameter is needed, a function will be faster than an equivalent called subroutine.

Parameters in all Microsoft languages are passed on the stack, and it is no secret that stack operations can be very slow. This is why functions are especially good when used *without* parameters. For instance, we have included a set of functions to return the status of the various keyboard settings. The usual method to determine the status of, say, the Alt key is to use BASIC's DEF SEG, PEEK, and AND commands:

```
DEF SEG = 0
AltKey = PEEK(&H417) AND 8
IF AltKey THEN PRINT_
 "the Alt key is depressed"
```

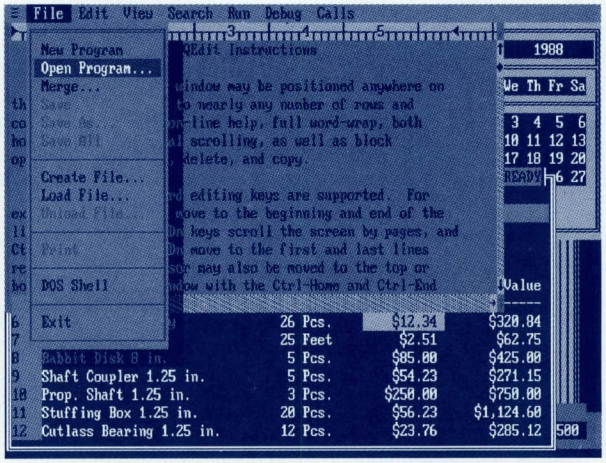
By using a dedicated function, you can replace all that code with a single statement like:

```
IF AltKey THEN PRINT_
 "the Alt key is depressed"
```

Better still, without passed param-

Figure 1: Sample BASIC Function Implemented in MASM

```
Xor  AX,AX            ;look at BIOS data in low memory
Mov  ES,AX
Mov  AL,ES:[417h]     ;get the keyboard status byte
And  AL,00001000b     ;clear all but the Alt Key bit
Ret                   ;return to BASIC, AX holds result
```



Pull-down menus and multiple windowed applications are all possible with BASIC. Note the scroll bars on the editor window.

18 eters only five instructions in assembler code are necessary to implement such a function (see **Figure 1**).

Assembly language surfaces again. It would appear that a knowledge of MASM might be necessary even for the BASIC programmer. What do you think?

Many programmers who use high-level languages would like to learn assembly language but are intimidated by what they believe will be a long and painful process. Learning assembly language is actually not that painful, and even a casual understanding from the perspective of a BASIC programmer is useful. Of course, you need not understand assembly language to use an assembler routine.

Are there any special things to know about assembly language functions?

The most obvious thing that comes to mind is that, before an assembly language function can be used in Microsoft BASIC, the program that uses the function must declare it. Having to declare procedures is new to BASIC, but in this case it is not unreasonable. Or else—in the Alt key example—Microsoft QuickBASIC, would assume that the Alt key is simply an integer variable. By declaring it ahead of time, Microsoft QuickBASIC knows that the Alt key is a function to be called, and the value returned in AX holds the result.

And using assembly language speeds things up here and there.

That's an equally important reason for using assembly language. We wanted to speed up those operations that typically are very slow in any high-level language. Again, assembly language is the key to great performance, and about two-thirds of the routines in QuickPak Professional are in assembler.

The majority of MS-DOS® language compilers, for example, do screen printing through the operating system in order to be compatible with as many machines as possible. Whether this is done via DOS or the BIOS, the results are too slow. Microsoft QuickBASIC 4 partly addresses this by writing directly to video memory, but there is still much room for improvement. Each character must be examined to see if it is a special control character and for each character it takes extra time just to see if the screen needs to be scrolled. We devised several quick printing routines to accommodate multiple video pages and to display new text without destroying the colors already on the screen.

One particularly useful routine displays a portion of a string array, containing it within a specified area of the screen. This greatly simplifies the creation of, for example, a browse facility, where you can scroll up and down through the entire array on the screen. Again, our intent was to have these routines do as much as possible and save the programmer unnecessary work.

Assembler routines would also give the BASIC programmer substantially faster array processing.

This is exactly where assembly language really gives BASIC a much needed boost, in the area of processing arrays. We have, for example, provided a set of functions for each of Microsoft BASIC's numeric array types to return the largest or smallest values. Using a dedicated assembler routine is typically six or seven times faster than an equivalent function in BASIC.

An even greater improvement can be had when saving an entire string array to disk and then reloading it later. One of the slowest operations BASIC performs is reading data from a sequential file; it must examine every single byte to see if it is either a carriage return that marks the end of a string, or the CHR$(26) byte that marks the end of a file. Creating a custom assembler routine to capture the entire file in one operation saves an enormous amount of time.

The fastest way to save a numeric array in BASIC is to use BSAVE, as opposed to making multiple PRINT statements to a file. The major problem with the latter is the overhead required to convert the values from the internal format used by the PC into the equivalent ASCII digits. Further, BSAVE takes up less disk space.

In the case of integers, only 2 bytes are used to store the number in memory, regardless of its value. Contrast that with up to five digits (six if the number is negative) to store the number as ASCII digits. Worse, each number in the file would also need either a carriage return/line feed pair, or a delimiting comma. BSAVE instead captures a "snapshot" of any area of memory, and sends it to disk in one operation. The complementary command is BLOAD, which retrieves a previously BSAVE'd file.

What about string arrays?

Unlike numeric arrays, BASIC string arrays do not occupy contiguous memory addresses. Instead, the descriptor tables are contiguous, and the actual data could be anywhere in near

memory. Therefore, before BSAVE can be used on a string array, the data from all of the elements must be gathered up into a single block. A pair of dedicated routines processes all of the string elements in one operation, and copies them to an integer array and back again. An additional routine lets the programmer retrieve an individual string element if needed.

Numeric arrays in Microsoft QuickBASIC are not restricted to near memory. Just being able to copy a string array out to far memory is a useful feature. And if far memory becomes congested, it's easy to save the arrays to disk to open additional space. Further, because we delimit the end of each string with a carriage return/line feed pair, the file is directly readable by any application.

You've provided BASIC users with two "new" data types. What are Very Long Integers and Bit Arrays?

While I was working on QuickPak Professional, several interesting concepts and routines resulted, including the development of two "new" variable types.

One important and long-overdue feature introduced with Microsoft QuickBASIC 4 is support for long (4-byte) integers. Where appropriate, long integers have a decided advantage over floating point numbers since you can process them very quickly, without any rounding errors. Accountants like that. Considered as pennies, their range of +/-$21,474,836.47 is generally adequate—except for serious financial work.

As a solution, we devised a set of routines for adding, subtracting, multiplying, and dividing what we call Very Long Integers. These variables use 8 bytes apiece and let you manipulate extremely large numbers without rounding errors. You assign a Very Long by aliasing it into a conventional double precision variable, and BASIC isn't any the wiser.

In the opposite direction, we also created a pair of routines to manipulate Bit Arrays. The smallest variable that BASIC provides is a 16-bit integer. When a program needs the range of values offered by integers, using them makes a lot of sense. But often all that is needed are simple true or false variables, and an integer array can waste an enormous amount of memory. A 1000-element bit array occupies only 125 bytes, compared to 2000 bytes for a conventional integer array.

There is a routine you supply that lets a BASIC program write to two monitors at the same time. How did you accomplish that?

It isn't difficult to create a routine that changes the active monitor, and the very act of switching monitors always clears the screen in the process. Because there is nothing to prevent a program from writing directly to screen memory for an inactive monitor, our solution was both effective and easy to implement.

All of the video routines in QuickPak Professional automatically determine the type of monitor installed when called. This is necessary for two reasons. First, the video segment is different for color and monochrome monitors, so the correct segment must be known before a routine can write directly to screen memory. Second, and equally important, CGA monitors create "snow" unless the reading and writing is synchronized to the horizontal retrace.

In the routine that writes to any monitor, though, the caller must instead indicate the type of monitor to write to with a code; 1 means monochrome, 2 is a CGA, and 3 means an EGA or VGA. Even though a PC can support two monitors at once, they must be different—that is, both cannot be color, or both monochrome. Thus the programmer would simply specify the monitor type that is not the current one. Of course there is also a function that reports the currently active type of monitor.

You've also given BASIC programmers a way to dump screen images regardless of the screen mode they may be in. What can you tell us about that?

We also wanted to address graphics. A major limitation in the GRAPHICS.EXE program that comes with DOS is that it works only in the CGA screen modes. In addition, GRAPHICS works only with printers that are compatible with the Epson®/IBM® command set. Since BASIC supports the EGA, VGA, and Hercules® standards, it was important to provide a routine that could print a graphics image from any of these modes.

We realized that to be truly useful a screen print routine must know not only the Epson printer codes (which all modern printers can emulate), but also the HP® LaserJet® codes. Since a LaserJet can print in several sizes, the caller should be able to specify which resolution to use. The only remaining problem was what to do with the colors.

A stunning three-dimensional pie chart displayed in sixteen colors is useless as a printout; all you'd get is a big black circle. Clearly, the best solution was to substitute a different hatching pattern for each of the possible screen colors. But what complicates things considerably is the different ways that graphics screen memory is organized.

In text modes, each successive character on the screen is contained in successive bytes in display memory. But in CGA graphics modes the screen memory is organized using a method known as interlacing. In an interlaced system, every other row is in a different block of memory, which makes accessing the memory much more difficult.

Hercules graphics also interlaces, except that it uses every fourth row. EGA and VGA adapters use yet another system, where each color is in an entirely different segment. This complicated both reading screen memory and translating the colors into hatching patterns.

One of your accomplishments was the creation of a word processor, QEdit, written in BASIC. BASIC's variable length strings must have been very useful there.

Languages like C or Pascal that permit only fixed length strings take much more programming to get the same results. The program either wastes a substantial amount of memory for text lines that are shorter than the maximum,

or it must maintain a table of pointers to keep track of where in memory each string begins. As characters and lines are inserted or deleted, the pointers must be constantly updated.

The fact that BASIC supports variable length strings reduced our effort considerably. BASIC does all of this very quickly and automatically, and wordwrapping in QEdit will keep up with even the fastest typist. [*See* ***Figure 2*** *for an example of a wordwrap function in BASIC, and* ***Figure 3*** *for an example of how to code a function to obtain screen colors—Ed.*]

Variable length strings need to be dynamically allocated. This must have caused some significant problems.

One of our biggest difficulties was creating the string array processing routines, which required additional research. In particular, we had to get a fair amount of information about Microsoft BASIC's internal workings. As you mentioned, since BASIC permits strings of nearly any length, they are allocated dynamically as necessary. This complicates memory management considerably and, not surprisingly, Microsoft considers many of those details to be proprietary.

This became evident immediately when we started writing the routines to sort a string array. I noted earlier that BASIC strings use a descriptor table to tell each string's length and memory location. At first glance, it seems that any two strings can be exchanged by just swapping their descriptor tables. Descriptors are well documented in the Microsoft QuickBASIC manuals, but unfortunately that isn't the entire story.

What is *not* mentioned is how string data is tied. It took us several days to figure it out by trial and error.

Once we could exchange strings for sorting purposes, it was a simple matter to insert or delete elements in an array. On a standard IBM PC, inserting a single string at the beginning of a 2000 element array takes more than two seconds using Microsoft QuickBASIC. Contrast that to less than a tenth of a second for the equivalent routine written in assembler.

Some of the string functions in QuickPak must have caused similar problems. But again it seems that by providing "functions" you've made life easier.

String functions were indeed yet another difficult feature to implement—and again, because of the lack of information. Besides the advantage of eliminating a passed parameter, a function that can directly return a string saves the programmer from having to set aside space beforehand.

One of the routines in QuickPak Professional obtains the current directory for a specified drive. Since a directory name can be as long as 64 characters, such a routine would first need a string of that length allocated. When the routine is finished, the extra characters at the end must be removed manually.

Because DOS uses a zero byte to mark the end of any strings it returns, the program must use BASIC's INSTR function to find that byte; then it keeps only those characters that precede it.

```
Dir$ = SPACE$(64)
  'set aside 64 bytes
Drive$ = "C"
  'specify which drive
CALL GetDir(Drive$, Dir$)
  'GetDir loads Dir$
Zero = INSTR(Dir$, 0)
  'find the zero byte
Dir$ = LEFT$(Dir$, Zero-1)
  'keep what's before the 0
```

When GetDir is designed as a string function, it is considerably easier to use:

Figure 2: WORDWRAP.BAS

```
DEFINT A-Z
DECLARE SUB WordWrap (X$, Wide)

CLS

A$ = "BASIC strings use a descriptor table ot tell each string's"
B$ = "length and memory location. At first glance, it seems that"
C$ = "any two strings can be exchanged by just swapping their"
D$ = "descriptor tables. Once we could exchange strings for"
E$ = "sorting purposes, it was a simple matter to insert or"
F$ = "delete elements in an array. On a standard IBM PC, inserting"
G$ = "a single string at the beginning of a 2000 element array"
H$ = "takes more thatn two seconds using Microsoft QuickBASIC."

W$ = A$ + B$ + C$ + D$ + E$ + F$ + G$ + H$
PRINT W$
PRINT
Wide = 60                'the maximum width of the display
WordWrap W$, Wide



SUB WordWrap (X$, Wide%)

  Length% = LEN(X$)            'remember the length
  Pointer% = 1                 'start at the beginning of the string

  'scan a block of sixty characters backwards, looking for a blank
  'stop at the first blank, or if we reached the end of the string
   DO
     FOR X% = Pointer% + Wide% TO Pointer% STEP -1
        IF MID$(X$, X%, 1) = " " OR X% = Length% + 1 THEN
         'LOCATE , LeftMargin   'optional to tab in the left edge
          PRINT MID$(X$, Pointer%, X% - Pointer%);
         'LPRINT [TAB(LeftMargin)]; MID$(X$, Pointer%, X% - Pointer%)
          Pointer% = X% + 1
          WHILE MID$(X$, Pointer%, 1) = " "
           Pointer% = Pointer% + 1 'swallow extra blanks to next word
          WEND
          IF POS(0) > 1 THEN PRINT  'if cursor didn't wrap next line
              EXIT FOR                  'done with this block
           END IF
       NEXT
    LOOP WHILE Pointer% < Length%       'loop until done

END SUB
```

```
Drive$ = "C"
Dir$ = GetDir$(Drive$)
```

or

```
Dir$ = GetDir$("a")
```

As a function, GetDir locates the zero byte, and then returns a string that is only as long as actually needed.

Let's go back and talk about QEdit again. You seem to have emulated the Microsoft QuickBASIC 4 edit commands in QEdit. With all of the editors already available, why did you create another?

Though there certainly are many editor programs available commercially, I don't know of any that come with free source code, and none meant to be added to another program. People often ask us to develop new routines, but by far the one they request most is a text editor that can be customized and added to their own programs. Of course, "editor" means different things to different people, and every programmer has his or her own idea of how one should work.

Because our customers are familiar with the Microsoft QuickBASIC 4 edit commands, it made the most sense to emulate those. But since we provide the editor's source code, it is easy to customize the various keystrokes that are recognized. But deciding on the editing commands was just the beginning.

To be truly useful, a text editor must be able to accept text both as individual lines and in the "one line per paragraph" method of many word processors. We also wanted the operator to be able to change margins, size the window, and scroll the text using a mouse—without the calling program having to do any additional work. Of course, keystrokes must be processed quickly enough to be totally transparent to the user.

QEdit contains a number of important features such as full mouse support, block operations, and on-line help. But perhaps the most important feature is the ability to remove features that aren't needed. After all, if someone wants a bare bones editor with margins and wordwrap, there should be a way to exclude the code for block operations and mouse support. We therefore placed those portions of QEdit into separate sections, where users can easily delete or remark them out.

One additional feature we wanted for our own use was block operations on columns, as well as for words and lines. Most editors work only in sentence mode, so that marking a block downward captures the entire length of the line. In a word processor this is often sufficient. But when programming, the ability to mark a long section and instantly change the indent level is very useful. A column mode also simplifies copying or moving a table of numbers.

Apparently QEdit was designed for use as a pop-up utility.

In reality, though QEdit is not meant as a standalone word processor, it certainly could be. It was, however, designed so that users can add it as a "pop-up" to a main BASIC program, thus showing that even sophisticated programs can be written in BASIC.

What sort of problems did you encounter?

Because of the number of features in QEdit, it uses four separate help screens,

Figure 3: GETCOLOR.BAS

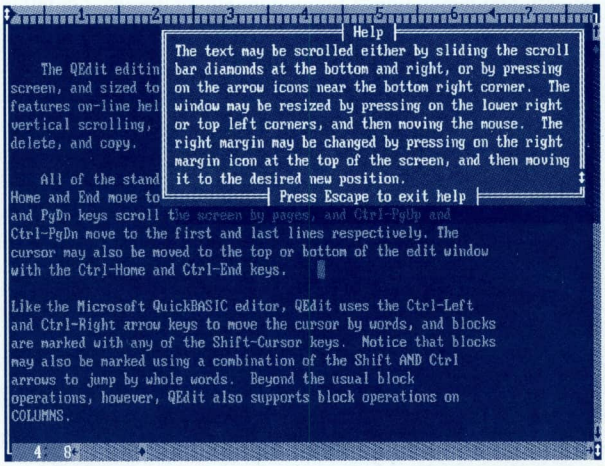
```
DEFINT A-Z
DECLARE SUB GetColor (FG, BG)            'gets BASIC's current colors
DECLARE SUB SplitColor (XColor, FG, BG) '.ASM - splits a color into FG
                                         ' and BG
DECLARE FUNCTION OneColor% (FG, BG)      '.ASM - combines FG/BG into one
                                         ' color


CLS
INPUT "Enter a foreground color value (0 to 31): ", FG
INPUT "Enter a background color value (0 to 7) : ", BG
COLOR FG, BG

PRINT : PRINT "BASIC's current color settings are: ";
GetColor FG, BG
PRINT FG; "and"; BG

PRINT "That combines to the single byte value of"; OneColor%(FG, BG)
PRINT "Broken back out results in";
SplitColor OneColor%(FG, BG), NewFG, NewBG
PRINT NewFG; "and"; NewBG

COLOR 7, 0      'restore defaults before ending

'This function obtains BASIC's current colors by first saving the
'character and color in the upper left corner of the screen.  Next,
'a blank space is printed there, and SCREEN is used to see what color
'was used.  Finally, the original screen contents are restored.
'
SUB GetColor (FG%, BG%) STATIC
  V% = CSRLIN                            'save the current cursor location
  H% = POS(0)
  SaveChar% = SCREEN(1, 1)               'save the current character
  SaveColor% = SCREEN(1, 1, 1)           'and its color
  SplitColor SaveColor%, SaveFG%, SaveBG%

  LOCATE 1, 1                            'print with BASIC's current color
  PRINT " "; CHR$(29);                   'back up the cursor to 1,1
  CurColor% = SCREEN(1, 1, 1)            'read the current color
  COLOR SaveFG%, SaveBG%                 'restore the original color at 1,1
  PRINT CHR$(SaveChar%);                 'and the character

  LOCATE V%, H%                          'put the cursor back where it was
  SplitColor CurColor%, FG%, BG%         'split the color into separate
                                         'FG and BG
  COLOR FG%, BG%                    'restore BASIC's current value for it
END SUB
```

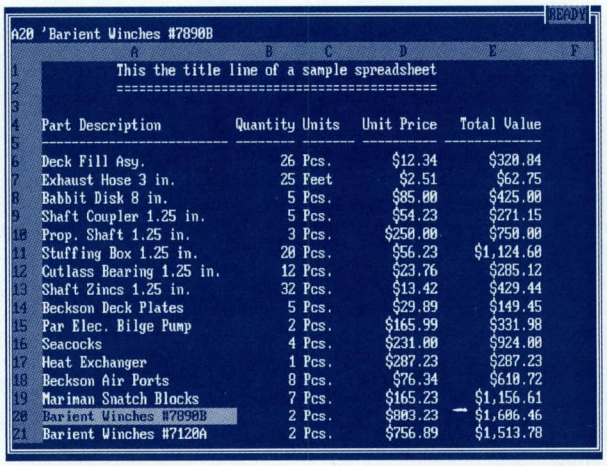
accessed with the F1 key. The help text is in the code segment of the program, and a special assembler routine retrieves it. This saves nearly 2Kb of string memory. Also, since users can remove the mouse and block operations, a variable is modified in each of those sections so the help code will know which screens are appropriate.

Besides storing the help text outside of string memory, we used the same technique for the cut-and-paste clipboard. Otherwise, with a fairly large document loaded, BASIC could run out of memory when capturing the block. Which brings up an important point. As difficult as it was to capture a column that may span several screens, deleting or pasting it somewhere else was even tougher. Moreover, to prevent the text block from stealing string space, the clipboard is kept in an integer array. We had already written a string manager to copy all or part of a string array to an integer array and back, but capturing and pasting a column required two additional custom assembler routines.

One routine copies elements from a string array to an integer array, but starting at a given offset and for a specified number of characters. If a line ends short of the column width, it pads the remainder with blanks. The second custom routine retrieves the string portions one by one from the integer array, so the wordwrap routine can insert the clipboard contents back into the text.

We also needed a routine to close a portion of a screen that had been saved earlier. Because QEdit is a pop-up, it saves the underlying screen automatically. When the screen size changes, it





This editor and spreadsheet were both developed using BASIC and the functions available through the Crescent Software library.

complicates matters.

So you also had to develop routines for sizing windows on the screen. How was this handled?

The user can at any time place the mouse cursor at the upper left or bottom right corner, and move the corner to a new location. Of course, sizeable windows are nothing new, but we wanted the window to actually change, rather than simply show where it would be upon release of the mouse button.

With some programs, changing the window size produces an enormous amount of flicker. In QEdit, we employed a custom routine to close only that portion of the window actually needed to reduce the window size.

We developed many other related routines both for QEdit and QuickPak Professional in general. For example, one quickly scans a string array for the number of active lines. Another contains the mouse cursor within a specified area on the screen. Another determines the current video mode so a program can accommodate 25, 43, or 50 lines automatically.

As you can see, an enormous amount of detail is required to implement a full-featured word processor, regardless of the language used. Microsoft QuickBASIC and the Microsoft BASIC 6.0 compiler are ideal for any application that requires extensive string manipulation, but they are especially appropriate for a word processor.

You have certainly made a strong case for BASIC as a "real" language.

I truly believe in BASIC as a serious applications language. It has made computer programming—real programming—accessible to millions of people. It is unfortunate that many otherwise informed programmers view BASIC 6.0 and Microsoft QuickBASIC as unsuitable for "real" applications simply because of the BASIC interpreter that comes with DOS. Besides exceptional performance, the Microsoft BASIC products also support recursion, structured code and data, and many other features needed for a modern, professional language. ❑

Organizing Data in Your C Program with Structures, Unions, and Typedefs

Greg Comeau

Many of you have no doubt used structures, unions, and typedefs in your C programming careers. In general, structures are easy to use, and they help in the programming paradigms involved with the correct usage of data in programs. They also keep code readable and maintainable. Unions and typedefs can perform some of these duties as well, but their usage tends to become more complicated and cryptic.

Since the mechanism of struct and union is pretty well established, I will not spend time describing their basic syntax (the mechanism by which the grammar of the C language dictates the way they may be written) or semantics (the way the compiler treats them in a meaningful fashion). Instead I will describe some of the idiosyncrasies that crop up with structures, unions, and typedefs, providing a resource to use to avoid coding bugs, recognize portability problems, and know more about the C language.

Recent Additions

The initial specification of C appeared publicly in the classic text *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie (herein referred to as K&R). In that specification, structure assignments, structure arguments, and functions that returned structures were not allowed. Present day compilers, however, allow these activities to occur, and these features have been formalized by the American National Standards Institute's draft proposal for C (ANSI C) which, I understand, will have no more public review periods and will at last become a bona fide standard sometime around March of 1989.

Another feature that has made its way into that draft is initialization of automatic structures. However, this is not available under all compilers, especially those that typically come with the UNIX® or XENIX® Compiler Development Set. For example, the structure definition in **Figure 1A** will not compile under those compilers unless the static storage class is added as an attribute as is shown in **Figure 1B**.

This places an extra burden on the programmer and is due to the compiler following the older language specification rather than some technical constraint in compiler technology. This restriction may also force you to fill up more of your programs' static storage space area than you'd like. If space is at a premium, this may present prob-

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.

THIS ARTICLE DESCRIBES SOME OF THE IDIOSYNCRASIES THAT CROP UP WITH STRUCTURES, UNIONS, AND TYPEDEFS, PROVIDING A RESOURCE TO USE TO AVOID CODING BUGS, RECOGNIZE PORTABILITY PROBLEMS, AND KNOW MORE ABOUT THE C LANGUAGE.

Figure 1: Simple Examples of Structs

A

```
1       main()
2       {
3           struct {
4               int a;
5           } s = { 1 };
6       }
```

B

```
1       main()
2       {
3           static struct {
4               int a;
5           } s = { 1 };
6       }
```

C

```
1       main()
2       {
3           struct {
4               int a;
5           } s1, s2;
6
7           if (s1 == s2)        /* syntax error */
8               printf("s1 == s2\n");
9           else
10              printf("s1 != s2\n");
11      }
```

Figure 2: Using Member Access Operators

```
1       struct s {
2           int     y;
3           int     *x;
4       };
5
6       struct s foo;
7       struct s *pfoo = &foo;
8
9       main()
10      {
11          printf("%d\n", foo.x);
12          printf("%d\n", pfoo->x);
13          printf("%d\n", &pfoo->x);
14          printf("%d\n", *pfoo->x);
15          printf("%d\n", &foo.x);
16          printf("%d\n", *&pfoo->x);
17          printf("%d\n", &*pfoo->x);
18          printf("%d\n", sizeof(int));
19          printf("%d\n", &foo);
20      }
```

Figure 3: Establishing Operator Precedence

```
1       &(pfoo->x)
2       *(pfoo->x)
3       &(foo.x)
4       &(*(pfoo->x)) or simply pfoo->x
5       *(&(pfoo->x)) or simply pfoo->x
```

lems. The alternative is to code explicit assignments for each member somewhere in the function. In terms of efficiency, this probably isn't going to make much of a difference since this is what the compiler would do with the initialization of the automatic. However, it would make your code more cluttered.

Automatic unions and arrays used to have this problem too. They have been allowed to be initialized without constraints by the ANSI C proposal. Since this has become common practice and has been mandated by ANSI, I would think twice about any vendor that does not yet support this.

Comparing Structures

If you were to try to compile the example code from **Figure 1c**, you'd find that it produces a syntax error. This is because you cannot compare structures or unions *by name* the way you'd perform a structure assignment (for example, struct1 = struct2;). The only way to accomplish the comparison is by checking each member of the structure individually.

This may sound like a silly restriction to place on a structure (like the inability to easily assign and initialize used to be) and it is—to an extent. However, there is a twofold reason for this. First, if the operation were allowed, it would be expected that it should work for the inequivalency operator (!=), for the relational operators (<, >, <=, >=), and perhaps even for some others. This would not be 100 percent desirable since it would tend to put the compiler through quite a few gyrations.

You may very well think "So what?" about this point in time. But I haven't told you the entire story yet. What happens is that a data aggregate like a structure could possibly have extraneous memory associated with it (refer to the sizeof section below), and generating proper code would be rather extravagant, especially to support the relational operators. Furthermore, cross compilation, or rather cross assembly, of such code could produce difficult to track bugs.

Member Access

Most C programmers I know have no problem understanding C's two member access operators: the . and the ->. The dot allows for the retrieval of a given member within a structure or union, and the arrow allows retrieval from a member referenced by a pointer to a structure or union. However, there seems to be much confusion about the use of other C operators with the member access operators. The ones that seem to cause the most confusion are the & (address of) and the * (indirection) operators. For the novice, something like &*p->x might as well be hieroglyphics. Actually, if you type in the sample listing in **Figure 2**, many of you, novice and experienced alike, may be surprised to see exactly what this construct and derivations of the construct do.

The use of the member access operators is mandated strictly by the operator precedence chart. This chart (one can be found on page 137 of the Microsoft® C

Compiler (referred to herein as MSC) Version 5.1 *Language Reference*) makes it clear that both the dot and arrow operators have the highest precedence allowed (along with ( ) and [ ]). Please remember this. As stated in "A Guide to Understanding Even the Most Complex C Declarations," *MSJ* (Vol. 3, No. 5), precedence is also an important declaration consideration. Knowing what the highest precedence operators are is one of the key facts C programmers should know (meaning have memorized) since many things in the language depend upon it. Also, from a coding viewpoint, it usually makes reading, writing, and maintenance a snap. I cannot emphasize this point strongly enough.

Figure 2 should now make more sense. I'll assume that most of you are fine until at least line 13. If we break down each of the subsequent statements, we should get the parenthesized results found in **Figure 3** (you can use it as a guide). Therefore, since pfoo->x means to access a member x of a structure, which is pointed to by pfoo, line 12 says to get the address of the member x, whose structure is being accessed indirectly via pfoo. Note that this does not mean to take the address of pfoo and treat it as a pointer to a structure that contains a member x.

Line 14 instructs the compiler to get the contents of the member x of the structure that pfoo points to. This works out fine since x is an int *. However, this does not mean to treat the contents of the pfoo variable as a struct pointer to obtain x. If it meant that, it would be equivalent to pfoo->x, and this obviously isn't the case.

Line 16 is a bit interesting to me since it is a case of operators negating each other. For instance, if we were to have a declaration such as int x, we could say *&x. To take this one step at a time, this construct first takes the address of x, which implicitly gets cast into an int *, and then obtains the contents of the value of that address. Well, isn't that simply the same as having just used x itself in the first place? This same form of destructive interference happens on line 16 since it's going to get the contents of the address of pfoo->x (and we know it's going to interpret it in this way because of operator precedence).

Finally, although line 17 is constructed somewhat differently from line 16, the end results are the same. However, the reason is admittedly cryptic and requires memorization since it is not immediately intuitive. For instance, in the code in **Figure 4**, one would expect &*pvar in line 10 to map into &9999 since *pvar is 9999 (which is an error, of course, since you can't take the address of a constant). Instead, if we look at this as though we were reading it as the address of the contents of pvar, then since the contents of pvar is var, its address is &var. This same logic applies to line 17 in **Figure 2**.

I've included lines 15, 18, and 19 in case you decide to investigate the example any further. It might be worth proving to yourself that you understand the above by assuming that the declaration of the member x was changed to char *x. For practice, given this situation, would you be able to change the printf format specifications in **Figure 2** to something more appropriate (such as %s or %c)?

Figure 4: Advanced Examples of Access Member Operators

```
1       main()
2       {
3           int     var = 9999;
4           int     *pvar = &var;
5
6           printf("%d\n", pvar);
7           printf("%d\n", *pvar);
8           printf("%d\n", &pvar);
9           printf("%d\n", *&pvar);
10          printf("%d\n", &*pvar);
11          printf("\n");
12          printf("%d\n", var);
13          printf("%d\n", &var);
14          printf("%d\n", *&var);
15          printf("%d\n", &*&var);
16          printf("%d\n", *&*&var);
17      }
```

The sizeof Structure

Another rather cryptic and nasty sort of thing to be aware of when using structures is their size. Basically, what you see is not necessarily what you get. But why should you care?

Let me explain by using another code sample (see **Figure 5**). I ran it on an 80386 computer and got the results shown. The sizeof(char) and the sizeof(int) print 1 and 4 since this is the size of a byte and a natural word on a 386 (under UNIX and the DOS large model). Note that things seem to be going along smoothly while printing the storage space requirements for chara and charab, but all of a sudden the space for huh seems to be out of whack. This is not a bug in the compiler or in the program. There are parts of C that have been allowed to be classified as undefined behavior and this is one of those areas. In this circumstance, huh takes up 8 bytes instead of 5 because intb uses 4 bytes and it must be aligned on a word boundary. Therefore, since chara only uses one byte, the compiler will insert a hole of 3 unnamed and inaccessible bytes after chara to en-

Figure 5: Program that Shows Structure Sizes

```
1       #define offsetof(type, identifier) (&(((type *)0)->\
        identifier))
2
3       main()
4       {
5               struct chara {
6               char    chara;
7           };
8
9           struct charab {
10              char    chara;
11              char    charb;
12          };
13
14          struct huh {
15              char    chara;
16              int     intb;
17          };
18
19          struct huh2 {
20              int     intb;
21              char    chara;
22          };
23
24          printf("%d\n", sizeof(char));
25          printf("%d\n", sizeof(int));
26
27          printf("%d\n", sizeof(struct chara));
28          printf("%d\n", sizeof(struct charab));
29          printf("%d\n", sizeof(struct huh));
30          printf("%d\n", sizeof(struct huh2));
31
32          printf("%d\n", offsetof(struct charab, charb));
33          printf("%d\n", offsetof(struct huh, intb));
34          printf("%d\n", offsetof(struct huh2, intb));
35      }
```

The output for this program on an 80386 computer is:

```
1
4
1
2
8
8
1
4
0
```

IN SYSTEMS PROGRAMMING, AN OPERATING SYSTEM RESOURCE OR REQUEST CALL, A DEVICE DRIVER, OR A SPECIFIC HARDWARE DEVICE MAY REQUIRE THAT DATA TRANSFERRED TO OR FROM IT HAVE SPECIFIC ALIGNMENT REQUIREMENTS. C CAN SATISFY THIS REQUIREMENT VIA UNIONS.

sure that intb ends up on the right boundary alignment.

Some of you may be saying, "Sure that's fine, but why not shift things around so that you don't waste any space?" That doesn't quite work either as can be seen when I printed out the size of huh2. It turns out to be the same size that huh was. Think about it—could this be a coincidence because of an inefficient or lazy compiler or is there some substance to it? The answer must be the latter since an array of type huh2 must be able to be put in storage in such a way that all huh2 intb variables would be word aligned.

Going one step further, this means that every structure also has an alignment requirement, a fact that is not readily apparent. Note that I'm not talking about structure tags here since they do not use any execution time memory; however, they will conform to these requirements and produce the correct sizes and offsets if interrogated.

In fairness, it is worth mentioning that not all CPUs have this type of alignment requirement. Some enforce alignment only to even addresses, others by multiples of 2, 4, 8, . . . , bytes. And others, even though they would like you to use the suggested alignment, have a lax but less efficient alignment (that is, none). Microsoft C (MSC) Version 5.1 supports this mechanism if you find that there are too many wasted holes in your structures and space is at a premium. You can enable this by using the /Zp command-line switch to CL or by using the #pragma pack statement within your code. These two techniques are explained in the Microsoft C Optimizing Compiler Version 5.1 *User's Guide* in section 3.3.15.

Reading/Writing Structures

Very often it is necessary to write a structure to a disk file or RAM or perhaps through a pipe or other interprocess communications facility available under such operating systems as OS/2, XENIX, or UNIX. Because the program "distributing" the structure may not have been compiled with the same packing options or may not even be running on the same machine that is responsible for processing any of the data within the structure, it is advised that you should always create a dummy stub program that will dump the values of the structure before assuming that the data within it must be correct. Remember—

alignment restrictions may create holes; holes change the physical layout of the structure. You cannot make assumptions without seeing the source code and/or compile options used.

Getting the offsetof

It is generally considered bad programming style to include hard-wired constants within your code. And because of potential structure holes, it is neither wise nor portable to use hard-wired constants to represent structure member offsets. For instance, returning to **Figure 5**, because of the inability to track it down easily, it would not be reasonable for you to keep track of how many bytes into huh that intb was located. Instead you should use the offsetof macro.

The offsetof macro is available with most recent compiler releases and can be found in <stddef.h>. It takes two arguments: a type representing a structure and a member of that structure. The result is the offset of the member in bytes from the beginning of the structure. I have included one possible implementation of the macro that will calculate a member's offset; it should work on most machines (see the #define offsetof in **Figure 5**). The basic idea behind the way the macro works is to pretend the structure begins at address zero [hence the (type*)0] so that any reference to any member from address zero must give its true relative offset in bytes. You are encouraged to type in **Figure 5** to experiment with the structures, especially lines 32–34.

Forcing Type Alignment

Up to this point I have been emphasizing the use of structures. Another popular C construct along these lines is the union construct. It is similar to a struct in its syntactical nomenclature but different in its semantics. Unfortunately, its function within C programs is often misunderstood.

Some of you are probably curious about my statement that unions are misunderstood, so let me define what a union is and what a union isn't (or rather, what it isn't supposed to be). A union is a variable that has the ability to hold one, and only one of many different types of named objects (that is, objects with overlapping storage) at a given point in time regardless of the types of those objects. Just think about that for a moment. Ramifications of this definition are as follows:

- The purpose of a union is to allow for the reuse of a memory location or variable.
- The memory spaces for all members of the union begin at the same address.
- The sizeof (a union) == sizeof (the union's largest member).
- The value of only one of the union's members may be stored within the union.
- As a style issue, it is desirable that all the union's members are related to a specific piece of program logic using it.

It should now be clear that unions can suffice as a method of forcing type alignment and for redefining types. Redefinition of types is described in the next section, and type alignment is described as follows.

Very often in systems programming an operating system resource or request call, a device driver, or a specific hardware device (perhaps using DMA) may require that data transferred to or from it have specific alignment requirements. Luckily C, the de facto systems programming language, can satisfy this requirement via unions.

For example, let's make a relatively safe assumption—

ANOTHER POPULAR C CONSTRUCT IS THE UNION CONSTRUCT. IT IS SIMILAR TO A STRUCT IN ITS SYNTACTICAL NOMENCLATURE BUT DIFFERENT IN ITS SEMANTICS. A UNION IS A VARIABLE THAT HAS THE ABILITY TO HOLD ONE, AND ONLY ONE OF MANY DIFFERENT TYPES OF NAMED OBJECTS AT A GIVEN POINT IN TIME REGARDLESS OF THE TYPES OF THOSE OBJECTS.

Figure 6A: Example of a Nonstandard Use of Unions

```
1       #define getmemory()     20 /* e.g. simulate RSX call */
2
3       char    *
4       toktoptr(token)
5       short   token;
6       {
7           union {
8               struct {
9                   short offset;
10                  short segment;
11              } segoff;
12              char    *ptr;
13          } convert;
14
15          convert.segoff.segment = token; /* get segment number */
16          convert.segoff.offset = 0;  /* offset is zero within the
                                           segment */
17
18          return (convert.ptr);       /* assume segment &
                                           offset overlay ptr
                                           perfectly */
19      }
20
21      main()
22      {
23          int token = getmemory(); /* RSX memory call */
24          char    *p = toktoptr(token);
25
26          printf("%d\n", p);
27      }
```

Figure 6B: Example 6A Rewritten in a Standard Manner

```
1       #define getmemory()     20 /* e.g. simulate RSX call */
2
3       char    *
4       toktoptr(token)
5       short   token;
6       {
7          return ((char *)(((long)token) <<
                        (sizeof(short) * 8)));
8       }
9
10      main()
11      {
12          int token = getmemory(); /* RSX memory call */
13          char    *p = toktoptr(token);
14
15          printf("%d\n", p);
16      }
```

IF ANY INITIAL SEGMENT OF OVERLAPPING STRUCTURES WITHIN A UNION IS THE SAME, A WRITE TO ONE OF THE OBJECTS CAN BE FOLLOWED BY A READ FROM ANOTHER OBJECT.

that most of the time an operating system or device is to be word aligned (which taken one step further on most machines usually means that there should be an even address boundary). Let's also assume that the info is to be passed into a hardware device expecting 6 bytes of information. Simply coding something like char info[6]; may or may not necessarily work since C doesn't guarantee that info will begin on an even address—therefore you've got a 50 percent chance of getting it right and even worse odds for tracking down the bug when code is changed 6 months down the road.

If you haven't guessed already, the way to ensure that info can be word aligned is through something like:

```
union device_data {
int dummy; /* alignment */
char    info[6];
};
```

Since dummy is word aligned (why?) and the address of each member of a union begins at the same location, info must also be word aligned! Of course, your code never needs to be concerned about dummy again since it has served its purpose. All very elegant, no?

Redefinitions: Abuse of Unions?

Because C does not enforce one facet of unions, namely that there may only be one object in use at a time, this is left up to the programmer. In other words, you are only supposed to take from a union what you put into it; therefore, if you assign to a particular member, you are only supposed to retrieve from that same member until you assign to another member. The problem here is that C leaves this up to you, at least at a syntactical level. For instance, given the declarations:

```
union example {
  int     i;
  double  d;
  } ex;
int     j;
```

probably no C compiler will prohibit you from coding:

```
ex.d = 999.999;
j = i;
```

even though you'd be guaranteed that j wouldn't hold anything valid, unless perhaps you were interested in getting a random number. However, if you are careful, this side effect can be turned to occasional

advantage, what I'd like to term a nonportable nicety—something I don't recommend that you use, but if you do, I hope that you document the fact and clearly understand what it is you are doing.

For example, I was working on a consulting project last year that required use of the Intel® RMX operating system. The operating system was running on an 80386, which is a segmented architecture machine. At one point in the project, we needed to obtain some dynamic memory. For some rather obtuse reason, it was decided that none of the standard C run-time routines such as malloc would be appropriate for what we wanted to do. Instead we issued an RMX system call in order to obtain the memory. The problem we were faced with then was that the call returned an entity called a token, which, every time we called it, turned out to be the beginning of a segment. However, the token only contained a 2-byte segment number and not the 2-byte zero offset as well, and we needed to use the token as a character pointer.

The way we resolved the problem was by using code similar to that listed in **Figure 6A**. However, given the reasons that have already been mentioned, this technique is not guaranteed to work under all C compilers. Furthermore, we're making the assumption that pointer = = int is valid and to make matters worse, our ordering of the segment and offset variables are also system-dependent since byte or word ordering is hardware-dependent. As you can see, all in all it is not a very healthy affair.

Knowing this, many of you probably are agreeing but also noticing how easily the conversion routine functioned. Before this looks too enticing, let's look at the preferred method of redefining types: the cast operator. **Figure 6B** has a version of toktoptr that uses casts.

Figure 7: Example of Combined Unions and Structs

```
1       #include <stdio.h>
2
3       union codedrecords {
4           struct {
5               int     a;
6               float   b;
7           } type1;
8           struct {
9               int     c;
10              long    d;
11          } type2;
12          struct {
13              int     e;
14              short   f;
15          } type3;
16      };
17
18      struct genericrecord {
19          int     recordtype;
20          union codedrecords cr;
21      };
22
23      struct genericrecord gr;
24
25      main()
26      {
27          /* ... */
28
29          switch(genericrecord.recordtype) {
30          case TYPE1:
31              /* code that uses genericrecord.cr.type1.? */
32              break;
33
34          case TYPE2:
35              /* code that uses genericrecord.cr.type2.? */
36              break;
37
38          case TYPE3:
39              /* code that uses genericrecord.cr.type3.? */
40              break;
41
42          default:
43              fprintf(stderr, "Invalid Record Type!!!");
44              exit (1);
45              break;
46          }
47
48          /* ... */
49      }
```

This may look rather ugly and the cast code is also making some nonportable assumptions. The best thing I can say is that you must recognize that using unions for redefinitions instead of as reusable storage may work, but the method is simply not valid and is totally nonportable even among compilers on the same machine. The chances are also high that it will produce incorrect code with many of the optimizing compilers currently on the market. The cast, although also system dependent in many ways, is generally the less problematic method, and besides it's "legal" C.

The reason I am explaining this is not to teach you a new trick but to make you aware of a bad coding practice and to prepare you for the unexpected if it ever pops up while you are maintaining a piece of code. Also, if you do decide to use this technique, you will know about the problems that can crop up because of it.

Unions in General

A more general use of unions is for building coded records. Since a union is capable of

declaring a valid identifier, it may appear within a structure (and vice versa). An example of this occurs in **Figure 7**, which illustrates a case in which a certain record is obtained, perhaps through a communications line, and needs to be processed. Each record has a code recordtype signifying what shape it is to take on so that it can be directed to the appropriate case of the switch statement that knows about it and the names of its members.

This is a very elegant and easily maintainable feature of C. I was able to create a data structure that did not have to use any unnecessary data space nor did I need to resort to playing any games with pointers to multiple structures to be able to use the correct type structure.

Note that contrary to what was discussed in the preceding section, you can reuse the first int of codedrecord's members without harm. This is an extension as a special case. That is, if any initial segment of overlapping structures within a union is the same, a write to one of the objects can be followed by a read from another object. For instance, if the logistics behind the construction of type1 and type2 data structures were similar, it would be valid to store a value in a and then use c with absolutely no ill effects because a and c are of the same type and located at the same offsets. As a style issue it may even be better to take a, c, and e out of union and make it a single entity in genericrecord. However, this would depend entirely on the logical connection between these identifiers as well as the logic behind the code that uses them.

One last point about unions: though the draft proposal of ANSI C now allows for union initialization to occur in C code, the initialization must be represented as a constant expression and the initialization expression can only initialize the first member of the union. This is something to be aware of if your compiler does a conversion to a different type and quietly performs the assignment, perhaps by truncation. You may have to swap the order of the variables in the union in some cases to ensure proper initialization.

The typedef

Before going into some explanations and examples of typedefs and structures, let me get two common misconceptions about typedefs and one source of confusion out of the way. First, even though the typedef keyword is syntactically categorized as a storage class specifier, it is a misnomer and does not allocate any execution time storage. It is only categorized as such for notational convenience.

Second and more pertinent to our discussion is that typedefs do not create or define a new type, as the keyword may imply. What they do is allow the programmer to create a new name for a base or derived type that already exists. In other words, a typedef allows you to create a synonym for the type.

Furthermore, the synonym is placed into the general name space (an area used to categorize identifiers) of the compiler's symbol table. The general name space holds function and most variable and enumeration names as well. The end result is that typedef names are practically no different from any others. That is to say, they serve to lock into a name, much like a struct or union tag does. In this way, typedef names can be used to classify identifiers declared with them.

Finally, an additional source of confusion with typedef is that you are not allowed to place any storage class specifiers within the type name being created during a typedef statement. This is because you are not allowed to have more than one storage class specifier within any of your declarations. Some would say this is a blessing in disguise since hiding storage class information within a typedef could result in code that is harder to maintain and write than it should be. In other words, it wouldn't be directly obvious from the declaration of a variable based on the typedef what all its attributes are. It's a thin line since I'm not sure what an object-oriented programmer would say about this abstraction.

While on this subject, one more thing to remember is that while you may not have storage class specifiers associated with the typedef name, you can use the const and volatile type qualifiers. However, if you use *incremental* typedefs, only one appearance of a given qualifier is allowed to be applied to the previous typedef declarator. I could be wrong, but I suspect that many compilers would have a problem issuing a proper diagnostic error about this.

Allocating Structures

There are two common techniques for allocating a structure. They involve the use of #define and typedef. Both can be used to achieve similar goals; however, their syntax, and amazingly their semantics, are very different. If possible the allocation should be performed via typedef. Let's see why.

Since K&R never elaborated upon the use of typedef, and since many of the earlier non-AT&T® C compilers usually didn't support it (possibly for that reason), until the past few years typedefs were either ignored or misunderstood. The usual method for allocating a structure was to use the #define directive. For instance, before

the void keyword became popular on all compilers, it was usually suggested to include

```
#define void int
```

in all programs, usually via a programmer supplied include file, say mydefs.h, that functioned similarly to the macros currently found in stddef.h. However, even though this worked without a hitch (try compiling the code listed in **Figure 8**), it will create problems with only slightly more complicated type specifications. Do not get off on the wrong track by using old code as an example.

The problems appear because #define is concerned only about textual substitutions of strings or tokens of strings. And this is fine since that is the duty of the preprocessor. However, it doesn't know or care about C syntax. The preprocessor is only responsible for accomplishing the text substitutions; as long as it is fed a valid C program, it must emit a valid C program. On the other hand, since the compiler does classify typedef names, it can and will enforce type checking of expressions involving typedefs.

If we use some of the examples presented by K&R, whose interpretation is more or less left as an exercise for the reader (even in the new revised second edition), we can investigate why #define will fail. Using their declaration:

```
typedef char  * String;
```

the equivalent #define would be:

```
#define MAXLINES (5)
#define String  char *
```

If we also use their sample invocations of String:

```
String p,
lineptr[MAXLINES],
alloc(int);
```

a first glance might not indicate any problems. However, if we follow through the text substitutions, the line gets passed (at least as a transparent step) to the C compiler as:

```
char * p, lineptr[5],
alloc(int);
```

certainly not what was desired.

What happened is that only p was declared as a char *. And lineptr and alloc were declared as an array of char and a function returning char, respectively. We were trying to obtain:

```
char *p, *lineptr[5],
*alloc(int);
```

Besides the #define problem, this also points out a pitfall of using a multideclarator declaration. You were warned of this in "A Guide to Understanding Even the Most Complex C Declarations."

As another simple case in which #define could not possibly work, consider:

```
typedef int * array20[20];
```

There's just no way to coerce it to produce:

```
array20  samplearray;
```

Finally, relating all this back to structs and unions, there are also the type checking capabilities of the compiler that are a concern to us. For instance, using K&R's Treenode exam-

Figure 8: Examples Comparing #define and typedef

A

```
1       #define v int   /* don't use void in this example since
                                    it's a keyword */
2
3       main()
4       {
5           int x;
6           v   y;
7
8           x = y;
9       }
```

B

```
1       typedef int v;  /* don't use void in this example since
                                    it's a keyword */
2
3       main()
4       {
5           int x;
6           v   y;
7
8           x = y;
9       }
```

THOUGH THE DRAFT PROPOSAL OF ANSI C NOW ALLOWS FOR UNION INITIALIZATION TO OCCUR IN C CODE, THE INITIALIZATION MUST BE PRESENTED AS A CONSTANT EXPRESSION AND THE INITIALIZATION EXPRESSION CAN ONLY INITIALIZE THE FIRST MEMBER OF THE UNION. THIS IS SOMETHING TO BE AWARE OF IF YOUR COMPILER DOES A CONVERSION TO A DIFFERENT TYPE.

THE BASICS OF STRUCTURES AND UNIONS ARE WELL DEFINED AND MOST PROGRAMMERS ARE CAPABLE OF USING THEM WITH REASONABLY GOOD RESULTS. HOWEVER USING THEM TO THEIR FULL CAPACITY REQUIRES A BIT MORE KNOWLEDGE THAN COMMONLY AVAILABLE. THIS ALSO APPLIES TO TYPEDEF, WHICH HAS OFTEN BEEN EITHER NEGLECTED OR MISUNDERSTOOD.

ple, an equivalent define (ignoring Treeptr for our purposes) would be:

```
#define Treenode struct {\
char  *word;\
int   count;\
}
```

Invocations of this might appear as follows:

```
Treenode  tn1, tn2;
Treenode  tn3, *ptn;
```

However, although tn1 and tn2 are assignment and member compatible, they have nothing to do with other declarations that might use Treenode even if they are pointers such as ptn. Therefore:

```
tn1 = tn2;
tn1.count = tn2.count;
```

are fine, but:

```
tn1 = tn3;
ptn = &tn1;
```

are undoubtedly syntax errors.

To make matters worse, if we use the style constraint of one declarator per declaration:

```
Treenode  tn1;
Treenode  tn2;
Treenode  tn3;
Treenode  *ptn;
```

none of these have the ability to have anything to do with the others—if we perform the preprocessor substitution, we get four unnamed yet distinct structure tags. The compiler doesn't care that they might all look the same. Every invocation of Treenode will create a new structure.

If instead we (properly) use a typedef:

```
typedef struct {
char  * word;
int count;
} Treenode;
```

all the executable statements above are valid. The compiler only creates one reference to the struct in the symbol table and after that everything falls into place including type checking. All is now syntactically sound. Since a typedef is C code, all the Treenode invocations will have the same type.

Proper Perspective

The basics of structures and unions are well defined and most programmers are capable of using them with reasonably good results. However using them to their full capacity and in a portable way requires a bit more knowledge than commonly available. This also applies to typedef, which has often been either neglected or misunderstood (*typedef becomes very important under the OS/2 and Presentation Manager programming environments—Ed.*). Since a structure is the main data construct in C (as well as in many other languages), now that you have these facts under your belt, you should have a better understanding of the C language. You can use it to your advantage in a wise, efficient, and portable manner.

Skipping over any detail of a language specification is a mistake. Certainly some things are useless and awkward, but it's disappointing to think that many programmers avoid parts of languages simply because they are somewhat complicated. Though C is many times a very terse language and often cryptic, an understanding of its more subtle and complex points opens up C's unique power.

My own experience with C, as well as that of other programmers, clearly shows that continuing to use it while not understanding it is not helpful. This usually results in slower development and mistakes that will be costly. If you stick with it and put in that extra energy to understand its advanced syntax and especially the underlying philosophy, you will be able to tap into its most powerful and advanced features. ❑

Whitewater's Actor®: An Introduction to Object-Oriented Programming Concepts

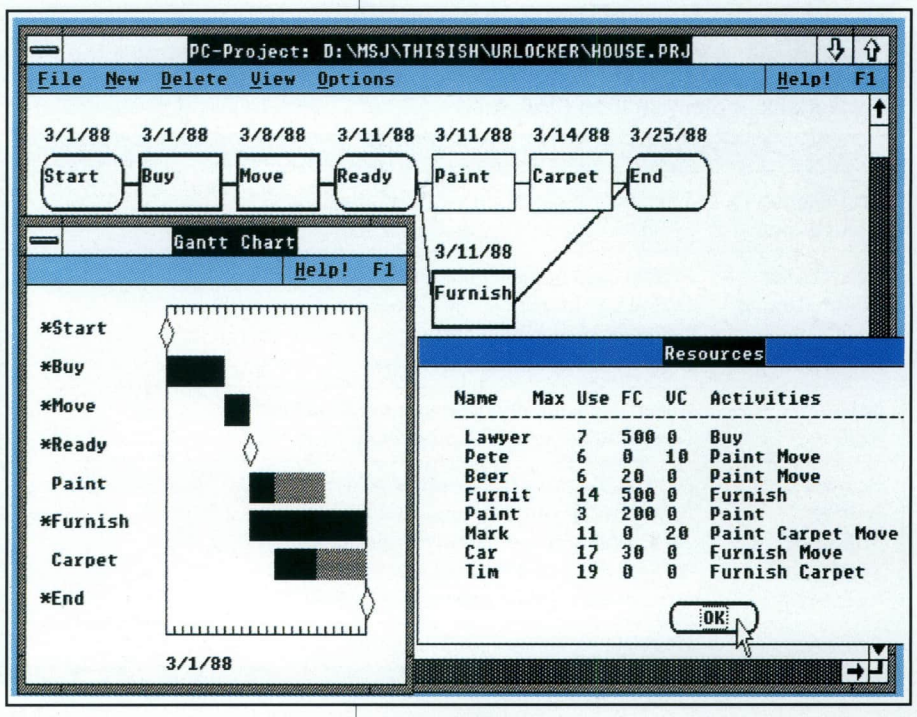
33

Zack Urlocker

Object-oriented programming techniques are not new, but they are becoming more popular as programmers tackle increasingly complex projects. Object-oriented programming can help simplify the development of elaborate programs by breaking them down into logical objects that manage their own behavior and hide internal complexity. Windowing applications in particular are easier to develop and maintain if object-oriented programming techniques are used. Although object-oriented programming is best done in a pure object-oriented language, such as Actor® or Smalltalk, it can also be used in other languages.

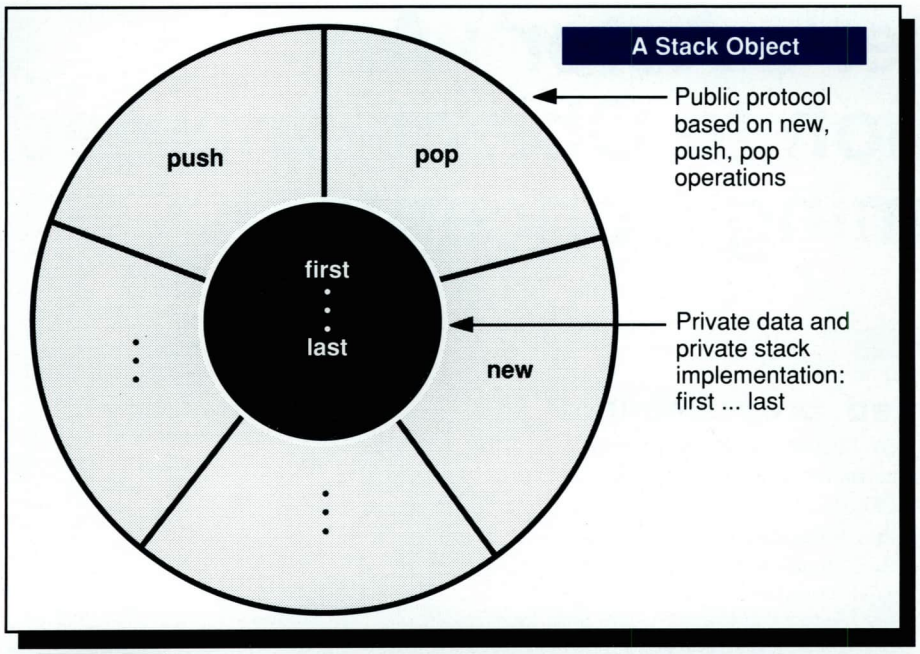
This article provides an overview of object-oriented programming, demonstrating how it can simplify the development of Windows programs. If you haven't done much programming in Microsoft® Windows, some knowledge about object-oriented programming can help you understand how Windows works. Most of the sample code is taken from PC–Project, a critical path project management program that I wrote in Actor.

PC–Project lets you model a real-world project by creating milestones and tasks, assigning times, and determining the critical path of the project. The critical path shows which activities, if delayed, will cause a delay in the overall completion of the project. PC–Project also lets you allocate resources and costs to tasks and create a Gantt timeline chart of the project. PC–Project is available with complete source code from various Bulletin Board systems or directly from the author. **Figure 1**



▲ **Figure 1** **The PC–Project application running under Windows, displaying a PERT diagram, a Gantt Chart, and the resources required for the project.**

Zack Urlocker is manager of developer relations at The Whitewater Group®, the developers of Actor.



▲ **Figure 2** **A stack object, such as the one shown here, illustrates the separation between the publicly available protocol (new, push, pop) and the private stack implementation (first, last).**

Figure 3: Classes in PC-Project

ProjWindow	Window that can display a PERT diagram
GanttWindow	Window that can display a Gantt chart
ActivDialog	Formal class of dialog box for activities
MStoneDialog	Dialog box for editing Milestones
TaskDialog	Dialog box for editing Tasks
PERTDialog	Dialog box for editing PERTTasks
Network	Generic network of nodes with a start and end
Node	Generic node capable of connecting itself
Project	Network that knows the critical path method
Activity	Node with an earlyStart and lateFinish
Milestone	An activity that uses no time or resources
Task	An activity that has time, resources, and cost
PERTTask	Task where the time is estimated by PERT
Resource	Used by a task; has a name and cost

IN AN OBJECT-ORIENTED LANGUAGE, BOTH DATA AND OPERATIONS THAT WORK ON THAT DATA ARE COMBINED INTO A SINGLE LOGICAL UNIT KNOWN AS AN OBJECT.

shows the PC–Project application running under Windows.

What Is Object-Oriented ?

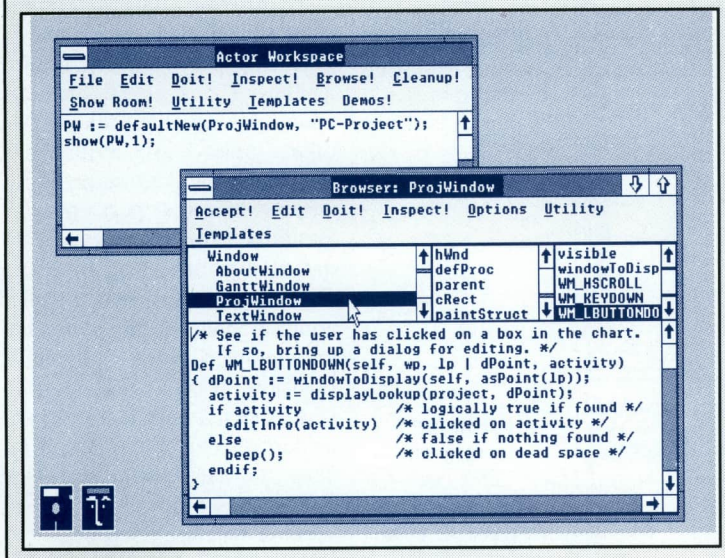
In traditional procedural languages like C or Pascal, the programmer defines data structures and writes functions and procedures to operate on the data. Although normally a correspondence exists between which functions operate on which types of data, most procedural languages offer no formal support for this correspondence; it is entirely the programmer's responsibility to manage such an abstraction.

In an object-oriented language, *both data and operations that work with that data* are combined into a single logical unit known as an object. Dividing a program into objects encompassing both data and operations makes the program more closely represent the logical design that is being implemented. As a result, object-oriented programs are generally easier to understand and maintain than procedural programs.

Object-oriented programming encourages the creation of abstract data types; that is, the implementation of an object is referred to abstractly and is encapsulated by high-level operations. Objects have a clear division between public protocol and private implementation. For example, we might have a stack object that defines a public protocol based on the push and pop operations. The stack may be implemented as an array with variables that maintains the first and last positions, but this representation would be considered private. By adhering to the public protocol, we can change the implementation of stacks, say, to linked lists, without having to rewrite any of our code. **Figure 2** illustrates a stack object and the separation of public protocol and private implementation.

Programming in an object-oriented language involves creating objects and sending them commands or messages to do things. For example, we can create a window with the caption Sample and then show it on the screen. In Actor, this is done using the messages shown below:

A Brief Introduction to Actor



▲ **Figure A**

```
Def run(self, resource, parent | retValue)
{
  retvalue := runModal(self, resource, parent);
  if retvalue == -1
        beep();
        errorBox("Warning", "Low on memory!");
  endif;
  ^retValue;
}
```

▲ **Figure B** **A sample method for the Dialog class that checks Windows memory when running a dialog box.**

Actor, designed and created by The Whitewater Group, is an object-oriented language that allows you to create standalone Microsoft Windows applications. Since Actor is an interactive environment that runs under Windows, you can type statements in the workspace window and get immediate feedback. Windows, menus, and dialogs can be created and modified directly in the development environment. Actor has a source-level debugger and an inspector that lets you debug programs interactively.

Code is written in a browser, a special editor that lets you create new classes and compiles methods as you write them. The compiler translates Actor statements into a low-level format used at run time. The browser also shows the hierarchy of classes in the system. **Figure A** shows a screen shot of a browser in the Actor development environment.

Actor is a pure object-oriented language. That means that everything in the system is an object and all operations are performed by sending messages to objects. Even an expression like 5 * X is a message to X to multiply itself by 5. In hybrid languages such as C++, some things are objects that respond to messages and others are not.

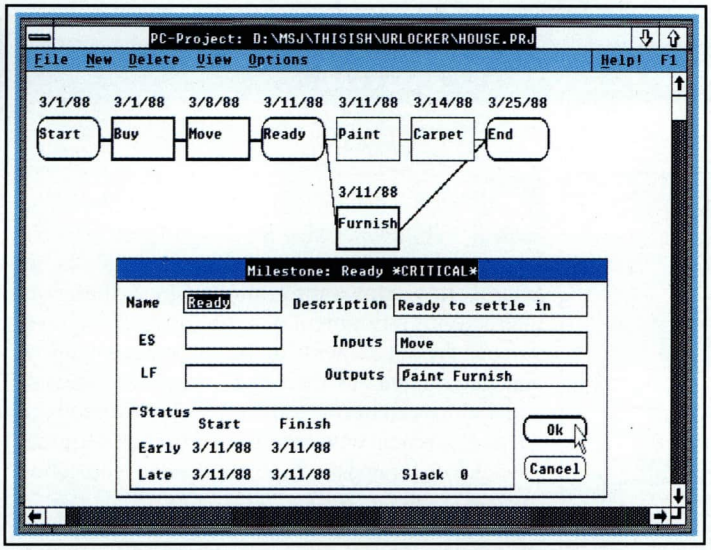
Actor's syntax is a combination of Pascal and C. For example, the := symbol is used for assignment, whereas curly braces, { }, denote a block. Comments appear within C-style delimiters, /* and */. Actor includes an if/else statement, a case/select statement, while loops, and so on.

Method definitions begin with the keyword Def, followed by the name of the method and parameters within parentheses. The first parameter in a method definition is the receiver of the message and is always called self. The vertical bar, |, is used to separate arguments from temporary local variables. No type definitions are necessary, since Actor determines the class of an object at run time. The sample method definition shown in **Figure B** has two arguments, resource and parent, and one local variable, retValue.

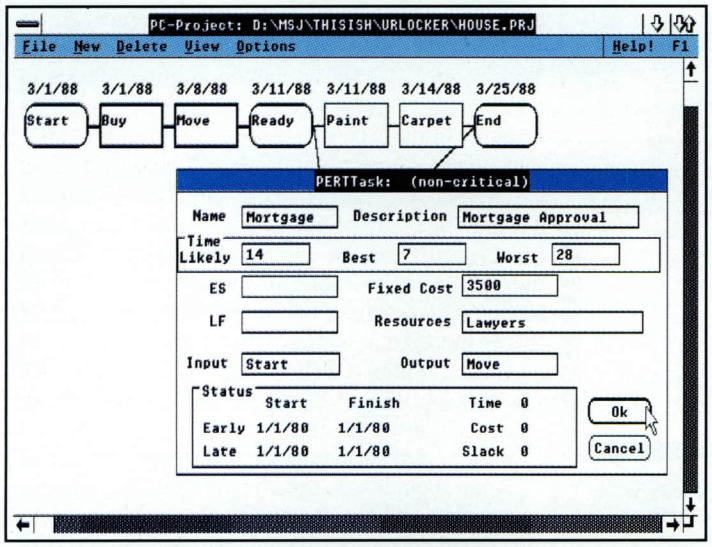
The caret, ^, is used to return a value. If no value is explicitly returned, the receiver of the message, self, is returned.

This very brief view of Actor indicates that it has a relatively standard procedural language syntax. It differs from procedural languages in the way it is programmed. Programming in Actor can be summarized as the process of creating objects and sending messages to the objects. Messages provide the objects with the information necessary to execute a method. In a sense, the message is matched to a method already defined within the object. Actor is dedicated to the goal of creating high-level applications that hide complex underlying issues.

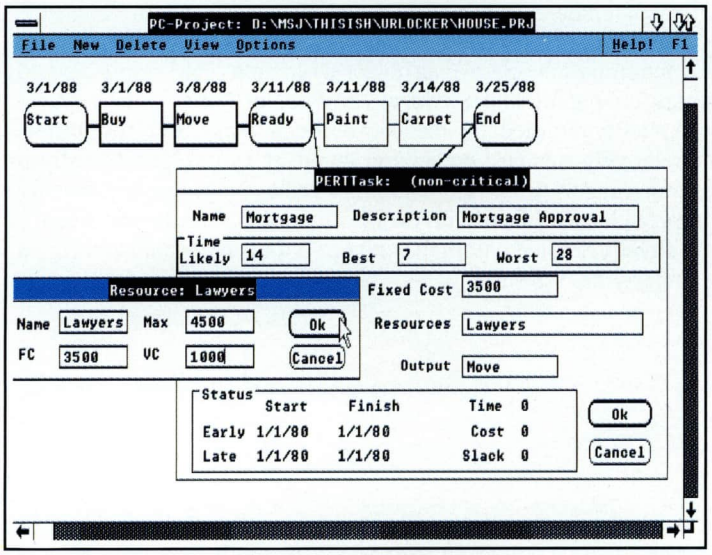
For more information about Actor, contact:
The Whitewater Group
Technology Innovation Center
906 University Place
Evanston, IL 60201
(312) 491-2370



This screen shows the information related to the critical milestone Ready. Note the output to both Paint and Furnish tasks.



Adding a noncritical PERTTask called Mortgage to HOUSE.PRJ. Its input is from the milestone start and its output is to the task Move.



Defining additional parameters for the resource Lawyers.

```
/* create it */
W := defaultNew(
    Window, "Sample");
show(W, 1); /*display it*/
close(W);   /* close it */
```

In this case Window is a predefined Actor class or type of object already in the system. Therefore dozens of lines of code are eliminated that would otherwise have to be written in C to accomplish the same thing without affecting performance. Window objects have private data that manage their location, size, caption, parent, handle, and so on. Window objects know how to create themselves, position themselves on the screen, resize, close, and so on, as part of their public protocol. Thus, the statement close(W); is a message being sent to the W window to close itself.

Although it may seem that messages are the same as function calls in other languages, they are not. The receiver of a message, which is the first parameter after the parenthesis, determines how to respond. Different classes of objects can respond to the same message in different ways, a language characteristic known as polymorphism.

For example, if W were a member of the ProjWindow class that received a close message, it would first check to make sure that the current project was saved. If W were a member of the GanttWindow class, it would inform its parent window that it was closing. In fact, W doesn't even need to be a window at all. Other objects, such as files or communication channels, could respond to a close message.

Polymorphism is achieved in Actor by defining a method, with the corresponding message name, for the class. A method definition is similar to a function definition in other languages. Polymorphism allows you to

write more general, reusable code, since you don't have to worry about what types of objects you are dealing with, as long as they follow the same public protocol. You can let the objects themselves manage the details of implementation.

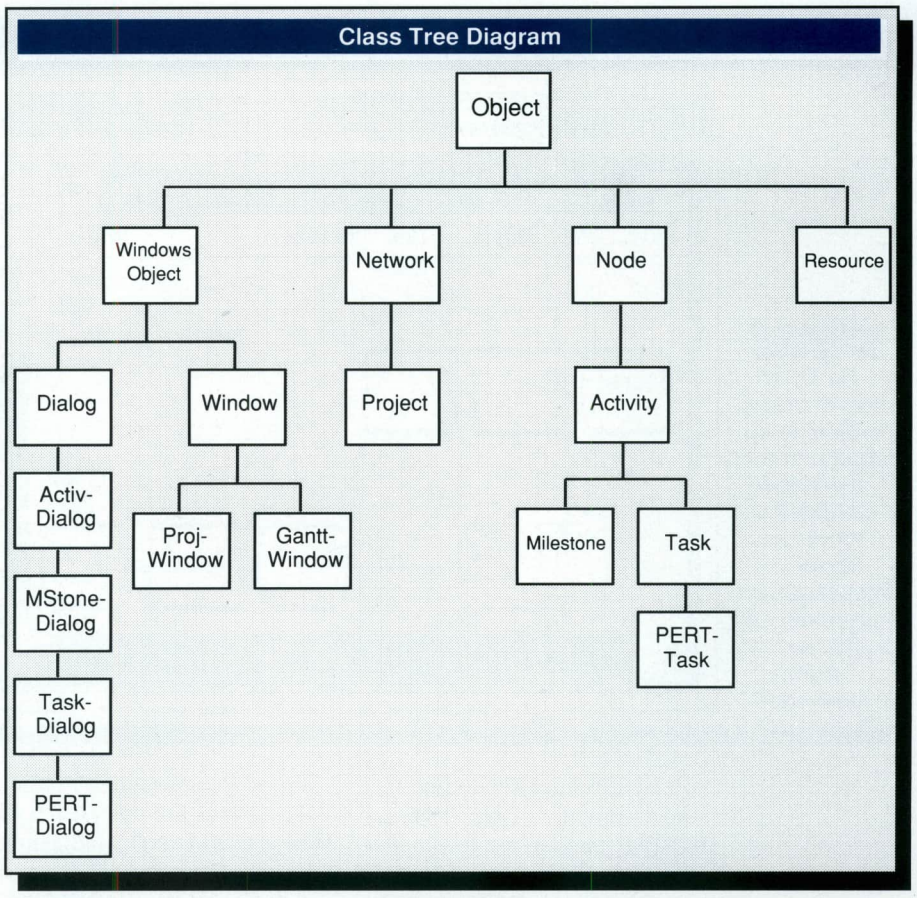
Inheritance

Objects are organized hierarchically in classes. Most object-oriented languages include classes for things like arrays, files, strings, stacks, and queues. In Actor there are also predefined classes for dealing with such Windows entities as text windows, dialog boxes, and scroll bars.

We can create new subclasses that inherit all the characteristics of an existing class. For example, in PC–Project the ProjWindow class descends from the Window class and adds to it functionality related to project management. Because of this inheritance capability, classes are much more powerful than data types in other languages. The advantage is that all the generic windowing capabilities, like resizing, displaying, and dragging work properly without your having to write or test a single line of code. Using inheritance you focus on those parts of the program that are application-specific instead of constantly "reinventing the window," so to speak.

Inheritance encourages the development of small, reusable classes that become building blocks for more sophisticated classes. This approach results in less code to maintain and test and more rapid development from prototype to final application.

In PC–Project I used inheritance to group the characteristics that are common to the dialog boxes used for editing activities and for the activities themselves. **Figure 3** describes the classes in PC–Project. **Figure 4** shows how they are related in the class tree.



▲ **Figure 4** **The class tree diagram illustrates the classes in PC–Project. Note that all classes descend from Object.**

How does object-oriented programming work with procedural languages? Purists maintain that object-oriented programming is possible only in a late-bound language that has a class facility and inheritance. However, object-oriented design techniques are applicable in many languages. For example, both Ada and Modula-2 include facilities that allow the creation of abstract data types. You can also add object-oriented extensions to C by using a C++ preprocessor. Any program can be designed, if not implemented, as logical objects that encompass data and operations with a clear separation of public protocol and private implementation.

Many programmers are

MOST OBJECT-ORIENTED LANGUAGES INCLUDE CLASSES FOR THINGS LIKE ARRAYS, FILES, STRINGS, STACKS, AND QUEUES. IN ACTOR THERE ARE ALSO PREDEFINED CLASSES FOR DEALING WITH SUCH WINDOWS ENTITIES AS TEXT WINDOWS, DIALOG BOXES, AND SCROLL BARS.

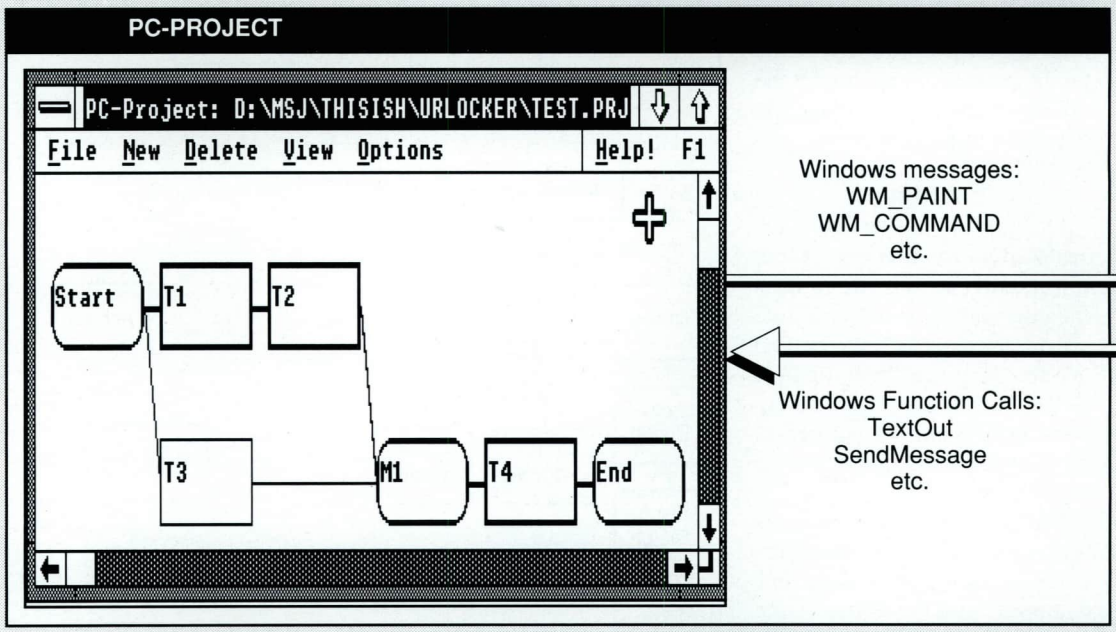


Figure 5
An Actor ProjWindow object is an abstraction of a window managed by Windows. Windows messages are automatically translated into Actor messages.

pleasantly surprised to find that object-oriented languages encourage them to use techniques they have been faking for years in other languages. Through the rest of this article I encourage you to consider how object-oriented techniques could be used in your current language.

Windows

Windows, like object-oriented languages, operates on a message-passing paradigm. Windows is an event-driven system, meaning that programs respond to events that the user or other programs initiate. These events correspond to actions like pressing a key, clicking the mouse, or selecting a menu item. Whenever an event occurs, Windows sends a message to notify the program.

More specifically, when the user presses a key, for example, Windows sends a WM_KEYDOWN message with the virtual key code of the key that was pressed. The "WM" is mnemonic for "Windows message." Other Windows messages are WM_COMMAND (indicating the user selected a menu command), WM_LBUTTONDOWN (the user clicked the left mouse button), WM_VSCROLL (the user clicked in the vertical scroll bar), and WM_PAINT (Windows wants the window to redraw itself).

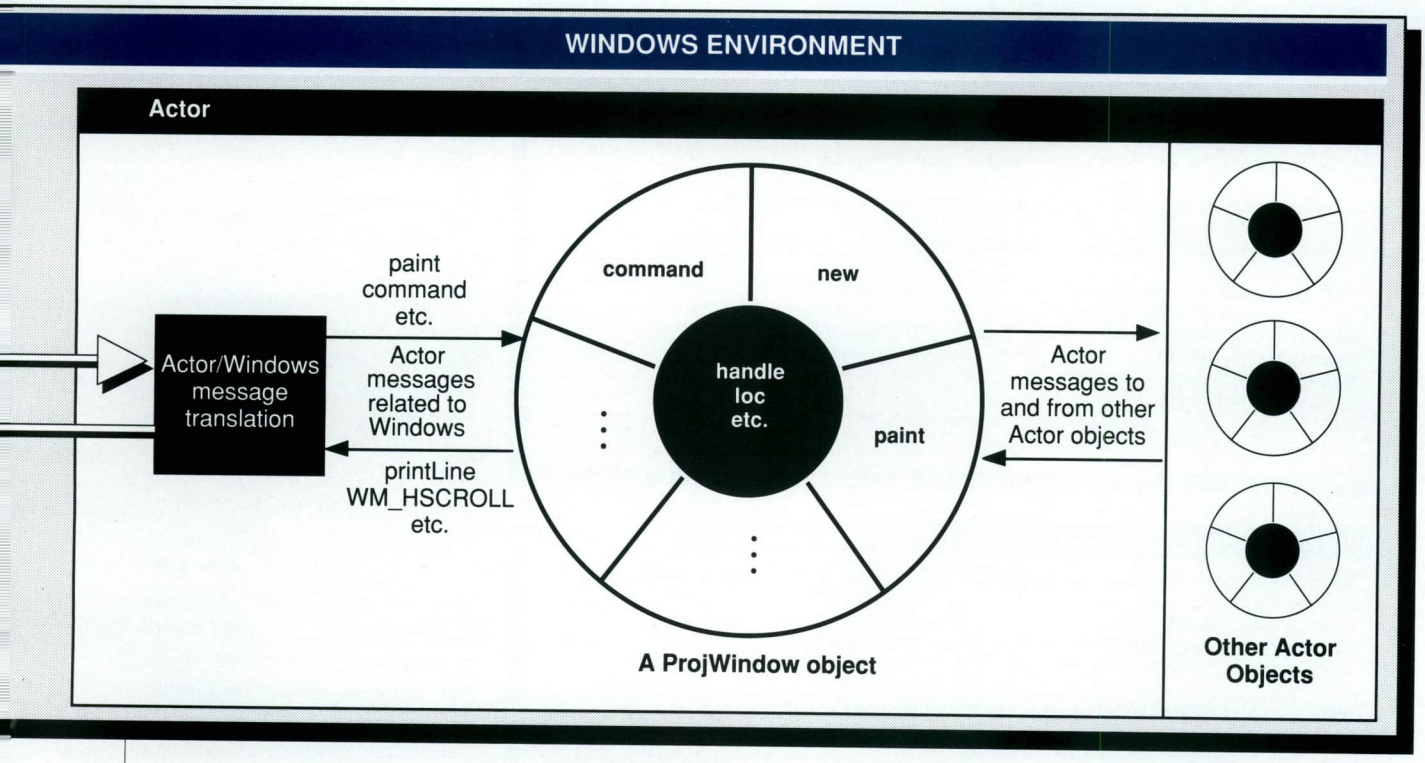
Windows messages are always sent with two parameters to convey additional information. These are known as the word parameter, or wParam, and the long parameter, or lParam. The wParam contains a 16-bit word value; the lParam sometimes contains a 32-bit long pointer to other data.

If an application has multiple windows, as PC–Project does, the Windows messages are sent to the appropriate window. For example, if you press the F1 function key while the Gantt window has the focus, a message is sent to that window; the main window is not informed. Making windows responsible only for their own events simplifies application development.

Actor Objects

The relationship between Actor objects and Windows entities is similar to the relationship between file variables and files in most high-level languages. For example, in Pascal, you can declare a variable of type File. To use the variable, however, you must assign it the name of an actual disk file. The file variable is an abstraction of the physical file on disk. In the same way, Actor objects belonging to classes like Window and Dialog are abstractions of underlying areas of memory managed by Windows.

Although both Actor and Windows send messages, the messages are processed separately. Unlike Windows messages, Actor messages are not queued at all and are therefore very efficient. The main function in a Windows application, called WinMain, normally



includes a very short loop that translates and dispatches Windows messages. The application must also define a WndProc function that processes the messages.

From an object-oriented perspective, it is the window itself that responds to the messages. After all, a window is not just a data structure, it is both the data and the functionality. Thus, Actor manages the WinMain and WndProc functions, the Windows message queue, and other low-level details.

Actor automatically translates Windows messages into equivalent Actor messages, enabling you to process all messages in the same way. The Actor classes Window and WindowsObject define many high-level messages that hide the generic details of Windows programming and allow the programmer to concentrate on application specific behavior. **Figure 5** shows the flow of messages between Windows and Actor objects.

The WM_PAINT method defined in the Actor class Window, automatically locks down an area of memory known as a display context used for redrawing. It then calls the Windows function BeginPaint, sends an Actor paint message, calls the EndPaint function, and lastly frees the memory used for drawing.

The WM_PAINT method defined for class Window is shown in **Figure 6**. Since this method is inherited by all descendants of the class Window, they only need to define a higher-level paint method that knows how to redraw the contents of the window. The Actor paint message will be sent whenever Windows sends a WM_PAINT message.

In PC-Project, the ProjWindow class defines a paint method to redraw a network or PERT diagram of the project. The paint method, shown in **Figure 7**, includes no calls to Windows functions to manage the display context, since this job will be handled by the WM_PAINT method of the Window class.

The paint method loops through all the nodes in the project, and if a node is visible, it sends a draw message to the node. To determine where to draw a node, a pos message is sent to the node to get its logical display position. For example, the first node in a project is at the logical display position (0,0). This position is converted into the Windows coordinate (10,30) by sending a displayToWindow message to the window, with the logical display position as an argument.

Notice that while it is the node's responsibility to manage its logical location, it is the window's responsibility to determine if the node is visible and convert the logical display location into its own Windows coordinates. Again, the goal in object-oriented programming is to let the objects manage their own behavior as much as possible.

Figure 6: WM_PAINT Method Defined for the Class Window

```
/* Trap MS-Window's message to paint self, a Window.
   Sends a paint(self) message with the display context.

   self is a Window or ancestor of class Window.
   wParam and lParam arguments are ignored.
   hDC, hPS, lpPS are local variables.
     hDC : handle to display context for drawing
     hPS : handle to paint struct
     lpPS : long pointer to locked down paint struct
*/
Def  WM_PAINT(self, wParam, lParam | hDC, hPS, lpPS)
{
   hPS := asHandle(paintStruct);          /* get hPS */
   lpPS := globalLock(hPS);               /* lock down mem */
   hDC := Call BeginPaint(hWnd, lpPS);    /* get hDC */
   paint(self, hDC);                      /* send paint msg */
   Call EndPaint(hWnd, lpPS);             /* done painting */
   globalUnlock(hPS);                     /* free memory */
   ^0;                                    /* ok, return 0 */

}
```

Figure 7: The Paint Method Defined for the Class ProjWindow

```
/* Respond to MS-Windows messages to paint the window as a PERT
   (network) diagram.
   Draw each visible node in its proper position.
   Display the name and any other info required.
   Then draw the lines to connect the outputs of the node.

   self is the ProjWindow that receives the message.
   hDC, a handle to a display context, is sent as an arg.
   wPoint, x, y are local variables.
   aNode is the temporary loop variable for the do message.
*/
Def  paint(self, hDC | wPoint, x, y)
{
  do(nodes(project),
    {using(aNode)

     wPoint := displayToWindow(self, pos(aNode));
     x := x(wPoint);          /* horiz windows posn */
     y := y(wPoint);          /* vert windows posn */

    if visible(self, aNode)
      draw(aNode, self, x, y, hDC);
      drawTextInfo(self, aNode, x, y, hDC);
    endIf;

    /* always draw connections since they may be visible */

    drawConnections(self, aNode,x,y, getOutputs(aNode),hDC);
  });
}
```

Windows Messages

In addition to Windows sending messages, like the WM_PAINT message described above, window objects can send messages to other windows or even to themselves. For example, in PC–Project if the user presses the up arrow, the window will scroll up as necessary. This is done by trapping the WM_KEYDOWN Windows message and sending a WM_VSCROLL Windows message to the window.

When a scroll message is sent, the wParam argument indicates the scroll direction, defined by a constant such as SB_LINEUP or SB_LINEDOWN. (The "SB" is mnemonic for "scroll bar.") In the case of a scroll message, the lParam argument is ignored, so by convention we send a long zero, 0L.

To send a scroll message to a window using C, we call the sendMessage function with a handle to the window that will receive the message, the Windows message constant, and the wParam and lParam arguments, thus:

```
sendMessage(W.hWnd,
            WM_VSCROLL,
            SB_LINEUP,
            0L);  /* C */
```

Since Actor automatically translates Windows messages into Actor messages of the same name, there is no need to call the sendMessage function (though you could if you wanted to). Instead the scroll message can be sent directly to the W window:

```
WM_VSCROLL(W, SB_LINEUP,
           0L);  /* Actor */
```

Actor will then call the sendMessage function with appropriate arguments. Note that in Actor the internal representation of the window and its handle are hidden; the window object is responsible for managing its data and responding to all messages. Of course, you could define a higher-level method, perhaps called scrollUp, that would hide the details of the WM_VSCROLL message and SB_LINEUP constant.

Once you understand how Windows sends messages in response to user events, a basic principle of object-oriented programming should become clear: objects are like event-driven data structures. This similarity makes programming for a windowing environment with an object-oriented language very natural.

A Keyboard Interface

Messages can easily be used to create a keyboard interface. Although PC–Project uses the mouse extensively, I wanted to make sure that the user could use the keyboard if he/she preferred to. Windows provides automatic support for keyboard menu commands, but what

about scrolling and selecting activities? Normally these are done by clicking the mouse in a scroll bar or clicking on an activity in the project window.

I was able to trap all key presses and simulate mouse actions in the main window of PC–Project by defining a WM_KEYDOWN method for the ProjWindow class. For example, if the user presses the up arrow, a WM_VSCROLL message is sent. If he/she presses the F2 function key or Enter, a WM_LBUTTONDOWN message is sent.

After years of conditioning with Lotus® 1-2-3®, many PC users intuitively press the slash (/) key to enter commands. Even Microsoft Excel, a model Windows program, employs this method as an alternative to the Windows user interface. Similarly, programs such as Microsoft Word use the Esc key to bring up the command menu. I wanted PC–Project to accommodate all these different user interfaces. But how?

Windows allows us to define accelerator keys that trigger WM_COMMAND messages rather than WM_KEYDOWN messages. The accelerator table is written as part of a resource script file and is separate from the application's source code.

I defined the slash and Esc keys as accelerators that would send a WM_COMMAND Windows message with the wParam argument as the constant PW_COMMAND_MODE. The WM_COMMAND message is trapped so that when wParam has the value PW_COMMAND_MODE, an Actor commandMode message is sent. If the keys are defined as accelerators, rather than trapped as in the WM_KEYDOWN method, they work in all windows of the application and there is no need to remove the keystroke from the input buffer.

PROJECT.RC

```
; Accelerators are used to enhance the keyboard interface
; note: cursor keys are not defined as accelerators and are
; trapped in the WM_KEYDOWN for ProjWindow
;

#define VK_SLASH    191                 ; For Lotus-like commands

PC-Project ACCELERATORS
BEGIN
  VK_SLASH, PW_COMMAND_MODE, VIRTKEY
  VK_ESC, PW_COMMAND_MODE, VIRTKEY
  VK_F1, PW_HELP, VIRTKEY
  "^o", PW_FILE_OPEN
  "^n", PW_FILE_NEW
END
```

How to Trap WM_COMMAND and commandMode Messages

```
/* Handle menu events and accelerator keys identically.
   self refers to the ProjWindow that receives the message.
*/
Def  WM_COMMAND(self, wParam, lParam)
{
  select
    case wParam == PW_FILE_OPEN         /* ^O or menu */
      fileOpenAs(self);
    endCase
    case wParam == PW_FILE_NEW          /* ^N or menu */
      fileNew(self);
    endCase
                .
                .  <other cases omitted... >
                .
    case wParam == PW_HELP              /* F1 or menu */
      help(self);
    endCase
    case wParam == PW_COMMAND_MODE      /* slash or Esc */
      commandMode(self);
    endCase
  endSelect;
}
/* Enter "command mode" in response to a slash key
   accelerator.  This simulates Lotus 1-2-3 style
   commands by sending an Alt key sysCommand message. */

Def  commandMode(self)

{
  WM_SYSCOMMAND(self, 0xF100, 0L);
}
```

▲ **Figure 8** **The accelerators used by PC-Project are shown here. The source code following them depicts how WM_COMMAND and commandMode messages are trapped.**

Figure 9: Initializing a Dictionary

```
/* initialize the actions dictionary */

actions := new(Dictionary, 10);
actions[PW_FILE_OPEN] := #fileOpenAs;
actions[PW_FILE_NEW] := #fileNew;
            :  <other cases omitted>
            :
actions[PW_HELP] := #help;
actions[PW_COMMAND_MODE] := #commandMode;
```

Figure 11: Trapping a Mouse Click

```
/* Respond to a left button mouse click message. The lParam is the
point in the window where clicked. Convert the window location into a
"logical" display point and then send a displayLookup message to the
project. This message will return the activity at the location, which
is considered to be logically true, or logically false if there is no
activity at that location. If there is an activity, edit it; otherwise
just beep. The variable self refers to the ProjWindow that receives
the message. The variables dPoint and activity are local.
*/
Def  WM_LBUTTONDOWN(self, wParam, lParam | dPoint, activity)
{
  dPoint := windowToDisplay(self, lParam);     /* convert */
  activity :=displayLookup(project, dPoint); /* find it */
  if activity              /* logically true if found */
    editInfo(activity);    /* user clicked on an activity */
  else                     /* false if nothing found */
    beep();                /* user clicked on dead space */
  endif;
}
```

The editInfo Method for Class Activity and Descendants

```
/* Display and edit an activity's information. Descendants that use
this method should have a dialogClass() method that returns the type
of dialog to be used.

   self refers to the activity that received the message.
   dlg and retValue are local variables.
   ThePort is the main window used as the dialog' s parent.
*/
Def  editInfo(self | dlg, retValue)
{
  showWaitCurs();                  /* show an hour glass */
  dlg := new(dialogClass(self));   /* we know what kind */
  setEditItem(dlg, self);          /* what to edit? self! */
  retValue := run(dlg, ThePort);   /* run the dialog */
  showOldCurs();                   /* change cursor back */
  ^retValue;                       /* return the run value */
}
```

The dialogClass Method for Class Milestone

```
/* Return the appropriate Actor dialog class for editing.
*/
Def  dialogClass(self)
{
  ^MStoneDialog;
}
```

```
Def command(self, wParam, lParam)
{
  perform(self, actions[wParam]);
}
```

▲ **Figure 10** **In an object-oriented approach to handling menu events, the action is looked up and the operation performed.**

But the question remained, how to respond to the commandMode message in order to activate the menu bar without selecting any item? To answer this question, I used the SPY.EXE utility of the Microsoft Windows Software Development Kit to see what messages Microsoft Excel sends when the slash key is pressed. It sends a WM_SYSCOMMAND message with the constant F100H.

The source code in **Figure 8** shows how the resources are defined as well as how the WM_COMMAND and commandMode messages are trapped. You can use these techniques to add a keyboard user interface to your own Windows programs in Actor or C.

The WM_COMMAND method is written in essentially the same style as would be used in C, by employing a lengthy case statement that determines what action to take. Although this approach works, it is not very object-oriented. A more typical approach in Actor would be to write a method called command that eliminates the case statement by using a lookup table known as a dictionary. Dictionary is a predefined Actor class that allows array-like access to elements of a collection using arbitrary keys.

For example, we would create a dictionary called actions that used as its key the values of the wParam argument and as values the literal message to be sent. The pound symbol (#) is used to specify a literal symbol name. The dictionary would be initialized as shown in **Figure 9**. The command method becomes much shorter; we simply look up the message in the actions dictionary and perform it. Compare this method, as shown in **Figure 10**, to the one shown in **Figure 8**.

Writing the command method in this way makes it much easier for descendant classes to modify their behavior without having to redefine the entire method. They only have to add or change elements in the actions table. This approach leads to much better code reusability than is possible with the procedural approach.

Dialog Boxes

Dialog boxes represent a more advanced challenge. In PC–Project I needed several types of dialog boxes to allow data to be edited. For example, when the user clicks the mouse button in the project window, I want to bring up a dialog box that lets the user edit the selected activity's name, description, starting date, and so on. This is complicated slightly by the fact that different types of activities have different data and thus require different dialog boxes. For example, tasks have time and cost, whereas milestones do not.

Because all the activity classes—that is, Milestones, Tasks, and PERTTasks (tasks with an estimated time)—are logically similar, they descend from a single class called Activity. The Activity class is known as a formal class, since we will never create objects of that class, only objects of the descendant classes.

By making good use of inheritance, we can minimize the amount of code that needs to be written. A general editInfo method can be written for the Activity class that will be inherited by Milestone, Task, and PERTTask. When the user clicks on an activity in the project window, we determine which activity is selected and then send it an editInfo message.

However, the editInfo method must be able to run the appropriate dialog box for each type of activity. How do we know which type of dialog box to run? We simply send a message to the activity. Descendants of the Activity class that use the editInfo method should define a method called dialogClass that returns the type of dialog to use when editing. The dialogClass method is considered part of the public protocol for activities.

The source code to trap the mouse click and edit an activity in Figure 11 shows the WM_LBUTTONDOWN method for the ProjWindow class.

In the same way that the Milestone class descends from the formal class Activity, the MStoneDialog class descends from the formal class ActivDialog, which in turn descends from the Actor class Dialog. In this section I will use higher-level Actor methods that hide some of the details of programming dialog boxes.

Dialogs are created by sending the dialog objects a runModal message. The runModal message requires as

Figure 12: Methods for the ActivDialog Class

```
/* This is a formal class to define behavior common to the various
activity dialog boxes in PC-Project. Descendants should define the
res() method to return the resource ID to be used, initDialog() to
initialize additional fields, and update() to update values in the
activity.

   ActivDialog descends from class Dialog and inherits all
   of its methods and variables.
*/

/* Set the object being edited. */
Def  setEditItem(self, anEditItem)
{
  activity := anEditItem;
}

/* Run the dialog with the appropriate resource. */
Def  run(self, parent | retValue)
{
  ^runModal(self, res(self), parent);
}

/* Initialize all of the fields in the dialog.
   Descendants may wish to initialize additional fields or override
   this method.
*/
Def  initDialog(self, wParam, lp)
{
  setText(self, makeCaption(activity));

  setItemText(self, NAME, getName(activity));
  setItemText(self, DESC, getDesc(activity));
  setItemText(self, UES, getUserEarlyStart(activity));
  setItemText(self, ULF, getUserLateFinish(activity));
  setItemText(self, ES, asString(getEarlyStart(activity)));
  setItemText(self, EF, asString(getEarlyFinish(activity)));
  setItemText(self, LS, asString(getLateStart(activity)));
  setItemText(self, LF, asString(getLateFinish(activity)));
  setItemText(self, SLACK, asString(getSlack(activity)));
}

/* Handle the Ok and Cancel buttons. If Ok was clicked, then update
   the activity. This command method is used by descendants. They
   will define their own update method. */
Def command(self, wParam, lParam)
{
  select
    case wParam == IDOK
       update(self);
       end(self, IDOK);
    endCase
    case wParam == IDCANCEL
       end(self, IDCANCEL);
    endCase
    default
      ^1;          /* ignore it */
  endSelect;
  ^0;
}

MStoneDialog class [in black bar]

/* The MstoneDialog class descends from class ActivDialog
   and inherits all of its methods and variables.
*/

/* Return the resource ID used with this dialog box. */
Def  res(self)
{
  ^MSTONE_BOX;
}


/* Initialize additional fields in the dialog.
   Uses the ancestor's initDialog first.
```

CONTINUED

Figure 12 CONTINUED

```
*/
Def  initDialog(self, wParam, lParam)
{
  initDialog(self:ActivDialog, wParam, lParam);
  setItemText(self, INPUT, getInputNames(activity));
  setItemText(self, OUTPUT, getOutputNames(activity));
}


/* Update the activity after Ok was pressed
   Inform the network if the name changes,
   check the connections and set the values.
*/
Def  update(self)
{
  setName(activity, getItemText(self, NAME));
  addNode(getNetwork(activity), activity);
  checkConnection(activity,
          getItemText(self, INPUT),
          getItemText(self, OUTPUT));
  setValues(self);
}


/* Set the values of the activity.  checkDate displays
   an error message if the date is illegal.
*/
Def  setValues(self | ues, ulf)
{
  ues := checkDate(getItemText(self, UES));
  ulf := checkDate(getItemText(self, ULF));
  setValues(activity, getItemText(self, NAME),
      getItemText(self, DESC), ues, ulf);
}
```

arguments the resource ID (a constant) and a parent window. The layout of the dialog and its fields, known as edit controls, are defined in the resource script file. Before a dialog actually runs, Actor sends an initDialog message. We can trap this message to load the edit controls with initial values. Since edit controls are objects, they manage the details of handling keyboard input, tabbing, and so on.

By making use of inheritance I have the ActivDialog class initialize the edit controls that it knows about and the MStoneDialog class initialize the additional edit controls that it adds. To initialize an edit control with a value, you send a setItemText message to the dialog and specify as arguments a constant indicating the edit control and the value to be used. For example, the message

```
setItemText(self, NAME,
     getName(activity));
```

will load the name of the activity being edited into the edit control named NAME.

I wrote several access methods, such as getName, getEarlyStart, and setValues that provide a safe way to access an activity's private data. Therefore, if I change the representation of a Milestone, I don't need to change code in other classes. This helps maintain a logical division in the program. The dialog box is responsible only for editing and does not need to be concerned with whether changes to the data require a recalculation of the critical path; that is the responsibility of the setValues method of the activity.

The command method defined for the ActivDialog is used to trap user events related to the dialog. In this case, there are only two events that we're interested in—clicking on OK or clicking on Cancel—any other event is ignored. If the user clicks on OK, an update message is sent to the dialog box. Although the command method is defined in the ActivDialog class, it is inherited by the MStoneDialog, TaskDialog, and PERTDialog classes, which define their own update method. The update method sends several other messages, including setValues, which informs the activity that was being edited of its new values so that it can take appropriate action. The code required for the dialog box classes is shown in **Figure 12**.

The other dialog classes, like TaskDialog or PERTDialog, work in a similar fashion, but they require less code because of inheritance. They merely have to initialize any additional edit controls in the initDialog message and define their own setValues method; everything else is inherited.

Worth the Effort

I hope that this "under-the-hood" discussion of PC–Project has helped illustrate the concepts of object-oriented programming and how Windows works. Sometimes it seems like a lot of work to program for Windows, but the end result, a program with a graphical user interface, consistent commands, and device independence makes it worthwhile.

Although it is difficult to make use of polymorphism and inheritance in procedural languages, I encourage you to use object-oriented design techniques no matter what language you are using. By designing objects that encompass both data and their operations and by clearly separating public protocol and private implementation, you will be able to write more understandable and maintainable code. ❑

MDI: An Emerging Standard for Manipulating Document Windows

45

Kevin P. Welch

The Multiple Document Interface (referred to herein as MDI) is a user interface style developed for Microsoft® Windows and OS/2 Presentation Manager (referred to herein as PM) that supports the viewing of multiple child windows within a main application. Each of these smaller child windows can be used to display different sets of data or multiple views of the same set of data.

This article describes MDI, focusing on the user interface (see **Figure 1**) as well as programming aspects of the standard. In the process, it describes a Windows library (MDI.LIB), which will let you easily incorporate the MDI interface into your own applications. The use of this library will then be demonstrated within the context of a simple application (COLORS.EXE). Finally, the MDI standard will be contrasted and compared to the IBM® Systems Application Architecture (SAA) Common User Access (CUA) guidelines for user interfaces.

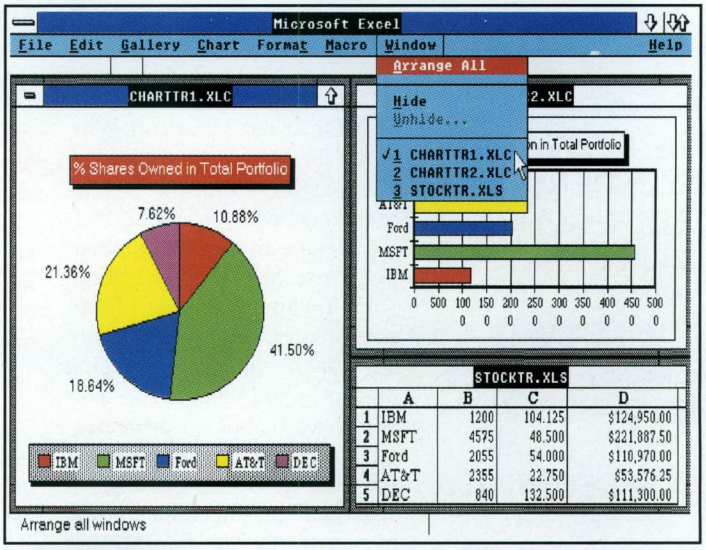
Background & Motivation

Windows and PM developers have long been fascinated with applications that contain windows within windows. This interest stems from both the natural capabilities of the host environments and the influence of other windowing systems. MDI can therefore be considered an outgrowth of programmers' interest and experience as they have attempted to create a shared-menu environment inside Windows. This work, after considerable refinement in the Windows environment, is now being applied to OS/2 PM.

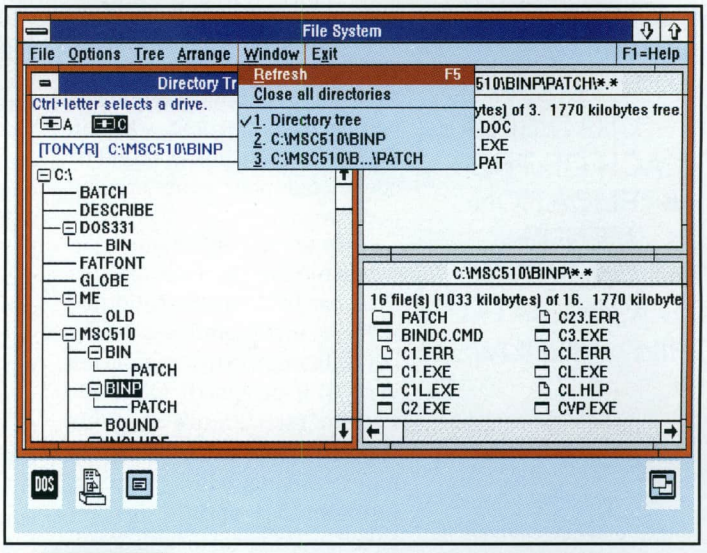
To a degree, the development of MDI satisfied some creative instincts expressed by developers while also providing multi-window functionality for IBM-compatible personal computers. In addition, its specification is facilitating the development of an entirely new class of interoperable applications that create similar user interfaces on the Macintosh, Windows, and OS/2 PM.

From this beginning, the MDI specification was refined and extended (primarily by Microsoft) in a determined effort to make it reasonably CUA conformant. The result of this

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

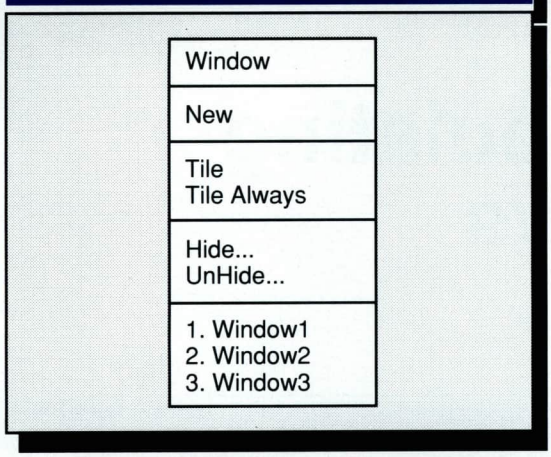


▲ **Figure 1** **Microsoft Excel utilizes the Multiple Document Interface.**



▲ **Figure 2** **The OS/2 Presentation Manager File System, based on the MDI protocol.**

Figure 3: Standard MDI Menu Template



THE FIRST THING TO UNDERSTAND ABOUT MDI IS ITS MAIN DESKTOP WINDOW. THIS WINDOW IS ALMOST ALWAYS RESIZABLE, WITH A TITLE BAR EQUIPPED WITH A STANDARD SYSTEM MENU AND MINIMIZE/MAXIMIZE ICONS. THE MAIN WINDOW IS ALSO USED TO DISPLAY EACH OF THE APPLICATION MENUS BELONGING TO ITS ASSOCIATED CHILD WINDOWS.

effort was the formal definition of MDI in the Microsoft Windows Software Development Kit (SDK) followed by its implementation in Microsoft Excel—the first major application to use the new specification.

Following the lead of Microsoft Excel, many software developers have tried using multiple child windows in their applications, but unfortunately only a few have succeeded in fully implementing the original specification. This failure is due in part to the fact that MDI is reasonably difficult to implement correctly since it requires a good low-level understanding of the underlying environment. Furthermore, implementing MDI in Windows involves a number of subtle tricks, which in the best circumstances might be considered poor programming practice.

Despite its difficulties, in recent months the acceptance of MDI has been further solidified with its incorporation into the new (although in my opinion not very well designed) OS/2 file system (see **Figure 2**). This coupled with the also new but more consistent PM programming model should further enhance its appeal to both developers and publishers alike.

Definition & Specification

From the start, it must be clearly understood that MDI is a user interface style; that is, it is not a set of absolute rules but a collection of a few underdefined guidelines. Although many software developers have implemented MDI within the general style guidelines, few have implemented it in exactly the same way.

The first thing to understand about MDI is its main, or top-level, desktop window. This window is almost always resizable, with a title bar equipped with a standard system menu and minimize/maximize icons. In most circumstances, the title bar, or caption, of the main window contains only the name of the application (more on this later). As with most windows, the standard system menu is provided with the following options and accelerators:

Restore	**Alt-F5**
Move	**Alt-F7**
Size	**Alt-F8**
Minimize	**Alt-F9**
Maximize	**Alt-F10**
Close	**Alt-F4**

The main window is also used to display each of the application menus belonging to its associated child windows. The application menus vary according to the type of document the active child window contains. For example, when the user moves from a child window that contains a chart document to one that contains a spreadsheet document, the main application menu changes to reflect the capabilities of the active child window. Some menu items can remain active regardless of which child window is selected (for example, various file or formatting commands that have equivalent meaning in different contexts).

In addition, the main menu provides a Window management pull-down menu. Commands on the pull-down menu are applicable regardless of the child window selected, allowing the user to manage the main window and all its children. The first part of the pull-down menu has commands that allow the user to manage the size, position, and visibility of each of the child windows. The next part has a list of all the currently visible (including iconic) child windows. In most cases, the Window pull-down menu looks something like **Figure 3**.

The New command lets the user create a new view into the currently selected document. That is, the user creates a new child window that contains the currently active document. Although this might not always be appropriate, it is useful in situations in which the user wishes to view a different portion of the same document. It could be used, for example, to support multiple enlargements of a drawing in a paint program: one window could contain the entire image and another a detailed view.

Since screen "real estate" is quite limited, most MDI implementations incorporate some mechanism to arrange the various child windows. Here, following the New command, are two commands that help manage the visual arrangement of the child windows.

The Tile command "tiles" each of the active child windows inside the parent client area. Although many effective tiling algorithms can be devised, most assign some sort of priority to the currently active child window and place it in the largest space available. Note that in most implementations the Tile command is a one-time event—any subsequent movement of the child windows will destroy the tiling.

The Tile Always command is an extension of the Tile command as it forces all of the child windows to remain continu-

ously tiled. When the size of one child window is adjusted, the relative sizes of the other children are changed to compensate. Any change to the parent window or to one of the children automatically causes a tiling to occur around it (much as it did with Windows Version 1.03). Although this technique has not been used in any major Windows or PM applications, it has many merits that warrant serious consideration.

Frequently users end up with many documents open simultaneously, resulting in a cluttered desktop. Following the tiling commands are therefore two commands that enable the user to hide or show one or more child windows. The Hide command lets the user hide the currently active child window (see **Figure 4**). One popular variation on this theme is to let the user simultaneously hide one or more child windows using a dialog box that contains a multiple selection list box. This makes it possible for the user to clear a portion of the desktop in one fluid motion.

When windows are hidden, they remain active but cannot be accessed. By using the Unhide command, the user can select one or more windows (from a list of all hidden windows) and make them visible again (see **Figure 5**). The windows can be restored to their original size and location or, if tiling is active, merged into the desktop.

The last group of items on the Window pull-down menu is a list of all currently active child windows. The windows are listed by title, with each title preceded by a digit that serves as a short mnemonic. This facilitates quick and consistent keyboard access to each child window regardless of the current title. If a currently active child window exists, it is indicated by a check mark beside its title.

In some applications, certain commands may only be applicable to the main window. When this is the case, the main window may be listed at the beginning of the window list. This allows the user to access the commands that are supported by the main window quite easily. Applications that do not need this feature can omit the main window from the window list.

Child Windows

The next thing to understand about MDI—after its main desktop window—is its associated child windows. Like the desktop, each child window is resizable and contains a title bar. The title bar normally has the name of the document being edited. If a single document is being viewed by more than one child window, a number is appended after the document name; for example,

```
CHART.XLC:1
CHART.XLC:2
CHART.XLC:3
```

Only one child window can be active at a time, and it is distinguished from the others by a change in the color or pattern of the title bar (usually with the same mechanisms used to differentiate the main desktop from other top-level windows). Note that the main desktop window remains active when one of the child windows is enabled. To a certain extent, this appears to be a visual contradiction since the input focus seems to be simultaneously shared between two windows, to say nothing of the programmatic hoops you have to jump through to accomplish this sleight of hand.

The active MDI child window also contains a control or system menu box. Although similar to the one maintained by the parent window, it is activated by the Alt-Minus key combination (I'm not completely sure of the rationale behind this). The commands on the child system menu are identical to those on the main window, except that the Alt key is replaced by the Ctrl key. The end result is a system menu that contains the following options and accelerators:

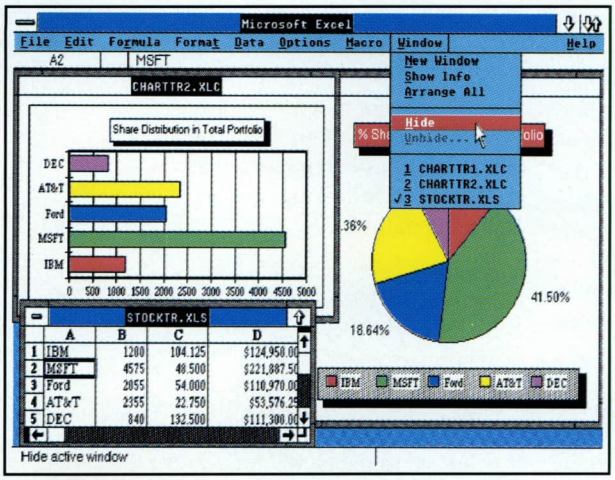
Restore	**Ctrl-F5**
Move	**Ctrl-F7**
Size	**Ctrl-F8**
Minimize	**Ctrl-F9**
Maximize	**Ctrl-F10**
Close	**Ctrl-F4**

It is left up to the application to disable or gray any of these commands that are inappropriate. In most implementations, only the Minimize command is disabled.

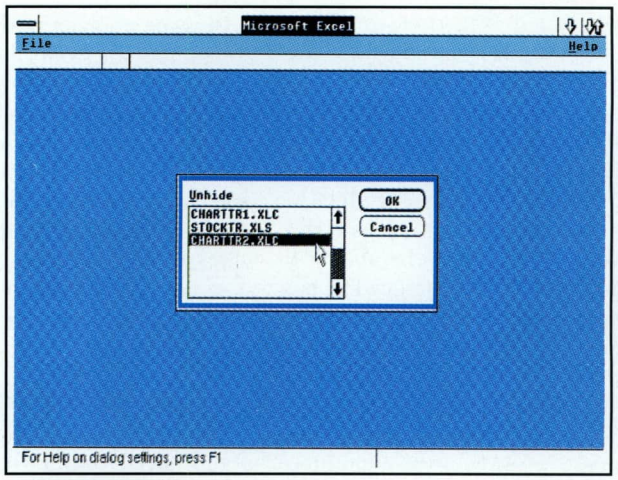
The Move and Size commands (accessed by Ctrl-F7 and Ctrl-F8, respectively) allow the location and size of the child window to be controlled. These functions mirror the ones available on the desktop but are restricted to keep the child window inside the parent. Like most operations, the movement and resizing of the child windows can also be accomplished using the mouse.

An interesting item to note is how the child window frame is handled when moved. Although the mouse is clipped to the client area of the desktop, in Windows the frame can extend outside the parent window boundary. In PM implementations of MDI, the frame is clipped by the system to the client area of the desktop.

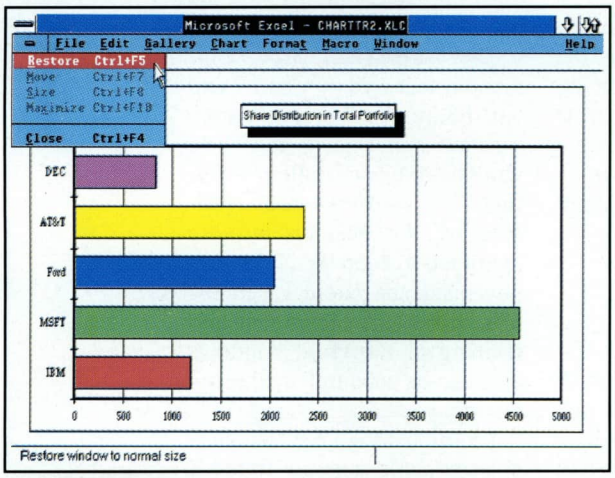
The Minimize command (Ctrl-F9), seldom implemented in Windows, reduces the child window to an icon inside the MDI desktop. The resulting icon can then be selected, moved around the desktop to a new location, and restored to its original size and location. As is the case with all visible child windows in the MDI desktop, the icon can be hidden or selected using the Window pull-down menu. Note that throughout this



▲ **Figure 4** **The standard MDI Hide command, part of the standard "Window" menu.**



▲ **Figure 5** **The Unhide dialog box.**



▲ **Figure 6** **Maximized child window. Note that the child window name has been added to the Microsoft Excel title bar.**

process the icon must remain inside the client area of the desktop window and cannot be moved elsewhere on the display. Although this is relatively easy to accomplish in OS/2 PM, it adds a whole new level of complexity to a Windows implementation of MDI (and so is seldom implemented). In PM, however, it is considerably easier to implement this feature, and I expect that more applications will take advantage of it when implementing MDI. Refer to the WITHIN sample application in the MS® OS/2 Software Development Kit Version 1.1 (OS/2 SDK) if you're curious.

The Maximize command (Ctrl-F10) causes the child window to be enlarged to fill the entire client area of the desktop window (see **Figure 6**). As a shortcut, you can use the mouse and click inside the maximize icon or double-click anywhere in the title bar.

Since the client area of the child window fills the main window, you can consider that the title bar of the child "slides under" the menu bar of the desktop window. When this happens, two other changes occur. The first regards the main window caption. Originally, it contains only the application name. But when a child window is maximized, the title bar of the desktop is changed to include the name of the currently active document, much as is done in normal, non-MDI applications. The second, and perhaps even more complicated change, involves the movement of the child window's system menu to the beginning of the application menu. This allows continued access to the child system menu, letting you close or restore the window to its original size.

If you think things are complicated enough, consider what happens when a new MDI child is created or an existing one is selected while another is maximized. The MDI specification dictates that when a different child window is selected or a new one created, it automatically assumes the characteristics of the previously selected window. This implies that if you create a new child window while in a maximized state, the new window will also be displayed in a maximized state. Similarly, when you close a maximized child window, the MDI desktop automatically selects the next available child window and maximizes it for you—whether you wanted it to or not.

The Restore command (Ctrl-F5), as you might expect, causes the maximized child window (or minimized if implemented) to be restored to its original size and location among the other windows, much as it does with top-level system menus.

Finally, the Close command (Ctrl-F4) destroys the currently selected child window. In situations in which the child window is one of several views into a common document, the title bars of the remaining windows are automatically renumbered to

reflect the change. If the child window being closed is the last one accessing a document, a dialog box is normally displayed to confirm any required save operations. As is the case with all system menus, double-clicking the mouse inside the system menu box is a shortcut for choosing the Close command.

Another item to note about all these commands is that they apply only to the child window that is currently active. This means only that the window has a system menu and minimize/maximize menu boxes. Unfortunately, the original MDI specification does not make this terribly clear. The end result is that each of the child windows is responsible for changing its visible attributes when it receives and loses the input focus.

If that isn't enough, the MDI specification also calls for a number of keyboard accelerators to move between the various child windows (see **Figure 7**). Despite the fact that the keyboard accelerators are not listed on the child system menu, they represent the only mechanism for moving between child windows without a mouse. It is left up to users to remember (assuming they read their manuals) what they are and how they work. Furthermore, the task of implementing these accelerators is yet another activity that must be managed by the already overburdened MDI desktop window.

Figure 7: MDI Keyboard Accelerators

F6	select next active document subdivision, clockwise
Shift-F6	select next active document subdivision, counterclockwise
Ctrl-F6	select next active document, from front to back
Shift-Ctrl-F6	select next active document, from back to front

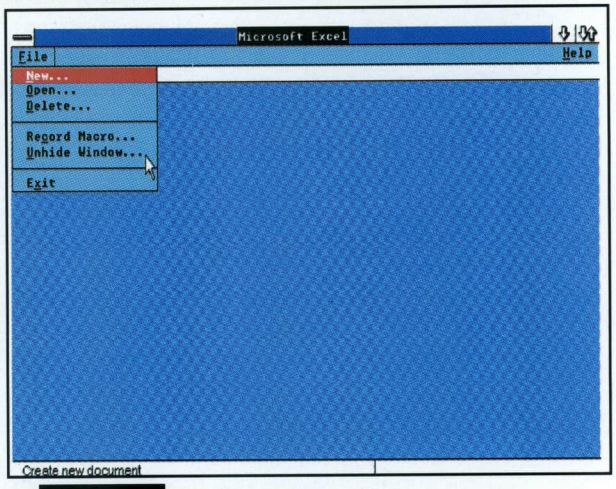
Design Issues

Before I get into the actual programming issues involved with implementing MDI, it seems appropriate to discuss the design issues that are bound to come up when working with the specification. Perhaps foremost is the additional complexity associated with using MDI. If you don't have the idea by now, it takes a great deal of time to implement MDI and get it to work correctly. From a design standpoint, MDI requires that each child window be object-oriented in nature (maintaining its own instance data) yet be able to access shared data that is held in common when multiple views are in effect. In addition, the standard has some serious performance implications since it introduces more support code and yet another level of hierarchy into the system.

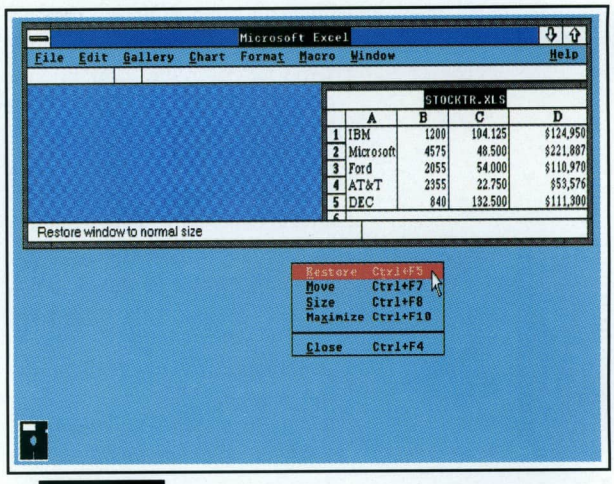
It is also natural to compare multiple instances of tightly coupled applications to the MDI alternative. On the positive side, a group of independent applications are often easier to design, implement, and test, especially when the environment takes care of the data-handling issues for you. This approach works especially well when using non-Windows-aware applications or those whose hold on the environment is tenuous in the best of circumstances.

On the negative side are the resulting cluttered display, lack of interoperable consistency with the Macintosh, and difficulties of well-integrated interprocess communication between separate applications. Furthermore, since many applications require the use of multiple views into the same document, MDI is something you will probably have to think seriously about.

Another troublesome design area with MDI is the way that it treats menus. Although menus are relatively simple to structure when each child window is homogeneous in nature and shares the same or similar capabilities, the design can be difficult when child windows are



▲ **Figure 8** **Microsoft Excel with all child windows hidden. Note that most menu items have disappeared and that Unhide Window is now part of the system menu.**



▲ **Figure 9** **When a child window is not in the visible part of the client area, the keyboard can still interact with that child window. However, some strange effects may occur, such as the child window's system menu appearing to float free from the actual application.**

Figure 10: MDI Application Programming Interface

MdiMainCreateWindow()	Main window creation function
MdiMainDefWindowProc()	Main MDI window default window function
MdiChildCreateWindow()	MDI child window creation function
MdiChildDefWindowProc()	MDI child window default window function
MdiGetMessage()	MDI application message retrieval function
MdiTranslateAccelerators()	MDI application translate accelerator function
MdiGetMenu()	MDI menu retrieval function
MdiSetAccel()	MDI set document window accelerator table function

very heterogeneous or involve compound documents.

Menu design is further complicated when the desktop is empty or all the child windows are hidden. Most of the normal menu options will be disabled and of no interest to the user. Many applications (for example Microsoft Excel, as shown in **Figure 8**) respond to this situation by severely pruning their menus, adding additional menu-handling complexity.

One of the most serious drawbacks of the MDI standard is the problem that results when the desktop window is resized. Although you can't move a child window completely outside the client area, it is possible to have it end up there if you change the size of the MDI parent. The end result is a window (perhaps even an active one) that is completely invisible. Despite the fact that you can't point to the invisible window with a mouse, you can use the child window accelerators to get to it—but you still can't see it.

An interesting visual phenomenon can be created if you use the Ctrl-Minus key combination while the active child window is outside the client area. As you would expect, the child's system menu appears but the child window remains hidden. The result is that the child's system menu appears unconnected to the desktop, magically making itself visible with no apparent connection to anything else (see **Figure 9**). Interesting, maybe, but certainly somewhat confusing for users.

MDI API

By now you may be wondering whether MDI is something you want to take on—especially since it doesn't really do anything except manage a collection of related child windows. But there is a good reason to use it—the definition and implementation of an Application Program Interface (API) that manages the MDI. The end result of the API is a small library of object modules (approximately 16Kb in total size) that performs all the work of integrating MDI into your application for you. And best of all, it's no extra charge with the price of your *MSJ* subscription.

In order to accomplish the task of integrating the API into an application I enlisted the help of friend, long-time associate, and Windows guru—Geoffrey Nicholls. Together we came up with an API that lets you write complex MDI applications as if they were standalone, independent applications.

We recognized from the start that developing such an API would be quite dirty (doing things we would never do in conventional Windows programming) and that we would really have to try to keep it small and simple. We also realized that we would not be able to implement the entire specification, only some of the more important facets—the rest we would leave to you. Finally, we wanted to make it as object-oriented as possible. After several false starts and rewrites we ended up accomplishing our goals but in so doing perhaps used property lists and window offsets to excess.

The MDI API we came up with in a sense represents an analog of the existing Windows API. From previous experience, we knew that to a large degree the operating characteristics of a window are defined by the default window message processing function. The MDI API attempts to change this foundation and give each window a new and different set of characteristics. The net result is a small number of routines with familiar parameter lists that can be used together to give your application MDI characteristics. They are shown in **Figure 10**.

Message Flow and Process Sequencing

In the next two sections we will examine the inner workings of the MDI API. We first describe the general message flow and processing sequence used by the API. Then we describe each of the top-level functions and explain some of the subtle ways in which they work. As you read these sections, refer to the MDI source code listings accompanying this article. The code is reasonably well documented, so you should be able to understand it if you have a good background in Windows programming.

Now, perhaps the most efficient way to learn about the MDI API is to study the MDI message flow diagram carefully (see **Figure 11**). It tracks the path of each message received by an application that uses the API, focusing on those that are of particular interest.

The first thing to notice is the rather normal message retrieval, translation, and dispatch loop at the top of the diagram. This occurs much as it would in any other Windows application, the only difference being in a spe-

Figure 11: MDI MESSAGE FLOW

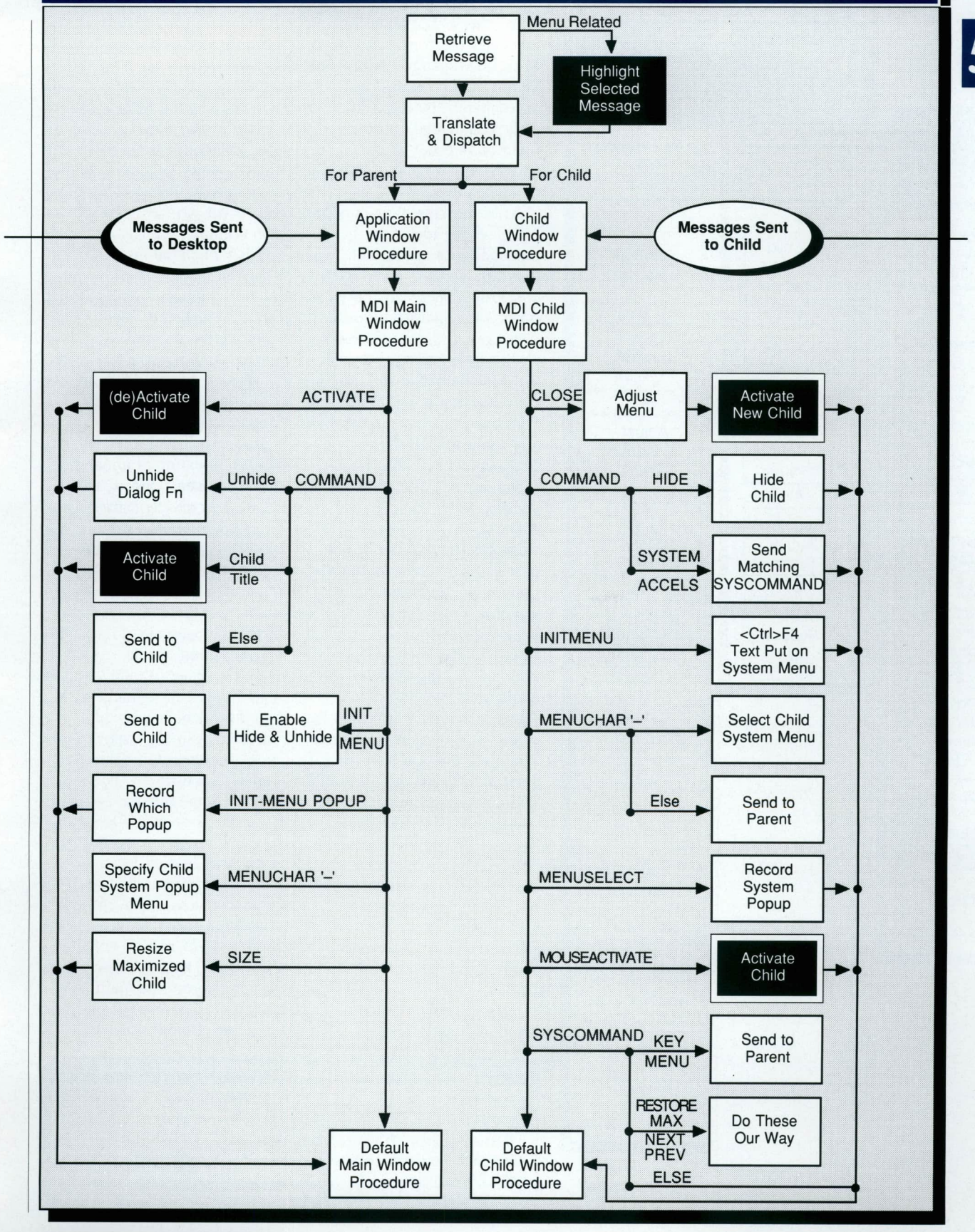


Figure 12: ACTIVATE MDI CHILD PROCESSING SEQUENCE

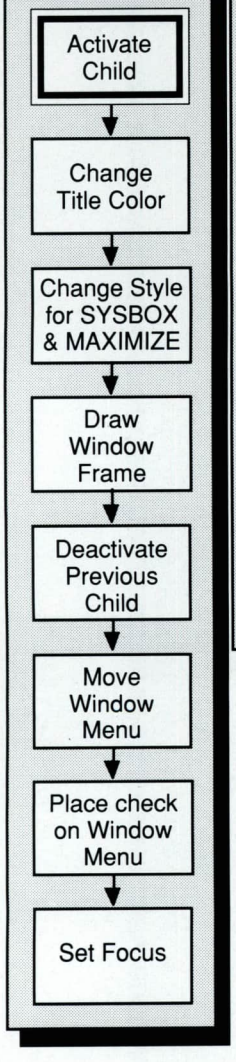


Figure 13: ZOOM MDI CHILD PROCESSING SEQUENCE

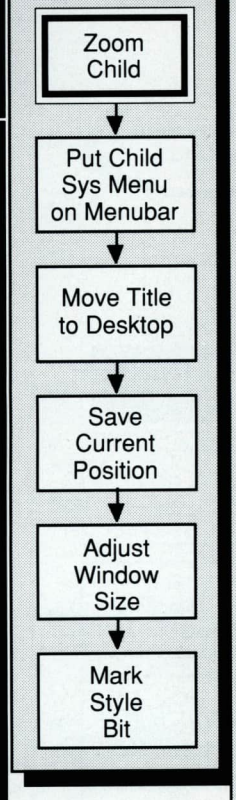
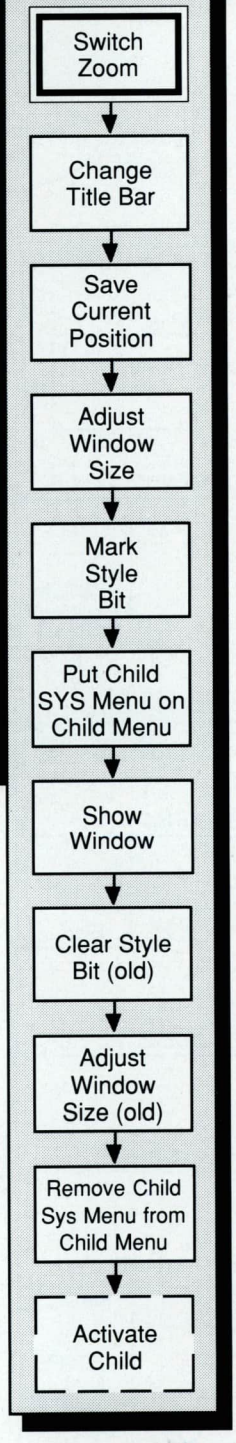


Figure 14: SWITCH ZOOMED CHILD PROCESSING SEQUENCE



cific check for menu-related messages (performed inside the MdiGetMessage function). When such messages are encountered, they are immediately dispatched to a window function in order to activate the various system and application menus correctly.

After the message-handling loop, each message is dispatched to an appropriate window function. As far as MDI is concerned, there are only two types of windows present—a desktop or parent window and child or document windows. On the left side of the diagram, the flow of events for the desktop window is listed; on the right side, a similar flow for each of the document windows is listed.

Tracing down the left, or desktop side, each message is processed by the main application window function, then passed on to the default MDI main window function. The remainder of events listed below occur inside this default function, finally ending in most messages being sent on to the standard DefWindowProc.

As you can see from the diagram, the default MDI main window function is primarily interested in activation, initialization, and command-related messages. All other messages are sent on without modification to the DefWindowProc. Of those intercepted, some result in a particular action being performed (like the activation of a particular child window); others are processed and sent directly to an appropriate child window—bypassing the default window function.

On the right, or document side of the diagram is the sequence of events that occur when messages are received by the default MDI child window function. As is the case with the left side, the flow of events listed occur inside this default function, ending with most of the messages being sent on to the standard DefWindowProc.

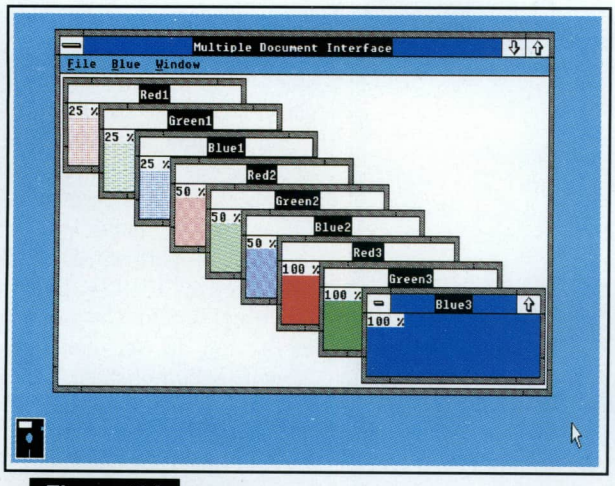
Like the default desktop window function, the child window function is primarily interested in activation, initialization, and command-related messages. Of particular importance are the various system commands. Some are handled directly and not passed to the system. In certain cases, such as those that involve activation of a menu or a new document window, the message is sent directly to the desktop.

In key places in the diagram you can see highlighted rectangles, which represent the activation of a child or document window. **Figure 12** separates this activation step into a number of smaller components.

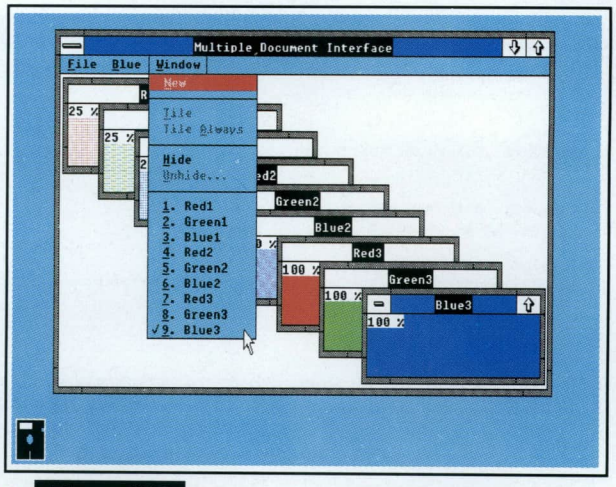
In a normal Windows application, the activation of a child window occurs with little fanfare, but in an MDI implementation a number of important steps must be performed. The first is changing the title bar color. Although Windows allows only one window to have the input focus, when a child window is selected it seems that both the desktop and the child are simultaneously active. This is done by manually sending a WM_NCPAINT message (with appropriate parameters) to the DefWindowProc of the window being activated.

Figure 15: Files for Creating the MDI API Library

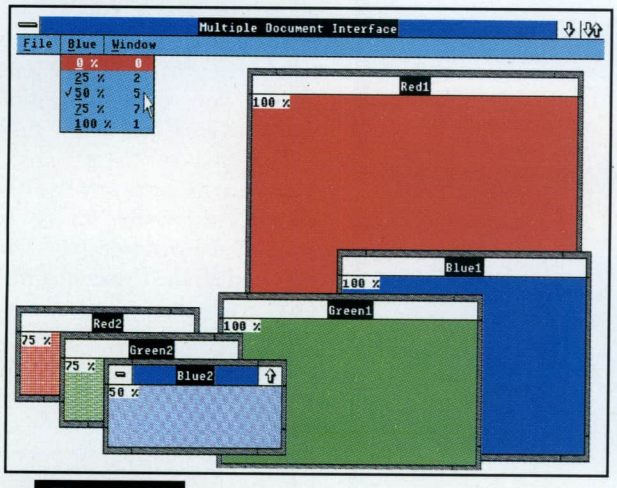
MDI	Library make file
MDI.H	Library header file
MDI1.C	Main MDI API functions
MDI2.C	Activation and switching of document windows
MDI3.C	Handling of menus and keyboard user interface



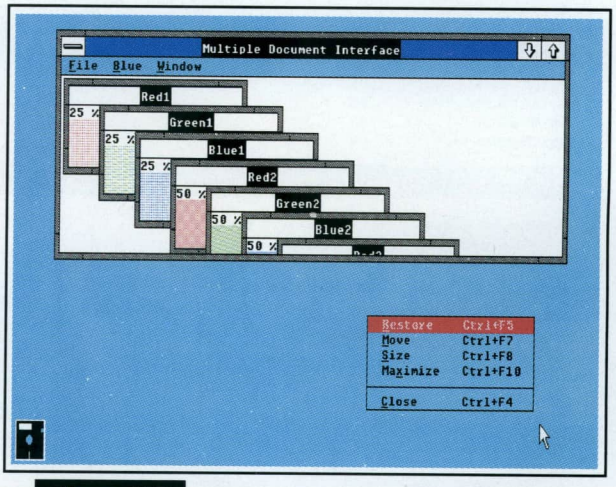
▲ **Figure 16A** The Colors program can create red, green, or blue child windows in varying densities of color. Each new child window is positioned by the application in a cascading fashion.



▲ **Figure 16B** Colors has the standard Window Menu which lists each child window.



▲ **Figure 16C** The child window process for blue allows the user to change color densities from the keyboard by selecting, respectively, the numbers shown in the menu above (2, 5, 7, or 1). The child windows can, of course, be resized or hidden.



▲ **Figure 16D** When a child window in Colors is off the client area, it is still accesible by way of the keyboard, complete with floating means, as was seen in figure 9.

Furthermore, under the interface specification, only the currently active document window contains an MDI system menu and maximize/minimize icons. Because of this, the second step is to change the style of the window to include these new attributes and then force the system to display the changes.

The next major task to perform when a document window is activated involves replacing the desktop menu. Each child window is associated with its own menu. When it is activated, the current desktop menu is replaced and the new one inserted, retaining all the attributes that it previously had. Finally, just before the input focus is transferred to the document, the Window pull-down menu has to be updated to reflect the current status of the desktop.

Like **Figure 12**, **Figures 13** and **14** list the sequence of events that transpires either when a document window is maximized or a different document window is selected while in maximized state. Of these two sequences, the only somewhat technical task is the automatic insertion of the MDI system menu at the beginning of the application menu. This involves retrieving the MDI menu icon from Windows with the following call:

```
hBitmap = LoadBitmap(
        NULL,
        MAKEINTRESOURCE(
        OBM_CLOSE) );
```

The resulting bitmap contains both the standard system menu

Figure 17: Source Code for COLORS.EXE

COLORS - MAKE File

```
# compilation flags
CFLAGS=-AM -c -Gsw -Osal -W2 -Zp

# COLORS
colors.res: colors.rc colors.ico colors.h mdi.h
   rc -r colors.rc

colors.obj: colors.c colors.h mdi.h
   cl $(CFLAGS) colors.c

colors.exe: colors.obj colors.def mdi.lib
   link4 colors,colors/ALIGN:16,colors,mdi+mlibw+mlibcew/NOE,colors
   rc colors.res

colors.exe: colors.res
   rc colors.res
```

COLORS.DEF - DEF File

```
NAME                    COLORS
DESCRIPTION             'Multiple Document Interface'
STUB                    'WINSTUB.EXE'

CODE                    MOVEABLE DISCARDABLE PURE LOADONCALL SHARED
DATA                    MOVEABLE MULTIPLE

HEAPSIZE                2048
STACKSIZE               2048

EXPORTS
   MainWndProc          @1
   ColorWndProc         @2
   MainDlgNew           @3
   MainDlgAbout         @4
   MdiMsgHook           @5
   MdiDlgUnhide         @6
```

COLORS.RC - Resource File

```
/* COLORS.RC - Resources for COLORS program */

/* COLORS section of file */

#include <windows.h>
#include "colors.h"
#include "mdi.h"

MainIcon ICON  colors.ico

MainMenu MENU
BEGIN
   POPUP "&File"
   BEGIN
      MENUITEM "&New...",          IDM_NEW
      MENUITEM "&Open...",         IDM_OPEN, GRAYED
      MENUITEM "&Save",            IDM_SAVE, GRAYED
      MENUITEM "Save &As...",      IDM_SAVEAS, GRAYED
      MENUITEM "&Close",           IDM_CLOSE, GRAYED
      MENUITEM SEPARATOR
      MENUITEM "&Exit",            IDM_EXIT
      MENUITEM "A&bout...",        IDM_ABOUT
   END
END

RedMenu MENU
BEGIN
   POPUP "&File"
   BEGIN
      MENUITEM "&New...",          IDM_NEW
      MENUITEM "&Open...",         IDM_OPEN, GRAYED
      MENUITEM "&Save",            IDM_SAVE, GRAYED
      MENUITEM "Save &As...",      IDM_SAVEAS, GRAYED
      MENUITEM "&Close",           IDM_CLOSE
      MENUITEM SEPARATOR
      MENUITEM "&Exit",            IDM_EXIT
      MENUITEM "A&bout...",        IDM_ABOUT
   END
   POPUP "&Red"
   BEGIN
      MENUITEM "&0 %",             IDM_0
      MENUITEM "&25 %",            IDM_25
      MENUITEM "&50 %",            IDM_50
      MENUITEM "&75 %",            IDM_75
      MENUITEM "&100 %",           IDM_100, CHECKED
```

CONTINUED

icon and the MDI one. After retrieving the bitmap dimensions (via a GetObject call), you can extract the MDI system menu bitmap for use when updating the application menu. This entire sequence of events (never before publicly documented) is performed by the MdiCreateChildSysBitmap function, should you have the inclination to see how it is actually accomplished.

Top-Level Functions

Now that I've discussed the message flow and process sequencing of MDI, I will focus on the top-level function calls provided by the API. Programmatically speaking, if you understand these functions, you will be able to use the MDI interface in your own applications without a great deal of difficulty.

The first of these calls is MdiMainCreateWindow, which is responsible for the creation of the main desktop window and all the associated MDI property lists required to make the interface work. The actual data structures used by the API are maintained with property lists attached to the desktop and document windows. A property list is an attempt to give each window access to some sort of instance data (to borrow an object-oriented programming term). This is accomplished by associating a window handle with a named block of memory. Using the interface provided by Windows, any window can set, enumerate, retrieve, and destroy properties. Although there is no predefined limit to the number of properties that a window can have, the property list itself, like other window-related data, is actually allocated in the local heap of the user library. The MdiMainCreateWindow function also attaches the Window pull-down menu to the main application menu, making the

MDI interface almost transparent to the desktop.

The second API call is MdiMainDefWindowProc. As described in the MDI message flow diagram, this function is responsible for the default processing of all desktop-related messages. More specifically, it is interested in messages that involve the activation and deactivation of the desktop, menu-related messages, and messages relating to the visibility and sizing of the associated document windows. Implemented as a large switch statement, it passes most of the messages on to the DefWindowProc.

The next API call is the MdiChildCreateWindow function. This function, identical in many ways to the standard Windows CreateWindow call, creates a new child window inside the desktop. Behind the scenes it sets up the related property lists, keeps track of the window menu and accelerator table, and activates the child window correctly, depending on the current state of the desktop.

As with the desktop window, each document window is associated with a default MDI message-processing function, MdiChildDefWindowProc. This function is responsible for the default handling of all document window related messages, especially those that involve system menu choices and window creation, activation, and destruction. Those messages not handled are either sent directly to the desktop window or are passed on to the DefWindowProc.

After the default child window function is a replacement for the standard Windows message function, MdiGetMessage. This function is responsible for retrieving all of the application messages from the system queue. It also checks for keyboard menu access and activates

Figure 17 CONTINUED

```
   END
END

GreenMenu MENU
BEGIN
   POPUP "&File"
   BEGIN
      MENUITEM "&New...",          IDM_NEW
      MENUITEM "&Open...",         IDM_OPEN, GRAYED
      MENUITEM "&Save",            IDM_SAVE, GRAYED
      MENUITEM "Save &As...",      IDM_SAVEAS, GRAYED
      MENUITEM "&Close",           IDM_CLOSE
      MENUITEM SEPARATOR
      MENUITEM "&Exit",            IDM_EXIT
      MENUITEM "A&bout...",        IDM_ABOUT
   END
   POPUP "&Green"
   BEGIN
      MENUITEM "&0 %",             IDM_0
      MENUITEM "&25 %",            IDM_25
      MENUITEM "&50 %",            IDM_50
      MENUITEM "&75 %",            IDM_75
      MENUITEM "&100 %",           IDM_100, CHECKED
   END
END

BlueMenu MENU
BEGIN
   POPUP "&File"
   BEGIN
      MENUITEM "&New...",          IDM_NEW
      MENUITEM "&Open...",         IDM_OPEN, GRAYED
      MENUITEM "&Save",            IDM_SAVE, GRAYED
      MENUITEM "Save &As...",      IDM_SAVEAS, GRAYED
      MENUITEM "&Close",           IDM_CLOSE
      MENUITEM SEPARATOR
      MENUITEM "&Exit",            IDM_EXIT
      MENUITEM "A&bout...",        IDM_ABOUT
   END
   POPUP "&Blue"
   BEGIN
      MENUITEM "&0 %\t  0",        IDM_0
      MENUITEM "&25 %\t  2",       IDM_25
      MENUITEM "&50 %\t  5",       IDM_50
      MENUITEM "&75 %\t  7",       IDM_75
      MENUITEM "&100 %\t  1",      IDM_100, CHECKED
   END
END

BlueAccel   ACCELERATORS
BEGIN
   "0",         IDM_0
   "2",         IDM_25
   "5",         IDM_50
   "7",         IDM_75
   "1",         IDM_100
END

STRINGTABLE
BEGIN
   IDS_TITLE,              "Multiple Document Interface"
   IDS_MAINCLASS,          "MdiMainClass"
   IDS_COLORCLASS,         "MdiChildClass"
END

MainNew  DIALOG   50, 50, 144, 60
STYLE WS_DLGFRAME | WS_POPUP
BEGIN
   GROUPBOX "New"                              -1,   4,  4, 100, 52
   RADIOBUTTON "&Red",                  DLGNEW_RED,   8, 16,  88, 10
   RADIOBUTTON "&Green",              DLGNEW_GREEN,   8, 28,  88, 10
   RADIOBUTTON "&Blue",                DLGNEW_BLUE,   8, 40,  88, 10
   DEFPUSHBUTTON "&OK"                       IDOK, 108,  8,  32, 14
   PUSHBUTTON "&Cancel"                  IDCANCEL, 108, 28,  32, 14
END

MainAbout   DIALOG   22, 17, 156, 100
STYLE WS_DLGFRAME | WS_POPUP
BEGIN
    CTEXT "Multiple Document Interface",        -1,  0,  8,152,  8
    CTEXT "By Geoffrey D. Nicholls",            -1,  0, 20,152,  8
    CTEXT "and Kevin P. Welch",                 -1,  0, 28,152,  8
    CTEXT "(C) Copyright 1988",                 -1,  0, 40,152,  8
    CTEXT "Eikon Systems, Inc.",                -1,  0, 48,152,  8
    CTEXT "Version 1.00  1-Nov-88",             -1,  0, 60,152,  8
    DEFPUSHBUTTON "Ok"                  IDOK, 60, 80, 32, 14, WS_GROUP
END
```

CONTINUED

Figure 17 CONTINUED

```
/* MDI section of file */

MdiMenu MENU
BEGIN
   MENUITEM "&New",                 IDM_NEWWINDOW, GRAYED
   MENUITEM SEPARATOR
   MENUITEM "&Tile",                IDM_ARRANGE, GRAYED
   MENUITEM "Tile &Always",         IDM_ARRANGEALL, GRAYED
   MENUITEM SEPARATOR
   MENUITEM "&Hide",                IDM_HIDE, GRAYED
   MENUITEM "&Unhide...",           IDM_UNHIDE
END

MdiChildAccel  ACCELERATORS
BEGIN
   VK_F4,      IDM_CLOSE,       VIRTKEY, NOINVERT, CONTROL
   VK_F5,      IDM_RESTORE,     VIRTKEY, NOINVERT, CONTROL
   VK_F6,      IDM_NEXTWINDOW,  VIRTKEY, NOINVERT, CONTROL
   VK_F6,      IDM_PREVWINDOW,  VIRTKEY, NOINVERT, CONTROL, SHIFT
   VK_F7,      IDM_MOVE,        VIRTKEY, NOINVERT, CONTROL
   VK_F8,      IDM_SIZE,        VIRTKEY, NOINVERT, CONTROL
   VK_F10,     IDM_MAXIMIZE,    VIRTKEY, NOINVERT, CONTROL
END

MdiUnhide   DIALOG 50, 50, 132, 68
STYLE WS_DLGFRAME | WS_POPUP
BEGIN
   LTEXT "&Unhide",                          -1,   4,  4,  88, 10
   LISTBOX                DLGUNHIDE_LB, 4, 16, 88, 48, WS_TABSTOP
   DEFPUSHBUTTON "&OK"                     IDOK,  96,  8,  32, 14
   PUSHBUTTON "&Cancel"                 IDCANCEL,  96, 28,  32, 14
END
```

COLORS.H - Header File

```
/ * COLORS.H - Include for COLORS program */

/* Resource file constants */

/* Strings */
#define  IDS_TITLE                1
#define     IDS_MAINCLASS         2
#define     IDS_COLORCLASS        3

/* Debugging menu choice */
#define     IDM_DEBUG             0x100

/* File Menu Choices */
#define     IDM_NEW               0x101
#define     IDM_OPEN              0x102
#define     IDM_SAVE              0x103
#define     IDM_SAVEAS            0x104
#define     IDM_PRINT             0x105
#define     IDM_ABOUT             0x106
#define     IDM_EXIT              0x107

/* Color Menu Choices */
#define     IDM_0                 0x108
#define     IDM_25                0x109
#define     IDM_50                0x10a
#define     IDM_75                0x10b
#define     IDM_100               0x10c

/* New dialog box */
#define     DLGNEW_RED            0x100
#define     DLGNEW_GREEN          0x101
#define     DLGNEW_BLUE           0x102


/* Window extra constants */

#define     WE_COLOR              0
#define     WE_SHADE              2
#define     WE_EXTRA              4

#define     COLOR_RED             0
#define     COLOR_GREEN           1
#define     COLOR_BLUE            2


/* Function prototypes */

int PASCAL WinMain( HANDLE, HANDLE, LPSTR, int );

HWND              MainInit( HANDLE, HANDLE, int );
```

CONTINUED

the correct menu when cursor keys are used while a pull-down menu is visible.

Following this is the MdiTranslateAccelerators function, which is responsible for translating each message according to the currently active accelerator table. Though most Windows applications have only one accelerator table, MDI applications can have several—one for the main desktop and one for each type of document window. This function automatically checks the state of the application and uses the appropriate accelerator table.

Finally, there are two utility functions contained in the MDI API—MdiGetMenu and MdiSetAccel—that are implemented as macros. These two functions are required since most applications need to define an accelerator table and access the current menu. The current menu handle is retrieved by the MdiGetMenu macro which returns it to the document window. The MdiSetAccel macro attaches an accelerator table to the property list of the document window. The accelerator table can then be used automatically when messages are translated and dispatched throughout the application.

Taken together, these eight functions represent the entire MDI API. Although you cannot use these functions with impunity, they should work well even in the most demanding applications. If carefully used, they hide most of the subtleties of MDI and let you focus on solving customer problems, not implementing yet another scheme for managing child windows. The only really nasty side effect is the installation of a system keyboard message hook. This hook intercepts cursor movement keystrokes while manipulating menus. Without this message hook it would be very difficult to

implement a truly authentic MDI keyboard user interface.

Building MDI.LIB

In order to build the MDI API library, you will have to create the files listed in **Figure 15** (*these files, not included here due to space considerations, are available for downloading from any* MSJ *bulletin board—Ed.*). In addition to these source files, you will need the Microsoft Windows Version 2.1 SDK and the Microsoft C Optimizing Compiler Version 5.1.

The library MAKE file (MDI) will compile each of the modules in the medium model and combine them into an object library using the LIB utility provided with the C compiler. The resulting library is then ready for use without modification by any medium model Windows application. If you wish, you can change the make file compilation flags and create equivalent small, compact, or large versions of the same library.

Using the MDI API

I will use the program COLORS.EXE to show how the MDI API is used in the context of a simple application. I chose this program since it will clearly demonstrate the simple and straightforward use of the MDI API. In many ways, COLORS can be considered a collection of three different, yet related programs. For one, although the three parts of COLORS share the same window procedure, they act as if they were three separate applications. Using the MDI API they are brought together into one desktop.

With the COLORS desktop, you can create a number of red, green, and blue colored document windows. The colored windows are created by using the New... option under the File pull-down menu. Each window

Figure 17 CONTINUED

```
long FAR PASCAL   MainWndProc( HWND, unsigned, WORD, LONG );
BOOL              ColorInit( HANDLE );
HWND              ColorCreate( HWND, int );
long FAR PASCAL   ColorWndProc( HWND, unsigned, WORD, LONG );
int FAR PASCAL    MainDlgNew( HWND, unsigned, WORD, LONG );
int FAR PASCAL    MainDlgAbout( HWND, unsigned, WORD, LONG );
```

COLORS.C - Source File for COLORS.EXE

```
/* COLORS.C - Colorful MDI Children */

#include <string.h>
#include <stdio.h>
#include <windows.h>

#include "colors.h"
#include "mdi.h"

/* Static variables */

/* Text for client area */
static char    *szShadings[5] = {"0 %", "25 %", "50 %",
                                 "75 %", "100 %"};

/* Titles of documents */
static char    *szTitles[3] = { "Red", "Green", "Blue" };

/* Count of each document (for titles) */
static int     wCounts[3] = { 0, 0, 0 };

/* Color & Shading table */
static DWORD   rgbColors[3][5] = {
                  {  /* RED */
                  RGB(255,255,255),     /*   0 % */
                  RGB(255,192,192),     /*  25 % */
                  RGB(255,128,128),     /*  50 % */
                  RGB(255,64,64),       /*  75 % */
                  RGB(255,0,0)          /* 100 % */
                  },
                  {  /* GREEN */
                  RGB(255,255,255),     /*   0 % */
                  RGB(192,255,192),     /*  25 % */
                  RGB(128,255,128),     /*  50 % */
                  RGB(64,255,64),       /*  75 % */
                  RGB(0,255,0)          /* 100 % */
                  },
                  {  /* BLUE */
                  RGB(255,255,255),     /*   0 % */
                  RGB(192,192,255),     /*  25 % */
                  RGB(128,128,255),     /*  50 % */
                  RGB(64,64,255),       /*  75 % */
                  RGB(0,0,255)          /* 100 % */
                  } };

/*  * First routine called by windows.  Calls the initialization routine
 * and contains the message loop. */

int PASCAL WinMain(
   HANDLE      hInst,
   HANDLE      hPrevInst,
   LPSTR       lpszCmdLine,
   int         nCmdShow )
{
   HWND        hwndColors;       /* Handle to our MDI desktop */
   MSG         msg;              /* Current message */

   /* Initialize things needed for this application */
   hwndColors = MainInit( hPrevInst, hInst, nCmdShow );
   if ( !hwndColors )
   {
      /* Failure to initialize */
      return NULL;
   }

   /* Process messages */
   while ( MdiGetMessage( hwndColors, &msg, NULL, NULL, NULL ) )
   {
      /* Normal message processing */
      if ( !MdiTranslateAccelerators( hwndColors, &msg ) )
      {
         TranslateMessage( &msg );
         DispatchMessage( &msg );
      }
   }

   /* Done */
```

CONTINUED

Figure 17 CONTINUED

```
   return msg.wParam;
}

/* * First, initialize the MDI desktop and the color document windows. Second, create
the MDI desktop and then create one red document. */

HWND MainInit(
   HANDLE      hPrevInst,
   HANDLE      hInst,
   int         nCmdShow )
{
   char        szTitle[80];      /* Title of our MDI desktop */
   char        szClass[80];      /* Class name of our MDI desktop */
   HWND        hwndColors;       /* Handle to our MDI desktop */
   WNDCLASS    WndClass;         /* Class structure */

   /* Window classes */
   if ( !hPrevInst )
   {
      /* Main window */
      LoadString( hInst, IDS_MAINCLASS, szClass, sizeof( szClass ) );

      /* Prepare registration */
      memset( &WndClass, 0, sizeof( WndClass ) );
      WndClass.style          = CS_HREDRAW | CS_VREDRAW;
      WndClass.lpfnWndProc    = MainWndProc;
      WndClass.hInstance      = hInst;
      WndClass.hIcon          = LoadIcon( hInst, "MainIcon" );
      WndClass.hCursor        = LoadCursor( NULL, IDC_ARROW );
      WndClass.hbrBackground  = COLOR_APPWORKSPACE + 1;
      WndClass.lpszMenuName   = "MainMenu";
      WndClass.lpszClassName  = szClass;

      /* Register main class */
      if ( !RegisterClass( &WndClass ) )
         return NULL;

      /* Allow each of the MDI children to do its own initialize */
      if ( !ColorInit(hInst) )
         return NULL;
   }

   /* Create our main overlapped window */
   LoadString( hInst, IDS_TITLE, szTitle, sizeof( szTitle ) );
   hwndColors = MdiMainCreateWindow( szClass,
      szTitle,
      WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
      CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
      NULL,
      NULL,
      hInst,
      NULL );

   /* Did we create successfully? */
   if ( !hwndColors )
      return NULL;

   /* Give us one red child to begin with */
   if ( !ColorCreate( hwndColors, COLOR_RED ) )
   {
      DestroyWindow( hwndColors );
      return NULL;
   }

   /* Ready */
   ShowWindow( hwndColors, nCmdShow );
   UpdateWindow( hwndColors );

   /* Done */
   return hwndColors;
}

/* Handle messages for our MDI desktop.  This includes any WM_COMMAND  messages that
are received when NO document is visible on the desktop.*/

long FAR PASCAL MainWndProc(
   HWND        hwndColors,
   unsigned    message,
   WORD        wParam,
   LONG        lParam )
{
   FARPROC     lpProc;           /* Procedure instance for dialogs */
   HANDLE      hInst;            /* Current instance handle */

   switch ( message )
   {
   case WM_COMMAND:
```

CONTINUED

is successively numbered and, via an associated pull-down menu, can be modified to display different intensities of color. Of the three types of document windows, the blue one is unique in that it is also associated with an accelerator table. By using keys 0, 1, 2, 5, and 7 you can change the intensity of the blue background color to 0 percent, 100 percent, 25 percent, 50 percent, and 75 percent respectively. See **Figure 16** for an example.

When several documents are present on the desktop, you can move from window to window using one of three mechanisms—selecting a window with the mouse, moving to another window using the keyboard user interface, or pulling down the Window menu and manually selecting a different window. Additionally, using the Window pull-down menu, you can hide the currently active document window or possibly redisplay hidden ones in a traditional MDI fashion.

You can build COLORS from the source code listed in **Figure 17**, which includes the following files:

COLORS
COLORS.DEF
COLORS.RC
COLORS.H
COLORS.C

Each reference to the MDI API is clearly identified and highlighted in **Figure 17**. The first reference to notice is in the application MAKE File. Here, COLORS is dependent on both MDI.H and MDI.LIB. In addition, the MDI library is referenced in the linkage command line, which allows COLORS to use any of the public MDI API routines we previously defined.

The next MDI reference of interest is contained in COLORS.DEF, which is where both the MdiMsgHook and

MdiDlgUnhide functions are exported. They both must be exported since they represent movable entry points used by the system. Failure to do so will cause serious problems.

The third reference to the MDI API is in COLORS.RC. Note the inclusion of the MDI.H header file and the definition of a number of MDI-related resources at the end of the file. The first of these resources, the MDI window menu, is used as a template for the Window pull-down menu. The MDI API creates a duplicate of this menu, attaching a list of the currently visible document windows at the end.

The next resource is the MDI child window accelerator table. This table, which is automatically loaded by the API, is used to implement the document window keyboard user interface. Last in the resource file is the template for the MDI Unhide dialog box. This dialog box is displayed when the Unhide... command is selected from the Window pull-down menu. With this dialog box you can select a hidden document window and have it redisplayed on the desktop. You can also change the style and characteristics of the dialog box to suite your application, although you should be careful not to alter the name and identifiers used.

Following the resource file is COLORS.C, which contains all of the C source code for the COLORS application and is structured much like any other windows program. COLORS.C (like COLORS.RC) also references MDI.H. In addition to defining all MDI related identifiers, MDI.H needs to be included since it defines function prototypes for each member of the MDI API.

The first MDI-related task COLORS performs is the creation of the main desktop window using the MdiMainCreateWindow

Figure 17 CONTINUED

```
      hInst = GetWindowWord( hwndColors, GWW_HINSTANCE );
      switch( wParam )
      {
      case IDM_NEW:
         /* New dialog box */
         lpProc = MakeProcInstance( MainDlgNew, hInst );
         switch( DialogBox( hInst, "MainNew", hwndColors, lpProc ) )
         {
         case DLGNEW_RED:
            ColorCreate( hwndColors, COLOR_RED );
            break;

         case DLGNEW_GREEN:
            ColorCreate( hwndColors, COLOR_GREEN );
            break;

         case DLGNEW_BLUE:
            ColorCreate( hwndColors, COLOR_BLUE );
            break;
         }
         FreeProcInstance( lpProc );
         break;

      case IDM_OPEN:
         break;

      case IDM_ABOUT:
         /* About dialog box */
         lpProc = MakeProcInstance( MainDlgAbout, hInst );
         DialogBox( hInst, "MainAbout", hwndColors, lpProc );
         FreeProcInstance( lpProc );
         break;

      case IDM_EXIT:
         /* Tell application to shut down */
         PostMessage( hwndColors, WM_SYSCOMMAND, SC_CLOSE, 0L );
         break;
      }
      break;

   case WM_DESTROY:
      PostQuitMessage( 0 );
      break;
   }
   return MdiMainDefWindowProc(hwndColors, message, wParam, lParam);
}

/* Register the document class. */

BOOL ColorInit(
   HANDLE      hInst )
{
   char        szClass[80];      /* Class name */
   WNDCLASS    WndClass;         /* Class structure */

   /* Get class name */
   LoadString( hInst, IDS_COLORCLASS, szClass, sizeof( szClass ) );

   /* Prepare registration */
   memset( &WndClass, 0, sizeof( WndClass ) );
   WndClass.style          = CS_HREDRAW | CS_VREDRAW;
   WndClass.lpfnWndProc    = ColorWndProc;
   WndClass.cbWndExtra     = WE_EXTRA;
   WndClass.hInstance      = hInst;
   WndClass.hCursor        = LoadCursor( NULL, IDC_ARROW );
   WndClass.hbrBackground  = GetStockObject( GRAY_BRUSH );
   WndClass.lpszClassName  = szClass;

   /* Register */
   return RegisterClass( &WndClass );
}

/* Create a document window of a given color on the MDI desktop. It loads the
appropriate menu, and accelerator table if the color is BLUE. It initializes
color and shading in the window extra words.*/

HWND ColorCreate(
   HWND        hwndParent,
   int         wType )
{
   char        szClass[80];      /* Class name for documents */
   char        szTitle[80];      /* Title for this document */
   HANDLE      hAccel = NULL;    /* Accelerator for blue doc only */
   HANDLE      hInst;            /* Current instance handle */
   HMENU       hmenuChild;       /* Handle to document's menu */
   HWND        hwndChild;        /* Handle to document */
```

CONTINUED

Figure 17 CONTINUED

```
   /* Get important info */
   hInst = GetWindowWord( hwndParent, GWW_HINSTANCE );
   LoadString( hInst, IDS_COLORCLASS, szClass, sizeof( szClass ) );
   sprintf( szTitle, "%s%d", szTitles[wType], ++wCounts[wType] );

   switch( wType )
   {
   case COLOR_RED:
      hmenuChild = LoadMenu( hInst, "RedMenu" );
      break;

   case COLOR_GREEN:
      hmenuChild = LoadMenu( hInst, "GreenMenu" );
      break;

   case COLOR_BLUE:
      hmenuChild = LoadMenu( hInst, "BlueMenu" );
      hAccel = LoadAccelerators( hInst, "BlueAccel" );
      break;
   }

   /* Create */
   hwndChild = MdiChildCreateWindow( szClass,
      szTitle,
      WS_MDICHILD,
      CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
      hwndParent,
      hmenuChild,
      hInst,
      0L );

   /* Success? */
   if ( hwndChild )
   {
      SetWindowWord( hwndChild, WE_SHADE, IDM_100 );
      SetWindowWord( hwndChild, WE_COLOR, wType );
      MdiSetAccel( hwndChild, hAccel );
   }

   return hwndChild;
}

/* Handle messages for our documents.  WM_COMMAND messages arrive at this
procedure just as if the menu were attached to this window.*/

long FAR PASCAL ColorWndProc(
   HWND        hwndChild,
   unsigned    message,
   WORD        wParam,
   LONG        lParam )
{
   char        szText[20];       /* Client area text */
   HBRUSH      hBrush;           /* Brush for filling document */
   PAINTSTRUCT Paint;            /* Paint structure */
   int         wColor;           /* Color of current document */
   int         wShade;           /* Shading of current document */

   switch ( message )
   {
   case WM_COMMAND:
      switch( wParam )
      {
      /* File menu */
      case IDM_SAVE:
      case IDM_SAVEAS:
      case IDM_PRINT:
         break;

      case IDM_CLOSE:
         PostMessage( hwndChild, WM_SYSCOMMAND, SC_CLOSE, lParam );
         break;

      case IDM_0:
      case IDM_25:
      case IDM_50:
      case IDM_75:
      case IDM_100:
         CheckMenuItem( MdiGetMenu( hwndChild ),
            SetWindowWord( hwndChild, WE_SHADE, wParam ),
            MF_UNCHECKED );
         CheckMenuItem( MdiGetMenu( hwndChild ),
            GetWindowWord( hwndChild, WE_SHADE ),
            MF_CHECKED );
         InvalidateRect( hwndChild, (LPRECT) NULL, TRUE );
         break;
      }
      break;
```

CONTINUED

function inside MainInit. This call creates an empty window that contains the default desktop menu. Not long after MdiMainCreateWindow is a call to ColorCreate, a utility function that creates a new document window using the MdiChildCreateWindow function and associates it with an appropriate accelerator table. In this case, the default action is to create a single red document window.

Once the desktop has been created and initialized, the application retrieves and processes all related messages. Like most Windows applications, this is accomplished with a simple GetMessage loop followed by the translation and dispatch of each message retrieved. In this case, however, MdiGetMessage and MdiTranslateAccelerators are used in place of the normal Windows functions.

The next reference to the MDI API occurs in the MainWndProc of COLORS. Each message that is dispatched by the message-processing loop is sent directly to the responsible window function. The MainWndProc handles all the messages that relate to the desktop window. In addition, since the desktop window is the only one that contains a menu, it also receives all menu-related messages.

The desktop message processing function traps only the file-related commands and passes the rest of the messages to the MdiMainDefWindowProc for additional handling. The MdiMainDefWindowProc in turn processes the commands in which it is interested (redirecting some to the appropriate document window function) and passes the rest on to the system via the DefWindowProc.

Throughout this process, the main window message processing function can receive menu commands belonging to any one

of the child windows. Because of this capability, it is recommended that each document window menu share a common set of commands that are applicable at the desktop level. In COLORS, these commands are all those listed under the File pull-down menu.

Note that it is only necessary to conceptually separate the desktop and document menus, not each of the associated document menus. This is because menu commands not intercepted by the desktop are only destined for the currently active document window, not for those that are inactive.

The last references to the MDI API occur in the ColorWndProc message-processing function. This function, shared among each of the colored child windows, responds to the document menu commands and paints the window background using the default color at the selected intensity level. Throughout ColorWndProc, MdiGetMenu is used in place of GetMenu. This is because the desktop window contains the menu for the document window and isn't always the immediate parent [else GetMenu(GetParent(hWnd)) would be a suitable alternative].

Like the desktop window function, ColorWndProc passes most of the messages on to the default MDI message-processing function, in this case MdiChildDefWindowProc. This function in turn processes a subset of the messages and passes the balance on to the DefWindowProc. In certain situations, messages are redirected to the desktop window and not sent directly to the system.

When you build COLORS, experiment with it and see how the internal functions respond in a variety of situations. Try and hide all the document windows or create new ones while one is in a maximized state. In partic-

Figure 17 CONTINUED

```
    case WM_ERASEBKGND:
        return TRUE;

    case WM_PAINT:
        wColor = GetWindowWord( hwndChild, WE_COLOR );
        wShade = GetWindowWord( hwndChild, WE_SHADE );
        BeginPaint( hwndChild, &Paint );
        hBrush = CreateSolidBrush( rgbColors[wColor][wShade - IDM_0]);
        FillRect( Paint.hdc, &Paint.rcPaint, hBrush );
        DeleteObject( hBrush );
        strcpy( szText, szShadings[wShade - IDM_0] );
        TextOut( Paint.hdc, 0, 0, szText, strlen( szText ) );
        EndPaint( hwndChild, &Paint );
        break;
    }
    return MdiChildDefWindowProc( hwndChild, message, wParam, lParam);
}

/* Handle the NEW dialog box.*/

int FAR PASCAL MainDlgNew(
    HWND        hDlg,
    unsigned    message,
    WORD        wParam,
    LONG        lParam )
{
    static int  iButton;          /* Keep track of radio buttons */
    int         iReturn = FALSE;  /* Return value */

    switch ( message )
    {
    case WM_INITDIALOG:
        SendMessage( hDlg, WM_COMMAND, DLGNEW_RED, 0L );
        iReturn = TRUE;
        break;

    case WM_COMMAND:
        switch( wParam )
        {
        case DLGNEW_RED:
        case DLGNEW_GREEN:
        case DLGNEW_BLUE:
            iButton = wParam;
            CheckRadioButton( hDlg, DLGNEW_RED, DLGNEW_BLUE, iButton );
            if ( HIWORD( lParam ) == BN_DOUBLECLICKED )
            {
                SendMessage( hDlg, WM_COMMAND, IDOK, 0L );
            }
            break;

        case IDOK:
            EndDialog( hDlg, iButton );
            break;

        case IDCANCEL:
            EndDialog( hDlg, 0 );
            break;
        }
        break;
    }
    return iReturn;
}

/* Handle the ABOUT dialog box.*/

int FAR PASCAL MainDlgAbout(
    HWND        hDlg,
    unsigned    message,
    WORD        wParam,
    LONG        lParam )
{
    int         iReturn = FALSE;  /* Return value */

    switch( message )
    {
    case WM_INITDIALOG:
        iReturn = TRUE;
        break;

    case WM_COMMAND:
        EndDialog( hDlg, TRUE );
        break;
    }
    return iReturn;
}
```

ular try out the keyboard user interface, moving from document to document without the aid of a mouse.

In a while you will begin to appreciate how much is going on in the background to make the interface work consistently. Yet despite the visual sophistication, there is the increased overhead required by the API. If you switch rapidly between different document windows, then the additional overhead will be readily apparent. Although in part due to the relatively simple-minded message-handling approach of COLORS (which passes everything on to the default window function), to a large degree it can be attributed to the MDI API itself.

Nevertheless, keep in mind that the MDI API implemented here was designed for clarity and readability, not for size and performance. Our internal working version of the API (on which the published library was based) implemented the full MDI specification considerably more efficiently than this one does (including Window New, Tiling, and the ability to minimize document windows). The central structure, however, remains the same—with a little tuning and enhancement, the base API presented here is capable of supporting world-class MDI applications with unparalleled ease. Coupled with a little rethinking of your current data-handling techniques, you will be able to adapt many of your existing Windows applications to the MDI user interface easily. And, perhaps best of all, with the MDI API you can accomplish this with few changes to your source code.

MDI and SAA

In addition to the interoperability benefits of MDI, one of the most significant forces behind its acceptance in the development community is IBM's SAA. Although a comprehensive overview of SAA is outside the scope of this article, we will briefly describe it to show how the SAA specification influences MDI.

SAA is a set of selected software interfaces, conventions, and protocols that serve as a common framework for application development, portability, and use across three major computing platforms—the IBM System/370, System/3X, and the personal computer.

A significant part of SAA is the CUA specification. This standard defines, in a lengthy set of rules and guidelines, what SAA-compliant user interfaces should look like and how they are to be used. The end result is a 300+ page document (available from your local IBM representative or branch office) that describes in laborious detail the SAA human/machine interface.

Why are SAA and CUA important? Regardless of what you think about them, the compelling fact of the matter is that many large corporations are attempting to settle on a common user interface that spans a variety of hardware platforms. This drive is in part motivated by the hope that users will be able to migrate easily from machine to machine without the customary learning curve associated with the transition. Corporations are starting to require vendors to provide SAA, CUA-compliant software. Microsoft has been actively trying to capitalize on this by making Windows SAA, CUA-compliant (hence all the unusual keyboard accelerators).

MDI, being an integral part of the Microsoft Windows strategy, fits into this overall standard. The net effect—and this is why MDI is important for you as a software developer—is that if you use the MDI interface (as opposed to some other scheme) in your application, your potential users will already be familiar with the interface and you could potentially sell more software. At the very least, you should take a close look at the IBM SAA, CUA specification and give it careful consideration. Personally, I have a hard time living with the constraints CUA puts on me as a developer, but I am willing to live with them if I can put my programs in front of a larger customer base.

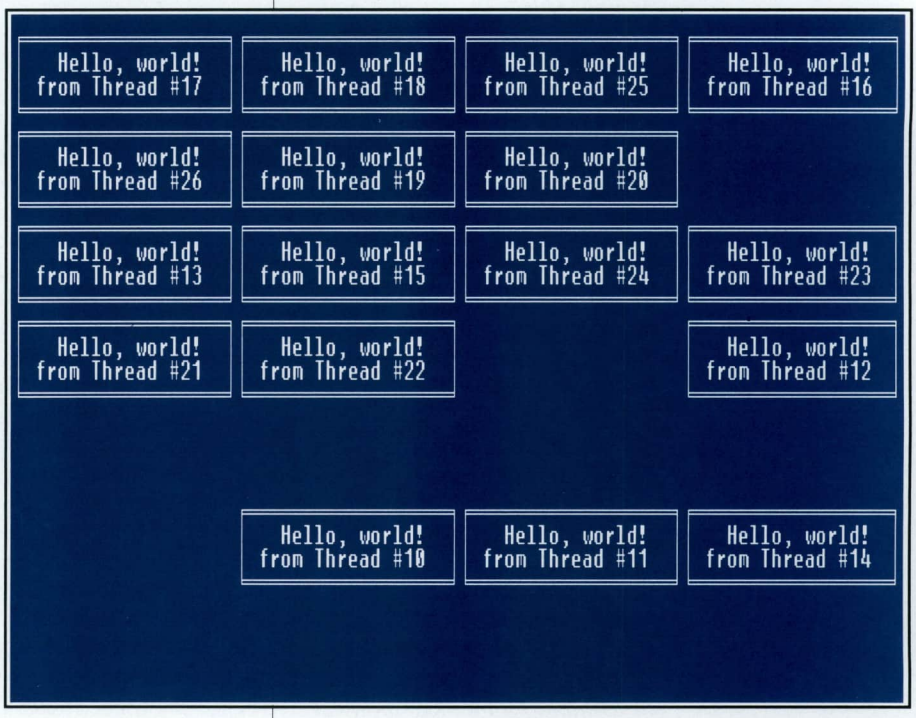
I hope this discussion has given you ideas and insights that will help you in your own development. MDI just might be the answer to some technical problem you are struggling with. As you consider MDI, realize that to a large extent it has evolved from the need for an organized way of handling multiple documents within a single desktop. This evolution has been at best troublesome and is still somewhat at odds with the underlying environment. Perhaps in the future something like the MDI API might be included in the Windows or OS/2 Presentation Manager API, saving both you and me a great deal of effort. Until that time, you have access to a little more information than you did before. ❑

Planning and Writing a Multithreaded OS/2 Program with Microsoft C

63

Richard Hale Shaw

From a programmer's perspective, OS/2 systems are a lot like a new programming language. In order to become fluent you have to begin learning the idiom by writing programs with it. In this article, we'll cover the highlights of setting up and installing the Microsoft C Version 5.1 compiler for OS/2 development and briefly look at the header files included with it. Then we'll take a closer look at the OS/2 Application Programming Interface (API) with the object of writing our first OS/2 program. Last, we'll begin to explore the world of multithreaded programming and produce a multithreaded version of the C programmer's much beloved HELLO.C program.



The Hello OS/2 multithread kernel program.

Compiler Setup

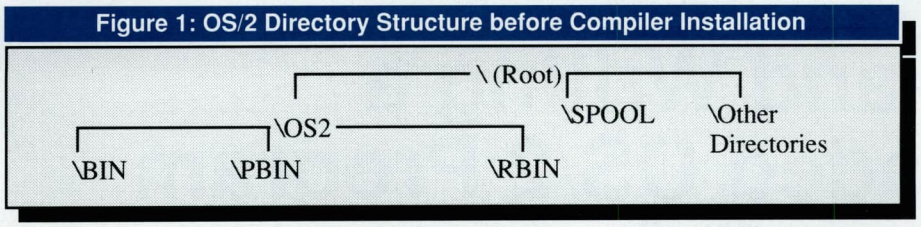
The first concern for installing Microsoft C 5.1 is to ensure that you have enough disk space. The minimum space needed for the protected mode version of the compiler is approximately 3.5Mb (which includes two libraries). If you install the full protected mode compiler, you'll need closer to 4.5Mb. If you're using the compiler to produce applications for both DOS and OS/2, you'll probably need over 6Mb. The Setup program can be instructed to install the portions of the compiler that you want.

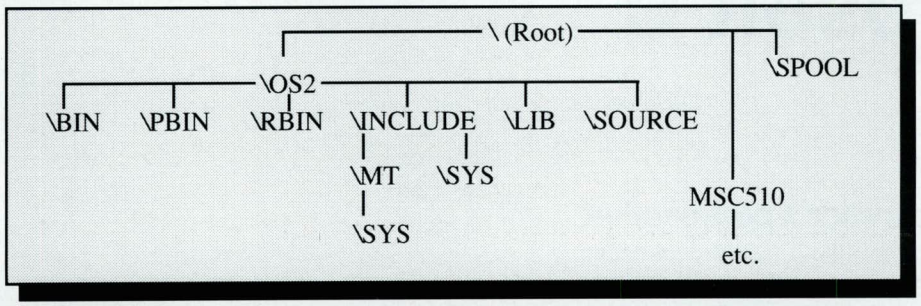
You must also decide which directory structure the compiler will use. We discussed the preferred OS/2 directory structure, shown in **Figure 1**, in the first article of this series. You may recall that the \OS2\PBIN directory contains only protected mode executables; \OS2\RBIN holds only real mode programs; and \OS2\BIN contains bound executables, that is, programs that can operate under DOS and OS/2.

If you install the compiler while you are running OS/2, you'll find that the Setup program recommends that you incorporate the traditional C compiler directories into the same structure, as shown in **Figure 2**. The \OS2\LIB directory will contain all compiler

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

Figure 1: OS/2 Directory Structure before Compiler Installation





▲ **Figure 2** **The OS/2 directory structure after compiler installation represents one possible subdirectory configuration. The make files for the article are based on this structure. Another option would be to incorporate the include, lib, and source subdirectories listed under the OS/2 subdirectory into the standard MSC510 subdirectories.**

libraries, \OS2\INCLUDE will contain the header files, and \OS2\SOURCE will be used for ancillary documentation and source files. Note that the compiler will also install an alternative set of header files under the \OS2\INCLUDE\MT directory. These are headers for the multithreaded version of the library, which we'll discuss later in this article.

Microsoft C 5.1 includes facilities for producing traditional DOS programs, OS/2 programs, and programs designed to execute under both environments (bound applications). To this end, one of the more significant features available when you install the compiler under OS/2 are combined libraries.

Previous versions of the C compiler already include a plethora of libraries, including three floating-point libraries and a graphics library. When you add to these libraries the need to keep versions of them for each memory model used, plus libraries for OS/2 development, you will find yourself running out of disk space fast. You can, however, combine the basic library for each memory model with the graphics library (if selected) and appropriate floating-point library. Combining the libraries allows you to keep only one library per memory model. The Installation/Setup program will do this automatically for you and will delete the leftover component libraries for you if you elect to do so.

Another important option is the ability to create a dual-mode or bound compiler. This is a version of the compiler that will run under both OS/2 and DOS. The Setup program will leave a copy of BINDC.CMD (BINDC.BAT if you installed the compiler while running DOS), which can be run to produce the bound version. We'll discuss the production of bound applications later, but for now keep in mind that a bound application is one that calls only the subset of API services known as Family Application Programming Interface (FAPI) functions. These are functions that are available under both DOS and OS/2. Although limiting yourself to these functions will restrict you from using OS/2 multitasking services (after all, DOS does not offer these services), it does ensure that a program will run without recompilation under both environments.

BINDC will leave a bound version of the compiler in the directory of your choice when it finishes. Since it's a bound executable, it is recommended that you have BINDC place its output in the \OS2\BIN directory (where dual-mode programs are kept) and delete the original copies of the compiler. You won't need them, and they'll just take up disk space. The dual-mode version will be bigger than the original, but you'll find it simpler to use only the one version.

The setup program produces two files: NEW-VARS.CMD and NEW-CONF.SYS. The NEW-VARS.CMD file contains the environment variable settings that should be in place when you run the compiler, based on the directory selections you made during installation. You can include these either in STARTUP.CMD or in the CMD file used for setting up your C development session through the OS/2 Program Starter [see "Using the OS/2 Environment to Develop DOS and OS/2 Applications," *MSJ* (Vol. 4, No. 1)]. NEW-CONF.SYS contains the additions that ought to be added to CONFIG.SYS (CONFIG.OS2 in older, dual-boot environments).

New Compiler Options

There are three compiler command-line options that are pertinent to OS/2 programming. The /Lr option designates the compilation of a conventional real mode executable (the default in earlier versions of the compiler). The /Lp option, however, signals the compiler to compile and link for the pro-

tected mode. When you use this switch, the protected mode version of a given library will be used. For instance, if a conventional, real mode, small memory model compilation used SLIBCE.LIB (the small model combined library with floating-point emulation), adding the /Lp option instructs the linker to use SLIBCEP.LIB (the protected mode version of the library).

The third option, /Fb, directs the linker to run BIND.EXE after linking is complete. BIND is a utility that converts protected mode programs into dual-mode applications, which can be run in either real or protected mode. With these options in mind, and using the directory structure discussed earlier, we can construct a generic MAKE file for which we use MAKE to build our first OS/2 programs (shown in **Figure 3**).

This MAKE file, which is called HELLO (since it will be used to compile HELLO.C), forms the basis for the MAKE files we'll use to build other programs. It sets up the environment variables INCLUDE and LIB for the compiler and passes a number of options to CL. Note that the MAKE file uses both /Lp and /Fb to instruct the compiler to produce a protected mode bound version of the program.

The /W3 option forces the compiler to use warning level 3, the most severe warning level available from the compiler. This option is very useful to ensure strict type checking of different objects and arguments against OS/2-type definitions. The /Zpe option combines two options: /Zp and /Ze. The former instructs the compiler to produce packed, or byte-aligned (as opposed to word-aligned), structures. The latter allows extensions to C, particularly the far, near, and pascal keywords, which OS/2 programs will reference frequently. The /Ox option turns on maximum optimization, and the /I option specifies the INCLUDE directory, which the compiler will use.

Figure 3: A Generic Make File for Bound OS/2 C Programs

```
#file: hello
#generic OS/2 MAKE file for producing 'bound' executables
#
#Currently set for making HELLO.C
#Usage: C>make hello

INCLUDE=\os2\include
LIB=\os2\lib
COPT=/Lp /Fb /W3 /Zpe /G2 /Ox /I$(INCLUDE)
#COPT=/Lp /Fb /W3 /Zpie /G2 /Od /I$(INCLUDE)

hello.exe: hello.c hello
    cl $(COPT) hello.c /link /noe

#end
```

The /G2 option forces the production of code for the 80286, which is a handy option, since 286 instructions are generally more efficient than pure 8086 instructions. Although this option restricts the program to run on either an 80286 or above, it's not a problem in the OS/2 environment, since OS/2 will only run on a machine with an 80286 or 80386 (the 80386 also executes 80286 code). However, it could cause problems if the bound executable is run under DOS on an 8088/8086.

Several additional points should be made about this MAKE file. First, note that for debugging purposes we will want to include symbolic debugging information with the /Zi option and turn off optimization with /Od. Thus, I've included the additional line of options for debugging. You can enable these options and disable the others by removing the # in front of one line and inserting a # in front of the other line of options (# is the comment marker in MAKE files). Also, note the use of the /NOE option that CL passes to the linker, which prevents the linker's extended library search. This option will ensure that the correct versions of the library functions are linked to the program.

OS/2 System Calls

OS/2 system services are available to application programs through the OS/2 API. Unlike the interrupt-based interface of DOS, the API makes all services available through system calls (also known as call-gates). While the DOS interface was limited to 256 functions (via Int 21h), there is no limit on the number of services that could be added to OS/2 in the future. And whereas DOS services required the use of registers to receive or return values, OS/2 system services pass these on the stack. Thus, all OS/2 services use the same format: if a system service is capable of returning error values, it will return a zero when successful and an unsigned non-zero integer on error. In addition, as stated earlier, the subset of API functions known as Family API (FAPI) functions will operate under both OS/2 and DOS, allowing you to write bound executable programs that operate in both environments.

When you include a call to an API function in your OS/2 program, the linker does not add the code for the function to the executable program as it would for a DOS program. Instead, it will add instructions for loading the function's code from the appropriate OS/2 dynamic-link library (DLL). When the program executes, OS/2 will load the necessary DLLs (if it hasn't already loaded them for some other application program).

Figure 4: OS/2 Header Files for Program Development

OS2.H	- Always #included in your program
OS2DEF.H	- Common definitions
BSE.H	- Base definitions, #includes the following:
BSEDOS.H	- Kernel services definitions
BSESUB.H	- Kbd, Vio, Mou definitions
BSEERR.H	- Error macros

Figure 5: OS/2 Header File Control Macros

Define this macro:	To include definitions/declarations for:
INCL_BASE	All services
INCL_DOS	Kernel services
INCL_SUB	Subsystem (Kbd, Vio, Mou)
INCL_DOSERRORS	Error macros
INCL_DOSPROCESS	Processes and threads calls
INCL_DOSINFOSEG	Information segment calls
INCL_DOSFILEMGR	File management calls
INCL_DOSMEMMGR	Memory management calls
INCL_DOSSEMAPHORES	Semaphore functions
INCL_DOSDATETIME	Date/Time and Timer calls
INCL_DOSMODULEMGR	Module management services
INCL_DOSNLS	National language services
INCL_DOSSIGNALS	Signal functions
INCL_DOSMONITORS	Monitor services
INCL_DOSSESMGR	Session management calls
INCL_DOSDEVICES	Device and IOPL services
INCL_DOSQUEUES	Queue functions
INCL_RESOURCES	Resource-support functions

MICROSOFT C 5.1 PROVIDES SIX NEW HEADER FILES, SOME OR ALL OF WHICH NEED TO BE INCLUDED IN PROGRAMS THAT TAKE ADVANTAGE OF THE OS/2 API. THESE HEADER FILES PROVIDE THE API DECLARATIONS, FUNCTION PROTOTYPES, MACROS, CONSTANTS, AND TYPE DEFINITIONS NEEDED BY AN OS/2 C PROGRAM.

Thus, the functions will be available to the program at run time.

Only one instance of an OS/2 API function will be loaded into memory, even if more than one application program is using it. Note that for FAPI functions in a bound executable program, the linker will include both the DLL calling code and the real mode executable code for the function. The former will be used when the program is operating in the OS/2 protected mode, and the latter will invoke Int 21h when in MS-DOS real mode.

Since OS/2 loads API routines from a DLL, they are located in a different segment from that of the calling routine and must be reached with a FAR call. There are four general types of parameters that can be passed to an API system service: byte (or char), word (or unsigned), double word (unsigned long), and address (far pointer). If the service requires only the value of the parameter, then a copy of it is passed (call by value). If the service requires a pointer to a parameter, however, the address is passed to the service (call by reference). Because the calls are FAR calls, addresses passed must also be FAR. To ensure that you are passing a far address correctly to a system service, you should declare the object as far, use a cast, or compile with the large data model. Meticulous use of the OS/2 object definitions found in the new header files (described below) will eliminate most problems.

You might also note that segment values that are found in far addresses are really selectors. Although they have the same segment:offset form found in far addresses under MS-DOS real mode, OS/2's protected mode requires a selector value. A selector is actually an index into a table of segment addresses and has no correspondence to a physical segment. Selectors must be used since OS/2's virtual memory management may change the physical segment location as it is moved around in memory or swapped to disk. The layer of abstraction they provide allows you to address a far object without having to know its real physical location in memory—or whether it's in memory at all (taken care of for you by OS/2).

API functions use the Pascal calling convention. This convention specifies that there be a fixed number of arguments to the function and that the arguments are pushed on the stack left to right (the order in which you specify them in your source code). In addition, Pascal calls do not require the function being called to clean up the stack. All these requirements result in smaller, faster function calls. This scheme is the reverse of the

traditional C calling convention, which pushes arguments right to left (allowing a variable number of arguments); requires the caller to clean up the stack; and generally produces slower, larger code. The OS/2 header files discussed below specify API functions as far pascal calls, so most of the time you will not have to take any steps beyond including the appropriate header files in your program. The *far pascal* convention is defined in the OS/2 header files as APIENTRY.

Additionally, API functions use the Microsoft® Windows convention of two- or three-phrase descriptive names with both uppercase and lowercase letters. Keyboard subsystem services are provided by functions whose names begin with Kbd; Mouse and Video subsystem services names begin with Mou and Vio respectively. The names of all remaining operating system services (OS/2 kernel calls) begin with Dos. Some brief examples of these include DosRead, DosCreateThread, KbdCharIn, and VioWrtTTY.

The documentation for the OS/2 API functions can be found in the OS/2 Programmer's Reference. This manual is a part of the Microsoft OS/2 Programmer's Toolkit and the Microsoft OS/2 Software Development Kit.

Header Files

Microsoft C 5.1 provides six new header files, some or all of which need to be included in programs that take advantage of the OS/2 API. These header files (listed in **Figure 4**) provide the API declarations, function prototypes, macros, constants, and type definitions needed by an OS/2 C program. As a routine part of getting started programming under OS/2, you should become intimately familiar with their contents. You'll find that most of the type definitions and structures use a naming convention similar to that used by the system services described earlier. Also, since many system services will place return values in the structures defined in these header files, OS/2 programming will be easier later if you study their contents now.

The OS/2 header files are hierarchically nested, so that you need only include OS2.H in your program most of the time. The remaining header files and definitions can be included by using various combinations of control macros (listed in **Figure 5**), which should be defined in your program before including OS2.H. This is particularly helpful when you are using only a small subset of API functions in your program, since the headers are large and compilation will proceed more quickly if you include only what the program requires.

OS2.H itself includes two header files. The first file, OS2DEF.H, contains most of the commonly used definitions, typedefs, macros, constants, and structures. The second, BSE.H, indirectly contains the base definitions for the various OS/2 subsystems (Keyboard, Video, Mouse, and Dos), plus error-handling macros by optionally including three additional header files: BSEDOS.H, BSESUB.H, and BSEERR.H.

BSEDOS.H contains definitions required for using the OS/2 kernel system service, and can be included by defining INCL_DOS before the #include for OS2.H in your program. BSESUB.H contains all the definitions required for using any of the OS/2 subsystems (Kbd, Mou or Vio) and is included by defining INCL_SUB. BSEERR.H contains all error-related macros and is included through INCL_DOSERRORS. If necessary, you can force the blanket inclusion of all API declarations and definitions by defining INCL_BASE in your program prior to the #include for OS2.H:

```
#define    INCL_BASE
#include   <os2.h>
```

Note that many commonly used components will be defined by default unless you define INCL_NOCOMMON in your program. You can also include specific kernel services components by defining the macros listed in **Figure 5**.

A First OS/2 Program

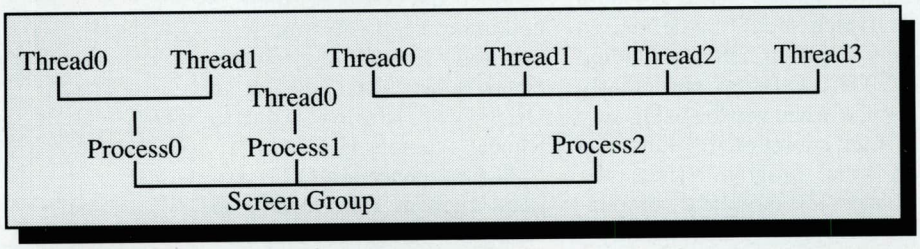
Once you have successfully installed the compiler, there is nothing to keep you from writing your first OS/2 program. **Figure 6** lists the code for an OS/2 version of Dennis Ritchie's famous HELLO.C.

At first glance, an OS/2 program such as this version of HELLO.C might appear extremely bizarre. It certainly does not resemble the Kernighan and Ritchie version that we've all come to know and love. Nevertheless, it accomplishes many of the same purposes. It allows us to begin to explore the OS/2 API; introduces us to a real use of some of the header files, defines, and functions; and most of all, gets us started writing our first real OS/2 program.

A second glance will reveal the familiar structure so often used to write maintainable, efficient C programs. You'll note the use of INCL_SUB and INCL_DOS to include function prototypes for the single call to the Vio subsystem and the single kernel call in the program. The printf call has been replaced with a call to VioWrtTTY, which prints a string on the logical screen used by our program (all video access in OS/2 is done through a logical screen group). Note that this function requires both the address of the string and the string length and that it appropriately handles the C escape

Figure 6: A Dual-Mode OS/2 Version of HELLO.C

```
/* hello.c RHS 10/14/88
 *
 * 1988 OS/2 version of    K&R's hello.c
 */

#define     INCL_SUB          /* for Vio calls used */
#define     INCL_DOS          /* for DosExit call used */
#include    <stdio.h>
#include    <string.h>
#include    <os2.h>

void main(void);          /* function prototype */

void main(void)
    {
    char *hello_str = "Hello, world!\r\n";
    int    len = strlen(hello_str);

    VioWrtTTy(hello_str,len,0);     /* print the message */
    DosExit(EXIT_PROCESS,0);        /* exit the program */
    }
```



▲ **Figure 7** **The OS/2 multitasking model represents its most significantly different feature from DOS. Note that ultimately the smallest unit of execution is a single thread. Each process must consist of at least one thread, although it can consist of more. All of the processes shown here share the same logical screen and keyboard and thus are part of the same screen group. If a process needs a different screen/keyboard combination, it must be started in a different screen group.**

sequences \r and \n, to generate a carriage-return/line-feed combination. Also note that all Vio calls require a zero for their last parameter.

After printing the string, a kernel call to DosExit terminates the entire program. The first parameter specifies whether the entire process or the current thread should be terminated (more on this in the next section). The second parameter is the exit code, which is passed to the parent process and is identical to the one that is passed in exit, the traditional C termination function.

Finally, note that this program can be compiled, linked, and bound to create a dual-mode application. Thus, the same executable program will run, unmodified, under either OS/2 or DOS. Note that the MAKE file for this program can be found in **Figure 3**. If you haven't already compiled and run a C program using API calls, I suggest you type in HELLO.C, compile it, and run it. It will certainly help make OS/2's magic more real and prepare you for our next step: the world of multithreaded programs.

Multiple Threads

The single most significant difference between OS/2 and DOS are the former's facilities for multitasking. Multitasking will dramatically increase the efficiency of most applications that are designed to take advantage of it. However, not only do OS/2's facilities allow for the simultaneous execution of more than one program, but what's more they permit OS/2 to execute different parts of the same program at the same time.

The OS/2 multitasking model, shown in **Figure 7**, is built of threads, processes, and screen groups. A thread is the smallest unit of execution, a piece of code dispatched by the system. Threads are organized into a process, or the portion of a program that controls the ownership of resources, such as files, memory, and threads. A process is composed of at least one thread and may consist of as many as 255 separate threads. A process's main thread is the one in which execution begins. Note that though a process may use a thread to manage a resource, a thread by itself does not own any resources. A thread inherits the environment (open files, and so on) of which it is a part and shares the same code and data segments as its parent process. Collectively, the processes that share the same logical keyboard and screen are a part of the same screen group.

The model allows for multitasking at all three levels (screen group, process, and thread). In this article, however, we're primarily concerned with the simultaneous execution of threads within the same process. Programs that execute in such a manner are called multithreaded programs.

Multiple Thread Execution

OS/2 lets concurrently executing threads share a single computer's CPU through the use of a preemptive, priority-based task scheduler (later editions of OS/2 will take advantage of multiple-CPU architectures). In reality, only one thread at a time is executing, but the CPU's attention turns so quickly from one thread to another that the threads appear to be executing at

the same time—as long as the applications that use multiple threads do not abuse system resources and CPU time in their multithreaded code.

The task scheduler controls which thread gets slices of CPU time and how much time is doled out to the thread. A thread's priority controls its access to the CPU (if it has a higher priority, it will get more CPU time relative to other threads). Thus, a previously idle, higher priority thread that is ready to run can preempt CPU time from a lower priority thread that is currently executing. On the other hand, a lower priority thread must wait until all higher priority threads are idle before the scheduler will give it CPU time.

The OS/2 task scheduler employs three categories of priorities for scheduling tasks. The highest priority is time-critical, which should be used by tasks that must respond to some type of regularly occurring event (like a communications stream or keyboard input). The second category, regular, is the default priority for a new thread and should be used by most normal threads. The last category, idle, is for threads that should execute when there are no higher priority tasks that are ready or able to execute (such as a print spooler). Note that there are 32 levels of priorities in each category and that the default priority for foreground processes is regular, level 0, whereas processes that run in the background are also regular but have a lower priority level. In general, foreground tasks are given a higher priority than background tasks.

The task scheduler always runs the highest priority thread that is capable of executing. If two or more threads with the same priority are ready to run, the scheduler will evenly grant them CPU time on a round-robin basis. If a thread is blocked—that is, it is waiting until some event occurs—OS/2 will suspend it and run another thread. Note that the TIMESLICE= statement in CONFIG.SYS controls the minimum and the maximum time slice values used by OS/2.

Figure 8: A Simple Keyboard Thread Program

```
/*
 * Simple keyboard thread example
 *
 * This program illustrates how a process might start
 * a keyboard thread which will terminate the process when
 * the user presses the Esc key.
 *
 * The program starts a keyboard thread which blocks on keyboard
 * input and terminates the entire program when the user presses
 * the Esc key.  All other keys are ignored and thrown away.
 */

#define    INCL_DOS
#define    INCL_SUB

#include   <os2.h>
#include   <mt\stdio.h>
#include   <mt\process.h>

#define    ESC    0x1b
#define    TRUE    1
#define    THREADSTACK    512

char keythreadstack[THREADSTACK];

void keyboard_thread(void);
void main(void);

void main(void)
    {
    TID    threadid;

    if(DosCreateThread(keyboard_thread, &threadid,
          &keythreadstack[THREADSTACK-1]))
        exit(-1);

    while(TRUE)    /* replace this with
          ;           code for main program */
    }


void keyboard_thread(void) /* keyboard thread code */
    {
    KBDKEYINFO keyinfo;

    while(TRUE)
        {
        KbdCharIn(&keyinfo,IO_WAIT,0);  /* wait for keystroke */
        if(keyinfo.chChar == ESC)      /* if ESC pressed, break */
            break;
        }
    DosExit(EXIT_PROCESS,0);           /* terminate the process */
    }
```

```
unsigned DosCreateThread(void (far *) functionptr(void),
                         TID *threadidptr, void *stack);
```

▲ **Figure 9 DosCreateThread, whose function prototype is shown above, can create a new thread whose code is found in the function pointed to by functionptr. The ID of the new thread is placed in threadidptr and the thread will use the stack whose top begins at the address pointed to by stack.**

Figure 10: Make File for Simple Keyboard Thread Program

```
#
# make file for key.c example found in Figure 8
#

INCLUDE=\os2\include\mt
LIB=\os2\lib
COPT=/Lp /W3 /Zp /Zie /Zl /G2s /I$(INCLUDE)  /Alfw

key.exe: key.c key
    cl $(COPT) key.c /link /co llibcmt
```

As mentioned above, a thread is blocked when it is waiting for an event to occur. A thread is running when it is being given CPU time slices. If a thread is no longer blocked but hasn't yet been given CPU time slices, then it is said to be ready to run.

Planning a Multithreaded Program

As mentioned above, a program or process can have more than one thread of execution. Using multiple threads lets it manipulate and control machine resources more efficiently than would a single-threaded application. For instance, printing, communications file transfers, and database sorting all are tasks that can be performed simultaneously by separate threads of execution while the main thread of an application program continues to serve the end user. Furthermore, in the multitasking environment of OS/2, a multiple-thread architecture is essential to help ensure that no single task will hog machine resources. Gordon Letwin, OS/2 architect, first identified this libertarian approach to sharing resources: programs must obey the rules in order to work together. This approach makes it obvious who the violator is when a program abuses the environment, and permits the system to operate in the most efficient manner possible.

Thus, planning a multithreaded application presupposes that the tasks that the program will perform can best be implemented by using multiple threads of execution. There are a number of caveats that help in planning such a program, which can be summarized as: never assume that OS/2 will execute a multitasking program or routine in a specific way. Corollaries of this rule include:

- Never assume that one routine will execute before another.
- Never assume that a given routine will execute for a given number of milliseconds.
- Never assume that future versions of OS/2 will schedule tasks the same way the current one does.
- Never assume that different threads will always be competing for CPU time. Future versions of OS/2 will run on parallel processors, allowing different threads to execute simultaneously. While you and I both know that they don't really execute concurrently at this point, treat them as if they already do.
- Never assume any direct correlations between CPU time slices and CPU cycles.
- Never assume that OS/2 can guarantee which thread will execute first at any point during the course of your program. Although the main thread is always the first thread to execute in the program, there are no guarantees on execution order once the second thread has begun.

Obviously, the last caveat doesn't mean that there aren't any controls available for manipulating events or serializing access to resources among multiple threads: that's where semaphores and priority levels come in. We'll discuss these later, but for now remember that when writing a multithreaded application, never assume!

A Multithread Program

The process of creating an additional thread is fairly simple in itself. For instance, suppose you wanted an application to run uninterrupted, ignoring all keyboard input until the user presses the Esc key, at which point the application would terminate. In a DOS application, there are two possible solutions: you could design the program to occasionally poll the keyboard, which is cumbersome in a complex application, or your program could trap the BIOS keyboard interrupt with code that would signal the main program if the Esc key is pressed.

Under OS/2 these approaches are neither necessary nor relevant. Instead, you can start a thread that blocks on keyboard input (that is, it waits until there is keyboard activity). When the user presses a key, the thread examines the key. If the key is any key other than Esc, it will continue to block. If the key is Esc, the thread will terminate the entire process.

A brief examination of the program shown in **Figure 8** will make this procedure clearer. The main thread begins where all C programs begin, with the call to main. The main thread creates the keyboard thread with the call to the API kernel function, DosCreateThread.

Note that DosCreateThread takes several parameters, as shown by the function prototype in **Figure 9**. The first parameter is the address of a function that contains the code for the thread. Here, the function is innocently named keyboard_thread. The

second parameter is the address of a variable into which OS/2 will place the thread's identifier once it has successfully created the thread. The final parameter is the address of the top of the stack allocated for the thread, which should be at least 512 bytes in size. Note that in order to pass the address of the top of the stack area, keythreadstack, we must pass the address of the last byte of keythreadstack in the manner shown.

As mentioned earlier, all API functions return nonzero on failure, so we know that a new thread was successfully created if DosCreateThread returns a zero value. The newly created thread will immediately begin execution of the code found in keyboard_thread. Although the call to DosCreateThread could have been placed almost anywhere, calling it early in the program ensures that the program will terminate immediately if the user presses the Escape key. Note that the while(TRUE) statement can be replaced with whatever code you would use for processing in the main thread.

Now let's take a look at the keyboard_thread function. The code for a thread should always be contained in a single function. You cannot incorporate the code for this function into main or any other function, nor can you call a thread function directly. Thus, thread functions are really an OS/2 extension to C.

Note that keyboard_thread begins executing immediately after creation and then goes into the loop to call the Kbd subsystem function, KbdCharIn. This function's first parameter is the address of an OS/2 KBDKEYINFO structure, which will contain information about the keys pressed upon return (this structure will be discussed in detail in an upcoming installment of this

Figure 11: USING DOSEXITLIST

OS/2 lets a process establish a set of routines that will always be called when the process terminates.Typically these routines are functions that free the process's resources (such as closing open files). Regardless of how and when the process terminates, OS/2 will execute the functions upon termination. An application can "register" functions that OS/2 will execute, thus ensuring an orderly shutdown and disposal of the process's resources in spite of the unexpected termination of the process.

The kernel function, DosExitList, registers the termination functions with OS/2, and terminates the functions themselves. The function prototype for DosExitList is:

```
unsigned DosExitList(unsigned code,void far
    *fptr(unsigned));
```

When registering the functions with OS/2, the first parameter can be either EXLST_ADD or EXLST_REMOVE, which add or remove a function from the list, respectively. By allowing a process to dynamically add and remove functions from the termination list, a process can control the destiny of its resources after its death (in a way, not unlike a human will). A process can remove the functions from the list prior to normal termination if they are no longer needed. The second parameter is, obviously, a pointer to the termination function being registered.

When OS/2 begins execution of the termination functions, the process and all of its threads have been destroyed, with the exception of the thread executing the DosExitList functions. OS/2 will transfer control to each function registered, but in no particular order. Once OS/2 has executed all the registered functions, the process ends.

The termination functions registered with DosExitList must be defined in the process (that is, in its code segment) and should be as short and fail-safe as possible. A termination function may call any OS/2 system function with the exception of DosCreateThread and DosExecPgm.

A skeleton definition of a termination function follows:

```
void far termfunc(unsigned code)
    {
    if(code != TC_EXIT)
        {
        .
        /* do cleanup here  */
        .
        }
    DosExitList(EXLST_EXIT,0);
    }
```

Note that a termination function has one parameter and no return value. The parameter will always be one of the following:

TC_EXIT	- normal exit
TC_HARDERROR	- hard-error abort
TC_TRAP	- trap operation
TC_KILLPROCESS	- unintercepted DosKillProcess

This allows the termination function to detect whether a normal termination of the process has occurred. It can also determine what actions the function should take.

Termination functions must terminate themselves by calling DosExitList(EXLST_EXIT,0). They cannot execute a 'return' (explicitly or implicity, by falling past the curly brace), or the process will hang and never terminate. The EXLST_EXIT code tells OS/2 that the termination processing is complete, and that it should call the next function on the termination list.

Using The Multithreaded Library: DosCreateThread vs. _beginthread

Figure A: Sketch of a multithreaded printf, Version 1

```
void printf(char *fmt,...)
    {
static long printfSem = 0L;
static char formatbuffer[BUFSIZ];

DosSemRequest(&printfSem,-1L);
                    :
DosSemClear(&printfSem);
}
```

Figure B: Sketch of a Multithreaded printf, Version 2

```
#define    MAXTHREADS    32
void printf(char *fmt,...)
{
static long printfSems[MAXTHREADS] =
    {0L,0L,0L,0L,0L,0L,0L,0L,0L,0L,
     0L,0L,0L,0L,0L,0L,0L,0L,0L,0L,
     0L,0L,0L,0L,0L,0L,0L,0L,0L,0L,
     0L,0L};

char formatbuffers[MAXTHREADS][BUFSIZ];
int semno;

for(semno = 0; semno < MAXTHREADS; semno++)
    if(!DosSemRequest(&printfSems[semno],0L))
        break;
assert(semno < MAXTHREADS);
                    :
DosSemClear(&printfSem[semno]);

}
```

Figure C: Function Protoype for _beginthread and _endthread

```
#include<mt\process.h>
#include<mt\stddef.h>

int cdecl far _beginthread(
          void (cdecl far *start_address) (void far *),
          void far *stack_end,
          unsigned stack_size,
          void far *arglist);
                    :
void far cdecl _endthread(void)
```

_BEGINTHREAD OFFERS MORE OF A C-LIKE APPROACH TO CREATING A NEW THREAD.

The OS/2 API interface for creating and terminating threads works through the kernel functions DosCreateThread and DosExit. Unfortunately, although the code for a thread must be contained within a function, you cannot pass parameters to it by using DosCreateThread. So if you prefer more of a C-like interface to write multithreaded programs, or you're going to use the multithreaded standard library, LLIBCMT.LIB, you should be familiar with an alternative interface via _beginthread and _endthread. If you plan on calling standard library functions from within your thread function, it's imperative that you use LLIBCMT.LIB.

The Case of printf

The discussion of printf in the main text illustrates the need for this interface. A function like printf uses a large internal buffer for formatting its output string. Although this buffer is adequate if a single thread is executing printf's code, the outcome is unpredictable if more than one thread is trying to execute printf at the same time. There are two ways that printf can be written to resolve this: you can use a semaphore to permit only one thread at a time to execute the code for printf, or you can use a set of semaphores to allow a finite number of threads to access printf simultaneously.

Let's take a closer look at these two solutions. With the first method (sketched in **Figure A**), a semaphore is set at the beginning of printf and cleared at the end. This approach serializes the code for printf so that only one thread can execute it at a time. Unfortunately, that means that any time a thread calls printf, it will block if some other thread is executing the printf code and will remain suspended until the thread using printf clears the semaphore. Additionally, there is no guarantee that the next thread will be allowed to execute printf, nor can the scheduler ensure which thread that will be. OS/2 cannot guarantee that the next thread you want to call printf will be the one allowed to execute it with this approach. Thus, a scenario might develop where a thread of lesser priority might constantly be preempted by higher priority

threads executing printf—and that can cause undesirable visual results.

Alternatively, the second method limits the number of threads that can simultaneously execute the code for a function like printf. However, it offers no possibility of collision between threads competing for access to printf's code. With this approach (illustrated in **Figure B**), printf is structured to provide a fixed set of formatting buffers, with access to each controlled by a different semaphore. Consequently, every thread is given its own private buffer while executing the code. The catch is the limit on the number of buffers and therefore the limited number of threads that can gain access.

This limit is one reason _beginthread is provided. The function offers more of a C-like approach to creating a new thread by allowing you to pass parameters to the thread and returning the thread ID when successful. But more specifically, it restricts the calling process to 32 threads, far less than the 255 threads that can be created by DosCreateThread. For many applications, however, 32 threads will be more than enough, allowing the multithreaded library, LLIBCMT.LIB, to operate on the assumption that no process will consist of more than 32 threads at a time. For this reason, _beginthread is available only when you use this library, and you should use it instead of DosCreateThread when writing a program that uses this library.

Using _beginthread

How does _beginthread work? From the function prototype shown in **Figure C**, you can see that, like DosCreateThread, it requires the address of a function that contains the code for the thread. Unlike DosCreateThread, however, _beginthread takes not the address of the top of the stack, but the address of a stack area as you would declare it in your C program (that is, the bottom of the stack). Therefore you must provide it with the stack size (the third parameter). Last of all, _beginthread takes a parameter that makes it more valuable than DosCreateThread in some cases: an argument parameter for passing arguments to the thread function itself. An example of this use of _beginthread can be found in the multithreaded HELLO0.C program. Finally, you'll notice that _beginthread itself returns –1 on error or the thread ID of the new thread.

Obviously, _beginthread must at some point call DosCreateThread. In fact, it even calls DosCreateThread when you have exceeded the number of threads allowed by the multithreaded library. The only way it can limit the number of threads to a process is to get the thread ID returned from its own call to DosCreateThread and return –1 if the ID is greater than 32. However, this reveals the use of two undocumented assumptions about OS/2: that DosCreateThread will always return the lowest available thread ID and that OS/2 will reuse thread IDs of previously terminated threads.

Linking LLIBCMT.LIB

To use _beginthread and its counterpart to DosExit, _endthread (also shown in **Figure C**), make sure that the LLIBCMT.LIB and DOSCALLS.LIB libraries are available in the current directory or in the directory pointed to by the LIB variable (in the environment or in your MAKE file—see the MAKE file for HELLO0.C as an example). The DOSCALLS.LIB library is required, since _beginthread, _endthread, and some of the other library functions will make calls to OS/2 API functions. LLIBCMT.LIB should be used in place of any other run-time libraries.

In addition, you may want to change your INCLUDE variable (again in the environment or in your MAKE file) to point to the MT directory that the compiler installed beneath the standard INCLUDE directory. This directory contains copies of the standard header files and should be used for creating multithreaded programs. You can set the INCLUDE variable on the compiler command line with the /I option as an alternative. Incidentally, the prototypes for _beginthread and _endthread can be found in PROCESS.H.

Limitations

Multithreaded code must make several assumptions as it executes. First, all code and data addresses are expected to be far. In addition, the code must assume that the data segment is fixed but should *not* assume that the stack and data segments are the same. In addition, conventional run-time stack checking must be turned off, since it is taken care of for each thread in a multithreaded program. You can conveniently use compiler switches to take care of these concerns by employing the /Alfw and /G2s options. Also, either the /Zl compiler option or the /NOD linker option should be used to prevent the default library search by the linker. The MAKE file for HELLO0.C illustrates this usage and can easily be adapted to compile and link your own multithreaded programs.

Finally, remember that multithreaded programs cannot be bound into dual-mode applications since there is no facility in MS-DOS for simultaneous execution of multiple threads of code.

series). The chChar member of the structure will contain the ASCII value of the key pressed. The second parameter specifies how long the function should wait until the user presses a key. Finally, passing IO_WAIT will cause the function to wait for the key indefinitely, in turn causing OS/2 to suspend the calling thread until that thread's screen group receives a key from the user.

Once the user presses a key, OS/2 will wake up the thread and return from the KbdCharIn call. The thread then examines the key and breaks out of the loop if it is the Esc key, calling DosExit to terminate the entire process. You might note that the thought behind the function is similar to object-oriented programming: the keyboard_thread function completely encapsulates the keyboard control and program termination sequence. The main program doesn't have to know how it works or what it does—that's taken care of for it by the thread itself.

A program's main thread must be kept alive until all other thread activity has finished or can be terminated. When the main thread dies, the other threads die. Thus, if you insert code in the main thread that will exit the process, the keyboard thread (and any other threads) will be destroyed with the main thread. If a secondary thread like the keyboard thread shown here reaches the end of its code without calling a termination routine, it will die, but it will not affect other threads. Threads can explicitly terminate themselves by means of a call to DosExit:

```
DosExit(EXIT_THREAD,
term_code);
```

or a thread can terminate the entire process via:

```
DosExit(EXIT_PROCESS,
term_code);
```

The MAKE file for this example can be found in **Figure 10**. The /Gs or /G2s options are used to turn off standard run-time stack checking, since the run-time stack checks will report false stack overflow errors in code for the thread. If, however, you wish to isolate this to the thread code itself, you can insert a #pragma:

```
#pragma check_stack(off)
/* thread function is
placed here */
#pragma check_stack(on)
```

which will turn off run-time stack checking for the code inserted between the #pragmas.

If the main thread needs to close files or shut down other processes or resources before terminating, the thread function could set a semaphore (as a flag) that could be checked occasionally by the main thread, instead of terminating outright. Another alternative would be to use DosExitList, shown in **Figure 11**.

Reentrance Issues

There is one important constraint to consider when writing multiple-thread programs. It is the problem encountered when more than one thread tries to simultaneously execute code that is being used by another thread. This problem does not arise with the OS/2 API functions: their code has been written for a multitasking environment. The standard C library and your own functions, however, are another matter.

Consider a scenario like the following: The standard library routine printf uses an internal buffer to format the characters that it will write to the standard output. Suppose one thread is in the middle of executing printf, with half of the buffer formatted, when another thread begins to execute printf's code, overwriting the buffer with its own characters—and resulting in chaos. This behavior is, of course, intolerable, and unless you're going to implement your own set of semaphores to control the use of every library routine (which *is not* a viable solution), you're going to end up with a mess on your hands.

There are, fortunately, two solutions to this problem. First, when writing multithread programs, refrain from using any standard library routines in any but the main thread. This guarantees that only one thread will be using the standard library routines at a time. With the exception of a few reentrant routines, the standard library routines are not reentrant and are designed for single-threaded execution (see **Figure 12**). If you must control access to a specific routine from the standard library (or one of your own routines), there are two ways you can do it:

- Use the DosEnterCritSec API call to temporarily freeze the other threads. Although this approach does work, it isn't the best solution and is mentioned here for informational purposes; there are too many things that can go wrong.
- Use a semaphore to control access to a function. This solution is more practicable, since only the threads that are trying to access the shared code will be affected—the rest will continue to execute (DosEnterCritSec, on the other hand, will freeze all other threads).

If you find it impossible to live without the standard library functions, there is one additional alternative: the multithreaded standard library. While earlier versions of the C compiler required that you distinguish between reentrant and nonreentrant functions, Microsoft now supports a version of the standard library, LLIBCMT.LIB, that is completely reentrant and supports multiple threads. If

you write programs that use this library, you must use the new _beginthread and _endthread functions that are contained in the library (instead of using DosCreateThread and DosExit). (For more information, see the sidebar "Using the Multithreaded Library: DosCreateThread vs. _beginthread.")

Finally, you can write your own functions to accommodate multiple threads. If you choose to do so, there are at least three key guidelines to follow. First, multithreaded functions cannot disable interrupts or issue an INT instruction. Second, they should not alter the contents of a segment register or perform segment manipulations. Last, there must be strict controls on access to global or static data by functions that can be called by multiple threads (alluded to earlier in the discussion of the standard library printf routine). The preferred mechanism for incorporating these controls into your program is OS/2 semaphores.

Thread Control

A detailed discussion of the use of OS/2 semaphores can be found in "Using OS/2 Semaphores to Coordinate Concurrent Threads of Execution," *MSJ* (Vol. 3, No. 3), but we will briefly reiterate some of the points made in that article that are pertinent here.

Although OS/2 offers several facilities for interprocess communication (pipes, queues, signals, and shared memory), semaphores are the preferred method of coordinating multiple threads. You can use them to serialize access to pieces of code or resources that cannot be shared. Alternatively, you can use semaphores when you need to have one thread signal to another that an event has occurred. Of the several types of semaphores OS/2 offers, RAM semaphores (used by threads in

abs	labs	memset	strcmpi	strnset
atoi	lfind	mkdir	strcpy	strrchr
atol	lsearch	movedata	stricmp	strrev
bsearch	memccpy	putch	strlen	strset
chdir	memchr	rmdir	strlwr	strstr
getpid	memcmp	segread	strncat	strupr
halloc	memcpy	strcat	strncmp	swab
hfree	memicmp	strchr	strnicmp	tolower
itoa	memmove	strcmp	strncpy	toupper

▲ **Figure 12** **The following routines in the Microsoft C Standard Library are reentrant and can be called by more than one thread of a multithreaded process simultaneously.**

```
#define    INCL_DOS
#define    INCL_SUB
#include<stdio.h>
#include<process.h>
#include<os2.h>

#define    ESC    0x1b
#define    TRUE 1

void keyboard_thread(void);
void main(void);

#define    THREADSTACK    512

char keythreadstack[THREADSTACK];
long CountSem =    0L;
unsigned count = 0;

void main(void)
    {
    TID    threadid;

    DosSemClear(&CountSem);

    if(DosCreateThread(keyboard_thread,&threadid,
          &keythreadstack[THREADSTACK-1]))
        exit(-1);

    while(TRUE)    /* insert code for main program here */
        {
        DosSleep(100L);
        DosSemRequest(&CountSem,-1L);
        if(count > 3)
            break;
        DosSemClear(&CountSem);
        }
    }

void keyboard_thread(void)  /* keyboard thread code */
    {
    KBDKEYINFO keyinfo;

    while(TRUE)
        {
        KbdCharIn(&keyinfo,IO_WAIT,0);  /* wait for keystroke */
        if(keyinfo.chChar == ESC)      /* if ESC pressed, break */
            {
            DosSemRequest(&CountSem,-1L);
            count++;
            DosSemClear(&CountSem);
            }
        }
    DosExit(EXIT_PROCESS,0);          /* terminate the process */
    }
```

▲ **Figure 13** **The keyboard scan program from Figure 8 is modified by using semaphores .**

Figure 14: HELLO0.C—A Multithreaded Version of HELLO.C

```
/* os2hello.c by RHS, 10-14-88
 *
 * OS/2 and 1988 version of K&R's hello.c
 * demonstrates multiple threads
 */

/*
This program provides an introduction to the use of threads and
semaphores under OS/2. It divides the screen up into a series of
logical frames. Each frame is a portion of the screen that is managed
(written to) by a single thread. The exact number of frames will
depend on the current screen length (25, 43 and 50 lines). Each thread
has its own data from which it knows where the frame can be found on
screen. This includes a semaphore which signals the thread when to
proceed. These elements can be found in the FRAME data type.

Upon receiving a signal from its semaphore (i.e., the semaphore has
been cleared), the thread either draws a message on the frame or
clears the frame, and reverses the flag that determines this. Then it
again blocks until its semaphore has been cleared again.

The main program thread starts by setting up the frame information:
checking the screen size, determining the number and size of the
frames. It also "randomly" selects the order in which the frames will
appear.

Then it sets each thread's semaphore and initiates each thread
(remember the threads will block until their semaphores are cleared.

Finally, the main program goes into an infinite loop, clearing each
thread's semaphore, sleeping for at least 1 millisecond, and then
continuing to the next thread. Thus the threads asynchronously call
the VIO subsystem to draw or clear each frame, while the main program
thread continues.

An optional parameter can be passed to set the number of milliseconds
passed to DosSleep, allowing the operator to more accurately "see" the
order in which the frames appear/erase. This value must always be at
least 1 to allow the main program thread to give time to the CPU
scheduler.

A call to _beginthread() early in main() sets up a thread to monitor
keyboard input. This thread blocks until a key is pressed, then
examines the key, and if the key is the Escape Key (27 decimal or
1bH), the thread calls DosExit to kill the whole process.
*/

#define  INCL_SUB
#define  INCL_DOSPROCESS

#include <os2.h>
#include <mt\stdio.h>
#include <mt\string.h>
#include <mt\assert.h>
#include <mt\stdlib.h>
#include <mt\process.h>

#if !defined(TRUE)
#define TRUE    1
#endif

#if !defined(FALSE)
#define FALSE   0
#endif

#define LINES25     4         /* height in lines of frames*/
#define LINES43     6
#define LINES50     7


#define MAXFRAMES    28       /* limited to max frames possible */
#define RAND()       (rand() % maxframes);
#define THREADSTACK  400      /* size of stack each thread*/
#define IDCOL        15
#define ESC          0x1b
```

CONTINUED

the same process) are the simplest to implement and are the easiest to deal with in terms of our first multithreaded program.

The MS-DOS operating system is a single-tasking environment: only one thread operates at a time. A program executing under DOS has considerable control over a resource, including the ability to disable interrupts and ensure uninterrupted access to it. Signaling between processes in the DOS environment is easy, since a global variable or flag can be used to coordinate different pieces of code. You can have one process wait while the flag is set, that is, until another process clears the flag. Once the flag has been cleared, the process continues, setting the flag for itself, safe in the knowledge that it alone has access to a particular resource, including the flag variable used for signaling.

Under OS/2, this type of signaling is not possible, since there is no way (without controls) to guarantee the order in which threads will execute. Further, one thread may be reading a flag while another may be setting or clearing it, or a second thread might end up setting the flag between the moment that the first thread stopped waiting and began to set the flag itself. Therefore, more than one thread might end up with access to the same resources. The outcome of such a situation is predictably disastrous. Furthermore, continually checking the value of such a flag uses the CPU unnecessarily, lowering the efficiency of the system.

Semaphores provide an elegant alternative solution to these problems in the context of the OS/2 multitasking environment. In one uninterruptible step a semaphore kernel call can test and set a semaphore. Thus, the semaphore controls access to a shared resource and allows one

task to signal another that an event has occurred.

To demonstrate a simple use of semaphores, suppose we added a facility to the keyboard thread program shown in **Figure 8**. This facility causes the keyboard thread to increment a counter every time the user presses the Esc key (instead of terminating the program). The main thread looks at the counter periodically, and as soon as the counter is greater than a certain value (say, 3), the main thread will terminate the program.

The problem in OS/2's multithreaded environment concerns the serialization of a resource, specifically the counter variable that more than one thread may be sharing. Here's where the semaphore comes in: by using a semaphore, we can serialize the access to the counter variable, so that only one thread at a time actually reads or writes it.

The revised listing, shown in **Figure 13** (it uses the same MAKE file mentioned earlier and shown in **Figure 10**) illustrates the solution. It creates a semaphore variable, CountSem, which the main thread clears with the call to DosSemClear. The while loop has been expanded to handle the reading of the counter variable. The main thread sleeps for 100 milliseconds (about three 32-millisecond time slices), then calls DosSemRequest to gain access to CountSem. The -1L parameter causes the function to block the calling thread until the semaphore has been cleared—that way it will not be able to access the counter variable if the keyboard thread has already gained control.

Next, the main thread evaluates the counter thread, breaks out of the loop, and terminates the program if the counter is greater than 3. Otherwise, it clears the semaphore (giving up ownership of the resource, the

Figure 14 CONTINUED

```
char *blank_str = "                                    "; /* string for
                                                          blanking frame */


    /* frame data */

char *hello_str25[LINES25+1] =
    {
    "╔═══════════════════╗",
    "║   Hello, world!   ║",
    "║ from Thread #     ║",
    "╚═══════════════════╝",
    "\0"
    };

char *hello_str43[LINES43+1] =
    {
    "╔═══════════════════╗",
    "║                   ║",
    "║   Hello, world!   ║",
    "║ from Thread #     ║",
    "║                   ║",
    "╚═══════════════════╝",
    "\0"
    };

char *hello_str50[LINES50+1] =
    {
    "╔═══════════════════╗",
    "║                   ║",
    "║   Hello, world!   ║",
    "║                   ║",
    "║ from Thread #     ║",
    "║                   ║",
    "╚═══════════════════╝",
    "\0"
    };


char **helloptr;
int numlines;

typedef struct _frame   /* frame structure */
    {
    unsigned    frame_cleared;
    unsigned    row;
    unsigned    col;
    unsigned    threadid;
    long        startsem;
    char        threadstack[THREADSTACK];
    } FRAME;

FRAME far *frames[MAXFRAMES];      /* pointers to frames */
unsigned maxframes;

ULONG sleeptime = 1L;              /* minim sleep time */
char keythreadstack[THREADSTACK];


    /* function prototypes */

void hello_thread(FRAME far *frameptr);
void keyboard_thread(void);
void main(int argc, char **argv);


void main(int argc, char **argv)
    {
    int row, col, maxrows, maxcols, len, i, loops = 0;
    VIOMODEINFO viomodeinfo;

    if(argc > 1)
        sleeptime = atol(argv[1]);
    if(sleeptime < 1L)
        sleeptime = 1L;
```

CONTINUED

Figure 14 CONTINUED

```
/* start keyboard thread */
if(_beginthread(keyboard_thread,keythreadstack,
                THREADSTACK,NULL) == -1)
    exit(-1);

viomodeinfo.cb = sizeof(viomodeinfo);
VioGetMode(&viomodeinfo,0);            /* get video info */

maxrows = viomodeinfo.row;
maxcols = viomodeinfo.col;

switch(maxrows)
    {
    case 25:
        helloptr = hello_str25;
        numlines = LINES25;
        break;
    case 43:
        helloptr = hello_str43;
        numlines = LINES43;
        break;
    case 50:
        helloptr = hello_str50;
        numlines = LINES50;
        break;
    default:                   /* fail if not 25,43,50 lines */
        assert(0);
        exit(-1);
    }

len = strlen(*helloptr);

maxframes = (maxrows / numlines) * (maxcols / len);

assert(maxframes <= MAXFRAMES);

for( i = 0; i < maxframes; i++)   /* initialize structures */
    {
    if(!(frames[i] = malloc(sizeof(FRAME))))
        exit(0);
    frames[i]->frame_cleared = FALSE;
    frames[i]->startsem = 0L;
    memset(frames[i]->threadstack,0xff,
           sizeof(frames[i]->threadstack));
    }

i = RAND();      /* get first random frame */


/* set up random appearance */

for(row = col = 0; loops < maxframes ; )     /* set row/col
                                                each frame */
    {
    if(!frames[i]->frame_cleared)
        {
        frames[i]->frame_cleared = TRUE; /* set for empty
                                            frame */
        frames[i]->row = row;          /* frame upper row */
        frames[i]->col = col;          /* frame left column */

        col += len;                    /* next column on
                                          this row */
        if(col >= maxcols)             /* go to next row? */
            {
            col = 0;                   /* reset for start
                                          column */
            row += numlines;           /* set for next row */
            }

        i = RAND();                    /* get next random
                                          frame */
        }
    else
        ++i;
```

CONTINUED

counter variable) and returns to the top of the loop. Note that the call to DosSleep suspends the current thread, allowing OS/2 to give CPU time to threads of the same or greater priority. Without the call to DosSleep, the thread would attempt to run in a continual loop, unnecessarily burning CPU time. Note, too, that the keyboard thread does not require DosSleep: the IO_WAIT parameter to KbdCharIn blocks that thread until there is keyboard input available.

The keyboard thread code also uses the CountSem semaphore to gain access to the counter variable. Every time the user presses the Esc key, the keyboard thread requests access to the semaphore, blocking until the semaphore has been cleared. Then it increments the counter variable and clears the semaphore. Again, this mechanism prevents it from accessing the counter variable at the same time as the main thread. Thus, access to the counter variable has been serialized and the activity of the two threads has been synchronized. From this simple example, we can now move to something a little more complex: a multithreaded version of HELLO.C.

A Multithreaded HELLO.C

The multithreaded version of HELLO.C is designed to help illustrate the use of threads and semaphores for serializing access to and controlling resources. Whereas the original HELLO.C simply printed a message on the screen and terminated, HELLO0.C (shown in the listing in **Figure 14**), logically subdivides the screen into frames and prints the message in each frame. The program assigns each frame a thread that is responsible for writing and clearing the message. The program's threads continue to

write and clear their messages until the user presses the Esc key. This frame-based format will be the basis for other example programs as we explore the OS/2 subsystems and other facilities in later articles in this series.

Each frame's thread receives a pointer to the frame's data structure, which contains the information the thread will use during the course of the program. This structure includes the frame's row/column coordinates, the thread's ID (returned from _beginthread), a semaphore that the main program thread will use to activate the thread, and the thread's stack. Exactly how many frames will appear on the screen depends on the screen mode when you run the program (25, 43, or 50 lines).

HELLO0.C can take one optional command-line argument: the number of milliseconds that the main thread should sleep between each activation of a frame thread. This argument allows you to slow down the visual activity deliberately so that you can see what's happening a little more clearly. The command-line parameter, stored in a variable called sleeptime, defaults to 1 millisecond, which the DosSleep kernel function will round up to 32 milliseconds—the minimum OS/2 time slice. It also prevents HELLO0.C from hogging too much CPU time. You might try running the program with a parameter of 50, 100, 500, or 1000 (the equivalent of 1 second) to get a better idea of what's happening.

The program begins by checking the command-line parameter and then calls _beginthread to spin off a keyboard thread, which blocks until keyboard input is received and terminates the program when the user presses the Esc key. The code for this thread is lifted

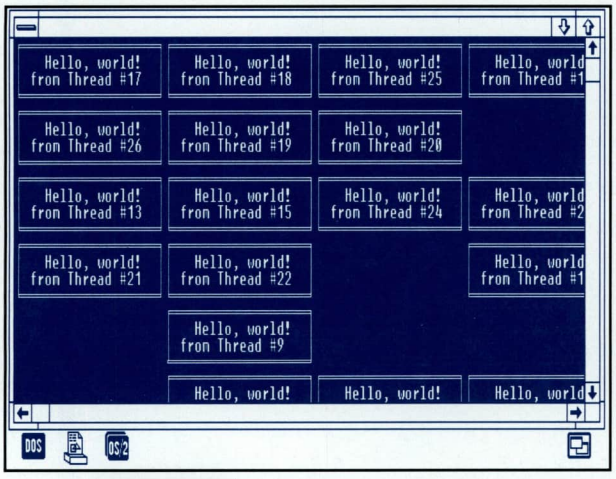
Figure 14 CONTINUED

```
        if(i >= maxframes)
            {
            i -= maxframes;
            loops++;                            /* keep track of #
                                                   of frames*/
            }
        }


    for( i = 0 ; i < maxframes; i++)            /* start a thread
                                                   for each */
        {
        DosSemSet(&frames[i]->startsem);        /* initially set
                                                   each sem. */


        /* start each thread */

        if((frames[i]->threadid = _beginthread(
                (void far *)hello_thread,
                (void far *)frames[i]->threadstack,
                THREADSTACK,
                (void far *)frames[i])) == -1)
            {
            maxframes = i;          /* reset maxframes on failure */
            break;
            }
        }

    while(TRUE)                     /* main loop */
        {


        /* swing thru frames, signalling to threads */

        for( i = 0; i < maxframes; i++)
            {
            DosSemClear(&frames[i]->startsem);    /* clear: thread
                                                     can go */
            DosSleep(sleeptime);                /* sleep a little */
            }
        }
    }


void hello_thread(FRAME far *frameptr)      /* frame thread
                                               function */

    {
    register char **p;
    register int line;
    int len = strlen(*helloptr);
    unsigned row, col = frameptr->col;
    char idstr[20];

    while(TRUE)
        {
        DosSemRequest(&frameptr->startsem,-1L); /* block until
                                                   cleared */
        itoa(frameptr->threadid,idstr,10);      /* init idstr */

        row = frameptr->row;            /* reset row */

        if(!frameptr->frame_cleared) /* if frame in use, erase */
            for( line = 0; line < numlines; line++, row++)
                VioWrtCharStr(blank_str,len,row,col,0);
        else                        /* else frame not in use */
            {
            p = helloptr;           /* print message */
            for( line = 0; **p; line++, row++, p++)
                VioWrtCharStr(*p,len,row,col,0);


            /* write id # in frame */

                VioWrtCharStr(idstr,3,
```

CONTINUED

Figure 14 CONTINUED

```
                                row-(numlines/2),
                                IDCOL+col,0);
            }


        /* toggle use flag */

        frameptr->frame_cleared = !frameptr->frame_cleared;
        }
    }

void keyboard_thread(void)
    {
    KBDKEYINFO keyinfo;

    while(TRUE)
        {
        KbdCharIn(&keyinfo,IO_WAIT,0); /* wait for keystroke */
        if(keyinfo.chChar == ESC)      /* break if ESC pressed */
            break;
        }
    DosExit(EXIT_PROCESS,0);           /* terminate process */
    }

/* end of hello0.c */
```



▲ **Figure 15** **Each frame of Hello, shown here running in a Presentation Manager text window, is created by an additional thread and is controlled by OS/2 RAM semaphores.**

directly from the earlier keyboard thread example, shown in **Figure 8**. Next the main thread goes through several housekeeping and preparatory steps: it obtains the video mode through a call to VioGetMode, sets up for the number of lines on the screen, and calculates the maximum number of frames to display. Then it allocates and initializes the frame structures (FRAME data types) and randomly chooses a FRAME for each screen frame, assigning the appropriate row/column coordinates. Consequently, the frames will seemingly appear and disappear in a random order, although the order remains static throughout the program.

The real fun in HELLO0.C begins when the main thread calls DosSemSet for each FRAME, followed by a call to _beginthread to start the FRAME's thread (which will block until the semaphore clears). Finally, the main thread enters a loop where it stays for the remainder of the program. In this loop, it activates the thread for each FRAME by clearing the FRAME semaphore. The call to DosSleep suspends the thread, forcing it to give up some CPU time before activating the next thread. The main thread will do this for every FRAME and then repeat the process. Adding calls to DosSleep when a thread is in a loop will make the program more efficient, since it eliminates a thread's ability to waste CPU time.

What does each FRAME thread do? The code for each FRAME thread is contained in the hello_thread function, which takes one parameter: the address of the thread's FRAME, which is passed to it by means of _beginthread. The FRAME thread code is structured around a loop, at the top of which is a call to the DosSemRequest kernel function. By passing the address of a semaphore and a –1 to this function, the calling thread blocks until the semaphore clears. Thus, each thread is idle until the main thread clears its semaphore.

Once activated, a FRAME thread will either clear or write its message, depending on a variable that is toggled every time the thread is active. Note that the VIO subsystem function, VioWrtCharStr, performs all video output and writes a specific number of characters from a string at a specific row/column coordinate on screen. An example of a FRAME thread's output is shown in **Figure 15**.

As mentioned earlier, the program will continue until the user presses the Esc key. At that time, the keyboard thread will wake up (it's been blocked in the absence of keyboard input) and terminate the program.

HELLO0.C is probably multithreaded overkill (how many times will you need to run as many as 25 threads in an application?), but it should get you off to a strong start and clarify the multithread issues addressed in the article. With this foundation, you'll be able to write some rather complex programs that use multiple threads. Indeed, program development will become even more interesting in the next issue, when we explore the VIO subsystem. ❑

MSJ Source Code Listings

All our source code listings can be found on Microsoft OnLine, CompuServe®, and two public access bulletin boards. On the East Coast, users can call (212) 889-6438 to join the RamNet bulletin board. On the West Coast, call (415) 284-9151 for the ComOne bulletin board. In either case, look for the MSJ directory. Communications parameters for public access bulletin boards: 1200 Baud (RamNet also 2400), word length 8, 1 stop bit, full duplex, no parity.

Microsoft®

000-000-225