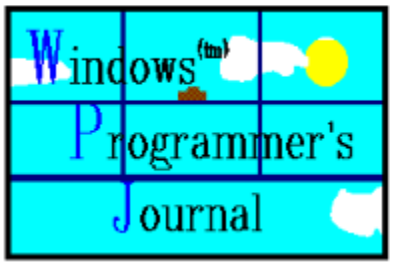


The Windows Programmer's Journal
Copyright 1993 by Peter J. Davis
and Mike Wallace

Volume 01
Number 08
September 93

A monthly forum for novice-advanced programmers to share ideas and concepts about programming in the Windows (tm) environment. Each issue is uploaded to the info systems listed below on the first of the month, but made available at the convenience of the sysops, so allow for a couple of days.

You can get in touch with the editors via Internet or Bitnet at:

Intenet: 71644.3570@compuserve.com or 71141.2071@compuserve.com

CompuServe: 71644,3570 (Pete) or 71141,2071 (Mike)

GEnie: P.DAVIS5

America Online: PeteDavis

Delphi: PeteDavis

or you can send paper mail to:
Windows Programmer's Journal
9436 Mirror Pond Dr.
Fairfax, Va. 22032

We can also be reached by phone at: (703) 503-3165.
The WPJ BBS can be reached at: (703) 503-3021.

The WPJ BBS is currently 2400 Baud (8N1). We'll be going to 14,400 in the near future, we hope.

- WPJ is available from the WINSDK WINADV, MSWIN32, BPASCAL and BCPPWIN forums on CompuServe, and the IBMPC, WINDOWS and BORLAND forums on GEnie. It is also available on America Online in the Programming library. On Internet, it's available on WSMR-SIMTEL20.ARMY.MIL and FTP.CICA.INDIANA.EDU. We upload it by the 1st of each month and it is usually available by the 3rd or 4th, depending on when the sysops receive it.

LEGAL STUFF

distributed in a common and easy to share form.

- No part of the Windows Programmer's Journal may be re-published or duplicated in part or whole, except in the complete and unmodified form of the Windows Programmer's Journal, without the express written permission of each individual author. The Windows Programmer's Journal may not be sold for profit without the express written permission of the Publishers, Peter Davis and Michael Wallace, and only then after they have obtained permission from the individual authors.

Table of Contents

Official Motto: Served only in the finest restaurants.

Windows Programmer's Journal Staff:

Publishers	**Pete and Mike**
Editor-in-Chief	**Pete Davis**
Managing Editor	**Mike Wallace**
Contributing Editor	**David Campbell**
Contributing Editor	**Dennis Chuah**
Graphic Artist (Bad)	**Pete Davis**
Graphic Artist (Good)	**Mark Coghlan**
Graphic Artist	**Dennis Chuah**
Graphic Artist	**Bernard Andrys**
Contributing Writer	Dennis Chuah
Contributing Writer	David S. Browne
Contributing Writer	Bernard Andrys
Contributing Writer	Tony Lee
Contributing Writer	Kurt Simmons

WPJ.INI

by Pete Davis

Well, the survey results continue to pour in. Keep them coming. If you haven't filled one out, there's another copy in this issue. The last day for the prize is September 30th. That means post-marked or the e-mail is dated by September 30th. We will do the drawing shortly after.

Speaking of the survey, I thought I'd give you all an idea of what the results have been like. A lot of you like the fact that we cover a wide variety of levels. We are commited to providing information for novice programmers as well as more advanced programmers.

We got a lot of good and bad feedback. In a sense, all feedback is good. When I say bad, I really mean critical. And that's a good thing. One thing that disturbed me is this. We got a few responses that mentioned errors, and one saying "Gross Errors", in the magazine. YES, there are "Gross Errors". To date, Mike and I roughly skim the articles. We check spelling and grammer. We don't do a whole lot more. Why? We don't have time. It takes at least 25 hours a month to put the magazine together as it is. I went through this last month, and some people don't seem to get it. That's not the only problem. As I said last month, we don't pay the writers and have a hard enough time getting them. If we made it difficult for them by making them do re-writes, we'd be putting out the Windows Programmer's Nothing. I'm serious about that. There are a couple people who might put up with it for a while, but if I wasn't getting paid, I sure wouldn't make a habit of it, and I don't blame our writers for having the same feeling about that. If you're going to put up with serious editing, you may as well get paid for it.

As far as "Gross Errors", Mike and I have always been willing to publish corrections, which we hoped we'd get more of, but people haven't sent them in. I got one in the reader's survey, and here it is:

In Bernard Andrys' article "GDI See GDI Do", he said that the PASCAL keyword in a function declaration meant that the parameters were passed by reference and not by value. (That was the gist that I got, anyway.) This is incorrect. What it actually has to do with is the way parameters are placed on the stack. The C calling convention places the arguments on the stack from right to left. For example, the call, myFunction(a, b, c) pushes the contents onto the stack starting with 'c' and ending with 'a'. The Pascal calling convention places the arguments on the stack from left to right, so with the previous example, 'a' is pushed onto the stack first, followed by 'b' and then 'c'. This gives C the ability to handle an unspecified number of parameters on the stack. It makes library calls like printf() possible. Although Pascal allows an unspecified number of parameters to be used for the Read, Write and some other procedures, this is handled by the compiler at compile time. In C, the parameters are handled at run-time by the function.

OK, that was easy and relatively painless. If anyone else has corrections, send them in. We'd be more than happy to tell everyone else. Our original intent was for this magazine to be a collective effort. With some of many of our readers and writers, it has been able to be that. It bothers me, though, when people say, "you should fix this, do that, fix that, and stop doing this..." and go on through a list 20 things and then say that they still want it for

free. Mike and I have done the best job we could given the restrictions on our time. This goes for the writers we've had and still have. They should all be applauded for their efforts in getting articles done and putting in the time.

I'm probably making a big deal out of a little thing, in fact, I am. Most of you had few or no complaints. I have complaints about the magazine, it's not perfect by any stretch, but most of you were very complimentary. We thank you all. Mike and I have enjoyed doing the magazine and we'll hopefully enjoy continuing it for years to come.

The survey, all-in-all was a great success. It has given us a better idea of the kinds of things we should focus on and the kinds of things we might want to consider leaving behind. Does that mean you should stop sending your comments, suggestions and critiques. No sir (ma'am)! Send them in. For those of you who have complaints, why did you wait until the survey to send them in? Send them in as soon as you have them. If there are problems with the magazine, we want to know about it. We want to do our best to please as many of you as possible.

Ok, so where does everything stand? At some point Mike and I would like to actually start printing the magazine. Many of you said you'd like us to keep the WinHelp format even if we go to print. We will do that. We don't know when we'll take the magazine to print and it may be some time. Mike and I are going to start trying to spend more time on the magazine if possible. We're going to try to improve the editing of the articles also. Again, all of this takes time. If the magazine were to go to print, it would become a full time job for Mike and I. The kind of "gross" mistakes seen in past issues would end (hopefully!).

We will keep everyone informed of any developments with the magazine. At this point we've talked with people involved in publishing about the possibility of printing the magazine. How soon that happens remains to be seen.

Peace.

Pete

Software Development '93

by Pete Davis

Mike and I went to the Software Development '93 conference in Boston. The conference ran the week of the August 23-27. Mike and I were there from the 24th to the 26th. This main point of the conference was to let Software Developers strut their stuff. All the big boys were there, Microsoft, Borland, Miller-Freeman and others.

So, what were people showing off? Well, I think the single product that impressed Mike and I the most, and this is strictly from a demo point-of-view, but Symantec's C++ development environment won hands down for best demo. They also won in the best "Star Trek: The Next Generation" rip-off. Basically they did a big show as part of their demo. It was cute, but the demo was really impressive. The development is a lot like Visual C++, but better. You can create your dialog boxes, menus, and other resources from within Symantec's Integrated Development and Debugging Environment (IDDE). You tie the buttons to dialog boxes and dialog boxes to menu items, and so on and so on. All the code will be generated by the IDDE. You can even throw in things like tool-bars. Yep, that's right, just throw in the tool bar, make your own icons for it and bang, all the code you need is there. It also allows you to go in and modify the code and have new generations of the design take on your existing code changes. We're hoping to get a review copy of this soon. It's a great product and it deserves some good press. (That's a hint to any of you who happen to work for Symantec!!!).

Another product that really blew peoples socks off was Nu-Mega's Bounds Checker 2.0, which was officially released during the conference. This new version of Bounds Checker has terrific improvements on version 1. More checking is done and the best feature is the new event tracing feature. Essentially, it will trace all API and library calls made by your program and keep track of what parameters were passed to each. It also keeps track of messages and other things. Really, it's just a plain fantastic product. Look forward to a review in the next issue.

Microsoft is working on MFC 2.5. One of the main improvements is encapsulation of OLE 2.0. A lot of people have agreed that OLE 2.0 is just too big and bulky for most people. MFC 2.5 is going to fix that problem. What about us C programmers? Well, the best thing to do might be to do all the OLE 2.0 work in C++ and then do the rest of your coding in C. Really, doing OLE 2.0 in C is just about impossible. In fact, we went to a class taught by a Microsoft employee, Nigel Thompson. Nigel humorously described his 4.5 months learning to do simple OLE (as he says, Objectionable Learning Experience) 2.0 things in C, like containers. We're talking about a guy who has, as a resource, the people who wrote OLE 2.0. Obviously if it took him 4.5 months to do this, most of us would probably be better off doing it with MFC 2.5, which is supposed to cut that 4.5 month learning curve down to 30 minutes or so. A slight improvement!

Borland announced OWL 2.0 which is planned to be a cross-platform class library. We'll see how far that goes. Microsoft released Visual C++ for Win32. That was all pretty public, though.

We got to meet some of our readers who recognized the names on our badges. I have to say, I was awfully surprised to have people recognize me at the conference. Anyway, the entire conference seemed to be a big success and Mike and I certainly enjoyed meeting everyone there, especially the readers that made it.

We were able to get some review copies of software and books, so you'll be seeing some reviews of some really good products and books in the near future. Hopefully Mike and

I will have enough warning for the next conference that we can let you all know we'll be there. Hopefully we'll be able to run into some more of you at other conferences. It's nice to meet the readers and hear first-hand what they have to say about the magazine.

WPJ Survey

Down the page is a survey. Please fill it out as completely as possible. Don't forget to put your address so we can mail your prize if you win.

The prize options are:

1) Win32 Reference Manuals (5-book set)

2) "Windows API Bible", "Windows Internals" and "Undocumented Windows"

3) Windows NT Resource Kit

Either print it out, fill it in, and send it via regular mail or edit it with a text editor and fill in the blanks and send it by E-Mail.

On Compuserve : 71644,3570
On Internet : 71644.3570 at compuserve.com
On GEnie : P.DAVIS5
On America Online: PeteDavis
On Delphi : PeteDavis

or regular mail to:

WPJ Reader Survey
9436 Mirror Pond Dr.
Fairfax, Va 22032 U.S.A.

Thanks for taking the time to fill out the survey.

Pete & Mike

Click Here To Print

Windows Programmer's Journal
Reader Survey: August, 1993

Name:_____ E-Mail Addresses

Profession:_____ Service Address

Title:_____ _____ _____

Company:_____ _____ _____

Address:_____ _____ _____

_____

_____

Phone #:_____ Fax #:_____

Primary operating system on your computer:_____

Secondary operating system:_____

Primary programming language used (include brand):_____

Other Programming languages used:_____

Libraries or Class Libraries used:_____

Development tools used (debuggers, editors, etc. Include brand.):_____

_____

_____

What I Like about the Windows Programmer's Journal:_____

_____

What I don't like about Windows Programmer's Journal:_____

_____

Things I'd like to see in the Windows Programmer's Journal:_____

_____

Other programming magazines I read:_____

Would you pay for printed copies of WPJ? (Circle one) YES NO

Prize I want:_____

Beginner's Column

By Dave Campbell

Remember when you were in school, and the teachers would show you the REAL hard way to do something, and just about the time you thought you understood it enough, they would show you the easy way? Well, I'm going to do that to you right now. The File Open box we've worked on the last two issues is going to go away, and be replaced by a single call to the Common Dialogs DLL, COMMDLG.DLL.

Common Dialogs allow the developer to call up a functional dialog box with a single call, rather than build the dialog box, compile in the dialog resource, write the code for handling the dialog box messages, etc. At the same time, the developer gets dialog boxes for quite a few of the standard configuration items that look identical to the ones everyone else is using, thereby making life easier for the user.

COMMDLG.DLL

COMMDLG.DLL contains dialogs for the following:

ChooseColor
ChooseFont
FindText
GetFileTitle
GetOpenFileName
GetSaveFileName
PrintDlg
ReplaceText

These may be used in Windows 3.0 or 3.1, but only in enhanced mode.

My intention was to show the use of just the GetOpenFileName box, but it seemed so easy, I went ahead and added an implementation of the ChooseColor box and ChooseFont box because they always seems to dress up an application. Once you see how easy this is, filling out the remainder of the menu selections with standard dialog boxes will become quite a bit less of a "chore".

Removed Code

First I'll quickly run through what we can throw away from last issue:

1) All the Fileo.* files, including the DLL that we built last time

2) The DoFileOpenDlg procedure in Hello.c

Hello.C

The #include of "fileo.h" is no longer needed, because we aren't using that DLL. However, we do need to add

```
#include <commdlg.h>
```

to pick up the items defined there.

Since we aren't going to use Fileo.DLL, we do not need

```
HANDLE hFileo
```

but we will need

```
HANDLE hCommDlg
```

so we just change that line, and the lines where the DLL is loaded to load COMMDLG.DLL and get its handle into hCommDlg.

'pof' and 'of' are removed, as they were used with the old code.

szFileSpec and szDefExt are moved to WndProc, and we add:

```
COLORREF ColorRef;
HANDLE   MyFont = NULL;
```

COLORREF

COLORREF is nothing more than a DWORD. Defining it as such, however, gives us a way to remember we are using an RGB color value. The RGB method of defining colors is a standardized way that will carry us into the 24-bit arena of colors, but is still useful on systems with 16 colors. Programming in this manner means a little overhead up-front, but ease of programming, and portability later on. The ColorRef variable defined here will be used to set the color of our Hello World text via a common dialog call.

MyFont

MyFont is a handle to the font we are going to select in the ChooseFont dialog box. Initially it is set to NULL, and this fact will be used in the code.

WinMain

The only changes to WinMain are the loading and freeing of COMMDLG.DLL instead of Fileo.DLL, and the freeing of the handle, MyFont, if necessary:

```
if (MyFont != NULL)
  DeleteObject(MyFont);
```

WndProc

WndProc is the place where all the action happens, so most of the changes are here.

Variables

The following are defined for our new version:

```
OPENFILENAME      openfile;
char              szFilter[25];
char              szExt[4 + 1];
char              szDirName[128];
char              szFileSpec[16];
char              szDefExt[5];
char              szFileTitle[128];
int                     i;

COLORREF          LocColorRef[16];
CHOOSECOLOR ChooseMyColor;

DWORD             dwFlags=CF_SCREENFONTS;
CHOOSEFONT        ChooseMyFont;
LOGFONT                 LogMyFont;

HANDLE                  TempFont;
```

OPENFILENAME is a structure used by the file dialogs in the common dialog DLL. It is defined as:

```
typedef struct tagOPENFILENAME
   {
   DWORD     lStructSize;
   HWND      hwndOwner;
   HINSTANCE hInstance;
   LPCSTR    lpstrFilter;
   LPSTR     lpstrCustomFilter;
   DWORD     nMaxCustFilter;
   DWORD     nFilterIndex;
   LPSTR     lpstrFile;
   DWORD     nMaxFile;
   LPSTR     lpstrFileTitle;
   DWORD     nMaxFileTitle;
```

```
    LPCSTR    lpstrInitialDir;
    LPCSTR    lpstrTitle;
    DWORD     Flags;
    UINT      nFileOffset;
    UINT      nFileExtension;
    LPCSTR    lpstrDefExt;
    LPARAM    lCustData;
    UINT      (CALLBACK *lpfnHook) (HWND, UINT, WPARAM, LPARAM);
    LPCSTR    lpTemplateName;
    } OPENFILENAME;
```

We are only going to use the minimum in our example, but I will mention the unused ones, to pique your interest to experiment.

CHOOSECOLOR

CHOOSECOLOR is a structure used by the color dialogs in commdlg.DLL:

```
typedef struct tagCHOOSECOLOR
    {
    DWORD   lStructSize;
    HWND    hwndOwner;
    HWND    hInstance;
    COLORREF rgbResult;
    COLORREF FAR* lpCustColors;
    DWORD   Flags;
    LPARAM  lCustData;
    UINT    (CALLBACK* lpfnHook)(HWND, UINT, WPARAM, LPARAM);
    LPCSTR  lpTemplateName;
    } CHOOSECOLOR;
```

Again, before we are finished, all the entries will be explained.

CHOOSEFONT

CHOOSEFONT is a structure used by the dialogs in commdlg.DLL:

```
typedef struct tagCHOOSEFONT
     DWORD           lStructSize;
     HWND            hwndOwner;
     HDC             hDC;
     LOGFONT FAR*    lpLogFont;
     int             iPointSize;
     DWORD           Flags;
     COLORREF        rgbColors;
     LPARAM          lCustData;
     UINT (CALLBACK* lpfnHook)(HWND, UINT, WPARAM, LPARAM);
     LPCSTR          lpTemplateName;
     HINSTANCE       hInstance;
     LPSTR           lpszStyle;
     UINT            nFontType;
     int             nSizeMin;
     int             nSizeMax;
```

```
} CHOOSEFONT;
```

LOGFONT

LOGFONT is a structure used by the dialogs in commdlg.DLL defining the characteristics of a font for drawing text on a window:

```
typedef struct tagLOGFONT
    int   lfHeight;
    int   lfWidth;
    int   lfEscapement;
    int   lfOrientation;
    int   lfWeight;
    BYTE  lfItalic;
    BYTE  lfUnderline;
    BYTE  lfStrikeOut;
    BYTE  lfCharSet;
    BYTE  lfOutPrecision;
    BYTE  lfClipPrecision;
    BYTE  lfQuality;
    BYTE  lfPitchAndFamily;
    BYTE  lfFaceName[LF_FACESIZE];
} LOGFONT;
```

The first change to WndProc is in the WM_CREATE message handler. Since we are gong to be changing colors, we assign ColorRef to be black:

```
ColorRef = RGB(256, 256, 256);
```

and continue on. Then in the WM_PAINT handler, we use ColorRef in the SetTextColor call:

```
SetTextColor(hdc, ColorRef);
```

Now when we change ColorRef, and force a WM_PAINT message, the text will be painted in a different color of our choice. Another change made to the WM_PAINT message handler is to check for a font change:

```
if (MyFont != NULL)
   TempFont = SelectObject(hdc, MyFont);

DrawText(...);

if (MyFont != NULL)
   SelectObject(hdc, TempFont);
```

If MyFont is set to a handle value, that font is selected into use by doing a SelectObject call. The return value is the previous handle, which we save in 'TempFont'. After displaying the text, we put the old font back into use by doing another SelectObject call. This will keep other display areas in our hdc from using the same font. In this particular application, it may not be necessary, but good habits are hard to break.

Files

The IDM_FILEOPEN message handler has changed quite a bit. We've removed the call to our DLL, and added the minimum that it would take to talk to the common dialog box GetOpenFileName procedure.

We first use the 'memset' command to set the structure openfile to all nulls, and insert the structure size into the structure at lStructSize:

```
   case IDM_FILEOPEN :
      memset(&openfile, 0, sizeof(OPENFILENAME));
      openfile.lStructSize = sizeof(OPENFILENAME);
```

We then declare the handle of the owner of this call as hWnd. This handle is used in the HELP call from the dialog box, to route the help message back to us:

```
      openfile.hwndOwner = hWnd;
```

In the standard file dialog boxes, there is always a combo box labeled "List Files of Type", and the list box contains entries such as "Text files (*.txt)" or "All files (*.*)". The developer defines those entries via the szFilter and szCustomFilter strings.

Each of the filter strings is a concatenation of string pairs each null-terminated, and the two filters are terminated by a double null, one for the final filter declaration, and one for the filter itself. This is reasonably confusing, but maybe a declaration will clear it up. If we want to have a filter of "Text files" be tied to "*.txt", we could fill our filter string as follows:

```
      sprintf(filter, "%s%c%s%c%c", "Text files", '\0', "*.txt", '\0', '\0');
```

An alternate method is useable at run-time, or if you define strings in a string table. Substitute an unused character for your string separator, and then at run-time, change all those to '\0':

```
      strcpy(szFilter, "Text Files|*.txt|");

      for (i = 0; szFilter[i] != '\0'; i++)
            {
            if (szFilter[i] == '|')
                  szFilter[i] = '\0';
      }
      openfile.lpstrFilter = (LPSTR)szFilter;
      openfile.nFilterIndex = 1;
```

Note the FilterIndex begins with "1", not 0. A value of 0 instructs Windows to use the CustomFilter string instead.

lpstrFile is a pointer to our returned file name, which we initialize to null:

```
      szFileName[0] = '\0';
      openfile.lpstrFile= (LPSTR)szFileName;
      openfile.nMaxFile = sizeof(szFileName);
```

If we insert a file spec in here, that is the name to which the box is initialized. This string must be at least 256 characters to avoid problems with Windows.

The FLAGS entry declares various parameters controlling the operation of the dialog

box. The parameters we are using are:

```
OFN_SHOWHELP | OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST
```

OFN_SHOWHELP instructs the commdlg code to display a help button in the dialog. As explained above, this will assert the help for our window, therefore the hwndOwner handle must not be NULL.

OFN_PATHMUSTEXIST instructs the commdlg code to only allow the user to type a path specification that exists currently on the system. An error is displayed if the path is non-existent.

OFN_FILEMUSTEXIST is similar to OFN_PATHMUSTEXIST in that only files that currently exist on the currently enabled path are allowed to be typed. Selecting OFN_FILEMUSTEXIST automatically selects OFN_PATHMUSTEXIST, but it is a helpful reminder to list both.

There are other parameters that may be defined for selecting multiple files, kicking off a new file message, enabling a hook function (explained below) or using a custom dialog box template, among others.

Next comes the call that does all the work:

```
GetOpenFileName(&openfile);
```

Normally, the result of this would be checked for errors, but in this application, we aren't searching for a file, or intending to open one. We are simply demonstrating the use of the dialog.

Staying standard with previous articles, we copy the returned file name to OutMsg, and finish this message handler:

```
lstrcpy(OutMsg, openfile.lpstrFile);
break;
```

Other parameters

The unused parameters of the OPENFILENAME structure are:

lpTemplateName declares a custom dialog template to be used in place of the common one. To do this, hInstance must be set to the instance value of the window owning the template, and the FLAGS value must enable templates.

nMaxCustFilter contains the size of the custom filter, if one is used.

lpstrFileTitle is a string that will receive the file name and extension only, that is the path specification will be stripped.

nMaxFileTitle is the length of lpstrFileTitle.

lpstrInitialDir is the starting directory of the dialog box. If this is NULL, the current directory is used. If, however, lpstrFile contains the full path specification and file specification for a file, the path portion of lpstrFile is used as the initial directory.

lpstrTitle is a user-definable title for the dialog box. A NULL will cause a standard default

value to be used.

nFileOffset is a returned value denoting where in the lpstrFile string the filename begins.

nFileExtension is a returned value denoting where in the lpstrFile string the file extension begins.

lpstrDefExt is a user-definable default extension to be used in the search if the user does not type an extension on a filename.

lpfnHook is a pointer to a function that snags dialog box messages prior to the common dialog box handler, allowing the user to intercept, or pre-process calls.

lCustData defines the data that is passed to the hook function.

That should be enough variation to make anyone pleased with "common" dialogs. And that is only one of the set.

Color

To enable a developer to allow an end-user to define colors for text or dialogs, the ChooseColor common dialog was built. An extra entry in the menu structure of our code, under "SAMPLE" was declared as IDM_COLOR, and handled as follows:

```
case IDM_COLOR :
      memset(&ChooseMyColor, 0, sizeof(CHOOSECOLOR));
      ChooseMyColor.lStructSize=sizeof(CHOOSECOLOR);
```

As with the File dialog, the CHOOSECOLOR structure must be declared, and its size passed to the dialog.

The initial 16 custom colors are declared going into the dialog, and coming out of the dialog via lpCustColors:

```
ChooseMyColor.lpCustColors=LocColorRef;
```

The handle of the window owning the dialog is declared as with the file dialog:

```
ChooseMyColor.hwndOwner = hWnd;
```

rgbResult is the value that is initially displayed with the dialog is opened, and the value that is returned by the user:

```
ChooseMyColor.rgbResult = RGB(256, 256, 256);
```

As with the file dialogs, flags may enable hooks, templates, help buttons, and control the full usage of the dialog. In our example, we have chosen to use the rbgResult value as the initially selected value.

```
ChooseMyColor.Flags = CC_RGBINIT;
```

The action happens with "ChooseColor". The dialog box is displayed, and the user is allowed to choose colors as in the control panel of Windows. The result may be checked for an error value (other than 0), and the resulting color is returned via rgbResult.

```
if (ChooseColor(&ChooseMyColor))
      {
      ColorRef = ChooseMyColor.rgbResult;
      InvalidateRect(hWnd, NULL, FALSE);
      UpdateWindow(hWnd);
      }
break;
```

In our case, rbgResult is written to ColorRef, and used when a repaint takes place, allowing the user to pick the color with which Hello/Goodbye Windows is painted. A repaint is forced by Invalidating the entire window client area (the NULL parameter), and not painting the background (the FALSE parameter).

UpdateWindow sends a WM_PAINT message straight to the window, causing the InvalidateRect area to be repainted. This takes care of the change we have made with the ChooseColor call, without having to resize the window to see the change.

The structure CHOOSECOLOR contains lpfnHook, lCustData and lpTemplateName as does the FILEOPEN structure.

Fonts

I don't really want to get into a full discussion of fonts at this time, because it is extensive, but I do want to show how to use the ChooseFont dialog box. If some of the information is sketchy, accept it for now, and we'll dig into it later, or look it up in Petzold.

An extra entry in the menu structure of our code, under "SAMPLE" was declared as IDM_FONT, and handled as follows:

```
case IDM_COLOR :
      memset(&ChooseMyFont, 0, sizeof(CHOOSEFONT));
      ChooseMyFont.lStructSize = sizeof(CHOOSEFONT);
      ChooseMyFont.hwndOwner = hWnd;
```

As with the other dialogs, the CHOOSEFONT structure must be declared, its size passed to the dialog, and our window handle inserted into
the structure.

A variable of LOGFONT type is pointed to by the lpLogFont structure entry, and filled out as follows:

```
ChooseMyFont.lpLogFont = &LogMyFont;
lstrcpy(LogMyFont.lfFaceName, "Times New Roman");
LogMyFont.lfHeight = 10;
LogMyFont.lfItalic = FALSE;
LogMyFont.lfWeight = FW_BOLD;
LogMyFont.lfStrikeOut = FALSE;
LogMyFont.lfUnderline = FALSE;
```

This declares 'Times New Roman' in bold 10 point type.

As I said above, at this point in time, I am choosing to ignore the other structure variables in LOGFONT, and at least for our application, the default values of NULL (as

inserted with the 'memset') work fine.

Our ColorRef value is inserted into the ChooseMyFont structure because color may come into play in the ChooseFont dialog:

```
ChooseMyFont.rgbColors = ColorRef;
```

The flags selected in our example are

```
ChooseMyFont.Flags = CF_SCREENFONTS | CF_EFFECTS
      | CF_INITTOLOGFONTSTRUCT;
```

CF_SCREENFONTS forces the dialog box to display only the fonts available currently for the screen.

CF_EFFECTS enables the strikeout, underline, and color effects of the dialog box.

CF_INITTOLOGFONTSTRUCT inits the dialog box to the LOGFONT structure as declared above.

Now the call to ChooseFont is made, and if successful, the font handle 'MyFont' is set to the new font chosen in the dialog, and ColorRef is set likewise.

```
if (ChooseFont(&ChooseMyFont))
      {
      if (MyFont != NULL)
            DeleteObject(MyFont);

      MyFont = CreateFontIndirect(&LogMyFont);
      ColorRef = ChooseMyFont.rgbColors;

      InvalidateRect(hWnd, NULL, TRUE);
      UpdateWindow(hWnd);
      }
break;
```

Include file

IDM_COLOR and IDM_FONT must be added to hello.h to allow access to the new menu items:

```
#define IDM_COLOR          326
#define IDM_FONT           327
```

Resource file

The addition of the Color selection to the menu is defined in hello.rc:

```
menuitem separator
menuitem "&Color",      IDM_COLOR
menuitem "&Font",       IDM_FONT
```

EndDialog

I have included the Microsoft make file, MHello.mak for this issue. I received no help so far on the request from help wizards in the last issue, so the offer still stands.

For the next issue, I had intentions of starting to discuss Help files, but WPJ seems to have started on that already, so I will continue with my schedule, and discuss displaying a dialog box as a main window.

Feel free to contact me in any of the ways below. Thanks to those of you that have e-mailed me questions; keep them coming.

Dave Campbell
WynApse PO Box 86247 Phoenix, AZ 85080-6247 (602)863-0411
wynapse@indirect.com
CIS: 72251,445
Phoenix ACM BBS (602) 970-0474 - WynApse SoftWare forum

ClickBar consists of a Dynamic Dialog pair that allows C developers for Windows 3.0/3.1 to insert a "toolbar" into their application. Microsoft, Borland, etc. developers may display a toolbar or tool palette inside their application, according to a DLG script in their .RC or .DLG file.

Borland developers may install ClickBar as a custom control, providing three custom widgets added to the Resource Workshop palette. These are three different button profiles defining 69 custom button images total. The buttons may be placed and assigned attributes identically to the intrinsic Resource Workshop widgets.

Source is provided for a complete example of using the tools, including the the use of custom (owner-drawn) bitmaps for buttons.

Version 1.5 uses single-image bitmaps to reduce the DLL size, and includes source for a subclass example for inserting a toolbar.

Registration is $35 and includes a registration number and printed manual.

WynApse
PO Box 86247 (602) 863-0411
Phoenix, AZ 85050-6247 CIS: 72251,445

Shared Global Memory

By Dennis Chuah

In this article I will present a way to implement shareable global memory in Windows 3.1. It is also the aim of this article to demonstrate two different methods of detecting whether an application has been loaded or not. Although the code is written in C, aiming for the Borland C++ compiler, the binaries produced and the associated header file can be used by any other C compiler that supports programming in Windows. It is also possible to use the libraries with Turbo Pascal or Borland Pascal with the proper import definitions. I am not an expert in Pascal and I do not own a Pascal compiler. So I hope someone who is better versed in Pascal can write a unit that allows the libraries to be used in Pascal programs.

First, let me introduce a few concepts of memory allocation in Windows 3.1. Windows 3.1 only runs in protected mode and therefore most of the memory allocation practices that were once used in real mode are now obsolete. Prior to version 3.0[1], Windows runs in real mode. The API inherits a number of real mode features from previous version of Windows. In real mode, Windows needs to perform memory moves to accommodate a multitasking environment. This is achieved through handles. When a block of memory is allocated, a handle is assigned to it. The owner can request that the block of memory be locked in memory (meaning it will not be moved by Windows). The locking process also retrieves a pointer to the block of memory. When the owner no longer needs to access the block of memory, it can tell Windows to unlock it. Windows is then free to move the block of memory around. The next time the owner needs to access the block of memory, it locks it and retrieve another pointer. The value of this pointer is not necessary the same as the previous pointer it retrieved (Windows may have moved the block of memory elsewhere). This complicated process is necessary because in real mode, pointers point to physical addresses. The only way for Windows to inform an application that a block of memory it owns has been moved is by passing a new pointer to it. The locking process ensures that Windows does not move memory that is still being actively used.

In protected mode however, pointers point to logical address. Each pointer, instead of having a segment:offset address, has a selector value and an offset address. The selector points to an entry in a global table (The Global Descriptor Table -- GDT) maintained by Windows. When a memory access occurs, the CPU translates the logical address in a pointer to physical address using the GDT. This process is transparent to software. When Windows running in protected mode needs to move blocks of memory, it only needs to update the GDT. Locked memory can also be moved as the CPU translates memory access via the GDT transparently. Therefore the process of allocating a handle, locking it and unlocking it becomes obsolete. The owner of a block of memory only needs to maintain a pointer to it. The following table illustrates the differences in the process of allocating

memory under different modes.

	Real mode	Protected mode
Allocating Memory	handle = GlobalAlloc...	handle = GlobalAlloc... ptr = GlobalLock (handle);
Accessing memory	ptr = GlobalLock (handle); *ptr = ... GlobaalUnlock (handle);	*ptr = ...
Freeing memory	GlobalFree (handle);	handle = LOWORD (GlobalHandle (SELECTOROF (ptr))); GlobalUnlock (handle); GlobalFree (handle);

The extra work in maintaining handles in protected mode is a legacy inherited from real mode. It is there for backward compatibility. For a more detailed discussion of Windows memory allocation I suggest C. Petzold, Programming Windows -- 2nd Ed., (1990) Microsoft Press. For more information in the Intel 80x86 protected mode memory management, consult Intels, 80386 System Software Writers Guide, (1988) Intel, Intels, 80386 Programmers Reference Manual, (1988) Intel, and J Crawford & P. Gelsinger, Programming the 80386, (1987) SYBEX Inc.

Shared Memory

Normally global memory allocated to an application belongs to the application and no other application can access it. Should any other application happen to access it, a GP fault will occur. To accommodate DDE, Windows 3.0 enables blocks of global memory allocated with the GMEM_DDESHARE flag to be shared by all applications. According to the API documentation, memory allocated with this flag is automatically freed when the owner terminates.

Now, suppose we want to allocate a block of memory that can be shared between two or more applications. One application can allocate it with the GMEM_DDESHARE set and pass the handle/pointer to the others. This is fine, provided the owning application does not terminate before any of the others that make use of the block of global memory. In the user friendly environment of Windows, the programmer cannot guarantee the life of an application. In fact, the programmer **should not** write code that exhibits this behaviour. Imagine the frustration of the user of not being able to close an application if it was the first one in a group to be opened!

To get around this problem, we must allocate shareable global memory that can live longer than the application that allocated it. I wrote a .DLL and a background application that only allocates shareable global memory. It involves writing an application that always remain active, but is hidden from the user. All requests to allocate shareable memory is made through the .DLL and carried out by the hidden application. This way, every block of shareable memory that is allocated remains allocated until explicitly freed or when Windows is terminated. This means that an application can request the .DLL to allocate a block of memory on its behalf, pass the pointer to other applications and terminate. When the block of memory is not required anymore, it can be freed.

More specifically, a .DLL when called for the first time can allocate shareable memory and store the pointer away. When it is called again (probably from another application), it can reuse the block of memory, without causing a GP fault.

The hidden application

Included is a file named SHAREMEM.ZIP. The source code of the hidden application (_MALLOC.EXE) is found in the root directory. Unzip this file with the -d flag to preserve its directory structure. The binaries, ie. the .DLL and the .EXE must be placed in the Windows directory, the System directory or any other directories pointed to by the PATH environment variable.

There are two sections in _MALLOC.EXE. The first initialises and registers the hidden window class. It also checks if there is another instance of itself in memory. It will not load if it found one. The second section contains a hidden window (a message target) that accepts requests to allocate shareable memory.

_MALLOC.EXE check whether there is another instance of itself via the call to RegisterClass. RegisterClass fails if an application registers more than once across instances. This is a simple way to check if another instance exist. The hidden window is registered as a global class with the CS_GLOBALCALSS style. When RegisterClass attempts to register a second class instance of a global class, it will fail. This prevents the user from accidently running a second instance of MALLOC.EXE.

The .DLL

The .DLL (MALLOC.DLL) interfaces MALLOC.EXE to applications that make use of it. It is located in the MALLOC directory of SHAREMEM.ZIP. Like _MALLOC.EXE, _MALLOC.DLL contains two sections. The first section performs initialisations and checks if _MALLOC.EXE has been loaded. If it not, it will be loaded. If MALLOC.DLL cannot load MALLOC.EXE, the initialisation will fail. The second section contains the shareable memory allocation API.

MALLOC.DLL determines whether _MALLOC.EXE is loaded by first registering a pivate Windows message. It then creates a hidden test window and broadcasts the message to all top level windows. The wParam of the message contains the handle of the test window.

```
HWND hWnd;
HINSTANCE hInst;
#define MAL_ACK 1
#define TEST_ID 0x41A5
  .
  .
  .
static void _gethwnd (void)
  {HWND hwndTest;

   hwndTest = CreateWindow (TEST_CLASS, "", WS_OVERLAPPED,
      0, 0, 0, 0, NULL, NULL, hInst, NULL);
   if (hwndTest == NULL) return;
   ShowWindow (hwndTest, SW_HIDE);
   SendMessage (HWND_BROADCAST, uTestMsg, (WPARAM) hwndTest,
      MAKELPARAM (0, TEST_ID));
   DestroyWindow (hwndTest);

   if (hWnd == NULL)
     {WinExec ("_MALLOC.EXE", SW_HIDE);
      hwndTest = CreateWindow (TEST_CLASS, "", WS_OVERLAPPED,
```

```
            0, 0, 0, 0, NULL, NULL, hInst, NULL);
         if (hwndTest == NULL) return;
         ShowWindow (hwndTest, SW_HIDE);
         SendMessage (HWND_BROADCAST, uTestMsg, (WPARAM) hwndTest,
            MAKELPARAM (0, TEST_ID));
         DestroyWindow (hwndTest);
        } // endif
     } // end gethwnd
```

The _gethwnd function first tests to see if _MALLOC.EXE is loaded. If not, it will attempt to load it. It then tests again to see if _MALLOC.EXE is loaded. If the second check fails, it assumes that it has failed to load _MALLOC.EXE and the initialisation fails. Every time the .DLL is called it checks whether the initialisation was successful. If it was not, the .DLL will return with error.

_MALLOC.EXE also registers the same Windows message. Its message procedure contains the following code extract.

```
UINT uRegMessage;
#define MAL_ACK 1
   .
   .
   .
if (msg == uRegMessage)
   {return SendMessage ((HWND) wParam, WM_COMMAND, HIWORD (lParam),
       MAKELPARAM (hWnd, MAL_ACK));
   } // endif
```

Where uRegMessage is the value returned by the RegisterWindowsMessage function.

When it receives the registered message, it sends a WM_COMMAND message to the sender with its hWnd in the loword of lParam. The following code extract is from the hidden test window of MALLOC.DLL:

```
LRESULT CALLBACK TestProc (HWND hWndTest, UINT msg, WPARAM wParam,
                           LPARAM lParam)
   {if (msg == WM_COMMAND)
      {if (wParam == TEST_ID && HIWORD (lParam) == MAL_ACK)
         {hWnd = (HWND) LOWORD (lParam);
         } // endif
       return 0;
      } // endif
    return DefWindowProc (hWndTest, msg, wParam, lParam);
   } // end TestProc
```

The test window then saves the hWnd of _MALLOC.EXEs hidden window to the variable hWnd. As hWnd is initialised to NULL every time the .DLL is loaded, a NULL value indicates that the test window did not receive the WM_COMMAND message. This implies that _MALLOC.EXE has not been loaded.

The second section of MALLOC.DLL contains these four API function calls.

API function name	Description/function
gmalloc	Allocates a block of shareable global memory.
grealloc	Changes the size of an already allocated block of global memory.

gfree	Frees a block of shareable global memory.
gisptr	Checks if a given pointer points to a block of shareable global memory.

The gmalloc function first checks if _MALLOC.EXE is loaded by calling _gethwnd. It fails if _MALLOC.EXE cannot be loaded. It then sends the MAL_ALLOC message to the hidden window of _MALLOC.EXE. The return value of the message is a pointer to the newly allocated shareable global memory. If there was an error in the process, the return value will be NULL. The following code extract is from _MALLOC.EXE.

```
void far *lpPtr;
HGLOBAL handle;
  .
  .
  .
case MAL_ALLOC:
   handle = GlobalAlloc (GMEM_DDESHARE | GHND, (DWORD) lParam);
   if (handle == NULL) return NULL;
   lpPtr = GlobalLock (handle);
   return (LRESULT) lpPtr;
```

This code simply allocates shareable global memory, locks it and returns a pointer to it.

The grealloc function checks if initialisation has been successful. It then sends the MAL_REALLOC message to the hidden window of _MALLOC.EXE. It fills the REALLOCSTRUCT structure and passes a pointer in the lParam of the message. The process is similar to gmallocs. When _MALLOC.EXE receives the MAL_REALLOC message, the following code extract processes it:

```
typedef struct tagREALLOCSTRUCT
  {DWORD dwSize;
   void far *lpPtr;
  } REALLOCSTRUCT, far *LPREALLOCSTRUCT;

LPREALLOCSTRUCT lpRa;
  .
  .
  .
case MAL_REALLOC:
   lpRa = (LPREALLOCSTRUCT) lParam;
   handle = (HANDLE) LOWORD (GlobalHandle (SELECTOROF (lpRa->lpPtr)));
   if (handle == NULL) return NULL;
   if (GlobalUnlock (handle)) return NULL;
   handle = GlobalReAlloc (handle, lpRa->dwSize, GMEM_ZEROINIT);
   if (handle == NULL) return NULL;
   lpPtr = GlobalLock (handle);
   return (LRESULT) lpPtr;
```

The handle to the allocated memory block is first retrieved via a call to GlobalHandle. The memory block is then unlocked. GlobalUnlock returns the number of times a handle has been locked. If the return value is greater than zero (meaning the memory is still locked), this message returns NULL. Otherwise, a call to GlobalRealloc is then issued to reallocate the memory. The block of memory is then locked and the pointer returned.

The gfree function sends the MAL_FREE to the hidden window of _MALLOC.EXE. The process is similar to greallocs except that the block of memory is freed instead of reallocated.

The following code extract shows how gisptr determines whether a pointer points to shareable global memory:

```
BOOL WINAPI gisptr (void far *lpPtr)
  {HGLOBAL handle;

   handle = (HGLOBAL) LOWORD (GlobalHandle ((UINT) FP_SEG (lpPtr)));
   if (handle == NULL) return FALSE;
   if (FP_OFF (lpPtr) != NULL) return FALSE;
   return TRUE;
  } // end gisptr
```

First, it tries to retrieve the handle to the block of memory. If that fails the pointer is not valid. It then checks if the offset of the pointer is 0 (NULL). Windows 3.x GlobalLock function always return pointers with 0 offset. This is not documented so it may change. The aim of this function is to provide an alternative to Windows 3.x IsBadHugeReadPtr and IsBadHugeWritePtr API functions.

My methods:

You may argue that you have a better method of determining whether an application is loaded (i.e., Using FindWindow or GetModule). Well, you may be right. I justify my methods because;

1. they are different,
2. they serve to illustrate the fact that there are more ways than one to achieve something in software, and
3. they are meant to demonstrate alternative (and interesting?) ways.

_MALLOC.EXE creates a hidden window, which only purpose is to receive messages. I call this kind of windows Message Targets. They serve only as a method of communications. Windows does not provide an object that an application can use to receive messages. The only way is to use a window that is always hidden. _MALLOC.EXE has no other window. It is a hidden application, meaning the user will never see it on the screen. In a way, it is like a .DLL. It provides services that other application can use. You might ask why dont I use a .DLL instead. The problem is any shareable global memory allocated by a .DLL is owned by the application that requested it. Our aim is to allocate global memory that has a longer lifespan than the application. To achieve that, the global memory must be allocated by an application that is always active. As this contradicts with the notion that the user **should have** the choice of terminating applications, _MALLOC.EXE is made to be hidden.

Notes:

[1] - Although prior to WIndows 3.0, there were Windows 286 and Windows 386, these were relatively unpopular and hence this discussion excludes them.

Other compilers/languages:

The source code provided will compile with Borland C++ version 3.1. To use the binaries with other C compilers, use the included wmalloc.h (This is located in the INCLUDE directory of SHAREMEM.ZIP) header file, the associated binaries (_MALLOC.EXE and MALLOC.DLL --These can be found in the BIN directory of SHAREMEM.ZIP.) and the import library, MALLOC.LIB (This is located in the LIB directory of SHAREMEM.ZIP). Windows.h must be included before wmalloc.h. Turbo Pascal users may want to create external referenced import functions to link up with MALLOC.DLL. Visual Basic and Word for Windows users can use the declare function/sub statements to link up with MALLOC.DLL.

Next article we will look at ways to use the shareable global memory blocks.

The author:

I am currently doing a PhD. degree in Electrical And Electronic Engineering at the University Of Canterbury. My programming experience dates back to the days where real programmers code with Z80 machine code. For the past two years I have taken an interest in Windows programming.

Please send any comments, questions, criticisms to me via
email: **chuah@elec.canterbury.ac.nz**
or post mail:**Dennis Chuah**
c/o The Electronic and Electrical Engineering Department
University of Canterbury
Private Bag
Christchurch
New Zealand.

All mail (including hate mail) is appreciated. I will try to answer all questions personally, or if the answer has general appeal, in the next issue. If you are sending me email, please make sure you provide me with an address that I can reach (most internet and bitnet nodes are reachable, and so is compuserve).

Windows Hooks

By David S. Browne

What is a Windows Hook ?

Ever want to get control just when Windows processes that keydown, or at that mouse move ? Well, Windows Hooks are for you ! A hook is really nothing more than Windows making a call to a DLL at various points in Kernel, User, GDI, etc. There are some gotchas, though, in using a hook and keeping Windows alive to process work, and that's what I hope to show here. There are three types of common hooks in Windows applications these days; 3.0 style hooks, 3.1 style hooks and Dynamic Intercept of Windows API functions for your own evil doings. The first two of these are somewhat documented in various Microsoft and OA ('Other Authors') books. The last is not documented by Microsoft at all, but has been shown in various trade publications. We will cover all three of these, in reverse order. We will present a real application usage of hooks when we look at the documented 3.0/3.1 hook functions. The application we will develop is a windows pop-up (I know, I'm still stuck in DOS !) that will allow you to activate it with any user defined hotkey and it will allow you to END or KILL any running window in your system. The problem with PROGMAN is that in many cases its END-TASK simply doesn't do the job. The target window must be reading its message queue to see the WM_QUIT message. The KILL function will call a nice new TOOLHELP debugging function to force kill any window, even if its not got its ear to the queue, so to speak.

Dynamic Intercept Of Windows API functions

Dynamic Intercept of Windows API functions is not the main basis of this article but I wanted to touch on the subject a bit before getting on with the real meat. When you issue a GetMessage or BeginPaint API call in your Windows program, it's nothing more than a call to a Windows system DLL to process this function. It is possible to intercept this to do any front end processing (or for that matter back end after the real API has finished playing) that you wish. Windows provides a function ("GetProcAddress") that will return the memory address of any function call you wish to look at. With this information you can store a jump at the DLL routine entry to your routine.

It's actually a bit more complicated than this as Windows has a habit of discarding code segments when it's short of memory and reloading fresh copies from disk as needed. So for this kind of dynamic intercept it's also necessary to use the TOOLHELP DLL to install a NotifyRegister function so that you see the segment reloads to re-apply your hook. If this is not done, then all will work fine until Windows decides that a bit more memory is in order, purges some code segments (that your hook is in !) then reloads a fresh copy and, Presto! Chango! NoHookO ! It's really not as complicated as it all sounds; a very good example of this was used in the May 1993 edition of Windows/DOS Developer's Journal. If you want to start intercepting Windows API functions then I recommend you read, then re-read, then sleep read the article ! Now to the beef !

Windows 3.0 and 3.1 Hooks

There are two API's to set a hook in Windows, the 3.0 SetWindowsHook and the 3.1 SetWindowsHookEx. The basic change is that the 3.0 SetWindowsHook is system wide, while the 3.1 version allows a hook to be JUST for a given instance or task. This allows you to develop that Keyboard Hook you always wanted, but were afraid of because of the

overhead of getting called for EVERY SYSTEM KEYBOARD EVENT. The example code we present here though uses a system-wide keyboard hook (in for a penny, in for a pound) so no matter where the current focus is we see the keydown and keyup. In addition to the hook function ID the main param you must provide on both of these calls is the address of the function that you are providing to process these hooks. This function must reside in a DLL. As with the 3.0 hook or a system wide 3.1 hook, it can be called in the instance of any task in the system. A DLL is required so as to be available to not just the API calling task.

There are twelve hooks you can set with these functions:

WH_CALLWNDPROC - WNDPROC Filter Hook for all SendMessage API Call's
WH_CBT - Computer Based Training Filter
WH_DEBUG - Called before any other hook
WH_GETMESSAGE - Called when an application issues a GetMessage API
WH_HARDWARE Called for any non-keyboard or non-mouse hardware message
WH_JOURNALPLAYBACK - Allows user to substitute mouse and keyboard input
WH_JOURNALRECORD - Allows user to save mouse and keyboard messages
WH_KEYBOARD - Called when any key is struck.
WH_MOUSE - Called for EVERY mouse movement in the system. (Did someone say Pentium ?)
WH_MSGFILTER - Called for user messages to any Dialog, Menu or Scroll Bar Window
WH_SHELL - Called whenever a new top level window is created or destroyed.
WH_SYSMSGFILTER - Called for system messages to any Dialog, Menu or Scroll Bar Window

Some documentation I have seen does not list the WH_SHELL hook as valid for SetWindowsHookEx. It is, it works, I've used it. Now for the actual API calls:

3.0

```
STATIC  HHOOK  hook1 = NULL;
                    .
                    .
                    .
hook1 =  SetWindowsHook(WH_KEYBOARD,  (HOOKPROC)KeyHook);
                    .
                    .
DWORD  CALLBACK  KeyHook(int iCode, WPARAM wParam, LPARAM lParam)
{
WhoKnowsOrCares();
?????????
}
```

3.1

```
STATIC HHOOK hook1 =  NULL;
                    .
                    .
                    .
hook1  =  SetWindowsHookEx(WH_KEYBOARD, (HOOKPROC)KeyHook,  hInstance,
NULL);
                    .
                    .
DWORD  CALLBACK  KeyHook(int iCode, WPARAM wParam, LPARAM lParam)
{
WeNowKnowAndCare();
CallNextHookEx(hook1, iCode, wParam, lParam);
```

```
return(0);
}
```

Simple right ? Really it is ! Honest ! Have I ever lied to you ?

Now to discuss an actual application usage for all this.

As I said, when you pop up Windows TASKMAN with the ever faithful Ctrl-Esc we have an END-TASK Button. Ever had a case where it didn't end that misbehaving task? Well that's because all it does is send a WM_CLOSE message to the application. If it is processing its messages normally it will see this message then terminate. I don't know about you but whenever I use that button the application is normally stalled! If I could end the application I would have done it by selecting close from the Application itself! OK, now we understand the problem. Solution time. We will implement a WH_KEYBOARD hook to scan for any ALT-S key sequence and if we see it pop up a window that will list all active top level windows on it and allow the user to END the task (same as the dreaded TASKMAN does!) or KILL the task by a call to the new 3.1 TOOLHELP DLL TerminateApp() function. The program is implemented in two separate modules WINSWAT, the main window procedure and SWATDLL the DLL with the keyboard hook function. If you look at WINSWAT it will look as any normal Windows C program does, it has a WinMain function that sets up a main window for the application and it has a WndProc for all the real processing. During the WM_CREATE processing, we make a call to InitialiseSwat to SWATDLL to setup the keyboard hook. The code in SWATDLL looks like this:

```
void WINAPI _export InitialiseSwat(BOOL fAction, HWND hwnd)
{
if (fAction)
      {
      OutputDebugString("SWATDLL: Pre--SetHook\n");
      hKhook = SetWindowsHookEx(WH_KEYBOARD, (HOOKPROC) KeyboardHook,
            hInstance, NULL);
      OutputDebugString("SWATDLL: Post-SetHook\n");
      hTASK = GetWindowTask(hwnd);
      hHWND = hwnd;
      }
else
      {
      UnhookWindowsHookEx(hKhook);
      hKhook  = NULL;
      hHWND = NULL;
      hTASK = NULL;
      }
}
```

This sets up a hook callback to KeyboardHook for keyboard events from any task in the system (The last NULL param says all tasks, but you could substitute a task value here if you ONLY wanted keyboard events for that task). The HWND param we save here is so that later in the actual keyboard hook code we can PostMessage to that window. The last part of the code is for termination to remove our hook. After this processing is complete, WINSWAT simply hides itself with a ShowWindow call and waits to be woken up with a message from SWATDLL.

If you look at the KeyboardHook code in SWATDLL you will see that we check for our key, the S key, then we make sure the ALT key is on. If all this is true we PostMessage a message to WINSWAT so that it will wake up. Take a look:

```
DWORD CALLBACK KeyboardHook(int nCode, WPARAM wParam, LPARAM lParam)
      {
      static BOOL fPost;
      BYTE iBit;
      WORD iWord;
      static UINT iQky = 83;

      if (nCode >= 0)
            if (wParam == iQky)
                  {
                  OutputDebugString("SWATDLL: Key Hook Hit\n");
                  iWord = HIWORD(lParam);
                  iBit  = HIBYTE(iWord);
                  if (iBit & 0x20)
                        {
                        fPost = PostMessage(hHWND, WM_COMMAND, WAKESIGL, 0);
                        return 1;
                        }
                  }

      CallNextHookEx(hKhook, nCode, wParam, lParam);
      return 0;
      }
```

When the WAKESIGL message is received in the WndProc WM_COMMAND processing section, it simply changes the Main Window to SHOW status, updates the listbox with all top level window names and then waits for a command. If a KILLBUTN is received, it is used the TerminateApp() function to really SWAT the thing out of Windows!

```
      hTask = GetWindowTask(WorkHwnd);
      TerminateApp(hTask, NO_UAE_BOX);
```

As you can see it's really not at all hard to use hooks. They can add functionality to any application when used in the correct manner. Don't substitute hooks, though, for things that could be easily done with normal Windows messages like WH_KEYDOWN, etc. Hooks can add overhead to your system, and that's something no one wants. Used correctly, however, they can add a multitude of features to your application.

All the source code, H files, Icons, Definition file etc. are included with this WPJ edition, take a look at it and get Hooking!

Customizing FileDialog in Visual C++

By Tony Lee

Overview

I have developed two application with Visual C++. Both of them are fairly good size applications with > 300K lines of C++ source code each. I use common dialogs extensively in my projects. In this article, I will try to show you how to use and customize the CFileDialog class in the Visual C++ environment. I would also like to share with you why I think C++ and object oriented development environment is such a great tool for developing Windows programs. Since I have limited time to write this article, I want to refer directly to the WinSDK help for most of the basic information related to Common Dialog. My background: I have been doing Windows programming for 4 months now. One of the applications I wrote is a diagnostic utility for a complicated instrument. The other application I wrote is a C/C++ cross reference utility that lets you cross reference C/C++ source code. As you can see, Windows programming with C++ is not hard at all.

What is Common Dialog?

Common dialog is a set of dialog box functions. These functions simplify the programming for the common dialogs such as File, Color, Font and Print Dialogs for Windows. You can find more information about them in the "Common Dialog Box OverView" section of the Windows SDK Help. [Editor's note: The common dialog routines are in the COMMDLG.DLL file - mfw]

Why C++?

One of the best reasons for using C++ in developing Windows programs is encapsulation. The class structure of the C++ langauge hides the most implicit details from you. For example, to implement the FileDialog using C and SDK, you have to write about 40 lines of C code. (You can count them yourself in the "Filename Dialog Boxes (3.1)" part of the Windows 3.1 SDK help.) You also need to understand the "OPENFILENAME" structure and the "GetOpenFileName" functions.

To implement the same open file dialog function in C++, you do the following:

```
      Void OnOpenFile() {
            CFileDialog dlg(TRUE, "txt", NULL, OFN_FILEMUSTEXIST |
OFN_HIDEREADONLY, "Text Files (*.txt) | *.txt All Files (*.*) |*.* ||");

            if (dlg.DoModal() != IDOK) return ;
            CString sFilename = dlg.GetPathName();
            // You can use the filename now.
      }
```

As you can see, the C++ implementation of the same feature is much more elegant and simpler than the C implementation. CFileDialog encapsulates all of the structures and function initializations.

Drawback of C++

What is the drawback of using C++ in implementing your Windows project? In my honest opinion, the best reason for using C++ is also the biggest drawback. If your project's features fit the class library very well, you can use the class directly or with little modifications. However, if you want to differentiate your product from what's avaiable in the market, you will have to modify the existing class library by creating inheritance from the base class, understanding the class interface and implementing the needed virtual functions of the inherited class. The benefit of the encapsulation in a class library also hide a lot of details from you. You have to spend time to peel the layers off the class hierarchy and understand each layer's functionalities so your new class adds features to the existing class in an optimized fashion. From my experience of doing C++ projects, to do a reasonable job in deriving a new class from a class framework you should do the following:

1. You need to understand the existing class hiearchy. In this example, you should know that the "CFileDialog" class is derived from the "CDialog" class. CDialog is derived from the "CWnd" class, and so on. You can find out the MFC class hierarchy relationship from the MFC Help's "Hierarchy Button".

2. You need to understand what are the public/protected data members and methods for every classes in that particular hiearchy.

3. You need to understand the virtual functions in the hiearchy and how these virtual functions interact between layers of the classes. For example, you should know that the Windows SDK's subclass mechanism is inherited in the "virtual LRESULT WindowProc(UINT message, WPARAM wParam, LPARAM lParam); " function.

As you can see, encapsulation really complicates the process of understanding the class library. In my project, in order to extend the existing class successfully, I have to spend about five times the amount of time to learn the existing C++ code than C code. After I understand the existing C++ code, I spend about the same amount of time writing the new code. The biggest benefit of object oriented design is that the interface to that particular object is very well defined by the class methods. Almost all of the modifications I made were internal to the class. The interface between this object to the rest of the system changes very little. Thus, after I finish modifying the class, with only minor change to the interface, I can easily test the new system. Testing of the new system is much easier.

Why customize the Common Dialog?

The simplest form Common dialog is great, if it fits your needs. With just six line of C++ code, your users can get the professionally designed dialog box for open and save file. Most important benefit of all, there are fewer chance for errors. However, most of the time, getting a single file name from the dialog box may not be exactly what you'd like to do. Also properly designed and customized dialog box make your program easier to use. It adds value and a professional look to your final product.

For example, in my Interactive Cross Reference for Windows (IXFW) program, the FileDialog has two additional controls:

1) Add a listbox to the dialog.

2) Add an "Add File" button to the dialog box.

When an user pushs the "Add File" button, the dialog box will add the selected files in

the file listbox to my custom listbox. This way, the user can select all the files he wants from different directories before the file dialog box is closed and the program continues. The user can save a lot of tedious mouse/keyboard actions when he tries to gather files from different directories. A very good value is added to the final product.

How to customize the common dialog in C?

The easiest way learn how to customize the file dialog box with C is by studying the excellent article "Using and Customizing Common Dialogs", written by Kraig Brockschmidt of Microsoft System Developers Relations. You can download this file from CompuServe. Basically, to customize the file dialog, you need to do the following:

1) You need to edit the dialog template by adding the new controls to the default template.

2) You have to hook the dialog message processing procedure by subclassing the dialog control.

3) You need to initalize the proper data structure.

4) You have to write message handler code to support the new controls in your dialog template.

How to customize the Common Dialog in C++?

You customize the File dialog in a very similar fashion. However, it's much more desirable to use all the Visual C++ 's tools in the customization process. Here is how to do that:

1) Use the "App Studio" to customize the template.
I did this by inserting the default file dialog template into my .rc file. You also include the "dlgs.h" in the include section of the .rc file, because most of the resource IDs defined for the file dialog box are located in the "dlgs.h" file.

2) Derive a new class from the CFileDialog class with "Class Wizard".
The best way to do this is to run the Class Wizard from inside the "App Studio". Since I can't tell the Class Wizard to derive a new class from the CFileDialog directly, I have to tell the Class Wizard to derive the new class for the dialog template from the CDialog class. After the Class Wizard created the necessary files for the new class, I edited the parent class from CDialog to CFileDialog. (You need to change the constructor for this particular implementation.)

3) Subclass the new dialog box.
I didn't do it here, since the dialog box is subclassed automatically in the MFC class hiearchy. However, if you want to use any of the fancy features like customizing the Common Dialog, you really should understand how MFC subclasses a window and how the messages are routed to your handlers. Since the subclass mechanism is in the MFC's class hiearchy already, I can use the Class Wizard to add handler functions to the dialog controls' messages. The handlers I added are: a) OnOK() - to handle the OK button from the default actions to the new action of adding selected files to my listbox. b) OnClickButton1() - for adding files to the listbox. c) OnClickButton2 - for deleting files from the listbox.

I also added several data members which map to the new control IDs with Class Wizard. This way, I can use the class methods of the CButton or CListBox instead of trying to figure out how to send control messages to those IDs. Isn't C++ wonderful?

One last data member I added to this particular class is the "CStringArray" for storing the collection of filenames after the OK button is pushed. You see, when you push the OK button, the filedialog will destoy the dialog box with all the controls in it. You have to copy the strings from the listbox control to a safe location before the listbox control is destroyed.

4) Properly initialize the new class in the constructor.

You have to tell the "OPENFILENAME" structure to use the new template. This is done in the constructor of this particular class.

5) Add methods to retrieve the collections of filenames from the dialog class.

Since the CStringArray data member is public, I choose to let the outside world have access to the filelist directly. Not the purest form of C++, but....

OK, all of the files needed to compile and run the new FileDialog Box are in the filedia.zip file. Have fun!

* If you have any comments, questions, suggestions, feel free to send email to Tony Lee at CompuServe 72064,1235.

** "Interactive Cross Reference for Windows" is a shareware utility that lets you build a database from existing source code so that you can quickly browse megabytes of source code. You can download the program from IBMPRO forum of the CIS.

Letters

From: Pat White <P.WHITE@fs2.mbs.ac.uk>
Date: 12 Aug 93 10:20:18 BST
Subject: WPJ Mag

Pete and Mike,

Many thanks for your efforts in producing the magazine - I'm brand new to Windows programming and am finding a lot of useful info. Just one very minor criticism - when you review shareware could you always include where it can be found, especially if it is in either the Simtel or CICA archive as both are widely mirrored here in Europe.

I am including in this note two lists you might want to include in a future issue. The file index is a combination of the READ.ME files, The article index is built from the contents of the WPJVxNx.TXT files. They cover the first 6 issues - I hope to retrieve number 7 today.

[Editor's Note: We felt this information would be useful for everyone, so it's included here and also seperately as an ASCII file. Thanks Pat.]

File index
Windows Programmer's Journal

Volume 1 Number 1
January 1993

The following files are contained in this archive:

TRASH.PAS	Pascal Source for trash can
TRASH.RC	Resource code for trash can
SUBMIT.TXT	How to submit an article to WPJ
LINKLIST.C	C source code for linked list
LINKLIST.MAK	Microsoft nMake compat make file
LINKLIST.DEF	Module Definition file for linked list
LINKLIST.RC	Resource file for Linked List
LINKLIST.H	Header file for Linked List
LINKLIST.ICO	Icon for Linked List
MAKELIST.BAT	MSC 6.00 with 3.0 SDK batchfile to make
PMDDE.C	Program Manager DDE C source code
WPJV1N1.TXT	Windows Programmer's Journal Volume 1 Number 1
README.TXT	Take a guess.

Volume 1 Number 2
February 1993

The following files are contained in this archive:

HELLO.ZIP	Source Code to the Hello World Program
LINKLIST.ZIP	Mike's revised Linked List program
HAXTON.ZIP	Rod's DLL examples
D&DCLIE.PAS	Andreas Furrer's Drag and Drop Client Program
D&DSERV.PAS	Andreas Furrer's Drag and Drop Server Program
WPJV1N2.TXT	Windows Programmer's Journal Volume 1 Number 2
READ.ME	Well, I screwed it up in the last issue (pete).

SUBMIT.TXT How to submit articles to us.

Volume 1 Number 3
March 1993

The following files are contained in this archive:

WPJV1N3.TXT The magazine in plain text format
WPJV1N3.HLP The magazine in Windows Help format
WPJ.BAT Quick way to view the WPJV1N3.HLP file in Windows
BEGINNER.ZIP Beginner's Column source code
OWNERBTN.ZIP Owner Draw Buttons source code.
INSTALL3.ZIP Source code for the Install Program article
PAINT.ZIP Advanced C++ source code
READ.ME Guess!
SUBMIT.TXT How to submit articles to us.

Volume 1 Number 4
April 1993

The following files are contained in this archive:

WPJV1N3.TXT The magazine in plain text format
HELPREAD.ME Explanation of why there's no help file this month.
ROD.ZIP Rod Haxton's DLL code.
HELLO.ZIP Beginner's Column source code (Dave Campbell)
ODLBOX.ZIP Owner-Draw list boxes
READ.ME Guess!
SUBMIT.TXT How to submit articles to us.

Volume 1 Number 5
May 1993

The following files are contained in this archive:

WPJV1N5.TXT The magazine in plain text format
WPJV1N5.HLP The magazine in WinHelp format
HELLO.ZIP Beginner's Column source code (Dave Campbell)
WINSTUB.ZIP Windows STUB source code
READ.ME Guess!
SUBMIT.TXT How to submit articles to us.

Volume 1 Number 6
June 1993

The following files are contained in this archive:

WPJV1N6.TXT The magazine in plain text format
WPJV1N6.HLP The magazine in Winhelp format
HELLO.ZIP Beginner's Column source code (Dave Campbell)
EDITPRO.ZIP Goes with Eric Glass' article in Vol. 1, No. 5.

READ.ME Guess!
SUBMIT.TXT How to submit articles to us.

Article index

Volume 1 Number 1

Volume 1 Number 2

Volume 1 Number 3

Keep up the good work.

Pat White

Pat White, Manchester Business School, Manchester, UK

JANET address - P.WHITE@UK.AC.MBS.FS2
EARN/BITNET - P.WHITE@FS2.MBS.AC.UK
INTERNET - P.WHITE@FS2.MBS.AC.UK
or - P.WHITE%FS2.MBS.AC.UK@NSFNET-RELAY
UUCP/USENET - P.WHITE%FS2.MBS.AC.UK@NSFNET-RELAY

[Thanks for the note and the index, Pat. I think people will find this useful. We'll have to start doing this every six issues or so. - mfw]

Getting In Touch With Us

Internet: 71644.3570@compuserve.com

GEnie: P.DAVIS5 (Pete)

CompuServe: 71141,2071 (Mike) 71644,3570 (Pete)

WPJ BBS (703) 503-3021 (Mike and Pete)

You can also send paper mail to:

Windows Programmer's Journal
9436 Mirror Pond Drive
Fairfax, VA 22032
U.S.A.

In future issues we will be posting e-mail addresses of contributors and columnists who don't mind you knowing their addresses. We will also contact any writers from previous issues and see if they want their mail addresses made available for you to respond to them. For now, send your comments to us and we'll forward them.

The Last Page

by Mike Wallace

"*The first thing we do, we kill all the OS/2 programmers.*"
-William Shakespeare

No, this isn't an OS/2-bashing column. Pete and I spent a couple of days last week at the Software Development '93 convention in Boston meeting people and going to conferences (see Pete's "Software Development '93" article elsewhere in this issue). Here are some notes:

One of the speakers I saw (from Borland) gave a presentation on an informal survey he took of programmers. One of the questions he asked was whether they thought programming was an art or a science. A majority said "an art", and I agreed with this at first, but then after thinking about it for a while I had to change my mind. Take physics, for example. Most people would consider that a science (versus an art). What's involved with a physicist's work? Usually it starts off with an idea or an observation about the way things are in nature. He then takes that idea and applies the laws of physics to it in an effort to understand what's happening. He's "doing Physics." For instance, in 1960, Edward Lorenz created a machine that simulated weather (the "Royal McBee") and after staring at it for a while, he thought he noticed a pattern to the cloud formations. He started working on the math, and along the way helped create the field of chaos, an unusual and fascinating branch of physics. My point is that he had an initial burst of creative thought that preceded the remainder of his work in the field, which sounds like programming. When I start on a program, I usually first think about how I'm going to approach the problem and then I use what I know about programming to implement my solution, so maybe programming is as much of a science as Physics, and not so much an art.

Another question this speaker asked in his survey was about favorite foods for programmers. The winners were pizza, beer and hamburgers. The loser, coming in as the favorite food of 1% of those programmers surveyed, was fresh vegetables. I'm glad I'm not the only one favoring beef over beets.

The speaker also asked about favorite outside hobbies. The winner was "Watching Star Trek reruns". How did I know he was going to say that? The audience gave a very loud cheer when he announced that. Am I the only programmer who doesn't watch "The Next Generation"? [Pete's Note: Don't buy that for a second. Mike reminded me it was on the other night! Obviously he keeps better track of it than me!] I went to Symantec's party where they announced their C++ compiler, and it was a 30-minute skit straight out of "Star Trek". They talked about getting to the planet C++ and fighting the "Borland-ians" and "Microsoft-ians" from their space ship. It was pretty corny.

I spent a good while talking to the president of a well-known software company about what albums are good for prgramming to (you can tell I did a lot of work at this convention). We agreed on Pink Floyd's "The Wall". I'd also have to nominate Pearl Jam's "Ten" and Joe Satriani's "Surfing with the Alien". I want your responses. What music do you listen to when you're pounding out code? Van Halen with Visual Basic? "Purple Haze" with Pascal? Send me your comments, and I'll reprint the best ones after I get enough for a column (so it may take a while). Hope you enjoy this issue. Talk to you next month.

Hacker's Gash

By Dennis Chuah

In this instalment of Hackers Gash, we will look at system level entities, the ToolHelp library and extend the discussion to private class dialogboxes. Plus, we will continue last months metafile discussion. Here is a summary of the topics presented:

1. Instances, Modules and Tasks
2. When Should HINSTANCES, HMODULES and HTASKS be Used
3. List of Modules
4. Private Dialogbox Class
5. Metafiles revisited

All example programs are compiled with Borland C++ with the following options:

Memory model: MEDIUM
Optimization: Fastest Code
Entry/Exit Code: Smart Callbacks
Floating Point: None
Instruction Set: 80286
Calling Convention: C

Instances, Modules and Tasks.

Most of you should have come across instances. Windows passes a handle to an application (HINSTANCE) in the WinMain function. Calls made to RegisterWindow and CreateWindow require this value. A HINSTANCE is something you have used often but have you stop to think of what a HINSTANCE is? Then there is the handle to a module (HMODULE) and the handle to a task (HTASK). You have heard of them before, but have never needed to use them. What are they anyway?

I wrote a .DLL and an application to illustrate these system entities. The source code and executable are in the file GASH.ZIP, in the TEST subdirectory. This .ZIP file contains subdirectories, so remember to use the **-d** flag when unzipping it. Figure 1a shows the output of the application.

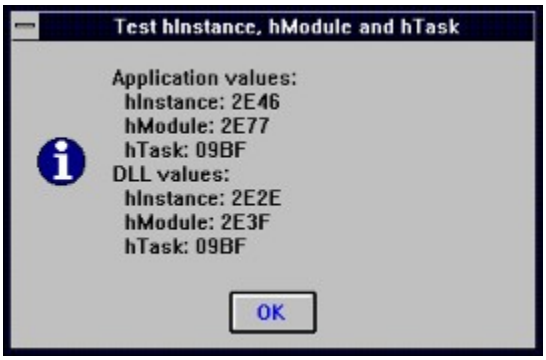


*Figure **1a***

I then ran another instance of the application and the result is shown in figure **1b**.

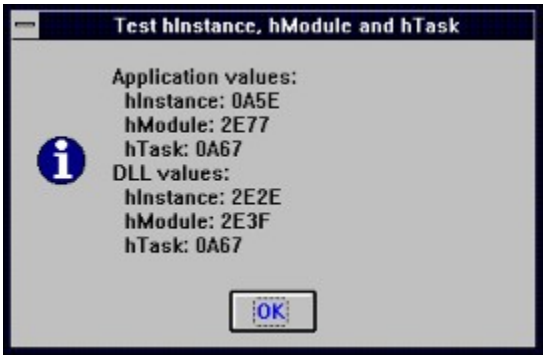


*Figure **1b***

With the aid of figures 1a and 1b, we can see that the HINSTANCE is unique for different instances of the application. This is sensible -- HINSTANCE uniquely identifies an application instance. The HINSTANCE for the .DLL however, is the same across different instances of the application that used it. This is also sensible, as there is only one instance of a .DLL no matter how many times it is used. HMODULE appears to be the same across different instances of the application and .DLL. HTASK on the other hand, is the same within an instance, whether it was obtained from the .EXE module or the .DLL.

A little explanation is in order. HINSTANCE identifies the current instance. This is why every instance of an application has a unique HINSTANCE. For every instance of an application, Windows creates a data segment (This is usually the case as the module definition file in an .EXE usually specifies multiple data segments). Although undocumented, I have found that HINSTANCE is the handle to the applications data segment.

Take figure 1a for example. I ran the test application and used Turbo Debugger to debug it. On examination, the DS register has the value, 0x2E47. I then inspected hInstance with the following Turbo Debugger typecast:

```
(gh2fp) (unsigned) hInstance
```

This is equivalent to doing a GlobalLock on hInstance. The value was, 2E47:0000.

Presto! The selector value was exactly the same as the value in the DS register.

Undocumented: HINSTANCE is the handle to the data segment.

Every .DLL has a unique HINSTANCE, but as the .DLL has only one data segment that is shared across every application that makes use of it, there is only one HINSTANCE value. As a rule, DLL modules are only instanced once no matter how many times they are used. EXE modules on the other hand are instanced once for each time they are used.

Knowing this is they key to understanding why exported functions in an application needs to instanced through MakeProcInstance and those in a .DLL do not. Window functions do not need to be instanced. A field in the windows information structure is GWW_HINSTANCE. When messages are dispatched, the correct data segment is bound to the window procedure via this HINSTANCE value. Subclass procedures must be instanced before use. The HINSTANCE in the GWW_HINSTANCE field refers to the window procedures data segment, not the subclass procedures. Also, the subclass procedure must not call the window procedure directly, but should instead use the CallWindowProc function. CallWindowProc binds the HINSTANCE value in the GWW_HINSTANCE field to the window procedure.

As explained in the last instalment, exported functions compiled with Borlands smart callbacks entry/exit code automatically bind their data segment on entry.

HMODULE refers to the file that Windows loads the code segments, resources, etc. It can either be an .EXE or a .DLL file. Well, actually, HMODULE is used to keep track of resources that are associated with the .EXE or .DLL file. Take a look at the following declaration for CLASSENTRY.

```
typedef struct tagCLASSENTRY
  {DWORD   dwSize;
   HMODULE hInst;
   char    szClassName[MAX_CLASSNAME + 1];
   WORD    wNext;
  } CLASSENTRY;
```

This structure is used by the ToolHelper function, ClassFirst. We can see that there is a HMODULE field in it. When a module is freed, all classes associated with the module will also be freed. When a class is registered, Windows obtains the HMODULE from the specified HINSTANCE. The class is then bound to the module specified by the HMODULE value. This is why in Windows 3.1, only one instance of an application is supposed to register classes. Every instance of an application share the same HMODULE.

Tasks are like logical CPUs. Each has its own instruction pointer and set of registers. Each task also has its own stack. The stack is created in the applications data segment (the data segment bound to the .EXE module). Every application on the desktop (including the shell application -- eg. Program Manager) is a task. In 386 enhanced mode, all tasks run in a virtual machine in protected mode. Every DOS window is a virtual machine that runs in real mode. Windows performs preemtive multitasking across virtual machines but only performs cooperative multitasking across the tasks within the protected mode virtual machine.

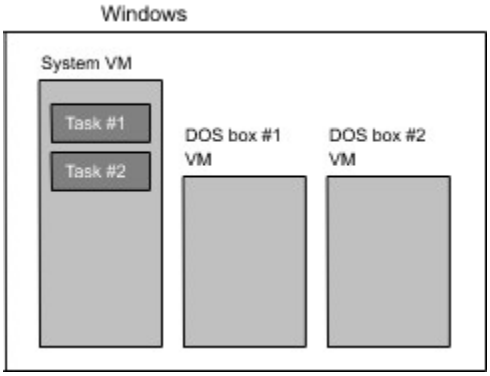


*Figure **2***

For each task, Windows creates an execution context (HINSTANCE) associated to an .EXE module. The task may call functions in other modules. The functions entry code will switch the execution context to the modules. Unless the .DLL switches stacks, it executes on the tasks stack. Execution context switching is transparent to the programmer, but incurs a penalty in CPU clock cycles.

Processors > 80286 cache their segment registers. Well, they actually cache the descriptors pointed to by selector registers (segment registers in real mode). As a result, when a selector is loaded, the CPU reloads the descriptor cache, taking up valuable CPU clock cycles. A call causing an execution context switch causes two descriptors to be reloaded (one each for the code segment and data segment).
When a task calls GetMessage, PeekMessage, WaitMessage, Yield or any other API that causes a dialog box to pop up, a task switch may occur. If there are other tasks waiting to be executed and any of those functions were called, Windows will switch the next task in the task queue in. Although task switches are transparent to the programmer, a misbehaving application can prevent Windows from task switching. As I have mentioned earlier, Windows operates a cooperative multitasking system, meaning it is up to well-behaved applications to give their time-slice back to Windows periodically. A task switch incurs a lot more CPU clock cycle penanty than an execution context switch.

Trap: Code that causes too many task switches can cause Windows to slow down due to CPU clock cycle penalty. This penalty is usually refered to as the multitasking overhead.

When Should HINSTANCE, HMODULE and HTASK be Used?

A number of Windows functions accept and return HINSTANCE. The following functions accepts HINSTANCE as a parameter.

FreeModule
GetModuleFileName
GetModuleUsage

It may seem strange that these functions accept HINSTANCE when they actually deal with modules. The fact is, both HINSTANCE and HMODULE can be used. It is relatively easy to obtain the HMODULE from a given HINSTANCE. The header file, WINDOWSX.H contains a macro definition called GetInstanceModule that accepts a HINSTANCE and returns a HMODULE. It passes the HINSTANCE to the low order word of the lpszModuleName parameter in the call to GetModuleHandle. This function returns the HMODULE. The following functions also accept HMODULE in addition to HINSTANCE.

FreeLibrary
GetProcAddress

The following functions return HINSTANCE:

LoadLibrary
LoadModule
WinExec

As explained earlier, the GetModuleHandle function returns a HMODULE value.

The class registration functions (RegisterClass, etc), window functions (CreateWindow, etc) and resource functions (LoadResource, etc) all accept HINSTANCE. MakeProcInstance and GetInstanceData expect HINSTANCE.
To extract HINSTANCE from a given HMODULE is not straightforward, and in some cases, impossible. This is because .EXE modules can have more than one execution context (one per instance). Therefore, given an .EXE HMODULE, it is still ambigiuos as to which HINSTANCE is to be extracted. If there is only one instance of an application running, the ToolHelper library can be used to extract the HINSTANCE.

However, it is possible to extract the HINSTANCE of a .DLL given the HMODULE. The following function obtains the HINSTANCE of a .DLL from a HMODULE. It retrieves the full path name of the module and loads it. This increments the usage count of the module. The LoadLibrary functions, in addition to loading the module, returns the HINSTANCE. The module is then freed to decrement the usage count and the HINSTANCE returned.

```
HINSTANCE GetDllModuleInstance (HMODULE hModule)
  {static char path[256];
   HINSTANCE hInstance;

   GetModuleFileName (hModule, path, 256);
   hInstance = LoadLibrary (path);
   FreeLibrary (hInstance);
   return hInstance;
  }
```

It assumes that the HMODULE passed to it is a handle to a .DLL module. GASH.ZIP contains the full source code to the above function, in the INSTANCE subdirectory.

Tip: As it is easier to obtain HINSTANCE, it is recommended that it be used in place of HMODULE where possible. It is the value passed to the WinMain and LibMain functions. The GetWindowWord function will return the HINSTANCE of a HWND if GWW_HINSTANCE is specified.

The GetCurrentTask function returns the HTASK for the current application. EnumTaskWindows and PostAppMessage both accept HTASK as a parameter. The ToolHelp library contains several functions that allow all tasks to be enumerated.

List of Modules

In the INSTANCE\TEST subdirectory of GASH.ZIP, there is a test application named, TESTINS.EXE. The full source code is also contained in the same subdirectory. TESTINS is an application that displays a list of modules in the system, their names, their usage count

and their full path name. The **Update** button updates the list. As the list is not updated automatically, any modules added or removed from the system will not show in the list. The **hInstance** button displays the hInstance of the module (provided it is a .DLL) in a messagebox.

The ToolHelp library is used to obtain the list of modules in the system. The ModuleFirst function was called to return the first module in Windows module list. Information is returned via the MODULEENTRY data structure. The ModuleNext function was then called repeatedly until every module was enumerated.

Tip: Although the ToolHelp library is meant to be used by debugging applications, it is by no means limited to these applications. Non-debugging applications can make use of the functions exported by the ToolHelp library to get access into Windows internal data. ToolHelp provides a documented set of API that allows applications to access undocumented Windows API. To use the ToolHelp library, include the TOOLHELP.H header file. All the exported functions are already in the default import library (IMPORT.LIB).

When a selection is made in the listbox, another ToolHelp function is called, the ModuleFindHandle. This function returns information regarding a hModule in the MODULEENTRY data structure. Information in the structure is then used to update the usage count and path information boxes.

When the **hInstance** button is pressed, ModuleFindHandle is called again. A rudimentary method was used to determine whether the selected module is a .DLL. If it was determined that it is a .DLL, the GetDLLModuleInstance function is called to return the hInstance of the .DLL.

Private Dialogbox Class

The TESTINS application uses a private class modeless dialogbox as its main window. The application is a simple one. There is no need for the normal sort of window. A dialogbox significantly simplifies the coding. Why did I use a private class dialogbox? In this example, the main reason was instructional.

Tip: Private class dialogboxes provide an easy way to associate an icon to a dialogbox. To create a private class dialogbox, define the dialogbox template in the resource file with the optional CLASS statement. Specify the name of the class.

```
MyDialogBox DIALOG 20, 20, 183, 193
STYLE ...
CLASS "myclassname"
BEGIN
   .
   .
   .
END
```

In your application, register the class with the handle to the icon assigned to the hIcon field of the WNDCLASS structure. Specify DefDialogProc in the lpfnWndProc field. **Important:** The cbWndExtra field must be larger or equal to DLGWINDOWEXTRA.

```
if (hPrev == NULL)
```

```
{wc.style = CS_HREDRAW | CS_VREDRAW;
 wc.lpfnWndProc = DefDlgProc;
 wc.cbClsExtra = 0;
 wc.cbWndExtra = DLGWINDOWEXTRA;
 wc.hInstance = hInstance;
 wc.hIcon = LoadIcon (hInstance, MAKEINTRESOURCE (MAIN));
 wc.hCursor = LoadCursor (NULL, IDC_ARROW);
 wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
 wc.lpszMenuName = NULL;
 wc.lpszClassName = MAINCLASS;
 RegisterClass (&wc);
}
```

If you wish to use any of the window extra bytes, you must add the number of extra bytes to this value. To access those bytes, start from offset DLGWINDOWEXTRA instead of 0. Use the usual way to create the dialogbox.

```
hWnd = CreateDialog (hInstance, "MyDialogBox", NULL, DlgProc);
```

CreateDialog create a modeless dialogbox, while DialogBox creates a modal one. Specifying DefDialogProc in the lpfnWndProc field allows the normal way of using a dialogbox (ie. using a DIALOGPROC callback) to be employed. This is easier than writing a window procedure that has DefDialogProc characteristics.

Trap: Failing to set the cbWndExtra field to DLGWINDOWEXTRA or larger will cause the USER module to GP fault!

Trap: Failing to call IsDialogMessage in the message loop will result in your dialog box losing the normal dialogbox keyboard interface (ie. tab moves from control to control).

Metafiles revisited:

Last month, we looked at using metafiles saved in an applications resources. GASH.ZIP contains the source code of a function that loads a metafile from an instances resources. It is in the MET1.C file in the METAFILE subdirectory. The associated header file, METAFILE.H, contains a #define that defines the symbol METAFILE to the value 2000. Include this file in your resource (.RC) file. To include a metafile, use the following line;

```
MetafileName  METAFILE "filename.wmf"
```

where MetafileName is the name to be given to your metafile. It can either be a #define integer symbol or a string symbol. If you use a #define integer symbol, be sure to use the MAKEINTRESOURCE macro.

I also mentioned that Windows API cannot handle metafiles with a placeable header. In this instalment I will show you the format of a placeable metafile header and write a few functions that can manipulate metafiles with placeable headers. MET2.C and MET3.C in the METAFILE subdirectory of GASH.ZIP contains two functions that reads and writes to disk based metafiles with placeable headers. The functions are GetMetaFileBetter and CopyMetaFileBetter.

Metafiles with placeable headers are ordinary metafiles with an additional 22-byte header. The following show the structure of this header:

```
typedef struct
  {DWORD   key;
   HANDLE  hmf;
   RECT    bbox;
   WORD    inch;
   DWORD   reserved;
   WORD    checksum;
  } METAFILEHEADER;
```

The key field contains the magic number, 0x9AC6CDD7L. This identifies the metafile as one with a placeable header. Ordinary metafiles do not have an identifying signature.

Tip: The first four bytes of a placeable metafile contains the following signature: 0x9AC6CDD7L. To identity ordinary metafiles is not as easy. You can use the signature 0x90001L but this may not be unique. Metafiles created by Windows version 3.0 and 3.1 contains the value 0x300 in the following 2 bytes. This value is 0x100 for metafiles created by earlier versions of Windows.

The hmf and reserved fields should be set to 0.

The bbox field specifies the bounding rectangle of the metafile. It is specified in metafile units.

The inch field specifies the metafile units in units per inch.

The check sum field is an XOR value of the first 10 words in the header. The following C statement calculates this value:

```
checksum = LOWORD (key) ^ HIWORD (key) ^
           LOWORD (*((DWORD *) &bbox)) ^ HIWORD (*((DWORD *) &bbox)) ^
           inch
```

The LOWORD/HIWORD key values can be replaced with the constants 0xCDD7 and 0x9AC6.

The GetMetaFileBetter function

The GetMetaFileBetter function reads a metafile just like the GetMetaFile function in Windows API. However, GetMetaFileBetter knows how to distinguish between an ordinary metafile and a metafile with a placeable header. It accepts one additional parameter, lpMh, a far pointer to the METAFILEHEADER structure. If a valid pointer is specified and a metafile with a placeable header was read, the placeable header will be copied into this structure.

GetMetaFileBetter loads the first 22 bytes of the specified metafile. It checks it this is a placeable header. If it is not, the GetMetaFile function is called to load the metafile. The lpMh parameter is not filled in. If GetMetaFileBetter finds a placeable header, memory is allocated to load the rest of the metafile, minus the placeable header. The metafile is then loaded and the SetMetaFileBitsBetter function is then called to convert the loaded file into a memory metafile.

The CopyMetaFileBetter function

The CopyMetaFileBetter function copies a metafile to disk with an optional placeable header, otherwise it functions similar to the CopyMetaFile function in Windows API. Unlike CopyMetaFile, CopyMetaFileBetter accepts one more parameter, lpMh, a far pointer to the METAFILEHEADER structure. If this pointer is valid and the structure filled in correctly, CopyMetaFileBetter will create a disk metafile with a placeable header. The placeable header will not be copied if a memory metafile is specified as the destination (by specifying NULL for the lpszFile parameter).

If the lpMh parameter is not a valid pointer or if the structure wasnt filled in correctly, or if a memory metafile is specified as the destination, CopyMetaFileBetter calls CopyMetaFile to copy the metafile. Otherwise, CopyMetaFileBetter opens the specified file and writes the placeable header. It then calls the CopyMetaFile function to create a duplicate of the metafile and passes the duplicate to GetMetaFileBits function to retrieve a handle to the metafile data. This data is then written to disk and the file closed.

An example usage of GetMetaFileBetter and CopyMetaFileBetter

The METAFILE\TEST subdirectory of GASH.ZIP contains an application (TESTMET.EXE) that tests the GetMetaFileBetter and CopyMetaFileBetter functions. TESTMET reads in GASH1.WMF, a metafile with a placeable header, and displays it in the client area of its window. It then saves two metafiles, GASH-YES.WMF and GASH-NO.WMF. GASH-NO.WMF is saved from the memory metafile read from GASH1.WMF. It is an ordinary metafile. GASH-YES.WMF is also saved from the same memory metafile but it contains a placeable header. You can use DOSs fc to compare GASH1.WMF and GASH-YES.WMF. GASH2.WMF is GASH1.WMF without the placeable header. Use fc again to compare GASH-NO.WMF with it. By the way, GASH1.WMF is this articles logo in metafile format.

Next month...

Next month, I will write on these topics:

More metafiles	DIBs in metafiles
	Dealing with misbehaving metafiles
	Decoding the placeable header information
	Text placement in metafiles
	Placing a metafile
Moveable windows	Child windows that resize and move every time the parent is resized
	Window panes - splitting a window into resizeable panes
MDI windows	Accelerator for each type of MDI windows
Dialogboxes	Integer vs string resource names
	Dialogbox font
	Dialogboxes as child windows
	Dialogboxes as MDI child windows
	Coloring dialogbox controls
	Keyboard interface and hotkeys

The author:

I am currently doing a PhD. degree in Electrical And Electronic Engineering at the University Of Canterbury. My programming experience dates back to the days where real programmers code with Z80 machine code. For the past two years I have taken an interest in Windows programming.

Please send any comments, questions, criticisms to me via:

email: **chuah@elec.canterbury.ac.nz**
or post mail: **Dennis Chuah**
c/o The Electronic and Electrical Engineering Department
University of Canterbury
Private Bag
Christchurch
New Zealand.

All mail (including hate mail) is appreciated. I will try to answer all questions personally, or if the answer has general appeal, in the next issue. If you will like to see a particular topic discussed in up and coming issues, drop me a line. I also appreciate discussions on topics published. If you are sending me email, please make sure you provide me with an address that I can reach (most internet and bitnet nodes are reachable, and so is compuserve).

Installing Windows NT
(As Done by a non-Windows Guru)

By Kurt Simmons

A couple of months ago at a company meeting it was decided that we should get in on the ground floor of software developing for Windows NT, and that we would order the preliminary release of the SDK. When I called Microsoft, I found that the CD was $69.00 and that the documentation would be more than $300, if you wanted a hard copy. I figured that if it was on the CD that would be sufficient, even if it was in PostScript format (though in fact some is in Windows Write format).

After breathlessly waiting a week, the box arrived; finally we too would enter the next step in giving a PC a "real" operating system. I had moved many of our files around to make over 200Meg free on a drive so that even Windows NT could not complain during installation. Then I found that we can't run Stacker with Windows NT, nor could we run the current drivers for our LAN (LBL). NT requires that you use 32bit drivers. While this will improve performance, it made life difficult at the current time. OK, back to moving files around and freeing up a drive that is unStacked. Two hours later, I am back on my way to installing NT.

NTFS: would I like to convert my drive to that format? Options are available to convert with or without destroying the existing files or to simply keep the existing FAT format. The first time through I figured I would keep the existing format (an option exists to convert later if we desire). After some trial and error of getting NT configured to work with the Ethernet cards, I decided to give up and completely re-install as I have changed so many of the settings (you'd think after doing this sort of thing for this long I would have learned to keep notes!) So, here I go again installing it for the second time. By now I have read about the advantages of using the NTFS for reasons of security and data integrity; I figured that it would be a good idea to convert the drive to the format that Windows NT will be most comfortable with.

Everything went along fine until I found out that "Converting while leaving your files intact" doesn't mean you can boot off of that drive anymore or even access it from MS-DOS! Well, since I needed to be able to run DOS as my primary OS at this point, I realized I must reformat the drive. Except I couldn't do that from within NT. So I booted from floppy and found that my D: drive was now listed as my C:, and C: was nowhere to be found. The installation program for Windows NT was kind enough to reformat the drive in MS-DOS for me after my partner got done performing CPR.

The moral of the above story is that taking a day or so to search the on-disk documentation might have done us some good. In the DOC\ENDUSER\WRITE (this is the first place I would look for information on "How to plan your installation") directory were a series of files on installing Windows NT. I found that I had become a little too used to the standard OS of the PC's being more than a bit lax in security, I should be able to destroy my system from the inside if I want to, right?

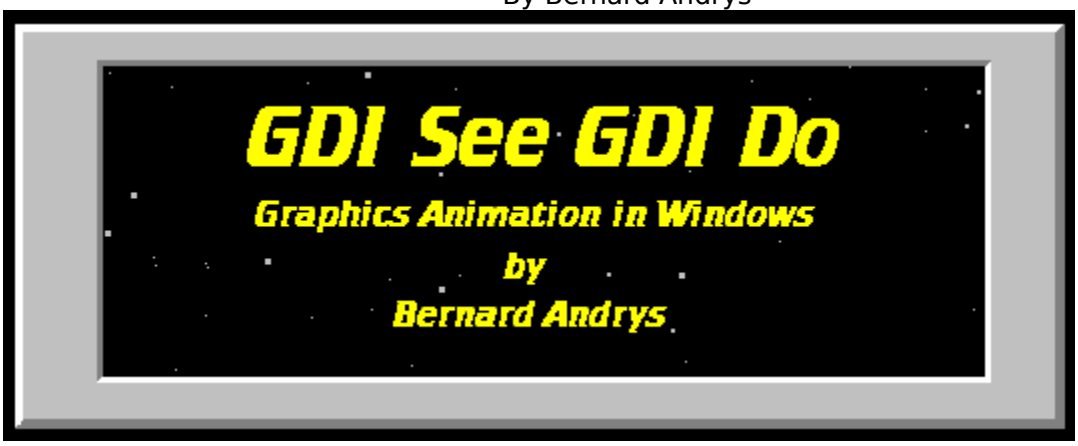
The first eight hours of NT seemed to have drained me to the point of actually considering giving up caffeine as a major food group, but after that life got easier. I hope to be able to continue my adventures in NT next month. The articles will be written from the point of a programmer but not a Windows NT guru, as I think many of us fit that description.

Kurt A. Simmons
Systems Administrator
Millennium Technologies, Inc.

Suite 100
160 Edgehill Rd.
Glenside, Pa 19038-3004
Voice (215) 886-7366 Fax (215) 886-8602

Gdi See Gdi Do
Graphics Animation in Windows

By Bernard Andrys



Well, it's next month already and I bet all 3 of my readers are wondering when I'll turn my "Do Nothing Program" into a way cool, time wasting, blow'em'all'up arcade game.

But first, a VERY, VERY important correction to the last article:

As was pointed out by Michael Birk (one of the three readers), using the PASCAL keyword for a function declaration does not make a function pass parameters by reference. I read about the purpose of the PASCAL keyword in an article but I really should have tried passing parameters using the PASCAL keyword before claiming that it works. I apologize for any confusion that it may have caused.

...and now back to blowing things up.

In this article, we'll at least get something moving. We're going to put the missile base on the screen and make it move when you press the cursor keys. Let's start with what we'll need to put a missile base on the screen and make it move:

1. A missile base
2. A way to put the missile base in the window.
3. A way to link the keyboard to the drawing of the missile base.

The first requirement is pretty easy. We need to define all the properties that a missile base has:

1. Its appearance
2. Its location
3. How many lives it has left.
4. Whether it can shoot or not. (We're only going to let it shoot one missile at a time.)
5. Has it been hit by an invader's missile? (so we know if we should draw an explosion instead of a missile base)

We'll work on the first two properties now and save the rest for later.

The first property, position, is its x and y coordinates. We'll use the variables x and y to represent the missile base's x and y coordinates. Before we use the variables x and y, we'll need to declare them. Declaring a variable tells the C compiler what type of variable

is going to be used. We should pick a data type for our variables x and y that is big enough to handle the largest values that might be assigned to x and y.

So how big are the data types that we have to work with?

Data Type	Size
char	-128 to 127
int	-32,768 to 32,767
unsigned char	0 to 255
unsigned int	0 to 65,535
unsigned int	0 to 4,294,967,295
float	3.4E38
double	1.7xE308
void	nothing

There are a lot of other data types supported by C compilers such as long int and short int but considering that Win32 and NT are just around the corner you probably don't need the confusion of learning X86 segmentation now. (Especially since this IS a beginner's column.) Windows has its own data types as well. For example HBITMAP and HINSTANCE are two data types that are specific to Windows. You would declare a variable as HBITMAP if the variable was going to contain a handle to a bitmap. A handle is a number that a program uses to refer to a file, bitmap, or some other object. You don't normally have to care about the size of Windows' own data types like HBITMAP or HINSTANCE because Windows' own data types are normally used as return values or parameters for Windows' own functions. For example, if we used a function called LoadBitmap() it would return a variable of type HBITMAP. The HBITMAP variable would then be used where ever we want to refer to the actual bitmap.

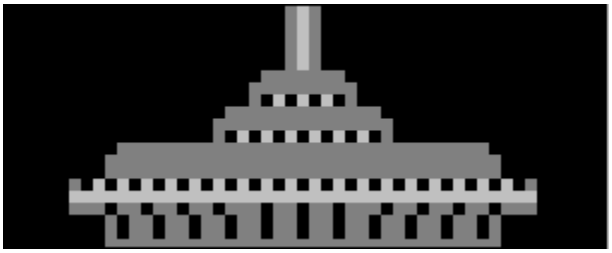
EXAMPLE:

```
MyFunction()
{
/* declare a variable called mybitmap which is data type HBITMAP */
HBITMAP mybitmap;
/* Window's function LoadBitmap() returns an HBITMAP variable */
mybitmap = LoadBitmap(hInstance, "bitmap");
}
```

Actually, Windows doesn't have its own data types. Data types like HBITMAP are defined in the file windows.h as an unsigned integer. You declare the variable as HBITMAP instead of unsigned integer because the size of HBITMAP might (and will) change under NT or maybe even in future versions of Windows like Windows 4.0. The size of HBITMAP could even change for each CPU that the program is compiled for (this is an NT consideration again).

Now that we know what data types are available, we can pick a data type for x and y. Since the window is 400 pixels high and 400 pixels wide, we should define x and y as type int. We'll probably have to use the position of the missile base in many places throughout our code so it's best that we define x and y as global variables by defining them in the header file "invid.h".

The second property, its appearance, is defined by a bitmap that we'll have represent the missile base. You can create the bitmap of the missile base using any program that can create .bmp files. We're going to use 16 color bitmaps so be sure to save the file in that format. Take a look at the bitmap, base.bmp, that I created to represent the missile base.



Notice that the bitmap has black pixels to its right and left. Since the missile base only moves right and left, and the background is solid black, we can cheat in creating the bitmap animation. We don't have to use a bitmap mask to draw the bitmap without disturbing the background. We also don't have to bother redrawing the background when the base is moved. As long as we move the base to the right or left 5 pixels or less at a time (There's a 5 pixel black border on each side of the base.), drawing the base covers over the previous base bitmap that was on the screen. This way we can just draw the bitmap to the screen directly.

To use a bitmap in a Windows program you first need to define it in your program's .rc file. Here is the contents of the invid.rc file:

```
#include "windows.h"
invid ICON        invid.ico
base  BITMAP      base.bmp
```

We've added the missile base's bitmap (base.bmp) to the end of the .rc file after the icon file. The first entry, base, will be what we call the bitmap file in the program's source code. The second entry, BITMAP, tells the resource compiler that this is a bitmap resource. The last entry, base.bmp, is the name of the bitmap file that we want to load. Now our program will contain the bitmap as a resource that can be loaded and used by our program. To use the resource, you call the function LoadBitmap(Program_Instance, "bitmap name"). The first parameter to LoadBitmap() is the handle of the program instance. So we'll need to save the hInstance variable that is passed to WinMain() by assigning it to our own variable hInst. That way we can use hInst when we call the LoadBitmap() function. The second parameter is the name of the bitmap that we defined in the .rc file. The LoadBitmap function returns a variable of type HBITMAP that we use whenever we want to refer to the bitmap. I added a variable called hbBase of type HBITMAP in our header file invid.h.

Fast and Friendly Animation

Now that we've got our missile base, how do we display it on the screen? There are two ways we can go about it:

1) Display the missile base at its position every time we get a Timer message.

2) Display the missile base at its position whenever it moves.

The first method would be ideal for multitasking friendliness. However, this method is too slow. At best, windows can only provide 18 timer messages per second. I originally thought that this would be fast enough. After all, TV is at 30 frames/sec, movies run at 24 frames/sec and Microsoft Video files look fine at 15 frames/sec. Unfortunately it didn't work out that way. 18 frames per second is just too slow for arcade game speed. We want the animation to be as smooth as possible. That means that if the user tapped the right cursor

key quickly, the missile base would move 1 pixel to the right. If the user held the cursor key down, the base would move 18 pixels/second across the screen. However, our window is currently 400 pixels wide. So it would take over 22 seconds to move the base from one end of the screen to the other. Twenty-two seconds is a LONG time in arcade style action games. This method just won't work with Windows only sending 18 timer messages per second. Well, we could make it work if we added motion blur but I'd rather not try and tackle motion blur ...yet. We could make this method work if we could get Windows to send us more than 18 timer messages per second. There are ways of doing this using the multimedia timer services but they seem fairly complicated to me. (The real reason is that I've been too lazy to upgrade from Quick C which only supports the Win 3.0 API.)

The second method is much easier to deal with but creates its own problems. Instead of having our program only do something in response to a message, we can write our program to run continuously. Unfortunately we need to be careful with this method because we want our program to be friendly to other Windows programs and not stop every other Windows program just because ours is running. To do this we can use Windows' PeekMessage() function to see if Windows has anything else to do. If it does, we'll let Windows take care of the other programs before coming back to our program. This way our program will run continuously as long as no other program has things to do. An obstacle to using PeekMessage is that you shouldn't keep your program in a PeekMessage loop because it keeps Windows from posting idle messages. Another problem with this method is that since our program runs as fast as the CPU will allow, we will have to do our own timing to keep the animation rate constant no matter what CPU we're running on.

I created two sample programs that demonstrate the difference between timer based animation and peek message based animation. The code in the two programs is essentially the same. Tball.exe creates a timer that sends 18 timer messages per second (the maximum under Win 3.1).

Condensed code:

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR lpCmdLine,
int nCmdShow)
{
  MSG msg;
  if (!hPrevInstance) InitApplication(hInstance));     InitInstance(hInstance,
nCmdShow)
  while (GetMessage(&msg, NULL, NULL,NULL))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return (msg.wParam);
}

LONG FAR PASCAL WndProc(HWND hWnd, unsigned message,
WORD wParam, LONG lParam)
{
  switch (message)
  {
    case WM_CREATE:
      // create a timer
      SetTimer (hWnd, 1, 1, NULL);
      // get handle to bitmap
      hbBall=LoadBitmap(hInst, "ball");
```

```
     break;

   case WM_TIMER:
/* if a timer message arrives, draw the ball */
     DrawBall();
     break;

   case WM_DESTROY:
     KillTimer(hWnd, 1);
     DeleteObject(hbBall);
     PostQuitMessage(0);
     break;

   default:
     return (DefWindowProc(hWnd, message, wParam, lParam));
  }
  return (NULL);
}

void DrawBall(void)
{
  MoveBall();
  DrawBitmap(hbBall, x, y);
  EraseOldBall();
  return;
}
```

click here

[Editor's note: To make this button work, you need to set the working directory (under "Properties" for the WPJ icon in Program Manager) to the directory you installed this issue of WPJ to. - mfw]

If you run Tball.exe, you'll notice that the ball moves fairly quickly and smoothly across the screen. Unfortunately this is as fast as you can get with timer based animation without making the animation jerky. The menu selection Speed allows you to control change in position of the ball per timer message. AnimeStep=1 means that the ball moves 1 pixel on every timer message. AnimeStep=50 means that the ball moves 50 pixels on every timer message. Changing how far the object moves per timer message is the only way to effectively control the speed of an object in timer based animation.

In contrast, if you run Pball.exe, you'll notice that the animation is smoother and faster. The code is different in one area, the message loop. Instead of the while(GetMessage()) loop, we have a loop that runs as long as a WM_QUIT message is not received. PeekMessage() checks to see if there are any messages waiting to be dispatched by Windows. If there aren't any messages waiting, the function DrawBall() is run. Notice that WndProc() doesn't do much. The Setup, Control, and Exiting of the Window is in WndProc, but all the animation is performed whenever Windows isn't doing anything else.

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
  MSG msg;
```

```
  if (!hPrevInstance) InitApplication(hInstance));     InitInstance(hInstance,
nCmdShow))
  while (TRUE)
  {
    if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
    {
      if (msg.message == WM_QUIT) break ;
      TranslateMessage (&msg) ;
      DispatchMessage (&msg) ;
    }
    else
    {
/* if no messages waiting, run DrawBall() */
      DrawBall() ;
    }
  }
  return (msg.wParam);
}

LONG FAR PASCAL WndProc(HWND hWnd, unsigned message, WORD wParam, LONG lParam)

{
  switch (message)
  {
    case WM_CREATE:
      hbBall=LoadBitmap(hInst, "ball");
      break;

    case WM_DESTROY:
      DeleteObject(hbBall);
      PostQuitMessage(0);
      break;

    default:
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }
  return (NULL);
}

void DrawBall(void)
{
  MoveBall();
  DrawBitmap(hbBall, x, y);
  EraseOldBall();
  return;
}
```

click here

The menu item Speed works the same in Pball as it did in Tball. AnimeStep controls how many pixels the ball is moved in every animation step. But instead of 18 animation steps per second, you have as many animation steps as your CPU can crank out before Windows has a message to process. If Windows has a message to process, the ball stops

moving while the message is handled. The ball will then go back to moving as soon as there are no more messages. Fortunately, messages are handled so quickly that you don't see any noticeable pause in the animation of the bouncing ball.

So let's apply this PeekMessage animation technique to The Game:

```
/* invid.c */
#include <windows.h>
#include "invid.h"

/*---------- Global Variables --------*/
/* In the real source code I keep my global variables in "invid.h" but it's
easier for you to read the source code if I put them here. */

/* Declare a structure called decodeWord that will be used by WinProc to run a
function in response to a message */
struct decodeWord {
            WORD Code;
            LONG (*Fxn)(HWND, WORD, WORD, LONG); };
/* An array called messages made up of decodeWord structures is created. The
array lists the Windows Message and the function that will be called when that
message is received.  I've added two messages, WM_KEYDOWN and WM_KEYUP.  When
WM_KEYDOWN or WM_KEYUP is received, the function DoKeydown or DoKeyUp is run.
*/
struct decodeWord messages[] = {
      WM_KEYDOWN, DoKeydown,
      WM_KEYUP, DoKeyup,
      WM_CREATE, DoCreate,
      WM_DESTROY, DoDestroy,} ;

/* Declare a character string that contains the name of the program.*/
char szAppName [] = "Invid";

/* Declare a variable called hInst that contains the data for a particular
instance of the program. */
HANDLE hInst;

/* Declare a variable called hWnd that contains a handle to the programs
window. */
HWND hWnd;

/* Declare an integer called AnimeStep.  AnimeStep is how many pixels a bitmap
should be moved during animation. */
int AnimeStep = 2;

/* Declare two variables of type BOOL (boolean).  That means that the
variables can be a 1 or a 0.  The variables fRightKeyState and fLeftKeyState
will be set by the functions DoKeyUp and DoKeyDown.  The variables will then
be used by other functions so that they can know the state of the right and
left cursor keys.  (I could have used the Windows function GetAsyncKeyState()
instead of waiting for a keypress message to be sent by Windows.  But I wanted
the program to be written in the standard Windows programming style of waiting
for a message.) */
BOOL  fRightKeyDown = FALSE, fLeftKeyDown = FALSE;
```

```
/* Decalaring a structure called BaseStruct that will contain everything about
the missile base.  I could have just as easily declared x, y, and Bmp as
separate variables.  Putting them into a structure groups related variables
under one title and makes it easy to remember what variable does what. */
struct BaseStruct {
/* Declare an integer called x that will contain the base's x coordinate */
 int x ;
/* Declare an integer called y that will contain the base's y coordinate */
 int y ;
/* Declare a variable called Bmp that contains a handle to the bitmap of the
missile base. */
 HBITMAP Bmp ;
 };

/* Declare a Structure called Base of type BaseStruct (defined above).  You
can now access the missile base's variables x, y and Bmp by using Base.x,
Base.y, and Base.Bmp. */
struct BaseStruct Base;


/*------------ WinMain ------------*/
int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance, LPSTR
lpszCmdParam, int nCmdShow)
{
  MSG msg ;
  hInst = hInstance;
  if (!hPrevInstance) InitApplication(hInstance);
  InitInstance(hInstance, nCmdShow);

/* This is the old message loop from the last program.
 while (GetMessage (&msg, NULL, 0, 0))
 {
  TranslateMessage (&msg) ;
  DispatchMessage (&msg) ;
 }
*/

/* This is the new message loop (from Petzold).  Instead of calling
GetMessage() to retrieve a message from Windows, this message loop calls
PeekMessage to see if there are any messages waiting.  If the waiting message
is a WM_QUIT message then break out of the loop and end our program.  If the
waiting message is not a WM_QUIT message then translate and dispatch the
message.  If there are no waiting messages then run our function DoInv().
This message loop will allow other Windows programs to run at the same time
but it will also keep idle messages from being generated.  We'll have to fix
it in a future article to make it more multitasking friendly.  */
while (TRUE)
 {
 if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
 {
  if (msg.message == WM_QUIT) break ;
  TranslateMessage (&msg) ;
  DispatchMessage (&msg) ; }
 }
 else
```

```
    {
    DoInv() ;
    }
  return (msg.wParam) ;
}

/*---------- InitApplication ---------*/
BOOL InitApplication(HANDLE hInstance)
{
  WNDCLASS wndclass;
  wndclass.style = 0 ;
  wndclass.lpfnWndProc = WndProc ;
  wndclass.cbClsExtra = 0 ;
  wndclass.cbWndExtra = 0 ;
  wndclass.hInstance = hInstance ;
  wndclass.hIcon = LoadIcon (hInstance, szAppName) ;
  wndclass.hCursor = LoadCursor (hInstance, IDC_ARROW);
  wndclass.hbrBackground = GetStockObject (BLACK_BRUSH);
  wndclass.lpszMenuName = NULL ;
  wndclass.lpszClassName = szAppName ;
  return(RegisterClass (&wndclass));
}

/*------------ InitInstance ----------*/
BOOL InitInstance(HANDLE hInstance, WORD nCmdShow)
{
  hWnd = CreateWindow (szAppName,
    "Invid",
    WS_MINIMIZEBOX | WS_POPUP | WS_CAPTION | WS_SYSMENU,
    CW_USEDEFAULT, CW_USEDEFAULT,
    WINDOW_WIDTH, WINDOW_HEIGHT ,
    NULL,
    NULL,
    hInstance,
    NULL) ;
  ShowWindow (hWnd, nCmdShow) ;
  UpdateWindow (hWnd) ;
  return (TRUE);
}

/*------------- WndProc ------------*/
long FAR PASCAL WndProc (HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
{
  int i;
  for(i=0; i < dim(messages); i++)
  {
  if(wMsg == messages[i].Code)
    return((*messages[i].Fxn)(hWnd, wMsg, wParam, lParam));
  }
  return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}

/*------------- DoCreate -----------*/
/* DoCreate() is called when our window receives a WM_CREATE message at
startup. */
```

```
LONG DoCreate(HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
{
/* Call a function called BaseInit that will initialize the variables in the
missile base structure Base. */
 BaseInit();
 return 0;
}

/*------------ BaseInit -----------*/
/* This function initializes the variables and loads the missile base's
bitmap. */
void BaseInit(void)
{
 Base.x = 40;
 Base.y = WINDOW_HEIGHT - 80;
 Base.Bmp = LoadBitmap(hInst, "base");
}

/*------------ DoDestroy ----------*/
/* DoDestroy is called when our program receives a WM_QUIT message */
LONG DoDestroy(HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
{
// Delete the handle to the missile base's bitmap
// before exiting. (otherwise Windows system
// resource's won't be released.
 DeleteObject(Base.Bmp);
 PostQuitMessage (0) ;
 return 0 ;
}

/*----------- DoKeyDown ---------*/
/* DoKeyDown is run whenever our program receives a WM_KEYDOWN message. */
LONG DoKeydown(HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
{
 switch (wParam) {
  case VK_RIGHT :
   fRightKeyDown = TRUE ;
   break  ;
  case VK_LEFT :
   fLeftKeyDown = TRUE ;
   break  ;
 }
  return 0 ;
}

/*---------- DoKeyUp ------------*/
/* DoKeyUp is run whenever our program receives a WM_KEYUP message.*/
LONG DoKeyUp(HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
{
 switch (wParam) {
  case VK_RIGHT :
   fRightKeyDown = FALSE ;
   break  ;
  case VK_LEFT :
   fLeftKeyDown = FALSE ;
```

```
    break  ;
  }
  return 0 ;
}

/*----------- DoInv -------------*/
/* This function runs whenever Windows isn't doing anything else */
void DoInv(void)
{
  BaseAnimation (hWnd);
  return ;
}

/*----------- BaseAnimation -------*/
/* This function draws the bitmap of the base at a new position based on
whether the cursor keys are pressed */
void BaseAnimation (void)
{
/* Move the x coordinate of the missile base based on which cursor key is
pressed. Base.x += AnimStep is shorthand forBase.x = Base.x + AnimeStep */
  if (fRightKeyDown) Base.x += AnimeStep;
  if (fLeftKeyDown) Base.x -= AnimeStep;
/* Keep the missile base on the screen */
  if (Base.x < 5)  Base.x = 5;
  if (Base.x > (WINDOW_WIDTH-50)) Base.x = WINDOW_WIDTH-50;
/* Another Petzold function slightly modified to make it simpler to use. (If
you havn't bought Programming Windows by Charles Petzold yet, stop reading
this and go to your local bookstore and buy it now!)  The function takes three
arguments.  The first argument is a handle to the bitmap that will be drawn.
The second and third parameters are the x and y position of the bitmap to be
drawn. */
  DrawBitmap (Base.Bmp, Base.x, Base.y);
}

/*----------- DrawBitmap ------------*/
/* This is where all the action takes place.  It's great for getting a bitmap
on your window as simply as possible.  Unfortunately it has performance
problems when you try and use it for all your graphics work. (This  might be
why Windows doesn't have this function built in)  */
void DrawBitmap (HBITMAP hBitmap, int xStart, int yStart)
{
/* Declare a handle to a device context.  A handle for a Device Context is
needed to draw to a window. */
  HDC hdc;
/* Declare a variable bm of type BITMAP.  It is used to store information
about a bitmap such as its size and colordepth. */
  BITMAP bm ;
/* Declare a handle to a device context.  This handle will be for a block of
memory that will be used as storage for the bitmap that will be drawn. */
  HDC hdcMem ;
/* Get a handle for a device context for our window. */
  hdc = GetDC(hWnd);
/* Get a handle for a device context to a block of memory.  The memory context
will be based on the device context of our Window. */
  hdcMem = CreateCompatibleDC (hdc) ;
```

```
/* Put the bitmap into the block of memory that was created above. */
  SelectObject (hdcMem, hBitmap) ;
/* Get information about the bitmap and put it in the structure bm */
  GetObject (hBitmap, sizeof (BITMAP), (LPSTR) &bm) ;
/* The BitBlt function copies the bitmap in memory (hdcMem) to the screen
(hdc) */
  BitBlt (hdc, xStart, yStart, bm.bmWidth, bm.bmHeight,
    hdcMem, 0, 0, SRCCOPY) ;
/* Free up the resources we used before leaving */
  DeleteDC (hdcMem) ;
  ReleaseDC (hWnd, hdc);
}
```


click here

When you run the program, it will put a small bitmap on the screen that moves back and forth when you press the cursor keys. If you're running it on a fast 486, you'll notice that it's too fast to be controllable in a game. Fortunately slowing down a program is always easier than speeding one up.

You now have the basics of an arcade game: Fast graphics with user interaction. In the next issue, we'll add the invaders and cover BitBlt'ing in detail.

I hope you enjoyed my second article. I spent a lot of time on formatting details for both the .hlp file and the .txt file. In the .hlp file I added bitmaps and linked bitmaps to run the sample programs while you're reading the .hlp file. I also spent a lot of time on the format of the source code with details such as using bold for the code and blue highlights for the start of functions. If you like this formatting, (or don't like it!) send me or the editors feedback.

About the author:

(short version)
SWM 25 ISO SWF. PC Compatible, no experience necessary.

(long version)
Bernard Andrys is a engineer at CSI, an ISDN networking development company. He holds a bachelor's degree in Mechanical Engineering from the University of Maryland at College Park. He can be reached through the Internet at andrys@csisdn.com or on the Windows Programming Journal BBS.

(longer version)
I was born in the house my father built. ...No wait, that's someone else. umm... Ok... I bet you're wondering what a Mechanical Engineer is doing working for a networking company and writing Windows program in his spare time. Well I'm wondering too. If you find out, write me. ...and another question. Is it the company I keep or do all programmers like The Hitchhiker's Guide to the Galaxy and Monty Python?