Windows(tm) Programmer's Journal

The Windows Programmer's Journal
Volume 01
Number 03
Mar 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:

HJ647C at GWUVM.BITNET or HJ647C at GWUVM.GWU.EDU

CompuServe: 71141 2071

GEnie: P.DAVIS5

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

Table of Contents

Windows Programmer's Journal Staff:

Publishers	**Pete and Mike**
Editor-in-Chief	**Pete Davis**
Managing Editor	**Mike Wallace**
Contributing Editor	**David Campbell**
Contributing Editor	**Andrew Bradnan**
Contributing Writer	Alex Federov
Contributing Writer	Andrew Bradnan
Contributing Writer	Jim Youngman
Contributing Writer	Todd Snoddy

WPJ.INI

by Pete Davis

Issue #3 wow... So far things have been going really well. The third issue is almost as big as the first and second put together! We've even got some extra articles for next month. We're getting a good response from the readers and the contributions are coming in. It seems like every day we get a letter from someone in another country. We've been getting mail from people in Romania, Czechoslovakia, Russia, Hong Kong, Australia, England, Germany, etc... In fact, the amount of mail we're getting is getting pretty high. We try to answer it all, but we don't always get a chance to. Also, some of the replies don't make it back. I've noticed this more often with certain Internet sights in the U.K. I don't know why that is, but the ones that have something like ac.uk in them don't seem to make it back. Sorry about that. The list goes on and on. Keep the mail coming, we love to get it.

I mentioned in the last issue about readers suggesting different formats for the text. So far, the biggest response has been in favor of the WINHELP format and the plain text format. That way you can read it on-screen and, if you want, print it out. Speaking of printing it out, we've had some negative responses about the printing format. I have to take responsibility for that. I had the page length a bit too long and a lot of you were getting pages with three or four lines of text. I'll try not to let that happen again.

As far as the Help format because we can do bitmaps now, it would be nice if we could get someone who is artistically inclined (Not us, as you'll see in this first issue) to give us a hand with some of the graphics. You don't have to be a Renoir, just better than us. If you can draw stick figures, you probably qualify.

So this issue we'll be starting with the WINHELP format. We hope you like it. We're pretty pleased with the idea. Right now we're using minimal tools for getting it into the help format, but we're looking into getting some better ones later on.

We'd like to thank the guys at America Online for giving us some time each month to get on and be able to stay in touch with our readers there. Mike will have more to say about all this in the Last Page. [See the "Special News" column - MW]

This month I'm going to be doing my article on printing that I should have done last month. Sorry for the delay, but we've just been real busy. I'm also going to do the third article in the series on the install program. This article is basically going to give some insight into the data structure that we're going to use for storing data about each of the files for the install.

This month David Campbell will be taking over the Beginners Column in C and we have Andrew Bradnan taking over the Beginners Column in C++. (We could still use someone to do the Beginner's Column in Pascal for Windows). This brings me to another point. Beginner's Column might be a bit of a misnomer. Yes, they'll all be basic, but like anything else, as time goes on, the articles will progress. They will all be, eventually, an intermediate programmer's column. There's only so much that you can say about the basics and eventually you have to move on.

We're starting a new feature this month called Hacker's Gash. Hacker's Gash is going to be a bunch of little tips and tricks that you can use in Windows. This is where the readers can really get in on the fun. Read through it, get an idea of what kinds of things we're doing and send in a list of your own tricks.

Ah good news, the BBS is finally up. Haven't replaced the hard drive, but I've done some serious work on it and did some pretty strenuous testing on it and it seems to be ok for now. I will need to replace it eventually, but we don't have the money right now for that. Besides being the first place to get the Windows Programmer's Journal, the WPJ BBS has the SIMTEL20 CD-ROM which has 600+ megs of public domain and shareware software. We will try to add a second CD-ROM in the near future and add some new disks. We'll probably try to add the CICA disk next, which has a very large library of Windows public domain and shareware software. The number's around here somewhere, but to make it easier, and as long as you're reading here, it's (703) 503-3021. (That's in the U.S. for you guys overseas.)

Mike and I have debated doing this but we're really getting to the point where we don't have much choice. Seeing as we're going into the Windows Help format, which will allow us to support graphics, we've decided that we're going to start allowing some advertisers in the magazine. We're going to keep the ads to a minimum, but, even though the magazine is free, there are some costs involved in putting it together and we really need some reimbursement for it. Right now, our Compuserve and Genie bills alone are costing us a fair amount. Running the BBS will cost us $25/month in phone bills alone, not to mention possible hardware maintenance. We're hoping to get a few advertisers to help offset some of those costs. The ads will be, most likely, from shareware authors. If you are a Windows shareware author and you're interested in advertising, get in touch with us and we'll discuss the costs, size, design, etc.

One last thing before I wrap up. We've gotten a lot of comments about the format of the magazine. Most of these were in reference to the format we're going to distribute it in. If you have preferences about the layout or other stylistic concerns, we'd like to hear them.

That's really about it for this month I guess. We just wanted to thank all of you for reading and to keep the comments and suggestions coming. We'd also like to thank those of you who have submitted articles and we ask that you please keep the articles coming in. It would be nice if we got a few more submissions each month so we could get the magazine up to the size that we'd really like. I think 50 pages would be a good size, although, with something like this, I suppose, the bigger the better, so more than 50 pages would be fantastic. Well, I've said enough. Enjoy the magazine.

_Pete Davis

P.S. I just read Mike's Last Page (He does it last so I didn't see it when I was writing the WPJ.INI) You might want to read that and then come back to this, but I wanted to give my own thoughts on what Mike said regarding the WPJ as a source of information. We are not here to be a sole source of information, nor will we always be correct. We do our best to make sure that the information we give out is correct. The only people who check the submissions are Mike and myself. Neither one of us has a PhD in Windows Programming. We both make mistakes. The WPJ is going to have errors in it. We're going to give out completely wrong information at times. When that happens, we will try to correct ourselves by the next issue (usually in response to someone saying "You guys screwed up!").

What I'm saying is that there are a lot of sources out there. You need to check out as many as you can. Cross- reference the information between different books and magazines and when you see an inconsistency, that's probably a fault.

Mike and I have several reasons for doing the WPJ. First of all, we felt beginners weren't being addressed well enough. Also, we wanted to cover as many Windows programming

languages and environments as possible. (We're getting there, but slowly.) Most important, I think, we wanted to address Windows programming and Windows programming only.

Windows is an enormous programming environment and there's tons of information to absorb. No one publication can even approach covering it all. I like to think that one of our advantages is that, unlike a lot of other programming magazines, we don't center on a topic each month. Other magazines might have, say, an issue on Custom Controls or an issue on Multi-media. Well, that's great, but what about those of us who don't work with Custom Controls or Multi-media? There are two issues that we don't care for. We try to keep a variety of topics each month so we can appeal to as many people as possible with each issue, but I digress.

To wrap up I just want to say, (this feels like deja vu, I'll check later, but I think I said this in the last issue also) don't count on us being right every time. If something we wrote about doesn't work, try it again. If it still doesn't work, we probably screwed up. Let us know, and we'll try to correct it by the next issue. I know, I'm getting repetitive. I know, I'm getting repetitive. Until the next issue, peace.

Letters

Date: 10-Feb-93 07:04 EST
From: Chris Newham [100027,16]
Subj: Windows Programmer's Journal
Hi,

I would just like to say that I have enjoyed the two issues of WPJ so far. You can add at least 1 to your readership numbers as my friend and I download it once between us.

The sections on DLLs & install progs are of particular interest to us as we have just finished work on a DLL and are now working on the install prog. A point that I think you might mention and give some consideration to for the install prog is this:

If you are installing DLLs then you need to check if the file exists first (easy); if it exists you then need to check its version control information to determine if the copy you wish to install is newer (also quite easy); the difficult part comes when you have determined that you need to replace the existing DLL but the DLL is already in use by Windows.

Windows will not let you delete or replace the file. Microsoft's install program works around this but we haven't yet worked out how; we think that it copies the new DLL to a temp dir or hides it somewhere then replaces it when Windows next starts up.

It's an interesting little problem and has at present got us stumped. If you know how to solve it let's see it published. If we find a solution we will let you know. [We're working on this problem, too. - MW]

I am very impressed by the quality of the journal and would like to see it remain in either text or Windows write format as my time on the PC is limited and so what I do is take a hard copy of the journal to work to read at lunch time so Winhelp format would not suit me.

Keep up the good work Best wishes Chris.

Date: 12-Feb-93 17:29 EST
From: Tammy Steele
Subj: WPJ

Hi Mike

I just downloaded my mail today. I generally only read it once a month. So, sorry for the delay in my comments about WPJ. I had a chance to skim the first issue and have just spent some time reading through the second issue. Of course I cannot make official "Microsoft" comments, but I can give you my opinion.

Generally speaking I think the WPJ will be another good place for us to point people to for information. One problem will be getting the Microsoft support engineers familiar with the content of the journal. Another problem is the accuracy of the articles. In order for Microsoft to point people to the articles, we need review the accuracy of them. We are pretty swamped now with too much to do, not enough people. I am sending mail to my group about the journal. So we'll see what happens.

Specifically I thought it was good that you discussed the feedback you received from the first journal. Especially the Linked List sample. If the sample were used for a large number of nodes, as I'm sure someone probably mentioned, it would not be a "cooperative"

windows app because each globalalloc eats a selector and the selectors (8K of them) are shared between *all* windows apps and the Windows system itself.

Also it would be useful for samples to be updated to the current software although there is still value to having the code and comments (ie, the DLL article.)

People who submit articles should check the MSKB and if they have areas that they are uncertain about, should post questions in the forum before they submit articles. For example, the DLL article talks about WEPS and says something like "the documentation says the WEP is called once...but I haven't seen it be called in Codeview in Windows 3.0." There is an article that discusses the reason why you can't see it be called in Codeview under Windows 3.0 and/or this question could have been easily answered in the DLL section on WINSDK. [This question is answered farther down in this column. - MW]

I like the coding style and the sample makefiles (the makefiles are easy to read.)

I'm sure you've had lots of comments as you mentioned, about how to format this. I personally think you should offer it in helpfile format *and* text format so that people have the option. [editors note: This seems to be the most common opinion.]

Overall I see this journal as a really great place for people to compile information and share it.
Thanks for all your work,
Tammy

Editor's Note: We got a letter from Craig Derouen (CompuServe ID: 75136, 1261) of Seattle. I'll reprint his letter here and let him explain his BBS:

I've looked at your 2 first issues of WPJ and am impressed. I run an extensive programmer's BBS out here in Seattle, primarily Windows code but also C, C++ and 8086 code for DOS. I carry both your issues. I am also in Fidonet and have all the listings for WPJ available for Frequest. Also I am a support BBS for WinTech,Windows/DOS,CUJ and PC Techniques Magazine. I have a LOT of code on line. Anyways here's the details:
Cornerstone BBS (2400,9600 baud)
(206) 362-4283
Fidonet 1:343/22

All magazine listings are available for first time callers; free downloads and some file requests are also available.

Editor's Note: In our last issue we had an article by Rod Haxton on DLLs. In the article, he mentioned that he had never seen WEP get called under CodeView in Windows 3.0. So, I posed the question to the WINSDK forum on CompuServe and got a response from Brian Scott of Microsoft. Thanks, Brian.
Mike,

If you implicitly link to a DLL in Windows 3.0, the WEP is called after the application has unloaded. Since Codeview stops debugging after the application terminates, it does not see the WEP get called. If you need to debug the WEP, you can load it using LoadLibrary() and free it using FreeLibrary(), instead of linking to the import library. In this case, you will see the WEP get called when you call FreeLibrary(). If you need to link to the DLL using an import library, then you can debug the WEP using something like WDEB386, putting

OutputDebugString() statements in your WEP, or moving the code in the WEP into a cleanup function that you call just before the application terminates.

Hope this helps
Brian Scott - Microsoft

Editor's Note: Pete and I started a beginner's column with the first issue, and Dave Campbell offered last month to take it over. Dave is a confirmed Windows hacker and so will be writing this column starting with this issue. We hope you like it. Questions should be directed to Dave. Means of reaching him are given at the end of his column.

Beginner's Column

By Dave Campbell

My name is Dave Campbell and I write Windows software under the business name WynApse. I am going to be writing the beginner's column until I get buried in work (wouldn't that be too bad?) or I get usurped by someone who wants to do this more than I do.

Up until recently I have been a Borland programmer, so the tools I am most familiar with are the Borland C++ suite, 3.0 and 3.1, in particular. Pete and Mike's previous two columns have been Microsoft, and I am leaning that way myself right now, so I will continue that. My intention is to provide code and ideas that aren't tied to Borland or Microsoft, but could be used with either compiler. A major difference between the two is make files, so I will stay away from those as much as possible.

My intention is to write code that is useable on 3.0/3.1 Windows. If enough questions come in for 3.1- specific applications, we will cover them. This leads directly into the request for requests. I would like to present code that is useful to all the readers, and the best way would be to be responsive to readers questions. Please send questions! I will list various ways of reaching me at the end of each article.

Now on to the fun stuff...Mike and Pete have been working on a Hello World program for a few issues, and I am going to add some configuration to that code and demonstrate some basic uses for radio buttons and INI files.

Radio Buttons

A radio button is one of several control types available to Windows programmers inside dialog boxes. They work similar to the buttons on your car radio...only one may be pressed at a time, at least in this application. There are other controls available in the basic set, and we will look into them later. For now, let's look a little deeper into the declaration of a radio button.

Radio button declarations are made in the .RC file, for example:

```
CONTROL "Hello", IDM_HELLO, "BUTTON", BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 20, 15, 42, 12
```

We will use one more button in our dialog box, and that is a pushbutton. The ever-present OK button to be precise. The control is a graphical representation of a 3D button with text written on it. By making a pushbutton an "OK button", we are really painting the word OK on the face of a 3D pushbutton. The button definition we are going to use is:

```
CONTROL "OK", IDOK, "BUTTON", BS_DEFPUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 29, 57, 24, 14
```

The explanation of the fields in these two declarations is covered later in this article.

INI files

INI files are used by Windows to control the environment under which Windows executes. Windows programs use INI files to manage the individual environmental changes available to users to customize the system to his needs. There are two ways to read a variable from an INI file, but only one way to write information into one:

Reading:

ReadPrivateProfileString
ReadPrivateProfileInt

Writing:

WritePrivateProfileString

This does seem clunky but it's what we've got to work with. When we get to that point, I'll demonstrate that it's not that big of a problem.

Windows handles the INI files very nicely in that you will always get something for your efforts. In the Read calls, a default value is given, in case the variable does not exist in the INI file (or the INI file doesn't exist, for that matter). In the Write call, if the variable doesn't exist (or even if the INI file doesn't exist) prior to the call, it will when the call returns. This means that you needn't be concerned about the pre-existence of an INI file before trying to use one.

Let's consider an INI file for our application, HELLO.INI:

```
[Hello]
Setup=0
```

This is the simplest form of an INI file: there is a single 'Application Name', [Hello], and a single 'Keyname', Setup, that is associated with it. The value for Setup is 0. During execution of our program, if we had a need for picking up the Setup value, we would simply use the line:

```
nDflt = ReadPrivateProfileInt("Hello", "Setup", 0, "hello.ini");
```

where nDflt is defined:

```
int nDflt;
```

This reads from hello.ini looking for the 'Setup' keyword under the 'Hello' application name, and assigning the value equivalence listed there, or 0 for default, to the variable nDflt.

If however, the INI file was defined as:

```
[Hello]
Setup=Hello World
```

you now cannot read the file with ReadPrivateProfileInt. Now the line to read the value is:

```
char szDflt[25];

ReadPrivateProfileString("Hello"  "Setup", szDflt, "Goodbye World", 13, "hello.ini");
```

This has some similarities and some differences from the one above. The "Hello", "Setup", and "hello.ini" are self-explanatory, but the string read also adds other parameters. The string name to store the result into is listed as the third parameter, in this case szDflt. The fifth parameter is an integer variable declaring the maximum number of characters accepted from the INI file, and the fourth parameter, in this case "Goodbye World", is the default value. If this line were to be executed and the INI file not even exist, szDflt would contain the string "Goodbye World". This is also the case if the file exists but does not contain either [Hello] or Setup, or both.

To change an INI file the WritePrivateProfileString function must be called:

```
char szDflt[25];
 int   nDflt;

wsprintf(szDflt   "%d", nDflt);
 WritePrivateProfileString("Hello", "Setup", (LPSTR) szDflt, "hello.ini");
```

'wsprintf' should be used in place of 'sprintf' because it is already in Windows, and will not cause the linking to another library. The downside is the need to cast the string as a LPSTR. wsprintf will build a string, in this case, containing the ASCII representation of the single integer nDflt. That string is then passed to the INI file using the syntax shown. If the Setup variable were a string, the variable to be inserted would be given in the WritePrivateProfileString call rather than using the intermediate wsprintf step.

Whew I'm glad that's over.

Dialog Box

The dialog box we're going to display will have two radio buttons, and an OK button. Typically, the buttons to complete a dialog box are located either along the bottom, or if the dialog is very busy, they can be placed along the right edge. Of course, this has nothing to do with programming Windows. This is all aesthetics and being kind to users. Look at a thousand Windows applications, and you'll get used to seeing things a certain way. Users get used to them being that way, and you will lose some people just with your user interface, or lack thereof.

The code I propose for the dialog box is:

```
Hello DIALOG 63   56, 83, 77
 STYLE WS_POPUP | WS_CAPTION
 CAPTION "Hello Setup"
 FONT 8, "Helv"
 BEGIN
 CONTROL "Hello",      IDM_HELLO,     "BUTTON", BS_AUTORADIOBUTTON | WS_CHILD |
WS_VISIBLE | WS_TABSTOP, 20, 15, 42, 12
 CONTROL "Goodbye",   IDM_GOODBYE, "BUTTON", BS_AUTORADIOBUTTON | WS_CHILD |
WS_VISIBLE | WS_TABSTOP, 20, 28, 42, 12
 CONTROL "OK", IDOK, "BUTTON", BS_DEFPUSHBUTTON | WS_CHILD | WS_VISIBLE |
WS_TABSTOP, 29, 57, 24, 14
 END
```

This code can be produced either by hand with any text editor, or by graphically placing the objects with Borland's Resource Workshop or some similar tool. I usually start with the Resource Workshop, and make small tweeks by hand. Large changes are best done graphically to ensure the placement, and logical usefulness of the dialog box is alright.

I am going to take a break right here and talk about coding style. This is something that is as peculiar to a person as their name. Everybody has their own, and I wouldn't think of trying to force my ideas on someone. But...(have you ever noticed there's always a big But around somewhere?), if you don't have your mind made up yet about those long Windows lines like the 100+ character ones above, please split them onto succeeding lines! Line-wrap on listings is UGLY!! OK, that's over, I've got it out of my system. But (another one), since I am writing this, you are going to have to put up with my style...

I'm done now. That short dissertation was free, now back to the program. The field explanations are as follows:

```
Hello DIALOG 63   56, 83, 77
```

This line defines the name "Hello" of the dialog, and the coordinates of the box relative to the client area of the parent window. The units are NOT pixels, and beyond that, I don't want to get into it now. Just play with it for now until you get it where you like it. The first two numbers are the x,y coordinates, and the second two are the width and height of the dialog.

STYLE WS_POPUP | WS_CAPTION

The STYLE line defines the manner in which the dialog box is built. WS_POPUP is pretty standard for dialog boxes. There are real differences between POPUP windows and OVERLAPPED windows, the main one being the CAPTION on a POPUP is an option. Because we are going to call this as a modal dialog box, we are going to give it a caption bar to allow it to be moved. 'Modal' dialog boxes, as opposed to 'modeless', disable the parent window, and demand attention until closed. Without the caption bar, the box cannot be moved.

CAPTION "Hello Setup"

The CAPTION line declares the caption used with the WS_CAPTION style parameter.

FONT 8 "Helv"

This defines the default font used throughout the dialog box. You could pick any conceivable font here, but it is best to be kind and only use those shipped to Windows users, or you are going to get some nasty mail messages.

BEGIN

BEGIN..END or .. define the body of the dialog box.

```
CONTROL "Hello"       IDM_HELLO,    "BUTTON", BS_AUTORADIOBUTTON | WS_CHILD |
WS_VISIBLE | WS_TABSTOP, 20, 15, 42, 12
 CONTROL "Goodbye",   IDM_GOODBYE, "BUTTON", BS_AUTORADIOBUTTON | WS_CHILD |
WS_VISIBLE | WS_TABSTOP, 20, 28, 42, 12
```

The two radio button definitions following the declaration CONTROL are:

- The text of the control in our case "Hello" or "Goodbye"

- The ID value to be returned to the parent window, IDM_HELLO or IDM_GOODBYE are defined in our ".H" file.

- BUTTON declares the control class. Buttons are generally small child windows.

- BS stands for BUTTON STYLE and BS_AUTORADIOBUTTON declares the buttons will only be pressed one at a time, and the button is automatically checked.

- WS_CHILD declares the dialog box as a child window. This means it resides within the boundaries of the parent window.

- WS_VISIBLE applies to overlapped and popup windows. The initial condition is visible.

- WS_TABSTOP specifies that the user may step through the control sequence with the tab key.

- 20 28, 42, 12 are the coordinates and size of the control, similar to that of the dialog box itself.

```
CONTROL "OK"   IDOK, "BUTTON",
 BS_DEFPUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 29, 57, 24, 14
```

The OK button declaration: everything should be self-explanatory except the BS_DEFPUSHBUTTON style. DEFPUSHBUTTON means that this is the default PUSHBUTTON. Pushbuttons are: OK, CANCEL, ABORT, RETRY, IGNORE, YES, NO, and HELP. Only one may be the default, and it has a dark border around it, so that if the Enter key is pressed, that is the one you get. IDOK is defined in windows.h, so we don't have to declare it.

The user will get this dialog box on the screen, and select one of the radio buttons, or toggle them back and forth a few times, then select OK. The dialog box procedure should read the INI file and preset the proper radio button to show the user the current value. Upon selecting OK, the procedure will have to set the selected information into the INI file.

That completes the dialog box procedure discussion. The code follows:

```
BOOL FAR PASCAL HelloDlgProc (HWND hDlg   WORD message, WORD wParam, LONG
lParam)

  switch (message)

        case WM_INITDIALOG :
              CheckRadioButton(hDlg   IDM_HELLO, IDM_GOODBYE, InitSettings);
              return TRUE;

        case WM_COMMAND :
              switch (wParam)

                    case IDM_HELLO :
                          WritePrivateProfileString("Hello" "Setup", "1", "Hello.ini");
                               InitSettings = wParam;
                               break;
                    case IDM_GOODBYE :
                          WritePrivateProfileString("Hello" "Setup", "0", Hello.ini");
                               InitSettings = wParam;
                               break;
                    case IDOK :
                               EndDialog(hDlg   wParam);
                               return TRUE;

             break;

  return FALSE;
        /* HelloDlgProc */
```

Notice the WM_INITDIALOG call:

```
CheckRadioButton(hDlg   IDM_HELLO, IDM_GOODBYE, InitSettings);
```

An assumption is being made here that the variable InitSettings has been read into our program somewhere, and is set to (in our case) either IDM_HELLO or IDM_GOODBYE. CheckRadioButton uses the dialog box handle hDlg to identify a numerical sequence of buttons from IDM_HELLO to IDM_GOODBYE, and ensures that only InitSettings is set.

More advanced Windows programmers will probably want to skip the

```
   case IDM_HELLO
.
case IDM_GOODBYE
```

statements altogether and wait until OK is pressed to check the state of the buttons. That is fine, and more streamlined, but for now, let's not leave anyone behind. I want all of us to learn this stuff.

Let's change one more function or else the whole idea of the dialog box selecting radio buttons is wasted. Let's read the INI file in WndProc, and change the text displayed based upon the Select value.

This is going to take two steps just like the two parts of the previous sentence:

```
    case WM_CREATE :
         InitSettings = GetPrivateProfileInt("Hello" "Setup", 1, "Hello.ini");
         InitSettings = InitSettings ? IDM_HELLO : IDM_GOODBYE;
         hdc = GetDC(hWnd);
         InvalidateRect(hWnd   NULL, TRUE);
         ReleaseDC(hWnd   hdc);
```

and:

```
/*---------------------------------------------------------------------------
 fall through to WM_PAINT...

---------------------------------------------------------------------------*/
   case WM_PAINT :
           hdc = BeginPaint(hWnd &ps);           /* returns pointer to hdc */
           GetClientRect(hWnd   &rect);

/*
 -1 tells the DrawText function to calculate length of string based on     NULL-termination
 */
        DrawText(hdc   (InitSettings == IDM_HELLO) ? "Hello Windows!" : "Goodbye
Windows!", -1, &rect, DT_SINGLELINE | DT_CENTER | DT_VCENTER);
      EndPaint(hWnd &ps);
 return 0;
```

The WM_CREATE case reads the INI file to find out which we want printed and sets up the 'InitSettings' variable used in the dialog box. Then we get a "handle to a device context" or hdc for our window.

A device context is the area into which we "paint" text in a window. In a dialog box, we can use wsprint, but in a window, we have to "DrawText", and we draw it into a device context. A device context could be a window or a printer page, Windows doesn't care. At this point, we are getting the device context for our main window's client area. We then report to Windows that the entire area is invalid, which will set us up for a repaint.

In handling the WM_PAINT case we again need an hdc, and this time do it with a call to BeginPaint, passing a pointer to a structure variable of type PAINTSTRUCT. BeginPaint has Windows fill in the ps for us, and is then available for our use. We aren't going to use it, however.

We call GetClientRect to get the dimensions of the client area into the rect structure.

DrawText uses the hdc the rect structure, and the InitSettings value to decide what to paint and where. Ultimately, either "Hello Windows!", or "Goodbye Windows!" is printed on a single line, centered horizontally and vertically: DT_SINGLELINE | DT_CENTER | DT_VCENTER. Notice the note above the DrawText line. Instead of telling Windows how many characters we are painting, let Windows do it for us!

EndPaint closes the ps structure and finishes our WM_PAINT case.

The Setup dialog

I almost forgot we do need to get the dialog box onto the screen. I have added three lines to the hello.c file:

```
hMenu = GetSystemMenu(hWndMain  FALSE);
 AppendMenu(hMenu, MF_SEPARATOR, 0, NULL);
 AppendMenu(hMenu, MF_STRING,     IDM_SETUP, "Setup...");
```

These get a handle to the system menu of the window, and insert a menu separator followed by the word "Setup...". When "Setup..." is chosen, the value IDM_SETUP is sent to our windows message loop:

```
      case WM_SYSCOMMAND :
            switch (wParam)

                  case IDM_SETUP :
                     lpfnHelloDlgProc = MakeProcInstance(HelloDlgProc, hInst);
                     DialogBox(hInst  "Hello", hWnd, lpfnHelloDlgProc);
                     FreeProcInstance(lpfnHelloDlgProc);
                     return 0;

           break;
```

This is handled as WM_SYSCOMMAND because the system menu is the one used.

We must get a long pointer to a function lpfnHelloDlgProc, to use in the DialogBox function call. The parameter "Hello" in the call is the title of the dialog box we built. Because of the 'DialogBox' call, this dialog box will be modal, and control will not return to the main window until OK is pressed in the dialog box.

Don't forget the classical Windows programmer's bug...any ideas?? Ok, since nobody is raising their hand...it's exporting the dialog box in the .def file. I have forgotten this so many times, I hesitate to admit it. If you don't list the dialog box in the .def file, it is guaranteed not to work in Windows 3.0, and will be extremely unreliable in 3.1. The next time you get an unreliable dialog box, remember the "classical Windows programmer's bug".

The files are archived with the magazine. I have made one other excursion, and that is the dialog box code is in a separate file named HELLO.DLG. The file HELLO.RC contains the following line to include the .DLG file:

```
rcinclude Hello.dlg
```

This is pretty standard among Windows programmers, because it keeps the dialog box code in its own file.

3.0/3.1/Borland

I just checked and found that I had produced great 3.1 code, but the thing wouldn't run under 3.0. If you are programming for the general public, you better keep at least a 'virgin' copy of 3.0 around on your hard disk for this sort of checking. Users tend to get a little touchy about that sort of thing. The fix is really easy for Microsoft people. The problem lies in the Resource Compilation stage. If you type 'RC -?' at the command line, you will see a '-30' switch listed. This is what will get you 3.0+ executables. That is the way I have it in the make file.

Borland 3.1 programmers have a couple more lines. In addition to the RC file change, you must also add the line:

```
#define WINVER 0x0300
```

ahead of your include of windows.h in your hello.c file. Also you must change the

```
wc.lpfnWndProc     = WndProc;                    /* Name of proc to handle window */
```

line to be:

```
wc.lpfnWndProc     = (WNDPROC)WndProc;       /* Name of proc to handle window */
```

I may have left off some stuff here if so, let me know about it. We're all in this thing together.

Please hang in there. If you are beyond the scope of this article, stick with me we are going to go places together. If you are way beyond this, write us an article. If you are bogged down, just compile it, and stare at the source. If you want help with something, send me a note.

That's it for this time. Next month I plan on building an About box with live compile date inserted, and I'll discuss Icons and Menus. In coming issues my intention is to discuss File Open boxes, Help files, Dialog Boxes as main windows, Obscure Dialog Box uses, Timers, and debugging. Feel free to contact me in any of the ways below. I want to rat out the things other people are having questions about, not just what I think people want to hear.

Dave Campbell WynApse PO Box 86247 Phoenix AZ 85080-6247

(602)863-0411 --- CIS: 72251 445

Phoenix ACM BBS (602) 970-0474 - WynApse SoftWare forum

Install Program Part III:
The SETUP.INF File

by Pete Davis

Before I get started this month there are a couple of things I wanted to talk about. First of all, because of some stuff coming up in the near future, I won't be able to do Part IV of the install program next month. It will, however, continue in May. Sorry about this, but Part IV is going to be a big one and I'm not going to have the time to do it yet.

I'd also like to respond to Chris Newham's letter (found, oddly enough, in the Letters section) regarding installing DLLs which are currently active. I have to admit I haven't yet tried this, but plan, by Part IV to have a solution. I went through Microsoft's code for their install program and I couldn't find anything that seemed to make exceptions for DLLs. This leads me to believe that the solution is something fairly simple. If worst comes to worst, you could always just find out who's using it and shut them down. I doubt this is a very good way to do it, but it's just a thought. Like I said, I'm going to look into it and I can hopefully have a solution to it by Part IV. (I better have a solution by then, 'cause Part IV is going to handle copying the files over.)

Ok this one's going to be short and sweet. There's not much to it. We're using a SETUP.INF file which is going to tell us what the name of our application is, how big it is, what files have to be installed, what disks they're on, whether or not they're executables, etc. I had two options for doing this. I could have used the GetPrivateProfileString and GetPrivateProfileInt functions and make it into a .INI file, but I wanted to maintain Windows 2.0 compatibility. (Just kidding :-) Actually, one reason I didn't is because I didn't think of it until it was too late. Actually, there are some problems with that approach. The problem is that we're dealing with multiple files and they're going to have the same entry names. I'm sure you understand completely now, right? Ok, here's an example of a SETUP.INF file and then I'll explain it again.

```
; Semicolons as is typical for these kinds of files, mean comments follow
 ; and the line is ignored.
 Application=My Application
 AppSize=1204558
 DefaultDir=
 ;
 ; Now we'll have information about each file
 ;
 ; CompName = Name of file compressed on install disk
 ; UCompName = Name of the file when we uncompress it.
 ; FileType = 0 - EXE    1 - DLL    2 - Other (Other is the default)
 ; FileSize = Uncompressed file size
 ; AddDir = Sub-directory name if it's a sub-dir of the DefaultDir
 ; (i.e. AddDir=would mean the file is in
 CompName=MYAPP.EX0
 UCompName=MYAPP.EXE
 FileType=0
 FileSize=10294
 CompName=DATA1.DA0
 UCompName=DATA1.DAt
 FileType=2
 AddDir=
```

Ok that should be enough for a sample. Now our code is going to start a separate node in our linked list of files to install each time it hits a CompName= statement. This is harder to do with the GetPrivateProfile... functions. We could throw in a new section like

[FILE1] for the first file, [FILE2] for the second, etc. But back to the topic, we're not doing it that way. I just wanted to give you ideas of how it could be done if you choose to do it that way.

All right well, that was all simple enough. Did I mention linked list in the last paragraph? Yup, that nasty phrase!!! Ok, it's pretty simple linked list. To make it easier, it's essentially a stack, so each time we get a new file, we just add it to the front of the list. The code is in the READSET.C file and the FILES.H file. I've commented it pretty heavily, so I won't go in depth here. It's all very simple.

In Part IV which will be in May, we're going to do a lot of the real work. Like I said, it's going to be a big one, and we're going to be tying in all the stuff from the first three parts. I might have to finish the entire thing in June, just because what's left is so big. I've tried to break this series up into easily recognizable parts, First I had the DDE with Program Manager, then I had the LZEXPAND.DLL part, and this month we had the READSET stuff. I feel like what's left all goes in the category of 'the rest of it', but there's so much, that I'll have to break it up into two hard do divide sections. I'll probably just do a few of the small things in each one, like the Progress Bar, creating directories, copying files, etc...

Oh well that's it for this month. Sorry it was so short and sorry I can't do it next month. If I can, I might try to do all of the rest in May to make up for not doing it at all in April. We'll see. Until next time.....

[Editor's Note: Last month, I started a Beginner's column on C++. Andrew Bradnan was kind enough to offer to take over the column, and will be writing it starting with this issue. Any questions you have can be directed to him. His CompuServe ID is given at the end of his column. - MW]

Home Cooking - C++ from Scratch

by Andrew Bradnan

This month I'll be starting a new column WPJ will be doing every month to introduce people to C++ and Windows. I am going to try and keep this simple enough for a programmer new to Windows programming and C++. Your brain may melt if you are at this stage, but give it a try and it will sink in after a while. Feel free to send me some email on Compuserve [70204,63]. I'll answer any questions and put the good ones at the end of next month's article.

I've read many books on C++ and I considered all but a few totally confusing. (I don't plan on adding my name to this list.) There is another rumor that C++ is really slow because it writes all sorts of code behind your back. This is kind of like complaining that C writes all kind of assembly behind your back. The compiler is supposed to write code for you. The great thing about C++ is that you can forget many of the details. I usually forget them all by myself. Instead of me rambling how great C++ is, let's find an example of exactly how we can forget some things. On purpose. The simplest place to start is constructors and destructors. Of all the confusing C++ terms (abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence), constructors fall into the abstraction and encapsulation category. More later. Since we almost always want to initialize our data structures (objects) to some known state, C++ will automatically call your own constructor every time. The converse is also true. When you delete an object, or it goes out of scope (falls out of a curly braces), you also want to free up memory, and set your variables to a used state. Where in the bleep do they dome from? You only have to do two things. First declare your member functions, and then define them. Piece of cake. For an example, Windows is kind enough to require us to create and destroy all sorts of stuff. There are over 63 calls to create something in the Windows API. All these objects (bitmaps, etc.) have to be closed, deleted, freed, or destroyed (pick your favorite synonym). Since we almost always trap the WM_CREATE message lets create a PAINTSTRUCT object. We'll call it a PAINT. If you want to put anything on the screen, you have to ask permission. Since Windows is kind enough to let many programs run at once, they have to share the screen. Kind of like kindergarten, and Windows is the teacher. To get permission you have to ask for a display context (think constructor). Your piece of the screen. A display context is provided when we call BeginPaint(...). Due to the limitations of DOS and/or Windows you can only have five DC in use at once. So after you are done you have to release the DC so that another program can draw in his corner of the sand box (think destructor). This is done with EndPaint(...). BeginPaint (...) also fills in a PAINTSTRUCT data structure. The only additional information filled in is whether you should paint the background, and the smallest rectangle you have to paint. In my tradition, let's write some code that uses our PAINT object. This will clarify what advantages we'll gain and what member functions we are going to have to write.

```
/*
 PAINT Object Test
 Written by Andrew Bradnan (c) 1993
 Note: The only C++ code is in the WM_PAINT case statement and in PAINT.H */
```

[Editors Note: There was code here. Because of problems in generating the help file, it was removed, but is in the plain text version of the magazine (WPJV1N3.TXT). We are investigating the problem and will try to prevent it from happening in the future.]

As you can see we have gone from four lines of code to two. Not too bad. The interesting part is that we no longer have to call BeginPaint or EndPaint, and we no longer care what those pesky PAINTSTRUCT members are. The compiler grabs it from the PAINT object for us, calling our declared explicit cast operators. There is also sort of an intentional bug. If you cover up only part of the client window and then bring the sample application back to the foreground, "PAINT Test!" doesn't draw in the middle of the client window. "PAINT Test!" will draw in the middle of the rectangle that was covered up. Windows is kind enough to let you do a little optimization should you want to. From the sample code above we can determine that we need a PAINT object with four member functions. One constructor, a destructor, a cast to an HDC, and a cast to a LPRECT.

[Editors Note: More code belongs here. See the above Editor's note for an explanation.]

As you can see strangely enough all the code is written in the header file. This is called inlining. The short story is that this allows the compiler to optimize your source code. It does this by replacing the function call with the code you have written. So our call to DrawText () really won't call two functions to get the parameters it will just reference the members in our PAINTSTRUCT ps. It is essentially like using a macro except you get all the type checking thrown in for free. Some statement will not inline but your compiler will let you know what these are. You will also note that the constructor looks a little weird. In the constructor, we can optionally tell the compiler how to initialize our members hWnd and ps. If we do not the compiler will create them, set all the members to zero, and then we would initialize the member within the curly braces. Obviously one more step than you want. To initialize hWnd we use the parameter passed in. Memory for hWnd is allocated and then filled with the parameter passed in. Which is exactly what we wanted. Space for ps is allocated, set to zero, and then we initialize it, using BeginPaint, in the function body. An extra step but it can't be avoided in this case. So the moral of the story is that C++ code can be quite easy to read when you are using the object. Writing the actual code is a little messier. Just remember you only have to get it right once. You can forget two function calls and three PAINTSTRUCT member names. You can even forget about PAINTSTRUCT. Readers familiar with some of the brand name applications frameworks may be wondering why I did not add DrawText() as a member function to our PAINT object. Doing this we could just remove the two cast operators and call DrawText with new parameters - DrawText (LPSTR szText, int cb, UINT fuFormat). The only problem is we have to learn more than we forget. Now you have two versions of DrawText(). This is supposed to be easier not more complicated. The second reason is that you can't DrawText a PAINT. English wise, this makes no sense. You can draw some text on a DC. That would make sense. You would also end up with every drawing function listed under the PAINT and/or DC object. Definitely confusing. Just look at Microsoft's AFX. Yuck! I have built the above example using BCW 3.1. With minimal changes it ought to work fine with MSC 7.0 and with Windows NT. Next month I'll start a little earlier so I can test it on all three platforms and send the appropriate make files to make your life easier. Again if you have any problems, questions, suggestions, or answers send me a note on Compuserve [70204,63].

Andrew Bradnan is president of Erudite Software, Inc. Their latest offering, which he wrote using C++, is called Noise for Windows, a sound utility for Windows 3.1. Feel free to contact him for information.

Creating And Using Owner Draw Buttons

By Todd Snoddy

Many of you may be wondering how some Windows programs display the fancy bitmapped buttons. Although these are not standard controls, Windows does provide the capability to utilize controls that are custom designed. Many times a properly displayed graphic can mean much more than a simple text word. I will try to explain how you can use your own custom buttons in your programs.

My sample code is written in Turbo Pascal for Windows, although the techniques apply just the same to Turbo C++ for Windows. I use Borland's Object Windows Library in my examples, but if you use Microsoft Foundation Classes it should still be possible to understand the basic concepts.

My code emulates some of the functions of Borland's BWCC.DLL, which many people use to enhance the visual appearance of their programs. This DLL provides an assortment of functions that can give your program that extra visual 3D look without too much work on your part. Unfortunately, the size of the DLL can be a big factor in deciding whether or not to use it, especially if you are writing a shareware program and want to keep its size down. You may also have your own reasons for not wanting to use the DLL, and this may cause you to look for other solutions.

I will demonstrate how to use what is called owner drawn controls. My examples show how to simulate the BWCC style buttons from BWCC.DLL using owner drawn buttons. If you want to use owner drawn buttons in your programs, it should be rather straightforward to use my code "straight out of the box" or modify it to suit your needs.

We'll start with the obvious question. What are owner drawn controls? An owner drawn control is a special type of control which is designed to allow the program to specify custom behavior. It is most often used for listboxes with graphics or for custom buttons.

Whenever a user clicks on an owner drawn control or it changes state in some other way, like getting the focus, a special message is sent to the owning window of this control. This message is called WM_DRAWITEM, and its purpose is to let the window know that one of it's owner drawn controls needs attention. Along with this message, a pointer to an information structure is sent to the window. This structure contains information about what exactly happened with the control, and which control it was. The pointer to this structure is passed in lParam. In Pascal format, the structure looks like:

```
  TDrawItemStruct = record
        CtlType: Word;              Control type   can be  odt_Button, odt_ComboBox,
odt_ListBox, odt_Menu
        CtlID: Word;                ID of Control
        itemID: Word;               Not used for buttons. Has different meanings for other types
of controls
        itemAction: Word;           What happened.   For buttons, tells if gained or lost focus, or
selected
        itemState: Word;            What state control should be in after this drawing.   Buttons
only use   ods_Disabled, ods_Focused, and ods_Selected
        hwndItem: HWnd;             Window handle for the control
        hDC: HDC;                   Display context to be used for drawing the control
        rcItem: TRect;              Defines clipping boundary rectangle for control
        itemData: Longint;          Not used for buttons.   Only used for listboxes and
comboboxes
 end;
```

The owning window can examine this structure and determine what needs to be done with the control. By looking in the CtlType field, it will know what type of control the message is for. For a owner drawn button, this will be odt_Button. The CtlID field contains the ID of the control. The itemID field is not used for owner draw buttons. The itemAction field tells what happened with the control to cause this WM_DRAWITEM message. It can contain oda_DrawEntire, oda_Focus, or oda_Select. Only oda_Focus and oda_Select are relevant for owner drawn buttons. If oda_Focus is set, then the focus for the button changed, and you must check itemState to see whether or not the control gained or lost the focus. If oda_Select is set, the selection state of the button changed, and you must check itemState to know what the new state is.

The itemState field specifies what state the control should be drawn in next. To check the values of itemAction and itemState, you must use the logical AND operation since they can contain more than one value. If (itemState AND ods_Focused) = TRUE, then the button has the focus. If (itemState AND ods_Selected) = TRUE, then the button is selected, or pushed.

The hwndItem field specifies the window handle for the control. You can use this to send messages to the control's window procedure. The hDC field is the display context that should be used when drawing the control. The rcItem field defines the clipping rectangle for the control. It is used mainly with owner drawn menus. The itemData field is only used for listboxes and comboboxes.

There are a couple of ways that your window procedure can process the WM_DRAWITEM message. It can either draw the control itself, or it can pass the message on to the window procedure of the control. This will use an object oriented technique and let the control draw itself instead of the main window procedure having to worry about how to draw each control. This is the technique that I used in my example code. The dialog window merely passes the WM_DRAWITEM message along to the control.

The control reacts to this message by looking at the TDrawItemStruct record and determining what state it should draw, and then draws the button using StretchBlt. I originally wrote this to draw with BitBlt, but when the program was tested under the 1024 x 768 resolution while using large fonts, it became obvious that hardcoding the size of the bitmap didn't work properly when the dialog sizes were increased. This problem has basically two solutions. Either use StretchBlt to draw the bitmap button at a larger than normal size, or have separate bitmaps depending on the resolution.

Both of these methods have their pros and cons, and in the long run I decided to just use StretchBlt. You will notice a degradation in the quality of the bitmaps if you do run in the high resolutions and use the large fonts because StretchBlt can't do a perfect job scaling an image up.

That's the basic idea for using owner drawn buttons. Things will probably be much clearer after actually looking at the source code. I'll briefly describe how to use it.

My code is primarily designed to be used for owner drawn buttons in a dialog box, although there's no reason why you can't use them in a normal window. There are several steps that you will have to take to use this code in your own programs.

1. DESIGN YOUR DIALOG. When designing your dialog, you will need to create a standard button for each owner draw button in the dialog. Remember what the button ID is, as that's what you'll need to know to associate your owner draw control object to this button. You will need to set the size of the button to be 32 x 20, which is half the size of the actual bitmap for Borland's standard VGA bitmaps. I'm assuming that you are using Borland's Resource Workshop to design your dialog. After you have all of your buttons

positioned where you want them and sized properly, bring up the control attributes dialog by double clicking on each control, and set the control type to owner draw button. After this, you won't be able to see the button anymore, but it will still be there. Save your dialog.

2. DESIGN THE BUTTON BITMAPS. You can use any Windows graphics program that can save in BMP format to design the actual bitmaps. The bitmaps will use the BWCC numbering scheme for their names. The numbering scheme is: normal = button ID + 1000, pressed = button ID + 3000, and focused = button ID + 5000. This means that if your button ID is 500, the bitmap number for the normal button without focus will 1500, for pressed 3500, and for focused it will be 5500. These are the names that you will give to the bitmaps when you save them. There is a shareware program written by N. Waltham called Buttons that will automate this task for you by creating all of the necessary bitmaps from one main bitmap. It automatically adds a 3D shadow effect and has some other useful features. This program is available on Compuserve in the BPASCAL forum, and the author can be contacted at 100013.3330@compuserve.com. Although you must register with the author, I don't mind sending copies of it via Internet email to interested users.

3. ADD NECESSARY CODE TO YOUR PROGRAM. You will need to add OwnDraw to your USES statement in your program. You will also need to make your dialog object a descendant of TOwnerDialog. The sample program shows an example of doing this. The last thing to do will be to use the NewButton method in your dialog's constructor for each owner draw button in that dialog. The NewButton method is called with the button ID, and a Boolean True or False depending on whether or not you want that button to be the default button in the dialog. If this is True, then it will be the button drawn with focus when your dialog is initially created.

That's all there is to it. When your dialog is displayed, the owner draw buttons will be displayed along with it. Of course, to get the buttons to do some useful function, you will need procedures in your dialog object to respond to each button selection. The sample program demonstrates this too.

As you can see using bitmapped buttons in your programs is not quite as difficult as it may at first look. When properly used, bitmaps can really make a big difference in the look of your program.

I welcome any comments you may have good or bad. I can be reached on Compuserve at 71044,1653, on America OnLine at TSnoddy, and on the Internet at tsnoddy@nyx.cs.du.edu.

Hacker's Gash

by Mike Wallace and Pete Davis

This is our first attempt at a tips/tricks column. If you couldn't figure out what the title meant, blame Pete. Here are three tricks we've come up with. Hope you like them. If you have any you want to share with our readers, send them in! Full credit will be given for all tips we publish, of course.

1) Menu bar on a dialog box: We spent a lot of time on this one, but the solution turned out to be a simple one. In the dialog box definition in either (a) the .RC file, or (b) a .DLG file you include in the .RC file, throw in the lines:

```
STYLE WS_OVERLAPPEDWINDOW
 MENU <menu name>
```

where <menu name> is a menu you have defined in the .RC file. The style WS_OVERLAPPEDWINDOW combines WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU and WS_THICKFRAME, and is a standard parent window.

2) Highlighting a button on a dialog box when the cursor is over it: By highlight, I mean the button moves "in" slightly, and it's a cool effect that can look pretty impressive. Add the following declarations to the function that controls the dialog box:

```
  static BOOL      ButtnDn;
static int       LastDown;
static int       IDNumber=0;
int               counter;
```

Next assume you have 5 buttons on your dialog box, and these buttons are identified by the constants IDB_BUTNx (where x is a number between 1 and 5), which are defined in the .H file for the .DLG file containing the definition for your dialog box. Also assume these five constants are defined sequentially, say, 101 to 105. The variable "hDlg" is the handle to the dialog box, and "message" is the message passed to the function by Windows (these are, respectively, the first and second parameters passed into a standard window-handling function). Add the following two cases inside the "switch(message) " structure:

```
case WM_SETCURSOR:

        ButtnDn= TRUE;
       LastDown= IDNumber;
       IDNumber= GetDlgCtrlID(wParam);
       for(counter=IDB_BUTN1; counter<=IDB_BUTN5; counter++)
            if(counter==IDNumber)
                  SendDlgItemMessage(hDlg   counter, BM_SETSTATE, TRUE, 0L);
        if(IDNumber != LastDown)
            SendDlgItemMessage(hDlg   LastDown, BM_SETSTATE, FALSE, 0L);
break;

case WM_NCHITTEST:

        if (ButtnDn)
            ButtnDn = FALSE;
            SendDlgItemMessage(hDlg   LastDown, BM_SETSTATE, FALSE, 0L);

   break;
```

3) Using DlgDirList for a subdirectory: I recently tried using the DlgDirList function

passing a pathname like "DATA*.DAT" to fill a list box with all the files in the DATA directory ending with .DAT (by the way, the double backslash ("\") is needed because the backslash is a special character in a C string). Afterwards, I found out that if this function is successful, it changes the current working directory to whatever is specified in the string (here, the "DATA" directory). Afterwards, I tried changing the directory back to the original working directory, but if the function got called again, chaos ensued. I managed to get around this by including in the program the following line:

```
#include <direct.h>
```

The direct.h file has prototypes for the directory-manipulating functions, including the one needed here: chdir, which changes the current working directory. I replaced the call to DlgDirList with the following code:

```
chdir("DATA");
 DlgDirList(hDlg, "*.DAT", IDL_POPUP, IDL_DIRSTRING, 0);
 chdir("..");
```

The "hDlg" variable is the handle to the dialog box, "IDL_POPUP" is the dialog box ID value for the list box control, "IDL_DIRSTRING" is the dialog box ID value for a static text control that will be updated with the current path name, and 0 is the DOS file attribute used to retrieve a list of read/write files with no other attributes set. There are other values you can use for the DOS file attribute value to retrieve, for example, subdirectories and drives. This code simply changes to the DATA directory, gets a list of the .DAT files in that directory and puts that list in the IDL_POPUP list box and the path in the IDL_DIRSTRING static text control, and then changes back to the parent directory. You can use the DlgDirSelect function to retrieve a selected filename from the list box.

Special News

by Mike Wallace

We recently got a letter from Todd Snoddy offering to put WPJ on the America Online information system. We said "Sure", and a couple of days later got a phone call from a sysop for the board. He liked the magazine (all contributing authors can now pat themselves on the back) and asked how we would feel about holding an on-line conference on AO, giving readers a chance to talk directly to us and ask us any questions they have about the journal. Sounded good to us, so we agreed, and now hope to bring this to you AO subscribers soon. The details are still being worked out, so watch the AO Programming Library for updates.

For the benefit of our readers without access to America Online, I'll try to write down what questions get asked and our answers and include them in our next issue.

Thanks Todd, and the great folks at America Online!

Windows 3.1: Using Version Stamping library

By Alex Fedorov

Windows 3.1 introduces the new resource type with ID number 16 called VS_FILE_INFO. This resource contains the symbolic data about the file version, its description, the company name and so on. Such information can be used to determine the file type/subtype and its version and can be used in installation programs. Resource data lives as a set of blocks; each block contains information of its own. The first one (with a fixed size) describes the TVS_FixedFileInfo structure. The fields of this structure contain the following information: file version, environment type and version, file type and subtype. There are several subblocks, which contains symbolic information with the following IDs (the table contains only those which must present):

ID	Contents
CompanyName	Company name
FileDescription	File description
FileVersion	File version
InternalName	Internal file name
ProductName	Product name
ProductVersion	Product version

Resource of this type can be created with resource editor, such as Resource Workshop (version 1.02 or higher), or with resource compiler (RC or BRC). The latter needs the resource template. VS_FILE_INFO resource for resource compiler looks like the following:

```
1 VERSIONINFO LOADONCALL MOVEABLE

FILEVERSION          3 10,0,61

PRODUCTVERSION              3 10,0,61
    FILEFLAGSMASK           VS_FF_DEBUG | VS_FF_PATCHED
    FILEFLAGS               VS_FF_DEBUG
    FILEOS                      VOS__WINDOWS16
    FILETYPE                VFT_APP
    FILESUBTYPE                 VFT2_UNKNOWN
    BEGIN
        BLOCK "StringFileInfo"
        BEGIN
            BLOCK "040904E4"
            BEGIN
              VALUE "CompanyName"    "Microsoft Corporation"
              VALUE "FileDescription"    "Windows Terminal application file"
              VALUE "FileVersion"    "3.10.061"
              VALUE "InternalName"    "Terminal"
              VALUE "LegalCopyright" "Copyright 51 Microsoft Corp. 1991"
              VALUE "OriginalFilename" "TERMINAL.EXE"
              VALUE "ProductName" "Microsoft56 Windows31 Operating System"
              VALUE "ProductVersion"     "3.10.061"
            END
        END
    END
```

To get access to VS_FILE_INFO resource data you can use the functions from VER Dynamic Link Library included with Windows 3.1.

[Editor's Note: There was code that belonged here. Because of problems encountered while

generating the help file, it had to be removed. The code is provided in the WPJV1N3.TXT file. We are investigating this problem and will try to find a solution before the next issue is released.]

Book Review

by Pete Davis

Microsoft Windows 3.1 Programmer's Reference Library

If you're a serious Windows programmer you've probably got Microsoft's series of books on Windows programming. The series is broken up into 4 volumes and two extra books, as follows:

- Volume 1: Overview

This is as it says, an overview. It covers a lot of different topics from how windows are managed, graphics, and how to write extensions for Control Panel and File Manager.

- Volume 2: Functions

This is an alphabetical listing of all the Windows 3.1 functions. It's a big one.

- Volume 3: Messages Structures and Macros

Like the previous two this one is exactly what it says it is. it's got some great information on all the structures, which comes in real handy.

- Volume 4: Resources

This one has a lot of information on file formats for a bunch of different Windows files (.GRP files, Meta Files, etc.) It's also got a bit of information on creating Help files, using assembly language in Windows, and more.

- Programming Tools

Not listed as a Volume. This is more of an additional Microsoft C reference. It covers a lot of the SDK tools, debugging with Codeview, data compressions and that kind of stuff.

- Guide to Programming

Also not listed as a Volume but I would have made it one. It covers different types of resources like Icons, Dialog boxes, etc. Printing, DLLs, MDI, Fonts, Assembly and C, and more.

Ok so there's our list. I would say, all- in-all, this is probably the most definitive resource on programming for Windows and that anyone planning on doing a lot of Windows programming should get the whole series. It's a big ticket item. The least expensive book is $23 (US) and the most expensive is $39 (US).

I suggest you get some other books and not limit yourself to just these. Although they cover most of the topics most programmer's will need, the books are lacking in some areas and there are some typos which have slowed this programmer down on more than one occasion. Every book has typos and that's why, as a serious programmer, your library should come from several sources so you can check the books against each other when you run into problems.

These books are however, well worth the cost. There's a lot of stuff covered in these books that you won't find in other sources. The detail on different file formats, for example, I haven't seen in any other books. The structures list is very complete and is a very handy reference. There's also some good information on memory management and DLLs.

What these books aren't

These books aren't everything and like I said, don't limit yourselves to just these. The Windows API bible (which Mike will be reviewing) has some great information too and it's a good one to check against Microsoft's Function reference when you get suspicious that a function isn't doing something it's supposed to do. Also, because it's Microsoft, you're not going to get the wealth of inside, under-the-hood, information like you get from Undocumented Windows (Andrew Schulman, David Maxey, Matt Pietrek. Addison-Wesley. See the review in WPJ Vol.1 No.1)

Unfortunately because Windows is such an enormous system and no single book can even begin to cover every aspect of it, we programmers have to get a lot of different books. If you're serious, though, you should consider setting aside the money to get this collection.

Book Review

by Mike Wallace

The Waite Group's Windows API Bible

by James L. Conger

Earlier in this issue Pete reviewed the Microsoft Windows 3.1 Programmer's Reference Library. Now it's my turn to review my Windows programming reference book of choice: The Waite Group's Windows API Bible by James Conger. Pardon my French, but this book kicks serious "lune". While the Microsoft books spread out everything across several books, the Waite Group threw it all into one book, making anything you want to look up easy to find. This book is a must-have for anyone wanting to learn how to program Windows. Here are the details:

The book is divided into 30 chapters each discussing a distinct area (e.g., creating windows, menus, windows messages, metafiles). Each chapter starts off with a discussion of its topic (usually several pages long), then a summary of the associated functions (a list of the function names with a one line purpose), and then a complete description of the same functions in the summary. Their descriptions are detailed and include suggested uses for the function, a great "see also" reference list (more complete than the Microsoft books), an explanation of all the parameters and the return value, and code (real programs) that shows how to use the function. It's my one stop for looking up a function. Plus, the inside front cover contains a list of all the API functions with a page number so I can quickly jump to the function I want to look up, followed by a list of messages grouped by category (e.g., button notification codes, windows messages).

But wait there's more...the book also includes a pullout reference card containing a list of all functions with the type declarations for all parameters and the return value, plus the same list of messages that's on the inside cover. The function list in the pullout card is organized the same way as the chapters, so that all the functions listed in the chapter on, say, sound functions, are grouped together in the pullout under the heading "Sound Functions." I couldn't ask for a better list. In the Microsoft books, you have to know the name of the function before you can look it up, and there have been many times when I didn't know what the function was called, but I knew what it did. This book lets me find the name of the function quickly, and then shows me how to use it.

In the appendices there is a discussion of useful macros from WINDOWS.H, mouse test hit codes, and a printout of WINDOWS.H, which has been very helpful to me on occasion. The book ends with a very complete index, making it extremely easy to look up related subjects.

There are very few faults with this book but there a few. One is the strange absence of the Windows API functions starting with the letter "N". I am not making this up. The Microsoft reference book on functions lists four functions starting with "N": NetBIOSCall, NotifyProc, NotifyRegister and NotifyUnRegister. None of these functions are in the API Bible. There is also no discussion of OLE or True Type fonts, although the author writes (in the introduction) these will be covered in a separate volume currently under development.

Also the Microsoft volume on programming tools (those included with the SDK, such as Spy) covers an area not addressed by the API Bible. If you want to use the Microsoft SDK, the API Bible won't be of much help, but this isn't much of a fault, because Microsoft isn't the only producer of Windows SDKs, and you're going to get manuals for any SDK you buy, so why include a lot of information that some readers won't need to know?

To me the faults of this book do not detract from its overall usefulness. It has paid for itself many times over, and when compared to the Microsoft reference books, the price seems insignificant. Total price for the six books in the Microsoft Windows 3.1 Programmer's

Reference Library: around $170- 180. The Windows API Bible: $40. I recommend this book. You'll thank me for it.

Printing in Windows

by Pete Davis

Ok I failed to get this article out last time and I've been feeling really guilty about it, so here it is. Writing this article hasn't been easy. Writing about printing reminds me of my first attempt at printing under Windows. It wasn't a very good one.

Actually printing under Windows isn't a big deal and if you're printing graphics, it's a lot easier than you'd think. My real problem with printing the first time was incomplete/incorrect documentation. I won't mention names, but I saw at least 3 examples of printing in different books on Windows programming and every one of them either misled me or had code that didn't work.

Here's the deal. When you print you MUST, and I mean MUST (I'm really serious about this), have an abort procedure. This is an absolute necessity. You MUST have it. You can't do without it. You'll have problems if you don't have an abort procedure, so write one. Are you starting to get the picture here? An abort procedure is NOT optional, you MUST have it.

Now you might be wondering why I'm emphasizing this point. Every book I read on printing made it sound like an option. I needed to print one page of text (and not much text at that) and didn't have a need for an abort procedure, so I didn't include one. I kept getting UAEs as soon as I tried to print. It even occurred to me that I might need the Abort procedure, and I specifically looked for a phrase which said something along those lines in every bit of documentation I had. Finally, 4 days later, I decided I'd just pop an abort procedure in. (I had tried just about everything else at that point, why not?) Of course, it worked, and I had wasted four days because I HAD TO HAVE AN ABORT PROCEDURE!!!!

So I'm assuming that, after reading this, none of you are going to spend four days trying to figure out why your print routine UAEs only to find that you forgot your abort procedure. If I hear of any of you doing this after reading this article, I'm going to come over and personally give you a whuppin (as we say down in Arkansas). If you haven't, at this point, realized that an abort procedure might possibly be a good idea to include in your print procedure, you fit into one of the following three categories:

1> You program on the cutting edge of Atari 2600 technology.
(or some other cutting edge of obsolescence)
2> Blind and not able to read any of this anyway.
3> Just plain stupid

There so I have that off my chest, on to the meat of this article, which is how to print. It's real easy!!!

The first thing we'll look at is our Abort procedure. (Did I mention that you need this part?) The abort procedure is real simple. It receives the device context of the printer and an error code. In our example, the nCode is the error code and if it's non-zero, you have an error.

Basically all your abort procedure has to do have a message loop. I don't do anything else with it, myself. It should look something like this.

```
BOOL FAR PASCAL AbortProc(HDC hdcPrint   short nCode)

MSG msg;

   while(!bPrintAbort && PeekMessage(&msg   NULL, 0, 0, PM_REMOVE))
         if(!hDlgPrint || !IsDialogMessage (hDlgPrint &msg))
```

```
            TranslateMessage(&msg);
            DispatchMessage(&msg);


  return !bPrintAbort;
```

The next thing on our agenda is the PrintDlgProc procedure. This is basically the dialog box procedure for our cancel printing dialog box. Although there is a cancel button in our dialog box, instead of checking specifically for the cancel button, our only concern is with a WM_COMMAND. (We only have one button, so what other WM_COMMANDS are they going to be sending?) If we get the WM_COMMAND, we basically abort the printout (by setting bPrintAbort = TRUE) and destroy our dialog box. Pretty darn simple.

```
BOOL FAR PASCAL PrintDlgProc(HWND hDlg   WORD message, WORD wParam, LONG lParam)


    if (message == WM_COMMAND)
            bPrintAbort = TRUE;
            EnableWindow(GetParent(hDlg)   TRUE);
            DestroyWindow(hDlg);
            hDlgPrint = 0;
            return TRUE;


        return FALSE;
```

The next procedure is our printing procedure. There are a couple of neat things here that I'd like to talk about. First of all, there's the four lines involved with getting information about our printer driver. What we're doing here is just getting the default printer device. We get the information via a GetProfileString command. Under the section "windows" and then the line starting with "device". That's our printer. All the information goes into szPrinter which we then break into the Device, Driver, and PortName using the strtok function. This basically breaks out our string into separate strings. The second parameter of strtok is the thing we want to break our string apart at. In this case, we want it broken up at the commas. Also, notice how we only pass szPrinter once. If we use NULL in the next two occurrences of strtok, it knows to continue using szPrinter and to continue from where we left off which, in this case, is at the last comma.

```
GetProfileString("windows"   "device", ",,,", szPrinter, 80);
 lpDevice = strtok(szPrinter, ",");
 lpDriver = strtok(NULL, ",");
 lpPortName = strtok(NULL, ",");
```

Our next job is pretty simple   we just create a device context based on the information about our driver.

```
/* Create the device context. */
 if ((hdcPrint = CreateDC(lpDriver, lpDevice, lpPortName, NULL)) == NULL)
 MessageBox(hWnd, "Could not assign printer.", "ERROR", MB_OK);
 return FALSE;
```

Here's another thing you don't really have to deal with in DOS. With Windows, you need to know where you are on the page. The reason is that you have to tell Windows when

you're done with the current page. This means you need to know the size of a page and the size of the text on the page. Now, this can be a bit of a pain, but it also allows for a lot of interesting stuff. For example, you don't have to start from the top of the page and go down. You can print the bottom part of the page, and then print something in the middle of the page, and so on. Then you just tell Windows to eject the page. In our case, we're going to be assuming 1 page or less of text and not really deal with that. (As they say in school books, this is left as an exercise for the reader.) All we're going to do is use the text size to tell us where to print the next line of text. First we have to use CurYPos as our current Y position on the page. (I love descriptive variable names.) Then we need to find out how tall a single character is. That's done from getting a text metric for the printer and adding the character's height, plus the external leading, which is kind of like the blank space between lines.

```
CurYPos = 1;
 GetTextMetrics(hdcPrint, &tm);
 yChar = tm.tmHeight + tm.tmExternalLeading;
```

The next step is to create our dialog box and abort procedure. By the way, the abort procedure is not optional.

```
lpfnPrintDlgProc = MakeProcInstance (PrintDlgProc hInst);
 hDlgPrint = CreateDialog(hInst, "PrintDlgBox", hWnd, lpfnPrintDlgProc);
 lpfnAbortProc = MakeProcInstance(AbortProc, hInst);
 bPrintAbort = FALSE;
```

Now the secret to printing is all in the Escape commands. (Actually, in Windows 3.1, they have commands you can use instead of Escape, but since we try to maintain our 3.0 backwards compatibility, we're going to use Escape.) The Escape is the heart of the printing routines. You use it to send different commands and/or data to the printer driver. The first one we'll send is our SETABORTPROC command. This sets up our all-important abort procedure.

```
Escape(hdcPrint  SETABORTPROC, 0, (LPSTR) lpfnAbortProc, NULL);
```

The next Escape we're going to send is our STARTDOC command. We're going to pass the name of our document and the length of the string that has the name of our document. Pretty straight-forward.

```
Escape(hdcPrint  STARTDOC, strlen(szMsg), (LPSTR) szMsg, NULL);
```

Since our example involves reading from a file, we're just going to read one line at a time from the file. We use a TextOut function to output our string to the printer driver. We need to give X and Y coordinates. In our case, the X is always 0 and the Y position is calculated from the GetTextMetric which we did above. We just add the text height to the current Y position after each line.

```
while (fgets(CurLine  132, inFile)!=NULL)
        TextOut(hdcPrint  0, CurYPos, CurLine, strlen(CurLine)-2);
        CurYPos += yChar;
```

When we're done we need to send a NEWFRAME command via the Escape function. This basically means to advance to the next page. Now, if you're printing multiple pages, you need to do a NEWFRAME between each page. In our case, we're only allowing one pages, so we do it right at the end.

```
Escape(hdcPrint  NEWFRAME, 0, NULL, NULL);
```

The last thing to do is send an ENDDOC. This basically tells the printer driver we're done and it can begin printing the document.

```
Escape(hdcPrint    ENDDOC, 0, NULL, NULL);
```

After all that we just close the file and delete the device context for our printer, then get a cup of coffee and a smoke and relax.

```
fclose(inFile);
 DeleteDC(hdcPrint);
```

Ok so it's a little more complex than printing text under DOS, but the thing I didn't mention is the graphics. Now, I'm not going to go into great detail about it, because it is a little more complex, but not much. Essentially, all you have to do is route all of your graphics to a printer device context instead of a screen device context. The complexity comes in when you're trying to figure out pages sizes and that kind of stuff, but for the most part, printing graphics is as easy as printing text.

There are some things we didn't cover here. One of them is a thing called banding, which is a way of printing your page in parts called bands. This is particularly useful for dot-matrix and other non-postscript printers. Banding makes it a bit faster to do the printing. Some printer drivers have the banding built-in and I, personally, have never had to work in an environment where printing had to be particularly fast, so I've avoided banding. If someone feels that banding is of particular importance, they're more than welcome to write an article on it.

That just about wraps up printing. It's really not all that complex, and as you can see, Windows gives you a lot of power as to how to handle it. There are all kinds of neat and nifty things you can do, like mixing graphics and text, which is a cinch. Try doing that in DOS. And last but not least, please, don't forget your abort procedure.

Advanced C++ and Windows

By Andrew Bradnan

Talking to the User
Overview

While Windows offers a great variety of ways to talk to the user, none of them are very easy. Even fewer are quick to implement. In this article we will look at some C++ classes to make things a little easier.

Output
Introduction

Let's look at a few ways to talk to the user. The simplest way to tell the user something is to create a message box. This is one of the easiest things to do in Windows. Luckily, Windows does most of the work for us. We are going to make it even easier. With the classes we are going to create you will never have to remember what the parameters are, what order they go in, and exactly how they spelled all the constant values you can pass in and that MessageBox() passes back. We also need a good way to tell the user and ourselves about error messages (God forbid).

Message Boxes

If you are at all familiar with the MessageBox () call you are well aware that there are several different incantations. Each has its own purpose and can be used for several circumstances. Its function prototype looks like this: int MessageBox (HWND hwndParent, LPSTR lpszText, LPSTR lpszTitle, UINT fuStyle); You can pass it 18 different flags (or combinations) and it will return seven different flags to tell you what the user did. Not as ugly as CreateWindow() but not easy either.

OKBOX

A stream is an abstraction referring to the flow of data from a producer to a consumer. The first output object we are going to create is an OKBOX. Once again it does exactly what it sounds like. It displays a message box with one "OK" button. The OKBOX will allow us to use C++ stream conventions. Let's first look at how we would like to use an OKBOX so we can write the appropriate member functions.

OKTEST.H

```
//
//   OKBOX Test Header
//   Andrew Bradnan (c) 1992
//   C++ Windows Version
//

#ifndef __OBJECTS_H
#include <objects.h>
#endif

class WINDOW : public BASEWINDOW
 public:
     WINDOW (LPCSTR lpcszClass   HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpszCmdLine, int nCmdShow) : BASEWINDOW (lpszClass, hInstance, hPrevInstance,
lpszCmdLine, nCmdShow) ;
     BOOL OnCreate (CREATESTRUCT FAR *lpCreateStruct);
;
```

OKTEST.CPP

```
//
//   OKBOX Test Module
//   Andrew Bradnan (c) 1992
//   C++ Windows Version
//

#include <oktest.h>

BOOL WINDOW::OnCreate:: (CREATESTRUCT FAR *lpCreateStruct)

      OKBOX msg ("OK Test Caption");
      msg << "User notification."
      return TRUE;


int PASCAL WinMain (HANDLE hInstance    HANDLE hPrevInstance, LPSTR lpszCmdLine, int
nCmdShow)
 WINDOW Window ("OK Test", hInstance, hPrevInstance, lpszCmdLine, nCmdLine);
 return Window.MessageLoop ();
```

We are using our WINDOW virtual function to trap the WM_CREATE message. Here we define msg as an OKBOX with the caption "OK Test Caption" When we want to send a message to the user we just stream some text to it. This will invoke the message box with the "User notification." It is much easier to read. Now that we have an easy way to send a message to the user or ourselves let's look in the "black box" to see how OKBOX works.

OKBOX.H

```
//
//   OKBOX Header
//   Andrew Bradnan (c) 1992
//   C++ Windows Version
//

#ifndef __OKBOX_H
 #define __OKBOX_H

#ifndef __WINSTR_H
 #include <winstr.h>
 #endif

class OKBOX
 public:
      OKBOX (LPSTR lpszCaption);
      int operator<< (STRING& strMessage);
 private:
      STRING strCaption;

 #endif // __OKBOX_H
```

OKBOX.CPP

```
//
//   OKBOX Module
```

```
//   Andrew Bradnan (c) 1992
//   C++ Windows Version
//

#include <okbox.h>

OKBOX::OKBOX (LPSTR lpszCaption) : strCaption (lpszCaption)
;

int OKBOX::operator<< (STRING strMessage)

      return MessageBox (NULL   strMessage, strCaption, MB_OK);
```

Short and sweet. We have a constructor that takes a string which we will save as the caption for the message box. Our other function, operator<<() is called by the compiler when ever it sees an OKBOX object on the left of the << and a STRING (or an object that can construct a STRING like a LPSTR) object on the right side.

ERRORS

Unfortunately errors are bound to occur. We will assume these are all going to be user errors. If we think a bit, an ERROR object is going to be coded almost exactly like an OKBOX. In fact, let's rewrite our OKBOX object so that we can take advantage of this. Quality if free, but we have to pay for it up front. So that we can reuse some of the code we will create an object call MSGBOX. MSGBOX will contain the code shared by the OKBOX and the ERROR object.

MSGBOX.H

```
//
//   MSGBOX Object Header
//   Andrew Bradnan (c) 1992
//   C++ Windows Version
//

#ifndef __MSGBOX_H
 #define __MSGBOX_H

#ifndef __WINSTR_H
 #include <winstr.h>
 #endif

class MSGBOX
 public:
      MSGBOX (LPSTR lpszCaption)
      : strCaption (lpszCaption)   fuStyle (Style)
      ;
      int operator<< (STRING& strMessage)

               return MessageBox (NULL   strOutput, strCaption, fuStyle);
      ;
 private:
      STRING strCaption;
      UINT fuStyle;
```

```
#endif // __MSGBOX_H
```

The MSGBOX object only adds fuStyle so that different message boxes can be created. Now we can rewrite OKBOX to use the code in MSGBOX. OKBOX will be inherited from MSGBOX. Since we will inherit all the public member functions, we will only have to write a short constructor.

OKBOX.H

```
//
//   OKBOX Object Header
//   Andrew Bradnan (c) 1992
//   C++ Windows Version
//

#ifndef __OKBOX_H
 #define __OKBOX_H

class OKBOX : public MSGBOX
 public:
     OKBOX (LPSTR lpszCaption) : MSGBOX (lpszCation MB_OK) ;


#endif // __OKBOX_H
```

The code for an ERROR object is exactly the same except for the value of fuStyle.

ERROR.H

```
//
//   ERROR Object Header
//   Andrew Bradnan (c) 1992
//   C++ Windows Version
//

#ifndef __ERROR_H
 #define __ERROR_H

class ERROR : public MSGBOX
 public:
     ERROR (LPSTR lpszCaption) : MSGBOX (lpszCation MB_ICONSTOP |
MB_SYSTEMMODAL | MB_OK) ; ;

#endif // __ERROR_H
```

Great! Now we can notify the user using the OKBOX object and tell the user about errors using the ERROR object.

Questions

Your programs often need to ask the user a question. If this is a yes or no affair we can use a message box. Let's look at how we would like to use an object like this. Then we will write the member functions to fill out the class.

QTEST.H

```
//
//   QUESTION Test Header
```

```
//   Andrew Bradnan (c) 1992
//   C++ Windows Version
//

#ifndef __OBJECTS_H
 #include <objects.h>
 #endif

class WINDOW : public BASEWINDOW
 public:
     WINDOW (LPCSTR lpcszClass    HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpszCmdLine, int nCmdShow)
     : BASEWINDOW (lpszClass    hInstance, hPrevInstance, lpszCmdLine, nCmdShow) ;
     BOOL OnCreate (CREATESTRUCT FAR *lpCreateStruct);


QTEST.CPP

//
//   QUESTION Test Header
//   Andrew Bradnan (c) 1992
//   C++ Windows Version
//

#include <qtest.h>

BOOL OnCreate (CREATESTRUCT FAR * lpCreateStruct)

     OKBOX ("Question Test");
     if ((QUESTION) "Do you like the object?")
          msg << "Great   I hope your programming is easier."
     else
          msg >> "What the hell do you know!";

  return TRUE;


int PASCAL WinMain (HANDLE hInstance    HANDLE hPrevInstance, LPSTR lpszCmdLine, int
nCmdShow)
     WINDOW Window ("Question Test"    hInstance, hPrevInstance, lpszCmdLine, nCmdLine);
     return Window.MessageLoop ();
```

Once again we are trapping the WM_CREATE message and trying out our QUESTION object. The code is kind of sneaky. I know one thing, you can actually read C++ code if you write your classes carefully. I would hate to write those four lines of code in C. They certainly would not be as easy to read. Let's look at how the C++ compiler helps us. First of all, when the compiler sees that we would like to cast the string to a QUESTION it creates a temporary QUESTION object using our string. Then since the if statement really wants a BOOL the compiler will cast the QUESTION to a BOOL. It is in this explicit cast, operator BOOL (), that will make the message box call. The cast member function will "look" to see if the user hit the "Yes" or "No" button, and return TRUE or FALSE.

If you didn't understand that let me explain it another way. The C++ compiler did all the work. Seeing as how I'm trying to champion code reuse let's try and use the MSGBOX object. We are only going to have to write two member functions for this object. The constructor and the cast member function.

QUESTION.H

```
//
//   QUESTION Object Header
//   Andrew Bradnan (c) 1992
//   C++ Windows Version
//

#ifndef __QUESTION_H
 #define __QUESTION_H

class QUESTION : public QUESTION
 public:
     QUESTION (LPSTR lpszOutput)
     : MSGBOX ("Warning!"    MB_QUESTION | MB_YESNO), strOutput (lpszOutput) ;

          operator BOOL (void)

               if (IDYES == operator<< (strOutput))
                         return TRUE;
               else
                         return FALSE;
     ;
 private:
     STRING strOutput;


#endif // QUESTION_H
```

As you can see the only action really takes place in the explicit cast, operator BOOL (). It is here that we call the member function operator<< () to invoke the message box call. We didn't do too much but transform one call to the MSGBOX class. Since we declared it inline we won't even gain any function call overhead. The C++ compiler will put this ugly code right where our beautiful code exists right now. Pretty damn cool.

Warnings

We also might want to warn the user of impending doom. The File Manager uses this to confirm that you really want to overwrite a file. If works almost like a QUESTION object except it can return whether the user chose "Yes", "No" or "Cancel".

WARNING.H

```
//
//   WARNING Object Header
//   Andrew Bradnan (c) 1992
//   C++ Windows Version
//

#ifndef __WARNING_H
 #define __WARNING_H

class WARNING : public MSGBOX
 public:
     WARNING (LPSTR lpszOutput)
     : MSGBOX ("Warning!"    MB_EXCLAMATION | MB_YESNOCANCEL) ;
```

```
        operator BOOL (void)

            BOOL fUserChoice;
            fUserChoice = operator<< (strOutput);
            if (fUserChoice == IDYES)
                    return TRUE;
            else if (fUserChoice == IDNO)
                    return FALSE;
            else // if (fUserChoice == IDCANCEL)
                    return -1;
    ;
 private:
    STRING strOutput;


#endif // WARNING_H
```

Wondrous.

Why you ask, go through all this trouble just for a MessageBox() call? Well, there are plenty of reasons. Let's review them. First, who can remember all those flag variables? Not me. Does the caption go first, like I know it should (and always write)? No! No matter how many times I write it the correct way, Windows just will not learn. Second, the folks at Microsoft just love to change things. The MessageBox() call may gain new functionality. Why romp through all the code you have written to change it for the latest greatest MessageBoxEx()? It would be a pain and you surely would miss some of your old code. Now all you need to do is make a change to the MSGBOX class. Third, you can add all sorts of checking to the base class. This really doesn't apply in this case but in more complicated classes this will be important. We even gain some checking from the string class. "How?" you ask. There is nothing to screw up with constant strings. I'm afraid not, my friend. Once upon a time, I forgot to export one of my functions (good thing you have never done that) that used one of these classes. My data segment was pointing off to never never land and luckily my STRING's told me as much. They didn't even GP fault, just quietly told me that I had screwed up. Windows 3.1 has brought multimedia to our clutches so let's look at a real example of how powerful C++ inheritance can be. We will be updating MSGBOX.

MSGBOX.H

```
//
//   MSGBOX Object Module
//   Andrew Bradnan (c) 1992
//   C++ Windows Version
//

#ifndef __MSGBOX_H
 #define __MSGBOX_H

#ifndef __WINSTR_H
 #include <winstr.h>
 #endif

class MSGBOX
 public:
    MSGBOX (LPSTR lpszCaption);
    : strCaption (lpszCaption)   fuStyle (Style)
    ;
```

```
    int operator<< (STRING& strMessage);

            MessageBeep (fuStyle);
            return MessageBox (NULL   strOutput, strCaption, fuStyle);


private:
    STRING strCaption;
    UINT fuStyle;


#endif // __MSGBOX_H
```

You have now added multimedia to everywhere you use message boxes. All by changing one measly line. You're a fast worker. Your boss will probably give you a raise. We have just implemented some easy ways to have dialog with the user. They are by no means complicated, but just think of the complicated things you can make this easy, even fun to program. There are plenty of other nasty constants you can play with to create you own classes. Then you can rip those pages right out of the Windows Programmer's Reference.

The Trials and Tribulations of an Antipodean Abecedarian who uses Turbo Pascal to Program Windows

Part 1

They Also Serve Who Only Stand and Wait

by Jim Youngman
[CompuServe 100250 60]

I live in Melbourne Australia and I work in a one man Operations Research department. The programming that I do is in order to solve OR problems that I encounter. I am very much isolated in my job although I do my best to keep in touch with others through professional societies and computer user groups. e- mail has proved invaluable since I had a modem installed a short time ago.

The problems I have struck are often simple in the end, but they can be confusing for a beginning Windows programmer such as myself. The documentation for the various programming languages does not tell you what you need to do; rather it tells you how to do something when you know what it is you need to do. Perhaps sharing some of these trivial trials and tribulations in a series of short articles might just help another beginner.

I have the job of writing a user friendly front end to a mathematical programming package. The end users will be line managers with a background in mechanics rather than computers. This led me to decide that the best medium for them to use the final application would be Windows (we are standardized on IBM compatible machines).

One of the first things I needed to do after setting up various screens using the Borland Resource Workshop was to find out how to run the mathematical programming package from within a Windows program. The package is DOS based and uses the Phar Lap DOS extender to run in 32 bit mode.

How could I do this? I could not find any clues in the manuals, nor in the only book on TPW that I had at the time: Tom Swan's Turbo Pascal for Windows 3.0 Programming (1991) which is an excellent introduction to the subject.

The books being no help I then went on to search the Borland Pascal Help file. Eventually I tried a search on Load and soon discovered the function LoadModule. This was described as loading and executing a Windows application, but there was a cross-reference to WinExec. The description of this function did not specifically mention either Windows or DOS applications, so I tried it:

```
procedure RunIPOPT;
 begin
      WinExec(hWindow 'c:', sw_Show)
 end;
```

This worked OK. However the IPOPT program displays masses of mathematical detail on screen as it is running. I do not want to confuse the end users with this detail, so I want to eventually hide the window in which it runs and bring up a message box to inform the user when the program has run to completion. For the time being I will leave the window visible (sw_Show) so that I can monitor progress:

```
procedure RunIPOPT;
 begin
      WinExec(hWindow 'c:',sw_Show);
      MessageBox(hWindow   'Program Completed','IPOPT', mb_Ok)
 end;
```

Surely this should work. But no! The message box came up on screen before IPOPT had even begun to run.

Again all my books failed me as did the Help file too this time. Borland's Windows API guide had a description of the command, but there were still no examples that showed how it might be used in practice. I was stuck!

All fairy tales have happy endings but they have to begin with "Once upon a time ...".

Once upon a time I was browsing in my favorite computer book store and discovered a copy of Neil Rubenking's Turbo Pascal for Windows Techniques (1992). A cursory look at the index found a reference to the WinExec function. I looked it up. It had exactly what I wanted. I bought a copy immediately. The book is another excellent reference that I now have on my shelf and reach for often.

It seems that the crucial point I was missing was the need to create a loop to test whether the DOS program was still running. I adapted Neil Rubenking's code to produce the following procedure:

```
procedure RunDos(hWindow: hWnd; CommandLine CompletionMessage, CompletionTitle:
PChar);

var IH : word;
       M   : TMsg;

begin
     IH := WinExec(CommandLine   SW_Show);
     if IH <= 32 then
              MessageBox(hWindow   CommandLine,
                       'Failed to run'+chr(13)+chr(10)+'Execution Error'+
chr(13)+chr(10)+'Contact Jim Youngman for Help',
                       mb_Ok + mb_IconHand)
                 else
                    begin
                       repeat
                          while PeekMessage(M   0, 0, 0, PM_REMOVE) do
                             begin
                                if M.Message = WM_QUIT then
                                   begin
                                      PostQuitMessage(M.wParam);
                                      Exit;
                                   end
                                else
                                   begin
                                      TranslateMessage(M);
                                      DispatchMessage(M);
                                   end;
                             end; end do
                        until GetModuleUsage(IH) = 0;
                        MessageBox(HWindow   CompletionMessage, CompletionTitle,
                                   mb_Ok + mb_IconInformation);
              end;
 end;
```

I lived happily ever after at least until the next problem.

Getting in touch with us:

Internet and Bitnet:

HJ647C at GWUVM.GWU.EDU -or- HJ647C at GWUVM.BITNET (Pete)

GEnie: P.DAVIS5 (Pete)

CompuServe: 71141 2071 (Mike)

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

Home Phone (703) 503-3165 (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 the first two 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

Things have been hectic in the WPJ offices (i.e., our basement) - you've been sending us great articles and mail with lots of input (see the Letters column). It's been great - you seem to like what we're doing, and we're having a good time. This month a tips/tricks column (Hacker's Gash) makes its debut in WPJ (Official slogan: "The original party snack"). It was written by us, but we're always glad to hear from you people. If you have any tricks you want to share, let us know about them! We seem to be accomplishing our basic goal of helping people learn how to program in Windows, but this goal brings up an issue I want to write a bit about.

In the Letters column there is a letter from Tammy Steele, formerly of the SDK forum on CompuServe, and in her letter she discusses the accuracy of WPJ. If the magazine isn't accurate, the Microsoft sysops aren't going to recommend WPJ as a source of helpful info. When we started this effort, I didn't envision Microsoft referring people needing help our way. Don't ask me why, I guess it just never occurred to me, but it illustrated to me the point that the available sources of help on this matter shouldn't be exclusive - that is, no one has a monolopy on this field (nor should anyone), because no one source can be everything to everybody. I'm not going to tell you that WPJ can answer every question you ever had about programming in Windows, because different sources offer different things to their readers. We write about the topics we think the most people can benefit from, but in some ways we're much different than, say, "Windows/DOS Developer's Journal". Not to say we're better, just different. Don't limit yourself to just one source of information.

If you have a specific question about Windows programming, you have several options: write us a letter and we'll try to answer it, or, if you have a ID on CompuServe, America Online, GEnie, etc., post it and see if you get a response. Microsoft also puts out a Developer CD full of articles (yes, I have this CD and it's a great source). What I'm leading up to is that the sysops for these services (and their members) help people learn this stuff, but sometimes a complete answer isn't possible (due to space constraints), so why not refer someone to WPJ if we have an article that answers the question? Much like I frequently refer to The Waite Group's Windows API Bible (see review in this issue), I also read some of the other Windows programming magazines. Sometimes they help, sometimes not. As long as WPJ is useful and offers something you can't get elsewhere, we'll stay around. A magazine shouldn't exist for its own sake, and judging from your response, we're succeeding in filling a niche. Pete and I didn't start this because we were bored; we saw an area we thought wasn't getting addressed, namely, help beginners learn the basics, while offering advanced articles for the people who have been around for a while. If you don't like what we're doing, send us a note. If we keep getting positive letters and don't hear anything to the contrary, we'll continue what we're doing. Tammy's concerns about accuracy are well-founded. We'll be the first ones to admit it if we make a mistake. I learn just as much from our contributing authors as you do, but if a mistake slips through, I want to know about it. With your help, we can make WPJ a helpful source of info. 'Nuff said.