The Windows Programmer's Journal Volume 01
Copyright 1993 by Peter J. Davis Number 07
and Mike Wallace August 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	**Bill Lenson**
Contributing Editor	**Dennis Chuah**
Contributing Editor	**Mike Strock**
Contributing Writer	Philip Sloss
Contributing Writer	Rodney Brown
Contributing Writer	Bernard Andrys
Contributing Writer	Todd Snoddy
Contributing Writer	David Falconer

WPJ.INI

by Pete Davis

Another month, another issue. We didn't do an issue last month because we didn't have enough articles early enough, but it gave us a chance to collect a lot of articles for this issue plus a few for next month's issue. That should give us a good head start.

This month you'll notice a survey accompanying the magazine. We're considering putting the magazine in print. Through the help of a publishing company, this may very well happen. That would mean some major changes which I want to get out into the open right away.

First of all, yes, the magazine would cost money. What would that money get you? It would get you 12 issues a year of a great magazine. Mike and I would be doing this for a living. That means that the articles would undergo serious editing, code and text would be checked and double-checked. We'd have time to make sure everything is right. At the moment, Mike and I are doing this part time and can barely afford to spend more than a few hours a month to work on the magazine.

In our current form, it's almost impossible to do any real editing. Why? Well, we can't pay the authors, and it's hard enough to get articles. If we made it difficult, people wouldn't write for us. Paying the authors would fix that. We'd also be able to get some of the big names out there, like Andrew Schulman, Matt Pietrek, etc.

What wouldn't change? One of our hottest areas is help for beginners. As long as I ever have anything to do with this magazine, that won't change. We will always be a place where beginners can turn for information and help. We'd also provide the more advanced articles that we've been providing over the past 6 issues.

Going into print would also allow us to reach a much broader audience than we reach now. We'd be able to get to the people who can't or haven't found us online.

People who subscribe prior to the first printed issue will get a discount of about 25% off the regular subscription rate of around $25. These numbers might change since we're still working out the details.

We're hoping you will all go for this. In all honesty, I don't know how long the magazine would be able to survive for free. Mike and I both do a several things for a living. That doesn't leave much time to work on the magazine and things are only getting worse. If we could make our living off the magazine, we could devote all of our time to it, which we'd both like to do.

Anyway, fill out the survey and send it to my userid on CompuServe, the Internet, or via regular mail. Next month we will draw a name from all the people who responded. The winner will get a choice between:

- The "Win32 Reference" manuals (a 5-book set)

 - A combo of the "Windows API Bible", "Windows Internals", and "Undocumented Windows"

 - The Windows NT Resource Kit

Send the response to me at:

CompuServe: 71644,3570
Internet: 71644.3570@compuserve.com

or regular mail to:

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

We look forward to getting your responses.

Peace.

_Pete Davis

Beginner's Column

By Dave Campbell

Let's see, I know I've got some code here that will show me how to put a dialog box into a DLL...no, not that file. Ok, ok, unzip that entire disk with a -v option into a file, and just search the thing. Oh, no! Guess I must have dumped it.

I sure hate it when I do that, don't you? Me too. And this month, since I want to follow up on my promise of putting the file open dialog box into a DLL, I had to redo it. But that wasn't all bad, since I'm going to explain it anyway.

The end result of this month's column is going to be code that runs identical to that of last month, the difference being that some of the code no longer resides in the executable. To accomplish this, we are going to pull some of the code from Hello.EXE and put it into a dynamic link library (DLL) called FileO.DLL. That's all there is to it. Now for next month...

DLL short-course

DLLs many times are very complex pieces of code, and by their very nature are reasonably tricky to program. That does not mean we should be afraid to make use of the power offered to us. We just need to be careful when using them, and follow the rules.

Most all of us are familiar with ".LIB" files. These are very much like a collection of ".OBJ" files that we can pull out during the building of our file, and save having to write the code again. These LIBs may be locally built, and contain code personally written, or may be third-party libraries costing hundreds of dollars. In either case, the code from the LIB becomes part of your EXE file after the build is completed. If you use the LIB for three different modules, you will have pieces of that LIB file in all three of your executables. That takes up space on your hard disk, and also in memory at execution time.

Execution time? Well, maybe not in DOS, but what if we had written three TSRs and used a third-party library for some of the functionality, and then loaded all three? It would be possible to have, at run-time, three exact copies of the same code in memory. Kinda wasteful, huh?

I hope I'm not boring everyone, I'm simply trying to set the stage for the power of a DLL. DLL stands for "Dynamic Link Library", and as the name implies, the linking takes place dynamically, or at run-time. In our TSR example above, if DLLs were available to us in the DOS world, only one copy of the common code would have been needed, thereby saving disk space because the EXE files would be smaller, and saving memory at run-time because the common code would only be loaded once.

That's exactly what we get with DLLs in Microsoft Windows. When a call is made to a function in a DLL, Windows executes the function as if the function were local to the window running.   The previous sentence is literally correct, and is one of the sticky parts of programming DLLs.

DS != SS

I have no intention of digging into this at the depth that is available in much of the literature, but I am going to try to give an explanation of what DS != SS means, since it is used quite often. The primary thing to remember is that DLLs do not have their own stacks. The calling application's stack gets used during execution of the DLL. This is a problem because of near/far references to data. In Windows programming, all near pointers are assumed to be DS-relative. To avoid a problem with this, either explicitly declare pointers to be FAR, or cast them that way as they are used.

Hello.C

First let's take a look at Hello.C. There's been quite a bit of code removed, and a little added in. I want to discuss the additions first.

In the include section of Hello.C, a #include of the new "fileo.h" was added. This is necessary because some definitions that were in Hello.H are now residing in FileO.H, and our file needs information about it. Almost any DLL that you will be using from a third-party vendor will have a DLL   included that you will have to link with. Normally the DLL will have the prototypes of the functions to call, and any data definitions, or messages that could be passed.

Next in the global definition area, we define a HANDLE type named "hFileo". This will be used to identify the DLL after it is loaded by our program.

```
HANDLE hFileo;
```

Next, a prototype was added for "DoFileOpenDlg". This will be the launch point to get into the DLL to use the dialog box we created two months ago:

```
void DoFileOpenDlg(void);
```

WinMain now contains the following code:

```
hFileo = LoadLibrary("FILEO.DLL");
if (hFileo < 32)
    {
    FreeLibrary(hFileo);
    MessageBox(GetFocus(),"Dynamic Link Library FILEO.DLL must be
                   present", szAppName, MB_ICONEXCLAMATION |MB_OK);
    return 0;
    }
```

This is the code that loads (possibly) the DLL into memory. I say possibly because if Windows finds the DLL already loaded, the currently loaded copy is used. If the load is successful, LoadLibrary will return a value greater than or equal to 32. This magic number is HINSTANCE_ERROR, and any   return value of this or lower is an error. On error, we report the reason to the user, and exit.

Before we exit our WinMain, we must free up the DLL. We are only concerned with cleaning up after ourselves. Windows will not allow the DLL to be removed if others are using it. That's what makes the system so nice to use:

```
FreeLibrary(hFileo);
```

WndProc is clean until we get to "case IDM_FILEOPEN":

```
case IDM_FILEOPEN :
       DoFileOpenDlg();
       lstrcpy(OutMsg, szFileName);
```

```
    break;
```

Instead of calling the dialog box procedure as we have done in the past, we are now calling a local procedure that will do that work for us. The code could have been inserted at this point, but for ease of reading, and comparison, it is cleaner to have it be isolated.

A change has been made in the reporting of the filename in that it is transfered locally through the global variable "szFileName". That will become clear shortly:

DoFileOpenDlg

This procedure is new to Hello, and as I said above, is the launcher for the DLL function. The mystery takes place in the first variable declared:

```
FARPROC lpfnFileOpenDlgProc =
                    GetProcAddress(hFileo,"FileOpenDlgProc");
```

This declares a far pointer to a function inside the DLL. GetProcAddress retrieves that pointer for us by name, given that we know the handle for the DLL. We know that because we loaded it, "hFileo". We also know the function name "FileOpenDlgProc", because we wrote that function two months ago. It just happens to be external to our program at this point. To execute that function, we do the following:

```
(*lpfnFileOpenDlgProc)();
```

The file name selected will be passed back via Hello.INI, and we retrieve that into szFileName, to be used above:

```
GetPrivateProfileString("FileO", "FileName",
                      "NoName", szFileName, 128, "Hello.ini");
```

The remainder of Hello.C is identical to the previous code with the exception of the Fileo.C portions removed.

Hello.DLG

Hello.DLG is now split between Hello.DLG and Fileo.DLG, but nothing is added or removed, other than the split.

Hello.H

Hello.H is split between Hello.H and Fileo.H.

Hello.DEF

Hello.DEF is identical to before with the exception of the exported FileOpenDlgProc function being removed.

FILEO

Fileo is a simple DLL and contains the primary pieces that every DLL contains:

LibMain and WEP

To simplify some of our code, and avoid any nastiness that may take away from the simplicity, static variables were defined for the filename and extension, and declared in the beginning of the Fileo code:

```
static HANDLE hInst;
static char szFileName[128];
static char szFileSpec[16] = "*.EXE";
static char szDefExt[5]      = "EXE";
static WORD wFileAttr         = DDL_DRIVES | DDL_DIRECTORY;
```

LibMain

The LibMain function is called by LibEntry, which is called by Windows when the DLL is loaded. The LibEntry routine is provided in the LIBENTRY.OBJ module. LibEntry initializes the DLL's heap (if a HEAPSIZE value is specified in the DLL's module-definition file) before calling the LibMain function.

This simply unlocks the data segment of the library (which is locked by the LocalInit call in LIBENTRY) and returns 1.

```
int FAR PASCAL LibMain (HANDLE hInstance, WORD wDataSeg,
                                WORD wHeapSize, LPSTR lpszCmdLine)
{
if (wHeapSize > 0)
    UnlockData(0);

hInst = hInstance;

return 1;
}
```

LibMain is the place to allocate any memory necessary for use in the DLL, and to do any setup functions.

WEP

WEP is called by Windows when the DLL is released, and provides a place for us to do cleanup before exiting. In Windows 3.1, WEP is no longer necessary, but if we are trying to write DLLS to be globally useful, WEP should be included.

```
int FAR PASCAL WEP (int bSystemExit) {
/*---------------------------------------------------------------
   get rid of things you allocated in the LibMain proc here
---------------------------------------------------------------*/

return TRUE;
}   /* WEP */
```

In the case of Fileo, WEP has no functionality.

Fileo Code

The code body for Fileo consists of the two functions "lstrchr" and "FileOpenDlgProc" from the old Hello.C file. "lstrchr" has no changes at all, and was simply copied here.

FileOpenDlgProc

This function was stripped straight from the old Hello.C, modified slightly, and renamed "DoFileOpenDlgProc". It no longer attempts to open the file selected, the name is simply shipped back to Hello.C via an ini file,"Hello.INI"

This is all accomplished through the one exported function, "FileOpenDlgProc"

FileOpenDlgProc

This function is small, and consists of the lines from the old Hello.C WndProc switch case:

```
lpfnDoFileOpenDlgProc =
                MakeProcInstance(DoFileOpenDlgProc, hInst);

DialogBox(hInst, "FILEOPEN", GetFocus(), lpfnDoFileOpenDlgProc);
FreeProcInstance(lpfnDoFileOpenDlgProc);
```

followed by the Hello.INI write of the filename:

```
WritePrivateProfileString("FileO", "FileName", szFileName, "Hello.ini");
```

Fileo.DEF

The DEF file will be recognizable, and has few changes from the EXE DEF file. As explained above, there is no stack, therefore no STACKSIZE. WEP must be exported, and our single exported function "FileOpenDlgProc" is there:

```
LIBRARY      FileO

DESCRIPTION 'FileOpen DLL'

EXETYPE      WINDOWS

STUB         'WINSTUB.EXE'

CODE         MOVEABLE PRELOAD
DATA         MOVEABLE PRELOAD SINGLE

HEAPSIZE     8192

EXPORTS      WEP              @1 RESIDENTNAME
            FileOpenDlgProc
```

Fileo.mak

The make file is cryptic as ever, but recognizable. The changes are few, and easy to spot.

Amazingly enough, that's all there is to writing and using a very simple DLL. I've glossed over many of the nasties, and have purposefully avoided others. The bottom line is we have a DLL that's useable and easy to read.

Make Files

I am providing 3 make files with this code. HELLO.MAK and FILEO.MAK are the Borland make files for Hello.EXE and Fileo.DLL. MHELLO.MAK is the Microsoft make file for Hello.EXE. Try as I may, I could not produce a working make file for Microsoft for FILEO.DLL. The Hello.EXE that I produce with either make file works with the Fileo.DLL I produce from Borland, but I ran out of ideas trying to produce a make file for Microsoft for Fileo.DLL. So, the proof is left to the student. The first one to send me one that works gets mentioned here next month.

ICONS

Did you really think I was going to leave and not mention last month's program? Unfortunately, I was so late getting my article in that I ran the magazine late, and my "Go Suns" hit the streets after the Bulls took the third in a row. Oh well, the idea still stands. Animated icons is something we don't see a lot of in Microsoft Windows. Building all the icon images is probably the reason why. I was going to take a shot at doing a sweep second-hand clock, and got bored after 15 seconds worth of images.

WinMain

The work for the animated icons starts in WinMain, where a timer is started:

```
SetTimer(hWndMain, ID_TIMER, 500,
            MakeProcInstance((FARPROC)TimerProc, hInst));
```

Windows allows up to 16 timers to be running at any given time. Normally, SetTimer would be checked for a return value, and if the value is NULL, display a message about too many timers.

The first parameter is the handle to the window whose procedure will receive the WM_TIMER messages.

The second parameter is the timer ID, to identify which of 16 possibles you now have running.

The third parameter is a word specifying the timer interval in milliseconds (1-65535). In our case, this is 500 msec, or 1/2 second.

The fourth parameter is the instance handle of this window.

InitInstance

Inside InitInstance, we now are loading an Icon, and this is the "WPJ" icon that will show on the group screen, and for short periods of time when iconized:

```
wc.hIcon             = LoadIcon(hInstance, "WPJ");   /* Generic Icon */
```

and this value is saved in a global:

```
hIconMain = wc.hIcon;
```

WndProc

The WM_PAINT message handler is altered to handle the icon painting:

```
    case WM_PAINT :
        if (IsIconic(hWnd))
            {
            BeginPaint(hWnd, &ps);

/*--------------------------------------------------------------
   Erase the background of the window with what's on the desktop.     This is so the desktop
bitmap will show through the transparent     areas of the icon.
--------------------------------------------------------------*/

            DefWindowProc(hWnd, WM_ICONERASEBKGND, (WORD)ps.hdc, 0L);

/*--------------------------------------------------------------
   Now draw the icon.
--------------------------------------------------------------*/
            DrawIcon(ps.hdc, 0,0, hIcon);
            EndPaint(hWnd, &ps);
            }
```

Notice that the icon drawn is passed by the generic handle "hIcon". This will allow us to change the icon image inside the Timer proc.

One more thing needs to be done inside WndProc, and that is how to handle the icon when it is dragged on the screen. Fortunately, Windows sends a message while in that state, WM_QUERYDRAGICON:

```
    case WM_QUERYDRAGICON:
        return((LONG)(WORD)hIconMain);
```

Consequently, when our WndProc receives that message, we return the default "WPJ" icon, for the duration of the drag.

TimerProc

The only function our timer proc does in this application is toggle the two bitmaps GO.ICO and SUNS.ICO:

```
BOOL FAR PASCAL TimerProc(HWND hWnd, WORD message, WORD wParam, LONG
lParam)
{
static BOOL which = 0;

if (IsIconic(hWnd))
    {
    if (which = !which)
        {
        hIcon = LoadIcon(hInst, "GO");
        InvalidateRect(hWnd, NULL, FALSE);
        SendMessage(hWndMain, WM_PAINT, NULL, NULL);
        }
    else
        {
        hIcon = LoadIcon(hInst, "SUNS");
        InvalidateRect(hWnd, NULL, FALSE);
        SendMessage(hWndMain, WM_PAINT, NULL, NULL);
        }
    }
}
```

Depending on the entry value of the static variable "which", one of the two icons is loaded, and the entire window (the icon area) is invalidated for painting, and a message is sent to our own WndProc to execute the WM_PAINT command, which we know from above will repaint the icon.

Why

All of this is entertaining, sort of, but not useful. I have simply shown how a timer works, and tried to take some of the mystery out by doing something stupid.

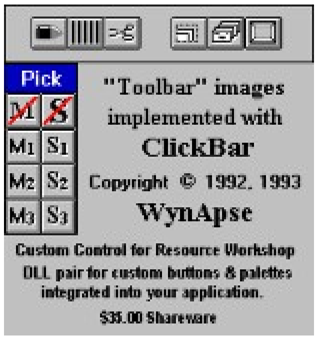
What if, instead of painting icons on a timer, we were painting windows? And, what if those windows were over the top of everything else, so that on a timer, the screen would suddenly get painted over. Why that sounds like a screen-saver! Well, kinda-sorta. We wouldn't want to do that on a hard timer, but it is close. Close enough that, given enough time, there will be the start of a series in the next issue called "Screen Savers Inside-Out". One more mystery unfurled.

EndDialog

That's about all there is to say about the code for this month. Unless something changes, I'm going to discuss common dialogs and show how to do all this the easy way next time. Feel free to contact me in any of the ways below. Thanks to those of you that have e-mailed me questions, keep them coming.

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

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



"Toolbar" images implemented with ClickBar Copyright © 1992, 1993 WynApse

Custom Control for Resource Workshop DLL pair for custom buttons & palettes integrated into your application.

$35.00 Shareware

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

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

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

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

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

Creating Help Files

By Todd Snoddy

Like most of the readers of Windows Programmer's Journal, I enjoy Windows programming. Although I don't currently have the blessing?/curse? of being a "full time programmer", I do consider myself to be a dedicated and serious Windows programming hobbyist. I would like to use the opportunity that WPJ provides to share with you some of the tools I use to increase my productivity. As my busy schedule permits, I am hoping to be able to review different programming utilities in upcoming issues.

I would like to be able to concentrate mainly on both shareware and free software. Although I don't have anything against commercial software, many of the software packages sold commercially are rather expensive, putting it out of the budget of most hobbyists or small business users. In addition, shareware has many advantages besides usually being less expensive. Many of the shareware products available fill a void that isn't quite covered by the major software publishers because they don't take the time to develop something that won't have mass market appeal. I'll try to point out some of these products in the future. Also, the quality of modern shareware can often approach or surpass that of commercial software.

My review topic this month will be Windows help files. I will cover a few tools that I use to develop my help files with, and actually present a sample help file that can be used as an example for your own help files. The tools that I will discuss are free, and are available from various sources, including Compuserve and numerous Internet ftp sites.

Before I get started with that however, I would like to take a brief moment to tell you a little about myself so that you can get an idea of my perspective. I am currently in the U.S. Army and stationed at Ft. Riley, KS. I work as a computer technician/soldier during the day, and in my free time I do a lot of Windows programming on personal projects. I am almost entirely self taught, and am competent in C, Pascal, and assembly. I have been programming Windows for around two years now, and C for about seven years. Prior to that, I programmed BASIC for several years. As mentioned before, programming is a major hobby of mine, but I also like to read and play pool, usually 9 ball.

Now to the good stuff. Currently, there is only one vendor who makes a compiler capable of creating Windows help files, and that is of course Microsoft. This help compiler takes a file that contains a special formatting language called Rich Text Format, or RTF, and converts it into a HLP file that can be used by the WinHelp engine included with Windows.

Contrary to what Microsoft would like to have us believe, it's not necessary to use an RTF word processor like their own (who's name we all know) to create Windows help files. Although the actual format of the HLP file is considered by many to be one of the best kept secrets in the industry, there are several low cost or free utilities available to generate the RTF file necessary for input to the Windows help compiler.

When I first wanted to create my own help files, I looked at several utilities to see what was available, especially since I didn't have several hundred dollars extra to buy an expensive word processor. The one that I found that was overall easiest to use is RTFGEN, by David Baldwin. This is an excellent utility whose use is free, not even shareware. With David's permission, an older command line version is included with this issue of WPJ, along with the Pascal source code, although a newer Windows version is available for free as well. For a small fee, you can even obtain the source for the Windows version.

RTFGEN's sole purpose is to convert standard ASCII text files into an acceptable RTF file recognizable by the help compiler. Using RTFGEN for creating your help files is almost like using a programming language like C.   Many people say that C is a high level language with the flexibility of a low level language like assembler.   RTFGEN will let you mix straight RTF code with your help source file for maximum control over the desired output, or you can use the default settings and just specify the actual topic information and let RTFGEN handle the rest.   RTFGEN has some special keywords that will actually be converted into RTF automatically, so you don't have to get your hands dirty if you really don't want to.

Creating a help file with RTFGEN involves a few steps.   First, you need to plan the layout of the help file.   The importance of this step can't be overstated.   A well thought out layout makes a big difference in the apparent quality of your product, especially when the user is totally lost and in a blind panic tries to bring up the online help instead of opening the manual.

The next step would be to create the help source files.   For this task, I use a free Windows editor called the Programmer's File Editor by Alan Phillips, but it can be done using any editor that reads and writes standard ASCII files, like DOS's EDIT command, or even the Windows Notepad.

The help source files contain the actual help topic information, including the topic text, keywords, and browse sequences.   These files will have the RTFGEN commands in them, and may also have RTF statements for added flexibility or control of the help file.   Each help topic will be in a specific format, and this will be covered shortly.

After creating the help source files, we are almost ready to compile them into a HLP file.   However, before this can be done, the source files need to be processed by RTFGEN.   This procedure converts the special RTFGEN commands into RTF automatically.   If you have any actual RTF statements in the source file along with the rest of the text, it will be left alone and passed on unmodified.   This allows the ultimate in customization of the help file.

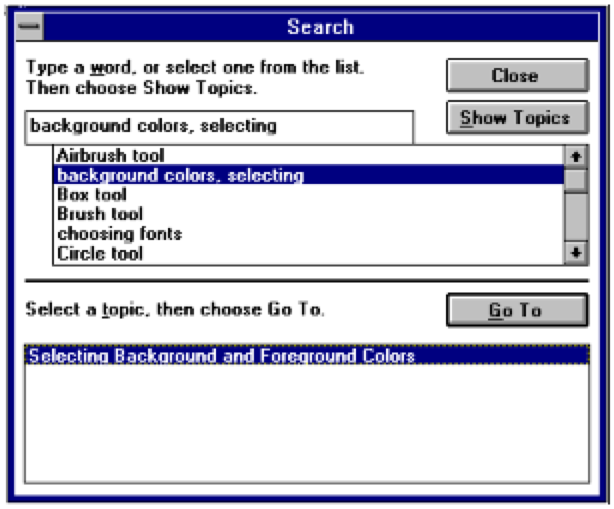
There is one final step required before actually compiling the help file.   This is to create a Help Project file, or HPJ file.   This file specifies certain options to the help compiler, including the source files for the help file, bitmaps, and any help macros used.   I won't cover this subject in more depth here as it is documented in other places like the help compiler documentation.   By the way, another excellent source of information on the format of the HPJ file as well as help file creation in general is the Help Author's Guide.   This is a help file created by someone at Microsoft but is not supported officially by them.   It is usually distributed as HAG.ZIP and can be found on Compuserve and Internet ftp sites such as FTP.CICA.INDIANA.EDU.   It contains a complete reference for the RTF command syntax.

That's all you need to create a help file.   That's all?   Well, as mentioned before, the help topics must be in a specific format.   David Baldwin's documentation for RTFGEN does a good job of explaining how to use it, but I'll briefly cover some of the highlights.

First, every RTFGEN command is started by a \ character, and most are terminated by a ` character.   For example, to specify a topic title, the command would be:

```
\title Selecting Background and Foreground Colors.`
```

This statement specifies what text will be displayed in the Go To list box when the user tries to search for the keywords "background colors, selecting".   See Figure 1 for an example.


Search
Type a word, or select one from the list.
Then choose Show Topics.
Close
Show Topics
background colors, selecting
Airbrush tool
background colors, selecting
Box tool
Brush tool
choosing fonts
Circle tool
Select a topic, then choose Go To.
Go To
Selecting Background and Foreground Colors
Figure 1

In Figure 1, the text shown in the upper list box is specified by the topic keywords. To get the currently selected text to be used for the shown topic, the following statement would be used:

```
\keyword background colors, selecting`
```

Every topic in the help file has to have some way to distinguish it as an individual topic and separate it from every other topic. This is known as the topic name. The topic name is never actually seen by the person viewing the help file, but is instead used as an internal marker to identify the individual topic to the help system. In RTFGEN, the topic is specified by a statement like:

```
\topic SelectingColors`
```

In the help source file, all of the above statements would be grouped together in the topic heading. The topic heading is separated from the rest of the topic text by using a minimum of five = characters at the end of the heading. In practice, this would look like:

```
\topic SelectingColors`
\title Selecting Background and Foreground Colors.`
\keyword background colors, selecting`
=====

This is the sample topic text for selecting a background and foreground color.

------
```

In the above example, the actual topic text follows the = character, and is terminated by five or more - characters. This is how RTFGEN determines the separation between topics, and based on their position, it will generate the appropriate RTF code for the help compiler.

RTFGEN can be used to generate hypertext jumps or popup text as well. For example, if you have some concepts in a topic that the user may not be familiar, you can designate certain keywords in the topic that the user can select on that brings up a popup window with a more detailed description or definition of the keyword.

To define a hypertext jump with RTFGEN, the text that will be selected by the user to jump to the other topic will be enclosed in brackets along with the name of the topic to be jumped to.   For example, Figure 2 shows a typical display of a hypertext jump.

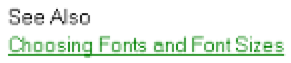

See Also
Choosing Fonts and Font Sizes

Figure 2

To duplicate this affect using RTFGEN, the following statement would be used in your help source file.

```
[Choosing Fonts and Font Sizes:ChoosingFonts]
```

In this example, the ChoosingFonts will represents the topic name that will be jumped to when the hypertext link is selected.

If instead you wanted to use a hypertext popup window for a definition, then you would use two sets of brackets.   An example of this is shown below.

```
picture elements ([[pels:pels_def]]).
```

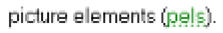

picture elements (pels).

Figure 3

Since I am a firm believer in the concept of explaining something and then giving an example of how to do it, a sample help source file is included, along with all of the required files to build it, except for the actual help compiler. You will need the Windows 3.1 version of the help compiler to build the sample.   After studying the sample files, as well as the documentation for RTFGEN, you should be well on your way to creating your own help files.

To compile the sample into a help file, use the command:

SAMPLGEN sample

This will invoke the SAMPLGEN.BAT batch file to convert your source file into RTF, and if everything goes ok, it will then run the help compiler.

There are many ways to customize your help files, and I may cover some techniques in future issues of WPJ as time permits, but if you want to learn some techniques on your own, beg, borrow, steal, or otherwise obtain a copy of the previously mentioned Help Author's Guide.   You can't beat the price since it's free, and it is full of useful information on RTF and creating help files.

I welcome any comments anyone may have on this article, as well as suggestions for other products to review in the future.   I can be reached on America Online at TSnoddy, on Compuserve at 71044,1653, and on the Internet at tsnoddy@nyx.cs.du.edu.   Although I'm very often quite busy, I will try to respond to any messages.

Also, if you do decide to start using RTFGEN, please contact the author and let him know what you think of it.   The world needs more software authors who are willing to make useful tools like this available for free, and when someone makes a contribution like that, receiving feedback from users provides a reason to produce other useful utilities too.   The author, David Baldwin, can be reached on Compuserve at 76327, 53.

Gdi See Gdi Do
Graphics Animation in Windows

By Bernard Andrys

In this series of articles, I will discuss the writing of a space invaders style game for Windows 3.1 in C. The articles will be directed to people who understand programming (such BASIC) and know a little about Windows but are new to Windows programming in C.

Experienced C programmers will immediately recognize the poor style of my code. In some cases, I will be doing this on purpose. If a reader is new to C AND Windows, they're probably going to have just as much of a problem following the C code as trying to follow the Windows specific areas of programming. (I know that I sure had problems learning both at the same time.) As a result, I will try to stick with easy to understand syntax like "x=x+1" instead of the C shorthand "++x". Later articles will use more of the shorthand notation as I and the readers get comfortable with C. The other reason for the bad style is that my coding philosophy is "Well it works... so I guess its good enough." This doesn't mean, however, that I don't want to learn how to write C better. If you are an experience C programmer, then please drop me a line about how I could improve my programming style.

What will be different is how we go about writting the program. I think that the best way to learn programming is to write a working program, no matter how poorly written, and then refine it as you learn more. So I will start with the standard "do nothing" windows program that just creates a window. Future articles will take this basic program and add the missle base, missle firing, invaders, invaders firing, and of course the ufo. From this simple program we add features and fix problems in future articles. Some of the improved features will be 256 color bitmaps and digitized sound. A couple of the problems that we will need to deal with are friendly multitasking and the efficient drawing of bitmaps. Once the game is finished, I would like to write about converting the program to C++.

If you are new to Windows programming in C then before you go on to read the program below, you'll need two books as references. The first is "The C Programming Language" by Dennis Ritchie and Brian Kernigan. The second is "Programming Windows" by Charles Petzold version 3.0. These two books have all the basics of C and Windows programming in C.

If you are knowledgeable about Windows programming in C then check out the "Duncan" format of my Windows C code. There's no case switch statement for processing messages. An array of structures contains the list of messages that our program will respond to along with the function that is called for each message.

A final note before the program starts: This program was written using Quick C for Windows. As far as I know, I didn't write anything that was compiler dependant (except the make file, but then I didn't write it, Quick C wrote it for me). If you have any problems compiling it, drop me a line so I can remove any compiler dependancies in future articles. I have been meaning to purchase Turbo C++ for Windows and/or Visual C++ so expect make files for one or both of those compilers in the near future.

```
// START OF PROGRAM LISTING

/* invid.c */

/*
There are two types of ways of adding comments into C source code.
The first way to add a comment is to put a   / *   before the comment
```

and a * / at the end of the comment. (I had to add a space
between the / and * because otherwise the C compiler would think that
it really was a comment. You can't put comments inside of comments.)
The other way to add a comment in your source code is to put a // at
the start of the comment. The comment will start at the // and go to
the end of the line.
*/

```
#include <windows.h>
#include "invid.h"
```

/*
The #include statement allows you to include another file in your source code. This saves the hassle of cutting and pasting the code in. The file that you include is put inside a pair of < > signs or a pair of " ". Using < and > tells the compiler to search through the path defined by an INCLUDE environment variable (use SET INCLUDE = path to include files). If you use " " instead, it tells the compiler to only check the current subdirectory. You normally use < and > for files that come with with the compiler. Quotes are normally used for your own include files. Every Windows program needs to include <windows.h>. C by itself doesn't have any functions for handling character input/output, character strings and other basic functions. Therefore most C programs will also include <stdio.h> and <stdlib.h> to have a bunch of functions to work with.

My own file "invid.h" contains my global variable declarations and function prototypes. You should already know that a global variable is a variable that all functions can share. In C, a global variable is created by declaring a variable above the function main() (or for Windows, the function WinMain()). Normally you put your global variables at the top of your program after the #include files (or in your own #include file). Global variables should normally be avoided because they make your program larger and make your code harder to manage as it grows.

Function prototypes are a list of the functions that are in your program. A function prototype includes the data types that the function will return and the data types of the parameters of the function. If you were to write a function called DrawPixel(), you would write:

```
int DrawPixel( int x, int y)
{
	...the function's code goes here
}
```

Int is the data type that the function returns. DrawPixel is the name of the function. X and y are the function's arguments with int x and int y saying that the arguments are of data type integer. To make a function prototype just copy:

```
int DrawPixel (int x, int y)
```

from your program, paste it at the top of your program (or put it in a separate #include file like I did with invid.h), and add a semi-colon to the end of the line.

The function prototype for function DrawPixel(x, y) is:

```
int DrawPixel (int x, int y);
```

The function prototype tells the compiler about your DrawPixel function before its used.

This lets the compiler check to see that you use your DrawPixel function with the right data types.
*/

```
struct decodeWord {
        WORD Code;
        LONG (*Fxn)(HWND, WORD, WORD, LONG); };

struct decodeWord messages[] = {
        WM_CREATE, DoCreate,
        WM_DESTROY, DoDestroy, } ;
```

/*
I have used a different method for having a Windows program respond to messages than what is normally seen in Windows programming books.   It is a method that I first saw used by Ray Duncan in PC Magazine. Instead of a large switch case statement, it uses an array of structures that list the messages that your program will respond to and the function that is called when that message is received.   (I'm going to skip covering the details of arrays and structures in C for now. Otherwise, I wouldn't have anything to write about in future articles!) This format allows you to easily add new messages without having to deal with a long and messy switch case statement.   To use the Duncan format, simply add the message that your program will respond to, a comma, the name of the function that will be called when you receive that message, and then another comma.   In the case above, if we wanted to have our program respond to a WM_SYSCOMMAND message (this is the message sent when you click on the system menu of a window) we would make the following change to the lines above.

```
struct decodeWord messages[] = {
        WM_CREATE, DoCreate,
        WM_SYSCOMMAND, DoSysCommand,   // the Windows Message, Our Function
        WM_DESTROY, DoDestroy, } ;
```

Now when Windows sends a WM_SYSCOMMAND message to our program, our program will run the function DoSysCommand.   (I could have called the function anything I wanted.)   We would then go to the bottom of the program code and write the function DoSysCommand().
*/

```
int PASCAL WinMain (HANDLE hInstance,
        HANDLE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow)
```

/*
This is start of the Windows program.   (A regular C program starts with a function called main().)   The keyword PASCAL tells the C compiler that this function will be called with the PASCAL calling convention.   This means that the code that is written inside of this function has access to the actual variables that are passed as parameters (i.e., hInstance, hPrevInstance, lpszCmdParam, nCmdShow).   A normal function in C doesn't let you change the original variable, it gives the function a copy of the variable to play with.

If the keyword PASCAL was NOT used above, then hInstance would contain a duplicate of the number that Windows uses to keep track of which copy of this program we have in memory. (In Windows, we can have the same program loaded several times.)   If we changed the value of hInstance, it wouldn't change what Windows thinks hInstance contains.   It would only change what our function thinks hInstance is.

Using the keyword PASCAL means that if we change the value of hInstance, it will change

```
what Windows thinks the value of hInstance is as well.   This can be very dangerous so you
should use care when dealing with the parameters to functions that are declared using the
PASCAL keyword.   So why use the PASCAL keyword?   There are three reasons for using the
PASCAL keyword.   The first is that when writing parts of a program, it's often easier to use
the keyword PASCAL to get access to the actual arguments than to have to deal with
creating a group of global variables or use a pointer to the arguements to get to the original
variable.   The next reason is that the PASCAL calling convention for functions makes
programs a little smaller and makes them run a little faster.   The last reason is that any
function that Windows will call in your program must be declare as a PASCAL function.   A
simple Windows program only has the functions Winmain() and WinProc() called by Windows.
Winmain() is the start of your program and WinProc() is the name usually given to the
function that Windows calls when it has a message for your program. A function that
Windows calls inside of your program is called a callback function and all callback functions
must use the PASCAL keyword.   I won't spend any time here discussing callback functions
because using them is a lot more messy than just adding the keyword PASCAL to the
function's declaration.
*/

{
	MSG msg ;
/*
Declare a variable msg of data type MSG (message).   This will be the variable that will hold
any messages that are sent to our program.
*/
	if (!hPrevInstance) InitApplication(hInstance);
/*
hPrevInstance is a number that Windows passes to WinMain as a parameter when run.   The
number identifies the previous copy of our program that is currently running.

If this is the only copy of the program running, hPrevInstance contains 0.   In C, the if()
statement executes the next statement if what is in the parenthese's is true (and in C, true
is anything except 0). The ! operator in C performs a logical NOT operation.   A logical NOT
operator changes a 0 to a 1 or a 1 to a 0.   Therefore if hPrevInstance is 0 (no previous
instance running) then !hPrevInstance equals 1 (True). So if !hPrevInstance is true then "if (!
hPrevstance)" runs the next statement which is InitApplication().   Once you've seen this a
couple of times, you'll automatically translate a statement like:

if (!hPrevInstance) InitApplication(hInstance);

to mean:

If there is not a previous instance of our program in memory then run the function
InitApplication().

InitApplication is a function that we will write to setup the information for this window that
can be shared between all copies of our program that are running.
*/
	InitInstance(hInstance, nCmdShow);
/*
Call InitInstance().   This is the function that sets up the information for this   program's
window that is unique for this particular copy in memory.
*/
	while (GetMessage (&msg, NULL, 0, 0))
	{
		TranslateMessage (&msg) ;
```

```
                DispatchMessage (&msg) ;
        }
/*
This is the message loop that keeps our program running.   Your program runs in this infinite
loop as long as the function GetMessage() returns TRUE (In C, TRUE is anything other than
0.)   The   GetMessage() function returns FALSE if a WM_QUIT message is received.   This
means that the while loop will stop if our program receives a WM_QUIT message and our
WinMain() program will continue to the lines below.
*/
        return (msg.wParam) ;
}
/*
This is the end of our Program.   What follows are the functions that WinMain calls and the
functions that we call when we receive a message as defined at the top of this program in
the array messages[].
*/


BOOL InitApplication(HANDLE hInstance)
/*
InitApplication will be the data that is shared between all instances of our program.
*/
{
        WNDCLASS wndclass;

        wndclass.style = 0 ;
        wndclass.lpfnWndProc = WndProc ;
/*
For this program WndProc() will be the name of the function that windows calls when it has a
message for our program.
*/
        wndclass.cbClsExtra = 0 ;
        wndclass.cbWndExtra = 0 ;
        wndclass.hInstance = hInstance ;
        wndclass.hIcon = LoadIcon (hInstance, szAppName) ;
/*
Load our icon for this program.   szAppName is a variable defined in invid.h.   (this is just a
short cut so that if I use this code to make a different program, I can just change the variable
szAppName in invid.h instead of having to go through all my source code to see where I
used the name of my program.)   Take a look at the file invid.rc for the rest of the
information about how you add an icon.
*/
        wndclass.hCursor = LoadCursor (hInstance, IDC_ARROW) ;
        wndclass.hbrBackground = GetStockObject (BLACK_BRUSH) ;
/*
the background of our window will be painted black.
*/
        wndclass.lpszMenuName = NULL ;
        wndclass.lpszClassName = szAppName ;

        return(RegisterClass (&wndclass));
}


BOOL InitInstance(HANDLE hInstance, WORD nCmdShow)
```

```
/*
The InitInstance() function will initialize the data that will be unique for a particular copy of
the program that is in running.
*/

{
        hWnd = CreateWindow (szAppName,
                "Invid",
                DS_SYSMODAL | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX,
/*
The flags DS_SYSMODAL etc. are constants defined in windows.h that tell windows how the
window should look.   An | symbol is used between the flags because it is the C operator for
logical OR.   If you have the binary numbers 1000 and 0100 then 1000 | 0100 = 1100.   So
by putting an | symbol between each flag, we combine the different flags into a single
number that Windows uses to figure out how the window should look.
*/
                CW_USEDEFAULT, CW_USEDEFAULT,
                WINDOW_WIDTH, WINDOW_HEIGHT ,
/*
A normal Windows program would use the Windows constants CW_USEDEFAULT above.
However, since this is a graphic game whose window cannot be resized (see DS_SYSMODAL
flag above), we are going to define precisely the size of the window in pixels.
WINDOW_WIDTH and   WINDOW_HEIGHT are constants that I've defined in "invid.h".
*/
                NULL,
                NULL,
                hInstance,
                NULL) ;
        ShowWindow (hWnd, nCmdShow) ;
        UpdateWindow (hWnd) ;
        return (TRUE);
/*
In <windows.h> TRUE is defined as 1 and FALSE is defined as 0.
*/
}

long FAR PASCAL WndProc (HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
/*
This is the function that Windows calls whenever it has a message for our window.   WinProc
goes through each item in the array messages[] (which was defined at the top of our
program) and checks to see if the message it received matches the message in the array.   If
the message does match, then it runs the function that goes with that message.
*/
{
        int i;

        for(i=0; i < dim(messages); i++)
        {
                if(wMsg == messages[i].Code)
                        return((*messages[i].Fxn)(hWnd, wMsg, wParam, lParam));
        }

        return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}
```

```
LONG DoCreate(HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
/*
This is where we will load our bitmaps and initialize some of our variables.   The program
doesn't do anything yet so there is nothing to create.
*/
{
return 0;
}


LONG DoDestroy(HWND hWnd, WORD wMsg, WORD wParam, LONG lParam)
/*
The program doesn't create anything yet so there is nothing to destroy before the program
exits.
*/
{
        PostQuitMessage (0) ;
        return 0 ;
}


// END OF PROGRAM LISTING
```

So that's it for this month.   Don't you just hate all these programs that do ABSOLUTELY NOTHING.   I sure do!   Unfortunately this article is already a bit long so you'll have to wait until next month before we turn this "do nothing" Windows program into a nifty little space invaders style game. If you'd like a peek at where I'm going with this article, run peek.exe. (You will probably need to play with the delay= value in the invid.ini file that is created in your Windows subdirectory after the program is run the first time.)

About the author:

(short version)
SWM 25 ISO SWF.   Enjoys long walks, sun, surf, skiing.

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

Book Review: Windows++

By Philip Sloss

I started trying to write Windows programs the old-fashioned way -- copying examples from magazines and books. This is still my preferred method when it comes to things I don't understand clearly. ("Things I Don't Understand Clearly" -- this is such an all-encompassing topic, that brevity and disk space prevent me from fully explaining it; however, you will find examples sprinkled throughout the rest of this article.) I do like trolling the book stores looking for interesting examples on Windows programming, and one of my first finds was a book called "Windows++."

"Windows++: Writing Reusable Windows Code in C++"

Often, you can find out an important thing in the preface to a book or article: what the book is about. The preface of "Windows++" states that it is "essentially a recipe for building a C++ class library interface to Windows." What really sold me on this book was that the author, Paul DiLascia, actually *shows* how to do this. Instead of finishing the book with a jumble of ideas that you have to put together, Paul goes the whole twelve yards, and shows how to put the library together. The end result is that "Windows++" is not just a book on how to build a class library, it is a class library.

Armed with (or, rather, disarmed with) a general naivete about Windows programming and a lack of enthusiasm for writing long repetitious code, I decided to use the Windows++ library to write Windows programs. The book and the library are mutually beneficial: not only does the library serve the book, but the book is the perfect reference for the library! When I'm having problems, I can see how something was implemented in the source code, and read a detailed explanation of it in the book. The book is also educational on Windows programming in general.

A couple of points: First, class libraries are very helpful and can be of great benefit. Microsoft's Visual C++ is permanently entwined in their class library, the Microsoft Foundation Class Library (MFC). MFC is the primary reason that Visual C++ is so useful. Second, and just as important, I think that one still has to spend time learning Windows the "C" way. There is no way around it, in my smug, slight opinion.

For example, I'll use someone I'm on speaking terms with -- me. Shortly after purchasing "Windows++," I went and got the bible -- "Programming Windows" by Charles Petzold. What happened with me was that I quickly began asking a lot of questions like "Well, how does this work?" (Compiler errors will cause one to speak out loud to no one in particular.) It helped to know how Windows++ calls were mapped to the Windows API, because then I had some clue as to what to expect. The general consensus is that Mr. Petzold's book is the best there is on Windows programming fundamentals -- I agree.

In other words, while I'm led to believe that class libraries like MFC and Windows++ can greatly simplify one's work, I still think most beginning Windows programmers are in for a future showdown with the Windows API.

"Hello, World"

The proliferation of the phrase "Hello, World" has extended beyond computer programming and now has completely taken over the information age. The standard introduction to just about anything -- renting a car, sky diving without a parachute (also known as sky jumping or sky plunging), underwater basket cooking -- is a square with the words "Hello, World" in the middle of it. On your first parachute-less sky dive, this is a good thing to aim for, as it will allow the authorities to easily locate your remains.

Windows programming adds a system menu, title bar, and maximize/minimize buttons to the "Hello, World" square. In addition, it requires a rather detailed preparation of structures and calls to functions with 423 parameters. It is also, apparently, the basis for beginning a Windows class library.

In C, the "Hello, World" program looks like this (adapted, with few alterations, from Petzold's "Programming Windows"; those of you who have seen this countless times before, please try not to become physically ill):

```
//////////////////////////////////////
#define STRICT
#include <windows.h>

LRESULT CALLBACK _export WndProc (HWND, UINT, WPARAM, LPARAM);

int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPCSTR lpszCmdParam, int nCmdShow)
{
     HWND          hwnd;
     MSG           msg;
     WNDCLASS      wndclass;
     static char szAppName[] = "Hello";

     if (!hPrevInstance) {
          wndclass.style          = CS_HREDRAW | CS_VREDRAW;
          wndclass.lpfnWndProc    = WndProc;
          wndclass.cbClsExtra     = 0;
          wndclass.cbWndExtra     = 0;
          wndclass.hInstance      = hInstance;
          wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION);
          wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW);
          wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
          wndclass.lpszMenuName   = NULL;
          wndclass.lpszClassName = szAppName;

          RegisterClass (&wndclass);
     }

     hwnd = CreateWindowEx(NULL,          // extended window style
                    szAppName,            // window class name
                    "The Hello Program",  // window caption
                    WS_OVERLAPPEDWINDOW,  // window style
                    CW_USEDEFAULT,        // initial x position
                    CW_USEDEFAULT,        // initial y position
                    CW_USEDEFAULT,        // initial x size
                    CW_USEDEFAULT,        // initial y size
                    NULL,                 // parent window handle
                    NULL,                 // window menu handle
                    hInstance,            // program instance handle
                    NULL);                // creation parameters

     ShowWindow (hwnd, nCmdShow);
     UpdateWindow (hwnd);
```

```
	while (GetMessage (&msg, NULL, 0, 0)) {
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}
	return msg.wParam;
}

LRESULT CALLBACK _export WndProc (HWND hwnd, UINT message,
					WPARAM wParam, LPARAM lParam)
{
	HDC		hdc;
	PAINTSTRUCT ps;
	RECT		rect;

	switch (message) {
		case WM_PAINT:
			hdc = BeginPaint (hwnd, &ps);
			GetClientRect (hwnd, &rect);
			DrawText (hdc, "Hello, Windows!", -1, &rect,
			     DT_SINGLELINE | DT_CENTER | DT_VCENTER);
			EndPaint (hwnd, &ps);
			return 0;

		case WM_DESTROY:
			PostQuitMessage (0);
			return 0;
	}

	return DefWindowProc (hwnd, message, wParam, lParam);
}
```

Again, all this gets you is a window that can be closed, with a line of text in it (although it does help pad out this article). And, for the most part, all programs have to have this code somewhere in them. This is where the class library comes in. A class library does repetitive things like filling the WNDCLASS structure and calling CreateWindowEx() with the correct parameters for you. You only have to define the styles and parameters specific to your application.

What "Hello, World" looks like using "Windows++"

For purposes of introduction, one of the first sections in the book shows what the Windows++ version of "Hello, World" looks like. It reduces the long, C-style program to this:

```
#include <wpp.h>	// main Windows++ header file

APPCLASS HelloWin : public WPMainWin {
public:
	HelloWin() { createWin("Hello"); }
	void paint(WPPaintStruct &ps) {
		WPRect clientArea = this;
		ps.drawText(clientArea, "Hello, Windows++.",
			DT_SINGLELINE | DT_CENTER | DT_VCENTER);
	}
};
```

```
void WPApp::main()
{
	mainWin = new HelloWin;
	run();
}
```

As the "Hello, World" example is a do-nothing program, this program has very little code -- a nice correspondence.   For a detailed explanation of how this is orchestrated, I strongly and emphatically suggest purchasing "Windows++."

What's the Difference?

What separates "Windows++" from some of the other books on C++ Windows programming is that "Windows++" is primarily about how the library fits together.   Often, all one gets is the source code to a class library at the back of the book and/or on disk and instructions on how to use it.   "Windows++" shows how anyone can build their own library -- it discusses some of the issues one has to deal with, and some of the standard and extended features one should consider incorporating in a class library.   At the same time, there are several instructive examples of real Windows programs written with the Windows++ class library.

And finally, in the last chapter, the reader is shown how to put the class library into a dynamic link library (DLL), and the issues and problems that any library has to deal with to put C++ classes into a DLL.

"How About a Sudden Ending?"

So, summing up: "long, redundant code -- bad, reusable C++ Windows classes -- good."   And a good book on how to make a C++ class library for Windows is "Windows++."

About the Author: An Unauthorized Autobiography of Philip Sloss

Born from a crude genetic experiment mixing the genes of a human sperm with tree sap, Mr. Sloss has led a far from legal existence since his grudging inception into the world. After being developed in an artificial womb, he was raised in a large culture of athlete's foot fungus.   Mr. Sloss was accepted into the aerospace engineering department at San Diego State University, and in May of 1992, having advanced to the rank of graduate, he left college and began a career in unemployment.   He is now under under psychiatric observation for "behavior unsuitable to inert objects."   Hospital officials report that Mr. Sloss' main hobby is spotting lint.

Book Review: ObjectWindows How-To
Author: Gary Syck
Publisher: Waite Group Press

By Rodney M. Brown

The other day, I was in my local WaldenBooks bookstore, browsing over the crop of C/C++ books. One book that caught my attention was "ObjectWindows How-To". This book offers hints/tips on programming in Borland C++/Turbo C++ with ObjectWindows.

The book is organized by problems, with the solution or "How-To" following. Each solution is graded in complexity; Easy, Moderate and Hard. After the solution is explained, it is followed with the author's C++ code. The C++ code is great for beginners. The author placed comments throughout the code explaining what each function does. The author also wrote a 2-3 page explanation of the code he wrote.

This book is a great reference. It is filled with solutions to printing, changing the appearance/color of windows, etc. Below is a list of all the problems that are explained in the book:

1. Making An ObjectWindows Application.
2. Changing the Class Information & Style of a Window.
3. Adding control windows to a window.
4. Adding menus to a program.
5. Creating an MDI application.
6. Making and Using DLLs.
7. Using the TDialog object class.
8. Making a file selection dialog.
9. Making a text search dialog.
10. Using text controls.
11. Using list box and combo box controls.
12. Using check box and radio button controls.
13. Using string tables.
14. Converting programs to another language.
15. Loading different menus from a resource file.
16. Adding and Deleting menu items.
17. Changing the attributes of a menu item.
18. Changing the check marks in menus.
19. Using bitmaps as menu items.
20. Creating pop-up menus.
21. Creating a text editor.
22. Getting input from the keyboard.
23. Getting input from the mouse.
24. Using the serial port.
25. Using the sound interface.
26. Handling the printer.
27. Drawing in a window.
28. Drawing different shapes.
29. Using bitmaps.
30. Using fonts.
31. Using metafiles.
32. Using the color palette.
33. Making 3-d dialog windows.
34. Creating custom controls.

35. Making a screen saver.
36. Making nonrectangular windows.
37. Using complex shapes as windows.
38. Creating a file object.
39. Saving different objects in a file.
40. Reading and writing parts of the file.
41. Creating a DDE server.
42. Creating a DDE client.
43. Getting updated data from the server.
44. Executing commands on the server.
45. Getting Drag-and-Drop files.
46. Creating an OLE client.
47. Creating an OLE server.

I hope I didn't bore you listing all of the subjects covered in the book. It is a great reference book for Borland C++/Turbo C++ ObjectWindows programmers. I predict that this book won't spend much time on my bookshelf.

Contacting me online:

CompuServe: 72163,2165
America Online: RodneyB172

You can also contact me at the following Internet addresses:

CompuServe: 72163.2165@compuserve.com
America Online: RodneyB172@aol.com

Book Review: Windows Programmer's Guide to Serial Communications
Author: Timothy S. Monk
Publisher: SAMS Publishing

By David Falconer

Book Details:
ISBN 0-672-30030-3
Includes Software Disk.
$39.95 USA

I have been searching for ages for a book which would teach me how to do Comms programming in Windows. Judging by the amount of questions in the Borland Windows forum on Compuserve, I am not alone in this quest. The book that seems to be most commonly recommended is Timothy Monk's "Windows Programmer's Guide to Serial Communications", so I bought a copy to see what it was like.

Perhaps I should explain a little about me, so you have an idea of the role the book was intended to play. I'm not a novice programmer, but not that experienced either, and I'm very new to Windows programming. Having worked my way through Petzold, I decided I was ready to tackle Comms, which is what I wanted to do in the first place! Also, I learn best by trial and error, and was looking for something to get me started and that would also act as a reference later on. I have not started putting the knowledge I have obtained from this book into practice yet - priorities change - but I thought I would share what it's like as a primer.

Firstly, this is not a primer for Windows programming in general. As a novice, I often found myself with Monk's book in one hand and Petzold in the other, but the Monk does provide little snippets of basic stuff that Petzold doesn't cover; connecting an application to a help file, for example. The other assumption that is made is that you have some familiarity with DOS serial Comms - at least the terms used.

The book is laid out in 7 chapters and 3 appendices, over the course of which you develop a fully working Comms program, called TSMTERM, starting with the very basics and ending up with an XMODEM file transfer protocol implementation. Each chapter starts with an overview of what will be covered, and finishes with a summary of what has been achieved. There are code segments Everywhere, and these are as much an integral part of the text as the writing between them is; often it is from the code that you find exactly how a particular function, or technique, is implemented. Fortunately all the code, plus fully compiled executables are provided on the accomanying disk, which saves on a lot of typing! Each time the code is modified, advanced upon in the text, a completely new set of source code is provided on the disk. This means a hefty 3 Megabytes of memory is needed for the files, but it saves getting confused over which version of which bit of the program you should be working with at any one stage. The areas that the chapters cover are:

Chapter 1: Serial Communications

This chapter contains an overview of serial Comms procedures, starting with the hardware; the RS-232 link, the 3 UARTS commonly found on the PC (8250/16450 and the newer 16550) in quite a bit of detail and finally the COM ports and their adresses and interrupts. Monk then goes on to discuss the principles of DOS Comms programming, with code segments, including a look at ring buffers. The final part of the chapter discusses what is going to happen in the rest of the book, by way of a "design brief" for the TSMTERM program.

Chapter 2: Windows Communications Basics.

This is basically in two major sections. In the first, Monk outlines the 17 functions of the Windows Communications API, which are fully documented in Appendix A, and shows how these can be used in a typical Comms session of open port, send/receive characters, close port. These initial examples use polling of the port in the main message loop of the Windows application as the method of getting information from the port into the application. Message-based methods, which were introduced with Windows 3.1, crop up later. In the second half of the chapter, Monk creates his own API for use with the TSMTERM program, in order to simplify some of the trickier aspects of using the raw function calls. These are then all encapsulated in a DLL. This is done very matter-of-factly, but the code for the LibMain and WEP functions are provided and briefly explained.

Chapter 3: Terminal Emulation

Again, this chapter is in two sections. The first part covers terminal emulations, including the full specification of an ANSI terminal, which is then implemented in a DLL. There is also a word about finite state machines, on which principles the ANSI emulation is based. The second part starts assembling the application from the components created so far. The processing of the basic Windows messages is covered, as is the Windows procedure, although this is of quite a complex variety, covering menu commands as well as the more straight forward messages. Knowledge of Dialog Box methods are assumed, but I learned quite a lot about their implementation from studying the code. This is a fully fledged application, and so includes an "about" box and interfaces to a basic help file. As well as the ANSI implementation, the program also allows the user to choose a basic TTY emulation.

Chapter 4: Errors, Events, and Flow Control

The good practice of calling GetCommError( ) after every read, write or port poll is introduced and TSMCOMM.DLL, created in chapter 2, is modified to include error processing. A similar process is then gone through for communications events using the GetCommEventMask( ) function and its associates. After both modifications to the DLL, the main application is also modified to give the user a choice of how to monitor errors and events. Finally there is a brief discussion of XON/XOFF and hardware flow control, after which both methods are added to TSMCOMM.DLL. Again the main application is modified to give the user a choice of what to use.

Chapter 5: Message-Based Communications

In Windows 3.1 Microsoft introduced the EnableCommNotification( ) function which adds a new message to the Windows system. This is as close as Windows gets to interrupt driven communications procedures, as the Comms system is interacted with in response to a received message rather than having to be polled in the message loop. Chapter 5 gives an overview of how
this system works and then adds the facility to TSMCOMM.DLL, and modifies the application to give the user the choice of system.

Chapter 6: Modem Communications

Chapter 6 starts with a look at where modems fit into serial communications systems. The command set for Hayes modems is then examined and a modem API is implemented in MODEM.DLL, which includes setup, dial and hangup routines. A modem commands section is then added to the main application. As an extention of this, a dialing directory is implemented, where the user can store the names of systems, their phone numbers, plus the port settings: Baud rate, start, stop and data bits. This too, is then added to the main program.

Chapter 7: File Transfer

To enable the TSMTERM program to download files from the bulletin boards whose numbers are stored in the newly created address book, Monk adds the popular XMODEM file transfer protocol. The chapter starts with a discussion of the protocol; the packet, error checking, transmission, reception, errors and timeouts. The protocol is then implemented in another DLL, and has options for using checksum or CRC error correction. These functions are then added to the main program. Selection of the file for transmission is carried out by use of the Common Dialog Box "Open File", so use of this feature of Windows 3.1 can be followed by examining the source code.

Appendix A: Windows Serial Communications API Reference

As mentioned above, this provides the documentaion for the functions and structures provided by Windows for communications systems. It includes the Windows 3.1 amendments and marks where the Window 3.1 implementation is different from 3.0.

Appendix B: TSM Communications API Reference.

This provides a full function by function description of Monk's own API, as used in the TSMTERM program. It describes the functions created throughout the book which are in TSMCOMM.DLL, MODEM.DLL and XMODEM.DLL. It is a useful summary and reference section.

Appendix C: ASCII Character reference.

This is a table in two parts. Part one gives the decimal value, hex value, display character and description for all of the 7 bit basic ASCII characters. Part two provides the same information for the extra members of the extended PC set.

I found the book helpful and easy to follow. The layout is a bit strange, with funny bits of ornamentation around the edges of the pages, but you soon get used to it. It goes from communications basics to quite an advanced program in only 413 pages (including the appendices) so it obviously doesn't hang around when explaining things to make sure you have got a particular point. But that's the nice thing about books; if you find you have missed something, you can always go back and re-examine it.

As a practical introduction to Windows Communications, I would heartily recommend this book. There may be others that do the job too, but if there are I haven't found them and what I see here, I like.

I confess that I haven't used the program, TSMTERM, and nor am I likely to; that's not why I bought the book. I wanted something that dissected a Windows Comms program, so I would know how to do it for myself, and that's what I seem to have got. As I mentioned at the outset, my own project has been "back burnered" for a while, but I have no qualms about tackling the Comms
aspect head on.

Dialog boxes are another matter though. Ah well, back to Petzold...

The author can be reached on CompuServe at 100063,1205.

Letters

Date: 22-Jun-93 09:10 EDT
From: Scott Guthrie [100060,471]
Subj: Install Program Prob with In use DLL's

Mike and Pete,

I just today downloaded issues 2-5 of your Journal, and I must say I'm impressed. Unfortunately, you seem to have ditched the Install program for a variety of reasons, only one of which I may know the answer to. Microsoft provides an exportable function in the USER.EXE module of Windows 3.1 called ExitWindowsExec which will exit Windows and run the executable indicated in the call and then (when the executable releases control of the processor) RESTARTS Windows. The executable it runs can include up to a 128 byte Path and file name, and an additional 128 bytes of parameters, switches, etc. The documentation I have read for this function says that it was included in Win3.1 precisely for the use of SETUP/INSTALL programs to use when they need to modify code that is currently in use by Windows (e.g., DLLs). Sorry I wasn't quicker off the mark on this, but I'm not even sure it is going to help you, as I haven't been able to get it to work precisely as advertised.

Cheers,
Scott Guthrie

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)

You can also send paper mail to:

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

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

The Last Page
by Mike Wallace

I got this in the mail a few days ago from Mike Strock...

Here is a wonderful quote I think you will just love.   It comes from the Journal American, page d6, in the 2nd article, 3rd paragraph in the 4th column.

"Electronic bulletin boards are fast becoming a popular way to get copies of programs for free, but the practice is illegal.   Bulletin boards are computers that can be reached by other computers via modem.   Most of them require the new users to upload an illegal program onto their machines (thereby incriminating themselves) before they are allowed to down-load other programs for free, said Alison Gilligan, international anti-piracy specialist at Microsoft."

Bulletin boards have been blamed for spreading illegal copies of software almost since the day they started appearing.   Software companies have long complained their very existence is in danger due to these BBSs spreading free copies of their software.   Is this fair to the majority of BBSs?   No, I don't think so.   I've been on a lot of boards, and most of the ones I've seen aren't doing anything illegal.   Maybe you need to know someone to get onto the illegal boards, but that doesn't legitimize the above quote from Microsoft.   The trend in this country to blame others for your own "bad luck" gets out of hand sometimes.   Say you start up a commercial software company and nobody buys your app.   Does that mean everyone has a free copy obtained from a BBS?   Maybe, but there's also a chance your program isn't worth buying, or people don't even know about it.   It's hard to buy every commercial program you want when each one may cost $200-300, or more.   The software companies say this is to recoup their losses from illegal (unpaid) copies floating around.   I think this may have the opposite effect - who can afford to buy software that's so expensive? Now, I'm not condoning software piracy.   Besides being illegal and immoral, I think everybody loses when nobody's paying for their software.   The concept of "shareware" should be extended to commercial software.   Will this ever happen?   Never mind whether it's even feasible.   I would probably own more software if I could try it out first and then pay for it if I decide I'll use it.   But as it is, I can't afford to pay big bucks for a program that I may never use once I've tried it out.   So, the result is there's probably a lot of good software for sale that would make my life easier, but I'll never know about it.   The same can probably be said for a lot of computer users.   Hmmmm...sounds like everybody's losing again.

A lot of these companies only appear to be in the business of manufacturing software so they can make a lot of money, and I think the people putting out quality software are. Take Borland, for example.   They put out a lot of good programs, and they're reaping the benefits.   Microsoft too.   Why they're complaining about BBSs is beyond me.   Let's look at one BBS and see how Microsoft has benefited from it: the WPJ BBS.   Besides carrying WPJ, there's a lot of other software (all legal) available from a CD-ROM that can be accessed by anybody that calls our BBS.   Most of it is devoted to products produced by Microsoft (e.g., Windows), and we're just part of a much larger network of BBSs all over the world supporting Microsoft.   True, a lot of commercial software companies live off Microsoft by writing apps helping people run Windows, but it goes both ways: Microsoft benefits by having so many companies support their products with add-on programs or whatever.   If I was going to choose an operating system for my computer, one thing I would look at would be the third-party support, and Microsoft is the overwhelming leader in that area for IBM-compatible PCs. So, to hear Microsoft complaining about BBSs is more than a little annoying.   Sounds like it's biting the hand that feeds it.

Hacker's Gash

By Dennis Chuah

This article contains a few of the tips, tricks and traps from my Windows Programming laboratory book. As I mainly use Borland C++, most of the discussion here will be slightly biased to Borland C++. However some issues also concern Windows programming in general.

It is sad that the Windows API documentation[1] leaves out certain bits of information that is sometimes crucial to the correct usage of the API. Take MakeProcInstance for example. The documentation states that MakeProcInstance must only be used to access functions from instances of the current module. This implies that MakeProcInstance will create a procedure instance given a hInstance and an exported procedure address provided it is called from the task that is associated with the hInstance. This is not always the case. Take a look at the following code extract:

Somewhere in the application ...

```
void CALLBACK SomeProc (void);
.
.
FARPROC lpfnSomeProc;
lpfnSomeProc = MakeProcInstanceForMe (hInstance, (FARPROC) SomeProc);
if (lpfnSomeProc == SomeProc) MessageBeep (0);
```

And somewhere else in the applications DLL ...

```
FARPROC WINAPI MakeProcInstanceForMe (HINSTANCE hInst, FARPROC lpfnFarProc)
  {return MakeProcInstance (hInst, lpfnFarProc);
  }
```

MakeProcInstanceForMe will return the address of SomeProc instead of the procedure instance. This is because MakeProcInstance will **always** return the same procedure address passed to it when it is called from a .DLL, regardless of what hInstance it is passed, a point missed out in the documentation. Note: the documentation merely states that

MakeProcInstance   is not required for library modules, hence:

Trap: Never use MakeProcInstance to create a procedure instance from a .DLL.   Always create the procedure instance first before passing it to a .DLL.

Tip: Use smart callbacks prologue and epilogue code if your compiler supports it.   This usually assumes DS is equal to SS.   For most applications this is true, except if you swap data or stack segments.   Callback procedures compiled with smart callbacks do not need procedure instances to bind the data segment.   The compiler just copies SS to DS in the prologue code.   As there is no need for procedure instances, there is no need to list it in the EXPORTS section of the module definition file.

Tip: Callback procedures in .DLLs need not (and should not) use MakeProcInstance to create procedure instance.   The compiler always assumes that DS is not equal to SS and there is only one data segment.   It will load the correct data segment.

Still on the subject of Windows API, there is a very little known (and very little documented) header file in Borland C++s include directory called WINDOWSX.H.   A wealth of macros can be found in this header file.   For example, the GlobalAllocPtr macro API allocates memory on the global heap, locks it and returns a pointer to the locked memory. This is similar to the malloc function that C programmers are more familiar with.   Using GlobalAllocPtr also saves from having to keep track of a handle.   Simply call GlobalFreePtr when the block of memory is no longer required.   Note: Locking memory has its history in the real mode version of Windows.   In protected mode, locking memory no longer has any meaning so it is all right to keep a block of memory locked for its lifetime.

Trick: The way GlobalAllocPtr is defined, it can cause the compiler to generate a Code has no effect warning.   To get around this, place a void typecast over GlobalAllocPtr when you call it.
For example:

```
DWORD far *lpDword;
   .
   .
lpDword = GlobalAllocPtr (GHND, sizeof (DWORD) * 20);
   .
   .
if (lpDword != NULL) (void) GlobalFreePtr (lpDword);
```

WINDOWSX.H also contains declarations for the Control Macro API.   Using this API instead of sending control messages makes your code easier to read.

```
Edit_SetTabStops (hEdit, nTabs, lpTabs);
```

is more legible than;

```
SendMessage (hEdit, EM_SETTABSTOPS, (WPARAM) nTabs,
    (LPARAM) (const int far *) lpTabs);
```

In addition, WINDOWSX.H also contains declarations for message crackers.   By using these macros, you will make your code more portable.   I have one criticism though, the macros are not defined to allow a pointer to be passed to the message handler functions.   If you havent used WINDOWSX.H before, I strongly encourage you to start now.

Metafiles:

A metafile is a series of GDI calls. It can be stored as a physical disk file or represented by a handle to a metafile (stored in memory).

Tip: Use a drawing package (such as Corel Draw) to generate metafiles that can be included as a user-defined resource in your Windows application. This way, you can draw graphics using the powerful drawing tools of such packages and include the graphic in your application.

Tip: Including a metafile as a user-defined resource in an application:
Say the metafile name is METAFILE.WMF.

... somewhere in the resource header file (say RES.H):

```
/* assign an ID value for metafile resource types */
#define METAFILE            2000
#define MYMETAFILEID   100
```

and somewhere in the resource definition file:

```
#include "res.h"
   .
   .
MYMETAFILEID METAFILE "metafile.wmf"
```

To use the metafile:

```
#include res.h
   .
   .
HGLOBAL handle;
HMETAFILE hMetafile;
HDC hDc;
RECT rect;
   .
   .
/* Load the metafile */
handle = LoadResource (hInstance,
FindResource (hInstance, MAKEINTRESOURCE (MYMETAFILEID),
                                    MAKEINTRESOURCE (METAFILE)));
/* no need to retrieve pointer as we only need the handle */
LockResource (handle);
hMetafile = (HMETAFILE) SetMetafileBitsBetter ((HMETAFILE) handle);
UnlockResource (handle);
/* Then draw it */
hDc = GetDC (hWnd);
GetClientRect (hWnd, &rect);
SetMapMode (hDc, MM_ANISOTROPIC);
SetViewportExt (hDc, rect.right, rect.bottom);
PlayMetaFile (hDc, hMetafile);
ReleaseDC (hWnd, hDc);
/* Clean up */
DeleteObject (hMetafile);
FreeResource (handle);
```

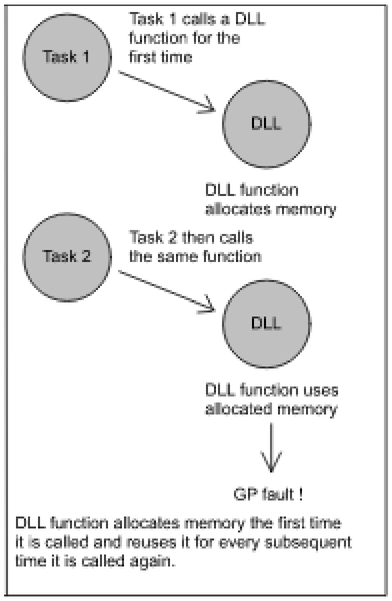
It is important to set the mapping mode to MM_ANISOTROPIC and the viewport extents to rectangle in which the metafile is to be drawn. This ensures that the whole metafile is drawn inside the specified rectangle. However, some metafiles behave badly and will draw outside the specified rectangle. Most metafiles set the window extent and origin, and draw within the limits of the coordinates specified. However, this behavior cannot be guaranteed for all metafiles.

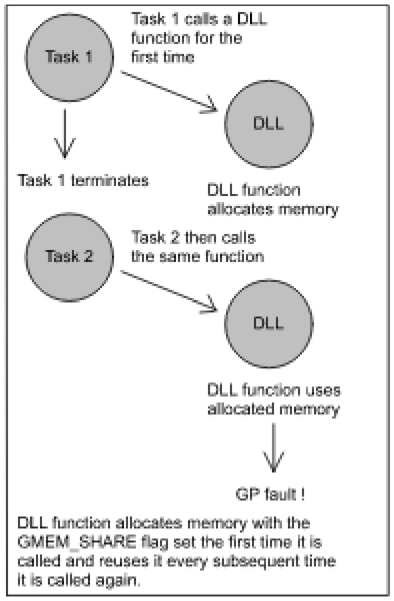
Trap: Forgetting to call LockResource can cause SetMetafileBitsBetter to return NULL.

Trap: Metafiles with placeable headers cannot be processed by Windows so make sure you save your metafile without the placeable header.

Miscellaneous:

Trap: Be very careful when allocating memory in a .DLL. Memory allocated by a .DLL belongs to the application that called the .DLL. Passing the handle or pointer to the block of memory to another application will most likely cause a GP fault. If a .DLL needs to share memory between applications, use the GMEM_SHARE flag when calling GlobalAlloc. This, however, poses another problem. Memory blocks allocated with the GMEM_SHARE flag, although can be shared, still belongs to the application that allocated it and will be freed when the application terminates. If another application calls the .DLL to access the memory after the application that allocated the memory terminates, a GP fault will occur. The following diagram better illustrates this.


Task 1
Task 1 calls a DLL function for the first time
DLL
DLL function allocates memory
Task 2
Task 2 then calls the same function
DLL
DLL function uses allocated memory
GP fault !
DLL function allocates memory the first time it is called and reuses it for every subsequent time it is called again.


Task 1
Task 1 calls a DLL function for the first time
Task 1 terminates
DLL
DLL function allocates memory
Task 2
Task 2 then calls the same function
DLL
DLL function uses allocated memory
GP fault !
DLL function allocates memory with the GMEM_SHARE flag set the first time it is called and reuses it every subsequent time it is called again.

Tip: So, the lessons here are;
a) if possible, let the application allocate memory and pass the handle/pointer to the .DLL,
b) if the .DLL must allocate memory, make sure it is used only for the application that caused it to be allocated,
c) if the .DLL has to allocate memory that will be shared between applications, write a small application that allocates memory on behalf of the .DLL. This application has to be active for at least the life of the .DLL. (See accompanying article -- **Allocating Shareable Memory**) (In next month's issue - Ed.)

Notes:
[1] - I define Windows API documentation to be information documented in Windows API vols 1-3. I know there are other sources of information, such as the numerous articles from Microsoft. However, it is my opinion that any information that is not included in a functions entry in the Windows API manuals should be classified as undocumented. If the set of

manuals are to be the primary source of reference material for Windows programming, it would be safe to assume that information documented elsewhere is not widely accessible to Windows programmers and therefore undocumented.

The author:
I am currently doing a PhD. degree in Electrical And Electronic Engineering at the University Of Canterbury.   My programming experience dates back to the days where real programmers code with Z80 machine code.   For the past two years I have taken an interest in Windows programming.
Please send any comments, questions, criticisms to me via;
email: **chuah@elec.canterbury.ac.nz**
or post mail: **Dennis Chuah**
c/o The Electronic and Electrical Engineering Department
University of Canterbury
Private Bag
Christchurch
New Zealand.
All mail (including hate mail) is appreciated.   I will try to answer all questions personally, or if the answer has general appeal, in the next issue.   If you are sending me email, please make sure you provide me with an address that I can reach (most internet and bitnet nodes are reachable, and so is CompuServe).

Pascal in 21 Steps
(A Beginner's Column for Turbo Pascal for Windows)

By Bill Lenson

Step Two - Divide and Conquer

REVIEW

Last month we talked about how to write a simple Pascal program. We showed you what variables and commands are. We also covered basic data types like integers, characters, and strings. Even though the examples where quite simple and didn't do anything practical, the purpose of that column was to introduce the basic program structure.

This month we pick up the pace quite a bit. By the end of this column you should be able to read input from the keyboard, decide if the input is valid, repeat actions over and over, and break your code up into chunks called functions and procedures. All this and we'll even show you how to write a simple calculator.

KEYBOARD IN / SCREEN OUT

Reading information from the keyboard is not that difficult. There are some tricks to it though to make your life easier. Last month you used the 'writeln' statement to write text to the screen. To read text you use 'readln'.

Readln (pronounced 'read line') pauses execution of your program until you type something and press the <Enter> key. For example,

```
Readln(s);
```

lets you type in some text and when you press <Enter>, puts it into the variable s. s can be any data type but some are better than others. If the s 'parameter' to the readln 'procedure' is an integer then you are only allowed to enter numbers. If it's a string type you can enter anything. When I say 'only allowed to enter a number', I mean that you can type in non-numbers but the program will crash when you press <Enter>. This is bad. Let me illustrate.

Type in the following program.

```
program Bad_Code;
uses WINCRT;
var
    num : integer;
begin
    readln(num);
    writeln(num);
end.
```

When you run this program a window will be created and the cursor will flash. Type in the number 23 and press <Enter>. Sure enough, 23 gets assigned to the num variable and gets displayed in the window by the writeln(num) statement. Now, rerun the program but type in Hello and press <Enter>. CRASH. Pascal tries to convert Hello into an integer so it can store it into the num variable. Since it doesn't know how to do this it bombs. It's a poor program that lets the user crash it so easily. Try to make the next program blow up.

Bet you can't (poor bet, there is a way but I'm not going to tell you how until next month).

```
program Better_Code;
uses WINCRT;
var
    numstr : string;
begin
    write('Please type a number and press <Enter> : ');
    readln(numstr);
    write('The number you typed was : ');
    writeln(numstr);
end.
```

No matter what you type in the program spits it back onto the screen. The only thing limiting in this code is the programmer has no way to control what the user types. You can't be sure the user types a number in the above example. We'll show you how to do that in the next section.

DECISIONS, DECISIONS

Quite often you want a program to do one thing when a certain input is typed in and do something else when another thing is entered. The most common way to do this in Pascal is the IF statement.

Before I show you the IF statement, let me first show you how you can convert a number stored as a string into an integer data type. Why we need to do this will become clear shortly. Try the following code.

```
program String_2_Number;
uses WINCRT;
var
    numstr : string;
    num    : integer;
    errcod : integer;
begin
    write('Please type a number and press <Enter> : ');
    readln(numstr);
    val(numstr, num, errcod);
    writeln('The number you typed was : ', num);
        { Notice we combined two write statements into one by passing }
        { two parameters separated by a comma.                         } end.
```

This program is similar to those above but if you don't enter a number a zero is printed instead of what you typed in. The val procedure tries to convert a string into a number, and if it can't it puts a zero in num and a number greater than zero in the errcod parameter.

Another thing to note about the above program is the stuff enclosed in { and }. These are comments and allow the programmer to describe what is happening in the code. All programs should have comments that let other programmers (or yourself) look at a program and tell what the code does. You can put anything in a comment as long as it starts with a { and ends with a }. As an aside, sometimes you'll see older Pascal programs with comments that start with a (* and end in a *). These work too but the convention these days is to use the braces.

I've been talking for a while about numbers and strings and how we want a number but we need to enter a string and then convert it to a number. What's all the hubbub about numbers and strings? Why not just leave them as strings? The answer is in the operations you can do to different data types in Pascal. For instance, you can't add two numbers together if they're stored in a string. Also, you can't concatenate (stick one string on the end of another) two numbers together if they're stored in integer types. Pascal is a typed language meaning that data is always stored in variables of a certain type. Each type has it's own set of built-in operations. Some string operations convert a string to upper case letters from lower case, extract some characters from the middle of a string, convert a string to a number, and so on. Most integer functions are math based such as adding two numbers, subtracting two numbers, multiplying two numbers, and so on.

This section is about Pascal decision making and program flow control using the IF statement so let's give a simple example.

```
program Better_String_2_Number;
uses WINCRT;
var
    numstr : string;
    num     : integer;
    errcod : integer;
begin
    write('Please type a number and press <Enter> : ');
    readln(numstr);
    val(numstr, num, errcod);
    if (errcod <> 0) then
        writeln('ERROR: The number you entered was invalid.')
    else
        writeln('The number you typed was : ', num);
end.
```

The IF statement needs only a little explanation. First the '<>' in Pascal means 'not equal to'. Also, there are no semicolon (;) before else's since the whole if..then..else combination makes a complete statement and last month we said that semicolons only come at the end of a statement (ugh!). So, the above is described as, 'If errcod is not equal to zero then write an error message else write the number entered to the window'.

In the above, the stuff (I didn't say this was fine literature) between the IF and the THEN is called the 'condition'. Conditions are logical statements where we compare things. The things we compare are usually either two variables or a variable and a value like we do in the above code. Let's assume we have two variables var1 and var2. Some simple 'logical operations' we can do are:

Logical Operation	Tests if...
(var1 <> var2)	==> var1 not equal to var2
(var1 = var2)	==> var1 equals var2
(var1 > var2)	==> var1 is greater than var2
(var1 >= var2)	==> var1 is greater than or equal to var2
(var1 < var2)	==> var1 is less than var2
(var1 <= var2)	==> var1 is less than or equal to var2

We'll use logical operations with IF statements many times in the future so don't worry if they look a little strange right now. Let's now move on to see how we can repeat something over and over.

LOOPS

The process of repeating something in Pascal is called looping. There are four ways to loop but only two of them are commonly used, one seldom used and one is absolutely forbidden to be used. The last two will be covered at a later time. This section presents the first two: the FOR and WHILE loops.

The FOR loop repeats some block of code a fixed number of times. Let's give a simple example:

```
program For_loop;
uses WINCRT;
var
    x : integer;
begin
    for x := 1 to 50 do
        write('Hello ');
end.
```

This program writes 50 'Hello''s on the screen. It's much simpler than typing out 50 write statements. The FOR statement is read 'for x equals one to fifty do write hello'. In other words, x is assigned the number one. The next statement is executed (the write). x is increased by one and checked to see if it's greater than 50. If it is, the FOR statement is finished and the statement after the write is executed. If it isn't, the write is executed and then x is increased by one and checked to see if it's greater than 50. And so on. In other words we repeat the write 50 times while we increase x from 1 to 50. On the 51st loop, the FOR is exited.

Let's try another.

```
program For_loop_2;
uses WINCRT;
var
    x : integer;
begin
    for x := 1 to 10 do
        writeln(x);
end.
```

This example loops 10 times and writes the contents of x each time through the loop. In other words, the numbers 1 to 10 are written on the screen. The starting number doesn't have to be 1, we could have said:

```
    for x := 5 to 15 do
        writeln(x);
```

which would write the 10 numbers from 5 to 15 on the screen.

The WHILE loop is a little different. WHILE will repeat until some condition is met. For example:

```
program While_loop;
uses WINCRT;
var
     x : integer;
begin
     x := 1;
     while (x <= 10) do begin
          writeln(x);
          x := x + 1;
     end;
end.
```

This program acts just like the For_loop_2 program above. In it we assign the value one to the x variable. Next, while x is less than or equal to the value ten (it starts at one so it is less than 10) repeat everything between the begin and end statement. What's between the begin and end is the meat of this WHILE loop. In it we write the x variable to the screen and then we add one to x. That adding one to x looks a little funny. We say 'x is assigned the value of x plus one. In other words, we do the stuff on the right of the ':=' and then put the result in the variable on the left of the ':='. If we said 'x := x + 2' we would see every other number from 1 to 10 (i.e., 1, 3, 5, 7, 9).

FOR loops are often used when we know in advance how many times we will be looping. WHILE loops, on the other hand, are used when we don't know how many times we'll be looping. In common programming practice, we would typically use FOR loops to initialize array variables (a future topic), or repeat something that never changes in the number of iterations. WHILE loops are commonly used for reading input from users or files. For example, reading keyboard input until the user types some exit command the program knows about. We'll illustrate each looping technique in more detail next month.

HINDSIGHT is 20-20 in PASCAL

Before we dive into a larger program, I thought I would point something out about Pascal that beginners must know. Statements in Pascal cannot see things written after the current line. On the flip side of the coin, statements can only see things in the code that occur on lines previous to the current one. In other words, when the compiler compiles your code into machine language, it starts at the top and remembers each line as it comes across it. This is while variables are declared BEFORE they are used. In the future you'll see that procedures and functions must be made before they are used. As an example:

```
var
     x : integer;
begin
     x := 3;
     y := 27;
...
```

The x:=3 works because x has been declared as an integer before it was used. The y:=27 causes a compile error because y has not been declared yet. It may be declared some time later in the program but that would be too late.

Why I'm bringing this up now is it should help a little bit when we take a look at procedures and functions. For now, just declare variables in the var section before you use them in code.

THE CALCULATOR - ATTEMPT #1

Let's take a little break from learning the language to try to apply what we've learned up until now. The best way to do that is by writing an example program. From past experience, the quickest way to learn a language (or a new language) is by writing example programs. You must type these programs in and test them for yourselves for maximum learning benefit.

In this section we'll try to build a simple calculator. It won't do a heck of a lot but I'm sure it will make understanding of the language easier. OK, so what's this calculator supposed to do? The first thing is it should ask the user for a number, an operation to perform, and then another number. Once these have been entered, the program will spit out the answer. The operations it can perform are addition (+), subtraction (-), multiplication (*), and division (/). One catch to watch out for is division. You don't want the user dividing by zero as this is an error and will cause your program to crash. You therefore need to print an error message yourself if they try to divide by zero.

Overwhelmed? Don't worry. There are ways to look at problems and break them down into smaller problems that are more manageable. Let me show you the program and I'll explain what it does.

```
program Calculator_Number_1;
uses WINCRT;
var
    numstr1    : string;
    numstr2    : string;
    operation : string;
    num1        : integer;
    num2        : integer;
    errcod     : integer;
begin
    writeln('Calculator Example #1');
    writeln;
    write('Please type a number and press <Enter> : ');
    readln(numstr1);
    write('Please enter the operation to perform (+, -, *, /) : ');
    readln(operation);
    write('Please type a number and press <Enter> : ');
    readln(numstr2);
    val(numstr1, num1, errcod);
    val(numstr2, num2, errcod);
    if (operation = '+') then
        writeln('Answer: ', num1+num2)
    else
        if (operation = '-') then
            writeln('Answer: ', num1-num2)
        else
            if (operation = '*') then
                writeln('Answer: ', num1*num2)
            else
                if (operation = '/') and (num2 <> 0) then
                    writeln('Answer: ', num1 div num2)
                    else
                         writeln('Error in input.   Program ending.');
end.
```

You should be able to read through this program if you take it slowly line by line. The first thing we do is declare all necessary variables. Then we ask the user for each number and an operation. The user must press <Enter> after each input. Next, we convert the number strings to integers. Notice we don't check for an error in converting the number strings, numstr1 and numstr2. We don't care what they enter. If they enter something invalid the integer we get from the conversion is just zero. It's left as an exercise for the reader to put in checks to test for invalid entries, report the problem and exit the program. TIP: You might want to look up the HALT procedure if you want to try this exercise.

OK, after the numbers are converted we perform the series of IF statement tests to perform the appropriate operation. Check out the last test for division. The condition statement in that if has two conditions: (operation = '/') and (num2 <> 0). If we were to read the last IF statement, it would be described something like: 'if the operation is division and the num2 isn't 0 then write the answer, else write an error message'. Words like 'and' which separate conditions will be discussed in more detail in upcoming months columns.

The programs are starting to look a little messy and I'm not using any comments (a bad habit but it won't help you to learn if I always tell you what I'm doing). We need to tidy things up a bit.

PROCEDURES

A typical Windows program can be several thousand lines long. If we have one big mainline, things will get beyond messy. Can you imagine thousands of statements between one begin and end? I can't. Programmers get around this problem by breaking up the code into chunks and giving a name to each piece. These 'chunks' are called procedures and functions.

Suppose the mainline of the calculator program above was written as:

```
begin
     Display_Title;
     Get_Inputs;
     Convert_Strings_to_Numbers;
     Display_Calculations;
end.
```

Isn't this a little more intuitive? All we need to do is tell the program what each of these procedures does. This is simple in this case. We simply take the groups of code that goes with each procedure and presto.

Here's the next calculator. Notice it's a little bigger but it's also broken up into nice manageable sections. Also notice, the procedures are written after the variable declarations but before the mainline. This way, the procedures can 'see' the variables because they're declare before it, and the mainline can 'see' the procedures because they're declared before it.

```
program Calculator_Number_1;
```

```
uses WINCRT;
var
    numstr1    : string;
    numstr2    : string;
    operation : string;
    num1        : integer;
    num2        : integer;
    errcod     : integer;

    procedure Display_Title;
    begin
        writeln('Calculator Example #1');
        writeln;
    end;

    procedure Get_Inputs;
    begin
        write('Please type a number and press <Enter> : ');
        readln(numstr1);
        write('Please enter the operation to perform (+, -, *, /) : ');
        readln(operation);
        write('Please type a number and press <Enter> : ');
        readln(numstr2);
    end;

    procedure Convert_Strings_to_Numbers;
    begin
        val(numstr1, num1, errcod);
        val(numstr2, num2, errcod);
    end;

    procedure Display_Calculations;
    begin
        if (operation = '+') then
            writeln('Answer: ', num1+num2)
        else
            if (operation = '-') then
                writeln('Answer: ', num1-num2)
            else
                if (operation = '*') then
                    writeln('Answer: ', num1*num2)
                else
                    if (operation = '/') and (num2 <> 0) then
                        writeln('Answer: ', num1 div num2)
                    else
                        writeln('Error in input.   Program ending.');
    end;

begin
    Display_Title;
    Get_Inputs;
    Convert_Strings_to_Numbers;
    Display_Calculations;
end.
```

Voila! Procedures are just like a little program within a program but they don't get executed until you call them as we do in the calculator mainline. The mainline is always the first thing that gets called in a program. It gets called automatically when the program is started. All other procedures and functions get called from the mainline. When looking at a program for the first time, skip down to the mainline and see what procedures are called. Some may be predefined in the language such as writeln and others may be user defined as in Display_Title.

Usually each procedure will have a lot of comments at the beginning of it. Even though they are often small, it's still best to make the reader aware of what each procedure does. It's also good practice to put a ton of comments at the beginning of the program to describe what it does (a sort of program overview).

It's interesting to note that a procedure can call another procedure. Such is the case in Display_Title for instance. Display_Title calls the procedure writeln. We can write our own procedures and have another one call it as long as it's declared BEFORE the one calling it. For instance, we could have called Display_Title from Get_Inputs but it didn't seem logical to me to do that. We could have though. This chaining of procedures is a very powerful feature indeed.

FUNCTIONS

A function is almost exactly the same as a procedure. The only difference is a function 'returns' a result whereas a procedure simply gets called and carries out it's instructions.

What do I mean, 'returns a result'? Well, let's look at math functions such as square root of a number, the sine of an angle, etc.. Then there's string functions such as get the string length, concatenate two strings together, etc.. There are conversion functions such as convert a string to a number, convert a number to a string, etc.. Functions do something and then return some result.

Function results have to be of a data type such as integer, string, char, or any other type. One interesting characteristic of functions is they can be used in expressions. Expressions are the right side of assignments (:=) or parameters to functions or procedures such as writeln. Let's give a couple examples using the SQRT (square root) function.

```
x := SQRT(9);
    {calculate the square root of 9 and assign that value to x}

x := SQRT(y);
    {calculate the square root of the contents of y and assign to x}

writeln(SQRT(y));
    {write the square root of y to the screen}
```

Some functions can be very simple (such as add the number three to a variable and return the result), or may be more complicated (such as a function used to calculate mortgages). Let's show how to write one of the simple ones this month - a function that simply returns the math number PI divided by 2. I think you should be getting the jist of how to write a program show I'll only show a function and how to call it.

```
function PI_DIV_2 : integer;
```

```
begin
    PI_DIV_2 := PI div 2;
end;

...

begin
    ...
    writeln(PI_DIV_2);
    ...
    x := PI_DIV_2 + 4;
    ...
end.
```

Instead of procedure, we start a function with the word function. Also, we must tell what type of value will be returned. That's the ': integer' part of the header. To tell the function to return the desired value, we appear to assign a variable with the same name as the function with the return result (in this case, the return result is 'PI div 2'). Once the function is defined, you can use it any way you would a predefined function.

NEXT MONTH

To wrap up, functions and procedures are very powerful additions to the Pascal language. We've touched on them here a little but you will see them over and over in the future and I'm sure you'll get very used to using them.

Next month I'll show how to pass a parameter to a function you define, just like SQRT takes one parameter and returns a result. I'll also show more data types, arrays, constants, introduce you to controlling the compiler in your code, and perhaps even how to write a simple game. See you then!

If you wish to contact me, please do so at the following e-mail addresses:

From	Send e-mail to
Compuserve:	>INTERNET:bill.lenson%canrem@uunet.ca
Internet:	bill.lenson%canrem@uunet.ca