March 1997

MICROSOFT SYSTEMS JOURNAL

# More Fun With MFC: DIBs, Palettes, Subclassing and a Gamut of Goodies, Part II

**Wouldn't it be nice if, instead of copying all that palette code from app to app, there was some way to encapsulate that palette code in a class you could instantiate in your app to just do palettes? Every application is slightly different, but there is a basic pattern to palettes.**

Paul DiLascia

**This article assumes you're familiar with C++, MFC, Win 32.**

Code for this article: MFCPart2.exe (69KB)

*Paul DiLascia is a freelance software consultant specializing in training and software development in C++ and Windows. He is the author of* Windows++: Writing Reusable Code in C++ *(Addison-Wesley, 1992).*

**After January's article**, I had to leave in a hurry. I raced over to Acme World Headquarters where I was scheduled to present my DIBVIEW program to Acme's Maximum Leader and beg for a deadline extension for writing Acme's XYZ file viewer. (He granted me two more days.) In case you've forgotten, I wrote an app called DIBVIEW that lets you look at DIBs (device-independent bitmaps). I implemented a class, CDib, that loads and draws bitmaps using some new Win32® functions and the DrawDib API from Video for Windows®. Then I added a whole bunch of code to handle palette messages—WM_QUERYNEWPALETTE and WM_ONPALETTECHANGED.

If your memory is a little fuzzy, now is a good time to go back and skim Part I ("More Fun with MFC: Dibs, Palettes, Subclassing, and a Gamut of Reusable Goodies," *MSJ* January 1996), since that's all the refresher you'll get. I've got a lot to cover, so I want to jump right in where I left off. This is where things start to get really interesting.

## Probing the Pattern in Palettes

All that palette stuff is pretty boring when you get right down to it. You read the manuals and the articles on MSDN, maybe copy some code from a sample program. Programmers have been doing palettes ever since Windows 3.0; just imagine how many programs are out there with palette-handling code similar to what's in DIBVIEW. With all that code, you'd think by now someone would've written a function to Just Do It. No such luck.

That's because most programmers follow the copy-and-paste school of software reusability: the way you reuse the code is to copy from one source file and paste into another. This is indeed a form of reusability, though it's more the sort of thing cavemen would think of than sophisticated programmers. Which is to say, primitive. Nevertheless, the copy-and-paste style of programming is so ingrained that most programmers don't think twice about it. Copy-and-paste even lives on in code generators, programs like AppWizard that are really no more than sophisticated copy-and-paste machines. If the behavior is so generic that you can write a program to write a program to implement it, why not just put it in a class or subroutine any app can call?

Wouldn't it be nice if, instead of copying all that palette code from app to app, there was some way to genuinely reuse it? What you need is some way to encapsulate that palette code in a class you could instantiate in your app to just do palettes. Every app is slightly different of course, but there is a basic pattern to palettes. A view realizes its palette in the foreground when it gets focus or its main frame gets WM_QUERYNEWPALETTE, and in the background when its main frame gets WM_PALETTECHANGED. The only thing that differs from app to app is the palette itself. There should be some way to capture the "palette pattern" in a reusable class.

In C++/MFC (or any other framework), the obvious thing to do is derive a new class, CViewThatHandlesPalettes, that apps can derive from. This class would have OnQueryNewPalette, OnPaletteChanged, OnSetFocus, and OnInitialUpdate handlers like the ones for CDIBView. But that won't work because you'd need a CScrollViewThatHandlesPalettes and a CFrameWndThatHandlesPalettes, one for CMDIFrameWnd and CWnd, and practically every window class there is in MFC. Even if you were willing to write all that code (perhaps using templates), it still wouldn't be enough.

Suppose some programmer is using a third-party library?

```
CWnd           // MFC
 CView         // MFC
  CBetterView  // Zippy Tools
CMyView        // app
```

Where are you, as a tool developer, going to insert CViewThatHandlesPalettes? Nowhere—there's no place to put it. The hierarchy is already cast in concrete and the code already written. This problem is endemic to MFC and any system with single-inheritance class hierarchies. You want the view to be a CBetterView and a CViewThatHandlesPalettes. The only solution is multiple inheritance.

```
class CMyView : public CBetterView,
   public CViewThatHandlesPalettes {
•
•
•
};
```

Unfortunately, MFC doesn't support multiple inheritance in this way. I don't know any framework that does. Most Windows-based C++ application frameworks are alike in this respect. They all use a single-inheritance model that makes it difficult to write reusable library extensions. So if Zippy Tools makes a CBetterView and Dippy Tools makes a CViewThatHandlesPalettes, you can't use both—you

have to choose one or the other. It's not just an issue for tool vendors, because good application programmers write tools too, like I did for DIBVIEW.

This problem doesn't exist in C. In C, you could do something like the following: write a special window proc, PaletteMsgProc, that handles WM_QUERYNEWPALETTE, WM_PALETTECHANGED, and WM_SETFOCUS as described previously. When it comes time to actually realize the palette, PaletteMsgProc sends a new message, called PALWM_REALIZE, with the HDC in WPARAM and the foreground/background flag in LPARAM. Any app that wants to use palettes would just have to install this window proc ahead of its own—in other words, subclass its windows—and realize its palette when it gets PALWM_REALIZE. That would work perfectly.

As a tool developer, you could even supply the convenient functions InstallPaletteHandler(HWND) and RemovePaletteHandler(HWND) to do the subclassing and unsubclassing, so the application programmer doesn't even know about PaletteMsgProc. Then all the application programmer has to do is call the install function and handle PALWM_REALIZE. In fact, even that is unnecessary if the install function accepts an HPALETTE. Just install the palette handler and forget about it. If Zippy Tools and Dippy Tools each produce something like this, you can use them both because, with Windows and C, you can subclass a window any number of times in any order. The class hierarchy (in the Windows sense) needn't be specified in advance. Each library just inserts its window proc in front of whatever is there before it. (I'm assuming that the Zippy and Dippy window procs provide unrelated functionality—one handles palette messages while the other does, say, 3D controls.)

So the real problem is that MFC doesn't let you subclass a window in the Windows sense. In MFC, every window is subclassed with the generic AfxWndProc. MFC (and every other C++/Windows framework I know) maps the Windows notion of subclassing to the C++ concept of derived classes. The way you modify the behavior of an existing window in C++ is to derive a new class and override one or more message handlers. It's a great model in many respects, but it ties you to a rigid class hierarchy that's hardwired into the source code. There's no way in C++ to say, "derive my new extension class from the 'most derived' class," which is exactly what Windows subclassing does and what you need to build some sort of reusable palette message handler.

The long and short of it is this: to reuse my now-perfect palette code from Part I, adding a window class is out; what you need is a way to subclass MFC windows.

## CMsgHook: Windows Subclassing in MFC

Just because MFC doesn't let you subclass its windows doesn't mean you can't do it anyway. After all, C++ is C (and don't ever forget it). All MFC does is make life complicated and force you to be more sneaky. For STEP4 of DIBVIEW, I implemented a new general-purpose class, CMsgHook, that hooks a CWnd object by subclassing it in the Windows sense—that is, by inserting its own window proc ahead of whatever proc is there currently, usually AfxWndProc. I thought of calling this new goodie CSubclassWnd since that's what it really does—subclass a window—but "subclass" is too fraught with misconception. CMsgHook emphasizes the fact that

my new class really just hooks messages. Also, CMsgHook is not derived from CWnd, though it does resemble CWnd in many respects. For example, it has WindowProc and Default functions just like CWnd.

The best way to understand how CMsgHook works is to look at how you'd use it in a program. The first thing you do is install the hook by calling CMsgHook::HookWindow.

```
CWnd* pMyWnd;             // some window
CMsgHook hook             // message hook
hook.HookWindow(pMyWnd); // now it's "hooked"
```

CMsgHook::HookWindow subclasses the window in the normal Windows sense.

```
// (simplified)
BOOL CMsgHook::HookWindow(CWnd* pWnd)
{
  m_pWndHooked = pWnd;
  m_pOldWndProc = SetWindowLong(pWnd->m_hWnd,
                                GWL_WNDPROC, HookWndProc);
  return TRUE;
}
```

CMsgHook stores the pointer to the window in a data member, m_pWndHooked, then subclasses the window by installing its own HookWndProc, saving whatever was there before. Now all WM_ messages for pMyWnd go to HookWndProc instead of AfxWndProc.

```
// (simplified)
LRESULT CALLBACK
HookWndProc(HWND hwnd, UINT msg, WPARAM wp, LPARAM lp)
{
  CMsgHook* pHook = GetMsgHookForThisHWND(hwnd);
  return pHook->WindowProc(msg, wp, lp);
}
```

When HookWndProc gets a message, the first thing it does is find the CMsgHook object attached to the HWND. I'll describe how it does that in just a moment; for now take my word that it can. Once HookWndProc has the right CMsgHook object, it calls that hook's WindowProc function.

```
// it's virtual!
LRESULT
CMsgHook::WindowProc(UINT msg, WPARAM wp, LPARAM lp)
{
  return ::CallWindowProc(m_pOldWndProc,
                          m_pWndHooked->m_hWnd, msg, wp, lp);
}
```

As you can see, WindowProc doesn't do much. It passes the message back to the original window proc. The overall result is ... nothing. But that's OK. CMsgHook isn't supposed to do anything; its only job is to handle the mechanics of hooking (subclassing) a window. To actually do something, you have to derive a new class from CMsgHook and override CMsgHook::WindowProc.

```
LRESULT
CMyMsgHook::WindowProc(UINT msg, WPARAM wp, LPARAM lp)
{
  if (msg==WM_QUERYNEWPALETTE) {
    // handle it
    return 0;
  }
  return CMsgHook::WindowProc(msg, wp, lp);
}
```

If the specialized hook handles the message, it can return a value; otherwise it must pass the message to CMsgHook:: WindowProc, which sends it back to the

original window proc. In effect, all CMsgHook does is convert the C-style window proc into a C++-style virtual function, CMsgHook:: WindowProc. This is exactly the same thing CWnd does.

The only part I left out of my explanation is how HookWndProc gets the CMsgHook associated with a window handle (HWND). This is where I followed my KISS (keep it simple stupid, not "I wanna rock and roll all night") philosophy of software development, which dictates that I always start small, then enhance. In my first implementation, I stored the hook as a global variable, theHook, which I accessed from HookWndProc. This meant I could have only one hook for the entire app. Not very useful!

As soon as I got CMsgHook working with one hook in a test program, I went back and added a more sophisticated mechanism. For lack of inspiration I imitated what MFC does for CWnd, which is to store the HWND/CWnd association in a lookup table. So I derived another class, CMsgHookMap, from CMapPtrToPtr and wrote functions to Add, Remove, and Lookup the hooks (see MsgHook.cpp in **Figure 1**). Pretty straightforward. So far everything is just like CWnd—with one extremely significant improvement. MFC lets you associate or "Attach" one HWND with only one CWnd object, which is why you can't subclass in MFC. In CMsgHook, however, I added a data member, m_pNext, that can point to another message hook for the same window.

```
class CMsgHook : public CObject {
protected:
  CMsgHook*  m_pNext;  // next in chain this window
};
```

To make multiple hooks work, I modified CMsgHook:: HookWindow so the first time a particular CWnd is hooked, it adds the hook to the global map. After that it merely appends the hook to the list of hooks for that window. HookWindow thus creates the data structure illustrated in **Figure 2**. Of course, I also had to modify CMsgHook::WindowProc to call the next hook in the chain.

```
LRESULT
CMsgHook::WindowProc(UINT msg, WPARAM wp, LPARAM lp)
{
  return m_pNext ?
    m_pNext->WindowProc(msg, wp, lp) :
    ::CallWindowProc(m_pOldWndProc,
       m_pWndHooked->m_hWnd,msg,wp,lp);
}
```

Each CMsgHook::WindowProc calls the next hook's WindowProc until control flows to the last hook, which then calls the original window proc, m_pOldWndProc. This way, a single CWnd object can have any number of hooks attached to it. Pretty cool.

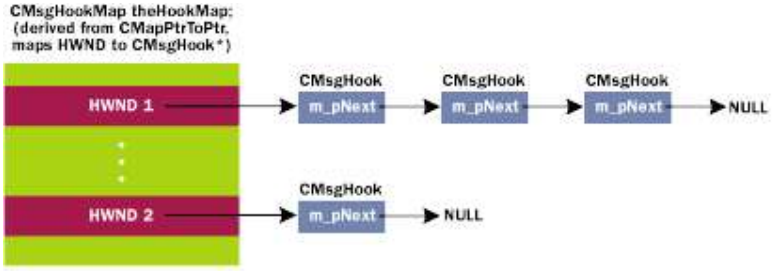


**Figure 2 CMsgHook Data Structures**

To test it all out, I wrote a program called HOOK that implements three different kind of hooks:

CMouseMsgHook, CKbdMsgHook, and CAllMsgHook (see **Figure 1**). These record all mouse, keyboard, or WM_ events in the TRACE output. Each hook is derived from CMsgHook, with a WindowProc function that uses DbgName(UINT), described in Part I, to TRACE the WM_ messages.

HOOK's main frame window contains an instance of each hook as a data member.

```
class CMainFrame : public CFrameWnd {
   CMouseMsgHook  m_mouseMsgHook;
   CKbdMsgHook    m_kbdMsgHook;
   CAllMsgHook    m_allMsgHook;
•
•
•
};
```

Initially, these hooks are all unhooked. When the user invokes a menu command to turn one of the hooks on, MFC calls my command handler in CMainFrame:

```
void CMainFrame::OnHookKbd()
{
  m_kbdMsgHook.HookWindow(
    m_kbdMsgHook.IsHooked() ? NULL : this);
}
```

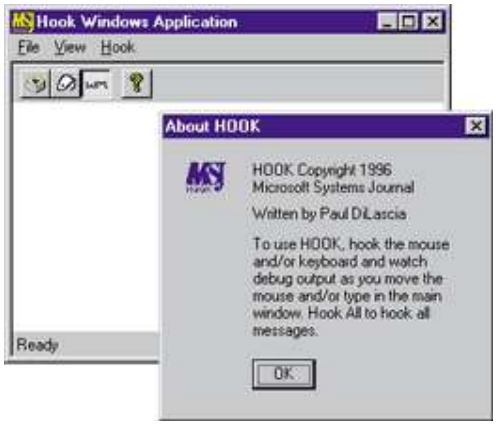


**Figure 3 The HOOK program**

OnHookKbd toggles the state of the keyboard hook; likewise for the other hooks. The user can turn each hook on or off independently of the others, proving that multiple hooks work. **Figure 3** shows HOOK running, and **Figure 4** shows the TRACE output generated with the All hook turned on. Indenting works courtesy of TRACEFN from Part I.


**Figure 4 TRACE output for All**

There are a few implementation details that I skipped in my explanation of CMsgHook. For one thing, HookWndProc automatically unhooks a hook when the window gets WM_NCDESTROY, in case you forgot. It also stores the current MSG information (HWND, WPARAM, and LPARAM) in AfxGetThreadState()->m_lastSentMsg, just like AfxWndProc does.

```
MSG& curMsg = AfxGetThreadState()
                ->m_lastSentMsg;
MSG  oldMsg = curMsg;   // save for nesting
curMsg.hwnd    = hwnd;
curMsg.message = msg;
curMsg.wParam  = wp;
curMsg.lParam  = lp;
•
•
•
curMsg = oldMsg;  // restore
```

MFC always saves the current message in the thread's global state, so I do the same thing in case any function that gets called while processing the message needs it. In particular, CMsgHook::Default uses AfxGetThread-State()->m_lastSentMsg to do the default thing—pass whatever the current message is to the original window proc. Just like CWnd::Default, CMsgHook:: Default is useful when you want to break your WindowProc override into separate handlers instead of a single giant switch statement, but you don't want to pass msg, wParam, and lParam everywhere. Note that it's crucial to save/restore the old MSG information in oldMsg, because control is likely to reenter HookWndProc any number of times down the call stack as one message begets another in typical Windows fashion.

Another absolutely crucial detail for DLLs is that you must initialize MFC's state when control enters your hook proc.

```
LRESULT CALLBACK
HookWndProc(HWND hwnd, UINT msg,
          WPARAM wp, LPARAM lp)
{
#ifdef _USRDLL
  // If this is a DLL, set up MFC state
AFX_MANAGE_STATE(AfxGetStaticModuleState());
#endif
     •
     •
     •
}
```

I discovered this one the hard way when I converted DIBVIEW to a Quick View file viewer (which is a DLL). You must use AFX_MANAGE_STATE at the top of any function—callback or C++ member—that can receive control from the outside. AFX_ MANAGE_STATE is what initializes the module or thread's global state information. I'll have a lot more to say about MFC states in a future article; for now, I'll just note this important detail and move on.

## CMsgHook versus CWnd

As I was writing CMsgHook, I had to chuckle to myself because I was essentially reinventing the wheel. CMsgHook does almost exactly the same thing as CWnd, with a couple of important twists. Number one, since CMsgHook is not derived from CWnd, CMsgHooks don't have message maps. This means you have to override WindowProc to handle messages; you don't get them served on a platter the way CWnd-derived objects are. Number two—and this is the whole point—CMsgHook lets you hook the same window any number of times with any number of different kinds of hooks.

The similarity to CWnd is so close I actually tried deriving CMsgHook from CWnd so it would really "be" a window. Then you could use message maps and CCmdTarget and everything else. But I couldn't make it fly. There are just too many places where MFC assumes —or ASSERTS—that there's only one CWnd object attached to an HWND. The AssertValid function for CWnd checks this, and there are countless places where MFC does ASSERT_VALID on a CWnd pointer.

I considered Detaching the original CWnd and Attaching each hook as it came into scope, but this seemed untenable since the context swap would have to occur whenever control flowed from one hook to another and to the CWnd. For example, if CMyMsgHook calls m_pWndHooked->AnyFunction, I'd have to Detach the hook and Attach the CWnd. This could be done by exposing m_pWndHooked through a Get function GetHookedWnd that did the context swap before returning m_pWndHooked. But then how do you re-Attach the hook whenever a CMsgHook or CMsgHook-derived member function gets control? You'd need some kind of HOOK_PROLOGUE macro to Detach the CWnd and re-Attach the hook. This would get pretty unwieldy, if it would even work at all—not to mention how it would affect performance. All the Attaching and Detaching would occur on every WM_*xxx* message. Forget it! (If you're totally lost, don't worry, just keep reading.)

It's a shame MFC doesn't support multiple subclassing. In principle, there's no reason MFC couldn't do subclassing using a daisy-chain approach like in **Figure 2**. Allowing true Windows-style subclassing would give programmers an elegant way to build some really neat MFC extensions, and would let you use multiple extensions in the same window object.

## Total Palette Bliss: CPalMsgHandler

Just in case you don't fully appreciate the significance of CMsgHook, it's time to return to palettes. I know, I know—you hate palettes by now. But CMsgHook ought to cure your palette ills forever. After this, you'll never again have to worry about palette messages, I promise!

After freezing DIBVIEW yet again as STEP3, to begin work on STEP4 (this is the last version, really), the first thing I did was derive a new message hook, CPalMsgHandler, to implement the palette pattern described earlier. CPalMsgHandler::WindowProc has a

switch statement that routes WM_QUERYNEWPALETTE, WM_PALETTECHANGED, and WM_SETFOCUS to specific message handler functions OnQueryNewPalette, OnPaletteChanged, and so on, which implement the pattern. The implementation is straightforward, mostly just a matter of copying the CView and CMainFrame code from STEP3 to CPalMsgHandler, changing all invocations through an implicit this pointer to invocations through m_pWndHooked.

Naturally, there were a few wrinkles. The biggest problem has to do with WM_INITIALUPDATE. If you remember, I had to call DoRealizePalette from CDIBView:: OnInitialUpdate because, for SDI apps, MFC doesn't load the document until after you get WM_QUERYNEWPALETTE, which is too late. In converting from CDIBView to CPalMsgHandler, this meant handling WM_INITIALUPDATE, which is a private MFC message defined in <afxpriv.h>. No big deal, just include the file. But when I ran the code, it didn't work; my palette hook never got WM_INITIALUPDATE. I set a break point and ran my code in the debugger to see if maybe MFC was calling OnInitialUpdate directly instead of sending it as a WM_ INITIALUPDATE message. What I discovered was even worse: MFC doesn't call OnInitialUpdate directly, it calls SendMessageToDescendants to broadcast the message from the frame to all its views.

SendMessageToDescendants contains the following lines:

```
if (bOnlyPerm) {
   AfxCallWndProc(pWnd, pWnd->m_hWnd, msg, wp, lp);
} else {
   ::SendMessage(hWndChild, msg, wp, lp);
}
```

I've omitted everything else to highlight the heinous crime: if bOnlyPerm is TRUE (send to permanent windows only—ones with CWnds attached), CWnd calls AfxCallWndProc.

```
LRESULT
AfxCallWndProc(CWnd* pWnd, HWND hWnd,
               UINT nMsg, WPARAM wp, LPARAM lp)
{
•
•
•
  lResult=pWnd->WindowProc(nMsg, wp, lp);
•
•
•
}
```

AfxCallWndProc calls CWnd::WindowProc directly! Can you believe it? Instead of going through Windows—and instead of calling SendMessage—MFC just short-circuits the whole show. Arrghhh! This sort of kludge makes me want to pull my hair out. Be warned: do not use SendMessageToDescendants with bOnlyPerm=TRUE if you install a new window proc, because the message is not really sent at all—MFC calls CWnd::WindowProc directly.

Unfortunately, while you are free to avoid SendMessageToDescendants, you can't control what MFC does, and CFrameWnd uses SendMessageToDescendants to broadcast WM_INITIALUPDATE. There are other places where this happens, too. For example, MFC uses SendMessageToDescendants to send WM_IDLEUPDATECMDUI. This means you can't write a CMsgHook that taps into idle processing, and that's a shame.

Now, in defense of the Friendly Redmondtonians, I

know what they're trying to do. They're trying to squeeze every last microdram of performance out of MFC, but the tiny bit of performance gained from a few extra cycles to go through SendMessage does not justify violating the integrity of the whole Windows system, which demands that every, and I mean every, message go through normal channels—that is, through whatever window proc is currently installed in the window. If MFC wants to be a little faster, it can use ::CallWindowProc(GetWindowLong(GWL_WNDPROC)).

What can I do to make CPalMsgHandler work? How can I hook WM_INITIALUPDATE when MFC sends it through secret channels? Frankly, there's nothing I can do. I looked around for some other message that gets sent around the same time, something to hook after a new doc is opened in an SDI app, but there isn't one. So to use CPalMsgHandler, you just have to remember to call DoRealizePalette(TRUE) from your OnInitialUpdate.

Aside from that, all you have to do is instantiate CPalMsgHandler objects in your frame and view classes, and then hook them up by calling CPalMsgHandler::Install from your OnCreate or OnInitialUpdate handler (see View.cpp and MainFrm.cpp in **Figure 5**).

The second problem I encountered implementing CPalMsgHandler is that it needs to handle messages a little differently depending on whether the window hooked is a frame window or a child view. If you remember, CMainFrame ::OnQueryNewPalette broadcasts the message to its children, whereas CDIBView::OnQueryNewPalette realizes its palette. Rather than implement two classes, CMainFramePalMsgHandler and CViewPalMsgHandler, I decided to test inside CPalMsgHandler whether the hooked window is a top-level frame or a child window. This approach is a little less object-oriented from a purist's perspective, but it makes CPalMsgHandler easier to use and more foolproof since there's only one class to deal with.

Finally, so that CPalMsgHandler works in apps that don't use doc/view, where the frame draws directly in its client area, I modified the logic slightly. CPalMsgHandler gives the frame window first crack at realizing the palette by calling DoRealizePalette first, in OnQueryNewPalette and OnPaletteChanged. In vanilla doc/view apps like DIBVIEW, you'll Install your main frame with a NULL palette, in which case DoRealizePalette does nothing and CPalMsgHandler passes the messages on to the frame's children. Main frames that draw can Install their CPalMsgHandlers with a non-NULL palette.

CPalMsgHandler works like a charm. More important, it's reusable. If I ever write another app that does palettes—maybe a video editor or paint program—I can just plop my palette handler in and give it a palette. That's a lot easier and more reliable than copying and pasting code from some other app and then changing the variable names (and probably making a mistake in the process). While the chances are pretty good that CPalMsgHandler will work straight out of the box in my new app, what if it doesn't? Maybe CPalMsgHandler isn't quite general enough. Maybe I'll have to modify it slightly, perhaps adding another virtual function. Or maybe I'll have to do something that makes it faster.

In fact, this actually happened in real life. When I converted DIBVIEW to a DLL running with apartment model threading, I had to add a line in OnSetFocus to call SetForegroundWindow to get the palette realization to work right. What's my point? That's how you improve

your code—by stressing it in different situations. But now when I do improve CPalMsgHandler, DIBVIEW and all the other apps that use it get the improvement free, just by recompiling with the new version. If I had copied the code from program to program and replicated source files like a human Xerox machine, it's unlikely I'd bother to back-port my improvements to all those apps. But when the code lives in only one place, it's easy.

Of course, it took me a little extra time and effort to encapsulate my palette-handling code in a reusable class, and time is precious when you have Acme's Maximum Leader breathing down your back. It also took some extra time to write CMsgHook. But look what I gained: CMsgHook is a totally general mechanism whose usefulness extends way beyond palettes. You can use it to encapsulate window behavior into little objects you plop in your window objects to do various things.

Windows is full of patterns like the palette. For example, when you want to draw your own title bar, there are several messages you have to handle: WM_NCPAINT, WM_SETTEXT, and WM_NCACTIVATE. The overall pattern of message handling is the same every time. The only thing that's different is what you do when it comes time to actually paint the caption. You could use CMsgHook to implement a CCaptionPainter class that handles all the mechanics of these messages and calls one virtual function, OnPaintCaption, to actually paint the caption. The possibilities are endless. You could use CMsgHook to implement a CDragMove class that handles WM_LBUTTONDOWN, WM_LBUTTONUP, and WM_MOUSEMOVE to support moving a window by dragging its client area. ISVs could use CMsgHook to develop extension classes that send private messages to themselves, without forcing application developers to derive from new classes. All the application programmer has to do is hook up the hooks and compile. As a tool supplier, you don't have to know which C++ class your hook will end up in (CView or CBetterView). As a tool consumer, you can use as many kinds of hooks as you need in your window object. And as I hope I've shown you, the supplier/consumer distinction is conceptual, not physical; any time you write an app, you can be both a supplier and a consumer of your own tools.

## Final Feature Fun

OK, now that you've learned how to load and draw DIBs, built a cool MFC subclassing extension, solved the palette problem, and saw what it means for software to be truly reusable, it's time to put the icing on the cake! I'm talking about features, of course. That's what makes software fun—doing neat things. In addition to displaying the image itself, the Acme spec also calls for displaying the information from the BITMAPINFOHEADER in the user's choice of font. And it's gotta do printing, too. In the hopes of impressing Maximum Leader and getting another deadline extension, I decided to add a zoom feature that magnifies or shrinks the image.

Printing is, for some reason, always the orphan child when it comes to application development, probably because the printer is black-and-white and down the hall, while the screen has lots of pretty colors and sits right in front of you. In any case, printing is one place MFC really shines. The framework has built-in command handlers and functions to run the proper dialogs. The default OnPrint function for CView calls your view's OnDraw function to draw on the printer instead of the screen. In other words, MFC makes printing work without you having to write even a single line of code!

Well, almost. **Figure 6** shows what happened (actual

**Figure 6 Too small**

size) the first time I tried printing one of my bitmaps. Great for printing postage stamps, but not for viewing hardcopy DIBs. The problem is that CDIBView uses MM_TEXT as the default drawing mode, which means all my dimensions (width and height) are measured in pixels, which have no intrinsic metric. If my screen has 75 dots per inch and my bitmap is 300 pixels wide, that's four inches. On the printer, where I have 300 dots per inch, that's only one inch—a bit hard to see without a magnifying glass. Alas, printing is never as simple as just drawing to the printer instead of the screen.

Fortunately, the pixel problem isn't hard to fix. Just override OnPrint and convert all units to something meaningful, like inches. To make printing WYSIWYG, I used the formula

```
xPrinter = xScreen * (PrinterPPI / ScreenPPI)
```

That is, to convert a dimension from screen to printer units, multiply by the ratio (PrinterPPI / ScreenPPI), where PrinterPPI is the number of pixels per inch for the printer and ScreenPPI is the number of pixels per inch for the screen. You can get the number of pixels per inch for any device by calling GetDeviceCaps(LOGPIXELSX). Since the aspect ratio is not necessarily one, there's also LOGPIXELSY for the y direction. I'll spare you the details; take a look at CDIBView::OnPrint in View.cpp in **Figure 5**.

Once I modified OnPrint, printing worked fine, but I should point out that there are other ways to handle metrics. If you use some other mapping mode like MM_LOMETRIC or MM_HIENGLISH, where units are in millimeters or inches, then the same OnDraw function will work for drawing on the screen or printer—but you have to convert your bitmap dimensions to millimeters or inches first, because the bitmap info is always in pixels. There's no way to avoid a conversion somewhere. For DIBs, the BITMAPINFOHEADER has two fields, biXPelsPerMeter and biYPelsPerMeter, that specify the number of pixels per meter in x and y for the device on which the image was originally captured. This information is there to tell you how big the image is in real life. Unfortunately, these fields are often zero because capture programs don't bother to fill them in or some editing tool throws them away. Nevertheless, good imaging tools do use them, so a commercial DIB program should look at biXPelsPerMeter and biYPelsPerMeter to calculate the display/print dimensions. I was lazy so I punted (I have to save something for Release 2).

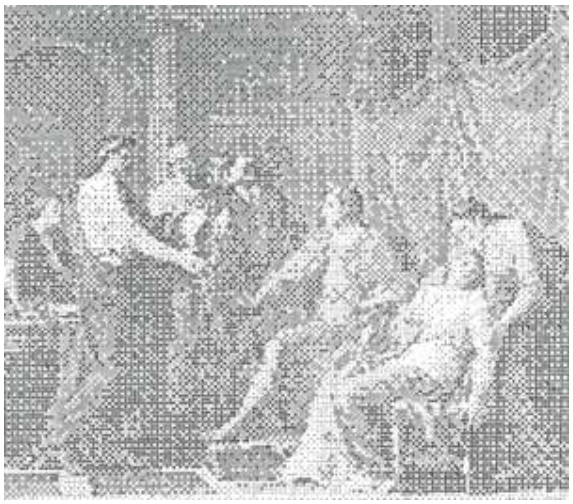

**Figure 7 Pre-DrawDib output**

If you remember from January, CDib uses DrawDib to do dithering. Printing is where DrawDib really shows its colors—even in black and white! **Figure 7** shows printing without dithering; **Figure 8** shows printing with it. Quite a difference!

**Figure 8 Printout with DrawDib**

Next on my feature list was displaying the information in the BITMAPINFOHEADER. This one is trivial. CDIBView:: DrawBITMAPINFOHEADER formats all the information into a string using printf—er, I mean CString::Format. It then calls DrawTextEx (which for some reason has no CDC wrapper) with DT_EXPANDTABS and DT_TABSTOP, to set a tab stop so the text lines up nicely whether the user chooses a monospaced or variable-width font.

## CFontUI: A Quick Font Goodie

Speaking of fonts, the spec calls for letting the user select the font: Arial, Copperplate, Dingbats, whatever. Also, since the UI guidelines for Quick View file viewers suggest using buttons to increment/decrement the font point size, and since my ultimate goal is to convert DIBVIEW into a file viewer, I decided now would be a good time to add the buttons. So I encapsulated all this font functionality into a single class, CFontUI, with one multipurpose function, OnChangeFont, that handles all three cases (see FontUI.h and FontUI.cpp in **Figure 5**).

```
CFont font;    // some font
CFontUI fui;   // Font UI
fui.OnChangeFont(font, opcode);
```

If opcode is negative, OnChangeFont shrinks the font; if opcode is positive, OnChangeFont makes the font bigger; if it's zero, OnChangeFont runs the common dialog CFontDialog. The details are just a lot of font mechanics: converting point sizes to pixel heights, calling CreateFontIndirect, running CFontDialog::DoModal, and so on. When CFontUI shrinks or grows the font, it uses an algorithm that adjusts the increment depending on how big the font currently is. For example, if the font is 10 points, CFontUI increments/decrements by one point; if the font is 32 points, CFontUI increments/decrements by four points, to 28 or 36 points. I got this idea from a sample text-file viewer on MSDN. Naturally, the function that implements the algorithm is virtual, so you can override it.

CFontUI also has Get/SetFontPointSize functions that do the standard points-to-logical-device-units conversion. As an added bonus, I threw in a couple of functions to load and save the font spec to the application profile (the .INI file or registry). If you have a font, all it takes to save it is a single line of code:

```
CFontUI().WriteProfileFont("Settings","Font",font);
```

My syntax may be unfamiliar to non-C++ gurus. I've used the ultra-terse trick of invoking the constructor to create a nameless CFontUI object on the stack, then call one of its member functions. The above line is equivalent to this:

```
CFontUI fui;
fui.WriteProfileFont("Settings",
                     "Font",font);
```

WriteProfileFont writes the information to the registry in the format shown in **Figure 9**; ReadProfileFont reads this format. I have to emphasize once more that by encapsulating all the font code in a little class, you don't just get a feature for DIBVIEW, you get a reusable software component. I'd long thought of modifying my TRACEWIN program to let users select different fonts, but never bothered because the feature would be useless without saving the font across sessions, and I was too lazy to do it. But once I had CFontUI, it took five minutes to update TRACEWIN with the new feature.

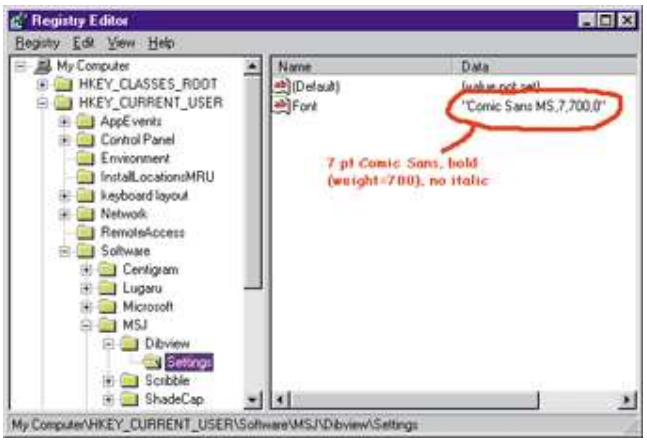


**Figure 9 WriteProfileFont registry entry**

One final trick before I leave fonts for good. When I first implemented the choice-of-fonts feature, I made the current font a data member of my view class.

```
class CDIBView : public CScrollView {
```

```
  CFont m_font;  // current font
•
•
•
}
```

Then I called all my nifty CFontUI functions with m_font as the CFont argument. This worked fine in the SDI version of DIBVIEW, but when I compiled it for MDI, I quickly realized I didn't want each view to have its own font; I wanted the whole app to have a single font. If the user changes the font, it should affect all views, not just the active one. At first I figured I'd have to move m_font to CApp or CMainFrame, but an even simpler solution came to me: make the font static—that is, a class global for CDIBView (I changed the name to g_font to remind you).

This worked fine, except the new problem was how to get all the views to recompute their scroll sizes and redisplay themselves when the user changes the font. UpdateAllViews is no good because it only updates the views for one document. I wanted to update all the views for all the documents.

The simplest solution proved to be broadcasting WM_INITIALUPDATE from the top-level frame to all children.

```
void CDIBView::OnFontChange(UINT nID)
{
  CFontUI fui;
  if (fui.OnChangeFont(...)) {
    GetTopLevelFrame()->
      SendMessageToDescendants(WM_INITIALUPDATE);
  }
}
```

(Notice I didn't call with bOnlyPerm=TRUE!) Now all the views get reinitialized. Since OnInitialUpdate is where I compute scroll sizes, this is just what I needed.

You needn't worry that WM_ INITIALUPDATE is undocumented; it isn't going anywhere. Just remember to #include <afxpriv.h>. One thing you do have to worry about is whether your OnInitialUpdate handler expects to get called again when it's not really first-time initialization, but ongoing reinitialization. For example, as a bonus feature to impress you-know-who, I modified OnInitialUpdate to call CScrollView:: SizeToFit so the window (MDI child or SDI frame) starts out the perfect size to accommodate its DIB with no scroll bars. Well, this is good the first time you run DIBVIEW, but I don't want the window changing its size every time the user changes the font! So I added a flag to confine this part of initialization to first-time-init only. Just something to be aware of.

My final feature is the magnify/zoom feature, which is really easy. I added a View Zoom command with popups for 1/4X, 1/2X, 1X, 2X and 4X magnification, and I also added mouse double-click handlers, so DIBVIEW doubles the zoom factor if the user double-clicks the mouse on the bitmap image. If the user double-clicks with the right button, DIBVIEW halves the zoom factor. (If the user double-clicks on the text, DIBVIEW runs the font dialog.) To implement zooming, all I had to do was add a scale factor, m_iZoom (this time I want each view to have its own), that ranges from -2 to +2, and a rectangle, m_rcDIB, that represents the client-area destination rectangle for drawing the DIB. I compute this rectangle in OnInitialUpdate by left or right-shifting the true DIB rectangle by my zoom factor.

**Figure 10** shows the final SDI version of DIBVIEW, and **Figure 11** shows the MDI version in all its glory, with font buttons, bitmap info displayed, generic palette handler class (you can't see that), and several images of

different sizes and color resolutions open at once. All the images show their true colors because I took the screen capture while running in 32-bit color mode.

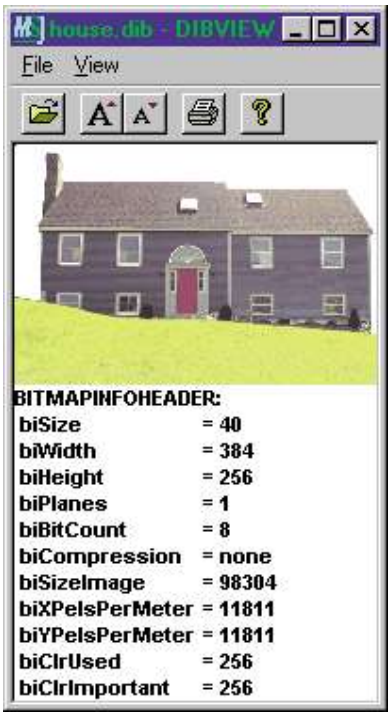


**Figure 10 SDI DIBVIEW**

**Figure 11 MDI DIBVIEW**

## To Be Continued...

I've shown you how to build several reusable classes that extend MFC and make writing apps easier. **Figure 12** lists them. I used these goodies to grow DIBVIEW from a baby bitmap viewer that didn't even get palettes right into a full-fledged app with printing, the user's choice of font, BITMAPINFOHEADER, and SDI and MDI versions. Each time I added some new feature, I took a little extra time to encapsulate it in a general-purpose, reusable class. If you take my lead and do things this way, you win on two counts. First, you get reusable objects that make the next app even easier to write. Second, even if you write only one app, its internal structure will be more coherent and hence bug-free when independent behavior is isolated in independent components. The alternative to this is called spaghetti code.

Despite Hurculean efforts to beat the clock, by the time I finished DIBVIEW I'd busted my schedule by at least seven hours. Needless to say, Mr. Maximum Leader was not thrilled. Fortunately, after I gave him a demo of DIBVIEW using some images I scanned off recent covers of Cosmopolitan magazine, his mood softened.

"OK, I'll give ya another two days."

Two days?! To convert DIBVIEW to a Quick View file

viewer? Yikes! Stay-tuned to see if I make it. (see Part III)

*From the March 1997 issue of Microsoft Systems Journal. Get it at your local newsstand, or better yet, subscribe.*