March 1996

MICROSOFT SYSTEMS JOURNAL

Extend the Windows 95 Shell with Application Desktop Toolbars

Jeffrey Richter

Jeffrey Richter wrote Advanced Windows *(Microsoft Press, 1995) and* Windows 95: A Developer's Guide *(M&T Books, 1995). Jeff is a consultant and teaches Win32-based programming seminars. He can be reached at v-jeffrr@microsoft.com.*

Click to open or copy the TOOLBARS project files.

The Windows® 95 shell is more than just a pretty face. It's easy to customize, unlike the old Program Manager. The Programmer's Guide to Microsoft Windows 95 (Microsoft Press, 1995) will give you an idea of the many ways in which the shell can be extended, but I've found the documentation to be scanty and sometimes incorrect. Here, I'm going to throw some light on the subject by examining one kind of shell extension, application desktop toolbars.

But first, let's go over exactly what I'm referring to with the "shell." After the system boots, the system creates the desktop window, the parent of all top-level windows that you create. The window procedure for the desktop window is inside USER.EXE and is responsible for displaying the user's selected wallpaper.

One of the last things that Windows 95 does when it boots is look in the SYSTEM.INI file. This file has an entry that looks like this:

```
[Windows]
Shell=Explorer.exe
```

This tells Windows 95 to execute the Explorer.exe application (you can change this entry if you prefer to have a different application run as your shell). Explorer.exe is the Windows 95 shell. When Explorer.exe is invoked, it creates a TaskBar window which, by default, is positioned at the bottom of your screen. Explorer.exe also creates a child window that fills the entire workarea but still displays the Desktop window's wallpaper (the term workarea will be covered later in this article). This child window is actually one of the new ListView common controls that always shows its contents using the Large Icon style. This ListView control is what we usually think of as the New Windows 95 Shell. Because this control covers the entire workarea, double-clicking the mouse over the ListView does NOT bring up the Task Manager because the mouse messages are not being received by the desktop window anymore.

Application Desktop Toolbars

An application desktop toolbar (or an appbar for short) is a window that attaches itself to an edge of your screen. An application can have more than one appbar-the program decides how many. The shell's taskbar window is a well-known example of an appbar. It is nothing more than a window that contains several child windows: the Start button, a notification window (which contains a clock), and a SysTabControl32 common control. What makes an appbar different from other windows is that it is always accessible to the user. This means the user always has the ability to start applications, find files, request help, shut down the system, and so on.

Currently, very few products offer appbars. In fact, the only software I'm aware of that ships with its own appbar is Microsoft® Office for Windows 95. Included in this suite is an app called MSOFFICE.EXE that creates an appbar window. **Figure 1** shows the shell's taskbar and the Microsoft Office shortcut bar.

Figure 1 Taskbar and shortcut bar

I expect many applications to offer application desktop toolbars eventually. Imagine a stock ticker application bar that constantly displays the current trading activity. The user could easily dock the appbar on an edge of the screen, and the information would always be visible.

Here are some other possibilities for application desktop toolbars:

- To constantly display system statistics such as CPU usage, available memory, available resources, paging file size, and so on.
- To report scores of sporting events (connecting to an online service provider).
- For quickly launching applications like the Microsoft Office shortcut bar.
- For real-time net chatting. The appbar window could have messages that come to you in a read-only edit control; messages that you send could be entered in a read/write edit control.
- A calculator.
- To display the time and location of your next meeting.
- As a phone dialer

Properties of Appbars

Though appbars are usually docked on an edge of the user's screen, they also can float. To make room for an appbar that's docked, the Windows 95 shell shrinks the size of the screen's workarea. The workarea is a new concept in Windows; it's the portion of the screen that is not occupied by any appbars. You query the system for the rectangle that bounds the current workarea by calling the SystemParametersInfo function:

```
  RECT rc;
SystemParametersInfo(SPI_GETWORKAREA, 0, &rc, 0);
// rc now contains the workarea in screen coordinates
```

Whenever the user maximizes a window, the system ensures that the window fills the workarea, not the full screen. If an appbar is docked on the edge of the screen, moved from one edge of the screen to another, or removed from an edge of the screen, the system automatically resizes and repositions any maximized windows so that they always fill the workarea completely. In addition, when the upper-left corner of the workarea changes, the system shifts all the application windows on the screen so that they always maintain their relative distance from that corner.

Although the system ensures that maximized windows are restricted to the workarea, the system does allow nonmaximized windows to overlap appbar windows. When overlapped, the windows follow the normal z-order rules. But both the shell's taskbar and the Office for Windows 95 shortcut bar offer an "Always on top" feature, and so can your appbar.

Because screen real estate is always at a premium, appbars also offer a feature called "autohide." When this is enabled, the appbar is almost completely off the screen when not in use. The appbar positions itself so that the bulk of its window is clipped off the screen: only a border a few pixels wide is visible. To use the appbar, the user positions the mouse over the border. When the appbar detects the mouse move message, it slides itself out onto the screen. When the mouse is moved away, the appbar slides itself back off the edge of the screen.

To freeze an appbar into place so it won't slither away, you click on it, which activates it and gives it the focus. To unfreeze the appbar, you merely activate another window.

Personally, I love the autohide feature of appbars and always set this feature on for the shell's taskbar. There is one important thing to note: the shell allows only one autohide appbar per edge of the screen (see **Figure 2**). The shell enforces this rule because two autohide appbars on the same edge would overlap each other completely. This would prevent the user from activating one of the appbars. You can perform the following experiment to see the shell enforce this rule: first dock the taskbar at the bottom of your screen, then dock the Office for Windows 95 shortcut bar

on the top of your screen; finally, turn the autohide feature on for both of these appbars. Now, drag the Office 95 shortcut bar to the bottom edge of your screen where the taskbar is located. The message box shown in **Figure 2** will appear and the shortcut bar will automatically turn off the autohide feature for itself.

Figure 2

An appbar can be moved by grabbing the appbar's client area and dragging. To determine which edge the appbar should be docked to, imagine that an X is drawn in the workareaasshowninFigure3.Theappbarisdockeddependingonwherethemouseisreleasedinrelationto the X.

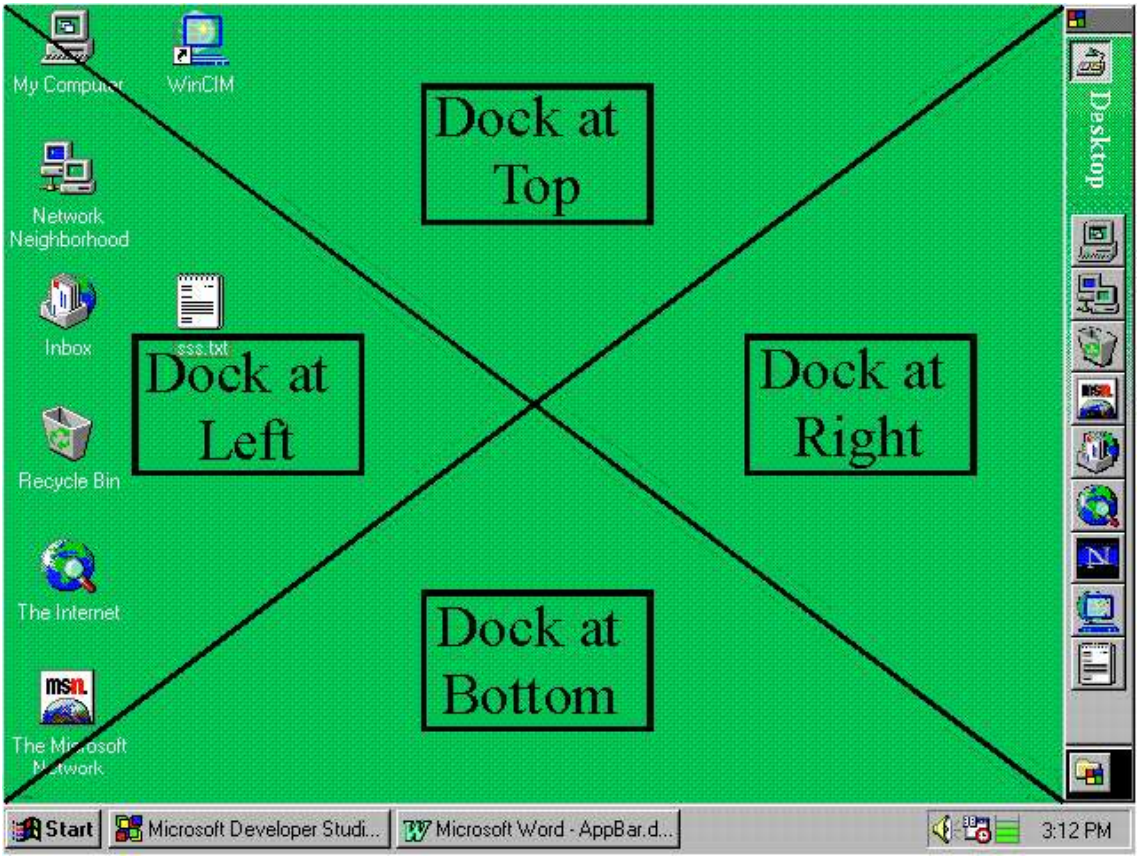

Figure 3 Docking appbars

Appbars can be resized by the user if the programmer has allowed for it. For example, the shell's taskbar allows itself to be resized. But when the taskbar is docked on the top or bottom of the screen, only its height can be adjusted; when it is docked on the left or right of the screen, only its width can be adjusted. The Office for Windows 95 shortcut bar allows itself to be resized only when it is floating.

Implementing an Appbar

Idecidedto create an MFC class, which I call CAppBar, that would package all of the features of application toolbars together so that it would be easy to create any type of appbar. My CAppBar class is derived from CDialog so that you can easily construct an appbar from a dialog box template.

Since the documentation on appbars is poor (actually, describing it as horrendous is paying it too much of a compliment), I will now properly explain the entire application desktop toolbar API. I'll show how to use my CAppBar class for your own appbars later in this article.

Working with appbars requires just one function:

```
UINTSHAppBarMessage(DWORDdwMessage,
                    PAPPBARDATApabd);
```

dwMessage tells the shell why you are calling it. You can pass any of the 10 ABM_xxx identifiers (discussed shortly) for this parameter. The pabd parameter is a pointer to an APPBARDATA structure that contains additional information:

```
typedef struct _AppBarData {
  DWORD  cbSize;
  HWND   hWnd;
  UINT   uCallbackMessage;
  UINT   uEdge;
  RECT   rc;
  LPARAM lParam;
} APPBARDATA, *PAPPBARDATA;
```

When initializing this structure, you must set the cbSize member to the size of APPBARDATA. The remaining members may or may not need to be initialized depending on the value of dwMessage.

The shell maintains a list of data structures that it uses to manage appbars. (Microsoft expects that appbars will be enhanced in future versions of Windows; therefore some of this information will change.)

```
struct INTERNALAPPBARDATA {
  HWND hWnd;
  UINT uCallbackMessage;
  UINT uEdge;
  RECT rc;
};
```

There is one structure in the list for each appbar window. If you want a window to be an appbar, you must register it with the shell by initializing an APPBARDATA structure and calling SHAppBarMessage as follows:

```
 APPBARDATA abd;
abd.cbSize = sizeof(abd);
abd.hWnd = hwndOfWindowToRegister;
abd.uCallbackMessage = WM_USER + 100;
// Ignored members: uEdge, rc, lParam
SHAppBarMessage(ABM_NEW, &abd);
```

The ABM_NEW message tells the shell to add a data structure to its internal list. The internal data structure's hWnd and uCallbackMessage members are initialized with the values passed in the APPBARDATA structure but no position information is retained at this point. If the window handle you pass is already registered with the shell, SHAppBarMessage does nothing and simply returns FALSE.

When your window is being destroyed, unregister it with the shell like so:

```
 APPBARDATA abd;
abd.cbSize = sizeof(abd);
abd.hWnd = hwndOfWindowToUnregister;
// Ignored members: uCallbackMessage, uEdge, rc, lParam
SHAppBarMessage(ABM_REMOVE, &abd);
```

The ABM_REMOVE message tells the shell to remove a data structure from its internal list. The APPBARDATA's hWnd member indicates which window to remove. Just before SHAppBarMessage returns, it sends out an ABN_POSCHANGED notification to all the remaining appbars so that they can reposition themselves. For example, an appbar might be able to move closer to the edge of

the screen if an appbar that was on the edge of the screen is unregistered. Appbar notifications are discussed later in this article.

Now, how do you tell the shell where you want the appbar to be positioned onscreen? It's not as simple and straightforward as you might like, because you want to ensure that your appbar does not overlap any others. First call SHAppBarMessage requesting a screen edge and a rectangle (in screen coordinates) for the window:

```
 APPBARDATA abd;
abd.cbSize = sizeof(abd);
abd.hWnd = hwndOfWindowToPosition;
abd.uEdge = uEdgeToDockOn;
SetRect(&abd.rc, 0, 0, GetSystemMetrics(SM_CXSCREEN),
        GetSystemMetrics(SM_CYSCREEN));
// Ignored members: uCallbackMessage, lParam
SHAppBarMessage(ABM_QUERYPOS, &abd);
```

Sending an ABM_QUERYPOS message causes SHAppBarMessage to compare your requested rectangle with the list of registered appbars. If your request interferes with any existing appbars, SHAppBarMessage alters the coordinates of the rectangle. Notice I always pass a rectangle that represents the full screen-this makes things easier. As SHAppBarMessage cycles through the registered appbars, it reduces the size of the rectangle. When SHAppBarMessage returns, abd.rc.left has the leftmost pixel at which my window can be, assuming that I want to dock on the left,abd.rc.top has the topmost pixel at which my window can be, assuming that I want to dock on the top, and so on.

Once you know where you can position the appbar, calculate its size.

```
 switch (abd.uEdge) {
   case ABE_LEFT:
     abd.rc.right = abd.rc.left + uWidthWhenDockedOnLeft;
     break;

    case ABE_TOP:
     abd.rc.bottom = abd.rc.top + uHeightWhenDockedOnTop;
     break;

   case ABE_RIGHT:
     abd.rc.left = abd.rc.right-uWidthWhenDockedOnRight;
     break;

   case ABE_BOTTOM:
     abd.rc.top=abd.rc.bottom-uWidthWhenDockedOnBottom;
     break;
}
```

Once you have the final rectangle coordinates for the appbar, call SHAppBarMessage again:

```
 // Leave all the members as they were when
// the ABM_QUERYPOS message was sent
// Ignored members: uCallbackMessage, lParam
SHAppBarMessage(ABM_SETPOS, &abd);
```

This time, the ABM_SETPOS message tells the shell where you really want to position your appbar. When you send an ABM_SETPOS message, SHAppBarMessage recursively calls itself (only once, mind you) with an ABM_QUERYPOS message. This is because Windows 95 is a preemptive multitasking system. While you were calculating the size of your rectangle, another thread could have been registering or unregistering another appbar window with the shell, possibly where you want yours.

After the shell has determined where you can go, it records your edge and rectangle in your appbar's internal data structure and then an ABN_POSCHANGED notification isbroadcasttoalltheotherregisteredappbar windows.

When SHAppBarMessage finishes processing the ABM_SETPOS message, you'll want to call SetWindowPos so that your appbar moves to the correct location:

```
SetWindowPos(abd.hWnd, NULL, abd.rc.left, abd.rc.top,
    abd.rc.right-abd.rc.left,abd.rc.bottom-abd.rc.top,
    SWP_NOZORDER | SWP_NOACTIVATE);
```

It's your responsibility to reposition your window; SHAppBarMessage never moves nor sizes any appbar windows.

Making It Autohide

The shell gives very little support for autohide appbars. For example, it would be very nice if the shell implemented the code to detect when an appbar window needs to slide on or off the screen, and it would also be nice if the shell did the work to slide the appbar window on and off the screen. But it doesn't, so you have to implement all this code yourself. All the shell does is keep an internal list of the autohide windows. It does this by maintaining a data structure that associates HWNDs with screen edges. Each HWND identifies the handle of an autohide window that is on an edge.

Here's how to set your appbar window as the autohide window for a particular edge:

```
 APPBARDATA abd;
abd.cbSize = sizeof(abd);
abd.hWnd = hwndOfWindowToMakeAutohide;
abd.uEdge = uEdgeToAutoHideOn;
abd.lParam = TRUE;      // Make us autohide
// Ignored members: uCallbackMessage, rc
SHAppBarMessage(ABM_SETAUTOHIDEBAR, &abd);
```

When you send the ABM_SETAUTOHIDEBAR message to SHAppBarMessage, the shell first checks to see if there is already a valid window handle set for the edge you specified in the APPBARDATA's uEdge member. If the edge doesn't have a valid window handle associated with it, the shell sets your appbar window as the autohide window and returns TRUE. If another window is set as the autohide window, SHAppBarMessage does nothing and returns FALSE. Your code should check this return value. If your appbar can't be set as autohide, you'll want to display a message box to the user like the one in **Figure 2**.

When you want to turn off autohide on your appbar window do the following:

```
 APPBARDATA abd;
abd.cbSize = sizeof(abd);
abd.hWnd = hwndOfWindowToMakeNotAutohide;
abd.uEdge = uEdgeToAutohideOn;
abd.lParam = FALSE;    // Make us NOT autohide
// Ignored members: uCallbackMessage, rc
SHAppBarMessage(ABM_SETAUTOHIDEBAR, &abd);
```

The only thing different in this call is that the APPBARDATA's lParam member is FALSE. This tells the shell to stop treating the specified appbar as an autohide appbar for that edge. When unregistering an autohide appbar window, SHAppBarMessage always returns TRUE.

Be aware that the shell does not use its two internal lists together, making more work for you. Let's say that you register an appbar window by sending an ABM_NEW message and register it as autohide by sending ABM_SETAUTOHIDEBAR. SHAppBarMessage should then resize the workarea portion of the screen-but this does not happen. When another appbar sends ABM_QUERYPOS or ABM_SETPOS messages, SHAppBarMessage should see that your window is autohide and should not consider your appbar's rectangle when determining the location of the appbar sending the message. Again, this does not happen.

So, you need to do additional work. To make your registered appbar autohide you have two choices.

First you can unregister it as a regular appbar by sending ABM_REMOVE. Since you will now be removed from the shell's internal list, your appbar window and its dimensions will no longer be considered when determining the workarea's rectangle and you will no longer have an effect on the position of other appbar windows.

Your other choice is to keep yourself registered but change your registered rectangle dimensions so that the shell thinks that your window is 0 pixels wide by 0 pixels high. This is what I did in my CAppBar class. In my SetState function, you'll see the following lines of code:

```
if (IsBarAutoHidden()) {
  SHAppBarMessage(ABM_SETPOS, ABE_LEFT,
                  FALSE, &CRect(0, 0, 0, 0));
                     .
                     .
                     .
```

At times, you may want to query the shell and ask it which window is autohide on a particular edge:

```
 APPBARDATA abd;
abd.cbSize = sizeof(abd);
abd.hWnd = hwndofOurAppBar;
abd.uEdge = uEdgeToCheck;
// Ignored members: uCallbackMessage, rc, lParam
HWND hwndAutohide = (HWND)
   SHAppBarMessage(ABM_GETAUTOHIDEBAR, &abd);
```

You have to cast SHAppBarMessage's return value to an HWND. The return value is NULL if no window is registered as an autohide window on the specified edge. (The Programmer's Guide to Microsoft Windows 95 states that you must initialize the hWnd member of the APPBARDATA structure before sending the ABM_GETAUTOHIDEBAR messages. Currently, the SHAppBarMessage function ignores the hWnd member when processing an ABM_GETAUTOHIDEBAR message. However, I was told that future versions of the shell will require this member. So heed the documentation and always initialize the hWnd member correctly when you send an ABM_GETAUTOHIDEBAR message.)

Now and then, you might want to query the shell about its taskbar. For example, you can find out if the taskbar is always-on-top or autohide by executing the following:

```
 APPBARDATA abd;
abd.cbSize = sizeof(abd);
// Ignored members: hwnd, uCallbackMessage, uEdge, rc,
// lParam
BOOL fTaskBarIsAlwaysOnTop =
   SHAppBarMessage(ABM_GETSTATE, &abd) & ABS_ALWAYSONTOP;
BOOL fTaskBarIsAutohide =
   SHAppBarMessage(ABM_GETSTATE, &abd) & ABS_AUTOHIDE;
```

You can also get the position of the shell's taskbar by doing this:

```
 APPBARDATA abd;
abd.cbSize = sizeof(abd);
// Ignored members: hwnd, uCallbackMessage, uEdge, rc,
// lParam
SHAppBarMessage(ABM_GETTASKBARPOS, &abd);
// abd.rc contains the rectangular location
// of the taskbar in screen coordinates.
```

Z-order

It's now time to look at the z-ordering of appbars. Whenever an appbar window receives a WM_ACTIVATE message, the appbar should notify the shell by executing the following code:

```
 APPBARDATA abd;
abd.cbSize = sizeof(abd);
abd.hWnd = hwndOfWindowBeingActivated;
// Ignored members: uCallbackMessage, uEdge, rc, lParam
SHAppBarMessage(ABM_ACTIVATE, &abd);
```

The ABM_ACTIVATE message tells the shell to move any autohide appbar on the appbar's edge to the foreground. Why? Imagine you have two appbars docked on the bottom of your screen. Both are in the topmost z-order but only one is autohide. Now, if the user works with the non-autohide

appbar, the system activates this window and forces it to cover up the autohide appbar. Because the autohide appbar is completely obscured by the non-autohide appbar, the user has no way to access it. Avoid this situation by sending the ABM_ACTIVATE message to SHAppBarMessage. When this message is sent, the shell always forces autohide appbars to be on top of the non-autohide appbars.

To make sure that autohide appbars are always on top of non-autohide appbars, your appbar must also monitor WM_WINDOWPOSCHANGED messages. When one is received, you must notify the shell:

```
 APPBARDATA abd;
abd.cbSize = sizeof(abd);
abd.hWnd = hwndOfWindowJustPositioned;
// Ignored members: uCallbackMessage,  // uEdge, rc, lParam
SHAppBarMessage(ABM_WINDOWPOSCHANGED,
                &abd);
```

Appbar Notifications

Periodically, the shell needs to send notifications to all the appbars. When you add your appbar to the shell's internal list by sending the ABM_NEW message, you specify a window message callback value. When the shell notifies your appbar of something, it sends your appbar the callback message you specified. When the appbar receives this message, the wParam parameter will be one the appbar notification codes: ABN_FULLSCREENAPP, ABN_POSCHANGED, ABN_STATECHANGE, or ABN_WINDOWARRANGE.

The ABN_FULLSCREENAPP notification is sent when a fullscreen application window is opened or closed. A window is fullscreen when its client area occupies the entire screen. Always-on-top appbars should take themselves out of the topmost z-order so that they do not cover the fullscreen window. lParam contains TRUE if a fullscreen window is opening and FALSE if one is closing.

Unfortunately, there is a serious bug in the shell's handling of the ABN_FULLSCREENAPP notification: appbars receive this notification only if the shell's taskbar is an always-on-top window and if it is not autohide. If the taskbar is not always-on-top or is autohide, your appbar will not get the proper set of ABN_FULLSCREENAPP notifications. Obviously, your appbar should receive these notifications regardless of the taskbar's settings. Microsoft tells me this bug is fixed in future versions of Windows.

The ABN_POSCHANGED notification is sent when any appbar alters its size by sending the ABM_SETPOS message to SHAppBarMessage. This notification is also sent when an appbar is removed by sending the ABM_REMOVE message, and when the taskbar is in autohide mode and the user causes it to slide on or off the screen. You should use this notification to force your appbar to reposition itself by sending an ABM_SETPOS message. Remember that the shell does not send an ABN_POSCHANGED notification to the same appbar that sends an ABM_SETPOS message. The lParam parameter is undefined for this notification.

ABN_STATECHANGE indicates that the user changed the taskbar's state. Using this notification, it is possible to implement your appbar so that its state mimics that of the shell'staskbar,sowhentheuser changes the taskbar's always-on-top and autohide states, your appbar follows suit.

Unfortunately, the implementation of the ABN_STATECHANGE notification exposes another bug in the system. When the user changes the taskbar's state, you should get a single ABN_STATECHANGE notification; you actually get notified once for each state supported by the taskbar. Since the taskbar supports two states (always-on-top and autohide), you get this notification twice even if the user only changes one of the taskbar's states. Microsoft claims this bug is fixed after the first beta of Windows NT™ 4.0 and in future versions of Windows 95.

This can make you run into problems. Imagine that the taskbar and your appbar are both docked at the bottom of the screen. The taskbar is autohide. The user opens the Taskbar Properties dialog (see **Figure 4**) and turns off both the always-on-top and autohide features. Internally, the shell first turns off the always-on-top feature and immediately broadcasts the

ABN_STATECHANGE notification. Because you want your appbar to mimic the state of the taskbar, your appbar responds to this notification by sending an ABM_GETSTATE message. The result of this message indicates that always-on-top is off. Since the shell has not yet had a chance to change the autohide feature of the taskbar, your appbar will think that autohide is on and will try to make itself autohide as well. Since you can't have two autohide appbars docked on the same edge, a message box like the one shown in **Figure 2** will appear. A user will be a bit confused when you display this message box in response to the user just turning autohide off!

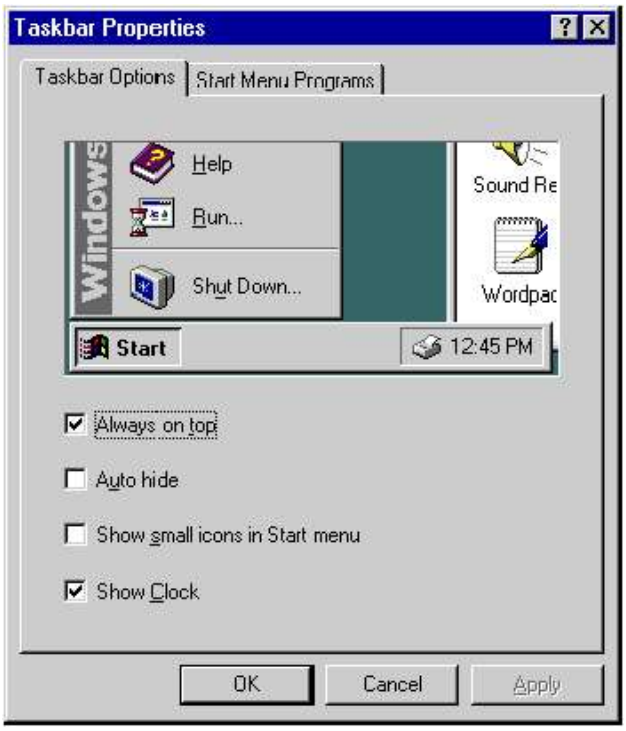

Figure 4 Taskbar properties

My CAppBar class solves this problem by remembering the current state of the taskbar and comparing the current state with the new state returned by sending an ABM_GETSTATE message. Then, for my appbar, I only change the states that have also changed on the taskbar. I implemented my CAppBar class this way so that it will function properly on future versions of Windows.

The last notification, ABN_WINDOWARRANGE, is sent when the user chooses either the cascade, tile horizontally, or tile vertically options from the shell's taskbar. Actually, this notification is sent to each appbar twice. Immediately after the user chooses one of these options, the shell sends this notification with the lParam set to TRUE, indicating that it's about to reposition all the windows. After each appbar processes this notification, the shell repositions all the windows and sends another ABN_WINDOWARRANGE notification. This time, the lParam parameter is set to FALSE to indicate that all the windows have been moved. The taskbar uses the first notification to save the state of all the existing windows before they are moved. This allows the taskbar to offer an Undo option so that the windows can be put back to their original positions.

The CAppBar Class

My CAppBar class encapsulates all of the idiosyncrasies of the new application desktop toolbars. This class was definitely one of the most difficult and challenging projects I've worked on in quite a while. When you examine the source code, you'll see that I have been extremely liberal with comments. The full source code is too long to reprint here, but you can get it electronically from any of the sources listed on page 5. **Figure 5** does show the AppBar.h header file, which contains the CAppBar class description. Now I'll describe how this class is implemented.

At the top of the class definition is a public section that specifiesfourstatichelperfunctions.Thesefunctionsarespecific to appbars, but you do not have to have an instance of aCAppBarcreatedtocallthem. Thefirst twofunctions, IsEdgeLeftOrRight and IsEdgeTopOrBottom, are just convenience functions that check if an edge is left/right or top/bottom.

ResetSystemKnowledge is an interesting function that makes it easier to debug a CAppBar-derived class. The CAppBar constructor calls this function but because this function is static, it can be called at any time. ResetSystemKnowledge forces the shell to send ABN_POSCHANGED notifications to all of the registered appbars in the system so that they will reposition themselves properly.

```
void CAppBar::ResetSystemKnowledge (void) {
  #ifdef _DEBUG
  // Only do this for debug builds.
  APPBARDATA abd;
  abd.cbSize = sizeof(abd);
  abd.hWnd = NULL;
  ::SHAppBarMessage(ABM_REMOVE, &abd);
  #endif
}
```

As you can see, this function just tells the shell to remove a window whose handle is NULL. There will never be a window with a handle of NULL but the shell does broadcast ABN_POSCHANGED notifications to the registered appbars anyway. So why bother? Suppose you are debugging your CAppBar-derived class and its window is docked on the edge of your screen. If you stop debugging, the window will be destroyed but an ABM_REMOVE message is not sent to the shell. The shell still thinks that your window is taking up screen real estate. This causes a blank rectangle to appear on your screen and keeps your workarea smaller than it should be. The next time you run/debug your application, ResetSystemKnowledge is called and your workarea will be configured properly.

The last static helper function is GetEdgeFromPoint. This function takes a set of ABF_XXX flags (defined in AppBar.h) and a point (in screen coordinates). The function compares the point with the rectangle bounding the workarea to determine if the point is closest to the left, top, right or bottom. Then GetEdgeFromPoint checks the flags passed to determine which edges are allowed and whether the appbar should float instead. The function returns ABE_LEFT, ABE_TOP, ABE_RIGHT, ABE_BOTTOM, or ABE_FLOAT.

The CAppBar class has a protected section that defines the internal implementation state variables. In general, a derived class should not alter any of these variables directly with the exception of m_fdwFlags. This member variable allows you to finetune the behavior of a CAppBar. Initialize this variable with a combination of the flags shown in **Figure 6**. Most CAppBar-derived classes will set this member to ABF_ALLOWANYWHERE and not use any of the other flags. You must specify at least one of the ABF_ALLOWLEFTRIGHT, ABF_ALLOWTOPBOTTOM, or ABF_ALLOWFLOAT flags or your appbar will not work correctly because you are telling the CAppBar class that the appbar cannot appear anywhere!

The CAppBar class also defines a protected data structure APPBARSTATE, and a member variable m_abs as an instance of this structure:

```
typedef struct {
  DWORD m_cbSize;         // Size of this structure
  UINT  m_uState;         // ABE_UNKNOWN, ABE_FLOAT, or
                          // ABE_edge
  BOOL  m_fAutoHide;      // Should AppBar be autohide
                          // when docked?
  BOOL  m_fAlwaysOnTop;   // Should AppBar always be on
                          // top?
  UINT  m_auDimsDock[4];  // Width/height for docked bar
                          // on 4 edges
  CRect m_rcFloat;        // Floating rectangle (in
                          // screen coordinates)
} APPBARSTATE, *PAPPBARSTATE;
APPBARSTATE m_abs;        // This AppBar's state info
```

This structure is the driving force behind the CAppBar implementation. It is used by derived classes, but derived classes should never touch the m_abs variable directly-member functions (contained in the public section following this section) exist that allow a derived class to access and alter the m_abs variable. In fact, almost all of the member functions alter the contents of this variable in one way or another. During its initialization, a derived class should allocate an

APPBARSTATE structure for itself, initialize all of the members, and then call the base class's SetState member function:

```
void SetState (APPBARSTATE& abs);
```

This function initializes CAppBar's internal m_abs variable. SetState also notifies the shell of the appbar's location, adjusts the size of the appbar, docks or floats the appbar, autohides the appbar, or adjusts its z-order. In other words, this function does a lot of work.

A derived class acquires the current state of the m_abs variable by calling GetState.

```
void GetState (APPBARSTATE* pabs);
```

There are additional overloaded forms of the SetState and GetState functions as well as other functions that are provided for convenience-all of them manipulate the internal m_abs variable.

The CAppBar class has a protected section in the header file marked "Internal implementation functions." As you can guess, the functions in this section exist for the internal implementation of the CAppBar class. You might find these functions useful, but you probably won't need them.

The SHAppBarMessage function is a wrapper around Win32® SHAppBarMessage. It exists purely as a convenience, which is why it has so many default parameters. It simply allocates an APPBARDATA structure, initializes it, and calls the Win32 SHAppBarMessage function.

CalcProposedState calls GetEdgeFromPoint passing the CAppBar's m_fdwFlags member and the point. This function also checks to see if the user is holding down the Control key forcing the appbar to float rather than dock. This feature allows the user to float an appbar even if the appbar is moved very close to an edge of the screen.

The GetRect function is used heavily throughout the CAppBar class's implementation. This function is passed a proposed state and a proposed rectangle identifying where the appbar should be on the screen. The function then examines the proposed state, sends the shell an ABM_QUERYPOS message and determines where the appbar can really be located. The rectangle pointed to by the prcProposed parameter is updated with the valid rectangle.

The ShowHiddenAppBar and SlideWindow functions are used together. ShowHiddenAppBar shows or hides the appbar if it is an autohide appbar. If the appbar is not autohide, the function just returns. If the appbar is autohide, the function determines the new location of the appbar and then calls the SlideWindow function, which slides the window on or off the screen. The SlideWindow function is not specific to appbars. You can easily steal this function and incorporate it into some other application if you want to be able to slide a window from one location to another.

The CAppBar class also has several protected virtual functions. A CAppBar-derived class will probably want to override some of these functions to get notifications of what's going on inside the base class.

The first overridable function, OnAppBarStateChange, notifies the derived class that the appbar has just changed its state. A derived class might use this notification to add a caption to the window when it is floating or take the caption away when the appbar is docked. The base class's implementation of this function does nothing.

The OnAppBarForcedToDocked function notifies a derived class that the appbar is autohide and that it tried to dock on an edge which already contains an autohide appbar. The derived class usually will not implement this function because the implementation in the base class displays a message box as shown in **Figure 2**. When the base class displays this message box, it first grabs the window's caption text and uses this text for the caption of the message box. You'll always want to give your appbar caption text, even if that caption is not normally displayed as part of the appbar's window.

The last four overridable functions are called when the appbar receives one of the four ABN_XXX notifications. Usually, the derived class will not implement these functions and will just use the base class's implementation.

When OnABNFullScreenApp is called, the base class's implementation keeps track of whether a full screen window is up and adjusts the z-order of the appbar accordingly.

When OnABNPosChanged is called, the base class's implementation adjusts the appbar's window location.

When OnABNStateChanged is called, it is passed a bit mask indicating which of the taskbar's states have changed and what the new states are. The base class's implementation of this function simply calls the MimicState function passing this same information,which has the effect of setting your appbar's state to reflect the taskbar's state depending on whether the ABF_MIMICTASKBARAUTOHIDE or ABF_MIMICTASKBARALWAYSONTOP flagsareset.TheCAppBarclass carefully examines only the states that have changed.

When OnABNWindowArrange is called, the base class's implementation does nothing. Most derived classes will not care about this notification and will not need to supply animplementation for this function.

CAppBar's remaining member functions are all window message handlers (see **Figure 7**).

The CSRBar Class

To test my CAppBar class, I wrote a sample app called SHELLRUN.EXE, which is an MFC version 4.0 dialog-based app. (To obtain complete source code listings, see page 5.) The application main dialog box is not derived directly from CDialog; it's derived instead from my CAppBar class. When you invoke the application, it creates an appbar and docks it by default on the top of your screen (see **Figure 8**).

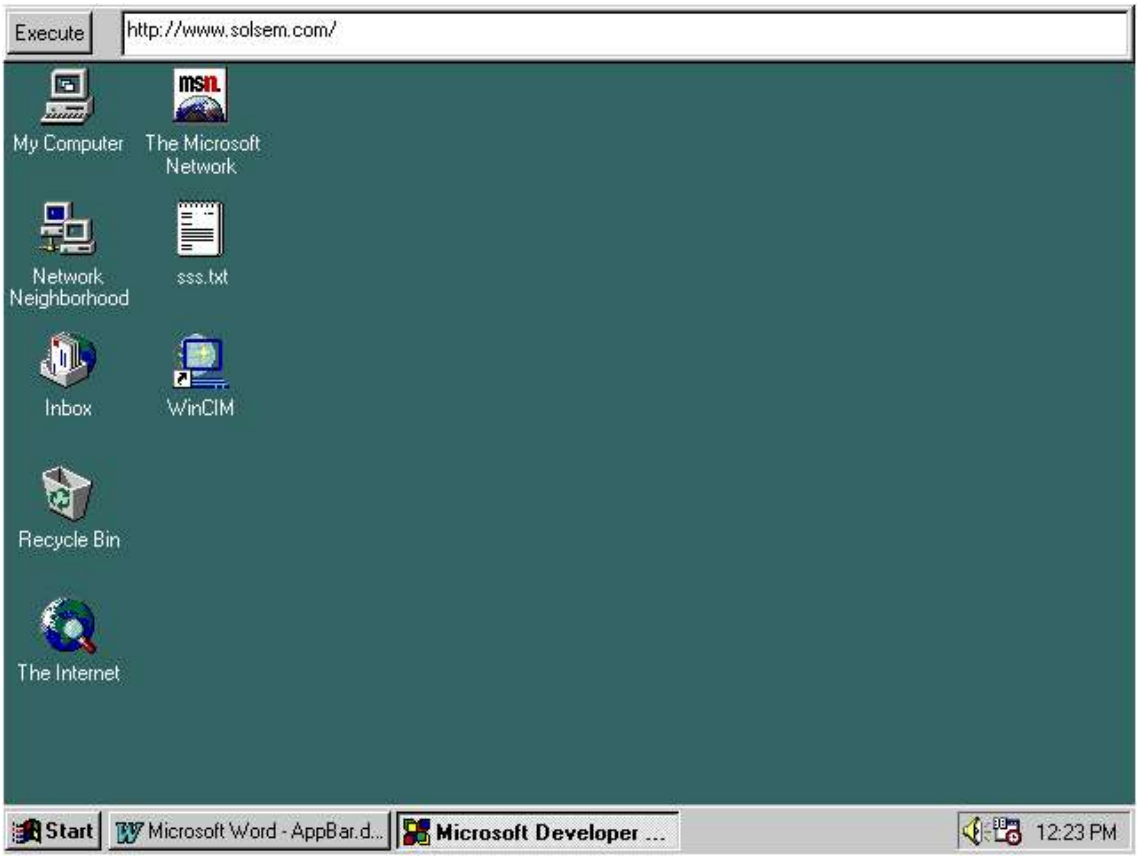

Figure 8

This appbar contains just two controls, an Execute button and an Edit control. In the edit control, you can enter any shell command that you'd like. Pressing Enter or Execute causes the appbar to parse the command and call the Win32 ShellExecute function to invoke it. For shell commands, you can enter program names like Notepad or Calc, drive letters or full paths to open Explorer windows, or you can enter document files and the application will be spawned automatically. You can even enter a web address (as shown in **Figure 8**) and the Internet Explorer will execute.

Because ShellRun is an appbar, you can, of course, dock the window on any edge of the screen; you can even make it float by dropping it in the middle of your screen. To move the appbar, you must click the mouse on the client area of the main window; you cannot click on any of the child windows. When the appbar is floating, it automatically adds a window caption as shown in **Figure 9**. When you design the dialog box template for your appbar, you'll want to make sure that the WS_EX_TOOLWINDOW style is turned on and that the WS_EX_APPWINDOW style is turned off. Tool windows never show up as buttons in the taskbar but application windows always do. By the way, when the system changes the screen's workarea, the system does not reposition any windows that have the WS_EX_TOOLWINDOW style. That's another reason for giving your appbar the WS_EX_TOOLWINDOW style.

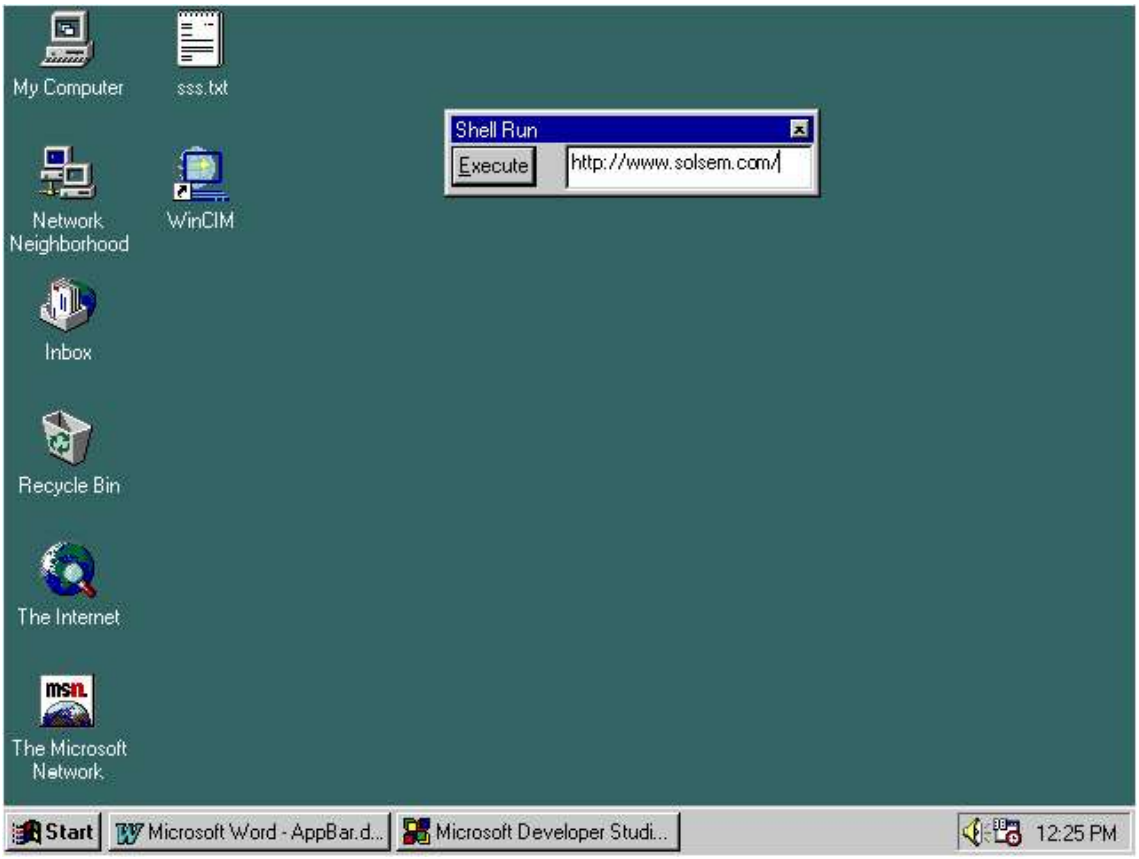

Figure 9

The ShellRun appbar tests some additional features too. When you right-click on the client area of the main window, a context menu appears as shown in **Figure 10**. This menu allows you to change the always-on-top and autohide features of the appbar. You can also display an About box and terminate the application.

Figure 10

Other features of the ShellRun application demonstrate resizing an appbar in discrete increments. You can test this while the appbar is docked, autohide, or floating. The CAppBar base class ensures that the appbar always resizes in discrete increments and your derived class receives WM_SIZE messages as this happens. Of course, you'll want to reorganize the child controls in your dialog box.

It is obvious that Microsoft's software engineers have spent a lot of time and energy on the new system shell. It's also obvious that the shell has gone through some major changes (probably due to usability testing) while Windows 95 was in beta test. Many of these changes have resulted in APIs that are not well-documented. But, with a little (OK, maybe a lot) of patience and

perseverance, adding shell enhancements to your own apps can give them a very professional and polished feel.

From the March 1996 issue of Microsoft Systems Journal.