Development of a Management Support System on the Windows Platform (III-Part 2):

Message Pumping and Message Handling

Hiroshi NOTO

Contents

Section 5 How MFC Uses Message Maps and Handles Messages

5.1 Message Handlers as Window Procedures

5.1.1 Window Classes

Window “classes” in traditional programming for Windows define the characteristics of a “class” (not as a C++ class that defines a user-defined data type and that an object is instantiated from) from which any number of windows can be created. This kind of class is a template or a model for creating windows. In Windows, every window has its window class that defines the attributes of the window such as the window’s icon, the window’s background and the window’s procedure. To create a window class in WinMain(), we call AfxEndDeferRegisterClass() that accepts a WNDCLASS structure defining the properties of

Key words: Command-Routing and Message-Dispatching Architecture, C++ with MFC (Microsoft Foundation Class) Library on Windows, Subclassing and Superclassing, Message Filter Hook and Computer-Based Training Application Hook, Default Window Procedure and Universal Window Procedure

the window class. AfxEndDeferRegisterClass() is already viewed in detail in article (I).*)

5.1.2 Traditional Message Loop

After the UpdateWindow() call (List 4-1 in (I)), the window is fully visible on the display. The application program must now make it ready to read keyboard and mouse input from us. Windows maintains the system queue and a thread message queue for each Windows application. The keyboard device driver, via Windows, converts an event such as keyboard input or mouse input into a "message" and places it in the system queue. When the application program with the input focus attempts to retrieve a message from its application thread message queue and that queue is empty, Windows looks for a message for that application in the system queue. It then transfers keyboard and/or mouse messages from the system queue to the application's thread queue. The program retrieves these messages from its thread queue by executing the following "traditional" block of code known as the message loop which is implemented as a basic while loop[4)-10)]:

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

The MSG structure is a structure that holds all the information about the message. The message data type structure (MSG) is already explained in Section 4 in (III-1).

The call to GetMessage() tells windows to retrieve the first message in the message queue (the thread's message queue, in this case). If a message is available, it is removed from the queue and copied to msg; otherwise, GetMessage() will not return until there is a message available. The return value from GetMessage() depends on the message it retrieved: If it is a WM_QUIT message it will return FALSE, if it is not, it will return TRUE.

The message loop provided by MFC then calls the TranslateMessage() function that translates virtual-key messages (or commonly referred to as *keystroke messages*) involving character keys into character messages (i.e. WM_CHAR messages). In Windows, "virtual key code" is a device-independent value translated from the scan code (i.e. two codes: one when the user presses a key and the other when the user releases the key) by the keyboard device driver. For example, if we get a WM_KEYDOWN message, TranslateMessage() will add a WM_CHAR message to our message queue. This is very useful because the

*) Hereafter the series of our articles "Development of a Management Support System on the Windows Platform"[1)-3)] will be abbreviated as (I), (II), (III-1) or (III-3).

WM_KEYDOWN only tells us what key has been pressed, not the character itself (i.e. “a” or “A”). Suppose the keystroke (the pressed key) is A key. Then the WM_KEYDOWN for the virtual key code VK_A could mean “a” or “A”, depending on the state of the Caps Lock and the Shift key. TranslateMessage() does the work of checking whether it should be capital or not for us. Thus in an MFC application we can use WM_CHAR messages without worrying about virtual key codes and shift states, each WM_CHAR message including a character code that maps directly to a symbol in the ANSI character set (Windows 98) or Unicode character set (Windows 2000 or Windows XP). More specifically, the character messages are posted to the thread’s message queue, to be read the next time the thread calls the GetMessage() function.

The call to DispatchMessage() calls the window procedure associated with the window that received the message. Messages dispatched with DispatchMessage() generate calls to the window procedure WndProc(). In a traditional program for Windows, without MFC, all messages to a window are processed in its window procedure WndProc(). A WndProc() is associated with a window by means of the “window class registration” process. The prototype of WndProc() is shown below in the next subsection along with AfxWndProc() (see the next subsection.). The message loop executes until GetMessage() returns FALSE or 0, which happens only when a WM_QUIT message is retrieved from the message queue. When this occurs, WinMain() ends and the program terminates. The traditional loop provided by MFC, just explained above, is sketched in (List 2-4 in (III-1)).

5.1.3 Window Procedures

A Window Procedure is a function called by the Message Loop. Whenever a message is sent to a window, the message loop looks at the window’s window class and calls the window procedure passing the message’s information. The two window procedures, one the old WndProc() and the other AfxWndProc(), are prototyped as follows:

```
LRESULT CALLBACK WndProc(
    HWND hwnd, // handle to window
    UINT message, // message identifier
    WPARAM wParam, // first message parameter
    LPARAM lParam // second message parameter
);
```

```
LRESULT CALLBACK AfxWndProc(
    HWND hWnd,
    UINT nMsg,
    WPARAM wParam,
    LPARAM lParam
```

```
);
```

The type of handle HWND is the handle to the window that received the message. This parameter is important since we might create more than one window using the same window class. The UINT variable message is the message identifier, and the last two parameters are the parameters sent with the message. Traditionally the typical window procedure is implemented as a set of switch statements, and a call to the default window procedure. The following callback WndProc() is an abbreviated set of switch statements just as an example:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT message,
                          WPARAM wParam, LPARAM lParam)  {
    switch (message)
    {
    case WM_CREATE:
        //Do some initialization, Play a sound or what ever you want
        return 0 ;
    case WM_PAINT:
        //Handle the WM_PAINT message
        return 0 ;
    case WM_DESTROY:
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

The switch-case block inspects the message identifier passed in the message parameter and runs the corresponding message handler. The PostQuitMessage() call sends a WM_QUIT message to the message loop, causing GetMessage() to return FALSE, and the Message Loop to halt. DefWindowProc() provides default processing for any window messages that are irrelevant to our application.

5.2 Message Mapping Architecture

5.2.1 Command-Routing and Message-Dispatching

As is seen in 5.1, the messages are handled, at the lowest level, by the message loop, MFC's conventional fashion of command-routing architecture. The MFC class library, however, provides some extra ways of handling messages more efficiently and neatly without relying on a set of switch statements and v-tables which are assigned to the WndProc() procedure. Instead of associating the specific message handler with a window, an MFC program uses message maps to get commands and messages to a command target which is

derived from the class CWnd.[7)–10)] Message maps associate specific commands and messages with member functions that handle those commands and messages. At the heart of message-mapping architecture lies a function AfxWndProc(). As we have already seen in Section 4 in (I) that there are four window classes to be registered. And the AfxWndProc() is not directly installed through those window classes. However AfxWndProc() is eventually wired by using Windows hooking mechanism: The CWnd class has a member function WindowProc(); AfxWndProc() ends up calling CWinApp's m_pMainWnd->WindowProc(); WindowProc() then routes the message to the right command target.

As is explained in (III-1), there are three types of messages: windows messages, command notifications, and command messages.[16)] The three messages and commands are categorized into two groups: windows messages (WM_ messages) and commands (WM_COMMAND notification messages). The MFC message maps deal with both window messages and commands.

Although there appears no window procedure AfxWndProc() explicitly seen for each of the four window classes, the commands and messages are surely handled by MFC's command-routing and message-dispatching architecture, respectively. We are going to see in the next article (III-3) in detail that MFC handles WM_COMMAND commands through the command-routing mechanism, which passes a WM_COMMAND to the appropriate command target and that MFC handles a window message through the message-dispatching mechanism, which passes a WM_ massage to the appropriate member function of the window.

The MFC window classes are registered with DefWindowProc(), not with AfxWndProc() as the message handler, which we have already seen in the AfxEndDeferRegisterClass() function in (II). The functionality associated with windows is to ensure that every message in the window is processed. This means that Windows should handle any message we do not process in our responsibility. The call to DefWindowProc() gives a shot at those messages which are fundamental to the window functionality and behavior but irrelevant to our application. On the other hand, the command messages we are concerned with finally end up in AfxWndProc() and are dispatched to various CWnd-derived command target objects. It is crucial, therefore, to understand how and why MFC sets up its own window procedure AfxWndProc(), since this AfxWndProc() lies at the heart of MFC. The entire message map mechanism whereby MFC routes commands and window messages (such as WM_SETFOCUS) to their handler functions (like OnSetFocus()) depends on subclassing (or superclassing) each MFC-controlled window with the MFC universal window procedure, AfxWndProc() as the message handler. Here "subclassing" (or superclassing) is the Windows term for replacing the WndProc of a window with a different WndProc (or calling the old WndProc for default (superclass) functionality).[17)] (see the next subsection.)

5.2.2 Subclassing and Superclassing in Hooking

How is it that the commands and messages end up in AfxWndProc()? We are now going

to explain the hooking mechanism which enables us to reach eventually AfxWndProc(). MFC maintains two of its own message hooks. The first hook is a message filter hook. A message filter hook monitors messages generated as a result of input events in a dialog box, message box, menu, or scroll bar. MFC installs the function _AfxMsgFilterHook() as the message hook. The second hook is to support computer-based training applications. MFC does not specifically support computer-based training. MFC installs the function _AfxCbtFilterHook() just so that MFC can hook in as soon as a window is created. MFC cannot wait until CreateWindowEx() returns, because by then a number of messages have been sent. The CWnd object has to be hooked up and subclasssed before any messages are delivered to the window. Because messages are directed to the hooks before anything else happens, there is an opportunity for certain messages to be intercepted in one of these hooks. As it turns out, MFC uses the computer-based training hook function to attach AfxWndProc() to the MFC windows as they are created. Here is an answer to how AfxWndProc() is hooked up to MFC's windows.

As we have reviewed the message handling mechanism so far, in MFC all windows are eventually based on the same window procedure (AfxWndProc()). AfxWndProc() once detects the window (HWND) that a message applies to and dispatches the message to a virtual function of a CWnd object (OnWndMsg()). The object parses the associated message map and calls the required handler (if any) or calls the default handler that - in turns - passes the original message to DefWindowProc() or to an original window procedure existing if the window was not created by MFC.

It might well be useful for an application to change the "default" window procedure dynamically at runtime depending on the messages the window receives. *Subclassing* is such a technique that allows an application to intercept and process messages sent or posted to a particular window before the window has a chance to process them by the "default" window procedure. By subclassing a window, an application can augment, modify, or monitor the behavior of the window. An application can subclass a window belonging to a system global window class, such as an edit box or a combo box. For example, an application could subclass an edit box control to prevent the control from accepting certain characters.

The need of subclassing happens when we have to make a particular task over a particular message for a variety of different windows. Each different window may be a CWnd-derived object, but if we have to trap some messages (for example to customize menu behavior or appearance in the same way for all our windows) we have to re-implement the same handlers for all the CWnd classes. That is where subclassing may be useful: We create another object that intercepts the window procedures, and associate an instance of the object for each of the window. This object defines what to do with the messages and calls the original window procedure when needed.

On the other hand, *Superclassing* is a technique that allows an application to create a new window class with the basic functionality of the existing class, plus enhancements provided by the application. A superclass is based on an existing window class called the *base class*.

Frequently, the base class is a system global window class such as an edit box control, but it can be any window class. A superclass has its own window procedure, called the superclass procedure. The superclass procedure can take three actions upon receiving a message: It can pass the message to the original window procedure, modify the message and pass it to the original window procedure, or process the message and not pass it to the original window procedure. If the superclass procedure processes a message, it can do so before, after, or both before and after it passes the message to the original window procedure.

Unlike a subclass procedure, a superclass procedure can process window creation messages (WM_NCCREATE, WM_CREATE, and so on), but it must also pass them to the original base-class window procedure so that the base-class window procedure can perform its initialization procedure.

In general "subclassing" or "superclassing" is the Windows term for replacing the WndProc of a window with a different WndProc or calling the old WndProc for default (superclass) functionality. This should not be confused with C++ class derivation (The C++ terminology uses the words "base" and "derived" while the Windows object model uses "super" and "sub".). C++ derivation with MFC and Windows subclassing are functionally very similar, except C++ does not support a feature similar to dynamic subclassing.[6), 11)-15)]

5.2.3 _AfxCbtFilterHook()

MFC installs the _AfxCbtFilterHook() function whenever a new CWnd-derived object is created.[8)] In Section 4 Registering Window Classes in (II), we have seen that CFrameWnd::LordFrame() calls AfxDeferRegisterClass() which, by definition, invokes a helper function AfxEndDeferRegisterClass() for registration of the window classes. The helper function initializes WNDCLASS::lpfnWndProc to DefWndProc(). The definition of this window procedure is shown in the List 5-1 below. MFC window classes are registered with DefWindowProc() as the default message handler.

List 5-1. CWnd::DefWindowProc() in WINCORE.CPP

```
LRESULT CWnd::DefWindowProc(UINT nMsg, WPARAM wParam, LPARAM
lParam)
{
// Default CWnd implementation
    if (m_pfnSuper != NULL)
        return ::CallWindowProc(m_pfnSuper, m_hWnd, nMsg, wParam, lParam);

    WNDPROC pfnWndProc;
    if ((pfnWndProc =*GetSuperWndProcAddr()) == NULL)
        return ::DefWindowProc(m_hWnd, nMsg, wParam, lParam);
    else
```

```
        return ::CallWindowProc(pfnWndProc, m_hWnd, nMsg, wParam, lParam);
}
```

In List 4-3 in (II) the application's CFrameWnd::LordFrame() further calls the member function CFrameWnd::Create() (which then calls CWnd::CreateEx()) with the registered window class name on which we will create a Windows window. As is shown in List 5-2 in (II), right before CWnd::CreateEx() makes a call to the CreateWindowEx() API function, CWnd::CreateEx() calls AfxHookWindowCreate() (List 5-2 below). And after creating the window, CWnd::CreateEx() calls AfxUnhookWindowCreate(). These functions set up and remove a flag which signifies WH_CBT (computer-based training) hook.

Windows calls a WH_CBT hook whenever creating or destroying a window. AfxHookWindowCreate() inserts _AfxCbtFilterHook() passing the information about the window being created (see List 5-2 in WINCORE.CPP). The call to SetWindowsHookEx() function in AfxHookWindowCreate() installs an application-defined hook procedure into a hook chain. We could install a hook procedure to monitor the system for certain types of events. A hook is a callback function in our program that is called from Windows whenever certain events happen. Because _AfxCbtFilterHook() is a computer-based training hook as noted a little before, Windows calls _AfxCbtFilterHook() before activating, creating, destroying, minimizing, maximizing, moving, or sizing a window. Windows also calls _AfxCbtFilterHook() before completing a system command, before removing a mouse or keyboard event from the system message queue, before setting the keyboard focus, and before synchronizing with the system message queue.

List 5-2. AfxHookWindowCreate() in WINCORE.CPP

```
void AFXAPI AfxHookWindowCreate(CWnd* pWnd)
{
    _AFX_THREAD_STATE* pThreadState = _afxThreadState.GetData();
    if (pThreadState->m_pWndInit == pWnd)
        return;

    if (pThreadState->m_hHookOldCbtFilter == NULL)
    {
        pThreadState->m_hHookOldCbtFilter = ::SetWindowsHookEx(WH_CBT,
            _AfxCbtFilterHook, NULL, ::GetCurrentThreadId());
        if (pThreadState->m_hHookOldCbtFilter == NULL)
            AfxThrowMemoryException();
    }
    ASSERT(pThreadState->m_hHookOldCbtFilter != NULL);
    ASSERT(pWnd != NULL);
```

```
    ASSERT(pWnd->m_hWnd == NULL); // only do once

    ASSERT(pThreadState->m_pWndInit == NULL); // hook not already in progress
    pThreadState->m_pWndInit = pWnd;
}
```

Then it is from here that _AfxCbtFilterHook() subclasses the window by installing the generic AfxWndProc(). _AfxCbtFilterHook(), however, does not use AfxWndProc() directly as the window procedure; in other words, how does the hook function get Windows to use AfxWndProc()?

There are two ways for MFC to subclass the window,[17] depending on whether we create the window from scratch or we subclass an existing window such as a dialog box control or the like.

5.2.3.1 Creating Our Own Window

We are now in _AfxCbtFilterHook() that sits there receiving messages. As shown in List 5-3, _AfxCbtFilterHook() ignores window messages until the HCBT_CREATEWND code is passed into _AfxCbtFilterHook(). When we start from scratch creating our window, _AfxCbtFilterHook() is called with the HCBT_CREATEWND code, that means a window is about to be created. In List 5-3, we find that the hook function attatches HWND hWnd to the object being created (pWndInit).

List 5-3. _AfxCbtFilterHook() in WINCORE.CPP

```
LRESULT CALLBACK
_AfxCbtFilterHook(int code, WPARAM wParam, LPARAM lParam)
{
    _AFX_THREAD_STATE* pThreadState = _afxThreadState.GetData();
    if (code != HCBT_CREATEWND)
    {
        // wait for HCBT_CREATEWND just pass others on...
        return CallNextHookEx(pThreadState->m_hHookOldCbtFilter, code,
            wParam, lParam);
    }

    ASSERT(lParam != NULL);
    LPCREATESTRUCT lpcs = ((LPCBT_CREATEWND)lParam)->lpcs;
    ASSERT(lpcs != NULL);

    CWnd* pWndInit = pThreadState->m_pWndInit;
    BOOL bContextIsDLL = afxContextIsDLL;
    if (pWndInit != NULL || (!(lpcs->style & WS_CHILD) && !bContextIsDLL))
    {
```

```
        // Note: special check to avoid subclassing the IME window
        if (_afxDBCS)
        {
            // check for cheap CS_IME style first...
            if (GetClassLong((HWND)wParam, GCL_STYLE) & CS_IME)
                goto lCallNextHook;

            // get class name of the window that is being created
            LPCTSTR pszClassName;
            TCHAR szClassName [_countof("ime")+1];
            if (HIWORD(lpcs->lpszClass))
            {
                pszClassName = lpcs->lpszClass;
            }
            else
            {
                szClassName [0] = '¥0';
                GlobalGetAtomName((ATOM)lpcs->lpszClass, szClassName,
                                  _countof(szClassName));
                pszClassName = szClassName;
            }

            // a little more expensive to test this way, but necessary...
            if (lstrcmpi(pszClassName, _T("ime")) == 0)
                goto lCallNextHook;
        }

        ASSERT(wParam != NULL); // should be non-NULL HWND
        HWND hWnd = (HWND)wParam;
        WNDPROC oldWndProc;
        if (pWndInit != NULL)
        {
#ifdef _AFXDLL
            AFX_MANAGE_STATE(pWndInit->m_pModuleState);
#endif

            // the window should not be in the permanent map at this time
            ASSERT(CWnd::FromHandlePermanent(hWnd) == NULL);

            // connect the HWND to pWndInit...
            pWndInit->Attach(hWnd);
            // allow other subclassing to occur first
            pWndInit->PreSubclassWindow();

            WNDPROC* pOldWndProc = pWndInit->GetSuperWndProcAddr();
            ASSERT(pOldWndProc != NULL);

#ifndef _AFX_NO_CTL3D_SUPPORT
```

```
                _AFX_CTL3D_STATE* pCtl3dState;
                DWORD dwFlags;
                if (!afxData.bWin4 && !bContextIsDLL &&
                    (pCtl3dState = _afxCtl3dState.GetDataNA()) != NULL &&
                    pCtl3dState->m_pfnSubclassDlgEx != NULL &&
                    (dwFlags = AfxCallWndProc(pWndInit, hWnd,
                                              WM_QUERY3DCONTROLS)) != 0)
                {
                    // was the class registered with AfxWndProc?
                    WNDPROC afxWndProc = AfxGetAfxWndProc();
                    BOOL bAfxWndProc = ((WNDPROC)
                        GetWindowLong(hWnd, GWL_WNDPROC) ==
                        afxWndProc);

                    pCtl3dState->m_pfnSubclassDlgEx(hWnd, dwFlags);

                    // subclass the window if not already wired to AfxWndProc
                    if (!bAfxWndProc)
                    {
                        // subclass the window with standard AfxWndProc
                        oldWndProc = (WNDPROC)SetWindowLong(hWnd,
                        GWL_WNDPROC,
                            (DWORD)afxWndProc);
                        ASSERT(oldWndProc != NULL);
                        *pOldWndProc = oldWndProc;
                    }
                }
                else
#endif
                {
                    // subclass the window with standard AfxWndProc
                    WNDPROC afxWndProc = AfxGetAfxWndProc();
                    oldWndProc = (WNDPROC)SetWindowLong(hWnd,
                      GWL_WNDPROC, (DWORD)afxWndProc);
                    ASSERT(oldWndProc != NULL);
                    if (oldWndProc != afxWndProc)
                        *pOldWndProc = oldWndProc;
                }
                pThreadState->m_pWndInit = NULL;
            }
            else
            {
                ASSERT(!bContextIsDLL); // should never get here

                // subclass the window with the proc which does gray backgrounds
                oldWndProc = (WNDPROC)GetWindowLong(hWnd,
                GWL_WNDPROC);
                if (oldWndProc != NULL && GetProp(hWnd, _afxOldWndProc) ==
                                                        NULL)
```

```
                {
                    SetProp(hWnd, _afxOldWndProc, oldWndProc);
                    if ((WNDPROC)GetProp(hWnd, _afxOldWndProc) ==
                        oldWndProc)
                    {
                        GlobalAddAtom(_afxOldWndProc);
                        SetWindowLong(hWnd, GWL_WNDPROC,
                            (DWORD)(pThreadState->m_bDlgCreate ?
                                _AfxGrayBackgroundWndProc :
                                    _AfxActivationWndProc));
                        ASSERT(oldWndProc != NULL);
                    }
                }
            }
        }

lCallNextHook:
        LRESULT lResult = CallNextHookEx(pThreadState->m_hHookOldCbt
            Filter, code, wParam, lParam);
#ifndef _AFXDLL
        if (bContextIsDLL)
        {
            ::UnhookWindowsHookEx(pThreadState->m_hHookOldCbtFilter);
            pThreadState->m_hHookOldCbtFilter = NULL;
        }
#endif
        return lResult;
}
```

Then CWnd::GetSuperWndProcAddr() returns an address where the old window procedure can be saved. This virtual function has a simple default implementation (List 5-4):

List 5-4. CWnd::GetSuperWndProcAddr() in WINCORE.CPP

```
WNDPROC* CWnd::GetSuperWndProcAddr()
{
    // Note: it is no longer necessary to override GetSuperWndProcAddr
    // for each control class with a different WNDCLASS.
    // This implementation now uses instance data, such that the previous
    // WNDPROC can be anything.

    return &m_pfnSuper;
}
```

Before saving the old window procedure, MFC calls PreSubclassWindow(), another CWnd virtual function that allows other subclassing to occur first. Finally the hook calls a function AfxGetAfxWndProc() (as shown in List 5-5) to get the window procedure (see List 5-3 _AfxCbtFilterHook.). If the window has the class type registered with AfxWndProc(), AfxGetAfxWndProc() simply returns the pointer to the AfxWndProc() in a non-DLL application, but in the case of a DLL the window procedure is stored in the module state (see AFX_MANAGE_STATE macro in AFXSTAT_.H). That is because the DLL must use the window procedure of whichever application is calling it at any given time, not its own.

List 5-5. AfxGetAfxWndProc() in WINCORE.CPP

```
WNDPROC AFXAPI AfxGetAfxWndProc()
{
#ifdef _AFXDLL
    return AfxGetModuleState()->m_pfnAfxWndProc;
#else
    return &AfxWndProc;
#endif
}
```

In List 5-3, the hook function, here, uses SetWindowLong() to wire AfxWndProc() up to the window, passing SetWindowLong() three parameters: the handle to the window to subclass, the GWL_WNDPROC flag and the address of the (new) subclass procedure. The index GWL_WNDPROC specifies the action of replacing the window procedure. The return value of SetWindowLong() is the address of the (previous) original window procedure.

Once _AfxCbtFilterHook() is finished, the window is hooked up. Control flows out of _AfxCbtFilterHook() and gets back to CWnd::CreateEx() which now calls AfxUnhookWindowCreate() to remove the CBT hook. As a result the role of the CBT hook is to trap the window's creation so that MFC can subclass the window, that is, install AfxWndProc().

Here shown is AfxWndProc() in List 5-6 which is stored in WINCORE.CPP, where the type definition LRESULT is declared as long data type, and the keyword CALLBACK specifies the __stdcall calling sequence to be used to push some number of parameters on the stack in right-to-left ordering so as to pass them to the function (To see this, move the caret to our desired data type and press F12, while interacting in VC++ 6.0.). AfxWndProc() returns an LRESULT value. A callback function is a function in an application that Windows calls back with information requested.

List 5-6. AfxWndProc() in WINCORE.CPP

```
LRESULT CALLBACK
AfxWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)
{
// The WndProc for all CWnd's and derived classes
    // special message which identifies the window as using AfxWndProc
    if (nMsg == WM_QUERYAFXWNDPROC)
        return 1;

    // all other messages route through message map
    CWnd* pWnd = CWnd::FromHandlePermanent(hWnd);
    ASSERT(pWnd != NULL);
    ASSERT(pWnd->m_hWnd == hWnd);
    return AfxCallWndProc(pWnd, hWnd, nMsg, wParam, lParam);
}
```

5.2.3.2 Subclassing an Existing Window

The second case is where we subclass an existing window. Suppose we are again in _AfxCbtFilterHook(). An example of "3D (three dimensional) controls and dialogs" is displayed between the conditional compilation groups #ifndef _AFX_NO_CTL3D_SUPPORT and #endif in List 5-3. It is said that the main reason Microsoft shifted from using AfxWndProc() as the registered window procedure to using DefWindowProc() is to support 3D controls which work through Microsoft's CTL3D. DLL (or CTL3D32.DLL).[8] In order for 3D controls to work, MFC had to ensure that the subclassing is in the following order: DefWindowProc() → CTL3D's WndProc() → AfxWndProc(). What this ordering indicates is that Microsoft had to allow CTL3D to subclass before AfxWndProc(), which means delaying hooking up AfxWndProc() until after pCtl3dState-> m_pfnSubclassDlgEx() is called. In this example, MFC registers everything with DefWindowProc(), subclasses calling CTL3D. DLL (or CTL3D32.DLL), and subclasses the window by calling SetWindowLong() to install AfxWndProc() through AfxGetAfxWndProc().

The application framework also subclasses an instance of a window by directly calling the SetWindowLong() function without using any hooks. The subclass procedure can reside in either the application's executable or a dynamic-link library (DLL). SetWindowLong() returns the address of the window's original window procedure. The application must save the address and use it in subsequent calls to the AfxCallWndProc() (or CallWindowProc()) function, in order to pass intercepted messages to the original window procedure when the messages are not processed by the new window procedure. The application must also have the original window procedure address to remove the subclass from the window. To remove

the subclass, the application calls SetWindowLong() again, passing the address of the original window procedure with the GWL_WNDPROC flag and the handle to the window.

5.2.4 _AfxMsgFilterHook()

_AfxMsgFilterHook() is another MFC hook procedure for the flag WH_MSGFILTER hook.[18] The WH_MSGFILTER hook is a task-specific hook that enables an application to monitor messages passed to a menu, scroll bar, message box, or dialog box created by the application that installed the hook procedure. _AfxMsgFilterHook() procedure is called by the application after a message generated by an input event (in a dialog box, message box, menu, or scroll bar) is retrieved from the message queue. Since messages sent directly to the window procedure (by SendMessage()) do not go through the message queue, this hook procedure cannot be called by Windows and cannot be used to monitor messages that are sent (by "Windows" or by the user) to a dialog box, message box, menu, or a scroll bar.

In a typical MFC application the WH_MSGFILTER is set in the global function AfxInitThread() with _AfxMsgFilterHook() as the callback or the hook procedure. The callback function for this hook is called after these messages are retrieved from the queue, just before dispatching them. ProcessMessageFilter() of CWinThread is called from the _AfxMsgFilterHook() hook procedure.

In general, all the keyboard and mouse messages, along with the WM_PAINT and WM_TIMER messages, are posted to the message queue. ProcessMessageFilter() will not be called for messages like WM_SETFOCUS, WM_KILLFOCUS, WM_SETCURSOR, WM_COMMAND, WM_CTLCOLOR, WM_ACTIVATE, etc., since these are sent directly to the window procedure.

Here we are going to trace the _AfxMsgFilterHook() (i.e. where the hook procedure is installed into a hook chain and where its installation calls the filter function) in our application MSS (Management Support System).[1] Now we pursue _AfxMsgFilterHook() starting with AfxWinMain(), followed by the functional chain that reaches the _AfxMsgFilterHook() that in turn calls ProcessMessageFilter() to monitor messages retrieved from the message queue: AfxWinMain() → AfxWinInit() → AfxInitThread() → ::SetWindowsHookEx() → _AfxMsgFilterHook() → ProcessMessageFilter().

Suppose we start the MSS application, control immediately arrives at AfxWinMain() which is shown in List 3-3 in (I). AfxWinMain() calls AfxWinInit() for AFX internal initialization. At the very end of the AfxWinInit()(List 3-5 in (II)) AfxInitThread() is called that is shown in List 5-7.

List 5-7. AfxInitThread() in THRDCORE.CPP

```
void AFXAPI AfxInitThread()
{
    if ( ! afxContextIsDLL)
```

```
    {
        //set message filter proc
        _AFX_THREAD_STATE* pThreadState = AfxGetThreadState();
        ASSERT(pThreadState->m_hHookOldMsgFilter == NULL);
        pThreadState->m_hHookOldMsgFilter=::SetWindowsHookEx(WH_MSGFILTER,
            _AfxMsgFilterHook, NULL, ::GetCurrentThreadId());

#ifndef _AFX_NO_CTL3D_SUPPORT
        // intialize CTL3D for this thread
        _AFX_CTL3D_STATE* pCtl3dState = _afxCtl3dState;
        if (pCtl3dState->m_pfnAutoSubclass != NULL)

    (*pCtl3dState->m_pfnAutoSubclass)(AfxGetInstanceHandle());

        // allocate thread local _AFX_CTL3D_THREAD just for automatic
termination
        _AFX_CTL3D_THREAD* pTemp = _afxCtl3dThread;
        pTemp; // avoid unused warning
#endif
    }
}
```

AfxInitThread() is a global function for thread initialization and thread cleanup. In the middle of List 5-7 we find that the call to ::SetWindowsHookEx(), as in the case of _AfxCbtFilterHook(), sets the message filter procedure _AfxMsgFilterHook() along with the type of hook procedure WH_MSGFILTER. List 5-8 displays the content of this callback function.

List 5-8. _AfxMsgFilterHook() in THRDCORE.CPP

```
LRESULT CALLBACK _AfxMsgFilterHook(int code, WPARAM wParam, LPARAM
lParam)
{
    CWinThread* pThread;
    if (afxContextIsDLL || (code < 0 && code != MSGF_DDEMGR) ||
        (pThread = AfxGetThread()) == NULL)
    {
        return ::CallNextHookEx(_afxThreadState->m_hHookOldMsgFilter,
            code, wParam, lParam);
    }
```

```
    ASSERT(pThread != NULL);
    return (LRESULT)pThread->ProcessMessageFilter(code, (LPMSG)lParam);
}
```

In List 5-8 _AfxMsgFilterHook() returns CWinThread::ProcessMessageFilter() at the very end of the function. In List 5-9 shown is the process message filter CWinThread::ProcessMessageFilter().

List 5-9. CWinThread::ProcessMessageFilter() in THRDCORE.CPP

```
BOOL CWinThread::ProcessMessageFilter(int code, LPMSG lpMsg)
{
    if(lpMsg == NULL)
        return FALSE; // not handled

    CFrameWnd* pTopFrameWnd;
    CWnd* pMainWnd;
    CWnd* pMsgWnd;
    switch (code)
    {

    case MSGF_DDEMGR:
        // Unlike other WH_MSGFILTER codes, MSGF_DDEMGR should
        //  never call the next hook.
        // By returning FALSE, the message will be dispatched
        //  instead (the default behavior).
        return FALSE;

    case MSGF_MENU:
        pMsgWnd = CWnd::FromHandle(lpMsg->hwnd);
        if (pMsgWnd != NULL)
        {
            pTopFrameWnd = pMsgWnd->GetTopLevelFrame();
            if (pTopFrameWnd != NULL &&
              pTopFrameWnd->IsTracking() && pTopFrameWnd->m_bHelpMode)
            {
                pMainWnd = AfxGetMainWnd();
                if ((m_pMainWnd != NULL) && (IsEnterKey(lpMsg)
                  || IsButtonUp(lpMsg)))
                {
```

```
    pMainWnd->SendMessage(WM_COMMAND, ID_HELP);
                    return TRUE;
                }
            }
        }
        //  fall through...

    case MSGF_DIALOGBOX:      // handles message boxes as well.
        pMainWnd = AfxGetMainWnd();
        if (afxData.nWinVer < 0x333 && pMainWnd != NULL &&
IsHelpKey(lpMsg))
        {
            pMainWnd->SendMessage(WM_COMMAND, ID_HELP);
            return TRUE;
        }

        if (code == MSGF_DIALOGBOX && m_pActiveWnd != NULL &&
            lpMsg->message >= WM_KEYFIRST && lpMsg->message <=
WM_KEYLAST)
        {
            // need to translate messages for the in-place container
            _AFX_THREAD_STATE* pThreadState = _afxThreadState.GetData();
            if (pThreadState->m_bInMsgFilter)
                return FALSE;
            pThreadState->m_bInMsgFilter = TRUE; // avoid reentering this code
            MSG msg = *lpMsg;
            if (m_pActiveWnd->IsWindowEnabled() && PreTranslateMessage(&msg))
            {
                pThreadState->m_bInMsgFilter = FALSE;
                return TRUE;
            }
            pThreadState->m_bInMsgFilter = FALSE; // ok again
        }
        break;
    }

return FALSE; // default to not handled
}
```

Immediately after ::SetWindowsHookEx(WH_MSGFILTER, _AfxMsgFilterHook, NULL,

::GetCurrentThreadId()) is called, our MSS application displays a splash screen and a pop-up dialog box menu appears (Fig. 2-1 in (I)). Then we realize that control arrives at CWinThread::ProcessMessageFilter() through _AfxMsgFilterHook() with the flag WH_MSGFILTER and comes into the switch statement and jumps to the case label MSGF_DIALOGBOX. Thus we can monitor the message generated by an input event in the dialog box, after it is retrieved from the message queue.

5.2.5 MFC Windows Wired to AfxWndProc()

Once _AfxCbtFilterHook() is finished, the window is hooked up. Now why do we have to call SetWindowLong() function to subclass the window (e.g. change the window procedure) after the window is created? The reason is that our window object could never get a chance to handle WM_CREATE and WM_NCCREATE, because these messages are sent by Windows itself from within ::CreateWindowEx() where Windows is creating the window. Then how can MFC hook up our window before MFC calls CreateWindowEx()? The answer is that "MFC sets a hook before Windows sends any messages to the window procedures." At this point of the control flow, Windows has already called the CBT hook after Windows created the window. Therefore MFC can get both a valid HWND type of the window and the WH_CBT hook. We emphasize again that the role of the CBT hook is to trap the window's creation and to enable MFC to subclass the window, that is, to install AfxWndProc().[17)]

From now on, messages for that window will go to AfxWndProc(), where they are handled by command-routing and message-dispatching architecture. So even though the window was originally registered with DefWindowProc() as the message-handling procedure, the framework effectively wires the windows up to AfxWndProc() whenever a CWnd-derived window is created.[8)]

It should be noted finally again that the reason for the existence of CBT hook is to trap the window's creation so that MFC can subclass the window to install AfxWndProc(). It is important to realize that the message-routing code is universal for all window classes and that MFC uses the same universal AfxWndProc() for all window objects.

BIBLIOGRAPHY

(1) Noto, Hirosi. Development of a Management Support System On the Windows Platform (I): Class structure of MFC and creation of user-defined classes, Hokusei Review, The School of Economics (Hokusei Gakuen University) Vol. 42, No. 2, March 2003.

(2) Noto, Hirosi. Development of a Management Support System on the Windows Platform (II): Registering Window Classes and Creating the Main Window, Hokusei Review, The School of Economics (Hokusei Gakuen University) Vol. 43, No. 2, March 2004.

(3) Noto, Hirosi. Development of a Management Support System on the Windows Platform (III-Part 1): Message Pumping and Message Handling, Hokusei Review, The School of Economics (Hokusei Gakuen University) Vol. 44, No. 1, March 2005.

(4) Brent E. Rector and Joseph M. Newcomer. Win32 Programming, Addison Wesley, 1997.

(5) Charles Petzold. Programming Windows 5th Edition, Microsoft Press, 1999.

(6) http://msdn.microsoft.com/library/default.asp?URL=/library/devprods/vs6/visualc/vctutor/tutorhm.htm

(7) http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_mfc_class_library_reference_introduction.asp

(8) George Shepherd and Scot Wingo. MFC Internals, Addison Wesley Developers Press, 1997.

(9) Jeff Prosise. Programming Windows with MFC 2nd Edition, Microsoft Press, 1999.

(10) Aran R. Feuer. MFC Programming, Addison Wesley, 1997.

(11) David J. Kruglinski, George Shepherd, and Scot Wingo. Programming Visual C++ (fifth edition), Microsoft Press, 1998.

(12) Stephen D. Gilbert and Bill McCarty. Visual C++ 6 Programming Blue Book, CORIOLIS, 1999.

(13) Hayasi, Haruhiko. A New Introduction to Visual C++ Ver. 5.0 (Beginners edition) (in Japanese), SoftBank Books, 1998.

(14) Yosida, Kouitirou. Kiwameru Visual C++, Gijutu (in Japanese) Hyouron-sya, 1998.

(15) Yamasita, Hirosi, Kuroba, Hiroaki, and Kuroiwa, Kentarou. C++ Programming Style (in Japanese), Ohmsha, 1994.

(16) http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_mfc_msg_structure.asp

(17) Paul DiLascia, Microsoft System Journal (1999)
http://www.microsoft.com/msj/0699/c/c0699.aspx

(18) THE MICROSOFT KNOWLEDGE BASE
http://www.sunsite.org.uk/sites/ftp.microsoft.com/MISC1/DEVELOPR/VISUAL_C/KB/Q166/2/12.TXT

[Abstract]

Development of a Management Support System on the Windows Platform (III-Part 2):

Message Pumping and Message Handling

Hiroshi NOTO

This paper studies the mechanism of message pumping and message handling on the Windows platform. The architecture of processing messages forms the core of the Windows Programming Model that realizes the event-driven programming technique on it. Windows calls the function associated with a window when an event occurs that might affect the window, passing messages in the argument of the call that describe the event. The message pump is a program loop that retrieves input messages from the application queue, translates them, and dispatches them to the relevant window procedures (i.e. functions). In the C++ processor with MFC (Microsoft Foundation Class) class library, the message routing and handling system called "message mapping" is implemented. MFC's message mapping technology neatly associates window messages and commands to the member functions of classes in windows. MFC provides message macros to generate message maps, which expand into code that defines and implements a message map for a CCmdTarget-based class. MFC's standard message-mapping is a reasonable alternative to handling messages via virtual class member functions, which have been carried out on the original Windows. The MFC's standard message-mapping eliminates the overhead of erroneous vtables (virtual function tables), it is compiler independent, and it is fairly efficient. It is possible to have a good grasp of how MFC handles the application aspect (initialization and message pump) and the window aspect (message handling) of a Windows application program by taking a close look at internals of MFC and by keeping track of the function calling series triggered by PumpMessage() of our own MSS (Management Support System) application as an example of message pumping and message handling.

Key words: Command-Routing and Message-Dispatching Architecture, C++ with MFC (Microsoft Foundation Class) Library on Windows, Subclassing and Superclassing, Message Filter Hook and Computer-Based Training Application Hook, Default Window Procedure and Universal Window Procedure

正　誤　表

北星学園大学　経済学部　北星論集　第45巻　第1号（通巻第48号）

頁・行目	誤	正
19頁 下から 6行目	superclassing	“superclassing”
27頁 上から 4行目	_AfxCbtFilterHook	_AfxCbtFilterHook()
28頁 下から10行目	CTL3D. DLL	CTL3D.DLL
28頁 下から 7行目	SetWindowLong()	SetWindowLong() with GWL_WNDPROC
33頁 下から 6行目	the windows	the window