Development of a Management Support System on the Windows Platform (III-Part 3):

Message Pumping and Message Handling

Hiroshi NOTO

Contents

Section 5 How MFC Uses Message Maps and Handles Messages

5.3. Messages Passed to Window

Up to the previous subsection, we have seen that the MFC class library[(8)–(11)] can subclass each of the MFC controlled windows to install AfxWndProc()[(18)] as the universal window procedure by hooking up the CWnd-derived windows' creation through the _AfxCbtFilterHook()[(18)] callback function. In other words, the computer-based training (CBT) hook of the

Key words: Command-Routing and Message-Dispatching Architecture, C++ with MFC (Microsoft Foundation Class) Library on Windows, WM_COMMAND in Commands and Control Notifications, Message Map Entry with Message, Standard Sequence of CCmdTarget-Derived Classes

CWnd-derived object can create a window with the required class and get the window wired to the generic window procedure AfxWndProc(). From now on all the messages and commands for the window that we are responsible for will go to the AfxWndProc().

In this subsection we take a close look at the processes that the AfxWndProc() goes through to handle the messages and commands that the window receives, by the MFC message mapping architecture, i.e. the command-routing and message-dispatching architecture.

5.3.1 Parameters Passed Over by Windows

We first very briefly summarize the parameters to be passed over by Windows[5],[6] to the functions in concern. A computer-based training (CBT) application, in the present case _AfxCbtFilterHook() has three parameters:

```
_AfxCbtFilterHook (int code, WPARAM wParam, LPARAM lParam)
```

_AfxCbtFilterHook() uses those parameters to receive useful notifications from the system:

int code specifies a code that the hook procedure uses to determine how to process the message such as HCBT_CREATEWND. AfxWndProc() is waiting for HCBT_CREATEWND code that signifies ‘a window is about to be created’.

WPARAM wParam specifies the handle to the new window.

LPARAM lParam specifies a long pointer to a HCBT_CREATEWND structure containing initialization parameters for the window. The parameters include the coordinates and dimensions of the window.

Windows calls back AfxWndProc() with the four parameters in the generic LRESULT/WPARAM/LPARAM format:

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

Two of them are the same that _AfxCbtFilterHook() receives. The handle type identifier **HWND** identifies the handle to a window. Its variable **hWnd** specifies the handle to the window which the message is directed to. The message (**UINT nMsg**) sent to the window needs to be handled by the message map macro and the handler. An MFC-based program deals with two kinds of messages[17]: (1) regular window messages (like WM_MOUSEMOVE, WM_LBUTTONDOWN) and (2) commands (messages generated from menus and controls and represented by WM_COMMAND message). Message maps handle both kinds of messages. Among the window messages, there is a specific message called WM_QUERYAFXWNDPROC which is sent very early in the window creation process. The message determines if the WndProc is AfxWndProc or not. The procedure AfxWndProc returns 1.

5.3.2 Windows Window Handles and MFC CWnd-Derived Objects

MFC represents windows in two ways: (1) by a unique system-defined window handle and (2) by the C++ class representing the window. MFC, on the other hand, provides two areas of functionality: (1) wrapping the regular Windows API functions (like Create() and ShowWindow()) and (2) giving higher-level MFC-related functionality, like default message handling DefWindowProc().

Native Windows code deals with window handles. MFC, on the other hand, is designed to work with, in general, CWnd objects. CWnd, therefore, encapsulates all the Windows API functions that take a window handle: CWnd wraps the API functions maintaining their respective member variables called 'm_hWnd' which represent regular API-level window handles (i.e. HWND). When we call a Windows API function in a CWnd-derived class, the CWnd version of the function uses the standard API function passing the object's window handle (m_hWnd). MFC frequently mixes native handles with MFC wrappers (i.e. CWnd-derived objects). The application framework requires a uniform mapping between window handles and the C++ objects that wrap them (window handles).

It is, therefore, very important in the Windows application development with MFC, to understand the difference between native window handles (HWNDs) and the MFC class objects representing windows in the Windows' message processing that features the handles to the windows and the calls to their member functions or handlers. For example, when Windows calls a window procedure, Windows passes a window handle as the first parameter. MFC's dispatch mechanism, however, works with CWnd-derived objects. In order for the message dispatching to work, MFC has to figure out which CWnd-derived object is associated with a particular handle.

It is easy to get the window handle from a CWnd object because the window handle is a data member of the class. However, there is no way to get from the window handle to the CWnd object without some extra way. MFC uses a class called CHandleMap to relate CWnd-derived objects to window handles[(9)]. The CHandleMap class maps window handles to MFC Windows objects. This means that when a window is created using CWnd (or CWnd-derived class), the window handle is attached to the CWnd object. MFC needs a mechanism like this: **Windows uses handles and MFC uses objects**. The application framework code can deal with C++ objects rather than window handles: when Windows calls a callback function, it passes a window handle as a parameter; MFC needs to translate that parameter into something it can deal with, i.e. CWnd-derived object. The CHandleMap carries two members of type CMapPtrToPtr. They are called m_permanentMap and m_temporaryMap. CHandleMap uses the CMapPtrToPtr capabilities to maintain the relationship between window handles and their associated MFC objects. The permanent map, m_permanentMap, maintains the handle/object map for the life of a program. The temporary map, m_temporaryMap, exists for the duration of a message. The permanent map stores those C++ objects that have been explicitly created by the developer. Whenever a CWnd-derived class is created, MFC inserts the mapping into the permanent dictionary. The mapping is removed whenever CWnd::OnNcDestroy() is called.

5.4 Message Handling Mechanism

Now we get back to the window procedure (or window handler, or window function), namely AfxWndProc() here in VisualC++ 6.0[(7),(12)–(16)] with MFC 4.2 library[(8)–(11)]. It should be noted as just described that there exists a single specific message that AfxWndProc() handles: WM_QUERYAFXWNDPROC (see List 5-6 in (III-2)*)). If the incoming message is WM_QUERYAFXWNDPROC, AfxWndProc() returns value 1. Applications can send the WM_QUERYAFXWNDPROC message to find out if the window is an MFC window using MFC's message-mapping system. If the message is not WM_QUERYAFXWNDPROC, AfxWndProc() goes on to process the message. That means all other massages are routed through the message map.

5.4.1 CHandleMap Global Thread State Object

In AfxWndProc() (List 5-6 in (III-2)), the framework retrieves the C++ object associated with the focused window by using CWnd::FromHandlePermanent() which is shown in List 5-10: the framework calls CWnd::FromHandlePermanent() passing it the focused window handle "hWnd". Then CWnd::FromHandlePermanent() looks up the entry in the permanent handle map and returns the existing MFC object (pWnd) that wraps the passed handle. This function does not create any temporary object.

List 5-10. CWnd::FromHandlePermanent() in WINCORE.CPP

```
CWnd* PASCAL CWnd::FromHandlePermanent(HWND hWnd)
{
        CHandleMap* pMap = afxMapHWND();
        CWnd* pWnd = NULL;
        if (pMap != NULL)
        {
                // only look in the permanent map - does no allocations
                pWnd = (CWnd*)pMap->LookupPermanent(hWnd);
                ASSERT(pWnd == NULL || pWnd->m_hWnd == hWnd);
        }
        return pWnd;
}
```

---

*) Hereafter the series of our articles "Development of a Management Support System on the Windows Platform"[(1)–(4)] will be abbreviated as (I), (II), (III-1) or (III-4).

We look through just more in detail the code in CWnd::FromHandlePermanent(). The afxMapHWND() function gets the global handle map (pMap) of the class CHandleMap that is explained in the previous subsection and returns the pointer to the handle map. In the afxMapHWND() function (see List 5-11) we find that the returned handle map is a member of the AFX_MODULE_THREAD_STATE object that is obtained by a call to AfxGetModuleThreadState().

List 5-11. afxMapHWND() in WINCORE.CPP

```
CHandleMap* PASCAL afxMapHWND(BOOL bCreate)
{
        AFX_MODULE_THREAD_STATE* pState = AfxGetModuleThreadState();
        if (pState->m_pmapHWND == NULL && bCreate)
        {
                BOOL bEnable = AfxEnableMemoryTracking(FALSE);
#ifndef _AFX_PORTABLE
                _PNH pnhOldHandler = AfxSetNewHandler(&AfxCriticalNewHandler);
#endif
                pState->m_pmapHWND = new CHandleMap(RUNTIME_CLASS(CTempWnd),
                        offsetof(CWnd, m_hWnd));

#ifndef _AFX_PORTABLE
                AfxSetNewHandler(pnhOldHandler);
#endif
                AfxEnableMemoryTracking(bEnable);
        }
        return pState->m_pmapHWND;
}
```

AFX_MODULE_THREAD_STATE is shown in List 5-12 below and is basically a class keeping information about the current thread state. MFC keeps a global object of this type on per thread basis (pState in the present case).

List 5-12. AFX_MODULE_THREAD_STATE in AFXSTAT_.H

```
// AFX_MODULE_THREAD_STATE (local to thread *and* module)
class AFX_MODULE_THREAD_STATE : public CNoTrackObject
{
public:
        AFX_MODULE_THREAD_STATE();
        virtual ~AFX_MODULE_THREAD_STATE();

        // current CWinThread pointer
        CWinThread* m_pCurrentWinThread;

        // list of CFrameWnd objects for thread
        CTypedSimpleList<CFrameWnd*> m_frameList;
```

```
        // temporary/permanent map state
        DWORD m_nTempMapLock;           // if not 0, temp maps locked
        CHandleMap* m_pmapHWND;
        CHandleMap* m_pmapHMENU;
        CHandleMap* m_pmapHDC;
        CHandleMap* m_pmapHGDIOBJ;
        CHandleMap* m_pmapHIMAGELIST;

        // thread-local MFC new handler (separate from C-runtime)
        _PNH m_pfnNewHandler;

#ifndef _AFX_NO_SOCKET_SUPPORT
        // WinSock specific thread state
        HWND m_hSocketWindow;
#ifdef _AFXDLL
        CEmbeddedButActsLikePtr<CMapPtrToPtr> m_pmapSocketHandle;
        CEmbeddedButActsLikePtr<CMapPtrToPtr> m_pmapDeadSockets;
        CEmbeddedButActsLikePtr<CPtrList> m_plistSocketNotifications;
#else
        CMapPtrToPtr* m_pmapSocketHandle;
        CMapPtrToPtr* m_pmapDeadSockets;
        CPtrList* m_plistSocketNotifications;
#endif
#endif
};
```

In List 5-12 we can see all handle maps of the concerned Windows objects like Window, Menu, DC, GdiObject and ImageList. In the present case AFX_MODULE_THREAD_STATE returns the corresponding member variable of the global thread state object m_pmapHWND which afxMapHWND() returns as CHandleMap* pMap pointer (i.e. pState->m_pmapHWND) (List 5-10). List 5-13 shows AfxGetModuleThreadState().

List 5-13. AfxGetModuleThreadState() in AFXSTATE.CPP

```
AFX_MODULE_THREAD_STATE* AFXAPI AfxGetModuleThreadState()
{
        return AfxGetModuleState()->m_thread.GetData();
}
```

In AfxGetModuleThreadState() we find that the object of AFX_MODULE_THREAD_STATE is brought about by AfxGetModuleState() through the following code which is a bit complicated.

```
AfxGetModuleState()->m_thread.GetData();        (*)
```

AfxGetModuleState() is a member function of AFX_MODULE_STATE class, as defined in List 5-14. The definition of the class AFX_MODULE_STATE is shown in List 2-2 in (II), which in turn keeps track of the current module state.

List 5-14. AfxGetModuleState() in AFXSTATE.CPP

```
AFX_MODULE_STATE* AFXAPI AfxGetModuleState()
{
        _AFX_THREAD_STATE* pState = _afxThreadState;
        AFX_MODULE_STATE* pResult;
        if (pState->m_pModuleState != NULL)
        {
                // thread state's module state serves as override
                pResult = pState->m_pModuleState;
        }
        else
        {
                // otherwise, use global app state
                pResult = _afxBaseModuleState.GetData();
        }
        ASSERT(pResult != NULL);
        return pResult;
}
```

The AfxGetModuleState() function defines pState which copies the thread state object _afxThreadState that instantiates the _AFX_THREAD_STATE class. The instance of _afxThreadState is realized through the macro THREAD_LOCAL which is found in AFX_STATE.CPP (List 5-15).

List 5-15. Thread local portions of the thread state in AFXSTATE.CPP

```
THREAD_LOCAL(_AFX_THREAD_STATE, _afxThreadState)
```

Here _afxThreadState is an instance of the CThreadLocal object (see below). The _AFX_THREAD_STATE class is elaborated later below in List 5-21. The object “m_thread” in (＊) above is an instance of the CThreadLocal class and the function of GetData() is a member function of the CThreadLocalObject class. Why and how does the above code make sense? The reason is the following. The class AFX_MODULE_STATE in List 2-2 in (II) reads at the very bottom of its definition like this (List 5-16):

List 5-16. Class AFX_MODULE_STATE (mostly omitted except for THREAD_LOCAL()) in AFXSTAT_.H

```
// AFX_MODULE_STATE (global data for a module)
class AFX_MODULE_STATE : public CNoTrackObject
{
```

```
public:
<<omitted>>
        // define thread local portions of module state
        THREAD_LOCAL(AFX_MODULE_THREAD_STATE, m_thread)
};
```

And the macro THREAD_LOCAL is defined in AFXTLS_.H (List 5-17).　MFC gives us some classes to store information private for each thread with the THREAD_LOCAL macro.

List 5-17. Macro THREAD_LOCAL() in AFXTLS_.H

```
#define THREAD_LOCAL(class_name, ident_name)
        AFX_DATADEF CThreadLocal<class_name> ident_name;
```

Finally the class CThreadLocal is defined like this (List 5-18):

List 5-18. Class CThreadLocal in AFXTLS_.H

```
template<class TYPE>
class CThreadLocal : public CThreadLocalObject
{
// Attributes
public:
        AFX_INLINE TYPE* GetData()
        {
                TYPE* pData = (TYPE*)CThreadLocalObject::GetData(&CreateObject);
                ASSERT(pData != NULL);
                return pData;
        }
        AFX_INLINE TYPE* GetDataNA()
        {
                TYPE* pData = (TYPE*)CThreadLocalObject::GetDataNA();
                return pData;
        }
        AFX_INLINE operator TYPE*()
                { return GetData(); }
        AFX_INLINE TYPE* operator->()
                { return GetData(); }

// Implementation
public:
        static CNoTrackObject* AFXAPI CreateObject()
                { return new TYPE; }
};
```

The AfxGetModuleThreadState() function in List 5-11, gets the pointer pResult to pState->

m_pModuleState that the AfxGetModuleState() returns, the latter being the member variable of _AFX_THREAD_STATE.

Thus in the class AFX_MODULE_STATE, the CThreadLocal object “m_thread” is defined. The CThreadLocal is a template class and the data TYPE is substituted for “AFX_MODULE_THREAD_STATE”. In CThreadLocal, AFX_INLINE TYPE* GetData() function is defined as its member function where CThreadLocalObject::GetData() is called and the returned value is casted from “CNoTrackObject” to the type “AFX_MODULE_THREAD_STATE” as a pointer (“pData”). In this way the code AfxGetModuleState()->m_thread.GetData() in (*) at page 38 and in List 5-13 returns the current thread local instance of AFX_MODULE_THREAD_STATE type as “pData” and afxMapHWND() returns the global handle map of the current thread local instance as “pState->m_pmapHWND” that is local to thread in List 5-11.

5.4.2 _AFX_THREAD_STATE Object with Message

Now we get back to CWnd::FromHandlePermanent() in List 5-10. The framework calls LookupPermanent() function. CHandleMap::LookupPermanent() is expanded inline like List 5-19.

List 5-19. CHandleMap::LookupPermanent in WINHAND_.H

```
inline CObject* CHandleMap::LookupPermanent(HANDLE h)
        { return (CObject*)m_permanentMap.GetValueAt((LPVOID)h); }
```

In the permanent handle map does the function m_permanentMap.GetValueAt() look up the entry of our present window handle (HWND hWnd) which is now passed to its argument HANDLE h that is then casted to LPVOID. Here the data type HANDLE represents 32-bit unsigned integer handle to an object and LPVOID represents the generic pointer type. Finally the looked-up object of the present window ends up in the object casted from CObject to CWnd.

The framework returns AfxCallWndProc() in AfxWndProc() (in List 5-6 in (III-2)). It should be noted that in addition to the first parameter (pWnd) as the pointer to a CWnd object, AfxCallWndProc() also has the second parameter (hWnd) as the window handle that is assigned to the CWnd object. This allows AfxCallWndProc() to maintain the record of the last message processed for use in handling exceptions and debugging, since it is that window that the message is sent to. In List 5-20 shown is AfxCallWndProc(). We notice how it looks like any other window procedure, except that the parameter includes a CWnd pointer as well.

List 5-20. AfxCallWndProc() in WINCORE.CPP

```
/////////////////////////////////////////////////////////////////////////////
// Official way to send message to a CWnd

LRESULT AFXAPI AfxCallWndProc(CWnd* pWnd, HWND hWnd, UINT nMsg,
	WPARAM wParam = 0, LPARAM lParam = 0)
{
	_AFX_THREAD_STATE* pThreadState = _afxThreadState.GetData();
	MSG oldState = pThreadState->m_lastSentMsg;   // save for nesting
	pThreadState->m_lastSentMsg.hwnd = hWnd;
	pThreadState->m_lastSentMsg.message = nMsg;
	pThreadState->m_lastSentMsg.wParam = wParam;
	pThreadState->m_lastSentMsg.lParam = lParam;

#ifdef _DEBUG
	if (afxTraceFlags & traceWinMsg)
		_AfxTraceMsg(_T("WndProc"), &pThreadState->m_lastSentMsg);
#endif

	// Catch exceptions thrown outside the scope of a callback
	// in debug builds and warn the user.
	LRESULT lResult;
	TRY
	{
#ifndef _AFX_NO_OCC_SUPPORT
		// special case for WM_DESTROY
		if ((nMsg == WM_DESTROY) && (pWnd->m_pCtrlCont != NULL))
			pWnd->m_pCtrlCont->OnUIActivate(NULL);
#endif

		// special case for WM_INITDIALOG
		CRect rectOld;
		DWORD dwStyle = 0;
		if (nMsg == WM_INITDIALOG)
			_AfxPreInitDialog(pWnd, &rectOld, &dwStyle);

		// delegate to object's WindowProc
		lResult = pWnd->WindowProc(nMsg, wParam, lParam);

		// more special case for WM_INITDIALOG
		if (nMsg == WM_INITDIALOG)
			_AfxPostInitDialog(pWnd, rectOld, dwStyle);
	}
	CATCH_ALL(e)
	{
		lResult = AfxGetThread()->ProcessWndProcException
		(e, &pThreadState->m_lastSentMsg);
		TRACE1("Warning: Uncaught exception in WindowProc
			(returning %ld).¥n", lResult);
		DELETE_EXCEPTION(e);
	}
	END_CATCH_ALL

	pThreadState->m_lastSentMsg = oldState;
	return lResult;
}
```

As shown above AfxCallWndProc() first examines the message to see if it is a WM_INITDIALOG, in which case it calls _AfxPreInitDialog(). This function is for the auto-center dialog feature: MFC caches certain styles before the dialog handles

WM_INITDIALOG. If it is appropriate to center the window (the window is still not visible and has not moved), then MFC automatically centers the dialog against its parent. The following sentence seen in AfxCallWndProc() in List 5-20 means that pThreadState is an object of the pointer type to _AFX_THREAD_STATE, which is shown in List 5-21 and is implemented through the definition of thread local portions of the thread state in List 5-15.

```
_AFX_THREAD_STATE* pThreadState = _afxThreadState.GetData();
```

The identifier _afxThreadState is an instance of the CThreadLocal object. The function AfxCallWndProc() saves the window handle (hwnd), the message (message), and the WPARM (wParam) and the LPARM (lParam) in the current thread state member variable, pThread->m_lastSentMsg. The message data structure (MSG) form is represented in Figure 4-1 in (III-1).

List 5-21. class _AFX_THREAD_STATE in AFXSTAT_.H

```
class _AFX_THREAD_STATE : public CNoTrackObject
{
public:
        _AFX_THREAD_STATE();
        virtual ~_AFX_THREAD_STATE();

        // override for m_pModuleState in _AFX_APP_STATE
        AFX_MODULE_STATE* m_pModuleState;
        AFX_MODULE_STATE* m_pPrevModuleState;

        // memory safety pool for temp maps
        void* m_pSafetyPoolBuffer;    // current buffer

        // thread local exception context
        AFX_EXCEPTION_CONTEXT m_exceptionContext;

        // CWnd create, gray dialog hook, and other hook data
        CWnd* m_pWndInit;
        CWnd* m_pAlternateWndInit;      // special case commdlg hooking
        DWORD m_dwPropStyle;
        DWORD m_dwPropExStyle;
        HWND m_hWndInit;
        BOOL m_bDlgCreate;
        HHOOK m_hHookOldCbtFilter;
        HHOOK m_hHookOldMsgFilter;

        // other CWnd modal data
        MSG m_lastSentMsg;              // see CWnd::WindowProc
        HWND m_hTrackingWindow;         // see CWnd::TrackPopupMenu
        HMENU m_hTrackingMenu;
        TCHAR m_szTempClassName[96];    // see AfxRegisterWndClass
        HWND m_hLockoutNotifyWindow;    // see CWnd::OnCommand
        BOOL m_bInMsgFilter;
```

```
        // other framework modal data
        CView* m_pRoutingView;          // see CCmdTarget::GetRoutingView
        CFrameWnd* m_pRoutingFrame;     // see CCmdTarget::GetRoutingFrame

        // MFC/DB thread-local data
        BOOL m_bWaitForDataSource;

        // common controls thread state
        CToolTipCtrl* m_pToolTip;
        CWnd* m_pLastHit;       // last window to own tooltip
        int m_nLastHit;         // last hittest code
        TOOLINFO m_lastInfo;    // last TOOLINFO structure
        int m_nLastStatus;      // last flyby status message
        CControlBar* m_pLastStatus; // last flyby status control bar

        // OLE control thread-local data
        CWnd* m_pWndPark;       // "parking space" window
        long m_nCtrlRef;        // reference count on parking window
        BOOL m_bNeedTerm;       // TRUE if OleUninitialize needs to be called
};
```

5.4.3 Window Object's Window Procedure WindowProc()

The function AfxCallWndProc() returns the window object's window procedure as lResult: pWnd->WindowProc(nMsg, wParam, lParam). Here shown is CWnd::Windowproc() in List 5-22.

List 5-22. CWnd::WindowProc() in WINCORE.CPP

```
// main WindowProc implementation

LRESULT CWnd::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
        // OnWndMsg does most of the work, except for DefWindowProc call
        LRESULT lResult = 0;
        if (!OnWndMsg(message, wParam, lParam, &lResult))
                lResult = DefWindowProc(message, wParam, lParam);
        return lResult;
}
```

CWnd::WindowProc() is virtual and overridable. CWnd::WindowProc() calls CWnd::OnWndMsg(). CWnd::OnWndMsg() is also virtual and overridable. CWnd::OnWndMsg() indicates whether or not a windows message was handled. It returns nonzero value if the message was handled; otherwise it returns 0. If OnWndMsg() returns FALSE (i.e. 0), then

CWnd::WindowProc() calls CWnd::DefWindowproc() that handles the messages irrelevant to our application. CWnd::DefWindowproc() is virtual and overridable as well. Thus our study of Message Handling procedure now goes into the CWnd::OnWndMsg() function.

5.4.4 Message-Handling inside CWnd::OnWndMsg()

Now Let us see in detail how MFC uses Messages and Message Maps. As explained in 4.2 Three Message Categories in (III-1) and 5.2.1 Command-Routing and Message-Dispatching in (III-2), an MFC-based program deals with two kinds of messages: (1) regular window messages (like WM_MOUSEMOVE, WM_LBUTTONDOWN) and (2) commands (that is, the messages generated from menus and controls and represented by WM_COMMAND message). Message maps handle both kinds of messages. The message-handling action really begins inside CWnd::OnWndMsg(). List 5-23 shows (some pared-down) source code in WINCORE.CPP. Let us briefly walk through OnWndMsg() before tracing messages through it. First, OnWndMsg() tries to filter out certain messages from the beginning: WM_COMMAND, WM_NOTIFY, WM_ACTIVATE, and WM_SETCURSOR. The framework has special ways of handling each of these messages. If the message is not one of those just listed, OnWndMsg() tries to look up the message in the message map. MFC keeps a message map entry cache that is accessible via a hash value. This is a great optimization because looking up a value in a hash table is much cheaper than walking the message map. CWnd::OnWndMsg() is where commands and regular window messages go their separate ways. If the message is a command message (that is, message == WM_COMMAND), then CWnd::OnWndMsg() calls OnCommand() (i.e. CWnd::OnCommand()). Otherwise, it retrieves the window object's message map to process the message (more on that in (Ⅲ-4)). Let us examine the command routing first.

List 5-23. The CWnd::OnWndMsg() (pared-down) in WINCORE.CPP

```
BOOL  CWnd::OnWndMsg(UINT  message,  WPARAM  wParam,  LPARAM  lParam,
                              LRESULT* pResult)
{
          LRESULT lResult = 0;

          // special case for commands
          if (message == WM_COMMAND)
          {
                    if (OnCommand(wParam, lParam))
                    {
                              lResult = 1;
                              goto LReturnTrue;
                    }
                    return FALSE;
          }

          // special case for notifies
          if (message == WM_NOTIFY)
          {
```

```
                NMHDR* pNMHDR = (NMHDR*)lParam;
        if (pNMHDR->hwndFrom != NULL && OnNotify(wParam, lParam, &lResult))
                        goto LReturnTrue;
                return FALSE;
        }

        // special case for activation
        if (message == WM_ACTIVATE)
            _AfxHandleActivate(this, wParam, CWnd::FromHandle((HWND)lParam));

        // special case for set cursor HTERROR
        if (message == WM_SETCURSOR &&
        _AfxHandleSetCursor(this, (short)LOWORD(lParam), HIWORD(lParam)))
        {
                lResult = 1;
                goto LReturnTrue;
        }

          const AFX_MSGMAP* pMessageMap; pMessageMap = GetMessageMap();
              UINT iHash;
          iHash = (LOWORD((DWORD)pMessageMap) ^ message) &   (iHashMax-1);
              AfxLockGlobals(CRIT_WINMSGCACHE);
              AFX_MSG_CACHE* pMsgCache; pMsgCache = &_afxMsgCache[iHash];
              const AFX_MSGMAP_ENTRY* lpEntry;
 if (message == pMsgCache->nMsg && pMessageMap == pMsgCache->pMessageMap)
        {
                // cache hit
                lpEntry = pMsgCache->lpEntry;
                AfxUnlockGlobals(CRIT_WINMSGCACHE);
                if (lpEntry == NULL)
                        return FALSE;

                // cache hit, and it needs to be handled
                if (message < 0xC000)
                        goto LDispatch;
                else
                        goto LDispatchRegistered;
        }
        else
        {
                // not in cache, look for it
                pMsgCache->nMsg = message;
                pMsgCache->pMessageMap = pMessageMap;

#ifdef _AFXDLL
                for (/* pMessageMap already init'ed */; pMessageMap != NULL;
                        pMessageMap = (*pMessageMap->pfnGetBaseMap)())
#else
                for (/* pMessageMap already init'ed */; pMessageMap != NULL;
                        pMessageMap = pMessageMap->pBaseMap)
#endif
                {
                        // Note: catch not so common but fatal mistake!!
                        //      BEGIN_MESSAGE_MAP(CMyWnd, CMyWnd)
#ifdef _AFXDLL
                ASSERT(pMessageMap != (*pMessageMap->pfnGetBaseMap)());
#else
                        ASSERT(pMessageMap != pMessageMap->pBaseMap);
#endif
```

```
                        if (message < 0xC000)
                        {
                                // constant window message
if((lpEntry = AfxFindMessageEntry(pMessageMap->lpEntries,message, 0, 0)) != NULL)
                                {
                                        pMsgCache->lpEntry = lpEntry;
                                        AfxUnlockGlobals(CRIT_WINMSGCACHE);
                                        goto LDispatch;
                                }
                        }
                        else
                        {
                                // registered windows message
                                lpEntry = pMessageMap->lpEntries;
                while((lpEntry = AfxFindMessageEntry(lpEntry, 0xC000, 0, 0)) != NULL)
                                {
                                        UINT* pnID = (UINT*)(lpEntry->nSig);
                                        ASSERT(*pnID >= 0xC000 || *pnID == 0);
                                                // must be successfully registered
                                        if (*pnID == message)
                                        {
                                                pMsgCache->lpEntry = lpEntry;
                                        AfxUnlockGlobals(CRIT_WINMSGCACHE);
                                                goto LDispatchRegistered;
                                        }
                                        lpEntry++;    // keep looking past this one
                                }
                        }
                }

                pMsgCache->lpEntry = NULL;
                AfxUnlockGlobals(CRIT_WINMSGCACHE);
                return FALSE;
        }
        ASSERT(FALSE);          // not reached

LDispatch:
        ASSERT(message < 0xC000);
        union MessageMapFunctions mmf;
        mmf.pfn = lpEntry->pfn;

        // if we've got WM_SETTINGCHANGE / WM_WININICHANGE, we need to
        // decide if we're going to call OnWinIniChange() or OnSettingChange()

        int nSig;
        nSig = lpEntry->nSig;
        if (lpEntry->nID == WM_SETTINGCHANGE)
        {
                DWORD dwVersion = GetVersion();
                if (LOBYTE(LOWORD(dwVersion)) >= 4)
                        nSig = AfxSig_vws;
                else
                        nSig = AfxSig_vs;
        }

        switch (nSig)
        {
        default:
                ASSERT(FALSE);
                break;
```

```
        case AfxSig_bD:
                lResult = (this->*mmf.pfn_bD)(CDC::FromHandle((HDC)wParam));
                break;

        case AfxSig_bb:         // AfxSig_bb, AfxSig_bw, AfxSig_bh
                lResult = (this->*mmf.pfn_bb)((BOOL)wParam);
                break;

        case AfxSig_bWww:     // really AfxSig_bWiw
         lResult = (this->*mmf.pfn_bWww)(CWnd::FromHandle((HWND)wParam),
                                         (short)LOWORD(lParam), HIWORD(lParam));
                break;

        case AfxSig_bWCDS:
            lResult = (this->*mmf.pfn_bWCDS)(CWnd::FromHandle((HWND)wParam),
                                             (COPYDATASTRUCT*)lParam);
                break;

        case AfxSig_bHELPINFO:
                lResult = (this->*mmf.pfn_bHELPINFO)((HELPINFO*)lParam);
                break;

        case AfxSig_hDWw:
                {
                // special case for OnCtlColor to avoid too many temporary objects
                        ASSERT(message == WM_CTLCOLOR);
                        AFX_CTLCOLOR* pCtl = (AFX_CTLCOLOR*)lParam;
                        CDC dcTemp; dcTemp.m_hDC = pCtl->hDC;
                        CWnd wndTemp; wndTemp.m_hWnd = pCtl->hWnd;
                        UINT nCtlType = pCtl->nCtlType;
                // if not coming from a permanent window, use stack temporary
                CWnd* pWnd = CWnd::FromHandlePermanent(wndTemp.m_hWnd);
                        if (pWnd == NULL)
                        {
#ifndef _AFX_NO_OCC_SUPPORT
                                // determine the site of the OLE control if it is one
                                COleControlSite* pSite;
                        if (m_pCtrlCont != NULL && (pSite = (COleControlSite*)
                m_pCtrlCont->m_siteMap.GetValueAt(wndTemp.m_hWnd)) != NULL)
                                {
                                        wndTemp.m_pCtrlSite = pSite;
                                }
#endif
                                pWnd = &wndTemp;
                        }
                HBRUSH hbr = (this->*mmf.pfn_hDWw)(&dcTemp, pWnd, nCtlType);
                        // fast detach of temporary objects
                        dcTemp.m_hDC = NULL;
                        wndTemp.m_hWnd = NULL;
                        lResult = (LRESULT)hbr;
                }
                break;

        case AfxSig_hDw:
                {
                    // special case for CtlColor to avoid too many temporary objects
                    ASSERT(message == WM_REFLECT_BASE+WM_CTLCOLOR);
                        AFX_CTLCOLOR* pCtl = (AFX_CTLCOLOR*)lParam;
                        CDC dcTemp; dcTemp.m_hDC = pCtl->hDC;
                        UINT nCtlType = pCtl->nCtlType;
                        HBRUSH hbr = (this->*mmf.pfn_hDw)(&dcTemp, nCtlType);
```

```
                        // fast detach of temporary objects
                        dcTemp.m_hDC = NULL;
                        lResult = (LRESULT)hbr;
                }
                break;

<<omitted>>

        case AfxSig_bwsp:
        lResult = (this->*mmf.pfn_bwsp)(LOWORD(wParam),
          (short) HIWORD(wParam), CPoint(LOWORD(lParam), HIWORD(lParam)));
                if (!lResult)
                        return FALSE;
        }
        goto LReturnTrue;

LDispatchRegistered:    // for registered windows messages
        ASSERT(message >= 0xC000);
        mmf.pfn = lpEntry->pfn;
        lResult = (this->*mmf.pfn_lwl)(wParam, lParam);

LReturnTrue:
        if (pResult != NULL)
                *pResult = lResult;
        return TRUE;
}
```

5.5 WM_COMMAND in Commands and Control Notifications

First we follow a WM_COMMAND message through the application framework to see where it is handled[(9),(18)].

5.5.1 Handling WM_COMMAND

We take command messages. Windows messages are usually sent to the main frame window, but command messages are then further routed to other objects. As is explained below the framework routes commands through a standard sequence of command-target objects, one of which is expected to have a handler for the command. Each command-target object checks its message map to see if it can handle the incoming message. The first stop a command makes on its way to its designated command target is CWnd::OnCommand().

5.5.2 OnCommand()

Since CWnd::OnCommand() is a virtual function, the framework calls the correct version. Suppose the message was generated for the main frame window, the framework calls the CFrameWnd version of OnCommand() (List 5-24).

List 5-24. CFrameWnd::OnCommand() in WINFRM.CPP

```
BOOL CFrameWnd::OnCommand(WPARAM wParam, LPARAM lParam)
        // return TRUE if command invocation was attempted
```

```
{
        HWND hWndCtrl = (HWND)lParam;
        UINT nID = LOWORD(wParam);

        CFrameWnd* pFrameWnd = GetTopLevelFrame();
        ASSERT_VALID(pFrameWnd);
        if (pFrameWnd->m_bHelpMode && hWndCtrl == NULL &&
                nID != ID_HELP && nID != ID_DEFAULT_HELP && nID != ID_CONTEXT_HELP)
        {
                // route as help
                if (!SendMessage(WM_COMMANDHELP, 0, HID_BASE_COMMAND+nID))
                        SendMessage(WM_COMMAND, ID_DEFAULT_HELP);
                return TRUE;
        }

        // route as normal command
        return CWnd::OnCommand(wParam, lParam);
}
```

By this point, the message is pared down to two parameters: WPARAM and LPARAM in the arguments of the function. If the message is a request for on-line help, the framework sends a WM_COMMANDHELP message to the frame window. Otherwise, the message is passed on to the base class's OnCommand(), CWnd::OnCommand().

List 5-25. CWnd::OnCommand() in WINCORE.CPP

```
BOOL CWnd::OnCommand(WPARAM wParam, LPARAM lParam)
        // return TRUE if command invocation was attempted
{
        UINT nID = LOWORD(wParam);
        HWND hWndCtrl = (HWND)lParam;
        int nCode = HIWORD(wParam);

        // default routing for command messages (through closure table)

        if (hWndCtrl == NULL)
        {
                // zero IDs for normal commands are not allowed
                if (nID == 0)
                        return FALSE;

                // make sure command has not become disabled before routing
                CTestCmdUI state;
                state.m_nID = nID;
                OnCmdMsg(nID, CN_UPDATE_COMMAND_UI, &state, NULL);
                if (!state.m_bEnabled)
                {
                        TRACE1("Warning: not executing disabled command %d¥n",
                        nID);
                        return TRUE;
                }

                // menu or accelerator
                nCode = CN_COMMAND;
        }
        else
        {
                // control notification
                ASSERT(nID == 0 || ::IsWindow(hWndCtrl));

                if (_afxThreadState->m_hLockoutNotifyWindow == m_hWnd)
                        return TRUE;            // locked out - ignore control notification
```

```
                    // reflect notification to child window control
                    if (ReflectLastMsg(hWndCtrl))
                            return TRUE;      // eaten by child

                    // zero IDs for normal commands are not allowed
                    if (nID == 0)
                            return FALSE;
            }

#ifdef _DEBUG
            if (nCode < 0 && nCode != (int)0x8000)
                    TRACE1("Implementation Warning: control notification = $%X.¥n",
                            nCode);
#endif

            return OnCmdMsg(nID, nCode, NULL, NULL);
}
```

CWnd::OnCommand() examines the LPARAM which represents the control that sends the message if the message is from a control. If the command was generated by a control the LPARAM contains the handle of the control window. If the message is a control notification (like EN_CHANGE or LBN_CHANGESEL), then the framework performs some special processing. If a notification message is from the child window message, OnCommand() sends the last message to the child window (i.e. ReflectLastMsg (hWndCtrl)). OnCommand(), then, returns.

Otherwise (i.e. hWndCtrl equal to NULL), CWnd::OnCommand() makes sure that the user-interface element that generated the command has not become disabled (for instance, a menu item is not undefined) and passes the message on to OnCmdMsg() (which is also virtual). Because the frame window is still trying to handle the message, CFrameWnd::OnCmdMsg() is the version that is called. This function is found in WINFRM.CPP (List 5-26):

List 5-26. CFrameWnd::OnCmdMsg() in WINFRM.CPP

```
/////////////////////////////////////////////////////////////////////////////
// CFrameWnd command/message routing

BOOL CFrameWnd::OnCmdMsg(UINT nID, int nCode, void* pExtra,
        AFX_CMDHANDLERINFO* pHandlerInfo)
{
        CPushRoutingFrame push(this);

        // pump through current view FIRST
        CView* pView = GetActiveView();
        if (pView != NULL && pView->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;

        // then pump through frame
        if (CWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;

        // last but not least, pump through app
        CWinApp* pApp = AfxGetApp();
        if (pApp != NULL && pApp->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;
```

```
        return FALSE;
}
```

CWnd::OnCommand() passes NULL for pExtra and pHandlerInfo when it calls CFrameWnd::OnCmdMsg(), because this information is not needed for handling commands (see the last two arguments in CFrameWnd::OnCmdMsg() in List 5-26). List 5-26 tells us that CFrameWnd::OnCmdMsg() pumps the message through the application components in this order: the active view → the active view's document → the main frame window → the application.

To route the command to the active view, CFrameWnd::OnCmdMsg() tries to find the frame's active view using CWnd::GetActiveView(). If CFrameWnd::OnCmdMsg() succeeds in finding the frame's active window, it calls the active view's OnCmdMsg() (pView->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo)). If the active view's OnCmdMsg() cannot deal with the command, the document takes a crack at the command (see CView::OnCmdMsg() in List 5-27 below). If CFrameWnd::OnCmdMsg() fails to find an active view, or the view and the document fail to handle the message, the frame window gets a chance to handle the message. Finally, if the frame window does not want the message, then the application attempts to process the message—CFrameWnd::OnCmdMsg() calls the application's OnCmdMsg() function (pApp->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo)).

Suppose the message has reached the active view in List 5-26, the function CView::OnCmdMsg() is invoked in VIEWCORE.CPP (List 5-27):

List 5-27. CView::OnCmdMsg() in VIEWCORE.CPP

```
/////////////////////////////////////////////////////////////////////////////
// Command routing

BOOL CView::OnCmdMsg(UINT nID, int nCode, void* pExtra,
        AFX_CMDHANDLERINFO* pHandlerInfo)
{
        // first pump through pane
        if (CWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;

        // then pump through document
        if (m_pDocument != NULL)
        {
                // special state for saving view before routing to document
                CPushRoutingView push(this);
                return m_pDocument->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);
        }

        return FALSE;
}
```

The framework gives the window pane*) part of the view a chance to respond to the message

*) When a window is split (or divided) into several pieces, each piece is called a "pane."

by calling CWnd::OnCmdMsg(). If the view pane cannot handle the message, the message is, according to the code in List 5-27, pumped through the document.

Because CWnd does not override OnCmdMsg(), the command goes straight to CCmdTarget::OnCmdMsg(), which is found in CMDTARG.CPP (List 5-28). In other words, CWnd::OnCmdMsg() inherits CCmdTarget::OnCmdMsg(). This is a very important point and indicates the inheritance of the class hierarchy structure.

List 5-28. CCmdTarget::OnCmdMsg() in CMDTARG.CPP

```
BOOL CCmdTarget::OnCmdMsg(UINT nID, int nCode, void* pExtra,
        AFX_CMDHANDLERINFO* pHandlerInfo);
{
#ifndef _AFX_NO_OCC_SUPPORT
        // OLE control events are a special case
        if (nCode == CN_EVENT)
        {
                ASSERT(afxOccManager != NULL);
                return afxOccManager->OnEvent(this, nID, (AFX_EVENT*)pExtra, pHandlerInfo);
        }
#endif // !_AFX_NO_OCC_SUPPORT

        // determine the message number and code (packed into nCode)
        const AFX_MSGMAP* pMessageMap;
        const AFX_MSGMAP_ENTRY* lpEntry;
        UINT nMsg = 0;

#ifndef _AFX_NO_DOCOBJECT_SUPPORT
        if (nCode == CN_OLECOMMAND)
        {
                BOOL bResult = FALSE;

                const AFX_OLECMDMAP* pOleCommandMap;
                const AFX_OLECMDMAP_ENTRY* pEntry;

                COleCmdUI* pUI = (COleCmdUI*) pExtra;
                const GUID* pguidCmdGroup = pUI->m_pguidCmdGroup;

#ifdef _AFXDLL
                for (pOleCommandMap = GetCommandMap(); pOleCommandMap != NULL && !bResult;
                        pOleCommandMap = pOleCommandMap->pfnGetBaseMap())
#else
                for (pOleCommandMap = GetCommandMap(); pOleCommandMap != NULL && !bResult;
                        pOleCommandMap = pOleCommandMap->pBaseMap)
#endif
                {
                        for (pEntry = pOleCommandMap->lpEntries;
                                pEntry->cmdID != 0 && pEntry->nID != 0 && !bResult;
                                pEntry++)
                        {
                                if (nID == pEntry->cmdID &&
                                        IsEqualNULLGuid(pguidCmdGroup, pEntry->pguid))
                                {
                                        pUI->m_nID = pEntry->nID;
                                        bResult = TRUE;
                                }
                        }
                }

                return bResult;
        }
#endif

        if (nCode != CN_UPDATE_COMMAND_UI)
        {
                nMsg = HIWORD(nCode);
                nCode = LOWORD(nCode);
        }

        // for backward compatibility HIWORD(nCode)==0 is WM_COMMAND
        if (nMsg == 0)
                nMsg = WM_COMMAND;
```

```
        // look through message map to see if it applies to us
#ifdef _AFXDLL
        for (pMessageMap = GetMessageMap(); pMessageMap != NULL;
          pMessageMap = (*pMessageMap->pfnGetBaseMap)())
#else
        for (pMessageMap = GetMessageMap(); pMessageMap != NULL;
          pMessageMap = pMessageMap->pBaseMap)
#endif
        {
                // Note: catches BEGIN_MESSAGE_MAP(CMyClass, CMyClass)!
#ifdef _AFXDLL
                ASSERT(pMessageMap != (*pMessageMap->pfnGetBaseMap)());
#else
                ASSERT(pMessageMap != pMessageMap->pBaseMap);
#endif

                lpEntry = AfxFindMessageEntry(pMessageMap->lpEntries, nMsg, nCode, nID);
                if (lpEntry != NULL)
                {
                        // found it
#ifdef _DEBUG
                        if (afxTraceFlags & traceCmdRouting)
                        {
                                if (nCode == CN_COMMAND)
                                {
                                        TRACE2("SENDING command id 0x%04X to %hs target.¥n", nID,
                                                GetRuntimeClass()->m_lpszClassName);
                                }
                                else if (nCode > CN_COMMAND)
                                {
                                        if (afxTraceFlags & traceWinMsg)
                                        {
                                                TRACE3("SENDING control notification %d from
control id 0x%04X to %hs window.¥n",
                                                        nCode,                    nID,
GetRuntimeClass()->m_lpszClassName);
                                        }
                                }
                        }
#endif //_DEBUG
                        return _AfxDispatchCmdMsg(this, nID, nCode,
                                lpEntry->pfn, pExtra, lpEntry->nSig, pHandlerInfo);
                }
        }
        return FALSE;   // not handled
}
```

CCmdTarget::OnCmdMsg() walks the message map trying to find a handler for the message. If necessary CCmdTarget::OnCmdMsg() gets back to the base class. If it finds one, it calls that function. If it cannot, CCmdTarget::OnCmdMsg() returns FALSE, and the document gets a chance to handle the message. If the document does not want anything to do with the message, then the message is handled by the CWnd's DefWindProc() (see List 5-22 CWnd::WindowProc()).

5.5.3 Entry for Message in Message Map

CCmdTarget::OnCmdMsg() searches the message in the message map by calling AfxFindMessageEntry() that is shown in List 5-29. If the function finds the entry for the message it returns lpEntry.

List 5-29. AfxFindMessageEntry() in WINCORE.CPP

```
// Routines for fast search of message maps

const AFX_MSGMAP_ENTRY* AFXAPI
AfxFindMessageEntry(const AFX_MSGMAP_ENTRY* lpEntry,
```

```
        UINT nMsg, UINT nCode, UINT nID)
{
#if defined(_M_IX86) && !defined(_AFX_PORTABLE)
// 32-bit Intel 386/486 version.

        ASSERT(offsetof(AFX_MSGMAP_ENTRY, nMessage) == 0);
        ASSERT(offsetof(AFX_MSGMAP_ENTRY, nCode) == 4);
        ASSERT(offsetof(AFX_MSGMAP_ENTRY, nID) == 8);
        ASSERT(offsetof(AFX_MSGMAP_ENTRY, nLastID) == 12);
        ASSERT(offsetof(AFX_MSGMAP_ENTRY, nSig) == 16);

        _asm
        {
                        MOV     EBX,lpEntry
                        MOV     EAX,nMsg
                        MOV     EDX,nCode
                        MOV     ECX,nID
        __loop:
                        CMP     DWORD PTR [EBX+16],0        ; nSig (0 => end)
                        JZ      __failed
                        CMP     EAX,DWORD PTR [EBX]         ; nMessage
                        JE      __found_message
        __next:
                        ADD     EBX,SIZE AFX_MSGMAP_ENTRY
                        JMP     short __loop
        __found_message:
                        CMP     EDX,DWORD PTR [EBX+4]       ; nCode
                        JNE     __next
        // message and code good so far
        // check the ID
                        CMP     ECX,DWORD PTR [EBX+8]       ; nID
                        JB      __next
                        CMP     ECX,DWORD PTR [EBX+12]      ; nLastID
                        JA      __next
        // found a match
                        MOV     lpEntry,EBX                 ; return EBX
                        JMP     short __end
        __failed:
                        XOR     EAX,EAX                     ; return NULL
                        MOV     lpEntry,EAX
        __end:
        }
        return lpEntry;
#else  // _AFX_PORTABLE
        // C version of search routine
        while (lpEntry->nSig != AfxSig_end)
        {
                if (lpEntry->nMessage == nMsg && lpEntry->nCode == nCode &&
                        nID >= lpEntry->nID && nID <= lpEntry->nLastID)
                {
                        return lpEntry;
                }
                lpEntry++;
        }
        return NULL;    // not found
```

```
#endif  // _AFX_PORTABLE
}
```

If CCmdTarget::OnCmdMsg() evaluates lpEnty as “not NULL” (i.e. finds a handler in the message map), then it calls _AfxDispatchCmdMsg() which is shown also in CMDTARG.CPP (List 5-30):

List 5-30. _AfxDispatchCmdMsg() in CMDTARG.CPP

```
/////////////////////////////////////////////////////////////////////////////
// CCmdTarget windows message dispatching

AFX_STATIC BOOL AFXAPI _AfxDispatchCmdMsg(CCmdTarget* pTarget, UINT nID, int nCode, AFX_PMSG pfn,
	void* pExtra, UINT nSig, AFX_CMDHANDLERINFO* pHandlerInfo)
		// return TRUE to stop routing
{
	ASSERT_VALID(pTarget);
	UNUSED(nCode);   // unused in release builds

	union MessageMapFunctions mmf;
	mmf.pfn = pfn;
	BOOL bResult = TRUE; // default is ok

	if (pHandlerInfo != NULL)
	{
		// just fill in the information, don't do it
		pHandlerInfo->pTarget = pTarget;
		pHandlerInfo->pmf = mmf.pfn;
		return TRUE;
	}

	switch (nSig)
	{
	case AfxSig_vv:
		// normal command or control notification
		ASSERT(CN_COMMAND == 0);        // CN_COMMAND same as BN_CLICKED
		ASSERT(pExtra == NULL);
		(pTarget->*mmf.pfn_COMMAND)();
		break;

	case AfxSig_bv:
		// normal command or control notification
		ASSERT(CN_COMMAND == 0);        // CN_COMMAND same as BN_CLICKED
		ASSERT(pExtra == NULL);
		bResult = (pTarget->*mmf.pfn_bCOMMAND)();
		break;

	case AfxSig_vw:
		// normal command or control notification in a range
		ASSERT(CN_COMMAND == 0);        // CN_COMMAND same as BN_CLICKED
		ASSERT(pExtra == NULL);
		(pTarget->*mmf.pfn_COMMAND_RANGE)(nID);
		break;

<<omitted>>

	case AfxSig_cmdui:
		{
			// ON_UPDATE_COMMAND_UI or ON_UPDATE_COMMAND_UI_REFLECT case
			ASSERT(CN_UPDATE_COMMAND_UI == (UINT)-1);
			ASSERT(nCode == CN_UPDATE_COMMAND_UI || nCode == 0xFFFF);
			ASSERT(pExtra != NULL);
			CCmdUI* pCmdUI = (CCmdUI*)pExtra;
			ASSERT(!pCmdUI->m_bContinueRouting);    // idle - not set
			(pTarget->*mmf.pfn_UPDATE_COMMAND_UI)(pCmdUI);
			bResult = !pCmdUI->m_bContinueRouting;
			pCmdUI->m_bContinueRouting = FALSE;     // go back to idle
		}
```

```
                break;

        case AfxSig_cmduiw:
                {
                        // ON_UPDATE_COMMAND_UI case
                        ASSERT(nCode == CN_UPDATE_COMMAND_UI);
                        ASSERT(pExtra != NULL);
                        CCmdUI* pCmdUI = (CCmdUI*)pExtra;
                        ASSERT(pCmdUI->m_nID == nID);           // sanity assert
                        ASSERT(!pCmdUI->m_bContinueRouting);    // idle - not set
                        (pTarget->*mmf.pfn_UPDATE_COMMAND_UI_RANGE)(pCmdUI, nID);
                        bResult = !pCmdUI->m_bContinueRouting;
                        pCmdUI->m_bContinueRouting = FALSE;     // go back to idle
                }
                break;

        // general extensibility hooks
        case AfxSig_vpv:
                (pTarget->*mmf.pfn_OTHER)(pExtra);
                break;
        case AfxSig_bpv:
                bResult = (pTarget->*mmf.pfn_OTHER_EX)(pExtra);
                break;

        default:   // illegal
                ASSERT(FALSE);
                return 0;
        }
        return bResult;
}
```

5.5.4 _AfxDispatchCmdMsg() Calling Message Handler

Since the function _AfxDispatchCmdMsg() is declared static (i.e. AFX_STATIC), it is visible only within CMDTARG.CPP. One of the parameters is the function signature. This signature comes from the message map entry itself. We have already seen the structure of the entries into the message map table AFX_MSGMAP_ENTRY in 3.2 in (III-1) which is cited here again for convenience. We notice that a pointer which points to the routine handling the message is also found within the message map entry (i.e. AFX_PMSG pfn).

List 5-31. struct AFX_MSGMAP_ENTRY in AFXWIN.H

```
struct AFX_MSGMAP_ENTRY
{
        UINT nMessage;   // windows message
        UINT nCode;      // control code or WM_NOTIFY code
        UINT nID;        // control ID (or 0 for windows messages)
        UINT nLastID;    // used for entries specifying a range of control id's
        UINT nSig;       // signature type (action) or pointer to message #
        AFX_PMSG pfn;    // routine to call (or special value)
};
```

It should be noted in List 5-30 that _AfxDispatchCmdMsg() switches on the function signature, performing different operations depending on whether the signature is for a regular command, an extended command, or a command user-interface handler. In the case of a regular menu command, the signature is AfxSig_vv (void return, void parameter list). _AfxDispatchCmdMsg() immediately calls the message handler, and the handler for that message is called.

If CCmdTarget::OnCmdMsg() fails to find a handler within the message map, it returns FALSE, which eventually causes CWnd::DefWindProc() to handle the messasge (see List 5-22).

Here we take one example. One of the most important messages of all is the WM_COMMAND message sent when we select an item from the menu. The low word of the message's wParam parameter holds the item's command ID. We can confirm it at the beginning of List 5-25. An ON_COMMAND macro in the message map links WM_COMMAND messages referencing a particular menu item to the class member function, or command handler of our choice (see List 3-5 in (III-1)). When OnWndMsg gets a message, it searches our window object's message map for an entry with a command ID that matches the received message. We take one more example from our own MSS application. When we start the application the pop-up menu appears immediately. The pop-up menu itself is a dialog box that contains [OK] and [Cancel] buttons in it. Suppose we select the first menu item "Describe" and click on the [OK] button. The event "clicking on [OK] button" does originate a WM_COMMAND message since the [OK] button control sends a notification to its parent i.e. its dialog box. We can trace the following function calling chain that the present WM_COMMAND triggers. We follow the function calling chain, starting with **AfxWndProc** (HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)[List 5-6 (III-2)] → **LRESULT AFXAPI AfxCallWndProc** (CWnd* pWnd, HWND hWnd, UINT nMsg, WPARAM wParam = 0, LPARAM lParam = 0)[List 5-20] → **LRESULT CWnd::WindowProc**(UINT message, WPARAM wParam, LPARAM lParam)[List 5-22] → **BOOL CWnd::OnWndMsg** (UINT message, WPARAM wParam, LPARAM lParam, LRESULT* pResult)[List 5-23] → **BOOL CWnd::OnCommand** (WPARAM wParam, LPARAM lParam) [List 5-25] → **BOOL CDialog:: OnCmdMsg** (UINT nID, int nCode, void* pExtra, AFX_CMDHANDLERINFO* pHandlerInfo)[List 5-32 below] → **AFX_STATIC BOOL AFXAPI _AfxDispatchCmdMsg** (CCmdTarget* pTarget, UINT nID, int nCode, AFX_PMSG pfn, void* pExtra, UINT nSig, AFX_CMDHANDLERINFO* pHandlerInfo)[List 5-30]. And in _AfxDispatchCmdMsg() the control enters the switch construction and ends up in the case "AfxSig_vv". The signature "AfxSig_vv" designates the type of the member function, in this case "void void", i.e. a parameterless member function with no return. It is understandable that the present command target class is CDialog.

```
switch (nSig) // nSig = signature code
        {
        case AfxSig_vv:
                // normal command or control notification
```

```
                ASSERT(CN_COMMAND == 0);          // CN_COMMAND same as BN_CLICKED
                ASSERT(pExtra == NULL);
                (pTarget->*mmf.pfn_COMMAND)();
                break;
<<omitted>>
        }
```

Here (pTarget->*mmf.pfn_COMMAND)() means that the object that is the current command target points the entry in the message map with the present message and that our handler is (pTarget->*mmf.pfn_COMMAND)().

List 5-32. CDialog::OnCmdMsg() in DLGCORE.CPP

```
BOOL CDialog::OnCmdMsg(UINT nID, int nCode, void* pExtra,
        AFX_CMDHANDLERINFO* pHandlerInfo)
{
        if (CWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;

        if ((nCode != CN_COMMAND && nCode != CN_UPDATE_COMMAND_UI) ||
                        !IS_COMMAND_ID(nID) || nID >= 0xf000)
        {
                // control notification or non-command button or system command
                return FALSE;           // not routed any further
        }

        // if we have an owner window, give it second crack
        CWnd* pOwner = GetParent();
        if (pOwner != NULL)
        {
#ifdef _DEBUG
                if (afxTraceFlags & traceCmdRouting)
                        TRACE1("Routing     command     id     0x%04X     to     owner
window.¥n", nID);
#endif
                ASSERT(pOwner != this);
                if (pOwner->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                        return TRUE;
        }

        // last crack goes to the current CWinThread object
        CWinThread* pThread = AfxGetThread();
        if (pThread != NULL)
        {
#ifdef _DEBUG
                if (afxTraceFlags & traceCmdRouting)
                        TRACE1("Routing command id 0x%04X to app.¥n", nID);
#endif
                if (pThread->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                        return TRUE;
        }
```

```
#ifdef _DEBUG
        if (afxTraceFlags & traceCmdRouting)
        {
                TRACE2("IGNORING command id 0x%04X sent to %hs dialog.¥n",
nID,
                        GetRuntimeClass()->m_lpszClassName);
        }
#endif
        return FALSE;
}
```

5.5.5 Standard Sequence of CCmdTarget-Derived Classes

We owe the description of the standard sequence of CCmdTarget-derived classes very much to Referece 9). As we have seen in detail above, MFC uses this command-routing scheme for all the CCmdTarget-derived classes. That includes classes derived from CWnd, CDocumnt, CView, and CFrameWnd. One interesting aspect of this arrangement is the path that commands take to get to their final destinations. All command messages take the same path for the first three steps. That is, the message first lands in AfxWndProc(), which gets the CWnd object from the HWND parameter and calls _AfxCallWndProc(). And _AfxCallWndProc() calls the CWnd-derived object's Windowproc(). From there, the message is routed to its inteded destination.

Here is a rundown of the path a command message takes to the various components of an MFC application.

Command to a Frame Window

Here is the path a WM_COMMAND message takes to an application's frame window. As with all Windows messages through an MFC program, the first stop is AfxWndProc(). This calls _AfxCallWndProc(), finally ending up in the specific Window's window procedure. From there the command message is routed to the appropriate command target.

AfxWndProc() → _AfxCallWndProc() → CWnd::Windowproc() → CWnd::OnWndMsg() → CFramwWnd::OnCommand() → CWnd::OnCommand() → CFrameWnd::OnCmdMsg() → CCmdTarget::OnCmdMsg() → _AfxDispatchCmdMsg() → CMainFrame::OnFrameAframecommand()

Command to a Document

Here is the path that a WM_COMMAND message takes to an application's document:

AfxWndProc() → _AfxCallWndProc() → CWnd::Windowproc() → CWnd::OnWndMsg() → CFramwWnd::OnCommand() → CWnd::OnCommand() → CFrameWnd::OnCmdMsg() → CView::OnCmdMsg() → CDocument::OnCmdMsg() → CCmdTarget::OnCmdMsg() → _AfxDispatchCmdMsg() → CSdiappDoc::OnDocAdoccommand()

Here shown is CDocument::OnCmdMsg() in List 5-33.

List 5-33. CDocument::OnCmdMsg() in DOCCORE.CPP

```
BOOL CDocument::OnCmdMsg(UINT nID, int nCode, void* pExtra,
        AFX_CMDHANDLERINFO* pHandlerInfo)
{
        if (CCmdTarget::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;

        // otherwise check template
        if (m_pDocTemplate != NULL &&
          m_pDocTemplate->OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
                return TRUE;

        return FALSE;
}
```

Command to a View

Here is the path that a WM_COMMAND message takes to an application's view:

AfxWndProc() → _AfxCallWndProc() → CWnd::Windowproc() → CWnd::OnWndMsg() → CFramwWnd::OnCommand() → CWnd::OnCommand() → CFrameWnd::OnCmdMsg() → CView::OnCmdMsg() → CCmdTarget::OnCmdMsg() → _AfxDispatchCmdMsg() → CSdiappView::OnViewAViewcommand()

Command to an App

Here is the path that a WM_COMMAND message takes to an application's CWinApp-derived object:

AfxWndProc() → _AfxCallWndProc() → CWnd::Windowproc() → CWnd::OnWndMsg() → CFramwWnd::OnCommand() → CWnd::OnCommand() → CFrameWnd::OnCmdMsg() → CView::OnCmdMsg() → CCmdTarget::OnCmdMsg() → _AfxDispatchCmdMsg() → CSdiappApp::OnAppAnappcommand()

Command to a Dialog Box

Dialog boxes also receive command messages. Here is the path a WM_COMMAND message takes to a dialog box:

AfxWndProc() → _AfxCallWndProc() → CWnd::Windowproc() → CWnd::OnWndMsg() → CWnd::OnCommand() → CFrameWnd::OnCmdMsg() → CDialog::OnCmdMsg() → CCmdTarget::OnCmdMsg() → _AfxDispatchCmdMsg() → CAboutDlg::OnAButton()

This is how command messages come through the framework. The message goes caroming like billiard balls between several different classes. Handling regular window messages (like WM_SIZE) is quite a bit simpler which is elaborated in (III-4).

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. 2, March 2005.

(4) Noto, Hirosi. Development of a Management Support System On the Windows Platform (III-Part 2): Message Pumping and Message Handling, Hokusei Review, The School of Economics (Hokusei Gakuen University) Vol. 45, No. 1, September 2005.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Abstract]

Development of a Management Support System on the Windows Platform (III-Part 3):

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 the 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, WM_COMMAND in Commands and Control Notifications, Message Map Entry with Message, Standard Sequence of CCmdTarget-Derived Classes