北星論集(経)　第46巻 第１号（通巻第50号）　September 2006

Development of a Management Support System on the Windows Platform (III-Part 4): Message Pumping and Message Handling

Hiroshi NOTO

Contents

Section 5　How MFC Uses Message Maps and Handles Messages

In Section 5 up to the previous subsection[5] have we seen the traditional message loop and the generic window procedure AfxWndProc()[6)−13)]. At the lowest level the messages are

Key Words: 1) Command-Routing and Message-Dispatching Architecture
2) C++ with MFC (Microsoft Foundation Class) Library on Windows
3) WM_COMMAND in Commands and Control Notifications
4) Regular Window Messages
5) Hooking into Message Loop (PreTranslateMessage())

handled by the traditional message loop. Messages dispatched with DispatchMessage() generate calls to the window procedure WndProc(). In a traditional program for Windows all messages to a window are processed in its window procedure WndProc(). The MFC class library, however, provides more efficient ways of handling messages by using Windows hooking mechanism that eventually wires messages to the AfxWndProc()[10), 13)]: messages are hooked up and AfxWndProc() handles the messages in command-routing and message-dispatching mechanism. MFC handles WM_COMMAND commands through the command-routing and window messages through the message-dispatching, the message maps associating the commands and messages with the member functions that handle the commands and messages. In (III-3)[*)] we have examined the message handling mechanism in WM_COMMAND in Commands and Control Notifications[12)].

5.6 Regular Window Messages

As with command messages, a regular window message[12)] starts out in AfxWndProc(), which goes on to turn the window handle into a CWnd object. AfxWndProc() then calls AfxCallWndProc(), which in turn calls the CWnd object's WindowProc(). WindowProc() then calls OnWndMsg(), which calls AfxFindMessageEntry() to find a handler for the message.

5.6.1 AfxFindMessageEntry()

The AfxFindMessageEntry() function is already shown in List 5-29 in (III-3) that is coded in Assembler in part. To find the message map entry in the table, OnWndMsg() first retrieves the CWnd object's message map. Then, given the first entry in that message map, AfxFindMessageEntry() walks the array of message map entries until it either (a) finds the message in the message map or (b) finds the end of the message map as marked by the AfxSig_end signature (this is what the END_MESSAGE_MAP macro in List 5-34 does [List 3-3 in (III-1)]).

List 5-34. END_MESSAGE_MAP() in AFXWIN.H

```
#define END_MESSAGE_MAP()
                    {0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 }
        };
```

When OnWndMsg() finds a handler, it calls the handler (see below). If the message map does not include a handler for a specific message, the framework calls the window's default window procedure DefWindProc() (List 5-22 in (III-3)). So, unlike commands, which are routed to several places, regular window messages go straight to the window for which

[*)] Hereafter the series of our articles "Development of a Management Support System on the Windows Platform"[1)−5)] will be abbreviated as (I), (II), (III-1) or (III-4).

they were intended. To begin with, let us take an example in our MSS (Management Support System) system. Suppose we change the size of a child frame window "Describe" or "Goalseek" or "Reasoning"[1]. A WM_SIZE message arrives at the child frame in question anytime the child window is resized. Here is the path the WM_SIZE message takes to an application's view. It should be noted how window messages make a beeline for their CWnd-derived objects.

AfxWndProc() → AfxCallWndProc() → CWnd::Windowproc() → CWnd::OnWndMsg() → CMDIChildWnd::OnSize() → CFrameWnd::OnSize() → CWnd::OnSize()

In our case the message map has an ON_WM_SIZE entry defined in AFXMSG_.H (see below) where we can see the handler function OnSize which takes a CWnd pointer. OnSize handler is characterized by the signature "AfxSig_vwii" in this case.

```
#define ON_WM_SIZE()
{WM_SIZE, 0, 0, 0, AfxSig_vwii,
(AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(UNIT, int, int))&OnSize},
```

The function CWnd::OnSize() is defined in AFXWIN2.INL. Here Default() of course calls DefWindProc().

```
_AFXWIN_INLINE void CWnd::OnSize(UNIT, int, int)
        {Default();}
```

5.6.2 Calling Member Function

Before leaving message maps, let us take a look at how MFC calls the member function for a particular window message once it finds the entry in the message map. Let us recall that one of the members of the message map entry structure is a pointer to the function handling the window message. To deal with this, OnWndMsg() declares a MessageMapFunction union on the stack (List 5-35).

List 5-35. a portion of the union MessageMapFunctions in AFXIMPL.H

```
union MessageMapFunctions
{
            AFX_PMSG pfn;    // generic member function pointer

            // specific type safe variants for WM_COMMAND and WM_NOTIFY messages
            void (AFX_MSG_CALL CCmdTarget::*pfn_COMMAND)();
            BOOL (AFX_MSG_CALL CCmdTarget::*pfn_bCOMMAND)();
            void (AFX_MSG_CALL CCmdTarget::*pfn_COMMAND_RANGE)(UINT);
            BOOL (AFX_MSG_CALL CCmdTarget::*pfn_COMMAND_EX)(UINT);

            void (AFX_MSG_CALL CCmdTarget::*pfn_UPDATE_COMMAND_UI)(CCmdUI*);
            void (AFX_MSG_CALL CCmdTarget::*pfn_UPDATE_COMMAND_UI_RANGE)(CCmdUI*, UINT);
            void  (AFX_MSG_CALL CCmdTarget::*pfn_OTHER)(void*);
            BOOL  (AFX_MSG_CALL CCmdTarget::*pfn_OTHER_EX)(void*);

            void  (AFX_MSG_CALL CCmdTarget::*pfn_NOTIFY)(NMHDR*, LRESULT*);
            BOOL  (AFX_MSG_CALL CCmdTarget::*pfn_bNOTIFY)(NMHDR*, LRESULT*);
            void  (AFX_MSG_CALL CCmdTarget::*pfn_NOTIFY_RANGE)(UINT, NMHDR*, LRESULT*);
            BOOL  (AFX_MSG_CALL CCmdTarget::*pfn_NOTIFY_EX)(UINT, NMHDR*, LRESULT*);

            // type safe variant for thread messages

            void  (AFX_MSG_CALL CWinThread::*pfn_THREAD)(WPARAM, LPARAM);

            // specific type safe variants for WM-style messages
            BOOL      (AFX_MSG_CALL CWnd::*pfn_bD)(CDC*);
            BOOL      (AFX_MSG_CALL CWnd::*pfn_bb)(BOOL);
            BOOL      (AFX_MSG_CALL CWnd::*pfn_bWww)(CWnd*, UINT, UINT);
            BOOL      (AFX_MSG_CALL CWnd::*pfn_bHELPINFO)(HELPINFO*);
            BOOL      (AFX_MSG_CALL CWnd::*pfn_bWCDS)(CWnd*, COPYDATASTRUCT*);
            HBRUSH   (AFX_MSG_CALL CWnd::*pfn_hDWw)(CDC*, CWnd*, UINT);
            HBRUSH   (AFX_MSG_CALL CWnd::*pfn_hDw)(CDC*, UINT);
            int       (AFX_MSG_CALL CWnd::*pfn_iwWw)(UINT, CWnd*, UINT);
            int       (AFX_MSG_CALL CWnd::*pfn_iww)(UINT, UINT);
            int       (AFX_MSG_CALL CWnd::*pfn_iWww)(CWnd*, UINT, UINT);
            int       (AFX_MSG_CALL CWnd::*pfn_is)(LPTSTR);
            LRESULT  (AFX_MSG_CALL CWnd::*pfn_lwl)(WPARAM, LPARAM);
            LRESULT  (AFX_MSG_CALL CWnd::*pfn_lwwM)(UINT, UINT, CMenu*);
            void      (AFX_MSG_CALL CWnd::*pfn_vv)(void);

            void      (AFX_MSG_CALL CWnd::*pfn_vw)(UINT);
            void      (AFX_MSG_CALL CWnd::*pfn_vww)(UINT, UINT);
            void      (AFX_MSG_CALL CWnd::*pfn_vvii)(int, int);
            void      (AFX_MSG_CALL CWnd::*pfn_vwww)(UINT, UINT, UINT);
            void      (AFX_MSG_CALL CWnd::*pfn_vwii)(UINT, int, int);
            void      (AFX_MSG_CALL CWnd::*pfn_vwl)(WPARAM, LPARAM);
            void      (AFX_MSG_CALL CWnd::*pfn_vbWW)(BOOL, CWnd*, CWnd*);
            void      (AFX_MSG_CALL CWnd::*pfn_vD)(CDC*);
            void      (AFX_MSG_CALL CWnd::*pfn_vM)(CMenu*);
            void      (AFX_MSG_CALL CWnd::*pfn_vMwb)(CMenu*, UINT, BOOL);

            void      (AFX_MSG_CALL CWnd::*pfn_vW)(CWnd*);
            void      (AFX_MSG_CALL CWnd::*pfn_vWww)(CWnd*, UINT, UINT);
            void      (AFX_MSG_CALL CWnd::*pfn_vWp)(CWnd*, CPoint);
            void      (AFX_MSG_CALL CWnd::*pfn_vWh)(CWnd*, HANDLE);
            void      (AFX_MSG_CALL CWnd::*pfn_vwW)(UINT, CWnd*);
            void      (AFX_MSG_CALL CWnd::*pfn_vwWb)(UINT, CWnd*, BOOL);
            void      (AFX_MSG_CALL CWnd::*pfn_vwwW)(UINT, UINT, CWnd*);
            void      (AFX_MSG_CALL CWnd::*pfn_vwwx)(UINT, UINT);
            void      (AFX_MSG_CALL CWnd::*pfn_vs)(LPTSTR);
            void      (AFX_MSG_CALL CWnd::*pfn_vOWNER)(int, LPTSTR);    // force return TRUE
            int       (AFX_MSG_CALL CWnd::*pfn_iis)(int, LPTSTR);
            UINT      (AFX_MSG_CALL CWnd::*pfn_wp)(CPoint);
            UINT      (AFX_MSG_CALL CWnd::*pfn_wv)(void);
            void      (AFX_MSG_CALL CWnd::*pfn_vPOS)(WINDOWPOS*);
            void      (AFX_MSG_CALL CWnd::*pfn_vCALC)(BOOL, NCCALCSIZE_PARAMS*);
            void      (AFX_MSG_CALL CWnd::*pfn_vwp)(UINT, CPoint);
            void      (AFX_MSG_CALL CWnd::*pfn_vwwh)(UINT, UINT, HANDLE);
            BOOL      (AFX_MSG_CALL CWnd::*pfn_bwsp)(UINT, short, CPoint);
            void      (AFX_MSG_CALL CWnd::*pfn_vws)(UINT, LPCTSTR);
};
```

The MessageMapFunctions union contains a generic AFX function pointer as a member, followed by prototypes for all the different kinds of functions that might be used for a message handler. OnWndMsg() sets the MessageMapFunctions union to the message handler's address mmf (List 5-23 in (III-3)):

```
mmf.pfn = lpEntry->pfn;
```

Next, OnWndMsg() goes on to find the signature that matches the signature for a message, e.g. WM_SIZE. Once OnWndMsg() finds a match, it pulls the necessary parameters from the WPARAM and LPARAM and calls the handler using the prototype included in the MessageMapFunctions structure. For example, here is the line of code that executes a handler for WM_SIZE in OnWndMsg() in WINCORE.CPP.

```
case AfxSig_vvii:
                (this->*mmf.pfn_vvii)((short)LOWORD(lParam), (short)HIWORD(lParam));
                break;
```

It should be emphasized that MFC's message-mapping code is sometimes a little hard to follow. However, it does work very well and it is an important component of Visual C++'s Wizard technology[8].

MFC does not switch on the message ID (WM_CREATE, WM_SETFOCUS, and so on), but rather on the signature of the message handler function. When OnWndMsg() gets a message, it searches our window object's message map for an entry with a message ID that matches the received message. Here is one more example. Suppose our message map has an ON_WM_SETFOCUS entry. ON_WM_SETFOCUS is a macro that generates an entry in our message map defined in AFXMSG_.H.

```
#define ON_WM_SETFOCUS()
{WM_SETFOCUS, 0, 0, 0, AfxSig_vW, &OnSetFocus
}
```

Among the members in the message map entry is the special code AfxSig_vW that identifies the signature of the handler function. In this case, OnSetFocus takes a CWnd pointer (W) and returns void (v) — hence AfxSig_vW. OnWndMsg uses this code to dispatch to the handler function:

```
switch (nSig)
{
case AfxSig_vW:
                (this->*mmf.pfn_vW)(CWnd::FromHandle((HWND)wParam));

break;
```

Here, this->*mmf.pfn_vW is our handler function. Other message map entries will have different signature codes.

Why does MFC dispatch on the signature code instead of the message ID? Because one of the things MFC must do before calling our handler function is to package its arguments

—that is, to cover WPARAM and LPARAM into more typesafe objects such as CWnd* or BOOL. It is more efficient to lump together all handler functions with a given signature so that MFC can cover the arguments before calling the handler function. There are about 56 different codes defined in a afxmsg_.h. It is important to understand how MFC dispatches WM_XXX messages to our handler functions, but it is equally important to realize that the message-routing code is universal for all window classes. MFC uses the same AfxWndProc() for all window objects[13].

5.7 Control Notifications (Other Kinds of Messages)

In addition to WM_COMMAND and the regular window messages, MFC handles certain other window messages in special ways[10]. The WM_COMMAND and various window messages (like WM_SIZE and WM_MOVE) are the most common kinds of messages our application will receive. However, MFC has special ways of handle the WM_NOIFY, the WM_ACTIVE, and the WM_SETCURSOR messages. Let us take a brief look at these special cases.

5.7.1 WM_NOTIFY and Message Reflection

There is another message called WM_NOTIFY. WM_NOTIFY is a generalized control notification from controls to their parents with any amount of additional data in a standardized fashion. WM_NOTIFY is always a notification, whereas WM_COMMAND can be either a command or a notification. CWnd::OnWndMsg() handles WM_NOTIFY specifically through CWnd::OnNotify() member function.

List 5-36. CWnd::OnNotify() in WINCORE.CPP

```
BOOL CWnd::OnNotify(WPARAM, LPARAM lParam, LRESULT* pResult)
{
        ASSERT(pResult != NULL);
        NMHDR* pNMHDR = (NMHDR*)lParam;
        HWND hWndCtrl = pNMHDR->hwndFrom;

        // get the child ID from the window itself
        UINT nID = _AfxGetDlgCtrlID(hWndCtrl);
        int nCode = pNMHDR->code;

        ASSERT(hWndCtrl != NULL);
        ASSERT(::IsWindow(hWndCtrl));

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

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

        AFX_NOTIFY notify;
        notify.pResult = pResult;
        notify.pNMHDR = pNMHDR;
        return OnCmdMsg(nID, MAKELONG(nCode, WM_NOTIFY), &notify, NULL);
}
```

Instead of passing the regular WPARAM and LPARAM arguments, WM_NOTIFY packs a NMHDR structure in the LPARAM. The NMHDR structure contains the handle (hwndFrom) and ID (idFrom) of the control sending the message and the notification code (code):

```
Struct NMHDR{
        HWND hwndFrom;   // control that sent notification
        UINT idFrom;       // ID of control
        UINT code;          // notification code
};
```

OnNotify() uses notify address (&notify) and ends up calling the virtual function OnCmdMsg(). OnCmdMsg() then sends the message directly to the control so that the control can deal with it (after all, it really is the control's responsibility).

Windows controls frequently send notificaiton messages to their parent windows. In Windows and in MFC before version 4.0, the parent window is responsible for handling these messages. In MFC 4.0 or later, in addition to the parent window's handling notification messages, MFC provides a new mechanism called "message reflection" that enables the control to get its notifications back from its parent window[10), 15), 16)]. The message reflection mechanism allows the notification messages from child controls to be handled in either the child control window or the parent window, or in both. When a WM_NOTIFY message is sent, the control is offered the first chance to handle it. If any other reflected message is sent, this means that the parent window has the first chance to handle it and that the control will receive the reflected message. To do so, it will need a handler function and an appropriate entry in the control's class message map. The new message-map macro for reflected messages is slightly different than for regular notifications: it has _REFLECT appended to its usual name. For instance, to handle a WM_NOTIFY message in the parent, we use the macro ON_NOTIFY in the parent's message map. To handle the reflected message in the child control, we use the ON_NOTIFY_REFLECT macro in the child control's message map. In this way, the control can handle the message rather than depends on the parent to deal with message reflection.

5.7.2 WM_ACTIVE

MFC's messaging architecture also handles WM_ACTIVE. OnWndMsg() calls the non-C++ member function _AfxHandleActivate(). This function checks to see if the WM_ACTIVE message is for the top-level window. If it is, _AfxHandleActivate() sends the WM_ACTIVETOPLEVEL message to the top-level window.

List 5-37. _AfxHandleActivate() in WINCORE.CPP

```
AFX_STATIC void AFXAPI
```

```
_AfxHandleActivate(CWnd* pWnd, WPARAM nState, CWnd* pWndOther)
{
        ASSERT(pWnd != NULL);

        // send WM_ACTIVATETOPLEVEL when top-level parents change
        CWnd* pTopLevel;
        if (!(pWnd->GetStyle() & WS_CHILD) &&
                (pTopLevel = pWnd->GetTopLevelParent()) != pWndOther->GetTopLevelParent())
        {
                // lParam points to window getting the WM_ACTIVATE message and
                //   hWndOther from the WM_ACTIVATE.
                HWND hWnd2[2];
                hWnd2[0] = pWnd->m_hWnd;
                hWnd2[1] = pWndOther->GetSafeHwnd();
                // send it...
                pTopLevel->SendMessage(WM_ACTIVATETOPLEVEL, nState, (LPARAM)&hWnd2[0]);
          }
}
```

5.7.3 WM_SETCURSOR

MFC handles the WM_SETCURSOR message within the function _AfxHandleSetCursor(). _AfxHandleSetCursor() checks to see if one of the mouse buttons is pressed. If a mouse button is down, _AfxHandleSetCursor() activates the last active window. This is necessary to work around a Windows bug. If a window is disabled and a modal dialog is parented to that disabled window, the application will not activate itself correctly when the user clicks on the disabled window. Instead, the user must click directly on the dialog itself. Besides if the dialog is small, it is probably covered with other windows. Normally, clicking on the disabled window would result in just a beep from DefWindowProc() in this circumstance. MFC fixes this Windows bug by activating the dialog box (and thus bringing the application to the top) when we click on a disabled window with an active modal dialog box.

List 5-38. _AfxHandleSetCursor() in WINCORE.CPP

```
AFX_STATIC BOOL AFXAPI
_AfxHandleSetCursor(CWnd* pWnd, UINT nHitTest, UINT nMsg)
{
        if (nHitTest == HTERROR &&
                (nMsg == WM_LBUTTONDOWN || nMsg == WM_MBUTTONDOWN ||
                 nMsg == WM_RBUTTONDOWN))
{
           // activate the last active window if not active
           CWnd* pLastActive = pWnd->GetTopLevelParent();
           if (pLastActive != NULL)
                   pLastActive = pLastActive->GetLastActivePopup();
           if (pLastActive != NULL &&
                   pLastActive != CWnd::GetForegroundWindow() &&
                   pLastActive->IsWindowEnabled())
           {
                   pLastActive->SetForegroundWindow();
                   return TRUE;
           }
}
return FALSE;
      }
```

5.8 Graphical Summary of MFC's Windows Message Handling Mechanism

In this subsection we are going to summarize how MFC handles the Windows messages depending on the three message categories: regular window messages, command messages and control notifications[12].

In Figure 5-1 the standard sequence of MFC's message handling mechanism[10), 14)] is displayed as the block diagram where the rectangles designate the member functions of classes and the ellipses particular commands belonging to the member functions drawn immediately before them. Let us start with the MFC application framework window procedure AfxWndProc(). As described earlier and as we all know, AfxWndProc() itself does not do the message dispatching at all. AfxWndProc(), therefore calls AfxCallWndProc(). AfxCallWndProc() makes a call to CWnd::Windowproc() that in turn calls CWnd::OnWndMsg(). The actual message dispatching is done in this function except for WM_COMMAND or WM_NOTIFY. The latter two commands, i.e. command messages and control notifications make a call to CCmdTarget::OnCmdMsg(), where AfxFindMessageEntry() calls _AfxDispatchCmdMsg() that gets the message map of our window class and that returns AFX_MSGMAP_ENTRY. Finally the required message handler function that the MessageMapFunctions union points to is executed like (this-> *mmf.pfn_vvii)() or (pTarget->*mmf.pfn_COMMAND)() depending on whether messages are regular window messages or command messages.

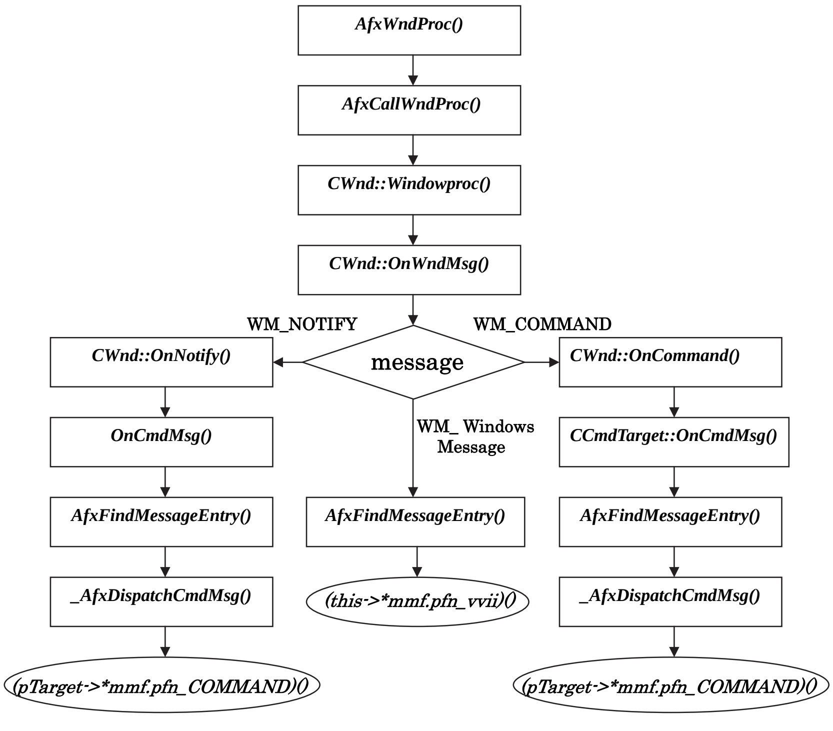

Figure 5-1. Standard sequence of MFC's message handling mechanism

Section 6.　Filtering Window Messages: Hooking into Message Loop

MFC implements a very ingenious and powerful feature in pumping messages. We can use PreTranslateMessage() to insert message handling into the message pump. PreTranslateMessage() is the approved way of altering the way of MFC's normal message dispatching. We should be careful, however, to override PreTranslateMessage() only at the level needed. As for the PreTranslateMessage() function, MFC gives us two places where we can hook into the message loop: CWinApp::PreTranslateMessage() and CWnd::PreTranslateMessage(). CWinApp::PreTranslateMessage() enables us to process window messages even before they get to any of our application's windows or command targets.
CWnd::PreTranslateMessage() is used by CWinApp (e.g. CWinApp::PreTranslateMessage()) to translate window messages before they are dispatched to the TranslateMessage() and DispatchMessage() Windows function.

In the main message pump inside CWinThread, specifically in CWinThread::PreTranslateMessage(), MFC calls a function CWnd::WalkPreTranslateTree() that loops through all the parent windows of the window for which the message is destined, calling CWnd::PreTranslateMessage() for each loop and stops if any parent window's PreTranslateMessage() returns TRUE. The upshot of the member function PreTranslateMessage() of CWinThread is that we can override PreTranslateMessage() to intercept messages sent to our window's descendants. This is another hooking function that filters window messages before they are dispatched to the Windows functions TranslateMessage() and DispatchMessage()[10), 17)].

6.1.　PreTranslateMessage()

As for the Message Pumping, MFC has another useful feature: we can plug into the message loop to handle messages before they ever get to their designated command targets or windows. The function for doing that is PreTranslateMessage(). There are few interesting cases where PreTranslateMessage() makes sense. For example one is in detecting some other application-wide keystroke or mouse operation. PreTranslateMessage() can come in quite handy in dialog-based applications for handling keyboard messages or several mouse-related messages (such as WM_LBUTTONDOWN or WM_LBUTTONDBLCLK). It is, however, not necessary to override, say, CMainFrame::PreTranslateMessage() if all we are interested in are messages going to children of a particular view. In that case we just override PreTranslateMessage() for the view class.

CWinApp::Run() calls CWinApp::PreTranslateMessage() before the message is processed by TranslateMessage() and DispatchMessage(): CWinThread::PumpMessage() [List 2-4 (III-1)] in CWinThread::Run() calls PreTranslateMessage()[see List 6-1]. This means that our application program CWinApp::Run() eventually calls CWinThread::PumpMessage() by inheritance.

List 6-1. CWinThread::PreTranslateMessage() in THRDCORE.CPP

```
BOOL CWinThread::PreTranslateMessage(MSG* pMsg)
{
        ASSERT_VALID(this);

        // if this is a thread-message, short-circuit this function
        if (pMsg->hwnd == NULL && DispatchThreadMessageEx(pMsg))
                return TRUE;

        // walk from target to main window
        CWnd* pMainWnd = AfxGetMainWnd();
        if (CWnd::WalkPreTranslateTree(pMainWnd->GetSafeHwnd(), pMsg))
                return TRUE;

        // in case of modeless dialogs, last chance route through main
        //    window's accelerator table
        if (pMainWnd != NULL)
        {
                CWnd* pWnd = CWnd::FromHandle(pMsg->hwnd);
                if (pWnd->GetTopLevelParent() != pMainWnd)
                        return pMainWnd->PreTranslateMessage(pMsg);
        }
        return FALSE;    // no special processing
}
```

Before calling WalkPreTranslateTree(), CWinApp::PreTranslateMessage() retrieves the window handle of the application's window by calling AfxGetMainWnd(). WalkPreTranslateTree() uses this handle to know when to stop calling PreTranslateMessage() walking from the target window to the main window (see the next subsection 6.2).

In the case of modeless dialogs, the application framework roots through the main window's accelerator table, i.e. through pMainWnd->PreTranslateMessage(). Here CWnd:: PreTranslateMessage() takes a single parameter: a pointer to the message structure (MSG* pMsg) [see List 6-2].

List 6-2. CWnd::PreTranslateMessage() in WINCORE.CPP

```
BOOL CWnd::PreTranslateMessage(MSG* pMsg)
{
    // handle tooltip messages (some messages cancel, some may cause it to popup)
    AFX_MODULE_STATE* pModuleState = _AFX_CMDTARGET_GETSTATE();
    if (pModuleState->m_pfnFilterToolTipMessage != NULL)
            (*pModuleState->m_pfnFilterToolTipMessage)(pMsg, this);

    // no default processing
    return FALSE;
}
```

By default, the CWnd-derived framework examines the message in CFrameWnd:: PreTranslateMessage() [List 6-3]. If the message is a mouse button down (WM_LBUTTONDOWN) or a mouse button down while the cursor is within the nonclient area of a window (WM_NCLBUTTONDOWN), CWnd::PreTranslateMessage() cancels any combobox or popups that could be in toolbars or dialog bars on the window.　Then CWnd::PreTranslateMessage() allows any tool tips to be filtered (i.e. removes any tool tips from the screen) by calling CancelToolTips().　This is realized in CWnd::PreTranslateMessage() in List 6-2 where (*pModuleStake->m_pfnFilterToolTipsMessagge)() is called to filter any tool tips.

List 6-3.　CFrameWnd::PreTranslateMessage() in WINFRM.CPP

```
BOOL CFrameWnd::PreTranslateMessage (MSG* pMsg)
{
        // check for special cancel modes for combo boxes
        if (pMsg->message == WM_LBUTTONDOWN || pMsg->message == WM_NCLBUTTONDOWN)
                AfxCancelModes(pMsg->hwnd);      // filter clicks

        // allow tooltip messages to be filtered
        if (CWnd::PreTranslateMessage(pMsg))
                return TRUE;

#ifndef _AFX_NO_OLE_SUPPORT
        // allow hook to consume message
        if (m_pNotifyHook != NULL && m_pNotifyHook->OnPreTranslateMessage (pMsg))
                return TRUE;
#endif

        if (pMsg->message >= WM_KEYFIRST && pMsg->message <= WM_KEYLAST)
        {
                // finally, translate the message
                HACCEL hAccel = GetDefaultAccelerator();
                return hAccel != NULL &&  ::TranslateAccelerator(m_hWnd, hAccel, pMsg);
        }
        return FALSE;
}
```

Since pModuleState is an instance of AFX_MODULE_STATE class (List 6-4), this member function invokes (PASCAL *m_pfnFilterToolTipMessage)() function as shown in List 6-4.

List 6-4.　class AFX_MODULE_STATE in AFXSTAT_.H

```
// AFX_MODULE_STATE (global data for a module)
class AFX_MODULE_STATE : public CNoTrackObject
{
public:
#ifdef  _AFXDLL
          AFX_MODULE_STATE(BOOL bDLL, WNDPROC pfnAfxWndProc, DWORD dwVersion);
          AFX_MODULE_STATE(BOOL bDLL, WNDPROC pfnAfxWndProc, DWORD dwVersion,
                        BOOL bSystem);
#else
          AFX_MODULE_STATE(BOOL bDLL);
#endif
          ~AFX_MODULE_STATE();

          CWinApp* m_pCurrentWinApp;
          HINSTANCE m_hCurrentInstanceHandle;
          HINSTANCE m_hCurrentResourceHandle;
```

```
        LPCTSTR m_lpszCurrentAppName;
        BYTE m_bDLL;     // TRUE if module is a DLL, FALSE if it is an EXE
        BYTE m_bSystem; // TRUE if module is a "system" module, FALSE if not
        BYTE m_bReserved[2]; // padding

        DWORD m_fRegisteredClasses; // flags for registered window classes

        // runtime class data
#ifdef  _AFXDLL
        CRuntimeClass* m_pClassInit;
#endif
        CTypedSimpleList<CRuntimeClass*> m_classList;

        // OLE object factories
#ifndef  _AFX_NO_OLE_SUPPORT
#ifdef  _AFXDLL
        COleObjectFactory* m_pFactoryInit;
#endif
        CTypedSimpleList<COleObjectFactory*> m_factoryList;
#endif
        // number of locked OLE objects
        long m_nObjectCount;
        BOOL m_bUserCtrl;

        // AfxRegisterClass and AfxRegisterWndClass data
        TCHAR m_szUnregisterList[4096];
#ifdef  _AFXDLL
        WNDPROC m_pfnAfxWndProc;
        DWORD m_dwVersion;  // version that module linked against
#endif
        // variables related to a given process in a module
        //  (used to be AFX_MODULE_PROCESS_STATE)
#ifdef  _AFX_OLD_EXCEPTIONS
        // exceptions
        AFX_TERM_PROC m_pfnTerminate;
#endif
        void (PASCAL *m_pfnFilterToolTipMessage)(MSG*, CWnd*);

#ifdef  _AFXDLL
        // CDynLinkLibrary objects (for resource chain)
        CTypedSimpleList<CDynLinkLibrary*> m_libraryList;

        // special case for MFCxxLOC.DLL (localized MFC resources)
        HINSTANCE m_appLangDLL;
#endif

#ifndef  _AFX_NO_OCC_SUPPORT
        // OLE control container manager
        COccManager* m_pOccManager;
        // locked OLE controls
        CTypedSimpleList<COleControlLock*> m_lockList;
#endif

#ifndef  _AFX_NO_DAO_SUPPORT
        _AFX_DAO_STATE* m_pDaoState;
#endif

#ifndef  _AFX_NO_OLE_SUPPORT
        // Type library caches
        CTypeLibCache m_typeLibCache;
        CTypeLibCacheMap* m_pTypeLibCacheMap;
#endif

        // define thread local portions of module state
        THREAD_LOCAL(AFX_MODULE_THREAD_STATE, m_thread)
};
```

CWnd::PreTranslateMessage() then deals with tool tips messages by calling the CWnd:: FilterToolTipMessage() function which is shown in List 6-5. Eventually the FilterToolTipMessage() member function is called by the framework to display or filter the tool tips messages associated with a button on the toolbar. It is normally called from PreTranslateMessage(). At the very bottom of CWnd::FilterToolTipMessage() (List 6-5) we can see that the CWnd::CancelToolTips() function is carried out.

List 6-5.　CWnd::FilterToolTipMessage()　in TOOLTIP.CPP

```
void CWnd::FilterToolTipMessage(MSG* pMsg)
{
        // this CWnd has tooltips enabled
        UINT message = pMsg->message;
        if ((message == WM_MOUSEMOVE || message == WM_NCMOUSEMOVE ||
                message == WM_LBUTTONUP || message == WM_RBUTTONUP ||
                message == WM_MBUTTONUP) &&
                (GetKeyState(VK_LBUTTON) >= 0 && GetKeyState(VK_RBUTTON) >= 0 &&
                GetKeyState(VK_MBUTTON) >= 0))
        {
                // make sure that tooltips are not already being handled
                CWnd* pWnd = CWnd::FromHandle(pMsg->hwnd);
                while (pWnd != NULL && !(pWnd->m_nFlags &
                                                        (WF_TOOLTIPS|WF_TRACKINGTOOLTIPS)))
                {
                        pWnd = pWnd->GetParent();
                }
                if (pWnd != this)
                {
                            if (pWnd == NULL)
                            {
                                    // tooltips not enabled on this CWnd, clear last state data
                                    _AFX_THREAD_STATE* pThreadState = _afxThreadState.GetData();
                                    pThreadState->m_pLastHit = NULL;
                                    pThreadState->m_nLastHit = -1;
                            }
                        return;
                }

                _AFX_THREAD_STATE* pThreadState = _afxThreadState.GetData();
                CToolTipCtrl* pToolTip = pThreadState->m_pToolTip;
                CWnd* pOwner = GetParentOwner();
                if (pToolTip != NULL && pToolTip->GetOwner() != pOwner)
                {
                        pToolTip->DestroyWindow();
                        delete pToolTip;
                        pThreadState->m_pToolTip = NULL;
                        pToolTip = NULL;
                }
                if (pToolTip == NULL)
                {
                        pToolTip = new CToolTipCtrl;
                        if (!pToolTip->Create(pOwner, TTS_ALWAYSTIP))
                        {
                                delete pToolTip;
                                return;
                        }
                        pToolTip->SendMessage(TTM_ACTIVATE, FALSE);
                        pThreadState->m_pToolTip = pToolTip;
                }
```

```
		ASSERT_VALID(pToolTip);
		ASSERT(::IsWindow(pToolTip->m_hWnd));

		// add a "dead-area" tool for areas between toolbar buttons
		TOOLINFO ti; memset(&ti, 0, sizeof(TOOLINFO));
		ti.cbSize = sizeof(AFX_OLDTOOLINFO);
		ti.uFlags = TTF_IDISHWND;
		ti.hwnd = m_hWnd;
		ti.uId = (UINT)m_hWnd;
		if (!pToolTip->SendMessage(TTM_GETTOOLINFO, 0, (LPARAM)&ti))
		{
			ASSERT(ti.uFlags == TTF_IDISHWND);
			ASSERT(ti.hwnd == m_hWnd);
			ASSERT(ti.uId == (UINT)m_hWnd);
			VERIFY(pToolTip->SendMessage(TTM_ADDTOOL, 0, (LPARAM)&ti));
		}

		// determine which tool was hit
		CPoint point = pMsg->pt;
		::ScreenToClient(m_hWnd, &point);
		TOOLINFO tiHit; memset(&tiHit, 0, sizeof(TOOLINFO));
		tiHit.cbSize = sizeof(AFX_OLDTOOLINFO);
		int nHit = OnToolHitTest(point, &tiHit);

		// build new toolinfo and if different than current, register it
		CWnd* pHitWnd = nHit == -1 ? NULL : this;
		if (pThreadState->m_nLastHit != nHit || pThreadState->m_pLastHit != pHitWnd)
		{
			if (nHit != -1)
			{
				// add new tool and activate the tip
				ti = tiHit;
				ti.uFlags &= ~(TTF_NOTBUTTON|TTF_ALWAYSTIP);
				if (m_nFlags & WF_TRACKINGTOOLTIPS)
					ti.uFlags |= TTF_TRACK;
				VERIFY(pToolTip->SendMessage(TTM_ADDTOOL, 0,
				(LPARAM)&ti));
				if ((tiHit.uFlags & TTF_ALWAYSTIP) || IsTopParentActive())
				{
					// allow the tooltip to popup when it should
					pToolTip->SendMessage(TTM_ACTIVATE, TRUE);
					if (m_nFlags & WF_TRACKINGTOOLTIPS)

pToolTip->SendMessage(TTM_TRACKACTIVATE, TRUE, (LPARAM)&ti);

					// bring the tooltip window above other popup windows
					::SetWindowPos(pToolTip->m_hWnd, HWND_TOP, 0, 0, 0, 0,

SWP_NOACTIVATE|SWP_NOSIZE|SWP_NOMOVE|SWP_NOOWNERZORDER);
				}
			}
			else
			{
				pToolTip->SendMessage(TTM_ACTIVATE, FALSE);
			}

			// relay mouse event before deleting old tool
			_AfxRelayToolTipMessage(pToolTip, pMsg);

			// now safe to delete the old tool
			if (pThreadState->m_lastInfo.cbSize >= sizeof(AFX_OLDTOOLINFO))
				pToolTip->SendMessage(TTM_DELTOOL, 0, (LPARAM)&
											pThreadState->m_lastInfo);

			pThreadState->m_pLastHit = pHitWnd;
			pThreadState->m_nLastHit = nHit;
```

```
                        pThreadState->m_lastInfo = tiHit;
                }
                else
                {
                        if (m_nFlags & WF_TRACKINGTOOLTIPS)
                        {
                                POINT pt;

                                ::GetCursorPos( &pt );
                                pToolTip->SendMessage(TTM_TRACKPOSITION, 0,
                                                                MAKELPARAM(pt.x, pt.y));
                        }
                        else
                        {
                                // relay mouse events through the tooltip
                                if (nHit != -1)
                                        _AfxRelayToolTipMessage(pToolTip, pMsg);
                        }
                }

                if ((tiHit.lpszText != LPSTR_TEXTCALLBACK) && (tiHit.hinst == 0))
                        free(tiHit.lpszText);
        }
        else if (m_nFlags & (WF_TOOLTIPS|WF_TRACKINGTOOLTIPS))
        {
                // make sure that tooltips are not already being handled
                CWnd* pWnd = CWnd::FromHandle(pMsg->hwnd);
                while (pWnd != NULL && pWnd != this && !(pWnd->m_nFlags &
                                                (WF_TOOLTIPS|WF_TRACKINGTOOLTIPS)))
                        pWnd = pWnd->GetParent();
                if (pWnd != this)
                        return;

                BOOL bKeys = (message >= WM_KEYFIRST && message <= WM_KEYLAST) ||
                        (message >= WM_SYSKEYFIRST && message <= WM_SYSKEYLAST);
                if ((m_nFlags & WF_TRACKINGTOOLTIPS) == 0 &&
                        (bKeys ||
                        (message == WM_LBUTTONDOWN || message == WMLBUTTONDBLCLK) ||
                        (message == WM_RBUTTONDOWN || message == WM_RBUTTONDBLCLK) ||
                        (message == WM_MBUTTONDOWN || message == WM_MBUTTONDBLCLK) ||
                        (message == WM_NCLBUTTONDOWN || messagev == WM_NCLBUTTONDBLCLK) ||
                        (message == WM_NCRBUTTONDOWN || message == WM_NCRBUTTONDBLCLK) ||
                        (message == WM_NCMBUTTONDOWN || message == WM_NCMBUTTONDBLCLK)))
                {
                        CWnd::CancelToolTips(bKeys);
                }
        }
}
```

The last "if statement" in List 6-3 handles the accelerators where GetDefaultAccelerator() gets a handle (HACCEL hAccel) to the accelerator table and the global function ::TranslateAccelerator() maps keystrokes to command IDs (i.e. WM_COMMAND messages). This is all the default implementation realized in CFrameWnd::PreTranslateMessage() in List 6-3.

6.2. WalkPreTranslateTree()

CWinApp::PreTranslateMessage() calls each of the windows from the target window (as designated by the window handle in the message structure) to the application's main window. CWnd::WalkPreTranslateTree() performs this process in CWinThread::

PreTranslateMessage(), as shown in List 6-6.

List 6-6. CWnd::WalkPreTranslateTree() in WINCORE.CPP

```
BOOL PASCAL CWnd::WalkPreTranslateTree(HWND hWndStop, MSG* pMsg)
{
        ASSERT(hWndStop == NULL || ::IsWindow(hWndStop));
        ASSERT(pMsg != NULL);

        // walk from the target window up to the hWndStop window checking
        //   if any window wants to translate this message

        for (HWND hWnd = pMsg->hwnd; hWnd != NULL; hWnd = ::GetParent(hWnd))
        {
                CWnd* pWnd = CWnd::FromHandlePermanent(hWnd);
                if (pWnd != NULL)
                {
                        // target window is a C++ window
                        if (pWnd->PreTranslateMessage(pMsg))
                                return TRUE; // trapped by target window (eg: accelerators)
                }

                // got to hWndStop window without interest
                if (hWnd == hWndStop)
                        break;
        }
        return FALSE;           // no special processing
}
```

CWinApp::PreTranslateMessage() retrieves the window handle of the application's main window by calling AfxGetMainWnd() (see List 6-1). Then it calls CWnd::WalkPreTranslateTree() that uses this handle to know when to stop calling PreTranslateMessage(pMsg). Starting inside-out (that is, starting from the target window), WalkPreTranslateTree() hits each window, trying to find one interested in the message. For each window in the tree, WalkPreTranslateTree() fetches the CWnd object from the window handle map (by CWnd::FromHandlePermanent() [List 5-10 in (III-3)]) and translates the message (by calling PreTranslateMessage(pMsg)) for each CWnd object. WalkPreTranslateTree() does this for each window until (1) the window handles the message or (2) WalkPreTranslateTree() reaches the application's main window. CWnd::PreTranslateMessage(pMsg) takes a single parameter: a pointer to the MSG structure.

6.3. Use or Not Use PreTranslateMessage()

Since CWinThread::PreTranslateMassage() is virtual, it is to be overridden by our own CWinApp::PreTranslateMessage() only in user-interface threads. When we override PreTranslateMessage(), we examine the MSG structure first. And if we are interested in the message, we handle (i.e. override) it within PreTranslateMessage() and return TRUE.

Generally speaking, we should not circumvent the normal dispatch path by using PreTranslateMessage() to handle any message sent to any window. For this sake we should use window procedures and MFC message maps. Although there are many cases in which

PreTranslateMessage() makes sense, we should use this only if the alternatives (i.e. the normal dispatch path) cannot accomplish the task[17]. CWinApp::PreTranslateMessage() returns TRUE when it finds a window interested in the message. If CWinApp::PreTranslateMessage() returns TRUE, then CWinApp's message pump does not dispatch the message. Therefore PreTranslateMessage() effectively "eats" the message.

Section 7 Conclusion

In a series of "Development of a Management Support System on the Windows Platform (III): Message Pumping and Message Handling" from Part 1 through Part4, we have studied the Windows message pumping and message handling mechanism, clarifying the difference between the Windows traditional message looping which relies on the WndProc() procedure and the MFC's message mapping architecture that hooks into the traditional message loop. The message maps associate specific commands and messages with the member functions of the CWnd objects that handle those commands and messages. At the heart of the message-mapping architecture lies the universal procedure AfxWndProc(). Although it is surely true that the MFC's hooking procedures are performed in our applications, their behaviors are hidden deep in the code of the MFC library.

In (III-1)[3] we have reviewed the Windows programming model where we grasped the basic idea of the Windows queue and the message loop that Windows adopts. The MFC class library provides another way of handling messages (i.e. the message mapping architecture) without relying on a set of switch statements and vtables which are assigned to the window procedure WndProc(). We took a close look at message map data structure and message macros MFC prepares, of which the structure is to become crucial in understanding the MFC message handling in later series of our articles. Finally we classified the messages into three message categories: regular window messages, command messages and control notifications.

In (III-2)[4] we first looked into the message handlers as the window procedures. Whenever a message is sent to a window, the message loop looks at the window's window class and calls the WndProc() passing the message's information. Next we examined another way of message handling, the MFC's message mapping architecture that associates command messages with a command target by command-routing and message-dispatching mechanism. The mechanism has led us to the concept of "subclassing" the window procedures which is realized by the hooking technique that enables an MFC CWnd object 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. It is _AfxCbtFilterHook() that MFC installs whenever a new CWnd(-derived) object is created so that the CWnd object can be hooked up and subclassed before any messages are delivered to that window. We found finally that once _AfxCbtFilterHook() is finished, the window is hooked up and messages for that window will all go to the universal AfxWndProc(). In other words, the framework effectively wires the windows up to AfxWndProc() whenever a CWnd-derived window is created.

In (III-3)[5] we further took a close look at the processes that AfxWndProc() goes through to handle the messages and commands that a window receives. Windows calls back AfxWndProc() with the handle to the focused window. We summarized here the parameters to be passed over by Windows to the window procedure. There are two ways to represent a window: by a unique system-defined window handle and by the CWnd(-derived) object representing the window. The native Windows code deals with window handles. The MFC library, on the other hand, is designed to work with in general CWnd objects. CWnd encapsulates all the Windows API functions that take window handles. In AfxWndProc() the framework retrieves the C++ object (i.e. CWnd(-derived) object) associated with the focused window by using CWnd::FromHandlePermanent() with the window handle. For the message dispatching to work, it is very important, therefore, to figure out which CWnd(-derived) target object is associated with a particular handle. In (III-3) we examined the real sequence of MFC message handling mechanism depending on a variety of CCmdTarget-derived objects, starting with the generic MFC's application framework procedure AfxWndProc(). We also found that the standard sequence of the MFC's message handling procedures differs from windows messages, to command messages and to command notifications.

Finally in (III-4), i.e. in the present article we have tried to totally understand the MFC's message handling processes on an equal footing by recognizing the variety of CCmdTarget-derived objects and the three categories of message types. We found that starting with MFC's AfxWndProc(), the standard sequence of the message handling procedures are neatly classified by the message categories specified by message ID's. MFC does switch, however, not on the message ID but on the signature of the message handler function. OnWndMsg() declares a MessageMapFunction union and retrieves the CWnd object's message map. Whatever commands or messages may be, AfxFindMessageEntry() eventually finds an entry of the message map that matches the signature of the message and walks the array of the message map entries in the message map. OnWndMsg() draws the message handler's address from the switch construction in the MessageMapFunction union with signature set to the switch variable. In the case of WM_COMMAND and WM_NOTIFY commands, the framework routes commands through a standard sequence of command-target objects, one of which is expected to have a handler for the command. Regular window messages, on the other hand, go straight to the window for which they are intended. The reason why the switch construction is evaluated not by message ID but by signature is that MFC packages WPARAM and LPARAM parameters into more type-safe objects and makes a call to the required handler function with those type-safe arguments. This conversion is carried out in CWnd::OnWndMsg() for regular window messages and OnCommand() or OnNotify() functions for commands or control notifications.

MFC has another powerful and useful feature in pumping messages. PreTransltaeMessage() is the approved way of altering the way of MFC's normal message dispatching. This is another hooking function that intercepts or filters window messages before they are dispatched to the Windows functions TranslateMessage() and DispatchMessage(). Although there are many cases in which PreTranslateMessage() makes sense, we should use this function only if the alternatives cannot accomplish the task.

In conclusion, MFC's standard message-mapping technology is a reasonable alternative to handling messages via virtual class member functions of the conventional Windows message looping. MFC's standard message-mapping technology eliminates the baggage of erroneous vtables. We should have a good grasp of how MFC handles the application (initialization and message pump) and window (message handling) aspects of a Windows application by looking into the inside of MFC's message filtering mechanism which is efficient and compiler independent but is hidden deep in the MFC's class library that itself is the wrapper of the huge legacy of API upon the Windows platform. We emphasize here again that it is equally important to realize that the message-routing code is universal for all window classes. MFC uses the same AfxWndProc() for all window objects.

Bibliography

(1) Noto, Hiroshi. 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, Hiroshi. 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, Hiroshi. 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, Hiroshi. 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) Noto, Hiroshi. Development of a Management Support System On the Windows Platform (III-Part 3): Message Pumping and Message Handling, Hokusei Review, The School of Economics (Hokusei Gakuen University) Vol. 45, No. 2, March 2006.

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

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

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

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

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

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

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

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

(14) Mehdi Mousavi. Win32 vs. MFC - Part II
http://www.codeproject.com/cpp/mfc_architecture2.asp

(15) MFC Library Reference TN062: Message Reflection for Windows Controls
http://msdn2.microsoft.com/en-us/eeah46xd(d=printer).aspx

(16) Alexander Fedorov. Message Reflection for WTL Controls
http://codeproject.com/wtl/wtl_message_reflection.asp

(17) Message APIs
http://www.flounder.com/messaging.htm

［Abstract］

Development of a Management Support System on the Windows Platform (III-Part 4): 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 with 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: 1) Command-Routing and Message-Dispatching Architecture
2) C++ with MFC (Microsoft Foundation Class) Library on Windows
3) WM_COMMAND in Commands and Control Notifications
4) Regular Window Messages
5) Hooking into Message Loop (PreTranslateMessage())

General Contents
of
Development of a Management Support System on the Windows Platform (III)