Development of a Management Support System on the Windows Platform（Ⅲ-Part 1）:

Message Pumping and Message Handling

Hiroshi NOTO

Contents

1. Introduction
2. Windows Programming Model
3. Message Map Data Structure and Message Map Macros
4. Windows Message Components and Message Type in Windows
5. How MFC Uses Message Maps and Handles Messages
6. Hooking into the Message Loop:PreTranslateMessage()
7. Conclusion

Section 1. Introduction

In our article series[1)-3)] of 'Development of a Management Support System（MSS）on the Windows Platform' we are studying Windows programming on the Windows operating system[4),5)] by Visual C++ language processor[6),11)-15)] with MFC library.[7)-10)] As an example of Windows programs we have taken an application we have recently developed which is called "Management Support System(MSS)". The first article subtitled 'Class structure of MFC and creation of user-defined classes' and the second article subtitled 'Registering Window Classes and Creating the Main Window'. The former article elaborated the class structure of our MSS by reviewing the hierarchical class structure of MFC after describing the design and the performance of our MSS. The latter one concentrated on the initialization processes of the Windows program and elucidated how and where the registration of window classes and the creation of the main window is carried out.

The purpose of the present article is to study the message pumping and the message handling procedures which are the core mechanisms both of the Windows operating system and of the C++ processor with the MFC library. A Windows application, therefore, written in C++ with MFC upon the Windows platform contains at least two distinct parts: a message pump and a message handling window procedure. MFC supports this basic Windows program structure by segregating Windows applications into two main pieces: a class representing the application and a class representing a window: (1) an application-specific part that handles initialization, creating one or more windows and sustaining a GetMessage()…DispatchMessage()

Key words : Windows message handling procedure, Message loop and message pumping（routing）, Windows Programming Model, Message mapping macros, MFC（Microsoft Foundation Class）class library

loop. (2) a window-specific part that takes care of drawing on the window, message handling, and so on. In MFC, these two main components are embodied within the CWinApp and the CWnd classes, respectively. In order to understand the Windows programming well from internals of MFC, we could not avoid figuring out the mechanism for the message pumping and the message handling.

In §2 the Windows programming model is viewed. We examine how the message handling is carried out in §3. Message Map Data Structure and Message Map Macros are explained in more detail. In §4 we represent Windows message components and message type in Windows. In §5 we work out how MFC actually uses Message Maps and handles messages. We very briefly summarize the hooking mechanism into the Message Loop in §6. The summary of the present article is stated §7.

Section 2. The Windows Programming Model

When (a) window(s) is (are) registered and created, the window(s) has (have) its (their) behavioral characteristics which are what the Windows Programming Model represents[9]. The Windows Programming Model provides the "object-oriented" approach where we create abstract data types. These abstract data types are commonly called "objects" and consist of a data structure and associated functions, commonly called" methods" that manipulate the data structure. In this model the basic object is a window and the methods are functions that handle certain notifications or announcements we can send it (the window) of various events occurred within the window. The core of the Windows Programming Model is to realize the event-driven programming mechanism in it. An event here could be a keystroke or a mouse click in a window, or a command for a window, e.g. to repaint the window, or a notification from user-interface objects: menus or toolbar buttons or the like. The information about the events is called a "message".

In the previous paragraph we stated in its meaning, "A message is sent to the window." Actually, Windows notifies a window when an event occurs that might affect the window. More precisely Windows calls the function associated with the window, passing the messages in the arguments of the call that describe the event that has occurred. However, a message is not sent directly without some "buffer" (as it were) from Windows to the window procedure that is assigned to the window where an event has occurred (see next subsection).

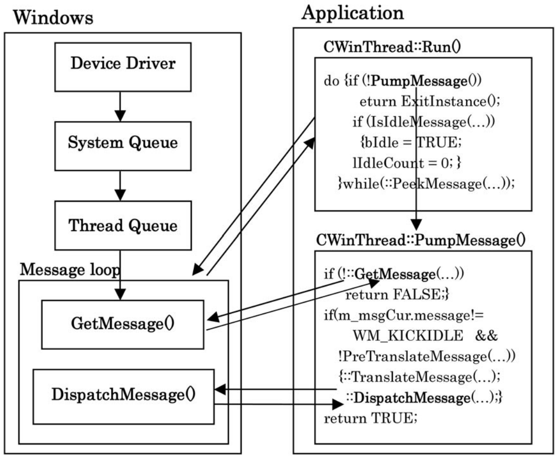
2.1. Windows Queues and the Message Loop

Many messages in Windows originate from devices. For example, depressing and releasing a key on the keyboard generates interrupts that are handled by the keyboard device driver. Moving the mouse and clicking the mouse buttons generate another interrupts that are handled by the mouse device driver. These device drivers call Windows to "translate" the hardware event into a message. The resulting message is then once placed into the Windows "system queue". There is only one system queue in Windows. Messages reside in the system queue only briefly. There is another queue that each running Windows application can have uniquely, called "thread queue". In Win32 (i.e. 32-bit Windows operating system), each running application can have multiple threads of execution, and each thread that is processing messages has its own unique thread queue. Windows transfers the messages in the system queue to the appropriate thread queue. Each thread queue, therefore, of a program holds all messages for all windows running in a particular thread.

Windows uses the "input focus" to decide which thread queue should receive the message. The input focus is an attribute possessed by only one window in the system at a time. The window with the input focus is the focal point for all keyboard input. Keyboard messages are moved from the system queue into the thread queue for the thread with a window that presently has the input focus. Mouse messages usually are sent to the window that is underneath the mouse pointer. When multiple windows are overlapped, the one on the top within the display receives the mouse message. The one exception to this rule involves "capturing" the mouse. When a Windows application captures the mouse (which is done by making a Windows function call), Windows moves all subsequent mouse messages from the system queue to the capturing window's thread queue, no matter where the mouse is pointing on the screen. The application must eventually release the captured mouse to allow other applications to use it.

When a program's thread queues are filling with messages, how does the program get the message from a thread queue and deliver it to the proper window function? We write a small piece of code called the "message loop". The message loop retrieves input messages from the application queue and dispatches them to the appropriate window functions. The message loop continually retrieves and dispatches messages until it retrieves a special message that signals that the loop should terminate (WM_QUIT message actually). One message loop is, as it were, the 'main body' of a Windows application: A Windows application initializes, repeatedly executes the message loop logic until instructed to stop, and then terminates.

A program calls the Windows GetMesssage() function to retrieve a message from its thread queue. Windows moves the message from the queue into a data area within the program. Now the program has the message. Then the message needs to be sent to the proper window function. To do this, the program calls the Windows DispatchMessage() function. Why must we call Windows to send a message to a window function within our program? A program may create more than one window; each window may have its own unique window function, or multiple windows may use the same window function. In addition, many of the window functions for window types provided by Windows are not in our program at all; they are inside Windows. The DispatchMessage() function in Windows hides all this complexity by determining which of the program's window functions or Window's built-in window functions gets the message. It (DispatchMessage() function) then calls the proper window function directly. Figure 2-1 shows the path that keyboard input takes all the



```
do {if (!PumpMessage())
      eturn ExitInstance();
    if (IsIdleMessage(...))
      {bIdle = TRUE;
    lIdleCount = 0; }
  }while(::PeekMessage(...));
```

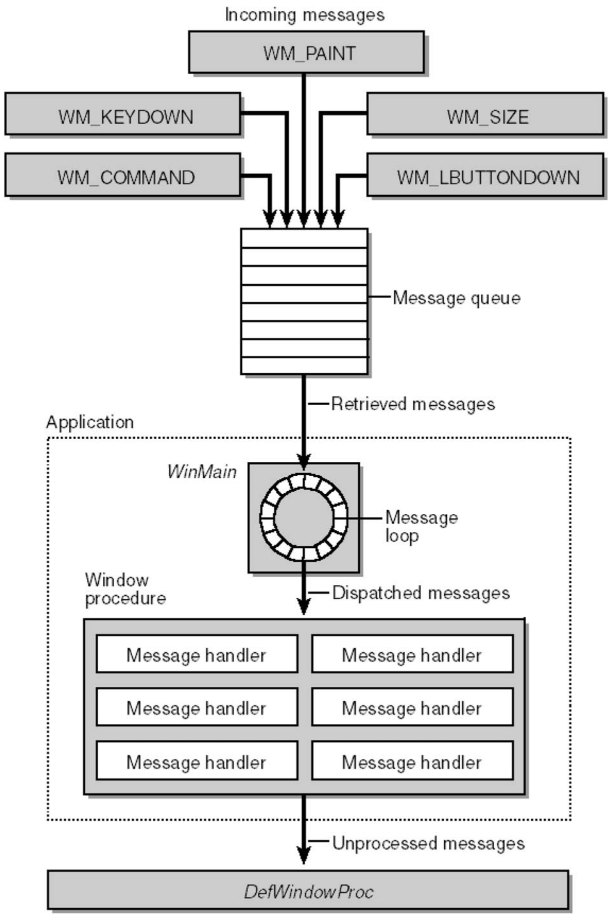
```
if (!::GetMessage(...))
    return FALSE;}
if(m_msgCur.message!=
     WM_KICKIDLE   &&
  !PreTranslateMessage(...))
  {::TranslateMessage(...);
   ::DispatchMessage(...);}
return TRUE;
```

Figure 2-1. Keyboard input to a single Windows application

way from an event-generation of a keystroke through the system queue, an application's thread queue(s), and to the relevant window function within the system.[4)]

2.2. Event-driven Programming Model

We illustrate the event-driven programming model in Figure 2-2, where applications respond to events by processing messages sent by Windows operating system. The entry point for a Windows program is a function named WinMain()*), but most of the action takes place in the window procedure. The window procedure or the method (sometimes thus called) processes messages sent to the window. WinMain() creates that window and then enters a message loop, alternately retrieving messages (GetMesssage() function) and dispatching them to the window procedure (DispatchMessage() function). Messages wait in a message queue (mostly in a thread queue) until they are retrieved. A typical Windows application performs the bulk of its processing in response to the messages it receives, and in between messages, it does little except wait for the next message to arrive.



Fiigure 2-2. The Windows prgramming mdel
Source: Aran R. Feuer. "MFC Programming, Addison Wesley, 1997.

* WinMain() was studied in detail in article (Ⅱ)

The message loop ends when a WM_QUIT message is retrieved from the message queue, signaling that it is time for the application to end. This message usually appears because the user selected Exit from the File menu, clicked the close button (the small button with an [x] in the window's upper right corner), or selected Close from the window's system menu. When the message loop ends, WinMain() returns and the application terminates.

Here we very briefly refer to the message types and the message form. Further details of the message types and forms are viewed later in §4.

The type of a message

Windows defines hundreds of different message types. Most messages have names that begin with the letters "WM" and an underscore, as in WM_CREATE and WM_PAINT. These messages can be classified in various ways, but for the moment classification is not nearly as important as realizing the critical role messages play in the operation of an application. The following table shows 10 of the most common messages. Some of them are already seen so far. A window receives a WM_PAINT message, for example, when its interior needs repainting. One way to characterize a Windows program is to think of it as a collection of message handlers. To a large extent, it is a program's unique way of responding to messages that gives the program its personality. In Table 2-1 we present most common Windows messages.

Table 2-1. Most common Windows messages

	Message Sent When
WM_CHAR	A character is input from the keyboard.
WM_COMMAND	The user selects an item from a menu, or a control sends a notification to its parent.
WM_CREATE	A window is created.
WM_DESTROY	A window is destroyed.
WM_LBUTTONDOWN	The left mouse button is pressed.
WM_LBUTTONUP	The left mouse button is released.
WM_MOUSEMOVE	The mouse pointer is moved.
WM_PAINT	A window needs repainting.
WM_QUIT	The application is about to terminate.
WM_SIZE	A window is resized.

The form of a message

A message manifests itself in the form of a call to a Windows' window procedure. Bundled with the call are four input parameters: the handle of the window to which the message is directed, a message ID, and two 32-bit parameters known as wParam and lParam. The window handle is a 32-bit value that uniquely identifies a window. Internally, the value refers to a data structure in which Windows stores relevant information about the window such as its size, style, and location on the screen. The message ID is a numeric value that identifies the message type: WM_CREATE, WM_PAINT, and so on. wParam and lParam contain information specific to the message type. When a WM_LBUTTONDOWN message arrives, for example, wParam holds a series of bit flags identifying the state of the Ctrl and Shift keys and of the mouse buttons. lParam holds two 16-bit values identifying the location of the mouse pointer when the click occurred.

Together, these parameters provide the window procedure with all the information it needs to process the WM_LBUTTONDOWN message.

2.3. Message Loop in the MSS Application

Here we try to trace where and how the message pumping mechanism works by invoking our own MSS which boosts and deals with the message loop, responding to events generated in window objects. The Windows application sets going AfxWinMain() function which is automatically called when we invoke the MSS. As soon as we start to use Multiple Document Templates of the MSS, MFC pops up a new dialog box. The application proceeds after we select one of the pop-up menu commands, for example, "describe" in the dialog box where the five Document Templates are featured. This selection continues processing AfxWinMain() function in WINMAIN.CPP and therein we reach pThread->Run() as seen in List 2-1.

List 2-1. AfxWinMain() in WINMAIN.CPP

```
int AFXAPI AfxWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPTSTR lpCmdLine, int nCmdShow)
{
        ASSERT(hPrevInstance == NULL);

        int nReturnCode = -1;
        CWinThread* pThread = AfxGetThread();
        CWinApp* pApp = AfxGetApp();

        // AFX internal initialization
        if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow))
                goto InitFailure;

        // App global initializations (rare)
        if (pApp != NULL && !pApp->InitApplication())
                goto InitFailure;

        // Perform specific initializations
        if (!pThread->InitInstance())
        {
                if (pThread->m_pMainWnd != NULL)
                {
                        TRACE0("Warning: Destroying non-NULL m_pMainWnd¥n");
                        pThread->m_pMainWnd->DestroyWindow();
                }
                nReturnCode = pThread->ExitInstance();
                goto InitFailure;
        }
        nReturnCode = pThread->Run();

InitFailure:
#ifdef _DEBUG
        // Check for missing AfxLockTempMap calls
        if (AfxGetModuleThreadState()->m_nTempMapLock != 0)
        {
                TRACE1("Warning: Temp map lock count non-zero (%ld).¥n",
                        AfxGetModuleThreadState()->m_nTempMapLock);
        }
        AfxLockTempMaps();
        AfxUnlockTempMaps(-1);
#endif

        AfxWinTerm();
        return nReturnCode;
}
```

Then pThread->Run() in AfxWinMain() is overridden by CWinApp::Run() in APPCORE.CPP. And CWinApp::Run() at the end of the function returns CWinThread::Run() in THRDCORE.CPP as shown in List 2-2.

List 2-2. CWinApp::Run() in APPCORE.CPP

```
// Main running routine until application exits
int CWinApp::Run()
{
        if (m_pMainWnd == NULL && AfxOleGetUserCtrl())
        {
                // Not launched /Embedding or /Automation, but has no main window!
                TRACE0("Warning: m_pMainWnd is NULL in CWinApp::Run - quitting
application.¥n");
                AfxPostQuitMessage(0);
        }
        return CWinThread::Run();
}
```

In CWinThread::Run(), as you can see in List 2-3, contained is a "do-while" message loop developed with MFC with an evaluation of a ::PeekMessage() Win32 API. As long as the ::PeekMessage() function returns a nonzero value, the loop calls PumpMessage() (i.e. **CWinThread::PumpMessage**) to perform normal message translation and dispatching until a WM_QUIT message is received.

List 2-3. CWinThread::Run() in THRDCORE.CPP

```
// main running routine until thread exits
int CWinThread::Run()
{
        ASSERT_VALID(this);

        // for tracking the idle time state
        BOOL bIdle = TRUE;
        LONG lIdleCount = 0;

        // acquire and dispatch messages until a WM_QUIT message is received.
        for (;;)
        {
                // phase1: check to see if we can do idle work
                while (bIdle &&
                        !::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE))
                {
                        // call OnIdle while in bIdle state
                        if (!OnIdle(lIdleCount++))
                                bIdle = FALSE; // assume "no idle" state
                }

                // phase2: pump messages while available
                do
                {
                        // pump message, but quit on WM_QUIT
                        if (!PumpMessage())
                                return ExitInstance();

                        // reset "no idle" state after pumping "normal" message
                        if (IsIdleMessage(&m_msgCur))
                        {
                                bIdle = TRUE;
                                lIdleCount = 0;
                        }
                } while (::PeekMessage(&m_msgCur, NULL, NULL, NULL,
PM_NOREMOVE));
        }

        ASSERT(FALSE); // not reachable
}
```

Although PumpMessage is undocumented, we can examine its source code in the THRDCORE.CPP file in List 2-4.

List 2-4. CWinThread::PumpMessage() in THRDCORE.CPP

```
// CWinThread implementation helpers
BOOL CWinThread::PumpMessage()
{
        ASSERT_VALID(this);

        if (!::GetMessage(&m_msgCur, NULL, NULL, NULL))
        {
#ifdef _DEBUG
                if (afxTraceFlags & traceAppMsg)
                        TRACE0("CWinThread::PumpMessage - Received WM_QUIT.¥n");
                m_nDisablePumpCount++; // application must die
                        // Note: prevents calling message loop things in 'ExitInstance'
                        // will never be decremented
#endif
                return FALSE;
        }

#ifdef _DEBUG
        if (m_nDisablePumpCount != 0)
        {
                TRACE0("Error: CWinThread::PumpMessage called when not permitted.¥n");
                ASSERT(FALSE);
        }
#endif

#ifdef _DEBUG
        if (afxTraceFlags & traceAppMsg)
                _AfxTraceMsg(_T("PumpMessage"), &m_msgCur);
#endif

        // process this message

        if (m_msgCur.message != WM_KICKIDLE && !PreTranslateMessage(&m_msgCur))
        {
                ::TranslateMessage(&m_msgCur);
                ::DispatchMessage(&m_msgCur);
        }
        return TRUE;
}
```

More specifically, the command PumpMessage() calls CWinThread::PumpMessage() in THRDCORE.CPP as shown in the List. In this function a set of the message pumping procedures(::GetMessage(), ::TranslateMessage(), and ::DispatchMessage()) are contained in the message loop.

Section 3. Message Map Data Structure and Message Map Macros

So far, we have seen the event-driven Windows programming mechanism in which an event occurred within a window is notified to the window as a message. The message this time is dispatched to the appropriate message handler or method which is a function to process the message responding to the event of the window. Every window class within a Windows application requires a message-handling procedure. Whenever Windows detects an event that is pertinent to a specific window, it generates a message and calls the window's message handler with information about the event.

In this section we consider how we can use C++ classes to handle Windows messages and look closely into the message-handling architecture specifically in MFC.[8)]

In Windows a message routing and handling system called "message map" is implemented. At the highest level, message maps simply associate window messages and commands to a class's member functions. MFC's message mapping technology is made up of two parts: (1) the CCmdTarget class and (2) message maps. The CCmdTarget class is the base class for any object that needs to receive window messages, commands, or both. In spite of the name "message map", message maps handle both messages and commands; all three categories of messages are elaborated in the next section as Message Categories. Message map data structure and message map macros are two other important aspects of the message-mapping system.

3.1 CCmdTarget

To be able to receive messages in an object, the class it belongs to must be derived from CCmdTarget. CCmdTarget-derived classes have the machinery necessary to deal with message maps. Any class derived from CCmdTarget can use a message map.

3.2 Message Map Data Structure

Let us take a peek at two data structures used to implement MFC's message mapping. The first is AFX_MSGMAP_ENTRY. This structure represents the actual entries into the message map table.

```
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)
};
```

Here the UINT type is a typedef for an unsigned int. A UINT variable is a 32-bit unsigned integer in Win32 application. The first field(nMessage) indicates the Windows message coming through the system. The second field(nCode) represents the control code or the WM_NOTIFY code. The third field(nID) refers to the control ID generating the message. Parameter number four (nLastID) is used for entries specifying a range of control identifiers. The nSig parameter indicates the signature of the function to handle the message. The last parameter(pfn) points to the routine handling the message.

The second structure is AFX_MSGMAP. The AFX_MSGMAP structure represents the actual message map:

```
struct AFX_MSGMAP
{
#ifdef _AFXDLL
        const AFX_MSGMAP* (PASCAL* pfnGetBaseMap)();
#else
        const AFX_MSGMAP* pBaseMap;
#endif
        const AFX_MSGMAP_ENTRY* lpEntries;
};
```

This structure has two parts: (1) a pointer to another AFX_MSGMAP structure(in practice, the base class's message map) and (2) an array of AFX_MSGMAP_ENTRY structures. It should be noticed how this structure is set up to be included in a linked list. A message map is basically an array of AFX_MSGMAP_ENTRY structure. Each class hierarchy used within an application maintains a linked list of message maps. This is how MFC implements inheritance using message maps. Basically, the framework walks the message maps back to the root class until it finds a function to handle the message.

Dynamic Processing of WM_COMMAND Messages

The message-map mechanism provided by the Microsoft Foundation Classes (MFC) can process WM_COMMAND messages for a constant ID. However, in some cases, an application needs to process WM_COMMAND messages for an ID that is not known until run time. This can occur when an application modifies menus or dynamically creates controls at run time. To process these messages, our application must override the CCmdTarget::OnCmdMsg() function.

3.3 Message Map Macros

MFC provides three macros to generate message maps: DECLARE_MESSAGE_MAP, BEGIN_MESSAGE_MAP, and END_MESSAGE_MAP. These macros expand into code that defines and implements a message map for a CCmdTarget-based class. When using message maps in our classes, the basic strategy is to include DECLARE_MESSAGE_MAP in our class definition, i.e. in our H file and then add BEGIN_MESSAGE_MAP, END_MESSAGE_MAP, and message-mapping information to our implementation file (CPP file). DECLARE_MESSAGE_MAP is declared in AFXWIN.H and looks like this (in List 3-1):

List 3-1. DECLARE_MESSAGE_MAP in AFXWIN.H

```
#ifdef _AFXDLL
#define DECLARE_MESSAGE_MAP()
private:
        static const AFX_MSGMAP_ENTRY _messageEntries[];
protected:
        static AFX_DATA const AFX_MSGMAP messageMap;
        static const AFX_MSGMAP* PASCAL _GetBaseMessageMap();
        virtual const AFX_MSGMAP* GetMessageMap() const;

#else
#define DECLARE_MESSAGE_MAP()
private:
        static const AFX_MSGMAP_ENTRY _messageEntries[];
protected:
        static AFX_DATA const AFX_MSGMAP messageMap;
        virtual const AFX_MSGMAP* GetMessageMap() const;

#endif
```

Here a symbol "_AFXDLL" is one of the DLL-related preprocessor symbols which means that the DLL contains MFC code and links to the shared MFC runtime. Using DECLARE_MESSAGE_MAP in a class declaration defines three things for the class: (1) an array of AFX_MSGMAP_ENTRY structure called _messageEntries[], (2) an AFX_MSGMAP structure called messageMap, and (3) a function to retrieve the class's message map (GetMessageMap()). Note that the message map entries (_messageEntries[]) and the message map structure (messageMap) are static members of the class. This means that there is one _ messageEntries array and one messageMap member for all objects within the class.

Here is a CWinApp-derived class CMSSApp in MSS.H which is the framework class of our MSS[2]. CMSSApp includes a message map where the macro DECLARE_MESSAGE_MAP is inserted. The preprocessor uses the message macros to generate message-mapping support code.

List 3-2. Macro DECLARE_MESSAGE_MAP in CMSSApp in MSS.H

```
// MSS.h : main header file for the MSS application
//
#if !defined(AFX_MSS_H__39C8F225_A7EB_11D3_9C1E_00000E49332F__INCLUDED_)
#define AFX_MSS_H__39C8F225_A7EB_11D3_9C1E_00000E49332F__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
```

```
#ifndef __AFXWIN_H__
        #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols

/////////////////////////////////////////////////////////////////////////////
// CMSSApp:
// See MSS.cpp for implementation of this class.
//

class CMSSApp : public CWinApp
{
public:
        CMSSApp();
//          CString m_vc_d;
// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CMSSApp)
        public:
        virtual BOOL InitInstance();
        //}}AFX_VIRTUAL

// Implementation

        //{{AFX_MSG(CMSSApp)
        afx_msg void OnAppAbout();
        afx_msg void OnCommonToVar();
        afx_msg void OnVarToModel();
        afx_msg void OnComScenarioToDatafile();
        afx_msg void OnRunExecute();
        afx_msg void OnGoalseekExecute();
        afx_msg void OnTeirituHou();
        afx_msg void OnCfin1Read();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous
line.

#endif // !defined(AFX_MSS_H__39C8F225_A7EB_11D3_9C1E_00000E49332F__INCLUDED_)
```

There are two more macros that finish the job: BEGIN_MESSAGE_MAP and END_MESSAGE_MAP. As the name implies, the macro begins a message map. The macro definition, as found in AFXWIN.H, looks like this:

List 3-3. BEGIN_MESSAGE_MAP() and END_MESSAGE_MAP() in AFXWIN.H

```
#ifdef _AFXDLL
#define BEGIN_MESSAGE_MAP(theClass, baseClass)
        const AFX_MSGMAP* PASCAL theClass::_GetBaseMessageMap()
                { return &baseClass::messageMap; }
        const AFX_MSGMAP* theClass::GetMessageMap() const
```

```
                { return &theClass::messageMap; }
        AFX_COMDAT AFX_DATADEF const AFX_MSGMAP theClass::messageMap =
        { &theClass::_GetBaseMessageMap, &theClass::_messageEntries[0] };
        AFX_COMDAT const AFX_MSGMAP_ENTRY theClass::_messageEntries[] =
        {

#else
#define BEGIN_MESSAGE_MAP(theClass, baseClass)
        const AFX_MSGMAP* theClass::GetMessageMap() const
                { return &theClass::messageMap; }
        AFX_COMDAT AFX_DATADEF const AFX_MSGMAP theClass::messageMap =
        { &baseClass::messageMap, &theClass::_messageEntries[0] };
        AFX_COMDAT const AFX_MSGMAP_ENTRY theClass::_messageEntries[] =
        {

#endif
```

When used together in an implemented file (.CPP file), these macros actually implement the message map. Here is again an example of the implementation of a CWinApp-derived class CMSSApp in MSS.CPP where inserted are macros BEGIN_MESSAGE_MAP and END_MESSAGE_MAP. In MSS.CPP each macro is expanded into its defined code.[2)]

List 3-4. Macros BEGIN_MESSAGE_MAP and END_MESSAGE_MAP in CMSSApp class in MSS.CPP

```
// MSS.cpp : Defines the class behaviors for the application.
//
#include "stdafx.h"
#include "MSS.h"
#include "MainFrm.h"
#include "ChildFrm.h"
#include "MSSDoc.h"
#include "MSS1Doc.h"
#include "MSS2Doc.h"
#include "MSSView.h"
#include "MSS1View.h"
#include "MSS2View.h"
#include "MSS3View.h"
#include "MSS4View.h"
#include "MSS4Doc.h"

#include "VarDialog.h"
#include "SelectDlg.h"
#include "SetGoalDlg.h"
#include "FixedRateDlg.h"
#include "WzdSplash.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMSSApp

BEGIN_MESSAGE_MAP(CMSSApp, CWinApp)
        //{{AFX_MSG_MAP(CMSSApp)
                // NOTE - the ClassWizard will add and remove mapping macros here.
                // DO NOT EDIT what you see in these blocks of generated code!
        ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
        ON_COMMAND(IDM_COMMON_TO_VAR, OnCommonToVar)
        ON_COMMAND(IDM_VAR_TO_MODEL, OnVarToModel)
        ON_COMMAND(IDM_COM_SCENARIO_TO_DATAFILE, OnComScenarioToDatafile)
```

```
          ON_COMMAND(IDM_RUN_EXECUTE, OnRunExecute)
          ON_COMMAND(IDM_GOALSEEK_EXECUTE, OnGoalseekExecute)
          ON_COMMAND(IDM_TEIRITU_HOU, OnTeirituHou)
          ON_COMMAND(IDM_Cfin1_Read, OnCfin1Read)
          //}}AFX_MSG_MAP
          // Standard file based document commands
          ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
          ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
          // Standard print setup command
          ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMSSApp construction
CMSSApp::CMSSApp()
{
          // TODO: add construction code here,
          // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CMSSApp object

CMSSApp static theApp;

/////////////////////////////////////////////////////////////////////////////
// CMSSApp initialization
```

First, the macros generate a GetMessageMap() function for the class. This function simply returns a reference to the class's message map. GetMessageMap() is used by the framework to retrieve the class's message map whenever it needs to. The macro then generates code that fills the class's AFX_MSGMAP structure. The first field points to the base class's message map (in this case the CWinApp's message map), creating a linked list all the way back to the root object. This allows the framework to check each of the base classes for a message handler whenever a specific message cannot be found. The linked list shows how MFC implements inheritance in message maps. The second field is set to point to the first message map entry for the class itself.

Finally, the macro generates the actual message table. Remember, the message map is simply a table of AFX_MSGMAP_ENTRY structures. Notice the additional macros enclosed within the message map. Sandwiched between the BEGIN_MESSAGE_MAP and END_MESSAGE_MAP macros are a set of these message entry macros. There are a number of these listed with MFC documentation. They all begin with "ON_…" These macros map Windows messages to their specific message handlers. The macros expand to fill the_messageEntries array for the class's message map. MFC defines various message map entry macros, which are listed in Table 3-1.

Table 3-1. MFC's message map entry macros

Message Type	Macro Form	Arguments
Predefined Windows Messages	ON_WM_XXXX	None
Commands	ON_COMMAND	Command ID, Handler name
Update commands	ON_UPDATE_COMMAND_UI	Command ID, Handler name
Control notifications	ON_XXXX	Control ID, Handler name
User-defined message	ON_MESSAGE	User-defined message ID, Handler name
Registered Windows message	ON_REGISTERED_MESSAGE	Registered message ID variable, Handler name
Range of command IDs	ON_COMMAND_RANGE	Start and end of a contiguous range of command IDs
Range of command IDs for updating	ON_UPDATE_COMMAND_UI_RANGE	Start and end of a contiguous range of command IDs
Range of control IDs	ON_CONTROL_RANGE	A control-notification code and the start and end of a contiguous range of command IDs

In the specific example just given in List 3-4, the message map uses the command type macro, "ON_COMMAND" which is found in AFXMSG_.H(List 3-5). In List 3-5 we also show a predefined Windows message macro, "ON_WM_LBUTTONDBLCLK" which responds to an event of "left mouse button double clicks".

List 3-5. ON_COMMAND macro and ON_WM_LBUTTONDBLCLK defined in AFXMSG_.H.

```
#define ON_COMMAND(id, memberFxn)
{ WM_COMMAND,
CN_COMMAND,
(WORD)id,
(WORD)id,
AfxSig_vv,
(AFX_PMSG)&memberFxn },

#define ON_WM_LBUTTONDBLCLK()
        { WM_LBUTTONDBLCLK,
0,
0,
0,
AfxSig_vwp,
                    (AFX_PMSG)(AFX_PMSGW)
(void (AFX_MSG_CALL CWnd::*)(UINT, CPoint))&OnLButtonDblClk },
```

Now go back and take a quick look at the AFX_MSGMAP_ENTRY structure. Let us notice how the macro neatly fills the structure. The signature value at the fifth column is used by the framework to signify the return type and the parameters of the message handling function. We examine the signature more in detail in §5.

Section 4. Windows Message Components and Message Type in Windows

In this section we are going to view the components of the Windows message in general and after that to group Windows messages into three categories.[16] There are three main categories: (1) Windows messages, (2) Control Notification Messages, and (3) Command Messages. Messages in the first two categories - Windows messages and control notifications - are handled by windows: objects of classes derived from class CWnd. This includes CFrameWnd, CMDIFrameWnd, CMDIChildWnd, CView, CDialog, and our own classes derived from these base classes. Such objects encapsulate an HWND, a handle to a Windows window. Those three message categories are summarized below.

4.1 Components of Windows Messages

Windows is an event-driven operating environment: Windows on the computer is watching the hardware for events. Whenever something interesting happens, Windows detects the event and passes that information to the applications it is hosting. The information about the events is called a message. All window messages have the same basic form. A window message has three components: (1)an unsigned integer containing the actual message,(2)WPARAM a word-size parameter(WPARAM is 32 bits in Win32), and (3)LPARAM a four-byte parameter. The form of a message is already briefly introduced in §2.

Windows has no knowledge about specific cases for each message. This means that each message must exhibit polymorphic behavior. In C++, "polymorphism" signifies the concept of a single interface for multiple functions. A window message has a single interface that has many different behaviors corresponding to each event in the window.

Of the three components of a window message, the unsigned integer always refers to the actual message. However, the WPARAM and the LPARAM can mean a variety of different things. In fact, the LPARAM component of a window message often contains additional data or points to a data structure required to handle the message. Take a typical message like WM_COMMAND. WM_COMMAND is sent to a window whenever (1) a menu item is selected. (2) a control sends a notification code to its parent window, or (3) an accelerated keystroke is translated. The low word (the low-order 16 bits) of the WPARAM parameter represents a number identifying the control. The high word (the high-order 16 bits) of the WPARAM is the notification code, indicating things like whether the user double-clicking on the control. The LPARAM represents the window handle of the control sending the message. It can get rather difficult trying to keep track of the messages and what the parameters mean.

We summarize the message data structure (MSG) form in Figure 4-1 with short remarks assigned to each member. Additional members, "time" and "pt" are also explained.[16]

Figure 4-1. the message data structure (MSG) form

```
The MSG structure has the following form:

typedef struct tagMSG {     // msg
   HWND   hwnd;
   UINT   message;
   WPARAM wParam;
   LPARAM lParam;
```

```
    DWORD  time;
    POINT  pt;
} MSG;
```

The MSG structure contains message information from a thread's message queue.

Members

hwnd: Identifies the window whose window procedure receives the message.

message: Specifies the message number.

wParam: Specifies additional information about the message. The exact meaning depends on the value of the message member.

lParam: Specifies additional information about the message. The exact meaning depends on the value of the message member.

time: Specifies the time at which the message was posted.

pt: Specifies the cursor position, in screen coordinates, when the message was posted.

4.2 Three Message Categories

(1)**Windows messages**

This includes primarily those messages beginning with the **WM_** prefix, except for **WM_COMMAND.** Windows messages are handled by windows and views (i.e. rendering or representing data). These messages often have parameters that are used in determining how to handle the message. A 16-bit value identifying the message. Constants corresponding to all message values are provided through WINUSER.H and begin with the **WM_** (which stands for "window message") prefix. For example, a mouse-button event message might be identified by the constant **WM_LBUTTON_DOWN** (left button pressed), which is the value "0 x 0201".

(2)**Control notifications**

This includes **WM_COMMAND** notification messages from controls and other child windows to their parent windows. For example, an edit control sends its parent a **WM_COMMAND** message containing the **EN_CHANGE** control-notification code when the user has taken an action that may have altered text in the edit control. The window's handler for the message responds to the notification message in some appropriate way, such as updating the text or retrieving the text in the control. The Windows Common Controls make use of the more powerful **WM_NOTIFY** for complex control notifications. This version of MFC has direct support for this new message with the **ON_NOTIFY** and **ON_NOTIFY_RANGE** macros. The framework routes control-notification messages like other **WM_** messages. One exception, however, is the **BN_CLICKED** control-notification message sent by buttons when the user clicks them. This message is treated specially as a command message and routed like other commands

(3)**Command messages**

This includes **WM_COMMAND** notification messages from user-interface objects: menus, toolbar buttons, and accelerator keys. The framework processes commands differently from other messages, and they can be handled by more kinds of objects. Command messages can be handled by a wider variety of objects:

documents, document templates, and the application object itself in addition to windows and views. When a command directly affects some particular object, it makes sense to have that object handle the command. For example, the Open command on the File menu is logically associated with the application: the application opens a specified document upon receiving the command(**ID_FILE_OPEN**). So the handler for the Open command is a member function of the application class(OnFileOpen()). Menu items, toolbar buttons, and accelerator keys are "user-interface objects" capable of generating commands. Each such user-interface object has an ID. We associate a user-interface object with a command by assigning the same ID to the object as the command. As explained in Messages, commands are implemented as special messages. Figure 4-2 "Commands in the Framework" below shows how the framework manages commands. When a user-interface object generates a command, such as **ID_FILE_OPEN,** one of the objects in our application handles the command—in the figure below, the application object's OnFileOpen() function is called via the application's message map.

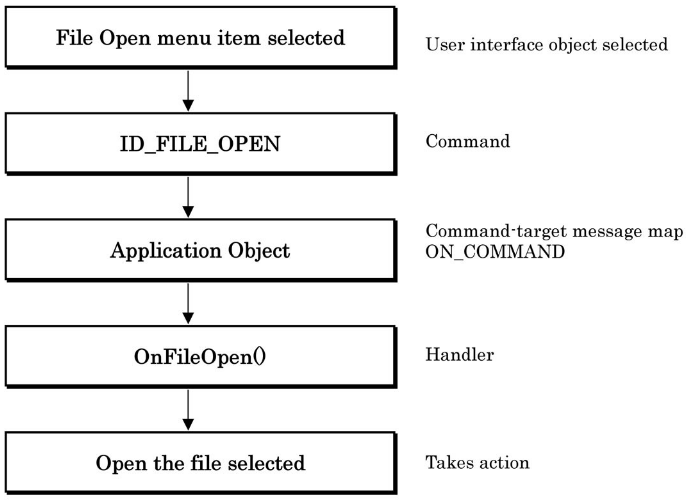


Figure 4-2. Commands in the Framework

[BIBLIOGRAPHY]

1) Noto, Hirosi. Interface Reflecting a Hybrid of the Decision Support System and the Expert System, Hokusei Review, The School of Economics (Hokusei Gakuen University) Vol. 37, 2000.

2) 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.

3) 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

[Abstract]

Development of a Management Support System on the Windows Platform（Ⅲ- Part 1）: Message Pumping and Message Handling

Hiroshi NOTO

We have studied 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. It（MFC's standard message-mapping）eliminates the overhead of erroneous vtables（virtual function tables）, it is compiler independent and it is fairly efficient. We should 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 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：Windows message handling procedure, Message loop and message pumping（routing）, Windows Programming Model, Message mapping macros, MFC（Microsoft Foundation Class）class library

正　誤　表

北星論集 経済学部 第44巻第2号（通巻第47号）

頁・行目	誤	正
4頁 上から1行目	an	(an)
4頁 下から3行目	Fiigure 2-2. The Windows prgramming model	Figure 2-2. Windows Programming Model
4頁 下から2行目	Programming,	Programming",
4頁 下から1行目	(Ⅱ)	(Ⅱ).
8頁 下から2行目	In this function	Through this function
11頁 下から5行目	_ messageEntries	_messageEntries
12頁 下から3行目	begins a message map.	beging and ends a message map.
13頁 上から15行目に追加	#define END_MESSAGE_MAP() {0,0,0,0,AfxSig_end,(AFX_PMSG)0} };	
14頁 下から2行目	the_messageEntries	the _messageEntries
15頁 下から20行目	ON_WM_LBUTTONDBLCLK defined	ON_WM_LBUTTONDBLCLK macro defined
17頁 上から5行目	A 16-bit value	UINT message is a 16-bit value
17頁 上から8行目	"0 x 0201"	"0x0201"
17頁 下から5行目	commands	commands.