Development of a Management Support System on the Windows Platform II
Registering Window Classes and Creating the Main Window

Hiroshi NOTO

Contents
1. Introduction
2. Two Classes Relevant to Initialization
3. Initialization
4. Registering Window Classes
5. Creating the Main Window
6. Summary

1. Introduction

The purpose of this article is to examine the inside of the Windows application i.e. the mechanism of the Windows program when it is started on the Windows operating system. We adopt here the Microsoft's Visual C++1 on Windows as a language processor. Visual C++ incorporates the MFC library2 into its own processing program. MFC is the C++ class library which provides an object-oriented wrapper around the Windows API [Application Programming Interface]3. Whereas the introduction of MFC simplifies a full-featured and robust application development and enables us to fully exploit Windows architecture, the mechanism of the Windows program has become hidden deep in the MFC library. In order to understand the Windows programming, therefore, it is instructive and effective to look into how Windows programs work and where in the programming codes Windows characteristic processes are executed.

As an example of the Windows application we take the Management Support System (MSS) program4, which we have recently developed in Seminar I and Seminar II for Juniors and Seniors, respectively I am in charge of in the Hokusei Gakuen University. The MSS program itself, however, is not the very first code that Windows executes when we start the application MSS. The Windows application generally requires a set of standard initializations to be done before the program really runs. Typically, these processes involve registering one or more window classes and creating one or more windows. Therefore in this article we concentrate on the initialization processes of the Windows programming and how and where they are realized in our MSS program.

Key words: Object-oriented programming, Windows programming, MFC class library, Initialization, Window classes registration

- 63 -
In Chapter 2 we first briefly recapitulate two classes in MFC library which are relevant to initialization procedures. In Chapter 3 we view the initialization of the Windows programming in general. We elucidate the process of registering window classes in Chapter 4. In Chapter 5 we clarify how and where the Windows program creates the main window. The summary of the present article is stated in Chapter 6.

2. Two Classes Relevant to Initialization

In this chapter we recap two classes CWinApp and AFX_MODULE_STATE in MFC just briefly within the scope of the initialization of a Windows application and the registration of window classes. Although AFX_MODULE_STATE is not given a name beginning with "C", which usually implies a class, it sure possesses the entity of a class in MFC. The two classes are well referred to in the following chapters and play important roles in initialization and registration.

2.1. CWinApp class in MFC

Windows applications contain the following two distinct parts: an application-specific part that handles initialization, creating one or more windows, and sustaining a message pump. DispatchMessage loop. A window-specific part that takes care of drawing on the window, message handling, and so on. In MFC the former function is embodied within the CWinApp as a class representing the application, and the latter function is embodied within the CWnd as a class representing a window.

CWinApp comprise a vast number of member functions and variables which are defined in AFXWIN.H. The class definition of CWinApp is shown in List 2-1.

List 2-1. CWinApp in AFXWIN.H

```c
class CWinApp : public CWinThread
{
    DECLARE_DYNAMIC(CWinApp)
public:
    // Constructor
    CWinApp(LPCTSTR lpszAppName = NULL); // app name defaults to EXE name
    // Attributes
    // Startup args (do not change)
    HINSTANCE m_hInstance;
    HINSTANCE m_hPrevInstance;
    LPTSTR m_lpCmdLine;
    int m_nCmdShow;
    // Running args (can be changed in InitInstance)
    LPCTSTR m_pszAppName; // human readable name
    // (from constructor or AFX_IDS_APP_TITLE)
    &LPCTSTR m_pszRegistryKey; // used for registry entries
    CDocManager* m_pDocManager;

    // Support for Shift+F1 help mode.
```

BOOL m_bHelpMode;    // are we in Shift+F1 mode?

public:     // set in constructor to override default
    LPCTSTR m_pszExeName;    // executable name (no spaces)
    LPCTSTR m_pszHelpFilePath; // default based on module path
    LPCTSTR m_pszProfileName; // default based on app name

    // Initialization Operations - should be done in InitInstance

    ..........
    ...........

public:     // Helper Operations - usually done in InitInstance
    // wrappers for Cursors
    // wrappers for Icons
    // Profile settings (to the app specific .INI file, or registry)

    // Running Operations - to be done on a running application
    // Dealing with document templates
    void AddDocTemplate(CDocTemplate* pTemplate);
    POSITION GetFirstDocTemplatePosition() const;
    CDocTemplate* GetNextDocTemplate(POSITION& pos) const;

    // Dealing with files
    virtual CDocument* OpenDocumentFile(LPCTSTR lpszFileName);  // open named file
    virtual void AddToRecentFileList(LPCTSTR lpszPathName);     // add to MRU

    // Printer DC Setup routine, 'struct tagPD' is a PRINTDLG structure
    // Command line parsing
    BOOL RunEmbedded();
    BOOL RunAutomated();
    void ParseCommandLine(CCommandLineInfo& rCmdInfo);
    BOOL ProcessShellCommand(CCommandLineInfo& rCmdInfo);

    // Overridables
    // hooks for your initialization code

    virtual BOOL InitApplication();

    // Functions for exiting

    // Advanced: to override message boxes and other hooks

    // Advanced: Help support

    // Command Handlers
    protected:
    // map to the following for file new/open
    afx_msg void OnFileNew();
    afx_msg void OnFileOpen();

    // map to the following to enable print setup
    afx_msg void OnFilePrintSetup();
// map to the following to enable help

// Implementation
protected:
HGLOBAL m_hDevMode; // printer Dev Mode
HGLOBAL m_hDevNames; // printer Device Names
DWORD m_dwPromptContext; // help context override for message box

int m_nWaitCursorCount; // for wait cursor (>0 => waiting)
HCURSOR m_hcurWaitCursorRestore; // old cursor to restore after wait cursor
CRecentFileList* m_pRecentFileList;

void UpdatePrinterSelection(BOOL bForceDefaults);
void SaveStdProfileSettings(); // save options to .INI file

public: // public for implementation access
CCommandLineInfo* m_pCmdInfo;

ATOM m_atomApp, m_atomSystemTopic; // for DDE open
UINT m_nNumPreviewPages; // number of default printed pages

size_t m_nSafetyPoolSize; // ideal size

void (AFXAPI* m_lpfnDaoTerm)();

void DevModeChange(LPTSTR lpDeviceName);
void SetCurrentHandles();
int GetOpenDocumentCount();

// helpers for standard commdlg dialogs
// overrides for implementation
virtual BOOL InitInstance();
virtual int ExitInstance(); // return app exit code
virtual int Run();
virtual BOOL OnIdle(LONG lCount); // return TRUE if more idle processing
virtual LRESULT ProcessWndProcException(CException* e, const MSG* pMsg);

public:
virtual ~CWinApp();

protected:
// (AFX_MSG(CWinApp)
afx_msg void OnAppExit();
afx_msg void OnUpdateRecentFileMenu(CCmdUI* pCmdUI);
afx_msg BOOL OnOpenRecentFile(UINT nID);
//}}AFX_MSG
DECLARE_MESSAGE_MAP()}

We summarize here the most important functions of CWinApp:
1. CWinApp uses a structure called CCommandLineInfo. CWinApp maintains a set of the command line parameters passed into WinMain. See the next chapter. They are the current instance
handle hWndInstance the previous instance handle hWndPrevInstance the command line parameters hWnd lpCmdLine and the show window flag hWnd nCmdShow

2. The Windows application needs to have a place where it can perform instance-specific initialization. In an MFC application this is performed in CWinApp::InitInstance MFC calls InitInstance before anything substantial happens in an application usually in the main window.

3. In MFC applications the message pump is supported by CWinApp. Calling CWinApp::Run starts a standard message pump GetMessage DispatchMessage loop. CWinApp’s message loop has explicit support for performing back-ground processing.

4. Finally, most applications need a place to perform shutdown and to clean up code. In an MFC application, ExitInstance serves this purpose.

2.2. AFX_MODULE_STATE class in MFC

Like most other Windows applications, MFC applications have to keep track of various items, including main window handles, resources, and module handles. To support these features MFC maintains much more information than a regular Windows application might. MFC defines a structure called AFX_MODULE_STATE in the AFXSTAT_H file. In the class definition in List 2-2, AFX_MODULE_STATE is derived from the base class CNoTrackObject. The base class, CNoTrackObject, is an undocumented base class. Designed for use by the MFC framework, classes derived from the CNoTrackObject class are exempt from memory leak detection.

List 2-2. AFX_MODULE_STATE in AFXSTAT_H

```cpp
// AFX_MODULE_STATE (global data for a module)
class AFX_MODULE_STATE : public CNoTrackObject
{
    public:
    #ifdef _AFXDLL
        AFX_MODULE_STATE(BOOL bDLL, WNDPROC fnAfxWndProc, DWORD dwVersion);
        AFX_MODULE_STATE(BOOL bDLL, WNDPROC fnAfxWndProc, 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_pszCurrentAppName;
    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
#endif
ifndef _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];
#endif
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)
#endif
ifdef _AFX_OLD_EXCEPTIONS
    // exceptions
   AFX_TERM_PROC m_pfnTerminate;
#endif
void (PASCAL *m_pfnFilterToolTipMessage)(MSG*, CWnd*);
#endif
ifdef _AFXDLL
    // CDynLinkLibrary objects (for resource chain)
    CTypedSimpleList<CDynLinkLibrary*>* m_libraryList;
    // special case for MFCxxLOC.DLL (localized MFC resources)
    HINSTANCE m_appLangDLL;
#endif
#endif
ifdef _AFX_NO_OCC_SUPPORT
    // OLE control container manager
    COccManager* m_pOccManager;
    // locked OLE controls
    CTypedSimpleList<COleControlLock*> m_lockList;
#endif
#endif
ifdef _AFX_NO.DAO_SUPPORT
    _AFX.DAO_STATE* m_pDaoState;
#endif
#endif
ifdef _AFX_NO_OLE_SUPPORT
    // Type library caches
    CTypeLibCache m_typeLibCache;
    CTypeLibCacheMap* m_pTypeLibCacheMap;
#endif

    // define thread local portions of module state
    THREAD_LOCAL(AFXXX_MODULE_THREAD_STATE, m_thread)
);
AFX_MODULE_STATE contains core information about the module; that is, information required by all MFC modules regardless of type—whether EXE or DLL. Collected within this structure are the module instance handle, the instance of the module from which to pull resources, pointer to the module's CWinApp-derived class, the name of the application, a pointer to the first node in the application's list of run-time class information structures and a pointer to an exception handler. Here is a rundown of the most important AFX_MODULE_STATE members:

- m_pCurrentWinApp: A pointer to a CWinApp.
- m_hCurrentInstanceHandle: The instance handle of this module.
- m_lpszCurrentAppName: A pointer to the application's name.
- m_bDLL: Indicates whether the module is a dynamic link library or an executable.
- m_classList: A pointer to the first run-time class in the application's list of CRuntimeClass structures.
- m_szUnregisterList[4096]: Maintains a list of registered window classes so that MFC can unregister them upon termination.
- m_fnAfxWndProc: Points to MFC's standard window procedure.
- m_fRegisteredClasses: Indicates which MFC window classes have already been registered.

3. Initialization

As remarked in Chapter 1, the MSS program MSS.exe is not the very first code that Windows executes when we start the application MSS. The linker inserts some start-up code that actually gets control from the Windows operating system. The start-up code in turn calls the function we view as the entry point of the application. The start-up code CRTEXE.c on the Visual C++ initializes the application. In our case the MSS application uses CRT DLL. The C/C++ run-time library CRT code performs the DLL startup sequence which initializes the C/C++ run-time library and invokes C++ constructors on static, non-local objects. However the CRTEXE.c itself could not explicitly be seen in the MSS program.

There are some initializations that have to be done for every running instance of the application. This means usually registering one or more window classes, and creating one or more windows, and creating and showing the main window.

3.1. WinMain()

In CRTEXE.c defined is WinMainCRTStartup function which is shown in List 3-1. By default the Visual C++ linker specifies the function WinMainCRTStartup as the C run-time library. This is the default starting address for an Windows application. The WinMain function here in WinMainCRTStartup is called by the Visual C++ as the initial entry point for a Windows-based application.

The roles of WinMain are the following three points:
1. Performs all necessary initialization which includes loading resources used by the program, registering window classes, and creating windows.
2. Executes a message loop fetching messages for the application and dispatching to the appropriate message-handling functions.
3. Terminates the application when the message loop detects a WM_QUERY message after freeing any resources possibly reserved by the initialization code.

```
void WinMainCRTStartup(void)
{
    int mainret;
    mainret = WinMain(
        GetModuleHandle(NULL),
        NULL,
        lpszCommandLine,
        StartupInfo.dwFlags & STARTF_USESHOWWINDOW
    ? StartupInfo.wShowWindow : SW_SHOWDEFAULT
    );
    exit(mainret);
}
```

The syntax of WinMain is described like this:

```
Syntax
int WinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpszCommandLine,
    int nCmdShow
);
```

The WinMain function requires four parameters. The first parameter sets a handle to the current instance of the application. The second parameter is a handle to the previous instance of the application. In the present version of Windows this parameter is always NULL. The third parameter is a pointer to a NULL-terminated string called lpszCommandLine which enables us to enter a command line to run a program. The forth parameter to the WinMain is nCmdShow which is an integer to specify how the application should display its main window. The exit module defines C run-time termination.

In the actual flow of the program, however, the WinMain function in WinMainCRTStartup is overridden by the following function in the APPMODUL.CPP in MFC with the same type of parameters as WinMain. This is where most programs perform application-specific and instance-specific initialization, as well as start up application's message loop.
extern "C" int WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine, int nCmdShow)
{
    // call shared/exported WinMain
    return AfxWinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow);
}

Here WINAPI specifies a calling sequence to be used to push some number of parameters on the stack in right-to-left ordering so as to pass them to the function. ".t" in front of WinMain is used for Unicode support. According to the program _tWinMain delegates processing to a function called AfxWinMain. The AfxWinMain is seen in the file WINMAIN.CPP below.

List 3-3. 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
            pThread->m_pMainWnd->DestroyWindow();
        }
The first thing AfxWinMain does is to declare a global pointer variable of type CWinApp and assign the value of AfxGetApp to it. It gets the single application object associated with the program in CWinApp-derived object is required by every MFC program. The C++ program constructs CWinApp-derived global object MMSS.CPP in our case even before WinMain is called. As long as CWinApp-derived object is included in our program, the object will be constructed by the time WinMain gets around to executing.

3.2. Constructing CWinApp

The constructor of CWinApp itself is found in APPCORE.CPP List 3-4. The main job of CWinApp's constructor is

1. to initialize CWinApp's member variables. CWinApp's constructor takes a single parameter: the name of the program. CWinApp sets its m_pszAppName variable to the value passed in. This parameter defaults to NULL.

2. to initialize the module's thread state structure of the AFX_THREAD_STATE class and module state structure of the AFX_MODULE_STATE class. CWinApp's constructor initializes the AFX_MODULE_STATE's m_pCurrentWinApp to the CWinApp being constructed. All of CWinApp's other members are set to NULL that is, the instance handle, the pointer to the main object,
window, the name of the application, and so on.

List 3-4. CWinApp::CWinApp() in APPCORE.CPP

```cpp
CWinApp::CWinApp(LPCTSTR lpszAppName)
{
    if (lpszAppName != NULL)
        m_pszAppName = _tcsdup(lpszAppName);
    else
        m_pszAppName = NULL;

    // initialize CWinThread state
    AFX_MODULE_STATE* pModuleState = _AFX_CMDTARGET_GETSTATE();
    AFX_MODULE_THREAD_STATE* pThreadState = pModuleState->m_thread;
    ASSERT(AfxGetThread() == NULL);
    pThreadState->m_pCurrentWinThread = this;
    ASSERT(AfxGetThread() == this);
    m_hThread = ::GetCurrentThreadId();
    m_nThreadId = ::GetCurrentThreadId();

    // initialize CWinApp state
    ASSERT(AfxCurrentWinApp == NULL); // only one CWinApp object please
    pModuleState->m_pCurrentWinApp = this;
    ASSERT(AfxGetApp() == this);

    // in non-running state until WinMain
    m_hInstance = NULL;
    m_pszHelpFilePath = NULL;
    m_pszProfileName = NULL;
    m_pszRegistryKey = NULL;
    m_pszExeName = NULL;
    m_pRecentFileList = NULL;
    m_pDocManager = NULL;
    m_atomApp = m_atomSystemTopic = NULL;
    m_lpCmdLine = NULL;
    m_pCmdInfo = NULL;

    // initialize wait cursor state
    m_nWaitCursorCount = 0;
    m_hcurWaitCursorRestore = NULL;

    // initialize current printer state
    m_hDevMode = NULL;
    m_hDevNames = NULL;
    m_nNumPreviewPages = 0; // not specified (defaults to 1)

    // initialize DAO state
    m_lpfnDaoTerm = NULL; // will be set if AfxDaoInit called

    // other initialization
    m_bHelpMode = FALSE;
    m_nSafetyPoolSize = 512; // default size
}
```
3.3. Initializing the Framework: AfxWinInit

Next, AfxWinMain calls AfxWinInit to initialize the framework. AfxWinInit is implemented in APPINIT.CPP as shown in List 3-5.

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

    // handle critical errors and avoid Windows message boxes
    SetErrorMode(SetErrorMode(SEM_FAILCRITICALERRORS) |
    SEM_NOOPENFILEERRORBOX);

    // set resource handles
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    pModuleState->m_hCurrentInstanceHandle = hInstance;
    pModuleState->m_hCurrentResourceHandle = hInstance;

    // fill in the initial state for the application
    CWinApp* pApp = AfxGetApp();
    if (pApp != NULL)
    {
        // Windows specific initialization (not done if no CWinApp)
        pApp->m_hInstance = hInstance;
        pApp->m_hPrevInstance = hPrevInstance;
        pApp->m_lpCmdLine = lpCmdLine;
        pApp->m_nCmdShow = nCmdShow;
        pApp->SetCurrentHandles();
    }

    // initialize thread specific data (for main thread)
    if (!afxContextIsDLL)
    {
        AfxInitThread();
    } else {
        return TRUE;
    }
```

Whenever a Windows application starts, Windows passes the four parameters to the application: the
current instance handle, the previous instance handle, the command line parameters, and the show command. 

AfxWinInit takes those four parameters as its arguments. AfxWinInit sets CWinApp::m_hInstance, 
CWinApp::m_hPrevInstance, CWinApp::lp_CmdLine, and CWinApp::nCmdShow to those passed 
parameters.

AfxWinInit sets the error mode for application using SetErrorMode. This designates what will 
cause the application to fail.

AfxWinInit then calls AfxGetModuleState to get the module's AFX_MODULE_STATE 
structure (See Chapter 2.2). AfxWinInit stores the module instance handle and 
the resource handle in AFX_MODULE_STATE::m_hCurrentInstanceHandle and 
AFX_MODULE_STATE::m_hCurrentResourceHandle, respectively. CWinApp keeps these parameters as member variables. At this 
point both the current instance handle and the resource handle point to the module's instance 
handle hInstance.

Then AfxWinInit calls the application object's SetCurrentHandles function to initialize the 
application name and path variables within CWinApp. SetCurrentHandles is shown in List 3-6.

List 3-6. CWinApp::SetCurrentHandles() in APPINIT.CPP

```cpp
void CWinApp::SetCurrentHandles()
{
    ASSERT(this == afxCurrentWinApp);
    ASSERT(afxCurrentAppName == NULL);

    AFX_MODULE_STATE* pModuleState = _AFX_CMDTARGET_GETSTATE();
    pModuleState->m_hCurrentInstanceHandle = m_hInstance;
    pModuleState->m_hCurrentResourceHandle = m_hInstance;

    // get path of executable
    TCHAR szBuff[_MAX_PATH];
    VERIFY(::GetModuleFileName(m_hInstance, szBuff, _MAX_PATH));

    LPCTSTR lpszExt = _tcsrchr(szBuff, '.');
    ASSERT(lpszExt != NULL);
    ASSERT(*lpszExt == '．');
    *lpszExt = 0; // no suffix

    TCHAR szExeName[_MAX_PATH];
    TCHAR szTitle[256];
    // get the exe title from the full path name [no extension]
    VERIFY(AfxGetFileName(szBuff, szExeName, _MAX_PATH) == 0);
    if (m_pszExeName == NULL)
    {
        BOOL bEnable = AfxEnableMemoryTracking(FALSE);
        m_pszExeName = _tcsdup(szExeName); // save non-localized name
        AfxEnableMemoryTracking(bEnable);
    }

    // m_pszAppName is the name used to present to the user
```
if (mpszAppName == NULL)
{
    BOOL bEnable = AfxEnableMemoryTracking(FALSE);
    if (AfxLoadString(AFIDS_APP_TITLE, szTitle) != 0)
        mpszAppName = _tcstoul(szTitle);  // human readable title
    else
        mpszAppName = _tcstoul(mpszExeName);  // same as EXE
    AfxEnableMemoryTracking(bEnable);
}

pModuleState->m_lpszCurrentAppName = mpszAppName;
ASSERT(afxCurrentAppName != NULL);

// get path of .HLP file
if (mpszHelpFilePath == NULL)
{
    lstrcpy(lpszExt, _T("\HLP"));
    BOOL bEnable = AfxEnableMemoryTracking(FALSE);
    mpszHelpFilePath = _tcstoul(szBuff);
    AfxEnableMemoryTracking(bEnable);
    *lpszExt = 'Y';  // back to no suffix
}

if (mpszProfileName == NULL)
{
    lstrcat(szExeName, _T("\INI"));  // will be enough room in buffer
    BOOL bEnable = AfxEnableMemoryTracking(FALSE);
    mpszProfileName = _tcstoul(szExeName);
    AfxEnableMemoryTracking(bEnable);
}
}

First SetCurrentHandles sets AFX_MODULE_STATE handles again. Then SetCurrentHandles uses GetModuleFileName to retrieve the module file name. SetCurrentHandles then initializes the mpszExeName to the executable file in the present case "MSS" SetCurrentHandles then initializes the mpszAppName to the title of the application "MSS" SetCurrentHandles also sets the application's AFX_MODULE_STATE pModuleState->m_lpszCurrentAppName to the same values as mpszAppName. SetCurrentHandles also fills CWinApp's help file and profile strings: mpszHelpFilePath and mpszProfileName. SetCurrentHandles Initializes mpszHelpFilePath to "S:\Prax2\VCPP\FIXEDRTENA\CTIVEX2\Debug\MSS.HLP" and mpszProfileName to "MSS.INI".

Up to here the application and the framework are both initialized properly. Now that the handles and file names are all initialized correctly, MFC continues with the rest of the application.

3.4. InitApplication and InitInstance

Next AfxWinMain calls the application's InitApplication and InitInstance. InitApplication is
implemented in APPCORE.CPP [See List 3-7] The InitInstance in CWinApp is implemented again in APPCORE.CPP [See List 3-8] In this section our MSS.CPP the application framework object of CMSSApp application class shows up explicitly in the process of initialization and registration.

List 3-7. CWinApp::InitApplication() in APPCORE.CPP

```cpp
BOOL CWinApp::InitApplication()
{
    if (CDocManager::pStaticDocManager != NULL)
    {
        if (m_pDocManager == NULL)
            m_pDocManager = CDocManager::pStaticDocManager;
        CDocManager::pStaticDocManager = NULL;
    }

    if (m_pDocManager != NULL)
        m_pDocManager->AddDocTemplate(NULL);
    else
        CDocManager::bStaticInit = FALSE;

    return TRUE;
}
```

List 3-8. CWinApp::InitInstance() in APPCORE.CPP

```cpp
BOOL CWinApp::InitInstance()
{
    return TRUE;
}
```

In Win32 system InitApplication actually does nothing with initialization. CWinApp's version of InitApplication initializes the application's document manager. All initialization should take place in InitInstance.

Now we are at InitInstance in AfxWinMain. AfxWinMain calls pThread->InitInstance pThread->InitInstance is overridden by CWinApp::InitInstance because CWinApp is derived from CWinThread. Whenever a program begins, it is necessary to perform initializations for a certain instance of the program. CWnd::InitInstance serves that purpose. CWinApp's default implementation of InitInstance does nothing as seen in List 3-8. It just returns TRUE. However, CWinApp::InitInstance is also virtual, so we can safely override it by our CMSSApp::InitInstance As a result it is pThread->InitInstance there that is overridden by the CMSSApp::InitInstance which will be shown in List 4-1 in the next chapter. This time CMSSApp::InitInstance in turn initializes the present application instance.

Activities that take place inside InitInstance include such tasks as setting all the documents for an application and showing the main window. Because the default version of InitInstance does nothing, it is up to us to make sure that such a window appears on the screen. In Chapter 4 we can see that CMSSApp::InitInstance registers the present application instance and displays the main window.
3.5. Priming the Message Pump: CWinApp::Run

The last thing WinMain does before leaving is to call the CWinApp-derived object's Run function. Run starts the ball rolling with the message loop. The Run function does a little more than just a generic GetMessage-DispatchMessage loop. The mechanism of message loop and message handling is to be reviewed and discussed in the forthcoming article. Because CWinApp is derived from CWinThread, at this point CWinApp-derived class "pThread" simply defers to the CWinThread's Run function to start the message pump.

```cpp
nReturnCode = pThread->Run();
```

4. Registering Window Classes

Initialization procedures now proceed to the execution of function pThread->InitInstance in AfxWinMain List 3-3. WINMAIN.CPP. The variable pThread which is actually a thread of execution within MFC is an instance of CWinThread. Since CWinApp is derived from CWinThread, CWinApp::InitInstance overrides pThread->InitInstance. This is where our application framework MSS.cpp shows up in the initialization. The application class CMSSApp is derived from CWinApp and the member function CMSSApp::InitInstance in turn overrides CWinApp::InitInstance and is implemented in MSS.CPP as shown in List 4-1.

```
/////////CMSSApp::InitInstance() in MSS.CPP

 Habd1CIMSSApp: InitInstance()
 {
   CWzSplash wndSplash;
   wndSplash.Create(IDB_WZDSPLASH);// 流画"MSSystem.bmp"のとき
   wndSplash.UpdateWindow(); //send WM_PAINT
   AfxEnableControlContainer();

   #ifdef _AFXDLL
   Enable3dControls(); // Call this when using MFC in a shared DLL
   #else
   Enable3dControlsStatic(); // Call this when linking to MFC statically.
   #endif
 ```
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
LoadStdProfileSettings(); // Load standard INI file options (including MRU)

// Register the application’s document templates. Document templates
// serve as the connection between documents, frame windows and views.

CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_MSSTYPE,
    RUNTIME_CLASS(CMSSDoc),
    RUNTIME_CLASS(CChildFrame), // main SDI frame window
    RUNTIME_CLASS(CMSSView));
AddDocTemplate(pDocTemplate);
(........)

// create main MDI Frame window
CMainFrame* pMainFrame = new CMainFrame;
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
    return FALSE;
m_pMainWnd = pMainFrame;

// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);

// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
    return FALSE;

// The main window has been initialized, so show and update it.
pMainFrame->ShowWindow(m_nCmdShow);
pMainFrame->UpdateWindow();

    return TRUE;
}
As pointed out in Chapter 3. Initialization, the Windows application generally requires a set of standard initializations to be done before the program really runs. Typically these involve registering one or more window classes and creating one or more windows.

The first operation an InitInstance function does is to register the window classes. A window class defines certain attributes common to all windows that are created based on that class. The structure type WNDCLASS defines ten attributes of a window:

```c
typedef struct tagWNDCLASS
{
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCSTR lpstrMenuName;
    LPCSTR lpstrClassName;
} WNDCLASS;
```

Those are all class attributes of a window. The first field is the window class style, for example, CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW, which causes Windows to detect a double-click for an application and to send double-click messages to all windows of this class, or which causes a window to be completely redrawn whenever the width of the client area (CS_HREDRAW) or the height of the client area (CS_VREDRAW) changes. These symbolic constants are combined with the C bitwise OR operator (|”). The second field of the WNDCLASS structure contains the address of the function that processes all messages for all windows that are created based on this window class. The third field specifies the memory to be allocated to contain information about a registered class. The fourth field specifies how many extra bytes should be allocated for each window structure created for this class. The fifth field of the WNDCLASS is hInstance. A window class is owned by the module instance assigned to this field. Most applications set this field to the instance handle that was passed to the application as a parameter of WinMain. The sixth field is for specifying the icons for a window class, hIcon, the handle of the large icons. The seventh field is hCursor for a cursor. The eighth field is a handle to a brush. Windows uses the brush small colored bit map pattern of pixels indicated in this field to paint the background of the client area of all windows created based on this class. The ninth field is a LPCTSTR pointer to the name of a menu resource used for all windows created based on the class. The tenth field is also a LPCTSTR pointer to the first character of the class name loaded by the LordString function.

CMSSApp::InitInstance calls the function pMainFrame->LoadFrame in MSS.CPP [List 4-1] pMainFrame->LoadFrame then calls inherits’ in this case CMDIFrameWnd::LoadFrame in
WINMDI.CPP  List 4-2  CMDIFrameWnd::LoadFrame then calls' CFrameWnd::LoadFrame in WINFRM.CPP  List 4-3  Here in WINFRM.CPP, we finally reach AfxDeferRegisterClass that registers window classes used in MFC.

Before a Windows application can display a window, the application has to register at least one window class. An MFC application is just like any other Windows applications, so it needs to register at least one window class as well. As was described above a window class defines very basic aspects of a window, such as its appearance via some flags and its behavior via a callback function MFC actually registers four standard window classes: regular child windows, a control bar window, an MDI frame window, and a window for an SDI or MDI child window.

List 4-2. CMDIFrameWnd::LoadFrame() in WINMDI.CPP

```cpp
BOOL CMDIFrameWnd::LoadFrame(UINT nIDResource, DWORD dwDefaultStyle,
   CWnd* pParentWnd, CCreateContext* pContext)
{
   if (!CFrameWnd::LoadFrame(nIDResource, dwDefaultStyle,
      pParentWnd, pContext))
      return FALSE;

   // save menu to use when no active MDI child window is present
   ASSERT(m_hWnd != NULL);
   m_hMenuDefault = ::GetMenu(m_hWnd);
   if (m_hMenuDefault == NULL)
      TRACE0("Warning: CMDIFrameWnd without a default menu.
   return TRUE;
}
```

List 4-3. CFrameWnd::LoadFrame() in WINFRM.CPP

```cpp
BOOL CFrameWnd::LoadFrame(UINT nIDResource, DWORD dwDefaultStyle,
   CWnd* pParentWnd, CCreateContext* pContext)
{
   // only do this once
   ASSERT_VALID_IDR(nIDResource);
   ASSERT(m_nIDHelp == 0 || m_nIDHelp == nIDResource);

   m_nIDHelp = nIDResource; // ID for help context (+HID_BASE_RESOURCE)
   CString strFullString;
   if (strFullString.LoadString(nIDResource))
      AfxExtractSubString(m_strTitle, strFullString, 0); // first sub-string

   VERIFY(AfxDeferRegisterClass(AXWNDFRAMEORVIEW_REG));

   // attempt to create the window
   LPCTSTR lpszClass = GetIconWndClass(dwDefaultStyle, nIDResource);
   LPCTSTR lpszTitle = m_strTitle;
   if (!Create(lpszClass, lpszTitle, dwDefaultStyle, rectDefault,
      pParentWnd, MAKEINTRESOURCE(nIDResource), 0L, pContext))
      return FALSE; // will self destruct on failure normally
```
The following examples show the four decoration window class names supplied for "nonstatic" and "debug" builds of an application, each corresponding to the above mentioned window class respectively:

AfxWnd42d: all child windows with no icon, arrow cursor and no background color.

AfxControlBar42d: the standard control bar implementation with no icon, arrow cursor and gray background color.

AfxMDIFrame42d: the MDI frame window - that is, the parent window - with icon, arrow cursor and no background color.

AfxFrameOrView42d: frame windows and views with icon, arrow cursor and background color.

The names for window classes are decorated with the MFC version number "42" which means "MFC4.2". In addition, MFC uses information about whether or not the application is statically linked and information about whether the application is a debug or release build of MFC in order to decorate the class names.

MFC provides a helper function or a macro routine for registering a window class. Given a set of attributes - window class style, cursor, background brush, and icon - a synthetic name is generated, and the resulting window class is registered. According to the AFXIMPL.H header file, MFC defines a macro AfxDeferRegisterClass fClass as a helper function AfxEndDeferRegisterClass fClass:

```cpp
#define AfxDeferRegisterClass(fClass) AfxEndDeferRegisterClass(fClass)
```

Therefore a call AfxDeferRegisterClass AFX_WNDFRAMEORVIEW_REG for example, in CFrameWnd::LoadFrame in List 4-3 invokes AfxEndDeferRegisterClass AFX_WNDFRAMEORVIEW_REG with a parameter AFX_WNDFRAMEORVIEW_REG. The following values including this parameter represent MFC’s four standard window classes in binary representation by bitmap which correspond to the above mentioned four standard window classes, respectively:
AFXWndReg Dx001 D
AFXWndControlBarReg Dx002 D
AFXWndMDIReg Dx004 D
AFXWndFrameReg Dx008 D

In the parentheses hexadecimal numbers are assigned for reference.

List 4-4. AfxEndDeferRegisterClass() in WINCORE.CPP

WINCORE.CPP
BOOL AFXAPI AfxEndDeferRegisterClass(LONG fToRegister)
{
    // mask off all classes that are already registered
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    fToRegister &= ~pModuleState->m_fRegisteredClasses;
    if (fToRegister == 0)
        return TRUE;

    LONG fRegisteredClasses = 0;

    // common initialization
    wndcls;
    memset(&wndcls, 0, sizeof(WNDCLASS)); // start with NULL defaults
    wndcls.lpfnWndProc = DefWindowProc;
    wndcls.hInstance = AfxGetInstanceHandle();
    wndcls.hCursor = afxData.hcurArrow;

    INITCOMMONCONTROLSEX init;
    init.dwSize = sizeof(init);

    // work to register classes as specified by fToRegister, populate fRegisteredClasses as we go
    if (fToRegister & AFXWndFrameReg)
    {
        // SDI Frame or MDI Child windows or views - normal colors
        wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
        wndcls.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
        if (_AfxRegisterWithIcon (&wndcls, _afxWndFrameOrView, AFX_IDI_STD_FRAME))
            fRegisteredClasses |= AFXWndFrameReg;
    }

    // save new state of registered controls
    pModuleState->m_fRegisteredClasses |= fRegisteredClasses;

    // must have registered at least as many classes as requested
    return (fToRegister & fRegisteredClasses) == fToRegister;
}
Now, let us take a close look at `AfxEndDeferRegisterClass` in WINCORE.CPP. At the beginning of the helper function, if the present module state `pModuleState->m_fRegisteredClasses` has already been registered `fToRegister`, then `fToRegister` is set to zero and the helper function returns TRUE. Otherwise, `fToRegister` maintains its passed-in value and `fRegisteredClasses` is set to zero.

In the middle of the helper function `AfxEndDeferRegisterClass` zeroes out a WNDCLASS structure using `memset` so that all fields except those being set explicitly are NULL or zero. `AfxEndDeferRegisterClass` initializes `WNDCLASS::lpfnWndProc` to `DefWindowProc`. `DefWindowProc` provides default processing for any window messages that an application does not process. The window class's `hInstance` handle is initialized to the current instance handle. `AfxEndDeferRegisterClass` sets the window's cursor to the regular arrow cursor. These values are common among all MFC window classes.

Once the class structure has been zeroed out and the common fields have been initialized, `AfxEndDeferRegisterClass` starts filling in the class structure at the end of the helper function, depending on the window class being registered. Since an argument `fToRegister` of type `AfxEndDeferRegisterClass` passes a window class `AFX_WNDFRAMEORVIEW_REG` in the present case `MSS` system `AfxEndDeferRegisterClass` is trying to register an SDI frame windows or MDI child windows or views with the name "AfxFrameOrView42d". It has a style of `CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW`. The background brush is the default color for a window. The window class is registered with the default SDI frame icon, called `AFX_IDI_STD_FRAME`. The actual registration is performed by another helper function `_AfxRegisterWithIcon` in WINCORE.CPP as shown in List 4-5. Here a `WNDCLASS`-type pointer variable `lpWndCls` is now given a constant pointer to the string `lpszClassName` to which the window class name "AfxFrameOrView42d" is assigned in our case.

Finally, the `WNDCLASS` structure of "pWndCls" is completely filled out. Then we can now register the window class by using `AfxRegisterClass` `pWndCls` However `AfxRegisterClass` does nothing except returning TRUE. As a result `AfxEndDeferRegisterClass` sets the global variable `fRegisteredClasses` to `AFX_WNDFRAMEORVIEW_REG`. MFC uses the global `fRegisteredClasses` variable to optimize the window registration. This is important because MFC attempts to register the window classes in many different places. At the very end of `AfxEndDeferRegisterClass` `AFX_MODULE_STATE`-type pointer variable `pModuleState` saves `fRegisteredClasses` and the helper function `AfxEndDeferRegisterClass` returns TRUE.

List 4-5. `_AfxRegisterWithIcon()` in WINCORE.CPP

```c
///
// Standard init called by WinMain
AFX_STATIC BOOL AFXAPI _AfxRegisterWithIcon(WNDCLASS* pWndCls,
LPCTSTR lpszClassName, UINT nIDIcon)
{
    pWndCls->lpszClassName = lpszClassName;
    HINSTANCE hInst = AfxFindResourceHandle(
        MAKEINTRESOURCE(nIDIcon), RT_GROUP_ICON);
```
if ((pWndCls->hIcon = ::LoadIcon(hInst, MAKEINTRESOURCE(nIDIcon))) == NULL) {
    // use default icon
    pWndCls->hIcon = ::LoadIcon(NULL, IDI_APPLICATION);
}
return AfxRegisterClass(pWndCls);
}

By now the WNDCLASS structure is completely filled out. AfxEndDeferRegisterClass uniquely identifies the class being registered or 0 if an error occurred. And a new window will be created with CreateWindow in the next chapter.

5. Creating the Main Window

Now that we have registered the window class, the next initialization process comes into the final step of creating and showing windows. Let us see the middle part of CFrameWnd::LoadFrame in WINFRM.CPP. LordFrame performs this task by calling the function Create The function Create here is implemented by CFrameWnd::Create in WINFRM.CPP as shown List 5-1.

List 5-1. CFrameWnd::Create() in WINFRM.CPP

BOOL CFrameWnd::Create(LPCTSTR lpszClassName,
    LPCTSTR lpszWindowName, DWORD dwStyle,
    const RECT& rect,
    CWnd* pParentWnd,
    LPCTSTR lpszMenuName,
    DWORD dwExStyle,
    CCreateContext* pContext)
{
    HMENU hMenu = NULL;
    if (lpszMenuName != NULL)
    {
        // load in a menu that will get destroyed when window gets destroyed
        HINSTANCE hInst = AfxFindResourceHandle(lpszMenuName, RT_MENU);
        if ((hMenu = ::LoadMenu(hInst, lpszMenuName)) == NULL)
        {
            TRACE0("Warning: failed to load menu for CFrameWnd.\n");
            PostNcDestroy(); // perhaps delete the C++ object
            return FALSE;
        }
    }

    m_strTitle = lpszWindowName; // save title for later
if (!CreateEx(dwExStyle, lpszClassName, lpzWindowName, dwStyle,
    rect.left, rect.top, rect.right - rect.left, rect.bottom - rect.top,
    pParentWnd->GetSafeHwnd(), hMenu, (LPVOID)pContext))
{
    TRACE0("Warning: failed to create CFrameWnd.\n");
    if (hMenu != NULL)
      DestroyMenu(hMenu);
    return FALSE;
}

return TRUE;

CFrameWnd::Create calls CreateEx which inherits CWnd::CreateEx in WINCORE.CPP

List 5-2 In the argument of the function there are LPCTSTR type local variables lpszClassName that contains the address of the registered window class  "\nwhich we will base a window soon to be created, and lpzWindowName that contains the address of the window name "MSS" and so on. In the middle of the program CWnd::CreateEx defines a local variable hWnd. The variable hWnd holds the window handle of the created main frame window. CreateEx in turn calls the global function :CreateWindowEx which creates an overlapped, pop-up, or child window with an extended window style specified by dwExStyle.

List 5-2. CWnd::CreateEx() in WINCORE.CPP

BOOL CWnd::CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName,
    LPCTSTR lpzWindowName, DWORD dwStyle,
    int x, int y, int nWidth, int nHeight,
    HWND hWndParent, HMENU nIDorHMenu, LPVOID lpParam)
{
    // allow modification of several common create parameters
    CREATESTRUCT cs;
    cs.dwExStyle = dwExStyle;
    cs.lpszClass = lpszClassName;
    cs.lpszName = lpzWindowName;
    cs.style = dwStyle;
    cs.x = x;
    cs.y = y;
    cs.cx = nWidth;
    cs.cy = nHeight;
    cs.hwndParent = hWndParent;
    cs.hMenu = nIDorHMenu;
    cs.hInstance = AfxGetInstanceHandle();
    cs.lpCreateParams = lpParam;

    if (!PreCreateWindow(cs))
    {
        PostNcDestroy();
        return FALSE;
    }
When we create a window, Windows does not return a pointer to its internal window structure. Instead we receive a window handle. We can give the window handle back to Windows whenever we need to identify the window. The values of the ClassName and WindowName variables are fetched from the program's string resource pool.

Finally we are ready to create our first window. Although we specified ten different characteristics when we defined the window class, the CreateEx function call requires twelve more. Unlike the window class characteristics, which apply to all windows based on the class, the characteristics supplied in the ::CreateWindowEx call apply only to the individual window being created. CreateEx returns TRUE and CreateEx returns TRUE in List 5-1.

Our very first window has now been created. The window, although created, has not yet been made visible. We make the window visible by using the ShowWindow function in MSS.CPP in List 4-1: pMainFrame->ShowWindow WINAPI which is shown in List 5-3. CWnd::ShowWindow INT nCmdShow with two arguments. That means MFC wraps the Windows API functions and CWnd::ShowWindow API function. CWnd encapsulates all the Windows API functions that take a window handle HWND m_hWnd where HWND: data type definition,
a Window handle and m_hWnd: a member variable. Thus ::ShowWindow is passed two parameters: the window handle returned by the CreateWindowEx call used to identify the window to show and the nCmdShow parameter originally passed to WinMain which specifies how the window should appear.

```cpp
pMainFrame->ShowWindow(m_nCmdShow);
pMainFrame->UpdateWindow();
```

The ShowWindow function call displays the window on the screen. The window will be normal-sized, maximized, or iconic depending on the value of the nCmdShow parameter. When the window is either normally displayed or maximized, or, conversely, not iconic, the client area of the window will be erased by painting it with the background brush specified in the window class. Incidentally, the UpdateWindow function forces the client area to be updated immediately if it needs it.

CMSSApp::InitInstance function has now successfully completed its only tasks, creating and displaying the application's main window, therefore it returns TRUE to WinMain.

List 5-3. CWnd::ShowWindow in WINOCC.CPP

```cpp
BOOL CWnd::ShowWindow(int nCmdShow)
{
    ASSERT(::IsWindow(m_hWnd));
    if (m_pCtrlSite == NULL)
        return ::ShowWindow(m_hWnd, nCmdShow);
    else
        return m_pCtrlSite->ShowWindow(nCmdShow);
}
```

6. Summary

We have examined the characteristic mechanism of the Windows programming which fully exploits object-oriented programming techniques based on the Visual C++ platform. The Visual C++ language processor invokes a full-featured MFC class library and MFC itself is built upon object-oriented techniques. Whereas Windows programming techniques provide a comprehensive, robust and visual software developing environment, their implementation of the objects, i.e. data and procedures, and their mechanism under the Visual C++ processor are hidden behind the interface that shows up between the language processor and the Windows application.

In the present article we have concentrated on what the initialization processes are all about that the Windows programming deals with and what their characteristics are. As an example of the Windows application, we used the MSS system we have recently developed in our Seminars in the Hokusei Gakuen University. The initialization is comprised of registering window classes and creating the main window.

We started with searching for the WinMain entry point that every Windows program is supposed to have. We finally reached the very sites where window classes registration and window creation are
performed. The "chasing" processes were exciting and instructive, since we had to thread our way through a
hierarchy of classes, recognizing abstraction of data and function of all the functionalities from the "base" to "derived" classes and polymorphism via virtual member functions of all of which characterize the essentials of what classes are which result in understanding the object-oriented Windows programming.

AxFEndDeferRegisterClass function called by InitInstance through CMDIFrameWnd::
LoadFrame in MSS.CPP registers window classes which define attributes common to all windows by filling the WNDCLASS structure. Before a Windows application can display a window, the application has to register at least one window class. MFC can register four standard window classes.

CFrameWnd::Create function called by InitInstance through CMDIFrameWnd::LoadFrame calls CWnd::CreateEx finally calls the global function ::CreateWindowEx which creates our main window. Unlike the window class characteristics, which apply to all windows based on that class, the characteristics supplied in the CreateWindowEx call apply only to the individual window to be created.

The window just created becomes visible by executing CWnd::ShowWindow() (called by InitInstance()) which invokes the global ::ShowWindow() API function.

The InitInstance function successfully completes its only tasks creating and displaying the application's main window and returns TRUE to WinMain

[Acknowledgments]

The present Management Support System interface programs are developed by using Microsoft Visual C++ ver6.0 under Windows2000 and WindowsXP. The PROLOG program is written in and processed by Strawberry PROLOG ver2.3 developed by D. D. Dobrev, Sofia, Bulgaria. Strawberry PROLOG performs under Windows2000 and WindowsXP.

[References]


Notes:

API here refers to Microsoft Platform Win32 Software Development Kit ー SDK ー Application Programming Interface ー API ー SDK provides a set of functions, data types, structures and tools for writing application programs upon Windows. The core Win32 API covers an extremely broad area of interfaces such as input and output devices, user interface elements, system services and graphic elements. MFC logically groups the Windows API using object-oriented principles of abstraction, encapsulation, inheritance, polymorphism, and modularity.

Unicode is a 16-bit character code that allows us to intermix a variety of international languages in our application.
[Abstract]

Development of a Management Support System on the Windows Platform(II): Registering Window Classes and Creating the Main Window

Hiroshi Noto

This paper is based on an examination of the mechanism of the Windows programming which fully exploits object-oriented programming techniques based on the Visual C++ platform. Since the Visual C++ language processor invokes a full-featured MFC class library and MFC itself is built upon object-oriented concepts, the mechanism of the Windows programming is invisible behind the interface. In this paper, we have 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 are carried out. As an example of a Windows application, we used the MSS (management support system) that we have recently developed. By searching for the WinMain() entry point, we finally reached the sites where window classes registration and window creation were performed. The "chasing" processes were very exciting and instructive, since we had to thread our way through a hierarchy of classes, recognizing abstraction, inheritance and polymorphism, all of which characterize the essentials of the object-oriented programming. Before a Windows application can display a window, the application has to register at least one window class. Unlike the window class characteristics which apply to all windows based on that class, the characteristics supplied in the window creation function call apply only to the individual window to be created. The InitInstance() function successfully completes initialization and returns TRUE to WinMain().

Key words: Object-oriented programming ☐ Windows programming ☐ MFC class library, Initialization ☐ Window classes registration