

1. Introduction

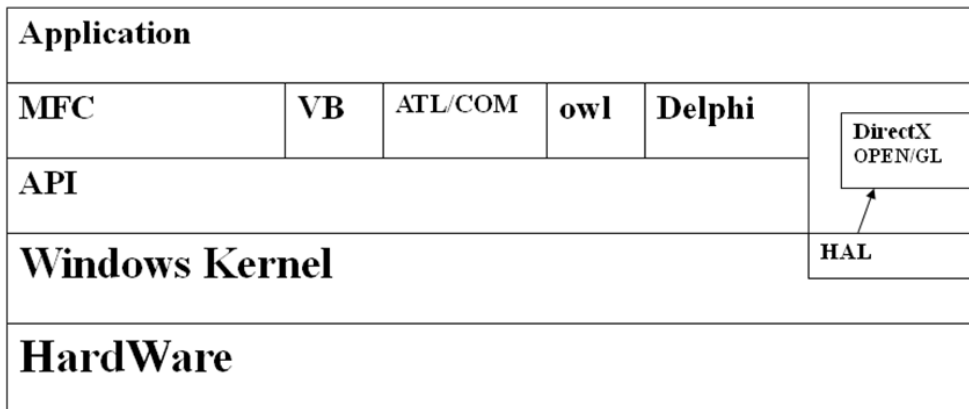
1. 1 API

Application Programming Interface

윈도우즈는 응용 프로그램을 위한 함수 집합을 제공하는데 이를 API라고 하며 좀 더 정확하게 표현하면 윈도우 API라고 한다. API는 특정 시스템(운영체제든 하드웨어든)을 위한 함수 집합을 이르는 일반명사이며 그 중의 하나가 윈도우 API이다.

API는 윈도우 버전에 따라 여러 종류가 있다. Win32API란 본격적인 32비트 운영체제인 윈도우 95/98과 NT에서 제공하는 32비트 윈도우즈를 위한 API이다. 64비트 CPU는 AMD의 X64와 인텔의 IA64 등이 있는데 이에 맞추어 64비트 운영체제인 Windows8 운영체제가 제공되고 있다. 참고로 Win32API의 모든 함수들은 거의 변화없이 64비트에서도 지원되며 32비트 응용 프로그램들도 약간의 노력으로 쉽게 64비트로 포팅할 수 있다. 본 책에서는 Win32API를 기반으로 내용을 다루고자 한다.

API는 운영체제가 직접 사용하는 라이브러리이다. 그림 1.1에 보면 Windows Kernel 상단에 API 레이어를 확인해 볼 수 있다.



[그림 1.1] API Positioning

API는 크게 3개의 모듈로 구분할 수 있다. 표 1.1에서 해당 내용을 살펴볼 수 있으며, 각각의 모듈은 관련 DLL에서 구현되어 있다.

모듈	DLL	설명
GDI Module	Gdi32.dll	화면이나 프린터 같은 장치의 출력을 관장하며 메시지를 관리 (예 : pen, brush, font, bitmap)
UI(User) Module	User32.dll	사용자 인터페이스 객체들을 관리 (예 : window, dialog, menu, cursor, icon, caret)
Kernel Module	Kernel32.dll	윈도우 OS의 핵심으로써 메모리관리, 파일 입출력, 프로그램의 로드와 실행 등 운영체제의 기본 기능을 수행 (예 : Process, thread, memory, I/O, Synchronization)

[표1.1] API 모듈

1. 2 Windows 의 특징

1.2.1 Windows의 특징

- 32bit 선점형 멀티태스킹과 멀티스레딩을 지원한다.
- GUI 환경이다.
- 동적 연결이라는 개념을 중심으로 동작한다.
- 장치에 독립적이다.

1.2.2 도스프로그래밍과 윈도우프로그래밍의 차이점

- 프로그램 흐름

도스 프로그램 : 절차적(또는 순차적)

프로그램의 실행 흐름이 프로그래머가 기술한 코드의 순서에 따라 진행된다.

윈도우 프로그램 : 이벤트 구동방식 또는 메시지 처리 방식

프로그램의 실행 흐름을 윈도우 시스템과 일을 분담하여 처리한다.

외부에서 일어나는 일을 윈도우 시스템이 감지하여 해당 프로그램에 메시지를 전달하여 준다.

프로그램은 이에 대한 처리만 한다.

프로그램이 한결 수월해 진다

참고로 spy++ 을 사용하여 특정 어플리케이션에서 발생하는 이벤트 정보를 확인해 볼 수 있다.

1.3 먼저 알아야 할 것들

다음 장부터 API를 이용한 GUI 프로그래밍 관련 코드를 학습하게 되는데, 이에 앞서 코드를 이해하기 위한 몇 가지 부분을 정리해 보고자 한다.

1.3.1 사용자 정의 데이터 타입

윈도우에서는 사용자 정의 데이터 타입들이 많이 있다. 이 데이터형은 windows.h 파일에 typedef 로 선언되어 있으며 표준 데이터형처럼 사용된다.

데이터형	의미
BYTE	unsigned char 형
CHAR	char 형
WORD	unsigned short 형
DWORD	unsigned long 형
LONG	Long과 동일
BOOL	Int 형 TRUE, FALSE 중 한 값을 가짐

[표1.2] 데이터 타입 정의

왜 사용자 정의 데이터 타입을 사용하는 것일까? 예를 들어 WORD형이 현재는 2byte의 부호없는 정수이지만 향후 부호없는 정수는 4byte나 혹은 다른 byte크기로 변환될 수 있다. 이러한 경우 WORD라는 사용자 정의 데이터 타입을 사용한다면 소스 수정없이 헤더 파일에서 WORD형의타입 정의만 바꾸고 소스를 다시 컴파일하면 된다. 즉, 중간타입을 정의해서 사용하는 이유는 이식성과 호환성 확보를 위해 필요하다.

단, int는 UINT라는 중간 타입이 있기는 하지만 C언어 자체에서 플랫폼에 종속적인 타입으로 정의하므로 그냥 사용해도 무방하다.

1.3.2 유니코드(Unicode)

유니코드는 16비트의 단일 값으로 지구상의 모든 문자를 표현할 수 있는 문자 코드 체계이다. 유니코드와 관련된 자세한 내용은 책 마지막 부분에서 자세히 다루도록 하고 여기서는 간단한 개념만 이해하도록 하자.

문자를 사용하기 위해 타입을 선택해야 하는데 이 때 일반적으로 char 타입을 사용해 왔다. 이는 DBCS(Doublebyte Charater Set) 방식으로 문자의 종류에 따라 1byte나 2byte로 표현하는 방식이다. 이와 달리 wchar_t 타입은 유니코드 방식으로 모든 문자를 2byte 형식으로 표현한다.(NULL 또한 2byte이다.)

결국 문자를 사용하기 위해서는 선택을 해야 하는데, 이러한 고민 없이 범용타입을 사용할 수 있다. 사용방식은 아래와 같다.

- 1) TCHAR.h 파일을 포함하자.

```
//TCHAR.h

#ifdef UNICODE
    typedef wchar_t TCHAR;
    #define _TEXT(X) L##X
    #define _T(X)      _TEXT(X)
    #define _tcslen    wcslen
#else
    typedef char TCHAR;
    #define _TEXT(X) X
    #define _T(X)      _TEXT(X)
    #define _tcslen    strlen
#endif
```

- 2) 범용타입을 사용하여 변수를 선언하자.

DBCS	Unicode	범용타입
char	wchar_t	TCHAR
char *	wchar_t *	LPTSTR
const char *	const wchar_t *	LPCTSTR

[표1.3] 문자 타입

범용타입은 TCHAR.h 파일에 정의된 조건부 컴파일문에 따라 환경에 따라 적절하게

타입 변환이 발생한다.

- 3) 상수 문자나 상수 문자열을 사용할 때는 반드시 `_TEXT` 키워드를 사용한다.
 상수 문자나 상수 문자열 사용시 컴파일러는 해당 문자열이 DBCS 형식의 문자(문자열인지) Unicode 형식의 문자(문자열)인지를 구분할 수 있어야 한다. 이 때 해당 키워드를 사용함으로 `TCHAR.h` 파일에 정의된 조건부 컴파일문에 따라 적절하게 변환된다.

```
#include <stdio.h>
#include <tchar.h>

void main()
{
    TCHAR* str = _TEXT("TEST");
    TCHAR ch = _TEXT('A');
}
```

- 4) 문자열을 다루는 함수들도 범용함수를 사용한다.

DBCS	Unicode	범용함수
<code>strlen</code>	<code>wcslen</code>	<code>_tcslen</code>
<code>strcpy</code>	<code>wcscpy</code>	<code>_tcscpy</code>
<code>strcat</code>	<code>wcscat</code>	<code>_tcscat</code>
<code>strcmp</code>	<code>wcscmp</code>	<code>_tcscmp</code>

[표1.4] 문자열을 다루는 함수

1.3.3 핸들(Handle)

핸들(Handle)이란 객체에 붙여진 번호이며 문법적으로는 32비트(또는 64비트)의 의미없는 정수값이다. 윈도우에서는 여러 가지 종류의 핸들이 사용되고 있다. 표1의 모듈에는 여러종류의 객체들이 존재하고(예를 들어 Pen, Brush, Window) 이들 객체를 제어하기 위한 값이 필요한데 이를 핸들이라고 한다.

핸들의 특징은 아래와 같다.

- 정수값이며, 사용목적은 객체를 구분하기 위함이다.
같은 종류의 핸들끼리는 절대로 중복된 값을 가지지 않는다. 물론 다른 종류의 핸들끼리는 중복된 값을 가질 수도 있다.
- 운영체제가 랜덤하게 발급해주는 정수값이다. 사용자는 발급된 정수값을 저장해 사용하면 된다.
예를 들어 Pen을 만들면 운영체제가 Pen을 만들고 식별할 수 있는 핸들값을 생성하여 리턴해준다. 사용자는 이 핸들을 잘 보관해 두었다가 해당 Pen을 제어할 때 사용하면 된다.
- 정수형이며, 그 실제값은 의미가 없다. 단순히 표식일 뿐이다.
- 예외없이 모든 핸들은 접두어 H로 시작되며 핸들값을 저장하기 위해 별도의 데이터 형을 정의해 두고 있으며, 모두 부호없는 정수형이다.

데이터형	의미
HWND	윈도우에 대한 핸들
HCUSOR	커서에 대한 핸들
HICON	아이콘에 대한 핸들
HMENU	메뉴에 대한 핸들

[표1.5] 핸들 예

1.3.4 인스턴스(Instnace)

프로그램은 명령들이 나열된 코드 영역(Code Segment)과 데이터를 보관하는 데이터 영역(Data Segment)으로 구분된다.

동일한 프로그램에 코드 영역까지 별도의 메모리를 할당하면 메모리만 낭비하게 된다. 실제 메모리 상에 할당된 객체를 인스턴스라 하며 코드 영역에 대한 모듈인스턴스 데이터 영역에 대한 데이터 인스턴스가 있다.

인스턴스라는 말은 클래스가 메모리에 실제로 구현된 실체를 의미한다. 윈도우용 프로그램은 여러 개의 프로그램이 동시에 실행되는 멀티태스킹 시스템일 뿐만 아니라 하나의 프로그램이 여러 번 실행될 수 있다. 이 때 실행되고 있는 각각의 프로그램을 프로그램 인스턴스라고 하며 간단히 줄여서 인스턴스라고 한다. 예를 들어 메모장이 두 번 실행되

어 있다고 해 보자(그림 1.2참조)

이 때 두 프로그램은 모두 메모장이지만 운영체제는 각각 다른 메모리를 사용하는 다른 프로그램으로 인식한다. 이 때 각 메모장은 서로 다른 인스턴스 핸들을 가지며 운영체제는 이 인스턴스 핸들값으로 두 개의 메모장을 서로 구별한다.



[그림 1.2] Instance

1.3.5 리소스(Resource)

리소스란 메뉴, 아이콘, 커서, 비트맵 등 사용자 인터페이스를 구성하는 자원들의 정적 데이터를 말한다. 프로그램 실행 중 변경되지 않는 정형화된 데이터로 C나 C++같은 언어로 기술하지 않고 리소스 스크립트에 의해 정의된다.

```
IDR_MAINFRAME MENU PRELOAD DISCARDABLE

BEGIN

    POPUP "파일(&F)"

    BEGIN

        MENUITEM "새 파일(&N)WtCtrl+N",          ID_FILE_NEW

        MENUITEM "열기(&O)...WtCtrl+O",          ID_FILE_OPEN

        MENUITEM "저장(&S)WtCtrl+S",              ID_FILE_SAVE

        MENUITEM "다른 이름으로 저장(&A)...",     ID_FILE_SAVE_AS
```

```

MENUITEM SEPARATOR

MENUITEM "인쇄(&P)...WtCtrl+P",      ID_FILE_PRINT

MENUITEM "인쇄 미리 보기(&V)",      ID_FILE_PRINT_PREVIEW

MENUITEM "인쇄 설정(&R)...",      ID_FILE_PRINT_SETUP

MENUITEM SEPARATOR

MENUITEM "최근 파일",      ID_FILE_MRU_FILE1,GRAYED

MENUITEM SEPARATOR

MENUITEM "종료(&X)",      ID_APP_EXIT

END

END

```

리소스는 프로그램 코드와 분리하여 작성되며 자체 컴파일 과정을 갖는다. 리소스 스크립트 파일(.RC)은 리소스 컴파일러(RC.exe)에 의해 이진화된 리소스 파일(.RES)이 만들어진다.

리소스를 별도로 정의하는 이유는 메모리를 효율적으로 사용하기 위함이다. 리소스에는 정적인 데이터가 있기 때문에 일반 변수와는 다른 메모리 관리를 한다. 보통 리소스 데이터는 필요한 시점에 파일로부터 로딩이 되며 여러 개의 프로그램이 실행되어 메모리가 부족시 리소스 데이터가 할당된 메모리 블록을 이동(Moveable) 시키거나 폐기(Discardable) 한다.

2. WinMain

2.1 _tWinMain

이번 장부터 윈도우 API를 사용하여 예제를 만들어 볼 것이다. 본 장에서는 프로그램의 시작점인 엔트리 포인트(Entry Point) 함수를 분석해 보고자 한다.

```
#include <windows.h>
#include <tchar.h>
int WINAPI _tWinMain( HINSTANCE hInst, HINSTANCE hPrev,
                     LPTSTR lpCmdLine, int nShowCmd)
{
    MessageBox(0, TEXT("Hello,API"),TEXT("First"), MB_OK);
    return 0;
}
```

예제 2.1 WinMain 함수

2.1.1 헤더 파일

첫 행을 보면 windows.h 가 인클루드되어 있다. 하나의 헤더 파일에 모든 API함수들의 원형과 사용하는 상수들이 정의가 되어 있기 때문에 windows.h 만 포함하면 된다. 물론 특별한 경우에는 해당하는 헤더 파일을 포함해야 하지만 예제 수준에서는 windows.h 만 포함하면 거의 문제가 없다.

Windows.h 파일은 기본적으로 데이터 타입, 함수 원형, 매크로 상수 등을 정의하며 그 외 윈도우 프로그래밍에 필요한 보조 헤더 파일을 포함하고 있다.

tchar.h 파일은 앞 장에서 말한 대로 범용타입을 사용하기 위한 헤더 이다.

2.1.2 시작점

윈도우 프로그램에서의 엔트리 포인트 함수는 환경에 따라 다양한 키워드가 사용될 수 있다.

엔트리 포인트 함수	내용
------------	----

WinMain	유니코드가 아닌 환경
wWinMain	유니코드 환경
_tWinMain	매크로이며 범용적인 형태

엔트리 포인트 함수

각 환경에 맞는 엔트리 포인트 함수를 사용해야 하며 이것이 제대로 해결 되지 않을 시 아래와 같은 링크에러가 발생한다.

(참조 위치 : __tmainCRTStartup 함수)에서 확인하지 못했습니다.

함수명 앞에 있는 WINAPI 지정자는 윈도우의 표준 호출 규약인 __stdcall을 사용한다는 의미이며 자세한 내용은 아래에서 설명하도록 하겠다.

```
//minwindef.h

#define WINAPI      __stdcall
```

2.1.3 리턴 타입과 인자

인자의 의미는 표 2.2 와 같다.

인자	의미
hInstance	프로그램의 인스턴스 핸들
hPrev	바로 앞에 실행된 현재 프로그램의 인스턴스 핸들, 없을 경우는 NULL이며 Win32에서는 항상 NULL이다. 16비트 호환성을 위한 인자이므로 변수명을 선언하지 않아도 무방하다.
lpCmdLine	명령행으로 입력된 프로그램 인자이다.
nShowCmd	프로그램이 실행될 형태이며, 최소화, 보통, 최대화 모양 등이 전달된다.

엔트리 포인트 함수 인자

API 함수들 중 HINSTANCE 타입을 요구하는 경우가 있다. 우선 HINSTANCE란 프로그램 자체를 일컫는 정수값이며 응용 프로그램의 인스턴스 핸들이라고 표현할 수 있다. 프로그램 내부에서 자기 자신을 가리키는 1인칭 대명사이다.

실제 타입은 void* 형태로 되어 있으며 아래함수를 통해 얻어 올 수 있다.

```
HINSTANCE hCurInst = GetModuleHandle(0);
```

2.1.2 MessageBox 함수

1번째 파라미터는 윈도우의 핸들, 2번째 파라미터는 메시지 출력 내용, 3번째 파라미터는 타이틀바 문자열, 4번째 파라미터는 버튼의 모양 및 ICON 모양을 나타내며 winapifamily.h 파일에 정의되어 있다.

```
//#include <winapifamily.h>
// *
// * MessageBox() Flags
// */
#define MB_OK 0x00000000L
#define MB_OKCANCEL 0x00000001L
#define MB_ABORTRETRYIGNORE 0x00000002L
#define MB_YESNOCANCEL 0x00000003L
#define MB_YESNO 0x00000004L
#define MB_RETRYCANCEL 0x00000005L
#if(WINVER >= 0x0500)
#define MB_CANCELTRYCONTINUE 0x00000006L
#endif /* WINVER >= 0x0500 */

#define MB_ICONHAND 0x00000010L
#define MB_ICONQUESTION 0x00000020L
#define MB_ICONEXCLAMATION 0x00000030L
#define MB_ICONASTERISK 0x00000040L
...
```

2. 2 함수 호출 규약

일반적으로 하나의 함수를 어떻게 컴파일 되도록 하느냐에 따른 호출 규약이 있다. 몇 가지만 살펴보도록 하자.

호출 규약	인자 전달	스택 제거	기타
-------	-------	-------	----

<code>_cdecl</code>	Right → Left	호출자가 인자 제거	C/C++ 함수의 기본 호출 규약
<code>_stdcall</code>	Right → Left	호출된 함수가 인자 제거	대부분의 System 함수가 사용
<code>this</code>	Right → Left	호출자가 인자 제거	C++ 클래스의 멤버 함수가 사용

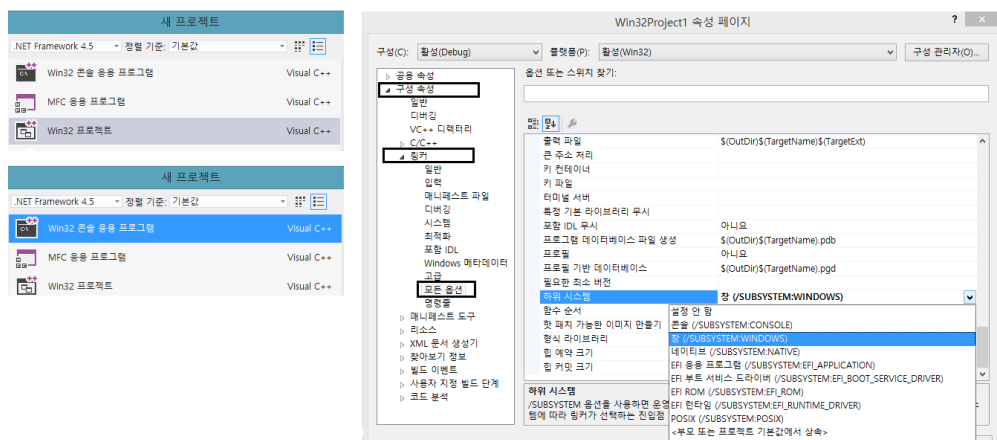
함수 호출 규약

`_stdcall`은 윈도우 프로그램에서 사용하는 대부분의 API가 사용하는 방식이다. `_cdecl`과의 차이점은 호출된 함수가 스택에서 인자를 제거한다. 이러한 이유 때문에 `_cdecl` 방식보다 메모리 사용이 효율적이다. Pascal 언어가 사용하던 방식이라서 Pascal 호출 규약이라고도 한다.

대다수의 윈도우 프로그램에서 사용하는 함수는 `_stdcall` 방식을 사용한다. WINAPI, CALBACK, APIENTRY 등의 매크로는 `_stdcall` 방식의 매크로이다.

2. 3 exe 타입

Microsoft Visual Studio를 사용하여 프로젝트를 만들 때 일반적으로 2가지 방식을 사용한다. 1번째는 윈도우 프로그래밍을 위한 Win32 프로젝트이고 2번째는 콘솔 프로그래밍을 위한 Win32 콘솔 응용프로그램이다. 생성된 프로젝트의 차이점은 어떠한 하위 시스템을 선택하느냐에 따른 부분이다. 그 외 차이는 없다.



[그림 2.1] exe 타입

그림 2.1에 보면 생성된 프로젝트의 하위 시스템을 변경할 수 있는데 이를 통해 .exe 파일을 실행하는 방법을 지시할 수 있다. 그리고 선택한 하위 시스템에 따라 링커가 선택하는

진입점 기호(또는 진입점 함수)가 결정되게 된다.

CUI(Console User Interface) – Win32 콘솔 응용프로그램

- Entry Point 수행 전 Console 창이 실행된다.
- Entry Point 함수는 main() 또는 wmain() 이다.
- Link Option 은 /SUBSYSTEM:CONSOLE 이다.

GUI(Graphic User Interface) – Win32 프로젝트

- Entry Point 수행 전 Console 창이 생성되지 않는다.
- Entry Point 함수는 WinMain(), wWinMain() 이다.
- Link Option 은 /SUBSYSTEM:WINDOWS 이다.

따라서 프로젝트를 목적과 다르게 생성한 경우는 속성 페이지에서 하위 시스템을 변경하면 된다. 또 다른 방법은 아래와 같이 코드 상단에 전처리 구문을 사용하여 변경할 수 있다.

```
#pragma comment (linker, "/subsystem:windows")  
#pragma comment (linker, "/subsystem:console")
```

2. 4 데이터 타입

API에서 사용되는 타입들은 아래와 같이 대부분 대문자로 구성되어 있다.

```
/* minwindef.h *****/  
...  
typedef unsigned long    DWORD;  
typedef int              BOOL;  
typedef unsigned char    BYTE;  
typedef unsigned short   WORD;  
...
```

또한 구조체 타입은 아래와 같은 형식으로 구현되어 있다. 따라서 포인터 변수 선언 시 2가지 방법으로 사용 가능하다.

- POINT * pt1;

- PPOINT pt2;

```
/* windef.h *****/
...
typedef struct tagPOINT
{
    LONG x;
    LONG y;
} POINT, *PPOINT, NEAR *NPPOINT, FAR *LPPOINT;
...
```

사용되는 특정 문자의 의미를 알면 타입을 구별하기가 쉽다.

키워드	내용
P	Pointer를 나타낸다.
LP	Win32에서 L 키워드는 의미가 없다. PSTR 과 LPSTR 은 동일한 타입이다.
T, t	범용 타입이다. LPSTR(char*) 과 LPTSTR(TCHAR*) 은 다르다.
W	Unicode Type이다.
C	const 를 의미한다. LPCTSTR
STR	문자열을 의미한다. PSTR(char*)
H	객체의 핸들을 의미한다. HWND, HPEN

3. Window

3. 1 윈도우 생성 단계

전 장에서 보았던 WinMain 함수에서 하는 가장 중요한 일은 메인 윈도우를 생성하는 일이다. Console Application은 콘솔창을 통해서 사용자로 부터 입력을 받고 출력물을 내보낸다. 하지만 Window Application은 콘솔창이 생성되지 않는다. 사용자로 부터 입력을 받거나 출력물을 내보내기 위해서 윈도우를 생성해야 한다. 그리고 윈도우를 생성하기 위해서는 반드시 거쳐야 하는 절차가 있다.



3.1.1 윈도우 클래스 만들기

모든 윈도우는 윈도우 클래스로부터 만들어 진다.

윈도우를 만들기 위해서는 윈도우 클래스를 먼저 만들어야 한다. 윈도우 클래스는 윈도우의 특징(배경색, Icon, 커서등)을 정의하는 구조체이며 winuser.h에 아래와 같이 정의되어 있다. 또한 만들어진 윈도우 클래스는 시스템에 등록하여야만 사용할 수 있다. 결국 하나의 윈도우를 만들기 위해서는 윈도우 클래스 생성 및 등록과정을 거쳐야 한다.

윈도우 클래스를 만들기 위해서는 WNDCLASS 와 WNDCLASSEX 를 사용할 수 있으며 후자는 확장된 구조체이다(cbSize, hIconSm 멤버가 추가 됨). 여기서는 확장 구조체 기반으로 설명하고자 한다.

```

/*winuser.h *****/
typedef struct tagWNDCLASSEXW {
    UINT        cbSize;

    /* Win 3.x */

    UINT        style;

    WNDPROC      lpfnWndProc;

    int         cbClsExtra;

    int         cbWndExtra;

    HINSTANCE    hInstance;

    HICON        hIcon;

    HCURSOR      hCursor;

    HBRUSH       hbrBackground;

    LPCWSTR      lpstrMenuName;

    LPCWSTR      lpstrClassName;

    /* Win 4.0 */

    HICON        hIconSm;
} WNDCLASSEXW, *PWNDCLASSEXW, NEAR *NPWNDCLASSEXW, FAR *LPWNDCLASSEXW;

...

#ifdef UNICODE
typedef WNDCLASSEXW WNDCLASSEX;
#else

```

해당 구조체는 12개나 되는 멤버를 가지고 있다. 각 멤버 의미는 다음과 같다.

- cbSize
구조체 크기를 지정한다.
ex) WNDCLASSEX wcx;
 wcx.cbSize = sizeof(WNDCLASSEX);
- style
윈도우의 스타일을 정의한다. 이 멤버가 가질 수 있는 값은 많지만 가장 많이 사용하는 값이 CS_HREDRAW와 CS_VREDRAW 이다. 이 두 값을 OR 연산자(|)로 연결하여 사용한다. 이 스타일은 윈도우의 수직(또는 수평) 크기가 변경 될 경우 윈도우를 다시 그린다는 뜻이다.

ex) `wcex.style = CS_HREDRAW | CS_VREDRAW`

- `lpfnWndProc`

윈도우의 메시지 처리 함수(윈도우 프로시저) 를 지정한다. 해당 함수 이름은 마음대로 정할 수 있지만 관습적으로 `WndProc` 이라는 이름을 사용한다.

윈도우 프로시저는 메시지가 발생할 때마다 호출되는 함수이며 모든 메시지를 처리하게 된다.

ex) `wcex.lpfnWndProc = WndProc; // WndProc이라는 이름의 윈도우 프로시저가
// 있다고 가정`

- `cbClsExtra`

클래스 정보를 저장할 수 있는 예약 영역이다. 사용하지 않을 경우는 0으로 지정한다.

ex) `wcex.cbClsExtra = 0;`

- `cbWndExtra`

윈도우 정보를 저장할 수 있는 예약 영역이다. 사용하지 않을 경우 0으로 지정한다.

ex) `wcex.cbWndExtra = 0;`

- `hInstance`

윈도우 프로시저를 담은 모듈의 핸들이며 `WinMain`의 첫번째 인자로 전달된 값을 대입하면 된다.

ex) `wcex.hInstance = hInstance;`

- `hIcon`

윈도우에서 사용할 큰 아이콘(32*32)의 핸들을 지정한다. `LoadIcon` 함수를 사용하여 아이콘을 읽어와 대입한다. 이 때 읽어오는 정보는 사용자가 직접 만든 아이콘이나 윈도우가 디폴트로 제공하는 아이콘을 사용할 수 있다. 아래 예는 전자에 해당된다.

ex) `wcex.hIcon
= LoadIcon(hInstance, MAKEINTRESOURCE(IDI_WIN32PROJECT1));`

- `hCursor`

윈도우에서 사용할 커서를 지정한다. `LoadCursor` 함수를 사용하여 커서를 읽어와 대입한다. 이 때 읽어오는 정보는 사용자가 직접 만든 커서나 윈도우가 디폴트로 제공하는 커서를 사용할 수 있다. 아래 예는 후자에 해당된다.

ex) `wcex.hCursor = LoadCursor(NULL, IDC_ARROW);`

- `hbrBackground`

윈도우의 배경 색상을 표현하기 위한 브러시를 지정한다. 사용자가 직접 만들 브러시를 사용할 수 있고 윈도우에서 기본적으로 제공되는 브러시를 사용할 수 있다. 예제는 후자에 해당된다.

`wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);`

- `lpszMenuName`

윈도우에 부착할 메뉴를 지정한다. 메뉴는 일반적으로 리소스 에디터에 의해 별도로 만들어진다. 사용하지 않을 경우 NULL로 지정한다. 아래 예는 리소스에 등록된 메뉴를 사용하는 경우이다.

ex) `wcex.lpszMenuName = MAKEINTRESOURCE(IDC_WIN32PROJECT1);`

- `lpszClassName`

윈도우 클래스의 이름을 문자열로 지정한다. 해당 이름은 윈도우가 생성될때 원하는 클래스정보를 가져오는 키가 된다.

ex) `wcex.lpszClassName = TEXT("First");`

- `hIconSm`

윈도우에서 사용할 작은 아이콘(16*16)을 지정한다. 0을 지정하면 큰 아이콘을 축소해서 사용한다.

ex) `wcex.hIconSm = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));`

참고로 윈도우 클래스는 3가지 종류가 있다.

- System Class

운영체제가 부팅될 때 시스템에 의해 등록되는 클래스

- Application Global Class

DLL 혹은 exe 에 의해 등록되고 다른 모듈에서 사용할 수 있는 클래스

- Application Local Class

Exe 에 의해 등록되고 등록된 모듈에서만 사용할 수 있다. 우리가 만들고자 하는 일반적인 클래스가 여기에 해당된다.

3.1.2 윈도우 클래스 등록하기

윈도우 클래스 구조체를 정의하였다면 RegisterClassEx 함수를 호출하여 윈도우 클래스를 등록한다. 어떠한 윈도우 클래스 구조체로 정의하였는지에 따라 적절한 함수를 사용한다.

해당 함수는 ATOM 값을 리턴하는 데 실패할 경우 0을 리턴한다. 리턴된 값은 윈도우를 생성할 때 사용될 수 있으나, 일반적으로 WNDCLASSEX 구조체 멤버인 lpzClassName 을 사용한다.

아래 함수는 모두 등록에 실패할 경우 0을 리턴한다.

```
// typedef WORD ATOM;  
  
ATOM RegisterClass( CONST WNDCLASS *lpwndClass);  
ATOM RegisterClassEx(CONST WNDCLASSEX* lpwcx);
```

3.1.3 윈도우 만들기

윈도우 클래스를 등록한 후에는 등록된 윈도우 클래스를 기본으로 아래 함수를 호출하여 윈도우를 생성한다.

참고로 나중에 보게 될 자식 윈도우를 만들 때도 아래 함수가 사용된다.

```
HWND CreateWindowW(  
    _In_opt_ LPCWSTR lpClassName,  
    _In_opt_ LPCWSTR lpWindowName,  
    _In_ DWORD dwStyle,  
    _In_ int X,  
    _In_ int Y,  
    _In_ int nWidth,  
    _In_ int nHeight,  
    _In_opt_ HWND hWndParent,
```

```

    _In_opt_ HMENU hMenu,
    _In_opt_ HINSTANCE hInstance,
    _In_opt_ LPVOID lpParam);

#ifdef UNICODE
#define CreateWindow CreateWindowW
#else
#define CreateWindow CreateWindowA
#endif // !UNICODE

```

```
CreateWindowEx( DWORD dwExStyle, ... )
```

확장 함수로 dwExStyle 멤버가 추가 되어 있다.

ex) WS_EX_TOPMOST 등의 속성을 지정할 수 있다.

- lpClassName
윈도우 클래스 이름을 지정한다.(등록된 윈도우 클래스 중 하나의 이름을 사용)
- lpWindowName
윈도우 캡션바에 나타날 문자열을 지정한다.
- dwStyle
만들고자 하는 윈도우의 형태를 지정한다. 다양한 상수가 정의되어 있고 OR 연산자를 사용하여 선택하여 사용한다.
정의된 내용은 크기 조절 가능 여부, 타이틀 바 생성 여부, 스크롤 바 유무 등이 있다
주로 사용되는 형태는 WS_OVERLAPPEDWINDOW 매크로로 지정되어 제공된다.
(시스템 메뉴, 최대, 최소 버튼, 타이틀 바, 경계선 등을 가진 윈도우 형태)

```

#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX)

```

특정 스타일을 제거하려면 &~ 연산을 사용하고,
특정 스타일을 추가하려면 | 연산자를 사용한다.

ex) WS_OVERLAPPEDWINDOW &~ WS_THICKFRAME // 경계선 제거 예

- x, y, nWidth, nHeight
윈도우의 위치(x,y)와 크기(nWidth, nHeight)를 지정하며 픽셀 단위를 사용한다.
CW_USEDEFAULT 를 지정하면 시스템이 임의로 위치와 크기를 지정한다.
- hWndParent
부모 윈도우가 있을 경우 부모 윈도우의 핸들을 지정한다. 없을 경우 NULL을 지정하며, 이 때 자신이 최상위 윈도우가 된다.
- hMenu
윈도우에서 사용할 메뉴의 핸들을 지정한다. 자식 윈도우를 만들 때는 자식윈도우의 ID값을 지정하는 용도로 사용된다.
- hInstance
WinMain의 인수로 전달된 응용 프로그램의 instance handle을 지정한다. XP 이후 부터는 이 값은 무시된다.
- lpParam
생성 인자를 전달할 목적으로 사용된다. 부모 윈도우 생성시는 주로 NULL을 사용한다.

해당 함수가 성공하면 생성된 윈도우 핸들(HWND)을 실패하면 0을 리턴한다.

3.1.4 윈도우 화면에 출력하기

CreateWindow 함수로 만든 윈도우는 아직 화면에 보이지 않는다. 아래의 함수는 생성된 윈도우를 화면에 보여주게 한다.

```
ShowWindow(    _In_ HWND hWnd,    _In_ int nCmdShow );
```

첫번째 인자는 화면에 보여 줄 윈도우의 핸들이다.(CreateWindow의 리턴 값) 두 번째 인자는 출력하는 방법을 지정하며 다음과 같은 매크로 상수들이 정의되어 있다.

매크로 상수	의미
SW_HIDE	윈도우를 숨긴다.

SW_MINIMIZE	윈도우를 최소화하고 활성화시키지 않는다.
SW_RESTORE	윈도우를 활성화시킨다.
SW_SHOW	윈도우를 활성화하여 보여준다.
SW_SHOWNORMAL	윈도우를 활성화하여 보여준다.

ShowWindow 를 호출 한 후 보통 아래 함수를 호출한다. 아래 함수는 윈도우가 보여지고 난 후 최초에 발생하는 WM_PAINT 메시지를 즉시 처리하기 위해 사용된다. 지금은 이 부분은 Device Context 개념에서 설명되어 진다.

```
UpdateWindow(_In_ HWND hWnd);
```

3.1.5 윈도우 생성 코드

위에서 설명한 코드를 순서대로 작성하면 화면에 윈도우가 출력된다. 아래 예제를 통해 다시 한번 흐름을 이해해 보도록 하자.

```
#include <windows.h>
#include <tchar.h>

#define MAX_LOADSTRING 100

TCHAR szTitle[MAX_LOADSTRING] = _TEXT("First Sample"); // 제목 표시줄
TCHAR szWindowClass[MAX_LOADSTRING] = _TEXT("BIT"); // 기본 창 클래스

// 함수: MyRegisterClass()
// 목적: 윈도우 클래스 생성 및 등록
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEX wcex;

    wcex.cbSize = sizeof(WNDCLASSEX);

    wcex.style = CS_HREDRAW | CS_VREDRAW;

    wcex.lpfnWndProc = DefWindowProc;

    wcex.cbClsExtra = 0;
```

```

        wcex.cbWndExtra          = 0;

        wcex.hInstance          = hInstance;

        wcex.hIcon              = LoadIcon(NULL, IDI_APPLICATION);

        wcex.hCursor            = LoadCursor(NULL, IDC_ARROW);

        wcex.hbrBackground      = (HBRUSH)(COLOR_WINDOW+1);

        wcex.lpszMenuName        = 0;

        wcex.lpszClassName       = szWindowClass;

        wcex.hIconSm            = 0;

        return RegisterClassEx(&wcex);
    }

// 함수: InitInstance(HINSTANCE, int)
// 목적: 윈도우 창을 만들고 핸들을 리턴
HWND InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;

    hWnd = CreateWindow( szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

    if (!hWnd) {
        return 0;
    }

    ShowWindow(hWnd, nCmdShow);

    UpdateWindow(hWnd);

    return hWnd;
}

int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpCmdLine, int nShowCmd)
{
    //1,2 윈도우 클래스 생성 및 등록

```

```

MyRegisterClass(hInst);

//3,4 윈도우 생성 및 화면 출력
HWND hwnd = InitInstance (hInst, nShowCmd);

if (hwnd == 0)
{
    return FALSE;
}

MessageBox(0, TEXT("Hello,API"),TEXT("First"), MB_OK);

return 0;
}

```

예제 3.1 WinMain함수

3.2 윈도우 핸들(Window Handle)

3.2.1 윈도우 핸들

앞 예제에서 보았던 ShowWindow() 함수는 윈도우를 화면에 다양한 형태로 출력할 때 사용한다. MoveWindow() 함수는 윈도우를 이동할 때 사용하며 GetWindowRect() 함수는 윈도우의 위치와 크기를 얻을 때 사용한다.

```

BOOL ShowWindow(HWND hWnd, int nCmdShow)
BOOL MoveWindow(HWND hWnd, int X, int Y, int nWidth, int nHeight,
                BOOL bRepaint);
BOOL GetWindowRect(HWND hwnd, LPRECT lpRect);

```

이 외 다수의 API 함수들은 객체를 제어하기 위한 목적으로 제공되며, 위에서 제시한 함수들은 그 중 윈도우를 제어하기 위한 목적의 함수이다. 따라서 첫번째 인자로 제어할 대상의 윈도우 핸들을 요구한다.

따라서 윈도우 핸들은 해당 윈도우를 제어하기 위한 필수적 정보이며, 이를 알고 있다면 다양한 API 함수를 사용하여 원하는 명령들을 내릴 수 있다.

3.2.1 윈도우 핸들 얻기

아래 함수를 사용하면 실행중인 윈도우 핸들을 얻을 수 있다.


```
HWND FindWindow(LPCTSTR lpCalssName, LPCTSTR lpWindowName);
```

1번째 인자로 윈도우를 만든 윈도우 클래스 이름, 2번째 인자로 윈도우의 캡션바에 있는 문자열을 지정한다. 만약 윈도우 클래스 이름을 모르거나 캡션바가 없는 경우라면 해당 항목을 0을 지정하면 된다.(반드시 둘 중 하나는 지정해야 한다.)

```
#include <windows.h>
#include <tchar.h>

int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpCmdLine, int nS
howCmd)
{
    HWND hwnd = FindWindow(0, _TEXT("계산기"));
    if( hwnd == 0 ) {
        MessageBox(0, _TEXT("계산기 핸들을 얻을 수 없습니다."),
            _TEXT("알림"), MB_OK|MB_ICONERROR);
        return 0;
    }
    else {
        TCHAR temp[20];
        wsprintf(temp, _TEXT("계산기 핸들 : %d"), hwnd);
        MessageBox(0, temp, _TEXT("알림"), MB_OK);
    }
    return 0;
}
```

예제 3.2 윈도우 핸들 얻기

3.2.2 윈도우 제어하기

앞서 설명했던 함수들을 이용하여 윈도우를 제어할 수 있다. 아래 예제를 확인해 보자.

```
#include <windows.h>
```

```

#include <tchar.h>

int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpCmdLine, int nShowCmd)
{
    HWND hwnd = FindWindow(0, _TEXT("계산기"));

    //계산기 클래스 이름 및 Rect 얻기
    TCHAR name[60];
    RECT rcCalc;
    GetClassName(hwnd, name, sizeof(name));
    GetWindowRect(hwnd, &rcCalc);

    //정보 출력
    TCHAR info[256];
    wprintf(info, _TEXT("클래스 명 : %s\n위치 : %d,%d ~ %d,%d"), name,
        rcCalc.left, rcCalc.top, rcCalc.right, rcCalc.bottom);
    MessageBox(0, info, TEXT("계산기 정보"), MB_OK);

    //계산기 이동하기
    MoveWindow(hwnd, 10, 10, rcCalc.right - rcCalc.left, rcCalc.bottom -
        rcCalc.top, TRUE);

    //계산기 숨기기
    MessageBox(0, _TEXT("계산기"), TEXT("계산기 숨기기"), MB_OK);
    ShowWindow(hwnd, SW_HIDE);

    //계산기 보이기
    MessageBox(0, _TEXT("계산기"), TEXT("계산기 보이기"), MB_OK);
    ShowWindow(hwnd, SW_SHOW);

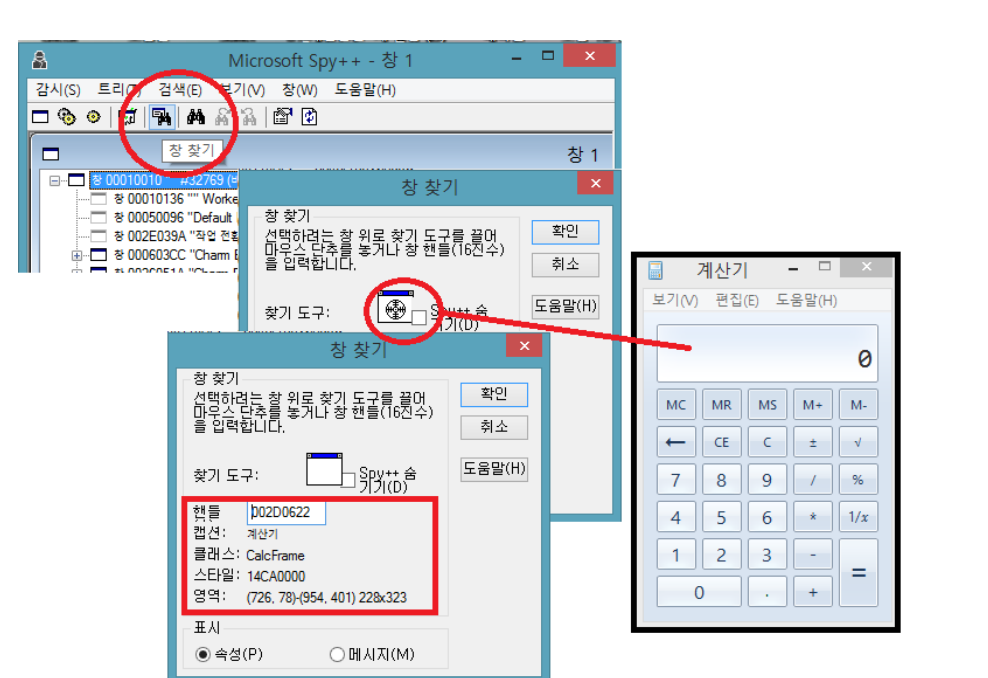
    //계산기 종료하기
    MessageBox(0, _TEXT("계산기"), TEXT("계산기 종료하기"), MB_OK);
    SendMessage(hwnd, WM_CLOSE, 0, 0);
}

```

```
    return 0;
}
```

예제 3.3 윈도우 제어하기

이러한 경우는 FindWindow의 1번째 인자를 통해서 핸들을 얻을 수 있다. 이 때 사용되는 정보는 윈도우 클래스 이름이다. 해당 정보는 VC++ 에서 제공되는 SPY++ 라는 도구를 사용하면 얻을 수 있다.



SPY++을 이용하여 윈도우 정보 얻기

간단한 사용 방법은 그림 3.2를 보면 창 찾기라는 제목의 툴바를 선택하면 창 찾기 팝업 윈도우를 생성하게 된다. 여기서 찾기 도구에 마우스 우클릭을 한 후 얻고자 하는 윈도우의 영역으로 드래그 하면 사각형 테두리에 있는 정보를 보여준다. (해당 윈도우의 핸들값, 캡션, 클래스명, 윈도우 스타일, 영역 등의 정보)

그 외 SPY++은 해당 윈도우에서 발생하는 이벤트 정보를 볼 수도 있고, 그 외 다양한 기능들을 제공하고 있다.

아래 예제는 타이틀바가 없는 작업바를 제어하는 코드이다.

```
#include <windows.h>
#include <tchar.h>

int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpCmdLine, int nS
howCmd)
{
    HWND hWnd = FindWindow(TEXT("Shell_TrayWnd"), 0);
    if( IsWindowVisible( hWnd ) )
        ShowWindow(hWnd, SW_HIDE);
    else
        ShowWindow(hWnd, SW_SHOW);
    return 0;
}
```

예제 3.4 작업바 제어하기

3.3 윈도우 클래스 정보

윈도우 클래스는 3가지 종류가 있다.

- 시스템 전역 클래스(System global class)
운영체제가 부팅될 때 등록되며 주로 컨트롤을 만들 때 사용된다.
Button, edit, scrollbar, listbox, 등이 그 예이며, 따라서 버튼 윈도우를 만들고 싶다면 등록된 "button"이라는 클래스명으로 참조해서 생성하면 된다.

- 응용 프로그램 전역 클래스(Application global class)
주로 DLL에 의해 등록되며 다른 프로그램에서 사용 될 수 있다.
뒤에서 보게 될 공통 컨트롤들이 이에 해당되며, 해당 클래스정보는 comctl32.dll 에 저장되어 있다.
- 응용 프로그램 로컬 클래스(Application local class)
응용 프로그램 자신이 메인 윈도우나 차일드 또는 커스텀 컨트롤을 만들기 위해 프로그램 상단에 등록하는 클래스이며 해당 프로그램이 종료되면 윈도우 클래스도 같이 파괴된다.
등록된 곳에서만 사용할 수 있다.

3.3.1 SetClassLong

WINDCLASS(EX) 구조체의 값은 윈도우 클래스를 등록할 때 정해지지만 등록된 후에도 수정할 수 있다.

<pre>// 32bit 전용 함수 : 기존 사용된 함수 DWORD GetClassLong(HWND hWnd, int nIndex); DWORD SetClassLong(HWND hWnd, int nIndex, LONG dwNewLong);</pre>
<pre>// 64bit 전용 지원 함수 : 대체된 함수 LONG_PTR GetClassLongPtr(HWND hWnd, int nIndex); LONG_PTR SetClassLongPtr(HWND hWnd, int nIndex, LONG_PTR dwNewLong);</pre>

위의 함수들은 실행 중 윈도우 클래스 정보를 획득하거나 변경할 때 사용되는 함수들이다. 과거에는 위 단의 함수들이 사용되었지만 64비트를 지원하는 아래의 함수로 완전히 대체되었다.

```
int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpCmdLine, int nShowCmd)
{
    MyRegisterClass(hInst);

    HWND hwnd = InitInstance (hInst, nShowCmd);

    if (hwnd == 0)    {                return FALSE;    }
}
```

```

//실시간 클래스 정보 수정
MessageBox(0, TEXT("실시간 정보 수정"),TEXT("First"), MB_OK);
SetClassLongPtr(hwnd, GCLP_HBRBACKGROUND, (LONG)GetStockObject(DKGRAY_
BRUSH));
InvalidateRect(hwnd, NULL, TRUE);
MessageBox(0, TEXT("Hello,API"),TEXT("First"), MB_OK);

//실시간 클래스 정보 획득
TCHAR temp[126];
DWORD flag = GetClassLongPtr(hwnd, GCLP_HBRBACKGROUND);
wsprintf(temp, _TEXT("R : %d, G : %d : B : %d"), GetRValue(flag), GetG
Value(flag), GetBValue(flag));
MessageBox(0, temp, TEXT("획득정보"), MB_OK);
return 0;
}

```

예제 3.5 ClassLong

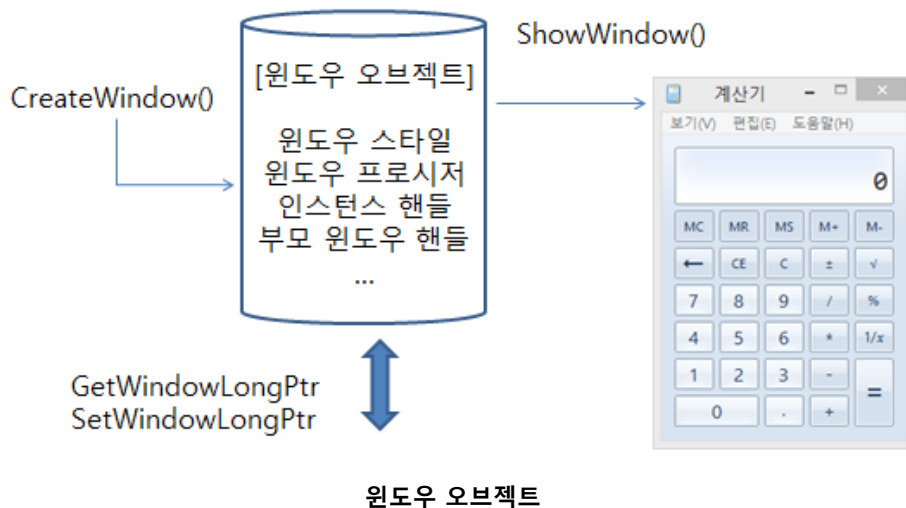
3.5 예제는 윈도우 배경색을 실시간 변경하고 변경된 정보를 확인해 보았다. 그 외 다양한 정보들을 변경하거나 얻어올 수 있다.(MSDN을 참조할 것)

3. 4 윈도우 객체(Window Object)

윈도우 객체는 운영체제가 하나의 윈도우를 관리하기 위해 생성하는 데이터 구조체이다. CreateWindow() 함수로 윈도우를 만들 때 생성되며 설정했던 윈도우의 속성들을 실행 중에 조사하거나 변경하고자 할 때는 다음 두 함수를 사용한다.

<pre> // 32bit 전용 함수 : 기존 사용된 함수 LONG GetWindowLong(HWND hWnd, int nIndex); DWORD SetWindowLong(HWND hWnd, int nIndex, LONG dwNewLong); </pre>
<pre> // 64bit 전용 지원 함수 : 대체된 함수 LONG_PTR GetWindowLongPtr(HWND hWnd, int nIndex); LONG_PTR SetWindowLongPtr(HWND hWnd, int nIndex, LONG_PTR dwNewLong); </pre>

위의 내용을 그림으로 표현하면 아래와 같다.



CreateWindow() 함수는 윈도우 오브젝트를 생성하고 ShowWindow() 함수는 생성된 윈도우 오브젝트 정보를 바탕으로 화면에 윈도우를 출력해준다.

GetWindowLongPtr() 함수는 저장된 윈도우 오브젝트에서 원하는 정보를 얻을 수 있고, SetWindowLongPtr() 함수는 저장된 윈도우 오브젝트 정보를 수정할 수 있다. 접근 가능한 정보는 아래 표와 같다.

인수	설명
GWL_EXSTYLE	확장 스타일
GWL_STYLE	윈도우 스타일
GWLP_WNDPROC	윈도우 프로시저 번지
GWLP_HINSTANCE	인스턴스 핸들
GWLP_HWNDPARENT	부모 윈도우의 핸들
GWLP_ID	ID
GWLP_USERDATA	윈도우와 관련된 사용자 데이터
GWLP_DLGPROC	대화상자 프로시저의 주소
GWLP_MSGRESULT	대화상자 프로시저의 리턴값

GWLP_USER	사용자 데이터
-----------	---------

윈도우 오브젝트 획득 인자 정보

```
BOOL fun_ModifyStyle(HWND hwnd, LONG_PTR Add, LONG_PTR Remove, BOOL bRedraw)
{
    BOOL bFlag = FALSE;
    LONG_PTR style = GetWindowLongPtr(hwnd, GWL_STYLE);
    style |= Add;
    style &= ~Remove;
    bFlag = (BOOL)SetWindowLongPtr(hwnd, GWL_STYLE, style);
    if( bFlag && bRedraw )
        SetWindowPos(hwnd, 0, 0, 0, 0, 0,
                      SWP_NOMOVE | SWP_NOSIZE | SWP_NOZORDER | SWP_DRAWFRAME);
    return bFlag;
}

int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpCmdLine, int nS
howCmd)
{
    MyRegisterClass(hInst);
    HWND hwnd = InitInstance (hInst, nShowCmd);
    if (hwnd == 0)    {
        return FALSE;
    }
    // 실시간 윈도우 객체 정보 수정
    HWND hWnd = FindWindow(0, _TEXT("계산기"));
    fun_ModifyStyle(hWnd, WS_THICKFRAME, WS_SYSMENU, TRUE);
    MessageBox(0, TEXT(""), TEXT(""), MB_OK);
    return 0;
}
```

예제 3.6 WindowLong

SetWindowPos() 함수는 윈도우의 Z-order, 위치, 크기 정보를 변경할 수 있는 함수이다.

1번째 인자는 대상 윈도우 핸들, 2,3번째 인자는 위치, 4,5번째 인자는 폭과 높이, 6번째 인자는 실제 변경여부를 flag 연산으로 설정하게 된다. 예제에서는 위치와 폭과 높이, z-order 값은 사용하지 않고, 프레임만 다시 그리는 값으로 설정되어 있다.

만약 2번째 인자에 WS_EX_TOPMOST 값을 주면 최상위 윈도우로 설정할 수 있다.

4. 메시지

4. 1 윈도우 프로시저

윈도우가 이전 운영체제인 도스와 구분되는 가장 큰 차이점은 메시지 기반의 운영체제라는 점이다. 윈도우에서 실행되는 응용 프로그램은 사용자 입력을 받기 위해 함수를 직접적으로 호출하는 경우가 없으며 시스템이 보내는 메시지를 기다릴 뿐이다.

윈도우 프로시저(Window Procedure)는 윈도우 클래스당 하나씩 배정되며 메시지에 대응하는 방식을 정의함으로써 윈도우의 행동 양식을 결정한다.

윈도우 프로시저는 아래와 같은 원형을 가진다.

***LRESULT CALLBACK WndProc(HWND hWnd, UINT iMessage,
WPARAM, wParam, LPARAM lParam);***

문법적으로 함수이며 이름은 사용자가 마음대로 바꿀 수 있다. 단 몇가지 일반 함수와 차이점이 있다.

- 리턴타입과 인자를 변경할 수 없다.
- 운영체제에 의해 호출되는 콜백함수다. 따라서 함수명 앞에 `_stdcall`을 추가한다.
- 사용자가 직접 함수의 이름으로 호출할 수 없다.

만약 호출이 필요하다면 `SendMessage` 등의 함수로 메시지를 보내 간접적으로 호출해야 한다.

인수의 내용은 아래와 같다.

인수	내용
<code>hWnd</code>	이 메시지를 받을 윈도우 핸들이다. 한 클래스로부터 여러 개의 윈도우가 만들어졌을 경우는 어떤 윈도우로 전달된 메시지인지 구분해야 하므로 이 인수가 필요하다.
<code>iMessage</code>	전달된 메시지 종류이다. <code>WM_CREATE</code> , <code>WM_PAINT</code> 등의 미리 정해진 매크로 상수값을 사용하여 어떤 메시지가 전달되었는지 구분한다.
<code>wParam</code> <code>lParam</code>	2번째 인자의 메시지별 부가적인 정보이다. 둘 다 32비트 정수값이며 메시지에 따라 이 값들의 의미는 달라진다.

윈도우 프로시저는 이 함수로 전달되는 무수히 많은 메시지들을 처리하므로 거대한 switch 문으로 구성된다. 처리하는 메시지가 많으면 많을수록 case 문이 많아질 것이며, 처리하지 않는 메시지는 반드시 DefWindowProc 으로 전달해야 한다.

일반적으로 처리된 메시지 결과로 0을 리턴한다.

[확인]

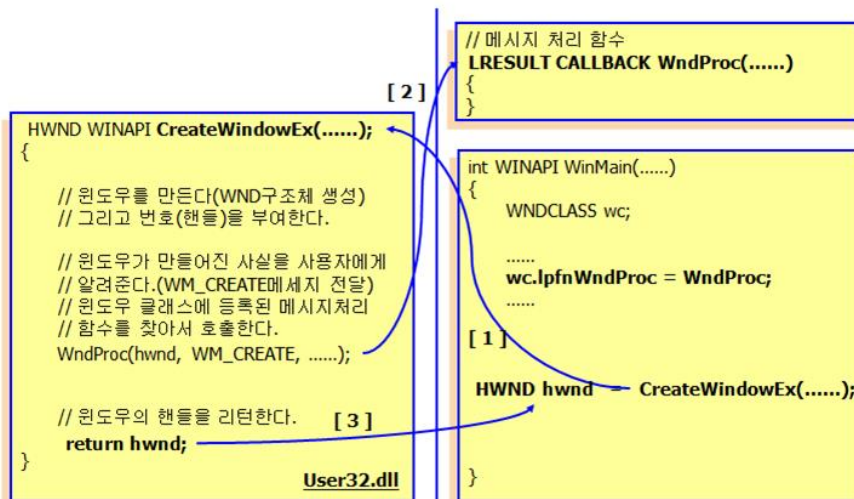
DefWindowProc 은 시스템에서 미리 제공되는 프로시저로써 일반적인 윈도우 이벤트별 기능들이 미리 구현되어 있다.

4. 2 메시지 큐

메시지는 크게 메시지 큐로 들어가는 큐 메시지와 메시지 큐에 들어가지 않고 곧바로 윈도우 프로시저로 보내지는 비큐메시지로 구분된다

4.2.1 비큐 메시지

비큐 메시지는 윈도우에게 특정 사실을 알려거나 명령을 내리기 위해 큐를 통하지 않고 바로 윈도우 프로시저로 보내지는 메시지이다.



위 그림은 WM_CREATE(대표적인 비큐메시지) 라는 메시지의 흐름이다.

- (1) 사용자가 윈도우를 만들기 위해 CreateWindowEx API를 호출하면 윈도우가 만들어 지고
- (2) 윈도우가 만들어졌다는 사실을 사용자에게 알리기 위해 WM_CREATE 라는 메시지가 메시지 처리 함수에 전달된다.
- (3) WM_CREATE 메시지 처리가 완료되면 CreateWindowEx 함수는 윈도우 핸들을 리턴 한다.

즉, WM_CREATE 메시지는 메시지 큐에 놓이지 않고 직접 윈도우 프로시저에 전달된다. (이 때 사용된 함수가 SendMessage 이다.) 이를 비큐 메시지라고 한다.

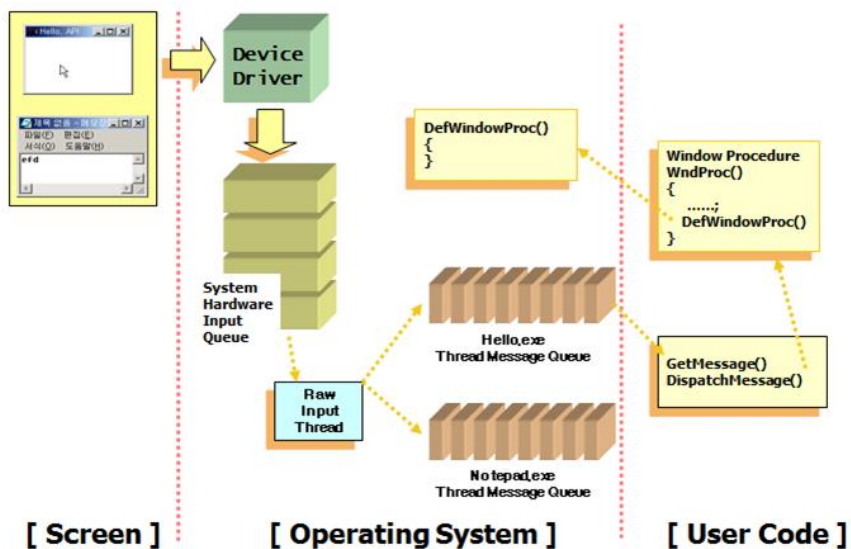
입력 메시지(마우스, 키보드 메시지)를 제외한 대부분의 메시지가 비큐 메시지이다.

시점	비큐 메시지
CreateWindow	WM_CREATE
MoveWindow	WM_SIZE, WM_MOVE
DestroyWindow	WM_DESTROY
활성 상태 변경	WM_ACTIVATE

비큐 메시지

4.2.2 큐 메시지

마우스 키보드 메시지들은 직접 윈도우 프로시저로 전달되지 않고 메시지 큐에 놓이게 되는 데 이를 큐 메시지라고 한다.



큐 메시지는 주로 사용자 입력으로 발생하는 데 WM_KEYDOWN, WM_LBUTTONDOWN 등이 대표적이며 그 외 WM_PAINT, WM_TIMER, WM_QUIT 등이 큐 메시지이다. 큐 메시지는 발생 직후 시스템 메시지 큐에 저장되어 스레드 메시지 큐로 보내지며 최종적으로 윈도우 프로시저에 의해 입력된 순서대로 처리된다.

큐에 놓인 메시지를 처리하려면 반드시 사용자가 큐에 놓여 있는 메시지를 가져오는 메시지 루프가 필요하다. GetMessage API 가 그 일을 수행한다.

```
MSG msg;
while ( GetMessage( &msg, 0, 0, 0 ) ) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

GetMessage API는 메시지 큐에 WM_QUIT 메시지가 있을 경우에만 TRUE 값을 리턴한다. 그래서 위 코드는 WM_QUIT 메시지가 큐에 들어올 때 까지는 무한 루프가 된다.

4.2.3 프로그램 종료

사용자가 만든 윈도우가 파괴될때 나오는 메시지는 WM_DESTROY 이다. 그러므로 사용자가 윈도우를 파괴 해도 프로그램은 계속 메시지 루프를 돌게 된다.(하지만 윈도우는 파괴 되었다.- 모니터 화면에는 더 이상 아무 것도 볼수 가 없다.)

윈도우를 1개 만든 경우 프로그램의 main 윈도우가 파괴 될때 프로그램을 같이 종료 되게 하려면 WM_DESTROY 메시지에서 WM_QUIT메세지를 메시지 Q에 넣는 작업을 해야 한다.

```
case WM_DESTROY:
    PostQuitMessage( 0 );
    return 0;
```

4.2.4 처리하지 않은 메시지

메시지 처리 함수에서 발생한 메시지를 처리하지 않은 경우 반드시 아래 함수로 보내서 default 처리가 되게 해야 한다.

***LRESULT CALLBACK DefWindowProc(HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam);***

해당 함수 호출시 인자는 윈도우 프로시저에 전달된 인자를 그대로 전달한다.

4. 3 완성된 API skeleton 코드

아래 코드는 지금까지 학습한 결과물이며, API를 이용한 프로그래밍 시 사용될 기본 코드이다. 앞으로 작성하는 예제의 기본 코드로 사용될 것이다.

```
#include <windows.h>
#include <tchar.h>

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )    {
        case WM_CREATE:
            return 0;
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc(hwnd, msg, wParam, lParam);
}

int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpCmdLine,
int nShowCmd)
{
    // 1. 윈도우 클래스 만들기
    WNDCLASS wc;

    wc.cbWndExtra        = 0;
    wc.cbClsExtra        = 0;
    wc.hbrBackground     = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.hCursor           = LoadCursor(0, IDC_ARROW);
```

```

wc.hIcon          = LoadIcon(0, IDI_APPLICATION);
wc.hInstance      = hInst;
wc.lpfnWndProc    = WndProc;
wc.lpszClassName  = TEXT("First");
wc.lpszMenuName   = 0;
wc.style          = 0;

// 2. 등록(레지스트리에)
RegisterClass(&wc);

// 3. 윈도우 창 만들기
HWND hwnd = CreateWindowEx( 0,          // WS_EX_TOPMOST
    TEXT("first"),          // 클래스 명
    TEXT("Hello"),          // 캡션바 내용
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, // 초기 위치
    0, 0,                  // 부모 윈도우 핸들, 메뉴 핸들
    hInst,                 // WinMain의 1번째 파라미터 (exe 주소)
    0);                   // 생성 인자

// 4. 윈도우 보여주기
ShowWindow(hwnd, SW_SHOW);
UpdateWindow(hwnd);

// 5. Message
MSG msg;
while( GetMessage( &msg, 0, 0, 0 ) )
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return 0;
}

```

예제 4.1 skeleton 코드

4. 4 PeekMessage

메시지 루프에서 제일 중요한 함수는 메시지를 가져 오는 GetMessage 함수이다. 이 함수는 스레드 메시지 큐에서 메시지를 가져오는데 메시지가 없으면 새로운 메시지가 전달될 때까지 리턴하지 않는다. 즉 메시지가 없을 경우 무한 대기한다.

만약, 메시지 큐에 메시지가 없을 경우 다른 작업을 하고 싶다면(유휴 시간) 가능할까? 그것을 가능케 하는 함수가 PeekMessage API 이다.

BOOL PeekMessage(LPMSG lpMsg, HWND hwnd, UINT wMsgFilterMin, UINT wMsgFilterMax, UINT wRemoveMsg);

이 함수는 메시지 큐에서 메시지를 꺼내거나 검사하되 메시지가 없더라도 즉각 리턴한다. 리턴값이 TRUE 이면 메시지가 있다는 뜻이고, FALSE 이면 메시지가 없다는 뜻이다. wRemoveMsg 는 메시지가 있을 경우 이 메시지를 큐에서 제거할 것인지 아닌지를 지정하는 데 PM_REMOVE 이면 큐에서 메시지를 제거하고, PM_NOREMOVE 이면 제거하지 않을 수도 있다.

```
// 5. Message
MSG msg;
while(true)
{
    if( PeekMessage( &msg, 0, 0, 0, PM_REMOVE ) )
    {
        if( msg.message == WM_QUIT)                break;

        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else
    {
        HDC hdc = GetDC(hwnd);

        SetPixel(hdc, rand()%500, rand()%400, RGB(rand()%256, rand()%256, rand()%256));

        ReleaseDC(hwnd, hdc);
    }
}
```



```

    }
}

```

예제 4.2 PeekMessage

4. 5 사용자 정의 메시지

4.5.1 메시지 범위

메시지의 명칭은 흔히 WM_PAINT, WM_CREATE 등과 같은 매크로 상수로 나타나는데 이 매크로의 실제 값은 정수로 정의되어 있다. 메시지간의 구분을 위한 표식이므로 중복되지 만 않으면 되고 이런 목적에는 정수형이 제일 적합하다.

메시지 ID의 실제값은 Winuser.h 파일에 다음과 같이 정의되어 있다.

메시지	값
WM_CREATE	1
WM_DESTROY	2
WM_MOVE	3
...	
WM_LBUTTONDOWN	0x201
...	

프로그래밍을할 때는 보통 WM_PAINT 등의 매크로 상수만을 사용하므로 메시지 ID의 실제 상수값은 굳이 몰라도 상관없다. 하지만 사용자 정의 메시지를 만드려면 나만의 메시지를 만들어야 하고, 그 때 기존 메시지의 ID값과 충돌이 나면 안된다.

메시지 ID의 데이터형은 UINT형, 즉 부호없는 정수형이므로 범위는 2의 32승, 즉 구분이 가능한 메시지의 종류가 무려 40억개에 이른다. 하지만 실제 사용되는 메시지의 수는 불과 천 개도 안되며 나머지 ID는 대부분 특수한 용도로 예약되어 있다.

아래는 메시지 ID의 범위와 용도를 정리한 것이다.

범위	용도
0 ~ WM_USER-1	시스템 메시지로 예약되어 있으며 WM_로 시작되는 대부분의 메시지는 이 영역에 예약되어 있다.
WM_USER ~	응용 프로그램이 자신의 필요에 따라 사용할 수 있는 메시지 영역

WM_APP -1	이다. 내부적인 용도로만 사용해야 하며 응용 프로그램간의 통신을 위해서 사용하면 안된다.
WM_APP~ 0xBFFF	응용 프로그램이 개별적인 용도로 사용할 수 있는 영역이며 시스템은 이 영역의 메시지를 전혀 사용하지 않는다.
0xc000 ~ 0xFFFF	실행중에 메시지를 등록하는 별도의 함수에 의해 정의된다. 예) RegisterWindowMessage()

결국 시스템이 정의한 메시지들은 모두 0x400 이하의 ID를 가지므로 WM_USER 이후의 메시지 ID는 사용자가 정의해서 사용할 수 있는 영역이다.

4.5.2 사용자 정의 메시지

응용 프로그램은 시스템이 정의한 메시지 외에 자신의 필요에 따라 고유의 메시지를 만들어 사용할 수 있다. 메시지는 약속이기 때문에 정하기 나름이며 wParam, lParam 의 용도도 편한대로 정해 사용할 수 있다.

사용자 정의 메시지를 사용하기 위해서는 다음과 같은 절차를 거친다.

- 1) 나만의 메시지를 정의한다.
- 2) 메시지 프로시저에서 해당 메시지를 필터링한다.
- 3) 원하는 시점에서 SendMessage 를 사용하여 사용자 정의 메시지를 호출한다.

```
#include <windows.h>
#include <tchar.h>

#define WM_MYMESSAGE WM_USER+100

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_MYMESSAGE:
        {
            TCHAR buf[20];

            wprintf(buf, TEXT( "%d+%d=%d" ), wParam, lParam, wParam+lParam);

            MessageBox(hwnd, buf, TEXT( "" ), MB_OK);
        }
    }
}
```

```

        }

        return 0;

    case WM_LBUTTONDOWN:

        SendMessage(hwnd, WM_MYMESSAGE, 10, 20);

        return 0;

    case WM_CREATE:

        return 0;

    case WM_DESTROY:

        PostQuitMessage(0);

        return 0;

    }

    return DefWindowProc(hwnd, msg, wParam, lParam);
}

```

예제 4.3 사용자 정의 메시지

사용자 정의 메시지는 윈도우끼리 메시지 통신 또한 가능하다. 다른 프로그램과 통신하는 예를 확인해 보자.

일단 서로 다른 프로그램끼리 통신을 하려면 약속된 메시지가 필요하다. 따라서 사용자 정의 메시지를 두 개의 프로그램에 동일하게 설정한 후 해당 메시지 형태로 전송하면 된다.

```

#include <Windows.h>
#include <stdio.h>
#define WM_MYMESSAGE WM_USER+100

void main()
{
    HWND hwnd = FindWindow(0, TEXT("Hello"));

    int value = SendMessage(hwnd, WM_MYMESSAGE, 10, 20);

    printf("%d\n", value);
}

```

예제 4.4 서로 다른 프로세스 끼리의 전송1

```

#define WM_MYMESSAGE WM_USER+100

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_MYMESSAGE:
            {
                TCHAR buf[20];

                wsprintf(buf, TEXT( "%d+%d=%d"), wParam, lParam, wParam+lParam);

                MessageBox(hwnd, buf, TEXT( "" ), MB_OK);

            }

            return wParam + lParam;

        case WM_DESTROY:
            PostQuitMessage(0);

            return 0;

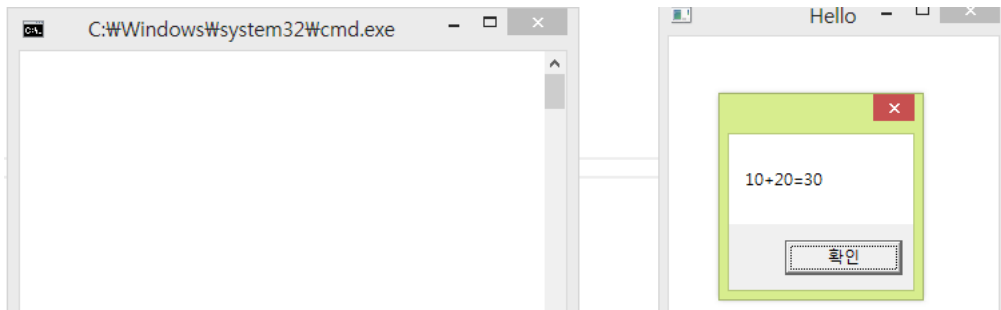
    }

    return DefWindowProc(hwnd, msg, wParam, lParam);
}

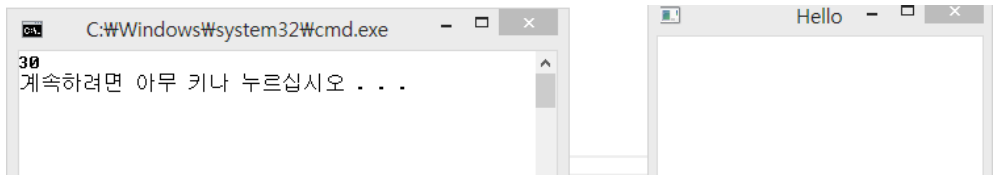
```

예제 4.4 서로 다른 프로세스 끼리의 전송2

메시지 박스 종료 전



메시지 박스 종료 후



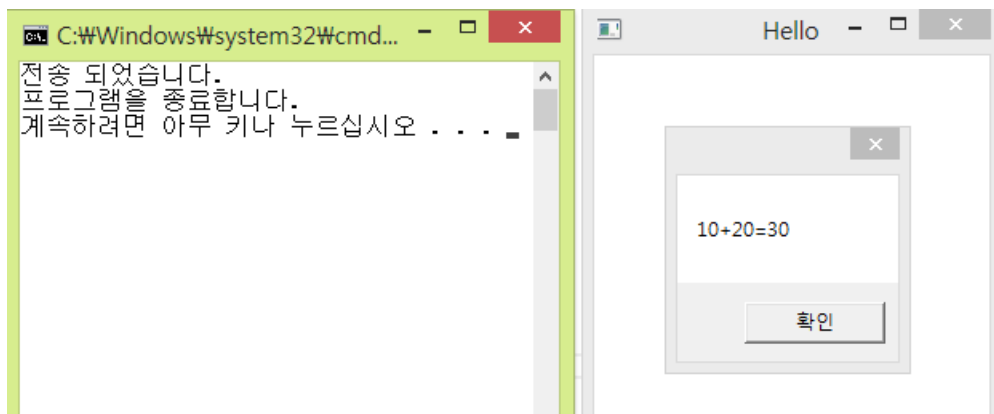
SendMessage는 호출 후 상대방이 리턴하기 전까지 리턴하지 못한다. 따라서 호출 한 프로그램은 상대방이 메시지박스를 닫기 전까지는 프로그램이 진행되지 못한다.

메시지 박스를 종료하면 리턴값이 호출한 프로그램 쪽으로 전달되게 된다.

다음은 동일한 API 프로그램을 PostMessage로 호출한 예이다. PostMessage는 SendMessage와는 달리 상대방 큐에 메시지를 넣는다. 따라서 큐에 넣고 바로 리턴하기 때문에 리턴 값은 성공과 실패 여부의 BOOL 형 값을 갖는다.

```
#include <Windows.h>
#include <stdio.h>
#define WM_MYMESSAGE WM_USER+100
void main()
{
    HWND hwnd = FindWindow(0, TEXT("Hello"));
    BOOL b = PostMessage(hwnd, WM_MYMESSAGE, 10, 20);
    if( b == true)
    {
        printf("전송 되었습니다.\n");
    }
    printf("프로그램을 종료합니다.\n");
}
```

예제 4.4 서로 다른 프로세스 끼리의 전송3



PostMessage API 호출 결과 상대방이 해당 메시지 처리를 다 끝내지 않았지만 호출자는 해당 함수가 리턴해서 종료됨을 볼 수 있다.

결국 SendMessage 와 PostMessage 는 둘 다 간단한 데이터를 전송할 수 있는 함수인데 SendMessage는 비큐메시지로 동작하고 PostMessage 는 큐 메시지로 동작한다.

5. DC & GDI

5.1 DC

5.1.1 Device Context란?

DC란 출력에 필요한 모든 정보를 가지는 구조체이며 GDI 모듈에 의해 관리된다. 문자열의 모양을 지정하는 폰트, 선의 색상과 굵기, 채움 무늬와 색상, 그리기 모드 등등이 모두 출력에 필요한 정보들이다.

윈도우를 만들고 그림을 그리려면 반드시 DC의 핸들을 얻어야 한다. DC는 임의의 장치(윈도우, 프린터)에 출력을 할 때 사용하는 다양한 속성의 지닌 일종의 구조체 이다. DC는 하나의 장치에 연결되어 있다. 사용자는 DC에 출력을 하면 DC가 지닌 여러 속성을 사용해서 DC가 연결된 장치에 출력된다.

DC는 하나의 메시지를 처리하는 동안 핸들을 열고 제거해야 한다. 즉, static 형태로 보관해서 사용할 수 없다.

DC를 얻는 방법은 다양하며, 제거 방법도 각기 다르다.

DC얻기	DC 해제	Description
GetDC()	ReleaseDC()	Client 영역을 위한 DC를 얻는다.
GetDCEx()	ReleaseDC()	GetDC()의 확장 함수
GetWindowDC()	ReleaseDC()	윈도우 전체를 위한 DC를 얻는다.
BeginPaint()	EndPaint()	무효화 영역을 위한 DC를 얻는다.
CreateCompatibleDC()	DeleteDC()	메모리 DC, 주로 비트맵을 사용하기 위해 얻는다
CreateDC()	DeleteDC()	바탕화면, 프린터를 위한 DC를 얻는다.
CreateMetaFile()	CloseMetaFile()	메타파일을 위한 DC를 얻는다.
CreateIC()	DeleteIC()	정보 dc를 얻는다.

클라이언트 DC 얻기

```
HDC GetDC(HWND hWnd);
```

ReleaseDC(HWND hwnd, HDC hdc);

[사용 예]

HDC hdc;

hdc = GetDC(hwnd);

//각종 출력문에서 hdc를 사용한다.

ReleaseDC(hwnd, hdc);

바탕화면 DC 얻기

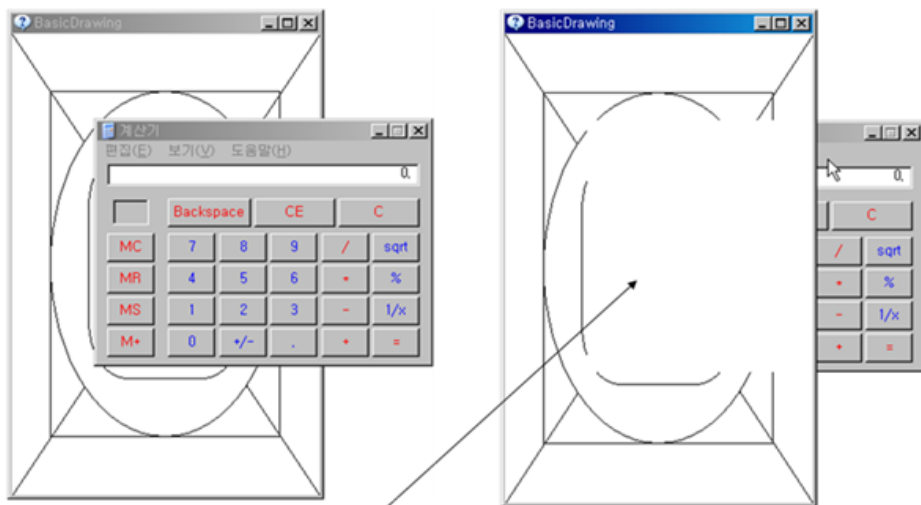
아래의 방법들을 사용해서 바탕화면을 위한 DC를 얻을 수 있다.

HDC hdc = GetDC(0);

HDC hdc = CreateDC("DISPLAY", 0, 0, 0);

5.1.2 무효화 영역

윈도우가 다른 윈도우에 뒤에 있다가 앞으로 나올 때 다시 그려야 하는 영역을 무효화 영역이라고 한다.



무효영역(Invalid Region) : 다시 그려져야 하는 영역

아래와 같은 경우에 무효화 영역이 발생한다.

- 윈도우를 옮기거나 제거 했을 때 이전에 감추어졌던 윈도우 영역이 보이게 될 때
- 윈도우의 크기를 조정했을 때(클래스 Style에 CS_HREDRAW 와 CS_VREDRAW Style 이 있는 경우)

- ScrollWindow(), ScrollDC() 함수를 사용하는 경우.
- InvalidateRect(), InvalidateRgn() 함수를 사용하는 경우
- 윈도우의 일부에 겹친 DialogBox나 메시지 박스가 제거 될 때
- 메뉴가 나타났다 사라질 때
- 풍선도움말이 나타났을 때

다음의 경우는 운영체제가 화면을 저장한 후 복구해 준다.

- 마우스 커서가 클라이언트 영역을 지나갈 때
- 아이콘이 클라이언트 영역을 지나갈 때

5.1.3 WM_PAINT

무효화 영역이 발생하면 운영체제는 WM_PAINT 메시지를 해당 윈도우 프로시저에 보낸다.

하나의 윈도우는 내부적으로 Paint Information Struct 를 가지고 있다. 이 구조체 안에 무효화 영역을 두러싸는 가장 작은 사각형의 좌표를 지닌다.

WM_PAINT는 2번 메시지 큐에 들어 오지 않고 단지 이 좌표만이 Update 된다.

GetUpdateRect() 를 사용하면 이 좌표를 얻을 수 있다.

5.1.4 BeginPaint/EndPaint

클라이언트 DC 얻기

WM_PAINT 메시지 루틴에서 클라이언트 DC를 얻고 사용할 수 있는 방법이다. WM_PAINT 메시지 처리 루틴에서는 DC 핸들을 GetDC로 얻지 않고 BeginPaint 함수로 얻으며 해제할 때는 EndPaint 함수를 사용한다.

정리하면, GetDC는 DC 핸들을 얻는 일반적인 방법이며, BeginPaint는 WM_PAINT 메시지 내에서 사용하는 특수한 방법이다.

```
HDC BeginPaint(HWND hwnd, LPPAINTSTRUCT lpPaint);  
BOOL EndPaint(HWND hwnd, CONST PAINTSTRUCT *lpPaint);
```

[사용 예]

```
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);
    //각종 출력문에서 hdc를 사용한다.
    EndPaint(hWnd, &ps);
}
return 0;
```

BeginPaint 함수는 윈도우 핸들 외에도 페인트 정보 구조체를 인수로 요구하여 이 구조체에 그림 그리기에 필요한 여러 가지 복잡한 정보를 리턴한다.

BeginPaint는 아래와 같은 일을 한다.

- DC를 만들고 UpdateRegion을 DC의 Clipping 영역으로 지정한다.
- 무효화 영역을 지우기 위해 WM_ERASEBKGND 메시지를 보낸다.
- PAINTSTRUCT 구조체를 채운다.
- Caret 이 있을 경우 그리기 전에 Hide하고 EndPaint() 에서 Show 한다.
- 무효화 영역을 유효화 한다.

참고로 아래의 코드는 절대로 사용하면 안된다.

```
case WM_PAINT:
    return 0;
```

무효화 영역이 유효화 되지 않으므로 WM_PAINT 가 무한반복 호출된다.

PAINTSTRUCT

PAINTSTRUCT 구조체는 발생된 무효화 영역에 대한 여러가지 정보를 얻어 오기 위해 사용한다.

```
typedef struct tagPAINTSTRUCT
{
    HDC hdc;           // dc handle
    BOOL fErase;       // 배경을 지운경우 FALSE, 지우지 않은 경우 TRUE
    RECT rcPaint;      // 클리핑 영역을 포함하는 최소 사각형
    BOOL fRestore;     //Reserved; used internally by the system
    BOOL fInCUpdate;   //Reserved; used internally by the system
```

```
        BYTE rgbReserved[32];        //Reserved; used internally by the system
    } PAINTSTRUCT, *PPAINTSTRUCT;
```

대부분의 경우 fErase는 FALSE 이다. 즉 운영체제가 배경을 지운다. 하지만,

InvalidateRect(hwnd, 0, FALSE);

이 경우 무효화 영역을 만들고 배경을 지우지 않게 한다. 그러므로 fErase 값은 TRUE이다.

rcPaint 는 현재 DC가 출력할 수 있는 클리핑 영역을 가르키는 데 이를 잘 이용할 경우 프로그램의 성능을 훨씬 뛰어나게 만들 수 있다.

화면에 원하는 그림을 출력하기 위해서는 반드시 WM_PAINT 메시지에서 출력해야 한다. WM_PAINT 메시지 에서는 클라이언트 영역에 그리기 위한 모든 정보를 알고 있어야 한다.

```
#include <windows.h>
#include <tchar.h>

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_LBUTTONDOWN:
        {
            HDC hdc = GetDC(hwnd);
            Rectangle(hdc, 10,10,100,100);
            ReleaseDC(hwnd, hdc);
        }

        return 0;

        case WM_DESTROY:
            PostQuitMessage(0);

            return 0;
```

```

    }

    return DefWindowProc(hwnd, msg, wParam, lParam);
}

```

예제 5.1 DC 사용하기

위 예제는 마우스 왼쪽 버튼 클릭시 클라이언트 DC를 얻어 사각형을 출력하는 예제이다. 다음 예제는 무효화 영역과 관련된 내용이다.

```

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_LBUTTONDOWN:
        {
            HDC hdc = GetDC(hwnd);

            Rectangle(hdc, 120, 10, 210, 100);

            ReleaseDC(hwnd, hdc);

        }

        return 0;

        case WM_RBUTTONDOWN:
        {
            RECT r = {120, 10, 210,100};

            InvalidateRect(hwnd, &r, TRUE);

        }

        return 0;

        case WM_PAINT:
        {
            PAINTSTRUCT ps;

            HDC hdc = BeginPaint(hwnd, &ps);

            Rectangle(hdc, 10,10,100,100);

            EndPaint(hwnd, &ps);

        }
    }
}

```

```

        return 0;

    case WM_DESTROY:
        PostQuitMessage(0);

        return 0;
    }

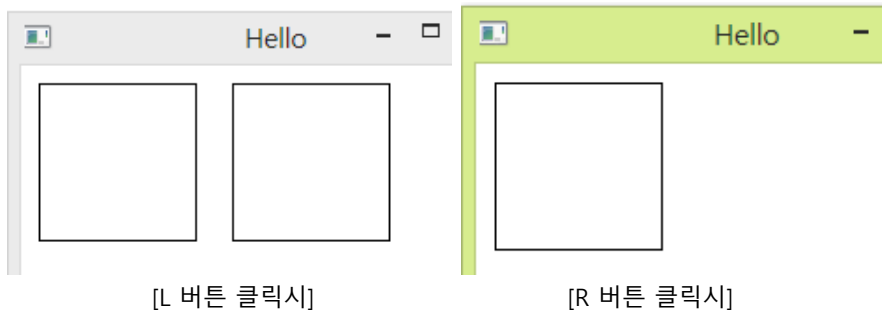
    return DefWindowProc(hwnd, msg, wParam, lParam);
}

```

예제 5.2 무효화 영역

프로그램을 실행하면 화면을 그리기 위해 WM_PAINT 메시지가 호출되며 좌측 상단에 사각형이 그려지게 된다. 만약, 마우스 왼쪽 버튼을 클릭하면 그려진 사각형 우측에 아래 그림처럼 사각형이 그려진다.

만약 마우스 오른쪽 버튼을 클릭하면 강제로 무효화 영역을 발생시키게 되며, 발생한 무효화 영역에 출력되었던 사각형은 사라지게 된다.



다음 예제를 분석해 보길 바란다.

```

#include <math.h>

#define NUM    1000
#define TWOPI  (2 * 3.14159)

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static int    cxClient, cyClient ;
    HDC          hdc ;

```

```

int          i ;

PAINTSTRUCT  ps ;

POINT        apt [NUM] ;

switch (message)      {

case WM_SIZE:

    cxClient = LOWORD (lParam) ;
    cyClient = HIWORD (lParam) ;

    return 0 ;


case WM_PAINT:

    hdc = BeginPaint (hwnd, &ps) ;

    MoveToEx (hdc, 0,          cyClient / 2, NULL) ;

    LineTo   (hdc, cxClient, cyClient / 2) ;

    for (i = 0 ; i < NUM ; i++)      {

        apt[i].x = i * cxClient / NUM ;

        apt[i].y = (int) (cyClient / 2 * (1 - sin (TWOPI * i / NUM))) ;

    }

    Polyline (hdc, apt, NUM) ;

    return 0 ;

case WM_DESTROY:

    PostQuitMessage (0) ;

    return 0 ;

}

return DefWindowProc (hwnd, message, wParam, lParam) ;

}

```

예제 5.3 사인곡선 그리기

5. 2 문자열

5.2.1 TextOut

문자열 출력을 하기 위해서는 아래의 함수를 사용한다.

BOOL TextOut(HDC hdc, int nXStart, int nYStart, LPCTSTR lpString, int cbString);

첫 번째 인수는 DC의 핸들이다.

두 번째, 세 번째 인수는 문자열이 출력될 좌표이며 윈도우의 작업영역 원점을 기준으로 한다.

세 번째 인수는 출력할 문자열을 담고 있는 문자열 포인터이며, 마지막 인수는 출력할 문자열의 길이이다.

문자열의 정렬 방법을 변경하는 함수는 SetTextAlign 이라는 함수이다.

UINT SetTextAlign(HDC hdc, UINT fMode);

첫 번째 인수는 DC의 핸들이다.

두 번째 인수로 지정하는 정렬 정보에 따라 hdc의 정렬 상태를 변경한다. 이후부터 출력되는 모든 문자열은 이 함수가 지정한 정렬 상태를 따른다.

세 번째 인수는 아래와 같으며 두 개 이상의 플래그를 OR 로 연결하여 지정한다.

값	설명
TA_TOP	지정한 좌표가 상단 좌표가 된다.
TA_BOTTOM	지정한 좌표가 하단 좌표가 된다.
TA_CENTER	지정한 좌표가 수평 중앙 좌표가 된다.
TA_LEFT	지정한 좌표가 수평 왼쪽 좌표가 된다.
TA_RIGHT	지정한 좌표가 수평 오른쪽 좌표가 된다.
TA_UPDATECP	지정한 좌표대신 CP를 사용하며 문자열 출력 후에 CP를 변경한다.
TA_NOUPDATECP	CP를 사용하지 않고 지정한 좌표를 사용하며 CP를 변경하지 않는다.

예제를 살펴보자.

```

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_LBUTTONDOWN:
        {
            HDC hdc = GetDC(hwnd);
            SetTextAlign(hdc, TA_UPDATECP);
        }
    }
}

```

```

        TextOut(hdc, 200, 200, TEXT("Buautiful Korea"), 15);
        TextOut(hdc, 200,220, TEXT("is My"), 5);
        TextOut(hdc, 200, 240, TEXT("Lovely Home Country"), 19);
        ReleaseDC(hwnd, hdc);
    }
    return 0;
case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hwnd, &ps);
        SetTextAlign(hdc, TA_CENTER);
        TextOut(hdc, 200, 60, TEXT("Buautiful Korea"), 15);
        TextOut(hdc, 200,80, TEXT("is My"), 5);
        TextOut(hdc, 200, 100, TEXT("Lovely Home Country"), 19);
        EndPaint(hwnd, &ps);
    }
return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
}
return DefWindowProc(hwnd, msg, wParam, lParam);
}

```

예제 5.4 TextOut



중앙에 WM_PAINT 메시지에서 출력된 결과물은 중앙 정렬하여 출력되었다.

마우스 왼쪽 버튼 클릭시 CP(Current Position)을 사용하여 정렬토록 설정하였다. 따라서
인자로 전달된 값들은 무효가 되고 출력될 커서의 위치에서 출력되게 된다.

이 기능을 사용하면 콘솔에 printf 로 출력하듯이 문자열을 연속적으로 출력할 수 있다.

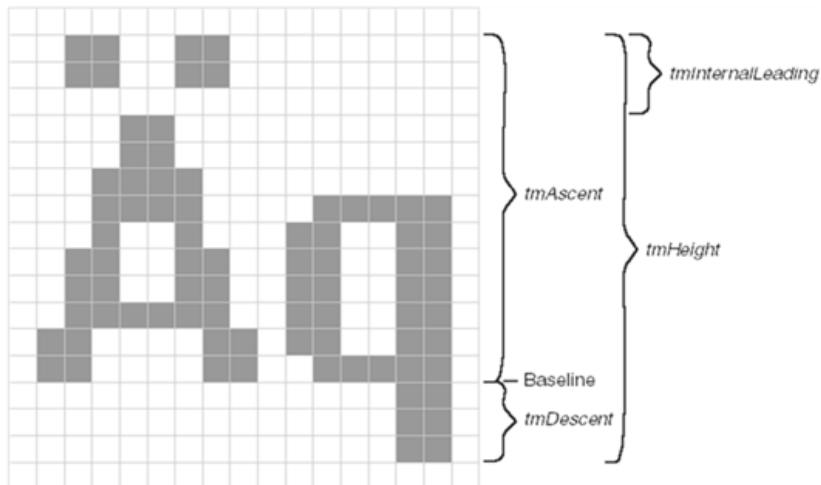
TextOut 함수를 사용해서 여러줄의 텍스트를 표시하려면 글꼴 크기를 결정할 필요가 있다. 현재 DC가 가지고 사용하고 있는 글꼴이 크기 정보를 알아 내려면 TEXTMETRIC 구조체와 GetTextMetrics() 함수를 사용하면 된다. 이때 TEXTMETRIC 구조체는 20개의 필드(MSDN 참조)를 가지지만 중요한 값은 아래와 같다.

tmHeight : 글자의 최대 세로 크기

tmAveCharWidth : 소문자의 평균 넓이

tmMaxCharWidth : 가장 넓은 글자의 폭

System 이 사용하는 글꼴의 크기는 해상도에 따라 달라 진다.



다음은 시스템 폰트에서 글자의 폭과 높이를 얻는 코드이다.

```
HRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    static int cxChar, cyChar;
    switch( msg )
    {
        case WM_CREATE:
        {
```

```

        TEXTMETRIC tm;

        HDC hdc;

        hdc = GetDC (hwnd) ;

        GetTextMetrics (hdc, &tm) ;

        cxChar = tm.tmAveCharWidth ;

        cyChar = tm.tmHeight + tm.tmExternalLeading ;

        ReleaseDC (hwnd, hdc) ;

    }

    return 0 ;

case WM_PAINT:
{

    PAINTSTRUCT ps;

    HDC hdc = BeginPaint(hwnd, &ps);

    TCHAR buf[20];

    wsprintf(buf, TEXT("%d, %d"), cxChar, cyChar);

    TextOut(hdc, 10, 10, buf, wcslen(buf));

    EndPaint(hwnd, &ps);

}

return 0;

case WM_DESTROY:

    PostQuitMessage(0);

    return 0;

}

return DefWindowProc(hwnd, msg, wParam, lParam);

}

```

예제 5.5 GetTextMetrics

5.2.2 DrawText

TextOut 은 한 줄만 출력하므로 기능이 너무 단순하다. 이보다 조금 더 복잡한 문자열 출력함수로 DrawText라는 함수가 있다.

***int DrawText(HDC hdc, LPCTSTR lpString, int nCount,
LPRECT lpRect, UINT uFormat);***

이 함수는 사각영역을 정의하여 영역 안에 문자열을 출력할 수 있으며 여러 가지 포맷을 설정하는 기능이 있다.

앞의 3개 인수는 TextOut 과 동일하다.

단 3번째 인수는 출력할 문자열의 길이인데 만약 -1을 지정하면 널 종료 문자열로 간주한다. TextOut과는 달리 널 종료 문자열을 인식하므로 문자열 상수를 곧바로 출력할 수 있다.

네 번째 인수는 출력할 영역을 담고 있는 구조체 포인터이다.

다섯번째 인수는 DrawText 함수가 문자열을 출력할 방법을 지정하는 플래그이며 다음과 같은 여러 가지 플래그의 조합을 지정한다.

값	설명
DT_LEFT	수평 왼쪽 정렬한다.
DT_RIGHT	수평 오른쪽 정렬한다.
DT_CENTER	수평 중앙 정렬한다.
DT_BOTTOM	사각영역의 바닥에 문자열을 출력한다.
DT_VCENTER	사각영역의 수직 중앙에 문자열을 출력한다.
DT_WORDBREAK	사각영역의 오른쪽 끝에서 자동 개행되도록 한다.
DT_SINGLELINE	한 줄로 출력한다.
DT_NOCLIP	사각영역의 경계를 벗어나도 문자열을 자르지 않고 그대로 출력한다.

수평, 수직 정렬 플래그와 자동 개행 플래그 등이 정의되어 있다. 아래 예제를 작성해 보자.

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_PAINT:
        {
            TCHAR str[] = TEXT("네 번째 인수는 출력할 영역을 담고 있는 구조체 포인터이다. 다섯번째 인수는 DrawText 함수가 문자열을 출력할 방법을지정하는 플래그이며 다음과 같은 여러 가지 플래그의 조합을 지정한다.");

            RECT rt = {100,100,400,300};
```

```

        PAINTSTRUCT ps;

        HDC hdc = BeginPaint(hwnd, &ps);

        DrawText(hdc, str, -1, &rt, DT_CENTER|DT_WORDBREAK);

        EndPaint(hwnd, &ps);

    }

    return 0;

    case WM_DESTROY:

        PostQuitMessage(0);

        return 0;

    }

    return DefWindowProc(hwnd, msg, wParam, lParam);
}

```

예제 5.6 DrawText

사각영역 rt를 정의하였고 출력할 문자열 str에 긴 문자열을 대입하였다. 그리고 길이의 값을 -1로 주어 문자열의 끝까지 출력했으며 중앙 정렬 및 자동 개행 플래그를 주었다. 실행 결과는 아래와 같다.

네 번째 인수는 출력할 영역을 담고 있는 구조체 포인터이다. 다섯 번째 인수는 **DrawText** 함수가 문자열을 출력할 방법을 지정하는 플래그이며 다음과 같은 여러 가지 플래그의 조합을 지정한다.

5. 3 그래픽 출력

5.2에서 문자열을 출력해 보았다. 그 외 다양한 API 함수로 그래픽 출력을 해보자. 그래픽 출력을 하는 방법은 문자열 출력 방법과 크게 다르지 않다. 아래는 그래픽 출력함수들이다.

COLORREF SetPixel(hdc, nXPos, nYPos, clrref);
DWORD MoveToEx(hdc, x, y, lpPoint);
BOOL LineTo(hdc, xEnd, yEnd)
BOOL Rectangle(hdc, nLeftRect, nTopRect, nRightRect, nBottomRect);

BOOL Ellipse(hdc, nLeftRect, nTopRect, nRightRect, nBottomRect);

아래 예제를 통해 간단히 사용방법을 익혀보자.

```

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);

            SetPixel(hdc, 10, 10, RGB(255, 0, 0));

            MoveToEx(hdc, 50,50, NULL);

            LineTo(hdc, 300,90);

            Rectangle(hdc, 50, 100, 200,180);

            Ellipse(hdc, 220, 100, 400, 200);

            EndPaint(hwnd, &ps);

        }
        return 0;

        case WM_DESTROY:
            PostQuitMessage(0);

            return 0;

    }

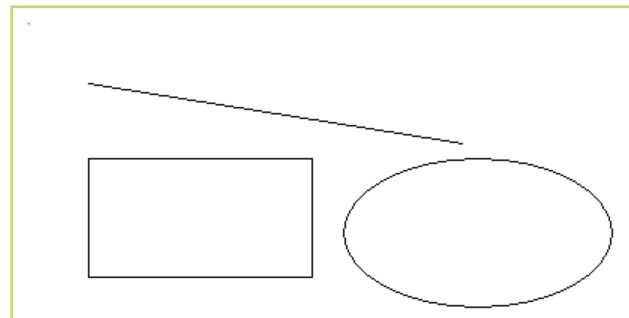
    return DefWindowProc(hwnd, msg, wParam, lParam);
}

```

예제 5.7 그래픽 출력

BeginPaint 함수를 호출하여 DC 핸들을 얻은 후 그래픽 출력 함수를 연속적으로 호출하였다. SetPixel 함수로 빨간색 점을 찍었는데 이 호출문에서 RGB(255,0,0)이 빨간색을 의미한다. MoveToEx 와 LineTo 함수로 선을 그었으며 Rectangle 함수로 사각형을 Ellipse 함수로 타원을 그려 보았다.

출력결과는 다음과 같다.



5. 4 GDI Object

5.4.1 GDI Object

GDI(Graphic Device Interface)는 화면, 프린터 등의 모든 출력 장치를 제어하는 윈도우원 핵심 모듈 중 하나이다. 윈도우 프로그램에서의 모든 출력은 GDI를 통해서 화면과 프린터로 나가게 되어 있다.

GDI 오브젝트란 그래픽 출력에 사용되는 도구들을 말하며 펜, 브러쉬, 비트맵, 폰트 등 이 모두 GDI 오브젝트이다.

GDI 오브젝트는 일종의 구조체이지만 핸들로 관리되므로 우리는 GDI 오브젝트를 생성하는 함수를 부르고 이 함수가 리턴하는 핸들을 받아서 사용하기만 하면 된다.

DC가 처음 만들어 질 때 GDI 오브젝트별 선택되는 디폴트 값은 아래와 같다.

GDI 오브젝트	핸들 타입	설명	디폴트
펜	HPEN	선을 그릴 때 사용	검정색의 가는 실선
브러쉬	HBRUSH	면을 채울 때 사용	흰색
폰트	HFONT	문자 출력에 사용	시스템 글꼴
비트맵	HBITMAP	비트맵 이미지	선택되지 않음
팔레트	HPALETTE	팔레트	선택되지 않음
리전	HRGN	화면상의 영역	선택되지 않음

5.4.2 스톡 오브젝트

스톡 오브젝트(Stock Object)는 윈도우가 미리 제공하는 GDI 오브젝트를 말한다. 만들지

않아도 언제나 사용할 수 있고 사용 후 파괴할 필요가 없다. 다음 함수로 핸들을 얻어서 사용한다.

HGDIOBJ GetStockObject(int fnObject);

사용 가능한 스톡 오브젝트는 다음과 같다. 주로 브러쉬와 펜이 스톡오브젝트로 제공된다

오브젝트	설명	오브젝트	설명
BLACK_BRUSH	검정색 브러쉬	GRAY_BRUSH	회색 브러쉬
NULL_BRUSH	투명 브러쉬	WHITE_BRUSH	흰색 브러쉬
DKGRAY_BRUSH	짙은 회색 브러쉬	LTGRAY_BRUSH	옅은 회색 브러쉬
DC_BRUSH	색상 브러쉬		
BLACK_PEN	검정색 펜	WHITE_PEN	흰색 펜
NULL_PEN	투명 펜	DC_PEN	색상 펜
ANSI_FIXED_FONT	고정폭 폰트	ANSI_VAR_FONT	가변폭 폰트
DEFAULT_PALETTE	시스템 팔레트		

```

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);
            HBRUSH brush = (HBRUSH)GetStockObject(GRAY_BRUSH);
            HBRUSH old = (HBRUSH)SelectObject(hdc, brush);
            Rectangle(hdc, 10,10,100,100);
            SelectObject(hdc, old);
            EndPaint(hwnd, &ps);
        }
        return 0;
        case WM_DESTROY:
    
```

```

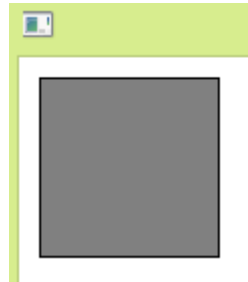
        PostQuitMessage(0);

        return 0;
    }

    return DefWindowProc(hwnd, msg, wParam, lParam);
}

```

예제 5.8 스톡 오브젝트 사용하기



회색의 사각형이 출력되었다.

위 코드를 통해 간단히 GDI 오브젝트 사용방법을 살펴보자.

디폴트 오브젝트를 사용하지 않고 내가 원하는 오브젝트를 사용하고 싶다면 아래의 절차를 거쳐야 한다.

- 1) 오브젝트를 생성한다.
 - 오브젝트를 생성하는 방법은 2가지가 있다.
 - 시스템이 생성한 오브젝트를 가져온다.

GetStockObject
 - 내가 원하는 형태의 오브젝트를 생성한다.

CreateXXX
- 2) 생성된 오브젝트를 선택한다.
 - GetStockObject 함수가 그 역할을 한다.

선택하고자 하는 오브젝트를 인자로 전달하면 원래 가지고 있었던 오브젝트를 리턴해준다.
- 3) 그래픽 함수로 원하는 도형을 출력한다.
- 4) 다시 원래의 오브젝트를 선택한다.
 - 2)번 과정에서 리턴된 오브젝트로 다시 복원한다.
- 5) 생성한 오브젝트를 소멸한다.
 - 만약, 1)번에서 생성한 오브젝트가 시스템 오브젝트라면 소멸할 필요가 없다.

즉, 내가 원하는 형태의 오브젝트 일경우만 아래의 함수로 소멸하면 된다.

DeleteObject()

5.4.3 색상

윈도우에서는 색상값을 표현하기 위해 COLORREF 라는 데이터형을 사용한다.
아래와 같이 정의되어 있다.

typedef DWORD COLORREF;

해당 데이터 타입에 색상을 저장하기 위해서는 RGB 매크로 함수를 사용하는데 이는 세 가지 인수를 가진다. (빨간색, 초록색, 파란색 순) 이 3가지 인수로 색상을 조합한다.
아래 표의 예로 이해를 돕도록 하자.

RGB값	색상	RGB값	색상
RGB(0,0,0)	검정색	RGB(255,255,255)	흰색
RGB(128,128,128)	회색	RGB(255,0,0);	빨간색
RGB(0,255,0)	초록색	RGB(0,0,255)	파란색
RGB(255,255,0);	노란색	RGB(255,0,255)	분홍색

위의 예 처럼 모든 인수는 0 ~ 255의 값을 갖는다.

아래의 매크로 함수는 COLORREF 형 변수에서 각 색상의 농도를 분리한다.

```
#define GetRValue(rgb) ((BYTE)(rgb))  
#define GetGValue(rgb) ((BYTE)(((WORD)(rgb))>>8))  
#define GetBValue(rgb) ((BYTE)((rgb)>>16))
```








5.4.4 펜

펜은 선을 그을 때 사용되는 GDI 오브젝트이다. 펜을 만들 때는 다음 함수를 사용한다.






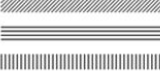
HPEN CreatePen(int fnPenStyle, int nWidth, COLORREF crColor);

첫 번째 인수는 선의 모양을 지정한다. 아래 그림의 Pen Style을 참조하자.
두 번째 인수는 선의 폭을 지정한다. 디폴트 선의 굵기는 1이다.
세 번째 인수는 선의 색상을 지정한다.

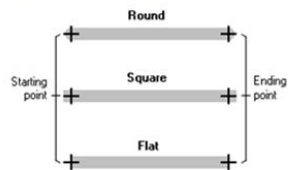
□ Pen Style

Solid	
Dash	
Dot	
Dash-dot	
Dash-dot-dot	
Null	
Inside-frame	





□ Pen Hatch

Forward diagonal	
Cross	
Diagonal cross	
Backward diagonal	
Horizontal	
Vertical	

□ Pen End Cap



□ Pen Pattern

Hatch	
Hollow	
Custom	
Solid	

□ Pen Join



□ Pen Width

□ Pen Color

다음 간단한 예제를 통해 펜의 사용방법을 정리해보도록 하자.

```

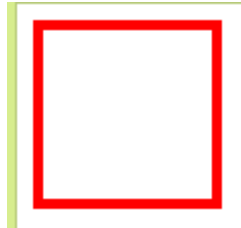
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);
            HPEN pen = CreatePen(PS_SOLID, 5, RGB(255,0, 0));
            HPEN old = (HPEN)SelectObject(hdc, pen);
            Rectangle(hdc, 10,10,100,100);
            DeleteObject(SelectObject(hdc, old));
            EndPaint(hwnd, &ps);
        }
        return 0;
    }

    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }
}

```

```
return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

예제 5.9 펜



개별적으로 CreatePen 의 인자를 변경시켜 가면서 속성 정보를 확인해 보도록 하자.

GDI 오브젝트를 만드러 사용하는 일반적인 절차를 다시 보도록 하자.

- 1) 핸들을 선언한다.
- 2) GDI 오브젝트를 생성한다.
- 3) DC에 선택하된 이 때 이전 핸들을 반드시 저장해 두어야 한다.
- 4) 사용한다.
- 5) 선택을 해제한다.(원래의 오브젝트를 재 선택한다.)
- 6) 삭제한다.

5.4.5 브러쉬

브러쉬는 채워지는 면을 채색하는 용도로 사용되는데 말 그대로 붓을 의미한다.

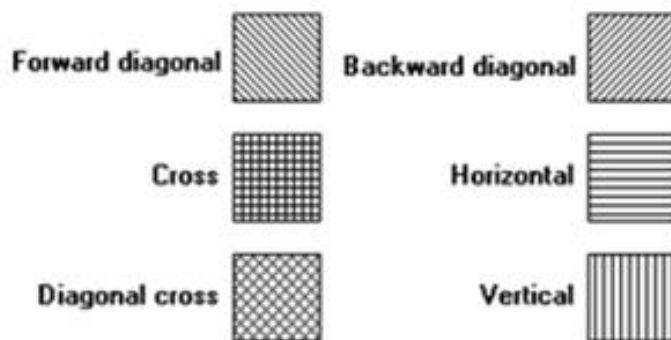
다음 함수로 브러쉬를 생성할 수 있다.

HBRUSH CreateSolidBrush(COLORREF crColor);

HBRUSH CreateHatchBrush(int fnStyle, COLORREF clrref);

첫 번째 함수는 단색 브러쉬만을 만들 수 있으며 색상만 인자로 전달하면 된다.

두 번째 함수는 색상 뿐 아니라 무늬를 지정할 수 있다. 무늬 종류는 아래와 같다.



HS_FDIAGONAL (우하향 줄무늬), HS_BDIAGONAL(좌하향 줄무늬)
 HS_CROSS(바둑판 모양) HS_HORIZONTAL(수평선)
 HS_DIAGCROSS(좌우하향 줄무늬) HS_VERTICAL(수직선)

간단한 사용 예를 보자.

```

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);
            HBRUSH brush = CreateHatchBrush(HS_CROSS, RGB(255,0, 0));
            HBRUSH old = (HBRUSH)SelectObject(hdc, brush);
            Rectangle(hdc, 10,10,100,100);
            DeleteObject(SelectObject(hdc, old));
            EndPaint(hwnd, &ps);
        }
        return 0;

        case WM_DESTROY:
            PostQuitMessage(0);
    }
}

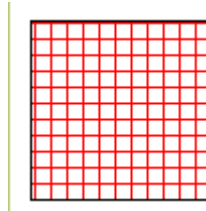
```

```
        return 0;

    }

    return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

예제 5.10 브러쉬



5. 4 기타

5.4.1 그리기 모드

화면에 무엇인가 그려져 있는 상황에서 그 위에 다른 무엇인가를 출력하면 원래 그려져 있던 그림은 새로 그려지는 그림에 덮여 지워진다.

그리기 모드란 도형이 그려질 때 원래 그려져 있던 그림과 새로 그려지는 그림과의 관계를 정의하는 것이다.

아래 그림은 흑백 그림을 4가지 비트 연산을 수행한 결과이다.



가로로 놓인 막대가 원래 그려져 있던 그림이며 새로로 놓여져 있는 막대가 새로 그려지는 그림이다.

- COPY : 새로 그려지는 그림이 기존 그림을 덮어 버리는 것
- OR : 두 그림의 대응되는 비트를 OR연산하여 새로 값을 써 넣음
(마치 새로 그려지는 그림이 셀로판지에 그려져 기존 그림위에 얹히는 것과 같은 효과)

- AND : 두 그림의 교집합 영역만 그려짐
- XOR : 두 그림 중 겹쳐지는 부분이 반전되는 효과

만약, 여러 가지 색상을 가지고 있는 경우는 위 보다 훨씬 더 복잡하다.

5.4.2 그리기 모드 종류

윈도우에서 사용하는 디폴트 그리기 모드는 R2_COPYPEN 모드이다. 그래서 그려지는 그림이 기존 그림을 덮어버린다. 아래 함수를 이용하면 그리기 모드를 변경할 수 도 있고 설정된 정보를 구할 수 있다.

int SetROP2(HDC hdc, int fnDrawMode);

int GetROP2(HDC hdc);

fnDrawMode 의 종류는 아래와 같다.

그리기 모드	설명
R2_BLACK	항상 검정색이다.
R2_WHITE	항상 흰색이다.
R2_NOP	아무런 그리기도 하지 않는다.
R2_NOT	원래의 그림을 반전시킨다.
R2_COPYPEN	원래의 그림을 덮어버리고 새 그림을 그린다.
R2_NOTCOPYPEN	새 그림을 반전시켜 그린다.
R2_MERGEPEEN	OR연산으로 두 그림을 합친다.
R2_MASKPEN	AND연산으로 겹치는 부분만 그린다.
R2_XORPEN	XOR연산으로 겹치는 부분만 반전시킨다.

다음 예제를 살펴보자.

```

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    static POINTS start, end;
    static BOOL bNowDraw = FALSE;
    switch( msg )
    {

```

```

case WM_LBUTTONDOWN:
{
    start = MAKEPOINTS(lParam);
    end = start;
    bNowDraw = TRUE;
}
return 0;

case WM_MOUSEMOVE:
{
    if(bNowDraw)
    {
        HDC hdc = GetDC(hwnd);
        SetROP2(hdc, R2_NOT);
        MoveToEx(hdc, start.x, start.y, NULL);
        LineTo(hdc, end.x, end.y);

        end = MAKEPOINTS(lParam);
        MoveToEx(hdc, start.x, start.y, NULL);
        LineTo(hdc, end.x, end.y);
        ReleaseDC(hwnd, hdc);
    }
}
return 0;

case WM_LBUTTONUP:
{
    bNowDraw = FALSE;
    HDC hdc = GetDC(hwnd);
    MoveToEx(hdc, start.x, start.y, NULL);
    LineTo(hdc, end.x, end.y);
    ReleaseDC(hwnd, hdc);
}

```

예제 5.11 반전모드_선그리기



그림에서 sx, sy 는 소스 상 start 에 저장된 x, y 좌표 이며,
 $oldx, oldy$ 는 소스 상 end 에 저장된 x, y 좌표 이다.

아래는 사각형을 반전모드로 그리는 예제이다.

```
static POINTS start, end;
switch( msg )
{
case WM_LBUTTONDOWN:
    start = end = MAKEPOINTS( lParam );
    SetCapture( hwnd );
    return 0;
case WM_LBUTTONUP:
    if ( GetCapture() == hwnd )
    {
        ReleaseCapture();

        // 최종선은 R2_COPYPEN으로 그려야 한다.
        HDC hdc = GetDC( hwnd ); // 디폴트 그리기 모드가 R2_COPYPEN 이다.
        SelectObject(hdc, GetStockObject(NULL_BRUSH));
        Rectangle(hdc, start.x, start.y, end.x, end.y);
        ReleaseDC( hwnd, hdc );
    }
    return 0;
case WM_MOUSEMOVE:
    if ( GetCapture() == hwnd ) // 캡처중일때만..
```



```

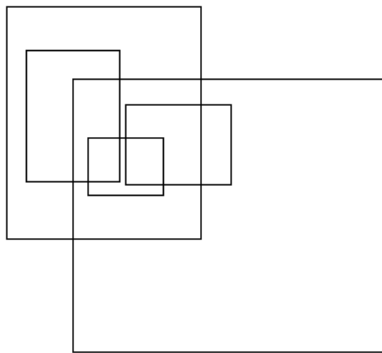
{
    POINTS pt = MAKEPOINTS( lParam );
    HDC hdc = GetDC( hwnd );

    SetROP2( hdc, R2_NOTXORPEN ); // 그리기 모드 변경
    SelectObject(hdc, GetStockObject(NULL_BRUSH));
    Rectangle(hdc, start.x, start.y, end.x, end.y);
    Rectangle(hdc, start.x, start.y, pt.x, pt.y);

    end = pt; // 현재 점을 Old에 보관
    ReleaseDC( hwnd, hdc );
}
return 0;

```

예제 5.12 반전모드_도형 그리기



위 예제는 반전모드-선 그리기 예제와 알고리즘이 동일하다.

5.4.3 윈도우 좌표 체계

윈도우 좌표 체계는 아래 2개로 구분할 수 있다.

- 논리좌표
대부분의 GDI 함수가 사용하는 좌표계이다.
- 장치좌표
비 GDI 함수가 사용하는 좌표계이다.
메시지의 파라미터로 들어오는 좌표이다.

Client 좌표, Window좌표, Screen 좌표 등이 있다.

논리 좌표와 장치 좌표 사이 변환해 주는 함수가 있다. (자세한 사항은 MSDN 참조)

DPTOLP()

LPTODP()

장치 좌표인 Client 좌표와 Screen 좌표를 변환해 주는 함수가 있다.(자세한 사항은 MSDN 참조)

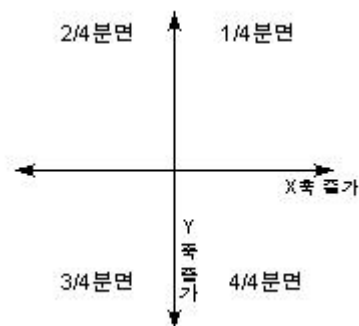
ClientToScreen()

ScreenToClient()

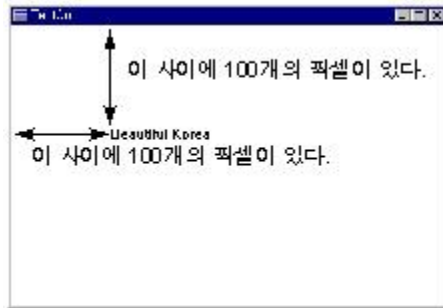
윈도우는 그래픽 기반의 GUI운영체제, 모든 출력은 점 단위로 이루어진다. 출력함수들은 출력 위치를 지정하는 좌표가 필요하다.

윈도우는 좌표체계는 기본값을 픽셀 단위를 사용하는데 픽셀이란 그래픽을 이루는 최소단위이다.

윈도우의 작업 영역 좌상단이 원점(0,0)이며, X축 좌표는 오른쪽으로 갈수록 증가하고 Y축 좌표는 아래쪽으로 갈수록 증가 한다.



만약 화면의 (100, 100)에 문자열을 출력하면 아래 그림과 같이 출력이 이루어 진다. 화면 출력은 아래에서 설명하는 맵핑 모드를 통해 변경시킬 수 있다.



5.4.4 맵핑 모드

맵핑 모드는 주어진 좌표가 화면상의 실제 어디에 해당하는지를 결정하는 방법이다. 즉, 논리좌표를 물리 좌표로 변환하는 방법을 의미한다.

예를 들어 마우스 이벤트를 통해서 좌표값을 얻었다고 가정하자. 이는 물리좌표이며, 만약 화면상 마우스 클릭한 위치에 사각형을 출력하기 위해서는 API 함수를 호출해야 한다.

이 때 논리 좌표를 전달해 줘야 하며, 마우스로 입력된 물리좌표를 논리좌표로 변환해서 전달해 주어야 한다.

디폴트 맵핑 모드는 MM_TEXT 로 되어 있으며 , 추가 사항은 아래 표와 같다.

맵핑 모드	단위	X축 증가	Y축 증가
MM_TEXT	픽셀	오른쪽	아래쪽
MM_LOMETRIC	0.1mm	오른쪽	왼쪽
MM_HIMETRIC	0.01mm	오른쪽	왼쪽
MM_LOENGLISH	0.01인치	오른쪽	왼쪽
MM_HIENGLISH	0.001인치	오른쪽	왼쪽
MM_TWIPS	1/1440인치	오른쪽	왼쪽
MM_ISOTROPIC	가변	가변	가변
MM_ANISOTROPIC	가변	가변	가변

아래 함수를 통해 맵핑 모드를 변경하거나 획득할 수 있다.

```
int SetMapMode(HDC hdc, int fnMapMode);
int GetMapMode(HDC hdc);
```

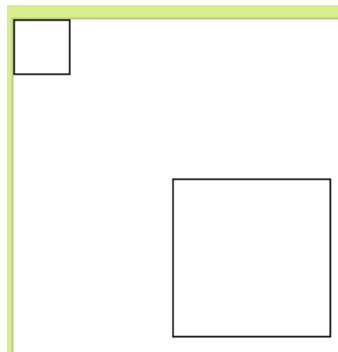
아래 예제를 살펴보자.

```
case WM_PAINT:
{
    PAINTSTRUCT ps;
    HDC hdc          = BeginPaint(hwnd, &ps);

    Rectangle(hdc, 100, 100, 200, 200);
    // 논리 1= 0.1mm(물리), y축 증가 위 )
    SetMapMode(hdc, MM_LOMETRIC);
    Rectangle(hdc, 0, 0, 100, -100);

    EndPaint(hwnd, &ps);
}
return 0;
```

예제 5.13 맵핑 모드



큰 사각형은 디폴트 맵핑모드로 출력한 결과이고, 좌상단 작은 사각형은 맵핑모드를 변경한 후 출력한 결과이다. 이 때 사용한 모드는 MM_LOMETRIC 으로서 단위는 0.1mm이고, y축 이동 방향이 반대이다.

6. Mouse

6.1 마우스 관련 정보

윈도우 환경에서 사용되는 각종요소들의 크기및 환경 설정 값을 얻을 때 는 아래의 함수를 사용한다.

int GetSystemMetrics(int nIndex);

nIndex 의 값에 따라 다양한 정보도 얻을 수 있다. 마우스 관련 정보를 얻기 위해서는 아래의 상수값 들을 사용하면 된다.

nIndex	Description
SM_MOUSEPRESENT	마우스가 존재 하지 않으면 0, 존재하면 0이 아닌 값이 리턴된다.
SM_MOUSEWHEELPRESENT	마우스 휠이 존재하면 0이 아닌 값이 리턴된다.
SM_CMOUSEBUTTONS	마우스 버튼의 개수가 리턴된다.
SM_CX(CY)DOUBLECLK	더블 클릭으로 처리되는 간격이 리턴된다.

```
case WM_LBUTTONDOWN:
{
    BOOL bPresent = GetSystemMetrics(SM_MOUSEPRESENT);
    BOOL bWheel = GetSystemMetrics(SM_MOUSEWHEELPRESENT);
    int nBtn = GetSystemMetrics(SM_CMOUSEBUTTONS);
    int scx = GetSystemMetrics( SM_CXSCREEN );
    int scy = GetSystemMetrics( SM_CYSCREEN );
    TCHAR info[128];
    wsprintf( info, _TEXT("%s %s is installed. (%d Buttons)\\n"),
        (bWheel ? _TEXT("Wheel") : _TEXT("")),
        ( bPresent ? _TEXT("Mouse") : _TEXT("No Mouse" )), nBtn);

    TCHAR temp[64];
```

```

wprintf( temp, _TEXT("WnScreen Resolution : %d * %d"), scx, scy);

wcscat( info, temp);

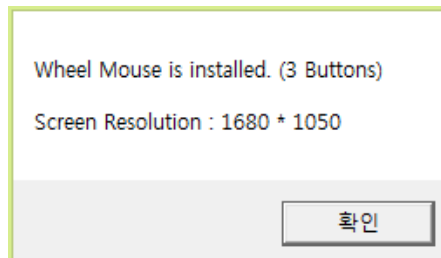
MessageBox(NULL, info , TEXT(""), MB_OK);

}

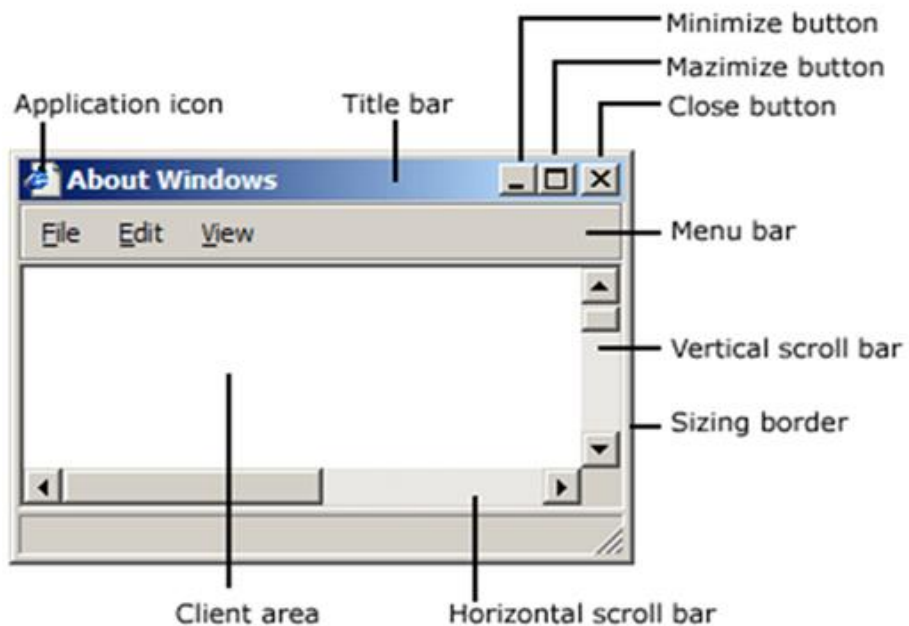
return 0;

```

예제 6.1 시스템 정보 얻기



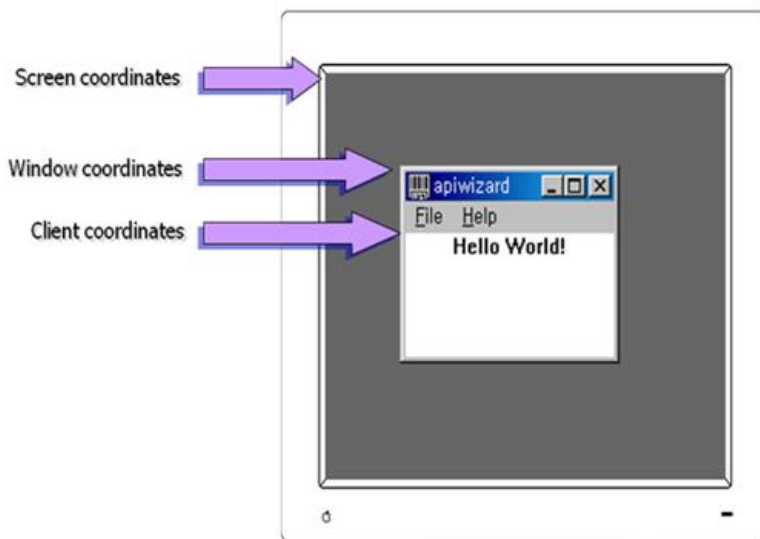
6. 2 윈도우의 명칭



윈도우는 크게 Client 영역과 Non-Client 영역으로 나눌 수 있다.

Client 영역	사용자로부터 마우스, 키보드 입력을 받고 사용자에게 출력을 보여 주기 위한 영역
Non Client 영역	Client 이외의 영역 캡션바, 메뉴, 최대화, 최소화, 닫기버튼, 스크롤바, 상태바 등으로 구성된다.

6. 3 장치(물리) 좌표계



윈도우에서는 아래의 3가지의 물리좌표계가 있다.

- 스크린 좌표계 : 모니터의 왼쪽 상단을 기준(0,0)으로 한 좌표계
- 윈도우 좌표계 : 윈도우의 왼쪽 상단을 기준(0,0)으로 한 좌표계
- 클라이언트 좌표계 : 윈도우 클라이언트 영역의 왼쪽 상단을 기준(0,0)으로 한 좌표계

아래의 함수들을 사용하면 클라이언트 좌표와 스크린 좌표간의 변환을 할 수 있다.

```
BOOL ClientToScreen(    HWND hWnd, // handle to window
                        LPPOINT lpPoint // client coordinates );
```

***BOOL ScreenToClient(**HWND** hWnd, // handle to window
LPPOINT lpPoint // screen coordinates);***

6. 4 마우스 메시지

6.4.1 마우스 메시지 종류

마우스 메시지는 크게 Client 영역 메시지와 non 클라이언트 영역 메시지로 구분할 수 있다. 그 종류는 아래에서 살펴보자.

Client 영역 메시지	Non client 영역 메시지	기타
WM_LBUTTONDOWN	WM_NCLBUTTONDOWN	WM_NCHITTEST WM_MOUSEWHEEL
WM_LBUTTONUP	WM_NCLBUTTONUP	
WM_LBUTTONDOWNBLCLK	WM_NCLBUTTONDOWNBLCLK	
WM_RBUTTONDOWN	WM_NCRBUTTONDOWN	
WM_RBUTTONUP	WM_NCRBUTTONUP	
WM_RBUTTONDOWNBLCLK	WM_NCRBUTTONDOWNBLCLK	
WM_MBUTTONDOWN	WM_NCMBUTTONDOWN	
WM_MBUTTONUP	WM_NCMBUTTONUP	
WM_MBUTTONDOWNBLCLK	WM_NCMBUTTONDOWNBLCLK	
WM_MOUSEMOVE	WM_NCMOUSEMOVE	

6.4.2 더블 클릭 메시지 발생하기

OS가 더블 클릭 메시지를 발생하려면 많은 비용이 든다. 그래서 더블클릭 메시지는 기본적으로는 발생하지 않는다. 특정 우원도우를 만들고 더블클릭 메시지를 받고 싶다면 아래처럼 클래스 스타일을 변경해야 한다.

```
WNDCLASS wc;

.....

wc.style = CS_DBLCLKS; // 더블클릭 메시지를 나오게 하는 클래스 스타일
RegisterClass(&wc);
```


6.4.3 마우스 부가 정보(클라이언트 영역)

마우스 메시지가 발생할 때 IParam에는 메시지 발생 당시의 마우스 좌표가 들어 있다. 상위 16비트에는 Y 좌표가 하위 16비트에는 X좌표가 들어 있다.

또한, 클라이언트 영역 마우스 메시지일 경우 클라이언트 좌표로, 비클라이언트 영역 메시지일 경우는 스크린 좌표로 들어온다. 아래의 매크로를 사용하면 IParam으로 부터 쉽게 x, y좌표를 얻을 수 있다.

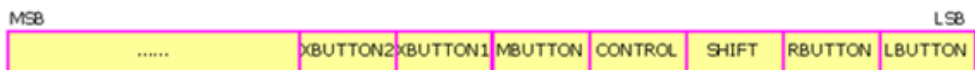
아래는 클라이언트 영역에서 발생한 마우스 이벤트 정보이다.

IParam



POINTS pt = MAKEPOINTS(IParam);
int x = LOWORD(IParam);
int y = HIWORD(IParam);

wParam



wParam에는 마우스 버튼의 상태와 키보드 조합 키(Shift, Ctrl)의 상태가 전달된다. 조합 키 상태는 다음 값들과 비트 연산을 해보면 알 수 있다.

값	설명
MK_CONTROL	Ctrl 키가 눌러져 있다.
MK_LBUTTON	마우스 왼쪽 버튼이 눌러져 있다.
MK_RBUTTON	마우스 오른쪽 버튼이 눌러져 있다.
MK_MBUTTON	마우스 중간 버튼이 눌러져 있다.
MK_SHIFT	Shift 키가 눌러져 있다.

예제를 살펴보자.

```
case WM_LBUTTONDOWN:
{
```

```

POINTS pt = MAKEPOINTS(IParam);

POINT pt1 = { LOWORD(IParam), HIWORD(IParam)};

TCHAR buf[50];

wsprintf(buf, TEXT("(%d:%d)WrWn(%d:%d)", pt.x, pt.y, pt1.x, pt1.y);

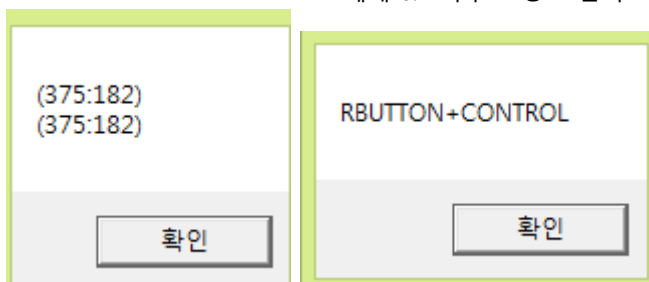
MessageBox(NULL, buf, TEXT(""), MB_OK);

}

case WM_RBUTTONDOWN:
{
    if(wParam & MK_SHIFT)
    {
        MessageBox(NULL, TEXT("RBUTTON+SHIFT"), TEXT(""), MB_OK);
    }
    else if(wParam & MK_CONTROL)
    {
        MessageBox(NULL, TEXT("RBUTTON+CONTROL"), TEXT(""), MB_OK);
    }
}
}

```

예제 6.2 마우스 정보 얻기



좌측 그림은 마우스 왼쪽 버튼 클릭시 출력결과이고, 우측 그림은 마우스 오른쪽 버튼과 컨트롤 조합키 클릭시 출력결과이다.

6.4.4 마우스 부가 정보(비클라이언트 영역)

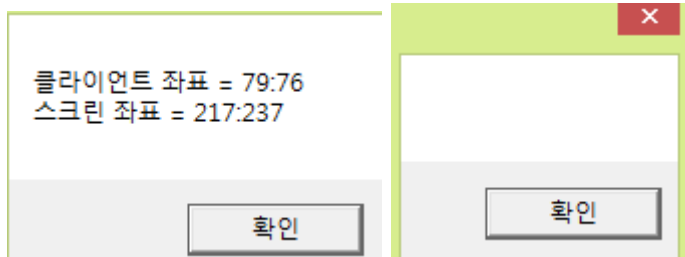
비 클라이언트 영역 마우스 메시지의 경우는 IParam의 경우 윈도우 좌표가 들어오고, wParam에는 hit-test code가 들어온다.

아래 예제를 살펴보자.

```
case WM_LBUTTONDOWN:
{
    POINT pt = {LOWORD(lParam), HIWORD(lParam)};
    POINT pt1 = pt;
    ClientToScreen(hwnd, &pt);
    TCHAR temp[256];
    wsprintf(temp, TEXT("클라이언트 좌표 = %d:%d\r\n스크린 좌표 = %d:%d"),
        pt1.x, pt1.y, pt.x, pt.y);
    MessageBox(NULL, temp, TEXT(""), MB_OK);
}
return 0;

case WM_NCLBUTTONDOWN:
{
    MessageBox(NULL, TEXT(""), TEXT(""), MB_OK);
}
return 0;
```

예제 6.3 비클라이언트 영역 마우스 정보 얻기

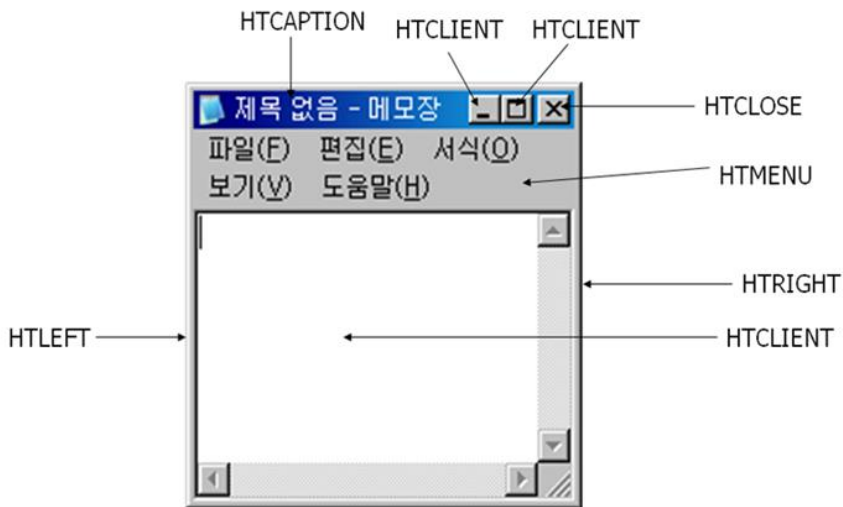


먼저 클라이언트 영역에서 마우스 왼쪽 버튼을 클릭한 결과는 좌측 메시지 박스이다. 마우스 왼쪽 버튼을 클릭하면 lParam으로부터 마우스 좌표 정보를 가져오고, 해당 좌표를 스크린 좌표로 변경한다.

비클라이언트 영역에서 마우스 왼쪽 버튼을 클릭하면 우측에 보이는 메시지 박스가 출력된다. 실제 프로그램 우상단에 있는 종료버튼을 클릭해도 해당 기능이 작동하지 않고 메시지 박스가 출력되는 것을 볼 수 있다.

6. 5 Hit Test Code

HitTest Code 는 현재 커서가 윈도우의 어느 부분에 있는지를 알려주는 상수이다. WM_NCHITTEST 메시지가 DefWindowProc()으로 전달될 때 이코드가 리턴된다.



Cursor가 움직이거나 마우스 버튼을 누르거나 놓을 때, System 은 커서아래 있는 윈도우(마우스를 캡처한 경우 캡처한 윈도우)에게 WM_NCHITTEST 메시지를 보낸다.(Sent). 이때 대부분 윈도우 프로시저는 이 메시지를 직접 처리하지 않고 DefWindowProc()으로 보내는데 DefWindowProc()은 커서의 좌표를 조사해서 커서가 현재 윈도우의 어느 부분에 커서가 있는지 나타내는 Hit Test Code를 리턴한다.

WM_NCHITTEST의 결과로 얻어진 Hit Test Code가 다음 마우스 메시지를 결정하는 데 사용된다.

Control 키를 누른 상태에서 클라이언트영역에서 마우스의 왼쪽 버튼을 누르면 윈도우를 이동하려면 아래처럼 한다.

```
case WM_NCHITTEST:
{
    DWORD code = DefWindowProc(hwnd, msg, wParam, lParam);

    if ( code == HTCLIENT && GetKeyState(VK_CONTROL) < 0 )
        code = HTCAPTION;

    if ( code == HTLEFT )    code = HTRIGHT;
```

```
        return code;
    }
```

예제 6.4 HitTest Code 변경하기

6. 6 마우스 캡처

일반적으로 마우스 메시지는, 메시지가 발생할 당시 커서의 아래 있는 윈도우에게 전달된다. 하지만 SetCapture() 함수를 사용 하므로서 이런 행동을 변경할 수 있다.

특정윈도우가 SetCapture() 함수를 사용해서 마우스를 캡처할 경우, 모든 마우스 메시지는 마우스를 캡처한 윈도우에게로 전달된다.

마우스 캡처 해제하기

마우스 캡처는 아래의 3가지 경우에 해지된다.

- ReleaseCapture() 함수를 호출 할 경우.
- 다른 윈도우가 마우스를 캡처 할 경우.
- 사용자가 다른 스레드가 만든 윈도우를 클릭 할 경우.

마우스 캡처가 해지될 경우, 캡처를 잃은 윈도우에 WM_CAPTURECHANGED 메시지가 전달된다. 이때 lParam에는 새롭게 마우스를 캡처 한 윈도우의 핸들이 들어 있다.

마우스 캡처 종류

- Foreground Capture.
Foreground 윈도우가 마우스를 캡처 한 경우. 모든 윈도우에서 발생한 마우스 메시지를 받을 수 있다.
- Background Capture.
Background 윈도우가 마우스를 캡처 한 경우. 동일 스레드내의 마우스 메시지만을 받을 수 있다.

마우스 캡처는 커서가 윈도우를 벗어나더라도 계속 메시지를 받고자 할때 유용하게 사용된다.

주소 BUTTONDOWN에서 캡처를 BUTTONUP에서 Release를 해준다.

현재 어떤 윈도우가 캡처를 했는지를 알아내려면 GetCapture() 함수를 사용한다.

마우스가 캡처 되어 있을 경우 WM_NCHITTEST, WM_SETCURSOR 메시지는 발생되지 않는다.

```

void PrintWindowInfo( HWND hwnd )
{
    TCHAR Info[1024] = {0};
    TCHAR temp[256];
    RECT rcWin;
    GetClassName( hwnd, temp, 256);
    wprintf( Info, TEXT("Window Class : %s\n"), temp);
    GetWindowText( hwnd, temp, 256);
    wprintf( Info + wcslen(Info), TEXT("Window Caption : %s\n"), temp);
    GetWindowRect( hwnd, &rcWin );
    wprintf( Info + wcslen(Info), TEXT("Window Position : (%d,%d)-(%d,%d)"),
        rcWin.left, rcWin.top, rcWin.right, rcWin.bottom);
    MessageBox( 0, Info, TEXT("Window Info"), MB_OK);
}

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_LBUTTONDOWN:
            SetCapture( hwnd );
            return 0;
        case WM_LBUTTONUP:
            if ( GetCapture() == hwnd )
            {
                ReleaseCapture();
                POINT pt;
                GetCursorPos( &pt );
                HWND hwndDest = WindowFromPoint( pt );
                PrintWindowInfo( hwndDest );
            }
            return 0;
    }
}

```

```

case WM_MOUSEMOVE:
{
    short xPos, yPos;

    TCHAR temp[256];

    xPos = LOWORD(lParam);
    yPos = HIWORD(lParam);

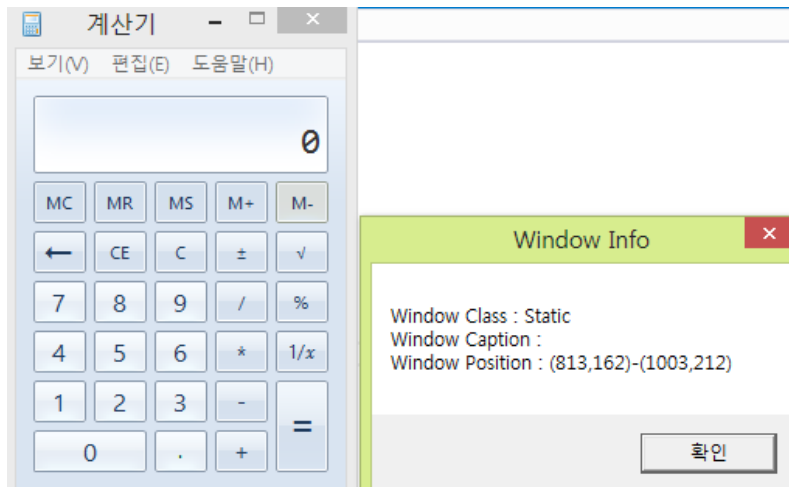
    wsprintf( temp, TEXT("Cursor Position : (%d, %d)"), xPos, yPos);

    SetWindowText( hwnd, temp);
}

return 0;

```

예제 6.5 마우스 캡처



7. Keyboard

7.1 키보드 메시지

7.1.1 전달되는 코드 값

윈도우 환경에서는 주로 마우스가 많이 사용되지만 문자를 입력받기 위한 가장 기본적인 입출력 장치는 키보드이다. 키보드로부터 입력이 발생했을 경우 윈도우즈는 포커스를 가진 프로그램에게 키보드 메시지(WM_CHAR, WM_KEYDOWN등)를 보내주며 프로그램은 이 메시지를 받아 키보드 입력을 처리한다.

여기서 포커스란 입력 초점이라는 뜻이며 키보드 입력을 받아들일 수 있는 상태라는 뜻이다. 포커스를 가진 프로그램이란 활성화되어 있는 윈도우를 말하며 한 번에 오직 하나의 프로그램만 활성화된다.

만약 키보드 입력이 발생하게 되면 입력된 키 입력 정보가 전달되는 데 이 때 전달되는 정보를 아래와 같이 구분할 수 있다.

- 스캔 코드

키보드를 누를 때 마다 하드웨어적으로 발생하는 키보드의 고유 코드 번호이다.

장치에 종속적이다. 따라서 하드웨어에 따라 키 값이 틀리다.

보통 자판 배열 순서를 갖는다. 예를 들어 Q는 16, W는 17, E는 18 등이다.

- 가상 키 코드

키보드 종류에 상관없이 각 키에 부여된 고유한 코드 값이다.

일반적으로 알파벳 순서로 코드 키 값이 부여되어 있다.

장치에 독립적이다. 따라서 하드웨어에 상관없이 동일 값이 생성된다.

대 소문자 구별이 안된다. 예를 들어 'A' 와 'a' 는 둘 다 65라는 키 값으로 전달된다.

- 아스키 코드

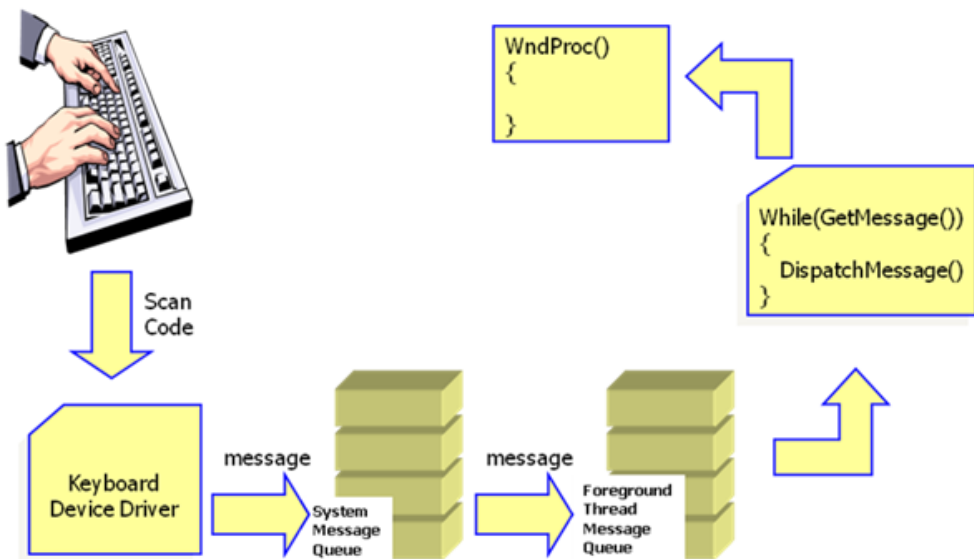
각 문자에 부여되는 코드값이다.

따라서 대문자 소문자가 구별된다.

7.1.2 키보드 메시지가 전달되는 과정

아래 그림을 살펴 보면 사용자가 키보드로 키입력을 하면 이벤트가 발생(스캔 코드 값 생성)한다. 발생한 스캔 코드는 키보드 디바이스 드라이버를 거쳐 시스템에서 발생하는 모든 메시지를 담는 System Message Queue 에 저장된다.

저장된 메시지는 모든 애플리케이션이 독립적으로 가지고 있는 Message Queue 에 전달되고 API 코드에서 작성된 메시지 루프에서 해당 메시지를 읽어 메시지 프로시저에 최종 전달되게 된다.



Foreground Thread

시스템 메시지 큐와 Foreground Thread 메시지 큐 사이에는 RIT라는 메시지 분배 역할을 하는 객체가 존재한다. RIT는 내부적으로 각 어플리케이션의 큐에 연결되어 있는 스레드의 ID를 보관하고 있다.

이와 같이 RIT와 연결되어 있는 스레드를 Foreground 스레드라고 한다. RIT는 시스템 메시지 Queue에서 키보드메시지를 꺼낼 경우 무조건 자신과 연결된 Foreground스레드의 메시지 Queue로 메시지를 전달한다.

Keyboard Focus

한 개의 스레드가 여러 개의 윈도우를 만들 경우 스레드의 메시지 Q에 전달된 메시지는 입력 포커스를 지닌 윈도우에게 전달된다. 아래 함수로 윈도우의 입력 포커스를 변경할 수 있다.

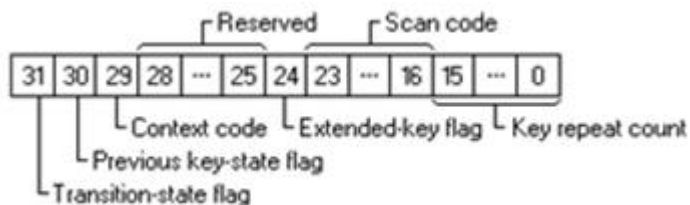
HWND SetFocus(HWND);

7.1.3 키보드 메시지 처리하기

WM_KEYDOWN, WM_KEYUP

사용자가 키보드를 누를 때 WM_KEYDOWN, WM_KEYUP 메시지가 발생한다. 이 때 wParam에는 가상 키 코드가 lParam에는 반복 횟수, scancode 및 여러 가지 추가 정보가 들어 온다.

아래는 lParam으로 전달된 정보들을 표현한 그림이다.



아래 예제는 해당 정보를 획득하는 방법을 보여준다.

```
#define MBox(x) MessageBox( 0, x, TEXT(""), MB_OK)
#define GetScanCode(x) ( ( x >> 16) & 0x00FF)

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_KEYDOWN:
        {
            TCHAR temp[128];
```

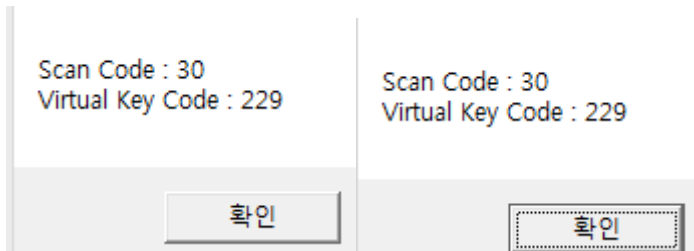
```

wprintf(temp, TEXT("Scan Code : %d\nVirtual Key Code : %d\n"), GetScanCode( IP
aram), wParam);

MBox( temp );
}

```

예제 7.1 WM_KEYDOWN



위 결과 이미지는 'a' 와 'A'를 입력했을 때 결과창이다. 즉, 스캔 코드 값과 가상키 코드 값은 대소문자 동일하다는 것을 볼 수 있다.

예제를 보면 가상 키 코드 값은 아래와 같이 얻을 수 있음을 알 수 있다.

int vCode = (int)wParam;

스캔 코드값은 lParam 의 16~23 비트에 있다. 따라서 아래와 같이 얻을 수 있다.

int sCode = (lParam & 0x00FF0000) >> 16;

아래는 가상키 코드표이다. 즉, 가상키 코드란 시스템에 장착된 키보드의 종류에 상관없이 키를 입력받기 위해 만들어진 코드 값인 것을 알 수 있다.

가상키 코드	값	키
VK_BACK	08	Backspace
VK_TAB	09	Tab
VK_RETURN	0D	Enter
VK_SHIFT	10	Shift
VK_CONTROL	11	Ctrl
VK_MENU	12	Alt
VK_PAUSE	13	Pause
VK_CAPITAL	14	Caps Lock
VK_ESCAPE	1B	Esc
VK_SPACE	20	스페이스
VK_PRIOR	21	PgUp
VK_NEXT	22	PgDn
VK_END	23	End
VK_HOME	24	Home
VK_LEFT	25	좌측 커서 이동키
VK_UP	26	위쪽 커서 이동키
미하생략		미하생략

위 의 예제에서 확인 한 것같이 숫자 및 영문자의 가상 키 코드값은 아스키 코드와 동일하다. 따라서 매크로 상수는 정의 되어 있지 않으므로 아스키 코드와 wParam을 바로 비교하면 된다.

```
static POINTS pt = {100, 100};

switch( msg )
{
case WM_KEYDOWN:      {
    switch(wParam){
        case VK_LEFT: pt.x -= 10; break;
        case VK_RIGHT:pt.x += 10; break;
        case VK_UP:   pt.y -= 10; break;
        case VK_DOWN: pt.y += 10; break;
    }
    HDC hdc = GetDC(hwnd);
    TextOut(hdc, pt.x, pt.y, TEXT("#"),1);
    ReleaseDC(hwnd, hdc);
}
return 0;
```

예제 7.2 WM_KEYDOWN으로 방향키 제어하기

WM_SYSKEYDOWN, WM_SYSKEYUP

사용자가 ALT 키와 다른 키보드를 같이 누를 때 WM_SYSKEYDOWN, WM_SYSKEYUP 메시지가 발생한다. 이 메시지는 반드시 DefWindowProc()으로 전달해야 한다. 만약 전달하지 않을 경우 ALT+F4 같은 키보드 명령이 동작하지 않는다.

```
// ALT+F4 등의 시스템 명령을 금지 시킨다.
case WM_SYSKEYDOWN:    return 0;
```

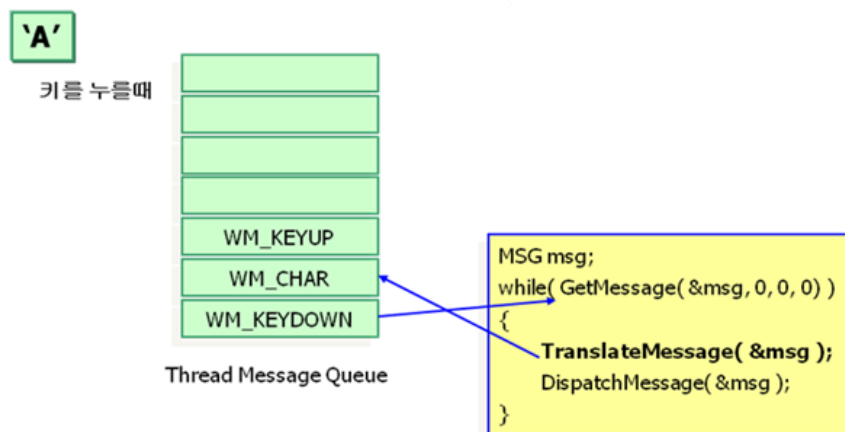
WM_CHAR

WM_KEYDOWN 메시지가 발생할 때 wParam, lParam 을 통해서 많은 정보를 얻을 수 있다. 하지만 문자 코드는 얻을 수 없다.

만약 WM_KEYDOWN 메시지를 아래 함수로 전달하면 아래 함수가 WM_CHAR 메시지를 생성해 준다. 보통 아래 함수는 메시지 루프에 작성한다.

BOOL TranslateMessage(const MSG *lpMsg);

이 때 WM_CHAR 메시지의 wParam 에는 가상 키 코드가 아닌 문자 코드(아스키 코드)가 들어 있다. 단 Cap, Num, 방향키 등은 문자키가 아니므로 WM_CHAR 메시지가 발생하지 않는다.



예제를 살펴보자.

```
#define MBox(x) MessageBox( 0, x, TEXT(""), MB_OK)

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    TCHAR temp[128];

    switch( msg )
    {
        case WM_CHAR:

            wsprintf(temp, TEXT("Character Code : %d\n"), wParam);
```

```
MBox( temp );
return 0;
```

예제 7.3 WM_CHAR 사용하기

Character Code : 97

Character Code : 65

확인

확인

위 결과 화면은 'a' 와 'A' 키를 누른 결과이다.

즉 각 종류에 따른 키는 아래의 이벤트에서 처리한다.

- 문자키는 WM_CHAR 메시지에서 처리한다.
- 기능키는 WM_KEYDOWN 메시지에서 처리한다.
- 방향키는 WM_KEYDOWN 에서만 발생한다.

7.1.4 키보드 상태 조사하기

아래 함수로 키보드 상태를 조사할 수 있다.

현재 메시지가 발생한 순간의 특정 키보드 상태를 조사한다.

SHORT GetKeyState(int nVirtKey);

현재 키보드의 상태를 조사한다.

SHORT GetAsynckKeyState(int vKey);

모든 가상 키의 상태를 얻어온다.

BOOL GetKeyboardState(PBYTE lpKeyState)

예제를 확인해 보자.

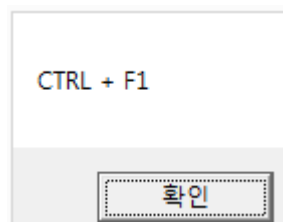
```
#define MBox(x) MessageBox( 0, x, TEXT(""), MB_OK)
#define IsKeyPress(vk) ( GetKeyState(vk) & 0xFF00)
#define IsKeyToggle(vk) ( GetKeyState(vk) & 0x00FF)
```

```

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_KEYDOWN:
            if ( wParam == VK_F1 && IsKeyPress(VK_CONTROL) )
            {
                MBox( TEXT("CTRL + F1") );
            }
            return 0;
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc( hwnd, msg, wParam, lParam);
}

```

예제 7.4 키보드 상태 조사



8. Timer

8. 1 시간 정보 획득하기

8.1.1 SystemTimer, LocalTime

현재 시간을 구하려면 SYSTEMTIME 구조체와 아래의 2가지 함수를 사용한다.

현재 지역 시간을 구한다.

```
void GetLocalTime( LPSYSTEMTIME lpSystemTime )
```

세계 표준 시간(UTC)를 구한다.

```
void GetSystemTime( LPSYSTEMTIME lpSystemTime )
```

또한 아래 함수들을 사용하면 SYSTEMTIME 구조체를 문자열로 변환 할 수 있다.

```
int GetTimeFormat( LCID Locale, DWORD dwFlags, CONST SYSTEMTIME *lpTime,  
LPCTSTR lpFormat, LPTSTR lpTimeStr, int cchTime );
```

```
int GetDateFormat( LCID Locale, DWORD dwFlags, CONST SYSTEMTIME *lpDate,  
LPCTSTR lpFormat, LPTSTR lpDateStr, int cchDate);
```

아래 예제는 현재 시간을 지역시간 혹은 세계표준 시간으로 얻어서 문자열로 출력한 예 이다.

```
case WM_BUTTONDOWN:
{
    SYSTEMTIME st;
    GetSystemTime(&st);
    TCHAR buf[50];

    GetDateFormat(LOCALE_USER_DEFAULT, 0, &st, TEXT("yyy년 M월 d일"), buf, 50);
    SetWindowText(hwnd, buf);
}
return 0;
```



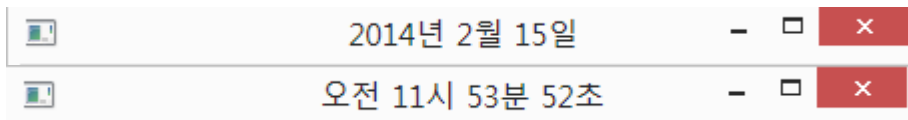
```

case WM_RBUTTONDOWN:
{
    SYSTEMTIME st;
    GetLocalTime(&st);
    TCHAR buf[50];

    GetTimeFormat(LOCALE_USER_DEFAULT, 0, &st, TEXT("tt h시 m분 s초"), buf, 50);
    SetWindowText(hwnd, buf);
}
return 0;

```

8.1 시간 정보 얻기



마우스 왼쪽 버튼과 마우스 오른쪽 버튼을 클릭한 결과이다.

아래 함수들을 사용하면 지역을 표준 시간으로 또는 표준시간을 지역시간으로 변경할 수 있다. 개인별 실습을 해보길 바란다.

```

BOOL SystemTimeToTzSpecificLocalTime(
    LPTIME_ZONE_INFORMATION lpTimeZone,
    LPSYSTEMTIME lpUniversalTime, LPSYSTEMTIME lpLocalTime );

```

```

BOOL TzSpecificLocalTimeToSystemTime(
    LPTIME_ZONE_INFORMATION lpTimeZoneInformation,
    LPSYSTEMTIME lpLocalTime, LPSYSTEMTIME lpUniversalTime );

```

8.1.2 SystemTimer, LocalTime

파일의 생성, 접근, 변경 시간 등은 FILETIME 구조체를 사용해서 보관된다.

FILETIME은 세계표준시간 1601년 1월 1일 0시 를 기준으로 얼마만큼 경과가 되었는지가 100 nano second 단위로 기록 되어 있다. FAT 파일 시스템에서는 지역 시간으로 NTFS 파일 시스템에서는 표준 시간을 기준으로 기록 한다. FILETIME 구조체는 아래와 같은 64 비트 값을 보관 할수 되어 있다.

특정 파일의 시간을 구하려면 아래 함수를 사용한다.

```
BOOL GetFileTime( HANDLE hFile, LPFILETIME lpCreationTime,  
LPFILETIME lpLastAccessTime,  
LPFILETIME lpLastWriteTime );
```

그리고 FILETIME 을 SYSTEMTIME 으로 변경하려면 아래 함수를 사용한다.

```
BOOL FileTimeToSystemTime( const FILETIME* lpFileTime,  
LPSYSTEMTIME lpSystemTime );
```

아래 예제는 특정 파일의 생성, 마지막, 접근, 마지막 변경된 시간을 출력해 주는 예제이다.

```
void PrintTime(const SYSTEMTIME* pst, HDC hdc, int y)
{
    TCHAR date[256] = {0};
    TCHAR time[256] = {0};
    GetDateFormat( LOCALE_USER_DEFAULT, 0, pst, TEXT("yyyy년 M월d일"), date, 256);
    GetTimeFormat( LOCALE_USER_DEFAULT, 0, pst, TEXT("tt h시 m분 s초"), time, 256);
    TextOut(hdc, 10,y+10, date, wcslen(date));
    TextOut(hdc, 10,y+30, time, wcslen(time));
}

BOOL PrintFileTime(HANDLE hFile, HDC hdc)
{
    FILETIME ftCreate, ftAccess, ftWrite;
    SYSTEMTIME stUTCCreate, stLocalCreate, stUTCAccess, stLocalAccess, stUTCWrite, stLocalWrite;

    if (!GetFileTime(hFile, &ftCreate, &ftAccess, &ftWrite)) return FALSE;
```

```

        FileTimeToSystemTime(&ftWrite, &stUTCCreate);
        SystemTimeToTzSpecificLocalTime(NULL, &stUTCCreate, &stLocalCreate);
        FileTimeToSystemTime(&ftAccess, &stUTCAccess);
        SystemTimeToTzSpecificLocalTime(NULL, &stUTCAccess, &stLocalAccess);
        FileTimeToSystemTime(&ftWrite, &stUTCWrite);
        SystemTimeToTzSpecificLocalTime(NULL, &stUTCWrite, &stLocalWrite);

        //Creation Time
        PrintTime( &stLocalCreate, hdc, 0);

        //Last-Access Time :
        PrintTime( &stLocalAccess, hdc, 50);

        //"Last-Write Time :
        PrintTime( &stLocalWrite, hdc, 100);

    return TRUE;
}

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
    case WM_LBUTTONDOWN:
    {
        HANDLE hFile = CreateFile( TEXT("C:\\Windows\\system32\\cmd.exe"),
            GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE, 0, OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL, 0);

        HDC hdc = GetDC(hwnd);
        PrintFileTime( hFile, hdc );
        ReleaseDC(hwnd, hdc);
    }
    }

    return 0;
}

```

예제 8.2 파일시간연기

2012년 7월 26일
오후 12시 20분 43초

2012년 7월 26일
오전 10시 6분 38초

2012년 7월 26일
오후 12시 20분 43초

위 예제는 순서대로 파일이 생성된 시간, 마지막 접근한 시간, 마지막으로 수정된 시간을 보여준다.

자세한 내용은 개별 소스 분석해 보길 바란다.

8.1.3 Windows Time

운영체제는 부팅한 후 경과된 시간을 1ms 단위로 기록하고 있는데 이를 Tick Count 라고 한다. 아래 함수를 사용하면 얻어 올수 있다.

DWORD GetTickCount()

1ms 단위로 기록 되기는 하지만 실제로는 3.1의 경우 16ms, 95/98/me는 55ms, NT/2000의 경우는 10ms마다 한번씩 기록한다. 또한 32비트 크기로 기록 되므로 2^{32} ms후 즉 49.7일 후 에는 다시 0으로 된다.

아래 예제는 운영체제를 부팅한 후 얼마만큼의 시간이 경과했는지를 보여주는 예제 이다.

```
case WM_LBUTTONDOWN:
{
    DWORD count = GetTickCount();

    int second      = count / 1000;

    int minute      = ( second % 3600 ) / 60;

    int hour        = ( second % 86400 ) / 3600;

    int day         = second / 864000;

    TCHAR buf[50];
```

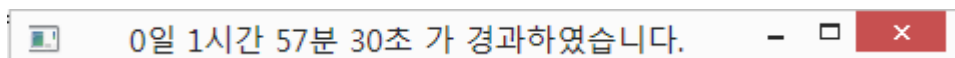
```

        wprintf(buf, TEXT( "%d일 %d시간 %d분 %d초 가 경과하였습니다. "),
                    day, hour, minute, second%60);

        SetWindowText(hwnd, buf);
    }
    return 0;

```

예제 8.3 GetTickCount



8. 2 WM_TIMER

키보드나 마우스는 사용자로부터 입력되는 메시지이다. 메시지는 사용자의 입력으로부터 유발되는 것이 보통이지만 사용자의 동작과는 상관없이 발생하는 메시지도 있다. 대표적으로 타이머 메시지인 WM_TIMER 를 들 수 있다. 이 메시지는 한 번 지정해 놓기만 하면 일정한 시간 간격을 두고 연속적으로 계속 발생한다.

즉, 주기적으로 같은 동작을 반복해야 한다면 여러 번 나누어 해야 할 일이 있을 때 이 메시지를 사용한다.

아래의 함수를 사용한다.

***UNIT SetTimer(HWND hWnd, UINT nIDEvent, UINT uElapse,
TIMERPROC lpTimerFunc);***

- 기능 : 타이머를 생성
- hWnd : 타이머 메시지를 받을 윈도우
- IDEvent : 타이머의 번호를 지정 (여러개의 타이머를 사용할 경우 겹치지 않도록 번호를 부여해야 한다.
WM_TIMER 메시지에서 타이머를 구분하기 위한 표식으로 사용된다.
- uElapse : 1/1000초 단위로 타이머의 주기를 설정 한다.
(1000 -> 타이머 메시지가 1초에 한번씩 발생)
- lpTimerFunc : 타이머 메시지가 발생할 때마다 호출될 함수를 지정한다.

NULL 을 입력할 경우 WM_TIMER 를 반복 호출한다.

- 리턴값은 타이머를 소유하는 윈도우 없이 타이머가 만들어졌을 경우만 의미가 있다.
즉 SetTimer의 첫번째 인수가 NULL일 경우에 한해 특별하게 사용 된다.

KillTimer(HWND hWnd, UINT ulEvent)

- 기능 : 설치된 타이머를 제거
- hWnd : 이 타이머를 소유한 윈도우 핸들
- ulEvent : 타이머 ID
타이머 ID는 SetTimer의 두번째 인수로 지정한 값을 의미한다.

타이머는 시스템 전역 자원이므로 한번 설정해 놓으면 윈도우가 파괴되어도 파괴되지 않고 계속 남아 있게 된다. 따라서 프로그램이 종료될 때 자신이 설정해 놓은 타이머는 직접 파괴해 주어야 한다.

```
case WM_CREATE:    {
                    SetTimer(hWnd, 1, 1000, NULL);
                }
return 0;

case WM_TIMER:     {
                    if(wParam == 1)
                    {
                        SYSTEMTIME st;
                        GetLocalTime(&st);
                        TCHAR buf[20];
                        wsprintf(buf, TEXT("현재 시간 : %d:%d:%d"),
                                st.wHour, st.wMinute, st.wSecond);
                        SetWindowText(hWnd, buf);
                    }
                }
return 0;
```

예제 8.4 Timer 사용

위 프로그램은 처음 실행한 직 후에는 시간이 보이지 시간이 출력되지 않다가 1초가 경과한 직후부터 타이틀바에 출력된다. 이는 WM_TIMER가 최초 호출되는 시점이 1초 후이기 때문이다. 이 문제를 해결하려면 강제로 WM_TIMER를 발생시켜야 한다.

```
case WM_CREATE:    {
    SetTimer(hwnd, 1, 1000, NULL);
    SendMessage(hwnd, WM_TIMER, 1, NULL); //추가된 부분
}
return 0;

case WM_TIMER:    {
    if(wParam == 1)    {
        SYSTEMTIME st;
        GetLocalTime(&st);
        TCHAR buf[20];
        wsprintf(buf, TEXT("현재 시간 : %d:%d:%d"),
            st.wHour, st.wMinute, st.wSecond);
        SetWindowText(hwnd, buf);
    }
}
return 0;
```

예제 8.5 Timer 사용 개선

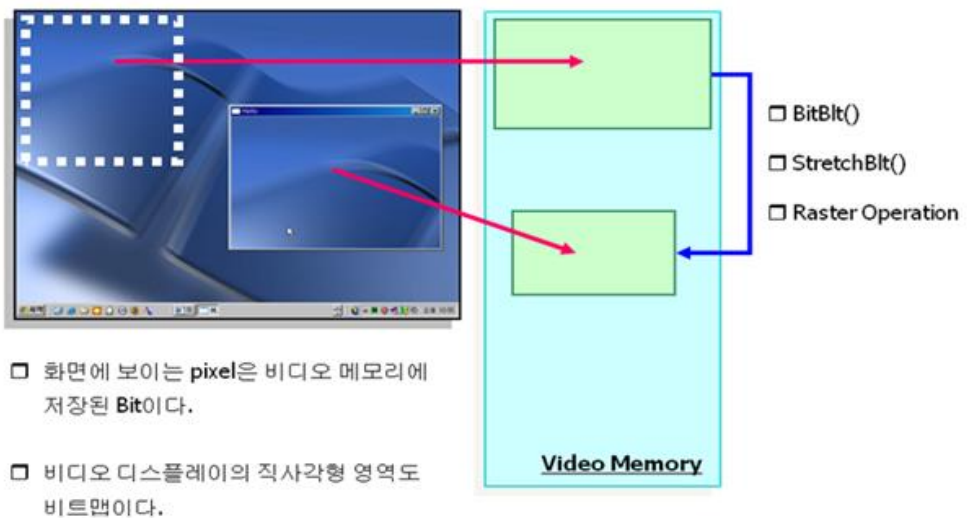
9. Bitmap

비트맵이란 이미지를 저장하고 있는 그래픽 오브젝트이다. 미리 그려진 그림의 각 픽셀 색상과 기타 이미지의 크기, 해상도 등의 정보를 가지고 있는 이미지 데이터의 덩어리라고 할 수 있다.

이번 장에서는 비트맵을 다루는 API 함수들을 살펴보고자 한다.

9.1 DC간 고속 복사

DC 간의 영역끼리 이미지를 고속 복사 하는 기능을 제공하는 API함수가 있다.



9.1.1 Bit Block Transfer

우리가 모니터 화면에서 볼 수 있는 그림들은 결국 비디오 메모리에 있는 Bit들이다. 결국 비디오 메모리를 하나의 커다란 비트맵으로 생각할 수 있다. 이때 아래 함수는 비디오 메모리의 특정 영역에 있는 Data를 다른 영역으로 옮길 때 사용된다.

```
BOOL BitBlt(HDC hdcDest,          // handle to destination DC  
            int nXDest,          // x-coord of destination upper-left corner  
            int nYDest,          // y-coord of destination upper-left corner  
            int nWidth,          // width of destination rectangle
```



```

int nHeight, // height of destination rectangle
HDC hdcSrc, // handle to source DC
int nXSrc, // x-coordinate of source upper-left corner
int nYSrc, // y-coordinate of source upper-left corner
DWORD dwRop // raster operation code );

```

아래 예제는 배경화면을 나타내는 DC 에서 윈도우의 ClientDC로 픽셀을 전송하는 예제이다.

```

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdcDisplay, hdcClient;
    static int cxClient, cyClient;
    switch( msg )
    {
        case WM_MOVE:
            InvalidateRect(hwnd, 0, TRUE);
            return 0;

        case WM_SIZE:
            cxClient = LOWORD(lParam);
            cyClient = HIWORD(lParam);
            return 0;

        case WM_ERASEBKGND:
            return TRUE;

        case WM_PAINT:
            hdcDisplay = CreateDC(TEXT("DISPLAY"), 0,0,0);
            hdcClient = BeginPaint( hwnd, &ps);
            BitBlt( hdcClient, 0, 0, cxClient, cyClient,
                    hdcDisplay, 0, 0, SRCCOPY);
            DeleteDC( hdcDisplay );
    }
}

```

```

        EndPaint( hwnd, &ps );

        return 0;

    case WM_DESTROY:

        PostQuitMessage( 0 );

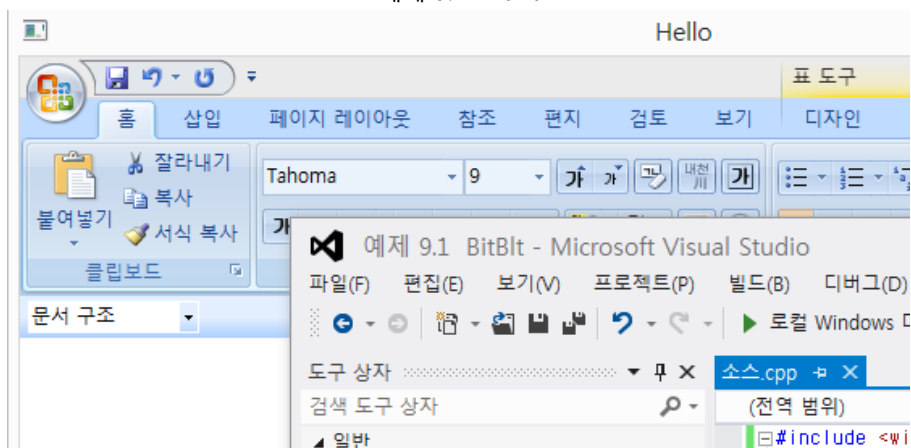
        return 0;

    }

    return DefWindowProc( hwnd, msg, wParam, lParam );
}

```

예제 9.1 BitBlt



BitBlt 함수에서 가장 중요한 점은 2개의 DC는 반드시 호환(Compatible, 색상면(plane)과 픽셀당 비트(bpp)가 동일)되어야 한다.

9.1.2 StretchBlt

BitBlt()함수는 크기를 늘이거나 줄일 수는 없다. 복사하면서 이미지의 크기를 늘리거나 줄이려는 경우는 아래의 함수를 사용한다.

```

BOOL StretchBlt( HDC hdcDest,      // handle to destination DC
                int nXOriginDest, // x-coord of destination upper-left corner
                int nYOriginDest, // y-coord of destination upper-left corner
                int nWidthDest,   // width of destination rectangle
                int nHeightDest,  // height of destination rectangle
                HDC hdcSrc,       // handle to source DC

```

```

int nXOriginSrc, // x-coord of source upper-left corner
int nYOriginSrc, // y-coord of source upper-left corner
int nWidthSrc,   // width of source rectangle
int nHeightSrc,  // height of source rectangle
DWORD dwRop      // raster operation code );

```

예제를 살펴보자.

```

HDC          hdcDisplay, hdcClient;

static int    cxClient, cyClient;

switch( msg )
{
case WM_SIZE:
    cxClient = LOWORD( lParam );
    cyClient = HIWORD( lParam );
    return 0;

case WM_LBUTTONDOWN:
    SetCapture( hwnd );
    return 0;

case WM_LBUTTONUP:
    if ( GetCapture() == hwnd )
        ReleaseCapture();
    return 0;

case WM_MOUSEMOVE:
    if ( GetCapture() == hwnd )
    {
        POINT pt;
        GetCursorPos( &pt );
        hdcDisplay = CreateDC( TEXT( "DISPLAY" ), 0, 0, 0 );
        hdcClient  = GetDC( hwnd );
        StretchBlt( hdcClient, 0, 0, cxClient, cyClient,
                    hdcDisplay, pt.x, pt.y, 100, 100,

```

```

        SRCCOPY);

DeleteDC( hdcDisplay );

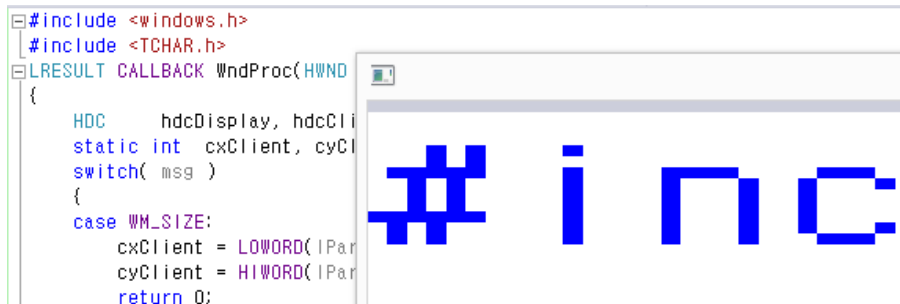
ReleaseDC( hwnd, hdcClient );

}

return 0;

```

예제 9.2 StretchBlt



출력 결과를 살펴보면 BitBlt 와는 달리 바탕화면을 확대해서 출력한 결과를 볼 수 있다. 인수를 보면 복사 대상과 복사원이 모두 폭과 높이를 가지고 있다. 복사원의 지정한 영역이 복사대상의 지정한 영역의 크기만큼 확대되어 출력된다. 물론 복사 대상의 영역이 복사원보다 더 좁다면 축소가 발생할 것이다.

9.1.3 레스터 연산

아래 표에 레스터 연산 관련된 내용을 볼 수 있다. 이 외에도 256개의 레스터 연산이 제공된다.

값	설명
BLACKNESS	대상영역을 검정색으로 가득 채운다.
DSTINVERT	화면을 반전시킨다.
MERGECOPY	소스 비트맵과 대상 화면을 AND 연산한다.
MERGEPAINT	소스 비트맵과 대상 화면을 OR 연산한다.
SRCCOPY	소스 영역을 대상 영역에 복사한다.
WHITENESS	대상영역을 흰색으로 채운다.

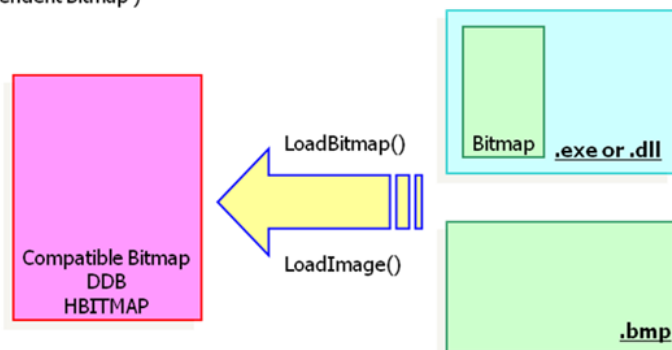
9. 2 비트맵 출력하기

윈도우즈가 지원하는 비트맵 포맷은 두 가지 종류가 있다.

첫 번째는 3.0 이전 버전에 사용하던 DDB(Device Dependent Bitmap)인데 이 비트맵은 출력 장치에 많이 의존되며 몇 가지 제한이 있다. DDB는 이미지의 크기와 색상에 관한 기본적인 정보와 그리고 이미지 데이터만으로 구성되어 있기 때문에 다양한 해상도의 장치에 광범위하게 사용되지 못하며 만들어진 장치 외의 다른 장치에서 출력하면 제대로 출력되지 못하는 경우가 있다.

두 번째는 DIB(Device Independent Bitmap) 이다. 이름이 의미하는 바대로 이 포맷의 비트맵은 장치에 독립적이기 때문에 어디서나 제 모양대로 출력될 수 있다. DIB는 DDB에 비해 색상 테이블과 해상도 정보 등의 추가 정보를 가지므로 장치에 종속되지 않으며 활용 용도가 훨씬 더 광범위하고 호환성이 뛰어나다. 확장자 BMP를 가지는 비트맵 파일들은 모두 DIB 포맷으로 저장된 파일이며 리소스 에디터에서 만들어 주는 비트맵들도 모두 DIB 이다.

- DDB (Device Dependant Bitmap)
- DIB (Device Independent Bitmap)



위 그림을 살펴보자.

- 하위 호환성을 위해 Win32 API는 여전히 DDB를 지원한다. 따라서 DC에 선택될 수 있는 비트맵은 DDB 뿐이다. DIB는 직접 DC에 선택될 수 없기 때문에 프로그램에서 곧바로 사용하기가 어렵다.
- 리소스나 디렉토리에 있는 파일이미지는 DIB 포맷이고, LoadBitmap 함수나 LoadImag

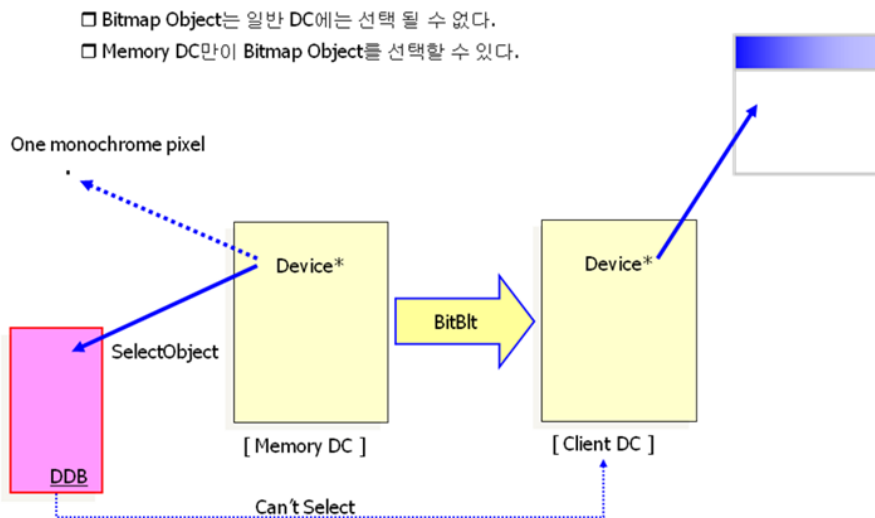
e 함수를 사용하여 로드하면 DDB 포맷으로 자동 변환된다.

9.2.1 LoadBitmap & LoadImage

윈도우즈 비트맵을 곧바로 화면 DC로 출력하는 함수는 제공하지 않는다. 메모리DC를 통해 먼저 출력한 후 고속복사하는 방법을 사용한다.

메모리 DC란 화면 DC와 동일한 특성을 가지며 그 내부에 출력 표면을 가진 메모리 영역이다. 메모리에 있기는 하지만 화면 DC에서 사용할 수 있는 모든 출력을 메모리 DC에서도 할 수 있다.

메모리 DC에 먼저 그림을 그린 후 사용자 눈에 그려지는 과정은 보여주지 않고 그 결과만 화면으로 고속 복사 한다.



메모리 DC를 만들 때는 아래 함수를 사용한다.

HDC CreateCompatibleDC(HDC hdc);

인수로 화면 DC를 주면 이 화면 DC와 동일한 특성을 가지는 DC를 메모리에 만들어 그 핸들값을 리턴해 준다.

비트맵을 읽어 올 때는 아래 함수를 사용한다.

HBITMAP LoadBitmap(HINSTANCE hInstance, LPCTSTR lpBitmapName);

첫 번째 인수는 비트맵 리소스를 가진 인스턴스의 핸들이며 두 번째 인수는 비트맵 리소스의 이름이다. 읽어온 비트맵을 SelectObject 함수로 메모리 DC에 선택하면 메모리 DC의 표면에는 리소스에서 읽어온 비트맵이 그려진다.

마지막으로 이전에 사용했던 고속 복사 함수를 사용하면 화면 출력이 된다.

아래 예제는 이러한 과정을 거쳐 출력하는 예이다.

```
case WM_LBUTTONDOWN:
{
    HDC hdc          = GetDC(hwnd);
    // 화면 DC와 호환(동일색상)되는 메모리 DC를 얻음..
    HDC memDC = CreateCompatibleDC(hdc);
    HBITMAP hBitmap = LoadBitmap(GetModuleHandle(0), MAKEINTRESOURCE(IDB_BITMAP1));
    BITMAP bm;
    GetObject(hBitmap, sizeof(bm), &bm);
    // 메모리 DC에 비트맵 선택
    SelectObject(memDC, hBitmap);
    // 비트맵으로출력
    TextOut(memDC, 5, 5, TEXT("LoadBitmap으로 출력"), 15);
    // 메모리 DC --> 화면 DC//
    POINTS pt = MAKEPOINTS(lParam);
    BitBlt(hdc, pt.x, pt.y, bm.bmWidth, bm.bmHeight, memDC, 0, 0, SRCCOPY);
    // 메모리 제거
    DeleteDC(memDC);
    ReleaseDC(hwnd, hdc);
    DeleteObject(hBitmap);
}
return 0;
case WM_RBUTTONDOWN:
{
```

```

HDC hdc          = GetDC(hwnd);

// 화면 DC와 호환(동일색상)되는 메모리 DC를 얻음..
HDC memDC = CreateCompatibleDC(hdc);

HBITMAP hBitmap = (HBITMAP)LoadImage( 0, // 리소스에서 로드할때만 사용
    TEXT("3.bmp"), IMAGE_BITMAP,
    0, 0, // 커서, Icon load시만 사용
    LR_LOADFROMFILE);

BITMAP bm;
GetObject(hBitmap, sizeof(bm), &bm);

// 메모리 DC에 비트맵 선택
SelectObject(memDC, hBitmap);

// 비트맵으로출력
TextOut(memDC, 5, 5, TEXT("LoadImage로 출력"), 13);

// 메모리 DC --> 화면 DC//
POINTS pt = MAKEPOINTS(IParam);
BitBlt(hdc, pt.x, pt.y, bm.bmWidth, bm.bmHeight, memDC, 0, 0, SRCCOPY);

// 메모리 제거
DeleteDC(memDC);
ReleaseDC(hwnd, hdc);
DeleteObject(hBitmap);
}
return 0;

```

예제 9.3 비트맵 출력하기

LoadBitmap으로 출력

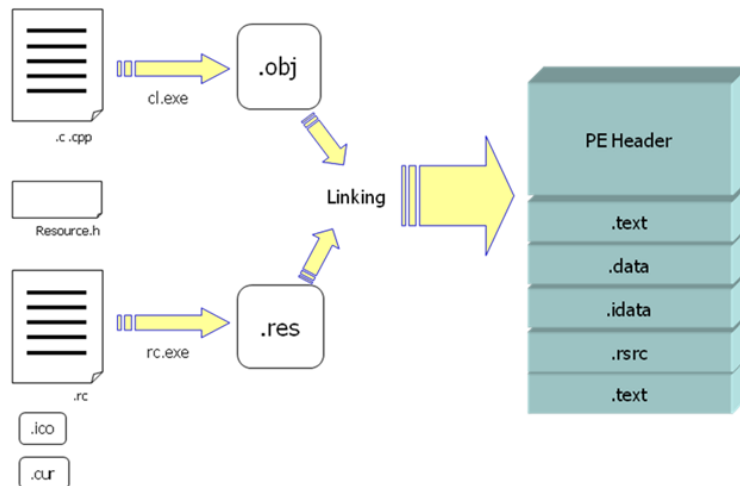


LoadImage으로 출력



10. Resource

프로그램은 코드와 데이터로 구성된다. 데이터는 프로그램의 처리 대상이며 코드는 데이터를 처리하는 수단이다. 데이터의 의미를 확장하여 코드가 아닌 모든 것을 데이터라고 할 때 비트맵, 아이콘, 메뉴, 문자열 등등 프로그램의 논리와 무관한 모든 것들이 데이터에 속한다. 이런 리소스들은 별도의 편집기로 만들어져 컴파일시에 실행파일에 결합된다.



Resource.h / xxx.rc

```

//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file
// Used by Sample.rc
//
#define IDI_MYICON 101

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 102
#define _APS_NEXT_COMMAND_VALUE 40001
#define _APS_NEXT_CONTROL_VALUE 1000
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif
    
```

□ Resource.h

□ xxx.rc

```

////////////////////////////////////
//
// Icon
//
// Icon with lowest ID value placed first to ensure application
// remains consistent on all systems.
IDI_MYICON ICON DISCARDABLE "myicon.ico"
#endif // Korean resources
////////////////////////////////////
    
```

리소스는 보통 리소스 편집기를 이용하여 구성하는 데 리소스 편집기로 구성된 내용들은 resource.h 파일과 *.rc 파일에 저장된다.

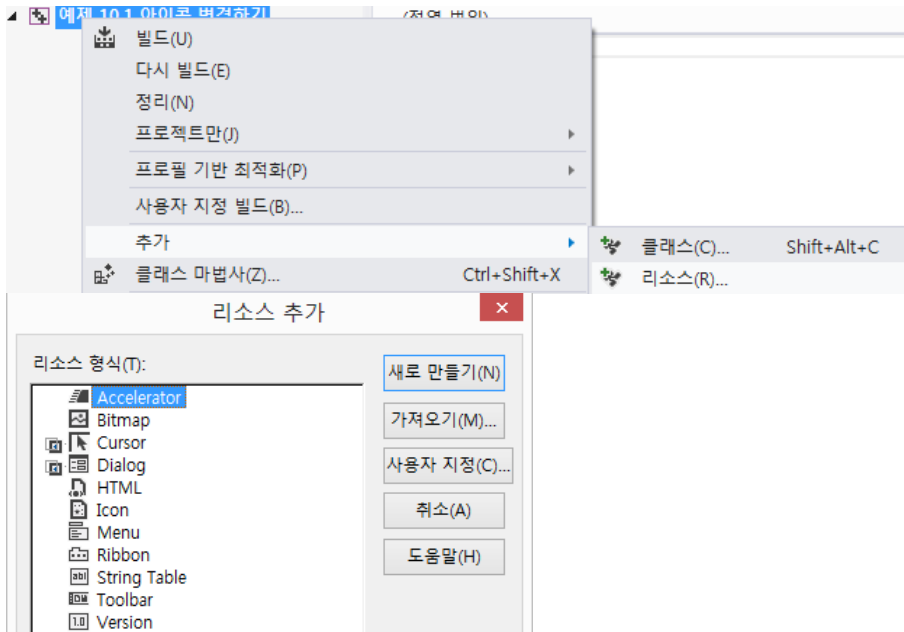
resource.h 파일에는 rc 파일에서 사용하고 있는 리소스들의 ID 값이 정의되며, rc 파일은 리소스 정보들이 스크립트 언어로 저장되게 된다.

간단한 리소스 사용법을 알아 보도록 하자.

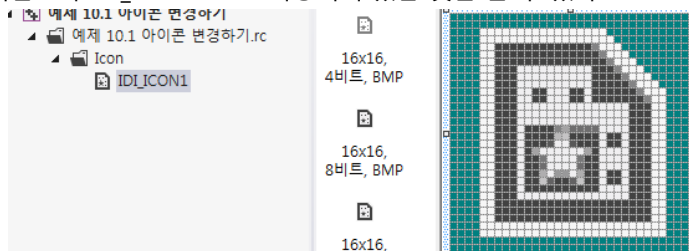
10. 1 아이콘

리소스를 사용하기 위해서는 아래의 과정이 필요하다.

먼저 리소스 뷰에서 원하는 리소스를 추가한다.



만약 아이콘을 추가하면 아래 그림처럼 하나의 아이콘이 추가된 것을 볼 수 있다. 그리고 아이콘의 기본ID가 IDI_ICON1 으로 저장되어 있는 것을 볼 수 있다.



예제를 살펴보자.

먼저 윈도우 클래스 생성시 리소스에 추가한 나만의 아이콘을 등록할 수 있다. 이 기능은 윈도우 클래스 생성시 가능하다.

물론, 동적으로 아이콘을 변경할 수 도 있고 예제와 같이 아이콘을 화면에 출력할 수 도 있다.

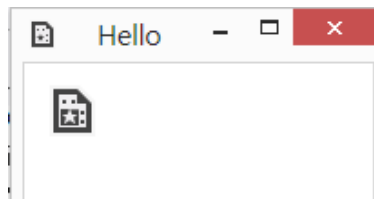
```
#include "resource.h"

int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpCmdLine, int nShowCmd)
{
    // 1. 윈도우 클래스 만들기
    WNDCLASS wc;

    wc.cbWndExtra        = 0;
    wc.cbClsExtra        = 0;
    wc.hbrBackground     = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.hCursor           = LoadCursor(0, IDC_ARROW);
    wc.hIcon             = LoadIcon(hInst, MAKEINTRESOURCE(IDI_ICON1));

    LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam){
        switch( msg )    {
            case WM_LBUTTONDOWN:
            {
                HDC hdc = GetDC(hwnd);
                HICON hIcon = LoadIcon( GetModuleHandle(0), MAKEINTRESOURCE(IDI_ICON1));
                DrawIcon(hdc, 10, 10, hIcon);
                ReleaseDC(hwnd, hdc);
            }
        }
    }
}
```

예제 10.1 아이콘 변경하기



위 그림은 사용자가 마우스 왼쪽 버튼을 클릭한 결과이다.

10. 2 커서

커서 또한 아이콘과 동일하게 리소스에 등록해 사용할 수 있으며, 이때 아래의 함수를 사용한다.

LoadCursor(HINSTANCE hInst, LPCTSTR hCursor);

아래 예제는 윈도우에 설치된 커서를 사용하고 있다.

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam) {
    static HCURSOR h1 = LoadCursorFromFile(TEXT("C:\\Windows\\ Cursors\\size1_i.cur"));
    static HCURSOR h2 = LoadCursorFromFile(TEXT("C:\\Windows\\ Cursors\\size1_i_l.cur"));
    static RECT rc = { 100,100,300,300}; // 클라이언트 좌표.

    switch( msg )
    {
    case WM_SETCURSOR:
        {
            int code = LOWORD( lParam); // hit test code가 들어 있다.
            POINT pt;
            GetCursorPos(&pt);           // 현재 커서의 위치(스크린 좌표)
            ScreenToClient( hwnd, &pt); // 스크린 좌표를 클라이언트 좌표로..
            // 클라이언트 안에서도 특정위치에서는 다른 커서를 사용하는 방법.
            if ( code == HTCLIENT && PtInRect( &rc, pt ) )
            {
                SetCursor( LoadCursor( 0, IDC_ARROW ) );
                return TRUE;
            }
            //그외의 Client에서는 DefWindowProc이 변경해 줄것이다.!!!!!!
            //-----
```

```

        if ( code == HTLEFT || code == HTRIGHT)
        {
            SetCursor( h2);

            return TRUE; // 커서를 변경한 경우 TRUE 리턴
        }

        else if ( code == HTTOP || code == HTBOTTOM )
        {
            SetCursor( h1 );

            return TRUE;
        }
    }

    // 내가 변경하지 않은 다른 부분을 변경하기 위해 아래 함수로
    // 반드시 보내야 한다.
    return DefWindowProc( hwnd, msg, wParam, lParam);

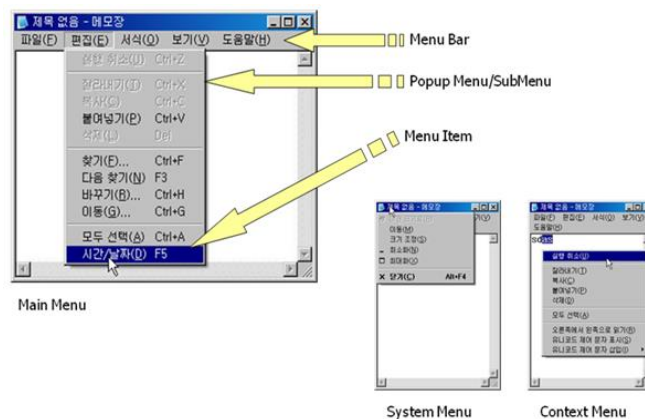
```

예제 10.2 커서 변경하기

10.3 메뉴

윈도우에는 3가지 종류의 메뉴가 있다.

- Main Menu
- System Menu(WS_SYSMENU style)
- Context Menu



10.3.1 Main Menu

메뉴는 윈도우즈용 프로그램이 제공하는 가장 표준적인 사용자 인터페이스이며 프로그램 전체 기능을 총괄적으로 제공해 주는 중요한 기능을 가지고 있다.

메뉴도 기존 리소스들과 마찬가지로 리소스 편집기를 이용해서 생성된다. 그 이후 어떠한 메뉴가 클릭되었다는 사실을 인지하는 코드를 구성하면 된다.

메뉴 부착 방법

메인 메뉴를 프로그램에 연결하는 방법은 다양하다.

[윈도우 클래스를 이용하는 방법]

```
wc.lpszMenuName = MAKEINTRESOURCE(IDR_MENU1);
```

[윈도우 생성 함수를 이용하는 방법]

```
CreateWindowEx(... hMenu, ..., 0); //
```

10번째 파라미터로 메뉴의 핸들값을 필요로 한다.

[API 함수를 사용하는 방법]

```
SetMenu(hwnd, hMenu)
```

이 방법은 메뉴를 동적으로 변경할 수 있다.

그 외 아래의 함수들을 통해 메뉴를 동적으로 변경할 수 있다.

사용 방법은 MSDN 을 참조하기 바란다.

[메인 메뉴의 핸들을 얻는 방법]

```
HMENU GetMenu(HWND hWnd)
```

[첫번째 인자의 서브 메뉴를 얻는 방법]

```
HMENU GetSubMenu(HMENU hMenu, int nPos);
```

[메뉴를 추가하는 방법]

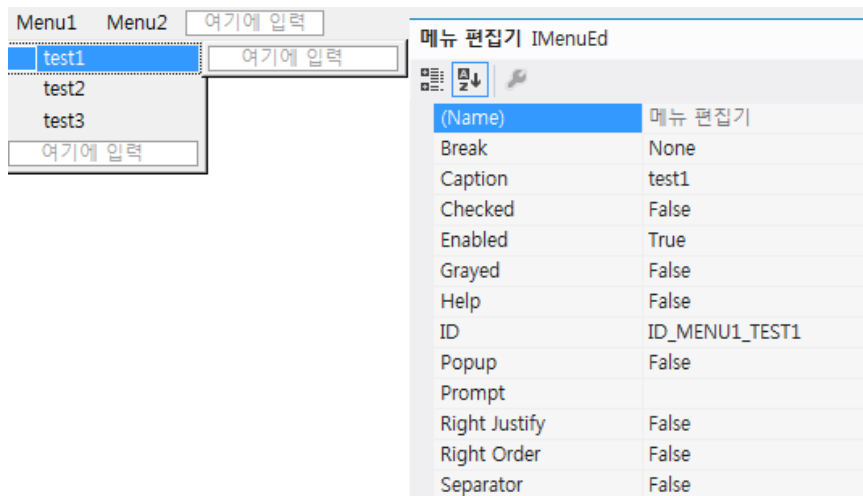
```
BOOL AppendMenu(HMENU hMenu, UINT uFlags, UNIT uIDNewItem,  
LPCTSTR lpNewItem);
```

[메뉴를 삭제하는 방법]

```
BOOL DeleteMenu(HMENU hMenu, UINT uPosition, UINT uFlags);
```

메뉴를 사용하는 예제를 살펴보자. 메뉴는 리소스 편집기로 생성하며 각각의 메뉴는 아래 그림의 좌측 메뉴 편집기를 통해 속성 값을 변경할 수 있다.

편집 하고자 하는 메뉴를 선택하고 마우스 오른쪽 버튼을 클릭하여 속성 메뉴를 선택하면 메뉴 편집기를 볼 수 가 있다.



```
int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpCmdLine, int nShowCmd)
{
    WNDCLASS wc;

    wc.cbWndExtra = 0;
    wc.cbClsExtra = 0;
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.hCursor = LoadCursor(0, IDC_ARROW);
    wc.hIcon = LoadIcon(0, IDI_APPLICATION);
    wc.hInstance = hInst;
    wc.lpfnWndProc = WndProc;
    wc.lpszClassName = TEXT("First");
}
```

```

wc.lpszMenuName          = MAKEINTRESOURCE( IDR_MENU1); //1)

wc.style                  = 0;

// 2. 등록(레지스트리에)
RegisterClass(&wc);

// 3. 윈도우 창 만들기
HMENU hMenu = LoadMenu(hInst, MAKEINTRESOURCE( IDR_MENU1));

HWND hwnd = CreateWindowEx( 0, TEXT("first"), TEXT("Hello"),

                           WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,

                           0, hMenu,                                // 2)

                           hInst, 0);

//-----

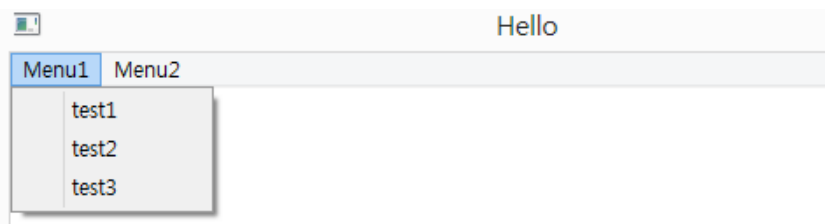
//3) 윈도우 생성 후...

SetMenu(hwnd, hMenu);

//-----

```

예제 10.3 메뉴 추가하기



예제 10.3 은 `_tWinMain` 함수에서 메뉴를 부착할 수 있는 3가지 방법을 보여주고 있다. 보통 1번과 2번 방법을 사용한다.

1번 방법은 윈도우 클래스에 붙이는 방법으로써 이를 통해 만들어지는 모든 윈도우들이 공통적인 메뉴를 갖게 되는 방법이며,

2번 방법은 생성하고자 하는 윈도우 창에 메뉴를 부착하는 방법이다.

메뉴를 동적으로 추가 삭제하는 방법

아래 예제는 마우스 왼쪽 버튼 클릭시 메뉴가 사라지고 마우스 오른쪽 버튼 클릭시 메뉴

가 생성된다.

```
static HMENU      hMenu = 0;

switch( msg )
{
case WM_LBUTTONDOWN:
    if( hMenu == 0 )
    {
        // 윈도우 메뉴의 핸들 얻기
        hMenu = GetMenu(hwnd);

        // 윈도우의 메뉴 부착 ... 0이므로 메뉴가 소멸됨..
        SetMenu(hwnd, 0);
    }

    return 0;

case WM_RBUTTONDOWN:
    if( hMenu != 0 )
    {
        // 윈도우의 메뉴 부착... hMenu...
        SetMenu(hwnd, hMenu);

        hMenu = 0;
    }

    return 0;
}
```

예제 10.4 메뉴 동적 변경하기

```
case WM_LBUTTONDOWN:
{
    HMENU      h = GetMenu(hwnd);

    // 메뉴를 추가... 어디에 추가되는가??
    AppendMenu(h, MF_POPUP, 5000, TEXT("추가메뉴")); // 5000 ==> ID값..

    HMENU h1 = GetSubMenu(h, 1);

    AppendMenu(h1, MF_STRING, 5001, TEXT("BBB"));
    AppendMenu(h1, MF_SEPARATOR, 5002, TEXT(""));
}
```

```

        // 메뉴바에 직접 추가한 경우에는 반드시 메뉴바를 다시 그려야 한다.
        DrawMenuBar(hwnd);

    }

    return 0;

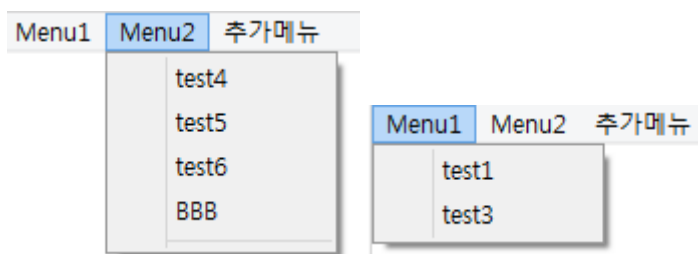
case WM_RBUTTONDOWN:
{
    HMENU h          = GetMenu(hwnd);
    HMENU hSub        = GetSubMenu(h, 0);
    RemoveMenu(hSub, ID_MENU1_TEST2, MF_BYCOMMAND);

}

return 0;

```

예제 10.5 메뉴 동적 변경하기2



실행 결과 이미지를 보면 왼쪽은 마우스 왼쪽 버튼을 클릭한 후 메뉴를 팝업시킨 이미지이며, 오른쪽은 마우스 오른쪽 버튼을 클릭한 후 메뉴를 팝업시킨 이미지이다.

선택된 메뉴를 코드에서 확인하는 방법

```

static int submenu1 = ID_MENU1_TEST1;
static int submenu2 = ID_MENU2_TEST4;

switch( msg )
{
    // PopUp 메뉴가 펼쳐지기 직전...

case WM_INITMENUPOPUP:
{
    //          HMENU hMenu = GetMenu(hwnd);

    HMENU hMenu = (HMENU)wParam;

```

```

        CheckMenuItem(hMenu, ID_MENU1_TEST1,
            submenu1 == ID_MENU1_TEST1 ? MF_CHECKED : MF_UNCHECKED);

        CheckMenuItem(hMenu, ID_MENU1_TEST2,
            submenu1 == ID_MENU1_TEST2 ? MF_CHECKED : MF_UNCHECKED);

        CheckMenuItem(hMenu, ID_MENU1_TEST3,
            submenu1 == ID_MENU1_TEST3 ? MF_CHECKED : MF_UNCHECKED);

        EnableMenuItem(hMenu, ID_MENU2_TEST4,
            submenu2 == ID_MENU2_TEST4 ? MF_GRAYED : MF_ENABLED);

        EnableMenuItem(hMenu, ID_MENU2_TEST5,
            submenu2 == ID_MENU2_TEST5 ? MF_GRAYED : MF_ENABLED);

        EnableMenuItem(hMenu, ID_MENU2_TEST6,
            submenu2 == ID_MENU2_TEST6 ? MF_GRAYED : MF_ENABLED);

    }

    return 0;

case WM_COMMAND:
    {
        switch( LOWORD(wParam))                // ID 조사
        {
            case ID_MENU1_TEST1:                SetWindowText(hwnd, TEXT("ID_MENU1_TEST1"));
                                                submenu1 = ID_MENU1_TEST1; break;

            case ID_MENU1_TEST2:                SetWindowText(hwnd, TEXT("ID_MENU1_TEST2"));
                                                submenu1 = ID_MENU1_TEST2; break;

            case ID_MENU1_TEST3:                SetWindowText(hwnd, TEXT("ID_MENU1_TEST3"));
                                                submenu1 = ID_MENU1_TEST3; break;

            case ID_MENU2_TEST4:                SetWindowText(hwnd, TEXT("ID_MENU2_TEST4"));
                                                submenu2 = ID_MENU2_TEST4; break;

            case ID_MENU2_TEST5:                SetWindowText(hwnd, TEXT("ID_MENU2_TEST5"));
                                                submenu2 = ID_MENU2_TEST5; break;

            case ID_MENU2_TEST6:                SetWindowText(hwnd, TEXT("ID_MENU2_TEST6"));
                                                submenu2 = ID_MENU2_TEST6; break;
        }
    }
}

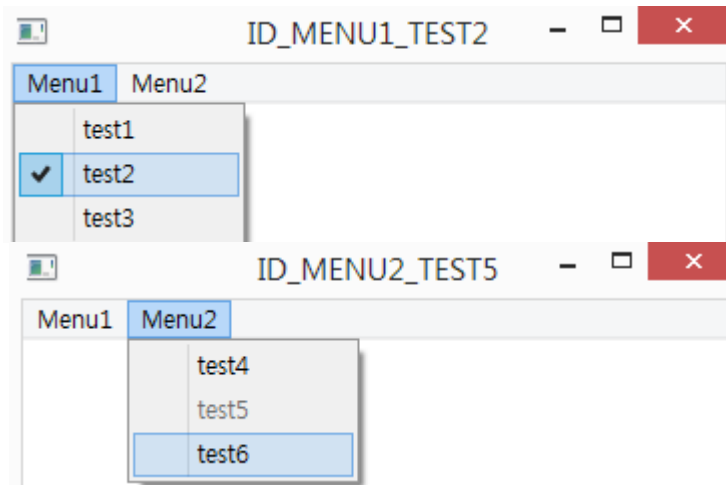
```

```

    }
}
return 0;

```

예제 10.6 메뉴 사용하기



10.3.2 System Menu



WS_SYSMENU Style을 가지는 윈도우의 캡션바에 있는 왼쪽 Icon을 클릭하면 나온다.
아래의 함수를 사용하면 시스템 메뉴의 핸들을 얻거나 시스템 메뉴를 초기화 할 수 있다.

```

GetSystemMenu( hwnd, FALSE );    // 시스템 메뉴의 핸들을 리턴 한다.
GetSystemMenu( hwnd, TRUE);     // 시스템 메뉴를 초기화 한다.

```

WM_SYSCOMMAND

사용자가 시스템 메뉴를 선택할 경우 WM_SYSCOMMAND 메시지가 발생한다.

WPARAM 에는 시스템 메뉴의 ID가 LPARAM에는 커서의 위치가 스크린 좌표로 들어 있다. WPARAM 의 하위 4비트는 시스템에 의해 내부적으로 사용되므로 사용자는 반드시 0xFFFF와 WPARAM를 AND연산해서 ID값을 조사해야 한다.

```
case WM_SYSCOMMAND:  
    id = wParam & 0xFFFF;
```

WM_SYSCOMMAND 메시지가 DefWindowProc으로 전달되면 미리정의된 작업을 수행하게 된다. 미리 정의된 System 명령에는 아래와 같은 것이 있다.

```
SC_CLOSE SC_CONTEXTHELP SC_DEFAULT SC_HOTKEY  
SC_HSCROLL SC_KEYMENU SC_MAXIMIZE SC_MINIMIZE  
SC_MONITORPOWER SC_MOUSEMENU SC_MOVE SC_NEXTWINDOW  
SC_PREVWINDOW SC_RESTORE SC_SCREENSAVE SC_SIZE  
SC_TASKLIST SC_VSCROLL
```

System Menu의 ID

사용자가 System Menu를 추가할 경우 다음을 주의 해야 한다.

- 기존의 시스템 메뉴의 ID가 0xF000~ 이후의 값을 사용하므로 사용자가 추가하는 ID값은 0x0010~0xEFF0 이어야 한다.
- 하위 4비트는 System에 의해 사용되므로 16의 배수로 만들어야 한다.

```
#define WM_SYS_TOPMOST 0x0010
```

10.3.3 Context Menu

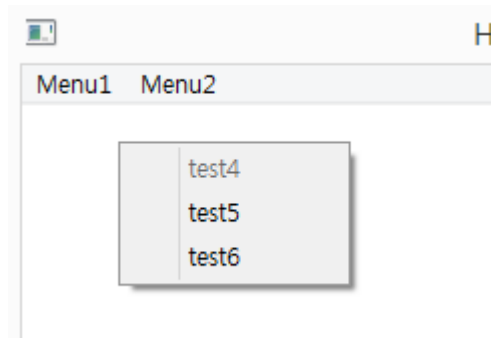
컨텍스트 메뉴란 클라이언트 영역에서 마우스 오른쪽 버튼을 클릭할 때 출력되는 메뉴를 말한다.



WM_RBUTTONDOWN
WM_CONTEXTMENU
TrackPopupMenu

```
case WM_CONTEXTMENU:
{
    HMENU hMenu
    = LoadMenu(GetModuleHandle(0), MAKEINTRESOURCE(IDR_MENU1));
    HMENU hSubMenu = GetSubMenu(hMenu, 1);
    POINT pt = { LOWORD(lParam), HIWORD(lParam) };
    // 스크린 좌표...
    TrackPopupMenu(hSubMenu, TPM_LEFTBUTTON, pt.x, pt.y, 0, hwnd, 0);
}
```

예제 10.7 컨텍스트 메뉴 사용하기



마우스 오른쪽 버튼을 클릭하면 위 화면처럼 컨텍스트 메뉴가 출력된다.

11. 컨트롤

컨트롤이란 사용자와의 인터페이스를 이루는 도구이다. 인터페이스를 이룬다는 말은 사용자로부터 명령과 입력을 받아들이고 출력 결과를 보여준다는 뜻이다.

컨트롤도 하나의 윈도우이다. 윈도우를 만들 때는 WNDCLASS 형의 구조체를 정의하고 RegisterClass 함수를 사용하여 등록한 후 CreateWindow 함수를 호출한다.

그러나 컨트롤은 윈도우즈가 운영체제 차원에서 제공해 주기 때문에 윈도우 클래스를 만들 필요 없이 윈도우즈에 미리 정의되어 있는 윈도우 클래스를 사용하기만 하면 된다. 미리 정의된 윈도우 클래스에는 다음과 같은 종류가 있다.

윈도우 클래스	컨트롤
button	버튼, 체크, 라디오
static	텍스트
scrollbar	스크롤 바
edit	에디트
listbox	리스트 박스
combobox	콤보 박스

이 윈도우 클래스들은 시스템 부팅시에 운영체제에 의해 등록되므로 윈도우 클래스를 따로 등록시킬 필요없이 CreateWindow 함수에 인수로 클래스 이름만 주면 된다.

11.1 컨트롤

11.1.1 컨트롤 생성하기

컨트롤을 생성하기 위해서는 2가지 정보가 필요하다.

첫 번째는 컨트롤의 ID 이고, 두 번째는 컨트롤의 핸들이다.

일반적으로 컨트롤의 ID는 통지메시지를 수신할 때(자식이 부모에게 전달하는 메시지) 사용하며, 컨트롤의 핸들은 부모가 컨트롤에게 메시지를 전달할 때 사용된다.

```

#define IDC_BUTTON 1
#define IDC_EDIT 2
#define IDC_LISTBOX 3
HWND hBtn, hEdit, hListBox;

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_CREATE:
            hEdit = CreateWindow( TEXT("Edit"), TEXT(""),
                                WS_CHILD | WS_VISIBLE | WS_BORDER,
                                10,10,90,20, hwnd, (HMENU)IDC_EDIT, 0,0);
            hBtn = CreateWindow( TEXT("button"), TEXT("Push"),
                                WS_CHILD | WS_VISIBLE | WS_BORDER ,
                                10,40,90,20, hwnd, (HMENU)IDC_BUTTON, 0,0);
            hListBox = CreateWindow( TEXT("listbox"), TEXT(""),
                                WS_CHILD | WS_VISIBLE | WS_BORDER,
                                150,10,90,90, hwnd, (HMENU)IDC_LISTBOX, 0,0);

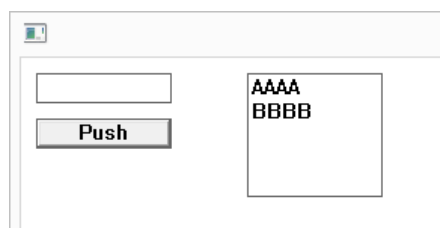
            // Edit가 입력 받는 글자 갯수를 제한한다.- 메시지를 보낸다.
            SendMessage( hEdit, EM_LIMITTEXT, 5, 0);

            // ListBox 에 항목을 추가한다.
            SendMessage( hListBox, LB_ADDSTRING, 0, (LPARAM)TEXT("AAAA"));
            SendMessage( hListBox, LB_ADDSTRING, 0, (LPARAM)TEXT("BBBB"));

            return 0;
    }
}

```

예제 11.1 컨트롤 생성하기



11.1.2 통지 메시지 처리하기

만약, 자식에게 이벤트가 발생하게 되면 부모에게 약속된 형태로 이벤트를 전달해 주는 데 이를 통지 메시지라 한다.

통지 메시지는 메뉴와 동일하게 WM_COMMAND 에서 처리하게 된다. 이 때 컨트롤의 ID, 해당 컨트롤에서 발생된 정보들이 전달되게 된다.

다음 예제를 살펴보자.

```
case WM_COMMAND:

    switch( LOWORD(wParam) ) // ID 조사.
    {
        case IDC_EDIT:                // EditBox 가 보낸경우
            if ( HIWORD(wParam) == EN_CHANGE ) //통지 코드 조사.
            {
                TCHAR s[256];

                GetWindowText( hEdit, s, 256); // Edit 에서 값을 얻는다.
                SetWindowText( hwnd, s); // 부모 윈도우의 캡션을 변경한다.
            }

            break;

        case IDC_BUTTON:
            if ( HIWORD( wParam) == BN_CLICKED ) // 버튼을 누를때 나오는 통지 코드
            {
                TCHAR s[256];

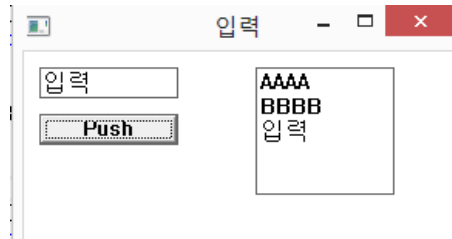
                GetWindowText(hEdit, s, sizeof(s));

                SendMessage( hListBox, LB_ADDSTRING, 0, (LPARAM)s);
            }

            break;
    }

    return 0;
```

예제 11.2 컨트롤 사용하기



위의 출력 결과처럼 사용자가 edit 컨트롤에 글자를 입력하면 즉시 타이틀바에 출력된다.
또한 버튼을 클릭하면 edit 컨트롤에서 글자를 얻어와 리스트 박스에 출력한다.

11. 2 컨트롤(자식 윈도우)의 원리

컨트롤은 윈도우를 만들 때 정의된다고 배웠다. 실제 정의되는 내용이 무엇이고 부모와 자식간의 약속된 메시지 전달이 어떠한 원리로 이루어 지는지 이해하기 위해 아래 예제를 분석해 보기 바란다.

```
#include <windows.h>

void Draw3dRect(HDC hdc, int x, int y, int xx, int yy,
               BOOL down, int width )
{
    COLORREF clrMain = RGB(192,192,192),
             clrLight = RGB(255,255,255),
             clrDark = RGB(128,128,128);

    HPEN hPen1, hPen2, hOldPen;
    HBRUSH hBrush, hOldBrush;

    if(down)
    {
        hPen2 = CreatePen(PS_SOLID,1,clrLight);
        hPen1 = CreatePen(PS_SOLID,1,clrDark);
    }
    else
```

```

{
    hPen1 = CreatePen(PS_SOLID,1,clrLight);
    hPen2 = CreatePen(PS_SOLID,1,clrDark);
}
hBrush = CreateSolidBrush( clrMain );
hOldPen = (HPEN)SelectObject(hdc, hPen1);
hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);

Rectangle(hdc, x , y, xx+1, yy+1);

for(int i=0; i < width; i++)
{
    SelectObject(hdc, hPen1);

    MoveToEx(hdc, xx - 1, y, 0 );
    LineTo(hdc, x, y);
    LineTo(hdc, x, yy - 1 );

    SelectObject(hdc, hPen2);

    MoveToEx(hdc, x, yy,0);
    LineTo(hdc, xx, yy);
    LineTo(hdc, xx, y);

    x++; y++; xx--; yy--;
}
SelectObject(hdc, hOldPen);
SelectObject(hdc, hOldBrush);

DeleteObject(hPen1);
DeleteObject(hPen2);

```

```

        DeleteObject(hBrush);
    }

    // 자식(버튼)이 부모에게 WM_COMMAND를 보낼때 사용할 통지코드(메세지를 보내는 이유)
#define BTN_LCLICK 1
#define BTN_RCLICK 2
#define BTN_LDBLCLK 3
#define BTN_RDBLCLK 4

    // 부모가 자식(버튼)을 만들어 놓고 자식에게 보낼수 있는 메세지를 설계한다.
    // 부모 - 자식 간의 약속.
#define BM_CHANGESTYLE WM_USER + 10
#define BM_CHANGETHICK WM_USER + 11

    //=====
    // 자식용 메세지 처리 함수.
    LRESULT CALLBACK ChildProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
    {
        static BOOL bDown = FALSE;
        static int nThick = 2;

        switch( msg )
        {

            // 부모가 보내준 메세지
            case BM_CHANGETHICK:

                nThick = (int)wParam; // wParam에 두께를 보내주기로 약속...
                InvalidateRect( hwnd, 0, FALSE);

                return 0;

            case WM_LBUTTONDOWN:

```

```

        bDown = TRUE; InvalidateRect( hwnd, 0, FALSE );
        SetCapture( hwnd );
        return 0;

case WM_LBUTTONDOWN:
    if ( GetCapture() == hwnd )
    {
        ReleaseCapture();
        bDown = FALSE;
        InvalidateRect( hwnd, 0, FALSE );

        //=====
        // 자신이 눌렀음을 부모에게 알려준다.
        HWND hParent = GetParent( hwnd );
        UINT id      = GetDlgCtrlID( hwnd );

        SendMessage( hParent, WM_COMMAND,
            MAKELONG( id, BN_LCLICK ), // 하위16: id, 상위16: 통지코드
                                (LPARAM)hwnd );
    }
    return 0;

case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint( hwnd, &ps );

        // 자식은 hdc 를 얻고 자신을 그리기 전에 부모에게 hdc 를 전달해준다.
        // 부모가 자식의 색상을 변경할 권한을 주기 위해서..
        // 이런 용도로 미리 만들어진 메시지가 WM_CTLCLORxxx 이다.

```

```

        // wParam 에는 hdc 를 lParam 에는 자신의 핸들을 넣어 준다.
        HWND hParent = GetParent( hwnd );

        SendMessage( hParent, WM_CTLCOLOBTN, (WPARAM)hdc, (LPARAM)hwnd );

        //=====

        RECT rc;
        GetClientRect( hwnd, &rc );

        Draw3dRect( hdc, 0, 0, rc.right, rc.bottom, bDown, nThick );

        // 부모가 전달한 캡션 문자열을 윈도우 가운데 출력한다
        TCHAR s[256];
        GetWindowText( hwnd, s, 256 );

        SetBkMode( hdc, TRANSPARENT );

        if ( bDown == TRUE )

            OffsetRect( &rc, nThick, nThick ); // 오른쪽 아래로 사각형을

            // nThick 만큼 이동

            DrawText( hdc, s, -1, &rc,

                DT_SINGLELINE | DT_CENTER | DT_VCENTER );

        //-----

```

```

        EndPaint( hwnd, &ps );

    }

    return 0;

// user.chol.com/~downboard/mine.txt 에서 Draw3dRect() 복사해 오세요
}

return DefWindowProc( hwnd, msg, wParam, lParam);
}

//=====
// 부모용 메세지 처리함수.
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{

    static HWND hChild;

    switch( msg )
    {
        // 자식이 자신을 그리기 전에 보내는 메세지 - 부모에게 색을 변경할 기회를 준다.
        case WM_CTLCOLORBTN:
            {

                HDC hdc = (HDC)wParam;
                HWND h = (HWND)lParam;

                if ( hChild == h ) // 자식이 2개 이상이라면 내가 원하는 자식인지 확인
                {

                    SetTextColor( hdc, RGB(255,0,0));

                }

            }

            return 0;
    }

```

```

case WM_LBUTTONDOWN:
{
    static int n = 2;

    ++n;

    // 자식에게 메시지를 보내서 두께를 변경하게 한다
    // wParam : 변경할 두께, lParam : not used
    SendMessage( hChild, BM_CHANGETHICK, n, 0);
}

return 0;

// 자식(버튼)이 보내는 메시지
case WM_COMMAND:
    switch( LOWORD(wParam) ) // 자식 ID
    {
    case 1:
        if ( HIWORD(wParam) == BTN_LCLICK ) // 통지코드 조사
        {

            MessageBox( hwnd, TEXT("Click!!"), TEXT(""), MB_OK);

        }

        break;

    }

    return 0;

case WM_CREATE:

    hChild = CreateWindow( TEXT("CHILD"), TEXT("자식"),

        WS_CHILD | WS_VISIBLE | WS_BORDER,

                                10, 10, 100, 100,

```



```

                                                                    hwnd,

// 부모 윈도우 핸들

(HMENU)1, // 자식은 메뉴를 가질수 없다. ID로 사용

                                                                    0, 0);

        return 0;

case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
}

return DefWindowProc( hwnd, msg, wParam, lParam);
}

int WINAPI WinMain( HINSTANCE hInst, HINSTANCE hPrevInstance,
                                                                    LPSTR    lpCmdLine, int nShowCmd )
{

// 1. 윈도우 클래스 만들기
WNDCLASS wc;

wc.cbWndExtra                = 0;
wc.cbClsExtra                = 0;
wc.hbrBackground             = (HBRUSH)GetStockObject(WHITE_BRUSH);
wc.hCursor                   = LoadCursor(0, IDC_ARROW);
wc.hIcon                     = LoadIcon(0, IDI_APPLICATION);
wc.hInstance                 = hInst;
wc.lpfnWndProc               = WndProc;

wc.lpszClassName             = TEXT("First");
wc.lpszMenuName               = 0;
wc.style                     = 0;

// 2. 등록(레지스트리에)

```

```

RegisterClass(&wc);

//=====

wc.lpfnWndProc = ChildProc;
wc.lpszClassName = TEXT("CHILD");

RegisterClass( &wc ); // 자식용 윈도우 클래스 등록
//=====

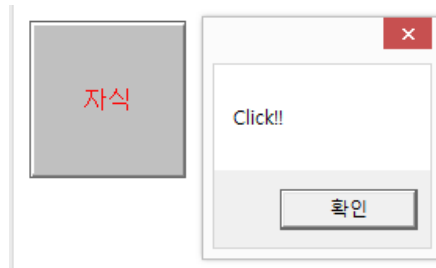
// 3. 윈도우 창 만들기
HWND hwnd = CreateWindowEx( 0, TEXT("first"), TEXT("Hello"),
                           WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                           0, 0, hInst, 0);

// 4. 윈도우 보여주기
ShowWindow(hwnd, SW_SHOW);
UpdateWindow(hwnd);

// 5. Message
MSG msg;
while( GetMessage( &msg, 0, 0, 0 ) )
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return 0;
}

```

예제 11.3 자식 윈도우의 원리



자식이 클릭되었을 때 부모에게 통지한 화면



부모의 클라이언트 영역에서 마우스 왼쪽 버튼을 클릭해서 자식에게 메시지를 전달한 결과 화면

11.3 서브 클래싱

서브 클래싱이란 윈도우 프로시저로 보내지는 메시지를 중간에 가로채는 기법이다. 중간에서 메시지를 조작함으로써 윈도우 모양을 변경하거나 동작을 감시할 수 있다.

아래는 숫자만 입력 가능한 에디트 컨트롤을 구현하기 위해 서브 클래싱을 사용한 예이다.

```

WNDPROC old; // 원래의 EditBox의 메세지 처리함수의 주소를 담을 변수
LRESULT CALLBACK foo( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch( msg )
    {
        case WM_CHAR:
            if ( ( wParam >= '0' && wParam <= '9' ) || wParam == 8 )

```

```

        return CallWindowProc( old, hwnd, msg, wParam, lParam);

        return 0; // 숫자 이외의 경우는 무시한다.
    }

    // 나머지 모든 메세지는 원래의 함수로 전달한다.
    return CallWindowProc( old, hwnd, msg, wParam, lParam);
}

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    static HWND hEdit;
    switch( msg )
    {
        case WM_CREATE:
            hEdit = CreateWindow( TEXT("edit"), TEXT(""), WS_CHILD | WS_VISIBLE | WS_BORDER
                                10,10,200,200, hwnd, (HMENU)1, 0, 0);
            old = (WNDPROC)SetWindowLong( hEdit, GWL_WNDPROC, (LONG)foo );
            SetFocus(hEdit);
            return 0;

            case WM_DESTROY:
                PostQuitMessage(0);
                return 0;
    }

    return DefWindowProc(hwnd, msg, wParam, lParam);
}

```

예제 11.4 서브 클래스링

11.4 공통 컨트롤

공통 컨트롤은 윈도우즈 95 이후에 추가된 컨트롤을 의미하며 개념 및 일반적 사용방법은 표준 컨트롤과 동일하다.

표준 컨트롤과는 약간의 차이점이 있는데 표준 컨트롤은 시스템에 내장되어 있기 때문에 언제든지 사용가능하다. 그러나 공통 컨트롤들은 DLL 에 의해 제공되기 때문에 이 DLL

이 로드되어 있지 않으면 윈도우 클래스가 정의되지 않으며 따라서 컨트롤을 생성할 수 없다. 그래서 공통 컨트롤을 사용하기 위해서는 아래의 초기화 작업이 필요하다.

- 1) CommonCtrl.h 를 포함시킨다.
- 2) WM_CREATE 에서 InitCommonControls(Ex) 함수를 호출해 준다.
- 3) ComCtl32.lib 를 링크시킨다. 아래 전처리 문으로도 가능하다.

```
#pragma comment(lib "comctl32.lib")
```

아래 예제를 통해 사용방법을 익혀보자.

```
// 공용컨트롤 사용시 필요한 h....
#include <comctl.h>
#pragma comment(lib, "comctl32.lib")

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    static HINSTANCE hInst;
    static HWND hProgress;
    static int Pos = 10;
    switch( msg )
    {
        case WM_CREATE:
            // 공용 컨트롤 사용시 컨트롤 초기화...
            InitCommonControls();
            hInst = ((LPCREATESTRUCT) lParam)->hInstance;
            hProgress = CreateWindow(TEXT("msctls_progress32"), TEXT(""),
                                    WS_CHILD | WS_VISIBLE | WS_BORDER | PBS_SMOOTH,
                                    10, 10, 300, 20, hwnd, (HMENU)1, hInst, 0);

            // 컨트롤 초기화
            SendMessage( hProgress, PBM_SETRANGE32, 0, 100);
            SendMessage( hProgress, PBM_SETPOS, 10, 0);

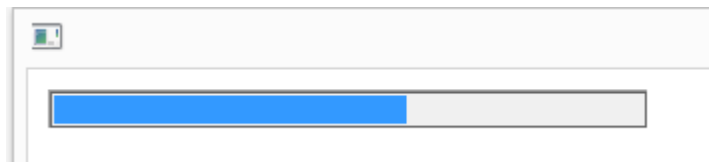
            return 0;
    }
}
```

```

case WM_LBUTTONDOWN:
    Pos += 10;
    SendMessage(hProgress, PBM_SETPOS, Pos, 0);
    return 0;
case WM_RBUTTONDOWN:
    Pos -= 10;
    SendMessage(hProgress, PBM_SETPOS, Pos, 0);
    return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
}
return DefWindowProc(hwnd, msg, wParam, lParam);
}

```

예제 11.5 공통 컨트롤



12. 대화상자

대화상자는 프로그램과 사용자간의 대화, 곧 명령 및 정보 전달을 위한 특별한 윈도우이다. 대화상자에는 버튼, 에디트 등의 컨트롤들이 배치되며 사용자는 대화상자를 호출한 후 컨트롤을 통해 자신의 의사를 표시하고 명령을 내리며 프로그램은 대화상자에 배치된 컨트롤들을 통해 현재 상태를 사용자에게 보여준다.

대화상자는 동작 방식에 따라 모달형과 모달리스형으로 나뉜다.

12.1 모달형 대화상자

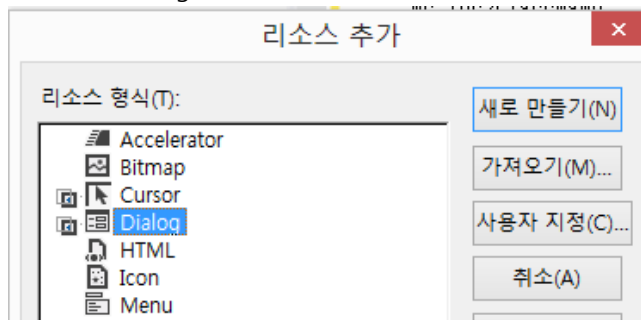
모달(Modal)형은 대화상자를 닫기 전에 다른 윈도우로 전환할 수 없으며 반드시 확인 또는 취소 버튼을 눌러 대화상자를 닫아야 다른 윈도우로 전환할 수 있다.

대화상자를 구현하기 위해서는 2가지가 있어야 한다.

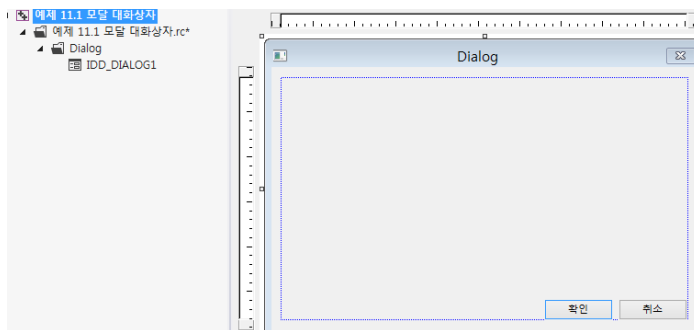
- 대화상자 템플릿 : 대화상자의 모양과 대화상자 내의 컨트롤 배치 상태가 저장되는 리소스 형태로 작성된다. 편집기로 제공되므로 어렵지 않게 디자인 할 수 있다.
- 대화상자 프로시저 : 윈도우 프로시저가 윈도우에서 발생하는 메시지를 처리하는 것과 마찬가지로 대화상자 프로시저는 대화상자에서 발생하는 메시지를 처리한다.

12.1.1 대화상자 템플릿 만들기

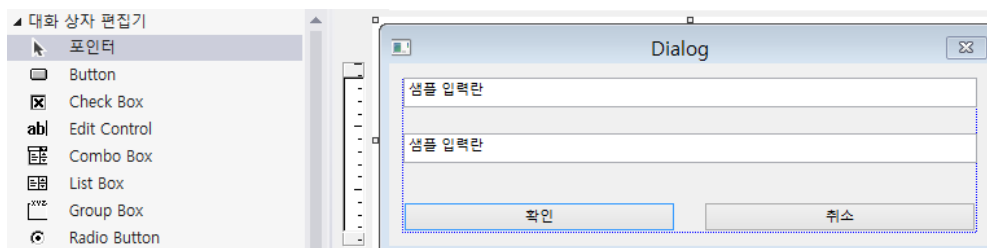
리소스 추가를 선택해서 Dialog 를 새로 만든다.



아래와 같이 IDD_DIALOG1 이라는 기본 ID를 갖는 다이얼로그가 생성된다.



도구상자에서 적절한 컨트롤을 등록시켜보자.



12.1.2 대화상자 프로시저

대화상자 프로시저의 모양은 아래와 같다.

```

BOOL CALLBACK DlgProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam){
    switch(msg)
    {
        case WM_INITDIALOG:
            break;

        case WM_COMMAND:
            switch(LOWORD(wParam))
            {
                case IDCANCEL:
                    EndDialog(hDlg, IDCANCEL);
                    return TRUE;
            }
    }
}

```



```

        return FALSE; // 메시지 처리를 안한 경우..
    }

```

프로시저이기 때문에 리턴값이나 인자는 메인 프로시저와 동일하다. 단, 아래와 같이 몇 가지 차이점이 있다.

- 1) 처리하지 않은 메시지인 경우 return 반드시 FALSE를 해야 한다.
- 2) WM_COMMAND 의 IDCANCEL 이벤트 발생시 반드시 EndDialog 를 호출해야 한다.
이 부분이 Main 프로시저의 종료 코드와 동일한 역할을 수행한다.
- 3) 최초 호출되는 부분은 WM_INITDIALOG 부분이다. 따라서 이 곳에서 초기화를 수행한다.

12.1.3 구현

```

typedef struct tagDATA{
    TCHAR str[20];
    int  num;
}DATA;

BOOL CALLBACK DigProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam){
    static DATA *pData;
    switch(msg)
    {
        // Dialog가 처음 나타날때 발생. 각 컨트롤을 초기화 한다.
        case WM_INITDIALOG:
            {
                pData = (DATA*)lParam;
                SetDlgItemText(hDlg, IDC_EDIT1, pData->str);
                SetDlgItemInt(hDlg, IDC_EDIT2, pData->num, 0);
            }
            break;
        case WM_COMMAND:

```

```

        switch(LOWORD(wParam))
        {
            case IDOK:
                GetDlgItemText(hDlg, IDC_EDIT1, pData->str, sizeof(pData->str));
                pData->num = GetDlgItemInt(hDlg, IDC_EDIT2, 0, 0);
                EndDialog(hDlg, IDOK);
                return TRUE;
            case IDCANCEL:
                EndDialog(hDlg, IDCANCEL);
                return TRUE;
        }
    }

    return FALSE; // 메시지 처리를 안한경우..
}

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam){
    switch( msg )
    {
        case WM_LBUTTONDOWN:
        {
            DATA data = {TEXT("홍길동"), 20 };
            UINT ret = DialogBoxParam( GetModuleHandle(0), // hinstance
                                     MAKEINTRESOURCE( IDD_DIALOG1 ),
                                     hwnd, // 부모
                                     DlgProc, // 메시지 함수.
                                     (LPARAM)&data); // WM_INITDIALOG의 lParam로 전달된다.
            if ( ret == IDOK )
            {
                // 이제 Dialog에 입력한 값은 data에 있다.
                TCHAR buf[128];
                wprintf(buf, TEXT("%s / %d"), data.str, data.num);
                SetWindowText(hwnd, buf);
            }
        }
    }
    return 0;
}

```

예제 11.1 모달 대화상자



첫 번째 이미지는 자식이 호출되었을 때이며 부모가 전달한 데이터가 출력됨을 알 수 있다. 두 번째 이미지는 모달 대화상자에서 정보를 수정해 확인 버튼을 클릭했을 때 부모의 타이틀 바에 자식에게서 전달 받은 정보가 출력된 것이다.

12.2 모달리스형 대화상자

모달리스(Modalless)형은 대화상자를 열어 놓은 채로 다른 윈도우로 전환할 수 있는 대화상자이다. 일반적으로 모달형에 비해 많이 사용되지는 않지만 필요에 의해 사용되어 진다. 모달리스형은 다른 작업을 하면서 열려 있을 수 있기 때문에 모달형 보다는 프로그래밍할 때 고려해야 할 사항이 더 많이 있다.

리소스는 모달리스형에서 구현한 것을 사용토록 하고 예제를 보도록 하자.

```
typedef struct tagDATA{
    TCHAR str[20];
    int num;
}DATA;

//-----
HWND g_hDlg; // Dialog 의 핸들
#define WM_APPLY WM_USER + 100
//-----
BOOL CALLBACK DigProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam){
```

```

static DATA *pData;

switch(msg)
{
    // Dialog가 처음 나타날때 발생. 각 컨트롤을 초기화 한다.
    case WM_INITDIALOG:
    {
        pData = (DATA*) lParam;

        SetDlgItemText(hDlg, IDC_EDIT1, pData->str);
        SetDlgItemInt(hDlg, IDC_EDIT2, pData->num, 0);
        HWND h = GetDlgItem(hDlg, IDOK);
        SetWindowText(h, TEXT("적용"));
    }
    break;
    case WM_COMMAND:
    switch(LOWORD(wParam))
    {
        case IDOK: //적용
            GetDlgItemText(hDlg, IDC_EDIT1, pData->str, sizeof(pData->str));
            pData->num = GetDlgItemInt(hDlg, IDC_EDIT2, 0, 0);
            SendMessage( GetParent( g_hDlg), WM_APPLY, 0, 0);
            return TRUE;

        case IDCANCEL:
            EndDialog(hDlg, IDCANCEL);
            return TRUE;
    }
}

return FALSE; // 메세지 처리를 안한경우..
}

LRESULT CALLBACK WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam){

    static DATA data= {TEXT("홍길동"), 20 };

    switch( msg )
    {

```

```

case WM_APPLY: // Dialog가 보내는 메시지
{
    TCHAR buf[128];

    wsprintf(buf, TEXT("%s / %d"), data.str, data.num);

    SetWindowText(hwnd, buf);

}

return 0;

case WM_LBUTTONDOWN: {
    if ( g_hDlg == 0)
    {
        // 모달리스 만들기.

        g_hDlg = CreateDialogParam( GetModuleHandle(0), // hinstance
            MAKEINTRESOURCE( IDD_DIALOG1),

            hwnd, // 부모
            DlgProc, // 메시지 함수.
            (LPARAM)&data);

        ShowWindow( g_hDlg, SW_SHOW );
    }

    else

        SetFocus( g_hDlg ); //이미 만들어진 경우 focus이동
    }

return 0;

```

예제 11.2 모달리스 대화상자



적용 버튼 클릭시 대화상자가 종료되지 않고 타이틀바의 정보를 갱신시키는 것을 확인할 수 있다.

12.3 대화상자 기반 윈도우 프로그램

WinMain에서 윈도우 창을 만들지 않고 대화상자로 메인 윈도우의 역할을 수행할 수 있다.

```
BOOL CALLBACK DlgProc( HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam){  
    switch (msg)  
    {  
        case WM_INITDIALOG: {  
            return TRUE;  
        }  
        case WM_COMMAND:  
            switch( LOWORD( wParam ) )  
            {  
                case IDCANCEL: EndDialog(hDlg, IDCANCEL); return 0;  
            }  
            return TRUE;  
        }  
    }  
    return FALSE;  
}  
  
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE , PSTR lpCmdLine, int nShowCmd){  
  
    UINT ret = DialogBox(hInst, // instance  
        MAKEINTRESOURCE( IDD_DIALOG1 ), // 다이얼로그 선택  
        0, // 부모 윈도우  
        DlgProc); // Proc..  
  
    return 0;  
}
```

예제 3.3 대화상자 기반 윈도우 프로그램