콜백 함수와 후킹 기법

(Callback functions and Hooking Techniques)

Win32 API 를 이용하여 어플리케이션을 개발하다 보면, 콜백 함수에 대한 내용을 많이 보게 된다. 그렇다면 이렇게 흔히 보게 되는 콜백 함수란 과연 무엇일까 ?
그리고, 이런 콜백 함수를 이용해서 할 수 있는 것 중에서 후킹 기법을 이용하면 윈도우 운영체제에서 사용할 수 있는 여러가지 마우스 작업이나 키보드 작업 등을 중간에서 조작하여 활용하는 것이 가능하다.
이번 장에서는 콜백 함수와 후킹 기법에 대해 알아보도록 한다.

콜백 함수 (Callback functions)

콜백 함수는 대단히 유용함에도 별로 사용되지 않는 것 중의 하나이다. 이를 활용하면 중복된 코드의 양을 줄일 수 있으며, 읽기가 쉬우면서도 직관적인 모듈을 디자인할 수 있다.
콜백이란 프로시저나 함수를 다른 프로시저나 함수의 파라미터로 넘겨서 그 함수에 특정 이벤트가 발생할 때 호출될 수 있도록 하는 함수를 말한다. 콜백 함수의 실행이 완료되면 제어권은 원래의 프로시저(함수)로 넘어온다.
예를 들어, 객체의 배열이 있고 이 객체들의 특정 메소드를 공통으로 실행시키고 싶다고 하자. 이 경우에는 배열을 루프를 돌리면서 특정 메소드를 호출하는 방법을 사용할 수 있다. 그러면, 동시가 아닌 여러 시점에서 여러 개의 다른 메소드를 실행시키고 싶다고 하자. 이 때에는 루프와 같은 단순한 방법으로는 해결할 수가 없다. 이럴 때 콜백 함수를 활용하면 유용한 해결책을 구할 수 있다. 즉, 객체들에 적용해야 하는 마스터 프로시저를 작성하고 이들 각각이 각 객체를 파라미터로 넘겨주면서 콜백 프로시저를 호출하면 된다.
또한, 콜백 함수를 달리 말하면 어플리케이션에서 구현된 함수의 주소를 DLL 에 포함되어 있는 함수에 보낼 수 있는 것을 말한다. DLL 의 함수는 어플리케이션의 함수에게 다시 정보를 담아서 보낸다.

콜백 함수와 Win32 API

콜백 함수를 설명할 때 빠지지 않고 단골로 설명하는 윈도우 API 함수에는 EnumFonts() 함수가 있다. EnumFonts() 함수는 주어진 장치 컨텍스트에서 사용이 가능한 폰트를 나열하는 함수이다. 각각의 나열되는 폰트는 EnumFonts() 함수가 어플리케이션의 함수로 콜백을 하게 된다. 즉, 각 폰트에 대한 정보를 돌려주는 것이다. 이런 과정이 나열할 폰트가 모두 나열되거나, 콜백 함수가 더 이상의 나열을 중지하고자 하는 의미의 0 을 반환할 때까

지 계속된다.
대부분의 콜백 함수를 파라미터로 사용할 수 있는 윈도우 API 함수는 lpData 파라미터 역시 사용할 수 있도록 허용하고 있다. 보통 이런 lpData 파라미터는 어플리케이션이 정의한 데이터를 나타내는 역할을 한다. lpData 파라미터가 DLL 에 전달될 때에는 보통 LongInt 형으로 전달된다. 그러므로, LongInt 형의 데이터가 아닌 보다 복잡한 형태의 데이터를 넘기고자 할 경우에는 데이터 구조체의 포인터를 LongInt 로 형변환(typecast)한 뒤에 DLL 의 함수를 호출한다. 그리고, DLL 의 함수에 의해서 처리된 후에 콜백 함수로 넘어오는 LongInt 형의 데이터를 다시 포인터로 형변환하면 데이터 구조체에 접근할 수 있게 된다.
앞에서도 잠시 언급했지만, 콜백 함수가 윈도우 API 함수에 의해 호출될 때에는 윈도우 API 함수에 넘겨준 lpData 파라미터가 다시 콜백 함수가 사용할 수 있도록 넘어오게 된다. 이렇게 함으로써 어플리케이션에서 전역 변수를 선언하지 않아도 프로세스 간의 장벽을 넘을 수 있게 된다. 간단한 예를 들어보면 모든 가능한 폰트를 드롭-다운 리스트 박스에 보여주는 대화 상자를 가정해 보자. 이렇게 하려면, WM_INITDIALOG 메시지를 처리할 때 EnumFonts() 함수를 호출해야 하는데, 이때 폰트의 정보를 처리할 콜백 함수의 주소를 담아서 호출해야 한다. 콜백 함수는 아마도 넘어온 각 폰트의 이름을 리스트 박스에 추가하는 역할을 하게 된다. 이때 문제가 하나 있는데, 콜백 함수가 리스트 박스에 대한 핸들을 가지고 있지 않기 때문에, 리스트 박스에 대한 핸들을 전역 변수에 담아서 처리해야 한다. 그런데, 이때 lpData 파라미터를 이용해서 리스트 박스의 핸들을 콜백 함수에서 받게 되면 이러한 문제를 해결할 수 있다.
물론 전역 변수를 쓰는 것도 하나의 해결책이 되겠지만, 언제나 전역 변수가 적은 수로 유지하는 것이 좋은 습관이므로 가능한 lpData 파라미터를 활용하는 것이 좋다.
전역 변수를 사용할 때 또 하나의 문제점은 콜백 함수를 16 비트 DLL 에 구현할 경우에는 데이터를 저장한 전역 변수가 콜백을 이용할 때 데이터가 파괴될 가능성이 있다. 예를 들어, 앞의 폰트 대화 상자를 여러 개의 어플리케이션에서 사용한다고 가정하자. 아마도 이런 경우에는 각각의 어플리케이션에서 폰트 대화 상자를 구현하기 보다는, 폰트 대화 상자를 DLL 에 구현하고 이를 사용하기를 원할 것이다. 그런데, 이때 DLL 에서 리스트 박스의 핸들을 저장하기 위해 전역 변수를 사용한다면, 하나 이상의 어플리케이션이 동시에 대화 상자를 로드할 때 문제가 발생할 수 있다. 각각의 폰트 대화 상자의 복사본(copy)은 각기 다른 리스트 박스의 핸들을 담게 되므로, 전역 변수를 사용해서는 문제가 발생하게 된다. 이럴 때에는 리스트 박스의 핸들을 lpData 파라미터를 이용해서 EnumFonts() 함수에 넘겨주면 된다. 이런 문제는 Win32 에서는 DLL 이 독립적인 기억 공간을 확보하기 때문에 문제가 되지 않는다.

콜백 함수를 이용한 API 활용 예제

콜백 함수를 이용해서 현재 시스템에서 돌아가는 메인 윈도우의 캡션을 보여주는 예제를 가

지고 콜백 함수의 기본적인 사용 방법을 익혀 보도록 하자.
먼저 사용할 윈도우 API 함수인 EnumWindows API 함수의 정의 부분을 살펴 보자.

```
function EnumWindows(lpEnumFunc: TFNWndEnumProc; lParam: LPARAM): BOOL; stdcall;
```

여기서 lpEnumFunc 의 파라미터는 콜백 함수의 주소를 넘기게 되며, lParam 파라미터는 어플리케이션에서 정의한 데이터를 나타낸다. 이때 콜백 함수의 프로시저 형은 윈도우 도움말을 바탕으로 어플리케이션의 type 선언문에서 선언해 주어야 한다. 그러므로, 다음과 같은 형태의 프로시저 형을 어플리케이션의 인터페이스 섹션에 선언한다. 여기서 파라미터의 데이터 형만 맞으면 콜백 함수로 사용하는데에는 전혀 지장이 없다.
사실 이런 프로시저 형을 직접 type 선언문에 선언하지 않아도 사용할 콜백 함수의 파라미터의 데이터 형과 순서, 수만 맞으면 아무런 상관없이 사용할 수 있다.

```
type
  EnumWIndowsProc = function(HWnd: THandle; Param: Pointer): Boolean; stdcall;
```

첫 번째 파라미터는 각 메인 윈도우의 핸들이고, 두 번째 파라미터는 EnumWindows 함수를 호출할 때 넘겨주는 값이다. 사실 파스칼에서 TFNWndEnumProc 형은 제대로 정의된 것이 아니고, 단지 포인터일 뿐이다. 즉, 함수에 적절한 파라미터를 넘겨주고 나서는 이를 포인터로 사용해서 호출하는 대신 그 함수의 주소를 이용한다는 것을 의미한다.
새로운 어플리케이션을 시작하고 폼에 리스트 박스와 버튼을 하나씩 얹어서 폼을 다음 그림과 같이 디자인 한다.

그러면 콜백 함수로 사용할 함수를 다음과 같이 제작한다.

```
function GetCaption(HWnd: THandle; Param: Pointer): Boolean; stdcall;
var
  Text: String;
begin
  SetLength(Text, 100);
  GetWindowText(HWnd, PChar(Text), 100);
  Form1.ListBox1.Items.Add(IntToStr(HWnd) + ':' + Text);
  Result := True;
end;
```

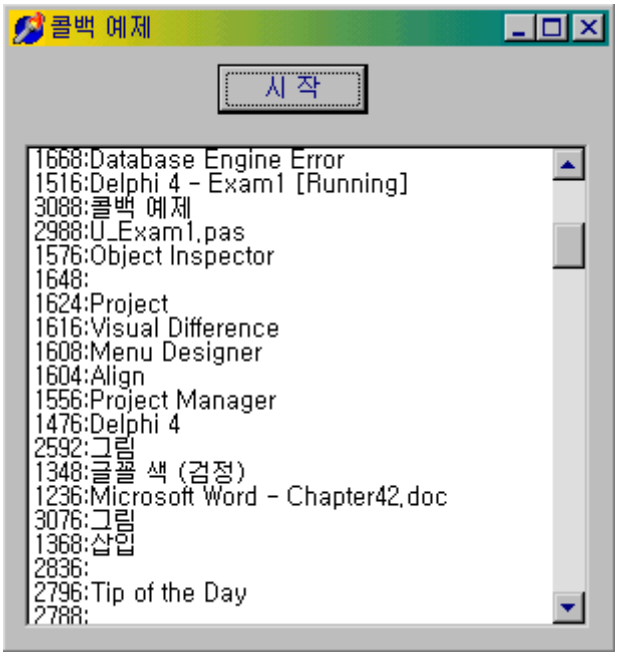
이런 콜백 함수는 파라미터만 맞으면 사용이 가능하다. 이 콜백 함수의 역할은 윈도우의 캡션을 문자열로 읽어서 리스트 박스에 추가한다.
그러면, 이 콜백 함수를 이용하기 위한 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성하도록 한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  EnumWin: EnumWindowsProc;
begin
  ListBox1.Items.Clear;
  EnumWin := GetCaption;
  EnumWindows(@EnumWin, 0);
end;
```

앞의 코드는 단지 이렇게 사용할 수도 있다는 것을 보여주기 위한 것으로, 변수로 선언한 EnumWin 의 선언과 이 변수에 GetCaption 함수의 주소를 저장해서 EnumWindows API 함수를 호출하는 과정을 다음과 같이 간단히 처리하는 것과 내용은 같은 것이다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Clear;
  EnumWindows(@GetCaption, 0);
end;
```

이제 이 프로그램을 실행하고 버튼을 클릭하면 다음과 같은 실행 화면을 볼 수 있을 것이다.



콜백 함수의 실용적 활용

콜백 함수를 이용하는 방법을 익히는 좋은 예제로 디렉토리를 뒤지면서 파일을 검색하고, 파일이 발견될 때마다 콜백 함수를 호출하여 이를 표시하는 등의 것을 예로 들 수 있겠다. 콜백 함수를 사용하기 위해서는 먼저 프로시저 형을 유닛의 type 섹션에 다음과 같이 선언해야 한다.

```
TFileSearchCallback = procedure(FileName: string) of Object;
```

이와 같이 콜백 함수를 사용하기 위해서는 객체의 메소드로서의 프로시저 형을 선언해서 사용한다. 여기서 파라미터 형이 맞을 경우 얼마든지 콜백으로 사용이 가능하다.
그리고 나서는 실제로 콜백 함수를 호출할 프로시저를 다음과 같이 선언한다.

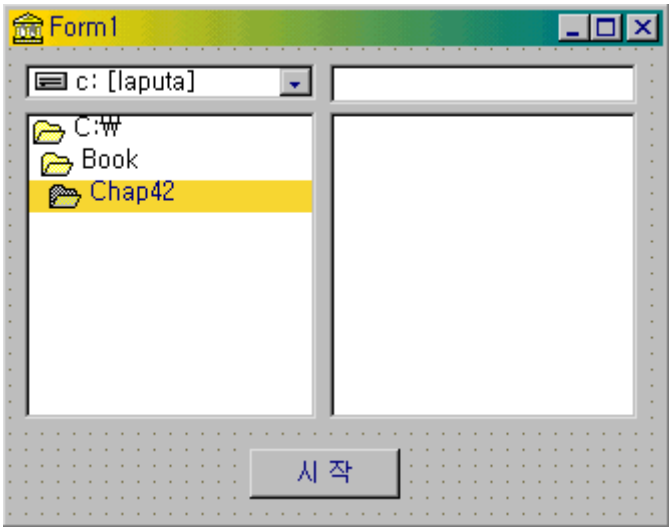
```
procedure SearchDirectory(Dir, Condition: string; cb: TFileSearchCallback);
```

이 프로시저의 형태를 살펴보면 검색을 할 디렉토리와 검색할 문자열을 나타내는 파라미터인 Dir, Condition 은 다른 프로시저의 형태와 별로 다를 것이 없지만, cb 파라미터에서 앞에서 선언한 TFileSearchCallback 프로시저 형을 사용하는 것이 중요한 부분이다. 이 파라미터의 의미는 여기에 콜백 함수를 대입하여 콜백을 이용할 수 있게 된다는 것이다.

이제 실제로 디렉토리를 검색하는 예제 어플리케이션을 작성해보도록 하자.
새로운 어플리케이션을 시작하자.
앞에서 설명한 대로 콜백 프로시저 형을 type 섹션에 선언하고, SearchDirectory 프로시저를 private 섹션에 선언한다.
그리고, 다음과 같이 구현한다.

```
procedure TForm1.SearchDirectory(Dir, Condition: string; cb: TFileSearchCallback);
var
  SearchRec: TSearchRec;
  Value: LongInt;
begin
  Value := FindFirst(Dir + '₩' + Condition, faAnyFile, SearchRec);
  while Value = 0 do
  begin
    cb(Dir + '₩'+ SearchRec.Name);
    Value := FindNext(SearchRec);
  end;
end;
```

기본적으로 이 프로시저는 Dir 파라미터로 넘어온 디렉토리를 검색해서 파일을 찾게 되면 콜백 함수를 호출하는데, 콜백 함수에게 현재 파일의 경로를 포함한 파일 이름을 파라미터로 넘겨주게 된다. 그러면 이제 실제로 콜백을 수행할 콜백 함수를 작성하고 예제를 눈에 보일 수 있도록 만들어 보자.
폼에 TButton, TListBox, TEdit, TDriveComboBox, TDirectoryListBox 컴포넌트를 하나씩 올려 놓도록 하자. 그리고 Button1 의 Caption 프로퍼티를 '시 작' 으로, Edit1 의 Text 프로퍼티를 ''로 다음과 같이 설정한다. 여기서 TEdit 컴포넌트에 검색할 문자열을 입력하고 버튼을 클릭하면 해당되는 파일을 찾아서 파일의 이름일 TListBox 컴포넌트에 추가하도록 하는 것이다. 그리고, TDriveComboBox 와 TDirectoryListBox 를 연결하기 위해 DriveComboBox1 의 DirList 프로퍼티를 DirectoryListBox1 으로 설정한다.



그리고 나서, 다음과 같이 콜백 프로시저를 private 섹션에 선언한다.

```
procedure FileSearchCallback(FileName: string);
```

이 콜백 함수는 디렉토리에서 파일을 찾게 되면, 파일 이름을 TListBox 컴포넌트에 추가하여 보여주는 간단한 함수로 다음과 같이 작성하도록 한다.

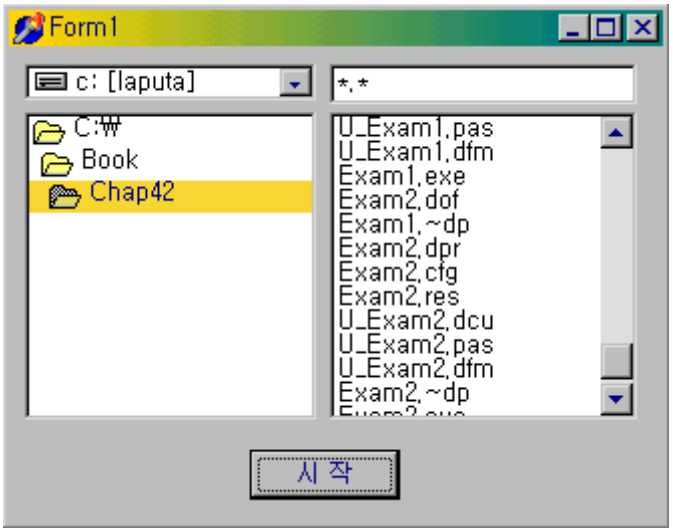
```
procedure TForm1.FileSearchCallback(FileName: string);
begin
  Listbox1.Items.Add(ExtractFileName(FileName))
end;
```

이제 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Clear;
  SearchDirectory(DirectoryListBox1.Directory, Edit1.Text, FileSearchCallback);
end;
```

여기서 앞에서 예를 든 다른 프로시저와는 달리 콜백 함수의 주소를 @FileSearchCallback 으로 넘겨주지 않고 직접 FileSearchCallback 으로 넘겨 주는 것에 주목한다. 이것은 델파이가 사용하는 오브젝트 파스칼의 특성에 기인한 것으로, C/C++의 경우에는 기본적으로 객체의 주소를 포인터로 선언하고, 이를 이용해서 프로그래밍을 해야 하므로 명시적으로 함

수의 주소를 가리키는 연산자를 사용해야 하지만, 오브젝트 파스칼은 모든 기본 객체 들이 참조로 사용되기 때문에 주소를 나타내는 '@'를 사용하지 않아도 잘 작동한다.
이제 이 어플리케이션을 컴파일해서 실행하고, 적당한 디렉토리와 검색 문자열을 입력한 후 버튼을 클릭하면 다음과 같은 실행 화면을 얻을 수 있을 것이다.



후크 함수 (Hook functions)

후크 함수는 윈도우 메시지 시스템에 삽입되어 메시지에 대한 처리가 일어나기 전에 메시지 스트림에 접근해서 이를 처리할 수 있는 콜백 함수이다. 흔히, 메시징 시스템에 다른 후크가 있는 경우가 있는데, 이 때에는 후크 연쇄(Hook chain)에서의 다음의 후크를 호출해서 이들이 모두 동작할 수 있도록 해야 한다.
후크를 이용해서 시스템에서 메시지 트래픽을 모니터하는 서브 루틴을 제작한다든가, 목적 윈도우 프로시저에 도달하기 전에 일부 메시지를 처리하는 등의 일을 할 수 있다.
후크는 각각의 메시지를 처리할 때 거쳐야 하는 과정을 증가시키기 때문에 시스템의 속도를 다소 저하시키는 경향이 있다. 그러므로, 후크는 꼭 필요할 때 설치했다가 가능한 빨리 제거해 주어야 한다.
후크 함수를 설치할 때에는 SetWindowsHookEx() API 함수를 이용한다. 이 함수를 호출하면 설치된 후크 함수에 대한 32 비트 핸들을 얻게 되며, 이 핸들을 이용해서 후크를 제거하거나 다음의 후크를 호출할 때 이용한다.
후크를 제거할 때에는 UnHookWindowsHookEx() API 함수를, 다음 후크를 호출할 때에는 CallNextHookEx() 함수를 호출한다.
이 때 시스템 전반에 걸친 호크는 후크 함수가 DLL 에 위치해야 하지만, 어플리케이션에 지정된 후크 함수는 어플리케이션이나 DLL 에 모두 위치할 수 있다.

후크 연쇄 (Hook Chains)

윈도우는 다른 종류의 많은 후크의 종류를 가지고 있다. 각각의 후크의 형은 윈도우 메시지 처리 메커니즘의 다른 면에 접근한다. 예를 들어, 마우스 메시지에 대한 메시지 트래픽을 모니터할 때에는 WM_MOUSE 후크를 이용한다.
윈도우는 각각의 후크 종류에 따라 분리된 후크 연쇄(hook chain)를 가지고 있다. 후크 연쇄는 후크 프로시저로 불리는 어플리케이션이 정의한 콜백 함수에 대한 포인터의 리스트이다. 특정한 종류의 후크와 연관된 메시지가 발생하면 윈도우는 메시지를 후크 연쇄에서 참조하는 각각의 후크 프로시저에 넘기게 된다. 후크 프로시저의 동작은 연관된 후크의 종류에 따라 달라진다. 일부의 후크 프로시저는 메시지를 모니터할 뿐이지만, 일부의 경우에는 메시지를 변경하거나 전달을 중지시킬 수 있다.

후크 프로시저 설치와 해제할 때 주의점

후크 프로시저를 설치할 때에는 SetWindowsHookEx 함수를 이용한다. 그리고, 이 함수를 호출할 때에는 호출하는 후크의 종류와 프로시저가 모든 쓰레드와 연관되어야 하는지, 아니면 특정 쓰레드와 연관되는지 여부 그리고 프로시저 엔트리 포인터 등을 지정하게 된다.
후크 프로시저를 설치하는 어플리케이션은 반드시 전역 후크 프로시저를 DLL 에 분리해야 한다. 대신 후크 프로시저를 설치하기 전에 DLL 모듈의 핸들을 가지고 있어야 한다. 즉, LoadLibrary 함수를 이용해서 DLL 모듈의 핸들을 얻은 후, 이렇게 핸들을 얻으면 GetProcAddress 함수를 이용해서 후크 프로시저의 주소를 얻을 수 있게 된다. 마지막으로 SetWindowsHookEx 함수를 이용해서 후크 프로시저의 주소를 적절한 후크 연쇄에 설치한다. SetWindowsHookEx 함수에는 모듈의 핸들과 후크 프로시저의 엔트리 포인트, 그리고 쓰레드에 대한 identifier 을 넘겨 주는데, 보통은 이 값으로 0 을 넘겨서 시스템의 모든 쓰레드에 반응할 수 있도록 한다.
특정 쓰레드에 대한 후크 프로시저를 해제(후크 연쇄에서 프로시저의 주소를 삭제하는 것)할 때에는 UnhookWindowsHookEx 함수에 후크 프로시저의 핸들을 지정해서 호출하게 된다. 전역 후크 프로시저를 해제할 때에도 UnhookWindowsHookEx 함수를 호출하기는 마찬가지이다. 그런데, 이 함수는 후크 프로시저를 담고 있는 DLL 을 해제하지 않는다. 이는 전역 후크 프로시저는 모든 윈도우 기반의 어플리케이션에서 호출되기 때문에, 이들이 암시적으로 LoadLibrary 함수를 호출하기 때문으로, 이들이 모두 FreeLibrary 를 호출하게 할 수 있는 방법이 없다. 그러므로, 결국에는 모든 어플리케이션을 중지시키거나 또는 이들이 모두 FreeLibrary 를 호출하게 해야 한다.
이를 해결하기 위해서 전역 후크 프로시저를 설치할 때에는 설치 함수를 DLL 에 후크 프로시저와 함께 제공한다. 이렇게 하면 설치 어플리케이션이 DLL 모듈의 핸들을 가지고 있을 필요가 없다. 각 어플리케이션은 반드시 DLL 과 연결해야만 후크를 설치할 수 있게 되고,

설치 함수는 SetWindowsHookEx 함수의 호출을 통해 DLL 의 모듈 핸들을 제공한다. 어플리케이션은 종료될 때 이 DLL 의 핸들을 통해서 후크를 해제하게 된다.

윈도우 후크의 종류

윈도우의 후크에는 다음과 같은 종류 들이 있다.

- WH_CALLWNDPROC:

SendMessage() 함수가 호출될 때마다 호출되는 윈도우 프로시저 후크

- WH_CBT:

CBT(computer based training) 후크는 윈도우를 생성, 파괴, 최대화, 이동, 크기 변화 등의 이벤트가 있거나, 마우스나 키보드 이벤트를 제거하기 전에 또는 입력 포커스의 변경이나 시스템 메시지 큐의 동기화기 있기 전에 호출되는 후크이다.

- WH_GETMESSAGE:

GetMessage() 함수가 어플리케이션 큐에서 메시지를 가져올 때마다 호출되는 후크이다.

- WH_HARDWARE:

어플리케이션이 GetMessage(), PeekMessage() 함수를 호출하고, 처리할 하드웨어 이벤트(마우스와 키보드 이벤트를 제외한)가 있을 때 호출되는 후크이다.

- WH_JOURNALRECORD:

시스템 메시지 큐에서 메시지를 삭제할 때 호출되는 후크이다.

- WH_JOURNALPLAYBACK:

마우스나 키보드 메시지를 시스템 메시지 큐에 삽입할 때 이용되는 후크이다.

- WH_KEYBOARD:

어플리케이션이 GetMessage(), PeekMessage() 함수를 호출하고, WM_KEYUP 또는 WM_KEYDOWN 키보드 메시지를 처리해야 할 때 호출되는 후크이다.

- WH_MOUSE:

어플리케이션이 GetMessage(), PeekMessage() 함수를 호출하고, 마우스 메시지를 처리해야 할 때 호출되는 후크이다.

- WH_MSGFILTER:

대화 상자나 메시지 상자 또는 메뉴가 메시지를 불러올 때, 메시지가 처리되기 전에 호출되는 후크이다.

- WH_SHELL:

시스템이 만들어 놓은 notification 메시지가 있을 때 호출되는 후크이다.

- WH_SYSMSGFILTER:

대화 상자나 메시지 상자 또는 메뉴가 메시지를 불러올 때, 메시지가 처리되기 전에 호출되는 시스템 전체에 대한 후크이다.

기본적인 후크 예제

후크를 어떻게 설치하고 해제하는지를 코드를 통해서 알아보자.
시스템 후크로 사용할 함수를 DLL 파일에 담고, 이 DLL 을 사용하여 시스템 후크를 설치, 제거하는 어플리케이션을 하나 제작한다.
DLL 파일은 ExamLib3.DLL, 어플리케이션은 Exam3.EXE 로 한다.
먼저, DLL 파일을 만들기 위해 새로운 DLL 프로젝트를 하나 시작한다.
그리고 다음의 코드를 입력하자.

```
library ExamLib3;

Uses
  SysUtils, Windows, Dialogs;

function KeyboardProc(Code, WParam, LParam: Integer): LRESULT; stdcall;
var
  Val: Integer;
begin
  Val := -1;
  if (Code >= 0) then Val := CallNextHookEx(0, Code, WParam, LParam);
  Result := Val;
end;

exports
  KeyboardProc index 1;
```

```
begin
end.
```

이 DLL 파일은 실제로 아무 동작을 하지 않고, 다음의 후크를 호출하는 역할 만을 한다.
그러면, DLL 파일을 이용해서 후크를 설치, 해제하는 어플리케이션을 제작하자.
폼을 생성할 때 후크를 설치하고, 폼을 파괴할 때 후크를 해제하도록 한다.
후크의 핸들을 public 멤버인 HandleHook 에 저장하고, DLL 의 핸들을 private 멤버인 hInst 에 저장하도록 하자.
어플리케이션의 소스는 다음과 같다.

```
unit U_Exam3;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    hDLL: HInst;
  public
    HandleHook: HHook;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
  hDLL := LoadLibrary('ExamLib3');
  if hDLL = 0 then
    ShowMessage('DLL 을 로드하지 못했습니다.');
  HandleHook := SetWindowsHookEx(WH_KEYBOARD,
    GetProcAddress(hDLL,PChar('KeyboardProc')), hDLL, 0);
  if HandleHook = 0 then
    ShowMessage('후크를 설치하지 못했습니다.');
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  if not UnhookWindowsHookEx(HandleHook) then
    ShowMessage('후크를 해제하지 못했습니다.');
end;

end.
```

즉, 후크를 설치할 때 우선 DLL 파일을 LoadLibrary 함수를 이용해서 메모리에 적재한 후 DLL 파일에 저장되어 있는 함수의 주소를 후크 프로시저로 설정한다. 이때 윈도우 API 함수인 SetWindowsHookEx 를 사용하는데, 후크를 설치하지 못하면 0 이 반환된다.
후크를 제거할 때에는 설치된 후크의 핸들을 이용해서 UnhookWindowsHookEx 함수를 호출하면 된다. 이 함수의 경우에는 후크를 제거하지 못하면 False 를 반환한다.

후크를 이용한 매크로 작성기 제작

이번에는 실제 사용 가능하도록 후크를 응용해보자.
아래아 한글이나 워드, 그 밖의 많은 에디터에 보면 대부분의 경우 키보드와 마우스 동작을 기록했다가 저장하고, 이를 다시 재생할 수 있는 매크로 기능을 제공한다. 그러면, 윈도우의 후크 프로시저를 이용해서 키보드와 마우스 입력을 기록했다가 이를 다시 재생할 수 있는 DLL 과 이를 이용한 간단한 예제 어플리케이션을 제작해 보자.
이 예제는 볼랜드에서 제공하는 기술 백서(Technical white paper)에 공개되었던 어플리케이션에 기초한 것임을 미리 밝혀 둔다.
이렇게 메시지를 기록하고, 재생하기 위해서는 시스템 전체에 적용되는(system wide) WH_JOURNALRECORD, WH_JOURNALPALYBACK 윈도우 후크를 이용한 후크 함수를 사용하게 된다. 이때 각각의 윈도우 후크에 대한 후크 프로시저를 JournalRecord()와

JournalPlayback()이라고 하자.
JournalRecord 와 JournalPlayback 후크의 콜백 함수를 설치하기 위해 윈도우 API 함수인 SetWindowsHookEx()에 후크 콜백 함수의 주소를 넘겨 주어야 한다.
이때 JournalRecord 와 JournalPlayback 콜백 함수는 시스템 후크 (system wide hook)이므므로, 반드시 DLL 에 위치해야 하며 후크 콜백 함수의 주소는 인스턴스화한 주소를 넘겨주지 않아도 된다. 이 함수를 호출하면 후크 콜백 함수에 대한 32 비트 핸들을 반환하게 되는데, 이 핸들을 이용해서 다른 후크 함수가 이미 후크 연쇄(hook chain)에 존재하는지 확인하고, 기록이 끝나면 후크를 연쇄에서 제거 하는 등의 일을 할 수 있다.
일단 JournalRecord 콜백 함수의 주소를 가지고 SetWindowsHookEx()를 호출하면, 윈도우는 즉시 메시지의 기록을 시작한다. 이때 JournalRecord 콜백 함수가 호출되는 조건은 다음과 같다.

1. 기록할 키보드나 마우스 이벤트가 존재할 때 (HC_ACTION)
2. 시스템이 모달 상태에 들어가거나 (HC_SYSMODALON), 모달 상태에서 빠져 나올 경우 (HC_SYSMODALOFF).
3. 시스템이 후크 연쇄에서 다음 후크를 호출하기 원할 때

키보드나 마우스 이벤트가 있을 경우에는 HC_ACTION 후크와 같은 경우인데, 이 이벤트를 기록할 수 있다. 이때 EVENTMSG 형의 구조체에 대한 포인터가 JournalRecord 콜백 함수의 lParam 파라미터에 담겨서 넘어오게 된다. 이벤트가 발생한 시스템 시간은 EVENTMSG 구조체의 time 파라미터에 담겨 있다. 재생을 하기 위해서는 EVENTMSG 구조체를 복사하고, 그 복사본의 time 을 적절하게 조절하면 된다. 즉, 메시지를 기록할 때 시스템 시간을 얻은 후 각 메시지의 시간에서 처음에 얻은 시스템 시간을 빼면, 기록이 시작된 후 얼마의 간격으로 메시지 들이 발생했는지 알아낼 수 있다. 이 값들을 재생이 시작되는 시간에 더하면 기록할 때와 같은 간격으로 메시지들을 발생시킬 수 있다.
시스템이 모달 상태에 들어 갔을 경우에는 HC_SYSMODALON 후크와 같은 경우로 이 때에는 잠시 기록을 멈추고, 후크 연쇄의 다음 메시지 후크를 호출하게 하여야 하며, 모달 상태에서 빠져 나올 경우의 후크인 HC_SYSMODALOFF 의 경우에 다시 기록을 재개하도록 하면 된다.
만약 코드 값이 0 보다 작을 경우에는 시스템이 후크 연쇄의 다음 메시지 후크를 호출할 것을 요구하게 되며, 이 때에는 이를 따라야 한다. 그리고, 기록이 끝났을 경우에는 윈도우 API 함수인 UnHookWindowsHookEx()을 SetWindowsHookEx() 함수를 호출했을 때 얻은 32 비트 핸들을 파라미터로 해서 호출하면 후크 콜백 함수를 후크 연쇄에서 제거하게 된다.
기록된 메시지를 재생하고자 할 때에는 또 다시 SetWindowsHookEx() API 함수를 JournalPlayback 콜백 함수의 주소를 파라미터로 해서 호출하면 윈도우는 즉시 재생을 시작하게 된다. 재생을 하는 동안에는 정상적인 마우스와 키보드 입력이 시스템에 의해 중지

된다.
JournalPlayback 콜백 함수는 다음의 조건에 부합될 때 각각 다음과 같은 역할을 하여야 한다.

1. HC_SKIP:

다음 메시지를 불러온다. 재생할 메시지가 더 이상 없으면 UnHookWindowsHookEx() 함수를 호출해서, 후크를 제거한다.

2. HC_GETNEXT:

현재의 메시지를 재생한다.

3. HC_SYSMODALON:

시스템이 모달 상태에 들어간 경우로, 메시지 재생 중에 모달 상태로 들어간다는 것은 시스템에 어떤 문제가 생겼음을 의미한다. 그러므로, 후크 연쇄의 다음 후크를 호출하여 이를 처리할 수 있도록 해준다.

4. HC_SYSMODALOFF:

시스템이 모달 상태에서 빠져 나온 경우로, 윈도우는 JournalPlayback 콜백 프로시저를 제거하고, 후크 연쇄의 다음 후크를 호출한다.

5. 코드가 0 보다 작은 경우:

시스템이 더 이상의 처리를 원하지 않고, 다음 후크를 호출하라고 요구하는 것이다.

이런 시스템 후크를 구현하기 위해서는 SetWindowsHookEx() 함수를 호출하기 전에, 첫번째 메시지를 받아서 시스템 시간을 얻어야만 각각의 기록된 메시지 들이 재생시에 동기화되어 동작하게 할 수 있다.
이때 윈도우는 같은 메시지를 한 번 이상 반복할 것인지 묻는다. 윈도우가 현재 메시지를 재생할 것인지를 처음 물어올 때 JournalPlayback 콜백 함수는 현재 시간과 메시지가 작동하도록 되어 있는 시간의 차이 부분을 반환한다. 만약 이 값이 음수라면 0 을 반환하도록 해야 한다. 또한, 같은 메시지가 한번 이상 요구될 경우에도 0 을 반환한다.

- 후크 DLL 의 동작 방식

실제 핵심적인 역할을 담당하는 DLL 파일에는 다음과 같은 세 가지 함수와 윈도우가 사용하게 될 두 가지 후크 함수가 포함되어 있다.

1. StartRecording()
2. StopRecording()
3. Playback()
4. JournalRecordProc()
5. JournalPlaybackProc()

StartRecord() 함수는 JournalRecordProc() 후크를 설치하고 이벤트를 기록하기 시작하는 역할을 한다. StartRecord()를 호출하다가 에러가 발생하거나, 이미 매크로를 기록하거나 재생하고 있을 때에는 더 이상의 처리를 하지 않고, 0 을 반환한다. 매크로의 기록이 이루어지는 중간에는 키보드와 마우스 이벤트를 배열에 저장한다. 참고로 지나치게 많은 수의 메모리 소모를 줄이기 위해서 최대치를 설정한다.
StopRecording()가 호출되면, 기록한 이벤트 들을 디스크에 저장한다. 그리고 나서, JournalRecordProc() 후크를 제거한다. 이때 아무것도 기록된 것이 없을 경우에는 -1 을 후크를 제거할 때 에러가 발생하면 -2 를 에러 코드로 반환한다. 에러가 없을 때에는 기록된 메시지의 수를 반환한다.
Playback() 함수는 기록된 매크로를 재생할 때 호출한다. 에러가 발생하거나, 재생할 것이 없으면 더 이상의 처리를 하지 않고 0 을 반환한다. 재생이 끝나면, 매크로를 재생하도록 호출한 어플리케이션을 콜백 한다. 그러므로, 이 함수를 호출하는 어플리케이션은 콜백 함수를 작성하여 재생이 끝났을 때의 처리를 해줄 수 있다.
데이터의 공유를 위해서 몇 개의 전역 변수를 선언했다. 물론 이러한 데이터 공유를 위해서는 메모리 맵 파일 등의 보다 세련된 방법을 선택할 수도 있겠으나, 여기에 대해서는 다음에 다루도록 한다. 일단 EventMsg 구조체 배열에 대한 포인터인 PMsgBuff 라는 전역 포인터를 정의하고, 이를 이용해서 메모리에 이벤트를 기록하고 재생한다. 일단 DLL 이 시작되면 이 포인터를 nil 로 초기화하고, 매크로를 기록하거나 재생할 때에만 이 포인터가 실제로 메모리 블록을 가리키도록 한다. 다른 경우에는 언제나 nil 값을 가지도록 설정함으로써 현재 매크로가 기록되거나, 재생되는지 여부를 확인할 수 있다.
그 밖에 사용하는 전역 변수에는 다음과 같은 내용들을 담게 된다.

1. TheHook: 후크 프로시저의 32 비트 핸들
2. StartTime: 매크로의 기록이나 재생이 시작되는 시간
3. MsgCount: 기록된 메시지의 총 수

4. CurrentMsg: 현재 재생되고 있는 메시지
5. ReportDelayTime: 지체된 시간을 리포트 해야 하는지 여부
6. SysModalOn: 시스템에 현재 모달 상태에 있는지 여부
7. cbPlaybackFinishedProc: 어플리케이션 콜백 함수의 인스턴스 주소
8. cbAppData: Playback() 함수에게 넘겨주는 어플리케이션 데이터의 파라미터

- 후크 DLL 의 구현

그러면, 실제로 DLL 을 제작해 보자.
먼저 DLL 의 프로젝트의 이름을 Hook2Lib.dpr 로 저장한다.
그리고, 사용할 데이터 형과 전역 변수를 다음과 같이 선언한다.

```
library Exam4Lib;

uses
 Windows;

type
   TwMsg = LongInt;
   TwParam = LongInt;
   TlParam = LongInt;

const
   MAXMSG = 6500;

type
   PEventMsg = ^TEventMsg;
   TMsgBuff = Array[0..MAXMSG] of TEventMsg;
   TcbPlaybackFinishedProc = procedure(AppData: Longint); stdcall;

var
   PMsgBuff: ^TMsgBuff;
   TheHook: HHook;
   StartTime: Longint;
   MsgCount: Longint;
   CurrentMsg: Longint;
```

```
  ReportDelayTime: Bool;
  SysModalOn: Bool;
  cbPlaybackFinishedProc: TcbPlaybackFinishedProc;
  cbAppData: Longint;
```

상수로 선언한 MAXMSG 는 메시지를 기록할 때 과도한 메모리 사용을 막기 위해서 한도를 정한 것으로 조절이 가능하다.
TcbPlaybackFinishedPrc 프로시저 형은 DLL 의 Playback() 함수를 호출하는 어플리케이션에 대한 콜백을 지원한다. 어플리케이션은 이 프로시저 형에 맞는 콜백 함수를 작성하고, DLL 의 Playback() 함수에 콜백 함수의 주소를 넘겨 주면 Playback() 함수의 종료와 함께 이 콜백 함수를 호출하게 된다.
나머지 전역 변수에 대해서는 앞에서도 간략히 언급 했으므로 자세한 설명은 생략 하겠다.
먼저 매크로를 기록할 후크 프로시저인 JournalRecordProc 함수의 구현 부분을 살펴 보자.

```
function JournalRecordProc(Code: Integer; wParam: TwParam; lParam: TlParam):
  Longint; stdcall;
begin
  Result := 0;
  case Code of
    HC_ACTION:
    begin
      if SysModalOn then Exit;
      if MsgCount > MAXMSG then Exit;
      PMsgBuff^[MsgCount] := PEventMsg(lParam)^;
      Dec(PMsgBuff^[MsgCount].Time, StartTime);
      Inc(MsgCount);
      Exit;
    end;
    HC_SYSMODALON:
    begin
      SysModalOn := True;
      CallNextHookEx(TheHook, Code, wParam, lParam);
      Exit;
    end;
    HC_SYSMODALOFF:
    begin
```

```
        SysModalOn := False;
        CallNextHookEx(TheHook, Code, wParam, lParam);
        Exit;
      end;
    end;
    if Code < 0 then
      Result := CallNextHookEx(TheHook, Code, wParam, lParam);
end;
```

일단 이 함수의 리턴 값을 0 으로 설정하고, Code 파라미터의 값에 따른 처리를 해 준다.
이 코드의 값이 HC_ACTION 인 경우에는 기록할 키보드나 마우스 이벤트가 있는 것이므로 lParam 의 메시지를 버퍼에 저장한다. 이때 주의할 것은 메시지의 시간을 기록하는 것인데, 최초에 StartRecord() 함수가 호출되면 StartTime 전역 변수에 기록 시작 시각이 기록되므로, 이벤트가 발생한 시각에서 StartTime 변수의 값을 빼면 기록 시작 시각에서부터 얼마 뒤에 일어난 이벤트인지 알 수 있다.
시스템이 모달 상태에 들어가거나 빠져 나오는 경우에는 SysModalOn 전역 변수의 값을 설정하고, CallNextHookEx 함수를 호출하여 다음 후크를 실행한다.
또한, 처리할 후크 코드가 아닌 경우에도 다음 후크를 실행한다.
그러면, 이러한 매크로 기록 후크 함수를 설치하고 기록을 시작하게 하는 StartRecording 함수를 구현 하도록 하자.
이 함수의 구현 부분은 다음과 같다.

```
function StartRecording: Integer; stdcall;
begin
  Result := 0;
  if pMsgBuff <> nil then Exit;
  GetMem(PMsgBuff, Sizeof(TMsgBuff));
  if PMsgBuff = nil then Exit;
  SysModalOn := False;
  MsgCount := 0;
  StartTime := GetTickCount;
  TheHook := SetWindowsHookEx(WH_JOURNALRECORD, JournalRecordProc,
    hInstance, 0);
  if TheHook <> 0 then
  begin
    Result := 1;
```

```
    Exit;
  end
  else
  begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
  end;
end;
```

먼저 기본적인 결과 값을 0 으로 설정하고, 메시지 버퍼를 검사해서 버퍼에 메모리가 할당되어 있으면 이는 메시지가 기록되어 있거나, 현재 기록 중임을 의미하므로 실행을 중지한다. 이상이 없으면 버퍼에 대한 메모리를 할당하고, 초기 전역 변수 값을 설정한다.
특히 StartTime 전역 변수의 값을 설정할 때에는 GetTickCount API 함수를 이용한다. 그리고, WH_JOURNALRECORD 후크에 대한 후크 함수를 설치한다. 이때 설치가 성공적이면 0 이 아닌 값이 리턴되므로 결과 값으로 1 을 반환하고, 0 이 리턴되면 버퍼에 대한 메모리를 해제한다.
매크로의 기록을 중지하는 StopRecording() 함수는 구현이 다소 복잡하다.
단순히 후크를 해제하는 것만이 아니라 버퍼에 기록된 메시지를 디스크에 저장하는 역할을 해주어야 한다. 그렇기 때문에 파라미터로 기록할 파일 이름을 PChar 형으로 넘겨 받는다. 실행 결과에 따라서 반환되는 결과 값이 다양한데, 성공적으로 메시지가 기록되고 디스크에 저장된 경우에는 저장된 메시지의 수를, 저장할 메시지가 없을 때에는 -1, 후크 함수를 제거하는데 실패한 경우에는 -2 를 반환하며 I/O 에러가 발생한 경우에는 0 을 반환한다.
구현 부분은 다음과 같다.

```
function StopRecording(lpFileName: PChar): LongInt; stdcall;
var
  TheFile: File;
begin
  if PMsgBuff = nil then
  begin
    Result := -1;
    Exit;
  end;
  if UnHookWindowsHookEx(TheHook) = False then
  begin
    Result := -2;
```

```
    Exit;
  end;
  TheHook := 0;
  if MsgCount > 0 then
  begin
    Assign(TheFile, lpFileName);
    {$I-}
    Rewrite(TheFile, Sizeof(TEventMsg));
    {$I+}
    if IOResult <> 0 then
    begin
      FreeMem(PMsgBuff, Sizeof(TMsgBuff));
      PMsgBuff := nil;
      Result := 0;
      Exit;
    end;
    {$I-}
    Blockwrite(TheFile, PMsgBuff^, MsgCount);
    {$I+}
    if IOResult <> 0 then
    begin
      FreeMem(PMsgBuff, Sizeof(TMsgBuff));
      PMsgBuff := nil;
      Result := 0;
      {$I-}
      Close(TheFile);
      {$I+}
      if IOResult <> 0 then Exit;
      Exit;
    end;
    {$I-}
    Close(TheFile);
    {$I+}
    if IOResult <> 0 then
    begin
      FreeMem(PMsgBuff, Sizeof(TMsgBuff));
```

```
        PMsgBuff := nil;
        Result := 0;
        Exit;
      end;
  end;
  FreeMem(PMsgBuff, Sizeof(TMsgBuff));
  PMsgBuff := nil;
  Result := MsgCount;
end;
```

소스가 다소 길지만 내용은 어렵지 않다. 그리고, 중복되는 부분이 있으므로 이를 단축하면 좋을 것이다. 특별히 설명할 만한 부분은 없지만 $I 컴파일러 지시자(compiler directive)를 통해서 I/O 에러를 처리하는 부분에 대해서 잠시 알아보자.
디폴트로 표준 I/O 프로시저와 함수를 호출하면 자동으로 에러를 검사하게 되어 있다. 만약 에러가 발생하면 예외를 발생시키거나 예외 처리가 안될 경우 프로그램은 실행을 중단한다. 이런 자동 검사를 여부를 결정하는 컴파일러 지시자가 $I 이다. {$I-}에 의해 I/O 에러 자동 검사가 꺼지게 되는데, 이때에는 I/O 에러가 발생해도 예외가 발생하지 않는다. 그러므로, I/O 작업을 한 뒤에는 IOResult 함수를 호출해서 에러 여부를 알아보아야 한다. 에러가 발생하지 않은 경우에는 0 을 반환한다.
앞의 StopRecording() 함수 역시 이런 방법을 이용해서 I/O 에러를 처리한다.
이렇게 해서 매크로를 기록하는 부분에 대한 구현이 모두 끝났다.
이번에는 매크로를 재생하는 부분을 구현하도록 하자.
먼저 후크 함수로 사용될 JournalPlayback() 함수를 다음과 같이 구현한다.
기본적인 처리 방침은 Code 값에 따라서, HC_SKIP 인 경우에는 다음 메시지를 처리하되, 더 이상의 메시지가 없으면 후크 함수를 제거한다. HC_GENNEXT 는 현재 메시지를 처리한다. 그 밖의 자세한 내용은 앞에서 설명했으므로 생략하도록 하겠다.
눈 여겨 보아야 할 부분은 code 값이 HC_SKIP 인 경우 후크를 제거할 때와 HC_SYSMODALOFF 인 경우 메시지 재생을 마치는 경우로 이때에는 어플리케이션에서 넘겨준 콜백 함수를 cbPlaybackFinishedProc(cbAppData)와 같은 형태로 호출한다. 이 콜백 함수는 재생이 끝났음을 어플리케이션에 알리면 동작하게 되는 함수이다.

```
function JournalPlaybackProc(Code: Integer; wParam: TwParam; lParam: TlParam):
  Longint; stdcall;
var
  TimeToFire: Longint;
begin
```

```
Result := 0;
case Code of
  HC_SKIP:
  begin
    Inc(CurrentMsg);
    ReportDelayTime := True;
    if CurrentMsg >= (MsgCount-1) then
      if TheHook <> 0 then
        if UnHookWindowsHookEx(TheHook) = True then
        begin
          TheHook := 0;
          FreeMem(PMsgBuff, Sizeof(TMsgBuff));
          PMsgBuff := nil;
          cbPlaybackFinishedProc(cbAppData);
        end;
    Exit;
  end;
  HC_GETNEXT:
  begin
    PEventMsg(lParam)^ := PMsgBuff^[CurrentMsg];
    PEventMsg(lParam)^.Time := StartTime + PMsgBuff^[CurrentMsg].Time;
    if ReportDelayTime then
    begin
      ReportDelayTime := False;
      TimeToFire := PEventMsg(lParam)^.Time - GetTickCount;
      if TimeToFire > 0 then Result := TimeToFire;
    end;
    Exit;
  end;
  HC_SYSMODALON:
  begin
    SysModalOn := True;
    CallNextHookEx(TheHook, Code, wParam, lParam);
    Exit;
  end;
  HC_SYSMODALOFF:
```

```
    begin
      SysModalOn := False;
      TheHook := 0;
      FreeMem(PMsgBuff, Sizeof(TMsgBuff));
      PMsgBuff := nil;
      cbPlaybackFinishedProc(cbAppData);
      CallNextHookEx(TheHook, Code, wParam, lParam);
      Exit;
    end;
  end;
  if Code < 0 then
    Result := CallNextHookEx(TheHook, Code, wParam, lParam);
end;
```

마지막으로 Playback() 함수를 구현하도록 하자.
이 함수는 재생할 매크로 파일의 이름을 PChar 데이터 형인 파라미터로 넘겨 받고, 이와 함께 재생이 끝난 후 호출할 어플리케이션의 콜백 함수와 어플리케이션 데이터를 파라미터로 넘겨 받아 사용한다.
구현 부분은 다음과 같다.

```
function Playback(lpFileName: PChar; EndPlayProc: TcbPlaybackFinishedProc;
  AppData: Longint): Integer; stdcall;
var
  TheFile: File;
begin
  Result := 0;
  if PMsgBuff <> nil then Exit;
  GetMem(PMsgBuff, Sizeof(TMsgBuff));
  if PMsgBuff = nil then Exit;
  Assign(TheFile, lpFileName);
  {$I-}
  Reset(TheFile, Sizeof(TEventMsg));
  {$I+}
  if IOResult <> 0 then
  begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
```

```
    PMsgBuff := nil;
    Exit;
  end;
  {$I-}
  MsgCount := FileSize(TheFile);
  {$I+}
  if IOResult <> 0 then
  begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
    {$I-}
      Close(TheFile);
    {$I+}
    if IOResult <> 0 then Exit;
    Exit;
  end;
  if MsgCount = 0 then
  begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
    {$I-}
    Close(TheFile);
    {$I+}
    if IOResult <> 0 then Exit;
    Exit;
  end;
  {$I-}
  Blockread(TheFile, PMsgBuff^, MsgCount);
  {$I+}
  if IOResult <> 0 then
  begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
    {$I-}
    Close(TheFile);
    {$I+}
```

```
    if IOResult <> 0 then Exit;
    Exit;
  end;
  {$I-}
  Close(TheFile);
  {$I+}
  if IOResult <> 0 then
  begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
    Exit;
  end;
  CurrentMsg := 0;
  ReportDelayTime := True;
  SysModalOn := False;
  cbPlaybackFinishedProc := EndPlayProc;
  cbAppData := AppData;
  StartTime := GetTickCount;
  TheHook := SetWindowsHookEx(WH_JOURNALPLAYBACK, JournalPlayBackProc,
    hInstance, 0);
  if TheHook = 0 then
  begin
    FreeMem(PMsgBuff, Sizeof(TMsgBuff));
    PMsgBuff := nil;
    Exit;
  end;
  Result := 1;
end;
```

소스가 다소 길지만, 많은 부분이 중복되고 StopRecording() 함수와 마찬가지로 I/O 에러 처리를 위해서 길어진 부분이 많기 때문이므로 이해하기는 쉬울 것이다.
일단 재생을 하기 위해 메모리 버퍼를 설정하고, 디스크에 있는 메시지를 버퍼에 읽어들인 후 콜백 함수와 어플리케이션 데이터를 전역 변수에 저장한다. 그리고, 현재 시각을 GetTickCount API 함수를 통해서 알아낸 후 WH_JOURNALPLAYBACK 후크에 대한 후크 함수를 설치한다. 재생이 성공적으로 이루어진 경우에는 1 을 반환한다.
이로써 DLL 파일의 구현이 모두 끝났다. 이를 어플리케이션에서 사용하려면 함수를

export 해야 한다. 이 부분은 다음과 같이 구현한다.

```
exports
  JournalRecordProc index 1 name 'JOURNALRECORDPROC' resident,
  StartRecording index 2 name 'STARTRECORDING' resident,
  StopRecording index 3 name 'STOPRECORDING' resident,
  JournalPlayBackProc index 4 name 'JOURNALPLAYBACKPROC' resident,
  Playback index 5 name 'PLAYBACK' resident;
```

그리고, DLL 이 처음 시작되면 메시지 버퍼는 메모리 블록을 가리키면 안되므로 다음과 같은 코드를 삽입한다.

```
begin
  PMsgBuff := nil;
end.
```

- 후크 DLL wrapper 의 작성

앞의 DLL 프로젝트를 컴파일 한 후, 맨 처음 예제와 같은 방식으로 어플리케이션에서 직접 DLL 을 적재해서 사용할 수도 있지만 보다 일반적이고, 쉽게 사용하기 위해서는 wrapper 유닛을 작성해서 사용하면 편리하다. Wrapper 유닛이란 DLL 의 함수를 쉽게 사용하기 위해서 파스칼 문법에 맞게 만든 인터페이스 유닛으로, 이를 이용하면 인터페이스 유닛의 이름을 사용하고자 하는 유닛의 uses 절에 추가하는 것으로 해결할 수 있다.
참고로, 델파이의 Win32 API 지원도 이러한 wrapper 유닛을 통해서 이루어 지는 것으로 windows.pas 등의 유닛이 대표적인 wrapper 유닛이다.
여기에 대해서는 DLL 을 다루는 장에서 더욱 자세하게 다루고 있으므로 이를 참고하기 바란다. 그러면, DLL 의 wrapper 유닛을 다음과 같이 작성한다.

```
unit HookExam;

interface

type
  TwMsg = LongInt;
  TwParam = LongInt;
  TlParam = LongInt;
```

```
  TcbPlaybackFinishedProc = procedure(AppData: Longint); stdcall;

function StartRecording: integer; stdcall;
function StopRecording(lpFileName: PChar): LongInt; stdcall;
function Playback(lpFileName: PChar; EndPlayProc: TcbPlaybackFinishedProc;
  AppData: Longint): Integer; stdcall;

implementation

function StartRecording; external 'ExamLib4' index 2;
function StopRecording; external 'ExamLib4' index 3;
function Playback; external 'ExamLib4' index 5;

end.
```

- 예제 어플리케이션의 동작 방식

후크 DLL 을 사용하는 어플리케이션에서는 콜백 함수로 PlaybackFinished()를 이용하도록 하자. 이 콜백 함수는 매크로의 재생이 완료되었을 때 호출된다. 즉, DLL 의 Playback() 함수에 PlaybackFinished() 콜백 함수의 주소를 넘겨 주어서 Playback()이 완료되는데로 이를 호출하게 된다. 그런데, PlaybackFinished() 함수의 주소를 저장하기 위해서 프로그램 시작시에 전역 변수를 하나 선언하고 프로그램이 종료될 때 이를 해제하도록 한다.
폼은 매크로의 기록을 시작할 때 사용하는 '기록 시작' 버튼, 기록을 중지할 때 사용하는 '기록 중지' 버튼, 매크로를 재생할 때 사용하는 '재 생' 버튼과 어플리케이션을 종료할 때 사용하는 '완 료'의 네 버튼으로 구성한다.
폼이 처음 생성될 때에는 '기록 시작'과 '완 료' 버튼을 사용 가능하도록 하고, '기록 중지'와 '재 생' 버튼은 기록을 중지시키거나, 재생할 것이 없으므로 이를 사용할 수 없도록 설정한다.
각 버튼의 역할은 다음과 같다.

1. 기록 시작:

더 이상의 메시지를 기록하거나 기록 중간에 어플리케이션을 종료하게 하면 안 되므로, '기록 시작'과 '완 료' 버튼을 사용 불가능하게 설정하고, 동시에 '기록 중지' 버튼은 사용이 가능하도록 설정한다. 그리고, DLL 의 StartRecording() 함수를 호출해서 메시지 기록을 시작한다. 이때 StartRecording() 함수가 에러 코드를 반환하면 사용자에게 에러 메시지를

보여주고, '기록 시작' 버튼이 눌리기 전의 상태로 재설정한다.

2. 기록 중지:

매크로를 저장할 파일 이름을 파라미터로 DLL 의 StopRecording() 함수를 호출한다. 그리고 '완 료'와 '기록 시작' 버튼을 사용 가능하도록 설정해서, 세션을 기록하거나 어플리케이션에서 빠져 나갈 수 있도록 허용한다. 또한, 기록이 성공적으로 이루어 졌을 경우에는 '재 생' 버튼도 사용할 수 있도록 설정한다.

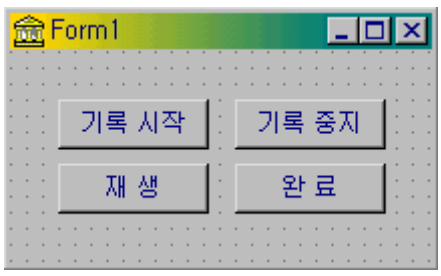
3. 재 생:

모든 버튼을 선택할 수 없도록 설정하고, 재생할 매크로가 저장된 파일 이름과 PlaybackFinished() 콜백 함수의 인스턴스 주소, 그리고 어플리케이션이 정의한 데이터를 받을 파라미터로 DLL 의 Playback() 함수를 호출한다.
매크로의 재생이 완료되면 후크 함수는 어플리케이션의 PlaybackFinished() 콜백 함수를 호출해서 재생이 완료되었음을 알림과 동시에 메인 윈도우를 어플리케이션 데이터로 넘겨준다. 이때 DLL 의 Playback() 함수가 에러 코드를 반환하면, '재 생' 버튼이 눌리기 이전의 상태로 버튼 들을 재설정한다.
PlaybackFininshed() 콜백 함수가 성공적으로 호출되면 '기록 시작', '재 생', '완 료' 버튼을 사용 가능하도록 설정한다.

4. 완 료:

프로그램을 완료한다.

그러면, 예제 어플리케이션의 폼을 다음과 같이 디자인 하자.



폼의 각 버튼의 역할은 앞에서 간단히 설명하였다.
먼저 매크로를 기록, 재생할 파일 이름을 다음과 같이 상수로 선언하고, 앞에서 작성한 후크 DLL 을 사용하기 위해 uses 절에 HookExam 을 추가한다.

```
const
  FILENAME = 'Hooking.MAC';
```

그리고, 처음 폼이 생성될 때의 버튼 들의 초기 값을 다음과 같이 설정한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1.Enabled := True;
  Button2.Enabled := False;
  Button3.Enabled := False;
  Button4.Enabled := True;
end;
```

여기서 Button1 은 '기록 시작', Button2 는 '기록 중지', Button3 는 '재 생', Button4 는 '완료' 버튼이다.
각 버튼의 역할 역시 해당되는 DLL 함수를 호출하고, 각 버튼의 Enabled 프로퍼티를 조절하는 것으로 각각의 이벤트 핸들러는 다음과 같다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Button2.Enabled := True;
  Button2.SetFocus;
  Button1.Enabled := False;
  Button3.Enabled := False;
  Button4.Enabled := False;
  if StartRecording = 0 then
  begin
    Button1.Enabled := True;
    Button1.SetFocus;
    Button2.Enabled := False;
    Button3.Enabled := False;
    Button4.Enabled := True;
    ShowMessage('기록을 시작할 수 없습니다 !');
  end;
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  if StopRecording(FILENAME) > 0 then
  begin
    Button1.Enabled := True;
    Button3.Enabled := True;
    Button3.SetFocus;
  end
  else
  begin
    Button1.Enabled := True;
    Button1.SetFocus;
    Button3.Enabled := False;
  end;
  Button2.Enabled := False;
  Button4.Enabled := True;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  Button1.Enabled := False;
  Button2.Enabled := False;
  Button3.Enabled := False;
  Button4.Enabled := False;
  if PlayBack(FILENAME, @PlaybackFinished, Handle) = 0 then
  begin
    Button1.Enabled := True;
    Button2.Enabled := False;
    Button3.Enabled := True;
    Button4.Enabled := True;
    Button3.SetFocus;
  end;
end;

procedure TForm1.Button4Click(Sender: TObject);
```

```
begin
  Close;
end;
```

간단하므로 이해에 별 무리가 없을 것으로 믿는다. 그런데 눈 여겨 보아야 할 부분은 Button3 의 OnClick 이벤트 핸들러에서 Playback() 함수를 호출하는 부분으로, 재생이 끝났을 때 이를 처리할 콜백 함수의 주소를 파라미터로 넘겨 주게 된다. 여기서는 PlaybackFinished 함수의 주소를 넘겨 주었다. 그러므로, 이 콜백 함수를 다음과 같이 작성한다.

```
procedure PlaybackFinished(AppData: LongInt); stdcall;
begin
  Form1.Button1.Enabled := True;
  Form1.Button2.Enabled := False;
  Form1.Button3.Enabled := True;
  Form1.Button4.Enabled := True;
  Form1.Button3.SetFocus;
end;
```

이 때 Form1 을 앞에 붙이는 이유는 이 함수가 TForm1 클래스의 메소드로 선언되지 않았기 때문이다.
어쨌든 후크 DLL 에 의해 Playback() 함수가 실행되면, 이 함수가 종료되는 데로 PlaybackFinished() 프로시저가 실행되어 버튼 들의 활성화 여부를 결정하게 된다.
이제 이 프로그램을 실행시키면 매크로 기록, 재생을 할 수 있게 된다.
'기록 시작' 버튼을 누르고 마음대로 마우스와 키보드를 이용한 작업을 하고, '기록 중지' 버튼을 누르면 이 메시지가 매크로로 기록이 될 것이며, 이를 '재 생' 버튼을 눌러서 재현할 수 있다. 한 번 해보면 얼마나 유용하게 쓰일 수 있을 지 알 수 있을 것이다.

정 리 (Summary)

이번 장에서는 Win32 환경에서 자주 사용되는 콜백 함수에 대한 설명과 콜백을 이용하여 구현할 수 있는 후킹 기법에 대해서 알아 보았다. 콜백 함수는 후킹 기법 말고도 다른 곳에서도 자주 사용되므로, 확실하게 이해해 두는 것이 좋다.