Autodesk®
Scaleform®

# Scaleform 게임 통신 개요

이 문서는 C++, 플래시, 액션스크립트 간에 Scaleform 3.1(이상)을 사용해서 통신하는 매커니즘에 대하여 설명한다.


저자: Mustafa Thamer



버전: 2.01

최종수정: 2010 년 9 월 21 일

# Copyright Notice

**Autodesk® Scaleform® 4.4**

**Disclaimer**



연락처:



| | |
|---|---|
| 문서 | Scaleform 게임 통신 개요 |
| 주소 | Autodesk Scaleform Corporation |
| | 6305 Ivy Lane, Suite 310 |
| | Greenbelt, MD 20770, USA |
| 웹사이트 | www.scaleform.com |
| 이메일 | info@scaleform.com |
| 직통전화 | (301) 446-3200 |
| 팩스 | (301) 446-3199 |

# 목차

# 1 C++, 플래시, 액션스크립트 인터페이싱 하기

플래시 액션스크립트 스크립트 언어는 쌍방향 무비 컨텐츠를 만들 수 있게 해준다. 버튼 클릭, 특정 프레임에 도달, 무비 로드 등의 이벤트는 동적으로 내용이 변경될 수 있도록 코드를 실행할 수도 있고, 무비의 흐름제어, 추가 무비 실행 등에도 사용된다. 액션 스크립트는 플래시에서 완벽한 미니게임을 만들 수 있을 정도로 강력하다. 대부분의 프로그래밍 언어와 마찬가지로 액션스크립트는 변수와 서브루틴을 지원하며 아이템이나 컨트롤을 표현하는 개체를 포함하고 있다.

애플리케이션과 플래시 컨텐츠 간에 통신을 하려면 복잡한 사용방법이 필요하다. Scaleform 는 플래시에서 제공하는 액션스크립트에서 이벤트와 데이터를 C++로 전달하는 표준 메커니즘을 지원한다. 또한, 직접 액션 스크립트 변수, 배열, 객체를 제어하고 액션스크립트의 서브루틴을 호출하는 편리한 C++인터페이스도 제공한다.

이 문서에서는 C++과 플래시에서 통신하는데 사용 가능한 몇 가지 메커니즘에 대하여 다룰 것이다.

| **ActionScript → C++** | **C++ → ActionScript** |
|---|---|
| **FSCommand**<br>간단. 문자열기반 함수 실행. 반환 값 없음. 사용중지됨. | **GFx::Movie::Get/SetVariable**<br>액션스크립트 데이터 접근, 문자열 경로 사용. |
| **ExternalInterface**<br>유연한 전달인자 관리, 반환 값이 있음, 사용권장. | **GFx::Movie::Invoke**<br>액션스크립트 함수 호출, 문자열 경로 사용. |
| **Direct Access Objects**<br>GFx::Value 를 데이터와 함수 접근 객체로 사용. 고 효율 | |

## *1.1 액션스크립트에서 C++로*

Scaleform 는 C++에서 액션스크립트로부터 이벤트를 받기 위해서 *FSCommand* 와 *ExternalInterface* 라는 2 가지 방법을 제공한다. 이들 메커니즘은 표준 액션스크립트 API 의 일부며 관련 자료는 플래시 문서에서 찾을 수 있다. FSCommand 와 ExternalInterface 모두 C++이벤트 핸들러를 등록해서 이벤트 통지를 받는다. 이들 핸들러는 Scaleform 의 상태 형식으로 등록되기 때문에 용도에 따라서 GFx::Loader, GFx::MovieDef , GFx::Movie 에서 등록이 가능하다. 자세한 내용은 Scaleform 상태(states) 관련 문서를 참고하라.

FSCommand 이벤트는 액션스크립트의 'fscommand'함수에 의해서 발생된다. Fscommand 는 딱 2 개의 문자열 인자를 전달할 수 있는데, 하나는 명령이고 하나는 데이터다. 이들 문자열 값은 C++핸들러에 2 개의 상수 문자열 포인터로 전달된다. C++ fscommand 핸들러는 안타깝게도 액션스크립트의 fscommand 가 반환 값을 받지 않는 관계로 값을 반환할 수 없다.

ExternalInterface 이벤트는 액션스크립트의 *flash.external.ExternalInterface.call* 함수가 호출되면 발생한다. 이 인터페이스는 명령이름 외에 임의의 액션스크립트 값을 전달할 수 있다. C++ ExternalInterface핸들러는 상수 문자열 포인터로 된 명령이름과 GFx::Value배열을 받게 된다. C++ ExternalInterface핸들러는 GFx::Value를 액션스크립트의 ExternalInterface.call에 반환할 수 있다. 액션스크립트의 ExternalInterface 클래스는 외부 API들 중의 하나이며, 액션 스크립트와 플래시 플레이어와 간단한 통신이 가능한 애플리케이션 프로그래밍 인터페이스이다. (지금의 경우에서는 애플리케이션으로 Scaleform 를 사용함)

유연성 등의 한계문제가 있어서 FSCommands 는 ExternalInterface 로 대치되었으면 더 이상 사용을 권장하지 않는다. 여기서는 구버전 지원 등의 문제로 언급되는 것이다.

FSCommands 및 ExternalInterface 는 Direct Access API 와 GFx::FunctionHandler 인터페이스에 비해 구식이 되었다. 이 인터페이스는 일반 함수같이 ActionScript VM 내부에서 직접 콜백을 C++ 메서드에 할당할 수 있게 한다. ActionScript 에서 그 함수가 Invoke 되었을 경우 C++ 콜백으로 Invoke 시킨다. Direct Access API 섹션을 보면 GFx::FunctionHandler 에 대해 더 많은 정보를 얻을 수 있다.

### 1.1.1 FSCommand 콜백

액션스크립트 fscommand 함수는 명령과 데이터 인자를 호스트 애플리케이션에 전달한다. 다음은 전형적인 액션스크립트 사용 예이다.

```
fscommand("setMode", "2");
```

bool 이나 정수 같은 비문자열 인자는 문자열로 변형된다.

액션스크립트의 fscommand 호출은 2 개의 문자열을 Scaleform 의 FSCommand 핸들러로 보낸다. 애플리케이션은 GFx::FSCommandHandler 를 상속받아서 fscommand 핸들러를 등록하고, GFx::Loader, GFx::MovieDef 나 개별 GFx::Movie 객체와 상태를 공유하도록 클래스의 인스턴스를 등록한다. 상태는 GFx::Loader -> GFx::MovieDef -> GFx::Movie 같이 위임된다는 것을 명심하자. 이들 나중에 나오는 어떤 인스턴스중에 하나라도 재지정(override)될 수 있다. 이 말은 특정 GFx::Movie 가 자신만의 GFx::RenderConfig(예를 들어 따로 EdgeAA 제어)이나 GFx::Translator(다른 언어로 번역되도록)를 가지게 하려면 이들 객체를 설정할 수 있으며 이렇게 설정된 어떤 상태든지 모두 위임사슬(delegation chain)의 순서대로 전달된다는 것이다(즉 GFx::Loader 처럼). 명령 핸들러가 GFx::Movie 에 설치되었다면 FSCommand 호출을 무비 인스턴스에서만 받을 것이다. GFxPlayerTiny 예는 이 절차를 보여주는 예다("FxPlayerFSCommandHandler"를 검색해 보라).

다음은 fscommand 핸들러 설저으이 예다. 핸들러는 GFx::FSCommandHandler 를 상속받았다.

```
class OurFSCommandHandler : public FSCommandHandler
{
    public:
     virtual void Callback(Movie* pmovie, const char* pcommand,
                           const char* parg)
    {
        Printf("FSCommand: %s, Args: %s", pcommand, parg);
    }
};
```

콜백 메서드는 액션스크립트의 fscommand 에서 전해진 2 개의 문자열 인자를 받으며, 또한 fscommand 를 호출한 무비 인스턴스의 포인터도 전달된다. 여기서의 fscommand 는 그냥 단순히 각각의 이벤트를 디버그 콘솔에 출력하도록 했다.

GFx::Loader 객체를 생성했으면 핸들러를 등록한다.

```
// Register our fscommand handler
Ptr<FSCommandHandler> pcommandHandler = *new OurFSCommandHandler;
gfxLoader->SetFSCommandHandler(pcommandHandler);
```

GFx::Loader 에 핸들러를 등록하면 모든 GFx::Movie 와 GFx::MovieDef 가 이 핸들러를 상속받는다. 이 기본 설정을 재정하면 SetFSCommandHandler 가 개별 무비 인스턴스에서 호출가능하다.

일반적으로 C++이벤트 핸들러는 넌블러킹(non-blocking)이며 가능한 한 빨리 호출자에게 반환해야 한다. 이벤트 핸들러는 일반적으로 Advance 나 Invoke 호출에서만 호출된다.

### 1.1.2 외부 인터페이스 API

플래시 ExternalInterface.call 메서드는 fscommand 와 비슷하지만 유연한 인자처리와 반환 값 때문에 더 유용하다. ExternalInterface 핸들러 등록은 fscommand 핸들러 등록과 유사하다. 다음은 숫자와 문자열 인자를 출력하는 ExternalInterface 핸들러의 예다.

```
class OurExternalInterfaceHandler : public ExternalInterface
{
    public:
     virtual void Callback(Movie* pmovieView, const char* methodName,
                                const Value* args, unsigned argCount)
    {
         Printf("ExternalInterface: %s, %d args: ", methodName, argCount);
          for(unsigned i = 0; i < argCount; i++)
          {
                 switch(args[i].GetType())
                 {
                     case Value::VT_Number:
                        Printf("%3.3f", args[i].GetNumber());
                        break;
                     case Value::VT_String:
```

```
                    Printf("%s", args[i].GetString());
                    break;
                }
                Printf("%s", (i == argCount - 1) ? "" : ", ");
            }
            Printf("\n");

            // return a value of 100
            Value retValue;
            retValue.SetNumber(100);
            pmovieView->SetExternalInterfaceRetVal(retValue);
        }
    };
```

일단 핸들러가 구현되면 다음처럼 GFx::Loader 에 인스턴스를 등록 할 수 있다.

```
Ptr<GFxExternalInterface> pEIHandler = *new OurExternalInterfaceHandler;
gfxLoader.SetExternalInterface(pEIHandler);
```

이렇게 하면 액션스크립트의 ExternalInterface 호출에 의해서 콜백 핸들러가 작동한다.

#### 1.1.2.1 FxDelegate

앞서 소개한 ExternalInterface 핸들러는 매우 기본적인 콜백 기능만을 제공한다. 실제로는 특정 메서드명에 의해서 ExternalInterface 콜백이 발생할 때 지정된 함수가 호출되도록 등록하고 싶을 것이다. 이를 위해서는 메서드명과 연관된 함수를 등록할 수 있도록 더 진보된 콜백 시스템이 필요하며, 또한 해쉬 테이블(혹은 비슷한 것)을 사용해서 적절한 GFx::Value 인자와 연계되도록 해야 한다. 여기서 소개하는 것은 FxDelegate 다(*Apps\Samples\GameDelegate\FxGameDelegate.h 참고*)

FxDelegate 는 GFxExternalInterface 로부터 상속된 것이다. Register/Unregister 핸들러 함수가 있으며 이 함수에 의해서 메서드명에 따른 위임자 핸들러를 설치할 수 있다.

다음은 FxDelegate 사용예다. FxDelegate 를 사용해서 밉맵을 구현한 HUD Kit 을 살펴보라. 다음은 FxDelegate 로 콜백을 등록한 밉맵 데모 코드다.

```
// Minimap Begin
void FxPlayerApp::Accept(FxDelegateHandler::CallbackProcessor* cbreg)
{
```

```
        cbreg->Process("registerMiniMapView", FxPlayerApp::RegisterMiniMapView);
        cbreg->Process("enableSimulation", FxPlayerApp::EnableSimulation);
        cbreg->Process("changeMode", FxPlayerApp::ChangeMode);
        cbreg->Process("captureStats", FxPlayerApp::CaptureStats);

        cbreg->Process("loadSettings", FxPlayerApp::LoadSettings);
        cbreg->Process("numFriendliesChange", FxPlayerApp::NumFriendliesChange);
        cbreg->Process("numEnemiesChange", FxPlayerApp::NumEnemiesChange);
        cbreg->Process("numFlagsChange", FxPlayerApp::NumFlagsChange);
        cbreg->Process("numObjectivesChange", FxPlayerApp::NumObjectivesChange);
        cbreg->Process("numWaypointsChange", FxPlayerApp::NumWaypointsChange);
        cbreg->Process("playerMovementChange", FxPlayerApp::PlayerMovementChange);
        cbreg->Process("botMovementChange", FxPlayerApp::BotMovementChange);

        cbreg->Process("showingUI", FxPlayerApp::ShowingUI);
    }
```

FxDelegate 는 한결 깔끔한 콜백 핸들링과 등록 시스템을 제공한다. 따라서 개발자들에게 이 예제가 어떻게 작동하는 지를 눈 여겨 봐 둘 것을 강력히 권고한다.

#### 1.1.2.2 액션스크립트에서 외부 인터페이스 사용

액션스크립트에서 외부 인터페이스 호출은 다음과 같이 초기화 된다.

```
import flash.external.*;
ExternalInterface.call("foo", arg);
```

여기서 'foo'는 메서드명이고 'arg'는 기본형 인자다. 인자는 콤마로 구분해서 0 개 혹은 그 이상을 지정할 수 있다. ExternalInterface API 와 관련해서는 플래시 문서를 참고하기 바란다.

ExternalInterface 가 앞서와 같이 임포트 되지 않았다면 ExternalInterface 호출은 항상 전체를 써줘야 한다.

```
flash.external.ExternalInterface.call("foo",arg)
```

#### 1.1.2.3 ExternalInterface.addCallback

ExternalInterface.addCallback 함수는 전역적으로 액션스크립트 메서드를 별명으로 등록한다. 이렇게 하면 C++에서 패스 없이도 호출이 가능하다. 이 함수는 메서드 등록과 컨텍스트 호출(this 객체)을 하나의 별명으로 가능하게 해준다. 등록된 별명은 C++ GFx::Movie::Invoke() 메서드로 호출 가능하다.

사용 예:

**액션스크립트 코드:**

```
import flash.external.*;

var txtField:TextField = this.createTextField("txtField",
                              this.getNextHighestDepth(), 0, 0, 200, 50);
var aliasName:String = "setText";
var instance:Object = txtField;
var method:Function = SetText;
var wasSuccessful:Boolean = ExternalInterface.addCallback(aliasName, instance,
                                                    method);

function SetText()
{
    trace(this + ".SetText ");
    this.text = "INVOKED!";
}
```

이제 this 와 함께 SetText 를 호출해서 txtField 에 값을 설정 하는 것이 가능하다.

**C++코드:**

```
pMovie->Invoke("setText", "");
```

결과는 "txtField.SetText"이며, txtField 의 텍스트 필드를 "INVOKED!"로 설정한다.

GFx::Movie::Invoke()에 관한 추가 정보는 1.2.2 를 참고하라.

## *1.2 C++에서 액션스크립트로*

앞 장에서 우리는 액션스크립트가 어떻게 C++을 호출하는지 설명했다. 이번 장에서는 반대 방향 통신은 어떻게 하는가 하는 것이다. 즉, Scaleform 기능을 사용해서 C++프로그램에서 플래시 무비와 통신하는 것 말이다. Scaleform 는 C++함수에서 액션스크립트 변수를 직접 get/set 하는 것을 지원한다(단순형, 복합형, 배열). 또한 액션스크립트 서브루틴 호출도 지원한다.

Direct Access API 는 더욱 큰 편리함과 접근을 위한 효율적인 인터페이스를 제공하며, C++에서 ActionScript 의 변수를 조작하는데 편리하다. Direct Access API 섹션을 보면 이 인터페이스에 대한 더 많은 정보를 얻을 수 있다.

### 1.2.1 액션스크립트 변수 조작

Scaleform 는 GetVariable 과 SetVariable 을 지원하는데, 이를 통해 액션스크립트 변수의 직접 조작이 가능하다. 성능이 중요한 경우에는 2 장의 Direct Access API 를 참고해서 변수와 객체 속성을 수정하는 효율적인 방법을 알 수 있을 것이다. Set/GetVariable 가 성능상의 이유로 권장되고 있지는 않지만 Direct Access API 에 비해서 편리한 경우가 있다. 예를 들어서 애플리케이션 실행주기 전체에서 딱 한번 호출되는 한 개의 값을 바꾸기 위해 Direct Access 호출인 GFx::Value::SetMember 를 사용할 필요는 없다. 특히, 타겟이 깊은 단계로 내포되어 있다면 말이다. 또한, 액션스크립트 객체의 참조를 얻어내는 것은 일반적으로 GetVariable 로 가능하다. 이를 이용해서 일단 GFx::Value 참조를 얻고, Direct Access 연산에 사용되는 것이다.

다음은 액션스크립트 카운터를 GetVariable 과 SetVariable 을 사용해서 증가시키는 예다.

```
int counter = (int)pHUDMovie->GetVariableDouble("_root.counter");
counter++;
pHUDMovie->SetVariable("_root.counter", Value((double)counter));
```

GetVariableDouble 은 _root.counter 변수의 값을 반환한다. 이 값은 자동적으로 C++더블형으로 변환된다. 초기에는 변수가 존재하지 않으므로 GetVariableDouble 이 0 을 반환한다. 그리고 나서

counter 가 다음 라인에서 증가되고 새로운 값이 _root.counter 에 SetVariable 을 사용해서 저장된다. GetVariable 과 SetVariable 의 다른 사용 예는 GFx::Movie 온라인 도움말을 참고하라.

SetVariable 은 GFx::Movie::SetVarType 타입의 세 번째 옵션 인자가 있는데, 이는 "sticky"를 선언하는데 사용된다. 이는 변수가 대입은 되었으나 플래시 타임라인에서 아직 생성되지 않은 경우에 유용하다. 예를 들어서 무비의 3 프레임까지는 _root.mytextfield 라는 텍스트 필드가 생성되지 않았다고 하자. 만약 SetVariable("_root.mytextfield.text", "testing", SV_Normal)가 무비 생성 직후인 1 프레임에서 호출되었다면 이러한 대입은 아무런 효과가 없을 것이다. 하지만, SV_Sticky (기본값)로 호출되었다면 _root.mytextfield.text 값이 3 프레임에서 유효할때까지 일단 큐에 들어가게된다. 이렇게 함으로써 C++에서 무비를 초기화하기 쉽게 된다. 일반적으로 SV_Normal 이 SV_Sticky 보다 효율적이므로 가능하다면 SV_Normal 을 사용하기 바란다.

데이터 배열에 대한 접근을 위해서 Scaleform 는 SetVariableArray 과 GetVariableArray 함수를 제공한다

SetVariableArray 는 배열요소를 지정된 구간에 지정된 형식으로 세팅한다. 만약 배열이 존재하지 않으면 생성해 버린다. 배열이 존재하지만 용량이 충분하지 않을 때는 적당한 크기로 조절한다. 하지만, 현재 크기보다 개수가 작은 배열요소일 경우에는 크기가 조절되지 않는다. GFx::Movie::SetVariableArraySize 는 배열의 크기를 설정하는데 현재 배열의 크기가 이전보다 더 적어질 경우에 유용하다.

다음 예는 SetVariableArray 를 사용해서 문자열 배열을 설정하는 예다.

```
int idxInASArray = 0;
const char* strarr[2];
strarr[0] = "This is the first string";
strarr[1] = "This is the second one";
pMovie->SetVariableArray(Movie::SA_String, "_root.strArray",
                                    idxInASArray, strarr, 2);
```

다음은 앞서 예제를 wide 문자열로 수정한 예다.

```
int idxInASArray = 0;
const wchar_t* strarr[2];
strarr[0] = L"This is the first string";
```

```
strarr[1] = L"This is the second one";
pMovie->SetVariableArray(Movie::SA_StringW, "_root.strArray",
                         idxInASArray, strarr, 2)
```

GetVariableArray 메서드는 제공된 데이터 버퍼를 AS 배열의 결과값으로 채운다. 버퍼는 요청된 아이템 개수를 담을 만큼 충분히 커야 한다.

### 1.2.2 액션스크립트 서브루틴

액션스크립트 변수 변경뿐만이 아니라 액션스크립트 메서드도 GFx::Movie::Invoke()를 통해 호출가능하다. 이 기능은 애니메이션 트리거, 현재 프레임 변경, UI 컨트롤 상태를 프로그램적으로 변경하기, 버튼이나 텍스트 같은 UI 컨텐츠를 동적으로 생성하기 등에 유용하다. 수행속도가 중요한 경우에는 Direct Access API 장을 참고하면 더 효율적인 메서드 호출과 컨트롤 액션 플로우를 알 수 있을 것이다.

사용 예:

다음 액션스크립트 함수는 슬라이더 그립 위치를 범위에 맞춰서 설정하는데 사용가능하다. mySlider' 객체가 _ root 레벨에 있다고 가정한다. SetSliderPos 는 mySlider' 객체 내부에 다음과 같이 정의되어 있다.

**액션스크립트 코드:**

```
this.SetSliderPos = function (pos)
{
   // Clamp the incoming position value
   if (pos<rangeMin) pos = rangeMin;
   if (pos>rangeMax) pos = rangeMax;
   gripClip._x = trackClip._width * ((pos-rangeMin)/(rangeMax-rangeMin));
   gripClip.gripPos = gripClip._x;
}
```

gripClip 은 mySlider 내에 내포된 무비클립이며 pos 는 항상 rangeMin/Max 범위에서 유효하다. 사용자는 다음과 같이 함수 호출한다.

**C++코드:**

```
Value result;
bool bInvoked = pMovie->Invoke("_root.mySlider.SetSliderPos", &result, "%d",
                               newPos);
```

Invoke 는 실제로 함수가 호출되면 true 를 그렇지 않으면 false 를 반환한다.

액션스크립트 함수를 호출할 때는 로드가 완료되어 있는지 확인해야 한다. Invoke 사용에 있어서의 일반적인 에러는 그 액션스크립트의 루틴이 아직 가용하지 않는 경우가 대부분이며, 이러한 경우 그 에러는 Scaleform 로그로 출력될 것이다. 액션스크립트 루틴은 연관된 프레임이 재생되거나 연관된 내포 객체가 로드되기 전에는 사용할 수 없다. 1 프레임에 있는 모든 액션스크립트 코드는 GFx::Movie::Advance 가 최초로 호출되거나 initFirstFrame 값을 true 로 해서 GFx::MovieDef::CreateInstance 가 호출되면 사용할 수 있다.

Invoke 에 덧붙여서, GFx::Movie::IsAvailable()메서드는 일반적으로 Invoke 를 호출 하기 전에 그 액션스크립트 함수가 존재하는지 확인하는데 사용된다.

```
Value result;
if (pMovie->IsAvailable("parentPath.mySlider.SetSliderPos"))
  pMovie->Invoke("parentPath.mySlider.SetSliderPos", &result, "%d", newPos);
```

이 예에서는 printf'스타일 Invoke 를 사용했다. 이 경우의 Invoke 는 포맷 문자열에 맞춰서 전달인자를 받는다. 다른 버전의 함수는 GFx::Value 를 사용해서 비문자열 인자를 효율적으로 처리하고 있다.

```
Value args[3], result;
args[0].SetNumber(i);
args[1].SetString("test");
args[2].SetNumber(3.5);
pMovie->Invoke("path.to.methodName", &result, args, 3);
```

InvokeArgs 는 전달인자 리스트의 포인터를 넘기기 위해서 va_list 인자를 취한다는 것 외에는 완전히 동일하다. Invoke 와 InvokeArgs 는 printf 와 vprintf 의 관계와 유사하다.

### 1.2.3 경로(Paths)

사용자 애플리케이션 내에서 플래시 요소에 접근할 때 객체 탐색에 전체 패쓰가 필요한 경우가 있다.
다음 GFx::Movie 함수들은 전체 경로가 필요하다.

```
Movie::IsAvailable(path)
Movie::SetVariable(path, value)
Movie::SetVariableDouble(path, value)
Movie::SetVariableArray(path, index, data, count)
Movie::SetVariableArraySize(path, count)
Movie::GetVariable(value, path)
Movie::GetVariableDouble( path)
Movie::GetVariableArray(path, index, data, count)
Movie::GetVariableArraySize(path)
Movie::Invoke(pathToMethodName, result, argList, ...)
```

내포된 객체 경로를 정확히 얻어내려면 모든 부모 무비클립이 유일한 인스턴스 명을 가져야 한다. 내포된 객체명은 점(.)으로 구분하며 대소문자를 구분한다.

다음 예에서 무비클립명 "clip2"는 메인 스테이지에 있는 다른 무비클립 "clip1"에 내포되어 있다.

메인 스테이지→clip1→clip2

1. clip1 은 메인 스테이지 내에
2. clip2 는 clip1 내에

유효한 경로는 다음과 같다.

```
"clip1.clip2"
"_root.clip1.clip2"
"_level0.clip1.clip2"
```

무효한 패쓰는 다음과 같다.

"Clip1.clip2" ←대소문자 오류. Clip1 은 소문자

"clip2" ←부모 clip1 패쓰명 없음

ActionScript2 에서 "_root"와 "_level0" 이름은 선택사항이며 특정 베이스 레벨 룩업을 강제하는 데 사용할 수 있습니다. 그렇지만 ActionScript 3 에서는 기본적으로 스테이지 단계에서 룩업이 이루어지기 때문에 이들 옵션이 필요합니다. 이전 버전과의 호환성을 위해 ActionScript 3 에서 루트에 액세스하는 데 사용할 수 있는 다음과 같은 별명들을 제공합니다. _root, _level0, root, level0. 레벨에서 별도의 SWF 파일을 불러오는 경우 _root 는 현재 레벨의 베이스를 참조하며 사용자가

특정 레벨을 선택할 수 있도록 위에 나온 구문처럼 _levelN 을 지정합니다.

참고

- 플래시 스튜디오의 테스트 무비(Ctrl+Enter), (**Ctrl+Alt+V**)(변수)를 누르면 객체와 변수 경로를 체크할 수 있다.

- 플래시 스튜디오의 씬 1 링크로 시작하는 타임라인 스퀀스는 타겟 경로를 나타내지 못한다. 오브젝트 경로에서 실제로 사용되지 않는 특정 요소들이 나열된다. ), (**Ctrl+Alt+V**)(변수)로 팝업을 열어서 실제 유효한 경로를 체크한다.

- 무비 내에서 무비를 loadMovie 명령으로 로드할때는 객체가 레벨을 변경시키기 때문에 문제가 발생할 수 있다. _level0 에 존재하던 객체들이 이제는 _level1 이나 _level2 나 타겟 무비클립 내에 있다(이는 어떤 레벨에 로드 했느냐에 따라 다르다). 다른 레이어에 있는 객체를 참고하려면 간단하게 `_levelN.objectName` 이나 `_levelN.objectPath.objectName` 를 사용하면 된다(여기서 N 은 레벨번호)

이번 장의 상당수 내용은 다음 2 개의 문서를 참고하였으므로 읽어볼 것을 권한다.

1. Paths to Objects and Variables  Jesse Stratford 저
2. Advanced Pathing  Jesse Stratford 저

# 2 Direct Access API

GFx::Value 의 Direct Access 지원은 Scaleform 3.1 에서 액션스크립트 런타임과 통신하는데 있어서 가장 중요한 개선사항이다. 액션스크립트 통신 API 는 복합객체(와 이들 객체내의 개별 멤버)에 대한 효율적 질의가 가능하도록 확장되었다. 예를 들면, 내포된 다른 종류로 이루어진(이질적인) 데이터 구조를 UI 에서 게임으로 주고받는 것이 이제 가능해졌다는 것이다. Direct Access API 를 사용하는 더 심도 있는 예를 보려면 HUD 키트에 있는 Minimap Demo 를 보기 바란다. 이 예제에서는 미니맵 상에서 대량의 플래시 객체를 업데이트 할 때 수행능력을 어떻게 향상시킬 수 있는지 알 수 있다.

Direct Access API 는 GFx::Values 로 표현되는데 간단한 액션스크립트 타입, 객체, 배열, 디스플레이 객체 등을 포함하고 있다. 디스플레이 객체는 특별한 형태의 객체형이며 스테이지 상의 요소들(무비클립, 버튼, 텍스트필드)에 해당한다. GFx::Value 클래스 API 는 배열로 처리하는 방법을 포함해서 멤버 값들을 설정하고 읽는 완벽한 함수셋을 제공한다. GFx::Movie 클래스 API 는 이제 객체와 배열을 생성하는 메서드를 포함하고 있다. 자세한 내용은 GFx::Value 에 대한 도움말을 참고하라.

Direct Access API 는 애플리케이션이 (GFx::Value 형)C++변수를 액션스크립트의 객체와 직접 결합하도록 해준다. 일단 이러한 결합이나 참조가 이루어지면 이들 변수는 손쉽게 또한 효율적으로 액션스크립트 객체를 수정하는데 사용할 수 있다. 이러한 개선이 있기 전에는 사용자가 GFx::Movie 를 경유하거나 문자열 경로를 지정해야만 했다. 이 방법은 경로를 파싱하거나 액션스크립트 객체를 찾는 등의 과정 때문에 수행성능 저하를 야기시켰다. 복합객체를 사용함으로써 저렴함 파싱 비용과 호출 할 때 마다 발생하던 탐색 오버헤드가 제거된 것이다.

Direct Access API 를 사용하면 과거에 GFx::Movie 레벨에서만 사용 가능하던 많은 처리들이 직접적인 액션스크립트 객체에 대해서 GFx::Value 형으로 가능하다. 따라서 더 깔끔하고 더 효율적인 코드가 나온다. 예를 들어서 root 에 존재하는 'foo'라는 객체의 메서드를 호출할 때 무비의 Invoke 를 호출 하는 것이 아니라 객체의 참조를 얻어 직접 Invoke 호출이 가능한 것이다.

1. GFx::Movie Invoke 메서드 (비효율적임):

```
Value ret;
Value args[N];
pMovie->Invoke("_root.foo.method", &ret, args, N);
```

2. Direct Access Invoke 메서드 (개선됨):

```
(foo 는 _root.foo 에 있는 액션스크립트 객체의 참조를 갖고 있다고 가정)
Value ret;
Value args[N];
bool = foo.Invoke("method", &ret, args, N);
```

Invoke 와 함께 Direct Access API 를 통해 액션스크립트 객체에 값을 get, set 하고, 호출에 대한 검증에 사용 가능한 GFx::Value::HasMember, GetMember, SetMember 가 제공된다.

다음은 (GFx::Value 에 저장된)movieClip 의 텍스트 필드를 업데이트하기 위해서 GetMember 를 사용하는 예이다.

```
Value tf;
MovieClip.GetMember("textField", &tf);
tf.SetText("hello");
```

이 예에서 우리는 먼저 movieClip 객체의 textfield 멤버를 얻고, 그리고 나서 SetText 함수를 사용해서 텍스트 값을 업데이트 하고 있다.

## *2.1 객체와 배열*

Direct Aceess API 에서 제공하는 또 하나의 중요한 기능은 액션스크립트 객체나 객체의 계층구조를 생성하는 것이다. 이 방법을 사용하면 임의의 깊이와 구조를 가지는 객체를 생성해서 C++에서 관리할 수 있다. 예를 들어서 다음 코드는 2 개의 객체를 계층구조로 생성하며, 두 객체 중 하나를 다른 하나의 자식으로 만든다.

```
Movie* pMovie = … ;
Value parent, child;
pMovie->CreateObject(&parent);
pMovie->CreateObject(&child);
parent.SetMember("child", child);
```

CreateObject 는 액션스크립트 객체의 인스턴스를 생성한다. 또한, 지정된 클래스형의 인스턴스가 필요한 경우에는 완전한 형태의 클래스명이 추가로 전달될 수 있다.

예:

```
Value mat, params[6];
// (set params[0..6] to matrix data) …
pMovie->CreateObject(&mat, "flash.geom.Matrix", params, 6);
```

객체와 배열을 모두 사용하는 좀 더 복잡한 경우를 생각해보자. 다음 예는 복합객체 요소 배열을 생성한다.

```
// (Assuming pMovie is a GFx::Movie*):
Value owner, childArr, childObj;
pMovie->CreateArray(&owner);         // create parent array
pMovie->CreateArray(&childArr);           // create child array
pMovie->CreateObject(&childObj);     // create child object
bool = owner.SetElement(0, childArr);       //set parent[0] to child array
bool = owner.SetElement(1, childObj);      // set parent[1]to child object
…
bool = foo.SetMember("owner", owner); // later, set the 'owner' variable in
                                   // object foo, to the parent object
```

GFx::Value::VisitMembers()함수는 객체의 public 멤버를 순회하는데 사용 가능하다. 이 함수는 ObjectVisitor 클래스의 인스턴스가 필요한데, 이 클래스는 간단한 Visit 콜백이 오버라이드(override)되어 있어야 한다. 이 함수는 객체형(배열 및 디스플레이 객체 포함)에만 유효하다. *VisitMembers* 를 사용해서 클래스 인스턴스를 완전히 탐색할 수는 없다는 것에 유의하자. 메서드들은 프로토타입(함수 원형)이기 때문에 열거(*enumerable)*가 불가능하다. 멤버의 접근성과 속성을 알아보려면 액션스크립트의 ASSetPropFlags 를 사용하라.

```
e.g.: _global.ASSetPropFlags(MyClass.prototype, ["someFunc", "__get__number",
                                          "test"], 6, 1);
```

(getter/setter)속성들은 ASSetPropFlags 를 사용하더라도 VisitMembers 로 절대로 열거할(enumerable) 수 없다. 이들은 복합객체 인터페이스로 가능하며, VT_Object 형태로 반환된다. (ASSetPropFlags 로 열거된 경우의)함수들은 VT_Object 형태로 반환된다.

## 2.2 디스플레이 객체

Direct Access API 는 특별한 형태의 객체형을 지원한다. 이 객체는 DisplayObject 인데 스테이지 상의 독립체인 무비클립, 버튼, 텍스트 필드와 연관된 것이다. 이들에 대한 출력 속성에 접근하기 위해서 DisplayInfo API 도 제공된다. DisplayInfo 를 사용하면 알파, 회전, 가시성, 위치, 옵셋, 크기 등의 디스플레이에 대한 객체 속성을 손쉽게 설정할 수 있다. 물론 이들 속성값에 SetMember 함수를 사용해도 되지만 말이다. SetDisplayInfo 를 호출하면 이들 객체의 출력 속성에 직접적으로 접근하기 때문에 가장 빠르다. 이들 메서드를 사용하면 대상 객체에 직접 접근하여 처리하기 때문에 GFx::Movie::SetVariable 보다 빠르다.

다음 코드는 SetDisplayInfo 를 사용해서 무비클립의 위치와 회전값을 변경하는 예이다.

```
// Assumes MovieClip is a GFx::Value*
Value::DisplayInfo info;
GPointF pt(100,100);
info.SetRotation(90);
info.SetPosition(pt.x, pt.y);
MovieClip.SetDisplayInfo(info);
```

## 2.3 함수 객체

ActionScript2 버추얼 머신에서의 함수들은 기본 객체와 비슷하다. 이 함수 객체들은 사용에 따라 성찰 (Introspection: 함수 객체로부터 메타 정보를 되돌리는 기능)정보를 제공하는 할당 멤버들이 될 수 있다. GFx::Movie::CreateFunction 메서드와 함께, 개발자들은 C++ 콜백 객체를 감싸는 함수 객체를 생성 할 수 있다. CreateFunction 에 의해 VM 상에서 그 함수 객체는 객체의 멤버처럼 할당이 된다. 이 AS 함수가 호출 될 때, 그 C++ 콜백은 차례로 호출된다. 이 기능은 개발자들이 그들 자신의 콜백 핸들러들을 추가적인 위임(fscommand 또는 ExternalInterface 를 통한)에 대한 오버헤드 없이 VM 에 직접 등록 하게 해준다.

다음은 커스텀 콜백 의 생성 및 AS 객체의 멤버를 할당하는 것에 대한 예제이다.

```
Value obj;
pmovie->GetVariable(&obj, "_root.obj");
```

```
class MyFunc : public FunctionHandler
{
    public:
        virtual void Call(const Params& params)
        {
              // Callback logic/handling
        }
};
Ptr<MyFunc> customFunc = *SF_HEAP_NEW(Memory::GetGlobalHeap()) MyFunc();
Value func;
pmovie->CreateFunction(&func, customFunc);
obj.SetMember("func", func);
```

이 메서드는 ActionScript 내부에서 호출될 수 있다. (_root 타임라인 안에서)

```
obj.func(param1, param2, param3);
```

Params 구조체는 다음이 포함된 Call()메서드를 전달 할 것이다.

- `pRetVal`: 반환 값으로 호출자에게 다시 전달되는 GFx::Value 의 포인터이다. 만약 복합 객체가 전달되면, 컨테이너 생성을 위해 이 포인터를 pMovie->CreateObject() 또는 CreateArray()에 전달한다. 원시 타입들(Numbers, Booleans 등)을 위한 적절한 setter 를 호출한다.
- `pMovie`: 함수가 호출된 무비클립의 포인터.
- `pThis`: 컨텍스트 호출자. 컨텍스트가 존재하지 않거나 무효화 되어있다면 undefined 일 것이다.
- `ArgCount`: 콜백으로 전달되는 인수들의 갯수
- `pArgs`: 콜백 함수로 전달되는 인수의 GFx::Value 배열. [] 연산자를 사용해서 개별적인 변수로 접근한다.

  예: `Value firstArg = params.pArgs[0];`
- `pArgsWithThisRef`: 인수들의 배열과 무언가가 리스트의 시작부분에 첨가 되었을 때의 컨텍스트. 이것은 다른 함수 객체들과 연결하는데 사용된다. (아래는 사용자 지정 동작에 대한 예제이다.)
- `pUserData`: 함수 객체가 등록 될 때 사용자 지정 데이터의 집합. 이 데이터는 다른 핸들러 메서드들의 콜백을 사용자 지정 위임(delegation)을 하는데 사용된다.

함수 객체들은 기존 함수의 시작 또는 끝에 사용자 지정 동작을 삽입하는 것뿐만 아니라 기존 함수를 오버라이드 하는데 사용될 수 있다. 또한 함수 객체들은 정적 메서드 또는 인스턴스 객체를 클래스 정의로써 등록 할 수 있다. 다음은 클래스를 VM 에 인스턴스화 하는 방법에 대한 예제이다.

```
Value networkProto, networkCtorFn, networkConnectFn;
class NetworkClass : public FunctionHandler
{
  public:
    enum Method
    {
        METHOD_Ctor,
        METHOD_Connet,
    };

    virtual ~NetworkClass() {}
    virtual void Call(const Params& params)
    {
        int method = int(params.pUserData);
        switch (method)
        {
        case METHOD_Ctor:
            // Custom logic
            break;
        case METHOD_Connet:
            // Custom logic
            break;
        }
    }
};
 Ptr<NetworkClass> networkClassDef = *SF_HEAP_NEW(Memory::GetGlobalHeap())
                                                NetworkClass();
 // Create the constructor function
 pmovie->CreateFunction(&networkCtorFn, networkClassDef,
                    (void*)NetworkClass::METHOD_Ctor);
 // Create the prototype object
 pmovie->CreateObject(&networkProto);
 // Set the prototype on the constructor function
 networkCtorFn.SetMember("prototype", networkProto);
 // Create the prototype method for 'connect'
 pmovie->CreateFunction(&networkConnectFn, networkClassDef,
                    (void*)NetworkClass::METHOD_Connet);
 networkProto.SetMember("connect", networkConnectFn);
 // Register the constructor function with _global
```

```
pmovie->SetVariable("_global.Network", networkCtorFn);
```

클래스는 이제 VM 에 인스턴스화 되었다.

```
var netObj:Object = new Network(param1, param2);
```

동작을 삽입하는 것은 VM 객체들의 수명에 대한 관리뿐 아니라 VM 에 대한 전문적인 지식을 갖추고 있는 개발자들이 사용하기를 권장한다. 다음은 동작 삽입에 대한 예제이다.

```
Value origFuncReal;
obj.GetMember("funcReal", &origFuncReal);
class FuncRealIntercept : public FunctionHandler
{
   Value OrigFunc;
   public:
          FuncRealIntercept(Value origFunc) : OrigFunc(origFunc) {}
          virtual ~FuncRealIntercept() {}
          virtual void Call(const Params& params)
          {
          // Intercept logic (beginning)
          OrigFunc.Invoke("call", params.pRetVal, params.pArgsWithThisRef,
                          params.ArgCount + 1);
          // Intercept logic (end)
          }
};
Value funcRealIntercept;
 Ptr<FuncRealIntercept> funcRealDef = *SF_HEAP_NEW(Memory::GetGlobalHeap())
                                       FuncRealIntercept(origFuncReal);
pmovie->CreateFunction(&funcRealIntercept, funcRealDef);
obj.SetMember("funcReal", funcRealIntercept);
```

주의: VM 객체에 대한 참조를 유지하는 것 때문에 개발자에 의해 사용자 지정 함수 내에서 컨텍스트 객체의 수명을 유지 해야 한다. 개발자는 .swf(GFx::Movie)가 소멸되기 이전에 이에 대한 참조가 아무것도 남지 않도록 해야 할 필요가 있다. 이 요구사항은 VM 으로 부터 복합 객체를 참조하고 있는 모든 GFx::Value 와 연관된 수명 관리 요구사항과 일치한다.

## *2.4 Direct Access 공통 인터페이스*

다음은 Direct Aceess API 의 public 멤버함수 들이다. GFx::Value 에 관한 최신 정보는 온라인 도움말을 보도록 하자.

### 2.4.1 Object 지원

| **설명** | **Public 메서드**<br>('foo'는 "_root.foo"에 있는 Object 를 참조한다고 가정) |
|---|---|
| 객체에 멤버가 존재하는지 체크 | `bool has = foo.HasMember("bar");` |
| 객체에서 값 얻기 | `Value bar;`<br>`bool = foo.GetMember("bar", &bar);` |
| 객체에 값 설정 | `Value val;`<br>`bool = foo.SetMember("bar", val);` |
| 객체의 메서드 호출 | `Value ret;`<br>`Value args[N];`<br>`bool = foo.Invoke("method", args, N, &ret);` *or*<br>`foo.Invoke("method", args, N);` |
| 객체 생성 | `Value obj;`<br>`pMovie->CreateObject(&obj);`<br>…<br>`bool = foo.SetMember("bar", obj);` |
| 객체 계층구조 생성 | `Value owner, child;`<br>`pMovie->CreateObject(&owner);`<br>`pMovie->CreateObject(&child);`<br>`bool = owner.SetMember("child", child);`<br>…<br>`bool = foo.SetMember("owner", owner);` |
| 객체의 멤버들 반복 | `Value::ObjectVisitor v;`<br>`foo.VisitMembers(&v);` |
| 객체의 멤버 삭제 | `bool = foo.DeleteMember("bar");` |

### 2.4.2 Array 지원

| 설명 | Public 메서드<br>('bar'는" _root.foo.bar"에 있는 Array 를 참조하며 'foo'는 Object 를 참조한다고 가정) |
|---|---|
| 배열 크기 측정 | `unsigned sz = bar.GetArraySize();` |
| 배열에서 요소 얻기 | `Value val;`<br>`bool = bar.GetElement(idx, &val);` |
| 배열에 요소 설정 | `Value val;`<br>`bool = bar.SetElement(idx, val);` |
| 배열크기 재설정 | `bool = bar.SetArraySize(N);` |
| 배열 생성 | `Value arr;`<br>`pMovie->CreateArray(&arr);`<br>…<br>`bool = foo.SetMember("bar", arr);` |
| 요소값 같은 다른 복합객체를 포함하는 배열생성 | `Value owner, childArr, childObj;`<br>`pMovie->CreateArray(&owner);`<br>`pMovie->CreateArray(&childArr);`<br>`pMovie->CreateObject(&childObj);`<br>`bool = owner.SetElement(0, childArr);`<br>`bool = owner.SetElement(1, childObj);`<br>…<br>`bool = foo.SetMember("owner", owner);` |
| 배열의 요소들 범위 내에서 반복 | `Enumeration`<br>`for (UPInt i=0; i <N; i++)`<br>`{ … bool = bar.GetElement(i+idx, &val) … }`<br><br>`Using a visitor pattern:`<br>`Value::ArrayVisitor v;`<br>`bar.VisitElements(v, idx, N);` |
| 배열생성 | `bool = bar.ClearElements();` |
| 스택 연산 | `PushBack`<br>`Value val;`<br>`bool = bar.PushBack(val);`<br>`PopBack`<br>`Value val;`<br>`bool = bar.PopBack(&val);` *or* `void bar.PopBack();` |
| 배열에서 요소 제거 | `Single element`<br>`bool = bar.RemoveElement(idx);`<br>`Series of elements`<br>`bool = bar.RemoveElements(idx, N);` |

### 2.4.3 디스플레이 객체에 대한 지원사항

| 설명 | 공통 메서드<br>('foo'는 디스플레이 객체(MovieClip, TextField, Button)를 참조한다는 것을 가정한다. "_root.foo.bar") |
|---|---|
| 현재의 디스플레이 정보를 얻음. | `Value::DisplayInfo info;`<br>`bool = foo.GetDisplayInfo(&info);` |
| 현재의 디스플레이 정보를 설정. | `Value::DisplayInfo info;`<br>`…`<br>`bool = foo.SetDisplayInfo(info);` |
| 현재의 디스플레이 행렬을 얻음. | `Matrix2_3 mat;`<br>`bool = foo.GetDisplayMatrix(&mat);` |
| 현재의 디스플레이 행결을 설정. | `Matrix2_3 mat;`<br>`…`<br>`bool = foo.SetDisplayMatrix(mat);` |
| 현재의 색상 변환을 얻음. | `Render::Cxform cxform;`<br>`bool = foo.GetColorTransform(&cxform);` |
| 현재의 색상 변환을 설정. | `Render::Cxform cxform;`<br>`…`<br>`bool = foo.SetColorTransform(cxform);` |

### 2.4.4 무비클립에 대한 지원사항

| 설명 | 공통 메서드<br>('foo'가 무비클립을 참조한다는 것을 가정한다. "_root.foo") |
|---|---|
| 무비클립에 심볼의 인스턴스를 붙여 넣음. | `Value newInstance;`<br>`bool = foo.AttachMovie(&newInstance, "SymbolName", "instanceName");` |
| 무비클립의 자식으로 비어있는 무비클립을 생성. | `Value emptyInstance;`<br>`bool = foo.CreateEmptyMovieClip(&emptyInstance, "instanceName");` |
| 이름으로 키프레임 재생. | `bool = foo.GotoAndPlay("myframe");` |
| 숫자로 키프레임 재생. | `bool = foo.GotoAndPlay(3);` |
| 이름으로 키프레임 정지. | `bool = foo.GotoAndStop("myframe");` |
| 숫자로 키프래임 정지 | `bool = foo.GotoAndStop(3);` |

### 2.4.5 텍스트 필드에 대한 지원사항

| 설명 | 공용 메서드<br>('foo'가 텍스트 필드를 참조하는 것을 가정한다. "_root.foo") |
|---|---|
| 텍스트필드의 문자를 얻음. | `Value text;`<br>`bool = foo.GetText(&text);` |
| 텍스트 필드의 문자를 설정. | `Value text;`<br>…<br>`bool = bar.SetText(text);` |
| HTML 텍스트를 얻음. | `Value htmlText;`<br>`bool = bar.GetTextHTML (&htmlText);` |
| HTML 텍스트를 설정. | `Value htmlText;`<br>…<br>`bool = bar.SetTextHTML(htmlText);` |