Autodesk®
Scaleform®

Scaleform Game Communication Overview

This document describes the communication mechanisms between C++, Flash and ActionScript using Scaleform 3.1 and later.

Author: Mustafa Thamer, Prasad Silva
Version: 2.01
Last Edited: September 21, 2010

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.2

How to Contact Autodesk Scaleform:

Document	Scaleform Game Communication Overview
Address	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
Website	www.scaleform.com
Email	info@scaleform.com
Direct	(301) 446-3200
Fax	(301) 446-3199

Table of Contents

1 Interfacing C++, Flash and ActionScript

Flash®'s ActionScript™ scripting language enables creation of interactive movie content. Events such as clicking a button, reaching a certain frame, or loading a movie can execute code to dynamically change movie content, control the flow of the movie, and even launch additional movies. ActionScript is powerful enough to create full mini-games entirely in Flash. Like most programming languages, ActionScript supports variables and subroutines and contains objects which can represent items or controls.

Communicating between the application and the Flash content is required for complex use cases. Autodesk Scaleform® supports the standard mechanisms provided by Flash that enables ActionScript to pass events and data back to the C++ application. It also provides a convenient C++ interface to directly manipulate ActionScript variables, arrays, and objects, as well as directly invoke ActionScript subroutines.

In this document we discuss the different mechanisms that are available when communicating between C++ and Flash. The following are the available options:

ActionScript → C++	**C++ → ActionScript**
FSCommand Simple, string-based function execution, no return values, deprecated	**GFx::Movie::Get/SetVariable** Access data in ActionScript, uses string path
ExternalInterface Flexible argument handling, supports return values, recommended	**GFx::Movie::Invoke** Call functions in ActionScript, uses string path
Direct Access API Uses GFx::Value as a direct reference to objects for data and function access, high performance. Set direct function callbacks in the ActionScript VM using GFx::FunctionHandler.	

1.1 ActionScript to C++

Scaleform provides two mechanisms for the C++ application to receive events from ActionScript: *FSCommand* and *ExternalInterface*. These mechanisms are part of the standard ActionScript API and more information on them can be found in the Flash documentation. Both FSCommand and ExternalInterface register a C++ event handler with Scaleform to receive event notification. These handlers are registered as Scaleform states and therefore can be registered on a GFx::Loader, GFx::MovieDef or GFx::Movie depending on functional requirements. For information about GFx states, please refer to the Scaleform documentation.

FSCommand events are triggered by the ActionScript 'fscommand' function. The fscommand function can only pass two string arguments from ActionScript, one for a command and one for a single data argument. These string values are received by the C++ handler as two const char pointers. The C++ fscommand handler unfortunately cannot return a value back as the ActionScript fscommand function has no return value support.

ExternalInterface events are triggered when ActionScript calls the *flash.external.ExternalInterface.call* function. This interface can pass an arbitrary list of ActionScript values in addition to a command name. The C++ ExternalInterface handler receives a const char pointer for the command name and an array of GFx::Value arguments corresponding to the ActionScript values passed from the runtime. The C++ ExternalInterface handler can also return a GFx::Value back to the runtime as the ActionScript ExternalInterface.call method supports a return value. The ActionScript ExternalInterface class is part of the External API, an application programming interface that enables straightforward communication between ActionScript and the Flash Player container, in this case the application using Scaleform.

Due to limited flexibility, FSCommands are made obsolete by ExternalInterface and are no longer recommended for use. They are described here for completeness and legacy support.

To an extent, both FSCommands and ExternalInterface are made obsolete by the Direct Access API and the GFx::FunctionHandler interface. This interface allows direct callbacks to C++ methods be assigned inside the ActionScript VM as normal functions. When the function is invoked in ActionScript, it will in turn invoke the C++ callback. For more information on GFx::FunctionHandler, please see the Direct Access API section.

1.1.1 FSCommand Callbacks

The ActionScript fscommand function passes a command and a data argument to the host application. The following is a typical usage in ActionScript:

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

Any non-string arguments to fscommand, such as boolean or integers will be converted to strings.

An ActionScript fscommand call passes two strings to the GFx FSCommand handler. An application registers a fscommand handler by sub-classing GFx::FSCommandHandler and registering an instance of the class as a shared state with either the GFx::Loader, GFx::MovieDef or with individual GFx::Movie objects. Keep in mind that the settings between state bags are intended to be delegated as follows: GFx::Loader -> GFx::MovieDef -> GFx::Movie. They can be overridden in any one of those later instances. This means that if you want a particular GFx::Movie to have its own GFx::RenderConfig (for separate EdgeAA control, for example) or GFx::Translator (so that it is translated to a different language), you can set it on that object and whatever states you apply there will take precedence over states applied on the object higher in the delegation chain, such as GFx::Loader. If a command handler is set on a GFx::Movie, it will receive callbacks for only the FSCommand calls invoked in that movie instance. The GFxPlayerTiny example demonstrates this process (search for “FxPlayerFSCommandHandler”).

The following is an example of fscommand handler setup. The handler is derived from GFx::FSCommandHandler.

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

The Callback method receives the two string arguments passed to fscommand in ActionScript as well as a pointer to the specific movie instance that invoked fscommand. Our custom handler simply prints each fscommand event to the debug console.

Next, register the handler after creating the GFx::Loader object:

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

Registering the handler with the GFx::Loader causes every GFx::Movie and GFx::MovieDef to inherit this handler. SetFSCommandHandler can be called on individual movie instances to override this default setting.

In general, C++ event handlers should be non-blocking and return to the caller as soon as possible. Event handlers are typically only called during Advance or Invoke calls.

1.1.2 The External Interface API

The Flash ExternalInterface.call method is similar to fscommand but is preferred because it provides more flexible argument handling and can return values. Registering an ExternalInterface handler is similar to registering an fscommand handler. Here is an example ExternalInterface handler that prints out number and string arguments.

```
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);
    }
};
```

Once the handler is implemented, an instance of it can then be registered with GFx::Loader as shown below:

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

Now the callback handler will be triggered by an ExternalInterface call in ActionScript.

1.1.2.1 FxDelegate

The ExternalInterface handler above provides very basic callback functionality. In practice, an application might like to register specific functions to be called whenever an ExternalInterface callback is triggered with a certain methodName. This requires a more advanced callback system which allows functions to be registered with a corresponding methodName, and then uses a hash table (or similar) to look up and execute them with the corresponding GFx::Value arguments. Such a system is provided in our samples and is called FxDelegate (see Apps\Samples\GameDelegate\FxGameDelegate.h).

FxDelegate derives from GFx::ExternalInterface, as expected. It has functions Register/Unregister Handlers which allows an app to install its own callback handlers as delegates, registered by a methodName.

For an example of FxDelegate usage, please take a look at our HUD Kit which utilizes FxDelegate for its minimap implementation. Here is the application code from the minimap demo which registers it's callback functions with FxDelegate:

```
// *** 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 is fully functional and is provided as an example of a more sophisticated callback handling and registration system. It is meant to be a starting point for the user and we encourage developers to look at the details of how it works and customize and extend it for their own needs.

1.1.2.2 Using External Interface from ActionScript

In ActionScript, an External Interface call would be initiated with the following code:

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

where ‘foo’ is the method name and ‘arg’ is an argument of basic type. You can specify 0 or more parameters, separated by commas. Please refer to the Flash documentation for more information about the ExternalInterface API.

If ExternalInterface has not been imported as in the code above, the ExternalInterface calls must be fully qualified:

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

1.1.2.3 ExternalInterface.addCallback

The ExternalInterface.addCallback function globally registers an ActionScript method under an alias. This allows registered methods to be invoked from C++ without a path prefix. It supports registering the method and the calling context ("this" objects) under a single alias. The registered alias can be called by the C++ GFx::Movie::Invoke() method.

Example:

AS Code:

```
import flash.external.*;

var txtField:TextField = this.createTextField("txtField",
                                    this.getNextHighestDepth(), 0, 0, 200, 50);

varaliasName: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!";
}
```

Now it is possible to call the SetText ActionScript function with "this" set to txtField by invoking the alias:

C++ Code:

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

The result will trace "txtField.SetText" and set the text "INVOKED!" to the "txtField" text field. See section 1.2.2 for more information on using GFx::Movie::Invoke().

1.2 C++ to ActionScript

The previous section explained how ActionScript can call into C++. This section describes how to communicate in the opposite direction, using Scaleform functions that enable the C++ program to communicate with a Flash movie. Scaleform supports C++ functions to directly get and set ActionScript variables (simple types, complex types and arrays) as well as invoke ActionScript subroutines.

The Direct Access API provides a more convenient and efficient interface to access and manipulate ActionScript variables from C++. For more information on this interface, please see the Direct Access API section.

1.2.1 Manipulating ActionScript Variables

Scaleform supports GetVariable and SetVariable, which enable direct manipulation of ActionScript variables. For performance critical use cases, please refer to the section 2 on the Direct Access API for a more efficient way to modify variables and object properties. Although Set/GetVariable are not recommended for performance reasons, there are some cases where they are more convenient to use than the Direct Access API. For instance, there's no need to use the Direct Access call GFx::Value::SetMember to set a single value once during the lifetime of the application, especially if the target is at a deep nesting level. Also, obtaining references to an ActionScript object is usually done via GetVariable – it is called once to get a GFx::Value reference which is then used for Direct Access operations.

The following example demonstrates incrementing an ActionScript counter using GetVariable and SetVariable:

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

GetVariableDouble returns the value of the _root.counter variable automatically converted to the C++ Double type. Initially, the variable does not exist and GetVariableDouble returns zero. The counter is then incremented on the next line and the new value is saved to _root.counter using SetVariable. The online documentation for GFx::Movie lists the different variations of GetVariable and SetVariable.

SetVariable has an optional third argument of type GFx::Movie::SetVarType that declares the assignment "sticky." This is useful when the variable being assigned has not yet been created on the Flash timeline. For example, suppose that the text field _root.mytextfield is not created until frame 3 of the movie. If SetVariable("_root.mytextfield.text", "testing", SV_Normal) is called on frame 1, right after

the movie is created, then the assignment would have no effect. However, if the call is made with SV_Sticky (the default value) then the request is queued up and applied once the _root.mytextfield.text value becomes valid on frame 3. This makes it easier to initialize movies from C++. Keep in mind that, generally, SV_Normal is more efficient than SV_Sticky, so SV_Normal should be used where possible.

To access arrays of data, Scaleform provides the functions SetVariableArray and GetVariableArray.

SetVariableArray sets array elements in specified range to data items of specified type. If the array does not exist, it is created. If an array already exists but does not contain enough items, it is resized appropriately. However, setting a number of elements less than the current size of the array will not cause the array to resize. GFx::Movie::SetVariableArraySize sets the size of the array, which is useful when setting an existing array with fewer elements than it had before.

The following example demonstrates the use of SetVariableArray to set an array of strings in AS:

```
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);
```

The following is another variant of the previous example, using wide-strings:

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

The GetVariableArray method fills the provided data buffer with results from an AS array. The buffer must be big enough to hold the number of items requested.

1.2.2 Executing ActionScript Subroutines

In addition to modifying ActionScript variables, ActionScript methods can be invoked using the GFx::Movie::Invoke() method. This is useful for performing more complicated processing, triggering animation, changing the current frame, programmatically changing the state of UI controls, and dynamically creating UI content such as new buttons or text. For performance critical use cases, please refer to the section on the Direct Access API for a more efficient way to invoke methods and control action flow.

Example:
The following ActionScript function can be used to set the slider grip position relative to the range.

Assume that the ‘mySlider’ object exists in the _root level. The SetSliderPos is defined inside the "mySlider" object as follows:

AS Code:

```
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;
}
```

where the "gripClip" is a nested movie clip in "mySlider" and pos always lives within the rangeMin/Max.

In the user application, the Invoke function is used as follows:

C++ Code:

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

Invoke returns true if the method was actually invoked or false otherwise.

When invoking an ActionScript function, you must make sure that it is loaded. One common error in using Invoke is calling an ActionScript routine that is not yet available, in which case an error will be printed to the Scaleform log. An ActionScript routine will not become available until the frame it is associated with has been played or the nested object it is associated with has been loaded. All ActionScript code in frame 1 will be available as soon as the first call to GFx::Movie::Advance is made, or if GFx::MovieDef::CreateInstance is called with initFirstFrame set to true.

Along with Invoke, the GFx::Movie::IsAvailable() method is typically used to ensure the AS function exists before it is invoked:

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

This example used the 'printf' style of Invoke. In this case Invoke takes arguments as a variable argument list described by a format string. Other versions of the function use GFx::Value to efficiently process non-string arguments. For example:

```
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 is identical to Invoke except that it takes a va_list argument to enable the application to supply a pointer to a variable argument list. The relationship between Invoke and InvokeArgs is similar to the relationship between printf and vprintf.

1.2.3 Paths

When accessing Flash elements from within the user application, fully qualified paths are often used to lookup objects.

The following GFx::Movie functions rely on fully qualified paths:

```
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, ...)
```

In order for nested object paths to resolve correctly, all parent movie clips must have unique **instance** names. Nested object names are separated using the dot "." and are **case sensitive**, as shown below.

In the following example, consider a movie clip with the instance name "clip2" nested inside another movie clip named "clip1", which is sitting on the main stage.

Main Stage -> clip1 -> clip2

1. clip1 is within/or on the main stage
2. clip2 is within clip1

Valid paths are:

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

Invalid paths are:
`"Clip1.clip2"` <- case mismatch - "Clip1" must be lowercase
`"clip2"` <- missing parent"clip1." - path name.

The `"_root"` and `"_level0"` names are optional in ActionScript 2 and can be used to force specific base level look up. However, they are required in ActionScript 3 as by default the lookup will occur at the stage level. For backward compatibility we have provided aliases in ActionScript 3 to access root with the following names: _root, _level0, root, level0. When loading separate SWF files into levels, `_root` will refer to the base of the current level, while specifying a `_levelN` (using the syntax shown above) allows you to select a specific level.

NOTES:

- You can check paths to objects and variables in Flash Studio's Test Movie (Ctrl-Enter) environment, by pressing (**Ctrl-Alt-V)** (variables).

- The timeline sequence in Flash Studio, beginning with Scene 1 link, does NOT dictate the target path. Certain elements are listed which are not actually used in the object path. Use (Ctrl-Alt-V) (variables) popup to check the actual valid paths.

- When loading movies within movies, using the loadMovie command, you may encounter problems due to objects changing levels. Objects which did exist on _level0 are now on _level1 or _level2 (depending on what Level you loaded to), or within a targeted Movie Clip. To reference an Object on another layer, simply use: `_levelN.objectName` or `_levelN.objectPath.objectName`, where N is the Level number.

Many elements of this section were adopted from the following two path tutorials, which we recommend reading:

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

2 The Direct Access API

The GFx::Value Direct Access support included in Scaleform 3.1 and higher versions is a major improvement to the way a game communicates with the AS runtime. The ActionScript communication API has been extended beyond simple types so that complex objects (and individual members within those objects) can be set and queried efficiently. For example, nested and heterogeneous data structures can now be passed back and forth from the UI to the game. For an in-depth example of using Direct Access API, please see our HUD Kit, which demonstrates the performance gains found when updating large numbers of Flash objects on a minimap.

With the Direct Access API, GFx::Values, can now store simple ActionScript types as well as references to Objects, Arrays and Display Objects. Display Objects are a special case of the Object type and correspond to entities on the stage (MovieClips, Buttons, and TextFields). The GFx::Value class API has a full set of functions for setting and getting their values and members and dealing with arrays. The GFx::Movie class API now includes methods to create objects and arrays. Please see the GFx::Value reference for the details.

The Direct Access API allows the application to bind a C++ variable (of type GFx::Value) directly to an object in ActionScript. Once this binding or reference is made, the variable can be used to easily and efficiently modify the ActionScript object. Prior to this change, users had to access AS objects through the GFx::Movie and by specifying a string path. This method incurred a performance penalty for parsing the path and finding the AS object. With direct access objects, this costly parsing and searching overhead on each call has been removed.

Many operations which were previously available only at the GFx::Movie level can now be applied directly to an ActionScript (AS) object, represented by the GFx::Value type. This results in much cleaner and more efficient code. For example, to call a method on an object called ‘foo’ located at the root, one could get a reference to the object and call Invoke on it directly, rather than use the Movie’s Invoke call.

1. GFx::Movie Invoke Method (less efficient):

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

2. Direct Access Invoke Method (better):

```
(Assumes 'foo' holds a reference to an AS object at "_root.foo")
Value ret;
Value args[N];
bool = foo.Invoke("method", &ret, args, N);
```

Along with Invoke, calls to check, get and set values on AS objects can now be done through the Direct Access API, using GFx::Value::HasMember, GetMember and SetMember.

Example using GetMember to update a text field on a movieClip (stored in a GFx::Value):

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

In the example above, we first get the textfield member of the movieClip object and then use the function SetText to update its text value.

2.1 Objects and Arrays

Another important capability supported by the Direct Access API is the ability to create an AS object or hierarchy of objects. In this way, objects of arbitrary depth and structure can be created and managed in C++. For example, the following code snippet could be used in order to create two objects in a hierarchy and have one be considered the child of the other:

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

CreateObject creates an instance of an ActionScript object. It also accepts an optional, fully qualified class name to be passed in, if an instance of a specific class type is required. For example:

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

Consider a more complicated situation that uses both Objects and Arrays. The following creates an array with complex object elements:

```
// (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
```

The function GFx::Value::VisitMembers() can be used to traverse the public members of an object. The function takes an instance of an ObjectVisitor class which has a simple Visit callback that must be overridden. This function is only valid for Object types (including Array and DisplayObject). *Note that you cannot use VisitMembers to introspect a class instance completely.* Methods are not enumerable because they live in the prototype. To force the visibility of members and properties, use the ASSetPropFlags method in ActionScript:

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

Properties (getter/setter) are never enumerable via VisitMembers, even with ASSetPropFlags. However, they are accessible via the Direct Access interface and returned as VT_Object. Functions are also returned as VT_Object (if they are made enumerable via ASSetPropFlags).

2.2 Display Objects

The Direct Access API exposes a special case of the Object type, called DisplayObject, which corresponds to entities on the stage, such as MovieClips, Buttons and TextFields. A custom DisplayInfo API is provided to access their display properties. Using the DisplayInfo API, you can easily set properties on a DisplayObject such as alpha, rotation, visibility, position, offset and scale. Although these display properties could also be set using the SetMember function, the SetDisplayInfo call is the fastest way to do this, since it directly modifies the object's display properties. Note that both of these methods are faster than calling GFx::Movie::SetVariable which does not operate directly on the target object.

The following code utilizes the SetDisplayInfo method to change the position and rotation of a movieclip instance:

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

2.3 Function Objects

Functions in the ActionScript2 Virtual Machine (AS2 VM) are similar to basic objects. These function objects can be assigned members, which could provide extra introspection information depending on the use case. With the GFx::Movie::CreateFunction method, developers are able to create function objects that wrap C++ callback objects. The function object returned by CreateFunction can then be assigned as a member of any object in the VM. When this AS function is invoked, the C++ callback is invoked in turn. This ability allows developers to register direct callbacks from the VM to their own callback handlers without the additional overhead of delegation (via fscommand or ExternalInterface).

The following example creates a custom callback and assigns it to a member of an AS object:

```
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);
```

This method can be invoked in ActionScript (in the _root timeline):

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

The Params structure passed to the Call() method will contain the following:

- `pRetVal`: Pointer to a GFx::Value that will be passed back to the caller as the return value. If passing back a complex object, then pass this pointer to pMovie->CreateObject() or CreateArray() to create the container. For primitive types (Numbers, Booleans, etc.), call the appropriate setter.
- `pMovie`: Pointer to the movie that invoked the function.
- `pThis`: The caller context. May be undefined if calling context is non-existent or invalid.
- `ArgCount`: Number of arguments passed to the callback.
- `pArgs`: Array of GFx::Values representing the arguments passed to the function callback. Access the individual aguments by using the [ ] operator.
  E.g.: `GFx::Value firstArg = params.pArgs[0];`

- `pArgsWithThisRef`: Array of arguments plus the calling context pre-pended. This is used for chaining other function objects (see below for example on injecting custom behavior).
- `pUserData`: Custom data set when the function object was registered. This data is useful for custom delegation of callbacks to different handler methods.

Function objects can be used to override existing functions in the VM as well as to inject custom behavior at the beginning or end of an existing function body. Function objects can also be register static methods or instance methods with a class definition. This can be extended to define a custom class. The following example creates a sample class that can be instantiated in the VM:

```
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);
```

The class can now be instantiated in the VM:

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

Injecting behavior is intended for developers with expert knowledge of the VM as well as lifetime management of VM objects. The following example demonstrates behavior injection:

```
    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);
```

NOTE: The lifetime of the GFx::Value held inside the custom function context object must be maintained by the developer because it holds a reference to a VM object. The developer is required to clear the reference before the .swf (GFx::Movie) dies. This requirement is consistent with the lifetime management requirement associated with all GFx::Values that hold references to complex objects from the VM.

2.4 The Direct Access Public Interface

Here are the public member functions in the Direct Access API. Please see the online documentation on GFx::Value for the latest information.

2.4.1 Object Support

Description	**Public Method** (Assumes 'foo' holds a reference to an Object at "_root.foo")
Check whether member exists in Object	`bool has = foo.HasMember("bar");`
Retrieve a value from an Object	`Value bar;` `bool = foo.GetMember("bar", &bar);`
Set a value of an Object	`Value val;` `bool = foo.SetMember("bar", val);`

Call a method of an Object	`Value ret;`
	`Value args[N];`
	`bool = foo.Invoke("method", args, N, &ret);` *or*
	`foo.Invoke("method", args, N);`

Create an Object	`Value obj;`
	`pMovie->CreateObject(&obj);`
	…
	`bool = foo.SetMember("bar", obj);`

Create an Object hierarchy	`Value owner, child;`
	`pMovie->CreateObject(&owner);`
	`pMovie->CreateObject(&child);`
	`bool = owner.SetMember("child", child);`
	…
	`bool = foo.SetMember("owner", owner);`

Iterate over members of an Object	`Value::ObjectVisitor v;` `foo.VisitMembers(&v);`
Delete a member from an Object	`bool = foo.DeleteMember("bar");`

2.4.2 Array Support

Description	Public Method (Assumes 'bar' holds a reference to an Array at "_root.foo.bar" and 'foo' holds a reference to a Object)
Determine array size	`unsigned sz = bar.GetArraySize();`
Retrieve an element from an Array	`Value val;` `bool = bar.GetElement(idx, &val);`
Set an element of an Array	`Value val;` `bool = bar.SetElement(idx, val);`
Resize an Array	`bool = bar.SetArraySize(N);`

Create an Array	`Value arr;`
	`pMovie->CreateArray(&arr);`
	…
	`bool = foo.SetMember("bar", arr);`

Create an Array containing other complex objects as elements	`Value owner, childArr, childObj;`
	`pMovie->CreateArray(&owner);`
	`pMovie->CreateArray(&childArr);`
	`pMovie->CreateObject(&childObj);`
	`bool = owner.SetElement(0, childArr);`
	`bool = owner.SetElement(1, childObj);`
	…
	`bool = foo.SetMember("owner", owner);`

Iterate over a range of elements in an Array	`Enumeration`
	`for (UPInt i=0; i <N; i++)`
	`{ … bool = bar.GetElement(i+idx, &val) … }`
	
	`Using a visitor pattern:`
	`Value::ArrayVisitor v;`
	`bar.VisitElements(v, idx, N);`

Clear an Array	`bool = bar.ClearElements();`

Stack Operations	`PushBack`
	`Value val;`
	`bool = bar.PushBack(val);`
	
	`PopBack`
	`Value val;`
	`bool = bar.PopBack(&val);` *or* `void bar.PopBack();`

Remove elements from an Array	`Single element`
	`bool = bar.RemoveElement(idx);`
	
	`Series of elements`
	`bool = bar.RemoveElements(idx, N);`

2.4.3 Display Object Support

Description	Public Method (Assumes 'foo' holds a reference to a display object (MovieClip, TextField, Button) at "_root.foo.bar")
Get the current display info	`Value::DisplayInfo info;` `bool = foo.GetDisplayInfo(&info);`

Set the current display info	`Value::DisplayInfo info;`
	…
	`bool = foo.SetDisplayInfo(info);`

Get the current display matrix	`Matrix2_3 mat;` `bool = foo.GetDisplayMatrix(&mat);`

Set the current display matrix	`Matrix2_3 mat;`
	…
	`bool = foo.SetDisplayMatrix(mat);`

Get the current color transform	`Render::Cxform cxform;` `bool = foo.GetColorTransform(&cxform);`

Set the current color transform	`Render::Cxform cxform;`
	…
	`bool = foo.SetColorTransform(cxform);`

2.4.4 MovieClip Support

Description	Public Method (Assumes 'foo' holds a reference to a MovieClip at "_root.foo")
Attach a symbol instance to the MovieClip	`Value newInstance;` `bool = foo.AttachMovie(&newInstance, "SymbolName", "instanceName");`
Create an empty movieclip as a child of the MovieClip	`Value emptyInstance;` `bool = foo.CreateEmptyMovieClip(&emptyInstance, "instanceName");`
Play a keyframe by name	`bool = foo.GotoAndPlay("myframe");`
Play a keyframe by number	`bool = foo.GotoAndPlay(3);`
Stop at a keyframe by name	`bool = foo.GotoAndStop("myframe");`
Stop at a keyframe by number	`bool = foo.GotoAndStop(3);`

2.4.5 Textfield Support

Description	Public Method (Assumes 'foo' holds a reference to a Textfield at "_root.foo")
Get the raw textField text	`Value text;` `bool = foo.GetText(&text);`

Set the raw textField text	`Value text;`
	…
	`bool = bar.SetText(text);`

Get the HTML text	`Value htmlText;` `bool = bar.GetTextHTML (&htmlText);`

Set the HTML text	`Value htmlText;`
	…
	`bool = bar.SetTextHTML(htmlText);`