2. Everyday bytes

Copying objects 1
Serializing/deserializing custom objects 3
Embedding resources 5
Injecting bytes 6
Progressive image loading 7
Loadbytes limitations 10
Compressing and uncompressing data 11
Generating an image file (PNG or JPEG) 15
Saving a binary stream (through a remote server) 20
Saving a binary stream (without a remote server) 22
Generate a PDF 22
From bytes to sound 27
From compressed bytes to sound 36
Parsing binary data 38
The SWF example 38
Advanced SWF parsing 51

In the previous chapter we discovered the important concepts to manipulate binary data in the Flash Player. It is now time to put all those concepts into practice to use them in real world projects. From binary asset embedding to sound manipulation to SWF parsing, we will see how to make this `ByteArray` API our best friend, to produce things we could never do before in ActionScript.

We will continue our ByteArray adventure through different examples you may come through during your every day work with Flash.

Copying objects

One of the very common things that the ByteArray class is used for, is object duplication. Remember that an AMF serializer and deserializer is available through the ByteArray API. The `writeObject` API is what we need :

```
// creates an empty ByteArray
var stream:ByteArray = new ByteArray();

// creates an object
var parameters:Object = { age : 25, name : "Bob" };

// serializes the object as amf and stores it into the ByteArray
stream.writeObject( parameters );
```

To retrieve the serialized instance we use the `readObject` API :

```
// creates an empty ByteArray
var stream:ByteArray = new ByteArray();

// creates an object
```

```
var parameters:Object = { age : 25, name : "Bob" };

// serializes the object as amf and stores it into the ByteArray
stream.writeObject( parameters );

// resets the position
stream.position = 0;

// reads the object copy
var objectCopy:Object = stream.readObject();
```

We can then include this logic inside a custom function :

```
function copyObject ( objectToCopy:* ):*
{
    var stream:ByteArray = new ByteArray();

    stream.writeObject( objectToCopy );

    stream.position = 0;

    return stream.readObject();
}
```

Anytime, we need to copy we can use our `copyObject` function :

```
// creates an object
var parameters:Object = { age : 25, name : "Bob" };

var parametersCopy:Object = copyObject ( parameters );

/* outputs :
name :  Bob
age  :  25
*/
for ( var p:String in parametersCopy )
    trace( p, " : ", parametersCopy[p] );

function copyObject ( objectToCopy:* ):*
{
    var stream:ByteArray = new ByteArray();

    stream.writeObject( objectToCopy );

    stream.position = 0;

    return stream.readObject();
}
```

By modifying the original object properties, we can see that we created a proper copy of `parameters` :

```
// creates an object
var parameters:Object = { age : 25, name : "Bob" };

var parametersCopy:Object = copyObject ( parameters );

parameters.name = "Stevie";

/* outputs :
name :  Bob
age  :  25
*/
for ( var p:String in parametersCopy )
    trace( p, " : ", parametersCopy[p] );

function copyObject ( objectToCopy:* ):*
{
    var stream:ByteArray = new ByteArray();

    stream.writeObject( objectToCopy );

    stream.position = 0;

    return stream.readObject();
}
```

Let's dig further and discover how we can save and restore more complex types, still with the help of our favorite API, `ByteArray`.

Serializing/deserializing custom objects

Note that this code will not work for custom types that you may define in your application. Very simple, scenario, let's say you need to use a `User` object as following :

```
package
{
    public class User
    {
         private var _firstName:String;
         private var _lastName:String;

         public function set firstName (firstName:String):void
         {
              _firstName = firstName;
         }

         public function set lastName (lastName:String):void
         {
              _lastName = lastName;
         }

         public function get firstName ():String
         {
              return _firstName;
         }

         public function get lastName ():String
         {
              return _lastName;
         }
    }
}
```

You may need to store your `User` instance on a server as a binary file or in a local `SharedObject`. If you try to store such a `User` instance inside your byte array and retrieve it later, when reading it, the Flash Player will look up internally if any `User` type has been registered before, if not it will be deserialized as a simple `Object` :

```
// creates an instance of User
var myUser:User = new User ();

// sets the members
myUser.firstName = "Stevie";
myUser.lastName = "Wonder";

// outputs :[object User]
trace ( myUser );

// create a ByteArray to store the instance
var bytes:ByteArray = new ByteArray();

// stores the instance
```

```
bytes.writeObject ( myUser );

// resets the position
bytes.position = 0;

// outputs : false
trace ( bytes.readObject() is User );
```

Now, let's use the `registerClassAlias` to inform the Flash Player to register the `User` type for automatic deserialization :

```
// creates an instance of User
var myUser:User = new User ();

// sets the members
myUser.firstName = "Stevie";
myUser.lastName = "Wonder";

// outputs :[object User]
trace ( myUser );

// registers the type User for deserialization
registerClassAlias ( "userTypeAlias", User );

// create a ByteArray to store the instance
var bytes:ByteArray = new ByteArray();

// stores the instance
bytes.writeObject ( myUser );

// resets the position
bytes.position = 0;

// reads the stored instance and automatically deserializes it to the User type
var storedUser:User = bytes.readObject() as User;

// outputs : true
trace ( storedUser is User );

// outputs : Stevie Wonder
trace ( storedUser.firstName, storedUser.lastName );
```

By using this technique we can store the state of any custom object type and restore it later. As you can see we are now dealing with a copy of our custom object type :

```
storedUser.firstName = "Bobby";
storedUser.lastName = "Womack";

// outputs : Stevie Wonder
trace ( myUser.firstName, myUser.lastName );

// outputs : Bobby Womack
trace ( storedUser.firstName, storedUser.lastName );
```

Keep in mind that some native types cannot be serialized/deserialized by AMF. This is the case for the graphical objects like `DisplayObject`, etc. So if you try to serialize a `MovieClip` for instance, (which would be extremeltey powerful) this will fail silently :

```
// creates a ByteArray
var bytes:ByteArray = new ByteArray();

// try storing a DisplayObject type
bytes.writeObject ( new MovieClip() );

// resets the position
bytes.position = 0;
```

```
// outputs : undefined
trace ( bytes.readObject() );
```

Support for display objects type by the AMF3 would be an extremely useful addition to the list of types supported by AMF.

Embedding resources

Another classic stuff that you may find useful is embedding external resources like XML files or Pixel Bender Kernel filters. A common problem is embedding runtime dependencies into the SWF so that it does not require any external file. Some hosting services refuse SWF's having external dependencies, so imagine you developed a tiny application or game and you quickly need to remove any external dependencies at the last minute, you can use the `Embed` tag for that.

In the code below, we use the `Embed` tag to embed an external Pixel Bender filter :

```
[Embed(source="myFilter.pbj", mimeType="application/octet-stream")]
var myShaderKernel:Class;
```

At compile time, the Pixel Bender filter will be Embedded as a `ByteArray`, notice the usage of the `mimeType application/octet-stream`, which allows us to embed the resource as a ByteArray.

We can also do that more commonly with XML to embed the XML stream without changing the source code of just pasting the XML stream inside the source code, this will be done at compile time and will be cleaner :

```
import flash.utils.ByteArray;

[Embed(source="test.xml", mimeType="application/octet-stream")]
var xmlStream:Class;

// instanciate the stream as a ByteArray
var xmlBytes:ByteArray = new xmlStream();

// read the XML String from the byte stream
var xmlString:String = xmlBytes.readUTFBytes( xmlBytes.bytesAvailable );

// instanciate a XML object by passing the content
var myXML:XML = new XML(xmlString);

/*
outputs :
<menu>
  <item/>
  <item/>
  <item/>
</menu>
*/
trace ( myXML );

// outputs : 3
trace ( myXML.item.length() );
```

Everything can be embedded this way, we are embedding here raw binary data that we can manipulate later with the `ByteArray` API.

But, I admit it, the previous code can be a little verbose, so by using a specific `mimeType` you can force the transcoder to map to a specific type and do the conversion for you. In the following code, we embed the XML the same way but by using the correct `mimeType` and the code becomes even simpler :

```
import flash.utils.ByteArray;

[Embed(source="test.xml", mimeType="text/xml")]
var xmlStream:Class;

// instanciate a XML object by passing the content
var myXML:XML = new XML (xmlStream.data);

/*
outputs :
<menu>
  <item/>
  <item/>
  <item/>
</menu>
*/
trace ( myXML );

// outputs : 3
trace ( myXML.item.length() );
```

Prtetty convenient. Let’s see now how we can inject bytes at runtime and reconstruct content from bytes. The Adobe Flash Player offers a very powerful API for this that we are going to cover now.

Injecting bytes

More commonly, this can be done with images, fonts, or even a SWF. There is no SWF type inside the Flash Player, but guess what, there is a `loadBytes` API on the `Loader` object.

This will allows us to inject the SWF as a `ByteArray` inside a `Loader` object and run it :

```
import flash.utils.ByteArray;
import flash.display.Loader;

[Embed(source="test-loading.swf", mimeType="application/octet-stream")]
var swfStream:Class;

// instanciate the stream as a ByteArray
var swfBytes:ByteArray = new swfStream();

// instanciate a Loader
var myLoader:Loader = new Loader();

// inject the embedded stream inside the Loader
// the SWF is executed automatically
myLoader.loadBytes(swfBytes);
```

To display it, we just need to add the `Loader` to the display list :

```
import flash.utils.ByteArray;
import flash.display.Loader;

[Embed(source="test-loading.swf", mimeType="application/octet-stream")]
var swfStream:Class;

// instanciate the stream as a ByteArray
var swfBytes:ByteArray = new swfStream();

// instanciate a Loader
var myLoader:Loader = new Loader();

// show the Loader object
addChild ( myLoader );

// inject the embedded stream inside the Loader
// the SWF is executed automatically
```

```
myLoader.loadBytes(swfBytes);
```

For types like images or sounds, you do not even have to specify the `mimeType`, the transcoder automatically maps the stream to the correct type without any tricks :

```
import flash.utils.ByteArray;
import flash.display.Bitmap;

[Embed(source="ferrari-demo.png")]
var myImageStream:Class;

// instanciate the image as a classic Bitmap instance
var myImage:Bitmap = new myImageStream();

// show it!
addChild ( myImage );
```

Let's play a little more with images, in the following section we are going to work with dynamic image loading, but in a new way.

Progressive image loading

Another great API related to bytes in the Flash Player is called `URLStream`, this API allows us to download content in the Flash Player and get access to the bytes as they are loaded. Think about a `Socket` API grabbing automatically content as if they were pushed by a remote server.

What is great about this is that almost all the APIs available on `ByteArray` are also available on `URLStream`. So what is the use case where you may need this API ? Well, first, let's try the following code, it may give you some cool ideas immediately :

```
// creates a URLStream object
var stream:URLStream = new URLStream();


// listen to the ProgressEvent.PROGRESS event to grab the incoming bytes
stream.addEventListener ( ProgressEvent.PROGRESS, onProgress );
stream.addEventListener ( Event.COMPLETE, onComplete );

// download the google.com index page
stream.load ( new URLRequest ("http://www.google.com" ) );

function onProgress ( e:ProgressEvent ):void
{
    trace ("progress");
}

function onComplete ( e:Event ):void
{
    trace ("complete");
}
```

We see that the `ProgressEvent.PROGRESS` is dispatched multiple times, and the `Event.COMPLETE` event once at the end. Nothing fancy here right ?

But you can actually have access to the bytes as they are coming in, whereas `Loader` just gives you access to the file once loaded. All you can do during the progress with such an object, is getting information on the total bytes to load and currently loaded.

Let's modify our previous code with the following change :

```
function onProgress ( e:ProgressEvent ):void
{
    trace ( stream.readUTFBytes ( stream.bytesAvailable ) );
}
```

We are now evaluating the bytes as a string as they are coming in. You should get in the output window the raw HTML content from the google page. Something like the following output (truncated for obvious purpose) :

```
<!doctype html><html><head><meta http-equiv="content-type" content="text/html; charset=ISO-8859-
1"><title>Google</title><script>window.google={kEI:"xTp8TZq4HpPWtQOd4Kz4Ag",kEXPI:"28479,28501,28595,290
35,29265,29279",kCSI:{e:"28479,28501,28595,29014,29135,29265,29279",ei:"xTp8TZq4HpPWtQOd4Kz4Ag",expi:"28
501,28595,29014,29135,29265,29279"},ml:function(){},kHL:"en",time:function(){return(new
Date).getTime()},log:function(c,d,
b){var a=new Image,e=google,g=e.lc,f=e.li;a.onerror=(a.onload=(a.onabort=function(){delete
g[f]}));g[f]=a;b=b||"/gen_204?atyp=i&ct="+c+"&cad="+d+"&zx="+google.time();a.src=b;e.li=f+1},lc:[],li:0,'
lt:{}…
```

That means we progressively have access to the bytes. This is an interesting feature cause this would allow us for instance, to retrieve the dimensions of an image before being completely loaded by parsing the header as soon as it is available.

We can also create a progressive loader, that could be used to show progressively an image being loaded. To reproduce that effect, we can write the following code :

```
// creates a URLStream object
var stream:URLStream = new URLStream();

// listen to the ProgressEvent.PROGRESS event to grab the incoming bytes
stream.addEventListener ( ProgressEvent.PROGRESS, onProgress );
stream.addEventListener ( Event.COMPLETE, onComplete );

// download the remote image
stream.load ( new URLRequest ("http://dl.dropbox.com/u/7009356/IMG_4958.jpg" ) );

// store the incoming bytes
var buffer:ByteArray = new ByteArray();

// Loader to display the picture
var loader:Loader = new Loader();

// show it
addChild ( loader );

function onProgress ( e:ProgressEvent ):void
{
    // we keep writing the bytes coming in
    stream.readBytes ( buffer, buffer.length );
    // we clear the previously loaded content
    loader.unload();
    // we inject the bytes to display the image
    loader.loadBytes ( buffer );
}

function onComplete ( e:Event ):void
{
    trace ("complete");
}
```

As the bytes are coming in we are displaying them through the `loadBytes` API, the Flash Player shows what can be displayed at the time we inject the bytes. The following picture illustrates the picture partially loaded :

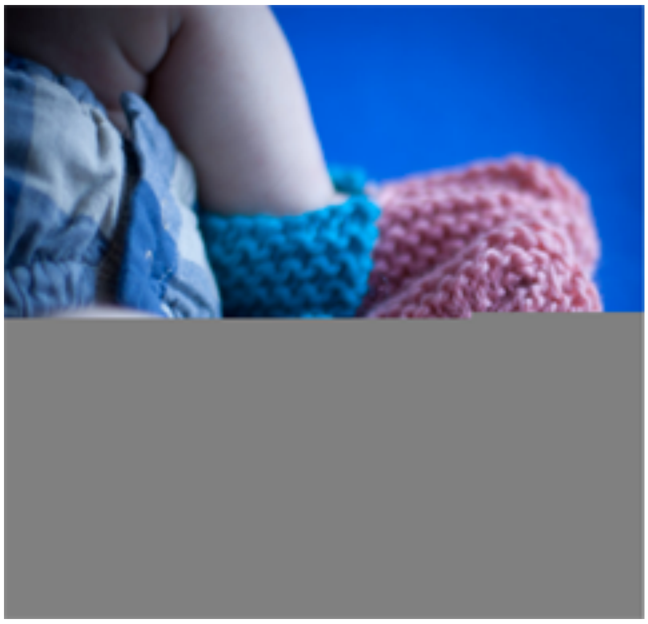


Figure 2.1

Image partially loaded.

Then, step by step, the image loads to the point the image is finally loaded and displayed :

Figure 2.2

Image completely loaded.

Now you may wonder the true value of this, but this gives you a better idea of what you can do with bytes when it comes to `URLStream`. As we just saw, `loadBytes` is a very powerful API but has its limitations. In the following section we will discover why developers can be sometimes limited with it.

Loadbytes limitations

Keep in mind that `loadBytes` is asynchronous, so when injecting the bytes you will need to wait for the `Event.COMPLETE` event before being able to retrieve anything from the loaded SWF, like size, class definitions etc. The code below illustrates the idea :

```
import flash.utils.ByteArray;
import flash.display.Loader;

[Embed(source="library.swf", mimeType="application/octet-stream")]
var library:Class;

// instanciate the stream as a ByteArray
var swfBytes:ByteArray = new library();

// instanciate a Loader
var myLoader:Loader = new Loader();

// show the Loader object
addChild ( myLoader );

// loadBytes is asynchronous, we need to wait for the complete event before retrieving content from the
SWF
myLoader.contentLoaderInfo.addEventListener ( Event.COMPLETE, onComplete );

// inject the embedded stream inside the Loader
// the SWF is executed automatically
myLoader.loadBytes(library);

// handler
function onComplete ( e:Event ):void
{
    trace("loading complete");
}
```

The definitions can be retrieved only when the loading is complete :

```
// handler
function onComplete ( e:Event ):void
{
    // retrieve a class definition
    var classDefinition:Class = e.currentTarget.applicationDomain.getDefinition("MyDefinition"):
}
```

And this limitation is true for every kind of file you are loading in the `Loader` object. A lot of developers expect to load an image and retrieve its dimensions in a synchronous way, but as with our runtime shared library, the dimensions of the image can be retrieved only when loading is complete and the image parsed by the Flash Player :

```
import flash.utils.ByteArray;
import flash.display.Loader;

[Embed(source="image.jpeg", mimeType="application/octet-stream")]
var imageStream:Class;

// instanciate the stream as a ByteArray
var jpegBytes:ByteArray = new imageStream();

// instanciate a Loader
var myLoader:Loader = new Loader();

// show the Loader object
addChild ( myLoader );

// loadBytes is asynchronous, we need to wait for the complete event before retrieving image dimensions
```

```
myLoader.contentLoaderInfo.addEventListener ( Event.COMPLETE, onComplete );

// inject the image stream inside the Loader
myLoader.loadBytes(jpegBytes);

// handler retrieving the image size
function onComplete ( e:Event ):void
{
    // outputs : width : 300 height : 400
    trace ( "width : " + e.currentTarget.width, "height : " + e.currentTarget.height );
}
```

In case you need a synchronous behavior, you will need to process the parsing manually. For this purpose I wrote in the past a synchronous JPEG decoder (http://www.bytearray.org/?p=1089) through the use of Adobe Alchemy.

> Alchemy is a research project that allows users to compile C and C++ code that is targeted to run on the open source ActionScript Virtual Machine (AVM2). The purpose of this preview is to assess the level of community interest in reusing existing C and C++ libraries in Web applications that run on Adobe® Flash® Player and Adobe AIR®.
>
> With Alchemy, Web application developers can now reuse hundreds of millions of lines of existing open source C and C++ client or server-side code on the Flash Platform. Alchemy brings the power of high performance C and C++ libraries to Web applications with minimal degradation on AVM2. The C/C++ code is compiled to ActionScript 3.0 as a SWF or SWC that runs on Adobe Flash Player 10 or Adobe AIR 1.5.

Let's cover now some cool methods available on the `ByteArray` class related to compression.

Note

- The `registerClassAlias` is a key API to do custom serialization/deserialization.
- Remember that `loadBytes` is asynchronous.
- The `Embed` tag allows us to embed anything with a SWF.

Compressing and uncompressing data

The `ByteArray` object implements two APIs for data compression :

- `compress` : compresses the data using the zlib algorithm.
- `deflate` : compresses the data using the deflate algorithm.

Two API are available for uncompressing :

- `uncompress` : decompresses the byte array using the zlib algorithm.
- `inflate` : uncompresses the data using the inflate algorithm.

The specification of the two algorithms can be found here:

- Zlib : http://www.ietf.org/rfc/rfc1950.txt.
- Deflate : http://www.ietf.org/rfc/rfc1951.txt

Those methods can be useful when writing encoders or decoders and data is being compressed using zlib or deflate. This will make your job easier by relying on native compression or decompression. A very common use case can benefit from runtime compression and decompression. Let's take the following scenario, you need to save on the user's computer a big amount of data as an XML stream.

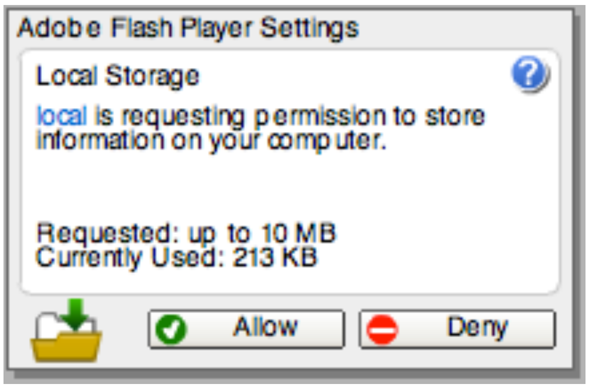
To save the data locally we are going to use the `flash.net.SharedObject` API which allows us to save permanent data as cookies. In the following example, we are loading a big amount of XML data (1.5mb) as a `ByteArray` stream :

```
var loader:URLLoader = new URLLoader();

loader.dataFormat = URLLoaderDataFormat.BINARY;

loader.load( new URLRequest ("donnees.xml") );

loader.addEventListener( Event.COMPLETE, onComplete );

function onComplete ( e:Event ):void
{
    // access the XML stream
    var streamXML:ByteArray = e.currentTarget.data;

    // outputs : 1547.358
    trace( streamXML.length / 1024 );
}
```

Once data is loaded, we save it locally through the help of the `SharedObject` and the `flush` API :

```
var loader:URLLoader = new URLLoader();

loader.dataFormat = URLLoaderDataFormat.BINARY;

loader.load( new URLRequest ("donnees.xml") );

loader.addEventListener( Event.COMPLETE, onComplete );

// crates a ShareObject called "cookie"
var sharedCookie:SharedObject = SharedObject.getLocal("cookie");

function onComplete ( pEvt:Event ):void
{
    // access the XML stream
    var streamXML:ByteArray = pEvt.currentTarget.data;

    // outputs : 1547.358
    trace( streamXML.length / 1024 );

    // copies the XML stream in the shared object
    sharedCookie.data.xmlData = streamXML;

    // saves the data
    sharedCookie.flush();
}
```

When the flush API is called, the Flash Player attempts to save the data, but the amount of data is too high to be saved transparently. The Flash Player triggers the Local Storage window and request the user approval to save the data. Remember that we are trying to save more than a megabyte of data which is uncommon.The following image illustrates such a panel :


Adobe Flash Player Settings

Local Storage

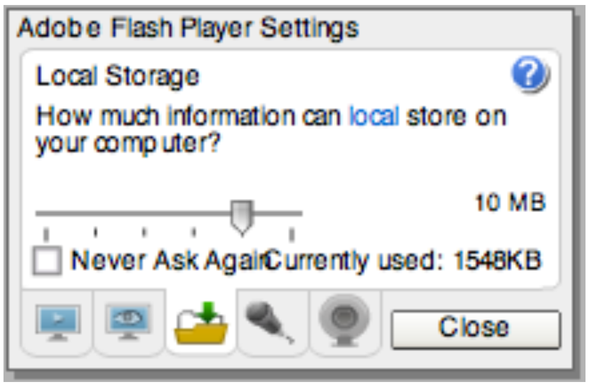
local is requesting permission to store information on your computer.

Requested: up to 10 MB
Currently Used: 213 KB

Figure 2.3

The Local Storage window requesting 10mb of space.

We can see that 10 mb are requested to save the data, in reality the amount of data is 1 548kb as the following figure illustrate, but the Flash Player requests more to make sure you will be fine for further savings :


Adobe Flash Player Settings

Local Storage

How much information can local store on your computer?

10 MB

Never Ask Again Currently used: 1548KB

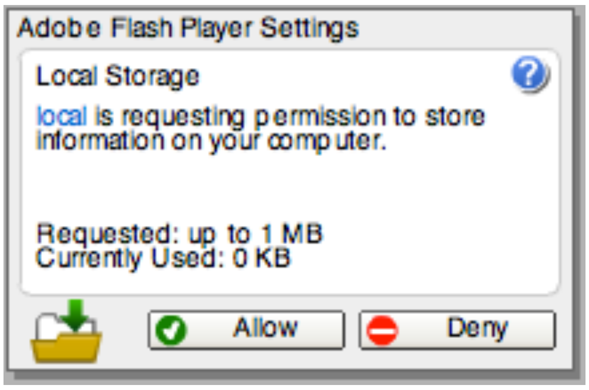
Figure 2.4

Space required to save our uncompressed XML stream.

A very simple workaround to limit the amount of data saved locally, is to compress the XML stream through the compression APIs available on the `ByteArray` object. In the following code, we are compressing the stream using the zlib algorithm through the help of the compress API. We can see a size reduction of 700% :

```
function onComplete ( pEvt:Event ):void
{
    // access the XML stream
    var streamXML:ByteArray = pEvt.currentTarget.data;

    // outputs : 1547.358
    trace( streamXML.length / 1000 );

    // compression du flux XML
    streamXML.compress();

    // affiche : 212.24
    trace( streamXML.length / 1024 );

    // copies the XML stream in the shared object
    sharedCookie.data.xmlData = streamXML;

    // saves the data
    sharedCookie.flush();
}
```

If we use this technique, the amount of data requested by the Flash Player falls to 1mb :


Adobe Flash Player Settings

Local Storage

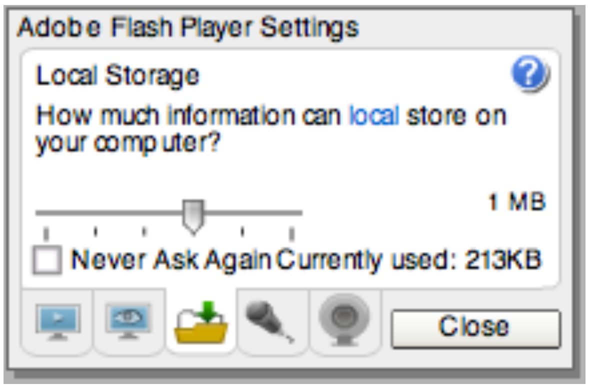
local is requesting permission to store information on your computer.

Requested: up to 1 MB
Currently Used: 0 KB

Figure 2.5

The Local Storage window requesting 1mb of space.

Once saved, we can see that the amount of data required to save our XML stream is now 213kb :


Adobe Flash Player Settings

Local Storage

How much information can local store on your computer?

1 MB

Never Ask Again Currently used: 213KB

Figure 2.6

Space required to save our compressed XML stream.

To read the data back, we just need to call the `uncompress` API then extract the `String` through the `readUTFBytes` API :

```
function onComplete ( pEvt:Event ):void
{
    // access the XML stream
    var streamXML:ByteArray = pEvt.currentTarget.data;

    // outputs : 1547.358
    trace( streamXML.length / 1000 );

    // compression du flux XML
    streamXML.compress();

    // affiche : 212.24
    trace( streamXML.length / 1024 );

    // copies the XML stream in the shared object
    sharedCookie.data.xmlData = streamXML;

    // saves the data
    sharedCookie.flush();

    // retrieves the saved XML data from the shared object
    var binaryXMLStream:ByteArray = sharedCookie.data.xmlData;

    // uncompresses the data
    binaryXMLStream.uncompress();

    // reads the stream as a UTF string
    var xmlString:String = binaryXMLStream.readUTFBytes ( binaryXMLStream.bytesAvailable );

    // creates an XML structure back
    var xmlData:XML = new XML ( xmlString );
}
```

A valid XML object is then recreated from the `String` we extracted from the `ByteArray`. Thanks to the compression, we were able to reduce the size of the XML stream or around 1 335kb. By using the deflate algorithm through `deflate`, we end up with similar results.

Note

- The `compress` API allows to compress data using the zlib algorithm.
- The `deflate` API allows us to compress data using the delate algorithm.
- The `uncompress` API allows us to uncompress data compressed wit the zlib algorithm.
- The `inflate` API allows us to uncompress data compressed wit the deflate algorithm.
- Text content is a very good candidate for compression. A lot of space can be saved this way.

Generating an image file (PNG or JPEG)

We are now going to save a JPEG file through a custom JPEG encoder. The first encoder we are going to use is the corelib JPEG encoder. The corelib package is a set of libaries provided by Adobe for ActionScript 3 developers. It includes a lot of nice classes to do things like PNG or JPEG encoding, JSON serialization or cryptography stuff and more.

You can download the corelib at the following link :

https://github.com/mikechambers/as3corelib

The corelib package contains a JPEG encoder we are going to use here to produce an image file from a `BitmapData` object, transformed to a JPEG binary stream :

```
import com.adobe.images.JPGEncoder;

// creates a big red non transparent image
var bitmap:BitmapData = new BitmapData ( 1024, 1024, false, 0x990000 );

// creates the JPEG encoder with a quality of 100
var jpegEncoder:JPEGEncoder = new JPEGEncoder ( 100 );

// encode the JPEG from the BitmapData object
var jpegBytes:ByteArray = jpegEncoder.encode ( bitmap );

// outputs : 2718
trace( jpegBytes.length );
```

By using some common optimizations techniques like the use of the `Vector` class introduced in Flash Player 10 or bitwise and strong typing optimizations we can highly optimize the encoding process.

Let's test the performance of the current one from the corelib package :

```
import com.adobe.images.JPGEncoder;

// creates a big red non transparent image
var bitmap:BitmapData = new BitmapData ( 1024, 1024, false, 0x990000 );

// creates the JPEG encoder
var jpegEncoder:JPEGEncoder = new JPEGEncoder ( 100 );

var savedTime:Number = getTimer();

// encode the JPEG from the BitmapData object
var jpegBytes:ByteArray = jpegEncoder.encode ( bitmap );

// outputs : 1542
trace ( getTimer() - savedTime );
```

1542 ms are required to encode the JPEG image. Not bad. Let's try now with an optimized version available at this link : http://www.bytearray.org/?p=775. By using this version we highly reduce the encoding time :

```
// creates a big red non transparent image
var bitmap:BitmapData = new BitmapData ( 1024, 1024, false, 0x990000 );

// creates the JPEG encoder (using this time the optimized version)
var jpegEncoder:JPEGEncoder = new JPEGEncoder ( 100 );

var savedTime:Number = getTimer();

// encode the JPEG from the BitmapData object
var jpegBytes:ByteArray = jpegEncoder.encode ( bitmap );

// outputs : 396
trace ( getTimer() - savedTime );
```

Of course, the beauty of this, is that once the JPEG bytes have been generated you can preview the JPEG image directly inside the Flash Player. We just used recently the `loadBytes` API to inject bytes.

In the following code, we instantiate an image from our library and compress it to a JPEG file using a quality of 100 :

```
// instantiate our custom image from the library
var bitmap:BitmapData = new CustomImage ();

// creates the JPEG encoder
var jpegEncoder:JPEGEncoder = new JPEGEncoder ( 100 );

var savedTime:Number = getTimer();

// encode the JPEG from the BitmapData object
var jpegBytes:ByteArray = jpegEncoder.encode ( bitmap );

// outputs : 454
trace ( getTimer() - savedTime );
```

At the same time we can also check the final image size :

```
// instantiate our custom image from the library
var bitmap:BitmapData = new CustomImage ();

// creates the JPEG encoder
var jpegEncoder:JPEGEncoder = new JPEGEncoder ( 100 );

var savedTime:Number = getTimer();

// encode the JPEG from the BitmapData object
```

```
var jpegBytes:ByteArray = jpegEncoder.encode ( bitmap );

// outputs : 454
trace ( getTimer() - savedTime );

// outputs : 310.6435546875
trace ( jpegBytes.length / 1024 );
```

Imagine now that you want the user of your application to preview the quality of the image that is going to be generated. You can use the `loadBytes` API to inject the JPEG image and have instant feedback over the quality of the final image :

```
// instantiate our custom image from the library
var bitmap:BitmapData = new CustomImage ();

// creates the JPEG encoder
var jpegEncoder:JPEGEncoder = new JPEGEncoder ( 100 );

// encode the JPEG from the BitmapData object
var jpegBytes:ByteArray = jpegEncoder.encode ( bitmap );

// outputs : 310.6435546875
trace ( jpegBytes.length / 1024 );

// creates a Loader object for preview
var loaderPreview:Loader = new Loader();

// inject the JPEG bytes
loaderPreview.loadBytes ( jpegBytes );

// show the image
addChild ( loaderPreview );
```

When running, you get the following result :

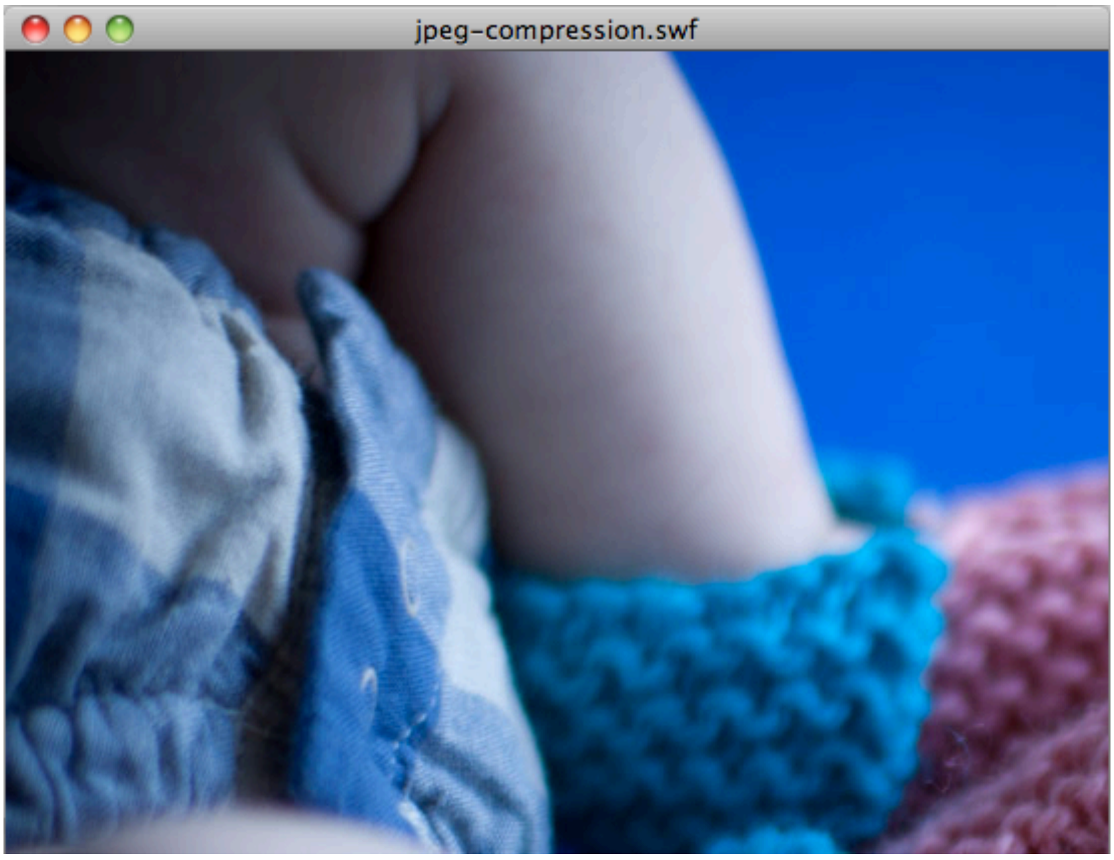


Figure 2.5
Image dynamically compressed to high quality JPEG.

You can see that the quality is really high here, but now let's change the quality down to 10 :

```
// creates the JPEG encoder
var jpegEncoder:JPEGEncoder = new JPEGEncoder ( 10 );
```

When testing, we can see that the encoder really reduced the image quality :

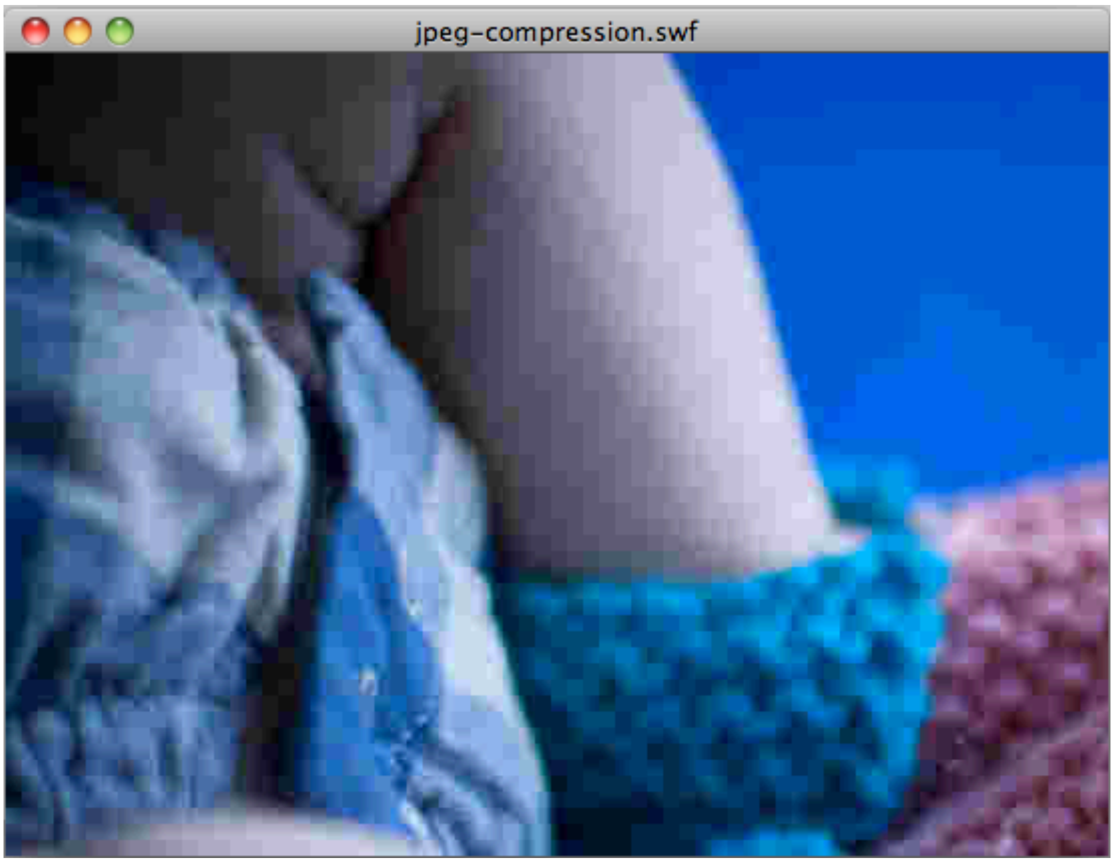


Figure 2.5
Image dynamically compressed to low quality JPEG.

If we check the image size, we see that lower the image quality reduced as expected the image size :

```
// instantiate our custom image from the library
var bitmap:BitmapData = new CustomImage ();

// creates the JPEG encoder
var jpegEncoder:JPEGEncoder = new JPEGEncoder ( 10 );

// encode the JPEG from the BitmapData object
var jpegBytes:ByteArray = jpegEncoder.encode ( bitmap );

// outputs : 25.126953125
trace ( jpegBytes.length / 1024 );

// creates a Loader object for preview
var loaderPreview:Loader = new Loader();

// inject the JPEG bytes
loaderPreview.loadBytes ( jpegBytes );

// show the image
```

```
addChild ( loaderPreview );
```

We just transformed an existing image from our library to a JPEG binary stream, but thanks to the `BitmapData.draw` API we can literally rasterize anything inside the Flash Player to a bitmap and compress this as a JPEG file.

Many scenarios can benefit from this, think about a tool which would take snapshots during the playback of a video, or a drawing tool, allowing the user to draw content and export his drawing as an image. To see some more examples of applications leveraging the `BitmapData` API, check the following links :

- Drawing Things : http://www.bytearray.org/?p=29
- AS3 Live JPEG Encoder : http://www.bytearray.org/?p=90
- FLV Encoder : http://www.zeropointnine.com/blog/flv-encoder-with-audio/

In the following example, we create an empty bitmap and rasterize the stage, we would basically capture everything on stage and take a snapshot of it, ready to be saved for later use :

```
import com.adobe.images.JPGEncoder;

// creates a big red non transparent image
var bitmap:BitmapData = new BitmapData ( stage.stageWidth, stage.stageHeight );

// rasterize everything on stage
bitmap.draw ( this );

// creates the JPEG encoder
var jpegEncoder:JPEGEncoder = new JPEGEncoder ( 100 );

// encode the JPEG from the BitmapData object
var jpegBytes:ByteArray = jpegEncoder.encode ( bitmap );
```

Remember that any display object can be passed to the `BitmapData.draw` API. But back to our image encoding. For info, we can optimize this encoding time even more by leveraging Alchemy. Through this compiler (http://labs.adobe.com/technologies/alchemy/) we are going to use a native code JPEG encoder library such as JPEGLib and compile this to an ActionScript 3 library. We are going to be able to leverage specific compiler optimizations which are not done by ASC but LLVM used by the Alchemy compiler, and produce highly optimized AS3 bytecode.

For more informations about this, check the following link: http://segfaultlabs.com/devlogs/alchemy-asynchronous-jpeg-encoding-2

We can also, generate a PNG file. For this, we will use a PNG encoder, also available in the corelib package :

```
import com.adobe.images.PNGEncoder;

// creates a transparent image matching the stage dimensions
var bitmap:BitmapData = new BitmapData ( stage.stageWidth, stage.stageHeight, true, 0 );

// rasterizes the stage into the bitmap
bitmap.draw ( this );

var savedTime:Number = getTimer();

// encode the PNG from the BitmapData object
var pngBytes:ByteArray = PNGEncoder.encode ( bitmap );

// outputs : 333
trace ( getTimer() - savedTime );
```

As we shall see later, each file can be identified by its header. Hence, by reading the PNG specification (http://www.w3.org/TR/PNG), we see that any valid PNG file must start with a signature composed of the following decimal values :

```
137 80 78 71 13 10 26 10
```

If we read the header from the stream we just generated, we see that it matches :

```
import com.adobe.images.PNGEncoder;

creates a transparent image matching the stage dimensions
var bitmap:BitmapData = new BitmapData ( stage.stageWidth, stage.stageHeight, true, 0 );

// rasterizes the stage into the bitmap
bitmap.draw ( this );

var savedTime:Number = getTimer();

// encode the JPEG from the BitmapData object
var pngBytes:ByteArray = PNGEncoder.encode ( bitmap );

// outputs : 333
trace ( getTimer() - savedTime );

// resets the stream position
pngBytes.position = 0;

// outputs : 137
trace ( pngBytes.readUnsignedByte() );

// outputs : 80
trace ( pngBytes.readUnsignedByte() );

// outputs : 78
trace ( pngBytes.readUnsignedByte() );

// outputs : 71
trace ( pngBytes.readUnsignedByte() );

// outputs : 13
trace ( pngBytes.readUnsignedByte() );

// outputs : 10
trace ( pngBytes.readUnsignedByte() );

// outputs : 26
trace ( pngBytes.readUnsignedByte() );

// outputs : 10
trace ( pngBytes.readUnsignedByte() );
```

Now we need to save this stream, and there are multiple ways to do this. Let's see the different techniques we can use.

Saving a binary stream (through a remote server)

In this first example, we are going to export the binary stream we just generated through the help of a server. O export the stream, we need to pass the binary data to a remove script, to offer the ability to doanload the stream through a dialog box. This is the trick we used before Flash Player 10, when the `FileReference` API did not allow to save content locally on the user's computer.

To do this, we are going to define an `export.php` script containing the code below :

```
<?php
if ( isset ( $GLOBALS["HTTP_RAW_POST_DATA"] )) {
    $flux = $GLOBALS["HTTP_RAW_POST_DATA"];
    header('Content-Type: image/png');
    header("Content-Disposition: attachment; filename=".$_GET['name']);
    echo $flux;

}  else echo 'An error occurred.';
?>
```

We are saving the `export.php` script of our server, then we pass the binary stream through the help of the `data` property on the `URLRequest` object :

```
import com.adobe.images.PNGEncoder;

creates a transparent image matching the stage dimensions
var bitmap:BitmapData = new BitmapData ( stage.stageWidth, stage.stageHeight, true, 0 );

// rasterizes the stage into the bitmap
bitmap.draw ( this );

var savedTime:Number = getTimer();

// encode the PNG from the BitmapData object
var pngBytes:ByteArray = PNGEncoder.encode ( bitmap );

// creates an HTTP header
var enteteHTTP:URLRequestHeader = new URLRequestHeader ("Content-type", "application/octet-stream");

// remote script URL
var requete:URLRequest = new URLRequest("http://localhost/export_image/export.php?name=sketch.jpg");

// add the custom HTTP header to our HTTP rquest
requete.requestHeaders.push(enteteHTTP);

// send the data through POST
requete.method = URLRequestMethod.POST;

// pass the PNG stream
requete.data = pngBytes;

// connect to the remote script
navigateToURL(requete, "_blank");
```

Note that in the code above, thanks to the `URLRequestHeader` object, we are indicating to the Flash Player not to treat the data being sent remotely as a String but as raw binary data.

Make sure to test the previous code inside a browser. The standalone version of the Flash Player does not allow us to send data through POST but only GET. We would end up with our image sent in the browser URL as a String.

If we wanted to save the image on the server, we would modify the remote server as following:

```
<?php

if ( isset ( $GLOBALS["HTTP_RAW_POST_DATA"] )) {

    $flux =  $GLOBALS["HTTP_RAW_POST_DATA"];

    $fp = fopen($_GET['name'], 'wb');
    fwrite($fp, $im);
    fclose($fp);
}

?>
```

As you can imagine, the script can be modified in many different ways, the image could also be saved or processed server side.

Saving a binary stream (without a remote server)

We are now going to save a file directly, without any server side interaction. This will be made possible through the `FileReference` class improved in Flash Player 10 giving the ability to save any stream locally through the `FileReference.save` API.

In the code below we do not rely on any server side script, the bytes are directly saved to the user's local disk by showing up a save as window :

```
import com.adobe.images.PNGEncoder;

creates a transparent image matching the stage dimensions
var bitmap:BitmapData = new BitmapData ( stage.stageWidth, stage.stageHeight, true, 0 );

// rasterizes the stage into the bitmap
bitmap.draw ( this );

var savedTime:Number = getTimer();

// encode the PNG from the BitmapData object
var pngBytes:ByteArray = PNGEncoder.encode ( bitmap );

// creates a FileReference object
var save:FileReference = new FileReference();

// écoute de l'événement MouseEvent.CLICK
stage.addEventListener ( MouseEvent.CLICK, sauvegardeImage );

function sauvegardeImage ( pEvt:MouseEvent ):void
{
    // triggers the save on the user's local disk
    save.save( pngBytes, "myImage.png" );
}
```

When the mouse is clicked on the stage, the save window is displayed, allowing the user to save the file where he wants to.

Note

- Flash Player 9 was unable to export a binary stream without a server side script.
- Since Flash Player 10, we can use the `save` API on the `FileReference` object.
- When sending binary data through http, we need to use the `URLRequestHeader` object to specify that data sent needs to be treated as binary.
- Binary data is stored in the `HTTP_RAW_POST_DATA` property of the `$GLOBALS` object : `$GLOBALS["HTTP_RAW_POST_DATA"]`.

Generate a PDF

In order to demonstrate in an other way, the power of the `ByteArray` API, we are going to generate dynamically another file now, a PDF file. For this, we will be using the AlivePDF library, simple and efficient which relies mainly on the `ByteArray` API internally. Other libraries are available today for this, Alessandro Crugnola's library called PurePDF ported from iPDF is an excellent library. To download PurePDF check the following link : http://code.google.com/p/purepdf/

It is actually possible to generate a PDF file with a simple `String`. A PDF file is made of a complex `String` as below :

```
%PDF-1.4
1 0 obj
<< /Type /Catalog
/Outlines 2 0 R
/Pages 3 0 R
>>
endobj
2 0 obj
<< /Type Outlines
/Count 0
>>
endobj
3 0 obj
<< /Type /Pages
/Kids [4 0 R]
/Count 1
>>
endobj
4 0 obj
<< /Type /Page
/Parent 3 0 R
/MediaBox [0 0 612 792]
/Contents 5 0 R
/Resources << /ProcSet 6 0 R >>
>>
endobj
5 0 obj
<< /Length 35 >>
stream
…Page-marking operators…
endstream
endobj
6 0 obj
[/PDF]
endobj
xref
0 7
0000000000 65535 f
0000000009 00000 n
0000000074 00000 n
0000000120 00000 n
0000000179 00000 n
0000000300 00000 n
0000000384 00000 n
trailer
<< /Size 7
/Root 1 0 R
>>
startxref
408
%%EOF
```

However, inserting content like images or any other binary data inside a PDF requires handling binary data, and this is where the `ByteArray` API comes to the rescue.

You can download the AlivePDF library at the following address : http://code.google.com/p/alivepdf/downloads/list

We will be using version 0.1.5RC here, once downloaded, we need to import the required classes, then we instantiate the `PDF` object :

```
import org.alivepdf.pdf.PDF;
import org.alivepdf.layout.Layout;
import org.alivepdf.layout.Orientation;
import org.alivepdf.layout.Unit;
import org.alivepdf.layout.Size;
import org.alivepdf.display.Display;
import org.alivepdf.saving.Method;
import org.alivepdf.fonts.CoreFont;
import org.alivepdf.links.HTTPLink;

var myPDF:PDF = new PDF ( Orientation.PORTRAIT, Unit.MM );
```

Then we define the displaying mode :

```
myPDF.setDisplayMode( Display.FULL_PAGE, Layout.SINGLE_PAGE );
```

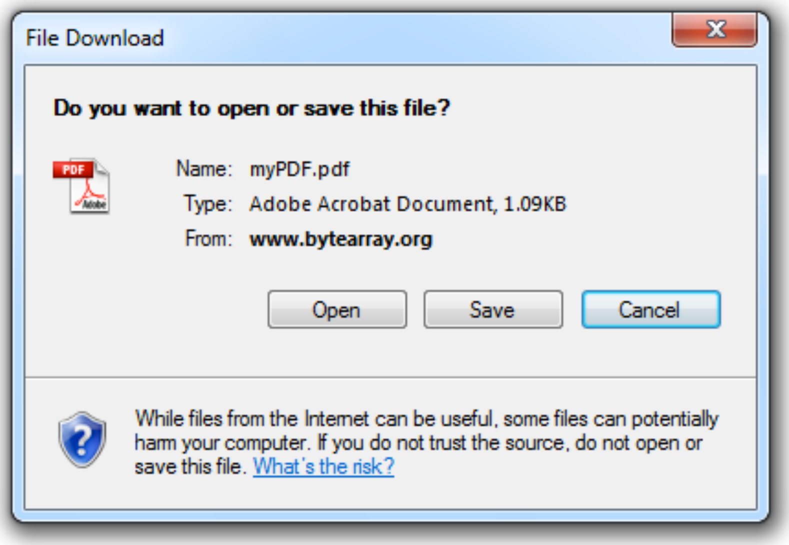
Then we add a page thanks to the `addPage` API :

```
myPDF.addPage();
```

Then we can save the PDF remotely thanks to the `create.php` script, which is exactly similar to the one we just created for saving our image server-side :

```
myPDF.save ( Method.REMOTE, 'http://localhost/pdf/create.php', Download.ATTACHMENT, 'monPDF.pdf' );
```

The save method requires the URL of the `create.php` script, to be found in the AlivePDF source package. By testing the code above with our script properly available on our remote server, we end up with the following result :


File Download

Do you want to open or save this file?

Name: myPDF.pdf
Type: Adobe Acrobat Document, 1.09KB
From: www.bytearray.org

While files from the Internet can be useful, some files can potentially harm your computer. If you do not trust the source, do not open or save this file. What's the risk?

Figure 2.7
Save-as dialog window to save the PDF.

Of course, our PDF will be empty with a blank page in it. To add some content to our PDF, let's import the following classes :

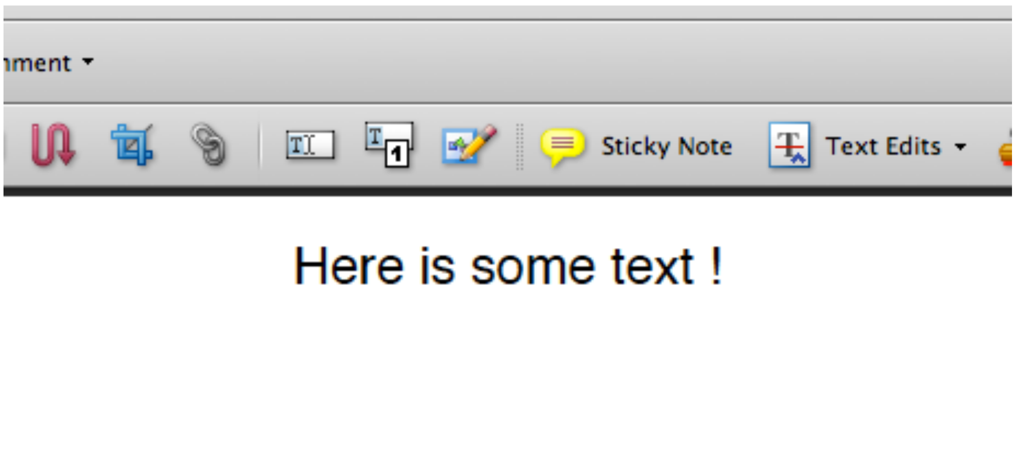
```
import org.alivepdf.fonts.Style;
import org.alivepdf.fonts.FontFamily;
import org.alivepdf.colors.RGBColor;
```

Then we add some text and a clickable link on the page :

```
// we add a page
myPDF.addPage();

// we set the text style color
```

```
myPDF.textStyle ( new RGBColor ( 0x000000 ), 1 );

// we create a system font
var coreFont:CoreFont = new CoreFont ( FontFamily.HELVETICA );

// we set the PDF document to use this font at this size
myPDF.setFont ( coreFont, 20 );

// we add some text
myPDF.addText ("Here is some text !", 70, 12);

// we add a clickable link
myPDF.addLink ( 70, 4, 52, 16, new HTTPLink ( "http://alivepdf.bytearray.org" ));
```

By testing the code above, we end up this time with a PDF with our clickable text on the first page redirecting to alivepdf.bytearray.org when clicked.



Here is some text !

Figure 2.8

Clickable text available on the first page.

Let's add an image now. We have our embedded bitmap associated to the class `Logo`, we simply pass this `Bitmap` object to the addImage API and we are done :

```
// we add a page
myPDF.addPage();

// create a BitmapData object
var pixels:Logo = new Logo(0,0);

// embeds it into a Bitmap
var logoImage:Bitmap = new Bitmap ( pixels );

// add it to the current page
myPDF.addImage ( logoImage );
```

The next image illustrates the result :

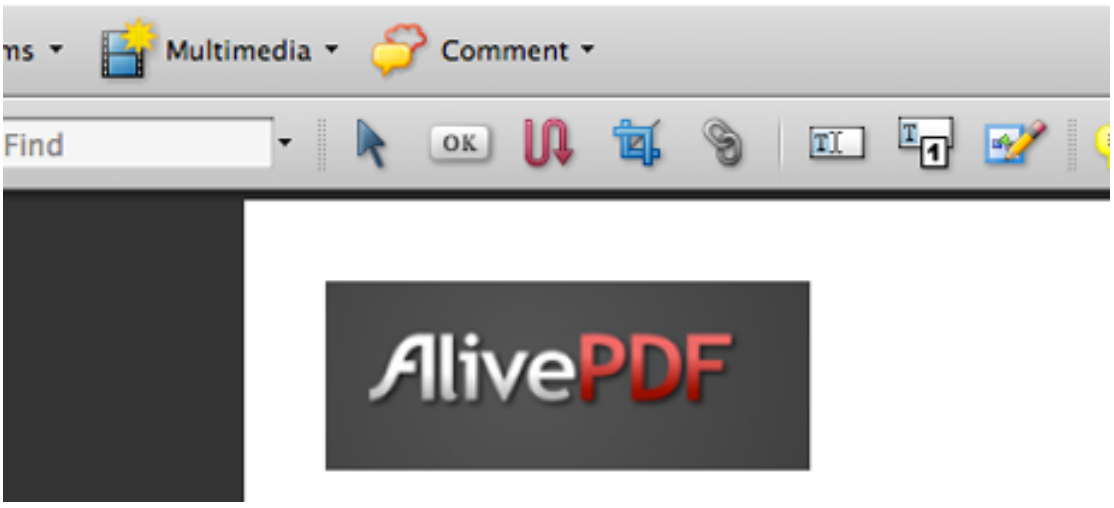


Figure 2.9
Image inside the PDF.

By using the `FileReference.save` API, we can very simply save our PDF locally with the following code :

```
var save:FileReference = new FileReference();

stage.addEventListener ( MouseEvent.CLICK, saveImage );

function saveImage ( e:MouseEvent ):void
{
    // triggers the saving of the PDF
    save.save ( myPDF.save ( Method.LOCAL ), "myPDF.pdf" );
}
```

We now have a nice solution for PDF dynamic generation on the client-side. Much appreciated for the bandwidth saving and code reduction, as no server-side scripting is involved.

From bytes to sound

I personally love generating files through the use of the `ByteArray` API. It is always cool to see the Flash Player generating a valid file that the user will be able to save to his computer and use immediately. In the following example, we are going to write a simple WAV encoder.

In reality, there is no real encoding algorithm involved here, but just a way to pack the raw PCM samples according to the WAV specification described below (taken from the following link : https://ccrma.stanford.edu/courses/422/projects/WaveFormat/) :

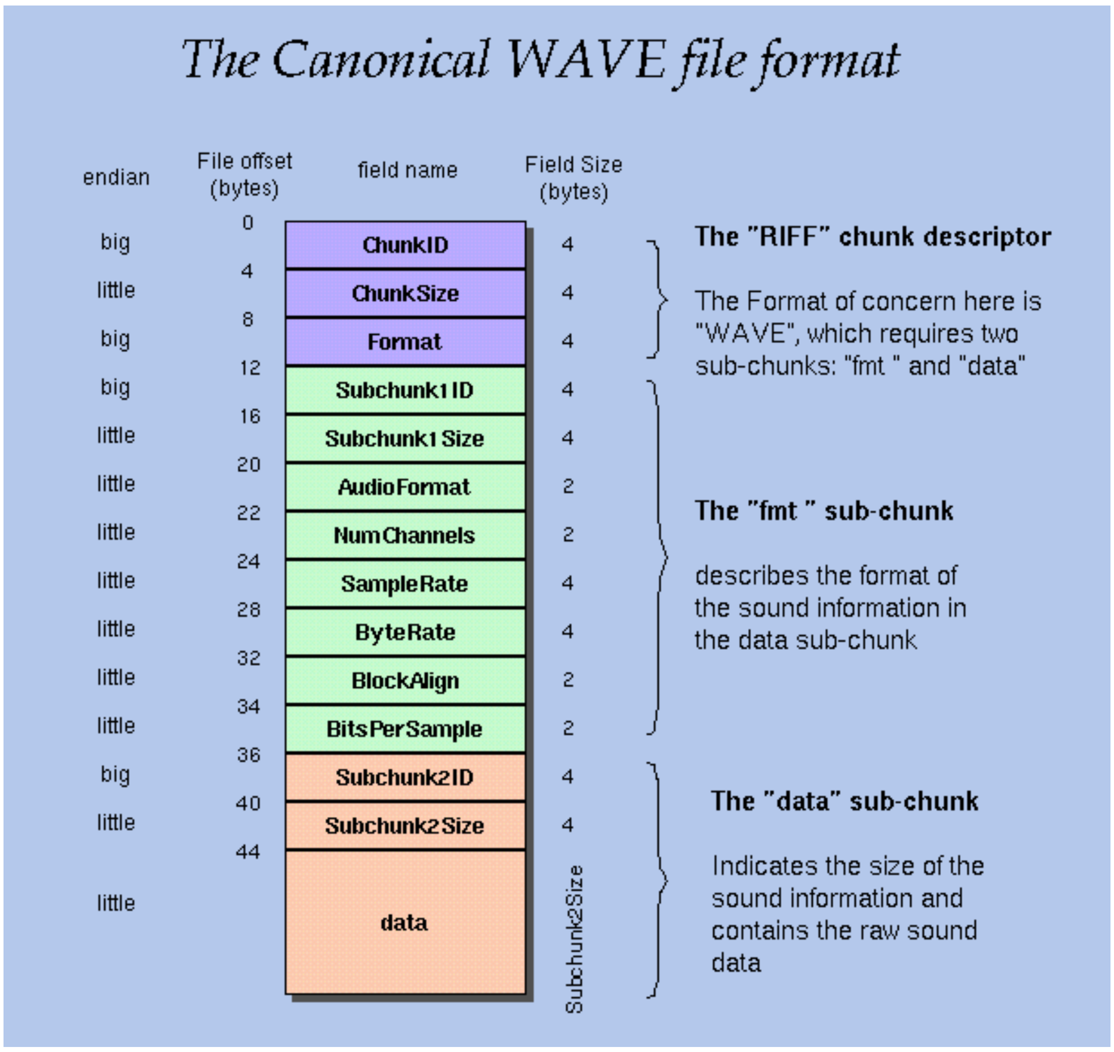


Figure 2.10
The WAV format.

We are going to focus on the WAV encoder and understand how it works. At the end of this section, saving a sound to our hard drive will be as easy as this :

```
// volume in the final WAV file will be downsampled to 50%
var volume:Number = .5;

// we create the WAV encoder to be used by MicRecorder
var wavEncoder:WaveEncoder = new WaveEncoder( volume );

// we create the MicRecorder object which does the job
```

```
var recorder:MicRecorder = new MicRecorder( wavEncoder );

// starts recording
recorder.record();

// stop recording
recorder.stop();
```

Same here, by using `FileReference` or a network API, you will be able to save the bytes to any location and reuse it. As with the PDF format, generating a file means that you should follow the specification of the file format. Of course, generating a file without any specification requires you to reverse engineer the specification. A task which can be really difficult for some formats. In our case today, it will be pretty easy, we will follow the WAV specification and create at runtime a valid WAV file.

To achieve this, we need first to retrieve the bytes we want to save. Those bytes will come from the Flash Player as raw PCM samples. Thanks to the new `SampleDataEvent` introduced in Flash Player 10.1, we will be able to listen to such an event, store the raw sound data coming from the `Microphone` and then pack those bytes into a WAV container :

```
// retrieve the Microphone
var microphone:Microphone = Microphone.getMicrophone();

// listen when data is coming in the Microphone object
microphone.addEventListener(SampleDataEvent.SAMPLE_DATA, onSampleData);
```

The `rate` property of the `Microphone` object is set to 44, to match the 44-kHz sample rate used by `Sound` objects :

```
// retrieve the Microphone
var microphone:Microphone = Microphone.getMicrophone();

// make the samples rate match the Sound object requirements
microphone.rate = 44;

// listen when data is coming in the Microphone object
microphone.addEventListener(SampleDataEvent.SAMPLE_DATA, onSampleData);

// our buffer to store the raw PCM data
var buffer:ByteArray = new ByteArray();

function onSampleData(event:SampleDataEvent):void
{
    // as long as there are bytes coming in, store them
    while(event.data.bytesAvailable > 0)
         buffer.writeFloat(event.data.readFloat());
}
```

As soon as the `Microphone` gets data, we start storing this in our buffer. Those bytes contains raw PCM samples. Not that we are storing the bytes here as mono.

To have a better understanding of the whole process, let’s have a look at our complete code :

```
import flash.events.MouseEvent;
import flash.events.SampleDataEvent;

// retrieve the Microphone
var microphone:Microphone = Microphone.getMicrophone();

// make the samples rate match the Sound object requirements
microphone.rate = 44;

// listen when data is coming in the Microphone object
```

```
microphone.addEventListener(SampleDataEvent.SAMPLE_DATA, onSampleData);

// our buffer to store the raw PCM data
var buffer:ByteArray = new ByteArray();

function onSampleData(event:SampleDataEvent):void
{
    // as long as there are bytes coming in, store them
    while(event.data.bytesAvailable > 0)
         buffer.writeFloat(event.data.readFloat());
}

stage.addEventListener(MouseEvent.CLICK, onClick);

function onClick (e:MouseEvent):void
{
    // stop writing data coming in
    microphone.removeEventListener(SampleDataEvent.SAMPLE_DATA, onSampleData);

    // reset position
    buffer.position = 0;

    // create an empty Sound object
    var sound:Sound = new Sound();

    // listen when the Sound object requests sound data
    sound.addEventListener(SampleDataEvent.SAMPLE_DATA, playbackSampleHandler);
    sound.play();
}

function playbackSampleHandler(event:SampleDataEvent):void
{
    for (var i:int = 0; i < 8192 && buffer.bytesAvailable > 0; i++)
    {
         // write the data stores back in the speakers when the Sound object requests samples
        var sample:Number = buffer.readFloat();
        event.data.writeFloat(sample);
        event.data.writeFloat(sample);
    }
}
```

A little while ago, I built a tiny library (MicRecorder - http://www.bytearray.org/?p=1858) to provide such a functionality. As mentioned earlier, capturing audio from the microphone and saving it as a WAV file, is then limited to the following straightforward code :

```
// volume in the final WAV file will be downsampled to 50%
var volume:Number = .5;

// we create the WAV encoder to be used by MicRecorder
var wavEncoder:WaveEncoder = new WaveEncoder( volume );

// we create the MicRecorder object which does the job
var recorder:MicRecorder = new MicRecorder( wavEncoder );

// starts recording
recorder.record();

// stop recording
recorder.stop();
```

Let's have a look at how we can provide this feature as a library. Below is the description of our `MicRecorder` class :

```
package org.bytearray.micrecorder
{
    import flash.events.Event;
    import flash.events.EventDispatcher;
    import flash.events.SampleDataEvent;
    import flash.events.StatusEvent;
    import flash.media.Microphone;
    import flash.utils.ByteArray;
    import flash.utils.getTimer;

    import org.bytearray.micrecorder.encoder.WaveEncoder;
    import org.bytearray.micrecorder.events.RecordingEvent;

    /**
     * Dispatched during the recording of the audio stream coming from the microphone.
     *
     * @eventType org.bytearray.micrecorder.RecordingEvent.RECORDING
     *
     * * @example
     * This example shows how to listen for such an event :
     * <div class="listing">
     * <pre>
     *
     * recorder.addEventListener ( RecordingEvent.RECORDING, onRecording );
     * </pre>
     * </div>
     */
    [Event(name='recording', type='org.bytearray.micrecorder.RecordingEvent')]

    /**
     * Dispatched when the creation of the output file is done.
     *
     * @eventType flash.events.Event.COMPLETE
     *
     * @example
     * This example shows how to listen for such an event :
     * <div class="listing">
     * <pre>
     *
     * recorder.addEventListener ( Event.COMPLETE, onRecordComplete );
     * </pre>
     * </div>
     */
    [Event(name='complete', type='flash.events.Event')]

    /**
     * This tiny helper class allows you to quickly record the audio stream coming from the Microphone a
save this as a physical file.
     * A WavEncoder is bundled to save the audio stream as a WAV file
     * @author Thibault Imbert - bytearray.org
     * @version 1.2
     *
     */
    public final class MicRecorder extends EventDispatcher
    {
         private var _gain:uint;
         private var _rate:uint;
         private var _silenceLevel:uint;
         private var _timeOut:uint;
         private var _difference:uint;
         private var _microphone:Microphone;
         private var _buffer:ByteArray = new ByteArray();
         private var _output:ByteArray;
         private var _encoder:IEncoder;

         private var _completeEvent:Event = new Event ( Event.COMPLETE );
         private var _recordingEvent:RecordingEvent = new RecordingEvent( RecordingEvent.RECORDING, 0 );

         /**
          *
          * @param encoder The audio encoder to use
          * @param microphone The microphone device to use
          * @param gain The gain
          * @param rate Audio rate
          * @param silenceLevel The silence level
          * @param timeOut The timeout
          *
          */
         public function MicRecorder(encoder:IEncoder, microphone:Microphone=null, gain:uint=100,
rate:uint=44, silenceLevel:uint=0, timeOut:uint=4000)
         {
              _encoder = encoder;
              _microphone = microphone;
              _gain = gain;
              _rate = rate;
              _silenceLevel = silenceLevel;
              _timeOut = timeOut;
         }

         /**
          * Starts recording from the default or specified microphone.
          * The first time the record() method is called the settings manager may pop-up to request acce
the Microphone.
          */
         public function record():void
         {
              if ( _microphone == null )
                   _microphone = Microphone.getMicrophone();

              _difference = getTimer();

              _microphone.setSilenceLevel(_silenceLevel, _timeOut);
              _microphone.gain = _gain;
              _microphone.rate = _rate;
              _buffer.length = 0;

              _microphone.addEventListener(SampleDataEvent.SAMPLE_DATA, onSampleData);
              _microphone.addEventListener(StatusEvent.STATUS, onStatus);
         }

         private function onStatus(event:StatusEvent):void
         {
              _difference = getTimer();
         }

         /**
          * Dispatched during the recording.
          * @param event
          */
         private function onSampleData(event:SampleDataEvent):void
         {
              _recordingEvent.time = getTimer() - _difference;

              dispatchEvent( _recordingEvent );

              while(event.data.bytesAvailable > 0)
                   _buffer.writeFloat(event.data.readFloat());
         }

         /**
          * Stop recording the audio stream and automatically starts the packaging of the output file.
          */
         public function stop():void
         {

              _microphone.removeEventListener(SampleDataEvent.SAMPLE_DATA, onSampleData);

              _buffer.position = 0;
              _output = _encoder.encode(_buffer, 1);
              dispatchEvent( _completeEvent );

         }

         /**
          *
          * @return
          *
          */
         public function get gain():uint
         {
              return _gain;
         }

         /**
          *
          * @param value
          *
          */
         public function set gain(value:uint):void
         {
              _gain = value;
         }

         /**
          *
          * @return
          *
          */
         public function get rate():uint
         {
              return _rate;
         }

         /**
          *
          * @param value
          *
          */
         public function set rate(value:uint):void
         {
              _rate = value;
         }

         /**
          *
          * @return
          *
          */
         public function get silenceLevel():uint
         {
              return _silenceLevel;
         }

         /**
          *
          * @param value
          *
          */
         public function set silenceLevel(value:uint):void
         {
              _silenceLevel = value;
         }


         /**
          *
```

```
          * @return
          *
          */
         public function get microphone():Microphone
         {
              return _microphone;
         }

         /**
          *
          * @param value
          *
          */
         public function set microphone(value:Microphone):void
         {
              _microphone = value;
         }

         /**
          *
          * @return
          *
          */
         public function get output():ByteArray
         {
              return _output;
         }

         /**
          *
          * @return
          *
          */
         public override function toString():String
         {
              return "[MicRecorder gain=" + _gain + " rate=" + _rate + " silenceLevel=" + _silenceLevel
timeOut=" + _timeOut + " microphone:" + _microphone + "]";
         }
    }
}
```

To make this `MicRecorder` object work, we need our encoder object to pass into its constuctor. First we define an `IEncoder` interface :

```
package org.bytearray.micrecorder
{
    import flash.utils.ByteArray;

    public interface IEncoder
    {
         function encode(samples:ByteArray, channels:int=2, bits:int=16, rate:int=44100):ByteArray;
    }
}
```

Then we define our `WavEncoder` class as following :

```
package org.bytearray.micrecorder.encoder
{
    import flash.events.Event;
    import flash.utils.ByteArray;
    import flash.utils.Endian;

    import org.bytearray.micrecorder.IEncoder;

    public class WaveEncoder implements IEncoder
    {
         private static const RIFF:String = "RIFF";
         private static const WAVE:String = "WAVE";
```

```
        private static const FMT:String = "fmt ";
        private static const DATA:String = "data";

        private var _bytes:ByteArray = new ByteArray();
        private var _buffer:ByteArray = new ByteArray();
        private var _volume:Number;

        /**
         *
         * @param volume
         *
         */
        public function WaveEncoder( volume:Number=1 )
        {
            _volume = volume;
        }

        /**
         *
         * @param samples
         * @param channels
         * @param bits
         * @param rate
         * @return
         *
         */
        public function encode( samples:ByteArray, channels:int=2, bits:int=16, rate:int=44100 ):ByteAr
        {
            var data:ByteArray = create( samples );

            _bytes.length = 0;
            _bytes.endian = Endian.LITTLE_ENDIAN;

            _bytes.writeUTFBytes( WaveEncoder.RIFF );
            _bytes.writeInt( uint( data.length + 44 ) );
            _bytes.writeUTFBytes( WaveEncoder.WAVE );
            _bytes.writeUTFBytes( WaveEncoder.FMT );
            _bytes.writeInt( uint( 16 ) );
            _bytes.writeShort( uint( 1 ) );
            _bytes.writeShort( channels );
            _bytes.writeInt( rate );
            _bytes.writeInt( uint( rate * channels * ( bits >> 3 ) ) );
            _bytes.writeShort( uint( channels * ( bits >> 3 ) ) );
            _bytes.writeShort( bits );
            _bytes.writeUTFBytes( WaveEncoder.DATA );
            _bytes.writeInt( data.length );
            _bytes.writeBytes( data );
            _bytes.position = 0;

            return _bytes;
        }

        private function create( bytes:ByteArray ):ByteArray
        {
            _buffer.endian = Endian.LITTLE_ENDIAN;
            _buffer.length = 0;
            bytes.position = 0;

            while( bytes.bytesAvailable )
                _buffer.writeShort( bytes.readFloat() * (0x7fff * _volume) );
            return _buffer;
        }
    }
}
```

The interesting part is the encode function which creates the header and pack the raw PCM bytes inside the WAV container. See, as with the SWF processing, we followed carefully the WAV specification and implemented a WAV encoder in a few lines of code.

Now, to be honest there is one thing that may drive you crazy if you had to recreate this WAV encoder. There is a magic number involved here, do you see it ?

```
    while( bytes.bytesAvailable )
                 _buffer.writeShort( bytes.readFloat() * (0x7fff * _volume) );
```

The WAV specification we have here does not really inform us about how we needto write the PCM samples in the WAV file. According to the picture (figure 2.10), we need to write the data. Not very clear. By reading the website (https://ccrma.stanford.edu/courses/422/projects/WaveFormat/) where this picture is extracted from, we can read at the bottom of the page, the following note :

8-bit samples are stored as unsigned bytes, ranging from 0 to 255. 16-bit samples are stored as 2's-complement signed integers, ranging from -32768 to 32767.

If we convert the hexadecimal value 0x7FFF to decimal, we realize that this is the value we use here. The raw PCM samples we are reteieveing from the microphone are ranging from -1 to 1, so remember that they need to be multiplied by this magic value to be brought to life in your speakers.

By using `FileReference` or a network API, you will be able to save the bytes to any location. The following code illustrates how we would do this thanks to `MicRecorder` :

```
var save:FileReference:FileReference = new FileReference();

private function onSave(e:Event):void
{
     save.save ( recorder.output, "recorded.wav" );
}
```

Of course, more sound format can be played or generated dynamically. A little while ago, when AIR 2.0 was introduced, I published an Adobe AIR application (http://www.bytearray.org/?p=1142) encoding the raw PCM waves to MP3 through the use of the Lame MP3 library. I was able to the native library through the use of the native process feature.

More recently, Hook (a digital production company) released on their blog an OGG Vorbis encode/decoder library ported to Flash thanks to Alchemy. For more details, make sure to check the following address : http://labs.byhook.com/2011/02/22/ogg-vorbis-encoder-decoder-for-flash/

In the following section, we will be playing with a brand new API which was added in the latest developments builds of the Flash Player.

From compressed bytes to sound

Recently, we introduced at Adobe, a new program called Incubator (http://labs.adobe.com/technologies/flashplatformruntimes/incubator/). This idea behind this initiative is to give developers early access to features we are considering for future releases and to get feedback from the community, Before Flash Player 11, there was no way to inject external sound bytes into a `Sound` object. Let's say you had an MP3 byte stream, you would not be able to inject it into a `Sound` object and play it. What people have been doing for the past years is generating an empty SWF file, inject the sound in it, and play it through the `loadBytes` API.

In the Incubator builds (version 11.0.0.58 available here for download : http://labs.adobe.com/downloads/flashplatformruntimes_incubator.html) we introduced a new API available on the `Sound` object which allows us to inject an MP3 stream into the `Sound` object. Remember that this API may change in the future or may never be released, but well, it is fun, so let's try it!

Here is the signature of the method we are going to try, this API loads an MP3 sound data from a `ByteArray` into a `Sound` object :

```
loadMP3FromByteArray(bytes:ByteArray, bytesLength:uint):void
```

In the code below, we use a `FileReference` object to select a file locally and inject it to the `Sound` object through the `loadMP3FromByteArray` API :

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.media.Sound;
    import flash.net.FileReference;

    public class TestLoadSound extends Sprite
    {
         private var file:FileReference = new FileReference();
         private var sound:Sound = new Sound();
         private var t:String;

         public function TestLoadSound()
         {
              stage.addEventListener(MouseEvent.CLICK,onClick);
              file.addEventListener(Event.SELECT, onSelect);
              file.addEventListener(Event.COMPLETE, onComplete);
         }

         private function onComplete(event:Event):void
         {
              // inject the selected sound bytes
              sound.loadMP3FromByteArray( file.data, file.data.length );
              sound.play();
         }

         private function onSelect(event:Event):void
         {
              file.load();
         }

         private function onClick(event:MouseEvent):void
         {
              file.browse();
         }
```

```
        }
}
```

When the file is selected, the sound plays nicely. Done! Pretty cool hu?

Ok, let’s have a look now at how we can parse a more complex data structure. We are using Flash, so let’s have a look at how we can parse a SWF file. Adobe published the SWF specification a few years ago, we are going to rely on it to parse a SWF and see what is inside.

Parsing binary data

Parsing binary data can be very useful. In the following example, we are going to parse a SWF file and extract information we need from it. Since the introduction of ActionScript 3 in Flash Player 9, the `ApplicationDomain` class allows developers to use and isolate and retrieve class definitions at runtime.

The SWF example

The limitation is that there is no way to actually extract definitions you do not know the names. For instance, let’s say that you know the name of a definition, you can just use the `ApplicationDomain.getDefinition` API. Now, let’s say we need to retrieve any linked classes attached to a symbol in the library of a SWF, to iterate through them, retrieve each definition, and use them. The data we are looking for is actually the SYMBOL class tag defined in a SWF.

In order to achieve this, we need to understand how a SWF is structured, as always the first thing to do is to study the file format specification you are going to work with. If there is no specification available publicly, then your task can be tougher depending on the file format. Fortunately, the SWF specification is available at this link : http://www.adobe.com/devnet/swf.html

Let’s take a look at how a SWF is structured. The most important thing to start with is the file header. It cover actually a lot of information. The header of a file is what defines the type of file and what to expect in it.

Here is below, according to the official SWF specification, the header of a SWF :

Field	Type	Comment
Signature	UI8	Signature byte : “F” indicates uncompressed “C” indicates compressed (SWF 6 and later only
Signature	UI8	Signature byte always “W
Signature	UI8	Signature byte always “S”

Version	UI8	Single byte file version (for example, 0x06 for SWF 6
FileLength	UI32	Length of entire file in bytes.
FrameSize	RECT	Frame size in twips
FrameRate	UI16	Frame delay in 8.8 fixed number of frames per second.
FrameCount	UI16	Total number of frames in file.

As you can see, the types here are either unsigned bytes (UI8) or unsigned integers (UI16) and we can see what to expect in the comment column. As previously covered, Flash Player does not handle those types at the ActionScript level, this means that we will need to use a uint (32 bit) to store a UI8 (8 bit uint) which is a waste of memory but we still do not have today more primitive types in AS3 to work with such data.

Let's make a simple test, you are now familiar with all the binary concepts and the `ByteArray` APIs. We first need to grab some SWF bytes to work with. Remember tha from an existing SWF you can access at runtime the current SWF bytes your code runs into.

To do this, just use the `bytes` property on the `LoaderInfo` object available from any `DisplayObject` attached to the SWF display list, here we will be using the stage :

```
import flash.display.DisplayObject;
import flash.display.Sprite;

var s:DisplayObject = new Sprite();

addChild ( s );

// outputs : [object ApplicationDomain]
trace ( s.loaderInfo.applicationDomain );

// outputs : FWSb
trace ( s.loaderInfo.bytes );
```

Notice how the Flash Player automatically tries to describe to a `String` representation the ByteArray retrieved through the `bytes` property. We can see in the outputs the first characters like FWS.

Now, let's reference the bytes and try retrieving the first characters and also the SWF version, to do this, it is very trivial. We will just use the `ByteArray.readUTFBytes()` method to retrieve the first 3 chars and we should get our string :

```
import flash.utils.ByteArray;

// outputs : [object ApplicationDomain]
trace ( this.loaderInfo.applicationDomain );
```

```
// grab the current SWF bytes
var swfBytes:ByteArray = this.loaderInfo.bytes;

// outputs : FWS
trace ( swfBytes.readUTFBytes(3) );
```

We actually retrieved 3 bytes, which correspond to the signature as ech byte represents one char. We end up with a `String` containing « FWS ». Pretty simple, right ?

To grab the current SWF version we could just keep consuming the bytes, as we can see on the SWF specification, the SWF version follows the signature, so let's keep reading the next byte :

```
import flash.utils.ByteArray;

// outputs : [object ApplicationDomain]
trace ( this.loaderInfo.applicationDomain );

// grab the current SWF bytes
var swfBytes:ByteArray = this.loaderInfo.bytes;

// outputs : FWS
trace ( swfBytes.readUTFBytes(3) );

// outputs : 11
trace ( swfBytes.readUnsignedByte() );
```

We actually detect here which SWF version has be compiled. Here we can see that our SWF version is 11 which is the SWF version being used for Flash Player 10.2. Then, according to the specification, we have the size of the SWF in bytes. As you can imagine, this is useful when you want to read the rest of the file or just separate multiple SWF's from one block of bytes.

The specification clearly mentions that this is where it started getting tricky :

The next four bytes represent an unsigned 32-bit integer indicating the file size. Here's where it starts getting tricky and machine architecture gets involved. The next four bytes are 0x4F000000 so that would imply that the file length is 1325400064 bytes, a very large number which doesn't make sense. What we failed to do is swap all the bytes.

In reality, it is not that hard, we just need to keep in mind endianness and respect it, a concept that we covered in the first chapter (*The first bits*) of this book. So we just read the next 4 bytes (UI32) in little-endian form.

Note that we switch back to big endian once we are done with the file length :

```
import flash.utils.ByteArray;

// outputs : [object ApplicationDomain]
trace ( this.loaderInfo.applicationDomain );

// grab the current SWF bytes
var swfBytes:ByteArray = this.loaderInfo.bytes;

// outputs : FWS
trace ( swfBytes.readUTFBytes(3) );

// outputs : 11
trace ( swfBytes.readUnsignedByte() );

swfBytes.endian = Endian.LITTLE_ENDIAN;
```

```
// outputs : 2372
trace ( swfBytes.readUnsignedInt() );

swfBytes.endian = Endian.BIG_ENDIAN;
```

Now, let's read what the specification also says about the SWF header :

CWS indicates that the entire file after the first 8 bytes (that is, after the FileLength field) was compressed by using the ZLIB open standard. The data format that the ZLIB library uses is described by Request for Comments (RFCs) documents 1950 to 1952. CWS file compression is permitted in SWF 6 or later only.

Does that ring a bell ? As we just saw, the signature informs us about the compression being used for the SWF. If the first char of the header signature is C (0x43) it means we need to uncompress the data before going further. The good news is that we actually have native compression or uncompression on the `ByteArray` class, thanks to the `compress` and `uncompress` APIs !

We can then add the following change in our code, to make sure that if the SWF is compressed, we uncompress it properly before reading further :

```
import flash.utils.ByteArray;

// outputs : [object ApplicationDomain]
trace ( this.loaderInfo.applicationDomain );

// grab the current SWF bytes
var swfBytes:ByteArray = this.loaderInfo.bytes;

// stores the compressed state (C or F)
var compressed:uint = swfBytes.readUnsignedByte();

// skip WS
swfBytes.position += 2;

// outputs : 11
trace ( swfBytes.readUnsignedByte() );

swfBytes.endian = Endian.LITTLE_ENDIAN;

// outputs : 2372
trace ( swfBytes.readUnsignedInt() );

swfBytes.endian = Endian.BIG_ENDIAN;

// creates an empty ByteArray to store the uncompressed SWF bytes
var swfBuffer:ByteArray = new ByteArray();

// copies the bytes
swfBytes.readBytes ( swfBuffer );

const COMPRESSED:uint = 0x43;

// if the SWF we are working on is compressed, we uncompress the swfBuffer
if ( compressed == COMPRESSED )
         swfBuffer.uncompress();
```

The next part of the stream, what is called the RECT in the specification is tricky to parse. Let's have a closer look at the details of the RECT chunk. In the specification here is what is said :

Field	Type	Comment
Nbits	UB[5]	Bits used for each subsequent field.
xMin	SB[Nbits]	x minimum position for rectangle in twips.
xMax	SB[Nbits]	x maximum position for rectangle in twips.
yMin	SB[Nbits]	y minimum position for rectangle in twips.
yMax	SB[Nbits]	y maximum position for rectangle in twips.

Now, let's see what a RECT looks like in binary notation :

```
0111 1000 0000 0000 0000 0101 0101 1111 0000 0000
0000 0000 0000 1111 1010 0000 0000 0000
```

There are five fields in a rectangle structure: Nbits, Xmin, Xmax, Ymin, Ymax. The unsigned Nbits field occupies the first five bits of the rectangle and indicates how long the next four signed fields are :

```
01111 -> 15
```

In this case, this means that each attribute, basically describing our stage dimensions, requires 15-bit space:

```
000000000000000 ‹ 0 = Xmin
010101011111000 ‹ 11000 = Xmax
000000000000000 ‹ 0 = Ymin
001111101000000 ‹ 8000 = Ymax
```

Which gives us the following code to parse the RECT and the frame rate and frame count :

```
import flash.utils.ByteArray;

// outputs : [object ApplicationDomain]
trace ( this.loaderInfo.applicationDomain );

// grab the current SWF bytes
var swfBytes:ByteArray = this.loaderInfo.bytes;
```

```
// stores the compressed state (C or F)
var compressed:uint = swfBytes.readUnsignedByte();

// skip WS
swfBytes.position += 2;

// outputs : 11
trace ( swfBytes.readUnsignedByte() );

swfBytes.endian = Endian.LITTLE_ENDIAN;

// outputs : 2372
trace ( swfBytes.readUnsignedInt() );

swfBytes.endian = Endian.BIG_ENDIAN;

// creates an empty ByteArray to store the uncompressed SWF bytes
var swfBuffer:ByteArray = new ByteArray();

// copies the bytes
swfBytes.readBytes ( swfBuffer );

const COMPRESSED:uint = 0x43;

// if the SWF we are working on is compressed, we uncompress the swfBuffer
if ( compressed == COMPRESSED )
         swfBuffer.uncompress();

var firstBRect:uint = swfBuffer.readUnsignedByte();

var size:uint = firstBRect >> 3;
var offset:uint = (size-3);

var threeBits:uint = firstBRect & 0x7;

var buffer:uint = 0;
var pointer:uint = 0;
var source:uint = swfBuffer.readUnsignedByte();

var xMin:uint = (readBits(offset) | (threeBits << offset)) / 20;
var yMin:uint = readBits(size) / 20;
var wMin:uint = readBits(size) / 20;
var hMin:uint = readBits(size) / 20;

// outputs : 0 550 0 400
trace ( xMin, yMin, wMin, hMin );

var frameRate:uint = swfBuffer.readShort() & 0xFF;

var numFrames:uint = swfBuffer.readShort();

var frameCount:uint = (numFrames >> 8) & 0xFF | ((numFrames & 0xFF) << 8);

// outputs : 24 1
trace ( frameRate, frameCount );

swfBuffer.endian = Endian.LITTLE_ENDIAN;

function readBits(numBits:uint):uint
{
    buffer = 0;
    var currentMask:uint;
    var bitState:uint;
    // for the number of bits to read
    for ( var i:uint = 0; i<numBits; i++)
    {
         // we create a mask which goes from left to right
```

```
          currentMask = (1 << 7) >> pointer++;
          // we store each bit state resulting from the mask operation
          bitState = uint((source & currentMask) != 0);
          // we store that bit state by recreating a value
          buffer |= bitState << ((numBits - 1) - i);
          // when we are running out of byte we read a new one and reset the pointer for the mask
          if ( pointer == 8 )
          {
               source = swfBuffer.readUnsignedByte();
               pointer = 0;
          }
     }
     return buffer;
}
```

So you may be thinking, well, what is going on here. A lot of code, but it is actually not hard, we just need to go step by step. After reading the SWF version we need to read the file size in little endian form, we could do the conversion manually but here we use the `endian` property to illustrate the value of it.

Right after this, we just parsed the RECT chunk, We did this through the `readBits` function we defined, which allows us to read a specific number of bits passed as parameter. As we saw in the previous chapter, there is no native API in the Flash Player allowing you to read a specific number of bits, to do this, we need to write our own helper function.

As defined in the table before, the RECT structure begins with 5 bits defining the number of bits that we need to read for each attribute (xMin, xMax, yMin and yMax). So we first read those 5 bits from the byte by a simple shifting to the right :

```
var size:uint = firstBRect >> 3;
```

As we only need the first 5 bits, the 3 last bits can be ignored here, which gives us the following result :

```
01111 -> 15
```

Our `readBits` function will start reading from the next byte, so we need to store those 3 shifted bits and inform the `readBits` function that we need to read 15 bits minus 3 for the first attribute (xMin) :

```
var firstBRect:uint = SWFBytes.readUnsignedByte();

var size:uint = firstBRect >> 3;
var offset:uint = (size-3);

var threeBits:uint = firstBRect & 0x7;
```

Then we read our 4 attributes. Note how we rebuild our first attribute from the 12-bit (15-3) read and shift the 3 bits we already saved to recompose our final value :

```
var source:uint = SWFBytes.readUnsignedByte();

var xMin:uint = (readBits(offset) | (threeBits << offset)) / 20;
var yMin:uint = readBits(size) / 20;
var wMin:uint = readBits(size) / 20;
var hMin:uint = readBits(size) / 20;

// outputs : 0 550 0 400
trace ( xMin, yMin, wMin, hMin );
```

Then we parsed the frame rate and frame count. Not that for the frame count, we did the conversion from big endian to little endian manually :

```
var frameRate:uint = SWFBytes.readShort() & 0xFF;

var numFrames:uint = SWFBytes.readShort();

var frameCount:uint = (numFrames >> 8) & 0xFF | ((numFrames & 0xFF) << 8);

// outputs : 24 1
trace ( frameRate, frameCount );
```

We are now done with the header, let's see what is next. Here is what the SWF specification says about what is next :

Following the header is a series of tagged data blocks. All tags share a common format, so any program parsing a SWF file can skip over blocks it does not understand. Data inside the block can point to offsets within the block, but can never point to an offset in another block. This ability enables tags to be removed, inserted, or modified by tools that process a SWF file.

Again, from the SWF specification, we can find the folllowing figure explaining how a SWF is structured :

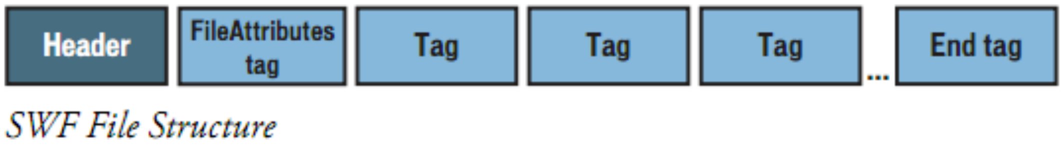


Figure 2.11
SWF structure.

So let's define our `SWFTag` object so that we can better work with a tag in our code :

```
package
{
    public class SWFTag
    {
        private var _tag:uint;
        private var _offset:uint;
        private var _endOffset:uint;

        public function SWFTag ( tag:uint, offset:uint )
        {
            _tag = tag;
            _offset = offset;
        }

        public function get tag():uint
        {
            return _tag;
        }

        public function set tag(tag:uint):void
        {
        }

        public function get offset():uint
        {
            return _offset;
        }
        public function set offset(offset:uint):void
        {
             _offset = offset;
        }

        public function get endOffset():uint
        {
             return _endOffset;
        }

        public function set endOffset(endOffset:uint):void
        {
             _endOffset = endOffset;
        }
    }
}
```

And here is our final code :

```
import flash.utils.ByteArray;

// outputs : [object ApplicationDomain]
trace ( this.loaderInfo.applicationDomain );

// grab the current SWF bytes
var swfBytes:ByteArray = this.loaderInfo.bytes;

// stores the compressed state (C or F)
var compressed:uint = swfBytes.readUnsignedByte();

// skip WS
swfBytes.position += 2;

// outputs : 11
trace ( swfBytes.readUnsignedByte() );

swfBytes.endian = Endian.LITTLE_ENDIAN;

// outputs : 2372
trace ( swfBytes.readUnsignedInt() );

swfBytes.endian = Endian.BIG_ENDIAN;

// creates an empty ByteArray to store the uncompressed SWF bytes
var swfBuffer:ByteArray = new ByteArray();

// copies the bytes
swfBytes.readBytes ( swfBuffer );

const COMPRESSED:uint = 0x43;

// if the SWF we are working on is compressed, we uncompress the swfBuffer
if ( compressed == COMPRESSED )
         swfBuffer.uncompress();

var firstBRect:uint = swfBuffer.readUnsignedByte();

var size:uint = firstBRect >> 3;
var offset:uint = (size-3);

var threeBits:uint = firstBRect & 0x7;

var buffer:uint = 0;
var pointer:uint = 0;
var source:uint = swfBuffer.readUnsignedByte();

var xMin:uint = (readBits(offset) | (threeBits << offset)) / 20;
var yMin:uint = readBits(size) / 20;
var wMin:uint = readBits(size) / 20;
var hMin:uint = readBits(size) / 20;

// outputs : 0 550 0 400
trace ( xMin, yMin, wMin, hMin );

var frameRate:uint = swfBuffer.readShort() & 0xFF;

var numFrames:uint = swfBuffer.readShort();

var frameCount:uint = (numFrames >> 8) & 0xFF | ((numFrames & 0xFF) << 8);

// outputs : 24 1
trace ( frameRate, frameCount );

swfBuffer.endian = Endian.LITTLE_ENDIAN;

var parsedTags:Vector.<SWFTag> = browseTables();

function readBits(numBits:uint):uint
{
    buffer = 0;
    var currentMask:uint;
    var bitState:uint;
    // for the number of bits to read
    for ( var i:uint = 0; i<numBits; i++)
    {
         // we create a mask which goes from left to right
         currentMask = (1 << 7) >> pointer++;
         // we store each bit state resulting from the mask operation
         bitState = uint((source & currentMask) != 0);
         // we store that bit state by recreating a value
         buffer |= bitState << ((numBits - 1) - i);
         // when we are running out of byte we read a new one and reset the pointer for the mask
         if ( pointer == 8 )
         {
              source = swfBuffer.readUnsignedByte();
              pointer = 0;
         }
    }
    return buffer;
}

function browseTables():Vector.<SWFTag>
{
    var currentTag:int;
    var step:int;
    var dictionary:Vector.<SWFTag> = new Vector.<SWFTag>();
    var infos:SWFTag;

    while ( (currentTag = ((swfBuffer.readShort() >> 6) & 0x3FF)) != 0 )
    {
         infos = new SWFTag(currentTag, swfBuffer.position);

         swfBuffer.position -= 2;
         step = swfBuffer.readShort() & 0x3F;

         trace ( currentTag );

         if ( step < 0x3F )
         {
              swfBuffer.position += step;

         } else
         {
              step = swfBuffer.readUnsignedInt();
              infos.offset = swfBuffer.position;
```

```
                swfBuffer.position += step;
            }

            infos.endOffset = swfBuffer.position;
            dictionary.push ( infos );

        }

        return dictionary;
}

var linkedSymbols:Vector.<String> = new Vector.<String>();

var symbolClassTag:uint = 76;

for each ( var tag:SWFTag in parsedTags)
{
    if ( tag.tag == symbolClassTag )
    {
        var tagOffset:uint = tag.offset;
        swfBuffer.position = tagOffset;
        var count:uint = swfBuffer.readShort();

        for (var i:uint = 0; i< count; i++)
        {
            swfBuffer.readUnsignedShort();

            var char:uint = swfBuffer.readByte();
            var className:String = new String();

            while (char != 0)
            {
                className += String.fromCharCode(char);
                char = swfBuffer.readByte();
            }
            linkedSymbols.push ( className );
        }
    }
}

// outputs : Symbol1,Symbol1copy,parse_fla.MainTimeline
trace ( linkedSymbols );
```

Let's output the current tag being introspected, in the following code we add a `trace` statement for `currentTag` :

```
function browseTables():Vector.<SWFTag>
{
    var currentTag:int;
    var step:int;
    var dictionary:Vector.<SWFTag> = new Vector.<SWFTag>();
    var infos:SWFTag;

    while ( (currentTag = ((SWFBytes.readShort() >> 6) & 0x3FF)) != 0 )
    {
        infos = new SWFTag(currentTag, SWFBytes.position);

        SWFBytes.position -= 2;
        step = SWFBytes.readShort() & 0x3F;

        trace ( currentTag );

        if ( step < 0x3F )
        {
            SWFBytes.position += step;

        } else
```

```
          {
               step = SWFBytes.readUnsignedInt();
               infos.offset = SWFBytes.position;
               SWFBytes.position += step;
          }

          infos.endOffset = SWFBytes.position;
          dictionary.push ( infos );

     }

     return dictionary;
}
```

By testing the previous code, we get the following output :

```
69
77
9
86
82
76
1
```

That looks pretty good, by looking at the SWF specification we can see that those numbers makes sense. According to our specification, we are actually reading the tags, and here is the associated information associated to each tag :

- 1 ShowFrame
- 9 SetBackgroundColor
- 69 FileAttributes
- 76 SymbolClass
- 77 Metadata
- 82 DoABC
- 86 DefineSceneAndFrameLabelData

We now have our index table of tags, by using a simple loop, we can iterate over them and now exactly where each tag begins and ends in our SWF stream, we basically created an index :

```
for each ( var tag:SWFTag in parsedTags )
{
    /* outputs :
    69 15 19
    77 25 1311
    9 1313 1316
    86 1322 1333
    82 1441 4358
    76 4364 4415
    1 4417 4417
    */
    trace ( tag.tag, tag.offset, tag.endOffset );
}
```

We now have a `Vector` filled with `SWFTag` objects, describing where each tag begins and end. That way, we can now very easily iterate over this vector and just jump into the location we need. In the following code, we iterate over the `Vector.<SWFTag>` to find the tag we need (SYMBOLCLASS) :

```
var symbolClassTag:uint = 76;

for each ( var tag:SWFTag in parsedTags )
```

```
{
    if ( tag.tag == symbolClassTag )
    {
         var tagOffset:uint = tag.offset;
         SWFBytes.position = tagOffset;
         var count:uint = SWFBytes.readShort();

         for (var i:uint = 0; i< count; i++)
         {
              SWFBytes.readUnsignedShort();

              var char:uint = SWFBytes.readByte();
              var className:String = new String();

              while (char != 0)
              {
                   className += String.fromCharCode(char);
                   char = SWFBytes.readByte();
              }
              /* outputs :
              Symbol1
              Symbol1copy
              parse_fla.MainTimeline
              */
              trace ( className );
         }
    }
}
```

In order to save the name of the class names we introspected, we created a `Vector.<String>` to save our class names:

```
var linkedSymbols:Vector.<String> = new Vector.<String>();

const symbolClassTag:uint = 76;

for each ( var tag:SWFTag in parsedTags)
{
    if ( tag.tag == symbolClassTag )
    {
         var tagOffset:uint = tag.offset;
         SWFBytes.position = tagOffset;
         var count:uint = SWFBytes.readShort();

         for (var i:uint = 0; i< count; i++)
         {
              SWFBytes.readUnsignedShort();

              var char:uint = SWFBytes.readByte();
              var className:String = new String();

              while (char != 0)
              {
                   className += String.fromCharCode(char);
                   char = SWFBytes.readByte();
              }
              linkedSymbols.push ( className );
         }
    }
}

// outputs : Symbol1,Symbol1copy,parse_fla.MainTimeline
trace ( linkedSymbols );
```

Perfect, now we can now use those names to actually retrieve all linked classes defined in our current SWF. Now, it gets really valuable when loading runtime shared libraries containing linked classes that you need to all introspect for instance.

SWFExplorer is a library I wrote a little while ago relies on this code and works the following way :

```
var explorer:SWFExplorer = new SWFExplorer();

explorer.load ( new URLRequest ( "library.swf" ));

explorer.addEventListener ( SWFExplorerEvent.COMPLETE, assetsReady );

function assetsReady (e:SWFExplorerEvent):void
{

    // outputs : org.groove.Funk,org.funk.Soul,org.groove.Jazz
    trace( e.definitions );

    // outputs : org.groove.Funk,org.funk.Soul,org.groove.Jazz
    trace( e.target.getDefinitions() );

    // ouputs : 3
    trace( e.target.getTotalDefinitions() );

}
```

Internally, SWFExplorer relies on the same concept and almost same code, you can download the library here : http://www.bytearray.org/?p=175

Advanced SWF parsing

If you want to go further and do much more advanced parsing of SWF files, you can use the great library from Claus Wahlers called AS3SWF which allows you to completely dissecte a SWF with support for all SWF tags. Let's make a simple test. First, download AS3SWF at the following link : https://github.com/claus/as3swf

Once the SWC linked, create a a blank .fla document and use the following code on the timeline :

```
import com.codeazur.as3swf.SWF;

var swf:SWF = new SWF(root.loaderInfo.bytes);

trace(swf);
```

By running the code above, we get the following advanced description :

```
[SWF]
  Header:
    Version: 11
    FileLength: 197456
    FileLengthCompressed: 197456
    FrameSize: (550,400)
    FrameRate: 24
    FrameCount: 1
  Tags:
    [69:FileAttributes] AS3: true, HasMetadata: true, UseDirectBlit: false, UseGPU: false, UseNetwork: f
    [77:Metadata]  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="" xmlns:xmp="http://ns.adobe.com/xap/1.0/">
    <xmp:CreatorTool>Adobe Flash Professional CS5.5</xmp:CreatorTool>
    <xmp:CreateDate>2011-03-11T00:17:09-08:00</xmp:CreateDate>
    <xmp:MetadataDate>2011-03-11T00:17:58-08:00</xmp:MetadataDate>
    <xmp:ModifyDate>2011-03-11T00:17:58-08:00</xmp:ModifyDate>
  </rdf:Description>
  <rdf:Description rdf:about="" xmlns:dc="http://purl.org/dc/elements/1.1/">
    <dc:format>application/x-shockwave-flash</dc:format>
  </rdf:Description>
  <rdf:Description rdf:about="" xmlns:xmpMM="http://ns.adobe.com/xap/1.0/mm/"
xmlns:stRef="http://ns.adobe.com/xap/1.0/sType/ResourceRef#">
    <xmpMM:InstanceID>xmp.iid:CA4C9A390A20681188C6B9B04E99E63C</xmpMM:InstanceID>
    <xmpMM:DocumentID>xmp.did:CA4C9A390A20681188C6B9B04E99E63C</xmpMM:DocumentID>
    <xmpMM:OriginalDocumentID>xmp.did:C94C9A390A20681188C6B9B04E99E63C</xmpMM:OriginalDocumentID>
    <xmpMM:DerivedFrom rdf:parseType="Resource">
      <stRef:instanceID>xmp.iid:C94C9A390A20681188C6B9B04E99E63C</stRef:instanceID>
      <stRef:documentID>xmp.did:C94C9A390A20681188C6B9B04E99E63C</stRef:documentID>
      <stRef:originalDocumentID>xmp.did:C94C9A390A20681188C6B9B04E99E63C</stRef:originalDocumentID>
    </xmpMM:DerivedFrom>
  </rdf:Description>
</rdf:RDF>
    [09:SetBackgroundColor] Color: #FFFFFF
    [86:DefineSceneAndFrameLabelData]
      Scenes:
        [0] Frame: 0, Name: Scene 1
    [82:DoABC] Lazy: true, Length: 196066
    [76:SymbolClass]
      Symbols:
        [0] TagID: 0, Name: as3swc_fla.MainTimeline
    [01:ShowFrame]
    [00:End]
  Scenes:
    Name: Scene 1, Frame: 0
  Frames:
    [0] Start: 0, Length: 7
```

Now, let's add a single shape on the first frame of an .fla document and retry our code :

```
[SWF]
  Header:
    Version: 11
    FileLength: 197527
    FileLengthCompressed: 197527
    FrameSize: (550,400)
    FrameRate: 24
    FrameCount: 1
  Tags:
    [69:FileAttributes] AS3: true, HasMetadata: true, UseDirectBlit: false, UseGPU: false, UseNetwork: f
    [77:Metadata]  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="" xmlns:xmp="http://ns.adobe.com/xap/1.0/">
    <xmp:CreatorTool>Adobe Flash Professional CS5.5</xmp:CreatorTool>
    <xmp:CreateDate>2011-03-11T00:17:09-08:00</xmp:CreateDate>
    <xmp:MetadataDate>2011-03-11T00:20:42-08:00</xmp:MetadataDate>
    <xmp:ModifyDate>2011-03-11T00:20:42-08:00</xmp:ModifyDate>
  </rdf:Description>
  <rdf:Description rdf:about="" xmlns:dc="http://purl.org/dc/elements/1.1/">
    <dc:format>application/x-shockwave-flash</dc:format>
  </rdf:Description>
  <rdf:Description rdf:about="" xmlns:xmpMM="http://ns.adobe.com/xap/1.0/mm/"
xmlns:stRef="http://ns.adobe.com/xap/1.0/sType/ResourceRef#">
    <xmpMM:InstanceID>xmp.iid:CB4C9A390A20681188C6B9B04E99E63C</xmpMM:InstanceID>
    <xmpMM:DocumentID>xmp.did:CB4C9A390A20681188C6B9B04E99E63C</xmpMM:DocumentID>
    <xmpMM:OriginalDocumentID>xmp.did:C94C9A390A20681188C6B9B04E99E63C</xmpMM:OriginalDocumentID>
    <xmpMM:DerivedFrom rdf:parseType="Resource">
      <stRef:instanceID>xmp.iid:C94C9A390A20681188C6B9B04E99E63C</stRef:instanceID>
      <stRef:documentID>xmp.did:C94C9A390A20681188C6B9B04E99E63C</stRef:documentID>
      <stRef:originalDocumentID>xmp.did:C94C9A390A20681188C6B9B04E99E63C</stRef:originalDocumentID>
    </xmpMM:DerivedFrom>
  </rdf:Description>
</rdf:RDF>
    [09:SetBackgroundColor] Color: #FFFFFF
    [86:DefineSceneAndFrameLabelData]
      Scenes:
        [0] Frame: 0, Name: Scene 1
    [83:DefineShape4] ID: 1, ShapeBounds: (1250,7510,1090,6089), EdgeBounds: (1260,7500,1100,6079)
      FillStyles:
        [1] [SWFFillStyle] Type: 0 (solid), Color: ff000000
        [2] [SWFFillStyle] Type: 0 (solid), Color: ffcc9933
      LineStyles:
        [1] [SWFLineStyle2] Width: 20, StartCaps: round, EndCaps: round, Joint: round, Color: ff000000
      ShapeRecords:
        [SWFShapeRecordStyleChange] MoveTo: 7500,6079, FillStyle1: 2, LineStyle: 1
        [SWFShapeRecordStraightEdge] Horizontal: -6240
        [SWFShapeRecordStraightEdge] Vertical: -4979
        [SWFShapeRecordStraightEdge] Horizontal: 6240
        [SWFShapeRecordStraightEdge] Vertical: 4979
        [SWFShapeRecordEnd]
    [26:PlaceObject2] Depth: 1, CharacterID: 1, Matrix: (1,1,0,0,0,0)
    [82:DoABC] Lazy: true, Length: 196066
    [76:SymbolClass]
      Symbols:
        [0] TagID: 0, Name: as3swc_fla.MainTimeline
    [01:ShowFrame]
    [00:End]
  Scenes:
    Name: Scene 1, Frame: 0
  Frames:
    [0] Start: 0, Length: 9
      Defined CharacterIDs: 1
      Depth: 1, CharacterId: 1, PlacedAt: 5, IsKeyframe
```

Brilliant! You can see that there are no limits to what you can do with the `ByteArray` API, from simole object copy, to sound generation of complex parsing, there are lot of things to explore !

Now let's prepare for the next chapter entitled *Encoders* where we will dig into file encoders and unveil the mysteries behind custom file generation. So more cool stuff to come!