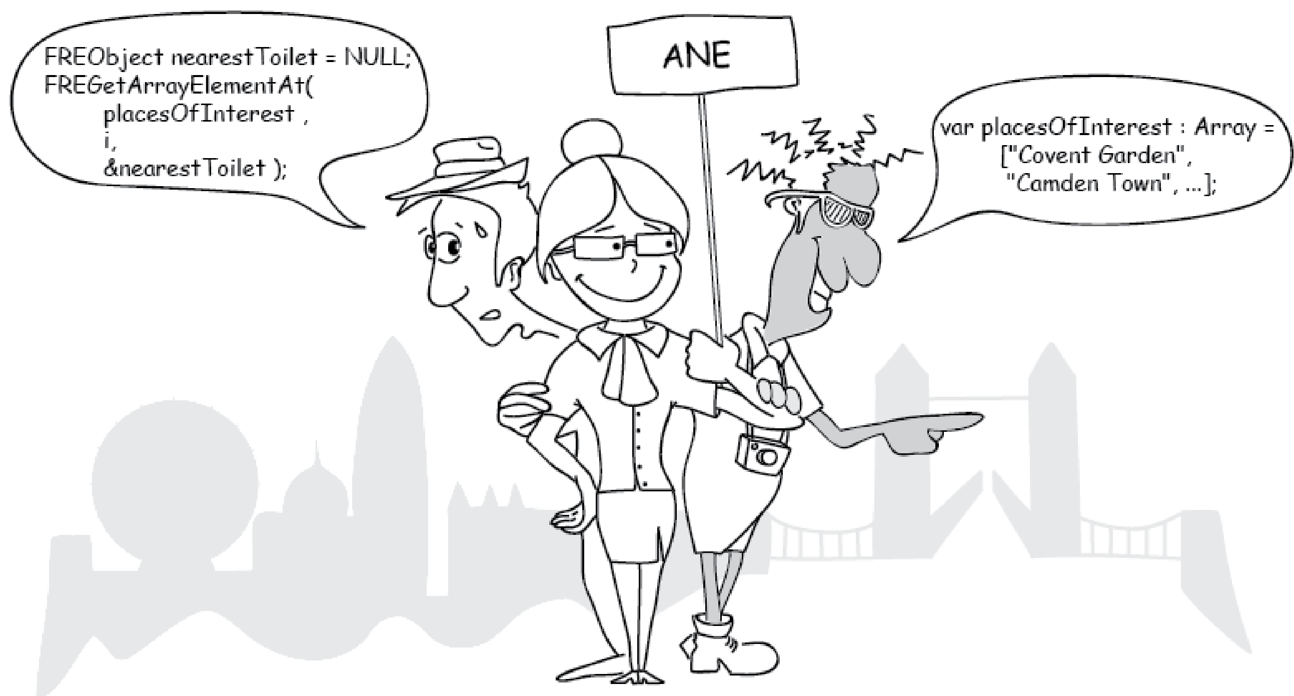
iOS vs. Action Script Data Types Guide

Keeping your cool when jumping between languages



© DiaDraw

iOS vs. ActionScript Data Types Guide

by Radoslava Leseva

Find us on the web at easynativeextensions.com

To report errors, please send a note to office@diadraw.com

Copyright © 2015 by DiaDraw

Project Editor: Hristo Lesev

Code Tester: Hristo Lesev

Cover and Interior Design: Radoslava Leseva

Proof Reader: Stephen Adams

Cover Illustration: Daniela Popova

Notice of Rights

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author. For information on getting permission for reprints or excerpts, contact office@diadraw.com.

Notice of Liability

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of the book, neither the author nor DiaDraw shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademarks

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and DiaDraw was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

Contents

Introduction 3

What readers of Easy Native Extensions say 4

What to expect 5

Got the code? 6

Set the scene 7

How ActionScript data is represented in C 9

What is void*? 9

How does void* help with AIR data types? 10

Quick preparation: add the headers 12

Auxiliary Functions 13

Sending events to ActionScript 13

Checking FREResult 17

Dealing with exceptions from FRE* functions 18

Changing the properties of ActionScript objects 20

Reading properties from ActionScript objects 22

Numeric and Boolean types 23

uint32_t 23

NSInteger 24

double 24

BOOL 24

NSNumber 24

Strings 25

Date and Time 25

Working with Objects 26

Containers: Dictionary, Set, Array, Vector 27

- Array 27
- Dictionary 27
- Set 27
- Vector 27
- When you know the data type 27

Dealing with byte arrays 28

Dealing with JSON 30

Handling NSError 30

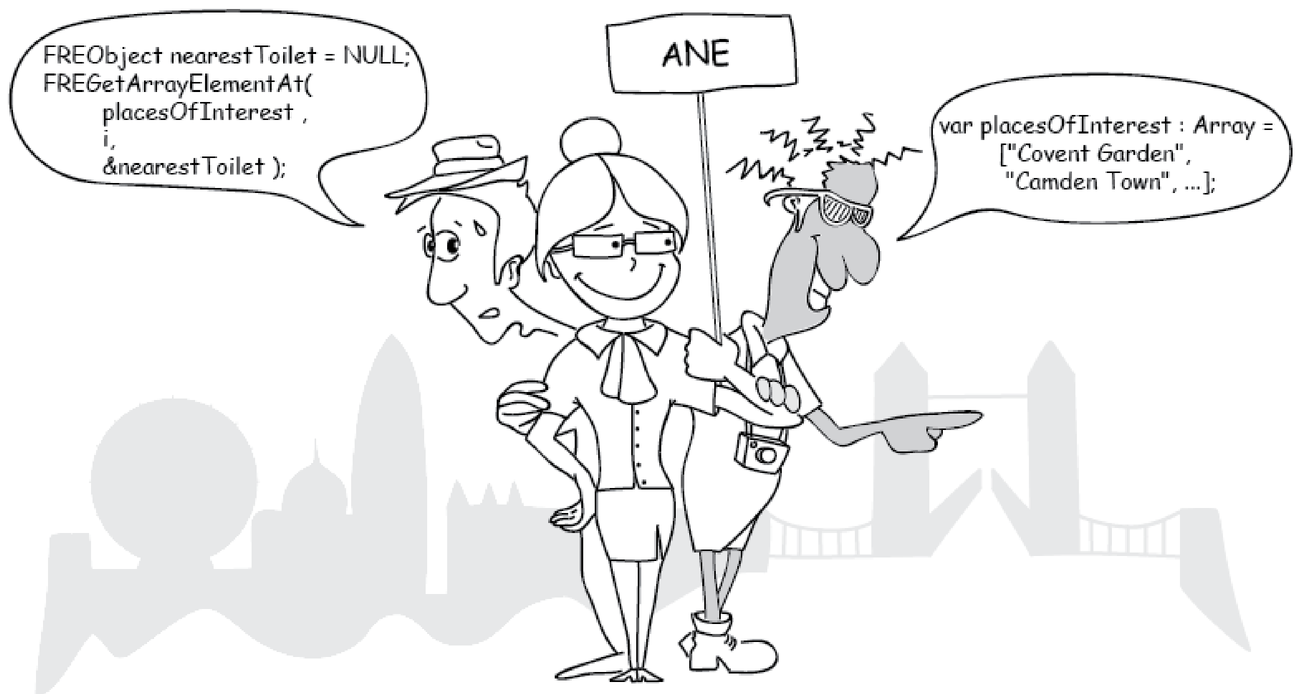
Image Data 31

Final step: check your header 33

Where to go from here 34

Introduction

So you took the cross-language rout of programming. It's fun, it provides variety and it often feels as if you are a tour guide for families visiting each other's countries. Done that in real life, trust me. You have several people tugging at your sleeve and asking things like "So what's that in Canadian dollars?" or "I don't speak ounces. Why won't they use the metric system?!" Usually all at the same time, as everyone knows it isn't impolite if you interrupt in a foreign language.



When it's two programming languages tugging at your sleeve, rather than well-meaning relatives, you find you initially spend most of your time working out how to translate different data types from one to the other and vice versa. The good thing about this is that it's usually a one-off exercise and you gradually build a small library of tricks and shortcuts that make your life easier and the development – faster with each new project.

The goal of this guide is to share with you the library of tricks and shortcuts that we have built and to put wind in your development sails even on your very first iOS + AIR project.

What readers of Easy Native Extensions say

" Well all I can say is that it was easily the best $30 I've spent in a while. I was back up and running in under a day and as an added bonus Radoslava does a brilliant job of detailing how to wrap up the whole build, packaging, and deployment process into a single Ant script. Her build script was much better than the crummy one I remember cobbling together for my original ANE attempts. One of the things I hadn't really tackled previously was learning how to properly debug my ANEs. Thankfully a companion debugging guide came bundled with the package, which takes you through the steps required to write a native iOS project that will wrap around a test AIR app. With that you'll be able to add breakpoints to Xcode and inspect your ANE's native library." Read more…

Christopher Caleb

Author of Flash iOS Apps Cookbook

www.yeahbutisitflash.com

"Your ebook on iOS native extensions (at least the part of it that I've read so far) is the best, clearest communication about software development that I've ever read. It is rare and completely refreshing to find programmers who can speak just as fluently and cleverly in their 'natural' language as in code, and who can also use elegant graphics to clear-up abstractions. Bravo, and thanks!"

Craig Umanoff

Moving Pictures

"@DiaDrawCom The book was great, very good tutorial and was easy to setup my custom ane. Thanks!"

@rudyvdblom

"Your eBook on ANE's is one of the best investments I have made. The excellent explanations and example code have saved me hours of trial-and-error. And I had planned to spend a couple of weeks developing custom email and dropbox extensions. Being able to download well-documented, ready-to-build source code gives me time to add extra features to my app. Very cool. Thanks."

Andrew Rapo

Quahog Entertainment

What to expect

At the end of this guide you will have:

1. A library of one-line functions, which you can call to convert most data types from ActionScript to Objective-C and vice versa.
2. An arsenal of additional helper functions to deal with exceptions, events, object properties, etc.
3. Understanding of how AIR data is represented in C, so you can extend your library should you ever need to.
4. Hours and days of saved time. So where would you like to spend it?

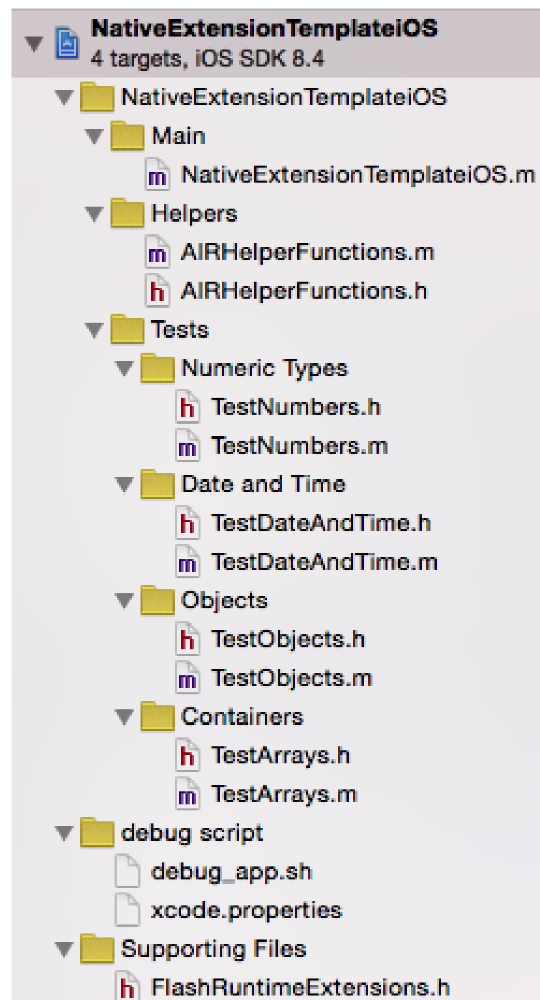
Got the code?

This eBook comes with accompanying code, which you can use in several different ways:

1. Test it as it is out of the package. I have added a few tests for you, which are exposed to ActionScript and all you need to do is call them. You will find these in the **Tests** group.

2. Ignore the code in the package and follow the steps in the book to build your own **AIRHelperFunctions** library. You can do that in the project you have created following the tutorials in our Easy Native Extensions main book or create a brand new set of helpers in another project you are working on.

3. Copy and paste **AIRHelperFunctions.m** and **AIRHelperFunctions.h** in a project you are working on for a client or for yourself. Import **AIRHelperFunctions.h** in your code and start using the helper functions.

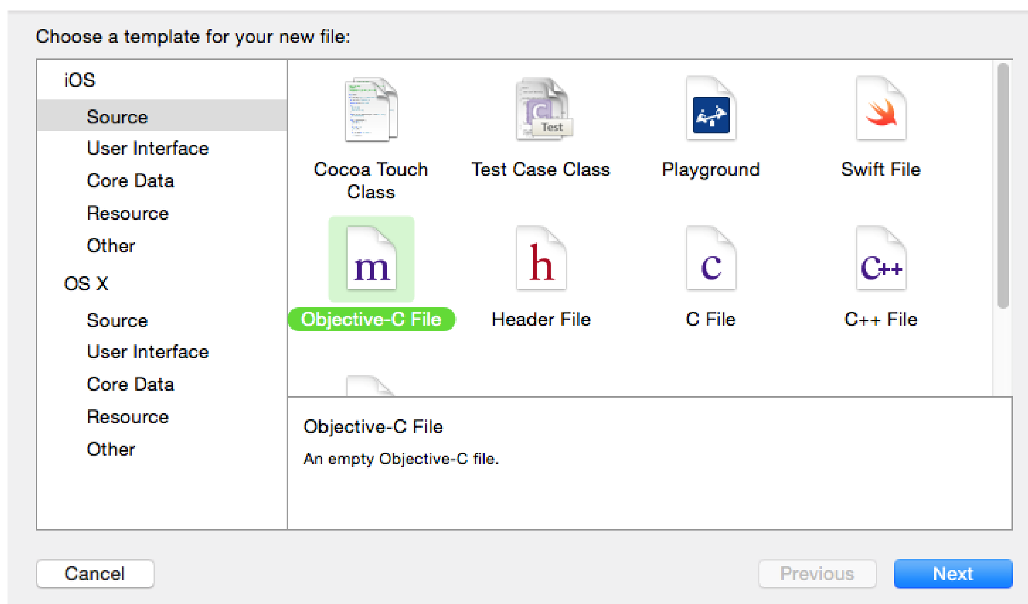


Set the scene

Let us start by adding a new group for helper files in your Xcode project. This step is more cosmetic than crucial, but helps keep your work organized.

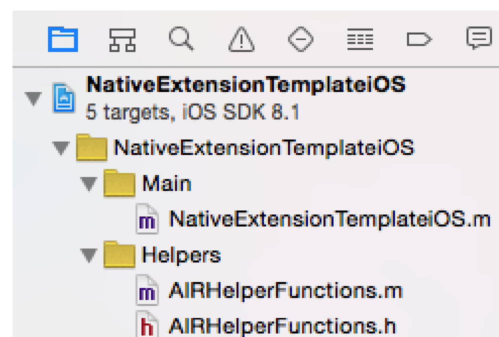
In Xcode select your native library project and do **File** > **New** > **Group**. Call the group **Helpers**. This will make a new virtual folder in your project. I am calling it 'virtual', because this will not create an actual folder on disk – you will need to do that by hand in Finder.

Now select the **Helpers** group and add a new file to it: **File** > **New** > **File**… Select **iOS** > **Source** > **Objective-C File** from the list of options:



Select **Empty file** from the next set of options, name your file **AIRHelperFunctions** and save it. Next, create add another file, a **Header File** this time, and give it the same name.

Both your helper files should now appear in the project tree under **Helpers**.



AIRHelperFunctions.m is where you will build a small library of functions for converting Objective-C and ActionScript data types. **AIRHelperFunctions.h** will make them visible to the rest of your code.

This is what the two files look like initially:

AIRHelperFunctions.m

```
//  AIRHelperFunctions.m
//  NativeExtensionTemplateiOS
//
//  Created by Yourself on today's date.
```

www.diadraw.com

```
//  Copyright (c) 2015 DiaDraw. All rights reserved.
//

#import <Foundation/Foundation.h>
```

AIRHelperFunctions.h

```
//  AIRHelperFunctions.h
//  NativeExtensionTemplateiOS
//
//  Created by Yourself on today's date.
//  Copyright (c) 2015 DiaDraw. All rights reserved.
//

#ifndef NativeExtensionTemplateiOS_AIRHelperFunctions_h
#define NativeExtensionTemplateiOS_AIRHelperFunctions_h


#endif
```

If you are new to any of the C language flavors: the #ifndef/#define/#endif construct above is called an #include guard. Including a header file with the #include directive in C is very similar to copying and pasting its contents into the file that includes it. That's all fine, but come linking time, if more than one other file included your header, any definitions from that header will appear in every file that included it and the linker will complain that 'such and such symbol has already been defined'. The #ifndef/#define/#endif prevent that: #ifndef checks if a macro (the name you put after the #ifndef) has already been defined and only allows the compiler to see the code between #ifndef and #endif if it hasn't been. #define defines the macro, so that the compiler will know about it after the first time it has reached that line of code (i.e. in the first file that includes your header).

Objective-C uses #import directives as well as #include directives. #import is supposed to be an improvement on #include by automatically preventing double inclusion.

Open **AIRHelperFunctions.m** and let us start adding magic to it.

How ActionScript data is represented in C

First, though, we need to have a word about how the ActionScript types are represented in the AIR C API, which is what you use when you make iOS Native Extensions for AIR.

Most of the data you will be passing around will be either an argument of a function you call or a result of a function you call.

When you implement a function on the native side which you want to be callable from ActionScript by doing

```
extensionContextInstance.call( “functionName”, argument_1, argument_2, … );
```

needs to implement the following signature:

```
FREObject functionName(
      FREContext ctx,
      void* funcData,
      uint32_t argc,
      FREObject argv[] )
```

You are already familiar with this signature from the chapter **Three ways to pass data from native code to ActionScript** in the **Easy Native Extensions** book. Flip back to it if you need a reminder.

The interesting thing here is FREObject, which we see in a couple of places:

- Arguments to the function arrive as an array of FREObject instances: this is what FREObject argv[] signifies.
- The function is expected to return a FREObject as a result.

If you open **FlashRuntimeExtensions.h** and look for the definition of FREObject, this is what you see:

```
typedef void * FREObject;
```

If you are familiar with any of the flavors of the C language, you probably know what this is and can jump to the practical stuff. Otherwise, read on.

What is void*?

This essentially means that **FREObject** is a **void pointer**. A pointer is a variable that holds a memory address. It can hold the memory address of something you expect to be of a particular type, for example:

```
int * pointerToInt;
```

Or it can hold a memory address of data which can be of any type and the type is not specified when the pointer is declared:

```
void * pointerToAnyType;
```

When you read the data located at the memory address which **pointerToAnyType** points to, you can interpret it as an **int**, as a **string** or as something else.

When you pass data from ActionScript to Objective-C, you can put pretty much anything in your list of arguments:

```
var argument_1 : int = 2;
var argument_2 : String = "How you doin'?";
extensionContextInstance.call( "functionName", argument_1, argument_2, … );
```

Both of these will arrive at the C side as **FREObject**, in other words as **void ***.

How does void* help with AIR data types?

The Adobe AIR C API gives you convenience functions that help convert a **FREObject** into the native type you expect. You can find a full list here.

To get the integer value out of the first argument, for example, you will need to do the following:

```
int32_t nativeInt = 0;
FREResult success = FREGetObjectAsInt32( argv[ 0 ], &nativeInt );

// FREResult is an enumerator which tells us whether the call succeeded.
// It may fail because argv[ 0 ] was not the type we expected,
// was not supplied by ActionScript at all,
// we made the call on the wrong thread, etc.
if ( FRE_OK == success )
{
    // we've got the int value successfully
}
```

Getting an Objective-C **NSString** out of the second argument requires a little more work:

```
uint32_t strLength = 0;
const uint8_t * strValue = NULL;
// This call gives us a pointer to a C-style string (an array of 8-bit characters)
// and how long it is:
FREResult success = FREGetObjectAsUTF8( argv[ 1 ], &strLength, &strValue );

// If all went well with the previous call,
// we can try and convert the C-style string to an Objective-C NSString object:
NSString * nativeString = NULL;
if ( ( FRE_OK == success ) && ( 0 < strLength ) && ( NULL != strValue ) )
{
    nativeString = [ NSString stringWithUTF8String: ( const char * ) strValue ];
}
```

Dealing with image data pointed to by a **FREObject** involves even more elaborate checks and conversions.

You can see how the work stacks up. Instead of adding six lines of code every time you want to read a string argument, how about making it a one-liner:

```
NSString * value = getNSStringFromFREObject( objectAS );
```

This is what you are going to build over the next few chapters: your own little library of one-liner helper functions that will be at your fingertips to save you time with any iOS ANE project.

Quick preparation: add the headers

You will be dealing with almost all of AIR's C API, so let us import its header into your header file. You will also need your code to know about most of the fundamental types in Objective-C. Open **AIRHelperFunctions.h** and add an import statement for **FlashRuntimeExtensions.h** and **Foundation.h**:

AIRHelperFunctions.h

```
//  AIRHelperFunctions.h
//  NativeExtensionTemplateiOS
//
//  Created by Yourself on today's date.
//  Copyright (c) 2015 DiaDraw. All rights reserved.
//

#ifndef NativeExtensionTemplateiOS_AIRHelperFunctions_h
#define NativeExtensionTemplateiOS_AIRHelperFunctions_h

#import <Foundation/Foundation.h>
#import "FlashRuntimeExtensions.h

#endif
```

Why import **FlashRuntimeExtensions.h** in **AIRHelperFunctions.h**, instead of **AIRHelperFunctions.m**? AIRHelperFunctions.m will have definitions of the functions that will do conversions from, say NSString to FREObject and AIRHelperFunctions.h will list the signatures of these functions, so they are visible to anyone who needs to call them. To make them visible to another file, you will need to import AIRHelperFnctions.h in it. This way anything that AIRHelperFunctions.h imports/includes will also be visible to the file that imported it and you won't have to add another import everywhere.

The first file where you will import **AIRHelperFunctions.h** is **AIRHelperFunctions.m**:

AIRHelperFunctions.m

```
//  AIRHelperFunctions.m
//  NativeExtensionTemplateiOS
//
//  Created by Yourself on today's date.
//  Copyright (c) 2015 DiaDraw. All rights reserved.
//
#ifndef NativeExtensionTemplateiOS_AIRHelperFunctions_h
#define NativeExtensionTemplateiOS_AIRHelperFunctions_h

#import <AIRHelperFunctions.h>

#endif
```

Since **AIRHelperFunctions.h** already imports **Foundation.h**, you don't need a second import of it in **AIRHelperFunctions.m**.

Auxiliary Functions

We will start at the deep end. Imagine groping your way to a pool's deep end to leave a few props that will help you swim towards it afterwards. Distant memory? Oh well, a bit too fresh for me, I only learned how to swim in 2014…

Enough with the metaphors. What you will do is this: define a handful of auxiliary functions that will make your life even easier when you start writing the helpers.

You will be adding code to **AIRHelperFunctions.m**, right after the **#import** directives.

Sending events to ActionScript

Sending an asynchronous event is one of the three ways of passing data from native code back to ActionScript. You may remember that from the chapter called **Three ways to pass data from native code to ActionScript** in the Easy Native Extensions book.

Sending an event is the only asynchronous way of passing data back and the only C API AIR call you can make on any thread other than the main one. It is your way of signaling to your ActionScript code when something has gone wrong or when the result of an asynchronous operation is ready, for example retrieving user names from a server.

What effectively happens is this:

1. You subscribe to status events delivered to your **ExtensionContext** instance in ActionScript:

ActionScript

```
m_extensionContext.addEventListener( StatusEvent.STATUS, onStatusEvent );
…
private function onStatusEvent( _event : StatusEvent ) : void
{
      trace( _event.code );
      trace( _event.level );
}
```

Then you call **FREDispatchStatusEventAsync** in your native code whenever you need to send an event that will be caught by that listener:

Objective-C

```
FREDispatchStatusEventAsync(
        ctx,
        ( const uint8_t * ) "This is the message CODE",
        ( const uint8_t * ) "This is the message LEVEL" );
```

FREDispatchStatusEventAsync takes a **FREContext** - a pointer[1] to the extension context on the native side and two string values, which are mapped to the **StatusEvent code** and **level** properties respectively:

```
FREResult FREDispatchStatusEventAsync(
        FREContext     ctx,
        const uint8_t* code,
        const uint8_t* level
);
```

So the first auxiliary function you add will be one that lets you send an event/message to ActionScript from anywhere without having to worry about supplying an extension context or converting strings to `const uint8_t*`.

For this purpose you will need your auxiliary function to have access to a context instance. I tend to make a copy of the one passed to the extension context initializer and keep it in my ANE's main native file as a global variable. Makes sense to use that instead of having another copy lurking around. In the declaration below the **extern** keyword makes that global copy available to the code in **AIRHelperFunctions.m**:

```
#pragma extern declarations
extern FREContext g_ctx;
```

Since we are in a declaring mood (or is it just me?), let's declare a few event types too:

```
#pragma event types
const NSString * MSG_ERROR          = @"ERROR";
const NSString * MSG_WARNING        = @"WARNING";
const NSString * MSG_INFO           = @"INFO";
```

[1] **FREContext is another void *.**

And finally, here is the auxiliary function you've been waiting for:

```
#pragma sending events to ActionScript
/**
 * Sends a flash.events.SatusEvent to ActionScript.
 * The dispatch happens asynchronously.
 *
 * @param messageType This will be mapped to StatusEvent's code property.
 *
 * @param message This will be mapped to StatusEvent's level property and
 *                can contain any information about the event in string form.
 */
void sendMessage( const NSString * const messageType, const NSString * const message )
{
    // Having a messageType is important, but the message itself can be left empty.
    static const NSString * MSG_DEFAULT_EVENT  = @"STATUS_EVENT";
    assert( messageType );

    if ( NULL != message )
    {
       FREDispatchStatusEventAsync( g_ctx, (uint8_t *) [messageType UTF8String],
                                    (uint8_t *) [message UTF8String] );
    }
    else
    {
       FREDispatchStatusEventAsync( g_ctx, (uint8_t *) [MSG_DEFAULT_EVENT UTF8String],
                                   (uint8_t *) [messageType UTF8String] );
    }
}
```

Now you can do this much shorter call in any part of your code:

```
sendMessage( MSG_ERROR, @”An error happened” );
```

What's with the **assert**?

We get asked that a lot. This concept was hammered in my head when I was a junior programmer, so I feel kind of old every time I am asked to explain it. But hey…

An **assert** is usually defined as a "debug-only macro that aborts execution if its argument is false"[2]. In other languages, like ActionScript, where you don't have macros, you can use other techniques to achieve the same effect. The idea is that you use an **assert** as a diagnostic tool to help you narrow down runtime problems in debug builds. If a function asserts that the arguments passed to it are valid (non-NULL, within an expected range or whatever else constitutes valid in the context of the function), you will know as soon as something dodgy comes in, because your app will crash inside that function in debug. Without the **assert** the function may not crash, but instead pass the bad data down the line, making the chain of responsibility, and thus the time it takes you to get to the source of the bad data, longer.

You've got a little more work to do, in order to make this function available for use in other files: add its signature to **AIRHelperFunctions.h** right after the **#import** directives:

AIRHelperFunctions.h

```
//  AIRHelperFunctions.m
//  NativeExtensionTemplateiOS
//
//  Created by Yourself on today's date.
//  Copyright (c) 2015 DiaDraw. All rights reserved.
//
#ifndef NativeExtensionTemplateiOS_AIRHelperFunctions_h
#define NativeExtensionTemplateiOS_AIRHelperFunctions_h

#import <Foundation/Foundation.h>
#import "FlashRuntimeExtensions.h"

void sendMessage( const NSString * const messageType, const NSString * const message );

#endif
```

You will need to do that for any function in **AIRHelperFunctions.m** that you want to be able to call from other files. I am hoping you will remember to do this with the functions we are about to start defining. But just in case, I will remind you at the end of the book what your header file should look like.

[2] Steve Maguire, *Writing Solid Code* (Redmond, Washington 1993), 16

Checking FREResult

If you have a peek inside **FlashRuntimeExtensions.h**, you will notice that most AIR C API functions return **FREResult**, which looks like this:

```
typedef enum {
    FRE_OK                  = 0,
    FRE_NO_SUCH_NAME        = 1,
    FRE_INVALID_OBJECT      = 2,
    FRE_TYPE_MISMATCH       = 3,
    FRE_ACTIONSCRIPT_ERROR  = 4,
    FRE_INVALID_ARGUMENT    = 5,
    FRE_READ_ONLY           = 6,
    FRE_WRONG_THREAD        = 7,
    FRE_ILLEGAL_STATE       = 8,
    FRE_INSUFFICIENT_MEMORY = 9,
    FREResult_ENUMPADDING   = 0xfffff /* will ensure that C and C++ treat
                                         this enum as the same size. */
} FREResult;
```

If you are new to any of the C language flavors, an **enum** is a data type, which holds together a set of named values, usually numeric. When an AIR C API function returns FRE_WRONG_TRHEAD, it's the same as returning the number 7, but a bit more meaningful.

Having a function that checks that result for you and does something when there is an error, i.e. anything different from **FRE_OK**, will save you having to repeat those checks over and over again. The **isFREResultOK** function defined below checks whether the **FREResult** we've got is an error and if it is, it sends an error event back to ActionScript and reports the error in Xcode's debug console:

```
#pragma report errors
/**
 * Checks if we have received an error code.
 * If yes, sends an error event to ActionScript
 * and logs it in the Xcode debugger's console.
 */
BOOL isFREResultOK( FREResult errorCode, NSString * errMessage )
{
    if ( FRE_OK == errorCode )
    {
        // No error code, things went well. Nothing to report.
        return TRUE;
    }

    // Construct a string that will have the error code in it:
    NSString * messageToReport =
            [ NSString stringWithFormat: @"%@ %d", errMessage, errorCode ];

    // Now log the message in the debug console:
    NSLog( @"%@", messageToReport );

    // And send it as an event to ActionScript:
```

www.diadraw.com

```
    sendMessage( MSG_ERROR, messageToReport );

    return FALSE;
}
```

Dealing with exceptions from FRE* functions

Another thing that may go wrong when you call an AIR C API function is that an exception may be thrown. You will notice that some functions take a parameter like this:

```
FREObject * thrownException
```

Getting information about the exception takes a few lines of code, which you probably don't want to repeat over and over either. The **hasThrownException** auxiliary function will save you the need to. It checks if an exception was indeed thrown and if one was, tries to get information about it and pass it back to ActionScript:

```
#pragma exceptions
/**
 * Checks if a call to a FRE* function threw an exception
 * and if an exception is caught, sends an error event to ActionScript.
 *
 * @param thrownException The exception instance.
 *
 * @return TRUE if an exception was caught, FALSE otherwise
 */
BOOL hasThrownException( FREObject thrownException )
{
    if ( NULL == thrownException )
    {
        // No exception was thrown. All clear.
        return FALSE;
    }

    // Check if we caught a valid exception object first:
    FREObjectType objectType;
    if ( FRE_OK != FREGetObjectType( thrownException, &objectType ) )
    {
      sendMessage( MSG_ERROR,
                  @"Exception was thrown,
                  but failed to obtain information about its type" );
        return TRUE;
    }

    // If the exception object is valid,
    // then get the exception information as a string:
    if ( FRE_TYPE_OBJECT == objectType )
    {
        FREObject exceptionTextAS = NULL;
        FREObject newException = NULL;

        if ( FRE_OK != FRECallObjectMethod(thrownException,
                       (const uint8_t *) "toString",
                       0,
                       NULL,
                       &exceptionTextAS,
                       &newException ) )
        {
            sendMessage( MSG_ERROR,
                         @"Exception was thrown,
                         but failed to obtain information about it" );
            return TRUE;
        }

        // Send an event to ActionScript with information on the exception:
        NSString * exceptionText = getNSStringFromFREObject( exceptionTextAS );
        sendMessage( MSG_ERROR,
                     [ NSString stringWithFormat: @"exception: %s",
                                 ( uint8_t * ) [ exceptionText UTF8String ] ] );
        return TRUE;
    }

    return FALSE;
}
```

If you are new to exceptions: you can think of an exception as an event. When thrown or dispatched, it disrupts the normal flow of code. For example: if a call to AIR in the middle of one of your functions throws an exception, the rest of the code in that function will not be executed. Execution will instead be passed to an exception handler if one is found in your function or the function that called it or the function that called the one that called it… If no exception handler is found, your app will likely be terminated. An exception handler is usually defined by a try-catch statement.

Changing the properties of ActionScript objects

The AIR C API strives to make most of what you can do in ActionScript possible to replicate in C. For example, if you have an object of type **Date**, named **date** and wanted to set its **day** property, you can do that. Only instead of the ActionScript one-liner

```
date.day = 1;
```

you need to write a few more lines and to a few checks to achieve the same in C. This is what the next function helps you with: set an arbitrary property of an arbitrary object, so long as you have a pointer to the object and know the property name.

Note that both the object whose property you are changing and the value you are changing to need to be **FREObject**s.

The call that does the actual job here is AIR C API's FRESetObjectProperty.

```
#pragma properties
/**
 * Sets a named property of an ActionScript object.
 * Calling setFREObjectProperty( asEvent, ( const uint8_t * ) "type", eventTypeAS );
 * where asEvent and eventTypeAS are FREObject instances,
 * is like calling event.type = "SOMETHING"; in ActionScript.
 *
 * @param objAS The ActionScript object.
 *
 * @param propertyName The name of the property to be set.
 *
 * @param propertyValue The value the property should be set to.
 *
 */
void setFREObjectProperty(
      FREObject objAS,
      const uint8_t * propertyName,
      FREObject propertyValue )
{
    assert( NULL != objAS );
    assert( NULL != propertyName );
    assert( NULL != propertyValue );

    FREObject thrownException = NULL;
    FREResult status =
```

```
      FRESetObjectProperty( objAS, propertyName, propertyValue, &thrownException );

    // Check if we encountered an error:
    isFREResultOK( status, @"Could not set an ActionScript object's property" );
    hasThrownException( thrownException );
}
```

Like this sample?

Get the full book and accompanying code here.

Numeric and Boolean types

Now that you have dipped in the deep end this whole chapter will seem easy breezy. We will start the helper functions with the simplest types: numeric and Boolean.

uint32_t

A lot of the time you need to deal with unsigned integers, so let us add a function that converts an ActionScript **int** or **Number** in the form of **FREObject** into a **uint32_t**. A **uint32_t** is what it says on the tin: a 32-bit unsigned integer type.

Open **AIRHelperFunctions.m** and add the helper function. The main thing here is the call to AIR C API's FREGetObjectASUint32.

```
#pragma uint32_t and FREObject
/**
 * Converts a FREObject to uint32_t.
 */
uint32_t getUInt32FromFREObject( FREObject uintAS )
{
    assert( NULL != uintAS );

    uint32_t result = 0;
    FREResult status = FREGetObjectAsUint32( uintAS, &result );
    isFREResultOK( status, @"Could not convert FREObject to uint32_t." );

    return result;
}
```

Next, add a helper function that does the opposite and converts an unsigned integer to **FREObject** – useful when returning results from **FREFunction**s.

```
/**
 * Converts a uint32_t to FREObject.
 */
FREObject getFREObjectFromUInt32( uint32_t val )
{
    // Create a FREObject
    FREObject valAS = NULL;

    // Then assign a numeric value to it
    FREResult status = FRENewObjectFromUint32( val, &valAS );
    isFREResultOK( status, @"Could not convert uin32_t to FREObject." );

    return valAS;
}
```

NSInteger

double

BOOL

NSNumber

Like this sample?

Get the full book and accompanying code here.

Strings
Date and Time

Like this sample?

Get the full book and accompanying code here.

Working with Objects

Converting date and time data already showed you how to deal with object types for which the AIR C API doesn't provide direct support. How about converting an arbitrary ActionScript type or object into an arbitrary Objective-C type?

The pair of functions below demonstrates how to do that. They are useful in situations where you don't necessarily know in advance the type of objects you will be dealing with: for example, when you have an array of values or objects passed by ActionScript to an Objective-C function.

The Objective-C type that can refer to an instance of any type or class is **id**. It is almost like **void***, but unlike **void***, which is just a pointer to a chunk of memory where there could be anything, typed or untyped, **id** usually refers to an Objective-C type.

Thus our first function will take a **FREObject**, check its type, create the corresponding Objective-C type and pass it back as **id**. The AIR nugget here is FREGetObjectType: a function that tells you what a **FREObject** is currently wrapping. Its result (returned as an output parameter) is one of the following:

```
typedef enum {
    FRE_TYPE_OBJECT           = 0,
    FRE_TYPE_NUMBER           = 1,
    FRE_TYPE_STRING           = 2,
    FRE_TYPE_BYTEARRAY        = 3,
    FRE_TYPE_ARRAY            = 4,
    FRE_TYPE_VECTOR           = 5,
    FRE_TYPE_BITMAPDATA       = 6,
    FRE_TYPE_BOOLEAN          = 7,
    FRE_TYPE_NULL             = 8,
    FREObjectType_ENUMPADDING = 0xfffff /* will ensure that C and C++
                                           treat this enum as the same size. */
} FREObjectType;
```

The first helper function, **getIdObjectFromFREObject** takes each of these possible types in turn and creates a corresponding Objective-C type. There are a couple of things to note here:

Like this sample?

Get the full book and accompanying code here.

Containers:
Dictionary, Set, Array, Vector

Array

Dictionary

Set

Vector

When you know the data type

Like this sample?

Get the full book and accompanying code here.

Dealing with byte arrays

Being able to pass chunks of bytes between your app and your ANE is useful for dealing with, say pixel information, contents of files or anything else you may want to encode in an array of bytes.

The AIR C API offers a set of structures and functions to help you deal with that:

- FREByteArray is a structure, which holds two pieces of information: a pointer to the memory location of the beginning of the byte array and a number indicating how many bytes there are. It maps to ActionScript's **ByteArray** class and looks like this in C:

```
typedef struct {
    uint32_t length;
    uint8_t* bytes;
} FREByteArray;
```

- The function FREAcquireByteArray gives you exclusive access to the bytes in the byte array. Exclusive in this case means that while you have hold of the bytes nothing else can modify them. The price you pay for this however is that you aren't allowed to make any other AIR C API calls until you have released your access with the next function.

- The function FREReleaseByteArray releases the bytes in the byte array and makes them accessible to other calls. Always make sure that a call to FREAcquireByteArray is paired with FREReleaseByteArray.

The helper function you are going to implement that reads the bytes from a **FREByteArray** into an Objective-C **NSData** class puts these two functions and the structure to use. Note that the constructor **dataWithBytes** makes a copy of the bytes.

Tip: **NSData** is an object-oriented wrapper for a byte buffer.

```
#pragma NSData and FREObject (FREByteArray)
/**
 * Converts an ActionScript ByteArray into native NSData.
 */
NSData * getNSDataFromFREByteArray( FREObject byteArrayObjectAS )
{
    // Get hold of the ByteArray object:
    FREByteArray byteArrayAS;
    FREResult status = FREAcquireByteArray( byteArrayObjectAS, &byteArrayAS );

    if ( !isFREResultOK( status, @"Could not acquire ByteArray." ) )
    {
        // Optional: Send an error event to ActionScript
        return NULL;
```

Like this sample?

Get the full book and accompanying code here.

Dealing with JSON

Handling NSError

Like this sample?

Get the full book and accompanying code here.

Image Data

Handling image data between iOS and ActionScript is quite useful in various situations: when you want to capture frames from the native camera and use them in ActionScript, for example. Or in the other direction: for passing images from ActionScript to native code to compose a video like our Screen Recorder ANE does.

A lot of the time you can treat the image data as a series of bytes and use the helpers you defined in the Dealing with byte arrays chapter. It is also useful to be able to handle **BitmapData**, especially when you want to pass **BitmapData** objects from AIR to iOS.

Here the helper takes a **BitmapData** object, converts is to **NSData** and waits for the caller to process the **NSData** before releasing the **BitmapData**. A bit of a paranoid approach, admittedly. For that purpose the caller must supply a callback for the native pixel data to be passed into for processing: **ProcessBytesCallback**.

```
#pragma BitmapData and NSData
/**
 * Copies the byes from an ActionScript BitmapData object into an NSData object.
 * Note that the result is delivered in a callback function supplied by the caller.
 * @param bitmapDataObj The ActionScript BitmapData object.
 * @param callback The function which will receive the result.
 */
void getNSDataFromFREObjectBitmapData(
      FREObject bitmapDataObj, ProcessBytesCallback callback )
{
    assert( NULL != bitmapDataObj );
    FREBitmapData frameBitmapData;

    FREResult status = FREAcquireBitmapData( bitmapDataObj, &frameBitmapData );
    assert( FRE_OK == status );

    int nComponents = 4; // ARGB
    int bytesPerRow = nComponents * frameBitmapData.lineStride32;
    NSUInteger totalBytes = bytesPerRow * frameBitmapData.height;

    NSData * bitmapData =
      [ NSData dataWithBytes: frameBitmapData.bits32 length: totalBytes ];

    // Here we call the callback function which will deal with the bytes in NSData
    // before we release the BitmapData object:
    callback( bitmapData );

    // Now release the BitmapData object:
    status = FREReleaseBitmapData( bitmapDataObj );
    assert( FRE_OK == status );
}
```

Now about that callback… It's a function that takes a pointer to **NSData** and doesn't return a result. The callback is where you do your native magic on the **BitmapData** pixels. It's best to define the callback's signature in the header file, so that the users of your helper function can see it. Here is the syntax:

AIRHelperFunctions.h

```
typedef void( ^ProcessBytesCallback )( NSData * byteData );
```

The next helper allows you to do the same when, instead of **BitmapData**, you have the image data in a **ByteArray** on the AIR side. Note that the result is again delivered in a callback. Hey, just because you are paranoid, it doesn't mean they aren't after you, as Joseph Heller put it…

Like this sample?

Get the full book and accompanying code here.

Final step: check your header

Like this sample?

Get the full book and accompanying code here.

Where to go from here

Check out the free online tutorials on EasyNativeExtensions.com and have a look at the free and commercial ANEs in our shop.

Thank you!

Thank you for picking up this **iOS Data Guide** and investing the time and money in it! We hope it will save you hours of work on more than one project and get the quality of your work to the highest level.

If you have a few minutes, we would love to hear what you thoughts about it.

Leave a comment on our blog or drop us an e-mail at office@diadraw.com.

Other ways to get in touch and share

@DiaDrawCom on Twitter

https://www.facebook.com/DiaDraw on Facebook

DiaDraw on LinkedIn

Thanks again and may the source be with you!