Native extensions for the standalone Dart VM

Written by William Hesse
May 2012

The extension mechanism discussed in this page is for deep integration of the VM. If you just need to call existing code written in C or C++, see C & C++ interop using FFI.

> The example code on this page has some known issues.

Dart programs running on the standalone Dart VM (*command-line apps*) can call C or C++ functions in a shared library, by means of native extensions. This article shows how to write and build such native extensions on Windows, macOS, and Linux.

You can provide two types of native extensions: asynchronous or synchronous. An *asynchronous extension* runs a native function on a separate thread, scheduled by the Dart VM. A *synchronous extension* uses the Dart virtual machine library's C API (the Dart Embedding API) directly and runs on the same thread as the Dart isolate. An asynchronous function is called by sending a message to a Dart port, receiving the response on a reply port.

Anatomy of a native extension

A Dart native extension contains two files: a Dart library and a native library. The Dart library defines classes and top-level functions as usual, but declares that some of these functions are implemented in native code, using the **native** keyword. The native library is a shared library, written in C or C++, that contains the implementations of those functions.

The Dart library specifies the native library using an `import` statement and the **dart-ext**: URI scheme. As of 1.20, the URI must either be an absolute path like `dart-ext:/path/to/extension`, or only the name of the extension, like `dart-ext:extension`. The VM modifies the URI to add platform specific prefixes and suffixes to the extension name. For example, `extension` becomes `libextension.so` on Linux. If the URI is an absolute path, the import fails if the file does not exist. If the URI is only the name of the extension, the VM first looks for the file adjacent to the importing Dart library. If it is not found there the VM passes the file name to the platform specific call for loading dynamic libraries (e.g. `dlopen` on Linux), which is free to follow its own search procedure.

Example code

The code for the sample extensions featured in this article is in the samples/sample_extension directory of the Dart repository.

The sample extensions call the C standard library's rand() and srand() functions, returning pseudorandom numbers to a Dart program. Because the native parts of the asynchronous and synchronous native extensions share most of their code, a single native source file (and resulting shared library) implements both extensions. The two extensions have separate Dart library files. Two additional Dart files provide examples of using and testing the asynchronous and synchronous extensions.

The shared library (native code) for the extensions shown in this article is called sample_extension. Its C++ file, sample_extension.cc, contains six functions that are called from Dart:

sample_extension_Init():
Called when the extension is loaded.

ResolveName():
Called the first time a native function with a given name is called, to resolve the Dart name of the native function into a C function pointer.

SystemRand() and SystemSrand():
Implement the synchronous extension. These are native functions called directly from Dart, and that call rand() and srand() from the C standard library.

wrappedRandomArray() and randomArrayServicePort():
Implement the asynchronous extension. randomArrayServicePort() creates a native port and associates it with wrappedRandomArray(). When Dart sends a message to the native port, the Dart VM schedules wrappedRandomArray() to run on a separate thread.

Some of the code in the shared library is setup and initialization code, which can be the same for all extensions. The functions sample_extension_Init() and ResolveName() should be almost the same in all extensions, and a version of randomArrayServicePort() must be in all asynchronous extensions.

The synchronous sample extension

Because the asynchronous extension works like a synchronous extension with some added functions, we'll show the synchronous extension first. First we'll show the Dart part of the extension and the sequence of function calls that happen when the extension is loaded. Then we explain how to use the Dart Embedding API, show the native code, and show what happens when the extension is called.

Here is the Dart part of the synchronous extension, called **sample_synchronous_extension.dart**:

```
library sample_synchronous_extension;

import 'dart-ext:sample_extension';

// The simplest way to call native code: top-level functions.
int systemRand() native "SystemRand";
bool systemSrand(int seed) native "SystemSrand";
```

The code that implements a native extension executes at two different times. First, it runs when the native extension is loaded. Later, it runs when a function with a native implementation is called.

Here is the sequence of events at load time, when a Dart app that imports sample_synchronous_extension.dart starts running:

1. The Dart library sample_synchronous_extension.dart is loaded, and the Dart VM hits the code **`import 'dart-ext:sample_extension'`**.
2. The VM loads the shared library 'sample_extension' from the directory containing the Dart library.
3. The function sample_extension_Init() in the shared library is called. It registers the shared library function ResolveName() as the name resolver for all native functions in the library sample_extension.dart. We'll see what the name resolver does when we look at synchronous native functions, below.

> **Note:** The filename of the shared library depends on the platform. On Windows, the VM loads sample_extension.dll, on Linux it loads libsample_extension.so, and on Mac it loads libsample_extension.dylib. We show how to build and link these shared libraries in an appendix at the end of the article.

Using the Dart Embedding API from native code

As the sample extensions show, the native shared library contains an initialization function, a name resolution function, and the native implementations of functions declared as native in the Dart part of the extension. The initialization function registers the native name resolution function as responsible for looking up native function names in this library. When a function declared as **`native "function_name"`** in the Dart library is called, the native library's name resolution function is called with the string "*function_name*" as an argument, plus the number of arguments in the function call. The name resolution function then returns a function pointer to the native implementation of that function. The initialization function and the name resolution function look pretty much the same in all Dart native extensions.

The functions in the native library use the Dart Embedding API to communicate with the VM, so the native code includes the header **dart_api.h**, which is in the SDK at dart-sdk/include/dart_api.h or in the repository at runtime/include/dart_api.h. The Dart Embedding API is the interface that embedders use to include the Dart VM in a web browser or in the standalone VM for the command line. It consists of about 100 function interfaces and many data type and data structure definitions. These are all shown, with comments, in dart_api.h. Examples of using them are in the unit test file runtime/vm/dart_api_impl_test.cc.

A native function to be called from Dart must have the type **Dart_NativeFunction**, which is defined in dart_api.h as:

```
typedef void (*Dart_NativeFunction)(Dart_NativeArguments arguments);
```

So a Dart_NativeFunction is a pointer to a function taking a Dart_NativeArguments object, and returning no value. The arguments object is a Dart object accessed by API functions that return the number of arguments, and return a Dart_Handle to the argument at a specified index. The native function returns a Dart object to the Dart app, as the return value, by storing it in the arguments object using the Dart_SetReturnValue() function.

Dart handles

The extension's native implementations of functions use Dart_Handles extensively. Calls in the Dart Embedding API return a Dart_Handle and often take Dart_Handles as arguments. A Dart_Handle is an opaque indirect pointer to an object on the Dart heap, and Dart_Handles are copied by value. These handles remain valid even when a garbage-collection phase moves Dart objects on the heap, so native code must use handles to store references to heap objects. Because these handles take resources to store and maintain, you must free them when they're no longer used. Until a handle is freed, the VM's garbage collector cannot collect the object it points to, even if there are no other references to it.

The Dart Embedding API automatically creates a new scope to manage the lifetime of handles in a native function. A local handle scope is created when the native function is entered, and is deleted when the function is exited. The scope is deleted if the function exits with PropagateError, as well as if it returns normally. Most handles and memory pointers returned by the Dart Embedding API are allocated in the current local scope, and will be invalid after the function returns. If the extension wants to keep a pointer to a Dart object for a long time, it should use a *persistent handle* (see Dart_NewPersistentHandle() and Dart_NewWeakPersistentHandle()), which remains valid after a local scope ends.

Calls into the Dart Embedding API might return errors in their Dart_Handle return values. These errors, which might be exceptions, should be passed up to the caller of the function as the return value.

Most of the functions in a native extension—the functions of type Dart_NativeFunction—have no return value and must pass the error up to the proper handler in another way. They call Dart_PropagateError to pass errors and control flow to where the error should be handled. The sample uses a helper function, called HandleError(), to make this convenient. A call to Dart_PropagateError() never returns.

The native code: sample_extension.cc

Now we'll show the native code for the sample extension, starting with the initialization function, then the native function implementations, and ending with the name resolution function. The two native functions implementing the asynchronous extension are shown later.

```
#include <string.h>
#include "dart_api.h"
// Forward declaration of ResolveName function.
Dart_NativeFunction ResolveName(Dart_Handle name, int argc, bool* auto_setup_scope);

// The name of the initialization function is the extension name followed
// by _Init.
DART_EXPORT Dart_Handle sample_extension_Init(Dart_Handle parent_library) {
  if (Dart_IsError(parent_library)) return parent_library;

  Dart_Handle result_code =
      Dart_SetNativeResolver(parent_library, ResolveName, NULL);
  if (Dart_IsError(result_code)) return result_code;

  return Dart_Null();
}

Dart_Handle HandleError(Dart_Handle handle) {
 if (Dart_IsError(handle)) Dart_PropagateError(handle);
 return handle;
}

// Native functions get their arguments in a Dart_NativeArguments structure
// and return their results with Dart_SetReturnValue.
void SystemRand(Dart_NativeArguments arguments) {
  Dart_Handle result = HandleError(Dart_NewInteger(rand()));
  Dart_SetReturnValue(arguments, result);
}

void SystemSrand(Dart_NativeArguments arguments) {
  bool success = false;
  Dart_Handle seed_object =
      HandleError(Dart_GetNativeArgument(arguments, 0));
  if (Dart_IsInteger(seed_object)) {
    bool fits;
    HandleError(Dart_IntegerFitsIntoInt64(seed_object, &fits));
    if (fits) {
      int64_t seed;
      HandleError(Dart_IntegerToInt64(seed_object, &seed));
      srand(static_cast<unsigned>(seed));
      success = true;
    }
  }
  Dart_SetReturnValue(arguments, HandleError(Dart_NewBoolean(success)));
}

Dart_NativeFunction ResolveName(Dart_Handle name, int argc, bool* auto_setup_scope) {
  // If we fail, we return NULL, and Dart throws an exception.
  if (!Dart_IsString(name)) return NULL;
  Dart_NativeFunction result = NULL;
  const char* cname;
  HandleError(Dart_StringToCString(name, &cname));

  if (strcmp("SystemRand", cname) == 0) result = SystemRand;
  if (strcmp("SystemSrand", cname) == 0) result = SystemSrand;
  return result;
}
```

Here is the sequence of events that happens at runtime, when the function systemRand() (defined in sample_synchronous_extension.dart) is called for the first time.

1. The function ResolveName() in the shared library is called with a Dart string containing "SystemRand" and the integer 0, representing the number of arguments in the call. The string "SystemRand" is the string literal following the **native** keyword in the declaration of systemRand().
2. ResolveName() returns a pointer to the native function SystemRand() in the shared library.
3. The arguments to the systemRand() call in Dart are packaged into a Dart_NativeArguments object, and SystemRand() is called with this object as its only argument.
4. SystemRand() does its computations, puts its return value into the Dart_NativeArguments object, and returns.
5. The Dart VM extracts the return value from the Dart_NativeArguments object, returning it as the result of the Dart call to systemRand().

On later calls to systemRand(), the result of the function lookup has been cached, so ResolveName() is not called again.

The asynchronous native extension

As we saw above, a synchronous extension uses the Dart Embedding API to work with Dart heap objects, and it runs on the main Dart thread for the current isolate. An asynchronous extension, on the other hand, does not use most of the Dart Embedding API, and it runs on a separate thread so as not to block the main Dart thread.

In many ways, asynchronous extensions are simpler to program than synchronous extensions. They use the native ports functions in the Dart Embedding API to schedule C functions on independent threads. To Dart code that uses the extension, it appears simply as a Dart SendPort. The messages posted to this port are automatically translated into a C structure called a Dart_CObject, containing C data types such as int, double, and char*. This C structure is then passed to a C function, which is run in an independent thread drawn from a pool of threads managed by the VM. The C function can respond by a Dart_CObject to a reply port. The Dart_CObject is translated back into a tree of Dart objects, and appears as a reply on the Dart async call's reply port. This automatic conversion of Dart objects into a Dart_CObject C structure replaces the use of the Dart Embedding API to fetch fields from objects and to convert Dart objects into C value types.

To create an asynchronous native extension, we do three things:

1. Wrap the C function we wish to call with a wrapper that converts the Dart_CObject input argument to the desired input parameters, converts the result of the function to a Dart_CObject, and posts it back to Dart.
2. Write a native function that creates a native port and attaches it to the wrapped function. This native function is a synchronous native method, and it's in a native extension that looks just like the synchronous extension above. We have just added the wrapped function from step 1 to the extension, as well.
3. Write a Dart class that fetches the native port and caches it. In that class, provide a function that forwards its arguments to the native port as a message, and calls a callback argument when it receives a reply to that message.

Wrapping the C function

Here is an example of a C function (actually, a C++ function, due to the use of reinterpret_cast) that creates an array of random bytes, given a seed and a length. It returns the data in a newly allocated array, which will be freed by the wrapper:

```
uint8_t* random_array(int seed, int length) {
  if (length <= 0 || length > 10000000) return NULL;

  uint8_t* values = reinterpret_cast<uint8_t*>(malloc(length));
  if (NULL == values) return NULL;

  srand(seed);
  for (int i = 0; i < length; ++i) {
    values[i] = rand() % 256;
  }
  return values;
}
```

To call this from Dart, we put it in a wrapper that unpacks the Dart_CObject containing seed and length, and that packs the result values into a Dart_CObject. A Dart_CObject can hold an integer (of various sizes), a double, a string, or an array of Dart_CObjects. It is implemented in dart_native_api.h as a struct containing a union. Look in dart_native_api.h to see the fields and tags used to access the union's members. After the Dart_CObject is posted, it and all its resources can be freed, since they have been copied into Dart objects on the Dart heap.

```
void wrappedRandomArray(Dart_Port dest_port_id,
                        Dart_Port reply_port_id,
                        Dart_CObject* message) {
  if (message->type == Dart_CObject::kArray &&
      2 == message->value.as_array.length) {
    // Use .as_array and .as_int32 to access the data in the Dart_CObject.
    Dart_CObject* param0 = message->value.as_array.values[0];
    Dart_CObject* param1 = message->value.as_array.values[1];
    if (param0->type == Dart_CObject::kInt32 &&
        param1->type == Dart_CObject::kInt32) {
      int length = param0->value.as_int32;
      int seed = param1->value.as_int32;

      uint8_t* values = randomArray(seed, length);

      if (values != NULL) {
        Dart_CObject result;
        result.type = Dart_CObject::kUint8Array;
        result.value.as_byte_array.values = values;
        result.value.as_byte_array.length = length;
        Dart_PostCObject(reply_port_id, &result);
        free(values);
        // It is OK that result is destroyed when function exits.
        // Dart_PostCObject has copied its data.
        return;
      }
    }
  }
  Dart_CObject result;
  result.type = Dart_CObject::kNull;
  Dart_PostCObject(reply_port_id, &result);
}
```

Dart_PostCObject() is the only Dart Embedding API function that should be called from the wrapper or the C function. Most of the API is illegal to call here, because there is no current isolate. No errors or exceptions can be thrown, so any error must be encoded in the reply message, to be decoded and thrown by the Dart part of the extension.

Setting up the native port

Now we set up the mechanism that calls this wrapped C function from Dart code, by sending a message. We create a native port that calls this function, and return a send port connected to that port. The Dart library gets the port from this function, and forwards calls to the port.

```
void randomArrayServicePort(Dart_NativeArguments arguments) {
  Dart_SetReturnValue(arguments, Dart_Null());
  Dart_Port service_port =
      Dart_NewNativePort("RandomArrayService", wrappedRandomArray, true);
  if (service_port != kIllegalPort) {
    Dart_Handle send_port = Dart_NewSendPort(service_port);
    Dart_SetReturnValue(arguments, send_port);
  }
}
```

Calling the native port from Dart

On the Dart side, we need a class that stores this send port, sending messages to it when a Dart asynchronous function with a callback is called. The Dart class gets the port the first time the function is called, caching it in the usual way. Here is the Dart library for the asynchronous extension:

```
library sample_asynchronous_extension;

import 'dart-ext:sample_extension';

// A class caches the native port used to call an asynchronous extension.
class RandomArray {
  static SendPort _port;

  void randomArray(int seed, int length, void callback(List result)) {
    var args = new List(2);
    args[0] = seed;
    args[1] = length;
    _servicePort.call(args).then((result) {
      if (result != null) {
        callback(result);
      } else {
        throw new Exception("Random array creation failed");
      }
    });
  }

  SendPort get _servicePort {
    if (_port == null) {
      _port = _newServicePort();
    }
    return _port;
  }

  SendPort _newServicePort() native "RandomArray_ServicePort";
}
```

Conclusion and further resources

You've seen both synchronous and asynchronous native extensions. We hope that you'll use these tools to provide access to existing C and C++ libraries, thereby adding useful new capabilities to the standalone Dart VM. We recommend using asynchronous extensions rather than synchronous extensions, because asynchronous extensions don't block the main Dart thread and can be simpler to implement. The built-in Dart I/O libraries are built around asynchronous calls to achieve high, non-blocking throughput. Extensions should have the same performance goals.

Appendix: Compiling and linking extensions

Building a shared library can be tricky, and the tools to do it are platform dependent. Building Dart native extensions is especially tricky because they are dynamically loaded, and they link to Dart Embedding API functions in the Dart library embedded in the executable that dynamically loads them.

As with all shared libraries, the compilation step must generate position- independent code. The linker step must specify that unresolved functions will be resolved in the executable when the library is loaded. We will show commands that do this on the Linux, Windows, and Mac platforms. If you download the dart source repository, the sample code also includes a platform-independent build system, called gyp, and a build file sample_extension.gypi that builds the sample extension.

Building on Linux

On Linux, you can compile the code in the samples/sample_extension directory like this:

```
g++ -fPIC -m32 -I{path to SDK include directory} -DDART_SHARED_LIB -c sample_extension.cc
```

To create the shared library from the object file:

```
gcc -shared -m32 -Wl,-soname,libsample_extension.so -o
libsample_extension.so sample_extension.o
```

Remove the -m32 to build a 64-bit library that runs with the 64-bit Dart standalone VM.

Building on Mac

1. Using Xcode (tested with Xcode 3.2.6), create a new project with the same name as the native extension, of type Framework & Library/BSD C Library, type dynamic.
2. Add the source files of your extension to the source section of the project.
3. Make the following changes in Project/Edit Project Settings, choosing the Build tab and All Configurations in the dialog box:
    1. In section Linking, line Other Linker Flags, add -undefined dynamic_lookup.
    2. In section Search Paths, line Header Search Paths, add the path to dart_api.h in the SDK download or the Dart repository checkout.
    3. In section Preprocessing, line Preprocessor Macros, add DART_SHARED_LIB=1
4. Choose the correct architecture (i386 or x86_64), and build by choosing Build/Build.

The resulting lib[extension_name].dylib will be in the **build/** subdirectory of your project location, so copy it to the desired location (probably the location of the Dart library part of the extension).

Building on Windows

The Windows DLL compilation is complicated by the fact that we need to link with library file, dart.lib, that does not contain code itself, but specifies that calls to the Dart Embedding API will be resolved by linking to the Dart executable, dart.exe, when the DLL is dynamically loaded. This library file is generated when building dart and is included in the Dart SDK.

1. Create a new project of type Win32/Win32 project in Visual Studio 2008 or 2010. Give the project the same name as the native extension. On the next screen of the wizard, change the application type to DLL and select “Empty project”, then choose finish.
2. Add the C/C++ files for the native extension to the source files folder in the project. Make sure to include the [extension name]_dllmain_win.cc file.
3. Change the following settings in the project’s properties:
    1. Configuration properties / Linker / Input / Additional dependencies: Add dart-sdk\bin\dart.lib, from the downloaded Dart SDK.
    2. Configuration properties / C/C++ / General / Additional Include Directories: Add the path to the directory containing dart_api.h, which is dart-sdk/include in the downloaded Dart SDK.
    3. Configuration properties / C/C++ / Preprocessor / Preprocessor Definitions: Add DART_SHARED_LIB. This is just to export the _init function from the DLL, since it has been declared as DART_EXPORT.
4. Build the project, and copy the DLL to the correct directory, relative to the Dart library part of the extension. Make sure to build a 32-bit DLL for use with the 32-bit SDK download, and a 64-bit DLL for use with the 64-bit download.