# Adding A New System Service To Android 5: Tips and How To

*By Jacopo Mondi Posted July 22, 2015 In Linaro Blog*

## Intro:

This article explains how to add a new service and associated application APIs to Android Lollipop 5

Starting from a stub HAL object, we'll tie Java application APIs to low level operations, exploring each layer of Android internal components, in which is quite easy to get lost due to their depth, complexity and lack of general documentation.

I'm going to call this project *joffee* for no particular reason (Java Coffee??) and since I need a name to append to the github repository where this code will be hosted… well, I wasn't able to do better than this.

Code for this example will be hosted in the following github repositories:

*framework/base — https://github.com/jmondi/framework_base_joffee*

*HAL — https://github.com/jmondi/hardware_joffee*

## Pre:

A general knowledge of Androidsystem layering is required to fully understand the following article. Knowing what an HAL object is, the role of JNI and the Binder in an Android system is a mandatory prerequisite.

The official Android documentation is light on details on this side, but on the web many resources are available searching for the above keywords. The best typographic resource on this is still *"Embedded Android"* from K. Yaghmour, which details many aspects of Android system internals, and provides links to other useful resources.

The official Android documentation is useful as well, specifically for what concerns higher level concepts; some keyword you may be interested in looking for on the web are, in no particular order: parcelable types, AIDL, Android remote services, Message Handlers and Loopers;

A working knowledge of Android application development, SDK components, developers API and Android filesystem, while not mandatory, is beneficial in order to expand the proposed examples and make something useful out of it.

## HW:

Since we are going to present a running code example along with the theoretical explanation of what is going on under the hood, a hardware platform to test it is of course needed.

We are using for this example the *NVidia Jetson Board*, a quad A-15 core development board where we are going to run Android 5.1. The Android binaries and other useful informations, such as how to flash and boot the board, can be found on projectara wiki page, for which this board has been used as development platform.

Follow the provided instructions to have a running Android installation on your board: If you have other development platform available they should fit as well, since there will be no direct hw interfacing involved in this example (please note that different releases of Android can differ in some internal details, this example refers to Android 5.1, as above said).

## The HAL:

We are now going to develop and deploy a very simple HAL object with a single function that prints out "Hello Android!" due to my lack of imagination in finding something more useful to do.

Usually, HAL functions instead of simply printing out a string, interface with the underlying kernel, typically though sysfs attributes. You can find a lot on this on the web!

Also, "Appendix A" of *Embedded Android* provides an in-depth analysis of HAL structure and components; read this first if you have problem understanding the following parts!

Even if you are expected to know what a HAL object is and what it does, it is interesting to spend a few word on HAL loading mechanism, and how you should make your HAL object available to the rest of Android system, but first, lets' take a look at the three components we need here:

*Android.mk — https://github.com/jmondi/hardware_joffee/blob/master/Android.mk*

The make target for our HAL is pretty simple; it instructs the build system on where to put the resulting .so object (*system/lib/hw)* and that we want to append the target hardware name to the library name (*joffee.tegra.so* in our case). Everything else is pretty straightforward!

*joffee.h* — https://github.com/jmondi/hardware_joffee/blob/master/joffee.h

The header represents the "contract" between the JNI/Java part of Android system with the HW specific part (the HAL). Respecting this "contract" guarantees that Android can run on every hardware for which the proper set of HALs have been implemented with little or no modifications (that's the theory, at least).

The header, as comments in code explain, provides a structure describing our device, which gather together the methods that have to be exposed to Android framework.

The first field of this structure (*struct joffee_device_t* in our example) has to be a standard Android component, a *struct hw_device_t* field.

Let's see why, in the actual HAL implementation:

*joffee.cpp* — https://github.com/jmondi/hardware_joffee/blob/master/joffee.cpp

This cpp (or C) module is the actual HAL implementation, where methods for accessing your hardware have to be implemented. We know that the resulting .so will be placed somewhere in *system/lib/hw,* but how is loading of this specific object performed in Android?

The answer is at the end of the source file, where a mandatory *struct hw_module_t HAL_MODULE_INFO_SYM* instance has to be provided.

Android features and HAL loading mechanism, implemented in libhardware library, that walks all the possible paths where an HAL can be placed, building several different library name and trying to load symbols from them.

When the proper .so object has been found, its symbols are loaded, in particular the *HAL_MODULE_INFO_SYM* field, which feature a *.open* method; this will be used by framework layer to actually "open" the library and get pointers to its additional methods.

The over-commented *joffee.cpp* implementation provides detail about how this happens, and what you have to be careful about; specifically in putting an *hw_device_t* filed as first member of your HAL device structure.

This neat hack, realizes the *Android standard way* for loading HAL object, and every library which wants to be used as an HAL by the system has to respect this conventions.

## The JNI Layer:

With the introduction of a JNI layer, we now move to the *frameworks/base* directory of Android sources, where a git project (named frameworks_base) implements the real meat of Android systems.

Framework_base is quite a huge project, where all the Android services, core libraries and application support libraries reside. Its organization is difficult to follow, with several layers scattered in different packages on the filesystem, and services implementation where design which can greatly differ.

Some principles, by the way, are common to the all of them fortunately. Again, the web is full of books and articles which can explain more that this single article.

We'll start our new service from the bottom, that is the connection between the C/C++ HAL layer and the Java framework implementation.

Each service that communicates with an HAL needs a way to interface to native code, and the Java programming language provides a tool which allows native code to be executed from Java programs (and the other way around).

The path of the JNI service directory is then — *frameworks/base/services/core/jni/*

where a series of cpp files, named with naming scheme resembling the service packages they are loaded by is present.

We are going to implement the *JoffeeService* service, thus out JNI file will be called *com_android_server_joffeeService.cpp*

https://github.com/jmondi/framework_base_joffee/blob/master/services/core/jni/com_android_server_joffeeService.cpp

We are exposing here two functions, one called at service startup and one that wraps the HAL method we want to use from Java.

In the init method we see the HAL loading mechanism in action, as it has been described in the previous paragraph.

The HAL object gets loaded, then, once we do have a pointer to its *open* method, the *joffee_device_t* structure gets filled with pointers to the HAL functions, so we can call them knowing the 'contract' specified by the header file *joffee.h;*

We need of course to add this new file to the Android build system and to register the method tablet to the global JNI *OnLoad* methods; we need then to modify *services/core/jni/Android.mk* to add the new JNI source file and *services/core/jni/onload.cpp* to call the method table registration.

---

***TIP:*** The header file *joffee.h* should be placed in a directory known to the build system; HAL header files are usually placed in *hardware/libhardware/include/hardware*

We can symlink our *joffee.h* there, or modify the build system *(services/core/jni/Android.mk)* in order to add our *hardware/joffee/ directory* to the inclusion path flags.

---

Once we have added out JNI to the system, we can build the service layer, and have a library ready to be deployed in a running system:

```
$mmm services
...
target Strip: libandroid_servers (out/target/product/jetson/obj/lib/libandroid_servers.so)
Install: out/target/product/jetson/system/lib/libandroid_servers.so
```

## The Service

Services are android libraries, usually written Java, which provide a remote endpoint for applications where to access system functionalities, privileged operations, and general abstraction to the underlying system.

Services run in privileged context compared to applications, and they perform sensible operations on their behalf.

It is common to compare Android services with Linux distributions daemons, and at some extent this comparison is acceptable. One of the main differences is that Linux daemons do not always interact with programs, which can access the filesystem and peripherals by means of dedicated libraries; in Android it is almost mandatory for applications to interface with a remote service, which performs security checks, guarantees safety for concurrent accesses and dispatches events to higher layers.

This 'strict' policy delivers a higher degree of consistency in Android system internals, were the roles of each components are more defined compared to other *nix distributions, resulting in a more defined layered structure, where exceptions are of course allowed, but where most of the system core components are designed respecting the same patterns.

---

***TIP:*** A note on frameworks/base/ repository organization:

While *frameworks/base* represents the most significant part of an Android system, and a lot of files are part of this repository, in general its organization can be divided in

*frameworks/base/service/core* -> The "right" side of the Binder.

Here are implemented the service interfaces;

generates services.jar

*framework/base/core/* -> The "left" side of the Binder.

Implements the application-facing part of an Android system, which speaks with services using interfaces; the packages implemented here compose the Android API, and are usually part of the SDK.

---

The most important part of a system service is thus, its interface.

Android revolves around the well known IPC mechanism implemented by the Binder, which provides an RPC-like system and allow *transactions* to happen between objects that know each other only by their respective interfaces, with no explicit dependencies at build time, nor (even more important) at deploy time.

Binder has a long history and the web is full of articles about it and its internals, including comparisons with similar tools known by Linux developers due to their extensive presence in many distributions (d-bus and other RPC or general IPC daemons and utilities).

Interfaces are defined in Android by mean of a special language, called AIDL (Android Interface Definition Language), which closely resemble a traditional Java Interface definition.

Android provides a set of tools which generates the necessary plumbing to connect that interfaces to the Binder, and have them accessible from our code.

Let’s start with I*JoffeeService.aidl*

https://github.com/jmondi/framework_base_joffee/blob/master/core/java/android/joffee/IJoffeeService.aidl

Interfaces get defined in the application facing part of the system, because they have to be visible to managers and application, we have prepared a directory for our interface in

*frameworks/base/core/java/android/joffee/* and modified the Android.mk accordingly.

Once we have defined an AIDL we have to implement the “real” service which will realize the above defined interface. Services are the “right” side of the Binder and reside in *frameworks/base/services/core/java/com/android/server*; Again we have prepared there a directory here to host our joffee service.

Let’s start from the basic here; since Android services are all started by the main system service, they have to extend the same super class, which is, with no surprise, *SystemService.*

In order to have out implementation of the above implemented AIDL interface, we need to *Publish* it to the Binder, and of course implement it, as an *IBinder* object;

In the service constructor we proceed in registering the service

```
public JoffeeService(Context context) {

  super(context);

  mContext = context;

  publishBinderService(context.JOFFEE_SERVICE, mService);
}
```

---

***TIP:*** There are many ways to register and publish a service; explore the SystemService class and other services to find the one that best fit your needs.

---

The service will of course need to talk to the JNI we have prepared, so at the end of the file we declare the prototypes of the *native* functions we want to call.

At the time of service start, we also init the native layer:

```
@Override

public void onStart() {

    mNativePointer = init_native();

}
```

Now it is time to implement the AIDL interface in the *IBinder* object we have published in the class constructor.

Since the interface is trivial, the implementation will also be very simple:

```
private final IBinder mService = new IJoffeeService.Stub {

    public void callJoffeeMethod() {

        joffeeFunction_native();

    }

}
```

In this way, when someone from the “left” side of the Binder will call *callJoffeeFunction*, it will simply trigger the JNI layer, which will then invoke the underlying HAL.

Now that we have implemented the service, we need to tweak the Android system components to start it, the Android system server location, is at

*frameworks/base/services/java/com/android/server/SystemServer.java*

Here we start the service, as the code on github shows.

---

**TIP:** Pay attention here, the SystemServer path is f*rameworks/base/services/java/…/* and not f*rameworks/base/services/core/…/*

Under "*core*" you will find *SystemService.java* not *Server* which is another Android component.

---

Now that we have implemented the "right" side, we need to add an API, to have application interact with our service.

## The Manager

Associated with each service, there usually is a so-called Manager (services are sometimes called **ManagerService*). Managers provide an application a suitable API, that becomes part of the SDK, and mediates between apps and remote services.

Our manager will use the remote services interface, and will not do anything particularly useful. In "real" use cases, managers take care of delivering notifications, filtering intents, and check permissions. In some cases managers tie directly into jni when HW access is performed directly from Java (eg. USB device)

Manager will be placed in — *frameworks/base/core/java/android/joffee* in the *android.joffee* package, where we put the AIDL interface of our service;

Implementation is trivial, as the service exposes a single method, which we wrap in what will become part of our new system API
https://github.com/jmondi/framework_base_joffee/blob/master/core/java/android/joffee/JoffeeManager.java

The implementation also contains some pointers to how you can *hide* methods from appearing in the public SDK using decorators and JavaDoc, take a look at other managers to see how they use them.

---

**Note 1:** *The proposed implementation is trivial, and does not features any advanced use of Binder RPC and argument marshalling/un-marshalling (or flattening/unflattening, see Binder documentation on this).*

*All the proposed methods do not accept parameters nor return anything.*

*It is of course possible to define types which can be passed from one side of the Binder to the other, those types are said to be Parcelable types. Android documentation provides some material on them, and you can have a look at android.hardware.input to see how they are used in manager-services communications*

**Note 2:** *What we have seen here is the definition of a service interface which is used by a manager. The other way around is of course possible, and it is useful to implement callbacks and listeners which notify applications about some specific event.*

*Again, have a look at input or USB managers to see how you can define and register an interface which services can call into and deliver messages to Manager or applications.*

---

## Registering service and Manager

Once we have implemented both managers and service, we need a way to retrieve them from application, and start calling their methods.   The default way to retrieve a manager instance is to use the *getSystemService* method, providing the right identifier.

We need to register in the execution context our new service and our manager in order to be able to retrieve them later, and we have to do that in *frameworks/base/core/java/android/app/ContextImpl.java*

```
registerService(JOFFEE_SERVICE, new ServiceFetcher() {

  public Object createService(ContextImpl ctx) {

    IBinder b = ServiceManager.getService(JOFFEE_SERVICE);

    return new JoffeeManager(ctx, IJoffeeService.Stub.asInterface(b));

 }});
```

Just adding this allows application  to later retrieve an instance to our JoffeeManager.

## Deploy and testing

Now that all pieces are in place, we just need to update the system API and build the SDK, to have our new objects available to applications.

```
$ make update-api

$ make sdk
```

And you can now make Android studio point to the newly built SDK (copy it somewhere, so you can avoid it gets overwritten by new builds)

---

**TIP:**  *After* building the SDK, you cannot build *frameworks/base/* alone anymore, you will get

make: *** No rule to make target *out/target/product/jetson/system/framework/framework-res.apk'*

You can overcome this with

```
$ mmma frameworks/base
```

but it takes a long time, otherwise build frameworks-res alone, then rebuild frameworks

```
$ mmm frameworks/base/core/res
$ mmm frameworks/base
```

---

Let’s now deploy all our pieces onto the real target

```
$ adb remount

$ mmm hardware/joffee/
$ adb push out/target/product/jetson/system/lib/hw/joffee.tegra.so system/lib/hw/

$ mmm frameworks/base/
$ adb push out/target/product/jetson/system/framework/framework.jar  system/framework/

$ mmm frameworks/base/services
$ adb push out/target/product/jetson/system/framework/services.jar system/framework/
$ adb push out/target/product/jetson/system/lib/libandroid_servers.so system/lib/

$ adb reboot
```

And make Android studio point to your newly build SDK (File->Project Structure -> SDK Location) to test our new API

```
$ make update-api
$ make sdk
$ cp -r out/host/linux-x86/sdk/jetson/android-sdk_eng.jmondi_linux-x86 ~/Android/Sdk_Joffee/
```

Now we can test our implementation with the simplest possible application

```
import android.app.Activity;
import android.content.Context;
import android.joffee.JoffeeManager;

    public class MainActivity extends Activity {
    private JoffeeManager joffeeManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    joffeeManager = getSystemService(JOFFEE_SERVICE);

     joffeeManager.callJoffeeMethod();
    }
```

```
        }
```

If everything has gone in the right way, you should see the “*Hello Android!!*” printout on logcat!

## Conclusions

Adding new services to Android requires a vast knowledge of many system aspects, and a lot of tweaking of existing parts, where most of the time the only resource you have is the existing code in AOSP.

Android features a nice structured and layered design, which allows it to expose a well defined and well documented set of API to application developer. The counter of all of this is the depth of its internals, and the number of “mobile parts” you have to touch to to include new features of modify existing ones.