**EXTENDING ANDROID HAL**

By

Regards

Ashish Tanwer

Masters, Computer Science, Stony Brook University

1063 morse avenue, apt 3-200, Sunnyvale, California-94089

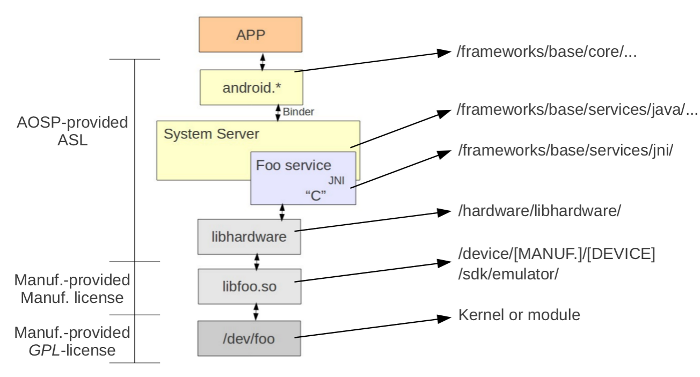
[ashishtanwer@gmail.com](mailto:ashishtanwer@gmail.com), +1-[(631) 551-3060](tel:%28631%29%20551-3060)

<http://ashishtanwer.info>, <http://in.linkedin.com/in/ashishtanwer>

<http://www.youtube.com/user/ashishtanwer> <https://github.com/ashishtanwer>

Android requires more than just proper device drivers to function on hardware. It defines a new Hardware Abstraction Layer (HAL) which defines an API for each type of hardware supported by Android's core. In order for an ODM's hardware to properly interface with Android, it must provide a hardware "module" (unrelated to kernel modules) which conforms to the API specified for that type of hardware.

Generally, each type of hardware supported by Android has a corresponding "System Service" and HAL definition. There's a Lights system service and a lights HAL definition. There's a WiFi manager system service and wifi HAL definition. The same goes for power management, location, sensors, etc. The following figure from our CC-BY-SA Embedded Android courseware illustrates this architecture:



There are two general categories of HAL modules, those explicitly loaded (through a runtime dlopen()) and those automatically loaded by the dynamic linker (libhardware\_legacy.so).

The API for the former are in

**<aosp>/hardware/libhardware/include/hardware**

and the API for the latter are in

**<aosp>/hardware/libhardware\_legacy/include/hardware\_legacy.**

The trend seems to be that Android is moving away from "legacy". The interface between those ".so" files and the actual drivers through /dev entries or otherwise is up to the ODM to specify. Android doesn't care about that. It only cares about the HAL .so modules.

One of the questions I often get is "how do I add support for my own type of hardware in Android?" To illustrate this, I've created a opersys-hal-hw type and have posted the code that implements this HAL type on github along with a very basic circular buffer driver.

The Code

<https://github.com/opersys/opersys-hal-hw>

<https://github.com/opersys/circular-driver>

If you copy the content of the opersys-hal-hw project over an existing 2.3.7\_r1 release of the AOSP and build it for the emulator, you should get yourself an image that comes up with the "opersys" service. That service has a very simple interface consisting of a read() and a write() call. If you follow the code, you will see how each of these are implemented at every layer of Android with the core definition of the new hardware type being in

**<aosp>hardware/libhardware/include/hardware/opersyshw.h**

**#ifndef ANDROID\_OPERSYSHW\_INTERFACE\_H**

**#define ANDROID\_OPERSYSHW\_INTERFACE\_H**

**#include**

**#include**

**#include**

**#include**

**\_\_BEGIN\_DECLS**

**#define OPERSYSHW\_HARDWARE\_MODULE\_ID "opersyshw"**

**struct opersyshw\_device\_t {**

**struct hw\_device\_t common;**

**int (\*read)(char\* buffer, int length);**

**int (\*write)(char\* buffer, int length);**

**int (\*test)(int value);**

**};**

**\_\_END\_DECLS**

**#endif // ANDROID\_OPERSYSHW\_INTERFACE\_H**

The code for the hardware module implementing this interface is in **<aosp>sdk/emulator/opersyshw/opersyshw\_qemu.c**

And as you can see by looking at this code, it essentially just does the "read" and "write" by opening /dev/circchar and reading/writing from/to it. In fact, if you've built the driver I pointed to earlier and have it in your emulator's /system/lib/modules/, the modified init configuration files I provide in the opersys-hal-hw project will automatically load it before system services are started and, therefore, the opersys hardware module will be able to talk to the "device" when the system services start.

The opersys hw module is itself loaded by the opersys system service's JNI code in **<aosp>/framework/base/services/jni/com\_android\_server\_OpersysService.cpp**

The system service itself is in this case implemented in Java and has been added to the same location as the other Java-coded system services as **<aosp>/framework/base/services/java/com/android/server/OpersysService.java**

The loading of the HAL module by the system service's JNI code is usually done through a call to hw\_get\_module(), which essentially results in a dlopen():

**static jint init\_native(JNIEnv \*env, jobject clazz)**

**{**

**int err;**

**hw\_module\_t\* module;**

**opersyshw\_device\_t\* dev = NULL;**

**err = hw\_get\_module(OPERSYSHW\_HARDWARE\_MODULE\_ID, (hw\_module\_t const\*\*)&module);**

**if (err == 0) {**

**if (module->methods->open(module, "", ((hw\_device\_t\*\*) &dev)) != 0)**

**return 0;**

**}**

**return (jint)dev;**

**}**

Yes, you're reading this right, I'm returning a pointer as an int to the Java layer. A student attending one of my classes actually asked it this way: "Does Android run on 64-bit machines?" Have a look at the Lights service, the above is inspired by it. And the pointer does come back:

**static int read\_native(JNIEnv \*env, jobject clazz, int ptr, jbyteArray buffer)**

**{**

**opersyshw\_device\_t\* dev = (opersyshw\_device\_t\*)ptr;**

**jbyte\* real\_byte\_array;**

**int length;**

**real\_byte\_array = env->GetByteArrayElements(buffer, NULL);**

**if (dev == NULL) {**

**return 0;**

**}**

**length = dev->read((char\*) real\_byte\_array, env->GetArrayLength(buffer));**

**env->ReleaseByteArrayElements(buffer, real\_byte\_array, 0);**

**return length;**

**}**

That's for how a system service talks to a new type of hardware through the HAL. There's also the matter of exposing the new system service's API to the upper layers. To do this, you need something like

**<aosp>/frameworks/base/core/java/android/os/IOpersysService.aidl:**

**package android.os;**

**interface IOpersysService {**

**/\*\***

**\* {@hide}**

**\*/**

**String read(int maxLength);**

**int write(String mString);**

**}**

This allows you to use something like the following HelloOpersysInternal app that builds as part of the AOSP to call on the new system service:

**package com.opersys.hellointernal;**

**import android.app.Activity;**

**import android.os.Bundle;**

**import android.util.Log;**

**import android.os.ServiceManager; // Will only work in AOSP**

**import android.os.IOpersysService; // Interface "hidden" in SDK**

**public class HelloOpersysInternalActivity extends Activity {**

**private static final String DTAG = "HelloOpersysInternal";**

**/\*\* Called when the activity is first created. \*/**

**@Override**

**public void onCreate(Bundle savedInstanceState) {**

**super.onCreate(savedInstanceState);**

**setContentView(R.layout.main);**

**IOpersysService om = IOpersysService.Stub.asInterface(ServiceManager.getService("opersys"));**

**try {**

**Log.d(DTAG, "Going to write to the \"opersys\" service");**

**om.write("Hello Opersys");**

**Log.d(DTAG, "Service returned: " + om.read(20));**

**}**

**catch (Exception e) {**

**Log.d(DTAG, "FAILED to call service");**

**e.printStackTrace();**

**}**

**}**

That, though, uses APIs which are only available within the AOSP's build. What you likely want is to generate a new SDK and call the new system service using the standard developer API calls:

**package com.opersys.hello;**

**import android.app.Activity;**

**import android.os.Bundle;**

**import android.os.OpersysManager;**

**import android.util.Log;**

**public class HelloOpersysActivity extends Activity {**

**private static final String DTAG = "HelloOpersysInternal";**

**/\*\* Called when the activity is first created. \*/**

**@Override**

**public void onCreate(Bundle savedInstanceState) {**

**super.onCreate(savedInstanceState);**

**setContentView(R.layout.main);**

**OpersysManager om = (OpersysManager) getSystemService(OPERSYS\_SERVICE);**

**Log.d(DTAG, "Going to write to the \"opersys\" service");**

**om.write("Hello Opersys");**

**Log.d(DTAG, "Service returned: " + om.read(20));**

**}**

That's taken care of by modifiying the getSystemService() implementation, the addition of OPERSYS\_SERVICE to the Context implementation and the creation of an OpersysManager class which is essentially a wrapper around the calls made to the opersys system service. Have a look at the files and modifications in <aosp>/frameworks/base/core/ for the details of this.

**Folder structure:**

The HAL source code reside under the hardware folder . OEMs maintain a top level root directory and internally there are several HAL modules for the corresponding hardware.

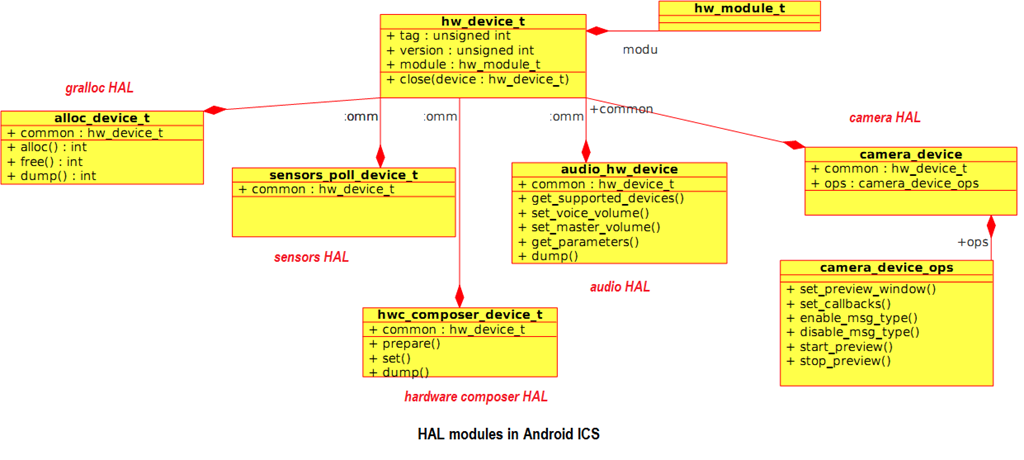
On device ,the HALs live as .so’s either in ‘/system/lib/hw’ or ‘/vendor/lib/hw’. The HAL modules follow a certain nomenclature as defined here . Android relies on this naming convention to load the right HAL module for a given hardware and board configuration.

**Runtime:**

The HALs get loaded by its clients using the hw\_get\_module. This method relies on the values set to ro.hardware , ro.product.board and some other ro properties to load the correct .so. The complete list is here.

For eg. The SurfaceFlinger is the system wide compositor and the hardware composer is a HAL that abstracts device specific composition logic. The SurfaceFlinger loads the hardware composer using the hw\_get\_module method to get a handle of the HAL.

Each HAL implements a set of hooks which get called by their invoking service. The contract between the system service and the HAL is well defined. Device makers should ensure these interfaces are implemented as expected by the Android services. The figure below tries to show some of the HALs on the Android ICS version and key methods that HAL writes should implement.



With the newer versions of Android ,the scope of HALs have expanded and it is likely they will be used not just to abstract hardware but to abstract any device specific behavior.

**Reference:**

<http://processors.wiki.ti.com/index.php/TI-Android-DevKit-Camera_Porting_Guide>

<https://sites.google.com/site/io/anatomy--physiology-of-an-android>

<http://www.opersys.com/blog/extending-android-hal>

<http://sujaiantony.wordpress.com/2012/06/25/an-android-101-hardware-and-hal/>

<https://github.com/opersys/opersys-hal-hw>

<https://github.com/opersys/circular-driver>