HIDL Java

Android O re-architects the Android OS to define clear interfaces between the device-independent Android platform and device- and vendor-specific code. Android already defines many such interfaces in the form of HAL interfaces, defined as C headers in **`hardware/libhardware`**. HIDL replaces these HAL interfaces with stable, versioned interfaces, which can be in Java (described below) or as client- and server-side HIDL interfaces in C++ (https://source.android.com/devices/architecture/hidl-cpp/index.html).

HIDL interfaces are intended to be used primarily from native code, and as a result HIDL is focused on the auto-generation of efficient code in C++. However, HIDL interfaces must also be able to be used directly from Java as some Android subsystems (such as Telephony) will most likely have Java HIDL interfaces.

The pages in this section describe the Java frontend for HIDL interfaces, detail how to create, register, and use services, and explain how HALs and HAL clients written in Java interact with the HIDL RPC system.

Being a client

To access an interface IFoo in package **`android.hardware.foo`** version 1.0 that is registered as service name **`foo-bar`**:

1. Add libraries:
   - Add the following to Android.mk:

     ```
     LOCAL_JAVA_LIBRARIES += android.hardware.foo-V1.0-java
     ```

     OR

   - Add the following to Android.bp:

     ```
     shared_libs: [
         /* … */
         "android.hardware.foo-V1.0-java",
     ],
     ```

     The static version of the library is also available as **`android.hardware.foo-V1.0-java-static`**.
2. Add the following to your Java file:

   ```
   import android.hardware.foo.V1_0.IFoo;
   ...
   IFoo server = IFoo.getService(); // throws exception if not available
   IFoo anotherServer = IFoo.getService("second_impl");
   server.doSomething(…);
   ```

Providing a service

Framework code in Java may need to serve interfaces to receive asynchronous callbacks from HALs.

Warning: Do not implement a driver (HAL) in Java. We strongly recommend you implement drivers in C++.

For interface **`IFooCallback`** in version 1.0 of package **`android.hardware.foo`**, you can implement your interface in Java using the following steps:

1. Define your interface in HIDL.
2. Open **`/tmp/android/hardware/foo/IFooCallback.java`** as a reference.
3. Create a new module for your Java implementation.
4. Examine the abstract class **`android.hardware.foo.V1_0.IFooCallback.Stub`**, then write a new class to extend it and implement the abstract methods.

Viewing auto-generated files

To view the automatically-generated files, run:

```
hidl-gen -o /tmp -Ljava \
  -randroid.hardware:hardware/interfaces \
  -randroid.hidl:system/libhidl/transport android.hardware.foo@1.0
```

These commands generate the directory **`/tmp/android/hardware/foo/1.0`**. For the file **`hardware/interfaces/foo/1.0/IFooCallback.hal`**, this generates the file **`/tmp/android/hardware/foo/1.0/IFooCallback.java`**, which encapsulates the Java interface, the proxy code, and the stubs (both proxy and stubs conform to the interface).

`-Lmakefile` generates the rules that run this command at build time and allow you to include **`android.hardware.foo-V1.0-java(-static)?`** and link against the appropriate files. A script that automatically does this for a project full of interfaces can be found at **`hardware/interfaces/update-makefiles.sh`**. The paths in this example are relative; hardware/interfaces can be a temporary directory under your code tree to enable you to develop a HAL prior to publishing it.

Running a service

The HAL provides an interface **`IFoo`**, which must make asynchronous callbacks to the framework over interface **`IFooCallback`**. The **`IFooCallback`** interface is not registered by name as a discoverable service; instead, **`IFoo`** must contain a method such as **`setFooCallback(IFooCallback x)`**.

To set up **`IFooCallback`** from version 1.0 of package **`android.hardware.foo`**, add **`android.hardware.foo-V1.0-java`** to **`Android.mk`**. The code to run the service is:

```
import android.hardware.foo.V1_0.IFoo;
import android.hardware.foo.V1_0.IFooCallback.Stub;
....
class FooCallback extends IFoo.Stub {
    // implement methods
}
....
// Get the service you will be receiving callbacks from.
// This also starts the threadpool for your callback service.
IFoo server = IFoo.getService(); // throws exception if not available
....
// This must be a persistent instance variable, not local,
//   to avoid premature garbage collection.
FooCallback mFooCallback = new FooCallback();
....
// Do this once to create the callback service and tell the "foo-bar" service
server.setFooCallback(mFooCallback);
```

Interface extensions

Assuming a given service implements an interface **`IFoo`** across all devices, it's possible that on a particular device the service may provide additional capabilities implemented in an interface extension **`IBetterFoo`**, i.e.:

```
interface IFoo {
   ...
};

interface IBetterFoo extends IFoo {
   ...
};
```

Calling code aware of the extended interface can use the **`castFrom()`** Java method to safely cast the base interface to the extended interface:

```
IFoo baseService = Foo.getService();
IBetterFoo extendedService = IBetterFoo.castFrom(baseService);
if (extendedService != null) {
  // The service implements the extended interface.
} else {
```

```
  // The service only implements the base interface.
}
```