HIDL

HAL interface definition language or HIDL (pronounced "hide-l") is an interface description language (IDL) to specify the interface between a HAL and its users. It allows specifying types and method calls, collected into interfaces and packages. More broadly, HIDL is a system for communicating between codebases that may be compiled independently.

HIDL is intended to be used for inter-process communication (IPC). Communication between processes is referred to as *Binderized* (https://source.android.com/devices/architecture/hidl/binder-ipc). For libraries that must be linked to a process, a passthough mode (#passthrough) is also available (not supported in Java).

HIDL specifies data structures and method signatures, organized in interfaces (similar to a class) that are collected into packages. The syntax of HIDL will look familiar to C++ and Java programmers, though with a different set of keywords. HIDL also uses Java-style annotations.

HIDL design

The goal of HIDL is that the framework can be replaced without having to rebuild HALs. HALs will be built by vendors or SOC makers and put in a `/vendor` partition on the device, enabling the framework, in its own partition, to be replaced with an OTA without recompiling the HALs.

HIDL design balances the following concerns:

- **Interoperability**. Create reliably interoperable interfaces between processes which may be compiled with various architectures, toolchains, and build configurations. HIDL interfaces are versioned and cannot be changed after they are published.
- **Efficiency**. HIDL tries to minimize the number of copy operations. HIDL-defined data is delivered to C++ code in C++ standard layout data structures that can be used without unpacking. HIDL also provides shared memory interfaces and, as RPCs are inherently somewhat slow, HIDL supports two ways to transfer data without using an RPC call: shared memory and a Fast Message Queue (FMQ).
- **Intuitive**. HIDL avoids thorny issues of memory ownership by using only `in` parameters for RPC (see Android Interface Definition Language (AIDL) (https://developer.android.com/guide/components/aidl.html)); values that cannot be efficiently returned from methods are returned via callback functions. Neither passing data into HIDL for transfer nor receiving data from HIDL changes the ownership of the data—ownership always remains with the calling function. Data needs to persist only for the duration of the called function and may be destroyed immediately after the called function returns.

Using passthrough mode

To update devices running earlier versions of Android to Android O, you can wrap both conventional (and legacy) HALs in a new HIDL interface that serves the HAL in binderized and same-process (passthrough) modes. This wrapping is transparent to both the HAL and the Android framework.

Passthrough mode is available only for C++ clients and implementations. Devices running earlier versions of Android do not have HALs written in Java, so Java HALs are inherently binderized.

Passthrough header files

When a `.hal` file is compiled, `hidl-gen` produces an extra passthrough header file `BsFoo.h` in addition to the headers used for binder communication; this header defines functions to be `dlopen`ed. As passthrough HALs run in the same process in which they are called, in most cases passthrough methods are invoked by direct function call (same thread). `oneway` methods run in their own thread as they are not intended to wait for the HAL to process them (this means any HAL that uses `oneway` methods in passthrough mode must be thread-safe).

Given an `IFoo.hal`, `BsFoo.h` wraps the HIDL-generated methods to provide additional features (such as making `oneway` transactions run in another thread). This file is similar to `BpFoo.h`, however instead of passing on calls IPC using binder, the desired functions are directly invoked. Future implementations of HALs **may provide** multiple implementations, such as FooFast HAL and a FooAccurate HAL. In such cases, a file for each additional implementation would be created (e.g., `PTFooFast.cpp` and `PTFooAccurate.cpp`).

Binderizing passthrough HALs

You can binderize HAL implementations that support passthrough mode. Given a HAL interface `a.b.c.d@M.N::IFoo`, two packages are created:

- `a.b.c.d@M.N::IFoo-impl`. Contains the implementation of the HAL and exposes function `IFoo* HIDL_FETCH_IFoo(const char* name)`. On legacy devices, this package is `dlopen`ed and the implementation is instantiated using `HIDL_FETCH_IFoo`. You can generate the base code using `hidl-gen` and `-Lc++-impl` and `-Landroidbp-impl`.
- `a.b.c.d@M.N::IFoo-service`. Opens the passthrough HAL and registers itself as a binderized service, enabling the same HAL implementation to be used as both passthrough and binderized.

Given the type `IFoo`, you can call `sp<IFoo> IFoo::getService(string name, bool getStub)` to get access to an instance of `IFoo`. If `getStub` is true, `getService` attempts to open the HAL only in passthrough mode. If `getStub` is false, `getService` attempts to find a binderized service; if that fails, it then tries to find the passthrough service. The `getStub` parameter should never be used except in `defaultPassthroughServiceImplementation`. (Devices launching with Android O are fully binderized devices, so opening a service in passthrough mode is disallowed.)

HIDL grammar

By design, the HIDL language is similar to C (but does not use the C preprocessor). All punctuation not described below (aside from the obvious use of `=` and `|`) is part of the grammar.

Note: For details on HIDL code style, see the Code Style Guide (https://source.android.com/devices/architecture/hidl/code-style.html).

- `/** */` indicates a documentation comment.
- `/* */` indicates a multiline comment.
- `//` indicates a comment to end of line. Aside from `//`, newlines are the same as any other whitespace.
- In the example grammar below, text from `//` to the end of the line is not part of the grammar but is instead a comment on the grammar.
- `[empty]` means that the term may be empty.
- `?` following a literal or term means it is optional.
- `...` indicates sequence containing zero or more items with separating punctuation as indicated. There are no variadic arguments in HIDL.
- Commas separate sequence elements.
- Semicolons terminate each element, including the last element.
- UPPERCASE is a nonterminal.
- *`italics`* is a token family such as *`integer`* or *`identifier`* (standard C parsing rules).
- *`constexpr`* is a C style constant expression (such as `1 + 1` and `1L << 3`).
- *`import_name`* is a package or interface name, qualified as described in HIDL Versioning (https://source.android.com/devices/architecture/hidl/versioning.html).
- Lowercase `words` are literal tokens.

Example:

```
ROOT =
    PACKAGE IMPORTS PREAMBLE { ITEM ITEM ... }  // not for types.hal
    PREAMBLE = interface identifier EXTENDS
  | PACKAGE IMPORTS ITEM ITEM...  // only for types.hal; no method definitions

ITEM =
    ANNOTATIONS? oneway? identifier(FIELD, FIELD ...) GENERATES?;
  |  struct identifier { SFIELD; SFIELD; ...};  // Note - no forward declarations
  |  union identifier { UFIELD; UFIELD; ...};
  |  enum identifier: TYPE { ENUM_ENTRY, ENUM_ENTRY ... }; // TYPE = enum or scalar
  |  typedef TYPE identifier;

VERSION = integer.integer;
```

```
PACKAGE = package android.hardware.identifier[.identifier[...]]@VERSION;

PREAMBLE = interface identifier EXTENDS

EXTENDS = <empty> | extends import_name  // must be interface, not package

GENERATES = generates (FIELD, FIELD ...)

// allows the Binder interface to be used as a type
// (similar to typedef'ing the final identifier)
IMPORTS =
   [empty]
  |  IMPORTS import import_name;

TYPE =
  uint8_t | int8_t | uint16_t | int16_t | uint32_t | int32_t | uint64_t | int64_t |
 float | double | bool | string
|  identifier  // must have been previously typedef'd
               // or defined with struct, union, enum, or import
|  memory
|  pointer
|  vec<TYPE>
|  bitfield<TYPE>  // TYPE is user-defined enum
|  fmq_sync<TYPE>
|  fmq_unsync<TYPE>
|  TYPE[SIZE]

FIELD =
   TYPE identifier

UFIELD =
   TYPE identifier
  |  struct identifier { FIELD; FIELD; ...} identifier;
  |  union identifier { FIELD; FIELD; ...} identifier;

SFIELD =
   TYPE identifier
  |  struct identifier { FIELD; FIELD; ...};
  |  union identifier { FIELD; FIELD; ...};
  |  struct identifier { FIELD; FIELD; ...} identifier;
  |  union identifier { FIELD; FIELD; ...} identifier;

SIZE =  // Must be greater than zero
     constexpr

ANNOTATIONS =
     [empty]
  |  ANNOTATIONS ANNOTATION

ANNOTATION =
  |  @identifier
  |  @identifier(VALUE)
  |  @identifier(ANNO_ENTRY, ANNO_ENTRY  ...)

ANNO_ENTRY =
     identifier=VALUE

VALUE =
     "any text including \" and other escapes"
  |  constexpr
  |  {VALUE, VALUE ...}  // only in annotations

ENUM_ENTRY =
     identifier
  |  identifier = constexpr
```

Terminology

This section uses the following HIDL-related terms:

binderized	Indicates HIDL is being used for remote procedure calls between processes, implemented over a Binder-like mechanism. See also *passthrough*.
callback, asynchronous	Interface served by a HAL user, passed to the HAL (via a HIDL method), and called by the HAL to return data at any time.
callback, synchronous	Returns data from a server's HIDL method implementation to the client. Unused for methods that return void or a single primitive value.
client	Process that calls methods of a particular interface. A HAL or framework process may be a client of one interface and a server of another. See also *passthrough*.
extends	Indicates an interface that adds methods and/or types to another interface. An interface can extend only one other interface. Can be used for a minor version increment in the same package name or for a new package (e.g. a vendor extension) to build on an older package.
generates	Indicates an interface method that returns values to the client. To return one non-primitive value, or more than one value, a synchronous callback function is generated.
interface	Collection of methods and types. Translated into a class in C++ or Java. All methods in an interface are called in the same direction: a client process invokes methods implemented by a server process.
oneway	When applied to a HIDL method, indicates the method returns no values and does not block.
package	Collection of interfaces and data types sharing a version.
passthrough	Mode of HIDL in which the server is a shared library, `dlopen`ed by the client. In passthrough mode, client and server are the same process but separate codebases. Used only to bring legacy codebases into the HIDL model. See also *Binderized*.
server	Process that implements methods of an interface. See also *passthrough*.
transport	HIDL infrastructure that moves data between the server and client.
version	Version of a package. Consists of two integers, major and minor. Minor version increments may add (but not change) types and methods.

Last updated August 21, 2017.