Code Style Guide

The HIDL code style resembles C++ code in the Android framework, with 4-space indents and mixed-case filenames. Package declarations, imports, and docstrings are similar to those in Java, with slight modifications.

The following examples for `IFoo.hal` and `types.hal` illustrate HIDL code styles and provide quick links to details on each style (`IFooClientCallback.hal`, `IBar.hal`, and `IBaz.hal` have been omitted).

hardware/interfaces/foo/1.0/IFoo.hal

```
/*
 * (License Notice)
 */

package android.hardware.foo@1.0;

import android.hardware.bar@1.0::IBar;

import IBaz;
import IFooClientCallback;

/**
 * IFoo is an interface that…
 */
interface IFoo {

    /**
     * This is a multiline docstring.
     *
     * @return result 0 if successful, nonzero otherwise.
     */
     foo() generates (FooStatus result);

    /**
     * Restart controller by power cycle.
     *
     * @param bar callback interface that…
     * @return result 0 if successful, nonzero otherwise.
     */
    powerCycle(IBar bar) generates (FooStatus result);

    /** Single line docstring. */
    baz();


    /**
     * The bar function.
     *
     * @param clientCallback callback after function is called
     * @param baz related baz object
     * @param data input data blob
     */
```

```
    bar(IFooClientCallback clientCallback,
        IBaz baz,
        FooData data);

};
```

hardware/interfaces/foo/1.0/types.hal

```
/*
 * (License Notice)
 */

package android.hardware.foo@1.0;

/** Replied status. */
enum Status : int32_t {
    OK,
    ERR_ARG, // invalid arguments
    ERR_UNKNOWN = -1, // note, no transport related errors
};

struct ArgData {
    int32_t[20] someArray;
    vec<uint8_t> data;
};
```

Naming conventions

Function names, variable names, and filenames should be descriptive; avoid over-abbreviation. Treat acronyms as words (e.g., use `INfc` instead of `INFC`).

Directory structure and file naming

The directory structure should appear as follows:

- *ROOT-DIRECTORY*
  - *MODULE*
    - *SUBMODULE* (optional, could be more than one level)
      - *VERSION*
        - Android.mk
        - I*INTERFACE_1*.hal
        - I*INTERFACE_2*.hal
        - …
        - I*INTERFACE_N*.hal
        - types.hal (optional)

Where:

- *ROOT-DIRECTORY* is:
  - hardware/interfaces for core HIDL packages.

    - `vendor/`*`VENDOR`*`/interfaces` for vendor packages, where *`VENDOR`* refers to an SoC vendor or an OEM/ODM.
- *`MODULE`* should be one lowercase word that describes the subsystem (e.g. `nfc`). If more than one word is needed, use nested *`SUBMODULE`*. There can be more than one level of nesting.
- *`VERSION`* should be the exact same version (major.minor) as described in Versions.
- `I`*`INTERFACE_X`* should be the interface name with `UpperCamelCase`/`PascalCase` (e.g. `INfc`) as described in Interface names.

Example:

- `hardware/interfaces`
    - `nfc`
        - `1.0`
            - `Android.mk`
            - `INfc.hal`
            - `INfcClientCallback.hal`
            - `types.hal`

Note: All files must have non-executable permissions (in Git).

Package names

Package names must use the following fully-qualified name (FQN) format (referred to as *`PACKAGE-NAME`*):

`PACKAGE``.`*`MODULE`*`[.`*`SUBMODULE`*`[.`*`SUBMODULE`*`[…]]]@`*`VERSION`*

Where:

- *`PACKAGE`* is the package that maps to the *`ROOT-DIRECTORY`*. In particular, *`PACKAGE`* is:
    - `android.hardware` for core HIDL packages (mapping to `hardware/interfaces`).
    - `vendor.`*`VENDOR`*`.hardware` for vendor packages, where *`VENDOR`* refers to an SoC vendor or an OEM/ODM (mapping to `vendor/`*`VENDOR`*`/interfaces`).
- *`MODULE`*`[.`*`SUBMODULE`*`[.`*`SUBMODULE`*`[…]]]@`*`VERSION`* are the exact same folder names in the structure described in Directory structure.
- Package names should be lowercase. If they are more than one word long, the words should either be used as submodules or written in `snake_case`.
- No spaces are allowed.

The FQN is always used in package declarations.

Versions

Versions should have the following format:

MAJOR.*MINOR*

Both the *MAJOR* and the *MINOR* version should be a single integer. HIDL uses semantic versioning rules.

Imports

An import has one of the following three formats:

- Whole-package imports: `import` *PACKAGE-NAME*`;`
- Partial imports: `import` *PACKAGE-NAME*`::`*UDT*`;` (or, if the imported type is in the same package,`import` *UDT*`;`
- Types-only imports: `import` *PACKAGE-NAME*`::types;`

The *PACKAGE-NAME* follows the format in Package names. The current package's `types.hal` (if it exists) is automatically imported (do not import it explicitly).

Fully qualified names (FQNs)

Use fully qualified names for a user-defined type import only when necessary. Omit *PACKAGE-NAME* if the import type is in the same package. An FQN must not contain spaces. Example of a fully qualified name:

```
android.hardware.nfc@1.0::INfcClientCallback
```

In another file under `android.hardware.nfc@1.0`, refer to the above interface as `INfcClientCallback`. Otherwise, use only the fully qualified name.

Grouping and ordering imports

Use an empty line after package declaration (before the imports). Each import should occupy a single line and should not be indented. Group imports in the following order:

1. Other `android.hardware` packages (use fully qualified names).
2. Other `vendor.`*VENDOR* packages (use fully qualified names).
    - Each vendor should be a group.
    - Order vendors alphabetically.
3. Imports from other interfaces in the same package (use simple names).

Use an empty line between groups. Inside each group, sort imports alphabetically. Example:

```
import android.hardware.nfc@1.0::INfc;
import android.hardware.nfc@1.0::INfcClientCallback;

// Importing the whole module.
import vendor.barvendor.bar@3.1;
```

```
import vendor.foovendor.foo@2.2::IFooBar;
import vendor.foovendor.foo@2.2::IFooFoo;

import IBar;
import IFoo;
```

Interface names

Interface names must start with an `I`, followed by an `UpperCamelCase`/`PascalCase` name. An interface with name `IFoo` must be defined in the file `IFoo.hal`. This file can contain definitions only for the `IFoo` interface (the interface *`INAME`* should be in *`INAME`*`.hal`).

Functions

For function names, arguments, and return variable names, use `lowerCamelCase`. Example:

```
open(INfcClientCallback clientCallback) generates (int32_t retVal);
oneway pingAlive(IFooCallback cb);
```

Struct/union field names

For struct/union field names, use `lowerCamelCase`. Example:

```
struct FooReply {
    vec<uint8_t> replyData;
}
```

Type names

Type names refer to struct/union definitions, enum type definitions, and `typedef`s. For these name, use `UpperCamelCase`/`PascalCase`. Examples:

```
enum NfcStatus : int32_t {
    /*...*/
};
struct NfcData {
    /*...*/
};
```

Enum values

Enum values should be `UPPER_CASE_WITH_UNDERSCORES`. When passing enum values as function arguments and returning them as function returns, use the actual enum type (not the underlying integer type). Example:

```
enum NfcStatus : int32_t {
    HAL_NFC_STATUS_OK               = 0,
    HAL_NFC_STATUS_FAILED           = 1,
    HAL_NFC_STATUS_ERR_TRANSPORT    = 2,
```

```
    HAL_NFC_STATUS_ERR_CMD_TIMEOUT  = 3,
    HAL_NFC_STATUS_REFUSED          = 4
};
```

Note: The underlying type of an enum type is explicitly declared after the colon. As it is not compiler dependent, using the actual enum type is clearer.

For fully qualified names for enum values, a **colon** is used between the enum type name and the enum value name:

PACKAGE-NAME::*UDT*[.*UDT*[.*UDT*[…]]]:*ENUM_VALUE_NAME*

There must not be spaces inside a fully qualified name. Use a fully qualified name only when necessary and omit unnecessary parts. Example:

```
android.hardware.foo@1.0::IFoo.IFooInternal.FooEnum:ENUM_OK
```

Comments

For a single line comment, both `//` and `/** */` are fine.

```
// This is a single line comment
/* This is also single line comment */
/** This is documentation comment */
```

- Use `//` mainly for:
  - trailing comments
  - Comments that will not be used for generated documentation
  - TODOs
- Use `/** */` for generated documentation. These can be applied only to type, method, field, and enum value declarations. Example:

```
/** Replied status */
enum TeleportStatus {
    /** Object entirely teleported. */
    OK              = 0,
    /** Methods return this if teleportation is not completed. */
    ERROR_TELEPORT  = 1,
    /**
     * Teleportation could not be completed due to an object
     * obstructing the path.
     */
    ERROR_OBJECT    = 2,
    ...
}
```

• • Multi-line comments should start a new line with `/**`, use `*` at the beginning of each line, and place `*/` on the last line all on its own (asterisks should align). Example:

```
/**
 * My multi-line
```

```
 * comment
 */
```

- Licensing notice and changelogs should start a new line with `/*` (a single asterisk), use `*` at the beginning of each line, and place `*/` on the last line all on its own (asterisks should align). Example:

```
/*
 * Copyright (C) 2017 The Android Open Source Project
 * ...
 */

/*
 * Changelog:
 * ...
 */
```

-

File comments

Start each file with the appropriate licensing notice. For core HALs, this should be the AOSP Apache license in [development/docs/copyright-templates/c.txt](). Remember to update the year and use `/* */` style multi-line comments as explained above.

You can optionally place an empty line after the license notice, followed by a changelog/versioning information. Use `/* */` style multi-line comments as explained above, place the empty line after the changelog, then follow with the package declaration.

TODO comments

TODOs should include the string `TODO` in all caps followed by a colon. Example:

```
// TODO: remove this code before foo is checked in.
```

TODO comments are allowed only during development; they must not exist in published interfaces.

Interface/Function comments (docstrings)

Use `/** */` for multi-line and single line docstrings. Do not use `//` for docstrings.

Docstrings for interfaces should describe general mechanisms of the interface, design rationale, purpose, etc. Docstrings for functions should be specific to the function (package-level documentation goes in a README file in the package directory).

```
/**
 * IFooController is the controller for foos.
 */
interface IFooController {
    /**
```

```
     * Opens the controller.
     *
     * @return status HAL_FOO_OK if successful.
     */
    open() generates (FooStatus status);

    /** Close the controller. */
    close();
};
```

You must add `@param`s and `@return`s for each parameter/return value:

- `@param` must be added for each parameter. It should be followed by the name of the parameter then the docstring.
- `@return` must be added for each return value. It should be followed by the name of the return value then the docstring.

Example:

```
/**
 * Explain what foo does.
 *
 * @param arg1 explain what arg1 is
 * @param arg2 explain what arg2 is
 * @return ret1 explain what ret1 is
 * @return ret2 explain what ret2 is
 */
foo(T arg1, T arg2) generates (S ret1, S ret2);
```

Formatting

General formatting rules include:

- **Line length**. Each line of text should be at most **80** columns long.
- **Whitespaces**. No trailing whitespace on lines; empty lines must not contain whitespaces.
- **Spaces vs. tabs**. Use only spaces.
- **Indent size**. Use **4** spaces for blocks and **8** spaces for line wraps
- **Bracing**. Except for annotation values, an **open** brace goes on the same line as preceding code but a **close** brace and the following semicolon occupies the entire line. Example:

  ```
  interface INfc {
      close();
  };
  ```

-

Package declaration

Package declaration should be at the top of the file after the license notice, should occupy the entire line, and should not be indented. Packages are declared using the following format (for name formatting, see Package names):

```
package PACKAGE-NAME;
```

Example:

```
package android.hardware.nfc@1.0;
```

Function declarations

Function name, parameters, `generates`, and return values should be on the same line if they fit. Example:

```
interface IFoo {
    /** ... */
    easyMethod(int32_t data) generates (int32_t result);
};
```

If they don't fit on the same line, attempt to put parameters and return values in the same indent level and distinguish `generate` to help the reader quickly see the parameters and return values. Example:

```
interface IFoo {
    suchALongMethodThatCannotFitInOneLine(int32_t theFirstVeryLongParameter,
                                          int32_t anotherVeryLongParameter);
    anEvenLongerMethodThatCannotFitInOneLine(int32_t theFirstLongParameter,
                                             int32_t
anotherVeryLongParameter)
                                    generates (int32_t theFirstReturnValue,
                                             int32_t anotherReturnValue);
    superSuperSuperSuperSuperSuperSuperLongMethodThatYouWillHateToType(
            int32_t theFirstVeryLongParameter, // 8 spaces
            int32_t anotherVeryLongParameter
        ) generates (
            int32_t theFirstReturnValue,
            int32_t anotherReturnValue
        );
    // method name is even shorter than 'generates'
    foobar(AReallyReallyLongType aReallyReallyLongParameter,
           AReallyReallyLongType anotherReallyReallyLongParameter)
        generates (ASuperLongType aSuperLongReturnValue, // 4 spaces
                   ASuperLongType anotherSuperLongReturnValue);
}
```

Additional details:

- An open parenthesis is always on the same line as the function name.
- No spaces between the function name and the open parenthesis.

- No spaces between the parentheses and parameters *except* when there are line feeds between them.
- If `generates` is on the same line as the previous closing parenthesis, use a preceding space. If `generates` is on the same line as the next open parenthesis, follow with a space.
- Align all parameters and return values (if possible).
- Default indentation is 4 spaces.
- Wrapped parameters are aligned to the first parameters on the previous line, otherwise they have an 8-space indent.

Annotations

Use the following format for annotations:

```
@annotate(keyword = value, keyword = {value, value, value})
```

Sort annotations in alphabetical order, and use spaces around equal signs. Example:

```
@callflow(key = value)
@entry
@exit
```

Ensure an annotation occupies the entire line. Examples:

```
// Good
@entry
@exit

// Bad
@entry @exit
```

If annotations cannot fit on the same line, indent with 8 spaces. Example:

```
@annotate(
        keyword = value,
        keyword = {
                value,
                value
        },
        keyword = value)
```

If the entire value array cannot fit in the same line, put line breaks after open braces `{` and after each comma inside the array. Place closing parenthesis immediately after the last value. Do not put the braces if there is only one value.

If the entire value array can fit in the same line, do not use spaces after open braces and before closing braces and use one space after each comma. Examples:

```
// Good
@callflow(key = {"val", "val"})
```

```
// Bad
@callflow(key = { "val","val" })
```

There must NOT be empty lines between annotations and the function declaration. Examples:

```
// Good
@entry
foo();

// Bad
@entry

foo();
```

Enum declarations

Use the following rules for enum declarations:

- If enum declarations are shared with another package, put the declarations in `types.hal` rather than embedding inside an interface.
- Use a space before and after the colon, and space after the underlying type before the open brace.
- The last enum value may or may not have an extra comma.

Struct declarations

Use the following rules for struct declarations:

- If struct declarations are shared with another package, put the declarations in `types.hal` rather than embedding inside an interface.
- Use a space after the struct type name before the open brace.
- Align field names (optional). Example:

  ```
  struct MyStruct {
      vec<uint8_t>   data;
      int32_t        someInt;
  }
  ```

-

Array declarations

Do not put spaces between the following:

- Element type and open square bracket.
- Open square bracket and array size.
- Array size and close square bracket.
- Close square bracket and the next open square bracket, if more than one dimension exists.

Examples:

```
// Good
int32_t[5] array;

// Good
int32_t[5][6] multiDimArray;

// Bad
int32_t [ 5 ] [ 6 ] array;
```

Vectors

Do not put spaces between the following:

- `vec` and open angle bracket.
- Open angle bracket and element type (*Exception: element type is also a* `vec`).
- Element type and close angle bracket (*Exception: element type is also a* `vec`).

Examples:

```
// Good
vec<int32_t> array;

// Good
vec<vec<int32_t>> array;

// Good
vec< vec<int32_t> > array;

// Bad
vec < int32_t > array;

// Bad
vec < vec < int32_t > > array;
```