[Android中HAL如何向上层提供接口总结](http://blog.csdn.net/flydream0/article/details/7086273)

标签： [android](http://www.csdn.net/tag/android)[module](http://www.csdn.net/tag/module)[struct](http://www.csdn.net/tag/struct)[methods](http://www.csdn.net/tag/methods)[数据结构](http://www.csdn.net/tag/%e6%95%b0%e6%8d%ae%e7%bb%93%e6%9e%84)[structure](http://www.csdn.net/tag/structure)

2011-12-20 10:48 9570人阅读 [评论](http://blog.csdn.net/flydream0/article/details/7086273#comments)(10) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/flydream0/article/details/7086273#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

Android（24） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/flydream0/article/details/7086273)

参考文献:

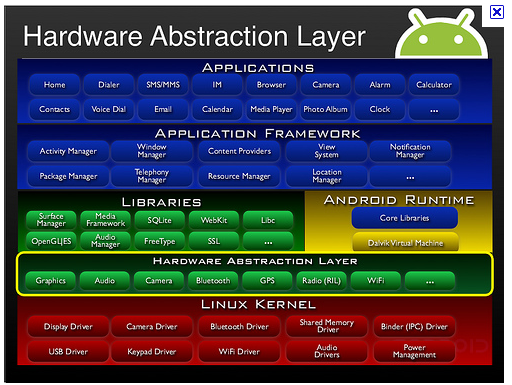
<http://blog.csdn.net/luoshengyang/article/details/6573809>

<http://blog.csdn.net/hongtao_liu/article/details/6060734>

建议阅读本文时先浏览以上两篇文章，本文是对上两篇文章在HAL对上层接口话题的一个总结.

**1 什么是HAL**

HAL的全称是Hardware Abstraction Layer,即硬件抽象层.其架构图如下:



Android的HAL是为了保护一些硬件提供商的知识产权而提出的，是为了避开linux的GPL束缚。思路是把控制硬件的动作都放到了Android HAL中，而linux driver仅仅完成一些简单的数据交互作用，甚至把硬件寄存器空间直接映射到user space。而Android是基于Aparch的license，因此硬件厂商可以只提供二进制代码，所以说Android只是一个开放的平台，并不是一个开源的平台。也许也正是因为Android不遵从GPL，所以Greg Kroah-Hartman才在2.6.33内核将Andorid驱动从linux中删除。GPL和硬件厂商目前还是有着无法弥合的裂痕。Android想要把这个问题处理好也是不容易的。

    总结下来，Android HAL存在的原因主要有：

    1. 并不是所有的硬件设备都有标准的linux kernel的接口

    2. KERNEL DRIVER涉及到GPL的版权。某些设备制造商并不原因公开硬件驱动，所以才去用HAL方式绕过GPL。

    3. 针对某些硬件，Android有一些特殊的需求.

**2 与接口相关的几个结构体**

首先来看三个与HAL对上层接口有关的几个结构体：

**[html]** [view plain](http://blog.csdn.net/flydream0/article/details/7086273) [copy](http://blog.csdn.net/flydream0/article/details/7086273)

1. struct hw\_module\_t;              //模块类型
2. struct hw\_module\_methods\_t;      //模块方法
3. struct hw\_device\_t;              //设备类型

这几个数据结构是在Android工作目录/hardware/libhardware/include/hardware/hardware.h文件中定义.

**3 解释**

一般来说，在写HAL相关代码时都得包含这个hardware.h头文件，所以有必要先了解一下这个头文件中的内容.

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7086273) [copy](http://blog.csdn.net/flydream0/article/details/7086273)

1. /\*
2. \* Copyright (C) 2008 The Android Open Source Project
3. \*
4. \* Licensed under the Apache License, Version 2.0 (the "License");
5. \* you may not use this file except in compliance with the License.
6. \* You may obtain a copy of the License at
7. \*
8. \*      http://www.apache.org/licenses/LICENSE-2.0
9. \*
10. \* Unless required by applicable law or agreed to in writing, software
11. \* distributed under the License is distributed on an "AS IS" BASIS,
12. \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13. \* See the License for the specific language governing permissions and
14. \* limitations under the License.
15. \*/
17. #ifndef ANDROID\_INCLUDE\_HARDWARE\_HARDWARE\_H
18. #define ANDROID\_INCLUDE\_HARDWARE\_HARDWARE\_H
20. #include <stdint.h>
21. #include <sys/cdefs.h>
23. #include <cutils/native\_handle.h>
24. #include <system/graphics.h>
26. \_\_BEGIN\_DECLS
28. /\*
29. \* Value for the hw\_module\_t.tag field
30. \*/
32. #define MAKE\_TAG\_CONSTANT(A,B,C,D) (((A) << 24) | ((B) << 16) | ((C) << 8) | (D))
34. #define HARDWARE\_MODULE\_TAG MAKE\_TAG\_CONSTANT('H', 'W', 'M', 'T')
35. #define HARDWARE\_DEVICE\_TAG MAKE\_TAG\_CONSTANT('H', 'W', 'D', 'T')
37. **struct** hw\_module\_t;
38. **struct** hw\_module\_methods\_t;
39. **struct** hw\_device\_t;
41. /\*\*
42. \* Every hardware module must have a data structure named HAL\_MODULE\_INFO\_SYM
43. \* and the fields of this data structure must begin with hw\_module\_t
44. \* followed by module specific information.
45. \*/
46. //每一个硬件模块都每必须有一个名为HAL\_MODULE\_INFO\_SYM的数据结构变量，它的第一个成员的类型必须为hw\_module\_t
47. **typedef** **struct** hw\_module\_t {
48. /\*\* tag must be initialized to HARDWARE\_MODULE\_TAG \*/
49. uint32\_t tag;
51. /\*\* major version number for the module \*/
52. uint16\_t version\_major;
54. /\*\* minor version number of the module \*/
55. uint16\_t version\_minor;
57. /\*\* Identifier of module \*/
58. **const** **char** \*id;
60. /\*\* Name of this module \*/
61. **const** **char** \*name;
63. /\*\* Author/owner/implementor of the module \*/
64. **const** **char** \*author;
66. /\*\* Modules methods \*/
67. //模块方法列表,指向hw\_module\_methods\_t\*
68. **struct** hw\_module\_methods\_t\* methods;
70. /\*\* module's dso \*/
71. **void**\* dso;
73. /\*\* padding to 128 bytes, reserved for future use \*/
74. uint32\_t reserved[32-7];
76. } hw\_module\_t;
78. **typedef** **struct** hw\_module\_methods\_t {                 //硬件模块方法列表的定义，这里只定义了一个open函数
79. /\*\* Open a specific device \*/
80. **int** (\*open)(**const** **struct** hw\_module\_t\* module, **const** **char**\* id, //注意这个open函数明确指出第三个参数的类型为struct hw\_device\_t\*\*
81. **struct** hw\_device\_t\*\* device);
82. } hw\_module\_methods\_t;
84. /\*\*
85. \* Every device data structure must begin with hw\_device\_t
86. \* followed by module specific public methods and attributes.
87. \*/
88. //每一个设备数据结构的第一个成员函数必须是hw\_device\_t类型，其次才是各个公共方法和属性
89. **typedef** **struct** hw\_device\_t {
90. /\*\* tag must be initialized to HARDWARE\_DEVICE\_TAG \*/
91. uint32\_t tag;
93. /\*\* version number for hw\_device\_t \*/
94. uint32\_t version;
96. /\*\* reference to the module this device belongs to \*/
97. **struct** hw\_module\_t\* module;
99. /\*\* padding reserved for future use \*/
100. uint32\_t reserved[12];
102. /\*\* Close this device \*/
103. **int** (\*close)(**struct** hw\_device\_t\* device);
105. } hw\_device\_t;
107. /\*\*
108. \* Name of the hal\_module\_info
109. \*/
110. #define HAL\_MODULE\_INFO\_SYM         HMI
112. /\*\*
113. \* Name of the hal\_module\_info as a string
114. \*/
115. #define HAL\_MODULE\_INFO\_SYM\_AS\_STR  "HMI"
117. /\*\*
118. \* Get the module info associated with a module by id.
119. \*
120. \* @return: 0 == success, <0 == error and \*module == NULL
121. \*/
122. **int** hw\_get\_module(**const** **char** \*id, **const** **struct** hw\_module\_t \*\*module);
124. /\*\*
125. \* Get the module info associated with a module instance by class 'class\_id'
126. \* and instance 'inst'.
127. \*
128. \* Some modules types necessitate multiple instances. For example audio supports
129. \* multiple concurrent interfaces and thus 'audio' is the module class
130. \* and 'primary' or 'a2dp' are module interfaces. This implies that the files
131. \* providing these modules would be named audio.primary.<variant>.so and
132. \* audio.a2dp.<variant>.so
133. \*
134. \* @return: 0 == success, <0 == error and \*module == NULL
135. \*/
136. **int** hw\_get\_module\_by\_class(**const** **char** \*class\_id, **const** **char** \*inst,
137. **const** **struct** hw\_module\_t \*\*module);
139. \_\_END\_DECLS
141. #endif  /\* ANDROID\_INCLUDE\_HARDWARE\_HARDWARE\_H \*/

由以上内容可以看出(typedef struct hw\_module\_t ,typedef struct hw\_device\_t)，如果我们要写一个自定义设备的驱动的HAL层时，我们得首先自定义两个数据结构:

假设我们要做的设备名为XXX:

在头文件中定义:XXX.h

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7086273) [copy](http://blog.csdn.net/flydream0/article/details/7086273)

1. /\*定义模块ID\*/
2. #define XXX\_HARDWARE\_MODULE\_ID "XXX"
4. /\*硬件模块结构体\*/
5. //见hardware.h中的hw\_module\_t定义的说明，xxx\_module\_t的第一个成员必须是hw\_module\_t类型，其次才是模块的一此相关信息，当然也可以不定义，
6. //这里就没有定义模块相关信息
7. **struct** xxx\_module\_t {
8. **struct** hw\_module\_t common;
9. };
11. /\*硬件接口结构体\*/
12. //见hardware.h中的hw\_device\_t的说明，要求自定义xxx\_device\_t的第一个成员必须是hw\_device\_t类型，其次才是其它的一些接口信息.
13. **struct** xxx\_device\_t {
14. **struct** hw\_device\_t common;
15. //以下成员是HAL对上层提供的接口或一些属性
16. **int** fd;
17. **int** (\*set\_val)(**struct** xxx\_device\_t\* dev, **int** val);
18. **int** (\*get\_val)(**struct** xxx\_device\_t\* dev, **int**\* val);
19. };

注:特别注意xxx\_device\_t的结构定义，这个才是HAL向上层提供接口函数的数据结构,其成员就是我们想要关心的接口函数.

接下来我们在实现文件XXX.c文件中定义一个xxx\_module\_t的变量:

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7086273) [copy](http://blog.csdn.net/flydream0/article/details/7086273)

1. /\*模块实例变量\*/
2. **struct** xxx\_module\_t HAL\_MODULE\_INFO\_SYM = {    //变量名必须为HAL\_MODULE\_INFO\_SYM,这是强制要求的，你要写Android的HAL就得遵循这个游戏规则,
3. //见hardware.h中的hw\_module\_t的类型信息说明.
4. common: {
5. tag: HARDWARE\_MODULE\_TAG,
6. version\_major: 1,
7. version\_minor: 0,
8. id: XXX\_HARDWARE\_MODULE\_ID,    //头文件中有定义
9. name: MODULE\_NAME,
10. author: MODULE\_AUTHOR,
11. methods: &xxx\_module\_methods,  //模块方法列表，在本地定义
12. }
13. };

注意到上面有HAL\_MODULE\_INFO\_SYM变量的成员common中包含一个函数列表xxx\_module\_methods,而这个成员函数列表是在本地自定义的。那么这个成员函数列是不是就是HAL向上层提供函数的地方呢？很失望，不是在这里，前面我们已经说过了，是在xxx\_device\_t中定义的,这个xxx\_module\_methods实际上只提供了一个open函数，就相当于只提供了一个模块初始化函数.其定义如下:

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7086273) [copy](http://blog.csdn.net/flydream0/article/details/7086273)

1. /\*模块方法表\*/
2. **static** **struct** hw\_module\_methods\_t xxx\_module\_methods = {
3. open: xxx\_device\_open
4. };

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

注意到，上边的函数列表中只列出了一个xxx\_device\_open函数，这个函数也是需要在本地实现的一个函数。前面说过，这个函数只相当于模块初始化函数。

那么HAL又到底是怎么将xxx\_device\_t中定义的接口提供到上层去的呢?

且看上面这个函数列表中唯一的一个xxx\_device\_open的定义:

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7086273) [copy](http://blog.csdn.net/flydream0/article/details/7086273)

1. **static** **int** xxx\_device\_open(**const** **struct** hw\_module\_t\* module, **const** **char**\* name, **struct** hw\_device\_t\*\* device) {
2. **struct** xxx\_device\_t\* dev;
3. dev = (**struct** hello\_device\_t\*)malloc(**sizeof**(**struct** xxx\_device\_t));//动态分配空间
5. **if**(!dev) {
6. LOGE("Hello Stub: failed to alloc space");
7. **return** -EFAULT;
8. }
10. memset(dev, 0, **sizeof**(**struct** xxx\_device\_t));
11. //对dev->common的内容赋值，
12. dev->common.tag = HARDWARE\_DEVICE\_TAG;
13. dev->common.version = 0;
14. dev->common.module = (hw\_module\_t\*)module;
15. dev->common.close = xxx\_device\_close;
16. //对dev其它成员赋值
17. dev->set\_val = xxx\_set\_val;
18. dev->get\_val = xxx\_get\_val;
20. **if**((dev->fd = open(DEVICE\_NAME, O\_RDWR)) == -1) {
21. LOGE("Hello Stub: failed to open /dev/hello -- %s.", strerror(errno));
22. free(dev);
23. **return** -EFAULT;
24. }
26. //输出&(dev->common),输出的并不是dev,而是&(dev->common)!(common内不是只包含了一个close接口吗?)
27. \*device = &(dev->common);
28. LOGI("Hello Stub: open /dev/hello successfully.");
30. **return** 0;
31. }

经验告诉我们，一般在进行模块初始化的时候，模块的接口函数也会“注册”，上面是模块初始化函数，那么接口注册在哪？于是我们找到\*device =&(dev->common);这行代码，可问题是，这样一来，返回给调用者不是&(dev->common)吗？而这个dev->common仅仅只包含了一个模块关闭接口！到底怎么回事？为什么不直接返回dev,dev下不是提供所有HAL向上层接口吗?

在回答上述问题之前，让我们先看一下这xxx\_device\_open函数原型,还是在hardware.h头文件中,找到下面几行代码:

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7086273) [copy](http://blog.csdn.net/flydream0/article/details/7086273)

1. **typedef** **struct** hw\_module\_methods\_t {
2. /\*\* Open a specific device \*/
3. **int** (\*open)(**const** **struct** hw\_module\_t\* module, **const** **char**\* id,
4. **struct** hw\_device\_t\*\* device);
6. } hw\_module\_methods\_t;

这是方法列表的定义，明确要求了方法列表中有且只一个open方法，即相当于模块初始化方法，且，这个方法的第三个参数明确指明了类型是struct hw\_device\_t \*\*,而不是用户自定义的xxx\_device\_t,这也就是解译了在open函数实现内为什么输出的是&(dev->common)而不是dev了，原来返回的类型在hardware.h中的open函数原型中明确指出只能返回hw\_device\_t类型.

可是，dev->common不是只包含close接口吗？做为HAL的上层，它又是怎么"看得到"HAL提供的全部接口的呢?

接下来，让我们来看看做为HAL上层，它又是怎么使用由HAL返回的dev->common的:

参考: [在Ubuntu为Android硬件抽象层（HAL）模块编写JNI方法提供Java访问硬件服务接口](http://blog.csdn.net/luoshengyang/article/details/6575988) 这篇文章,从中可以看到这么几行代码：

**[cpp]** [view plain](http://blog.csdn.net/flydream0/article/details/7086273) [copy](http://blog.csdn.net/flydream0/article/details/7086273)

1. /\*通过硬件抽象层定义的硬件模块打开接口打开硬件设备\*/
2. **static** **inline** **int** hello\_device\_open(**const** hw\_module\_t\* module, **struct** hello\_device\_t\*\* device) {
3. **return** module->methods->open(module, HELLO\_HARDWARE\_MODULE\_ID, (**struct** hw\_device\_t\*\*)device);
4. }

由此可见，返回的&(dev->common)最终会返回给struce hello\_device\_t \*\*类型的输出变量device,换句话说，类型为hw\_device\_t的dev->common在初始化函数open返回后，会强制转化为xxx\_device\_t来使用，终于明白了，原来如此！另外，在hardware.h中对xxx\_device\_t类型有说明，要求它的第一个成员的类型必须是hw\_device\_t，原来是为了HAL上层使用时的强制转化的目的，如果xxx\_device\_t的第一个成员类型不是hw\_device\_t，那么HAL上层使用中强制转化就没有意义了，这个时候，就真的“看不到”HAL提供的接口了.

此外，在hardware.h头文件中，还有明确要求定义xxx\_module\_t类型时，明确要求第一个成员变量类型必须为hw\_module\_t，这也是为了方便找到其第一个成员变量common,进而找到本地定义的方法列表，从而调用open函数进行模块初始化.

综上所述，HAL是通过struct xxx\_device\_t这个结构体向上层提供接口的.

即：接口包含在struct xxx\_device\_t这个结构体内。

而具体执行是通过struct xxx\_module\_t HAL\_MODULE\_INFO\_SYM这个结构体变量的函数列表成员下的open函数来返回给上层的.