**目 录**

[1 Android Camera 1](#_Toc402345950)

[1.1 overview 1](#_Toc402345951)

[1.1.1 Function 1](#_Toc402345952)

[1.1.2 系统组成 1](#_Toc402345953)

[1.1.3 系统结构 6](#_Toc402345954)

[1.1.4 源码分布 6](#_Toc402345955)

[1.1.5 预览（Preview） 7](#_Toc402345956)

[1.1.6 拍照(takePicture) 10](#_Toc402345957)

[1.1.7 录像(Recording) 12](#_Toc402345958)

[1.2 camera hal 13](#_Toc402345959)

[1.2.1 function 13](#_Toc402345960)

[1.2.2 interface 13](#_Toc402345961)

[1.2.3 设备访问 19](#_Toc402345962)

[1.2.4 消息响应 20](#_Toc402345963)

[1.2.5 callback 20](#_Toc402345964)

[1.3 Hal Porting 22](#_Toc402345965)

[1.3.1 环境建立 22](#_Toc402345966)

[1.3.2 ModuleInfo 22](#_Toc402345967)

[1.3.3 系统结构 23](#_Toc402345968)

[1.3.4 图像捕获 24](#_Toc402345969)

[1.3.5 图像显示 25](#_Toc402345970)

[1.3.6 数据回传 27](#_Toc402345971)

[1.3.7 拍照 27](#_Toc402345972)

[1.3.8 录像 27](#_Toc402345973)

[2 Linux Camera 29](#_Toc402345974)

[2.1 V4L2 29](#_Toc402345975)

[2.1.1 使用流程 29](#_Toc402345976)

[2.1.2 常用命令 29](#_Toc402345977)

[2.1.3 设备控制 30](#_Toc402345978)

[2.2 UVC 33](#_Toc402345979)

[2.2.1 uvc 驱动 33](#_Toc402345980)

[2.2.2 usb camera 33](#_Toc402345981)

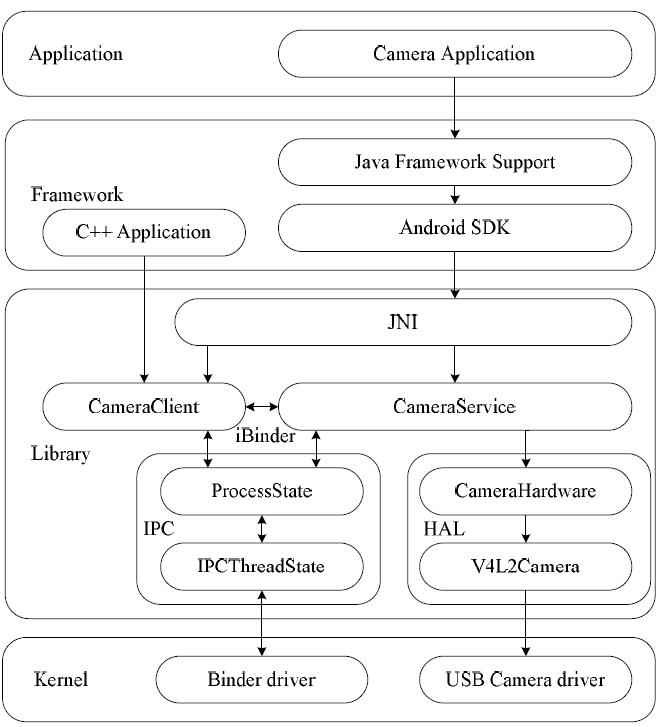
# Android Camera

## overview

### Function

Camera 在Andriod中属于视频输入系统，其一般包含预览、录像和拍照的功能。

### 系统组成



1. app

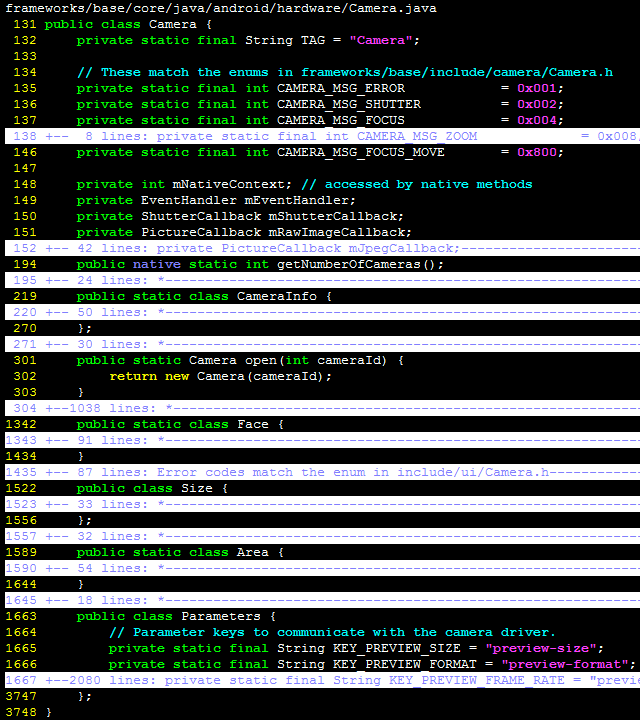
通过调用Framework层的android.hardware.camera类来实现具体的功能由JAVA 代码实现，最终打包成APK。

原生应用代码在Camera.java(packages\apps\camera\src\com\android\camera。

1. framework

提供android.hardware.camera类给app使用。由JAVA代码实现，通过JNI机制调用native代码实现的具体功能。

1. 源码：frameworks/base/core/java/android/hardware/Camera.java。



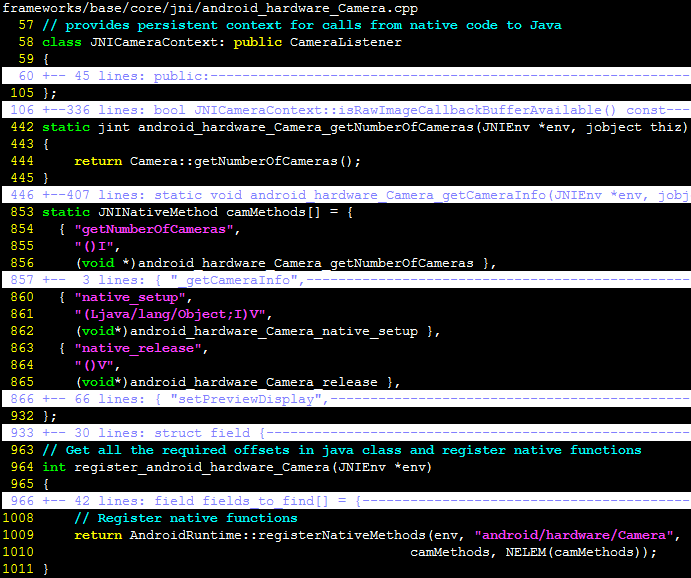
1. app参数传递

参数以map的方式组织存储，使用Parameters类中形如“preview-size=640X480;preview-format=yuv422sp;”格式的字符串传到service端。

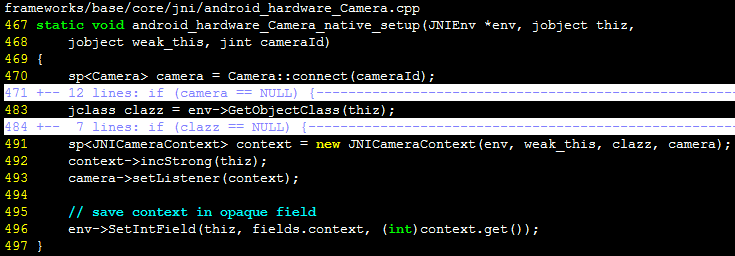
1. JNI

实现framework中JAVA类android.hardware.Camera的本地代码调用。

1. 源码： frameworks/base/core/jni/android\_hardware\_Camera.cpp。



1. 通过register\_android\_hardware\_Camera(JNIEnv \*env)函数(cpp)将native函数注册到虚拟机中，以供framework层的JAVA 代码调用。
2. 通过native\_setup/native\_release函数(java)打开/关闭camera。



1. native函数通过调用libcamera\_client.so中的**Camera**类实现具体的功能。
2. client

client相关的类源码位置：frameworks/av/camera/,各类实现文件如下：

**Camera**类-->Camera.cpp,Camera.h

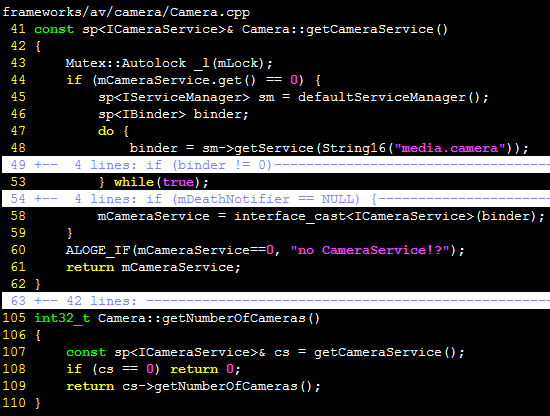
CameraParameters类-->CameraParameters.cpp,CameraParameters.h

Icamera类-->ICamera.cpp,ICamera.h

IcameraClient类-->ICameraClient.cpp,ICameraClient.h

IcameraService类-->ICameraService.cpp,ICameraService.h

1. Icamera、IcameraClient、IcameraService三个类是按照Binder IPC通信要求的框架实现的，用来与service端交互来实现具体功能。
2. CameraParameters 接收android.hardware.camera::Parameters类为参数，解析与格式化所有的参数设置。
3. Camera继承自BnCameraClient🡪ICameraClient。

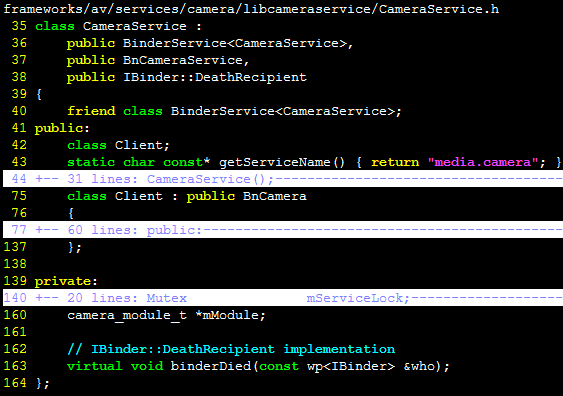


1. service

service相关的类源码位置：

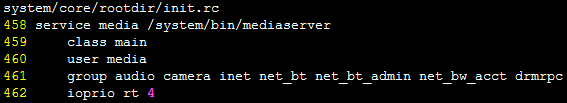
frameworks/av/services/camera/libcameraservice，编译入libcameraservice.so,如下：

1. CameraService类，继承自BnCameraService🡪ICameraService。

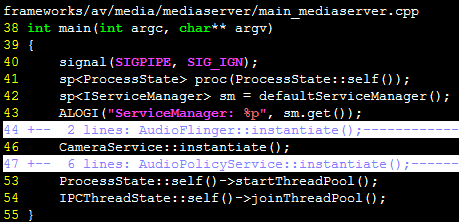


1. CameraService::Client类，继承自BnCamera🡪ICamera，通过调用HAL层来实现具体的功能。
2. Service启动

在init.rc中有如下代码用于执行/system/bin/mediaserver，启动多媒体服务进程。

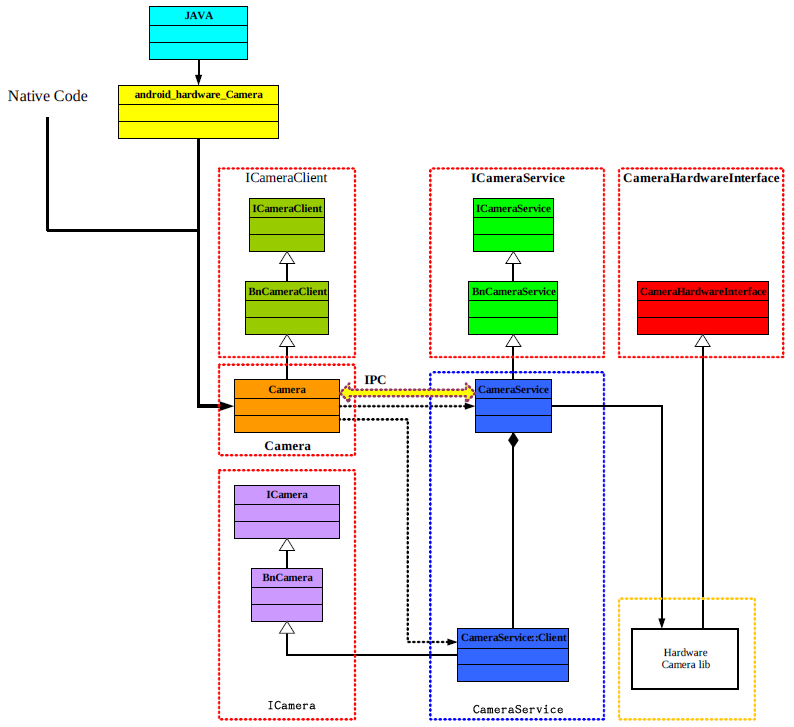


mediaserver的实现中创建了CameraService实例，名称为“media.camera”。



### 系统结构

各模块之间的结构如下：



### 源码分布

Camera部分框架部分包含在ui库中，中间层的实现是CameraService，通过调用下层的Camera HAL来实现功能。

Android2.3.4中CameraHardwareInterface是一个纯虚基类,实际的Camera HAL层只需要实现CameraHardwareInterface中定义的所有的接口即可.

在Android4.0中, CameraHardwareInterface不再是一个简单的纯虚接口类, 而是将所有的接口实现都转接到了camera\_device\_t对象中。

1. Android 4.0 (ICS)
2. Android Framework

Java: framwworks/base/core/java/android/hardware

JNI: frameworks/base/core/jni

1. Camera Service

frameworks/base/services/camera/libcameraservice/

1. IBinder Interfaces

frameworks/base/include/camera/ICamera.h

1. IBinder Implementation

frameworks/base/libs/camera/ICamera.cpp etc

1. Camera HAL Interface

frameworks/base/services/camera/libcameraservice/CameraHardwareInterface.h

1. Camera HAL

hardware/<vendor>/camera

1. Android 4.2 (Jelly Bean)
2. Android Framework

Java: frameworks/base/core/java/android/hardware

JNI: frameworks/base/core/jni

1. Camera Service

frameworks/av/services/camera/libcameraservice/

1. IBinder Interfaces

frameworks/av/include/camera/ICamera.h etc.

1. IBinder Implementation

frameworks/av/camera/ICamera.cpp etc.

1. Camera HAL Interface

frameworks/av/services/camera/libcameraservice/CameraHardwareInterface.h

1. Camera HAL

hardware/<vendor>/camera

### 预览（Preview）

应用程序Camera 调用startpreview 接口，利用底层的Binder 机制，通过客户端CameraClient 通知服务 端CameraService，服务端通过内部类调用硬件抽象层中提供的功能，完成具体的操作。

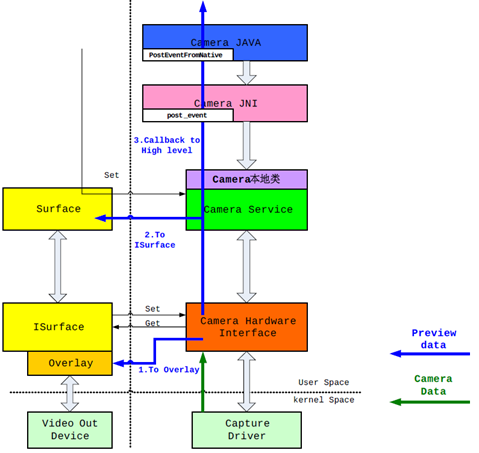
硬件抽象层的CameraHardware 创建previewthread 线程和Loop 线程， 前者通过GrabPreveiwFrame 接口取得V4L2Camera 采集摄像头的一帧数据，放入指定缓冲区，并通过convert 将数据转换为RGB565 ，之后向服务端发送取景消息

CAMERA\_MSG\_PREVIEW\_FRAME，从而实现了底层数据到缓冲区ISurface 的映射；后者将硬件设备采集的数据通过postBuffer不断的传递给显示缓冲区framebuffer，完成取景显示。

1. 数据流

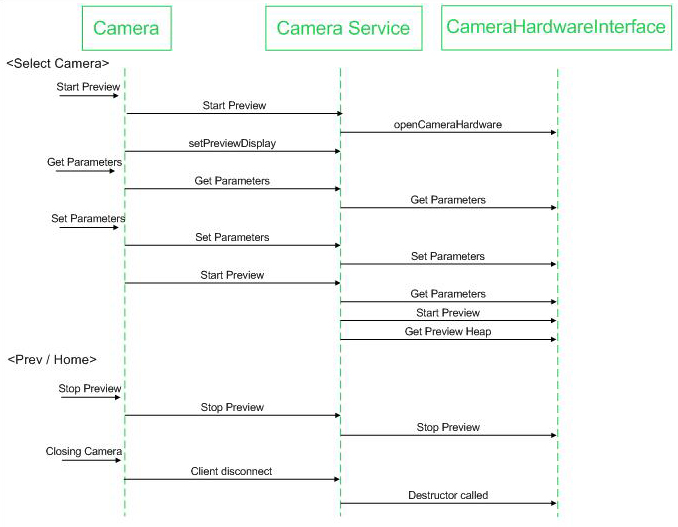
preview 数据不会通过Binder IPC机制从service端复制到client端，但会通过回调函数与消息的机制将preview 数据的buffer地址传到client端，最终可在JAVA AP中操作处理这个preview数据。实现有3种方式：

1. Camera类通过Callback送给上层，由上层处理。
2. 在CameraHAL中，直接送给Overlay。
3. 在CameraService中，调用ISurface的postBuffer接口，送出数据。



在Android照相机/摄像机应用程序中，使用的是b和c这两种方法，具体由CameraService读取CameraHAL的useOverlay 接口来实现。

1. 系统运行



1. 打开内核设备文件。

CameraHardwareInterface.h中定义的 openCameraHardware()打开linux kernel中的camera driver的设备文件（如/dev/video0），创建初始化一些相关的类的实例。

1. 设置摄像头的工作参数。

CameraHardwareInterface.h中定义的 setParameters()函数，在这一步可以通过参数告诉camera HAL使用哪一个硬件摄像头，以及它工作的参数（size，format等），并在HAL层分配存储preview数据的buffers（如果buffers是在linux kernel中的camera driver中分配的，在这一步也会拿到这些buffers mmap后的地址指针）。

1. 设置显示目标。

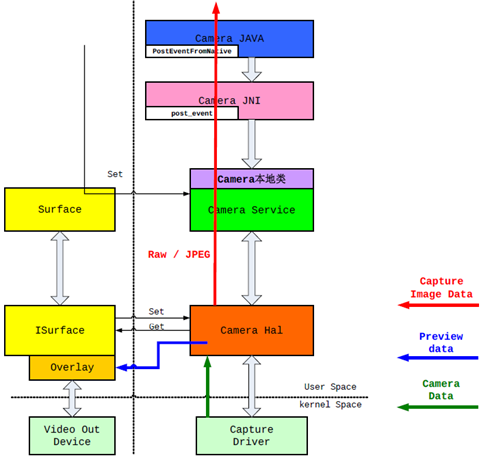
需在JAVA APP中创建一个surface然后传递到CameraService中。会调用到libcameraservice.so中的setPreviewDisplay(const sp<ISurface>& surface)函数中。在这里分两种情况考虑：一种是不使用overlay；一种是使用overlay显示。如果不使用overlay那设置显示目标最后就在libcameraservice.so中，不会进Camera HAL动态库。并将上一步拿到的preview数据buffers地址注册到surface中。 如果使用overlay那在libcameraservice.so中会通过传进来的Isurface创建Overlay类的实例，然后调用CameraHardwareInterface.h中定义的 setOverlay()设置到Camera HAL动态库中。

1. 开始preview工作。

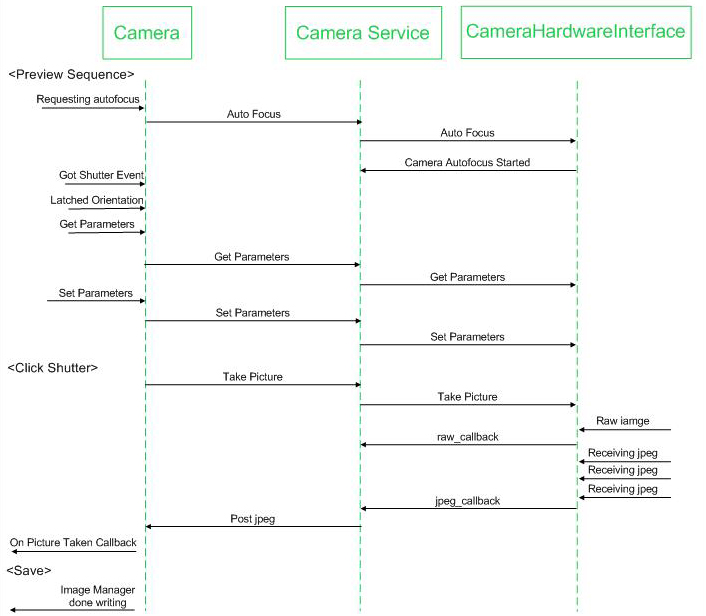
最终调用到CameraHardwareInterface.h中定义的 startPreview()函数。如果不使用overlay，Camera HAL得到linux kernel中的preview数据后回调通知到libcameraservice.so中。在libcameraservice.so中会使用上一步的surface进行显示。如果使用overlay，Camera HAL得到linux kernel中的preview数据后直接交给Overlay对象，然后有Overlay HAL去显示。

### 拍照(takePicture)

1. 数据流



1. 系统运行



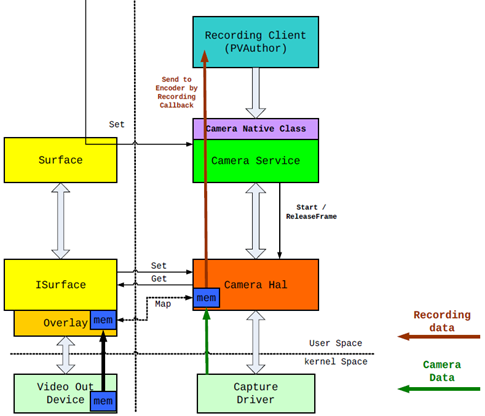
在取景模式下，通过ClickShutterButton 接口，照相机从取景模式进入拍照模式。在拍照模式下， 应用程序通过发送focus 请求给硬件抽象层中的CameraHardware ，CameraHardware 将完成focus 请求的具体操作，并通过notify 回调函数返回消息CAMERA\_MSG\_FOCUS。之后上层通过takepicture 接口启动硬件抽象层中的captureThread线程，线程启动之后将通过notify 回调函数向服务端CameraService 发送消息CAMERA\_MSG\_SHUTTER，此时进入真正的拍照模式。

接下来CameraHardware 将调用底层的GrabRawFrame 获取图像照片，首先将原始数据转化为与显示缓冲区一致的数据格式，通过回调函数notify 向服务端CameraService 发送消息CAMERA\_MSG\_RAW\_IMAGE，服务端收到消息后调用RawPicture，然后由SurfaceFlinger 完成照片的显示。

随后，底层将调用GrabJpegFrame返回Jpeg格式的照片，通过回调函数notify向服务端CameraService发送消息CAMERA\_MSG\_COMPRESSED\_IMAGE，服务端收到消息后调用handleCompressedPicture 将消息和照片数据内容都发送给客户端、JNI、上层应用程序，从而实现照片的存储。

### 录像(Recording)

1. 数据流



## camera hal

### function

Android通过HAL层来保证底层硬件（驱动）改变时，只需修改对应的HAL层代码，这样frameWork层与app的都不用改变。

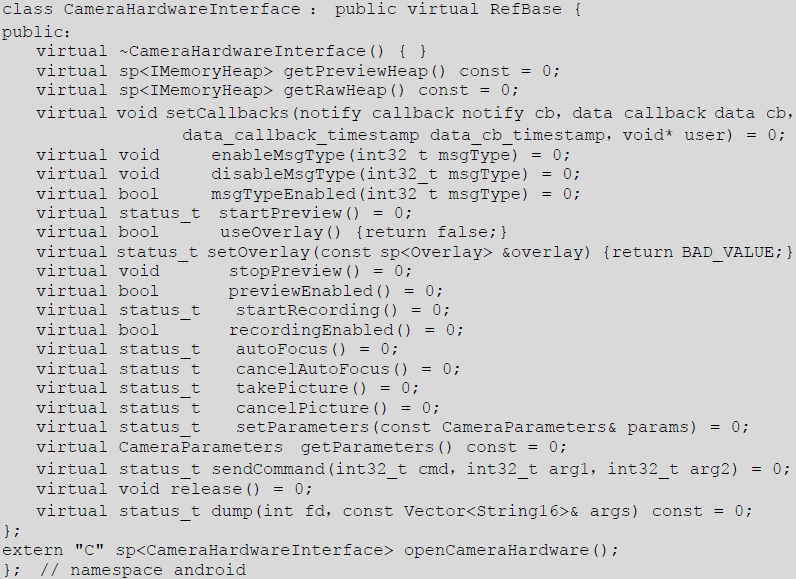
camera HAL一般编译为一个动态库libcamera.so（动态库名字可以改，只需要与Android.mk一致即可)。

### interface

HAL通过被Libcameraservice.so::CameraService::Client调用而实现具体camera功能，HAL需要实现HAL接口类。

1. 类的定义文件如下：

frameworks/base/include/camera/CameraHardwareInterface.h



在Android4.0版本以后，camera的HAL实现已经不直接使用该类，而是通过该类间接调用hal api实现。

1. API版本

在hardware/libhardware/include/hardware/中定义了两个camera api的规范，分别为：camera2.h(API2.0) 与camera.h(API1.0)。

在实现HAL层时模块信息需要和实现的API版本信息一致。

1. API1.0模块信息

hardware/libhardware/include/hardware/camera.h

282 **typedef** **struct** camera\_device {

283 **/\*\***

284  **\* camera\_device.common.version must be in the range**

285  **\* HARDWARE\_DEVICE\_API\_VERSION(0,0)-(1,FF). CAMERA\_DEVICE\_API\_VERSION\_1\_0 is**

286  **\* recommended.**

287  **\*/**

288 hw\_device\_t common;

289 camera\_device\_ops\_t \*ops;

290 **void** \*priv;

291 } camera\_device\_t;

292

1. API1.0接口信息

hardware/libhardware/include/hardware/camera.h

96 **typedef** **struct** camera\_device\_ops {

97 **/\*\* Set the ANativeWindow to which preview frames are sent \*/**

98 **int** (\*set\_preview\_window)(**struct** camera\_device \*,

99 **struct** preview\_stream\_ops \*window);

100

101 **/\*\* Set the notification and data callbacks \*/**

102 **void** (\*set\_callbacks)(**struct** camera\_device \*,

103 camera\_notify\_callback notify\_cb,

104 camera\_data\_callback data\_cb,

105 camera\_data\_timestamp\_callback data\_cb\_timestamp,

106 camera\_request\_memory get\_memory,

107 **void** \*user);

108

109 **/\*\***

110  **\* The following three functions all take a msg\_type, which is a bitmask of**

111  **\* the messages defined in include/ui/Camera.h**

112  **\*/**

113

114 **/\*\***

115  **\* Enable a message, or set of messages.**

116  **\*/**

117 **void** (\*enable\_msg\_type)(**struct** camera\_device \*, **int32\_t** msg\_type);

118

119 **/\*\***

120  **\* Disable a message, or a set of messages.**

121  **\***

122  **\* Once received a call to disableMsgType(CAMERA\_MSG\_VIDEO\_FRAME), camera**

123  **\* HAL should not rely on its client to call releaseRecordingFrame() to**

124  **\* release video recording frames sent out by the cameral HAL before and**

125  **\* after the disableMsgType(CAMERA\_MSG\_VIDEO\_FRAME) call. Camera HAL**

126  **\* clients must not modify/access any video recording frame after calling**

127  **\* disableMsgType(CAMERA\_MSG\_VIDEO\_FRAME).**

128  **\*/**

129 **void** (\*disable\_msg\_type)(**struct** camera\_device \*, **int32\_t** msg\_type);

130

131 **/\*\***

132  **\* Query whether a message, or a set of messages, is enabled. Note that**

133  **\* this is operates as an AND, if any of the messages queried are off, this**

134  **\* will return false.**

135  **\*/**

136 **int** (\*msg\_type\_enabled)(**struct** camera\_device \*, **int32\_t** msg\_type);

137

138 **/\*\***

139  **\* Start preview mode.**

140  **\*/**

141 **int** (\*start\_preview)(**struct** camera\_device \*);

142

143 **/\*\***

144  **\* Stop a previously started preview.**

145  **\*/**

146 **void** (\*stop\_preview)(**struct** camera\_device \*);

147

148 **/\*\***

149  **\* Returns true if preview is enabled.**

150  **\*/**

151 **int** (\*preview\_enabled)(**struct** camera\_device \*);

152

153 **/\*\***

154  **\* Request the camera HAL to store meta data or real YUV data in the video**

155  **\* buffers sent out via CAMERA\_MSG\_VIDEO\_FRAME for a recording session. If**

156  **\* it is not called, the default camera HAL behavior is to store real YUV**

157  **\* data in the video buffers.**

158  **\***

159  **\* This method should be called before startRecording() in order to be**

160  **\* effective.**

161  **\***

162  **\* If meta data is stored in the video buffers, it is up to the receiver of**

163  **\* the video buffers to interpret the contents and to find the actual frame**

164  **\* data with the help of the meta data in the buffer. How this is done is**

165  **\* outside of the scope of this method.**

166  **\***

167  **\* Some camera HALs may not support storing meta data in the video buffers,**

168  **\* but all camera HALs should support storing real YUV data in the video**

169  **\* buffers. If the camera HAL does not support storing the meta data in the**

170  **\* video buffers when it is requested to do do, INVALID\_OPERATION must be**

171  **\* returned. It is very useful for the camera HAL to pass meta data rather**

172  **\* than the actual frame data directly to the video encoder, since the**

173  **\* amount of the uncompressed frame data can be very large if video size is**

174  **\* large.**

175  **\***

176  **\* @param enable if true to instruct the camera HAL to store**

177  **\* meta data in the video buffers; false to instruct**

178  **\* the camera HAL to store real YUV data in the video**

179  **\* buffers.**

180  **\***

181  **\* @return OK on success.**

182  **\*/**

183 **int** (\*store\_meta\_data\_in\_buffers)(**struct** camera\_device \*, **int** enable);

184

185 **/\*\***

186  **\* Start record mode. When a record image is available, a**

187  **\* CAMERA\_MSG\_VIDEO\_FRAME message is sent with the corresponding**

188  **\* frame. Every record frame must be released by a camera HAL client via**

189  **\* releaseRecordingFrame() before the client calls**

190  **\* disableMsgType(CAMERA\_MSG\_VIDEO\_FRAME). After the client calls**

191  **\* disableMsgType(CAMERA\_MSG\_VIDEO\_FRAME), it is the camera HAL's**

192  **\* responsibility to manage the life-cycle of the video recording frames,**

193  **\* and the client must not modify/access any video recording frames.**

194  **\*/**

195 **int** (\*start\_recording)(**struct** camera\_device \*);

196

197 **/\*\***

198  **\* Stop a previously started recording.**

199  **\*/**

200 **void** (\*stop\_recording)(**struct** camera\_device \*);

201

202 **/\*\***

203  **\* Returns true if recording is enabled.**

204  **\*/**

205 **int** (\*recording\_enabled)(**struct** camera\_device \*);

206

207 **/\*\***

208  **\* Release a record frame previously returned by CAMERA\_MSG\_VIDEO\_FRAME.**

209  **\***

210  **\* It is camera HAL client's responsibility to release video recording**

211  **\* frames sent out by the camera HAL before the camera HAL receives a call**

212  **\* to disableMsgType(CAMERA\_MSG\_VIDEO\_FRAME). After it receives the call to**

213  **\* disableMsgType(CAMERA\_MSG\_VIDEO\_FRAME), it is the camera HAL's**

214  **\* responsibility to manage the life-cycle of the video recording frames.**

215  **\*/**

216 **void** (\*release\_recording\_frame)(**struct** camera\_device \*,

217 **const** **void** \*opaque);

218

219 **/\*\***

220  **\* Start auto focus, the notification callback routine is called with**

221  **\* CAMERA\_MSG\_FOCUS once when focusing is complete. autoFocus() will be**

222  **\* called again if another auto focus is needed.**

223  **\*/**

224 **int** (\*auto\_focus)(**struct** camera\_device \*);

225

226 **/\*\***

227  **\* Cancels auto-focus function. If the auto-focus is still in progress,**

228  **\* this function will cancel it. Whether the auto-focus is in progress or**

229  **\* not, this function will return the focus position to the default. If**

230  **\* the camera does not support auto-focus, this is a no-op.**

231  **\*/**

232 **int** (\*cancel\_auto\_focus)(**struct** camera\_device \*);

233

234 **/\*\***

235  **\* Take a picture.**

236  **\*/**

237 **int** (\*take\_picture)(**struct** camera\_device \*);

238

239 **/\*\***

240  **\* Cancel a picture that was started with takePicture. Calling this method**

241  **\* when no picture is being taken is a no-op.**

242  **\*/**

243 **int** (\*cancel\_picture)(**struct** camera\_device \*);

244

245 **/\*\***

246  **\* Set the camera parameters. This returns BAD\_VALUE if any parameter is**

247  **\* invalid or not supported.**

248  **\*/**

249 **int** (\*set\_parameters)(**struct** camera\_device \*, **const** **char** \*parms);

250

251 **/\*\* Retrieve the camera parameters. The buffer returned by the camera HAL**

252  **must be returned back to it with put\_parameters, if put\_parameters**

253  **is not NULL.**

254  **\*/**

255 **char** \*(\*get\_parameters)(**struct** camera\_device \*);

256

257 **/\*\* The camera HAL uses its own memory to pass us the parameters when we**

258  **call get\_parameters. Use this function to return the memory back to**

259  **the camera HAL, if put\_parameters is not NULL. If put\_parameters**

260  **is NULL, then you have to use free() to release the memory.**

261  **\*/**

262 **void** (\*put\_parameters)(**struct** camera\_device \*, **char** \*);

263

264 **/\*\***

265  **\* Send command to camera driver.**

266  **\*/**

267 **int** (\*send\_command)(**struct** camera\_device \*,

268 **int32\_t** cmd, **int32\_t** arg1, **int32\_t** arg2);

269

270 **/\*\***

271  **\* Release the hardware resources owned by this object. Note that this is**

272  **\* \*not\* done in the destructor.**

273  **\*/**

274 **void** (\*release)(**struct** camera\_device \*);

275

276 **/\*\***

277  **\* Dump state of the camera hardware**

278  **\*/**

279 **int** (\*dump)(**struct** camera\_device \*, **int** fd);

280 } camera\_device\_ops\_t;

1. API1.0 preview接口

hardware/libhardware/include/hardware/camera.h

70 **typedef** **struct** preview\_stream\_ops {

71 **int** (\*dequeue\_buffer)(**struct** preview\_stream\_ops\* w,

72 buffer\_handle\_t\*\* buffer, **int** \*stride);

73 **int** (\*enqueue\_buffer)(**struct** preview\_stream\_ops\* w,

74 buffer\_handle\_t\* buffer);

75 **int** (\*cancel\_buffer)(**struct** preview\_stream\_ops\* w,

76 buffer\_handle\_t\* buffer);

77 **int** (\*set\_buffer\_count)(**struct** preview\_stream\_ops\* w, **int** count);

78 **int** (\*set\_buffers\_geometry)(**struct** preview\_stream\_ops\* pw,

79 **int** w, **int** h, **int** format);

80 **int** (\*set\_crop)(**struct** preview\_stream\_ops \*w,

81 **int** left, **int** top, **int** right, **int** bottom);

82 **int** (\*set\_usage)(**struct** preview\_stream\_ops\* w, **int** usage);

83 **int** (\*set\_swap\_interval)(**struct** preview\_stream\_ops \*w, **int** interval);

84 **int** (\*get\_min\_undequeued\_buffer\_count)(**const** **struct** preview\_stream\_ops \*w,

85 **int** \*count);

86 **int** (\*lock\_buffer)(**struct** preview\_stream\_ops\* w,

87 buffer\_handle\_t\* buffer);

88 **// Timestamps are measured in nanoseconds, and must be comparable**

89 **// and monotonically increasing between two frames in the same**

90 **// preview stream. They do not need to be comparable between**

91 **// consecutive or parallel preview streams, cameras, or app runs.**

92 **int** (\*set\_timestamp)(**struct** preview\_stream\_ops \*w, **int64\_t** timestamp);

93 } preview\_stream\_ops\_t;

### 设备访问

1. 方式

一般使用V4L2方式对设备进行访问。通过V4L2 command从linux kernel中的camera driver得到视频数据。然后交给surface(overlay)显示或者保存为文件。

1. 权限

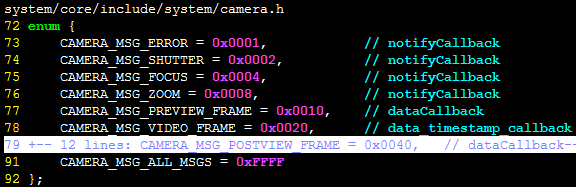
/ueventd.rc中增加权限配置(对应源码目录:android/system/core/rootdir)：

/dev/video\* 0660 system camera

### 消息响应

HAL层在检测到如下消息时，需要向上层返回数据或状态：

1. 定义



1. 作用

|  |  |  |
| --- | --- | --- |
| value | | function |
| 0x001 | CAMERA\_MSG\_ERROR | 错误消息 |
| 0x002 | CAMERA\_MSG\_SHUTTER | 快门消息 |
| 0x004 | CAMERA\_MSG\_FOCUS | 聚焦消息 |
| 0x008 | CAMERA\_MSG\_ZOOM | 缩放消息 |
| 0x010 | CAMERA\_MSG\_PREVIEW\_FRAME | 预览帧消息 |
| 0x020 | CAMERA\_MSG\_VIDEO\_FRAME | 视频帧消息 |
| 0x040 | CAMERA\_MSG\_POSTVIEW\_FRAME | 拍照后停止帧消息 |
| 0x080 | CAMERA\_MSG\_RAW\_IMAGE | 原始数据格式照片光消息 |
| 0x100 | CAMERA\_MSG\_COMPRESSED\_IMAGE | 压缩格式照片消息 |
| 0x1FF | CAMERA\_MSG\_ALL\_MSGS | 所有消息 |

### callback

callback是在上层在调用HAL层时配置的功能接口，用于实现数据和状态在HAL层与client层之间的传输。

1. camera\_notify\_callback

用来传递一些消息: 如快门消息，自动对焦完成消息，出错消息，拍照编码结束等, 参数@msg\_type可以是：

CAMERA\_MSG\_ERROR

CAMERA\_MSG\_SHUTTER

CAMERA\_MSG\_FOCUS

CAMERA\_MSG\_ZOOM

CAMERA\_MSG\_FOCUS\_MOVE

1. camera\_data\_callback

该函数用于从camera HAL返回数据，返回的数据可以是：

preview的数据帧，

preview的元数据，

以及压缩为jpeg格式的帧数据

参数@msg\_type可以是：

CAMERA\_MSG\_PREVIEW\_FRAME

CAMERA\_MSG\_PREVIEW\_METADATA

CAMERA\_MSG\_COMPRESSED\_IMAGE

CAMERA\_MSG\_RAW\_IMAGE

CAMERA\_MSG\_RAW\_IMAGE\_NOTIFY

CAMERA\_MSG\_POSTVIEW\_FRAME

参数@data 用来返回图像数据

参数@metadata 用来返回图像的元数据

1. camera\_data\_timestamp\_callback

该回调函数用来从Camera HAL返回帧数据以及对应的时间戳，这个callback用于录像的场景。

@timestamp：生成帧数据的时间戳

@msg\_type: 取值为CAMERA\_MSG\_VIDEO\_FRAME

@data：用来返回帧数据

## Hal Porting

### 环境建立

1. 源码路径

hardware/ti/omap4xxx/camera/

将该文件夹拷贝一份到device/novatek/nt72668\_common/下。

1. 编译配置
2. 增加module

device/novatek/nt72668\_common/device.mk

26 **# Camera**

27 **PRODUCT\_PACKAGES** += **\**

28 camera.**$(TARGET\_PRODUCT)**

1. 配置module的名称

device/novatek/nt72668\_common/camera/Android.mk

5 **LOCAL\_MODULE** := camera.**$(TARGET\_PRODUCT)**

### ModuleInfo

device/novatek/nt72668\_common/camera/CameraHal\_Module.cpp

71 **static** **struct** hw\_module\_methods\_t camera\_module\_methods = {

72 **open**: camera\_device\_open

73 };

74

75 camera\_module\_t HAL\_MODULE\_INFO\_SYM = {

76 **common**: {

77 **tag**: HARDWARE\_MODULE\_TAG,

78 version\_major: **1**,

79 **version\_minor**: **0**,

80 **id**: CAMERA\_HARDWARE\_MODULE\_ID,

81 **name**: **"CameraHal Module"**,

82 **author**: **"novatek"**,

83 **methods**: &camera\_module\_methods,

84 **dso**: **NULL**, **/\* remove compilation warnings \*/**

85 **reserved**: {**0**}, **/\* remove compilation warnings \*/**

86 },

87 **get\_number\_of\_cameras**: camera\_get\_number\_of\_cameras,

88 **get\_camera\_info**: camera\_get\_camera\_info,

89 };

1. 模块类型标识

tag:表示该模块为HW模块。

1. API版本

版本号，表示为1.0版本。

1. 模块ID

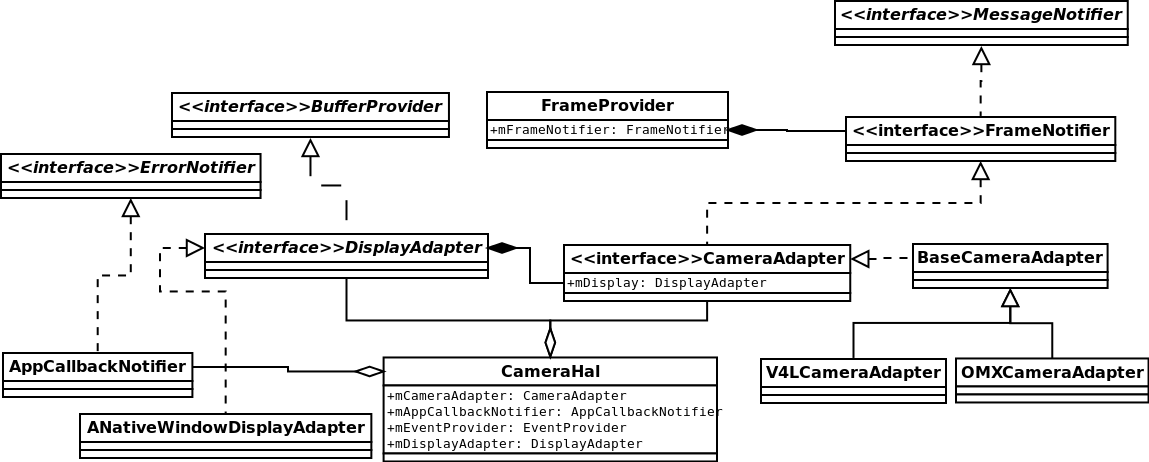
camera模块的ID，表示模块的功能用于实现camera接口。

1. methods

对应版API要求实现的接口。

### 系统结构

该HAL模块的代码已经模块化设计，移植时只需对平台相关的模块进行实现就可以，结构如下：



1. CameraHal

抽象层代码，忽略视频显示和视频捕获的实现细节。

1. BaseCameraAdapter

视频捕获的通用控制逻辑。

1. AnativeWindowDisplayAdapter(平台相关)

具体的视频显示控制逻辑。

1. AppCallbackNotifier

通过注册的callback将hal层的数据和状态反馈给app层。

1. V4LcameraAdapter/ OMXCameraAdapter(平台相关)

具体的视频视频捕获逻辑，本文档中为V4L方式。

1. AppCallbackNotifier(部分平台相关)

用于将数据和状态返回给APP，其中部分功能和平台相关。

### 图像捕获

图像捕获是camera工作的基础，其他的功能都依赖该模块的实现。需要实现的接口如下：

1. CameraAdapter\_Capabilities

|  |  |  |
| --- | --- | --- |
| extern "C" int CameraAdapter\_Capabilities(CameraProperties::Properties\* properties\_array, const unsigned int starting\_camera,  const unsigned int max\_camera) | | |
| 功能 | 获取可用设备的信息。 | |
| 参数 | properties\_array | 设备信息，包括设备的名称，设备路径 |

1. CameraAdapter\_Factory

|  |  |
| --- | --- |
| extern "C" CameraAdapter\* CameraAdapter\_Factory() | |
| 功能 | 获取设备实例，该接口会在HAL层被调用，用于确定视频捕获采用哪种方式。  只能定义一个，如果OMX模块中有需要注释掉，保证系统走的是V4L流程。 |

1. V4LCameraAdapter::initialize

|  |  |  |
| --- | --- | --- |
| status\_t V4LCameraAdapter::initialize(CameraProperties::Properties\* caps) | | |
| 功能 | 初始化设备 | |
| 参数 | caps | 设备信息，包括设备的名称，设备路径 |

1. V4LCameraAdapter::UseBuffersPreview

|  |  |  |
| --- | --- | --- |
| status\_t V4LCameraAdapter::UseBuffersPreview(void\* bufArr, int num) | | |
| 功能 | 在使用图像前，配置图像捕获队列的大小。 | |
| 参数 | bufArr | 该buffer为num大小的数组，表示视频显示的buffer，此处用于和视频捕获的buffer建立关联。 |
| 备注 | 如果使用映射内核内存的方式（V4L2\_MEMORY\_MMAP），构建预览的内存MemoryHeapBase需要从V4L2驱动程序中得到内存指针；  如果使用用户空间内存的方式（V4L2\_MEMORY\_USERPTR），MemoryHeapBase中开辟的内存是在用户空间建立的； | |

1. V4LCameraAdapter::GetFrame

|  |  |  |
| --- | --- | --- |
| char \* V4LCameraAdapter::GetFrame(int &index) | | |
| 功能 | 获取一个准备好的帧 | |
| 参数 | index | 捕获帧buffer index |

1. V4LCameraAdapter::fillThisBuffer

|  |  |  |
| --- | --- | --- |
| status\_t V4LCameraAdapter::fillThisBuffer(void\* frameBuf, CameraFrame::FrameType frameType) | | |
| 功能 | 图像在使用完后，请求对用过的buffer进行重填(捕获图像) | |
| 参数 | frameBuf | 显示模块的buffer值，需要找到对应的视频捕获buffer,然后再进行填充。 |

### 图像显示

图像显示是camera进入后的默认功能，用于将捕获的数据显示到用户指定的区域，需要实现的接口：

1. ANativeWindowDisplayAdapter::allocateBuffer

|  |  |  |
| --- | --- | --- |
| void\* ANativeWindowDisplayAdapter::allocateBuffer(int width, int height, const char\* format, int &bytes, int numBufs) | | |
| 功能 | 在进行预览功能前，通过此接口获取显示源使用的buffer，该buffer将会回传给系统的显示surface作为显示源数据。 | |
| 参数 | width | 宽 |
| height | 高 |
| format | 格式，如“YUV420SP” |
| bytes | buffer大小 |
| numBufs | 显示buffer的个数 |

1. ANativeWindowDisplayAdapter::freeBuffer

|  |  |  |
| --- | --- | --- |
| int ANativeWindowDisplayAdapter::freeBuffer(void\* buf) | | |
| 功能 | 释放显示源buffer | |
| 参数 | buf | 对应的buf |

1. ANativeWindowDisplayAdapter:: PostFrame

|  |  |  |
| --- | --- | --- |
| status\_t ANativeWindowDisplayAdapter::PostFrame(ANativeWindowDisplayAdapter::DisplayFrame &dispFrame) | | |
| 功能 | 将填充好的显示源buffer传给surface，该接口在视频捕获模块填充好一个帧后会被调用。 | |
| 参数 | dispFrame | 帧信息 |

### 数据回传

在数据准备好后，经过framework配置的回调将HAL层的数据和状态返回给APP.

1. AppCallbackNotifier::notifyFrame

|  |  |
| --- | --- |
| void AppCallbackNotifier::notifyFrame() | |
| 功能 | 当收到帧准备好的消息，会调用该接口通知其他模块处理。 |

### 拍照

拍照的实现是在视频捕获后，在AppCallbackNotifier：：notifyFrame将数据进行编码后回传给APP进行处理。

其需要处理的部分为Encoder\_libjpeg的调试，该类中调用标准的jpeglib库的接口实现，将YUV数据编码为jpeg图像。

1. size\_t Encoder\_libjpeg::encode(params\* input)

|  |  |  |
| --- | --- | --- |
| size\_t Encoder\_libjpeg::encode(params\* input) | | |
| 功能 | jpeg编码 | |
| 参数 | input | 配置源数据，以及jpeg编码参数。 |

### 录像

录像的实现是在视频捕获后，在AppCallbackNotifier：：notifyFrame将数据进行编码后回传给APP进行处理。

该部分移植时需要处理的是：

1. AppCallbackNotifier::releaseRecordingFrame

|  |  |  |
| --- | --- | --- |
| status\_t AppCallbackNotifier::releaseRecordingFrame(const void\* mem) | | |
| 功能 | 当Encoder将一帧图像编码完后，调用该接口进行释放 | |
| 参数 | mem | 对应的数据源buffer. |
| 备注 | 需要对每次传给APP的数据进行管理，只有在调用该接口的mem才能进行新的数据填充。 | |

# Linux Camera

## V4L2

### 使用流程

1. 打开设备

打开对应的设备文件（如/dev/video0），并通过ioctrl访问camera driver。

1. 取得设备的capability

查看设备具有什么功能，比如是否具有视频输入,或者音频输入输出等。<VIDIOC\_QUERYCAP,struct v4l2\_capability>。

1. 选择视频输入

一个视频设备可以有多个视频输入。<VIDIOC\_S\_INPUT,struct v4l2\_input>。

1. 设置视频的制式和帧格式

制式包括PAL，NTSC，帧的格式个包括宽度和高度等。<VIDIOC\_S\_STD,VIDIOC\_S\_FMT,struct v4l2\_std\_id,struct v4l2\_format>

1. 向驱动申请帧缓冲

一般不超过5个。<VIDIOC\_REQBUFS，struct v4l2\_requestbuffers>

1. 在预览的线程中，使用VIDIOC\_DQBUF调用阻塞等待视频帧的到来，处理完成后使用VIDIOC\_QBUF调用将帧内存再次压入队列，等待下一帧的到来。
2. 将申请到的帧缓冲映射到用户空间，这样就可以直接操作采集到的帧了，而不必去复制。<mmap>
3. 将申请到的帧缓冲全部入队列，以便存放采集到的数据。<VIDIOC\_QBUF,struct v4l2\_buffer>
4. 开始视频的采集。<VIDIOC\_STREAMON>
5. 出队列以取得已采集数据的帧缓冲，取得原始采集数据。<VIDIOC\_DQBUF>
6. 将缓冲重新入队列尾,这样可以循环采集。<VIDIOC\_QBUF>
7. 停止视频的采集。<VIDIOC\_STREAMOFF>
8. 关闭设备。close(fd);

### 常用命令

1. V4L2命令含义

|  |  |  |  |
| --- | --- | --- | --- |
| **命令类型** | **Output** | | **Capture** |
| VIDIOC\_REQBUFS | 内存分配/释放(如果req.count为0则释放) | | |
| VIDIOC\_QUERYBUF | 将分配的内存转换成对应的物理地址 | | |
| VIDIOC\_QUERYCAP | 查询设备所支持的功能 | | |
| VIDIOC\_ENUM\_FMT | 获取当前驱动支持的视频格式 | | |
| VIDIOC\_S\_FMT | 设置设备的帧格式 | | |
| VIDIOC\_G\_FMT | 读取设备的帧格式 | | |
| VIDIOC\_TRY\_FMT | 验证设备的显示格式 | | |
| VIDIOC\_CROPCAP | 查询设备的修剪能力 | | |
| VIDIOC\_S\_CROP | 设置视频显示的边框 | 设置视频采集的边框 | |
| VIDIOC\_G\_CROP | 读取视频显示的边框 | 读取视频采集的边框 | |
| VIDIOC\_QBUF | 让内核显示该buf内容 | 让内核将数据采集到该buf | |
| VIDIOC\_DQBUF | 从内核取一个空闲buf用于填充数据 | 从内核中取一个已填充数据的buf用于显示 | |
| VIDIOC\_STREAMON | 启动显示 | 启动采集 | |
| VIDIOC\_STREAMOFF | 停止显示 | 停止采集 | |
| VIDIOC\_QUERYSTD | 检查设备支持的视频标准 | | |

### 设备控制

获得 device 的 capability 以后，可以根据应用程序的功能要求对设备参数进行一系列的设置 ，这些参数又分为两部分，一个是 user contrl ，另一个是 extended control 。

1. user control
2. 常见control id

|  |  |  |
| --- | --- | --- |
| ID type | value type | function |
| V4L2\_CID\_BASE |  | 第一个预定义的 ID |
| V4L2\_CID\_USER\_BASE |  | 等同于 V4L2\_CID\_BASE |
| V4L2\_CID\_BRIGHTNESS | integer | 亮度 |
| V4L2\_CID\_AUTO\_WHITE\_BALANCE | boolean | camera 的自动白平衡 |
| V4L2\_CID\_EXPOSURE | integer | camera 的爆光时间 |
| V4L2\_CID\_LASTP1 |  | 最后一个预定义的 ID |
| V4L2\_CID\_PRIVATE\_BASE |  | 驱动定义的 control ID 起始 |

1. 获取control支持项

可以通过 VIDIOC\_QUERYCTRL 和 VIDIOC\_QUERYMENU ioctls 来枚举出有效的 control ID ，及其属性，比如说 ID 值，类型，是否有效，是否可修改，最大值，最小值，步长等等 。

int ioctl(int fd, int request,struct v4l2\_queryctrl \*argp);

int ioctl(int fd, int request, struct v4l2\_querymenu \*argp);

主要的数据结构是 v4l2\_queryctrl 和 v4l2\_querymenu 。

通过 V4L2\_CID\_BASE 和 V4L2\_CID\_LASTP1 可以枚举出所有的预定义 control ID ；

通过 V4L2\_CID\_PRIVATE\_BASE 来枚举出所有的驱动定义的 control ID 。

Menu 实际上是同一个 ID 可能具有多个选项的目录。

1. 更新control

获得 user control ID 以后，可以对其中可以修改的 ID 按照应用程序的要求进行修改 VIDIOC\_G\_CTRL, VIDIOC\_S\_CTRL ：

int ioctl(int fd, int request, struct v4l2\_control \*argp);

1. extended control

除了 user control 之外还有一个就是扩展控制，扩展控制可以同时原子的对多个 ID 进行 control ，

1. 相关命令是三个：

VIDIOC\_G\_EXT\_CTRLS

VIDIOC\_S\_EXT\_CTRLS

VIDIOC\_TRY\_EXT\_CTRLS

1. 更新接口

int ioctl(int fd, int request, struct v4l2\_ext\_controls \*argp);

其中最重要的是 v4l2\_ext\_controls 这个数据结构，配置项如下：

|  |  |
| --- | --- |
| \_\_u32 ctrl\_class | 现在 spec 中只定义了两种类型的 class ： V4L2\_CTRL\_CLASS\_USER 和 V4L2\_CTRL\_CLASS\_MPEG |
| \_\_u32 count | ctrl 数组中的 control ，即 v4l2\_ext\_control 的个数 |
| struct v4l2\_ext\_control \* controls | control 数组， v4l2\_ext\_control 包含要设定的 ID ，以及 value 应用程序可以使用V4L2\_CTRL\_FLAG\_NEXT\_CTRL 来对扩展 control 进行枚举， V4L2\_CTRL\_FLAG\_NEXT\_CTRL 返回下一个 ID 更高的 control ID |

1. 获取所有支持项

struct v4l2\_queryctrl qctrl;

qctrl.id = V4L2\_CTRL\_FLAG\_NEXT\_CTRL;

while (0 == ioctl (fd, VIDIOC\_QUERYCTRL, &qctrl)) {

/\* ... \*/

qctrl.id |= V4L2\_CTRL\_FLAG\_NEXT\_CTRL;

}

1. 获取特定支持项

要枚举指定的 control class 中的 control 可以使用下面的方法：

qctrl.id = V4L2\_CTRL\_CLASS\_MPEG | V4L2\_CTRL\_FLAG\_NEXT\_CTRL;

while (0 == ioctl (fd, VIDIOC\_QUERYCTRL, &qctrl)) {

if (V4L2\_CTRL\_ID2CLASS (qctrl.id) != V4L2\_CTRL\_CLASS\_MPEG)

break;

/\* ... \*/

qctrl.id |= V4L2\_CTRL\_FLAG\_NEXT\_CTRL;

}

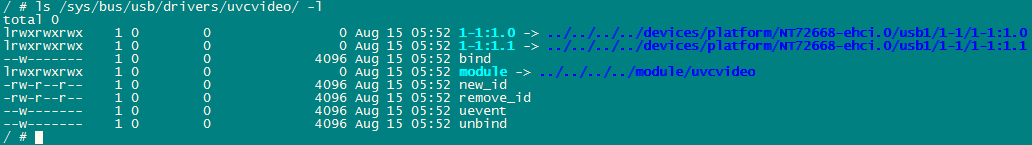
## UVC

usb video class用于实现usb的视频数据传输与控制。

### uvc 驱动

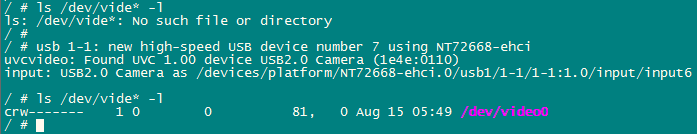
1. 驱动信息





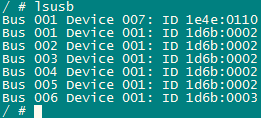
### usb camera

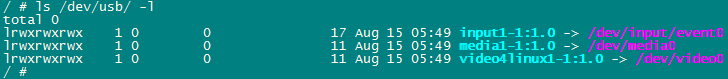
1. 插入USB摄像头

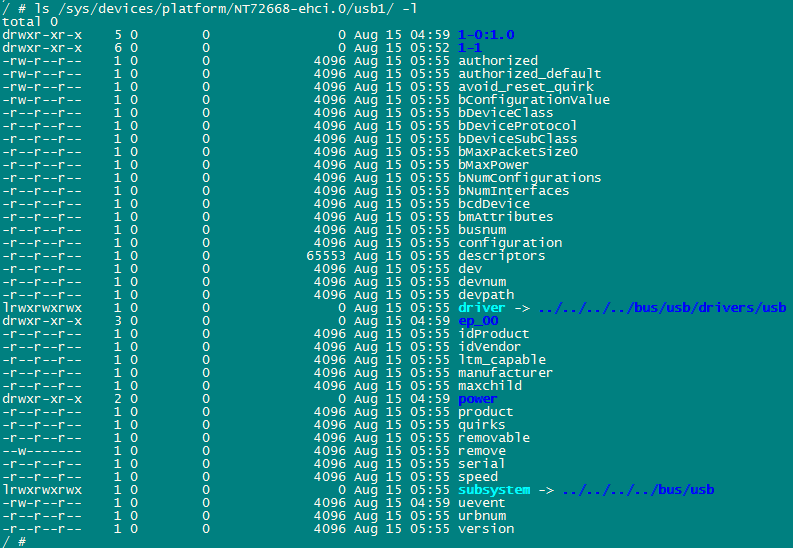


Vendor ID 为1e4e，Product ID 为0110.

1. 查看usb设备







usb camera

