夏天人字拖

Don't Be Afraid To Dream Big

Qualcomm Camera HAL 2.0

我们知道在HAL的Vendor实现当中会动态去load一个名字为camera.$platform$.so的档案，然后去加载Android HAL当中定义的方法，这里以Camera HAL 2.0并且Qualcomm msm8960为例子看下，结合之前的一篇文章(http://guoh.org/lifelog/2013/07/glance-at-camera-hal-2-0/)。

(注：这篇文章已经草稿比较久了，但是一直没有发出来，因为手里的这版代码没有设备可以跑，另外也无法确定代码是否完全正确，至少发现了一些地方都是stub实现，文中可能存在一些错误，如发现不正确的地方欢迎指出，我也会尽量发现错误并修正！)

我们知道在camera2.h当中定义了很多方法，那么在msm8960 HAL就是在如下地方

/path/to/qcam-hal/QCamera/HAL2
这编译出来就是一个camera.$platform$.so，请看它的实现
首先是HAL2/wrapper/QualcommCamera.h|cpp

```
/**
 * The functions need to be provided by the camera HAL.
 *
 * If getNumberOfCameras() returns N, the valid cameraId for getCa
 * and openCameraHardware() is 0 to N-1.
 */

static hw_module_methods_t camera_module_methods = {
    open: camera_device_open,
};

static hw_module_t camera_common  = {
    tag: HARDWARE_MODULE_TAG,
    module_api_version: CAMERA_MODULE_API_VERSION_2_0, // 这样Came
    hal_api_version: HARDWARE_HAL_API_VERSION,
    id: CAMERA_HARDWARE_MODULE_ID,
    name: "Qcamera",
    author:"Qcom",
    methods: &camera_module_methods,
    dso: NULL,
    reserved:  {0},
};

```

```
    common: camera_common,
    get_number_of_cameras: get_number_of_cameras,
    get_camera_info: get_camera_info,
};

camera2_device_ops_t camera_ops = { // 注意这些绑定的函数
    set_request_queue_src_ops:           android::set_request_queu
    notify_request_queue_not_empty:      android::notify_request_q
    set_frame_queue_dst_ops:             android::set_frame_queue_
    get_in_progress_count:               android::get_in_progress_
    flush_captures_in_progress:          android::flush_captures_i
    construct_default_request:           android::construct_defaul

    allocate_stream:                     android::allocate_stream,
    register_stream_buffers:             android::register_stream_
    release_stream:                      android::release_stream,

    allocate_reprocess_stream:           android::allocate_reproce
    allocate_reprocess_stream_from_stream: android::allocate_repro
    release_reprocess_stream:            android::release_reproces

    trigger_action:                      android::trigger_action,
    set_notify_callback:                 android::set_notify_callb
    get_metadata_vendor_tag_ops:         android::get_metadata_ven
    dump:                                android::dump,
};

typedef struct { // 注意这个是Qualcomm自己定义的一个wrap结构
  camera2_device_t hw_dev; // 这里是标准的
  QCameraHardwareInterface *hardware;
  int camera_released;
  int cameraId;
} camera_hardware_t;

/* HAL should return NULL if it fails to open camera hardware. */
extern "C" int  camera_device_open(
  const struct hw_module_t* module, const char* id,
          struct hw_device_t** hw_device)
{
    int rc = -1;
    int mode = 0;
    camera2_device_t *device = NULL;
    if (module && id && hw_device) {
        int cameraId = atoi(id);

        if (!strcmp(module->name, camera_common.name)) {
            camera_hardware_t *camHal =
                (camera_hardware_t *) malloc(sizeof (camera_hardwa
            if (!camHal) {
                *hw_device = NULL;
                ALOGE("%s:  end in no mem", __func__);
                return rc;
            }
            /* we have the camera_hardware obj malloced */
            memset(camHal, 0, sizeof (camera_hardware_t));
            camHal->hardware = new QCameraHardwareInterface(camera
            if (camHal->hardware && camHal->hardware->isCameraRead
                camHal->cameraId = cameraId;
                device = &camHal->hw_dev; // 这里camera2_device_t
                device->common.close = close_camera_device; // 初始
                device->common.version = CAMERA_DEVICE_API_VERSION
                device->ops = &camera_ops;
                device->priv = (void *)camHal;
                rc =  0;
            } else {
                if (camHal->hardware) {
                    delete camHal->hardware;
                    camHal->hardware = NULL;
                }
                free(camHal);
                device = NULL;
            }
        }
    }
    /* pass actual hw_device ptr to framework. This amkes that we
    *hw_device = (hw_device_t*)&device->common; // 这就是kernel或者
    return rc;
}
```

看看allocate stream

```
int allocate_stream(const struct camera2_device *device,
        uint32_t width,
        uint32_t height,
        int      format,
        const camera2_stream_ops_t *stream_ops,
        uint32_t *stream_id,
        uint32_t *format_actual,
        uint32_t *usage,
        uint32_t *max_buffers)
{
    QCameraHardwareInterface *hardware = util_get_Hal_obj(device);
    hardware->allocate_stream(width, height, format, stream_ops,
            stream_id, format_actual, usage, max_buffers);
    return rc;
}
```

这里注意QCameraHardwareInterface在QCameraHWI.h|cpp当中

```
int QCameraHardwareInterface::allocate_stream(
    uint32_t width,
    uint32_t height, int format,
    const camera2_stream_ops_t *stream_ops,
    uint32_t *stream_id,
    uint32_t *format_actual,
    uint32_t *usage,
    uint32_t *max_buffers)
{
    int ret = OK;
    QCameraStream *stream = NULL;
    camera_mode_t myMode = (camera_mode_t)(CAMERA_MODE_2D|CAMERA_NO

    stream = QCameraStream_preview::createInstance(
                        mCameraHandle->camera_handle,
                        mChannelId,
                        width,
                        height,
                        format,
                        mCameraHandle,
                        myMode);

    stream->setPreviewWindow(stream_ops); // 这里，也就是只要通过该方
    *stream_id = stream->getStreamId();
    *max_buffers= stream->getMaxBuffers(); // 从HAL得到的
    *usage = GRALLOC_USAGE_HW_CAMERA_WRITE | CAMERA_GRALLOC_HEAP_ID
```

```
        | CAMERA_GRALLOC_FALLBACK_HEAP_ID;
    /* Set to an arbitrary format SUPPORTED by gralloc */
    *format_actual = HAL_PIXEL_FORMAT_YCrCb_420_SP;

    return ret;
}
```

QCameraStream_preview::createInstance直接调用自己的构造方法，也就是下面

(相关class在QCameraStream.h|cpp和QCameraStream_Preview.cpp)

```
QCameraStream_preview::QCameraStream_preview(uint32_t CameraHandle,
                        uint32_t ChannelId,
                        uint32_t Width,
                        uint32_t Height,
                        int requestedFormat,
                        mm_camera_vtbl_t *mm_ops,
                        camera_mode_t mode) :
                 QCameraStream(CameraHandle,
                        ChannelId,
                        Width,
                        Height,
                        mm_ops,
                        mode),
                 mLastQueuedFrame(NULL),
                 mDisplayBuf(NULL),
                 mNumFDRcvd(0)
{
    mStreamId = allocateStreamId(); // 分配stream id(根据mStreamTabl

    switch (requestedFormat) { // max buffer number
    case CAMERA2_HAL_PIXEL_FORMAT_OPAQUE:
        mMaxBuffers = 5;
        break;
    case HAL_PIXEL_FORMAT_BLOB:
        mMaxBuffers = 1;
        break;
    default:
        ALOGE("Unsupported requested format %d", requestedFormat);
        mMaxBuffers = 1;
        break;
    }
    /*TODO: There has to be a better way to do this*/
}
```

再看看

/path/to/qcam-hal/QCamera/stack/mm-camera-interface/mm_camera_interface.h

当中

```
typedef struct {
    uint32_t camera_handle;        /* camera object handle */
    mm_camera_info_t *camera_info; /* reference pointer of camear in
    mm_camera_ops_t *ops;          /* API call table */
} mm_camera_vtbl_t;
```

mm_camera_interface.c
当中

```
/* camera ops v-table */
static mm_camera_ops_t mm_camera_ops = {
    .sync = mm_camera_intf_sync,
    .is_event_supported = mm_camera_intf_is_event_supported,
    .register_event_notify = mm_camera_intf_register_event_notify,
    .qbuf = mm_camera_intf_qbuf,
    .camera_close = mm_camera_intf_close,
    .query_2nd_sensor_info = mm_camera_intf_query_2nd_sensor_info,
    .is_parm_supported = mm_camera_intf_is_parm_supported,
    .set_parm = mm_camera_intf_set_parm,
    .get_parm = mm_camera_intf_get_parm,
    .ch_acquire = mm_camera_intf_add_channel,
    .ch_release = mm_camera_intf_del_channel,
    .add_stream = mm_camera_intf_add_stream,
    .del_stream = mm_camera_intf_del_stream,
    .config_stream = mm_camera_intf_config_stream,
    .init_stream_bundle = mm_camera_intf_bundle_streams,
    .destroy_stream_bundle = mm_camera_intf_destroy_bundle,
    .start_streams = mm_camera_intf_start_streams,
    .stop_streams = mm_camera_intf_stop_streams,
    .async_teardown_streams = mm_camera_intf_async_teardown_streams
    .request_super_buf = mm_camera_intf_request_super_buf,
    .cancel_super_buf_request = mm_camera_intf_cancel_super_buf_req
    .start_focus = mm_camera_intf_start_focus,
    .abort_focus = mm_camera_intf_abort_focus,
    .prepare_snapshot = mm_camera_intf_prepare_snapshot,
    .set_stream_parm = mm_camera_intf_set_stream_parm,
    .get_stream_parm = mm_camera_intf_get_stream_parm
};
```

以start stream为例子

```
mm_camera_intf_start_streams(mm_camera_interface
    mm_camera_start_streams(mm_camera
        mm_channel_fsm_fn(mm_camera_channel
            mm_channel_fsm_fn_active(mm_camera_channel
                mm_channel_start_streams(mm_camera_channel
                    mm_stream_fsm_fn(mm_camera_stream
                        mm_stream_fsm_reg(mm_camera_stream
                            mm_camera_cmd_thread_launch(mm_camera_da
                            mm_stream_streamon(mm_camera_stream
```

注意：本文当中，如上这种梯度摆放，表示是调用关系，如果梯度是一样的，就表示这些方法是在上层同一个方法里面被调用的

```
int32_t mm_stream_streamon(mm_stream_t *my_obj)
{
    int32_t rc;
    enum v4l2_buf_type buf_type = V4L2_BUF_TYPE_VIDEO_CAPTURE_MPLAN

    /* Add fd to data poll thread */
    rc = mm_camera_poll_thread_add_poll_fd(&my_obj->ch_obj->poll_th
                                           my_obj->my_hdl,
                                           my_obj->fd,
                                                (void*)my_obj);
    if (rc < 0) {
        return rc;
    }
    rc = ioctl(my_obj->fd, VIDIOC_STREAMON, &buf_type);
    if (rc < 0) {
        CDBG_ERROR("%s: ioctl VIDIOC_STREAMON failed: rc=%d\n",
                   __func__, rc);
        /* remove fd from data poll thread in case of failure */
        mm_camera_poll_thread_del_poll_fd(&my_obj->ch_obj->poll_thr
    }
    return rc;
}
```

看到ioctl，VIDIOC_STREAMON，可以高兴一下了，这就是V4L2规范当中用户空间和内核空间通信的方法，V4L2(Video for Linux Two)是一种经典而且成熟的视频通信协议，之前是V4L，不清楚的可以去下载它的规范，另外The Video4Linux2 (http://lwn.net/Articles/203924/)也是很好的资料。
这里简单介绍下：

> *open(VIDEO_DEVICE_NAME, …) // 开启视频设备，一般在程序初始化的时候调用*
>
> *ioctl(…) // 主要是一些需要传输数据量很小的控制操作*
> *这里可以用的参数很多，并且通常来说我们会按照以下方式来使用，比如*
> *VIDIOC_QUERYCAP // 查询设备能干什么*
> *VIDIOC_CROPCAP // 查询设备crop能力*
> *VIDIOC_S_* // set/get方法，设置/获取参数*
> *VIDIOC_G_**
> *VIDIOC_REQBUFS // 分配buffer，可以有多种方式*
> *VIDIOC_QUERYBUF // 查询分配的buffer的信息*
> *VIDIOC_QBUF // QUEUE BUFFER 把buffer压入DRV缓存队列(这时候buffer是空的)*
> *VIDIOC_STREAMON // 开始视频数据传输*
> *VIDIOC_DQBUF // DEQUEUE BUFFER 把buffer从DRV缓存队列中取出(这时候buffer是有数据的)*
>
> *[0…n]*
> *QBUF -> DQBUF // 可以一直重复这个动作*

> *VIDIOC_STREAMOFF // 停止视频数据传输*
>
> *close(VIDEO_DEVICE_FD) // 关闭设备*
> *上面就是主要的函数和简单的调用顺序，另外还有几个函数*
>
> *select() // 等待事件发生，主要用在我们把存frame的buffer推给DRV以后，等待它的反应*
> *mmap/munmap // 主要处理我们request的buffer的，buffer分配在设备的内存空间的时候需要*

并且看看mm_camera_stream这个文件里面也都是这么实现的。

看完这里，我们回过头来继续看QCam HAL，当然它实现的细节也不是我上面start stream所列的那么简单，但是其实也不算复杂，觉得重要的就是状态和用到的结构。

首先是channel状态，目前只支持1个channel，但是可以有多个streams(后面会介绍，而且目前最多支持8个streams)

```
/* mm_channel */
typedef enum {
    MM_CHANNEL_STATE_NOTUSED = 0,   /* not used */
    MM_CHANNEL_STATE_STOPPED,       /* stopped */
    MM_CHANNEL_STATE_ACTIVE,        /* active, at least one stream a
    MM_CHANNEL_STATE_PAUSED,        /* paused */
    MM_CHANNEL_STATE_MAX
} mm_channel_state_type_t;
```

它可以执行的事件

```
typedef enum {
    MM_CHANNEL_EVT_ADD_STREAM,
    MM_CHANNEL_EVT_DEL_STREAM,
    MM_CHANNEL_EVT_START_STREAM,
    MM_CHANNEL_EVT_STOP_STREAM,
    MM_CHANNEL_EVT_TEARDOWN_STREAM,
    MM_CHANNEL_EVT_CONFIG_STREAM,
    MM_CHANNEL_EVT_PAUSE,
    MM_CHANNEL_EVT_RESUME,
    MM_CHANNEL_EVT_INIT_BUNDLE,
    MM_CHANNEL_EVT_DESTROY_BUNDLE,
    MM_CHANNEL_EVT_REQUEST_SUPER_BUF,
    MM_CHANNEL_EVT_CANCEL_REQUEST_SUPER_BUF,
    MM_CHANNEL_EVT_START_FOCUS,
    MM_CHANNEL_EVT_ABORT_FOCUS,
    MM_CHANNEL_EVT_PREPARE_SNAPSHOT,
    MM_CHANNEL_EVT_SET_STREAM_PARM,
    MM_CHANNEL_EVT_GET_STREAM_PARM,
    MM_CHANNEL_EVT_DELETE,
    MM_CHANNEL_EVT_MAX
} mm_channel_evt_type_t;
```

```
/* mm_stream */
typedef enum { // 这里的状态要仔细，每执行一次方法，状态就需要变化
    MM_STREAM_STATE_NOTUSED = 0,      /* not used */
    MM_STREAM_STATE_INITED,           /* inited  */
    MM_STREAM_STATE_ACQUIRED,         /* acquired, fd opened  */
    MM_STREAM_STATE_CFG,              /* fmt & dim configured */
    MM_STREAM_STATE_BUFFED,           /* buf allocated */
    MM_STREAM_STATE_REG,              /* buf regged, stream off */
    MM_STREAM_STATE_ACTIVE_STREAM_ON, /* active with stream on */
    MM_STREAM_STATE_ACTIVE_STREAM_OFF, /* active with stream off */
    MM_STREAM_STATE_MAX
} mm_stream_state_type_t;
```

同样，stream可以执行的事件

```
typedef enum {
    MM_STREAM_EVT_ACQUIRE,
    MM_STREAM_EVT_RELEASE,
    MM_STREAM_EVT_SET_FMT,
    MM_STREAM_EVT_GET_BUF,
    MM_STREAM_EVT_PUT_BUF,
    MM_STREAM_EVT_REG_BUF,
    MM_STREAM_EVT_UNREG_BUF,
    MM_STREAM_EVT_START,
    MM_STREAM_EVT_STOP,
    MM_STREAM_EVT_QBUF,
    MM_STREAM_EVT_SET_PARM,
    MM_STREAM_EVT_GET_PARM,
    MM_STREAM_EVT_MAX
} mm_stream_evt_type_t;
```

这里每次执行函数的时候都需要检查channel/stream的状态，只有状态正确的时候才会去执行

比如你可以观察到

mm_channel的mm_channel_state_type_t state;

mm_stream的mm_stream_state_type_t state;

均表示这个结构当前的状态

另外

struct mm_camera_obj

struct mm_channel

struct mm_stream

这三个也是自上而下包含的，并且stream和channel还会持有父结构(暂且这么称呼，实际为container关系)的引用。

实际上Vendor的HAL每个都有自己实现的方法，也可能包含很多特有的东西，比如这里它会喂给ioctl一些特有的命令或者数据结构，这些我们就只有在做特定平台的时候去考虑了。这些都可能千变万化，比如OMAP4它同DRV沟通是透过rpmsg，并用OpenMAX的一套规范来实现的。

理论就这么多，接着看一个实例，比如我们在Camera Service要去start preview：

```
Camera2Client::startPreviewL
    StreamingProcessor->updatePreviewStream
        Camera2Device->createStream
            StreamAdapter->connectToDevice
                camera2_device_t->ops->allocate_stream // 上面有分析
                native_window_api_*或者native_window_*

    StreamingProcessor->startStream
        Camera2Device->setStreamingRequest
            Camera2Device::RequestQueue->setStreamSlot // 创建一个st
                Camera2Device::RequestQueue->signalConsumerLocked
```

```
status_t Camera2Device::MetadataQueue::signalConsumerLocked() {
    status_t res = OK;
    notEmpty.signal();
    if (mSignalConsumer && mDevice != NULL) {
        mSignalConsumer = false;
        mMutex.unlock();
        res = mDevice->ops->notify_request_queue_not_empty(mDevice)





        mMutex.lock();
    }
    return res;
}
```

然而在Qualcomm HAL当中

```
int notify_request_queue_not_empty(const struct camera2_device *devi
    QCameraHardwareInterface->notify_request_queue_not_empty()
        pthread_create(&mCommandThread, &attr, command_thread, (void
```

```
void *command_thread(void *obj)
{
    ...
    pme->runCommandThread(obj);
}
```

```
void QCameraHardwareInterface::runCommandThread(void *data)
{
    /**
     * This function implements the main service routine for the in
     * frame requests, this thread routine is started everytime we
     * notify_request_queue_not_empty trigger, this thread makes th
     * assumption that once it receives a NULL on a dequest_request
     * there will be a fresh notify_request_queue_not_empty call th
     * invoked thereby launching a new instance of this thread. The
     * once we get a NULL on a dequeue request we simply let this t
     */
    int res;
    camera_metadata_t *request=NULL;
    mPendingRequests=0;

    while (mRequestQueueSrc) { // mRequestQueueSrc是通过set_request_
                               // 参见Camera2Device::MetadataQueue:
                               // 在Camera2Device::initialize当中被
        ALOGV("%s:Dequeue request using mRequestQueueSrc:%p",__func
        mRequestQueueSrc->dequeue_request(mRequestQueueSrc, &reques
        if (request==NULL) {
            ALOGE("%s:No more requests available from src command \
                    thread dying",__func__);
            return;
        }
        mPendingRequests++;

        /* Set the metadata values */

        /* Wait for the SOF for the new metadata values to be appli

        /* Check the streams that need to be active in the stream r
        sort_camera_metadata(request);

        camera_metadata_entry_t streams;
        res = find_camera_metadata_entry(request,
                ANDROID_REQUEST_OUTPUT_STREAMS,
                &streams);
        if (res != NO_ERROR) {
            ALOGE("%s: error reading output stream tag", __FUNCTION
            return;
        }

        res = tryRestartStreams(streams); // 会去prepareStream和stre
        if (res != NO_ERROR) {
            ALOGE("error tryRestartStreams %d", res);
            return;
        }

        /* 3rd pass: Turn on all streams requested */
        for (uint32_t i = 0; i < streams.count; i++) {
            int streamId = streams.data.u8[i];
            QCameraStream *stream = QCameraStream::getStreamAtId(st

            /* Increment the frame pending count in each stream cla

            /* Assuming we will have the stream obj in had at this
             * may be multiple objs in which case we loop through a
            stream->onNewRequest();
        }
        ALOGV("%s:Freeing request using mRequestQueueSrc:%p",__func
        /* Free the request buffer */
        mRequestQueueSrc->free_request(mRequestQueueSrc,request);
        mPendingRequests--;
        ALOGV("%s:Completed request",__func__);
    }

    QCameraStream::streamOffAll();
}
```

下面这个方法解释mRequestQueueSrc来自何处

```
// Connect to camera2 HAL as consumer (input requests/reprocessing)
status_t Camera2Device::MetadataQueue::setConsumerDevice(camera2_de
    ATRACE_CALL();
    status_t res;
    res = d->ops->set_request_queue_src_ops(d,
            this);
    if (res != OK) return res;
    mDevice = d;
    return OK;
}
```

因为

```
QCameraStream_preview->prepareStream
    QCameraStream->initStream
        mm_camera_vtbl_t->ops->add_stream(... stream_cb_routine ...)
            mm_camera_add_stream
                mm_channel_fsm_fn(..., MM_CHANNEL_EVT_ADD_STREAM, ..
                    mm_channel_fsm_fn_stopped
                        mm_channel_add_stream(..., mm_camera_buf_not
                            mm_stream_fsm_inited
```

而

在mm_channel_add_stream当中有把mm_camera_buf_notify_t包装到mm_stream_t

```
mm_stream_t *stream_obj = NULL;
/* initialize stream object */
memset(stream_obj, 0, sizeof(mm_stream_t));
/* cd through intf always palced at idx 0 of buf_cb */
stream_obj->buf_cb[0].cb = buf_cb; // callback
stream_obj->buf_cb[0].user_data = user_data;
stream_obj->buf_cb[0].cb_count = -1; /* infinite by default */ // 默
```

并且mm_stream_fsm_inited，传进来的event参数也是MM_STREAM_EVT_ACQUIRE

```
int32_t mm_stream_fsm_inited(mm_stream_t *my_obj,
                             mm_stream_evt_type_t evt,
                             void * in_val,
                             void * out_val)
{
    int32_t rc = 0;
    char dev_name[MM_CAMERA_DEV_NAME_LEN];

    switch (evt) {
    case MM_STREAM_EVT_ACQUIRE:
        if ((NULL == my_obj->ch_obj) || (NULL == my_obj->ch_obj->ca
            CDBG_ERROR("%s: NULL channel or camera obj\n", __func__
            rc = -1;
            break;
        }
        snprintf(dev_name, sizeof(dev_name), "/dev/%s",
                 mm_camera_util_get_dev_name(my_obj->ch_obj->cam_ob

        my_obj->fd = open(dev_name, O_RDWR | O_NONBLOCK); // 打开视
        if (my_obj->fd <= 0) {
            CDBG_ERROR("%s: open dev returned %d\n", __func__, my_o
            rc = -1;
            break;
        }
        rc = mm_stream_set_ext_mode(my_obj);
        if (0 == rc) {
            my_obj->state = MM_STREAM_STATE_ACQUIRED; // mm_stream_
        } else {
            /* failed setting ext_mode
             * close fd */
            if(my_obj->fd > 0) {
                close(my_obj->fd);
                my_obj->fd = -1;
            }
            break;
        }
        rc = get_stream_inst_handle(my_obj);
        if(rc) {
            if(my_obj->fd > 0) {
                close(my_obj->fd);
                my_obj->fd = -1;
            }
        }
        break;
    default:
        CDBG_ERROR("%s: Invalid evt=%d, stream_state=%d",
                   __func__,evt,my_obj->state);
        rc = -1;
        break;
    }
    return rc;
}
```

还有

```
QCameraStream->streamOn
    mm_camera_vtbl_t->ops->start_streams
        mm_camera_intf_start_streams
            mm_camera_start_streams
                mm_channel_fsm_fn(..., MM_CHANNEL_EVT_START_STREAM,
                    mm_stream_fsm_fn(..., MM_STREAM_EVT_START, ...)
                        mm_camera_cmd_thread_launch // 启动CB线程
                        mm_stream_streamon(mm_stream_t)
                            mm_camera_poll_thread_add_poll_fd(..., m
```

而

```
static void mm_stream_data_notify(void* user_data)
{
    mm_stream_t *my_obj = (mm_stream_t*)user_data;
    int32_t idx = -1, i, rc;
    uint8_t has_cb = 0;
    mm_camera_buf_info_t buf_info;

    if (NULL == my_obj) {
        return;
    }

    if (MM_STREAM_STATE_ACTIVE_STREAM_ON != my_obj->state) {
        /* this Cb will only received in active_stream_on state
         * if not so, return here */
        CDBG_ERROR("%s: ERROR!! Wrong state (%d) to receive data no
                   __func__, my_obj->state);
        return;
    }

    memset(&buf_info, 0, sizeof(mm_camera_buf_info_t));

    pthread_mutex_lock(&my_obj->buf_lock);
    rc = mm_stream_read_msm_frame(my_obj, &buf_info); // 通过ioctl(
    if (rc != 0) {
        pthread_mutex_unlock(&my_obj->buf_lock);
        return;
    }
    idx = buf_info.buf->buf_idx;

    /* update buffer location */
    my_obj->buf_status[idx].in_kernel = 0;

    /* update buf ref count */
    if (my_obj->is_bundled) {
        /* need to add into super buf since bundled, add ref count
        my_obj->buf_status[idx].buf_refcnt++;
    }

    for (i=0; i < MM_CAMERA_STREAM_BUF_CB_MAX; i++) {
        if(NULL != my_obj->buf_cb[i].cb) {
            /* for every CB, add ref count */
            my_obj->buf_status[idx].buf_refcnt++;
            has_cb = 1;
        }
    }
    pthread_mutex_unlock(&my_obj->buf_lock);

    mm_stream_handle_rcvd_buf(my_obj, &buf_info); // mm_camera_queu
                                                  // 前提是有注册cal
                                                  // 然后mm_camera_
                                                  // 轮循读取数据，
}
```

这样就会导致在stream on的时候stream_cb_routine(实现在QCameraStream当中)就会一直执行

```
void stream_cb_routine(mm_camera_super_buf_t *bufs,
                       void *userdata)
{
    QCameraStream *p_obj=(QCameraStream*) userdata;
    switch (p_obj->mExtImgMode) { // 这个mode在prepareStream的时候就
    case MM_CAMERA_PREVIEW:
        ALOGE("%s : callback for MM_CAMERA_PREVIEW", __func__);
        ((QCameraStream_preview *)p_obj)->dataCallback(bufs); // CA
        break;
    case MM_CAMERA_VIDEO:
        ALOGE("%s : callback for MM_CAMERA_VIDEO", __func__);
        ((QCameraStream_preview *)p_obj)->dataCallback(bufs);
        break;
    case MM_CAMERA_SNAPSHOT_MAIN:
        ALOGE("%s : callback for MM_CAMERA_SNAPSHOT_MAIN", __func__
        p_obj->p_mm_ops->ops->qbuf(p_obj->mCameraHandle,
                                   p_obj->mChannelId,
                                   bufs->bufs[0]);
        break;
    case MM_CAMERA_SNAPSHOT_THUMBNAIL:
        break;
    default:
        break;
    }
}
```

```
void QCameraStream::dataCallback(mm_camera_super_buf_t *bufs)
{
    if (mPendingCount != 0) { // 这个dataCallback是一直在都在回来么？
                               // 而且从代码来看设置下去的callback次数
                               // 似乎只能这样才能解释，否则没人触发的
                               // 这里也感知不到
        ALOGD("Got frame request");
        pthread_mutex_lock(&mFrameDeliveredMutex);
        mPendingCount--;
        ALOGD("Completed frame request");
        pthread_cond_signal(&mFrameDeliveredCond);
        pthread_mutex_unlock(&mFrameDeliveredMutex);
        processPreviewFrame(bufs);
    } else {
        p_mm_ops->ops->qbuf(mCameraHandle,
                mChannelId, bufs->bufs[0]); // 如果没有需要数据的情况
    }
}
```

比较好奇的是在手里这版QCam HAL的code当中camera2_frame_queue_dst_ops_t没有被用到

```
int QCameraHardwareInterface::set_frame_queue_dst_ops(
    const camera2_frame_queue_dst_ops_t *frame_dst_ops)
{
    mFrameQueueDst = frame_dst_ops; // 这个现在似乎没有用到嘛
    return OK;
}
```

这样Camera Service的FrameProcessor的Camera2Device->getNextFrame就永远也获取不到数据，不知道是不是我手里的这版代码的问题，而且在最新的Qualcomm Camera HAL代码也不在AOSP树当中了，而是直接以proprietary形式给的so档，这只是题外话。

所以总体来看，这里可能有几个QCameraStream，每个stream负责自己的事情。他们之间也有相互关系，比如有可能新的stream进来会导致其他已经stream-on的stream重新启动。

在Camera HAL 2.0当中我们还有个重点就是re-process stream
简单的说就是把output stream作为input stream再次添加到BufferQueue中，让

其他的consumer来处理，就类似一个chain一样。
目前在ZslProcessor当中有用到。

```
ZslProcessor->updateStream
    Camera2Device->createStream
    Camera2Device->createReprocessStreamFromStream // release的时候是
        new ReprocessStreamAdapter
        ReprocessStreamAdapter->connectToDevice
            camera2_device_t->ops->allocate_reprocess_stream_from_st
```

这里ReprocessStreamAdapter实际就是camera2_stream_in_ops_t，负责管理re-process的stream。

但是这版的代码Qualcomm也似乎没有去实现，所以暂时到此为止，如果后面找到相应的代码，再来看。

所以看完这么多不必觉得惊讶，站在Camera Service的立场，它持有两个MetadataQueue，mRequestQueue和mFrameQueue。
app请求的动作，比如set parameter/start preview/start recording会直接转化为request，放到mRequestQueue，然后去重启preview/recording stream。
比如capture也会转换为request，放到mRequestQueue。
如果有必要，会通过notify_request_queue_not_empty去通知QCam HAL有请求需要处理，然后QCam HAL会启动一个线程(QCameraHardwareInterface::runCommandThread)去做处理。直到所有request处理完毕退出线程。
在这个处理的过程当中会分别调用到每个stream的processPreviewFrame，有必要的话它每个都会调用自己后续的callback。
还有一个实现的细节就是，stream_cb_routine是从start stream就有开始注册在同一个channel上的，而stream_cb_routine间接调用QCameraStream::dataCallback(当然stream_cb_routine有去指定这个callback回来的原因是什么，就好调用对应的dataCallback)，这个callback是一直都在回来，所以每次new request让mPendingCount加1之后，dataCallback回来才会调用processPreviewFrame，否则就直接把buffer再次压回DRV队列当中。

```
void QCameraStream::dataCallback(mm_camera_super_buf_t *bufs)
{
    if (mPendingCount != 0) { // 这个dataCallback是一直在都在回来么？
                               // 而且从代码来看设置下去的callback次娄
                               // 似乎只能这样才能解释，否则没人触发的
                               // 这里也感知不到
        ALOGD("Got frame request");
        pthread_mutex_lock(&mFrameDeliveredMutex);
        mPendingCount--;
        ALOGD("Completed frame request");
        pthread_cond_signal(&mFrameDeliveredCond);
        pthread_mutex_unlock(&mFrameDeliveredMutex);
        processPreviewFrame(bufs);
    } else {
        p_mm_ops->ops->qbuf(mCameraHandle,
                        mChannelId, bufs->bufs[0]); // 如果没有需要数据的情况
        }
}
```

```
void QCameraStream::onNewRequest()
{
    ALOGI("%s:E",__func__);
    pthread_mutex_lock(&mFrameDeliveredMutex);
    ALOGI("Sending Frame request");
    mPendingCount++;
    pthread_cond_wait(&mFrameDeliveredCond, &mFrameDeliveredMutex);
    ALOGV("Got frame");
    pthread_mutex_unlock(&mFrameDeliveredMutex);
    ALOGV("%s:X",__func__);
}
```

processPreviewFrame会调用到创建这个stream的时候关联进来的那个BufferQueue的enqueue_buffer方法，把数据塞到BufferQueue中，然后对应的consumer就会收到了。
比如在Android Camera HAL 2.0当中目前有
camera2/BurstCapture.h
camera2/CallbackProcessor.h
camera2/JpegProcessor.h
camera2/StreamingProcessor.h
camera2/ZslProcessor.h
实现了对应的Consumer::FrameAvailableListener，但是burst-capture现在可以不考虑，因为都还只是stub实现。

ZslProcessor.h和CaptureSequencer.h都有去实现FrameProcessor::FilteredListener的onFrameAvailable(…)
但是我们之前讲过这版QCam HAL没有实现，所以FrameProcessor是无法获取到meta data的。
所以这样来看onFrameAbailable都不会得到通知。(我相信是我手里的这版代码的问题啦)

之前我们说过QCam HAL有部分东西没有实现，所以mFrameQueue就不会有数据，但是它本来应该是DRV回来的元数据会queue到这里面。

另外
CaptureSequencer.h还有去实现onCaptureAvailable，当JpegProcessor处理完了会通知它。

好奇？多个stream(s)不是同时返回的，这样如果CPU处理快慢不同就会有时间差？
还有很好奇DRV是如何处理Video snapshot的，如果buffer是顺序的，就会存在

Video少一个frame，如果不是顺序的，那就是DRV一次返回多个buffer？以前真没有想过这个问题@_@

August 25, 2013 guohai Android, C++, Multimedia Camera

3 thoughts on “Qualcomm Camera HAL 2.0”

wade

August 1, 2014 at 5:29 pm

請問一下，若已經編譯出這個 hal 層，如何才能讓上層 camera APP 使用 UVC camera?
需要設定 setprop 什麼值嗎？

alien75

September 4, 2014 at 10:39 am

楼主，请问你对2.0的研究有没有进一步深入？在你这篇文章中提到的channel和stream，按我的理解是：channel对应具体的硬件设备，所以只有一个；而stream是对物理数据的引用，所以最多可以有8个。这种设计方式在实际应用场景中，可以让不同的应用(最多8个不同进程)使用同一个硬件的数据，而不会产生相互影响。我的理解是否正确，请指教。谢谢！

guohai

September 24, 2014 at 9:45 am

不好意思，各位，现在弄Audio相关的了，很久没有看过Camera，暂时没有办法解答各位的疑惑

Proudly powered by WordPress