Table of Contents

Buffer management in video driver : ex: vivi driver

Step 1: Kernel version , Driver location

VERSION = 3
PATCHLEVEL = 16
SUBLEVEL = 71
EXTRAVERSION =
Kernel Version: **3.16.71**
Driver location: **drivers/media/platform/vivi.c**

Step 2: Initialize queue

```
struct vb2_queue *q;
/* initialize queue */
        q = &dev->vb_vidq;
        q->type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        q->io_modes = VB2_MMAP | VB2_USERPTR | VB2_DMABUF | VB2_READ;
        q->drv_priv = dev;
        q->buf_struct_size = sizeof(struct vivi_buffer);
        q->ops = &vivi_video_qops;
        q->mem_ops = &vb2_vmalloc_memops;
        q->timestamp_flags = V4L2_BUF_FLAG_TIMESTAMP_MONOTONIC;

        ret = vb2_queue_init(q);
```

Step 3: vb2 ops

```
static const struct vb2_ops vivi_video_qops = {
        .queue_setup           = queue_setup,
        .buf_prepare           = buffer_prepare,
        .buf_queue             = buffer_queue,
        .start_streaming       = start_streaming,
        .stop_streaming        = stop_streaming,
        .wait_prepare          = vivi_unlock,
        .wait_finish           = vivi_lock,
};
```

```
static int queue_setup(struct vb2_queue *vq, const struct v4l2_format *fmt,
                       unsigned int *nbuffers, unsigned int *nplanes,
                       unsigned int sizes[], void *alloc_ctxs[])
{
        //should say: number of buffers, number of planes, each buffer size
}

static int buffer_prepare(struct vb2_buffer *vb)
{
        //check and verify the buffer size(returned by videobuffer), with pre calculated buffer size
        if (vb2_plane_size(vb, 0) < size) {
            dprintk(dev, 1, "%s data will not fit into plane (%lu < %lu)\n",
                        __func__, vb2_plane_size(vb, 0), size);
            return -EINVAL;
      }
      vb2_set_plane_payload(&buf->vb, 0, size);
 }

static void buffer_queue(struct vb2_buffer *vb)
{
      struct vivi_dev *dev = vb2_get_drv_priv(vb->vb2_queue);
      struct vivi_buffer *buf = container_of(vb, struct vivi_buffer, vb);
      struct vivi_dmaqueue *vidq = &dev->vidq;
      unsigned long flags = 0;

      dprintk(dev, 1, "%s\n", __func__);

      spin_lock_irqsave(&dev->slock, flags);
      list_add_tail(&buf->list, &vidq->active);
      spin_unlock_irqrestore(&dev->slock, flags);
}

static int start_streaming(struct vb2_queue *vq, unsigned int count)
{
      struct vivi_dev *dev = vb2_get_drv_priv(vq);
      int err;

      dprintk(dev, 1, "%s\n", __func__);
      dev->seq_count = 0;
      err = vivi_start_generating(dev);
      if (err) {
            struct vivi_buffer *buf, *tmp;

            list_for_each_entry_safe(buf, tmp, &dev->vidq.active, list) {
                  list_del(&buf->list);
                  vb2_buffer_done(&buf->vb, VB2_BUF_STATE_QUEUED);
            }
      }
```

```
        return err;
}

/* abort streaming and wait for last buffer */
static void stop_streaming(struct vb2_queue *vq)
{
        struct vivi_dev *dev = vb2_get_drv_priv(vq);
        dprintk(dev, 1, "%s\n", __func__);
        vivi_stop_generating(dev);
}

static void vivi_lock(struct vb2_queue *vq)
{
        struct vivi_dev *dev = vb2_get_drv_priv(vq);
        mutex_lock(&dev->mutex);
}

static void vivi_unlock(struct vb2_queue *vq)
{
        struct vivi_dev *dev = vb2_get_drv_priv(vq);
        mutex_unlock(&dev->mutex);
}
```

Step 4: v4l2 ioctl ops

```
static const struct v4l2_ioctl_ops vivi_ioctl_ops = {

        .vidioc_reqbufs       = vb2_ioctl_reqbufs,
        .vidioc_create_bufs   = vb2_ioctl_create_bufs,
        .vidioc_prepare_buf   = vb2_ioctl_prepare_buf,
        .vidioc_querybuf      = vb2_ioctl_querybuf,
        .vidioc_qbuf          = vb2_ioctl_qbuf,
        .vidioc_dqbuf         = vb2_ioctl_dqbuf,

        .vidioc_streamon      = vb2_ioctl_streamon,
        .vidioc_streamoff     = vb2_ioctl_streamoff,
};
```

file: drivers/media/v4l2-core/videobuf2-core.c

Step 5: vb2_ioctl_reqbufs

```
int vb2_ioctl_reqbufs(struct file *file, void *priv,   struct v4l2_requestbuffers *p)
{
```

```
	struct video_device *vdev = video_devdata(file);
	int res = __verify_memory_type(vdev->queue, p->memory, p->type);

	if (res)
		return res;
	if (vb2_queue_is_busy(vdev, file))
		return -EBUSY;
	res = __reqbufs(vdev->queue, p);
	/* If count == 0, then the owner has released all buffers and he
	   is no longer owner of the queue. Otherwise we have a new owner. */
	if (res == 0)
		vdev->queue->owner = p->count ? file->private_data : NULL;
	return res;
}

EXPORT_SYMBOL_GPL(vb2_ioctl_reqbufs);


static int __reqbufs(struct vb2_queue *q, struct v4l2_requestbuffers *req)
{
	/*
	 * Make sure the requested values and current defaults are sane.
	 */
	num_buffers = min_t(unsigned int, req->count, VIDEO_MAX_FRAME);
	num_buffers = max_t(unsigned int, num_buffers, q->min_buffers_needed);
	memset(q->plane_sizes, 0, sizeof(q->plane_sizes));
	memset(q->alloc_ctx, 0, sizeof(q->alloc_ctx));
	q->memory = req->memory;

	/*
	 * Ask the driver how many buffers and planes per buffer it requires.
	 * Driver also sets the size and allocator context for each plane.
	 */
	ret = call_qop(q, queue_setup, q, NULL, &num_buffers, &num_planes,
			q->plane_sizes, q->alloc_ctx);
	if (ret)
		return ret;

	/* Finally, allocate buffers and video memory */
	allocated_buffers = __vb2_queue_alloc(q, req->memory, num_buffers, num_planes);
	if (allocated_buffers == 0) {
		dprintk(1, "memory allocation failed\n");
		return -ENOMEM;
	}
	 q->num_buffers = allocated_buffers;
	 /*
	 * Return the number of successfully allocated buffers
	 * to the userspace.
	 */
```

```
	req->count = allocated_buffers;
	q->waiting_for_buffers = !V4L2_TYPE_IS_OUTPUT(q->type);

	return 0;
}
```

Step 6: vb2_ioctl_querybuf

```
int vb2_ioctl_querybuf(struct file *file, void *priv, struct v4l2_buffer *p)
{
	struct video_device *vdev = video_devdata(file);

	/* No need to call vb2_queue_is_busy(), anyone can query buffers. */
	return vb2_querybuf(vdev->queue, p);
}
EXPORT_SYMBOL_GPL(vb2_ioctl_querybuf);

int vb2_querybuf(struct vb2_queue *q, struct v4l2_buffer *b)
{
	struct vb2_buffer *vb;
	int ret;

	if (b->type != q->type) {
		dprintk(1, "wrong buffer type\n");
		return -EINVAL;
	}

	if (b->index >= q->num_buffers) {
		dprintk(1, "buffer index out of range\n");
		return -EINVAL;
	}
	vb = q->bufs[b->index];
	ret = __verify_planes_array(vb, b);
	if (!ret)
		__fill_v4l2_buffer(vb, b);
	return ret;
}
EXPORT_SYMBOL(vb2_querybuf);

static void __fill_v4l2_buffer(struct vb2_buffer *vb, struct v4l2_buffer *b)
{
	struct vb2_queue *q = vb->vb2_queue;

	/* Copy back data such as timestamp, flags, etc. */
	memcpy(b, &vb->v4l2_buf, offsetof(struct v4l2_buffer, m));
	b->reserved2 = vb->v4l2_buf.reserved2;
	b->reserved = vb->v4l2_buf.reserved;
```

```
if (V4L2_TYPE_IS_MULTIPLANAR(q->type)) {
        /*
         * Fill in plane-related data if userspace provided an array
         * for it. The caller has already verified memory and size.
         */
        b->length = vb->num_planes;
        memcpy(b->m.planes, vb->v4l2_planes,
                b->length * sizeof(struct v4l2_plane));
} else {
        /*
         * We use length and offset in v4l2_planes array even for
         * single-planar buffers, but userspace does not.
         */
        b->length = vb->v4l2_planes[0].length;
        b->bytesused = vb->v4l2_planes[0].bytesused;
        if (q->memory == V4L2_MEMORY_MMAP)
                b->m.offset = vb->v4l2_planes[0].m.mem_offset;
        else if (q->memory == V4L2_MEMORY_USERPTR)
                b->m.userptr = vb->v4l2_planes[0].m.userptr;
        else if (q->memory == V4L2_MEMORY_DMABUF)
                b->m.fd = vb->v4l2_planes[0].m.fd;
}

/*
 * Clear any buffer state related flags.
 */
b->flags &= ~V4L2_BUFFER_MASK_FLAGS;
b->flags |= q->timestamp_flags & V4L2_BUF_FLAG_TIMESTAMP_MASK;
if ((q->timestamp_flags & V4L2_BUF_FLAG_TIMESTAMP_MASK) !=
    V4L2_BUF_FLAG_TIMESTAMP_COPY) {
        /*
         * For non-COPY timestamps, drop timestamp source bits
         * and obtain the timestamp source from the queue.
         */
        b->flags &= ~V4L2_BUF_FLAG_TSTAMP_SRC_MASK;
        b->flags |= q->timestamp_flags & V4L2_BUF_FLAG_TSTAMP_SRC_MASK;
}

switch (vb->state) {
case VB2_BUF_STATE_QUEUED:
case VB2_BUF_STATE_ACTIVE:
        b->flags |= V4L2_BUF_FLAG_QUEUED;
        break;
case VB2_BUF_STATE_ERROR:
        b->flags |= V4L2_BUF_FLAG_ERROR;
        /* fall through */
case VB2_BUF_STATE_DONE:
        b->flags |= V4L2_BUF_FLAG_DONE;
```

```
            break;
        case VB2_BUF_STATE_PREPARED:
            b->flags |= V4L2_BUF_FLAG_PREPARED;
            break;
        case VB2_BUF_STATE_PREPARING:
        case VB2_BUF_STATE_DEQUEUED:
            /* nothing */
            break;
        }

        if (__buffer_in_use(q, vb))
            b->flags |= V4L2_BUF_FLAG_MAPPED;
}
```

Step 7: vb2_ioctl_qbuf

```
int vb2_ioctl_qbuf(struct file *file, void *priv, struct v4l2_buffer *p)
{
        struct video_device *vdev = video_devdata(file);

        if (vb2_queue_is_busy(vdev, file))
            return -EBUSY;
        return vb2_qbuf(vdev->queue, p);
}
EXPORT_SYMBOL_GPL(vb2_ioctl_qbuf);

int vb2_qbuf(struct vb2_queue *q, struct v4l2_buffer *b)
{
        if (vb2_fileio_is_active(q)) {
            dprintk(1, "file io in progress\n");
            return -EBUSY;
        }

        return vb2_internal_qbuf(q, b);
}
EXPORT_SYMBOL_GPL(vb2_qbuf);


static int vb2_internal_qbuf(struct vb2_queue *q, struct v4l2_buffer *b)
{
        int ret = vb2_queue_or_prepare_buf(q, b, "qbuf");
        struct vb2_buffer *vb;

        if (ret)
            return ret;
```

```
vb = q->bufs[b->index];
switch (vb->state) {
case VB2_BUF_STATE_DEQUEUED:
        ret = __buf_prepare(vb, b);
        if (ret)
                return ret;
        break;
case VB2_BUF_STATE_PREPARED:
        break;
case VB2_BUF_STATE_PREPARING:
        dprintk(1, "buffer still being prepared\n");
        return -EINVAL;
default:
        dprintk(1, "invalid buffer state %d\n", vb->state);
        return -EINVAL;
}

/*
 * Add to the queued buffers list, a buffer will stay on it until
 * dequeued in dqbuf.
 */
list_add_tail(&vb->queued_entry, &q->queued_list);
q->queued_count++;
q->waiting_for_buffers = false;
vb->state = VB2_BUF_STATE_QUEUED;

/*
 * If already streaming, give the buffer to driver for processing.
 * If not, the buffer will be given to driver on next streamon.
 */
if (q->start_streaming_called)
        __enqueue_in_driver(vb);

/* Fill buffer information for the userspace */
__fill_v4l2_buffer(vb, b);

/*
 * If streamon has been called, and we haven't yet called
 * start_streaming() since not enough buffers were queued, and
 * we now have reached the minimum number of queued buffers,
 * then we can finally call start_streaming().
 */
if (q->streaming && !q->start_streaming_called &&
    q->queued_count >= q->min_buffers_needed) {
        ret = vb2_start_streaming(q);
        if (ret)
                return ret;
}
```

```
	dprintk(1, "qbuf of buffer %d succeeded\n", vb->v4l2_buf.index);
	return 0;
}
```

Step 8: vb2_ioctl_dqbuf

```
int vb2_ioctl_dqbuf(struct file *file, void *priv, struct v4l2_buffer *p)
{
	struct video_device *vdev = video_devdata(file);

	if (vb2_queue_is_busy(vdev, file))
		return -EBUSY;
	return vb2_dqbuf(vdev->queue, p, file->f_flags & O_NONBLOCK);
}
EXPORT_SYMBOL_GPL(vb2_ioctl_dqbuf);

int vb2_dqbuf(struct vb2_queue *q, struct v4l2_buffer *b, bool nonblocking)
{
	if (vb2_fileio_is_active(q)) {
		dprintk(1, "file io in progress\n");
		return -EBUSY;
	}
	return vb2_internal_dqbuf(q, b, nonblocking);
}
EXPORT_SYMBOL_GPL(vb2_dqbuf);


static int vb2_internal_dqbuf(struct vb2_queue *q, struct v4l2_buffer *b, bool nonblocking)
{
	struct vb2_buffer *vb = NULL;
	int ret;

	if (b->type != q->type) {
		dprintk(1, "invalid buffer type\n");
		return -EINVAL;
	}
	ret = __vb2_get_done_vb(q, &vb, b, nonblocking);
	if (ret < 0)
		return ret;

	switch (vb->state) {
	case VB2_BUF_STATE_DONE:
		dprintk(3, "returning done buffer\n");
		break;
	case VB2_BUF_STATE_ERROR:
		dprintk(3, "returning done buffer with errors\n");
```

```
                break;
        default:
                dprintk(1, "invalid buffer state\n");
                return -EINVAL;
        }

        call_void_vb_qop(vb, buf_finish, vb);

         /* Fill buffer information for the userspace */
        __fill_v4l2_buffer(vb, b);
        /* Remove from videobuf queue */
        list_del(&vb->queued_entry);
        q->queued_count--;
        /* go back to dequeued state */
        __vb2_dqbuf(vb);

        dprintk(1, "dqbuf of buffer %d, with state %d\n",
                        vb->v4l2_buf.index, vb->state);

        /*
         * After calling the VIDIOC_DQBUF V4L2_BUF_FLAG_DONE must be
         * cleared.
         */
        b->flags &= ~V4L2_BUF_FLAG_DONE;
        return 0;
}



/**
 * __vb2_dqbuf() - bring back the buffer to the DEQUEUED state
 */
static void __vb2_dqbuf(struct vb2_buffer *vb)
{
        struct vb2_queue *q = vb->vb2_queue;
        unsigned int i;

        /* nothing to do if the buffer is already dequeued */
        if (vb->state == VB2_BUF_STATE_DEQUEUED)
                return;

        vb->state = VB2_BUF_STATE_DEQUEUED;

        /* unmap DMABUF buffer */
        if (q->memory == V4L2_MEMORY_DMABUF)
                for (i = 0; i < vb->num_planes; ++i) {
                        if (!vb->planes[i].dbuf_mapped)
                                continue;
```

```
                call_void_memop(vb, unmap_dmabuf, vb->planes[i].mem_priv);
                vb->planes[i].dbuf_mapped = 0;
        }
}
```

Step 9: vb2_ioctl_streamon

```
int vb2_ioctl_streamon(struct file *file, void *priv, enum v4l2_buf_type i)
{
        struct video_device *vdev = video_devdata(file);

        if (vb2_queue_is_busy(vdev, file))
                return -EBUSY;
        return vb2_streamon(vdev->queue, i);
}
EXPORT_SYMBOL_GPL(vb2_ioctl_streamon);

int vb2_streamon(struct vb2_queue *q, enum v4l2_buf_type type)
{
        if (vb2_fileio_is_active(q)) {
                dprintk(1, "file io in progress\n");
                return -EBUSY;
        }
        return vb2_internal_streamon(q, type);
}
EXPORT_SYMBOL_GPL(vb2_streamon);

static int vb2_internal_streamon(struct vb2_queue *q, enum v4l2_buf_type type)
{
        int ret;

        if (type != q->type) {
                dprintk(1, "invalid stream type\n");
                return -EINVAL;
        }

        if (q->streaming) {
                dprintk(3, "already streaming\n");
                return 0;
        }

        if (!q->num_buffers) {
                dprintk(1, "no buffers have been allocated\n");
                return -EINVAL;
        }
```

```
	if (q->num_buffers < q->min_buffers_needed) {
		dprintk(1, "need at least %u allocated buffers\n",
				q->min_buffers_needed);
		return -EINVAL;
	}
	/*
	 * Tell driver to start streaming provided sufficient buffers
	 * are available.
	 */
	if (q->queued_count >= q->min_buffers_needed) {
		ret = vb2_start_streaming(q);
		if (ret)
			return ret;
	}

	q->streaming = 1;

	dprintk(3, "successful\n");
	return 0;
}

static int vb2_start_streaming(struct vb2_queue *q)
{
	struct vb2_buffer *vb;
	int ret;

	/*
	 * If any buffers were queued before streamon,
	 * we can now pass them to driver for processing.
	 */
	list_for_each_entry(vb, &q->queued_list, queued_entry)
		__enqueue_in_driver(vb);

	/* Tell the driver to start streaming */
	q->start_streaming_called = 1;
	ret = call_qop(q, start_streaming, q,
			atomic_read(&q->owned_by_drv_count));
	if (!ret)
		return 0;
}

static void __enqueue_in_driver(struct vb2_buffer *vb)
{
	struct vb2_queue *q = vb->vb2_queue;
	unsigned int plane;

	vb->state = VB2_BUF_STATE_ACTIVE;
	atomic_inc(&q->owned_by_drv_count);
```

```
	/* sync buffers */
	for (plane = 0; plane < vb->num_planes; ++plane)
		call_void_memop(vb, prepare, vb->planes[plane].mem_priv);

	call_void_vb_qop(vb, buf_queue, vb);
}
```

Step 9: vb2_ioctl_streamoff

```
int vb2_ioctl_streamoff(struct file *file, void *priv, enum v4l2_buf_type i)
{
	struct video_device *vdev = video_devdata(file);

	if (vb2_queue_is_busy(vdev, file))
		return -EBUSY;
	return vb2_streamoff(vdev->queue, i);
}
EXPORT_SYMBOL_GPL(vb2_ioctl_streamoff);

int vb2_streamoff(struct vb2_queue *q, enum v4l2_buf_type type)
{
	if (vb2_fileio_is_active(q)) {
		dprintk(1, "file io in progress\n");
		return -EBUSY;
	}
	return vb2_internal_streamoff(q, type);
}
EXPORT_SYMBOL_GPL(vb2_streamoff);

static int vb2_internal_streamoff(struct vb2_queue *q, enum v4l2_buf_type type)
{
	if (type != q->type) {
		dprintk(1, "invalid stream type\n");
		return -EINVAL;
	}

	/*
	 * Cancel will pause streaming and remove all buffers from the driver
	 * and videobuf, effectively returning control over them to userspace.
	 *
	 * Note that we do this even if q->streaming == 0: if you prepare or
	 * queue buffers, and then call streamoff without ever having called
	 * streamon, you would still expect those buffers to be returned to
	 * their normal dequeued state.
	 */
	__vb2_queue_cancel(q);
```

```
	q->waiting_for_buffers = !V4L2_TYPE_IS_OUTPUT(q->type);

	dprintk(3, "successful\n");
	return 0;
}

static void __vb2_queue_cancel(struct vb2_queue *q)
{
	unsigned int i;

	/*
	 * Tell driver to stop all transactions and release all queued
	 * buffers.
	 */
	if (q->start_streaming_called)
		call_void_qop(q, stop_streaming, q);

	if (WARN_ON(atomic_read(&q->owned_by_drv_count))) {
		for (i = 0; i < q->num_buffers; ++i)
			if (q->bufs[i]->state == VB2_BUF_STATE_ACTIVE)
				vb2_buffer_done(q->bufs[i], VB2_BUF_STATE_ERROR);
		/* Must be zero now */
		WARN_ON(atomic_read(&q->owned_by_drv_count));
	}

	q->streaming = 0;
	q->start_streaming_called = 0;
	q->queued_count = 0;

/*
	 * Remove all buffers from videobuf's list...
	 */
	INIT_LIST_HEAD(&q->queued_list);
	/*
	 * ...and done list; userspace will not receive any buffers it
	 * has not already dequeued before initiating cancel.
	 */
	INIT_LIST_HEAD(&q->done_list);
	atomic_set(&q->owned_by_drv_count, 0);
	wake_up_all(&q->done_wq);

	/*
	 * Reinitialize all buffers for next use.
	 * Make sure to call buf_finish for any queued buffers. Normally
	 * that's done in dqbuf, but that's not going to happen when we
	 * cancel the whole queue. Note: this code belongs here, not in
	 * __vb2_dqbuf() since in vb2_internal_dqbuf() there is a critical
	 * call to __fill_v4l2_buffer() after buf_finish(). That order can't
	 * be changed, so we can't move the buf_finish() to __vb2_dqbuf().
```

```
         */
        for (i = 0; i < q->num_buffers; ++i) {
                struct vb2_buffer *vb = q->bufs[i];

                if (vb->state != VB2_BUF_STATE_DEQUEUED) {
                        vb->state = VB2_BUF_STATE_PREPARED;
                        call_void_vb_qop(vb, buf_finish, vb);
                }
                __vb2_dqbuf(vb);
        }
}
static void __vb2_dqbuf(struct vb2_buffer *vb)
{
        struct vb2_queue *q = vb->vb2_queue;
        unsigned int i;

        /* nothing to do if the buffer is already dequeued */
        if (vb->state == VB2_BUF_STATE_DEQUEUED)
                return;

        vb->state = VB2_BUF_STATE_DEQUEUED;

        /* unmap DMABUF buffer */
        if (q->memory == V4L2_MEMORY_DMABUF)
                for (i = 0; i < vb->num_planes; ++i) {
                        if (!vb->planes[i].dbuf_mapped)
                                continue;
                        call_void_memop(vb, unmap_dmabuf, vb->planes[i].mem_priv);
                        vb->planes[i].dbuf_mapped = 0;
                }
}
```

Step 10: Call back Table

S. No	struct v4l2_ioctl_ops	struct vb2_ops *ops;	struct vb2_mem_ops *mem_ops;	Buffer state/flag
1	.vidioc_reqbufs , vb2_ioctl_reqbufs, __reqbufs	queue_setup,	__vb2_queue_alloc	
2	.vidioc_querybuf, vb2_ioctl_querybuf,vb2_querybuf, __fill_v4l2_buffer			VB2_BUF_STATE_PREPARED
3	.vidioc_qbuf, vb2_ioctl_qbuf,vb2_qbuf, vb2_internal_qbuf,			VB2_BUF_STATE_QUEUED, V4L2_BUF_FLAG_

	__fill_v4l2_buffer			QUEUED
4	.vidioc_dqbuf, vb2_ioctl_dqbuf, vb2_dqbuf, vb2_internal_dqbuf,__vb2_get_done_vb, __vb2_wait_for_done_vb, __vb2_dqbuf	buf_finish, wait_prepare, wait_finish,		VB2_BUF_STATE_DONE, V4L2_BUF_FLAG_DONE, VB2_BUF_STATE_DEQUEUED
5	. vidioc_streamon,vb2_ioctl_streamon,vb2_streamon, vb2_internal_streamon,vb2_start_streaming, __enqueue_in_driver	buf_queue, start_streaming,		VB2_BUF_STATE_ACTIVE
6	vidioc_streamoff, vb2_ioctl_streamoff, vb2_streamoff, vb2_internal_streamoff, __vb2_queue_cancel, __vb2_dqbuf	stop_streaming, buf_finish,		VB2_BUF_STATE_DEQUEUED, VB2_BUF_STATE_PREPARED, VB2_BUF_STATE_DEQUEUED