An introduction to the videobuf layer

Jonathan Corbet <corbet@lwn.net>

Current as of 2.6.33

The videobuf layer functions as a sort of glue layer between a V4L2 driver and user space. It handles the allocation and management of buffers for the storage of video frames. There is a set of functions which can be used to implement many of the standard POSIX I/O system calls, including read(), poll(), and, happily, mmap(). Another set of functions can be used to implement the bulk of the V4L2 ioctl() calls related to streaming I/O, including buffer allocation, queueing and dequeueing, and streaming control. Using videobuf imposes a few design decisions on the driver author, but the payback comes in the form of reduced code in the driver and a consistent implementation of the V4L2 user-space API.

Buffer types

Not all video devices use the same kind of buffers. In fact, there are (at least) three common variations:

- Buffers which are scattered in both the physical and (kernel) virtual address spaces. (Almost) all user-space buffers are like this, but it makes great sense to allocate kernel-space buffers this way as well when it is possible. Unfortunately, it is not always possible; working with this kind of buffer normally requires hardware which can do scatter/gather DMA operations.

- Buffers which are physically scattered, but which are virtually contiguous; buffers allocated with vmalloc(), in other words. These buffers are just as hard to use for DMA operations, but they can be useful in situations where DMA is not available but virtually-contiguous buffers are convenient.

- Buffers which are physically contiguous. Allocation of this kind of buffer can be unreliable on fragmented systems, but simpler DMA controllers cannot deal with anything else.

Videobuf can work with all three types of buffers, but the driver author must pick one at the outset and design the driver around that decision.

[It's worth noting that there's a fourth kind of buffer: "overlay" buffers which are located within the system's video memory. The overlay functionality is considered to be deprecated for most use, but it still shows up occasionally in system-on-chip drivers where the performance benefits merit the use of this technique. Overlay buffers can be handled as a form of scattered buffer, but there are very few implementations in the kernel and a description of this technique is currently beyond the scope of this document.]

Data structures, callbacks, and initialization

Depending on which type of buffers are being used, the driver should include one of the following files:

```
    <media/videobuf-dma-sg.h>           /* Physically scattered */
    <media/videobuf-vmalloc.h>          /* vmalloc() buffers    */
```

```
    <media/videobuf-dma-contig.h>       /* Physically contiguous */
```

The driver's data structure describing a V4L2 device should include a struct videobuf_queue instance for the management of the buffer queue, along with a list_head for the queue of available buffers. There will also need to be an interrupt-safe spinlock which is used to protect (at least) the queue.

The next step is to write four simple callbacks to help videobuf deal with the management of buffers:

```
    struct videobuf_queue_ops {
        int (*buf_setup)(struct videobuf_queue *q,
                         unsigned int *count, unsigned int *size);
        int (*buf_prepare)(struct videobuf_queue *q,
                           struct videobuf_buffer *vb,
                           enum v4l2_field field);
        void (*buf_queue)(struct videobuf_queue *q,
                          struct videobuf_buffer *vb);
        void (*buf_release)(struct videobuf_queue *q,
                            struct videobuf_buffer *vb);
    };
```

buf_setup() is called early in the I/O process, when streaming is being initiated; its purpose is to tell videobuf about the I/O stream. The count parameter will be a suggested number of buffers to use; the driver should check it for rationality and adjust it if need be. As a practical rule, a minimum of two buffers are needed for proper streaming, and there is usually a maximum (which cannot exceed 32) which makes sense for each device. The size parameter should be set to the expected (maximum) size for each frame of data.

Each buffer (in the form of a struct videobuf_buffer pointer) will be passed to buf_prepare(), which should set the buffer's size, width, height, and field fields properly. If the buffer's state field is VIDEOBUF_NEEDS_INIT, the driver should pass it to:

```
    int videobuf_iolock(struct videobuf_queue* q, struct videobuf_buffer *vb,
                        struct v4l2_framebuffer *fbuf);
```

Among other things, this call will usually allocate memory for the buffer. Finally, the buf_prepare() function should set the buffer's state to VIDEOBUF_PREPARED.

When a buffer is queued for I/O, it is passed to buf_queue(), which should put it onto the driver's list of available buffers and set its state to VIDEOBUF_QUEUED. Note that this function is called with the queue spinlock held; if it tries to acquire it as well things will come to a screeching halt. Yes, this is the voice of experience. Note also that videobuf may wait on the first buffer in the queue; placing other buffers in front of it could again gum up the works. So use list_add_tail() to enqueue buffers.

Finally, buf_release() is called when a buffer is no longer intended to be used. The driver should ensure that there is no I/O active on the buffer, then pass it to the appropriate free routine(s):

```
    /* Scatter/gather drivers */
    int videobuf_dma_unmap(struct videobuf_queue *q,
                           struct videobuf_dmabuf *dma);
    int videobuf_dma_free(struct videobuf_dmabuf *dma);

    /* vmalloc drivers */
    void videobuf_vmalloc_free (struct videobuf_buffer *buf);

    /* Contiguous drivers */
    void videobuf_dma_contig_free(struct videobuf_queue *q,
                                  struct videobuf_buffer *buf);
```

One way to ensure that a buffer is no longer under I/O is to pass it to:

```
    int videobuf_waiton(struct videobuf_buffer *vb, int non_blocking, int intr);
```

Here, vb is the buffer, non_blocking indicates whether non-blocking I/O should be used (it should be zero in the buf_release() case), and intr controls whether an interruptible wait is used.

File operations

At this point, much of the work is done; much of the rest is slipping videobuf calls into the implementation of the other driver callbacks. The first step is in the open() function, which must initialize the videobuf queue. The function to use depends on the type of buffer used:

```
    void videobuf_queue_sg_init(struct videobuf_queue *q,
                                struct videobuf_queue_ops *ops,
                                struct device *dev,
                                spinlock_t *irqlock,
                                enum v4l2_buf_type type,
                                enum v4l2_field field,
                                unsigned int msize,
                                void *priv);

    void videobuf_queue_vmalloc_init(struct videobuf_queue *q,
                                struct videobuf_queue_ops *ops,
                                struct device *dev,
                                spinlock_t *irqlock,
                                enum v4l2_buf_type type,
                                enum v4l2_field field,
                                unsigned int msize,
                                void *priv);

    void videobuf_queue_dma_contig_init(struct videobuf_queue *q,
                                       struct videobuf_queue_ops *ops,
                                       struct device *dev,
                                       spinlock_t *irqlock,
                                       enum v4l2_buf_type type,
                                       enum v4l2_field field,
                                       unsigned int msize,
                                       void *priv);
```

In each case, the parameters are the same: q is the queue structure for the device, ops is the set of callbacks as described above, dev is the device structure for this video device, irqlock is an interrupt-safe spinlock to protect access to the data structures, type is the buffer type used by the device (cameras will use V4L2_BUF_TYPE_VIDEO_CAPTURE, for example), field describes which field is being captured (often V4L2_FIELD_NONE for progressive devices), msize is the size of any containing structure used around struct videobuf_buffer, and priv is a private data pointer which shows up in the priv_data field of struct videobuf_queue. Note that these are void functions which, evidently, are immune to failure.

V4L2 capture drivers can be written to support either of two APIs: the read() system call and the rather more complicated streaming mechanism. As a general rule, it is necessary to support both to ensure that all applications have a chance of working with the device. Videobuf makes it easy to do that with the same code. To implement read(), the driver need only make a call to one of:

```
    ssize_t videobuf_read_one(struct videobuf_queue *q,
                              char __user *data, size_t count,
                              loff_t *ppos, int nonblocking);

    ssize_t videobuf_read_stream(struct videobuf_queue *q,
                                 char __user *data, size_t count,
                                 loff_t *ppos, int vbihack, int nonblocking);
```

Either one of these functions will read frame data into data, returning the amount actually read; the difference is that videobuf_read_one() will only read a single frame, while videobuf_read_stream() will read multiple frames if they are needed to satisfy the count requested by the application. A typical driver read() implementation will start the capture engine, call one of the above functions, then stop the engine before returning (though a smarter implementation might leave the engine running for a little while in anticipation of another read() call happening in the near future).

The poll() function can usually be implemented with a direct call to:

```
    unsigned int videobuf_poll_stream(struct file *file,
                                      struct videobuf_queue *q,
                                      poll_table *wait);
```

Note that the actual wait queue eventually used will be the one associated with the first available buffer.

When streaming I/O is done to kernel-space buffers, the driver must support the mmap() system call to enable user space to access the data. In many V4L2 drivers, the often-complex mmap() implementation simplifies to a single call to:

```
    int videobuf_mmap_mapper(struct videobuf_queue *q,
                             struct vm_area_struct *vma);
```

Everything else is handled by the videobuf code.

The release() function requires two separate videobuf calls:

```
    void videobuf_stop(struct videobuf_queue *q);
    int videobuf_mmap_free(struct videobuf_queue *q);
```

The call to videobuf_stop() terminates any I/O in progress - though it is still up to the driver to stop the capture engine. The call to videobuf_mmap_free() will ensure that all buffers have been unmapped; if so, they will all be passed to the buf_release() callback. If buffers remain mapped, videobuf_mmap_free() returns an error code instead. The purpose is clearly to cause the closing of the file descriptor to fail if buffers are still mapped, but every driver in the 2.6.32 kernel cheerfully ignores its return value.

ioctl() operations

The V4L2 API includes a very long list of driver callbacks to respond to the many ioctl() commands made available to user space. A number of these - those associated with streaming I/O - turn almost directly into videobuf calls. The relevant helper functions are:

```
    int videobuf_reqbufs(struct videobuf_queue *q,
                         struct v4l2_requestbuffers *req);
    int videobuf_querybuf(struct videobuf_queue *q, struct v4l2_buffer *b);
    int videobuf_qbuf(struct videobuf_queue *q, struct v4l2_buffer *b);
    int videobuf_dqbuf(struct videobuf_queue *q, struct v4l2_buffer *b,
                       int nonblocking);
    int videobuf_streamon(struct videobuf_queue *q);
    int videobuf_streamoff(struct videobuf_queue *q);
    int videobuf_cgmbuf(struct videobuf_queue *q, struct video_mbuf *mbuf,
                        int count);
```

So, for example, a VIDIOC_REQBUFS call turns into a call to the driver's vidioc_reqbufs() callback which, in turn, usually only needs to locate the proper struct videobuf_queue pointer and pass it to videobuf_reqbufs(). These support functions can replace a great deal of buffer management boilerplate in a lot of V4L2 drivers.

The vidioc_streamon() and vidioc_streamoff() functions will be a bit more complex, of course, since they will also need to deal with starting and stopping the capture engine. videobuf_cgmbuf(), called from the driver's vidiocgmbuf() function, only exists if the V4L1 compatibility module has been selected with CONFIG_VIDEO_V4L1_COMPAT, so its use must be surrounded with #ifdef directives.

Buffer allocation

Thus far, we have talked about buffers, but have not looked at how they are allocated. The scatter/gather case is the most complex on this front. For allocation, the driver can leave buffer allocation entirely up to the videobuf layer; in this case, buffers will be allocated as anonymous user-space pages and will be very scattered indeed. If the application is using user-space buffers, no allocation is needed; the videobuf layer will take care of calling get_user_pages() and filling in the scatterlist array.

If the driver needs to do its own memory allocation, it should be done in the vidioc_reqbufs() function, *after* calling videobuf_reqbufs(). The first step is a call to:

```
    struct videobuf_dmabuf *videobuf_to_dma(struct videobuf_buffer *buf);
```

The returned videobuf_dmabuf structure (defined in
<media/videobuf-dma-sg.h>) includes a couple of relevant fields:

```
    struct scatterlist  *sglist;
    int                 sglen;
```

The driver must allocate an appropriately-sized scatterlist array and populate it with pointers to the pieces of the allocated buffer; sglen should be set to the length of the array.

Drivers using the vmalloc() method need not (and cannot) concern themselves with buffer allocation at all; videobuf will handle those details. The same is normally true of contiguous-DMA drivers as well; videobuf will allocate the buffers (with dma_alloc_coherent()) when it sees fit. That means that these drivers may be trying to do high-order allocations at any time, an operation which is not always guaranteed to work. Some drivers play tricks by allocating DMA space at system boot time; videobuf does not currently play well with those drivers.

As of 2.6.31, contiguous-DMA drivers can work with a user-supplied buffer, as long as that buffer is physically contiguous. Normal user-space allocations will not meet that criterion, but buffers obtained from other kernel drivers, or those contained within huge pages, will work with these drivers.

Filling the buffers

The final part of a videobuf implementation has no direct callback - it's the portion of the code which actually puts frame data into the buffers, usually in response to interrupts from the device. For all types of drivers, this process works approximately as follows:

 - Obtain the next available buffer and make sure that somebody is actually waiting for it.

 - Get a pointer to the memory and put video data there.

 - Mark the buffer as done and wake up the process waiting for it.

Step (1) above is done by looking at the driver-managed list_head structure - the one which is filled in the buf_queue() callback. Because starting the engine and enqueueing buffers are done in separate steps, it's possible for the engine to be running without any buffers available - in the vmalloc() case especially. So the driver should be prepared for the list to be empty. It is equally possible that nobody is yet interested in the buffer; the driver should not remove it from the list or fill it until a process is waiting on it. That test can be done by examining the buffer's done field (a wait_queue_head_t structure) with waitqueue_active().

A buffer's state should be set to VIDEOBUF_ACTIVE before being mapped for DMA; that ensures that the videobuf layer will not try to do anything with it while the device is transferring data.

For scatter/gather drivers, the needed memory pointers will be found in the scatterlist structure described above. Drivers using the vmalloc() method can get a memory pointer with:

```
    void *videobuf_to_vmalloc(struct videobuf_buffer *buf);
```

For contiguous DMA drivers, the function to use is:

```
    dma_addr_t videobuf_to_dma_contig(struct videobuf_buffer *buf);
```

The contiguous DMA API goes out of its way to hide the kernel-space address of the DMA buffer from drivers.

The final step is to set the size field of the relevant videobuf_buffer structure to the actual size of the captured image, set state to VIDEOBUF_DONE, then call wake_up() on the done queue. At this point, the buffer is owned by the videobuf layer and the driver should not touch it again.

Developers who are interested in more information can go into the relevant header files; there are a few low-level functions declared there which have not been talked about here. Also worthwhile is the vivi driver (drivers/media/video/vivi.c), which is maintained as an example of how V4L2 drivers should be written. Vivi only uses the vmalloc() API, but it's good enough to get started with. Note also that all of these calls are exported GPL-only, so they will not be available to non-GPL kernel modules.