Future of the videobuf framework

Pawel Osciak
p.osciak@samsung.com

Samsung Electronics

Video for Linux 2 summit
Helsinki, June 2010

Videobuf advantages

1. **Queue management and V4L2 API helpers**
   - to make it easier for drivers to implement V4L2 API, assure compliance...
   - to ease high-level buffer management, prevent code duplication, bugs...
   - get for free: streaming and read/write support…

2. **Video memory management, standard solutions for most typical situations**
   - physically contiguous memory
   - scatter-gather
   - contiguous in virtual memory

- **But... it is not as widely used as one would expect...**

Videobuf problems

- **Laurent already mentioned many of them**

- **V4L2 violations(!)**
- **Not enough flexibility, all or nothing approach for memory handling code**
- **Not ready for new, emerging requirements**
  - non-coherent cache architectures
  - different memory allocation strategies
  - IOMMU
  - ...

- **Difficult to maintain; drivers use it in obscure ways**
  - introducing changes to videobuf requires full knowledge of all drivers
- **Code duplication, obscure code, bad practices, inconsistencies**
- **Very little in-code documentation**
- **dma-sg is scary**

It is high time to do something

- **More and more new drivers coming out with their own code for common tasks**
- **We are losing**
  - time – reinventing the wheel in each driver
  - the opportunity to have less code to maintain
  - the advantage of having drivers that are smaller and easier to understand for others
  - the benefit of having our code maintained by others
- **Developers are frustrated that they cannot use videobuf, even if they would like to**

Why not just refactor videobuf?

- A major rewrite of drivers required
- Moving allocation to REQBUFS is a huge change,
  changing free/cancel, streamoff/streamon behaviors is no small task either
- Videobuf is V4L1 compatible (or at least DMA-SG claims to be)
- Too much deadweight; maintaining compatibility would be very difficult
- It may sound like a (too) easy way out, but…

What should stay

- **Videobuf queue – overall, generic concepts, frame management**
- **V4L2 ioctl and file operations handling support**
- **Driver callbacks and memory type helpers – the overall concept**
- **Memory-independent buffer (frame) management – including:**
  - handling cancels, unexpected closes…
  - memory leaks prevention
  - API compliance
  - ...

What should be improved/added

- Clear separation between queue management and memory handling
- V4L2 API compliance
- Memory allocation and mapping
- Streamoff/streamon handling
- iolock() – redesign
- Cache synchronization support
- Multi-plane video frames
- Waiting for buffers to be processed, out-of-order dequeuing

Clear roles

- **Videobuf queue**
  - manage buffers on a higher, memory-independent level
  - provide V4L2 API helper functions
  - should not be aware of memory handling at all
- **Videobuf memtype**
  - functions for video memory allocation, synchronization, mapping...
  - modular; pluggable/reusable parts
- **Driver**
  - act as a go-between
  - choose the tools it wants to use from a provided, standard pool
  - override everything else

V4L2 API compliance

- **Buffer allocation**
  - has to be performed on REQBUFS, not mmap
  - support freeing with REQBUFS = 0
- **streamoff/streamon**
  - do not free buffers on streamoff
- **Proper support for other than CAPTURE types**
  - videobuf has originally been written for capture devices only
- **...**

Memory allocation and mapping

- **Allocation should be performed on REQBUFS instead of mmap()**
- **It should be possible to free buffers with REQBUFS(0)**
- **An ability to plug-in custom allocation mechanisms is required**
- **Memory mapping functions could be pluggable as well**
- **DMA-SG module is a real mess – Laurent provided a new, clean implementation *which could not be integrated into videobuf1***

Memory allocators

- **Video data memory management in videobuf**
  - memory is allocated on mmap (or even on VM fault sometimes)
  - fixed methods are used for allocation and management (e.g.: dma_alloc_coherent() for physically contiguous memory)
  - drivers cannot utilize/plug-in their own methods

- **„Memory types” in videobuf are „take all or nothing”**
  - no way to override, no „ops”

- **The result**
  - drivers using parts of videobuf memory code only
  - code duplication, (big) chunks of videobuf code get copied
  - drivers not using videobuf at all

Requirements and considerations

- **Device requirements**
  - buffer contiguity
  - own memory pools
  - allocation from specific memory banks
  - allocation in a specific arrangement
- **Mapping**
  - specific CPU flags
  - problems with remapping and cache coherency, different flags
  - VM_PFNMAP memory
- **Other requirements**
  - reference counting
- **Solutions**
  - bootmem allocators
  - memory pools…

Rethinking memory types

- **Have a general pool of functions**
  - provide existing methods as standard solutions
  - let drivers choose from among them or provide their own
- **Uncouple videobuf queue code from memory type code**
  - let drivers stand between them and choose what to do
- **Provide new callbacks for drivers**
  - buffer_alloc() – called on REQBUFS
  - buffer_free() – called on REQBUFS(0) and on cleanup

Memory allocation TODO

- **Move memory allocation out of memory type mmap functions**
  - obvious problem: existing drivers depend on this
- **Allow drivers to plug-in their own memory allocation functions**
- **Store per-buffer private data related to allocation**

- **Could be done for videobuf1, or at least parts of it**
- **To work around the mmap allocation problem:**
  - allow drivers to initialize memory type code with their own allocation routines
  - do not allocate on mmap if a driver provided its own implementation
  - make core aware of that and make it call the provided allocation routines on reqbufs

Streamoff/streamon

- streamoff() currently frees buffers (!)
- So it is not possible to resume with streamon after “pausing” using streamoff
- New memory handling would fix this
- Again – big change for drivers

iolock() (and sync())

- **iolock() is a callback implemented by memory handling modules**
  - “do anything required to prepare a buffer for use by hardware”
- **iolock() is used for too many things**
  - buffer validation
  - bounce buffer allocation
  - page pinning
  - physical contiguity verification
  - scatter-gather list creation
  - cache synchronization (not currently)
  - IOMMU management (not currently)
- **iolock() is called on QBUF – might be too late**
  - verification, preparation, sync (...) of large buffers (e.g. 10 Mpix pictures) **takes time**

Rethinking iolock() and sync()

- **Preparing buffers for hardware**
  - actions performed once per buffer (on streamon/after allocation?)
  - actions performed before each HW operation (on each qbuf)
- **Returning buffers back to userspace**
  - actions performed after each HW operation (on dequeue)
  - actions performed before releasing memory

- **Extend the current API for drivers into:**
  - buffer_init() – once per buffer (e.g.: pin pages, verify contiguity, IOMMU mapping...)
  - buffer_prepare() – on every queue (e.g.: sync cache, copy to bounce buffer...)
  - buffer_finish() – on every dequeue (e.g.: sync cache, copy back...)
  - buffer_cleanup() – before releasing memory (e.g.: unmap...)

- **Get rid of iolock and sync from videobuf,**
  let drivers do what they need and call helpers (if required) from the above functions

Cache synchronization

- **Non-cache coherent architectures require cache synchronization before and after a hardware operation**
- **Currently we have a sync() call, but called after an operation only**
  - for cache sync
  - for copying data back from bounce buffers
- **But is called after a HW operation only**
- **Userspace sometimes knows that sync is not required**
- **We need:**
  - sync calls before an operation
  - add (a) flag(s) for userspace to indicate that a buffer does not have to be cache-synced
- **Can be performed with the new API in buffer_prepare() and buffer_finish()**

Multi-plane frames

- **Currently it is assumed that all video data of one frame is kept in one, contiguous memory buffer**
- **The idea is to have multiple memory buffers per frame – <u>planes</u>**

- **Some hardware requires several, physically discontiguous memory buffers**
- **Userspace might also want to pass video data in separate buffers**
  - e.g. Y, Cb and Cr planes in 3 separate buffers
- **Can be used for non-video data/metadata as well**
- **Some planes (video data) can be of MMAP-type (i.e. provided by drivers), while others can be USERPTR (i.e. provided by userspace)**

- **Generally doable with the current videobuf, although with some difficulties**
  - DMA-SG V4L1, mmap compatibility

Plane struct

```
struct v4l2_plane {
        __u32                   bytesused;

        union {
                __u32           offset;
                unsigned long   userptr;
        } m;
        __u32                   length;
        __u32                   hdr_size;
        __u32                   reserved[12];
};
```

Buffer struct

```
struct v4l2_buffer {
        __u32                   index;
        enum v4l2_buf_type      type;
        __u32                   bytesused;
        __u32                   flags;
        enum v4l2_field         field;
        struct timeval          timestamp;
        struct v4l2_timecode    timecode;
        __u32                   sequence;

        /* memory location */
        enum v4l2_memory        memory;
        union {
                __u32                   offset;
                unsigned long           userptr;
                struct v4l2_plane       *planes;
        } m;
        __u32                   length;
        __u32                   input;
        __u32                   reserved;
};
```

New: buffer dequeuing/waiting mechanisms

- **V4L2 API – DQBUF**
  - return a buffer (any); can be identified by index
  - no particular order enforced
- **Currently in videobuf**
  - buffers are stored in the same order as queued (FIFO)
  - passed to drivers in FIFO order
  - dqbuf and poll only consider the buffer that was **queued** first
- **Why change this?**
  - some devices require this – if they return buffers in a non-FIFO order, e.g. video codecs
  - operations on some buffers may be finished faster than on others (parallel in-device processing (?))

New: buffer dequeuing/waiting mechanisms

- **Current videobuf implementation**
  - each videobuf_buffer includes a waitqueue
  - dqbuf/poll take the first buffer and sleep on its waitqueue
  - drivers wake_up() those waitqueues

- **Proposed changes**
  - add a list of buffers that have finished being processed (**done_list**)
  - have a general per-videobuf_queue waitqueue (**done_wait**)

- **New mechanism**
  - drivers mark buffers as done with **videobuf_finish**()
  - **videobuf_finish**() adds buffers to the done list and wakes up done queue sleepers
  - **dqbuf**() and **poll**() sleep on the **done_wait** waitqueue

- **Old behavior, including the ability to wait for particular buffers, is preserved**
- **Or maybe get rid of per-buffer waitqueues after all? Do we really need this?**

Smaller stuff

- **Ensure full support for other queue types (other than CAPTURE)**
- **Drop V4L1 support**
- **Remove unused/unneeded variables**
  - videobuf_buffer: width, height, bytesperline... (format is managed by drivers)
- **Improve naming, reduce code duplication...**
  - videobuf_buffer -> videobuf_frame
  - videobuf_frame contains 1..n videobuf_planes

Converting existing drivers to videobuf2

- **Memory allocation – moved to reqbufs, with all implications**
- **Adapt to the new freeing/cleanup/cancel behavior**
- **Make sure streamoff works as expected**
- **Add implementation for new driver API functions**
- **Adapt to multi-planes**
  - quite simple, current buffers become multiplane buffers with one plane

Thank you!

Questions, suggestions, comments please?