

# An Introduction to the Video4Linux Framework

Hans Verkuil  
Cisco Systems Norway

# Features & Architecture



# Features

- Video capture/output and tuning (/dev/videoX, streaming and control)
- Video capture and output overlay (/dev/videoX, control)
- Memory-to-Memory (aka codec) devices (/dev/videoX, streaming and control)
- Raw and Sliced VBI capture and output (/dev/vbiX, streaming and control)
- Radio tuning and modulating (/dev/radioX, control, ALSA for streaming)
- RDS receiver/transmitter (/dev/radioX, streaming and control)
- Software Defined Radio (/dev/swradioX, streaming and control)
- Low-level sub-device control (/dev/v4l-subdevX, control)
- Device topology discovery/control (/dev/mediaX, control)

# Driver architecture

- The bridge driver controls the platform/USB/PCI/... hardware that is responsible for the DMA transfers.
- Based on the board configuration (USB ID, PCI ID, kernel config, device tree, module options) the necessary *sub-device* drivers are loaded.
- The bridge driver finally registers the device nodes it needs.
- Consequences for the Device Tree model: sub-devices need to defer initialization until the bridge driver has been loaded. The bridge driver needs to postpone initializing sub-devices until all required sub-devices have been loaded (v4l2-async).

# V4L2 PCI Skeleton Driver Basics



# Features of the Skeleton Driver

- It has two inputs: input 0 is an S-Video input (SDTV) and input 1 is an HDMI input (HDTV) up to 74.25 MHz pixelclock.
- It supports the 4:2:2 YUYV image format.
- It supports brightness, contrast, saturation and hue controls.

# struct v4l2\_device (1)

- Top level struct.
- Misnomer: a better name would have been v4l2\_root.
- v4l2\_device\_(un)register should have been called v4l2\_root\_init/exit.
- Maintains list of sub-devices.
- Has notify() callback for sub-devices.
- Has release() callback called when the last device reference goes away.

# struct v4l2\_device (2)

```
#include <linux/videodev2.h>
#include <media/v4l2-device.h>

MODULE_DESCRIPTION("V4L2 PCI Skeleton Driver");
MODULE_AUTHOR("Hans Verkuil");
MODULE_LICENSE("GPL v2");
MODULE_DEVICE_TABLE(pci, skeleton_pci_tbl);

struct skeleton {
    struct pci_dev *pdev;
    struct v4l2_device v4l2_dev;
};

static const struct pci_device_id skeleton_pci_tbl[] = {
    { PCI_DEVICE(PCI_VENDOR_ID_FOO, PCI_DEVICE_ID_BAR) },
    { 0, }
};

<skeleton_probe>
<skeleton_remove>

static struct pci_driver skeleton_driver = {
    .name = KBUILD_MODNAME,
    .probe = skeleton_probe,
    .remove = skeleton_remove,
    .id_table = skeleton_pci_tbl,
};

module_pci_driver(skeleton_driver);
```



# struct v4l2\_device (3)

```
static int skeleton_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
{
    struct skeleton *skel;
    int ret;

    pci_enable_device(pdev);
    pci_set_dma_mask(pdev, DMA_BIT_MASK(32));
    skel = devm_kzalloc(&pdev->dev, sizeof(struct skeleton), GFP_KERNEL);
    if (!skel)
        return -ENOMEM;
    skel->pdev = pdev;
    ret = v4l2_device_register(&pdev->dev, &skel->v4l2_dev);
    if (ret)
        goto disable_pci;
    dev_info(&pdev->dev, "V4L2 PCI Skeleton Driver loaded\n");
    return 0;

disable_pci:
    pci_disable_device(pdev);
    return ret;
}

static void skeleton_remove(struct pci_dev *pdev)
{
    struct v4l2_device *v4l2_dev = pci_get_drvdata(pdev);
    struct skeleton *skel = container_of(v4l2_dev, struct skeleton, v4l2_dev);

    v4l2_device_unregister(&skel->v4l2_dev);
    pci_disable_device(skel->pdev);
}
```

# struct video\_device (1)

- Represents a video/radio/vbi/v4l2\_subdev device node in /dev.
- Pointer to v4l2\_file\_operations for the file operations.
- Pointer to v4l2\_ioctl\_ops for ioctl operations.
- Often represents a DMA engine as well: pointer to vb2\_queue.
- Core locking support: lock mutex, vb2\_queue.lock:
  - If lock == NULL, then the driver does all locking.
  - If lock != NULL but vb2\_queue.lock == NULL, then all ioctls are serialized through that lock, including the streaming ioctls.
  - If vb2\_queue.lock is also != NULL then that lock is used for all the streaming ioctls: useful if other ioctls can hold the core lock for a long time (typical for USB drivers).
  - The driver always does all the locking for non-ioctl file operations.
- My personal recommendation: use core locking.

# struct video\_device (2)

```
struct skeleton {
    struct pci_dev *pdev;
    struct v4l2_device v4l2_dev;
    struct video_device vdev;
    struct mutex lock;
};

static int skeleton_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
{
    ...
    mutex_init(&skel->lock);
    vdev = &skel->vdev;
    strncpy(vdev->name, KBUILD_MODNAME, sizeof(vdev->name));
    vdev->release = video_device_release_empty;
    vdev->fops = &skel_fops,
    vdev->ioctl_ops = &skel_ioctl_ops,
    vdev->lock = &skel->lock;
    vdev->v4l2_dev = &skel->v4l2_dev;
    /* Supported SDTV standards, if any */
    vdev->tvnorms = V4L2_STD_ALL;
    set_bit(V4L2_FL_USE_FH_PRIO, &vdev->flags);
    video_set_drvdata(vdev, skel);

    ret = video_register_device(vdev, VFL_TYPE_GRABBER, -1);
    if (ret)
        goto v4l2_dev_unreg;

    dev_info(&pdev->dev, "V4L2 PCI Skeleton Driver loaded\n");
    return 0;
    ...
}
```

# struct video\_device (3)

```
static int skeleton_querycap(struct file *file, void *priv,
                             struct v4l2_capability *cap)
{
    struct skeleton *skel = video_drvdata(file);

    strncpy(cap->driver, KBUILD_MODNAME, sizeof(cap->driver));
    strncpy(cap->card, "V4L2 PCI Skeleton", sizeof(cap->card));
    snprintf(cap->bus_info, sizeof(cap->bus_info), "PCI:%s",
             pci_name(skel->pdev));
    cap->device_caps = V4L2_CAP_VIDEO_CAPTURE | V4L2_CAP_READWRITE |
                     V4L2_CAP_STREAMING;
    cap->capabilities = cap->device_caps | V4L2_CAP_DEVICE_CAPS;
    return 0;
}

static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    .vidioc_querycap = skeleton_querycap,
};

static const struct v4l2_file_operations skel_fops = {
    .owner = THIS_MODULE,
    .open = v4l2_fh_open,
    .release = v4l2_fh_release,
    .unlocked_ioctl = video_ioctl2,
};
```

# Input ioctls (1)

```
static int skeleton_enum_input(struct file *file, void *priv,
                              struct v4l2_input *i)
{
    if (i->index > 1)
        return -EINVAL;
    i->type = V4L2_INPUT_TYPE_CAMERA;
    if (i->index == 0) {
        i->std = V4L2_STD_ALL;
        strncpy(i->name, "S-Video", sizeof(i->name));
        i->capabilities = V4L2_IN_CAP_STD;
    } else {
        i->std = 0;
        strncpy(i->name, "HDMI", sizeof(i->name));
        i->capabilities = V4L2_IN_CAP_DV_TIMINGS;
    }
    return 0;
}

static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    .vidioc_enum_input = skeleton_enum_input,
};
```

# Input ioctls (2)

```
static int skeleton_s_input(struct file *file, void *priv, unsigned int i)
{
    struct skeleton *skel = video_drvdata(file);

    if (i > 1)
        return -EINVAL;
    skel->input = i;
    skel->vdev.tvnorms = i ? 0 : V4L2_STD_ALL;
    skeleton_fill_pix_format(skel, &skel->format);
    return 0;
}
```

```
static int skeleton_g_input(struct file *file, void *priv, unsigned int *i)
{
    struct skeleton *skel = video_drvdata(file);

    *i = skel->input;
    return 0;
}
```

```
static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    .vidioc_g_input = skeleton_g_input,
    .vidioc_s_input = skeleton_s_input,
};
```

# SDTV Standards ioctls (1)

```
static int skeleton_s_std(struct file *file, void *priv, v4l2_std_id std)
{
    struct skeleton *skel = video_drvdata(file);

    if (skel->input)
        return -ENODATA;
    if (std == skel->std)
        return 0;
    /* TODO: handle changing std */
    skel->std = std;
    skeleton_fill_pix_format(skel, &skel->format);
    return 0;
}
```

```
static int skeleton_g_std(struct file *file, void *priv, v4l2_std_id *std)
{
    struct skeleton *skel = video_drvdata(file);

    if (skel->input)
        return -ENODATA;
    *std = skel->std;
    return 0;
}
```

```
static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    .vidioc_g_std = skeleton_g_std,
    .vidioc_s_std = skeleton_s_std,
};
```

# SDTV Standards ioctls (2)

```
static int skeleton_querystd(struct file *file, void *priv,
                            v4l2_std_id *std)
{
    struct skeleton *skel = video_drvdata(file);

    if (skel->input)
        return -ENODATA;
    /* TODO: Query currently seen standard. */
    return 0;
}

static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    .vidioc_querystd = skeleton_querystd,
};
```



# DV Timings ioctls (1)

```
static const struct v4l2_dv_timings_cap skel_timings_cap = {
    .type = V4L2_DV_BT_656_1120,
    /* keep this initialization for compatibility with GCC < 4.4.6 */
    .reserved = { 0 },
    V4L2_INIT_BT_TIMINGS(
        720, 1920,          /* min/max width */
        480, 1080,         /* min/max height */
        27000000, 74250000, /* min/max pixelclock*/
        V4L2_DV_BT_STD_CEA861, /* Supported standards */
        /* capabilities */
        V4L2_DV_BT_CAP_INTERLACED | V4L2_DV_BT_CAP_PROGRESSIVE
    )
};

static int skeleton_dv_timings_cap(struct file *file, void *fh,
                                   struct v4l2_dv_timings_cap *cap)
{
    struct skeleton *skel = video_drvdata(file);

    if (skel->input == 0)
        return -ENODATA;
    *cap = skel_timings_cap;
    return 0;
}

static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    .vidioc_dv_timings_cap = skeleton_dv_timings_cap,
};
```

# DV Timings ioctls (2)

```
static int skeleton_s_dv_timings(struct file *file, void *_fh,
                                struct v4l2_dv_timings *timings)
{
    struct skeleton *skel = video_drvdata(file);

    if (skel->input == 0)
        return -ENODATA;
    if (!v4l2_valid_dv_timings(timings, &skel_timings_cap, NULL, NULL))
        return -EINVAL;
    if (!v4l2_find_dv_timings_cap(timings, &skel_timings_cap, 0, NULL, NULL))
        return -EINVAL;
    if (v4l2_match_dv_timings(timings, &skel->timings, 0))
        return 0;
    /* TODO: Configure new timings */
    skel->timings = *timings;
    skeleton_fill_pix_format(skel, &skel->format);
    return 0;
}

static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    .vidioc_s_dv_timings = skeleton_s_dv_timings,
};
```

# DV Timings ioctls (3)

```
static int skeleton_g_dv_timings(struct file *file, void *_fh,
                                struct v4l2_dv_timings *timings)
{
    struct skeleton *skel = video_drvdata(file);

    if (skel->input == 0)
        return -ENODATA;
    *timings = skel->timings;
    return 0;
}

static int skeleton_enum_dv_timings(struct file *file, void *_fh,
                                    struct v4l2_enum_dv_timings *timings)
{
    struct skeleton *skel = video_drvdata(file);

    if (skel->input == 0)
        return -ENODATA;
    return v4l2_enum_dv_timings_cap(timings, &skel_timings_cap, NULL, NULL);
}

static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    .vidioc_g_dv_timings = skeleton_g_dv_timings,
    .vidioc_enum_dv_timings = skeleton_enum_dv_timings,
};
```

# DV Timings ioctls (4)

```
static int skeleton_query_dv_timings(struct file *file, void *_fh,
                                     struct v4l2_dv_timings *timings)
{
    struct skeleton *skel = video_drvdata(file);

    if (skel->input == 0)
        return -ENODATA;

    /* TODO: Query currently seen timings. */
    detect_timings();
    if (no_signal)
        return -ENOLINK;
    if (cannot_lock_to_signal)
        return -ENOLCK;
    if (signal_out_of_range_of_capabilities)
        return -ERANGE;

    /* Useful for debugging */
    if (debug)
        v4l2_print_dv_timings(skel->v4l2_dev.name,
                              "query_dv_timings:",
                              timings, true);

    return 0;
}

static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    .vidioc_query_dv_timings = skeleton_query_dv_timings,
};
```

# Format ioctls (1)

```
static int skeleton_s_fmt_vid_cap(struct file *file, void *priv,
                                  struct v4l2_format *f)
{
    struct skeleton *skel = video_drvdata(file);
    int ret;

    ret = skeleton_try_fmt_vid_cap(file, priv, f);
    if (ret)
        return ret;
    /* TODO: change format */
    skel->format = f->fmt.pix;
    return 0;
}

static int skeleton_g_fmt_vid_cap(struct file *file, void *priv,
                                  struct v4l2_format *f)
{
    struct skeleton *skel = video_drvdata(file);

    f->fmt.pix = skel->format;
    return 0;
}

static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    .vidioc_s_fmt_vid_cap = skeleton_s_fmt_vid_cap,
    .vidioc_g_fmt_vid_cap = skeleton_g_fmt_vid_cap,
};
```

# Format ioctls (2)

```
static int skeleton_enum_fmt_vid_cap(struct file *file, void *priv,
                                     struct v4l2_fmtdesc *f)
{
    if (f->index != 0)
        return -EINVAL;
    strncpy(f->description, "4:2:2, packed, YUYV",
            sizeof(f->description));
    f->pixelformat = V4L2_PIX_FMT_YUYV;
    f->flags = 0;
    return 0;
}

static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    .vidioc_enum_fmt_vid_cap = skeleton_enum_fmt_vid_cap,
};
```

# Format ioctls (3)

```
static void skeleton_fill_pix_format(struct skeleton *skel, struct v4l2_pix_format *pix)
{
    pix->pixelformat = V4L2_PIX_FMT_YUYV;
    if (skel->input == 0) {
        pix->width = 720;
        pix->height = (skel->std & V4L2_STD_525_60) ? 480 : 576;
        pix->field = V4L2_FIELD_INTERLACED;
        pix->colospace = V4L2_COLORSPACE_SMPTE170M;
    } else {
        pix->width = skel->timings.bt.width;
        pix->height = skel->timings.bt.height;
        if (skel->timings.bt.interlaced) {
            pix->field = V4L2_FIELD_ALTERNATE;
            pix->height /= 2;
        } else {
            pix->field = V4L2_FIELD_NONE;
        }
        pix->colospace = V4L2_COLORSPACE_REC709;
    }
    pix->bytesperline = pix->width * 2;
    pix->sizeimage = pix->bytesperline * pix->height;
    pix->priv = 0;
}
```

```
static int skeleton_try_fmt_vid_cap(struct file *file, void *priv, struct v4l2_format *f)
{
    struct skeleton *skel = video_drvdata(file);
    struct v4l2_pix_format *pix = &f->fmt.pix;

    if (pix->pixelformat != V4L2_PIX_FMT_YUYV)
        return -EINVAL;
    skeleton_fill_pix_format(skel, pix);
    return 0;
}
```

```
static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    .vidioc_try_fmt_vid_cap = skeleton_try_fmt_vid_cap,
};
```

# V4L2 PCI Skeleton Driver Streaming





# Streaming Modes

- Read and Write
- Memory Mapped Streaming I/O: memory allocated by the driver, mmap()ed into userspace.
- User Pointer Streaming I/O: memory allocated by userspace, requires scatter-gather DMA support.
- DMABUF Streaming I/O: memory allocated by another device, exported as a DMABUF file handler and imported in this driver.

# Streaming Support (1)

```
#include <media/videobuf2-dma-contig.h>

struct skeleton {
    ...
    struct vb2_queue queue;
    struct vb2_alloc_ctx *alloc_ctx;

    spinlock_t qlock;
    struct list_head buf_list;
    unsigned field;
    unsigned sequence;
};

struct skel_buffer {
    struct vb2_buffer vb;
    struct list_head list;
};

static inline struct skel_buffer *to_skel_buffer(struct vb2_buffer *vb2)
{
    return container_of(vb2, struct skel_buffer, vb);
}
```

# Streaming Support (2)

```
static int skeleton_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
{
    ...
    q = &skel->queue;
    q->type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    q->io_modes = VB2_MMAP | VB2_DMABUF | VB2_READ;
    q->drv_priv = skel;
    q->buf_struct_size = sizeof(struct skel_buffer);
    q->ops = &skel_qops;
    q->mem_ops = &vb2_dma_contig_memops;
    q->timestamp_flags = V4L2_BUF_FLAG_TIMESTAMP_MONOTONIC;
    q->lock = &skel->lock;
    q->gfp_flags = GFP_DMA32;
    q->min_buffers_needed = 2;
    ret = vb2_queue_init(q);
    if (ret)
        goto v4l2_dev_unreg;

    skel->alloc_ctx = vb2_dma_contig_init_ctx(&pdev->dev);
    if (IS_ERR(skel->alloc_ctx)) {
        dev_err(&pdev->dev, "Can't allocate buffer context");
        ret = PTR_ERR(skel->alloc_ctx);
        goto v4l2_dev_unreg;
    }
    INIT_LIST_HEAD(&skel->buf_list);
    spin_lock_init(&skel->qlock);
    ...
    vdev->queue = q;
    ...
}
```

# Streaming Support (3)

```
static struct vb2_ops skel_qops = {
    .queue_setup      = queue_setup,
    .buf_prepare      = buffer_prepare,
    .buf_queue        = buffer_queue,
    .start_streaming  = start_streaming,
    .stop_streaming   = stop_streaming,
    .wait_prepare     = vb2_ops_wait_prepare,
    .wait_finish      = vb2_ops_wait_finish,
};

static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    ...
    .vidioc_reqbufs   = vb2_ioctl_reqbufs,
    .vidioc_create_bufs = vb2_ioctl_create_bufs,
    .vidioc_querybuf   = vb2_ioctl_querybuf,
    .vidioc_qbuf       = vb2_ioctl_qbuf,
    .vidioc_dqbuf      = vb2_ioctl_dqbuf,
    .vidioc_expbuf     = vb2_ioctl_expbuf,
    .vidioc_streamon   = vb2_ioctl_streamon,
    .vidioc_streamoff  = vb2_ioctl_streamoff,
};

static const struct v4l2_file_operations skel_fops = {
    .owner = THIS_MODULE,
    .open = v4l2_fh_open,
    .release = vb2_fop_release,
    .unlocked_ioctl = video_ioctl2,
    .read = vb2_fop_read,
    .mmap = vb2_fop_mmap,
    .poll = vb2_fop_poll,
};
```

# Streaming Support (4)

```
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[])
{
    struct skeleton *skel = vb2_get_drv_priv(vq);

    skel->field = skel->format.field;
    if (skel->field == V4L2_FIELD_ALTERNATE) {
        if (vb2_fileio_is_active(q))
            return -EINVAL;
        skel->field = V4L2_FIELD_TOP;
    }
    if (vq->num_buffers + *nbuffers < 3)
        *nbuffers = 3 - vq->num_buffers;

    if (fmt && fmt->fmt.pix.sizeimage < skel->format.sizeimage)
        return -EINVAL;
    *nplanes = 1;
    sizes[0] = fmt ? fmt->fmt.pix.sizeimage :
                skel->format.sizeimage;
    alloc_ctxs[0] = skel->alloc_ctx;
    return 0;
}
```

# Streaming Support (5)

```
static void all_buffers_done(struct skeleton *skel, enum vb2_buffer_state state)
{
    struct skel_buffer *buf, *node;
    unsigned long flags;

    spin_lock_irqsave(&skel->qlock, flags);
    list_for_each_entry_safe(buf, node, &skel->buf_list, list) {
        vb2_buffer_done(&buf->vb, state);
        list_del(&buf->list);
    }
    spin_unlock_irqrestore(&skel->qlock, flags);
}
```

```
static int start_streaming(struct vb2_queue *vq, unsigned int count)
{
    struct skeleton *skel = vb2_get_drv_priv(vq);
    int ret = 0;

    skel->sequence = 0;
    /* TODO: start DMA */
    if (ret)
        all_buffers_done(skel, VB2_BUF_STATE_QUEUED);
    return ret;
}
```

```
static void stop_streaming(struct vb2_queue *vq)
{
    struct skeleton *skel = vb2_get_drv_priv(vq);

    /* TODO: stop DMA */
    all_buffers_done(skel, VB2_BUF_STATE_ERROR);
}
```

# Streaming Support (6)

```
static int buffer_prepare(struct vb2_buffer *vb)
{
    struct skeleton *skel = vb2_get_drv_priv(vb->vb2_queue);
    unsigned long size = skel->format.sizeimage;

    if (vb2_plane_size(vb, 0) < size) {
        dev_err(&skel->pdev->dev, "buffer too small (%lu < %lu)\n",
                vb2_plane_size(vb, 0), size);
        return -EINVAL;
    }
    vb2_set_plane_payload(vb, 0, size);
    return 0;
}
```

```
static void buffer_queue(struct vb2_buffer *vb)
{
    struct skeleton *skel = vb2_get_drv_priv(vb->vb2_queue);
    struct skel_buffer *buf = to_skel_buffer(vb);
    unsigned long flags;

    spin_lock_irqsave(&skel->qlock, flags);
    list_add_tail(&buf->list, &skel->buf_list);

    /* TODO: Update any DMA pointers if necessary */

    spin_unlock_irqrestore(&skel->qlock, flags);
}
```

# Streaming Support (7)

```
static irqreturn_t skeleton_irq(int irq, void *dev_id)
{
    struct skeleton *skel = dev_id;

    /* TODO: handle interrupt */

    if (captured_new_frame) {
        ...
        spin_lock(&skel->qlock);
        list_del(&new_buf->list);
        spin_unlock(&skel->qlock);
        new_buf->vb.v4l2_buf.sequence = skel->sequence++;
        new_buf->vb.v4l2_buf.field = skel->field;
        if (skel->format.field == V4L2_FIELD_ALTERNATE) {
            if (skel->field == V4L2_FIELD_BOTTOM)
                skel->field = V4L2_FIELD_TOP;
            else if (skel->field == V4L2_FIELD_TOP)
                skel->field = V4L2_FIELD_BOTTOM;
        }
        v4l2_get_timestamp(&new_buf->vb.v4l2_buf.timestamp);
        vb2_buffer_done(&new_buf->vb, VB2_BUF_STATE_DONE);
    }
    return IRQ_HANDLED;
}
```



# Streaming Support (8)

Add this check:

```
if (vb2_is_busy(&skel->queue))  
    return -EBUSY;
```

to:

```
skeleton_s_input()  
skeleton_s_std()  
skeleton_s_dv_timings()  
skeleton_s_fmt_vid_cap()
```

# V4L2 PCI Skeleton Driver Control Framework



# Control Framework

- Can inherit controls from other control handlers, particularly from sub-devices.
- Controls can be combined to clusters if they have to be set together.
- Validation and atomicity is handled by the framework.
- Integrates with the event handling to allow control events (i.e. get an event when a control changes value or state).
- Bridge driver can be notified when a control of a sub-device changes.
- Support for auto-clusters. For example: AUTOGAIN and GAIN controls.
- It is possible to set a control handler at the `v4l2_device` level, at the `video_device` level or at the `v4l2_fh` level. In sub-devices the control handler is always at the `v4l2_subdev` level.

# Control Support (1)

```
#include <media/v4l2-ctrls.h>
#include <media/v4l2-event.h>

struct skeleton {
    ...
    struct v4l2_ctrl_handler ctrl_handler;
    ...
};

static const struct v4l2_ctrl_ops skel_ctrl_ops = {
    .s_ctrl = skeleton_s_ctrl,
};

static const struct v4l2_ioctl_ops skel_ioctl_ops = {
    ...
    .vidioc_log_status = v4l2_ctrl_log_status,
    .vidioc_subscribe_event = v4l2_ctrl_subscribe_event,
    .vidioc_unsubscribe_event = v4l2_event_unsubscribe,
};
```

# Control Support (2)

```
static int skeleton_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
{
    ...
    struct v4l2_ctrl_handler *hdl;
    ...
    hdl = &skel->ctrl_handler;
    v4l2_ctrl_handler_init(hdl, 4);
    v4l2_ctrl_new_std(hdl, &skel_ctrl_ops,
                     V4L2_CID_BRIGHTNESS, 0, 255, 1, 127);
    v4l2_ctrl_new_std(hdl, &skel_ctrl_ops,
                     V4L2_CID_CONTRAST, 0, 255, 1, 16);
    v4l2_ctrl_new_std(hdl, &skel_ctrl_ops,
                     V4L2_CID_SATURATION, 0, 255, 1, 127);
    v4l2_ctrl_new_std(hdl, &skel_ctrl_ops,
                     V4L2_CID_HUE, -128, 127, 1, 0);
    if (hdl->error) {
        ret = hdl->error;
        goto free_hdl;
    }
    skel->v4l2_dev.ctrl_handler = hdl;
    ...

free_hdl:
    v4l2_ctrl_handler_free(&skel->ctrl_handler);
    v4l2_device_unregister(&skel->v4l2_dev);
disable_pci:
    pci_disable_device(pdev);
    return ret;
}
```

# Control Support (3)

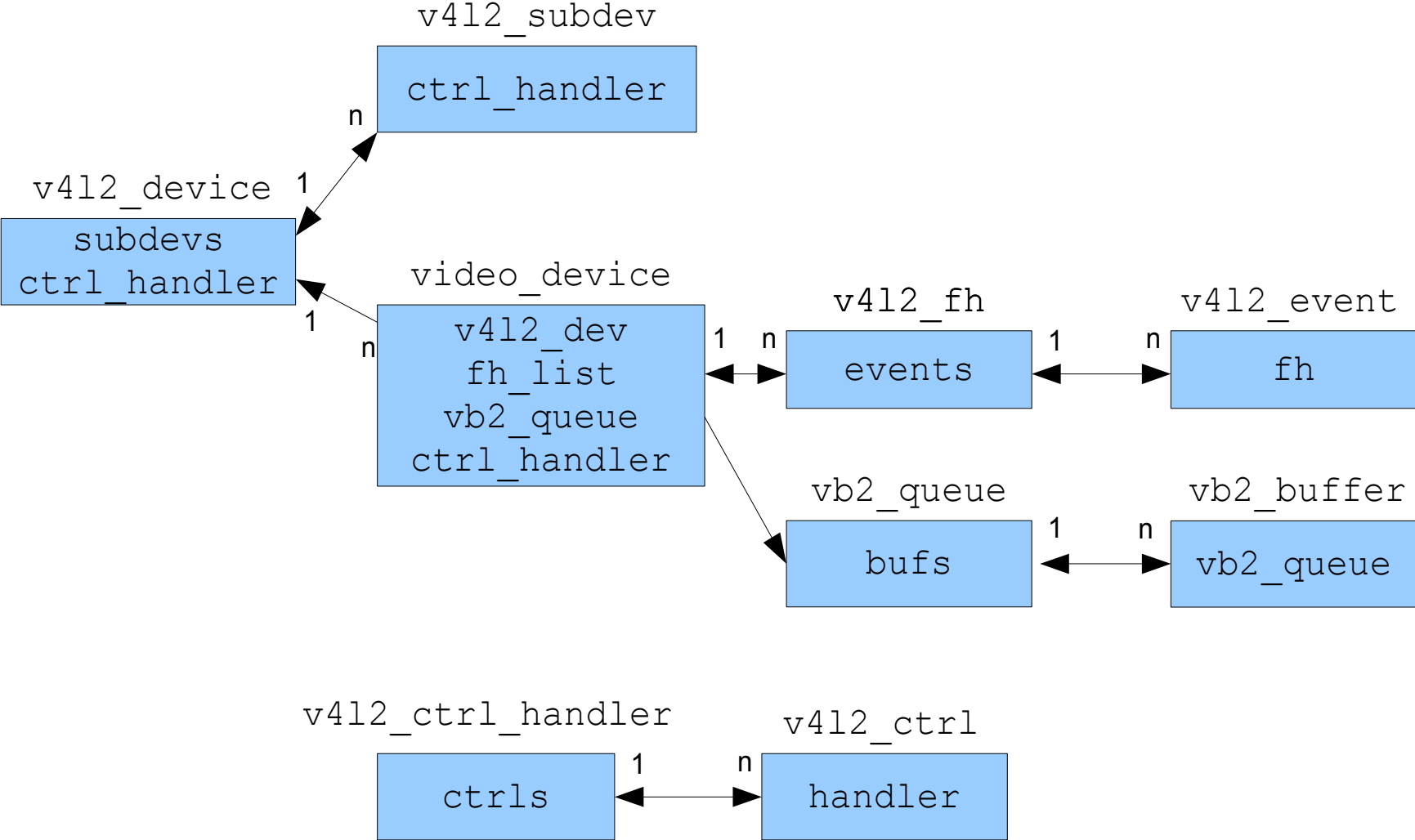
```
static int skeleton_s_ctrl(struct v4l2_ctrl *ctrl)
{
    struct skeleton *skel =
        container_of(ctrl->handler, struct skeleton, ctrl_handler);

    switch (ctrl->id) {
    case V4L2_CID_BRIGHTNESS:
        /* TODO: set brightness to ctrl->val */
        break;
    case V4L2_CID_CONTRAST:
        /* TODO: set contrast to ctrl->val */
        break;
    case V4L2_CID_SATURATION:
        /* TODO: set saturation to ctrl->val */
        break;
    case V4L2_CID_HUE:
        /* TODO: set hue to ctrl->val */
        break;
    default:
        return -EINVAL;
    }
    return 0;
}
```

# V4L2 Framework



# V4L2 Framework





# Resources



# Resources

- Linux Media Infrastructure API: <http://linuxtv.org/downloads/v4l-dvb-apis>. Latest version: <http://hverkuil.home.xs4all.nl/spec/media.html>
- Documentation/video4linux/v4l2-framework.txt and v4l2-controls.txt
- include/media/videobuf2-core.h
- Upstream media git repository: [http://git.linuxtv.org/media\\_tree.git](http://git.linuxtv.org/media_tree.git)
- Skeleton driver: Documentation/video4linux/v4l2-pci-skeleton.c
- v4l-utils git repository: <http://git.linuxtv.org/v4l-utils.git>
- linux-media mailinglist & irc channel: <http://linuxtv.org/lists.php>

# Utilities



# Utilities

- v4l2-ctl: Swiss army knife for v4l2.
- v4l2-compliance: V4L2 compliance driver testing.
- v4l2-dbg: allows access to `DBG_G_CHIP_INFO`, `DBG_G/S_REGISTER`.
- qv4l2: Qt test application.
- Core debugging: `'echo 1 >/sys/class/video4linux/video0/debug'`.
  - 1: show ioctl name
  - 2: show arguments as well

# Thank You!

