Table of Contents

V4L2 events

Step 1: Introduction

The V4L2 events provide a generic way to pass events to user space. The driver must use **v4l2_fh** to be able to support **V4L2 events.**

Events are subscribed per-file handle. An event specification consists of a type and is optionally associated with an object identified through the id field. If unused, then the id is 0. So an event is uniquely identified by the (type, id) tuple.

Step 2: v4l2_subdev_fops

```
const struct v4l2_file_operations v4l2_subdev_fops = {
	.owner = THIS_MODULE,
	.open = subdev_open,
	.unlocked_ioctl = subdev_ioctl,
#ifdef CONFIG_COMPAT
	.compat_ioctl32 = subdev_compat_ioctl32,
#endif
	.release = subdev_close,
	.poll = subdev_poll,
};
```

Step 3: subdev_open

```
static int subdev_open(struct file *file)
{
	struct video_device *vdev = video_devdata(file);
	struct v4l2_subdev *sd = vdev_to_v4l2_subdev(vdev);
	struct v4l2_subdev_fh *subdev_fh;
	int ret;

	subdev_fh = kzalloc(sizeof(*subdev_fh), GFP_KERNEL);
```

```
        if (subdev_fh == NULL)
                return -ENOMEM;

        ret = subdev_fh_init(subdev_fh, sd);
        if (ret) {
                kfree(subdev_fh);
                return ret;
        }

        v4l2_fh_init(&subdev_fh->vfh, vdev);
        v4l2_fh_add(&subdev_fh->vfh);
        file->private_data = &subdev_fh->vfh;
}



void v4l2_fh_init(struct v4l2_fh *fh, struct video_device *vdev)
{
        fh->vdev = vdev;
        /* Inherit from video_device. May be overridden by the driver. */
        fh->ctrl_handler = vdev->ctrl_handler;
        INIT_LIST_HEAD(&fh->list);
        set_bit(V4L2_FL_USES_V4L2_FH, &fh->vdev->flags);
        /*
         * determine_valid_ioctls() does not know if struct v4l2_fh
         * is used by this driver, but here we do. So enable the
         * prio ioctls here.
         */
        set_bit(_IOC_NR(VIDIOC_G_PRIORITY), vdev->valid_ioctls);
        set_bit(_IOC_NR(VIDIOC_S_PRIORITY), vdev->valid_ioctls);
        fh->prio = V4L2_PRIORITY_UNSET;
        init_waitqueue_head(&fh->wait);
        INIT_LIST_HEAD(&fh->available);
        INIT_LIST_HEAD(&fh->subscribed);
        fh->sequence = -1;
        mutex_init(&fh->subscribe_lock);
}
EXPORT_SYMBOL_GPL(v4l2_fh_init);

void v4l2_fh_add(struct v4l2_fh *fh)
{
        unsigned long flags;

        v4l2_prio_open(fh->vdev->prio, &fh->prio);
        spin_lock_irqsave(&fh->vdev->fh_lock, flags);
        list_add(&fh->list, &fh->vdev->fh_list);
        spin_unlock_irqrestore(&fh->vdev->fh_lock, flags);
}
```

EXPORT_SYMBOL_GPL(v4l2_fh_add);

Step 4: subdev_close

```
static int subdev_close(struct file *file)
{
	struct video_device *vdev = video_devdata(file);
	struct v4l2_subdev *sd = vdev_to_v4l2_subdev(vdev);
	struct v4l2_fh *vfh = file->private_data;
	struct v4l2_subdev_fh *subdev_fh = to_v4l2_subdev_fh(vfh);

	if (sd->internal_ops && sd->internal_ops->close)
		sd->internal_ops->close(sd, subdev_fh);
#if defined(CONFIG_MEDIA_CONTROLLER)
	if (sd->v4l2_dev->mdev)
		media_entity_put(&sd->entity);
#endif
	 v4l2_fh_del(vfh);
	 v4l2_fh_exit(vfh);
	 subdev_fh_free(subdev_fh);
	kfree(subdev_fh);
	file->private_data = NULL;

	return 0;
}
```

Step 5: subdev_ioctl

```
static long subdev_ioctl(struct file *file, unsigned int cmd,
	unsigned long arg)
{
	return video_usercopy(file, cmd, arg, subdev_do_ioctl_lock);
}

static long subdev_do_ioctl_lock(struct file *file, unsigned int cmd, void
*arg)
{
	struct video_device *vdev = video_devdata(file);
	struct mutex *lock = vdev->lock;
	long ret = -ENODEV;

	if (lock && mutex_lock_interruptible(lock))
```

```
		return -ERESTARTSYS;
	if (video_is_registered(vdev))
		ret = subdev_do_ioctl(file, cmd, arg);
	if (lock)
		mutex_unlock(lock);
	return ret;
}


static long subdev_do_ioctl(struct file *file, unsigned int cmd, void *arg)
{
	struct video_device *vdev = video_devdata(file);
	struct v4l2_subdev *sd = vdev_to_v4l2_subdev(vdev);
	struct v4l2_fh *vfh = file->private_data;
#if defined(CONFIG_VIDEO_V4L2_SUBDEV_API)
	struct v4l2_subdev_fh *subdev_fh = to_v4l2_subdev_fh(vfh);
	int rval;
#endif

	switch (cmd) {
	case VIDIOC_DQEVENT:
		if (!(sd->flags & V4L2_SUBDEV_FL_HAS_EVENTS))
			return -ENOIOCTLCMD;

		return v4l2_event_dequeue(vfh, arg, file->f_flags & O_NONBLOCK);

	case VIDIOC_SUBSCRIBE_EVENT:
		return v4l2_subdev_call(sd, core, subscribe_event, vfh, arg);

	case VIDIOC_UNSUBSCRIBE_EVENT:
		return v4l2_subdev_call(sd, core, unsubscribe_event, vfh, arg);
	 }
}
```

Step 6: Events

```
/*
 *	E V E N T S
 */

#define V4L2_EVENT_ALL				0
#define V4L2_EVENT_VSYNC			1
#define V4L2_EVENT_EOS				2
#define V4L2_EVENT_CTRL				3
#define V4L2_EVENT_FRAME_SYNC			4
```

```
#define V4L2_EVENT_SOURCE_CHANGE       5
#define V4L2_EVENT_MOTION_DET          6
#define V4L2_EVENT_PRIVATE_START       0x08000000


/*
 * Events
 *
* V4L2_EVENT_XLNXSDIRX_UNDERFLOW:
Video in to AXI4 Stream core underflowed
* V4L2_EVENT_XLNXSDIRX_OVERFLOW:
Video in to AXI4 Stream core overflowed
 */

#define V4L2_EVENT_XLNXSDIRX_CLASS      (V4L2_EVENT_PRIVATE_START |
0x200)
#define V4L2_EVENT_XLNXSDIRX_UNDERFLOW  (V4L2_EVENT_XLNXSDIRX_CLASS
| 0x1)
#define V4L2_EVENT_XLNXSDIRX_OVERFLOW   (V4L2_EVENT_XLNXSDIRX_CLASS |
0x2)
```

Step 7: subscribe_event & unsubscribe_event

```
static const struct v4l2_subdev_core_ops xsdirxss_core_ops = {
        .log_status = xsdirxss_log_status,
        .subscribe_event = xsdirxss_subscribe_event,
        .unsubscribe_event = xsdirxss_unsubscribe_event
};



static int xsdirxss_subscribe_event(struct v4l2_subdev *sd,
                                    struct v4l2_fh *fh,
                                    struct v4l2_event_subscription *sub)
{
        int ret;
        struct xsdirxss_state *xsdirxss = to_xsdirxssstate(sd);
        struct xsdirxss_core *core = &xsdirxss->core;

        switch (sub->type) {
        case V4L2_EVENT_XLNXSDIRX_UNDERFLOW:
        case V4L2_EVENT_XLNXSDIRX_OVERFLOW:
                ret = v4l2_event_subscribe(fh, sub, XSDIRX_MAX_EVENTS, NULL);
```

```
        break;
    case V4L2_EVENT_SOURCE_CHANGE:
            ret = v4l2_src_change_event_subscribe(fh, sub);
            break;
     default:
            return -EINVAL;
     }
     dev_dbg(core->dev, "Event subscribed : 0x%08x\n", sub->type);
     return ret;
}

static int xsdirxss_unsubscribe_event(struct v4l2_subdev *sd,
                                      struct v4l2_fh *fh,
                                      struct v4l2_event_subscription *sub)
{
     struct xsdirxss_state *xsdirxss = to_xsdirxssstate(sd);
     struct xsdirxss_core *core = &xsdirxss->core;

     dev_dbg(core->dev, "Event unsubscribe : 0x%08x\n", sub->type);
     return v4l2_event_unsubscribe(fh, sub);
}
```

Step 8: xsdirxss_irq_handler

```
static irqreturn_t xsdirxss_irq_handler(int irq, void *dev_id)
{
     struct xsdirxss_state *state = (struct xsdirxss_state *)dev_id;
     struct xsdirxss_core *core = &state->core;
     u32 status;

     status = xsdirxss_read(core, XSDIRX_ISR_REG);
     dev_dbg(core->dev, "interrupt status = 0x%08x\n", status);

     if (!status)
            return IRQ_NONE;

     xsdirxss_write(core, XSDIRX_ISR_REG, status);

     if (status & XSDIRX_INTR_VIDLOCK_MASK ||
        status & XSDIRX_INTR_VIDUNLOCK_MASK) {
            u32 val1, val2;

            dev_dbg(core->dev, "video lock/unlock interrupt\n");
```

```
            xsdirx_streamflow_control(core, false);
            state->streaming = false;
            val1 = xsdirxss_read(core, XSDIRX_MODE_DET_STAT_REG);
            val2 = xsdirxss_read(core, XSDIRX_TS_DET_STAT_REG);
            if ((val1 & XSDIRX_MODE_DET_STAT_MODE_LOCK_MASK) &&
               (val2 & XSDIRX_TS_DET_STAT_LOCKED_MASK)) {
                  u32 mask = XSDIRX_RST_CTRL_RST_CRC_ERRCNT_MASK |
                           XSDIRX_RST_CTRL_RST_EDH_ERRCNT_MASK;

                  dev_dbg(core->dev, "video lock interrupt\n");

                  xsdirxss_set(core, XSDIRX_RST_CTRL_REG, mask);
                  xsdirxss_clr(core, XSDIRX_RST_CTRL_REG, mask);

                  val1 = xsdirxss_read(core, XSDIRX_ST352_VALID_REG);
                  val2 = xsdirxss_read(core, XSDIRX_ST352_DS1_REG);

                  dev_dbg(core->dev, "valid st352 mask = 0x%08x\n", val1);
                  dev_dbg(core->dev, "st352 payload = 0x%08x\n", val2);

                  if (!xsdirx_get_stream_properties(state)) {
                        state->vidlocked = true;
                  } else {
                        dev_err(core->dev, "Unable to get stream properties!\n");
                        state->vidlocked = false;
                  }
            } else {
                  dev_dbg(core->dev, "video unlock interrupt\n");
                  state->vidlocked = false;
            }

            memset(&state->event, 0, sizeof(state->event));
            state->event.type = V4L2_EVENT_SOURCE_CHANGE;
            state->event.u.src_change.changes =
                  V4L2_EVENT_SRC_CH_RESOLUTION;
             v4l2_subdev_notify_event(&state->subdev, &state->event);
      }

      if (status & XSDIRX_INTR_UNDERFLOW_MASK) {
            dev_dbg(core->dev, "Video in to AXI4 Stream core underflow
interrupt\n");

            memset(&state->event, 0, sizeof(state->event));
            state->event.type = V4L2_EVENT_XLNXSDIRX_UNDERFLOW;
            v4l2_subdev_notify_event(&state->subdev, &state->event);
      }

      if (status & XSDIRX_INTR_OVERFLOW_MASK) {
```

```
                dev_dbg(core->dev, "Video in to AXI4 Stream core overflow
interrupt\n");

                memset(&state->event, 0, sizeof(state->event));
                state->event.type = V4L2_EVENT_XLNXSDIRX_OVERFLOW;
                v4l2_subdev_notify_event(&state->subdev, &state->event);
        }
        return IRQ_HANDLED;
}
```

Step 9: subdev_poll

```
static __poll_t subdev_poll(struct file *file, poll_table *wait)
{
        struct video_device *vdev = video_devdata(file);
        struct v4l2_subdev *sd = vdev_to_v4l2_subdev(vdev);
        struct v4l2_fh *fh = file->private_data;

        if (!(sd->flags & V4L2_SUBDEV_FL_HAS_EVENTS))
                return EPOLLERR;

        poll_wait(file, &fh->wait, wait);

        if (v4l2_event_pending(fh))
                return EPOLLPRI;

        return 0;
}
```