Overview of the V4L2 driver framework

This text documents the various structures provided by the V4L2 framework and their relationships.

Introduction

The V4L2 drivers tend to be very complex due to the complexity of the hardware: most devices have multiple ICs, export multiple device nodes in /dev, and create also non-V4L2 devices such as DVB, ALSA, FB, I2C and input (IR) devices.

Especially the fact that V4L2 drivers have to setup supporting ICs to do audio/video muxing/encoding/decoding makes it more complex than most. Usually these ICs are connected to the main bridge driver through one or more I2C busses, but other busses can also be used. Such devices are called 'sub-devices'.

For a long time the framework was limited to the video_device struct for creating V4L device nodes and video_buf for handling the video buffers (note that this document does not discuss the video_buf framework).

This meant that all drivers had to do the setup of device instances and connecting to sub-devices themselves. Some of this is quite complicated to do right and many drivers never did do it correctly.

There is also a lot of common code that could never be refactored due to the lack of a framework.

So this framework sets up the basic building blocks that all drivers need and this same framework should make it much easier to refactor common code into utility functions shared by all drivers.

Structure of a driver

All drivers have the following structure:

1) A struct for each device instance containing the device state.

2) A way of initializing and commanding sub-devices (if any).

3) Creating V4L2 device nodes (/dev/videoX, /dev/vbiX, /dev/radioX and /dev/vtxX) and keeping track of device-node specific data.

4) Filehandle-specific structs containing per-filehandle data;

5) video buffer handling.

This is a rough schematic of how it all relates:

```
    device instances
```

```
      |
      +-sub-device instances
      |
      \-V4L2 device nodes
          |
          \-filehandle instances
```

Structure of the framework

The framework closely resembles the driver structure: it has a v4l2_device struct for the device instance data, a v4l2_subdev struct to refer to sub-device instances, the video_device struct stores V4L2 device node data and in the future a v4l2_fh struct will keep track of filehandle instances (this is not yet implemented).

struct v4l2_device

Each device instance is represented by a struct v4l2_device (v4l2-device.h). Very simple devices can just allocate this struct, but most of the time you would embed this struct inside a larger struct.

You must register the device instance:

```
	v4l2_device_register(struct device *dev, struct v4l2_device *v4l2_dev);
```

Registration will initialize the v4l2_device struct and link dev->driver_data to v4l2_dev. If v4l2_dev->name is empty then it will be set to a value derived from dev (driver name followed by the bus_id, to be precise). If you set it up before calling v4l2_device_register then it will be untouched. If dev is NULL, then you *must* setup v4l2_dev->name before calling v4l2_device_register.

You can use v4l2_device_set_name() to set the name based on a driver name and a driver-global atomic_t instance. This will generate names like ivtv0, ivtv1, etc. If the name ends with a digit, then it will insert a dash: cx18-0, cx18-1, etc. This function returns the instance number.

The first 'dev' argument is normally the struct device pointer of a pci_dev, usb_interface or platform_device. It is rare for dev to be NULL, but it happens with ISA devices or when one device creates multiple PCI devices, thus making it impossible to associate v4l2_dev with a particular parent.

You can also supply a notify() callback that can be called by sub-devices to notify you of events. Whether you need to set this depends on the sub-device. Any notifications a sub-device supports must be defined in a header in include/media/<subdevice>.h.

You unregister with:

```
	v4l2_device_unregister(struct v4l2_device *v4l2_dev);
```

Unregistering will also automatically unregister all subdevs from the device.

If you have a hotpluggable device (e.g. a USB device), then when a disconnect happens the parent device becomes invalid. Since v4l2_device has a pointer to that parent device it has to be cleared as well to mark that the parent is gone. To do this call:

```
	v4l2_device_disconnect(struct v4l2_device *v4l2_dev);
```

This does *not* unregister the subdevs, so you still need to call the v4l2_device_unregister() function for that. If your driver is not hotpluggable, then there is no need to call v4l2_device_disconnect().

Sometimes you need to iterate over all devices registered by a specific driver. This is usually the case if multiple device drivers use the same hardware. E.g. the ivtvfb driver is a framebuffer driver that uses the ivtv hardware. The same is true for alsa drivers for example.

You can iterate over all registered devices as follows:

```
static int callback(struct device *dev, void *p)
{
	struct v4l2_device *v4l2_dev = dev_get_drvdata(dev);

	/* test if this device was inited */
	if (v4l2_dev == NULL)
		return 0;
	...
	return 0;
}

int iterate(void *p)
{
	struct device_driver *drv;
	int err;

	/* Find driver 'ivtv' on the PCI bus.
	   pci_bus_type is a global. For USB busses use usb_bus_type. */
	drv = driver_find("ivtv", &pci_bus_type);
	/* iterate over all ivtv device instances */
	err = driver_for_each_device(drv, NULL, p, callback);
	put_driver(drv);
	return err;
}
```

Sometimes you need to keep a running counter of the device instance. This is commonly used to map a device instance to an index of a module option array.

The recommended approach is as follows:

```
static atomic_t drv_instance = ATOMIC_INIT(0);

static int __devinit drv_probe(struct pci_dev *pdev,
				const struct pci_device_id *pci_id)
{
	...
	state->instance = atomic_inc_return(&drv_instance) - 1;
}
```

struct v4l2_subdev

Many drivers need to communicate with sub-devices. These devices can do all sort of tasks, but most commonly they handle audio and/or video muxing, encoding or decoding. For webcams common sub-devices are sensors and camera controllers.

Usually these are I2C devices, but not necessarily. In order to provide the driver with a consistent interface to these sub-devices the v4l2_subdev struct (v4l2-subdev.h) was created.

Each sub-device driver must have a v4l2_subdev struct. This struct can be stand-alone for simple sub-devices or it might be embedded in a larger struct if more state information needs to be stored. Usually there is a low-level device struct (e.g. i2c_client) that contains the device data as setup by the kernel. It is recommended to store that pointer in the private data of v4l2_subdev using v4l2_set_subdevdata(). That makes it easy to go from a v4l2_subdev to the actual low-level bus-specific device data.

You also need a way to go from the low-level struct to v4l2_subdev. For the common i2c_client struct the i2c_set_clientdata() call is used to store a v4l2_subdev pointer, for other busses you may have to use other methods.

From the bridge driver perspective you load the sub-device module and somehow obtain the v4l2_subdev pointer. For i2c devices this is easy: you call i2c_get_clientdata(). For other busses something similar needs to be done. Helper functions exists for sub-devices on an I2C bus that do most of this tricky work for you.

Each v4l2_subdev contains function pointers that sub-device drivers can implement (or leave NULL if it is not applicable). Since sub-devices can do so many different things and you do not want to end up with a huge ops struct of which only a handful of ops are commonly implemented, the function pointers are sorted according to category and each category has its own ops struct.

The top-level ops struct contains pointers to the category ops structs, which may be NULL if the subdev driver does not support anything from that category.

It looks like this:

```
struct v4l2_subdev_core_ops {
        int (*g_chip_ident)(struct v4l2_subdev *sd, struct v4l2_dbg_chip_ident
*chip);
        int (*log_status)(struct v4l2_subdev *sd);
        int (*init)(struct v4l2_subdev *sd, u32 val);
        ...
};

struct v4l2_subdev_tuner_ops {
        ...
};

struct v4l2_subdev_audio_ops {
```

```
        ...
};

struct v4l2_subdev_video_ops {
        ...
};

struct v4l2_subdev_ops {
        const struct v4l2_subdev_core_ops  *core;
        const struct v4l2_subdev_tuner_ops *tuner;
        const struct v4l2_subdev_audio_ops *audio;
        const struct v4l2_subdev_video_ops *video;
};
```

The core ops are common to all subdevs, the other categories are implemented depending on the sub-device. E.g. a video device is unlikely to support the audio ops and vice versa.

This setup limits the number of function pointers while still making it easy to add new ops and categories.

A sub-device driver initializes the v4l2_subdev struct using:

```
        v4l2_subdev_init(sd, &ops);
```

Afterwards you need to initialize subdev->name with a unique name and set the module owner. This is done for you if you use the i2c helper functions.

A device (bridge) driver needs to register the v4l2_subdev with the v4l2_device:

```
        int err = v4l2_device_register_subdev(v4l2_dev, sd);
```

This can fail if the subdev module disappeared before it could be registered. After this function was called successfully the subdev->dev field points to the v4l2_device.

You can unregister a sub-device using:

```
        v4l2_device_unregister_subdev(sd);
```

Afterwards the subdev module can be unloaded and sd->dev == NULL.

You can call an ops function either directly:

```
        err = sd->ops->core->g_chip_ident(sd, &chip);
```

but it is better and easier to use this macro:

```
        err = v4l2_subdev_call(sd, core, g_chip_ident, &chip);
```

The macro will to the right NULL pointer checks and returns -ENODEV if subdev is NULL, -ENOIOCTLCMD if either subdev->core or subdev->core->g_chip_ident is NULL, or the actual result of the subdev->ops->core->g_chip_ident ops.

It is also possible to call all or a subset of the sub-devices:

```
        v4l2_device_call_all(v4l2_dev, 0, core, g_chip_ident, &chip);
```

Any subdev that does not support this ops is skipped and error results are ignored. If you want to check for errors use this:

```
        err = v4l2_device_call_until_err(v4l2_dev, 0, core, g_chip_ident,
&chip);
```

Any error except -ENOIOCTLCMD will exit the loop with that error. If no errors (except -ENOIOCTLCMD) occured, then 0 is returned.

The second argument to both calls is a group ID. If 0, then all subdevs are called. If non-zero, then only those whose group ID match that value will be called. Before a bridge driver registers a subdev it can set sd->grp_id to whatever value it wants (it's 0 by default). This value is owned by the bridge driver and the sub-device driver will never modify or use it.

The group ID gives the bridge driver more control how callbacks are called. For example, there may be multiple audio chips on a board, each capable of changing the volume. But usually only one will actually be used when the user want to change the volume. You can set the group ID for that subdev to e.g. AUDIO_CONTROLLER and specify that as the group ID value when calling v4l2_device_call_all(). That ensures that it will only go to the subdev that needs it.

If the sub-device needs to notify its v4l2_device parent of an event, then it can call v4l2_subdev_notify(sd, notification, arg). This macro checks whether there is a notify() callback defined and returns -ENODEV if not. Otherwise the result of the notify() call is returned.

The advantage of using v4l2_subdev is that it is a generic struct and does not contain any knowledge about the underlying hardware. So a driver might contain several subdevs that use an I2C bus, but also a subdev that is controlled through GPIO pins. This distinction is only relevant when setting up the device, but once the subdev is registered it is completely transparent.

I2C sub-device drivers

Since these drivers are so common, special helper functions are available to ease the use of these drivers (v4l2-common.h).

The recommended method of adding v4l2_subdev support to an I2C driver is to embed the v4l2_subdev struct into the state struct that is created for each I2C device instance. Very simple devices have no state struct and in that case you can just create a v4l2_subdev directly.

A typical state struct would look like this (where 'chipname' is replaced by the name of the chip):

```
struct chipname_state {
        struct v4l2_subdev sd;
        ...  /* additional state fields */
};
```

Initialize the v4l2_subdev struct as follows:

```
        v4l2_i2c_subdev_init(&state->sd, client, subdev_ops);
```

This function will fill in all the fields of v4l2_subdev and ensure that the v4l2_subdev and i2c_client both point to one another.

You should also add a helper inline function to go from a v4l2_subdev pointer to a chipname_state struct:

```
static inline struct chipname_state *to_state(struct v4l2_subdev *sd)
{
        return container_of(sd, struct chipname_state, sd);
}
```

Use this to go from the v4l2_subdev struct to the i2c_client struct:

```
        struct i2c_client *client = v4l2_get_subdevdata(sd);
```

And this to go from an i2c_client to a v4l2_subdev struct:

```
        struct v4l2_subdev *sd = i2c_get_clientdata(client);
```

Make sure to call v4l2_device_unregister_subdev(sd) when the remove() callback is called. This will unregister the sub-device from the bridge driver. It is safe to call this even if the sub-device was never registered.

You need to do this because when the bridge driver destroys the i2c adapter the remove() callbacks are called of the i2c devices on that adapter. After that the corresponding v4l2_subdev structures are invalid, so they have to be unregistered first. Calling v4l2_device_unregister_subdev(sd) from the remove() callback ensures that this is always done correctly.

The bridge driver also has some helper functions it can use:

```
struct v4l2_subdev *sd = v4l2_i2c_new_subdev(v4l2_dev, adapter,
               "module_foo", "chipid", 0x36, NULL);
```

This loads the given module (can be NULL if no module needs to be loaded) and calls i2c_new_device() with the given i2c_adapter and chip/address arguments. If all goes well, then it registers the subdev with the v4l2_device.

You can also use the last argument of v4l2_i2c_new_subdev() to pass an array of possible I2C addresses that it should probe. These probe addresses are only used if the previous argument is 0. A non-zero argument means that you know the exact i2c address so in that case no probing will take place.

Both functions return NULL if something went wrong.

Note that the chipid you pass to v4l2_i2c_new_subdev() is usually the same as the module name. It allows you to specify a chip variant, e.g. "saa7114" or "saa7115". In general though the i2c driver autodetects this. The use of chipid is something that needs to be looked at more closely at a later date. It differs between i2c drivers and as such can be confusing.

To see which chip variants are supported you can look in the i2c driver code for the i2c_device_id table. This lists all the possibilities.

There are two more helper functions:

v4l2_i2c_new_subdev_cfg: this function adds new irq and platform_data arguments and has both 'addr' and 'probed_addrs' arguments: if addr is not 0 then that will be used (non-probing variant), otherwise the probed_addrs are probed.

For example: this will probe for address 0x10:

```
struct v4l2_subdev *sd = v4l2_i2c_new_subdev_cfg(v4l2_dev, adapter,
               "module_foo", "chipid", 0, NULL, 0, I2C_ADDRS(0x10));
```

v4l2_i2c_new_subdev_board uses an i2c_board_info struct which is passed to the i2c driver and replaces the irq, platform_data and addr arguments.

If the subdev supports the s_config core ops, then that op is called with the irq and platform_data arguments after the subdev was setup. The older v4l2_i2c_new_(probed_)subdev functions will call s_config as well, but with irq set to 0 and platform_data set to NULL.

struct video_device

The actual device nodes in the /dev directory are created using the video_device struct (v4l2-dev.h). This struct can either be allocated dynamically or embedded in a larger struct.

To allocate it dynamically use:

```
        struct video_device *vdev = video_device_alloc();

        if (vdev == NULL)
                return -ENOMEM;

        vdev->release = video_device_release;
```

If you embed it in a larger struct, then you must set the release() callback to your own function:

```
        struct video_device *vdev = &my_vdev->vdev;

        vdev->release = my_vdev_release;
```

The release callback must be set and it is called when the last user of the video device exits.

The default video_device_release() callback just calls kfree to free the allocated memory.

You should also set these fields:

- v4l2_dev: set to the v4l2_device parent device.
- name: set to something descriptive and unique.

- fops: set to the v4l2_file_operations struct.
- ioctl_ops: if you use the v4l2_ioctl_ops to simplify ioctl maintenance
  (highly recommended to use this and it might become compulsory in the
  future!), then set this to your v4l2_ioctl_ops struct.
- parent: you only set this if v4l2_device was registered with NULL as
  the parent device struct. This only happens in cases where one hardware
  device has multiple PCI devices that all share the same v4l2_device core.

  The cx88 driver is an example of this: one core v4l2_device struct, but
  it is used by both an raw video PCI device (cx8800) and a MPEG PCI device
  (cx8802). Since the v4l2_device cannot be associated with a particular
  PCI device it is setup without a parent device. But when the struct
  video_device is setup you do know which parent PCI device to use.

If you use v4l2_ioctl_ops, then you should set either .unlocked_ioctl or
.ioctl to video_ioctl2 in your v4l2_file_operations struct.

The v4l2_file_operations struct is a subset of file_operations. The main
difference is that the inode argument is omitted since it is never used.

video_device registration

Next you register the video device: this will create the character device
for you.

```
	err = video_register_device(vdev, VFL_TYPE_GRABBER, -1);
	if (err) {
		video_device_release(vdev); /* or kfree(my_vdev); */
		return err;
	}
```

Which device is registered depends on the type argument. The following
types exist:

VFL_TYPE_GRABBER: videoX for video input/output devices
VFL_TYPE_VBI: vbiX for vertical blank data (i.e. closed captions, teletext)
VFL_TYPE_RADIO: radioX for radio tuners
VFL_TYPE_VTX: vtxX for teletext devices (deprecated, don't use)

The last argument gives you a certain amount of control over the device
device node number used (i.e. the X in videoX). Normally you will pass -1
to let the v4l2 framework pick the first free number. But sometimes users
want to select a specific node number. It is common that drivers allow
the user to select a specific device node number through a driver module
option. That number is then passed to this function and video_register_device
will attempt to select that device node number. If that number was already
in use, then the next free device node number will be selected and it
will send a warning to the kernel log.

Another use-case is if a driver creates many devices. In that case it can
be useful to place different video devices in separate ranges. For example,
video capture devices start at 0, video output devices start at 16.
So you can use the last argument to specify a minimum device node number
and the v4l2 framework will try to pick the first free number that is equal

or higher to what you passed. If that fails, then it will just pick the
first free number.

Since in this case you do not care about a warning about not being able
to select the specified device node number, you can call the function
video_register_device_no_warn() instead.

Whenever a device node is created some attributes are also created for you.
If you look in /sys/class/video4linux you see the devices. Go into e.g.
video0 and you will see 'name' and 'index' attributes. The 'name' attribute
is the 'name' field of the video_device struct.

The 'index' attribute is the index of the device node: for each call to
video_register_device() the index is just increased by 1. The first video
device node you register always starts with index 0.

Users can setup udev rules that utilize the index attribute to make fancy
device names (e.g. 'mpegX' for MPEG video capture device nodes).

After the device was successfully registered, then you can use these fields:

- vfl_type: the device type passed to video_register_device.
- minor: the assigned device minor number.
- num: the device node number (i.e. the X in videoX).
- index: the device index number.

If the registration failed, then you need to call video_device_release()
to free the allocated video_device struct, or free your own struct if the
video_device was embedded in it. The vdev->release() callback will never
be called if the registration failed, nor should you ever attempt to
unregister the device if the registration failed.

video_device cleanup

When the video device nodes have to be removed, either during the unload
of the driver or because the USB device was disconnected, then you should
unregister them:

```
        video_unregister_device(vdev);
```

This will remove the device nodes from sysfs (causing udev to remove them
from /dev).

After video_unregister_device() returns no new opens can be done. However,
in the case of USB devices some application might still have one of these
device nodes open. So after the unregister all file operations will return
an error as well, except for the ioctl and unlocked_ioctl file operations:
those will still be passed on since some buffer ioctls may still be needed.

When the last user of the video device node exits, then the vdev->release()
callback is called and you can do the final cleanup there.

video_device helper functions

There are a few useful helper functions:

- file/video_device private data

You can set/get driver private data in the video_device struct using:

```
void *video_get_drvdata(struct video_device *vdev);
void video_set_drvdata(struct video_device *vdev, void *data);
```

Note that you can safely call video_set_drvdata() before calling video_register_device().

And this function:

```
struct video_device *video_devdata(struct file *file);
```

returns the video_device belonging to the file struct.

The video_drvdata function combines video_get_drvdata with video_devdata:

```
void *video_drvdata(struct file *file);
```

You can go from a video_device struct to the v4l2_device struct using:

```
struct v4l2_device *v4l2_dev = vdev->v4l2_dev;
```

- Device node name

The video_device node kernel name can be retrieved using

```
const char *video_device_node_name(struct video_device *vdev);
```

The name is used as a hint by userspace tools such as udev. The function should be used where possible instead of accessing the video_device::num and video_device::minor fields.

video buffer helper functions

The v4l2 core API provides a set of standard methods (called "videobuf") for dealing with video buffers. Those methods allow a driver to implement read(), mmap() and overlay() in a consistent way. There are currently methods for using video buffers on devices that supports DMA with scatter/gather method (videobuf-dma-sg), DMA with linear access (videobuf-dma-contig), and vmalloced buffers, mostly used on USB drivers (videobuf-vmalloc).

Please see Documentation/video4linux/videobuf for more information on how to use the videobuf layer.

struct v4l2_fh

struct v4l2_fh provides a way to easily keep file handle specific data that is used by the V4L2 framework. Using v4l2_fh is optional for drivers.

The users of v4l2_fh (in the V4L2 framework, not the driver) know whether a driver uses v4l2_fh as its file->private_data pointer by testing the V4L2_FL_USES_V4L2_FH bit in video_device->flags.

Useful functions:

- v4l2_fh_init()

  Initialise the file handle. This *MUST* be performed in the driver's v4l2_file_operations->open() handler.

- v4l2_fh_add()

  Add a v4l2_fh to video_device file handle list. May be called after initialising the file handle.

- v4l2_fh_del()

  Unassociate the file handle from video_device(). The file handle exit function may now be called.

- v4l2_fh_exit()

  Uninitialise the file handle. After uninitialisation the v4l2_fh memory can be freed.

struct v4l2_fh is allocated as a part of the driver's own file handle structure and is set to file->private_data in the driver's open function by the driver. Drivers can extract their own file handle structure by using the container_of macro. Example:

```
struct my_fh {
	int blah;
	struct v4l2_fh fh;
};

...

int my_open(struct file *file)
{
	struct my_fh *my_fh;
	struct video_device *vfd;
	int ret;

	...

	ret = v4l2_fh_init(&my_fh->fh, vfd);
	if (ret)
		return ret;

	v4l2_fh_add(&my_fh->fh);
```

```
        file->private_data = &my_fh->fh;

        ...
}

int my_release(struct file *file)
{
        struct v4l2_fh *fh = file->private_data;
        struct my_fh *my_fh = container_of(fh, struct my_fh, fh);

        ...
}
```

V4L2 events

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.

Useful functions:

- v4l2_event_alloc()

  To use events, the driver must allocate events for the file handle. By calling the function more than once, the driver may assure that at least n events in total have been allocated. The function may not be called in atomic context.

- v4l2_event_queue()

  Queue events to video device. The driver's only responsibility is to fill in the type and the data fields. The other fields will be filled in by V4L2.

- v4l2_event_subscribe()

  The video_device->ioctl_ops->vidioc_subscribe_event must check the driver is able to produce events with specified event id. Then it calls v4l2_event_subscribe() to subscribe the event.

- v4l2_event_unsubscribe()

  vidioc_unsubscribe_event in struct v4l2_ioctl_ops. A driver may use v4l2_event_unsubscribe() directly unless it wants to be involved in unsubscription process.

  The special type V4L2_EVENT_ALL may be used to unsubscribe all events. The drivers may want to handle this in a special way.

- v4l2_event_pending()

  Returns the number of pending events. Useful when implementing poll.

Drivers do not initialise events directly. The events are initialised through v4l2_fh_init() if video_device->ioctl_ops->vidioc_subscribe_event is non-NULL. This *MUST* be performed in the driver's v4l2_file_operations->open() handler.

Events are delivered to user space through the poll system call. The driver can use v4l2_fh->events->wait wait_queue_head_t as the argument for poll_wait().

There are standard and private events. New standard events must use the smallest available event type. The drivers must allocate their events from their own class starting from class base. Class base is V4L2_EVENT_PRIVATE_START + n * 1000 where n is the lowest available number. The first event type in the class is reserved for future use, so the first available event type is 'class base + 1'.

An example on how the V4L2 events may be used can be found in the OMAP 3 ISP driver available at <URL:http://gitorious.org/omap3camera> as of writing this.