V4L2 Controls

Introduction

The V4L2 control API seems simple enough, but quickly becomes very hard to implement correctly in drivers. But much of the code needed to handle controls is actually not driver specific and can be moved to the V4L core framework.

After all, the only part that a driver developer is interested in is:

1. How do I add a control?
2. How do I set the control's value? (i.e. s_ctrl)

And occasionally:

3. How do I get the control's value? (i.e. g_volatile_ctrl)
4. How do I validate the user's proposed control value? (i.e. try_ctrl)

All the rest is something that can be done centrally.

The control framework was created in order to implement all the rules of the V4L2 specification with respect to controls in a central place. And to make life as easy as possible for the driver developer.

Note that the control framework relies on the presence of a struct :c:type:`v4l2_device` for V4L2 drivers and struct v4l2_subdev for sub-device drivers.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst`, **line 29**); ***backlink***

Unknown interpreted text role "c:type".

Objects in the framework

There are two main objects:

The :c:type:`v4l2_ctrl` object describes the control properties and keeps track of the control's value (both the current value and the proposed new value).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst`, **line 39**); ***backlink***

Unknown interpreted text role "c:type".

:c:type:`v4l2_ctrl_handler` is the object that keeps track of controls. It maintains a list of v4l2_ctrl objects that it owns and another list of references to controls, possibly to controls owned by other handlers.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst`, **line 43**); ***backlink***

Unknown interpreted text role "c:type".

Basic usage for V4L2 and sub-device drivers

1. Prepare the driver:

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

1.1) Add the handler to your driver's top-level struct:

For V4L2 drivers:

```
struct foo_dev {
        ...
        struct v4l2_device v4l2_dev;
        ...
        struct v4l2_ctrl_handler ctrl_handler;
        ...
};
```

For sub-device drivers:

```
struct foo_dev {
        ...
        struct v4l2_subdev sd;
        ...
        struct v4l2_ctrl_handler ctrl_handler;
        ...
};
```

1.2) Initialize the handler:

```
v4l2_ctrl_handler_init(&foo->ctrl_handler, nr_of_controls);
```

The second argument is a hint telling the function how many controls this handler is expected to handle. It will allocate a hashtable based on this information. It is a hint only.

1.3) Hook the control handler into the driver:

For V4L2 drivers:

```
foo->v4l2_dev.ctrl_handler = &foo->ctrl_handler;
```

For sub-device drivers:

```
foo->sd.ctrl_handler = &foo->ctrl_handler;
```

1.4) Clean up the handler at the end:

```
v4l2_ctrl_handler_free(&foo->ctrl_handler);
```

2. Add controls:

You add non-menu controls by calling :c:func:`v4l2_ctrl_new_std`:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst`, line 116); *backlink*

Unknown interpreted text role "c:func".

```
struct v4l2_ctrl *v4l2_ctrl_new_std(struct v4l2_ctrl_handler *hdl,
                const struct v4l2_ctrl_ops *ops,
                u32 id, s32 min, s32 max, u32 step, s32 def);
```

Menu and integer menu controls are added by calling :c:func:`v4l2_ctrl_new_std_menu`:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst`, line 124); *backlink*

Unknown interpreted text role "c:func".

```
struct v4l2_ctrl *v4l2_ctrl_new_std_menu(struct v4l2_ctrl_handler *hdl,
                const struct v4l2_ctrl_ops *ops,
                u32 id, s32 max, s32 skip_mask, s32 def);
```

Menu controls with a driver specific menu are added by calling :c:func:`v4l2_ctrl_new_std_menu_items`:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst`, line 133); *backlink*

Unknown interpreted text role "c:func".

```
struct v4l2_ctrl *v4l2_ctrl_new_std_menu_items(
                struct v4l2_ctrl_handler *hdl,
                const struct v4l2_ctrl_ops *ops, u32 id, s32 max,
                s32 skip_mask, s32 def, const char * const *qmenu);
```

Standard compound controls can be added by calling :c:func:`v4l2_ctrl_new_std_compound`:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api] [media]v4l2-controls.rst, line 143); *backlink*

Unknown interpreted text role "c:func".

```
struct v4l2_ctrl *v4l2_ctrl_new_std_compound(struct v4l2_ctrl_handler *hdl,
                const struct v4l2_ctrl_ops *ops, u32 id,
                const union v4l2_ctrl_ptr p_def);
```

Integer menu controls with a driver specific menu can be added by calling :c:func:`v4l2_ctrl_new_int_menu`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst, line 152); *backlink*

Unknown interpreted text role "c:func".

```
struct v4l2_ctrl *v4l2_ctrl_new_int_menu(struct v4l2_ctrl_handler *hdl,
                const struct v4l2_ctrl_ops *ops,
                u32 id, s32 max, s32 def, const s64 *qmenu_int);
```

These functions are typically called right after the :c:func:`v4l2_ctrl_handler_init`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst, line 161); *backlink*

Unknown interpreted text role "c:func".

```
static const s64 exp_bias_qmenu[] = {
       -2, -1, 0, 1, 2
};
static const char * const test_pattern[] = {
        "Disabled",
        "Vertical Bars",
        "Solid Black",
        "Solid White",
};

v4l2_ctrl_handler_init(&foo->ctrl_handler, nr_of_controls);
v4l2_ctrl_new_std(&foo->ctrl_handler, &foo_ctrl_ops,
                V4L2_CID_BRIGHTNESS, 0, 255, 1, 128);
v4l2_ctrl_new_std(&foo->ctrl_handler, &foo_ctrl_ops,
                V4L2_CID_CONTRAST, 0, 255, 1, 128);
v4l2_ctrl_new_std_menu(&foo->ctrl_handler, &foo_ctrl_ops,
                V4L2_CID_POWER_LINE_FREQUENCY,
                V4L2_CID_POWER_LINE_FREQUENCY_60HZ, 0,
                V4L2_CID_POWER_LINE_FREQUENCY_DISABLED);
v4l2_ctrl_new_int_menu(&foo->ctrl_handler, &foo_ctrl_ops,
                V4L2_CID_EXPOSURE_BIAS,
                ARRAY_SIZE(exp_bias_qmenu) - 1,
                ARRAY_SIZE(exp_bias_qmenu) / 2 - 1,
                exp_bias_qmenu);
v4l2_ctrl_new_std_menu_items(&foo->ctrl_handler, &foo_ctrl_ops,
                V4L2_CID_TEST_PATTERN, ARRAY_SIZE(test_pattern) - 1, 0,
                0, test_pattern);
...
if (foo->ctrl_handler.error) {
        int err = foo->ctrl_handler.error;

        v4l2_ctrl_handler_free(&foo->ctrl_handler);
        return err;
}
```

The :c:func:`v4l2_ctrl_new_std` function returns the v4l2_ctrl pointer to the new control, but if you do not need to access the pointer outside the control ops, then there is no need to store it.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst, line 201); *backlink*

Unknown interpreted text role "c:func".

The :c:func:`v4l2_ctrl_new_std` function will fill in most fields based on the control ID except for the min, max, step and default values. These are passed in the last four arguments. These values are driver specific while control attributes like type, name, flags are all global. The control's current value will be set to the default value.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst`, line 205); *backlink*

Unknown interpreted text role "c:func".

The :c:func:`v4l2_ctrl_new_std_menu` function is very similar but it is used for menu controls. There is no min argument since that is always 0 for menu controls, and instead of a step there is a skip_mask argument: if bit X is 1, then menu item X is skipped.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst`, line 211); *backlink*

Unknown interpreted text role "c:func".

The :c:func:`v4l2_ctrl_new_int_menu` function creates a new standard integer menu control with driver-specific items in the menu. It differs from v4l2_ctrl_new_std_menu in that it doesn't have the mask argument and takes as the last argument an array of signed 64-bit integers that form an exact menu item list.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst`, line 216); *backlink*

Unknown interpreted text role "c:func".

The :c:func:`v4l2_ctrl_new_std_menu_items` function is very similar to v4l2_ctrl_new_std_menu but takes an extra parameter qmenu, which is the driver specific menu for an otherwise standard menu control. A good example for this control is the test pattern control for capture/display/sensors devices that have the capability to generate test patterns. These test patterns are hardware specific, so the contents of the menu will vary from device to device.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst`, line 222); *backlink*

Unknown interpreted text role "c:func".

Note that if something fails, the function will return NULL or an error and set ctrl_handler->error to the error code. If ctrl_handler->error was already set, then it will just return and do nothing. This is also true for v4l2_ctrl_handler_init if it cannot allocate the internal data structure.

This makes it easy to init the handler and just add all controls and only check the error code at the end. Saves a lot of repetitive error checking.

It is recommended to add controls in ascending control ID order: it will be a bit faster that way.

3. Optionally force initial control setup:

```
v4l2_ctrl_handler_setup(&foo->ctrl_handler);
```

This will call s_ctrl for all controls unconditionally. Effectively this initializes the hardware to the default control values. It is recommended that you do this as this ensures that both the internal data structures and the hardware are in sync.

4. Finally: implement the :c:type:`v4l2_ctrl_ops`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst`, line 252); *backlink*

Unknown interpreted text role "c:type".

```
static const struct v4l2_ctrl_ops foo_ctrl_ops = {
        .s_ctrl = foo_s_ctrl,
};
```

Usually all you need is s_ctrl:

```
static int foo_s_ctrl(struct v4l2_ctrl *ctrl)
{
        struct foo *state = container_of(ctrl->handler, struct foo, ctrl_handler);

        switch (ctrl->id) {
        case V4L2_CID_BRIGHTNESS:
                write_reg(0x123, ctrl->val);
                break;
        case V4L2_CID_CONTRAST:
                write_reg(0x456, ctrl->val);
                break;
        }
        return 0;
}
```

The control ops are called with the v4l2_ctrl pointer as argument. The new control value has already been validated, so all you need to do is to actually update the hardware registers.

You're done! And this is sufficient for most of the drivers we have. No need to do any validation of control values, or implement QUERYCTRL, QUERY_EXT_CTRL and QUERYMENU. And G/S_CTRL as well as G/TRY/S_EXT_CTRLS are automatically supported.

> **Note**
>
> The remainder sections deal with more advanced controls topics and scenarios. In practice the basic usage as described above is sufficient for most drivers.

Inheriting Sub-device Controls

When a sub-device is registered with a V4L2 driver by calling v4l2_device_register_subdev() and the ctrl_handler fields of both v4l2_subdev and v4l2_device are set, then the controls of the subdev will become automatically available in the V4L2 driver as well. If the subdev driver contains controls that already exist in the V4L2 driver, then those will be skipped (so a V4L2 driver can always override a subdev control).

What happens here is that v4l2_device_register_subdev() calls v4l2_ctrl_add_handler() adding the controls of the subdev to the controls of v4l2_device.

Accessing Control Values

The following union is used inside the control framework to access control values:

```
union v4l2_ctrl_ptr {
        s32 *p_s32;
        s64 *p_s64;
        char *p_char;
        void *p;
};
```

The v4l2_ctrl struct contains these fields that can be used to access both current and new values:

```
s32 val;
struct {
        s32 val;
} cur;


union v4l2_ctrl_ptr p_new;
union v4l2_ctrl_ptr p_cur;
```

If the control has a simple s32 type, then:

```
&ctrl->val == ctrl->p_new.p_s32
&ctrl->cur.val == ctrl->p_cur.p_s32
```

For all other types use ctrl->p_cur.p<something>. Basically the val and cur.val fields can be considered an alias since these are used so often.

Within the control ops you can freely use these. The val and cur.val speak for themselves. The p_char pointers point to character buffers of length ctrl->maximum + 1, and are always 0-terminated.

Unless the control is marked volatile the p_cur field points to the current cached control value. When you create a new control this value is made identical to the default value. After calling v4l2_ctrl_handler_setup() this value is passed to the hardware. It is generally a good idea to call this function.

Whenever a new value is set that new value is automatically cached. This means that most drivers do not need to implement the g_volatile_ctrl() op. The exception is for controls that return a volatile register such as a signal strength read-out that changes continuously. In that case you will need to implement g_volatile_ctrl like this:

```
static int foo_g_volatile_ctrl(struct v4l2_ctrl *ctrl)
{
        switch (ctrl->id) {
        case V4L2_CID_BRIGHTNESS:
                ctrl->val = read_reg(0x123);
                break;
        }
}
```

Note that you use the 'new value' union as well in g_volatile_ctrl. In general controls that need to implement g_volatile_ctrl are read-only controls. If they are not, a V4L2_EVENT_CTRL_CH_VALUE will not be generated when the control changes.

To mark a control as volatile you have to set V4L2_CTRL_FLAG_VOLATILE:

```
ctrl = v4l2_ctrl_new_std(&sd->ctrl_handler, ...);
if (ctrl)
        ctrl->flags |= V4L2_CTRL_FLAG_VOLATILE;
```

For try/s_ctrl the new values (i.e. as passed by the user) are filled in and you can modify them in try_ctrl or set them in s_ctrl. The 'cur' union contains the current value, which you can use (but not change!) as well.

If s_ctrl returns 0 (OK), then the control framework will copy the new final values to the 'cur' union.

While in g_volatile/s/try_ctrl you can access the value of all controls owned by the same handler since the handler's lock is held. If you need to access the value of controls owned by other handlers, then you have to be very careful not to introduce deadlocks.

Outside of the control ops you have to go through to helper functions to get or set a single control value safely in your driver:

```
s32 v4l2_ctrl_g_ctrl(struct v4l2_ctrl *ctrl);
int v4l2_ctrl_s_ctrl(struct v4l2_ctrl *ctrl, s32 val);
```

These functions go through the control framework just as VIDIOC_G/S_CTRL ioctls do. Don't use these inside the control ops g_volatile/s/try_ctrl, though, that will result in a deadlock since these helpers lock the handler as well.

You can also take the handler lock yourself:

```
mutex_lock(&state->ctrl_handler.lock);
pr_info("String value is '%s'\n", ctrl1->p_cur.p_char);
pr_info("Integer value is '%s'\n", ctrl2->cur.val);
mutex_unlock(&state->ctrl_handler.lock);
```

Menu Controls

The v4l2_ctrl struct contains this union:

```
union {
        u32 step;
        u32 menu_skip_mask;
};
```

For menu controls menu_skip_mask is used. What it does is that it allows you to easily exclude certain menu items. This is used in the VIDIOC_QUERYMENU implementation where you can return -EINVAL if a certain menu item is not present. Note that VIDIOC_QUERYCTRL always returns a step value of 1 for menu controls.

A good example is the MPEG Audio Layer II Bitrate menu control where the menu is a list of standardized possible bitrates. But in practice hardware implementations will only support a subset of those. By setting the skip mask you can tell the framework which menu items should be skipped. Setting it to 0 means that all menu items are supported.

You set this mask either through the v4l2_ctrl_config struct for a custom control, or by calling v4l2_ctrl_new_std_menu().

Custom Controls

Driver specific controls can be created using v4l2_ctrl_new_custom():

```
static const struct v4l2_ctrl_config ctrl_filter = {
        .ops = &ctrl_custom_ops,
        .id = V4L2_CID_MPEG_CX2341X_VIDEO_SPATIAL_FILTER,
        .name = "Spatial Filter",
        .type = V4L2_CTRL_TYPE_INTEGER,
        .flags = V4L2_CTRL_FLAG_SLIDER,
        .max = 15,
        .step = 1,
};

ctrl = v4l2_ctrl_new_custom(&foo->ctrl_handler, &ctrl_filter, NULL);
```

The last argument is the priv pointer which can be set to driver-specific private data.

The v4l2_ctrl_config struct also has a field to set the is_private flag.

If the name field is not set, then the framework will assume this is a standard control and will fill in the name, type and flags fields accordingly.

Active and Grabbed Controls

If you get more complex relationships between controls, then you may have to activate and deactivate controls. For example, if the Chroma AGC control is on, then the Chroma Gain control is inactive. That is, you may set it, but the value will not be used by the hardware as long as the automatic gain control is on. Typically user interfaces can disable such input fields.

You can set the 'active' status using v4l2_ctrl_activate(). By default all controls are active. Note that the framework does not check for this flag. It is meant purely for GUIs. The function is typically called from within s_ctrl.

The other flag is the 'grabbed' flag. A grabbed control means that you cannot change it because it is in use by some resource. Typical examples are MPEG bitrate controls that cannot be changed while capturing is in progress.

If a control is set to 'grabbed' using v4l2_ctrl_grab(), then the framework will return -EBUSY if an attempt is made to set this control. The v4l2_ctrl_grab() function is typically called from the driver when it starts or stops streaming.

Control Clusters

By default all controls are independent from the others. But in more complex scenarios you can get dependencies from one control to another. In that case you need to 'cluster' them:

```
struct foo {
        struct v4l2_ctrl_handler ctrl_handler;
#define AUDIO_CL_VOLUME (0)
#define AUDIO_CL_MUTE   (1)
        struct v4l2_ctrl *audio_cluster[2];
        ...
};

state->audio_cluster[AUDIO_CL_VOLUME] =
        v4l2_ctrl_new_std(&state->ctrl_handler, ...);
state->audio_cluster[AUDIO_CL_MUTE] =
        v4l2_ctrl_new_std(&state->ctrl_handler, ...);
v4l2_ctrl_cluster(ARRAY_SIZE(state->audio_cluster), state->audio_cluster);
```

From now on whenever one or more of the controls belonging to the same cluster is set (or 'gotten', or 'tried'), only the control ops of the first control ('volume' in this example) is called. You effectively create a new composite control. Similar to how a 'struct' works in C.

So when s_ctrl is called with V4L2_CID_AUDIO_VOLUME as argument, you should set all two controls belonging to the audio_cluster:

```
static int foo_s_ctrl(struct v4l2_ctrl *ctrl)
{
        struct foo *state = container_of(ctrl->handler, struct foo, ctrl_handler);

        switch (ctrl->id) {
        case V4L2_CID_AUDIO_VOLUME: {
                struct v4l2_ctrl *mute = ctrl->cluster[AUDIO_CL_MUTE];

                write_reg(0x123, mute->val ? 0 : ctrl->val);
                break;
        }
        case V4L2_CID_CONTRAST:
                write_reg(0x456, ctrl->val);
                break;
        }
        return 0;
}
```

In the example above the following are equivalent for the VOLUME case:

```
ctrl == ctrl->cluster[AUDIO_CL_VOLUME] == state->audio_cluster[AUDIO_CL_VOLUME]
ctrl->cluster[AUDIO_CL_MUTE] == state->audio_cluster[AUDIO_CL_MUTE]
```

In practice using cluster arrays like this becomes very tiresome. So instead the following equivalent method is used:

```
struct {
        /* audio cluster */
        struct v4l2_ctrl *volume;
        struct v4l2_ctrl *mute;
};
```

The anonymous struct is used to clearly 'cluster' these two control pointers, but it serves no other purpose. The effect is the same as creating an array with two control pointers. So you can just do:

```
state->volume = v4l2_ctrl_new_std(&state->ctrl_handler, ...);
state->mute = v4l2_ctrl_new_std(&state->ctrl_handler, ...);
v4l2_ctrl_cluster(2, &state->volume);
```

And in foo_s_ctrl you can use these pointers directly: state->mute->val.

Note that controls in a cluster may be NULL. For example, if for some reason mute was never added (because the hardware doesn't support that particular feature), then mute will be NULL. So in that case we have a cluster of 2 controls, of which only 1 is actually instantiated. The only restriction is that the first control of the cluster must always be present, since that is the 'master' control of the cluster. The master control is the one that identifies the cluster and that provides the pointer to the v4l2_ctrl_ops struct that is used for that cluster.

Obviously, all controls in the cluster array must be initialized to either a valid control or to NULL.

In rare cases you might want to know which controls of a cluster actually were set explicitly by the user. For this you can check the 'is_new' flag of each control. For example, in the case of a volume/mute cluster the 'is_new' flag of the mute control would be set if the user called VIDIOC_S_CTRL for mute only. If the user would call VIDIOC_S_EXT_CTRLS for both mute and volume controls, then the 'is_new' flag would be 1 for both controls.

The 'is_new' flag is always 1 when called from v4l2_ctrl_handler_setup().

Handling autogain/gain-type Controls with Auto Clusters

A common type of control cluster is one that handles 'auto-foo/foo'-type controls. Typical examples are autogain/gain, autoexposure/exposure, autowhitebalance/red balance/blue balance. In all cases you have one control that determines whether another control is handled automatically by the hardware, or whether it is under manual control from the user.

If the cluster is in automatic mode, then the manual controls should be marked inactive and volatile. When the volatile controls are read the g_volatile_ctrl operation should return the value that the hardware's automatic mode set up automatically.

If the cluster is put in manual mode, then the manual controls should become active again and the volatile flag is cleared (so g_volatile_ctrl is no longer called while in manual mode). In addition just before switching to manual mode the current values as determined by the auto mode are copied as the new manual values.

Finally the V4L2_CTRL_FLAG_UPDATE should be set for the auto control since changing that control affects the control flags of the manual controls.

In order to simplify this a special variation of v4l2_ctrl_cluster was introduced:

```
void v4l2_ctrl_auto_cluster(unsigned ncontrols, struct v4l2_ctrl **controls,
                            u8 manual_val, bool set_volatile);
```

The first two arguments are identical to v4l2_ctrl_cluster. The third argument tells the framework which value switches the cluster into manual mode. The last argument will optionally set V4L2_CTRL_FLAG_VOLATILE for the non-auto controls. If it is false, then the manual controls are never volatile. You would typically use that if the hardware does not give you the option to read back to values as determined by the auto mode (e.g. if autogain is on, the hardware doesn't allow you to obtain the current gain value).

The first control of the cluster is assumed to be the 'auto' control.

Using this function will ensure that you don't need to handle all the complex flag and volatile handling.

VIDIOC_LOG_STATUS Support

This ioctl allow you to dump the current status of a driver to the kernel log. The v4l2_ctrl_handler_log_status(ctrl_handler, prefix) can be used to dump the value of the controls owned by the given handler to the log. You can supply a prefix as well. If the prefix didn't end with a space, then ': ' will be added for you.

Different Handlers for Different Video Nodes

Usually the V4L2 driver has just one control handler that is global for all video nodes. But you can also specify different control handlers for different video nodes. You can do that by manually setting the ctrl_handler field of struct video_device.

That is no problem if there are no subdevs involved but if there are, then you need to block the automatic merging of subdev controls to the global control handler. You do that by simply setting the ctrl_handler field in struct v4l2_device to NULL. Now v4l2_device_register_subdev() will no longer merge subdev controls.

After each subdev was added, you will then have to call v4l2_ctrl_add_handler manually to add the subdev's control handler (sd->ctrl_handler) to the desired control handler. This control handler may be specific to the video_device or for a subset of video_device's. For example: the radio device nodes only have audio controls, while the video and vbi device nodes share the same control handler for the audio and video controls.

If you want to have one handler (e.g. for a radio device node) have a subset of another handler (e.g. for a video device node), then you should first add the controls to the first handler, add the other controls to the second handler and finally add the first handler to the second. For example:

```
v4l2_ctrl_new_std(&radio_ctrl_handler, &radio_ops, V4L2_CID_AUDIO_VOLUME, ...);
v4l2_ctrl_new_std(&radio_ctrl_handler, &radio_ops, V4L2_CID_AUDIO_MUTE, ...);
v4l2_ctrl_new_std(&video_ctrl_handler, &video_ops, V4L2_CID_BRIGHTNESS, ...);
v4l2_ctrl_new_std(&video_ctrl_handler, &video_ops, V4L2_CID_CONTRAST, ...);
v4l2_ctrl_add_handler(&video_ctrl_handler, &radio_ctrl_handler, NULL);
```

The last argument to v4l2_ctrl_add_handler() is a filter function that allows you to filter which controls will be added. Set it to NULL if you want to add all controls.

Or you can add specific controls to a handler:

```
volume = v4l2_ctrl_new_std(&video_ctrl_handler, &ops, V4L2_CID_AUDIO_VOLUME, ...);
v4l2_ctrl_new_std(&video_ctrl_handler, &ops, V4L2_CID_BRIGHTNESS, ...);
v4l2_ctrl_new_std(&video_ctrl_handler, &ops, V4L2_CID_CONTRAST, ...);
```

What you should not do is make two identical controls for two handlers. For example:

```
v4l2_ctrl_new_std(&radio_ctrl_handler, &radio_ops, V4L2_CID_AUDIO_MUTE, ...);
v4l2_ctrl_new_std(&video_ctrl_handler, &video_ops, V4L2_CID_AUDIO_MUTE, ...);
```

This would be bad since muting the radio would not change the video mute control. The rule is to have one control for each hardware 'knob' that you can twiddle.

Finding Controls

Normally you have created the controls yourself and you can store the struct v4l2_ctrl pointer into your own struct.

But sometimes you need to find a control from another handler that you do not own. For example, if you have to find a volume control from a subdev.

You can do that by calling v4l2_ctrl_find:

```
struct v4l2_ctrl *volume;

volume = v4l2_ctrl_find(sd->ctrl_handler, V4L2_CID_AUDIO_VOLUME);
```

Since v4l2_ctrl_find will lock the handler you have to be careful where you use it. For example, this is not a good idea:

```
struct v4l2_ctrl_handler ctrl_handler;

v4l2_ctrl_new_std(&ctrl_handler, &video_ops, V4L2_CID_BRIGHTNESS, ...);
v4l2_ctrl_new_std(&ctrl_handler, &video_ops, V4L2_CID_CONTRAST, ...);
```

...and in video_ops.s_ctrl:

```
case V4L2_CID_BRIGHTNESS:
        contrast = v4l2_find_ctrl(&ctrl_handler, V4L2_CID_CONTRAST);
        ...
```

When s_ctrl is called by the framework the ctrl_handler.lock is already taken, so attempting to find another control from the same handler will deadlock.

It is recommended not to use this function from inside the control ops.

Preventing Controls inheritance

When one control handler is added to another using v4l2_ctrl_add_handler, then by default all controls from one are merged to the other. But a subdev might have low-level controls that make sense for some advanced embedded system, but not when it is used in consumer-level hardware. In that case you want to keep those low-level controls local to the subdev. You can do this by simply setting the 'is_private' flag of the control to 1:

```
static const struct v4l2_ctrl_config ctrl_private = {
        .ops = &ctrl_custom_ops,
        .id = V4L2_CID_...,
        .name = "Some Private Control",
        .type = V4L2_CTRL_TYPE_INTEGER,
        .max = 15,
        .step = 1,
        .is_private = 1,
};

ctrl = v4l2_ctrl_new_custom(&foo->ctrl_handler, &ctrl_private, NULL);
```

These controls will now be skipped when v4l2_ctrl_add_handler is called.

V4L2_CTRL_TYPE_CTRL_CLASS Controls

Controls of this type can be used by GUIs to get the name of the control class. A fully featured GUI can make a dialog with multiple tabs with each tab containing the controls belonging to a particular control class. The name of each tab can be found by querying a special control with ID <control class | 1>.

Drivers do not have to care about this. The framework will automatically add a control of this type whenever the first control belonging to a new control class is added.

Adding Notify Callbacks

Sometimes the platform or bridge driver needs to be notified when a control from a sub-device driver changes. You can set a notify callback by calling this function:

```
void v4l2_ctrl_notify(struct v4l2_ctrl *ctrl,
        void (*notify)(struct v4l2_ctrl *ctrl, void *priv), void *priv);
```

Whenever the give control changes value the notify callback will be called with a pointer to the control and the priv pointer that was passed with v4l2_ctrl_notify. Note that the control's handler lock is held when the notify function is called.

There can be only one notify function per control handler. Any attempt to set another notify function will cause a WARN_ON.

v4l2_ctrl functions and data structures

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\media\[linux-master][Documentation][driver-api][media]v4l2-controls.rst`, line 823)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/media/v4l2-ctrls.h
```