V4L2 M2M as the driver framework for Video Processing IP

Karthik Poduval /Amazon Lab126

#ossuminmit
What is V4L2 M2M?

- A driver framework for memory to memory devices
- Memory to memory devices take data from memory do some processing and write out processed data back to memory
- Different from traditional V4L2 devices which are either
  - Capture
  - Output
- V4L2 M2M will have both output and capture
- V4L2 M2M supports multiple contexts which traditional V4L2 devices do not (usually)
Memory to Memory Device Logical View

DDR → M2M Device
Read

M2M Device → DDR
Write
V4L2 Output Device

- **DDR**
- **Output Device**
- **Read**
V4L2 M2M Device

DDDR

Read

Write

M2M Device
Typical V4L2 Application Workflow

- **Application**
  - **V4L2 Kernel API**
    - **V4L2 Core**
    - **VB2 Core**
  - Driver
  - Device

- **buffers**
- **setFormat**
- **allocateBuffers**
- **startStreaming**

- **setFormat**
- **allocateBuffers**
- **startStreaming**
V4L2 M2M Application Workflow

Application

V4L2 Kernel API

Driver

Device
• Individual contexts queue buffers on their output and capture VB2 queues

• From there it makes it to M2M per context ready queue when VB2 call driver’s queue_buffer callback

• Once required number of output and capture buffers are ready, move context to job queue

• The device_run callback of driver is invoked

• Device run pulls necessary number of buffers from output and capture ready queue for the current context

• Once device finishes processing the job, call vb2_buffer done to return buffer to application and m2m_job_finish to remove context from job queue.
An Example M2M Scaler Device

- A virtual M2M scaler device
- Device is built using STB Image Resize Library
- Device is virtual and emulated using QEMU
- Using yocto and Linux kernel 5.15 we build a device driver for the virtual device
- Using libcamera C++ Library we also demonstrate a simple downscaling example
Virtual memory-2-memory Scaler

This is the data sheet for a QEMU based Virtual memory-2-memory scaler. The scaler can perform upscaling as well as downscaling. No ratios need to be supplied instead the actual input and output sizes are supplied. The scaler only supports B&I image format and both input and output must be in this format.

Register Map

**Offset 0x00 - Input Configuration 1**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:15</td>
<td>Input Width</td>
</tr>
<tr>
<td>16:31</td>
<td>Input Height</td>
</tr>
</tbody>
</table>

**Offset 0x04 - Input Configuration 2**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:15</td>
<td>Input Stride (in bytes)</td>
</tr>
<tr>
<td>16:31</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**Offset 0x08 - Output Configuration 1**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:15</td>
<td>Output Width</td>
</tr>
<tr>
<td>16:31</td>
<td>Output Height</td>
</tr>
</tbody>
</table>

**Offset 0x0C - Output Configuration 2**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:15</td>
<td>Output Stride (in bytes)</td>
</tr>
<tr>
<td>16:31</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**Offset 0x10 - Input Buffer DMA Address**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Input Buffer DMA Address</td>
</tr>
</tbody>
</table>

**Offset 0x14 - Output Buffer DMA Address**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:31</td>
<td>Output Buffer DMA Address</td>
</tr>
</tbody>
</table>

**Offset 0x18 - Control and Status**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Start processing</td>
</tr>
<tr>
<td>1</td>
<td>Enable interrupts</td>
</tr>
<tr>
<td>2</td>
<td>Reset</td>
</tr>
<tr>
<td>3:4</td>
<td>Status</td>
</tr>
<tr>
<td></td>
<td>0: idle</td>
</tr>
<tr>
<td></td>
<td>1: processing</td>
</tr>
<tr>
<td></td>
<td>2: done</td>
</tr>
<tr>
<td></td>
<td>3: done but has error</td>
</tr>
</tbody>
</table>

**Programming Model**

1. Program input width and height register 0x0
2. Program input stride register 0x4
3. Program output width and height register 0x8
4. Program output stride register 0xc
5. Program input buffer DMA address register 0x10
6. Program output buffer DMA address register 0x14
7. Write the start bit (bit 0) of register 0x18
8. Either poll status or wait for interrupt (if interrupt enabled 0x14:1)
9. Discard output frame if error status bits
Driver – Platform Driver

```c
static const struct of_device_id m2m_scaler_dt_ids[] = {
    {.compatible = "virtual,m2m-scaler", .data = NULL },
};
MODULE_DEVICE_TABLE(of, m2m_scaler_dt_ids);

static struct platform_driver m2m_scaler_driver = {
    .probe    = m2m_scaler_probe,
    .remove   = m2m_scaler_remove,
    .driver   = {
        .name    = MEM2MEM_NAME,
        .of_match_table = m2m_scaler_dt_ids,
    },
};
module_platform_driver(m2m_scaler_driver);
```
Driver – probe

- Use regmap to program device
- Allocate regfields to make programming easier
- Register threaded IRQ so we can return completed frame from IRQ itself
Driver – probe

```c
ret = v4l2_device_register(&dev->dev, &device->v4l2_dev);
if (ret) {
    dev_err(dev, "could not register video device rc=%d\n", ret);
    return ret;
}

device->video_dev = m2n_scaler_video_dev;
vfd = &device->video_dev;
vfd->lock = &device->lock;
vfd->v4l2_dev = &device->v4l2_dev;

/* set the video device private data structure to struct m2n_scaler */
/* instance */
video_set_drvdata(vfd, device);

/* also set the platform private to the same */
platform_set_drvdata(pdev, device);

snprintf(vfd->name, sizeof(vfd->name), "%s", MEM2MEM_NAME);

device->m2n_dev = v4l2_n2n_init(&m2n_ops);
if (IS_ERR(device->m2n_dev)) {
    v4l2_err(v4l2_dev, "Failed to init m2n2mem device\n");
    ret = PTR_ERR(device->m2n_dev);
    goto err_v4l2;
}

ret = video_register_device(vfd, VFL_TYPE_VIDEO, 0);
if (ret) {
    v4l2_err(v4l2_dev, "Failed to register video device\n");
    goto err_m2n;
}
regmap_field_write(device->enable_interrupts, 1);
```

- Register v4l2 device
- Initialize m2m_ops
- Register video device
- Enable interrupts
Driver – probe

- Media controller is also supported
- Mostly for purposes media request API on output video node

```c
#ifndef CONFIG_MEDIA_CONTROLLER
    device->mdev.dev = &pdev->dev;
    strcpy(device->mdev.model, MEM2MEM_NAME, sizeof(device->mdev.model));
    strcpy(device->mdev.bus_info, PLATFORM M2M-SCALER, sizeof(device->mdev.bus_info));
    media_device_init(&device->mdev);
    device->mdev.ops = &m2m_media_ops;
    device->v4l2_dev.mdev = &device->mdev;

    ret = v4l2_m2m_register_media_controller(device->m2m_dev, vfd, MEDIA_ENT_F_PROC_VIDEO_COMPOSER);
    if(ret)
        v4l2_err(v4l2_dev, "Failed to init media controller\n");
        goto err_m2m;

    ret = media_device_register(&device->mdev);
    if(ret)
        v4l2_err(v4l2_dev, "Failed to register media device\n");
        goto err_m2m;
#endif
```

m2ms-source
/dev/video0

m2ms-proc

m2ms-sink
/dev/video0
Driver – ops

- **m2m_media_ops**: to support media request API
- **m2m_ops**: device_run callback actually runs the m2m job on the device
- **m2m_scaler_fops**: file operations to create and release context
Driver – open

```c
static int m2m_scaler_open(struct file *file) {
    struct m2m_scaler *device = video_drvdata(file);
    struct m2m_scaler_ctx *ctx = NULL;
    struct v4l2_device *v4l2_dev = &device->v4l2_dev;
    int rc = 8;

    if (mutex_lock_interruptible(&device->lock))
        return -EREPORTSERS;
    ctx = kzalloc(sizeof(*ctx), GFP_KERNEL);
    if (!ctx) {
        rc = -ENOMEM;
        goto open_unlock;
    }
    v4l2_fh_init(&ctx->fh, video_devdata(file));
    file->private_data = &ctx->fh;
    ctx->device = device;
    ctx->fh.m2m_ctx = v4l2_m2m_ctx_init(device->m2n_dev, ctx, &queue_init);
    if (IS_ERR(ctx->fh.m2m_ctx)) {
        rc = PTR_ERR(ctx->fh.m2m_ctx);
        v4l2_fh_exit(&ctx->fh);
        kfree(ctx);
        goto open_unlock;
    }
    v4l2_fh_add(&ctx->fh);
}
```

- Create a context
- Pass queue_init callback
- Initialize driver context related data structure
Driver – open

- Driver context initialization involves setting up the default output and capture formats.
- For this driver it's fixed to V4L2_PIX_FMT_RGB24.
Driver – release

- Uninitialize and free up the context

```c
static int m2m_scaler_release(struct file *file) {
    struct m2m_scaler *device = video_drvdata(file);
    struct m2m_scaler_ctx *ctx = file2ctx(file);
    struct v4l2_device *v4l2_dev = &device->v4l2_dev;

    v4l2_dbg(1, debug, v4l2_dev, "Releasing instance %p\n", ctx);
    v4l2_fh_del(&ctx->fh);
    v4l2_fh_exit(&ctx->fh);
    mutex_lock(&device->lock);
    v4l2_m2m_ctx_release(ctx->fh.m2m_ctx);
    mutex_unlock(&device->lock);
    kfree(ctx);

    return 0;
}
```
Driver – queue_init

- Setup output and capture video node VB2 ops
- For this device only physically contiguous memories are supported, hence use vb2_dma_contig_memops

```c
static int queue_init(void *priv, struct vb2_queue *src_vq,
                      struct vb2_queue *dst_vq)
{
    struct m2m scaler_ctx *ctx = priv;
    int ret;

    src_vq->type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
    src_vq->io_modes = VB2_MMAP | VB2_DMABUF;
    src_vq->drv_priv = ctx;
    src_vq->buf struct size = sizeof(struct v4l2 m2m_buffer);
    src_vq->ops = &m2m scaler_qops;
    src_vq->mem_ops = &vb2_dma_contig_memops;
    src_vq->timestamp_flags = V4L2_BUF_FLAG_TIMESTAMP_COPY;
    src_vq->lock = &ctx->device->lock;
    src_vq->dev = &ctx->device->v4l2_dev.dev;

    ret = vb2_queue_init(src_vq);
    if (ret)
        return ret;

    dst_vq->type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    dst_vq->io_modes = VB2_MMAP | VB2_DMABUF;
    dst_vq->drv_priv = ctx;
    dst_vq->buf struct size = sizeof(struct v4l2 m2m_buffer);
    dst_vq->ops = &m2m scaler_qops;
    dst_vq->mem_ops = &vb2_dma_contig_memops;
    dst_vq->timestamp_flags = V4L2_BUF_FLAG_TIMESTAMP_COPY;
    dst_vq->lock = &ctx->device->lock;
    dst_vq->dev = &ctx->device->v4l2_dev.dev;

    return vb2_queue_init(dst_vq);
}
```
Driver – queue_setup

```c
static int m2m_scaler_queue_setup(struct vb2_queue *vq,
    unsigned int *nbuffers, unsigned int *nplanes,
    unsigned int sizes[], struct device *alloc_devs[])
{
    struct m2m_scaler_ctx *ctx = vb2_get_drv_priv(vq);
    struct m2m_scaler *device = ctx->device;
    struct v4l2_device *v4l2_dev = &device->v4l2_dev;
    unsigned int count = *nbuffers;

    struct v4l2_format *fmt;
    fmt = m2m_scaler_get_format(ctx, vq->type);
    if(IS_ERR(fmt))
        return -EINVAL;

    *nplanes = 1;
    sizes[0] = fmt->fmt.pix.sizeimage;

    v4l2_dbg(1, debug, v4l2_dev, "get %d buffer(s) of size %d each.\n", count, sizes[0]);

    return 0;
}
```

- This callback tells VB2 the actual number of planes and size of each plane
Driver – buf_prepare

- This callback is used to prepare the buffer before queuing it to its VB2 queue
- In this call the plane payload size is set

```c
static int m2mScaler_buf_prepare(struct vb2_buffer *vb) {
    struct m2m_scaler_ctx *ctx = vb2_get_drv_priv(vb->vb2_queue);
    struct m2m_scaler *device = ctx->device;
    struct v4l2_device *v4l2_dev = &device->v4l2_dev;
    struct v4l2_format *fmt;
    fmt = m2m_scaler_get_format(ctx, vb->type);
    if(IS_ERR(fmt))
        return -EINVAL;

    v4l2_dbg(1, debug, v4l2_dev, "type: %d\n", vb->vb2_queue->type);
    vb2_set_plane_payload(vb, 0, fmt->fmt.pix.sizeimage);
    return 0;
}
```
Driver – queue, start and stop streaming

- Queue calls to `v4l2_m2m_buf_queue`
- Start streaming initialized sequence number
- Stop streaming drains the output and capture queues and returns the buffers as errors
Driver – ioctl ops

- Only format ioctl are implemented
- Since both output and capture nodes support same format, common set of callbacks are used

```c
static const struct v4l2_ioctl_ops m2m_scaler_ioctl_ops = {
    .vidioc_querycap = m2m_scaler_querycap,
    .vidioc_enum_fmt_vid_cap = m2m_scaler_enum_fmt,
    .vidioc_g_fmt_vid_cap = m2m_scaler_g_fmt,
    .vidioc_try_fmt_vid_cap = m2m_scaler_try_fmt,
    .vidioc_s_fmt_vid_cap = m2m_scaler_s_fmt,
    .vidioc_enum_fmt_vid_out = m2m_scaler_enum_fmt,
    .vidioc_g_fmt_vid_out = m2m_scaler_g_fmt,
    .vidioc_try_fmt_vid_out = m2m_scaler_try_fmt,
    .vidioc_s_fmt_vid_out = m2m_scaler_s_fmt,
    .vidioc_reqbuffs = v4l2_m2m_ioctl_reqbufs,
    .vidioc_querybuf = v4l2_m2m_ioctl_querybuf,
    .vidioc_qbuf = v4l2_m2m_ioctl_qbuf,
    .vidioc_dqbuf = v4l2_m2m_ioctl_dqbuf,
    .vidioc_prepare_buf = v4l2_m2m_ioctl_prepare_buf,
    .vidioc_create_bufs = v4l2_m2m_ioctl_create_bufs,
    .vidioc_expbuf = v4l2_m2m_ioctl_expbuf,
    .vidioc_streamon = v4l2_m2m_ioctl_streamon,
    .vidioc_streamoff = v4l2_m2m_ioctl_streamoff,
};
```
Driver – ioctl ops

- Ensure resolution limits are met
Driver – device run

Here is where the actual programming to the device occurs.

Once a job is given to the device, V4L2 M2M framework will wait for the job to complete before calling the device_run again.

Source and destination buffers are kept on their respective queues.

device_run need not be synchronous, which is typically the case.

Device starts the m2m processing and delivers an interrupt after processing is done.

```c
static void m2mScaler_device_run(void *priv)
{
    struct m2mScaler_ctx *ctx = priv;
    struct m2mScaler *device = ctx->device;
    struct v4l2_buf **src_buf, **dst_buf;
    dma_addr_t input_addr, output_addr;
    uint32_t width, height, stride;
    uint32_t owidth, oheight, ostride;
    struct v4l2_device *v4l2_dev = &device->v4l2_dev;

    src_buf = v4l2_m2m_next_src_buf(ctx->fh.m2m_ctx);
    dst_buf = v4l2_m2m_next_dst_buf(ctx->fh.m2m_ctx);

    /* program the scaler*/
    regmap_field_write(device->reset, 1);

    /* program resolution info */
    iwidth = ctx->fmt.fmt.pix.width;
    iheight = ctx->fmt.fmt.pix.height;
    istride = iwidth * 2;
    regmap_field_write(device->input_width, iwidth);
    regmap_field_write(device->input_height, iheight);
    regmap_field_write(device->input_stride, istride);

    owidth = ctx->fmt.fmt_pix_width;
    oheight = ctx->fmt.fmt_pix_height;
    ostride = owidth * 2;
    regmap_field_write(device->output_width, owidth);
    regmap_field_write(device->output_height, oheight);
    regmap_field_write(device->output_stride, ostride);

    v4l2_dbg(1, debug, v4l2_dev, "%s: width %d, height %d, input_width %d, input_height %d, input_stride %d\n", __func__, iwidth, iheight, istride);
    v4l2_dbg(1, debug, v4l2_dev, "%s: owidth %d, oheight %d, output_width %d, output_height %d, output_stride %d, output_addr %d\n", __func__, owidth, oheight, ostride, output_addr);

    /* program dma addresses */
    input_addr = v4l2_dma_contig Plane dma_addr(src_buf->v4l2_buf, 0);
    output_addr = v4l2_dma_contig Plane dma_addr(dst_buf->v4l2_buf, 0);
    regmap_field_write(device->input_addr, input_addr);
    regmap_field_write(device->output_addr, output_addr);

    /* start processing */
    regmap_field_write(device->start_processing, 1);
}
```
Driver – job complete

- Upon job completion, device will issue an interrupt
- In the ISR check to see if the m2m processioning job was successful
- Now remove the source and destination buffers from their m2m queues
- Return buffers back to their VB2 queues
- Signal m2m framework about job completion
Test Application

- Simple downsampling test
- Input buffer is 640x480 while output buffer is expected to be 320x240
Test Application

```cpp
using namespace std;
using namespace llbcamera;

class M2MScaler {
public:
    M2MScaler(Size input, Size output) : captureFrames_(0), outputFrames_(0) {
        inputSize_ = input;
        outputSize_ = output;

        enumerator_ = DeviceEnumerator::create();
        if (enumerator_) {
            cerr << "Failed to create device enumerator" << endl;
            assert(0);
        }

        if (enumerator_->enumerate()) {
            cerr << "Failed to enumerate media devices" << endl;
            assert(0);
        }

        DeviceMatch dm("m2ms");
        dm.add("m2ms-source");
        dm.add("m2ms-slink");
        media_ = enumerator_->search(dm);
        if (media_) {
            cerr << "No m2ms device found" << endl;
            assert(0);
        }
    }
};
```

- create M2MScaler class that uses libcamera’s helper classes
- Use the device enumerator that uses media controller API to discover the device driver
- Take input and output buffer sizes as class argument (Size is a libcamera class that stores width and height)
Test Application

```c
int run()
{
    constexpr unsigned int bufferCount = 1;

    EventDispatcher *dispatcher = Thread::current()->eventDispatcher();
    int ret;

    MediaEntity *entity = media_->getEntityByName("n2m-source");
    n2nScaler_ = new V4L2M2MDevice(entity->deviceNode());
    if (n2nScaler_->open()) {
        cerr << "Failed to open N2M scaler device" << endl;
        return -1;
    }

    V4L2VideoDevice *capture = n2nScaler_->capture();
    V4L2VideoDevice *output = n2nScaler_->output();

    V4L2DeviceFormat format = {};
    if (capture->getFormat(&format)) {
        cerr << "Failed to get capture format" << endl;
        return -1;
    }

    format.size = outputSize_;
    if (capture->setFormat(&format)) {
        cerr << "Failed to set capture format" << endl;
        return -1;
    }

    format.size = inputSize_;
    if (output->setFormat(&format)) {
        cerr << "Failed to set output format" << endl;
        return -1;
    }
}
```

- Find and open the video node (this will create an m2m context)
- Find the default output and capture format
- Update output and capture resolution (our driver only supports one format but different resolutions within)
Test Application

- Install callbacks to handle output and capture buffers when they get returned from the driver
- Queue the output and capture buffers
- Memory map the buffers as well
- memcpy know buffer (640x480 resolution) to mapped output buffers
- Stream on both output and capture video nodes
Test Application

- After stream on wait for required number of output and capture buffers to cycle through the driver
- Once 4 capture buffers cycled through the driver, stream off the output and capture nodes
Test Application

```c
void outputBufferComplete(FrameBuffer *buffer) {
    cout << "Received output buffer" << endl;
    outputFrames_++;
    // Requeue the buffer for further use. */
    n2nScaler_->output()->queueBuffer(buffer);
}

void receiveCaptureBuffer(FrameBuffer *buffer) {
    cout << "Received capture buffer" << endl;
    captureFrames_++;
    #ifdef DATA_CHECK
int cookie = buffer->cookie();
assert(mmcio(outputMappedBuffer_[cookie].planes()[0].data(), bbb_splash_resize_rgb, outputMappedBuffer_[cookie].planes()[0].size()) == 0);
    // Requeue the buffer for further use. */
    n2nScaler_->capture()->queueBuffer(buffer);
    #endif
}
```

- In the `outputBufferComplete` callback, simply return the buffer back to driver.
- In the `receiveCaptureBuffer` callback, check to see if the returned buffer's contents match with our expectation.
Test Application

```c
void outputBufferComplete(FrameBuffer *buffer) {
    cout << "Received output buffer" << endl;
    outputFrames_++;
    /* Requeue the buffer for further use . */
    n2mScaler_->output()->queueBuffer(buffer);
}

void receiveCaptureBuffer(FrameBuffer *buffer) {
    cout << "Received capture buffer" << endl;
    captureFrames_++;
    #ifdef DATA_CHECK
    int cookie = buffer->cookie();
    assert(memcmp(outputMappedBuffer_[cookie].planes[0].data(), bbb_splash_resize_rgb, outputMappedBuffer_[cookie].planes[0].size()) == 0);
    #endif
    /* Requeue the buffer for further use . */
    n2mScaler_->capture()->queueBuffer(buffer);
}
```

- In the `outputBufferComplete` callback, simply return the buffer back to driver.
- In the `receiveCaptureBuffer` callback, check to see if the returned buffer's contents match with our expectation.
git clone --recurse-submodules -j8 -b elc-2022 https://github.com/karthikpoduval/yoe-distro.git yoe-m2m-elc-2022

cd yoe-m2m-elc-2022

source qemuarm64-envsetup.sh

bitbake v4l2-m2m-example-image

runqemu nographic slirp

#inside QEMU

m2m-scaler-test
References

- Kernel V4L2 Documentation - https://docs.kernel.org/driver-api/media/v4l2-videobuf2.html
- M2M Scaler Driver - https://github.com/karthikpoduval/v4l2-m2m-scaler-driver/blob/elc-2022/v4l2-m2m-scaler.c
- M2M Scaler Test Application - https://github.com/karthikpoduval/meta-v4l2-m2m-example/blob/elc-2022/recipes-multimedia/m2m-scaler-test/src/m2m-example.cpp
- M2M Scaler Device (QEMU) - https://github.com/karthikpoduval/qemu/blob/elc-2022/hw/misc/m2m_scaler.c
Questions ?