International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2022): 7.942

V4L2 Mem-2-Mem as the Driver Framework for Your Video Processing IP

Karthik Poduval

Abstract: ***V4L2 M2M or mem2mem is a kernel framework that enables the use of V4L2 API for drivers of IP devices that classify themselves as memory-to-memory. This is different from the usual V4L2 output and capture devices which are memory-to-hardware or hardware-to-memory. This paper aims to explain the fundamentals of V4L2 M2M using a QEMU based virtual scaler example.***

Keywords: Video4Linux2, V4L2, mem-2-mem, M2M, Linux Device Drivers

1. Introduction

V4L2 [1] mem2mem [2] or M2M is a driver framework in the Linux kernel specially designed for memory to memory devices. The framework is a subset of the traditional V4L2 framework that applies to either capture or output devices. Capture devices refer to those which generate video Figure 1 and the video content is captured by the device driver and transferred by the V4L2 framework to the application. An output device does the opposite where video content is delivered from the application to V4L2 framework and via driver to external video devices Figure 2. For M2M devices the capture and output paths are to and from memory Figure 3.

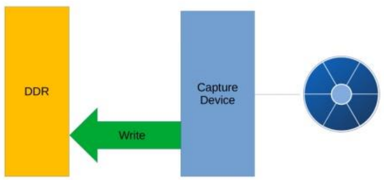


Figure 1: Capture Device

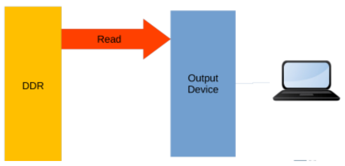


Figure 2: Output Device

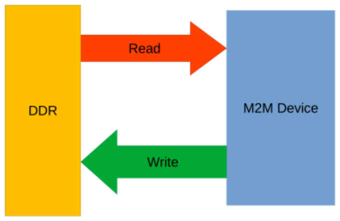


Figure 3: M2M Device

A typical V4L2 Application is typically done for either a capture or output device. For both cases the application flow is as shown in Figure 4 where application allocates a set of buffer using V4L2 API and queues them over and dequeues them from the V4L2 framework. The V4L2 framework uses a buffer manager called VB2 that handles buffers and interacts with the driver which passes the buffers to hardware and back. For capture applications the buffers returned by the hardware are used by the application while for output devices the application fills the buffers before queuing them in.

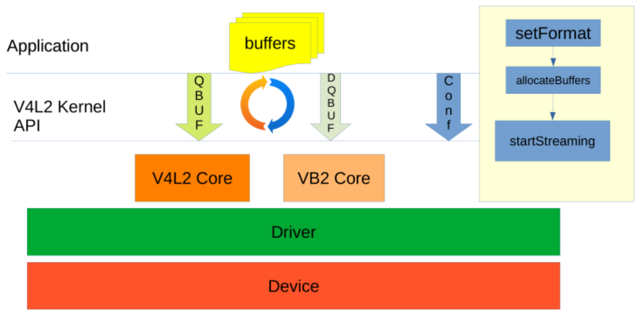


Figure 4: Typical V4L2 Application

In the case of V4L2 M2M devices, the application needs to have 2 sets of buffer pools for the output and capture parts of the M2M device as shown in Figure 5. Another interesting

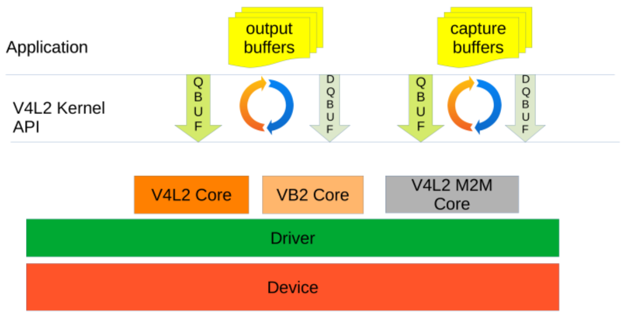


Figure 5: Typical V4L2 M2M Application

feature of the V4L2 M2M framework is that it supports multi context. Each time the video device node is opened, a context gets created and multiple applications could share a common hardware using the V4L2 M2M framework using the multiple contexts i. e. each time the application opened the device node and received a file descriptor, it can be used to operate on the context associated with the file descriptor. To explain this further, let us examine Figure 6. The light green boxes represent individual contexts. Each contexts contain a output and capture buffer pool. Application queues buffers to output and capture VB2 queues using V4L2 API. From the VB2 queue the buffers make it to V4L2 M2M queue. Note that each context has its own VB2 and V4L2 M2M queues. Each hardware instance however has a single V4L2 M2M job queue, so contexts themselves get queued to the job queue. From this queue the contexts are picked up one at a time and driver's device run callback is invoked.

The device run function pulls out the latest buffers from the context and passes it to hardware. Once the hardware finishes processing of the job i. e. reading from output buffer, processing and writing to capture buffer, it may trigger an interrupt and in the handler of that interrupt, the driver can return the buffers back to VB2 and V4L2 M2M queues. This is in a nutshell how the V4L2 M2M framework functions. To better the understanding, let us study an example scaler hardware in next section.

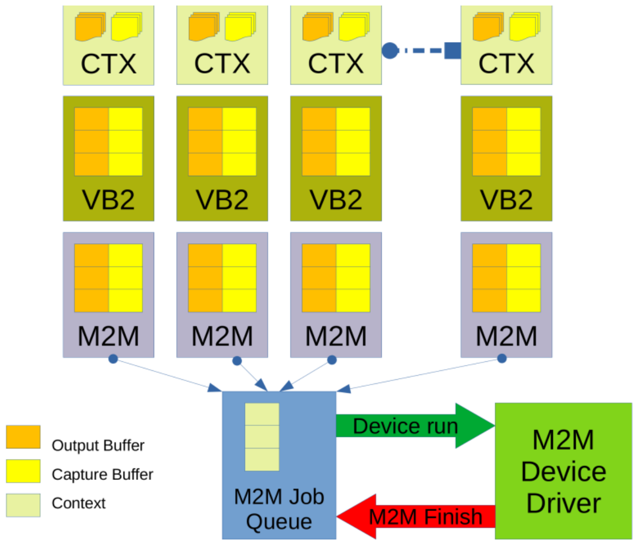


Figure 6: V4L2 M2M Architecture

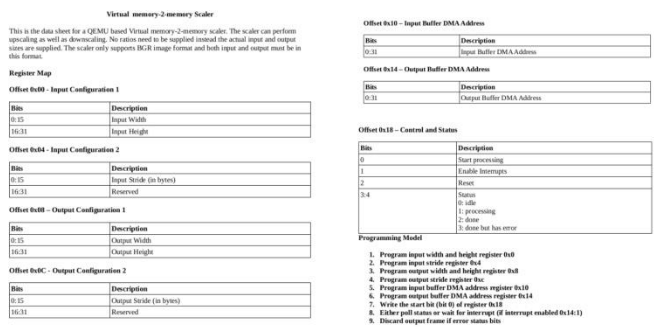
I. Scaler Hardware

In this section we introduce a virtual M2M scaler hardware that can both upscale and downscale an image. The hardware itself is virtual but interfaces like a real hardware to a guest Linux OS in QEMU [3]. Let us examine the data sheet for this hardware in Figure 7.

We can also examine the programming sequence from Figure 7 where to use the scaler the input/output width, height and stride need to be programmed. Following it, the DMA address registers need to be programmed. To start operation, the start bit needs to be written. The virtual hardware will trigger an interrupt to inform the driver when the scaling has been completed.

II. V4L2 M2M LINUX DRIVER

Now we will attempt to write a device driver for this virtual scaler hardware. We will write a Linux platform device driver



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 BGR image format and both input and output must be in this format.

Register Map

Offset 0x00 - Input Configuration 1

Bits	Description
0:15	Input Width
16:31	Input Height

Offset 0x04 - Input Configuration 2

Bits	Description
0:15	Input Stride (in bytes)
16:31	Reserved

Offset 0x08 – Output Configuration 1

Bits	Description
0:15	Output Width
16:31	Output Height

Offset 0x0C - Output Configuration 2

Bits	Description
0:15	Output Stride (in bytes)
16:31	Reserved

Offset 0x10 – Input Buffer DMA Address

Bits	Description
0:31	Input Buffer DMA Address

Offset 0x14 – Output Buffer DMA Address

Bits	Description
0:31	Output Buffer DMA Address

Offset 0x18 – Control and Status

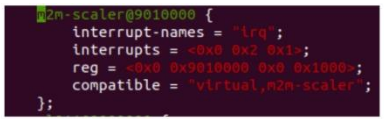
Bits	Description
0	Start processing
1	Enable Interrupts
2	Reset
3:4	Status 0: idle 1: processing 2: done 3: done but has error

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**

Figure 7: Virtual M2M Scaler Data Sheet

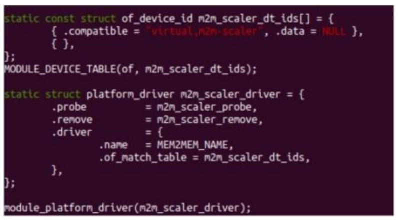
for this hardware as it's been connected to QEMU's virt hard-ware's MMIO (Memory Mapped IO) at address 0x09010000. The QEMU machine already creates a device tree entry for this and passes it to Linux guest OS as shown in Figure 8.



```
m2m-scaler@9010000 {
        interrupt-names = "irq";
        interrupts = <0x0 0x2 0x1>;
        reg = <0x0 0x9010000 0x0 0x1000>;
        compatible = "virtual,m2m-scaler";
};
```

Figure 8: Device Tree Entry

The platform driver entry point should look like the following Figure 9. We can see that the same compatible string as the device tree is placed which will make the Linux platform driver framework find and match this driver to the platform device.



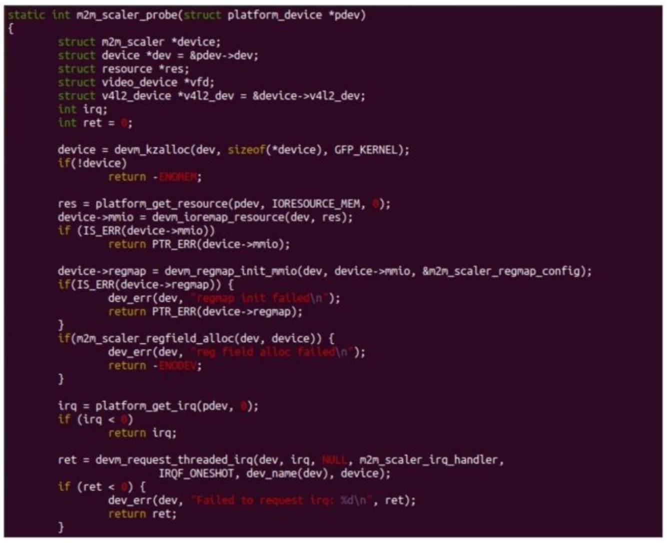
```
static const struct of_device_id m2m_scaler_dt_ids[] = {
        { .compatible = "virtual,m2m-scaler", .data = NULL },
        { },
};
MODULE_DEVICE_TABLE(of, m2m_scaler_dt_ids);

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);
```

Figure 9: Platform Driver Entry

Once Linux boots, the platform device framework of Linux should now call the probe method after matching the compatible string. Let us examine the probe method and its key functionalities.

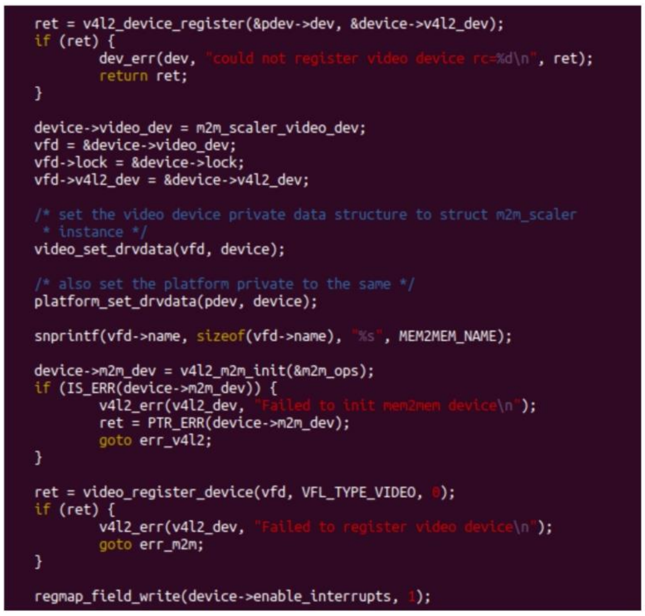
- grab resources like MMIO, IRQ
- use regmap library to do register manipulation
- register a threaded IRQ handler so that frame completion can be signalled from the interrupt handler
- In the other parts of probe function Figure 11 the following steps are taken.
- register V4L2 device
- initialize V4L2 M2M ops
- register video device
- write a register to enable the interrupt

An optional media controller registration is done in the final parts of probe Figure 12. This registration is useful for using media controller features such as discovery and media request API. A media device graph for this



```c
static int m2m_scaler_probe(struct platform_device *pdev)
{
        struct m2m_scaler *device;
        struct device *dev = &pdev->dev;
        struct resource *res;
        struct video_device *vfd;
        struct v4l2_device *v4l2_dev = &device->v4l2_dev;
        int irq;
        int ret = 0;

        device = devm_kzalloc(dev, sizeof(*device), GFP_KERNEL);
        if(!device)
                return -ENOMEM;

        res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        device->mmio = devm_ioremap_resource(dev, res);
        if (IS_ERR(device->mmio))
                return PTR_ERR(device->mmio);

        device->regmap = devm_regmap_init_mmio(dev, device->mmio, &m2m_scaler_regmap_config);
        if(IS_ERR(device->regmap)) {
                dev_err(dev, "regmap init failed\n");
                return PTR_ERR(device->regmap);
        }
        if(m2m_scaler_regfield_alloc(dev, device)) {
                dev_err(dev, "reg field alloc failed\n");
                return -ENODEV;
        }

        irq = platform_get_irq(pdev, 0);
        if (irq < 0)
                return irq;

        ret = devm_request_threaded_irq(dev, irq, NULL, m2m_scaler_irq_handler,
                        IRQF_ONESHOT, dev_name(dev), device);
        if (ret < 0) {
                dev_err(dev, "Failed to request irq: %d\n", ret);
                return ret;
        }
```

Figure 10: Platform Driver Probe



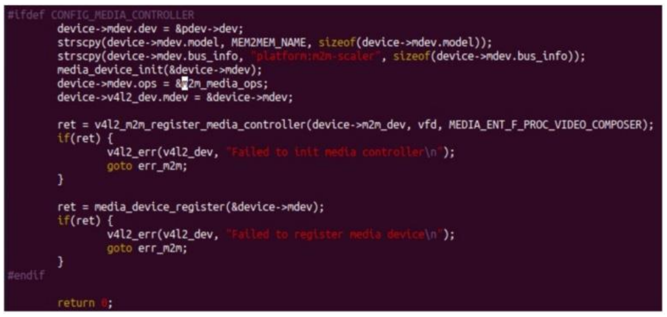
```c
ret = v4l2_device_register(&pdev->dev, &device->v4l2_dev);
if (ret) {
        dev_err(dev, "could not register video device rc=%d\n", ret);
        return ret;
}

device->video_dev = m2m_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 m2m_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->m2m_dev = v4l2_m2m_init(&m2m_ops);
if (IS_ERR(device->m2m_dev)) {
        v4l2_err(v4l2_dev, "Failed to init mem2mem device\n");
        ret = PTR_ERR(device->m2m_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_m2m;
}

regmap_field_write(device->enable_interrupts, 1);
```

Figure 11: Platform Driver Probe contd.

device may look like Figure 13, generated with command media-ctl-- print-dot-d <media device>. Now let us examine the driver ops as shwon in Figure 14.

- m2m_media_ops: These ops are there to support media request API
- m2m_ops: The V4L2 M2M driver ops, this driver implement a device run which actually runs an M2M job on the device
- m2m_scaler_fops: The File operations for this driver needed for context creation.
- m2m_scaler_ioctl_ops: The File operations for the V4L2 APIs.

The driver file open Figure 15 m2m_scaler_open function is where a new M2M context (Figure 16) gets created and initialized and added to the V4L2 file handle (which will be used by the other API to retrieve the context). This is the mechanism used by V4L2 M2M framework to support multi context, each time application opens the video device and gets a file descriptor, a new context gets created and added to the file handle.



```c
#ifdef CONFIG_MEDIA_CONTROLLER
        device->mdev.dev = &pdev->dev;
        strscpy(device->mdev.model, MEM2MEM_NAME, sizeof(device->mdev.model));
        strscpy(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

        return 0;
```

Figure 12: Platform Driver Probe contd.

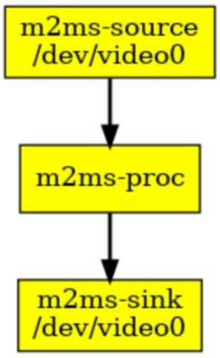
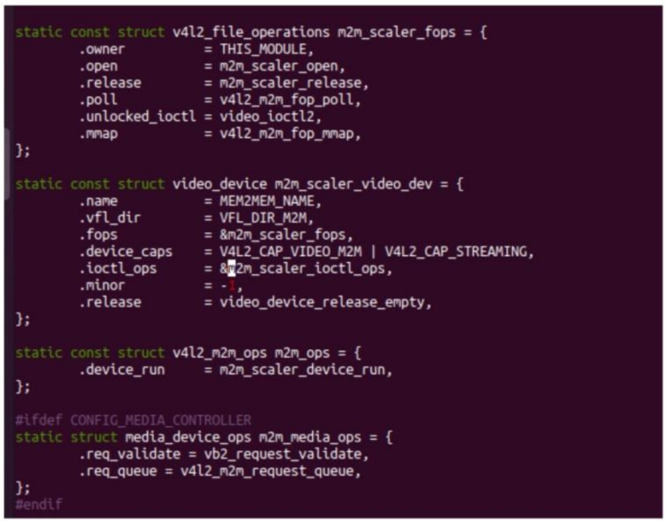


Figure 13: Media Graph

The driver also initializes its own context variables as shown in Figure 17. The driver it sets up default width, height and pixel format for output and capture sides of V4L2 M2M.

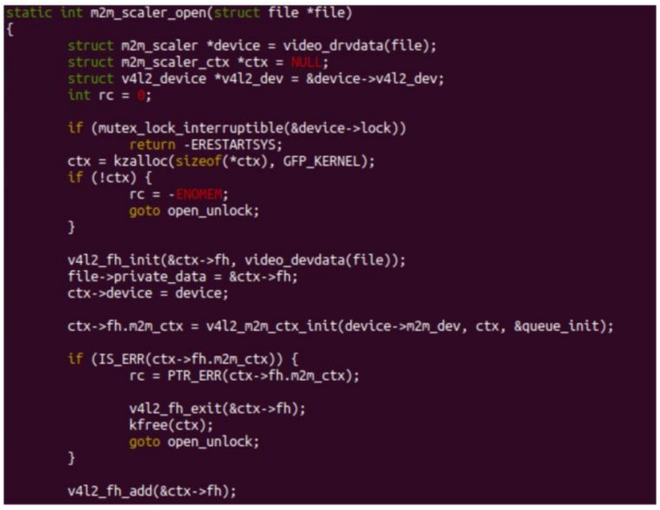
When the application closes the file descriptor the driver's release function pointer gets invokes Figure 18. Here the context is being released and freed.

As a part of context initialization in Figure 15 aqueue_init method is passed, let us look into this Figure 19. In this function driver initializes the output and capture queues for V4L2 M2M. It also deices on what all types of io_modes must be supported by the driver, most modern drivers would support MMAP and DMABUF io_modes. The vb2_ops can be seen in Figure 20. Let us examine the m2m_scaler_queue_setup Figure 21. This is a callback to VB2 layer as the driver needs to inform VB2 about the number of planes in the format and the actual size of each plane based on which VB2 will allocate memory for the buffer. As this driver only supports an RGB888 format its a single plane and size image is computer from the default or changes made while trying to set the format. A driver helper function names m2m_scaler_get_format is used to get the format for output or capture side so that the same m2m_scaler_buf_prepare can be used for output and capture queue. The next interesting VB2 callback is buf_prepare Figure 21.



```
static const struct v4l2_file_operations m2m_scaler_fops = {
        .owner          = THIS_MODULE,
        .open           = m2m_scaler_open,
        .release        = m2m_scaler_release,
        .poll           = v4l2_m2m_fop_poll,
        .unlocked_ioctl = video_ioctl2,
        .mmap           = v4l2_m2m_fop_mmap,
};

static const struct video_device m2m_scaler_video_dev = {
        .name           = MEM2MEM_NAME,
        .vfl_dir        = VFL_DIR_M2M,
        .fops           = &m2m_scaler_fops,
        .device_caps    = V4L2_CAP_VIDEO_M2M | V4L2_CAP_STREAMING,
        .ioctl_ops      = &m2m_scaler_ioctl_ops,
        .minor          = -1,
        .release        = video_device_release_empty,
};

static const struct v4l2_m2m_ops m2m_ops = {
        .device_run     = m2m_scaler_device_run,
};

#ifdef CONFIG_MEDIA_CONTROLLER
static struct media_device_ops m2m_media_ops = {
        .req_validate = vb2_request_validate,
        .req_queue = v4l2_m2m_request_queue,
};
#endif
```

Figure 14: Driver Ops

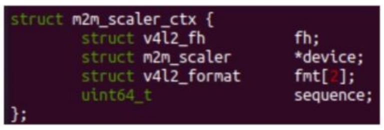


```
static int m2m_scaler_open(struct file *file)
{
        struct m2m_scaler *device = video_drvdata(file);
        struct m2m_scaler_ctx *ctx = NULL;
        struct v4l2_device *v4l2_dev = &device->v4l2_dev;
        int rc = 0;

        if (mutex_lock_interruptible(&device->lock))
                return -ERESTARTSYS;
        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->m2m_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);
```

Figure 15: Driver File Open

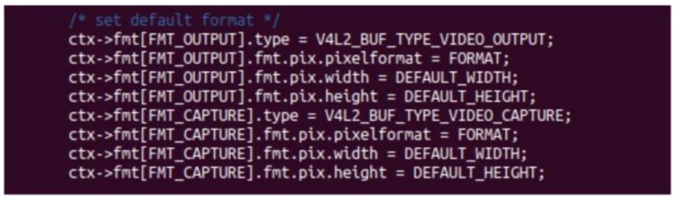
This callback is used to prepare the buffer before queuing it to this VB2 queue. The driver sets the plane payload size. Some other important callbacks can be seen in Figure 23. m2m_scaler_buf_queue is a driver wrapper that simply calls v4l2_m2m_buf_queue after extracting the context data structure. The m2m_scaler_start_streaming API initializes a sequence number that the driver will populate for each buffer returned after processing is complete by the hardware. The m2m_scaler_stop_streaming performs cleanup and returns all the queued buffers back to the frame-work as error buffers as the intent to stream has ended and those buffers aren't actually processed by hardware and hence must be marked as errors.

Figure 24 shows the V4L2 IOCTL callbacks. Most of them point to V4L2 M2M framework's APIs but some of them driver overrides to local methods to set, get and enumerate



```
struct m2m_scaler_ctx {
        struct v4l2_fh          fh;
        struct m2m_scaler       *device;
        struct v4l2_format      fmt[2];
        uint64_t                sequence;
};
```

Figure 16: Context Data Structure



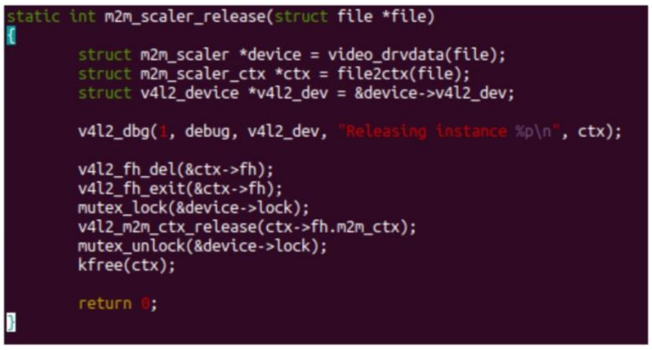
```
        /* set default format */
        ctx->fmt[FMT_OUTPUT].type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
        ctx->fmt[FMT_OUTPUT].fmt.pix.pixelformat = FORMAT;
        ctx->fmt[FMT_OUTPUT].fmt.pix.width = DEFAULT_WIDTH;
        ctx->fmt[FMT_OUTPUT].fmt.pix.height = DEFAULT_HEIGHT;
        ctx->fmt[FMT_CAPTURE].type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        ctx->fmt[FMT_CAPTURE].fmt.pix.pixelformat = FORMAT;
        ctx->fmt[FMT_CAPTURE].fmt.pix.width = DEFAULT_WIDTH;
        ctx->fmt[FMT_CAPTURE].fmt.pix.height = DEFAULT_HEIGHT;
```

Figure 17: Driver Context Initialization

formats. The driver uses same callbacks between output and capture sides.

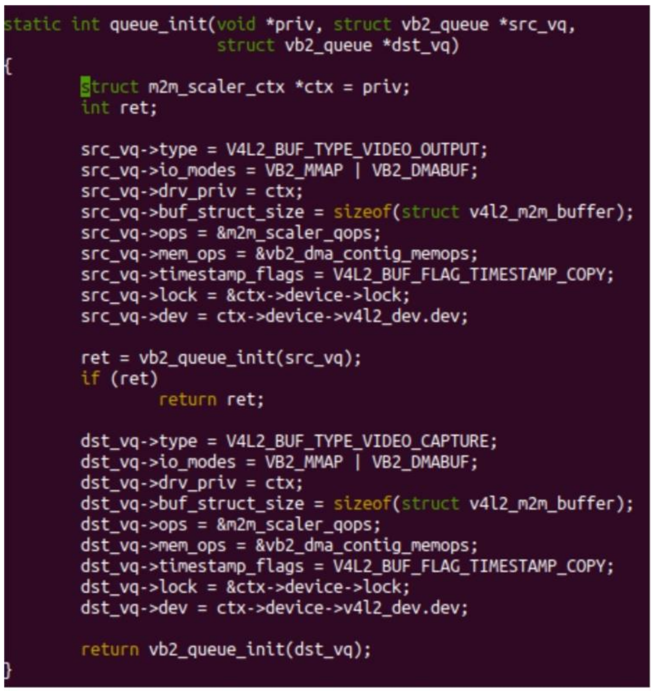
Referring to Figures 25 and 26, the m2m_scaler_try_fmt API checks to see if format can be supported and accordingly corrects it to something supported if set wrongly by the application. The m2m_sclaer_enum_fmt API enumerates all the supported format which is only a 1 format enumeration for this driver. The m2m_scaler_s_fmt API actually sets the format and stores it into the context which will be used later by the m2m_scaler_queue_setup and m2m_scaler_buf_prepare VB2 callback API's from Figures 21 and 22. The m2m_scaler_g_fmt API returns the current set format to the application, it would return the default value (if not set previously) that was initialled in the context by the m2m_scaler_open Figure 17.

Figure 27 shows the function m2m_scaler_device_run function where the actual hardware programming is performed. The source and destination buffers are fetched from the output and captures queue. By the time these v4l2_m2m_next_src_buf and v4l2_m2m_next_dst_buf methods are called, the V4L2 M2M framework has already done enough validation to ensure that the driver device_run callback is only invoked when there is at least one buffer queued into the output and capture queues. Certain V4L2 M2M drives like video encoders can also decide to encode several buffers into a single encoded buffer and in such cases another device_ready optional callback can be used to let V4L2 M2M framework know about any deviation from default assumption of one output and one capture buffer to be queued before invoking the device_run callback. The next thing is to reset the hardware as different contexts might use hardware very differently and it is important to reset the hardware before device_run to be truly stateless. From the context the input and output parameters like width, height and stride are fetched and hardware is programmed with such information. Next thing is to get the DMA addresses using some VB2 API and program the hardware's DMA pointer registers. Finally the processing is started by writing to the start_processing bit of the hardware. V4L2 M2M framework ensures that while device_run callback



```
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;
}
```

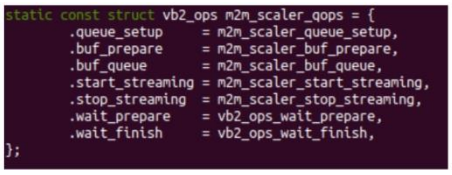
Figure 18: Driver File Release



```
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);
}
```

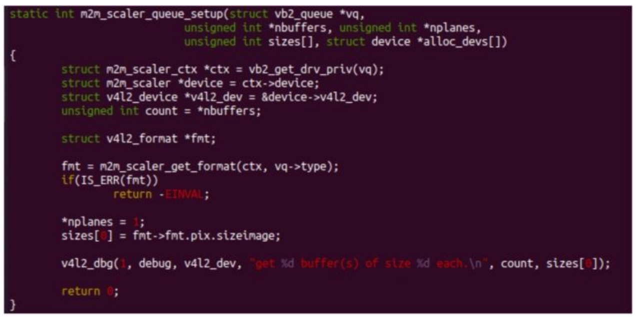
Figure 19: Queue Init

is invoked, a second invocation wouldn't happen until the v4m2_m2m_job_finish API is called for that context job. We has configured the driver to register a threaded ISR. As shown in Figure 28, the ISR would determine if hardware successfully completed the scaling operation and call VB2 and V4L2 M2M to complete the job and return the buffers back to VB2 and hence back to the application. As you can notice the actual buffers are removed from the queues only in the ISR as device_ run needn't do that as it only needs to get the addresses at that stage and the actual buffers can be removed from queues at the time of buffer completion in the ISR. The call to v4m2_m2m_job_finish signals V4L2 M2M framework to now schedule another context to the same driver.



```
static const struct vb2_ops m2m_scaler_qops = {
        .queue_setup     = m2m_scaler_queue_setup,
        .buf_prepare     = m2m_scaler_buf_prepare,
        .buf_queue       = m2m_scaler_buf_queue,
        .start_streaming = m2m_scaler_start_streaming,
        .stop_streaming  = m2m_scaler_stop_streaming,
        .wait_prepare    = vb2_ops_wait_prepare,
        .wait_finish     = vb2_ops_wait_finish,
};
```

Figure 20:VB2 Ops



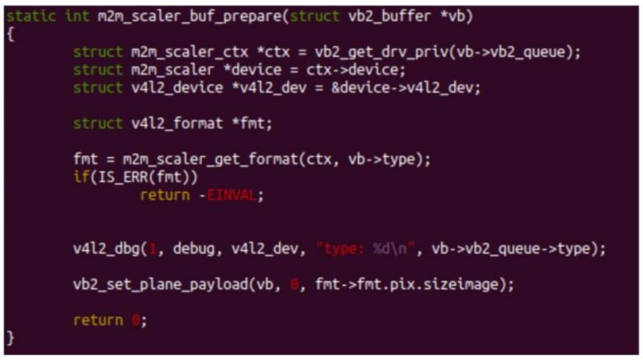
```
static int m2m_scaler_queue_setup(struct vb2_queue *vq,
                                  unsigned int *nbuffers, unsigned int *nplanes,
                                  unsigned int sizes[], struct device *alloc_devs[])
{
        struct m2m_scaler_ctx *ctx = vb2_get_drv_priv(vq);
        struct 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;
}
```

Figure 21: VB2 Queue Setup

2. Test Application

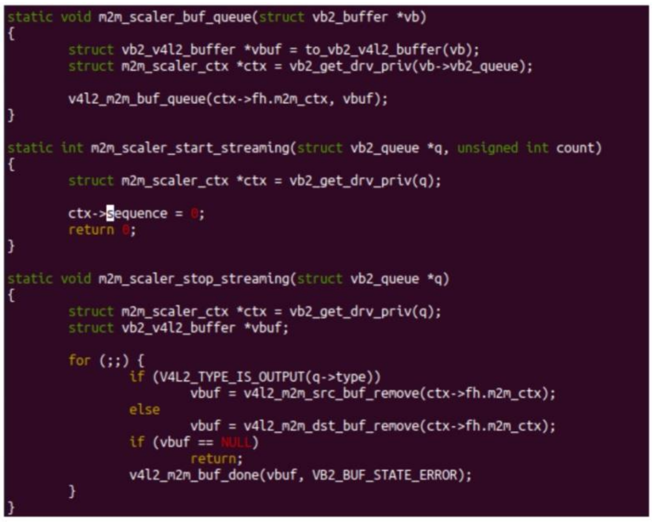
In order to test the driver, we need an application that can interact with the M2M scaler using V4L2 API. It needs to open a output and capture sides, setup formats, queue buffers and start streaming all using the opened file descriptor (context). To abstract the V4L2, let us use libcamera's powerful abstraction of V4L2 in C++. The basic application C++ class is shown in Figure 29. The enumerator_ is used to find the scaler using media controller discovery API via the libcamera class Device Enumerator. m2mScaler_ class handles the V4L2 M2M device APIs. Capture and output buffers are handled in vectors capture Buffers_ and output Buffers. The input Size_ and output Size_ are the scaling parameters. Input MappedBuffer_and output Mapped Buffer help to memory map the buffers. The buffers themselves are DMABUF file descriptors as V4L2 is configured to export its memory as DMABUF file descriptors that are memory mapped for use.

The application main should look as shown in Figure 30 which demonstrates a 640x480 to 320x240 downscale operation. The application enumeration can be seen as shown in Figure 31 and this uses the media controller name set by the driver. Now let us look at the run method of the M2MScaler test class Figure 33. We first find the media entity and grab pointers to output and captures sides of the M2M Scaler. We then use the input and output size parameters and set them to input and output side formats. Next, we connect callbacks to handle output buffer completion and receive capture buffer, callbacks sown in Figure 32. Then the buffers are allocated using the driver and the allocated buffers are also queued into the driver while also copying the contents of the input image to the output side buffers. The image is itself an RGB 24 bit format and stored as an array into a header file using xxd program. The output callback is supposed to queue the next input buffer into the driver, but our simple application queues the same



```
static int m2m_scaler_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;
}
```

Figure 22: VB2 Buf Prepare



```
static void m2m_scaler_buf_queue(struct vb2_buffer *vb)
{
        struct vb2_v4l2_buffer *vbuf = to_vb2_v4l2_buffer(vb);
        struct m2m_scaler_ctx *ctx = vb2_get_drv_priv(vb->vb2_queue);

        v4l2_m2m_buf_queue(ctx->fh.m2m_ctx, vbuf);
}

static int m2m_scaler_start_streaming(struct vb2_queue *q, unsigned int count)
{
        struct m2m_scaler_ctx *ctx = vb2_get_drv_priv(q);

        ctx->sequence = 0;
        return 0;
}

static void m2m_scaler_stop_streaming(struct vb2_queue *q)
{
        struct m2m_scaler_ctx *ctx = vb2_get_drv_priv(q);
        struct vb2_v4l2_buffer *vbuf;

        for (;;) {
                if (V4L2_TYPE_IS_OUTPUT(q->type))
                        vbuf = v4l2_m2m_src_buf_remove(ctx->fh.m2m_ctx);
                else
                        vbuf = v4l2_m2m_dst_buf_remove(ctx->fh.m2m_ctx);
                if (vbuf == NULL)
                        return;
                v4l2_m2m_buf_done(vbuf, VB2_BUF_STATE_ERROR);
        }
}
```

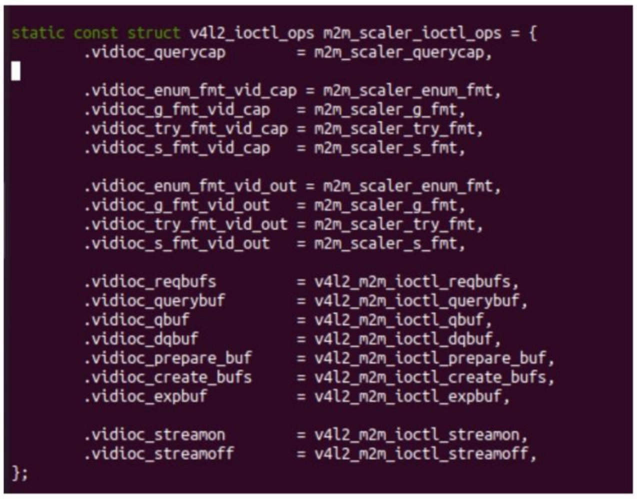
Figure 23: Other Callback API

buffer. Then the streamOn API is called for both the output and capture sides. The capture buffer callback has an optional step where contents are compared to a ground truth vector to verify if the M2M Scaler has performed scaling correctly or not. The application waits for desired number of frames to complete and then exits.

3. Build This System

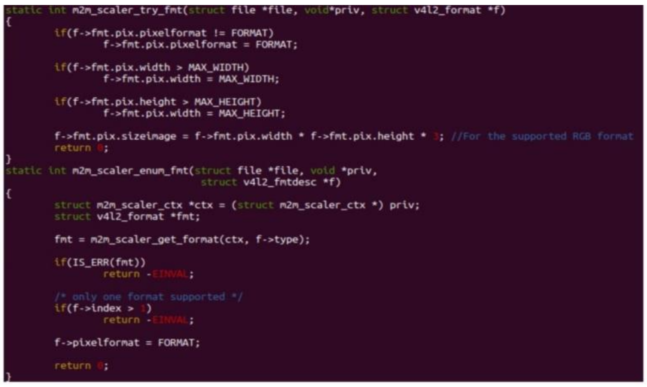
The system can be built using the following steps.
git clone-- recurse-submodules-j8
-b elc-2022 https: //github. com/karthikpoduval
/yoe-distro. git m2m cd m2m
source qemuarm64-envsetup. sh bitbake v4l2-m2m-example-image runqemu nographic slirp #inside QEMU m2m-scaler-test

- M2M Scaler driver https: //github. com/karthikpoduval/v4l2-m2m-scaler-driver/blob/elc-2022/v4l2-m2m-scaler. 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_reqbufs         = 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,
};
```

Figure 24: IOCTLs



```
static int m2m_scaler_try_fmt(struct file *file, void*priv, struct v4l2_format *f)
{
        if(f->fmt.pix.pixelformat != FORMAT)
                f->fmt.pix.pixelformat = FORMAT;

        if(f->fmt.pix.width > MAX_WIDTH)
                f->fmt.pix.width = MAX_WIDTH;

        if(f->fmt.pix.height > MAX_HEIGHT)
                f->fmt.pix.width = MAX_HEIGHT;

        f->fmt.pix.sizeimage = f->fmt.pix.width * f->fmt.pix.height * 3; //For the supported RGB format
        return 0;
}
static int m2m_scaler_enum_fmt(struct file *file, void *priv,
                                struct v4l2_fmtdesc *f)
{
        struct m2m_scaler_ctx *ctx = (struct m2m_scaler_ctx *) priv;
        struct v4l2_format *fmt;

        fmt = m2m_scaler_get_format(ctx, f->type);

        if(IS_ERR(fmt))
                return -EINVAL;

        /* only one format supported */
        if(f->index > 1)
                return -EINVAL;

        f->pixelformat = FORMAT;

        return 0;
}
```

Figure 25: Try and Enum Formats

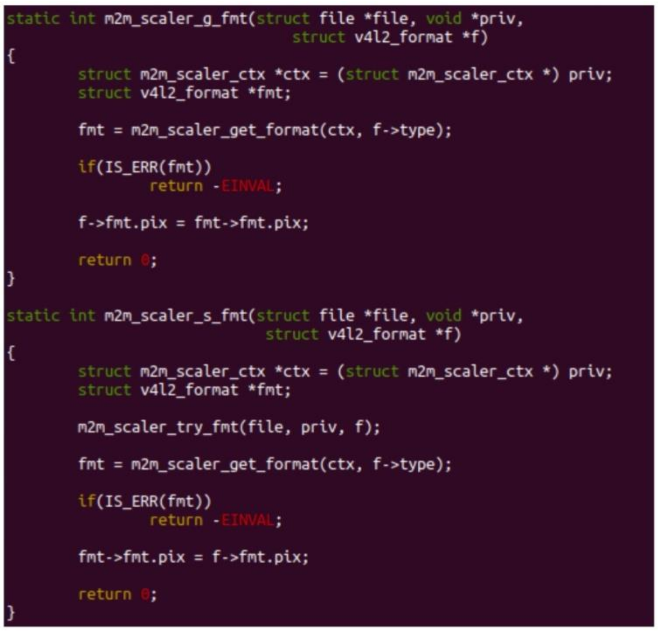
- M2M Scaler Test Application https: //github. com/karthikpoduval/meta-v4l2-m2m-example/blob/elc-2022/recipes-multimedia/m2m-scaler-test/src/m2m-example. cpp
- M2M Scaler Datasheet https: //github. com/karthikpoduval/meta-v4l2-m2m-example/blob/elc-2022/m2m-scaler-datasheet. pdf

4. Conclusion

Using the above driver and virtual hardware, one can understand how to write a driver for their video processing IP using the V4L2 M2M framework. The driver can serve as a starting point driver while creating a new driver for a video processing IP and/or serve as a useful reference.

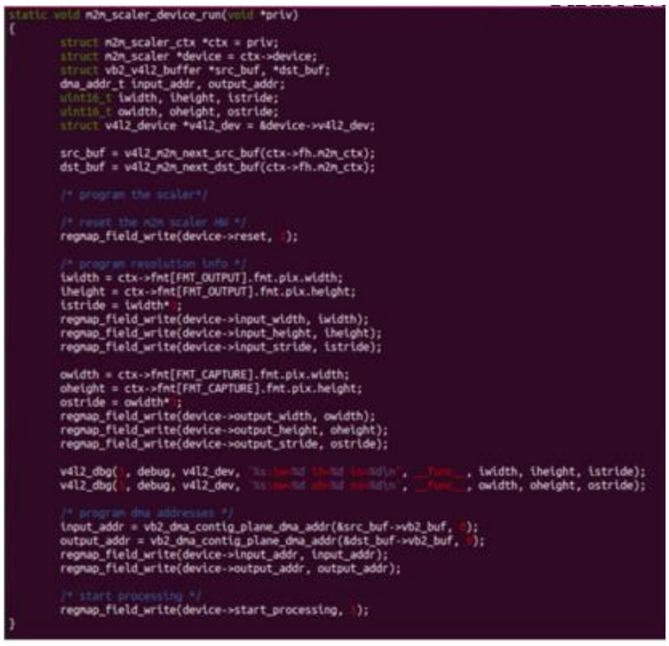
References

[1] "Linux kernel v4l2 documentaion. " [Online]. Available: https: //www. kernel. org/doc/html/v4.19/media/uapi/v4l/v4l2. html
[2] P. Osciak, "Mem-to-mem device framework, " 12 2009. [Online].
[3] Available: https: //lwn. net/Articles/367881/
[4] F. Bellard, "Qemu, a fast and portable dynamic translator. " in *USENIX annual technical conference*, *FREENIX Track*, vol.41. Califor-nia, USA, 2005, p.46.



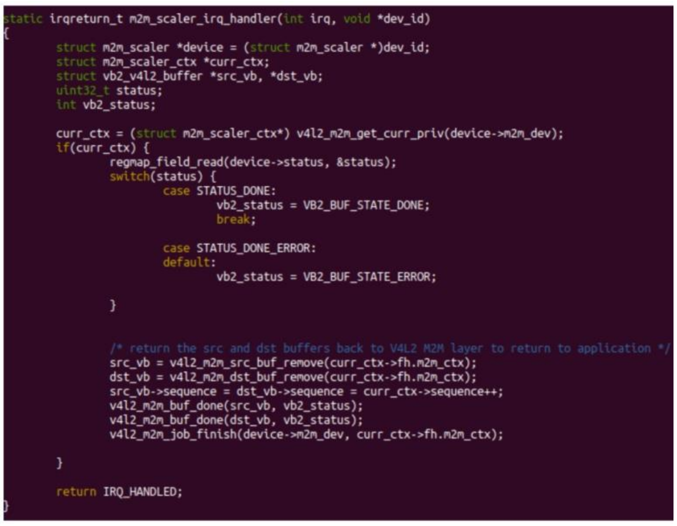
```
static int m2m_scaler_g_fmt(struct file *file, void *priv,
                                        struct v4l2_format *f)
{
        struct m2m_scaler_ctx *ctx = (struct m2m_scaler_ctx *) priv;
        struct v4l2_format *fmt;

        fmt = m2m_scaler_get_format(ctx, f->type);

        if(IS_ERR(fmt))
                return -EINVAL;

        f->fmt.pix = fmt->fmt.pix;

        return 0;
}

static int m2m_scaler_s_fmt(struct file *file, void *priv,
                                        struct v4l2_format *f)
{
        struct m2m_scaler_ctx *ctx = (struct m2m_scaler_ctx *) priv;
        struct v4l2_format *fmt;

        m2m_scaler_try_fmt(file, priv, f);

        fmt = m2m_scaler_get_format(ctx, f->type);

        if(IS_ERR(fmt))
                return -EINVAL;

        fmt->fmt.pix = f->fmt.pix;

        return 0;
}
```

Figure 26: Get and Set Format



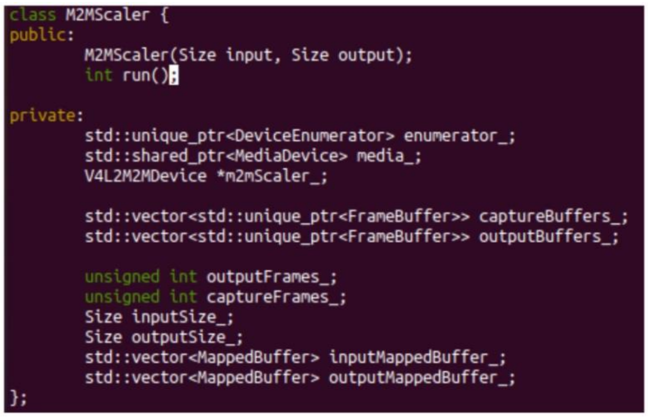
```
static void m2m_scaler_device_run(void *priv)
{
        struct m2m_scaler_ctx *ctx = priv;
        struct m2m_scaler *device = ctx->device;
        struct vb2_v4l2_buffer *src_buf, *dst_buf;
        dma_addr_t input_addr, output_addr;
        uint16_t iwidth, iheight, istride;
        uint16_t owidth, 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*/

        /* reset the m2m scaler HW */
        regmap_field_write(device->reset, 1);

        /* program resolution info */
        iwidth = ctx->fmt[FMT_OUTPUT].fmt.pix.width;
        iheight = ctx->fmt[FMT_OUTPUT].fmt.pix.height;
        istride = iwidth*3;
        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_CAPTURE].fmt.pix.width;
        oheight = ctx->fmt[FMT_CAPTURE].fmt.pix.height;
        ostride = owidth*3;
        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: iw=%d ih=%d is=%d\n", __func__, iwidth, iheight, istride);
        v4l2_dbg(1, debug, v4l2_dev, "%s: ow=%d oh=%d os=%d\n", __func__, owidth, oheight, ostride);

        /* program dma addresses */
        input_addr = vb2_dma_contig_plane_dma_addr(&src_buf->vb2_buf, 0);
        output_addr = vb2_dma_contig_plane_dma_addr(&dst_buf->vb2_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);
}
```

Figure 27: Device Run



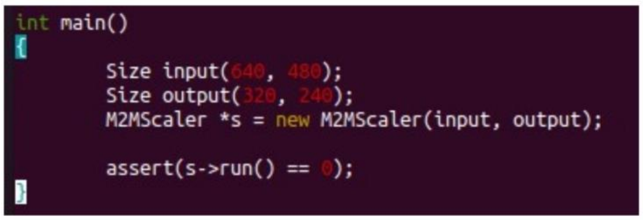
```
static irqreturn_t m2m_scaler_irq_handler(int irq, void *dev_id)
{
        struct m2m_scaler *device = (struct m2m_scaler *)dev_id;
        struct m2m_scaler_ctx *curr_ctx;
        struct vb2_v4l2_buffer *src_vb, *dst_vb;
        uint32_t status;
        int vb2_status;

        curr_ctx = (struct m2m_scaler_ctx*) v4l2_m2m_get_curr_priv(device->m2m_dev);
        if(curr_ctx) {
                regmap_field_read(device->status, &status);
                switch(status) {
                        case STATUS_DONE:
                                vb2_status = VB2_BUF_STATE_DONE;
                                break;

                        case STATUS_DONE_ERROR:
                        default:
                                vb2_status = VB2_BUF_STATE_ERROR;

                }


                /* return the src and dst buffers back to V4L2 M2M layer to return to application */
                src_vb = v4l2_m2m_src_buf_remove(curr_ctx->fh.m2m_ctx);
                dst_vb = v4l2_m2m_dst_buf_remove(curr_ctx->fh.m2m_ctx);
                src_vb->sequence = dst_vb->sequence = curr_ctx->sequence++;
                v4l2_m2m_buf_done(src_vb, vb2_status);
                v4l2_m2m_buf_done(dst_vb, vb2_status);
                v4l2_m2m_job_finish(device->m2m_dev, curr_ctx->fh.m2m_ctx);

        }

        return IRQ_HANDLED;
}
```

Figure 28: ISR



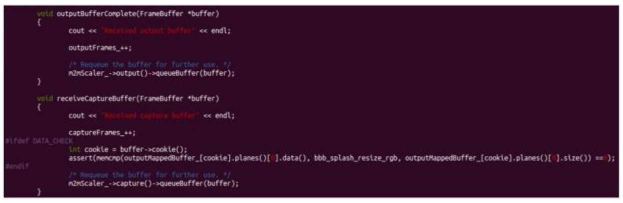
```
class M2MScaler {
public:
        M2MScaler(Size input, Size output);
        int run();

private:
        std::unique_ptr<DeviceEnumerator> enumerator_;
        std::shared_ptr<MediaDevice> media_;
        V4L2M2MDevice *m2mScaler_;

        std::vector<std::unique_ptr<FrameBuffer>> captureBuffers_;
        std::vector<std::unique_ptr<FrameBuffer>> outputBuffers_;

        unsigned int outputFrames_;
        unsigned int captureFrames_;
        Size inputSize_;
        Size outputSize_;
        std::vector<MappedBuffer> inputMappedBuffer_;
        std::vector<MappedBuffer> outputMappedBuffer_;
};
```

Figure 29: Application Class



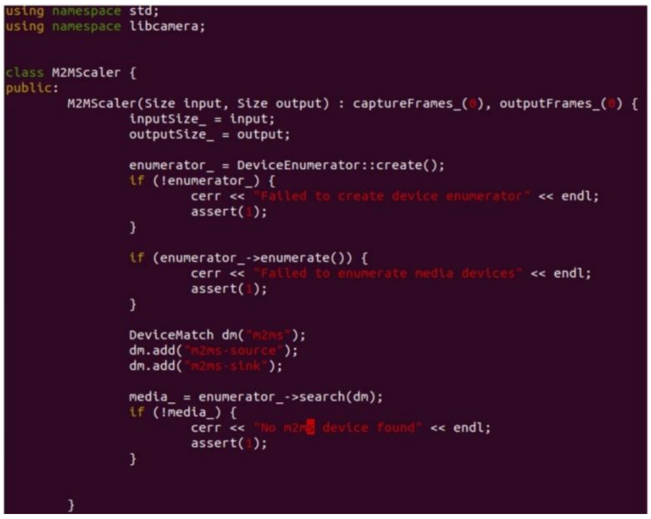
```
int main()
{
        Size input(640, 480);
        Size output(320, 240);
        M2MScaler *s = new M2MScaler(input, output);

        assert(s->run() == 0);
}
```

Figure 30: Application Main



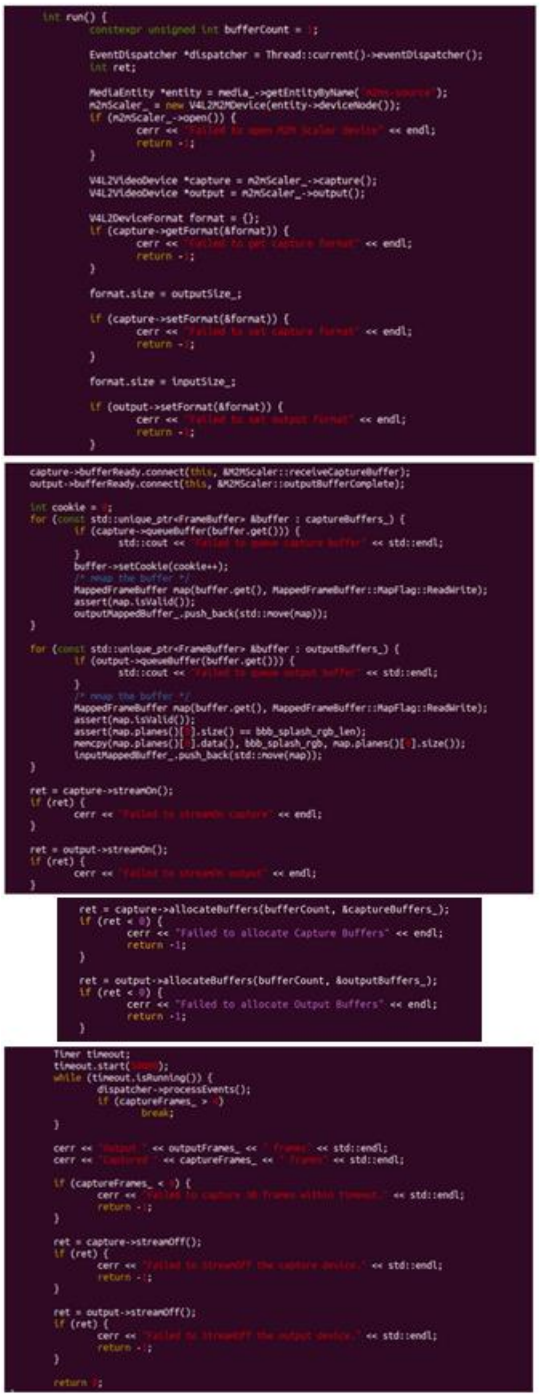
```
        void outputBufferComplete(FrameBuffer *buffer)
        {
                cout << "Received output buffer" << endl;

                outputFrames_++;

                /* Requeue the buffer for further use. */
                m2mScaler_->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. */
                m2mScaler_->capture()->queueBuffer(buffer);
        }
```

Figure 31: Application Enumeration



```
using namespace std;
using namespace libcamera;


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(1);
                }

                if (enumerator_->enumerate()) {
                        cerr << "Failed to enumerate media devices" << endl;
                        assert(1);
                }

                DeviceMatch dm("m2ms");
                dm.add("m2ms-source");
                dm.add("m2ms-sink");

                media_ = enumerator_->search(dm);
                if (!media_) {
                        cerr << "No m2ms device found" << endl;
                        assert(1);
                }


        }
```

Figure 32: Application Callbacks



```
    int run() {
            constexpr unsigned int bufferCount = [illegible];

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

            MediaEntity *entity = media_->getEntityByName("m2m-source");
            m2mScaler_ = new V4L2M2MDevice(entity->deviceNode());
            if (m2mScaler_->open()) {
                    cerr << "Failed to open M2M Scaler device" << endl;
                    return -1;
            }

            V4L2VideoDevice *capture = m2mScaler_->capture();
            V4L2VideoDevice *output = m2mScaler_->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;
            }
```

```
    capture->bufferReady.connect(this, &M2MScaler::receiveCaptureBuffer);
    output->bufferReady.connect(this, &M2MScaler::outputBufferComplete);

    int cookie = 0;
    for (const std::unique_ptr<FrameBuffer> &buffer : captureBuffers_) {
            if (capture->queueBuffer(buffer.get())) {
                    std::cout << "Failed to queue capture buffer" << std::endl;
            }
            buffer->setCookie(cookie++);
            /* mmap the buffer */
            MappedFrameBuffer map(buffer.get(), MappedFrameBuffer::MapFlag::ReadWrite);
            assert(map.isValid());
            outputMappedBuffer_.push_back(std::move(map));
    }

    for (const std::unique_ptr<FrameBuffer> &buffer : outputBuffers_) {
            if (output->queueBuffer(buffer.get())) {
                    std::cout << "Failed to queue output buffer" << std::endl;
            }
            /* mmap the buffer */
            MappedFrameBuffer map(buffer.get(), MappedFrameBuffer::MapFlag::ReadWrite);
            assert(map.isValid());
            assert(map.planes()[0].size() == bbb_splash_rgb_len);
            memcpy(map.planes()[0].data(), bbb_splash_rgb, map.planes()[0].size());
            inputMappedBuffer_.push_back(std::move(map));
    }

    ret = capture->streamOn();
    if (ret) {
            cerr << "Failed to streamOn capture" << endl;
    }

    ret = output->streamOn();
    if (ret) {
            cerr << "Failed to streamOn output" << endl;
    }
```

```
    ret = capture->allocateBuffers(bufferCount, &captureBuffers_);
    if (ret < 0) {
            cerr << "Failed to allocate Capture Buffers" << endl;
            return -1;
    }

    ret = output->allocateBuffers(bufferCount, &outputBuffers_);
    if (ret < 0) {
            cerr << "Failed to allocate Output Buffers" << endl;
            return -1;
    }
```

```
        Timer timeout;
        timeout.start([illegible]);
        while (timeout.isRunning()) {
                dispatcher->processEvents();
                if (captureFrames_ > [illegible])
                        break;
        }

        cerr << "Output " << outputFrames_ << " frames" << std::endl;
        cerr << "Captured " << captureFrames_ << " frames" << std::endl;

        if (captureFrames_ < [illegible]) {
                cerr << "Failed to capture 10 frames within timeout." << std::endl;
                return -1;
        }

        ret = capture->streamOff();
        if (ret) {
                cerr << "Failed to streamOff the capture device." << std::endl;
                return -1;
        }

        ret = output->streamOff();
        if (ret) {
                cerr << "Failed to streamOff the output device." << std::endl;
                return -1;
        }

        return 0;
```

Figure 33: Application Run