Augmented reality navigation

Martin Jaros

Contents

Preface

Introduction

1 Augmented reality

1.1 Design goals

Design goals and project overview

1.2 Hardware limitations

Developing an application for an embedded device faces a basic problem, as there are big differences between these devices it is hard to support the hardware and make the application portable. In order to reuse code and reduce application size, libraries are generally used. To provide enough abstraction operating system is used. There are many kernels specially tailored for embedded applications such as FreeRTOS, Linux or proprietary VxWorks, Windows CE. Linux kernel has been chosen for this project.

Advantages of the Linux kernel

- free and open-source, well documented
- highly configurable and portable
- highly standardized, POSIX compliant
- large amount of drivers, good manufacturer support
- great community support, many tutorials

Disadvantages of the Linux kernel

- large code base
- steep learning curve
- high hardware requirements

While the application is designed to be highly portable depending only on the kernel itself, several devices has been chosen as the reference.

OMAP4460 application processor[1]

- two ARM Cortex-A9 SMP general-purpose processors
- IVA 3 video accelerator, 1080p capable
- image signal processor, 20MP capable
- SGX540 3D graphics accelerator, OpenGL ES 2.0 compatible
- HDMI v1.3 video output

MPU-9150 motion tracking device[2]

- embedded MPU-6050 3-axis gyroscope and accelerometer
- embedded AK8975 3-axis digital compass
- fully programmable, I^2C interface

OV5640 image sensor[3]

- 1080p, 5MP resolution
- raw RGB or YUV output

[1] OMAP4460 Technical reference manual
http://www.ti.com/litv/pdf/swpu235aa

[2] MPU-9150 Product specification
http://invensense.com/mems/gyro/documents/PS-MPU-9150A-00v4_3.pdf

[3] OV5640 Product brief
http://www.ovt.com/download_document.php?type=sensor&sensorid=93

2 Application

2.1 Linux kernel

Programs running in Linux are divided into two groups, *kernel-space* and *user-space*. Only kernel and its runtime modules are allowed to execute in *kernel-space*, while all other programs runs as processes in *user-space*.

kernel-space

- real-time CPU usage
- physical memory access

user-space

- scheduled CPU usage
- virtual memory access

In Linux each process runs in a sandbox, isolated from the rest of the system. Processes access virtual memory unique to them, they cannot access memory assigned for other processes nor memory managed by the kernel. Their execution is not real-time, but they are assigned restricted processor time by the kernel. They may communicate with outside environment by several means

- Arguments and environment variables
- Standard input, output and error output
- Virtual File System
- Signals
- Sockets
- Memory mapping

Each process is ran with several arguments in a specific environment with three default file descriptors. For example running

```
VARIABLE=value ./executable argument1 argument2 <input 1>output 2>error
```

will execute *executable* with environment variable *VARIABLE* of value *value* with two arguments *argument1* and *argument2*. Standard input will be read from file *input* while regular output will be written to file *output* and error output to file *error*. This process may further communicate by accessing files in the Virtual File System, kernel may expose useful process information for example via `procfs` file-system usually mounted at `/proc`. Other types of communication are signals (which may be sent between processes or by kernel) and network sockets. With internal network loop-back device, network style inter process communication is possible using standard protocols (UDP, TCP, . . .). Memory mapping is a way to request access to some part of the physical memory.

Processes may run with numerous threads, each thread has preemptively scheduled execution. Threads share memory within a process, memory access to these shared resources must done with care to avoid race conditions and data corruption. Kernel provides *mutex* objects to lock threads and avoid simultaneous memory access. Each shared resource should be attached to a *mutex*, which is locked during access to this resource. Thread must not lock *mutex* while still holding lock to this or any other *mutex* in order to avoid dead-locking.

Source example for using posix threads

```c
#include <stdio.h>
#include <pthread.h>

/*
 * In this example two threads are created, they access shared resources (stdin and stdout)
 * Mutex is used to restrict this access and avoid memory corruption,
 * so only one thread may access the shared resource at one time.
 */

void *worker1(void *arg)
{
    pthread_mutex_t *mutex = (pthread_mutex_t*)arg;
    static char buffer[64];

    // Lock mutex to restrict access to stdin and stdout
    pthread_mutex_lock(mutex);
    printf("This is worker 1, enter something: ");
    scanf("%64s", buffer);
    pthread_mutex_unlock(mutex);

    return (void*)buffer;
}

void *worker2(void *arg)
{
    pthread_mutex_t *mutex = (pthread_mutex_t*)arg;
    static char buffer[64];

    // Lock mutex to restrict access to stdin and stdout
    pthread_mutex_lock(mutex);
    printf("This is worker 2, enter something: ");
    scanf("%64s", buffer);
    pthread_mutex_unlock(mutex);
```

```c

    return (void*)buffer;
}

int main()
{
    pthread_mutex_t mutex;
    pthread_t thread1, thread2;
    char *retval1, *retval2;

    // Initialize two threads with shared mutex, use default parameters
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&thread1, NULL, worker1, (void*)&mutex);
    pthread_create(&thread2, NULL, worker2, (void*)&mutex);

    // Wait for both threads to finish and display results
    pthread_join(thread1, (void**)&retval1);
    pthread_join(thread2, (void**)&retval2);
    printf("Thread 1 returned with `%s`.\n", retval1);
    printf("Thread 2 returned with `%s`.\n", retval2);

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

Linux kernel has monolithic structure, so all device drivers resides in the kernel. From application point of view, this means that all peripheral access must be done through the standard library and Virtual File System. Individual devices are accessible as device files defined by major and minor number typically located at `/dev`. These files could be created automatically by kernel (`devtmpfs` file-system), by daemon (`udev(8)`)), or manually by `mknod(1)`. Complete kernel device model is exported as `sysfs` file-system and typically mounted at `/sys`.

Function name	Access type	Typical usage
`select()`, `poll()`	event	Synchronization, multiplexing, event handling
`ioctl()`	structure	Configuration, register access
`read()`, `write()`	stream	Raw data buffers, byte streams
`mmap()`	block	High throughput data transfers

Table 1: Available functions for working with device file descriptors

For example let's assume a generic peripheral device connected by the I^{2}C bus. First, to tell kernel there is such a device, the `sysfs` file-system may be used

```
echo $DEVICE_NAME $DEVICE_ADDRESS > /sys/bus/i2c/devices/i2c-1/new_device
```

This should create a special file in `/dev`, which should be opened by `open()` to get a file descriptor for this device. Device driver may export some *ioctl* requests, each request is defined by a number and a structure passed between the application and the kernel. Driver should define requests for controlling the device, maybe accessing its internal registers and configuring a data stream. Each request is called by

```
ioctl(fd, REQNUM, &data);
```

where *fd* is the file descriptor, *REQNUM* is the request number defined in the driver header and *data* is the structure passed to the kernel. This request will be synchronously processed by the kernel and the result stored in the *data* structure. Let's assume this devices has been configured to stream an integer value every second to the application. To synchronize with this timing application may use

```
struct pollfd fds = {fd, POLLIN};
poll(&fds, 1, -1);
```

which will block infinitely until there is a value ready to be read. To actually read it,

```
int buffer[1];
ssize_t num = read(fd, buffer, sizeof(buffer));
```

will copy this value to the buffer. Copying causes performance issues if there are very large amounts of data. To access this data directly without copying them, application has to map physical memory used by the driver. This allows for example direct access to a DMA channel, it should be noted that this memory may still be needed by kernel, so there should be some kind of dynamic access restriction, possibly via *ioctl* requests (this would be driver specific).

2.2 Video subsystem

Video support in Linux kernel is maintained by the LinuxTV[4] project, it implements the *videodev2* kernel module and defines the *V4L2* interface. Modules are part of the mainline kernel at `drivers/media/video/*` with header

[4] LinuxTV project
http://linuxtv.org/

`linux/videodev2.h`. The core module is enabled by the *VIDEO_V4L2* configuration option, specific device drivers should be enabled by their respective options. *V4L2* is the latest revision and is the most widespread video interface throughout Linux, drives are available from most hardware manufactures and usually mainlined or available as patches. The Linux Media Infrastructure API[5] is a well documented interface shared by all devices. It provides abstraction layer for various device implementations, separating the platform details from the applications. Each video device has its device file and is controlled via *ioctl* calls. For streaming standard I/O functions are supported, but the memory mapping is preferred, this allows passing only pointers between the application and the kernel, instead of unnecessary copying the data around.

Name	**Description**
VIDIOC_QUERYCAP	Query device capabilities
VIDIOC_G_FMT	Get the data format
VIDIOC_S_FMT	Set the data format
VIDIOC_REQBUFS	Initiate memory mapping
VIDIOC_QUERYBUF	Query the status of a buffer
VIDIOC_QBUF	Enqueue buffer to the kernel
VIDEOC_DQBUF	Dequeue buffer from the kernel
VIDIOC_STREAMON	Start streaming
VIDIOC_STREAMOFF	Stop streaming

Table 2: ioctl calls defined in `linux/videodev2.h`

Application sets the format first, then requests and maps buffers from the kernel. Buffers are exchanged between the kernel and the application. When the buffer is enqueued, it will be available for the kernel to capture data to it. When the buffer is dequeued, kernel will not access the buffer and application may read the data. After all buffer are enqueued application starts the stream. Polling is used to wait for the kernel until it fills the buffer, buffer should not be accessed simultaneously by the kernel and the application. After processing the buffer, application should return it back to the kernel queue. Note that buffers should be properly unmapped by the application after stopping the stream.

Source example for simple video capture

[5] Linux Media Infrastructure API
http://linuxtv.org/downloads/v4l-dvb-apis/

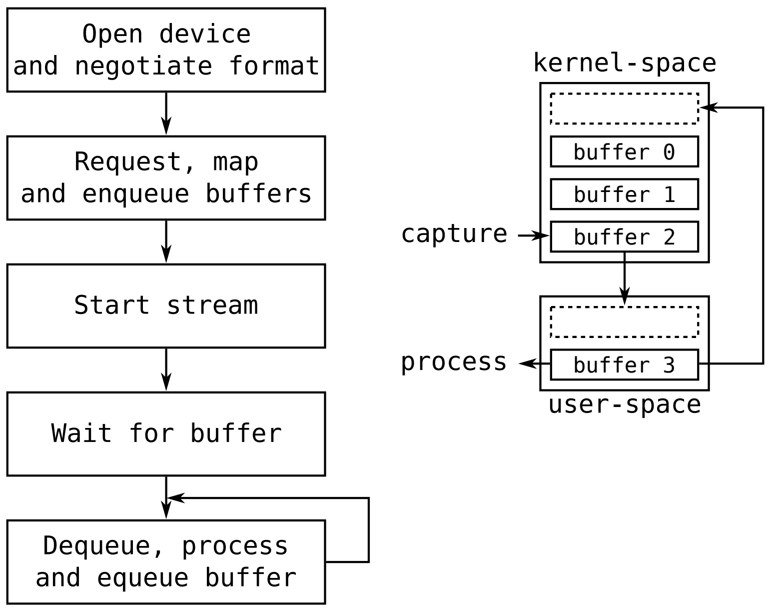

Figure 1: V4L2 capture

```
#include <fcntl.h>
#include <unistd.h>
#include <poll.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <linux/videodev2.h>

int main()
{
    // Open device
    int fd = open("/dev/video0", O_RDWR | O_NONBLOCK);

    // Set video format
    struct v4l2_format format =
    {
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .fmt =
        {
            .pix =
            {
                .width = 320,
                .height = 240,
                .pixelformat = V4L2_PIX_FMT_RGB32,
```

```c
                .field = V4L2_FIELD_NONE,
            },
        },
    };
    ioctl(fd, VIDIOC_S_FMT, &format);

    // Request buffers
    struct v4l2_requestbuffers requestbuffers =
    {
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .memory = V4L2_MEMORY_MMAP,
        .count = 4,
    };
    ioctl(fd, VIDIOC_REQBUFS, &requestbuffers);
    void *pbuffers[requestbuffers.count];

    // Map and enqueue buffers
    int i;
    for(i = 0; i < requestbuffers.count; i++)
    {
        struct v4l2_buffer buffer =
        {
            .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
            .memory = V4L2_MEMORY_MMAP,
            .index = i,
        };
        ioctl(fd, VIDIOC_QUERYBUF, &buffer);
        pbuffers[i] = mmap(NULL, buffer.length, PROT_READ | PROT_WRITE, MAP_SHARED, fd, buff
        ioctl(fd, VIDIOC_QBUF, &buffer);
    }

    // Start stream
    enum v4l2_buf_type buf_type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    ioctl(fd, VIDIOC_STREAMON, &buf_type);

    while(1)
    {
        // Synchronize
        struct pollfd fds =
        {
            .fd = fd,
            .events = POLLIN
        };
        poll(&fds, 1, -1);

        // Dump buffer to stdout
```

```c
        struct v4l2_buffer buffer =
        {
            .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
            .memory = V4L2_MEMORY_MMAP,
        };
        ioctl(fd, VIDIOC_DQBUF, &buffer);
        write(1, pbuffers[buffer.index], buffer.bytesused);
        ioctl(fd, VIDIOC_QBUF, &buffer);
    }
}
```

The image format is specified using the little-endian four-character code (FOURCC). V4L2 defines several formats and provides `v4l2_fourcc()` macro to create a format code from four characters. As described later in the graphics subsystem chapter, graphics uses natively the *RGB4* format. This format is defined as a single plane with one sample per pixel and four bytes per sample. These bytes represents red, green and blue channel values respectively. Image size is therefore $width \cdot height \cdot 4$ bytes. Many image sensors however support YUV color-space, for example the *YU12* format. This one is defined as three planes, the first plane with one luminance sample per pixel and the second and third plane with one chroma sample per four pixels (2 pixels per row, interleaved). Each sample has one byte, this format is also referenced as *YUV 4:2:0* and its image size is $width \cdot height \cdot 1.5$ bytes. The luminance and chroma of a pixel is defined as

(1) $E_Y = W_R \cdot E_R + (1 - W_R - W_B) \cdot E_G + W_B \cdot E_B$

(2) $E_{C_r} = \frac{0.5(E_R - E_Y)}{1 - W_R}$

(3) $E_{C_b} = \frac{0.5(E_B - E_Y)}{1 - W_B}$

where $\mathrm{E_R}$, $\mathrm{E_G}$, $\mathrm{E_B}$ are normalized color values and $\mathrm{W_R}$, $\mathrm{W_B}$ are their weights. ITU-R Rec. BT.601[6] defines weights as 0.299 and 0.114 respectively, it also defines how they are quantized

(4) $Y = 219E_Y + 16$

(5) $C_r = 224E_{C_r} + 128$

(6) $C_b = 224E_{C_b} + 128$

To calculate R, G, B values from Y, Cr, Cb values, inverse formulas must be used

[6] ITU-R Recommendation BT.601-7
http://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.601-7-201103-I!!PDF-E.pdf

(7) $E_Y = \frac{Y-16}{219}$

(8) $E_{C_r} = \frac{C_r-128}{224}$

(9) $E_{C_b} = \frac{C_b-128}{224}$

(10) $E_R = E_Y + 2E_{C_r}(1 - W_R)$

(11) $E_G = E_Y - 2E_{C_r}\frac{W_R-{W_R}^2}{W_G} - 2E_{C_b}\frac{W_B-{W_B}^2}{W_G}$

(12) $E_B = E_Y + 2E_{C_b}(1 - W_B)$

GLSL implementation of the YUV to RGB conversion (see graphics subsystem chapter for description of GLSL)

```
uniform sampler2D texY, texU, texV;
varying vec2 texPos;

void main()
{
    float y = texture2D(texY, texPos).a * 1.1644 - 0.062745;
    float u = texture2D(texU, texPos / 2).a - 0.5;
    float v = texture2D(texV, texPos / 2).a - 0.5;

    gl_FragColor = vec4(
        y + 1.596 * v,
        y - 0.39176 * v - 0.81297 * u,
        y + 2.0172 * u,
        1.0);
}
```

v4l2 loopback, H.264 decode

2.3 Graphics subsystem

Graphics stack, OpenGL ES 2.0

2.4 Inertial measurement subsystem

Industrial I/O module and drivers, DCM algorithm

2.5 Satellite navigation subsystem

TTY module, stty, socat, GPS, NMEA 0183

3 Hardware

Existing modules, designs

4 Conclusion

Conclusion

References

References