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^{2}C 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

```
1   #include <stdio.h>
2   #include <pthread.h>
3
4   void *worker1(void *arg)
5   {
6       pthread_mutex_t *mutex = (pthread_mutex_t*)arg;
7       static char buffer[64];
8
9       // Lock mutex to restrict access to stdin and stdout
10      pthread_mutex_lock(mutex);
11      printf("This is worker 1, enter something: ");
12      scanf("%64s", buffer);
13      pthread_mutex_unlock(mutex);
14
15      return (void*)buffer;
16  }
17
18  void *worker2(void *arg)
19  {
20      pthread_mutex_t *mutex = (pthread_mutex_t*)arg;
21      static char buffer[64];
22
23      // Lock mutex to restrict access to stdin and stdout
24      pthread_mutex_lock(mutex);
25      printf("This is worker 2, enter something: ");
26      scanf("%64s", buffer);
27      pthread_mutex_unlock(mutex);
28
29      return (void*)buffer;
30  }
31
32  int main()
33  {
34      pthread_mutex_t mutex;
```

7

```
35      pthread_t thread1, thread2;
36      char *retval1, *retval2;
37
38      // Initialize two threads with shared mutex, use default parameters
39      pthread_mutex_init(&mutex, NULL);
40      pthread_create(&thread1, NULL, worker1, (void*)&mutex);
41      pthread_create(&thread2, NULL, worker2, (void*)&mutex);
42
43      // Wait for both threads to finish and display results
44      pthread_join(thread1, (void**)&retval1);
45      pthread_join(thread2, (void**)&retval2);
46      printf("Thread 1 returned with `%s`.\n", retval1);
47      printf("Thread 2 returned with `%s`.\n", retval2);
48
49      pthread_mutex_destroy(&mutex);
50      return 0;
51  }
```

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^2C 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 `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.

---

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

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

```c
#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
```
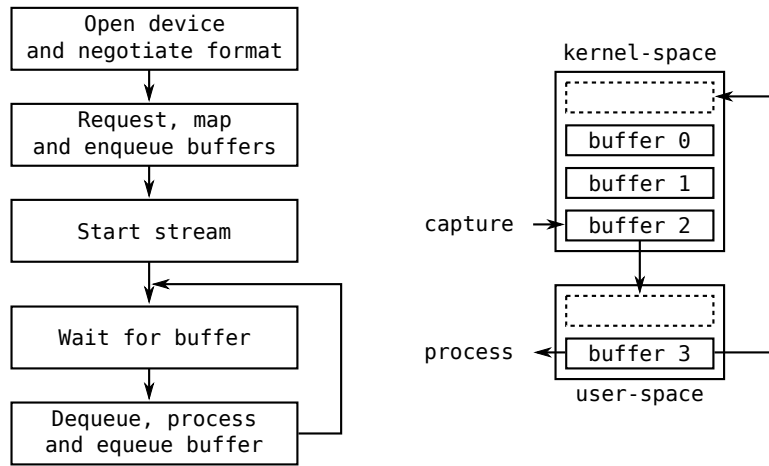
Figure 1: V4L2 capture

```
14    struct v4l2_format format =
15    {
16        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
17        .fmt =
18        {
19            .pix =
20            {
21                .width = 320,
22                .height = 240,
23                .pixelformat = V4L2_PIX_FMT_RGB32,
24                .field = V4L2_FIELD_NONE,
25                .colorspace = V4L2_COLORSPACE_SMPTE170M,
26            },
27        },
28    };
29    ioctl(fd, VIDIOC_S_FMT, &format);
30
31    // Request buffers
32    struct v4l2_requestbuffers requestbuffers =
33    {
34        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
35        .memory = V4L2_MEMORY_MMAP,
36        .count = 4,
37    };
38    ioctl(fd, VIDIOC_REQBUFS, &requestbuffers);
39    void *pbuffers[requestbuffers.count];
40
```

```
41      // Map and enqueue buffers
42      int i;
43      for(i = 0; i < requestbuffers.count; i++)
44      {
45          struct v4l2_buffer buffer =
46          {
47              .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
48              .memory = V4L2_MEMORY_MMAP,
49              .index = i,
50          };
51          ioctl(fd, VIDIOC_QUERYBUF, &buffer);
52          pbuffers[i] = mmap(NULL, buffer.length,
53                          PROT_READ | PROT_WRITE, MAP_SHARED,
54                          fd, buffer.m.offset);
55          ioctl(fd, VIDIOC_QBUF, &buffer);
56      }
57
58      // Start stream
59      enum v4l2_buf_type buf_type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
60      ioctl(fd, VIDIOC_STREAMON, &buf_type);
61
62      while(1)
63      {
64          // Synchronize
65          struct pollfd fds =
66          {
67              .fd = fd,
68              .events = POLLIN
69          };
70          poll(&fds, 1, -1);
71
72          // Dump buffer to stdout
73          struct v4l2_buffer buffer =
74          {
75              .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
76              .memory = V4L2_MEMORY_MMAP,
77          };
78          ioctl(fd, VIDIOC_DQBUF, &buffer);
79          write(1, pbuffers[buffer.index], buffer.bytesused);
80          ioctl(fd, VIDIOC_QBUF, &buffer);
81      }
82  }
```

The image format is specified using the little-endian four-character code

(FOURCC). V4L2 defines[6] 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 E_R, E_G, E_B are normalized color values and W_R, W_B are their weights. ITU-R Rec. BT.601[7] 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

(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)　$E_B = E_Y + 2E_{C_b}(1 - W_B)$

---

[6]V4L2 image format definitions
http://linuxtv.org/downloads/v4l-dvb-apis/pixfmt.html
[7]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

It should be noted that not all devices may use the BT.601 recommendation, V4L2 refers to it as *V4L2_COLORSPACE_SMPTE170M* in the *VIDIOC_S_FMT* request structure.

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

```
1  uniform sampler2D texY, texU, texV;
2  varying vec2 texCoord;
3
4  void main()
5  {
6      float y = texture2D(texY, texCoord).a * 1.1644 - 0.062745;
7      float u = texture2D(texU, texCoord / 2).a - 0.5;
8      float v = texture2D(texV, texCoord / 2).a - 0.5;
9
10     gl_FragColor = vec4(
11         y + 1.596 * v,
12         y - 0.39176 * v - 0.81297 * u,
13         y + 2.0172 * u,
14         1.0);
15 }
```

There is a module *v4l2loopback*[8] which creates a video loop-back device, similar to network loop-back, allowing piping two video applications together. This is very useful not only for testing, but also for implementation of intermediate decoders. GStreamer[9] is a powerful multimedia framework widespread in Linux distributions, composed of a core infrastructure and hundreds of plug-ins. This command will create synthetic *RGB4* video stream for the application, useful for testing

```
modprobe v4l2loopback
gst-launch videotestsrc pattern=solid-color foreground-color=0xE0F0E0 ! \
"video/x-raw,format=RGBx,width=800,height=600,framerate=20/1" \
!  v4l2sink device=/dev/video0
```

Texas Instruments distributes a meta package[10] for their OMAP platform featuring all required modules and DSP firmware. This includes kernel modules for *SysLink* inter-chip communication library, *Distributed Codec Engine* library and *ducati* plug-in for GStreamer. With the meta-package installed, it is very easy and efficient to implement mainstream encoded video formats. For example

---

[8]V4L2 loopback device module
https://github.com/umlaeute/v4l2loopback
[9]GStreamer home page
http://gstreamer.freedesktop.org/
[10]TI OMAP trunk PPA
https://launchpad.net/~tiomap-dev/+archive/omap-trunk

following command will create GStreamer pipeline to receive video payload over a network socket from an IP camera, decode it and push it to the loop-back device for the application. MPEG-4 AVC (H.264) decoder of the IVA 3 is used in this example.

```
modprobe v4l2loopback
gst-launch udpsrc port=5004 caps=\
"application/x-rtp,media=video,payload=96,clock-rate=90000,encoding-name=H264" \
! rtph264depay ! h264parse ! ducatih264dec ! v4l2sink device=/dev/video0
```

On OMAP4460 this would consume only about 15% of the CPU time as the decoding is done by the IVA 3 video accelerator in parallel to the CPU which only passes pointers around and handles synchronization. Output format is *NV12* which is similar to *YU12* format described earlier, but there is only one chroma plane with two-byte samples, first byte being the U channel and the second byte the V channel, sampling is same 4:2:0. The YUV to RGB color space conversion must take place here, preferably implemented on the GPU as described above.

Cortex-A9 cores on the OMAP4460 also have the NEON co-processor, capable of vector floating point math. Although not very supported by the GCC C compiler, there are many assembly written libraries implementing coders with the NEON acceleration. For example the *libjpeg-turbo*[11] library is implementing the *libjpeg* interface. It is useful for USB cameras, as the USB throughput is not high enough for raw high definition video, but is sufficient with JPEG coding (as most USB cameras supports JPEG, but does not support H.264). 1080p JPEG stream decoded with this library via its GStreamer plug-in will consume about 90% of the single CPU core time (note that there are two CPU cores available). However, comparable to the AVC, JPEG encoding will cause visible quality degradation in the raw stream (video looks grainy).

2.3 Graphics subsystem

Graphics stack, OpenGL ES 2.0

---

[11]libjpeg-turbo library
http://www.libjpeg-turbo.org/

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

[1] BIMBER, O.; RASKAR, R. Spatial augmented reality: merging real and virtual worlds. Wellesley: A K Peters, 2005, 369 p. ISBN 15-688-1230-2.

[2] JAZAR, Reza N. Theory of applied robotics: kinematics, dynamics, and control. 2nd ed. New York: Springer, 2010, 883 p. ISBN 978-1-4419-1749-2.

[3] KENNEDY, Melita. Understanding map projections. Redlands: ESRI, 2000, 110 p. ISBN 15-894-8003-1.

[4] Texas Instruments. OMAP 4460 Multimedia Device [online]. 2012 - [cit. 8. Nov 2012]. Available: http://www.ti.com/product/omap4460.

[5] InvenSense. MPU-9150 Nine-Axis MEMS MotionTracking™ Device [online]. 2013 - [cit. 10. Nov 2013]. Available: http://www.invensense.com/mems/gyro/mpu9150.html.

[6] Consultative Committee on International Radio. ITU-R Recommendation BT.601 [online]. 2011 - [cit. 10. Nov 2013]. Available: http://www.itu.int/rec/R-REC-BT.601/en.

[7] Khronos Group. OpenGL ES 2.X - for Programmable Hardware [online]. 2013 - [cit. 10. Nov 2013]. Available: http://www.khronos.org/opengles/2_X/.

[8] National Marine Electronics Association. NMEA 0183 [online]. 2008 - [cit. 10. Nov 2013]. Available: http://www.nmea.org/content/nmea_standards/nmea_0183_v_410.asp.