# Developing a Linux Driver for the StrongARM Platform

Greg Reynolds

September 2, 2003

# Contents

## 1 Introduction

The code presented here was developed to take data from a CMOS camera module connected to a StrongARM microprocessor running GNU/Linux. The concepts demonstrated include:

- Reading from hardware connected as a static/flash memory;
- Communicating with hardware using $I^2C$ at the device driver level;
- Handling interrupts generated by external hardware.

Much of the code here is contextually, though largely not hardware, dependent on the project for which it was developed.

## 2 Device Driver

There are generally three types of device driver under Linux; `char`, `block` and `network` drivers. A `char` driver is the simplistic byte-shifting type and it this type that is used here. A `char` device is accessed in the traditional Unix way through an entry in the `/dev` directory. The driver must provide a standard set of functions which will correspond to operations such as `open`, `read`, `ioctl` all of which happen to the file descriptor corresponding to the open `/dev` device. Taking the following code for example:

Listing 1: Accessing a device, via it's driver, from userland.

```
int fd;
unsigned char ucBuff[PREDEFINED_SIZE];

/* open the device */
fd = open("/dev/my_device", O_RDWR);

/* read back some data from it */
read(fd, ucBuff, PREDEFINED_SIZE);

/* send some device specific commands */
ioctl(fd, MOVE_ROBOT_ARM, NULL);

...
```

### 2.1 Developing as a Module

While it may, in many cases, be desirable to provide static-kernel code for a driver, during development recompiling and rebooting to test the code every time is time-consuming. It is therefore highly desirable to develop the code as a module. [1]

When writing a driver, standard C function calls are not available, (i.e. no `glibc`). Only functions provided by the kernel may be used. To see a list of all the functions provided by the kernel, examine `/proc/ksyms`. The output format when making the module is standard ELF object code, all the linking happens at run time.

To load a module, i.e. make it available to run use the `insmod` command. To check which modules are loaded use the `lsmod` command. To unload a module run the `rmmod` command.

[1]Note also that it is preferable to develop at the console (i.e. not X), otherwise output from the driver will be suppressed and only available in the log files.

## 2.2 Module Skeleton

When a module is loaded and unloaded, the functions init_module and cleanup_module respectively, are called. The names of these functions are by default those specified, however it is possible to specify the names corresponding to these functions using the module_init and module_exit macros. For example:

Listing 2: Skeleton module code.

```
/* tell the kernel what the init and cleanup functions are */
#include<linux/config.h>
#include<linux/module.h>
#include<linux/kernel.h>

#ifdef MODVERSIONS
#include <linux/modversions.h>
#endif

#define DRIVER_NAME "ov7620_module"

module_exit(ov7620_cleanup);
module_init(ov7620_init);

/*
 * This gets called when the module is loaded.
 */
int ov7620_init(void)
{
  printk(KERN_INFO "%s␣loaded.", DRIVER_NAME);
}

/*
 * This gets called when the module is unloaded.
 */
void ov7620_cleanup(void)
{
...
}
```

Note the use of the printk function, a function exported by the kernel which behaves like printf.

### 2.2.1 Module Versions

Once compiled, embedded in the object code is a version number. This version number is the same as the source of the kernel, so the version of the kernel that the module must run under, must be compiled from the same source that the module is based on. A comprehensive discussion of versioning issues can be found in [1]. The file linux/modversions.h must exist, to create this run make dep on the kernel sources. If make dep fails then the chances are this file will not exist and the symbol names in the module won't match the kernel. To fix make dep involved (in this case) editing a variety of Makefiles throughout the kernel source tree.

### 2.2.2 Compiling the Code

```
KERNDELDIR=/home/gmr/kernels/linux
CC=/skiff/local/bin/arm-linux-gcc
```

```
LD=/skiff/local/bin/arm-linux-ld
NM=/skiff/local/bin/arm-linux-nm

include /home/gmr/kernels/linux/.config

CFLAGS+=-c -Wall -D__KERNEL__ -DMODULE -I/home/gmr/kernels/linux/include -O
LFLAGS+=

ifdef CONFIG_SMP
 CFLAGS+=-D__SMP__ -DSMP
endif


all: ov7620.o
        $(NM) > ./syms

clean:
        rm ov7620.o
        rm syms

ov7620.o: ov7620.c
        $(CC) $(CFLAGS) ov7620.c -o ov7620.o
```

This shows the basic `Makefile` requirements to build the module.

## 2.3 How the Driver Works

Figure 2.3 shows how the driver works. Once the driver and camera have been initialized, nothing happens until the renderer program is run. Once the renderer is run it enables an interrupt in the camera driver, as soon as this interrupt happens data is copied from the camera to the driver's buffer. The interrupt is then disabled and the renderer program converts the data to a real color space and plots it. Once it has been plotted the interrupt is re-enabled. In order for the image to be stable, because of the slow FIFO hardware, the frame rate had to be reduced to about 2FPS.

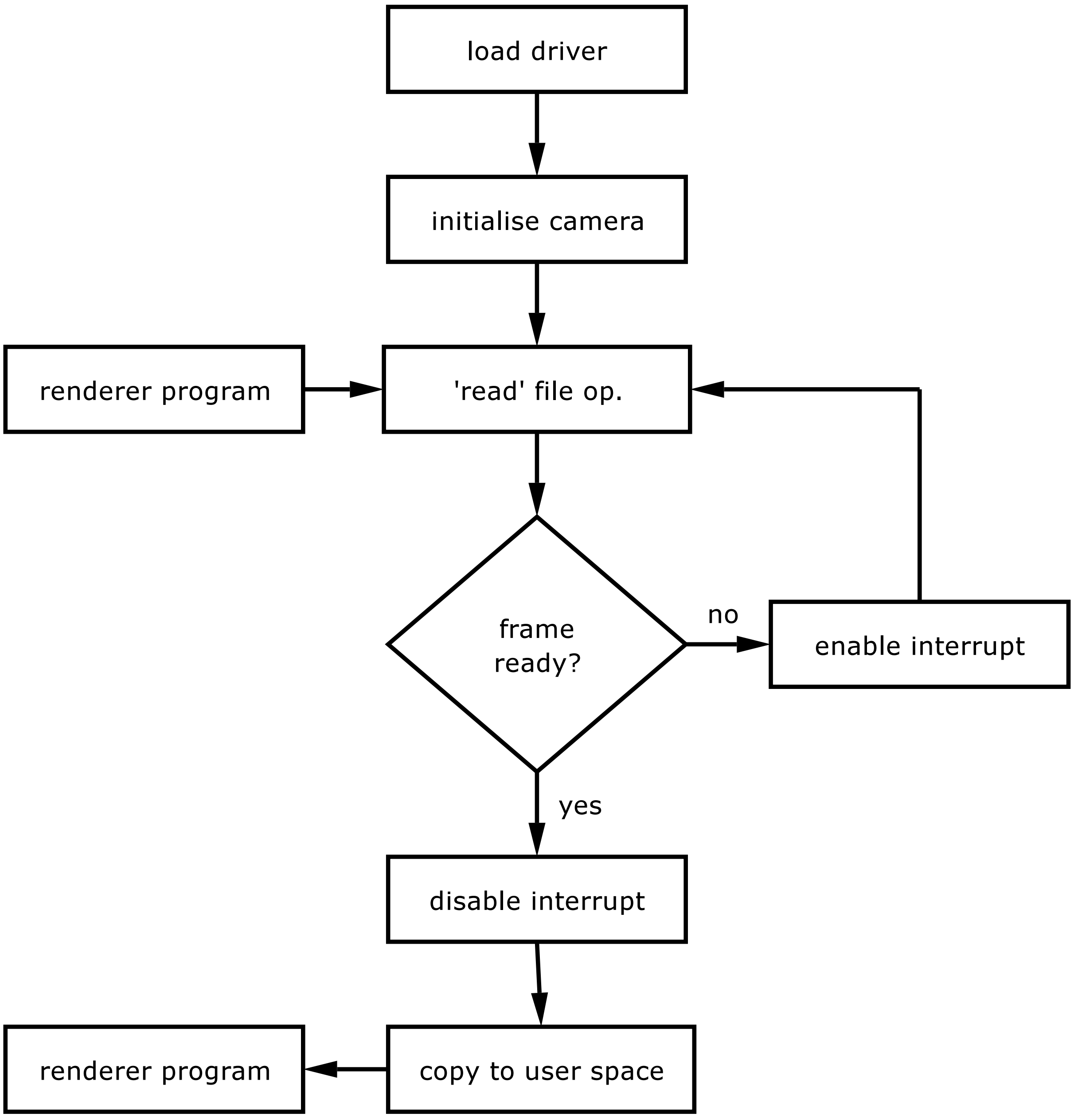

Figure 1: A flow chart showing the general operation of the driver.

## 2.4 Camera Data Format

The Omnivision OV7620 camera module has the following features that were made use of in this project:

- Variable resolution up to 640x480.
- Full color capture, output in YUV 4:2:2 format.
- Output rate at variable rates, up to 60FPS.
- 16-bit interface bus.
- I$^2$C interface for configuration registers.

Data comes off the camera in a raster format, i.e. line after line in vertical sequence. Although the module can output a variety of formats, the YUV 4:2:2 format was chosen. In this mode the camera outputs a luminance (Y) value for every pixel that is captured. The chrominance values (U and V) are shared for each pair of pixels. For example, $pixel(0,0)$ has data $(Y_0, U_{0,1}, V_{0,1})$ and $pixel(0,1)$ has $(Y_1, U_{0,1}, V_{0,1})$. This is illustrated below:

| Clock Pulse | 0 | 1 | 2 | 3 | ... |
|---|---|---|---|---|---|
| Y (lower byte) | $Y_0$ | $Y_1$ | $Y_2$ | $Y_3$ | ... |
| UV (upper byte) | $U_{0,1}$ | $V_{0,1}$ | $U_{2,3}$ | $V_{2,3}$ | ... |

The camera also provides a variety of synchronization signals so that the master device can identify when the data is valid.:

- `VSYNC` is asserted at the start of every frame.
- `HREF` is asserted at the start of every row of data (e.g. there will be 480 `HREF` pulses per frame).
- A synchronization word of value 0x8010 is sent at the start and end of every row of data.

Once the data comes off the camera it needs to be converted from YUV to RGB. The conversion formula is shown in (1).

$$\begin{aligned} R &= U + Y \\ B &= V + Y \\ G &= 1.11Y - 0.34R - 0.12B \end{aligned} \tag{1}$$

Doing this as a floating point operation is very slow. An integer conversion method was used.

## 2.5 Interrupt Handlers

Interrupt service routines (ISR) are called when the specify interrupt they are assigned to is triggered. In this case of this driver the only interrupt used is the `VSYNC` signal. Every time a frame is ready the camera generates a `VSYNC` interrupt.

Listing 3: Installing an interrupt handler.

```
/* bottom half for VSYNC ISR */
DECLARE_TASKLET(ov7620_copy, /* name of tasklet, arbitrary */
                ov7620_mem_transfer, /* tasklet function */
                0); /* flags, leave as 0 */

/*
```

```
 * This is the 'open' function for the /dev/omni node.
 */
static int ov7620_open(struct inode* pinode, struct file* pfile)
{
  MOD_INC_USE_COUNT;

  /* Setup all the GPIO for interrupt input, FIFO reset and LCD output:
   * GPIO 21 is VSYNC from OV7620 interrupt (rising-edge).
   * GPIO 11 is FIFO reset (active low).
   * GPIO 20 is camera reset. */
  GPDR = GPIO_GPIO11 | GPIO_GPIO20 | GPIO_GPIO2 |
    GPIO_GPIO3 | GPIO_GPIO4 | GPIO_GPIO5 |
    GPIO_GPIO6 | GPIO_GPIO7 | GPIO_GPIO8 |
    GPIO_GPIO9;


  set_GPIO_IRQ_edge(GPIO_GPIO21, GPIO_RISING_EDGE);

  /* request interrupt for VSYNC */
  if( request_irq(IRQ_GPIO21, &ov7620_interrupt, 0, NULL, NULL) != 0 ) {
    printk(KERN_WARNING "Couldn't␣get␣IRQ.");
    return -1;
  }

  return 0;
}

/*
 * This is triggered on the VSYNC interrupt, it resets the FIFO and
 * then calls the tasklet to do the real hardwork.
 */
static void ov7620_interrupt(int irq, void* dev_id, struct pt_regs* pregs)
{
  /* reset the FIFO so that it buffers the video data */
  reset_fifo();

  tasklet_schedule(&ov7620_copy);
}


/*
 * This is where the frame is copied over from the FIFO buffer to
 * our buffer.
 */
static void ov7620_mem_transfer(unsigned long unused)
{
  ...
}
```

The ISR is broken into two parts, the upper half (the real ISR) and the bottom half. The bottom half is executed at a lower priority, this is essential so that the CPU does not become tied up. In the upper half of an interrupt handler, for example, it would not be desirable to execute code that took 5s, because of the need to deal with other IRQs.

It is also useful to enable and disable a particular IRQ. In this case, for example, the VSYNC IRQ was disabled by the read function, so that the frame data could be copied to userland without it being overwritten by the next frame. For example:

Listing 4: Enabling and disabling IRQs.

```
disable_irq(IRQ_GPIO21);

/* work that needs to be done before we can take another frame */
...

enable_irq(IRQ_GPIO21);
```

There are flags that can be passed to request_irq to stipulate that it should be run without other interrupts. More information is available in [1].

## 2.6 Reading from the Hardware

In this case, the cameras was connected via two FIFOs and several latches to static memory bank 3. Accessing the camera involves reading from the memory at this location. To read an entire frame involves reading from the same location 640x480x(6/4) times.

Before the memory can be accessed and address needs to be mapped from the physical address (in this case 0x19000000) to a virtual address that the kernel understands. This is shown in Listing 5.

Listing 5: Mapping from physical to virtual addresses.

```
#define CAMERA_BASE 0x19000000

/* this is the base address for all operations */
static void* g_io;

/*
 * This gets called when the module is loaded.
 */
int ov7620_init(void)
{
  ...

  /* get the virtual address from the physical address */
  g_io = (void*)ioremap(CAMERA_BASE, 2);

  ...
}

/* wrapper function for reading data */
inline static unsigned long ov7620_readw(long lunused)
{
  /* check that the FIFO is not empty before we read the data */
  while( (GPLR & GPIO_GPIO14) == 0 );

  return readw(g_io);
}
```

Accessing the device is theoretically possible simply by indexing the pointer, e.g. `myByte = g_io[0];` but this is not good practice. Instead it is better to use one of a variety of functions, of which `readw()` was used in this case. In this driver this function was wrapped by the function named `ov7620_readw()` so that the data from the FIFO is guaranteed to be from the camera.

When accessing a device, if repeated calls are made to `readw()` the so called 'DMA starvation' effect may occur, the result being that LCD image jitters. More on this can be found on the ARM Linux mailing lists.

## 2.7 Accessing the Device via I$^2$C

The kernel I$^2$C system requires the following options to be set when compiling on the SA1100:

- `CONFIG_I2C`
- `CONFIG_I2C_ALGOBIT`
- `CONFIG_I2C_BIT_SA1100_GPIO`
- `CONFIG_I2C_CHARDEV`
- `CONFIG_I2C_PROC`
- `CONFIG_L3`
- `CONFIG_BIT_SA1100_GPIO`

The driver developed in this project is partly based on the stock kernel driver for the USB-wrapped OV7620 camera module used in some webcams.

The kernel I$^2$C support works by populating several structures and calling several functions. After this, and after some initial communication with the device, the driver can access the device at any point using simple send and receive functions.

### 2.7.1 Registering the I$^2$C Functionality

Listing 6: Initializing the basic structure of I$^2$C .

```
struct i2c_driver ov7620_i2c_driver = {
        .name =                 DRIVER_NAME,
        .id =                   I2C_DRIVERID_OV7620,
        .flags =                I2C_DF_NOTIFY,
        .attach_adapter =       ov7620_i2c_attach,
        .detach_client =        ov7620_i2c_detach,
};
```

Once this structure has been declared and populated, the initialization function of the driver calls the `i2c_add_driver` and passes it the address of this structure.

The `i2c_driver` structure's members that were used as follows, more documentation can be found in `$KERNEL_SRC/Documentation/i2c/writing-clients`:

- `name` is a descriptive string, arbitrary.
- `id` is the I$^2$C ID number, in this case the one from the stock kernel driver was used.
- `flags` should be left as shown (apparently).

- attach_adaptor is the address of a function that is called when the driver is added, this does much of the $I^2C$ initialization.
- detach_client is the address of the cleanup function.

### 2.7.2 The $I^2C$ attach Function

This function is called when the $I^2C$ functionality is registered, it does the initial probing and populates an i2c_client structure.

Listing 7: Initializing an i2c_client structure.

```
static struct i2c_client client_template = {
        .id =           -1,
        .driver =       &ov7620_i2c_driver,
};


/*
 * Create a client and find the camera on it, largely
 * stolen from OV511 driver.
 */
static int ov7620_i2c_attach(struct i2c_adapter *adap)
{
  int rc = 0;
  struct i2c_client *c;

  /* create structure to represent ov7620 as client */
  c = (struct i2c_client*)kmalloc(sizeof *c, GFP_KERNEL);

  if (!c)
    return -ENOMEM;

  /* copy this to our globally defined i2c_client client_template */
  memcpy(c, &client_template, sizeof *c);

  c->adapter = adap;
  strcpy(c->name, DRIVER_NAME);

  /* set the I2C address of the camera */
  c->addr = OV7620_SID;

  /* detect the camera using a function specific to this camera */
  rc = ovcamchip_detect(c);

  if (rc < 0) {
    printk(KERN_WARNING "\nCamera␣not␣found.");
    return -EIO;
  }

  /* attach the client */
  i2c_attach_client(c);
  return 0;
}
```

Listing 8: Detecting the camera I$^2$C support.

```
/* this stolen (largely) from OV511 driver by Mark McClelland */
static int ovcamchip_detect(struct i2c_client *c)
{
  int i, success, rc;
  unsigned char high, low, val;

  /* Reset the chip */
  ov_write(c, 0x12, 0x80);

  /* Wait for it to initialize */
  set_current_state(TASK_UNINTERRUPTIBLE);
  schedule_timeout(1 + 150 * HZ / 1000);

  for (i = 0, success = 0; i < I2C_DETECT_RETRIES && !success; i++) {
    if (ov_read(c, GENERIC_REG_ID_HIGH, &high) >= 0) {
      if (ov_read(c, GENERIC_REG_ID_LOW, &low) >= 0) {
        if (high == 0x7F && low == 0xA2) {
          success = 1;
          continue;
        }
      }
    }

    /* Reset the chip */
    ov_write(c, 0x12, 0x80);

    /* Wait for it to initialize */
    set_current_state(TASK_UNINTERRUPTIBLE);
    schedule_timeout(1 + 150 * HZ / 1000);

    /* Dummy read to sync I2C */
    ov_read(c, 0x00, &low);
  }

  if (!success)
    return -EIO;

  /* Detect chip (sub)type */
  rc = ov_read(c, GENERIC_REG_COM_I, &val);

  if( rc < 0 ) {
    printk(KERN_WARNING "\nError,␣couldn’t␣find␣camera␣on␣I2C␣bus.");
    return -EIO;
  }

  if( (val & 3) == 0 ) {
    /* success */
    /*    printk(KERN_WARNING "\nFound OV7620 camera OK.");*/
    return 0;
  } else {
    printk(KERN_WARNING "\nError,␣camera␣responded␣but␣not␣OK.");
    return -EIO;
  }
}
```

### 2.7.3 The $I^2C$ `detach` Function

Listing 9: The function called when the $I^2C$ support is removed.

```
static int ov7620_i2c_detach(struct i2c_client *c)
{
        g_pi2c = NULL;
        i2c_detach_client(c);
        kfree(c);
        return 0;
}
```

### 2.7.4 Communicating with the Camera via $I^2C$

Writing data to the camera is achieved with the `i2c_master_send` function. The first parameter is the `i2c_client*` that was initialized in Listing 7, the second parameter is an array of bytes, the first being the address of the register, the second the value to put in the register and any additional data, the third parameter is the length of the array. For example:

```
i2c_master_send(g_pi2c, &ov7620_settings[n], 2);
```

Reading data back from the camera is achieved with the `i2c_master_recv` function. This has the same parameters as the first function. In the case of this camera, it is necessary to write to a register before reading from it, as it is the write command that sets the address. For example:

```
i2c_master_recv(g_pi2c, &ov7620_settings[n], 1);
```

## 3 The Userland Rendering Program

Listing 10: The rather messy rendering program.

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <linux/fb.h>
#include <sys/mman.h>
#include <math.h>

#define FUDGE601 16
#define limit(x)        (x<0 ? 0 : x>255 ? 255 : x)

struct fb_var_screeninfo vinfo;
struct fb_fix_screeninfo finfo;

/*
 * Taken from VLSI Vision Ltd's driver.
 *
 */
static inline void YUVPixelToRGB24(unsigned char* YUVPixel,
                                   unsigned char* RGBPixel)
{
        int yval, uval, vval, yval2;
        int guv, rv, bu;
        static int value = 0;

        yval = *YUVPixel++ - FUDGE601;
        uval = *YUVPixel++;
        yval2 = *YUVPixel++ - FUDGE601;
        vval = *YUVPixel++;

        uval -= 128;
        vval -= 128;

        yval = 19 * yval;
        yval2 = 19 * yval2;

        bu =  (uval<<5);
        guv =  (- 6*uval - 13*vval);
        rv =  26*vval;

        *RGBPixel++ = (unsigned char)limit(((yval + bu)>>4) );
        *RGBPixel++ = (unsigned char)limit(((yval + guv )>>4));
        *RGBPixel++ = (unsigned char)limit(((yval + rv)>>4)  );

        *RGBPixel++ = (unsigned char)limit(((yval2 + bu)>>4) );
        *RGBPixel++ = (unsigned char)limit(((yval2 + guv )>>4));
        *RGBPixel++ = (unsigned char)limit(((yval2 + rv)>>4)  );

        value++;
        if( value > 255 )
```

```
            value = 0;

        return;
}

static inline u_int32_t pack_pixel(unsigned char* prgb)
{
  u_int32_t tmp;

  tmp = (((prgb[0] >> (8-vinfo.red.length)) << vinfo.red.offset) +
         ((prgb[1] >> (8-vinfo.green.length)) << vinfo.green.offset) +
         ((prgb[2] >> (8-vinfo.blue.length)) << vinfo.blue.offset));

  return tmp;
}


int main()
{
    int fbfd = 0;
    struct fb_cmap cmap;

    unsigned long int screensize = 0;
    char *fbp = 0;
    unsigned int x = 0, y = 0;
    long int location = 0;
    u_int32_t c;
    int nIndex;

    int fd_cam;
    unsigned char* pframe;
    unsigned char yuv_pixels[4];
    unsigned char rgb_pixels[6];
    unsigned char y1;
    unsigned char y2;

    // Open the file for reading and writing
    fbfd = open("/dev/fb0", O_RDWR);
    if (!fbfd) {
        printf("Error:␣cannot␣open␣framebuffer␣device.\n");
        exit(1);
    }
    printf("The␣framebuffer␣device␣was␣opened␣successfully.\n");

    // Get fixed screen information
    if (ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo)) {
        printf("Error␣reading␣fixed␣information.\n");
        exit(2);
    }

    // Get variable screen information
    if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)) {
```

```
    printf("Error␣reading␣variable␣information.\n");
    exit(3);
}

printf("%dx%d,␣%dbpp\n", vinfo.xres, vinfo.yres, vinfo.bits_per_pixel );

// Figure out the size of the screen in bytes
screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;

// Map the device to memory
fbp = (char *)mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED,
                   fbfd, 0);

if ((int)fbp == -1) {
    printf("Error:␣failed␣to␣map␣framebuffer␣device␣to␣memory.\n");
    exit(4);
}
printf("The␣framebuffer␣device␣was␣mapped␣to␣memory␣successfully.\n");

x = 0; y = 0;       // Where we are going to put the pixel

fd_cam = open("/dev/omni", O_RDWR);

if(0 == fd_cam) {
  printf("\nError␣opening␣camera.");
  exit(4);
}

pframe = (unsigned char*)malloc(sizeof(unsigned char)*640*480*3);

if( NULL == pframe ) {
  printf("\nCouldn't␣allocate␣frame␣memory.");
  exit(4);
}

while(1) {

  while( read(fd_cam, pframe, 640*480*3) <= 0 )
    usleep(100);


  nIndex = 0;
  for ( y = 0; y < 480; y++ )
    for ( x = 0; x < 640; ) {


      YUVPixelToRGB24(&pframe[nIndex], rgb_pixels);
      nIndex+=4;

      /* plot pixel 1 */
      c = pack_pixel(&rgb_pixels[0]);
```

```
            /* calculate position */
            location = (x+vinfo.xoffset) * 2 +
              (y+vinfo.yoffset) * finfo.line_length;
            x++;

            /* output at that position */
            *((u_int16_t*)(fbp + location)) = (u_int16_t)c;

            /* plot pixel 2 */
            c = pack_pixel(&rgb_pixels[3]);

            /* calculate position */
            location = (x+vinfo.xoffset) * 2 +
              (y+vinfo.yoffset) * finfo.line_length;
            x++;

            /* output at the that position */
            *((u_int16_t*)(fbp + location)) = (u_int16_t)c;
          }

      }
      munmap(fbp, screensize);
      close(fd_cam);
      close(fbfd);
      return 0;
}
```

## References

[1] Alessandro Rubini and Jonathon Corbet, Linux Device Drivers, 2001, O'Reilly Press.