Independent Study

**Time-Lapse video generation using BeagleBoard xM and LI-3M01 HD Camera Board.**

Aim:

This independent study involved interfacing of the LI-3M01 HD Camera Board to the beagle board and getting images from it. Then using the images to form a video.

This paper describes the various steps needed to be followed in order to grab the images from the camera board at real time. This paper also consists of sections introducing the reader to device driver in general.
**Independent Study**

**Porting Angstrom on to BeagleBoardxM and Cross Compiler:**

Porting Angstrom Linux on to the beagle board and the arm cross-compiler are explained in Nisheeth's IS report. This report can be found in: [http://ece.colorado.edu/~ecen5623/ecen/labs/Linux/IS/](http://ece.colorado.edu/~ecen5623/ecen/labs/Linux/IS/)

Please use the files provided in this package while porting Angstrom. I have made some additional changes to the files which now include support for Video for Linux V4l.

**LI-3M01 HD Camera Board:**

This is a 3 mega-pixel camera board which can be interfaced to the Beagleboard xM through its camera interface. A document describing the features and applications of this board is included in this package named "LI-3M01_HD_Camera_BOARD.pdf".

After plugging this camera board into the camera interface of the Beagleboard. We should see the new device on the /dev directory. Depending on the number of camera device connected our board will be seen as video0 or video1.

**Video4Linux Programming:**

Video4Linux is intended to provide a common programming interface for the many TV and capture cards now on the market, as well as parallel port and USB video cameras. Radio, teletext decoders and vertical blanking data interfaces are also provided. Video4Linux package will provide us with APIs to open and scan these video devices to either query their capability or read from the device.

To better understand how V4l works we need to first understand how Linux Kernel is designed and how driver is written and how it can be accessed.

**Device Drivers- Basics:**

A device driver is the lowest level of software as it is directly bound to the hardware features of a device. The kernel can be considered as an application running on top of device drivers. Each driver manages one or more pieces of hardware while the kernel handles process scheduling, filesystem access, interrupts, etc.
Independent Study

Drivers can be integrated directly into the kernel, or can be designed as loadable modules. The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module.

- **Character Device:-**

  A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; this usually applies to frame grabbers, where the applications can access the whole acquired image using mmap or lseek.

- **Block devices:-**

  Like char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is something that can host a filesystem, such as a disk. In most Unix systems, a block device can be accessed only as multiples of a block, where a block is usually one kilobyte of data or another power of 2. Linux allows the application to read and write a block device like a char device — it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node and the difference between them is transparent to the user. A block driver offers the kernel the same interface as a char driver, as well as an additional block-oriented interface that is invisible to the user or applications opening the /dev entry points. That block interface, though, is essential to be able to mount a filesystem.
Independent Study

- Network interfaces:

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Though both Telnet and FTP connections are stream oriented, they transmit using the same device; the device doesn’t see the individual streams, but only the data packets. Not being a stream-oriented device, a network interface isn’t easily mapped to a node in the filesystem, as /dev/tty1 is. The Unix way to provide access to interfaces is still by assigning a unique name to them (such as eth0), but that name doesn’t have a corresponding entry in the filesystem. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission.

There are other types of devices which do not fit precisely into these 3 types. For example SCSI devices share an underlying protocol regardless of function. Writing a driver for this device involves a driver for the controller hardware which may run many devices. USB devices also share an underlying protocol. There is a lower layer of drivers tied to the controller hardware, and then device specific drivers for the various peripherals connected to the bus.

User applications and daemons interact with peripheral devices using the same basic system calls irrespective of the specific nature of the device. Network devices operate in a different way and is not discussed in this paper.

For each one of the limited number of these system calls, there is a corresponding entry point in the driver. The main ones for character drivers are:

open(), release(), read(), write(), lseek(), ioctl and mmap().
Independent Study

The Kernel:

When talk device drivers, it’s important to make the distinction between “user space” and “kernel space”.

- **Kernel space.** Linux (which is a kernel) manages the machine's hardware in a simple and efficient manner, offering the user a simple and uniform programming interface. In the same way, the kernel, and in particular its device drivers, form a bridge or interface between the end-user/programmer and the hardware. Any subroutines or functions forming part of the kernel (modules and device drivers, for example) are considered to be part of kernel space.

- **User space.** End-user programs, like the UNIX shell or other GUI based applications, are part of the user space. Obviously, these applications need to interact with the system’s hardware. However, they don’t do so directly, but through the kernel supported functions.

The kernel offers several subroutines or functions in user space, which allow the end-user application programmer to interact with the hardware. Usually, in UNIX or Linux systems, this dialogue is performed through functions or subroutines in order to read and write files. The reason for this is that in Unix devices are seen, from the point of view of the user, as files.

On the other hand, in kernel space Linux also offers several functions or subroutines to perform the low level interactions directly with the hardware, and allow the transfer of information from kernel to user space.

Usually, for each function in user space (allowing the use of devices or files), there exists an equivalent in kernel space (allowing the transfer of information from the kernel to the user and vice-versa).

All driver code is written in the form of modules and inserted into the kernel code. These drivers react to either the system call generated by a user or to an input from the device. open(), release(), read(), write(), lseek(), ioctl and mmap() are all system calls that the user space generates which asks the kernel to do a specific operation on the device.

open(), release(), read(), write() are the simple ones which as the name suggests are used to open a device, close a device, read from a device and write to a device. Most of the code explained later include ioctl() calls to interact with the camera device.
Independent Study

ioctl is short for input/output control, is a system call for device-specific operations and other operations which cannot be expressed by regular system calls. It takes a parameter specifying a request code; the effect of a call depends completely on the request code. Request codes are often device-specific. For instance, a CD-ROM driver which can get a physical device to eject a disc would provide an ioctl request code to do that. Device-independent request codes are sometimes used to give userland access to kernel functions which are only used by core system software or still under development.

mmap is a POSIX-compliant Unix system call that maps files or devices into memory. It is a method of memory-mapped file I/O. It naturally implements demand paging, because initially file contents are not entirely read from disk and do not use physical RAM at all.

I think I have explained all that is need to start using the V4l package and access our camera device.

Frame Capture using V4L

V4L APIs provide us with the functionality to control the capture device. This statement will become more clear after we take a look at the 8 step process for capturing frames.

Step1: Open Device

We need to open the device. (Remember that everything in Linux is treated similar to a file.)

```c
int fd;
fd = open("/dev/video0", O_RDWR);
```

We are opening a device and assigning a file descriptor to it. Please note that "video0" is just an example. Your device could be assigned a new number.
Independent Study

Step 2: Set Format

struct v4l2_format defines various elements to store the format of frames. The capture device will capture the frames and save them as the format specifies. Please note that you cannot exceed the hardware limitations of the device.

```c
struct v4l2_format format;
format.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
format.fmt.pix.width = 1920;
format.fmt.pix.height = 1080;
format.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
ioctl(fd, VIDIOC_S_FMT, &format);
```

We are setting the type to video capture. And setting the resolution to 1920*1080. A lower resolution can also be set. The pixelformat is now set so that the camera captures the raw frames. This can be set to "V4L2_FMT_FLAG_COMPRESSED" if the frames are to be in some compressed format.

When the application wants to change the hardware's format for real, it does a "VIDIOC_S_FMT" call through ioctl.

Step 3: Allocate Memory

This ioctl is used to initiate memory mapped or user pointer I/O. Memory mapped buffers are located in device memory and must be allocated with this ioctl before they can be mapped into the application's address space. User buffers are allocated by applications themselves, and this ioctl is merely used to switch the driver into user pointer I/O mode.

To allocate device buffers applications initialize three fields of a v4l2_requestbuffers structure. They set the type field to the respective stream or buffer type, the count field to the desired number of buffers, and memory must be set to V4L2_MEMORY_MMAP. When the ioctl is called with a pointer to this structure the driver attempts to allocate the requested number of buffers and stores the actual number allocated in the count field. It can be smaller than the number requested, even zero, when the driver runs out of free memory. A larger number is possible
Independent Study

when the driver requires more buffers to function correctly.[1] When memory mapping I/O is not supported the ioctl returns an EINVAL error code.

```c
struct v4l2_requestbuffers req;
void **mem;
req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
req.count = 4;
req.memory = V4L2_MEMORY_MMAP;
ioctl(fd, VIDIOC_REQBUFS, &req);
printf(“Driver allocated %u buffers
”, req.count);
mem = malloc(req.count, sizeof *mem);
```

Step 4: Map Memory

mmap() is used to map each of the address locations in step 3 to the device to store the frames.

```c
struct v4l2_buffer buffer;
unsigned int i;
for (i = 0; i < req.count; ++i) {
    buffer.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buffer.memory = V4L2_MEMORY_MMAP;
    buffer.index = i;
    ioctl(fd, VIDIOC_QUERYBUF, &buffer);
    mem[i] = mmap(0, buffer.length,
                  PROT_READ|PROT_WRITE,
                  MAP_SHARED, fd, buffer.m.offset);
}
```

Step 5: Start video stream and queue

Applications call the VIDIOC_QBUF ioctl to enqueue an empty (capturing) or filled (output) buffer in the driver's incoming queue. The semantics depend on the selected I/O method.

To enqueue a memory mapped buffer applications set the type field of a struct v4l2_buffer to the same buffer type as previously struct v4l2_format type and struct
Independent Study

v4l2_requestbuffers type, the memory field to V4L2_MEMORY_MMAP and the index field. Valid index numbers range from zero to the number of buffers allocated with VIDIOC_REQBUFS (struct v4l2_requestbuffers count) minus one. The contents of the struct v4l2_buffer returned by a VIDIOC_QUERYBUF ioctl will do as well.

To enqueue a user pointer buffer applications set the type field of a struct v4l2_buffer to the same buffer type as previously struct v4l2_format type and struct v4l2_requestbuffers type, the memory field to V4L2_MEMORY_USERPTR and the m.userptr field to the address of the buffer and length to its size. When the buffer is intended for output additional fields must be set as above. This ioctl locks the memory pages of the buffer in physical memory, they cannot be swapped out to disk. Buffers remain locked until dequeued, until the VIDIOC_STREAMOFF or VIDIOC_REQBUFS ioctl are called, or until the device is closed.

Applications call the VIDIOC_DQBUF ioctl to dequeue a filled (capturing) or displayed (output) buffer from the driver's outgoing queue. They just set the type and memory fields of a struct v4l2_buffer as above, when VIDIOC_DQBUF is called with a pointer to this structure the driver fills the remaining fields or returns an error code.

By default VIDIOC_DQBUF blocks when no buffer is in the outgoing queue. When the O_NONBLOCK flag was given to the open() function, VIDIOC_DQBUF returns immediately with an EAGAIN error code when no buffer is available.

```c
struct v4l2_buffer buffer;
int type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
unsigned int i;
for (i = 0; i < req.count; ++i) {
    buffer.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buffer.memory = V4L2_MEMORY_MMAP;
    buffer.index = i;
    ioctl(fd, VIDIOC_QBUF, &buffer);
}
ioctl(fd, VIDIOC_STREAMON, &type);
```
Independent Study

```c
While (1) {
    buffer.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buffer.memory = V4L2_MEMORY_MMAP;
    ioctl(fd, VIDIOC_DQBUF, &buffer);
    process_buffer(buffer, mem[buffer.index]);
    ioctl(fd, VIDIOC_QBUF, &buffer);
}
```

The VIDIOC_STREAMON and VIDIOC_STREAMOFF ioctl start and stop the capture or output process during streaming (memory mapping or user pointer) I/O.

Specifically the capture hardware is disabled and no input buffers are filled (if there are any empty buffers in the incoming queue) until VIDIOC_STREAMON has been called. Accordingly the output hardware is disabled, no video signal is produced until VIDIOC_STREAMON has been called. The ioctl will succeed only when at least one output buffer is in the incoming queue.

The VIDIOC_STREAMOFF ioctl, apart of aborting or finishing any DMA in progress, unlocks any user pointer buffers locked in physical memory, and it removes all buffers from the incoming and outgoing queues. That means all images captured but not dequeued yet will be lost, likewise all images enqueued for output but not transmitted yet. I/O returns to the same state as after calling VIDIOC_REQBUFS and can be restarted accordingly.

Both ioctl take a pointer to an integer, the desired buffer or stream type. This is the same as struct v4l2_requestbuffers type.

As and when we get new data we can save it in the flash drive of the beagleboard from the ram.

**Converting YUVs to RGB**

YUV is a color space typically used as part of a color image pipeline. It encodes a color image or video taking human perception into account, allowing reduced bandwidth for chrominance components, thereby typically enabling transmission errors or compression artifacts to be more efficiently masked by the human perception than using a "direct" RGB-representation. Other color spaces have similar properties, and the main reason to implement or investigate properties of Y'UV
would be for interfacing with analog or digital television or photographic equipment that conforms to certain Y'UV standards.

Most of the format supported by image viewing software such as gimp or bitman do not support YUV files.

We need to convert the YUVs to RGB before we can view them normally.

Here is the standard formula to be used for this conversion:

\[
\begin{align*}
B &= 1.164(Y - 16) + 2.018(U - 128) \\
G &= 1.164(Y - 16) - 0.813(V - 128) - 0.391(U - 128) \\
R &= 1.164(Y - 16) + 1.596(V - 128)
\end{align*}
\]

**Time-Lapse Video:**

A time lapse video can be generated by using all the frames captured at 1fps by using the ffmpeg command:

```
ffmpeg -r 1 -b 1800 -i output/pic-%d.ppm Timelapse.mp4
```

"-r 1" specifies that the video needs to be made at 1 frame a second.

**Result:**

A time-lapse video generated at 1fps is also included in this package. This video was made of frames of 640x480. The maximum resolution of this camera is 1920x1080 and the ioctl call can be used to decrease the resolution in the hardware.
Conclusion and Future Scope:

The LI-3M01 HD is a very versatile device which can be used for developing many image processing applications. The ability to control resolution and compression provides a very good working environment for developing a variety of applications.

This project can be extended to include real time streaming. Where as you can send frames at real time via TCP socket to a host machine. In order to make this work the bug in the ethernet port of Beagleboard rev C needs to be fixed.