DaVinci / OMAP Software Design Workshop

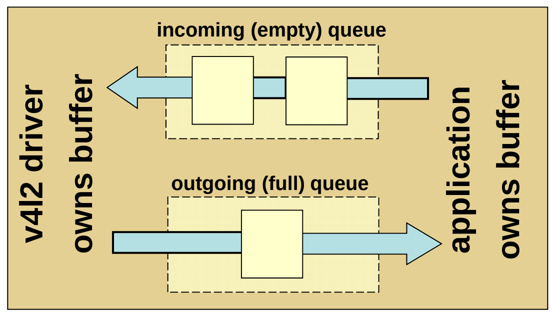
Outline

- **v4L2 Capture Driver**
  - Using mmap
  - v4L2 Coding
- **FBdev Display Driver**
  - Video Planes
  - FBdev Coding
- **Video Display Boot Args**
- **Lab Exercise**

v4l2 Driver Overview

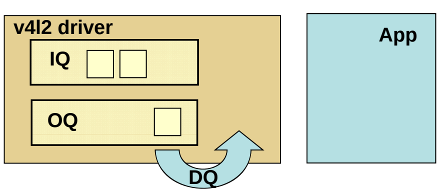
- **v4l2 is a standard Linux video driver used in many linux systems**
- **Supports video input and output**
  - **In this workshop we use it for video input only**
- **Device node**
  - **Node name: /dev/video0**
  - **Uses major number 81**
- **v4l2 spec:** http://linuxtv.org/wiki/index.php/Main_Page

v4l2 Driver Queue Structure



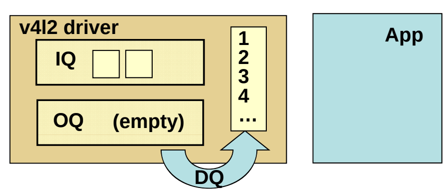
- Application takes ownership of a full video buffer from the outgoing driver queue using the VIDIOC_DQBUF ioctl
- After using the buffer, application returns ownership of the buffer to the driver by using VIDIOC_QBUF ioctl to place it on the incoming queue

v4l2 Enqueue and Dequeue



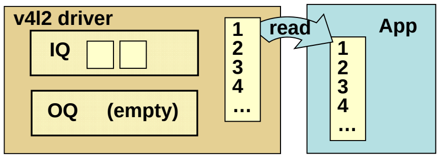
- **Buffers typically exist in driver's memory space**
- **The dequeue call makes the data available to the app**

v4l2 Enqueue and Dequeue



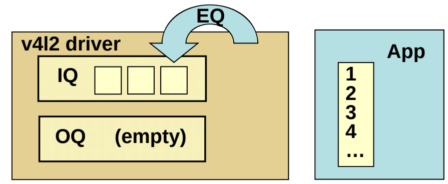
- **Buffers typically exist in driver's memory space**
- **The dequeue call makes the buffer available to the app**
- **Even after DQ, buffers still exist in the driver's memory space but not the application's**

v4l2 Enqueue and Dequeue



- Buffers typically exist in driver's memory space
- The dequeue call makes the buffer available to the app
- Even after DQ, buffers still exist in the driver's memory space but not the application's
- read and write operations copy buffers from driver's memory space to app's or vice-versa

v4l2 Enqueue and Dequeue



- Buffers typically exist in driver's memory space
- These buffers exist in the driver's memory space but not the application's
- read and write operations copy buffers from driver's memory space to app's or vice-versa

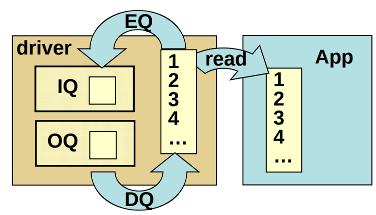
Is there a better method than "read"?

Outline

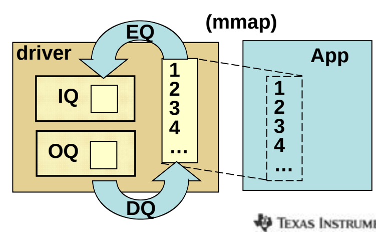
- **v4L2 Capture Driver**
  - Using mmap
  - v4L2 Coding
- **FBdev Display Driver**
  - Video Planes
  - FBdev Coding
- **Video Display Boot Args**
- **Lab Exercise**

mmap – A Better Way

- Standard read and write copy data from driver buffer to a new buffer in application process's memory space

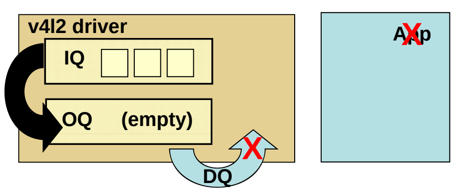


- Use mmap to expand the application process's memory space to include the driver buffer
- Returns a pointer to the location in the app's memory space



v4l2 Queue Synchronization

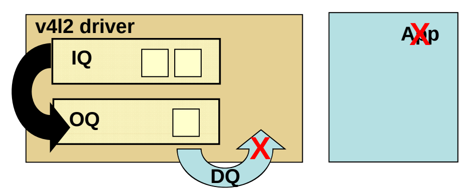
Driver fills IQ buf and places on OQ



- The VIDIOC_DQBUF ioctl blocks the thread's execution waits if until a buffer is available on the output queue
- When driver adds a new, full buffer to the output queue, the application process is released
- Dequeue call completes and application resumes with the following set of commands

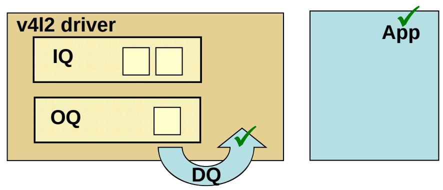
v4l2 Queue Synchronization

Driver fills IQ buf and places on OQ



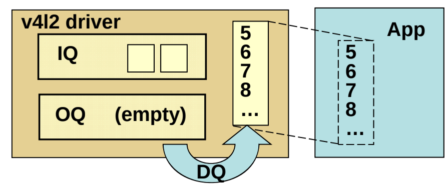
- The VIDIOC_DQBUF ioctl blocks the thread's execution waits if until a buffer is available on the output queue
- When driver adds a new, full buffer to the output queue, the application process is released
- Dequeue call completes and application resumes with the following set of commands

v4l2 Queue Synchronization



- The VIDIOC_DQBUF ioctl blocks the thread's execution waits if until a buffer is available on the output queue
- When driver adds a new, full buffer to the output queue, the application process is released
- Dequeue call completes and application resumes with the following set of commands

v4l2 Synchronization



- The VIDIOC_DQBUF ioctl blocks the thread's execution waits if until a buffer is available on the output queue
- When driver adds a new, full buffer to the output queue, the application process is released
- Dequeue call completes and application resumes with the following set of commands

Outline

- **v4L2 Capture Driver**
  - Using mmap
  - v4L2 Coding
- **FBdev Display Driver**
  - Video Planes
  - FBdev Coding
- **Video Display Boot Args**
- **Lab Exercise**

v4l2 Buffer Passing Procedure

```
while(cond){
   ioctl(v4l2_input_fd, VIDIOC_DQBUF, &buf);
   bufPtr = mmap(NULL,                          // "start" address (ususally 0)
                 buf.length,                    // length
                 PROT_READ | PROT_WRITE,        // allowed use for map'd memory
                 MAP_SHARED,                    // allow sharing of mapped bufs
                 v4l2_input_fd,                 // driver/file descriptor
                 buf.m.offset);                 // offset requested from "start"
   doSomething(bufPtr, buf.length, ...);
   munmap(bufPtr, buf.length);
   ioctl(v4l2_input_fd, VIDIOC_QBUF, &buf);
}
```

- A simple flow would be: (1) DQBUF the buffer, (2) map it into user space, (3) use the buffer, (4) unmap it, (5) put it back on the driver's queue
- More efficient code would map each driver buffer once during initialization, instead of mapping and unmapping within the loop
- Alternatively, later versions of the driver allow you to create the buffer in 'user' space and pass it to the driver

Commonly Used v4l2 ioctl's

Data Structures

```
struct v4l2_requestbuffers req;
req.count;      // how many buffers to request
req.type;       // capture, output, overlay
req.memory;     // mmap, userptr, overlay

struct v4l2_buffer buf;
buf.index;      // which driver buffer
buf.type;       // matches req.type
buf.memory;     // matches req.memory
buf.m.offset;   // location of buffer in driver mem
```

Request the driver allocate a new buffer

```
ioctl(fd, VIDIOC_REQBUFS, &req);
```

Get information on a driver buffer

```
ioctl(fd, VIDIOC_QUERYBUF, &buf);
```

Enqueue and Dequeue buffers to/from driver

```
ioctl(fd, VIDIOC_QBUF, &buf);
Ioctl(fd, VIDIOC_DQBUF, &buf);
```

Outline

- **v4L2 Capture Driver**
  - Using mmap
  - v4L2 Coding
- **FBdev Display Driver**
  - Video Planes
  - FBdev Coding
- **Video Display Boot Args**
- **Lab Exercise**

FBdev Driver Overview

- **FBdev is a standard Linux video output driver used in many linux systems**
- **Can be used to map the frame buffer of a display device into user space**
- **Device nodes have major number 29**
- **Device nodes have a minor number x**

```
# ls -ls /dev/fb*
0 lrwxrwxrwx 1 root root      3 Jan  1  1970 /dev/fb -> fb0
0 crw-rw---- 1 root video 29, 0 Jan  1  1970 /dev/fb0
0 crw-rw---- 1 root video 29, 1 Jan  1  1970 /dev/fb1
0 crw-rw---- 1 root video 29, 2 Jan  1  1970 /dev/fb2
```

FBdev Driver Overview

- **Uses `/dev/fbx` node naming convention**
- **`/dev/fb0` -> GFX**
- **`/dev/fb1` -> Video1**
- **`/dev/fb2` -> Video2**

Outline

- **v4L2 Capture Driver**
  - Using mmap
  - v4L2 Coding
- **FBdev Display Driver**
  - Video Planes
  - FBdev Coding
- **Video Display Boot Args**
- **Lab Exercise**

OMAP35xx Display Subsystem & Connectivity

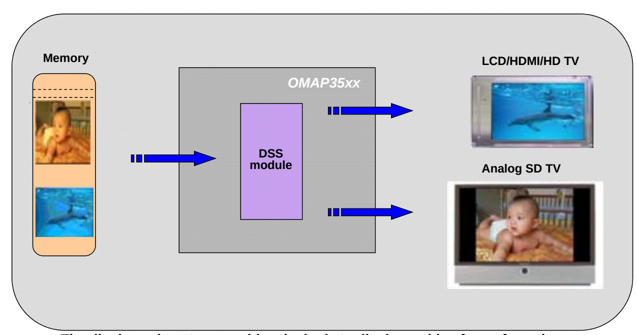
Original slides by Gustavo Martinez
Adapted & Presented by Steve Clynes

Adapted & Presented by Mark A. Yoder

Agenda

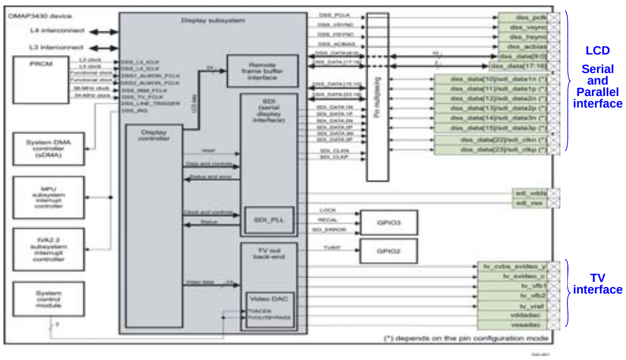
- High-Level Overview
- **Display Subsystem Features**
- **Display Controller**
  - **Direct Memory Access (DMA)**
  - **Graphics Pipeline**
  - **Video Pipeline**
  - **Overlay Manager**
  - **Output Paths**
- Remote Frame Buffer Interface (RFBI)
  - Bypass Mode (RFBI disabled)
  - RFBI Mode (RFBI enabled)
- Video Encoder (VENC)

Display Subsystem (DSS)

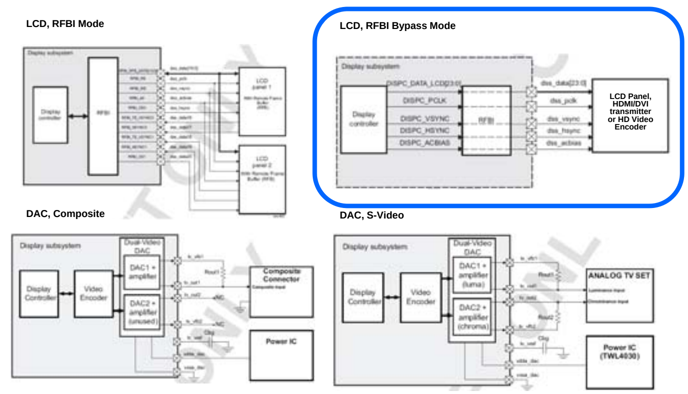


The display subsystem provides the logic to display a video frame from the memory frame buffer (SDRAM) on a liquid-crystal display (LCD) or/and TV.

DSS Configuration



DSS Configuration



Agenda

- **High-Level Overview**
- **Display Subsystem Features**
- **Display Controller**
  - **Direct Memory Access (DMA)**
  - **Graphics Pipeline**
  - **Video Pipeline**
  - **Overlay Manager**
  - **Output Paths**
- **Remote Frame Buffer Interface (RFBI)**
  - **Bypass Mode (RFBI disabled)**
  - **RFBI Mode (RFBI enabled)**
- **Video Encoder (VENC)**

DSS Features

- Display Controller
  - Display modes :
    - Programmable pixel display modes (1, 2, 4, 8, 12, 16, 18 and 24 bit-per-pixel modes)
    - 256 x 24-bit entries palette in RGB
    - **Programmable display size (max resolution is 2048 lines by 2048 pixels)**
    - **Programmable pixel rate up to 74.25MHz***
    - **Rule of thumb: max frame rate = (74250000 / (X * Y * 1.3))**
  - Display support :
    - Passive & Active Matrix panel
    - Remote Frame Buffer support through the RFBI module
    - 12/16/18/24-bit active matrix panel interface

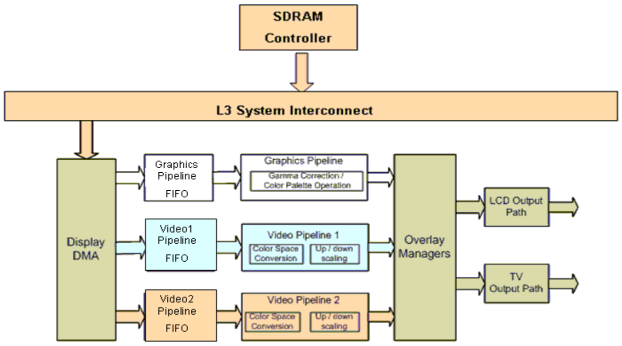
DSS Features

- Display Controller (cont):
  - Signal processing :
    - Overlay support for Graphics, Video1, and Video2
    - Video resizer : up-sampling (up to x8) down-sampling (down to 1/4)
    - Rotation 90°, 180°, and 270° — Doesn't work
    - Transparency color key (source and destination)
    - Programmable video color space conversion YUV 4:2:2 into RGB
    - Gamma curve support
    - Mirroring support — Doesn't work
    - Programmable Color Phase Rotation (CPR)

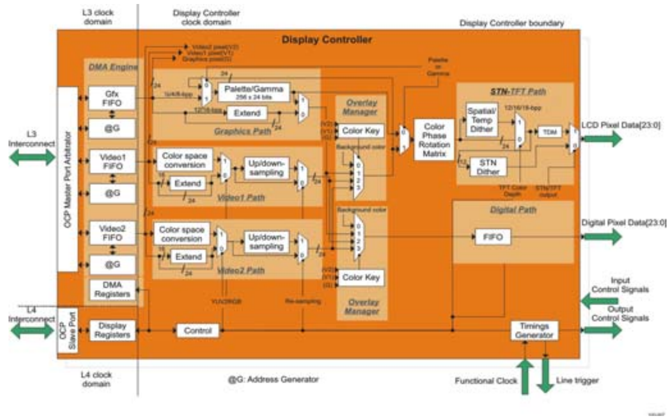
Agenda

- **High-Level Overview**
- **Display Subsystem Features**
- **Display Controller**
  - **Direct Memory Access (DMA)**
  - **Graphics Pipeline**
  - **Video Pipeline**
  - **Overlay Manager**
  - **Output Paths**
- **Remote Frame Buffer Interface (RFBI)**
  - **Bypass Mode (RFBI disabled)**
  - **RFBI Mode (RFBI enabled)**
- **Video Encoder (VENC)**

DISPC High-Level Block Diagram



DISPC Detailed Block Diagram



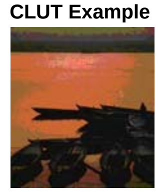
DISPC Graphics Pipeline

- Color Palette LUT (also used for Gamma correction)
  - Input data in 1, 2, 4, or 8bpp can be trans-coded using look-up tables (LUT) to 24bpp, RGB.
  - Input data is used to index a table (color palette) of 24-bit RGB values.
    - 1bpp indexes 2-entry color palette
    - 4bpp indexes 16-entry color palette
    - 8bpp indexes 256-entry color palette
  - Reduces memory space requirements for simple graphics like icons / user interfaces containing only a limited number of colors.
  - Color indexing system can be used to assign key colors to input data.
  - The table can be reloaded every frame, once or never.

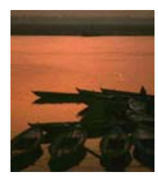
DISPC Graphics Pipeline

- **CLUT Example**

Original image : 24-bit per pixel (16777216 colors)



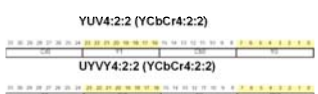
- 8 bits per pixel
- Few colors are available to express the subtle shades of light on the water.

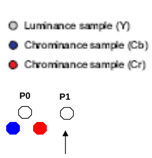


- 8 bits per pixel
- The LUT assigns most of the 256 possible colors to the colors of the water and the realism is restored.

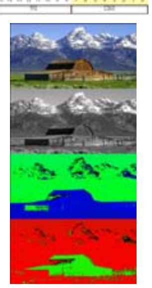
DISPC Video Pipeline

- Video pipe input can be YUV4:2:2, UYVY4:2:2 or RGB (16, 24 bpp)
  - The Y in YUV stands for "luma," which is brightness, or lightness, and black and white.
  - U and V provide color information and are "color difference" signals of blue minus luma (B-Y) and red minus luma (R-Y).
  - Cb and Cr are sampled at a lower rate than Y, which is technically known as "chroma subsampling."
  - Some color information in the video signal is being discarded, but not brightness (luma) information.



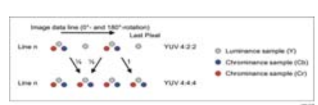


P1 needs Cb and Cr components of P0

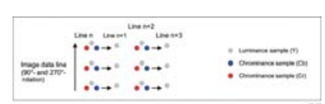


DISPC Video Pipeline

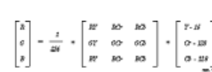
- Color Space Conversion:
  - Converts the video encoded pixel values from YCbCr4:2:2 format into RGB24 (working internal format) in two steps.
  - First step: Converts YCbCr4:2:2 to YCbCr4:4:4
    - Interpolation of the chrominance values (Cb/Cr) using the neighboring pixels.



The chrominance samples are averaged using the two adjacent values (0 and 180 rotation)



The chrominance samples are duplicated from the adjacent values (90 and 270 rotation)



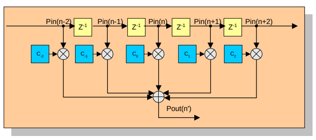
  - Second Step: Converts YCbCr4:4:4 to RGB24
    - 3x3 Programmable Conversion Matrix (11-bit coefficients).

DISPC Video Pipeline

- Video Up/Down Sampling
  - Each video pipeline has a dedicated image scaler
    - Capable of up or down-scaling the video image in both dimensions
      - Increases or decreases the image size
      - Zoom factor between 25…800% in each dimension
    - Works with RGB24 input pixels
  - Up and down-scaling both use the same filters (poly-phase filter)
    - Different coefficients and increment steps
    - 8-bit coefficients
  - User programs the filter coefficients (once) and the scaling factor (CPU TRM)

$$VIDFIRHINC[12{:}0] = 1024 \times \left(\frac{VIDORGSIZEX[10{:}0]}{VIDSIZEX[10{:}0]}\right)$$

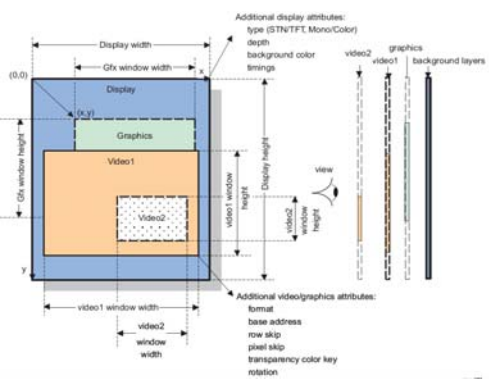
$$VIDFIRVINC[12{:}0] = 1024 \times \left(\frac{VIDORGSIZEY[10{:}0]}{VIDSIZEY[10{:}0]}\right)$$



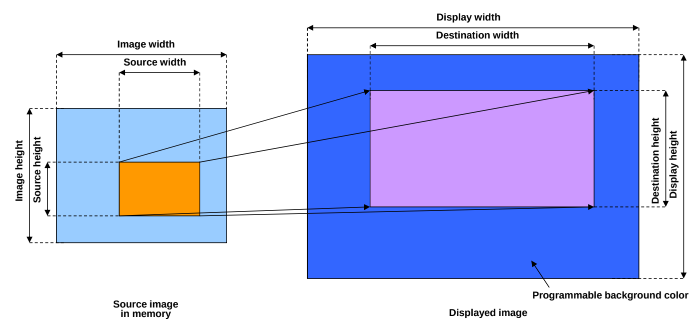
DISPC Overlay Manager

- Combines different input components into one output image
  - Each input component can be any size up to full display screen size
  - Each input component can start at any location
  - Up to three input components exist:
    - Graphics
    - Video 1
    - Video 2
  - Additionally a solid background color can be selected
- Two overlay managers exist:
  - Overlay manager 1 for LCD output
  - Overlay manager 2 for TV output
  - Cannot share same input source
- Orders the different input components
  - Video2 >> Video1 >> Graphics >> Background in normal mode
  - Graphics >> Video2 >> Video1 >> Background in blend mode

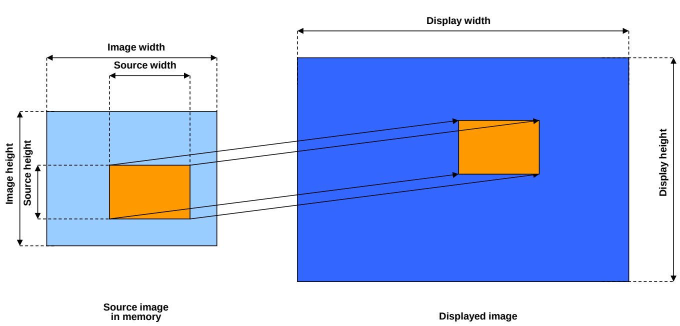
DISPC Overlay Manager



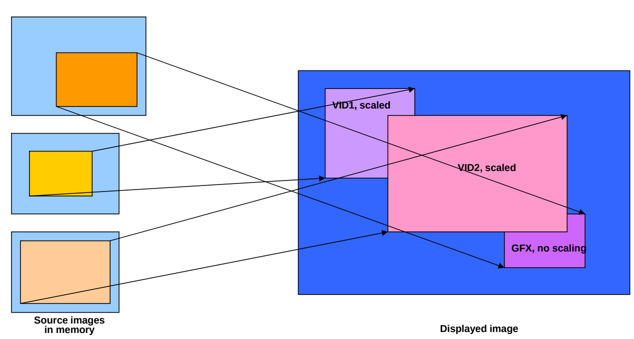
DISPC VIDn Path Image Mapping



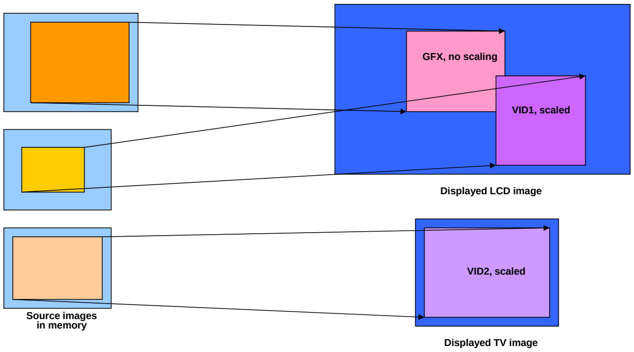
DISPC GFX Image Mapping



DISPC Image Mapping Examples

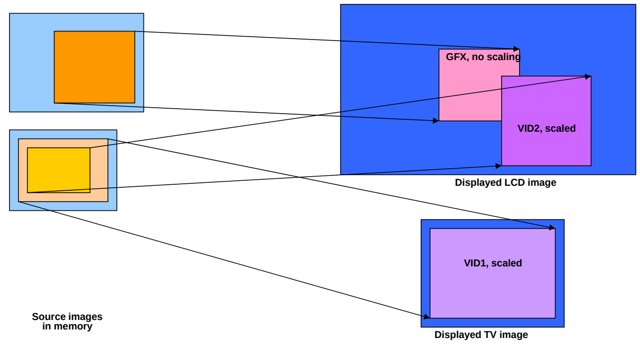


DISPC Image Mapping Examples



Displayed LCD image

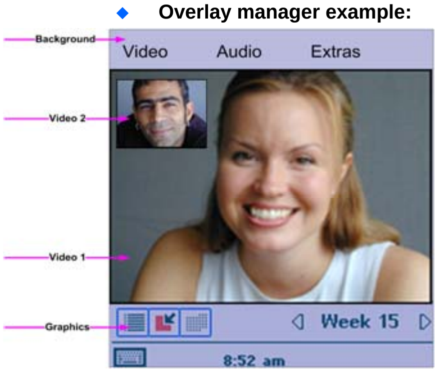
DISPC Image Mapping Examples



Displayed LCD image

DISPC Overlay Manager

- **Overlay manager example:**

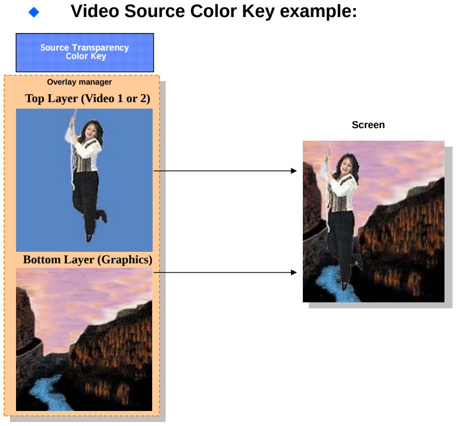


DISPC Overlay Manager

- Transparency Color Keys
  - Two different transparency color keys to define what data will or will not be displayed.
  - Source transparency color key, Video
    - Defines a video pixel color considered as transparent.
      - Pixels of this color will NOT be displayed even though they are in front.
    - Makes lower layers visible
    - Cannot be used with up or down sampling
  - Destination transparency color key, Graphics
    - Defines the encoded pixels in the video layers to be displayed
      - Used to bring graphics into the foreground by hiding the video pixels
      - All graphics that do not have the key color will be in the foreground
    - Works only in the graphics region when graphics and video overlap
  - Only one key mechanism can be used at the same time

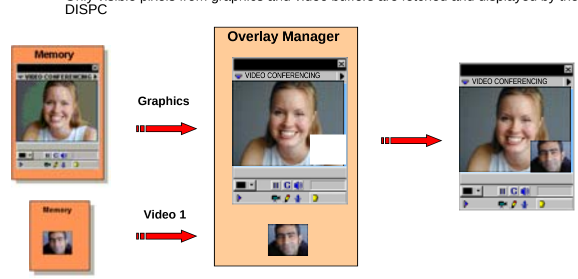
DISPC Overlay Manager

- **Video Source Color Key example:**



DISPC Overlay Manager

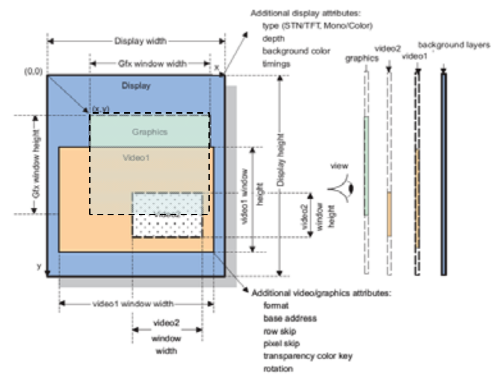
- Overlay Optimization
  - This feature can be used to fetch from the memory only the pixels needed.
    - At least video window 1 and graphics window must be enabled
    - The graphics pixels under the video 1 will not be fetched from the memory
    - The transparency color key must be disabled
  - Reduces the peak bandwidth
    - Only visible pixels from graphics and video buffers are fetched and displayed by the DISPC



Agenda

- High-Level Overview
- Display Subsystem Features
- Display Controller
  - Direct Memory Access (DMA)
  - Graphics Pipeline
  - Video Pipeline
  - Overlay Manager
  - Output Paths
- Remote Frame Buffer Interface (RFBI)
  - Bypass Mode (RFBI disabled)
  - RFBI Mode (RFBI enabled)
- Video Encoder (VENC)

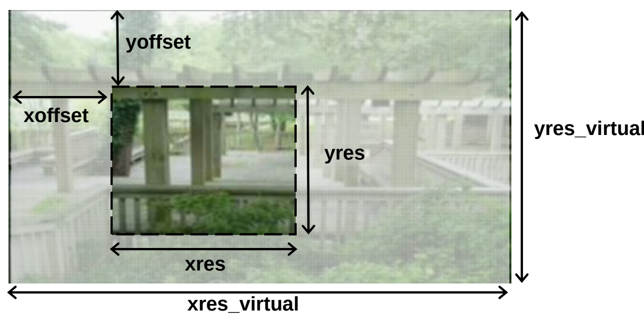
DISPC Overlay Manager



Outline

- **v4L2 Capture Driver**
  - Using mmap
  - v4L2 Coding
- **FBdev Display Driver**
  - Video Planes
  - FBdev Coding
- **Video Display Boot Args**
- **Lab Exercise**

For Each /dev/fb*x* Video Plane



- **Named FBdev because it gives direct access to the display device's frame buffer**
- **FBIOPAN_DISPLAY allows users to pan the active display region within a virtual display buffer**

DSS2 Details

- A **framebuffer** can be connected to multiple overlays to show the same pixel data on all of the overlays
- Any framebuffer can be connected to any overlay
- An **overlay** can be connected to one overlay manager
- An **overlay manager** can be connected to *one* display
- See `…/Documentation/arm/OMAP/DSS`

Default setup on OMAP3

- **Here's the default setup on OMAP3 SDP board. All planes go to LCD. DVI and TV-out are not in use. The columns from left to right are:**
  - **framebuffers,**
  - **overlays,**
  - **overlay managers,**
  - **displays.**

```
fb0 --- gfx-\            DVI
fb1 --- vid1 --+- lcd ---- lcd
fb2 --- vid2 -/   tv ----- tv
```

- **Framebuffers are handled by omapfb, and the rest by the DSS.**

Hands on #4: DSS2 Details

- DSS2 uses **sysfs**
- The framebuffers are in **/sys/class/graphics/**

```
# ls -F /sys/class/graphics/
fb0@  fb1@  fb2@  fbcon@
# cd /sys/class/graphics/fb0
# ls -F
bits_per_pixel  dev      modes           pan        rotate_type  subsystem@
blank           device@  name            phys_addr  size         uevent
console         mirror   overlays        power/     state        virt_addr
cursor          mode     overlays_rotate rotate     stride       virtual_size
```

DSS2 Details

- The overlays, managers and displays are in **/sys/devices/platform/omapdss/**

```
# cd /sys/devices/platform/omapdss/
# ls -F
display0@  manager1/                   overlay0/  subsystem@
display1@  microamps_requested_vdda_dac overlay1/  uevent
driver@    microamps_requested_vdds_dsi overlay2/
manager0/  modalias                    power/
```

DSS2 Details

```
# cd overlay0
# ls -F
enabled       input_size  name         position
global_alpha  manager     output_size  screen_width
# cat position
0,0
# cat manager
lcd
# cat output_size
640,480
```

Investigate mplayer

- Start BigBuckBunny
- Discover how it uses DSS

```
cd /sys/devices/platform/omapdss
ls
cd overlay0
cat enabled
```

- Repeat for overlay1 and 2
- Which are being used?
- cd to it

Play with scale and position

```
cd /sys/devices/platform/omapdss/overlay2
cat position
echo 100,100 >  position
cat output_size
echo 320,400 > output_size
```

- Try transparency

```
echo 127 > global_alpha
cd ../manager0
echo 1 > alpha_blending_enabled
echo 1 > trans_key_enabled
echo 65535 > trans_key_value
```

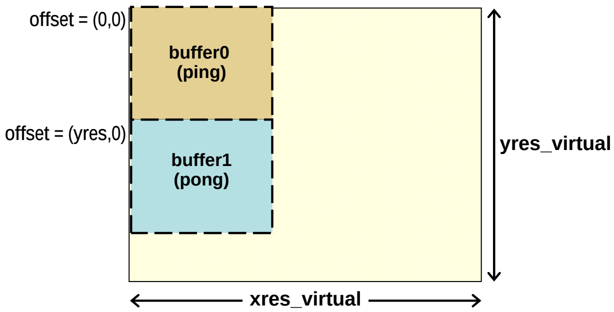
- Caution: You may have to disable an overlay before changing some values.

```
echo “0” > $ovl0/enabled
```

Outline

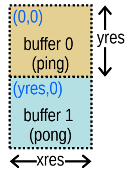
- **v4L2 Capture Driver**
  - Using mmap
  - v4L2 Coding
- **FBdev Display Driver**
  - Video Planes
  - FBdev Coding
- **Video Display Boot Args**
- **Lab Exercise**

Ping-pong Buffers with FBdev



- FBdev has no video buffer queue (provides direct access to the display device's frame buffer)
- Use FBIOPAN_DISPLAY to switch between 2 or more buffers in the virtual buffer space
- Use FBIO_WAITFORVSYNC to block process until current buffer scan completes, then switch.

Buffer Synchronization




```
pVirtualBuf = mmap(NULL,display_size * NUM_BUFS,
                   PROT_READ | PROT_WRITE,
                   MAP_SHARED,
                   FbdevFd, 0);
while(cond){
   // map next frame from virtual buffer
   display_index = (display_index +1) % NUM_BUFS;
   ioctl(pFbdevFd, FBIOGET_VSCREENINFO, &vinfo);
   vinfo.yoffset = vinfo.yres * display_index;

   // write pixels into next video frame
   genPicture(…);

   // switch to next frame
   ioctl(pFbdevFd, FBIOPAN_DISPLAY, &vinfo);

   // wait for current frame to complete
   ioctl(pFbdevFd, FBIO_WAITFORVSYNC, NULL);
}
```

Outline

- **v4L2 Capture Driver**
  - Using mmap
  - v4L2 Coding
- **FBdev Display Driver**
  - Video Planes
  - FBdev Coding
- **Video Display Boot Args**
- **Lab Exercise**

Lab 7

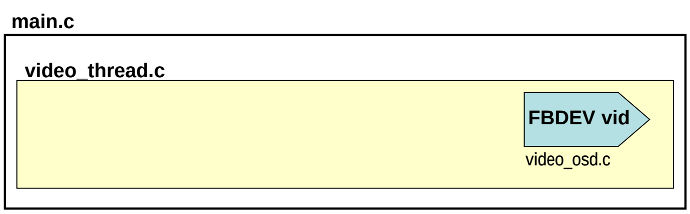
Lab 7a: OSD setup

Lab 7b: Video Recording

Lab 7c: Video Playback

Lab 7d: Video Loop-thru

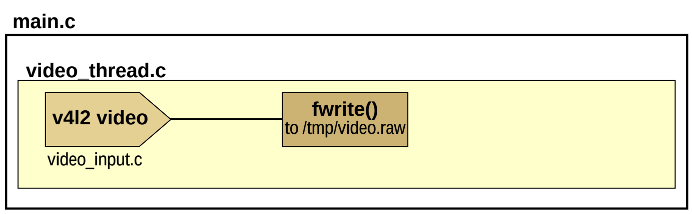
lab07a_osd_setup



- **Goal: to build an on_screen display for the DM644x using the FBDEV driver.**
- Inspection lab only.

1. Create your own custom picture for the OSD window (using gimp).
2. Inspect video_thread.c and helper functions (inside video_osd.c).
3. Build, run. Result: see your customer banner displayed on screen (no video yet…).

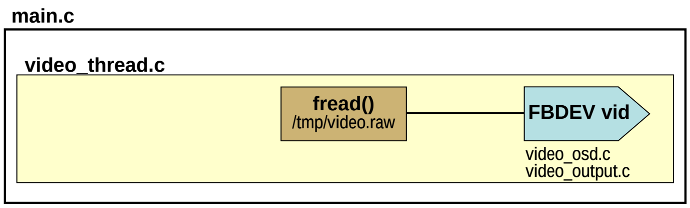
lab07b_video_record



- **Goal: Examine v4L2 video capture via a simple video recorder app.**
- Inspection lab only.

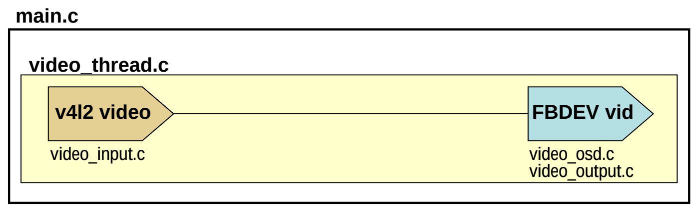
1. Examine helper functions (setup, cleanup, wait_for_frame) in video_input.c.
2. Examine video_thread_fxn() in video_thread.c.
3. Examine main.c (how the signal handler is created/used, then calls video_thread_fxn).
4. Build, run. Result: create a file (video.raw) that contains about 2s of captured video.

lab07c_video_playback



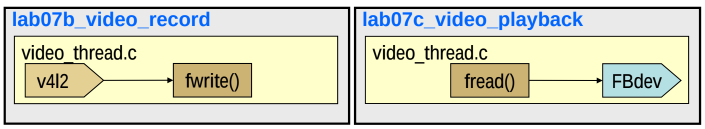
- **Goal: Examine FBdev display driver using a video display app. This app will play back the file recorded in lab07b (and add OSD from 07a).**
- Inspection lab only.
  1. Examine video_output.c and its helper functions.
  2. Ensure video.raw still exists in `/tmp` (and has a file size greater than zero).
  3. Build, run. Result: video.raw file is displayed on the screen (along with your OSD).

lab07d_video_loopthru



- **Goal: Combine the recorder (lab07b) and playback (lab07c) into a video loopthru application.**
- Hey – YOU get to do this yourself (no more inspection stuff…).
  1. Answer a few questions about the big picture (covered in the next few slides…).
  2. Copy files from lab07c (playback) to lab07d (loopthru).
  3. Add video *input* files from lab07b (record) to lab07d (loopthru).
  4. Make code modifications to stitch the record to the playback (covered in the next few slides…).
  5. Build, run. Result: video is captured (v4L2) and then displayed (FBdev) with your OSD.

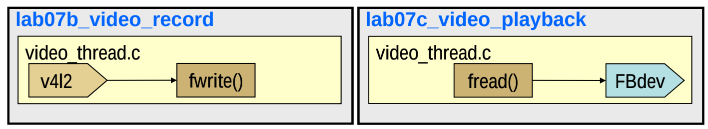
lab07d_video_loopthru



Lab 7b
What functions "get" a video frame?
Get data:
1.
2.
Put data:
1. fwrite() frame size
2. fwrite() the video frame

Lab 7c
What functions "display" a frame?
Get data:
1.
2.
Put Data:
1.
2.

lab07d_video_loopthru

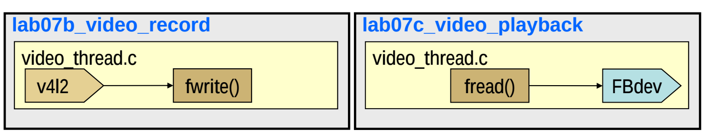


Lab 7b
What functions "get" a video frame?
Get data:
1. dequeue buffer

Put data:
1. fwrite() frame size
2. fwrite() the video frame

Lab 7c
What functions "display" a frame?
Get data:
1. fread() frame size
2. fread() the video frame
Put Data:
1. write frame
2. flip buffers

lab07d_video_loopthru



Lab 7b
What functions "get" a video frame?
Get data:
1. dequeue buffer

Put data:
1. fwrite() frame size
2. fwrite() the video frame

~~Lab 7c~~ Lab 7d
What functions "display" a frame?
Get data:
1. ~~fread() frame size~~
2. ~~fread() the video frame~~
Put Data:
1. write frame
2. flip buffers

For Lab07d:
- Take the code from lab07c
- Replace the fread's with the v4l2 capture code found in lab 07b

Lab 7c ➔ Lab 7d

(starting point)

Lab 7c		Lab 7d

video_thread.c
…
while {
fread frame size
fread frame to working buffer
-
-
flip buffers between working/displayed
}
	Replace with v4l2 code capture from Lab07b
	
	Add code to copy input to output and release the buffer
		video_thread.c
		…
		while {
		get input buffer (DQBUF)
		
		memcopy (input to output)
		release input buffer (QBUF)
		flip buffers between working/displayed
		}