OMAP35x EVM Linux PSP

User Guide

TEXAS INSTRUMENTS

03.00.00.03

Publication date 27 November 2009

IMPORTANT NOTICE

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address:

Texas Instruments,
Post Office Box 655303,
Dallas, Texas 75265

Table of Contents

List of Figures

List of Tables

Read This First

About This Manual

This document describes how to install and work with Texas Instruments' Platform Support Package (PSP) for OMAP35x platform running Linux.

This PSP provides a fundamental software platform for development, deployment and execution on. It abstracts the functionality provided by the hardware. The product forms the basis for all application development on this platform.

In this context, the document contains instructions to:

- Install the release
- Build the sources contained in the release

The document also provides detailed description of drivers and modules specific to this platform - as implemented in the PSP.

How to Use This Manual

This document includes the following chapters:

- Chapter 1, *Installation* - describes the installation procedure for OMAP35x EVM Linux PSP package.
- Chapter 2, *x-loader* - describes the procedure to build and execute the x-loader. and
- Chapter 3, *U-Boot* - describes the procedure to build and execute U-Boot.

- Chapter 4, *Kernel* - describes the procedure to build and execute the Linux kernel.
- Chapter 5, *Audio Driver* - describes the implementation of audio driver.
- Chapter 6, *Display Driver* - describes the implementation of video display driver.
- Chapter 7, *Resizer Driver* - describes the implementation of resizer driver.
- Chapter 8, *Daughter Card Module* - describes the features available on Daughter card.
- Chapter 9, *Capture Driver* - describes the implementation of video capture driver.
- Chapter 10, *USB Driver* - describes the implementation of USB driver.
- Chapter 11, *MMC Driver* - describes the implementation of MMC driver.
- Chapter 12, *Power Management* - describes the power management frameworks.

Please go through the Release Notes document available in the release package before starting the installation.

Notation of information elements

The document may contain these additional elements:

Warning

This is an example of warning message. It usually indicates a non-recoverable change, e.g. formatting a filesystem.

Caution

This is an example of caution message.

Important

This is an example of important message.

Note

This is an example of additional note. This usually indicates additional information in the current context.

Tip

This is an example of a useful tip.

If You Need Assistance

For any assistance, please send an mail to software support [mailto:softwaresupport@ti.com].

Trademarks

OMAP™ is a trademark of Texas Instruments Incorporated.

All other trademarks are the property of the respective owner.

Installation

Abstract

This chapter describes the layout of the Linux PSP package for OMAP35x EVM and steps to install on your development host.

Table of Contents

1.1. System Requirements

Hardware Requirements:

- OMAP35x EVM

Software Requirements:

- CodeSourcery ARM tool chain version 2009-q1

Important

OMAP EVM2 - Main Board (REV G) and OMAP35XX Processor Board with OMAP3530 ES3.1 processor - has been used for development and test for this release.

1.2. Installation

Extract the contents of release package with the following command:

```
$ tar -xvfz  OMAP35x-PSP-SDK-MM.mm.pp.bb.tgz
```

This creates a directory OMAP35x-PSP-SDK-MM.mm.pp.bb with the following contents:

```
\---AM35x-OMAP35x-PSP-SDK-MM.mm.pp.bb
    |    Software-manifest.html
    |    Arago-FS-Software-manifest.html
    +----docs
    |    |----Building-RootFs-Arago.html
    |    |----DataSheet-MM.mm.pp.bb.pdf
    |    |----ReleaseNotes-MM.mm.pp.bb.pdf
    |    |----am3517
    |    |    `----UserGuide-MM.mm.pp.bb.pdf
    |    |----omap3530
    |    |    `----UserGuide-MM.mm.pp.bb.pdf
    +----host-tools
    |    |----linux
    |    |    `----signGP
    |    |----src
    |    |    `----signGP.c
    +----images
    |    |----boot-strap
    |    |    |----am3517
    |    |    |    `----x-load.bin.ift
    |    |    |----omap3530
    |    |    |    `----x-load.bin.ift
    |    |----fs
    |    |    |----nfs-base.tar.gz
    |    |    |----ramdisk-base.gz
    |    |    |----rootfs-base.jffs2
    |    |    |----am3517
    |    |    |    |----nfs.tar.gz
    |    |    |    |----ramdisk.gz
    |    |    |    `----rootfs.jffs2
    |    |    |----omap3530
    |    |    |    |----nfs.tar.gz
    |    |    |    |----ramdisk.gz
    |    |    |    `----rootfs.jffs2
    |    |----kernel
    |    |    |----am3517
    |    |    |    `----uImage
    |    |    |----omap3530
    |    |    |    `----uImage
    |    |----u-boot
    |    |    |----am3517
    |    |    |    `----u-boot.bin
    |    |    |----omap3530
    |     |     |     `----u-boot.bin
    +----scripts
    |     |----am3517
    |     |     |----Readme.txt
    |     |     |----initenv-micron.txt
    |     |     `----reflash-micron.txt
    |     |----omap3530
    |     |     |----Readme.txt
    |     |     |----initenv-micron.txt
    |     |     `----reflash-micron.txt
    +----src
    |     |----boot-strap
    |     |     |----ChangeLog-MM.mm.pp.bb
    |     |     |----ShortLog
    |     |     |----Unified-patch-MM.mm.pp.bb.gz
    |     |     |----diffstat-MM.mm.pp.bb
    |     |     |----x-loader-patches-MM.mm.pp.bb.tar.gz
    |     |     `----x-loader-MM.mm.pp.bb.tar.gz
    |     |----examples
    |     |     |----examples.tar.gz
    |     |----kernel
    |     |     |----Readme.txt
    |     |     |----ChangeLog-MM.mm.pp.bb
    |     |     |----ShortLog
    |     |     |----Unified-patch-MM.mm.pp.bb.gz
    |     |     |----diffstat-MM.mm.pp.bb
    |     |     |----kernel-patches-MM.mm.pp.bb.tar.gz
    |     |     `----linux-MM.mm.pp.bb.tar.gz
    |     |----u-boot
    |           |----Readme.txt
    |           |----ChangeLog-MM.mm.pp.bb
    |           |----ShortLog
    |           |----Unified-patch-MM.mm.pp.bb.gz
    |           |----diffstat-MM.mm.pp.bb
    |           |----u-boot-patches-MM.mm.pp.bb.tar.gz
    |           `----u-boot-MM.mm.pp.bb.tar.gz
    +----test-suite
         `----lftb-MM.mm.pp.bb.tar.gz
```

Important

The values of *MM*, *mm*, *pp* and *bb* in this illustration will vary across the releases and actually depends on individual component versions.

1.3. Installation Steps

Instructions for initial setup of the EVM are contained in the OMAP3 EVM Users Guide included with the EVM kit.

1.4. Environment Setup

1. Set the environment variable PATH to contain the binaries of the CodeSourcery cross-compiler tool-chain.

2. For example, in bash:

   ```
   $ export PATH=/opt/toolchain/2009-q1/bin:$PATH
   ```

   Add location of u-boot tools to the PATH environment variable.

3. For example, in bash:

   ```
   $ export PATH=/opt/u-boot/tools:$PATH
   ```

Note

Actual instructions and the path setting will depend upon your shell and location of the tools

1.5. Setup NFS filesystem

This step is required when root filesystem is mounted from an NFS location.

Extract the contents of the NFS image (nfs.tar.gz) to a directory exported via NFS.

```
$ cd /opt/nfs/target
$ tar xvfz nfs.tar.gz
```

Important

Execute this command as **root** user. Some of the files included in this archive require root permissions for creation.

x-loader

Abstract

This chapter describes the steps required to build and execute the x-loader.

Table of Contents

2.1. Introduction

X-loader is loaded by ROM boot loader into internal RAM. X-loader support boot from OneNAND, NAND, MMC/SD.

2.2. Compiling X-Loader

Change to the base of the X-Loader directory.

```
$ cd ./x-load
```

Remove the intermediate files generated during build. This step is not necessary when building for the first time.

```
$ make CROSS_COMPILE=arm-none-linux-gnueabi- ARCH=arm distclean
```

Choose the configuration for OMAP3 EVM.

```
$ make  CROSS_COMPILE=arm-none-linux-gnueabi- ARCH=arm
 omap3evm_config
```

Initiate the build.

```
$ make CROSS_COMPILE=arm-none-linux-gnueabi- ARCH=arm
```

On successful completion, file `x-load.bin` will be created in the current directory.

2.2.1. Signing x-load.bin

The file `x-load.bin` needs to be signed before it can be used by the ROM bootloader. The **signGP** tool required for signing is available in the release package under the folder - `host-tools/linux`.

To sign the X-Loader binary:

```
$ signGP x-load.bin
```

The signing utility creates `x-load.bin.ift` in the current directory.

2.3. Saving x-loader on target media

2.3.1. OneNAND

To flash the x-loader into OneNAND, execute following commands at the U-Boot prompt:

```
OMAP3_EVM # mw.b 0x80000000 0xFF 0x100000
OMAP3_EVM # tftp 0x80000000 x-load.bin.ift
```

Note

On Older U-boot versions(from PSP 1.0.x releases), the OneNand will have to be unlocked before write/erase operation. For subsequent releases of u-boot, this step is not required.

```
OMAP3_EVM # onenand unlock 0x000000 0x20000
```

```
OMAP3_EVM # onenand erase 0x00000000 0x00080000
OMAP3_EVM # onenand write 0x80000000 0x0 0x10000
```

2.3.2. NAND

To flash the x-loader into Micron NAND, execute following commands at the U-Boot prompt:

```
OMAP3_EVM # mw.b 0x80000000 0xFF 0x100000
OMAP3_EVM # tftp 0x80000000 x-load.bin.ift
OMAP3_EVM # nand erase 0 40000
OMAP3_EVM # nandecc hw
OMAP3_EVM # nand write.i 0x80000000 0 40000
```

Note

Syntax of `nandecc` command has changed since from PSP 1.0.x releases.

2.3.3. MMC/SD Card

Copy `x-load.bin.ift` to the MMC/SD card and rename it as `MLO`.

Important

The ROM bootloader scans only initial FAT entries for this binary. Ensure that `MLO` is the first file to be copied on the card.

Once the U-Boot and Linux kernel are built, `u-boot.bin`, `uImage` and `ramdisk.gz` should be copied to the card.

Important

The MMC/SD card should have a valid bootable partition on the card before it can be used as boot media. See Section 14.1, “Creating bootable partition on MMC/SD Card” for necessary steps.

U-Boot

Abstract

This chapter describes the steps required to build and configure u-boot to use different filesystems during the kernel boot.

It also describes new commands for managing bad blocks.

Table of Contents

3.1. Compiling U-Boot

Change to the base of the u-boot directory.

```
$ cd ./u-boot
```

Remove the intermediate files generated during build. This step is not necessary when building for the first time.

```
$ make CROSS_COMPILE=arm-none-linux-gnueabi- ARCH=arm distclean
```

Choose the configuration for OMAP3 EVM.

```
$ make CROSS_COMPILE=arm-none-linux-gnueabi- ARCH=arm
 omap3_evm_config
```

Initiate the build.

```
$ make CROSS_COMPILE=arm-none-linux-gnueabi- ARCH=arm
```

On successful completion, file `u-boot.bin` will be created in the current directory.

Note

The u-boot build commands have changed from the previous release.

3.2. Flashing U-Boot

3.2.1. OneNAND

To flash `u-boot.bin` to the OneNAND execute the commands listed below:

```
OMAP3_EVM # mw.b 0x80000000 0xFF 0x100000
OMAP3_EVM # tftp 0x80000000 u-boot.bin
```

Note

With Older U-boot versions (from PSP 1.0.x releases), the OneNand will have to be unlocked before erase/write operation. For subsequent releases of u-boot, this step is not required.

```
OMAP3_EVM # onenand unlock 0x000000 0x300000
```

```
OMAP3_EVM # onenand erase 0x00080000  0x001C0000
OMAP3_EVM # onenand write 0x80000000 0x80000 0x1C0000
```

3.2.2. Micron NAND

To flash `u-boot.bin` to the Micron NAND execute the commands listed below:

```
OMAP3_EVM # mw.b 0x80000000 0xFF 0x100000
OMAP3_EVM # tftp 0x80000000 u-boot.bin
OMAP3_EVM # nand erase 0x80000 0x1C0000
OMAP3_EVM # nandecc sw
OMAP3_EVM # nand write.i 0x80000000 0x80000 0x1C0000
```

Note

The syntax of `nandecc` command has changed since the PSP 1.0.x release.

3.3. Configuring U-Boot

This section assumes that EVM has been setup properly.

1. Enable UART1 on the EVM : On Jumper J8 select 1-2
2. Connect EVM (UART1) to the HOST PC through serial cable.
3. Start a terminal emulator (e.g. Hyperterm) on the HOST PC.
4. Power on EVM and wait for u-boot to come up.

Important

Some commands entered on the console are long. The command text may appear wrapped in the document. Wherever indicated, these commands must be entered in a single line.

3.3.1. Using ramdisk image

Set the `bootargs`:

```
OMAP3_EVM # setenv bootargs mem=128M console=ttyS0,115200n8
          root=/dev/ram0 rw initrd=0x81600000,40M ip=dhcp
 mpurate=600
```

Note

The entire command should be entered in a single line.

Set the `bootcmd`:

```
OMAP3_EVM # setenv bootcmd 'dhcp;
          tftp 0x80000000 uImage;tftp 0x81600000 ramdisk.gz;
          bootm 80000000'
```

Note

The entire command should be entered in a single line.

3.3.2. Using NFS (Default U-Boot configuration)

Set the `bootargs`:

```
OMAP3_EVM # setenv bootargs console=ttyS0,115200n8 noinitrd
          ip=dhcp rw root=/dev/nfs nfsroot=192.168.1.101:
          /opt/nfs/target,nolock mem=128M mpurate=600
```

Note

- The entire command should be entered in a single line.
- Replace NFS server IP address(192.168.1.101) and mount path (`/opt/nfs/target`) with actuals based on your NFS server setting.

Set the `bootcmd`:

```
OMAP3_EVM # setenv 'bootcmd dhcp;tftp 0x80000000 uImage;bootm'
```

3.3.2.1. Using NFS without DHCP

Disable the DHCP support in the build configuration:

```
Device Drivers
    Networking Support
       Networking options
             IP: DHCP Support
```

Set the `bootargs`:

```
OMAP3_EVM # setenv bootargs 'console=ttyS0,115200n8 noinitrd rw
          root=/dev/nfs nfsroot=192.168.1.101:
          /opt/nfs/target,nolock mem=128M mpurate=600'
```

Set the `bootcmd`:

```
OMAP3_EVM # setenv bootcmd 'dhcp;setenv addip setenv bootargs
 $(bootargs)
          ip=$(ipaddr):$(serverip):$(gatewayip):$(netmask):
          $(hostname)::off eth=$(ethaddr);run addip;
          tftp 0x80000000 uImage; bootm 0x80000000'
```

Note

The entire command should be entered in a single line.

Important

Save the changes to these variables on the flash with u-boot command - **saveenv**.

3.4. Managing OneNAND

The u-boot has been updated to include bad block management for OneNAND. These updates also impacted behavior of existing OneNAND commands. This section describes the new and modified commands added for the purpose.

3.4.1. Marking a bad block

To forcefully mark a block as bad:

```
OMAP3_EVM # onenand markbad <offset>
```

For example, to mark block 32 (assuming erase block size of 128Kbytes) as bad block - offset = blocknum * 128 * 1024:

```
OMAP3_EVM # onenand markbad 0x400000
```

3.4.2. Erasing OneNAND

To erase OneNAND blocks in the address range:

```
OMAP3_EVM # onenand erase <stoffaddr> <endoffaddr>
 or
OMAP3_EVM # onenand erase block <stblknum-endblknum>
```

Note

The behavior of this command was modified.

This commands skips bad blocks (both factory or user marked) encountered within the specified range.

Important

If the erase operation fails, the block is marked bad and the command aborts. To continue erase operation, the command needs to be re-executed for the remaining blocks in the range.

For example, to erase blocks 32 through 34:

```
OMAP3_EVM # onenand erase 0x00400000 0x00440000
 or
OMAP3_EVM # onenand erase block 32-34
```

3.4.3. Writing to OneNAND

To write *len* bytes of data from a memory buffer located at *addr*to the OneNAND block *offset*:

```
OMAP3_EVM # onenand write <addr> <offset> <len>
```

Note

The behavior of this command was modified.

If a bad block is encountered during the write operation, it is skipped and the write operation continues from next 'good' block.

Important

If the write fails on ECC check, the block where the failure occurred is marked bad and write operation is aborted. The command needs to be re- executed to complete the write operation. The offset and length for reading have to be page aligned else the command will abort.

For example, to write 0x40000 bytes from memory buffer at address 0x80000000 to OneNAND - starting at block 32 (offset 0x400000):

```
OMAP3_EVM # onenand write 0x80000000 0x400000 0x40000
```

3.4.4. Reading from OneNAND

To read *len* bytes of data from OneNAND block at *offset* to memory buffer located at *addr*:

```
OMAP3_EVM # onenand read <addr> <offset> <len>
```

Note

The behavior of this command was modified.

If a bad block is encountered during the read operation, it is skipped and the read operation continues from next 'good' block.

Important

If the read fails on ECC check, the block where the failure occurred is marked bad and read operation is aborted. The command needs to be re- executed to complete the read operation. But, the data in just marked bad block is irrecoverably lost. The offset and length for reading have to be page aligned else the command will abort.

For example, to read 0x40000 bytes from OneNAND - starting at block 32 (offset 0x400000) to memory buffer at address 0x80000000:

```
OMAP3_EVM # onenand read 0x80000000 0x400000 0x40000
```

3.4.5. Scrubbing OneNAND

This command operation is similar to the `erase` command, with a difference that it doesn't care for bad blocks. It attempts to erase all blocks in the specified address range.

To scrub OneNAND blocks in the address range:

```
OMAP3_EVM # onenand scrub <stoffaddr> <eoffaddr>
 or
OMAP3_EVM # onenand scrub block <stblknum-endblknum>
```

Note

This is a new command.

Important

The command does not check whether the block is a user marked or factory marked bad block. This command fails on a factory marked bad block.

Important

If the erase operation fails, the block is marked as bad and the command aborts. The command needs to be re-executed for the remaining blocks in the range.

3.5. Managing NAND

The u-boot has been updated to include NAND flash support

3.5.1. Marking a bad block

To forcefully mark a block as bad:

```
OMAP3_EVM # nand markbad <offset>
```

Note

This is a new command.

For example, to mark block 32 (assuming erase block size of 128Kbytes) as bad block - offset = blocknum * 128 * 1024:

```
OMAP3_EVM # nand markbad 0x400000
```

3.5.2. Viewing bad blocks

Gives a list of bad blocks in NAND

```
OMAP3_EVM # nand bad
```

Note

The user marked bad blocks can be viewed by using this command only after a reset.

3.5.3. Erasing NAND

To erase NAND blocks in the address range or using block numbers

```
OMAP3_EVM # nand erase <stoffaddr> <len>
```

Note

The behavior of this command was modified.

This commands skips bad blocks (both factory or user marked) encountered within the specified range.

Important

If the erase operation fails, the block is marked bad and the command aborts. To continue erase operation, the command needs to be re-executed for the remaining blocks in the range.

For example, to erase blocks 32 through 34

```
OMAP3_EVM # nand erase 0x00400000 0x40000
```

3.5.4. Writing to NAND

To write *len* bytes of data from a memory buffer located at *addr*to the NAND block *offset*:

```
OMAP3_EVM # nand write <addr> <offset> <len>
```

Note

The behavior of this command was modified.

If a bad block is encountered during the write operation, it is skipped and the write operation continues from next 'good' block.

Important

If the write fails on ECC check, the block where the failure occurred is marked bad and write operation is aborted. The command needs to be re- executed to complete the write operation. The offset and length for reading have to be page aligned else the command will abort.

For example, to write 0x40000 bytes from memory buffer at address 0x80000000 to NAND - starting at block 32 (offset 0x400000):

```
OMAP3_EVM # nand write 0x80000000 0x400000 0x40000
```

3.5.5. Reading from NAND

To read *len* bytes of data from NAND block at *offset* to memory buffer located at *addr*:

```
OMAP3_EVM # nand read <addr> <offset> <len>
```

Note

The behavior of this command was modified.

If a bad block is encountered during the read operation, it is skipped and the read operation continues from next 'good' block.

Important

If the read fails on ECC check, the block where the failure occurred is marked bad and read operation is aborted. The command needs to be re- executed to complete the read operation. But, the data in just marked bad block is irrecoverably lost. The offset and length for reading have to be page aligned else the command will abort.

For example, to read 0x40000 bytes from NAND - starting at block 32 (offset 0x400000) to memory buffer at address 0x80000000:

```
OMAP3_EVM # nand read 0x80000000 0x400000 0x40000
```

3.5.6. Unlocking NAND address space

To unlock NAND flash for writing

```
OMAP3_EVM # nand unlock <offset> <len>
```

Note

This is a new command.

For example, to unlock block 32 (assuming erase block size of 128Kbytes)

```
OMAP3_EVM # nand unlock 0x20000
```

3.5.7. NAND ECC algorithm selection

To select ECC algorithm for NAND

```
OMAP3_EVM # nandecc <sw/hw>
```

Note

To write X-loader from U-Boot, ECC algorithm to be selected is HW since bootrom uses this algorithm for reading. To write U-Boot from U-Boot, ECC algorithm to be selected is SW.

```
OMAP3_EVM # nandecc hw
 or
OMAP3_EVM # nandecc sw
```

3.6. MUSB Host support

The u-boot now supports USB Mass storage class (MSC) on the MUSB port. It can be used to load any file from USB MSC device.

Important

Ensure that USB MSC device is connected to the MUSB port before issuing any of the commands described in this section.

To initialize the USB subsystem:

```
OMAP3_EVM# usb start
```

All the connected devices will, now, get recognized.

To view all connected USB devices in a tree form:

```
OMAP3_EVM# usb tree
```

To view all Mass Storage USB devices:

```
OMAP3_EVM# usb storage
```

To view filesystem information of MSC device:

```
OMAP3_EVM# fatinfo usb D:P
```

This command shows filesystem information of a partition on the MSC device.

Note

Substitute `D` with the storage device number and `p` with the partition number on the device.

To load a file from MSC device:

```
OMAP3_EVM# fatload usb D:P <addr>ADDR> <file-name>
```

This command reads specified file from MSC device and writes its contents at the specified address.

Note

Substitute `D` with the storage device number and `p` with the partition number on the device.

Kernel

Abstract

This chapter describes the steps required to build and configure the Linux kernel. It also provides basic steps to boot kernel on the EVM.

Table of Contents

4.1. Compiling Linux Kernel

Change to the base of the Linux source directory.

Create default configuration for the OMAP3EVM.

```
$ make CROSS_COMPILE=arm-none-linux-gnueabi- ARCH=arm
 omap3_evm_defconfig
```

Initiate the build.

Note

For the kernel image (`uImage`) to be built, `mkimage` utility must be included in the path. `mkimage` utility is generated (under tools folder) while building `u-boot.bin`.

```
$ make CROSS_COMPILE=arm-none-linux-gnueabi- ARCH=arm uImage
```

On successful completion, file `uImage` will be created in the directory `./arch/arm/boot`.

Copy this file to the root directory of your TFTP server.

4.2. Configuring Linux Kernel

Before building the Linux kernel, it should be configured for a specific platform. This chapter describes steps to configure the kernel for **OMAP35x EVM** and illustrates related configuration items for reference.

To create default configuration:

```
$ make CROSS_COMPILE=arm-none-linux-gnueabi- ARCH=arm
 omap3_evm_defconfig
```

To view configuration interactively:

```
$ make CROSS_COMPILE=arm-none-linux-gnueabi- ARCH=arm menuconfig
```

From the onscreen menu, select **System Type**:

```
    General setup  --->
[*] Enable loadable module support  --->
[*] Enable the block layer  --->
    System Type  --->
    Bus support  --->
    Kernel Features  --->
    ...
    ...
```

These items would be selected by default:

- OMAP35x Family
- OMAP 3530 EVM board

```
    ARM system type (TI OMAP)  --->
    TI OMAP Implementations  --->
-*- OMAP34xx Based System
-*-   OMAP3430 support
[*] OMAP35x Family
    *** OMAP Board Type ***
[ ] OMAP3 LDP board
[ ] OMAP 3430 SDP board
[*] OMAP 3530 EVM board
    ...
    ...
```

Choose **Exit** to successively to return to previous menu(s) and eventually back to the shell.

Some of the *key* drivers are enabled in the default configuration are:

- Serial port
- Mentor USB in OTG mode
- USB EHCI
- Ethernet
- MMC/SD
- Video Display
- Audio
- NAND and OneNAND
- Touchscreen

4.3. Booting Linux Kernel

4.3.1. Selecting boot mode

The boot mode is selected by DIP switch SW4 on the main board. This selection identifies the location from where the *x-loader* and *u-boot* binaries are loaded for execution.

The switch positions differ across the boards populated with Samsung OneNAND and Micron NAND parts.

4.3.1.1. EVM populated with Samsung OneNAND

To boot from OneNAND, use either of following switch settings:

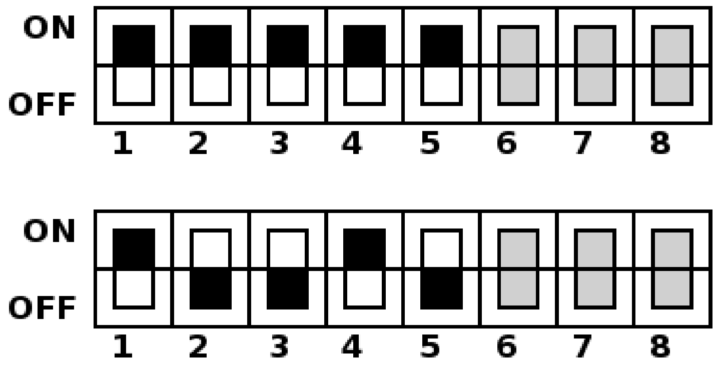


Figure 4.1. Boot from OneNAND

To boot from MMC, use either of following switch settings:

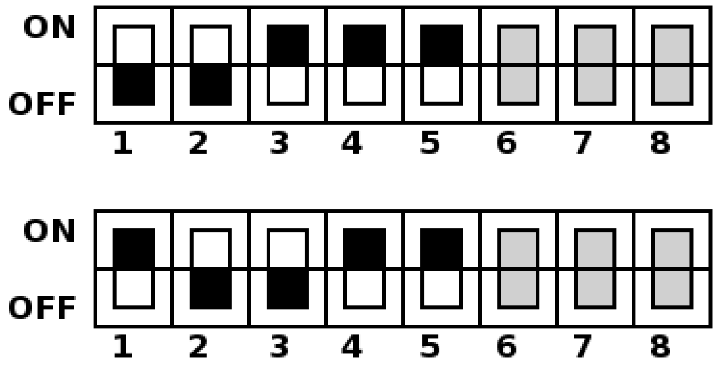


Figure 4.2. Boot from MMC (on EVM with Samsung OneNAND)

4.3.1.2. EVM populated with Micron NAND

To boot from NAND, use either of following switch settings:

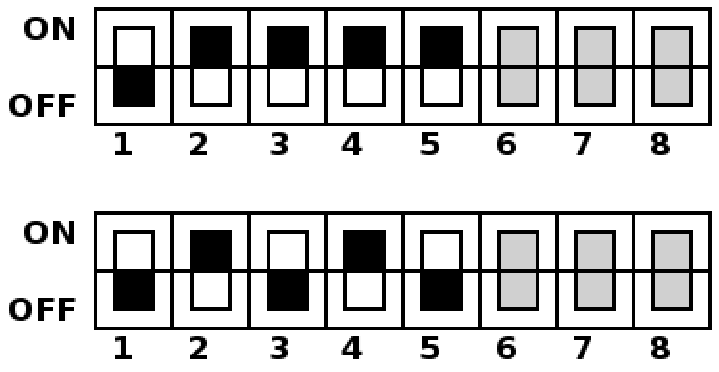


Figure 4.3. Boot from NAND

To boot from MMC, use either of following switch settings:

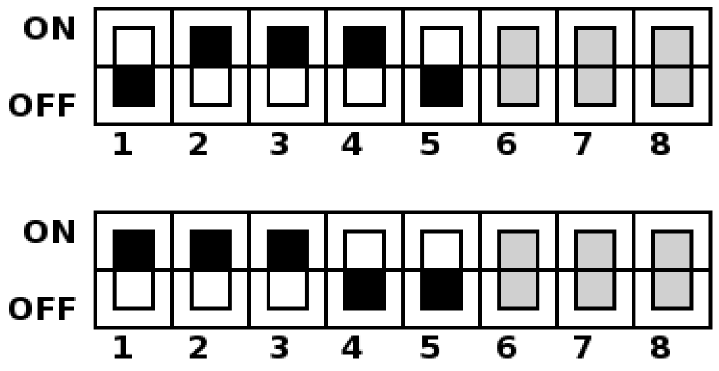


Figure 4.4. Boot from MMC (on EVM with Micron NAND)

Note

Position of switches SW4-6, SW4-7 and SW4-8 is **Don't Care**. These are grayed in the illustrations above.

Important

Ensure that u-boot environment variables `bootargs` and `bootcmd` are properly set. See section 3.3 for more details.

4.3.2. Boot from NAND/OneNAND

Power on EVM and wait for u-boot to come up.

When kernel image and filesystem are flashed on the NAND/OneNAND device:

```
/* NAND Boot mode */
$ nand read.i 0x80000000 280000 500000
$ setenv bootargs 'mem=128M console=ttyS0,115200n8 noinitrd
 root=/dev/mtdblock4 rw rootfstype=jffs2 ip=dhcp'
$ bootm 0x80000000

/* ONENAND Boot mode*/
$ onenand read 0x80000000 0x280000 0x0220000
$ setenv bootargs 'mem=128M console=ttyS0,115200n8 noinitrd
 root=/dev/mtdblock4 rw rootfstype=jffs2 ip=dhcp'
$ bootm 0x80000000
```

When kernel image is flashed on the NAND/OneNAND device, and NFS mounted filesystem is being used:

```
/* NAND Boot mode */
$ nand read.i 0x80000000 280000 500000
$ setenv bootargs 'mem=128M console=ttyS0,115200n8 noinitrd rw
 root=/dev/nfs nfsroot=/mnt/nfs,nolock ip=dhcp'
$ bootm 0x80000000

/* ONENAND Boot mode*/
$ onenand read 0x80000000 0x280000 0x0220000
$ setenv bootargs 'mem=128M console=ttyS0,115200n8 noinitrd rw
 root=/dev/nfs nfsroot=/mnt/nfs,nolock ip=dhcp'
$ bootm 0x80000000
```

When kernel image and ramdisk image are fetched from a tftp server:

```
$ setenv autoload no
$ dhcp
$ setenv serverip <Server IP Address>
$ tftp 0x80000000 uImage
$ tftp 0x82000000 ramdisk.gz
$ setenv bootargs 'mem=128M console=ttyS0,115200n8 root=/dev/ram0
 initrd=0x82000000,40M ramdisk_size=32768 ip=dhcp'
$ bootm 0x80000000
```

4.3.3. Boot from MMC

Power on EVM and wait for u-boot to come up.

When kernel image and filesystem (ramdisk) are available on the MMC card:

```
$ mmc init
$ fatload mmc 0 0x80000000 uImage
$ fatload mmc 0 0x80000000 ramdisk.gz
$ setenv bootargs 'mem=128M console=ttyS0,115200n8 root=/dev/ram0
 initrd=0x82000000,40M ramdisk_size=32768 ip=dhcp'
$ bootm 0x80000000
```

When kernel image is available on the MMC card and NFS mounted filesystem is being used:

```
$ mmc init
$ fatload mmc 0 0x80000000 uImage
$ setenv bootargs 'mem=128M console=ttyS0,115200n8 noinitrd rw
 root=/dev/nfs nfsroot=/mnt/nfs,nolock ip=dhcp'
$ bootm 0x80000000
```

When kernel image is available on the MMC card and filesystem on the NAND device is used:

```
$ mmc init
$ fatload mmc 0 0x80000000 uImage
$ setenv bootargs 'mem=128M console=ttyS0,115200n8 noinitrd
 root=/dev/mtdblock4 rw rootfstype=jffs2 ip=dhcp'
$ bootm 0x80000000
```

Note

Once the Linux kernel boots, login as "root". No password is required.

Audio Driver

Abstract

This chapter provides details on how to configure the audio driver, its interfaces and a simple application code illustrates the use of this interface.

Table of Contents

5.1. Introduction

The TPS65950 audio module contains audio analog inputs and outputs. It is connected to the main OMAP35x processor through the TDM/I2S interface (audio interface) and used to transmit and receive audio data. The TPS65950 codec is connected via Multi-Channel Buffered Serial Port (McBSP) interface, a communication peripheral, to the main processor.

McBSP provides a full-duplex direct serial interface between the device (OMAP35x processor) and other devices in the system such as the TPS65950 codec. It provides a direct interface to industry standard codecs, analog interface chips (AICs) and other serially connected A/D and D/A devices:

- Inter-IC Sound (I2S) compliant devices
- Pulse Code Modulation (PCM) devices
- Time Division Multiplexed (TDM) bus devices.

The TPS65950 audio module is controlled by internal registers that can be accessed by the high speed I2C control interface.

This user manual defines and describes the usage of user level and platform level interfaces of the ALSA SoC Audio driver.

5.1.1. References

1. ALSA SoC Project Homepage [http://www.alsa-project.org/main/index.php/ASoC]
2. ALSA Project Homepage [http://www.alsa-project.org/main/index.php/Main_Page]
3. ALSA User Space Library [http://www.alsa-project.org/alsa-doc/alsa-lib/]
4. Using ALSA Audio API [http://www.equalarea.com/paul/alsa-audio.html/]

   Author: Paul Davis
5. TPS65950: Integrated Power Management IC with 3 DC/DC's, 11 LDO's, Audio Codec, USB HS Transceiver, Charger [http://focus.ti.com/docs/prod/folders/print/tps65950.html]

5.1.2. Acronyms & Definitions

Acronym	Definition
ALSA	Advanced Linux Sound Architecture

Acronym	**Definition**
ALSA SoC	ALSA System on Chip
DMA	Direct Memory Access
I2C	Inter-Integrated Circuit
McBSP	Multi-channel Buffered Serial Port
PCM	Pulse Code Modulation
TDM	Time Division Multiplexing
OSS	Open Sound System
I2S	Inter-IC Sound

Table 5.1. Audio Driver: Acronyms

5.2. Features

This section describes the features supported by ALSA SoC Audio driver.

- Supports TPS65950 audio codec in ALSA SoC framework.
- Supports audio in both mono and stereo modes.
- Multiple sample rate support (8 KHz, 11.025 KHz, 12 KHz, 16 KHz, 22.05 KHz, 24 KHz, 32 KHz, 44.1 KHz and 48 KHz) for both capture and playback.
- Supports simultaneous playback and record (full-duplex mode).
- 16 Bit Little Endian Signed PCM data.
- I2S mode of operation.
- Interleaved access mode.
- Start, stop, pause and resume feature.
- Supports mixer interface for TPS65950 audio codec.
- McBSP is configured as slave and TPS65950 Codec is configured as master.
- Supports MMAP mode for both playback and capture.

Important

Audio capture channels AUXL and AUXR are by default disabled. To enable them, use the following 'amixer' commands:

amixer cset name='Analog Left AUXL Capture Switch' 1

amixer cset name='Analog Right AUXR Capture Switch' 1

5.3. ALSA SoC Architecture

5.3.1. Introduction

The overall project goal of the ALSA System on Chip (ASoC) layer is to provide better ALSA support for embedded system on chip processors and portable audio codecs. Currently there is some support in the kernel for SoC audio, however it has some limitations:

- Currently, codec drivers are often tightly coupled to the underlying SoC cpu. This is not really ideal and leads to code duplication.
- There is no standard method to signal user initiated audio events e.g. Headphone/Mic insertion, Headphone/Mic detection after an insertion event.
- Current drivers tend to power up the entire codec when playing (or recording) audio. This is fine for a PC, but tends to waste a lot of power on portable devices. There is also no support for saving power via changing codec oversampling rates, bias currents, etc.

5.3.2. Design

The ASoC layer is designed to address these issues and provide the following features:

- Codec independence: Allows reuse of codec drivers on other platforms and machines.
- Easy I2S/PCM audio interface setup between codec and SoC. Each SoC interface and codec registers it's audio interface capabilities with the core and are subsequently matched and configured when the application hw params are known.
- Dynamic Audio Power Management (DAPM): DAPM automatically sets the codec to it's minimum power state at all times. This includes powering up/down internal power blocks depending on the internal codec audio routing and any active streams.
- Pop and click reduction: Pops and clicks can be reduced by powering the codec up/down in the correct sequence (including using digital mute). ASoC signals the codec when to change power states.

To achieve all this, ASoC basically splits an embedded audio system into three components:

- Codec driver: The codec driver is platform independent and contains audio controls, audio interface capabilities, codec dapm definition and codec IO functions.
- Platform driver: The platform driver contains the audio dma engine and audio interface drivers (e.g. I2S, AC97, PCM) for that platform.

- Machine driver: The machine driver handles any machine specific controls and audio events i.e. turning on an amp at start of playback.

Following architecture diagram shows all the components and the interactions among them:

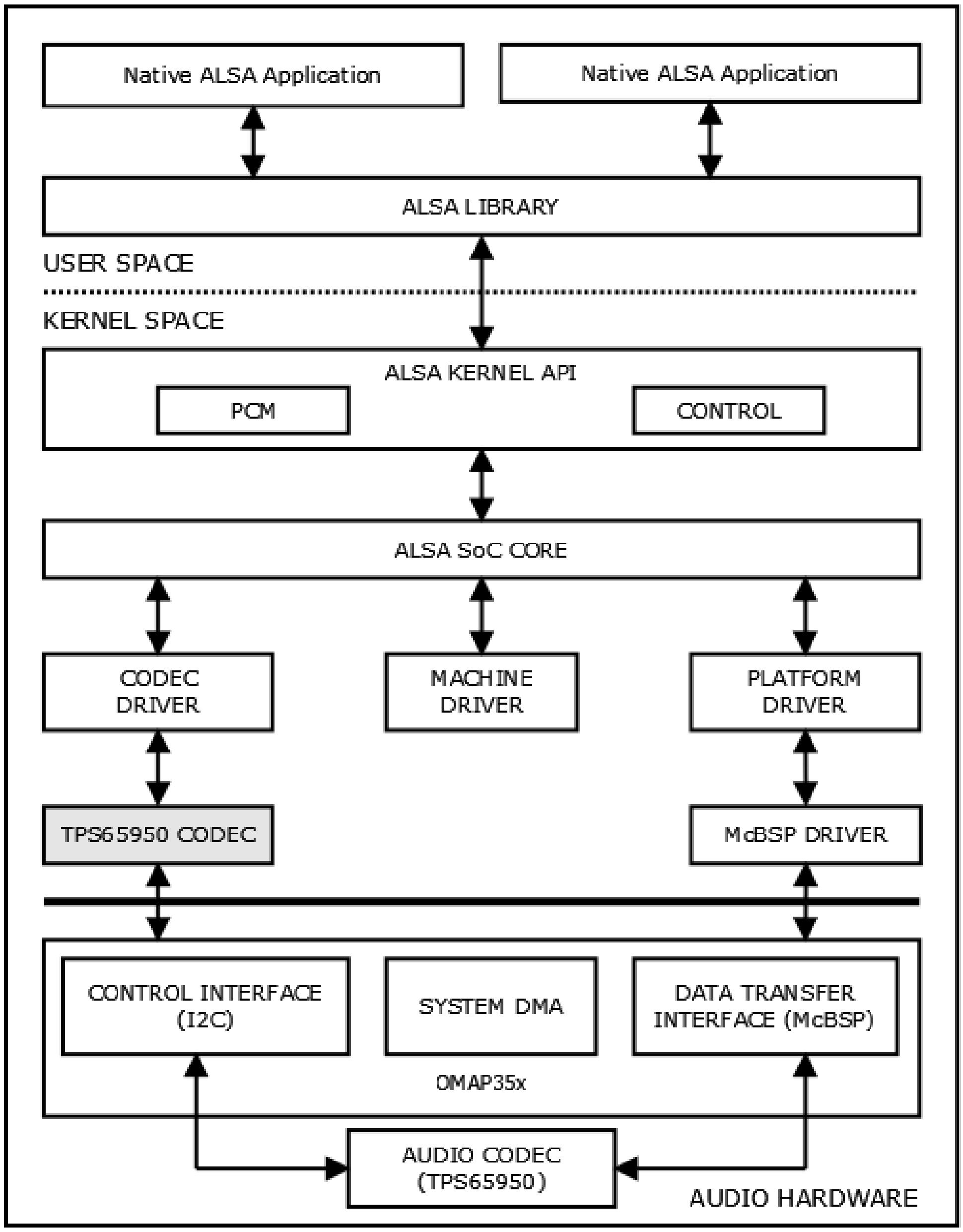


Figure 5.1. ALSA SoC Architecture

5.4. Configuration

To enable/disable audio support, start the *Linux Kernel Configuration* tool.

```
$ make menuconfig
```

Select *Device Drivers* from the main menu.

```
    ...
    ...
    Power management options  --->
[*] Networking support  --->
    Device Drivers  --->
    File systems  --->
    Kernel hacking  --->
    ...
    ...
```

Select *Sound card support* as shown here:

```
    ...
    ...
    Multimedia devices  --->
    Graphics support   --->
<*> Sound card support  --->
[*] HID Devices  --->
[*] USB support  --->
    ...
    ...
```

Select *Advanced Linux Sound Architecture* as shown here:

```
--- Sound card support
<*>   Advanced Linux Sound Architecture  --->
< >   Open Sound System (DEPRECATED)  --->
```

Select *ALSA for SoC audio support* as shown here:

```
    ...
    ...
[*] ARM sound devices (NEW)  --->
[*] USB sound devices (NEW)  --->
<*> ALSA for SoC audio support  --->
```

Select SoC Audio for TI OMAP EVMs as shown here:

```
--- ALSA for SoC audio support
<*>   SoC Audio for the Texas Instruments OMAP chips
<*>   SoC Audio support for OMAP3EVM board
< >   Build all ASoC CODEC drivers (NEW)
```

Make sure that McBSP support is enabled. To check the same:

Select *System Type* from the main menu.

```
    ...
    ...
[*] Enable loadable module support  --->
[*] Enable the block layer  --->
    System Type  --->
    Bus support  --->
    Kernel Features  --->
    ...
    ...
```

Select *TI OMAP Implementations* as shown here:

```
    ARM system type (TI OMAP)   --->
    TI OMAP Implementations  --->
-*- OMAP34xx Based System
-*-   OMAP3430 support
    ...
    ...
```

McBSP support should be selected:

```
    ...
    ...
[ ]   Multiplexing debug output
[*]   Warn about pins the bootloader didn't set up
-*- McBSP support
< > Mailbox framework support
    System timer (Use 32KHz timer)  --->
    ...
    ...
```

5.5. Application Interface

This section provides the details of the Application Interface for the ALSA Audio driver.

Application developer uses ALSA-lib, a user space library, rather than the kernel API. The library offers 100% of the functionality of the kernel API, but adds major improvements in usability, making the application code simpler and better looking.

The online-documentation for the same is available at:

http://www.alsa-project.org/alsa-doc/alsa-lib/

5.5.1. Device Interface

The operational interface in `/dev/` contains three main types of devices: (a) PCM devices for recording or playing digitized sound samples, (b) CTL devices that allow manipulating the internal mixer and routing of the card, and (c) MIDI devices to control the MIDI port of the card, if any.

Name	**Description**
/dev/snd/controlC0	Control devices (i.e. mixer, etc).
/dev/snd/pcmC0D0c	PCM Card 0 Device 0 Capture device.
/dev/snd/pcmC0D0p	PCM Card 0 Device 0 Playback device..

Table 5.2. Device Interface

5.5.2. Proc Interface

The `/proc/asound` kernel interface is a status and configuration interface. A lot of useful information about the sound system can be found in the `/proc/asound` subdirectory.

See the table below for different proc entries in `/proc/asound`:

Name	**Description**
cards	List of registered cards.
version	Version and date the driver was built on.
devices	List of registered ALSA devices.
pcm	The list of allocated PCM streams.
cardX/ (X = 0-7)	The card specific directory.
cardX/pcm0p	The directory of the given PCM playback stream.

Name	**Description**
cardX/pcm0c	The directory of the given PCM capture stream.

Table 5.3. Proc Interface

5.5.3. Commonly Used APIs

Some of the commonly used APIs to write an ALSA based application are:

Name	**Description**
snd_pcm_open	Opens a PCM stream.
snd_pcm_close	Closes a previously opened PCM stream.
snd_pcm_hw_params_any	Fill params with a full configuration space for a PCM.
snd_pcm_hw_params_test_<<parameter>>	Test the availability of important parameters like number of channels, sample rate etc. snd_pcm_hw_params_test_format, snd_pcm_hw_params_test_rate, etc.
snd_pcm_hw_params_set_<<parameter>>	Set the different configuration parameters. snd_pcm_hw_params_set_format, snd_pcm_hw_params_set_rate, etc.
snd_pcm_hw_params	Install one PCM hardware configuration chosen from a configuration space.
snd_pcm_writei	Write interleaved frames to a PCM.
snd_pcm_readi	Read interleaved frames from a PCM.
snd_pcm_prepare	Prepare PCM for use.
snd_pcm_drop	Stop a PCM dropping pending frames.
snd_pcm_drain	Stop a PCM preserving pending frames.

Table 5.4. Commonly Used APIs

5.5.4. User Space Interactions

This section depicts the sequence of operations for a simple playback and capture application.

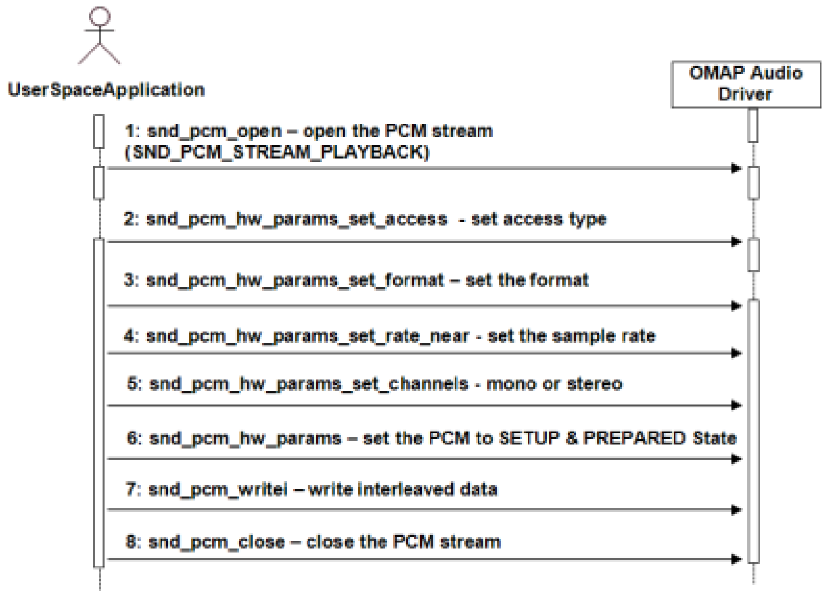

UserSpaceApplication
OMAP Audio Driver
1: snd_pcm_open – open the PCM stream (SND_PCM_STREAM_PLAYBACK)
2: snd_pcm_hw_params_set_access - set access type
3: snd_pcm_hw_params_set_format – set the format
4: snd_pcm_hw_params_set_rate_near - set the sample rate
5: snd_pcm_hw_params_set_channels - mono or stereo
6: snd_pcm_hw_params – set the PCM to SETUP & PREPARED State
7: snd_pcm_writei – write interleaved data
8: snd_pcm_close – close the PCM stream

Figure 5.2. OMAP3 ALSA Driver : Half duplex playback

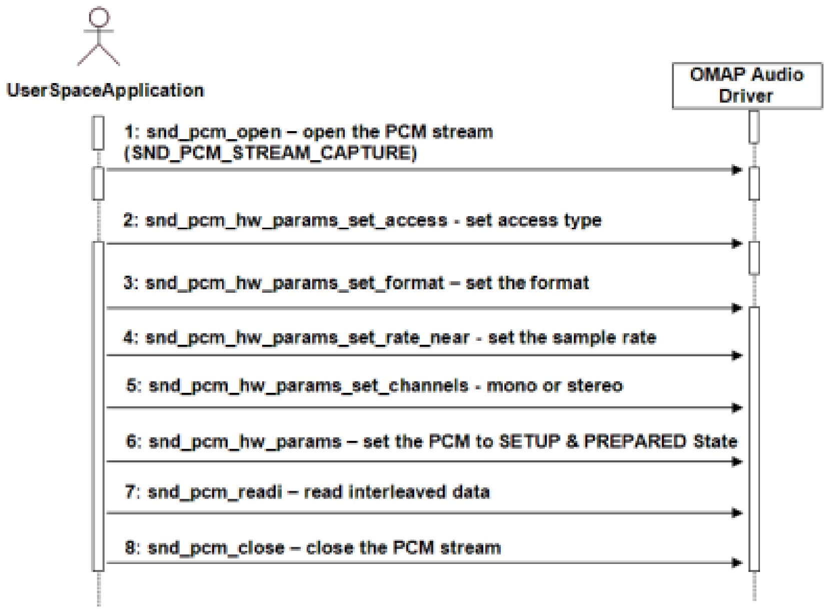

UserSpaceApplication
OMAP Audio Driver
1: snd_pcm_open – open the PCM stream (SND_PCM_STREAM_CAPTURE)
2: snd_pcm_hw_params_set_access - set access type
3: snd_pcm_hw_params_set_format – set the format
4: snd_pcm_hw_params_set_rate_near - set the sample rate
5: snd_pcm_hw_params_set_channels - mono or stereo
6: snd_pcm_hw_params – set the PCM to SETUP & PREPARED State
7: snd_pcm_readi – read interleaved data
8: snd_pcm_close – close the PCM stream

Figure 5.3. OMAP3 ALSA Driver : Half duplex record

5.6. Sample Applications

This chapter describes the sample application provided along with the package. The binary and the source for these sample application can are available in the Examples directory of the Release Package folder.

5.6.1. Introduction

Writing an audio application involves the following steps:

- Opening the audio device.
- Set the parameters of the device.
- Receive audio data from the device or deliver audio data to the device.
- Close the device.

These steps are explained in detail in this section.

Note

User space ALSA libraries can be downloaded from this link [http://www.alsa-project.org/main/index.php/Download].

User needs to build and install them before he starts using the ALSA based applications.

5.6.2. A minimal playback application

This program opens an audio interface for playback, configures it for stereo, 16 bit, 44.1kHz, interleaved conventional read/write access. Then its delivers a chunk of random data to it, and exits. It represents about the simplest possible use of the ALSA Audio API, and isn't meant to be a real program.

5.6.2.1. Opening the audio device

To write a simple PCM application for ALSA, we first need a handle for the PCM device. Then we have to specify the direction of the PCM stream, which can be either playback or capture. We also have to provide some information about the configuration we would like to use, like buffer size, sample rate, pcm data format. So, first we declare:

```
#include <stdio.h>
#include <stdlib.h>
#include <alsa/asoundlib.h>

#define BUFF_SIZE 4096
```

```
int main (int argc, char *argv[])
{
  int err;
  short buf[BUFF_SIZE];
  int rate = 44100; /* Sample rate */
  unsigned int exact_rate;   /* Sample rate returned by */

  /* Handle for the PCM device */
  snd_pcm_t *playback_handle;

  /* Playback stream */
  snd_pcm_stream_t stream = SND_PCM_STREAM_PLAYBACK;

  /* This structure contains information about    */
  /* the hardware and can be used to specify the  */
  /* configuration to be used for the PCM stream. */
  snd_pcm_hw_params_t *hw_params;
```

The most important ALSA interfaces to the PCM devices are the "plughw" and the "hw" interface. If you use the "plughw" interface, you need not care much about the sound hardware. If your sound card does not support the sample rate or sample format you specify, your data will be automatically converted. This also applies to the access type and the number of channels. With the "hw" interface, you have to check whether your hardware supports the configuration you would like to use. Otherwise, user can use the default interface for playback by:

```
  /* Name of the PCM device, like plughw:0,0          */
  /* The first number is the number of the soundcard, */
  /* the second number is the number of the device.   */

  static char *device = "default";  /* playback device */
```

Now we can open the PCM device:

```
  /* Open PCM. The last parameter of this function is the mode. */
  if ((err = snd_pcm_open (&playback_handle,
                      device, stream, 0)) < 0) {
    fprintf (stderr, "cannot open audio device (%s)\n",
    snd_strerror (err));
    exit (1);
  }
```

5.6.2.2. Setting the parameters of the device

Now we initialize the variables and allocate the `hwparams` structure:

```
  /* Allocate the snd_pcm_hw_params_t structure on the stack. */
  if ((err = snd_pcm_hw_params_malloc (&hw_params)) < 0) {
    fprintf (stderr, "cannot allocate hardware parameters (%s)\n",
    snd_strerror (err));
    exit (1);
  }
```

Before we can write PCM data to the soundcard, we have to specify access type, sample format, sample rate, number of channels, number of periods and period size. First, we initialize the hwparams structure with the full configuration space of the soundcard:

```
  /* Init hwparams with full configuration space */
  if ((err = snd_pcm_hw_params_any (playback_handle,
                                    hw_params)) < 0) {
    fprintf (stderr, "cannot initialize hardware
                        parameter structure (%s)\n",
    snd_strerror (err));
    exit (1);
  }
```

Now configure the desired parameters. For this example, we assume that the soundcard can be configured for stereo playback of 16 Bit Little Endian data, sampled at 44100 Hz. Therefore, we restrict the configuration space to match this configuration only.

The access type specifies the way in which multi-channel data is stored in the buffer. For INTERLEAVED access, each frame in the buffer contains the consecutive sample data for the channels. For 16 Bit stereo data, this means that the buffer contains alternating words of sample data for the left and right channel.

```
  /* Set access type. */
  if ((err = snd_pcm_hw_params_set_access (playback_handle,
                hw_params, SND_PCM_ACCESS_RW_INTERLEAVED)) < 0) {
    fprintf (stderr, "cannot set access type (%s)\n",
    snd_strerror (err));
    exit (1);
  }

  /* Set sample format */
  if ((err = snd_pcm_hw_params_set_format (playback_handle,
                hw_params, SND_PCM_FORMAT_S16_LE)) < 0) {
    fprintf (stderr, "cannot set sample format (%s)\n",
    snd_strerror (err));
    exit (1);
  }

  /* Set sample rate. If the exact rate is not supported */
  /* by the hardware, use nearest possible rate.         */
  exact_rate = rate;

  if ((err = snd_pcm_hw_params_set_rate_near (playback_handle,
                hw_params, &exact_rate, 0)) < 0) {
    fprintf (stderr, "cannot set sample rate (%s)\n",
    snd_strerror (err));
    exit (1);
  }

  if (rate != exact_rate) {
    fprintf(stderr, "The rate %d Hz is not supported by
                        your hardware.\n ==> Using %d
                        Hz instead.\n", rate, exact_rate);
  }

  /* Set number of channels */
  if ((err = snd_pcm_hw_params_set_channels (playback_handle,
                                   hw_params, 2)) < 0) {
    fprintf (stderr, "cannot set channel count (%s)\n",
    snd_strerror (err));
    exit (1);
  }
```

Now we apply the configuration to the PCM device pointed to by `pcm_handle` and prepare the PCM device.

```
  /* Apply HW parameter settings to PCM device and prepare
   * device.
   */
  if ((err = snd_pcm_hw_params (playback_handle,
                hw_params)) < 0) {
    fprintf (stderr, "cannot set parameters (%s)\n",
    snd_strerror (err));
    exit (1);
  }

  snd_pcm_hw_params_free (hw_params);

  if ((err = snd_pcm_prepare (playback_handle)) < 0) {
    fprintf (stderr, "cannot prepare audio
                interface for use (%s)\n", snd_strerror (err));
    exit (1);
  }
```

5.6.2.3. Writing data to the device

After the PCM device is configured, we can start writing PCM data to it. The first write access will start the PCM playback. For interleaved write access, we use the function:

```
  /* Write some junk data to produce sound. */
  if ((err =
        snd_pcm_writei (playback_handle, buf, BUFF_SIZE/2))
                   != BUFF_SIZE/2) {
    fprintf (stderr, "write to audio interface failed (%s)\n",
    snd_strerror (err));
    exit (1);
  } else {
    fprintf (stdout, "snd_pcm_writei successful\n");
  }
```

After the PCM playback is started, we have to make sure that our application sends enough data to the soundcard buffer. Otherwise, a buffer under-run will occur. After such an under-run has occurred, `snd_pcm_prepare` should be called.

5.6.2.4. Closing the device

After the data has been transferred, the device needs to be closed by calling:

```
  snd_pcm_close (playback_handle);
  exit (0);
}
```

5.6.3. A minimal record application

This program opens an audio interface for capture, configures it for stereo, 16 bit, 44.1kHz, interleaved conventional read/write access. Then its reads a chunk of random data from it, and exits. It isn't meant to be a real program.

Note that it is not possible to use one pcm handle for both playback and capture. So you have to configure two handles if you want to access the PCM device in both directions.

```
#include <stdio.h>
#include <stdlib.h>
#include <alsa/asoundlib.h>

#define BUFF_SIZE 4096

int main (int argc, char *argv[])
{
  int err;
  short buf[BUFF_SIZE];
  int rate = 44100; /* Sample rate */
  int exact_rate;   /* Sample rate returned by */

  snd_pcm_t *capture_handle;

  /* This structure contains information about    */
  /* the hardware and can be used to specify the  */
  /* configuration to be used for the PCM stream. */
  snd_pcm_hw_params_t *hw_params;

  /* Name of the PCM device, like hw:0,0 */
  /* The first number is the number of the soundcard, */
  /* the second number is the number of the device.   */
  static char *device = "default"; /* capture device  */
```

```
/* Open PCM. The last parameter of this function is
 * the mode.
 */
if ((err = snd_pcm_open (&capture_handle, device,
                SND_PCM_STREAM_CAPTURE, 0)) < 0) {
  fprintf (stderr, "cannot open audio device (%s)\n",
  snd_strerror (err));
  exit (1);
}

memset(buf,0,BUFF_SIZE);

/* Allocate the snd_pcm_hw_params_t structure on the stack. */
if ((err = snd_pcm_hw_params_malloc (&hw_params)) < 0) {
  fprintf (stderr, "cannot allocate hardware
                    parameter structure (%s)\n",
  snd_strerror (err));
  exit (1);
}

/* Init hwparams with full configuration space */
if ((err = snd_pcm_hw_params_any (capture_handle,
                                hw_params)) < 0) {
  fprintf (stderr, "cannot initialize hardware
                    parameter structure (%s)\n",
  snd_strerror (err));
  exit (1);
}

/* Set access type. */
if ((err = snd_pcm_hw_params_set_access (capture_handle,
                hw_params,
                SND_PCM_ACCESS_RW_INTERLEAVED)) < 0) {
  fprintf (stderr, "cannot set access type (%s)\n",
  snd_strerror (err));
  exit (1);
}

/* Set sample format */
if ((err = snd_pcm_hw_params_set_format (capture_handle,
                hw_params,
                SND_PCM_FORMAT_S16_LE)) < 0) {
  fprintf (stderr, "cannot set sample format (%s)\n",
  snd_strerror (err));
  exit (1);
}

/* Set sample rate. If the exact rate is not supported */
/* by the hardware, use nearest possible rate.   */
exact_rate = rate;

if ((err = snd_pcm_hw_params_set_rate_near (capture_handle,
                hw_params, &exact_rate, 0)) < 0) {
  fprintf (stderr, "cannot set sample rate (%s)\n",
  snd_strerror (err));
  exit (1);
}
```

```
  if (rate != exact_rate) {
    fprintf(stderr, "The rate %d Hz is not supported "
                    "by your hardware.\n ==> Using %d "
                    "Hz instead.\n", rate, exact_rate);
  }

  /* Set number of channels */
  if ((err = snd_pcm_hw_params_set_channels(capture_handle,
                  hw_params, 2)) < 0) {
    fprintf (stderr, "cannot set channel count (%s)\n",
    snd_strerror (err));
    exit (1);
  }

  /* Apply HW parameter settings to PCM device and
   * prepare device.
   */
  if ((err = snd_pcm_hw_params (capture_handle,
                  hw_params)) < 0) {
    fprintf (stderr, "cannot set parameters (%s)\n",
    snd_strerror (err));
    exit (1);
  }

  snd_pcm_hw_params_free (hw_params);

  if ((err = snd_pcm_prepare (capture_handle)) < 0) {
    fprintf (stderr, "cannot prepare audio interface for use
 (%s)\n",
    snd_strerror (err));
    exit (1);
  }

  /* Read data into the buffer. */
  if ((err = snd_pcm_readi (capture_handle, buf, 128)) != 128) {
    fprintf (stderr, "read from audio interface failed (%s)\n",
    snd_strerror (err));
    exit (1);
  } else {
    fprintf (stdout, "snd_pcm_readi successful\n");
  }

  snd_pcm_close (capture_handle);
  exit (0);
}
```

5.7. Revision History

0.95	Original version
0.97	Added proc and device related information and reorganized the content.
0.97p1	Added constraint that configuration of capture and playback streams in different sampling rates is not possible because of McBSP instance 2 limitation.
02.00.00	Moved to kernel version 2.6.26 and alsa core version 1.0.16.
02.01.00	Moved to ALSA SoC layer v1.0.18a and kernel version 2.6.29.
02.01.01	Moved to ALSA SoC layer v1.0.19.
03.00.00	Moved to ALSA SoC layer v1.0.20 and kernel version 2.6.31-rc7.

Display Driver

Abstract

This chapter provides detailed description of feature set and software interface for the display driver implementation.

Table of Contents

6.1. Introduction

Display Sub-System hardware integrates one graphics pipeline, two video pipelines, and two overlay managers (one for digital and one for analog interface). Digital interface is used for LCD and DVI output and analog interface is used for TV out.

The primary functionality of the display driver is to provide interfaces to user level applications and management of Display Sub-System hardware.

This section defines and describes the usage of user level interfaces of Video Display Driver.

Note

Please note that the AM3517 Display Sub-System module is same as OMAP35x, so terms have been used inter-changeably and referred as OMAP35x in this document.

6.1.1. References

1. Video for Linux Two Home Page [http://linux.bytesex.org/v4l2/]
2. Video for Linux Two API Specification [http://v4l2spec.bytesex.org/v4l2spec/v4l2.pdf]

6.1.2. Acronyms & Definitions

Acronym	Definition
V4L2	Video for Linux Two
DSS	Display SubSystem
NTSC	National Television System Committee
PAL	Phase Alternating Line
LCD	Liquid Crystal Display
DVI	Digital Visual Interface

Table 6.1. Video Display Driver: Acronyms

6.1.3. Hardware Overview

The display subsystem provides the logic to display a video frame from the memory frame buffer (either SDRAM or SRAM) on a liquid-crystal display (LCD) panel or a TV set. The display subsystem integrates the following elements

- Display controller (DISPC) module

- Remote frame buffer interface (RFBI) module
- Serial display interface (SDI) complex input/output (I/O) module with the associated phased-locked loop (PLL)
- Display serial interface (DSI) complex I/O module and a DSI protocol engine
- DSI PLL controller that drives a DSI PLL and high-speed (HS) divider
- NTSC/PAL video encoder

6.2. Features

6.2.1. Overview

The Display driver supports the following features:

- Supports LCD display interface at VGA resolution (480*640)
- Supports TV display interface at NTSC/PAL resolutions (both S-Video out and Composite out is supported)
- Supports DVI digital interface (mode selection via boot argument).
- Supports Graphics pipeline and two video pipelines. Graphics pipeline is supported through fbdev and video pipelines through V4L2.
- Supported color formats: On OSD (Graphics pipeline): RGB565, RGB888, ARGB and RGBA. On Video pipelines: YUV422 interleaved, RGB565, RGB888.
- Configuration of parameters such as height and width of display screen, bits-per-pixel etc.
- Supports setting up of OSD and Video pipeline destinations (TV or LCD) through syfs interface.
- Supports buffer management through memory mapped and user pointer buffer exchange for application usage (mmaped).
- Supports rotation - 0, 90, 180 and 270 degrees on LCD and TV output
- Supports destination and source colorkeying on Video pipelines through V4L2.
- Supports alpha blending through ARGB pixel format on Video2 pipeline and RGBA and ARGB format on graphics pipeline and global alpha blending

6.3. Architecture

This chapter describes the Driver Architecture and Design concepts

6.3.1. Driver Architecture

OMAP35x display hardware integrates one graphics pipeline, two video pipelines, and two overlay managers (one for digital and one for analog interface). Digital interface is used for LCD and DVI output and analog interface is used for TV out.

The primary functionality of the display driver is to provide interfaces to user level applications and management to OMAP35x display hardware. This includes, but is not limited to:

- GUI rendering through the graphics pipeline.
- Static image or video rendering through two video pipelines.
- Connecting each of three pipelines to either LCD or TV output so the display layer is presented on the selected output path.
- Image processing (cropping, rotation, mirroring, color conversion, resizing, and etc).

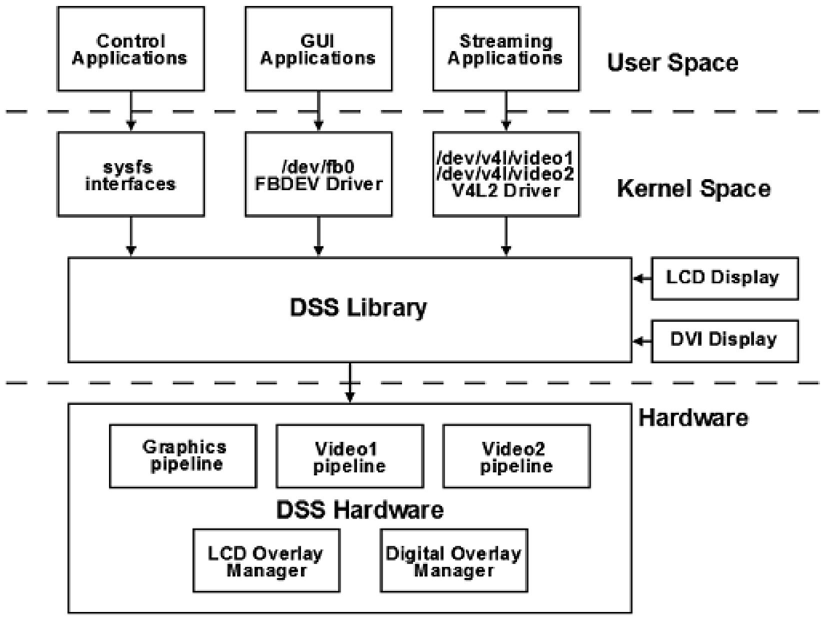


Figure 6.1. OMAP35x Display Subsystem Architecture

6.3.2. Software Design Interfaces

Above figure shows the major components that makes up the DSS software sub-system

Display Library

This is a HAL/functional layer controlling the bulk of DSS hardware. It exposes the number of APIs controlling the overlay managers, clock, and pipelines to the user interface drivers like V4L2 and FBDEV.

It also exposes the functions for registering and de-registering of the various display devices like LCD and DVI to the DSS overlay managers.

SYSFS interfaces

The SYSFS interfaces are mostly used as the control path for configuring the DSS parameters which are common between FBDEV and V4L2 like the alpha blending, color keying, etc.

It is also used for switching the output of the pipeline to either LCD or Digital overlay manager. In future sysfs entries might also be used to switch the modes like NTSC, PAL on TV and 480P, 720P on DVI outputs.

Note

Please note that due to clock source limitation while switching the output DSS2 throws error message "Could not find exact pixel clock" (In order to fix this we need to use DSI input clock source).

Frame Buffer Driver

This driver is registered with the FBDEV subsystem, and is responsible for managing the graphics layer frame buffer. Driver creates `/dev/fb0` as the device node. Application can open this device node to open the driver and negotiate parameters with the driver through frame buffer ioctls. Application maps driver allocated buffers in the application memory space and fills them for the driver to display.

Video Applications & V4L2 subsystem

Video applications (camera, camcorder, image viewer, etc.) use the standard V4L2 APIs to render static images and video to the video layers, or capture/preview camera images.

This driver is responsible for managing the video layers' frame buffers. It is a V4L2 compliant driver with some additions to implement special software requirements that target OMAP35x hardware features . This driver conforms to the Linux driver model. For using the driver, application should create the device nodes `/dev/video1`and `/dev/video2`device nodes for two video layers. Application can open the driver by opening these device nodes and negotiate the parameters by V4L2 ioctls. Initially application can request the driver to allocate number of buffers and MMAPs these buffers. Then the application can fill up these buffers and pass them to driver for display by using the standard V4L2 streaming ioctls.

6.4. Usage

6.4.1. Opening and Closing of Driver

The device can be opened using open call from the application, with the device name and mode of operation as parameters. Application can open the driver only in blocking mode. Non-blocking mode of open is not supported.

V4L2 Driver

The driver will expose two software channels (/dev/video1 and /dev/video2), one for each video pipeline. Both of these channels supports only blocking mode of operations. These channels can only be opened once.

```
/* Open a video Display logical channel in blocking mode */
fd = open ("/dev/video1", O_RDWR);
if (fd == -1) {
 perror("failed to open display device\n");
 return -1;
}
/* closing of channels */
close (fd);
```

FBDEV Driver

The driver will expose one software channels (/dev/fb0) for the graphics pipeline. The driver cannot be opened multiple times. Driver can be opened only once.

```
/* Open a graphics Display logical channel in blocking mode */
fd = open ("/dev/fb0", O_RDWR);
if (fd == -1) {
 perror("failed to open display device\n");
 return -1;
}
/* closing of channels */
close (fd);
```

6.4.2. Command Line arguments

V4L2 Driver

V4L2 driver supports set of command line argument for, default number of buffers, their buffer size, enable/disable VRFB buffer allocation and debug option for both the video pipelines.

V4L2 driver uses the VRFB buffers for rotation. Because of the limitation of the VRFB engine these buffers are quite big in size. Please refer to the Buffer Management section for required and allocated size of the VRFB buffers. VRFB buffers are allocated by driver during `vidioc_reqbufs` ioctl if the rotation is enabled and freed during `vidioc_streamoff`. But under heavy system load, memory fragmentation may occur and VFRB buffer allocation may fail. To address this issue V4L2 driver provides command line argument to allocate the VRFB buffers at driver init time and buffers will be freed when driver is unloaded.

Below is the list of arguments which V4L2 driver supports -

Argument	**Description**
video1_numbuffers	Number of buffers to be allocated at init time for Video1 device.
video2_numbuffers	Number of buffers to be allocated at init time for Video2 device.
video1_bufsize	Size of the buffer to be allocated for video1 device
video2_bufsize	Size of the buffer to be allocated for video2 device
vid1_static_vrfb_alloc	Static allocation of the VRFB buffer for video1 device
vid2_static_vrfb_alloc	Static allocation of the VRFB buffer for video2 device
debug	Enable debug messaging

Table 6.2. Acronyms

For dynamic build of the driver, these argument are specified at the time of inserting the driver. For static build of the driver, these argument can be specified along with boot time arguments. Following example shows how to specify command line argument for static and dynamic build.

Insert the dynamically built module with following parameters

```
# insmod omap_vout.ko video1_numbuffers=3 video2_numbuffers=3
 video1_bufsize=644000 video2_bufsize=644000
 vid1_static_vrfb_alloc=y vid2_static_vrfb_alloc=y
```

Set the `bootargs` for statically compiled driver from bootloader:

```
OMAP3_EVM # setenv bootargs console=ttyS0,115200n8
 mem=128M root=/dev/nfs noinitrd
 nfsroot=172.24.190.19:nfs-server/home,nolock ip=dhcp
 omap_vout.video1_numbuffers=3 omap_vout.video2_numbuffers=3
 omap_vout.video1_bufsize=64400 omap_vout.video2_bufsize=64400
 omap_vout.vid1_static_vrfb_alloc=y
 omap_vout.vid2_static_vrfb_alloc=y mpurate=600
```

Note

The entire command should be entered in a single line.

FBDEV Driver

FBDEV driver supports set of command line argument for enabling/ setting rotation angle, enable/disable VRFB rotation, default mode, size of vram and debug option. These command line arguments can only be used with boot time arguments as FBDEV driver only supports static build.

Below is the list of arguments which Fbdev driver supports -

Argument	Description
mode	Default video mode for specified displays
vram	VRAM allocated memory for a framebuffer, user can individually configure VRAM buffers for each plane/device node.
debug	Enable debug printing. You have to have OMAPFB debug support enabled in kernel config
vrfb	Use VRFB rotation for framebuffer
rotate	Default rotation applied to framebuffer

Table 6.3. Acronyms

Following example shows how to specify 90 degree rotation in boot time argument.

Set the `bootargs` for enabling rotation:

```
OMAP3_EVM # setenv bootargs console=ttyS0,115200n8 mem=128M
 noinitrd root=/dev/nfs nfsroot=172.24.190.19:nfs-server/
home,nolock ip=dhcp omapfb.rotate=1 omapfb.vrfb=y mpurate=600
```

Following example shows how to specify size of framebuffer in boot time argument.

Set the `bootargs` for specifying size of framebuffer:

```
OMAP3_EVM # setenv bootargs console=ttyS0,115200n8 mem=128M
 noinitrd root=/dev/nfs nfsroot=172.24.190.19:nfs-server/
home,nolock ip=dhcp vram=20M omapfb.vram=0:20M mpurate=600
```

Note

The entire command should be entered in a single line.

Misc (DSS) Argument

There are few arguments which allows control over core DSS functionality.

Argument	Description
def_disp	Name of default display, to which all overlays will be connected.
debug	Enable debug printing.

Table 6.4. Acronyms

Usage:

```
OMAP3_EVM # setenv bootargs console=ttyS0,115200n8 mem=128M
 noinitrd root=/dev/nfs nfsroot=172.24.190.19:nfs-server/
home,nolock ip=dhcp omapdss.def_disp="dvi" omapdss.debug=y
 mpurate=600
```

Note

The entire command should be entered in a single line.

6.4.3. Buffer Management

Driver	Without Rotation	With Rotation
FBDEV Driver	A single buffer of size 480*640*2 bytes, can be changed through bootargs	A single buffer of size 2048*640*2 bytes, can be changed through bootargs
V4L2 Driver	Single buffer takes 1280*720*4 bytes. Number of buffers is configurable using VIDIOC_REQBUFS ioctl and command line argument.	Same requirement as without rotation. Additionally allocates one buffer of size 3686400 bytes for each context. Number of context are same as the number of buffers allocated using REQBUFS, which is not more than four.

Table 6.5. Memory requirement for V4L2 and FBDEV driver Buffers

Note

Please note that user must configure the required amount of buffer size through command line argument in case of Framebuffer VRFB rotation.

V4L2 Driver

Memory Mapped buffer mode and User pointer buffer mode are the two memory allocation modes supported by driver.

In Memory map buffer mode, application can request memory from the driver by calling VIDIOC_REQBUFS ioctl. In this mode, maximum number of buffers is limited to VIDEO_MAX_FRAME (defined in driver header files) and is limited by the available memory in the kernel. If driver is not able to allocate the requested number of buffer, it will return the number of buffer it is able to allocate. The main steps that the application must perform for buffer allocation are:

1) Allocating Memory

This ioctl is used to allocate memory for frame buffers. This is a necessary ioctl for streaming IO. It has to be called for both drivers buffer mode and user buffer mode. Using this ioctl, driver will identify whether driver buffer mode or user buffer mode will be used.

Ioctl: VIDIOC_REQBUFS

It takes a pointer to instance of the `v4l2_requestbuffers` structure as an argument.

User can specify the buffer type (`V4L2_BUF_TYPE_VIDEO_OUTPUT`), number of buffers, and memory type (`V4L2_MEMORY_MMAP`, `V4L2_MEMORY_USERPTR`)at the time of buffer allocation. In case of driver buffer mode, this ioctl also returns the actual number of buffers allocated in count member of `v4l2_requestbuffer` structure

It can be called with zero number of buffers to free up all the buffers already allocated. It also frees allocated buffers when application changes buffer exchange mechanism. Driver always allocates buffers of maximum image size supported. If application wants to change buffer size, it can be done through video1_buffsize and video2_buffsize command line arguments

When rotation is enabled, driver also allocates buffer for the VRFB virtual memory space along with the mmap or user buffer. It allocates same number of buffers as the mmap or user buffers. Maximum number of buffers, which can be allocated, is 4 when rotation is enabled.

```
/* structure to store buffer request parameters */
struct v4l2_requestbuffers reqbuf;
reqbuf.count = numbuffers;
reqbuf.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
reqbuf.memory = V4L2_MEMORY_MMAP;
ret = ioctl(fd , VIDIOC_REQBUFS, &reqbuf);
if(ret < 0) {
    printf("cannot allocate memory\n");
    close(fd);
    return -1;
}
```

2) Getting physical address

This ioctl is used to query buffer information like buffer size and buffer physical address. This physical address is used in m-mapping the buffers. This ioctl is necessary for driver buffer mode as it provides the physical address of buffers, which are used to mmap system call the buffers.

Ioctl: VIDIOC_QUERYBUF

It takes a pointer to instance of `v4l2_buffer` structure as an argument. User has to specify the buffer type (V4L2_BUF_TYPE_VIDEO_OUTPUT), buffer index, and memory type (V4L2_MEMORY_MMAP)at the time of querying.

```
/* allocate buffer by VIDIOC_REQBUFS */

/* structure to query the physical address
of allocated buffer */
struct v4l2_buffer buffer;
/* buffer index for querying -0 */
buffer.index = 0;
buffer.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
buffer.memory = V4L2_MEMORY_MMAP;

if (ioctl(fd, VIDIOC_QUERYBUF, &buffer) < 0) {
    printf("buffer query error.\n");
    close(fd);
    exit(-1);
}
/*The buffer.m.offset will contain the physical
address returned from driver*/
```

3) Mapping Kernel space address to user space

Mapping the kernel buffer to the user space can be done via mmap. User can pass buffer size and physical address of buffer for getting the user space address

```
/* allocate buffer by VIDIOC_REQBUFS */
/* query the buffer using VIDIOC_QUERYBUF */
/* addr hold the user space address */
unsigned int addr;
Addr = mmap(NULL, buffer.size,PROT_READ | PROT_WRITE, MAP_SHARED,
   fd, buffer.m.offset);
/* buffer.m.offset is same as returned from VIDIOC_QUERYBUF */
```

FBDEV Driver

FBDEV driver supports only memory mapped buffers. Driver allocates one physically contiguous buffers, which can support 640X480 resolution for 16 bpp format. Following steps are required to map buffers in application memory space

1) Getting fix screen information

FBIOGET_FSCREENINFO ioctl is used to get the not-changing screen information like physical address of the buffer, size of the buffer, line length.

```
/* Getting fix screen information */
struct fb_fix_screeninfo fix;
ret = ioctl(fd, FBIOGET_FSCREENINFO, &fix);
if(ret < 0) {
    printf("Cannot get fix screen information\n");
    close(fd);
    exit(0);
}
printf("Line length = %d\n",fix.line_length);
printf("Physical Address = %x\n",fix.smem_start);
printf("Buffer Length = %d\n",fix.smem_len);
```

2) Getting Variable screen information

FBIOGET_VSCREENINFO ioctl is used to get the variable screen information like resolution, bits per pixel etc.

```
/* Getting fix screen information */
struct fb_var_screeninfo var;
ret = ioctl(fd, FBIOGET_VSCREENINFO, &var);
if(ret < 0) {
    printf("Cannot get variable screen information\n");
    close(fd);
    exit(0);
}
printf("Resolution = %dx%d\n",var.xred, var.yres);
printf("bites per pixel = %d\n",var.bpp);
```

3) Mapping Kernel space address to user space

Mapping the kernel buffer to the user space can be done via mmap system call.

```
/* addr hold the user space address */
unsigned int addr, buffersize;
```

```
/* Get the fix screen info */
/* Get the variable screen information */
buffersize = fix.line_length * var.yres;
addr = mmap(NULL, buffersize, PROT_READ | PROT_WRITE, MAP_SHARED,
   fd, 0);
/* buffer.m.offset is same as returned from VIDIOC_QUERYBUF */
```

6.4.4. Rotation

Rotation is implemented with use of Rotation Engine module in Virtual Rotation Frame Buffer module in OMAP35X. Rotation engine supports rotation of an image with degree 0, 90, 180 and 270. There are 12 contexts available for rotating an image and there are four virtual memory space associated with each context. To rotate an image, image is written to 0 degree virtual memory for a context and rotated image can read back from the virtual memory for that angle of the same context.

For using Rotation Engine, User has to allocate physical memory and provide address of the memory to the rotation engine. The buffer size for this physical buffer should be large enough to store the image to be rotated. When program writes to the virtual address of the context, rotation engine write to this memory space and when program reads image from virtual address, rotation engine reads image from this buffer with rotation angle.

V4L2 Driver

V4L2 driver supports rotation by using rotation engine in the VRFB module. Driver allocates physical buffers, required for the rotation engine, when application calls VIDIOC_REQBUFS ioctl. Therefore, when this ioctl is called driver allocates buffers for storing image and allocates buffers for the rotation engine. It also programs VRFB rotation engine when this ioctl is called. At the time of enqueing memory mapped buffer, driver copies entire image from mmaped buffer to buffer for the rotation engine using DMA. DSS is programmed to take image from VRFB memory space when rotation is enabled. So DSS always gets rotated image. Maximum four buffers can be allocated using REQBUFS ioctl when rotation is enabled.

Driver provides ioctl interface for enabling/disabling and changing the rotation angle. These ioctls are VIDIOC_S_CTRL/VIDIOC_G_CTRL as drive allocates buffer for VRFB during REQBUFS ioctl, application has to enable/set the rotation angle before calling REQBUFS ioctl. After enabling rotation, application can change the rotation angle. Rotation angle cannot be changed while streaming is on. Following code shows how to set rotation angle to 90 degree.

Important

Rotation value must be set using `VIDIOC_S_CTRL` before setting any format using `VIDIOC_S_FMT` as `VIDIOC_S_FMT` uses rotation value for calculating buffer formats. Also VIDIOC_S_FMT ioctl must be called after changing the rotation angle to change parameters as per the new rotation angle

```
struct v4l2_control control;
int degree = 90;

control.id = V4L2_CID_ROTATE;
control.value = degree;
ret = ioctl(fd, VIDIOC_S_CTRL, &control);
 if (ret < 0) {
  perror("VIDIOC_S_CTRL\n");
        close(fd);
        exit(0);
    }
/* Rotation angle is now set to 90 degree. Application can now do
 streaming to see rotated image*/
```

FBDEV Driver

FBDEV driver supports rotation by using rotation engine in the VRFB module. For using this feature of the driver, rotation has to be enabled. Application can enable rotation by enabling/setting rotation angle in boot time argument of the kernel for FBDEV driver. Applications can thus use the FBIOPUT_VSCREENINFO ioctl to set the rotation angle. Applications have to set the 'rotate' field in the fb_var_screeninfo structure equal to the angle of rotation (0, 90, 180 or 270) and call this ioctl. Frame buffer driver also supports the rotation through sysfs entry. Any one of the two method can be used to configure rotation.

Constraint: While doing rotation x-resolution virtual should be equal to x-resolution. y-resolution virtual should be greater than or equal to y-resolution. Please note that VRFB rotation engine requires alignment of 32 bytes in horizontal size and 32 lines in vertical size. So while doing rotation x-resolution should be 32 byte aligned and y resolution and y resolution virtual should be 32 lines aligned. For example for 360X360 required resolution with 16bpp no of bytes per line comes to 360*2=720. Which is not 32 byte aligned. While no of lines comes to 360 which is also not 32 lines aligned. So actual resolution should be set to 368X368. But if same resolution is required for 32bpp then no of bytes per line comes to 360*4 that is 1440. Which is 32 byte aligned so actual resolution should be set to 360X368. Also the maximum y-res virtual possible is 2048 because of VRFB limitation when rotation enabled.

var.rotate variable should not be modified when rotation is not selected through command line arguments else behaviour is unexpected.

Important

By default frame buffer driver allocates the buffer for single VGA (480x640) frame considering 0 degree rotation.

This allocation can be overridden using the command line arguments "vram=<size> and omapfb.vram=<fb>:<size>

Memory requirement can be calculated by following equation

(2048 * max(xres_virtual, yres_virtual) * max_Bpp)* NO_OF_BUFFERS, 2048 is the default pitch required by VRFB, xres_virtual = maximum virtual x-resolution required, yres_virtual = maximum virtual y-resolution required, max_Bpp = maximum bytes per pixel required, NO_OF_BUFFERS = Number of buffers required for panning.

So for 720*1280 resolution with 32bpp with two buffers with 90 or 270 degree rotation it comes to (2048 * 1280 * 4) * 2 = 20971520 bytes which rounds upto 20M bytes. so above command line arguments will look like

"vram=20M omapfb.vram=0:20M"

Please refer to the section Supported Command line Argument.

Following code listings demos how to set the rotation in frame buffer driver using ioctl and sysfs entry.

```
struct fb_var_screeninfo var;
/* Set the rotation through ioctl. */
/* Get the Variable screen info through "FBIOGET_VSCREENINFO" */
var.rotate = 1; /* To set rotation angle to 90 degree */
if (ioctl(fd, FBIOPUT_VSCREENINFO, &var)<0) {
 perror("Error:FBIOPUT_VSCREENINFO\n");
 close(fd);
 exit(4);
}
```

Setting the rotation through sysfs where 0 - 0 degree, 1 - 90 degree, 2 - 180 degree and 3 - 270 degree respectively

```
# echo 1 >  /sys/class/graphics/fb0/rotate
```

6.4.5. Color Keying

There are two types of transparent color keys: Video source transparency and graphics destination transparency key. The encoded pixel color value is compared to the transparency color key. For CLUT bitmaps, the palette index is compared to the transparency color key and not to the palette value pointed out by the palette index.

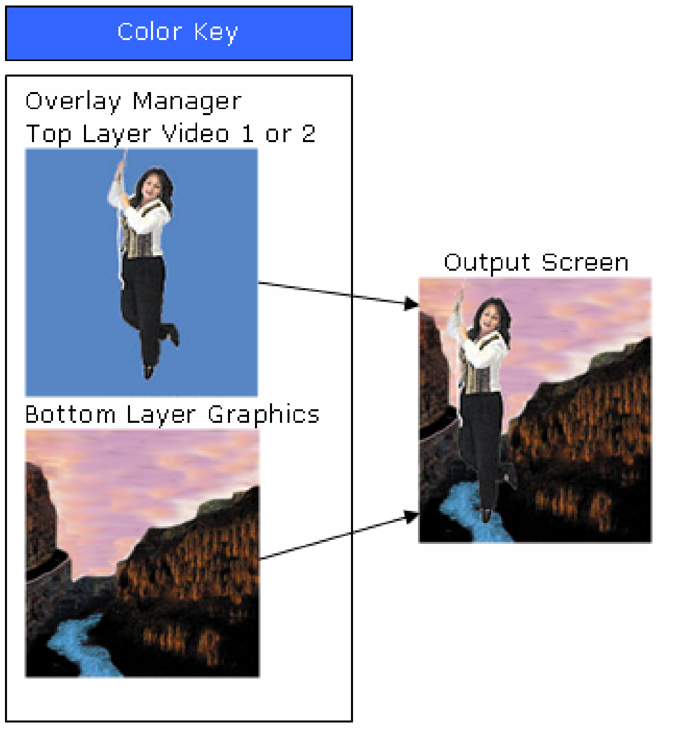


Figure 6.2. Video source color Keying

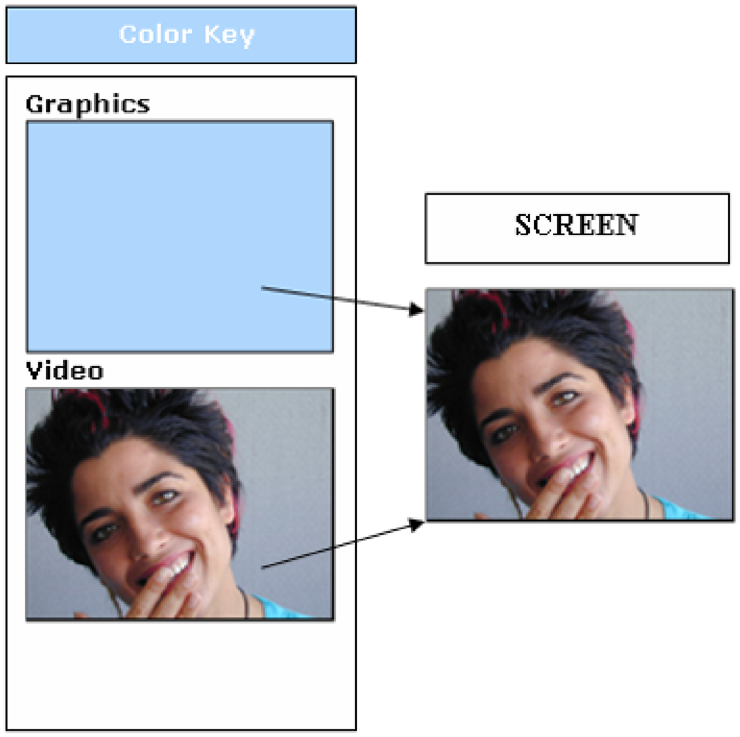


Figure 6.3. Video destination color Keying

Constraint: The video source transparency color key and graphics destination transparency color key cannot be active at the same time. Color keys are only available in V4L2 Driver.

Video source transparency color key value allows defining a color that the matching pixels with that color in the video pipelines are replaced by the pixels in graphics pipeline. It is limited to RGB formats only and non-scaling cases.

The Graphics destination color key allows defining a color that the non-matching pixels in the graphics pipelines prevent video overlay. The destination transparency color key is applicable only in the graphics region when graphics and video overlap. Otherwise, the destination transparency color key is ignored.

One of the colors keys can be activated at a time. This implies both key cannot be used simultaneously. All color key related IOCTLs are not pipeline oriented. An application can configure keys through either of two device nodes. Following example shows how to enable source color key.

```
struct v4l2_framebuffer framebuffer;

ret = ioctl (fd, VIDIOC_G_FBUF, &framebuffer);
```

```
if (ret < 0) {
 perror ("VIDIOC_G_FBUF");
 close(fd);
 exit(1);
}
/* Set SRC_COLOR_KEYING if device supports that */
if(framebuffer.capability & V4L2_FBUF_CAP_SRC_CHROMAKEY) {

 framebuffer.flags |= V4L2_FBUF_FLAG_SRC_CHROMAKEY;
 ret = ioctl (fd, VIDIOC_S_FBUF, &framebuffer);
 if (ret < 0) {
  perror ("VIDIOC_S_FBUF");
  close(fd);
  exit(1);
 }
}
```

The code snippet below illustrates how to disable source color keying

```
struct v4l2_framebuffer framebuffer;
ret = ioctl (fd, VIDIOC_G_FBUF, &framebuffer);
if (ret < 0) {
 perror ("VIDIOC_G_FBUF");
 close(fd);
 exit(1);
}
if(framebuffer.capability & V4L2_FBUF_CAP_SRC_CHROMAKEY) {

 framebuffer.flags &= ~V4L2_FBUF_FLAG_SRC_CHROMAKEY;

 ret = ioctl (fd, VIDIOC_S_FBUF, &framebuffer);
 if (ret < 0) {
  perror ("VIDIOC_S_FBUF");
  close(fd);
  exit(1);
 }
}
```

The code snippet below illustrates how to enable destination color keying

```
struct v4l2_framebuffer framebuffer;

ret = ioctl (fd, VIDIOC_G_FBUF, &framebuffer);
if (ret < 0) {
 perror ("VIDIOC_G_FBUF");
 close(fd);
 exit(1);
}
/* Set SRC_COLOR_KEYING if device supports that */
if(framebuffer.capability & V4L2_FBUF_CAP_CHROMAKEY) {

 framebuffer.flags |= V4L2_FBUF_FLAG_CHROMAKEY;
 ret = ioctl (fd, VIDIOC_S_FBUF, &framebuffer);
 if (ret < 0) {
  perror ("VIDIOC_S_FBUF");
  close(fd);
  exit(1);
 }
}
```

The code snippet below illustrates how to disable destination color keying

```
struct v4l2_framebuffer framebuffer;
ret = ioctl (fd, VIDIOC_G_FBUF, &framebuffer);
if (ret < 0) {
 perror ("VIDIOC_G_FBUF");
 close(fd);
 exit(1);
}
if(framebuffer.capability & V4L2_FBUF_CAP_CHROMAKEY) {

 framebuffer.flags &= ~V4L2_FBUF_FLAG_CHROMAKEY;

 ret = ioctl (fd, VIDIOC_S_FBUF, &framebuffer);
 if (ret < 0) {
  perror ("VIDIOC_S_FBUF");
  close(fd);
  exit(1);
 }
}
```

Below program listing shows how to set the chromakey value. Please note that chroma key value should be set before enabling the chroma keying. Overlay manager should not be changed between the setting up of chroma key and enabling the chroma keying.

```
struct v4l2_format fmt;
u8 chromakey = 0xF800; /* Red color RGB565 format */
fmt.type = V4L2_BUF_TYPE_VIDEO_OVERLAY;

ret = ioctl(fd, VIDIOC_G_FMT, &fmt);
if (ret < 0) {
 perror("VIDIOC_G_FMT\n");
 close(fd);
 exit(0);
}
fmt.fmt.win.chromakey = chromakey;

ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if (ret < 0) {
 perror("VIDIOC_G_FMT\n");
 close(fd);
 exit(0);
}
```

The code snippet below illustrates how to get the chromakey value.

```
struct v4l2_format fmt;
fmt.type = V4L2_BUF_TYPE_VIDEO_OVERLAY;
```

```
ret = ioctl(fd, VIDIOC_G_FMT, &fmt);
if (ret < 0) {
 perror("VIDIOC_G_FMT\n");
 close(fd);
 exit(0);
}
printf("Global alpha value read is %d\n", fmt.fmt.win.chromakey);
```

6.4.6. Alpha Blending

Alpha blending is a process of blending a foreground color with a background color and producing a new blended color. New blended color depends on the transparency factor referred to as alpha factor of the foreground color. If the alpha factor is 100% then blended image will have only foreground color. If the alpha factor is 0% blended image will have only back ground color. Any value between 0 to 100% will blend the foreground and background color to produce new blended color depending upon the alpha factor.

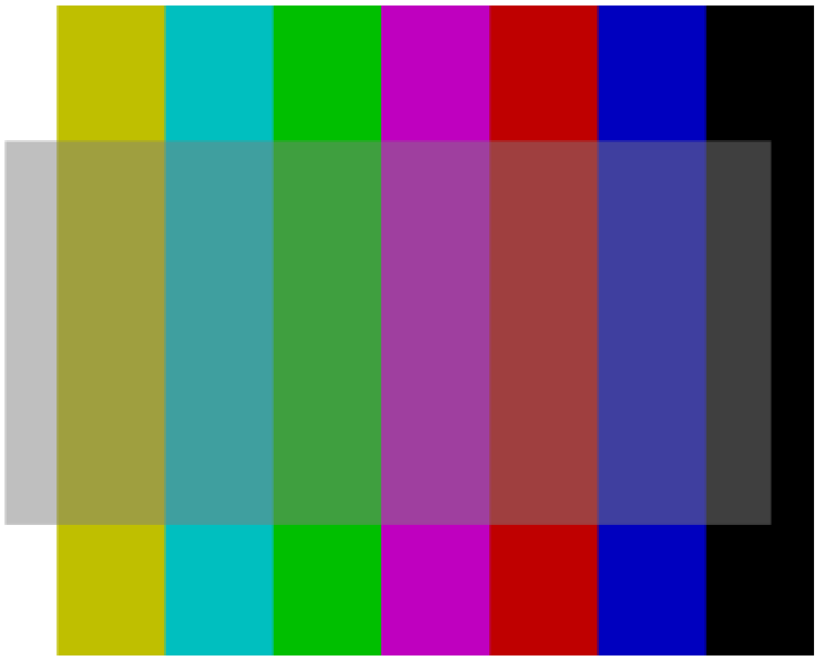


Figure 6.4. Alpha blending with almost 50% transparency

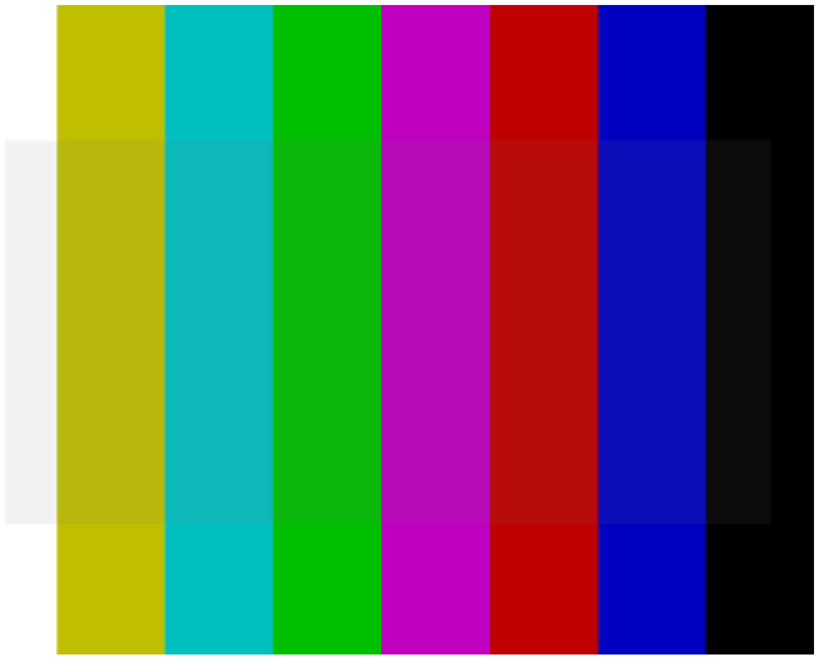


Figure 6.5. Alpha blending with almost 100% transparency

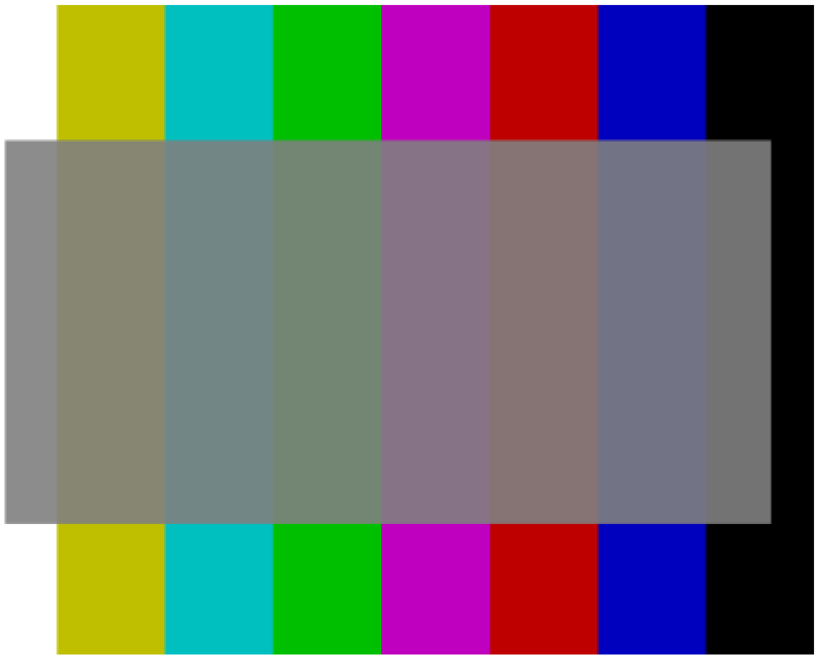


Figure 6.6. Alpha blending with almost 0% transparency

Overlay manager of DSS is capable of supporting the alpha blending. This is done by displaying more than one layer (video and graphics) to the same output device, TV or LCD. Overlay manager supports normal mode and alpha mode of operation. In normal mode graphics plane is at bottom on top of it is video1 and video2 is on top of video1. While in alpha mode video1 plane is at bottom, video2 is on top of video1, and graphics plane is above video2. Alpha mode is selectable on any of the output device TV or LCD.

Video2 and graphics layer of the DSS is capable of supporting alpha blending. Two types of alpha blending is supported global and pixel alpha blending. ARGB and RGBA formats of the video2 and graphics pipeline supports pixel based alpha blending. In which A represent the alpha value for each pixel. Thus, each pixel can have different alpha value. While global alpha is the constant alpha factor for the pipeline for all the pixels. Both can be used in conjunction.

Both V4L2 and Frame buffer driver supports alpha blending based on pixel format for video2 and graphics pipeline respectively. Global alpha blending is also supported through V4L2 and Fbdev ioctls. Before using any of the alpha blending methods alpha blending needs to be enabled on the selected output device through V4L2 ioctl. Alpha blending will be enabled on the output device to which video pipeline is connected

Following program listing will enable alpha blending through V4L2 driver ioctl -

```
struct v4l2_framebuffer framebuffer;

ret = ioctl (fd, VIDIOC_G_FBUF, &framebuffer);
if (ret < 0) {
 perror ("VIDIOC_S_FBUF");
 close(fd);
 return 0;
}

framebuffer.flags |= V4L2_FBUF_FLAG_LOCAL_ALPHA;
ret = ioctl (fd, VIDIOC_S_FBUF, &framebuffer);
if (ret < 0) {
 perror ("VIDIOC_S_FBUF");
 close(fd);
 return 0;
}
```

Following program listing will disable alpha blending through V4L2 driver ioctl -

```
struct v4l2_framebuffer framebuffer;

ret = ioctl (fd, VIDIOC_G_FBUF, &framebuffer);
if (ret < 0) {
 perror ("VIDIOC_S_FBUF");
 close(fd);
 return 0;
}

framebuffer.flags &= ~V4L2_FBUF_FLAG_LOCAL_ALPHA;
```

```
ret = ioctl (fd, VIDIOC_S_FBUF, &framebuffer);
if (ret < 0) {
 perror ("VIDIOC_S_FBUF");
 close(fd);
 return 0;
}
```

Following program listing will enable/disable alpha blending through SYSFS entry -

```
# echo 0/1 > /sys/devices/platform/omapdss/manager<index>/
alpha_blending_enabled

where,
 0/1     => 0 - Disable, 1 - Enable.
 index   => 0 - LCD Manager
        1 - TV Manager.
```

V4L2 Driver

V4l2 driver supports alpha blending through ARGB pixel format as well as global alpha value.

To set the pixel alpha value set ARGB format by setting format type to V4L2_PIX_FMT_RGB32. Call VIDIOC_S_FMTioctl of the driver to set it to ARGB format. Note: RGBA format is not supported.

```
struct v4l2_format fmt;
/* Set the video type*/
fmt.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
/* Set the width and height of the picture*/
fmt.fmt.pix.width = 400;
fmt.fmt.pix.height = 400;
/* Set the format to ARGB */
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_RGB32;
/* Call set format Ioctl */
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if (ret < 0) {
    perror("VIDIOC_S_FMT\n");
    close(fd);
    exit(0);
}
```

Setting the global alpha value is supported through V4L2_BUF_TYPE_VIDEO_OVERLAY format type. Below programlisting shows how to set the global alpha value for video2 pipeline.

```
struct v4l2_format fmt;
u8 global_alpha = 128;
fmt.type = V4L2_BUF_TYPE_VIDEO_OVERLAY;

ret = ioctl(fd, VIDIOC_G_FMT, &fmt);
```

```
if (ret < 0) {
 perror("VIDIOC_G_FMT\n");
 close(fd);
 exit(0);
}
fmt.fmt.win.global_alpha = global_alpha;

ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if (ret < 0) {
 perror("VIDIOC_G_FMT\n");
 close(fd);
 exit(0);
}
```

FBDEV Driver

Frame buffer driver supports setting of pixel alpha value as well as global alpha value

Pixel alpha value is supported through 32 bpp. Setting the offsets correctly will set the pixel format as ARGB or RGBA. Below program listing shows how to set ARGB pixel format.

```
fb_var_screeninfo var;
/* Get variable screen information. Variable screen information
 * gives information like size of the image, bites per pixel,
 * virtual size of the image etc. */
ret = ioctl(fd, FBIOGET_VSCREENINFO, &var);
if (ret < 0) {
    perror("Error reading variable information.\n");
    close(fd);
    exit(3);
}
/* Set bits per pixel and offsets*/
var.red.length= 8;
var.green.length = 8;
var.blue.length = 8;
var.transp.length= 8;
var.transp.offset = 24;
var.red.offset = 16;
var.green.offset =8;
var.blue.offset = 0;
var.bits_per_pixel = 32;
if (ioctl(fd, FBIOPUT_VSCREENINFO, &var)<0) {
    perror("Error:FBIOPUT_VSCREENINFO\n");
    close(fd);
    exit(4);
}
```

User can set the global alpha value for graphics pipeline using sysfs entry, as shown below -

```
# echo <global alpha value> > /sys/devices/platform/omapdss/
overlay0/global_alpha
```

Note

Before using the global alpha or pixel based alpha on graphics pipeline. Alpha blending needs to be enabled using either sysfs entry described below or V4L2 ioctl described under V4L2 driver in this section.

```
# echo 1 > /sys/devices/platform/omapdss/manager0/
alpha_blending_enabled
```

6.4.7. Buffer Format

Buffer format describes the pixel format in the image. It also describes the memory organization of each color component within the pixel format. In all buffer formats, blue value is always stored in least significant bits, then green value and then red value.

V4L2 Driver

Video layer supports following buffer format: YUYV, UYVY, RGB565, RGB24 (packed and unpacked). The corresponding v4l2 defines for pixel format are `V4L2_PIX_FMT_YUYV`, `V4L2_PIX_FMT_UYVY`, `V4L2_PIX_FMT_RGB565`, `V4L2_PIX_FMT_RGB24` `(packed)`, `V4L2_PIX_FMT_RGB32`. (For video1 and video2 V4L2_PIX_FMT_RGB32 corresponds to RGB24 unpacked).

Buffer format can be changed using VIDIOC_S_FMT ioctl with type as V4L2_BUF_TYPE_VIDEO_OUTPUT and appropriate pixel format type. Following example shows how to change pixel format to RGB565

```
struct v4l2_format fmt;
fmt.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_RGB565;
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if (ret < 0) {
    perror("VIDIOC_S_FMT\n");
    close(fd);
    exit(0);
}
```

FBDEV Driver

Graphics layer supports following buffer format: RGB24(un-packed) ARGB, RGBA and RGB565. Buffer format can be changed in FBDEV driver by using bpp, red, green, and blue fields of fb_vscreeninfo structure and ioctl FBIOPUT_VSCREENINFO. Application needs to specify bits per pixel and length and offset of red, green and blue component. Bits-per-pixel and color depth in the pixel aren't quite the same thing. The display controller supports color depths of 1, 2, 4, 8, 12, 16, 24 and 32 bits. Color depth and bits-per-pixel are the same for depths of 1, 2, 4, 8, and 16 bits, but for a color depth of 12 bits the pixel data is padded to 16 bits-per-pixel, and for a color depth of 24 bits the pixel data is padded to 32 bits-per-pixel. So application has to specify bits per pixel 16 and 32 for the color depth 12 and 24. To specify exact color depth, red, green and blue member of the fb_varscreeninfo can be used. Following example shows how to set 12 and 24 bits per pixels.

```
Struct fb_varscreeninfo var;
var.bpp = 16;
var.red.length = var.green.length = var.blue.length = 4;
var.red.offset = 8;
var.green.offset = 4;
var.blue.offset = 0;
ret = ioctl(fd, FBIOPUT_VSCREENINFO, &var);
if (ret < 0) {
 perror("FBIOPUT_VSCREENINFO\n");
 close(fd);
 exit(0);
}
```

Buffer Formats

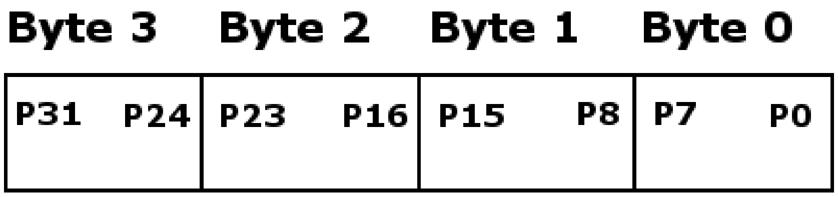


Figure 6.7. 1-BPP Data Memory Organization

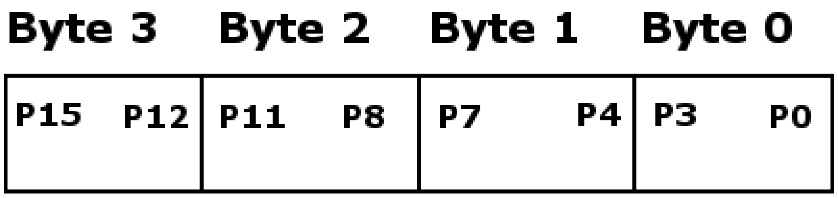


Figure 6.8. 2-BPP Data Memory Organization

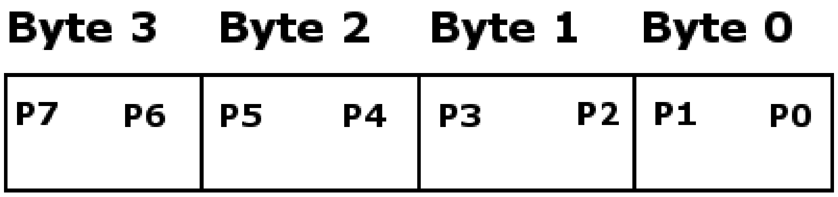


Figure 6.9. 4-BPP Data Memory Organization

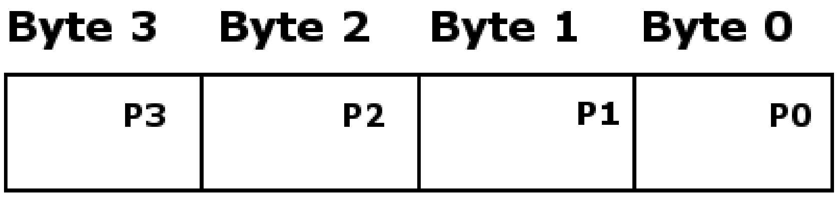


Figure 6.10. 8-BPP Data Memory Organization

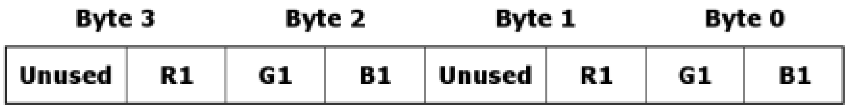


Figure 6.11. 12-BPP Data Memory Organization

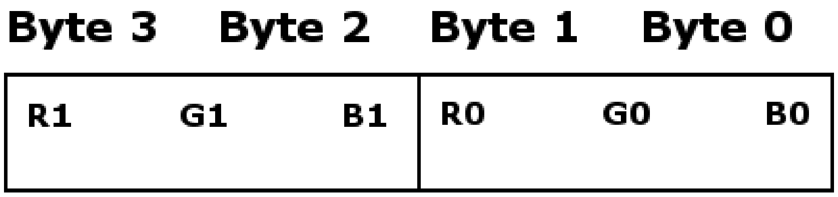


Figure 6.12. 16-BPP Data Memory Organization

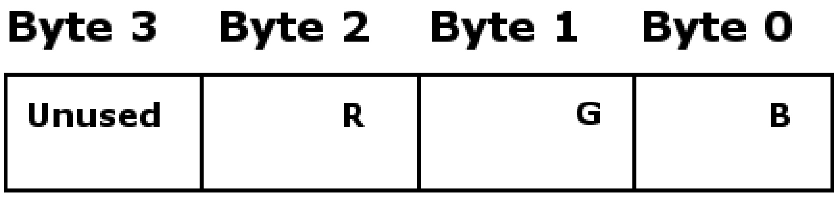


Figure 6.13. 24-BPP Data Memory Organization

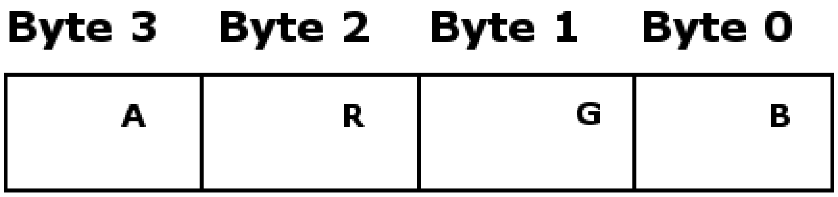


Figure 6.14. ARGB 32-BPP Data Memory Organization

Byte 3	Byte 2	Byte 1	Byte 0
R	G	B	A

Figure 6.15. RGBA 32-BPP Data Memory Organization

	Byte 3	Byte 2	Byte 1	Byte 0
W0	B1	R0	G0	B0
W1	G2	B2	R1	G1
W2	R3	G3	B3	R2

Figure 6.16. 24-BPP Packed Data Memory Organization

Byte 3	Byte 2	Byte 1	Byte 0
Cr0	Y1	Cb0	Y0

Figure 6.17. UYVY 4:2:2 Data Memory Organization

Byte 3	Byte 2	Byte 1	Byte 0
Y1	Cr0	Y0	Cb0

Figure 6.18. YUV2 4:2:2 Data Memory Organization

6.4.8. Display Window

The video pipelines can be connected to either an DVI output LCD output or a TV output either through boot time parameter or through SYSFS interface. Although the display Driver computes a default display window whenever the image size or cropping is changed, an application should position the display window via the VIDIOC_S_FMT I/O control with the V4L2_BUF_TYPE_VIDEO_OVERLAY buffer type. When a switch from LCD to TV or from TV to LCD happens, an application is expected to adjust the display window. V4L2 driver only supports change of display window.

Following example shows how to change display window size.

```
struct v4l2_format fmt;
Fmt.type = V4L2_BUF_TYPE_VIDEO_OVERLAY;
fmt.fmt.win.w.left = 0;
fmt.fmt.win.w.top = 0;
fmt.fmt.win.w.width = 200;
fmt.fmt.win.w.height = 200;
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if (ret < 0) {
 perror("VIDIOC_S_FMT\n");
 close(fd);
 exit(0);
}
/* Display window size and position is changed now */
```

6.4.9. Cropping

The V4L2 Driver allows an application to define a rectangular portion of the image to be rendered via the `VIDIOC_S_CROP` Ioctl with the `V4L2_BUF_TYPE_VIDEO_OUTPUT` buffer type. When application calls `VIDIOC_S_FMT` ioctl, driver sets default cropping rectangle that is the largest rectangle no larger than the image size and display windows size. The default cropping rectangle is centered in the image. All cropping dimensions are rounded down to even numbers. Changing the size of the cropping rectangle will in general also result in a new default display window. As stated above, an application must adjust the display window accordingly.

Following example shows how to change crop size.

```
struct v4l2_crop crop;
crop.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
crop.c.left = 0;
crop.c.top = 0;
crop.c.width = 320;
crop.c.height = 320;
ret = ioctl(fd, VIDIOC_S_CROP, &crop);
if (ret < 0) {
    perror("VIDIOC_S_CROP\n");
    close(fd);
    exit(0);
}
/* Image cropping rectangle is now changed */
```

6.4.10. Scaling

Video pipe line contains scaling unit which is used when transferring pixels from the system memory to the LCD panel or the TV set. The scaling unit consists of two scaling blocks: The vertical scaling block followed by the horizontal scaling block. The two scaling units are independent: Neither of them, only one, or both can be used simultaneously.

As scaling unit is on video pipeline, scaling is only supported in V4L2 driver. Scaling is not explicitly exposed at the API level. Instead, the horizontal and vertical scaling factors are based on the display window and the image cropping rectangle. The horizontal scaling factor is computed by dividing the width of the display window by the width of the cropping rectangle. Similarly, the vertical scaling factor is computed by dividing the height of the display window by the height of the cropping rectangle.

Down-scaling is limited upto factor 0.5 and the up-scaling factor to 8 in the software, while hardware supports from 0.25x to 8x both horizontally and vertically . The display Driver makes sure the limits are never exceeded.

The code snippet below illustrates how to scale image by factor of 2.

```
struct v4l2_format fmt;
struct v4l2_crop crop;
/* Changing display window size to 200x200 */
fmt.type = V4L2_BUF_TYPE_VIDEO_OVERLAY;
fmt.fmt.win.w.left = 0;
fmt.fmt.win.w.top = 0;
fmt.fmt.win.w.width = 200;
fmt.fmt.win.w.height = 200;
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if (ret < 0) {
    perror("VIDIOC_S_FMT\n");
    close(fd);
    exit(0);
}
/* Changing crop window size to 400x400 */
crop.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
crop.c.left = 0;
crop.c.top = 0;
crop.c.width = 400;
crop.c.height = 400;
ret = ioctl(fd, VIDIOC_S_CROP, &crop);
if (ret < 0) {
    perror("VIDIOC_S_CROP\n");
    close(fd);
    exit(0);
}
/* Image should be now scaled by factor 2 */
```

6.4.11. Color look table

The graphics pipeline supports the color look up table. The CLUT mode uses the encoded pixel values from the input image as pointers to index the 24-bit-wide CLUT value: 1-BPP pixels address 2 entries, 2-BPP pixels address 4 entries, 4-BPP pixels address 16 entries, and 8-BPP pixels address 256 entries.

Driver supports 1, 2, 4 and 8 bits per pixel image format using color lookup table. FBIOPUTCMAP and FBIOGETCMAP can be used to set and get the color map table. When CLUT is set, the driver makes the hardware to reload the CLUT.

Following example shows how to change CLUT.

```
struct fb_cmap cmap;
unsigned short r[4]={0xFF,0x00, 0x00, 0xFF};
unsigned short g[4]={0x00, 0xFF, 0x00, 0xFF};
unsigned short b[4]={0x00, 0x00, 0xFF, 0x00};
cmap.len = 4;
cmap.red = r;
cmap.green = g;
cmap.blue = b;
if (ioctl(fd, FBIOPUTCMAP, &cmap)) {
    perror("FBIOPUTCMAP\n");
    exit(3);
}
```

6.4.12. Streaming

V4L2 driver supports the streaming of the buffer. To do streaming minimum of three buffers should be requested by the application by using VIDIOC_REQBUFS ioctl. Once driver allocates the requested buffers application should call VIDIOC_QUERYBUF and mmap to get the physical address of the buffers and map the kernel memory to user space as explained earlier. Following are the steps to enable streaming.

1. Fill the buffers with the image to be displayed in the proper format.

2. Queue buffers to the driver queue using VIDIOC_QBUF ioctl.

3. Start streaming using VIDIOC_STREAMON ioctl.

4. Call VIDIOC_DQBUF to get the displayed buffer.

5. Repeat steps 1, 2, 4 and 5 in a loop for the frame count to be displayed.

6. Call VIDIOC_STREAMOFF ioctl to stop streaming.

Following example shows how to do streaming with V4l2 driver.

```
/* Initially fill the buffer */
```

```
 struct v4l2_requestbuffers req;
 struct v4l2_buffer buf;
 struct v4l2_format fmt;
/* Fill the buffers with the image */

 /* Enqueue buffers */
for (i = 0; i < req.count; i++) {
    buf.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
    buf.index = i;
    buf.memory = V4L2_MEMORY_MMAP;
    ret = ioctl(fd, VIDIOC_QBUF, &buf);
    if (ret < 0) {
        perror("VIDIOC_QBUF\n");
        for (j = 0; j < req.count; j++){
   /* Unmap all the buffers if call fails */
   exit(0);
        }
    printf("VIDIOC_QBUF = %d\n",i);
    }
}
/* Start streaming */
a = 0;
ret = ioctl(fd, VIDIOC_STREAMON, &a);
if (ret < 0) {
    perror("VIDIOC_STREAMON\n");
    for (i = 0; i < req.count; i++)
        /* Unmap all the buffers if call fails */
  exit(0);
    }
/* loop for streaming with 500 Frames*/
for(i = 0 ;i < LOOPCOUNT ;i ++) {
    ret = ioctl(fd, VIDIOC_DQBUF, &buf);
    if(ret < 0){
        perror("VIDIOC_DQBUF\n");
        for (j = 0; j < req.count; j++){
            /* Unmap all the buffers if call fails */
   exit(0);
        }
 }
    /* Fill the buffer with new data
    fill(buff_info[buf.index].start, fmt.fmt.pix.width,
 fmt.fmt.pix.height,0);
 /Queue the buffer again */
    ret = ioctl(fd, VIDIOC_QBUF, &buf);
    if(ret < 0){
        perror("VIDIOC_QBUF\n");
        for (j = 0; j < req.count; j++){
           /* Unmap all the buffers if call fails */
   exit(0);
        }
    }
}

/* Streaming off */
    ret = ioctl(fd, VIDIOC_STREAMOFF, &a);
    if (ret < 0) {
        perror("VIDIOC_STREAMOFF\n");
```

```
                for (i = 0; i < req.count; i++){
                        /* Unmap all the buffers if call fails */
         exit(0);
        }
          }
```

6.5. Software Interfaces

6.5.1. Frame-Buffer Driver Interface

6.5.1.1. Application Interface

`open ()`

To open a framebuffer device

`close ()`

To close a framebuffer device

`ioctl ()`

To send ioctl commands to the framebuffer driver.

`mmap ()`

To obtain the framebuffer region as mmap'ed area in user space.

6.5.1.2. Supported Standard IOCTLs

`FBIOGET_VSCREENINFO`, `FBIOPUT_VSCREENINFO`

These I/O controls are used to query and set the so-called variable screen info. This allows an application to query or change the display mode, including the color depth, resolution, timing etc. These I/O controls accept a pointer to a struct fb_var_screeninfo structure. The video mode data supplied in the fb_var_screeninfo struct is translated to values loaded into the display controller registers.

`FBIOGET_FSCREENINFO`

This I/O control can be used by applications to get the fixed properties of the display, e.g. the start address of the framebuffer memory. This I/O control accepts a pointer to a struct fb_fix_screeninfo

`FBIOGETCMAP`, `FBIOPUTCMAP`

These I/O controls are used to get and set the color-map for the framebuffer. These I/O controls accept a pointer to a struct fb_cmap structure.

`FBIO_BLANK`

This I/O control is used to blank or unblank the framebuffer console.

6.5.1.3. Supported Custom IOCTLs

OMAPFB_WAITFORVSYNC

This ioctl can be used to put an application to sleep until next vertical sync interval of the display.

OMAPFB_GET_VRAM_INFO

Ioctl returns the configured/allocated vram information.

OMAPFB_QUERY_MEM

Returns the size and type of the frame buffer.

```
Data Structure:

#define OMAPFB_MEMTYPE_SDRAM            0
#define OMAPFB_MEMTYPE_SRAM             1
#define OMAPFB_MEMTYPE_MAX              1

struct omapfb_mem_info {
 __u32 size;
 __u8  type;
 __u8  reserved[3];
};

Usage:

struct omapfb_mem_info mi;
if (ioctl(fb, OMAPFB_QUERY_MEM, &mi)) {
 perror("Error: OMAPFB_QUERY_MEM.\n");
 exit(1);
}
printf("size - %d\n", mi.size);
printf("type - %d\n", mi.type);
```

OMAPFB_SETUP_MEM

Allows user to setup the frame buffer memory, like size and type.

```
Data Structure:

#define OMAPFB_MEMTYPE_SDRAM            0
#define OMAPFB_MEMTYPE_SRAM             1
#define OMAPFB_MEMTYPE_MAX              1

struct omapfb_mem_info {
 __u32 size;
 __u8  type;
 __u8  reserved[3];
};

Usage:

struct omapfb_mem_info mi;
mi.size = <Expected size of buffer>
mi.type = <Expected type of buffer>
if (ioctl(fb, OMAPFB_SETUP_MEM, &mi)) {
 perror("Error: OMAPFB_SETUP_MEM.\n");
 exit(1);
}
```

OMAPFB_QUERY_PLANE

Query the plane (gfx) and returns the omapfb_plane_info information -

```
Data Structure:

struct omapfb_plane_info {
 __u32 pos_x;
 __u32 pos_y;
 __u8  enabled;
 __u8  channel_out;
 __u8  mirror;
 __u8  reserved1;
 __u32 out_width;
 __u32 out_height;
 __u32 reserved2[12];
};

Usage:

struct omapfb_plane_info pi;
if (ioctl(fb, OMAPFB_QUERY_PLANE, &pi)) {
 perror("Error: OMAPFB_QUERY_PLANE.\n");
 exit(1);
}
```

OMAPFB_SETUP_PLANE

TBD.

6.5.1.4. Data Structures

fb_var_screeninfo

This structure is used to query and set the so-called variable screen information. This allows an application to query or change the display mode, including the color depth, resolution, timing etc.

`fb_fix_screeninfo`

This structure is used by applications to get the fixed properties of the display, e.g. the start address of the framebuffer memory, framebuffer length etc.

`fb_cmap`

This structure is used to get/set the color-map for the framebuffer

6.5.2. V4L2 Driver Interface

6.5.2.1. Application Interface

`open`

To open a video device

`close`

To close a video device

`ioctl`

To send ioctl commands to the display driver.

`mmap`

To memory map a driver allocated buffer to user space

6.5.2.2. Supported Standard IOCTLs

Note

This section describes the standard V4L2 IOCTLs supported by the Display Driver. Standard IOCTLs that are not listed here are not supported. The Display Driver handles the unsupported ones by returning `EINVAL`error code.

`VIDIOC_QUERYCAP`

This is used to query the driver's capability. The video driver fills a v4l2_capability struct indicating the driver is capable of output and streaming.

`VIDIOC_ENUM_FMT`

This is used to enumerate the image formats that are supported by the driver. The driver fills a v4l2_fmtdesc struct.

VIDIOC_G_FMT

This is used to get the current image format or display window depending on the buffer type. The driver fills the information to a v4l2_format struct.

VIDIOC_TRY_FMT

This is used to validate a new image format or a new display window depending on the buffer type. The driver may change the passed values if they are not supported. Application should check what is granted.

VIDIOC_S_FMT

This is used to set a new image format or a new display window depending on the buffer type. The driver may change the passed values if they are not supported. Application should check what is granted if VIDIOC_TRY_FMT is not used first.

VIDIOC_CROPCAP

This is used to get the default cropping rectangle based on the current image size and the current display panel size. The driver fills a v4l2_cropcap struct.

VIDIOC_G_CROP

This is used to get the current cropping rectangle. The driver fills a v4l2_crop struct.

VIDIOC_S_CROP

This is used to set a new cropping rectangle. The driver fills a v4l2_crop struct. Application should check what is granted.

VIDIOC_REQBUFS

This is used to request a number of buffers that can later be memory mapped. The driver fills a v4l2_requestbuffers struct. Application should check how many buffers are granted.

VIDIOC_QUERYBUF

This is used to get a buffer's information so mmap can be called for that buffer. The driver fills a v4l2_buffer struct.

VIDIOC_QBUF

This is used to queue a buffer by passing a v4l2_buffer struct associated to that buffer.

VIDIOC_DQBUF

This is used to dequeue a buffer by passing a v4l2_buffer struct associated to that buffer.

VIDIOC_STREAMON

This is used to turn on streaming. After that, any VIDIOC_QBUF results in an image being rendered.

VIDIOC_S_CTRL VIDIOC_G_CTRL VIDIOC_QUERYCTRL

These ioctls are used to set/get and query various V4L2 controls like rotation, mirror and background color. Currently only rotation is supported

VIDIOC_STREAMOFF

This is used to turn off streaming.

6.5.3. SYSFS Software Interfaces

User can control all dynamic configuration of DSS core and Fbdev functionality thorugh SYSFS interface.

6.5.3.1. Frame-buffer Driver sysfs attributes

Following attributes are available for user control -

```
root@omap3evm:~#
root@omap3evm:~# ls -1 /sys/class/graphics/fb0/
bits_per_pixel
lank
console
cursor
dev
device
mirror
mode
modes
name
overlays
overlays_rotate
pan
phys_addr
power
rotate
rotate_type
size
state
stride
```

```
subsystem
uevent
virt_addr
virtual_size
root@omap3evm:~#
```

SYSFS attribute	Description
bits_per_pixel	Allows user to control bits per pixel configuration, currently the supported values are 16, 24 and 32. `# echo 16/24/32 > /sys/class/graphics/fb0/bits_per_pixel`
blank	Allows user to control lcd display blanking configuration independently. `# echo 0/4  > /sys/class/graphics/fb0/blank` Values only 0(FB_BLANK_UNBLANK) and 4(FB_BLANK_POWERDOWN) is supported
rotate	Allows user to control rotation through this entry, `# echo 0/1/2/3 > /sys/class/graphics/fb0/rotate` 0 - 0 degree, 1 - 90 degree, 2 - 180 degree and 3 - 270 degree respectively.
rotate_type	Allows user to control rotation type through this entry, `# echo 0/1 > /sys/class/graphics/fb0/rotate_type` 0 - DMA based rotation, 1 - VRFB based rotation. Currently only VRFB based rotation is supported.
virtual_size	Allows user to configure xres_virtual and yres_virtual parameters of frame-buffer, `# cat /sys/class/graphics/fb0/virtual_size` 480,640

SYSFS attribute	Description
virt_addr	Readonly entry, displays virtual address of the frame-buffer memory.
phys_addr	Readonly entry, displays physical address of the frame-buffer memory.

Table 6.6. Frame-buffer Driver sysfs attributes

6.5.3.2. DSS Library sysfs attributes

DSS library provides/exports following attributes, which explained in detail below -

```
root@omap3evm:~#
root@omap3evm:~# ls -1 /sys/devices/platform/omapdss/
bus
display0
display1
display2
driver
manager0
manager1
microamps_requested_vdda_dac
modalias
overlay0
overlay1
overlay2
power
subsystem
uevent
root@omap3evm:~#
```

6.5.3.2.1. DSS Library: display0/1/2

In all total 3 output displays are supported on EVM,

```
root@omap3evm:~#
root@omap3evm:~# ls -1 /sys/devices/platform/omapdss/display0/
bus
driver
enabled
microamps_requested_vdvi
mirror
name
power
rotate
subsystem
tear_elim
timings
```

```
uevent
update_mode
wss
root@omap3evm:~#
```

SYSFS attribute	Description
enabled	User can enable/disable the display through this entry
timings	Displays the timing configuration for specific display panel
name	Shows name of the display panel/output

Table 6.7. DSS Library-display0/1/2: sysfs attributes

6.5.3.2.2. DSS Library: Manager0/1

In all total 2 managers are supported on EVM,

```
root@omap3evm:~#
root@omap3evm:~# ls -1 /sys/devices/platform/omapdss/manager0/
alpha_blending_enabled
default_color
display
name
trans_key_enabled
trans_key_type
trans_key_value
root@omap3evm:~#
```

SYSFS attribute	Description
alpha_blending_ enabled	User can enable/disable Alpha-blending through this entry.
display	Allows user to control the output display, user can set the output to any of the display.
trans_key_enabled	User can enable/disable Transparency key keying through this entry.
trans_key_type	User can control the Transparency key type here.
trans_key_value	User can configure Transparency color keying value through this entry.

Table 6.8. DSS Library-Manager0/1: sysfs attributes

6.5.3.2.3. DSS Library: Overlay0/1/2

In all total 3 Overlays/Planes/Pipelines are supported on EVM,

```
root@omap3evm:~#
root@omap3evm:~# ls -1 /sys/devices/platform/omapdss/overlay0/
enabled
global_alpha
input_size
manager
name
output_size
position
screen_width
root@omap3evm:~#
```

SYSFS attribute	Description
enabled	User can enable/disable overlay through this entry.
global_alpha	User can configure global alpha value through this entry.
manager	Allows control over manager <-> overlay interface, user can configure any overlay to any of the manager.

Table 6.9. DSS Library-Overlay0/1/2: sysfs attributes

6.5.4. Miscellaneous Configurations

The default setup/configuration is -

```
      GFX   => - -  \       DVI
                     \
      Vid1  =>     => =>    LCD
                     /
      Vid2  => _ _  /       TV
```

User can control/configure the various interfaces like, overlay <=> manager <=> display. This section demonstrate/explains the dynamic switching of output using above interfaces.

6.5.4.1. Switching output from LCD to DVI

Follow the steps below to switch output from LCD to DVI interface:

- Disable LCD display

```
# echo 0 > /sys/devices/platform/omapdss/display0/enabled
```

- Disable manager link to display

```
# echo "" > /sys/devices/platform/omapdss/manager0/display
```

- Configure the framebuffer driver for target display panel size

```
# fbset -fb /dev/fb0 -xres $w -yres $h -vxres $w -vyres $h
```

- Configure manager to DVI display interface

```
# echo "dvi" > /sys/devices/platform/omapdss/manager0/display
```

- Enable DVI display

```
# echo 1 > /sys/devices/platform/omapdss/display2/enabled
```

Note

Similar steps must be followed for switching from DVI to LCD.

Note

Please note that the user can read the panel configuration through sysfs entry "/sys/devices/platform/omapdss/display<index>/timings".

6.5.4.2. Switching Overlay0 (GFX) output from LCD to TV

Follow below steps to switch output from LCD to TV interface -

- Disable GFX overlay

```
# echo 0 > /sys/devices/platform/omapdss/overlay0/enabled
```

- Disable GFX overlay link to LCD manager

```
# echo "" > /sys/devices/platform/omapdss/overlay0/manager
```

- Disable LCD display output

```
# echo 0 > /sys/devices/platform/omapdss/display0/enabled
```

- Configure the framebuffer driver for target display panel size

```
# fbset -fb /dev/fb0 -xres $w -yres $h -vxres $w -vyres $h
```

- Switch GFX overlay to TV manager

```
# echo "tv" > /sys/devices/platform/omapdss/overlay0/manager
```

- Enable TV display interface

```
# echo 1 > /sys/devices/platform/omapdss/display1/enabled
```

- Enable GFX overlay

```
# echo 1 > /sys/devices/platform/omapdss/overlay0/enabled
```

Note

Similar steps must follow for other (Video 1 & 2) overlays.

6.6. Driver Configuration

6.6.1. V4L2 video driver

To V4L2 video driver start the *Linux Kernel Configuration* tool.

```
$ make menuconfig ARCH=arm
```

Select *Device Drivers* from the main menu.

```
    ...
    ...
    Kernel Features  --->
    Boot options  --->
    CPU Power Management  --->
    Floating point emulation  --->
    Userspace binary formats  --->
    Power management options  --->
[*] Networking support  --->
    Device Drivers  --->
    ...
    ...
```

Select *Multimedia support* from the menu.

```
    ...
    ...
    Sonics Silicon Backplane  --->
    Multifunction device drivers  --->
[*] Voltage and Current Regulator Support  --->
<*> Multimedia support  --->
    Graphics support  --->
<*> Sound card support  --->
[*] HID Devices  --->
[*] USB support  --->
    ...
    ...
```

Select *Video For Linux* from the menu.

```
    ...
    ...
      *** Multimedia core support ***
<*>   Video For Linux
[*]     Enable Video For Linux API 1 (DEPRECATED)
```

```
< >   DVB for Linux
    ...
    ...
```

Select *Video capture adapters* from the same menu. Press <ENTER> to enter the corresponding sub-menu.

```
    ...
    ...
[ ]   Customize analog and hybrid tuner modules to build  --->
[*]   Video capture adapters  --->
[ ]   Radio Adapters  --->
[ ]   DAB adapters
    ...
    ...
```

Select *TI Media Drivers* from the menu. After selecting this option sub-menu will appear.

```
    ...
    ...
<*>   OMAP ISP Resizer
<*>   TI Media Drivers
    ...
    ...
```

Select *OMAP2/OMAP3 V4L2-DSS drivers* from the menu.

```
    ...
    ...
<*>   TI Media Drivers
< >      VPSS System module driver
<*>   OMAP2/OMAP3 V4L2-DSS drivers
< >   SoC camera support
    ...
    ...
```

6.6.2. Framebuffer driver

```
$ make menuconfig
```

Select *Device Drivers* from the main menu.

```
    ...
    ...
    Kernel Features  --->
    Boot options  --->
    CPU Power Management  --->
    Floating point emulation  --->
    Userspace binary formats  --->
    Power management options  --->
[*] Networking support  --->
    Device Drivers  --->
    ...
    ...
```

Select *Graphics support* from the menu.

```
    ...
    ...
    Sonics Silicon Backplane  --->
    Multifunction device drivers  --->
[*] Voltage and Current Regulator Support  --->
<*> Multimedia support  --->
    Graphics support  --->
<*> Sound card support  --->
[*] HID Devices  --->
[*] USB support  --->
    ...
    ...
```

Select *Support for frame buffer devices* from the menu.

```
    ...
    ...
<M> Lowlevel video output switch controls
<*> Support for frame buffer devices  --->
< > E-Ink Broadsheet/Epson S1D13521 controller support
[ ] Check bootloader initialization
-*- OMAP2/3 Display Subsystem support (EXPERIMENTAL)  --->
[ ] Backlight & LCD device support  --->
    ...
    ...
```

Select *OMAP2/3 Display Subsystem support (EXPERIMENTAL)* from the same menu.

```
    ...
    ...
<M> Lowlevel video output switch controls
<*> Support for frame buffer devices  --->
[ ] Check bootloader initialization
-*- OMAP2/3 Display Subsystem support (EXPERIMENTAL)  --->
[ ] Backlight & LCD device support  --->
```

```
    ...
    ...
```

Configure default VRAM size to the required/expected size of buffer, the default is 4MB.

```
    ...
--- OMAP2/3 Display Subsystem support (EXPERIMENTAL)
(4)   VRAM size (MB)
[ ]   Debug support
    ...
    ...
```

Select *VENC support* from the menu.

```
    ...
    ...
[ ]   Debug support
[ ]   RFBI support
[*]   VENC support
         OMAP2_VENC_OUT_TYPE (Use S-Video output interface)  --->
 [ ]   SDI support
    ...
    ...
```

The default TV out interface is *S-Video*. Other option available is *Composite*.

Note

On OMAP3EVM-1 (< Rev-E) SW1.6 is used to select between S-Video and Composite outputs. **ON :- S-Video, OFF :- Composite**

Select minimum functional/pixel clock ratio for scaling to required value, the default value if 4.

```
    ...
    ...
[ ]   Fake VSYNC irq from manual update displays
(4)   Minimum FCK/PCK ratio (for scaling)
<*>   OMAP2/3 frame buffer support (EXPERIMENTAL)  --->
    ...
    ...
```

Select *OMAP2/3 frame buffer support (EXPERIMENTAL)* from the same menu. Press <ENTER> to enter the corresponding sub-menu.

```
    ...
    ...
[ ]   Fake VSYNC irq from manual update displays
(0)   Minimum FCK/PCK ratio (for scaling)
<*>   OMAP2/3 frame buffer support (EXPERIMENTAL)  --->
      OMAP2/3 Display Device Drivers  --->
    ...
    ...
```

Value for *Number of framebuffers* can be changed here.

```
    ...
    ...
 [ ]   Force main display to automatic update mode
 (1)   Number of framebuffers
    ...
    ...
```

Note

If this value is set as **1**, the graphics pipeline of the DSS is controlled by the FBDEV interface and both video pipelines by the V4L2 interface.

If this value is set as **2**, the graphics pipeline and one video pipeline is controlled by the FBDEV interface and one video pipeline by the V4L2 interface.

If this value is set as **3**, all 3 pipelines are controlled by the FBDEV interface.

Select the supported display panels, as shown below

```
<*> Generic Panel
< > Samsung LTE430WQ-F0C LCD Panel
<*> Sharp LS037V7DW01 LCD Panel
< > Sharp LQ043T1DG01 LCD Panel
```

6.7. Sample Application Flow

This chapter describes the application flow using the V4l2 and FBDEV drivers.

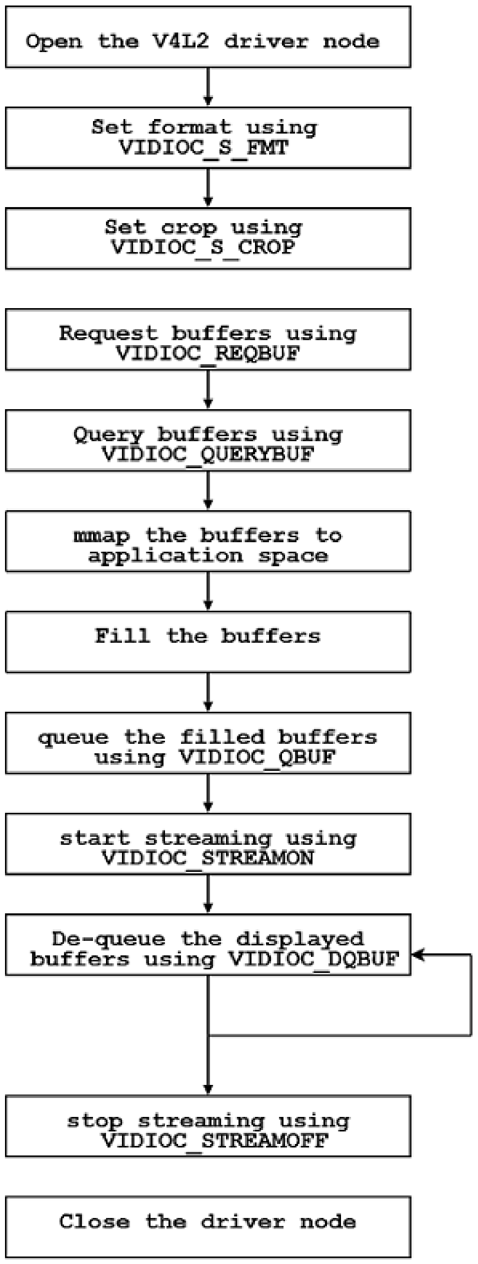

Open the V4L2 driver node
Set format using VIDIOC_S_FMT
Set crop using VIDIOC_S_CROP
Request buffers using VIDIOC_REQBUF
Query buffers using VIDIOC_QUERYBUF
mmap the buffers to application space
Fill the buffers
queue the filled buffers using VIDIOC_QBUF
start streaming using VIDIOC_STREAMON
De-queue the displayed buffers using VIDIOC_DQBUF
stop streaming using VIDIOC_STREAMOFF
Close the driver node

Figure 6.19. Application for v4l2 driver using MMAP buffers

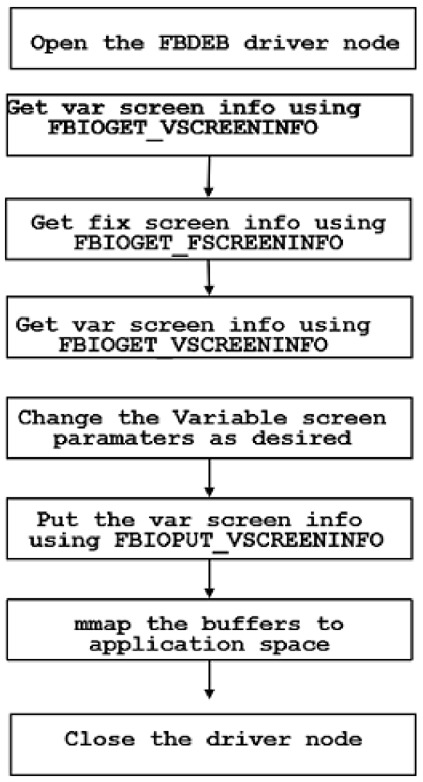

Open the FBDEB driver node
Get var screen info using FBIOGET_VSCREENINFO
Get fix screen info using FBIOGET_FSCREENINFO
Get var screen info using FBIOGET_VSCREENINFO
Change the Variable screen paramaters as desired
Put the var screen info using FBIOPUT_VSCREENINFO
mmap the buffers to application space
Close the driver node

Figure 6.20. Application for FBDEV driver

6.8. Revision History

03.00.00.02	Updated for 03.00.00.02 PSP release
03.00.00.03	Updated for 03.00.00.03 PSP release

Resizer Driver

Abstract

This chapter provides detailed description of feature set and software interface for the video resizer driver implementation.

Table of Contents

7.1. Introduction

This section provides overview of the Resizer Hardware.

7.1.1. References

- Video for Linux Two API Specification [http://v4l2spec.bytesex.org/v4l2spec/v4l2.pdf]
- Linux Device Drivers Edition-3 [http://lwn.net/Kernel/LDD3/]

7.1.2. Acronyms

- V4L2: Video For Linux 2

7.1.3. Hardware Overview

The OMAP Resizer module enables up scaling and down scaling. It resizes YUV422 image and stores output image in the RAM. The following figure shows the block diagram for Resizer module.

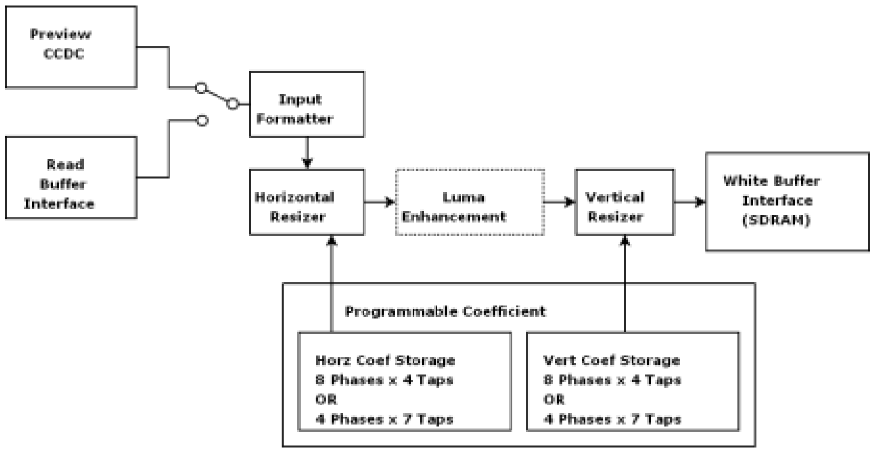


Figure 7.1. OMAP Resizer HW Block Diagram

The Resizer module performs digital zoom either up sampling or down sampling on image/video data within a range of 0.25x to 4x resizing. The input source can be sent to either the preview engine/CCDC or memory, and the output is sent to memory.

The Resizer module performs horizontal resizing, then vertical resizing, independently. Between them, there is an optional edge-enhancement feature. This process is shown in the above Figure.

7.2. Features

This section describes features supported by the resizer driver.

7.2.1. Overview of features supported

The Resizer driver supports the following features:

- Resizes input frame stored in RAM and stores output frame in RAM.
- Supports resizing from 1/4x to 4x.
- Supports independent horizontal and vertical resizing.
- Supports YUV422 packed data and Color Separate data.
- Supports driver allocated and user provided buffers.
- Supports Luminance Enhancement.
- Supports configuration of read request cycles.

7.2.2. Usage of Features

Following sections provides details about the drive supported features.

7.2.2.1. Opening and Closing the Driver

The device can be opened using open call from the application with device name and mode of operation as parameters. Mode can be blocking, non-blocking and readwrite. Application can open the driver in either blocking mode or non-blocking mode. If driver is opened in blocking mode, RSZ_RESIZE ioctl will block until resizing task is over for that channel. If the driver is opened in non-blocking mode, RSZ_RESIZE ioctl returns if hardware is busy serving other channel.

Driver can be opened multiple times. Driver maintains software channels for all opened instances. If multiple resizing task is submitted at the same time, driver serializes the resizing task.

To close a specific device, application calls the close function with the file handle.

```
/* call to open a Resizer logical channel in blocking mode */
rszfd_blocking = open ("/dev/omap-resizer", O_RDWR);
if (rszfd_blocking == -1) {
 perror("failed to open Resizer device\n");
 return -1;
}
/* closing of channels */
```

```
close (rszfd_blocking);
```

7.2.2.2. Buffer Management

Resizer Driver requires buffers for storing input/output images. Buffers can be allocated by the driver itself or application can provide the buffers. These buffers need to be virtually contiguous. Physically buffers can be scattered in the memory.

The Resizer Driver supports two memory usage models.

- Memory map/Driver allocated buffer mode
- User Pointer Exchange

7.2.2.2.1. Memory map/Driver Allocated buffer

In Memory map buffer mode, application can request memory from the driver by calling `RSZ_REQBUF` ioctl. With this ioctl, application passes pointer to the structure `v4l2_requestbuffers` . In this structure, buffer exchange mechanism can be specified. For memory map buffer exchange mechanism, it should be `V4L2_MEMORY_MMAP`. Driver always allocates buffer of maximum size required to store input or output image. If output image width is less than input image width, a single buffer can be used to store input and output images. Otherwise, at least, two buffers are required to store input and output images. Application can request as many buffers as it wants. Buffer with the index 0 is always used as the input buffer and other buffer can be used as the output buffer.

The main steps that the application must perform for buffer allocation are:

- Allocating Memory
- Getting Physical Address
- Mapping Kernel Space Address to User Space

Allocating Memory:

Application can allocate buffers using `RSZ_REQBUF` IOCTL. While allocating the buffers, the application have to specify the buffer type, number of buffers and memory type. Here the buffer type must be `V4L2_BUF_TYPE_VIDEO_CAPTURE`. Number of buffers can be greater than or equal to one. If output image is less than input image, number of buffers can be one. Drivers always uses maximum of input and output image size as the buffer size.

This ioctl takes object of `v4l2_request` buffer structure.

```
/* structure to store buffer request parameters */
```

```
struct v4l2_requestbuffers reqbuf;
reqbuf. type =V4L2_BUF_TYPE_VIDEO_CAPTURE;
reqbuf.count = 2; /* number of buffers */
reqbuf.memory = V4L2_MEMORY_MMAP; /* Type of buffer exchange
 mechanism */
if(ioctl(rszfd, RSZ_REQBUF, &reqbuf)< 0) {
        perror("RSZ_REQBUF failed\n");
        close(rszfd);
        exit(-1);
}
/* The above example will allocate 2 buffers */
```

Getting Physical Address:

The RSZ_QUERYBUF IOCTL can obtain the physical address of the allocated buffer. Application has to specify the index, buffer type and buffer's memory type at time of calling this ioctl. Buffer type must be V4L2_BUF_TYPE_VIDEO_CAPTURE. Index of each type of buffer starts from 0. Buffer memory type must be V4L2_MEMORY_MMAP for driver allocated buffers. The driver fills the size and physical address, and then returns to the application so that the relevant data can be used to mmap the buffer to user space.

This ioctl takes object of the structure v4l2_buffer.

```
/* allocate buffer by RSZ_REQBUF */
/* structure to query the physical address of allocated buffer */
struct v4l2_buffer buffer;
buffer.index = 0; /* buffer index - 0 */
buffer.type = V4L2_BUF_TYPE_VIDEO_CAPTURE; /* Input buffer */
if (ioctl(rszfd, RSZ_QUERYBUF, &buffer) < 0) {
        perror("RSZ_QUERYBUF ioctl failed\n");
        close(rszfd);
        exit(-1);
}
/* The buffer.m.offset will contain the physical address returned
 from driver */
```

Mapping Kernel Space Address to User Space:

Mapping the kernel buffer to the user space is done via the Linux mmap system call. Application must pass the buffer size and buffer's physical address for getting the user mapped address.

```
/* allocate buffer by RSZ_REQBUF */
/* query the buffer using RSZ_REQBUF */
/* addr hold the user space address */
unsigned long addr;
addr = mmap(NULL, buffer.size, PROT_READ | PROT_WRITE,
      MAP_SHARED, rszfd, buffer.offset);
/* buffer.offset is same as returned from RSZ_QUERYBUF */
```

7.2.2.2.2. User Pointer Exchange

Application informs driver whether to use memory mapped buffer or user buffer in memory allocation operation (RSZ_REQBUFS ioctl). This ioctl is being used when request for buffer allocation is submitted to the driver. Along with ioctl, application has to specify either memory mapped or user buffer to be used. If user provided buffer is used, application has to pass memory type as V4L2_MEMORY_USERPTR. Then at the time of en-queueing buffers, application can specify pointer to the virtually contiguous buffer allocated by the application and size of the buffer. This size of the buffer, specified at the time of en-queueing buffer, must be page-aligned.

```
/* structure to store buffer request parameters */
struct v4l2_requestbuffers reqbuf;
reqbuf. type =V4L2_BUF_TYPE_VIDEO_CAPTURE;
reqbuf.count = 2; /* number of buffers */
reqbuf.memory = V4L2_MEMORY_USERPTR; /* Type of buffer exchange
 mechanism */
if(ioctl(rszfd, RSZ_REQBUF, &reqbuf)< 0) {
        perror("RSZ_REQBUF failed\n");
        close(rszfd);
        exit(-1);
}
/* This above example will allocate 2 buffer descriptors */
```

7.2.2.3. Parameter Configuration

7.2.2.3.1. Resizing

The Resizer module takes input image from RAM, resizes it horizontally and vertically using given resizing ratio and stores output image in RAM. The Resizer module can up scale or down scale image data with independent resizing factors in the horizontal and vertical directions. The resizing ratio is calculated using formula 256/N where value of N can range from 64 to 1024. The Resizer module uses the same resampling algorithm for the horizontal and vertical directions. The resizing/resampling algorithm uses a programmable polyphase sample rate converter (resampler). The polyphase filter coefficients are programmable so that any user-specified filter can be implemented.

For horizontal and vertical direction, application has to provide 32 coefficients. These coefficient values are dependent on resizing ratio. The Resizer hardware uses 4 taps and 8 phases filters for the resizing range of 1/2x to 4x and 7 taps and 4 phases filters for a resizing range of 1/4x to 1/2x for both the direction. So different set of coefficients must be provided for 4 taps and 8 phases filters and 7 taps and 4 phases filter.

As the hardware uses multi tape poly phase filters, filter requires more input pixel than following equation calculates.

```
Input size = output size * N / 256; /* 256 where N is from 64 to
 1024 */
```

Input size is also dependent on the starting phase and rounding issues in the resizing algorithm of the hardware. The input width and height parameters must be programmed strictly according to these equations given in following table otherwise, incorrect hardware operation may occur.

	Ratio 1/2x to 4x	**Ration 1/4x to 1/2x**
Input width	(32*sph+(ow-1)*hrsz +16)>>(8+7)	(64*sph+(ow-1)*hrsz +32)>>(8+7)
Input height	(32*spv+(oh-1)*vrsz +16)>>(8+4)	(64*spv+(oh-1)*vrsz +32)>>(8+7)

Table 7.1. Resizer: Input Size Calculation

```
     Where
        sph = horizontal starting phase
        spv = vertical starting phase
        ow = output width
        oh = output height
        hrsz = horizontal resize value
        vrsz = vertical resize value.
```

Application sets the resizing ratio by providing input size and output size parameters in `RSZ_S_PARAMS` ioctl. This ioctl takes object of `rsz_params` structures. Application provides input width, pitch and height and output width, pitch and height. Driver calculates the resizing ratio for horizontal and vertical direction using below equation.

```
Horizontal_ratio = (input_width-N)*256/(output_width-1);
```

`Where N = 7 for ratio in between 1/4x to 1/2x 4 for ratio in between 1/2x to 4x`

Similar equation is used to calculate vertical resizing ratio. So this equation must be used by the application when calculating input and output size for the given resizing ratio.

Actual Resizing operation is performed when application calls RSZ_RESIZE ioctl. This ioctl programs Resizer Hardware, submits resizing task and waits for it to be completed. This ioctl will block the application if the resizer driver is opened in blocking mode. If it is opened in non-blocking mode, it will simply return with busy if the hardware is busy. Before submitting resizing task, the input and output buffers must be en-queued to the driver so the drive will come to know which buffers to be used as input and output. Also resize ioctl, as an side effect, removed buffers from the queue. So if resizing task required to be resubmitted, buffers must be en-queued again. So whenever resizing task is submitted, input and output buffers must also be enqueued first.

7.2.2.3.2. Chroma Algorithm

Chroma components, which are 2:1 horizontally down sampled with respect to luma, have two methods of horizontal resizing: `Filtering with luma`, and `Bilinear interpolation`. The Chroma algorithm option can be selected in the cbilin field of `rsz_params` structure. However, filtering with luma is only intended for down sampling, and bilinear interpolation is only intended for up sampling.

There are two possible values of cbilin member of `rsz_params` structures. `0` and `1`. `0` indicates that the chrominance uses the same processing as the luminance and `1` indicates that the chrominance uses bilinear interpolation processing.

7.2.2.3.3. Input/output image format

Resizer module supports two types of image format. One is YUV422 packed data and the other is color separate data. Input image format can be selected by providing one of `RSZ_INTYPE_YCBCR422_16BIT` or `RSZ_INTYPE_PLANAR_8BIT` value to inptype member of `rsz_params` structure. Configured input image format is used for both input and output image.

When YUV422 interleaved (packed) image format is selected, resizer module resizes entire image and stores it in output buffer.

When color separate image format is selected, application has to resize each of the color components separately. Application must open three instances of the resizer driver and resize each color components in one instance separately. Application must provide correct input and output size of the each color components when resizing color components.

7.2.2.3.4. Pixel Format

Resizer module supports two pixel format YUYV and UYVY. Pixel format can be selected by providing one of `RSZ_PIX_FMT_YUYV` or `RSZ_PIX_FMT_UYVY` value to `pix_fmt` member of `rsz_params` structure. Configured pixel format is used for both input and output image. When color separate input data is used, this field is ignored.

7.2.2.3.5. Luma Enhancement

Edge enhancement can be applied to the horizontally resized luminance component before the output of the horizontal stage is sent to the line memories and the vertical stage. type member of `rsz_yenh` member of `rsz_params` structure can be set to disable edge enhancement, or to select a 3-tap or a 5-tap horizontal high-pass filter (HPF) for luminance enhancement. So possible values of type is 0, 1 or 2 for disabling luma enhancement, selecting 3 tap filed or selecting 5 tap filter. If edge enhancement is selected, the two left-most and two right-most pixels in each line are not outputted to the line memories and the vertical stage. When luma enhancement is enabled, maximum output width can be 1280 when 1/2x to 4x vertical resizing ratio is selected and 640 when 1/4x to 1/2x vertical resizing ratio is selected.

Luma enhancement algorithm is as follows.

HPF (Y_IN) = Y_IN convolved with {[-0.5, 1, 0.5] or [-0.25, -0.5, 1.5, -0.5, -0.25]}

Implemented as [-1, 2, -1] >> 1, [-1, -2, 6, -2, -1] >> 2)

Saturate HPF(Y) between -256 and +255

Hpgain = (|HPF(Y)| - CORE) * SLOP

Saturate hpgain between 0 and GAIN

Y_OUT = Y_IN + (HPF (Y_IN) * hpgain + 8) >> 4

Saturate Y_OUT between 0 and 255

Application have to provide core, slope and gain members of rsz_yenh member of rsz_params structure.

7.2.2.3.6. Configuring the Read cycle for Resizer module

ISP module supports configuration of number of clock cycles between two consecutive read request from resizer module. Value supported is 0 - 0x3FF.

```
unsigned int read_exp;
read_exp = 0xe;
ret_val = ioctl(fd, RSZ_S_EXP, &read_exp);
if (ret_val){
 printf("\nUnable to set the read cycle expand register\n");
 return ret_val;
}
```

The default configuration of read cycle is 0xE.

7.2.3. Constraints

- For driver allocated buffers, driver allocates maximum size of buffers for both input and output.
- All input/output buffers addresses and pitch must be 32 bytes aligned.
- Output image size cannot be more than 2047x2047.

- Output width must be even.
- Output width must be 16 byte aligned for vertical resizing.
- The horizontal start pixel must be within the range: 0 to 15 for color interleaved, 0 to 31 for color separate data.

7.3. Architecture

Following block diagram shows the basic architecture of the Resizer Driver -

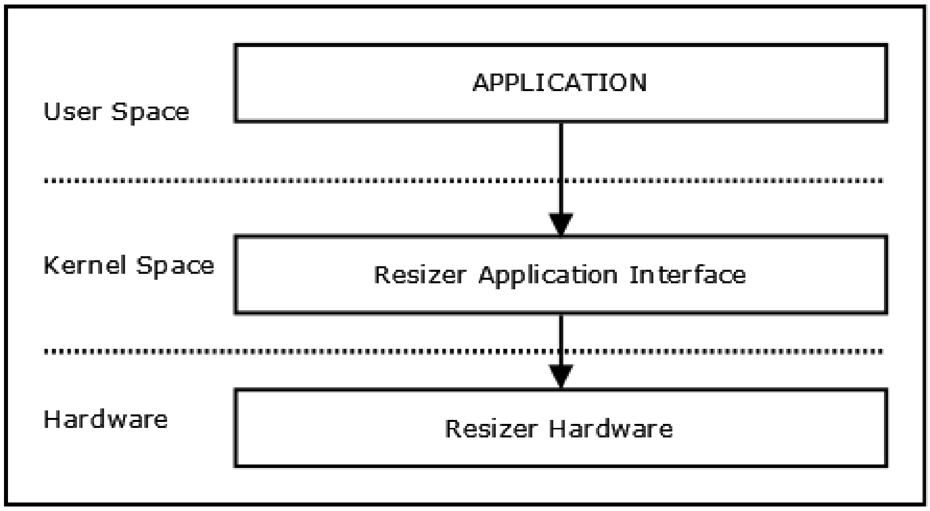


Figure 7.2. Basic Architecture of Resizer Driver

Resizer Driver provides Resizer Hardware access to a channel by using Linux Character driver interface. Driver supports all the features supported by the hardware. It provides easy way of configuring the hardware. To understand this, that the hardware module driver implements, is briefly described in this section.

7.4. Software Interface

This section describes the Data Structures, Enumerations, and API Specifications used in the OMAP Resizer Driver.

7.4.1. Application Programming Interface

7.4.1.1. open

Description

Opens the device driver for processing.
Prototype

```
int fd = open(device_name, mode);
```

Field	Description
`device_name`	It is `/dev/omap-resizer`
`mode`	O_RDWR or ORed with O_NONBLOCK

Table 7.2. Resizer: open System Call arguments

Return Values

Zero on success, or negative if an error has occurred.

7.4.1.2. close

Description

Close the device.
Prototype

```
close(fd);
```

Field	Description
`fd`	File descriptor returned from open call.

Table 7.3. Resizer: close system call arguments

Return Values

Zero on success.

`EINTR`, if driver could not get the handle.

7.4.1.3. mmap

Description

Map the kernel space buffer to user space.
Prototype

```
void * mmap(void *, size_t image_size, int prot, int
  flags, int fd, off_t offset)
```

Field	**Description**
`void *`	Generally NULL
`image_size`	Buffer size that needs to be mapped
`flag`	`PROT_SHARED`
`fd`	File descriptor
`offset`	Physical address of the buffer

Table 7.4. Resizer: mmap system call arguments

Return Values

Zero on success.

`EAGAIN`, if the address is not found.

7.4.1.4. munmap

Description

Unmap the frame buffers that were previously mapped to user space using `mmap()` system call.
Prototype

```
void *munmap(void *start_addr, size_t length)
```

Field	**Description**
`start_addr`	Start address of buffer which is to be unmapped.
`length`	Length of buffer.

Table 7.5. Resizer: munmap system call arguments

Return Values

Zero on success, or Negative if an error has occurred.

7.4.2. IOCTLs

7.4.2.1. RSZ_S_PARAMS

Description

Set the resizer parameters necessary for processing.
Prototype

```
int ioctl(int fd, RSZ_S_PARAMS, struct rsz_params *argp)
```

Field	Description
`fd`	File handle associated with fd.
`cmd`	`RSZ_S_PARAMS` ioctl command.
`argp`	Pointer to `rsz_params` structure.

Table 7.6. Resizer: ioctl `RSZ_S_PARAMS` arguments

Return Values

Zero on success,

`EINVAL`, if parameters are incorrect.

`EINTR`, if device is in use by the same channel handle.

7.4.2.2. RSZ_G_PARAMS

Description

Get the Resizer parameters that are previously being set.
Prototype

```
int ioctl(int fd, RSZ_G_PARAMS,struct rsz_params *argp)
```

Field	Description
`fd`	File handle associated with fd.
`cmd`	`RSZ_G_PARAMS` ioctl command.
`argp`	Pointer to `rsz_params` structure.

Table 7.7. Resizer: ioctl `RSZ_G_PARAMS` arguments

Return Values

Zero on success,

`EINVAL`, if device is not configured before calling this API.

7.4.2.3. RSZ_G_STATUS

Description

Get the channel status for the particular current Resizer channel.
Prototype

```
int ioctl(int fd, RSZ_G_STATUS, struct rsz_status *argp)
```

Field	Description
`fd`	File handle associated with fd.
`cmd`	RSZ_G_STATUS ioctl command.
`argp`	Pointer to `rsz_status` structure.

Table 7.8. Resizer: ioctl RSZ_G_STATUS argument

Return Values

Zero on success,

7.4.2.4. RSZ_S_EXP

Description

Configure the Read cycle required for Resizer module. This configuration is provided per channel.
Prototype

```
          int ioctl(int fd, RSZ_S_EXP, unsigned int *argp)
```

Field	Description
`fd`	File handle associated with fd.
`cmd`	RSZ_S_EXP ioctl command.
`argp`	Pointer to `unsigned int`.

Table 7.9. Resizer: ioctl RSZ_S_EXP argument

Return Values

Zero on success,

7.4.2.5. RSZ_RESIZE

Description

Starts the Resizer processing for the parameters previously set by RSZ_S_PARAMS
Prototype

```
int ioctl(int fd, RSZ_RESIZE, int *argp)
```

Field	Description
`fd`	File handle associated with fd.
`cmd`	`RSZ_RESIZE` ioctl command.
`argp`	Pointer to int.

Table 7.10. Resizer: ioctl `RSZ_RESIZE` arguments

Return Values

Zero on success,

`EINVAL`, if parameters are incorrect.

`EBUSY/EINTR`, if device is in use by the same channel handle.

7.4.2.6. RSZ_REQBUF

Description

Request to allocate buffers
Prototype

```
int ioctl(int fd, RSZ_REQBUF, struct v4l2_requestbuffers *argp)
```

Field	Description
`fd`	File handle associated with fd.
`cmd`	`RSZ_REQBUF` ioctl command.
`argp`	Pointer to `v4l2_requestbuffers` structure.

Table 7.11. Resizer: ioctl `RSZ_REQBUF` arguments

Return Values

Zero on success,

`ENOMEM`, if memory is not available.

`EINTR`, if device is in use by the same channel handle.

7.4.2.7. RSZ_QUERYBUF

Description

Request physical address of buffers allocated by the `RSZ_REQBUF`
Prototype

```
int ioctl(int fd, RSZ_QUERYBUF, struct v4l2_buffer *argp)
```

Field	Description
`fd`	File handle associated with fd.
`cmd`	`RSZ_QUERYBUF` ioctl command.
`argp`	Pointer to `v4l2_buffer` structure.

Table 7.12. Resizer: ioctl `RSZ_QUERYBUF` arguments

Return Values

Zero on success,

`EINVAL/EFAULT`, if parameters are incorrect.

`EINTR`, if device is in use by the same channel handle.

7.4.2.8. RSZ_QUEUEBUF

Description

Queue the buffer for resize operation.
Prototype

```
int ioctl(int fd, RSZ_QUEUEBUF, struct v4l2_buffer *argp)
```

Field	Description
`fd`	File handle associated with fd.
`cmd`	`RSZ_QUEUEBUF` ioctl command.
`argp`	Pointer to `v4l2_buffer` structure.

Table 7.13. Resizer: ioctl `RSZ_QUEUEBUF` arguments

Return Values

Zero on success,

`EINVAL/EFAULT`, if parameters are incorrect.

`EINTR`, if device is in use by the same channel handle.

7.4.3. Data Structures

7.4.3.1. Resizer Parameters Configuration Structure

```
struct rsz_params {
        __s32 in_hsize;
        __s32 in_vsize;
        __s32 in_pitch;
        __s32 inptyp;
        __s32 vert_starting_pixel;
        __s32 horz_starting_pixel;
        __s32 cbilin;
        __s32 pix_fmt;
        __s32 out_hsize;
        __s32 out_vsize;
        __s32 out_pitch;
        __s32 hstph;
        __s32 vstph;
        __u16 tap4filt_coeffs[32];
        __u16 tap7filt_coeffs[32];
 struct rsz_yenh yenh_params;
} ;
```

Name	Description
`in_hsize`	Width of the input image in pixels.
`in_vsize`	Height of the input image in pixels.
`in_pitch`	Pitch of input image in bytes.
`inptype`	Input image format.
`vert_starting_pixel`	Vertical starting pixel.
`horz_starting_pixel`	Horizontal starting pixel.
`cbilin`	Chroma resizing algorithm.
`pix_fmt`	Image Pixel format for `YUV422` image.
`out_hsize`	Width of the output image in pixels.
`out_vsize`	Height of the output image in pixels.
`out_pitch`	Pitch of the output image in bytes.
`hstph`	Horizontal starting phase.
`vstph`	Vertical starting phase.
`tap4filt_coeffs`	Set of coefficients for scaling ratio 0.5x - 4x.
`tap7filt_coeffs`	Set of coefficients for scaling ratio 0.25x - 0.5x.
`yenh_params`	Luma Enhancement parameters.

Table 7.14. Resizer: Parameters Configuration Structure fields

7.4.3.2. Request Buffer Structure

```
struct v4l2_requestbuffer {
 unsigned int type;
 unsigned int count;
 enum v4l2_memory memory;
 ...
}
```

Name	Description
`type`	Buffer type `V4L2_BUF_TYPE_VIDEO_CAPTURE`.

Name	Description
count	Number of buffers to be allocated.
memory	Type of the buffer exchange mechanism requested.

Table 7.15. Resizer: Request Buffer Structure fields

7.4.3.3. Buffer structure

```
struct v4l2_buffer {
 unsigned int index;
 unsigned int type;
 enum v4l2_memory memory;
 union {
  unsigned long offset;
  unsigned long userptr
 }m;
}
```

Name	Description
index	Index of the input/output buffer.
type	Type of the buffer is V4L2_BUF_TYPE_VIDEO_CAPTURE.
offset/userptr	Physical/virtual address of the buffer.
memory	Type of memory, V4L2_MEMORY_MMAP or V4L2_MEMORY_USERPTR.

Table 7.16. Resizer: Buffer structure fields

7.4.3.4. Luma enhancement structure

```
struct rsz_yenh {
 __s32 type;
 __u8 gain;
 __u8 char slop;
 __u8 core;
}
```

Name	Description
type	Luma Enhancement algorithm.
gain	Gain.
slop	Slop.
core	Core.

Table 7.17. Resizer: Luma enhancement structure fields

7.4.3.5. Status structure

```
struct rsz_status {
 __s32 chan_busy;
 __s32 hw_busy;
 __s32 src;
}
```

Name	Description
chan_busy	Status of the channel.
hw_busy	Status of the hardware.
src	Input source.

Table 7.18. Resizer: Status structure fields

7.4.3.6. Crop Size structure

```
struct rsz_cropsize {
 __u32 hcrop;
 __u32 vcrop;
}
```

Name	Description
hcrop	Number of pixels cropped in horizontal direction.
vcrop	Number of pixels cropped in vertical direction.

Table 7.19. Resizer: Crop Size structure fields

7.4.3.7. Input/Output image format

This describes the input and output image format, which can be YUV interleaved 16 bit or planar 8 bit. This can be specified in `inptype` field of the `rsz_params` structure.

```
#define RSZ_INTYPE_YCBCR422_16BIT 0
#define RSZ_INTYPE_PLANAR_8BIT  1
```

7.4.3.8. Pixel Format

This describes pixel format for the YUV interleaved data. This can be specified in `pix_fmt` member of `rsz_params` structure.

```
#define RSZ_PIX_FMT_UYVY  1 /* cb:y:cr:y */
#define RSZ_PIX_FMT_YUYV  0 /* y:cb:y:cr */
```

7.5. Driver Configuration

7.5.1. Configuration Steps

To enable OMAP Resizer driver, start *Linux Kernel Configuration* tool.

```
$ make menuconfig  ARCH=arm
```

Select *Device Drivers* from the main menu.

```
    ...
    ...
    Kernel Features  --->
    Boot options  --->
    CPU Power Management  --->
    Floating point emulation  --->
    Userspace binary formats  --->
    Power management options  --->
[*] Networking support  --->
    Device Drivers  --->
    ...
    ...
```

Select *Multimedia support* from the menu.

```
    ...
    ...
    Sonics Silicon Backplane  --->
    Multifunction device drivers  --->
[*] Voltage and Current Regulator Support  --->
<*> Multimedia support  --->
    Graphics support  --->
<*> Sound card support  --->
[*] HID Devices  --->
[*] USB support  --->
    ...
    ...
```

Select *Video For Linux* from the menu.

```
    ...
    ...
      *** Multimedia core support ***
<*>   Video For Linux
[*]     Enable Video For Linux API 1 (DEPRECATED)
```

```
< >   DVB for Linux
    ...
    ...
```

Select *Video capture adapters* from the same menu. Press <ENTER> to enter the corresponding sub-menu.

```
    ...
    ...
[ ]   Customize analog and hybrid tuner modules to build  --->
[*]   Video capture adapters  --->
[ ]   Radio Adapters  --->
[ ]   DAB adapters
    ...
    ...
```

Select *OMAP ISP Resizer* from the menu.

```
    ...
    ...
<*>   OMAP 3 Camera support
< >   OMAP ISP Previewer
<*>   OMAP ISP Resizer
<*>   TI Media Drivers
    ...
    ...
```

7.6. Sample Application Flow

This section shows application flow diagram for resizer application.

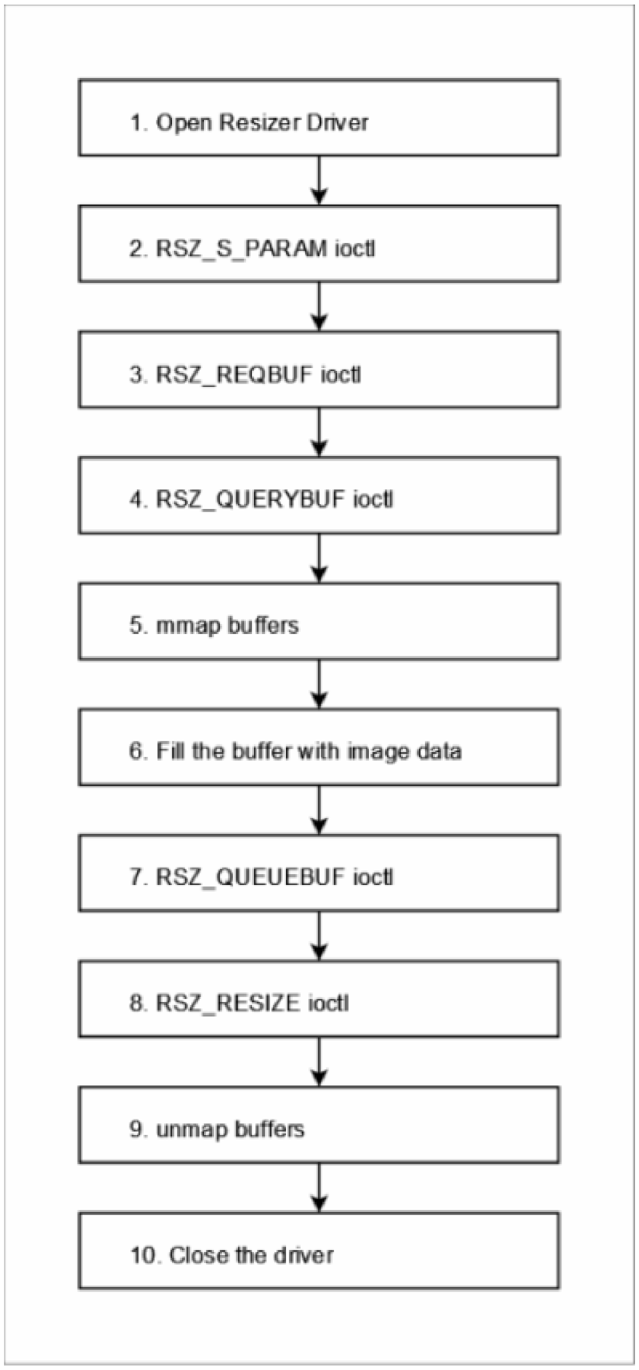


Figure 7.3. Resizer Sample Application Flow

7.7. Revision History

0.97	Initial Draft.
02.00.01	Updated for the second snapshot release.
03.00.00.02	Updated for PSP03.00.00.03 release.

Daughter Card Module

Abstract

This chapter provides detailed description of feature set supported on Daughter Card and software package.

Table of Contents

8.1. Mass Market Daughter Card

8.1.1. Acronyms & Definitions

Acronym	Definition
MMDC	Multi-Media Daughter Card/Customer Daughter Card

Table 8.1. MMDC Acronyms

8.1.2. Introduction

OMAP35x daughter-card (MMDC) supports following features which are not available on the main OMAP3EVM-1 (<Rev-E).

Note

Please note that all the MMDC components/peripherals have been moved on-board for OMAP3EVM-2 (>=Rev-E). So this section is applicable only for OMAP3EVM-1 (<Rev-E).

1. TVP5146 decoder interface supporting BT656 format.
2. Supports 3 types of video input types - S-Video, Composite and component.
3. Supports 8/10 bit output interface from TVP5146.
4. Supports interface for Micron sensor.
5. HSUSB TRANSCEIVER- USB83320 supporting EHCI on port 2

8.2. Block Diagram

The top level block depicts the features supported on the daughter-card.

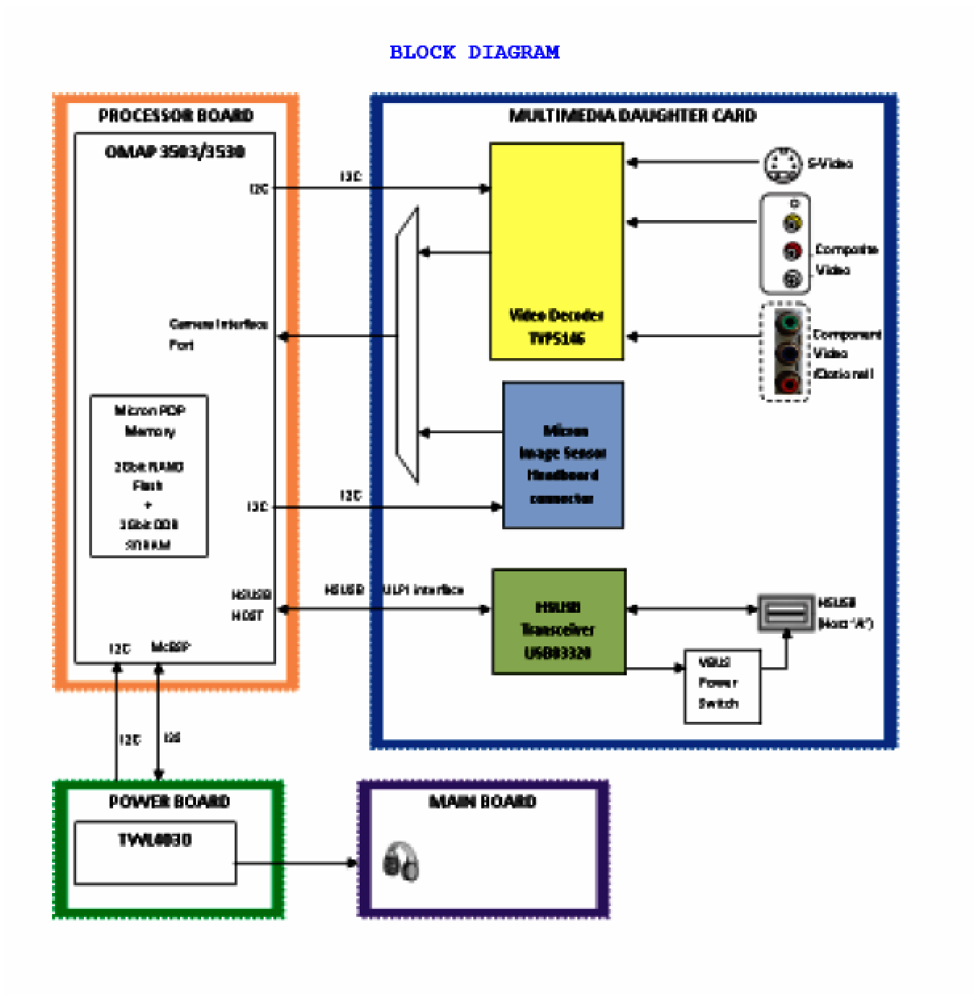


Figure 8.1. Block Diagram

8.3. Board Illustration

The various connectors and hardware modules on the daughter card are illustrated in the picture below:

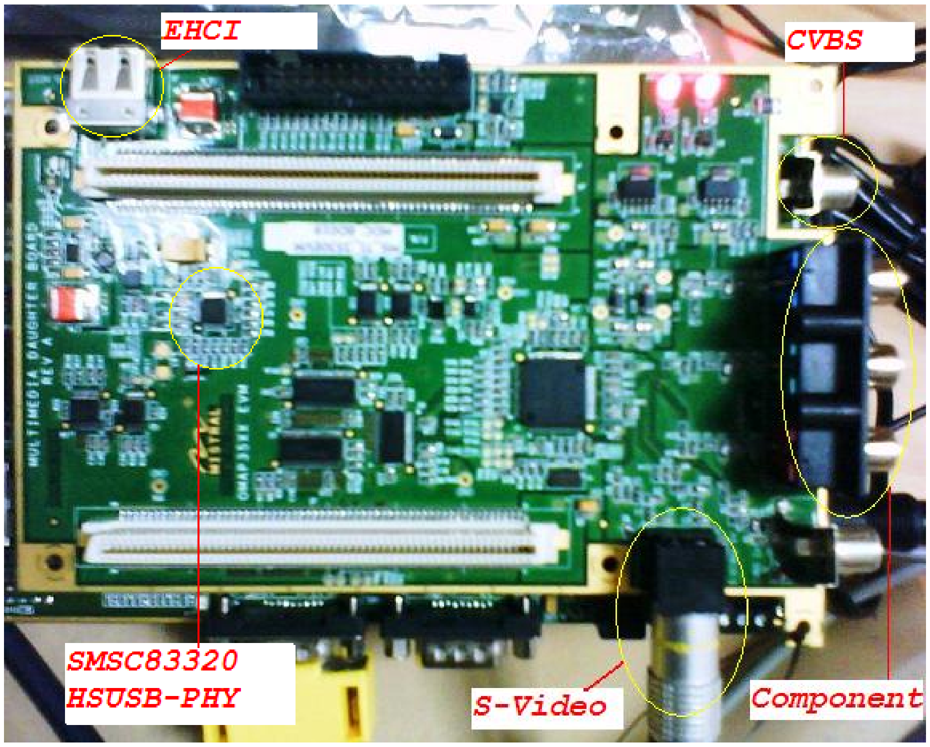


Figure 8.2. Board Illustration

8.4. Features supported under software

- Video capture (BT656 interface) using the TVP5146 decoder.
- Support Composite and S-video interface only.
- EHCI on USB port-2 using HSUSB TRANSCEIVER- USB83320.

Capture Driver

Abstract

This chapter provides detailed description of feature set and software interface for the video Capture driver implementation.

Table of Contents

9.1. Introduction

The camera ISP is a key component for imaging and video applications such as video preview, video record, and still-image capture with or without digital zooming.

The camera ISP provides the system interface and the processing capability to connect RAW image-sensor modules and video decoders to the OMAP35x device.

The capture module consists of the following interfaces:

- One S-video SD input in BT.656 format.
- One Composite SD input in BT.656 format.

Both these video inputs are connected to one TVP5146 decoder and the application can select between these two inputs using standard V4L2 interface.

Note

Only one input can be captured or selected at any given point of time.

The following figure shows the basic block diagram of capture interface.

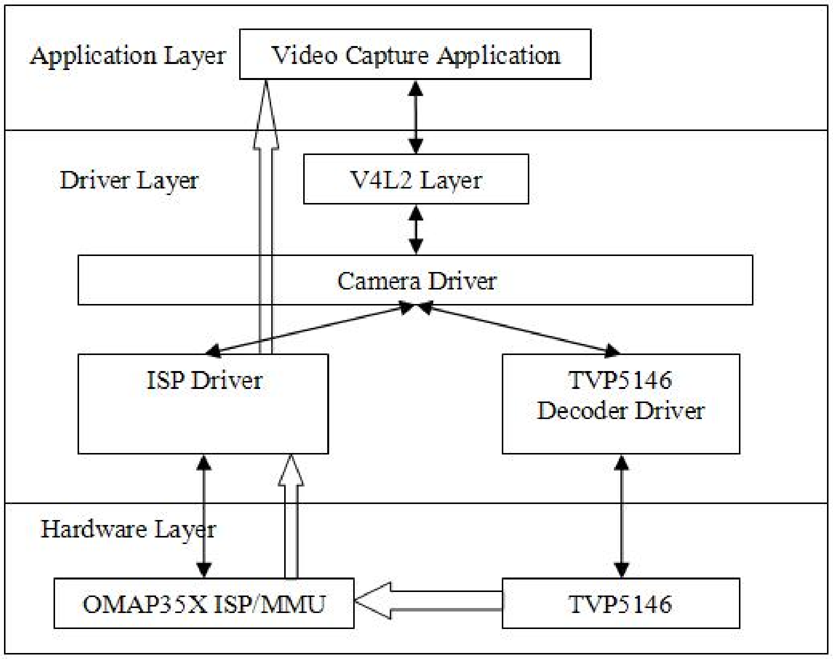


Figure 9.1. Capture Driver Component Overview

The following figure shows the physical connection and inputs for TVP5146 decoder.

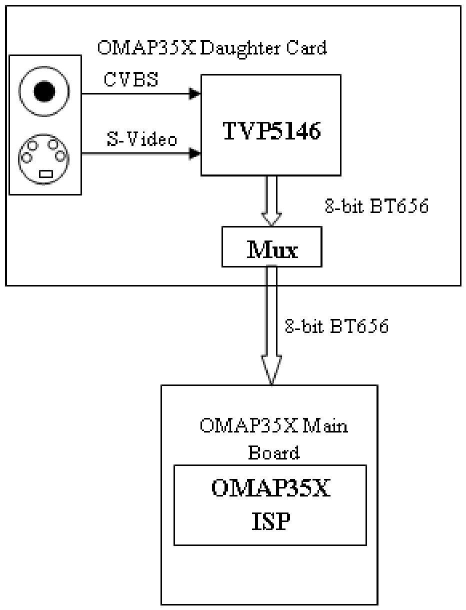


Figure 9.2. Capture Physical Input Interface

The V4L2 Capture driver model is used for capture module. The V4L2 driver model is widely used across many platforms in the Linux community. V4L2 provides good streaming support and support for many buffer formats. It also has its own buffer management mechanism that can be used.

9.1.1. References

1. OMAP35x Camera Interface Subsystem (ISP) TRM

   Author: Texas Instruments, Inc.

   Literature Number: SPRUFA2

2. OMAP35x Memory Management Units (MMUs)TRM
   Author: Texas Instruments, Inc.

   Literature Number: SPRUFF5

3. Video for Linux Two API Specification
   Author: Michael H Schimek

   Version: 0.23

9.1.2. Acronyms & Definitions

Acronym	Definition
MMDC	Mass Market Daughter Card/Customer Daughter Card
3A	Auto White Balance, Auto Focus, Auto Exposure
API	Application Programming Interface
CCDC	Input interface block of ISP
DMA	Direct Memory Access
I/O	Input & Output
IOCTL	Input & Output Control
MMU	Memory Management Unit
V4L2	Video for Linux specification version 2
YUV	Luminance + 2 Chrominance Difference Signals (Y, Cr, Cb) Color Encoding

Table 9.1. Capture Driver Acronyms

9.2. Features

The ISP Capture Driver provides the following features:

- Supports one software channel of capture and a corresponding device node (/dev/video0) is created.
- Supports single I/O instance and multiple control instances.
- Supports buffer access mechanism through memory mapping and user pointers.
- Supports dynamic switching among input interfaces with some necessary restrictions wherever applicable.
- Supports NTSC and PAL standard on Composite and S-Video interfaces.
- Supports 8-bit BT.656 capture in UYVY and YUYV interleaved formats.
- Supports standard V4L2 IOCTLs to get/set various control parameters like brightness, contrast, saturation, hue and auto gain control.
- TVP5146 (TVP514x) decoder driver module can be used statically or dynamically (insmod and rmmod supported).
- In USERPTR mode of operation both malloc'd and IO mapped buffers are supported.
- The camera ISP driver supports both static into kernel and modular build.

9.3. Architecture

9.3.1. System Diagram

Following block diagram shows basic architecture of the ISP Capture Driver.

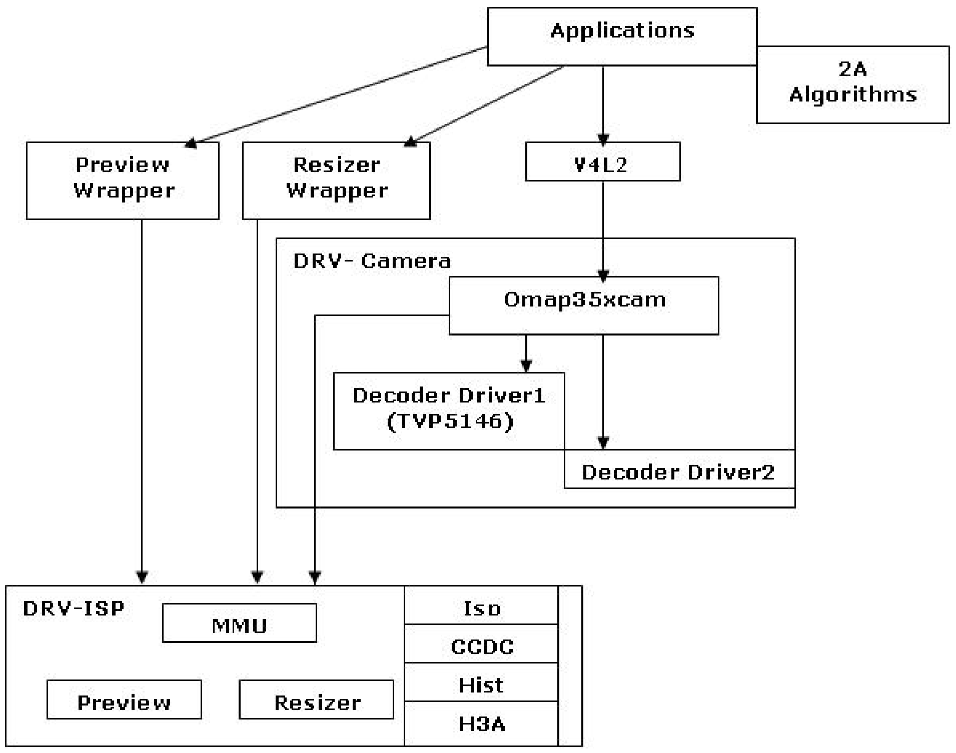


Figure 9.3. Capture Driver Basic Architecture

The system architecture diagram illustrates the software components that are relevant to the Camera Driver. Some components are outside the scope of this design document. The following is a brief description of each component in the figure.

Camera Applications: Camera applications refer to any application that accesses the device node that is served by the Camera Driver. These applications are not in the scope of this design. They are here to present the environment in which the Camera Driver is used.

V4L2 Subsystem: The Linux V4L2 subsystem is used as an infrastructure to support the operation of the Camera Driver. Camera applications mainly use the V4L2 API to access the Camera Driver functionality. A Linux 2.6 V4L2 implementation is used in order to support the standard features that are defined in the V4L2 specification.

Video Buffer Library: This library comes with V4L2. It provides helper functions to cleanly manage the video buffers through a video buffer queue object.

Camera Driver: The Camera Driver allows capturing video through an external decoder. It is a V4L2-compliant driver with addition of an OMAP3 ISP hardware feature. This driver conforms to the Linux driver model for power management. The camera driver is registered to the V4L2 layer as a master device driver. Any slave decoder driver added to the V4L2 layer will be attached to this driver through the new V4L2 master-slave interface layer. The current implementation supports only one slave device.

Decoder Driver: The Camera Driver is designed to be OMAP dependent, but platform and board independent. It is the decoder driver that manages the board connectivity. A decoder driver must implement the new V4L2 master-slave interface. It should register to the V4L2 layer as a slave device. Changing a decoder requires implementation of a new decoder driver; it does not require changing the Camera Driver. Each decoder driver exports a set of IOCTLs to the master device through function pointers.

ISP Library: The ISP library exports APIs to configure ISP module and clocks to the sensor/decoder. It is the central interrupt handler where callback routines for ISP interrupts are handled. This also manages the video buffers.

CCDC library: CCDC is a HW block in Camera ISP which acts as a data input port. It receives data from the sensor/decoder through parallel or serial interface. The CCDC library exports API to configure CCDC module. It is configured by the ISP driver based on the sensor/decoder attached and desired output from the camera driver.

MMU library: MMU is a HW block in Camera ISP that handles the translation from virtual into physical addresses. The camera subsystem issues virtual addresses to the ISP MMU and the ISP MMU translates these virtual addresses into physical addresses to access the actual memory. Using this the camera driver captures video data in fragmented physical memory without moving data. The MMU library exports API to configure MMU module.

Preview library: Preview is a HW block in Camera ISP which is responsible for image processing and color conversion. It has HW blocks for image processing algorithms. Preview library allows camera driver to configure, enable and disable the individual HW blocks in the preview module. This module will be used only when a RAW sensor is connected to the ISP.

Resizer library: Resizer is a HW block in Camera ISP which is responsible for image downscaling and upscaling. It has HW filters which resize the input image based on configuration. Resizer library allows camera driver to query and configure the resizer module. Resizer in OMAP3 ISP supports resizing ratios from 1/4 to 4. Resizer also has multipass approach which can be used to overcome this limitation. Current camera driver only supports on the fly mode of operation. In this mode image is taken from sensor and passed to application without any memory to memory operations in ISP and so multipass resizer operations are not supported.

H3A library: H3A is a HW block in Camera ISP which is responsible for collecting image statistics that can be used by other algorithms. It generates auto focus, auto white balance, auto exposure and histogram statistics. H3A library allows user space algorithms to configure and request these statistics through custom IOCTLs.

9.3.2. Software Design Interfaces

9.3.2.1. Opening and Closing of driver

The device can be opened using open call from the application, with the device name and mode of operation as parameters. Application should open the driver in blocking mode. In this mode, DQBUF IOCTL will not return until an empty frame is available.

```
/* Open a video capture logical channel in blocking mode */
fd = open("/dev/video0", O_RDWR);
if (fd == -1) {
 perror("failed to open Capture device\n");
 return -1;
}
/* closing of channel */
close (fd);
```

9.3.2.2. Buffer Management

ISP Capture driver can work with physically non-contiguous buffers. It uses the ISP MMU to capture data to buffers scattered to a set of page frames. Hence, in user pointer mode the application can allocate buffers in user space, which need not be physically contiguous, and pass this directly to driver for capture operation. The only restriction for the user buffer is that, the buffer should be aligned to 32 bytes boundary. The driver supports both memory usage modes:

1) Memory map buffer mode

2) User Pointer mode

In Memory map buffer mode, application can request memory from the driver by calling `VIDIOC_REQBUFS` IOCTL. In user buffer mode, application needs to allocate memory using some other mechanism in user space like `malloc` or `memalign`. In driver buffer mode, maximum number of buffers is limited to `VIDEO_MAX_FRAME` (defined in driver header files) and is limited by the available memory in the kernel.

The main steps that the application must perform for buffer allocation are:

1) Allocating Memory

2) Getting Physical Address

3) Mapping Kernel Space Address to User Space

1. Allocating Memory

This IOCTL is used to allocate memory for frame buffers. This is the necessary IOCTL for streaming IO. It has to be called for both driver buffer mode and user buffer mode. Using this IOCTL, driver will know whether driver buffer mode or user buffer mode will be used.

Ioctl: VIDIOC_REQBUFS

It takes a pointer to instance of `v4l2_requestbuffers` structure as an argument.

User should specify buffer type as (`V4L2_BUF_TYPE_VIDEO_CAPTURE`), number of buffers, and memory type (`V4L2_MEMORY_MMAP`, `V4L2_MEMORY_USERPTR`) at the time of buffer allocation.

Constraint: This IOCTL can be called only once from the application. This IOCTL is necessary IOCTL.

Example:

```
/* structure to store buffer request parameters */
struct v4l2_requestbuffers reqbuf;

reqbuf.count = numbuffers;
reqbuf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
reqbuf.memory = V4L2_MEMORY_MMAP;
ret = ioctl(fd, VIDIOC_REQBUFS, &reqbuf);
if (ret < 0) {
    printf("cannot allocate memory\n");
    close(fd);
    return -1;
}

printf("Number of buffers allocated = %d\n", reqbuf.count);
```

2. Getting Physical Address

This IOCTL is used to query buffer information like buffer size and buffer physical address. This physical address is used in mmapping the buffers. This IOCTL is necessary for driver buffer mode as it provides the physical address of buffers, which are used to mmap system call the buffers.

Ioctl: VIDIOC_QUERYBUF

It takes a pointer to instance of `v4l2_buffer` structure as an argument.

User has to specify buffer type as (`V4L2_BUF_TYPE_VIDEO_CAPTURE`), buffer index, and memory type (`V4L2_MEMORY_MMAP`) at the time of querying.

Example:

```
/* allocate buffer by VIDIOC_REQBUFS */
/* structure to query the physical address of allocated buffer */
struct v4l2_buffer buffer;

buffer.index = 0; /* buffer index for quering -0 */
buffer.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buffer.memory = V4L2_MEMORY_MMAP;
if (ioctl(fd, VIDIOC_QUERYBUF, &buffer) < -1) {
    printf("buffer query error.\n");
    close(fd);
    exit(-1);
}


The buffer.m.offset will contain the physical address returned
 from driver.
```

3. Mapping Kernel Space Address to User Space

Mapping the kernel buffer to the user space can be done via mmap. This is only required for MMAP buffer mode. User can pass buffer size and physical address of buffer for getting the user space address.

Example:

```
/* allocate buffer by VIDIOC_REQBUFS */
/* query the buffer using VIDIOC_QUERYBUF */
/* addr hold the user space address */
int addr;
addr = mmap(NULL, buffer.size,PROT_READ | PROT_WRITE,
            MAP_SHARED, fd, buffer.m.offset);


/* buffer.m.offset is same as returned from VIDIOC_QUERYBUF */
```

9.3.2.3. Query Capabilities

This IOCTL is used to verify kernel devices compatibility with V4L2 specification and to obtain information about individual hardware capabilities. In this case, it will return capabilities provided by ISP capture driver and current decoder driver.

Ioctl: VIDIOC_QUERYCAP

Capabilities can be video capture (`V4L2_CAP_VIDEO_CAPTURE`) and streaming (`V4L2_CAP_STREAMING`).

It takes pointer to `v4l2_capability` structure as an argument.

Capabilities can be accessed by capabilities field in the `v4l2_capability` structure.

Example:

```
struct v4l2_capability capability;

ret = ioctl(fd, VIDIOC_QUERYCAP, &capability);
if (ret < 0) {
    printf("Cannot do QUERYCAP\n");
    return -1;
}

if (capability.capabilities & V4L2_CAP_VIDEO_CAPTURE) {
    printf("Capture capability is supported\n");
}
if (capability.capabilities & V4L2_CAP_STREAMING) {
    printf("Streaming is supported\n");
}
```

9.3.2.4. Input Enumeration

This IOCTL is used to enumerate the information of available inputs (analog interface). It includes information like name of input type and supported standards for that input type.

Ioctl: VIDIOC_ENUMINPUT

It takes pointer to `v4l2_input` structure. Application provides the index number for which it requires the information, in index member of `v4l2_input` structure.

Index with value zero indicates first input type of the decoder. It returns combination of the standards supported on this input in the std member of `v4l2_input` structure.

Example:

```
struct v4l2_input input;

i = 0;
while(1) {
    input.index = i;
    ret = ioctl(fd, VIDIOC_ENUMINPUT, &input);
    if (ret < 0)
        break;

    printf("name = %s\n", input.name);
    i++;
}
```

9.3.2.5. Set Input

This IOCTL is used to set input type (analog interface type).

Ioctl: VIDIOC_S_INPUT

This IOCTL takes pointer to integer containing index of the input which has to be set.

Application will provide the index number as an argument.

```
        0 - Composite input,
        1 - S-Video input.
```

Example:

```
int index = 1; /*To set S-Video input*/
struct v4l2_input input;

ret = ioctl(fd, VIDIOC_S_INPUT, &index);
if (ret < 0) {
    perror("VIDIOC_S_INPUT\n");
    close(fd);
    return -1;
}

input.index = index;
ret = ioctl(fd, VIDIOC_ENUMINPUT, &input);
if (ret < 0) {
    perror("VIDIOC_ENUMINPUT\n");
    close(fd);
    return -1;
}

printf("name of the input = %s\n",input.name);
```

9.3.2.6. Get Input

This IOCTL is used to get the current input type (analog interface type).

Ioctl: VIDIOC_G_INPUT

This IOCTL takes pointer to integer using which the detected inputs will be returned. It will return the software managed input detected during open system call.

Application will provide the index number as an output argument.

Example:

```
int input;
struct v4l2_input input;

ret = ioctl(fd, VIDIOC_G_INPUT, &input);
if (ret < 0) {
    perror("VIDIOC_G_INPUT\n");
    close(fd);
    return -1;
}

input.index = index;
ret = ioctl(fd, VIDIOC_ENUMINPUT, &input);
if (ret < 0) {
    perror("VIDIOC_ENUMINPUT\n");
    close(fd);
    return -1;
}

printf("name of the input = %s\n", input.name);
```

9.3.2.7. Standard Enumeration

This IOCTL is used to enumerate the information regarding video standards.

This IOCTL is used to enumerate all the standards supported by the registered decoder.

Ioctl: VIDIOC_ENUMSTD

This IOCTL takes a pointer to `v4l2_standard` structure. Application provides the index of the standard to be enumerated in the index member of this structure. It provides information like standard name, standard ID defined at V4L2 header files (few new standards are included in the respective decoder header files, which were not available in standard V4L2 header files), and numerator and denominator values for frame period and frame lines.

It takes index as an argument as a part of `v4l2_standard` structure.

Index with value zero provides information for the first standard among all the standards of all the registered decoders.

If the index value exceeds the number of supported standards, it returns an error.

Example:

```
struct v4l2_standard standard;

i = 0;
while(1) {
    standard.index = i;
    ret = ioctl(fd, VIDIOC_ENUMSTD, &standard);
    if (ret < 0)
        break;

    printf("name = %s\n", std.name);
    printf("framelines = %d\n", std.framelines);
    printf("numerator = %d\n",
           std.frameperiod.numerator);
    printf("denominator = %d\n",
           std.frameperiod.denominator);
    i++;
}
```

9.3.2.8. Standard Detection

This IOCTL is used to detect the current video standard set in the current decoder.

Ioctl: VIDIOC_QUERYSTD

It takes a pointer to `v4l2_std_id` instance as an output argument. Driver will call the current decoder's function internally (which has been initialized) to detect the current standard set in hardware. Support of this IOCTL depends on decoder device, whether it can detect a standard or not.

Note: This IOCTL should be called by the application so that the camera driver can configure ISP properly with the detected decoder standard.

Standard IDs are defined in the V4L2 header files

Example:

```
v4l2_std_id std;
struct v4l2_standard standard;

ret = ioctl(fd, VIDIOC_QUERYSTD, &std);
if (ret < 0) {
    perror("VIDIOC_QUERYSTD\n");
    close(fd);
    return -1;
}

while(1) {
    standard.index = i;
    ret = ioctl(fd, VIDIOC_ENUMSTD, &standard);
    if (ret < 0)
        break;

    if (standard.std & std) {
        printf("%s standard detected\n",
               standard.name);
        break;
    }
    i++;
}
```

9.3.2.9. Set Standard

This IOCTL is used to set the standard in the decoder.

Ioctl: VIDIOC_S_STD

It takes a pointer to `v4l2_std_id` instance as an input argument. If the standard is not supported by the decoder, the driver will return an error

Standard IDs are defined in the V4L2 header files (few new standards are included in respective decoder header files, which were not available in standard V4L2 header files).

Note: Application need not call this IOCTL as the decoder can auto detect the current standard. This is required only when the application needs to set a particular standard. In this case, the decoder driver auto detect function is disabled. Auto detect can be enabled again only by closing and re-opening the driver.

Example:

```
v4l2_std_id std = V4L2_STD_NTSC;

ret = ioctl(fd, VIDIOC_S_STD, &std);
if (ret < 0) {
    perror("S_STD\n");
    close(fd);
    return -1;
}

while(1) {
    standard.index = i;
    ret = ioctl(fd, VIDIOC_ENUMSTD, &standard);
    if (ret < 0)
        break;

    if (standard.std & std) {
        printf("%s standard is selected\n");
        break;
    }
    i++;
}
```

9.3.2.10. Get Standard

This IOCTL is used to get the current standard in the current decoder.

Ioctl: VIDIOC_G_STD

It takes a pointer to `v4l2_std_id` instance as an output argument.

Standard IDs are defined in the V4L2 header files

Example:

```
v4l2_std_id std;

ret = ioctl(fd, VIDIOC_G_STD, &std);
if (ret < 0) {
    perror("G_STD\n");
    close(fd);
    return -1;
}

while(1) {
    standard.index = i;
    ret = ioctl(fd, VIDIOC_ENUMSTD, &standard);
    if (ret < 0)
        break;

    if (standard.std & std) {
         printf("%s standard is selected\n");
         break;
    }
    i++;
}
```

9.3.2.11. Format Enumeration

This IOCTL is used to enumerate the information of pixel formats. The driver supports only two pixel form at -8-bit UYVY interleaved and 8-bit YUYV interleaved.

Ioctl: VIDIOC_ENUM_FMT

It takes a pointer to instance of `v4l2_fmtdesc` structure as an output parameter.

Application must provide the buffer type in the type argument of `v4l2_fmtdesc` structure as `V4L2_BUF_TYPE_VIDEO_CAPTURE` and index member of this structure as zero.

Example:

```
struct v4l2_fmtdesc fmt;

i = 0;
while(1) {
    fmt.index = i;
    ret = ioctl(fd, VIDIOC_ENUM_FMT, &fmt);
    if (ret < 0)
        break;

    printf("description = %s\n",fmt.description);
    if (fmt.type == V4L2_BUF_TYPE_VIDEO_CAPTURE)
        printf("Video capture type\n");
    if (fmt.pixelformat == V4L2_PIX_FMT_YUYV)
        printf("V4L2_PIX_FMT_YUYV\n");
    i++;
}
```

9.3.2.12. Set Format

This IOCTL is used to set the format parameters. The format parameters are line offset, storage format, pixel format, and so on. This IOCTL is one of the necessary IOCTL. If it is not set, it uses the following default values:

- Default storage format - V4L2_FIELD_INTERLACED

This IOCTL expects proper width and height members of the `v4l2_format` structure from application as per the standard selected.

Please note that, `V4L2_FIELD_INTERLACED` is the only storage format supported.

The application can decide the buffer pixel format using pixelformat member of this IOCTL. The current driver supports - 8-bit UYVY interleaved and 8-bit YUYV interleaved formats.

The desired pitch of the buffer can be set by using the bytesperline member. The pitch should be at least one line size in bytes. When changing the pitch, the application should also modify the sizeimage member accordingly - sizeimage should be at least pitch * image height.

The driver allocates buffer of size sizeimage member of the `v4l2_format` structure passed through this IOCTL for both mmap buffer and user pointer mode. Driver validates the provided buffer size along with the other members and uses this buffer size for calculating offsets for storing video data.

This IOCTL is a necessary IOCTL for the user buffer mode because driver will know the buffer size for user buffer mode. If it not called for the user buffer mode, driver assumes the default buffer size and calculates offsets accordingly.

Ioctl: VIDIOC_S_FMT

It will take pointer to instance of v4l2_format structure as an input parameter.

If the type member is `V4L2_BUF_TYPE_VIDEO_CAPTURE`, it checks pixel format, pitch value, and image size. It returns an error, if the parameters are invalid.

Example:

```
struct v4l2_format fmt;

fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;
/* for  NTSC standard */
fmt.fmt.pix.width = 720;
fmt.fmt.pix.height = 480;
fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if (ret < 0) {
    perror("VIDIOC_S_FMT\n");
    close(fd);
    return -1;
}
```

9.3.2.13. Get Format

This IOCTL is used to get the current format parameters.

Ioctl: VIDIOC_G_FMT

It takes a pointer to instance of `v4l2_format` structure as an input parameter.

Driver provides format parameters in the structure pointer passed as an argument.

`v4l2_format` structure contains parameters like pixel format, image size, bytes per line, and field type.

For type `V4L2_BUF_TYPE_VIDEO_CAPTURE`, the `v4l2_pix_format` structure of fmt union is filled.

Example:

```
struct v4l2_format fmt;

fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
ret = ioctl(fd, VIDIOC_G_FMT, &fmt);
if (ret < 0) {
    perror("VIDIOC_G_FMT\n");
    close(fd);
    return -1;
}

if (fmt.fmt.pix.pixelformat == V4L2_PIX_FMT_YUYV)
    printf("8-bit UYVY pixel format\n");

printf("Size of the buffer = %d\n", fmt.fmt.pix.sizeimage);
printf("Line offset = %d\n", fmt.fmt.pix.bytesperline);

if (fmt.fmt.pix.field == V4L2_FIELD_INTERLACED)
    printf("Storate format is interlaced frame format");
```

9.3.2.14. Try Format

This IOCTL is used to validate the format parameters provided by the application. It checks parameters and returns the correct parameter, if any parameter is incorrect. It returns error only if the parameters passed are ambiguous.

Ioctl: VIDIOC_TRY_FMT

It takes a pointer to instance of v4l2_format structure as an input/output parameter

If the type member is V4L2_BUF_TYPE_VIDEO_CAPTURE, it checks pixel format, pitch value, and image size. It returns errors to the application, if the parameters are invalid.

Example:

```
struct v4l2_format fmt;

fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;
fmt.fmt.pix.sizeimage = size;
fmt.fmt.pix.bytesperline = pitch;
fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
ret = ioctl(fd, VIDIOC_TRY_FMT, &fmt);
if (ret < 0) {
    perror("VIDIOC_TRY_FMT\n");
    close(fd);
    return -1;
}
```

9.3.2.15. Query Control

This IOCTL is used to get the information of controls that is, brightness, contrast, and so on supported by the current decoder.

Ioctl: VIDIOC_QUERYCTRL

This IOCTL takes a pointer to the instance of `v4l2_queryctrl` structure as the argument and returns the control information in the same pointer. Application provides the control ID in the `v4l2_queryctrl` id member in this structure. This control ID is defined in V4L2 header file, for which information is needed.

If the control command specified by Id is not supported in current decoder, driver will return an error.

Example:

```
struct v4l2_queryctrl ctrl;
```

```
ctrl.id = V4L2_CID_CONTRAST;
ret = ioctl(fd, VIDIOC_QUERYCTRL, &ctrl);
if (ret < 0) {
    perror("VIDIOC_QUERYCTRL \n");
    close(fd);
    return -1;
}

printf("name = %s\n", ctrl.name);
printf("min = %d max = %d step = %d default = %d\n",
       ctrl.minimum, ctrl.maximum, ctrl.step, ctrl.default_value);
```

9.3.2.16. Set Control

This IOCTL is used to set the value for a particular control in current decoder. To set the control value, this IOCTL can also be called when streaming is on.

Ioctl: VIDIOC_S_CTRL

It takes a pointer to instance of `v4l2_control` structure as an input parameter.

Application provides control ID and control values in the `v4l2_control` id and value member in this structure. If the control command specified by Id is not supported in the current decoder and if value of the control is out of range, driver returns an error. Otherwise, it sets the control in the registers.

Example:

```
struct v4l2_control ctrl;

ctrl.id = V4L2_CID_CONTRAST;
ctrl.value = 100;
ret = ioctl(fd, VIDIOC_S_CTRL, &ctrl);
if (ret < 0) {
    perror("VIDIOC_S_CTRL\n");
    close(fd);
    return -1;
}
```

9.3.2.17. Get Control

This IOCTL is used to get the value for a particular control in the current decoder.

Ioctl: VIDIOC_G_CTRL

It takes a pointer to instance of `v4l2_control` structure as an output parameter. Application provides the control ID of id member in this structure. If the control command specified by Id is not supported in the current decoder, driver returns an error. Otherwise, it returns the value of the control in the value member of the `v4l2_control` structure.

Example:

```
struct v4l2_control ctrl;

ctrl.id = V4L2_CID_CONTRAST;
ret = ioctl(fd, VIDIOC_G_CTRL, &ctrl);
if (ret < 0) {
    perror("VIDIOC_G_CTRL\n");
    close(fd);
    return -1;
}
printf("value = %x\n", ctrl.value);
```

9.3.2.18. Queue Buffer

This IOCTL is used to enqueue the buffer in buffer queue. This IOCTL will enqueue an empty buffer in the driver buffer queue. This IOCTL is one of necessary IOCTL for streaming IO. If no buffer is enqueued before starting streaming, driver returns an error as there is no buffer available. So at least one buffer must be enqueued before starting streaming. This IOCTL is also used to enqueue empty buffers after streaming is started.

Ioctl: VIDIOC_QBUF

This IOCTL takes a pointer to instance of `v4l2_buffer` structure as an argument. Application has to specify the buffer type (`V4L2_BUF_TYPE_VIDEO_CAPTURE`), buffer index, and memory type (`V4L2_MEMORY_MMAP or V4L2_MEMORY_USERPTR`) at the time of queuing. For the user pointer buffer exchange mechanism, application also has to provide buffer pointer in the m.userptr member of `v4l2_buffer` structure.

Driver will enqueue buffer in the driver's incoming queue.

It will take pointer to instance of v4l2_ buffer structure as an input parameter.

Example:

```
struct v4l2_buffer buf;

buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.type = V4L2_MEMORY_MMAP;
buf.index = 0;
ret = ioctl(fd, VIDIOC_QBUF, &buf);
if (ret < 0) {
    perror("VIDIOC_QBUF\n");
    close(fd);
    return -1;
}
```

9.3.2.19. Dequeue Buffer

This IOCTL is used to dequeue the buffer in the buffer queue. This IOCTL will dequeue the captured buffer from buffer queue of the driver. This IOCTL is one of necessary IOCTL for the streaming IO. This IOCTL can be used only after streaming is started. This IOCTL will block until an empty buffer is available.

Note: The application can dequeue all buffers from the driver - the driver will not hold the last buffer to itself. In this case, the driver will disable the capture operation and the capture operation resumes when a buffer is queued to the driver again.

Ioctl: VIDIOC_DQBUF

It takes a pointer to instance of v4l2_buffer structure as an output parameter.

Application has to specify the buffer type (`V4L2_BUF_TYPE_VIDEO_CAPTURE`) and memory type (`V4L2_MEMORY_MMAP` or `V4L2_MEMORY_USERPTR`) at the time of dequeueing.

If this IOCTL is called with the file descriptor, with which `VIDIOC_REQBUF` is not performed, driver will return an error.

Driver will enqueue buffer, if the buffer queue is not empty.

Example:

```
struct v4l2_buffer buf;

buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.type = V4L2_MEMORY_MMAP;
ret = ioctl(fd, VIDIOC_DQBUF, &buf);
if (ret < 0) {
    perror("VIDIOC_DQBUF\n");
    close(fd);
    return -1;
}
```

9.3.2.20. Stream On

This IOCTL is used to start video capture functionality.

Ioctl: VIDIOC_STREAMON

If streaming is already started, this IOCTL call returns an error.

Example:

```
v4l2_buf_type buftype = V4L2_BUF_TYPE_VIDEO_CAPTURE;
ret = ioctl(fd, VIDIOC_STREAMON, &buftype);
if (ret < 0) {
    perror("VIDIOC_STREAMON \n");
    close(fd);
    return -1;
}
```

9.3.2.21. Stream Off

This IOCTL is used to stop video capture functionality.

Ioctl: VIDIOC_STREAMOFF

If streaming is not started, this IOCTL call returns an error.

Example:

```
v4l2_buf_type buftype = V4L2_BUF_TYPE_VIDEO_CAPTURE;
ret = ioctl(fd, VIDIOC_STREAMOFF, &buftype);
if (ret < 0) {
    perror("VIDIOC_STREAMOFF \n");
    close(fd);
    return -1;
}
```

9.4. Driver Configuration

9.4.1. Configuration Steps

To enable capture driver support in the kernel, start *Linux Kernel Configuration* tool.

```
$ make menuconfig  ARCH=arm
```

Select *Device Drivers* from the main menu.

```
    ...
    ...
    Kernel Features  --->
    Boot options  --->
    CPU Power Management  --->
    Floating point emulation  --->
    Userspace binary formats  --->
    Power management options  --->
[*] Networking support  --->
    Device Drivers  --->
    ...
    ...
```

Select *Multimedia support* from the menu.

```
    ...
    ...
    Sonics Silicon Backplane  --->
    Multifunction device drivers  --->
[*] Voltage and Current Regulator Support  --->
<*> Multimedia support  --->
    Graphics support  --->
<*> Sound card support  --->
[*] HID Devices  --->
[*] USB support  --->
    ...
    ...
```

Select *Video For Linux* from the menu.

```
    ...
    ...
      *** Multimedia core support ***
<*>   Video For Linux
```

```
[*]     Enable Video For Linux API 1 (DEPRECATED)
< >   DVB for Linux
    ...
    ...
```

Select *Video capture adapters* from the same menu. Press <ENTER> to enter the corresponding sub-menu.

```
    ...
    ...
[ ]   Customize analog and hybrid tuner modules to build  --->
[*]   Video capture adapters  --->
[ ]   Radio Adapters  --->
[ ]   DAB adapters
    ...
    ...
```

Select *OMAP ISP Resizer* from the menu.

```
    ...
    ...
< >   SAA5249 Teletext processor
<*>   OMAP 3 Camera support
< >   OMAP ISP Previewer
<*>   OMAP ISP Resizer
    ...
    ...
```

De-Select *Autoselect pertinent encoders/decoders and other helper chips* from the same menu option. After De-selecting this option, new option *Encoders/decoders and other helper chips* will drop down.

```
    ...
    --- Video capture adapters
[ ]   Enable advanced debug functionality
[ ]   Enable old-style fixed minor ranges for video devices
[ ]   Autoselect pertinent encoders/decoders and other helper
 chips
        Encoders/decoders and other helper chips  --->
< >   Virtual Video Driver
< >   CPiA Video For Linux
```

Go inside option *Encoders/decoders and other helper chips*.

```
    ...
    --- Video capture adapters
[ ]   Enable advanced debug functionality
[ ]   Enable old-style fixed minor ranges for video devices
[ ]   Autoselect pertinent encoders/decoders and other helper
 chips
        Encoders/decoders and other helper chips --->
< >   Virtual Video Driver
< >   CPiA Video For Linux
```

Select TVP514x Video decoder driver from the menu.

```
    ...
    ...
< > Philips SAA7171/3/4 audio/video decoders
< > Philips SAA7191 video decoder
<*> Texas Instruments TVP514x video decoder
< > Texas Instruments TVP5150 video decoder
   ...
   ...
```

9.4.2. Installation

Note

Please note that the software detects and configures the peripherals dynamically/run-time depending on EVM revision. In case of OMAP3EVM-1 (<Rev-E) it configures the peripherals on MMDC and in case of OMAP3EVM-2 (>=Rev-E) it configures On-board peripherals.

9.4.2.1. Driver built statically

If the OMAP35x Camera driver and TVP514x driver are built statically into the kernel, it is activated during boot-up. There is no special procedure to install the driver.

9.4.2.2. Driver built as loadable module

The OMAP35x Camera driver and OMAP35x daughter card (applicable for OMAP3EVM-1 (<Rev-E)) driver cannot be build as a loadable module. both the TVP514x driver and Capture master driver can be build as a module. If the driver has been configured to be a loadable module, then the driver is built as a module with the name tvp514x.ko and omap34xxcam.ko, which will be placed under the directory `drivers/media/video` in the kernel tree.

Copy this driver file on to the target board and issue the following command to insert the driver:

```
$ insmod omap34xxcam.ko
$ insmod tvp514x.ko
```

To remove the driver, issue the following command:

```
$ code>rmmod omap34xxcam.ko
$ code>rmmod tvp514x.ko
```

9.5. Sample Applications

This chapter describes the sample application provided along with the package. The binary and the source for these sample application can are available in the Examples directory of the Release Package folder.

9.5.1. Introduction

Writing a capture application involves the following steps:

- Opening the capture device.
- Set the parameters of the device.
- Allocate and initialize capture buffer
- Receive video data from the device.
- Close the device.

9.5.2. Hardware Setup

Following are the steps required to run the capture sample application:

- If you are using OMAP3EVM-1 (<Rev-E) revision board, connect the OMAP35x daughter card module containing the TVP5146 decoder to the OMAP35x main board. For OMAP3EVM-2 (>=Rev-E), all the peripherals including TVP5146 decoder is present on board.
- Connect a DVD player/camera generating a NTSC video signal to the S-Video or Composite jack of the daughter card or EVM.
- Run the sample application after booting the kernel.

9.5.3. Sample Applications

Following are the list of capture sample application provided with the release:

- **MMAP Loopback Application (saMmapLoopback.c):**

  This sample application using driver allocated buffers to capture video data from any one of the active inputs and displays the video in the LCD using display driver.

- **USERPTR Loopback Application (saUserPtrLoopback.c):**

  This sample application using User allocated buffers to capture video data from any one of the active inputs and displays the video in the LCD using display driver. The application makes use of V4L2 display driver buffers as a user pointer in capture driver.

USB Driver

Abstract

This chapter provides detailed description of feature set and software interface for the USB driver.

Table of Contents

10.1. Introduction

TI OMAP35x has a host cum gadget controller MUSB OTG, an EHCI and its companion OHCI controller. There are three USB ports which are to be controlled by either EHCI or OHCI controller individually.

In ES2.0/2.1 silicon all the three port can either be configured in PHY mode or in TLL mode at a time. This limitation got resolved in ES3.0/3.1 silicon where PHY/TLL mode selection can be done on per port basis.

The salient features of the MUSB OTG controller are:

- High/full speed operation as USB peripheral.
- High/full/low speed operation as Host controller.
- The host controller for a multi-point USB system (when connected via hub).
- USB On-The-Go compliant USB controller.
- 15 Transmit and 15 Receive Endpoints other than the mandatory Control Endpoint 0.
- 16 Kilobytes of Endpoint FIFO RAM for USB packet buffering.
- Double buffering FIFO.
- Support for Bulk split and Bulk combine
- Support for high bandwidth Isochronous transfer
- Dual Mode HS DMA controller with 8 channels.

10.1.1. References

1. OMAP35x Technical Reference Manual

10.1.2. Hardware Overview

The OMAP35x MUSB OTG controller sits on the L3 and L4 interconnect. It can be an L3 master while performing DMA transfers and an L4 target when host CPU/DMA engine is the master.

OMAP3EVM-1 (<=Rev-E) has an OTG compliant USB PHY from NXP (ISP 1504) and OMAP3EVM-2 (>=Rev-E) has NXP USB PHY ISP1507.

The USB controller in the SoC is connected to the NXP PHY located on the EVM. A mini-AB USB port connects to the PHY. Hence, there is only one root port for the USB controller.

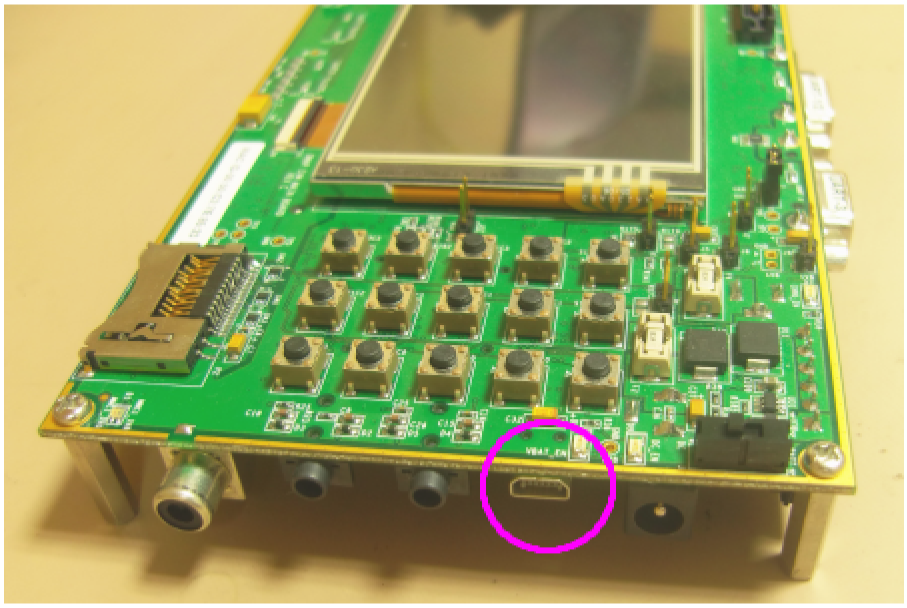


Figure 10.1. MUSB OTG: Location of Mini-AB receptacle on the EVM

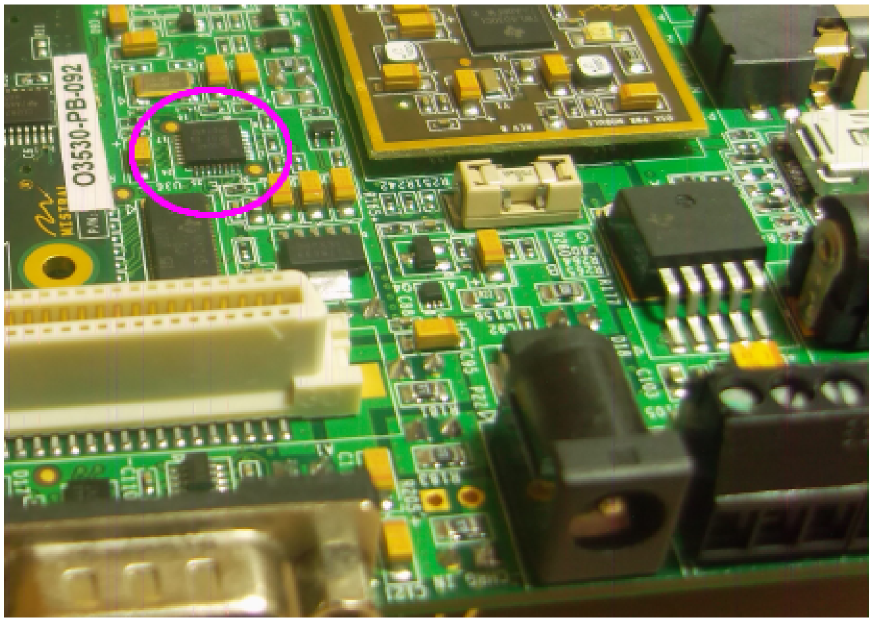


Figure 10.2. MUSB OTG: Location of USB PHY from NXP on the EVM

The OMAP35x HS USB port2 is connected to SMSC USB83320 high speed PHY on Mistral/Multimedia daughter card (MMDC) attached to OMAP3EVM-1 (<=Rev-E) whereas on OMAP3EVM-2 (>=Rev-E) it is connected to SMSC USB3320 PHY.

Port1 and Port3 are not available either on MMDC attached to OMAP3EVM-1 (<=Rev-E) or OMAP3EVM-2 (>=Rev-E).

10.2. Features

The MUSB OTG and EHCI drivers supports a significant subset of all the features provided by the USB controller. The following section discusses the supported features in this release.

The Driver supports the following features for MUSB OTG port:

- Can be built in-kernel (part of vmlinux) as well as a driver module (musb_hdrc.ko).
- Audio Class in Host mode.
- Video Class in Host mode.
- Mass Storage Class in Host mode.
- Mass Storage Class in Gadget mode.
- Hub Class in Host mode.
- Human Interface Devices (HID) in Host mode.
- Communication Device Class (CDC) in Gadget mode.
- Remote Network Driver Interface Specification (RNDIS) Gadget support.
- OTG support which includes support for Host Negotiation Protocol (HNP) and Session Request Protocol (SRP).

The Driver supports the following features for EHCI host port:

- Can be built in-kernel (part of vmlinux) as well as a driver module.
- Human Interface Devices (HID) via a high speed hub.
- Mass Storage Class.
- Audio Class.
- Video Class.
- Hub Class.

10.3. Driver configuration

The MUSB OTG controller is used in Host and Gadget modes while EHCI is used only in Host mode. The following section shows the configuration options for USB and its associated class drivers.

10.3.1. USB phy selection for MUSB OTG port

Please select NOP USB transceiver for MUSB support.

```
Device Drivers --->
  USB support --->
  *** OTG and related infrastructure ***
  [ ] GPIO based peripheral-only VBUS sensing 'transceiver'
  [ ] Philips ISP1301 with OMAP OTG
  [ ] TWL4030 USB Transceiver Driver
  [*] NOP USB Transceiver Driver
```

10.3.2. USB controller in host mode

10.3.2.1. MUSB OTG Host Configuration

```
Device Drivers --->
  USB support --->
  <*> Support for Host-side USB
      *** Miscellaneous USB options ***
  [*] USB device filesystem
  [*] USB device class-devices (DEPRECATED)
      *** USB Host Controller Drivers ***
  <*> Inventra Highspeed Dual Role Controller (TI, ...)
            *** OMAP 343x high speed USB support ***
            Driver Mode (USB Host) --->
  [ ] Disable DMA (always use PIO)
  [*] Use System DMA for Rx endpoints
  [*] Enable debugging messages
```

10.3.2.2. EHCI Configuration

Port-2 will automatically be selected for OMAP3EVM and would be configured in PHY mode.

```
Device Drivers --->
  USB support --->
  <*> Support for Host-side USB
      *** Miscellaneous USB options ***
  [*] USB device filesystem
  [*] USB device class-devices (DEPRECATED)
  <*> EHCI HCD (USB2.0) Support
  [ ] Root hub transaction translators
      *** USB Host Controller Drivers ***
```

10.3.3. MUSB OTG controller in gadget mode

10.3.3.1. Configuration

Please do not disable support for host side usb as this will disable EHCI host interface also. Gadget option in driver mode will appear only when gadget support is also selected. Please enable gadget support as given below.

```
Device Drivers --->
  USB support --->
   <*> USB Gadget Support --->
[ ] Debugging messages (DEVELOPMENT) NEW
[ ] Debugging information files (DEVELOPMENT) NEW
(2) Maximum VBUS power usage (2-500mA) NEW
   USB Peripheral Controller (Inventra HDRC Peripheral(TI, ...))
--->
   <M> USB Gadget Drivers
   <M> File-backed Storage Gadget
```

Please make sure that Inventra HDRC is selected as USB peripheral controller which will appear only when "USB Peripheral (gadget stack)" is selected in driver mode as shown below so after selecting Gadget Support go back to driver mode option to select "USB Peripheral (gadget stack) " and then come back again to select Inventra HDRC as USB peripheral controller.

```
Device Drivers --->
  USB support --->
  <*> Support for Host-side USB
      *** Miscellaneous USB options ***
  [*] USB device filesystem
  [*] USB device class-devices (DEPRECATED)
      *** USB Host Controller Drivers ***
  <*> Inventra Highspeed Dual Role Controller (TI, ...)
            *** OMAP 343x high speed USB support ***
            Driver Mode (USB Peripheral (gadget stack)) --->
  [ ] Disable DMA (always use PIO)
  [*] Use System DMA for Rx endpoints
  [*] Enable debugging messages
```

10.3.4. MUSB OTG controller in OTG mode

10.3.4.1. OTG Configuration

Both Host and Gadget driver should be selected for OTG support. If gadget driver is build as module then the host side module will be initialized only after gadget module is inserted after bootup.

If "Rely on targeted peripheral list" is also selected then make sure to update "drivers/usb/core/otg_whitelist.h" with the desired supported device class identification ids.

OTG option in driver mode will appear only when gadget support is also selected. Please enable gadget support as given below.

```
Device Drivers --->
  USB support --->
   <*> USB Gadget Support --->
[ ] Debugging messages (DEVELOPMENT) NEW
[ ] Debugging information files (DEVELOPMENT) NEW
(2) Maximum VBUS power usage (2-500mA) NEW
   USB Peripheral Controller (Inventra HDRC Peripheral(TI, ...))
--->
   <M> USB Gadget Drivers
   <M> File-backed Storage Gadget
```

Please make sure that Inventra HDRC is selected as USB peripheral controller which will appear only when OTG is selected as below.

```
Device Drivers --->
  USB support --->
  <*> Support for Host-side USB
      *** Miscellaneous USB options ***
  [*] USB device filesystem
  [*] USB device class-devices (DEPRECATED)
      *** USB Host Controller Drivers ***
  <*> Inventra Highspeed Dual Role Controller (TI, ...)
            *** OMAP 343x high speed USB support ***
            Driver Mode (Both Host and peripheral : USB OTG (On
 The Go) Device) --->
  [ ] Disable DMA (always use PIO)
  [*] Use System DMA for Rx endpoints
  [*] Enable debugging messages
```

10.3.5. Host mode applications

10.3.5.1. Mass Storage Driver

This figure illustrates the stack diagram of the system with USB Mass Storage class.

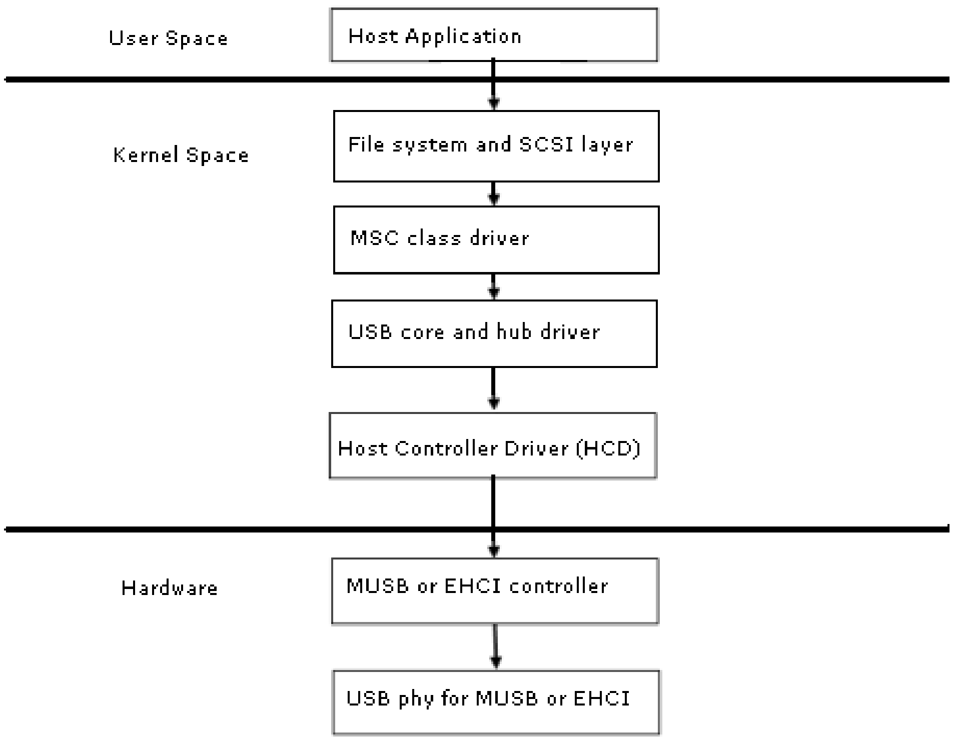


Figure 10.3. USB Driver: Illustration of Mass Storage Class

10.3.6. USB Controller and USB MSC HOST

10.3.6.1. Configuration

```
Device Drivers --->
     SCSI device support --->
      <*> SCSI device support
      [*] legacy /proc/scsi/support
      --- SCSI support type (disk, tape, CD-ROM)
      <*> SCSI disk support
     USB support --->
      <*> Support for Host-side USB
      *** Miscellaneous USB options ***
      [*] USB device filesystem
      [*] USB device class-devices (DEPRECATED)
      *** USB Host Controller Drivers ***
      <*> Inventra Highspeed Dual Role Controller (TI, ...)
            *** OMAP 343x high speed USB support ***
            Driver Mode (USB Host) --->
      [ ] Disable DMA (always use PIO)
      [*] Use System DMA for Rx endpoints
      [*] Enable debugging messages
      --- USB Device Class drivers
      <*> USB Mass Storage support
```

10.3.6.2. Device nodes

The SCSI sub system creates /dev/sd* devices with help of mdev.

10.3.7. USB HID Class

USB Mouse and Keyboards that conform to the USB HID specifications are supported.

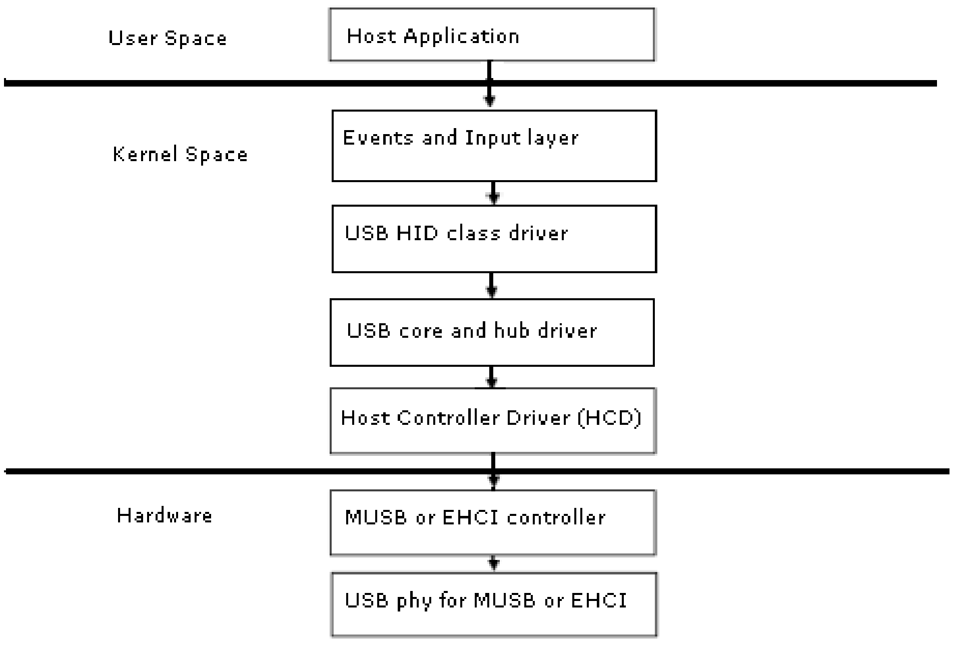


Figure 10.4. USB Driver: Illustration of HID Class

10.3.8. USB Controller and USB HID

10.3.8.1. Configuration

```
Device Drivers --->
     USB support --->
      <*> Support for Host-side USB
      *** Miscellaneous USB options ***
      [*] USB device filesystem
      [*] USB device class-devices (DEPRECATED)
      *** USB Host Controller Drivers ***
      <*> Inventra Highspeed Dual Role Controller (TI, ...)
                *** OMAP 343x high speed USB support ***
                Driver Mode (USB Host) --->
      [ ] Disable DMA (always use PIO)
      [*] Use System DMA for Rx endpoints
      [*] Enable debugging messages
     HID Devices --->
      <*> Generic HID Support
         *** USB Input Devices ***
      <*> USB Human Interface Device(full HID) support
```

10.3.8.2. Device nodes

The event sub system creates /dev/input/event* devices with the help of mdev.

10.3.9. USB Audio

10.3.9.1. Configuration

```
Device Drivers --->
    Sound --->
     <*> Sound card support
       Advanced Linux Sound Architecture --->
       <*> Advanced Linux Sound Architecture
       [*] Dynamic device file minor number
       [*] Support old ALSA API
          USB devices --->
          <*> USB Audio/MIDI driver
    USB support --->
       <*> Support for Host-side USB
       *** Miscellaneous USB options ***
       [*] USB device filesystem
       [*] USB device class-devices (DEPRECATED)
       *** USB Host Controller Drivers ***
       <*> Inventra Highspeed Dual Role Controller (TI, ...)
                *** OMAP 343x high speed USB support ***
                 Driver Mode (USB Host) --->
       [ ] Disable DMA (always use PIO)
       [*] Use System DMA for Rx endpoints
       [*] Enable debugging messages
```

10.3.9.2. Resources

For testing USB Audio support we need any ALSA compliant audio player/ capture application. Kindly read the Audio driver section to get more inputs on this.

10.3.10. USB Video

10.3.10.1. Configuration

```
  Device Drivers --->
    Multimedia devices --->
          *** Multimedia core support ***
      <*> Video for Linux
      [*] Enable Video for Linux API 1 (DEPRICATED)
      [*] Enable Video for Linux API 1 (compatible) layer
          *** Multimedia Drivers ***
      [*] Video capture adapters --->
          [*] V4L USB devices --->
              <*> USB Video Class (UVC)
    USB Support --->
      <*> Support for Host-side USB
      *** Miscellaneous USB options ***
      [*] USB device filesystem
      [*] USB device class-devices (DEPRECATED)
      *** USB Host Controller Drivers ***
      <*> Inventra Highspeed Dual Role Controller (TI, ...)
                *** OMAP 343x high speed USB support ***
                Driver Mode (USB Host) --->
      [ ] Disable DMA (always use PIO)
      [*] Use System DMA for Rx endpoints
      [*] Enable debugging messages
```

10.3.10.2. Resources

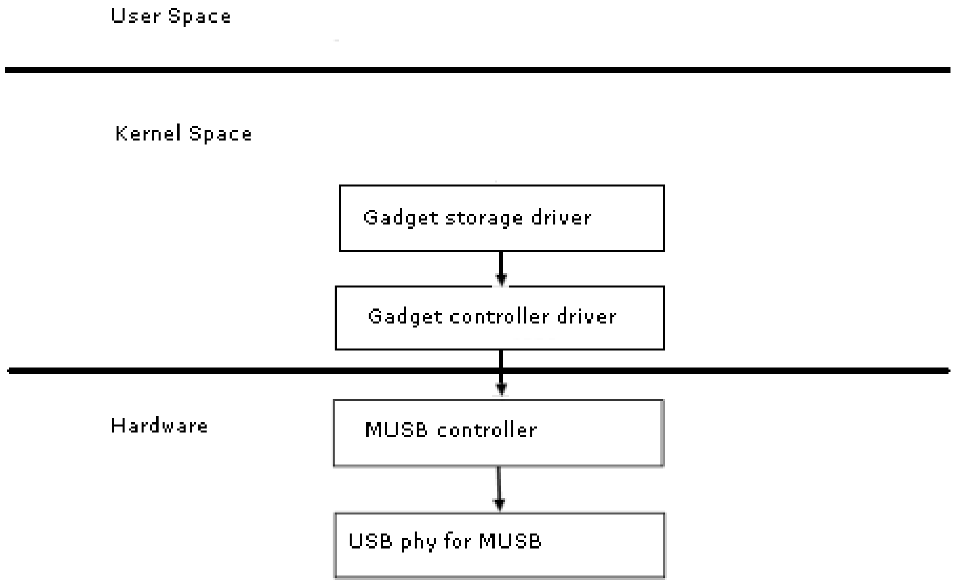
For testing USB Video support we need a user level application like mplayer to stream video from an USB camera.

If you are using mplayer as the capture application, then you must export the DISPLAY to a X server. Then, execute the following command:

```
$ mplayer tv:// -tv driver=v4l2:width=320:height=240
```

10.3.11. Gadget Mode Applications

File Storage Gadget: This is the Mass storage gadget driver.



10.3.11.1. Configuration

```
Device Drivers --->
  USB support --->
   <*> Support for USB Gadgets
   USB Peripheral Controller (Inventra HDRC Peripheral(TI, ...))
--->
   <M> USB Gadget Drivers
   <M> File-backed Storage Gadget

   <*> Inventra Highspeed Dual Role Controller (TI, ...)
         *** OMAP 343x high speed USB support ***
         Driver Mode (USB Peripheral (gadget stack)) --->
   [ ] Disable DMA (always use PIO)
   [*] Use System DMA for Rx endpoints
   [*] Enable debugging messages
```

10.3.11.2. Installation of File Storage Gadget Driver

Let us assume that we are interested in exposing /dev/mmcblk0 block device to the file storage gadget driver. To that effect we need to issue the following command to load the file storage gadget driver.

```
$ insmod <g_file_storage.ko> file=/dev/mmcblk0 stall=0
```

10.3.12. CDC/RNDIS gadget

The CDC RNDIS gadget driver that is used to send standard Ethernet frames using USB. Please enable "Use System DMA for Rx endpoints" to fix the flood ping hang issue with packet size of more than 16KB.

10.3.12.1. Configuration for USB controller and CDC/RNDIS Gadget

```
Device Drivers --->
 USB support --->
 <*> Support for USB Gadgets
 USB Peripheral Controller (Inventra HDRC Peripheral (TI, ...))
 --->
 <M> USB Gadget Drivers
 <M> Ethernet Gadget
 [*]   RNDIS support (EXPERIMENTAL) (NEW)

 <*> Inventra Highspeed Dual Role Controller (TI, ...)
           *** OMAP 343x high speed USB support ***
           Driver Mode (USB Peripheral (gadget stack)) --->
 [ ] Disable DMA (always use PIO)
 [*] Use System DMA for Rx endpoints
 [*] Enable debugging messages
```

Please do not select RNDIS support for testing ethernet gadget with Linux 2.4, IXIA and MACOS host machine.

```
 USB Peripheral Controller (Inventra HDRC Peripheral (TI, ...))
 --->
 <M> USB Gadget Drivers
 <M> Ethernet Gadget
 [ ]   RNDIS support (EXPERIMENTAL) (NEW)
```

10.3.12.2. Installation of CDC/RNDIS Gadget Driver

Installing the CDC/RNDIS gadget driver is as follows:

```
$ insmod  <path to g_ether.ko>
```

10.3.12.3. Setting up USBNet

The CDC/RNDIS Gadget driver will create a Ethernet device by the name usb0. You need to assign an IP address to the device and bring up the device. The typical command for that would be:

```
$ ifconfig usb0  <IP_ADDR> netmask 255.255.255.0 up
```

For details on usage of USBNet, refer this url. [http://embedded.seattle.intel research.net/wiki/index.php?title=Setting_up_USBnet]

10.3.13. USB EHCI Electrical testing

USB EHCI electrical test is supported in software. Please use below command to perform various electrical tests.

```
$ echo 'Options' > sys/devices/platform/ehci-omap.0/portN
```

Where 'options' can be,

- reset --> Reset Device
- t-j --> Send TEST_J on suspended port
- t-k --> Send TEST_K on suspended port
- t-pkt --> Send TEST_PACKET[53] on suspended port
- t-force --> Send TEST_FORCE_ENABLE on suspended port
- t-se0 --> Send TEST_SE0_NAK on suspended port

10.3.14. USB OTG (HNP/SRP) testing

Please choose the configuration as described in driver configuration section for OTG and follow the steps below for testing.

1. Boot the OTG build image on two OMAP35x EVM.
2. If gadget driver is built as module then insert it to complete USB initialization.
3. Connect mini-A side of the OTG cable to one of the EVM (say EVM-1) and mini-B side on the other (say EVM-2).

   In this scenario EVM-1 will become initial host or A-device and EVM-2 will become initial device or B-device. A-device will provide bus power throughout the bus communication even if it becomes peripheral using HNP.

   There will not be any connect event at this point of time as Vbus power is not yet switched-on. Vbus power can be switched-on from A-device or from B-device using SRP.
4. Request to switch-on the Vbus power using below command on any EVM.

```
$ echo "F" > /proc/driver/musb_hdrc
```

If this command is executed on B-device then SRP protocol will be used to request A-device to switch-on the Vbus power.

5. Now the connect event occurs, enumeration will complete and gadget driver on B-device will be ready to use if this driver is in "Targeted Peripheral List (TPL)" of A-device.

   If TPL is disabled on A-device then gadget driver will be ready to use soon after enumeration.

   If TPL is enabled and gadget driver of B-device is not in TPL list of A-device then there will be an automatic trial of HNP from usb core by suspending the bus. This will cause a role switch and B-device will enumerate A-device. Now the gadget driver of A-device will be configured if it is on the TPL list of B-device.

   Currently this is the only way possible for HNP testing but we have added a suspend proc entry to start HNP in other than this scenario.

6. Complete all the communication between A-device and B-device.

7. Start HNP by executing below command on host side.

   ```
   $ echo "S" > /proc/driver/musb_hdrc
   ```

   It will suspend the bus and role-switch will follow after that.

8. Repeat step 4, 5, 6 and 7 for further testing.

10.4. Software Interface

The USB driver exposes its state/control through the sysfs and the procfs interfaces. The following sections talks about these.

10.4.1. sysfs

SYSFS attribute	Description
mode	The entry `/sys/devices/platform/musb_hdrc.0/mode` is a read-only entry. It will show the state of the OTG (though this feature is not supported) state machine. This will be true even if the driver has been compiled without OTG support. Only the states like A_HOST, B_PERIPHERAL, that makes sense for non-OTG will show up.
vbus	The entry `/sys/devices/platform/musb_hdrc.0/vbus` is a write-only entry. It is used to set the VBUS timeout value during OTG. If the current OTG state is a_wait_bcon then then urb submission is disabled.

Table 10.1. USB Driver: sysfs attributes

10.4.2. procfs

The procfs entry `/proc/driver/musb_hdrc` is used to control the driver behaviour as well as check the status of the driver.

The following command will show the usage of this proc entry

```
$ echo "?" > /proc/driver/musb_hdrc
```

Specifically the most important usage of this entry would be to start an USB session(host mode) by issuing the following command:

```
$ echo "F" > /proc/driver/musb_hdrc
```

10.5. Revision history

02.00.00.00	Initial release.
03.00.00.02	Update for OTG and EHCI support.

MMC Driver

Abstract

This chapter provides detailed description of feature set and software interface for the MMC driver.

Table of Contents

11.1. Introduction

TI OMAP 35x has an multimedia card high-speed/secure data/secure digital I/O (MMC/SD/SDIO) host controller, which provides an interface between microprocessor and either MMC, SD memory cards, or SDIO cards. The current version of the user guide talks about the MMC/SD controller. The MMC driver is implemented on top of host controller as a HS-MMC controller driver and supports MMC, SD, SD High Speed and SDHC cards. The salient features of the aforementioned HS-MMC host controller are:

- Full compliance with MMC/SD command/response sets as defined in the Specification.
- Support:
  - 1-bit or 4-bit transfer mode specifications for SD and SDIO cards
  - 1-bit, 4-bit, or 8-bit transfer mode specifications for MMC cards
- Built-in 1024-byte buffer for read or write
- 32-bit-wide access bus to maximize bus throughput
- Single interrupt line for multiple interrupt source events
- Two slave DMA channels (1 for TX, 1 for RX)
- Designed for low power and Programmable clock generation

11.1.1. References

1. MMCA Homepage [http://www.mmca.org/home]
2. SD ORG Homepage [http://www.sdcard.org/home]

11.1.2. Acronyms & Definitions

Acronym	Definition
MMC	Multimedia card
HS-MMC	High Speed MMC
SD	Secure Digital
SDHC	SD High Capacity
SDIO	SD Input/Output

Table 11.1. MMC Driver Acronyms

11.2. Features

The Driver supports the following features:

- The driver is built in-kernel (part of vmlinux).
- MMC cards including High Speed cards.
- SD cards including SD High Speed and SDHC cards.
- Uses block bounce buffer to aggregate scattered blocks

Power Management

Abstract

Get to know the power management infrastructure available. This chapter also provides brief introduction to `cpuidle` framework as implemented in this release.

Table of Contents

12.1. Introduction

OMAP35x silicon provides a rich set of power management features. These features are described in detail in the OMAP35x TRM.

In summary:

- Clock control at the module and clock domain level.
- 16 power domains i.e. 16 sets of one or more hardware modules sharing same power source.
- Control of scalable voltage domains.
- Independent scaling of OPPs for the VDD1 and VDD2.

  MPU and IVA (in case of OMAP3530) share the voltage domain VDD1. Other modules are located in VDD2.
- Support for transitioning power and voltage domains to retention/off and wakeup on event.

12.1.1. References

1. Proceedings of the Linux Symposium, June 27-30, 2007 [http://ols.108.redhat.com/2007/Reprints/pallipadi-Reprint.pdf]

   Authors: Venkatesh Pallipadi, Shaohua Li, Adam Belay
2. OMAP Power Management [http://elinux.org/OMAP_Power_Management]

   Power Management features are being developed on `pm` branch of the `linux-omap` git tree. This page provides latest status of PM features on this branch.

12.2. Features

The power management features available in this release are based on the proposed PM interface for OMAP. This interface is described in the file `Documentation/arm/OMAP/omap_pm` in the Linux kernel sources.

The features supported in this release are:

- Dynamic Tick (NO_HZ) framework.
- The *cpuidle* framework with MPU and Core transition to retention (RET) and OFF states.

  The *menu* governor is supported.
- Static selection of VDD1 OPP via bootarg - `mpurate`.
  - VDD1 OPP can be scaled (if silicon supports) upto 720MHz.
  - When OPP1 is selected for VDD1, the VDD2 is set at OPP2.
- Basic implementation for *cpufreq*.
- Support SmartReflex with automatic (hardware-controlled) mode of operation.

12.3. Architecture

12.3.1. cpuidle

The *cpuidle* framework consists of two key components:

- A governor that decides the target C-state of the system.
- A driver that implements the functions to transition to target C-state.

12.3.1.1. System Diagram

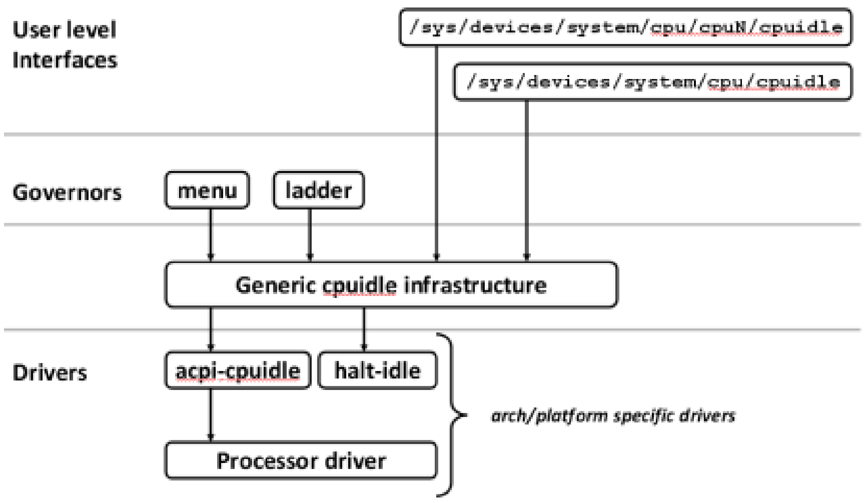

User level Interfaces
/sys/devices/system/cpu/cpuN/cpuidle
/sys/devices/system/cpu/cpuidle
Governors
menu
ladder
Generic cpuidle infrastructure
Drivers
acpi-cpuidle
halt-idle
Processor driver
arch/platform specific drivers

Figure 12.1. cpuidle overview

The idle loop is executed when the Linux scheduler has no thread to run. When the idle loop is executed, current 'governor' is called to decide the target C-state. Governor decides whether to continue in current state/ transition to a different state. Current 'driver' is called to transition to the selected state.

12.3.1.2. C-states

A C-state is used to identify the power state supported through the cpu idle loop. Each C-state is characterized by its:

- Power consumption
- Wakeup latency
- Preservation of processor state while in 'the' state.

The definition of C-states in the OMAP3 are a combination of the MPU and CORE states. Currently these C-states have been defined:

State	Description
C1	MPU WFI + Core active
C2	MPU WFI + Core inactive
C3	MPU RET + Core inactive
C4	MPU OFF + Core inactive
C5	MPU RET + CORE RET
C6	MPU OFF + CORE RET
C7	MPU OFF + CORE OFF

Table 12.1. C-states in OMAP3

12.3.1.3. CPU Idle Governor

The current implementation supports the 'menu' governor to decide the target C-state of the system.

12.3.1.4. CPU Idle Driver

The cpuidle driver registers itself with the framework during boot-up and populates the C-sates with exit latency, target residency (minimum period for which the state should be maintained for it to be useful) and flag to check the bus activity.

In ACPI implementation, flag `CPUIDLE_FLAG_CHECK_BM` is used to specify the states requiring bus monitoring interface to be checked. In the OMAP3 implementation, this flag is used to identify the C-states that require CORE domain activity to be checked.

Once the governor has decided the target C-state, the control reaches the function `omap3_enter_idle()`. Here, the C-state is adjusted based on the value of *valid* flag corresponding to the chosen state.

Note

The value of *valid* flag for the idle states relates to the flag `enable_off_mode`. If transition to OFF mode is disabled, the idle states that require MPU to be turned OFF are made *valid*.

12.3.1.5. Performance considerations

Once idle power management is enabled, the system will transition across sleep states of varying latency. This transition can impact the runtime performance of the drivers.

The flags `sleep_while_idle` and `enable_off_mode` can be used to control the run-time behavior of the *cpuidle* driver. are now accessible via `debugfs`.

Important

In previous kernel versions, these flags were accessible via `sysfs`. In this kernel version, they are accessible via `sysfs`.

In this kernel version, these flags can be accessed via `debugfs`, if configuration options `CONFIG_PM_DEBUG` and `CONFIG_DEBUG_FS`.

See Section 12.4, “Configuration” for steps to enables these configuration options.

12.3.2. Dynamic Tick Suppression

The dynamic tick suppression is achieved through generic Linux framework for the same.

A 32K timer (HZ=128) is used by the tick suppression algorithm.

12.3.3. Suspend & Resume

The suspend operation results in the system transitioning to the lowest power state being supported.

The drivers implement the `suspend()` function defined in the LDM. When the suspend for the system is asserted, the `suspend()` function is called for all drivers. The drivers release the clocks to reach the desired low power state.

The actual transition to suspend is implemented in the function `omap3_pm_suspend()`.

12.4. Configuration

To enable/ disable power management start the *Linux Kernel Configuration* tool.

```
$ make menuconfig
```

Select *Power management options* from the main menu.

```
    ...
    ...
    Boot options  --->
    CPU Power Management  --->
    Floating point emulation  --->
    Userspace binary formats  --->
    Power management options  --->
[*] Networking support  --->
    Device Drivers  --->
    ...
    ...
```

Select *Power Management support* to toggle the power management support.

```
[*] Power Management support
[ ]   Power Management Debug Support
[*] Suspend to RAM and standby
< > Advanced Power Management Emulation
```

12.4.1. cpuidle

Start the *Linux Kernel Configuration* tool.

```
$ make menuconfig
```

Select *CPU Power Management* from the main menu.

```
    ...
    ...
    System Type  --->
    Bus support  --->
    Kernel Features  --->
    Boot options  --->
    CPU Power Management  --->
    Floating point emulation  --->
    Userspace binary formats  --->
    ...
    ...
```

Select *CPU idle PM support* to enable the cpuidle driver.

```
[ ] CPU Frequency scaling
[*] CPU idle PM support
```

12.4.1.1. Enabling debug filesystem

Start the *Linux Kernel Configuration* tool.

```
$ make menuconfig
```

Select *Kernel hacking* from the main menu.

```
    File systems  --->
    Kernel hacking  --->
    Security options  --->
-*- Cryptographic API  --->
```

Select *Debug Filesystem* from the next menu.

```
[ ] Enable unused/obsolete exported symbols
[*] Debug Filesystem
[ ] Run 'make headers_check' when building vmlinux
[*] Kernel debugging
```

12.4.1.2. Debugging support in Power Management

Start the *Linux Kernel Configuration* tool.

```
$ make menuconfig
```

Select *Power management options* from the main menu.

```
    ...
    ...
    Floating point emulation  --->
    Userspace binary formats  --->
    Power management options  --->
[*] Networking support  --->
    Device Drivers  --->
    ...
    ...
```

Select *Power Management support* from the next menu.

```
[*] Power Management support
[*]   Power Management Debug Support
[*] Suspend to RAM and standby
< > Advanced Power Management Emulation
```

12.4.2. cpufreq

Start the *Linux Kernel Configuration* tool.

```
$ make menuconfig
```

Select *CPU Power Management* from the main menu.

```
    ...
    ...
    System Type  --->
    Bus support  --->
    Kernel Features  --->
    Boot options  --->
    CPU Power Management  --->
    Floating point emulation  --->
    Userspace binary formats  --->
    ...
    ...
```

Select *CPU idle PM support* to enable the cpuidle driver.

```
[*] CPU Frequency scaling
[ ] CPU idle PM support
```

12.4.3. SmartReflex

Start the *Linux Kernel Configuration* tool.

```
$ make menuconfig
```

Select *System Type* from the main menu.

```
    ...
    ...
[*] Enable the block layer  --->
    System Type  --->
    Bus support  --->
    Boot options  --->
    CPU Power Management  --->
    ...
    ...
```

Select *TI OMAP Implementations* from the menu.

```
    ARM system type (TI OMAP)  --->
    TI OMAP Implementations  --->
-*- OMAP34xx Based System
-*-   OMAP3430 support
[*] OMAP35x Family
    ...
    ...
```

Select *SmartReflex support* from the menu.

```
    ...
    ...
[ ] Emit debug messages from clockdomain layer
[*] SmartReflex support
[ ]   SmartReflex testing support
    ...
    ...
```

12.5. Software Interface

The *cpuidle* framework defines a standard interface through `/sys` interface.

12.5.1. cpuidle

The parameters controlling *cpuidle* can be viewed via via `/sys` interface.

```
$ ls -1 /sys/devices/system/cpu/cpuidle/
current_driver
current_governor_ro
$
```

current_governor_ro lists the current governor.

```
$ cat /sys/devices/system/cpu/cpuidle/current_governor_ro
menu
$
```

current_driver lists the current driver.

```
$ cat /sys/devices/system/cpu/cpuidle/current_driver
omap3_idle
$
```

The *cpuidle* interface also exports information about each idle state. This information is organized in a directory corresponding to each idle state.

```
$ ls -1 /sys/devices/system/cpu/cpu0/cpuidle
state0
state1
state2
state3
state4
state5
state6
$
```

```
$ ls -1 /sys/devices/system/cpu/cpu0/cpuidle/state0
desc
latency
name
```

```
power
time
usage
```

12.5.1.1. Mounting debug filesystem

To mount the filesystem, execute these commands:

```
$ mkdir /dbg
$ mount -t debugfs debugfs /dbg
```

Note

These commands assume `/dbg` as mount point. Appropriate changes should be made if a different mount point is being used.

12.5.1.2. Idle state transition

To allow/prevent the processor to enter idle states, execute these commands:

```
$ echo 1 > /dbg/pm_debug/sleep_while_idle
$ echo 0 > /dbg/pm_debug/sleep_while_idle
```

To allow/ prevent transition to OFF mode:

```
$ echo 1 > /dbg/pm_debug/enable_off_mode
$ echo 0 > /dbg/pm_debug/enable_off_mode
```

12.5.2. Suspend & Resume

The suspend for device can be asserted as follows:

```
$ echo -n "mem" > /sys/power/state
```

To wakeup, press a key on the OMAP3EVM keypad; or tap any on the serial console.

12.5.3. SmartReflex

To enable/ disable SmartReflex for VDD1:

```
$ echo 1 > /sys/power/sr_vdd1_autocomp
$ echo 0 > /sys/power/sr_vdd1_autocomp
```

To enable/ disable SmartReflex for VDD2:

```
$ echo 1 > /sys/power/sr_vdd2_autocomp
$ echo 0 > /sys/power/sr_vdd2_autocomp
```

12.6. Revision History

02.00.00	Initial version for this GIT based release.
02.01.00	Updated C-state definition.
02.01.01	Updated C-state definition. Moved configuration information from DataSheet Release specific updates.
03.00.00	Updated information based on the latest kernel and features supported.

Power Management IC

Abstract

This chapter provides details on how to configure the PMIC driver, its interfaces and a simple application code illustrating the use of this interface.

Table of Contents

13.1. Introduction

A Power Management IC (PMIC) is a device that contains one or more regulators (voltage or current) and often contains other susbsystems like audio codec, keypad etc. From the power management perspective, its main function is to regulate the output power from input power or simply enable/disable the output as and when required.

A PMIC can have two different types of regulators: voltage regulator or current regulator. Voltage regulators are used to enable/disable the output voltage and/ or regulate the output voltage. Current regulators preform the same functions with the output current. PMICs from TI contain two different types of voltage regulators:

- DCDC: Highly efficient and self-regulating step down DC to DC converter.
- LDO (low-dropout): DC linear voltage regulator which can operate with a very small input-output differential voltage.

The PMIC is controlled by internal registers that can be accessed by the I2C control interface.

This user manual defines and describes the usage of user level and platform level interfaces of the PMIC consumer driver.

13.1.1. References

1. Linux Voltage and Current Regulator Framework [http://opensource.wolfsonmicro.com/node/15]
2. Linux kernel documentation: Documentation/power/regulator and Documentation/ABI/testing/sysfs-class-regulator
3. TPS65950: Integrated Power Management IC with 3 DC/DC's, 11 LDO's, Audio Codec, USB HS Transceiver, Charger
   TPS65950 [http://focus.ti.com/docs/prod/folders/print/tps65950.html]
4. TPS65023 - 6-channel Power Mgmt IC with 3DC/DCs, 3 LDOs, I2C Interface and DVS, Optimized for DaVinci DSPs

   Literature Number: SLVS670G
   TPS65023 [http://focus.ti.com/docs/prod/folders/print/tps65023.html]
5. TPS65073 - 5-Channel Power Management IC with 3 DC/DCs, 2 LDOs in 6x6mm QFN
   TPS65073 [http://focus.ti.com/docs/prod/folders/print/tps65073.html]

13.1.2. Acronyms & Definitions

Acronym	**Definition**
PMIC	Power Management IC
VRF	Voltage Regulator Framework
LDO	Low Drop-Out

Table 13.1. PMIC Driver: Acronyms

13.2. Features

This section describes the supported features and constraints of the PMIC drivers.

13.2.1. Features Supported

- Support for TPS65950, TPS65023 and TPS65073 PMICs in Linux Voltage Regulator Framework.
- Enabling / disabling of voltage regulators.
- Changing the output voltage, if permitted by the voltage regulator.
- Reading the status (enabled/disabled) of the voltage regulator.
- Reading other useful information about the regulator via sysfs interface.

13.2.2. Constraints

None

13.3. Configuration

To enable/disable VRF support, start the *Linux Kernel Configuration* tool.

```
$ make menuconfig
```

Check whether I2C support is enabled or not; it is required for the voltage regulator. Select *Device Drivers* from the main menu:

```
    ...
    ...
    Power management options  --->
[*] Networking support  --->
    Device Drivers  --->
    File systems  --->
    Kernel hacking  --->
    ...
    ...
```

Then select *I2C support* as shown here:

```
    ...
    ...
    Input device support  --->
    Character devices  --->
<*> I2C support  --->
[ ] SPI support  --->
-*- GPIO Support  --->
    ...
    ...
```

Select *I2C Hardware Bus support* after selecting "I2C device interface" from the menu, as shown here:

```
    ...
    ...
<*>   I2C device interface
<*>   Autoselect pertinent helper modules (NEW)
      I2C Hardware Bus support  --->
      Miscellaneous I2C Chip support  --->
[ ]   I2C Core debugging messages (NEW)
    ...
    ...
```

Select *OMAP I2C adapter* as shown here:

```
    ...
    ...
< > GPIO-based bitbanging I2C (NEW)
< > OpenCores I2C Controller (NEW)
<*> OMAP I2C adapter
< > Simtec Generic I2C interface (NEW)
    *** External I2C/SMBus adapter drivers ***
    ...
    ...
```

Now select *System Type* from the main menu.

```
    ...
    ...
[*] Enable loadable module support  --->
[*] Enable the block layer  --->
    System Type  --->
    Bus support  --->
    Kernel Features  --->
    ...
    ...
```

Make sure that the PMIC present on the OMAP3530 EVM is selected here:

```
    ...
    ...
[ ] Gumstix Overo board
[*] OMAP3530 EVM board   --->
[*]   TWL4030/TPS65950 Power Module
[ ] OMAP3517/ AM3517 EVM board  --->
[ ] OMAP3 Pandora  --->
[ ] OMAP 3430 SDP board  --->
    ...
    ...
```

Come back to the main menu now and select Device Drivers as shown here:

```
    ...
    ...
    Power management options  --->
[*] Networking support  --->
    Device Drivers  --->
    File systems  --->
    Kernel hacking  --->
    ...
    ...
```

Select *Voltage and Current Regulator Support* as shown here:

```
[ ] DMA Engine support  --->
[ ] Auxiliary Display support  --->
[*] Voltage and Current Regulator Support  --->
< > Userspace I/O drivers  --->
[ ] Staging drivers  --->
```

Driver for the PMIC selected earlier should have been automatically selected, as shown here:

```
< >   Maxim 1586/1587 voltage regulator
-*-   TI TWL4030/TWL5030/TPS695x0 PMIC
< >   National Semiconductors LP3971 PMIC regulator driver
< >   TI TPS65023 Power regulators
```

13.4. Application Interface

This section provides the details of the application interface for the PMIC regulator driver.

Client device drivers are the ones which use PMIC regulator drivers to enable/ disable and/or regulate output voltage/current. Specifically, a client driver uses:

- Consumer driver interface: This uses a similar API to the kernel clock interface in that consumer drivers can get and put a regulator (like they can with clocks) and get/set voltage, current limit, enable and disable. This allows consumer complete control over their supply voltage and current limit.
- Sysfs interface: The linux voltage regulator framework also exports a lot of useful voltage/current/opmode data to userspace via sysfs. This could be used to help monitor device power consumption and status.

13.4.1. Consumer driver interface

As mentioned above, this interface provides complete control to the consumer driver over their supply voltage and/or current limit. Some of the commonly used APIs to achieve this are:

Name	Description
regulator_get	Get handle to a regulator
regulator_put	Release a regulator
regulator_enable	Enable a regulator
regulator_disable	Disable a regulator
regulator_is_enabled	Check status of the regulator
regulator_set_voltage	Change the output voltage
regulator_get_voltage	Fetch the current voltage
regulator_set_current_limit	Change the current limit
regulator_get_current_limit	Fetch the existing current limit

Table 13.2. Commonly Used APIs

13.4.2. Sysfs interface

The `/sys/class/regulator` interface can be used to read-back information which the VRF exports to the user-space.

See the table below for different sysfs entries in `/sys/class/regulator`:

Name	Description
name	string identifying the regulator, may be empty
type	regulator type (voltage, current)
state	regulator enable status (enabled, disabled)
microvolts	regulator output voltage (in microvolts)
microamps	regulator output current limit (in microamps)
min_microvolts	minimum safe working output voltage setting
max_microvolts	maximum safe working output voltage setting
min_microamps	minimum safe working output current setting
max_microamps	maximum safe working output current setting

Table 13.3. Sysfs interface

13.5. Writing a Consumer Driver

This chapter describes the steps required in writing a consumer driver to regulate the output voltage and to enable/disable the regulator. User should refer to "`include/linux/regulator/consumer.h`" for the complete list of supported APIs.

Writing a consumer driver involves the following steps:

- Getting the required regulator handle: Consumer driver has to first obtain a handle to the desired regulator, by passing the correct supply.

  ```
  int ret;
  struct regulator *reg;
  const char *supply = "vdd1";
  int min_uV, max_uV;

  reg = regulator_get(NULL, supply);
  ```

- Enabling it, if not already enabled: A regulator needs to be enabled before it can be used to change the output voltage. Regulator handle, obtained in the previous step, should be used now to enable it:

  ```
  ret = regulator_enable(reg);
  ```

  After enabling it, user can check the status of the regulator by:

  ```
  printk (KERN_INFO "Regulator Enabled = %d\n",
  regulator_is_enabled(reg));
  ```

- Changing the existing voltage: Consumer driver has to pass the appropriate minimum and maximum voltage levels, as desired by the use-case, to change the output voltage.

  ```
  ret = regulator_set_voltage(reg, min_uV, max_uV);
  ```

- Reading the existing voltage: Consumer driver can read back the existing voltage to check if the voltage was set properly or not.

  ```
  printk (KERN_INFO "Regulator Voltage = %d\n",
  regulator_get_voltage(reg));
  ```

- Disabling the regulator: Consumer driver can disable the regulator, if required. The framework ensures that the regulator is not disabled if other consumers are still using the same regulator.

  ```
  ret = regulator_disable(reg);
  ```

- Releasing the regulator handle: Consumer driver can release the regulator if it is no more required.

  ```
  regulator_put(reg);
  ```

  After that, the handle becomes no more valid and should not be used for any further operations.

13.6. Revision History

03.00.00	Initial version

Appendix

Abstract

This chapter contains useful information referenced in the earlier sections of the document.

Table of Contents

14.1. Creating bootable partition on MMC/SD Card

The MMC/SD card should have a valid bootable partition on the card before it can be used as boot media.

Download and install HP USB Disk Storage Format Tool [http://www.sysanalyser.com/sp27213.exe] on a Microsoft Windows host.

Follow these steps to format the card:

- Connect the card reader to the host machine where the formatting tool was installed.
- Insert the MMC/SD card into the card reader.
- Launch the HP USB Disk Storage Format Tool.
- Select **FAT32** as File System.
- Click on **Start**.
- After formatting is done Click **OK**.