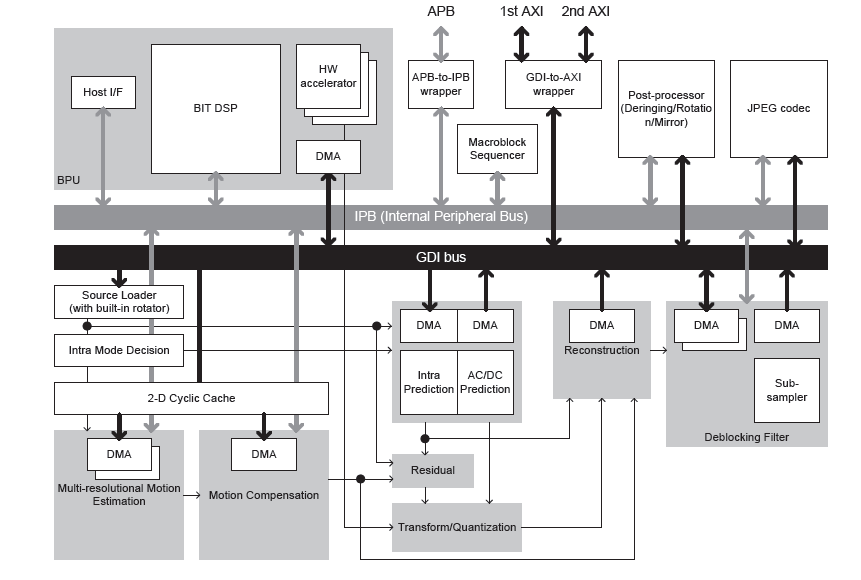
VPU Firmware Guide

Overview

Video Processing Uint of i.MX6Q is a high performance multi-standard video codec, it is located in the memory map at 0204 0000h.

VPU has an embedded BIT processor controlling internal video processing sub-blocks and communicating with the host processor through the IP bus. VPU can directly access the memory through the AXI bus for data throughput.



Error! No text of specified style in document.‑1 VPU block diagram

* 1. Features summary

VPU supports the following features:

* Video/Image Encode
  + H.264 BP/CBP/MP/HP
  + VC-1 SP/MP/AP
  + MPEG-4 SP/ASP
  + H.263 P0/P3
  + MPEG-1/2 MP/HP
  + Divx (Xvid) HP/PP/HTP/HDP
  + RV8/9/10
  + Sorenson Spark
  + VP8(1280x720)
  + AVS
  + H.264-MVC (1280x720)
  + MJPEG BP
* Video/Image Decode
* H.264 BP/CBP
* MPEG-4 SP
* H.263 P0/P3
* MJPEG BP encoding
* Multi-Instance
  + VPU can support infinite instances of decoder plus encoder concurrently by switching the contexts of codec in frame based mode which are stored in the memory. It is very helpful for multi-channel decoder applications.
* Performance
  + full HD video decoder up to 1920\*1088@30fps plus D1@303fps
  + full HD encoder up to 1920\*1088@30fps
  + MJPEG codec up to 8192\*8192

In the firmware driver, H264/VC-1 decoding and H264 encoding are demonstrated. Dual video decoder plus display concurrently are supported too.

* 1. Modes of operation

0.3.1 Input Stream

Regarding the input stream, VPU can work in stream mode or file play mode.

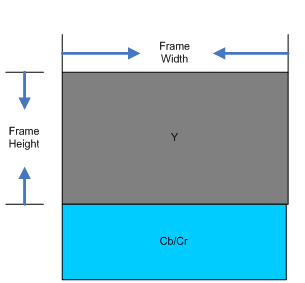
In streaming mode, the raw bitstream are put into the stream buffer if there is space left, the read and write pointer of the VPU will record and indicate the current status. The stream buffer is in ring-buffer mode. When the write pointer reaches the end, it will wrap back to the start. VPU itself will analyze the bitstream and start decoding by checking the start code. In encoder, the size of a frame is fixed and VPU will get the right data from the buffer.

In file play mode, only one whole frame is filled into the frame buffer. The next frame would be filled after the current frame is processed. Currently file play mode is not supported in iMx6Q VPU firmware.

0.3.2 Output Stream

Regarding the output, VPU can work in tiled mode or linear mode.

In linlear mode, the video output is in frame mode, that means a whole frame would be produced and stored in the registered frame buffer. The output data can be shown as below(no matter in progressive or interlaced mode, the output will be put into continuous frame buffer as we defined):



0‑ NV12 data format layout

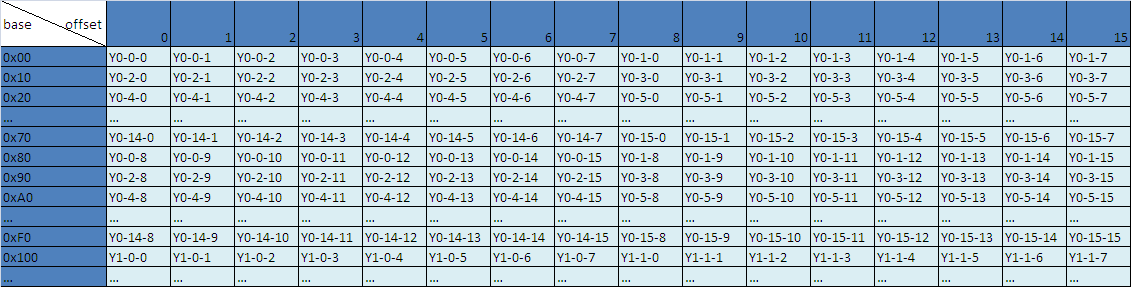
For YUV420 partial interleaved mode, which is also know as NV12, the data lies as

Y(0,0)Y(0,1)..Y(0,fw-1)Y(1,0)..Y(fh-1,fw-1)

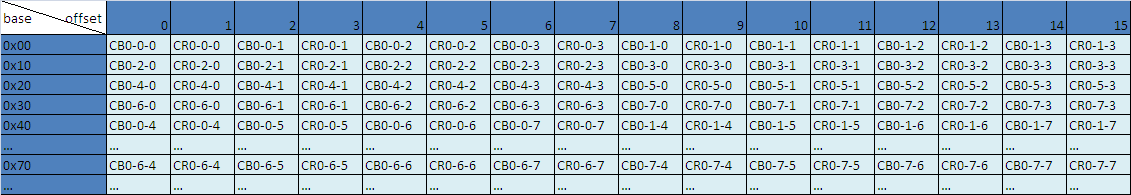
Cb(0,0)Cr(0,0)Cb(0,1)Cr(0,1)..Cb(0,fw/2-1)Cr(0,fw/2-1)Cb(1,0)Cr(1,0)..Cb(fh/2-1,fw/2-1)Cr(fh/2-1,fw/2-1)

In tiled mode, the video output is in 16\*16 block format. The benefit is that 16\*16 is the size of macroblocks for some codec standard such as AVC, and they are frequently exchanged between VPU internal ram/cache and the external memory. The data arranged in block mode can be more fastly loaded(due the burst size and cache line benifit) than frame mode. With the tiled mode enabled, the decoding performance could be increased about 10%.

Below is the data arrangement in the memory for progressive YCbCr4:2:0. The M-N-O-P format indicates the position of a pixel component, in which M means the component name(Y, Cb or Cr), N means the block number, O means the line number inside the block(For Luma, it varies from 0~7, for Chroma, it varies from 0~3), P means the pixel index inside the line.



0‑3 Luma data layout in VPU output tiled mode



0‑4 Chroma data layout in VPU output tiled mode

* 1. Clocks



VPU will work on 264MHz. the clock source could be AXI clock or the PLL2 PFD0/PFD2.

* 1. Resets and Interrupts

There is two ways to reset the VPU. One is through the system reset controller, by setting the vpu\_sw\_reset to 1 and wait for it to be self cleared. Another way is to reset the internal VPU bus and modules by setting the register BIT\_SW\_RESET.

VPU can support various interrupts by setting the register BIT\_INT\_ENABLE. The most useful is DEC\_PIC\_RUN/ENC\_PIC\_RUN which indicates that the current frame for decoding/encoding has been finished.

* 1. Initializing the driver

0.6.1 VPU common init

Before starting the codec tasks, VPU must be initialized for the first time. The host processor should do the following operations steps by step, all these things are done in VPU\_Init().

1. set the IO system. Reserve some chunks of memory for VPU work buffer and DMA usage.
2. Initialize the codec instances.
3. Check if VPU has been initialized, by checking the PC pointer of BIT processor. the PC not equaling to zero indicates that VPU has been initialized, Initialization procedure would be finished at once. Or else, go to the next steps.
4. Download the firmware to VPU work buffer, which is accessible by VPU directly during runtime.
5. Download the first 4 KB of the microcode (firmware) to program memory in BIT Processor.
6. Set the BIT Processor buffer pointers for working buffer, parameter buffer, and code buffer.

The working buffer will store the context of codec instances, so its size should be increased as many instances. The common parts takes 210Kbytes and for an extra instance, 47Kbyte is needed. That means the working buffer size should be no less than 210K + MAX\_NUM\_INSTANCES \* 47Kbytes.

Code buffer is used to store the firmware binary. It should be no less than the firmware size of VPU, and it will varies on different versions of firmware or different VPU product.

1. Set the control options (full/empty check) and endian mode of bitstream buffer. Little endian and big endian can be supported.
2. Set frame buffer endian and chrominance option (CbCr Interleave or plannar.) Pay attention that if VDOA is enabled, the output data must be in CbCr interleaved mode.
3. Set the Interrupt Enable register.
4. Enable the BIT Processor by setting the register BIT\_CODE\_RUN to be 1.

0.6.2 VPU decoder init

By calling decoder\_setup(), a decoding instance will be created.

1. first of all, open an instance by call VPU\_DecOpen. An codec instance will be allocated and configured. Parameters such as codec standard(AVC, VC-1, MPEG4, etc.), instance index, data map would be set here. Pay attention that in order to support multi-instances, the context of the instance would be saved for future task switches. The following registers should be backup and restore before instances switch:

These are the registers to backup and restore before instances are switched over.

• BIT\_BIT\_STREAM\_CTRL

• BIT\_FRAME\_MEM\_CTRL

• BIT\_BIT\_STREAM\_PARAM

• BIT\_RD\_PTR

• BIT\_WR\_PTR

• BIT\_AXI\_SRAM\_USE

• BIT\_FRM\_DIS\_FLAG

1. feeding bitstream into the bitstream buffer. Meanwhile the BIT\_WR\_PTR should be updated.
2. Parse the bitstream. In this step, Sequence Initialization will be done. Taking AVC as the example, the VPU will search for the start code, get all the configuration informations from the stream. Picture size, frame rate, profile, level and number of frame buffers needed would be got.
3. Allocate the buffers and register it to the VPU. All these buffers are used for the VPU output. Buffer numbers will be no less than the minimum needed.

Now we can start the video decoding by calling VPU\_DecStartOneFrame.

0.6.3 VPU encoder init

By calling encoder\_setup(), a encoding instance will be created.

1. first of all, open an instance by call VPU\_EncOpen. An codec instance will be allocated and configured. In this step, the picture size(width and height), codec stardard, data format, GOP size and frame rate should be set.
2. feeding bitstream into the bitstream buffer. VPU has two schemes for bitstream buffer in encoder. One is the ring buffer mode, in this mode a single fixed-size buffer will be used as if it is connected end-to end. A read and a write pointer would be used to indicate the usage of the buffer. This is very usefull in case the system memory is very limited. The other scheme is line-buffer mode.in this mode a whole frame would be put into the bitstream buffer and VPU start encoding from the start to the end of the frame without exchanging data with the host.
3. In this step, Sequence Initialization will be done. In this final stage of configuration, the returned parameter from VPU\_EncGetInitialInfo(), the minimum number of frame buffer means that applications should reserve at least the same number of frame buffers to VPU for proper encoding operation.
4. Allocate the buffers and register it to the VPU. All these buffers are used for the VPU output. Buffer numbers will be no less than the minimum needed.
5. When opening an encoder instance is completed by calling VPU\_EncGetInitialInfo(), applications MUST
6. generate the high-level header syntaxes such as VOS/VO/VOL headers in MPEG-4 and SPS/PPS in AVC from VPU by using VPU\_EncGiveCommand().

There are two possible ways for generating these header syntaxes via PARA\_BUF or via bitstream buffer. The recommended way for getting header syntaxes is to use ENC\_PUT\_AVC/MP4\_HEADER command via bitstream buffer. If applications use this set of commands, then the header syntaxes as a result will be stored into the bitstream buffer according to the given endian setting.The other way for generating header syntaxes is via PARA\_BUF. It can be done by using ENC\_GET\_VOS\_HEADER, ENC\_GET\_VO\_HEADER or ENC\_GET\_VOL\_HEADER in MPEG-4 case, and ENC\_GET\_SPS\_RBSP or ENC\_GET\_PPS\_RBSP in H.264 case. Regardless of streaming mode, this command will generate header syntaxes and write them to PARA\_BUF instead of the bitstream buffer. But in this case, endian setting will always be Big-Endian. Thus, on Little-Endian system Endian conversion should be performed by the host processor.

Now we can start the video decoding by calling VPU\_EncStartOneFrame. After the frame encoding is finished, the host processor can get the output from the ring or line stream buffer, and store them to the destination.

0.6.4 multi-instance

In order to support this multi-instance operation, BIT processor uses an internal context parameter set for each decoder instance.

While creating a new instance and starting a picture processing, a set of these context parameters will be created and updated automatically within VPU. Because of this internal context management scheme, different decoder tasks running on host processor can control VPU operations independently with their own instance numbers.

When creating a new instance, application task will get a new handle specifying an instance if a new handle isavailable o n VPU. And all the following operations for the give application task could be separately handled on VPU by using this task-specific handle.

Since VPU can only perform one picture processing task at a time, each application would share the unique hardware resources in time-division mode, that means each task should check whether VPU is ready or not before starting a new picture operation.

By calling a function for closing a certain instance, application can easily terminate a single task of video operation on VPU.

* 1. Testing the driver

VPU encoder and decoder test cases have been demonstrated, including multi-instatnce demo(dual video decoder + display, the default video standard is AVC, the first display is Hannstar LVDS panel, the second display is embedded HDMI display.)

In decoder test, there are two modes to select:

1. Endless test
2. Play the file to the end.

In the encoder test, the default encode stardard is AVC, and default input size is 320\*240. The output is stored in the memory. After the encoding finished, the user can select to play the encoded file at once, or else you can check the data in PC side by dumping them to files using debug tools.

* 1. Source code structure

The code structure is as below:

|  |  |
| --- | --- |
| Description | Location |
| Low-level driver source | ./src/sdk/vpu/drv/vpu\_lib.c  ./src/sdk/vpu/drv/vpu\_gdi.c  ./src/sdk/vpu/drv/vpu\_util.c  ./src/sdk/vpu/drv/vpu\_io.c |
| Header files | ./src/sdk/vpu/inc/BitAsmTable\_CODA\_960.h  ./src/sdk/vpu/inc/vpu\_gdi.h  ./src/sdk/vpu/inc/vpu\_io.h  ./src/sdk/vpu/inc/vpu\_lib.h  ./src/sdk/vpu/inc/vpu\_reg.h  ./src/sdk/vpu/inc/vpu\_test.h  ./src/sdk/vpu/inc/vpu\_util.h |
| Test cases | ./src/sdk/vpu/test/vpu\_test.c  ./src/sdk/vpu/test/dec.c  ./src/sdk/vpu/test/enc.c  ./src/sdk/vpu/test/fb.c  ./src/sdk/vpu/test/utils.c |

* 1. demo setup guide

This guide is elaborated to show how to setup and run the dual video + dual display demo on i.Mx6x EVB board.

**Text for user input.**

***Text for important hints.***

1. Create the image over SD card. Under linux, using fidsk/mkfs.vfat/dd to create a bootable image together with the FAT32 file system on the same SD card.

* sudo fidsk /dev/sdx , sdx is the device name of your SD card.

**sudo fdisk /dev/sdb**

Command (m for help): **m**

***Delete existing partition if there is.***

Command (m for help): **d**

Selected partition 1

Command (m for help): **n**

Command action

e extended

p primary partition (1-4)

**p**

Partition number (1-4): **1**

First cylinder (1-1023, default 1): **256**

***Here the start address should be larger than 2M(space reserved to burn the test binary). for one cylinder it is 4k. Here I reserved 1G space.***

Last cylinder or +size or +sizeM or +sizeK (256-1023, default 1023): **1023**

Command (m for help): w

The partition table has been altered!

Calling ioctl() to re-read partition table.

Syncing disks.

Now I have one partitions on the SD card.

**cat /proc/partitions**

major minor #blocks name

8 0 78125000 sda

8 1 104391 sda1

8 2 78019672 sda2

253 0 75956224 dm-0

253 1 2031616 dm-1

8 16 3872256 sdb

8 17 2904576 sdb1

* Using mkfs.vfat to format the partition

**sudo mkfs.vfat /dev/sdb1**

* Copy two video clips to the SD card. ***Note that the filename should be less than 8 characters with “.264” extension. The video should be RAW h264 encoded files with no any container. The program will find the first two valid 264 files for playing.***
* Burn the image to the SD card.

**sudo dd if=output/mx61/evb/bin/mx61evb-vdec-sdk.bin of=/dev/sdb seek=2 skip=2 && sync**

***NOTE that seek=2 skip=2 is mandatory, without them the MBR of the file system would be overwritten!!***

1. put the SD card into SLOT4
2. set the boot switch to boot from SD4
3. plug the Hannstar LVDS panel into LVDS0 connector
4. plug the HDMI cable to J5 for the secondary display
5. connect the serial cable and 5V power supply, power on the board.