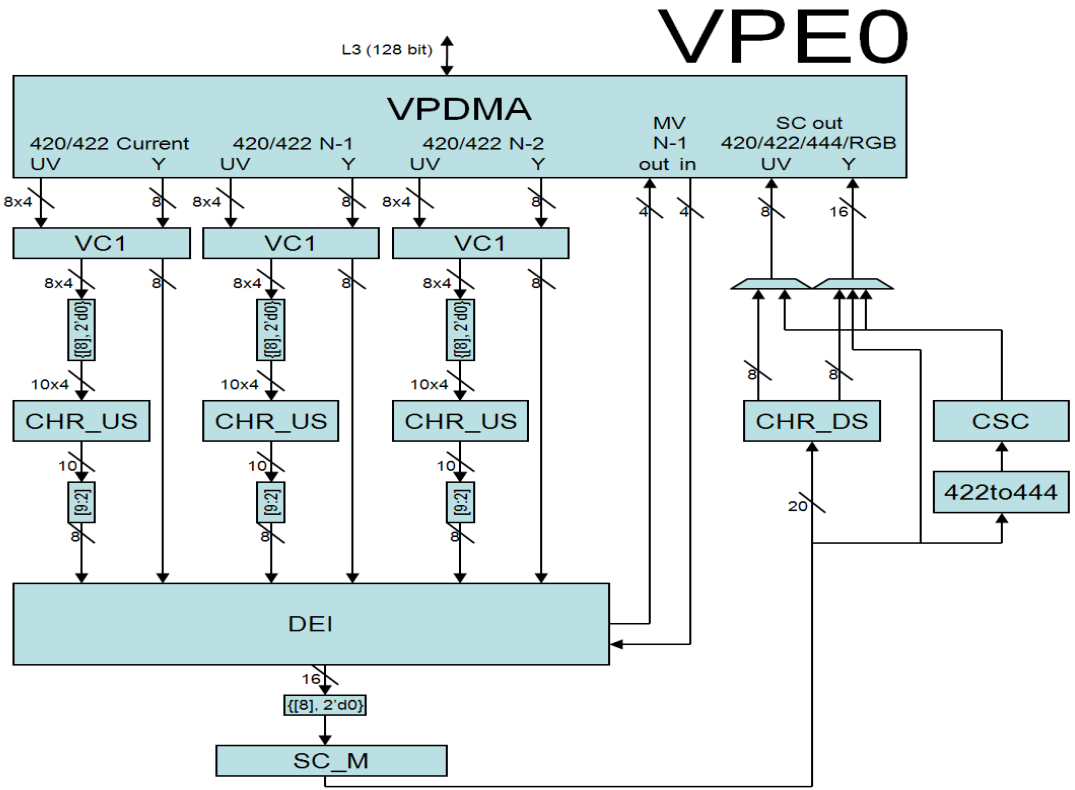
VPE Overview
(Video Processing Engine)

9th March 2017
Version 1.0

Agenda

- VPE Introduction
  - Features
  - SC
  - DEI
  - VPDMA
- VPE Driver Overview
  - FVID2 Interface
  - Driver Design

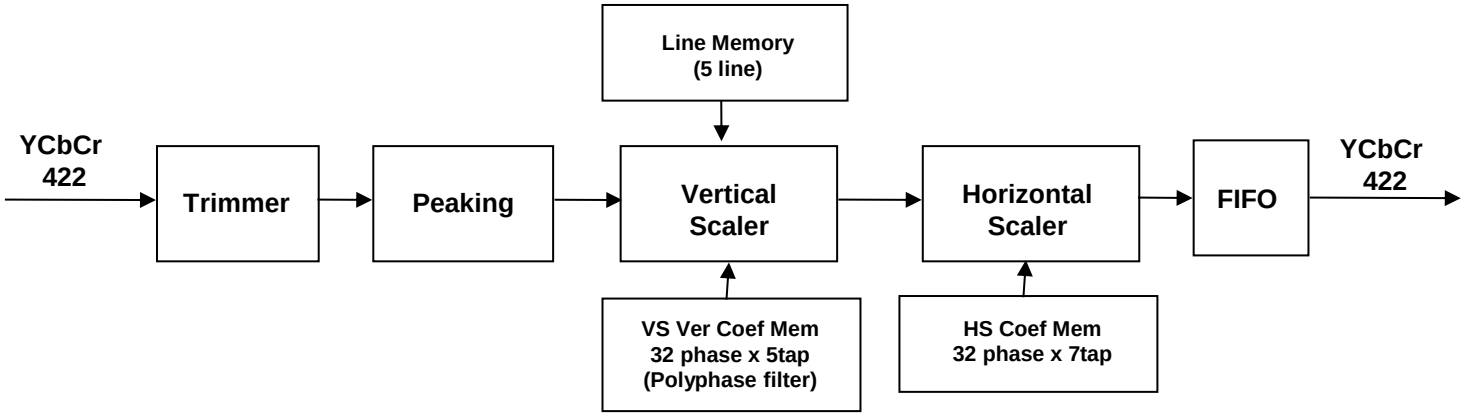
VPE Block Diagram



VPE Features

- Format conversion
  - Inputs: YUV422I, YUV420SP (Uses CHR_US), YUV422SP
  - Outputs: YUV422I, YUV420SP (Uses CHR_DS), YUV422SP, RGB888 (Uses CSC), YUV444I
- Deinterlacing (interlaced to progressive)
  - 4 field motion based algorithm
  - up to two 1080i video sources
  - DEI can be bypassed in case of progressive input
- Scaling from 1/8x to 2048 pixels
- Color space conversion
  - YUV to RGB color space conversion using CSC block
- VC-1 Range Mapping and Range Reduction
- Supports up to 304 MHz clock
- Tiled (2D) input/output

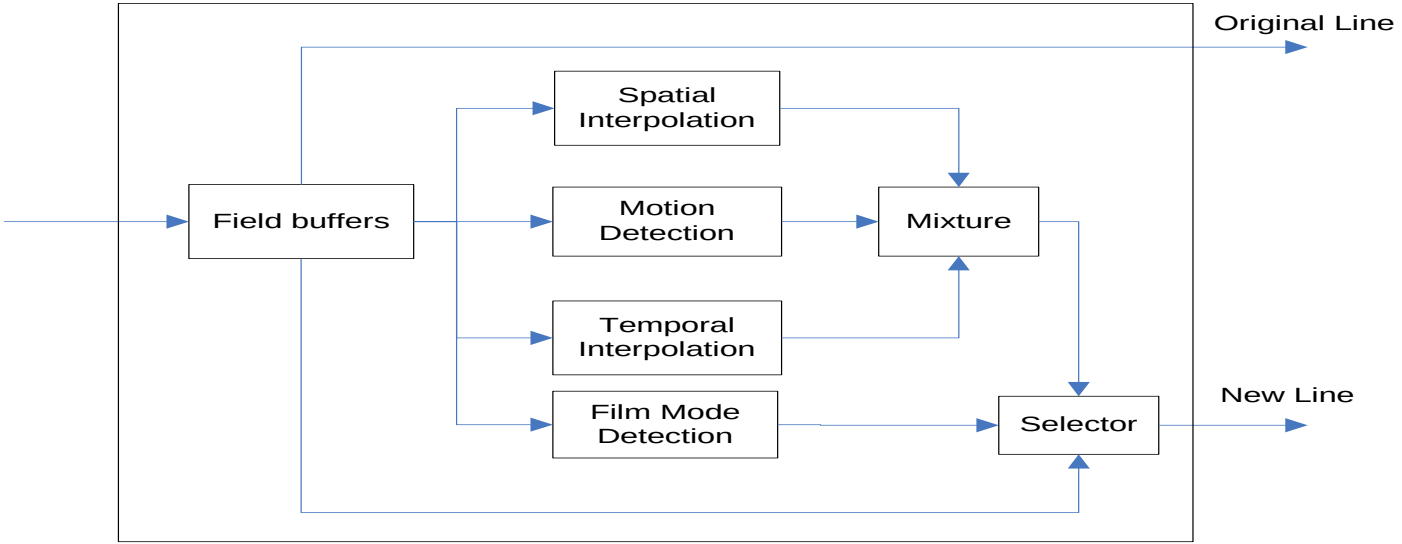
SC Block



SC Features

- Vertical and horizontal up and down scaling
- Polyphase filter upscaling
- Running average vertical down scaling
- Decimation and polyphase filtering for horizontal scaling
- Non-linear scaling for stretched/compressed left and right sides
- Input image trimmer for pan/scan support
- Pre-scaling peaking filter for enhanced sharpness
- Scale field as frame
- Interlacing of scaled output
- Full 1080p input and output support
- Scaling filter Coefficient memory download

Motion-Adaptive Deinterlacer Block



$$\hat{y}(j,i,n) = \alpha y_{spat}(j,i,n) + (1-\alpha) y_{temp}(j,i,n)$$

DEI Features

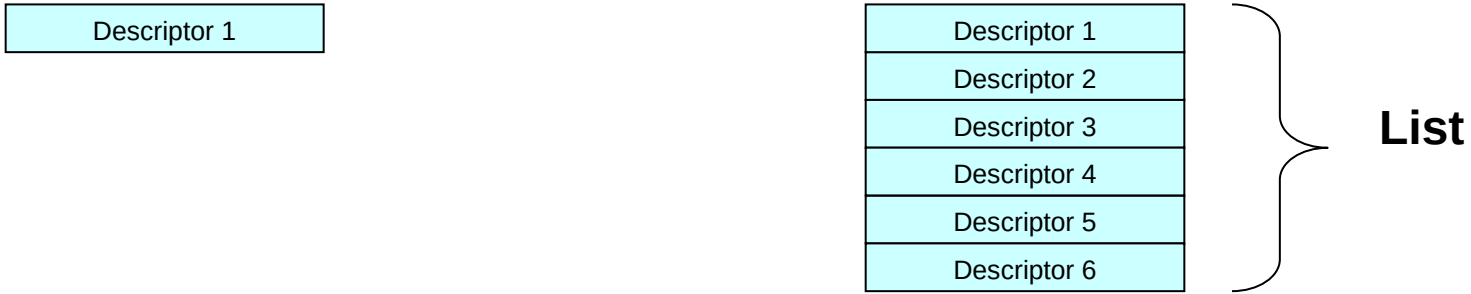
- Motion-adaptive deinterlacing
  - Motion detection is based on Luma only
  - 4-field data is used
- Motion Detection (MDT)
  - Examines 3 fields of input video data (luma only) and calculates a 4 bit motion vector to drive the Edge Directed Interpolation Block
- Edge-Directed Interpolation (EDI)
  - Edge detection using luma pixels in a 2x7 window
  - Soft-switch between edge directed interpolation and vertical interpolation depending on the confidence factor
- Film Mode Detection (FMD)
  - 3-2 pull down detection (NTSC style)
  - 2-2 pull down detection (PAL style)
  - Bad Edit Detection (BED)
- Progressive or interlaced bypass mode

VPDMA

- The VPDMA’s primary function is to move data between external memory and internal processing modules that source or sink data.
- Clients
  - The modules that source or sink data are referred to as clients. i.e. the physical interface between the processing module and external memory is called a client
  - Each client can be configured to have specific start event which allows for the channel data to start flowing to or from the external client
- Channel
  - A channel is setup/mechanism inside the VPDMA to connect a specific memory buffer/transfer to a specific client

Descriptors/List

- Crux of VPDMA programming is descriptors.
- Descriptors defines how data will be transferred through different channel. In other words, channel is described through a Data Transfer descriptor.
- The client that the channel is mapped to interprets the information in the descriptor to perform the requested data transfer.
- List is a group of descriptors that makes up a set of DMA transfers that need to be completed.
- List can contain any kind of descriptor without limitation and be of any size

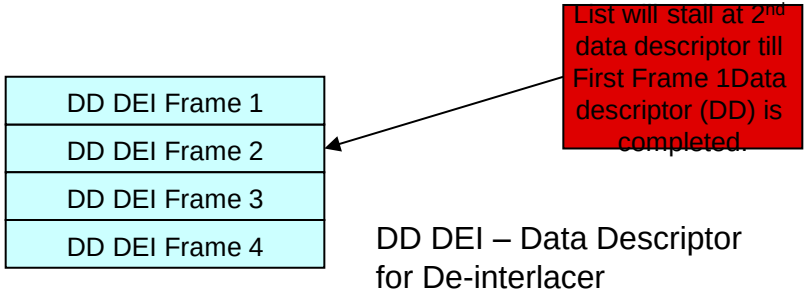


How VPDMA operates

- The VPDMA Controller works on lists of descriptors.
- CPU creates the lists of descriptors in DDR in the order it wants them executed
- CPU then writes the location of the list to the LIST_ADDR register, followed by writing the size, type of list and list number to the LIST_ATTR register and post list
- The List Manager module of VPDMA (manages different lists) will then schedule a DMA transfer to pull in the portion of the list that it can store in internal VPDMA memory
- List Manager will sequentially process descriptors in the list and does data transfer

List Contd..

- Data transfer could be initiated by list. Each List represents data descriptors.
- VPE supports maximum up to 8 lists but there is no restriction on the size of each list.
- A Data Transfer Descriptor will be removed from the list when the resource specified by the Channel field is free. If the Channel is not free when the list reaches a data transfer descriptor then the list will stall until the current transfer on the channel has completed



Descriptors Contd..

- There three kinds of descriptors which could be programmed in the list.
  - Data Transfer Descriptor – describes video data transfer
  - Configurations Descriptor – Used for set up client configuration like scalar configuration for scaling ratio, crop window etc
  - Control Descriptor – Used in list to give control commands to list manager like wait till transfer is complete before initiating next transfer etc

Driver Programming Difference with VPDMA

Register based programming

1. Program hardware parameters in the registers
2. Program destination buff address
3. Program source buff address
4. Enable interrupt generation after completion of operation
5. Trigger h/w
6. Wait for interrupt

Descriptor based programming

1. Program config Descriptor for scalar parameters
2. Program DD for destination
3. Program DD for source
4. Program SI control descriptor for interrupt
5. Post list
6. Wait for interrupt

The main advantage with descriptor based programming is that reconfiguring the hardware register (say for another context) takes very minimal CPU cycles as these are already pre-programmed and kept in the memory per context

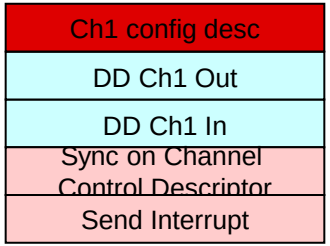
Typical Descriptor Layout

- Three frames from different channel has to be scaled to different ratio and application needs intimation after resizing of all three frames

Descriptor based programming

1. Program config Descriptor for scalar parameters
2. Program DD for destination
3. Program DD for source
4. Program SI control descriptor for interrupt
5. Post list
6. Wait for interrupt

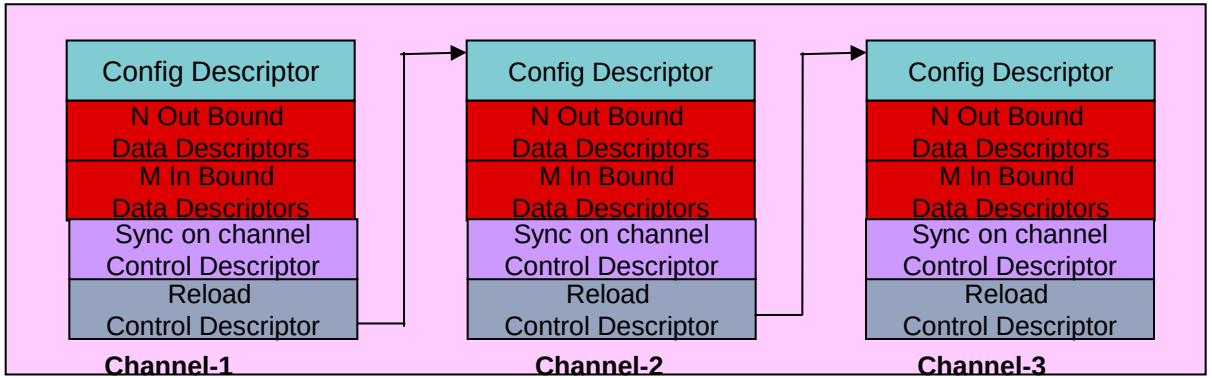
1. Program descriptor in DDR



2. These set of descriptor creates list in DDR
3. Write address of this list in LIST_ADDR and LIST_ATTRIB register which posts list
4. Wait for interrupt from SI which indicates completion

Typical Descriptor Layout Contd..

- Why do need “Sync on channel control descriptor for output client”
  - NF (new frame signal) flows from output to input client
  - It is possible that because of the pipeline delay, nf signal for the channel-1 will arrive at the inbound client after vpdma parses config descriptor for channel-2.
  - This will cause problem for channel 1 and it means that channel 2 config has been applied to channel 1.
  - Using Sync on Channel, it has been tried to delay parsing of config descriptor for channel 2



VPE Driver Overview (based on FVID2 Interface)

FVID2 Introduction

- What is FVID2?
  - Next version of FVID and it addresses the different limitations of FVID
  - Provides interface to streaming operations like queuing of buffers to driver and getting back a buffer from the driver
  - Abstracts the underlying hardware for the video application with a standard set of interface
  - Gives a same look and feel for video applications across different SOC
  - Interface is independent of OS/Hardware/Driver
  - FVID2 is currently supported on BIOS6
- What is not FVID2?
  - Not the actual driver
  - Does not define hardware specific APIs and structures

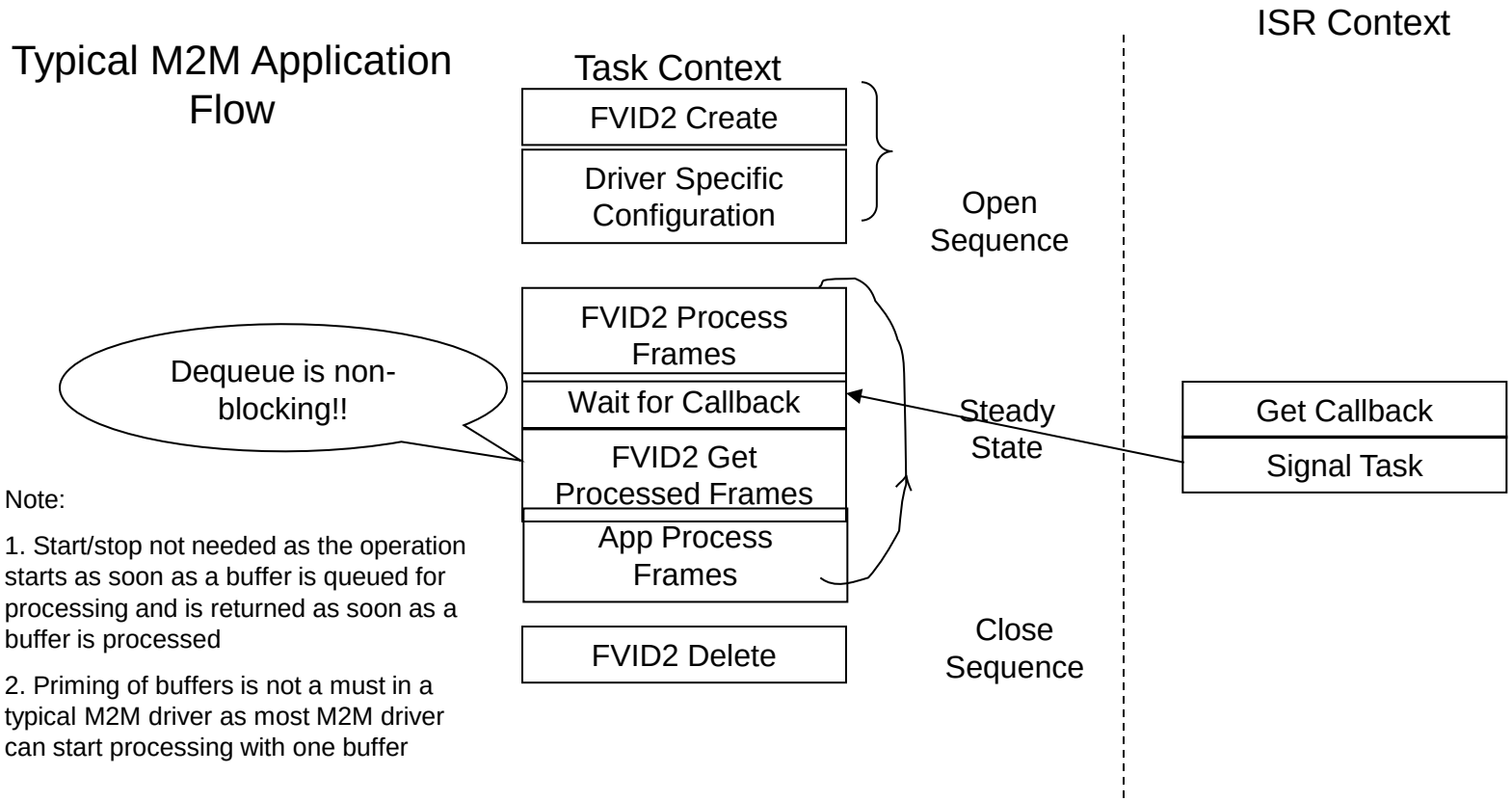
Understanding FVID2 - Interfaces

- *FVID2_init*
  - Initializes the drivers and the hardware. Should be called before calling any of the FVID2 functions
- *FVID2_deInit*
  - Un-initializes the drivers and the hardware
- *FVID2_create*
  - Opens a instance/channel video driver
- *FVID2_delete*
  - Closes a instance/channel of a video driver
- *FVID2_control*
  - To send standard (set/get format, alloc/free buffers etc..) or device/driver specific control commands to video driver
- *FVID2_queue*
  - Submit a video buffer to video driver. Used in display/capture drivers
- *FVID2_dequeue*
  - Get back a video buffer from the video driver. Used in display/capture drivers

Understanding FVID2 – Interfaces Contd.

- *FVID2_processFrames*
  - Submit video buffers to video driver for processing. Used only in M2M drivers
- *FVID2_getProcessedFrames*
  - Get back the processed video buffers from video driver. Used only in M2M drivers
- *FVID2_start*
  - Start video capture or display operation. Not used in M2M drivers
- *FVID2_stop*
  - Stop video capture or display operation. Not used in M2M drivers

Understanding FVID2 – M2M Application Flow



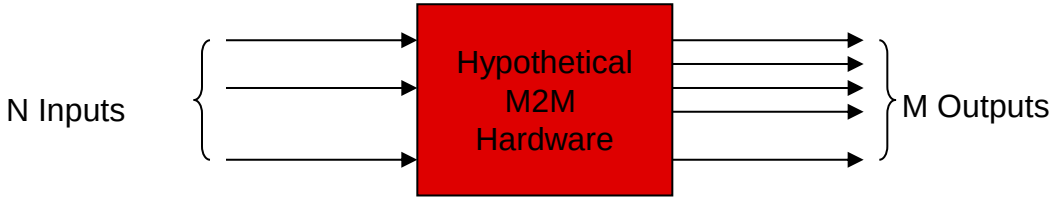
Note:

1. Start/stop not needed as the operation starts as soon as a buffer is queued for processing and is returned as soon as a buffer is processed

2. Priming of buffers is not a must in a typical M2M driver as most M2M driver can start processing with one buffer

Understanding FVID2 – M2M Interface

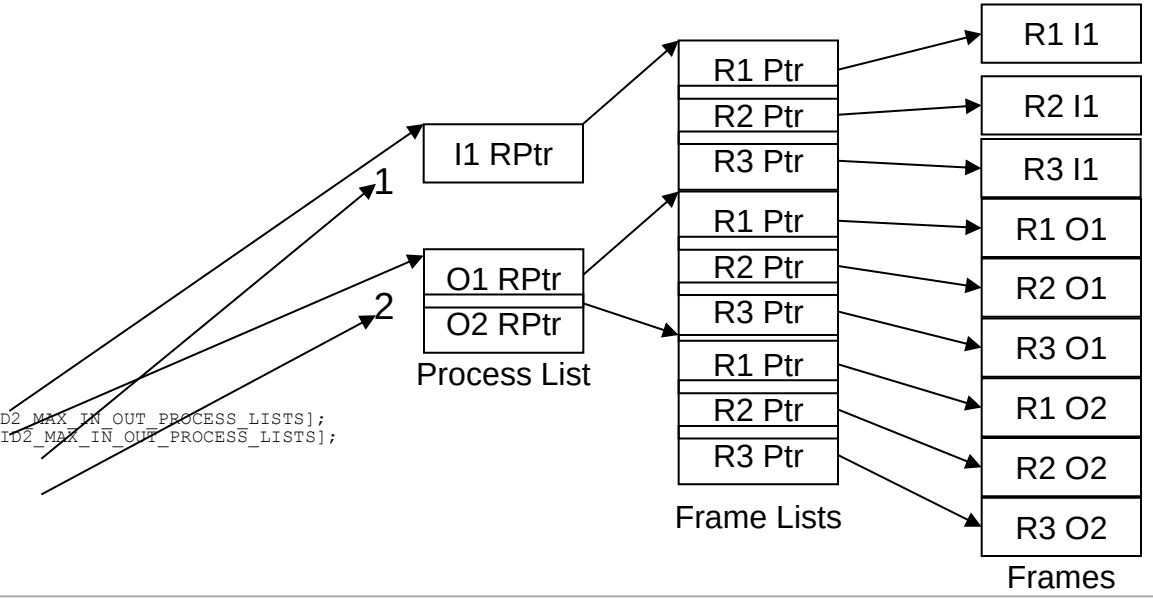
Key M2M Driver Requirements	Covered already by FVID2
Non-blocking	Callbacks supported ✓
Support multiple request per call	Frame List supports this ✓
Support multiple inputs and multiple outputs	Is covered now!! ✓



Understanding FVID2 – M2M Interface Contd…

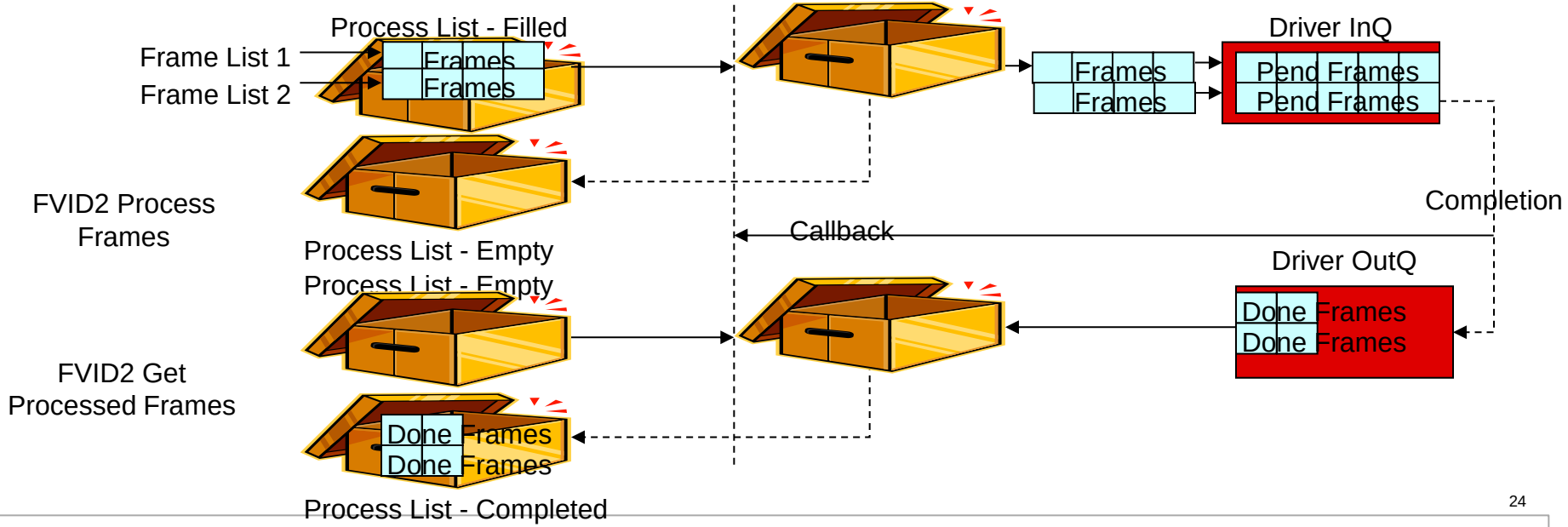
- FVID2 FrameList -> Multiple buffers/requests per input/output
- FVID2_ProcessList -> Multiple input and multiple output which in turn could contain multiple requests
- FVID2_ProcessList contains an array of FVID2_FrameList pointers for both input streams and output streams separately
- Two more new APIs – FVID2_processFrames(), FVID2_getProcessedFrames() instead of FVID2_queue(), FVID2_dequeue() which are used for stream drivers (display/capture)

```
typedef struct FVID2_Frame_t
{
    Ptr                 addr[2][3];
    UInt32              channelNum;
    /* Other members not shown */
} FVID2_Frame;

typedef struct FVID2_FrameList_t
{
    FVID2_Frame         *frames[64];
    UInt32              numFrames;
    /* Other members not shown */
} FVID2_FrameList;

typedef struct FVID2_ProcessList_t
{
    FVID2_FrameList     *inFrameList[FVID2_MAX_IN_OUT_PROCESS_LISTS];
    FVID2_FrameList     *outFrameList[FVID2_MAX_IN_OUT_PROCESS_LISTS];
    UInt32              numInLists;
    UInt32              numOutLists;
} FVID2_ProcessList;
```



Understanding FVID2 – M2M Interface

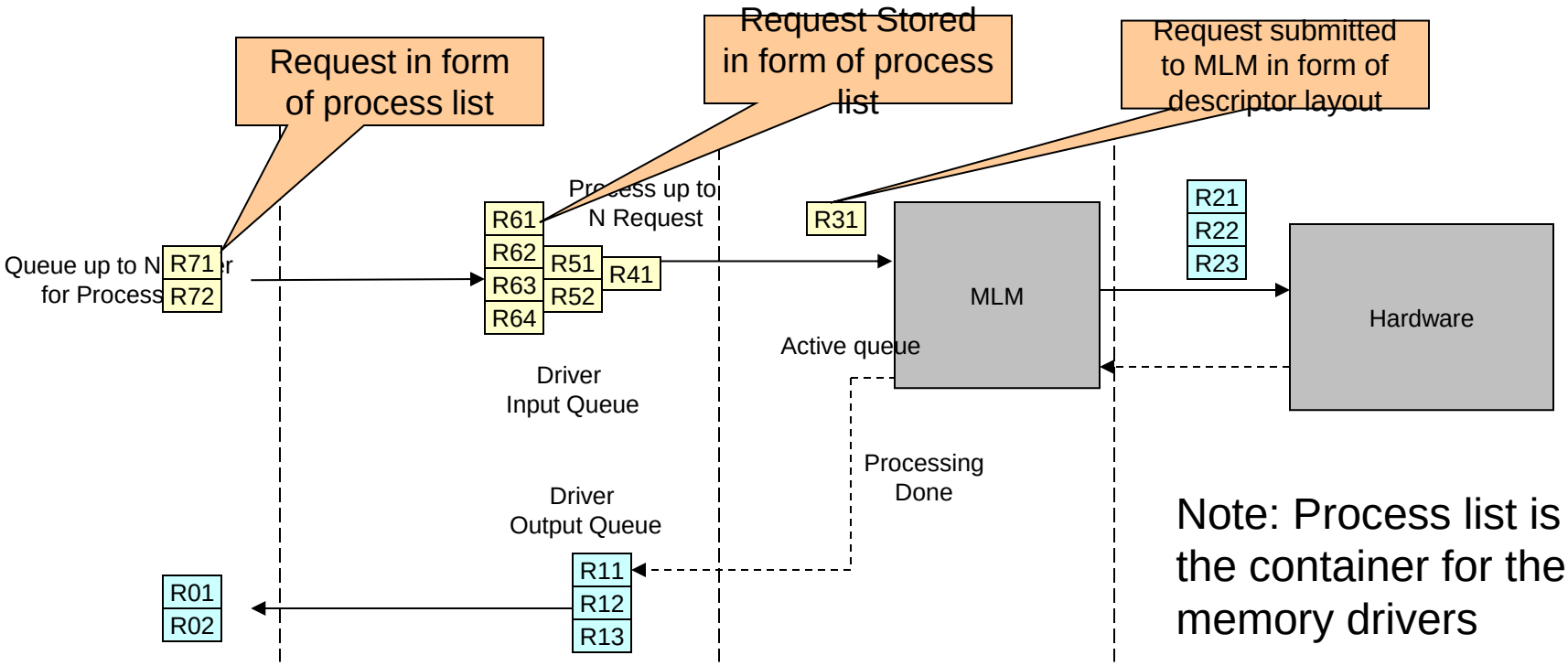
- Similar to queue/dequeue calls, only frame lists are queued/dequeued in/from the driver
- Process list is not queued/dequeued
- Now Process list acts like a container
- Hence application can't reuse the submitted frame lists till it gets them back from driver



Generic M2M Driver Requirements Summary…

- All Memory to Memory (M2M) drivers are non-blocking i.e. asynchronous drivers. Blocking calls not supported!!
- Unlike display and capture drivers, it could be opened multiple times – supports multiple handles (N) for the same driver
- Each call/request can consist of set of buffers
- M (no of picture or frame in each request) is fixed per handle at time of the time of driver open
- Callback will be generated after processing all requests in a given set
- Each handle/channel can have different configuration

VPE Driver Flow

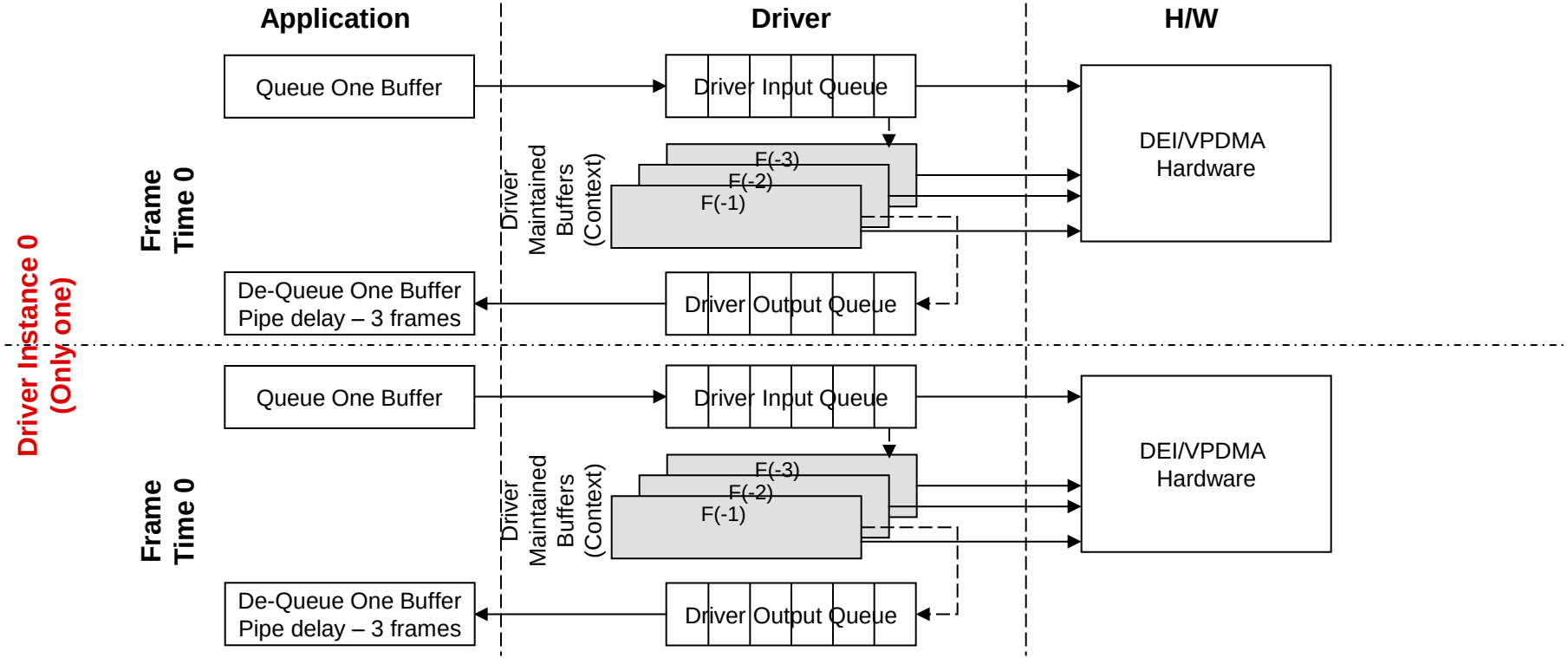


Note: Process list is the container for the memory drivers

Handling of context fields

- VPE driver
  - Takes one field (F1), de-interlaces, provides output (OF1).
  - **F1 is held back by the driver, as context** field. **Apps SHOULD NOT re-use F1**.
  - On input field (F2), uses context field (F1), de-interlaces F2 and provides output (OF2).
  - **F2 is held back by the driver, as context** field. (context fields are F1 & F2) **Apps SHOULD NOT re-use F1 & F2**.
  - On input field (F3), uses context field (F1 & F2), de-interlaces F3 and provides (OF3).
  - **F3 is held back by the driver, as context** field. (context fields are F1, F2 & F3) **Apps SHOULD NOT re-use F1, F2 & F3**.
  - On input field (F4), uses context field (F1, F2 & F3), de-interlaces F4 and provides (OF4).
  - **F4 is held back by the driver, as context** field. (context fields are F2, F3 & F4) **Apps SHOULD NOT re-use F2, F3 & F4, Apps could re-use F1**

Handling of context fields (Queuing)



Questions?
Thank You