FVID2 Interface

9th March 2017
Version 1.0

Agenda

- Introduction
  - Challenges in Defining Video Driver Interface
  - What is and what is not FVID2?
- FVID to FVID2
  - Limitation of FVID
- Understanding FVID2
  - Interfaces
  - Application Flow
  - Features supported

Introduction - Interface definition challenges

- Typical problems with conventional video interfaces
  - Should provide same look and feel for video applications across different Silicon
  - Interface should be independent of OS/Hardware/Driver so that it is scalable
  - Common APIs for display, capture and memory operation like scaling – reduces data structure manipulation/copy from one operation to other in different stages of operation in chain
  - Video composition- Multi-window Operation – Multiple buffers representing a single frame, say, 16 video streams representing one frame
    - Challenge is that same interface should be scalable for 1 video stream as well
  - Multi channel capture say 16 D1/64 CIF capture – interface should be capable to take multiple buffers
    - Challenge is that same interface should be scalable for 1 video stream as well
  - Multiplexed capture - One queue could have multiple dequeue/callback
  - All drivers should be non-blocking i.e. display, capture and memory operation

Introduction - Interface definition challenges contd..

- Typical problems with conventional Memory drivers
  - Memory operation should have similar capabilities to take inputs from multiple streams as talked in section for reducing software/driver overheads
    - Challenge is that same interface should be scalable for 1 to n video stream at both input and output
  - Number of input and output streams differ depending upon memory operations like
    - Scalar requires one input and one output
    - De-interlace may require 2/3 previous frames i.e. 2/3 input and one output
    - Noise filter may require future frame i.e. 2 input and one output
    - Two scalar in inline with capability to go back in memory – requires one input and two output

    Challenge is dealing of these variations in the single API
  - Video composition in the memory driver
  - Sliced based support for low latency application
  - Interface should be scalable for OSD/graphics plane

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 DSP/BIOS OS
- What is not FVID2?
  - Not the actual driver
  - Does not define hardware specific APIs and structures

FVID to FVID2 – Limitation of FVID

Feature	Supported in FVID?	Supported in FVID2?
Non-blocking interface with callback for display/capture/M2M (memory to memory) drivers	No – Even though supported by GIO/SIO of BIOS	Yes
Multi-window Operation – Multiple buffers representing a single frame	No	Yes
Multiplexed capture/M2M drivers – Queue/Dequeue can take multiple requests	No	Yes
Multiplexed capture - One queue could have multiple dequeue/callback	No	Yes
M2M driver interface	No – Supports only stream IO	Yes
Per frame changes like scaling, positioning etc…	Not supported directly – can use the reserved field of FVID Frame structure	Yes
Field based capture/display (in case of de-interlaced display)	No	Yes
Sliced based capture/memory operation	No	Yes

Understanding FVID2 - Interfaces

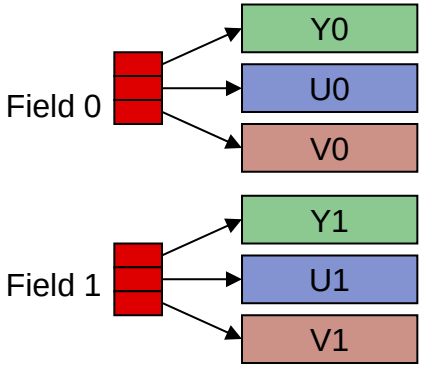
- *FVID2_init*
  - Initializes the drivers and the hardware. Should be called before calling any of the FVID2 functions
- *FVID2_deInit*
  - Un-initializes the drivers and the hardware
- *FVID2_create*
  - Opens a instance/channel video driver
- *FVID2_delete*
  - Closes a instance/channel of a video 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
- *FVID2_processFrames*
  - Submit video buffers to video driver for processing. Used only in M2M drivers

Understanding FVID2 - Interfaces

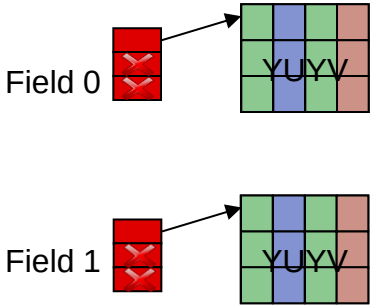
- *FVID2_getProcessedFrames*
  - Get back the processed video buffers from video driver. Used only in M2M drivers
- *FVID2_start*
  - Start video capture or display operation. Not used in M2M drivers
- *FVID2_stop*
  - Stop video capture or display operation. Not used in M2M drivers
- FVID2_Frame
  - Represents the video frame buffer along with other meta data
  - This is the entity which is exchanged between driver and application and not the buffer address pointers
  - Meta data include timestamp, field ID, per frame configuration, application data etc…
  - Since video buffers can have up to 3 planes and two fields (in the case of YUV planar interlaced), buffer addresses are represented using a two dimensional array of pointers of size 2 (field) x 3 (planes)

Understanding FVID2 - Interfaces

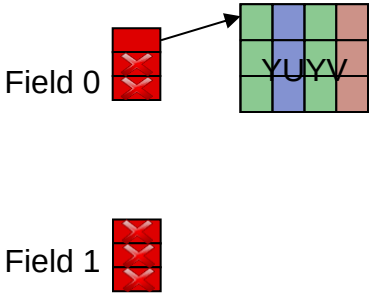
- FVID2_Frame – How address pointers are used?



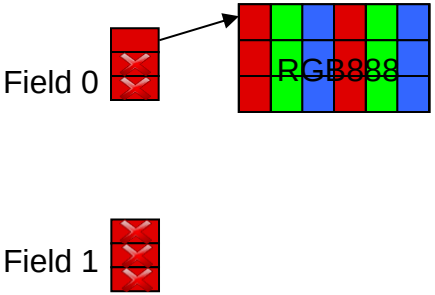
YUV 422/420 Planar Buffer Format (Interlaced)



YUV 422 Interleaved Buffer Format (Interlaced)



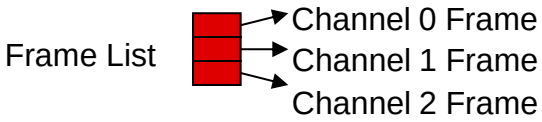
YUV 422 Interleaved Buffer Format (Progressive)

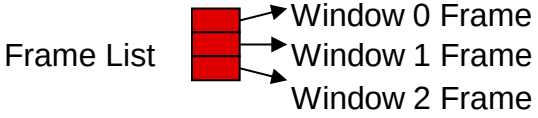


RGB 888 Packed Buffer Format (Progressive)

Understanding FVID2 - Interfaces

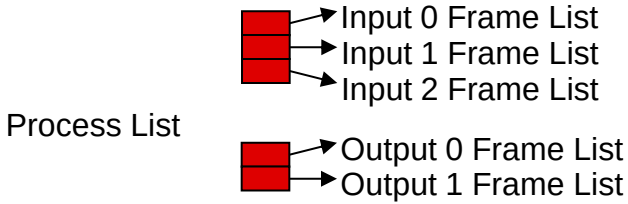
- FVID2_FrameList
  - Represents N FVID2_Frame
  - N Frames could represent
    - Different capture channels in multiplexed capture
    - Buffer address for each of the window in multi window mode
  - N is fixed at 64 in the current implementation



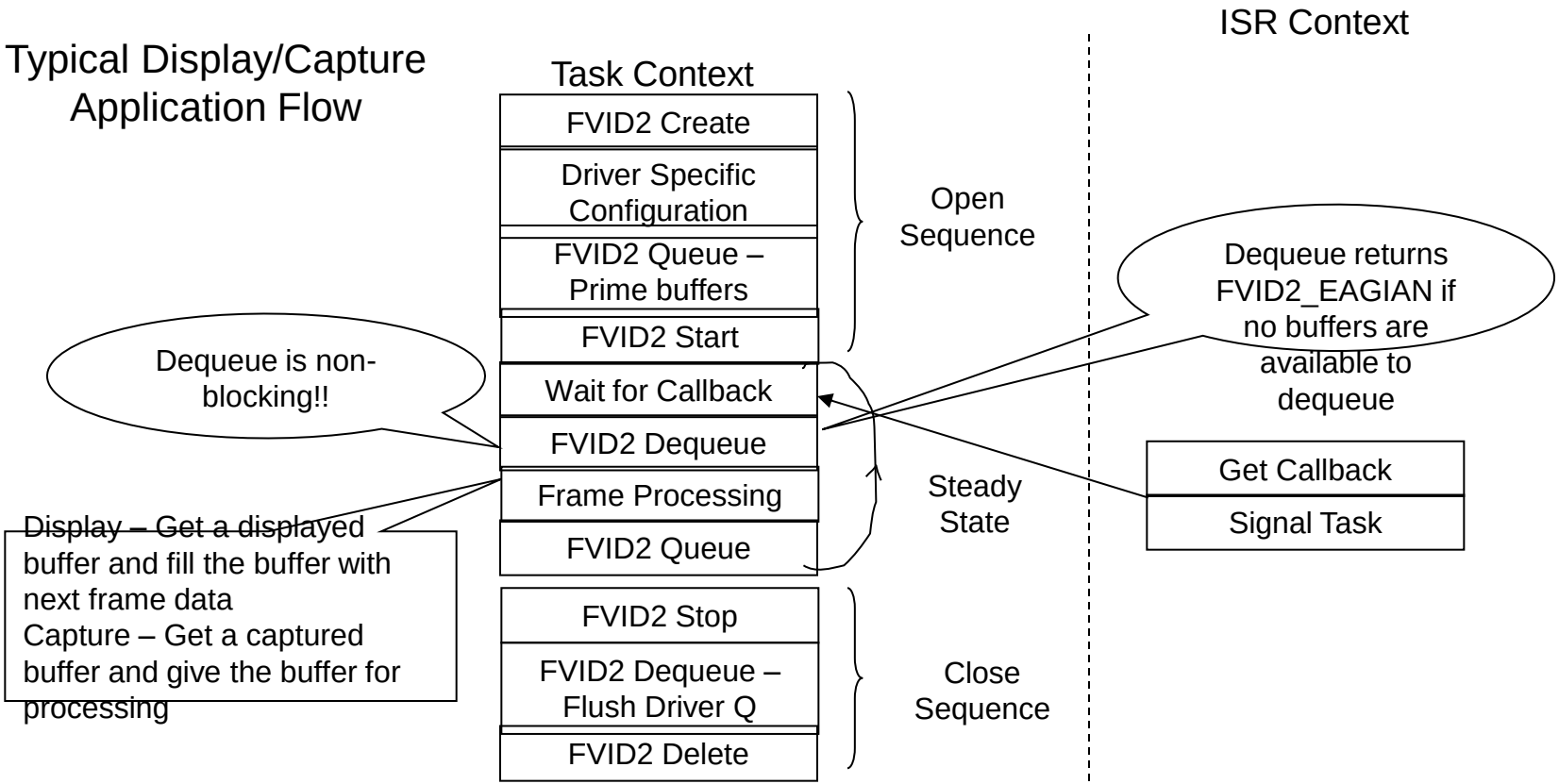


Understanding FVID2 - Interfaces

- FVID2_ProcessList
  - Represents M FVID2_FrameList for input and output frames
  - Each frame list represents a N frame buffers for each of the inputs and outputs in M2M drivers
  - Used only in M2M drivers

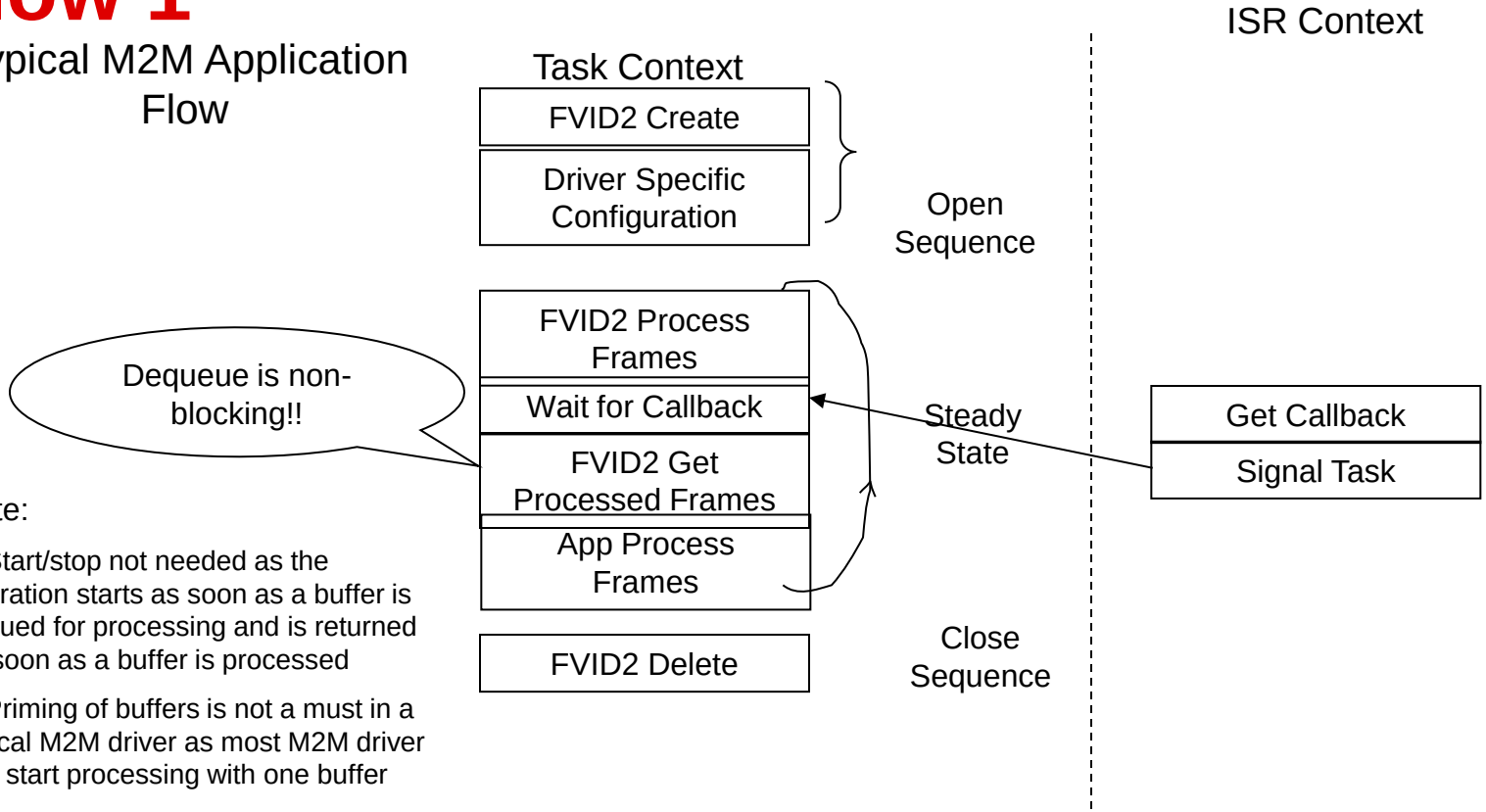


Understanding FVID2 – Application Flow 1



Understanding FVID2 – M2M Application Flow 1

Typical M2M Application Flow



Note:

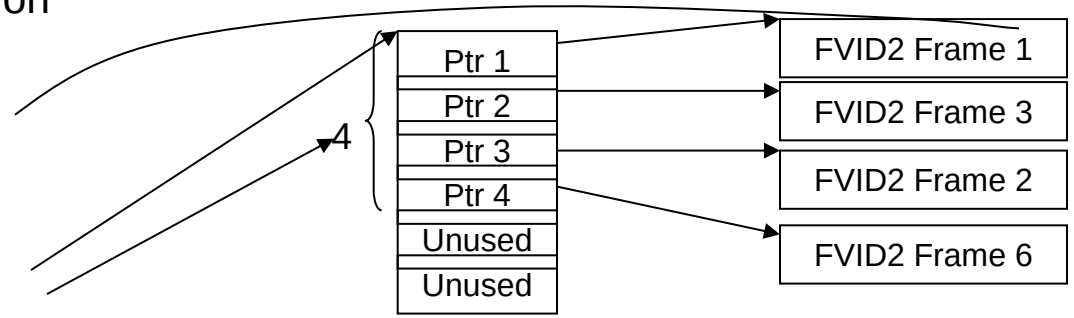
1. Start/stop not needed as the operation starts as soon as a buffer is queued for processing and is returned as soon as a buffer is processed

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 – Multiple Frames Per Request Feature

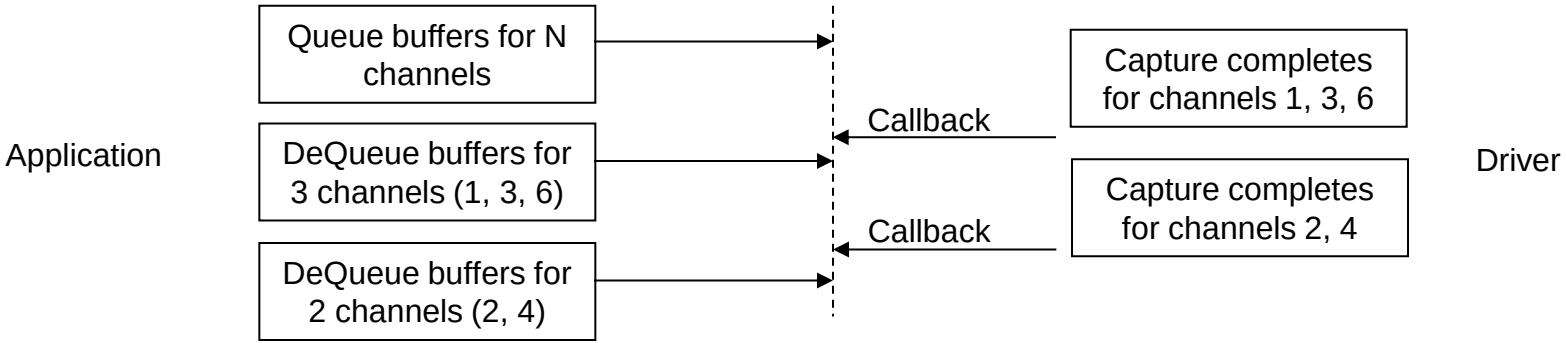
- This feature supports
  - Multi-window display (Multiple buffers belonging to one stream)
  - Multiplexed capture (Multiple buffers belonging to multiple streams)
  - Multiple M2M request per call (Multiple request belonging to multiple streams/channels)
- FVID2 Frame -> one buffer/request
- FVID2 FrameList -> Multiple buffers/requests
- FVID2_FrameList contains an array of FVID2_Frame pointers whose size if fixed at 64 in the current implementation

```
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;
```



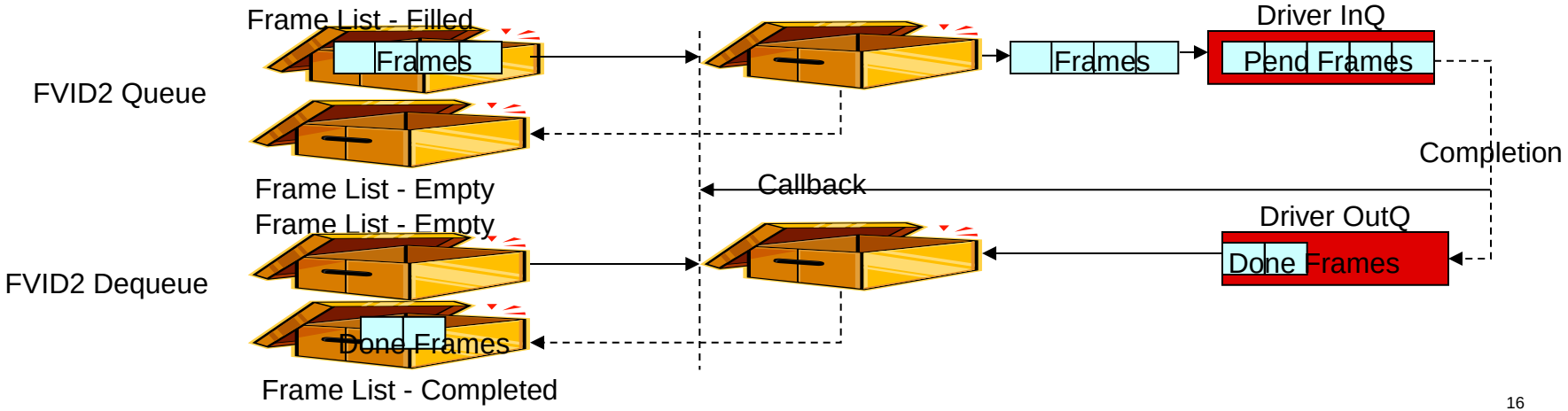
Understanding FVID2 – One Q, Multiple DQ

- Used in multiplexed capture
  - While priming, application submits buffers for all the channels using Queue call
  - Since multiplexed inputs could be asynchronous, capture could complete at different time for each of the inputs
  - Application wants to process the buffers as soon as they are captured
  - Hence they are de-queued immediately without waiting for other channels to complete
  - This will result in multiple dequeue for a single queue



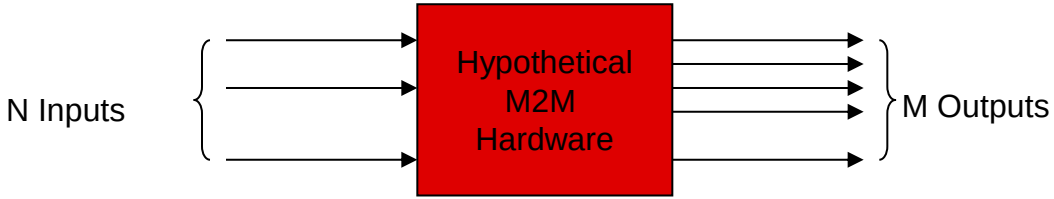
Understanding FVID2 – One Q, Multiple DQ Contd…

- How is this achieved?
  - Only frames are queued/dequeued in/from the driver
  - Frame list is not queued/dequeued
  - Frame list acts like a container to submit the frames to the driver in Queue call and take back the frames from the driver in dequeue call
  - For queue call, application is free to re-use the same frame list again without dequeuing
  - For dequeue call, the application has to provide the frame list to the driver and the driver copies the captured frame to the frame list



Understanding FVID2 – M2M Interface

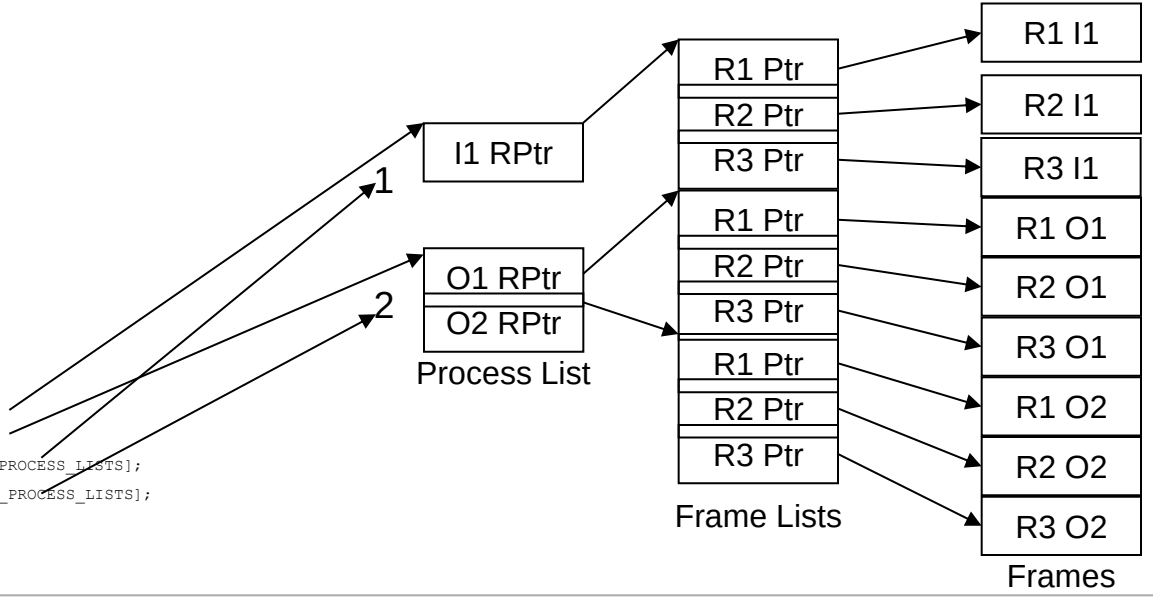
Key M2M Driver Requirements	Covered already by FVID2
Non-blocking	Callbacks supported ✓
Support multiple request per call	Frame List supports this ✓
Support multiple inputs and multiple outputs	Is covered now!! ✓



Understanding FVID2 – M2M Interface Contd…

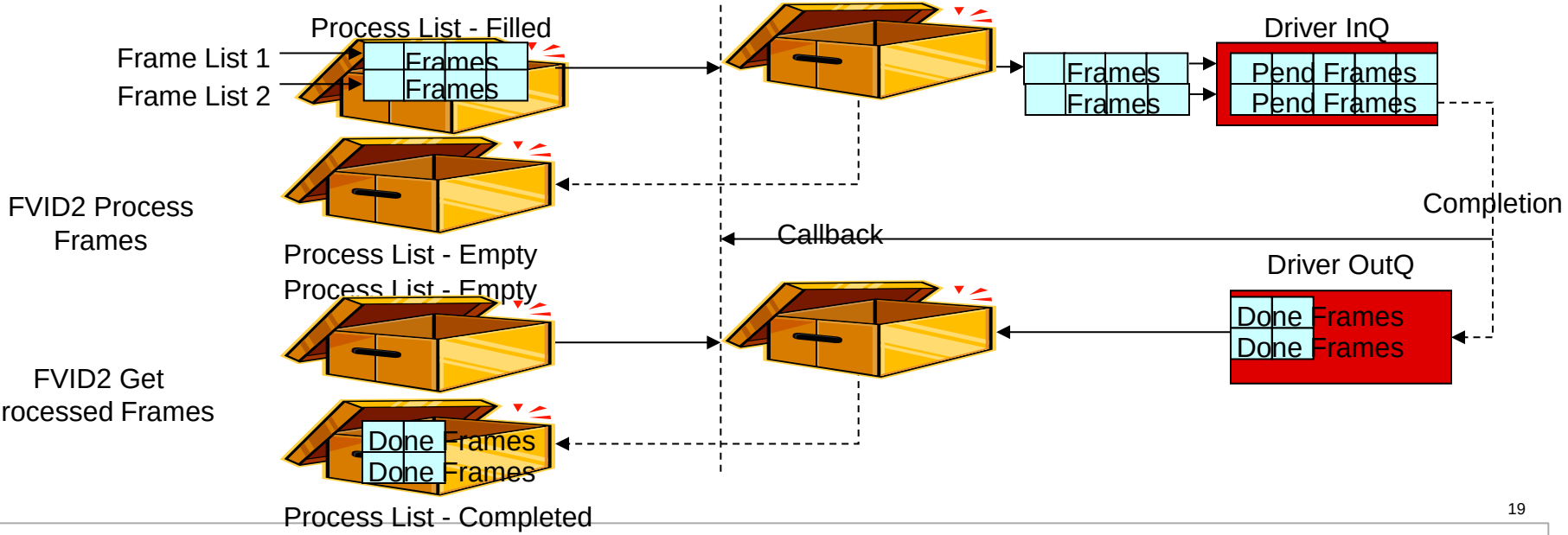
- FVID2 FrameList -> Multiple buffers/requests per input/output
- FVID2_ProcessList -> Multiple input and multiple output which in turn could contain multiple requests
- FVID2_ProcessList contains an array of FVID2_FrameList pointers for both input streams and output streams separately
- Two more new 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 Contd…

- 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



Understanding FVID2 – M2M Interface Contd…

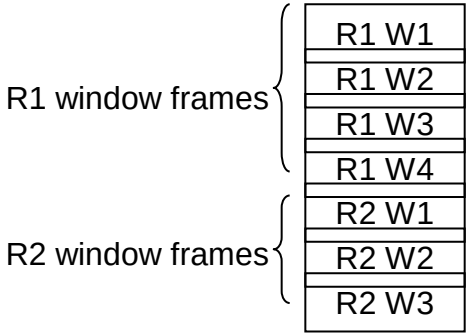
Multi-window Mode

- Process list -> One M2M request containing multiple frame list for each of the ip/op paths
- Frame list -> Has multiple frame pointers for the buffer pointers for each of the request
- Multi window mode -> Each frame could contain W windows
- Frame pointer could represent only one window
- Do we need another dimension/structure/dereferencing to obtain this?
- Proposed Solution:
  - No need to define another structure
  - Frame list will contain the frame pointers for all the windows of all the request

Understanding FVID2 – M2M Interface Contd…

Multi-window Mode

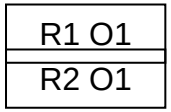
- Consider a memory driver which gets video from multiple buffers, composite, scales it and write it back to memory
- Assume two channels – 1st uses four windows and 2nd channel uses three
- Frame list for the above two request at one call looks like



In Frame List

numFrames = 7

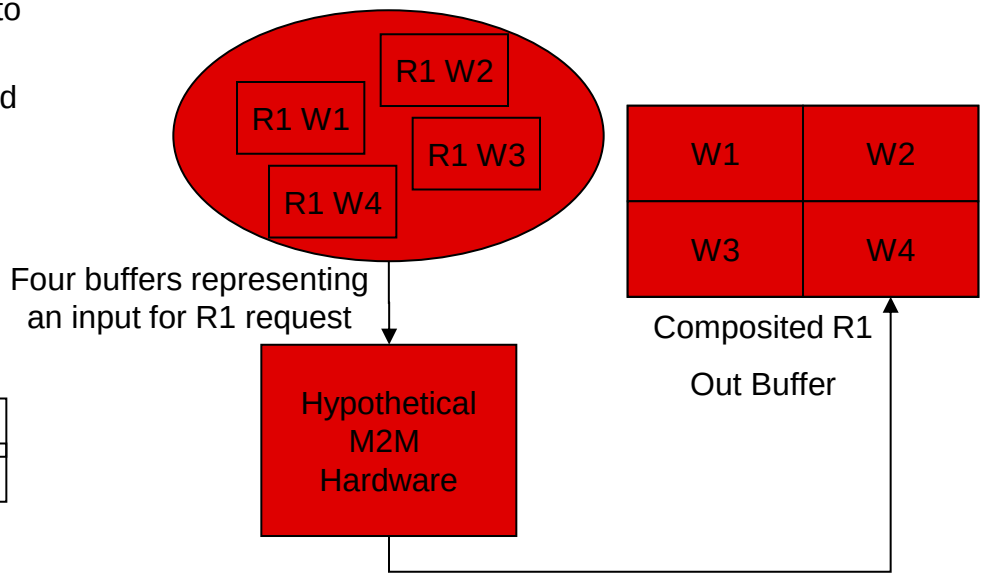
(4 + 3)



Out Frame List

numFrames = 2

(1+1)



Understanding FVID2 – M2M Interface Contd…

Multi-window Mode

- Advantage
  - No need extra structure or third dimension or another dereferencing
  - Application can reuse the same frame pointer structure from a different driver for the window buffers
- Other points
  - The order (left to right, top to bottom etc…) of the window buffers in a frame list is not defined by FVID2. The individual drivers can impose the order based on its implementation
  - Implicit splitting of window buffers based on configured multi-window mode for each channel (In previous example, 4 and 3 is not passed anywhere in the queue call!!)

Understanding FVID2 – Slice Based Operation

- Used in systems which need low latency video processing
- Each frame is split in to N number of equal size slices
- Typical low latency application might have 4/8 slices per frame
- Application needs to be intimated as soon as a slice is captured or processed

Understanding FVID2 – Slice Based Operation in M2M drivers

- Slice level processing can be enabled for each channel using create time configuration.
- Slices can be Queued as they are available using same frame processing API.
- Slice information has to be updated for each slice inside ‘FVID2_Frame’ under ‘FVID2_SliceInfo’.

```
typedef struct
  {
      UInt32              sliceNum;
      /**< Current slice Number in this frame,
           range is from 0 to (NoOfSlicesInFrame-1)  */
      UInt32              numSlcInLines;
      /**< Number of lines available in the frame at the end of this slice. */
      UInt32              numSlcOutLines;
      /**< Number of lines generated in output buffer after processing
           current slice */
  } FVID2_SliceInfo;
```

- M2M driver will process the given slice and updates output info for further processing in chain. Simultaneously Capture or Video Decoder may add subsequent slice to the frame.
- M2M driver keeps a copy of frame info and works using that to avoid conflict with other driver updating this data.
- Slice processing completion call back is similar to Frame based processing.

Understanding FVID2 – Slice Based Operation in Capture Driver

- Uses a separate callback mechanism compared to the FVID2 completion callback
- Callback gives reference to the current frame being captured/processed
- Slice callback will be called per channel per stream of the driver instance
- Prototype of Slice based callback

```
typedef Int32 (*FVID2_SliceCbFxn) (FVID2_Handle handle,
                                   FVID2_Frame *Frame);
```

Understanding FVID2 – Error Callback

- Queue operations are asynchronous and most probably returns success even though there are some error in the submitted frames
- Errors are checked in the FVID2 frames only at the time of processing of the frames – which might occur at a later point of time
- Also there could be asynchronous hardware errors
- So there should be some mechanism through which errors are returned to the application asynchronously
- Hence error callbacks are used

Understanding FVID2 – Error Callback Contd…

- When error occurs driver needs a free container (frame list in case of display/capture drivers or process list in case of M2M drivers) to return the queued buffers back to the application
- Hence at the time of channel creation, application gives an empty container to the driver along with application error handler
- This container will be used at the time of error callback
- Prototype of error callback

```
typedef Ptr (*FVID2_ErrCbFxn) (FVID2_Handle handle,
                               Ptr appData,
                               Ptr errData,
                               Ptr reserved);
```

Application can return the same container or a different container depending on whether the application empties the container in the error callback or not

Points to either a frame or process list

Benefits of new FVID2 interface

- It address all concerns/challenges discussed for the existing video driver interface
- There is no other interface which allows multi channel capture/ multi window display along with memory operation
- This helps in improving performance because
  - It does not requires data structure to be populated or modified when application is working in chain as capture -> Noise filter-> Deinterlace->scale->display
  - It makes application simpler to be developed and debugged
  - It has increased ease of use for API

Questions?
Thank You