Zynq UltraScale+ MPSoC Base Targeted Reference Design

User Guide

UG1221 (v2016.3) December 15, 2016

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/29/2016	2016.1	Initial Xilinx release.
07/22/2016	2016.2	Updated for Vivado Design Suite 2016.2: Added "GPU" to hardware interfaces and IP under Key Features. Changed link under Design Modules from the wiki site to the HeadStart Lounge and updated link under Tutorials to the Base TRD wiki site. Deleted steps 2 and 4 under Tutorials and added reference tutorial (last bullet). Added second to last sentence to second paragraph under Boot Process. Added "Load PMU FW" component to Figure 6-1. Clarified Message Passing section (text only). Changed "PCA9546" to PCA9548" in Figure 8-7.
12/15/2016	2016.3	Updated for Vivado Design Suite 2016.3: Updated Reference Design Overview. Replaced Chapter 2, Reference Design. Updated Figure 3-1 and the following paragraph. Updated Figure 4-1 and `perfapm` library descriptions in Chapter 4, RPU-1 Software Stack (Bare-metal). Updated Figure 6-1, Figure 6-2, and DDR region descriptions under Memories in Chapter 6. Updated Figure 7-1, Figure 7-4, and Figure 7-5. Added X11 section, deleted first paragraph under EGLFS QPA, and modified "Evdev" section to "Libinput" in Chapter 7, APU Software Platform. Updated Table 8-2 and clock descriptions under Clocks, Resets and Interrupts in Chapter 8.

XILINX
ALL PROGRAMMABLE™

Table of Contents

Revision History 2

Chapter 1: Introduction

Zynq UltraScale+ MPSOC Overview 6
Reference Design Overview 7
Key Features 9

Chapter 2: Reference Design

Design Modules 12
Design Components 14

Chapter 3: APU Application (Linux)

Introduction 16
GUI Application 18
Video Library 21
2D Filter Plug-in 26
Performance Monitor Client Library 28

Chapter 4: RPU-1 Software Stack (Bare-metal)

Introduction 29
Performance Monitor Library 30
Performance Monitor Applications 32
Bare-metal BSP 32

Chapter 5: RPU-0 Software Stack (FreeRTOS)

Introduction 33
Heartbeat Application 34
FreeRTOS BSP 34

Chapter 6: System Considerations

Boot Process 35
Global Address Map 37
Video Buffer Formats 39
Performance Metrics 41

Chapter 7: APU Software Platform

Introduction 45
Video 47
Display 51
Graphics 55
Vision 58
Inter-process Communication 59

Chapter 8: Hardware Platform

Introduction 63
Video Pipelines 64
Clocks, Resets and Interrupts 70
I2C Bus Topology 72
Auxiliary Peripherals 72

Appendix A: Additional Resources and Legal Notices

Xilinx Resources 74
Solution Centers 74
References 74
Please Read: Important Legal Notices 75

Introduction

The Zynq® UltraScale+™ MPSoC base targeted reference design (TRD) is an embedded video processing application that is partitioned between the SoC's processing system (PS) and programmable logic (PL) for optimal performance. The design demonstrates the value of offloading computation intensive image processing tasks such as a 2D-convolution filter from the PS onto PL. The benefits achieved are two-fold:

1. Ultra HD video stream real-time processing at 30 frames per second.
2. Freed-up CPU resources for application-specific tasks.

This user guide describes the architecture of the reference design and provides a functional description of its components. It is organized as follows:

- This chapter provides a high-level overview of the Zynq UltraScale+ MPSoC device architecture, the reference design architecture, and a summary of key features.
- Chapter 2, Reference Design gives an overview of the design modules and design components that make up this reference design. It provides a link to the Base TRD wiki which contains design tutorials.
- Chapter 3, APU Application (Linux) describes the Linux software application running on the application processing unit (APU).
- Chapter 4, RPU-1 Software Stack (Bare-metal) describes the bare-metal software application and stack running on the second core of the real-time processing unit (RPU-1).
- Chapter 5, RPU-0 Software Stack (FreeRTOS) describes the FreeRTOS software application and stack running on the first core of the real-time processing unit (RPU-0).
- Chapter 6, System Considerations details system architecture considerations including boot flow, system address map, video buffer formats, and performance analysis.
- Chapter 7, APU Software Platform describes the APU software platform covering the middleware and operating system layers of the Linux software stack.
- Chapter 8, Hardware Platform describes the hardware platform of the design including key PS and PL peripherals.
- Appendix A, Additional Resources and Legal Notices lists additional resources and references.

Zynq UltraScale+ MPSOC Overview

The Zynq device is a heterogeneous, multi-processing SoC built upon the 16 nm FinFET process node from TSMC. Figure 1-1 shows a high-level block diagram of the device architecture and key building blocks inside the processing system (PS) and the programmable logic (PL).

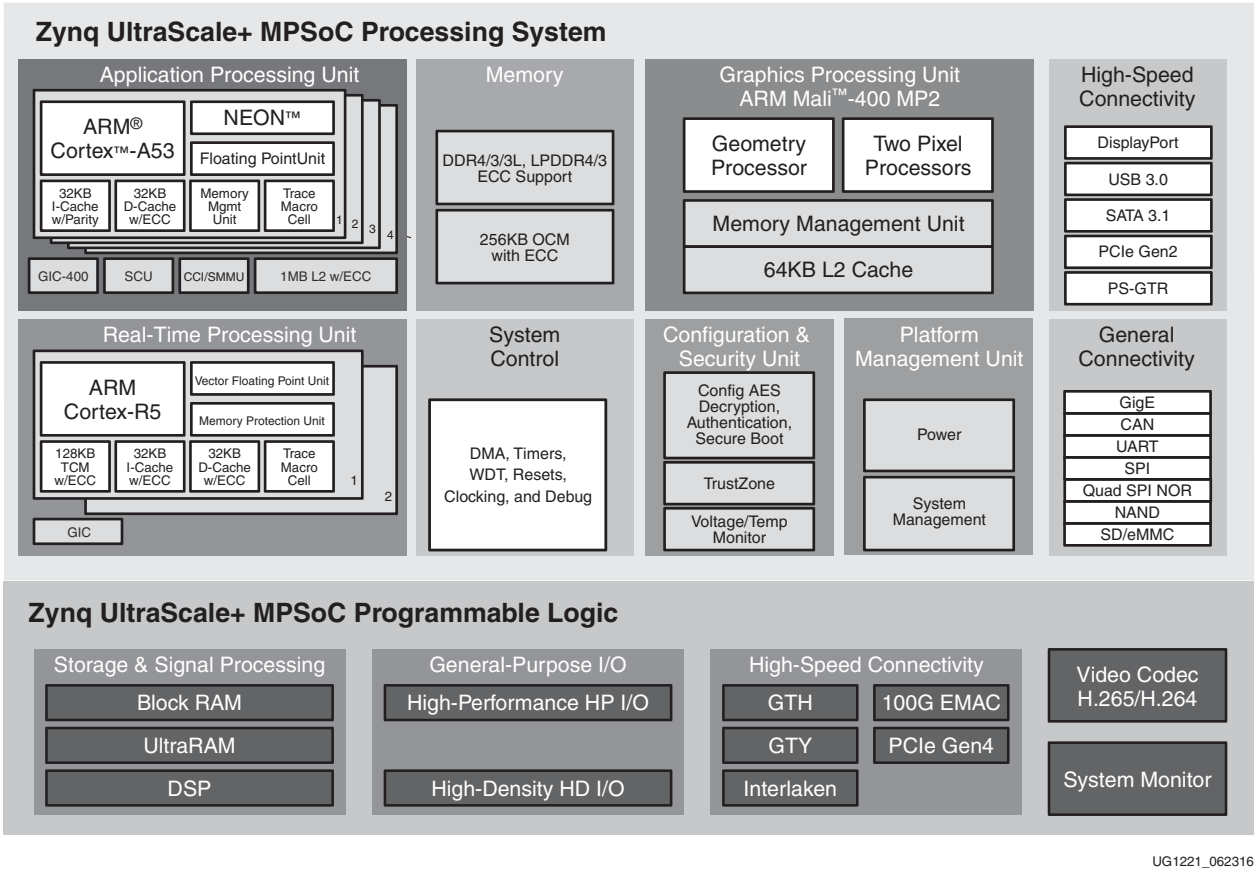

Figure 1-1: **Zynq UltraScale+ MPSoC Block Diagram**

The following summarizes the MPSoC's key features:

- Application processing unit (APU) with 64-bit quad-core ARM Cortex-A53 processor
- Real-time processing unit (RPU) with 32-bit dual-core ARM Cortex-R5 processor
- Multimedia blocks
  - Graphics processing unit (GPU), ARM Mali-400MP2
  - Video encoder/decoder unit (VCU) up to 4K 60 fps
  - DisplayPort interface up to 4K 30 fps

- High-speed peripherals
  - PCIe root complex (Gen1 or Gen2) and endpoint (x1, x2, and x4 lanes)
  - USB 3.0/2.0 with host, device, and OTG modes
  - SATA 3.1 host
- Low-speed peripherals
  - Gigabit Ethernet, CAN, UART, SPI, Quad-SPI, NAND, SD/eMMC, I2C, and GPIO
- Platform management unit (PMU)
- Configuration security unit (CSU)
- 6-port DDR controller with ECC, supporting x32 and x64 DDR4/3/3L and LPDDR4/3

Reference Design Overview

The MPSoC device has a heterogeneous processor architecture. The TRD makes use of multiple processing units available inside the PS using the following software configuration:

- The application processing unit (APU) consists of four ARM Cortex-A53 cores configured to run in SMP (symmetric multi-processing) Linux mode. The application's main task is to configure and control the video pipelines via a graphical user interface (GUI). It also communicates with one of the RPU cores to visualize system performance.
- The real-time processing unit (RPU) consist of two ARM Cortex-R5 cores configured to run in split mode (asymmetric multi-processing).
  - RPU-0 is configured to run FreeRTOS, an embedded real-time operating system. It is booted very early in the boot process to enable execution of safety/security-critical or real-time applications before the rest of the system is booted (see Boot Process, page 35). The application is a simple multi-threaded heartbeat example that continuously prints to the UART showing that the system is alive. It serves as a placeholder for user created applications.
  - RPU-1 is configured to run bare-metal. It is booted as a slave to the APU which downloads the firmware code after Linux has booted and provides performance monitoring capabilities by reading the AXI performance monitors (APM) inside the PS. RPU-1 communicates with the APU via the Xilinx® OpenAMP framework (see Inter-process Communication, page 59 for details on inter-process communication).

Figure 1-2 shows the software state after the boot process has completed and the individual applications have been started on the target processing units. The TRD does not make use of virtualization and therefore does not run a hypervisor on the APU.

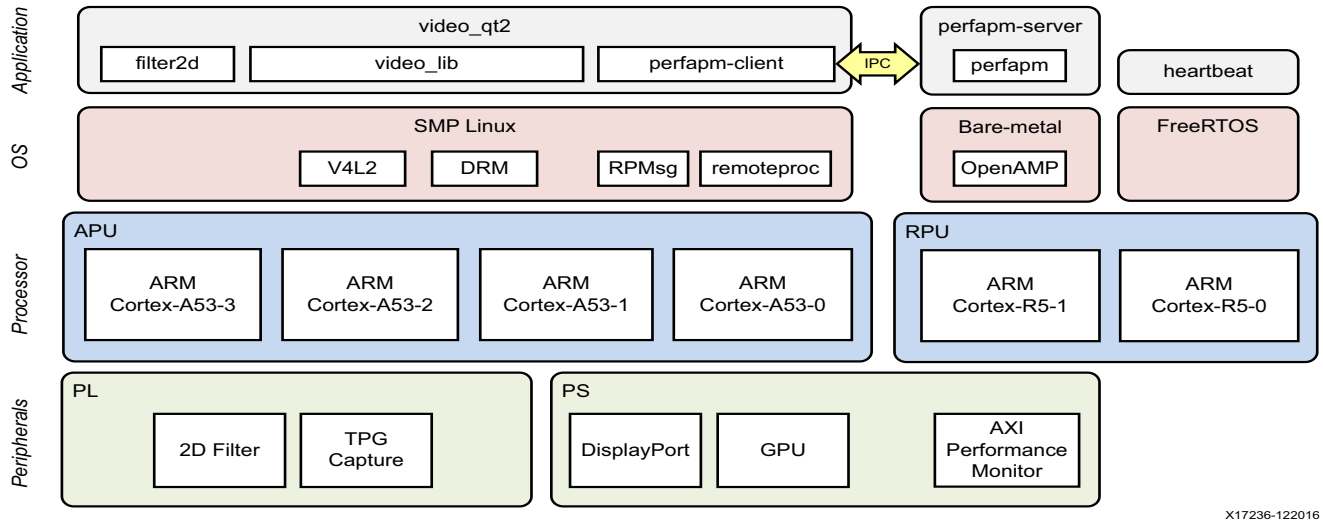

Figure 1-2: **Key Reference Design Components by Processing Unit**

The APU application controls the following video data paths implemented in a combination of PS and PL (see Figure 1-3):

- Capture pipeline capturing video frames from a test pattern generator (TPG) implemented inside the PL into DDR memory.
- Memory-to-memory (M2M) pipeline implementing a typical video processing algorithm, in this case a programmable 2D convolution filter. The algorithm can be implemented inside the PS as software function or as hardware accelerator inside the PL. Video frames are read from DDR memory, processed by the software function or accelerator, and then written back to memory.
- Display pipeline implemented inside the PS reading video frames from memory and sending them to a monitor via the DisplayPort. The display pipeline supports two layers: one for video, the other for graphics. The graphics layer is rendered by the GPU.

RPU-1 reads performance metrics from the AXI performance monitor (APM) and sends the data to the APU via IPC using shared virtual ring (vring) buffers in DDR memory. The same DDR memory is shared between the APU, RPU, and GPU.

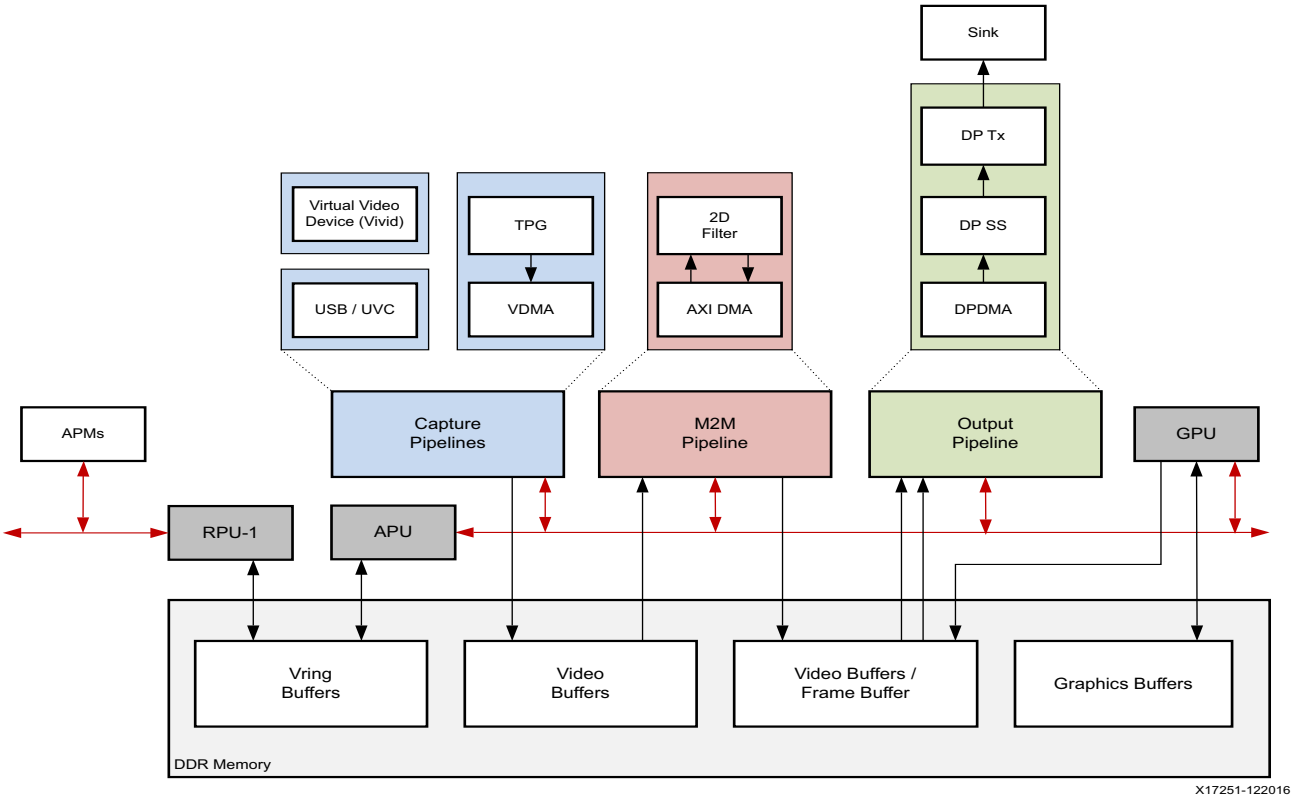

Figure 1-3: **Base TRD Block Diagram**

Key Features

The following summarizes the TRD's key features.

Target platforms and extensions:

- ZCU102 evaluation board (see *ZCU102 Evaluation Board User Guide* (UG1182) [Ref 3])

Xilinx tools:

- SDSoC
- PetaLinux Tools
- Xilinx SDK
- Vivado Design Suite

Hardware interfaces and IP:

- GPU
- Video Inputs
  - TPG
  - USB Webcam (optional)
  - Vivid (Virtual video device)
- Video Outputs
  - DisplayPort Tx
- Video Processing
  - 2D Convolution Filter
- Auxiliary Peripherals
  - SD
  - I2C
  - GPIO
  - Ethernet
  - UART
  - USB 2.0
  - APM

Software components:

- Operating systems
  - APU: SMP Linux
  - RPU-0: FreeRTOS
  - RPU-1: Bare-metal
- Linux frameworks/libraries:
  - Video: Video4Linux (V4L2), Media Controller
  - Display: DRM/KMS, X-Server (X.Org)
  - Graphics: Qt5, OpenGL ES2
  - Vision: OpenCV
  - Inter-process communication: OpenAMP

- User applications:
  - APU: Video control application with GUI
  - RPU-0: Multi-threaded heartbeat application
  - RPU-1: Performance monitoring application

Supported video formats:

- Resolutions:
  - 720p60
  - 1080p60
  - 2160p30
- Pixel formats:
  - YUV 4:2:2 16 bit for video
  - ARGB 32 bit for graphics

Reference Design

Design Modules

The Base TRD consists of nine design modules (DM) that build on top of each other or are a combination or previous modules where DM9 is the full-fledged reference design as summarized in the previous chapter. The following listing gives a short summary of each of the nine design modules:

DM1 - APU SMP Linux

This module shows how to build and run a SMP Linux image for the APU. The Linux image boots a serial console on UART0. The Linux rootfs is pre-configured with selected open-source libraries and applications such as `bash` or `vi`. The Linux image enables Ethernet and USB host mode, and supports external file systems on SD, USB or SATA.

DM2 - RPU0 FreeRTOS Application

This module shows how to build and run a FreeRTOS application on RPU0. The application periodically prints messages to UART1 to demonstrate continuous operation.

DM3 - RPU1 Bare-metal Application

This module shows how to build and run a bare-metal application on RPU1. The application implements a performance monitor that measures DDR throughput and prints the results to UART1.

DM4 - APU/RPU1 Inter Process Communication

This module combines DM1 and DM3. The RPU1 (remote) application is booted by the APU (master) using the `remoteproc` framework. Both processors communicate via IPC using the RPMsg (APU) and OpenAMP (RPU1) frameworks. The performance monitor server running on RPU1 sends data to a client running on the APU. The client prints the received data to UART0.

DM5 - APU GUI Application

This module shows how to build and run an application with graphical user interface (GUI) on the APU using the Qt toolkit with OpenGL acceleration by the GPU. The application demonstrates how to capture video from a virtual video device (vivid) or optionally USB webcam and display it on the monitor through DP.

DM6 - PL Video Capture

This module extends DM5 and shows how to capture video from a test pattern generator (TPG) implemented in the PL.

DM7 - Image Processing using OpenCV

This module extends DM6 and shows how to implement an image processing algorithm running on the APU using the OpenCV libraries. A simple 2D convolution filter is used as an example.

DM8 - PL Acceleration

This module extends DM7 and shows how to refactor the image processing function so it can be accelerated in the PL. The SDSoC tool allows to estimate the performance increase, use high-level synthesis (HLS) to create RTL from a C algorithm, and automatically insert data movers along with the required drivers.

DM9 - Full-fledged Base TRD

This module combines DM2, DM4 and DM8. The data provided by the performance monitor server running on RPU1 is now plotted on the Qt GUI. In parallel, the FreeRTOS application is continuously printing messages on UART1 without interfering with RPU1 or the APU. The following metrics are visualized to compare the performance between the OpenCV algorithm (DM7) and the PL accelerator (DM8): CPU utilization, memory bandwidth, and frame rate.

Table 2-1 shows for each design module (rows), which other modules (columns) it builds upon or is a combination of.

Table 2-1: **Design Module Dependency Matrix**

	DM1	DM2	DM3	DM4	DM5	DM6	DM7	DM8
DM1								
DM2								
DM3								
DM4	+		+					
DM5	+							
DM6	+				+			
DM7	+				+	+		
DM8	+				+	+	+	
DM9	+	+	+	+	+	+	+	+

The Base TRD wiki at http://www.wiki.xilinx.com/Zynq+UltraScale+MPSoC+Base+TRD provides additional content including:

- Prerequisites for building and running the reference design
- Instructions for running the pre-built SD card image on the evaluation board
- Detailed step-by-step design and tool flow tutorials for each design module

Design Components

The reference design zip file can be downloaded from the *Documentation & Designs* tab at: https://www.xilinx.com/products/boards-and-kits/zcu102. The file contains the following components grouped by target processing unit or PL:

- APU
  - `perfapm-client`: Library that receives data from the `perfapm-server` application running on RPU1 using the RPMsg framework.
  - `perfapm-client-test`: Application that uses the `perfapm-client` library and prints the received performance numbers on UART0.
  - `petalinux_bsp`: PetaLinux board support package (BSP) to build a pre-configured SMP Linux image for the APU. The BSP includes the following components: first-stage boot loader (FSBL), ARM trusted firmware (ATF), u-boot, Linux kernel, device tree, and root file system (`rootfs`). It largely serves as the SDSoC software platform.
  - `filter2d`: Library and SDSoC example design that implements an image processing algorithm using the OpenCV library or the hardware optimized `hls_video` library.
  - `video_lib`: Library that manages the video capture, processing, and display pipelines using the V4L2 and DRM frameworks.
  - `video_qt2`: Application that uses the `video_lib` and `filter2d` libraries and provides a GUI to control and visualize various parameters of this design.
- PL
  - `vivado`: Vivado IP Integrator design that implements the TPG capture pipeline in the PL. It serves as the SDSoC hardware platform which inserts accelerator and data movers into this design.
- PMU
  - `pmu_fw`: Firmware that monitors and manages power states.

- RPU0
  - `heartbeat`: Application that periodically prints messages to UART1 to demonstrate continuous operation.
- RPU1
  - `perfapm`: Library that reads performance numbers from AXI performance counters (APM) inside the PS.
  - `perfapm-ctl`: Application that uses the `perfapm` library and prints the performance numbers on UART1.
  - `perfapm-server`: Application that uses the `perfapm` library and sends data to the `perfapm-client` library running on the APU using the OpenAMP framework.

Table 2-2 lists all the design components delivered with the TRD and what components are used for each design module.

Table 2-2: **Design Component to Design Module Mapping**

Design Component	Design Module								
	1	2	3	4	5	6	7	8	9
apu/perfapm-client/perfapm-client				√					√
apu/perfapm-client/perfapm-client-test				√					
apu/petalinux_bsp	√			√	√	√	√	√	√
apu/video_app/video_lib					√	√	√	√	√
apu/video_app/video_qt2					√	√	√	√	√
apu/zcu102_base_trd/samples/filter2d							√	√	√
apu/zcu102_base_trd/hw/vivado						√			
pmu/pmu_fw	√	√	√	√	√	√	√	√	√
rpu0/heartbeat		√							√
rpu1/perfapm-server/perfapm			√	√					√
rpu1/perfapm-server/perfapm-ctl			√						
rpu1/perfapm-server/perfapm-server				√					√

APU Application (Linux)

Introduction

The APU Linux software stack is divided into an application layer and a platform layer. The application layer is purely implemented in Linux user-space whereas the platform layer contains middleware (user-space libraries) and operating system (OS) components (kernel-space drivers etc.). Figure 3-1 shows a simplified version of the APU Linux software stack. This chapter focuses on the application layer implemented in user-space. Chapter 7, APU Software Platform, describes the platform layer.

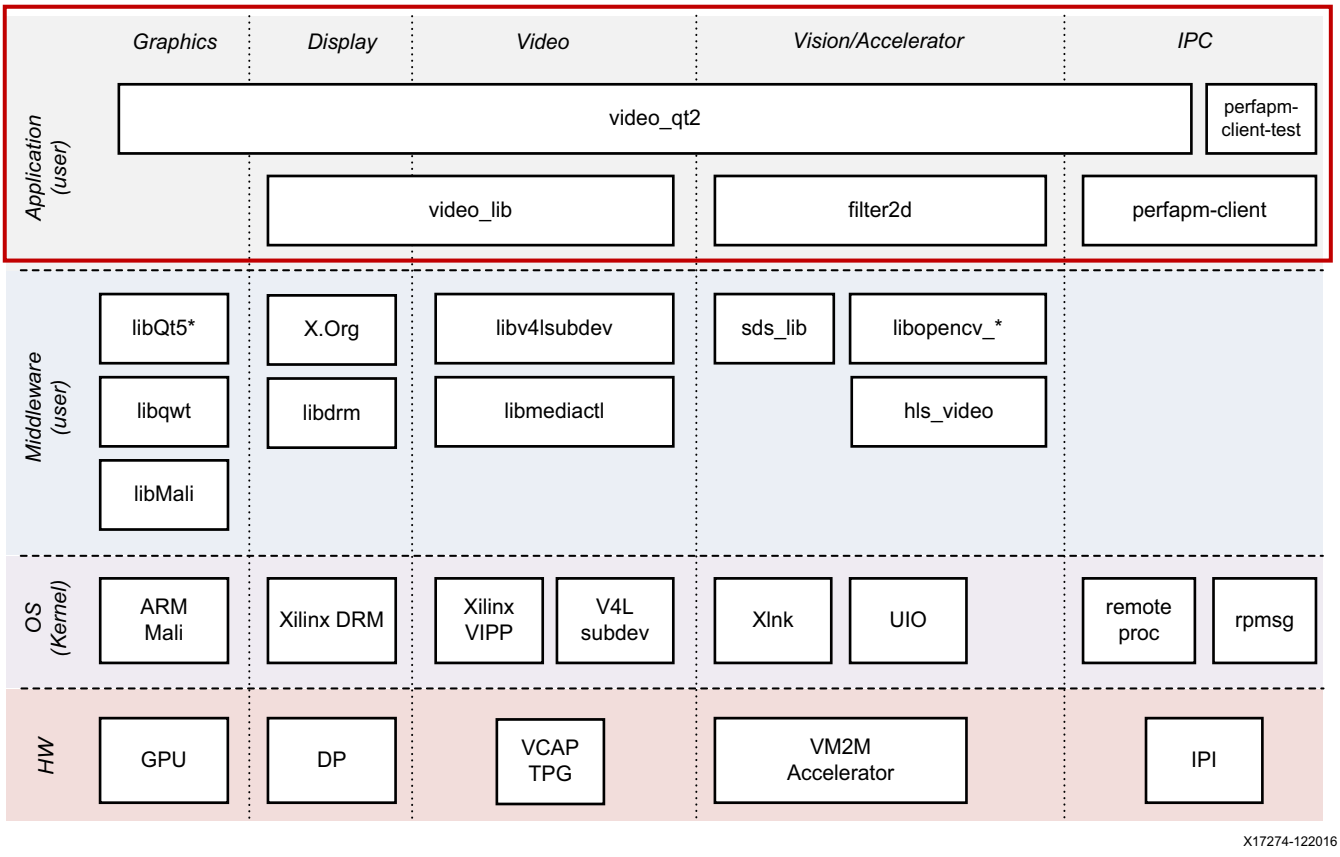

Figure 3-1: **Linux Software Stack and Vertical Domains**

Two APU applications are provided. The first application `perfapm-client-test` is a simple test application that builds on top of the `perfapm-client` library and prints the performance numbers received from the `perfapm-server` application running on RPU-1 on UART0. The second application `video_qt2` is a multi-threaded Linux application with the following four main tasks:

- Display unprocessed video from one of the sources
- Apply processing function in either software or hardware
- Provide a GUI for user input
- Interface with lower level layers in the stack to control video pipeline parameters and video data flow

The application consists of multiple components that have been specifically developed for the Base TRD and are explained in more detail in the following:

- GUI application (`video_qt2`)
- Video library (`video_lib`)
- 2D filter plug-in (`filter2d`)
- Performance monitor client library (`perfapm-client`)

Figure 3-2 shows a block diagram of the application interfaces between the individual components.

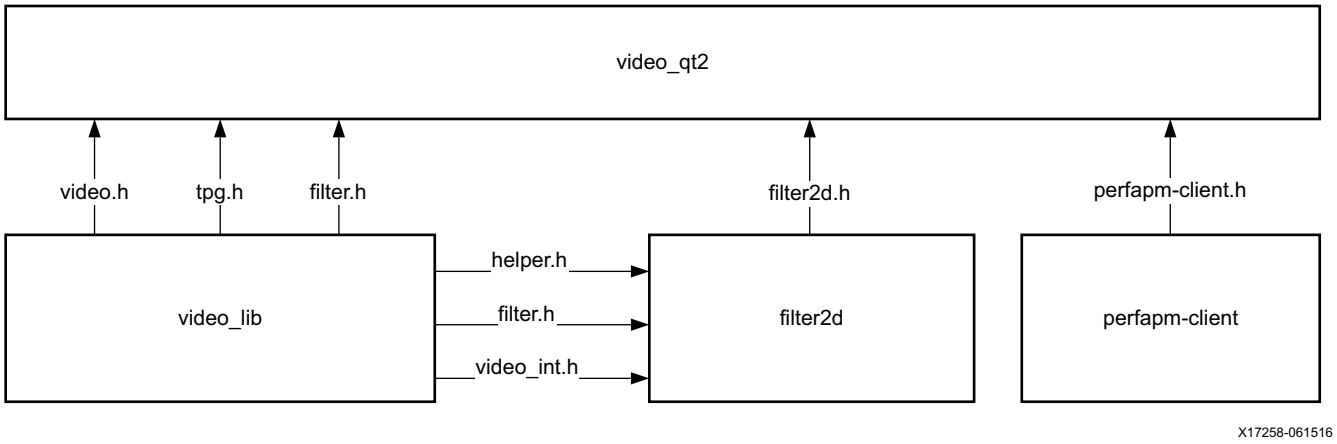

Figure 3-2: **Video Application Interfaces**

GUI Application

The `video_qt2` application is a multi-threaded Linux application that uses the Qt graphics toolkit to render a graphical user interface (GUI). The GUI provides control knobs for user input and a display area to show the captured video stream. The following resolutions are supported: 720p, 1080p and 2160p.

The GUI shown in Figure 3-3 contains the following control elements displayed on top of the video output area:

- Control bar (bottom)
- Video info panel (top-left)
- System performance panels (top-right)

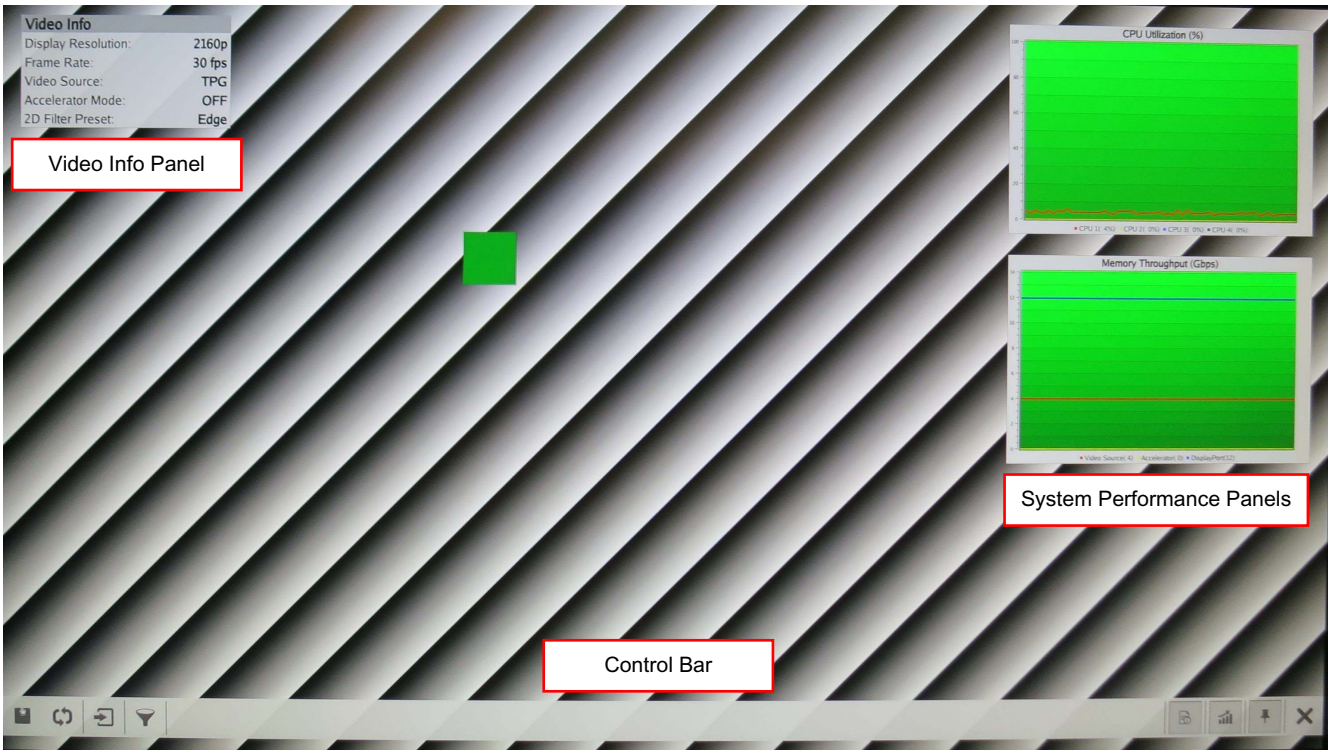

Figure 3-3: **Control Elements Displayed on Top of the Video Output Area**

Control Bar

The control bar is displayed at the bottom of the screen. It contains the following control items, from left to right:

- Start/stop video stream and demo mode
- Video source selection and settings
- Accelerator mode selection and settings
- Show/hide video info panel
- Show/hide system performance panels
- Pin/un-pin control bar
- Exit application

Video Info Panel

The video info panel displays the following information:

- Display resolution
- Frame rate (in fps)
- Video source
- Accelerator mode
- 2D filter preset

System Performance Panels

Two panels for system performance monitoring are shown in the top-right corner in Figure 3-3. The view can be toggled between numerical and graph view upon mouse-click:

- The first panel prints or plots the CPU utilization for each of the four A53 cores inside the APU.
- The second panel prints or plots the memory throughput for accumulated AXI read and write transactions on the DDR controller ports connected to video capture, video processing (2D filter), and DisplayPort.

TPG Settings Panel

This panel (Figure 3-4) can be accessed from the video source settings icon. It provides the following control knobs:

- TPG pattern (color bar, zone plate, checkerboard, tartan bars, etc.)
- Motion speed
- Foreground overlay (moving box or cross hairs)

- Box color
- Box size
- Horizontal zone plate speed/delta
- Vertical zone plate speed/delta
- Cross hairs X/Y coordinates

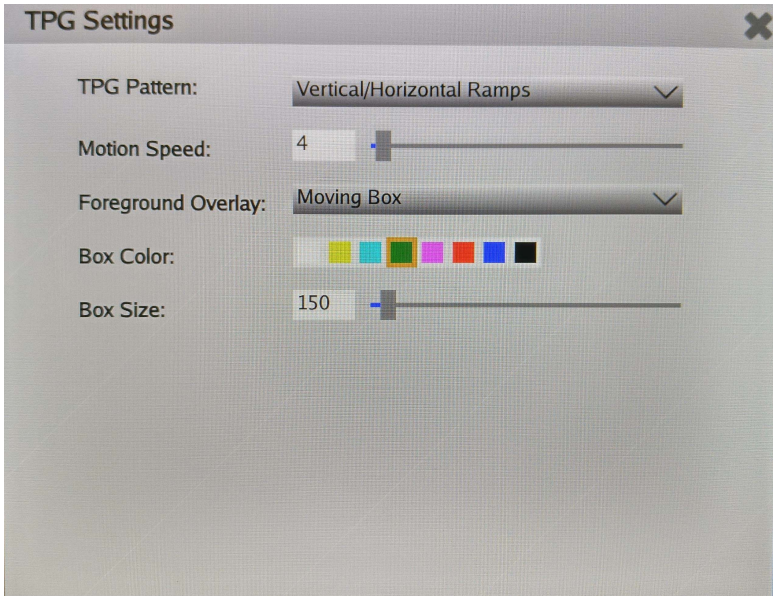

Figure 3-4: **TPG Settings Panel**

2D Filter Settings Panel

This panel (Figure 3-5) can be accessed from the filter mode settings icon. It provides the following control knobs:

- Filter presets (blur, sharpen, emboss, sobel, custom, etc.)
- Filter coefficients

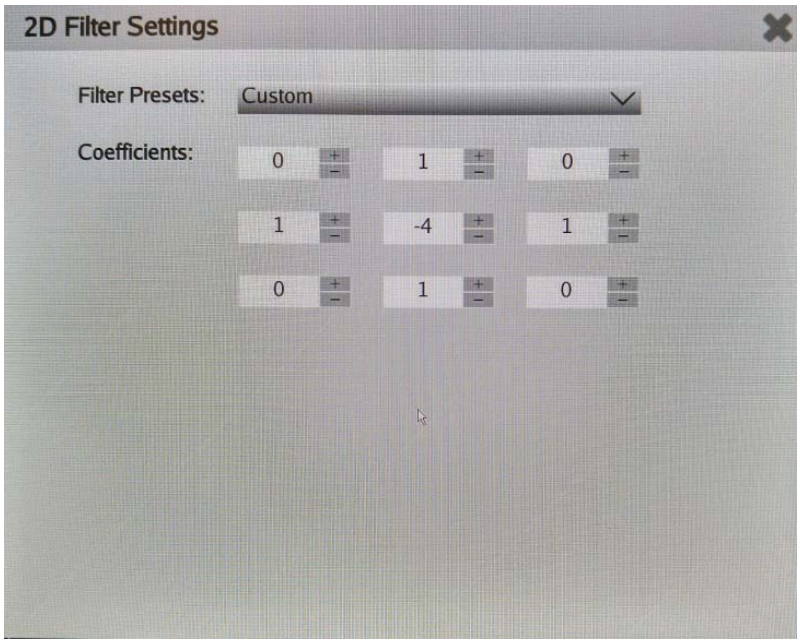

Figure 3-5: **2D Filter Settings Panel**

Video Library

The `video_lib` library configures various video pipelines in the design and controls the data flow through these pipelines. It implements the following features:

- Display configuration
- Media pipeline configuration for video capture
- Video pipeline control
- Video buffer management

The video_lib library exports and imports the following interfaces as shown in Figure 3-2, page 17:

- Video pipeline control (to GUI application)
- TPG video source controls (to GUI application)
- API for video accelerator plug-ins (to 2D filter)
- Interfaces from various middleware layers (V4L2, media controller, DRM)

Display Configuration

The `libdrm` library is used to configure the display pipeline and to send video buffers to the display controller. The library implements the following functionality:

- De-/initialize DRM module
- Set CRTC mode
- Set plane mode
  - Enable/disable plane
  - Set dimensions (width/height)
  - Set x/y-position
  - Set buffer index
- Set plane properties
  - Set transparency value
  - Enable/disable global alpha
  - Set z-position
- Create video buffers

The `video_lib` library configures the CRTC based on the monitor's EDID information with the video resolution of the display. It then sets the mode of the video plane to match the CRTC resolution and configures other plane properties. The graphics plane is configured by the Qt EGLFS backend outside of this library. The pixel format for each of the two planes is configured statically in the device-tree. Finally, the library requests buffers that are then exported using the DMABUF framework so they can be filled by a video capture device. The filled buffers are then sent to the video plane.

Media Pipeline Configuration

The video capture pipeline present in this design is: TPG input. It implements a media controller interface that allows you to configure the media pipeline and its sub-devices. The `libmediactl` and `libv4l2subdev` libraries provides the following functionality:

- Enumerate entities, pads and links
- Configure sub-devices
  - Set media bus format
  - Set dimensions (width/height)

The `video_lib` library sets the media bus format and video resolution on each sub-device source and sink pad for the entire media pipeline. The formats between pads that are connected through links need to match.

Video Pipeline Control

The V4L2 framework is used to capture video frames at the video device nodes. It provides the following functionality:

- Start/stop video stream
- Query capabilities
- Get/set controls
- Get/set pixel format
- Buffer operations
  - Request buffers
  - Queue/de-queue buffers

The `video_lib` library sets the pixel format on the video node (which corresponds to a DMA engine) from which data should be captured. Video buffers that are exported by the DRM device are queued at the V4L2 video node. If at least one buffer is queued, the user can start the video stream and capture video frames. When the user stops the video stream, any DMA in progress is aborted and any queued buffers are removed from the incoming and outgoing queues. You can set standard or custom controls on V4L sub-devices to change hardware parameters while the pipeline is stopped or running.

Video Buffer Management

The `video_lib` library uses the DMABUF framework (see Figure 3-6) for sharing buffers between a display device (DRM), a video capture device (V4L2), and a video processing accelerator (XLNK).

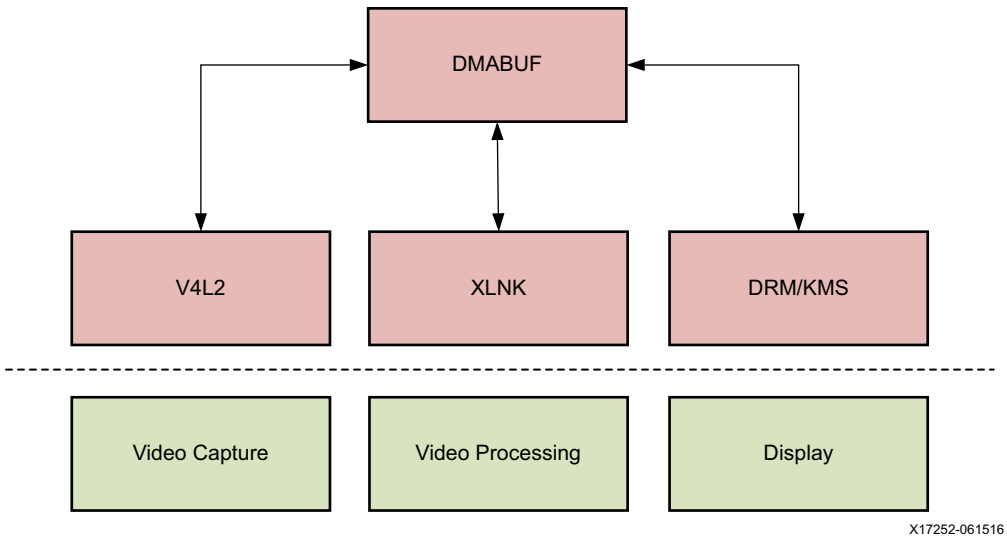

Figure 3-6: **DMABUF Sharing Mechanism**

The following steps are performed in which steps 3 through 8 are performed in a continuous loop (also referred to as *event loop*), (see Figure 3-7):

1. The DRM device driver allocates video buffers on behalf of the application, which are then exported to user-space as file descriptors.
2. The V4L2 capture device needs to be set to DMABUF streaming I/O mode using the VIDIOC_REQBUFS ioctl so the file descriptors can be imported without performing any direct allocation.
3. After assigning the buffer index and setting the exported buffer descriptor, empty buffers are queued to the video capture device's incoming queue by using the VIDIOC_QBUF ioctl.
4. The buffer is now locked so it can be filled by the hardware, typically a DMA engine. To start streaming i.e., to start capturing video frames into the queued buffers, the VIDIOC_STREAMON ioctl is used.
5. In the event loop, the application calls the poll() function on the file descriptor which waits until a buffer has been filled by the driver by setting the POLLIN flag.
6. The filled buffers can then be de-queued from the outgoing queue of the capture device using the VIDIOC_DQBUF ioctl.
7. The buffers are then passed to the DRM device to be displayed. The DRM driver reads from the same buffer until the next buffer is ready. Again, a poll() function is used, this time on the DRM side, to wait for a page-flip event. Page-flip typically occurs during the vertical blanking period i.e. between two video frames. It is a technique by which DRM configures a plane with the new buffer index to be selected for the next scan-out.
8. If the page-flip succeeds, the previous buffer is released, re-queued on the V4L2 side, and DRM stores the index of the new buffer. This way, buffers are continuously moved from capture to display device and back in a circular fashion.
9. Lastly, video streaming is turned off on the capture device using the VIDIOC_STREAMOFF ioctl. Apart from aborting or finishing any DMA in progress, it removes all buffers from the incoming and outgoing queues.

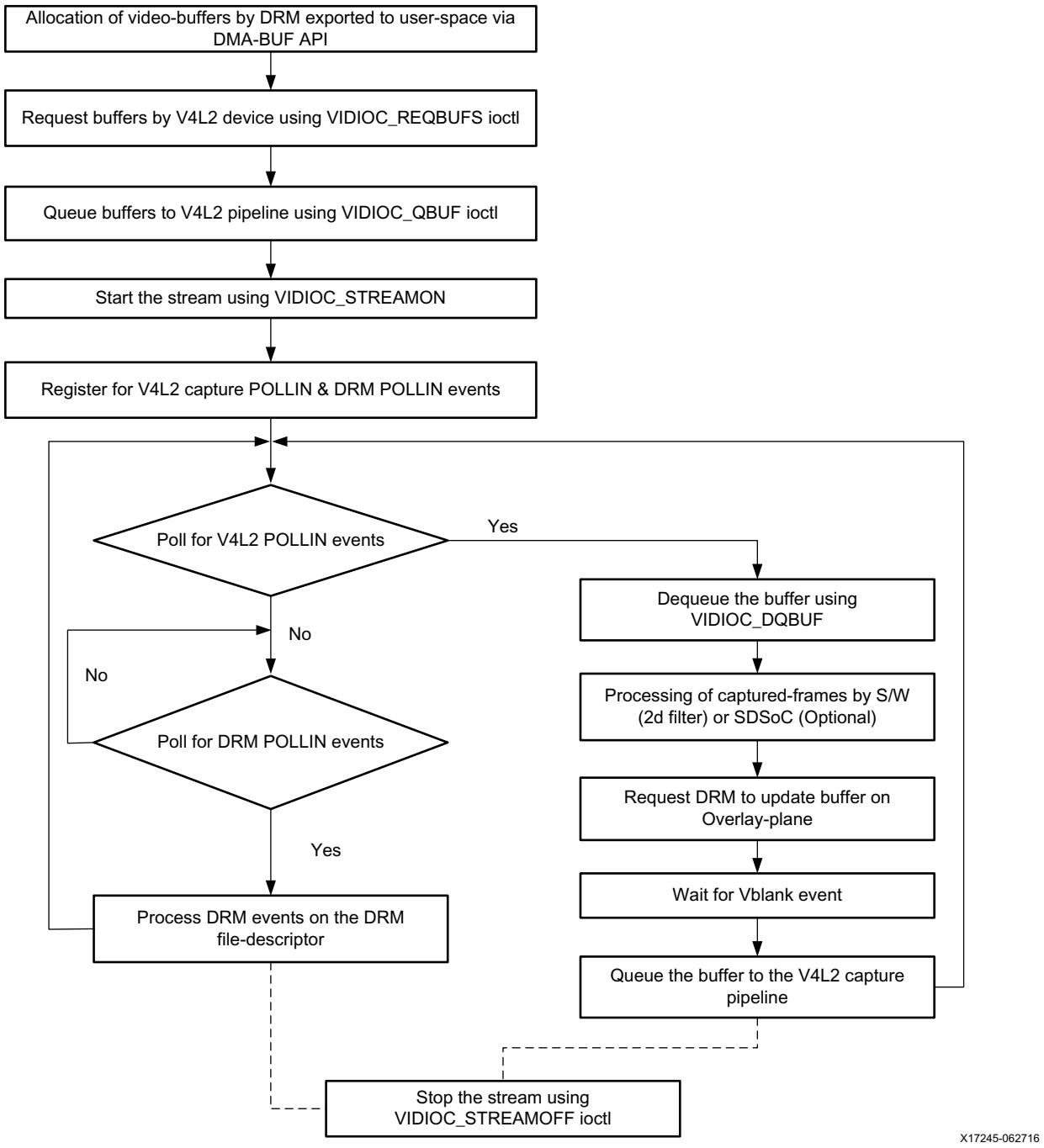

Figure 3-7: **Buffer Management Flow between V4L and DRM**

Alternatively, a software processing function or a hardware accelerator can be inserted between the capture and display pipelines. In this case, this is demonstrated with a 2D convolution filter.

Regardless whether the processing is done in software or hardware, the same function call interface is used. If the function is accelerated in hardware, the software function call is replaced by the SDSoC compiler with a *stub function* that intercepts the call and diverts the data flow into the PL instead of running the algorithm on the APU.

In both cases, the function consumes an input buffer from a capture device and produces an output buffer. It is important to note that the function call is blocking and only returns after the processing has finished and the output buffer has been written.

Video Accelerator Plug-in Interface

The `video_lib` library provides a simple and generic interface for accelerator plug-ins as defined in `filter.h`. Two functions need to be implemented by the accelerator:

1. The first function initializes the accelerator. Typically, default values are assigned to any user programmable parameters so the accelerator comes up in a well-defined state.
2. The second function implements the accelerator algorithm. Typically, this function consumes one or more input buffers and produces one output buffer. Additionally, key parameters that describe the video frame dimensions are passed i.e., width, height, and stride.

2D Filter Plug-in

In this example, a 2D convolution filter is implemented. Convolution is a common image processing technique that changes the intensity of a pixel to reflect the intensities of the surrounding pixels. This is widely used in image filters to achieve popular image effects like blur, sharpen, and edge detection [Ref 4].

The implemented algorithm uses a 3x3 kernel with programmable filter coefficients. The numbers inside the kernel determine how to transform the pixels from the original image into the pixels of the processed image (see Figure 3-8).

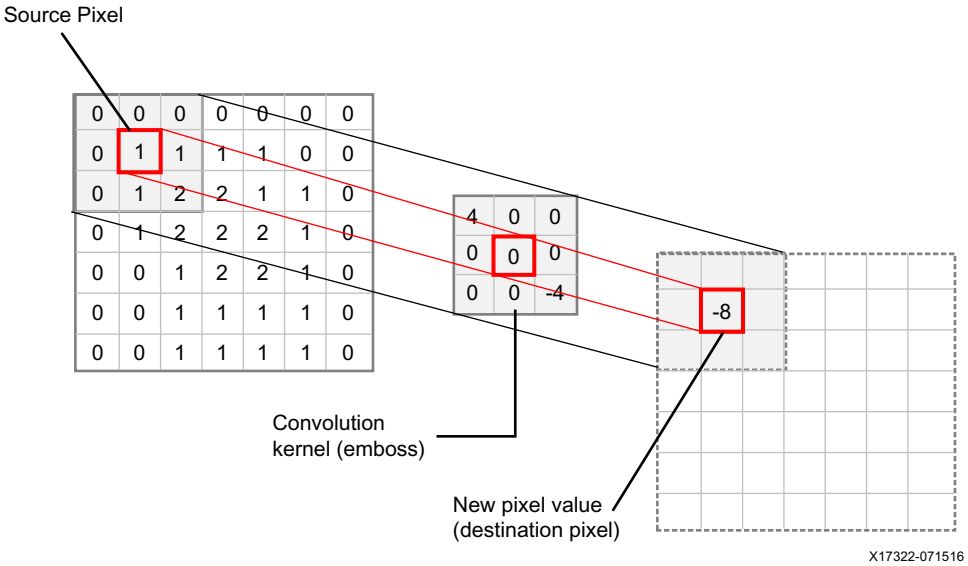

Figure 3-8: **2D Convolution with 3x3 Kernel**

The algorithm performs a two dimensional (2D) convolution for each pixel of the input image with a 3x3 kernel. Convolution is the sum of products, one for each coefficient/source pixel pair. As we are using a 3x3 kernel, in this case it is the sum of nine products.

The result of this operation is the new intensity value of the center pixel in the output image. This scheme is repeated for every pixel of the image in raster-scan order i.e., line-by-line from top-left to bottom-right. In total, width x height 2D convolution operations are performed to process the entire image.

The pixel format used in this design is YUYV which is a packed format with 16 bits per pixel (see Video Buffer Formats, page 39). Each pixel can be divided into two 8-bit components (or channels): one for luma (Y), the other for chroma (Cb/Cr alternating).

In this implementation, only the Y component is processed by the 2D convolution filter which is essentially a grayscale image. The reason is that the human eye is more sensitive to intensity than color. The combined Cb/Cr channel which accounts for the color components is merged back into the final output image unmodified.

In addition to implementing the generic interface, as described in Video Accelerator Plug-in Interface, page 26, this plug-in also exports a 2D filter specific interface that allows the user to program the kernel coefficients. This can be done either by setting the numerical values, or by selecting from a set of pre-defined *filter effects* such as blur, sharpen, edge detection etc.

The following OpenCV function is used for the software algorithm implementation [Ref 5]:

```
void filter2D(InputArray src, OutputArray dst, int ddepth,
              InputArray kernel, Point anchor=Point(-1,-1),
              double delta=0, int borderType=BORDER_DEFAULT)
```

The main function arguments are input image *src*, output image *dst*, and kernel *kernel*. The function consumes one frame, applies the provided kernel on the image and produces one output frame.

Before moving this function to the PL, the code needs to be re-factored so Vivado HLS can generate RTL as not all constructs used in the OpenCV function can be mapped to hardware. The Xilinx HLS video library contains an equivalent function to the filter2d OpenCV function which can be compiled into optimized hardware (see *Vivado Design Suite User Guide - High-Level Synthesis* (UG902) [Ref 6] and *Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries* (XAPP1167) [Ref 7]):

```
void Filter2D(Mat<IMG_HEIGHT, IMG_WIDTH, SRC_T> &src,
              Mat<IMG_HEIGHT, IMG_WIDTH, DST_T> &dst,
              Window<K_HEIGHT, K_WIDTH, KN_T> &kernel,
              Point_<POINT_T> anchor)
```

Similar types are used for the arguments src, dst, and kernel as for the OpenCV equivalent. The major difference is that all size parameters need to be compile time constants (IMG_HEIGHT, IMG_WIDTH etc.) and thus a template function and types are used.

Performance Monitor Client Library

The `perfapm-client` library provides an interface to the `video_qt2` application for reading memory throughput performance numbers. The `perfapm-client` implements an OpenAMP interface that allows communication with the `perfapm-server` application running on RPU-1 (see Chapter 1) through the remote processor messaging (RPMsg) framework.

The *perfapm-client* requests data from the *perfapm-server* on a need basis. The following steps are performed:

- Call `perfoacl_init_all()` function to initialize all available APM instances
- Call `perfoacl_get_rd_wr_cnt()` function to read the values of read/write counters filled by the perfapm-server
- Call `perfoacl_deinit_all()` function to disables all available APM instances

The client application then passes the data to the GUI application where it is plotted on a graph. This allows you to monitor and visualize the memory throughput caused by the video traffic live at run-time.

RPU-1 Software Stack (Bare-metal)

Introduction

Figure 4-1 shows the overall software stack running on RPU-1. The BSP layer contains the bare-metal implementation along with the AXI performance monitor (APM) driver, inter processor interrupt (IPI) driver, and OpenAMP library.

The application layer contains the `perfapm` library and the `perfapm-ctl` and `perfapm-server` applications. The `perfapm` library uses APIs provided by the APM driver to read various performance metrics from the APM hardware instances inside the PS.

The `perfapm-ctl` application uses APIs provided by the `perfapm` library and prints the performance data on UART1. The `perfapm-server` application uses APIs provided by the `perfapm` and OpenAMP libraries to package the performance data so it can be sent to a remote application via the OpenAMP framework.

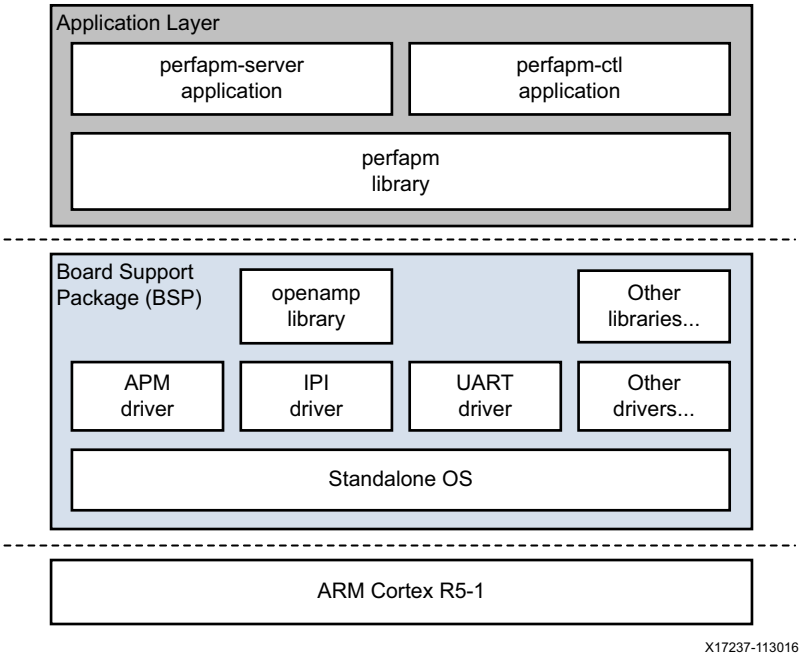

Figure 4-1: **Bare-metal Software Stack Running on RPU-1**

Performance Monitor Library

The AXI performance monitor (APM) block enables AXI system performance measurement. The block captures configurable real-time performance metrics such as throughput and latency of AXI interfaces. As shown in Figure 4-1, there are a total of four APMs monitoring nine AXI interface points inside the PS:

- One OCM switch to OCM monitor point
- One LPD switch to FPD switch (CCI) monitor point
- One CCI to core switch monitor point
- Six DDR controller monitor points
  - slot 0 for OCM switch to OCM
  - slot 1 for CCI ACE-Lite master port 0 to DDR
  - slot 2 for CCI ACE-Lite master port 1 to DDR
  - slot 3 for DisplayPort + PL_HP0
  - slot 4 for PL_HP1 + PL_HP2
  - slot 5 for PL_HP3 + FPD-DMA

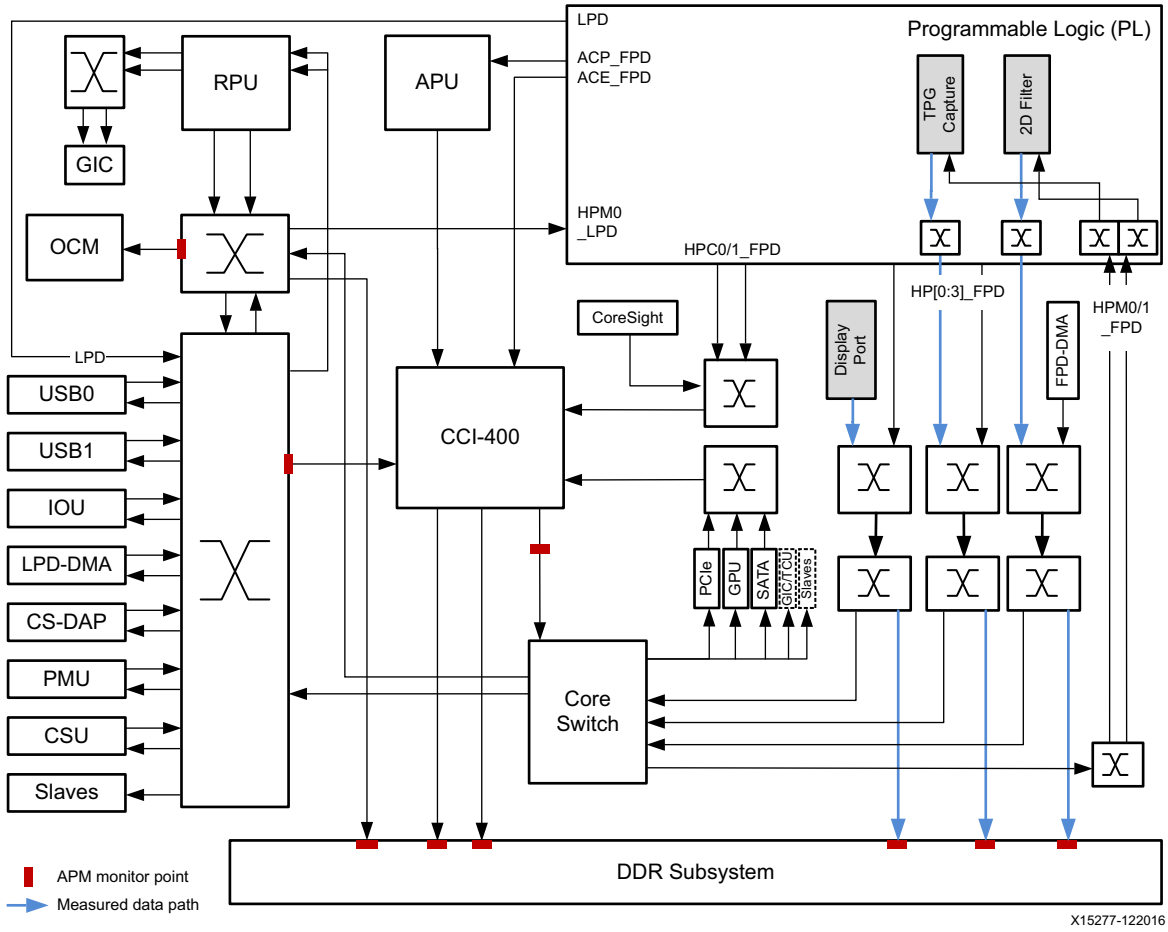

Figure 4-2: **PS Interconnect with APM Monitor Points**

The APMs are configured in profile mode, each providing an event count module that includes two profile counters per APM monitor point. The two counters are software configurable and can be used to monitor different metrics. Both profile counters are 32 bits in size each.

The `perfapm` library is implemented in bare-metal targeting RPU-1. The library provides APIs to configure above APMs to read various performance metrics:

- Read/write transaction count
- Read/write latency
- Read/write byte count
- Slave write/Master read idle cycle count

In this design, the two counters of each monitor point are configured to monitor total read and write byte counts using the APM bare-metal driver. Read and write byte counts are then added up in software to calculate the total throughput for each monitor point.

Performance Monitor Applications

The `perfapm-ctl` and `perfapm-server` applications sit on top of the `perfapm` library and are implemented in bare-metal and executed on RPU-1. The `perfapm-server` packages the data acquired by the library so it can be sent to a remote communication endpoint using the OpenAMP framework whereas the `perfapm-ctl` application prints the same data to the UART1.

Specifically, the applications read the following data using APIs provided by the `perfapm` library:

- Two APM profile counters (total read and write byte counts) for the OCM monitor point
- Two APM profile counters (total read and write byte counts) for the LPD monitor point
- Ten APM profile counters (total read and write byte counts) for five out of six DDR monitor points

The `perfapm-server` application acts as one endpoint of the OpenAMP communication providing data to the remote endpoint running on the APU. On the client side, a small library `perfapm-client` (see Performance Monitor Client Library, page 28) receives the data from the `perfapm-server`.

The OpenAMP communication between the two endpoints follows the master (`perfapm-client`) and slave (`perfapm-server`) concept at the protocol level where the master initiates the communication.

Shared DDR memory is used to communicate between OpenAMP master and slave. The *remoteproc* kernel module on the master side allocates this shared memory and updates the slave about the memory location during firmware download.

The shared memory data structures are ring buffers provided by an underlying virtIO component. There are two ring buffers, one for transmit (TX) and one for receive (RX). Each ring buffer has 256 descriptors/buffers and each buffer is 512 bytes in size. Hence, the remoteproc kernel module allocates 256 KB of memory that is shared between master and slave. See Inter-process Communication, page 59 for more information.

Bare-metal BSP

The bare-metal board support package (BSP) is targeting RPU-1 and shared between the `perfapm` library, `perfapm-ctl`, and `perfapm-server` applications. The BSP includes the OpenAMP driver needed by the `perfapm-server` application.

RPU-0 Software Stack (FreeRTOS)

Introduction

Figure 5-1 shows the overall software stack running on RPU-0. The BSP layer contains the FreeRTOS OS along with required peripheral drivers and libraries. The application layer contains the heartbeat application which is a simple dual-task application that prints periodic status message to the UART.

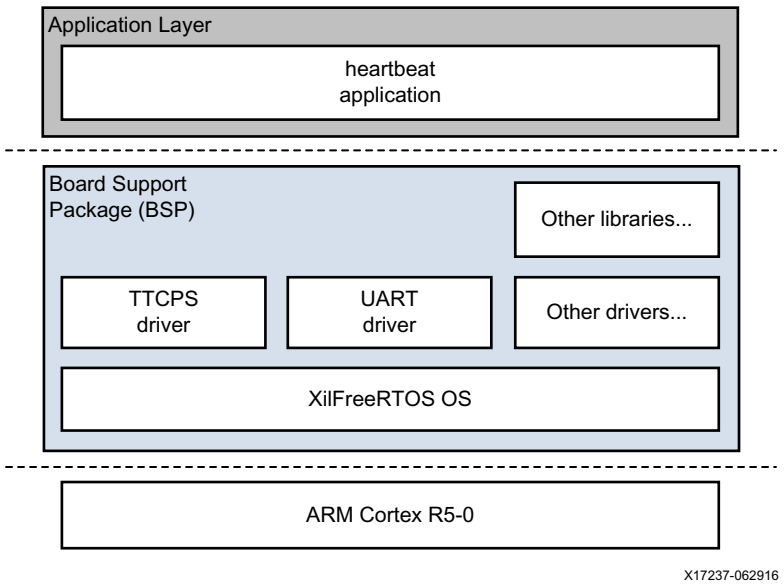

Figure 5-1: **FreeRTOS Software Stack Running on RPU-0**

Heartbeat Application

The heartbeat application executes on RPU-0 running FreeRTOS and performs the following tasks:

- Creates two tasks, one for Tx and one for Rx. The Tx task is given a lower priority than the Rx task, so the Rx task leaves the blocked state and pre-empts the Tx task as soon as the Tx task places an item in the queue.
- Creates the queue used by the tasks. The Rx task has a higher priority than the Tx task and preempts the Tx task and removes values from the queue as soon as the Tx task writes to the queue. Therefore the queue can never have more than one item in it.
- Starts the tasks and timer. Both tasks send heartbeat messages to each other to demonstrate uninterrupted communication, independent from the other processing units in the system. The Tx task prints the messages to UART-1.

FreeRTOS BSP

The FreeRTOS BSP is targeting RPU-0. FreeRTOS is a popular real-time operating system kernel for embedded devices that has been ported to the Zynq UltraScale+ MPSoC. FreeRTOS is designed to be small and simple for very fast execution.

FreeRTOS provides methods for multiple threads or tasks, mutexes, semaphores and software timers. FreeRTOS can be thought of as a t*hread library* rather than a full-blown operating system.

FreeRTOS implements multiple threads by having the host program call a thread tick method at regular short intervals. The thread tick method switches tasks depending on priority and a round-robin scheduling scheme. The interval is 1/1000 of a second via an interrupt from a hardware timer.

System Considerations

Boot Process

The reference design uses a non-secure boot flow and SD boot mode. The sequence diagram in Figure 6-1 shows the exact steps and order in which the individual boot components are loaded and executed.

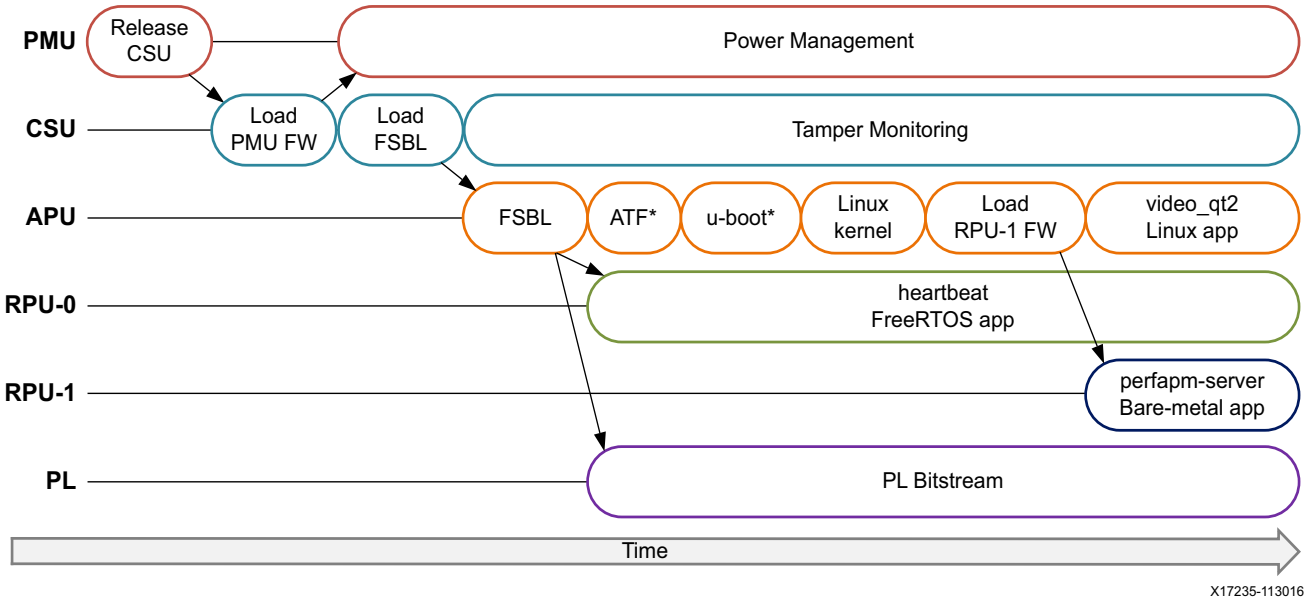

Figure 6-1: **Boot Flow Diagram**

The platform management unit (PMU) is responsible for handling the primary pre-boot tasks and is the first unit to *wake up* after power-on-reset (POR). After the initial boot-up process, the PMU continues to run and is responsible for handling various clocks and resets of the system as well as system power management. In the pre-configuration stage, the PMU executes the PMU ROM and releases the reset of the configuration security unit (CSU). It then enters the PMU server mode where it monitors power.

The CSU handles the configuration stages and executes the boot ROM as soon as it comes out of reset. The boot ROM determines the boot mode by reading the boot mode register, it initializes the OCM, and reads the boot header. The CSU loads the PMU firmware into the PMU RAM and signals to the PMU to execute the firmware which provides advanced power

management features instead of the PMU ROM. It then loads the first stage boot loader (FSBL) into on-chip memory (OCM) and switches into tamper monitoring mode.

In this design, the FSBL is executed on APU-0. It initializes the PS and configures the PL, RPU and APU based on the boot image header information. The following steps are performed:

1. The PL is configured with a bitstream and the PL reset is de-asserted.
2. The RPU-0 application executable is loaded into a combination of TCM and OCM.
3. The ARM trusted firmware (ATF) is loaded into OCM and executed on APU-0.
4. The second stage boot loader u-boot is loaded into DDR to be executed by APU-0.

After the FSBL finishes, the RPU-0 and APU-0 cores are brought out of reset and start executing. Note that at this point, RPU-1 is still held in reset as no executable has been loaded thus far.

The `heartbeat` application starts executing on RPU-0. It is a simple, multi-threaded application running on FreeRTOS. It demonstrates the interaction between two tasks within the same application. This application runs independent from the APU and RPU-1 cores.

In parallel, the APU-0 runs u-boot which performs some basic system initialization and loads the Linux kernel, device tree, and rootfs into DDR. It then boots the Linux kernel at which point all four A53 cores inside the APU are run in SMP mode.

When Linux is fully booted, the APU master loads the `perfapm-server` bare-metal firmware onto the RPU-1 slave using the `remoteproc` framework. A custom `rpmsg` user driver kernel module establishes a communication channel between the APU and RPU-1 and allocates two ring buffers which are used to send messages between the two processing units.

Lastly, the `video_qt2` Linux user-space application is started which configures and controls the video pipeline. The built-in `perfapm-client` library communicates with the RPU-1 firmware to exchange system performance monitoring data.

For more information on the boot process, see chapters 2 and 7 in *Zynq UltraScale+ MPSoC Software Developer Guide* (UG1137) [Ref 8], and chapter 8 in *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 9].

Global Address Map

Peripherals

Figure 6-2 shows the global address map of the Zynq MPSoC device in the center with key low-power domain (LPD) and full-power domain (FPD) PS peripherals on the right and PL peripherals on the left. Both PS and PL peripherals are mapped to 32-bit address space so they can be accessed from both APU and RPU. PL peripherals are mapped to two separate 256 MB address regions via dedicated high-performance master ports (HPM):

- IPs that are part of the base platform are connected to HPM0
- IPs that are generated by SDSoC are connected to HPM1

For more information on system addresses, see chapter 8 in *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 9].

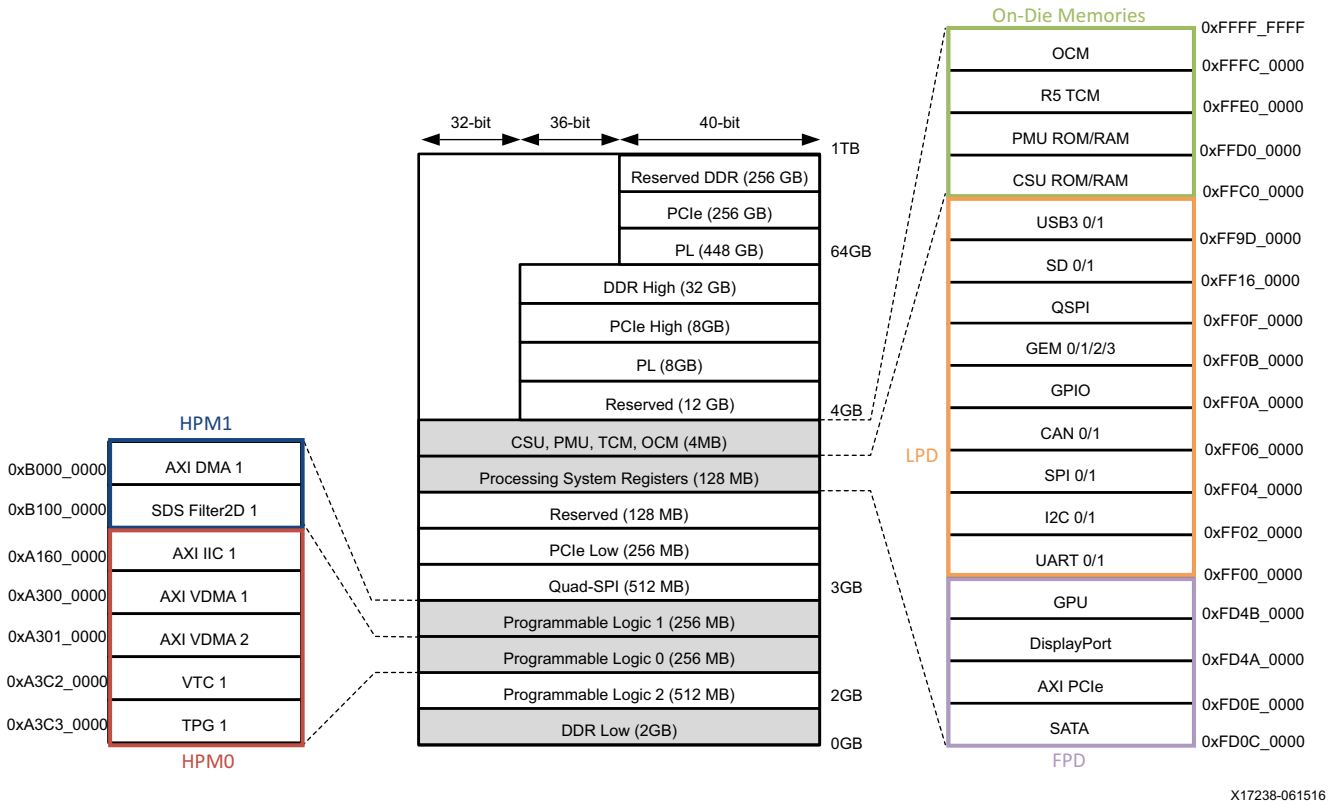

Figure 6-2: **Global Address Map**

Memories

Figure 6-2 also shows the different DDR memory regions (split in low and high) and the various on-die memories (top-right) used to store all program code and run-time data. These DDR memory regions are:

- On-chip memory (OCM)
- RPU tightly-coupled memory (TCM)
- CSU ROM/RAM
- PMU ROM/RAM

The DMA instances in the PL use a 36-bit address space so they can access the DDR Low and DDR High address regions for receiving and transmitting video buffers to be shared with the APU application. Table 6-1 lists the APU and RPU software components used in this design and where they are stored or executed from in memory.

IMPORTANT: Great caution needs to be taken when assigning memory regions for the individual components so they don't overlap each other and thus lead to run-time errors.

OCM is a shared resource and can be accessed by multiple processing units. The FSBL is loaded into OCM by the boot ROM. The FSBL mapping leaves a gap toward the upper section of OCM where the ARM trusted firmware (ATF) is loaded.

As the RPU is configured for split mode, the two RPU cores have their own private tightly coupled memories, 128 KB each (TCM-0 and TCM-1). TCM cannot be accessed by any other processing units.

The `heartbeat` application running on RPU-0 and the `perfapm-server` firmware running on RPU-1 are loaded into DDR memory. The DDR memory regions have to be reserved upfront in the device tree.

The Linux kernel components are loaded into DDR by u-boot. The flattened image tree (FIT) image format is used which is a flexible, monolithic binary that contains all boot components which ensures that the kernel, device-tree, and rootfs do not overlap. The `video_qt2` application is an executable in elf format. The Linux operating system takes care of loading and executing the binary as well as mapping it into its virtual address space.

Table 6-1: **Software Executables and their Memory Regions**

Design Component	Processing Unit	Memory
FSBL	APU-0	OCM
heartbeat	RPU-0	DDR
ARM trusted firmware (ATF)	APU-0	OCM
u-boot	APU-0	DDR

Table 6-1: **Software Executables and their Memory Regions *(Cont'd)***

Design Component	Processing Unit	Memory
Linux kernel/device tree/rootfs	APU (SMP)	DDR
video_qt2	APU (SMP)	DDR
perfapm-server	RPU-1	DDR

Table 6-2 lists important data structures stored in DDR memory and shared between multiple processing units or hardware devices. Vring buffers and video buffers are allocated by kernel drivers using the contiguous memory allocator (CMA).

Table 6-2: **Shared Data Structures in DDR Memory**

Data Structure	Shared Between	Memory Properties
Video buffers	• APU • PS DPDMA • PL DMA/VDMAs • GPU	Non-coherent, contiguous
Vring buffers (OpenAMP)	• APU • RPU-1	Coherent, contiguous

For more information on TCM and OCM, see chapters 4 and 16 in *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 9].

Video Buffer Formats

The TRD uses two layers (or planes) that get alpha blended inside the display subsystem which sends a single video stream to the DisplayPort transmitter. The bottom layer is used for video frames while the top layer is used for graphics. The graphics layer consist of the GUI and is rendered by the GPU. It overlays certain areas of the video frame with GUI control elements while other parts of the frame are transparent. A mechanism called *pixel alpha* is used to control the opacity of each pixel in the graphics plane.

The pixel format used for the graphics plane is called ARGB8888 or AR24 (see Figure 6-3, top). It is a packed format that uses 32 bits to store the data value of one pixel (32 bits per pixel or bpp), 8 bits per component (bpc) also called color depth or bit depth. The individual components are: alpha value (A), red color (R), green color (G), blue color (B). The alpha component describes the opacity of the pixel: an alpha value of 0% means the pixel is fully transparent (invisible); an alpha value of 100% means that the pixel if fully opaque.

The pixel format used for the video plane is YUYV which is a packed format that uses 16 bpp and 8bpc (see Figure 6-3, bottom). YUYV is commonly used for 4:2:2 sub-sampled YUV images where the luma component Y has twice as many samples (or double the horizontal resolution) as each of the U/V chroma components (also referred to as Cb/Cr components). A 32 bit word is also referred to as macro-pixel as it holds information for two pixels where

Cb and Cr are common for both Y0 and Y1 samples. This results in an average value of 16 bpp.

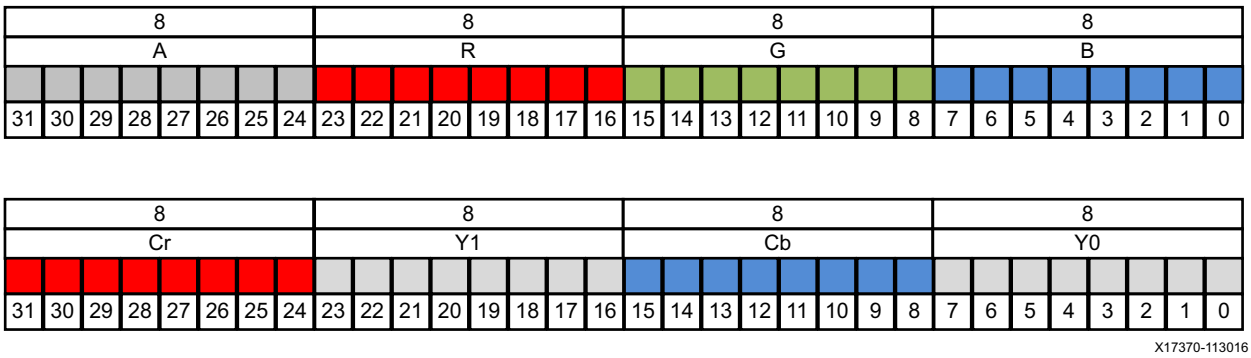

Figure 6-3: **AR24 (Top) and YUYV (Bottom) Pixel Formats in Little Endian Notation**

Aside from the pixel format, a video buffer is further described by a number of other parameters (see Figure 6-4). For this design, the relevant parameters are width, height, and stride as the PS display pipeline does not allow for setting an x or y offset:

- The active area is the part of the video buffer that is visible on the screen. The active area is defined by the height and width parameters, also called the video dimensions. Those are typically expressed in number of pixels as the bits per pixel depend on the pixel format as explained above.
- The stride or pitch is the number of bytes from the first pixel of a line to the first pixel of the next line of video. In the simplest case, the stride equals the width multiplied by the bits per pixel, converted to bytes. For example: AR24 requires 32 bpp which is four bytes per pixel. A video buffer with an active area of 1920x1080 pixels therefore has a stride of 4 x 1920=7,680 Bytes. Some DMA engines require the stride to be power of two to optimize memory accesses. In this design, the stride always equals the width in bytes.

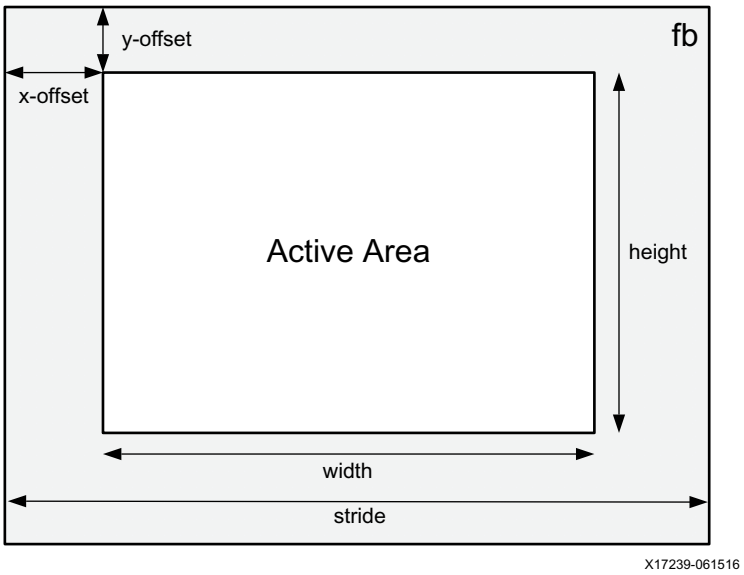

Figure 6-4: **Video Buffer Parameters**

Performance Metrics

Memory Throughput

The memory throughput for a video stream of resolution 3840 by 2160 pixels at 30 frames per second (2160p30) using a YUYV pixel format, is calculated as:

3840 x 2160 x 16 bits x 30/s ≈ 4 Gb/s

Or using a AB24 pixel format, as:

3840 x 2160 x 32 bits x 30/s ≈ 8 Gb/s

These numbers differ, depending on the targeted video resolution.

To calculate the total memory throughput, you simply need to multiply the throughput of a single stream with the number of streams being read/written to/from memory. Assuming the same resolution and frame rate as in above example and the following video pipeline configuration:

- video capture: 1 YUYV stream
- video processing: 2 YUYV streams (Rx + Tx)
- display:
  - video layer: 1 YUYV stream
  - graphics layer: 1 AR24 stream

The total memory throughput is calculated as:

$$4 \times 4\ \text{Gb/s} + 1 \times 8\ \text{Gb/s} = 24\ \text{Gb/s}$$

This configuration corresponds to the maximum memory throughput supported in this reference designs. The above calculation only takes into account video traffic but no other memory traffic for example as generated by the CPU, RPU, or GPU. Especially the memory throughput generated by the GPU for graphics rendering can be significant depending on the complexity of the scene and needs to be considered when architecting the system.

On the ZCU102, a 64-bit DDR4-2133 DIMM is used. Thus, the theoretical maximum memory throughput is calculated as:

$$64\ \text{bits} \times 2{,}133\ \text{MHz} \approx 136\ \text{Gb/s}$$

The maximum memory utilization based on a worst-case video traffic of 24 Gb/s is about 18% which leaves plenty of headroom for more demanding applications:

$$24\ \text{Gb/s} \div 136\ \text{Gb/s} \approx 0.18$$

CPU Utilization

The CPU utilization and load distribution across the individual processing cores inside the APU is highly dependent on the computational workload demanded by the application and on its implementation. In this example, a 2D convolution algorithm is used based on the standard OpenCV library implementation (see 2D Filter Plug-in, page 26 for details).

The 2D filter algorithm is used as is and not specifically optimized for the APU or ARMv8 (micro-) architecture. Neither does it exploit data level parallelism using ARM's NEON SIMD (single data, multiple word) instruction set extension, nor does it make use of thread level parallelism for example by partitioning the video frame to process sub-frames on separate processing cores.

As a result, depending on the resolution of the captured video stream, the frame rate drops significantly to single-digit values if the 2D filter is run purely in software. For example, a 1080p input stream at 60 fps can only be processed at less than 2 fps using this non-optimized, single-threaded implementation of the algorithm. At the same time, the utilization of a single core increases to 100% when the 2D filter is running, occupying all its resources while the other three cores are almost idle.

By profiling the software code, one can see that the majority of the time is spent in the 2D filter algorithm. The algorithm performs pixel processing operations which are not a good fit for a general-purpose CPU as operations are performed in a sequential manner. However, it is well suited for a parallel, pipelined processing architecture as the same operation is performed on every pixel of the image with limited number of data dependencies; this is a great fit for the PL.

By accelerating the 2D filter in the PL, the input frame rate of the video stream can be maintained and the load of all APU cores goes down to low single digit values. This comes at the cost of increased memory throughput (see previous section). Essentially the APU is only performing basic housekeeping now i.e. configuring the hardware pipelines and managing the video buffer flow which is done at the frame-level and not at the pixel-level and therefore a much better fit.

Quality of Service

The Zynq UltraScale+ MPSoC DDR controller has six dedicated ports that are connected to different interconnect structures and AXI masters inside the PS and PL (see Figure 4-2, page 31). The arbitration scheme between those ports is priority based with multiple tiers of arbitration stages.

1. Read/Write arbitration combines multiple reads or writes together and tries to minimize direction switches to improve memory bus efficiency.
2. Read/Write priorities define relative priorities between traffic classes for reads and writes individually. Only one traffic class can be assigned to each DDR controller port.
3. Port command priorities are per transaction and can change dynamically based on AXI signals. They are set individually for data reads (ARQoS) and writes (AWQoS).
4. Round-robin arbitration to resolve ties in the final stage.

The Base TRD uses a combination of (2) and (3) to optimize performance. For read/write priorities (2), the following traffic classes are defined:

- Video/isochronous traffic class is a real-time, fixed bandwidth, fixed maximum latency class. Typically, a long latency and low priority is acceptable, but the latency must be bounded in all cases and never exceed a predefined maximum value.
- Low latency (LL) traffic class is a low-latency, high-priority class. It is typically assigned the highest memory access priority and can only be surpassed by a video class transaction that has exceeded its threshold maximum latency.
- Best effort (BE) traffic class is used for all other traffic types. This class of traffic is typically assigned the lowest memory access priority.

For PL AXI masters connected to HP ports, command priorities (3) can be either controlled dynamically from dedicated AXI QoS signals (on a per transaction basis) or statically configured via register interface.

Table 6-3 lists the traffic class and the HP port and its command priority (if applicable) for each of the four video/display pipelines used in the design the DDR controller port to which it is connected.

Table 6-3: **Quality of Service Summary**

Pipeline	HP Port	ARQoS[3:0]/ AWQoS[3:0]	Traffic class	DDR Controller Port
Display	-	0xB/ -	Video	3
TPG capture	HP1	- /0x0	Best effort	4
M2M accelerator	HP3	0x0/0x0	Best effort	5

Different ports into the DDR controller are used for video capture (TPG), video acceleration, and display to achieve highest throughput. PL masters are connected to HP ports to fit this scheme as the HP port to DDR controller port connections are fixed.

The display pipeline is configured for video traffic class as it is a video-timing accurate device with fixed size requests at regular intervals determined by the consumption rate. The consumption rate depends on the chosen video format, frame rate and resolution (see Memory Throughput, page 41).

Although the video capture pipeline for TPG is also video timing accurate, the VDMA currently lacks support for writing or fetching video data according to the video class traffic type definition. The HP1 port is thus configured for best effort data traffic instead of video.

The accelerator pipeline is not video timing accurate and the transaction pattern is greedy rather than regular intervals as typically used in video sources and sinks. As soon as a captured video frame is available in memory, the accelerator network tries to read the entire data as quickly as possible and send it back to memory without considering blanking intervals in between video lines or frames (see Figure 8-4, page 66 for video timing description). The traffic class is therefore *best effort*.

The Base TRD uses static command QoS. The corresponding read and/or write data priority values for display, capture, and m2m have been identified through experimentation. For more information on QoS, refer to [10], chapters 13, 15, and 33 in *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 9].

APU Software Platform

Introduction

This chapter describes the APU Linux software platform which is further sub-divided into a middleware layer and an operating system (OS) layer (see Figure 7-1). We are looking at these two layers in conjunction as they interact closely for most Linux subsystems. These layers are further grouped by vertical domains which reflect the organization of this chapter:

- Video
- Display
- Graphics
- Vision
- Inter-process communication (IPC)

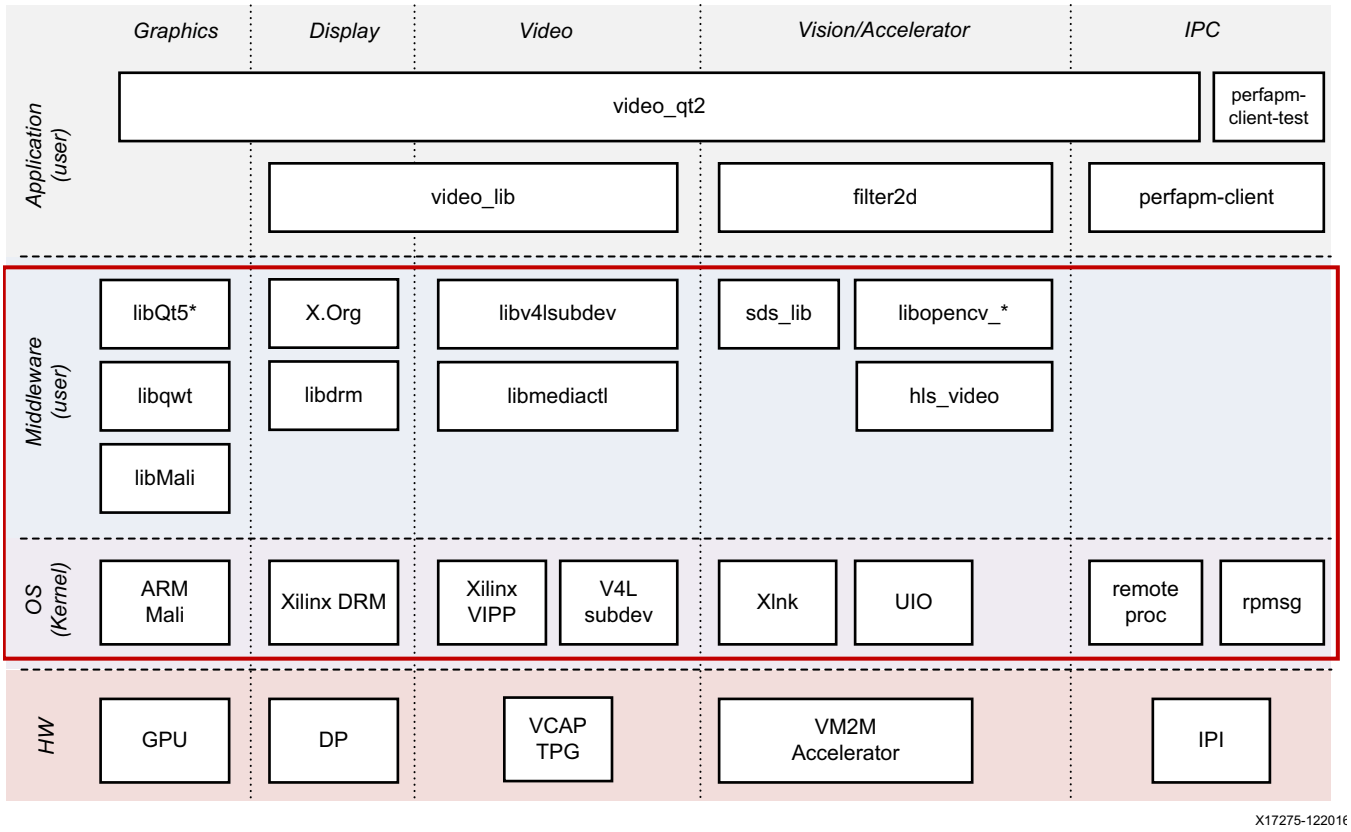

Figure 7-1: **APU Linux Software Platform**

The middleware layer is a horizontal layer implemented in user-space and provides the following functionality:

- Interfaces with the application layer
- Provides access to kernel frameworks
- Implements domain-specific functionality that can be re-used in different applications (e.g., OpenCV for computer vision)

The OS layer is a horizontal layer implemented in kernel-space and provides the following functionality:

- Provides a stable, well-defined API to user-space
- Includes device drivers and kernel frameworks (subsystems)
- Accesses the hardware

Video

In order to model and control video capture pipelines such as the ones used in this TRD on Linux systems, multiple kernel frameworks and APIs are required to work in concert. For simplicity, we refer to the overall solution as Video4Linux (V4L2) although the framework only provides part of the required functionality. The individual components are discussed in the following sections.

Driver Architecture

Figure 7-2 shows how the generic V4L2 driver model of a video pipeline is mapped to the TPG capture pipeline. The video pipeline driver loads the necessary sub-device drivers and registers the device nodes it needs, based on the video pipeline configuration specified in the device tree. The framework exposes the following device node types to user space to control certain aspects of the pipeline:

- Media device node: `/dev/media*`
- Video device node: `/dev/video*`
- V4L sub-device node: `/dev/v4l-subdev*`

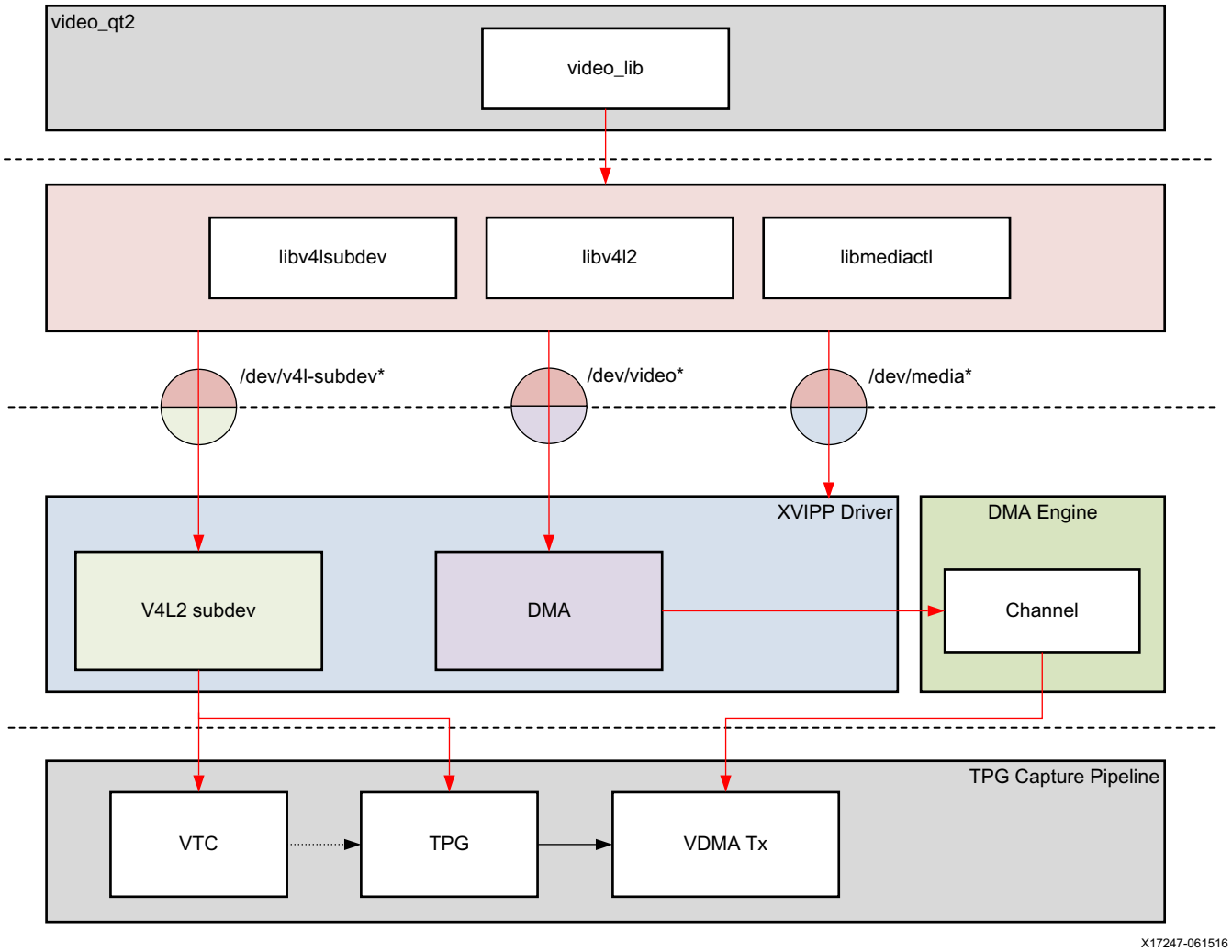

Figure 7-2: **V4L2 Driver Stack**

Media Framework

The main goal of the media framework is to discover the device topology of a video pipeline and to configure it at runtime. To achieve this, pipelines are modeled as an oriented graph of building blocks called entities connected through pads.

An entity is a basic media hardware building block. It can correspond to a large variety of blocks such as physical hardware devices (e.g. image sensors), logical hardware devices (e.g., soft IP cores inside the PL), DMA channels or physical connectors. Physical or logical devices are modeled as sub-device nodes and DMA channels as video nodes.

A pad is a connection endpoint through which an entity can interact with other entities. Data produced by an entity flows from the entity's output to one or more entity inputs. A link is a point-to-point oriented connection between two pads, either on the same entity or on different entities. Data flows from a source pad to a sink pad.

A media device node is created that allows the user space application to configure the video pipeline and its sub-devices through the `libmediactl` and `libv4l2subdev` libraries. The media controller API provides the following functionality:

- Enumerate entities, pads and links
- Configure pads
  - Set media bus format
  - Set dimensions (width/height)
- Configure links
  - Enable/disable
  - Validate formats

Figure 7-3 shows the media graph for the TPG video capture pipeline as generated by the `media-ctl` utility. The TPG sub-device is shown in green with its corresponding control interface address and sub-device node in the center. The numbers on the edges are pads and the solid arrows represent active links. The yellow boxes are video nodes that correspond to VDMA channels, in this case write channels (outputs).

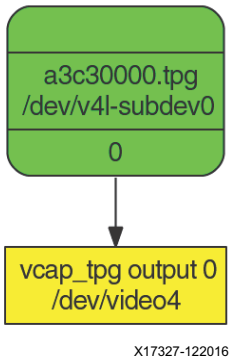

Figure 7-3: **Video Capture Media Pipeline**

V4L2 Framework

The V4L2 framework is responsible for capturing video frames at the video device node, typically representing a DMA channel, and transferring those video frames to user space. The framework consists of multiple sub-components that provide certain functionality.

Before video frames can be captured, the buffer type and pixel format need to be set using the `VIDOC_S_FMT` ioctl. On success the driver may program the hardware, allocate resources and generally prepare for data exchange.

Optionally, the user can set additional control parameters on V4L devices and sub-devices. The V4L2 control framework provides ioctls for many commonly used, standard controls like

brightness, contrast etc. as well as device-specific, custom controls. For example, the TPG sub-device driver implements the standard control for selecting a test pattern and several custom controls e.g. foreground overlay (moving box or cross hairs), motion speed etc.

The videobuf2 API implements three basic buffer types of which only physically contiguous in memory is supported in this driver due to the hardware capabilities of the VDMA. Videobuf2 provides a kernel internal API for buffer allocation and management as well as a user-space facing API.

Videobuf2 supports three different memory models for allocating buffers. The `VIDIOC_QUERYCAP` and `VIDIOC_REQBUFS` ioctls are used to determine the I/O mode and memory type. In this design, the streaming I/O mode in combination with the DMABUF memory type is used.

DMABUF is dedicated to sharing DMA buffers between different devices, such as V4L devices or other video-related devices like a DRM display device (see Video Buffer Management, page 23). In DMABUF, buffers are allocated by a driver on behalf of an application. Next, these buffers are exported to the application as file descriptors.

For capturing applications it is customary to queue a number of empty buffers using the `VIDIOC_QBUF` ioctl. The application waits until a filled buffer can be de-queued with the `VIDIOC_DQBUF` ioctl and re-queues the buffer when the data is no longer needed. To start and stop capturing applications, the `VIDIOC_STREAMON` and `VIDIOC_STREAMOFF` ioctls are used.

Video IP drivers

Xilinx adopted the V4L2 framework for part of its video IP portfolio. The currently supported video IPs and corresponding drivers are listed at http://www.wiki.xilinx.com/Linux+Drivers under V4L2. Each V4L driver has a sub-page that lists driver specific details and provides pointers to additional documentation. Table 7-1 gives a quick overview of the drivers used in this design.

Table 7-1: **V4L2 Drivers Used in Capture Pipelines**

Linux Driver	Function
Xilinx Video Pipeline (XVIPP)	• Configures video pipeline and register media, video and sub-device nodes. • Configures all entities in the pipeline and validate links. • Starts/stops video stream.
Xilinx Video Test Pattern Generator	• Sets media bus format on TPG output pad. • Configures VTC to provide correct timing to TPG. • Sets TPG control parameters such as test pattern, foreground overlay, motion speed, etc.

Display

Linux kernel and user-space frameworks for display and graphics are intertwined and the software stack can be quite complex with many layers and different standards / APIs. On the kernel side, the display and graphics portions are split with each having their own APIs. However, both are commonly referred to as a single framework, namely DRM/KMS.

This split is advantageous, especially for SoCs that often have dedicated hardware blocks for display and graphics. The display pipeline driver responsible for interfacing with the display uses the kernel mode setting (KMS) API and the GPU responsible for drawing objects into memory uses the direct rendering manager (DRM) API. Both APIs are accessed from user-space through a single device node.

Direct Rendering Manager

The direct rendering manager (DRM) is a subsystem of the Linux kernel responsible for interfacing with a GPU. DRM exposes an API that user space programs can use to send commands and data to the GPU. The ARM Mali driver uses a proprietary driver stack which is discussed in the next section. Therefore this section focuses on the common infrastructure portion around memory allocation and management that is shared with the KMS API.

Driver Features

The Xilinx DRM driver uses the GEM memory manager, and implements DRM PRIME buffer sharing. PRIME is the cross device buffer sharing framework in DRM. To user-space PRIME buffers are DMABUF-based file descriptors.

The DRM GEM/CMA helpers use the CMA allocator as a means to provide buffer objects that are physically contiguous in memory. This is useful for display drivers that are unable to map scattered buffers via an IOMMU.

Frame buffers are abstract memory objects that provide a source of pixels to scan out to a CRTC. Applications explicitly request the creation of frame buffers through the `DRM_IOCTL_MODE_ADDFB(2)` ioctls and receive an opaque handle that can be passed to the KMS CRTC control, plane configuration and page flip functions.

Dumb Buffer Objects

The KMS API doesn't standardize backing storage object creation and leaves it to driver-specific ioctls. Furthermore actually creating a buffer object even for GEM-based drivers is done through a driver-specific ioctl - GEM only has a common userspace interface for sharing and destroying objects. Dumb objects partly alleviate the problem by providing a standard API to create dumb buffers suitable for scanout, which can then be used to create KMS frame buffers.

Kernel Mode Setting

Mode setting is an operation that sets the display mode including video resolution and refresh rate. It was traditionally done in user-space by the X-server which caused a number of issues due to accessing low-level hardware from user-space which, if done wrong, can lead to system instabilities. The mode setting API was added to the kernel DRM framework, hence the name *kernel mode setting*.

The KMS API is responsible for handling the frame buffer and planes, setting the mode, and performing page-flips (switching between buffers). The KMS device is modeled as a set of planes, CRTCs, encoders, and connectors as shown in the top half of Figure 7-4. The bottom half of the figure shows how the driver model maps to the physical hardware components inside the PS display pipeline.

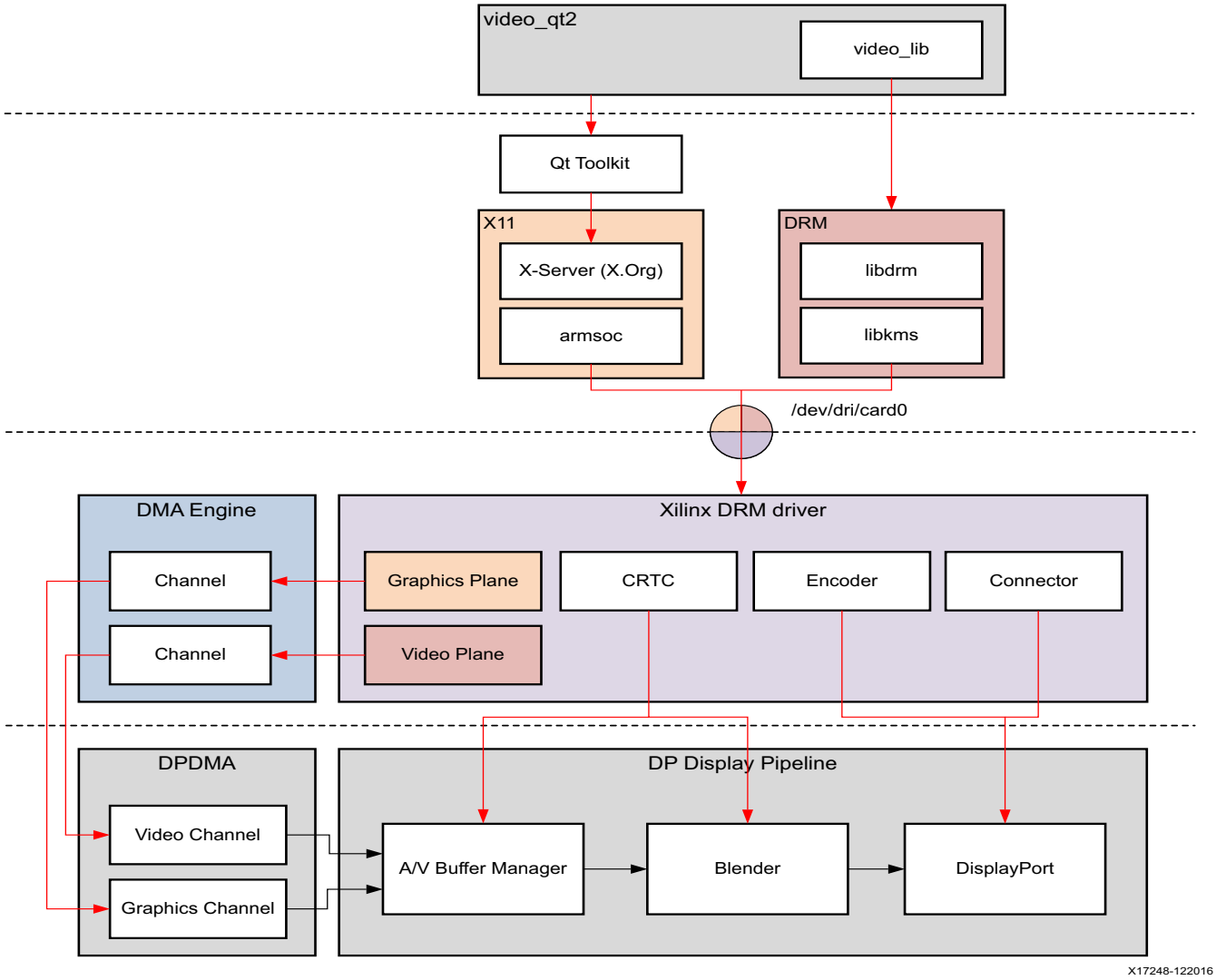

Figure 7-4: **DRM/KMS Driver Stack**

CRTC

CRTC is an antiquated term that stands for cathode ray tube controller, which today would be simply named display controller as CRT monitors have disappeared and many other display types are available. The CRTC is an abstraction that is responsible for composing the frame to be scanned out to the display and setting the mode of the display.

In the Xilinx DRM driver, the CRTC is represented by the buffer manager and blender hardware blocks. The frame buffer (primary plane) to be scanned out can be overlayed and/or alpha-blended with a second plane inside the blender. The hardware supports up to two planes, one for video and one for graphics. The z-order (foreground or background position) of the planes and the alpha mode (global or pixel-alpha) can be configured through the driver via custom properties.

The pixel formats of the video and graphics planes can be configured individually and a variety of formats are supported. In the current implementation, the pixel formats are set statically in the device tree. Pixel unpacking and format conversions are handled by the buffer manager and blender. The DRM driver configures the hardware accordingly so this is transparent to the user.

A page-flip is the operation that configures a plane with the new buffer index to be selected for the next scan-out. The new buffer is prepared while the current buffer is being scanned out and the flip typically happens during vertical blanking to avoid image tearing.

Plane

A plane represents an image source that can be blended with or overlayed on top of a CRTC frame buffer during the scan-out process. Planes are associated with a frame buffer to optionally crop a portion of the image memory (source) and scale it to a destination size. The PS display pipeline does not support cropping or scaling, therefore both video and graphics plane dimensions have to match the CRTC mode (i.e., the resolution set on the display).

The Xilinx DRM driver supports the universal plane feature, therefore the primary plane and overlay planes can be configured through the same API. As planes are modeled inside KMS, the physical hardware device that reads the data from memory is a DMA whose driver is implemented using the `dmaengine` Linux framework. The DPDMA is a 6-channel DMA engine that support a (up to) 3-channel video stream, a 1-channel graphics stream and two channels for audio.

Encoder

An encoder takes pixel data from a CRTC and converts it to a format suitable for any attached connectors. There are many different display protocols defined, such as HDMI or DisplayPort. The PS display pipeline has a DisplayPort transmitter built in. The encoded video data is then sent to the serial I/O unit (SIOU) which serializes the data using the gigabit transceivers (PS GTRs) before it goes out via the physical DP connector to the display.

Connector

The connector models the physical interface to the display. The DisplayPort protocol uses a query mechanism to receive data about the monitor resolution, and refresh rate by reading the extended display identification data (EDID) (see *VESA Standard* [Ref 10]) stored inside the monitor. This data can then be used to correctly set the CRTC mode. DisplayPort also supports hot-plug events to detect if a cable has been connected or disconnected as well as handling display power management signaling (DPMS) power modes.

Libdrm

The framework exposes two device nodes to user space: the `/dev/dri/card*` device node and an emulated `/dev/fb*` device node for backward compatibility with the legacy `fbdev` Linux framework. The latter is not used in this design.

`libdrm` was created to facilitate the interface of user space programs with the DRM subsystem. This library is merely a wrapper that provides a function written in C for every ioctl of the DRM API, as well as constants, structures and other helper elements. The use of `libdrm` not only avoids exposing the kernel interface directly to user space, but presents the usual advantages of reusing and sharing code between programs.

X11

The X window system or short X11 (for protocol version 11) provides the basic framework for a GUI environment. X uses a network-transparent communication protocol following a client-server model where the X server typically runs on the local machine while the X client can run on the same or on a remote machine. This reverses the typical client-server notion known from networking where the client runs locally and the server on a remote machine.

Display Server

The X server is the center piece of the X window system (see Figure 7-4). It takes input from connected input devices such as a mouse or keyboard and sends events to the client. Similarly, a client can request graphical output to be displayed on the screen. X does not mandate the user interface which is typically done by the GUI tookit. In this design, the open-source display server `X.Org` is used. It implements the X11 protocol and interfaces with the following components:

- `video_qt2` application through Qt toolkit with `eglfs_x11` platform abstraction (top) - see EGLFS QPA, page 57
- Mali user-space library with X11 backend (right) - see Mali Driver, page 58
- DP display controller through the `armsoc` driver and DRI device node (bottom)

The DRM driver enables multi-master mode so the DRI device node can be shared between the `video_lib` library for accessing the video layer through `libdrm` and the `armsoc` driver for accessing the graphics layer through X11.

Graphics

Qt Toolkit

Qt is a full development framework with tools designed to streamline the creation of applications and user interfaces for desktop, embedded, and mobile platforms. Qt uses standard C++ with extensions including signals and slots that simplify handling of events. This helps in the development of both the GUI and server applications which receive their own set of event information and should process them accordingly. Figure 7-5 shows the end-to-end Linux graphics stack relevant for this application.

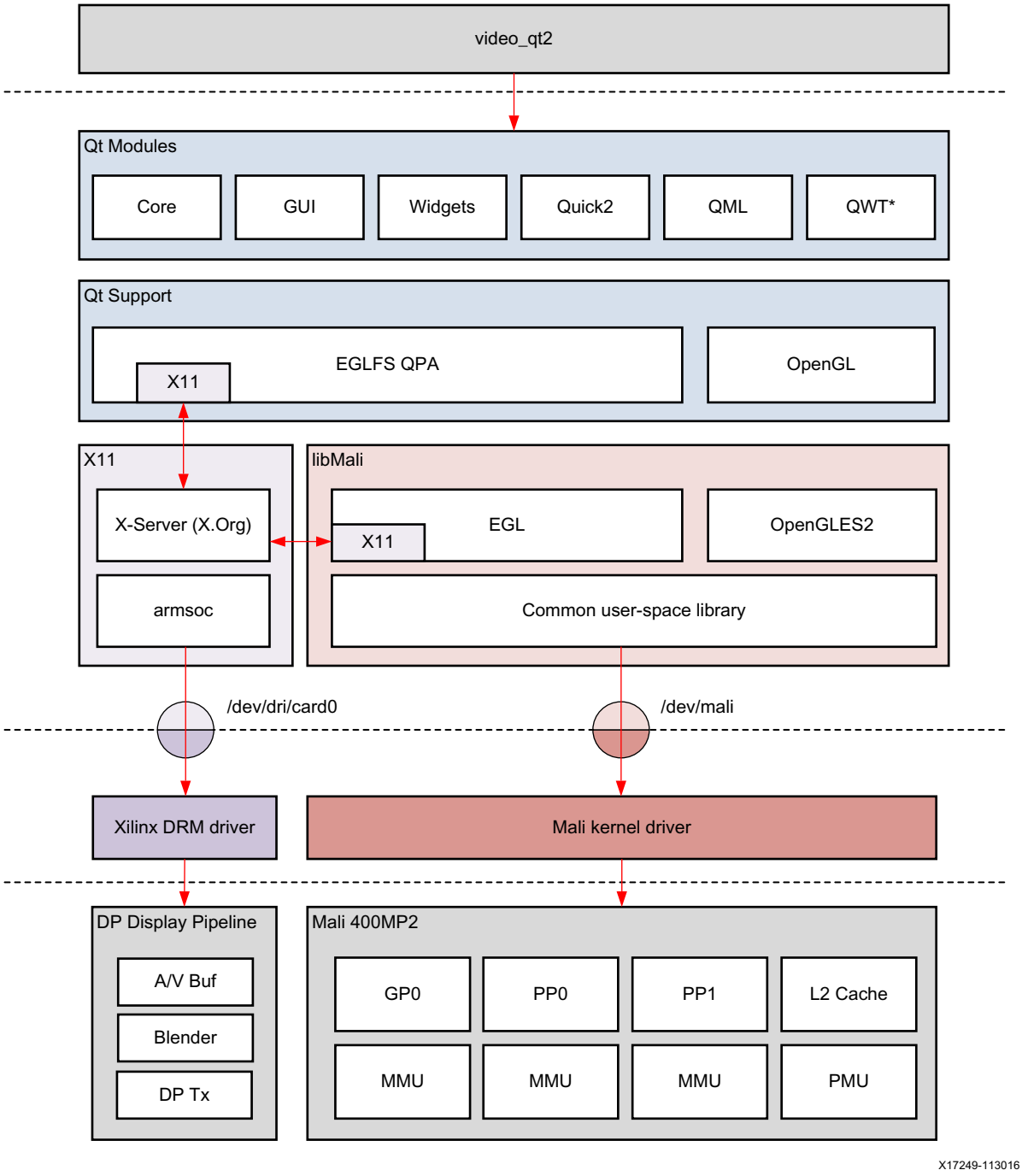

Figure 7-5: **Graphics Stack**

Qt Modules

Qt consists of many different components and supports different plug-ins and interfaces with lower level APIs in the stack. The `video_qt2` application is based on Qt Quick which includes a declarative scripting language called QML that allows using JavaScript to provide the logic. Table 7-2 shows an overview and provides a brief description of the Qt modules used in this application. The Qwt module is a third-party component provided outside the stock Qt release.

Table 7-2: **Qt Modules**

Module	Description
Core	Core non-graphical classes used by other modules.
GUI	Base classes for graphical user interface (GUI) components. Includes OpenGL.
Widgets	Classes to extend Qt GUI with C++ widgets.
Quick2	A declarative framework for building highly dynamic applications with custom user interfaces.
Qml	Classes for QML and JavaScript languages.
Qwt	Utility classes to provide 2D plots, scales, sliders, dials, compasses, and other ranges of widgets for technical applications.

EGLFS QPA

EGLFS is a Qt platform abstraction (QPA) plug-in for running Qt5 applications on top of EGL and OpenGL ES 2.0. EGL is an interface between OpenGL and the native windowing system used for context and surface management. In this design, the EGLFS plug-in is based on the X11 backend matching the Mali user-space driver configuration. EGLFS supports Qt Quick2 as well as native OpenGL applications and is the recommended plug-in for modern embedded Linux devices that include a GPU.

EGLFS forces the first top-level window to become full screen. This window is also chosen to be the root widget window into which all other top-level widgets (e.g., dialogs, popup menus or combo-box dropdowns) are composited. This is necessary because with EGLFS there is always exactly one native window and EGL window surface, and these belong to the widget or window that is created first.

Libinput

Qt also handles input devices such as a mouse or keyboard which are not shown in Figure 7-5, page 56. The mouse and keyboard inputs are handled through the `libinput` library which is enabled through the corresponding EGLFS plug-in. Device discovery is handled by `libudev` which allows for hot-plugging (connecting or disconnecting the input device while the Qt application is running). For more information on EGLFS and `libinput`, see the *Qt for Embedded Linux* website [Ref 11].

Mali Driver

The ARM Mali 400 MP2 GPU consists of one geometry processor (GP0) and two pixel processors (PP0/PP1) with a dedicated MMU for each processor core. It has its own L2 cache and interface to the power management unit (PMU), see Figure 7-5, page 56.

The Mali driver stack consists of an open-source GPU kernel driver and a closed-source user-space driver. The user-space driver is compiled into a single library (`libMali`) that interfaces with the kernel driver through the `/dev/mali` device node. The user-space driver is configured to support the `fbdev` backend interfacing with the emulated frame buffer device node, `/dev/fb0` created by the Xilinx DRM driver.

The `libMali` user-space library implements the OpenGLES 2.0 API which is used by the Qt toolkit for hardware-accelerated graphics rendering. The Mali driver also supports DMABUF which provides a mechanism for sharing buffers between devices and frameworks through file descriptors without expensive memory copies (0-copy sharing).

Vision

OpenCV (open source computer vision) is the most popular and advanced code library for computer vision related applications, spanning from many very basic tasks (capture and pre-processing of image data) to high-level algorithms (feature extraction, motion tracking, machine learning). It is free software and provides a rich API in C, C++, Java and Python. The library itself is platform-independent and often used for real-time image processing and computer vision.

The Xilinx HLS video library is an FPGA hardware-optimized computer vision library that supports a small subset of OpenCV-like functions. In this design, a 2D convolution filter is first prototyped using the stock OpenCV function in pure software and then ported to the HLS-optimized equivalent function for hardware acceleration (see 2D Filter Plug-in, page 26). For more information on the HLS video library, refer to *Vivado Design Suite User Guide - High-Level Synthesis* (UG902) [Ref 6] and *Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries* (XAPP1167) [Ref 7].

Inter-process Communication

Xilinx open asymmetric multi-processing (OpenAMP) is a framework providing the software components needed to enable the development of software applications for asymmetric multi-processing (AMP) systems. The Base TRD uses the following configuration:

- APU running SMP Linux is the master
- RPU-1 running bare-metal is the remote

The following is a list of services that Linux needs to provide to be able to communicate with the remote processor:

- Load firmware into the RPU-1 core
- Control RPU-1 execution (start, stop, etc.)
- Manage resources (memory, interrupt mappings, etc.)
- Provide a method to send/receive messages

All of these services are provided through a combination of the `zynqmp_r5_remoteproc` and `rpmsg_user_dev_driver` Linux drivers. The Linux frameworks used are `remoteproc`, `RPMsg`, and `virtio`. Figure 7-6 shows an overview of the Linux software stack.

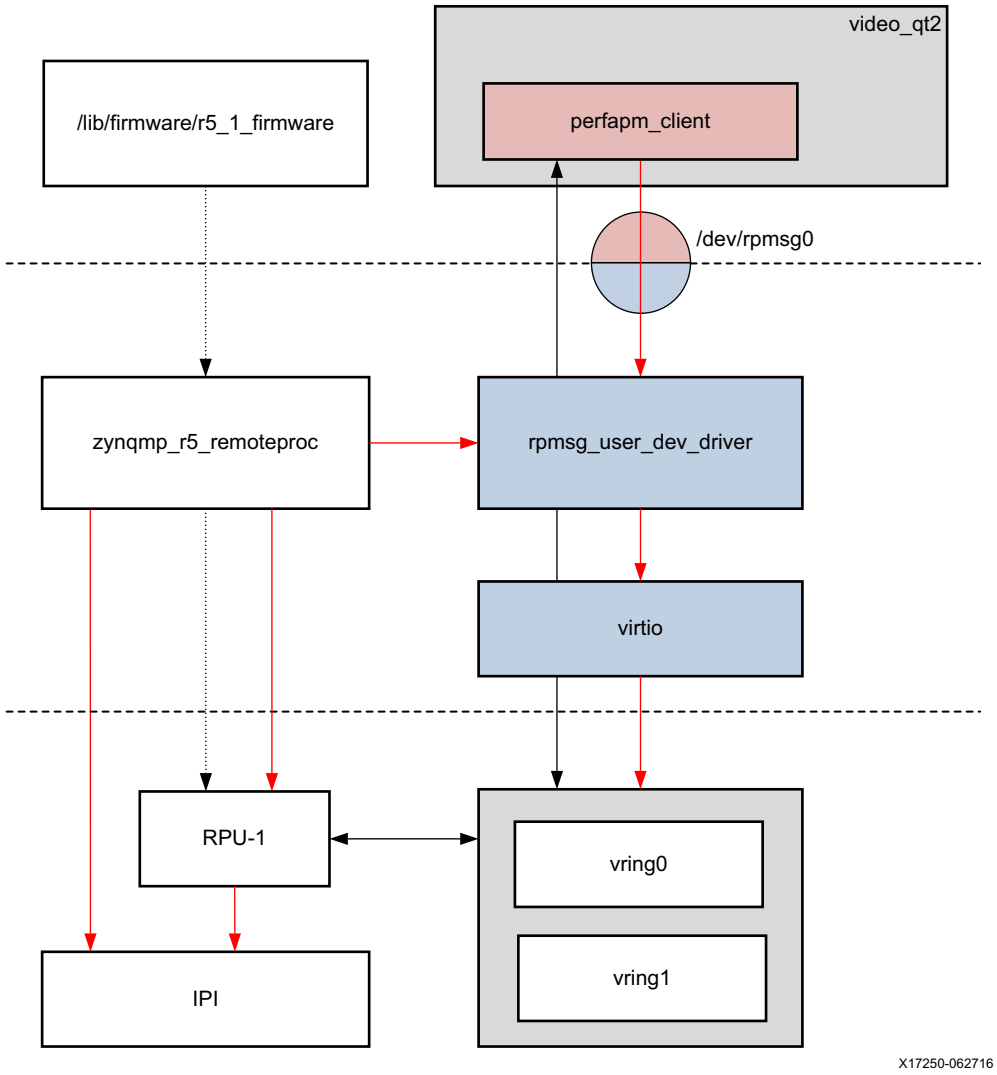

Figure 7-6: **IPC Software Stack**

Linux Components

The OpenAMP framework uses these key components:

- `virtIO` is a virtualization standard for network and disk device drivers where only the driver on the guest device is aware it is running in a virtual environment, and cooperates with the hypervisor. This concept is used by `RPMsg` and `remoteproc` for a processor to communicate to the remote.
- The remote processor (`remoteproc`) framework controls the life cycle management (LCM) of the remote processor from the master processor. The `remoteproc` API that OpenAMP uses is compliant with the infrastructure present in the Linux kernel 3.18 and later. It uses information published through the firmware resource table to allocate system resources and to create `virtIO` devices.

- The remote processor message (RPMsg) framework allows inter-process communications (IPC) between software running on independent cores in an AMP system. This is also compliant with the RPMsg bus infrastructure present in the Linux kernel version 3.18 and later.

The Linux kernel infrastructure supports LCM and IPC via remoteproc and RPMsg, but does not include source code required to support other non-Linux platforms running on the remote processor, such as bare-metal or FreeRTOS applications. The OpenAMP framework provides this missing functionality by implementing the infrastructure required for FreeRTOS and bare-metal environments to communicate with a Linux master in an AMP system.

Communication Channel

It is common for the master processor in an AMP system to bring up software on the remote cores on a demand-driven basis. These cores then communicate using inter-process communication (IPC). It allows the master processor to off-load work to the remote processors. Such activities are coordinated and managed by the OpenAMP framework.

To create a communication channel between master and remote:

1. The Linux udev device manager loads the zynqmp-r5-remoteproc kernel module pointing to the firmware r5_1_firmware to be loaded on RPU-1. The firmware is stored in the /lib/firmware directory.
2. The zynqmp-r5-remoteproc module configures all of the resources requested by the firmware based on the resource table section inside the firmware binary. This includes creating the vring data structures using shared DDR memory (reserved in the device tree).
3. The master boots the remote processor by loading the firmware for RPU-1 into TCM-1 as specified in the resource table.
4. The remote processor creates and initializes the virtIO resources and the RPMsg channels for the master.
5. The user loads the rpmsg_user_dev_driver kernel module which receives the RPMsg channels created by the remote and invokes the callback channel. The master responds to the remote context, acknowledging the remote processor and application.
6. The remote invokes the RPMsg channel that was registered. The RPMsg channel is now established, and both sides can use RPMsg calls to communicate.

To destroy a communication channel between master and remote:

1. The master application sends an application-specific shutdown message to the remote application.
2. The remote application cleans up its resources and sends an acknowledgment to the master.

3. The remote frees the `remoteproc` resources on its side.
4. The master shuts down the remote and frees the `remoteproc` resources on its side.

Message Passing

At a high-level the APU master sends a message to the RPU-1 remote e.g., a request to read the APM counters (see Performance Monitor Client Library, page 28). The RPU-1 remote in turn sends back a message to the APU master e.g., the data payload for the requested APM counters (see Performance Monitor Applications, page 32).

The message passing mechanism itself is unidirectional, therefore two separate, shared vring buffers are used: vring0 is used by the APU master to send messages to the RPU-1 remote and vring1 is used by the RPU-1 remote to send messages back to the APU master. Figure 7-7 shows the latter case but same concept applies to the first case.

To signal messaging events such as availability of new data or consumption of data from the master to the remote and vice versa, an inter processor interrupt (IPI) is used. The IPI block inside the PS provides the ability for any processing unit (source) to interrupt another processing unit (destination). The source sets the interrupt bit in its trigger register corresponding to the destination. The destination clears the interrupt which is reflected in the source's observation register. The RPMsg driver on the APU master side and the OpenAMP framework on the RPU-1 remote side build on top of IPI.

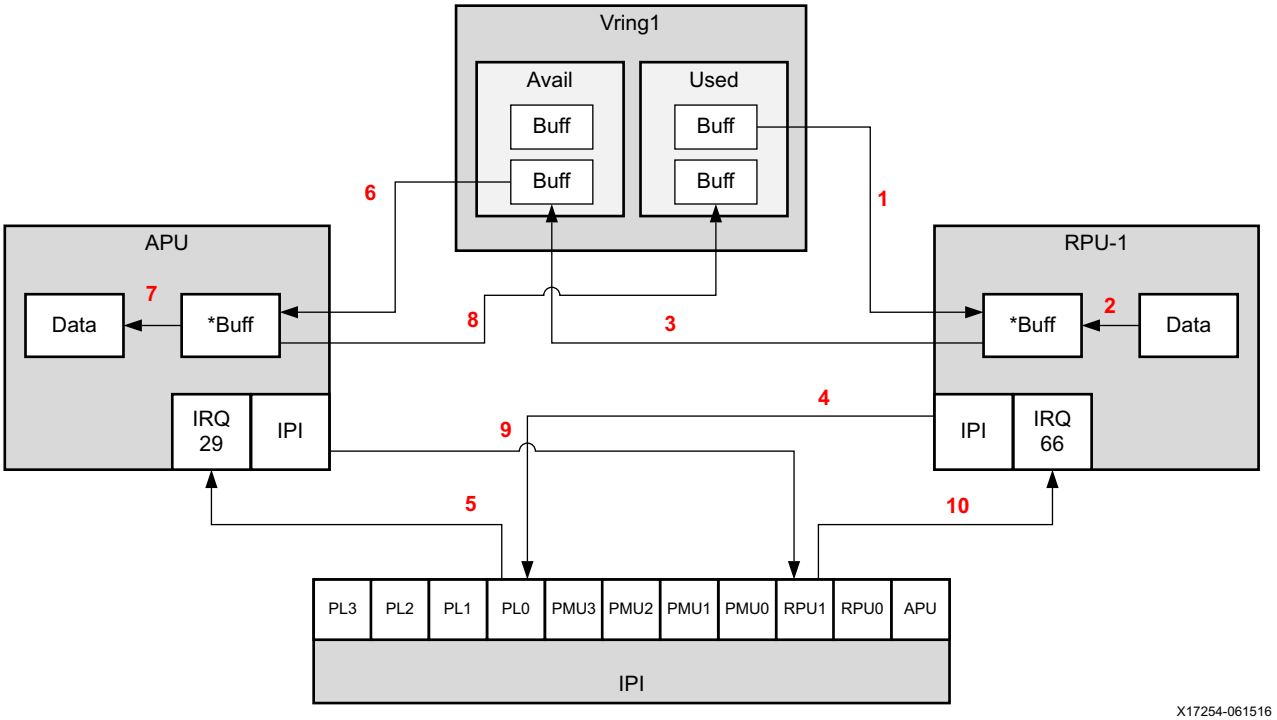

Figure 7-7: **APU/RPU-1 Communication via RPMsg**

Hardware Platform

Introduction

This chapter describes the reference design hardware architecture. Figure 8-1 shows a block diagram of the design components inside the PS and PL on the ZCU102 base board.

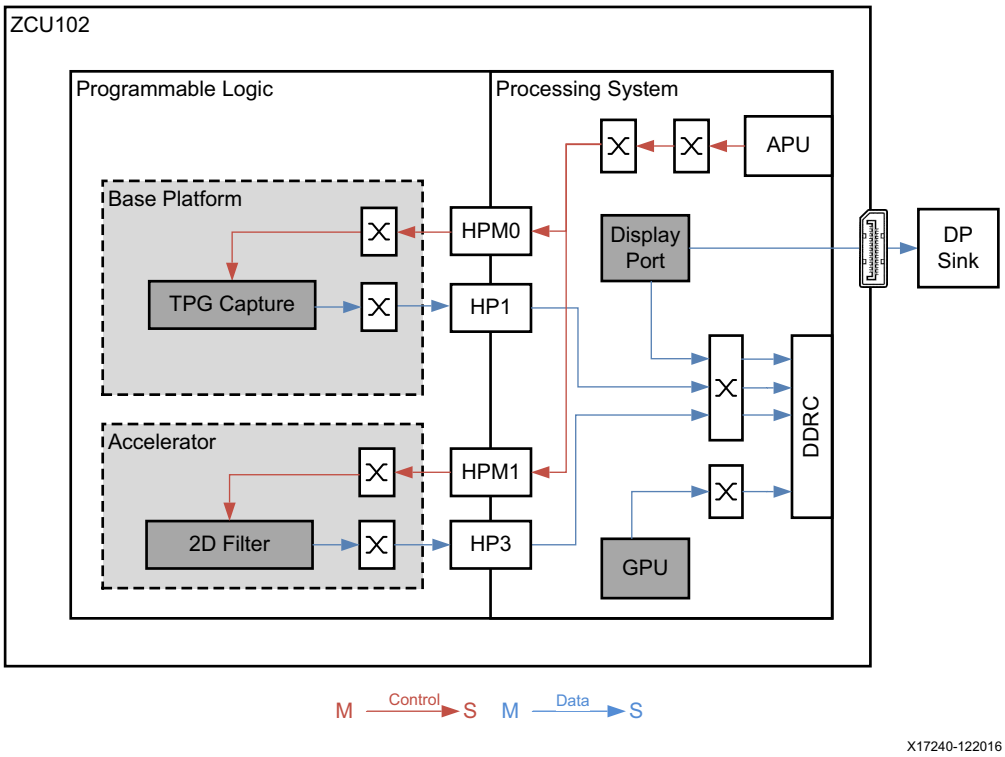

Figure 8-1: **Hardware Platform and Generated Accelerator/Data Motion Network**

At a high-level, the design consists of three different video pipelines:

- Test pattern generator (TPG) capture pipeline (PL)
- Memory-to-memory (M2M) processing pipeline (PL)
- Display pipeline (PS + ZCU102)

The block diagram also highlights the two partitions of the design:

- Hardware base platform, which mainly consist of the I/O interfaces such as DisplayPort output (this part of the design is fixed).
- Hardware accelerator and corresponding data motion network (this part is generated by the SDSoC tool and is automatically added into the PL design).

The individual components inside the four video pipelines are described in more detail in the following sections.

Video Pipelines

Three types of video pipelines are differentiated in this design:

- The capture pipeline produces video frames internally (such as the TPG). The captured video frames are written into memory.
- The memory-to-memory (M2M) pipeline reads video frames from memory, does certain processing, and then writes the processed frames back into memory.
- The output pipeline reads video frames from memory and sends the frames to a sink. In this case the sink is a display and therefore this pipeline is also referred to as a display pipeline.

TPG Capture Pipeline

The TPG capture pipeline is shown in Figure 8-2.

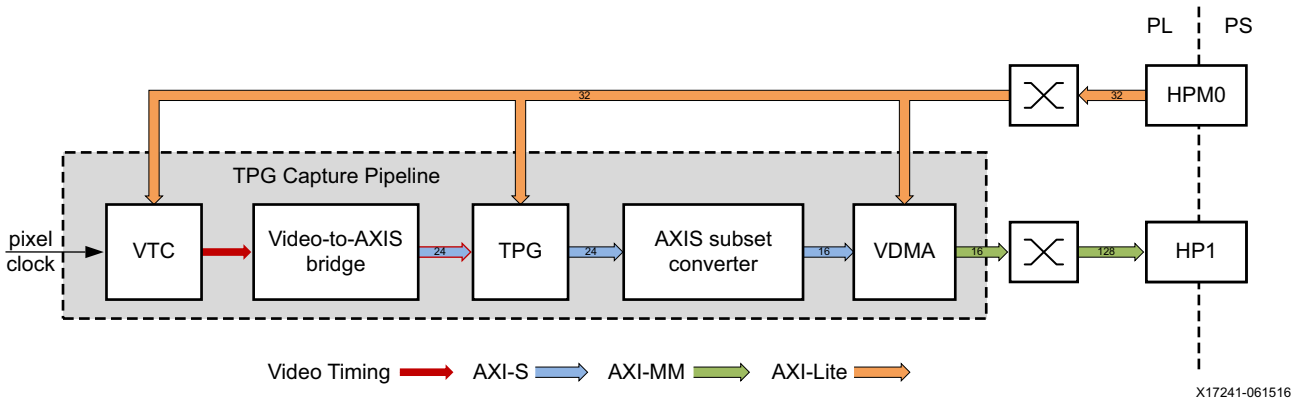

Figure 8-2: **TPG Video Capture Pipeline**

This pipeline consists of three main components, each of them controlled by the APU via an AXI-Lite based register interface:

- The video timing controller (VTC) generates video timing signals including horizontal and vertical sync and blanking signals. The timing signals are converted to AXI-Stream using the video-to-AXI-Stream bridge with the data bus tied off. The video timing over AXI-Stream bus is connected to the input interface of the TPG, thus making the TPG behave like a timing-accurate video source with a set frame rate as opposed to using the free-running mode.
- The test pattern generator (TPG) can be configured to generate various test patterns including color bars, zone plates, moving ramps, moving box etc. The color space format is configurable and set to YUV 4:2:2 in this design. Therefore only the lower 16 bits of the 24-bit output AXI-Stream interface are used; the upper 8 bits are discarded in the AXI-Stream subset converter IP core.
- The video DMA (VDMA) consists of a write channel that converts the AXI-Stream data from the TPG to AXI-Memory-mapped (s2mm) for direct memory access into DDR memory. The output is connected to the HP1 high performance PS/PL interface via an AXI interconnect. For each video frame transfer, an interrupt is generated. The VDMA is operated in park mode and software is responsible for moving video buffers and synchronizing access between pipeline endpoints.

The pipeline can be configured for different video resolutions and frame rates at run-time. In this design, you can choose between 720p60, 1080p60, and 2160p30. The bus format is fixed at YUV 4:2:2, 8 bits per component and one pixel per clock. The bus width can be either 16-bit or 24-bit in which case the upper 8 bits are padded (Figure 8-3).

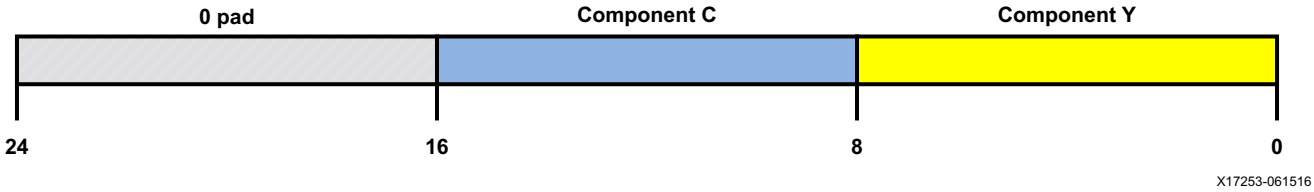

Figure 8-3: **AXI-Stream Data Bus Encoding for YCrCb**

A video frame period consists of active and non-active time intervals, also called blanking intervals. During active intervals, video pixel data is transferred. Historically, CRT monitors used horizontal and vertical blanking intervals to reposition the electron beam from the end of a line to the beginning of the next line (horizontal blanking) or from the end of a frame (bottom right) to the start of a frame (top left) (vertical blanking). The horizontal and vertical blanking intervals are further sub-divided into front-porch, sync, and back-porch as shown in Figure 8-4. See *AXI4-Stream Video IP and System Design Guide* (UG934) [Ref 12] for more details on video timing parameters.

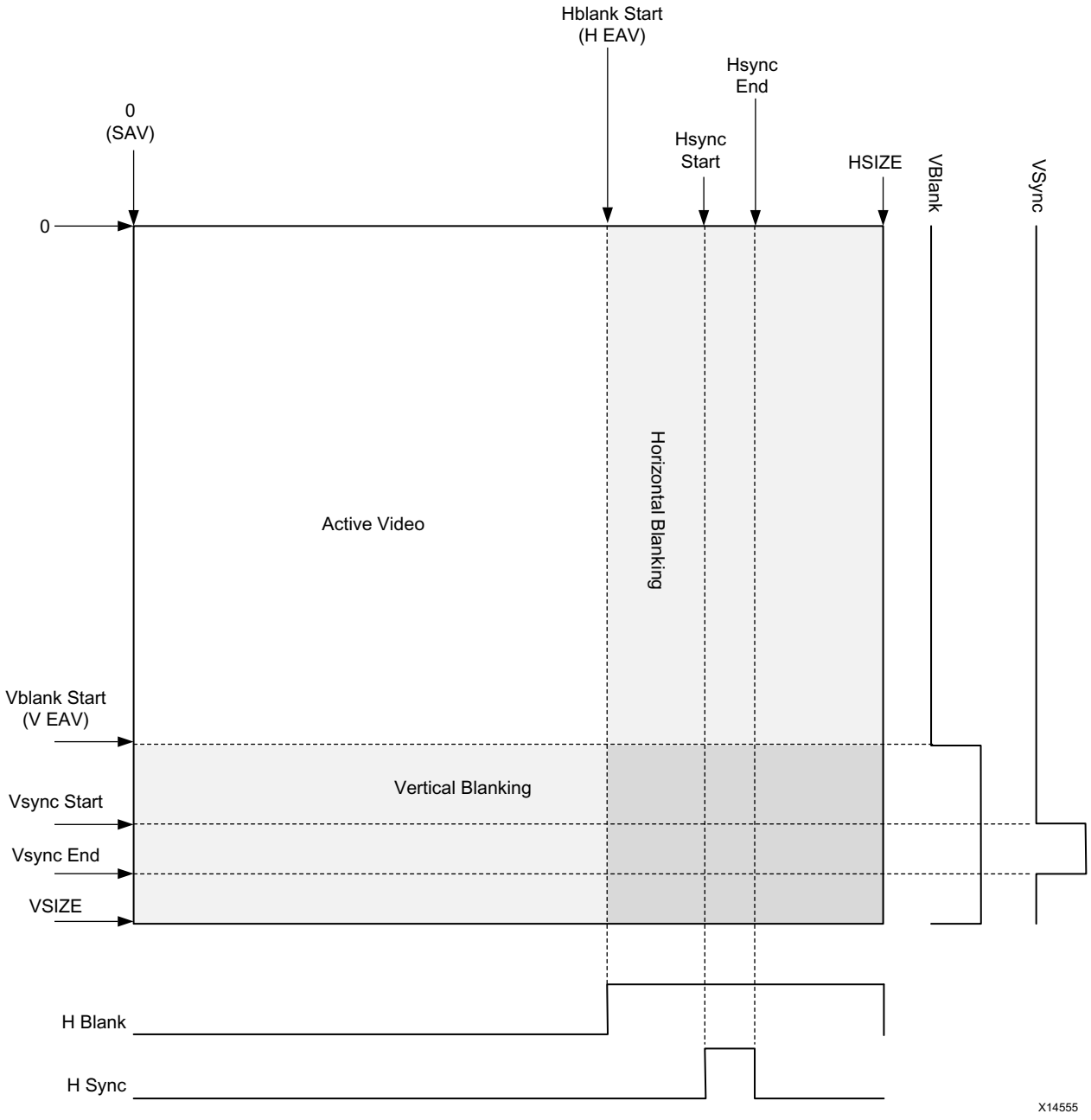

Figure 8-4: **Active and Blanking Intervals of a Video Frame**

For example, a standard 2160p30 video stream has a resolution of 3840 horizontal pixels and 2160 lines, and is transferred at a frame rate of 30 frames per second (fps). The *p* stands for progressive which means that a full frame is transferred each frame period. The vertical blanking accounts for 90 pixels per line (8+10+72) and the horizontal blanking for 560 lines per video frame (176+88+296). The required pixel clock frequency is calculated as:

$$(3840 + 560) \times (2160 + 90) \times 30 \text{ Hz} = 297 \text{ MHz}$$

The VTC drives the horizontal and vertical sync and blanking signals, as well as active video (data enable) based on this scheme. The pixel clock is provided by an I2C-programmable Si570 clock synthesizer on the ZCU102 board. The pixel clock is shared with the display pipeline whereas the display pipeline controls the clock frequency and therefore the TPG resolution and frame rate is always in sync with the display.

Note that the exact video timing parameters for each supported resolution are typically stored inside the monitor's extended display identification data (EDID). The display driver queries the EDID from the monitor and then sets the timing parameters accordingly.

2D Filter M2M Pipeline

The memory-to-memory (m2m) pipeline with the 2D convolution filter (see Figure 8-5) is entirely generated by the SDSoC tool based on a C-code description. The 2D filter function is translated to RTL using the Vivado HLS compiler. The data motion network used to transfer video buffers to/from memory and to program parameters like video dimensions and filter coefficients is inferred automatically by the SDSoC compiler.

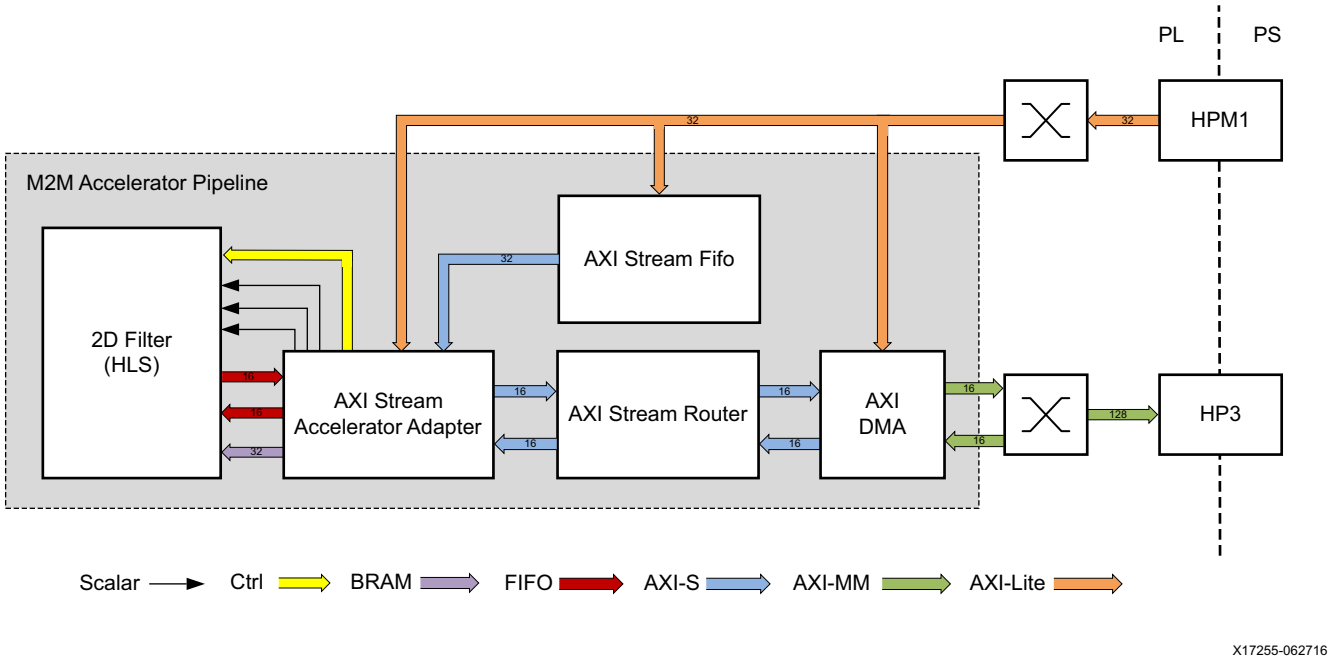

Figure 8-5: **M2M Processing Pipeline Showing Hardware Accelerator and Data Motion Network**

The HLS generated 2D filter accelerator connects to an AXI-Stream accelerator adapter that drives all inputs and captures all outputs. The adapter has two FIFO interfaces for streaming video data to and from the accelerator. The adapter converts the video data streams to the AXI-Stream format to connect to the AXI DMA Rx and Tx channels via the AXI-Stream router. The filter coefficients are passed via a BRAM interface and the data mover is a simple AXI-Stream FIFO. The width, height, and stride parameters are simple control signals (scalars) driven by the adapter.

Note that the m2m pipeline is not video-timing accurate. The DMA reads and writes video frames without inserting any horizontal blanking in between video lines. An interrupt is issued by the DMA engine separately for the read and write channels upon completion of a frame read/write operation. The QoS traffic class is therefore set to best effort instead of video (see Quality of Service, page 43).

The DMA engine translates AXI-Stream data to AXI-Memory-mapped. The output is connected to the HP3 high performance PS/PL interface via the AXI interconnect. A separate HP port is used compared to the two capture pipelines to split the traffic across two dedicated DDRC ports. For the AXI-Lite control interfaces, a separate PS/PL interface, HPM1 is used as opposed to HPM0 which used for the base platform modules.

DP Tx Display Pipeline

The display pipeline (see Figure 8-6) is configured to read video frames from memory via two separate channels: one for video, the other for graphics. The video and graphics layers are alpha-blended to create a single output video stream that is sent to the monitor via the DisplayPort transmitter. This design does not use the audio feature of the DisplayPort controller, therefore it is not discussed in this user guide. The major components used in this design, as shown in the figure, are:

- DisplayPort DMA (DPDMA)
- Audio/Video (A/V) buffer manager
- Video blender
- DisplayPort (DP) transmitter
- PS-GTR gigabit transceivers

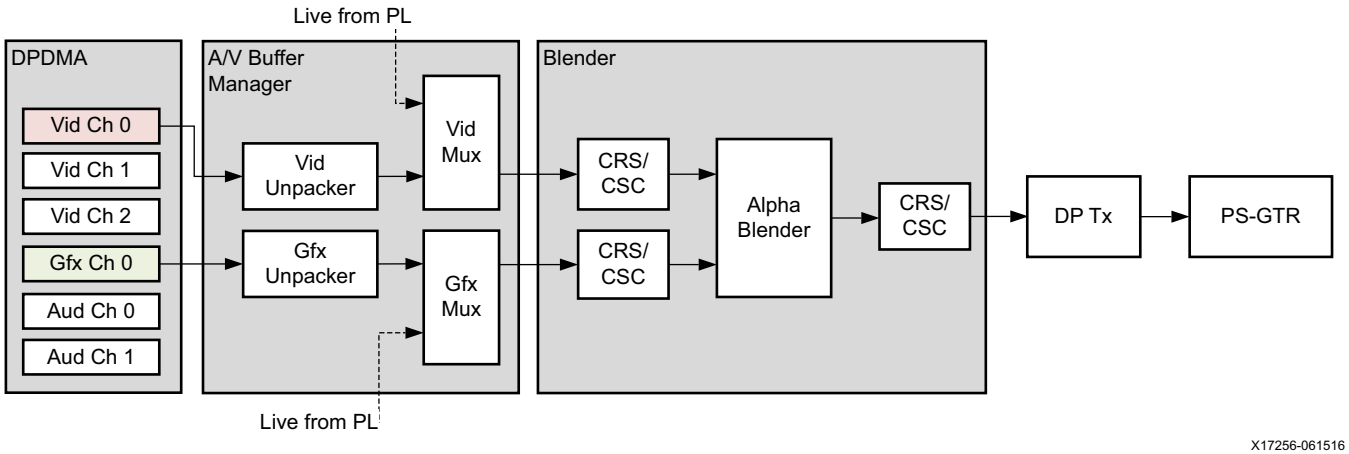

Figure 8-6: **Display Pipeline Showing DPDMA, A/V Buffer Manager, Video Blender, and DP Transmitter**

The DPDMA is a 6-channel DMA engine that fetches data from memory and forwards it to the A/V buffer manager. The video layer can consist of up to three channels, depending on

the chosen pixel format whereas the graphics layer is always a single channel. The used pixel formats are described in Video Buffer Formats, page 39. The remaining two channels are used for audio.

The A/V buffer manager can receive data either from the DPDMA (non-live mode) or from the PL (live mode) or a combination of the two. In this design only non-live mode is used for both video and graphics. The three video channels feed into a video pixel unpacker and the graphics channel into a graphics pixel unpacker. Because the data is not timed in non-live mode, video timing is locally generated using the internal video timing controller. A stream selector forwards the selected video and graphics streams to the dual-stream video blender.

The video blender unit consists of input color space converters (CSC) and chroma re-samplers (CRS), one pair per stream, a dual-stream alpha blender, and one output color space converter and chroma re-sampler. The two streams have to have the same dimensions and color format before entering the blender. The alpha blender can be configured for global alpha (single alpha value for the entire stream) or per pixel alpha. A single output stream is sent to the DisplayPort transmitter.

The DisplayPort transmitter supports the DisplayPort v1.2a protocol. It does not support multi-stream transport or other optional features. The DP transmitter is responsible for managing the link and physical layer functionality. The controller packs video data into transfer units and sends them over the main link. In addition to the main link, the controller has an auxiliary channel, which is used for source/sink communication.

Four high-speed gigabit transceivers (PS-GTRs) are implemented in the serial input output unit (SIOU) and shared between the following controllers: PCIe, USB 3.0, DP, SATA, and SGMII Ethernet. The DP controller supports up to two lanes at a maximum line rate of 5.4 Gb/s. The link rate and lane count are configurable based on bandwidth requirements.

For more information on the DisplayPort controller and the PS-GTR interface, see chapters 31 and 27 in *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 9].

Clocks, Resets and Interrupts

Table 8-1 lists the clock frequencies of key PS components.

Table 8-1: **Key PS Component Clock Frequencies**

PS Component	Clock Frequency
APU	1,100 MHz
RPU	500 MHz
GPU	500 MHz
DDR	533 (1,066) MHz
PL2 (clk_pl_2)	300 MHz

Table 8-2 identifies the five clock domains of the PL design, their source, and their clock frequencies.

Table 8-2: **System Clocks**

PL Clock Domain	Clock Source	Clock Frequency
clk50	clk_pl_2	50 MHz
clk75	clk_pl_2	75 MHz
clk150	clk_pl_2	150 MHz
clk300	clk_pl_2	300 MHz
si570_clk	External differential input clock	User programmable

The *clk_pl_2* clock is provided by the RPLL inside the PS and is used as reference input clock for the *clk_wiz_1* mixed-mode clock manager (MMCM) instance inside the PL. This clock does not drive any loads directly. The *clk_wiz_1* instance is used to de-skew the clock and to provide four phase-aligned output clocks, *clk50, clk75, clk150,* and *clk300*.

The *clk50* clock is generated by the *clk_wiz_1* MMCM instance. It is used to drive most of the AXI-Lite control interfaces in the PL such as TPG, VTC, VDMAs, etc. AXI-Lite interfaces are typically used in the control path to configure IP registers and therefore can operate at a lower frequency than data path interfaces.

The *clk300* clock is generated by the *clk_wiz_1* MMCM instance. It is used to drive the AXI-MM and AXI-Stream interfaces of the capture pipelines in the PL. These interfaces are in the data path and therefore need to support the maximum performance of 2160p30 which roughly corresponds to a 300 MHz clock. The AXI-Lite interfaces of HLS based IP cores as well as SDSoC generated modules are also based on *clk300* as opposed to *clk50* as HLS IPs typically share a common input clock between control and data interfaces.

The *m2m* pipeline clock can be selected by the user from the SDSoC GUI when moving the accelerator to hardware. The available options are clk75, clk150, and clk300 which roughly correspond to 720p60, 1080p60, and 2160p30. The same *clk_wiz_1* MMCM instance is used to generate these clocks.

The *si570_clk* clock is an external clock generated by the *Si570* clock synthesizer on the ZCU102 board. It is used to drive the display controller as well as the VTC providing timing for the TPG. The *Si570* is programmed at run-time via I2C and the clock is configured to match the desired display resolution. The Video-to-AXI-S bridge in the TPG capture pipeline handles the clock domain crossing from *si570_clk* to *clk300*. In the display pipeline, the *si570_clk* is provided through the DP live video input interface to the internal timing generator.

Master reset *pl_resetn0* is generated by the PS during boot and is used as input to the two *proc_sys_reset* modules in the PL. Each module generates synchronous, active-Low and active-High interconnect and peripheral resets that drive all PL IP cores synchronous to the respective *clk50* and *clk300* clock domains.

Apart from these system resets, there is an asynchronous reset driven by a PS GPIO pin. The respective device driver controls this reset which can be toggled at run-time. This is used to reset the TPG IP core. Table 8-3 summarizes the PL resets used in this design.

Table 8-3: **System and User Resets**

Reset Source	Purpose
PS pl_reset0	PL reset for proc_sys_reset modules
proc_sys_reset_clk50	Synchronous resets for clk50 clock domain
proc_sys_reset_clk300	Synchronous resets for clk300 clock domain
PS GPIO 79	Asynchronous reset for TPG

Table 8-4 lists the PL-to-PS interrupts used in this design. All but one of the PL interrupts are associated with Video DMA (*axi_vdma_1*, *axi_vdma_2*) or AXI DMA (*dm_0*) instances to signal completion of a video frame read (mm2s) or write (s2mm) operation to the APU.

Table 8-4: **PL-to-PS Interrupts**

Instance	Port	ID
axi_vdma_2	s2mm_introut	89
axi_iic_1	iic2intc_irpt	90
axi_vdma_1	s2mm_introut	91
dm_0	mm2s_introut	92
dm_0	s2mm_introut	93

I2C Bus Topology

I2C is a two-wire bus for attaching low-speed peripherals. It uses two bidirectional open-drain lines, SDA (serial data) and SCL (serial clock), pulled up with resistors. In standard mode, a 7-bit address space and a 400 kHz bus speed are used. In this reference design, PS I2C and AXI I2C (PL) controllers are used to configure several I2C slaves. The I2C topology limited to the devices used in this design is shown in Figure 8-7.

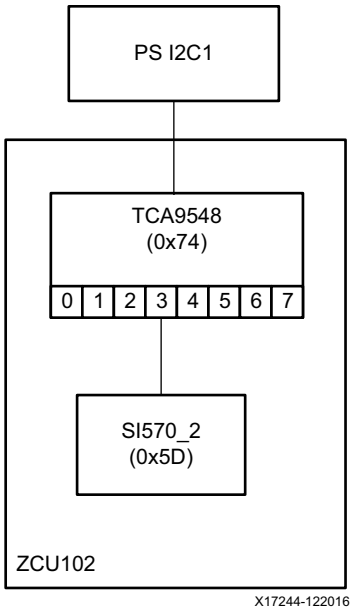

Figure 8-7: **I2C Bus Topology**

The PS I2C1 controller is connected via MIO to an 8-channel I2C multiplexer (TCA9548) on the ZCU102 board at address `0x74`. The I2C multiplexer at address `0x74` has a Si570 programmable clock synthesizer connected to channel 3 at address `0x5D`.

The Si570 clock synthesizer is connected to the PL via a differential clock pair. It generates the pixel clock to drive the PS display controller and the VTC connected to the TPG based on the desired video resolution.

Auxiliary Peripherals

Other low-speed auxiliary interfaces used in this design are as follows:

- UART (2x PS): the first PS UART is owned by the APU and prints Linux boot and debug messages to the serial console. A serial login prompt is provided in addition to the `fbconsole` login prompt. The second PS UART is owned by RPU-0 and prints

application messages to the serial console periodically. Both UARTs are connected to a quad USB-UART bridge on the ZCU102 board.

- USB 2.0 (1x PS) is used to connect a mouse and keyboard to the ZCU102 board to operate and navigate the application.
- I2C (1x PS) is used to configure various slave peripherals on the ZCU102 board (see I2C Bus Topology, page 72).
- GPIO (1x PS) is used to drive the reset signals of the TPG in the PL (see Clocks, Resets and Interrupts, page 70).
- SD (1x PS) is the boot device and holds the entire reference design image including FSBL, PL bitstream, u-boot, Linux kernel, root file system, RPU-1 firmware, and APU/RPU-0 applications (see Boot Process, page 35.)

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

For continual updates, add the Answer Record to your myAlerts.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

References

The most up-to-date information for this design is available at these websites:

http://www.wiki.xilinx.com/Zynq+UltraScale+MPSoC+Base+TRD

https://www.xilinx.com/products/boards-and-kits/zcu102.

These documents and sites provide supplemental material:

1. *SDSoC Environment User Guide* (UG1027)
2. *SDSoC Environment User Guide - Platforms and Libraries* (UG1146)
3. *ZCU102 Evaluation Board User Guide* (UG1182)
4. Performing Convolution Operations website
5. OpenCV - 2D Filter API website
6. *Vivado Design Suite User Guide - High-Level Synthesis* (UG902)

7. *Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries* (XAPP1167)
8. *Zynq UltraScale+ MPSoC Software Developer Guide* (UG1137)
9. *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085)
10. VESA E-EDID Standard website
11. Qt5 Embedded Linux website
12. *AXI4-Stream Video IP and System Design Guide* (UG934)
13. Recommendation ITU-R BT.656-4 website
14. FRU Information Storage for IPMI website
15. *OpenAMP Framework for Zynq Devices Getting Started Guide* (UG1186)
16. L. Pinchart, *Anatomy of an Atomic KMS Driver*, Kernel Recipes, Paris, 2015
17. *UltraFast Embedded Design Methodology Guide* (UG1046)
18. L. Pinchart, *V4L2 on Steroids - The Request API*, Embedded Linux Conference, San Diego, 2016
19. Xilinx V4L2 Pipeline Driver website

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

© Copyright 2016 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.