

Zynq UltraScale+ MPSoC Base Targeted Reference Design

User Guide

UG1221 (v2018.3) December 10, 2018



Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/10/2018	2018.3	Added information about xlnxvideosrc and xlnxvideosink GStreamer plugins, information about gstdx GStreamer plugins, and information about the clocking wizard used in the TPG capture pipeline. Updated Table 2-2 and Design Components in Chapter 2 . Updated Video Library and GStreamer Video Library , added Xilinx Video Source and Sink GStreamer Plugins and SDx GStreamer Plugins in Chapter 3 . Updated Video Pipelines and Clocks, Resets and Interrupts in Chapter 8 . Updated Figure 3-1 and Figure 3-2 , added Figure 3-12 and Figure 3-13 , updated Figure 6-2 , Figure 7-2 , and Figure 7-4 .
07/13/2018	2018.2	Released with Vivado Design Suite 2018.2 with no changes from previous version.
06/14/2018	2018.1	Ported reference design software stack to use GStreamer multimedia framework. Added file source support. Updated for Vivado Design Suite 2018.1 SDSoC tools. Updated Table 2-2 . Updated Figure 1-1 , Figure 1-2 , Figure 3-1 , Figure 3-2 , added Figure 3-6 , deleted Figure 3-9 , added Figure 3-9 and Figure 3-10 , updated Figure 3-11 , updated Figure 7-1 , Figure 7-2 , and Figure 7-3 . Updated Design Components in Chapter 2 , Added File Source Settings Panel in Chapter 3 . Updated Video Library in Chapter 3 .
02/06/2018	2017.4	Updated Figure 1-2 , Figure 3-1 , Figure 3-5 , Figure 7-1 , Figure 7-2 , Figure 7-3 , and Figure 8-7 . Updated APU Design Components in Chapter 2 and Table 2-2 . Added IMX274 image sensor controls under CSI Settings Panel , updated Performance Monitor Client Library , and function names under Optical Flow GStreamer SDx Plugin in Chapter 3 . Added IMX274 Image Sensor driver to Table 7-1 and deleted note under Vision in Chapter 7 . Updated IDs in Table 8-4 and added the paragraph following the table.
09/20/2017	2017.2	Added information about the FMC daughter card with an image sensor connected via MIPI CSI-2 Rx throughout the document. Added CSI Settings Panel in Chapter 3 . Updated I2C Bus Topology in Chapter 8 . Updated Figure 1-2 , Figure 1-3 , Figure 3-1 , Figure 3-2 , Figure 4-2 , Figure 6-2 , Figure 7-1 , Figure 7-1 , Figure 8-1 , and Figure 8-10 . Updated Table 7-1 , and Table 8-2 through Table 8-4 .
07/19/2017	2017.1	Updated Reference Design Overview . Add description of the Optical Flow GStreamer SDx Plugin and the Optical Flow M2M Pipeline . Updated performance metrics based on 2160p60. Updated QoS for HDMI Tx. Updated V4L driver stack for HDMI Rx capture pipeline (added HDMI Rx Capture Pipeline). Updated DRM driver stack for HDMI Tx display pipeline (added HDMI Tx Display Pipeline). Updated Global Address Map . Updated Clocks, Resets and Interrupts . Updated I2C Bus Topology . Updated Auxiliary Peripherals . Added Demo Mode Settings Panel description.
03/22/2017	2016.4	Released with Vivado Design Suite 2016.4 with no changes from previous version.

Date	Version	Revision
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 <code>perfapm</code> 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 .
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 8-11 .
06/29/2016	2016.1	Initial Xilinx release.

Table of Contents

Revision History	2
Chapter 1: Introduction	
Zynq UltraScale+ MPSOC Overview	7
Reference Design Overview	8
Key Features	11
Chapter 2: Reference Design	
Design Modules	13
Design Components	15
Chapter 3: APU Application (Linux)	
Introduction	18
GUI Application	20
Video Library	26
GStreamer Video Library	27
Performance Monitor Client Library	32
Xilinx Video Source and Sink GStreamer Plugins	32
SDx GStreamer Plugins	34
Chapter 4: RPU-1 Software Stack (Bare-metal)	
Introduction	38
Performance Monitor Library	39
Performance Monitor Applications	41
Bare-metal BSP	41
Chapter 5: RPU-0 Software Stack (FreeRTOS)	
Introduction	42
Heartbeat Application	43
FreeRTOS BSP	43
Chapter 6: System Considerations	
Boot Process	44
Global Address Map	46
Video Buffer Formats	48

Performance Metrics	50
---------------------------	----

Chapter 7: APU Software Platform

Introduction	54
Video	56
Display	61
Graphics	66
Vision	69
Inter-process Communication	69

Chapter 8: Hardware Platform

Introduction	74
Video Pipelines	75
Clocks, Resets and Interrupts	86
I2C Bus Topology	89
Auxiliary Peripherals	90

Appendix A: Additional Resources and Legal Notices

Xilinx Resources	91
Solution Centers	91
Documentation Navigator and Design Hubs	91
References	92
Please Read: Important Legal Notices	93

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:

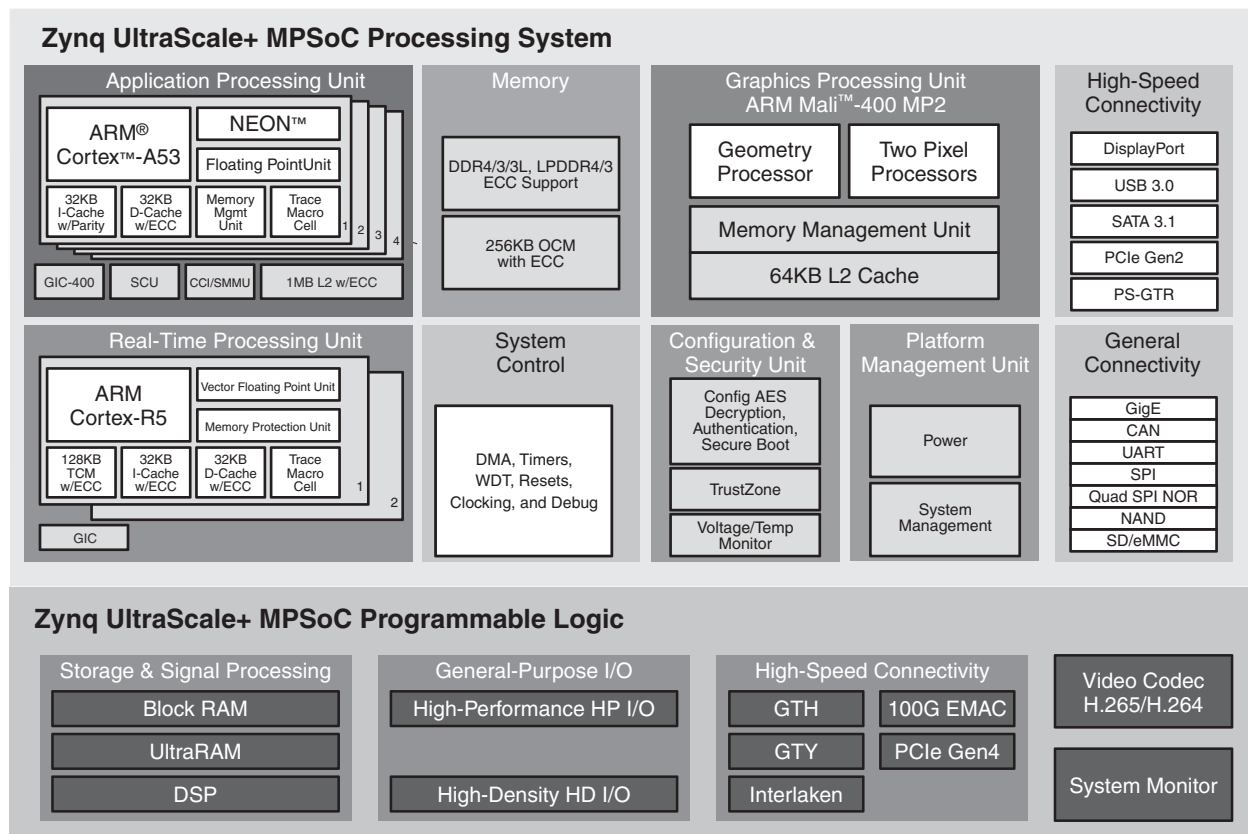
1. Ultra HD video stream real-time processing up to 60 frames per second.
2. Freed-up CPU resources for application-specific tasks.

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).



UG1221_062316

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

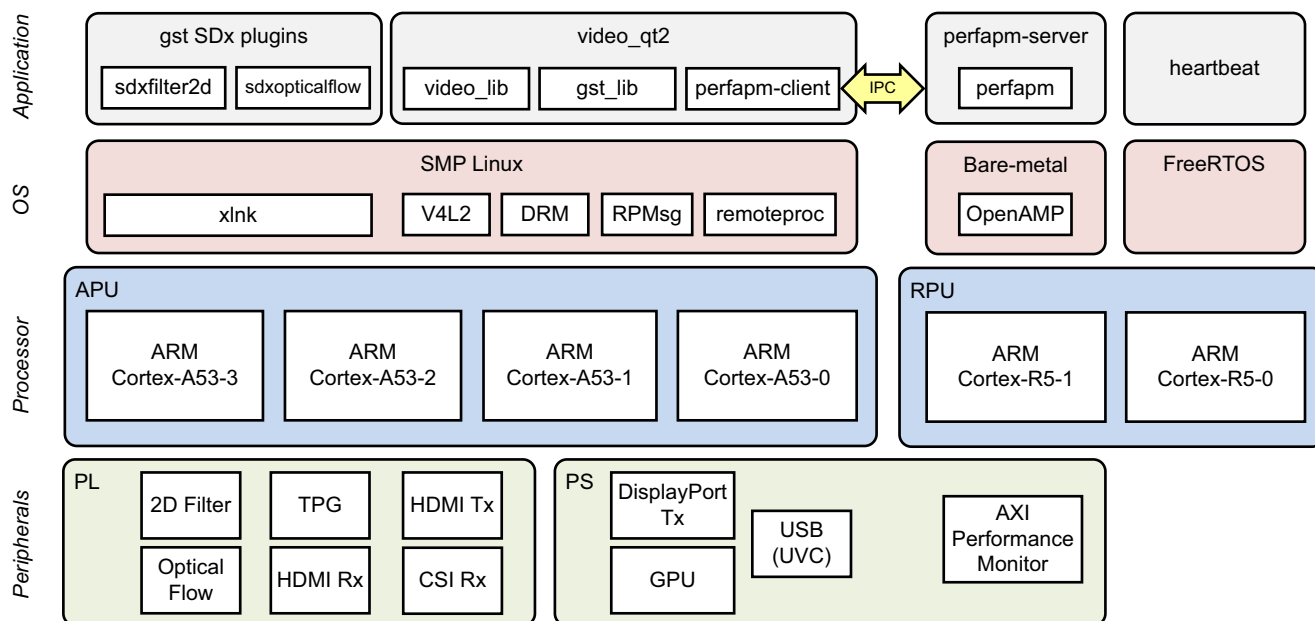
- High-speed peripherals
 - PCIe root complex (Gen1 or Gen2) and endpoint (x1, x2, and x4 lanes)
 - USB 3.0/2.0 with host, device, and OTG modes
 - SATA 3.1 host
- Low-speed peripherals
 - Gigabit Ethernet, CAN, UART, SPI, Quad-SPI, NAND, SD/eMMC, I2C, and GPIO
- Platform management unit (PMU)
- Configuration 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 44](#)). 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 69](#) 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.



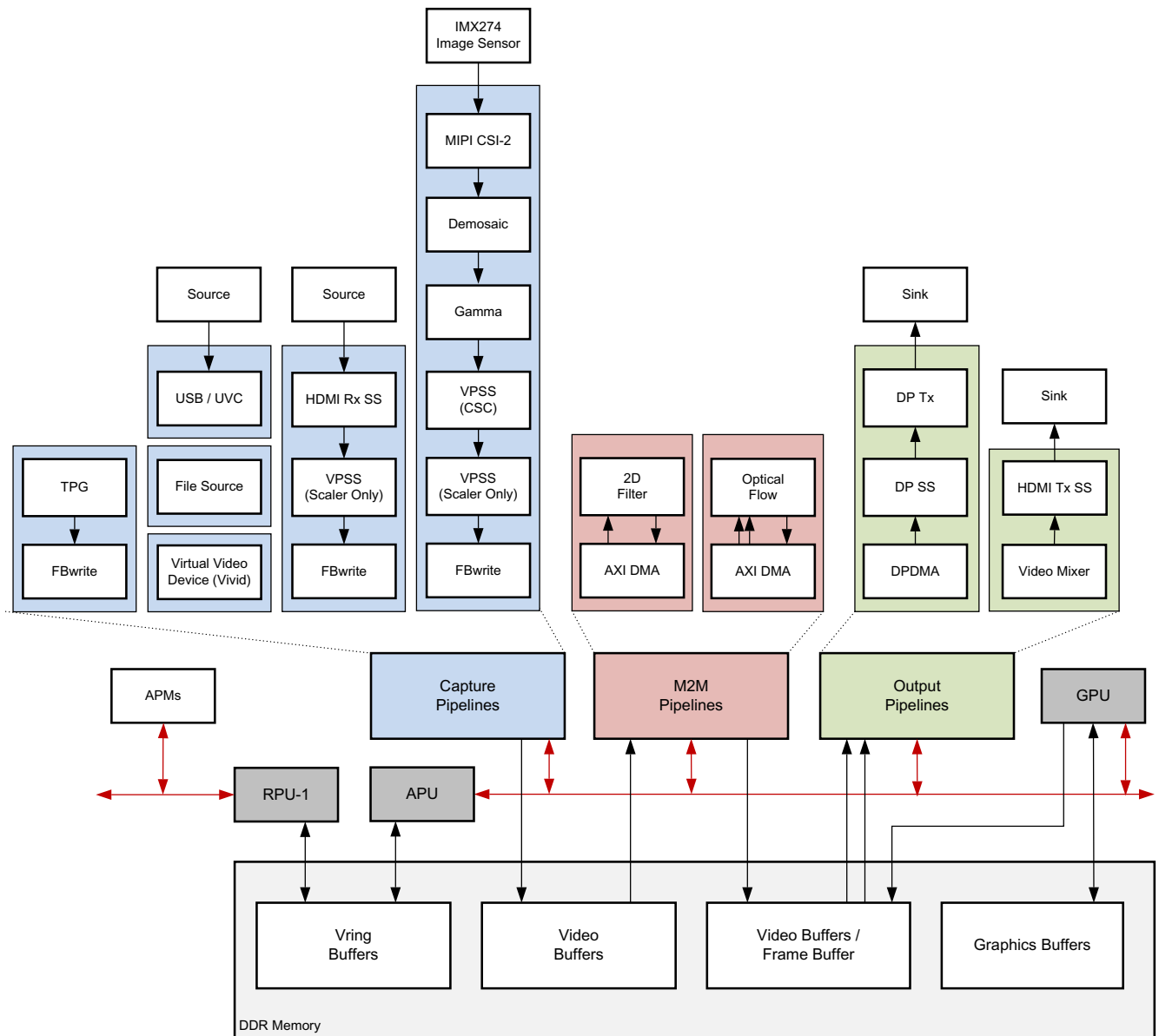
X17236-051018

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 into DDR memory from: an HDMI source connected through the PL, an image sensor on an FMC daughter card connected via MIPI CSI-2 Rx through the PL, a test pattern generator implemented inside the PL (TPG (PL)), a USB webcam using the USB interface inside the PS, and a test pattern generator emulated in pure software (TPG (SW)) as a virtual video device (vivid).
- Memory-to-memory (M2M) pipeline implementing typical video processing algorithms: a programmable 2D convolution filter and a dense optical flow algorithm. The algorithms can be implemented inside the PS as software functions or as hardware accelerators 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 reading video frames from memory and sending them to a monitor via: DisplayPort Tx inside the PS or HDMI Tx through the PL. Both support 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.



X17251-051418

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) [\[Ref 3\]](#))
- LI-IMX274MIPI-FMC image sensor daughter card (optional) [\[Ref 4\]](#)

Xilinx tools:

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

Hardware interfaces and IP:

- GPU
- Video Inputs
 - TPG (PL) - referred to as TPG from here on
 - HDMI Rx
 - MIPI CSI-2 Rx
 - USB Webcam
 - TPG (SW) - referred to as vivid (virtual video device) from here on
 - File (raw input only)
- Video Outputs
 - DisplayPort Tx
 - HDMI Tx
- Video Processing
 - 2D Convolution Filter
 - Dense Optical Flow
- Auxiliary Peripherals
 - SD
 - I2C

- GPIO
- Ethernet
- UART
- USB
- 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
 - 2160p60 (HDMI Tx only)
- Pixel formats:
 - YUV 4:2:2 16 bit for video
 - ARGB 32 bit for graphics

Reference Design

Design Modules

The Base TRD consists of ten design modules (DM) that build on top of each other or are a combination of previous modules where DM10 is the full-fledged reference design as summarized in the previous chapter. The following listing gives a short summary of each of the ten 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 a 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 a USB webcam, and display the video on the monitor through the DisplayPort (DP).

DM6 - PL Video Capture

This module extends DM5 and shows how to capture video from an HDMI source, an image sensor connected via CSI-2 Rx, or a test pattern generator (TPG) implemented in the PL. The video can be displayed via DP Tx through the PS or via HDMI Tx through 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 - Multiple PL Accelerators

This module extends DM8 and shows how to add a second accelerator to the design using the same SDSoC tool flow as in DM8. The algorithm is a dense optical flow computer vision function.

DM10 - Full-fledged Base TRD

This module combines DM2, DM4 and DM9. 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 implementation of the algorithm (DM7) and the two PL accelerators (DM9): 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	DM9
DM1									
DM2									
DM3									
DM4	+		+						
DM5	+								
DM6	+				+				
DM7	+				+	+			
DM8	+				+	+	+		

Table 2-1: Design Module Dependency Matrix (Cont'd)

	DM1	DM2	DM3	DM4	DM5	DM6	DM7	DM8	DM9
DM9	+				+	+	+	+	
DM10	+	+	+	+	+	+	+	+	+

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
- Required hardware for design implementation

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 RPMMsg 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: PMU firmware, 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.
 - `gstsdxbase`: Library and GStreamer base class that provides common functionality for SDSoC GStreamer plugins.
 - `gstsdxallocator`: Library that provides sds memory allocator for SDSoC GStreamer plugins and applications.
 - `gstsdxfilter2d`: GStreamer plugin for a 2D convolution filter image processing kernel. The algorithm can run as pure software implementation using the `OpenCV` libraries or as SDSoC hardware accelerator using the `xfOpenCV` libraries.

- `gstsdsopticalflow`: GStreamer plugin for the Lukas Kanade dense optical flow algorithm implemented as SDSoC hardware accelerator using the `xfOpenCV` libraries.
- `xlnxvideosrc`: GStreamer bin plugin on top of `v4l2src` element to select and configure video source type.
- `xlnxvideosink`: GStreamer bin plugin on top of `kmssink` element to select and configure display sink type.
- `gst_lib`: Library that creates and manages GStreamer pipelines of up to three video components (capture, processing and display).
- `video_lib`: Library that manages the media initialization of video capture pipelines using the media controller framework. It also manages certain aspects of display controller initialization using the DRM framework.
- `video_qt2`: Application that uses the `video_lib` and `gst_lib` 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.
- 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	10
heartbeat		√								√
perfapm-ctl			√							
perfapm-server				√						√

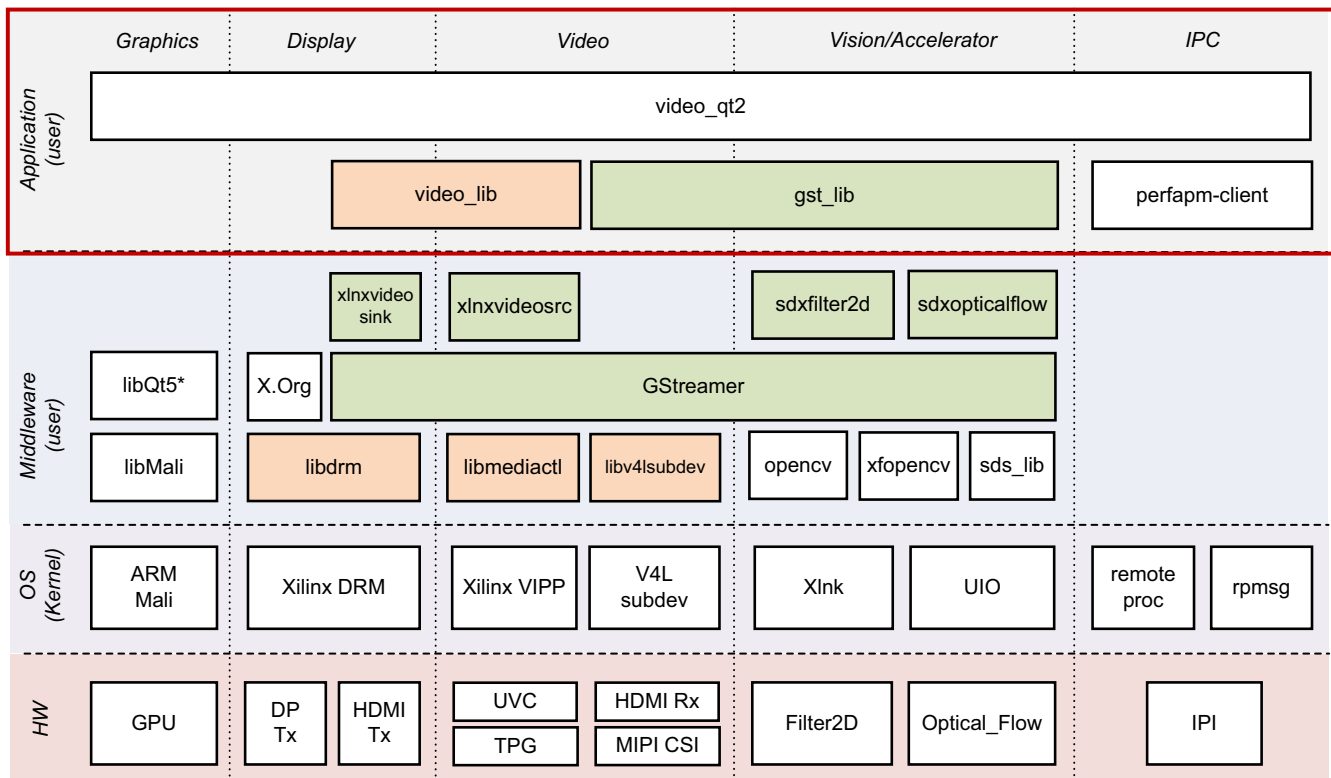
Table 2-2: Design Component to Design Module Mapping (Cont'd)

Design Component	Design Module									
	1	2	3	4	5	6	7	8	9	10
perfapm-client				√						
video_qt2					√	√	√	√	√	√
gstsdxfilter2d							√	√	√	√
gstsdsopticalflow									√	√

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.



X17274-102418

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`)
- GStreamer video library (`gst_lib`)
- Performance monitor client library (`perfapm-client`)

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

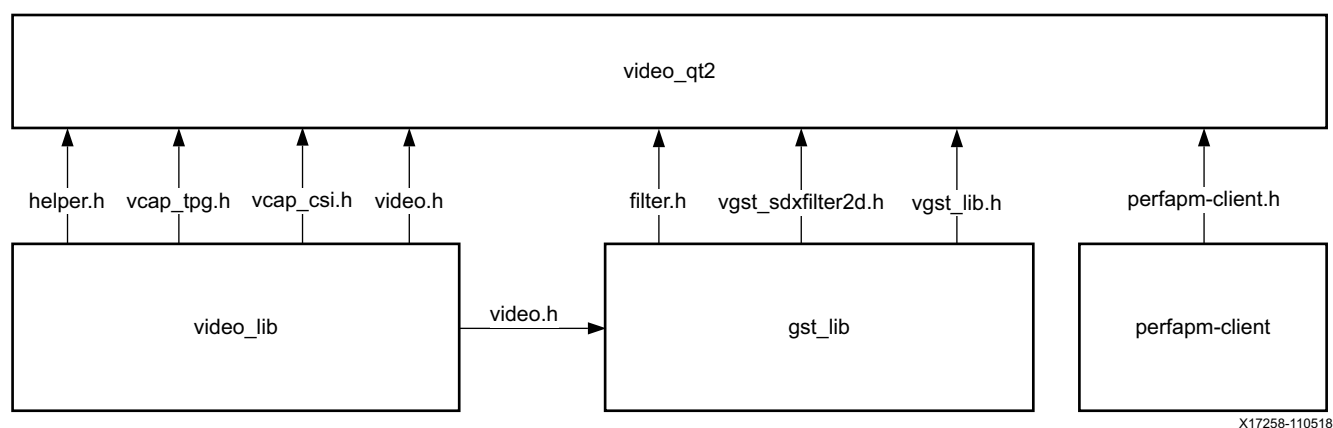


Figure 3-2: Video Application Interfaces

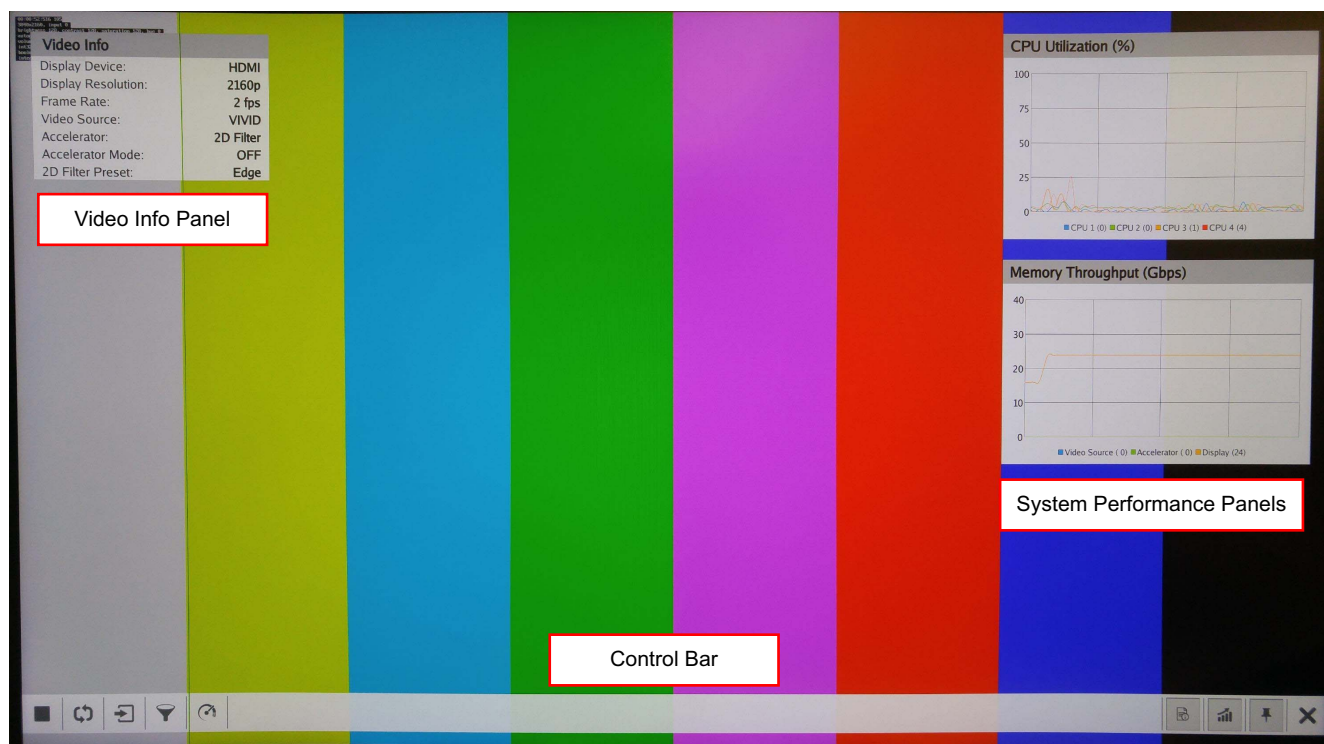
X17258-110518

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)



X17325-061917

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 manual mode
- Start/stop demo mode
- Video source selection and settings
- Accelerator selection and settings
- Accelerator mode selection
- 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 device
- Display resolution
- Frame rate (in fps)
- Video source
- Accelerator
- 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

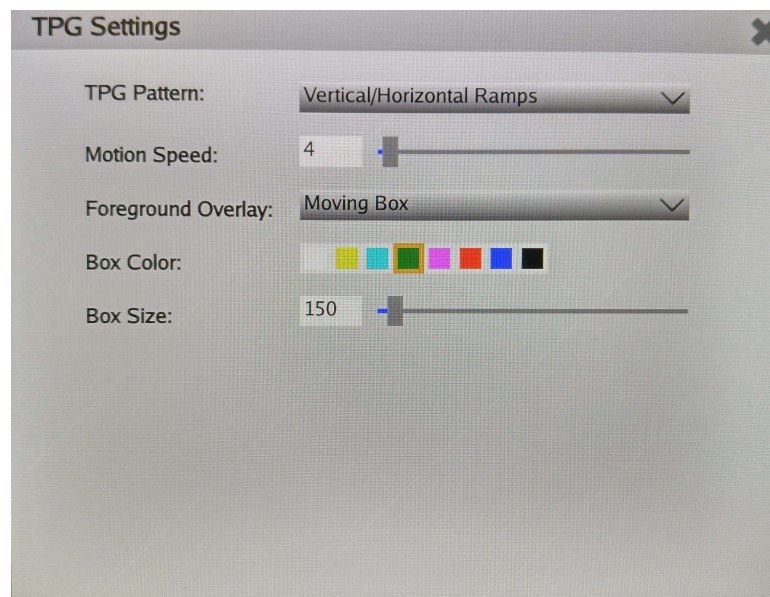


Figure 3-4: TPG Settings Panel

CSI Settings Panel

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

- IMX274 image sensor
 - Test Pattern
 - Exposure
 - Gain
 - Vertical Flip
- Gamma
 - Red gamma correction
 - Green gamma correction
 - Blue gamma correction
- CSC
 - Brightness
 - Contrast
 - Red gain
 - Green gain
 - Blue gain

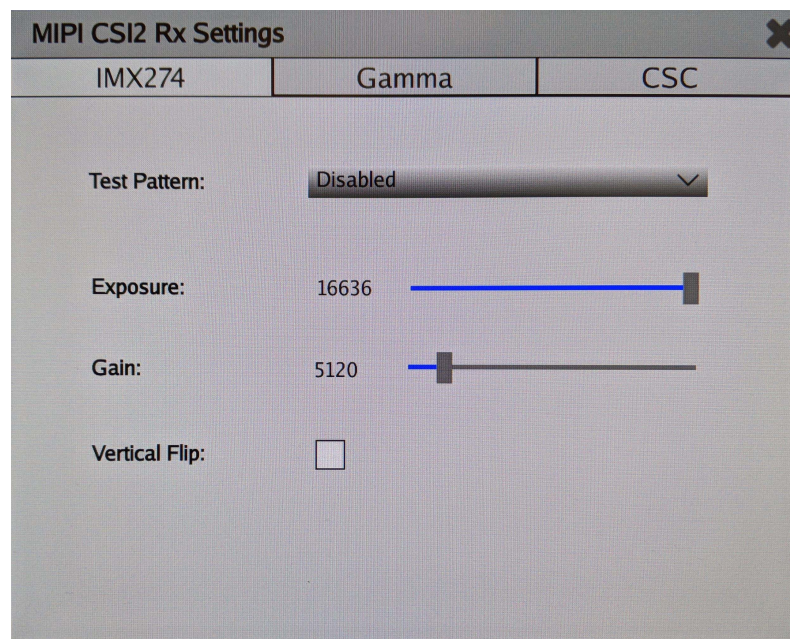


Figure 3-5: CSI Settings Panel

File Source Settings Panel

This panel (Figure 3-6) can be accessed from the video source settings icon. It provides the following control knobs:

- File Browser to select video file for playback

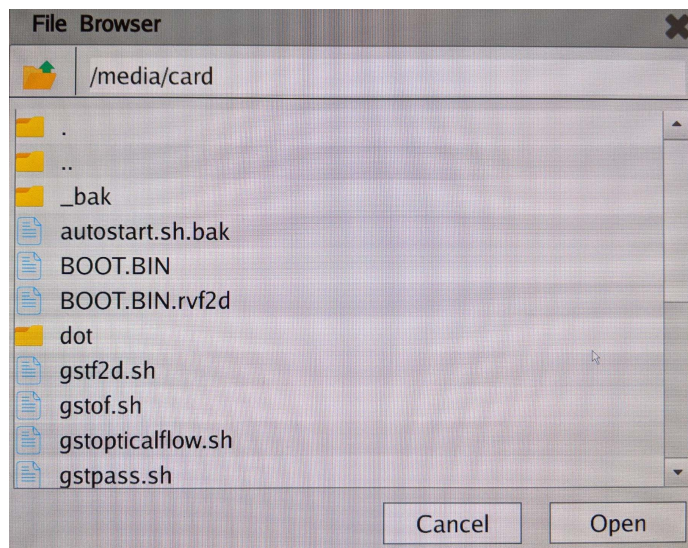


Figure 3-6: File Browser Panel

2D Filter Settings Panel

This panel (Figure 3-7) can be accessed from the filter mode settings icon. It provides the following control knobs:

- Filter Presets (blur, sharpen, emboss, sobel, custom, etc.)
- Coefficients

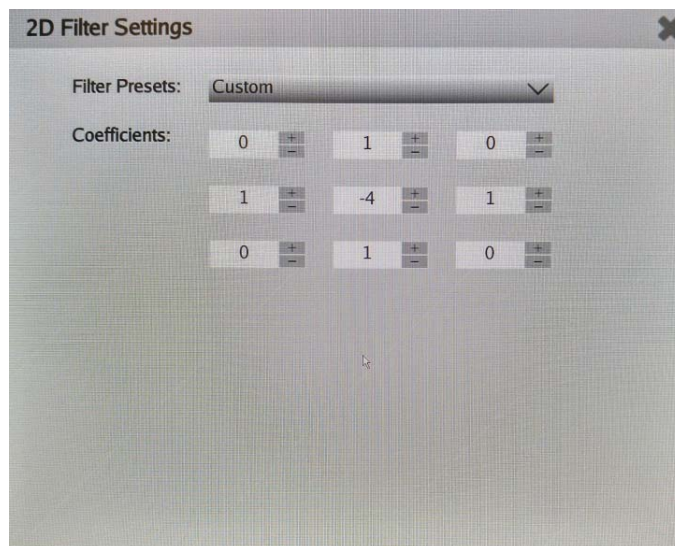


Figure 3-7: 2D Filter Settings Panel

Demo Mode Settings Panel

This panel (Figure 3-8) can be accessed from the demo mode settings icon. It provides the following control knobs:

- Time Interval
- Demo Sequence Count
- Per sequence drop down menus for video source, accelerator, and accelerator mode

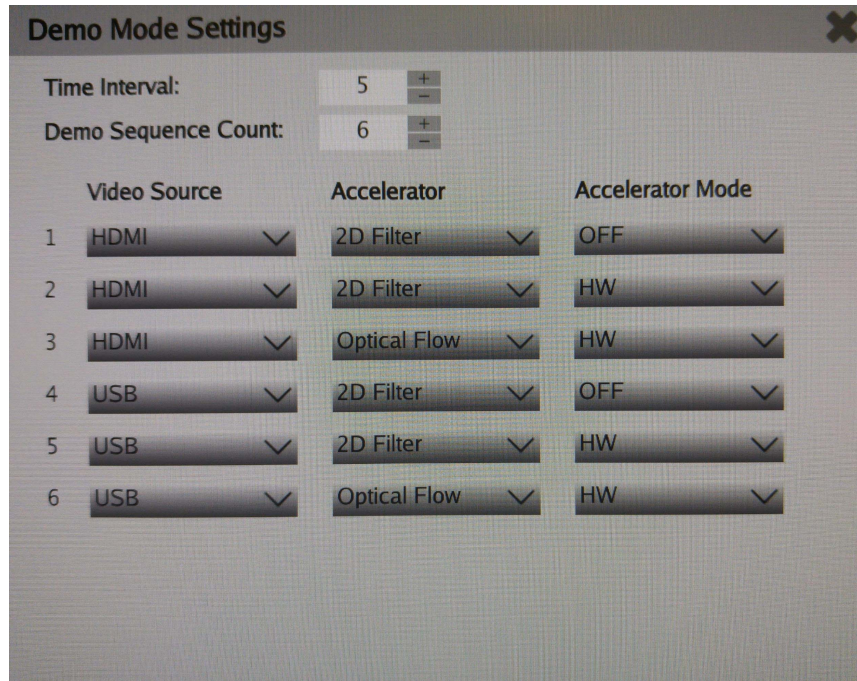


Figure 3-8: Demo Mode Settings Panel

Video Library

The `video_lib` library implements the following features:

- Display configuration
- Media pipeline configuration for video capture
- Video sub-device control

The `video_lib` library exports and imports the following interfaces as shown in [Figure 3-2, page 19](#):

- TPG video source controls (to `video_qt2`)
- CSI video source controls (to `video_qt2`)
- Media pipeline initialization (to `xlnxvideosrc`)
- Display initialization (to `xlnxvideosink`)
- Interfaces from middleware layers (media controller, DRM)

Display Configuration

The `video_lib` library uses the `libdrm` library to configure the CRTC based on the monitor's EDID information with the video resolution of the display. It also configures plane properties such as alpha value. The graphics plane is configured by the Qt EGLFS backend outside of this library. The video data flow and buffer pools are managed by `gst_lib` and the `kmssink` plugin, also outside of this library. The pixel format for each of the two planes is configured statically in the device-tree.

Media Pipeline Configuration

The `video_lib` library implements a media controller interface that allows you to configure the media pipeline and its sub-devices. It uses the `libmediactl` and `libv4l2subdev` libraries which provide the following functionality:

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

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. The video data flow and buffer pools are managed by `gst_lib` and the `v4l2src` plugin outside of this library.

GStreamer Video Library

GStreamer is a cross-platform open source multimedia framework that provides infrastructure to integrate multiple multimedia components and create pipelines/graphs. GStreamer graphs are made of two or more plugin elements which are delivered as shared libraries.

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

- Selection of a video-source GStreamer plugin
- Selection of a video-sink GStreamer plugin
- Selection of a video-processing GStreamer plugin
- Creation of a GStreamer graph based on above plugins plus capabilities
- Configuration of properties of above GStreamer plugins
- Control of a GStreamer pipeline/graph

The `gst_lib` library exports and imports the following interfaces as shown in [Figure 3-2, page 19](#):

- GStreamer pipeline control and configuration (to `video_lib`)
- GStreamer SDx accelerator plugin properties (to `video_qt2`)

GStreamer Plugins

The following three GStreamer plugin categories are used by the Xilinx Gst Library:

- Available video-source options:
 - `xlnxvideosrc` plugin: V4l2 sources e.g. USB webcam, Vivid, TPG, HDMI Rx, MIPI CSI-2 Rx
 - `multisrc` plugin: video file source for raw video files
- Available video-sink options:
 - `xlnxvideosink` plugin: KMS display sinks e.g. DP Tx, HDMI Tx
 - `fpsidsplaysink` plugin: frames-per-seconds information for above 2 sinks

- Available video-processing options:
 - `sdxfilter2d` plugin: 2D filter SDx accelerator
 - `sdxopticalflow` plugin: optical flow SDx accelerator

GStreamer Capabilities

The pads are the element's interface to the outside world. Data streams from one element's source pad to another element's sink pad. The specific type of media that the element can handle will be exposed by the pad's capabilities.

The following capabilities are used between video-source plugin and its peer plugin (either video-sink or video-processing). These capabilities (also called capsfilter) are specified by `gst_lib` while constructing a GStreamer graph, for example:

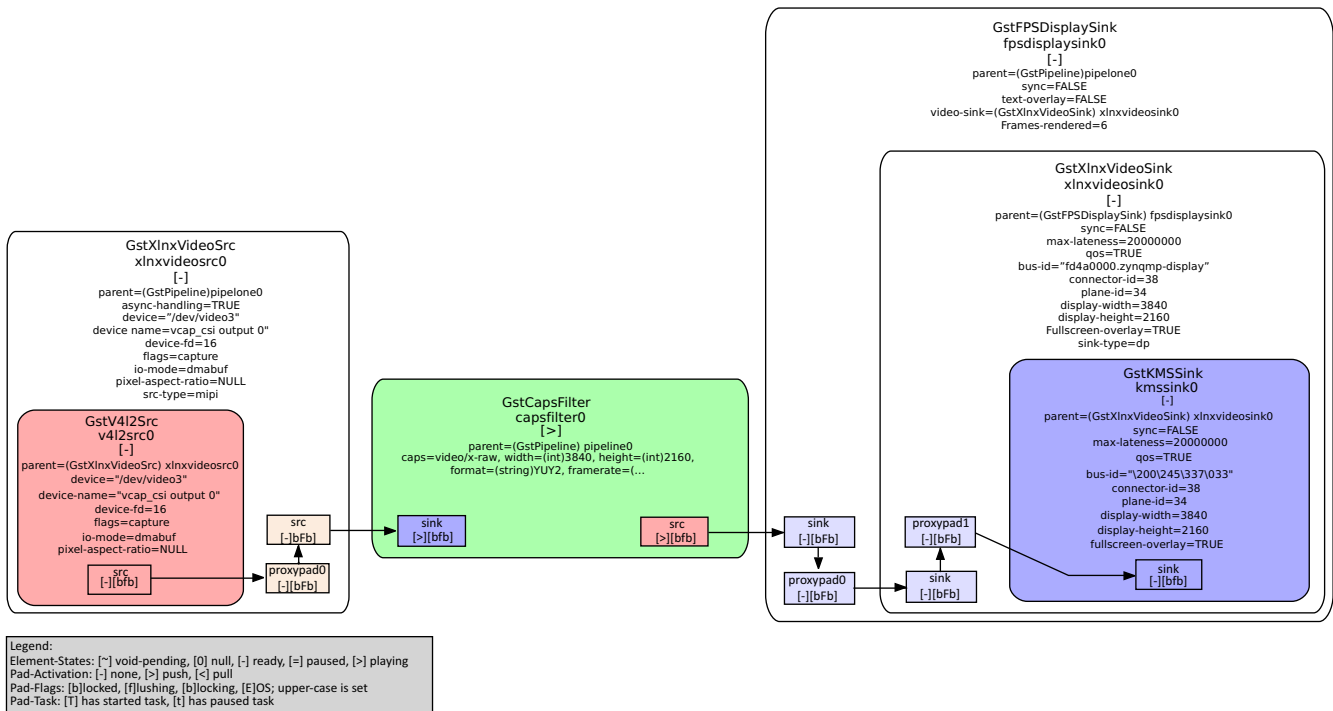
```
"video/x-raw, width=<width of videosrc>, height=<height of
videosrc>, format=YUY2, framerate=<fps/1>"
```

If `multisrc` is used as video-source plugin, the `videoparse` element is used instead of a capsfilter to parse the raw video file and transform it to frames:

```
"videoparse width=<width of filesrc> height=<height of filesrc>
format=yuy2 framerate=<fps/1>"
```

GStreamer Graphs

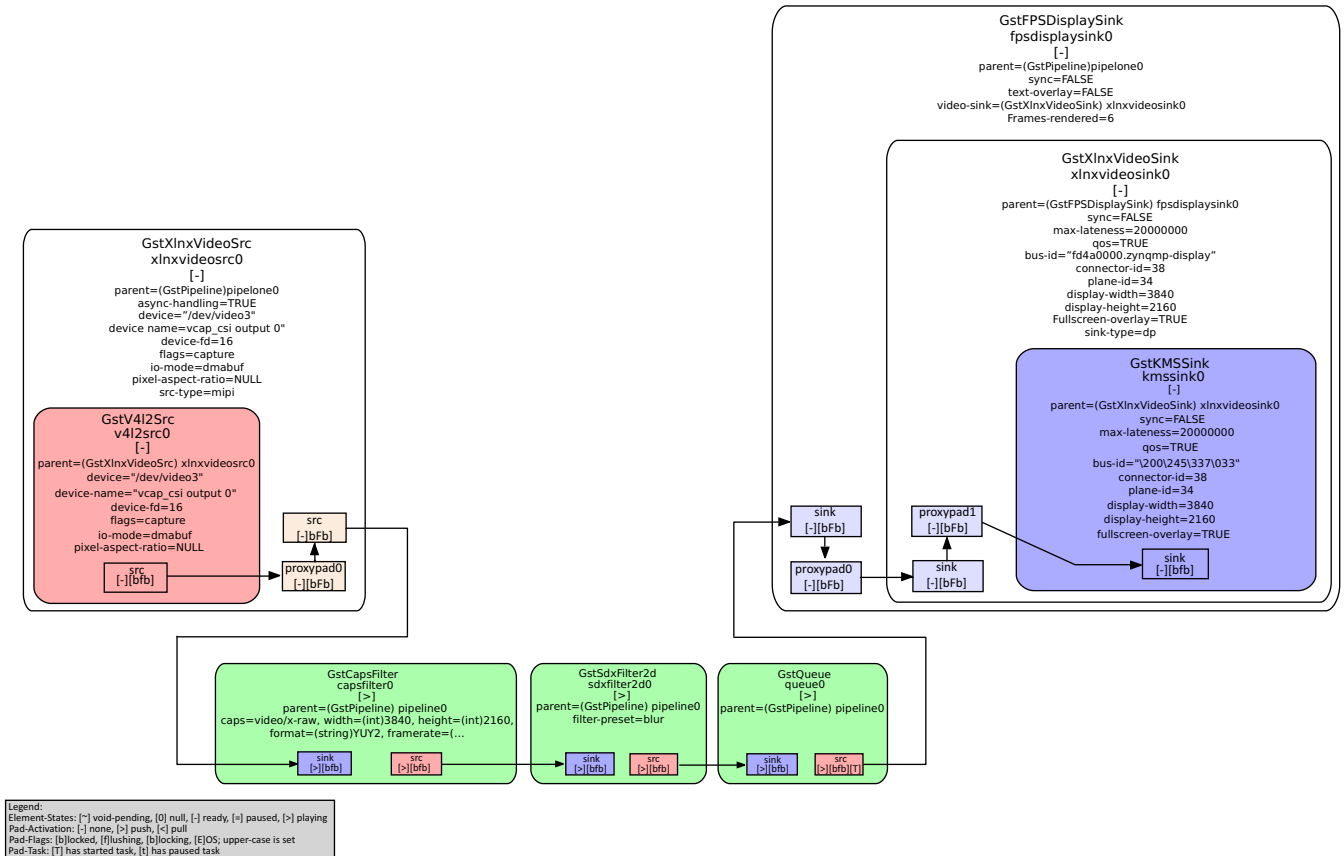
There are only two data modifying elements in the pass-through graph: one from video-source category and one from the video-sink category; video-processing elements are absent. The capsfilter is an element that is linked as part of the pipeline between the source and sink elements but does not modify the data as such. Figure 3-9 shows a pass-through graph consisting of a video source connected to a display sink.



X20874-111318

Figure 3-9: Passthrough Mode: xlnxvideosrc Connected to xlnxvideosink

In the accelerator GStreamer graph, a video-processing GStreamer element processes incoming video from a video source before sending it to the video sink. There are three plugins in the accelerator graph: one from the video-source category, one from the video-processing category, and one from the video-sink category. Figure 3-10 shows a live video source connected to an SDx accelerator connected to a display sink.



X20873-111318

Figure 3-10: Accelerator Mode: xlnxvideosrc Connected to sdxfilter Connected to xlnxvideosink

GStreamer Pipeline Control

The GStreamer framework is used to control the GStreamer graph. It provides the following functionality:

- Start/stop video stream inside a graph
- Get/set controls
- Buffer operations
- Get frames-per-second information

There are four states defined in the GStreamer graph: "NULL", "READY", "PAUSED", and "PLAYING". The `gst_lib` library sets the "PLAYING" state of a GStreamer graph to start the pipeline and the "NULL" state to stop the pipeline.

GStreamer abstracts buffer allocation and pooling. Custom allocators and buffer pools can be implemented to accommodate custom use-cases and constraints. The video source controls buffer allocation, but the sink can propose parameters in the negotiation phase.

The DMABUF framework is used to import and export buffers in a 0-copy fashion between pipeline elements, required for high-performance pipelines (see [Figure 3-11](#)). The `v4l2src` element has a property named `io-mode` which allows allocation and export of DMABUFs to its peer element. The `kmssink` element allows import as well as export of DMABUFs to/from its peer element. The `gstsdxbase` class allows only import of DMABUFs, which means it relies on DMABUFs being allocated by its peer elements connected to the source and sink pads. Note that DMABUFs are not necessarily physically contiguous depending on the underlying kernel device driver.

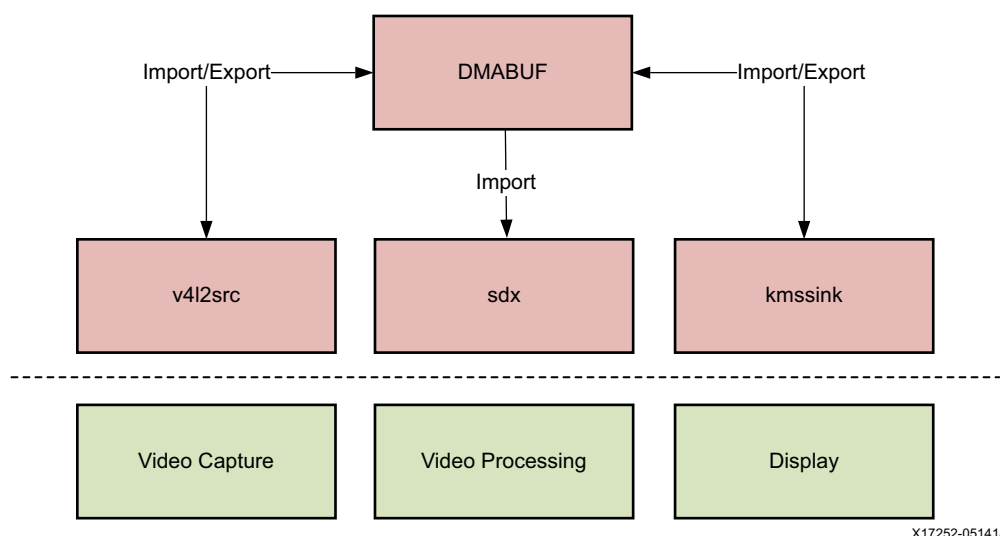


Figure 3-11: DMABUF Sharing Mechanism

The `gstsdxallocator` class implements a custom allocator for the `gstsdxbase` class using the `sds_alloc_non_coherent` and `sds_free` sdslib functions which allows sharing buffers in a 0-copy fashion between two `gstsdx` elements.

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.

Xilinx Video Source and Sink GStreamer Plugins

The Xilinx video source and sink GStreamer plugins are designed to simplify the usage of the capture and display devices in this design as they take care of any initialization and configuration that the user has to take care of otherwise. [Figure 3-12](#) shows the general architecture of the Xilinx video source and video sink GStreamer plugins. The gray-colored boxes are components developed by Xilinx whereas the white boxes are open-source components.

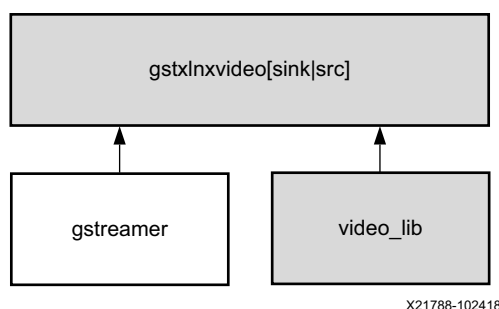


Figure 3-12: Gstreamer Xilinx Video Source and Sink Plugin Architecture

The Xilinx video source plugin `xlnxvideosrc` is a bin element that contains the `v4l2src` element which is a standard GStreamer plugin. In addition, it uses the Xilinx `video_lib` library to configure the media pipelines of the supported video sources in this design (see [Video Library, page 26](#) for details). The `src-type` property is used to select the video source; valid options are:

- `hdmi`
- `mipi`
- `tpg`
- `usbcam`
- `vivid`

All other properties are inherited from the `v4l2src` element.

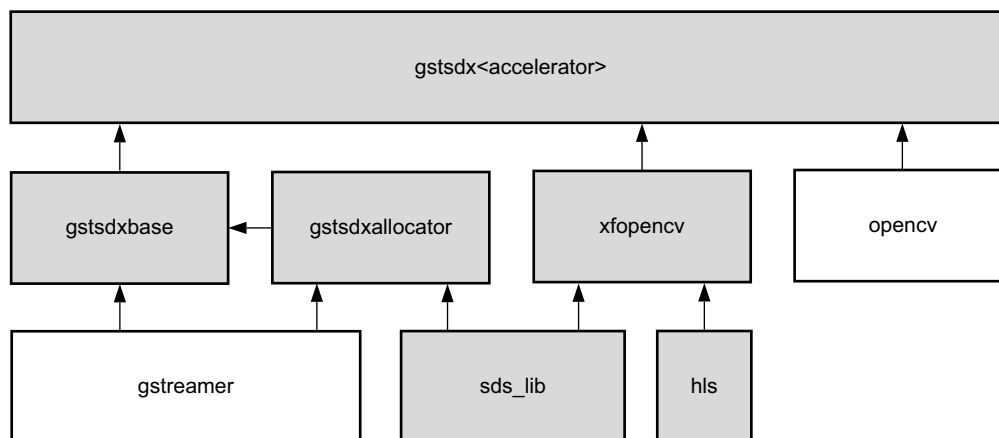
The Xilinx video sink plugin `xlnxvideosink` is a bin element that contains the `kmssink` element which is a standard GStreamer plugin. In addition, it links in the Xilinx `video_lib` library which is used to set QoS settings matching the selected display device. The `sink-type` property is used to select the video sink; valid options are:

- `dp`
- `hdmi`

All other properties are inherited from the `kmssink` element.

SDx GStreamer Plugins

The SDx GStreamer plugins are designed to implement memory-to-memory accelerators that can easily and seamlessly interface with other GStreamer elements like video sources and sinks. Figure 3-13 shows the general architecture of the SDx GStreamer plugins. The gray-colored boxes are components developed by Xilinx whereas the white boxes are open-source components.



X21789-102418

Figure 3-13: GStreamer SDx Plugin Architecture

An SDx accelerator element has one source and one sink pad; it can consume N temporal input frames from its source pad and produce one output frame on its sink pad. All SDx accelerator plugins inherit from the generic SDx base class which in turn inherits from the GStreamer video transform class.

The SDx base class provides common infrastructure that is shared across all accelerators. It also provides a generic `filter-mode` property which allows the user to switch between a hardware-accelerated version of the algorithm or a pure software implementation. Note that it is not mandatory for accelerator plugins to implement both modes. Accelerator plugins can implement additional accelerator-specific properties. The SDx allocator class wraps the low-level SDx memory allocation and dmabuf routines from the `sds_lib` library.

The Xilinx `xfopencv` libraries provide hardware-optimized implementations of a subset of the `OpenCV` libraries. They are implemented in C-code that is then synthesized to FPGA hardware using High Level Synthesis (HLS). The `sds_lib` and `hls` libraries are used for memory allocation as well as memory and hardware interface generation.

2D Filter GStreamer SDx Plugin

In this example, a 2D convolution filter is implemented in two different versions: (1) as pure software-implementation using the OpenCV library [Ref 6], and (2) as hardware-optimized implementation using the Xilinx xfOpenCV library [Ref 7].

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 5].

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 3-14).

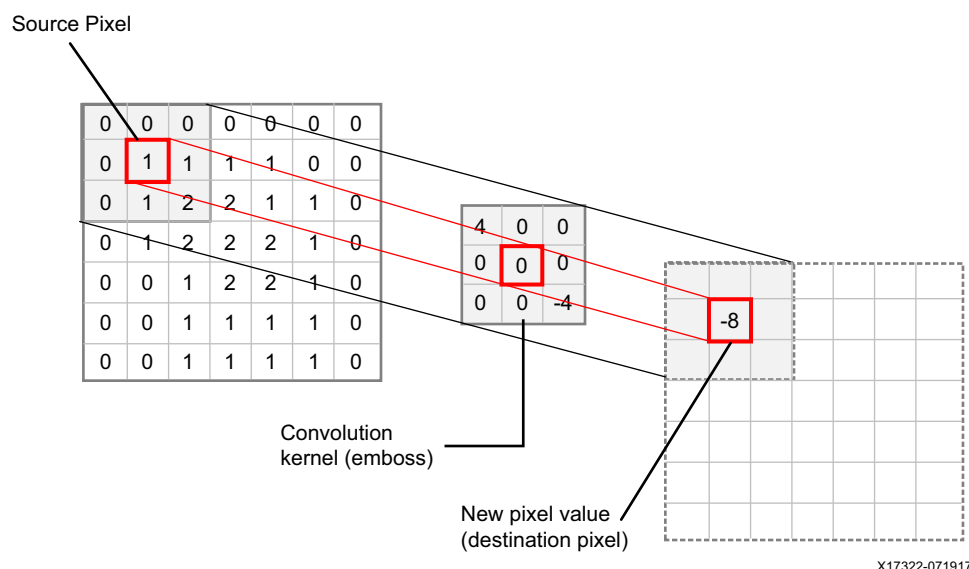


Figure 3-14: 2D Convolution with 3x3 Kernel

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 48). 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.

The library implements a transform function that takes an input frame and produces an output frame. It also exports an 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 presets* such as blur, sharpen, edge detection, etc.

The SDSoc example uses three hardware-accelerated functions to process the video:

- The first function, `read_f2d_input`, extracts the luma component from the input image to prepare it for the next, main processing step that operates on luma only. The main processing function, `filter2d_sds` uses the `xfOpenCV` function `filter2D`, as described in the *Xilinx OpenCV User Guide* (UG1233) [Ref 7], with a 3x3 window size, one pixel per clock and a maximum resolution of 3840x2160.
- As final step, the `write_f2d_output` function merges the unmodified UV component with the modified luma component from the main processing function.

Optical Flow GStreamer SDx Plugin

This example implements a dense optical flow algorithm. The algorithm processes two consecutive frames and generates a 2D vector field of motion vectors, where each vector represents the movement of points from first to second frame. For more details, refer to Dense Non Pyramidal LK Optical Flow in the *Xilinx OpenCV Library User Guide* (UG1233) [Ref 10]. The filter operates on a YUYV pixel format on the input and output side. The pixel format YUYV is a packed format with 16 bits per pixel (see [Video Buffer Formats](#), page 48). Each pixel can be divided into two 8-bit components (or channels): one for luma (Y), the other for chroma (U, V). The SDSoc example uses three hardware-accelerated functions to process the video.

- The first function, `read_optflow_input`, extracts the luma component from the input image to prepare it for the next, main processing step that operates on luma only.
- The main processing step uses the `xfOpenCV` function `xFDenseNonPyrLKOpticalFlow`, as described in UG1233 [Ref 7], with a 25x25 window size, two pixel per clock and a maximum resolution of 3840x2160. The optical flow algorithm computes the motion vectors from the luma components of two consecutive input images.
- As final step, the `write_optflow_output` function maps the motion vectors to individual components of the YUYV output image. In detail, motion in X-direction (*flowx*) is mapped to the U and motion in the Y-direction (*flowy*) is mapped to the V-component of the image. The magnitude of the motion;

$$\sqrt{(flowx^2 + flowy^2)}$$

is mapped to the Y component (the output shows moving pixels in different colors depending on the direction of their movement and at different intensity, depending on their speed of movement). [Figure 3-15](#) illustrates the UV color plane for $Y=0.5$.

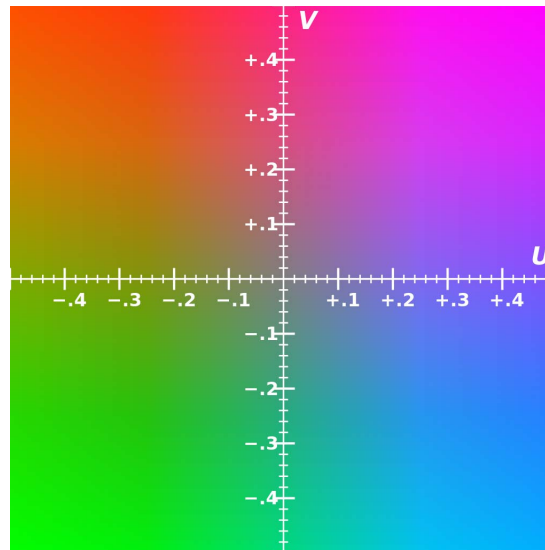


Figure 3-15: UV Color Plane for $Y=0.5$

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.

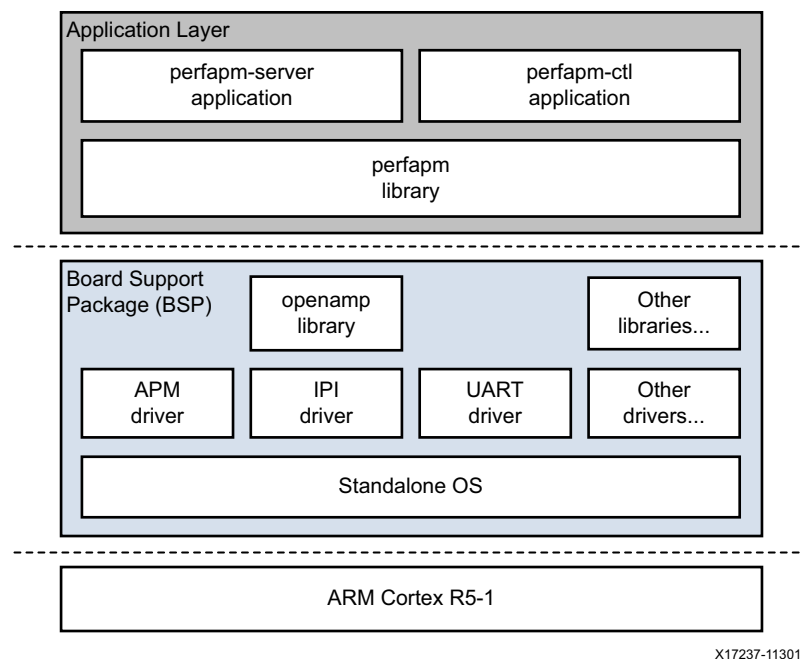
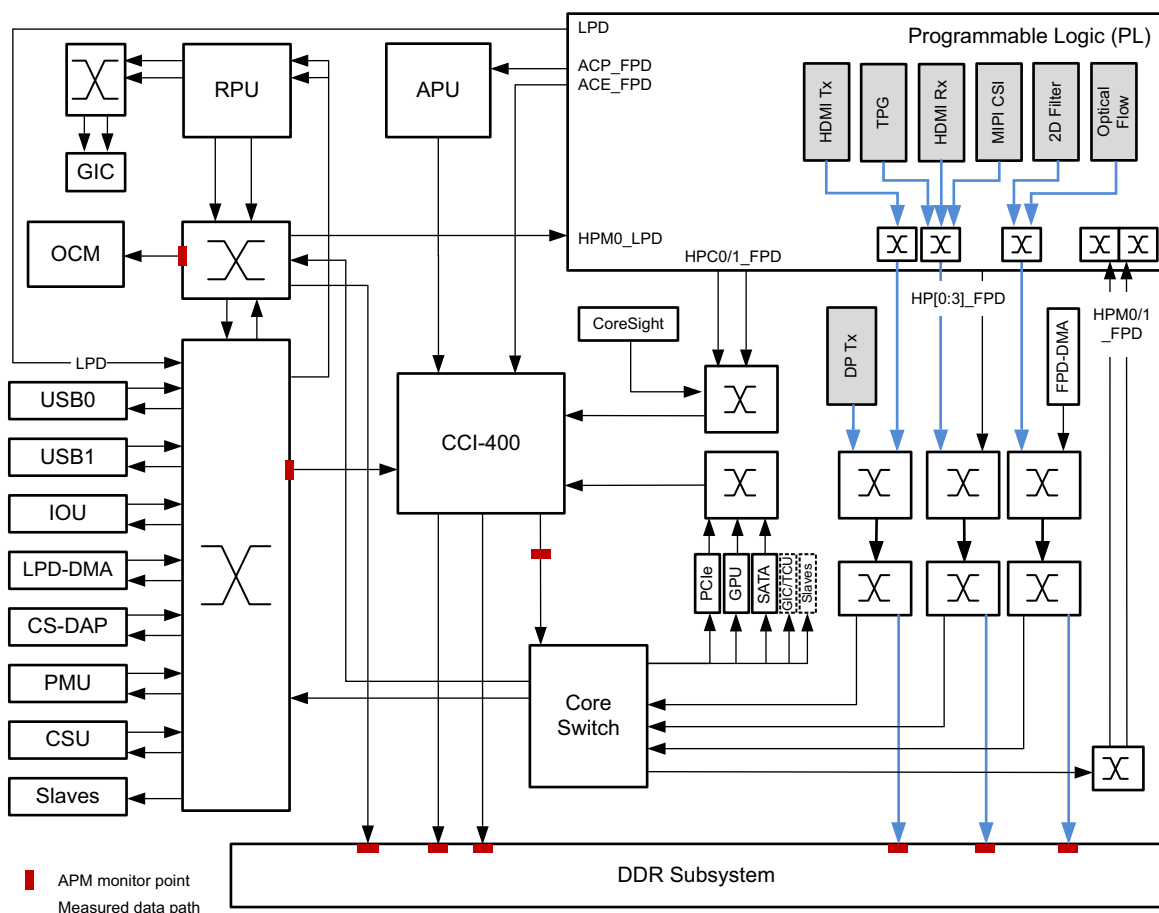


Figure 4-1: Bare-metal Software Stack Running on RPU-1

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



X15277-090817

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 32](#)) 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 plus an additional 32 KB for the descriptors themselves. See [Inter-process Communication, page 69](#) 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 messages to the UART.

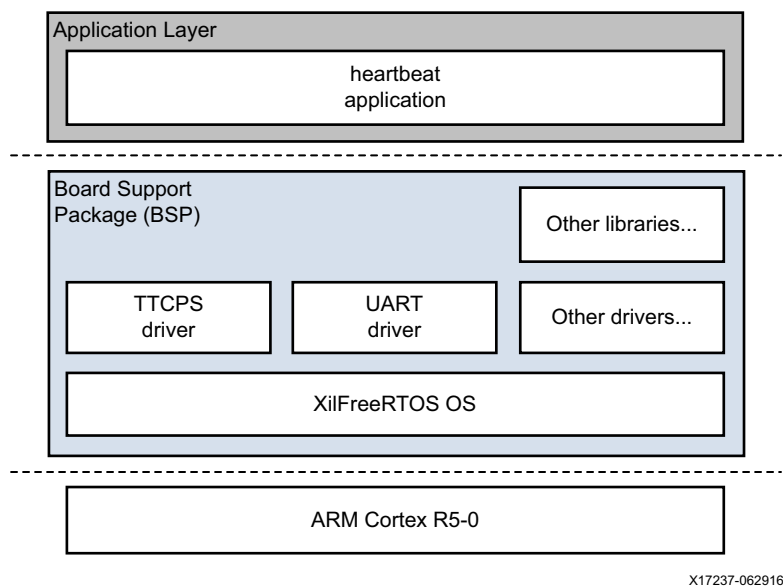


Figure 5-1: FreeRTOS Software Stack Running on RPU-0

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 *thread 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.

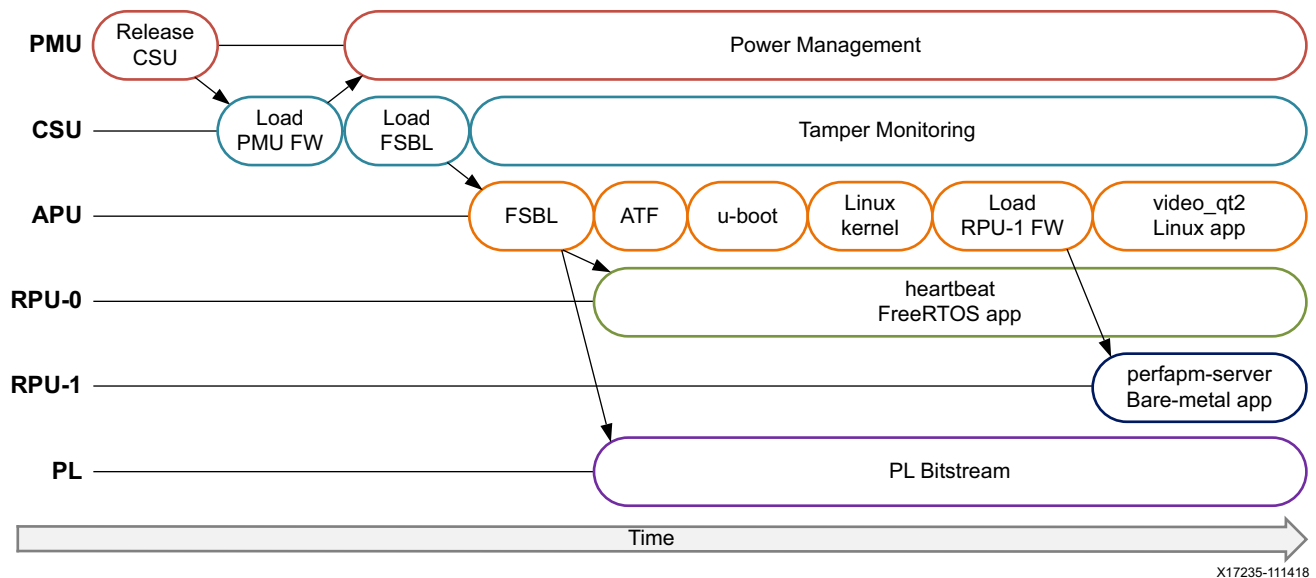


Figure 6-1: Boot Flow Diagram

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 DDR.
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, RPU-0 is brought out of reset and starts 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 9], and chapter 8 in *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 11].

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 have an AXI-lite interface with dedicated clock are connected to HPM0
- IPs that share a clock between AXI-lite and streaming or memory-mapped interfaces (typically-HLS based IPs) are connected to HPM1.

For more information on system addresses, see chapter 8 in *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 11].

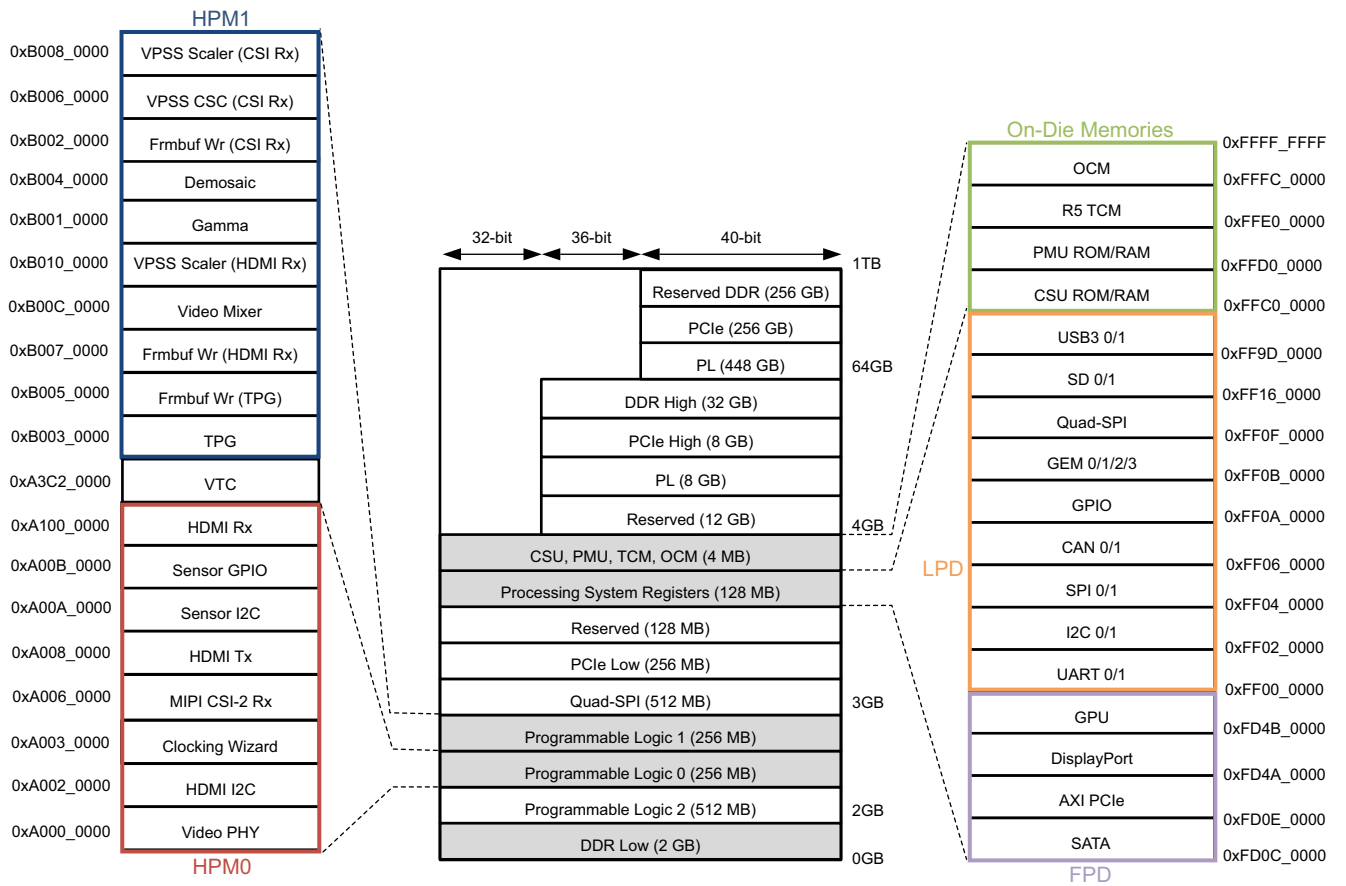


Figure 6-2: Global Address Map

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 Framebuffer Write and Video Mixer instances in the PL are limited to a 32-bit address space so they can only access the DDR Low address region 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 is loaded into DDR memory and the `perfapm-server` firmware running on RPU-1 is loaded into a combination of TCM and 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	TCM + 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	<ul style="list-style-type: none"> • APU • DPDMA (PS) • Framebuffer Write, Video Mixer (PL) • GPU • SDSoC-generated data movers (PL) 	Non-coherent, contiguous
Vring buffers (OpenAMP)	<ul style="list-style-type: none"> • 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 11].

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 or HDMI 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 blending* 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% (or 0) means the pixel is fully transparent (invisible); an alpha value of 100% (or 255) means that the pixel is 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.

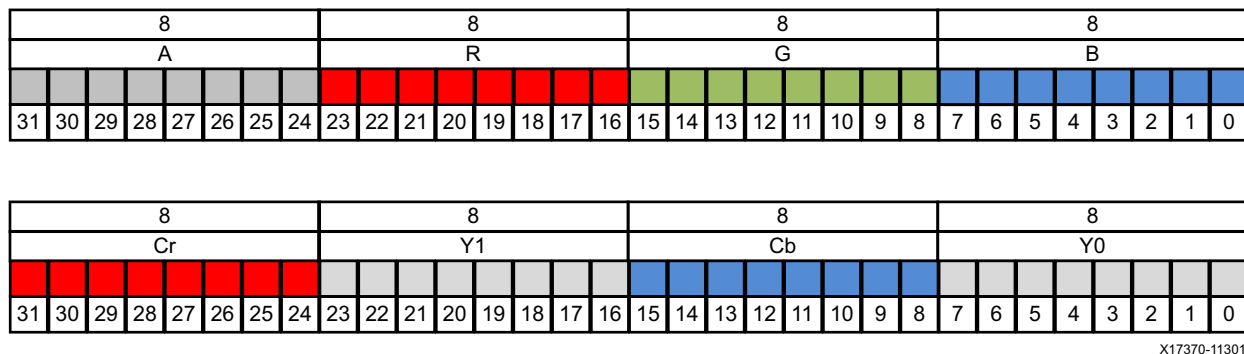


Figure 6-3: AR24 (Top) and YUYV (Bottom) Pixel Formats in Little Endian Notation

Aside from the pixel format, a video buffer is further described by a number of other parameters (see Figure 6-4):

- 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 1920 x 1080 pixels therefore has a stride of $4 \times 1920 = 7,680$ 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.
- The PS DP Tx display pipeline does not allow for setting the X and Y offsets unlike the PL HDMI Tx display pipeline where the offsets are programmable. In this design, the X and Y offsets are always set to 0 for both display pipelines.

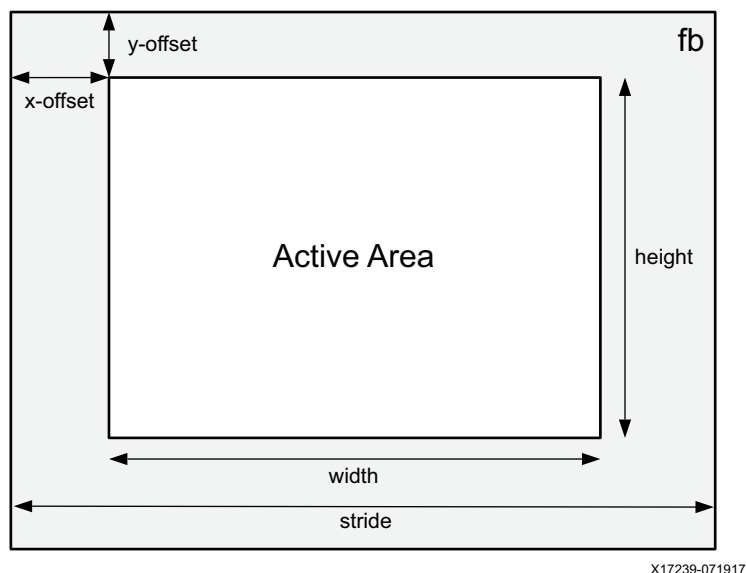


Figure 6-4: Video Buffer Parameters

Performance Metrics

Memory Throughput

The following calculation represents the maximum performance of this design; similar calculations can be made for lower performance points. Note that 2160p60 is only possible via HDMI Tx while DP Tx is limited to 2160p30.

The memory throughput for a video stream of resolution 3840 by 2160 pixels at 60 frames per second (2160p60) using a YUYV pixel format, is calculated as:

$$3840 \times 2160 \times 16 \text{ bits} \times 60/\text{s} \approx 8 \text{ Gb/s}$$

Or using a AB24 pixel format, as:

$$3840 \times 2160 \times 32 \text{ bits} \times 60/\text{s} \approx 16 \text{ Gb/s}$$

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 (optical flow): 2 YUYV input streams, 1 YUYV output stream

- display:
 - video layer: 1 YUYV stream
 - graphics layer: 1 AR24 stream

The total memory throughput is calculated as:

$$5 \times 8 \text{ Gb/s} + 1 \times 16 \text{ Gb/s} = 56 \text{ Gb/s}$$

This 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 56 Gb/s is about 41% which leaves some headroom for more demanding applications:

$$56 \text{ Gb/s} \div 136 \text{ Gb/s} \approx 0.41$$

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 [Performance Monitor Client Library, page 32](#) 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 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 40](#)). 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, the HP port and its command priority (if applicable) for each of the video/display pipelines used in the design, and 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
DP Tx Display	-	0xB/-	Video	3
HDMI Tx Display	HP0	0xF/-	Best Effort	3
TPG/HDMI Rx/CSI Rx Capture	HP1	- /0x0	Best Effort	4
2D Filter/Optical Flow Accelerator	HP3	0x0/0x0	Best Effort	5

Different ports into the DDR controller are used for video capture (TPG, HDMI Rx, or CSI Rx), video acceleration (2D Filter or Optical Flow), and DP 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 DP Tx 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 50](#)). DP Tx and HDMI Tx are connected to the same DDR controller port (DDRC 3). As both interfaces require different QoS settings, the values are programmed dynamically by the application, depending on which output is currently active.

Although the video capture and display pipelines implemented in the PL are also video timing accurate, the corresponding IPs currently lack support for writing or fetching video data according to the video class traffic type definition. The HP1 port used for video capture and the HP0 port used for display are thus configured for best effort data traffic instead of video.

The accelerator pipelines are 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 78](#) for video timing description). The traffic class is therefore *best effort*.

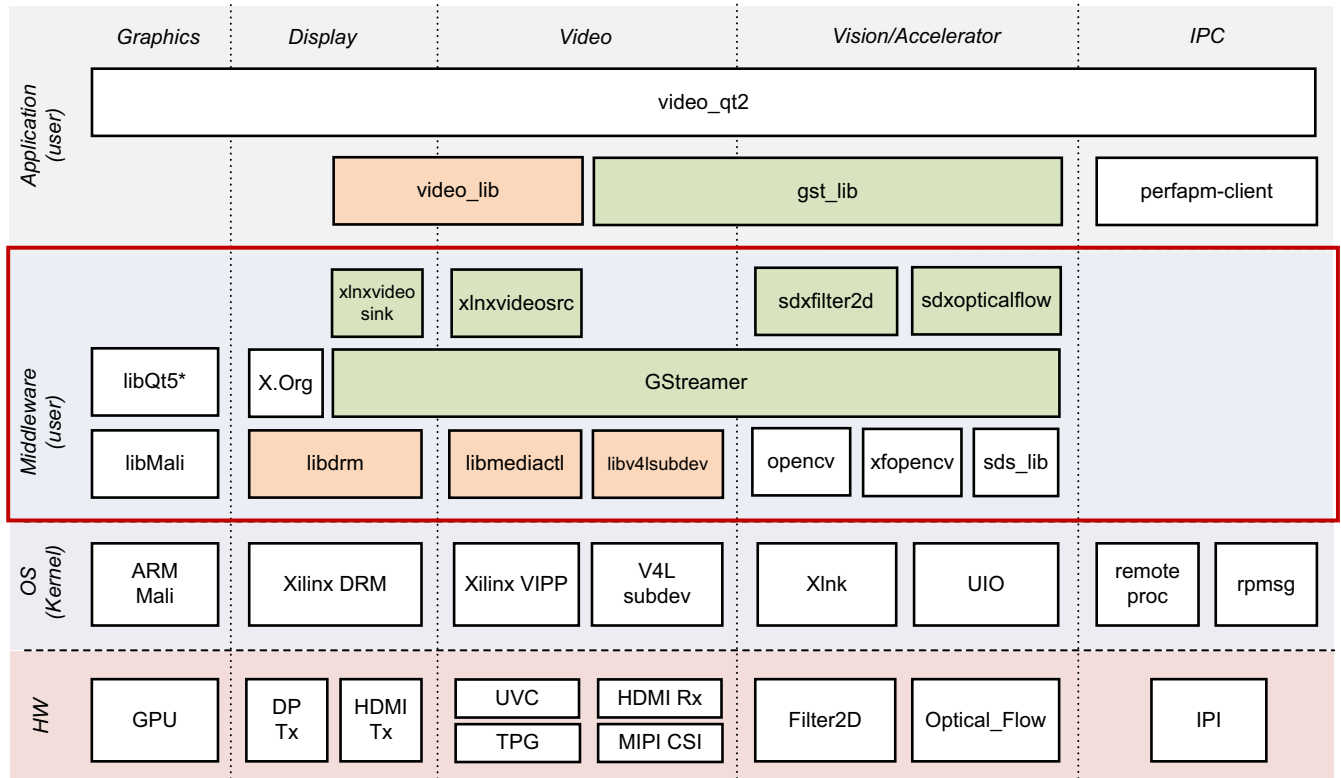
The Base TRD uses static command QoS. The corresponding read and/or write data priority values for display, capture, and processing have been identified through experimentation. For more information on QoS, refer to [10], chapters 13, 15, and 33 in *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 11\]](#).

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)



X17275-102418

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

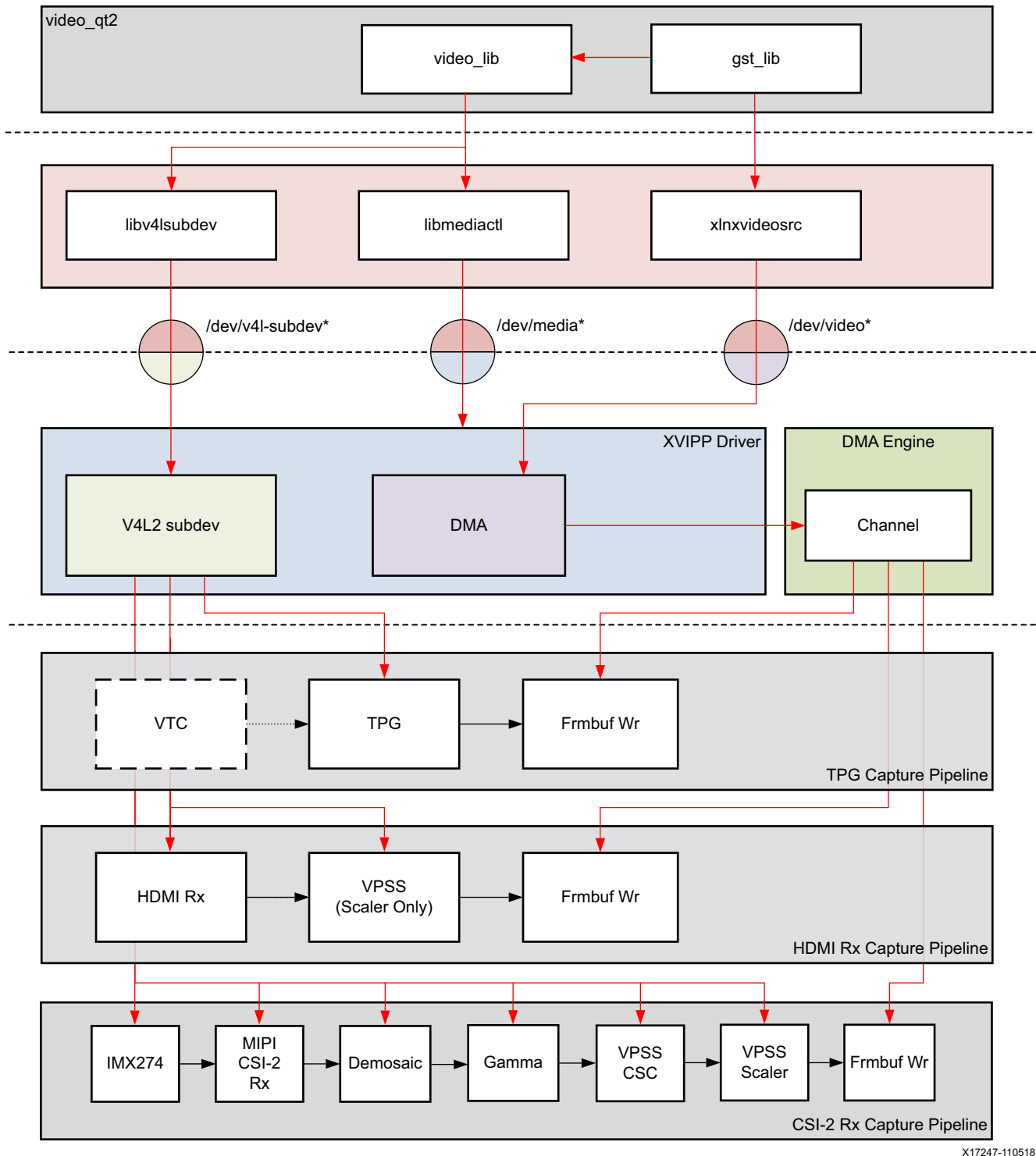
Video

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, HDMI Rx, and CSI Rx capture pipelines. 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*`



X17247-110518

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.

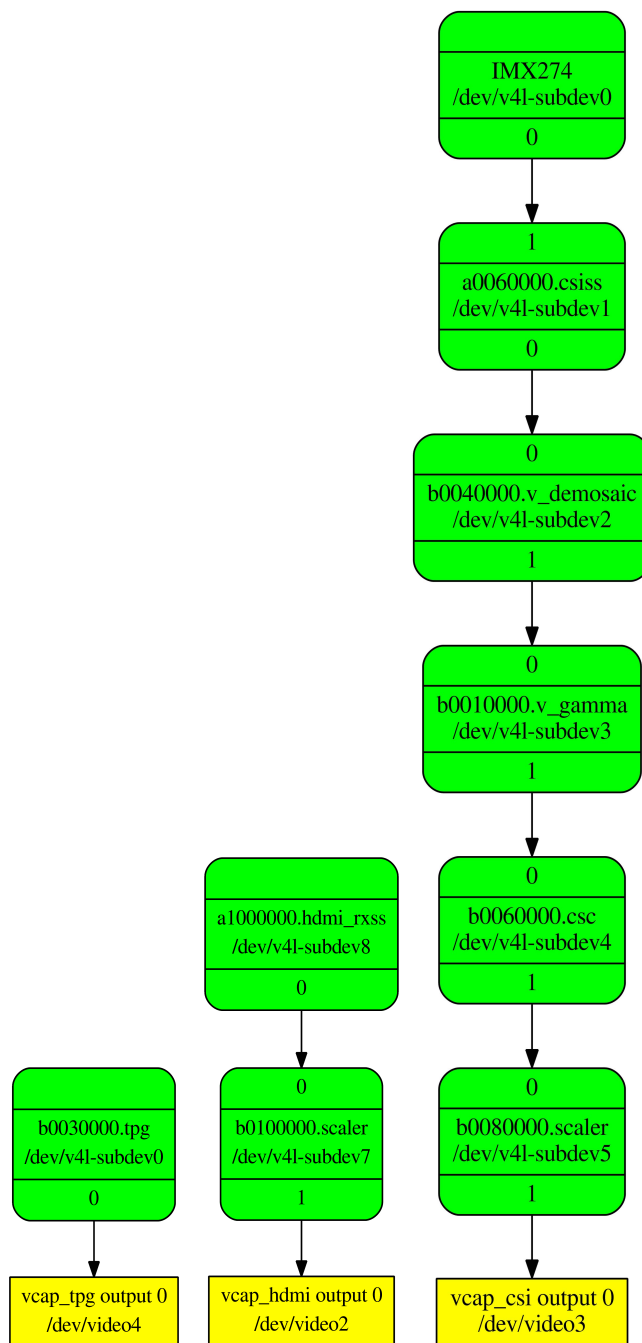
An entity is a basic media hardware building block. It can correspond to a large variety of blocks such as physical hardware devices (e.g. image sensors), logical hardware devices (e.g., soft IP cores inside the PL), DMA channels or physical connectors. Physical or logical devices are modeled as sub-device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



X17327-012318

Figure 7-3: Video Capture Media Pipelines: TPG Left, HDMI Rx Center, CSI Rx Right

V4L2 Framework

The V4L2 framework is responsible for capturing video frames at the video device node, typically representing a DMA channel, and making those video frames available 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 `VIDIOC_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 Frame Buffer Write IP. 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 [GStreamer Pipeline Control, page 30](#)). In DMABUF, buffers are allocated by a driver on behalf of an application. 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)	<ul style="list-style-type: none"> Configures video pipeline and register media, video and sub-device nodes. Configures all entities in the pipeline and validate links. Configures and controls DMA engines (Xilinx Video Framebuffer Write). Starts/stops video stream.
Xilinx Video Test Pattern Generator	<ul style="list-style-type: none"> Sets media bus format and resolution 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.
Xilinx HDMI Rx Subsystem	<ul style="list-style-type: none"> Query digital video (DV) timings on output pad. Sets media bus format and resolution on output pad.
Xilinx Video Processing Subsystem (Scaler Only configuration) 2X	<ul style="list-style-type: none"> Sets media bus format and resolution on input pad. Sets media bus format and resolution on output pad (both can be different from the input configuration which enables color space conversion and scaling).
MIPI CSI-2 Rx	<ul style="list-style-type: none"> Sets media bus format and resolution on input pad. Sets media bus format and resolution on output pad.
Xilinx Video Demosaic	<ul style="list-style-type: none"> Sets media bus format and resolution on input pad. Sets media bus format and resolution on output pad.
Xilinx Video Gamma LUT	<ul style="list-style-type: none"> Sets media bus format and resolution on input pad. Sets media bus format and resolution on output pad. Sets Gamma control parameters: red/green/blue gamma control.
VPSS (Color Space Converter (CSC) configuration)	<ul style="list-style-type: none"> Sets media bus format and resolution on input pad. Sets media bus format and resolution on output pad. Sets CSC control parameters: brightness, contrast, red/green/blue gain.
IMX274 Image Sensor	<ul style="list-style-type: none"> Sets media bus format and resolution on output pad. Sets sensor control parameters: exposure, gain, test pattern, vertical flip.

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.

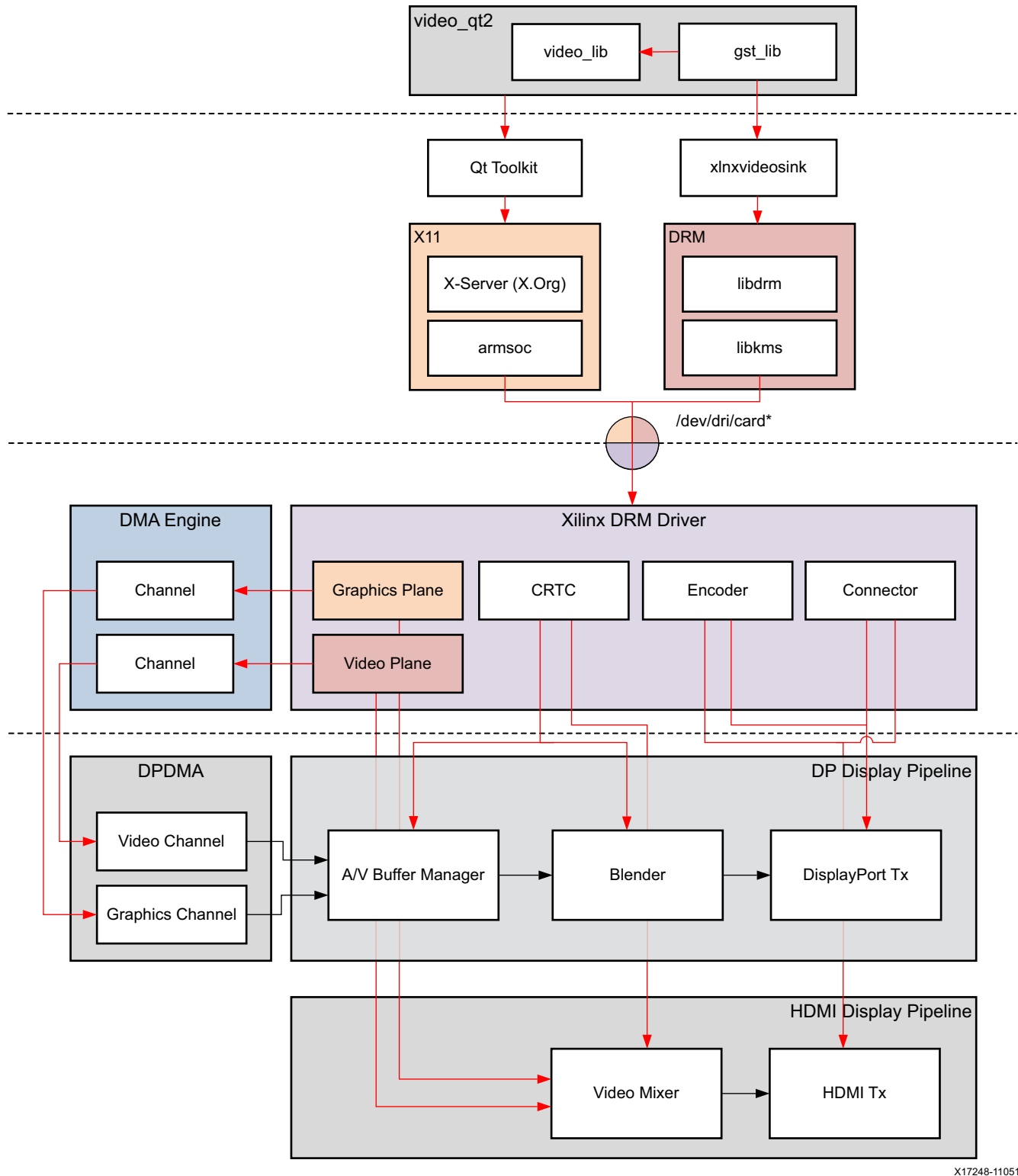
Dumb Buffer Objects

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 DP Tx display pipeline as well as the HDMI Tx display pipeline.



X17248-110518

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 overlaid and/or alpha-blended with a second plane inside the blender. The DP Tx 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 at run-time and a variety of formats are supported. The default pixel formats for each plane 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 overlaid 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 DP Tx display pipeline, unlike the HDMI Tx 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. The primary plane on the video mixer is configurable and set to the top-most plane to match the DP Tx pipeline.

As planes are modeled inside KMS, the physical hardware device that reads the data from memory is typically a DMA whose driver is implemented using the `dmaengine` Linux framework. The DPDMA is a 6-channel DMA engine that supports a (up to) 3-channel video stream, a 1-channel graphics stream and two channels for audio (not used in this design). The video mixer uses built-in AXI master interfaces to fetch video frames from memory.

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 (SIU) which serializes the data using the gigabit transceivers (PS GTRs) before it goes out via the physical DP connector to the display. The PL display pipeline uses a HDMI transmitter which sends the encoded video data to the Video PHY. The Video PHY serializes the data using the GTH transceivers in the PL before it goes out via the HDMI Tx connector.

Connector

The connector models the physical interface to the display. Both DisplayPort and HDMI protocols use 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 12]) stored inside the monitor. This data can then be used to correctly set the CRTC mode. The DisplayPort and HDMI also support 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 per display pipeline 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 toolkit. In this design, the open-source display server `X.Org` is used. It implements the X11 protocol and interfaces with the following components:

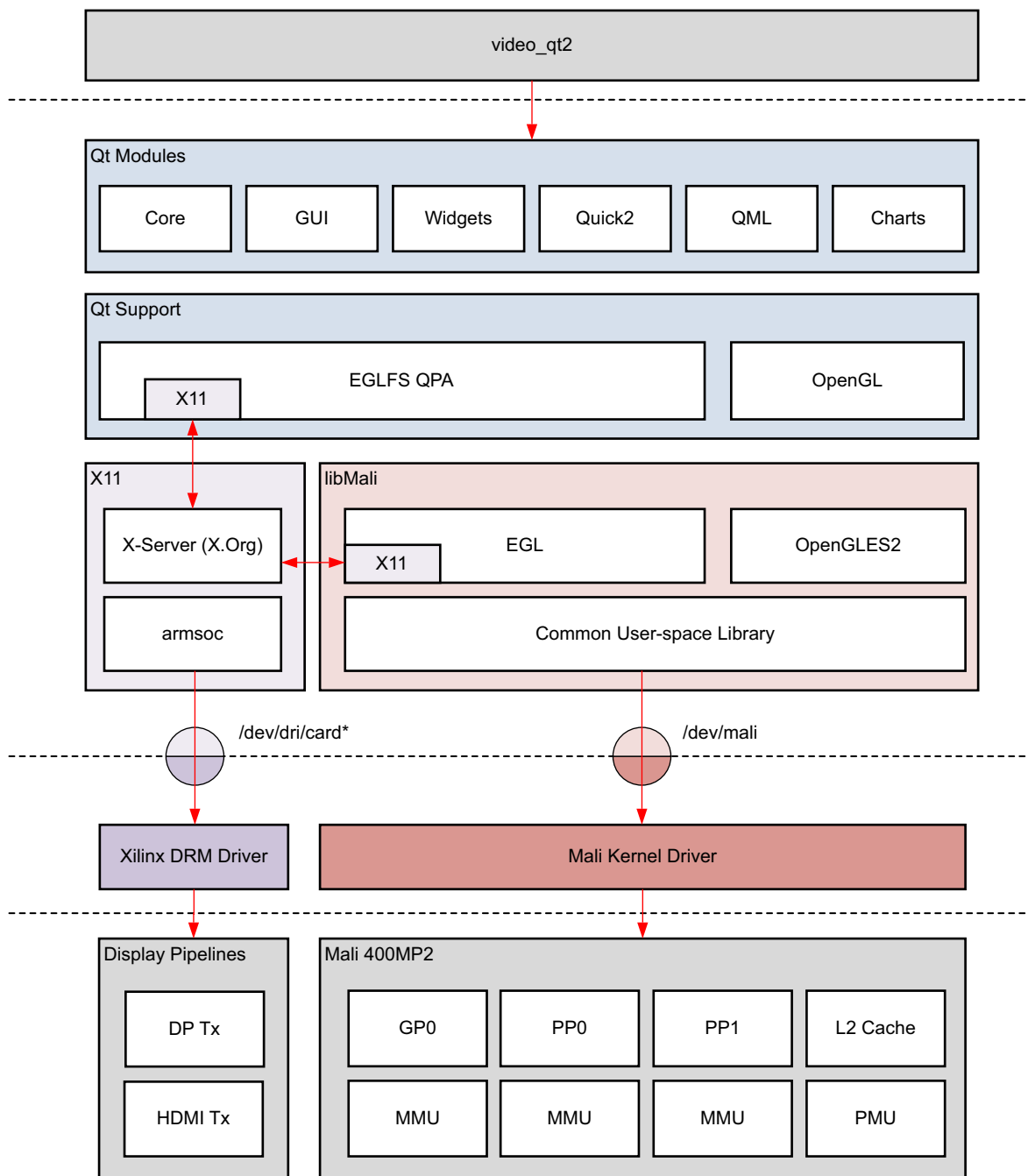
- `video_qt2` application through Qt toolkit with `eglfs_x11` platform abstraction (top)
- see [EGLFS QPA, page 68](#)
- Mali user-space library with X11 backend (right) - see [Mali Driver, page 69](#)
- DP or HDMI display controller through the `armsoc` driver and DRI device nodes (bottom)

The DRM driver enables multi-master mode so the DRI device nodes 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. The `armsoc` driver can target either display pipeline by selecting the respective DRI node in the `xorg.conf` file.

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.



X17249-060217

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.
Charts	A set of easy to use chart components using the Qt Graphics View Framework.

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 67. 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 13].

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 67](#).

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 X11 backend interfacing with the X-Server.

The `libMali` user-space library implements the OpenGL ES 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 Fast OpenCV (`xfOpenCV`) libraries are FPGA hardware-optimized computer vision libraries that support a 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 equivalent hardware-optimized `xfOpenCV` function for hardware acceleration (see [Performance Monitor Client Library, page 32](#)). Similarly, the optical flow algorithm (see [Optical Flow GStreamer SDx Plugin, page 36](#)) is implemented using the corresponding hardware optimized `xfOpenCV` function.

Inter-process Communication

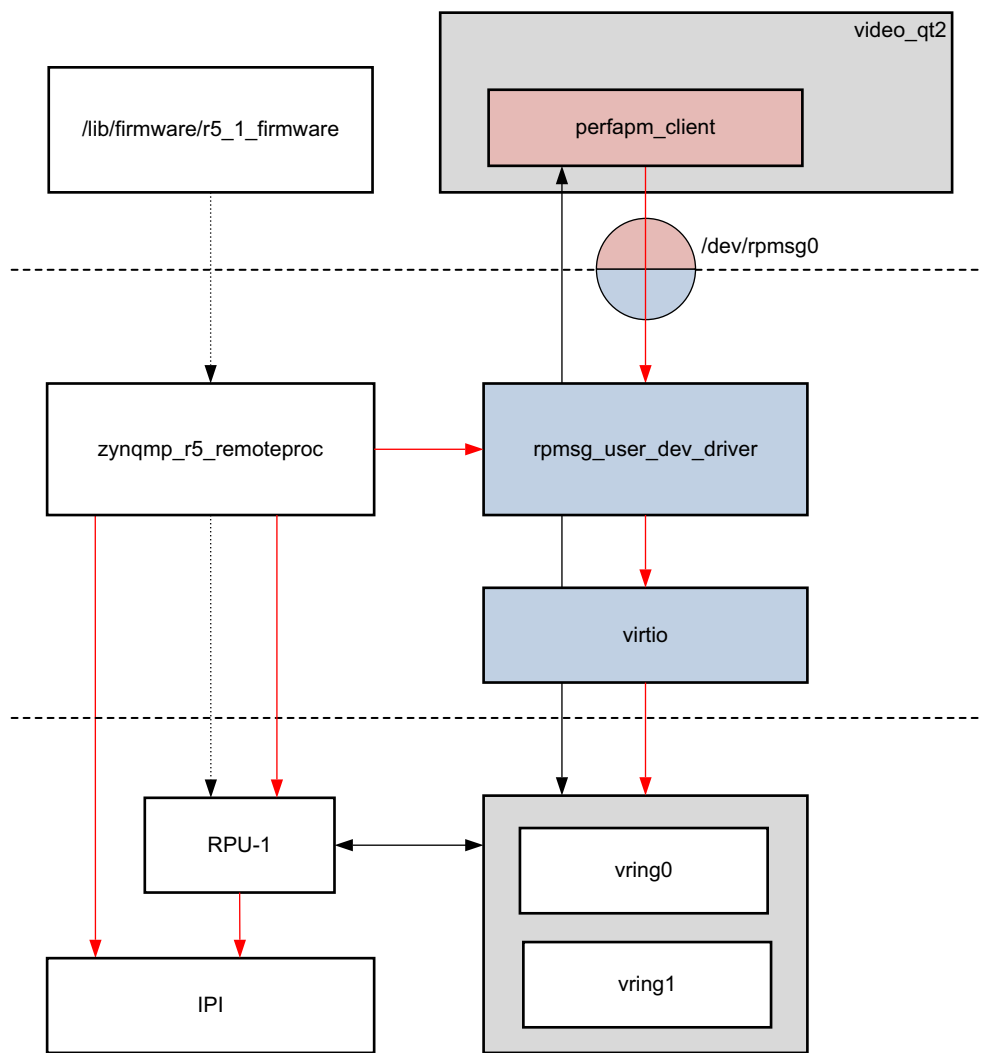
Xilinx open asymmetric multi-processing (OpenAMP) is a framework providing the software components needed to enable the development of software applications for asymmetric multi-processing (AMP) systems. The Base TRD uses the following configuration:

- 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.



X17250-071917

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 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. It then starts the remote processor.

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:

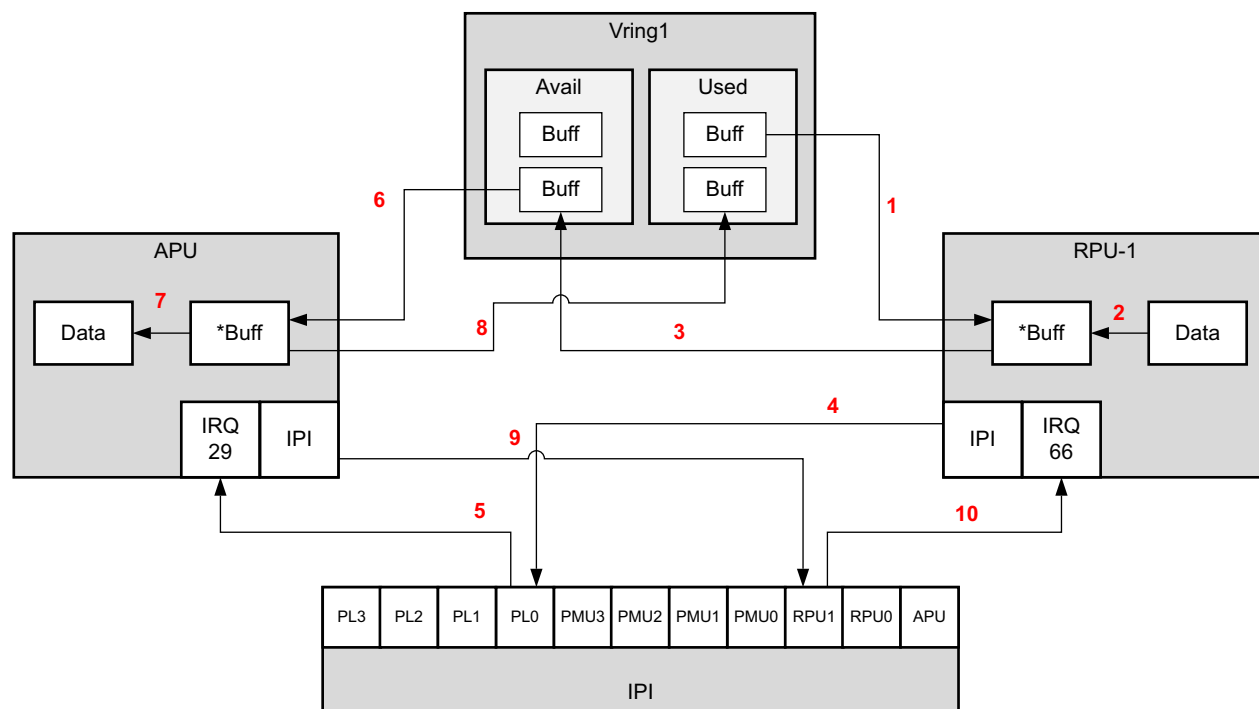
1. The master application sends an application-specific shutdown message to the remote application.
2. The remote application cleans up its resources and sends an acknowledgment to the master.
3. The remote frees the `remoteproc` resources on its side.
4. The master shuts down the remote and frees the `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 32](#)). 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 41](#)).

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.



X17254-061516

Figure 7-7: APU/RPU-1 Communication via RPMs

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 and the LI-IMX274MIPI-FMC daughter card.

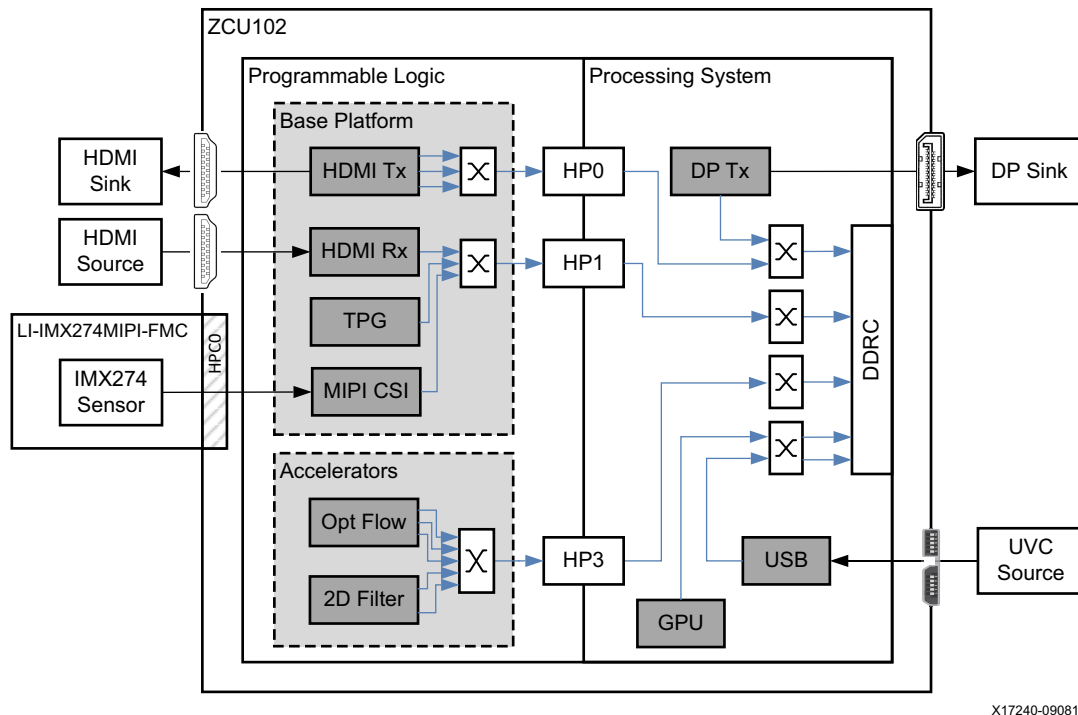


Figure 8-1: Hardware Platform and Generated Accelerator/Data Motion Network

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

- USB universal video class (UVC) capture pipeline (PS)
- Test pattern generator (TPG) capture pipeline (PL)
- HDMI Rx capture pipeline (PL)
- MIPI CSI-2 Rx capture pipeline (FMC + PL)

- 2D Filter processing pipeline (PL)
- Optical Flow processing pipeline (PL)
- HDMI Tx display pipeline (PL)
- DP Tx display pipeline (PS)

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

- Hardware base platform, which mainly consist of the I/O interfaces such as DP Tx, HDMI Tx, HDMI Rx, MIPI CSI-2 Rx, USB-UVC (this part of the design is fixed).
- Hardware accelerators (Filter 2D and Optical Flow) 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 video pipelines are described in more detail in the following sections.

Video Pipelines

Three types of video pipelines are differentiated in this design:

- A capture pipeline receives frames from an external source or produces video frames internally. The captured video frames are written into memory.
- A memory-to-memory (M2M) pipeline reads video frames from memory, does certain processing, and then writes the processed frames back into memory.
- An 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.

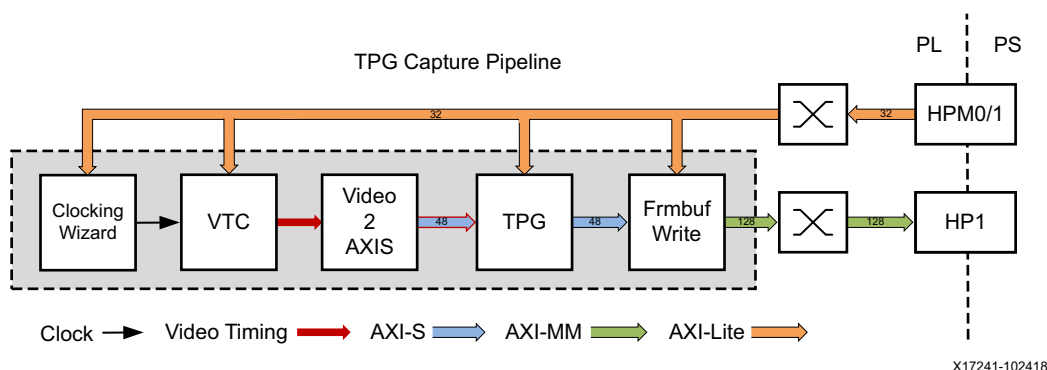
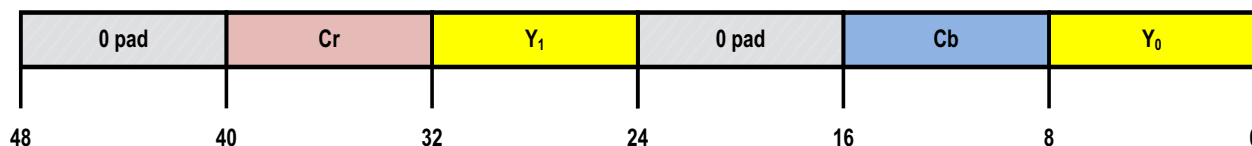


Figure 8-2: TPG Video Capture Pipeline

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

- The Clocking Wizard is an IP core wrapper on top of the mixed-mode clock manager (MMCM) primitive. It provides an AXI-Lite interface for dynamically reconfiguring the output clock frequency. The output clock is connected to the VTC and provides the pixel clock matching the requested video resolution. For more information refer to the *Clocking Wizard v6.0 LogiCORE IP Product Guide* (PG065) [Ref 22].
- 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. For more information refer to the *Video Timing Controller v6.1 LogiCORE IP Product Guide* (PG016) [Ref 23].
- The Video 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. A GPIO is used to reset the core between resolution changes. For more information refer to the *Video Test Pattern Generator v7.0 LogiCORE IP Product Guide* (PG103) [Ref 24].
- The Video Frame Buffer Write IP provides high-bandwidth direct memory access between memory and AXI4-Stream video type target peripherals, which support the AXI4-Stream Video protocol. In this pipeline, the IP takes AXI4-Stream input data from the TPG and converts it to AXI4-MM format. 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. A GPIO is used to reset the core between resolution changes. For more information refer to the *Video Frame Buffer Read v1.0 and Video Frame Buffer Write v1.0 LogiCORE IP Product Guide* (PG278) [Ref 25].

The pipeline can be configured for different video resolutions and frame rates at run-time. In this design, you can choose between 720p60, 1080p60, 2160p30, and 2160p60. The color format is configurable but in this design we only use YUV 4:2:2, 8 bits per component. The bus is configured for 2 ppc (pixel per clock) which means the AXI interfaces are double wide and transport two pixels worth of data per clock cycle. This results in a bus width of 48 bits where the upper 8 bits of each 24-bit word are padded (Figure 8-3).



X17253-060217

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 14] for more details on video timing parameters.

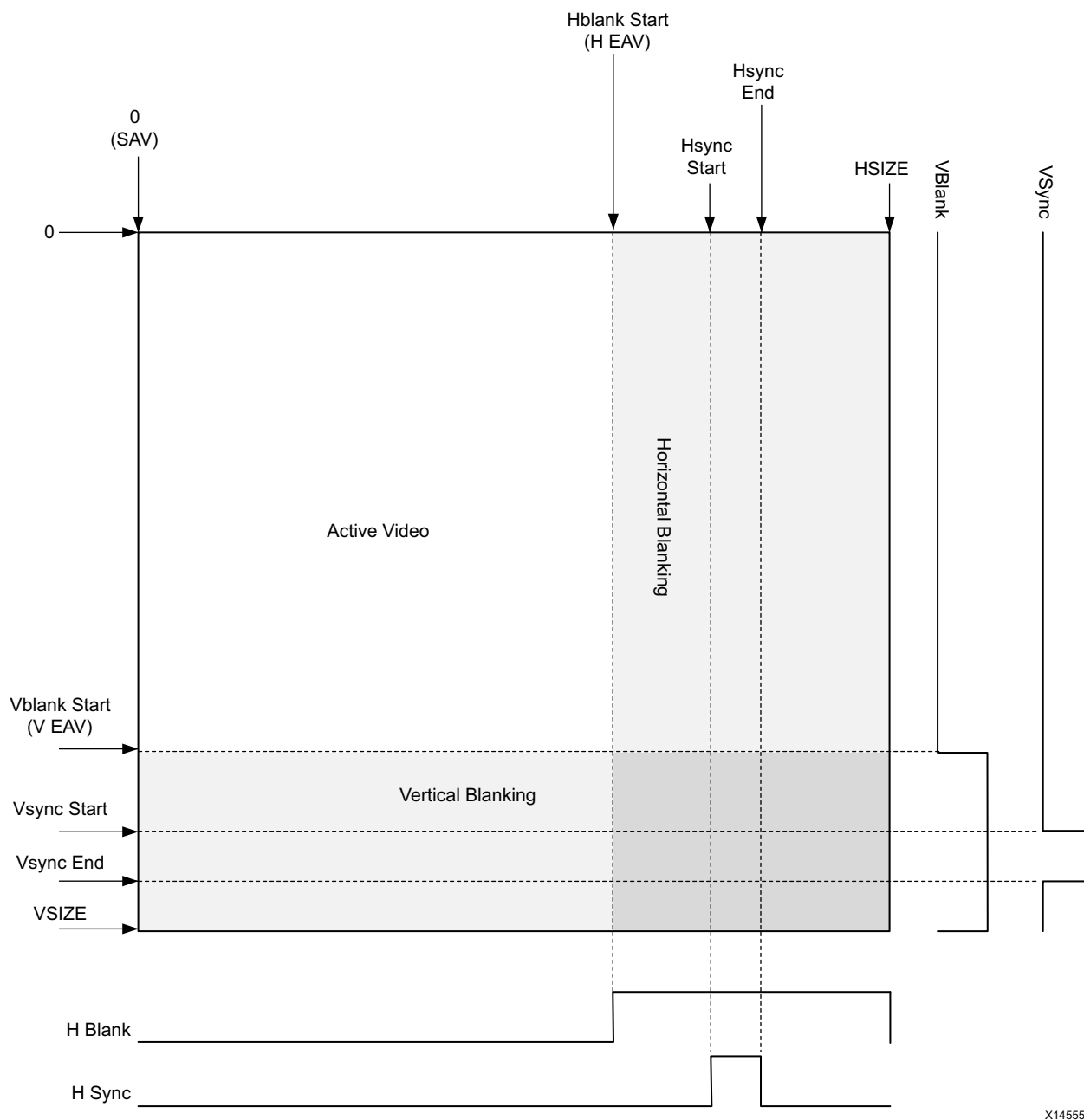


Figure 8-4: Active and Blanking Intervals of a Video Frame

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 pixel clock is generated by the Clocking Wizard and the VTC drives the horizontal and vertical sync and blanking signals, as well as active video (data enable) based on this scheme.

HDMI Rx Capture Pipeline

The HDMI receiver capture pipeline is shown in Figure 8-5.

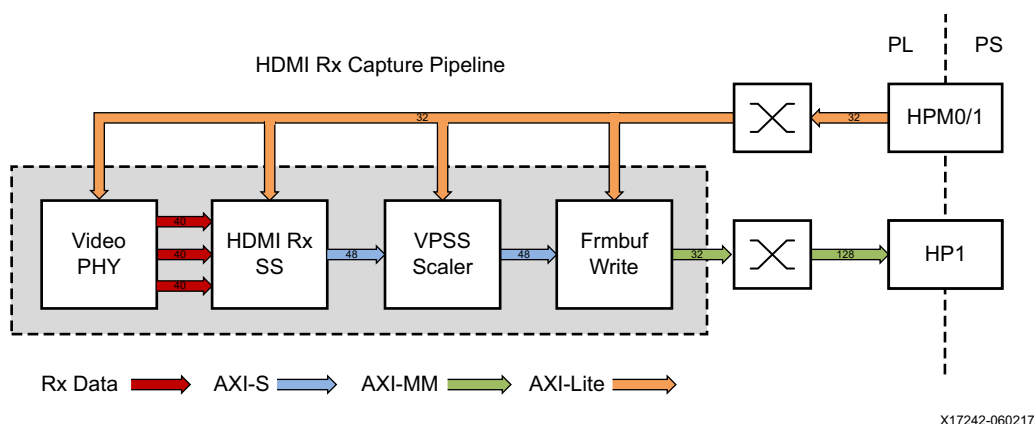


Figure 8-5: HDMI Video Capture Pipeline

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

- The Video PHY Controller (VPHY) enables plug-and-play connectivity with Video Transmit or Receive subsystems. The interface between the media access control (MAC) and physical (PHY) layers are standardized to enable ease of use in accessing shared gigabit-transceiver (GT) resources. The data recovery unit (DRU) is used to support lower line rates for the HDMI protocol. An AXI4-Lite register interface is provided to enable dynamic accesses of transceiver controls/status. For more information refer to the *Video PHY Controller v2.0 Product Guide* (PG230) [Ref 26].
- The HDMI Receiver Subsystem (HDMI Rx) interfaces with PHY layers and provides HDMI decoding functionality. The subsystem is a hierarchical IP that bundles a collection of HDMI RX-related IP sub-cores and outputs them as a single IP. The subsystem receives the captured TMDS data from the video PHY layer. It then extracts the video stream from the HDMI stream and in this design converts it to an AXI4-Stream output interface. For more information refer to the *HDMI 1.4/2.0 Receiver Subsystem v2.0 Product Guide* (PG236) [Ref 27].
- The Video Processing Subsystem (VPSS) is a collection of video processing IP subcores. In this design, the VPSS uses the *Scaler only* configuration which provides scaling, color space conversion, and chroma resampling functionality. The VPSS takes AXI4-Stream input data from the HDMI Rx Subsystem and depending on the input format and resolution, converts and scales it to the desired output format and resolution again using AXI4-Stream. A GPIO is used to reset the subsystem between resolution changes.

For more information refer to the *Video Processing Subsystem v2.0 Product Guide* (PG231) [Ref 28].

- The Video Frame Buffer Write IP uses the same configuration as the one in the TPG capture pipeline. It takes AXI4-Stream input data from the VPSS and converts it to AXI4-MM format. 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. A GPIO is used to reset the IP between resolution changes.

Similar to the TPG pipeline, the HDMI Rx, VPSS Scaler, and Frame Buffer Write IPs are configured to transport 2ppc, enabling up to 2160p60 performance. Although the color format and depth at the HDMI Rx are determined by the HDMI source, the VPSS always converts the format to YUV 4:2:2, 8 bpc which is then written to memory by the Frame Buffer Write IP as packed 16-bit YUYV.

MIPI CSI-2 Rx Capture Pipeline

The MIPI CSI-2 receiver capture pipeline is shown in Figure 8-6.

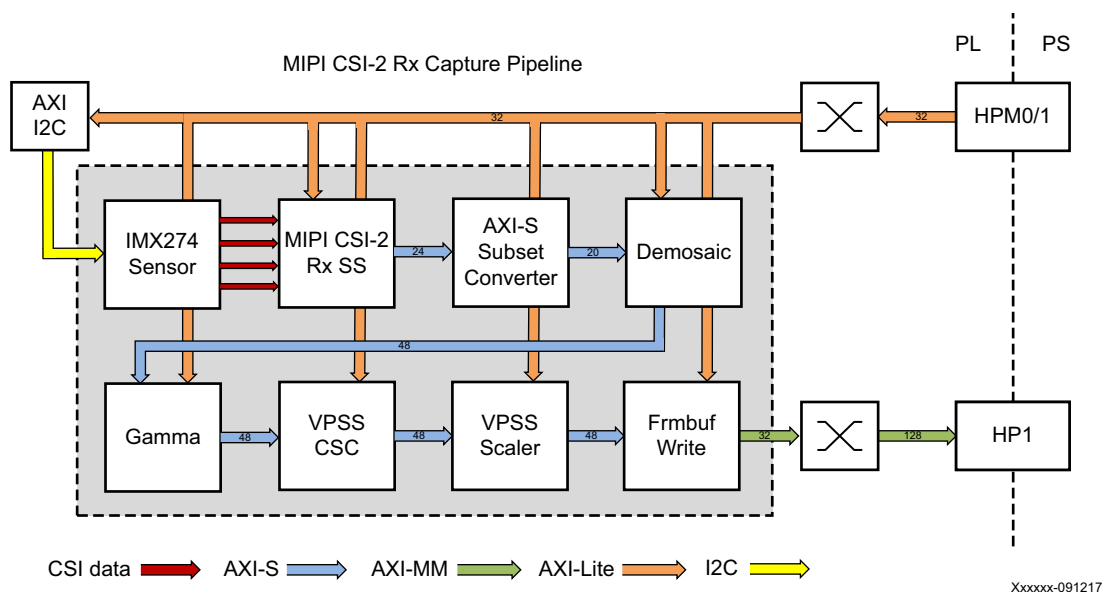


Figure 8-6: CSI Video Capture Pipeline

This pipeline consists of eight components, of which six are controlled by the APU via an AXI-Lite based register interface; one is controlled by the APU via an I2C register interface, and one is configured statically:

- The Sony IMX274 is a 1/2.5 inch CMOS digital image sensor with an active imaging pixel array of 3864H x2196V. The image sensor is controlled via an I2C interface using an AXI I2C controller in the PL. It is mounted on a FMC daughter card and has a MIPI output interface that is connected to the MIPI CSI-2 RX subsystem inside the PL. For more information refer to the LI-IMX274MIPI-FMC_datasheet [Ref 4].

- The MIPI CSI-2 Receiver Subsystem (CSI Rx) includes a MIPI D-PHY core that connects four data lanes and one clock lane to the sensor on the FMC card. It implements a CSI-2 receive interface according to the MIPI CSI-2 standard v1.1. The subsystem captures images from the IMX274 sensor in RAW10 format and outputs AXI4-Stream video data. For more information refer to the *MIPI CSI-2 Receiver Subsystem v2.2 Product Guide* (PG232) [Ref 31].
- The AXI subset converter is a statically configured IP core that converts the raw 10-bit (RAW10) AXI4-Stream input data to raw 8-bit (RAW8) AXI4-Stream output data by truncating the two least significant bits (LSB) of each data word.
- The Demosaic IP core reconstructs sub-sampled color data for images captured by a Bayer color filter array image sensor. The color filter array overlaid on the silicon substrate enables CMOS image sensors to measure local light intensities that correspond to different wavelengths. However, the sensor measures the intensity of only one principal color at any location (pixel). The Demosaic IP receives the RAW8 AXI4-Stream input data and interpolates the missing color components for every pixel to generate a 24-bit, 8bpc RGB output image transported via AXI4-Stream. A GPIO is used to reset the IP between resolution changes.
- The Gamma LUT IP core is implemented using a look-up table (LUT) structure that is programmed to implement a gamma correction curve transform on the input image data. Programmable number of gamma tables enable having separate gamma tables for all color channels, in this case red, green, and blue. The Gamma IP takes AXI4-Stream input data and produces AXI4-Stream output data, both in 24-bit RGB format. A GPIO is used to reset the IP between resolution changes.
- The Video Processing Subsystem (VPSS) is a collection of video processing IP subcores. This instance is uses the *Color Space Converter (CSC)* configuration to perform color correction tasks including contrast, brightness, and red/green/blue gain control. The CSC takes AXI4-Stream input data and produces AXI4-Stream output data, both in 24-bit RGB format. A GPIO is used to reset the subsystem between resolution changes. For more information refer to the *Video Processing Subsystem v2.0 Product Guide* (PG231) [Ref 28].
- The Video Processing Subsystem (VPSS) is a collection of video processing IP subcores. This instance uses the *Scaler only* configuration which provides scaling, color space conversion, and chroma resampling functionality. The VPSS takes AXI4-Stream input data in 24-bit RGB format and converts it to a 16-bit, 8bpc YUV 4:2:2 output format using AXI4-Stream. A GPIO is used to reset the subsystem between resolution changes.
- The Video Frame Buffer Write IP uses the same configuration as the one in the TPG and HDMI Rx capture pipelines. It takes YUV 4:2:2 sub-sampled AXI4-Stream input data and converts it to AXI4-MM format which is written to memory as 16-bit packed YUYV. The AXI-MM interface is connected to the HP1 high performance PS/PL port via an AXI interconnect. For each video frame transfer, an interrupt is generated. A GPIO is used to reset the IP between resolution changes.

Similar to the TPG and HDMI Rx capture pipelines, all the IPs in this pipeline are configured to transport 2ppc, enabling up to 2160p60 performance.

2D Filter M2M Pipeline

The memory-to-memory (m2m) pipeline with the 2D convolution filter (see [Figure 8-7](#)) 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.

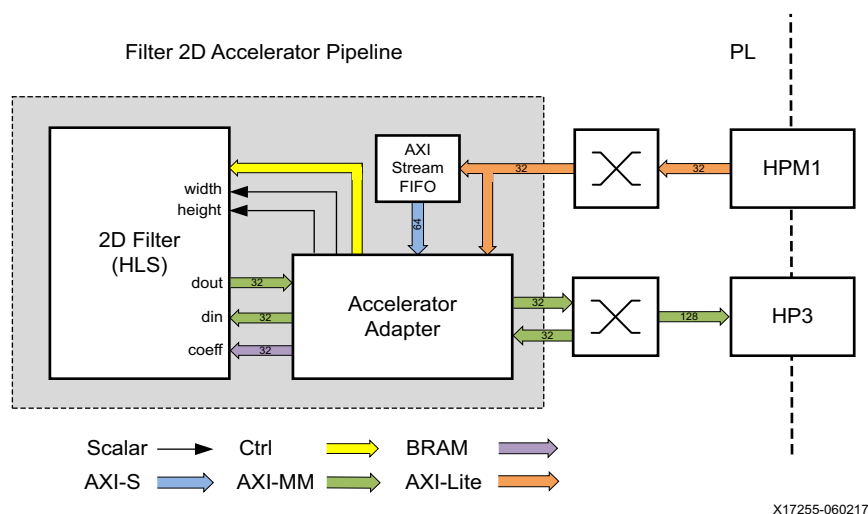


Figure 8-7: M2M Processing Pipeline Showing Hardware Accelerator and Data Motion Network

The HLS generated 2D filter accelerator is controlled by an accelerator adapter that drives all inputs and captures all outputs. The accelerator adapter has four AXI-MM interfaces: two for transferring video data from the HP3 port through the adapter to the accelerator; and two interfaces for the opposite direction. The filter coefficients are passed via a BRAM interface and the data mover is a simple AXI-Stream FIFO plus a data width converter. The width and height parameters are simple control signals (scalars) driven by the adapter.

Note that the m2m pipeline is not video-timing accurate. The data movers read and write video frames without inserting any horizontal blanking in between video lines. The QoS traffic class is therefore set to best effort instead of video (see [Quality of Service, page 52](#)). 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.

Optical Flow M2M Pipeline

The Optical Flow accelerator pipeline is conceptually identical with the 2D filter accelerator pipeline with a few notable differences:

- The optical flow accelerator consumes two video input frames and produces one output frame whereas the 2D filter has one input and one output.
- There is no block RAM interface for passing additional parameters to the accelerator at run-time. All parameters are compile time constants.

The Optical Flow pipeline is shown in [Figure 8-8](#).

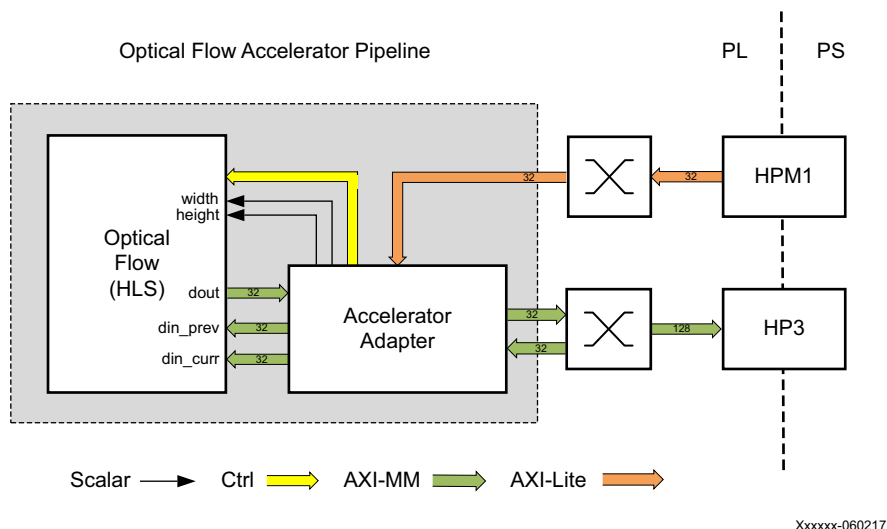
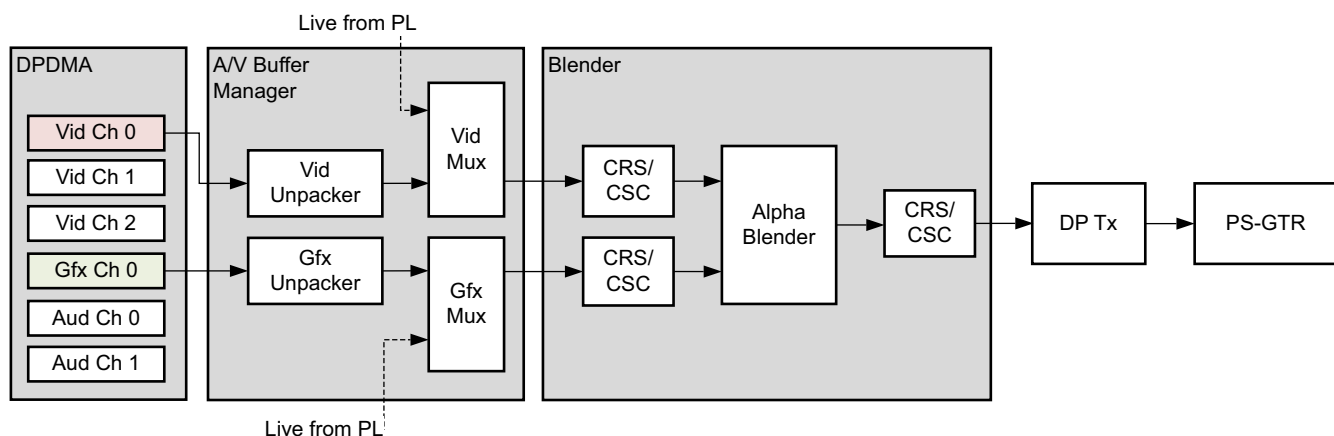


Figure 8-8: Optical Flow Pipeline

DP Tx Display Pipeline

The DP Tx display pipeline (see [Figure 8-9](#)) 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



X17256-071917

Figure 8-9: 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 48](#). 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 (SIU) 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 11].

HDMI Tx Display Pipeline

The HDMI transmitter display pipeline is shown in Figure 8-10.

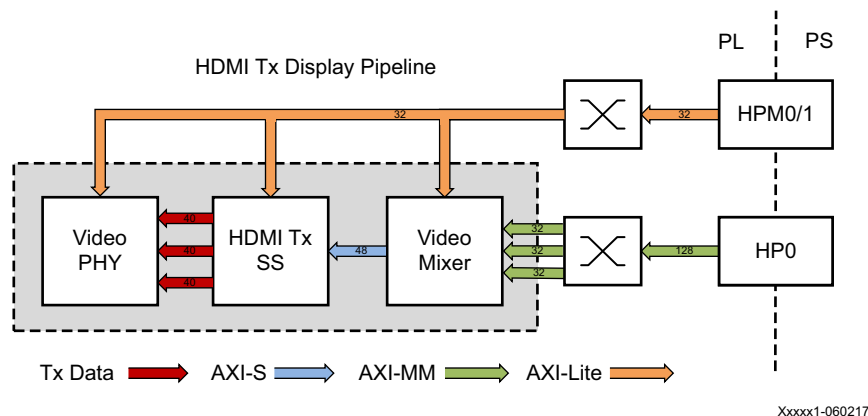


Figure 8-10: HDMI Tx Display Pipeline

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

- The video mixer IP core is configured to support blending of up to two video layers and one graphics layer into one single output video stream. The three layers are configured to be memory-mapped AXI4 interfaces connected to the HP0 high performance PS/PL interface via an AXI interconnect; the main AXI4-Stream layer is unused. The two video layers are configured for 16-bit YUYV, the graphics layer is configured for 32-bit ARGB, (see [Video Buffer Formats, page 48](#) for details). A built-in color space converter and chroma resampler convert the input formats to a 24-bit RGB output format. Pixel-alpha blending is used to blend the graphics layer with the underlying video layers. The AXI4-Stream output interface is a 48-bit bus that transports 2 ppc for up to 2160p60 performance. It is connected to the HDMI Tx subsystem input interface. A GPIO is used to reset the subsystem between resolution changes. For more information refer to the *Video Mixer v1.0 LogiCORE IP Product Guide* (PG243) [Ref 29].
- The HDMI Transmitter Subsystem (HDMI Tx) interfaces with PHY layers and provides HDMI encoding functionality. The subsystem is a hierarchical IP that bundles a collection of HDMI TX-related IP sub-cores and outputs them as a single IP. The subsystem generates an HDMI stream from the incoming AXI4-Stream video data and sends the generated TMDS data to the video PHY layer. For more information refer to the *HDMI 1.4/2.0 Transmitter Subsystem v2.0 Product Guide* (PG235) [Ref 30].
- The Video PHY Controller is shared between the HDMI Rx and HDMI Tx pipelines. Refer to [HDMI Rx Capture Pipeline, page 79](#) for more information on the VPHY and its configuration.

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	1,200 MHz
RPU	500 MHz
GPU	500 MHz
DDR	533 (1,066) MHz

Table 8-2 identifies the main clocks of the PL design, their source, their clock frequency, and function.

Table 8-2: System Clocks

Clock	Clock Source	Clock Frequency	Function
PL0 clock	PS	50 MHz	Clock source for clocking wizard
clk50	Clocking Wizard	50 MHz	AXI Lite clock
clk75	Clocking Wizard	75 MHz	Accelerator clock 1
clk150	Clocking Wizard	150 MHz	Accelerator clock 2
clk300	Clocking Wizard	300 MHz	AXI MM clock, AXI Stream clock, Accelerator clock 3
HDMI Tx GT reference clock	Si5324 (external)	Variable	Clock source for various HDMI Tx clocks
HDMI Tx Link clock	Video PHY	Variable	Clock used for data interface between HDMI Tx and Video PHY
HDMI Tx Video clock	Video PHY	Variable	Clock used for native HDMI Tx video interface
HDMI Tx TMDS clock	Video PHY	Variable	Synchronous clock to HDMI Tx interface
HDMI Rx GT reference clock /TMDS clock	HDMI source (external)	Variable	Synchronous clock to HDMI Rx interface
HDMI Rx Link clock	Video PHY	Variable	Clock used for data interface between HDMI Rx and Video PHY
HDMI Rx Video clock	Video PHY	Variable	Clock used for native HDMI Rx video interface
HDMI Rx DRU clock	Si570 (external)	156.25 MHz	Clock for data recovery unit for low line rates
DP Video reference clock	PS	Variable	DP Tx pixel clock
MIPI D-PHY core clock	Clocking Wizard	200 MHz	Clock used for control logic and input to MMCM
MIPI D-PHY Rx clock	FMC (external)	10-187.5 MHz ⁽¹⁾	Clock received on RX clock lane and used for high-speed data reception
MIPI CSI-2 Rx core clock	Clocking Wizard	300 MHz ⁽²⁾	Clock source for all IP cores in subsystem

Table 8-2: System Clocks (Cont'd)

Clock	Clock Source	Clock Frequency	Function
TPG video clock	Clocking Wizard	Variable	Pixel clock that drives VTC in TPG capture pipeline

Notes:

1. Derived from the line rate divided by 8.
2. Minimum frequency depends on data rate, lane, and width configuration.

The PL0 clock is provided by the IOPLL inside the PS and is used as reference input clock for the Clocking Wizard instance inside the PL. This clock does not drive any loads directly. A Clocking Wizard 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 Clocking Wizard instance. It is used to drive most of the AXI-Lite control interfaces in the PL. 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 Clocking Wizard 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 2160p60 which roughly corresponds to a 300 MHz clock at 2 ppc. 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* data mover and accelerator pipeline clocks can be selected by the user from the SDSoC GUI. The available options are *clk75*, *clk150*, and *clk300* which roughly correspond to 720p60, 1080p60, and 2160p30 at 1 ppc, or 2160p60 at 2 ppc. The same Clocking Wizard instance is used to generate these clocks.

For details on the HDMI Rx, HDMI Tx and VPHY clocking structure and requirements refer to the respective sections in PG230 [Ref 26], (PG235) [Ref 30], and (PG236) [Ref 27]. For HDMI Tx, an external Si5324 clock chip is used to generate the GT reference clock depending on the display resolution; for HDMI Rx, the GT reference clock is provided by the HDMI source. Various other HDMI related clocks are derived from the respective GT reference clocks and generated internally by the VPHY; only for the DRU a fixed reference clock is provided externally by an Si570 clock chip.

For details on the MIPI CSI-2 Rx clocking structure and requirements refer to the *MIPI CSI-2 Receiver Subsystem v2.2 Product Guide* (PG232) [Ref 31].

A second Clocking Wizard using the PL0 source clock generates the TPG pixel clock that is consumed by the VTC. The VTC in turn provides video timing signals to the TPG. The TPG pixel clock can be configured dynamically based on the requested input resolution.

The master reset *pl_resetn0* is generated by the PS during boot and is used as input to the four Processing System 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*, *clk75*, *clk150*, and *clk300* clock domains.

Apart from these system resets, there are asynchronous resets driven by PS GPIO pins. The respective device drivers control these resets which can be toggled at run-time to reset HLS-based cores. [Table 8-3](#) summarizes the PL resets used in this design.

Table 8-3: System and User Resets

Reset Source	Purpose
pl_resetn0	PL reset for proc_sys_reset modules
proc_sys_reset_clk50	Synchronous resets for clk50 clock domain
proc_sys_reset_clk75	Synchronous resets for clk75 clock domain
proc_sys_reset_clk150	Synchronous resets for clk150 clock domain
proc_sys_reset_clk300	Synchronous resets for clk300 clock domain
PS GPIO 79	Asynchronous reset for TPG
PS GPIO 80	Asynchronous reset for Frame Buffer Write (CSI Rx)
PS GPIO 81	Asynchronous reset for Frame Buffer Write (TPG)
PS GPIO 82	Asynchronous reset for VPSS Scaler (CSI Rx)
PS GPIO 83	Asynchronous reset for Video Mixer
PS GPIO 84	Asynchronous reset for VPSS CSC (CSI Rx)
PS GPIO 85	Asynchronous reset for Demosaic (CSI Rx)
PS GPIO 86	Asynchronous reset for Gamma LUT (CSI Rx)
PS GPIO 87	Asynchronous reset for VPSS Scaler (HDMI Rx)
PS GPIO 88	Asynchronous reset for Frame Buffer Write (HDMI Rx)

[Table 8-4](#) lists the PL-to-PS interrupts used in this design.

Table 8-4: PL-to-PS Interrupts

ID	Instance
0	TPG FB WR
1	HDMI Rx FB WR
2	HDMI Rx SS
3	VPHY
4	HDMI Tx SS
5	HDMI CTL I2C
6	CSI Rx SS
7	CSI Rx FB WR
8	Sensor CTL I2C
9	Video Mixer

The design uses two cascaded interrupt controllers: an AXI interrupt controller in the PL that connects to the general interrupt controller (GIC) in the PS. The AXI interrupt controller bundles all nine PL IP interrupts into a single GIC interrupt ID (89), thus making 15 PL interrupts available for SDSoC.

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 8-11](#).

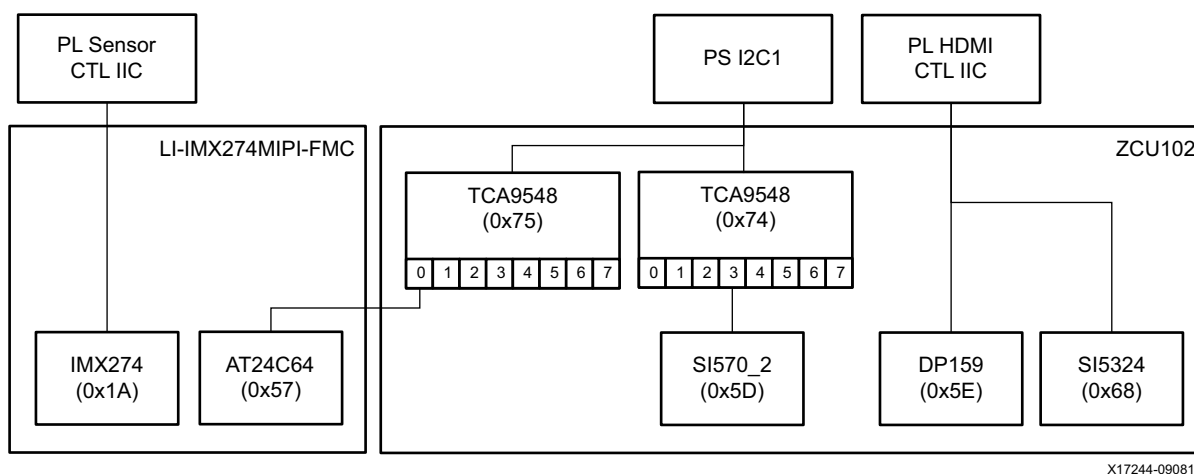


Figure 8-11: I2C Bus Topology

The PS I2C1 controller is connected via MIO to two 8-channel I2C multiplexers (TCA9548) on the ZCU102 board, one at address 0x74, the other at address 0x75.

The I2C multiplexer at address 0x74 has a Si570 programmable clock synthesizer at address 0x5D connected to channel 3. The Si570 clock synthesizer is connected to the PL via a differential clock pair. It generates a fixed reference clock for the HDMI Rx data recovery unit (DRU), see [Clocks, Resets and Interrupts, page 86](#) for more information.

The I2C multiplexer at address 0x75 has a 64 kb EEPROM at address 0x57 on the FMC daughter card connected to channel 0. The EEPROM is used to store FRU (Field Replaceable Unit) information data, as defined in the Intel Platform Management FRU Information Storage Definition [\[Ref 32\]](#).

The PL HDMI I2C controller is connected to two clock devices that are on a shared I2C bus. The SI5324 at address 0x68 is a programmable, low-bandwidth, jitter-attenuating, precision clock multiplier. Its output clock is used to drive the HDMI Tx GT reference clock.

The DP159 at address 0x5E is a DisplayPort to TMDS retimer supporting HDMI output signals. For more information on clocking see [Clocks, Resets and Interrupts, page 86](#).

The PL sensor I2C controller is connected to a Sony IMX274 image sensor at address 0x1A on the FMC daughter card. Refer to [MIPI CSI-2 Rx Capture Pipeline, page 80](#) for more information.

Auxiliary Peripherals

Other auxiliary interfaces used in this design are as follows:

- UART (2x PS): the first PS UART is owned by the APU. It prints Linux boot and debug messages to the serial console and provides a serial login prompt. The second PS UART is owned by either RPU-0 or RPU-1 and prints application messages to the serial console. Both UARTs are connected to a quad USB-UART bridge on the ZCU102 board.
- USB (1x PS) is used to connect a mouse to the ZCU102 board to operate the GUI, a webcam that serves as an additional video source (optional), and a USB file storage device (optional). Multiple devices can be connected via a USB hub.
- I2C (1x PS, 1x PL) is used to configure slave peripherals on the ZCU102 board (see [I2C Bus Topology, page 89](#)).
- GPIO (1x PS) is used to drive the reset signals of various HLS-based IPs in the PL. (see [Clocks, Resets and Interrupts, page 86](#)).
- SD (1x PS) is the boot device and holds the entire reference design image including FSBL, PMU firmware, PL bitstream, u-boot, Linux kernel, root file system, RPU-1 firmware, and APU/RPU-0 applications (see [Boot Process, page 44](#)).
- SATA (1x PS) is used to connect a file storage device (optional).
- Ethernet (1x PS) is used to connect to a network (optional).

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.

Documentation Navigator and Design Hubs

Xilinx[®] Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado[®] IDE, select **Help > Documentation and Tutorials**.
- On Windows, select **Start > All Programs > Xilinx Design Tools > DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

Note: For more information on Documentation Navigator, see the [Documentation Navigator](#) page on the Xilinx website.

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 Platform Development User Guide* ([UG1146](#))
3. *ZCU102 Evaluation Board User Guide* ([UG1182](#))
4. [Leopard Imaging Inc.](#) website
5. [Performing Convolution Operations](#) website
6. [OpenCV - 2D Filter API](#) website
7. *Xilinx OpenCV User Guide* ([UG1233](#))
8. *Vivado Design Suite User Guide - High-Level Synthesis* ([UG902](#))
9. *Zynq UltraScale+ MPSoC Software Developer Guide* ([UG1137](#))
10. *Xilinx OpenCV Library User Guide* ([UG1233](#))
11. *Zynq UltraScale+ MPSoC Technical Reference Manual* ([UG1085](#))
12. [VESA E-EDID Standard](#) website
13. [Qt5 Embedded Linux](#) website
14. *AXI4-Stream Video IP and System Design Guide* ([UG934](#))
15. [Recommendation ITU-R BT.656-4](#) website
16. [FRU Information Storage for IPMI](#) website
17. *OpenAMP Framework for Zynq Devices Getting Started Guide* ([UG1186](#))
18. L. Pinchart, *Anatomy of an Atomic KMS Driver*, Kernel Recipes, Paris, 2015
19. *UltraFast Embedded Design Methodology Guide* ([UG1046](#))
20. L. Pinchart, *V4L2 on Steroids - The Request API*, Embedded Linux Conference, San Diego, 2016
21. [Xilinx V4L2 Pipeline Driver](#) website
22. *Clocking Wizard v6.0 LogiCORE IP Product Guide* ([PG065](#))
23. *Video Timing Controller v6.1 LogiCORE IP Product Guide* ([PG016](#))
24. *Video Test Pattern Generator v7.0 LogiCORE IP Product Guide* ([PG103](#))

25. *Video Frame Buffer Read v1.0 and Video Frame Buffer Write v1.0 LogiCORE IP Product Guide* ([PG278](#))
26. *Video PHY Controller v2.0 Product Guide* ([PG230](#))
27. *HDMI 1.4/2.0 Receiver Subsystem v2.0 Product Guide* ([PG236](#))
28. *Video Processing Subsystem v2.0 Product Guide* ([PG231](#))
29. *Video Mixer v1.0 LogiCORE IP Product Guide* ([PG243](#))
30. *HDMI 1.4/2.0 Transmitter Subsystem v2.0 Product Guide* ([PG235](#))
31. *MIPI CSI-2 Receiver Subsystem v2.2 Product Guide* ([PG232](#))
32. [Platform Management FRU Information Storage Definition](#)

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 2016-2018 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.