TRD:

TRD (Targeted Reference Design):

- A targeted reference design is a predefined configuration or setup that serves as a starting point for developing a specific type of system or product. In this context, it's likely related to a video processing system or a similar application.

VCU (Video Codec Unit):

- The VCU is a component or module designed for video encoding and decoding. It's responsible for processing video data, compressing it for storage or transmission (encoding), and decompressing it for display or further processing (decoding).

Performance Parameters:

- These are variables or settings that influence how well the VCU performs its tasks. Performance parameters can include things like encoding speed, video quality, compression ratios, and other factors relevant to video processing.

Encoder and Decoder Blocks:

- The encoder and decoder are fundamental components of a video codec. The encoder converts raw video data into a compressed format, while the decoder reverses this process, converting compressed data back into a usable video format.

Optimal Configurations:

- Finding the optimal configurations means determining the best settings for the encoder and decoder blocks to achieve a balance between performance and desired outcomes. This could involve adjusting parameters to improve video quality, reduce file sizes, or enhance processing speed.

Platform for Tuning:

- The TRD serves as a platform or framework for making adjustments to the VCU's performance parameters. It provides a controlled environment where engineers and developers can experiment with different settings to find the most efficient and effective configurations for the encoder and decoder blocks.

Arriving at Optimal Configurations:

- The goal is to experiment with various configurations until the best combination of settings is identified. This involves testing and evaluating the performance of the VCU under different conditions to ensure that it meets specific requirements or criteria.

In summary, the statement is describing how the Targeted Reference Design (TRD) is used as a controlled environment or framework for adjusting the performance parameters of the Video Codec Unit (VCU), specifically focusing on the encoder and decoder blocks. The ultimate aim is to find the optimal configurations that result in the desired outcomes for video processing, such as high-quality compression, efficient decoding, or other relevant criteria.

The TRD demonstrates the following hard block features in the PS and PL:

VCU Hard Block:

- The VCU (Video Codec Unit) hard block is a dedicated hardware module for video processing. In this case, it's capable of handling video resolutions up to 4K (3840 x 2160 or 4096 x 2160).

Simultaneous Encoding and Decoding:

- The VCU is capable of simultaneously encoding and decoding both single and multiple video streams. This is a significant feature for applications requiring real-time video processing.

PS DisplayPort Controller:

- The Processing System includes a DisplayPort controller capable of handling 4K video resolution at 30 Hz. DisplayPort is a digital display interface commonly used to connect video sources to displays.

PL-based HDMI-TX/SDI-TX:

- The Programmable Logic includes modules for HDMI and SDI transmission, supporting 4K video resolutions at 60 Hz. HDMI and SDI are standards for transmitting high-definition video and audio signals.

GPU for GUI Rendering:

- A Graphics Processing Unit (GPU) is used for rendering a Graphical User Interface (GUI). This implies that the system has a visual interface, likely for user interaction and control.

Extensible Platform:

- The platform is extensible and supports various technologies and standards:
- **GStreamer v1.18.5:** GStreamer is a multimedia framework that facilitates the construction of multimedia pipelines. It's likely used for managing and processing audio and video streams in a flexible and modular way.
- **Standard Linux Software Frameworks:** The system is built on standard Linux software frameworks, providing a familiar and widely-used environment for software development.
- **OpenMAX™ v1.1.2:** OpenMAX is a set of application programming interfaces (APIs) for multimedia processing, and in this case, it provides a client interface for interacting with the VCU.
- **Modular and Hierarchical Architecture:** The system architecture is designed to be modular and hierarchical, allowing for flexibility and scalability. Partner modules can be integrated as needed.
- **Configurable IP Subsystems:** The Intellectual Property (IP) subsystems in the programmable logic are configurable, allowing customization to meet specific application requirements.

System Software Configuration:

- The provided link leads to documentation that likely contains information on how to configure the system software. This could include details on setting up the Linux environment, configuring the multimedia pipeline using GStreamer, and other relevant configurations.

In summary, the TRD is a comprehensive solution for multimedia processing, leveraging both hardware (VCU, GPU) and software (Linux, GStreamer, OpenMAX) components to support 4K video processing, GUI rendering, and a flexible, extensible architecture.

Zynq UltraScale+ MPSoC Overview

Application Processing Unit (APU):

- The MPSoC includes an APU featuring a 64-bit quad-core Arm Cortex-A53 processor. Cortex-A53 is a power-efficient processor commonly used for general-purpose applications.

Real-Time Processing Unit (RPU):

- The MPSoC incorporates an RPU with a 32-bit dual-core Arm Cortex-R5 processor. Cortex-R5 is designed for real-time processing applications, offering deterministic and reliable performance.

Multimedia Blocks:

- **Graphics Processing Unit (GPU):** The MPSoC includes an Arm Mali-400MP2 GPU. The Mali-400MP2 is a graphics processor used for rendering graphical content and supporting visual applications.
- **Video Codec Unit (VCU):** The VCU is a video codec unit capable of encoding and decoding video content. It supports resolutions up to 4K (3840 x 2160) at 60 frames per second (FPS).

DisplayPort Controller Interface:

- The MPSoC features a DisplayPort controller interface supporting resolutions up to 4K (3840 x 2160) at 30 FPS. This interface facilitates the connection to displays using the DisplayPort standard.

High-Speed Peripherals:

- **PCIe (Peripheral Component Interconnect Express):** The MPSoC includes PCIe support as both a root complex and endpoint, offering flexibility with Gen1 or Gen2 configurations and different lane options (x1, x2, x4).
- **USB 3.0/2.0:** USB support with host, device, and on-the-go (OTG) modes.
- **SATA 3.1 Host:** Support for SATA (Serial ATA) 3.1 for high-speed data transfer with storage devices.

Low-Speed Peripherals:

- Various low-speed peripherals are supported, including Gigabit Ethernet, Controller Area Network (CAN), Universal Asynchronous Receiver-Transmitter (UART), Serial Peripheral Interface (SPI), Quad SPI, NAND flash memory, Secure Digital Embedded Multimedia Card (SD/eMMC), Inter-IC (I2C), and General Purpose I/O (GPIO).

Platform Management Unit (PMU):

- The PMU is responsible for managing various aspects of the platform, including power management, system monitoring, and control.

Configuration Security Unit (CSU):

- The CSU is likely responsible for ensuring the security of the MPSoC's configuration. It plays a role in safeguarding the configuration settings from unauthorized access or tampering.

6-Port DDR Controller:

- The MPSoC includes a DDR (Double Data Rate) controller with error correction code (ECC) support. It can handle x32 and x64 DDR4/3/3L and LPDDR4/3 memory configurations. DDR controllers manage the communication between the processor and the system's main memory.

In summary, the MPSoC is a versatile and powerful system incorporating application and real-time processing units, multimedia capabilities, high-speed and low-speed peripherals, and security features. It is designed to handle a range of tasks, from general-purpose computing to real-time processing and multimedia applications.

Reference Design Overview

Heterogeneous Processor Architecture:

- The MPSoC features a heterogeneous processor architecture, meaning it includes different types of processors. The description focuses on the APU (Application Processing Unit), which consists of quad Arm Cortex-A53 cores configured to run in SMP (Symmetric Multiprocessing) Linux mode.

Application Task and GUI:

- The main task of the application running on the APU is to configure and control video pipelines. This is accomplished through a graphical user application based on Qt v5.9.4, a popular cross-platform application framework.

Video Data Paths:

- The APU application controls various video data paths implemented in the Processing System (PS) and Programmable Logic (PL). These include:
- Capture Pipeline: Captures video frames from various sources such as HDMI, image sensors via MIPI CSI-2 RX Subsystem, SDI sources, and a Test Pattern Generator (TPG) implemented in the PL. Video can also be sourced from SATA drives, USB 3.0 devices, or an SD card.
- Processing Pipeline: Involves the Video Codec Unit (VCU) for encoding and decoding video frames. Video frames are read from DDR memory, processed by the VCU, and written back to memory.
- Display Pipeline: Reads video frames from memory and sends them to a monitor via the DisplayPort TX Controller inside the PS, SDI Transmitter Subsystem through the PL, or HDMI Transmitter Subsystem through the PL. The DisplayPort TX Controller and the SDI Transmitter Subsystem support multiple layers for video and graphics.
- Audio Capture and Renderer Pipelines: Capture and playback audio frames from HDMI-RX, SDI-RX, and I2S-RX interfaces, and playback through HDMI-TX, SDI-TX, DP, and I2S-TX interfaces.
- LLP2 Pipeline: Streams out and in live captured video at ultra-low latencies using Sync IP.

Graphics Rendering:

- The graphics layer for the display pipeline is rendered by the GPU (Graphics Processing Unit).

Software state:

- The software state is depicted in a figure after the boot process has completed, and individual applications have started on the processing units. The design does not use virtualization, and there is no hypervisor running on the APU.

In summary, the Reference Design (TRD) showcases the utilization of the MPSoC's capabilities, involving different processing units, video capture and processing pipelines, display output, audio processing, and low-latency video streaming. It provides a comprehensive solution for video and audio applications with a focus on real-time processing and multimedia capabilities.

Software state:

The "Software State after Boot" refers to the condition of the system's software components after the completion of the boot process. Here's a brief explanation of the key points mentioned:

Completion of Boot Process:

- The boot process is the series of steps a computer system goes through when it is powered on or restarted. This process initializes the hardware, loads the operating system into memory, and prepares the system for user interaction.

Individual Applications Started:

- After the boot process is complete, individual applications are launched on the processing units. In this context, the focus is on the Application Processing Unit (APU) with quad Arm Cortex-A53 cores configured to run in Symmetric Multiprocessing (SMP) Linux mode.

No Virtualization Used:

- The design explicitly states that virtualization is not employed. Virtualization is a technique that allows multiple operating systems or instances to run on a single physical machine. In this case, there is no hypervisor (a software layer that manages virtual machines) running on the APU.

System State Illustration:

- The reference design likely includes a visual representation or figure illustrating the state of the system's software components post-boot. This figure could show the APU with its quad-core Cortex-A53 processors actively running, along with any other relevant software modules.

Focused Execution:

- Without virtualization or a hypervisor, the APU is dedicated to running the intended applications without the overhead of managing virtualized environments. This approach is common in scenarios where the system's primary purpose is to execute specific tasks efficiently.

In summary, the "Software State after Boot" represents the state of the system's software components, particularly on the APU, after the completion of the boot process. The absence of virtualization and a hypervisor suggests a direct and dedicated execution environment for the intended applications without the added complexity of virtualized infrastructure.

Example for No Virtualization Used:

Analogy: Office Environment

No Virtualization (Dedicated Environment):

- Imagine a traditional office environment where each employee has their own physical desk, computer, and workspace. In this scenario:
- Each employee (equivalent to the APU) has a dedicated workspace (Cortex-A53 processors) for their tasks.
- There is no shared desk or computer. Each employee works independently without interference from others.
- This represents a non-virtualized environment. The employees are like the dedicated processing units, and their desks are akin to the system's hardware resources.

With Virtualization:

- Now, let's introduce virtualization. In this scenario:
- A hypervisor (manager) allows multiple employees to share a single computer and desk. Each employee has a virtualized workspace.
- The hypervisor manages and allocates resources to each virtual workspace, ensuring that they operate independently and do not interfere with each other.
- This is similar to running multiple virtual machines (VMs) on a single physical server in the IT context.

Application to the Given Scenario:

- In the provided scenario, the system is like the non-virtualized office environment. The APU (employee) has its dedicated processing units (workspace), and there is no hypervisor (manager) overseeing virtualization.
- The absence of virtualization means that the APU, with its quad-core Cortex-A53 processors, directly runs the intended applications without the need for a virtualization layer. Each core is akin to an independent employee handling specific tasks.
- The "Software State after Boot" figure in the reference design could be visualized as a snapshot of the APU actively running its applications, just like employees at their desks in our analogy.

In summary, not using virtualization in this context is akin to having a dedicated, non-shared environment for the APU to efficiently execute tasks without the overhead of managing virtualized instances.

SMP Linux mode:

"SMP" stands for Symmetric Multiprocessing. In computing, SMP refers to a multiprocessor computer architecture in which two or more identical processors are connected to a single, shared main memory and are controlled by a single operating system instance. In SMP systems, each processor has equal access to resources (like memory or I/O devices) and performs identical tasks, but they operate independently.

Key features of SMP include:

Multiple Processors:

- SMP systems have two or more processors (also known as cores) working together to execute tasks.

Shared Memory:

- All processors in an SMP system share the same main memory. This shared memory allows for communication and data sharing among processors.

Single Operating System Instance:

- SMP systems are managed by a single instance of an operating system. This operating system is responsible for distributing tasks among the processors and coordinating their activities.

Parallel Processing:

- SMP enables parallel processing, where multiple processors can work on different parts of a task simultaneously. This parallelism can lead to improved performance for multithreaded or parallelizable applications.

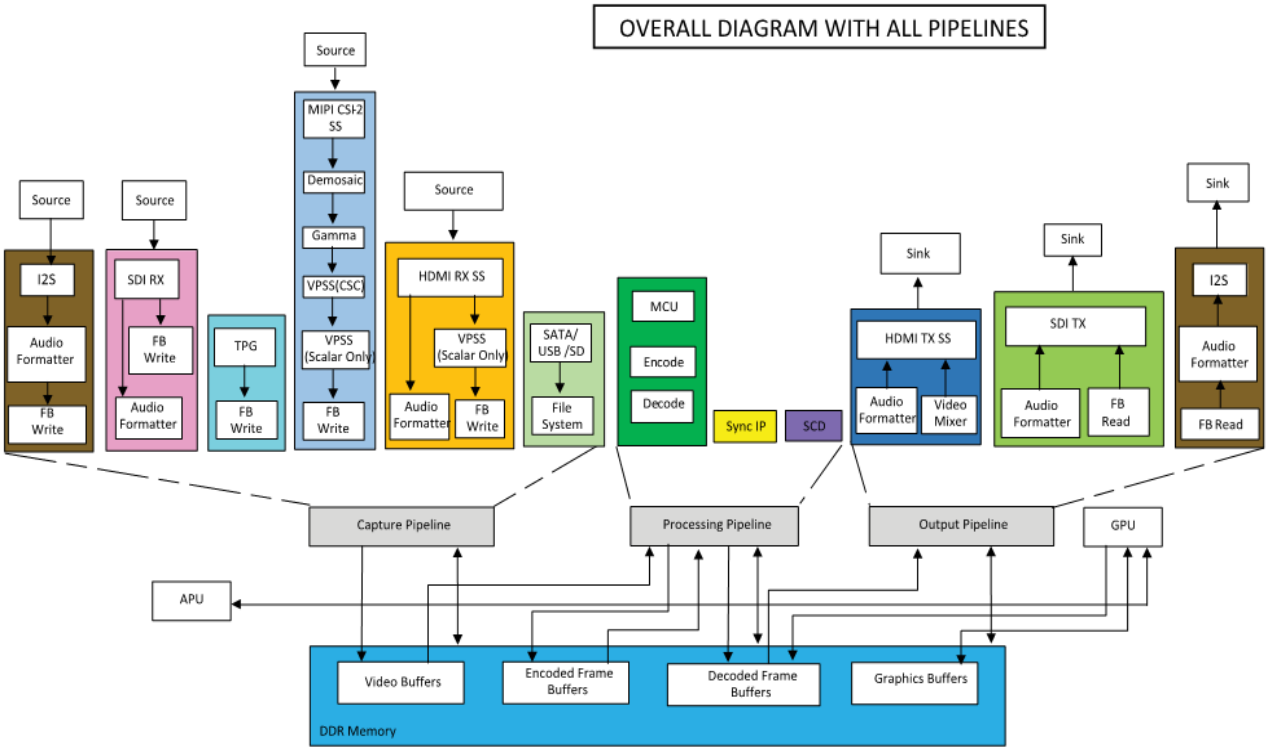
Load Balancing:

- The operating system in an SMP environment often implements load balancing mechanisms to distribute tasks evenly among the available processors. This ensures efficient utilization of resources.

In the context of "SMP Linux mode," it likely refers to running a Linux operating system in a configuration that supports Symmetric Multiprocessing. This means that the Linux kernel is designed to work seamlessly on systems with multiple processors or cores, allowing them to collaborate on running applications and tasks concurrently.

SMP is widely used in modern computing systems, including servers, desktops, and some high-performance computing environments, to enhance overall system performance and responsiveness.

VCU TRD Block Diagram



The remaining blocks are common to all designs. See Targeted Reference Design Details for more details.

Common Blocks:

- The reference design includes certain blocks that are common to all designs. These could be fundamental components or functionalities that are shared among different configurations or use cases. Unfortunately, the specific details of these common blocks are not provided in your message.

Target Platform:

- The reference design is intended for the ZCU106 evaluation board. The ZCU106 is a development board from Xilinx, designed for the Zynq UltraScale+ MPSoC (Multiprocessor System on Chip). This board provides a platform for developing and testing applications based on the Zynq UltraScale+ architecture.

Onboard Connectors:

- The ZCU106 evaluation board features onboard connectors for HDMI, SDI (Serial Digital Interface), and DisplayPort. These connectors facilitate connectivity with external devices and peripherals, allowing the system to interface with various video sources and displays.

Clock Sources:

- The evaluation board provides several clock sources that are crucial for video processing and synchronization. These include:
- **HDMI Reference Clock:** A clock signal derived from the HDMI interface, used for synchronization and timing in HDMI-related operations.

- **Data Recovery Unit (DRU) Clock:** This clock is likely associated with the Data Recovery Unit, which is used to extract clock and data information from a serial data stream.
- **Reference Clock:** Another clock signal used as a general reference for the overall design.

PS_REF_CLK:

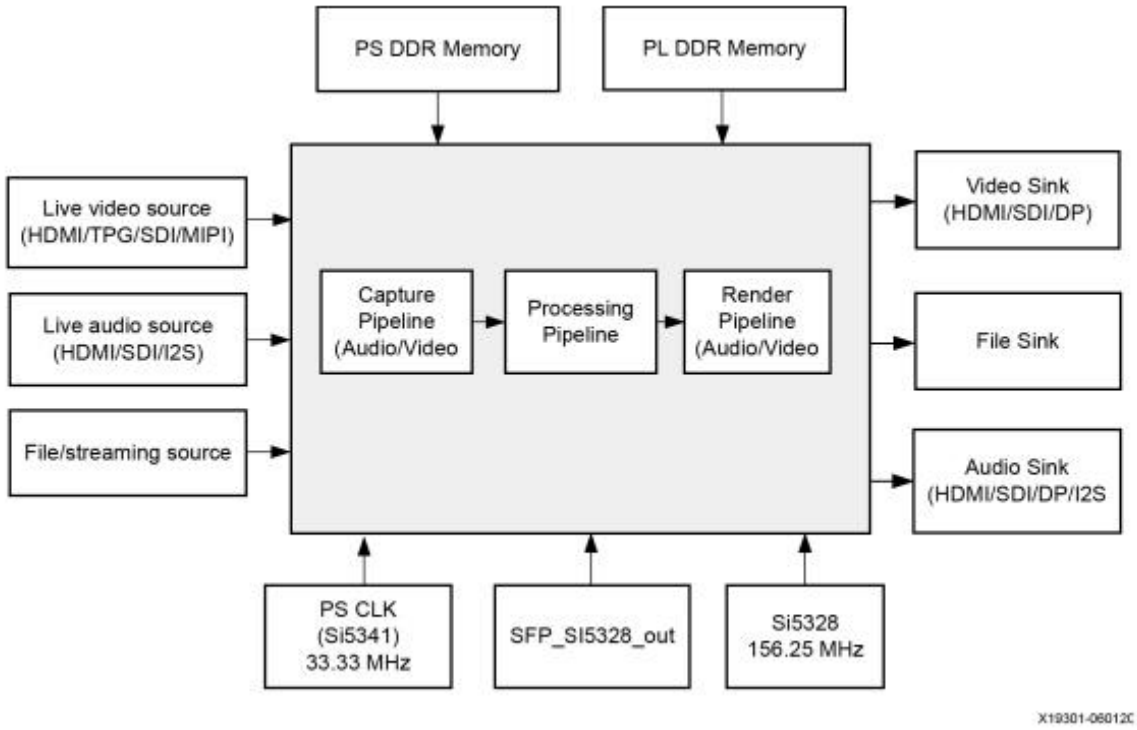
- The PS_REF_CLK is a clock signal sourced from a dedicated clock generator on the evaluation board. It is likely used as a reference clock for the processing system (PS) part of the Zynq UltraScale+ MPSoC.

Block Diagram:

- Unfortunately, the specific block diagram or details of the TRD, as mentioned in your message, are not provided. Typically, a block diagram visually represents the components and their interconnections in a system, offering a high-level overview of the design.

Without the specific details of the common blocks, it's challenging to provide a more detailed explanation. If you have access to the full Targeted Reference Design documentation, you may find additional details about the common blocks, their functionalities, and their roles in the overall system architecture.

Figure: High-Level Block Diagram of ZCU106 Device Architecture



Key Features:

AMD Tools:

- **AMD Vivado™ Design Suite 2023.1:**
- Vivado is a comprehensive development environment for Xilinx FPGA and SoC designs. The AMD version is specified as 2023.1, indicating the version number.
- **PetaLinux tools 2023.1:**
- PetaLinux is a development environment used for creating embedded Linux systems for Xilinx devices. The AMD version mentioned is 2023.1.

Hardware Interfaces and IP:

Video Inputs:

- **TPG (Test Pattern Generator):**
- A component that generates test patterns for video testing and evaluation.
- **HDMI RX (Receiver):**
- Interface for receiving HDMI video signals.
- **MIPI CSI-2 RX:**
- Interface for receiving video signals using the MIPI CSI-2 standard, commonly used in mobile and camera systems.
- **File Source (SD Card, SATA, USB 3.0 Drives):**
- Interfaces for reading video data from storage devices like SD cards, SATA drives, and USB 3.0 drives.
- **SDI RX (Serial Digital Interface Receiver):**
- Interface for receiving video signals using the SDI standard, often used in professional video production.

- **Stream In:**
- Generic term suggesting the capability to receive video streams.

Video Outputs:

- **DisplayPort TX Controller:**
- Interface for transmitting video signals using the DisplayPort standard.
- **HDMI TX (Transmitter):**
- Interface for transmitting HDMI video signals.
- **SDI TX (Transmitter):**
- Interface for transmitting SDI video signals.
- **PICXO IP:**
- Unspecified IP, but likely related to video processing or interfacing.

Audio Inputs:

- **HDMI RX:**
- Interface for receiving audio signals embedded in HDMI video streams.
- **SDI RX:**
- Interface for receiving audio signals embedded in SDI video streams.
- **I2S RX (Inter-IC Sound Receiver):**
- Interface for receiving audio signals using the I2S standard.

Audio Outputs:

- **HDMI TX:**
- Interface for transmitting audio signals embedded in HDMI video streams.
- **SDI TX:**
- Interface for transmitting audio signals embedded in SDI video streams.
- **I2S TX (Inter-IC Sound Transmitter):**
- Interface for transmitting audio signals using the I2S standard.

Video Compression/Decompression:

- **VCU Hard Block:**
- Video Codec Unit for hardware-accelerated video compression and decompression.

Auxiliary Peripherals:

- Various peripheral interfaces including SD, I2C, GPIO, 1G/10G Ethernet, UART, USB 2.0/USB 3.0, AXI Performance Monitor (APM), PCIe, Digilent PMOD audio card (I2S2), 3.5 mm auxiliary cables, and speakers.

Software Components:

- **Operating Systems:**
- APU (Application Processing Unit) runs SMP Linux.
- **Linux Frameworks/Libraries:**
- Video: Video4Linux (V4L2), Media controller.
- Audio: libalsa.

- Display: Direct Rendering Manager/Kernel Mode Setting (DRM/KMS), X-Server (X.Org).
- **Graphics:**
- Qt5, OpenGL ES2.
- **User Application:**
- APU: GStreamer-based command line application, QT GUI application.

Supported Video Formats:

- **Input Resolution:**
- DCI 4Kp60 (4096 x 2160), 4Kp30 (3840 x 2160), 1080p60 (1920 x 1080), 1080p30 (1920 x 1080).
- **Output Resolution:**
- DCI 4Kp60 (4096 x 2160) for HDMI/SDI, 4Kp30 (3840 x 2160) for HDMI and DisplayPort, Native 1080p60 on both DisplayPort and HDMI.
- **Pixel Formats:**
- NV12, NV16, XV15, XV20, YU24, X403.

This system appears to be a sophisticated video processing setup with a focus on high-resolution video input and output, various video interfaces, audio processing, and support for common industry standards. The combination of AMD tools, Vivado, PetaLinux, and the specified hardware and software components suggests a comprehensive solution for multimedia applications.

Targeted Reference Design Details

Overview:
The "Multi-Stream Audio Design" module is designed to handle the capture and processing of both audio and video data. It incorporates features such as support for Scene Change Detection (SCD) IP and allows for streaming audio/video data through various interfaces.

Key Components and Functionality:

- **Data Capture:**
- This module enables the capture of audio data from I2S RX and HDMI RX sources. Additionally, video data can be captured from HDMI RX and MIPI RX subsystems.
- **Scene Change Detection (SCD) IP:**
- The design includes support for Scene Change Detection IP. SCD is a technology that identifies changes in a video scene, which can be useful for various applications, such as video analytics and content analysis.
- **Stream-based SCD:**
- SCD is supported in the HDMI pipeline in this design. This implies that Scene Change Detection is applied specifically to the HDMI video stream, detecting changes in scenes.
- **Audio/Video Playback and Recording:**

- The captured audio and video data can be played through HDMI TX in the Programmable Logic (PL) part of the system. Additionally, the data can be recorded in SD cards or USB/SATA drives.
- **Ethernet Interface:**
- The module provides the capability to stream in and out the audio/video data through an Ethernet interface. This suggests the possibility of transmitting and receiving audio/video data over a network using Ethernet.

Supported Streams:

- **Stream 1:**
- **Input Source:** Video and audio are captured from HDMI RX.
- **Output Sink:** Video and audio are played on HDMI TX.
- **Stream 2:**
- **Input Source:** Video is captured from the MIPI RX subsystem, and audio is captured from the I2S RX subsystem.
- **Output Sink:** Video is played on HDMI TX, and audio is played on I2S TX.

Use Cases:

- **Stream 1: HDMI RX Loop:**
- This stream captures both video and audio from an HDMI source and plays them back through HDMI TX. This loopback configuration can be useful for testing and verification.
- **Stream 2: MIPI RX and I2S RX:**
- This stream captures video from the MIPI RX subsystem and audio from the I2S RX subsystem. The captured video is played on HDMI TX, and the audio is played on I2S TX. This configuration may be suitable for scenarios where video and audio are coming from different sources.

In summary, the "Multi-Stream Audio Design" module is a versatile component of the VCU TRD, allowing for the capture, processing, and streaming of audio and video data from various sources, with specific support for Scene Change Detection and different playback configurations.

Example for Scene Change Detection

Consider a surveillance camera system that employs Scene Change Detection to enhance its functionality:

Scenario Setup:

- The surveillance camera is continuously recording video footage in a specific area.

Normal Surveillance Mode:

- During normal operation, the camera captures the ongoing activities in the monitored area. The video stream consists of regular scenes with people moving around, occasional vehicles passing by, and typical environmental changes.

Scene Change Event:

- Suddenly, an event occurs, such as a person entering the scene wearing a bright-colored uniform. This change is significant and alters the visual characteristics of the scene.

SCD Activation:

- The Scene Change Detection algorithm detects this abrupt change in the video stream. It recognizes the shift from a normal scene to an eventful scene with a person wearing a distinct uniform.

Notification or Action Trigger:

- Upon detecting the scene change, the surveillance system can initiate specific actions, such as:
- Sending an alert to security personnel about the unusual event.
- Starting a recording with higher resolution for detailed analysis.
- Adjusting the camera settings, such as zoom or focus, to get a clearer view of the changed scene.
- Marking the event in the video timeline for easy retrieval during later analysis.

Enhanced Video Analytics:

- Video analytics applications can benefit from SCD to identify and analyze significant events. For example:
- Retail analytics: Detecting when a new customer enters a store.
- Traffic monitoring: Recognizing sudden changes, like a road accident or traffic congestion.
- Smart cities: Identifying unusual activities or events in public spaces.

Content Analysis and Summarization:

- SCD is valuable for content analysis and summarization. For instance:
- Creating a summarized version of a long video stream by highlighting scenes with changes or events.
- Automatically generating video highlights based on detected scene changes.

Optimizing Bandwidth Usage:

- In a video streaming application, SCD can be used to optimize bandwidth usage. If there are no significant scene changes, the video compression settings can be adjusted dynamically to save bandwidth.

In this real-time example, Scene Change Detection is a critical component of a surveillance system, allowing for the automatic identification of noteworthy events or changes in a video stream. It enhances the system's ability to respond promptly to important events and facilitates efficient content analysis.

Full-fledged VCU TRD:

Overview:
The "Full-fledged VCU TRD" module is designed to provide comprehensive video processing capabilities using the Video Codec Unit (VCU) within the system. It includes features such as video capture, Scene Change Detection (SCD), display, recording, and streaming.

Key Components and Functionality:
Video Capture Sources:

- This module allows video capture from multiple sources:
- HDMI source

- Image sensor connected through CSI-2 RX (Camera Serial Interface)
- Test Pattern Generator (TPG) implemented in the Programmable Logic (PL)

Scene Change Detection (SCD) IP:

- Similar to the Multi-Stream Audio Design module, this module also incorporates support for Scene Change Detection IP (SCD IP).

SCD in Memory-Based Mode:

- SCD is supported in memory-based mode, indicating that the Scene Change Detection process analyzes changes in the video frames stored in memory. This mode allows for detecting significant alterations in the visual content.

Video Display:

- The captured video can be displayed via DisplayPort TX (DP TX) through the processing system (PS). This means the video can be shown on a display or monitor connected to the DisplayPort interface.

Video Output - HDMI TX:

- Additionally, the video can be output through HDMI TX via the Programmable Logic (PL). HDMI TX is responsible for transmitting the video signal over an HDMI interface, possibly for connecting to external devices such as TVs or monitors.

Video Recording:

- The captured video can be recorded in SD cards or USB/SATA drives. This allows for storage and retrieval of video content for later analysis or playback.

Streaming:

- The module supports streaming of encoded data through an Ethernet interface. This means that the video content, possibly after being processed or encoded by the VCU, can be transmitted over a network through an Ethernet connection.

Use Cases:

Video Capture from Various Sources:

- The module supports versatile video capture from HDMI, image sensors, and the Test Pattern Generator. This flexibility allows for testing and demonstration purposes.

Scene Change Detection:

- The inclusion of Scene Change Detection enhances the module's capabilities to identify and respond to significant changes in the video content.

Display and Output Options:

- The ability to display video via DP TX and output it through HDMI TX provides flexibility in how the processed video is presented and transmitted to external devices.

Recording and Streaming:

- Video recording to SD cards or USB/SATA drives enables local storage, while streaming through an Ethernet interface facilitates remote access to the video content.

In summary, the "Full-fledged VCU TRD" module is a comprehensive solution for video processing, capture, and distribution. It incorporates various features for versatile use, including Scene Change Detection, multiple video capture sources, display options, and support for recording and streaming.

PL DDR HDR10 HDMI Video Capture and Display:

Certainly! Let's break down the information provided about the module that supports the reception and insertion of HDR10 static metadata for HDMI:

Key Features and Functionality:

HDR10 Metadata:

- The module supports the reception and insertion of High Dynamic Range (HDR) static metadata specifically for the HDR10 standard used in HDMI. HDR10 metadata includes critical information necessary to support High Dynamic Range content.

Critical Information:

- HDR10 metadata carries essential details required to reproduce HDR content faithfully. This information is crucial for maintaining color accuracy, brightness levels, and other parameters associated with HDR video.

Pipeline Support:

- The HDR10 metadata is carried throughout the video processing pipeline, from the source to the sink. This ensures that the HDR information is maintained and utilized across various stages of video processing.

Capture of HDR10 Video:

- The module enables the capture of HDR10 video from an HDMI-Rx subsystem implemented in the Programmable Logic (PL). This implies that the module can receive and process HDR10 content from an HDMI source.

Display through HDR10 Compatible HDMI-Tx:

- The processed HDR10 video can be displayed through an HDR10-compatible HDMI-Tx (HDMI Transmitter) implemented in the PL. This allows for the transmission of HDR content to external displays or devices that support HDR10.

Recording in SD Cards or USB/SATA Drives:

- The captured and processed HDR10 video can be recorded in SD cards or USB/SATA drives. This enables the storage of HDR content for future playback or analysis.

Streaming:

- Similar to the previous module descriptions, this module supports streaming of HDR10 encoded data through an Ethernet interface. This means that HDR10-encoded video content can be transmitted over a network.

Support for RAW and Processed HDR10 Use-Cases:

- The module is designed to support both single-stream RAW HDR10 (without VCU - Video Codec Unit) and processed HDR10 use-cases with VCU. This flexibility allows for different processing scenarios depending on the application requirements.

Support for XV20 and XV15 Formats:

- The module supports HDR10 use-cases for XV20 and XV15 formats. XV20 and XV15 likely refer to specific pixel formats or color representations associated with HDR video.

Resolution Support:

- The module is capable of supporting DCI 4K resolution (4096 x 2160) at 60 frames per second (FPS). This implies that it can handle high-resolution HDR10 video content.

Use Cases:

HDR Content Processing:

- The primary purpose of this module is to process HDR10 content, ensuring that the HDR metadata is preserved and utilized for accurate rendering on compatible displays.

Versatile Output Options:

- The ability to display, record, and stream HDR10 content provides flexibility in how the processed HDR video is presented, stored, and distributed.

Support for Different HDR10 Use-Cases:

- The module accommodates various HDR10 use-cases, including both raw and processed scenarios, allowing for versatility in application deployment.

In summary, the described module is specifically tailored for processing HDR10 content, supporting the entire pipeline from reception to display, recording, and streaming. Its flexibility in handling different HDR10 use-cases and resolutions makes it suitable for a range of applications requiring HDR video capabilities.

LLP2 PS DDR NV12 HDMI Video Capture and HDMI Display:

Key Components and Functionality:

Video and Audio Capture:

- The module captures both video and audio data from an HDMI Rx subsystem implemented in the PL. This could involve receiving video and audio signals from an external HDMI source.

HDMI Display:

- The captured video and audio data can be displayed through the HDMI Tx subsystem implemented in the PL. This allows for the presentation of the content on an HDMI-connected display or monitor.

Ultra-Low Latency Streaming:

- Using Sync IP, the module supports ultra-low latency streaming of live captured video and audio through an Ethernet interface. Sync IP facilitates synchronized communication between different elements of the system to minimize latency.

NV12 Pixel Format:

- The module supports multi-stream video for the NV12 pixel format. NV12 is a YCbCr 4:2:0 image format commonly used in video compression and playback.

Single-Stream Audio:

- The module supports single-stream audio, suggesting that it can handle audio data associated with the captured video.

VCU Encoder and Decoder Operation:

- The Video Codec Unit (VCU) encoder and decoder operate in slice mode. In slice mode, an input frame is divided into multiple slices (8 or 16) horizontally. This allows for efficient encoding and decoding of video data.

Slice-Level Operation:

- The encoder generates a "slice_done" interrupt at the end of each slice, and the decoder starts processing data as soon as one slice is ready in its circular buffer. This slice-level operation helps in achieving better parallelism and reduces the need to wait for complete frame data.

Sync IP for Buffer Synchronization:

- Sync IP is responsible for synchronizing buffers between the Capture DMA (Direct Memory Access) and VCU encoder. It tracks **AXI transaction-level** progress to ensure synchronization at the granularity of AXI transactions.

Buffer Write and Read Order:

- The capture element (FB write DMA) writes video buffers in raster-scan order. Sync IP monitors buffer levels during DMA writes and allows the encoder to read input buffer data immediately if the requested data is already written. It blocks the encoder if the data is not yet available.

Decoder and Display Synchronization:

- On the decoder side, the VCU decoder writes decoded video buffer data into DRAM in **block-raster scan order**, and the display reads data in raster-scan order. Software ensures a phase difference of approximately "~frame_period/2" to prevent display under-run problems.

Use Cases:

Real-Time Video Capture and Display:

- The module is suitable for applications requiring real-time video capture and display with low latency.

Ultra-Low Latency Streaming:

- The capability for ultra-low latency streaming through an Ethernet interface makes this module suitable for applications where minimal delay is critical, such as live broadcasting or interactive video communication.

NV12 Video Format Support:

- Support for the NV12 pixel format makes the module versatile for handling various video compression and playback scenarios.

Synchronized Operation with Encoder and Decoder:

- The design of the module ensures efficient synchronization between the encoder, decoder, and display elements, optimizing the processing flow.

In summary, the "LLP2 PS DDR NV12 HDMI Video Capture and HDMI Display" module provides a comprehensive solution for capturing, processing, and displaying video and audio data with a focus on low-latency streaming and synchronization between different components of the system.

AXI transaction-level

AXI (Advanced eXtensible Interface) is a widely used interconnect standard in digital systems design, particularly in FPGA and ASIC designs. AXI transaction-level progress

refers to the progress of data transfers and communication between different components in a system at the granularity of AXI transactions. AXI transactions represent individual data transfers between a master and a slave component in a system.

Let's break down the concept of AXI transaction-level progress with a real-time example:

Real-Time Example: AXI Transaction in a Memory-Mapped System
Consider a system with a processor (master) communicating with a memory module (slave) over the AXI interface. The processor wants to read data from a specific address in memory.

Initialization:

- The processor initiates an AXI read transaction by asserting the appropriate AXI signals (address, control, etc.).

Address Phase:

- During the address phase of the AXI transaction, the processor sends the target address to the memory module. This represents the location in memory from which data is to be read.

Read Data Phase:

- In response to the address phase, the memory module retrieves the data from the specified address and sends it back to the processor during the data phase of the transaction.

Transaction Completion:

- The AXI transaction is considered complete when the data has been successfully transferred, and any necessary acknowledgments or responses have been exchanged between the master and the slave.

AXI Transaction-Level Progress:
Now, let's relate this to AXI transaction-level progress:

Granularity of Progress:

- AXI transaction-level progress means tracking the progress of individual transactions. In the example, progress is tracked during the address phase and the subsequent data phase of the transaction.

Transaction Success/Failure:

- Progress is determined by the successful completion of each transaction. If the processor successfully reads the data from the memory module without errors, the transaction is considered to have progressed.

Synchronization and Flow Control:

- AXI transaction-level progress involves synchronization and flow control mechanisms to ensure that transactions proceed smoothly. Signals like read and write strobes, acknowledgment signals, and ready/valid handshaking are used to manage the flow of data.

Real-Time Significance:
In a real-time scenario, AXI transaction-level progress is critical for ensuring efficient and reliable communication between different components of a digital system. For example:

High-Performance Systems:

- In high-performance systems, tracking AXI transaction-level progress allows components to work in parallel, optimizing data transfer rates and reducing latency.

Real-Time Processing:

- In real-time applications, timely completion of transactions is essential. AXI transaction-level progress ensures that data is delivered or received within specified time constraints.

Error Detection and Handling:

- Progress monitoring helps detect and handle errors. If a transaction fails to complete successfully, appropriate error-handling mechanisms can be triggered.

In summary, AXI transaction-level progress is about tracking the advancement of individual transactions in a system, and it is crucial for achieving efficient and reliable communication between different components, especially in real-time and high-performance applications.

What is block-raster scan order?

"Block-raster scan order" refers to a specific way in which data is organized or accessed in memory, typically in the context of image or video processing. In block-raster scan order, the data is arranged in blocks, and within each block, a raster scan order is followed. Let's break down the concept and provide a real-time example:

Concept Explanation:

Raster Scan Order:

- In a raster scan, data is read or processed row by row from left to right, and then from top to bottom. This is the order in which pixels are typically displayed on a screen. For example, if you have an image, a raster scan would read the pixels from the top-left corner, moving across each row before progressing to the next row.

Block-Raster Scan Order:

- Block-raster scan order introduces the concept of organizing data into blocks or chunks. Within each block, a raster scan order is followed. This means that data is processed within a block in a raster scan manner, and then the processing continues to the next block.

Real-Time Example: Image Processing:

Consider an image processing scenario where you have an image divided into 4x4 blocks. Each block is processed in a raster scan order, and then the processing moves to the next block. This is an example of block-raster scan order.

Image Organization:

- The image is divided into 4x4 blocks. Each block contains 16 pixels.

Processing Block by Block:

- Starting from the top-left corner, the processing algorithm reads and processes the pixels within the first 4x4 block in a raster scan order.

Moving to the Next Block:

- After processing the first block, the algorithm moves to the next block, which might be immediately to the right of the first block or in a predefined order.

Raster Scan Within Each Block:

- Within each block, the processing follows a raster scan order. For example, it may process pixels from left to right within each row and then move to the next row.

Continuing Through Blocks:

- The algorithm continues processing each block in a block-raster scan order until the entire image is processed.

Real-Time Significance:

Parallelism:

- Block-raster scan order can be efficient for parallel processing, especially if different blocks can be processed simultaneously by different processing units.

Memory Access Optimization:

- Organizing data in blocks can optimize memory access patterns, improving cache locality and reducing memory access times.

Sequential Processing:

- While raster scanning within each block is sequential, the block-raster scan order allows for a degree of parallelism when processing multiple blocks.

In summary, block-raster scan order is a way of organizing and processing data in a structured manner, and the real-time example in image processing demonstrates how it can be applied to efficiently process data in blocks while maintaining a raster scan order within each block.

LLP2 PL DDR HDMI Video Capture and HDMI Display

1. Capture and Display Setup:

- **Capture Source:**
- Video is captured from an HDMI RX subsystem implemented in the Programmable Logic (PL). This could involve receiving video signals from an external HDMI source.
- **Display Destination:**
- The captured video can be displayed through the HDMI TX subsystem also implemented in the PL. This allows for the presentation of the captured video on an HDMI-connected display or monitor.

2. Stream-In and Stream-Out:

- **Ultra-Low Latency Streaming:**
- The module supports streaming of live captured video frames through an Ethernet interface with ultra-low latencies using Sync IP. This implies that the captured video can be transmitted in real-time over a network.

3. Pixel Format Support:

- **Pixel Formats:**
- The module supports multi-stream for NV16 and XV20 pixel formats. NV16 and XV20 are specific pixel formats used to represent color information in the video. NV16 is a YCbCr 4:2:2 format, and XV20 is another video pixel format.

4. DDR Configuration:

- **DDR Allocation:**
- In this design, PL_DDR (Programmable Logic DDR) is used for decoding, and PS_DDR (Processor System DDR) is used for encoding. This allocation is done to ensure that there is enough DDR bandwidth to support high-bandwidth VCU applications that require simultaneous encoder and decoder operations, especially for transcoding at 4K @ 60 FPS.

5. VCU Encoder and Decoder Operation:

- **Slice Mode Operation:**
- Both the VCU encoder and decoder operate in slice mode. In slice mode, an input frame is divided into multiple slices horizontally (8 or 16). This allows for efficient processing of video data.
- **Interrupt Generation:**
- The encoder generates a "slice_done" interrupt at the end of each slice, indicating the completion of that part of the processing. This allows for immediate handling of the generated NAL unit data without waiting for the entire frame to be processed.
- **Decoding Start on Slice Ready:**
- The VCU decoder starts processing data as soon as one slice of data is ready in its circular buffer. This approach reduces the need to wait for the complete frame data before starting the decoding process.

6. Sync IP for Buffer Synchronization:

- **AXI Transaction-Level Tracking:**
- Sync IP performs AXI transaction-level tracking. This means that the synchronization between the producer (Capture DMA) and consumer (VCU encoder) is done at the granularity of AXI transactions rather than the video buffer level. This optimization ensures efficient coordination between the different components.

7. Buffer Write and Read Order:

- **Capture Element (FB Write DMA):**
- Video buffers are written in raster-scan order during the capture element's operation.
- **Monitoring Buffer Level:**
- Sync IP monitors the buffer level while the capture element is writing into DRAM. It allows the encoder to read input buffer data if the requested data is already written by DMA. If not, it blocks the encoder until DMA completes its writes.
- **Decoder Buffer Writing and Reading:**
- On the decoder side, the VCU decoder writes decoded video buffer data into DRAM in block-raster scan order. The display reads data in raster-scan order. To avoid display under-run problems, software ensures a phase difference of "~frame_period/2," ensuring that the decoder is ahead compared to the display.

8. Display Synchronization:

- **Synchronization for Display:**
- Software ensures a phase difference of "~frame_period/2" to avoid display under-run problems. This synchronization ensures that the decoder is ahead compared to the display, preventing issues related to data availability for display.

In summary, the module is designed to efficiently capture, process, and display video with a focus on low-latency streaming, pixel format support, and optimized DDR usage for high-bandwidth VCU applications. The use of slice mode operation and synchronization techniques enhances the efficiency of video processing.

LLP2 PL DDR HLG SDI Video Capture and Display

1. Capture Setup:

- **Capture Source:**
- The module enables the capture of HLG (high dynamic range)/non-HLG video from an SDI RX subsystem implemented in the Programmable Logic (PL). SDI (Serial Digital Interface) is a standard for transmitting video signals.

2. Encoding/Decoding and Transmission:

- **HLG Video Encoding/Decoding:**
- The module supports the encoding/decoding and transmission of HLG video along with backward-compatible standard dynamic range (SDR) for SDI. HLG is a high dynamic range standard used for enhanced contrast and color.
- **Transmission and Recording:**
- The video can be displayed through SDI TX through the PL and recorded in SD cards or USB/SATA drives. This implies that the captured video can be both displayed in real-time and saved for later use.

3. Stream-In and Stream-Out:

- **Ultra-Low Latency Streaming:**
- Similar to the previous module, this module supports streaming of encoded data through an Ethernet interface with ultra-low latencies. It enables real-time transmission or reception of video data over a network.

4. SDI Dynamic Frame Rate Support:

- **Dynamic Frame Rate Support:**
- SDI with dynamic frame rate support is enabled. This feature allows the SDI interface to handle different frame rates dynamically, providing flexibility in video capture and transmission.

5. Audio Support:

- **8-Channel Audio:**
- The module supports single-stream 8-channel audio. This indicates that audio data accompanying the video can be captured, processed, and transmitted.

6. DDR Configuration:

- **DDR Allocation:**
- Similar to the previous module, PL_DDR (Programmable Logic DDR) is used for decoding, and PS_DDR (Processor System DDR) is used for encoding. This allocation aims to optimize DDR bandwidth for high-bandwidth VCU applications requiring simultaneous encoder and decoder operations, especially for transcoding at a maximum resolution of 4K @ 60 FPS.

7. AMD Low Latency Pipeline:

- **VCU Encoder and Decoder in Slice Mode:**
- The module supports AMD's low latency pipeline, where both the VCU encoder and decoder operate in slice mode. Similar to the previous explanation, this involves dividing an input frame into multiple slices horizontally for efficient processing.

8. Buffer Synchronization:

- **SyncIP Element:**

- A hardware synchronization element called SyncIP is responsible for synchronizing buffers between the capture DMA and the VCU encoder, as well as between the VCU decoder and the display element.

9. Buffer Write and Read Order:

- **Capture Element (FB Write DMA):**
- Video buffers are written in raster-scan order during the capture element's operation.
- **Monitoring Buffer Level:**
- SyncIP monitors the buffer level while the capture element is writing into DRAM. It allows the encoder to read input buffer data if the requested data is already written by DMA. If not, it blocks the encoder until DMA completes its writes.
- **Decoder Buffer Writing and Reading:**
- On the decoder side, the VCU decoder writes decoded video buffer data into DRAM in block-raster scan order, and the display reads data in raster-scan order. This approach is designed to prevent display under-run problems.

10. Display Synchronization:

- **Synchronization for Display:**
- Software ensures a phase difference of "~frame_period/2" to avoid display under-run problems. This synchronization ensures that the decoder is ahead compared to the display, preventing issues related to data availability for display.

In summary, this module is designed to capture, process, and display HLG/non-HLG video with dynamic frame rate support and audio capabilities. It emphasizes low-latency streaming and efficient use of DDR bandwidth for high-bandwidth VCU applications. The AMD low latency pipeline, slice mode operation, and buffer synchronization contribute to the optimization of the video processing pipeline.

VCU TRD YUV444 Video Capture and Display

1. Capture Setup:

- **Capture Source:**
- The module enables the capture of YUV444 8-bit and 10-bit video from HDMI-RX. YUV444 is a color space representation that stores the brightness and color information for each pixel without any subsampling.

2. Display and Recording:

- **Display Destination:**
- The captured video can be displayed on HDMI-TX or DisplayPort. This means that the captured video can be shown on an external display device.
- **Recording:**
- Additionally, the captured video can be recorded on SD cards or USB/SATA drives. This allows for the storage of the captured video for later use.

3. Stream-In and Stream-Out:

- **Ultra-Low Latency Streaming:**
- Similar to the previous modules, this module supports streaming of encoded data through an Ethernet interface with ultra-low latencies. It enables real-time transmission or reception of video data over a network.

4. Supported Formats:

- **YUV444 Formats:**
- The module supports up to single-stream 4kp30 YU24 and X403 formats. These are specific YUV444 formats with different resolutions and bit depths.

5. YUV444 Processing:

- **Hardware Limitation:**
- It's mentioned that the AMD Zynq™ UltraScale+™ VCU hardware does not inherently support YUV444 encode/decode processing. The VCU hardware supports YUV 4:0:0, 4:2:0, and 4:2:2 sub-sampling modes.
- **Handling YUV444:**
- However, the feature enables encoding YUV444P (single planar) video frames using the VCU. This involves reading them as YUV4:0:0 of width x 3*height buffer at the encoder side.
- **Decoder Output Handling:**
- At the decoder side, the VCU decoder output reads them as width x 3*height YUV4:0:0 raw video frame. However, the display treats the buffer as a YUV444 planar buffer. This involves handling YUV444 format in the decoding and display stages.

6. Software Enhancements:

- **Software Frameworks and Drivers:**
- The feature includes enhancing FB_RD, FB_WR IP/drivers, and V4l2 and DRM frameworks to support the YUV444 planar buffer format. This indicates that both the read and write components and the associated software frameworks are modified to handle YUV444 format.

7. Additional Documentation:

- **Zynq UltraScale+ MPSoC VCU TRD Wiki:**
- The Zynq UltraScale+ MPSoC VCU TRD wiki for 2023.1 provides additional content, including prerequisites for building and running the reference designs, instructions for running the pre-built SD card image on the evaluation board, and detailed step-by-step design and tool flow tutorials for each design module.

In summary, this module focuses on capturing YUV444 video from HDMI-RX, displaying it on HDMI-TX or DisplayPort, recording it on storage devices, and supporting real-time streaming. It addresses the hardware limitation related to YUV444 processing by implementing specific handling methods at the encoder and decoder stages, along with necessary software enhancements. The additional documentation in the wiki provides further guidance for users.

Design Components

The reference design zip files can be downloaded from the Zynq UltraScale+ MPSoC

ZCU106 Evaluation Kit Documentation website. The file contains the following components

grouped by APU or PL.

Certainly, let's break down the components included in the reference design for the Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit:

APU Components:

vcu_apm_lib:

- *Description:* This is a library that provides an interface to query read and write throughput of the VCU (Video Codec Unit) encoder/decoder. It likely includes functions and routines that allow the system to gather performance metrics from the VCU.

vcu_gst_lib:

- *Description:* This is an interface library responsible for managing the video/audio-video capture, processing, and display pipelines. It utilizes various frameworks such as GStreamer, V4L2 (Video for Linux 2), ALSA (Advanced Linux Sound Architecture), and DRM (Direct Rendering Manager). The library likely provides abstraction and functions to control and coordinate these components.

petalinux_bsp:

- *Description:* This is a PetaLinux board support package (BSP) specifically tailored to build a pre-configured SMP (Symmetric Multi-Processing) Linux image for the APU. It includes essential components like the First Stage Boot Loader (FSBL), Arm Trusted Firmware (ATF), U-Boot, Linux kernel, Device tree, PMU firmware, and the Root file system (rootfs).

vcu_qt:

- *Description:* This is an application that utilizes the vcu_gst_lib, vcu_apm_lib, and vcu_video_lib libraries. It provides a graphical user interface (GUI) to control and visualize various parameters of the design. It's mentioned that the GUI is supported only on DP (DisplayPort).

vcu_video_lib:

- *Description:* This library is responsible for configuring various video pipelines within the design. It likely includes functions to set up and manage video capture, processing, and display configurations.

vcu_gst_app:

- *Description:* This is a command-line application that uses the vcu_gst_lib, vcu_apm_lib, and vcu_video_lib libraries. It allows users to configure and run the capture, display, record, stream-in, and stream-out pipelines through the command line. This provides a more scriptable and automated way to interact with the video processing system.

PL Components:

AMD Vivado™ IP Integrator Design:

- *Description:* This component is an IP integrator design created using Vivado™. It integrates the capture, processing (encode/decode), and display pipeline components in the Programmable Logic (PL). This design likely includes blocks or modules representing the video processing stages implemented in FPGA fabric.

In summary, the APU components include libraries, BSP for Linux image generation, a GUI application, and command-line tools for interacting with the video processing system. On the

PL side, there's the Vivado™ IP Integrator Design that represents the hardware implementation of the video processing pipeline. These components collectively form a comprehensive solution for video processing on the ZCU106 Evaluation Kit.

APU Software Platform

Introduction

This chapter describes the application processing unit (APU) Linux software platform, which is further subdivided into a middleware layer, an operating system (OS) layer, and an application stack (see This Figure). The two layers are examined in conjunction because they interact closely for most Linux subsystems. These layers are further grouped by vertical domains which reflect the organization of this chapter:

- Video
- Audio
- Display
- Graphics
- Accelerator

Certainly, let's provide an example for each of the vertical domains within the APU software platform, including Video, Audio, Display, Graphics, and Accelerator:

1. Video:
- **Middleware Layer:** GStreamer
- **Operating System Layer:** Linux Kernel, Video4Linux (V4L2)
- **Application Stack:** Video Player Application

Example Scenario: A video player application is developed using GStreamer, a powerful multimedia framework. The Linux kernel provides the necessary video drivers, and V4L2 is used for capturing and processing video streams. The video player application utilizes GStreamer to decode, process, and display video content.

2. Audio:
- **Middleware Layer:** Advanced Linux Sound Architecture (ALSA)
- **Operating System Layer:** Linux Kernel
- **Application Stack:** Audio Playback Application

Example Scenario: An audio playback application is developed using ALSA, which is the standard audio architecture for Linux. The Linux kernel provides audio drivers, and the application uses ALSA to interface with the audio hardware, enabling the playback of audio files.

3. Display:
- **Middleware Layer:** Direct Rendering Manager (DRM), Kernel Mode Setting (KMS)
- **Operating System Layer:** Linux Kernel
- **Application Stack:** Display Manager

Example Scenario: A display manager application is implemented using DRM and KMS. The Linux kernel, with DRM support, provides the necessary display drivers. The display manager application controls the display configuration, resolution, and multiple screen management.

4. Graphics:

- **Middleware Layer:** Qt, OpenGL ES
- **Operating System Layer:** Linux Kernel
- **Application Stack:** GUI-Based Graphics Application

Example Scenario: A graphical user interface (GUI) application is developed using Qt and OpenGL ES. The Linux kernel provides graphics drivers, and the GUI application leverages Qt for creating an interactive and visually appealing user interface.

5. Accelerator:

- **Middleware Layer:** OpenCL, CUDA
- **Operating System Layer:** Linux Kernel
- **Application Stack:** Hardware Accelerated Application

Example Scenario: An application is designed to leverage hardware acceleration using OpenCL or CUDA. The Linux kernel provides the necessary drivers for the accelerated hardware, and the application is programmed to offload specific tasks to the hardware accelerator for improved performance.

These examples illustrate how each vertical domain within the APU software platform is structured, highlighting the middleware, operating system, and application components involved in various use cases.

1. Middleware Layer:

- **Definition:** The middleware layer comprises software components that act as intermediaries, providing services and functionalities to applications. These components abstract underlying complexities and facilitate communication between the OS and applications.

Example Middleware Components:

- **GStreamer (Video):** GStreamer is a multimedia framework that allows the construction of complex multimedia pipelines for video processing, streaming, and playback.
- **Advanced Linux Sound Architecture (ALSA) (Audio):** ALSA provides audio support in Linux, offering an API for soundcard drivers and user-space applications.
- **Direct Rendering Manager (DRM) and Kernel Mode Setting (KMS) (Display):** DRM and KMS are components that manage graphics hardware and enable user-space applications to interact with display devices.
- **Qt and OpenGL ES (Graphics):** Qt is a cross-platform application framework, and OpenGL ES is a subset of the OpenGL graphics API optimized for embedded systems.

- **OpenCL, CUDA (Accelerator):** OpenCL and CUDA are frameworks for programming heterogeneous systems with accelerators like GPUs, allowing applications to leverage hardware acceleration.

2. Operating System (OS) Layer:

- **Definition:** The OS layer includes the Linux kernel, which acts as the core of the operating system. It manages hardware resources, provides system services, and serves as an interface between hardware and software.

Example OS Components:

- **Linux Kernel:** The Linux kernel is responsible for managing CPU, memory, devices, and system calls. It provides essential services to applications and ensures hardware abstraction.

3. Application Stack:

- **Definition:** The application stack represents the actual user-facing software applications that run on top of the middleware and OS layers. Applications leverage the services provided by middleware and interact with the OS for hardware access.

Example Applications:

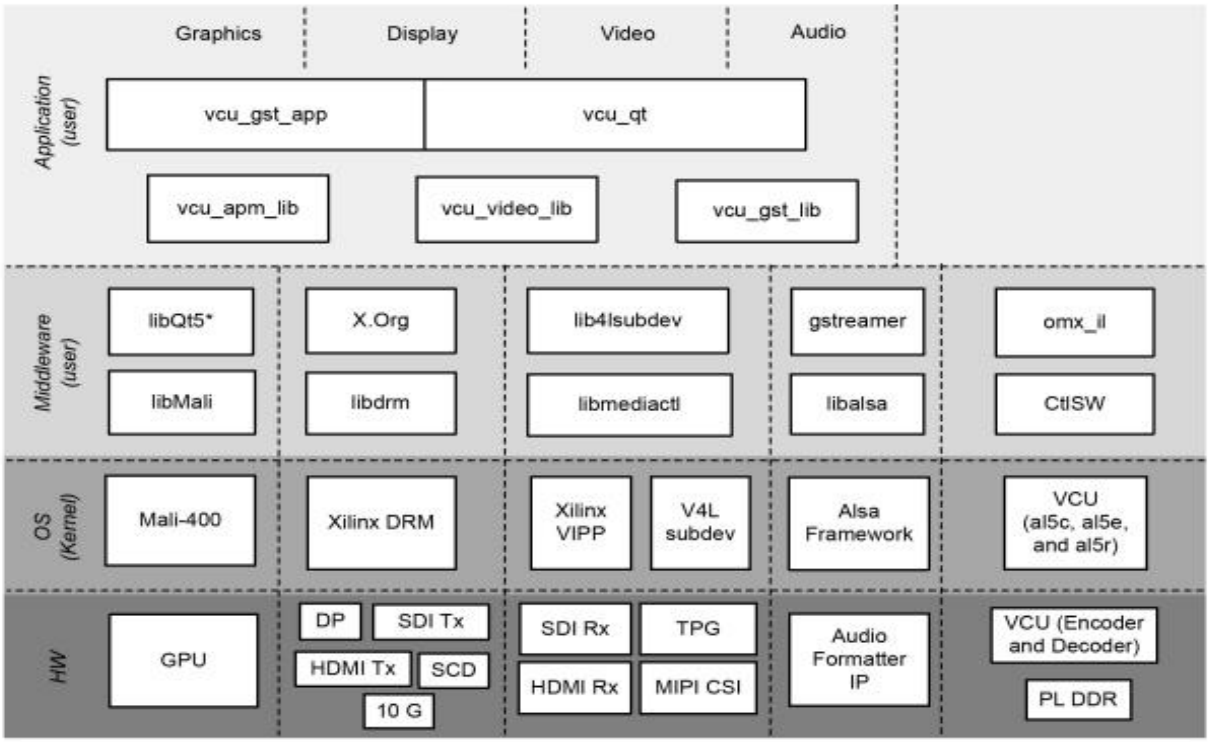
- **Video Player Application (Video):** A user-facing application that utilizes GStreamer to play, process, and display video content.
- **Audio Playback Application (Audio):** An application that uses ALSA to play audio files, providing a user interface for controlling audio playback.
- **Display Manager (Display):** An application responsible for managing display configurations, resolutions, and multiple screens, utilizing DRM and KMS.
- **GUI-Based Graphics Application (Graphics):** A graphical user interface (GUI) application created using Qt and OpenGL ES, offering an interactive user experience.
- **Hardware Accelerated Application (Accelerator):** An application that offloads specific tasks to hardware accelerators using OpenCL or CUDA for improved performance.

This breakdown illustrates how the APU software platform is organized into layers and vertical domains, showcasing the middleware components, the Linux kernel in the OS layer, and the diverse set of user-facing applications in the application stack.

Software Architecture

This Figure shows the APU Linux software platform.

APU Linux Software Platform



Middleware Layer:

- **Definition:** The middleware layer is a horizontal layer implemented in user-space, serving as an intermediary between the application layer and the OS layer. It abstracts complexities and provides services that facilitate communication with the underlying kernel frameworks.

Middleware Functionality:

- **Interfaces with the Application Layer:**
- The middleware layer acts as a bridge between applications and the lower-level functionalities provided by the OS. It defines interfaces and APIs that applications can use to access specific services.
- **Provides Access to Kernel Frameworks:**
- It interacts with various kernel frameworks and APIs to leverage the functionality exposed by the operating system. This allows applications to access hardware resources and services through a higher-level interface.

OS Layer:

- **Definition:** The OS layer is a horizontal layer implemented in the kernel-space. It forms the core of the operating system, managing hardware resources, providing system services, and defining a stable API for user-space applications.

OS Functionality:

- **Provides a Stable, Well-Defined API to User-Space:**
- The OS layer defines a set of application programming interfaces (APIs) that applications in user-space can use. These APIs provide a stable and standardized way for applications to interact with the underlying system.
- **Includes Device Drivers and Kernel Frameworks (Subsystems):**
- Device drivers are components that allow the OS to communicate with hardware devices. Kernel frameworks or subsystems are organized sets of functionalities that

the OS provides, often related to specific aspects like file systems, networking, or, in this case, video processing.

- **Accesses the Hardware:**
- The OS layer directly manages hardware resources, including CPUs, memory, and peripherals. It acts as an interface between applications and the hardware, ensuring proper resource allocation and access control.

Video Domain:

- **Definition:** The Video domain involves modeling and controlling video capture pipelines on Linux systems, requiring the collaboration of multiple kernel frameworks and APIs. In this context, Video4Linux (V4L2) is a key framework.

Video Domain Components (V4L2):

- **Functionality:**
- **Video4Linux (V4L2):** This is a framework that facilitates video capture and output on Linux systems. It provides a set of APIs for applications to work with video devices, such as cameras or video capture cards.
- **Components:** While the overall solution is referred to as V4L2, individual components, including specific kernel frameworks and APIs, contribute to the complete functionality needed for video capture pipelines.

This structure allows applications to interact with video devices in a standardized way, abstracting the complexities of the underlying hardware and providing a consistent interface for video-related operations.

Driver Architecture

V4L2 Driver Stack Overview:

1. Video Pipeline Driver:

- **Function:**
- The video pipeline driver is responsible for managing the entire video processing pipeline. It loads the necessary subdevice drivers and registers the device nodes based on the video pipeline configuration specified in the device tree.

2. Device Nodes Exposed to User Space:

- The V4L2 framework exposes different types of device nodes to user space, allowing control over certain aspects of the pipeline:
- **Media Device Node: /dev/media*** (e.g., **/dev/media1**, **/dev/media2**, etc.).
- **Video Device Node: /dev/video*** (e.g., **/dev/video1**, **/dev/video2**, etc.).
- **V4L Subdevice Node: /dev/v4l-subdev*** (e.g., **/dev/v4l-subdev1**, **/dev/v4l-subdev2**, etc.).

Data Flow within Software:

1. V4L2 Source Driver (Capture Device):

- **Action:**
- Allocates a frame buffer for the capture device.

2. V4L2 Framework:

- **Action:**
- Imports/exports the DMA_BUF file descriptor (FD) to the next GStreamer element.

3. Encoder:

- **Steps:**
- **Reads Source Buffer:**
- The encoder reads the source buffer from the capture device.
- **Encodes Frame:**
- Encodes the source buffer.
- **Writes Encoded Bitstream:**
- Writes the encoded bitstream to a bitstream buffer.

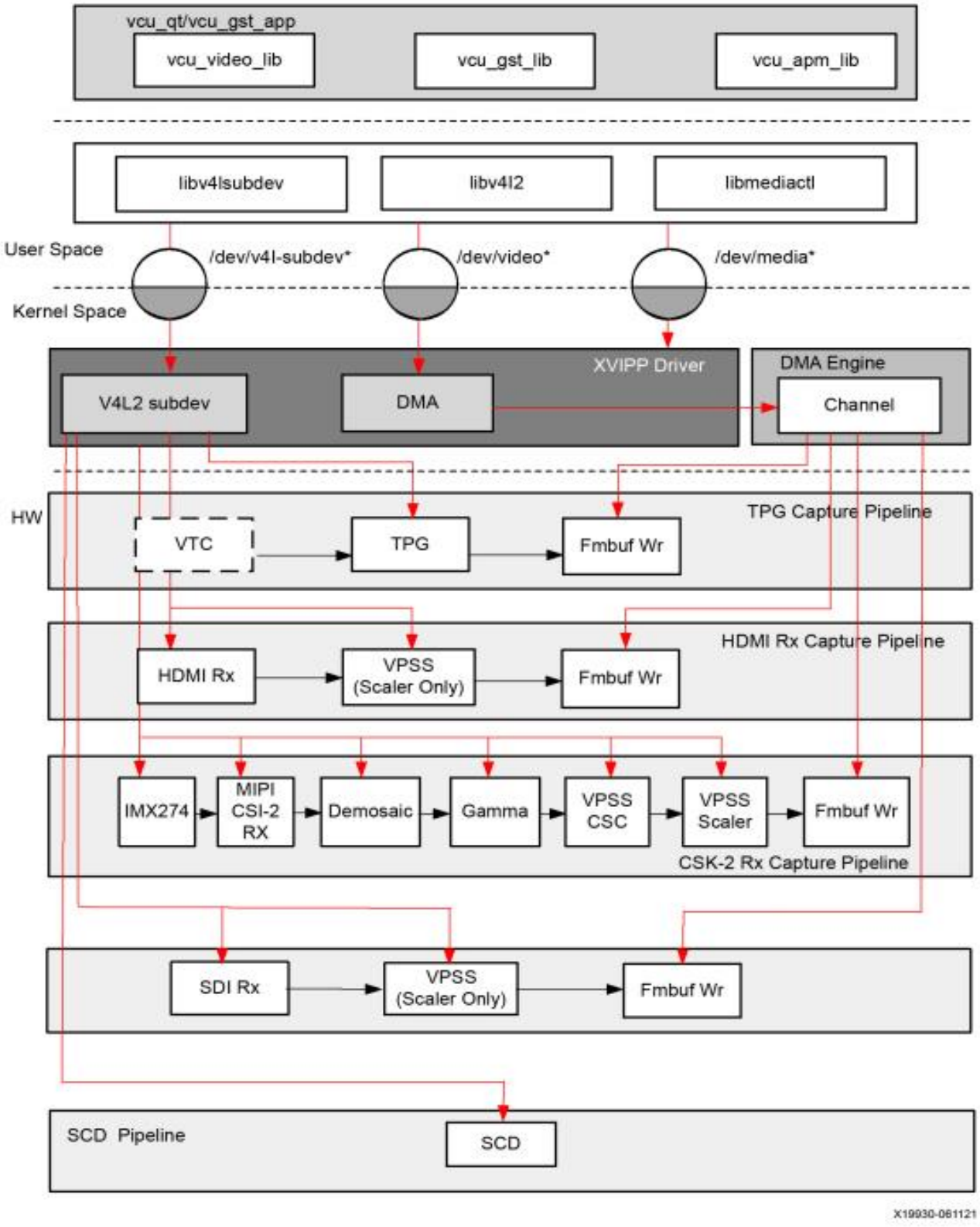
4. Decoder:

- **Steps:**
- **Allocates Decoded Frame Buffer:**
- The decoder allocates a frame buffer for the decoded output.
- **Reads Bitstream Buffer:**
- Reads the bitstream buffer produced by the encoder.
- **Writes Decoded Frame:**
- **Writes the decoded frame buffer into memory.**

5. Display (DRM):

- **Action:**
- The decoder shares the decoded frame buffer using the DMA_BUF framework with the DRM (Direct Rendering Manager) display device.

This data flow illustrates the sequence of operations within the software for video processing, involving capture, encoding, decoding, and display. The DMA_BUF framework is utilized for efficient sharing of buffers between different components in the pipeline.



Media Framework

The provided information describes the concept of a media framework, which is commonly used in video processing systems to model and configure video pipelines. Here's an explanation of the key terms and concepts:

Purpose: The main goal of the media framework is to discover the device topology of a video pipeline and configure it at runtime.
Modeling: Video pipelines are modeled as an oriented graph of building blocks called entities connected through pads.
Entities and Pads:

- Entity: A basic media hardware building block, representing various components like physical hardware devices (e.g., image sensors), logical hardware devices (e.g., soft IP cores inside the PL), DMA channels, or physical connectors.
- Pad: A connection endpoint on an entity through which the entity can interact with other entities. Data flows from an entity's output pad to one or more input pads on other entities.

Link:

- A point-to-point oriented connection between two pads, either on the same entity or different entities. Data flows from a source pad to a sink pad.

Media Device Node:

- A node created to allow user space applications to configure the video pipeline and its subdevices. Configuration is typically done through libraries such as libmediactl and libv4l2subdev.

Media Controller API:

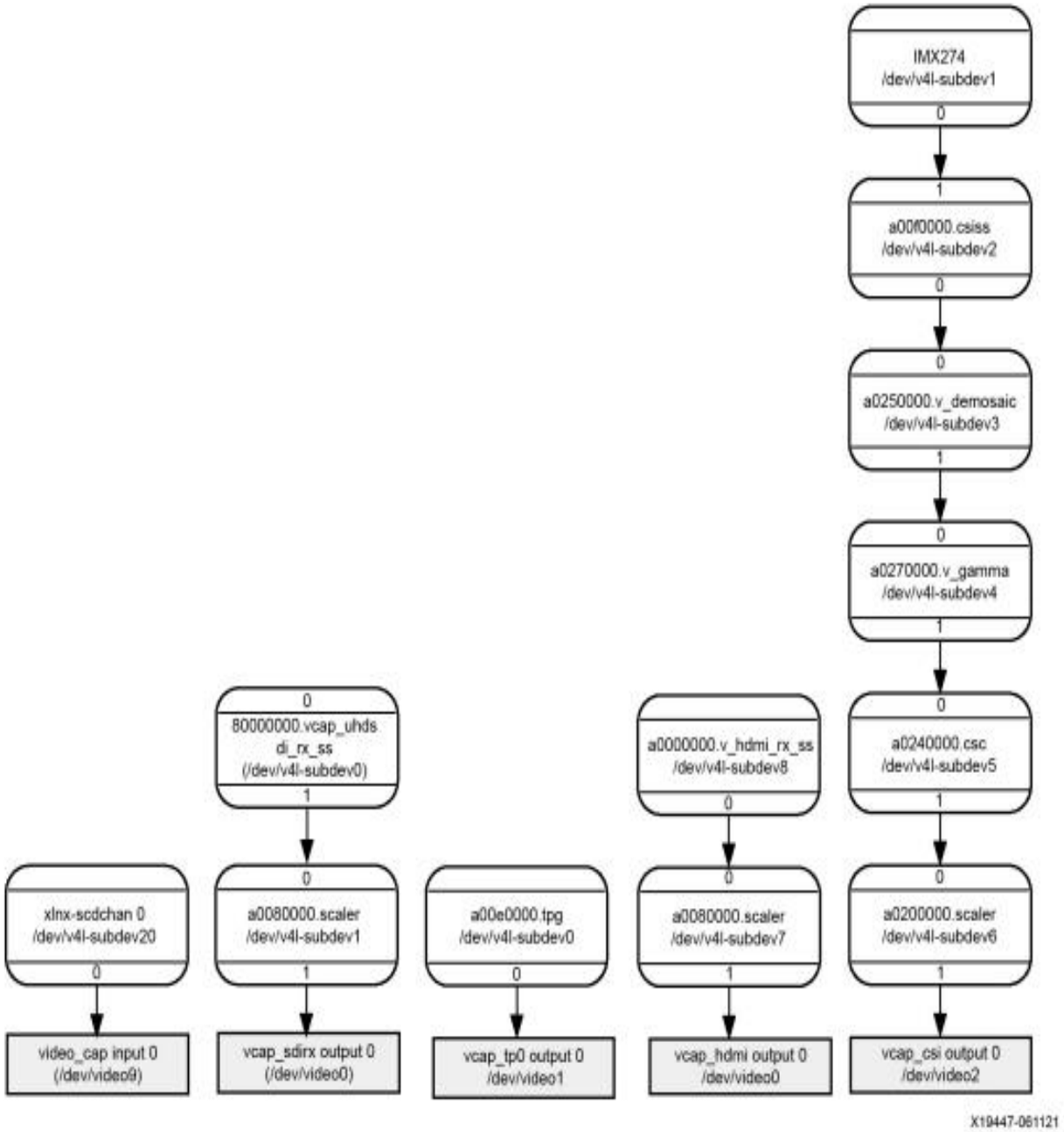
- Provides functionality for:
- Enumerating entities, pads, and links in the media graph.
- Configuring pads, including setting media bus format and dimensions (width/height).
- Configuring links, enabling/disabling, and validating formats.

Media Graph Example:

- The provided figure shows a media graph for SDI-RX, TPG, HDMI RX, and CSI RX video capture pipelines. It visualizes entities, pads, and links. TPG (Test Pattern Generator) is shown as a subdevice with a control interface address. The edges represent pads, and arrows show active links. Grey boxes represent video nodes corresponding to Frame Buffer Write channels.

In summary, the media framework provides a structured approach to model, discover, and configure the components of a video pipeline in a flexible and extensible manner. The Media Controller API facilitates interaction and configuration of these components.

Figure: Video Capture Media Pipelines from Left: SDI, TPG, HDMI RX, and CSI RX



-

Graphics with Qt:

- **Qt Framework:** Qt is a comprehensive development framework designed for creating applications and user interfaces across various platforms, including desktop, embedded, and mobile.
- **C++ with Extensions:** Qt uses standard C++ with extensions, including signals and slots, which simplify event handling. This facilitates the development of both GUI and server applications.

Display and Graphics Frameworks:

- **Kernel and User-Space Frameworks:** In Linux, display and graphics frameworks are complex and intertwined, with distinct layers and standards.
- **DRM/KMS:** Refers to the split framework for display and graphics in the Linux kernel. DRM (Direct Rendering Manager) manages GPU functions, and KMS (Kernel Mode Setting) handles display. Both are accessible from user space through a single device node.

Accelerator - Video Codec Unit (VCU):

- **VCU Core:** The Video Codec Unit core supports multi-standard video encoding and decoding, including H.264 and H.265 standards.
- **Software Stack:** The VCU software stack includes a custom kernel module and a user-space library known as Control Software (CtrlSW).
- **Integration Layers:** OpenMAX (OMX) integration layer is integrated on top of CtrlSW. GStreamer is used to integrate OMX IL component and other multimedia elements.
- **OpenMAX (OMX):** OpenMAX is a cross-platform API providing streaming media codec and application portability by enabling accelerated multimedia components.
- **GStreamer:** GStreamer is a cross-platform/open-source multimedia framework. It facilitates plug-ins, data flow, media type handling, and negotiation. Developers can work at three levels: CtrlSW, OMX IL, and GStreamer.

In summary, the information outlines the use of Qt for graphics development, the complexities of Linux display and graphics frameworks, and the architecture of the Video Codec Unit (VCU) software stack, including the integration of OpenMAX and GStreamer for multimedia processing. Developers have the flexibility to work at different levels of the software stack for VCU, depending on their requirements.

Software Stack

The software stack for the APU (Application Processing Unit) Linux multimedia system is divided into two main layers: the application layer and the platform layer. Here's an explanation of each:

Application Layer:

- **User-Space Implementation:** The application layer is entirely implemented in the Linux user-space. This means that the components in this layer run in the user-space environment of the Linux operating system.
- **Focus on User-Space Operation:** This layer is primarily concerned with functionalities that operate at the user level, providing features and services for applications.

Platform Layer:

- **Middleware and OS Components:** The platform layer contains both middleware (user-space libraries) and operating system (OS) components (kernel-space drivers).
- **Middleware:** Refers to user-space libraries or frameworks that provide higher-level functionalities to applications.

- **OS Components (Kernel-Space):** These are components that operate in the kernel-space of the operating system, interacting more closely with the hardware.

Software Stack Overview:

- **Simplified Stack Overview:** The provided figure illustrates a simplified version of the Linux software stack for the APU multimedia system.
- **Focus on Application Layer:** The chapter primarily focuses on the application layer, which is implemented in the user-space environment.

User Application and GStreamer:

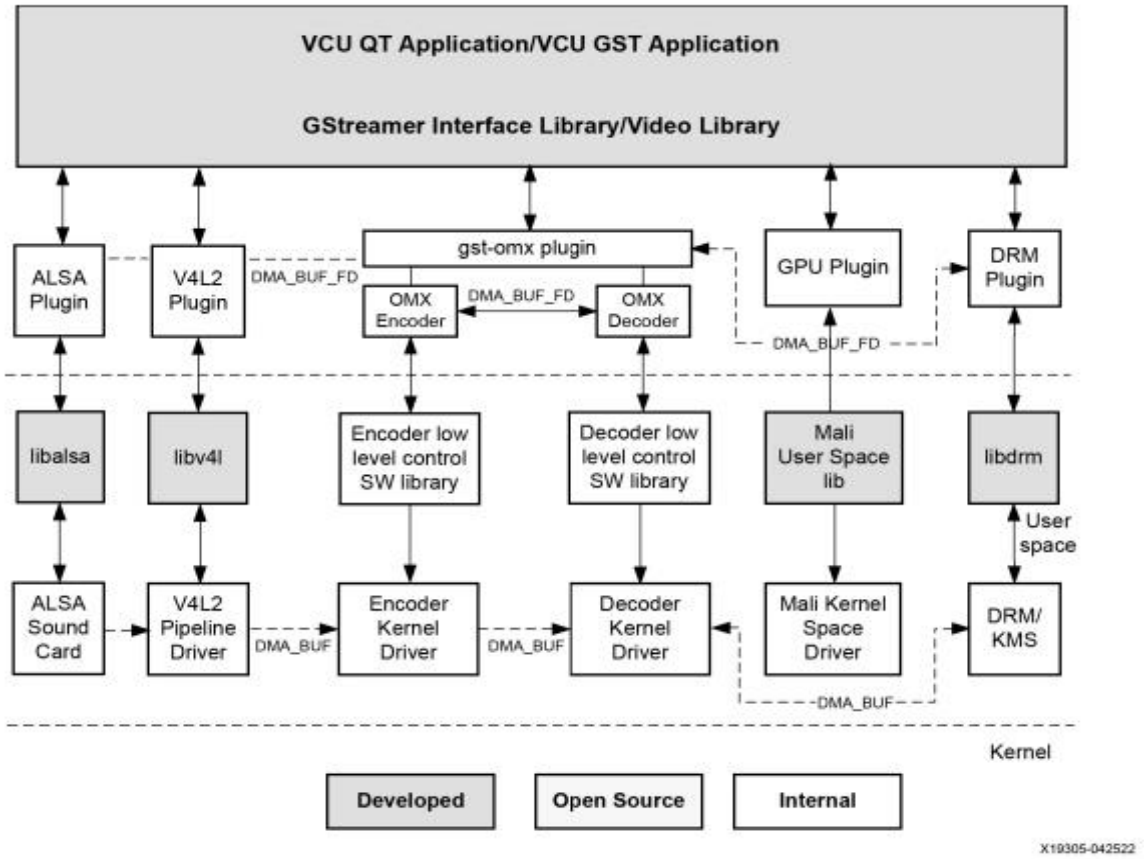
- **User Application:** A user application based on GStreamer is mentioned as a demonstration of the TRD (Targeted Reference Design) features.
- **GStreamer:** GStreamer is a multimedia framework used for constructing graphs of media-handling components. It facilitates the creation of multimedia applications.

Table: Software Stack Components:

- **Software Components Description:** The table likely provides a breakdown of various software components within the APU Linux multimedia software stack, offering details about their roles and functionalities.

In summary, the APU Linux multimedia software stack is organized into layers, with the application layer focusing on user-space operations and the platform layer containing middleware and OS components. The GStreamer framework is highlighted as a tool for building multimedia applications in the user-space. The table likely provides a detailed overview of specific software components in the stack.

Figure: TRD Software Stack



Software Stack Components

Component	Description
Kernel drivers	This layer contains the kernel drivers for HDMI, Test Pattern Generator (TPG), IMX274 sensor driver, MIPI CSI-2 RX Subsystem, AMD Video Demosaic, AMD Video Gamma LUT, VPSS Color Space Converter (CSC), AMD Video Processing Subsystem (VPSS Only configuration, 2X configuration), HDMI TX Subsystem, HDMI RX Subsystem, AMD Video Pipeline (XVIPP), Mixer, VCU, AMD PL sound card, AMD Audio Formatter, DisplayPort controller, Sync IP, and the Mali GPU.
User space libraries	User space libraries include the media and v4l2 lib for the video pipeline, GStreamer libraries, lib_decode libraries for VCU,

	libdrm for the DRM device, libalsa, and Mali user-space libraries for the GPU.
OpenMAX v1.1.2	The OpenMAX integration layer (IL) components for encoder and decoder provides an abstraction for VCU to a user space media framework like GStreamer (a complete, cross-platform solution to play, record, convert, and stream audio and video) [Ref 5] . It implements a standard application programming interface (API) for the user space media framework.
GStreamer framework	GStreamer is the cross-platform/open source multimedia framework, and provides the infrastructure to integrate multiple multimedia components and create pipelines. Various GStreamer plug-ins are used for input, filter, and display components.

The TRD (Targeted Reference Design) application, named "vcu_qt," is a multi-threaded Linux application designed for the VCU (Video Codec Unit) TRD. It performs several key tasks and consists of various components developed specifically for the TRD. Here's an explanation of its main tasks and components:

Main Tasks of the Application:

- **Displays Unprocessed Video:** The application is responsible for displaying unprocessed video from one or more sources. This involves presenting the raw video data to the user interface.
- **Applies Processing Function:** The application applies a processing function, which may involve video encoding and decoding. These functions are essential for manipulating video data as it flows through the pipeline.
- **Provides GUI for User Input:** A Graphical User Interface (GUI) is integrated into the application to allow users to interact with and control various aspects of the video pipeline. The GUI provides a user-friendly way to input commands or settings.
- **Interfaces with Lower-Level Layers:** The application interfaces with lower-level layers in the software stack to control video pipeline parameters and manage the flow of video data.

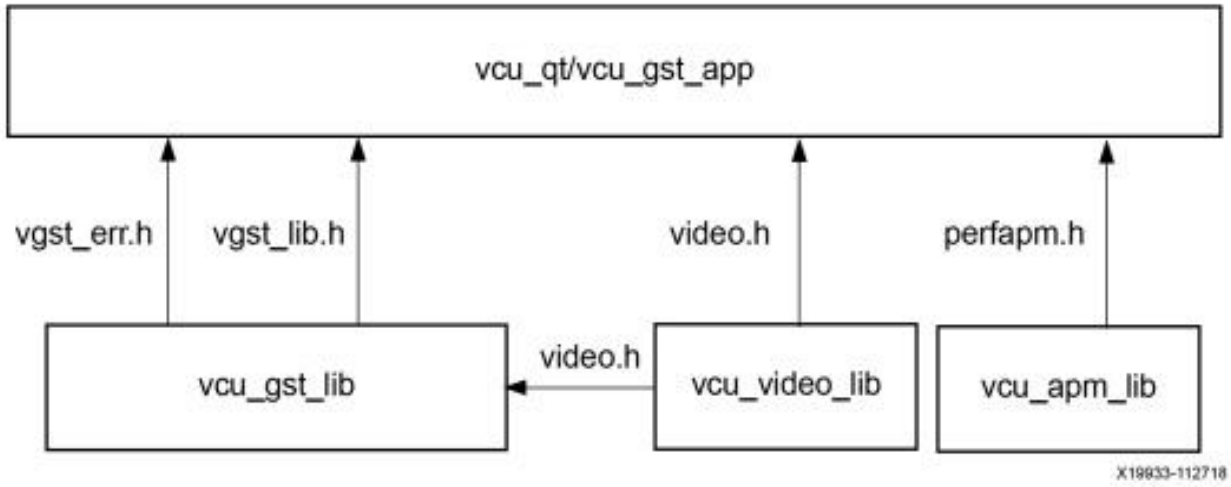
Components of the Application:

- **GUI Application (vcu_qt):** This component is responsible for creating and managing the graphical user interface. It allows users to interact with the application, providing a visual representation of video data and control options.
- **GStreamer Interface Library (vcu_gst_lib):** This library serves as an interface between the application and the GStreamer multimedia framework. GStreamer is used for constructing multimedia pipelines, and this library helps manage the integration of GStreamer functionalities into the application.

- **Video Library (vcu_video_lib):** The video library is likely designed to configure and control various aspects of the video pipeline within the application. It may provide functionalities related to video processing, capture, and display.
- **AXI Performance Monitor (APM) Library (vcu_apm_lib):** This library interfaces with the AXI Performance Monitor, which is a hardware monitoring tool. It allows the application to gather performance-related data and monitor the behavior of the video processing units.
- **GStreamer Command Line Application (vcu_gst_app):** This component is a command line application that uses GStreamer functionalities. It provides a way to configure and run the capture, display, record, stream-in, and stream-out pipelines through the command line.

In summary, the vcu_qt application is a comprehensive solution for managing video processing tasks, providing a user interface, and interacting with lower-level software components and hardware resources. Its various components contribute to the overall functionality of the VCU TRD.

Video Application Interfaces



GUI Application (vcu_qt)

The vcu_qt application is a multi-threaded Linux application that uses the Qt graphics toolkit to render a GUI. The GUI provides control knobs for user input and a display area to show the captured video stream. The GUI shown in This Figure contains the following control elements displayed on top of the video output area:

• Control bar (top)

• Video info panel (top-right)

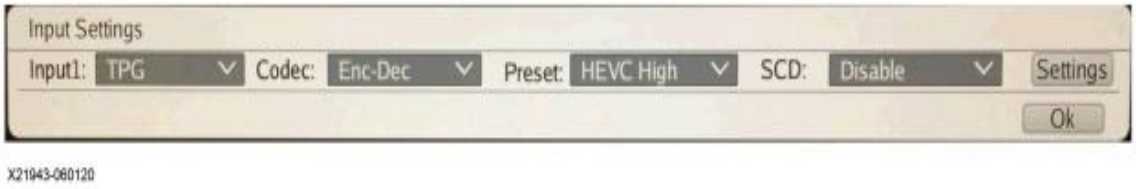
• System performance panels (bottom)

Control Elements Displayed on Top of the Video Output Area

Number of Inputs

This determines the number of active video sources. In the current version of the TRD, a maximum of four sources are supported (the default value is one). See This Figure for input settings.

Input Settings



Input Options

The following 4K/1080p video sources (3840 x 2160) are available:

• HDMI: Implemented in the PL

• File: TS/MP4/MKV streams reside in the SD card or USB/SATA drives

• Test Pattern Generator (TPG): Implemented in the PL

• CSI: Implemented in the PL (o ption: MIPI: MIPI CSI, model LI-IMX274 MIPI-FMC v1.1)

• SDI: Implemented in the PL

• Stream In: Stream from network or Internet

Codec

• **Enc** —This option selects encoder in the pipeline.

• **Enc-Dec** —This option selects encode and decode in the pipeline.

• **Pass-through** —This option selects displaying the raw video source.

SCD

• **Enable** — To enable SCD

• **Disable** — To disable SCD

Preset

There are six predefined presets. If you edit any control options, preset the mode switches to **Custom** . See Table: Predefined Preset Descriptions .

Predefined Preset Descriptions

Preset	Description
AVC Low (1)	Encoder type = H264, bitrate = 10 Mb/s
AVC Medium (1)	Encoder type = H264, bitrate = 30 Mb/s
AVC High (1)	Encoder type = H264, bitrate = 60 Mb/s
HEVC Low (1)	Encoder type = H265, bitrate = 10 Mb/s
HEVC Medium (1)	Encoder type = H265, bitrate = 30 Mb/s
HEVC High (1)	Encoder type = H265, bitrate = 60 Mb/s
Custom	User-specific options
Notes: 1. The following settings are common for these options: Profile = **High** for H264 and **Main** for H265, Rate control = **CBR** , Filler data = **true** , QP = **auto** , L2 cache = **true** , Latency mode = **Normal** , Low bandwidth = **false** , GoP (Group of Pictures) mode = **Basic** , B-frame = **0** , Slice = **8,** and GoP length = **60** (see This Figure).	

Output Options

This option allows you to select the sink for the pipeline. Supported output sink types are:

• DisplayPort

• Record

• Stream Out

For the **DisplayPort** output option, either the **enc-dec Codec** or **Pass-through** option can be selected.

For the **Record** and **Stream Out** output options, only **Encode** can be selected in **Codec** .

Encoder Parameter Panel Settings

Encoder Parameter	Setting
L2 Cache	Enable or disable L2cache buffer in the encoding process.
Latency Mode	Encoder latency mode. It can be normal or sub_frame mode.
Bitrate	Encoding bitrate. In digital multimedia, bitrate often refers to the number of bits used per unit of playback time to represent a continuous medium such as audio or video after source coding (data compression).
Filler Data	Filler data network abstraction layer (NAL) units for CBR rate control. It can be enabled or disabled. Applies to CBR mode only.
GoP Mode	Group of Pictures mode. It can be Basic, low_delay_p, or low_delay_b.
Low Bandwidth	If enabled, decreases the vertical search range used for P-frame motion estimation to reduce the bandwidth.
GoP Length	In video coding, a group of pictures, or GoP structure, specifies the order in which intra- and inter-frames are arranged. And GoP Length is a length between two intra-frames. The GoP is a collection of successive pictures within a coded video stream. Each coded video stream consists of successive GoPs from which the visible frames are generated. Its range is from 1– 1000. The GoP length must be a multiple of B-Frames+1.
Slice	Number of slices produced for each frame. Each slice contains one or more complete macroblock/CTU row(s). Slices are distributed over the frame as regularly as possible. If slice size is also defined, more slices can be produced to fit the slice size requirement. Range: • 4-22 4K resolution with HEVC codec • 4-32 4K resolution with AVC codec • 4-32 1080p resolution with HEVC codec • 4-32 1080p resolution with AVC codec
QP	Quantization in an encoder is controlled by a quantization parameter. It specifies how to generate the QP per coding unit (CU). Two modes are supported:

	• Uniform: All CUs of the slice use the same QP • Auto: The QP is chosen according to the CU content using a pre-programmed lookup table.
Rate Control	Selects the way the bit rate is controlled: CBR: Use *constant bit rate* control. VBR: Use *variable bit rate* control. LOW_LATENCY: Use variable bit rate for low latency application.
B-frame	Short for bidirectional frame or bidirectional predictive frame, a video compression method used by the MPEG standard. The setting ranges from 0 to 4.
Profile	The standard defines a sets of capabilities, which are referred to as profiles, targeting specific classes of applications. These are declared as a profile code (profile_idc) and a set of constraints applied in the encoder. This allows a decoder to recognize the requirements to decode that specific stream. H264 supports Baseline, Main, and High profile. In H265, only the Main profile is supported.
Encoder	This can be either H264 or H265.

Certainly! Let's go through the parameters and their explanations, along with real-time examples where applicable:

L2 Cache:

- *Explanation:* Enables or disables L2 cache buffer in the encoding process.
- *Example:* If enabled, it utilizes L2 cache for improved performance in video encoding.

Latency Mode:

- *Explanation:* Encoder latency mode. It can be normal or sub_frame mode.
- *Example:* In "sub_frame" mode, the encoder may introduce lower latency suitable for real-time applications.

Bitrate:

- *Explanation:* Encoding bitrate, representing the number of bits used per unit of playback time.
- *Example:* Setting a bitrate of 5000 kbps means the encoder aims to use 5000 kilobits per second for the encoded video.

Filler Data:

- *Explanation:* Filler data network abstraction layer (NAL) units for CBR rate control. It can be enabled or disabled.
- *Example:* Enabling filler data in CBR mode may help maintain a constant bitrate.

GoP Mode:

- *Explanation:* Group of Pictures mode. It can be Basic, low_delay_p, or low_delay_b.
- *Example:* Choosing "low_delay_b" can improve the efficiency of B-frame usage in the Group of Pictures.

Low Bandwidth:

- *Explanation:* If enabled, decreases the vertical search range used for P-frame motion estimation to reduce bandwidth.
- *Example:* Enabling low bandwidth mode may be suitable for scenarios with limited network capacity.

GoP Length:

- *Explanation:* The length between two intra-frames in the Group of Pictures.
- *Example:* Setting GoP length to 30 means there are 30 frames between two consecutive intra-frames.

Slice:

- *Explanation:* Number of slices produced for each frame.
- *Example:* A setting of 16 means each frame is divided into 16 slices for parallel processing.

QP (Quantization Parameter):

- *Explanation:* Specifies how to generate the QP per coding unit (CU).
- *Example:* Uniform mode uses the same QP for all CUs, while Auto mode dynamically adjusts QP based on content.

Rate Control:

- *Explanation:* Selects the way the bit rate is controlled (CBR, VBR, LOW_LATENCY).
- *Example:* Choosing CBR ensures a constant bit rate for the encoded video.

B-frame:

- *Explanation:* Short for bidirectional frame, specifying the number of bidirectional predictive frames.
- *Example:* Setting B-frames to 2 allows the encoder to use up to 2 bidirectional frames in compression.

Profile:

- *Explanation:* Defines sets of capabilities or profiles targeting specific classes of applications.
- *Example:* H.264 supports profiles like Baseline, Main, and High, each with different feature sets.

Encoder:

- *Explanation:* Specifies the encoder type, either H.264 or H.265.
- *Example:* Choosing H.264 as the encoder for video compression.

These parameters collectively define how the video encoding process behaves and the characteristics of the encoded video output. The optimal configuration depends on the specific requirements and constraints of the application or use case.

Record Panel Settings

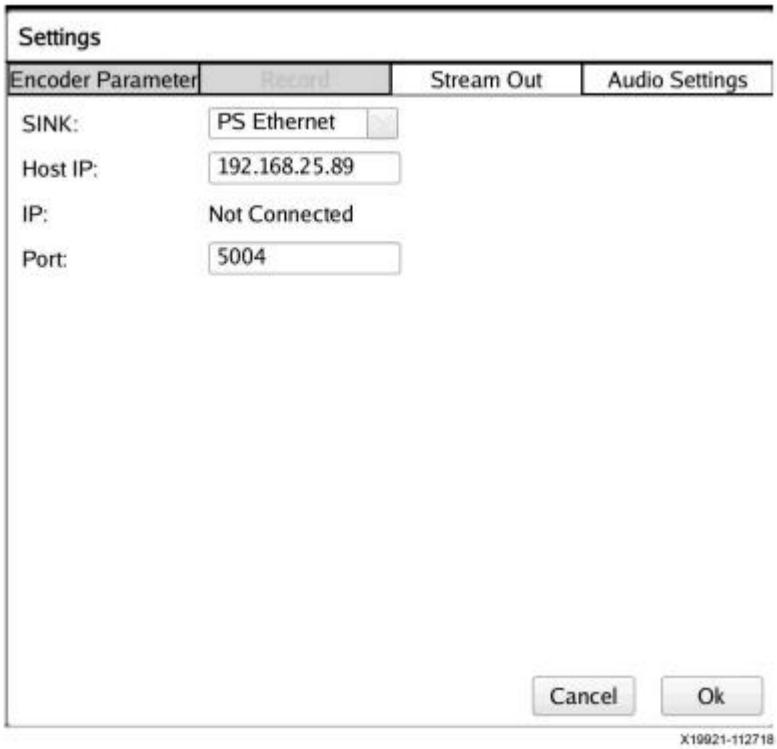
Parameter	Setting
Storage	This option specifies the storage device for the recorded file. The list is dynamically populated based on mounted storage devices. Supported storage devices include SD cards and USB/SATA drives. Note: Because of speed and storage constraints, using SATA or USB 3.0 with an ext4-formatted storage device is recommended for recording.
Output File Name	Name of the output file. A recorded file is saved as *source* _H26x_rec_< *timestamp* >ts where source can be HDMI, TPG, or MIPI and codec can be H264/H265.
Duration	This option specifies the recording time duration. It ranges from 1–3 minutes.

Stream Out

The Stream Out panel allows you to configure streaming parameters. See This Figure and Table: Stream Out Panel Settings .

Stream Out Panel

X-Ref Target - Figure 3-12



Stream Out Panel Settings

Parameter	Setting

SINK	Provides the sink option for the stream out case. It is set to **PS Ethernet** .
Host IP	Provides the option to enter the Host IP address.
IP	Shows the IP address of the board if the Ethernet link is up. If no Ethernet link is connected, it shows **Not Connected.**
Port	Port number of the Ethernet link. The default is **5004** .

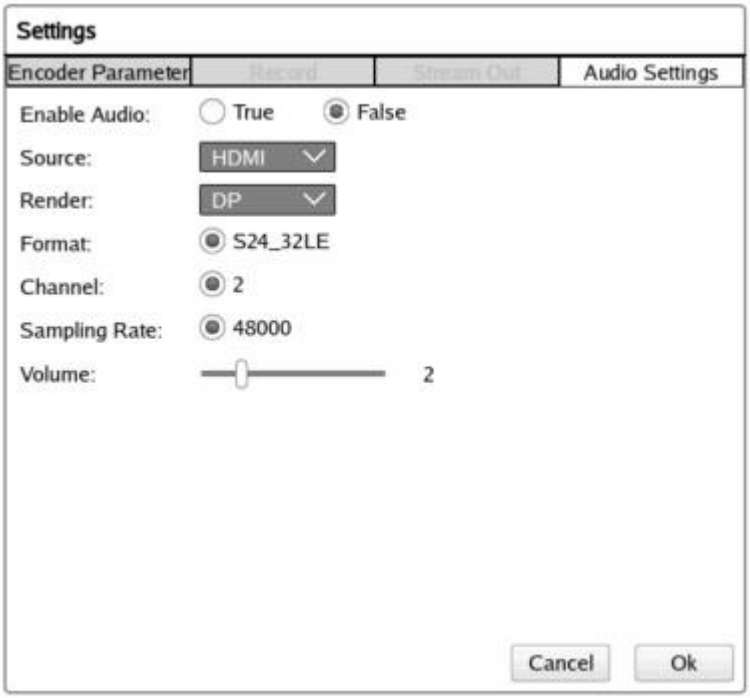
Note: IDR is not user configurable. In the encoder code, the idr value = gop-length.

Audio Setting

The Audio Settings panel is shown in This Figure and described in Table: Audio Settings .

Audio Setting Panel

X-Ref Target - Figure 3-13



Audio Settings

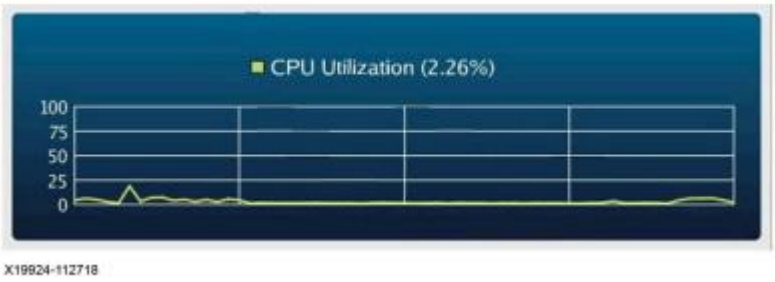
Parameter	Setting
Enable Audio	Enable or disable audio in pipeline.
Format	Audio format. Currently S24_32LE format is supported.
Channel	Number of audio channels. Currently two channels are supported.
Sampling Rate	Audio sampling rate. Currently 48000 is supported.
Volume	Audio volume. Ranges from 0 to 10, default value is 2.
Source	Available sources are HDMI and I2S.
Renderer	Available renderers are DP and I2S.

Monitor Options

The GUI monitors CPU utilization and bandwidth utilization for encoder and decoder AXI ports. See This Figure through This Figure .

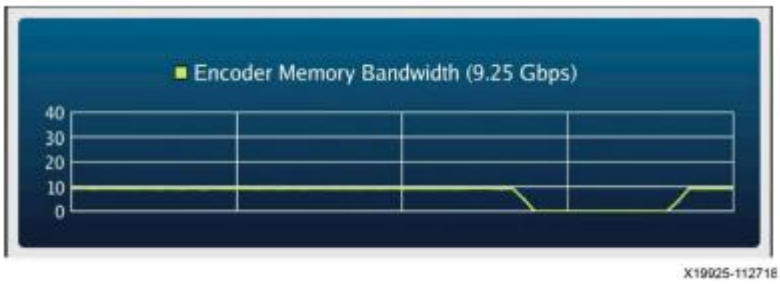
CPU Utilization Plot

X-Ref Target - Figure 3-14



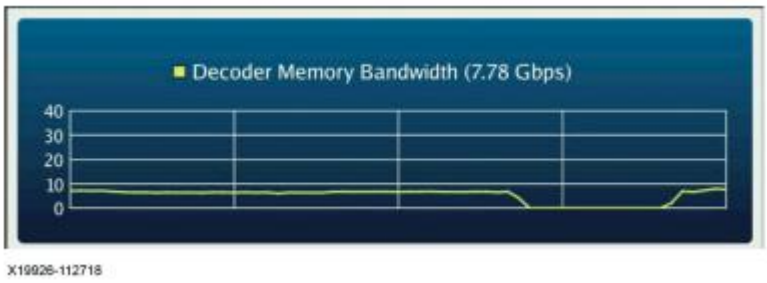
Encoder Bandwidth Utilization Plot

X-Ref Target - Figure 3-15



Decoder Bandwidth Utilization Plot

X-Ref Target - Figure 3-16



GStreamer Application (vcu_gst_app)

Certainly, let's break down the information provided:

GStreamer Application (vcu_gst_app):

- *Explanation:* **vcu_gst_app** is a command line, multi-threaded Linux application that utilizes the functionality exposed by the **vcu_gst_lib** interface. Unlike **vcu_qt**, which

is a GUI application launched once, **vcu_gst_app** is designed for manual execution, requiring input configuration each time it runs.

Input Configuration File (input.cfg):

- *Explanation:* **input.cfg** is a plain text configuration file required by the **vcu_gst_app**. This file contains settings and parameters necessary for configuring the video pipeline. The file format and contents are specified in a way that the application can read and interpret.

Usage Scenario:

- *Explanation:* Users run **vcu_gst_app** from the command line, providing the input configuration file as an argument. The application reads the configuration, configures the video pipeline using **vcu_gst_lib** functions, and runs the multimedia processing tasks specified in the configuration.
- # Example command to run vcu_gst_app with input configuration file
  vcu_gst_app input.cfg

- In this example, **input.cfg** is the name of the configuration file.

Input Configuration File Format:

- *Explanation:* The **input.cfg** file is expected to follow a specific format, and it contains information such as video resolution, encoding parameters, pipeline settings, and any other relevant configuration needed for the video processing tasks.
- *Example **input.cfg** contents:*
- # Sample input.cfg
  [VideoPipeline]
  Resolution = 1920x1080
  Bitrate = 5000
  EncoderMode = CBR

- The actual format and parameters would depend on the requirements of the application and the capabilities provided by the **vcu_gst_lib**.

In summary, **vcu_gst_app** is a command line application that allows users to manually configure and run the video processing pipeline using the **vcu_gst_lib** interface. The configuration details are specified in the **input.cfg** file, which follows a predefined format. This approach provides flexibility for users to customize and experiment with different configurations for multimedia processing tasks.

GStreamer Interface Library (vcu_gst_lib)

The **VCU GStreamer Interface Library (vcu_gst_lib)** is a software library that interacts with the GStreamer multimedia framework to configure various video pipelines in a design and control the data flow through these pipelines. GStreamer is a powerful open-source multimedia framework that facilitates the construction of graphs for handling multimedia data flows.

Let's break down the key features and explanations:

Display Configuration:

- *Explanation:* The library facilitates the configuration of the display parameters, which may include settings such as resolution, format, and source type (e.g., v4l2src for video capture, filesrc for reading from a file).

VCU Configuration:

- *Explanation:* It allows the configuration of the Video Codec Unit (VCU), which is responsible for video encoding and decoding. This configuration may involve settings related to the encoder parameters.

Video Pipeline Control:

- *Explanation:* The library provides controls to start and stop the video pipeline. Starting the pipeline initiates the flow of video data through the configured processing stages.

Audio Pipeline Control:

- *Explanation:* Similar to video pipeline control, the library may handle controls related to audio processing stages if audio is part of the multimedia design.

Video Buffer Management:

- *Explanation:* Efficient management of video buffers, which involves handling memory allocation, deallocation, and efficient data transfer between different stages of the video pipeline.

Exported Interfaces:

- *Explanation:* The library exposes interfaces (functions or methods) that allow external components or applications to interact with and control the video pipeline. These interfaces may include functions to set video pipeline parameters, encoder parameters, start/stop the pipeline, calculate frames per second (FPS), handle errors, calculate bit rate, and poll for end-of-stream events.

Functionality Examples:

- ***Setting Video Pipeline Parameters****:* Configure resolution, format, and source type.
- ***Setting Encoder Parameters****:* Adjust encoding settings such as bitrate, GOP (Group of Pictures) length, and quantization parameters.
- ***Pipeline Control:*** Start and stop the video pipeline as needed.
- ***FPS Calculation:*** Determine the frames per second processed by the pipeline.
- ***Error Handling:*** Manage and respond to errors that may occur during the multimedia processing.
- ***Bit Rate Calculation:*** Calculate the bitrate for file or stream-in playback.
- ***End-of-Stream (EOS) Polling:*** Monitor and respond to the end of the multimedia stream.

In summary, the **vcu_gst_lib** is a crucial interface library that abstracts the complexities of configuring and managing video and audio pipelines in a design using the GStreamer framework. It provides a convenient set of functions to control various aspects of multimedia processing and facilitates integration with other software components.

Description

GStreamer is a library for constructing graphs of media-handling components. The applications it supports range from simple playback and audio/video streaming to complex audio (mixing) and video processing.

GStreamer uses a plug-in architecture which makes the most of GStreamer functionality implemented as shared libraries. The GStreamer base functionality contains functions for registering and loading plug-ins and for providing the fundamentals of all classes in the form of base classes. Plug-in libraries get dynamically loaded to support a wide spectrum of codecs, container formats, and input/output drivers.
Table: GStreamer Plug-ins describes the plug-ins used in the GStreamer interface library.
GStreamer Plug-ins

Plug-in	Description
v4l2src	v4l2src can be used to capture video from V4L2 devices such as AMD HDMI-RX and TPG. Example pipeline: `gst-launch-1.0 v4l2src ! kmssink` This pipeline shows the video captured from a /dev/video0 and rendered on a display unit.
kmssink	The kmssink is a simple video sink that renders raw video frames directly in a plane of a DRM device. Example pipeline: `gst-launch-1.0 v4l2src ! “video/x-raw, format=NV12, width=3840, height=2160” ! kmssink`
omxh26xdec	Decoder omxh26xdec is a hardware-accelerated video decoder that decodes encoded video frames. Example pipeline: `gst-launch-1.0 filesrc location=/media/card/abc.mp4 ! qtdemux ! h26xparse ! omxh26xdec ! kmssink` This pipeline shows an .mp4 multiplexed file where the encoded format is h26x encoded video. Note: Use omxh264dec for H264 decoding and omxh265dec for H265 decoding.
omxh26xenc	Encoder omxh26xenc is a hardware-accelerated video encoder that encodes raw video frames. Example pipeline: `gst-launch-1.0 v4l2src ! omxh26xenc ! filesink location=out.h26x`

	This pipeline shows the video captured from a V4L2 device that delivers raw data. The data is encoded to the h26x encoded video type and stored to a file. Note: Use omxh264enc for H264 encoding and omxh265enc for H265 encoding.
alsasrc	The alsasrc plug-in can be used to capture audio from audio devices such as AMD HDMI-RX. Example pipeline: `gst-launch-1.0 alsasrc device=hw:1,1 ! queue ! audioconvert ! audioresample ! audio/x-raw, rate=48000, channels=2, format=S24_32LE ! alsasink device="hw:1,0"` This pipeline shows the audio captured from an ALSA source and plays on an ALSA sink.
alsasink	The alsasink is a simple audio sink that plays raw audio frames. Example pipeline: `gst-launch-1.0 alsasrc device=hw:1,1 ! queue ! audioconvert ! audioresample ! audio/x-raw, rate=48000, channels=2, format=S24_32LE ! alsasink device="hw:1,0"` This pipeline shows the audio captured from the ALSA source and plays on an ALSA sink.
faad	Decoder faad is an audio decoder that decodes encoded audio frames. Example pipeline: `gst-launch-1.0 filesrc location=out.ts ! tsdemux ! aacparse ! faad ! audioconvert ! audioresample ! audio/x-raw,rate=48000,channnels=2, format=S24_32LE ! alsasink device="hw:1,0"` This pipeline shows a .ts multiplexed file where the encoded format is aac encoded audio. The data is decoded and played on an ALSA sink device.
faac	Encoder faac is an audio encoder that encodes raw audio frames. Example pipeline: `gst-launch-1.0 alsasrc device=hw:1,1 num-buffers=500 ! audio/x-raw, format=S24_32LE, rate=48000 ,channels=2 !`

	queue ! audioconvert ! audioresample ! faac ! aacparse ! mpegtsmux ! filesink location=out.ts This pipeline shows the audio captured from an ALSA device that delivers raw data. The data is encoded to aac format and stored to a file.
xilinxscd	xilinxscd is hardware-accelerated IP that enables detection of scene change in a video stream. This plugin generates upstream events whenever there is scene change in an incoming video stream so the encoder can insert an Intra frame to improve video quality. Example pipeline: gst-launch-1.0 -v v4l2src ! video/x-raw, width=3840, height=2160, format=NV12, framerate=60/1 ! xilinxscd io-mode=5 ! omxh26xenc ! filesink location=/run/out.h26x This pipeline shows the video captured from a V4L2 device that delivers raw data. This raw data is passed through the xilinxscd plugin which analyzes the stream in runtime and provides an event to the encoder whether any scene change is detected in a video stream or not. The encoder uses this information to insert an I-frame in an encoded bit-stream. Use omxh264enc for H264 encoding and omxh265enc for H265 encoding.
appsrc	The appsrc element can be used by applications to insert data into a GStreamer pipeline. Unlike most GStreamer elements, appsrc provides external API functions.
appsink	appsink is a sink plugin that supports many different methods, enabling the application to manage the GStreamer data in a pipeline. Unlike most GStreamer elements, appsink provides external API functions.

VCU Encoder/Decoder Features

Region of Interest Encoding:

- *Explanation:* This feature allows users to tag specific regions in a video frame for encoding with user-supplied quality settings (high, medium, low, or don't-care) relative to the picture background. Untagged regions are encoded with the default settings.

Scene Change Detection (SCD):

- *Explanation:* The Scene Change Detection (SCD) functionality generates an SCD event. This event is passed along with the buffer to the encoder. The encoder uses this information to make decisions such as inserting an I-frame (Intra-frame) instead of a P-frame or a B-frame. Inserting an I-frame at the point where a scene change is detected helps retain the quality of the video.

Interlaced Video:

- *Explanation:* VCU supports both encoding and decoding of H.265 interlaced video. Interlaced video is a video compression technique in which each frame is partitioned into two fields, and these fields are captured and played back sequentially, providing a smoother video display for certain types of content.

DCI-4k Encode/Decode:

- *Explanation:* VCU is capable of encoding or decoding video at a resolution of 4096x2160p60 (DCI-4k) provided that the VCU Core clock frequency is set to 712 MHz. DCI-4k is a standard for digital cinema projection and represents a resolution of 4096x2160 pixels.

Low-Latency (LLP1):

- *Explanation:* In Low-Latency Profile 1 (LLP1), the frame is divided into multiple slices. The VCU encoder output and decoder input are processed in slice mode, while the VCU Encoder input and Decoder output still operate in frame mode.

Low-Latency (LLP2):

- *Explanation:* In Low-Latency Profile 2 (LLP2), the frame is divided into multiple slices, and the VCU encoder output and decoder input are processed in slice mode. The producer (Capture DMA) and the consumer (VCU Encoder) work on the same buffer with synchronization IP in place. There is no synchronization IP between the decoder and display. The decoder signals the display component when half of the input frame is ready.

XAVC (eXtensible Audio Video Codec):

- *Explanation:* The VCU encoder can produce xAVC Intra or xAVC Long GOP compliant bitstreams. XAVC is commonly used for video record use-cases. It's important to note that h264parse does not support XAVC parsing. Therefore, XAVC may not be valid for serial and streaming use-cases, as the parser does not support these scenarios.

In summary, the VCU encoder/decoder features cover a range of capabilities including region of interest encoding, scene change detection, support for interlaced video, DCI-4k encoding/decoding, and various low-latency profiles. The support for XAVC further extends its use in video recording applications.

Multi-Stream

When the number of inputs is more than one in the command line application, it is a multi-stream use case. In multi-stream use cases, multiple HDMIs are the same replica of a single source. Table: Multi-Stream Features provides information about the multi-stream features supported in various designs.

Multi-Stream Features

Design	Input Source	Format	Resolution
Full-fledged VCU TRD	HDMI-RX Single Sensor MIPI CSI2-RX TPG	NV12	4kp60 2-4kp30 4-1080p60 8-1080p30
LLP2 PS DDR NV12 HDMI Audio Video Capture and HDMI Display	HDMI-RX	NV12	4kp60 2-4kp30 4-1080p60 for encoder 2-1080p60 for decoder
LLP2 PL DDR HDMI Video Capture and HDMI Display	HDMI-RX	NV16, XV20	4kp60 2-4kp30 1080p60 for encoder 4-1080p60 for decoder
Multi-Stream Audio Design	HDMI-RX Single Sensor MIPI CSI2-RX	NV12	4kp60 2-4kp30 2-1080p60

The command line application (vcu_gst_app) is a versatile tool that supports various video processing scenarios, including multi-streaming, multi-recording, and multi-display. The use cases mentioned in the description provide examples of different configurations and setups for the vcu_gst_app.

Multi-Streaming Use Case (Figure 1):

- **Configuration:** The application is configured to run three HDMI streams and one MIPI stream simultaneously, all in 1080p60 resolution.
- **Source Types:** The input sources can be Test Pattern Generator (TPG), HDMI, or MIPI.
- **Purpose:** This scenario is suitable for applications where multiple video streams need to be processed simultaneously, such as video surveillance or multi-camera setups.

Figure 1

(Multi-Stream—3 HDMI and 1 MIPI Input Sources @ 1080p60)

Multi-Recording and Display Use Case (Figure 2):

- **Configuration:** The application is configured to run seven HDMI streams and one MIPI stream in multi-stream mode, all in 1080p30 resolution.
- **Source Types:** For 8-1080p30 input, the source types can be MIPI or HDMI.
- **Purpose:** This setup showcases the capability to handle multiple input streams for both recording and display. In this case, the application is capable of processing eight different video streams simultaneously. Note that only half of each stream is displayed in this specific configuration to showcase all eight streams on a single screen.

In summary, the vcu_gst_app provides a flexible and powerful command line interface to configure and control video processing pipelines. It can handle scenarios involving different video sources, resolutions, and multiple streams simultaneously, making it suitable for a range of applications where advanced video processing capabilities are required.

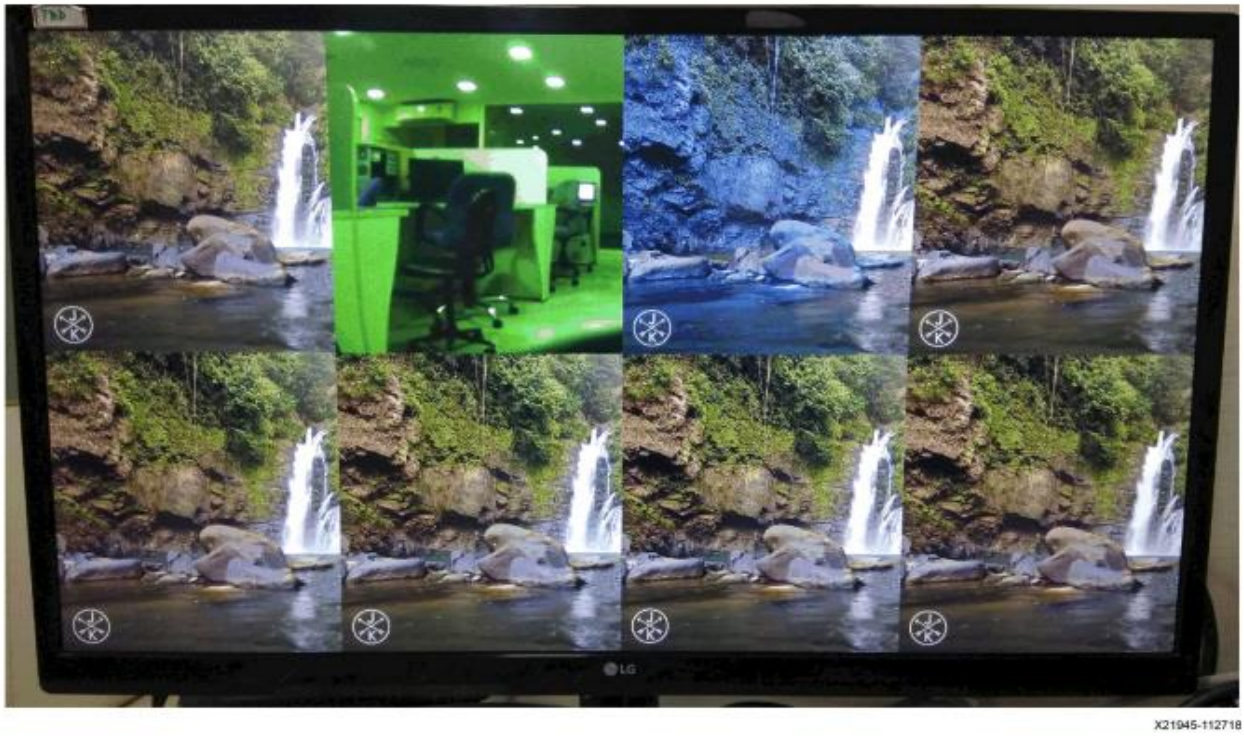


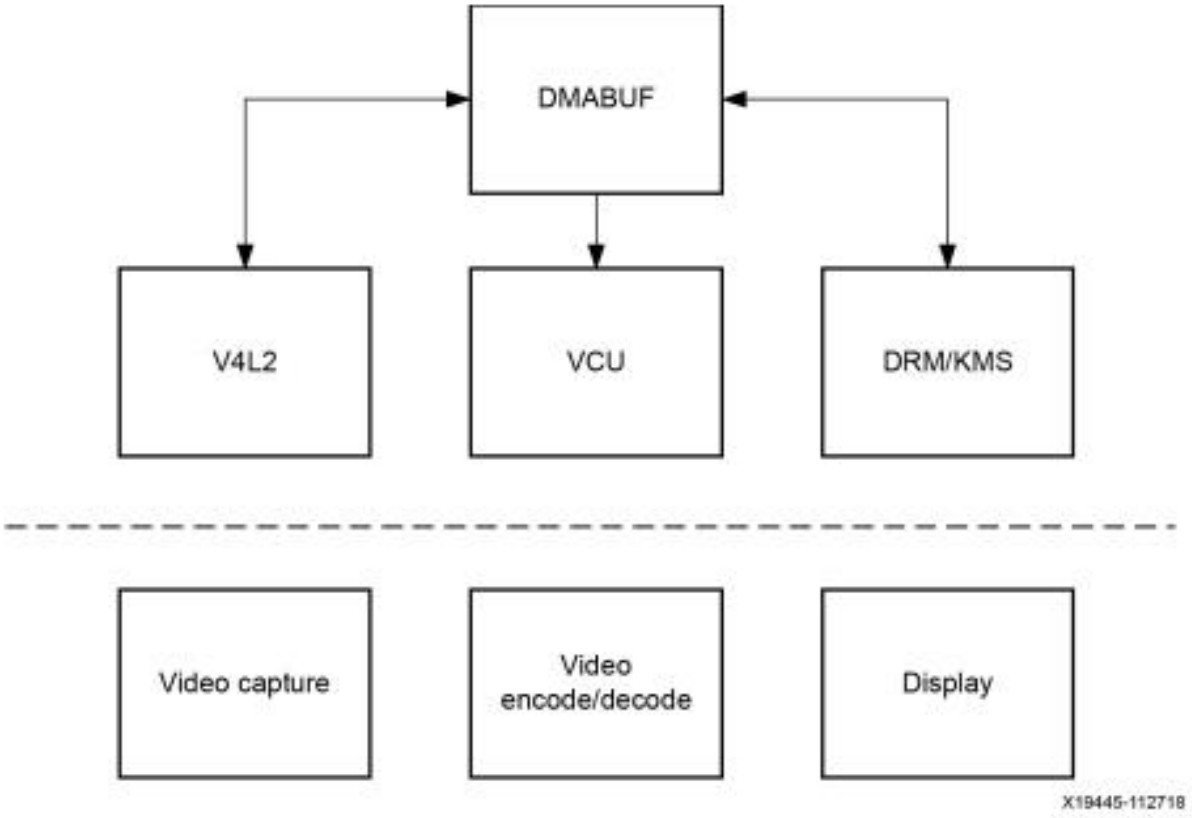
Figure 2

(Multi-Stream—7 HDMI and 1 MIPI Input Sources @ 1080p30)

Video Buffer Management

In the case of a raw/processed pipeline, the video capture device (v4l2src), video processing accelerator (VCU element), and kmssink plugin use DMABUF framework for sharing buffers between peer elements (see This below Figure).

The provided text outlines the process of DMA (Direct Memory Access) buffer sharing in a video processing pipeline, specifically in the context of a raw/processed pipeline involving a video capture device (v4l2src), a video processing accelerator (VCU element), and a display plugin (kmssink). DMA buffer sharing is a mechanism for efficiently transferring video data between different components in the pipeline.



Above figure: Buffer Sharing

The following steps are performed in DMA buffer sharing.

In the capture-encode side:

1. The V4L2 capture device (the client driver) allocates buffer.

2. The v4l2src plug-in exports/imports the DMA buffer to the gst-omx plug-in.

3. The gst-omx plug-in passes the file descriptor to the encoder driver.

4. The encoder driver uses the DMA_BUF framework and reads the kernel buffer for encoding.

In the playback side:

1. The decoder driver allocates DMA buffer.

2. The gst-omx plug-in exports the file descriptor (FD) to the kmssink plug-in.

3. The kmssink plug-in passes the file descriptor to the DisplayPort controller driver.

4. The DisplayPort driver uses the kernel DMA_BUF framework to know the decoder buffer location.

5. The DisplayPort DMA reads the decoded buffer without copying the buffer in kernel memory.

Here's a step-by-step explanation:

Capture-Encode Side:

Buffer Allocation by V4L2 Capture Device:

- The V4L2 capture device (client driver) allocates a buffer to store the raw video data.

DMA Buffer Export/Import by v4l2src:

- The v4l2src plugin, responsible for capturing video frames, exports/imports the DMA buffer to the gst-omx plugin. This involves making the buffer accessible to other components in the pipeline.

File Descriptor Passing to Encoder Driver:

- The gst-omx plugin, designed for integrating OpenMAX components (including the VCU element), passes the file descriptor of the DMA buffer to the encoder driver.

DMA_BUF Framework Usage by Encoder:

- The encoder driver utilizes the DMA_BUF framework to read the kernel buffer for encoding. This involves efficient access to the video data for compression.

Playback Side:

DMA Buffer Allocation by Decoder Driver:

- The decoder driver allocates a DMA buffer to store the decoded video data.

File Descriptor Export by gst-omx:

- The gst-omx plugin exports the file descriptor (FD) associated with the DMA buffer to the kmssink plugin. This allows the decoded video data to be passed efficiently to the display subsystem.

File Descriptor Passing to DisplayPort Controller:

- The kmssink plugin passes the file descriptor to the DisplayPort controller driver. The DisplayPort controller is responsible for managing the display hardware.

Kernel DMA_BUF Framework Usage by DisplayPort Driver:

- The DisplayPort driver utilizes the kernel DMA_BUF framework to determine the location of the decoder buffer. This enables the efficient reading of the decoded buffer without the need for copying the buffer in kernel memory.

Decoded Buffer Reading by DisplayPort DMA:

- The DisplayPort DMA reads the decoded buffer directly without copying the buffer in the kernel memory. This direct access enhances efficiency in the video playback process.

In summary, DMA buffer sharing optimizes the transfer of video data between the capture-encode side and the playback side, ensuring efficient processing and display of video frames in the pipeline.

Detailed explaination of buffer sharing:

Certainly! Let's delve into the detailed explanation of the Capture-Encode Side and Playback Side processes in the context of DMA (Direct Memory Access) buffer sharing in a video processing pipeline.

Capture-Encode Side:

1. Buffer Allocation by V4L2 Capture Device:

- The V4L2 capture device, often a hardware component, is the client driver responsible for capturing raw video data. It allocates a buffer in memory to temporarily store the raw video frames it captures from a video source.

2. DMA Buffer Export/Import by v4l2src:

- The v4l2src plugin, part of the GStreamer framework, is responsible for interfacing with the V4L2 capture device and capturing video frames. It facilitates the export/import of the DMA buffer to and from the gst-omx plugin. This step ensures that the buffer is accessible to other components in the pipeline.

3. File Descriptor Passing to Encoder Driver:

- The gst-omx plugin, designed to integrate OpenMAX components (such as the VCU encoder), takes the file descriptor associated with the DMA buffer from v4l2src. It then passes this file descriptor to the encoder driver. This communication allows the encoder to work with the video data stored in the buffer.

4. DMA_BUF Framework Usage by Encoder:

- The encoder driver, leveraging the DMA_BUF framework, efficiently accesses the kernel buffer containing the raw video data. This is a crucial step in the video processing pipeline as it allows the encoder to compress the video frames before further processing.

Playback Side:

1. DMA Buffer Allocation by Decoder Driver:

- On the playback side, the decoder driver allocates a DMA buffer to store the decoded video data. This buffer will be used to hold the frames after they have been processed by the decoder.

2. File Descriptor Export by gst-omx:

- The gst-omx plugin exports the file descriptor associated with the DMA buffer containing decoded video data. This export is performed after the decoding process, and the file descriptor is passed to the kmssink plugin.

3. File Descriptor Passing to DisplayPort Controller:

- The kmssink plugin, responsible for displaying video frames, passes the file descriptor received from gst-omx to the DisplayPort controller driver. This controller is responsible for managing the display hardware, which could be connected to a monitor or another display device.

4. Kernel DMA_BUF Framework Usage by DisplayPort Driver:

- The DisplayPort driver utilizes the kernel DMA_BUF framework to determine the location of the DMA buffer containing the decoded video data. This usage is crucial for efficiently reading the decoded buffer without the need for copying the data within the kernel memory.

5. Decoded Buffer Reading by DisplayPort DMA:

- The DisplayPort DMA (Direct Memory Access) component reads the decoded buffer directly from the DMA buffer. This direct access enhances efficiency in the video playback process by avoiding unnecessary data copying within the kernel memory.

In Summary:

- DMA buffer sharing optimizes the transfer of video data between the capture-encode side and the playback side, ensuring efficient processing and display of video frames in the video processing pipeline. It minimizes unnecessary data copies and enhances overall system performance.

The steps in the buffer sharing detailed explaination:

Capture-Encode Side:

2. DMA Buffer Export/Import by v4l2src:

Certainly! Let's break down the step "DMA Buffer Export/Import by v4l2src" into more specific points:

- **Interfacing with V4L2 Capture Device:**
  - The v4l2src plugin communicates with the V4L2 (Video for Linux 2) capture device, which is responsible for capturing raw video frames from a video source, such as a camera or video input.
- **Buffer Capture and Allocation:**
  - V4L2 capture device captures raw video frames and allocates a buffer in memory to store these frames temporarily.
- **DMA Buffer Export:**
  - v4l2src, being part of the GStreamer framework, uses the DMA_BUF framework to export the allocated buffer. This involves making the buffer accessible to other components in the GStreamer pipeline, allowing for efficient data sharing.
- **Import by gst-omx Plugin:**
  - The exported DMA buffer is then imported by the gst-omx plugin. This import process ensures that the gst-omx plugin, responsible for integrating OpenMAX components, can access and work with the DMA buffer containing raw video data.
- **Facilitating Buffer Accessibility:**
  - By facilitating the export/import process, v4l2src ensures that the DMA buffer becomes a shared resource within the GStreamer pipeline. Other components downstream in the pipeline, such as encoders or processors, can now efficiently access and manipulate the video data stored in this buffer.
- **Efficient Data Transfer:**
  - The use of the DMA_BUF framework for export/import is designed for efficient data transfer between different components in the pipeline. Direct Memory Access (DMA) techniques are employed to minimize CPU involvement, enhancing data transfer speed and overall system performance.
- **Enabling Further Processing:**

    - Once the DMA buffer is successfully imported by components like gst-omx, it becomes a central data container for raw video frames. This enables subsequent components, such as encoders, to efficiently perform processing tasks without the need for redundant data copying.

In summary, the "DMA Buffer Export/Import by v4l2src" step establishes a shared mechanism for video data exchange within the GStreamer pipeline, ensuring efficient and low-latency communication between the V4L2 capture device and downstream components like the gst-omx plugin.

3. File Descriptor Passing to Encoder Driver:
Certainly! Let's break down the step "File Descriptor Passing to Encoder Driver" into more specific points:

- **File Descriptor Retrieval:**
    - The gst-omx plugin, which is responsible for integrating OpenMAX components, retrieves the file descriptor associated with the DMA buffer from v4l2src. This file descriptor is a numerical identifier that represents the opened DMA buffer.
- **Integration of OpenMAX Components:**
    - The gst-omx plugin is specifically designed to work with OpenMAX components, which include the VCU (Video Codec Unit) encoder. OpenMAX is a standardized API for media processing that provides a framework for integrating multimedia components.
- **Passing File Descriptor to Encoder Driver:**
    - Once the file descriptor is obtained, the gst-omx plugin passes this file descriptor to the encoder driver. The encoder driver is the software component responsible for controlling and managing the VCU encoder hardware.
- **Communication with Encoder:**
    - The file descriptor serves as a communication link between the gst-omx plugin and the encoder driver. It allows the encoder driver to locate and access the DMA buffer containing the raw video data captured by the V4L2 capture device.
- **Access to Video Data:**
    - With the file descriptor, the encoder driver gains access to the video data stored in the DMA buffer. This is a crucial step for the encoder as it needs to read the raw video frames to perform compression and encoding tasks.
- **Efficient Data Handling:**
    - The use of file descriptors, along with the DMA_BUF framework, enables efficient data handling between different components in the GStreamer pipeline. It minimizes unnecessary data copying and facilitates direct access to shared memory buffers.
- **Encoder Configuration:**
    - The encoder driver, armed with the file descriptor and associated buffer, configures the VCU encoder based on the characteristics of the raw video

frames. This includes setting parameters such as resolution, format, and encoding options.

In summary, the "File Descriptor Passing to Encoder Driver" step establishes a communication pathway from the gst-omx plugin to the VCU encoder driver, ensuring that the encoder can access and process the raw video data captured by the V4L2 capture device efficiently.

4. DMA_BUF Framework Usage by Encoder:
Certainly! Let's dive into the details of "DMA_BUF Framework Usage by Encoder":

DMA_BUF Framework Usage by Encoder:

- **Efficient Buffer Access:**
  - The encoder driver utilizes the DMA_BUF framework, a kernel-level mechanism for sharing and accessing buffers between different components in a system. This framework ensures efficient and direct access to the kernel buffer that holds the raw video data.
- **Integration with DMA_BUF Framework:**
  - The encoder is integrated with the DMA_BUF framework to take advantage of its capabilities. This integration involves using the framework's APIs and functionalities to work with shared buffers seamlessly.
- **File Descriptor Utilization:**
  - The file descriptor passed from the gst-omx plugin (which, in turn, received it from v4l2src) is a key element in this step. The file descriptor serves as an identifier for the shared DMA buffer and allows the encoder to locate and access the raw video data.
- **Reading Kernel Buffer for Compression:**
  - Leveraging the information provided by the DMA_BUF framework and the associated file descriptor, the encoder reads the contents of the kernel buffer that holds the raw video frames. This read operation is a crucial preparatory step for the subsequent compression process.
- **Compression of Video Frames:**
  - Once the raw video data is obtained from the DMA buffer, the encoder performs video compression. The compression algorithm (e.g., H.264 or H.265) is applied to reduce the size of the video frames, making them suitable for efficient storage, transmission, or further processing.
- **Passing Compressed Data Downstream:**
  - After compression, the encoder passes the compressed video frames downstream in the processing pipeline. These compressed frames are then ready for storage, transmission, or additional processing by subsequent components in the pipeline.

In summary, the DMA_BUF framework usage by the encoder ensures a streamlined and efficient process of accessing raw video frames, a crucial step in the video processing pipeline, enabling subsequent compression for optimized data handling.

Playback Side:

1. DMA Buffer Allocation by Decoder Driver:

DMA Buffer Allocation by Decoder Driver:

- **Memory Allocation for Decoded Video Data:**
  - The decoder driver, responsible for handling the decoding process, allocates a DMA buffer in memory. This buffer serves as the designated space to store the decoded video frames after they have undergone the decoding process.
- **Buffer Size Determination:**
  - The size of the allocated DMA buffer is determined based on factors such as the video resolution, color format, and any additional requirements specified by the decoding process. This ensures that the buffer has sufficient capacity to accommodate the decoded frames.
- **Buffer Location in Kernel Memory:**
  - The allocated DMA buffer is established within the kernel memory space. This location is critical for efficient and direct access to the decoded video data by subsequent components in the playback pipeline.
- **Buffer Identifier and Control:**
  - The decoder driver assigns an identifier or file descriptor to the allocated DMA buffer. This identifier is crucial for tracking and referencing the buffer throughout its lifecycle. Additionally, the driver sets up necessary controls and configurations for the proper functioning of the DMA buffer.
- **Ready for Decoded Frame Storage:**
  - With the DMA buffer allocated, configured, and ready, the decoder driver is prepared to receive and store the decoded video frames. The buffer acts as a temporary storage space for the frames as they transition from the decoding process to the final stages of the playback pipeline.

In summary, the "DMA Buffer Allocation by Decoder Driver" step ensures the availability of a designated memory space to hold the decoded video frames. This is a foundational element in the playback process, enabling efficient storage and subsequent handling of the video data in the kernel memory.

2. File Descriptor Export by gst-omx:

In the context of DMA buffer sharing in a video processing pipeline, the step "File Descriptor Export by gst-omx" involves the following actions:

- **Decoded Video Data Processing:**
  - After the decoder has processed the video frames, the decoded data is stored in a DMA buffer.
- **File Descriptor Export:**

    - The gst-omx plugin, which is responsible for integrating OpenMAX components (including the decoder), performs an export operation on the file descriptor associated with the DMA buffer containing the decoded video data.
- **Passing to kmssink Plugin:**
    - Subsequently, the exported file descriptor is passed to the kmssink plugin.
- **Communication with Display subsystem:**
    - The kmssink plugin is responsible for handling the display of video frames. The file descriptor provides a means for communication between different components in the pipeline, enabling efficient transfer of the decoded video data from the decoder to the display subsystem.

In summary, this step ensures that the decoded video data, stored in the DMA buffer, can be efficiently communicated to the display subsystem (possibly through the kmssink plugin) for rendering on the screen or further processing in the video playback side of the pipeline. The file descriptor serves as a key mechanism for sharing information about the location and access to the DMA buffer among different components.

3. File Descriptor Passing to DisplayPort Controller:
In the context of DMA buffer sharing in a video processing pipeline, the step "File Descriptor Passing to DisplayPort Controller" involves the following actions:

- **File Descriptor Reception by kmssink:**
    - The kmssink plugin, responsible for rendering or displaying video frames, receives the file descriptor that was exported by the gst-omx plugin. This file descriptor is associated with the DMA buffer containing the decoded video data.
- **Passing to DisplayPort Controller:**
    - The kmssink plugin then passes the received file descriptor to the DisplayPort controller driver. The DisplayPort controller is a hardware or software component responsible for managing the display subsystem and interfacing with the display hardware.
- **Communication with Display Hardware:**
    - The DisplayPort controller, upon receiving the file descriptor, utilizes the associated information to efficiently access the DMA buffer containing the decoded video data. This communication allows the controller to know the location and details of the buffer without the need for data copying in the kernel memory.
- **Efficient Display Processing:**
    - With access to the DMA buffer, the DisplayPort controller can efficiently read the decoded video data and process it for display on the connected monitor or display device.

In summary, this step ensures that the decoded video data is seamlessly passed from the kmssink plugin to the DisplayPort controller, facilitating efficient communication and processing for display on the connected hardware. The file descriptor acts as a key

mechanism for coordinating the transfer of video data among different components in the video playback side of the pipeline.

4. Kernel DMA_BUF Framework Usage by DisplayPort Driver:
In the context of DMA buffer sharing in a video processing pipeline, the step "Kernel DMA_BUF Framework Usage by DisplayPort Driver" involves the following actions:

- **Utilizing Kernel DMA_BUF Framework:**
  - The DisplayPort driver, which is responsible for managing the communication with the display hardware, leverages the kernel DMA_BUF framework.
- **Determining Buffer Location:**
  - The key purpose of this step is to determine the location of the DMA buffer that contains the decoded video data. The DMA_BUF framework provides a mechanism for sharing buffers between different components in the pipeline.
- **Efficient Buffer Access:**
  - By utilizing the kernel DMA_BUF framework, the DisplayPort driver can efficiently access the decoded video data without the need for copying the data within the kernel memory. This direct access enhances efficiency in the video playback process.
- **Coordinating with Display Hardware:**
  - The information obtained from the DMA buffer, such as its location and details, allows the DisplayPort driver to effectively coordinate with the connected display hardware. This coordination is essential for rendering the video frames on the display device.
- **Enhancing Performance:**
  - The direct access to the DMA buffer without copying data within the kernel memory enhances the overall performance of the display subsystem. It reduces unnecessary overhead and ensures a smoother and more efficient video playback experience.

In summary, this step ensures that the DisplayPort driver efficiently utilizes the kernel DMA_BUF framework to access the decoded video data, facilitating seamless communication with the display hardware and enhancing the performance of the video playback process.

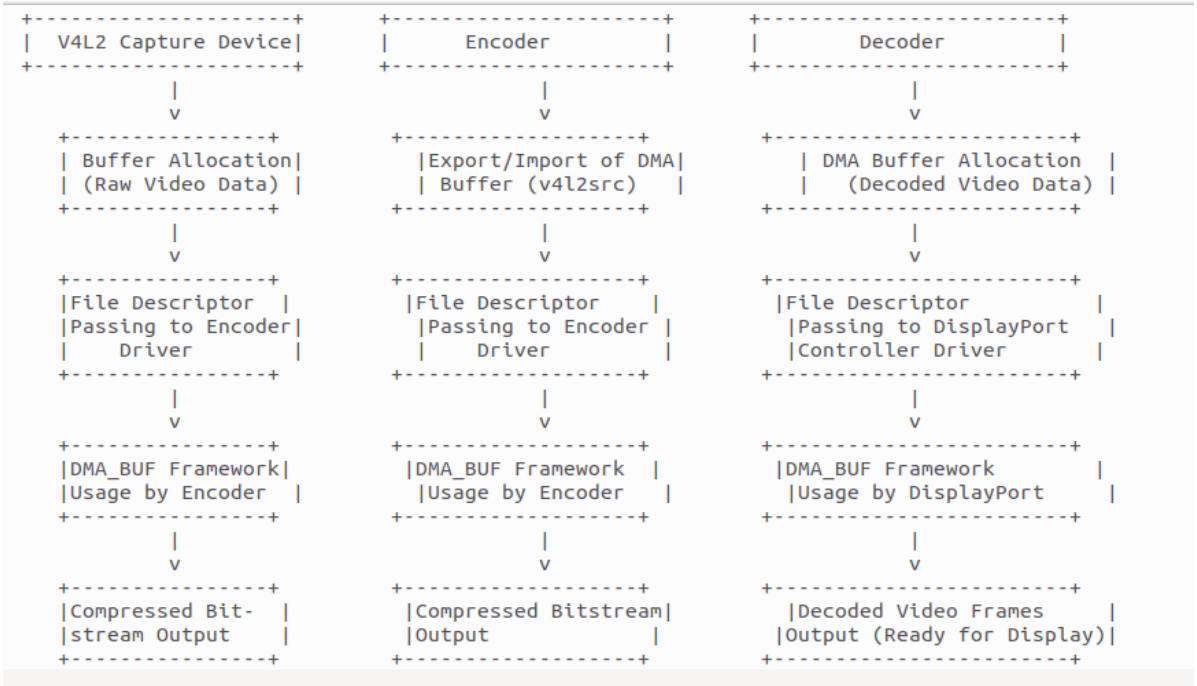
5. Decoded Buffer Reading by DisplayPort DMA:
In the context of DMA buffer sharing in a video processing pipeline, the step "Decoded Buffer Reading by DisplayPort DMA" involves the following actions:

- **Direct Reading from DMA Buffer:**
  - The DisplayPort DMA (Direct Memory Access) component directly reads the decoded video data from the DMA buffer. DMA allows peripherals, such as the DisplayPort controller, to access system memory (DMA buffer) without involving the CPU. This direct reading is performed to efficiently retrieve the decoded frames.
- **Avoiding Data Copying:**

    - By directly accessing the DMA buffer, the DisplayPort DMA avoids unnecessary data copying within the kernel memory. DMA enables data transfers between peripherals and memory without CPU intervention, leading to reduced latency and improved overall system performance.
- **Efficiency in Video Playback:**
    - The direct reading of the decoded buffer by the DisplayPort DMA enhances efficiency in the video playback process. It ensures a smooth and streamlined flow of video frames from the DMA buffer to the display hardware without introducing additional delays or computational overhead.
- **Synchronization with Display Timing:**
    - The DisplayPort DMA is synchronized with the display timing and requirements. It reads the decoded frames at the appropriate moments, aligning with the display refresh rate and other parameters to ensure coherent and synchronized video playback.
- **Enhanced Performance:**
    - The use of DMA for reading the decoded buffer contributes to enhanced performance during video playback. The efficiency gained by avoiding unnecessary data copying and ensuring direct access results in a more responsive and optimized video display on the connected monitor or display device.

In summary, this step ensures that the DisplayPort DMA efficiently reads the decoded video data directly from the DMA buffer, leveraging DMA capabilities to enhance overall performance and achieve seamless video playback.

Flow diagram buffer processing and sharing



AXI Performance Monitor (APM) Library (vcu_apm_lib)

The AXI Performance Monitor (APM) Library (**vcu_apm_lib**) is designed to provide an interface to the **vcu_qt** application, allowing it to read performance metrics related to memory throughput for the VCU (Video Codec Unit) encoder and decoder. The APM is a specialized component used for monitoring and analyzing the performance of the AXI (Advanced eXtensible Interface) interconnect, which is commonly used in FPGA-based systems for high-speed communication between different IP blocks.

Here's an explanation of the programming model and functions provided by the APM Library:

Programming Model:

- **Initialization:**
  - The application, in this case, **vcu_qt**, starts by calling **perf_monitor_init()** during its initialization phase. This function is responsible for setting up and initializing the necessary components for monitoring the AXI Performance.
- **Periodic Monitoring:**
  - During the operation of the application, the **vcu_qt** program periodically calls the **perf_monitor_get_rd_wr_cnt()** function for each VCU APM. This API (Application Programming Interface) returns the number of read and write transactions happening on the AXI Performance Monitor port, measured in bytes.
- **Deinitialization:**
  - When the application is about to exit, it calls **perf_monitor_deinit()**. This function is responsible for cleaning up and releasing any resources associated with the AXI Performance Monitoring.

Functions:

- **perf_monitor_init():**
  - This function initializes the AXI Performance Monitoring system. It sets up the necessary configurations and resources to start monitoring the AXI interconnect.
- **perf_monitor_get_rd_wr_cnt():**
  - This function is called periodically to retrieve the number of read and write transactions happening on the AXI Performance Monitor port. The returned values provide insights into the memory throughput performance of the VCU encoder and decoder.
- **perf_monitor_deinit():**
  - This function is called during the application's exit phase to clean up and release any resources associated with the AXI Performance Monitoring.

Purpose:
The primary purpose of using the AXI Performance Monitor and the associated library is to gather performance metrics related to memory transactions in the VCU encoder and decoder.

Monitoring read and write transactions helps in assessing the efficiency and workload of the memory subsystem, which is crucial for video processing tasks. This information can be valuable for optimizing and fine-tuning the system for better performance.

In summary, the APM Library provides a convenient interface for the **vcu_qt** application to monitor and gather insights into the memory throughput performance of the VCU encoder and decoder during runtime.

Video Library (vcu_video_lib)

The **vcu_video_lib** library plays a crucial role in configuring various video pipelines within a design. It provides an interface for querying display configurations and facilitates the configuration of media pipelines for video capture. Additionally, the library exports and imports interfaces to and from other components, enhancing interoperability with different middleware layers.

Key Functions and Responsibilities:

- **Configuring Video Pipelines:**
  - The primary responsibility of **vcu_video_lib** is to configure video pipelines within the system. This involves setting up the necessary parameters, connections, and controls for capturing and processing video.
- **Querying Display Configurations:**
  - The library has the capability to query display configurations. This means it can retrieve information about the available display settings, resolutions, and other relevant parameters. This information is crucial for ensuring compatibility and optimal video playback.
- **TPG Video Source Controls:**
  - The library exports TPG (Test Pattern Generator) video source controls to the **vcu_gst_lib** library. TPG is a component used to generate test patterns, and by exporting controls, the video library enables the **vcu_gst_lib** to interact with and control the TPG as needed.
- **CSI Video Source Controls:**
  - Similarly, the library exports CSI (Camera Serial Interface) video source controls to the **vcu_gst_lib** library. CSI is a standard interface for connecting cameras to processors, and by exporting controls, the video library allows the **vcu_gst_lib** to manage and control CSI video sources.
- **Interfaces to Middleware Layers:**
  - **vcu_video_lib** interacts with various middleware layers, including Video for Linux 2 (V4L2), Media Controller, and Direct Rendering Manager (DRM). These middleware layers are essential components in the Linux multimedia framework and are used for managing video devices, controlling media pipelines, and handling display rendering.

Collaboration with vcu_gst_lib:

- The library collaborates closely with the **vcu_gst_lib** library, exporting controls related to TPG and CSI video sources. This collaboration ensures that

the GStreamer interface library can effectively manage and control video sources in the design.

Purpose:

- The primary purpose of **vcu_video_lib** is to serve as a configuration and management interface for video pipelines in the system. It abstracts the complexity of video configuration and provides a standardized way for different components to interact with and control video sources.

In summary, **vcu_video_lib** is a critical component in a multimedia system, responsible for configuring video pipelines, querying display configurations, and facilitating communication with middleware layers and other libraries like **vcu_gst_lib**.

Explanation:

- **Initialization and Configuration:**
- The application starts by initializing the **vcu_video_lib** using **vcu_video_lib::initialize()**.
- **Video Pipeline Configuration:**
- The application configures the video pipeline for video capture using **vcu_video_lib::configureVideoPipeline(/* parameters */)**. This involves setting up parameters such as resolution, format, and source type.
- **Display Configuration Query:**
- The application queries display configurations using **vcu_video_lib::queryDisplayConfig()**. The obtained **DisplayConfig** object contains information about available display settings.
- **Interacting with TPG and CSI:**
- The application exports controls related to TPG and CSI video sources using **vcu_video_lib::exportTPGControls(/* controls */)**, **vcu_video_lib::exportCSIControls(/* controls */)**. This allows external components, like **vcu_gst_lib**, to interact with and control these video sources.
- **Middleware Interaction:**
- The application interacts with middleware layers, such as V4L2, Media Controller, and DRM, using **vcu_video_lib::interactWithMiddleware(/* parameters */)**. This ensures proper integration with the multimedia framework.
- **Cleanup:**
- Finally, the application cleans up and deinitializes the **vcu_video_lib** using **vcu_video_lib::deinitialize()**.

Query Display Configurations

In the context of your description, it appears that the **vcu_video_lib** uses the **libdrm** library to validate the resolution's support by the monitor and to query the native resolution of the monitor. Let's break down the key components and their roles:

- **libdrm Library:**

    - **libdrm** is a user-space library that interacts with the Direct Rendering Manager (DRM) subsystem of the Linux kernel. It provides an interface for user-space programs to communicate with graphics drivers.
    - In the described scenario, **libdrm** is used for resolution validation and querying the native resolution of the monitor.
- **Resolution Validation:**
    - When configuring display settings, it's essential to ensure that the chosen resolution is supported by the connected monitor. This involves checking whether the graphics hardware, drivers, and the monitor itself can handle the specified resolution.
    - **libdrm** likely provides functions or mechanisms to perform this resolution validation against the capabilities of the graphics hardware and the connected display.
- **Query Native Resolution:**
    - The native resolution of a monitor refers to the physical resolution at which the monitor operates optimally. It's the resolution at which each pixel of the source content maps directly to a pixel on the monitor.
- **libdrm** is used to query the native resolution of the monitor, providing information about the optimal display settings.
- **Graphics Plane Configuration:**
    - The graphics plane, in the context of the Qt EGLFS backend, represents the hardware layer responsible for rendering graphics. The configuration of this plane involves setting up parameters such as resolution, pixel format, and possibly other display attributes.
    - The pixel format for each of the two planes is configured statically in the device tree. This means that the format in which pixel data is stored and transmitted between different components (e.g., GPU, display) is predetermined and set in the system's device tree configuration.
- **Qt EGLFS Backend:**
    - EGLFS is a plugin for Qt that allows applications to run directly on the framebuffer, bypassing the window system. It's designed for embedded systems with limited resources.
    - The backend configures the graphics plane outside of the **vcu_video_lib** library. It's responsible for handling the low-level graphics setup, including interfacing with the GPU and managing the framebuffer.

In summary, the described process involves using **libdrm** for resolution validation and querying the native resolution of the monitor. The Qt EGLFS backend configures the graphics plane, and the pixel format for each plane is statically set in the device tree. This ensures that the display settings align with the capabilities of the monitor and the graphics hardware.

Media Pipeline Configuration

The description pertains to the configuration of a video capture pipeline in a multimedia system, incorporating various input sources like TPG (Test Pattern Generator), HDMI, MIPI,

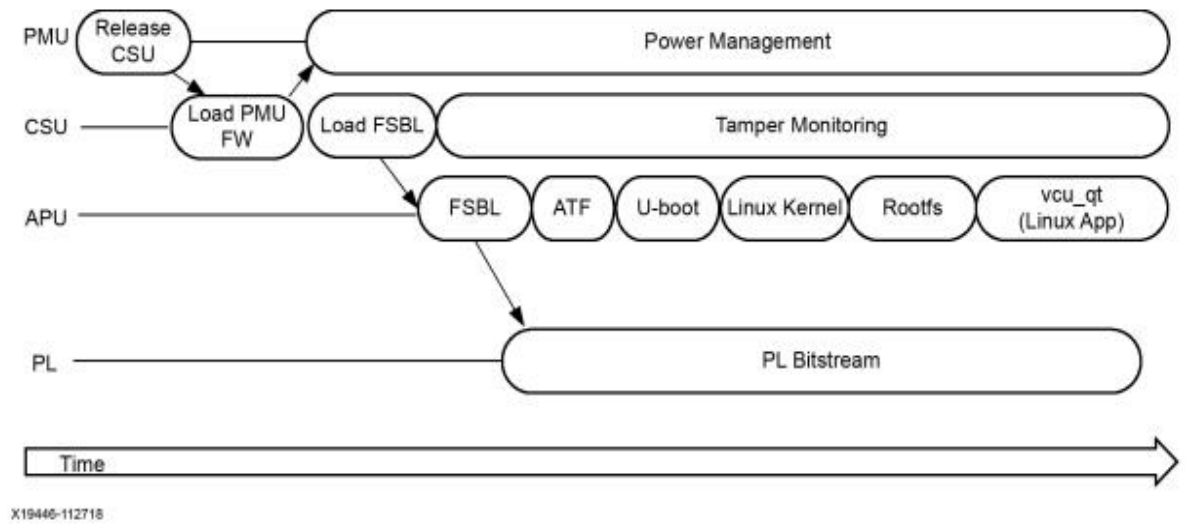
SDI, and SCD (Scene Change Detection). Let's break down the key components and processes:

- **Video Capture Pipeline:**
  - The video capture pipeline is a sequence of processing stages and devices responsible for capturing and handling video data from different sources. In this case, the pipeline includes various input sources such as TPG, HDMI, MIPI, SDI, and SCD.
- **Media Controller Interface:**
  - The media controller interface provides a unified way to configure the media pipeline and its sub-devices. It abstracts the control and configuration of multimedia devices and allows for seamless interaction with different components in the pipeline.
- **Libmediactl and Libv4l2subdev Libraries:**
  - These libraries, namely libmediactl and libv4l2subdev, are utilized to interact with the media controller interface and configure sub-devices in the pipeline.
- **Functionality Provided:**
  - *Enumerate Entities, Pads, and Links:* The ability to list and identify entities (basic building blocks), pads (connection endpoints), and links (connections between pads) in the media pipeline.
  - *Configure Sub-Devices:*
  - **Set Media Bus Format:** Specify the format of the media bus, which defines how pixel data is represented and transmitted.
  - **Set Dimensions (Width/Height):** Define the resolution or dimensions (width and height) of the video frames.
- **Video_Lib Library:**
  - The video_lib library is responsible for configuring the media pipeline by setting the media bus format and video resolution on each sub-device source and sink pad.
- **Media Bus Format Matching:** It's highlighted that the formats between pads connected through links need to match. This is crucial for ensuring that the data can seamlessly flow through the pipeline without compatibility issues.

In summary, the described system utilizes the media controller interface and associated libraries to configure a video capture pipeline. The configuration involves specifying the media bus format and video resolution for each sub-device in the pipeline. Ensuring matching formats between connected pads guarantees a smooth flow of video data through the pipeline, accommodating diverse input sources like TPG, HDMI, MIPI, SDI, and SCD.

Boot Process

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



Boot Process Sequence:

- **Power-On Reset (POR):**
  - The platform management unit (PMU) is the first to wake up after the power-on reset.
  - Responsible for primary pre-boot tasks and subsequent system power management.
- **PMU Pre-Configuration Stage:**
  - In the pre-configuration stage, the PMU executes its ROM (Read-Only Memory).
  - Releases the reset of the Configuration Security Unit (CSU).
- **PMU Server Mode:**
  - The PMU enters the PMU server mode after the initial boot process.
  - Responsible for handling clocks, resets, and system power management.
- **CSU Boot ROM Execution:**
  - The CSU, after being released from reset, executes its boot ROM.
  - Determines the boot mode by reading the boot mode register.
  - Initializes the on-chip memory (OCM) and reads the boot header.
- **PMU Firmware Execution:**
  - The CSU loads the PMU firmware into the PMU RAM.
  - Signals to the PMU to execute the firmware, providing advanced management features.
  - Loads the First Stage Boot Loader (FSBL) into OCM.
- **FSBL Execution on APU-0:**
  - The FSBL, executed on APU-0, initializes the Processing System (PS).
  - Configures the Programmable Logic (PL) and APU based on boot image header information.
- **PL Configuration:**
  - Configures the PL with a bitstream, and the PL reset is deasserted.
- **Arm Trusted Firmware (ATF) Execution:**
  - ATF is loaded into OCM and executed on APU-0.
  - Responsible for setting up secure execution environments and configuring the system.

- **U-Boot Loading into DDR:**
    - The second stage boot loader, U-Boot, is loaded into DDR (Double Data Rate memory).
    - APU-0 will execute U-Boot, which is a versatile bootloader used in embedded systems.

Summary:

- The boot process begins with the PMU handling pre-boot tasks and continues to manage system elements.
- The CSU executes its boot ROM, determining boot mode and initializing memory.
- PMU firmware is loaded, providing advanced features, and FSBL is loaded for APU-0 initialization.
- PL is configured, ATF is executed, and U-Boot is loaded into DDR for further system setup.

This sequence ensures a systematic and controlled boot process, allowing each component to play a specific role in initializing and configuring the system for subsequent stages. The use of multiple stages provides flexibility and modularity in the boot process.

Detailed Explaination of every step:

Power-On Reset (POR):

- **Initiation of System:**
    - When the system is powered on or undergoes a reset, the Power-On Reset (POR) phase is triggered.
- **Role of the Platform Management Unit (PMU):**
    - The Platform Management Unit (PMU) is a dedicated component designed to handle various tasks related to system initialization and management.
- **First Unit to Wake Up:**
    - After the power-on reset event, the PMU is the very first unit to wake up and become active.
    - This early activation is crucial for initiating and organising the primary pre-boot tasks.
- **Primary Pre-Boot Tasks:**
    - During the POR phase, the PMU is responsible for executing essential pre-boot tasks.
    - These tasks may include configuring initial system parameters, setting up clocks, and initializing critical components needed for the subsequent boot stages.
- **System Power Management:**
    - In addition to pre-boot tasks, the PMU is responsible for managing system power.
    - This involves monitoring power-related aspects, handling power domains, and ensuring that the system operates within specified power constraints.
- **Clock and Reset Handling:**
    - The PMU plays a crucial role in managing clocks and resets during the early stages of system startup.

    - It ensures that the system components are brought to a known and consistent state for the subsequent boot process.
- **Transition to Pre-Configuration Stage:**
    - Following the initial pre-boot tasks, the PMU may transition into the pre-configuration stage.
    - In this stage, additional configurations and initialization steps are performed to prepare the system for subsequent boot components.
- **Overall System Initialization:**
    - The PMU's involvement in the POR phase contributes to the overall initialization of the system.
    - It sets the stage for the execution of subsequent boot components, such as the Configuration Security Unit (CSU), boot ROMs, and bootloader stages.

Summary:
The Power-On Reset (POR) phase, initiated when the system is powered on or experiences a reset, marks the beginning of the system's boot process. The PMU, as the first unit to wake up, takes on the responsibility of executing primary pre-boot tasks, managing system power, handling clocks and resets, and transitioning the system to subsequent boot stages. Its role is fundamental in establishing a stable and controlled environment for the system's initialization and boot sequence.

2.PMU Pre-Configuration Stage:

- **PMU Executes its ROM (Read-Only Memory):**
    - The PMU has a built-in Read-Only Memory (ROM) that contains firmware or code necessary for early-stage system initialization.
    - Executing the PMU ROM involves running the pre-defined code within the ROM to perform essential tasks required for system setup.
- **Release Reset of the Configuration Security Unit (CSU):**
    - The Configuration Security Unit (CSU) is a component responsible for handling configuration stages in the system.
    - During the pre-configuration stage, the PMU takes the action of releasing the reset signal for the CSU.
    - Releasing the reset means allowing the CSU to transition from a reset state to an operational state.

The execution of the PMU ROM and the release of the CSU reset are critical steps in preparing the system for subsequent boot stages. Here's what these steps accomplish:

- **PMU ROM Execution:**
    - The PMU ROM code likely includes low-level initialization routines and configurations needed to set up essential system parameters.
    - It may handle tasks related to clock configurations, power management, and other fundamental settings.
- **Release of CSU Reset:**

    - The CSU is involved in the configuration stages of the boot process, and releasing its reset allows it to become active.
    - The CSU is responsible for executing the boot ROM, which is a firmware component that plays a key role in determining the boot mode, initializing on-chip memory (OCM), and reading the boot header.

By executing the PMU ROM and releasing the CSU reset during the pre-configuration stage, the system establishes a foundation for subsequent boot components to operate. These actions ensure that the system progresses through the early boot stages in a controlled and coordinated manner, leading to the successful initialization and configuration of the entire system.

3.PMU Server Mode:

When the Platform Management Unit (PMU) enters the PMU Server Mode after the initial boot process, it takes on responsibilities related to managing clocks, resets, and system power. Here's a breakdown of what this step entails:

- **Entering PMU Server Mode:**
    - After the initial boot process, the PMU transitions into what is referred to as the "PMU Server Mode." This mode represents a state where the PMU takes on more advanced and ongoing management tasks.
- **Responsibilities in PMU Server Mode:**
    - **Handling Clocks:** The PMU, in Server Mode, is responsible for managing various clocks within the system. This involves controlling the frequency and distribution of clock signals to different components of the system. Efficient clock management is crucial for overall system performance.
    - **Handling Resets:** The PMU continues to manage resets, ensuring that components of the system are appropriately reset or released from reset as needed. Proper reset management is essential for maintaining the stability and reliability of the system.
    - **System Power Management:** The PMU in Server Mode is involved in system-level power management. This includes tasks such as monitoring power usage, adjusting power states of different components, and possibly implementing power-saving measures to optimize energy efficiency.
- **Ongoing System Monitoring:**
    - In PMU Server Mode, the PMU continuously monitors the state of the system. This monitoring may involve checking for any irregularities, responding to system events, and taking corrective actions to ensure smooth operation.
- **Coordinating Advanced Features:**
    - The transition to PMU Server Mode often involves handing off control from the PMU's Read-Only Memory (ROM) to more advanced firmware or software running on the PMU. This firmware or software may provide advanced management features beyond the basic tasks performed during the initial boot.

The entry into PMU Server Mode signifies a shift from the initial boot process, where primary pre-boot tasks were handled, to an ongoing operational state. In this mode, the PMU takes on a role as a central manager for critical aspects such as clocks, resets, and power. Its responsibilities include maintaining the health and efficiency of the system, responding to dynamic conditions, and ensuring that the system operates within specified parameters throughout its runtime.

4.CSU Boot ROM Execution:

The CSU (Configuration Security Unit) Boot ROM Execution involves specific steps carried out by the CSU component during the early stages of the system boot process. Here's a detailed explanation of each step:

- **CSU Released from Reset:**
  - When the system transitions from the pre-configuration stage, the CSU is released from its reset state. The release from reset is a crucial step in enabling the CSU to execute its boot ROM and perform subsequent boot-related tasks.
- **Execution of CSU Boot ROM:**
  - Once released from reset, the CSU proceeds to execute its boot ROM. The boot ROM is a small piece of firmware stored in non-volatile memory (ROM) within the CSU. It contains essential instructions for the early stages of the boot process.
- **Determining Boot Mode:**
  - During the execution of its boot ROM, the CSU reads the boot mode register. The boot mode register holds information about the selected boot mode for the system. The boot mode is a configuration that specifies where the system should load its initial instructions and data from.
- **Initializing On-Chip Memory (OCM):**
  - After determining the boot mode, the CSU initializes the on-chip memory (OCM). The OCM is a type of memory embedded within the chip, and its initialization is a preparatory step for loading and executing subsequent stages of the boot process.
- **Reading Boot Header:**
  - With the OCM initialized, the CSU reads the boot header. The boot header contains critical information about the structure and characteristics of the boot image, such as the locations of different components like the First Stage Boot Loader (FSBL) and other firmware.
- **Finally it is done :**
  - **Boot Mode Determination:** By reading the boot mode register, the CSU determines the selected boot mode, which could be SD boot mode, JTAG boot mode, or another specified mode. This information guides subsequent actions in the boot process.
  - **OCM Initialization:** Initializing the on-chip memory prepares a portion of memory for storing and executing code during the early boot stages. This is crucial for loading and executing subsequent bootloader stages and firmware.

- **Boot Header Reading:** The information obtained from the boot header is critical for understanding the structure of the boot image, including the location and format of subsequent stages in the boot process.

In summary, the CSU's Boot ROM Execution involves initializing key components, determining the boot mode, and gathering essential information from the boot header. These steps set the stage for the subsequent stages of the boot process, including the loading and execution of the FSBL and other firmware components.

5.PMU Firmware Execution:

The PMU Firmware Execution involves specific steps performed by the CSU (Configuration Security Unit) during the boot process. Let's break down each step for a detailed explanation:

- **Loading PMU Firmware into PMU RAM:**
  - After the CSU (Configuration Security Unit) has executed its boot ROM and performed initial boot-related tasks, it loads the PMU (Platform Management Unit) firmware into the PMU RAM. The PMU firmware is a piece of software that provides advanced management features and capabilities beyond what is available in the PMU ROM.
- **Signaling PMU to Execute Firmware:**
  - Once the PMU firmware has been loaded into the PMU RAM, the CSU signals the PMU to execute the loaded firmware. This involves passing control to the PMU firmware, allowing it to take over the management and control of various aspects of the platform.
- **Finally it is for :**
  - **Advanced Management Features:** The PMU firmware is designed to provide advanced management features beyond the capabilities of the PMU ROM. These features may include sophisticated power management, clock control, and other platform-specific functions.
  - **Customized Platform Behavior:** By loading and executing the PMU firmware, the system gains access to a set of customized functionalities and behaviors. This firmware can implement platform-specific policies, respond to external events, and manage resources dynamically.
  - **Transition from Basic Pre-Boot Tasks:** The execution of PMU firmware marks a transition from the basic pre-boot tasks, handled by the PMU ROM and early stages of boot loaders, to a more advanced and capable management state.
  - **Dynamic Power Management:** The PMU firmware is often responsible for dynamic power management, where it can adjust clock frequencies, put certain components into low-power states, and optimize power consumption based on the system's operational requirements.
  - **System-Level Control:** The PMU firmware's execution gives the platform more comprehensive control over its components. This includes managing clock domains, handling resets, and overseeing system-level configurations.

In summary, loading and executing the PMU firmware represent a critical step in the boot process, enabling the platform to move beyond basic pre-boot tasks and access advanced management features for dynamic control and customization. The PMU firmware plays a crucial role in tailoring the behavior of the system based on specific platform requirements and operational conditions.

6.FSBL Execution on APU-0:

The execution of the First Stage Boot Loader (FSBL) on APU-0 (Application Processing Unit 0) is a crucial phase in the boot process. Let's break down each step for a detailed explanation:

- **FSBL Execution on APU-0:**
  - The FSBL is a small piece of software responsible for managing the initial stages of the boot process. It is executed on the Application Processing Unit 0 (APU-0), which is part of the Processing System (PS) in the FPGA (Field-Programmable Gate Array) architecture.
- **FSBL Initialization of Processing System (PS):**
  - The primary responsibility of the FSBL is to initialize the Processing System (PS). The PS typically includes components like the CPU cores, memory controllers, peripheral interfaces, and other essential elements that make up the processing subsystem.
- **FSBL Configuration of Programmable Logic (PL) and APU:**
  - In addition to initializing the PS, the FSBL configures the Programmable Logic (PL) and APU (Application Processing Unit) based on information provided in the boot image header. The PL is the part of the FPGA that can be dynamically reconfigured to implement custom logic, and the APU is a processing unit that executes application code.

- **Finally, it is for:**
  - **System Initialization:** The FSBL plays a crucial role in the system's initialization by setting up and configuring the PS. This involves configuring the CPU cores, memory controllers, and other components to establish a functional and operational state.
  - **PL Configuration:** The FSBL configures the Programmable Logic (PL) portion of the FPGA based on the boot image header information. This configuration involves specifying the functionality of the PL, which can include loading a bitstream to define the custom logic implemented in the FPGA.
  - **APU Configuration:** The APU, being part of the PS, is configured by the FSBL. This configuration may include setting up the processor cores, cache, and other parameters necessary for the APU to execute application code.
  - **Boot Image Header Information:** The FSBL relies on information stored in the boot image header to determine how to initialize and configure the system. This header contains metadata and configuration details needed for a successful boot.

- **Transition to Next Boot Stages:** Once the FSBL has completed its tasks, it facilitates the transition to subsequent boot loader stages, such as the Arm Trusted Firmware (ATF) and the second stage boot loader (U-Boot), which continue the boot process and load the operating system.

In summary, the execution of the FSBL on APU-0 is a critical step that sets the foundation for the overall boot process. It initializes the system, configures the FPGA, and prepares the APU for subsequent stages in the boot sequence. The FSBL ensures that the system transitions from a powered-off or reset state to a state where more advanced software components can take over.

7.PL Configuration:

The step "Configures the PL with a bitstream, and the PL reset is deasserted" refers to the configuration of the Programmable Logic (PL) portion of the FPGA (Field-Programmable Gate Array) during the boot process. Let's break down what this step entails:

- **Configures the PL with a Bitstream:**
  - The PL in an FPGA is a region that can be dynamically reconfigured to implement custom logic circuits. The configuration data, known as a bitstream, is a binary file that describes the specific configuration of the logic elements, interconnections, and other parameters within the PL.
  - During this step, the bitstream is loaded into the PL. This bitstream essentially defines the desired functionality of the PL, specifying how the programmable logic resources should be interconnected to perform specific tasks or functions.
- **PL Reset Deassertion:**
  - After the PL is configured with the desired bitstream, the PL reset is deasserted. Asserting a reset means putting the respective part of the system (in this case, the PL) into a known, initial state. Deasserting the reset releases the PL from this initial state and allows it to operate based on the newly loaded configuration.

- **Finally, it is for :**
  - **Custom Logic Implementation:** The configuration of the PL with a specific bitstream allows the FPGA to implement custom logic circuits tailored to the requirements of the application or system. The bitstream essentially defines the digital circuitry, interconnections, and behavior of the logic within the PL.
  - **Dynamic Reconfiguration:** FPGAs provide the unique capability of dynamic reconfiguration, allowing the PL to be reprogrammed while the rest of the system is running. This flexibility is valuable in applications where different tasks or functions need to be executed at different times, and reconfiguration enables adapting to changing requirements.
  - **Initialization Completion:** The deassertion of the PL reset signifies that the PL is ready to operate according to the newly loaded configuration. At this point, the PL becomes an active part of the system, contributing to the overall functionality and processing capabilities.

- **Transition to Next Boot Stages:** The successful configuration of the PL is a crucial step in the overall boot process. It sets the stage for subsequent boot loader stages, allowing for the execution of more advanced software components and the eventual loading of the operating system.

In summary, the PL configuration involves loading a bitstream to define custom logic in the FPGA's programmable logic region. Deasserting the PL reset allows the PL to operate based on this newly configured logic, contributing to the overall functionality of the FPGA-based system.

8.Arm Trusted Firmware (ATF) Execution:

The step "Arm Trusted Firmware (ATF) Execution: ATF is loaded into OCM and executed on APU-0" refers to the execution of the Arm Trusted Firmware (ATF) on the Application Processing Unit 0 (APU-0) in the boot process. Let's break down what this step entails:

- **ATF Loading into OCM (On-Chip Memory):**
  - The ATF, which is a set of boot firmware components designed to establish a secure execution environment, is loaded into the OCM. The OCM is a type of memory that is often located on the FPGA and provides fast and direct access for the processing units.
- **Execution on APU-0:**
  - After loading into the OCM, the ATF is executed on APU-0. The Application Processing Unit (APU) is typically a part of the Processing System (PS) in an FPGA and is responsible for general-purpose processing tasks.
- **Responsible for Setting up Secure Execution Environments:**
  - ATF plays a crucial role in establishing a secure execution environment on the system. This involves configuring various security features and initializing the necessary components to ensure a secure boot process.
- **Configuring the System:**
  - ATF is responsible for configuring different aspects of the system, including the processor, memory, and security features. It sets up the system to meet the security requirements and prepares it for subsequent stages of the boot process.

- **Finally, it is for:**
  - **Secure Boot Initialization:** ATF is designed to support secure boot processes, ensuring that the system boots securely and establishes a trusted execution environment.
  - **Security Configuration:** ATF configures security features such as TrustZone, which provides a mechanism for isolating secure and non-secure code execution. This helps in creating a secure execution environment on the FPGA.
  - **System Initialization:** ATF initializes various system components, ensuring that the processor and memory subsystems are properly configured for subsequent boot loader stages.

- **Handover to U-Boot:** ATF typically hands over control to the next stage in the boot process, which is U-Boot (the second stage boot loader). U-Boot takes over the bootstrapping process and continues with loading the operating system or other software components.

In summary, the execution of ATF on APU-0 involves loading ATF into OCM, setting up a secure execution environment, configuring security features, and initializing the system for subsequent boot loader stages. ATF plays a critical role in the secure boot process of the FPGA-based system.

9.U-Boot Loading into DDR:

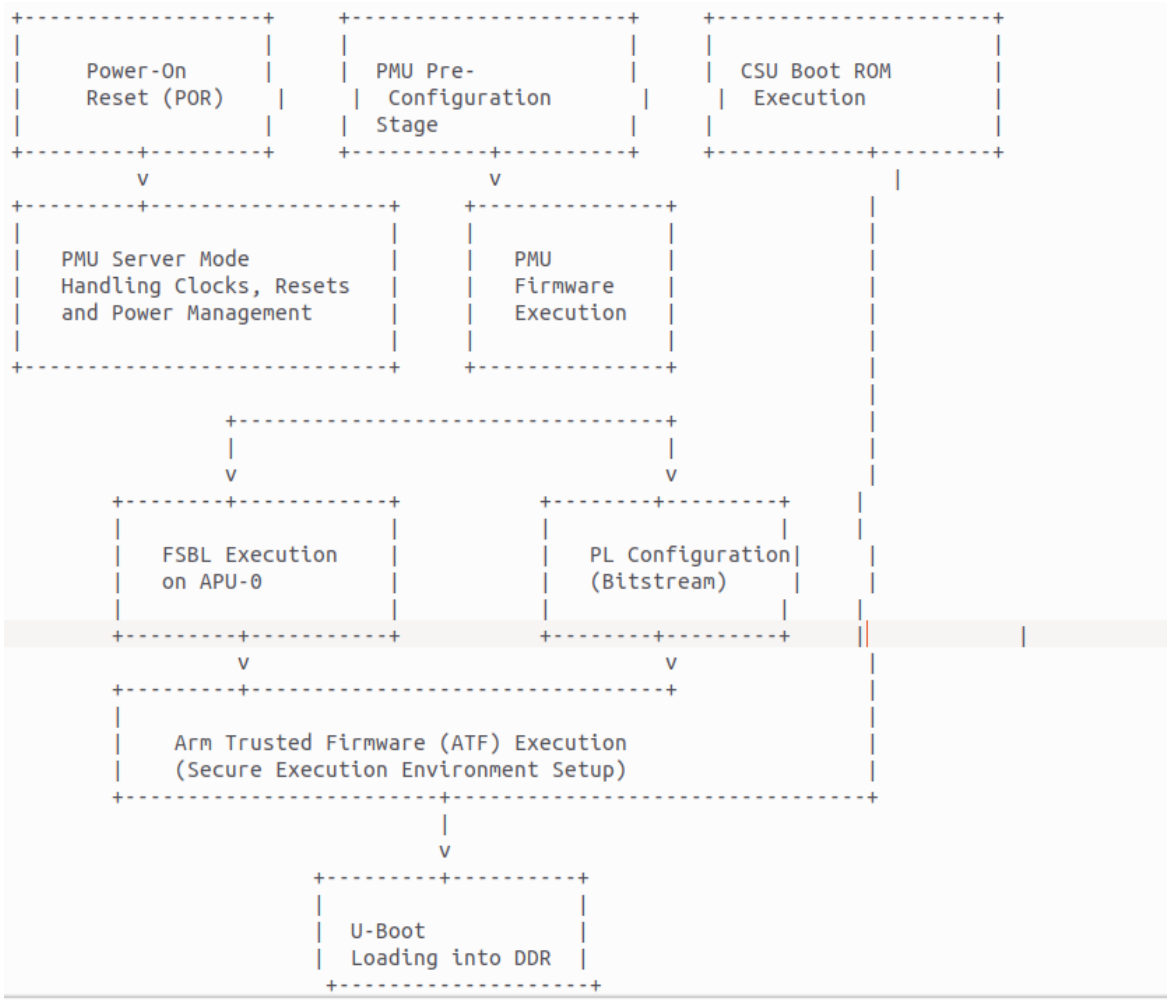
The step "U-Boot Loading into DDR: The second stage boot loader, U-Boot, is loaded into DDR (Double Data Rate memory) to be executed by APU-0" refers to the process of loading the U-Boot bootloader into the main memory, specifically the Double Data Rate (DDR) memory, and preparing it for execution by the Application Processing Unit 0 (APU-0). Let's break down what this step entails:

- **U-Boot Loading into DDR:**
  - U-Boot, being the second stage bootloader in the boot process, is loaded from a storage device (commonly an SD card or other non-volatile memory) into the DDR memory. DDR memory is a type of volatile memory used for storing data and code that is accessible by the processor.
- **Execution by APU-0:**
  - Once U-Boot is successfully loaded into DDR, control is transferred to APU-0, and the processor begins executing the instructions stored in U-Boot. APU-0 is responsible for general-purpose processing tasks and is part of the Processing System (PS) in an FPGA.
- **U-Boot's Role:**
  - U-Boot (Universal Bootloader) is a widely used and flexible bootloader in embedded systems. Its primary role is to initialize the hardware, configure the system, and load the operating system kernel or other software components into memory for execution.
- **Hardware Initialization:**
  - U-Boot performs initialization tasks for various hardware components, including memory, peripherals, and other system-specific configurations. This ensures that the system is properly set up for subsequent stages of the boot process.
- **User Interaction and Configuration:**
  - U-Boot often provides a command-line interface, allowing users to interact with and configure the bootloader. Users can use U-Boot to modify boot parameters, select different boot options, or perform diagnostic tasks.
- **Loading Operating System or Kernel:**
  - U-Boot's main responsibility is to load the operating system kernel or other software components into memory. It typically supports loading from different storage media, such as SD cards, NAND flash, or network storage.

- **Finally, it is for:**
    - **Prepare for OS Loading:** Loading U-Boot into DDR is a crucial step in the boot process as it prepares the system for loading the operating system or kernel. U-Boot serves as an intermediary between the hardware initialization stage and the execution of the main software components.
    - **Hardware Configuration:** U-Boot handles the initialization and configuration of essential hardware components, ensuring that the system is in a suitable state for the subsequent stages of the boot process.
    - **User Interaction:** U-Boot provides a user-friendly interface that allows users to interact with the bootloader, configure boot parameters, and perform troubleshooting or maintenance tasks.
    - **Handover to Operating System:** Once U-Boot completes its tasks, it hands over control to the operating system kernel or the next stage in the boot process, facilitating the execution of the main software components.

In summary, the loading of U-Boot into DDR marks a critical phase in the boot process, where the bootloader prepares the system for the execution of the operating system or other software components on the FPGA-based system.

Flow chart


Power-On Reset (POR)
PMU Pre-Configuration Stage
CSU Boot ROM Execution
PMU Server Mode Handling Clocks, Resets and Power Management
PMU Firmware Execution
FSBL Execution on APU-0
PL Configuration (Bitstream)
Arm Trusted Firmware (ATF) Execution (Secure Execution Environment Setup)
U-Boot Loading into DDR