StarFive 赛昉科技

JH7110 GPU Developing and Porting Guide

VisionFive 2

Version: 1.0

Date: 2022/12/30

Doc ID: JH7110-DGEN-015

Legal Statements

Important legal notice before reading this documentation.

PROPRIETARY NOTICE

Information in this document is provided "as is," with all faults. Contents may be periodically updated or revised due to product development. Shanghai StarFive Technology Co., Ltd. (hereinafter "StarFive") reserves the right to make changes without further notice to any products herein.

StarFive expressly disclaims all warranties, representations, and conditions of any kind, whether express or implied, including, but not limited to, the implied warranties or conditions of merchantability, fitness for a particular purpose, and non-infringement.

StarFive does not assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation indirect, incidental, special, exemplary, or consequential damages.

Contact Us

Address: Room 502, Building 2, No. 61 Shengxia Rd., China (Shanghai) Pilot Free Trade Zone, Shanghai, 201203, China

Website: http://www.starfivetech.com

Email:

- Sales: sales@starfivetech.com
- Support: support@starfivetech.com

Contents

List of Tables

List of Figures

Preface

About this guide and technical support information.

About this document

This document mainly provides the SDK developers and porting administrators with the programing basics and debugging know-how for GPU and graphics of the StarFive next generation SoC platform - JH7110.

Audience

This document mainly serves the GPU and graphics relevant driver developers. If you are developing other modules, place a request to your sales or support consultant for our complete documentation set on JH7110.

Revision History

Table 0-1 Revision History

Version	Released	Revision
1.0	2022/12/30	First official release.

Notes and notices

The following notes and notices might appear in this guide:

- **Tip:**
  Suggests how to apply the information in a topic or step.
- **Note:**
  Explains a special case or expands on an important point.
- **Important:**
  Points out critical information concerning a topic or step.
- **CAUTION:**
  Indicates that an action or step can cause loss of data, security problems, or performance issues.
- **Warning:**
  Indicates that an action or step can result in physical harm or cause damage to hardware.

1. Introduction

1.1. Overview

The graphics module on the StarFive Linux platform is a RISC-V Linux platform using a GPU module. The advantage is that the general architecture is easy to customize, and many existing components can be used. The development of many existing basic open source projects has begun to use StarFive platform as the RISC-V compatible platform. But the disadvantage is that RISC-V is new and the ecosystem is to be developed. Thus there are not as many practical applications, and a lot of them are still on the way.

This GPU package contains source code of several graphics' examples for OpenGL ES 3.2 and OpenGL ES 1.1 x11 API, Framebuffer, and XWayland graphical back-ends. These applications show that the graphics acceleration is working for different APIs. The package includes samples, demo code, and documentation for working with the JH7110 family of graphics cores.

1.2. Feature Introduction

The StarFive JH7110 GPU module provides a complete graphics acceleration platform based on open standards, supporting 3D and GPGPU calculations. The StarFive Linux GPU provides OpenGL ES, EGL, and OpenCL API, but does not support OpenGL. The supported types are as follows:

- OpenGL ES 3.2
- OpenGL ES 1.1
- Vulkan 1.2
- OpenCL 1.2

2. Software Modules

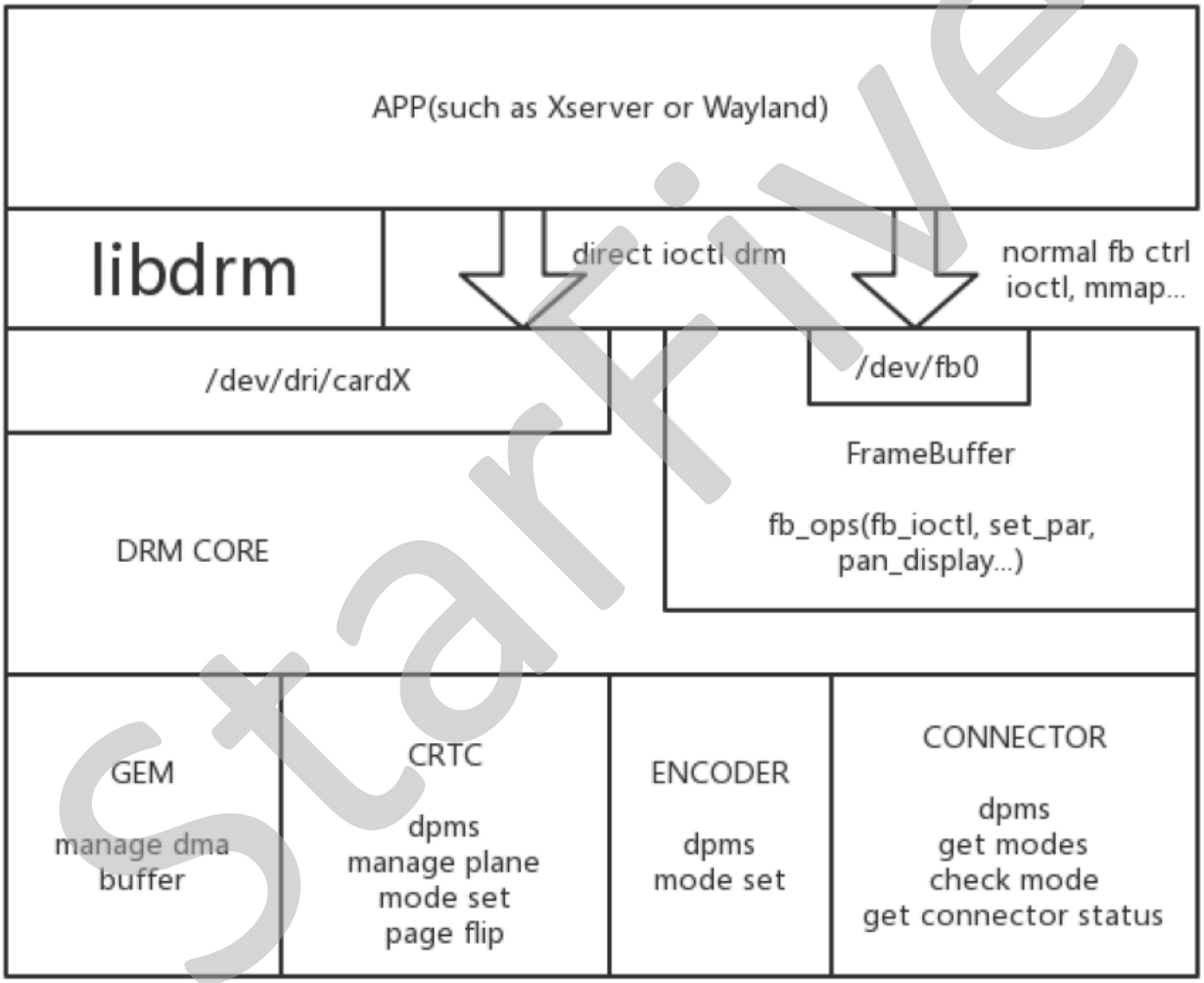
For GPU software modules, you mainly have to understand the relationships between LIBDRM, wayland and the x11 compositor - mesa.

2.1. LIBDRM

LIBDRM is a cross-driver middle-ware that enables user space applications (for example, Mesa and 2D drivers) to communicate with the kernel through DRI.

The following diagram shows the DRM structure.

Figure 2-1 DRM Structure



LIBDRM is a library under DRM for communication between driver and user layer.

2.2. X11

X11 is an Open Source version of the X Window System that supports many UNIX® and UNIX-like operating systems on a variety of platforms. X11 is similar to general desktop platforms, with weak GPU performance problems. To run it, simply by typing `Xorg`.

Xorg is a full featured X server and supports several mechanisms for supplying/obtaining configuration and run-time parameters: command line options, environment variables, the `xorg.conf` configuration files, auto-detection, and fallback

defaults. Typically we uses a configuration file called `xorg.conf` and configuration files with the `suffix .conf` in a directory called `xorg.conf.d` for its initial setup.

2.3. Wayland

It is recommended to use Yocto Buildroot SDK for Wayland development. Wayland is better than X11 in efficiency, mainly due to compatibility issues. If you need multiple windows instead of a desktop, you can try Wayland.

Part of the Wayland project is also the Weston reference implementation of a Wayland compositor. Weston can run as an X client or under Linux KMS and ships with a few demo clients. The Weston compositor is a minimal and fast compositor and is suitable for many embedded and mobile use cases. Most users can simply run it by typing `weston`. This will launch Weston inside whatever environment you launch it from: when launched from a text console, it will take over that console. When launched from inside an existing Wayland or X11 session, it will start a 'nested' instance of Weston inside a window in that session. Help is available by running `weston --help`, or `man weston`, which will list the available configuration options and display back-ends. It can also be configured through a file on disk; The configuration file is called `weston.ini` for its setup.

2.4. NULLDRMWS

3. Display Architecture

The JH7110 GPU module has the following display architecture options.

- X11
- Wayland
- NULLDRMWS

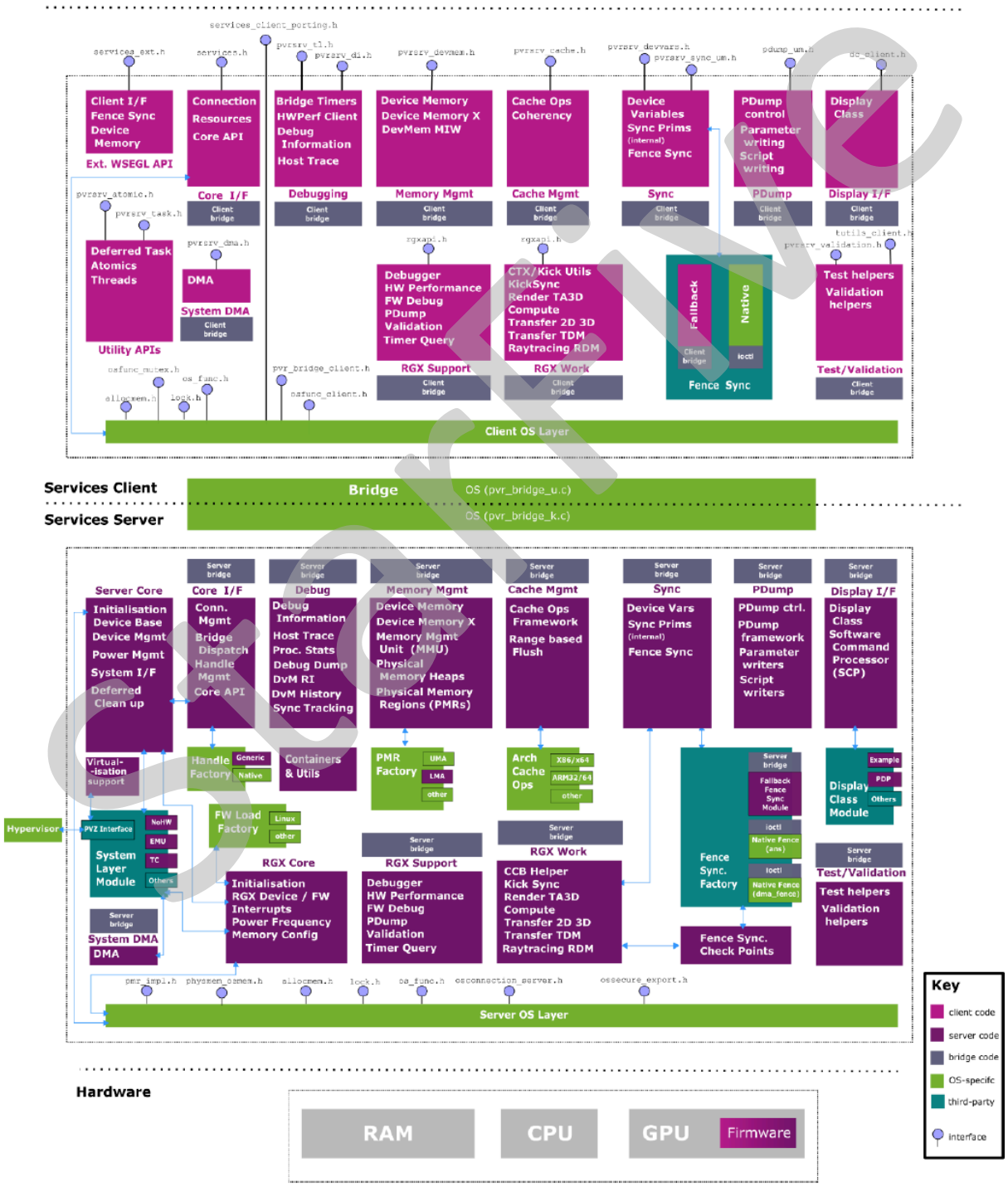
4. Porting

4.1. Service Architecture

The Services software stack is split between ‘client’ services and ‘server’ services.

Services Client should not reside in the privileged Kernel domain, whereas Services Server needs to reside in the privileged Kernel domain (however that may be achieved in the target OS).

Figure 4-1 Service Architecture



4.2. System Abstraction

This section describes all aspects of the System (or SoC) abstraction layer that allows a single common code base to be adapted to JH7110 containing a StarFive GPU core. The System layer was designed with a goal to minimise the differences between JH7110 code and the system porting effort. The System layer is a software abstraction layer between the hardware and the rest of the GPU source code.

4.2.1. System Porting Files

When porting Services to a new SoC the first thing is to create a new system folder.

```
services/system/<new system name>
```

Copy files from a reference SoC to this folder, typically:

```
sysconfig.c
sysconfig.h
sysinfo.h
```

4.2.1.1. Sysconfig.h

In this header the default GPU clock frequency must be specified (in MHZ), for example, 100 MHz.

The following code block provides an example.

```
define RGX_NOHW_CORE_CLOCK_SPEED 100000000
```

4.2.1.2. Sysconfig.c

Generally, sysconfig.c encapsulates all the JH7110 system specific code. The functions in this file are used to define the system attributes.

4.2.1.3. Sysinfo.h

In this header, the frequency with which the Device Health Check watchdog is called is specified (in ms).

```
#define DEVICES_WATCHDOG_POWER_ON_SLEEP_TIMEOUT (1500)
```

Part of the Device Health Check watchdog's behaviour is to sample the buffer read/write offsets for all active Firmware contexts to detect if a context has not progressed within the **DEVICES_WATCHDOG_POWER_ON_SLEEP_TIMEOUT** period (despite the device being idle). If this is detected then a debug message is written to the kernel log and the driver will attempt to 'unblock' the stalled context by force signalling the fence condition which is preventing the context from progressing. This behaviour is known as Sync Lockup Recovery (SLR).

If the **DEVICES_WATCHDOG_POWER_ON_SLEEP_TIMEOUT** is too long (For example, more than 5 seconds), a context will have to be in a stalled state for a longer time before SLR is actioned, reducing the benefit of the SLR feature.

4.2.2. System Initialization

4.2.2.1. SysDevInit

It is necessary to provide an implementation of the following function:

```
PVRSRV_ERROR SysDevInit(void *pvOSDevice, PVRSRV_DEVICE_CONFIG **ppsDevConfig)
```

The **SysDevInit()** function is called by the GPU Services Server device create function **(PVRSRVDeviceCreate())**, via a per-OS call, for each device found in the system. This function must do any SoC specific device initialisation as well as allocating and setting up a number of data structures for the given **pvOSDevice**. These structures are presented in the following sub-sections.

4.2.2.2. PVRSRV_DEVICE_CONFIG

PVRSRV_DEVICE_CONFIG is the primary data structure setup by the **SysDevInit()** function.

```
struct _PVRSRV_DEVICE_CONFIG_
{
/*! OS device passed to SysDevInit (linux: 'struct device') */
void *pvOSDevice;
/*!
*! Service representation of pvOSDevice. Should be set to NULL when the
*! config is created in SysDevInit. Set by Services once a device node has
*! been created for this config and unset before SysDevDeInit is called.
*/
struct _PVRSRV_DEVICE_NODE_ *psDevNode;
/*! Name of the device */
IMG_CHAR *pszName; // typically the name of the SoC
/*! Version of the device (optional) */
IMG_CHAR *pszVersion;
/*! Register bank address */
IMG_CPU_PHYADDR sRegsCpuPBase; // physical address of the GPU register bank
/*! Register bank size */
IMG_UINT32 ui32RegsSize; // size in bytes of the GPU register bank
/*! Device interrupt number */
IMG_UINT32 ui32IRQ; // IRQ number for the GPU device
PVRSRV_DEVICE_SNOOP_MODE eCacheSnoopingMode;
/*! Device specific data handle */
IMG_HANDLE hDevData; // handle (actually a pointer) to RGX_DATA
/*! System specific data. This gets passed into system callback func-tions */
IMG_HANDLE hSysData; // handle (actually a pointer) to optional SoC specific device data
IMG_BOOL bHasNonMappableLocalMemory;
/*! Indicates if system supports FBCDC v3.1 */
IMG_BOOL bHasFBCDCVersion31;
/*! Physical Heap definitions for this device.
* eDefaultHeap must be set to GPU_LOCAL or CPU_LOCAL. Specifying any other value
* (e.g. DEFAULT) will lead to an error at device discovery.
* pasPhysHeap array must contain at least one PhysHeap, the declared default heap.
*/
PVRSRV_PHYS_HEAP eDefaultHeap;
PHYS_HEAP_CONFIG *pasPhysHeaps; // pointer to array of physical heap configs
IMG_UINT32 ui32PhysHeapCount; // physical heap count
/*!
*! Callbacks to change system device power state at the beginning and end
*! of a power state change (optional).
*/
PFN_SYS_PRE_POWER pfnPrePowerState; // should perform SoC specific set-up required pre-power
transition for this device
PFN_SYS_POST_POWER pfnPostPowerState; // should perform SoC specific setup that is required
post-power transition for this device
PFN_SYS_GET_POWER pfnGpuDomainPower; // should query the state of the SoC power domain
which supplies the GPU
/*! Callback to obtain the clock frequency from the device (optional) */
PFN_SYS_DEV_CLK_FREQ_GET pfnClockFreqGet; // add this call back the current core clock
frequency (note: DFVS can change clock frequency dynamically
#if defined(SUPPORT_SOC_TIMER)
/*! Callback to read SoC timer register value (mandatory). */
PFN_SYS_DEV_SOC_TIMER_READ pfnSoCTimerRead;
#endif
Confidential Imagination Technologies
PowerVR Graphics GPU Services 11 Revision 1.17.83
/*!
*! Callback to handle memory budgeting. Can be used to reject alloca-tions
*! over a certain size (optional).
*/
PFN_SYS_DEV_CHECK_MEM_ALLOC_SIZE pfnCheckMemAllocSize;
/*!
```

```
*! Callback to perform host CPU cache maintenance. Might be needed for
*! architectures which allow extensions such as RISC-V (optional).
*/
PFN_SYS_DEV_HOST_CACHE_MAINTENANCE pfnHostCacheMaintenance;
IMG_BOOL bHasPhysicalCacheMaintenance;
#if defined(SUPPORT_TRUSTED_DEVICE)
/*!
*! Callback to send FW image and FW boot time parameters to the trusted
*! device.
*/
PFN_TD_SEND_FW_IMAGE pfnTDSendFWImage;
/*!
*! Callback to send parameters needed in a power transition to the trusted
*! device.
*/
PFN_TD_SET_POWER_PARAMS pfnTDSetPowerParams;
/*! Callbacks to ping the trusted device to securely run RGXStart/Stop() */
PFN_TD_RGXSTART pfnTDRGXStart;
PFN_TD_RGXSTOP pfnTDRGXStop;
#endif /* defined(SUPPORT_TRUSTED_DEVICE) */
/*! Function that does device feature specific system layer initialisa-tion */
PFN_SYS_DEV_FEAT_DEP_INIT pfnSysDevFeatureDepInit;
#if defined(SUPPORT_LINUX_DVFS) || defined(SUPPORT_PDVS)
PVRSRV_DVFS sDVFS; // Used by the IMG DVFS governor
#endif
#if defined(SUPPORT_ALT_REGBASE)
IMG_DEV_PHYADDR sAltRegsGpuPBase;
#endif
/*!
*! Indicates if device physical address 0x0 might be used as GPU memory
*! (e.g. LMA system or UMA system with CPU PA 0x0 reserved by the OS,
*! but CPU PA != device PA and device PA 0x0 available for the GPU)
*/
IMG_BOOL bDevicePA0IsValid;
/*!
*! Callback to notify system layer of device errors.
*! NB. implementers should ensure that the minimal amount of work is
*! done in the callback function, as it will be executed in the main
*! RGX MISR. (e.g. any blocking or lengthy work should be performed by
*! a worker queue/thread instead.)
*/
PFN_SYS_DEV_ERROR_NOTIFY pfnSysDevErrorNotify;
Confidential Imagination Technologies
PowerVR Graphics GPU Services 12 Revision 1.17.83
/*!
*! Slave DMA channel request callbacks
*/
PFN_SLAVE_DMA_CHAN pfnSlaveDMAGetChan;
PFN_SLAVE_DMA_FREE pfnSlaveDMAFreeChan;
/*!
*! Conversion of device memory to DMA addresses
*/
PFN_DEV_PHY_ADDR_2_DMA_ADDR pfnDevPhysAddr2DmaAddr;
/*!
*! DMA channel names
*/
IMG_CHAR *pszDmaTxChanName;
IMG_CHAR *pszDmaRxChanName;
/*!
*! DMA device transfer restrictions
*/
IMG_UINT32 ui32DmaAlignment;
IMG_UINT32 ui32DmaTransferUnit;
/*!
*! System-wide presence of DMA capabilities
*/
IMG_BOOL bHasDma;
};
```

In the above code block:

eCacheSnoopingMode should be specified as follows:

- **PVRSRV_DEVICE_SNOOP_NONE** - This means that neither the CPU nor GPU supports having their caches snooped.
- **PVRSRV_DEVICE_SNOOP_CPU_ONLY** - This means that the CPU will allow the GPU to inspect the CPU cache. The GPU cache is not visible to the CPU.
- **PVRSRV_DEVICE_SNOOP_DEVICE_ONLY** - This means that the GPU will allow the CPU to inspect the GPU cache. The CPU cache is not visible to the GPU.
- **PVRSRV_DEVICE_SNOOP_CROSS** - This means that both CPU and GPU support cache snooping. They can each inspect the others caches.
- **PVRSRV_DEVICE_SNOOP_EMULATED** - This means that neither the CPU nor GPU supports having their caches snooped but the GPU software/firmware should emulate snooping by maintaining the CPU and GPU data caches appropriately.

eDefaultHeap determines which heap to use for the majority of device memory allocations. The server will allocate from the default heap unless a physical heap is specified in the allocation.

eDefaultHeap should be specified as follows:

- **PVRSRV_PHYS_HEAP_GPU_LOCAL** - This means that the device allocations will be fulfilled with memory that is physically (in terms of access time and throughput) closest to the GPU.
- **PVRSRV_PHYS_HEAP_CPU_LOCAL** - This means that the device allocations will be fulfilled with memory that is physically (in terms of access time and throughput) closest to the CPU. **eDefaultHeap** must be set to a physical heap provided in the **pasPhysHeaps** array member. **bDevicePA0IsValid** should be set to **IMG_TRUE** if device physical address 0x0 is available and can be accessed by the GPU, **IMG_FALSE** otherwise.

Examples of systems where device physical address 0x0 is valid:

- LMA with local memory mapped in the GPU from physical address 0x0.
- UMA with CPU physical address != device physical address and 0x0 is a valid device address.

Examples of systems where device physical address 0x0 is not valid:

- Regular UMA with device and CPU physical address 0x0 reserved by the OS.
- UMA with a memory protection unit forbidding GPU accesses outside of specific ranges.

When System DMA is available, the following fields should be specified:

- **pfnSlaveDMAGetChan** and **pfnSlaveDMAFreeChan**: Should return and free DMA Engine Channels by their DMA driver-relative names. Upper GPU Service layers will also make use of the fields right below when calling these two functions.
- **pszDmaTxChanName** and **pszDmaRxChanName**: System DMA Engine driver names for send and receive channels. The above assignments will depend on these two, and should be set to whatever channel names for the DMA Engine driver the customer wants to connect with the Services DMA API.
- **pfnDevPhysAddr2DmaAddr**: Should perform the translation between RGX device physical addresses and DMA addresses that the underlying DMA device understands. Necessary for mapping DMA buffers before a SG transfer and programming the DMA hardware.
- **ui32DmaAlignment** and **ui32DmaTransferUnit** are used if the hardware has alignment and transfer size restrictions. **ui32DmaAlignment** is in bytes and refers to the byte alignment required at the start of a transfer buffer. **ui32DmaTransferUnit** is in bytes and refers to the minimum size of transfer supported by the hardware.
- **bHasDma**: Informs the upper layers of the Services kernel driver that all of the above DMA functionality is actually present in the system.

pasPhysHeaps (PHYS_HEAP_CONFIG)

Structure used to describe a physical Heap supported by a system. A system layer module can declare multiple physical heaps for different purposes. At a minimum a system must provide one physical heap tagged for **PHYS_HEAP_USAGE_GPU_LOCAL**.

A heap represents a discrete pool of physical memory and how it is managed, as well as other associating properties and address translation logic.

4.2.2.3. SysDebugInfo

It is necessary to provide an implementation of the following function:

```
PVRSRV_ERROR SysDebugInfo(PVRSRV_DEVICE_CONFIG *psDevConfig,
DUMPDEBUG_PRINTF_FUNC *pfnDumpDebugPrintf,
void *pvDumpDebugFile)
```

The **SysDebugInfo()** function is called via the **PVRSRVDebugRequest()** function and is required to dump JH7110 specific information, using the provided print function, typically at the point of a failure. Examples include the JH7110 temperature printed out in the log:

```
PVR_LOG(("Chip temperature: %d degrees C", uiTemperature))
```

4.2.2.4. SysInstallDeviceLISR Function

It is necessary to provide an implementation of the following function:

```
PVRSRV_ERROR SysInstallDeviceLISR(IMG_HANDLE hSysData,
IMG_UINT32 ui32IRQ,
const IMG_CHAR *pszName,
PFN_LISR pfnLISR,
void *pvData,
IMG_HANDLE *phLISRData)
```

SysInstallDeviceLISR() is called by the GPU Services server device initialisation code in order to install a device interrupt handling function. Typically, it is sufficient to call the **OSInstallSystemLISR()** helper function.

4.2.2.5. SysUninstallDeviceLISR

It's necessary to provide an implementation of the following function:

```
PVRSRV_ERROR SysUninstallDeviceLISR(IMG_HANDLE hLISRData)
```

SysUninstallDeviceLISR() is called by the GPU Services server device de-initialisation code in order to uninstall the device interrupt handling function, which was installed by **SysInstallDeviceLISR()**. If the **OSInstallSystemLISR()** function was used previously then this should be done by calling the **OSUninstallSystemLISR()** helper function.

4.3. SoC with shared SLC

If the *System Level Cache (SLC)* in the SoC is shared between the GPU and other devices (e.g. video cores), it is necessary to build the GPU with the **SUPPORT_SHARED_SLC** build option. When this build option is used, the driver does not attempt to reset or initialize the SLC as part of the GPU initialization. Instead, the system layer must call the function **RGXInitSLC()** after the SLC has powered up and before any device attempts to use it.

4.4. GPU Interrupt Service Routines

Typically there is an interrupt service routine (ISR) for the GPU and another for the display hardware's VSync event. On completion of the port it is necessary to check that the ISR for the GPU is scheduled on completion of every GPU operation scheduled by the firmware. It is also necessary to check the VSync event on the display hardware additionally schedules an independent Vsync specific ISR for every VSync event.

These checks are very important since the platform can seem functional without one or both of these ISRs functioning correctly. This can result in a lengthy debug investigation weeks or months later and cause seemingly unrelated issues.

4.5. System DMA

The GPU can be built with support for DMA transfers between host and device memory. If the system integrator has provided a hardware DMA Engine driver to allow data transfers between system memory and device memory, then the GPU can be built to provide APIs to integrate with this. It is expected that the vendor will provide a DMA engine that conforms to the Linux Kernel DMA Engine API interface.

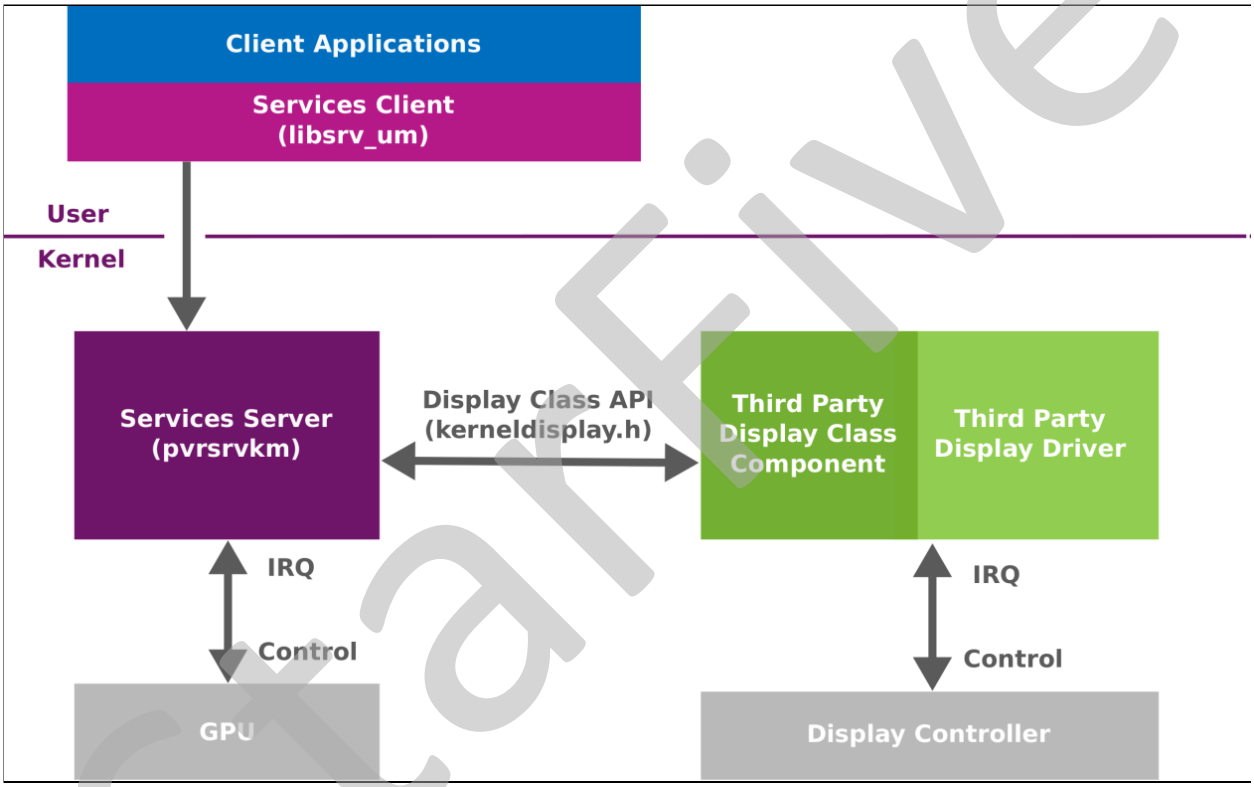
5. Display Integration

5.1. Display Class Architecture

Design considerations:

- Important for independent display hardware to be 'coordinated' with Services PowerVR Rogue class GPUs
- Third party Display Class API provides a consistent interface between Services and third-party display device drivers
- Abstracts control of display hardware via the Display Class API
- Provides maximum performance while maintaining the order of operations on shared resources.

Figure 5-1 Display Class Architecture



1. **Client Application**: It has a Services Client component built into it which provides the Services API and the Display Class client API. This is often a window manager that directly or indirectly interfaces to other rendering applications.
2. **Services Server**: The 'kernel mode' Services component implementing the Display Class logic and initiates calls into the third-party Display Class Component.
3. **Third Party Display Device Driver**: This is a device driver for controlling the display hardware.

'Interfacing code' is added to the driver allowing the display device driver to be integrated with Services. There are two sub-components:

- **Display Driver** (light green): This contains device specific code necessary to drive the display. This source code is outside of the GPU.
- **Display Class Component** (dark green): This contains display device specific code to implement the **Display Class** callback functions invoked by the **Services Server** and the code calling exported functions from the **Services Server**.

5.2. Display Context

A display context is a container in which buffers are allocated or imported and operations such as reconfigurations are made. It's purely a SW concept and is there to ensure that operations issued on the same display context are issued in order.

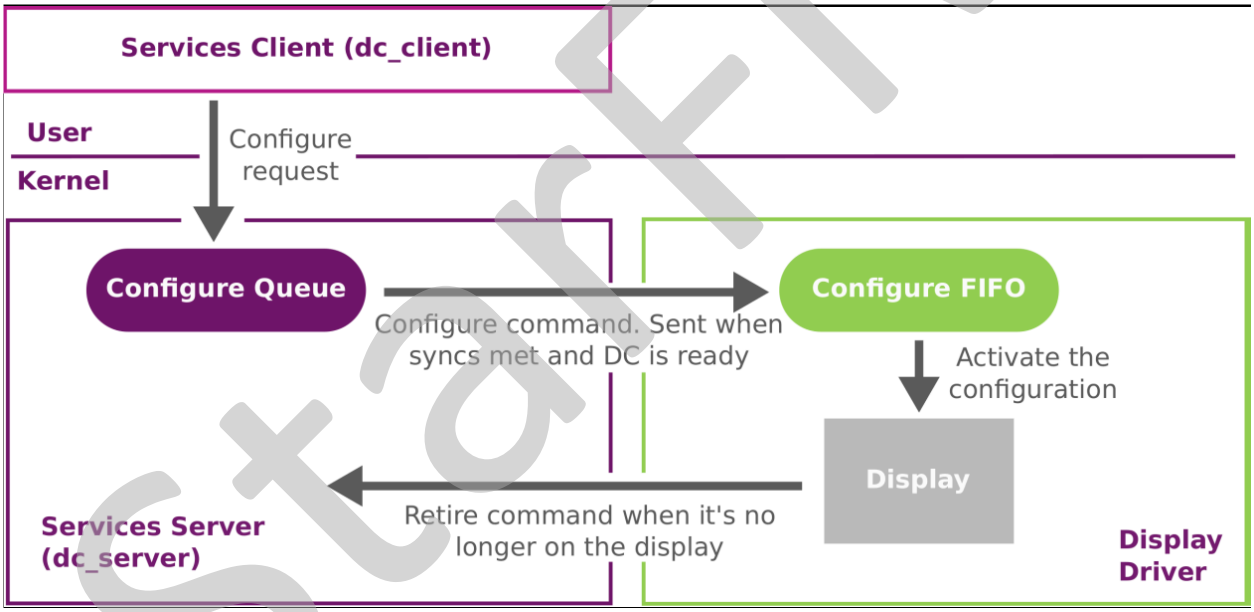
5.2.1. Display Configuration

Clients can configure the display to present one or more buffers with a certain configuration for each buffer. Due to the pipelined nature of the driver these operations will normally be linked to synchronization fences that allow the configuration request to be queued but not enacted until required (e.g. when the GPU has finished a render). The Services server handles all knowledge of synchronization and will only issue the configuration request when the operation is allowed to run, i.e. the synchronisation fence is met and the display is able to process a new command.

The third-party display driver must support multiple commands in flight; that is, configuration requests that have been sent to it but have not yet been retired. This is due to the fact that the display driver is not allowed to retire the active configuration until a new request has been issued and is on the display. When a configuration has been finished, it has been displayed for the specified period and is no longer being used by the display, it is retired. This is notified to Services server from the display driver to ensure that any required synchronisation signalling is done.

A number of these configuration requests can be issued by the client without having to wait until they have been retired ensuring that Services can keep all hardware devices running by having operations queued, minimizing the system idle time.

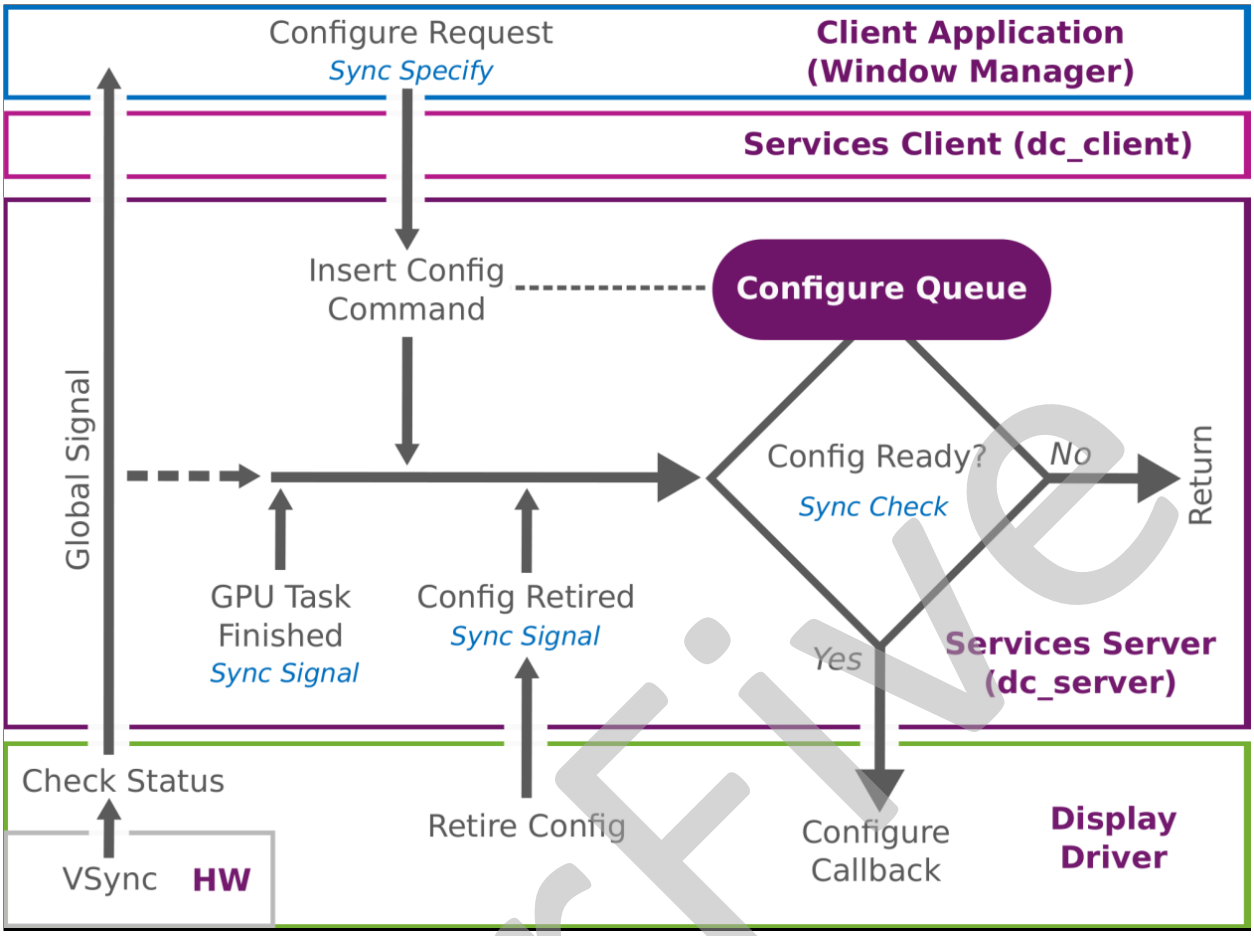
Figure 5-2 Configuration Flow



5.3. Display and GPU Synchronization

The client application is expected to manage the synchronization of the GPU and the display controller and must specify the correct synchronisation fence in the configure request. Usually a synchronization update fence from the last submission of rendering GPU work to the display buffer is supplied, merged in with any other dependencies. When this fence is met, the display buffer is ready to be shown.

Figure 5-3 Event Flow



The above diagram shows how synchronisation is embedded in the configuration flow. The blue Sync comments above show when fences may be given, checked or signalled. The general flow has three major stages:

1. When the client application is requesting a configuration change to display a buffer, it usually specifies an acquire fence that will be signalled as soon as the buffer is ready to be displayed. That means usually the client application or the GPU has finished writing to it.
2. There are three events which trigger a check of the synchronization fence of the next command in the configuration queue. These events could possibly have unblocked the next configuration command:
    a. A GPU task has finished and signalled a update fence.
    b. Insertion of a new command in the configure queue.
    c. A configuration has been retired, i.e. a buffer is again ready to be used.
3. If the synchronization fence with the next command in the queue is signalled and the command is ready to be submitted Services will call into the display driver via a registered configuration callback to display the buffer. As soon as the buffer is not on display anymore the display driver must retire the configuration with a call into services. Services then will signal the synchronisation release fence associated with that configuration, and this may unblock another item in the configuration queue or a GPU task.
4. The drawn VSync event is special in that it will not change any synchronisation fences but will signal a waiting client application to wake up. The client application itself can choose if it wants to be notified on a VSync event by calling **PVRSRVDCSetVSyncReporting** from the Services client library. This will ask the display driver to call the exported server function **PVRSRVCheckStatus** every time a VSync event occurred. **PVRSRVCheckStatus** again will signal a Services server event object that the client application can wait for by calling **PVRSRVEventObjectWait**.

5.4. Integrating Display Classes

This section discusses which steps it takes to port the Display Class to a new system. It will refer to the Display Class example implementation and the Services unit test **rgx_blit_test** found under:

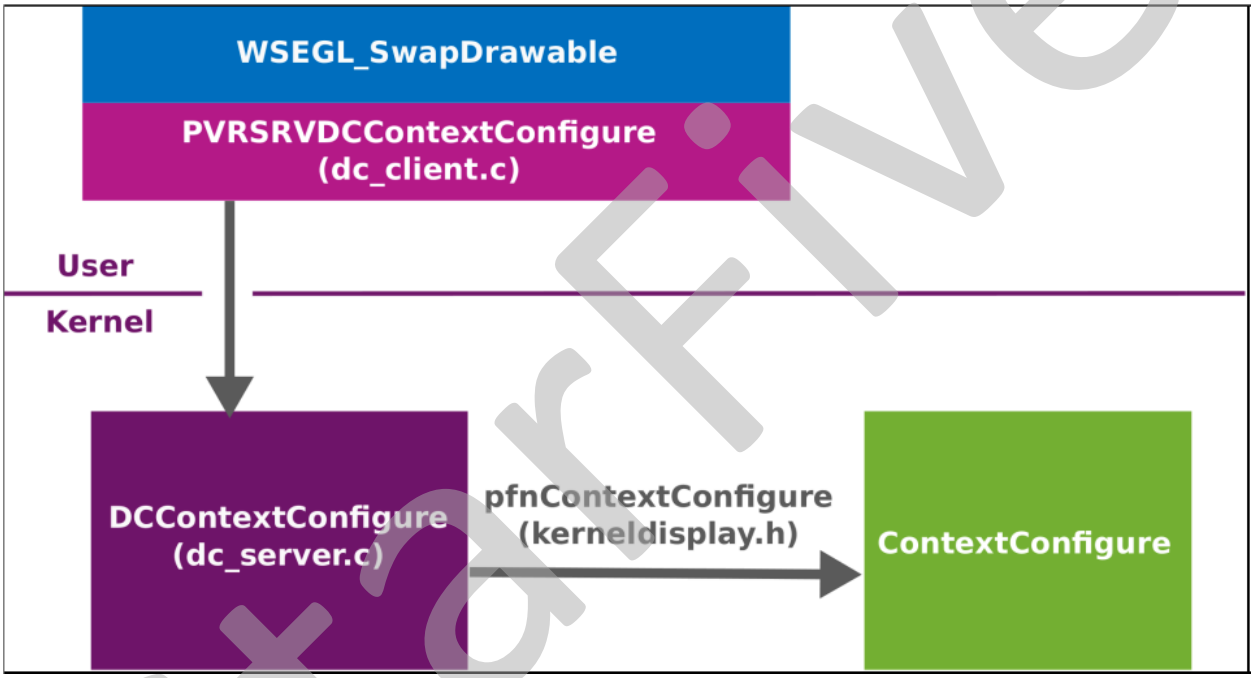
```
GPU_PATH/services/display/dc_example/...
```

```
GPU _PATH/unittests/services/rogue/external/rgx_blit_test/...
```

The **dc_example** implementation shows how a Display Class component implementation could look like and the test is exemplifying how to integrate the Services Display Class client function calls in the rendering workflow of a client application. Usually this client application is a window manager but, in this example, (**rgx_blit_test**) the rendering application directly makes calls to the Display Class client API.

The dc_example module was designed to work without actual display hardware and just implements the basics to make it work with the Display Class workflow.

Figure 5-4 Call Graph For Re-Configuration



The above diagram shows a typical function call stack for a reconfiguration call that originates from the window manager; in this case a Services WSEGL implementation. It is based on the diagram in Service Architecture *(on page 12)* and is the path most calls from the window manager (client application) to the Display Class client interface take.

5.4.1. Collecting Porting Requirements

This section provides some suggestions on how to best start the porting.

It makes sense to first implement the callbacks for the Third-party Display Class Component and then integrating the Services client-side functionality with the window manager.

First, one should assemble features and restrictions of the display controller hardware and the display driver. Answering the following questions is usually a good start:

- Does the controller have an MMU or can it only access physically contiguous memory?
- Are you able to map/unmap buffers at any time? Do you want to keep them mapped?
- Can the controller access only memory from a special region?
- Do you support multiple buffers and therefore allow display flips?

- Does the driver have its own mechanism to allocate memory?
- Can the display configuration change and does Services need to know about it?
- Does the controller emit VSync-events?
- Should the window manager be able to turn off the screen?

5.4.2. Implementing a Third Party Display Class Component

The answers to the questions in the previous section should give the implementer a good overview of the requirements and will help to choose which parts of the function table to implement. The used parameter structures can be looked up either in the integration reference or directly in the headers.

```
GPU_PATH/include/dc_external.h
GPU_PATH/include/rogue/pvrsrv_surface.h
```

5.5. Screen Parameter Adjustment

On the Linux platform, you can use `modetest` to adjust the screen's hue, saturation, contrast and brightness and other properties.

The DRM drivers of LIBDRM and kernel are required to support atomic property.

- LIBDRM:

  ```
  331017ae06a modetest: Add option to enable atomic capabilities
  ```

- Kernel:
  - For DSI:

    ```
    3fdcc6dc0779 drm/verisilicon: dsi: add support legacy api to set property
    ```

  - For DP:

    ```
    0de45ac60e93 drm/verisilicon: dp: add support legacy api to set property
    ```

  - For LVDS:

    ```
    ad0afccfcd79 drm/verisilicon: lvds: add support legacy api to set property
    ```

  - For RGB:

    ```
    1210fcf23a85 drm/verisilicon: rgb: add support legacy api to set property
    ```

Use `modetest` and `modetest -w option: -w <obj_id>:<prop_name>:` to set related properties.

The following code block provides an example of setting the color hue of an HDMI screen:

```
modetest -M
Connectors:
id    encoder   status     name      size (mm)    modes   encoders
116     0       connected  HDMI-A-1  0x0             1       115
  modes:
        index name refresh (Hz) hdisp hss hse htot vdisp vss vse vtot
  #0 640x480 59.94 640 656 752 800 480 490 492 525 25175 flags: nhsync, nvsync; type: driver
  props:
...
```

You will get the response that the HDMI screen connector ID is 116.

The following code block provides an example of setting the hue value of the HDMI screen to 60 from the default hue value of 50, and the adjustable range is 0-100.

```
modetest -M StarFive -a -w 116:hue:60
```

In the same approach, you can set the hue, saturation, contrast, and brightness of other screens just replacing the comments.

6. Buildroot Weston

6.1. Introduction

Weston is the official implementation reference of Wayland open source display protocol, and Weston 9.0(or 8.0).

There are multiple ways to configure StarFive Buildroot SDK Weston:

- Command line options: That is, the options of the command when starting Weston.

  The following code block provides an example.

  ```
  weston --tty=2
  ```

- Configuration file: The `weston.ini` configuration file.

  The configuration is located in `/etc/xdg/weston/weston.ini`, corresponding to the location of SDK code: `buildroot/package/weston/weston.ini`.

  Please refer to: https://fossies.org/linux/weston/man/weston.ini.man.

- Special environment variables

  Generally, these environment variables are set in these places:

  - The start-up script of Weston, in `/etc/init.d/rc.pvr` of the SDK firmware.

    The following code block provides an example.

    ```
    # /etc/init.d/rc.pvr start
    ...
    export WESTON_DRM_MIRROR=1 # should be set before starting
    Weston
    ...
    weston&
    ```

  - The environment script of Weston, in `/root/run_weston.sh` of the SDK firmware, corresponding to the location of SDK code: `buildroot/package/weston/run_weston.sh`.

- Dynamic configuration file:

  For DRM back-end, Buildroot SDK Weston provides some dynamic configuration support, such as dynamic display configuration files, the default path is `/tmp/.weston_drm.conf`. The dynamic configuration can be specified by the environment variable **WESTON_DRM_CONFIG**.

- udev rules: Some configuration of input devices in Weston should be set by udev rules.

6.2. Configuration

6.2.1. Status Bar

Weston supports setting the background color and position of status bar in the shell section of `weston.ini` configuration file, and setting the quick start program in the launcher section, for example:

```
# /etc/xdg/weston/weston.ini

[shell]
panel-color=0x90ff0000
# the color format is ARGB8888

panel-position=bottom
# top|bottom|left|right|none,none is disable
```

```
[launcher]
icon=/usr/share/icons/gnome/24x24/apps/utilities-terminal.png
# icon path

path=/usr/bin/gnome-terminal
# quick start command
```

Currently, Weston does not support setting the size of status bar. You have to modify in the code level when need some adjustments:

```
// weston/clients/desktop-shell.c

static void panel_configure(void *data,
struct weston_desktop_shell *desktop_shell, uint32_t edges, struct window *window, int32_t width, int32_t
 height)
{
...
switch (desktop->panel_position) {
case  WESTON_DESKTOP_SHELL_PANEL_POSITION_TOP: case  WESTON_DESKTOP_SHELL_PANEL_POSITION_BOTTOM:
height = 32; # height break;
case  WESTON_DESKTOP_SHELL_PANEL_POSITION_LEFT: case  WESTON_DESKTOP_SHELL_PANEL_POSITION_RIGHT:
switch (desktop->clock_format) { case CLOCK_FORMAT_NONE:
width = 32; break;
case CLOCK_FORMAT_MINUTES:
width = 150; break;
case CLOCK_FORMAT_SECONDS:
width = 170; break;
}
break;
}
```

6.2.2. Background

Weston supports setting the background pattern and color in the shell section of the `weston.ini` configuration file.

The following code block provides an example.

```
# /etc/xdg/weston/weston.ini

[shell]
background-image=/usr/share/backgrounds/gnome/Aqua.jpg
# Background pattern (wallpaper) absolute path

background-type=tile
# scale|scale-crop|tile

background-color=0xff002244
# The color format is ARGB8888, effective when no background pattern is set
```

6.2.3. Idle Time and Lock Screen

Idle Time

The idle timeout of Weston can be configured in the command options or in the core section of `weston.ini`.

The following code blocks provide 2 examples.

- ```
  # /root/run_weston.sh
  ...
  weston --idle-time=0& # 0 means idle mode is disabled, in
  seconds
  ```
- ```
  # /etc/xdg/weston/weston.ini

  [core]
  idle-time=10
  ```

Lock Screen

Lock screen of Weston can be configured in the shell section of `weston.ini`.

The following code block provides an example.

```
# /etc/xdg/weston/weston.ini
[shell]
locking=false
# lock screen is disabled
lockscreen-icon=/usr/share/icons/gnome/256x256/actions/lock.png
# unlock button icon
lockscreen=/usr/share/backgrounds/gnome/Garden.jpg
# background of lock screen
```

6.2.4. Color Format

The default display format of Weston in the Buildroot SDK is ARGB8888. For some low-performance platforms, you can configure RGB565 in the core section in the `weston.ini`.

The following code block provides an example.

```
# /etc/xdg/weston/weston.ini
[core]
gbm-format=rgb565
# xrgb8888|rgb565|xrgb2101010
```

You can also configure the display format of each screen individually in the output section of `weston.ini`.

The following code block provides an example.

```
# /etc/xdg/weston/weston.ini
[output]
name=LVDS-1
# output name can be seen in the Weston startup log, such as: Output LVDS-1,
(connector 71, crtc 60)
gbm-format=rgb565
# xrgb8888|rgb565|xrgb2101010
```

6.2.5. Display Orientation

You can configure the display orientation of screens in the output section of `weston.ini`.

The following code block provides an example.

```
# /etc/xdg/weston/weston.ini
[output]
name=LVDS-1
transform=90
# normal|90|180|270|flipped|flipped-90|flipped-180|flipped-270
```

If you want to configure the orientation dynamically, you can use the dynamic configuration file.

The following code block provides an example of setting the rotation of 90 degrees on all screens.

```
echo "output:all:rotate90" > /tmp/.weston_drm.conf
```

The following code block provides an example of setting the rotation of 180 degrees on the eDP screen 1.

```
echo "output:eDP-1::rotate180" > /tmp/.weston_drm.conf #
```

6.2.6. Resolution and Scale

You can configure the screen resolution and scale of Weston in the output section of `weston.ini`.

The following code block provides an example.

```
# /etc/xdg/weston/weston.ini
[output]
name=LVDS-1
mode=1280x800
# should be an effective resolution supported by the screen
scale=2
# value must be an integer, support application-level scaling
```

If you want to scale to a specific resolution (without changing the physical resolution), you can configure through **WESTON_DRM_VIRTUAL_SIZE** environment variable.

The following code block provides an example.

```
# /root/run_weston.sh
export WESTON_DRM_VIRTUAL_SIZE=1024x768
```

If you want to configure the resolution and scaling dynamically, you can use the dynamic configuration file.

The following code block provides an example of changing the resolution of HDMI-A-1 to 800 × 600.

```
echo "output:HDMI-A-1:mode=800x600" > /tmp/.weston_drm.conf
```

The following code block provides an example of setting eDP-1 display to the position of (10, 20), and scaling the display to 400 × 600.

```
echo "output:eDP-1:rect=<10,20,410,620>" > /tmp/.weston_drm.conf
```

When the VOP hardware doesn't support scaling, these kinds of scale requires RGA 2D acceleration from StarFive.

6.2.7. Freeze Screen

When Weston is started, there will be a black screen that switches between boot logo and UI display temporarily.

If you want to prevent this black screen, you can freeze the Weston screen content temporarily.

The following code block provides an example of adding warm-up to Weston's command options.

```
# /root/run_weston.sh
start)
...
weston --warm-up&
```

The following code block provides an example of freezing 1 specific display and unfreezing after 1 second.

```
# /root/run_weston.sh
start)
...
export WESTON_FREEZE_DISPLAY=/tmp/.weston_freeze
touch /tmp/.weston_freeze
weston&
...
sleep 1 && rm /tmp/.weston_freeze& # unfreeze after 1 second
```

The following code block provides an example of freezing all the displays and unfreezing after 1 second:

```
# /root/run_weston.sh
start)
...
echo "output:all:freeze" > /tmp/.weston_drm.conf # Freeze the display
weston&
...
sleep 1 && \
echo "output:all:unfreeze" > /tmp/.weston_drm.conf& # unfreeze after 1 second
```

6.2.8. Screen Status

The DRM framework supports setting force status for screens.

The following code block provides an example.

```
echo on > /sys/class/drm/card0-HDMI-A-1/status # Force HDMI-A-1 connected
#on|off|detect, detect means hot-plug
```

For more specific screen status configuration, You can configure settings in the dynamic configuration file.

The following list provides some examples for different scenarios.

- Turn off the DSI display.

  ```
  echo "output:DSI-1:off" >> /tmp/.weston_drm.conf
  ```

- Turn on the eDP display.

  ```
  echo "output:eDP-1:on" >> /tmp/.weston_drm.conf
  ```

- Turn off the display.

  ```
  echo "compositor:state:off" >> /tmp/.weston_drm.conf
  ```

- Turn off display and enable touch-and-wakeup.

  ```
  echo "compositor:state:sleep" >> /tmp/.weston_drm.conf
  ```

- Turn on the display.

  ```
  echo "compositor:state:on" >> /tmp/.weston_drm.conf
  ```

6.2.9. Multiple Screens

The Buildroot SDK Weston supports multi-screen with the same or different display and hot-plug functions.

You can differentiate screens based on the name of DRM (obtained through Weston start-up log or `/sys/class/drm/card0-<name>`).

You can configure the settings in environment variables.

The following list provides examples for different scenarios.

- Specify HDMI-A-1 as the main display:

  ```
  export WESTON_DRM_PRIMARY=HDMI-A-1
  ```

- In mirror mode (multi-screen with the same display), if you skip this environment variable, the system will show different displays:

  ```
  export WESTON_DRM_MIRROR=1
  ```

- In mirror mode, scaling maintains the aspect ratio. If you skip this environment variable, the system will show forced full-screen:

  ```
  export WESTON_DRM_KEEP_RATIO=1
  ```

- Turn off the built-in monitor automatically when an external monitor is connected:

  ```
  export WESTON_DRM_PREFER_EXTERNAL=1
  ```

- When an external monitor is connected, keep the first external monitor as the main display by default:

  ```
  export WESTON_DRM_PREFER_EXTERNAL_DUAL=1
  ```

When the VOP hardware doesn't support scaling, it would try to use StarFive RGA 2D acceleration.

It also supports disabling the specified screen individually in the output section of `weston.ini`.

The following code block provides an example.

```
# /etc/xdg/weston/weston.ini
[output]
```

```
name=LVDS-1

mode=off
# off|current|preferred|<WIDTHxHEIGHT@RATE>
```

6.2.10. Input Device

No Screen

The Weston service requires at least one input device by default. If there is no input device, you need to configure the special settings in the core section of `weston.ini`.

The following code block provides an example.

```
# /etc/xdg/weston/weston.ini
[core]
require-input=false
```

Multiple Screens

If there are multiple screens in Weston, you should bound the input devices to screens.

- You can configure it in the `udev` rules in the **WL_OUTPUT** environment of the input device.

  The following code block provides an example.

  ```
  # /lib/udev/rules.d/99-goodix-ts.rules
  ATTRS{idVendor}=="dead", ATTRS{idProduct}=="beef", ENV{WL_OUTPUT}="HDMI-A-1"
  ```

- Or you can configure it in the `udev` rules in **WL_SEAT** environment in `udev` rules.

  The following code blocks provide an example.

  ```
  # /lib/udev/rules.d/99-goodix-ts.rules
  ATTRS{idVendor}=="dead", ATTRS{idProduct}=="beef", ENV{WL_SEAT}="seat1"
  ```

  ```
  # /etc/xdg/weston/weston.ini
  [output]
  name=LVDS-1
  seat=seat1
  ```

6.2.11. Input Device Calibration

Touch Screen

If you need to calibrate the touch screen, you can use the **WESTON_TOUCH_CALIBRATION** environment variable.

The following code block provides an example.

```
# /root/run_weston.sh
export WESTON_TOUCH_CALIBRATION="1.013788 0.0 -0.061495 0.0 1.332709 -0.276154"
```

Calibration Tool

You can also use the Weston calibration tool **weston-calibrator** to obtain the calibration parameters. After running this tool, a number of random points will be generated, and then click them in sequence to output the calibration parameters.

The following code block provides an example.

```
Final calibration values: 1.013788 0.0 -0.061495 0.0 1.332709 -0.276154
```

Or you can use the new Weston touch calibrator: **weston-touch-calibrator**.

The following code block provides an example.

```
# /etc/xdg/weston/weston.ini
[libinput]
touchscreen_calibrator=true
calibration_helper=/bin/weston-calibration-helper.sh
```