Application Note

Debugging GPU Driver Issues on TDA4x and AM6x Devices

Erick Narvaez, Krunal Bhargav *EPD Processors*

ABSTRACT

Texas Instruments' processors can contain a graphics processing unit (GPU) which accelerates the computations for graphics-related tasks. Occasionally there are bugs in the graphics processing stack that cause issues in the applications. Due to the complexity of the processing stack, which spans many software layers and runs on long hardware pipelines internally, these issues can be difficult to debug. Furthermore, different components in the layers can be created by different developers and bringing them all together can be a challenge. This guide presents the different issues that can be seen from a graphics application and the steps for a systematic debug of GPU-related problems.

Table of Contents

Trademarks

Linux® is a registered trademark of Linus Torvalds.
Imagination Technologies® is a registered trademark of Imagination Technologies Limited.
All trademarks are the property of their respective owners.

1 Introduction

This application note gives an overview of the debug process for GPU related issues. The ability to identify a graphics-related problem and knowledge of the process of reporting the issue and providing the necessary information are key to a successful debug.

The first step is identifying the problems with graphics applications and linking them to symptoms of common issues. A complete list cannot be created since side-effects of these bugs can span the whole system. Instead, a rough list is presented that can help identify the most common problems. The second step is understanding the information needed to identify and solve the issue. There are a variety of debug logs and failure details that are required to characterize the severity of the failure and pin-point the culprit. Finally, if the initial information and basic logging is not sufficient, some advanced debug logging techniques are required. The most important knowledge from this guide is the terminology used around graphics issues and the tools available specifically for the GPU.

2 Common Issues with Graphics Applications

This section describes the common issues faced when dealing with graphics-related problems. These issues usually manifest on the display or as interruptions to the flow of data in the application.

2.1 System or Application Freeze

Freezing is the state where the application has halted the processing of data. A freeze can be at the application or system level. At the application level the freezing is isolated to the process that was running the graphics workload. At the system level, there can be other symptoms in the OS such as a kernel panic, which causes other applications to fail as well. This distinction is important because freezing can indicate the severity of the failure as well as the location. A freeze is usually accompanied by a log dump of the process stack where the exception halted the execution or no indication of the failure at all. Section 2.1.1 is an example of a kernel panic that occurred and caused the system to freeze.

2.1.1 Typical Kernel Panic Logs

```
root@tda4vm-sk:~# [ 5894.898990] Unable to handle kernel NULL pointer dereferen0
[ 5894.907770] Mem abort info:
[ 5894.907940] Unable to handle kernel paging request at virtual address ffb8000
[ 5894.910552] ESR = 0x96000046
[ 5894.918446] Adjusting arch_sys_counter more than 11% (651145911 vs 93113548)
[ 5894.921484] EC = 0x25: DABT (current EL), IL = 32 bits
[ 5894.928527] Unable to handle kernel paging request at virtual address ffd2004
[ 5894.933800] SET = 0, FnV = 0
[ 5894.941691] Mem abort info:
...
[ 5894.984251] Hardware name: Texas Instruments J721E SK (DT)
[ 5894.984254] pstate: 80000085 (Nzcv daIf -PAN -UAO -TCO BTYPE=--)
[ 5894.984265] pc : _raw_write_lock_irqsave+0x168/0x318
[ 5894.984270] lr : try_to_wake_up+0x5c/0x4e0
[ 5894.984271] sp : ffff8000113afdd0
[ 5894.984273] x29: ffff8000113afdd0 x28: ffff8000100d8ad0
[ 5894.984277] x27: ffff00087fa88300 x26: 0000000000000006
...
[ 5894.984319] x1 : 0000000000000000 x0 : 0000000000010003
[ 5894.984323] Call trace:
...
[ 5894.984397] el1_sync_handler+0xac/0xc8
[ 5894.984399] el1_sync+0x88/0x10
[ 5894.984401] 08ff00800v10f826b8
[ 5894.984406] Code: 451806018d5334611 526b0a90 f8800871 f8850fc60)
[ 5894.984413] ---[ end trace 2f5eabcaa9b203ad ]---
[ 5894.984416] Kernel panic - not syncing: Oops: Fatal exception in interrupt
[ 5894.984419] SMP: stopping secondary CPUs
[ 5896.056422] SMP: failed to stop secondary CPUs 0-1
[ 5896.056429] Kernel Offset: disabled
[ 5896.056431] CPU features: 0x0040022,20006008
[ 5896.056433] Memory Limit: none
[ 5896.481800] ---[ end Kernel panic - not syncing: Oops: Fatal exception in in-
```

2.2 Screen Tearing

Screen tearing is notorious in graphics applications, most prevalent in video games. The GPU runs at a particular frame rate and the display can have a separate frame rate. The synchronization between the GPU and the display can have issues when screen tearing or screen flickering occurs. Describing what the tearing looks like is important. Horizontal lines indicate that the GPU was in the middle of updating the previous frame with the next frame data, but the update was prematurely consumed by the display. Observe the pixel data around the tearing region and how far down the screen the tear is because this can indicate how severely the synchronization is being missed. Figure 2-1 is an example of screen tearing due to missing synchronization between the display and the graphics application.

Notice slightly above the halfway mark that there is an inconsistency in the triangles from a previous frame. This is the classic example of a *screen tearing* issue.

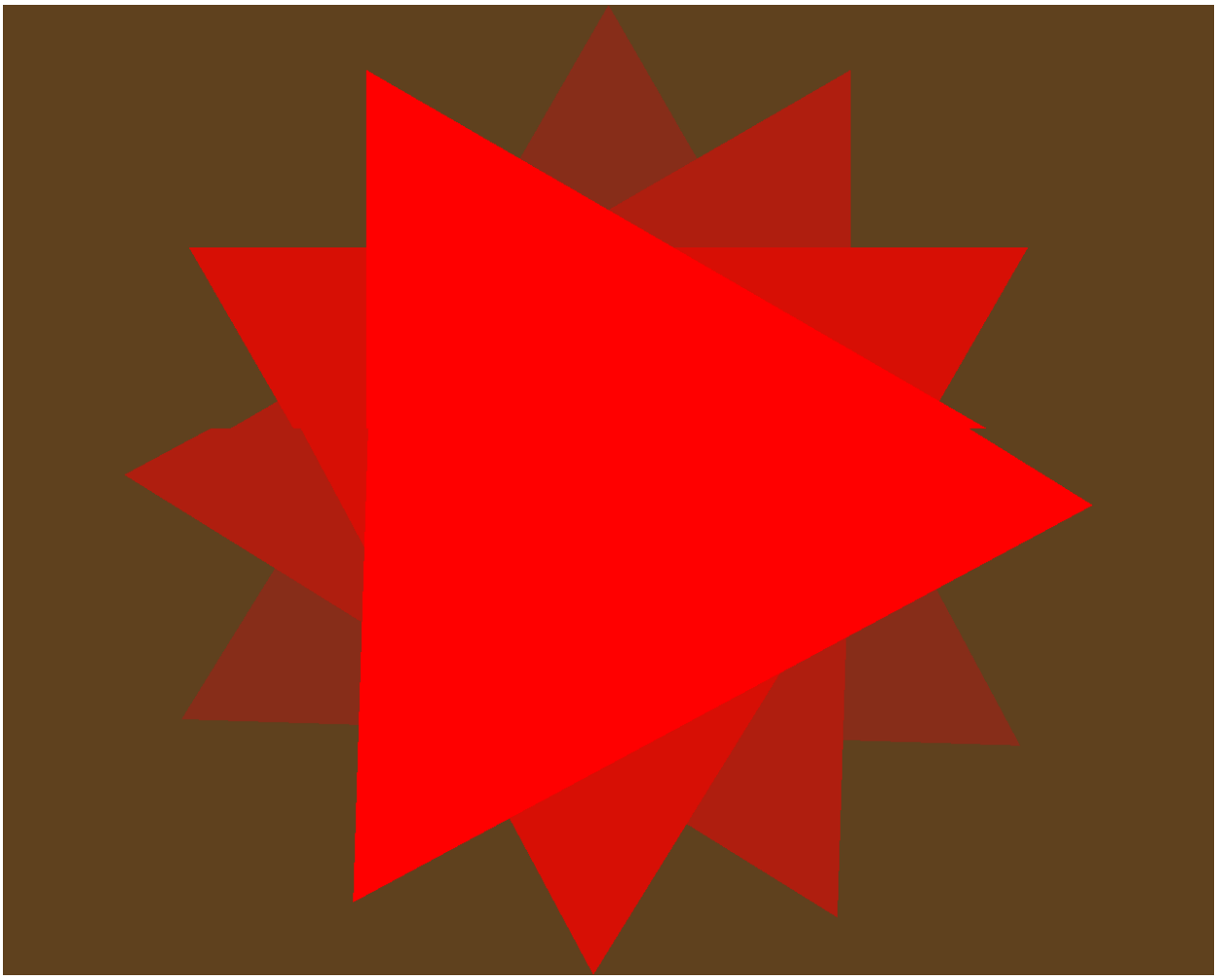

Figure 2-1. Screen Tearing Example

2.3 Artifacts or Corruption in the Screen

Artifacts or corruption in the screen issues are more difficult to track down because the issues can be caused by many more things. From bugs in the driver API to issues in the hardware, the range of causes is quite large. But identification of the underlying reason for the issues can be quickly determined through some of the tools introduced later in this document. Mainly, note if the corruption is isolated to a region on the output of the GPU, how long the corruption remains, and how much coverage the corruption has.

Figure 2-2 is fabricated to show different types of artifacting. Usually, the artifacts are random data, and spread out in a large chunk, repetitive small chunks or long chunks. Regardless of how the corruption appears, there is usually stale data or random data getting introduced into the framebuffer through corruption at any of the levels of the rendering pipeline.

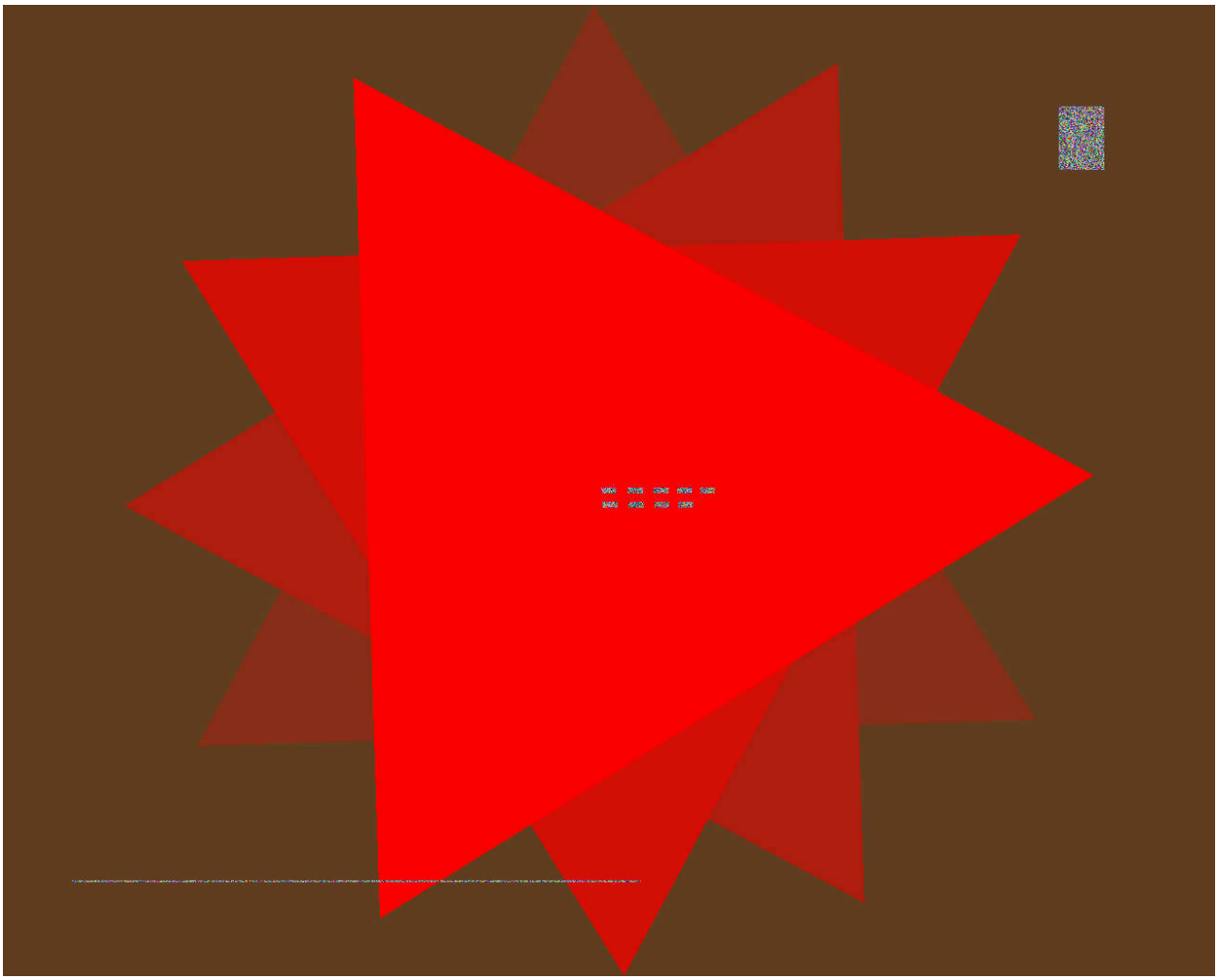

Figure 2-2. Artifacting Example

2.4 Blank Screen

If the graphics driver is not rendering due to issues in the program and halting of the code execution, it is possible there is no rendering resulting in a blank screen. Usually this issue is related to graphics APIs silently failing or something in the graphics software stack not being configured correctly.

2.5 Low Frame Rate

The application can still render sometimes if the system falls back to software rendering (software rasterization). The GPU driver can still be present but due to misconfiguration or a timing issue during the boot sequence, the windowing system can fail to recognized the GPU and fall back to rendering using the central processing unit (CPU). In these cases, simply checking the loading on the GPU is the simplest way to make sure it is being used. Partial utilization is also possible so check that all parts of the application are rendering with the GPU.

2.6 GPU Driver Logs and Hardware Recoveries

The GPU is equipped with a framework to identify issues in the graphics processing and issue hardware resets or recoveries to alleviate the symptoms. There are counters in the driver keeping track of the number of recoveries that have occurred. The cause of the hardware recovery (HWR) needs to be analyzed because these errors indicate a failure has occurred during GPU processing. GPU HWR logs have a standard look, and the easiest way to identify them is to look for the heading *PVR* in the logs. The logs outline some generic information in the console such as the driver version and build options. Logs also include more specific information relating to the crash. This data is dumped by the firmware in a standardized format that can be shared with TI to be analyzed. The relevant data needed for identifying the issue is not always available in these logs so the assigned engineer can request collecting the logs again with more verbosity and with modified drivers to get the necessary information.

There are cases when the GPU does not issue a HWR, but other messages related to the GPU driver are displayed and the GPU processing is affected. Similar to the previously-mentioned case, these logs need to be shared with TI for analysis, and can indicate other types of failures in the GPU driver. Section 2.6.1 is an example of a typical HWR.

An easy way to search for these is to search for *PVR* in the `dmesg` logs. The *PVR Log Dump Collection* section describes how to collect these logs to share with TI.

2.6.1 Typical GPU HWR Logs

```
[  275.343261] PVR_K:  228: ------------[ PVR DBG: START (High) ]------------
[  275.350212] PVR_K:  228: OS kernel info: Linux 6.1.46-g5892b80d6b #1 SMP PREEMPT Wed Apr  3
19:34:28 UTC 2024 aarch64
[  275.360827] PVR_K:  228: DDK info: Rogue_DDK_Linux_WS rogueddk 23.2@6460340 (release)
j721s2_linux
[  275.369798] PVR_K:  228: Time now: 275369792us
[  275.374260] PVR_K:  228: Services State: OK
[  275.378483] PVR_K:  228: Server Errors: 0
[  275.382800] PVR_K:  228: Connections Device ID:0(128) P1128-V1-T1153-VayaDriveConsol
[  275.390613] PVR_K:  228: ------[ Driver Info ]------
[  275.395699] PVR_K:  228: Comparison of UM/KM components: MATCHING
[  275.405345] PVR_K:  228: KM Arch: 64 Bit
[  275.410821] PVR_K:  228: Driver Mode: Native
[  275.415527] PVR_K:  228: UM Connected Clients: 64 Bit
[  275.420814] PVR_K:  228: UM info: 23.2 @  6460340 (release) build options: 0x80000810
[  275.428899] PVR_K:  228: KM info: 23.2 @  6460340 (release) build options: 0x00000810
[  275.437396] PVR_K:  228: Window system: wayland
[  275.442353] PVR_K:  228: ------[ Server Thread Summary ]------
...
[  275.481396] PVR_K:  228: ------[ RGX Device ID:0 Start ]------
[  275.487504] PVR_K:  228: ------[ RGX Info ]------
...
[  275.789271] PVR_K:  228: ------[ RGX registers ]------
...
[  276.648689] PVR_K:  228: ---- [ RISC-V internal state ] ----
...
[  276.759651] PVR_K:  228: ------[ RGX FW Trace Info ]------
...
[  276.777139] PVR_K:  228: ------[ Full CCB Status ]------
...
[  276.868040] PVR_K:  228: ------[ AppHint Settings ]------
...
[  276.940907] PVR_K:  228: ------[ Active Sync Checkpoints ]------
...
[  277.035347] PVR_K:  228: ------------[ PVR DBG: END ]------------
[  277.043511] ------------[ cut here ]------------
...
[  277.303180] ---[ end trace 0000000000000000 ]---
```

3 Support Flow for Graphics Issues

To quickly solve a graphics issue, use the following steps in a graphics debug scenario. Fast response times and detailed observations on the issue can expedite the process.

Note

The ability to reproduce the issue quickly (less than an hour) and on all devices (50% or more devices) is the limiting factor of solving the issue. Otherwise collecting logs to solve the issue is difficult.

3.1 Submit Preliminary Description

Now that a graphics-related issue has been identified, the following information needs to be collected for the next steps of the debug. The more complete this information is presented, the smoother the debug experience.

1. Symptoms: What are the symptoms seen at the system level? At the console output? Section 2 detailed many of the common symptoms, and some can happen at the same time.
2. Set up: How is the GPU used? Is the display a native Linux® display, or is this a remote display? Is this a vision processing pipeline in a RTOS plus Linux setup? What is the GPU driver and Software Development Kit (SDK) version used?
3. Scenario: In what situation did this issue occur? Where there other processors running? Is the processing part of a larger vision processing pipeline or similar? Does the issue occur immediately after starting the application or after some time?
4. Reproducibility: How easily can the issue be reproduced? Does the issue only happen on one board or does the issue happen on all boards? How long does the issue take to reproduce?
5. Recovery: Does the issue recover without any intervention, or does the system stay in a crashed state?
6. Logging: Please share all of the logs in the console and in the kernel (`dmesg`), or note that there are no logs when the failure happens.

This preliminary information is required throughout the debug and goes in tandem with the logs to identify the issue.

3.2 Determine if the Issue Reproduces on TI EVM

In the best situation, the issue reproduces on a TI EVM on the latest SDK version available. This way, TI engineers can replicate the issue locally and make progress without requiring back-and-forth with the customer. Unfortunately, users have custom hardware and custom software which can be proprietary and impossible to share. Furthermore, the issue sometimes only replicates in a production system where the system is under a high load and requires a much more complex setup than is feasible to recreate. These complications cause a remote replication of the issue to be cumbersome.

But if there is low overhead to port the application to the TI EVM and reproduce the issue there, this greatly speeds up the investigation and is worthwhile to explore depending on the level of urgency. Otherwise, the investigation continues on the custom hardware.

3.3 Provide Follow-Up Testing and Logs

Most issues are resolved by upgrading to the latest driver. If an issue is not solved, the user can work with the TI engineer to address the issue by providing logs and applying delivered patches to the GPU driver. Understanding how to update the GPU drivers in the system to expedite the debug process is important knowledge. Section 5 has a detailed explanation on building and installing graphics drivers on a Linux system. Furthermore, Section 4 also serves as a reference for the different tools that can be used in the debug process.

4 Tools for GPU Driver Debug

After initial analysis of the issue description and logs, it is sometimes necessary to introduce advanced debug tooling and techniques to condense the investigation. There are driver tools available with the graphics driver that can provide firmware and API traces. Verbose logging and modifications to the graphics driver is sometimes necessary. The following sections describe these procedures to provide clarity and reference.

4.1 Driver Status in Linux® DebugFS

The GPU driver adds an entry in the Linux kernel `debugfs` (`/sys/kernel/debug/pvr`) that exposes relevant statistics. These can be used to identify the GPU driver version, loading, and configurations. The first to explore is the *status* entry:

```
root@j721s2-evm:/sys/kernel/debug/pvr# cat status
Driver Status: OK

Device ID: 0:128
Firmware Status: OK
Server Errors: 0
HWR Event Count: 0
CRR Event Count: 0
SLR Event Count: 0
WGP Error Count: 0
TRP Error Count: 0
FWF Event Count: 0
APM Event Count: 15
GPU Utilisation: 0%

VM0
2D Utilisation: 0%
GEOM Utilisation: 0%
3D Utilisation: 0%
CDM Utilisation: 0%
RAY Utilisation: 0%
GEOM2 Utilisation: 0%
```

The important sections are the following:

- Firmware Status: Current status of the firmware running internally in the GPU. Make sure that the firmware is OK, otherwise the GPU status has deteriorated and the processing is compromised.
- HWR Event Count: Number of HWRs that have occurred. In normal circumstances, this number stays at 0. If this number is greater than 0, please submit a ticket to analyze the criticality of the issue.
- GPU Utilization: This percentage is a rough generalization of the GPU loading. More precise measurements can be done with PVRTune (an Imagination Technologies® tool). Look for PVRTune instructions in a future TI application note.

4.2 Driver AppHints

The `debugfs` also contains information on the *AppHints* used by the GPU driver. AppHints are runtime configurations for modifying the behavior of the driver. However, the details of these AppHints is not exposed as the details are not useful for the regular operation of the driver. Instead, these hints help to tweak some of the characteristics of the driver and are very useful in situations where advanced debugging is necessary. These AppHints can be set for all graphics programs (default section) or specific ones through the `/etc/powervr.ini` file:

```
[default]
ParamBufferSize=2097152
[gles2test1]
ParamBufferSize=16777216
```

The syntax is the executable name in square brackets, followed by the AppHints to be enabled. The default section applies to all executables, while any other section can be named for a specific application. In this instance, the unit test case `gles2test1` runs with `ParamBufferSize` smaller than other graphics programs.

4.3 PVR Log Dump Collection

As mentioned in Section 2.6, the HWR logs contain logs for the GPU driver and firmware. To collect the logs from the GPU, a utility is available called `pvrlogdump`. `pvrlogdump` comes with the GPU driver installation. The utility saves the logs in the `/tmp` directory by default and output the names of the files so the files are easy to extract. Alongside the files generated by `pvrlogdump`, it is also useful to share the console of the execution of the app as this shows when the HWR happened. The following is an example of the `pvrlogdump` command output:

```
root@j721s2-evm:~# pvrlogdump
Checking driver state ............... initialised
Checking for debugfs ................ found
Checking for lockdep ................ not found
Checking for ftrace ................. not found
Checking for firmware log groups .... not found
There are no AppHints enabled in /etc/powervr.ini or in debugfs for any of the firmware log groups.
Unless 'pvrdebug' tool was used for that purpose there will be no information in firmware log.
Please consider enabling some of the firmware log groups before the problem occurs.
Dumping data .........................
[ 163.767302] PVR_K: 1005: User requested PVR debug info
[ 163.772711] PVR_K: 1005: ------------[ PVR DBG: START (High) ]------------
…
[ 164.363015] PVR_K: 1005: ------------[ PVR DBG: END ]------------
done
Archiving data ...................... done
File /tmp/pvrlogdump_2402262237.txt.gz was created.
```

4.4 Adding Log Groups to Firmware Traces

The GPU driver can enable more tracing in the firmware logs through the `pvrdebug` command. There are various log groups that can be enabled in the firmware, and the required groups are communicated during the debug. To enable more (or less) log groups, the command can be used as follows:

```
pvrdebug -loggroups main,mts,hwr
```

4.5 Disabling the Driver After Hardware Recovery

Because the logging of the firmware is done on a small buffer, the GPU continues to overwrite the firmware logs after the crash has occurred. One way to get around this limitation is to freeze the firmware as soon as a HWR has happened. The command below halts the firmware and logging to save the last executed commands:

```
echo "Y">/sys/kernel/debug/pvr/apphint/0/AssertOnHWRTrigger
```

4.6 Disable Autoloading of the GPU Driver

The GPU driver sometimes requires extra parameters when being launched. This requires the GPU driver to not auto-load so that the driver can be manually started. To blacklist the GPU driver, disable `modprobe` by adding the following line to `/etc/modprobe.d/blacklist.conf`:

```
blacklist pvrsrvkm
```

In SDK 8.6 and previously (GPU driver 1.15 and earlier), there is an initialization script (`/etc/init.d/rc.pvr`) that is installed by the GPU driver. This script can be disabled by moving the script out of the directory, such as moving the script to the home directory.

To manually start the GPU after being blacklisted, simply insert the GPU module using `insmod` or `modprobe`.

5 Integrating Patched GPU Drivers

TI provides patched graphics drivers for identified issues. To understand the nature of the graphics driver, this section gives an overview of the GPU drivers and steps to generally integrate patches.

The GPU driver is composed of two parts: user-mode libraries and driver (UM) and kernel-mode (KM) driver. The UM is closed-source, meaning TI does not release the source code publicly. The KM is open-source, meaning TI makes the source available in the SDK or on public repositories. To install each on the system, different steps need to be taken.

5.1 UM Libraries Installation

The UM libraries are a collection of pre-built files which TI packages on the public-facing `umlibs` public repository: https://git.ti.com/cgit/graphics/ti-img-rogue-umlibs/.

These libraries are prebuilt for each release, and the build system adds the libraries to the generated filesystem. Users can also update the filesystem by manually copying these libraries, which are already in the correct file structure. The following command is used as reference, assuming the filesystem is on an SD card that has the `rootfs` partition mounted at `/media/user/rootfs`:

```
cd ti-img-rogue-umlibs
sudo cp -r targetfs/j721e_linux/wayland/release/* /media/user/rootfs
```

The update is complete. The GPU driver hooks into the initialization scripts and is loaded as the system boots up.

During debugs, updated libraries with bug fixes are provided in the same manner. Install the updates by replacing the existing files in the filesystem.

5.2 KM Libraries Installation

The KM driver is provided in TI's SDK under the following directory: `ti-processor-sdk-linux-adas-j721s2-evm-09_02_00_05/board-support/extra-drivers/ti-img-rogue-driver-23.3.6512818`

The driver is also posted in TI's public repository: https://git.ti.com/cgit/graphics/ti-img-rogue-driver/.

The build instructions can be extracted from TI's SDK build infrastructure, but the instructions are added here for clarity. The environment variables, which can be set in the shell *or* passed on the make command, are as follows:

Table 5-1. Build Environment Variables for KM Driver

Environment Variable Name	Value	Description
ARCH	arm64	Architecture of target CPU
CROSS_COMPILE	aarch64-none-linux-gnu-	The cross compiler being used
KERNELDIR	<absolute path to linux kernel on the system>	Path to the Linux kernel source directory
RGX_BVNC	36.53.104.796 - TDA4VL, TDA4VH, TDA4AEN, AM62P 22.104.208.318 - TDA4VM 33.15.11.3 - AM62x	Version of GPU to use, particular version of hardware revision
BUILD	release or debug	Build profile to use
PVR_BUILD_DIR	<soc>_linux	Which build directory to use within the driver, each SOC has a directory
WINDOW_SYSTEM	lws-generic	Which window system to use, only lws-generic is supported after SDK 9.0.

With these variables set, the make command ought to work. Build within the `ti-processor-sdk-linux-adas-j721s2-evm-09_02_00_05/board-support/extra-drivers/ti-img-rogue-driver-23.3.6512818/build/linux/j721s2_linux` in the SDK or the `ti-img-rogue-driver/build/linux/j721s2_linux`, if manually cloned.

To install on the target filesystem, the SDK runs the following make command from the create binary directory: `binary_j721s2_linux_wayland_release/target_aarch64/kbuild`

```
make -C ${LINUXKERNEL_INSTALL_DIR} INSTALL_MOD_PATH=${DESTDIR} INSTALL_MOD_STRIP=${INSTALL_MOD_STRIP} M=`pwd` modules_install
```

If installing from the build directory, one or more environment variable is required: `DISCIMAGE=<path-to-rootfs-root-directory>`

Example:

```
export DISCIMAGE=/media/user/rootfs
sudo -E env PATH=$PATH make install
```

Another way to install the KM library is to use the `install.sh` script in the output directory:

Example:

```
cd binary_j721s2_linux_lws-generic_release
sudo ./install.sh --root <path-to-rootfs-directory>
```

Note

If you rebuild the Linux kernel, the version can change and the GPU Kernel driver must be reinstalled.

Always check that you rebuild and reinstall the GPU kernel driver when rebuilding the Linux kernel if the Linux kernel version changes. Most notably, when installing the Linux kernel modules, a folder is created in the filesystem such as `/lib/modules/6.1.80-ti-g2e423244f8c0`. Modifying a kernel usually introduces the suffix `-dirty` which causes a new modules directory to be created. Thus, the GPU kernel driver must be reinstalled as well into the new kernel module directory.

Example: Manual Build of GPU Kernel Driver for AM62

```
make \
ARCH=arm64 CROSS_COMPILE=/home/toolchains/arm-gnu-toolchain-11.3.rel1-x86_64-aarch64-none-linux-gnu/bin/aarch64-none-linux-gnu- \
KERNELDIR=/home//am625_sdk/ti-processor-sdk-linux-am62xx-evm-09.01.00.08/board-support/ti-linux-kernel-6.1.46+gitAUTOINC+247b2535b2-g247b2535b2 \
BUILD=release PVR_BUILD_DIR=am62_linux
```

Example: Manual Installation of GPU Kernel Driver

```
cd binary_j721s2_linux_lws-generic_release
sudo install.sh -root <path to rootfs>
```

5.3 Post-Installation Steps

Now that both UM and KM libraries are updated, the system is ready for testing. When the Linux prompt appears, determine if the libraries were auto-loaded by running `dmesg | grep -i pvr` and view the output as follows:

```
root@j721s2-evm:~# dmesg | grep -i pvr
[ 5.085836] pvrsrvkm: loading out-of-tree module taints kernel.
[ 5.222004] PVR_K: 189: Read BVNC 36.53.104.796 from HW device registers
[ 5.415963] PVR_K: 189: RGX Device registered BVNC 36.53.104.796 with 1 core in the system
[ 5.672381] [drm] Initialized pvr 1.15.6133109 20170530 for 4e20000000.gpu on minor 0
[ 9.168474] PVR_K: 274: RGX Firmware image 'rgx.fw.36.53.104.796' loaded
[ 9.186869] PVR_K: 274: Shader binary image 'rgx.sh.36.53.104.796' loaded
```

Notice the *RGX Firmware image * loaded*. This means the GPU has loaded the firmware.

If you want to sanity check that the GPU is running, run this unit-test application that is part of the GPU driver:

```
root@j784s4-evm:~# rgx_compute_test
------------------ RGX compute test -----------------
```

```
----------------------- Start -----------------------
Call PVRSRVConnectionCreateDevice with a valid
argument:

Connecting to first (0) default pvr
device

 OK
Create dev var context:
 OK
Looking up General heap handle
 OK
Getting event object
 OK
Creating robustness buffer
 OK
Mapping robustness buffer
 OK
Creating Compute Context
 OK
Creating Buffer
Creating DWord for CDM Event Object
...
Destroy Compute Context
 OK

Total time: 0ms
Disconnect from services:
 OK
------------------------ End ------------------------
```

These results indicate that the GPU driver is running correctly. Further issues can result from applications trying to access a display, but that is a separate discussion. At this point, the user knows that the GPU driver is up and running.

6 Summary

Graphics issues can happen for many reasons, from the application software to the hardware design. Isolating the issue and finding the root-cause can be done efficiently when there is a good understanding of the reasons issues can happen and a structured approach is taken.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265