# Improvements To Kernel Testing For NVIDIA DRIVE OS Tegra Display

**Silicon Valley Project Center**

By:

Miles Gregg

Nathan Wong

Cather Zhang

Project Advisor: Mark Claypool

NVIDIA Sponsors: Allen Martin, Ishwarya Balaji Gururajan

NVIDIA Mentors: Adithya Sanjeev Byalpi, Matthew Trost

Submitted on 12/15/2023

## ABSTRACT

The NVIDIA Tegra chip is used in the automotive industry, with an emerging usage in autonomous vehicles. Tegra display testing faces challenges that include a lack of negative testing in the test infrastructure and failure to meet some safety requirements to comply with international and industry standards. We implemented more testing on the input/output control function calls to verify the display system behaves as expected with both valid and invalid inputs and fixed any system bugs that were revealed by the implemented test cases. We also refactored the codebase to satisfy different requirements for standard and safety builds. In total, we implemented 18 test cases, fixed 4 system bugs, integrated test cases to Tegra software's continuous integration and continuous development pipeline, and refactored 3 functionalities on safety system build. With the work the team has accomplished, the display system is more robust, the codebase is also cleaner and more concise.

## 1) INTRODUCTION

The NVIDIA Tegra chip is a system-on-chip (SoC) that is used in various industries such as mobile devices (phones, laptops), gaming devices (NVIDIA Shield Portable, NVIDIA Shield TV), embedded systems (drones, robotics), and automotive systems. There is emerging usage of the Tegra chip in autonomous vehicles (AVs). The NVIDIA Orin and Tegra products are a critical part within DRIVE OS teams to develop code for various parts of the operating system and within AVs. The NVIDIA Orin is the main computer that is used within NVIDIA's AVs as shown in Figure 1. These computers are operated by the Tegra X1 chip. The Orin board architecture layout includes various interfaces that connect to other physical hardware that is within a self-driving car. One prime example of this is the Orin Display Driver (ODD). The ODD is used by the DRIVE OS team for AV displays.

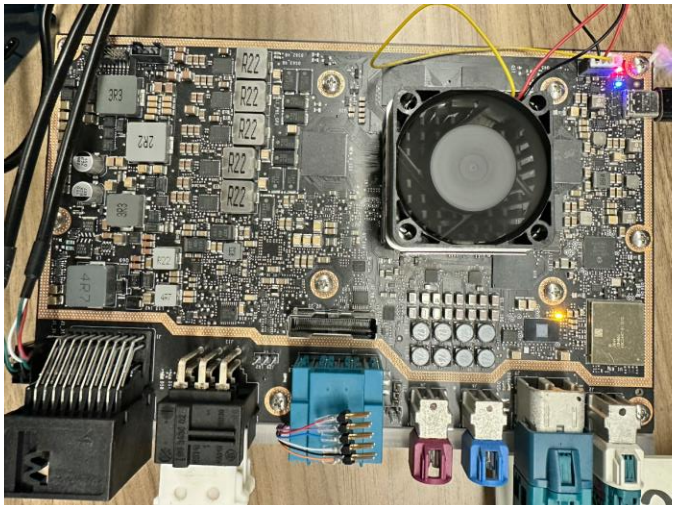


**Figure 1:** NVIDIA Orin Board

However, the public has a concern: if it is safe to use in an AV. To tackle this question and concern, ensuring safety is critical. Therefore, a robust and safe operating system remains a priority for NVIDIA DRIVE OS.

NVIDIA DRIVE OS is the operating system (combination of QNX and Linux) that is specifically designed around safety and security for AVs. Its respective operating system and software stack are solely intended for the use of developing and deploying AV software and applications onto DRIVE AGX-based hardware. The purpose of the design and architecture around the operating system is to incorporate a safe and secure executable environment for safety-critical applications on-board the AVs.

BlackBerry QNX for Safety is a full-featured, real-time OS for safety-critical embedded software used in safety sectors such as automotives [1]. QNX can handle high-priority tasks with low latency, which makes it well-suited for applications such as automotive systems. Another major benefit of QNX is the microkernel architecture, which minimizes downtime and cyberattack surfaces through isolation and separation mechanisms.

DRIVE OS must be thoroughly tested to comply with safety requirements, avoid regression issues, ensure system functionality, and fix defects within the codebase. Each team working on DRIVE OS is responsible for developing unit and integration tests for their group's work. Our team works exclusively on the display device driver for DRIVE OS. Testing the display driver involves a combination of positive and negative testing where positive testing validates driver functionality on valid inputs while negative testing validates driver failures on invalid inputs.

Currently, many of the tests on the display driver are positive tests. The team has already confirmed that many functions within the device driver produce the expected action and results. However, the current testing infrastructure lacks negative tests. The behavior of the driver involving image projection and memory allocation with invalid inputs remains unverified and undocumented. Additionally, some positive tests for the driver need to be updated to run more efficiently within the Continuous Integration and Continuous Development (CI/CD) pipeline.

Our team's approach to further develop the testing infrastructure required us to identify gaps within existing tests and blueprint integration-level tests that would result in the failure of the device driver. Each test endured a lengthy verification process where we confirmed that a test could run to completion, target the correct issue, and integrate properly within the CI/CD pipeline. Alongside this verification process, we documented concepts related to each test to increase overall understanding and enabled extension of the device driver for the display team.

Ultimately, our efforts resulted in the addition of 18 integration tests. Through each of our tests, we verified the driver's ability to handle invalid input and discovered various system bugs that needed correction. Correcting each defect resulted in a more robust device driver and increased the display team's confidence in the functionality of DRIVE OS.

The following chapters of this report detail the key concepts of our project and are organized as follows. Chapter 2 refers to the background where more technical terms related to the project were explained. Chapter 3 describes the methods used to approach the problem. Chapter 4 details our implementation towards our project objectives. Chapter 5 concludes with summary on our project outcomes. Chapter 6 highlights further work that can be done to extend our project.

## 2) BACKGROUND

Testing DRIVE OS requires intricate knowledge of the Tegra recovery system, hardware abstractions, system calls, and verification methods. Understanding each of these topics not only facilitates designing a test but also with comprehending their implications. Furthermore, these topics are repeatedly referenced throughout the implementation and verification processes of our project.

### 2.1 Tegra Recovery Architecture

The recovery system on Tegra platform consists of two main parts: Tegra SoC and microcontroller unit (MCU) AURIX as shown within Figure 2. Both parts are connected to a host computer through USB serial, allowing users to communicate from the host machine to the target board through minicom. Inside a Tegra SoC minicom terminal, users can access the QNX shell of the board, while AURIX minicom allows users to reset and recover the board. Because Tegra SoC and AURIX have separate Embedded MultiMediaCard (EMMC) storage, users can recover the board by flashing a new image without the risk of bricking the board. Even if an image is corrupted and results in the board being non-responsive in any way, the board can still be recovered by flashing a new, non-corrupted image.

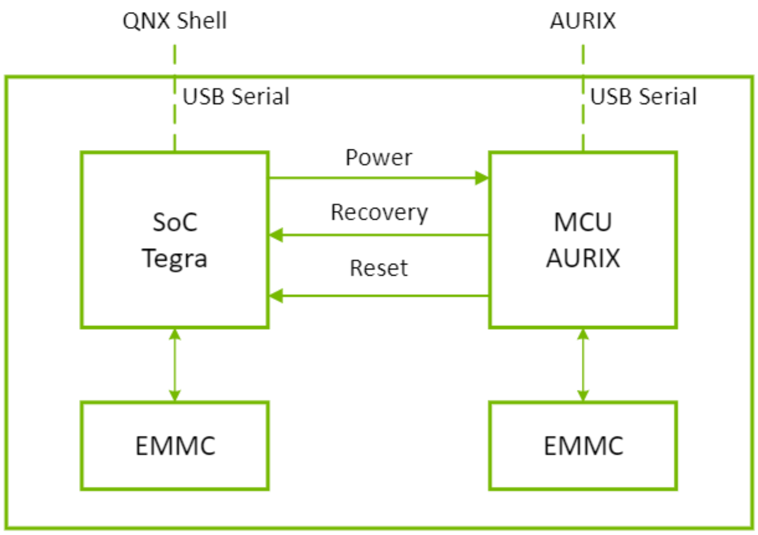


**Figure 2:** Tegra Recovery Architecture

## 2.2 Hardware Abstraction

The Pixel Processing Pipeline is the fundamental process workflow for how pixels get processed, transformed, and then displayed onto the display for users to view. Hardware on board is utilized to take a compositor blended screen-size pixel stream and drive it onto a single sink display for a user to see. The Pixel Processing Pipeline involves hardware and software to work together with one another to properly compute the necessary pixels to be piped onto the display. The various key components of the Pixel Processing Pipeline are described below:

**Channel**: is the control interface for the set of hardware resources on the DRIVE OS computer. A channel contains a push-buffer memory which allows the software to interact and write commands to hardware, and there is a channel state which is maintained by the hardware that the software can then read. NVKMS (NVIDIA Kernal Management System) currently interacts with window channels and core channels in which there is one for each one respectively.

- Window Channel: configures the alpha/opaque blending, color key, and other attributes of its specific window.
- Core Channel: configures all heads, specifies the background color of the compositor for the pipeline.

**Window:** is also referenced to as an overlay or layer which is a rectangular section of the screen.

**Head**: corresponds to the hardware on Orin that is used to take a compositor-blended screen-size pixel stream and drive it onto a single sink display. Orin board only has one set of output links and pads which are used to drive DP (Display Port) or HDMI (High-Definition Multimedia Interface) ports.

**Dev**: is a device which is either a single GPU or multiple GPUs linked by Scalable Link Interface (SLI). A multiple-GPU configuration does not pertain to Tegra because there is only one sub-device within ODD.

**Disp**: represents an individual programmable display engine of a GPU, which there is one on the Tegra chip.

**Connector**: represents the physical electrical connection to the GPU.

**Dpy**: represents the connection from the display device to the system hardware, which in our case because of the DP-MST this means that serval dpys can mapped onto the same connecter.

**Surface**: represents the memory to be sent out onto specific window.

Diagram below will be replaced by custom one that will be made

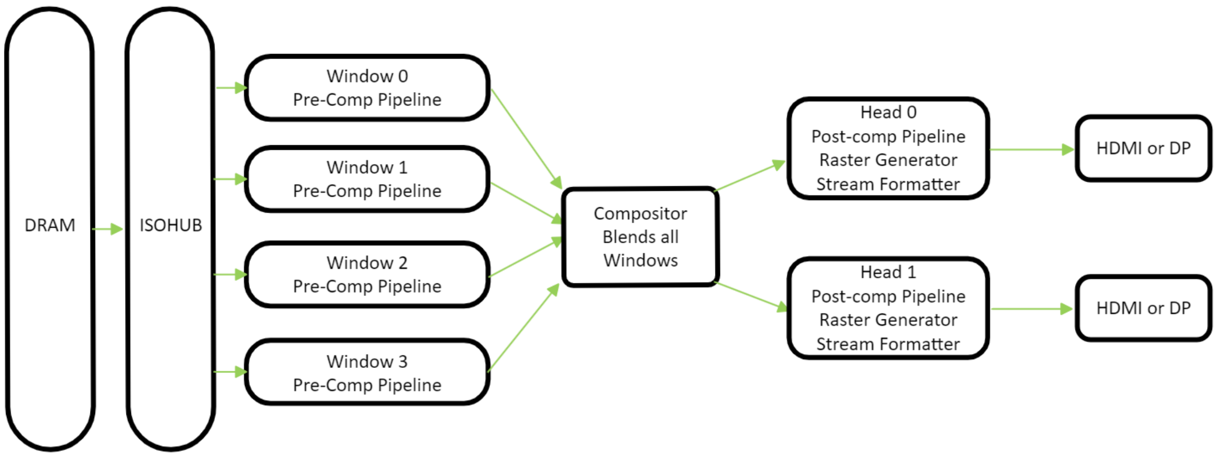

DRAM
ISOHUB
Window 0 Pre-Comp Pipeline
Window 1 Pre-Comp Pipeline
Window 2 Pre-Comp Pipeline
Window 3 Pre-Comp Pipeline
Compositor Blends all Windows
Head 0 Post-comp Pipeline Raster Generator Stream Formatter
Head 1 Post-comp Pipeline Raster Generator Stream Formatter
HDMI or DP
HDMI or DP

**Figure 3:** Pixel Processing Pipeline on Orin

## 2.3 Device Input and Output Control

An input/output control (IOCtl) call is a system call in the QNX operating system that allows the operating system to communicate with an external device. An IOCtl call can facilitate the implementation of a device driver by manipulating attributes in an operating file related to the device [2]. For example, the IOCtl calls for a display device can check and set the mode timings or produce images onto the screen. Within QNX, an IOCtl call accepts three arguments: a file descriptor for the device that will be manipulated, a request code, and arguments for the request. Each request code is a series of bytes that uniquely define the code, detail the action, define the arguments, and specify the direction of data transfer. With a request code, a device driver can identify the correct action and manipulate the correct configuration files on the device.

The testing infrastructure created for this project was built exclusively on the display driver. Furthermore, only a subset of all request codes used in the display driver were needed to complete code coverage within the driver. The request codes and actions of IOCtl calls used in this project are listed below:

- Register Surface IOCtl
- Unregister Surface IOCtl
- Flip IOCtl

**Register Surface IOCtl**: The register surface IOCtl is responsible for preparing a surface within memory. The register surface IOCtl accepts parameters specifying the device owner, width and height, memory layout, and pixel layout of the surface to create a surface that matches the arguments. Once the surface has been created and placed in memory, the register surface IOCtl will also set a surface handle which will allow each client to reference and utilize the surface.

**Unregister Surface IOCtl**: The unregister surface IOCtl is responsible for deallocating a registered surface and freeing its information from memory. The unregister surface IOCtl accepts parameters specifying the device owner and surface handle of the surface that will be unregistered. Through the surface handle, the IOCtl can identify the correct surface in memory that needs to be deallocated. Once a surface has been unregistered, clients can no longer reference and utilize the surface through the surface handle.

**Flip IOCtl**: The flip IOCtl is responsible for grabbing the information of a surface in memory, populating a layer with this information, and assigning these layers to a head. The flip IOCtl can then project the pixel information of the surface onto the display to create a visual. The flip IOCtl accepts parameters specifying which layers will be populated with what surface alongside the heads that these layers will be assigned to. Once the parameters have been populated, the assignment of heads to layers will be validated and compared to the configuration of the head to window mapping configuration file.

## 2.4 System Images

When building a system image, there are various flags that indicate the specific image.

1) Build type: **Standard** build is mainly used for development purposes. The default audience for standard builds is internal. **Safety** build is used for production purposes and thus has many restrictions on process abilities and permissions. The default audience for safety builds is external.

2) Log level: **Release** build only maintains error logs, while **debug** build maintains error, debug, and info logs.

3) Target board: since different boards have different device configurations, image build needs to specify the target board type. The boards that were used in the project were: p3663-a01, p3663-a01-f1, and p3710-10-a01.

4) Target OS: **Embedded QNX** is mainly used for production and **embedded Linux** is used for development. The project only focused on QNX image.

## 2.5 Head to Window Mapping Configuration File

The main purpose of the head to window mapping is to map specific heads on the board to windows which get transformed through the compositor portion of the pixel processing pipeline. The Drive Tree Config is saved within the kernel software which is in direct contact with the hardware side. The device tree source include(.dtsi) file stores the drive tree config where the nvidia,window-head-mask hex configuration is stored at. The head to window mapping table from bits to window mapping shows as follows in Table 1:

**Table 1:** Head to Window Mapping Table

| Head-Bitmask | Window-Number |
|---|---|
| BITMASK (0-7) | 0 |
| BITMASK (8-15) | 1 |
| BITMASK (16-23) | 2 |
| BITMASK (24-31) | 3 |
| BITMASK (32-39) | 4 |
| BITMASK (40-47) | 5 |
| BITMASK (48-55) | 6 |
| BITMASK (56-63) | 7 |

A head refers to how many displays are connected to the computer which in our case is for the ODD. The default configuration of ODD includes a max of 2 heads and 4 windows. There are 64 bits dedicated for the head to window mapping where there are 8 sets of 8 bits and each set of 8 bits is mapped to a given window (0-7). When inputting a head into a specific window you must start at the right most w_0 window. The bitmask calculation for the head to window mapping is as follows:

$$BITMASK(b_x - b_y) = \{w_n \mid w \text{ is a window and } n \text{ is window number}\}$$

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 |
| w_7 | w_6 | w_5 | w_4 | w_3 | w_2 | w_1 | w_0 |

All valid heads start at:

$$Head_{valid}(n) = \{\mathrm{n} \mid \mathrm{n} > 1\}$$

When the head is 0 this means that that specific window is not mapped to a head.

| Decimal | Binary | Hexadecimal |
|---|---|---|
| 000 | 00000000 | 0x00 |
| 001 | 00000001 | 0x01 |
| 002 | 00000010 | 0x02 |
| 003 | 00000011 | 0x03 |
| 004 | 00000100 | 0x04 |
| 005 | 00000101 | 0x05 |
| 006 | 00000110 | 0x06 |
| 007 | 00000111 | 0x07 |
| … | … | … |
| 255 | 11111111 | 0xFF |

On the Pixel Processing Pipeline for ORIN CPU the default and required number of windows is 4 and the required number of heads is 2. The mapping of the device tree configuration within the hardware abstraction is shown in Figure 4 below.

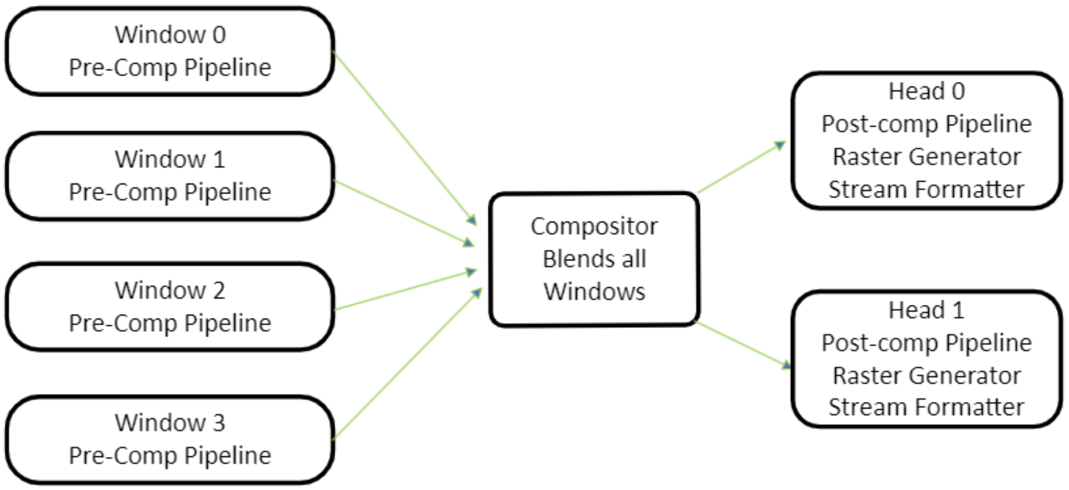
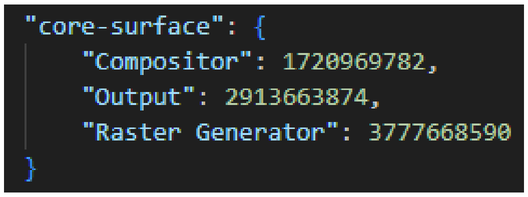


**Figure 4:** Window to Head Abstraction

## 2.6 Cyclic-Redundancy Check

A cyclic-redundancy check (CRC) is a data integrity verification method used to help detect errors in data after it has been transmitted from a source to a destination. A CRC generates values from the data before it is transmitted. These values are then compared to the generated values of the data after transmission. If the values before and after transmission are not equivalent, then the CRC can identify that the data has been corrupted. A CRC generates values by performing modulo-2-polynomial division on sections of the data [3] which effectively generates a hash value for each section.

In AVs, CRC checking is a crucial feature that guarantees the functionality of its safety system. The features of an AV like a rear-view camera, warning system, and lane control all interact with the driver through the display. Therefore, running CRC checks ensure that the driver is receiving accurate and timely information from the screen. Furthermore, these checks can help identify dead pixels or dead spots on the screen if a series of frames are flagged as corrupted.



```
"core-surface": {
    "Compositor": 1720969782,
    "Output": 2913663874,
    "Raster Generator": 3777668590
}
```

**Figure 5:** The Hash Values of a Single Frame in CRC

In this project, CRCs were utilized to detect transmission errors for each frame that is projected onto the screen. When any client wants to display a visual onto the screen, the device driver creates the final image by compiling the desired surfaces into a single buffer. This buffer is serialized and fed into the display for projection. A CRC partitions the buffer and generates three different values: raster generator, compositor, and output. With these values, the CRC can identify corruption if the hashes before serialization differ from the hashes after serialization.

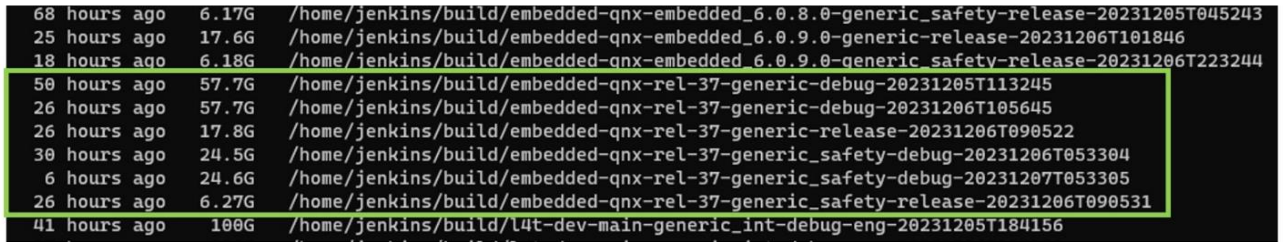
## 3) METHODS

The test integration cycle was critical for our methods used because of the flow in relation to the development and testing portions of the project. NVIDIA DRIVE OS Tegra Display team utilizes physical Drive AV boards to deploy changes within the codebase to verify its functionality with in-code test cases and physical verification from the developers. The build process confirmed functionality of any changes within the codebase. After functionality was confirmed the verification process within the CI/CD pipeline tested our changes against the entire DRIVE OS codebase. After changes were confirmed within DRIVE OS codebase there were documentation changes for the code within the system.

### 3.1 Build Process

NVIDIA DRIVE OS Tegra Display team has three different boards p3663-a01, p3663-a01-f1, and p3710-10-a01 to run and verify code changes that were made. There are three main relationships between our build process to develop, build, and flash developed code onto any of the Drive AV computer boards. The process for developing code within the team and deploying to a computer board is the following (also shown in Figure 7):

1. **Drive Farm:** Software developers at NVIDIA use Drive Farm which is a cloud environment that hosts daily trees which are DRIVE OS builds that are synchronized between all software teams working on DRIVE OS. Different build environments are needed based on which team a software developer is on. The Tegra Display team and the MQP team used rel-37 builds, which is a combination of both Linux and QNX operating systems. Figure 6 below shows the various builds that we could use, which change based on different tasks being worked on. After a developer grabs/snapshots a DRIVE OS tree, they can make any necessary changes within the codebase and then build an image for their current tree.



```
 68 hours ago   6.17G   /home/jenkins/build/embedded-qnx-embedded_6.0.8.0-generic_safety-release-20231205T045243
 25 hours ago   17.6G   /home/jenkins/build/embedded-qnx-embedded_6.0.9.0-generic-release-20231206T101846
 18 hours ago   6.18G   /home/jenkins/build/embedded-qnx-embedded_6.0.9.0-generic_safety-release-20231206T223244
 50 hours ago   57.7G   /home/jenkins/build/embedded-qnx-rel-37-generic-debug-20231205T113245
 26 hours ago   57.7G   /home/jenkins/build/embedded-qnx-rel-37-generic-debug-20231206T105645
 26 hours ago   17.8G   /home/jenkins/build/embedded-qnx-rel-37-generic-release-20231206T090522
 30 hours ago   24.5G   /home/jenkins/build/embedded-qnx-rel-37-generic_safety-debug-20231206T053304
  6 hours ago   24.6G   /home/jenkins/build/embedded-qnx-rel-37-generic_safety-debug-20231207T053305
 26 hours ago   6.27G   /home/jenkins/build/embedded-qnx-rel-37-generic_safety-release-20231206T090531
 41 hours ago    100G   /home/jenkins/build/l4t-dev-main-generic_int-debug-eng-20231205T184156
```

**Figure 6:** Tree Search Output from Drive Farm

2. **SCP Files to Host Computer:** to properly deploy changes onto the Drive AV computer there is a middleman host computer that is connected to the Drive AV computer. From a developer's Drive Farm environment, the developer uses SCP (secure transfer protocol) to transfer files from the build over to the host computer.

3. **Flash OS to Drive AV Computer:** After all files have been deployed onto the host computer, the new DRIVE OS is flashed onto the Drive AV board. The flashing process removes the entire OS that was on the Drive AV board and then installs the new DRIVE OS that is on the host computer.

4. **Run Tests:** The last part of the pipeline is to run the necessary test cases that were changed by the developer. This is done with assertions within the codebase and sometimes with visual verification from the developer looking at the display next to the Drive AV board.

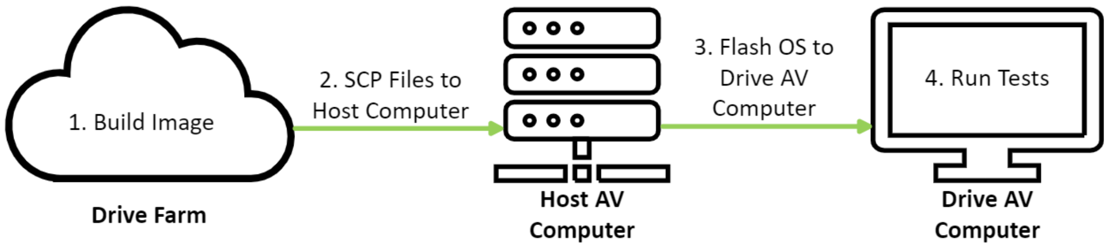


**Figure 7:** Local Build and Deployment Process

## 3.2 Verification Process (CI/CD Pipeline)

Once a tree is copied (snapshot) from cloud to development environment Drive Farm, we can run coverage deviation on a clean repository to create a baseline (see Section 3.2.4), which is a one time setup for each tree (shown in Figure 8 blue section).

NVIDIA uses Gerrit for code review. The review process (shown in Figure 8 green section) requires developers to push changes as a patchset with the covdev summary of the changes, local verification, +1 from a reviewer, +2 from another reviewer, and a successful run from GVS. When all requirements are met, the committed patchset is automatically picked by the promotion cycle and merged to the rel-37 branch of the code base.

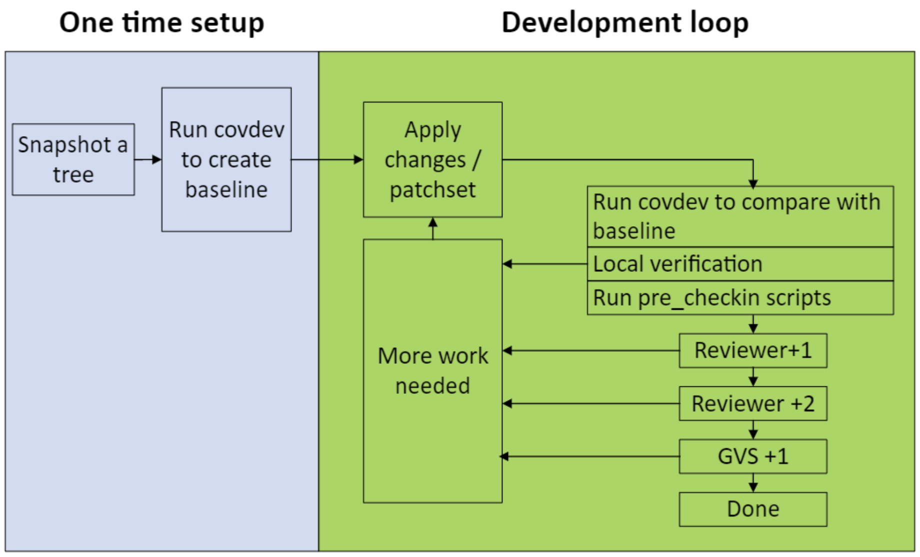


**Figure 8:** Development Cycle

### 3.2.1 Local Verification

Local verification involves building and flashing the image onto the target board. With standard builds, tests are located in the /samples folder. Safety builds do not have Tegra display test binary packaged, so we need to scp the test binary to the board /tmp folder. For each test developed, a local verification is necessary. The command to run tests varies depending on the test mode and steps needed. For example:

1) To run a surface layout test:

```
/samples/tegradisp/tegradisp-test -t surface-layout -v -D
```

2) To run a window notifier test that requires CRC check:

```
/samples/tegradisp/tegradisp-test -t window-notifier -H roc -G /tmp -v -D
```

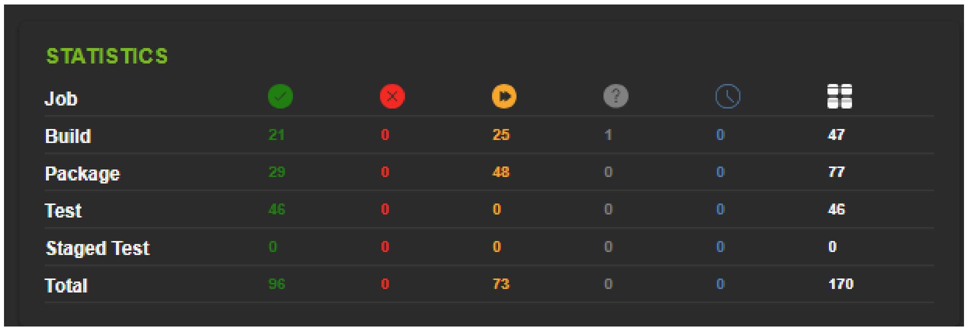
If the test and visual verification (if needed) passes, then the local verification is completed. If the test or visual verification fails, then further investigations of the failure is needed. Running `slog2info | grep modeset | grep ERROR` on the board root terminal returns a list of error logs of the previous activities.

### 3.2.2 Pre-Check-In Script

The pre-check-in script is the second part of local verification. This script runs all other tests related to the display driver and its associated IOCtl calls. After finishing and verifying our own test, we scp the latest version of the pre-check-in script to the board. We then execute this bash script which automates the process of running all display tests. The script prints the result of each test and its associated logs which we look over for errors. This script guarantees that our changes have not influenced other tests and ensures that all tests continue to work properly. Once we verify that every test in the script passes, our changes can then be reviewed by senior developers for +1 and +2 approval.

### 3.2.3 GVS

Gerrit Virtual Submit (GVS) is a testbot responsible for running a developer's changes against the entirety of DRIVE OS. When a change is submitted to GVS, the testbot prepares a remote server, implements the change into the codebase, and builds every system image related to DRIVE OS. GVS then installs every package required for the operating system and begins executing scripts to pressure test the new change with the rest of the codebase.



STATISTICS

| Job | ✓ | ✗ | ⏩ | ? | 🕒 | ▦ |
|---|---|---|---|---|---|---|
| Build | 21 | 0 | 25 | 1 | 0 | 47 |
| Package | 29 | 0 | 48 | 0 | 0 | 77 |
| Test | 46 | 0 | 0 | 0 | 0 | 46 |
| Staged Test | 0 | 0 | 0 | 0 | 0 | 0 |
| Total | 96 | 0 | 73 | 0 | 0 | 170 |

**Figure 9:** Sample GVS Report after Submitting Changes

Like the pre-check-in script, GVS guarantees that our changes do not corrupt any downstream operations for not only display tests but for the entirety of DRIVE OS that other teams are developing on. GVS represents the final step of our verification process and serves as the final approval before our changes are merged into the codebase.

If a change fails at any part of the GVS run, then this change cannot be merged into the codebase. This policy ensures that the code on GVS remains correct so that all developers can pull this codebase and begin developing changes on a fresh branch. The state of GVS allows us to compare our local environment to a golden standard to help us search for any inconsistencies on our drive farm.

### 3.2.4 Covdev

Covdev is a NVIDIA internal auditing tool for coverity (static code analysis) which provides accurate delta between 2 baselines with deviation information. Covdev can detect code violations against MISRA C, which is a set of software development guidelines for C programming language developed by MISRA (Motor Industry Software Reliability Association).

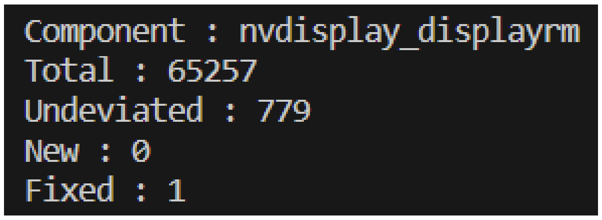
For example, to create a baseline for display driver repository, developers must be on top of tree (no changes), and run:

```
covdev --create-baseline display_driver
```

After committing changes, command below can be run to compare with the baseline:

```
covdev --compare-baseline display_driver -m -n
```

The output is a summary of deviated information on code violation of the new changes compared to the baseline as shown in Figure 10.  It reports how many violations are there total in the code repo, number of undeviated violations, number of new violations introduced by the new change, and number of violations that are fixed by the new change. The summary is automatically appended to the commit message.



```
Component : nvdisplay_displayrm
Total : 65257
Undeviated : 779
New : 0
Fixed : 1
```

**Figure 10:** Sample covdev Summary Output

## 3.3 Test Integration and Documentation

For each new test developed, after it has been merged into the development branch, it needs to be added to the GVS script so that it can be integrated into the CI/CD pipeline for future development. The test also needs to be added to the SWITS (Software Integration Test Specification) documentation for safety compliance, including the test name, description, list of IOCtl APIs used, requirement id, test design, test procedure, and expected result.

# 4) IMPLEMENTATION

To fulfill the project objectives, the team has completed various test cases, including positive and negative IOCtl test cases, head to window mapping test cases, and operating system test cases. In addition to tests, the team also performed refactoring work on the existing codebase such as removing outdated code and compiling out certain functionalities on safety builds.

## 4.1 Test Cases

Our team worked on various test cases for IOCtls in the display driver. These tests ranged from positive to negative tests that verified the behavior of IOCtls when provided both valid and invalid inputs. The IOCtl tests were extensively documented and grouped by the IOCtl they manipulated. Details such as test file name, test function name, and verification method are all documented alongside the concepts related to the test.

### 4.1.1 Register/Unregister IOCtl Tests

The register surface and unregister surface IOCtls are responsible for allocating and deallocating a surface in memory, respectively. The IOCtls accept parameters that reference the surface and describe the properties of the surface that will be allocated or deallocated. The tests in this section detail the behavior of the register and unregister surface IOCtls when the parameters are manipulated to both valid and invalid inputs.

#### *Surface Register Device Handle Negative Test*

| | |
|---|---|
| Test File Name | tegradisp-test-surface.c |
| Test Case Name | nvKmsTestRegisterSurfaceNegative (new) |
| Negative / Positive | Negative |
| IOCTL Calls Used | RegisterSurface |
| Verification Method | IOCtl fails |
| Bugs Found? | No |

## *Purpose*

This test investigates what happens to the register-surface IOCtl when an invalid device handle is provided. An invalid device handle is defined as any index that results in a null pointer (no device information) being returned. We expect the register-surface IOCtl to fail as registration should not be possible with an undefined memory location for devices.

## *Concepts & Relevant Background*

A device represents a single GPU used for the display. When a device is allocated, all the information is loaded into RAM. When a register-surface call occurs, the IOCtl uses the device handle to find the appropriate device information from memory and loads that information to associate surface information with the device. The register-surface IOCtl then validates all device information before allocating a surface that the device will own. The device handle is simply an unsigned 32-bit integer. This device handle serves as an index into an array of pointers, where the pointer is the location of the device information. A device handle of 0 is an invalid device handle as a user cannot request no information when requesting device information.

## *Test Step*

**Setup:**

1. Initialize fd, pDevice, pConfig, and params
2. Create invalid device handles (min handle value, one greater than device handles allocated, handle in between min and max handle values, max handle value)

**Test:**

1. For each invalid device handle, set the device handle of params to the value and attempt to register-surface
2. Expect the register-surface IOCtl call to fail due to referencing an invalid location in memory

**Test tear down:**

1. Free pConfig
2. Free pDevice

3. Close fd

### *Verification*

For this test, registering a surface using the various invalid device handles should cause the IOCtl to fail, and the tester should be unable to see anything on the display screen.

## *Surface Register Device Sizes Negative Test*

| | |
|---|---|
| Test File Name | tegradisp-test-surface.c |
| Test Case Name | nvKmsTestRegisterSurfaceNegative (new) |
| Negative / Positive | Negative |
| IOCTL Calls Used | RegisterSurface |
| Verification Method | IOCtl fails |
| Bugs Found? | No |

### *Purpose*

This test investigates what happens to the register-surface IOCtl when registering a surface that has dimensions that cannot be represented by the display. For example, registering a surface with 0 pixels of width and height, or a surface that is larger than the screen is invalid. The register-surface IOCtl should fail as the display should not try to render surfaces it cannot physically handle.

### *Concepts & Relevant Background*

A device holds all information pertaining to the heads and layers of the display and can be used to derive the configurations of the display. The configuration of the display holds information like the dimensions of the screen in pixels and the mode timings of the display. When a register-surface call occurs, the dimensions of the surface must be specified. Before the surface can be allocated, the specified dimensions are validated against the dimensions of the display. The register-surface call will fail if the requested dimensions for the surface exceeds the dimensions of the display.

### *Test Step*

**Setup:**

1. Initialize fd, pDevice, pConfig, and params
2. Create invalid surface sizes (min width and height, invalid width with valid height, valid width and invalid height, invalid width and invalid height, max width and height)

**Test:**

1. For each invalid surface size, set the width and height of params to the value and attempt to register-surface
2. Expect the register-surface IOCtl call to fail due to allocating a surface that cannot fit on the display

**Test tear down:**

1. Free pConfig
2. Free pDevice
3. Close fd

### *Verification*

For this test, each invalid surface dimension should fail the register-surface IOCtl, and the tester should see nothing on the display screen.

## *Surface Unregister Invalid Handle Test*

| | |
|---|---|
| Test File Name | tegradisp-test-surface.c |
| Test Case Name | nvkmsTestSurfaceUnregisterNegative (new) |
| Negative / Positive | Negative |
| IOCTL Calls Used | SurfaceUnregister |
| Verification Method | Test case pass |
| Bugs Found? | Yes |

## *Purpose*

The purpose of this test was to verify that the surface unregister IOCtl fails when feeding in an invalid surface buffer handle during surface deregistration or attempting to deregister the same surface twice.

## *Test Step*

**PART 1: Invalid Buffer Handle**

**Setup:**

1. Initialize all pointer values to null (pDevice, pConfig, pSurface), fd = -1
2. Open NvKMS
3. Allocate pDevice

**Test:**

1. Unregister an unallocated surface (if any fails, go to done)
    a. surfaceHandle = 0 (invalid range, NULL pEvoSurface)
    b. surfaceHandle = 1 (valid range, NULL pEvoSurface)
    c. surfaceHandle = 30 (valid range, NULL pEvoSurface)
    d. surfaceHandle = 0xFFFFFFFF (invalid range, NULL pEvoSurface)

**PART 2: Unregister The Same Surface Twice**

**Setup:**

1. I nitialize pConfig
2. Allocate pSurface

**Test:**

1. Deregister pSurface
2. Deregister pSurface again

**Test tear down:**

1. free pConfig
2. free pSurface
3. free pDevice
4. close(fd)

*Verification*

Part 1 of the test should result in unregister surface IOCtl to fail with error that it attempts to unregister null surface pointer.

In part 2 of the test, the first deregister call should succeed, and then second one should fail.

*Debugging*

1. With the test properly setup, it was observed that the test failed in part 1. During some investigation, it was discovered that in surface unregister IOCtl source code, deregister a null surface pointer did not return a false status. The solution was to add a fatal error when surface pointer is null.

2. With the solution to the first bug implemented, there were then failures with free device and releasing ownership IOCtl during test tear down for all test cases. During investigation of source code for free device IOCtl, it was discovered that a device console surface handle was also being unregistered. Since the console surface handle is no longer in use, attempting to free it resulted in a null surface pointer and caused a fatal error during free device process. The solution was to remove the unused console surface handle and its relevant functions.

### 4.1.2 Flip IOCtl Tests

A flip IOCtl is responsible for projecting an image onto the display screen. The flip IOCtl accepts parameters that define which layers will be populated by what surface and the specified head that the layers will be pushed to. The tests in this section detail the behavior of the flip IOCtl when the parameters are manipulated to both valid and invalid inputs.

#### *Alpha-Opaque Blending Positive Test*

| | |
|---|---|
| Test File Name | tegradisp-test-flip.c |
| Test Case Name | NvKmsTestLayerAlphaBlending (edited) |

| | |
|---|---|
| Negative / Positive | Positive |
| IOCTL Calls Used | Flip |
| Verification Method | Visual, CRC |
| Bugs Found? | No |

### *Purpose*

This test verifies that flipping layers with adjusted alpha values on different alpha-blending modes will display correctly by showcasing a changing transparency.

### *Concepts & Relevant Background*

When a surface is flipped onto a screen, the blending mode of the layer must be defined prior to the flip occurring. The blending mode of layers defines the transparency of a layer. Defining a layer with alpha blending enabled allows the color to have a percentage of transparency take effect, which allows colors to blend when two layers lie atop each other. A layer with opaque blending has no transparency. As a result, underlying colors do not show from beneath an opaque layer. For alpha blending there are multiple modes defined: pre-multiplied surface, pre-multiplied pixel, non-pre-multiplied surface, and non-pre-multiplied pixel. There is also an entirely transparent blending mode. Each of these modes defines how the transparency of each pixel is calculated.

### *Test Step*

**Setup:**

1. Initialize flipState, pDevice, and an array with all blending modes
2. Allocate a flipState
3. Set the device from flip state
4. Test and check with CRC a red screen to make sure flipping works properly

**Test:**

1. Flip a surface with opaque blending (doesn't matter what the alpha value is since opaque ignores alpha)

2. Flip a surface with the remaining blending modes by looping through the alpha values from 0.0 to 1.0 at increments of 0.1 with each blending mode

3. Check every flip with a CRC

4. Expect all CRC checks to pass

**Test tear down:**

1. Free flip state

### Verification

The test flips a red screen on the display. A green surface is then flipped atop the red screen. The transparency levels of the green screen are then adjusted, and a yellow screen should become more visible as the transparency of the green screen goes down. This should be verified five times for the different blending modes.

### Further General Work

Working on the positive test for alpha-opaque blending revealed an error within GVS. When pulling the CRCs from GVS and running them on a local environment, every CRC file failed against the check locally. This behavior indicates that GVS CRCs were either incorrect or that CRC checks were not running on GVS correctly. On further investigation, the error on GVS stemmed from a malformed test script that was not properly executing tests with a CRC check. This script allowed all incorrect CRCs to pass on GVS and merge into the main codebase. Therefore, a fix was issued to both correct the script and all CRCs currently on GVS. With this fix, CRC checking now runs properly on GVS for all current and future tests.

## *Alpha-Opaque Blending Negative Test*

| | |
|---|---|
| Test File Name | tegradisp-test-flip.c |
| Test Case Name | NvKmsTestLayerAlphaBlendingNegative (new) |
| Negative / Positive | Negative |
| IOCTL Calls Used | Flip |
| Verification Method | IOCtl fails |
| Bugs Found? | No |

## *Purpose*

This test investigates what happens when two layers with identical depths are flipped onto the same head. Identical depths are an errant input and cannot be rendered by the display, resulting in the flip IOCtl to fail.

## *Concepts & Relevant Background*

When a surface is flipped onto a screen, the depth of the layer must be defined prior to the flip occurring. The depth of the layer defines which layer lies atop another layer on the screen. The lower the depth, the closer to the top that layer is. For example, a layer with a depth of 0 that takes up the entire height and width of the screen prevents a layer with a depth of 1 from being seen.

## *Test Step*

**Setup:**

1. Initialize flipState, pDevice, compParams, size, and rrParams
2. Allocate a flip state
3. Set the device from flip state
4. Set width and height

**Test:**

1. Create compParams with opaque blending and depth 3
2. Flip green surface with depth 3
3. Flip yellow surface with depth 3
4. Expect flip to fail on yellow surface due to same depth
5. Clear flips
6. Change compParams to alpha blending
7. Flip green surface with depth 3
8. Flip yellow surface with depth 3
9. Expect flip to fail on yellow surface due to same depth

**Test tear down:**

1. Free flip state

### Verification

For the negative test, the flip IOCtl should fail on the second flip of each blending mode. When the first surface is loaded onto a layer with a depth of 3, there are no other layers flipped concurrently, which means no conflicts occur with rendering the depths of the layers. However, on the second flip of a layer with a depth of 3, there is now a conflict between the second layer being flipped and the first layer that was already flipped. Flip validation should identify this conflict and fail the IOCtl call.

## Flip with Invalid Surface Handles Test

| | |
|---|---|
| Test File Name | tegradisp-test-flip.c |
| Test Case Name | nvKmsTestFlipBase (new) |
| Negative / Positive | Negative |
| IOCTL Calls Used | Flip |
| Verification Method | IOCtl fails |
| Bugs Found? | No |

### Purpose

This test investigates what happens to the flip IOCtl when an invalid surface handle is provided for the flip. An invalid surface handle is defined as any index that results in a null surface pointer (no surface information) being returned. The flip IOCtl is expected to fail because a flip is not be possible with an undefined memory location.

### Concepts & Relevant Background

Whenever a surface is allocated, the information of this buffer is loaded into RAM. After loading into RAM, a surface handle is generated which allows any client to reference the information of a surface and read its buffer from RAM. The surface handle is an unsigned 32-bit integer. Any client can access any of the allocated surfaces if they use a valid surface handle. Surfaces are not allocated for any singular client. When a flip occurs, all surface handles are checked for validity to ensure that no undefined memory locations are used for a flip. A surface handle of 0 for the flip IOCtl is a valid surface handle and represents an unpopulated layer that is ignored when the flip displays all defined layers onto the screen.

### *Test Step*

**Setup:**

1. Initialize flipState, validSurface, invalidSurface, size, compParams, and rrParams
2. Allocate a flipState
3. Set width and height
4. Test and check a flip with validSurface to make sure flipping works properly

**Test:**

1. Create invalid surface handles (one greater than allocated, in between max pointers allowed, above max pointers allowed, the highest value of an unsigned 32-bit integer)
2. For each invalid surface handle, set the surface handle of the invalidSurface to the value and attempt to flip using the invalidSurface
3. Expect the flip to fail due to referencing an invalid location in memory

**Test tear down:**

1. Free flip state

### *Verification*

For this test, a flip with a valid surface handle is performed first to verify that the flip IOCtl can display a visual onto the screen if the surface handle references the correct location in memory. After verifying this behavior, the test can then use a range of invalid surface handles and attempt to flip using these handles. Each of the invalid surface handle flips should fail, and the tester should be unable to see anything on the display screen.

### *Flip with Invalid Layer Test*

| | |
|---|---|
| Test File Name | tegradisp-test-flip.c |
| Test Case Name | nvKmsTestFlipBase (new) |
| Negative / Positive | Negative |
| IOCTL Calls Used | Flip |
| Verification Method | IOCtl fails |
| Bugs Found? | No |

#### *Purpose*

This test investigates what happens to the flip IOCtl when a single flip occurs and it attempts to flip multiple layers and at least one invalid layer is defined for the flip. All layers in a flip must be validated before a flip occurs. Therefore, the flip IOCtl is expected to fail if it cannot use some of the information in the input.

#### *Concepts & Relevant Background*

Whenever a flip is performed, the specifications of the flip are validated by checking the fields of the heads and layers. Within a function called that validates flip parameters within the flip IOCtl, the layers of a flip are checked for violations. Violations such as flipping layers on inactive heads, flipping layers with the same depth, and flipping layers with an invalid surface handle are all caught by this function. The function performs the previous check iteratively on each head and layer. When the first violation is caught, the function will fail the IOCtl immediately.

#### *Test Step*

**Setup:**

1. Initialize flipState, size, params, compParams, rrParams, invalidSurface
2. Allocate a flipState
3. Set width and height

**Test:**

1. Create an invalid surface handle and set the surface handle of invalidSurface to this value

2. Prepare two layers to flip: valid surface on 0th layer, invalid surface on 1st layer
3. Expect the flip IOCtl to fail due to one layer being defined incorrectly

**Test tear down:**

1. Free flip state

### *Verification*

For this negative test, the flip IOCtl should fail after the flip call. During flip validation, the flip call loops through each layer defined in params. There should be no issues validating the first layer defined in parameters but should flag the second layer due to the invalid surface handle. The flip IOCtl should subsequently fail. The tester should be unable to see anything on the display screen.

## *Flip with Extra Layers Test*

| | |
|---|---|
| Test File Name | tegradisp-test-flip.c |
| Test Case Name | nvKmsTestFlipBase (new) |
| Negative / Positive | Negative |
| IOCTL Calls Used | Flip |
| Verification Method | IOCtl fails |
| Bugs Found? | No |

### *Purpose*

This test investigates what happens to the flip IOCtl when it attempts to flip more layers than currently allocated to a head. Since a head has a limited number of layers that it can use to help render visuals, an extra layer cannot be rendered by the head. Therefore, the flip IOCtl is expected to fail if it cannot use some of the information in the input.

### *Concepts & Relevant Background*

From the configuration file, a device recognizes the number of layers allocated to each head. When a device attempts to flip several layers onto the screen, each layer is counted and checked against the allocation in the configuration file. If the number of layers being flipped for a head exceeds the number of layers allocated to that head, then the flip IOCtl errors out on validation, and the flip is not performed.

### *Test Step*

**Setup:**

1. Initialize flipState, size, params, compParams, and rrParams
2. Allocate a flipState
3. Set width and height

**Test:**

1. Prepare to flip numLayers+1 valid layers
2. Flip params which will simultaneously flip the numLayers+1 layers
3. Expect the flip to fail due to flipping more layers than allocated to a head

**Test tear down:**

1. Free flip state

### *Verification*

For this negative test, the flip IOCtl should fail after the flip call. During flip validation, the flip call loops through each layer defined in the parameters. There should be no issues validating the layers that fit within the allocation of the head. However, on the extra layer, flip validation should error out because it has identified a defined layer that is outside its allocation of layers. The tester should be unable to see anything on the display screen.

### *Debugging*

When creating this test, flip validation initially failed at recognizing extra layers and chose to silently ignore these layers instead of failing the IOCtl call. The extra layers were ignored due to the configuration of the device. Since the device recognized the number of layers allocated to each head at runtime, it would not attempt to look at layers beyond this allocation. Therefore, extra layers would never be validated**,** and the flip would happen successfully. However, this is not the intended behavior for the flip IOCtl. Therefore, changes were made to the flip validation source code. The flip IOCtl now checks layers beyond what is allocated for the device in the flip. If the number of layers defined in the flip exceeds the number of layers allocated, then the IOCtl now fails. Having the flip IOCtl validate layers outside the number allocated to a head allowed the flip IOCtl to correctly identify and catch extra layer flips.

### *Overlay Minimal Scaling Test*

| | |
|---|---|
| Test File Name | tegradisp-test-flip.c |
| Test Case Name | nvKmsTestMinimalScaling (new) |
| Negative / Positive | Positive |
| IOCTL Calls Used | Flip |
| Verification Method | Visual, CRC |
| Bugs Found? | No |

#### *Purpose*

The purpose of this test was to verify overlay scaling succeeds when there is 1 row and 1 column changes for each direction (up or down).

#### *Concepts & Relevant Background*

The existing overlay scaling test consists of a sequence of maximal up and down scaling. It is not feasible to add to the CI/CD pipeline because it requires a specific setting for IMP (is mode possible) configuration, which configures device timing, clock, and other settings. The change on IMP settings could potentially break other tests on GVS. Since this test is a minimal scaling test, it is less stressful and can be easily integrated to GVS, so that overlay scaling functionality can be tested within the CI/CD pipeline. The scaling depends on input size and output size parameters when calling the flip overlay function.

#### *Test Step*

**Setup:**

1. Allocate FlipState

2. Get pOverlaySurface from FlipState and set it to CMYKWRGB color bars

**Test:**

1. Perform a combination of 9 scaling tests with different height and width (no scaling, min downscaling, min upscaling) in different directions
2. Verify CRC

**Test tear down:**

1. Free FlipState

*Verification*

The first flip should display CMYKWRGB color bars with full screen size (see Figure 11). The CRC check should pass. The second flip should also have the CMYKWRGB color bars but with a very slight yellow line on the right and bottom (see Figure 12). The same verification is needed for the rest of the directions used in scaling.

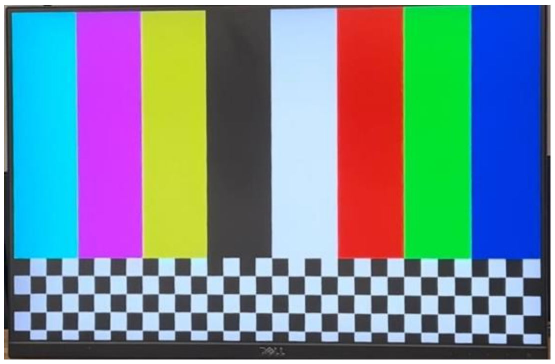


**Figure 11:** First Flip

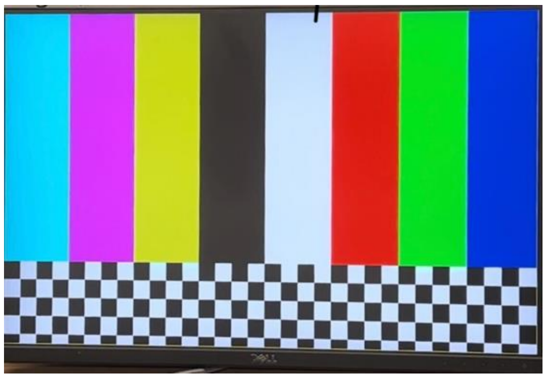


**Figure 12:** Second Flip

*Further Work*

Currently, the test only passes the first three flips (original in and out, height minimal down scaling, and height minimal up scaling). The display shows no further updates with the rest of the test cases, and eventually hangs at the last test case (minimal upscaling for both height and width). The suspected reason of failure is incorrect IMP configurations, but more investigation is required for the test case to pass.

## *Window Completion Notifiers Test*

| | |
|---|---|
| Test File Name | tegradisp-test-syncpt.c |
| Test Case Name | nvKmsTestWindowNotifier (new) |
| Negative / Positive | Both |
| IOCTL Calls Used | Flip |
| Verification Method | Visual, CRC pass, Notifier State |
| Bugs Found? | No |

### *Purpose*

The purpose of this test was to verify window completion notifier states with a successful flip, an outright failed flip, and a sequence of flips on different layers are as expected.

### *Concepts & Relevant Background*

The window notifier surface is a non-ISO surface used to report the status of a flip on another allocated surface. For the notifier to update a status, the pointer to a notifier surface needs to be passed in a part of the flip parameter.  The window notifier can have the following status: NOT_BEGUN, BEGUN, and FINISHED. When a flip is blocked or failed, the window notifier status should be NOT_BEGUN, which is also the initialized status. When a non-null flip has successfully being performed, the notifier status is updated to BEGUN to inform that the display is continuously scanning out pixel values from the surface memory. When a null flip is performed, the notifier state is updated to FINISHED to inform that the display is no longer scanning from the surface memory.

### *Test Step*

**Setup:**

1. Initialize pDevice, pConfig, fd, pSurface[], pNotif[]
2. open nvkms
3. allocate device
4. allocate config
5. grab ownership

6. initialize the maximum number of layers (numLayers) that is configured on the device

7. allocate an array (numLayers size) of surface with decreasing window sizes and populate the surfaces with alternating color (red and green)

8. allocate an array (numLayers size) of notifier surfaces

**Test:**

**1. Test failed flip with invalid handle**

This is the first part of the whole test, where we pass in the first surface registered in the array (pSurface). Then, another surface is registered inside this partial test, which is used as an invalid surface by manually setting pInvalidSurface->surfaceHandle to 0xffffffff (pInvalidSurface). It is also populated to green with a smaller window size.

1) Perform flip on pSurface, check notifier state
2) Check CRC
3) Reset the notifier
4) Perform flip on pInvalidSurface, check notifier state
5) Check CRC

**Tear down:**

restore pInvalidSurface->surfaceHandle to original value and free pInvalidSurface

**2. Test multiple window flip**

This is the second part of the whole test, where we perform a sequential flips on the array of pSurfaces and check the corresponding notifier state and CRC value after each flip.

**Test tear down:**

1. Free notifiers
2. Free surfaces
3. Release ownership
4. Free config
5. Free device
6. Close fd

*Verification*

For the first part of the test, after the first flip, which should succeed, the notifier state should be BEGUN, and the screen is all red with a CRC check. The second flip should report an

IOCtl flip failure. Verify that the notifier state is NOT_BEGUN and the screen remains red with another CRC check.

In the second part of the test, for each flip of the surface pointer array (from i to numLayers), verify that the notifier state for pNotif[j] for j <=i is BEGUN, and otherwise NOT_BEGUN. Therefore, any happened flip should have state BEGUN and remain BEGUN, and any unhappened flips should have state NOT_BEGUN till the flip happens. The window should have alternating red and green boxes with decreasing size. The test verifies CRC after each flip.

### *Transfer Functions Degamma/Regamma Test*

| | |
|---|---|
| Test File Name | tegradisp-test-transfer-functions.c |
| Test Case Name | nvKmsTestTransferFunctions (new) |
| Negative / Positive | Negative |
| IOCTL Calls Used | Flip, Query CRC |
| Verification Method | Visual, CRC |
| Bugs Found? | No |

#### *Purpose*

This test case specifically feeds in non-surface gamma values into the different gamma surface, which cause the CRC checks to fail (shown in Figure 13). The transfer-functions test was expanded to verify that different degammas are used for linear vs sRGB/Rec709 inputs, and that the same degamma is used for sRGB and Rec709 inputs. The surface is specified as sRGB or Rec709, which communicates to the driver via flip parameters. We create a mismatch by applying a linear transfer function on the actual pixels. Therefore, we can compare the output CRCs on the same input (linear) pixels when the driver is told the surface is Linear, sRGB, or Rec709. This allows us to confirm that the driver degamma selection logic is working as intended.

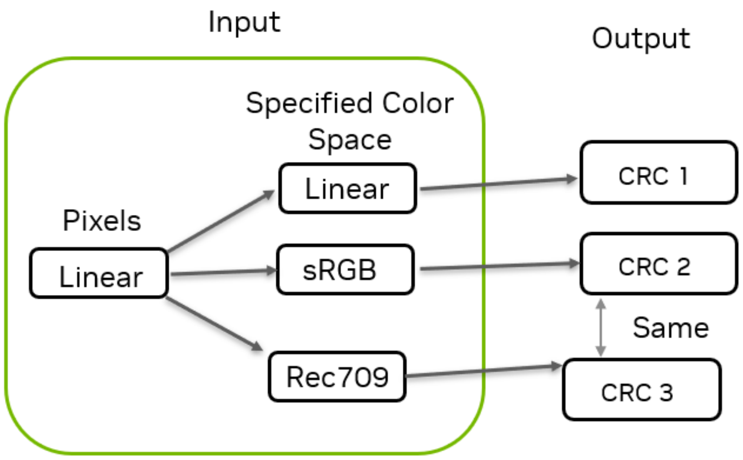


**Figure 13:** Negative Transfer Function Test

*Concepts & Relevant Background*

The previous testing of the transfer function (shown in Figure 14) only tests out direct one-to-one mapping of the degamma and regamma transfer protocols onto the screen. Before, this generalized test case only tested out the direct mapping with the static CRC JSON files within storage. This works when the transfer functions for linear, sRGB, and Rec709 are transferred onto the screen and then checked with the static CRC JSON files.

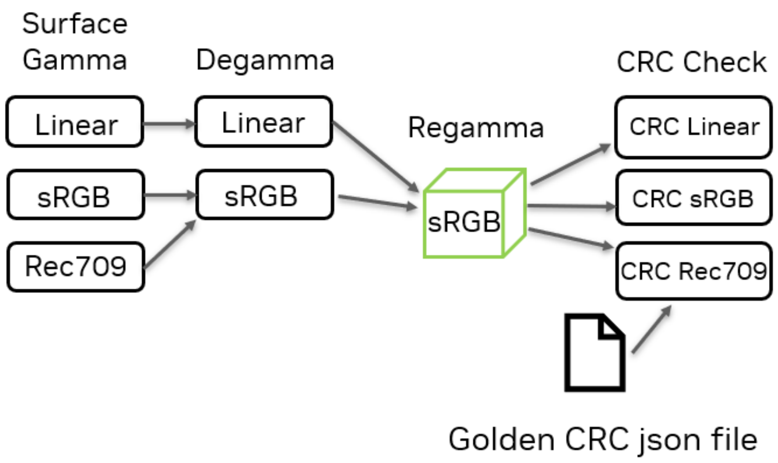


**Figure 14:** Positive Transfer Function Test

*Test Step*

**Setup:**

1. Initialize pDevice, pConfig, fd, pSurface, internalCRC[2]

2. Open nvkms
3. Allocate device
4. Allocate config
5. Grab ownership

**Test:**

1. Iterate through sRGB and Rec709 gamma surfaces
    a. Allocate surface for current gamma
    b. Draw gradient bands with linear gamma (this fills up the pixel buffer)
    c. Flip buffer onto screen
    d. Query current CRC values for the screen
    e. Initialize CRC values within internalCRC at the current gamma surface
2. Verify that the internalCRC raster, output, and compositor values at sRGB are not equal to the internalCRC raster, output, and compositor values at Rec709.
3. Verify that the CRC of sRGB from internalCRC and linear are not equal to each other

**Test tear down:**

1. Free Config
2. Release Ownership
3. Free Device
4. Close nvKms file descriptor

## *Verification*

Verify that the internal CRC raster, output, and compositor values at sRGB are not equal to the internal CRC raster, output, and compositor values at Rec709. The second check will then verify that the CRC of sRGB from internal CRC and linear are not equal. Figure 15 below shows the normal gamma transfer from the positive test case, while Figure 16 below shows the negative test case causing the displacement within the pixels when doing the degamma/regamma.

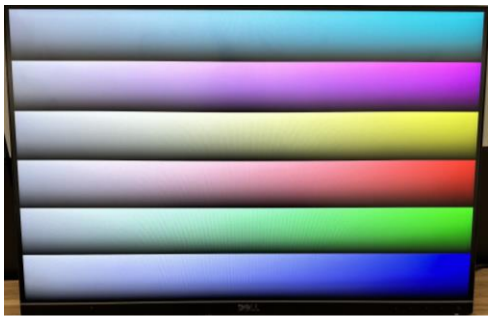


**Figure 15:** Normal Gamma Transfer

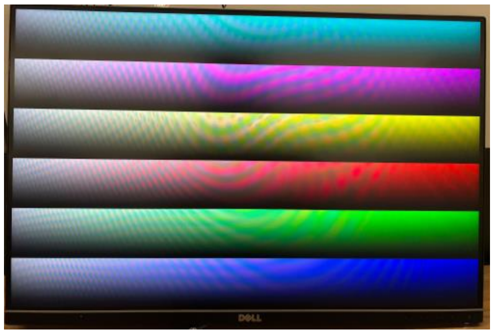


**Figure 16:** Negative Gamma Transfer

### 4.1.3 Head-Window Mapping Config Tests

The head to window mapping within the drive tree configuration is a critical part within the ODD since it tells the kernel level software how many heads and windows are allocated for whichever board it is on. All examples below are from the Orin board head to window mapping which has 1 head and 4 windows valid windows.

#### *Example 1 (Valid window on Invalid head):*

**Description:** Map window 0 (valid) to head 3 (invalid)

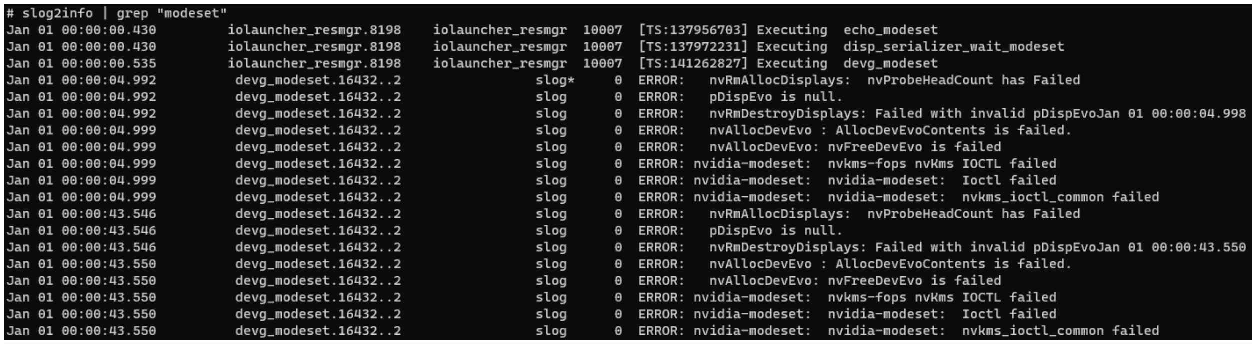
64-bit layout:

```
00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000011
w_7       w_6       w_5       w_4       w_3       w_2       w_1       w_0
```

Drive Config Layout:

```
display@13800000 {
        nvidia,window-head-mask = <0x00000000 0x00000003>;
};
```

Driver Failure to Load from slog2info:



```
# slog2info | grep "modeset"
Jan 01 00:00:00.430         iolauncher_resmgr.8198    iolauncher_resmgr  10007  [TS:137956703] Executing  echo_modeset
Jan 01 00:00:00.430         iolauncher_resmgr.8198    iolauncher_resmgr  10007  [TS:137972231] Executing  disp_serializer_wait_modeset
Jan 01 00:00:00.535         iolauncher_resmgr.8198    iolauncher_resmgr  10007  [TS:141262827] Executing  devg_modeset
Jan 01 00:00:04.992          devg_modeset.16432..2                 slog*      0  ERROR:   nvRmAllocDisplays:  nvProbeHeadCount has Failed
Jan 01 00:00:04.992          devg_modeset.16432..2                 slog       0  ERROR:   pDispEvo is null.
Jan 01 00:00:04.992          devg_modeset.16432..2                 slog       0  ERROR:   nvRmDestroyDisplays: Failed with invalid pDispEvoJan 01 00:00:04.998
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR:   nvAllocDevEvo : AllocDevEvoContents is failed.
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR:   nvAllocDevEvo: nvFreeDevEvo is failed
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvkms-fops nvKms IOCTL failed
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvidia-modeset:  Ioctl failed
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvidia-modeset:  nvkms_ioctl_common failed
Jan 01 00:00:43.546          devg_modeset.16432..2                 slog       0  ERROR:   nvRmAllocDisplays:  nvProbeHeadCount has Failed
Jan 01 00:00:43.546          devg_modeset.16432..2                 slog       0  ERROR:   pDispEvo is null.
Jan 01 00:00:43.546          devg_modeset.16432..2                 slog       0  ERROR:   nvRmDestroyDisplays: Failed with invalid pDispEvoJan 01 00:00:43.550
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR:   nvAllocDevEvo : AllocDevEvoContents is failed.
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR:   nvAllocDevEvo: nvFreeDevEvo is failed
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvkms-fops nvKms IOCTL failed
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvidia-modeset:  Ioctl failed
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvidia-modeset:  nvkms_ioctl_common failed
```

**Figure 17:** Driver Bailing on Valid Window on Invalid Head

## *Example 2 (Invalid window on valid head):*

**Description:** Map window (invalid) to head 1 (valid)

64-bit layout:

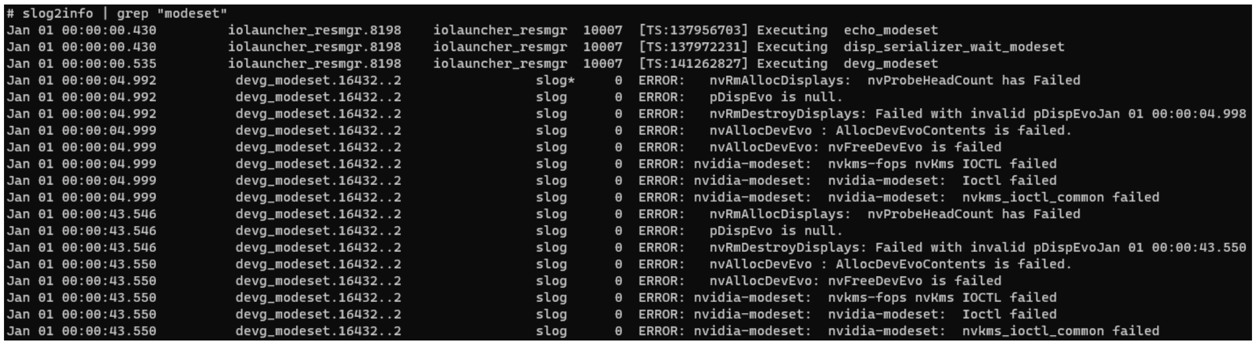
00000000 00000000 00000000 00000001 00000000 00000000 00000000 00000000

w_7 w_6 w_5 w_4 w_3 w_2 w_1 w_0

Drive Config Layout:

```
display@13800000 {
        nvidia,window-head-mask = <0x00000001 0x00000000>;
};
```

Driver Failure to Load from slog2info:



```
# slog2info | grep "modeset"
Jan 01 00:00:00.430         iolauncher_resmgr.8198    iolauncher_resmgr  10007  [TS:137956703] Executing  echo_modeset
Jan 01 00:00:00.430         iolauncher_resmgr.8198    iolauncher_resmgr  10007  [TS:137972231] Executing  disp_serializer_wait_modeset
Jan 01 00:00:00.535         iolauncher_resmgr.8198    iolauncher_resmgr  10007  [TS:141262827] Executing  devg_modeset
Jan 01 00:00:04.992          devg_modeset.16432..2                 slog*      0  ERROR:   nvRmAllocDisplays:  nvProbeHeadCount has Failed
Jan 01 00:00:04.992          devg_modeset.16432..2                 slog       0  ERROR:   pDispEvo is null.
Jan 01 00:00:04.992          devg_modeset.16432..2                 slog       0  ERROR:   nvRmDestroyDisplays: Failed with invalid pDispEvoJan 01 00:00:04.998
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR:   nvAllocDevEvo : AllocDevEvoContents is failed.
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR:   nvAllocDevEvo: nvFreeDevEvo is failed
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvkms-fops nvKms IOCTL failed
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvidia-modeset:  Ioctl failed
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvidia-modeset:  nvkms_ioctl_common failed
Jan 01 00:00:43.546          devg_modeset.16432..2                 slog       0  ERROR:   nvRmAllocDisplays:  nvProbeHeadCount has Failed
Jan 01 00:00:43.546          devg_modeset.16432..2                 slog       0  ERROR:   pDispEvo is null.
Jan 01 00:00:43.546          devg_modeset.16432..2                 slog       0  ERROR:   nvRmDestroyDisplays: Failed with invalid pDispEvoJan 01 00:00:43.550
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR:   nvAllocDevEvo : AllocDevEvoContents is failed.
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR:   nvAllocDevEvo: nvFreeDevEvo is failed
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvkms-fops nvKms IOCTL failed
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvidia-modeset:  Ioctl failed
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvidia-modeset:  nvkms_ioctl_common failed
```

**Figure 18:** Driver Bailing on Invalid Window on Head

## *Example 3 (Invalid window on Invalid head):*

**Description:** Map window (invalid) to head 1 (valid). This specific example shows using a 68-bit window to head mapping which is simply not allowed and will cause the driver to fail.

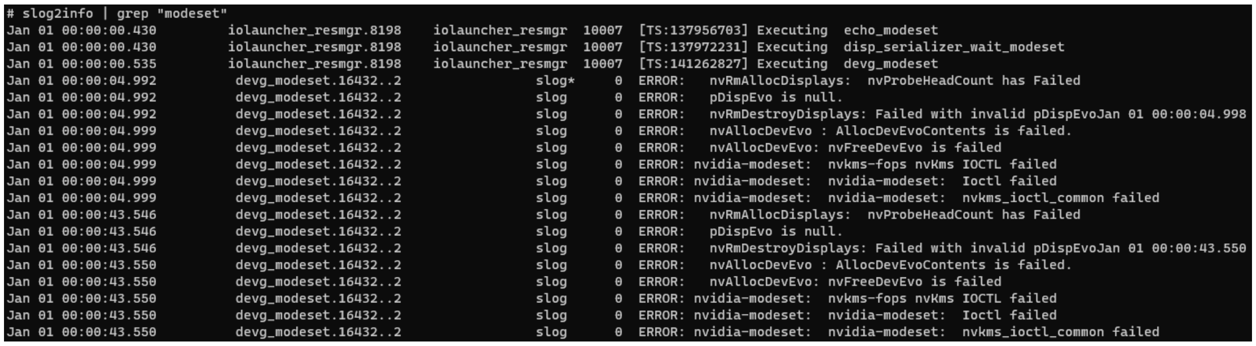
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001

w_8 w_7 w_6 w_5 w_4 w_3 w_2 w_1 w_0

Drive Config Layout:

```
display@13800000 {
        nvidia,window-head-mask = <0x00000000 0x0000000001>;
};
```

Driver Failure to Load from slog2info:



```
# slog2info | grep "modeset"
Jan 01 00:00:00.430         iolauncher_resmgr.8198    iolauncher_resmgr  10007  [TS:137956703] Executing  echo_modeset
Jan 01 00:00:00.430         iolauncher_resmgr.8198    iolauncher_resmgr  10007  [TS:137972231] Executing  disp_serializer_wait_modeset
Jan 01 00:00:00.535         iolauncher_resmgr.8198    iolauncher_resmgr  10007  [TS:141262827] Executing  devg_modeset
Jan 01 00:00:04.992          devg_modeset.16432..2                 slog*      0  ERROR:   nvRmAllocDisplays:  nvProbeHeadCount has Failed
Jan 01 00:00:04.992          devg_modeset.16432..2                 slog       0  ERROR:   pDispEvo is null.
Jan 01 00:00:04.992          devg_modeset.16432..2                 slog       0  ERROR:   nvRmDestroyDisplays: Failed with invalid pDispEvoJan 01 00:00:04.998
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR:   nvAllocDevEvo : AllocDevEvoContents is failed.
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR:   nvAllocDevEvo: nvFreeDevEvo is failed
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvkms-fops nvKms IOCTL failed
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvidia-modeset:  Ioctl failed
Jan 01 00:00:04.999          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvidia-modeset:  nvkms_ioctl_common failed
Jan 01 00:00:43.546          devg_modeset.16432..2                 slog       0  ERROR:   nvRmAllocDisplays:  nvProbeHeadCount has Failed
Jan 01 00:00:43.546          devg_modeset.16432..2                 slog       0  ERROR:   pDispEvo is null.
Jan 01 00:00:43.546          devg_modeset.16432..2                 slog       0  ERROR:   nvRmDestroyDisplays: Failed with invalid pDispEvoJan 01 00:00:43.550
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR:   nvAllocDevEvo : AllocDevEvoContents is failed.
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR:   nvAllocDevEvo: nvFreeDevEvo is failed
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvkms-fops nvKms IOCTL failed
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvidia-modeset:  Ioctl failed
Jan 01 00:00:43.550          devg_modeset.16432..2                 slog       0  ERROR: nvidia-modeset:  nvidia-modeset:  nvkms_ioctl_common failed
```

**Figure 19:** Driver Bailing on Invalid Window on Invalid Head

## *Example 4 (Valid Windows on One Valid Head):*

**Description:** Map 4 valid windows onto head 1 (valid).

00000000 00000000 00000000 00000000 00000001 00000001 00000001 00000001

w_7 w_6 w_5 w_4 w_3 w_2 w_1 w_0

Drive Config Layout:

```
display@13800000 {
        nvidia,window-head-mask = < 0x00000000 0x01010101>;
};
```

### 4.1.4 Other Tests

The following tests do not involve any IOCtl calls. The thread priority verification test is on the QNX operating system level, while the NVKMS handle deregistration test case tests the API used by client applications to close the kernel management system.

#### *Thread Priority Verification*

| | |
|---|---|
| Test File Name | tegradisp-test-statemgr.c |
| Test Case Name | nvKmsTestStateMgr (edited) |
| Negative / Positive | Positive |
| IOCTL Calls Used | N/A |
| Verification Method | Test Case Pass |
| Bugs Found? | Yes |

*Purpose*

The thread priority verification is added as part of the tegradisp-statemgr-test. The purpose of this test is to verify that devg-modeset and devg-disp-serializer threads have the correct priority during INIT_DONE and OPERATIONAL modes as shown in Table 2 and 3.

**Table 2:** devg-modeset Process Threads

| devg-modeset Thread | Name | INIT_DONE Priority | OPERATIONAL Priority |
|---|---|---|---|
| Display HW interrupt handling thread | modeset_hw_intr | 35 | 27 |
| RM Deferred work thread used for SV deferred work | modeset_rm_worker | 35 | 23 |

|  |  |  |  |
|---|---|---|---|
| Timer event thread used for FF detection | modeset_timer_thread | 35 | 30 |
| NVDVMS thread for state management | modeset_nvdvms | 35 | 23 |

**Table 3:** devg-disp-serializer process threads

| devg-disp-serializer Thread | Name | INIT_DONE Priority | OPERATIONAL Priority |
|---|---|---|---|
| NVDVMS thread for state management | serializer_nvdvms | 35 | 23 |
| GPIO ERRB irq thread | serializer_errb | 35 | 30 |

**Test pre-req step:**

1. For each thread that needs to be tested, thread name is set during its creation using pthread_setname_np() API. This also satisfies a safety requirement for process monitoring. For safety builds, there is limited permission and abilities. We are unable to set a thread name from a different process. Therefore, the thread is setting its own name inside its thread function.
2. Enable devg-serializer for simulation mode.

**NOTE:** Some board has display serializer disabled. Testing for serializer threads is skipped if that is the case.

*Concepts & Relevant Background*

There are different modes that the board is at in this test: INIT_DONE, OPERATIONAL, DEINIT_PREPARE, and DEINT.

- **INIT_DONE:** the board has successfully booted up and initialized all processes, thread priorities are set to high.

- **OPERATIONAL:** when the board is performing any tasks such as flipping a surface, thread priority is set to normal priority.
- **DEINIT_PREPARE:** the board is ready to tear down processes and shut down.
- **DEINT:** the board has shut down; all processes are killed.

To acquire thread information from the testing environment, first process id of the threads is needed. The devctl(proc_fd) system call on command "DCMD_PROC_TIDSTATUS" allows us to iterate through all threads in the process by incrementing thread id (until the devctl call fails).

**Template:**

```
void
do_process (int pid, int fd, char *name)
{
  procfs_status   status;
  printf ("PROCESS ID %d\n", pid);

// now iterate through all the threads
  status.tid = 1;
  while (1) {
    if (devctl (fd, DCMD_PROC_TIDSTATUS, &status, sizeof (status), 0)
!= EOK) {
      break;
    } else {
      do_thread (fd, status.tid, &status);
      status.tid++;
    }
  }
}
```

Using QNX system call ThreadCtlExt(process_id, thread_id) on command "_NTO_TCTL_NAME", thread name of given process id and thread id can be acquired.

*Test Step*

**Setup:**

1. Initialize nvdvms state, rrParams, serializer_enabled (false)
2. Verify state is INIT_DONE
3. Allocate device, config, surface

**Test (new steps are marked in red):**

1. Wait for INIT_DONE state. All APIs are allowed in this state.

2. Note down devg-modeset and devg-disp-serializer process id.

3. Allocate device and perform modeset and flip. This should succeed.

4. Check thread priorities for devg-modeset and devg-disp-serializer (if serializer is enabled)

5. Transition VM to OPERATIONAL state

6. Perform Flip. This should Pass.

7. Perform modeset. This should Fail.

8. Check thread priorities for devg-modeset and devg-disp-serializer (if serializer is enabled)

9. Transition to DEINIT_PREPARE

10. Perform modeset. This should Fail.

11. Deallocate device.

12. Transition to DEINIT.

### *Verification*

This test was verified on a p3663 board and p3710 board. Both a positive and a manually negative test were conducted. All thread priorities for the positive should match the table given above in INIT_DONE and OPERATIONAL mode. All thread priorities for the negative test should match the expected values that were manually adjusted. For the p3710 board, serializer process is not initialized, and its threads are not tested.

### *Debugging*

The serializer errb thread reported incorrect priority in OPERATIONAL mode during initial testing. It was discovered that the thread was being set to the priority of a lock thread. The lock thread was not created due to a check for lock pin presence (in this case, lock pin is absent and there should not be a lock thread). However, there was no check for lock pin presence to set the thread priority when mode changes to OPERATIONAL. Thus, the priority for lock thread was still set even though the thread was not created and mistakenly set to the errb thread instead. The solution was to add the same pin check for priority set.

## *NvKMS Handle Unregister Test*

| | |
|---|---|
| Test File Name | tegradisp-test-basic.c |
| Test Case Name | nvkmsTestBasic (edited) |
| Negative / Positive | Negative |
| IOCTL Calls Used | N/A |
| Verification Method | Test case pass |
| Bugs Found? | No |

### *Purpose*

The purpose of this test was to verify that the NVKMS close API fails when feeding in a negative file descriptor.

### *Test Step*

The test was added to nvkmsTestBasic (marked in red)

**Setup:**

1. initialize fd, pDevice
2. open nvkms
3. allocate pDevice

**Test:**

1. deregister -1 as NVKMS fd

**Test tear down:**

1. free device
2. close(fd)

### *Verification*

Deregister -1 as file descriptor should not return 0 (success) in the test.

## 4.2 Code Refactoring

There are certain parts within the codebase that are needed to be refactored to change the architecture within ODD. These changes were essential to the safety within the system builds of DRIVE OS.

### 4.2.1 Compile-out IOCtl's on Safety Builds

Compiling out IOCtl's is a thorough process because they play important in communicating from the user space and the kernel space. There are some IOCtl calls that we do not want to make publicly available through an API. Therefore, we not only have to properly compile them out of safety builds, but also to ensure the functions' usability within standard builds. Depending on if the IOCtl is called within tests cases, the refactoring part needs to be done methodically to preserve the ODD code architecture.

As a result of compiling out ValidateModexIndex and SetMode IOCtls from safety builds, they are now deprecated functions within the ODD codebase. The process of compiling out these IOCtl calls from safety builds is shown within Figure 20.

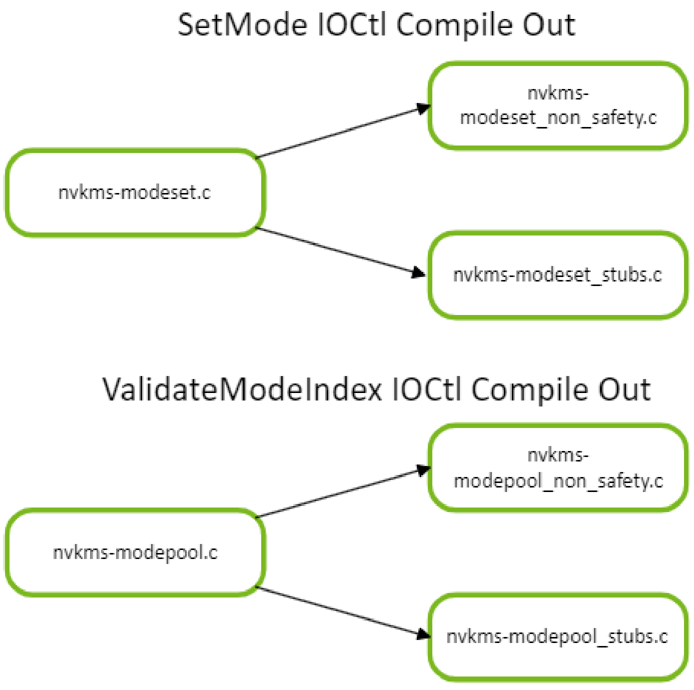

SetMode IOCtl Compile Out
nvkms-modeset.c
nvkms-modeset_non_safety.c
nvkms-modeset_stubs.c
ValidateModeIndex IOCtl Compile Out
nvkms-modepool.c
nvkms-modepool_non_safety.c
nvkms-modepool_stubs.c

**Figure 20:** Compiling Out Modes

The reason for compiling out of the SetMode IOCtl call from safety builds was because that we do not want a client facing API to be accessing this important IOCtl call. The SetMode IOCtl call changes various screen parameters such as screen size, head information, and other system critical features. Compiling out the SetMode does not remove it entirely from the safety builds; however, ODD can still access the IOCtl call to properly setup the screen and parameters on bootup.

### 4.2.2 Compile-out Logging Functions on Safety Builds

Logging functions existed within NVKMS that were unnecessary for safety builds within trees for DRIVE OS, causing unnecessary logging within the safety builds. The process of compiling out these functions within safety builds required editing the nvkms-modepool_non_safety.c file to include the logging functions within the non_safety builds of DRIVE OS. However, to make sure that the logging functions will not be used within the safety builds of DRIVE OS, we need to include stubbed out versions of the logging functions within the nvkms-modepool_stubs.c file. Depending on which build the system image (release or safety) is currently being used, the MakeFile for that system image changes because of the different safety C file that needs to be used.

### 4.2.3 Remove Client Specified LUT Set

Currently, client specified LUTs in ODD is not supported. The process of removing related functions and header files require moving certain functions or definitions to other API files as well as changing the MakeFile for both standard and safety builds. In this operation, file nvkms-color-management.c, nvkms-color-management.h and nvkms-color-management-stub.c were completely removed. The related functions inside nvkms-modeset.c were also removed.

## 5) CONCLUSION

The usage of Tegra chip in AVs is safety critical. The display driver system plays an important role in updating information in real time. Thorough and extensive test cases for the display driver, both positive and negative ones, are necessary to confirm that the system behaves as expected. Compliance with international and industry standards is also critical to prevent potential road hazards from occurring.

However, the challenge faced by the Tegra display driver team was a lack of negative testing and adherence to some safety standards within the codebase. Furthermore, some deprecated functionalities existed within safety system builds.

Our team focused on implementing tests, fixing bugs revealed by the added tests, and re-factoring to help solve these problems faced by the display driver system. The team implemented 18 integration tests, which verified the driver's behavior with various valid and invalid inputs to different display driver system functionalities. Every test that was implemented underwent an extensive verification process to confirm proper performance. Each implemented test case was also able to check off at least one safety requirement that was not met with the previous test infrastructure.

While developing ODD, our team has fixed relevant bugs within the kernel level source code. The solutions for the revealed bugs improved and streamlined the performance of the testing infrastructure that developers within the Tegra display team use. Alongside the source code, bugs were found and fixed within the CI/CD pipeline. With the work implemented, ODD now has a more robust display driver and increased reliance within DRIVE OS.

Additionally, our team completed an extensive code refactoring process for different IOCtl calls and logging functions. The refactoring process included compiling out unneeded functionalities from the safety builds, which allowed for a more streamlined safety system on DRIVE OS. This process was critical to comply with international and MISRA (Motor Industry Software Reliability Association) road safety standards. With this work, the ODD display driver now has a cleaner and more concise codebase.

## 6) FUTURE WORK

Future work for this project includes continued development of the testing infrastructure, integration of tests into test scripts, and further investigation of display driver configuration files. Each of these avenues provides an opportunity to improve the reliability of the display driver and DRIVE OS.

The IOCtl calls that were utilized throughout this project (register surface, unregister surface, and flip) only represent a subset of all IOCtls present in the display driver. Therefore, future projects can expand upon our work on IOCtl negative testing by exploring more invalid inputs on different IOCtls. Testing these untouched IOCtls can also provide ample amounts of documentation that further elucidates the innerworkings of the display driver.

While each completed test helps improve the stability of DRIVE OS, the utility of these tests can be extended by integrating them into testing scripts. Not only can testing scripts automate the process of executing all newly created tests, but they can also be placed within GVS. Within GVS, the testing scripts will be executed with every change that is pushed by a developer. These scripts will then ensure that new changes for DRIVE OS do not compromise the integrity of the display driver.

The last area of extension in our project includes the configuration files that are utilized on boot-up of the display driver. Like the head-to-window mapping configuration file, the display driver also utilizes another configuration file used for configuring the mode timings of the display. Through this file, the resolution, refresh rate, and internal clock timings can all be manipulated. The values within the mode timings configuration can be manipulated similarly to the head-to-window mapping file to better understand configurations that result in the driver crashing.

## 7) REFERENCES

[1] “QNX Operating System for Safety.” *BlackBerry QNX*, https://blackberry.qnx.com/en/products/safety-certified/qnx-os-for-safety. Accessed 4 Dec. 2023.

[2] “IOCtl(), vIOCtl().” *BlackBerry QNX*, https://www.qnx.com/developers/docs/7.1/#com.qnx.doc.neutrino.lib_ref/topic/i/IOCtl.html. Accessed 6 Dec. 2023.

[3] “How to Calculate the CRC.” *Educative*, https://www.educative.io/answers/how-to-calculate-the-crc. Accessed 6 Dec. 2023.