**KLE Society’s**

**KLE Technological University**



SCHOOL

OF

COMPUTER SCIENCE & ENGINEERING

Mini Project

on

**Log Sensor Driver API calls to display in Sensor Driver Inspector GUI**

|  |  |
| --- | --- |
| **NAME** | **USN** |
| Shrihari Hampiholi | 01FE21BCS204 |
| Shraddha Kulkarni | 01FE21BCS203 |
| Sameer Umadi | 01FE21BCS206 |
| Deepa Badawadagi | 01FE21BCS353 |

Submitted By:

Semester: V, 2023-2024

Under the Guidance of

Mrs. Vijayalakshami S

Mr. Mark Krueger

ACKNOWLEDGMENT

We would like to express our deepest gratitude to our esteemed mentors,

Mrs. Vijayalakshmi Sajjanar from KLE Tech, whose guidance, support, and expertise have been invaluable throughout this project. Their insightful feedback and encouragement have greatly contributed to the success of our endeavor. We also extend our sincere appreciation to Mr. Mark Krueger, our mentor from NVIDIA, whose expertise and mentorship have been instrumental in shaping our project and enhancing our understanding of the subject matter.

ABSTRACT

Capturing and displaying Sensor Driver API calls within the Sensor Driver Inspector GUI is pivotal for enhancing the logging framework on the Drive Orin platform, facilitating improved debugging and optimization of sensor driver interactions.

In the evolving landscape of automotive technology, the NVIDIA Drive Orin platform stands at the forefront, offering unparalleled computational capabilities tailored for autonomous vehicles. A critical component of this ecosystem is the robust interaction between software and hardware interfaces, particularly through Sensor Driver API calls. These interfaces enable real-time data acquisition and control, pivotal for the autonomous systems' decision-making processes. This project aims to enhance the existing logging framework for Sensor Driver API calls on the NVIDIA Drive Orin platform. The enhancement will focus on extending the logging capabilities to include new API calls and refining the parser to ensure the efficient display of logged information within the NVIDIA Sensor Driver Inspector GUI. By improving these aspects, the project endeavors to provide developers and engineers with a more comprehensive and intuitive tool for debugging and optimizing sensor driver interactions, thereby accelerating development cycles and improving system reliability.

|  |  |
| --- | --- |
| **Table of Contents** |  |
| **Chapters** | **Page**  **No** |
| **1. Introduction**  1.1 Objectives  1.2 Problem statement  1.3 Literature survey  1.3.1 The NVIDIA Jetson Nano  1.3.2 Standard API calls that Jetson supports  1.4 Organisation of the report | **1-8** |
| **2. System design**  2.1 Framework | **9** |
| **3. Implementation Details**  3.1 Implementation of the logger and parser | **10-14** |
| **4. Results and Discussions**  4.1 Results Analysis | **14-16** |
| **5. Conclusion & Future scope**  5.1 Conclusion  5.2 Future Scope  5.2.1 Application in the societal context | **17-18** |
| **5. References** | **19** |

**List of Figures**

|  |  |  |
| --- | --- | --- |
| 1.1 | The NVIDIA Jetson Nano | 7 |
| 2.1 | A flow-chart view of the main goals to be achieved | 9 |
| 3.1 | Contents of the files main.c and random.c, it illustrates the use of "shims", a method that enables the redirection of an API call to a one that we can designate & control | 11 |
| 3.2 | dlysm hijacking mechanism | 12 |
| 3.3 | Dynamic Linker Symbol Resolution and Redirection | 13 |
| 3.4 | Implementation of "Fake" Create Function Wrapper | 13 |
| 3.5 | A sample log report encompassing of API calls | 14 |
| 4.1 | Block diagram of the parser as a Black box | 15 |
| 4.2 | Graphical user interface (GUI) view containing a collection of recorded function calls | 15 |
| 4.3 | The three different views that help in relaying the information through different lenses to the user | 16 |

**Chapter 1**

**Introduction**

In this chapter, we explore the necessity of developing a comprehensive system for the NVIDIA Drive Orin platform, specifically aimed at enhancing logging capabilities for Sensor Driver API calls. Our project is geared towards extending the existing logging framework to accommodate 30 newer calls crucial for the platform's optimal performance. Additionally, we highlight the importance of effectively displaying the information captured in the logging process through the NVIDIA Sensor Driver Inspector GUI.

The evolution of technology in various industries, including autonomous systems, requires constant innovation and improvement of software and hardware interfaces. In this landscape, logging Sensor Driver API calls is crucial for facilitating seamless communication between components like the NVIDIA Drive Orin platform and connected devices such as camera modules. Our goal is to expand the logging capabilities to cover newer API calls, providing developers with a comprehensive understanding of sensor driver interactions. This enhancement enables more precise diagnostics and troubleshooting, contributing to the overall reliability and efficiency of the system.

Furthermore, our project also encompasses the enhancement of the existing parser to ensure the proper display of information captured during logging. The NVIDIA Sensor Driver Inspector GUI serves as a crucial interface for visualizing logged data, and by refining the parser, we strive to improve the interpretability and usability of this interface. Our objective is to empower developers and engineers with a robust toolset for analysing and optimizing sensor driver interactions, ultimately accelerating the development cycles and enhancing the reliability of autonomous vehicle systems.

Throughout the course of this chapter, we will delve into the technical intricacies involved in extending the logging framework and refining the parser. We will discuss the methodologies employed to accommodate the newer API calls and optimize the visualization of logged data within the NVIDIA Sensor Driver Inspector GUI. Additionally, we will explore the potential benefits of our project for developers and engineers working on the NVIDIA Drive Orin platform, highlighting the pivotal role of advanced logging and parsing techniques in ensuring the seamless operation of autonomous vehicle technologies.

**1.1 Objectives**

The primary objectives of this work are centered on developing a sophisticated tool designed to enhance the logging and parsing of Sensor Driver API calls on the NVIDIA Drive Orin platform. This tool aims to facilitate a deeper understanding and clearer visibility of the sensor-driver interactions, crucial for optimizing the performance and reliability of sensor operations. To accomplish these goals, the sequential tasks outlined are as follows:

* **Extend Logging Capabilities**: Enhance the current framework to include logging for an additional 30 Sensor Driver API calls. This expansion is crucial for capturing a broader spectrum of interactions and operations within the system, ensuring that newer functionalities are adequately monitored and analyzed.
* **Improve Data Parsing**: Refine the existing parser to more effectively process and interpret the information captured through logging. The goal is to ensure that the data is not only accurately captured but also presented in a manner that is easily understandable and actionable for developers.
* **Enhance GUI Display**: Integrate the enhanced logging and parsing functionalities with the NVIDIA Sensor Driver Inspector GUI. This involves ensuring that the newly logged data and the insights derived from them are effectively displayed within the GUI. The interface should allow users to navigate, interpret, and utilize the logged information efficiently, thereby facilitating a more streamlined development and troubleshooting process.

**1.2 Problem statement**

**“Log Sensor Driver API calls to display in Sensor Driver Inspector GUI”**

This project aims to meticulously log Sensor Driver API calls, ensuring that every interaction between the NVIDIA Drive Orin platform and its sensor drivers is captured and recorded. These logs will then be methodically displayed within the Sensor Driver Inspector GUI, providing a clear, accessible view of the sensor operations and interactions for debugging and optimization purposes.

**1.3 Literature survey**

**1.3.1 The NVIDIA Jetson Nano**

The Nano consists of:​

1. **GPU**: NVIDIA Maxwell architecture with 128 NVIDIA CUDA® cores​
2. **CPU**: Quad-core ARM Cortex-A57 MPCore processor​
3. **Memory**: 4 GB 64-bit LPDDR4, 1600MHz 25.6 GB/s​
4. **Storage**: 16 GB eMMC 5.1​
5. **Video Encoder**: 250MP/sec at various resolutions​
6. **Camera**: Facilitating 12 lanes (3x4 or 4x2) MIPI CSI-2 D-PHY interface 1.1 (1.5 Gb/s per pair)​
7. **Other PINs**:​
   1. **GPIO**: Set of pins that can be configured to be either inputs or outputs. Here, we can use the sensors that our diagram provided us with before.​
   2. **I2C**: Allows for Serial Communication protocol that is specifically for connecting to low-power devices, supporting long cable lengths. For example, LiDAR.​
   3. **I2S**: Allows for Serial Communication protocol that is specifically for audio data.​
   4. **SPI**: Allows for Serial Communication protocol that is commonly used to connect to peripherals such as flash memory, SD cards, and LCD displays.​
   5. **UART**: Allows for Serial Communication protocol that is commonly used to connect to terminals and other devices that require asynchronous communication.



Figure 1.1 The NVIDIA Jetson Nano

* + 1. **Standard API calls that Jetson supports**
* **Class Name**: ***gstCamera***
* **Member Functions**
* **Member Variables**

***Capture()*** : Capture the next image frame from the camera.

***CaptureRGBA()*** : Capture the next image frame from the camera and convert it to float4 RGBA format.

***Close()*** : Stop streaming the camera.

***GetType():*** Return the interface type (gstCamera::Type).

***Open()***: Begin streaming the camera.

***SetZeroCopy:*** Set whether converted RGB(A) images should use ZeroCopy buffer allocation.

***Create()***: Creates a MIPI CSI or V4L2 camera device.

***DefaultHeight***: Default camera height, unless otherwise specified during Create().

***DefaultWidth***: Default camera width, unless otherwise specified during Create().

***Type***: Specifies the type identifier of gstCamera class, MIPI CSI or V4L2.

There are several more member functions supported by this class, which can be viewed in greater detail [here](https://rawgit.com/dusty-nv/jetson-inference/master/docs/html/group__camera.html).

**1.4 Organization of the report**

Chapter 2 System design describes the functional block diagram.

Chapter 3 Implementation Details discuss about specifications and system architecture, alternative metrics we tried and input formats.

Chapter 4 Result and discussions about result analysis.

Chapter 5 Conclusions and future scope describes conclusion of this work and future scope of this problem statement.

**Chapter 2**

**System design**

A diagram of a software

Description automatically generated

Figure 2.1: A flow-chart view of the main goals to be achieved

In Figure 2.1, the architectural framework of our project, focusing on the integration of a sophisticated logging and parsing mechanism. At the core of this enhancement is the development of an "Extended Logger." This component builds upon the capabilities of the pre-existing logging system, with added functionality to meticulously record API calls. The output from the Extended Logger is a comprehensive log file that also serves as the input for the subsequent component in our framework, the "Extended Parser."

The Extended Parser, an augmentation of the existing parsing system, is specifically tailored to interpret and analyze the API calls logged by the Extended Logger. Its primary function is to distill the logged data into detailed, actionable insights. These insights are then visualized in a GUI application, offering users a granular view of the API interactions within the system. This enables users to gain a deep understanding of API interactions within the system, facilitated through an intuitive graphical interface. Our approach to logging and parsing offers a comprehensive analysis of system operations, ensuring users have access to detailed and actionable information.

**Chapter 3**

**Implementation details**

This section provides an overview of the implementation details concerning the logging of API calls and the development of the parser for the project.

**3.1 Implementation of logger and parser**

To implement logging for the Camera Module API calls on the NVIDIA Drive Orin platform, we first need to understand the specific tasks performed by these API calls in the original source code. These calls are integral to the functionality of the camera module and include:

* SetPower: This call controls the power supplied to the camera module. It manages the activation and deactivation of the camera module, allowing it to conserve power when not in use or activate for capturing images.
* SetConfig: This call configures the camera module with specific settings. It includes parameters such as the Deserializer and Serializer protocols, frame rate, and other essential configurations required for proper operation.
* Init: The Init function initializes additional properties necessary to initiate the capture pipeline. It typically takes the configuration object initialized by SetConfig and further initializes the camera module object, preparing it for data capture.
* ReadEEPROM & WriteEEPROM: These calls enable reading from and writing to the Electrically Erasable Programmable Read-Only Memory (EEPROM) of the camera module. They allow for storing and retrieving essential data or configurations, facilitating customization and maintenance of the camera module.

These API calls are encapsulated as member functions within a C++ class named CNvMCameraModule. Our task is to log these calls in a standardized format, ensuring that all relevant information is captured accurately for later analysis and debugging. In essence, the logging implementation involves intercepting these API calls and recording pertinent information such as the call name, parameters passed, and any relevant metadata. By doing so, we can create a comprehensive log of the camera module's interactions, providing valuable insights into its operation and facilitating troubleshooting and optimization efforts.

Recognizing the challenge of logging API calls without direct access to the source code, we explored various techniques to achieve this objective. Upon evaluation, we identified "shims" as a suitable method for our requirements. This approach allows us to intercept API calls and redirect them to designated functions, providing an opportunity to log crucial information such as the API's name and its parameters.

A screen shot of a computer code

Description automatically generated

A computer screen with text on it

Description automatically generated

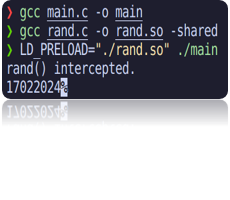
Figure 3.1: Contents of the files main.c and random.c , it illustrates the use of "*shims*", a method that enables the redirection of an API call to a one that we can designate & control.

Now, if we compile the two files as below:

> gcc main.c -o main

> gcc rand.c -o rand.so -shared

And now run the whole thing as ‘LD PRELOAD="./rand.so" ./main’ , we can see that we get ,



Using "shims" facilitates the logging process by enabling us to capture essential details both before and after the execution of the original call, all without directly modifying the source code. This capability empowers us to comprehensively monitor and analyze the program's behavior, facilitating debugging, optimization, and other diagnostic tasks. By adopting this approach, we gain valuable insights into the program's operation without the need for invasive alterations to the original codebase.

Moving forward with the previously showcased knowledge, a new obstacle emerged. The source code itself utilizes the `dlsym` function to obtain a function pointer to a function named `CNvMCameraModule\_Create`. This function, when invoked, is responsible for returning a Camera Module object known as `CNvMCameraModule`. The task at hand involves logging the 30 function calls present within this `CNvMCameraModule` object. However, intercepting the `dlsym` function itself presents a challenge, as mishandling it can result in an infinite recursion. Furthermore, redirecting the `dlsym` of the source code to our custom function led to another challenge, as it needs to hold the address of our function.

A diagram of a library

Description automatically generated

Figure 3.2: dlysm hijacking mechanism

In Figure 3.2, Essentially, the creation of our versions of `dlsym` and `CNvMCameraModule\_Create` is depicted. These custom functions are invoked when the source code attempts to call the "real" ones. Upon invocation, we ensure logging of the information, proceed to call the genuine `CNvMCameraModule\_Create`, retrieve the returned object, and then return it to the caller. This process allowed for the seamless interception and logging of the desired function calls.

A computer screen shot of text

Description automatically generated

Figure 3.3: Dynamic Linker Symbol Resolution and Redirection

In Figure 3.5, In Figure 3.5, we introduce a custom implementation of `dlsym` designed to intercept and redirect calls to the `Create` function, similar to our previous redirection of the `rand()` function. By dynamically loading a library containing our version of `dlsym`, we ensure that when the source code attempts to resolve symbols using `dlsym`, it is instead rerouted to our custom logic. This technique allows for dynamic modification and monitoring of function calls at runtime, enhancing control over application behavior without altering its source code.

A computer screen shot of a black background

Description automatically generatedA screen shot of a computer code

Description automatically generated

Figure 3.4: Implementation of "Fake" Create Function Wrapper

In Figures 3.6, we present a "fake" Create function wrapper designed to intercept calls to the original Create function. This interception enables us to call the genuine Create function internally, manipulate or wrap its return value as necessary, and then return this modified object back to the caller. This approach provides a powerful mechanism for customizing the behavior of existing functions without altering the original application code.

**3.1.1 Format of logging the calls**

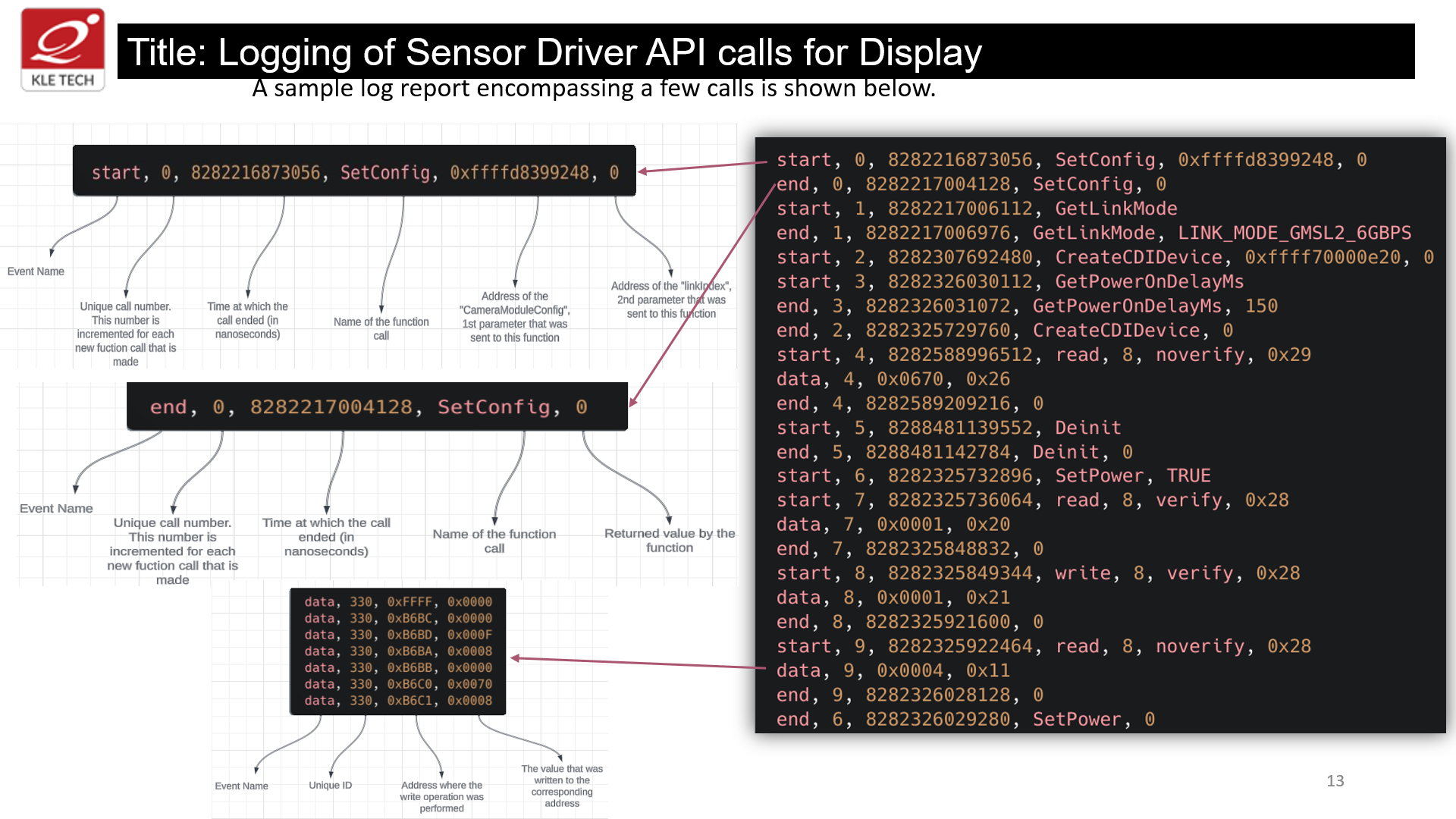


Figure 3.5: A sample log report encompassing of API calls.

Figure 3.7 displays a detailed sample log report, which meticulously documents a series of function calls. It delineates the start segment, end segment, and data segment for each call, providing a comprehensive and structured overview of the application's operational dynamics during runtime.

**Chapter 4**

**Results and discussion**

The project's results, as showcased within the NVIDIA Sensor Inspector GUI, highlight key performance metrics and system behavior captured during program execution. The graphical interface offers insights into resource utilization, temperature fluctuations, and other critical parameters monitored by the sensor driver.

The logger, meanwhile, generates textual output detailing function calls, complete with timestamps indicating when each function commenced and concluded. This textual representation provides a granular view of program execution.

Parsing of this textual data within the NVIDIA Sensor Driver Inspector GUI enables a visual depiction of the chronological sequence of function calls throughout the program's lifespan. This visual representation presents the same detailed information as the logger but in a format conducive to easy interpretation within the GUI environment, aiding in performance analysis and optimization efforts.

A diagram of a flowchart

Description automatically generated

Figure 4.1: Block diagram of the parser as a Black box

**4.1.1 Result Analysis and Validation**

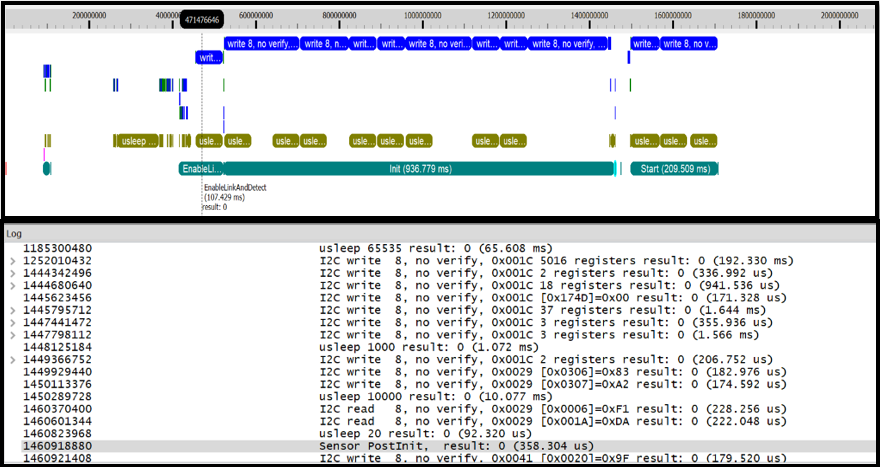
****

Figure 4.2: Graphical user interface (GUI) view containing a collection of recorded function calls

Figure 4.2 presents a graphical user interface (GUI) view containing a collection of recorded function calls. This display offers a comprehensive visual representation of the various calls made during program execution. The GUI serves as a central hub for analyzing and interpreting the recorded data, providing insights into the sequence and frequency of function invocations.

A screenshot of a computer

Description automatically generated

Figure 4.3: The three different views that help in relaying the information through different lenses to the user.

Consider an example of an I2C write call, which logs the device address, the number of registers involved, and the corresponding data-address pair. Figure 4.3 typically showcases the three different aspects of views—Partial Log Output, Hovered Output, and Detailed Log that serve distinct purposes in relaying this information to the user. The Partial Log Output offers a concise overview in the GUI, while the Hovered Output provides detailed information upon user interaction. Furthermore, the Detailed Log presents a comprehensive view of each Partial Log, facilitating in-depth analysis and understanding. Together, these views offer multiple perspectives for interpreting and analyzing recorded data.

**Chapter 5**

**Conclusions and Future Scope**

**5.1 Conclusion**

We have successfully developed a comprehensive system for logging Sensor Driver API calls and displaying them in the Sensor Driver Inspector GUI. This system enhances the visibility and analysis of API interactions within sensor drivers, leveraging an advanced logging mechanism and a sophisticated parsing tool. By integrating these components into an intuitive GUI, we have provided users with an unparalleled ability to monitor, diagnose, and optimize sensor driver operations.

The Extended Logger and Extended Parser form the backbone of this system, ensuring that every API call is meticulously recorded and analyzed. The insights generated from this analysis are then clearly visualized in the Sensor Driver Inspector GUI, offering a detailed and actionable view of the system's operations. This project not only advances the state of sensor driver diagnostics but also sets a new standard for the development and maintenance of sensor driver software.

In conclusion, our project represents a significant leap forward in sensor driver technology, providing essential tools for developers and engineers to enhance system performance and reliability. Through this initiative, we have laid the groundwork for future innovations in sensor driver analysis and debugging.

**5.2 Future scope**

**5.2.1 Application in the societal context**

In a societal context, the project holds the potential to greatly benefit various industries reliant on sensor technology, including automotive, healthcare, and environmental monitoring. By enabling precise monitoring and analysis of sensor driver API calls, our system can enhance the safety, efficiency, and sustainability of numerous applications.

To extend the project's societal impact, we could explore integrating machine learning algorithms to predict sensor behavior and preemptively address potential issues. Additionally, collaboration with regulatory bodies and standards organizations could ensure compliance with safety and environmental regulations, further enhancing public trust in sensor-driven technologies. Furthermore, fostering open-source collaboration could accelerate innovation and broaden accessibility, allowing diverse communities to leverage and contribute to the project's development.

References

* <https://man7.org/linux/man-pages/man3/dlsym.3.html>
* <https://www.ibm.com/docs/en/zos/2.2.0?topic=functions-dlsym-obtain-address-symbol-from-dlopen-object>
* <https://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html>
* https://www.bing.com/ck/a?!&&p=b25de1aec3f0a24fJmltdHM9MTcwODA0MTYwMCZpZ3VpZD0yOGFmYTcxMC02OWRmLTY3MTctMDkxNi1iNjBkNjg3MjY2MWMmaW5zaWQ9NTI2Mg&ptn=3&ver=2&hsh=3&fclid=28afa710-69df-6717-0916-b60d6872661c&psq=linux&u=a1aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvTGludXg&ntb=1
* <https://www.bing.com/ck/a?!&&p=3a7e6ce988465a9aJmltdHM9MTcwODA0MTYwMCZpZ3VpZD0yOGFmYTcxMC02OWRmLTY3MTctMDkxNi1iNjBkNjg3MjY2MWMmaW5zaWQ9NTIyOQ&ptn=3&ver=2&hsh=3&fclid=28afa710-69df-6717-0916-b60d6872661c&psq=nividia&u=a1aHR0cHM6Ly93d3cubnZpZGlhLmluL0Rvd25sb2FkL2luZGV4LmFzcHg_bGFuZz1lbi1pbg&ntb=1>
* <https://www.bing.com/ck/a?!&&p=4938c2b021b6485cJmltdHM9MTcwODA0MTYwMCZpZ3VpZD0yOGFmYTcxMC02OWRmLTY3MTctMDkxNi1iNjBkNjg3MjY2MWMmaW5zaWQ9NTI2Mw&ptn=3&ver=2&hsh=3&fclid=28afa710-69df-6717-0916-b60d6872661c&psq=arm64&u=a1aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvQVJNX2FyY2hpdGVjdHVyZV9mYW1pbHk&ntb=1>
* <https://www.bing.com/ck/a?!&&p=9465df25e8b03997JmltdHM9MTcwODA0MTYwMCZpZ3VpZD0yOGFmYTcxMC02OWRmLTY3MTctMDkxNi1iNjBkNjg3MjY2MWMmaW5zaWQ9NTIyMA&ptn=3&ver=2&hsh=3&fclid=28afa710-69df-6717-0916-b60d6872661c&psq=docker&u=a1aHR0cHM6Ly93d3cuZG9ja2VyLmNvbS8&ntb=1>
* <https://www.bing.com/aclk?ld=e8_KwpbV-9GmaTJOrzwQYzhTVUCUy4H8CF-CUp3CfypSGgeYrc7F0cjRTdqHGNiWsvBbhYQTUaewVkGstNEC-DkE9edS8rx4wimw0pfPu12sy330h__qMVjk7O4TMReKe0IGslQTCf4cIlNwM5CUQQrvxDKCojxQpGeny7dqi3ncevLJpowAYqw_wpAkrli8V3yehqsg&u=aHR0cHMlM2ElMmYlMmZ3d3cubWFrZS5jb20lMmZlbiUyZnJlZ2lzdGVyJTNmcGMlM2RpbnRlZ3JvMQ&rlid=6706dd12bc5f1d9df296b8dd228c6df6&ntb=1>
* <https://www.bing.com/ck/a?!&&p=96d7ecced08d1effJmltdHM9MTcwODA0MTYwMCZpZ3VpZD0yOGFmYTcxMC02OWRmLTY3MTctMDkxNi1iNjBkNjg3MjY2MWMmaW5zaWQ9NTIzMQ&ptn=3&ver=2&hsh=3&fclid=28afa710-69df-6717-0916-b60d6872661c&psq=cmake&u=a1aHR0cHM6Ly9jbWFrZS5vcmcv&ntb=1>
* <https://www.bing.com/ck/a?!&&p=f38340b65e4b7dffJmltdHM9MTcwODA0MTYwMCZpZ3VpZD0yOGFmYTcxMC02OWRmLTY3MTctMDkxNi1iNjBkNjg3MjY2MWMmaW5zaWQ9NTI4MQ&ptn=3&ver=2&hsh=3&fclid=28afa710-69df-6717-0916-b60d6872661c&psq=oop&u=a1aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvT2JqZWN0LW9yaWVudGVkX3Byb2dyYW1taW5n&ntb=1>