INTERNSHIP REPORT

AT

U R RAO SATELLITE CENTER

BANGALORE

Submitted By :

SANGE SOUMYA

Fourth Year Information Technology

Under the Guidance of :

Paras Gupta, Munish Agarwal, Suma Hiremath

CONTROL SYSTEM GROUP

U R RAO SATELLITE CENTER - ISRO

CERTIFICATE

यू.आर.राव उपग्रह केंद्र
अन्तरिक्ष विभाग
भारत सरकार
हवाई पत्तन मार्ग, विमानपुर डाक घर
बेंगलूर–560 017

U R RAO SATELLITE CENTRE
DEPARTMENT OF SPACE
GOVERNMENT OF INDIA
AIRPORT ROAD, VIMANAPURA POST
BANGALORE - 560 017

DATE : 30-07-2025

Internship Certificate

This is to certify that Ms. Sange Soumya

of Malla Reddy Engineering College for Women *has undergone*

Internship in Controls System Group *at this Centre*

during the period from 01/07/2025 to 30/07/2025.

COURSE : B.Tech (IT)

INTERNSHIP TITLE : GUI Interface Development for Linux Based Utilities

U.R. RAO SATELLITE CENTRE HRDD · PPEG BENGALURU

(Dr. Madhusudhana C S)
Group Director PPEG, URSC

INDEX

INTRODUCTION

PROJECT TITLE : GUI Interface Development for Linux Based Utilities

The project, undertaken during an internship at UR Rao Satellite Centre (URSC), ISRO, focused on creating a graphical interface for executing low-level utilities (refer to command-line programs (programs that run by typing text commands in a terminal or command prompt instead of clicking buttons in a graphical interface.) written mainly in ADA and executed in a Linux environment.).

Examples of low-level utilities in the project:

rrrMMU – A command-line tool that processes .map files and generates corresponding output files (like .2 files). This is used for mapping memory or configuration details needed for subsystems.

exe tools – Executables that run ADA-based algorithms for testing and validation of subsystem parameters.

Other ADA-based scripts/programs – Supporting utilities for interpreting input files, checking configurations, and generating processed results.

These utilities are essential tools used by ISRO engineers for satellite subsystem analysis and testing.

(A **satellite** is made up of many small parts called **subsystems** – like **power system** (solar panels, batteries), **communication system** (antenna, transponders), **control system** (keeps the satellite stable), **thermal system** (controls temperature), etc.

🔷 **Subsystem analysis** means **checking each part separately** to make sure it works properly. For example, testing if the solar panels give enough power, or if the antenna can send signals correctly.

🔷 **Subsystem testing** means **actually running tests (in labs or with software tools)** to see how these parts behave in real conditions, like in space (temperature, radiation, no gravity).

👉 In short: **Satellite subsystem analysis and testing is the process of studying and testing each small part of the satellite to ensure it works correctly before the whole satellite is launched.** 🚀)

By running these low-level utilities, engineers can identify the exact stage or component where an error occurs, which helps in debugging and improving the reliability of satellite subsystems.

Commands :

./exe rPAA_Angle_Out_New rmtpgmpaa.lst > rPAA_Angle_Out_New.1

rrrMMU rPAA_Angle_Out_New.1 31750fuse.map > rPAA_Angle_Out_New.2

After the user uploads the required **three input files** – typically:

- **.lst file** (program or parameter list)
- **.map file** (memory or configuration mapping)
- **.txt file** (optional configuration/reference file)

The **low-level utilities** (like exe and rrrMMU) automatically work with these inputs. The process runs a set of **predefined commands** in the backend:

1. exe takes the .lst file and generates an intermediate output (.1 file).
2. rrrMMU takes the .1 file and the .map file to produce the final processed output (.2 file).

So, instead of the engineer manually typing commands, the GUI handles everything:

- **User action:** Upload input files → Click run.
- **System action:** Invoke utilities (exe, rrrMMU) with proper syntax.
- **Output:** Final .2 file ready for download and further analysis.

They work directly with input files (such as .map, .lst, and .txt files) and generate output files that help engineers validate configurations or debug system behavior. Due to their critical role in subsystem operations, any error during execution could lead to significant delays or misinterpretation of results. To overcome this challenge, a user-friendly GUI was developed that runs these utilities in the background while allowing the operator to interact through a simplified web

interface. This reduces manual intervention, minimizes syntax-related errors, and ensures smoother workflows for ISRO engineers.

The utilities often required uploading input files and configuration maps (e.g., .map files), running command-line tools like rrrMMU, and analyzing the generated output files. The GUI provides buttons and file inputs to handle these operations transparently. Technologies like Python's subprocess module played a key role in bridging the frontend with these powerful low-level utilities.

Why this project

This project delivers a **single-page web UI** that:

- Accepts the three required inputs (31750fuse.map, rmtpgmpaa.lst, and procname.txt).
- **Parses procname.txt** (first line) to derive the base artifact name (e.g., rPAA_Angle_Out_New).
- **Automates both shell invocations** with deterministic redirection to create baseName.1 and baseName.2.
- Offers **immediate, one-click downloads** for the generated files.

LINUX COMMANDS

- ls : list files in current directory
- ls -l : list files with detailed information
- pwd : print current working directory
- mkdir dirname : create a new directory
- rmdir dirname : remove an empty directory
- rm filename : removes a file
- rm -r dirname : remove a directory and its contents directory
- touch filename : creates an empty file / update timestamp.
- vi filename : opens the file
- vi filename : opens file, i : insert, then write the code, press ESC button, enter :wq then press enter button, run the file like python3 app2.py, then you will be getting one link then paste it in the browser you can see the UI.

TECHNOLOGIES

Python (Flask Framework)

- **Why**: To create the backend server and handle requests from the web interface. Flask is lightweight and easy to integrate.
  Flask provides a **lightweight web server** to connect the GUI (HTML upload page) with backend processing (Linux utilities).
  It allows you to:
  Accept input files through a simple web form.
  Trigger backend scripts (app1.py) to process uploaded files.
  Return processed results (.2 output files) for download.
  Without Flask, users would have to run commands manually in the Linux terminal. Flask makes this process **user-friendly and accessible through a browser**.

- **Where**: Used in app1.py for the main file upload and interface logic.

  File: app2.py (Flask backend for UI and uploads).Flask is used to **create routes** (@app.route) for handling requests from the web interface.
  Example:

  - @app.route('/', methods=['GET', 'POST']): Handles file uploads from users.
  - @app.route('/download/<filename>'): Provides download functionality for generated output files.

The Flask render_template() function is used to load HTML pages (like upload.html and result.html).

Flask send_from_directory() is used to send output files back to the user.

HTML5

- **Why**: To build the web interface where users can upload files and interact with the system.
- **Where**: Used in upload.html , result.html for the file upload form and UI.

CSS

- **Why**: To style the web pages and make the interface user-friendly.
- **Where**: Applied in the templates to improve the look and feel of the GUI.

Python Subprocess Module

- **Why**: To run Linux command-line utilities (like rrrMMU, exe) directly from Python code. This bridges the GUI with low-level tools.

  The project required running **Linux command-line utilities** (rrrMMU, exe) that are ADA-based.

  Python's subprocess module provides a way to **invoke shell commands directly from Python code**.

  This acts as a bridge between the **web-based Flask GUI** and the **low-level executables** in the Linux environment.

Without subprocess, Flask/Python code cannot directly execute these compiled tools.

- **Where**: Used in app1.py when executing ADA-based utilities.

Functions like run_rrrMMU_and_save_output() and run_exe_using_bash() internally use subprocess.run() or subprocess.Popen().

Example: When you pass a procname, the code triggers rrrMMU with required input files (.map, .lst, .1) and stores the .2 output.

Similarly, the exe command is called through subprocess inside Python functions to generate further outputs.

Bash Script (invoked via Python)

- **Why**: To automate execution of sequences of Linux commands, simplify repetitive tasks, and ensure proper environment setup. Python calls these scripts for smoother execution.

- **Where**: Triggered in app1.py when handling backend operations.

  In your project, bash scripts are **triggered from app1.py** whenever the backend needs to run ADA-based Linux utilities (like rrrMMU or exe).

  These scripts contain a sequence of Linux commands that must be executed in a particular order (e.g., running rrrMMU with .1 input and .map files to generate .2 output).

Instead of writing multiple subprocess.run() calls in Python for every step, the script groups them and executes them in one go.

Linux Environment

- **Why**: The low-level utilities were originally written for Linux systems. Linux ensures compatibility, stability, and proper execution of ADA-based tools.
- **Where**: The entire backend execution of utilities runs in the Linux shell.

File Handling (Python)

- **Why**: To store uploaded input files, configuration maps, and save output files in structured folders.
- **Where**: Implemented in app2.py for handling uploads and in app2.py for saving generated results.

METHODOLOGIES

1. **Requirement Analysis**
    - Understood the needs of ISRO engineers and the pain points in using ADA utilities through Linux commands.
    - Identified the need for a GUI to simplify execution and reduce manual errors.

2. **Modular Development**
    - Divided the project into clear modules:
        - File Upload (handling .map, .lst, .txt files)
        - Backend Execution (running ADA utilities like rrrMMU through Python subprocess/bash)
        - Result Management (storing and naming outputs properly)
        - User Interface (Flask + HTML templates)

3. **Automation**
    - Automated execution of ADA utilities by wrapping Linux commands in Python's subprocess and bash scripts.
    - Reduced manual typing of commands, ensuring consistent execution.

4. **Error Handling & Validation**
    - Added checks for correct file formats before running utilities.
    - Captured command execution errors and displayed them in the web interface.

5. **User-Centered Design**
    - Developed a simple and intuitive web interface using Flask and HTML.
    - Buttons, file inputs, and download links were designed for ease of use by non-programmer engineers.

6. **Consistency & Traceability**
    - Enforced fixed input/output file locations and naming conventions.
    - Ensured every run could be traced back for reviews or investigations.

7. **Testing & Validation**
    - Tested the GUI with different sets of input files to ensure utilities executed correctly.
    - Verified generated outputs with manual command-line runs to confirm accuracy.

8. **Integration**
    - Integrated backend scripts, bash scripts, and GUI seamlessly.
    - Ensured that engineers could work end-to-end from uploading files to downloading outputs within the same interface.

OBJECTIVES

1. **Simplify Utility Execution**
   Develop a graphical interface that allows ISRO engineers to execute ADA-based Linux utilities without needing to type complex commands manually.

2. **Reduce Human Errors**
   Minimize syntax mistakes and path-related errors by automating command execution and file handling.

3. **Improve Workflow Efficiency**
   Speed up the process of uploading, executing, and analyzing outputs so engineers can focus more on result validation and system improvements.

4. **Enhance User Experience**
   Provide a user-friendly web-based interface to make interaction easier for non-expert users.

5. **Enable File Management**
   Support seamless uploading of input files (e.g., .map, .lst, .txt) and ensure that output files are properly generated and stored for further use.

6. **Integrate Automation**
   Use Python scripts and Bash automation to bridge frontend interactions with backend command-line utilities.

7. **Ensure Consistency & Repeatability**
   Maintain predictable and traceable results by enforcing structured input/output handling.

8. **Support Auditability**
   Save inputs and outputs in predefined locations to simplify verification, reviews, and non-conformance investigations.

9. **Bridge Frontend and Backend**
   Use Flask to connect the graphical interface with backend processing, ensuring smooth interaction between users and command-line tools.

10. **Promote Reusability and Scalability**
    Design the system so it can be extended to support additional utilities or adapted for other ISRO projects in the future.

MODULES

User Interface (UI) Module

- Provides a web-based GUI for engineers.
- Allows file uploads (.map, .lst, .txt, etc.).
- Simplifies interaction with backend tools using buttons and forms.

File Upload & Storage Module

- Handles uploading of input files and configuration maps.
- Stores files in a structured way for easy access during processing.

Backend Processing Module

- Uses Python's subprocess to run low-level ADA-based utilities like rrrMMU.
- Executes bash scripts for automation.
- Ensures command execution happens in the background without manual intervention.

Result Management Module

- Collects the generated output files from utilities.
- Renames and saves outputs in specific folders for traceability.
- Provides downloadable links via the GUI.

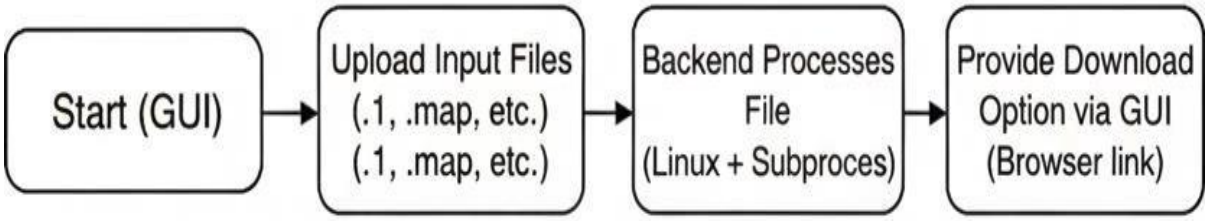
Error Handling & Validation Module

- Validates file types and inputs before execution.
- Captures errors from backend commands.
- Displays user-friendly error messages to avoid confusion.

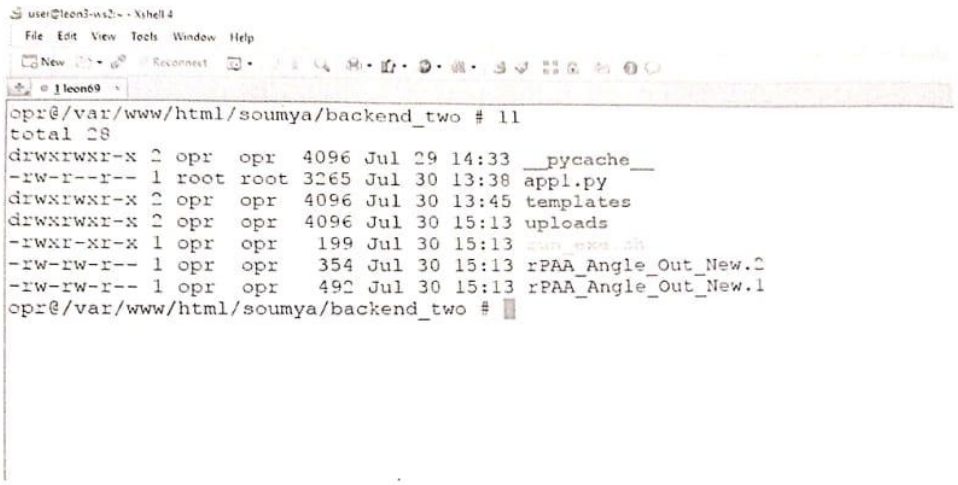
Integration Module

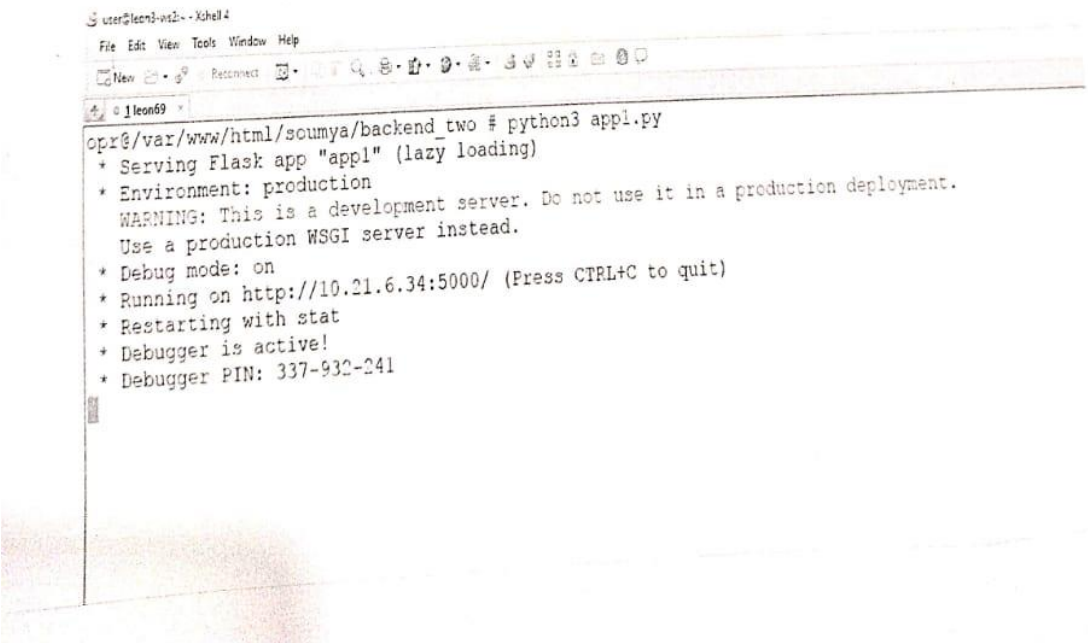
- Connects frontend (Flask UI) with backend utilities.
- Ensures smooth communication between user actions and command-line processes.
- Bridges the gap between modern GUI and low-level ADA utilities.

DATA FLOW DIAGRAM



OUTPUTS


File Edit View Tools Window Help
New Reconnect
1 leon69
opr@/var/www/html/soumya/backend_two # ll
total 28
drwxrwxr-x 2 opr opr 4096 Jul 29 14:33 __pycache__
-rw-r--r-- 1 root root 3265 Jul 30 13:38 app1.py
drwxrwxr-x 2 opr opr 4096 Jul 30 13:45 templates
drwxrwxr-x 2 opr opr 4096 Jul 30 15:13 uploads
-rwxr-xr-x 1 opr opr 199 Jul 30 15:13
-rw-rw-r-- 1 opr opr 354 Jul 30 15:13 rPAA_Angle_Out_New.2
-rw-rw-r-- 1 opr opr 492 Jul 30 15:13 rPAA_Angle_Out_New.1
opr@/var/www/html/soumya/backend_two #


File Edit View Tools Window Help
New Reconnect
1 leon69
opr@/var/www/html/soumya/backend_two # python3 app1.py
* Serving Flask app "app1" (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: on
* Running on http://10.21.6.34:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 337-932-241

Upload 3 Required Files

Choose File No file chosen

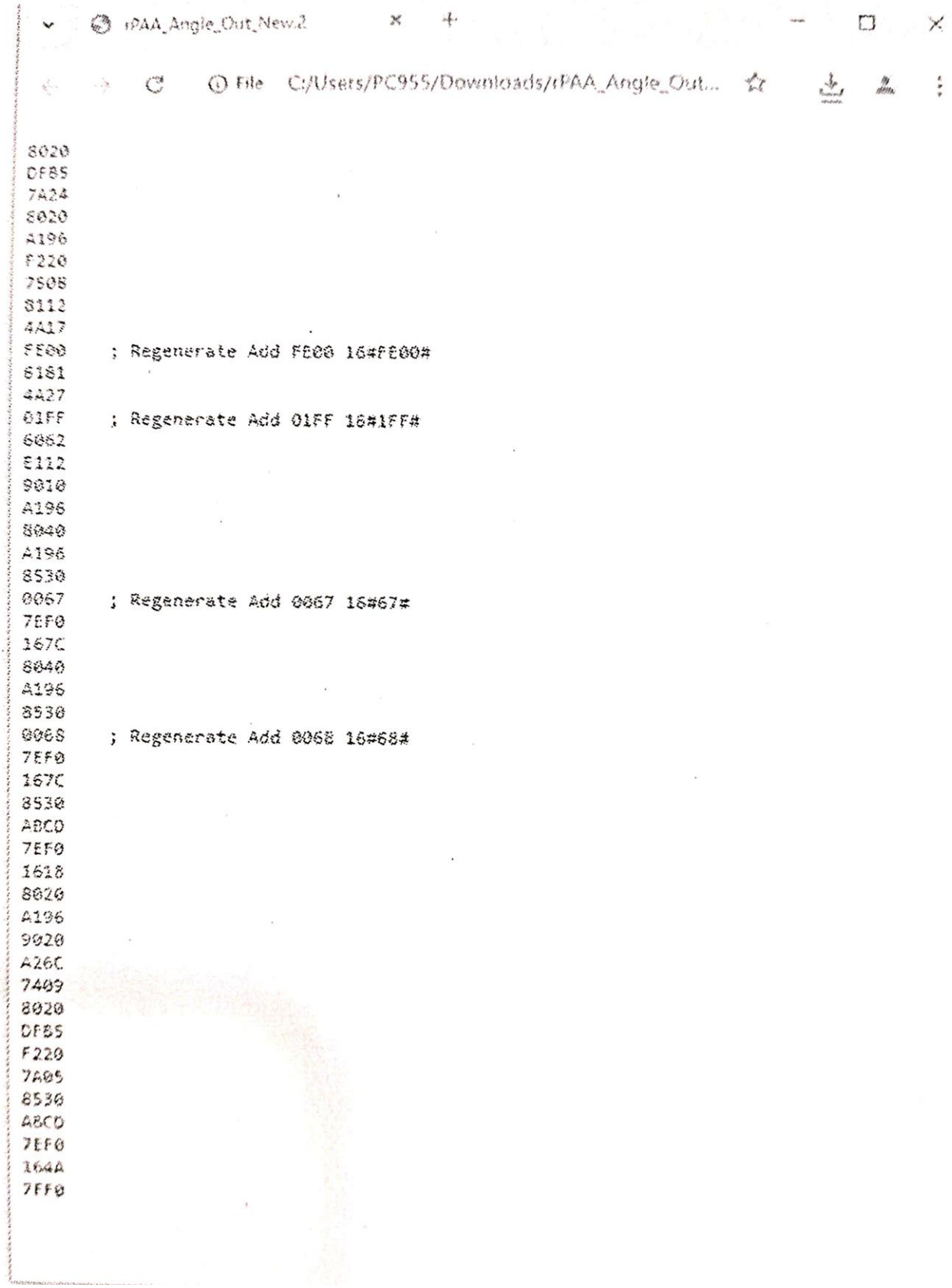
Choose File No file chosen

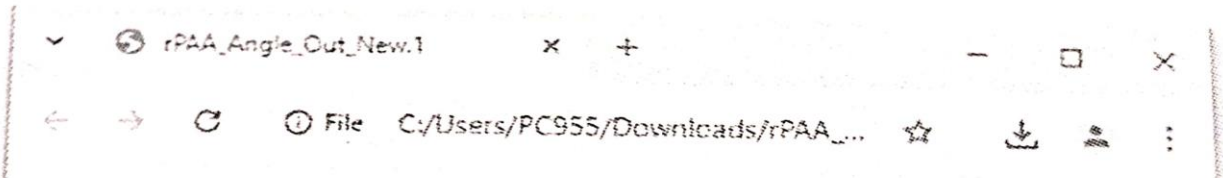
Choose File No file chosen

Files Uploaded and Processed Successfully

Download rPAA_Angle_Out_New.1

Download rPAA_Angle_Out_New.2


rPAA_Angle_Out_New.2
File C:/Users/PC955/Downloads/rPAA_Angle_Out...
8020
DFB5
7A24
8020
A196
F220
7505
8112
4A17
FE00 ; Regenerate Add FE00 16#FE00#
6181
4A27
01FF ; Regenerate Add 01FF 16#1FF#
6062
E112
9010
A196
8040
A196
8530
0067 ; Regenerate Add 0067 16#67#
7EF0
167C
8040
A196
8530
0068 ; Regenerate Add 0068 16#68#
7EF0
167C
8530
ABCD
7EF0
1618
8020
A196
9020
A26C
7409
8020
DFB5
F220
7A05
8530
ABCD
7EF0
164A
7FF0




```
8020
H MAJORCYCLECNTR_ 0000
7A24
8020
H AZIMELEV_ 0000
F220
7505
8112
4A17
I FE00 16#FE00#
6181
4A27
I 01FF 16#1FF#
6062
E112
9010
H AZIMELEV_ 0000
8040
H AZIMELEV_ 0000
8530
I 0067 16#67#
7EF0
H ROUTPUT_PORT_ 0000
8040
H AZIMELEV_ 0000
8530
I 0068 16#68#
7EF0
H ROUTPUT_PORT_ 0000
8530
H OPAADATAREADY2_ 0000
7EF0
H RSET_OP_PORT_BIT_ 0000
8020
H AZIMELEV_ 0000
9020
H TMBUFDGA_ 0000
7409
8020
H MAJORCYCLECNTR_ 0000
F220
7A05
8530
H OPAADATAREADY2_ 0000
7EF0
H RRESET_OP_PORT_BIT_ 0000
7FF0
```

CONCLUSION

The internship project at UR Rao Satellite Centre (URSC), ISRO, successfully demonstrated how complex, low-level command-line utilities used in satellite subsystem workflows can be simplified and streamlined through a user-friendly graphical interface. By developing a Flask-based GUI that integrates backend Python scripts, subprocess calls, and file handling mechanisms, the project reduced manual intervention, minimized errors, and accelerated the execution of critical utilities like *rrrMMU*.

This solution not only made the operations more accessible to engineers but also ensured consistency, reliability, and auditability of results, which are essential in space missions where accuracy and repeatability are paramount. Furthermore, the integration of bash scripts, automation workflows, and structured file storage ensured that every execution was well-documented and reproducible.

In conclusion, the project bridged the gap between low-level ADA/Linux utilities and high-level operator interactions, ultimately contributing to faster, safer, and more efficient satellite subsystem operations at ISRO.