SE 4485: Software Engineering Projects

Spring 2025

Final Report

| Group Number | 9 |
| --- | --- |
| Project Title | QNX and Python Implementation |
| Sponsoring Company | Communications and Power Industries |
| Sponsor(s) | Luke Carr |
| Students | 1. Muhammad Ali sxa220115@utdallas.edu  2. Nicholas Anderson nba220001@utdallas.edu  3. Tabark Abaid tya210000@utdallas.edu  4. Diego Ibarra dxi200004@utdallas.edu  5. Saghar Abdi sxa190148@utdallas.edu  6. Khaled Elkhaled kye210001@utdallas.edu |

QNX and Python Implementation Final Report

Executive Summary

Table of Contents

[**1. Introduction**](#) **5**

[**2. Project Management Plan**](#) **5**

[**3. Requirement Specifications**](#) **5**

[Use Case Name: Start Python Interpreter](#_tza31rq7l9ox) 6

[Participating Actors:](#_d4ccxafnpl4l) 6

[Entry Conditions:](#_i26peb93wpva) 6

[Normal Flow of Events:](#_tqo6h223ufpv) 6

[Exit Conditions:](#_4abb0zs9c0cw) 7

[Exceptions (Alternate Flow of Events):](#_cu2efrf8wn1c) 7

[Special Requirements:](#_tap5o34qztd8) 7

[Participating Actors:](#_y77zf7plxc58) 7

[Entry Conditions:](#_l4p2o1csmpvu) 7

[Normal Flow of Events:](#_77oxbxykin4l) 7

[Exit Conditions:](#_lype7daqo2c7) 7

[Exceptions (Alternate Flow of Events):](#_gndyxxb93cb) 7

[Special Requirements:](#_7n96tcf3lcq7) 8

[Participating Actors:](#_trwrzf52j710) 8

[Entry Conditions:](#_czfozqaotjv1) 8

[Normal Flow of Events:](#_omuvlcy3dc3j) 8

[Exit Conditions:](#_74l07026uk3c) 8

[Exceptions (Alternate Flow of Events):](#_kiyz4sjuzjfi) 8

[Special Requirements:](#_vk3bts3tws8z) 8

[Use Case Name: Return Output from Python to C](#_k8tpn8eh7dgt) 8

[Participating Actors:](#_l000u28u6lgt) 8

[Entry Conditions:](#_l1y38byvc1er) 9

[Normal Flow of Events:](#_9k0851h9rg3a) 9

[Exit Conditions:](#_pqo9pevbk9oh) 9

[Exceptions (Alternate Flow of Events):](#_wn7buv90wsk2) 9

[Special Requirements:](#_v6ne62uhkac) 9

[Use Case Name: Handle Python Execution Errors](#_6houbyxnlzt4) 9

[Participating Actors:](#_if7lzgbdc6kh) 9

[Entry Conditions:](#_2nfjv5j4o54i) 9

[Normal Flow of Events:](#_t2r0aonn88fr) 9

[Exit Conditions:](#_xujuimqg4yfr) 9

[Exceptions (Alternate Flow of Events):](#_r01g6tcc240u) 10

[Special Requirements:](#_7lifrw8wrc6k) 10

[Use Case Name: Close Python Interpreter](#_yww5ylqpmix5) 10

[Participating Actors:](#_g68ahn3mjhi4) 10

[Entry Conditions:](#_2qgpptwciaip) 10

[Normal Flow of Events:](#_7qhraz416kvj) 10

[Exit Conditions:](#_yi6ijbholk61) 10

[Exceptions (Alternate Flow of Events):](#_cg1ec2y0i4rq) 10

[Special Requirements:](#_vkmx221kpj2k) 10

[Use Case Implementation 2:](#_3bmz37aajdve) 11

[Participating Actors:](#_4vla6y329ltj) 11

[Entry Conditions:](#_7jh0lxht6zlw) 11

[Normal Flow of Events:](#_yaqz3zai6ima) 12

[1. Python application requests to load a shared library (.so file).](#_j3hn5r8mm5ir) 12

[Exit Conditions:](#_53z74351k04f) 12

[Exceptions (Alternate Flow of Events):](#_6y4umuv288mh) 12

[Special Requirements:](#_fh0mc15bdqap) 12

[Participating Actors:](#_3c8dekas0rhf) 12

[Entry Conditions:](#_e53attay37wt) 12

[Normal Flow of Events:](#_58p13u53yrq1) 12

[Exit Conditions:](#_15d367je4mo) 13

[Exceptions (Alternate Flow of Events):](#_8rxbjrlw68kh) 13

[Special Requirements:](#_8gy366l333hp) 13

[Participating Actors:](#_6x4tvvy9ah61) 13

[Entry Conditions:](#_mjekqydf0fnm) 13

[Normal Flow of Events:](#_3qt21beoqwt1) 13

[Exit Conditions:](#_99dvn6q88p57) 13

[Exceptions (Alternate Flow of Events):](#_yrxys9ww3cs6) 13

[Special Requirements:](#_e7vvgxct3hvj) 13

[**4. Architecture  
  
ARCHITECTURAL STYLE(S) USED**](#) **23**

[**ARCHITECTURAL MODEL**](#_t2c9l6rilunr) **23**

[**5. Design  
  
GUI (Graphical User Interface) Design**](#) **28**

[**6. Test Plan  
  
REQUIREMENTS/SPECIFICATIONS-BASED SYSTEM LEVEL TEST CASES**](#) **35**

[TECHNIQUES FOR TEST GENERATION](#_pgks5hv6i3aw) 36

[TRACEABILITY OF TEST CASES TO USE CASES](#_agt2kp7hgrik) 39

[**7. Evidence the Document Has Been Placed under Configuration Management**](#_ds5852h6yl99) **41**

[**8. Engineering Standards and Multiple Constraints**](#) **41**

[**9. Additional References**](#) **41**

[**Acknowledgment**](#) **41**

List of Figures

List of Tables

# Introduction

This document provides a final report for the “QNX and Python Implementation” project, outlining the integration of QNX with Python for system interaction. It covers the project’s scope and goals, implementation steps, and outcomes.

This project integrates QNX with Python to enable communication between Python and C functions, supporting real-time performance in embedded systems. It allows Python scripts to run from C and C functions to run from Python. A Flask-based REST API provides external control of devices and system interactions.

This document is structured as follows: the Introduction, which covers the project’s purpose, goals, and overview; the Project Management Plan, which outlines the project organization, risk analysis, and schedule; the Requirement Specifications, detailing use cases and system interactions; the Architecture section, which describes the system design; the Design section, which focuses on the system’s design components and user interface; the Test Plan, which defines system level tests; and the Configuration Management section that covers the managements and standards followed.

Terms, Acronyms, and Abbreviations:

* **API**: Application Programming Interface
* **QNX**: A real-time operating system designed for embedded systems
* **JWT:** JSON Web Token, a compact token used for securely transmitting information
* **Ctypes:** A Python library used for interacting with C functions
* **Flask:** A web framework used to build the REST API in Python
* **RTOS:** Real-Time Operating System

# Project Management Plan

# PROJECT ORGANIZATION

**Team Structure and Roles**

The development team for the **QNX and Python Implementation** project consists of the six members whose responsibilities will facilitate a structured functional workflow. Specific roles are assigned as follows:

1. **Project Manager: Muhammad Ali**

* Supervise the project with respect to its objectives.
* Manage communication with the sponsor, stakeholders, and team members.
* Facilitate meetings, track progress, and mitigate risks.
* Adhere to all deadlines and milestones of the project.

1. **Lead Developer: Nicholas Anderson**

* Oversees the implementation of Python integration in the QNX environment.
* Gives technical guidance to team members.
* Code review and ensure that software quality standards are being met.
* Assist debugging and troubleshooting.

1. **Embedded Systems Engineer: Tabark Abaid**

* Ensures that QNX is compatible with CPI hardware.
* Deals with real-time processing, optimization of system performance, and designing mechanism for hardware-software interaction.

1. **Software Developer: Diego Ibarra**

* Writes and implements the modules required for interaction between Python and QNX.
* Testing and debugging software.
* Testing that Python scripts are working properly within the QNX system.

1. **Quality Assurance & Testing Manager: Saghar Abdi**

* Creating test cases and frameworks for testing.
* Conducting performance testing and stability of the system.
* Defect tracking and ensuring fixes are completed before delivery.

1. **Configuration & Documentation Manager: Khaled Elkhaled**

* Manage version control through GitHub in conjunction with Google Docs.
* Document pre-development, system design, and technical specification.
* Compliance to IEEE and ISO standard.

**Project Organization Rationale:**

The QNX and Python Implementation project structured project organization ensures that the project will be completed smoothly and in a timely manner upholding standards of quality and maintainability.

# LIFE-CYCLE MODEL USED

· For this project, our team will be using a combination of Waterfall and Agile Lifecycle Models to ensure better results. The waterfall Lifecycle Model will be used in the beginning stages to complete tasks such as planning, requirements gathering, and design. The Agile Lifecycle Model will then be used during the project's development phase. The flexibility of this Model will allow the team to build the system in smaller sessions, test it, and proceed to the next stages.

· We chose to use a combination of Waterfall-Agile models because it allows us to have a clear and structured start. The Waterfall Model allows us to establish objectives and design the system before starting the development phase. As we move on to development, the Agile approach provides us flexibility, allowing us to make changes if we run into unexpected errors when integrating QNX and Python. Additionally, Agile provides a possibility for our team to receive continuous feedback, which will help us stay aligned with the project requirements. Combining the two models allows us to plan efficiently from the beginning and remain flexible as the project evolves, which will help us minimize risks as we move forward.

# RISK ANALYSIS

# Identified Risks

| Risk | Description | Likelihood  (L/M/H) | Impact  (L/M/H) |
| --- | --- | --- | --- |
| Software Hardware Issues | Python may not function smoothly within QNX | M | H |
| Real-Time Performance Limitations | Python may introduce performance bottlenecks, impacting QNX's real-time capabilities. | M | H |
| Resource Availability | Lack of access to required CPI hardware for testing may delay development. | H | H |
| Scope Creep | Additional feature requests from stakeholders may extend project scope beyond initial estimates. | M | M |
| Hardware Malfunctions | CPI hardware may fail during development, causing delays. | L | H |

*Table 1: Risk Analysis of Identified Risks*

# Risk Likelihood and Impact

**High Likelihood, High Impact (HH)** – Needs an immediate plan to fix.

**Medium Likelihood, High Impact (MH)** – Take steps to prevent it from happening.

**High Likelihood, Medium Impact (HM)** – Keep a close watch on it.

**Low Likelihood, High Impact (LH)** – Have a backup plan in case it happens.

**Low Likelihood, Low Impact (LL)** – Keep an eye on it, but no action is needed right away.

# Risk Mitigation Strategies

| Risk | Strategy |
| --- | --- |
| Software Hardware Issues | Conduct early testing to detect issues before full implementation. |
| Real-Time Performance Limitations | Optimize Python scripts to minimize latency |
| Resource Availability | Secure access to CPI hardware early in the project |
| Scope Creep | Establish strict project scope and deliverables. Review and approve additional feature requests only if time and resources allow. |
| Hardware Malfunctions | Have backup hardware available. Maintain plan for switching to alternative platform |

*Table 2: Risk Mitigation Strategies*

# 

# SOFTWARE AND HARDWARE RESOURCE REQUIREMENTS

**Software Requirements**

| **Software** | **Purpose** | **Lab Availability** |
| --- | --- | --- |
| Python 3.11+ | Primary language for scripting and automation | Yes |
| QNX Software Development Platform | Primary operating system | Yes |
| Git/Github | Version control | Yes |
| QNX IDE | IDE for QNX app development | Yes |
| PyQNX Library | Python binding with QNX processes | Yes |
| QNX Qnet | Network file system for communication | Yes |
| VMware | Virtualization | Yes |

*Table 3: List of Software Requirements* **Hardware Requirements**

| **Hardware** | **Purpose** | **Lab Availability** |
| --- | --- | --- |
| CPI Embedded Hardware | For QNX-Python integration devices | Yes |
| Ethernet Switch & Cables | Device testing network communication | Yes |
| Serial Debugger | Access QNX devices for debugging | Yes |
| Multimeter | Voltage and current in hardware | Yes |

*Table 4: List of Hardware Requirements and their Purpose* **Availability in Lab**

For the time being all the tools required now and later are available in the lab.

# TEAM MEMBER LEARNING

**ALL**

**​​​​**All team members gained hands-on experience in Python scripting, C programming, and cross-language integration using ctypes and the Python C API.

**Muhammad Ali**

Deepened understanding of real-time operating systems through working with QNX and adapting to embedded constraints.

**Nicholas Anderson**

Learned and enhanced understanding of Flask API development and RESTful communication practices.

**Tabark Abaid**

Gained better understanding on how to structure and write software engineering documents, such as test plans, requirements specs, and architecture diagrams.

**Diego Ibarra**

Learned how to work with QNX, a real-time operating system, including deploying applications through the QNX Momentics IDE and monitoring processes in a real-time environment

**Saghar Abdi**

Became familiar with embedded system constraints, like working within limited memory, ensuring low-latency communication

**Khaled Elkhaled**

Working with virtual machines to simulate QNX environments for testing, allowing us to understand memory management, process recovery, and performance under constrained conditions.

# DELIVERABLES AND SCHEDULE

Project Management Plan -> February 7

Requirements Documentation -> February 21

Architecture Documentation -> March 21

Detailed Design Documentation -> April 4

Test Plan -> April 18

Final Presentation and Demo -> May 3

Final Report -> May 9

These deliverables follow a waterfall method but we will be allowed to make changes to the deliverables.

# MONITORING, REPORTING, AND CONTROLLING MECHANISMS

**Reporting Structure**

1. Status Report
   * Frequency: Weekly
   * Content: Progress summary, completed tasks, schedule updates
   * Rationale: Keeps sponsor and team-members informed of any potential issues or delays with deliverables
2. Quality Assurance Report
   * Frequency: At milestone completion
   * Content: Testing results, defect tracking, possible improvements
   * Rationale: Ensures deliverables meet expectations and align with requirements
3. Risk Report
   * Frequency: At milestone completion
   * Content: New risks, risk updates, mitigation strategies
   * Rationale: Ensures proactive risk management

**Monitoring Tools and Techniques**

1. Github
   * Version Control and Code Tracking
   * Bug Monitoring
2. Google Docs

**Control Mechanisms**

1. Schedule Control
   * Gantt Chart
   * Critical Path
   * Rationale: Ensures timely completion of tasks and helps prevent project delays
2. Quality Control
   * Defect Tracking
   * Testing processes
   * Rationale: Ensures deliverables meet predefined quality standards
3. Risk Control
   * Risk assessments
   * Risk meetings
   * Rationale: Identifies, analyzes and mitigates potential risks

# 

# 

# PROFESSIONAL STANDARDS

# 

# Team Expectations

# Maintain Professional Communication

# Use respectful and clear communication in meetings, emails, and reports.

# Document discussions and decisions in meeting minutes.

# Provide constructive feedback and accept criticism professionally.

Collaborate Effectively

* Attend all scheduled meetings unless prior notice is given.
* Actively participate in discussions etc
* Assist teammates when possible and share knowledge openly.

Commit to Responsibilities

* Complete assigned tasks on time and meet quality expectations.
* Communicate about any delays or challenges.

Adhere to Technical and Ethical Standards

* Follow industry best practices and IEEE/ISO standards (detailed in Section 13).

# 

# Scholastic Honesty and Conduct

To maintain academic integrity, all team members must:

* Submit original work
* Avoid plagiarism
* Do not engage in cheating, falsification of results, or misrepresentation of contributions.
* Report any unethical behavior

First occurrence → The issue will be discussed and documented in meeting minutes.

Second occurrence → The instructor will be notified, and a formal warning will be issued.

Third occurrence → The violating member may be removed from the team and receive a prorated grade based on participation.

# Quality and Deadline Standards

**Code Quality:**

* Code must be well-structured, documented, and tested before submission.
* Use version control (Github) to track changes and maintain clean commits.

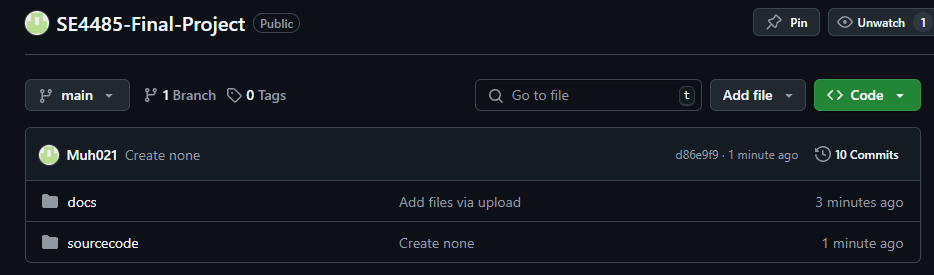
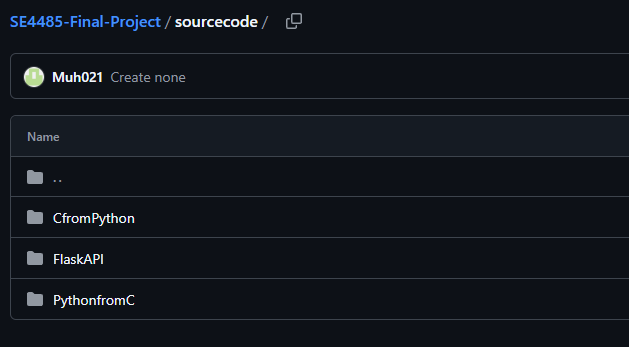
**Documentation Standards:**

* Follow a consistent format for reports, including clear explanations and references.
* All technical documents must be peer-reviewed before final submission.

**Deadlines and Milestones:**

* Each milestone must be completed at least 24 hours before the deadline for review.
* Any delays must be communicated immediately to the team lead and instructor.

# EVIDENCE ALL THE ARTIFACTS HAVE BEEN PLACED UNDER CONFIGURATION MANAGEMENT

**Impact of the Project on Individuals and Organizations**

This project, which focuses on enabling and optimizing intercommunication between Python and C within embedded QNX environments, is expected to have a meaningful and multifaceted impact on both individuals and organizations. At its core, the project addresses a critical need in embedded systems development: the integration of high-level scripting with low-level control to enhance flexibility, maintainability, and innovation in real-time, safety-critical applications.

**Impact on Individuals and Society:**

For individual engineers, researchers, and developers, this project lowers the barrier to entry in real-time embedded development by allowing them to leverage Python’s ease of use and powerful libraries for diagnostics, data processing, and automation. This accelerates learning, reduces development time, and empowers a broader range of professionals—including those from non-traditional or interdisciplinary backgrounds—to contribute to embedded systems projects.

From a societal perspective, the research supports the development of safer, smarter, and more adaptive systems across sectors that impact public health, safety, and welfare. For instance:

* In **healthcare**, embedded systems in medical devices can benefit from more efficient prototyping and faster response to software updates or bug fixes.
* In **automotive and aerospace**, where QNX is widely used for its reliability, enhanced development capabilities can lead to quicker deployment of safety-critical features.
* In **public infrastructure** (e.g., transportation, energy), better tools for monitoring and diagnostics can help prevent failures and ensure continuity of essential services.

**Impact on Organizations:**

Organizations can benefit from a hybrid development model that combines Python’s agility with C’s performance and determinism. This allows for more modular and maintainable codebases, easier testing frameworks, and the ability to integrate modern analytics or machine learning features into legacy systems. These capabilities directly support cost savings, increased productivity, and reduced risk in highly regulated industries.

**Global, Cultural, Social, Environmental, and Economic Considerations:**

* **Global and Cultural**: By leveraging open and widely adopted technologies like Python, this project encourages collaboration across international teams and cultures. It enables technology transfer and knowledge sharing between regions with varying levels of technical infrastructure.
* **Social**: The democratization of embedded development tools promotes diversity and inclusivity in the field, supporting broader participation in high-tech industries.
* **Environmental**: Efficient development and testing reduce resource consumption and e-waste by extending the life-cycle of embedded hardware and decreasing the need for physical prototypes.
* **Economic**: The project contributes to economic growth by improving development efficiency and reducing product development costs, particularly in small and medium enterprises (SMEs) that may lack the resources for large C-based engineering teams.

# Requirement Specifications

Stakeholders for the system are developers, sponsors, and project managers

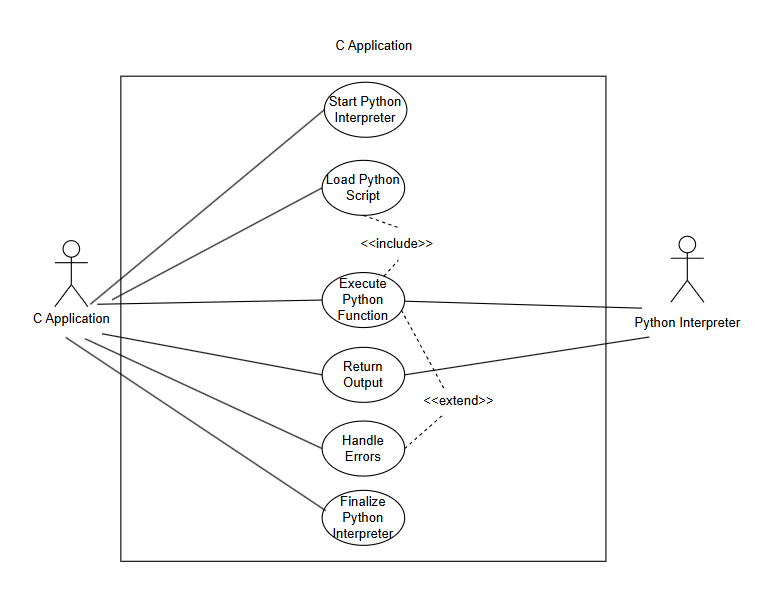
The three implementations are:

Executing Python from C

Executing C from Python

Creating a REST API using Flask

**Use Case Implementation 1:**

****

*Figure 1: Implementation 1 Use Case*

**Description:** Use Case Model for Executing Python Code from C.

#### Use Case Name: Start Python Interpreter

#### Participating Actors:

* C Application (initiates the interpreter)
* Python Interpreter (executes the script)

#### Entry Conditions:

1. The C application is running.
2. The Python environment is available on the system.

#### Normal Flow of Events:

1. The C application calls Py\_Initialize() to start the Python interpreter.
2. The Python runtime is loaded into memory.
3. The system checks if Python is properly initialized.
4. The C application proceeds to the next step (loading a script).

#### Exit Conditions:

1. Python interpreter is successfully initialized.
2. The system is ready to execute Python code.

#### Exceptions (Alternate Flow of Events):

1. Python is not installed then returns an error message and terminates.
2. Initialization fails then the system logs an error and attempts recovery.

#### Special Requirements:

1. Must work within a QNX real-time OS environment.

**Use Case Name:** Load Python Script

#### Participating Actors:

1. C Application
2. Python Interpreter

#### Entry Conditions:

1. Python interpreter is running (Py\_Initialize() was successful).
2. The script file exists and is accessible.

#### Normal Flow of Events:

1. The C application loads the Python script using PyImport\_ImportModule().
2. The system verifies that the script exists and is formatted correctly.
3. The script is prepared for execution.

#### Exit Conditions:

1. The script is successfully loaded.
2. The C application is ready to execute Python functions.

#### Exceptions (Alternate Flow of Events):

1. File not found → Return an error message.
2. Syntax error in script → Log error and notify user.

#### Special Requirements:

1. Ensure file path security to prevent unauthorized execution.

**Use Case Name:** Execute Python Function

#### Participating Actors:

1. C Application
2. Python Interpreter

#### Entry Conditions:

1. The Python interpreter is running.
2. The Python script has been successfully loaded.
3. The function to be executed exists in the script.

#### Normal Flow of Events:

1. The C application retrieves the function using PyObject\_GetAttrString().
2. The function is called with arguments using PyObject\_CallObject().
3. The Python function executes.
4. The output is prepared for return to the C application.

#### Exit Conditions:

1. The function executes successfully.
2. The result is available for the C application.

#### Exceptions (Alternate Flow of Events):

1. Function not found → Log error and return failure.
2. Execution failure → Catch and handle Python exceptions.

#### Special Requirements:

1. Ensure proper data type conversion between C and Python.

#### Use Case Name: Return Output from Python to C

#### Participating Actors:

1. C Application
2. Python Interpreter

#### Entry Conditions:

1. The function execution is complete.
2. A result (if applicable) is available.

#### Normal Flow of Events:

1. The Python function returns the result to the C application.
2. The result is processed using PyLong\_AsLong(), PyFloat\_AsDouble(), etc.
3. The C application stores or displays the output.

#### Exit Conditions:

1. The result is successfully received and processed.

#### Exceptions (Alternate Flow of Events):

1. Invalid return type → Convert result or handle error.
2. No return value → Handle gracefully.

#### Special Requirements:

1. Must support different data types (int, float, string, arrays).

#### Use Case Name: Handle Python Execution Errors

#### Participating Actors:

1. C Application

#### Entry Conditions:

1. An error occurs in any of the previous steps.

#### Normal Flow of Events:

1. The C application detects an error.
2. The error message is logged or displayed.
3. The C program decides whether to retry or exit.

#### Exit Conditions:

1. Error is handled properly without crashing the system.

#### Exceptions (Alternate Flow of Events):

1. Unrecoverable errors → Terminate execution.
2. Recoverable errors → Retry or request new input.

#### Special Requirements:

1. Must not crash the QNX system.

#### Use Case Name: Close Python Interpreter

#### Participating Actors:

1. C Application

#### Entry Conditions:

1. Execution is complete, or an error occurred.

#### Normal Flow of Events:

1. The C application calls Py\_Finalize().
2. The Python interpreter is shut down.
3. System memory is freed.

#### Exit Conditions:

1. Python is successfully closed without memory leaks.

#### Exceptions (Alternate Flow of Events):

1. Interpreter fails to close → Log error and force shutdown.

#### Special Requirements:

1. Ensure no memory leaks before closing.

#### 

#### 

#### 

#### 

#### 

#### 

#### Use Case Implementation 2:

#### 

*Figure 2: Implementation 2 Use Case*

**Description:** Use Case model for utilizing C functions and C data types in Python.

**Use Case Name:**  Call and return C function from shared library

#### Participating Actors:

* Python Application
* QNX OS

#### Entry Conditions:

1. Python App is running on QNX OS.
2. Valid C shared library is accessible to the Python Application.
3. Ctypes module is available to the application.
4. C function is properly defined.

#### Normal Flow of Events:

#### Python application requests to load a shared library (.so file).

1. System loads shared library(s) into memory.
2. The Python application calls a C function from the shared library.
3. Data is passed between Python and C, typed conversion happens as necessary.
4. C function is called and executes.
5. C function returns result.
6. Python application receives the return value.

#### Exit Conditions:

1. C function executes successfully and returns the result to Python.

#### Exceptions (Alternate Flow of Events):

1. Shared library not found -> system returns OSError.
2. Function not found -> Python throws AttributeError.

#### Special Requirements:

1. QNX OS must support shared libraries.
2. C function must be properly compiled and exported.

**Use Case Name:**  Python Application requests shared library

#### Participating Actors:

* Python Application
* QNX OS

#### Entry Conditions:

1. Python application is running on QNX OS
2. Shared library exists and is accessible
3. Interfacing library is available in the python environment

#### Normal Flow of Events:

1. Python application requests to load a shared library using ctypes.
2. System searches for specific .so files in predetermined library paths.
3. The .so file is located and loaded into memory.
4. Shared library is ready for function calls from the Python application.

#### Exit Conditions:

1. Shared library is successfully loaded into memory and linked for use.
2. Python application(s) can call functions from the shared library.

#### Exceptions (Alternate Flow of Events):

1. Shared library not found -> System throws OSError
2. Linking error -> output missing dependency(s)

#### Special Requirements:

1. Library search path must be set correctly
2. QNX OS must have .so file(s) in system directories

**Use Case Name:**  Pass data type between Python and C

#### Participating Actors:

* Python Application
* QNX OS

#### Entry Conditions:

1. The Python application is running on QNX OS.
2. C shared library is accessible for QNX.
3. The Python application has loaded the shared library using ctypes.
4. The Python application has correctly defined function argument types.

#### Normal Flow of Events:

1. Python calls a C function, passing defined arguments.
2. C function receives the value(s).
3. C function returns the result to Python.
4. Python receives returned value.

#### Exit Conditions:

1. C function properly processes passed data type(s).
2. If a return value is expected, Python correctly receives it.

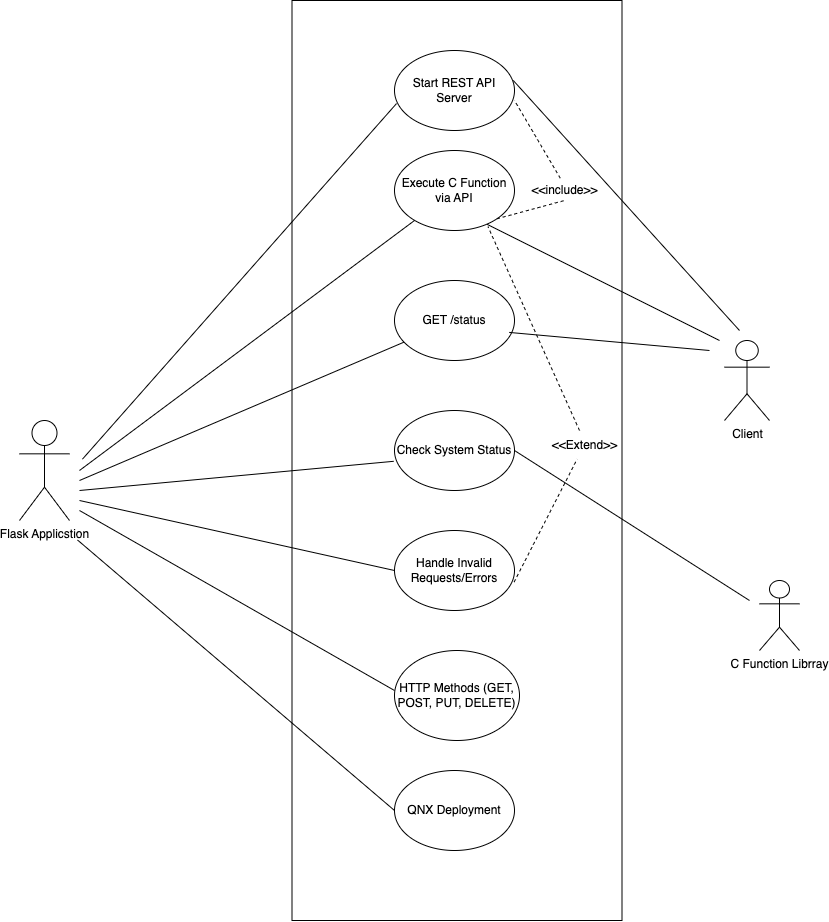
#### Exceptions (Alternate Flow of Events):

1. Data type mismatch -> possible unpredictable behavior, or receive TypeError.
2. Function not found -> receive AttributeError.
3. Segmentation Fault -> program crashes.

#### Special Requirements:

1. QNX OS must support shared libraries.
2. Shared library must be accessible.

**Use Case Implementation 3:**

****

*Figure 3: Use Case Implementation 3*

**Description:** Use Case Model for REST API Using Flask for interacting with data from CPI.

**Use Case Name:** Start REST API Server

**Participating Actors:**

1. Flask Application (initiates the API server) Client (makes requests to the API)
2. Client

**Entry Conditions:**

1. The system is running.
2. The required Python and Flask environment is available.

**Normal Flow of Events:**

1. The Flask application starts and initializes routes.
2. The system checks if Flask is properly initialized.
3. The API server begins listening for requests.

**Exit Conditions:**

1. Flask API server is successfully initialized.
2. The system is ready to handle API requests.

**Exceptions (Alternate Flow of Events):**

1. Flask is not installed → Return an error message and terminate. Initialization fails → The system logs an error and attempts recovery.

**Special Requirements:**

1. Must work within a QNX-compatible environment.
2. Ensure suitable API authentication and authorization protocols.

**Use Case Name:** Execute C Function via API

**Participating Actors:**

1. Flask Application
2. C Function Library
3. Client

**Entry Conditions:**

1. The Flask API server is running.
2. The C function library is accessible.

**Normal Flow of Events:**

1. Client sends a POST request to /execute\_c\_function with parameters.
2. Flask receives the request and extracts parameters.
3. Flask calls the corresponding C function using ctypes or a binding.
4. The C function executes and returns a result.
5. Flask formats the result into a JSON response and sends it to the client.

**Exit Conditions:**

1. The function executes successfully.
2. A structured JSON response is sent.

**Exceptions (Alternate Flow of Events):**

1. Invalid input → Return error message. C function call fails → Log error and return failure.

**Special Requirements:**

1. Ensure proper data type conversion between Python and C.

**Use Case Name:** GET Status

**Participating Actors:**

1. Flask Application
2. Client

**Entry Conditions:**

1. The Flask API server is running.

**Normal Flow of Events:**

1. Client sends a GET request to /status.
2. Flask checks system status and resource availability.
3. Flask sends a structured JSON response with the system status.

**Exit Conditions:**

1. System status is retrieved and sent to the client.

**Exceptions (Alternate Flow of Events):**

1. Status check fails → Log error and return failure.

**Special Requirements:**

1. Must return meaningful status codes.

**Use Case:** Check System Status

**Participating Actors:**

1. Flask Application
2. Cilent

**Entry Conditions:**

1. The Flask API server is running

**Normal Flow of Events:**

1. The Flask API performs internal health checks.
2. The system monitors CPU load, memory use, and active processes.
3. Flask collects system logs and diagnostic information.
4. The system status is formatted and returned to the client.

**Exit Conditions:**

1. System health information is retrieved and provided to the client

**Exceptions (Alternate Flow of Events):**

1. Failure in retrieving system status → Log error and return failure message.

**Special Requirements:**

1. Extend GET /status functionality to include deep system diagnostics.
2. Provide structured logs and metrics for debugging.

**Use Case:** Handle Invalid Requests/Errors

**Participating Actors:**

1. Flask Application
2. Client

**Entry Conditions:**

1. An invalid request is received by the API.

**Normal Flow of Events:**

1. Flask detects an invalid request
2. The API returns an error response with an appropriate HTTP status code and error message.

**Exit Conditions:**

1. The client receives an informative error response.

**Exceptions (Alternate Flow of Events):**

1. None

**Special Requirements:**

1. Must handle errors gracefully without crashing the system.
2. Should provide detailed error messages for debugging

**Use Case:** HTTP Methods (GET, POST, PUT, DELETE)

**Participating Actors:**

1. Flask Application
2. Client

**Entry Conditions:**

1. The Flask API server is running.
2. The client sends a valid request using HTTP methods.

**Normal Flow of Events:**

1. The API receives a request using one of the supported HTTP methods.
2. The API validates the request structure and authentication.
3. The API processes the request and interacts with the database or system resources.
4. A response is generated based on the HTTP method.

* GET: Retrieve data.
* POST: Create new data.
* PUT: Update existing data.
* DELETE: Remove data from the system.

1. Flask sends the response back to the client.

**Exit Conditions:**

1. The request is processed successfully, and the client receives a valid response.

**Exceptions (Alternate Flow of Events):**

1. Invalid HTTP method used → Return 405
2. Method Not Allowed. Missing or incorrect parameters → Return 400 Bad Request
3. Unauthorized access → Return 403 Forbidden.

**Special Requirements:**

1. Ensure secure API authentication and role-based access control (RBAC).
2. Support JSON request and response formats.

**Use Case:** QNX Deployment

**Participating Actors:**

1. Flask Application
2. QNX System

**Entry Conditions:**

1. The QNX system is available and running.
2. The API is prepared for deployment in the QNX environment.

**Normal Flow of Events:**

1. The Flask application is packaged for QNX deployment.
2. System checks dependencies for compatibility.
3. The application is deployed onto the QNX embedded system.
4. The Flask API is tested for stability within the QNX environment.
5. The system ensures the API is properly communicating with QNX components.

**Exit Conditions:**

1. The API is successfully deployed and running on QNX.
2. The system is ready to handle live requests.

**Exceptions (Alternate Flow of Events):**

1. Dependency issues → Log error and halt deployment.
2. Deployment failure → Rollback to the previous stable version.

**Special Requirements**:

1. Must be optimized for real-time QNX embedded environments.
2. Ensure resource constraints are met (low memory and CPU usage).

**Rationale:** We broke down the process into separate use cases to make each step clear and easy to follow. This helps with troubleshooting since errors can happen at different stages like executing a C function or checking system status. Lastly, this approach ensures better documentation and understanding for anyone working on or reviewing the project.

**NON-FUNCTIONAL REQUIREMENTS**

**7.1 NFR1:** System Availability  
Description: The system should have an uptime of 99.9%, minimizing interruptions to service. Critical functionalities must have redundancy (backups) to prevent single points of failure. The system should support automatic recovery from minor faults without requiring a full restart. Python scripts running on QNX must not cause system crashes or memory leaks. QNX should detect and restart any failed Python processes within 5 seconds.

Redundancy**:** Critical components like servers, processes, and scripts must have backups to take over in case of failure. If a Python script crashes, QNX should automatically restart it or switch to a backup.

Failover Mechanisms: The system should have automatic failover to ensure continuous operation, such as switching to a replica database if the primary server fails.

Monitoring: QNX should continuously track Python processes and system health to detect and resolve failures quickly.

**7.2 NFR2:** Performance  
Description: The system must meet performance benchmarks to run efficiently. Critical operations should respond within 500ms under normal conditions. It should also support at least 100 concurrent users or processes without any drop in performance.

Load Testing: The system's performance should be tested under heavy loads to ensure it can handle peak usage.

Resource Optimization: Python scripts and QNX processes should be optimized to use minimal CPU and memory.

Scalability: The system should support horizontal scaling by adding more resources, such as extra servers or processes, to handle increased demand.

**7.3 NFR3:** System Usability Description: The Python interface should provide clear and structured logging, including timestamps, error levels, and detailed diagnostics.The system should support command-lineutilities for executing and monitoring Python-QNX interactions. Error messages should be human-readable, with actionable recommendations when failures occur.

Logging: Logs should capture detailed information, including error stack traces and performance metrics, to help diagnose issues.

User Feedback: Error messages should clearly explain issues and suggest solutions, such as indicating "Memory limit exceeded" with guidance to optimize or increase memory allocation.

Command-Line Tools: Utilities should be available for starting, stopping, and monitoring Python-QNX interactions to simplify system management.

**7.4 NFR4:** Maintainability & Scalability Description: The system must support modular components, allowing new Python scripts or QNX functionalities to be added with minimal modifications.Python-QNX integration should be documented with API specifications, ensuring easy maintenance and upgrades.All configuration parameters should be stored in a single, easily editable configuration file, reducing human error.The system should allow remote debugging and updates to avoid unnecessary physical access to CPI’s hardware.

Modularity: Components should be independent, allowing updates or replacements without affecting the whole system

Documentation: Detailed documentation should include API specs, configuration guides, and troubleshooting instructions.

**7.5 NFR5:** Fault Tolerance Description:The system must be reliable and capable of handling faults gracefully to ensure continuous operation. The system should automatically detect failures and switch to backup systems when necessary. Error handling mechanisms must provide meaningful feedback to users and attempt automatic recovery when possible. Regular backups of critical data should be maintained to prevent data loss. The system should be resilient to hardware malfunctions, ensuring continued operation even if one component fails. The system should perform error detection and correction in data transmission to prevent corruption.

Graceful Degradation: The system should keep running with limited functionality instead of crashing. For example, if a sensor fails, it should still process data from other sensors.

Automatic Recovery: The system should recover from failures without manual input, such as restarting a failed Python script or switching to a backup.

Data Integrity**:** Use checksums or error-correcting codes to ensure data remains accurate during transmission.

**7.6 NFR6:** Security Requirements Description: All communication between Python and QNX components must be encrypted using industry-standard encryption protocols. The system should support role-based access control (RBAC) to prevent unauthorized modifications to QNX settings. Python scripts should run in a restricted execution environment to prevent unauthorized system access. Sensitive data should be securely stored and access-controlled.

Graceful Degradation: The system should keep running with limited functionality instead of crashing. For example, if a sensor fails, it should still process data from other sensors.

Automatic Recovery: The system should recover from failures without manual input, such as restarting a failed Python script or switching to a backup.

Data Integrity: Use checksums or error-correcting codes to ensure data remains accurate during transmission.

**7.7 NFR7:** Hardware Requirements Description: The system must be optimized for CPI’s embedded hardware, ensuring compatibility with ARM and x86-based QNX devices. It should function correctly with existing CPI communication protocols. The Python runtime should not exceed 100MB of memory usage to maintain optimal system performance and prevent limited system resources.

Hardware Compatibility: Test and validate the system on all supported hardware platforms, such as ARM and x86.

Resource Constraints: Optimize Python scripts and QNX processes to use minimal resources for efficient performance on embedded hardware.

Communication Protocols: Ensure smooth integration with CPI’s existing protocols to prevent compatibility issues.

**7.8 NFR8:** Portability Description: The system should work across different hardware platforms and operating systems supported by QNX and Python, avoiding platform-specific dependencies when possible.

Architecture Compatibility: Ensure Python scripts and QNX configurations support both ARM and x86 architectures.

Cross-Platform Development: Use cross-platform libraries and tools to reduce platform-specific code.

Documentation: Provide clear guidelines for porting the system to new environments.

**7.9 NFR9:** Interoperability Description: The system should integrate smoothly with third-party tools, libraries, and protocols used in CPI’s ecosystem.

Protocol Compatibility: Ensure support for industry-standard protocols like MQTT, HTTP, and TCP/IP.

Data Exchange: Support common formats such as JSON, XML, and CSV for seamless data sharing.

Integration Support: Provide APIs or interfaces for connecting with external systems.

**7.10 NFR10:** Testability Description: The system should support testing at all levels, including unit, integration, and system testing.

Automation Support: Include hooks for automated testing and debugging.

Logging & Diagnostics: Provide logging and diagnostic tools to simplify testing and troubleshooting.

Mock Interfaces: Use mock interfaces or simulators to test Python-QNX interactions separately.

Modular Design: Ensure components are independent and can be tested individually.

# Architecture ARCHITECTURAL STYLE(S) USED

In this project, we followed a combination of Client-Server and Layered Architecture styles. These styles helped us organize the system into manageable parts and made it easier to connect low-level system operations with high-level application logic.

Layered Architecture:

For our first and second implementation of just communication between C and Python, we built using a layered architecture. This design separates the system into different levels, each with its own responsibilities, making the system easier to manage, test, and update.

Hardware and QNX Layer (C code):

This is the bottom layer where core system tasks are handled. It runs on the QNX real-time operating system and takes care of low-level operations like data collection or processing.

Integration Layer (C ↔ Python Communication):

This layer connects the C and Python components. In one direction, it lets the C code trigger Python scripts for logging or reporting. In the other direction, it allows Python to access C functions for better performance. This two-way communication helps both parts of the system work together smoothly.

Application Layer (Python):

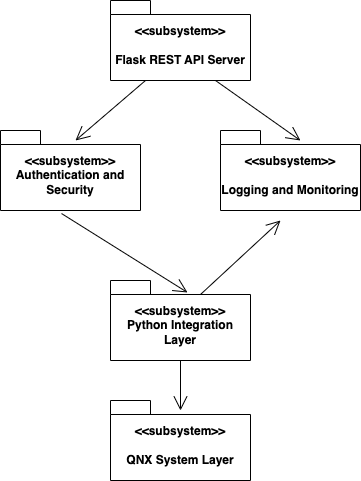
This is the high-level layer where Python handles tasks like generating reports, formatting logs, or running data-related operations. Separating this from the C code keeps the system flexible and easier to modify when needed.

Client-Server Architecture:

Our project includes a REST API built with Flask, which follows the client-server model. In this design, the API acts as the server, handling requests from clients, which can be other systems, tools, or users.The server receives HTTP requests, processes them (sometimes with help from the Python or C layers), and sends responses back to the client. This approach makes it easy to integrate our system with other tools or services, and provides a clean, scalable way to interact with the application from outside.

# ARCHITECTURAL MODEL

* use packages stereotyped as subsystems <<subsystem>>
* no classes in the architectural model



*Figure 4: Architectural Model of QNX Flask API*

* Flask REST API System
* Sends processing requests to the integration layer
* Interacts with the authentication layer to validate JWT tokens
* Sends a log of user activity or system errors to the logging subsystem
* Python Integration Layer
* Translates API input into system calls
* Communicates with QNX system layer
* Sends a log of system responses to the logging subsystem
* Authentication and Security
* Controls the system access and token validation
* Logging and Monitoring
* Receives logging data from REST API Server and Python Integration Layer

**Technology**

Programming Languages: C, Python.

Frameworks: Flask (REST API)

Communication: C Python API

Operating System: QNX RTOS

Version Control: Git/Github

Database: SQLite

**Software**

QNX Development Tools: QNX Software Development Platform (Includes QNX Compiler, Debugger, and Build System), Momentics IDE (For QNX-based C programming), GDB (For debugging C programs in QNX). QNX File System Tools (For interacting with QNX RTOS at the file level).

Flask API Development & Security: Flask-RESTful is used in designing REST API endpoints. JWT-Extended will used for handling JWT-based authentication Handling cross-origin requests is taken care of by Flask-CORS

Python Development Tools: The main scripting language is Python 3.10+ . Pip & venv (Python Package Manager & Virtual Environments) are used for dependency management. Automated testing of Python components will be handled by Pytest & unittest.

**Hardware**

Embedded System Hardware: CPI Power Supply Controllers (For handling hardware power and state transitions), CPI RF Transmitters & Communication Hardware (For transmitting industrial control signals), QNX-Compatible Embedded Board (Running QNX RTOS).

Processor & Memory Considerations: ARM Cortex-A Series / x86-based Industrial CPUs for low-power, high-efficiency for real-time processing. At least 512MB RAM is required. The recommended 1GB + is used to support Python processes and API requests. Flash Storage (eMMC/SSD) will track logging system interactions and storing configuration files.

Security & Compliance Hardware (if needed): Trusted Platform Module (TPM) is for securing boot and cryptographic key storage. Hardware Security Module (HSM) will handle encryption operations securely.

**Application and Database Server Communication**

Data Flow:

1. Client Sends API Request. A client application (external user, monitoring system, or another service) makes an HTTP request to the Flask REST API.
2. Flask API Processes the Request. The Flask API receives the request and determines whether it needs to fetch data from the database.
3. Database Query Execution. If data is needed, the Flask API constructs a SQL query and sends it to the database server. The database retrieves the requested data or modifies records The database processes the query and returns the results.
4. Flask API Formats the Response. The API receives the query results, converts them into JSON format, and sends the response back to the client.
5. Client Receives and Displays Data. The client application parses the JSON response and presents it in a user-friendly format.

Communication Protocols:

REST API over HTTP(S) - The Flask API communicates with clients using HTTP/HTTPS, ensuring secure data transmission. It uses TLS 1.2+ encryption to prevent eavesdropping.

Database Querying via SQL - The Flask API communicates with the database using SQL queries.

Authentication with JWT - Each API request includes a JWT token to ensure only authorized users can access the database.

The API server maintains persistent database connections to handle multiple requests efficiently.

Performance Communication Considerations: The database uses indexes on frequently searched columns to speed up queries. Frequently requested data is cached to reduce database load. If the database receives high traffic, it can be distributed across multiple servers. Flask API can be scaled to handle increased client requests.

RATIONALE FOR YOUR ARCHITECTURAL STYLE AND MODEL

This project follows a combination of Layered and Client-Server Architecture.

The rationale behind this choice is to ensure modularity, scalability, fault tolerance, and real-time performance while maintaining ease of development and future expansion.

**Client-Server Rationale**

Remote Accessibility & Interoperability: The system is designed to allow external clients to interact via REST API endpoints. This makes it possible to integrate with external tools, third-party systems, and other applications.

Standardized Communication (RESTful API):Using Flask-based REST APIs ensures that different applications can interact with the system using standard HTTP requests (GET, POST, PUT, DELETE). This makes the system platform-independent and ensures long-term usability.

Security & Controlled Access: The API ensures that only authenticated users can trigger C functions or modify settings. This prevents unauthorized modifications to the QNX system and to hardware.

Scalability: If more clients need access, the API server can scale by deploying additional instances. This makes it possible to expand to a distributed system without modifying the core QNX-Python integration.

**Layered Architecture Rationale**

Encapsulation and Modularity:The system is divided into separate layers (hardware interaction, Python integration, and API) to allow independent development and maintenance. This prevents changes in one layer from affecting other layers.

Security & Access Control: Direct access to QNX is restricted to the lower system layers. The Flask API acts as a controlled interface, enforcing authentication and data validation. This reduces the risk of malicious or accidental system modifications.

Maintainability: If new hardware is added, only the QNX Layer needs modification. If new API features are added, only the Flask Layer is affected. This modular design allows future expansion with minimal impact on existing functionality.

Separation of Concerns: Each layer will have a specific role. The QNX Layer (C code) will handle real-time execution and direct hardware control. Python Integration Layer will bridge Python and C using C API and ctypes. The Flask API Layer exposes REST API endpoints for external communication. This ensures clear separation of functionality, making debugging and testing easier.

**Constraint Handling Rationale**

Low Latency: C is used for real-time hardware communication because Python is not optimized for real-time execution. The Python layer only performs non-time-sensitive computations, ensuring that real-time constraints are met.

Process: Python scripts run in isolated processes to prevent memory leaks or failures from affecting the QNX core system. QNX can restart failed Python processes without disrupting real-time operations**.**

**Security & Fault Tolerance Rationale**

Restrict Direct Access to QNX: The QNX layer does not expose direct shell access. All modifications will only happen through controlled API requests.

Recovery: If a Python function fails, the system logs the error and retries execution. If the API crashes, QNX continues operating, ensuring that hardware remains functional.

Data Encryption: All API interactions use HTTPS with TLS 1.2+ encryption. User authentication will be enforced via JWT-based token authentication to verify users or systems accessing the API. All interactions are logged for security audits.

TRACEABILITY FROM REQUIREMENTS TO ARCHITECTURE

* provide a mapping between requirements and architecture
* clearly describe how each requirement in the *Requirements Documentation* is captured in the architecture

To ensure that the system architecture aligns with the project requirements, a traceability matrix is used to map each requirement to specific architectural components. This approach helps verify that all functional and non-functional requirements are addressed within the system design.

| **Requirement ID** | **Description** | **Mapped Component** |
| --- | --- | --- |
| RQ-001 | Secure authentication | JWT Authentication Service |
| RQ-002 | Real-time data processing | QNX Real-time Processing Module |
| RQ-003 | Python-C interoperability | Python-C Bridge |
| RQ-004 | System monitoring | Logging & Monitoring Service |
| RQ-005 | REST API for communication | Flask API Layer |
| RQ-006 | Hardware compatibility | Embedded System Hardware |
| RQ-007 | Scalability and Modularity | Layered Architecture |
| RQ-008 | Error Handling and Recovery | Fault-Tolerance Mechanisms |
| RQ-009 | Data Encryption | Security & Compliance Hardware |
| RQ-010 | Performance Optimization | Database Indexing & Caching |

This traceability ensures that all system requirements are effectively implemented and tested within the architectural design.

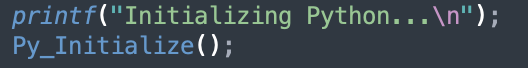
# Design GUI (Graphical User Interface) Design

* Implementation 1:
* **Step 1: Starting the C program**

The C program initiates execution and prepares the system to interface with Python. It creates the required conditions to use Python functions from inside a C program.

* **Step 2: Initializing Python**

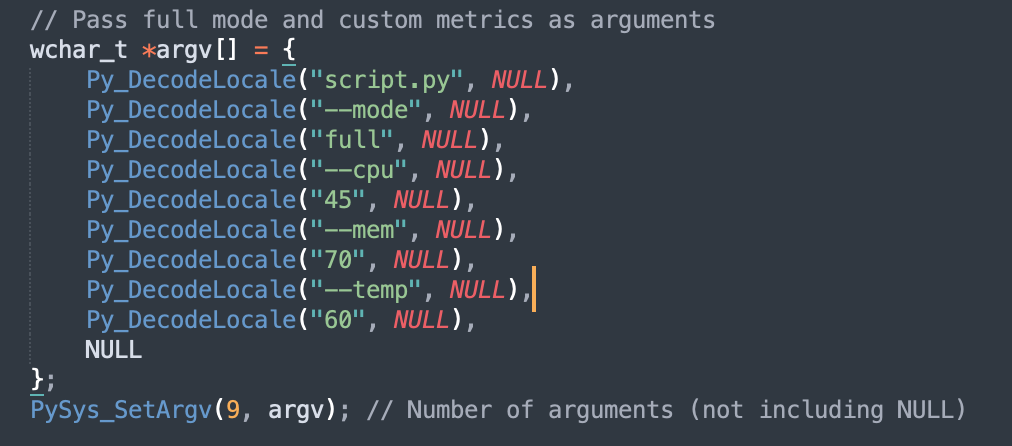
The Python interpreter has to be initialized within the C program before any Python code can be run on the system. The Py\_Initialize() function performs the initiation of the Python interpreter and prepares the environment for the code to be run.



*Figure 5: Python initializing in C*

* **Step 3: Passing Arguments**

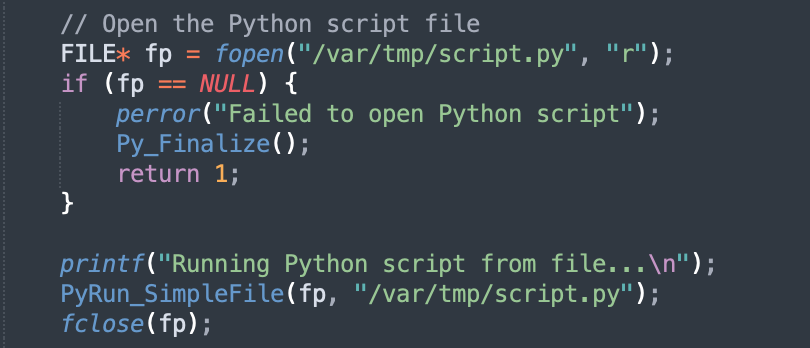
The C program then will pass command-line arguments to the Python script. These arguments determine how the program would behave. The PySys\_SetArgv() function mimics the process of sending command-line arguments to Python just as it would in a standard Python environment.



*Figure 6: Passing Args to script*

* **Step 4: Initiating the execution of the Python script**

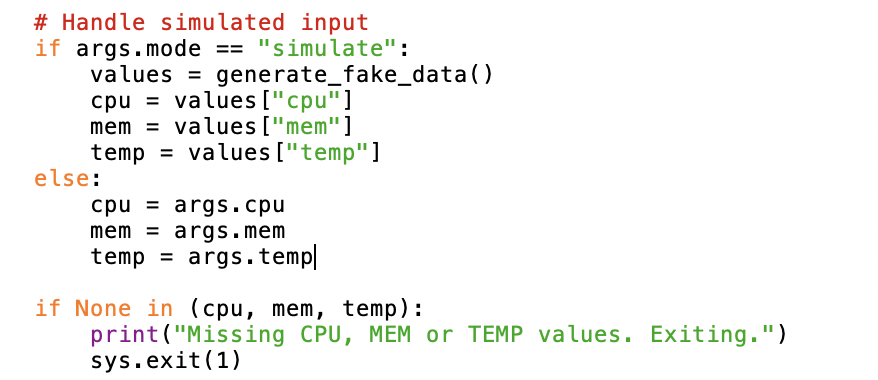
Using PyRun\_SimpleFile(), the C program launches the Python script (script.py) and calls Python to run it. This function uses the initialized Python interpreter to read and run the script line by line.



*Figure 7: Execution of Python Script in C*

* **Step 5: Executing the Python script**

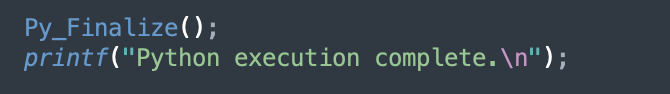
The Python script will process the arguments passed from the C program, such as: --mode, --cpu, --mem, --temp, etc... and use them to perform some action. The script will access these arguments through thePySys\_SetArgv function, which holds the command-line arguments.



*Figure 8: Example Python Script*

* **Step 6: Finalizing Python**

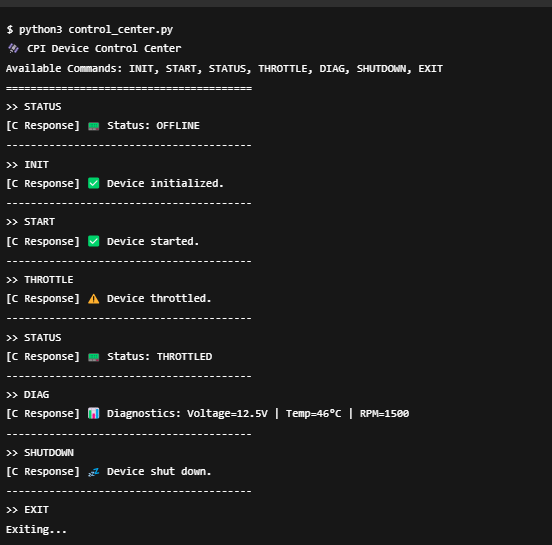
To properly terminate the Python interpreter after the Python script has completed running, the C program invokes Py\_Finalize(). This ensures the prevention of memory leaks or other issues that could arise if the interpreter is left running after the script has finished.

****

*Figure 9: Python finalizing example*

* Implementation 2:
* **Step 1: Launch the CLI**

The user executes (control\_center.py), launching a menu driven CLI that listens for typed commands like INIT, START, or STATUS. This interface is designed for real-time device control, particularly on embedded systems running QNX.



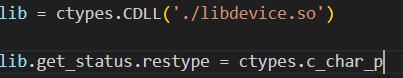
*Figure 10: Command Line Interface for Second Implementation*

* **Step 2: Load the Library**

Using ctypes.CDLL(), Python loads the precompiled C shared object file (libdevice.so), exposing all the core C functions required to control the hardware device.

* **Step 3: Define Function Signatures**

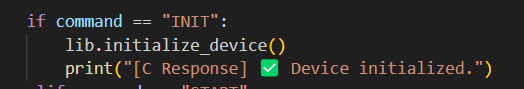
Before invoking any C function, Python sets up argument and return types to ensure smooth and type-safe data exchange between the two languages.



*Figure 11: Loading ctypes Library*

* **Step 4: Handle User Commands**

Each typed command is mapped to a corresponding C function. For instance, the INIT command triggers initialize\_device(), and the output is displayed back to the user with helpful icons and feedback labels.



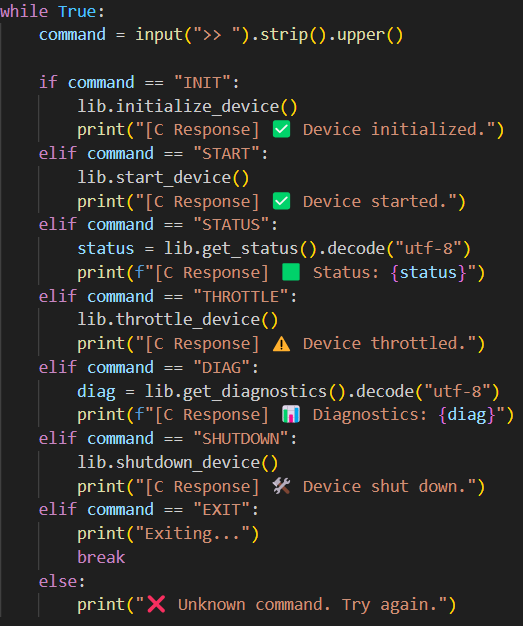
*Figure 12: Ctypes call to shared library*

* **Step 5: Display System Feedback**

The output from the C functions is shown to the user in a clear, nicely formatted way with helpful icons, making it easier for engineers and testers to use.

* **Step 6: Exit**

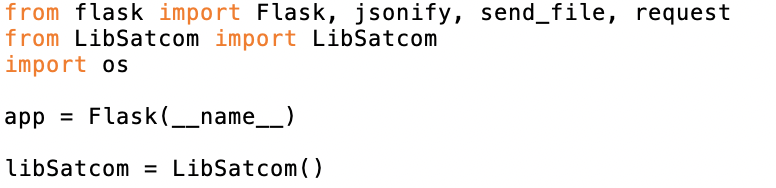
The program remains in a loop, continuously accepting and processing commands until the user enters EXIT, at which point it shuts down gracefully.



*Figure 13: Shared Library command to be called*

* Implementation 3:
* **Step 1: Setup and Initialization**

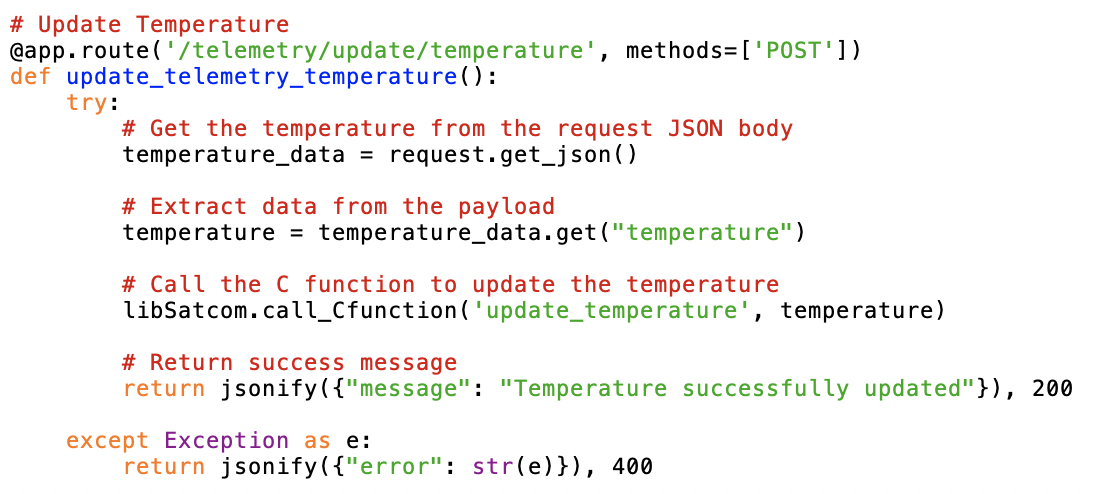
The api.py file uses Flask to set up the REST API. To establish the communication link with the LibSatcom class, we load the required libraries, and initialize the Flask application.



*Figure 14: Setup for Flask Application to be run on QNX*

* **Step 2: Defining API Endpoints**

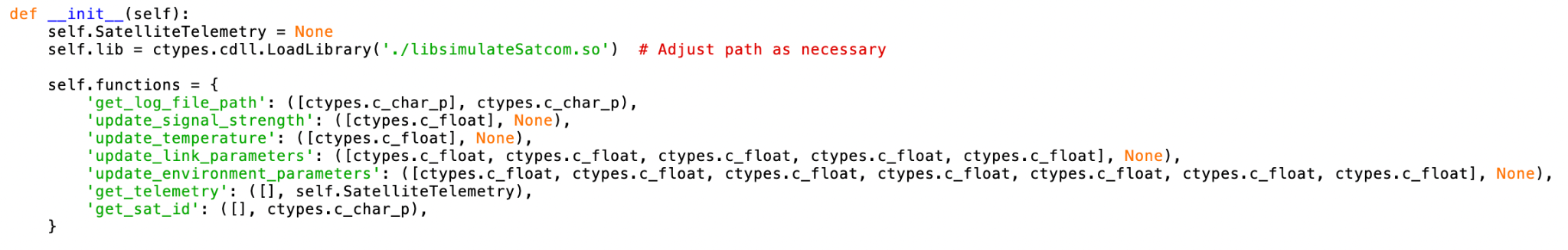
Here, we are creating routes in the REST API. Every route connects to a specific endpoint that users or external systems can use to communicate with the application. These routes will perform a variety of actions, such as receiving telemetry data, retrieving information, and updating satellite communication parameters. The following code snippet is how this part works on one parameter:

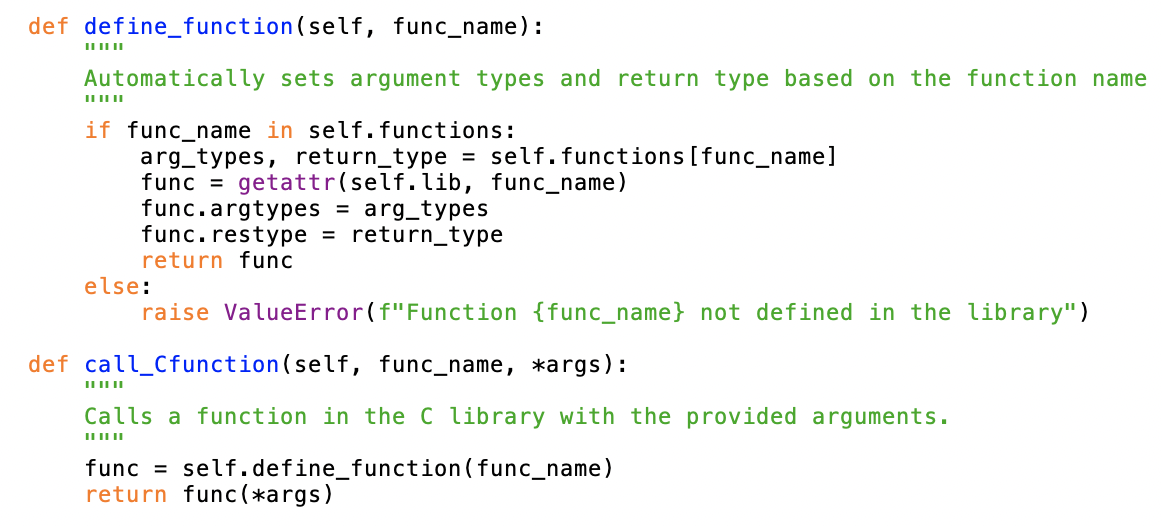


*Figure 15: Example API request POST*

* **Step 3: Interaction between API and libsatcom**

The API calls functions from libsatcom.py, which then interact with the C functions via ctypes to update or retrieve satellite telemetry data. The REST API endpoints receive requests, then they call functions from libsatcom and pass the received data to the respective C function that performs the action.

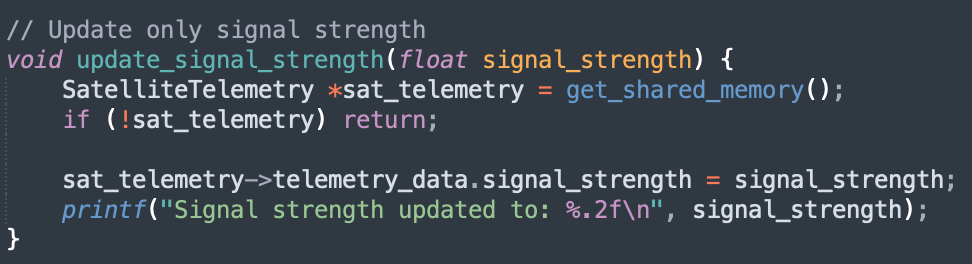




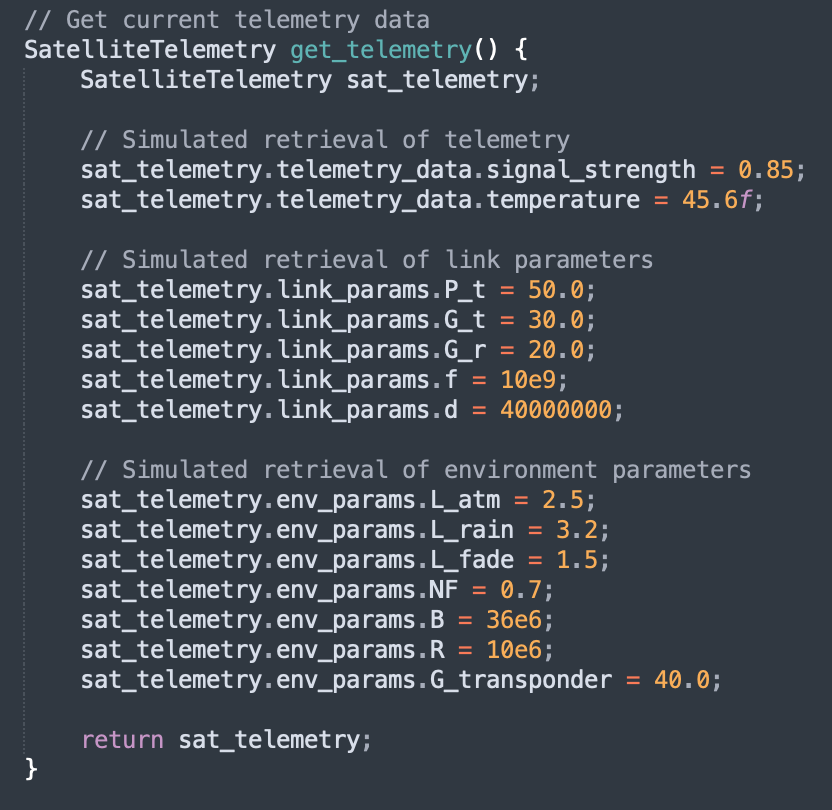
*Figure 16: Custom Shared Library Functions*

* **Step 4: Interactions between C and Python**

The C program, simulate\_satcom.c, provides functions that handle the telemetry tasks such as updating info and retrieving data. The python program, libsatcom.py, is the connection link between the Python code and the C functions. It uses ctypes to call the necessary C functions. The API invokes the C functions through the Python code and passes data from the API endpoints to the C functions. The C functions perform the operations and return the results to the python layer.



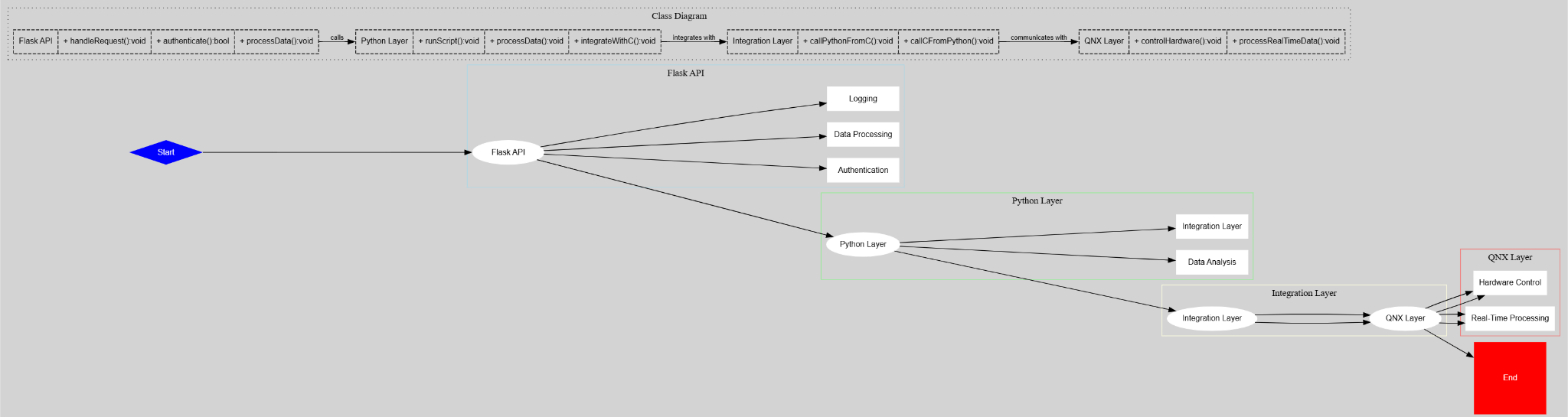
*Figure 17: Example C function to be called*



*Figure 18: Hardcoded data for struct SatteliteTelemetry*

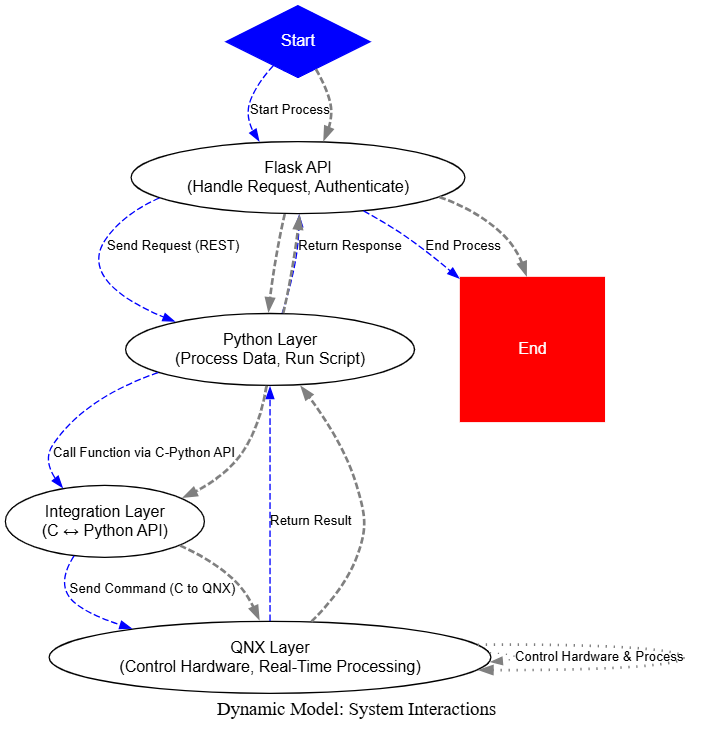
STATIC MODEL

CLASS DIAGRAMS

*Figure 19: Static Model of Diagrams*

DYNAMIC MODEL

SEQUENCE DIAGRAMS



*Figure 20: Dynamic Model of Whole System Interactions*

RATIONALE FOR YOUR DETAILED DESIGN MODEL:

The static model is made up of modular layers (Flask API, Python Layer, Integration Layer, and QNX Layer) for maximum scalability and maintainability. This architecture provides maximum flexibility for modification as it allows low-level operations(QNX Operations) to be separated from high-level logic(Python). The integration layer enables seamless communication, with the Flask API providing standardized external access through RESTful endpoints. The dynamic model illustrates the ordered interactions between components that guarantee proper data flow and task execution. Here, request processing between the Flask API, Python, and QNX layers is operationalized for communication and real-time work.This model acts to ensure dependable execution and error management across the system.

**TRACEABILITY FROM REQUIREMENTS TO DETAILED DESIGN MODEL**

### 

| **Requirement ID** | **Description** | **Mapped Component** |
| --- | --- | --- |
| RQ-001 | Secure authentication | Authentication Service |
| RQ-002 | Data input and validation | Input Validation Module |
| RQ-003 | Report generation | Report Generation Service |
| RQ-004 | System performance optimization | Performance Optimization Strategies |
| RQ-005 | Security compliance | Security & Compliance Module |
| RQ-006 | Scalability and Modularity | Layered System Architecture |
| RQ-007 | Error Handling and Recovery | Fault-Tolerance Mechanisms |
| RQ-008 | Data Encryption | Encryption and Security Layer |
| RQ-009 | REST API for communication | API Gateway & Communication Layer |
| RQ-010 | System monitoring | Logging & Monitoring Service |

# Test Plan REQUIREMENTS/SPECIFICATIONS-BASED SYSTEM LEVEL TEST CASES

| Test Case ID | Requirement ID | Description | Input | Expected Output | Pass/Fail Criteria |
| --- | --- | --- | --- | --- | --- |
| TC-001 | RQ-001 | Verify API returns JWT on valid login | POST /login with valid username/password | Response includes a JWT token | JWT token present in response body |
| TC-002 | RQ-001 | Verify API denies access without authentication | GET /device/status without JWT token | 401 Unauthorized | Response code is 401 |
| TC-003 | RQ-002 | Verify Python CLI can send INIT command to C function | `python3 cli.py --init` | C function `init()` is triggered, and returns “Initialized” | Console prints confirmation of successful init |
| TC-004 | RQ-003 | Verify Python CLI can send START command to device | `python3 cli.py --start` | C function `start()` is triggered, and returns “Started” | Console prints confirmation of successful start |
| TC-005 | RQ-004 | Verify C code executes embedded Python script | Run C program calling embedded script that prints output | Output from Python appears in terminal | Printed Python output is correct and visible |
| TC-006 | RQ-005 | Verify REST API accepts START command via POST | POST /device/start with valid JWT token | Device state changes to “running” | JSON response confirms new state |
| TC-007 | RQ-006 | Verify expired JWT is rejected | POST /device/start with expired token | 401 Unauthorized | Response code is 401 |
| TC-008 | RQ-007 | Verify system recovers after Python script crashes | Simulate Python script crash (e.g., exception or segfault) | System logs crash and restarts the script | Script restarts automatically; log shows recovery |
| TC-009 | RQ-008 | Verify throttle value is accepted via REST API | POST /device/throttle with `{"value": 75}` and JWT | Device receives throttle value and adjusts | Response confirms new throttle; device reflects it |
| TC-010 | RQ-009 | Verify GET /device/status returns correct status | GET /device/status with valid JWT | JSON object includes `status`, `uptime`, `last\_command` fields | Response matches current internal device state |
| TC-011 | RQ-010 | Verify memory usage stays under 100MB during runtime | Run full system including C + Python for 10 minutes | Max memory usage ≤ 100MB | Use monitoring tool like `valgrind` or `top`; logs show limit |
| TC-012 | RQ-012 | Verify system logs error when invalid command is sent | Send malformed JSON or invalid CLI command | Log entry records error with timestamp and context | Log file includes entry about the invalid request |

*Table 7: List of Requirements*

#### TECHNIQUES FOR TEST GENERATION

To create effective test cases on the QNX and Python Integration System, both black-box and white-box testing methods will be applied. As such, these methods would give us an opportunity to analyze the system from two different points of view: functionality (black-box) and internal workings (white-box).

**Black Box Testing:**

* Black box testing is looking at the functionality of the system without knowledge of internal workings. It tests the system as a whole and based only on requirements and specifications. It validates the behavior of the system against user expectations and functional requirements.

1. Test Generation Approach:

* Functional Tests: These will be based on what is functionally important according to the Requirements Documentation. Examples include:
* Verify that the system generates and returns the correct JWT token upon successful authentication (RQ-001).
* Verify that the system handles real-time data processing correctly (RQ-002).
* Verify that the REST API returns correct responses for various HTTP methods (RQ-005).

1. User Scenarios: Use cases and user stories will guide the creation of test scenarios, ensuring that every feature described in the requirements is validated by the tests. For example, we will simulate a user accessing the system via the REST API and check the authentication and data retrieval processes.
2. Error Handling: Test cases will be generated to check the graceful handling of all mentioned errors by the system. For example:

* Using invalid input and checking that appropriate messages are provided.
* Recovery mechanisms will be tested for system failure situations, like having the Python script crash.

1. Advantages:

* From a user perspective focusing on system behavior.
* No knowledge is needed about the internal code structure of a system.
* Its case is excellent in verifying E2E scenario functions.

1. Disadvantages:

* Cannot help in identifying issues related to the internal structure of the system, such as logic or inefficiency in the code.

**White-Box Testing**

White-box testing is testing the internal logic and structure of the system. It basically focuses on the code and its paths, by examining whether all internal functions are implemented properly and whether the system behaves according to expectations on a detailed level.

Test Generation Method:

1. Code Coverage: We will ensure that all branches, conditions, and paths of the code of the system are tested using code coverage tools. This includes testing the Python-C interoperability functions, making sure that Python functions are called from C correctly and vice versa.

* For example, PyObject\_CallObject() and PySys\_SetArgv() should be tested under the following conditions.

1. Path Testing: Tests will be designed to ascertain that every execution path truly is executed including critical paths that involve all the respective areas of system interaction between Python, C, and QNX.

* This means that data and flow must be tested between python and c functions to verify that the information has been passed and returned correctly.

1. Unit Testing: Tests concentrating on individual components or parts of the system. For example, unit tests will be produced for certain Python functions related to data processing or for the functions interfacing QNX hardware.
2. Error Handling Tests: Identify how the system deals with faults at the code level and whether recovery mechanisms are available in the event of unforeseen failures such as in Python scripts or API calls.
3. Advantages:

* Provides insight into how its internals and weaknesses potentially behave.
* Performance bottlenecks, security loopholes, or logical flaws in code can be detected.
* It ensures that all code paths are exercised and checked.

1. Disadvantages:

* Requires extensive knowledge in the code base.
* Might not be possible to cover every user scenario through such testing or fully capture the usage scenario across an end-to-end system.

**Criteria for Measuring the Quality of Tests**

To ensure the quality of our test cases, we will use the following criteria:

1. Coverage:

* Code Coverage: Coverage of the code in overall tests (functions branches loops).
* Requirement Coverage: There will be test cases for every requirement so that each requirement in the documentation is well documented.

1. Defect Detection Rate: The ratio of defects found in testing to those found later in post-release. Higher is better in regard to the test quality.
2. Test Case Effectiveness:

* True Positive Rate: Defect detection
* False Positive Rate: Reporting of failure which is incorrect.

1. Efficiency:

* Test execution time: Testing effectively gives maximum coverage in the least possible time.
* Resource Utilization: Not all tests should be resource consumptive.

1. Maintainability:

* Low Modification Cost: All these tests could change as the system changes.
* Reusability: These tests should be reused over different modules.

1. Traceability:

* Ensuring each test case is traceable to a specific requirement.

1. Robustness:

* The tests should check for proper input error handling and ensure the system recovers from unexpected inputs.

#### TRACEABILITY OF TEST CASES TO USE CASES

* provide a mapping between test cases and use cases
* clearly describe how each requirement in the *Requirements Documentation* is captured in testing

| **Use Case** | **Requirement ID** | **Description** | **Related Test Cases** |
| --- | --- | --- | --- |
| Use Case 1: Execute Python Script from C | RQ-004 | C code must execute a Python script via the Python C API | TC-005 |
| Use Case 2: Control Device from Python via C | RQ-002, RQ-003 | Python CLI sends commands like INIT, START, THROTTLE to control the device by calling C functions | TC-003, TC-004 |
| Use Case 3: REST API Receives Device Control Requests | RQ-005, RQ-008, RQ-009 | REST API endpoints allow clients to control the device remotely | TC-006, TC-009, TC-010 |
| Use Case 4: Authentication via JWT | RQ-001, RQ-006 | API must return a JWT token on successful login and restrict access for unauthenticated users | TC-001, TC-002, TC-007 |
| Use Case 5: Diagnostics and Error Reporting | RQ-005, RQ-012 | System provides diagnostics and logs or reports any device errors | TC-010, TC-012 |
| Use Case 6: Resilience to Python Crashes | RQ-007 | If a Python script crashes during runtime, the system should automatically recover or restart the process | TC-008 |
| Use Case 7: Resource Usage Constraint | RQ-010 | Python runtime must stay under 100MB of memory usage to ensure performance on embedded hardware | TC-011 |

# *Table 8:*

# Evidence the Document Has Been Placed under Configuration Management

# Engineering Standards and Multiple Constraints

* IEEE Std 1058-1998: Software Project Management Plans [[pdf](https://course.techconf.org/se4485/IEEE/IEEE-Std-1058-1998-Software-Project-Management-Plans.pdf)]
* PMBOK® Guide: Project Management Body of Knowledge [[pdf](https://course.techconf.org/se4485/IEEE/PMBOKR.pdf)]
* IEEE Std 12207: Software Life Cycle Processes [[pdf](https://course.techconf.org/se4485/IEEE/IEEE%2012207%20(2017)%20-%20Software%20Life%20Cycle%20Processes.pdf)]
* IEEE Std 15939: Measurement Process [[pdf](https://course.techconf.org/se4485/IEEE/IEEE%2015939%20(2017)%20-%20Measurement%20Process.pdf)]
* ISO/IEC/IEEE Std 29148-2018: Systems and Software Engineering
* Life Cycle Processes
* Requirements Engineering [[pdf](https://course.techconf.org/se4485/IEEE/ISO-IEC-IEEE-29148-2018.pdf)]  
  IEEE Std 830-1998: Software Requirements [[pdf](https://course.techconf.org/se4485/IEEE/IEEE%20Std%20830-1998-Software-Requirements.pdf)]
* IEEE Std 29148: Requirements Engineering [[pdf](https://course.techconf.org/se4485/IEEE/IEEE%2029148%20(2011)%20-%20Requirements%20Engineering.pdf)]
* IEEE Std 1471-2000: Software Architecture [[pdf](https://course.techconf.org/se4485/IEEE/IEEE-Std-1471-2000-Software-Architecture.pdf)]
* ISO/IEC/IEEE Std 42030:2019: Software, Systems and Enterprise
* Architecture Evaluation Framework [[pdf](https://course.techconf.org/se4485/IEEE/ISO-IEC-IEEE-42030-2019.pdf)]
* IEEE Std 1016-1998-(Revision-2009): Software Design [[pdf](https://course.techconf.org/se4485/IEEE/IEEE-Std-1016-1998-(Revision-2009)-Software-Design.pdf)]
* IEEE Std 829-1983: Software Testing [[pdf](https://course.techconf.org/se4485/IEEE/IEEE%20Std%20829-1983-Software-Testing.pdf)]
* ISO/IEC/IEEE Std 29119-1-(Revision-2022): Part 1 - Software Testing General Concepts [[pdf](https://course.techconf.org/se4485/IEEE/IEEE-Std-29119-1-(Revision-2022)-Software-Testing-General-Concepts.pdf)]
* ISO/IEC/IEEE Std 29119-2-(Revision-2021): Part 2 - Test Process [[pdf](https://course.techconf.org/se4485/IEEE/IEEE-Std-29119-2-(Revision-2021)-Test-Process.pdf)]
* ISO/IEC/IEEE Std 29119-3-(Revision-2021): Part 3 - Test Documentation [[pdf](https://course.techconf.org/se4485/IEEE/IEEE-Std-29119-3-(Revision-2021)-Test-Documentation.pdf)]
* ISO/IEC/IEEE Std 29119-4-(Revision-2021): Part 4 - Test Techniques [[pdf](https://course.techconf.org/se4485/IEEE/IEEE%2029119.4%20(2021)%20-%20Test%20Techniques.pdf)]

# Additional References

* Larson, E. and Gray, C., 2014. Project Management: The Managerial Process. McGraw Hill
* Humphrey, W.S. and Thomas, W.R., 2010. Reflections on Management: How to Manage Your Software Projects, Your Teams, Your Boss, and Yourself. Pearson Education
* Lamsweerde, A.V., 2009. Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley
* Lattanze, A.J., 2008. Architecting Software Intensive Systems: A Practitioner’s Guide. CRC Press
* Bass, L., Clements, P. and Kazman, R., 2003. Software Architecture in Practice. Addison-Wesley
* Larman, C., 2012. Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Iterative Development. Pearson Education
* Hyman, B., 1998. Fundamentals of Engineering Design. New Jersey: Prentice Hall
* Simon, H.A., 2014. A Student's Introduction to Engineering Design: Pergamon Unified Engineering Series (Vol. 21). Elsevier
* Jorgensen, P.C., 2013. *Software Testing: A Craftsman's Approach.* Auerbach Publications
* Mathur, A.P., 2013. *Foundations of Software Testing, 2/e.* Pearson Education

# Acknowledgment