SE 4485: Software Engineering Projects

Spring 2025

Requirement Documentation

| 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 Requirement Documentation

**ABSTRACT**

·

The purpose of this document is to outline the functional and non-functional requirements of the system, define its use cases, and establish a structured approach for implementation. The document details the technical specifications, system architecture, and constraints necessary for successful execution. The project ensures a seamless integration between Python and C in a real-time embedded system environment by sticking to these requirements.

**TABLE OF CONTENTS**

1. Abstract
2. List of Figures
3. List of Tables
4. Introduction
   * 4.1 Introduction to the Document
   * 4.2 Purpose and Scope
   * 4.3 Description of Document Structure
5. Use Case Model for Functional Requirements
   * 5.1 Graphic Use Case Model
   * 5.2 Textual Description for Each Use Case
     + Use Case Name
     + Participating Actors
     + Entry Conditions
     + Normal Flow of Events
     + Exit Conditions
     + Exceptions (Alternate Flow of Events)
     + Special Requirements
6. Rationale for the Use Case Model
7. Non-Functional Requirements
8. Evidence the Document Has Been Placed Under Configuration Management
9. Engineering Standards and Multiple Constraints
10. Additional References

LIST OF FIGURES

1. Use Case Diagram for First Implementation
2. Use Case Diagram for Second Implementation
3. Use Case Diagram for Third Implementation

**INTRODUCTION**

The QNX and Python Implementation Requirement Documentation serves as a foundational guide for the project, outlining the key objectives, functionalities, and integration strategies between QNX, C, and Python. This document ensures that all stakeholders, including developers, sponsors, and project managers, have a clear understanding of the technical and functional requirements necessary for successful execution.

The purpose of this document is to define well-structured use cases, system architecture, and constraints, this document establishes a structured approach for development and testing. It covers a detailed description of each Use Case for our implementation. We will define functional and non-functional requirements which will help us with a structured approach for the development of the project.

This document is structured to provide a comprehensive overview of the requirements, architecture, and implementation details of the project.

It includes the following sections:

* Introduction – Provides an overview of the project and document structure.
* Use Case Model for Functional Requirements – Defines the core functionalities of the system using graphical and textual use case descriptions.
* Rationale for the Use Case Model – Justifies the use case selection and structure.
* Non-Functional Requirements – Covers performance, security, reliability, and scalability considerations.
* Evidence of Configuration Management – Details how document versions and changes are managed.
* Engineering Standards and Constraints – Lists industry standards and constraints followed in the project.
* Additional References – Provides citations and references relevant to the project.

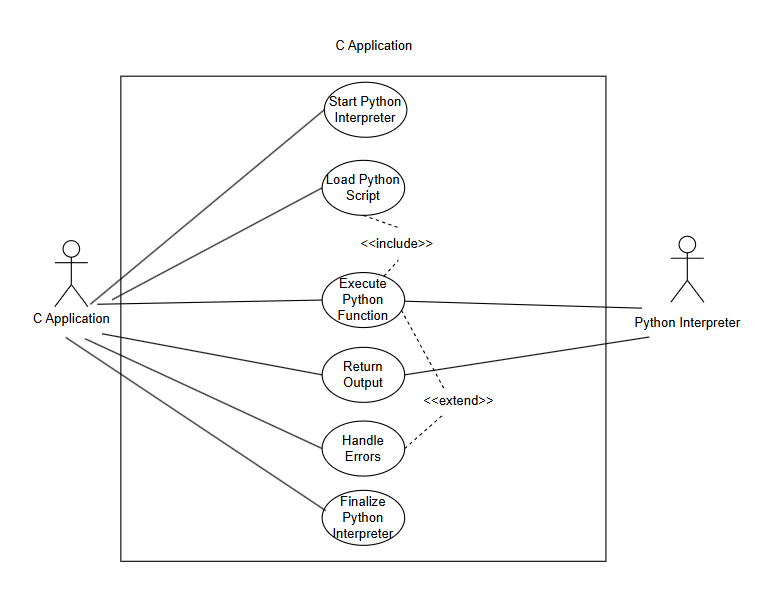
The three implementations are:

Executing Python from C

Executing C from Python

Creating a REST API using Flask

**Use Case Implementation 1:**

****

**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 return an error message and terminate.
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:**

#### 

**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.

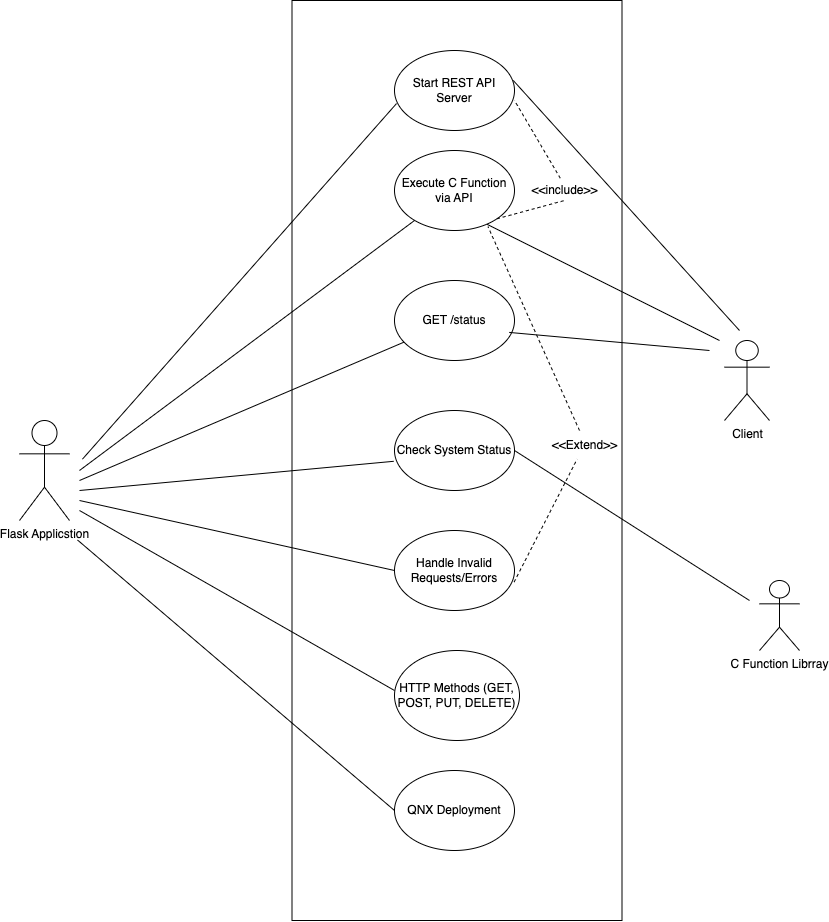
#### **Exceptions (Alternate Flow of Events):**

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:**

****

**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.

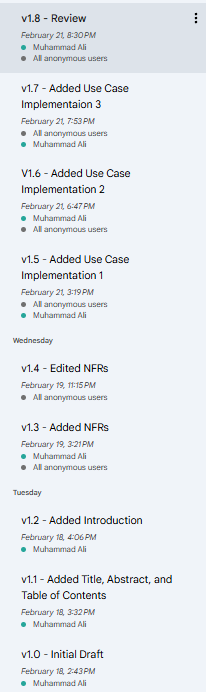
Automation Support: Include hooks for automated testing and debugging.

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.

EVIDENCE THE DOCUMENT HAS BEEN PLACED UNDER CONFIGURATION MANAGEMENT



| Version | Date | Author | Changes Made | Previous Versions | Reviewers | Ship-it |
| --- | --- | --- | --- | --- | --- | --- |
| v1.8 | 2/21/2025 | Muhammad Ali, Nicholas Anderson, Saghar Abdi | Review | v1.7 | Khaled Elkhaled, Diego Ibarra | Approved |
| v1.7 | 2/21/2025 | Khaled Elkhaled, Tabark Abaid, Saghar Abdi | Added Use Case Implementation 3 | v1.6 | Nicholas Anderson, Muhammad Ali | Approved |
| v1.6 | 2/21/2025 | Nicholas Anderson, Khaled Elkhaled | Added Use Case Implementation 2 | v1.5 | Saghar Abdi, Diego Ibarra | Approved |
| v1.5 | 2/21/2025 | Muhammad Ali, Diego Ibarra | Added Use Case Implementation 1 | v1.4 | Nicholas Anderson, Saghar Abdi | Approved |
| v1.4 | 2/19/2025 | Diego Ibarra | Edited Non Functional Requirements | v1.3 | Nicholas Anderson, Tabark Abaid | Approved |
| v1.3 | 2/19/2025 | Diego Ibarra | Added Non Functional Requirements | v1.2 | Nicholas Anderson, Khaled Elkhaled | Approved |
| v1.2 | 2/18/2025 | Muhammad | Added Introduction | v1.1 | Nicholas Anderson, Saghar Abdi | Approved |
| v1.1 | 2/18/2025 | Muhammad Ali | Added Title, Abstract, and Table of Contents | v1.0 | Diego Ibarra, Khaled Elkhaled | Approved |
| v1.0 | 2/18/2025 | Muhammad Ali | Initial Draft | - | - | - |

ENGINEERING STANDARDS AND MULTIPLE CONSTRAINTS

· 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)]

· ISO/IEC/IEEE Std 29148-2018: Systems and Software Engineering

o Life Cycle Processes

o Requirements Engineering [[pdf](https://course.techconf.org/se4485/IEEE/ISO-IEC-IEEE-29148-2018.pdf)]

ADDITIONAL REFERENCES

· Lamsweerde, A.V., 2009. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* John Wiley