Embedded Linux Capstone Project Rubric and Session Guide

Juan Bernardo Gómez-Mendoza
Embedded Linux System Programming 2025-2S

October 16, 2025

1 Embedded Linux Capstone Project Rubric

1.1 Problem Statement, System Overview, and Requirements Specification — 20%

- **Excellent**: Persuasive motivation tied to a real user or problem, supported with market or research input and measurable objectives. Cohesive end-to-end system description, block diagram, hardware interfaces, data/control flow, and platform rationale with trade-offs. Quantified functional and non-functional requirements with testable acceptance criteria and traceability to the problem statement.
- **Proficient**: Clear problem framing with relevant context and achievable goals. Consistent architecture description with diagrams and key hardware/software interfaces. Mostly measurable requirements with some traceability.
- **Developing**: Motivation present but vague or unsupported; goals loosely defined. Partial overview; missing diagram or hardware/software integration details. Mix of requirements and implementation notes; limited measurability.
- **Incomplete**: Missing or generic problem statement; no clear objectives. Fragmented or absent system description. Requirements missing, ambiguous, or unstated.

1.2 Verification Plan and Unit Test Design and Implementation — 20%

- **Excellent**: Comprehensive verification plan. Automated unit test scripts for all modules. Pass/fail evidence for all unit tests. Bug tracking for unit test failures. Code coverage of at least 85% is demonstrated.
- **Proficient**: Documented verification plan. Unit tests for most modules with logs/screenshots. Reproducibility steps for unit tests. Code coverage of at least 70% is demonstrated.
- **Developing**: Some manual or ad hoc unit tests. Limited evidence and no automation. Code coverage not measured.
- **Incomplete**: No meaningful unit testing or documentation.

1.3 Integration Test Design and Test Results — 40%

- **Excellent**: Comprehensive integration test plan including hardware-in-the-loop testing. Automated integration test scripts. Pass/fail evidence for all integration tests. Bug tracking for integration test failures. Test results are presented and discussed.
- **Proficient**: Documented integration tests with logs/screenshots. Reproducibility steps for integration tests. Test results are presented.
- **Developing**: Some manual or ad hoc integration tests. Limited evidence and no automation. Test results are not discussed.
- **Incomplete**: No meaningful integration testing or documentation.

1.4 Final Documentation — 20%

- **Excellent**: Well-structured Markdown documentation (README, setup, user guide, developer guide), updated architecture/design notes, and changelog. Code is well-commented.
- **Proficient**: Clear README with build/run instructions, user instructions, and design summary. Code is commented.
- **Developing**: Basic README; sparse instructions or missing developer info. Code has some comments.
- **Incomplete**: Disorganized repository; critical documentation missing. Code is not commented.

1.5 Submission Requirements

- Public or access-granted GitHub/GitLab repository.
- Markdown documentation covering problem statement, architecture, requirements, design, setup, usage, tests, and known issues.
- Build scripts, configuration, and instructions runnable on Raspberry Pi or Lichee RV Dock.
- Test evidence (logs, screenshots, videos).
- Final demo and reflective report summarizing successes, challenges, and future work.

---

2 Session Guide: Block Diagrams, Requirements, and Verification

2.1 Part 1: Architecture and Requirements

2.1.1 Introduction to System Architecture with Block Diagrams

What is a Block Diagram? A block diagram is a high-level, visual representation of a system. It's designed to show the main components and how they interact, without getting bogged down in the low-level details. The goal is to provide a clear overview of the system's structure. For this project, you will create two types of block diagrams: one for the hardware and one for the software.

Hardware Block Diagram This diagram focuses on the physical components of your system. It should include all the key hardware parts and show how they are physically connected. At the core of your diagram will be your main microprocessor running Linux (e.g., a Raspberry Pi or Lichee RV board).

- **Components to include**: Your Linux-based microprocessor, any optional microcontrollers, sensors, actuators, power systems, and communication interfaces (e.g., UART, SPI, I2C, Wi-Fi/Bluetooth modules).
- **Connections**: Lines between components should represent physical connections, like data buses, power lines, or direct I/O links.

Exercise 1: Your Project's Hardware Block Diagram

- **Task**: Start by placing your main microprocessor (Raspberry Pi or Lichee RV) at the center of your diagram. Then, think about the other physical components of your project. What sensors, actuators, displays, or other parts do you need? Draw a block for each and connect them to the main processor or to each other as appropriate.

Software Block Diagram This diagram illustrates the architecture of your software running on the Linux microprocessor. It shows the main software components and how they exchange data or control signals.

- **Components to include**: Your main application(s), any custom Linux kernel modules or drivers you plan to write, user-space libraries, communication protocols, and scripts. Think about the boundary between kernel space and user space.
- **Connections**: Lines represent the flow of data or control between modules. This could be through system calls, IPC (Inter-Process Communication), or other mechanisms.

Exercise 2: Your Project's Software Block Diagram

- **Task**: Think about the software architecture of your project running on the Linux system. Will you need to write a custom kernel driver for a specific sensor? Will you have a main application running in user space? Will different processes need to communicate? Draw a block diagram showing these software components and how they will interact.

2.1.2 Eliciting System Requirements

What are System Requirements? Requirements are clear, concise statements that describe what the system must do or what qualities it must have. A good set of requirements is crucial for a successful project, as it defines the target you are aiming for.

Functional vs. Non-Functional Requirements

- **Functional Requirements** define **what the system does**. They describe the specific behaviors, functions, or operations of the system.
  - *Example*: "The system shall allow the user to select the coffee strength on a scale of 1 to 5."
- **Non-Functional Requirements** define **how the system should perform its functions**. They describe qualities like performance, reliability, security, or usability.
  - *Example*: "The system shall brew a cup of coffee in under 90 seconds." (Performance)
  - *Example*: "The mobile app shall be compatible with Android 10 and newer." (Compatibility)

Important For this project, the focus is on **eliciting a complete and clear set of requirements**. Don't get stuck on whether a requirement is perfectly functional or non-functional. The goal is to capture everything the system needs to do and all the constraints it must operate under.

Tips for Good Requirements

- **Be Specific and Unambiguous**: Avoid vague terms. "Fast" is not a good requirement; "under 100ms" is.
- **Make them Testable**: For each requirement, think about how you would prove that the system meets it. If you can't test it, it's not a good requirement.
- **Think About the User**: Put yourself in the shoes of the end-user. What do they need to accomplish?

Exercise 3: Your Project's Requirements

- **Task**: Based on your project's problem statement, write down an initial list of at least 5 functional requirements and 3 non-functional requirements. Remember to make them specific, unambiguous, and testable.

2.2 Part 2: Devising a Verification Plan

2.2.1 From Requirements to Testable Cases

A requirement is only useful if you can prove that you've met it. This is where the verification plan comes in. For every requirement you've written, you need to think about how you will test it. This process will often reveal weaknesses in your requirements, forcing you to go back and make them more specific and measurable.

2.2.2 Why Requirements Must Be Testable

Consider a requirement like: "The system should be fast." How do you test that? "Fast" is subjective. A better requirement would be: "The system shall process the input and produce an output in less than 500ms." This is measurable and therefore testable.

Exercise 4: Making Requirements Testable

- **Task**: Take one of the functional requirements you wrote for your project. Is it testable as written? If not, rewrite it to be more specific and measurable. Think about what you would need to measure to prove the requirement is met. For example, if your requirement is about reading a sensor, how quickly does it need to be read? What is the required precision?

2.2.3 Elements of a Verification Plan

A verification plan is a structured way to organize your tests. For each requirement, you will create one or more test cases. A good practice is to create a table with the following columns for each test case. The first set of columns defines the test case itself, while the second set of columns logs the results of each test run.

Test Case Definition

- **Test Case ID**: A unique identifier for the test (e.g., TC-001).
- **Requirement ID**: The ID of the requirement this test case is verifying. It's a good practice to give your requirements unique IDs (e.g., REQ-001, REQ-002).
- **Goal**: A brief description of what this test case aims to verify.
- **Test Procedure**: Step-by-step instructions on how to perform the test. What inputs are needed? What actions should be performed?
- **Expected Outcome**: What is the expected result of the test if the system behaves correctly?
- **Priority**: How important is this test? (e.g., High, Medium, Low). This often corresponds to the priority of the requirement.

Test Run Log

- **Execution Date**: The date when the test was executed.
- **Tester**: The name of the person who executed the test.
- **Outcome**: The result of the test (e.g., Pass, Fail, Blocked).
- **Observations**: Any relevant comments or notes from the test execution, such as error messages or unexpected behavior.

Example Verification Plan Table

Test Case ID	**Req. ID**	**Goal**	**Test Procedure**	**Expected Outcome**	**Prio.**	**Exec. Date**	**Tester**	**Outcome**	**Observations**
TC-001	REQ-001	Verify user can log in with valid credentials.	1. Launch app. 2. Enter valid username. 3. Enter valid password. 4. Tap "Login" button.	User is successfully logged in and redirected to the main screen.	High	2025-10-26	J. Doe	Pass	-

Test Case ID	Req. ID	Goal	Test Procedure	Expected Outcome	Prio.	Exec. Date	Tester	Outcome	Observations
TC-002	REQ-001	Verify user cannot log in with invalid credentials.	1. Launch app. 2. Enter invalid username. 3. Enter invalid password. 4. Tap "Login" button.	An error message "Invalid username or password" is displayed.	High	2025-10-26	J. Doe	Fail	User was logged in. Bug #123 raised.
TC-002	REQ-001	Verify user cannot log in with invalid credentials.	1. Launch app. 2. Enter invalid username. 3. Enter invalid password. 4. Tap "Login" button.	An error message "Invalid username or password" is displayed.	High	2025-10-27	A. Smith	Pass	Bug #123 fixed and verified.

Exercise 5: Creating a Test Case

- **Task**: Choose one of your measurable, functional requirements from the previous exercise. Create a test case for it using the structure described above. Fill out the columns: Test Case ID, Requirement ID, Goal, Test Procedure, Expected Outcome, and Priority.

2.3 Your Goal for the Next Checkpoint

Based on these exercises, your task is to produce the following for your own capstone project:

1. A **hardware block diagram** showing the main physical components and their connections.
2. A **software block diagram** showing the key software modules and their interactions.
3. An initial list of **system requirements** (both functional and non-functional).
4. An initial **verification plan** for your most critical requirements, presented as a table of test cases.