Hardware Virtualization and Trust Report

Overview

The course Hardware Virtualization and Trust introduced us to fundamental concepts in hardware-software interactions, embedded systems, and the embedded Rust programming language. Through a series of labs and projects, we explored the principles of hardware abstraction layers (HAL), low-level programming, and embedded development.

Knowledge acquired:

1. Understanding Hardware-Software Interactions

- Learned hardware components like CPU, stack, heap, memory registers, and interrupts.
- Explored the role of the hardware abstraction layer (HAL) in providing a standardized interface between software and hardware.
- Studied key architectures such as AVR (Atmega328p) and Tensilica LX106 (ESP8266).

2. Programming in Rust

- Memory Safety: Studied Rust's ownership model, borrowing, and its no_std feature for bare-metal environments.
- Cross-Compilation: Compiled Rust code for different targets such as avr-unknown-gnu-atmega328 and xtensa_esp8266_none_elf.
- Toolchain: Utilized Rust's ecosystem (rustc, cargo, and nightly builds) for efficient embedded programming.
- Developed modular code with conditional compilation (#[cfg(feature = ...)]), ensuring compatibility across multiple targets.

3. Hardware Abstraction Layer (HAL) Development through the project

- Created HAL modules for GPIO, SPI, and USART features.
- GPIO: Implemented pin configuration, state reading, and toggling for LEDs.
- SPI: Designed an SPI interface for communication between controllers and peripherals.
- USART: Developed serial communication protocols to send and receive data.
- Abstracted hardware-specific details to make the code reusable across platforms.

4. Debugging and Emulation

- Used both QEMU and SimAVR to emulate target hardware for debugging without physical devices.
- Configured USART emulation via serial ports to monitor and debug embedded applications.

5. Project Highlights

- Target-Specific Implementations.
- Managed Atmega328p and ESP8266-specific registers and operations.
- Provided modularity using feature flags in Cargo.toml.

- Performance Optimization.
- Leveraged Rust's unsafe blocks for direct register manipulation while ensuring safety.
- Incorporated delays using calibrated loops for precise timing control.

Challenges and Resolutions

- Toolchain Bugs: Encountered and resolved linking errors by manually linking .elf files and converting them to .hex format using AVR-GCC and AVRDUDE.
- Limited Documentation: Adapted to low-level programming challenges by referring to datasheets and experimenting with QEMU and SimAVR.

Conclusion

This whole course emphasized a practical, hands-on approach to understanding and implementing hardware-software integration gave us the experience from writing a HAL in Rust, debugging with QEMU and SimAVR, and handling multiple architectures.