HAL Implementation

This project implements a Hardware Abstraction Layer (HAL) in Rust, designed to simplify hardware peripheral usage across two architectures: Atmega328p and RISC-V. The implementation focuses on modularity and abstraction, allowing developers to work with peripherals like USART, GPIO, SPI, and I2C without needing to interact directly with low-level registers. Below is an overview of the functionality provided by each module and how they contribute to the overall HAL design.

main.rs

The main.rs file acts as the entry point of the program and demonstrates how to use the HAL in a practical example. In the provided implementation, the USART module is initialized with a baud rate of 9600, and the string "Hello World\n" is transmitted continuously. This example illustrates the abstraction provided by the USART module, as the same high-level code works seamlessly on both Atmega328p and RISC-V architectures. Conditional compilation (#[cfg(target_arch)]) ensures that the correct hardware-specific implementation is included based on the target architecture. The file showcases how the HAL simplifies peripheral interactions and provides a foundation for further application development.

usart.rs

The USART module abstracts the complexity of serial communication by providing a consistent API for initializing, transmitting, and receiving data. The init function configures the USART module by calculating the appropriate baud rate divider based on the clock frequency and writing it to the relevant registers (UBRR0H and UBRR0L for

Atmega328p or UART0_DIV for RISC-V). It also enables the transmission and reception channels, ensuring the USART is ready for communication.

The transmit function allows sending one byte of data at a time. It ensures the transmit buffer is ready by polling the appropriate status register (UCSR0A for AVR or UART0_TXDATA for RISC-V) before writing the data to the transmit register. Similarly, the receive function reads a single byte of data from the USART module, waiting for the receive buffer to indicate that data is available. By encapsulating these operations, the module provides a simple interface for reliable serial communication.

gpio.rs

The GPIO module provides an abstraction for configuring and controlling digital pins. The new function initializes a GPIO instance, associating it with the relevant registers based on the pin and port (for AVR) or the architecture-specific GPIO base addresses (for RISC-V). This ensures that each GPIO object knows which hardware registers it controls.

The set_mode function configures a pin as either input or output. On AVR, this involves manipulating the Data Direction Register (e.g., DDRB), while on RISC-V, the input and output enable registers (GPIO_INPUT_EN and GPIO_OUTPUT_EN) are used. The function abstracts these details, making it easy to switch pin modes without direct register interaction.

For output pins, the set_level function sets the pin state to high or low. It updates the appropriate register (PORTB for AVR or GPIO_OUTPUT_VAL for RISC-V) to control the pin voltage. For input pins, the read_level function retrieves the current state of the pin by reading from the input register (PINB for AVR or GPIO_INPUT_VAL for RISC-V). This modular design enables consistent GPIO operations regardless of the target architecture.

spi.rs

The SPI module enables synchronous communication with peripheral devices. The spi_init function sets up the SPI peripheral by configuring the clock rate, data order, and SPI mode. On AVR, it writes to specific control registers (e.g., SPCR) to enable the SPI master mode and define the clock frequency. On RISC-V, it performs similar tasks using architecture-specific registers.

The spi_transmit function handles the sending and receiving of data over the SPI bus. Since SPI is a full-duplex protocol, this function writes data to the transmit buffer while simultaneously reading the response from the receive buffer. It ensures the

transmission is complete by polling the status register before proceeding, providing a reliable interface for SPI communication.

i2c.rs

The I2C module abstracts the operations required for communication over the I2C bus. The i2c_init function configures the clock prescaler and enables the I2C interface, ensuring compatibility with standard devices. On AVR, this involves writing to the TWBR register to set the clock frequency, while on RISC-V, a similar configuration is performed using the relevant hardware registers.

The i2c_write_byte function sends a single byte to a specific device address. It generates a START condition, sends the device address and the data byte, and terminates with a STOP condition. The function handles the necessary waiting and flag polling to ensure each operation completes successfully. Similarly, the i2c_read_byte function reads a byte from a device by generating a START condition, sending the device address with the read bit, and receiving the data. It abstracts the complexities of managing the I2C state machine, providing a simple and reliable interface for multi-device communication.