Hardware Virtualization Project Report

KACI Inès, MATHEWS Louis-Marie, MORDI David

Date: 4 January 2025

1. Introduction

This report covers the code implementations for different hardware virtualization techniques, including GPIO, USART, SPI, and I2C. Each section presents how the protocol works, shows the code, and explains what it does in detail. The code is designed to work on various architectures like ARM (Cortex-M) and AVR, for flexibility.

2. GPIO Protocol and Implementation

2.1. Protocol

GPIO (General Purpose Input/Output) allows to control individual pins of a microcontroller for interfacing with external devices like LEDs or switches.The main operation is configuring pins as input or output and toggling their states.

2.2. ARM Cortex-M

Code

```
#![no_std]
#![no_main]

use panic_halt as _;
use tp1::gpio::{Gpio, GpioTrait};

#[cfg(target_arch = "arm")]
use tp1::rcc::Rcc;

#[cfg(target_arch = "arm")]
use tp1::gpio::stm32f1::GpioPort;

#[cfg(target_arch = "arm")]
use cortex_m_rt::entry;

#[cfg(target_arch = "arm")]
#[entry]
fn main() -> ! {
    unsafe {
        Rcc::enable_gpio_port_clock(GpioPort::C);
        let gpio = Gpio {port: GpioPort::C};
        gpio.set_pin_output(13);

        loop {
            gpio.set_pin_high(13);
            delay(8_000_000);
            gpio.set_pin_low(13);
            delay(8_000_000);
        }
    }
}
```

Comments

- **Clock Enable**: The clock for GPIOC is enabled to permit pin manipulation.
- **Pin Configuration**: The pin PC13 is set as an output for toggling the LED.
- **LED Control**: A loop allows the LED's state to change with delays.

2.3. AVR

Code

```
#[cfg(target_arch = "avr")]
#[no_mangle]
fn main() -> ! {
    let gpio = Gpio;
    gpio.set_pin_output(5);
    gpio.set_pin_high(5);

    loop {}
}
```

Comments

- Configures pin 5 as an output and sets it high to activate the connected device.
- The code is not exactly the same for the AVR target and the STM32F1 target because some details are platform-specific and cannot be abstracted by a common interface of the library. Other details are common though, such as the Gpio trait.

3. USART Protocol and Implementation

3.1. Protocol

USART (Universal Synchronous/Asynchronous Receiver Transmitter) facilitates serial communication between devices by transmitting data byte by byte.

3.2. ARM Cortex-M

Code

```
#![no_std]
#![no_main]

use panic_halt as _;
use tp1::usart::{Usart, UsartTrait};

#[cfg(target_arch = "arm")]
use tp1::usart::stm32f1::UsartPeripheral;

#[cfg(target_arch = "arm")]
use cortex_m_rt::entry;

#[cfg(target_arch = "arm")]
#[entry]
fn main() -> ! {
    let usart = Usart{
        peripheral: UsartPeripheral::Usart1,
        use_9_bit_words: false,
    };
    usart.init();

    loop {
        let message = "Hello USART\r\n";
        usart.send_message(message);
        usart.set_listening_status(true);
        let byte = usart.receive_byte();
```

```
        usart.transmit_byte(byte);
        usart.set_listening_status(false);
    }
}
```

Comments

- **Initialization**: Configures USART1 for the communication.
- **Message Transmission**: Sends a test message.
- **Echo Mechanism**: Receives and retransmits a byte when the message is received.

3.3. AVR

Code

```
#[cfg(target_arch = "avr")]
#[no_mangle]
fn main() {
    let usart = Usart {};
    usart.init();
    usart.send_message("Hello, USART!");

    loop {
        let received_byte = usart.receive_byte();
        usart.send_message("Recu : ");
        usart.transmit_byte(received_byte);
    }
}
```

Comments

- Similar to ARM but adds acknowledgment messages for received data by displaying the received message.

4. SPI Protocol and Implementation

4.1. Protocol

SPI (Serial Peripheral Interface) is a synchronous protocol of communication used for high-speed data transfer between microcontrollers and peripherals like sensors and memory devices.

4.2. ARM Cortex-M

Code

```
#![no_std]
#![no_main]

use panic_halt as _;
use tp1::spi::{Spi, SpiTrait};

#[cfg(target_arch = "arm")]
use tp1::spi::stm32f1::SpiPeripheral;

#[cfg(target_arch = "arm")]
use cortex_m_rt::entry;

#[cfg(target_arch = "arm")]
```

```
#[entry]
fn main() -> ! {
    let spi = Spi {
        peripheral: SpiPeripheral::Spi1,
        clock_polarity: 0,
        clock_phase: 1,
    };
    spi.init();

    loop {
        let data = [0x01, 0x02, 0x03];
        spi.send_data(&data);
        let response = spi.receive_data();
        process_response(response);
    }
}
```

Comments

- **Initialization**: Configures SPI1 with specific clock settings.
- **Data Transfer**: Proceeds to send and receive data.
- **Processing**: Handles the response data in a loop.

4.3. AVR

Code

```
#[cfg(target_arch = "avr")]
#[no_mangle]
fn main() {
    let spi = Spi {};
    spi.init();
    let data = [0x0A, 0x0B];
    spi.send_data(&data);

    loop {
        let received = spi.receive_data();
        spi.send_data(&received);
    }
}
```

Comments

- Similar to ARM.

5. I2C Protocol and Implementation

5.1. Protocol

I2C (Inter-Integrated Circuit) is a protocol allowing several master-slave communication using only two lines, making it a popular choice for low-data rates, cheap communication lines. It is commonly used for short-distance communication between devices on a PCB.

5.2. ARM Cortex-M

Code

```
#![no_std]
#![no_main]

use panic_halt as _;
use tp1::i2c::{I2c, I2cTrait};

#[cfg(target_arch = "arm")]
use tp1::i2c::stm32f1::I2cPeripheral;

#[cfg(target_arch = "arm")]
use cortex_m_rt::entry;

#[cfg(target_arch = "arm")]
#[entry]
fn main() -> ! {
    let i2c = I2c {
        peripheral: I2cPeripheral::I2c1,
    };
    i2c.init();

    loop {
        let address = 0x50;
        let data_to_send = [0xA5, 0xB4];
        i2c.write_data(address, &data_to_send);

        let mut received_data = [0; 2];
        i2c.read_data(address, &mut received_data);
        process_received_data(received_data);
    }
}
```

Comments

- Configures I2C1 and writes/reads the data.

5.3. AVR

Code

```
#[cfg(target_arch = "avr")]
#[no_mangle]
fn main() {
    let i2c = I2c {};
    i2c.init();
    let address = 0x60;
    let data = [0x01, 0x02];
    i2c.write_data(address, &data);

    loop {
        let mut buffer = [0; 2];
        i2c.read_data(address, &mut buffer);
    }
}
```

Comments

- Use of the I2C communication protocol with a specific address.

6. Emulation

In order to test the code without access to any hardware, we used two emulation software:

- QEMU
- Renode

While more mature and more established, QEMU does not have sufficient support for certain targets and peripherals, such as the STM32F1 and testing certain peripherals such as I2C and GPIO.

Renode, although it supports less targets, is much more extendable. The specific platform configuration can be defined in a Renode Platform File (.repl) or in a Renode Script (.resc). The script file can load an ELF file into the flash memory of the virtual MCU, and monitor it (such as seeing which pin is active or not).

7. Conclusion

Throught the implementations of these presented code, we could highlight the versatility of hardware virtualization across GPIO, USART, SPI, and I2C protocols. By using both ARM and AVR architectures, this report is practical as a reference for embedded systems development in future projects.