МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій

Кафедра комп'ютерної інженерії та інформаційних систем

ЛАБОРАТОРНА РОБОТА №3

з дисципліни «Теорія і технології проектування спеціалізованих операційних систем»

ЛРКІ. 180102.22.01.02 ПЗ

Галузь знань \_\_\_\_\_\_\_\_\_\_12 – Інформаційні технології\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Спеціальність \_\_\_\_\_\_\_\_\_123 –Комп’ютерна інженерія\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Виконав:

студент 1 курсу, групи КІ2м-22-1 Головатюк А.О.

(Підпис)

Перевірив: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Лисенко С.М.

(Підпис)

Хмельницький – 2023 рік

**Тема:**

Проєктування спеціалізованої RTOS для Інтернету речей на платформі Raspberry Pi. Реалізація рівнів привілеїв. Апаратне налагодження по JTAG. Реалізація рівнів привілеїв. Тестування на реальній платформі

**Завдання:**

Реалізувати етапи 8, 9 проєктування спеціалізованої RTOS для платформи Raspberry Pi3. Протестувати поточну версію ядра та зробити відповідні висновки. Пояснити відмінності реалізації для платформи Raspberry Pi4. Оформити звіт з виконаної роботи.

**Виконання завдання:**

Етап 8.

На цьому етапі відбувається налаштування програмного та апаратного забезпечення для налагодження Raspberry Pi через JTAG. Опис кроків, які потрібно виконати:

1. Виконати команду "make jtagboot" і тримати термінал відкритим. Це запускає процес завантаження допоміжного бінарного файлу в Raspberry Pi за допомогою chainboot, який переводить виконуюче ядро в стан очікування.
2. Підключити USB-кабель для з'єднання з послідовним портом (serial USB). Це дозволить спостерігати вивід UART під час завантаження фактичного ядра.
3. Підключити USB-пристрій відладчика JTAG, такий як Olimex ARM-USB-TINY-H, до Raspberry Pi за допомогою підключення через GPIO за допомогою джамперів.
4. В новому терміналі виконати команду "make openocd" для запуску OpenOCD (Open On-Chip Debugger), яке встановлює з'єднання з JTAG. OpenOCD виявляє чотири ядра Raspberry Pi і відкриває чотири мережеві порти, до яких можна підключитися для налагодження відповідних ядер.
5. В новому терміналі виконати команду "make gdb" або "make gdb-opt0" для запуску AArch64 версії gdb (GNU Debugger). Це дозволяє налаштовувати точки зупинки, завантажувати ядро через JTAG, маніпулювати лічильниками програм Raspberry Pi та виконувати покрокове виконання коду.

Під час цих кроків налаштовується взаємодія між програмним та апаратним забезпеченням Raspberry Pi для забезпечення налагодження через JTAG. Виконання цих кроків дозволяє налагоджувати складні проблеми, пов'язані з апаратурою, які не завжди можна вирішити за допомогою QEMU або інших інструментів.

Код реалізації:

Файл Cargo.toml

[package]

name = "mingo"

version = "0.8.0"

authors = ["Andre Richter <andre.o.richter@gmail.com>"]

edition = "2021"

[profile.release]

lto = true

[features]

default = []

bsp\_rpi3 = ["tock-registers"]

bsp\_rpi4 = ["tock-registers"]

[[bin]]

name = "kernel"

path = "src/main.rs"

[dependencies]

tock-registers = { version = "0.8.x", default-features = false, features = ["register\_types"], optional = true }

[target.'cfg(target\_arch = "aarch64")'.dependencies]

aarch64-cpu = { version = "9.x.x" }

Файл Makefile

include ../common/docker.mk

include ../common/format.mk

include ../common/operating\_system.mk

BSP ?= rpi3

DEV\_SERIAL ?= /dev/ttyUSB0

QEMU\_MISSING\_STRING = "This board is not yet supported for QEMU."

ifeq ($(BSP),rpi3)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE = raspi3

QEMU\_RELEASE\_ARGS = -serial stdio -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

OPENOCD\_ARG = -f /openocd/tcl/interface/ftdi/olimex-arm-usb-tiny-h.cfg -f /openocd/rpi3.cfg

JTAG\_BOOT\_IMAGE = ../X1\_JTAG\_boot/jtag\_boot\_rpi3.img

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a53

else ifeq ($(BSP),rpi4)

TARGET = aarch64-unknown-none-softfloat

KERNEL\_BIN = kernel8.img

QEMU\_BINARY = qemu-system-aarch64

QEMU\_MACHINE\_TYPE =

QEMU\_RELEASE\_ARGS = -serial stdio -display none

OBJDUMP\_BINARY = aarch64-none-elf-objdump

NM\_BINARY = aarch64-none-elf-nm

READELF\_BINARY = aarch64-none-elf-readelf

OPENOCD\_ARG = -f /openocd/tcl/interface/ftdi/olimex-arm-usb-tiny-h.cfg -f /openocd/rpi4.cfg

JTAG\_BOOT\_IMAGE = ../X1\_JTAG\_boot/jtag\_boot\_rpi4.img

LD\_SCRIPT\_PATH = $(shell pwd)/src/bsp/raspberrypi

RUSTC\_MISC\_ARGS = -C target-cpu=cortex-a72

endif

export LD\_SCRIPT\_PATH

KERNEL\_MANIFEST = Cargo.toml

KERNEL\_LINKER\_SCRIPT = kernel.ld

LAST\_BUILD\_CONFIG = target/$(BSP).build\_config

KERNEL\_ELF = target/$(TARGET)/release/kernel

# This parses cargo's dep-info file.

# https://doc.rust-lang.org/cargo/guide/build-cache.html#dep-info-files

KERNEL\_ELF\_DEPS = $(filter-out %: ,$(file < $(KERNEL\_ELF).d)) $(KERNEL\_MANIFEST) $(LAST\_BUILD\_CONFIG)

RUSTFLAGS = $(RUSTC\_MISC\_ARGS) \

-C link-arg=--library-path=$(LD\_SCRIPT\_PATH) \

-C link-arg=--script=$(KERNEL\_LINKER\_SCRIPT)

RUSTFLAGS\_PEDANTIC = $(RUSTFLAGS) \

-D warnings \

-D missing\_docs

FEATURES = --features bsp\_$(BSP)

COMPILER\_ARGS = --target=$(TARGET) \

$(FEATURES) \

--release

RUSTC\_CMD = cargo rustc $(COMPILER\_ARGS)

DOC\_CMD = cargo doc $(COMPILER\_ARGS)

CLIPPY\_CMD = cargo clippy $(COMPILER\_ARGS)

OBJCOPY\_CMD = rust-objcopy \

--strip-all \

-O binary

EXEC\_QEMU = $(QEMU\_BINARY) -M $(QEMU\_MACHINE\_TYPE)

EXEC\_TEST\_DISPATCH = ruby ../common/tests/dispatch.rb

EXEC\_MINIPUSH = ruby ../common/serial/minipush.rb

DOCKER\_CMD = docker run -t --rm -v $(shell pwd):/work/tutorial -w /work/tutorial

DOCKER\_CMD\_INTERACT = $(DOCKER\_CMD) -i

DOCKER\_ARG\_DIR\_COMMON = -v $(shell pwd)/../common:/work/common

DOCKER\_ARG\_DIR\_JTAG = -v $(shell pwd)/../X1\_JTAG\_boot:/work/X1\_JTAG\_boot

DOCKER\_ARG\_DEV = --privileged -v /dev:/dev

DOCKER\_ARG\_NET = --network host

DOCKER\_QEMU = $(DOCKER\_CMD\_INTERACT) $(DOCKER\_IMAGE)

DOCKER\_TOOLS = $(DOCKER\_CMD) $(DOCKER\_IMAGE)

DOCKER\_TEST = $(DOCKER\_CMD) $(DOCKER\_ARG\_DIR\_COMMON) $(DOCKER\_IMAGE)

DOCKER\_GDB = $(DOCKER\_CMD\_INTERACT) $(DOCKER\_ARG\_NET) $(DOCKER\_IMAGE)

ifeq ($(shell uname -s),Linux)

DOCKER\_CMD\_DEV = $(DOCKER\_CMD\_INTERACT) $(DOCKER\_ARG\_DEV)

DOCKER\_CHAINBOOT = $(DOCKER\_CMD\_DEV) $(DOCKER\_ARG\_DIR\_COMMON) $(DOCKER\_IMAGE)

DOCKER\_JTAGBOOT = $(DOCKER\_CMD\_DEV) $(DOCKER\_ARG\_DIR\_COMMON) $(DOCKER\_ARG\_DIR\_JTAG) $(DOCKER\_IMAGE)

DOCKER\_OPENOCD = $(DOCKER\_CMD\_DEV) $(DOCKER\_ARG\_NET) $(DOCKER\_IMAGE)

else

DOCKER\_OPENOCD = echo "Not yet supported on non-Linux systems."; \#

endif

.PHONY: all doc qemu chainboot clippy clean readelf objdump nm check

all: $(KERNEL\_BIN)

$(LAST\_BUILD\_CONFIG):

@rm -f target/\*.build\_config

@mkdir -p target

@touch $(LAST\_BUILD\_CONFIG)

$(KERNEL\_ELF): $(KERNEL\_ELF\_DEPS)

$(call color\_header, "Compiling kernel ELF - $(BSP)")

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(RUSTC\_CMD)

$(KERNEL\_BIN): $(KERNEL\_ELF)

$(call color\_header, "Generating stripped binary")

@$(OBJCOPY\_CMD) $(KERNEL\_ELF) $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Name")

@echo $(KERNEL\_BIN)

$(call color\_progress\_prefix, "Size")

$(call disk\_usage\_KiB, $(KERNEL\_BIN))

doc:

$(call color\_header, "Generating docs")

@$(DOC\_CMD) --document-private-items --open

ifeq ($(QEMU\_MACHINE\_TYPE),) # QEMU is not supported for the board.

qemu:

$(call color\_header, "$(QEMU\_MISSING\_STRING)")

else # QEMU is supported.

qemu: $(KERNEL\_BIN)

$(call color\_header, "Launching QEMU")

@$(DOCKER\_QEMU) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) -kernel $(KERNEL\_BIN)

endif

chainboot: $(KERNEL\_BIN)

@$(DOCKER\_CHAINBOOT) $(EXEC\_MINIPUSH) $(DEV\_SERIAL) $(KERNEL\_BIN)

clippy:

@RUSTFLAGS="$(RUSTFLAGS\_PEDANTIC)" $(CLIPPY\_CMD)

clean:

rm -rf target $(KERNEL\_BIN)

readelf: $(KERNEL\_ELF)

$(call color\_header, "Launching readelf")

@$(DOCKER\_TOOLS) $(READELF\_BINARY) --headers $(KERNEL\_ELF)

objdump: $(KERNEL\_ELF)

$(call color\_header, "Launching objdump")

@$(DOCKER\_TOOLS) $(OBJDUMP\_BINARY) --disassemble --demangle \

--section .text \

--section .rodata \

$(KERNEL\_ELF) | rustfilt

nm: $(KERNEL\_ELF)

$(call color\_header, "Launching nm")

@$(DOCKER\_TOOLS) $(NM\_BINARY) --demangle --print-size $(KERNEL\_ELF) | sort | rustfilt

.PHONY: jtagboot openocd gdb gdb-opt0

jtagboot:

@$(DOCKER\_JTAGBOOT) $(EXEC\_MINIPUSH) $(DEV\_SERIAL) $(JTAG\_BOOT\_IMAGE)

openocd:

$(call color\_header, "Launching OpenOCD")

@$(DOCKER\_OPENOCD) openocd $(OPENOCD\_ARG)

gdb: RUSTC\_MISC\_ARGS += -C debuginfo=2

gdb-opt0: RUSTC\_MISC\_ARGS += -C debuginfo=2 -C opt-level=0

gdb gdb-opt0: $(KERNEL\_ELF)

$(call color\_header, "Launching GDB")

@$(DOCKER\_GDB) gdb-multiarch -q $(KERNEL\_ELF)

.PHONY: test test\_boot

ifeq ($(QEMU\_MACHINE\_TYPE),) # QEMU is not supported for the board.

test\_boot test:

$(call color\_header, "$(QEMU\_MISSING\_STRING)")

else # QEMU is supported.

test\_boot: $(KERNEL\_BIN)

$(call color\_header, "Boot test - $(BSP)")

@$(DOCKER\_TEST) $(EXEC\_TEST\_DISPATCH) $(EXEC\_QEMU) $(QEMU\_RELEASE\_ARGS) -kernel $(KERNEL\_BIN)

test: test\_boot

endif

Файл driver.rs

use crate::{

info,

synchronization::{interface::Mutex, NullLock},

};

const NUM\_DRIVERS: usize = 5;

struct DriverManagerInner {

next\_index: usize,

descriptors: [Option<DeviceDriverDescriptor>; NUM\_DRIVERS],

}

pub mod interface {

pub trait DeviceDriver {

fn compatible(&self) -> &'static str;

unsafe fn init(&self) -> Result<(), &'static str> {

Ok(())

}

}

}

pub type DeviceDriverPostInitCallback = unsafe fn() -> Result<(), &'static str>;

#[derive(Copy, Clone)]

pub struct DeviceDriverDescriptor {

device\_driver: &'static (dyn interface::DeviceDriver + Sync),

post\_init\_callback: Option<DeviceDriverPostInitCallback>,

}

pub struct DriverManager {

inner: NullLock<DriverManagerInner>,

}

static DRIVER\_MANAGER: DriverManager = DriverManager::new();

impl DriverManagerInner {

pub const fn new() -> Self {

Self {

next\_index: 0,

descriptors: [None; NUM\_DRIVERS],

}

}

}

impl DeviceDriverDescriptor {

pub fn new(

device\_driver: &'static (dyn interface::DeviceDriver + Sync),

post\_init\_callback: Option<DeviceDriverPostInitCallback>,

) -> Self {

Self {

device\_driver,

post\_init\_callback,

}

}

}

pub fn driver\_manager() -> &'static DriverManager {

&DRIVER\_MANAGER

}

impl DriverManager {

pub const fn new() -> Self {

Self {

inner: NullLock::new(DriverManagerInner::new()),

}

}

pub fn register\_driver(&self, descriptor: DeviceDriverDescriptor) {

self.inner.lock(|inner| {

inner.descriptors[inner.next\_index] = Some(descriptor);

inner.next\_index += 1;

})

}

fn for\_each\_descriptor<'a>(&'a self, f: impl FnMut(&'a DeviceDriverDescriptor)) {

self.inner.lock(|inner| {

inner

.descriptors

.iter()

.filter\_map(|x| x.as\_ref())

.for\_each(f)

})

}

pub unsafe fn init\_drivers(&self) {

self.for\_each\_descriptor(|descriptor| {

// 1. Initialize driver.

if let Err(x) = descriptor.device\_driver.init() {

panic!(

"Error initializing driver: {}: {}",

descriptor.device\_driver.compatible(),

x

);

}

if let Some(callback) = &descriptor.post\_init\_callback {

if let Err(x) = callback() {

panic!(

"Error during driver post-init callback: {}: {}",

descriptor.device\_driver.compatible(),

x

);

}

}

});

}

pub fn enumerate(&self) {

let mut i: usize = 1;

self.for\_each\_descriptor(|descriptor| {

info!(" {}. {}", i, descriptor.device\_driver.compatible());

i += 1;

});

}

}

Результат:



Етап 9.

На наступному етапі відбувається реалізація рівнів привілеїв в контексті Raspberry Pi. Завантаження починається в рівні привілеїв гіпервізора (EL2 в AArch64) і переходить до рівня привілеїв ядра (EL1).

Спочатку в коді завантаження додається перевірка, щоб переконатися, що виконання відбувається в EL2. Якщо це не так, ядро паркується. Це забезпечує, що виконання коду продовжується лише в EL2.

Далі, за допомогою виклику prepare\_el2\_to\_el1\_transition(), готується перехід від EL2 до EL1. Це включає налаштування різних реєстрів та параметрів для переходу від більш привілейованого рівня EL2 до менш привілейованого рівня EL1.

Одним з таких налаштувань є встановлення прапорів для включення регістрів таймера в EL1 та встановлення віртуального зміщення на нуль для отримання реального фізичного значення таймера.

Також встановлюється режим виконання AArch64 для EL1, а не AArch32.

Остаточним кроком є виклик інструкції ERET, яка здійснює перехід від вищого рівня EL2 до нижчого рівня EL1. Це викликає функцію kernel\_init() в EL1.

Код реалізації:

Файл Cargo.toml

[package]

name = "mingo"

version = "0.9.0"

authors = ["Andre Richter <andre.o.richter@gmail.com>"]

edition = "2021"

[profile.release]

lto = true

[features]

default = []

bsp\_rpi3 = ["tock-registers"]

bsp\_rpi4 = ["tock-registers"]

[[bin]]

name = "kernel"

path = "src/main.rs"

[dependencies]

tock-registers = { version = "0.8.x", default-features = false, features = ["register\_types"], optional = true }

[target.'cfg(target\_arch = "aarch64")'.dependencies]

aarch64-cpu = { version = "9.x.x" }

Файл boot.rs

#[cfg(target\_arch = "aarch64")]

#[path = "../\_arch/aarch64/cpu/boot.rs"]

mod arch\_boot;

Файл main.rs

mod bsp;

mod console;

mod cpu;

mod driver;

mod exception;

mod panic\_wait;

mod print;

mod synchronization;

mod time;

unsafe fn kernel\_init() -> ! {

if let Err(x) = bsp::driver::init() {

panic!("Error initializing BSP driver subsystem: {}", x);

}

driver::driver\_manager().init\_drivers();

kernel\_main()

}

fn kernel\_main() -> ! {

use console::console;

use core::time::Duration;

info!(

"{} version {}",

env!("CARGO\_PKG\_NAME"),

env!("CARGO\_PKG\_VERSION")

);

info!("Booting on: {}", bsp::board\_name());

let (\_, privilege\_level) = exception::current\_privilege\_level();

info!("Current privilege level: {}", privilege\_level);

info!("Exception handling state:");

exception::asynchronous::print\_state();

info!(

"Architectural timer resolution: {} ns",

time::time\_manager().resolution().as\_nanos()

);

info!("Drivers loaded:");

driver::driver\_manager().enumerate();

info!("Timer test, spinning for 1 second");

time::time\_manager().spin\_for(Duration::from\_secs(1));

info!("Echoing input now");

console().clear\_rx();

loop {

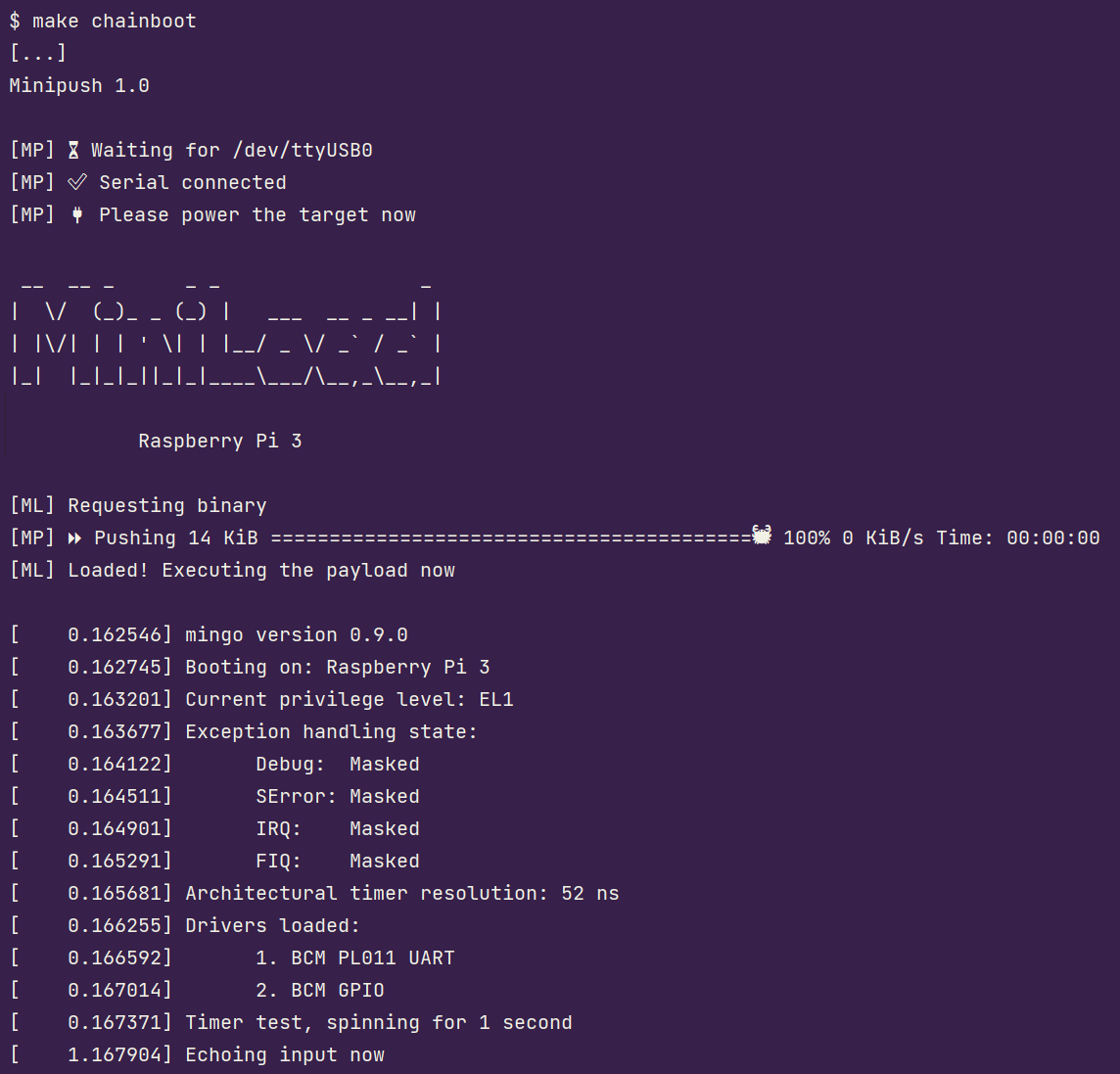
let c = console().read\_char();

console().write\_char(c);

}

}

Результат:



**Висновок:**

Останні два кроки, які описуються, є важними етапами в розробці та налагодженні Raspberry Pi через JTAG. Налагодження через JTAG дозволяє отримати доступ до програмного та апаратного забезпечення пристрою, що є корисним при розробці та налагодженні вбудованих систем.

Під час налаштування програмного та апаратного забезпечення для налагодження Raspberry Pi через JTAG, виконуються ряд кроків, таких як налаштування GPIO для JTAG-інтерфейсу, забезпечення правильного живлення, встановлення необхідного програмного забезпечення, наприклад, OpenOCD, і підключення до JTAG-адаптера. Це дозволяє розробникам використовувати різноманітні інструменти та можливості для налагодження й аналізу роботи пристрою.

Одна з відмінностей між реалізаціями для платформ Raspberry Pi 3 та Raspberry Pi 4 полягає в їх архітектурі та рівнях привілеїв. У Raspberry Pi 3 використовується 32-розрядна архітектура ARMv8, в той час як Raspberry Pi 4 використовує 64-розрядну архітектуру ARMv8. Це призводить до різних процесів налаштування та реалізації рівнів привілеїв. Наприклад, в Raspberry Pi 3 перехід від рівня EL2 до EL1 відбувається за допомогою інструкції ERET, тоді як в Raspberry Pi 4 використовується перехід через віртуальне зміщення та інші налаштування реєстрів.

У загальному, налаштування програмного та апаратного забезпечення для налагодження Raspberry Pi через JTAG та реалізація рівнів привілеїв в контексті Raspberry Pi є важливими етапами, що дозволяють розробникам отримувати доступ та контроль над пристроєм для ефективної розробки, налагодження та тестування вбудованих систем.