

# RISC-V 与 Rust 语言与嵌入式

车春池

2021-03-24

email: [linuxgnulover@gmail.com](mailto:linuxgnulover@gmail.com)

github: <https://github.com/SKTT1Ryze>

## Outline

- RISC-V 生态
- RISC-V 与嵌入式
- Rust 语言简介
- Rust 语言与嵌入式
- Rust 语言与 RISC-V 嵌入式

## RISC-V 生态

- 开放的精简指令集架构，完全开源，设计简洁，模块化设计（基本指令集 + 扩展指令集）
- 与 X86 架构对比：简化芯片开发，抛弃历史包裹，但目前在高性能处理核领域稍逊
- 与 Arm 架构对比：开放，大道至简，但生态上需要进一步发展
- 问题：IP 碎片化和高性能

## RISC-V 生态

### RISC-V 在嵌入式领域大有可为


“对比 Arm, RISC-V 在成本, 功耗, 性能上目前都没有显著的优势, ”, 国内某知名芯片厂商产品经理, “但 RISC-V 的开放性会给嵌入式行业带来一个新的模式。”

(出处: 2020-12-26 全国操作系统大赛研讨会现场台下)

## RISC-V 与嵌入式

- 功耗与性能：和 Arm 不分上下
- 成本：对比 Arm 较低
  - 设计成本较低：简洁的设计风格（规整的指令结构，简单的寄存器组成），模块化
  - 架构授权：完全开源，而 Arm 需要支付昂贵的授权费


# RISC-V vs Arm



## ARM Cortex-A5 vs. RISC-V Rocket

Category	ARM Cortex-A5	RISC-V Rocket
ISA	32-bit ARM v7	64-bit RISC-V v2
Architecture	Single-Issue In-Order	Single-Issue In-Order 5-stage
Performance	1.57 DMIPS/MHz	1.72 DMIPS/MHz
Process	TSMC 40GPLUS	TSMC 40GPLUS
Area w/o Caches	0.27 mm <sup>2</sup>	0.14 mm <sup>2</sup>
Area with 16K Caches	0.53 mm <sup>2</sup>	0.39 mm <sup>2</sup>
Area Efficiency	2.96 DMIPS/MHz/mm <sup>2</sup>	4.41 DMIPS/MHz/mm <sup>2</sup>
Frequency	>1GHz	>1GHz
Dynamic Power	<0.08 mW/MHz	0.034 mW/MHz

- PPA reporting conditions  
- 85% utilization, use Dhrystone for benchmark, frequency/power at TT 0.9V 25C, all regular VT transistors  
- 10% higher in DMIPS/MHz, 49% more area-efficient

 张国斌

## Rust 语言简介

- 使用所有权模型管理内存（同时兼顾安全和高性能）
- 所有权保证了不会出现二次释放问题
- 生命周期机制保证了不会出现裸指针
- 强大的类型系统（泛型，trait）

# Rust 语言与嵌入式

## Rust 语言非常适合嵌入式软件开发

- 极小的运行时 -> 性能
- 独特的内存管理 -> 安全（所有权模型 -> 外设抽象）
- （包管理工具，强大的宏） -> 生产效率
- 丰富的编译目标（RISC-V, Arm, MIPS）
- **异步支持**（最小的异步抽象库 `nb` 和内置的 `async/await` 语法）



# 寄存器抽象

tock 的寄存器抽象:

```
const UARTLITE_MMIO: usize = 0x4060_0000;
register_structs! {
    /// UartLite MMIO
    /// |offset|register|description|
    /// |---|---|---|
    /// |0h|Rx FIFO|receive data fifo|
    /// |04h|Tx FIFO|send data fifo|
    /// |08h|status reg|IP 核状态寄存器|
    /// |0ch|control reg|IP 核控制寄存器|
    pub UartLite {
        (0x00 => rx_fifo: ReadOnly<u32>),
        (0x04 => tx_fifo: ReadWrite<u32>),
        (0x08 => stat_reg: ReadOnly<u32, Status::Register>),
        (0x0c => ctrl_reg: ReadWrite<u32, Control::Register>),
        (0x10 => @END),
    },
}
```

# 寄存器抽象

宏生成的结构体：

```
#[repr(C)]  
struct UartLite {  
    rx_fifo: ReadOnly<u32>,  
    tx_fifo: ReadWrite<u32>,  
    stat_reg: ReadOnly<u32, Status::Register>,  
    ctrl_reg: ReadWrite<u32, Control::Register>  
}
```

# 对寄存器抽象进行封装

```
impl UartLite {
    pub fn new() -> &'static mut UartLite {
        unsafe { &mut *(UARTLITE_MMIO as *mut UartLite) }
    }
    pub fn init(&mut self) {
        self.ctrl_reg.write(Control::RST_TX.val(1));
        self.ctrl_reg.write(Control::RST_RX.val(1));
    }
    pub fn putchar(&mut self, ch: char) {
        while self.stat_reg.is_set(Status::TX_FULL) {}
        self.tx_fifo.set(ch as u32);
    }
    pub fn getchar(&self) -> Result<u8, ()> {
        match self.stat_reg.is_set(Status::RX_VALID) {
            true => Ok(self.rx_fifo.get() as u8),
            false => Err(()),
        }
    }
}
```

# Rust 语言与嵌入式

## Rust 语言非常适合嵌入式软件开发

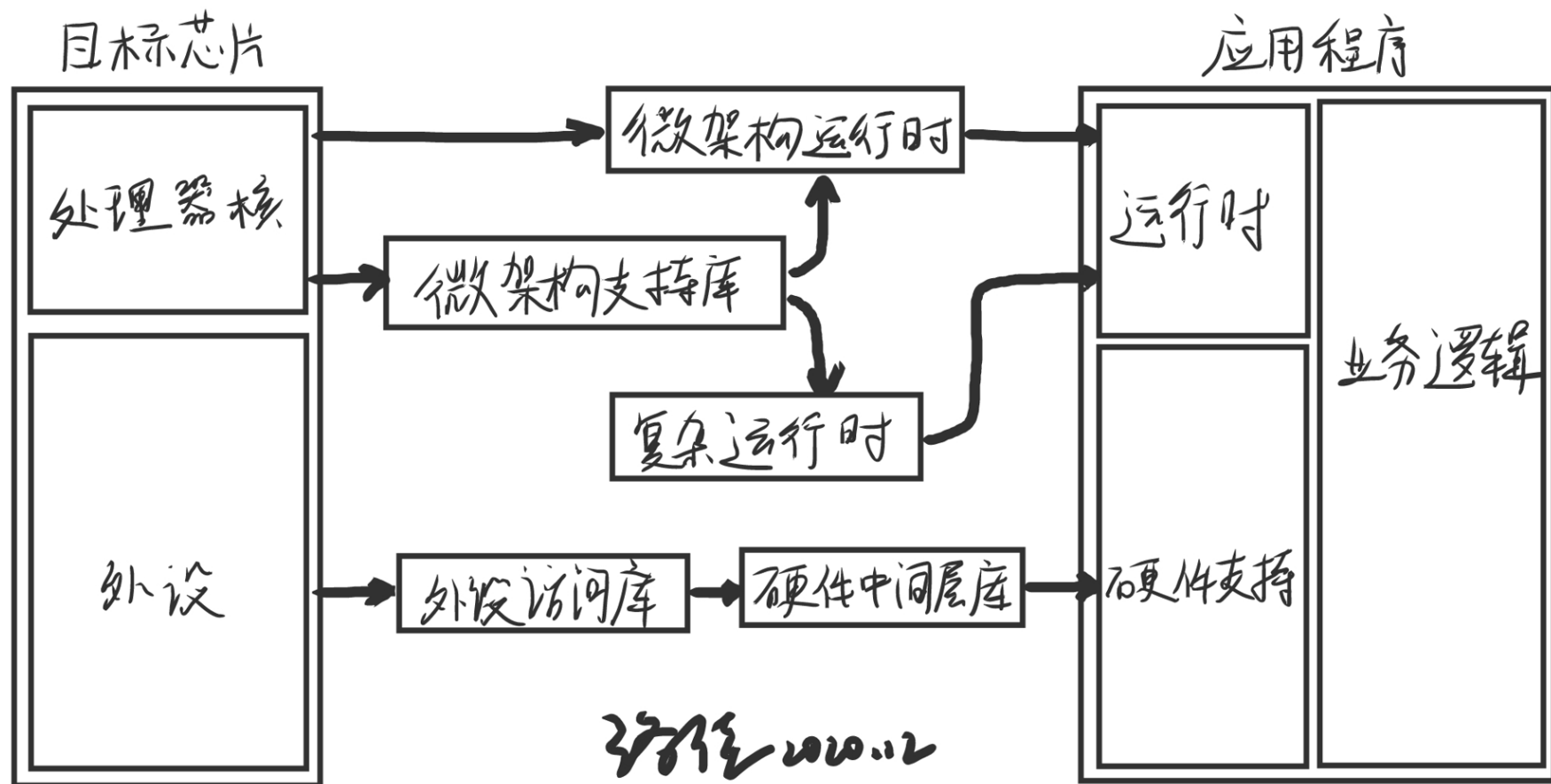
- 极小的运行时 -> 性能
- 独特的内存管理 -> 安全（所有权模型 -> 外设抽象）
- （包管理工具，强大的宏） -> 生产效率
- 丰富的编译目标（RISC-V, Arm, MIPS）
- **异步支持**（最小的异步抽象库 `nb` 和内置的 `async/await` 语法）

## Rust Async in Embedded

```
use embedded_hal::serial;
impl serial::Read<u8> for UartLite {
    type Error = Infallible;
    fn try_read(&mut self) -> nb::Result<u8, Self::Error> {
        match self.stat_reg.is_set(Status::RX_VALID) {
            true => Ok(self.rx_fifo.get() as u8),
            false => Err(nb::Error::WouldBlock),
        }
    }
}
```



## Rust语言的嵌入式生态 (2020年12月)



## 微架构运行时库

- 运行第一条**逻辑代码**之前的准备工作
- 通过过程宏提供函数入口
- 处理机器模式中断和异常

# 使用运行时库的一个例子

```
[no_std]
[no_main]

extern crate panic_halt;
use riscv_rt::entry;

// 使用 entry 标记逻辑代码入口点
// 在这之前 riscv-rt 已经做了一些准备工作
#[entry]
fn main() -> ! {
    // 逻辑代码
    loop { }
}
```



# 运行时库处理机器模式中断和异常

```
#[link_section = ".trap.rust"]
#[export_name = "_start_trap_rust"]
pub extern "C" fn start_trap_rust(trap_frame: *const TrapFrame) {
    extern "C" {
        fn ExceptionHandler(trap_frame: &TrapFrame);
        fn DefaultHandler();
    }
    unsafe {
        let cause = mcause::read();
        if cause.is_exception() {
            ExceptionHandler(&*trap_frame)
        } else {
            let code = cause.code();
            if code < __INTERRUPTS.len() {
                let h = &__INTERRUPTS[code];
                if h.reserved == 0 {
                    DefaultHandler();
                } else {
                    (h.handler)();
                }
            } else {
                DefaultHandler();
            }
        }
    }
}
```

# 运行时库处理机器模式中断和异常

```
#[no_mangle]
pub static __INTERRUPTS: [Vector; 12] = [
    Vector { handler: UserSoft },
    Vector {
        handler: SupervisorSoft,
    },
    Vector { reserved: 0 },
    Vector {
        handler: MachineSoft,
    },
    Vector { handler: UserTimer },
    Vector {
        handler: SupervisorTimer,
    },
    Vector { reserved: 0 },
    Vector {
        handler: MachineTimer,
    },
    Vector {
        handler: UserExternal,
    },
    Vector {
        handler: SupervisorExternal,
    },
    Vector { reserved: 0 },
    Vector {
        handler: MachineExternal,
    },
];
```

## embedded-hal 标准

- Rust 嵌入式社区统一的标准
- 这个库是对外设本身的抽象，提供一系列的 Trait,不涉及具体的实现
- 实现由各个开发版（SoC）的实现库来完成（stm32f30x-hal, k210-hal）
- 基于特定硬件的外设访问库来实现

## Rust Async in Embedded

```
use embedded_hal::serial;
impl serial::Read<u8> for UartLite {
    type Error = Infallible;
    fn try_read(&mut self) -> nb::Result<u8, Self::Error> {
        match self.stat_reg.is_set(Status::RX_VALID) {
            true => Ok(self.rx_fifo.get() as u8),
            false => Err(nb::Error::WouldBlock),
        }
    }
}
```

# Rust 语言与 RISC-V 嵌入式

## 以 RustSBI 为样本，对比竞品 OpenSBI 来分析

- RISC-V SBI 标准
- RustSBI
- 2000 多行代码量，对于 k210 平台的支持依赖于 k210-hal 和 k210-pac 这两个嵌入式支持库，得益于 Rust 嵌入式生态
- 对于 k210 的实现缺陷，通过 RustSBI 来弥补，得益于 SBI 标准的设计优越性
- RustSBI 模块化设计，对于某个特定的平台（硬件或模拟器），只要提供相应的 pac 库和 hal 库，即可很容易适配

## 在 k210 平台上模拟 `sfence.vma` 指令运行:

```
Trap::Exception(Exception::IllegalInstruction) => {
    let vaddr = mepc::read();
    let ins = unsafe { get_vaddr_u32(vaddr) };
    if ins & 0xFFFFF07F == 0xC0102073 { // rdttime instruction
        todo!()
    } else if ins & 0xFE007FFF == 0x12000073 { // sfence.vma instruction
        // k210 平台上不存在 sfence.vma 指令, 但存在 sfence.vm 指令, 因此我们在 SBI 里面模拟 sfence.vma 指令执行过程
        // 取出 satp 寄存器的值
        let satp_bits = satp::read().bits();
        // 获取根页表的 ppn
        let ppn = satp_bits & 0xFFF_FFFF_FFFF; // 43..0 PPN WARL
        // 写到 sptbr
        let sptbr_bits = ppn & 0x3F_FFFF_FFFF;
        unsafe { llvm_asm!("csw 0x180, $0"::"r"(sptbr_bits)) }; // write to sptbr
        // enable paging (in v1.9.1, mstatus: | 28..24 VM[4:0] WARL | ... )
        let mut mstatus_bits: usize;
        unsafe { llvm_asm!("csrr $0, mstatus"::"r"(mstatus_bits)) };
        mstatus_bits &= !0x1F00_0000;
        mstatus_bits |= 9 << 24;
        unsafe { llvm_asm!("csw mstatus, $0"::"r"(mstatus_bits)) };
        // 模拟 sfence.vma 指令
        unsafe { llvm_asm!(".word 0x10400073") }; // sfence.vm x0
        mepc::write(mepc::read().wrapping_add(4)); // skip current instruction
    } else {
        panic!("invalid instruction! mepc: {:016x?}, instruction: {:08x?}", mepc::read(), ins);
    }
}
```

## RustSBI

k210 平台没有 S 态外部中断，通过 SBI 来解决这个问题：

- 通过一个 SBI 调用，让操作系统内核传一个 S 态外部中断处理函数指针给 SBI
- SBI 收到 M 态外部中断，跳转到这个函数地址去运行
- 不是很优美的处理方法：该函数运行在 M 态

## RustSBI

```
Trap::Exception(Exception::SupervisorEnvCall) => {  
    if trap_frame.a7 == 0x0A000004 && trap_frame.a6 == 0x210 {  
        // trap_frame.a0 是操作系统内核传过来的函数指针  
        unsafe { DEVINTRENTY = trap_frame.a0; }  
        // enable mext  
        unsafe { mie::set_mext(); }  
        // return values  
        trap_frame.a0 = 0; // SbiRet::error = SBI_SUCCESS  
        trap_frame.a1 = 0; // SbiRet::value = 0  
    } else {  
        todo!()  
    }  
    mepc::write(mepc::read().wrapping_add(4));  
}
```



# RustSBI

```
Trap::Interrupt(Interrupt::MachineExternal) => {  
    // RustSBI 收到 M 态外部中断  
    unsafe {  
        let mut mstatus: usize;  
        llvm_asm!("csrr $0, mstatus" : "=r"(mstatus) ::: "volatile");  
        mstatus |= 1 << 17;  
        let mpp = (mstatus >> 11) & 3;  
        mstatus = mstatus & !(3 << 11);  
        mstatus |= 1 << 11;  
        llvm_asm!("csw mstatus, $0" :: "r"(mstatus) :: "volatile");  
        fn devintr() {  
            unsafe {  
                // 跳转到 DEVINTRETRY 中运行  
                llvm_asm!("jalr 0($0)" :: "r"(DEVINTRETRY) : "ra" : "volatile");  
            }  
        }  
        devintr();  
        mstatus = mstatus & !(3 << 11);  
        mstatus |= mpp << 11;  
        mstatus -= 1 << 17;  
        llvm_asm!("csw mstatus, $0" :: "r"(mstatus) :: "volatile");  
    }  
}
```

**谢谢各位**