

YOUR LOGO

2025 基于Rust语言的组件 化系统驱动



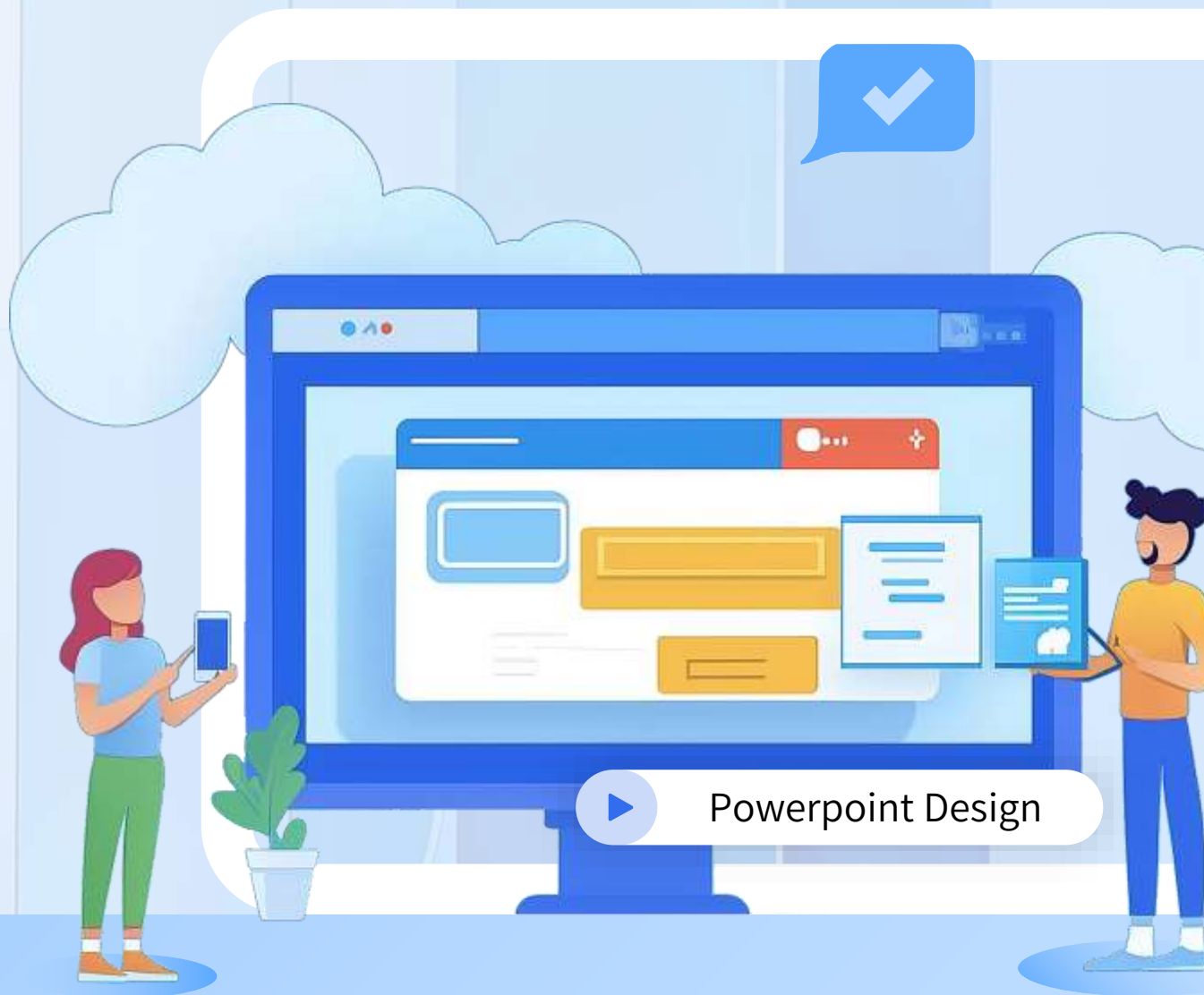
PowerPoint Design



主讲人：郑龙兵



时间：2025.6.11



目录

CONTENTS



01 研究背景与意义

02 核心技术概述

03 系统整体架构

04 系统启动流程

05 寄存器访问层设计

06 系统实现

07 致谢

YOUR LOGO

Part 01

研究背景与意义

Powerpoint Design



PowerPoint Design



边缘计算驱动需求

安全性需求

边缘计算设备部署广泛，面临网络攻击风险。传统C语言易因内存错误引发安全漏洞。

Rust语言所有权机制可以杜绝内存越界、数据竞争问题，提升边缘计算设备安全性。



效率需求

边缘计算需快速处理数据，传统驱动开发复杂、调试困难。

Rust组件化设计降低开发复杂度，提高开发效率，缩短产品上市周期。



Kendryte K230应用场景

Kendryte K230是典型AI异构计算芯片，适用于智能安防、工业自动化等边缘计算场景。

为其开发组件化系统驱动，可充分发挥芯片性能，满足边缘计算需求。

Rust语言优势

内存安全

Rust所有权系统自动管理内存，避免手动内存管理导致的错误。

C语言需手动管理内存，易出现内存泄漏、野指针等问题。

错误处理

Rust使用Result/Option类型处理错误，强制开发者处理错误情况。

C语言依赖返回值和错误码，易被开发者忽略，导致潜在问题。

资源管理

Rust采用RAII模式，资源在作用域结束时自动释放。

C语言需手动释放资源，易因忘记释放导致资源泄露。

并发控制

Rust通过Send/Sync标记确保并发安全，避免数据竞争。

C语言依赖低级并发原语，开发者需手动管理并发，易出错。

组件化设计价值



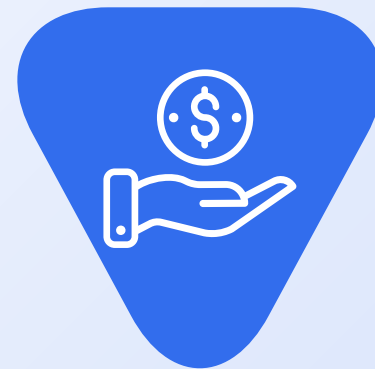
降低开发复杂度

组件化将系统分解为独立模块，各模块功能清晰，易于开发和维护。传统单体架构代码耦合度高，修改一处可能影响全局。



提高复用性

组件化模块可复用于不同项目，减少重复开发工作。单体架构代码复用性差，需重新开发类似功能。



便于团队协作

组件化允许不同团队独立开发模块，提高协作效率。单体架构团队需协调开发，易出现冲突。

YOUR LOGO

Part 02

核心技术概述

Powerpoint Design



PowerPoint Design



xtask模式

xtask构建任务以Rust代码方式统一实现和管理避免了多种构建工具混用带来的复杂性。

统一构建流程管理

xtask支持自定义构建任务，如固件加密、镜像打包等。满足项目特殊需求，提升项目灵活性。

自定义构建任务

xtask与Cargo集成，无缝融入Rust开发流程。无需额外工具，降低开发门槛。

与Cargo集成

分层抽象架构

微架构包 (Microarchitecture Crate)

微架构包封装芯片底层架构细节，提供基础硬件接口。
为上层模块提供稳定硬件支持，隔离硬件差异。

外设访问包 (PAC)

外设访问包 (PAC) 提供外设寄存器映射，简化外设访问操作。开发者可通过PAC访问外设寄存器，实现外设控制与数据交互。

硬件抽象层 (HAL)

HAL基于PAC，提供通用外设操作接口。
抽象外设细节，方便上层应用开发。

板级支持包 (BSP)

BSP针对具体开发板，配置硬件资源和外设。
适配不同开发板，简化软件开发。

统一抽象层

embedded-hal框架

embedded-hal是Rust嵌入式硬件抽象层框架，定义通用外设接口。提高代码复用性，方便跨平台开发。

embedded-io框架

embedded-io框架规范流式字节输入输出操作。统一外设通信方式，简化应用开发。

框架的兼容性与扩展性

embedded-hal和embedded-io框架兼容多种硬件平台。支持扩展新外设接口，适应不同开发需求。

YOUR LOGO

Part 03

系统整体架构

Powerpoint Design



PowerPoint Design



四大核心模块

01

kendryte-hal: 硬件抽象层

kendryte-hal核心目标在于将底层硬件的复杂性有效屏蔽。开发者可以通过统一的外设驱动接口访问各类硬件资源，而无需关心底层寄存器和硬件细节。

02

kendryte-rt: 运行时支持

kendryte-rt提供了系统初始化、启动流程管理、时钟配置以及外设初始化准备等低层任务。

03

examples: 示例代码

examples提供多种示例代码，展示系统功能和开发方法。帮助开发者快速上手，提高开发效率。

04

xtask: 构建工具

xtask负责项目构建、固件加密、镜像打包等任务。统一构建流程，简化开发配置。

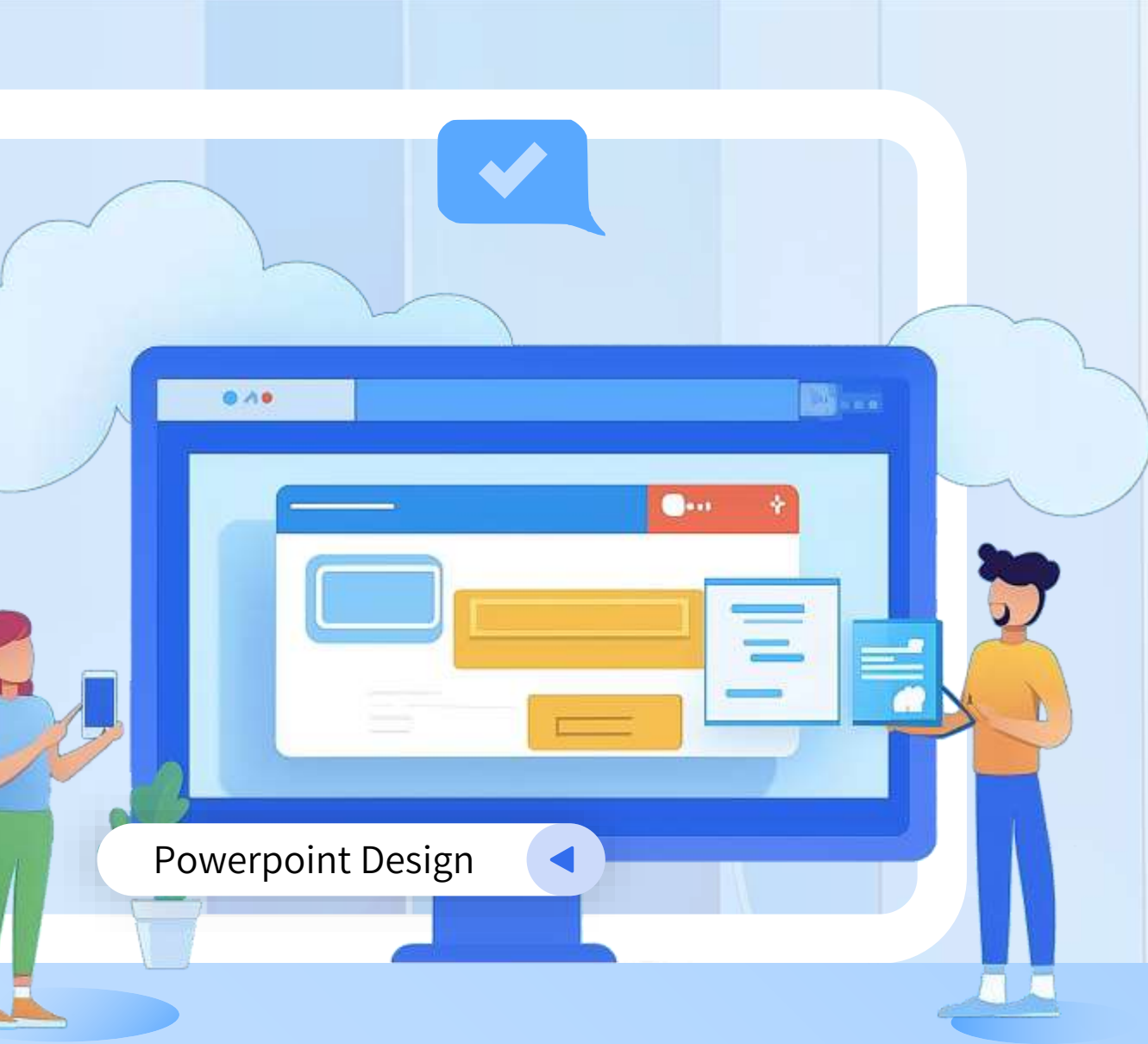
YOUR LOGO

Part 04

系统启动流程

Powerpoint Design

PowerPoint Design



链接脚本配置

内存布局规划

链接脚本定义系统内存布局，分配代码段、数据段、堆栈等区域。根据芯片特性合理规划内存，确保系统高效运行。

段分配与对齐

链接脚本指定各段的起始地址和对齐方式。
优化内存访问效率，减少内存碎片。

```
1 OUTPUT_ARCH(riscv)
2 ENTRY(_start)
3 MEMORY {
4     SPL : ORIGIN = 0x80300000, LENGTH = 0x100000
5 }
6 SECTIONS
7 {
8     .text : ALIGN(4) {
9         stext = .;
10        KEEP(*(.text.entry))
11        *(.text .text.*)
12        . = ALIGN(4);
13        etext = .;
14    } > SPL
15    .rodata : ALIGN(4) {
16        srodata = .;
17        *(.rodata .rodata.*)
18        *(.srodata .srodata.*)
19        . = ALIGN(4);
20        erodata = .;
21    } > SPL
22    .data : ALIGN(4) {
23        sdata = .;
24        *(.data .data.*)
25        *(.sdata .sdata.*)
26        . = ALIGN(4);
27        edata = .;
28    } > SPL
29    sidata = LOADADDR(.data);
30    .bss (NOLOAD) : ALIGN(4) {
31        *(.bss.uninit)
32        sbss = .;
33        *(.bss .bss.*)
34        *(.sbss .sbss.*)
35        ebss = .;
36    } > SPL
37    /DISCARD/ : {
38        *(.eh_frame)
39    }
40 }
```

汇编启动阶段

中断关闭与栈配置

启动时关闭中断，配置栈空间，确保系统初始化安全。

防止中断干扰初始化过程，避免潜在问题。

BSS段清零

将BSS段清零，初始化未初始化的全局变量。

确保变量初始值正确，避免未定义行为。

```
汇编代码

1  #[cfg(all(feature = "k230"))]
2  #[unsafe(naked)]
3  #[unsafe(link_section = ".text.entry")]
4  #[unsafe(export_name = "_start")]
5  unsafe extern "C" fn start() -> ! {
6      use crate::arch::rvi::Stack;
7      use crate::soc::main;
8      const STACK_SIZE: usize = 32 * 1024;
9
10     #[unsafe(link_section = ".bss.uninit")]
11     static mut STACK: Stack<STACK_SIZE> = Stack([0; STACK_SIZE]);
12
13     core::arch::naked_asm!(
14         "csrw    mie, zero",
15         "la      sp, {stack}"
16         "li      t0, {stack_size}"
17         "add     sp, sp, t0",
18         "la      t1, sbss"
19         "la      t2, ebss"
20         "1: bgeu  t1, t2, 2f"
21         "sw      zero, 0(t1)"
22         "addi    t1, t1, 4"
23         "j       1b"
24         "2:",
25         "call    {main}",
26         "3: wfi"
27         "j       3b",
28
29         stack      = sym STACK,
30         stack_size = const STACK_SIZE,
31         main       = sym main,
32     )
33 }
34 }
```


外设初始化准备

外设配置

配置外设参数，如时钟、中断等。
确保外设正常工作，满足系统需求。

外设实例创建

根据硬件配置，创建外设实例。
为外设操作提供基础支持。

```
1  #[allow(unused)]
2  #[doc(hidden)]
3  #[inline(always)]
4  pub fn __rom_init_params() -> (Peripherals, Clocks) {
5      let iomux = unsafe { iomux::Instance::transmute_at(0x9110_5000) };
6      let peripherals = Peripherals {
7          iomux: pad::pads::Pads::new(iomux.inner_mut()),
8          gpio0: unsafe { gpio::Instance::<0>::transmute_at(0x9140_B000) },
9          gpio1: unsafe { gpio::Instance::<1>::transmute_at(0x9140_C000) },
10         uart0: unsafe { uart::Instance::<0>::transmute_at(0x9140_0000) },
11         uart1: unsafe { uart::Instance::<1>::transmute_at(0x9140_1000) },
12         uart2: unsafe { uart::Instance::<2>::transmute_at(0x9140_2000) },
13         uart3: unsafe { uart::Instance::<3>::transmute_at(0x9140_3000) },
14         uart4: unsafe { uart::Instance::<4>::transmute_at(0x9140_4000) },
15     };
16     (peripherals, Clocks)
17 }
```


应用程序执行

entry宏定义

使用entry宏定义程序入口，简化入口函数编写。
提高代码可读性和可维护性。

主函数调用

调用主函数，开始应用程序执行。
主函数是应用程序的核心，负责业务逻辑处理。

应用程序代码

```
1  #![no_std]
2  #![no_main]
3
4  use kendryte_hal::gpio::{Output, PinState,
5                             StatefulOutputPin};
6  use kendryte_rt::{Clocks, Peripherals, entry};
7  use kendryte_hal::iomux::Strength;
8  use panic_halt as _;
9
10 #[entry]
11 fn main(p: Peripherals, _c: Clocks) -> ! {
12     let mut led = Output::new(p.gpio0, p.iomux.io19,
13                               PinState::High, Strength::_7);
14     loop {
15         led.toggle().ok();
16         riscv::asm::delay(10_000_000);
17     }
18 }
```

YOUR LOGO

Part 05

寄存器访问层设计

Powerpoint Design



PowerPoint Design



技术选型分析



原始unsafe与volatile内存访问

原始unsafe访问内存，需手动管理内存安全。
volatile修饰内存地址，防止编译器优化，但易出错。



MMIO结构体映射

MMIO结构体映射内存地址，提供面向对象的访问方式。
提高代码可读性，但需手动管理内存安全。



volatile-register库手动构建

使用volatile-register库手动构建寄存器访问接口。
提供安全的寄存器访问，但需手动编写代码。



svd2rust自动化生成

svd2rust根据芯片SVD文件自动生成寄存器访问代码。
提高开发效率，但生成代码可能需手动调整。

选择volatile-register手写方案的原因



原因

在本项目中，由于不适用SVD文件，我们无法在Kendryte芯片上应用svd2rust等自动生成工具。经过评估，选择使用volatile-register手写方案来实现寄存器访问层。这种方案虽然需要手动编写代码，但能提供基本的安全保证同时保持代码的可维护性。

```
volatile-register手写方案

1  #[repr(C)]
2  pub struct RegisterBlock {
3      pub(crate) pads: [RW<Pad>; 64],
4  }
5  #[bitenum(u1, exhaustive = true)]
6  #[derive(Debug, PartialEq, Eq)]
7  pub enum SlewRate {
8      Fast = 0b0,
9      Slow = 0b1,
10 }
11 #[bitenum(u4, exhaustive = true)]
12 #[derive(Debug, PartialEq, Eq)]
13 pub enum Strength {
14     _0 = 0b0000,
15     _1 = 0b0001,
16     _2 = 0b0010,
17     _3 = 0b0011,
18     // ... 省略
19 }
20 #[bitfield(u32)]
21 pub struct Pad {
22     #[bit(31, r)]
23     pub data_input: u1,
24     #[bits(11..=13, rw)]
25     pub function_select: u3,
26     #[bit(10, rw)]
27     pub slew_rate: SlewRate,
28     #[bit(8, rw)]
29     pub input_enable: bool,
30     #[bit(7, rw)]
31     pub output_enable: bool,
32     #[bit(6, rw)]
33     pub pull_up_enable: bool,
34     #[bit(5, rw)]
35     pub pull_down_enable: bool,
36     #[bits(1..=4, rw)]
37     pub drive_strength: Strength,
38     #[bit(0, rw)]
39     pub schmitt_trigger_enable: bool,
40 }
```

YOUR LOGO

Part 06

系统实现

Powerpoint Design



PowerPoint Design



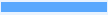
外设所有权系统



IO引脚独占性保障

外设所有权系统确保IO引脚独占性，
避免资源冲突。

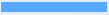
提高系统稳定性，减少潜在问题。



外设实例资源冲突防止

防止外设实例资源冲突，确保外设正
常工作。

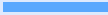
提高资源利用率，避免资源浪费。



IO引脚与外设实例对应关系验证

验证IO引脚与外设实例对应关系，确
保配置正确。

避免错误配置导致的系统故障。



IO引脚独占性设计

对IO引脚进行建模，主要用于实现IO引脚的独占性。

其主要有两个结构体：Pad 和 FlexPad。其中，Pad 结构体带有常量泛型参数 N，用以确定对应的IO引脚实例；而 FlexPad 实际上是“类型擦除”后的 Pad，即去除了常量泛型参数 N，所有IO引脚都可通过 FlexPad 进行抽象访问。Pad 可以使用 as_mut_flex_pad 方法转化为 FlexPad 的独占引用，其中 Pad 与 FlexPad 的生命周期绑定在一起。这样设计可以实现 Pad 与 FlexPad 之间是互斥关系，且 FlexPad 生命周期结束后可以继续使用 Pad。

Pads 结构体则持有 64 个 IO 引脚的独占引用，确保同一时间每个引脚只会被一个功能独占，然后供上层应用程序使用。

```
IO引脚独占性设计

1  #[repr(transparent)]
2  pub struct Pad<const N: usize> {
3      inner: RW<crate::iomux::Pad>,
4  }
5  impl<const N: usize> Pad<N> {
6      pub fn as_flexible_mut(&mut self) -> &mut FlexPad {
7          unsafe { core::mem::transmute(self) }
8      }
9  }
10 #[repr(transparent)]
11 pub struct FlexPad {
12     inner: RW<crate::iomux::Pad>,
13 }
14 pub struct Pads {
15     pub io0: &'static mut Pad<0>,
16     pub io1: &'static mut Pad<1>,
17     pub io2: &'static mut Pad<2>,
18     pub io3: &'static mut Pad<3>,
19     pub io4: &'static mut Pad<4>,
20     pub io5: &'static mut Pad<5>,
21     pub io6: &'static mut Pad<6>,
22     pub io7: &'static mut Pad<7>,
23     // ... 省略后续56的类似的io
24 }
25
```

IO引脚独占性测试

本测试是对IO引脚独占性的测试，主要通过多次使用同一个IO引脚来达到竞争条件。在达成竞争条件下，编译器按照预期给出了多次使用可变引用到错误，这验证了IO引脚独占性实现的正确性。

```
error[E0499]: cannot borrow `*p.iomux.io19` as mutable more than once at a time
--> examples/peripherals/gpio-blinky-demo/src/main.rs:13:41
|
12 | let mut led0 = Output::new(p.gpio0, p.iomux.io19, PinState::High, Drive::Level7);
|                                     ----- first mutable borrow occurs here
13 | let mut led1 = Output::new(p.gpio0, p.iomux.io19, PinState::High, Drive::Level7);
|                                     ^^^^^^^^^^^^^ second mutable borrow occurs here
14 | loop {
15 |     led0.toggle().ok();
|     ---- first borrow later used here

For more information about this error, try `rustc --explain E0499`.
```

引脚独占性测试

```
IO引脚独占性测试

1  #[entry]
2  fn main(p: Peripherals, _c: Clocks) -> ! {
3      let mut led0 = Output::new(p.gpio0,
p.iomux.io19, PinState::High, Drive::Level7); // 第一次
使用io19
4      let mut led1 = Output::new(p.gpio0,
p.iomux.io19, PinState::High, Drive::Level7); // 再次使
用io19
5      loop {
6          led0.toggle().ok();
7          led1.toggle().ok();
8          riscv::asm::delay(10_000_000);
9      }
10 }
```


外设实例设计

对外设实例进行建模，主要用于防止资源冲突。外设实例建模的原则依赖于IO复用的互斥或独立关系。以 GPIO 为例，每个IO引脚在复用GPIO功能之间是独立的，因此GPIO组的建模采用共享引用（&）。而每个UART实例的发送和接收等功能都有一个或以上的引脚可以复用，而对应特定功能复用的引脚之间是互斥关系，因此建模采用独占引用（&mut），以保证资源的独占性进而防止资源冲突。

```
1 pub trait SharedInstance {
2     type Target;
3     unsafe fn transmute_at(addr: usize) -> &'static Self
4     where
5         Self: Sized,
6         Self::Target: Sized,
7     {
8         unsafe { &*(addr as *const Self::Target as *const Self) }
9     }
10
11     fn inner(&self) -> &'static Self::Target
12     where
13         Self: Sized,
14         Self::Target: Sized,
15     {
16         unsafe { &*(self as *const Self as *const Self::Target) }
17     }
18 }
19 pub trait ExclusiveInstance {
20     type Target;
21     unsafe fn transmute_at(addr: usize) -> &'static mut Self
22     where
23         Self: Sized,
24         Self::Target: Sized,
25     {
26         unsafe { &mut *(addr as *mut Self::Target as *mut Self) }
27     }
28
29     fn inner(&self) -> &'static Self::Target
30     where
31         Self: Sized,
32         Self::Target: Sized,
33     {
34         unsafe { &*(self as *const Self as *const Self::Target) }
35     }
36
37     fn inner_mut(&mut self) -> &'static mut Self::Target
38     where
39         Self: Sized,
40         Self::Target: Sized,
41     {
42         unsafe { &mut *(self as *mut Self as *mut Self::Target) }
43     }
44 }
```

IO引脚与外设实例间的对应关系

接着对IO引脚与外设实例间的对应关系进行建模，主要用于在编译期校验硬件实例与IO引脚对应关系的正确性。对于UART功能，分别定义了UartSoutFunction、UartSinFunction等trait。这些trait使用泛型常量N来区分不同的UART实例，每个trait提供对应的功能设置方法。

IO引脚与外设实例间的对应关系

```
1 pub trait UartSoutFunction<const N: usize> {  
2     fn set_uart_sout_function(&mut self) -> &mut Self;  
3 }  
4 pub trait UartSinFunction<const N: usize> {  
5     fn set_uart_sin_function(&mut self) -> &mut Self;  
6 }
```

IO引脚与外设实例间的对应关系

```
1 impl<'tx, 'rx, 'd> BlockingUart<'tx, 'rx, 'd> {  
2     pub fn new<const UART_NUM: usize, const  
3         TX_PAD_NUM: usize, const RX_PAD_NUM: usize>(  
4         instance: &'d mut Instance<UART_NUM>,  
5         tx: Option<&'tx mut Pad<TX_PAD_NUM>>,  
6         rx: Option<&'rx mut Pad<RX_PAD_NUM>>,  
7         config: Config,  
8         clocks: Clocks,  
9     ) -> Self  
10     where  
11         Pad<TX_PAD_NUM>: UartSoutFunction<UART_NUM>,  
12         Pad<RX_PAD_NUM>: UartSinFunction<UART_NUM>,  
13     {  
14         // ... 省略  
15     }
```

IO引脚与外设实例对应关系测试

本测试是对IO引脚与外设实例对应关系的测试，主要通过给UART传递错误的TX引脚与RX引脚，。编译器不仅按照预期给出了new函数中TX引脚与RX引脚对应参数类型不正确，还给出了正确的参数的类型，这验证了IO引脚与外设实例对应关系实现的正确性。

```
IO引脚与外设实例对应关系测试

1 #[entry]
2 fn main(p: Peripherals, c: Clocks) -> ! {
3     // 传递错误的TX引脚与RX引脚
4     let mut serial0 = BlockingUart::new(p.uart0,
5     p.iomux.io48, p.iomux.io49, Config::new(), c);
6     loop {
7         riscv::asm::delay(10_000_000);
8     }
9 }
```

```
error[E0308]: arguments to this function are incorrect
--> examples/peripherals/uart-demo/src/main.rs:10:23
10 |     let mut serial0 = BlockingUart::new(p.uart0, p.iomux.io48, p.iomux.io49, Config::new(), c);
    |                                     ^^^^^^^^^^^^^^^^^
note: expected '38', found '48'
--> examples/peripherals/uart-demo/src/main.rs:10:50
10 |     let mut serial0 = BlockingUart::new(p.uart0, p.iomux.io48, p.iomux.io49, Config::new(), c);
    |                                     ^^^^^^^^^^^^^^^^^
= note: expected mutable reference '&mut kendryte_hal::pad::Pad<38>'
         found mutable reference '&'static mut kendryte_hal::pad::Pad<48>'
note: expected '39', found '49'
--> examples/peripherals/uart-demo/src/main.rs:10:64
10 |     let mut serial0 = BlockingUart::new(p.uart0, p.iomux.io48, p.iomux.io49, Config::new(), c);
    |                                     ^^^^^^^^^^^^^^^^^
= note: expected mutable reference '&mut kendryte_hal::pad::Pad<39>'
         found mutable reference '&'static mut kendryte_hal::pad::Pad<49>'
note: associated function defined here
--> /Users/zhenglongbing/Rust/kendryte-hal/kendryte-hal/src/uart/blocking.rs:88:12
88 |     pub fn new<const UART_NUM: usize, const TX_PAD_NUM: usize, const RX_PAD_NUM: usize>(
    |         ^^^
For more information about this error, try `rustc --explain E0308`.
```

引脚与外设实例对应关系测试

GPIO模块

01

输入/输出引脚抽象

GPIO模块抽象输入/输出引脚，提供统一接口。
简化引脚操作，方便应用开发。

02

embedded-hal特征实现

实现embedded-hal的GPIO特征，
确保代码复用性。
提高代码通用性，方便跨平台开发。

03

引脚配置与状态管理

提供引脚配置接口，管理引脚状态。
确保引脚操作正确，避免潜在问题。

GPIO模块核心代码

GPIO模块的核心架构有着三个关键组成部分：GPIO实例（Instance）、输入引脚（Input）及输出引脚（Output）。

其中GPIO实例利用常量泛型参数N区分不同的GPIO组来持有对应的寄存器块引用，并且依靠常量泛型参数N来在编译期校验硬件实例的正确性。

输入与输出引脚结构体分别对应GPIO的输入与输出功能，其包含了寄存器块引用、PAD配置及端口引脚信息，并且通过包装FlexPad的独占引用确保对引脚资源的安全访问。

02

```
GPIO

1  #[repr(transparent)]
2  pub struct Instance<const N: usize> {
3      inner: RegisterBlock,
4  }
5  impl<const N: usize> SharedInstance for Instance<N> {
6      type Target = RegisterBlock;
7  }
8  /// Represents a GPIO input pin.
9  pub struct Input<'pad> {
10     inner: &'static RegisterBlock,
11     pad: &'pad mut FlexPad,
12     port: Port,
13     pin_num: usize,
14 }
15 /// Represents a GPIO output pin.
16 pub struct Output<'pad> {
17     inner: &'static RegisterBlock,
18     pad: &'pad mut FlexPad,
19     port: Port,
20     pin_num: usize,
21 }
```

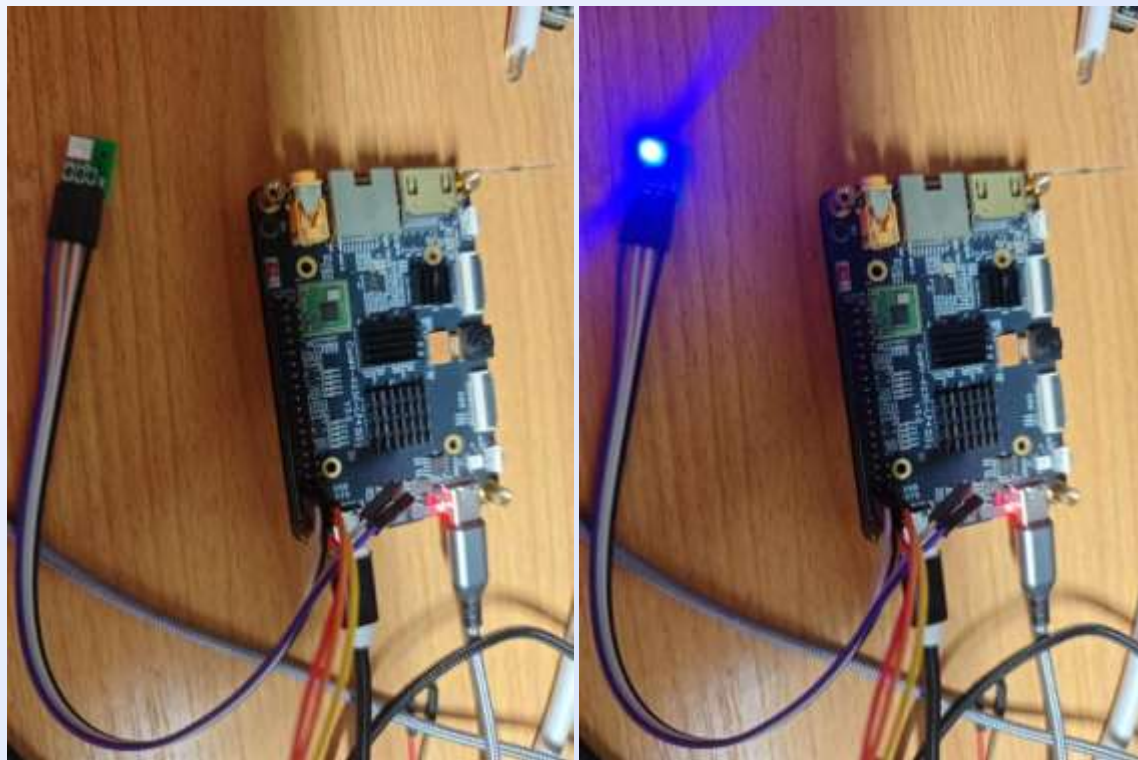
LED控制测试

该测试通过如下步骤进行：

- (1) 配置指定GPIO引脚为输出模式
- (2) 设置适当的驱动强度参数
- (3) 执行周期性的引脚状态切换操作

02

```
led
1 #[entry]
2 fn main(p: Peripherals, _c: Clocks) -> ! {
3     let mut led = Output::new(p.gpio0, p.iomux.io19,
4     PinState::High, Drive::Level7);
5     loop {
6         led.toggle().ok();
7         riscv::asm::delay(10_000_000);
8     }
9 }
```



LED控制测试结果

按键交互测试

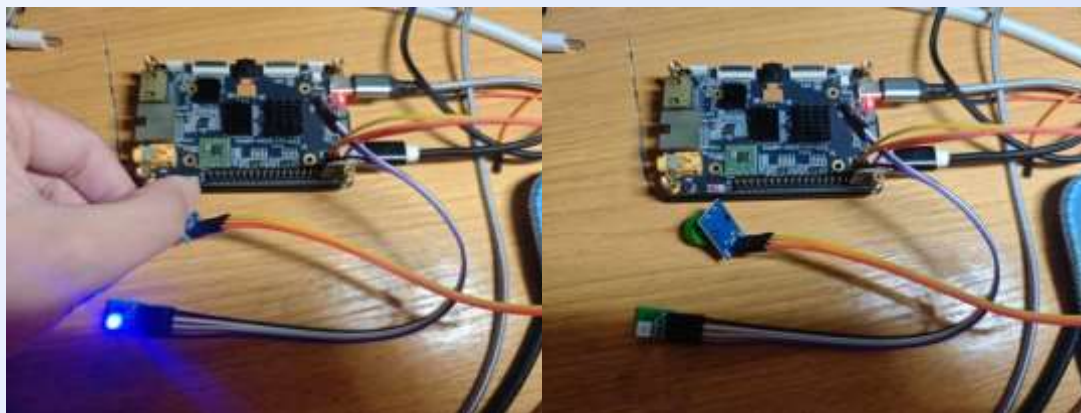
该测试通过如下步骤进行：

- (1) 配置LED控制引脚为输出模式
- (2) 配置按键引脚为输入模式并启用下拉电阻
- (3) 通过轮询按键状态来控制LED亮灭

02

```
按键交互

1  #[entry]
2  fn main(p: Peripherals, _c: Clocks) -> ! {
3      let mut led = Output::new(p.gpio0, p.iomux.io19,
4      PinState::High, Drive::Level7);
5      let mut button = Input::new(p.gpio0,
6      p.iomux.io20, Pull::Down);
7      loop {
8          match button.pin_state() {
9              PinState::High => led.set_high().ok(),
10             PinState::Low => led.set_low().ok(),
11         };
12         riscv::asm::delay(100_000);
13     }
14 }
```



按键交互测试结果

UART模块

01

阻塞式通信实现

UART模块实现阻塞式通信，确保数据传输可靠。

简化通信编程，提高开发效率。

02

配置管理与参数设置

提供UART配置接口，支持波特率、数据位等参数设置。

满足不同通信需求，提高系统灵活性。

03

数据收发管理

管理UART数据收发，确保数据完整传输。
提高通信效率，减少数据丢失。

UART模块核心代码

UART模块包含三个主要组成部分：UART实例（Instance）、阻塞式UART（BlockingUart）以及独立的发送与接收组件（BlockingUartTx和BlockingUartRx）。

UART实例利用常量泛型参数N区分不同的UART来持有对应的寄存器块引用，并且依靠常量泛型参数N来在编译期校验硬件实例的正确性。

BlockingUart封装了完整的收发功能，而BlockingUartTx和BlockingUartRx分别负责发送和接收操作。BlockingUart可以通过split方法消耗自身来得到BlockingUartTx和BlockingUartRx，以达到将UART的发送与接收功能分割成独立的两个功能的目的。

02

```
UART
1  #[repr(transparent)]
2  pub struct Instance<const N: usize> {
3      inner: RegisterBlock,
4  }
5  impl<const N: usize> ExclusiveInstance for
    Instance<N> {
6      type Target = RegisterBlock;
7  }
8  /// A wrapper struct for UART that provides blocking
    operations.
9  pub struct BlockingUart<'tx, 'rx, 'd> {
10     inner: &'static RegisterBlock,
11     tx: Option<BlockingUartTx<'tx, 'd>>,
12     rx: Option<BlockingUartRx<'rx, 'd>>,
13     _marker: PhantomData<&'d ()>,
14 }
```

UART通信测试

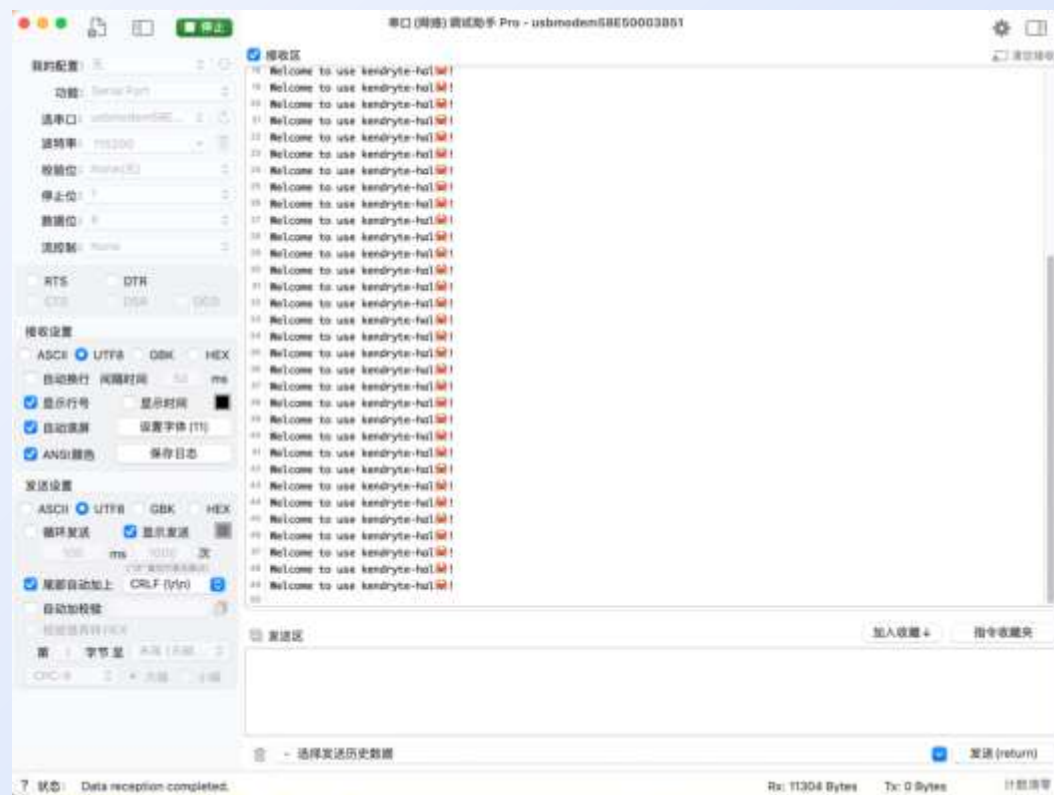
该测试通过如下步骤进行：

- (1) 同时初始化两个UART实例
- (2) 配置相关通信参数
- (3) 执行周期性的数据发送操作

02

```
UART通信

1  #[entry]
2  fn main(p: Peripherals, c: Clocks) -> ! {
3      let mut serial0 = BlockingUart::new(p.uart0, p.iomux.io38,
4      p.iomux.io39, Config::new(), c);
5      let mut serial3 = BlockingUart::new(p.uart3, p.iomux.io50,
6      p.iomux.io51, Config::new(), c);
7      loop {
8          writeln!(serial0, "Welcome to use kendryte-hal🐞!").ok();
9          writeln!(serial3, "Welcome to use kendryte-hal🐞!").ok();
10         riscv::asm::delay(10_000_000);
11     }
12 }
```



UART通信测试结果

固件加密

无加密模式

提供无加密模式，满足简单应用场景。
简化开发流程，降低开发难度。

01

SM4-CBC+SM2加密

支持SM4- CBC加密算法和SM2
签名算法，保障固件安全。
符合国内安全标准，提高固件
安全性。

02

AES-GCM+RSA-2048加密

支持AES- GCM加密算法和RSA-
2048签名算法，满足国际安全
需求。
提高固件安全性，防止固件被
篡改。

03

镜像打包

格式标准化

将固件打包为标准化格式，便于烧录和部署。
提高固件兼容性，简化部署流程。

对齐要求

按照芯片要求对齐固件，确保固件正确加载。
避免固件加载错误，提高系统稳定性。

```
镜像打包

1 pub fn gen_image(firmware: &[u8], encryption:
  EncryptionType) -> XtaskResult<Vec<u8>> {
2     println!("----- Generating image -----");
3     let mut image = vec![0; 0x1000000];
4     image.extend(MAGIC.as_bytes());
5     println!("the magic is: {}", MAGIC);
6     match encryption {
7         EncryptionType::None =>
8         handle_none_encryption(&mut image, firmware)?,
9         EncryptionType::Sm4 =>
10        handle_sm4_encryption(&mut image, firmware)?,
11        EncryptionType::Aes =>
12        handle_aes_encryption(&mut image, firmware)?,
13    }
14    if image.len() % 512 != 0 {
15        let padding_size = 512 - image.len() % 512;
16        image.extend(vec![0; padding_size]);
17    }
18    Ok(image)
19 }
```

YOUR LOGO

Part 07

致谢

Powerpoint Design



PowerPoint Design



感谢实习导师与组长的悉心指导

在这个项目中，非常感谢我的实习导师蒋周奇和我的实习组长董庆，非常感谢他们在此项目中对我的帮助和鼓励，没有他们，就不会有现在这篇论文的出现。

1_21045334-郑龙兵-开题报告.docx
76.7KB

我说得不一定对，供参考

好，我看看

毕业论文-意见版.docx
1.59MB

来，我写了一些意见，请查收

郑龙兵_毕业论文-v3.0.docx
1.80MB

你看下一些论文建议

有问题再交流一下

加油

如果愿意的话，我可以看看答辩ppt

或者可以帮助你排练

YOUR LOGO

2025
谢谢大家



PowerPoint Design



主讲人：郑龙兵



时间：2025.6.11

