

## 课程内容勘误表

- 1. 第4章-嵌入式开发介绍
  - 1.1. 【勘误 4-1】
  - 1.2. 【勘误 4-2】
- 2. 第5章 RISC-V 汇编语言编程
  - 2.1. 【勘误 5-1】
  - 2.2. 【勘误 5-2】
  - 2.3. 【勘误 5-3】
- 3. 第6章-RVOS 介绍
  - 3.1. 【勘误 6-1】
- 4. 第7章 Hello RVOS
  - 4.1. 【勘误 7-1】
  - 4.2. 【勘误 7-2】
- 5. 第 8 章 内存管理
  - 5.1. 【勘误 8-1】
- 6. 第 9 章 上下文切换和协作式多任务
  - 6.1. 【勘误 9-1】
- 7. 第 10 章 Trap 和 Exception
  - 7.1. 【勘误 10-1】
  - 7.2. 【勘误 10-2】
- 8. 第 11 章 外部设备中断
  - 8.1. 【勘误 11-1】

# 1. 第4章-嵌入式开发介绍

---

## 1.1. 【勘误 4-1】

参考 [issue 描述](#)。视频 03:09 左右处的课件描述有错误。

错误的讲稿页视频截图（[视频 P6](#) 播放第 3 分钟左右处）：

交叉编译

ISCAS NIST

- 参与编译和运行的机器根据其角色可以分成以下三类：
  - 构建（build）系统：生成编译器可执行程序的计算机。
  - 主机（host）系统：运行编译器可执行程序，编译链接应用程序的计算机系统。
  - 目标（target）系统：运行应用程序的计算机系统。
- 根据 build/host/target 的不同组合我们可以得到如下的编译方式分类：
  - 本地（native）编译：build == host == target
  - 交叉（cross）编译：build == host != target

Navigation icons: back, forward, search, etc.

改正后正确的讲稿页截图：

交叉编译

ISCAS NIST

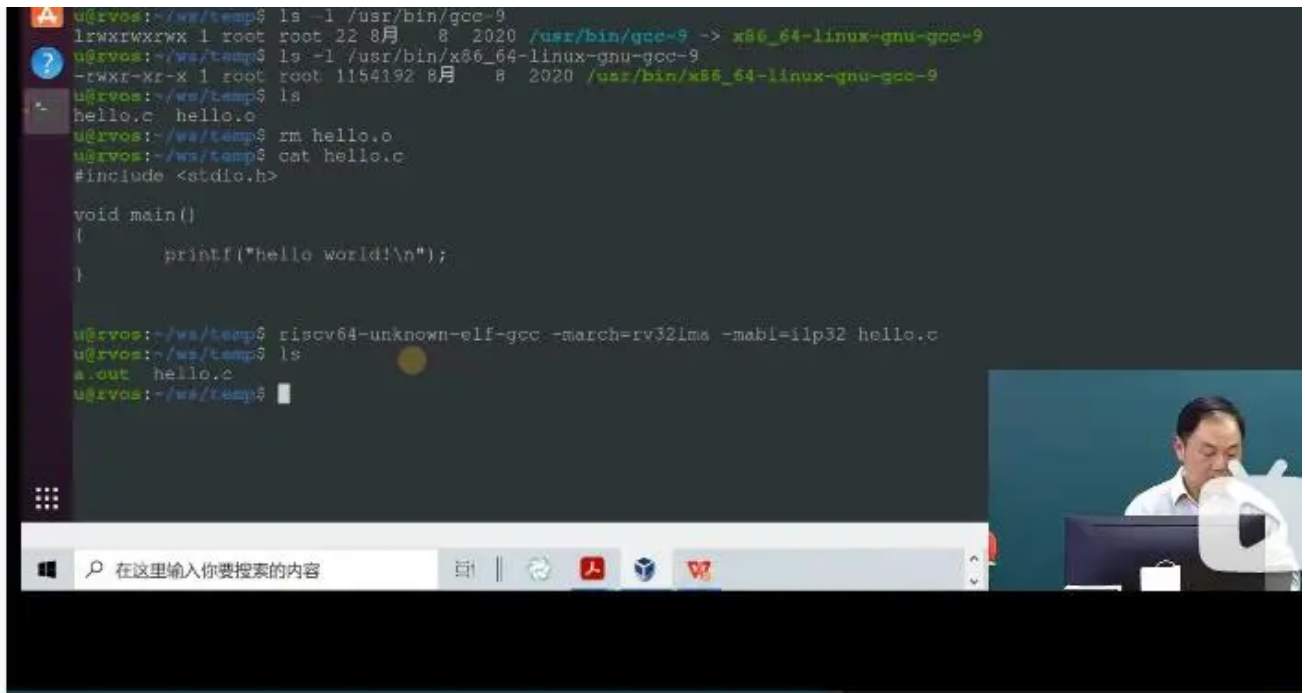
- 参与编译和运行的机器根据其角色可以分成以下三类：
  - 构建（build）系统：执行编译构建动作的计算机。
  - 主机（host）系统：运行 build 系统生成的可执行程序的计算机系统。
  - 目标（target）系统：特别地，当以上生成的可执行程序是 GCC 时，我们用 target 来描述用来运行 GCC 将生成的可执行程序的计算机系统。
- 根据 build/host/target 的不同组合我们可以得到如下的编译方式分类：
  - 本地（native）编译：build == host == target
  - 交叉（cross）编译：build == host != target

## 1.2. 【勘误 4-2】

参考 [issue 描述](#)。

很多同学跟着视频操作时编译碰到一个 stdio.h 找不到的错误。

part 6, 20分钟左右，当时上课时的确是用 riscv64-unknown-elf-gcc 编译了一个含有 #include <stdio.h> 的程序。如下图所示：



```
u@rvos:~/ws/temp$ ls -l /usr/bin/gcc-9
lrwxrwxrwx 1 root root 22 8月  8 2020 /usr/bin/gcc-9 -> x86_64-linux-gnu-gcc-9
u@rvos:~/ws/temp$ ls -l /usr/bin/x86_64-linux-gnu-gcc-9
-rwxr-xr-x 1 root root 1154192 8月  8 2020 /usr/bin/x86_64-linux-gnu-gcc-9
u@rvos:~/ws/temp$ ls
hello.c  hello.o
u@rvos:~/ws/temp$ rm hello.o
u@rvos:~/ws/temp$ cat hello.c
#include <stdio.h>

void main()
{
    printf("hello world!\n");
}

u@rvos:~/ws/temp$ riscv64-unknown-elf-gcc -march=rv32ima -mabi=ilp32 hello.c
u@rvos:~/ws/temp$ ls
a.out  hello.c
u@rvos:~/ws/temp$
```

解释一下原因：历史是这样的，最早我们提供了自己的工具链，包括 qemu，以 tar 包的形式，这个 tar 包里的 riscv64-unknown-elf-gcc 是来自 sifive 制作的一个版本，这个 gcc 是可以处理 `#include <stdio.h>` 的，原因我在上面简单分析过，和制作 gcc 过程中的配置有关。后来发现有些同学下载 tar 包以及自己安装有困难，后来发现如果是 ubuntu 20 就 apt 直接安装，编译我们的 rvos 仓库代码也是没有问题的，ubuntu 18 的我们还是提供了下载的 tar 自己安装。但可惜的后来发现 ubuntu 20 apt 自带的 riscv64-unknown-elf-gcc 是不能缺省搜索包含 `stdio.h` 的。造成了同学们跟着视频操作导致的问题。为此非常抱歉给大家带来一些混乱。

anyway，解决方法也很简单，对课程仓库代码没有影响，唯一的影响就是对视频中的那个示例，大家如果一定要尝试编译，请换成 ubuntu apt 安装的 riscv64-linux-gnu-gcc 即可。

## 2. 第5章 RISC-V 汇编语言编程

### 2.1. 【勘误 5-1】

视频“第5章-part7-RISC-V 汇编语言编程” 54:58 左右在讲解 `asm/code/cc_nested` 那个例子中对“jal square”这条语句解释说是“尾调用”，这个解释不准确，而且在本例子的场景下也不涉及“尾调用”的概念，比较恰当的解释应该是 **leaf call**，这里仅仅想表达 `aa_bb()` 这个函数调用了 `square()` 函数，而 `square()` 函数内部不会再调用其他的函数了。

### 2.2. 【勘误 5-2】

视频“第5章-part7-RISC-V 汇编语言编程” 26:30 左右在讲解“tail offset”时对 **尾调用** 的解释不对。正确的尾调用的意思是指 **A tailcall is a call to a function whose value is immediately returned.** 用 C 语言举个例子：

```
void func_A()
{
    .....
    func_B();
}
```

```
.....  
}  
  
void func_B()  
{  
    ...  
    return func_C();  
    ...  
}
```

这里 `return func_C();` 这样的代码生成对应的汇编语句就是一个 **尾调用(tail call)**。因为 `func_B()` 在调用 `func_C()` 后不需要返回 `func_B()` 函数自身继续下面的步骤，而是直接返回到更上一级 `func_B()` 被调用的地方，所以我们在汇编中调用 `func_C()` 时并不需要保存 `return address(x1)`，所以这也是 `tail offset` 中对应的 `jalr` 指令的第二个参数使用 `x0` 而不是 `x1` 的原因。

### 2.3. 【勘误 5-3】

ADDI 指令中立即数可以表达的数值范围应该为：[-2048, 2047]，课件上错误写成了 [-2048, 2047)。

## 3. 第6章-RVOS 介绍

---

### 3.1. 【勘误 6-1】

“课程项目简介”那一页的 git 仓库路径有误

错误的讲稿页视频截图（[视频 P15](#) 播放第 11 分钟左右处）：

#### 课程项目简介

ISCAS NIST

# RVOS

RVOS (<https://www.rt-thread.org/>) 是一个用于教学演示的操作系统内核。诞生于 2021 年。采用 BSD 2-Clause 许可证发布。

- 设计小巧，整个核心有效代码不超过 1000 行；
- 可读性强，易维护，绝大部分代码为 C 语言，很少部分采用汇编；
- 演示了简单的内存分配管理实现；
- 演示了可抢占多线程调度实现，线程调度采用轮转调度法；
- 演示了简单的任务互斥实现；
- 演示了软件定时器实现；
- 演示了系统调用实现（M 和 U 模式）；
- 支持 RV32；
- 支持 QEMU-virt 平台。

改正后正确的讲稿页截图：

## 课程项目简介

ISCAS NIST

# RVOS

RVOS (<https://github.com/plctlab/riscv-operating-system-mooc>) 是一个用于教学演示的操作系统内核。诞生于 2021 年。采用 BSD 2-Clause 许可证发布。

- 设计小巧，整个核心有效代码 ~ 1000 行；
- 可读性强，易维护，绝大部分代码为 C 语言，很少部分采用汇编；
- 演示了简单的内存分配管理实现；
- 演示了可抢占多线程调度实现，线程调度采用轮转调度法；
- 演示了简单的任务互斥实现；
- 演示了软件定时器实现；
- 演示了系统调用实现（M + U 模式）；
- 支持 RV32；
- 支持 QEMU-virt 平台。

## 4. 第7章 Hello RVOS

---

### 4.1. 【勘误 7-1】

参考 [issue 描述](#)。相关代码已经在 v0.8 版本中改正。

### 4.2. 【勘误 7-2】

参考 [issue 描述](#)。相关代码已经在 v0.9 版本中改正。

## 5. 第 8 章 内存管理

---

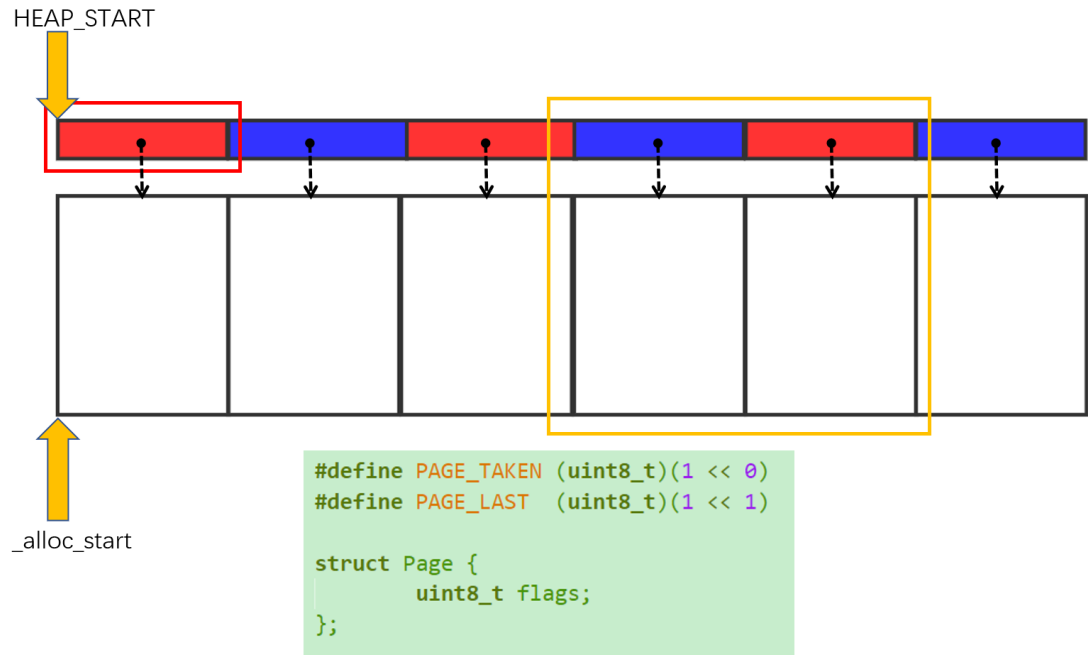
### 5.1. 【勘误 8-1】

"Page 描述符数据结构设计" 那一页的图片有误。

错误的讲稿页视频截图 (视频 P18 播放第 49 分钟左右处) :

Page 描述符数据结构设计

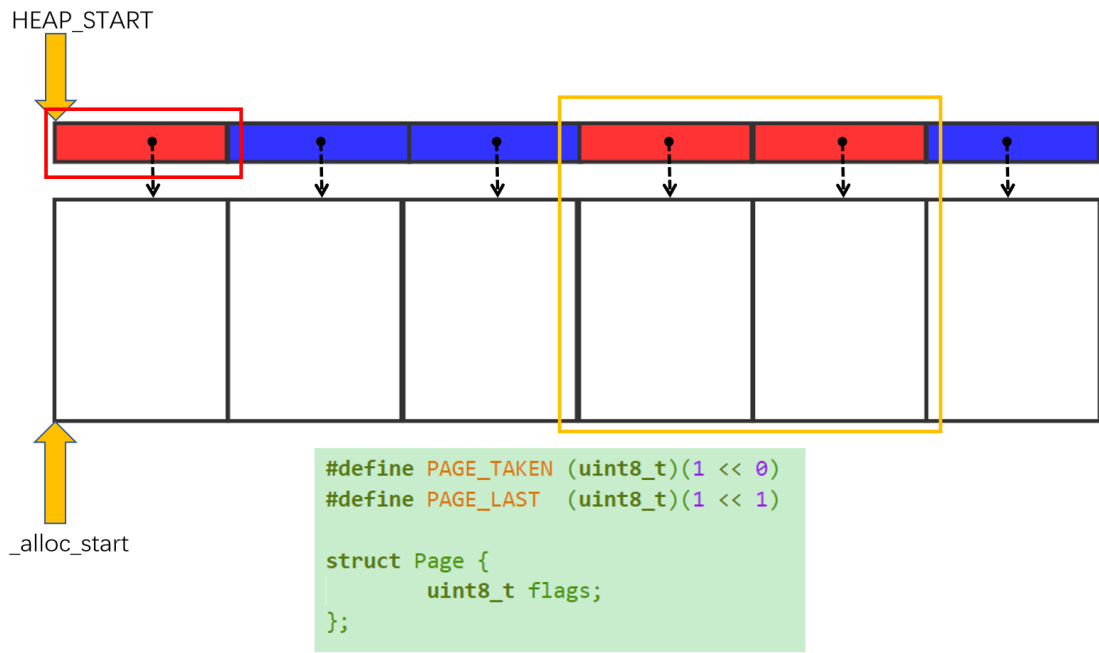
ISCAS NIST



改正后正确的讲稿页截图:

Page 描述符数据结构设计

ISCAS NIST



## 6. 第 9 章 上下文切换和协作式多任务

### 6.1. 【勘误 9-1】

参考 [issue 描述](#)。相关代码在 v0.9.1 版本中改正，并修改有关的课件 [ch09-context-switch.pdf](#) 中涉及 `switch_to()` 函数的代码截图。

## 7. 第 10 章 Trap 和 Exception

---

### 7.1. 【勘误 10-1】

参考 [issue 描述](#)。相关代码在 v0.9.1 版本中改正，并修改有关的课件 [ch10-trap-exception.pdf](#) 中涉及 `trap_vector()` 函数的代码截图。

### 7.2. 【勘误 10-2】

参考 [issue 描述](#)。课程中有关 vectored 方式的讲解有问题，Vectored 模式下针对 interrupt 的数组中不是放的下一级跳转函数的地址。

参考 The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified 的 Page 27 有关 Vectored 模式的处理机制，摘抄如下：

When MODE=Vectored, all synchronous exceptions into machine mode cause the pc to be set to the address in the BASE field, whereas interrupts cause the pc to be set to the address in the BASE field plus four times the interrupt cause number.

规范上只是说 PC 会跳转到具体数组项的入口地址，但数组中并不一定放的是下一级函数的地址。一个实际的例子可以参考提供的参考文档：<https://starfivetech.com/uploads/sifive-interrupt-cookbook-v1p2.pdf> 的 3.1.4 节中的 Figure 1: CLINT Vector Table Example Using Jump Instructions。我们看到实际数组中存放的是 j 指令，实现了我们所期望的二级跳转。

## 8. 第 11 章 外部设备中断

---


### 8.1. 【勘误 11-1】

“PLIC 编程接口 - 寄存器”介绍 Pending 寄存器那一页的内存映射地址的公式有错误，详细描述参考 [issue 描述](#)。




错误的讲稿页视频截图 (视频 P21 播放第 20 分钟左右处)：

PLIC 编程接口 - 寄存器




可编程寄存器	功能描述	内存映射地址
Pending	用于指示某一路中断源是否发生。	$\text{BASE} + 0\text{x}1000 + ((\text{interrupt-id}) / 32)$

- 每个 PLIC 包含 2 个 32 位的 Pending 寄存器，每一个 bit 对应一个中断源，如果为 1 表示该中断源上发生了中断（进入 Pending 状态），有待 hart 处理，否则表示该中断源上当前无中断发生。
- Pending 寄存器中断的 Pending 状态可以通过 claim 方式清除。
- 第一个 Pending 寄存器的第 0 位对应不存在的 0 号中断源，其值永远为 0。



改正后正确的讲稿页截图：

PLIC 编程接口 - 寄存器



可编程寄存器	功能描述	内存映射地址
Pending	用于指示某一路中断源是否发生。	$\text{BASE} + 0\text{x}1000 + ((\text{interrupt-id}) / 32) * 4$

- 每个 PLIC 包含 2 个 32 位的 Pending 寄存器，每一个 bit 对应一个中断源，如果为 1 表示该中断源上发生了中断（进入 Pending 状态），有待 hart 处理，否则表示该中断源上当前无中断发生。
- Pending 寄存器中断的 Pending 状态可以通过 claim 方式清除。
- 第一个 Pending 寄存器的第 0 位对应不存在的 0 号中断源，其值永远为 0。

相关代码涉及 `PLIC_PENDING` 这个宏的定义，已经在 v0.9 版本中改正。