

中山大学本科生 优秀毕业论文

鸿蒙LiteOS-M到RISC-V平台的移植

学位申请人：陈禹诚

导 师：黄 聃

专 业：信息与计算科学

学 院：计算机学院

2021年6月



本科生毕业论文（设计）

题目： 鸿蒙 LiteOS-M 到 RISC-V
平台的移植

姓 名 陈禹诚

学 号 17342003

院 系 计算机学院

专 业 信息与计算科学

指导教师 黄聃 (副教授)

2021 年 5 月 25 日

鸿蒙 LiteOS-M 到 RISC-V 平台的移植


Porting Harmony LiteOS-M to RISC-V Platform

姓 名	陈 禹 诚
学 号	17342003
院 系	计算机学院
专 业	信息与计算科学
指导教师	黄 聃 (副教授)

2021 年 5 月 25 日

学术诚信声明

本人郑重声明：所呈交的毕业论文（设计），是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文（设计）不包含任何其他个人或集体已经发表或撰写过的作品成果。对本论文（设计）的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本论文（设计）的知识产权归属于培养单位。本人完全意识到本声明的法律结果由本人承担。

作者签名：

日期：2021 年 5 月 25 日

【摘 要】

LiteOS-M 是鸿蒙开源操作系统 (OpenHarmony) 项目下的针对小型设备提出的轻量级操作系统内核, 这类小型设备特征是内存在 128KB-1MB 上下, 在物流、照明、传感、智能家居等领域有广泛的运用。RISC-V (Reduced Instruction Set Computer V) 是 2010 年起源于加州大学伯克利分校的指令集架构, 其简洁、开源、可扩展、免费等特点得到了近年来学术界和工业界愈发广泛等关注。

本文将上述轻量级开源操作系统与开源指令集架构相结合, 将 LiteOS-M 移植到 RISC-V 的 Picorio EVB 平台上。首先进行内核汇编代码移植, 用 RISC-V 指令集架构汇编实现内核启动、上下文切换、中断异常处理等内核汇编代码, 并用软件指令集架构模拟器初步调试运行。接着通过板级内核代码适配, 将内核在 PicoRio EVB 平台上运行。最后将计时器, GPIO, I2C 等硬件抽象层下的硬件驱动移植到 LiteOS-M。

本文的贡献在于 (1) 进一步完善 RISC-V 开源社区的软件基础设施建设, 加速 RISC-V 指令集架构的软件生态发展。因为软件基础设施如编译器, 操作系统等的高质量配套供给是否到位会很大程度影响到一个指令集的应用。(2) 增加 OpenHarmony 支持的硬件平台。将 LiteOS-M 移植到 RISC-V 的硬件平台能够扩大 OpenHarmony 开源操作系统的运用范围。(3) 开源, 免费的软硬件解决方案权益不被公司控制, 也不会被国家所垄断, 能允许人们自由更改用于商业, 教学, 科研等。软硬件开源方案的增加丰富能够产生值得提倡的社会效益。

【关键词】 LiteOS 操作系统, RISC-V 指令集架构, OpenHarmony

[ABSTRACT]

LiteOS-M is a lightweight operating system kernel proposed for small devices under the OpenHarmony open source operating system project. This type of small device features a memory of up to 128KB to 1MB, and has a wide range of fields in logistics, lighting, sensing, and smart home application. RISC-V (Reduced Instruction Set Computer V) is an ISA (Instruction Set Architecture) that originated from the University of California, Berkeley in 2010. Its features simplicity, open source, scalability, and free cost. It has attracted more and more attention from academia and industry in recent years.

This paper combines the above-mentioned lightweight open source OS with an open source ISA, by porting LiteOS-M to the RISC-V PicoRio EVB platform. First, the kernel assembly code is ported. The RISC-V assembly is used to realize the kernel assembly code like the kernel startup, context switching, interrupt exception handling. The software ISA simulator is used for preliminary running the modified system. Then through board-level kernel code adaptation, the kernel can run on the PicoRio EVB platform. Finally, the hardware drivers under the hardware abstraction layer such as timer, GPIOs, and I2C are ported to LiteOS-M.

The contributions of this paper (1) Improve the software infrastructure of the RISC-V open source community and accelerate the software eco development of the RISC-V. This is because whether the high-quality supporting supply of software infrastructure such as compilers and OS is in place will greatly affect the application of an ISA. (2) Increase the hardware platform supported by OpenHarmony. Porting LiteOS-M to the RISC-V hardware platform can expand the scope of application of the OpenHarmony system. (3) The rights and interests of open source and free software and hardware solutions are not controlled by the company or monopolized by the state, allowing people to freely change them for use in business, teaching, scientific research, etc.

[Keywords] LiteOS-M Operating System, the OpenHarmony Project, RISC-V

目录

一、 绪论	1
1.1 选题背景与意义	1
1.2 国内外研究现状和相关工作	3
1.3 本文的工作和贡献	3
1.4 本文的论文结构与章节安排	4
二、 背景知识	6
2.1 RISC-V SPEC	6
2.2 RISC-V 交叉编译工具链	6
2.3 Spike: RISC-V 指令集架构模拟器	6
2.4 LiteOS-M	7
2.5 PicoRio EVB	7
三、 移植工作	9
3.1 内核汇编代码移植	9
3.2 内核板级代码适配	15
3.3 驱动代码移植	16
四、 测试评估与应用	23
4.1 内核测试	23
4.2 硬件抽象层测试	23
4.3 实验结果	25
4.4 现实应用与社会效益	29
五、 总结	30
六、 未来工作的展望	31
参考文献	32

附录 A 代码补充	36
A.1 GPIOSetIrq	36
A.2 内核任务切换测试	37
A.3 GPIO 读写测试	37
A.4 GPIO 中断测试	38

一、诸论

1.1 选题背景与意义

计算机系统的高速发展很大程度上归功于不同子系统之间有高效的抽象接口,使得信息能够在不同子系统中有效地传递,来协调整个计算机系统的工作。软硬件接口毫无疑问是计算机系统最重要的接口之一。IBM 360^[2] 是世界上第一台以指令集架构规范化计算机软硬件交流接口的机器,在当时 IBM 发布了一个系列六种使用不同硬件实现方法,成本性能完全不同的机器。由于它们规范化了软硬件接口(指令集架构),这些硬件不同的机器能够执行同样的软件。IBM 360 系列机器实现了软硬件设计的解耦,也第一次引入了指令集架构这个概念。指令集架构是一台计算机的抽象模型,硬件对指令集的不同实现则是对指令集架构的实例化。指令集架构一般会定义数据类型,寄存器,主存管理,内存模型,IO 等规范,并以指令的形式为软件定义接口,上层软件基础设施如编译器,操作系统会基于指令集架构进行设计。

在指令集架构的发展演化过程中,出现了复杂指令集架构(CISC)与精简指令集架构(RISC)两个不同类别的指令集架构。复杂指令集架构的特征是一条指令的语义含义较多,一条复杂指令集指令语义等价于好几条底层指令的语义。而精简指令集则提供高度优化过的少数比较常用的指令^{[3][5][4]}。目前以 x86 为代表的复杂指令集架构在商业上广泛运用于个人电脑,服务器等运用场景。而以 ARM 为代表的精简指令集架构商业上广泛运用于手机,平板,嵌入式设备,并正在往个人电脑与服务器领域渗透^[7](例如 Apple M1 计算机与 Ampere Altra Max)。

过去的几十年以来,Intel 和 ARM 两家公司控制了 x86 和 ARM 这两种指令集架构。从物联网设备,到手机,到智能电脑,到云服务器,在这些设备中,我们几乎找不到不含有 Intel 和 ARM 专利的设备。现在几乎所有任意尺寸的电子产品都会包含处理器,越来越多的公司希望针对自身需求设计自己的处理器降低成本。而这些商用指令集架构的专利保护与它的复杂性是对人们自行设计创新处理器的巨大阻碍:(1) 公司依靠售卖知识产权核的设计或者芯片来盈利,他们不希望其他人能够自由的去用指令集架构去做出硬件实现来争夺他们的市场。(2) 目前商用指令集如复杂指令集 x86 极其复杂,用硬件来完整实现它们门槛极高,而如果没有完整的对其进行实现,未经特定修改的软件是无法在上面运行的。即便是属于精简指令集架构的 ARM 也非常复杂,例如 ARMv7 仅仅是整数指令都超过了 600

条,即便法律上得到了 ARM 的授权,基于这个指令集架构设计一个具体的硬件实现门槛也是极高的^[1]。

考虑到过去的比如 TCP/IP 网络协议接口开源形成标准对计算机界产生革命般的进步,指令集架构作为软硬件的接口也应当开源并形成标准。开源的指令集架构 (1) 能通过自由市场竞争促进更多的创新与好的架构的诞生; (2) 能共享好的核的设计,这些核能复用降低成本,并因为开源透明而减少设计错误与漏洞后门; (3) 能降低处理器成本,这意味着更多电子设备能用上处理器,这能进一步降低成为物联网设备的门槛^[8]。目前开源的指令集架构如 SPARC, MIPS, OpenRISC 有比较多的缺陷,比如不支持地址无关码,不支持压缩码,没有分离的特权指令等等。

基于以上因素,2010 年 RISC-V 指令集架构在加州大学伯克利分校应运而生。它的特点有: (1) 不会针对任何微架构做特定的优化,因为一旦对某种架构过分优化了,就会损失对其他微架构的友好程度; (2) 免费开源,这能减少构建新架构的代价,降低学界业界架构研发门槛与代价^[1]。随着 CMOS 电路晶体管增速在放缓,摩尔定律逐渐失效,想要进一步增长性能,人们需要根据自身应用场景定制专用化的计算设备。作为开源的指令集架构, RISC-V 允许用户自行添加扩展指令集,研发针对自身需求场景的自定义芯片。相比于其他的精简指令集架构, RISC-V 具有更高层度的灵活性以及更低的成本去针对一个应用场景设计生产定制化的专用芯片^[9]。

在过去的几年里, RISC-V 已经悄然进入了主流的计算领域,科技市场调查机构 Semico Research Corp 预测在 2025 年 RISC-V 核的 CPU 市场会增长到 624 亿美元的规模,约占 CPU 总市场的 6%,平均每年增长 158%(CAGR)^[10]。三星,西部数据,英伟达,高通等行业龙头已经宣布它们会在硬盘, GPU, AI 加速核等领域使用 RISC-V 相关的应用。

在物联网领域,人们对小型的,电池供能的设备需求在快速增长,这些设备对制造成本,低功耗,时变工作量等特性有比较高的要求。RISC-V 因灵活,开放,安全,简洁等优势得到了物联网领域学界与业界的广泛研究与运用^{[11][12][13][14]}。

OpenHarmony^[15] 是源于华为的面向全场景分布式开源操作系统,目前该项目隶属于开放原子基金会 (OpenAtom Foundation), OpenHarmony 在物联网设备领域获得了广泛的关注和运用。OpenHarmony 采用了组件化分层设计方案,允许用户根据自身设备的性能和资源选择相应的组件定制操作系统,不同配置的 OpenHarmony 支持在 128KB-128MB 不同大小的设备上运行。从小型的几百 KB 的穿戴设备,到大型几百 MB 的摄像头设备,都可以运行 OpenHarmony,并能基于同一套系统,分布式调用各个设备的服务组件。

OpenHarmony 遵循分层设计，自顶向下依次分为 4 层：（1）应用层；（2）框架层；（3）系统服务层；（4）内核层与驱动层。应用层运行面向用户的程序，如桌面，电话，设置，控制栏等。框架层为应用开发提供语言框架，如 JS UI 框架，Ability 框架等。系统服务层会具体实现一些系统基本能力例如语言运行时系统、分布式软件栈。并为应用层提供如位置，事件通知等软件服务。在内核与驱动层，OpenHarmony 针对内核与驱动设计了一套抽象接口：KAL(Kernel Abstract Layer) 与 HDF(Hardware Driver Foundation)。KAL 统一了内核对外提供的接口服务，允许用户使用不同的内核组件（例如在 LiteOS-M，LiteOS-A，Linux 之间做出选择）。HDF 规定了硬件驱动的开发和接口标准供驱动开发人员使用。

LiteOS-M 是 OpenHarmony 内核层针对几百 KB 级别物联网设备的轻量级内核组件。其对上定义了 KAL(Kernel Abstract Layer) 和 HAL(Hardware Abstract Layer) 接口提供服务。KAL 主要是 POSIX 或 CMSIS 接口，HAL 则是硬件驱动对上层的接口。它具有小体积（最小内核尺寸只有几 KB），低功耗，高性能等特点，被运用在抄表，停车，路灯，环保。物流，传感器，智能家居等多个场景，是 OpenHarmony 的重要组成部分。

1.2 国内外研究现状和相关工作

RISC-V 指令集架构作为最具前景的开源精简指令集，它的发展和运用目前很大程度上被软件基础设施（操作系统，编译工具链等）所束缚。在 RISC-V 的软件列表上^[16]，以操作系统为例，目前只能看到部分主流的操作系统如 Linux，FreeRTOS 等操作系统有 RISC-V 的开源版本^{[17][18]}，和一批如 PikeOS，Vxworks RTOS 不开源的商用操作系统^{[20][21]}。至于 LiteOS，无论是华为的 LiteOS 代码仓库^[19]，还是 OpenHarmony 的 LiteOS 代码仓库^[22]，开源的 RISC-V 指令集架构平台的移植代码还有待人们来补充。

1.3 本文的工作和贡献

本文尝试将 RISC-V 这个极具潜力的开源指令集架构与开源的 OpenHarmony 的 LiteOS-M 操作系统相结合，在 PicoRio EVB 这个 RISC-V 平台上运行 LiteOS-M，面临的挑战有：

- 1) 内核汇编代码的移植。LiteOS 的代码主要由 C 代码与汇编代码构成，内核中很多核心的机制都是由汇编代码完成的，例如内核启动时机器级寄存器和 C 语言运行栈的初始化，任务机制的上下文切换，异常的寄存器现场保护，中

断处理等。而汇编代码是和指令集架构捆绑的，不同指令集架构的汇编代码不同。从其他平台移植 LiteOS-M 到 RISC-V 平台需要将内核的汇编代码用 RISC-V 的汇编实现一遍。

- 2) 板级内核代码的适配，内核汇编移植完后，可以用软件指令集架构模拟器 Spike 对内核代码进行调试，能够比较方便的定位错误的代码地址。但是操作系统二进制文件能在 Spike 上运行并不意味着能在 PicoRio EVB 真实硬件上运行。要想在真实硬件上输出调试信息，就需要 UART 支持，而 UART 的移植工作只能在真实硬件上进行。这意味着在 UART 移植成功前，在真实硬件上的调试，都是处于无法打印调试信息的“盲调”状态，这显著增加了定位错误的难度。
- 3) 硬件驱动的适配。平台级中断控制器 PLIC 是 RISC-V 平台上实现外部中断下细分各类硬件设备中断的基础。本文需要实现平台级中断控制例程，并在平台级中断例程调用各类硬件驱动的中断的处理例程。同时，本文需要支持硬件抽象层同类设备可能有多个不同实例挂载不同硬件驱动函数的设计，为上层应用提供硬件抽象层统一的接口。

由于这是将开源操作系统移植到开源指令集架构的硬件平台上形成的开源的软硬件解决方案，本文的贡献有：

- 1) 进一步完善 RISC-V 开源社区的软件基础设施建设，加速 RISC-V 指令集架构的软件生态发展。因为一个指令集能否得到广泛的应用，不仅仅取决于有没有高效的硬件设计，同时还取决于软件基础设施如编译器，操作系统等的高质量的配套供给是否到位。
- 2) 增加 OpenHarmony 支持的硬件平台。目前 OpenHarmony 的 LiteOS-M 支持的是一些 ARM 指令集架构下的目标芯片。将 LiteOS-M 移植到 RISC-V 的硬件平台下能够扩大 OpenHarmony 开源操作系统的运用范围。
- 3) 开源，免费的软硬件解决方案权益不被公司控制，也不会被国家所垄断，能允许人们自由更改用于商业，教学，科研等。软硬件开源方案的增加丰富能够产生值得提倡的社会效益。

1.4 本文的论文结构与章节安排

本文共六章，第一章是绪论，阐明本文的背景信息，选题意义以及国内外一些其它相关的工作。

第二章展开谈论了一些关于本工作会用到的知识背景，如交叉编译工具链，指

令集架构模拟器等。

第三章谈论移植工作的细节，共三部分，分别是内核汇编级代码移植、内核板级代码适配以及驱动代码移植。

第四章提供了移植系统正确性的测试，测试包括内核汇编代码以及硬件抽象层各个驱动接口的移植代码的正确性测试。

第五章对全文进行总结。

第六章基于本工作以及相关工作的背景对未来工作进行展望。

二、背景知识

本章将围绕移植 OpenHarmony LiteOS-M 到 RISC-V 平台这项工作,介绍 RISC-V (交叉编译器,软件模拟器等), LiteOS-M 以及 RISC-V 硬件平台 Picorio EVB 相关的一些背景知识。

2.1 RISC-V SPEC

RISC-V 指令集架构的标准目前是由 RISC-V International^[23] 维护的,目前有 Unprivileged Spec, Privileged Spec, External Debug Support, Trace Spec 四个部分,分别描述了非特权级架构,特权级架构,调试支持架构,处理器分支跟踪架构。移植操作系统会主要使用到一些特权和非特权的汇编指令,指令的具体语义能在非特权级架构和特权级架构的 SPEC 找到对应的解释。

2.2 RISC-V 交叉编译工具链

交叉编译通常指在一个平台编译另一个平台的可运行二进制文件。人们一般在个人电脑 (x86) 平台上编译出能在手机 (ARM), 嵌入式设备 (ARM, RISC-V) 上运行的程序,并将程序导入目标设备运行。本文将会利用 RISC-V GNU 交叉编译工具链^[24],在个人电脑 (x86) 端将移植修改后的操作系统编译到 RISC-V 的目标平台上。

2.3 Spike: RISC-V 指令集架构模拟器

Spike 指令集架构模拟器是通过软件方式对指令级架构进行模拟的工具。Spike 自身是一个 x86 的程序,它模拟了 RISC-V CPU 如何工作,一个目标平台是 RISC-V 的二进制文件是可以在 Spike 上运行的。有了 Spike,我们可以直接在 x86 个人电脑上运行一个 RISC-V 的程序,并对其进行调试。Spike 支持断点,汇编,输出寄存器等调试功能,能够方便移植系统过程中发现问题。



图 2-1 OpenHarmony 框架

2.4 LiteOS-M

Openharmony^[15] 设计分为 4 层，分别是应用层，框架层，系统服务层与内核层，如图 2-1。其中应用层运行面向用户的程序，框架层为应用层开发者提供语言框架支持，系统服务层提供基础分布式服务和框架层运行时服务支持，内核层通过内核抽象层和硬件抽象层对上提供统一的内核服务接口和硬件驱动接口。其中内核抽象层下面的内核可以是 LiteOS-M，可以是 LiteOS-A，也可以是 Linux，它们的内核服务被统一抽象成归一化的内核抽象层接口，对上提供统一的语义。

本文要移植的 LiteOS-M 位于内核层，LiteOS-M 对上层提供统一的内核抽象层接口服务。同时硬件抽象层的代码也需要修改适配本文的目标 RISC-V 平台：PicoRio EVB，来提供如 GPIO，UART，I2C 等 IO 服务。

2.5 PicoRio EVB



图 2-2 PicoRIO EVB

PicoRio图 2-2是类似于 Raspberry Pi^[26] 的一款开源的 RISC-V 版级微型电脑系统，有开源，低功耗，体积小等特点。和 Raspberry Pi 不同的是，PicoRio 会将其 SoC，板级电路，软件基础设施等设计开源，而基于 ARM 的 Raspberry Pi 的相关

设计是闭源的，并且 PicoRio 的功耗在 50mW 到 300mW 之间，远低于 Raspberry Pi^[25]。

PicoRio EVB 是 PicoRio 的一款测试评估板，支持 RISC-V IMC 指令，有 L1 缓存 8KB，L2 缓存 256KB，支持 I2C，SPI，GPIO，UART 等 IO 端口。

三、移植工作

本文的移植工作分为 3 部分，如图 3-1，首先将 LiteOS-M 内核汇编代码改为 RISC-V 的汇编代码，并尝试在 Spike 指令集架构软件模拟器上初步运行并调试，当 LiteOS-M 能够在 Spike 上运行后，将进行内核板级代码适配，适配后将编译好的二进制文件通过 SPI 录入 PicoRio EVB 平台上运行，运行时会遇到一些因移植导致的内核错误，需要移植 UART 驱动打印调试信息进行调试解决。内核能够成功在 PicoRio EVB 平台上正常运行、切换任务后，再进行驱动代码移植。本文将 GPIO, I2C, UART, Timer 等常用驱动移植到了 LiteOS-M 系统，并设计了一些测试确认驱动的正常运行。

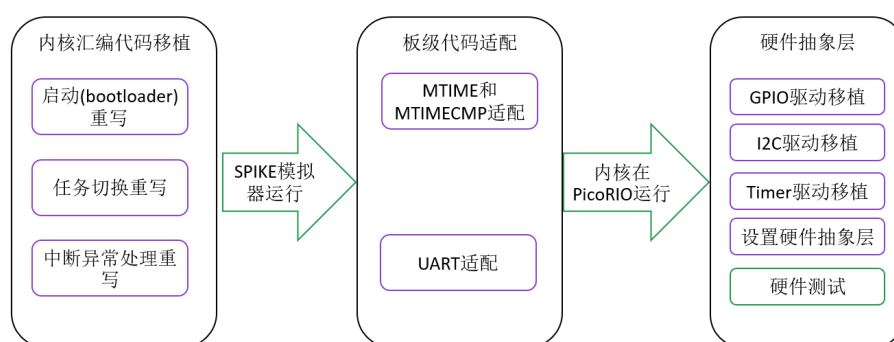


图 3-1 移植流程

3.1 内核汇编代码移植

操作系统作为管理硬件资源的软件，其中涉及到中断处理，上下文切换，操作系统启动等工作，这些工作和硬件指令集架构相关，无法用高级语言实现，需要对应指令集架构的汇编语言来实现，即本文需要将原内核 ARM 汇编利用 RISC-V 汇编代码重写一遍。

3.1.1 特权级寄存器的处理

汇编代码的实现涉及到大量的特权级寄存器的读写，本小节将描述移植过程中需要处理的特权级寄存器。

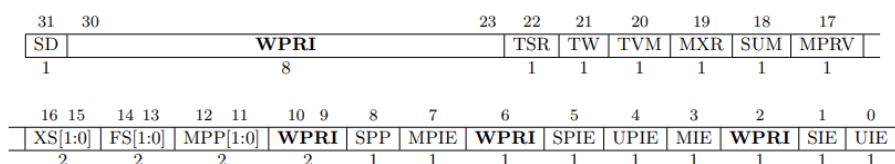


图 3-2 mstatus 字段图

3.1.1.1 mstatus 寄存器

mstatus 寄存器如图 3-2，在 mstatus 寄存器中，本文会处理 MPP，MIE，MPIE 三个字段。

MPP(Machine Previous Privilege) 位于 **mstatus** 寄存器第 11, 12 位 bit。表示上一个特权级是多少, 0 代表 **User**, 1 代表 **Supervisor**, 3 代表 **Machine**, 当调用 **mret** 指令时, 特权级会返回到上一个特权级的值, 也就是 **MPP** 的值。有趣的是, **RISC-V** 的标准没有定义寄存器去储存当前特权级的值是多少, 硬件实现可以实现寄存器来存储当前特权级的值方便解码, 但是软件就没有对应的指令来访问存储当前特权级指令的寄存器。这可以加强安全性。

MIE(Machine Interrupt Enable) 值为 1 时机器级全局中断处于开启状态，允许中断。MIE 值为 0 意味着中断关闭。MPIE(Machine Previous Interrupt Enable) 指上一个特权级的中断是否开启。当 `mret` 触发时，MIE 的值会被 MPIE 的值取代，类似于 `MPP` 对当前特权级的取代。

3.1.1.2 mie 寄存器

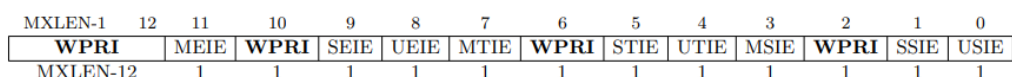


图 3-3 mie 字段图

mie 寄存器字段如图 3-3，mie 寄存器本文会处理 MEIP(Machine External Interrupt is Pending) 字段，如果 MEIP 的值为 1，说明有外部中断待处理，如果 mstatus 的 MIE 字段也允许中断，则会去处理中断。

3.1.1.3 mtvec 寄存器

mtvec(Machine Trap-Vector base-address register) 寄存器前两位记录该中断向量的存储格式，第 2 位之后的字段存储中断向量的地址。本文会利用 mtvec 寄存器来完成中断异常处理相关工作。

3.1.1.4 mcause 寄存器

mcause 寄存器需处理的字段分为两个字段，Interrupt 字段和 Exception Code 字段。Interrupt 字段为 1 代表这是个中断，不是异常，此时 Exception Code 能取 9 个值，分别代表 9 不同的中断，既用户 (User)，监管者 (Supervisor)，机器 (Machine) 三种特权级和软件中断，时钟中断，外部中断三种中断类型相组合，例如用户软件中断。当 Interrupt 字段为 0 时，代表这个中断是异常，Exception 字段有 14 种取值，分别代表不同的异常，例如指令访问错误，非法指令，断点等。

3.1.2 RISC-V 汇编实现启动模块

启动部分的汇编代码的实现主要是设置一些寄存器的初值，并建立 C 语言执行的栈环境，之后跳转到 LiteOS 内核的主函数。

本文约定起始 DRAM 起始地址，以及 PC 起始地址均为 0x80000000，二进制文件可以利用 .ld 链接脚本文件确保 .text 段的入口布局在 0x80000000，同时运行软件模拟器 Spike 时可以加上 pc=0x80000000 参数来指定 pc 初始位于 0x80000000 的位置。

启动部分汇编代码实现逻辑如下：

- 1) 设置 mstatus 的 MPP 字段为 3。
 - LiteOS-M 是轻量级物联网操作系统，全程在机器特权级下运行。故 MPP 应当为 3 确保 mret 后特权级仍然为机器特权级。
- 2) 清空 mie 寄存器。
 - 初始状态下是没有外部中断的，要确保 mie 被初始化。
- 3) 设置 mtvec 的值指向中断向量表的地址。
 - 只有设置了 mtvec 才能正常触发中断以及异常处理。
- 4) 设置栈指针寄存器 SP 的值为栈基址，建立 C 语言执行环境，栈基址是在链接脚本文件定义的栈段确定的。
- 5) 跳转到 LiteOS 的 main 主函数，启动汇编代码部分结束，main 函数后由 C 语言代码接管。

3.1.3 RISC-V 汇编实现 LiteOS-M 任务切换机制

操作系统任务切换机制涉及到中断，上下文保存相关的问题，这些都是在汇编层面进行操作的。本文需要对这些汇编用 RISC-V 指令集架构汇编进行重写。

3.1.3.1 任务切换机制

本小节对本文重写汇编涉及到的任务切换框架进行整体性描述。

- 任务的创建
 - 程序员可以利用 `LOS_TaskCreate` 注册任务，需要提供任务的栈大小以及任务名，任务优先级等信息。创建任务会为任务初始化任务控制块 (TCB) 以及分配并初始栈环境。创建好的任务会加入操作系统的等待执行的任务的优先队列里，优先队列内顺序以任务优先级为关键字。
- 进入任务调度模式
 - 在内核初始化完成之后，LiteOS 会通过 `LOS_StartToRun` 进入任务调度模式，开始执行等待列表中的任务。
- 任务的调度与切换
 - 全局变量 `g_losTask` 是一个结构体，结构体内由 `newTask` 和 `runTask` 两个 TCB 组成，`runTask` 的 TCB 是目前正在运行的任务的 TCB，`newTask` 的 TCB 是根据优先级等调度规则，下一个将要运行的任务的 TCB。
 - `LOS_Schedule` 函数是 LiteOS 的调度函数，每次调用该函数，该函数会先更新 `newTask` 的 TCB，然后如果发现 `newTask` 不等于 `runTask`，则调用汇编逻辑进行上下文切换，将 `runTask` 用 `newTask` 替换掉。

本文移植的工作主要是实现 `LOS_StartToRun`，任务上下文切换相关的内容。

3.1.3.2 RISC-V 汇编实现 `LOS_StartToRun`

当内核初始化完毕，LiteOS 会通过 `LOS_StartToRun` 进入任务调度模式开始执行任务。在 `LOS_StartToRun` 中我们会初始化 `g_taskScheduled` 为 1 来开启调度，因为如果 `g_taskScheduled` 为 0，`LOS_Schedule` 函数只会更新 `newTask`，但是不会执行切换。接着，在汇编中将 `runTask` 赋为 `newTask` (`newTask` 会在任务创建时就维护好)，并将 `newTask` 的执行上下文赋给各个寄存器（在创建任务初始化任务栈空间时，会把寄存器上下文存在栈中）。切换 TCB 以及设置 TCB 过程中，要确保中断是关闭的。可以通过清除 `mstatus` 的 MIE 字段来关闭中断。切换完 TCB 以及 sp 寄存器后可以直接将栈中原 `newTask` 的 `mstatus` 取出恢复中断。

因此，本文实现 `LOS_StartToRun` 方案是：

- 1) 关闭中断
 - 利用 `csrci` 指令清除 `mstatus` 的 MIE 字段解除使能
- 2) 将 `g_taskScheduled` 调度使能变量赋值为 1

- 先用 `la` 指令读取 `g_taskScheduled` 地址，再将 1 的值利用 `li` 读入寄存器，再用 `sw` 指令将该值存入 `la` 得到的地址。
- 跳转至切换任务的逻辑，利用切换任务逻辑将 `newTask` 的 TCB 覆盖 `runTask` 的 TCB

3.1.3.3 RISC-V 汇编实现上下文切换

上下文切换指当前在运行 `runTask` 的 TCB 对应的任务，由于时间片耗尽，或是调用了任务休眠函数等原因，LiteOS 会调用 `LOS_Schedule` 函数将当前 `runTask` 换掉。

本文实现的切换上下文汇编函数会先将包括 `mstatus` 和 `mepc` (`mepc` 设置成 `ra` 寄存器的值，也就是进入上下文切换函数前的原任务的地址) 在内的上下文寄存器保存。并设置 TCB 的运行状态字段，再将 `newTask` 拷贝覆盖 `runTask`。接着获取了原 `newTask` 的栈指针后恢复原 `newTask` 的上下文。恢复 `mepc` 的之后调用 `mret` 指令，`pc` 指针将继续执行原 `newTask` 的 `mepc` 地址之后的指令。

因此本文对上下文切换的实现逻辑如下：

- 1) 用 `csrrc` 改变 `mstatus` 关闭中断
- 2) 用 `la` 读取 `g_losTask` 的地址
- 3) 取出 `g_losTask` 的 `newTask` 和 `runTask` 指针指向的地址
- 4) 如果地址相等，说明 `newTask=runTask`，用 `csrw` 写 `mstatus` 开启中断，不执行切换，返回上一层调用
- 5) 如果地址不相等，则开始执行上下文切换逻辑
- 6) 用 `csrr` 与 `sw` 保存 `mstatus` 与 `mepc` 的值入栈
- 7) 保存被调用者的寄存器 (CALLEE Register)，即 `s0-s11`
- 8) `lh` 读取 `runTask` 当前的状态到临时寄存器
- 9) `and` 将临时寄存器的是否在运行的 `flag` 设置成 1
- 10) `sh` 临时寄存器的值存回 `runTask` 任务控制块
- 11) `sw` 将 `runTask` 任务控制块存入栈中
- 12) 执行切换成 `newTask` 的逻辑
- 13) 先利用 `la` 将 `runTask` 的地址读出来，再利用 `lw` 将 `newTask` 的值读出来，最后利用 `sw` 将 `newTask` 的值存入 `runTask`，实现对 TCB 的拷贝
- 14) 利用 `lh` 读取当前 `runTask` 的值到临时寄存器，利用 `ori` 将临时寄存器运行 `flag` 置位为 1，再将其存入 `runTask` 中，完成对运行 `flag` 的赋值
- 15) 利用 `lw` 读取栈中 TCB 存的栈指针 `sp` 的值，赋值给 `sp`

- 16) 利用 `csrw` 写 `mstatus` 寄存开启全局中断，并利用栈中存的 `mepc` 值更新当前 `mepc` 值
- 17) 弹出所有栈中寄存器上下文
- 18) 执行 `mret`，跳转到 `mepc` 的地址，也就是栈中恢复的 `mepc` 的值

3.1.4 RISC-V 汇编实现中断处理

出现中断时，`pc` 会跳转到 `mtvec` 地址，执行中断处理的汇编函数。

本文实现的中断处理首先通过读取 `mcause` 的 `Interrupt` 字段确定中断类型。分两种情况讨论：

如果 `Interrupt` 值为 0，意味着这是异常，需要读取当前寄存器的值存在栈中，保留异常现场，调用 LiteOS 的 `OsExcEntry` 输出在栈中保留的寄存器值以及任务控制块的值，供使用者分析调试，输出完异常后再进入执行对应错误的中断处理向量，执行完中断处理向量后，会跳转至上下文切换逻辑将任务切换成 `newTask`。

如果 `Interrupt` 值为 1，意味着这是中断，将直接进入中断向量进行处理。以机器时间中断为例，在内核初始化阶段会用 `LOS_HwiCreate` 来注册硬件中断，注册需要传入机器时间中断的中断码和中断处理例程。当机器时间中断触发后，进入 `mtvec` 的中断处理程序，会跳到 `OsHwiInterruptDone` 查找中断向量找到当时注册的中断处理例程，并执行。执行完中断处理例程后会跳转到任务上下文切换逻辑，将 `runTask` 切换成 `newTask`（除非任务处于锁状态 `g_losTaskLock` 不为 0）。

因此，本文对中断异常处理的流程是：

- 1) 保存 ECALL 级别的寄存器，例如 `sp`，`a0-a7`，`ra` 寄存器等，存入栈空间。
- 2) 接着用 `csrr` 读取 `mcause` 的字段到临时寄存器，取出临时寄存器的 `mode` 字段，判断是异常还是中断。
- 3) 如果是异常，跳入异常处理逻辑
 - 异常处理逻辑会保存剩下的所有寄存器到栈里面
 - 并用 `csrr` 读取诸如 `mtval`，`medeleg`，`mstatus`，`mepc` 等寄存器到临时变量，再用 `sw` 指令存临时变量到栈里面。以此来保存 CSR 异常现场。
 - 接着跳转至 C 语言实现的 `OsExcEntry`，打印刚刚存的寄存器以及任务状态控制块（TCB）的异常上下文现场。
 - 打印完后跳转回中断处理逻辑
- 4) 如果不是异常，直接执行中断处理逻辑，跳转至 `OsHwiInterruptDone`
- 5) 在 `OsHwiInterruptDone` 内，会用中断号访问中断向量数组 `g_hwiForm[]`。
- 6) 取得中断向量数组的中断函数地址后，跳转至中断服务例程。

3.1.5 软件指令集架构模拟器运行

本文在做出上述更改后，可以在内核进入任务调度前注册几个简单的样例任务，然后将 LiteOS-M 利用 RISC-V 交叉编译工具链（riscv32-unknown-elf-***）将代码编译成 elf 文件，在 spike 软件指令集架构模拟器上运行。Spike 支持断点，步进，打印寄存器等调试功能。调试成功后可以见样例任务切换交替执行。

3.2 内核板级代码适配

内核板级代码适配主要工作是将内核时钟频率和 PicoRio EVB 的时钟频率对齐。并且为了确认板级代码适配成功，需要适配 UART 来输出 PicoRio EVB 的打印信息。

3.2.1 mtime 寄存器的处理

在 RISC-V 指令集架构下，mtime 隔固定周期会正比例增加一个常量。PicoRio EVB 平台，mtime 的值和周期数呈 1:1 的关系增加。

3.2.2 mtimecmp 寄存器的处理

每当 mtime 的值大于或等于 mtimecmp 的值时，就会触发机器级时间中断，mcause 的 Exception 中断码为 7。这个中断会在内核初始化的过程做注册，并且由 mie 的 MTIE 字段使能。本文定义时钟中断(Tick)间隔为 20000 个周期(cycle)。考虑到 PicoRio EVB 时钟频率为 200000000，故设置 LOSCFG_BASE_CORE_TICK_PER_SECOND 为 1000。将 mtime 初值设置成 0，mtimecmp 设置成 20000，然后每次触发时钟中断后都将 mtimecmp 增加 20000 来作为下一次时钟中断触发的条件。

3.2.3 UART 适配

设置好内核板级代码的适配后，将 LiteOS-M 编译成二进制，通过串口导入 PicoRio EVB 平台上，并且将 PicoRio EVB 的 pc 寄存器的值设置成 0x80000000，LiteOS 便可以运行，并且交替执行样例的任务。但是没有 UART 打印 LiteOS-M 的调试信息，我们无从得知 LiteOS-M 在 PicoRio 里面是否真正在运行，是否遇到内核错误，任务是否能成功切换等。故我们需要提前适配 UART 驱动来完成对内核板级代码适配的调试。

UART 采用轮询的方式对外进行输出，每输出一个字符 ch，驱动函数会将字

符写到 UART_THR 的寄存器地址上，然后轮询读取 UART_LSR 寄存器的状态值，如果该状态的 TEMT 和 THRE 字段同时为 1，说明输出成功，结束轮询，可以继续输出下一个字符。

本文实现了以下 UART 接口：

1) rvHal_uart_init()

- 初始化 LCR, IER, FCR, MCR 等寄存器
- 初始化波特率为 38400

2) rvHal_uart_tx_poll(char ch)

- 每次将 ch 放入 THR 地址
- 轮询 LSR_ADDR 的 TEMT 和 THRE 字段，为 0 则输出成功，结束轮询，继续输出其他字符

适配完毕 UART 后，可以用 minicom 串口信息输出工具实时打印 UART 的输出。有了 UART，在内核代码调试过程中能够通过输出中间结果快速定位内核出错位置，完成调试。

内核代码适配成功后，可以在 minicom 上通过打印的信息观察到任务能够执行以及交替切换。

3.3 驱动代码移植

内核代码能够在 PicoRio EVB 平台上正常运行后，要想提供更多的硬件服务，需要移植硬件驱动到 LiteOS-M 上。本文在驱动代码移植的工作主要为移植 timer, GPIO, i2c 到 LiteOS-M 上。

3.3.1 平台级中断控制 (PLIC) 的处理

本文通过利用 RISC-V 指令集架构独有的 PLIC，实现了对不同驱动设备中断的分类处理。

PLIC(Platform-Level Interrupt Controller) 是 RISC-V 定义的将设备的中断作为外部中断带优先级地分发给 RISC-V 核的控制器。在 LiteOS-M 和 PicoRio EVB 的场景，只有一个 RISC-V 核，PLIC 的流程是通过机器级外部中断调用 PLIC 的中断处理例程 OsMachineExternalInterrupt，例程会读取 PLIC 的 Src (中断源) 寄存器来判断是哪种类型的 PLIC 外部中断 (例如 timer 的中断值是 17，GPIO 的中断值是 15)。

更简化的理解，可以认为 PLIC 是依托机器级外部中断，对不同硬件设备中

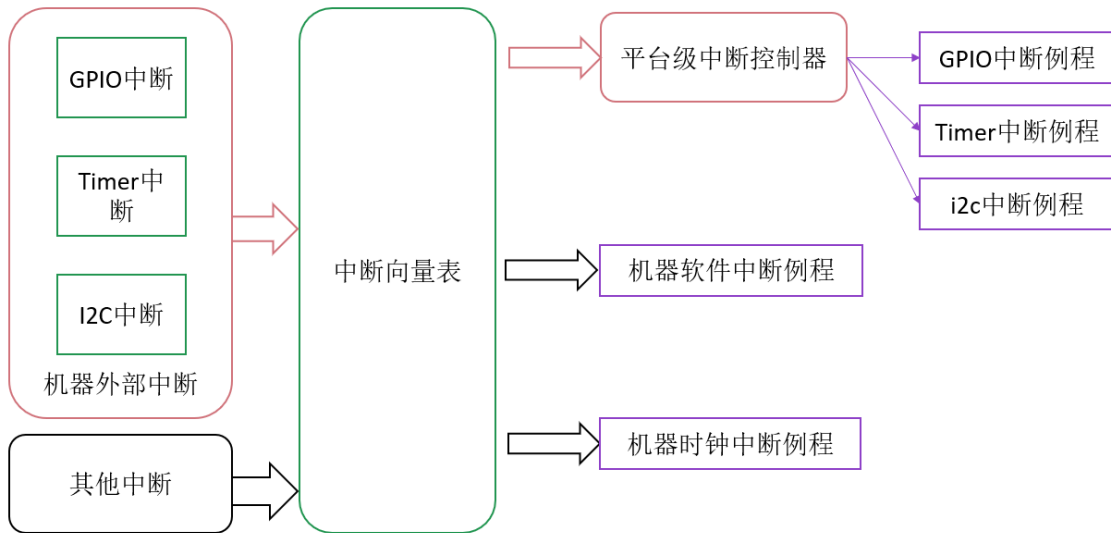


图 3-4 平台级中断控制器

断的更细一层的分类，并对每一种硬件设备中断类型有与之对应的中断服务例程。要想支持 PLIC，硬件上需要一些 PLIC 相关的寄存器和中断逻辑，例如 Src 寄存器存硬件设备中断的类型。这里面需要一些硬件逻辑和寄存器来支持。

本文根据中断类型来调用对应的 PLIC 中断服务例程如图 3-4。PLIC 中断服务需要硬件设备驱动程序在初始化的时候注册 PLIC 中断，传入对应 PLIC 中断类型号码和对应类型的 PLIC 中断服务例程。

本文通过实现机器外部中断来实现平台级中断的分发机制如下：

```

static VOID OsMachineExternalInterrupt(UINT32 arg) {
    unsigned int irq;
    irq = readPlicReg(RegSrc(0)); // 读取PLIC中断码
    if (irq < NUM_IRQ_EXT_MAX) {
        if (irq_ext_handlers[irq]) { // 执行中断函数
            (irq_ext_handlers[irq])->hook();
        }
        else { // 这个中断号没有注册中断函数
            PRINTK("Error: no handler found");
        }
    }
    else { // 中断号溢出
        PRINTK("Error: irq number out of range");
    }
}
    
```

}

首先可以注意到这是机器级外部中断的中断服务例程，在这个中断服务例程里面本文实现了 PLIC 的中断号分发机制，实现逻辑如下：

- 1) 读取 PLIC 的 src 寄存器的值，这个值决定了这个中断是哪个外部硬件发起的
- 2) 根据 src 寄存器的值，调用对应中断服务例程 irq_ext_handlers[irq]
- 3) 如果对应 irq_ext_handlers[irq] 没有被注册，则报错输出 debug log。
- 4) 如果对应寄存器的值超出 irq 索引最大值，则报错输出对应 debug log。

3.3.2 timer

DW_apb_timer 是为用户层提供计时中断服务的硬件 IP，该 IP 含有 8 个相同且独立的计时器，每个计时器有独立的寄存器配置、使能地址。它通过 PLIC 发生机器级外部中断，在 PLIC 处理例程中，会调用计时器驱动初始化时传入的中断处理函数，这个函数用户可以自己配置，例如实现灯泡定时闪烁，计时器中断例程就可以是点亮、熄灭灯泡的逻辑。

本文定义了一个数据结构 timer_cfg_struct 来完成寄存器的配置，内含定时时间，PWM，计时器模式等成员变量，以及配置函数 rvhal_timer_config，将数据结构和计时器 id 传入配置函数，会直接将成员变量的值写入对应寄存器。

本文对计时器的开启与关闭的实现是通过更改 TimerNControl 的第 1 个使能字段进行实现。

以下将介绍 timer 驱动接口：

1) rvhal_timer_init(TimerDev* host, int32_t id)

- 重置第 id 个 timer 的 reset 寄存器和 SEL 寄存器。reset 寄存器是重置寄存器状态，SEL 寄存器会将对应 pin 选择该 timer
- 注册 timer 中断到平台级中断控制器中，这样触发的 timer 平台级中断能够执行到注册的中断服务函数
- 将该驱动的驱动函数全部挂载在该硬件抽象类实例 host 的对应函数成员变量：
 - 抽象类 Config 对应 rvhal_timer_config, 抽象类 Deinit 对应 rvhal_timer_deinit,
 - 抽象类 Disable 对应 rvhal_timer_disable 抽象类 Enable 对应 rvhal_timer_enable。

2) rvhal_timer_config(TimerDev* host, void* config)

- config 数据结构带有对 host 指向的 timer 的初始化数据，该函数会将这

些数据写入对应 timer 的寄存器里面。

3) `rvhal_timer_enable(TimerDev* host)`

- 本接口会将 timer 使能寄存器写使能。调用此接口，定时器便开始计时。

4) `rvhal_timer_interrupt(void)`

- 这个函数是 timer 的中断处理函数，timer 计时器时间到，会触发机器外部中断，机器级外部中断会通过 PLIC 中断分发函数调用本函数。这个函数交由用户定义，用以执行用户的需求功能。

5) `rvhal_timer_disable(TimerDev *host)`

- 本函数会解除 host 对应的计时器的使能寄存器的使能位。该定时器将停止工作。

6) `rvhal_timer_deinit(TimerDev *host)`

- 该函数会注销硬件抽象类实例 host 以及其所对应的 timer。

3.3.3 GPIO

GPIO(General Purpose Input/Output) 是微控制器和其他电子设备进行交流的标准接口。GPIO 可以与例如传感器，灯泡，显示器等设备相连，进行信息传输，控制等活动。

GPIO 的每个 pin 有输入，输出两种状态，本文实现了 `GPIOSetDir` 与 `GPIOGetDir`，可以用 `GPIOSetDir`/`GPIOGetDir` 来设置或读取 pin 的状态。用杜邦线将一个输出 pin 和输入 pin 相连，当用 `GPIOWrite` 写输出 pin 时，可以在输入 pin 用 `GPIORead` 读到写的值。

本文实现了 GPIO 的 PLIC 中断，一共有五种触发中断的方式，分别是水平和边缘两种触发模式和高电压和低电压两种触发条件的组合，共四种如图 3-5，外加一种无论是高电压边缘还是低电压边缘都会触发的模式。例如高电压边缘触发模式会在 pin 从低电压变成高电压时触发 PLIC 中断，PLIC 会调用 GPIO 的中断处理例程，GPIO 中断处理例程会找出是哪个 pin 触发的中断，并调用对应 pin 的中断处理例程。

本文实现了 GPIO 的以下接口：

1) `rvhal_GpioInit()`

- 本函数会初始化 GPIO 所有 Pin，并为 GPIO 注册 GPIO 的 PLIC 中断服务程序。

2) `rvhal_GpioRead(uint16_t gpio, uint16_t *val)`

- 该函数会读取 gpio 编号的 GPIO pin，并将值通过 val 指针的形式传出。

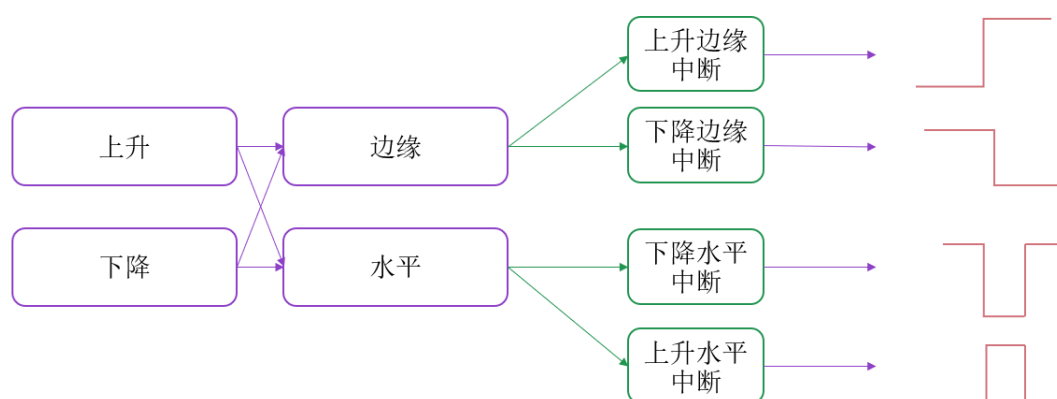


图 3-5 GPIO 的 2x2 触发组合

3) rvhal_GpioWrite(uint16_t gpio, uint16_t val)

- 该函数会将 val 值写到编号为 gpio 的 pin 口上。

4) rvhal_GpioSetDir(uint16_t gpio, uint16_t dir)

- 设置编号为 gpio 的 pin 的输入输出状态，dir 为 0 设为输入状态，dir 为 1 设为输出状态。

5) rvhal_GpioGetDir(uint16_t gpio, uint16_t *dir)

- 读取编号为 gpio 的 pin 的输入输出状态，*dir 为 0 设为输入状态，*dir 为 1 设为输出状态。

6) rvhal_GpioSetIrq(uint16_t gpio, uint16_t mode, GpioIrqFunc func, void *arg)

- 这是在设置 GPIO 的中断类型，通过 mode 的值判断中断是边沿触发，还是水平触发，是高电位触发，还是低电位触发等。
- 该函数会根据中断触发类型，对 GPIO_INTTYPE_LEVEL(水平，边沿)，GPIO_INT_POLARITY (高电位，低电位)，GPIO_INT_BOTHEDGE (第五种中断类型) 寄存器进行设置。并注册中断服务函数 func。细节可以参考节 A.1。

7) rvhal_GpioUnSetIrq(uint16_t gpio)

- 通过写 GPIO_INTTYPE_LEVEL(水平，边沿)，GPIO_INT_POLARITY (高电位，低电位)，GPIO_INT_BOTHEDGE (第五种中断类型) 清空 GPIO 中断的设置。

8) rvhal_GpioEnableIrq(uint16_t gpio)

- 对编号为 gpio 的 pin 使能 GPIO 中断，使能后，GPIO 电压变化达到中断类型要求便会通过机器外部中断触发对应的 GPIO PLIC 中断。

9) rvhal_GpioDisableIrq(uint16_t gpio)

- 通过写 GPIO_INTEN 解除对编号为 gpio 的 pin 的 GPIO 中断使能。

10) rvhal_gpio_irq_handler()

- 这是 GPIO 的 PLIC 中断服务例程，它会管理所有的 GPIO Pin 引发的中断。
- 该例程会穷举所有 Pin，询问是否是第 i 个 Pin 发生中断，如果是，则调用第 i 个 Pin 在 rvhal_GpioSetIrq 处设置第 i 个 GPIO 中断服务例程。

3.3.4 I2C

I2C(I-Squared-C) 是一种同步多主从串口通讯协议。本文将 PicoRio EVB 上 DW_apb_i2c 硬件对应的支持驱动移植到了 LiteOS-M 上，支持向其他设备发送，接收数据或指令。I2C 驱动在初始化时会向 PLIC 注册 I2C 中断处理例程，当上层应用想要使用 I2C 进行通讯时，可以调用 I2CTransfer 函数，I2CTransfer 函数会解除对 I2C PLIC 中断的屏蔽，指定 I2C 收发 buffer 地址，并使能 I2C 模块。

当触发 I2C 中断时会通过寄存器对应字段判断是发送，还是接收数据。判断完后会通过读写指定的收发 buffer 来完成数据的传输。

3.3.5 硬件适配层

硬件适配层是将移植过来驱动的接口来适配表达 LiteOS 硬件抽象层接口的层次。



图 3-6 硬件抽象层

本文对每一类硬件服务定义了一个对应的设备抽象类，设备抽象类的成员函数可以挂载不同该类设备的驱动具体实现。一个硬件抽象层接口往往会指明操作的对应类的设备 id 或 handler，接着会调用对应设备抽象类 handler 的成员函数，来执行对应该类设备对应 id 的硬件驱动程序。

以 Timer 为例，在硬件抽象层有例如：TimerOpen, TimerConfig, TimerEnable 等

面向上层应用的接口。TimerOpen(id) 会实例化编号为 id 的 TimerDev 抽象类, 并将该 TimerDev 初始化, 为该 TimerDev 挂载对应硬件的驱动函数, 并返回该 TimerDev 的指针作为 handler 给上层应用。当上层应用想要执行例如 TimerEnable 时, 可以直接传入 TimerDev 的 handler, 接着就会执行对应 handler 的 TimerEnable 的函数实例, 比如 TimerDev 的 handler 是 DW_apb_timer 时, 这个 TimerEnable 会调用 DW_apb_timer 的使能函数。具体如图 3-6

硬件抽象层通过抽象化每类设备的行为, 定义了接口。本文的硬件适配层对每类抽象硬件都定义了一个 handler 数组, 来管理同类硬件不同设备, 以及对应的实现驱动与数据结构, 是对硬件抽象层的接口的通用化的实现方式。

四、测试评估与应用

本章将介绍对移植工作的测试与评估的方案以及结果。

4.1 内核测试

移植工作对内核的主要改动位于任务切换，中断处理。本文将从任务切换和中断处理两个方向测试内核功能是否正确。

4.1.1 任务切换

本文设计了两个样例任务 `SampleTaskEntry`，具体见附录节 A.2。每个样例任务主体为一个无限 `while` 循环，循环内调用了一个 `LOS_TaskDelay` 和 `printf` 输出任务自身信息。两个样例任务 `LOS_TaskDelay` 时间设置多几个不同的进行测试，这样可以测试任务切换正确性的同时测试机器时钟相关配置是否正确。测试结果在图 4-2 可见。

4.1.2 中断处理

中断处理测试希望确认异常处理的代码块能够正常输出内核异常信息。本文在运行样例任务时，故意插入一个越界的数组访问，运行时会出现异常并进入中断处理流程，保存上下文寄存器内容，调用 `OsExcEntry` 将寄存器信息打印出来，并输出任务的异常信息。通过该实验可以确认中断处理以及异常处理的汇编代码块能够执行达到预期效果。

4.2 硬件抽象层测试

本节将介绍对移植涉及到的硬件抽象层接口的测试。

4.2.1 Timer

Timer 的测试相对简单，本文在 Timer 的中断处理函数中输出中断提示，利用 `TimerOpen` 和 `TimerConfig` 创建并配置一个计时器，在 `TimerEnable` 使能后，会间隔一定时间发生中断并输出中断提示。测试结果见图 4-2。

4.2.2 GPIO

GPIO 的测试分为两部分，GPIO 读写测试和中断测试。

4.2.2.1 GPIO 读写测试

本文在 GPIO 读写测试中用杜邦线连接两个 GPIO Pin，并利用驱动函数设置输入 Pin 和输出 Pin。设置好后进行了 32 次写测试，其中 16 次写 0，16 次写 1，每次写测试伴随 10000 次读测试，观察到输出 Pin 写的值和输入 Pin 读出来的值一样，达到预期。

测试细节可以见节 A.3。

4.2.2.2 GPIO 中断测试

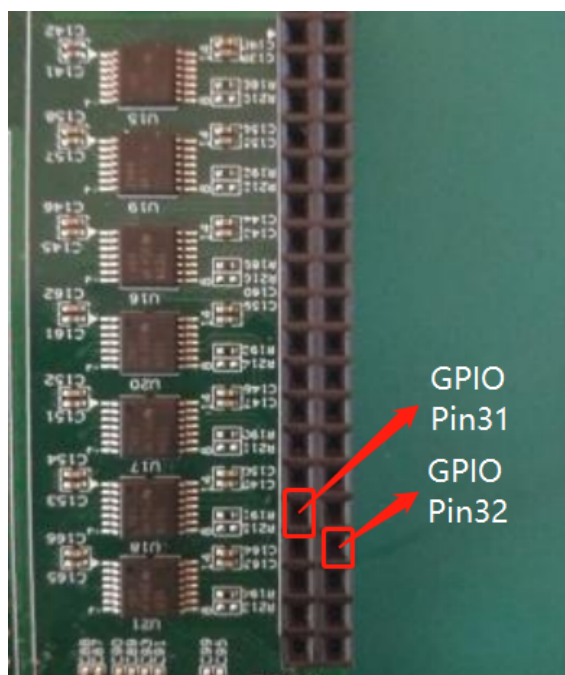


图 4-1 GPIO pin 31 - GPIO pin 32

本文在 GPIO 中断测试中同样用杜邦线连接两个 GPIO Pin，例如图 4-1 连接 Pin 32 与 Pin 31，将 GPIO 电位设置成低电位。随后设置 GPIO 的 PLIC 中断，PLIC 的中断函数会打印提示信息表示中断被触发方便观察。设置完中断后，会用 GPIOWrite 反复改变 Pin 的电位，当出现上升沿时，中断会被触发并打印信息，符合预期。接着解除 GPIO 中断，再反复改变 Pin 的电位，中断不会再出现，符合预期。

测试细节可以见节 A.4，测试结果可以见图 4-2。

4.2.3 I2C 测试

本文采用了 SENSIRION 的温湿度传感器 SHT2x 作为 I2C 通讯串口的应用, 来测试 I2C 的硬件抽象接口是否正确工作。

4.2.3.1 SHT2x 简介

SHT2x 是 SENSIRION 的数字温湿度传感器, 具有低电压, 稳定工作等特点, 利用 I2C 协议进行输出。

4.2.3.2 SHT2x 温湿度测试

本文通过 I2C 协议与 SHT2x 组件进行交流, 得出当前环境的温度值与湿度值, 来确认 I2C 硬件驱动程序的正确性。

SHT2x 的实现如下:

- 1) 首先利用硬件抽象接口初始化一个连向 SHT2x 的 Pin 对应的 I2C 实例
- 2) 利用 I2CTransfer 函数使能 I2C 中断以及指明接收 buffer 地址
- 3) 当中断发生时在 I2C 中断处理例程向 SHT2x 传输, type 参数, 指明是读取温度还是湿度
- 4) 再次等待 I2C 接收中断, 接收两个 byte 的温度或湿度数据
- 5) 通过 SHT2x 的使用手册的公式计算出温度和湿度值
- 6) 测试测出温度和湿度符合测试时的温湿度情况, 说明 I2C 模块正确工作。

测试结果可见图 4-2。

4.3 实验结果

4.3.1 Bootloader/任务切换机制/硬件抽象层驱动

本文根据 PicoRIO EVB 的 Databook, 将电源, UART, 温湿度传感器 (i2c) 和 PicoRIO EVB 对应引脚 (Pin) 进行连接。执行脚本将 LiteOS-M 的二进制文件烧录进 PicoRIO EVB, 并将 PC 设置成代码段入口地址 0x80000000:

```
${HOST_BIN_DIR}/fesvr2spi.o ++$(liteos-elf)
++set_pc=0x80000000 ++release_orv32_reset
++release_cache_reset
```

烧录后, 启动 minicom 信息控制台, 用 UART 查看内核输出的实验结果图 4-2。

[illegible]

图 4-2 minicom 输出的结果

在图 4-2 中，共有 7 个部分的结果已在图中用数字标注，现对各个部分结果进行解释：

- 1) minicom 启动欢迎信息，本文采用 minicom 2.7 版本
- 2) LiteOS-M 进入内核的欢迎信息，这条信息能够确认：
 - 启动模块 (bootloader) 正确
 - 内核已进入 C 语言环境
 - UART 驱动正常工作
- 3) I2C 测试通过提示
 - 测试环境是在空调环境下，测试环境相对干燥
 - 测试温度结果为 26 度，湿度结果为 54，符合当时的温湿度情况
- 4) timer 中断触发
 - 可以注意到 timer 中断触发前已经在切换任务了
 - 可以通过配置 timer 中断间隔参数线性改动 timer 终端间隔，timer 中断会出现在对应的位置。从而得知 timer 是正确的。
- 5) GPIO 测试
 - GPIO 读写测试成功，共检测 32 次

- GPIO 中断触发测试成功，上升沿中断被触发两次

6) 内核任务切换

- 可以看到两个任务按 1: 2 的次数轮流出现。
- TaskSample1 本文设置的 delay 是 1000ms, TaskSample2 设置的 delay 是 500ms。
- 切换效果符合预期，说明任务切换模块汇编正确。

4.3.2 异常中断处理

本文在 TaskSample 中故意插入：

```
printf("TaskSampleEntry1 running...\n\r");
g_memStart[0x7fffffff] = 123;
```

这是一条显然会数组越界产生访存异常的指令。预期结果应当是输出完 printf 中的内容后，对数组 g_memStart 访问越界产生异常，进入中断向量表的异常处理逻辑，保存现场上下文寄存器后输出上下文寄存器以及 TCB 内容。

以下是寄存器上下文输出结果上半部分：可以观察到图 4-3 在出现了 printf 的

```
TaskSampleEntry1 running...

Exception Information
Exc_type : Oops - Store/AMO access fault!
taskName = TaskSampleEntry1
taskID = 2
system mem addr:0x80005980
mepc      = 0x8000031e
mstatus   = 0x1800
mtval     = 0x597f
mcause    = 0x7
ra        = 0x8000031e
sp        = 0x80008028
gp        = 0x80005e60
tp        = 0x0
t0        = 0x5050505
t1        = 0x4040404
t2        = 0x0
s0        = 0x800080d8
s1        = 0x5980
```

图 4-3 寄存器上下文现场 1

内容”TaskSampleEntry1 running”后立即出现了异常中断输出信息。根据图 4-3，我们可以获得以下 debug 信息：

- 1) 异常类型: Store/AMO access fault, 即访存指令错误
- 2) 产生错误的任务: TaskSampleEntry1, 错误任务编号为 2, 该结果符合我们实验预期
- 3) 系统内存地址: 0x80005980, 可以注意到系统设置的内存基址是 m_aucSysMem0, 从图 4-4 的 bss 段可知地址为 0x80005980。符合实验预期。

Sections:						
Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00005550	80000000	80000000	00000080	2**5
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
1	.rodata	00000100	80005550	80005550	000055d0	2**2
			CONTENTS, ALLOC, LOAD, READONLY, DATA			
2	.data	00000320	80005660	80005660	000056e0	2**5
			CONTENTS, ALLOC, LOAD, DATA			
3	.bss	000103c0	80005980	80005980	00005a00	2**6

图 4-4 二进制 Objdump Section 信息

- 4) mepc: 当异常发生时, mepc 会保存发生错误时的 pc 地址, 在异常处理逻辑会被压入栈中后被输出, 图 4-3 的 mepc 值为 0x8000031e。为了验证该结果的正确性, 本文将 LiteOS-M 二进制通过 Objdump -S 查看对应地址汇编如图 4-5, 可见:

- (a) 0x800002ea 是 TaskSampleEntry1 的入口
- (b) 0x8000031e 是本文设置的越界代码

g_memStart[0 x 7 f f f f f f f] = 123;

所在的位置, 上一条指令 jal 对应的是上述的 printf 指令。结果符合预期。

```

322 800002ea <TaskSampleEntry1>:
323 800002ea: 1101          addi sp,sp,-32
324 800002ec: cc22          sw s0,24(sp)
325 800002ee: ca26          sw s1,20(sp)
326 800002f0: c84a          sw s2,16(sp)
327 800002f2: c64e          sw s3,12(sp)
328 800002f4: ce06          sw ra,28(sp)
329 800002f6: 1000          addi s0,sp,32
330 800002f8: 80004537      lui a0,0x80004
331 800002fc: 9e850513      addi a0,a0,-1560 # 800039e8 <__start_and_irq_stack_top+0xffffb4a8>
332 80000300: 6499          lui s1,0x6
333 80000302: 2a0d          jal 80000434 <printf>
334 80000304: 800049b7      lui s3,0x80004
335 80000308: 98048493      addi s1,s1,-1664 # 5980 <START_AND_IRQ_STACK_SIZE+0x3980>
336 8000030c: 07b00913      li s2,123
337 80000310: 3e800513      li a0,1000
338 80000314: 4b2010ef      jal ra,800017c6 <LOS_TaskDelay>
339 80000318: a1098513      addi a0,s3,-1520 # 80003a10 <__start_and_irq_stack_top+0xffffb4d0>
340 8000031c: 2a21          jal 80000434 <printf>
341 8000031e: ff248fa3      sb s2,-1(s1)
342 80000322: b7fd          j 80000310 <TaskSampleEntry1+0x26>
    
```

图 4-5 二进制 Objdump 反汇编信息

以下是 TCB 的异常输出图 4-6, 可以从图 4-6 获得以下调试信息:

ID	Pri	Status	name
--	--	-----	----
0	0	QueuePend	Swt_Task
1	31	Ready	IdleCore000
2	6	Running	TaskSampleEntry1

图 4-6 异常 TCB 现场输出

- 任务队列被输出，可看到所有任务的状态，优先级，名字，编号。

4.3.3 测试总结

本文通过上述实验可以确认内核汇编，i2c，GPIO，timer 等硬件驱动，以及硬件抽象层的接口能够正确工作。移植的 LiteOS-M 可以在 PicoRIO EVB 上正常运行执行功能。至此，移植宣告成功。

4.4 现实应用与社会效益

本文的工作处于开源状态，人们可以在本文的工作上进行应用，改良，教学，产生值得提倡的社会效应。

目前本工作在工业，教学等方面得到了公司，教育机构的采用。例如：

- 1) 某 RISC-V 芯片公司的系列芯片系统采用了本工作的 LiteOS-M 作为其操作系统软件基础设施的一部分。
- 2) 在知乎，电子发烧友等平台上，相关教学机构采用本工作作为其单片机、嵌入式开发等课程使用的操作系统。

五、总结

本文详细介绍了将开源的 LiteOS-M 轻量级实时操作系统内核移植到开源的指令集架构 RISC-V 的 PicoRio EVB 平台上的方法与过程。

首先进行内核汇编重写，将内核启动，任务上下文切换，中断处理与异常处理等指令集架构相关的汇编代码用 RISC-V 指令集架构汇编实现，并在 RISC-V 指令集架构软件模拟器 Spike 上调试运行。

接着进行板级内核代码适配，将 LiteOS 的时钟频率和时钟中断频率根据需要与 RISC-V PicoRio EVB 平台适配。为了能在 PicoRio EVB 上输出内核以及任务运行信息，本文适配了 UART 将 LiteOS-M 在板上的运行信息输出到 minicom 上供人查看。

最后是硬件抽象层接口下的驱动代码的移植，实现了 PLIC 中断服务例程后，本文移植了计时器，GPIO，I2C 等硬件模块的驱动程序，并添加了硬件适配层支持硬件抽象层管理多个同类型的不同硬件实例。

在内核测试方面，本文测试了内核任务切换和中断异常处理，对内核修改部分正确性进行了核对。在硬件抽象层测试方面，本文也测试了硬件抽象层接口是否能准确工作，通过中断打印、读写测试、温湿度传感器应用等方式分别测试了计时器，GPIO，I2C 的硬件抽象接口，确保了它们的正确性。

在现实运用与社会效益方面，人们可以自由地采用本工作作为自身的解决方案，本工作在工业，教学领域得到了一些芯片公司与教育机构的采用，作为其操作系统软件基础设施的一部分。

六、未来工作的展望

在本文的移植工作中，将 ARM 内核汇编代码改写成 RISC-V 汇编代码占了相当大的工作量。因为和高层语言相比，汇编语言要关注更多的硬件细节，并且语义粒度较细，编程复杂度较高。这是跨指令集移植软件基础设施的一大障碍。

同时本文作者了解到，对于大型操作系统的移植往往涉及到大规模依赖包的移植，这些依赖包往往只支持 x86 或 ARM 指令集架构。有的依赖包缺乏代码无从编译成 RISC-V 的二进制文件；有的依赖包代码能找到但是重写 Makefile 会耗费工程师相当大的精力；有的依赖包代码则版本无法对齐会产生许多错误。

如果未来能实现一个跨指令集架构翻译工具，对汇编代码或二进制文件进行跨指令集翻译，将会很大程度地减少上述移植的工作量。

跨指令集翻译可以通过点对点的方式^[28]或提升到中间语言（IR）再通过后端编译的方式实现。

点对点的方式指将源汇编指令直接透过语法树生成目标指令集指令，中间需对寄存器进行重命名。其实现简单，但是可扩展性比较弱，并且无法利用编译基础设施的优化。

中间语言的方式有动态翻译和静态翻译以及系统级与用户级之分^{[29][31][32][33]}，静态翻译在代码识别和间接跳转指令等准确率不足，动态翻译无法进行大幅度的编译优化。

目前还没有针对 RISC-V 比较高效且准确率高的系统级翻译工具。本文认为未来的工作可以考虑往跨指令集架构翻译的方向去做，从而降低跨指令集架构移植软件基础设施生态的成本。

参考文献

- [1] Waterman, A. S. (2016). Design of the RISC-V Instruction Set Architecture. UC Berkeley. ProQuest ID: Waterman_berkeley_0028E_15908. Merritt ID: ark:/13030/m50c9hkd. Retrieved from <https://escholarship.org/uc/item/7zj0b3m7>
- [2] Gene M. Amdahl and Gerrit A. Blaauw and Frederick P. Brooks Jr. Architecture of the IBM System/360, *IBM J. Res. Dev.*, 87–101(1964)
- [3] Patterson, David A., and David R. Ditzel. "The case for the reduced instruction set computer." *ACM SIGARCH Computer Architecture News* 8.6 (1980): 25-33.
- [4] Dileep Bhandarkar. 1997. RISC versus CISC: a tale of two chips. *SIGARCH Comput. Archit. News* 25, 1 (March 1997), 1–12. DOI:<https://doi.org/10.1145/250015.250016>
- [5] Bhandarkar, Dileep, and Douglas W. Clark. "Performance from architecture: comparing a RISC and a CISC with similar hardware organization." *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*. 1991.
- [6] Solomon, Baruch, et al. "Micro-operation cache: A power aware frontend for variable instruction length isa." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 11.5 (2003): 801-811.
- [7] Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan, and Karthikeyan Sankaralingam. 2015. ISA Wars: Understanding the Relevance of ISA being RISC or CISC to Performance, Power, and Energy on Modern Architectures. *ACM Trans. Comput. Syst.* 33, 1, Article 3 (March 2015), 34 pages. DOI:<https://doi.org/10.1145/2699682>
- [8] Asanović, Krste, and David A. Patterson. "Instruction sets should be free: The case for risc-v." *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

- [9] Greengard, Samuel. "Will risc-v revolutionize computing?." Communications of the ACM 63.5 (2020): 30-32.
- [10] Semico Research Corp., "RISC-V Market Analysis: The New Kid On the Block", <https://semico.com/content/risc-v-market-analysis-new-kid-block> (2019)
- [11] Schiavone, Pasquale Davide, et al. "Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications." 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS). IEEE, 2017.
- [12] Gautschi, Michael, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 25, no. 10 (2017): 2700-2713.
- [13] Flamand, Eric, et al. "GAP-8: A RISC-V SoC for AI at the Edge of the IoT." 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP). IEEE, 2018.
- [14] L. Auer, C. Skubich and M. Hiller, "A Security Architecture for RISC-V based IoT Devices," 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 2019, pp. 1154-1159, doi: 10.23919/DATE.2019.8714822.
- [15] OpenHarmony, [2021-4-10]. <https://gitee.com/openharmony>
- [16] Github:RISC-V software list, [2021-4-9]. <https://github.com/riscv/riscv-software-list#os-and-os-kernels>
- [17] Github: RISC-V Linux, [2021-4-9]. <https://github.com/riscvarchive/riscv-linux>
- [18] RISC-V FreeRTOS, [2021-4-11]. <https://www.freertos.org/Using-FreeRTOS-on-RISC-V.html>
- [19] Github: RISC-V LiteOS,[2021-4-3]. https://github.com/LiteOS/LiteOS_Lab/
- [20] Sysgo: RISC-V PikeOS,(2020-4-1)[2021-4-10]. <https://www.sysgo.com/press-releases/sysgo-adds-risc-v-support-to-its-pikeos-real-time-operating-system>

- [21] Windriver: RISC-V VxWorks RTOS, (2019-12-10)[2021-4-11].<https://www.windriver.com/news/press/news-22570>
- [22] Gitee: OpenHarmony LiteOS,[2020-11-1]. https://gitee.com/openharmony/kernel_liteos_m
- [23] RISC-V International,[2021-4-19]. <https://riscv.org/>
- [24] Github:RISC-V GNU Toolchain,[2021-4-10]. <https://github.com/riscv/riscv-gnu-toolchain>
- [25] RISC-V International: PicoRIO for RISC-V like Raspberry Pi for ARM,[2021-4-11]. <https://riscv.org/blog/2020/09/picorio-for-risc-v-like-raspberry-pi-for-arm/>
- [26] Raspberry Pi,[2021-4-19]. <https://www.raspberrypi.org/>
- [27] SENSIRION SHT2x,[2021-4-19]. <https://www.sensirion.com/en/environmental-sensors/humidity-sensors/humidity-temperature-sensor-sht2x-digital-i2c-accurate/>
- [28] Github:ARM Aarch64 to RISC-V Transpiler,[2020-12-28]
<https://github.com/schorrm/arm2riscv>
- [29] Bellard, Fabrice. "QEMU, a fast and portable dynamic translator." USENIX annual technical conference, FREENIX Track. Vol. 41. 2005.
- [30] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. Rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In Proceedings of the 26th International Conference on Compiler Construction (CC 2017). Association for Computing Machinery, New York, NY, USA, 131–141. DOI:<https://doi.org/10.1145/3033019.3033028>
- [31] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20). Association for Computing Machinery, New York, NY, USA, 133–147. DOI:<https://doi.org/10.1145/3373376.3378470>
- [32] Bansal S, Aiken A. Binary Translation Using Peephole Superoptimizers[C]//OSDI. 2008, 8: 177-192.

- [33] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. 2020. BinRec: dynamic binary lifting and recompilation. In Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20). Association for Computing Machinery, New York, NY, USA, Article 36, 1–16. DOI:<https://doi.org/10.1145/3342195.3387550>

附录 A 代码补充

A.1 GPIOSetIrq

```
// 节选 GpioSetIrq 的 switch 模块与设置中断例程部分代码
int32_t rvhal_GpioSetIrq(uint16_t gpio, uint16_t mode,
                        GpioIrqFunc func, void *arg)
-----

switch (mode)
{
case (OSAL_IRQF_TRIGGER_RISING |
      OSAL_IRQF_TRIGGER_FALLING):
    int_type = GPIO_INT_TYPE_BOTH;
    break;
case OSAL_IRQF_TRIGGER_RISING:
    int_type = GPIO_INT_TYPE_EDGE;
    int_polarity = GPIO_INT_POLARITY_HIGH;
    break;
case OSAL_IRQF_TRIGGER_FALLING:
    int_type = GPIO_INT_TYPE_EDGE;
    int_polarity = GPIO_INT_POLARITY_LOW;
    break;
case OSAL_IRQF_TRIGGER_HIGH:
    int_type = GPIO_INT_TYPE_LEVEL;
    int_polarity = GPIO_INT_POLARITY_HIGH;
    break;
case OSAL_IRQF_TRIGGER_LOW:
    int_type = GPIO_INT_TYPE_LEVEL;
    int_polarity = GPIO_INT_POLARITY_LOW;
    break;
case OSAL_IRQF_TRIGGER_NONE:
```

```

        return 0;
default:
        return -1;
}
irq_gpio_handlers[gpio].arg = arg;
irq_gpio_handlers[gpio].func = func;

```

A.2 内核任务切换测试

```

VOID TaskSampleEntry1(VOID)
{
    while(1) {
        LOS_TaskDelay(1000);
        printf("TaskSample1 helloworld...\n\r");
    }
}

```

```

VOID TaskSampleEntry2(VOID)
{
    while(1) {
        LOS_TaskDelay(500);
        printf("TaskSample2 helloworld...\n\r");
    }
}

```

A.3 GPIO 读写测试

```

// 设置连接的两个GPIO Pin的输入口和输出口
GpioSetDir(GPIO_TEST_PIN_IN, GPIO_DIR_IN);
GpioSetDir(GPIO_TEST_PIN_OUT, GPIO_DIR_OUT);
uint16_t in_value, out_value;
// 测试设置是否成功
GpioGetDir(GPIO_TEST_PIN_OUT, &out_value);

```

```
GpioGetDir(GPIO_TEST_PIN_IN, &in_value);
printf("out dir: %u\n\r", out_value);
printf("in dir: %u\n\r", in_value);
// GPIO 读写测试
for (i = 0; i < GPIO_TEST_TIMES; i++) {
    out_value = i & 1;
    GpioWrite(GPIO_TEST_PIN_OUT, out_value);
    for (j = 0; j < 10000; j++);
    GpioRead(GPIO_TEST_PIN_IN, &in_value);
    if (out_value != in_value) {
        err = 1;
        break;
    }
}
```

A.4 GPIO 中断测试

```
// GPIO_TEST_PIN_IN和GPIO_TEST_PIN_OUT
// 两个Pin已经连接好并设置好了输入输出模式
// 将Pin初始化为低电位 (GPIO_VAL_LOW)
GpioWrite(GPIO_TEST_PIN_OUT, GPIO_VAL_LOW);
// 设置中断为上升沿触发，使能中断
GpioSetIrq(GPIO_TEST_PIN_IN, OSAL_IRQF_TRIGGER_RISING,
            gpio_test_int, int_context);
GpioEnableIrq(GPIO_TEST_PIN_IN);
for (j=0; j<10000; j++);
// 上升沿触发中断
GpioWrite(GPIO_TEST_PIN_OUT, GPIO_VAL_HIGH);
// 下降沿不触发中断
GpioWrite(GPIO_TEST_PIN_OUT, GPIO_VAL_LOW);
// 上升沿触发中断
GpioWrite(GPIO_TEST_PIN_OUT, GPIO_VAL_HIGH);
// 解除中断模式，后续不会触发中断
```

```
GpioUnSetIrq(GPIO_TEST_PIN_IN);  
GpioWrite(GPIO_TEST_PIN_OUT, GPIO_VAL_LOW);  
GpioWrite(GPIO_TEST_PIN_OUT, GPIO_VAL_HIGH);  
GpioWrite(GPIO_TEST_PIN_OUT, GPIO_VAL_LOW);  
GpioWrite(GPIO_TEST_PIN_OUT, GPIO_VAL_HIGH);  
GpioWrite(GPIO_TEST_PIN_OUT, GPIO_VAL_LOW);  
GpioWrite(GPIO_TEST_PIN_OUT, GPIO_VAL_HIGH);
```