

rust_kernel 可行性报告

小组成员：段逸凡 陆万航 王浩宇 邱浩宸 雷婷

- rust_kernel 可行性报告
 - 1.理论依据
 - 1.1 Rust对嵌入式操作系统开发的支持
 - 1.1.1 内联汇编
 - 1.1.2 外部函数接口
 - 1.1.3 稳定的libcore
 - 1.1.4 总结
 - 1.2 ARM对嵌入式操作系统的支持
 - 1.2.1 ARM微处理器的特点
 - 1.2.2 ARM 微处理器的结构
 - 1.2.3 ARM微处理器的寄存器结构和工作模式
 - 1.2.4 ARM微处理器的指令结构
 - 2.技术依据
 - 2.1 重新编写的操作系统的规模
 - 2.2 freeRTOS 实时嵌入式系统简介
 - 2.2.1 优势
 - 2.2.2 freeRTOS的基本组成及功能
 - 3.创新点
 - 4.项目设计
 - 4.1 重写操作系统的路径选择
 - 4.2 具体实现--Nordic开发板
 - 4.2.1 开发板简介
 - 4.2.2 开发板优势和特点
 - 4.3 目标和展望
 - 5.参考文献

1.理论依据

1.1 Rust对嵌入式操作系统开发的支持

Rust是一种偏向于底层的语言，凭借着自身的安全性、高效性和并发性，成为了系统开发中的一门优秀的语言。同时，Rust对于嵌入式的开发也有着良好的支持，因此，利用Rust来编写嵌入式操作系统是可操作的、有效的。

1.1.1 内联汇编

在操作系统的开发中，有时，为了极端底层操作和性能要求，我们希望能够直接控制 CPU。在这个时候，高级语言的抽象特性使得我们对于CPU具体是如何工作的犹如雾里看花，即使是依靠反汇编也很难做到对CPU的精准操作，给我们带来了不小的阻碍。于是，我们应当使用汇编语言来编写这部分的代码。

Rust 通过 `asm!` 宏来支持使用内联汇编。通过这个功能，我们可以在Rust代码中直接嵌入汇编代码，并把变量和实际的寄存器关联起来。比如下面的例子，就是在正常的Rust代码中嵌入x86汇编指令：

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn add(a: i32, b: i32) -> i32 {
    let c: i32;
    unsafe {
        asm!("add $2, $0"
            : "=r"(c)
            : "0"(a), "r"(b)
            );
    }
    c
}
# #[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
# fn add(a: i32, b: i32) -> i32 { a + b }

fn main() {
    assert_eq!(add(3, 14159), 14162)
}
```

实际上，虽然这里的例子使用了 x86/x86-64 汇编，但所有平台都受支持。

1.1.2 外部函数接口

Rust是一门年轻而小众的语言，这使得它的库远不如C/C++等语言一样丰富。C语言作为一种常用的嵌入式开发语言，有着丰富的库函数。因此，如果能够利用C的库，那么这对于Rust而言会是一个重大的收获。

庆幸的是，Rust支持外部函数接口。虽然目前Rust还不支持直接调用C++库，但是，我们可以通过使用snappy压缩/解压缩库来为Rust编写外部语言代码绑定—— snappy 库包含一个 C 接口（记录在 snappy-c.h中）。比如下面就是一个调用C语言函数的例子：

```

# #![feature(libc)]
extern crate libc;
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}

```

事实上，我们完全可以实现Rust与C的混搭。不仅在Rust中调用C代码是可行的，在C中调用Rust代码也是合法的。这种特性使得我们的工作具有灵活性，我们可以用Rust编写重要的、安全性要求高的代码，而那些不重要的代码可以留用（毕竟，这些代码可能在很久之前就已经写好了，它们的安全性可能经过了时间的检验，改写它们带来的好处并不大）。

1.1.3 稳定的libcore

早在2016年的Rust 1.6版本，Rust就实现了libcore的稳定，这使得使用稳定的Rust进行OS和嵌入式开发成为可能。^[1]

libcore提供了一个底层的、平台无关的基础，Rust标准库libstd就是基于它构建像内存管理、I/O和并发等高级功能。因此，libcore是最底层，OS和嵌入式软件开发人员常常更喜欢以它为基础构建应用程序。这样说来，libcore的稳定为各种底层软件提供了稳固的基础。

Rust核心团队成员Steve Klabnik表示，稳定的libcore的重要性不可低估：“为了可以基于稳定的Rust进行OS/嵌入式开发，这是重要的一步。因此，这很重要，但那仍然只是第一步。我确实认为，表明Rust是最底层软件开发的一个可行选项，对于Rust未来的发展而言很重要。”

1.1.4 总结

Rust语言本身对于嵌入式操作系统的开发有着良好的支持，上述特性使得利用Rust开发一个嵌入式操作系统成为一件可行的事情。

此外，斯坦福大学于今年春季学期新开设了一门操作系统课程cs140e，和传统的操作系统课程cs140不同，该课程利用Rust语言在树莓派上编写嵌入式操作系统，并实现虚拟内存、进程、文件系统等模块，虽然该系统更加偏向于教学和试验性质，而不是真正实现一个可用的操作系统，但是也具有一定的实用价值和参考意义。可见，Rust开发嵌入式操作系统的可行性是得到相当程度的认可的。

同时，Rust团队发布的Rust 2018路线图将开发效率放在了首位，并把目标锁定在了Web Service、WebAssembly、基于命令行的应用程序和嵌入式设备四个领域。显然，Rust对嵌入式开发的支持将会进一步增强。

1.2 ARM对嵌入式操作系统的支持

ARM开发板，即以英国ARM（Advanced RISC Machines）公司的内核芯片作为CPU，同时附加其他外围功能的嵌入式开发板，可以用来评估内核芯片的功能和研发各类科技产品。

英国ARM公司是嵌入式RISC处理器的IP（知识产权）供应商，它为ARM架构处理器提供ARM处理器内核（如ARM7TDMI、ARM9TDMI及ARM10TDMI等）。各半导体公司在上述处理器内核基础上进行再设计，嵌入各种外围和处理部件，形成各种MCU。目前，基于ARM内核的芯片在嵌入式处理器市场上占据75%的份额。

ARM作为嵌入式系统的处理器，具有低电压，低功耗和高集成度等优点，并具有开放性和可扩充性。事实上，ARM内核已成为嵌入式系统首选的处理器内核。目前国内已有大量基于ARM的嵌入式实时操作系统研究与实现的论文。由此可见，ARM对嵌入式实时操作系统的支持相当完备，利用ARM架构开发嵌入式操作系统是合理而有效的。

1.2.1 ARM微处理器的特点

采用RISC架构的ARM微处理器一般具有如下特点

- 体积小、低功耗、低成本、高性能
- 支持位双指令集,能很好的兼容位器件
- 大量使用寄存器,指令执行速度更快
- 大多数数据操作都在寄存器中完成
- 寻址方式灵活简单,执行效率高
- 指令长度固定

1.2.2 ARM 微处理器的结构

传统的CISC（Complex Instruction Set Computer，复杂指令集计算机）结构有其固有的缺点,即随着计算机技术的发展而不断引入新的复杂的指令集,为支持这些新增的指令,计算机的体系结构会越来越复杂,然而,在CISC指令集的各种指令中,其使用频率却相差悬殊,只有大约十条指令会被反复使用,占整个程序代码的90%以上。而余下的指令却不被经常使用,在程序设计中只占5%左右。调查显示，在x86指令集体系中，使用频率最高的指令及其频率如下：

执行频率排序	80X86指令	指令执行频率
1	Load	22%
2	条件分支	20%
3	比较	16%
4	Store	12%
5	加	8%
6	与	6%
7	减	5%
8	寄存器—寄存器间数据移动	4%
9	调用	1%
10	返回	1%
合 计		96%

显然,传统的CISC是不太合理的。基于上述不合理性,美国加州大学伯克利分校提出了RISC（Reduced Instruction Set Computer，精简指令集计算机）的概念，RISC并非只是简单地去减少指令,而是把着眼点放在了如何使计算机的结构更加简单合理上，以提高运算速度。ARM微处理器采取RISC结构，优先选取使用频率最高的简单指令,避免复杂指令并将指令长度固定,从而减少指令格式和寻址方式的种类。[2]。此外，ARM架构中的RISC架构可以减小工作量，比较适合我们在此基础上编写操作系统，不会造成太过繁琐的操作，能有效实现目标。

1.2.3 ARM微处理器的寄存器结构和工作模式

ARM 体系结构共有 37 个 32 位寄存器：1 个专用的程序计数器（PC），1 个专用的当前程序状态寄存器（CPSR），5 个备份程序状态寄存器（SPSR）和 30 个通用寄存器，其中通用寄存器分为7 个组，根据处理器不同的模式来访问不同组的通用寄存器，每种处理器模式，除了 FIQ 模式，能够访问一组特定的通用寄存器（r0-r12），一个特定模式的堆栈指针（r13），一个特定模式的连接寄存器（r14），程序计数器（r15）以及 CPSR 与该模式下的 SPSR（不包括系统与用户模式）。

ARM 寄存器图如下：

ARM 系统与用户 模式	ARM 管理 模式	ARM 中断 模式	ARM 快速中断 模式	ARM 中止 模式	ARM 未定义 模式
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8	r8	r8_fiq	r8	r8
r9	r9	r9	r9_fiq	r9	r9
r10	r10	r10	r10_fiq	r10	r10
r11	r11	r11	r11_fiq	r11	r11
r12	r12	r12	r12_fiq	r12	r12
r13(sp)	r13_svc	r13_irq	r13_fiq	r13_abt	r13_und
r14(lr)	r14_svc	r14_irq	r14_fiq	r14_abt	r14_und
r15(pc)	r15	r15	r15	r15	r15
cpsr	cpsr_svc spsr_svc	cpsr_irq spsr_irq	cpsr_fiq spsr_fiq	cpsr_abt spsr_abt	cpsr_und spsr_und

ARM 处理器包含七种处理器工作模式-系统模式、用户模式、管理模式、中断模式、快速中断模式、中止模式、未定义模式，在每一种处理器模式下均有一组相应的寄存器与之对应。即在任意一种处理器模式下,可访问的寄存器包括15个通用寄存器（R0-R14）,1至2个状态寄存器和程序计数器。这种多工作模式与寄存器分配使得在ARM被用来开发嵌入式实时操作系统时显得得心应手。在所有的寄存器中,有些是在7种处理器模式下共用的同一个物理寄存器,而有些寄存器则是在不同的处理器模式下有不同的物理寄存器。并且，当前的 CPSR 寄存器低五位决定现在所处的模式。通常来说可以通过对 CPSR 的后五位赋值来改变 CPU 的工作状态，但是在实际的应用场合 CPU 工作状态的改变往往是外界触发的，比如中断触发、异常等。在每种处理器模式下，将会有特定的寄存器可见。比如在嵌入式操作系统开发中系统管理模式看到得 r13 是 r13_svc，当中断发生后，中断处理程序中同样使用 r13，这时r13 是指 r13_irq 这个寄存器，而切换前的 r13_svc 就不可见了。[3]

1.2.4 ARM微处理器的指令结构

ARM微处理器的在较新的体系结构中支持两种指令集：ARM指令集和Thumb指令集。其中,ARM指令为32位的长度,Thumb指令为16位长度。Thumb指令集为ARM指令集的功能子集,但与等价的代码相比较,可节省30%~40%以上的存储空间,同时具备32位代码的所有优点。这样就为嵌入式操作系统的开发节省了很多空间。

2.技术依据

根据调研报告中所呈现的内容和Rust对嵌入式系统的支持，可以得出利用Rust语言来编写操作系统内核从理论上来讲是切实可行的，下面就实际情况来分析用Rust编写操作系统内核的技术可操作

性。

2.1 重新编写的操作系统的规模

在现阶段，根据操作系统的规模，我们可以把可用Rust实现的操作系统内核分为以下几类：

1. 完备的系统内核
2. 普通微内核
3. 嵌入式微内核
4. 嵌入式实时微内核

以上四种不同类型的操作系统内核在规模上有着显著的区别，下面从工程量和实际意义的角度对这四种方案进行考量：

首先，从完备内核和微内核的代码量的角度考虑，显见，方案二、三、四的代码规模远小于方案一（考虑到许多额外功能，完备内核的代码量往往是微内核的数倍，从本次实验的角度来看比较难以实现），从工作时间的角度考虑，优先从方案二、三、四中做出选择，以减少工作量。

其次，从实际意义的角度考虑，方案二、三、四实现的都是微内核，其工作量大致相当，但是考虑到当下嵌入式系统的广泛应用和现有的Nordic开发板，编写嵌入式微内核要比普通微内核更具有实际意义。而且，为充分利用提供的开发版所具有的蓝牙功能，有必要实现嵌入式实时系统，从而更好地发挥开发板的功能。因此，优先选择编写嵌入式实时微内核。

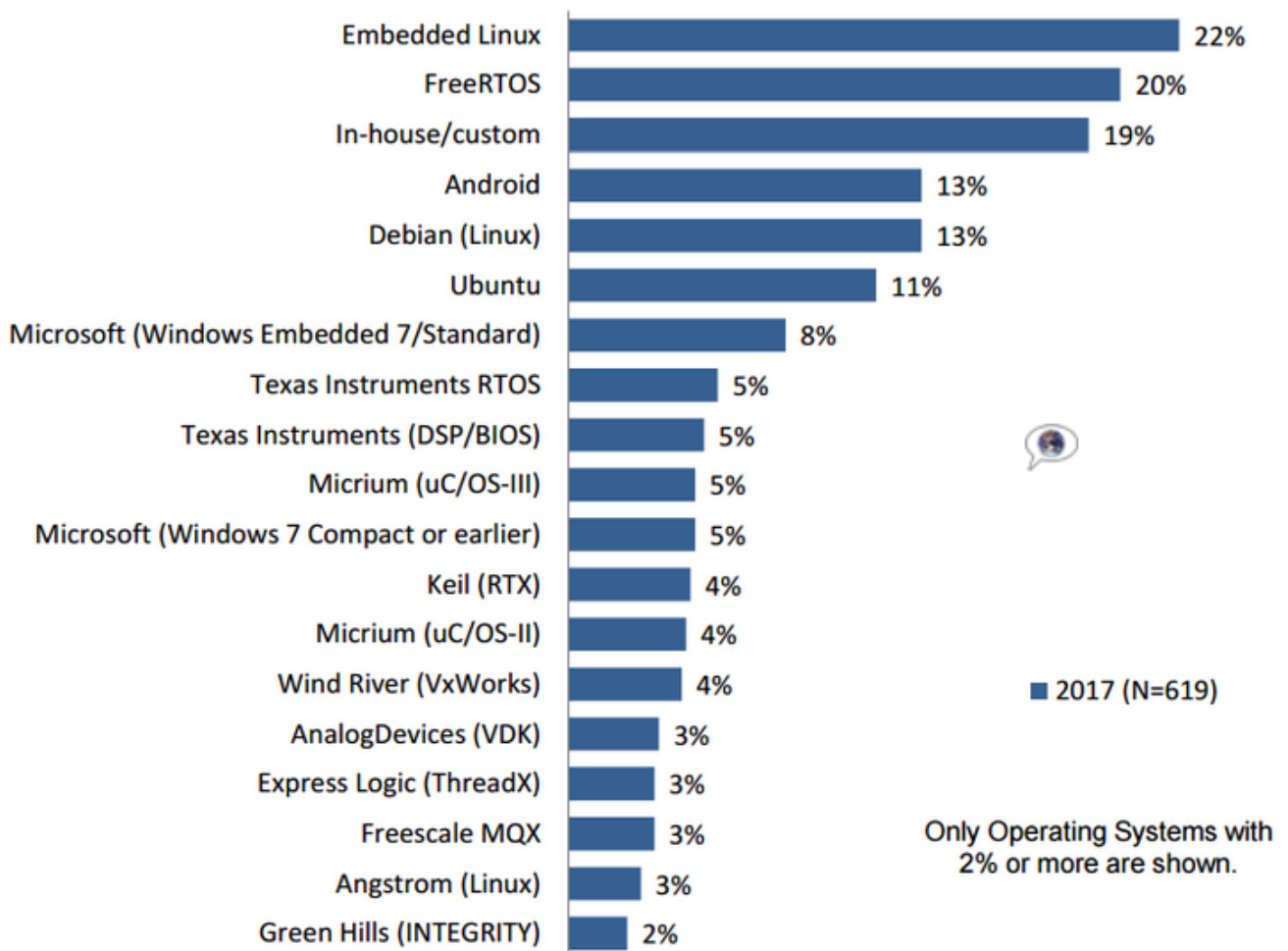
在近一个月的调研和反复的讨论后，对比了现存的众多RTOS，并分析了其理论可行性和实际可操作性后，我们最终决定选择freeRTOS作为我们的参考源码。

2.2 freeRTOS 实时嵌入式系统简介

2.2.1 优势

选择freeRTOS的理由有如下几点：

1. 免费且开源，在其官网上有所有版本的源代码下载，有助于我们增强对RTOS的理解，也便于我们对其进行裁剪，优化。
2. 被大量的嵌入式开发者认可，并且使用人数保持了高速增长的趋势，根据统计数据显示，freeRTOS在全球的嵌入式系统中，在过去的几年内，稳居前五，有若干年甚至达到了榜首，每年都约有20%的嵌入式开发者基于其进行开发。下图为2017年的统计数据：



2017年RTOS使用榜

3. 文档齐全，作为一个开源项目，在其官网上可以找到所需的任何文档，这对于深入理解嵌入式操作系统有着巨大的帮助。
4. 相比于一些其他的大型嵌入式系统而言，freeRTOS作为轻量级OS较为简单，但是在嵌入式操作系统所遇到的绝大多数应用场景下，freeRTOS都可以满足需求。

2.2.2 freeRTOS的基本组成及功能

为了增强我们对freeRTOS的了解。我们在其官方文档中，寻找了对freeRTOS的整体架构的详细介绍。从宏观上看，freeRTOS可分为以下几个模块：

- 任务管理
- 队列管理
- 中断管理
- 资源管理
- 内存管理
- 错误排查

下面对其分别进行简要的介绍：

1. 任务管理

在freeRTOS中，每个执行的线程都被成为任务。所以任务管理是freeRTOS的基础。

freeRTOS作为硬实时多任务系统，任务之间的调度与切换显得极为重要。所以在freeRTOS中，任务都是由一个函数建立，在创建任务的过程中，可以设置其相关参数和优先级等。每个任务都被赋予了一个优先级，系统基于优先级进行执行任务的选择，同时系统又设置了一系列的规则，以保证不会有任务由于优先级过低而一直不被执行。

2. 队列管理

基于freeRTOS的应用程序，由上文可知，都是一组相互独立的任务构成的，但是只由一个一个小任务构成的程序是无法实现复杂的功能的，所以在其之间一定有一个使其相互通信的工具，这个工具正是队列。

3. 中断管理

嵌入式实时系统需要对整个系统环境产生的事件作出反应。如，以太网外围部件收到了一个数据包(事件)，需要送到TCP/IP 协议栈进行处理(反应)。更复杂的系统需要处理来自各种源头产生的事件，这些事件对处理时间和响应时间都有不同的要求。在各种情况下，都需要作出合理的判断，以达到最佳事件处理的实现策略。中断管理正是对其负责。

4. 资源管理

多任务系统中存在一种潜在的风险。当一个任务在使用某个资源的过程中，即还没有完全结束对资源的访问时，便被切出运行态，使得资源处于非一致，不完整的状态。如果这个时候有另一个任务或者中断来访问这个资源，则会导致数据损坏或是其它相似的错误。资源管理模块主要处理的是这种问题。

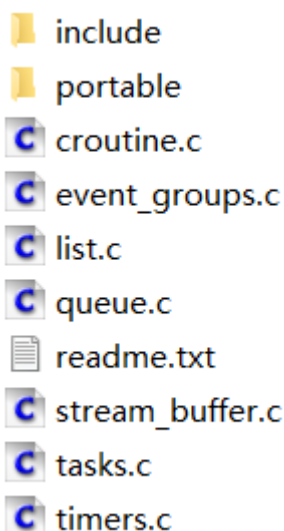
5. 内存管理

不同的嵌入式系统具有不同的内存配置和时间要求。所以单一的内存分配算法只可能适合部分应用程序。因此，FreeRTOS 将内存分配作为可移植层面(相对于基本的内核代码部分而言)。这使得不同的应用程序可以提供适合自身的具体实现。freeRTOS提供了操作内存的基本函数，并且给出了五种内存分配的样例（10.0.1版本）。

6. 错误排查

错误排查主要是为了方便新人用户，迅速的解决自己容易遇到的问题。其中最主要的是栈溢出的问题。

最后我们又大概浏览了其源代码，所包含的内容与上述内容大致相符。下图为freeRTOS截至目前，最新版本（10.0.1）的源代码。其中，portable中为大量的可移植程序的样例。

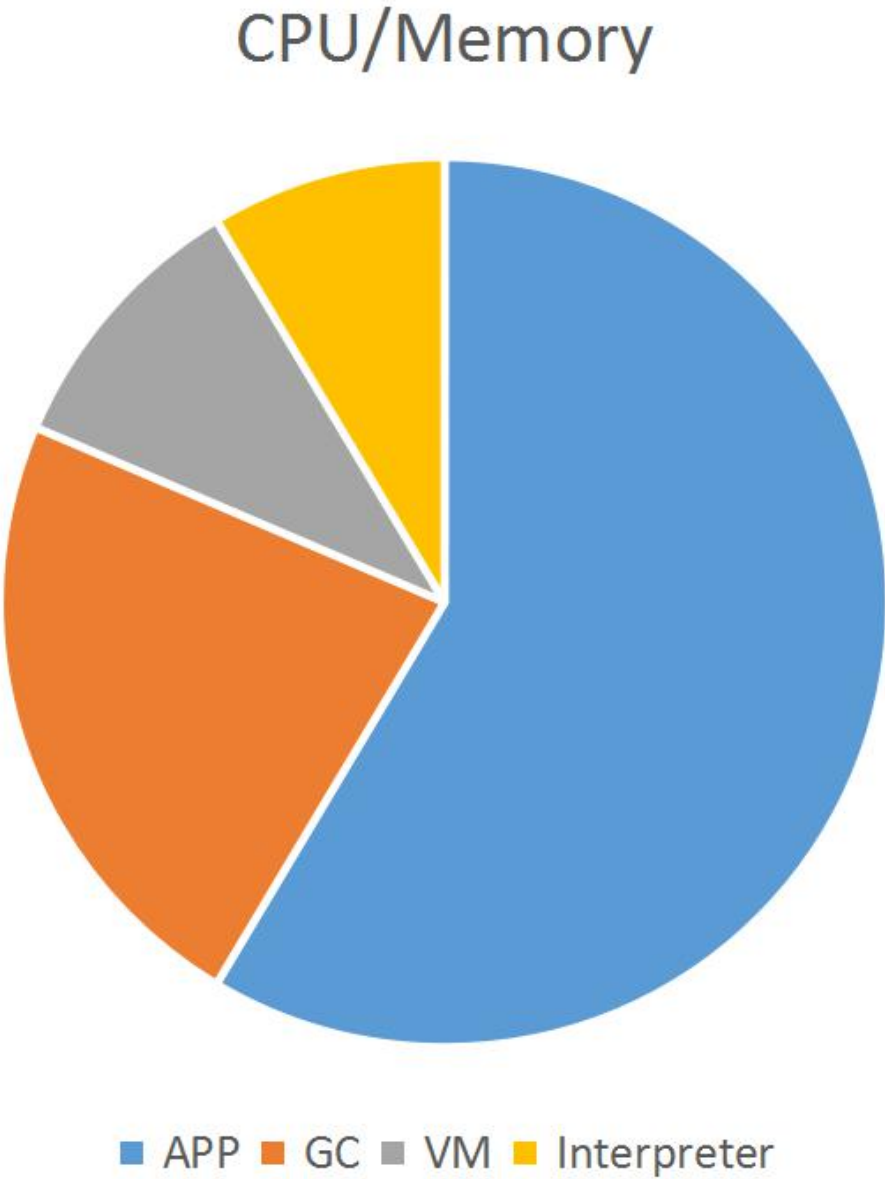


- include
- portable
- croutine.c
- event_groups.c
- list.c
- queue.c
- readme.txt
- stream_buffer.c
- tasks.c
- timers.c

3.创新点

由于近年来系统编程、物联网、数据中心的需求越来越广泛，对CPU和内存的高效利用、系统安全及内存安全提出了巨大的挑战。这就要求系统和应用软件要高效利用硬件，而程序运行在VM上，或后台跑GC便会白白浪费了宝贵的CPU和内存资源，于是同时拥有很强的控制性和很强的安全性的Rust语言便开启了一个对传统革新的契机。

在这样的社会背景之下，我们的项目基于Rust语言的独特性和相较于传统的C语言和C++的创新之处，提出了与传统嵌入式系统开发不同的创新思路，这其中的核心部分即Rust语言独特的优越性。Rust语言具有零运行时、没有垃圾回收机制（GC）、内存安全等创新点，特别适合于嵌入式操作系统的开发。而且，目前使用Rust开发的项目相对较少，我们使用Rust重写freeRTOS将是一个不小的创新，也许会对日后真正的Rust商业级嵌入式操作系统项目带来一定的帮助。



图：不同程序的内存/CPU占用情况

4. 项目设计

4.1 重写操作系统的路径选择

在确定了重写的操作系统规模之后，目前来看，若要利用Rust来重写freeRTOS的内核，有以下几条路可以走：

1. 将一切现有内核源码推倒重来，根据Rust自身的高安全性的特点来编写内核源码。
2. 根据已有的嵌入式系统的c语言源码，针对C语言的缺点来用Rust改写代码实现原来的功能，在已有的基础上重构内核。
3. 通过了解已有的嵌入式操作系统内核必要的模块，并利用Rust来重写这些函数和模块。

以上三条路径都能实现目标，但是显然，方案一、方案三的难度系数显然远大于方案二，但是其可靠性也得到了充分的保障，因为从语法和编译的角度来说，C代码和Rust代码的区别较大，改写时容易产生错误。由于Rust对于程序可靠性的要求较高，所以在编译时，一些c语言能够编译通过的程序改写为Rust后可能就无法通过（除非使用unsafe代码块，但这就失去了改写的意义），如在处理数组越界问题上，Rust的语法就比C要严格得多。因此，对于代码在语法上的简单改写很可能会引起编译上的问题，造成不必要的浪费。

但是，完全重写内核中的每一个函数的工作量是难以想象的，嵌入式操作系统是一个工程量相当巨大的项目，其内核中包含了上万行代码，考虑到团队中大部分人对于Rust语言的熟悉程度和未来预计的工作时间，将整个操作系统复现出来的难度相当大，所以方案一的可行性较低。

对于方案三来说，阅读源码并重写函数在很大程度上降低了工作难度。但是如上所述，即便只是单个函数，其代码量也是相当巨大的，以FreeRTOSv10.0.1为例，在目录FreeRTOS\Demo\AVR32_UC3下的main.c其代码量就在500行以上（在整个内核源码中，此函数的代码量实际上还是偏小的），而且内核调用的函数中有很多算法是目前的课程所没有涉及到的，所以完全重写函数也是不现实的。

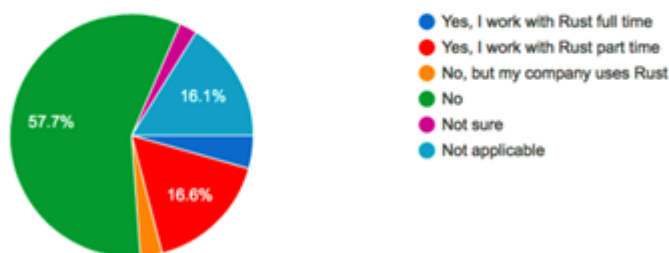
综上所述，方案一、方案三都因为其难度过大而被否决，故下面就方案二的可行性进行讨论：

首先，相比于其他两个方案，方案二的工作已经小了很多。由于Rust的编程特性和C有着共同之处（这一点在调研报告中有所提及），所以在C的基础上修改成Rust语言较为简便，同时也可以加深两种语言之间的比较，从而加深对于Rust的独特性的认识和理解。

其次，Rust有与C语言匹配的接口，和C语言有着很好的兼容性。所以，有一些次要模块可以直接复用C语言的源码，并在Rust编写的核心部分调用C语言接口，从而简化过程。而且，混合语言编程也是目前的编程发展方向之一，在Rust语言的年度调查报告中也显示，相当一部分企业员工使用Rust和其他语言进行混合编程，占总体人数的16.6%，因此，采用Rust和C混合编程的做法是可行而高效的。

Do you or your company use Rust at work?

3,581 responses



再次，阅读C语言的源码可以加深对于操作系统执行过程和相关算法的理解。要想重写操作系统，首先必须得理解操作系统，而通过阅读C语言源码，可以更好地理解操作系统的组成和相关功能的实现，从而更快更好地写出rust版本的嵌入式操作系统。

最后，如上文所说，根据The University of Stanford所新开设的操作系统课程cs140e的相关课程安排[5]可以看出，Stanford的课程安排中，将Rust和C语言的对比学习放在课程作业的第一项，旨在让学生建立起对于Rust的语言特点的整体认识并将其和C语言进行比较。而且，从课程难度来说，cs140e的整体课程和作业难度和本课程大作业难度相近。因此，将cs140e的课程要求和实验内容作为参照，选择嵌入式系统的c语言代码进行改写比较符合目前我们的编程水平和算法知识，是可行性较高的一个方案。

综上所述，虽然方案二存在着诸如编译难通过等问题，但是其相对较小的工作量、易于实现的接口、带来的对于操作系统更为深刻的理解，使我们仍然倾向于方案二，即根据已有的嵌入式操作系统的C语言源码，通过针对C语言缺点和漏洞来编写代码，在已有的基础上重构内核。

因此，我们决定利用Rust来重构一个嵌入式实时微内核，并且通过阅读和改写相关的C语言源码来实现。

4.2 具体实现--Nordic开发板

4.2.1 开发板简介

根据与导师的交流，我们打算利用导师提供的Nordic nRF52840进行开发工作。nRF52840开发板是由Nordic公司生产的，具有高级多协议特征的系统级芯片。它支持蓝牙五，搭载有ARM架构的Cortex-M4F 处理器，具有1MB flash 和 256KB RAM，可以更好地支持有着复杂算法指令的高级程序，并且可以实现包括省电蓝牙模式在内的多协议进程。而且，nRF52840开发板提供了包括NFC，USB在内的丰富的外设接口。更为重要的是，在面对安全性这个物联网中至关重要的属性时，开发板具有高端安全功能，可通过ARM CryptoCEII加密系统和完整的AES 128位加密套件实现最佳安全性。

4.2.2 开发板优势和特点

Nordic nRF52840开发板最大的优势，无疑是对蓝牙五的硬件支持。相比于蓝牙四，蓝牙五拥有2倍速度，4倍距离（约300米），8倍广播数据量，更低的能耗，甚至可以与wifi配合实现室内的较为精准的定位。开发板借助蓝牙五，瞄准了物联网（IoT）这个庞大的市场，可应用在多种场景，如高性能的可穿戴设备，高可靠性的移动支付设备，智慧家庭，智慧城市（由于蓝牙五的超远距离）等等。

4.3 目标和展望

在未来的工作中，我们的初级目标是利用Rust语言成功编写一个完整的，安全的，稳定的，快速的嵌入式操作系统，实现系统的基本功能。在此基础上，考虑到开发板蓝牙模块的优势，希望能为系统设计一些符合开发板特性的功能，争取使我们编写的系统与开发板匹配，能充分利用蓝牙功能，实现数据传输和远程控制，使之具有较高的实用价值。我们希望，在物联网这个大潮流中，能做出切实可行的产品，为“万物互联，无处不连”的物联网终极目标作出自己的贡献。

5.参考文献

- [1] Rust 1.6 Brings Stable Support for OS and Embedded Development
- [2] 《嵌入式实时操作系统在ARM系列微处理器上的移植研究》，吴伟，昆明理工大学
- [3] 《嵌入式实时操作系统内核设计与实现》，王云飞，电子科技大学
- [4] 《Rust编程语言 核心优势和核心竞争力》，庄晓立，【QCon】 2016 全球软件开发大会
- [5] [cs140e课程主页](#)
- [6] [Nordic nRF52840官方网站](#)