

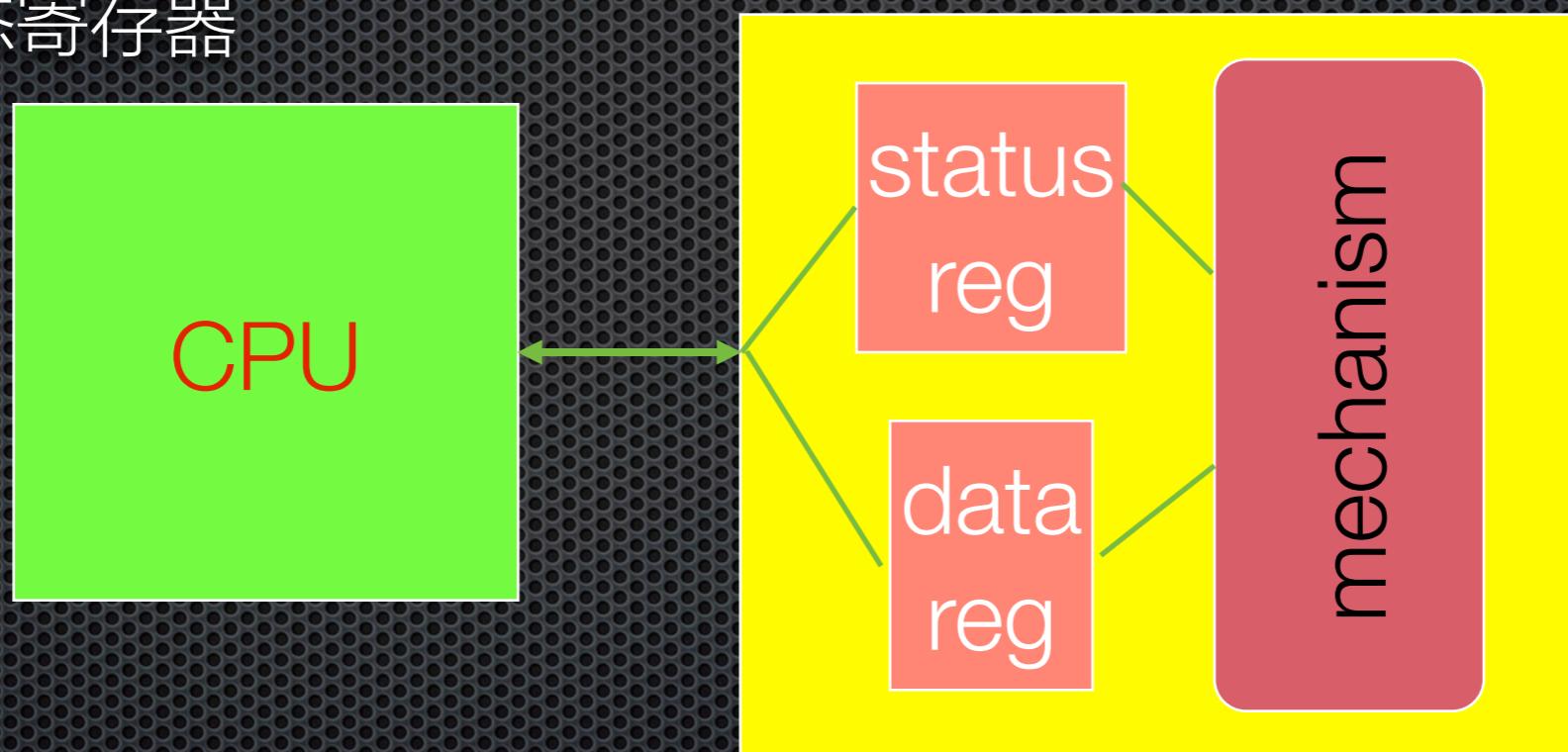
基于总线的计算机系统

目录

- 输入与输出
- 总线

I/O接口与设备

- I/O接口为了能够充当设备与计算机的桥梁，需要多个寄存器
 - 数据寄存器
 - 控制寄存器
 - 状态寄存器



分类 — 从属关系

- 系统设备：操作系统启动时已经在系统中注册的标准设备
 - 例如NOR/NAND闪存，触摸面板等
 - 这些设备的操作系统中有设备驱动程序和管理程序
 - 用户应用程序只需要调用操作系统提供的标准命令或函数就可以使用这些设备
- 用户设备：操作系统启动时未在系统中注册的非标准设备
 - 通常设备驱动程序是由用户提供的。用户必须以某种方式将这些设备的控制权转移到操作系统进行管理
 - 典型的设备包括SD卡、U盘等

分类 — 使用

- 专用设备：同一时间只能被一个进程使用的设备。对于多个并发进程，每个进程使用设备是互斥的。一旦操作系统将设备分配给一个特定的进程，它将被该进程独占，直到该进程使用后释放它
- 共享设备：可被多个进程同时寻址的设备。共享设备必须是可寻址的，并且是随机寻址的。共享设备机制可以提高每个设备的利用率
- 虚拟设备：通过虚拟技术将一台独占设备虚拟成多台逻辑设备，供多个用户进程同时使用，通常把这种经过虚拟的设备称为虚拟设备

类别 — 特征

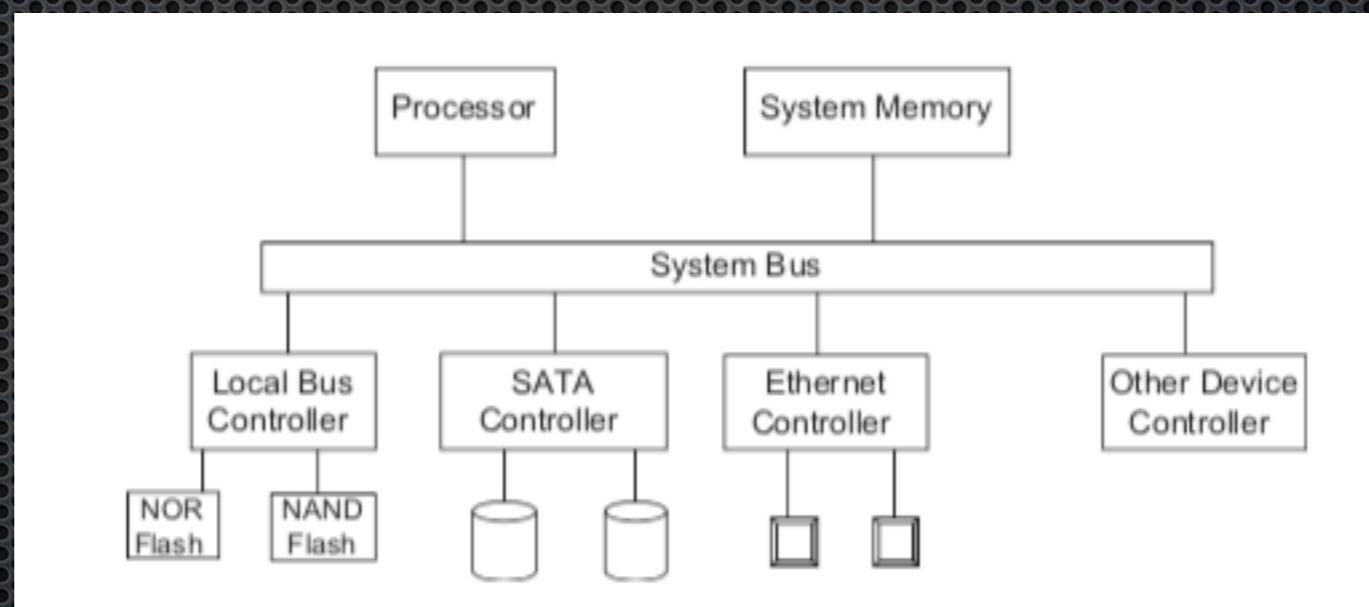
- 存储设备：用于存储信息的设备。嵌入式系统中的典型例子包括硬盘、固态硬盘、NOR/NAND闪存
- I / O设备：
 - 输入设备：输入设备负责将信息从外部输入到内部系统，如触摸面板、条形码扫描仪等
 - 输出设备：输出设备负责将嵌入式系统处理后的信息输出到外部世界，如液晶显示器、扬声器等

类别 — 信息传输单元

- 块设备：这种类型的设备以数据块为单位组织和交换数据
 - 是一种结构化设备
 - 典型的设备是硬盘
 - 在I/O操作中，即使只是一个单字节的读/写，也应该读或写整个数据块
- 字符设备：这类设备以字符单位组织和交换数据
 - 是一种非结构化设备
 - 字符设备有很多种，如串口、触摸面板、打印机等
 - 字符设备的基本特征是传输速率低且不可寻址，当字符设备执行I/O操作时，经常使用中断

I/O设备

- 通常I/O设备由两部分组成
 - 机械部件
 - 电子部件
 - 电子部分称为设备控制器或适配器

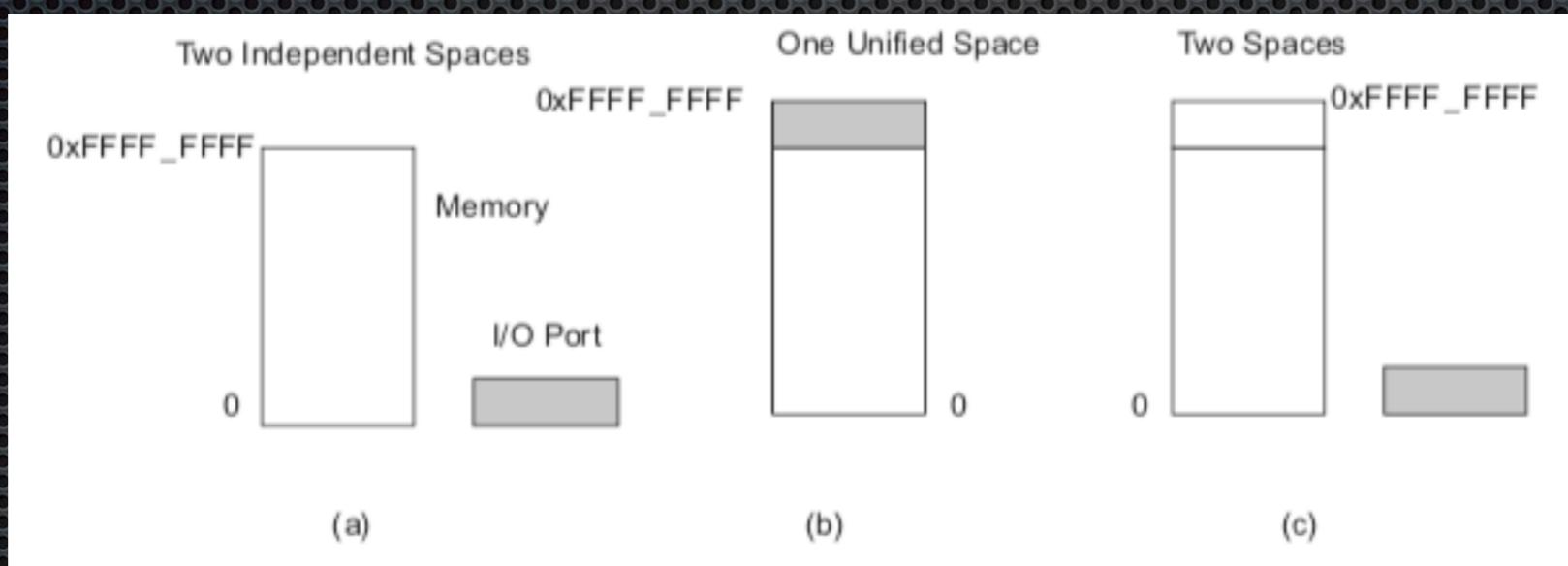


应用: 8251 UART

- 通用异步收发器，Universal asynchronous receiver transmitter (UART)
 - 包括了RS232、RS449、RS423、RS422和RS485等接口标准规范和总线标准规范，即UART是异步串行通信口的总称
 - 规定了通信口的电气特性、传输速率、连接特性和接口的机械特性等内容

可编程I/O

- 在通信过程中选择控制寄存器或数据缓冲区的三种方法
 - 独立I/O端口
 - 内存映射I / O
 - 混合解决方案，混合模型包括内存映射的I/O数据缓冲区和用于控制寄存器的独立I/O端口
- Intel x86提供了in、out指令，大多数其他cpu使用内存映射I/O。



(a) 独立 I/O 和内存空间, (b) 内存映射I/O, (c) 混合方案

独立I/O端口

- 优点：
 - I/O独立编址，不占用内存空间
 - 使用I/O指令，程序清晰，很容易看出是I/O操作还是存储器操作
 - 译码电路比较简单(因为I/O端口的地址空间一般较小，所用地址线也就较少)
- 缺点：
 - 只能用专门的I/O指令，访问端口的方法少

内存映射I/O

- 内存映射I/O的优势可以概括为：
 - 在内存映射I/O模式中，设备控制寄存器只是内存中的变量，可以像其他变量一样在C语言中寻址。因此，I/O设备驱动程序可以完全用C语言编写
 - 在这种模式下，不需要特殊的保护机制来阻止用户进程执行I/O操作
- 内存映射I/O模式的缺点可以概括为：
 - 目前大多数嵌入式处理器都支持内存缓存。缓存设备控制寄存器会导致灾难。为了防止这种情况，必须为硬件提供选择性禁用缓存的功能，这将增加嵌入式系统中硬件和软件的复杂性
 - 如果只有一个地址空间，所有内存模块和所有I/O设备必须检查所有内存引用，以决定响应哪一个，这将严重影响系统性能

ARM 内存映射I/O

- 定义设备地址:

```
DEV1 EQU 0x1000
```

- 读/写代码:

```
LDR r1,#DEV1 ; set up device adrs
```

```
LDR r0,[r1] ; read DEV1
```

```
LDR r0,#8 ; set up value to write
```

```
STR r0,[r1] ; write value to device
```

取/存

- ◆ 传统的高级语言接口:

```
int peek(char *location) {  
    return *location; }
```

```
void poke(char *location, char newval) {  
    (*location) = newval; }
```

忙等IO (Busy/wait)

- 最基本的方式
 - 使用指令测试设备忙闲

```
current_char = mystring;  
while (*current_char != '\0') {  
    poke(OUT_CHAR,*current_char);  
    while (peek(OUT_STATUS) != 0);  
    current_char++;  
}
```

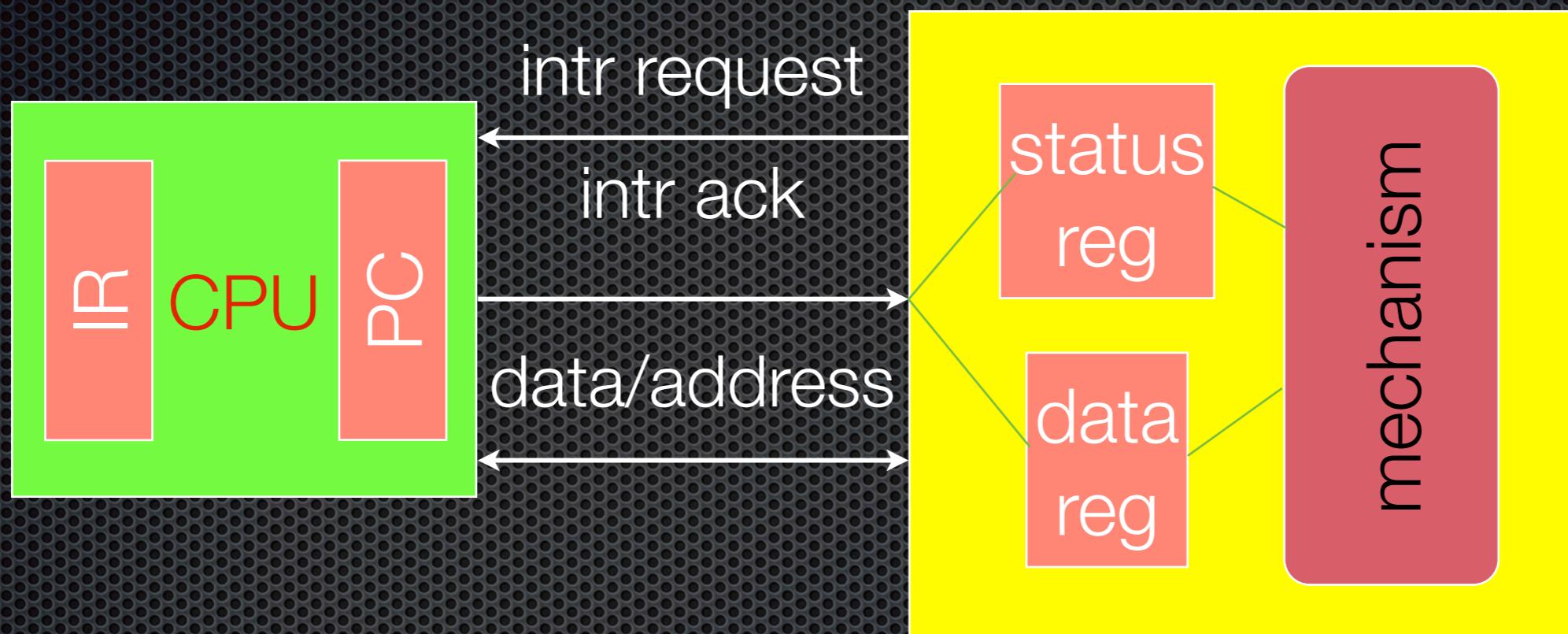
将字符从输入设备复制到输出设备

```
while (TRUE) {  
    /* read */  
    while (peek(IN_STATUS) == 0);  
    achar = (char)peek(IN_DATA);  
    /* write */  
    poke(OUT_DATA, achar);  
    poke(OUT_STATUS, 1);  
    while (peek(OUT_STATUS) != 0);  
}
```

中断I/O

- 忙/等I/O是非常低效的
 - CPU在测试设备时不能做其他工作
 - 很难同时进行I/O
- 中断允许设备改变CPU中的控制流
 - 调用子例程来处理设备

中断接口



中断行为

- 基于子程序调用机制
- 中断强制下一条指令是预定义地址的子程序调用



中断物理接口

- CPU与设备之间通过CPU总线连接
- CPU和设备握手
- 设备发出中断请求
- CPU在能够处理中断时确认中断

例如：字符I/O处理程序

```
void input_handler() {  
    achar = peek(IN_DATA);  
    gotchar = TRUE;  
    poke(IN_STATUS,0);  
}  
  
void output_handler() {  
}
```

例如：中断驱动的主程序

```
main() {  
    while (TRUE) {  
        if (gotchar) {  
            poke(OUT_DATA,achar);  
            poke(OUT_STATUS,1);  
            gotchar = FALSE;  
        }  
    }  
    other processing....  
}
```

例子: 使用缓冲区的中断I/O

- 字符缓冲队列

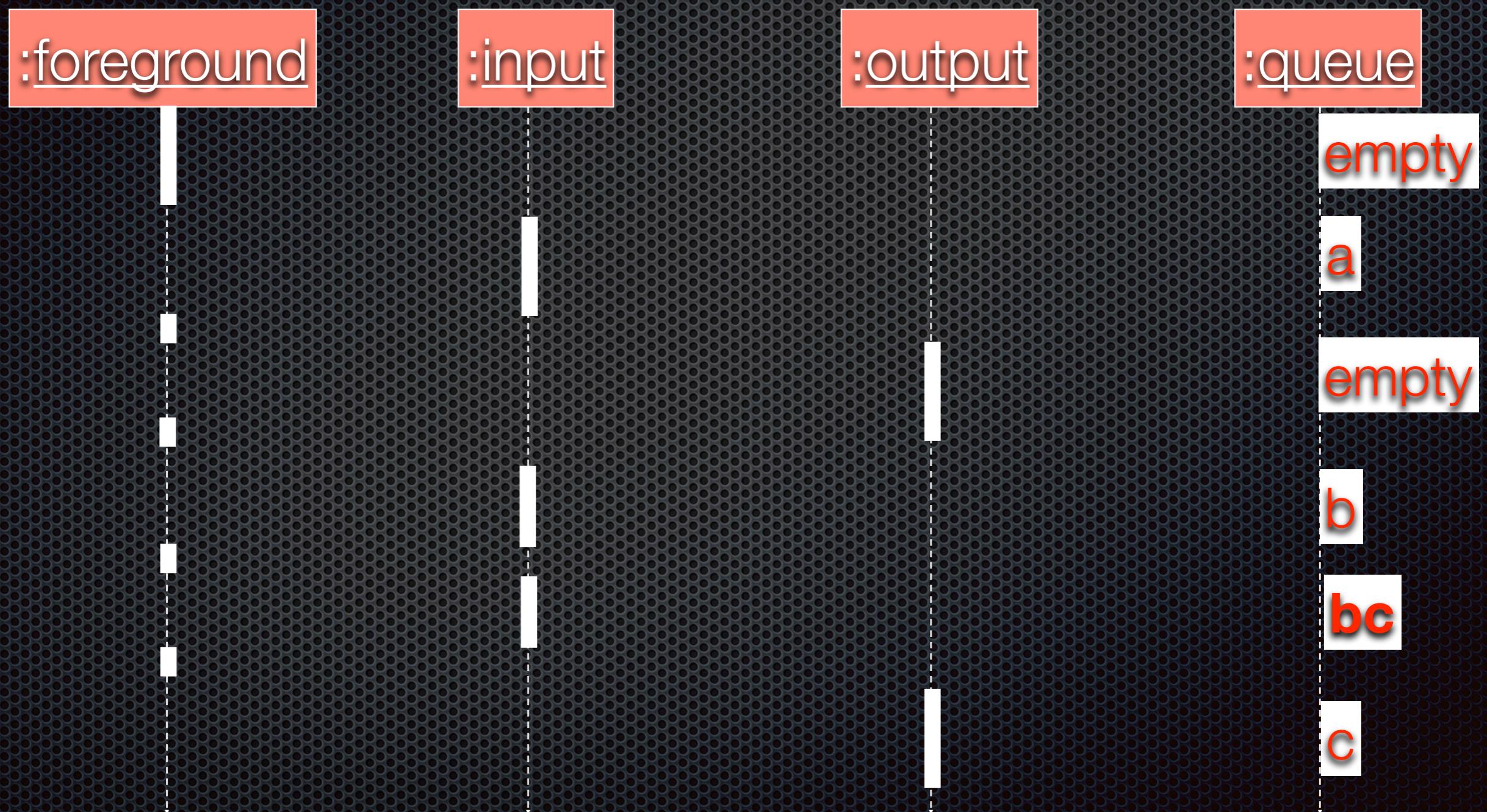


leave one empty slot to allow full buffer to be detected

基于缓冲区的输入处理程序

```
void input_handler() {  
    char achar;  
    if (full_buffer()) error = 1;  
    else {  
        achar = peek(IN_DATA);  
        add_char(achar); }  
        poke(IN_STATUS,0);  
        if (nchars == 1)  
        {  
            poke(OUT_DATA,remove_char());  
            poke(OUT_STATUS,1); }  
    }
```

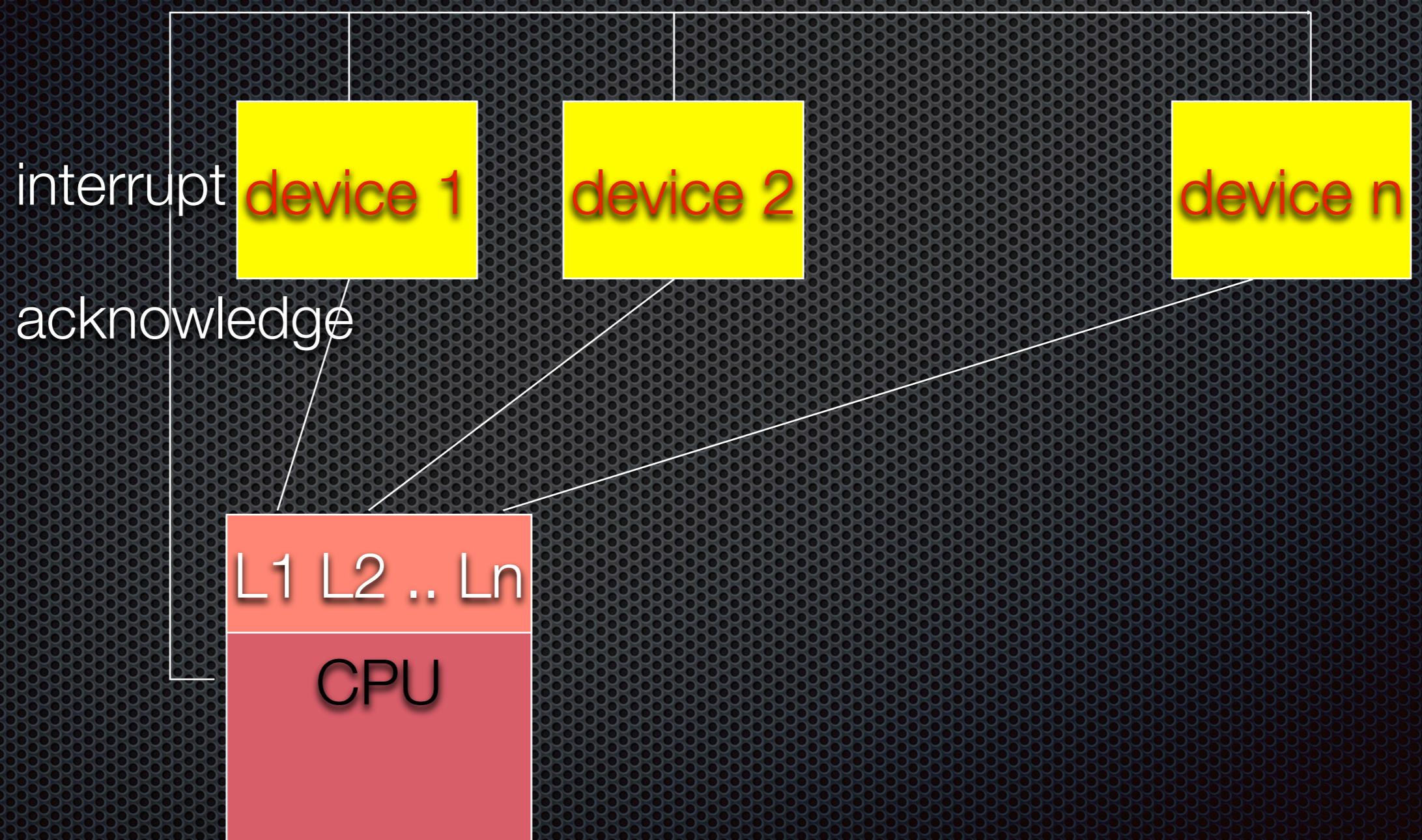
I/O时序图



优先级和向量

- 有两种机制允许我们使中断更灵活具体
 - 优先级：决定哪个中断首先得到CPU
 - 向量：决定每种类型的中断调用什么代码
- 机制是正交的：大多数cpu都提供这两种机制

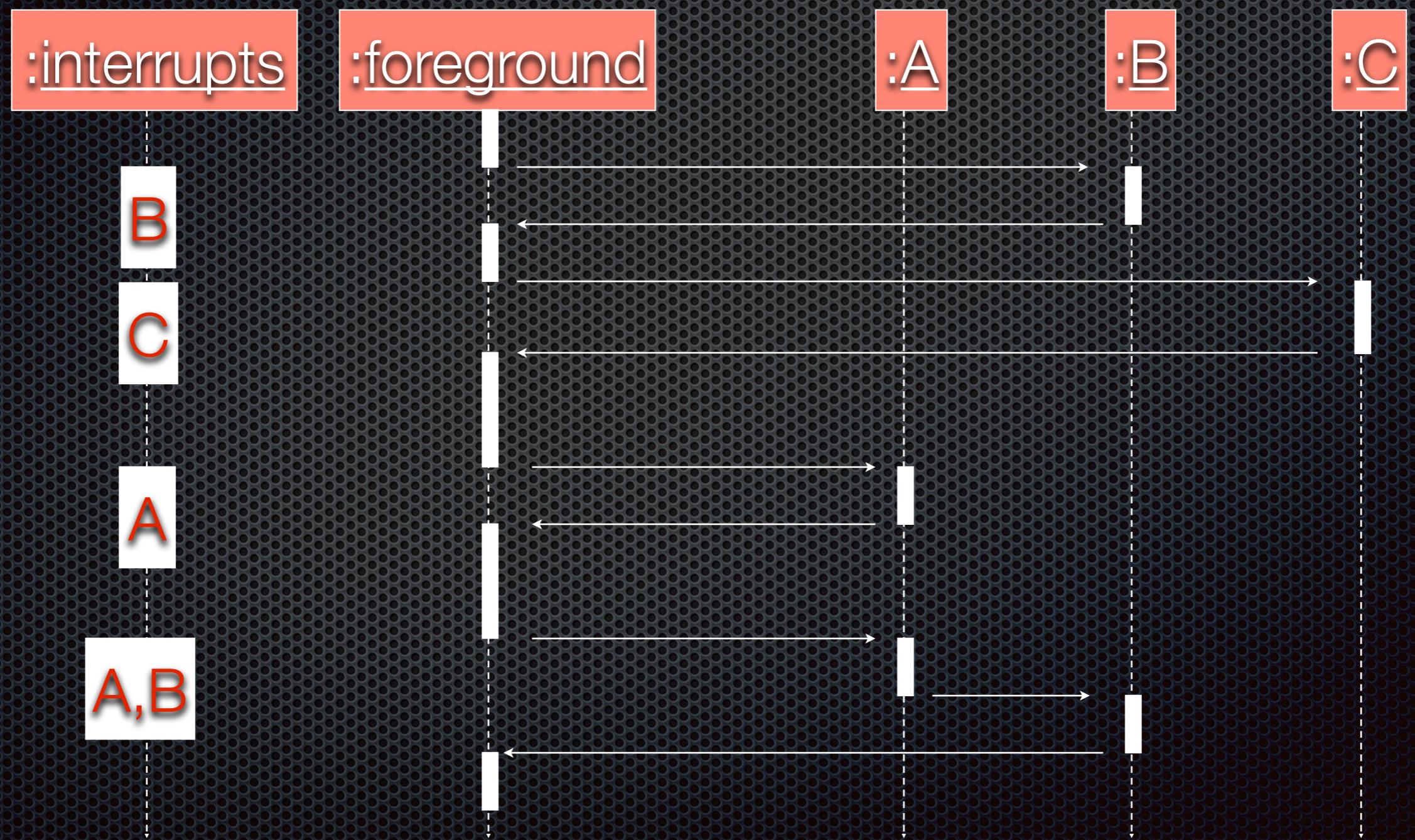
优先级中断



中断优先级

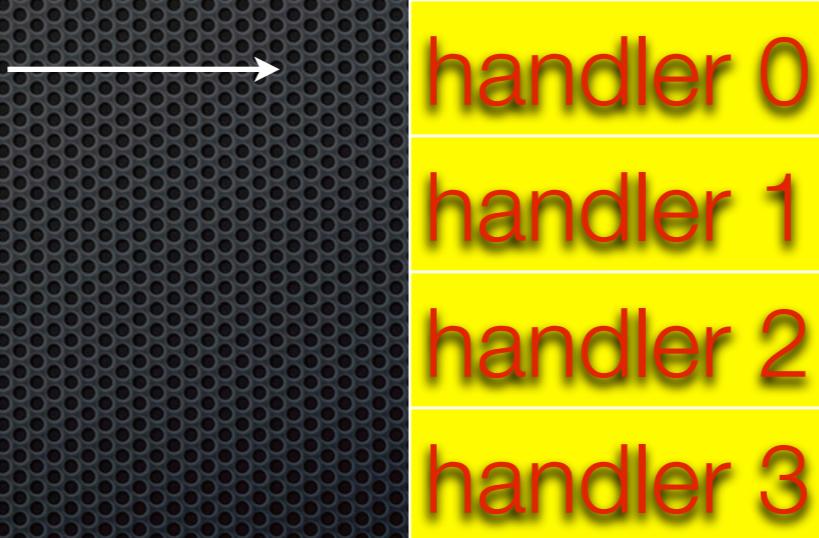
- 屏蔽：在挂起的中断完成之前，不会识别优先级低于当前优先级的中断
- 不可屏蔽中断(NMI)：最高优先级，从不被屏蔽
 - 通常用于断电

例子：优先级 I/O

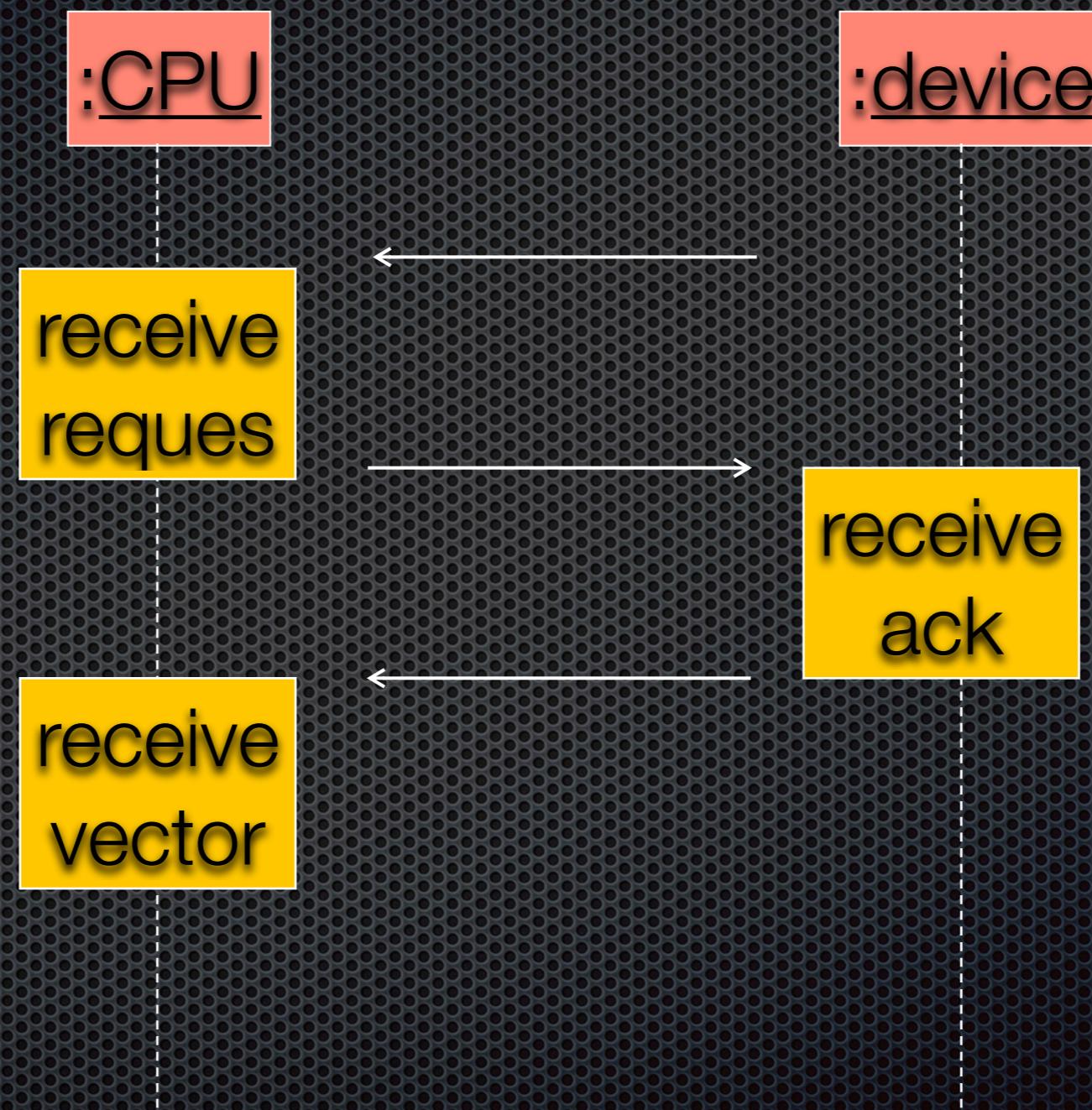


中断向量

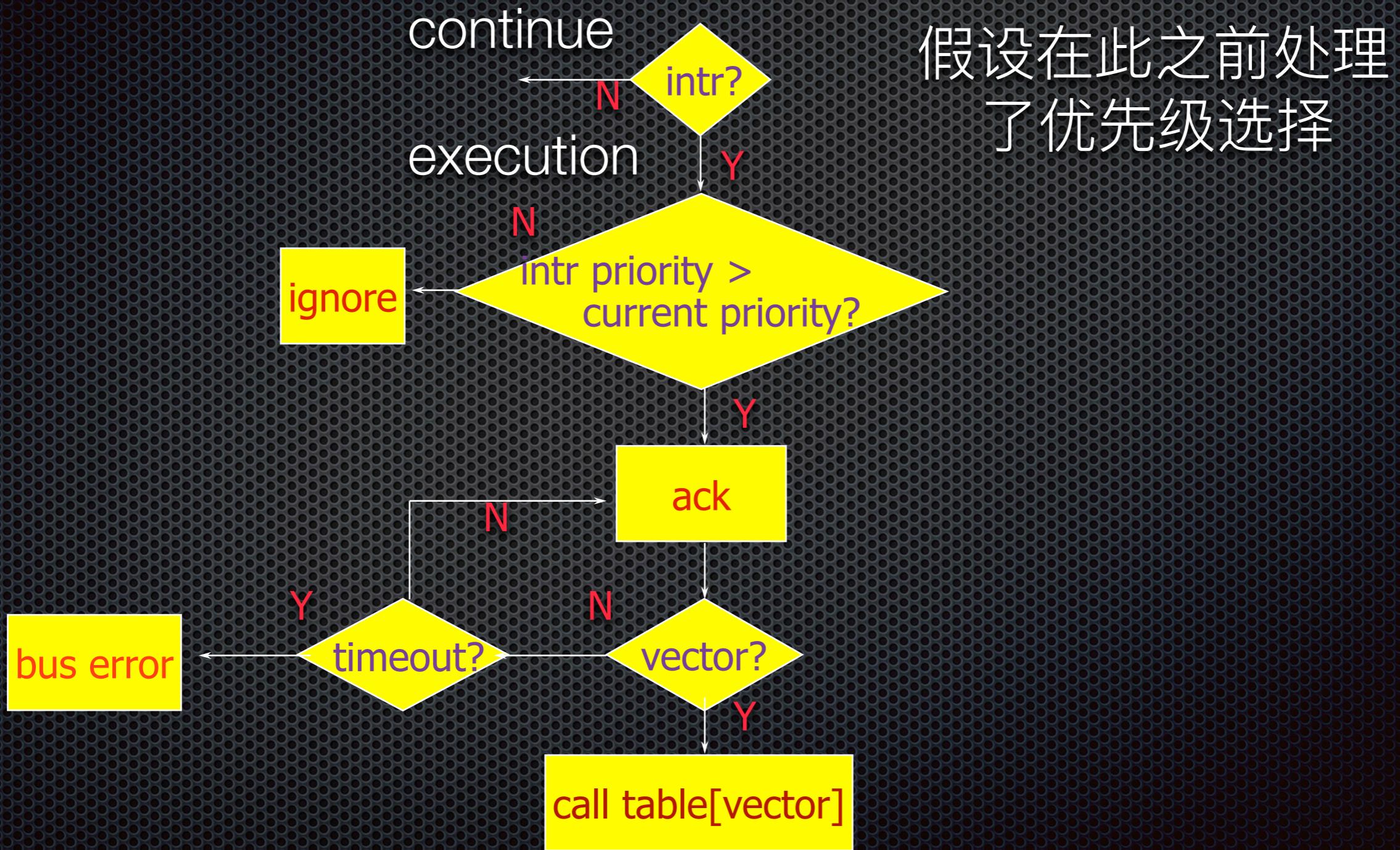
- 允许不同的设备由不同的代码处理
 - 提高灵活性，定义服务于来自设备的请求的中断程序的能力，向量号作为存储在内存中的中断向量表的索引
- 中断向量表：



中断向量获取



通用中断机制



中断序列

- CPU确认请求
- 设备发送向量
- CPU调用中断处理程序
- 软件处理请求
- CPU恢复前台程序状态

中断开销的来源

- 中断处理程序执行时间
- 中断机制开销
- 寄存器保存/恢复
- 流水线相关的开销
- 缓存相关的开销

中断设计指南

- 糟糕的代码很难调试，而糟糕的ISR实际上是不可调试的
- 如何？
 - 首先，在布局中断映射之前，不要考虑为新的嵌入式系统编写任何代码，列出每个中断，并描述程序应该做什么
 - 上述中断设计图是一份预算，预计在什么情况下中断需要花费的时间
 - 粗略估计每个ISR的复杂度
 - ISR的基本原则是保持处理程序简短
 - 当然，短是用时间来衡量的，而不是用代码大小来衡量的。避免循环。避免冗长复杂的指令(重复的动作，可怕的数学，等等)
 - 在ISR中尽快重新启用中断。先做硬件关键和不可重入的事情，然后执行中断启用指令。给其他ISR机会去做他们的事情

中断设计图

	Latency	Max-time	Freq	Description
INT1		1000 μsec	1000 μsec	timer
INT2		100 μsec	100 μsec	send data
INT3		250 μsec	250 μsec	Serial data in
INT4		15 μsec	100 μsec	write tape
NMI	200 μsec	500 μs	once!	System crash

- 用指向空例程的指针填充所有未使用的中断向量

```
Vect_table:  
    dl    start_up          ; power up vector  
    dl    null_isr          ; unused vector  
    dl    null_isr          ; unused vector  
    dl    timer_isr         ; main tick timer ISR  
    dl    serial_in_isr    ; serial receive ISR  
    dl    serial_out_isr   ; serial transmit ISR  
    dl    null_isr          ; unused vector  
    dl    null_isr          ; unused vector  
  
null_isr:  
        jmp   null_isr       ; spurious intr routine  
                                ; set BP here!
```

C语言还是汇编语言？

- 如果程序使用汇编语言，则将转换为大致数量指令的时间。如果平均一条指令需要 x 微秒(取决于时钟速率、等待状态等)，那么很容易得到代码允许复杂度的临界估计
- C语言就更麻烦。事实上，用C语言科学地编写中断处理程序是不可能的！很难想象运行一行C代码要花多长时间。由于每行代码的时间变化很大，甚至无法进行估计
- 字符串比较可能导致运行库调用，其结果完全不可预测
- FOR循环可能需要一些简单的整数比较或大量的处理开销
- 可能会发现一个编译器在中断处理方面慢得可怜。要么试试另一个，要么切换到汇编
- 尽量使用C语言

调试INT/INTA周期

- 设备硬件产生中断脉冲
- 中断控制器(如果有的话)以优先级处理多个同时发生的请求，并向处理器发出单个中断
- CPU响应一个中断确认周期
- 控制器在总线上发出一个中断向量
- CPU读取向量并计算向量所指向的内存中的地址，然后获取这个值
- CPU推入当前上下文，禁用中断，并跳转到ISR

查找丢失的中断

- 可以使用单个上行/下行计数器构建一个小电路，该计数器对每个中断进行计数，并减少每个中断确认的计数。如果计数器总是显示0或1的值，则一切正常
- 一个经验法则将帮助最小化丢失的中断：在最早的安全点重新启用ISR中的中断

避免NMI

- NMI（不可屏蔽的中断）用于电源故障、系统关闭和即将发生的灾难，定时器或UART中断不是
- NMI甚至会破坏编码良好的中断处理程序，因为大多数isr在服务硬件的前几行代码中都是不可重入的。NMI也会阻碍堆栈管理工作。
- NMI与大多数工具的混合效果很差。调试任何ISR-NMI或其他方式都是令人恼火的。很少有工具能很好地在ISR内单步执行和设置断点。

断点问题

- 虽然断点确实是很棒的调试辅助工具，但是对于嵌入式代码就像海森堡的不确定性原理一样
- 通过使用实时trace，这是所有仿真器和一些智能逻辑分析仪都具有的功能。

简易ISR调试

- 调试ISR最快的方法是什么？
 - 不！！！
 - 如果您的ISR只有10或20行代码，肉眼检查一下就可以，不要启动各种复杂和不可预测的工具
 - 保持处理程序的简单和简短

可重入

- 在嵌入式世界中，例程必须满足以下条件才能重入：
 - 它以原子方式使用所有共享变量，除非将每个共享变量分配给函数的特定实例
 - 它不调用不可重入的函数
 - 它不以非原子的方式使用硬件

<http://www.ganssle.com/articles/begincornerent.htm>

原子变量

- 第一条和最后一条规则都使用了“原子”这个词，这个词来自希腊语，意思是“不可分割的”。在计算机世界中，“原子的”是指不能被中断的操作
 - mov ax,bx
 - temp=foobar; temp+=1; foobar=temp;
 - foobar+=1;

```
mov ax,[foobar]  
inc ax  
mov [foobar],ax
```

```
inc [foobar]
```

```
lock inc [foobar]
```

- 规则2告诉我们，调用函数继承了被调用函数的可重入问题
- 规则3是一个独特的隐含警告，硬件看起来很像一个变量；如果处理一个设备需要多个I/O操作，就会出现重入问题

保持代码可重入性

- 消除不可重入代码的最佳选择如下：
 - 第一条经验法则是避免共享变量。全局变量是没完没了的调试问题和代码失败的根源。使用自动变量或动态分配内存。
 - 最常见的方法是在不可重入代码期间禁用中断。
 - 信号量

递归函数

- 如果一个函数调用自己，那么它就是递归的
- 所以所有递归函数都必须是可重入的…但并非所有可重入函数都是递归的

异步硬件/固件

```
int timer_hi;  
  
interrupt timer(){  
    ++timer_hi;}  
  
long timer_read(void){  
    unsigned int low, high;  
  
    low =inword(hardware_register);  
  
    high=timer_hi;  
  
    return (high<<16+low);} 
```

竞态条件

- 设备或系统出现不恰当的执行时序，而得到不正确的结果
- timer_read的竞争条件之一可能是：
 - 读取硬件，然后得到值0Xffff
 - 在从变量timer_hi中获取时间的高十六位数值之前，硬件再次增加，到0x0000
 - 溢出触发中断，ISR运行，此时timer_hi是0001，而不是几纳秒之前的0
 - ISR返回；timer_read例程，不知道发生了中断，只是将新的0001与先前读取的定时器值0Xffff，并返回0x1ffff，一个非常不正确的值

解决方法

- 最简单的方法是在尝试读取计时器之前停止计时器
 - 丢失时间。在此期间关闭中断将消除不必要的任务，但会增加系统延迟和复杂性
- 另一种解决方案是先读取timer_hi变量，然后读取硬件计时器，然后重新读取timer_hi。如果两个变量的值不相同，就会发生中断。迭代，直到两个变量的读值相等
 - 好处是：数据正确，中断保持开启，系统不会丢失计数
 - 缺点是：在重负载的多任务环境中，例程可能会循环相当长的时间才获得两个相同的读数，函数的执行时间是不确定的。

- 另一种替代方法是简单地禁用读取前后的中断

```
long timer_read(void){  
    unsigned int low, high;  
    push_interrupt_state;  
    disable_interrupts;  
    low=inword(Timer_register);  
    high=timer_hi;  
    if(inword(timer_overflow))  
    {      ++high;  
          low=inword(timer_register);};  
    pop_interrupt_state;  
    return (((ulong)high)<<16+(ulong)low);  
}
```

目录

- 输入与输出
- 总线

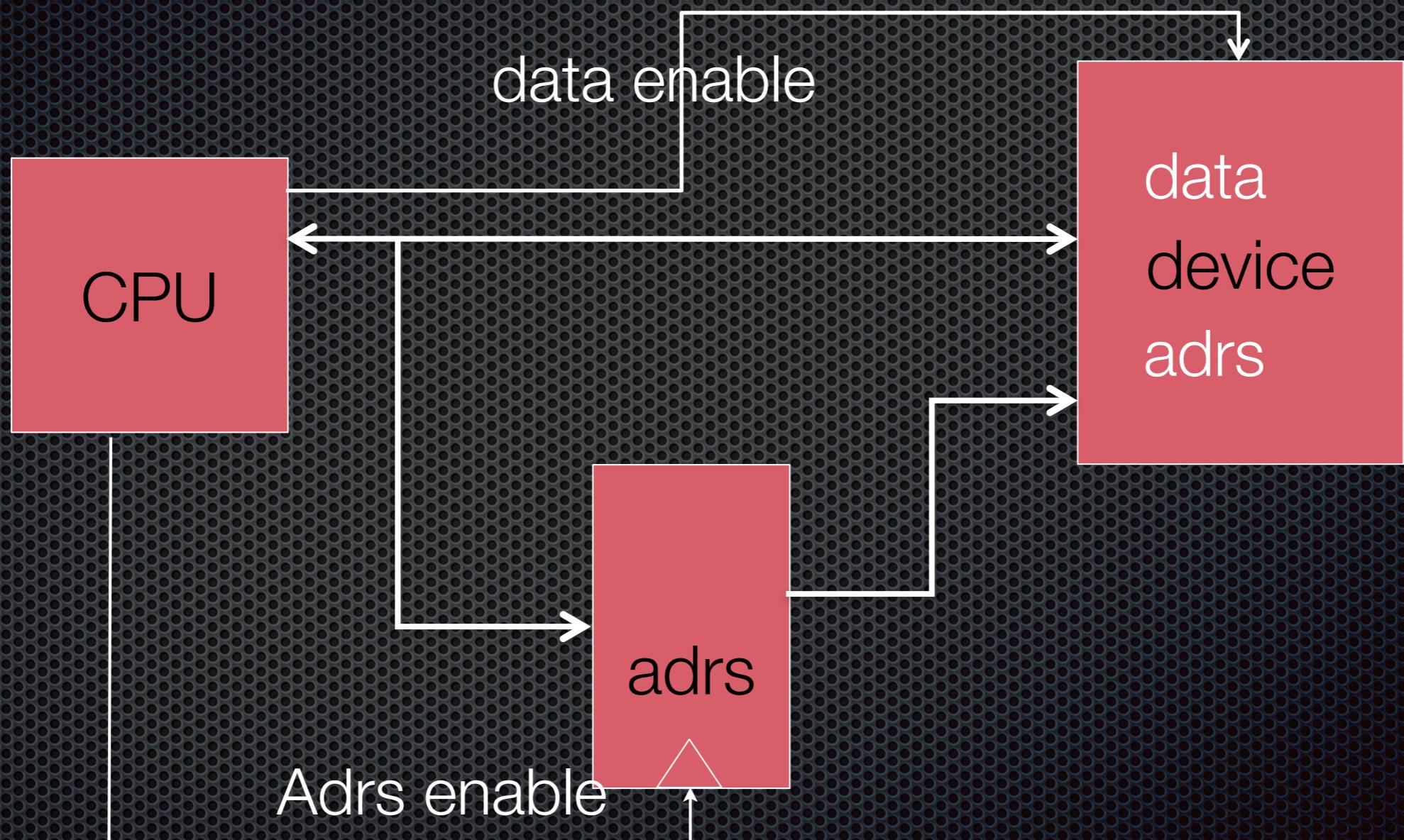
CPU总线

- 总线（Bus）是计算机各种功能部件之间传递信息的公共通信干线
- 总线是：
 - 一组传送线路
 - 相关的通信协议

总线协议

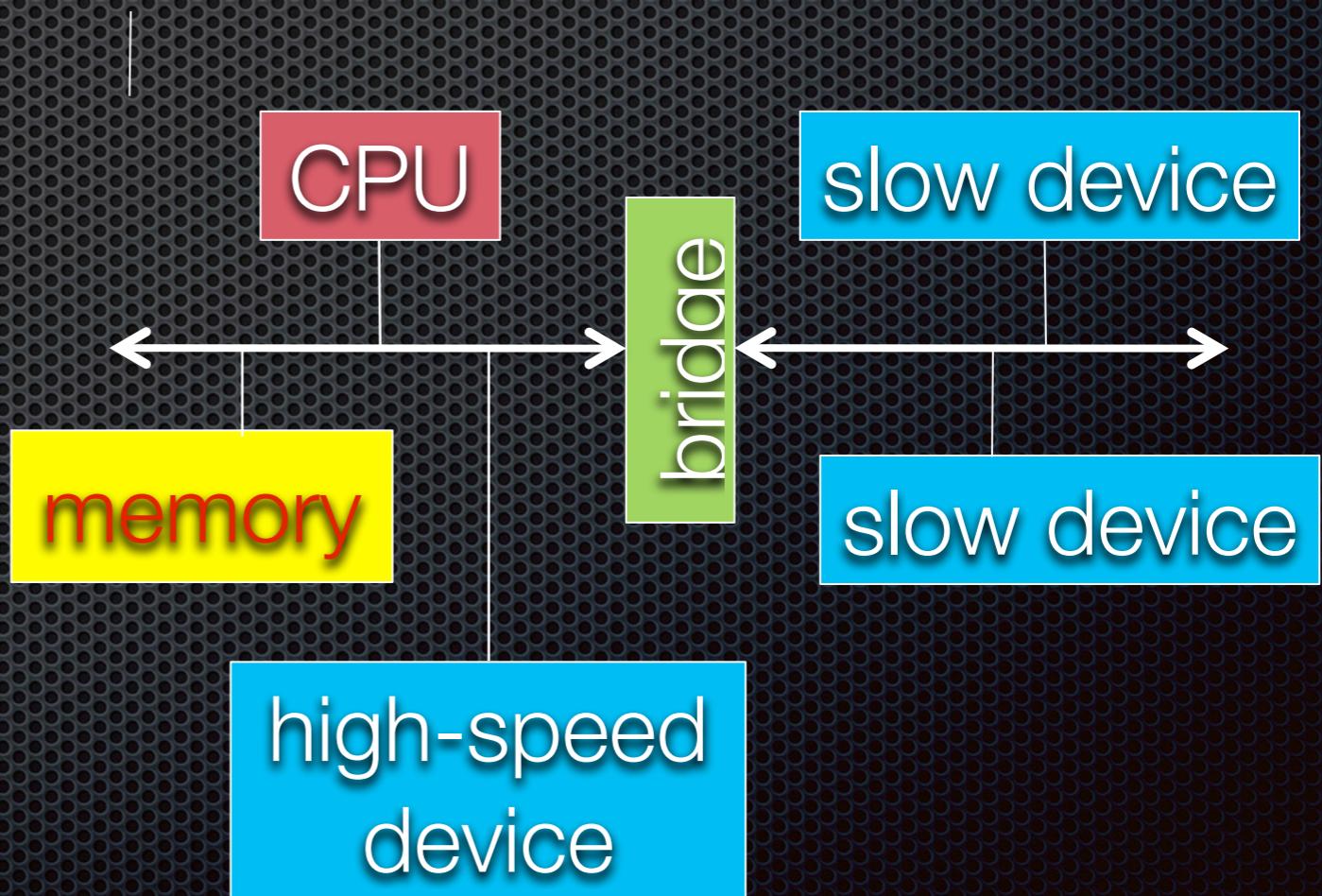
- 总线协议决定设备如何通信
- 总线上的设备经历一系列的状态
- 协议由状态机指定，协议中的每个参与者都有一个状态机
- 可包含异步逻辑行为

总线复用

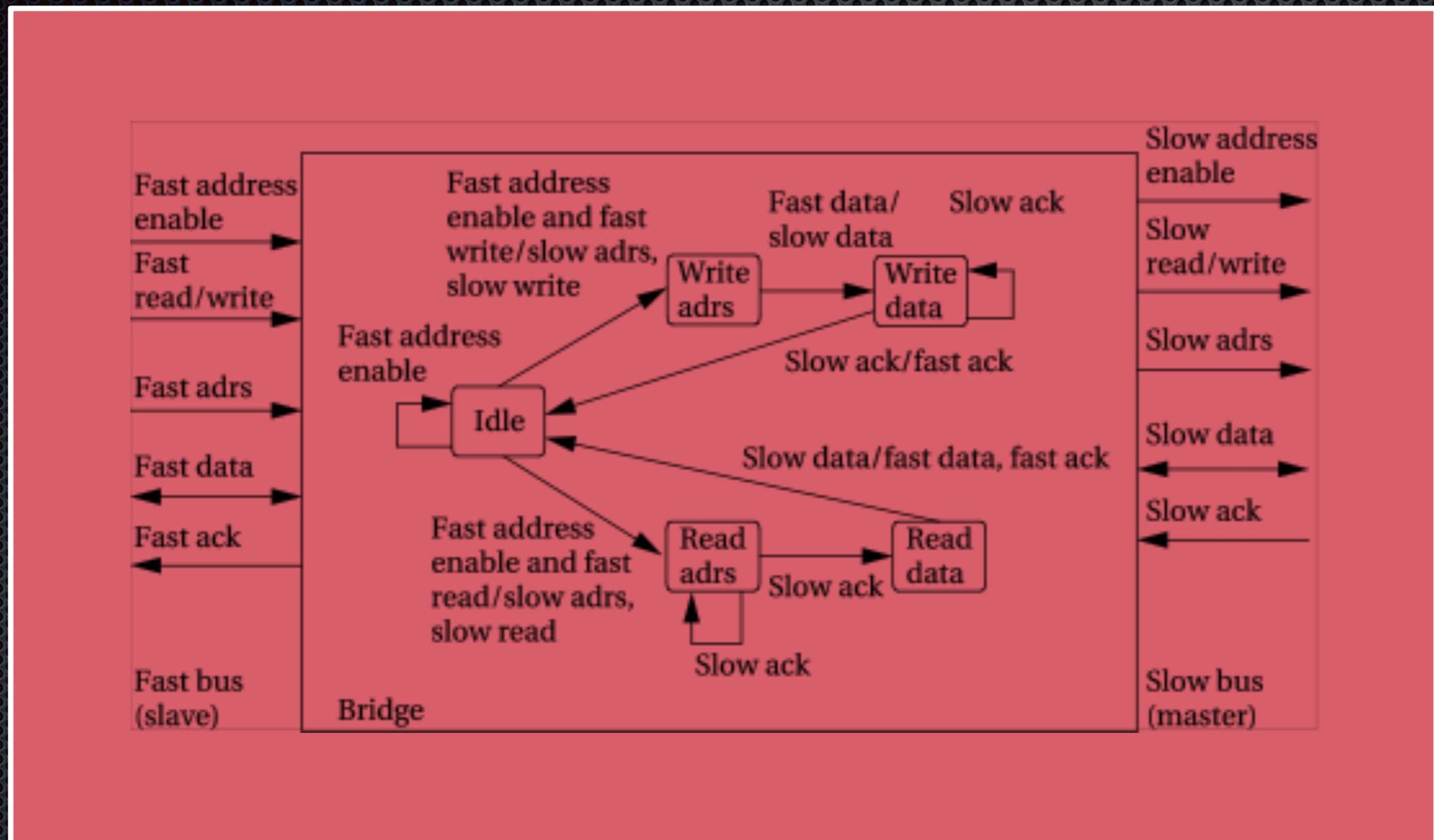


系统总线配置

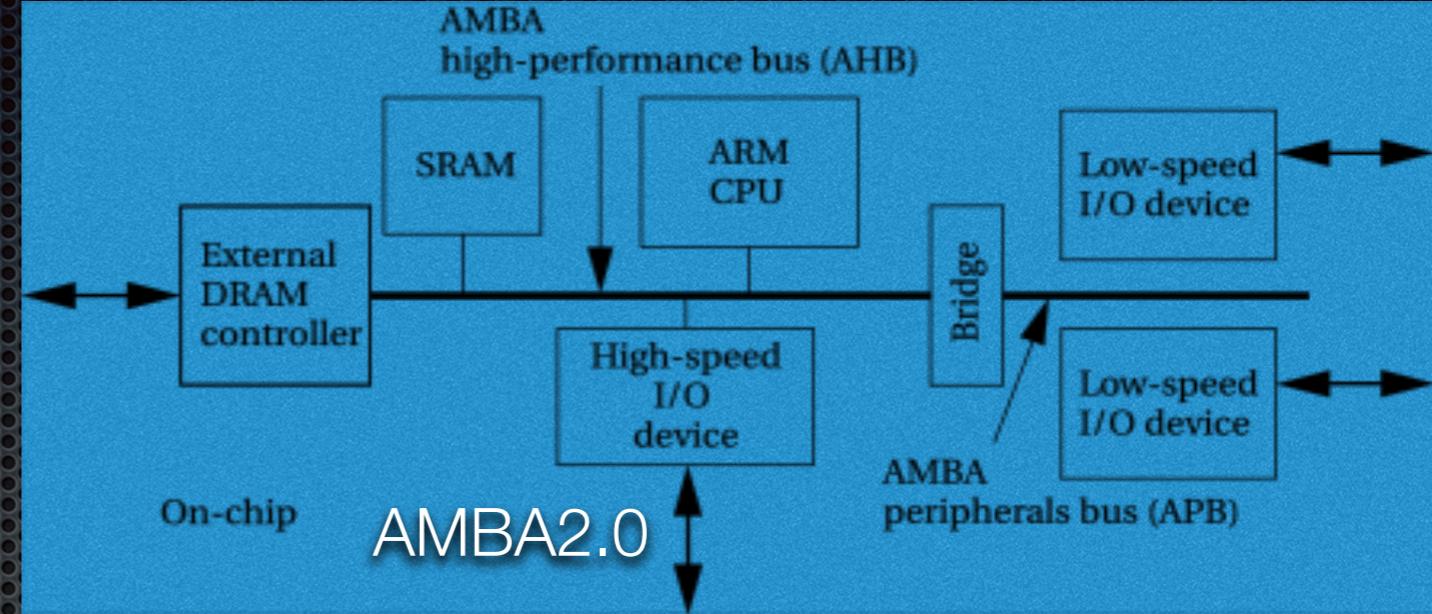
- 多总线允许并行:
 - 一个总线连接慢速设备
 - 另一个独立总线连接高速设备
- 桥连接两个总线



桥状态图



ARM AMBA总线



◆ 分类：

- ◆ AHB(Advanced High-performance Bus) 高级高性能总线
 - ◆ 主要是针对高效率、高频宽及快速系统模块所设计的总线，它可以连接如微处理器、芯片上或芯片外的内存模块和DMA等高效率模块
- ◆ APB (Advanced Peripheral Bus) 高级外围总线
 - ◆ 速度更慢，成本更低，主要用在低速且低功率的外围，可针对外围设备作功率消耗及复杂接口的最佳化，APB在AHB和低带宽的外围设备之间提供了通信的桥梁
- ◆ AXI （ Advanced eXtensible Interface ） 高级可拓展接口
 - ◆ AXI是在AMBA3.0的协议中增加的，可以用于ARM和FPGA的高速数据交互
 - ◆ 高速度、高带宽，管道化互联，单向通道，只需要首地址，读写并行，支持乱序，支持非对齐操作，有效支持初始延迟较高的外设，连线非常多。

比较

总线	AXI	AHB	APB
总线宽度	8, 16, 32, 64, 128, 256, 512, 1024	32, 64, 128, 256	8, 16, 32
地址宽度	32	32	32
通道特性	读写地址通道 读写数据通道均独立	读写地址通道共用读写 数据通道	读写地址通道共用读写数 据通道 不支持读写并行操作
体系结构	多主/从设备 仲裁机制	多主/从设备 仲裁机制	单主设备（桥）/多从设备 无仲裁
数据协议	支持流水线 支持突发传输 支持乱序访问 字节/半字/字 大小端对齐 非对齐操作	支持流水线 支持突发传输 支持乱序访问 字节/半字/字 大小端对齐 不支持非对齐操作	一次读/写传输占两个时钟 周期 不支持突发传输
传输方式	支持读写并行操作	不支持读写并行操作	不支持读写并行操作
时序	同步	同步	同步
互联	多路	多路	无定义

常用总线

UART

CAN

USB

SPI

I²C

认知度高
经济有效
简单

安全
快速

快速
即插即用硬件
简单
低成本

快速
广发接受
低成本
大型系列

简单
认知度高
广泛接受
即插即用
大型系列
经济有效

功能有限
点对点

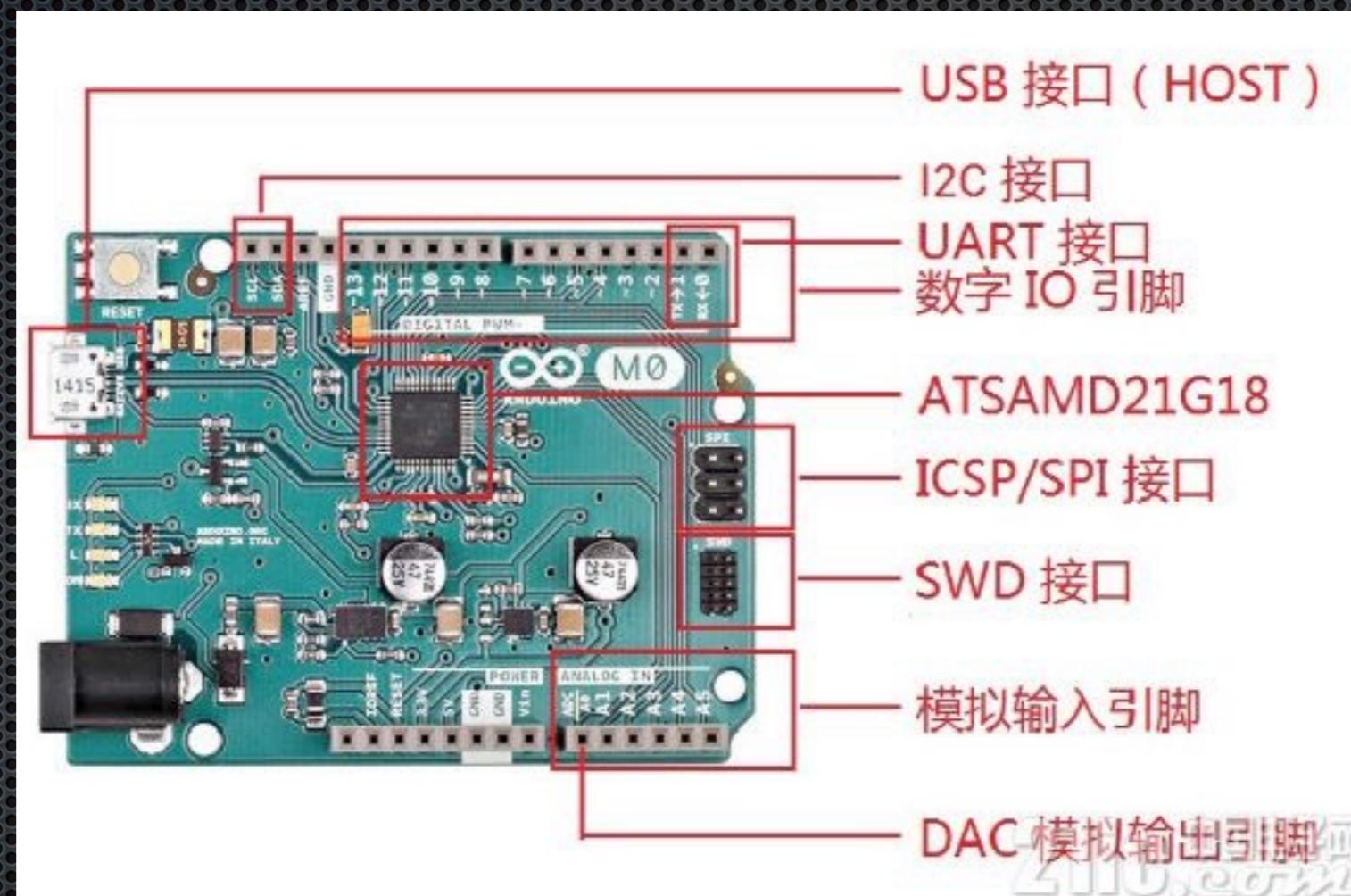
复杂
源自汽车
固件昂贵

需要强大主机
无即插即用软
件，需指定驱动

无即插即用硬件
没有固定标准

有限速率

Arduino Uno R3



Raspberry Pi 3 GPIO Header

<i>Pin#</i>	<i>NAME</i>		<i>NAME</i>	<i>Pin#</i>
01	3.3v DC Power		DC Power 5v	02
03	GPIO02 (SDA1 , I ² C)		DC Power 5v	04
05	GPIO03 (SCL1 , I ² C)		Ground	06
07	GPIO04 (GPIO_GCLK)		(TXD0) GPIO14	08
09	Ground		(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)		(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)		Ground	14
15	GPIO22 (GPIO_GEN3)		(GPIO_GEN4) GPIO23	16
17	3.3v DC Power		(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)		Ground	20
21	GPIO09 (SPI_MISO)		(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)		(SPI_CE0_N) GPIO08	24
25	Ground		(SPI_CE1_N) GPIO07	26
27	ID_SD (I ² C ID EEPROM)		(I ² C ID EEPROM) ID_SC	28
29	GPIO05		Ground	30
31	GPIO06		GPIO12	32
33	GPIO13		Ground	34
35	GPIO19		GPIO16	36
37	GPIO26		GPIO20	38
39	Ground		GPIO21	