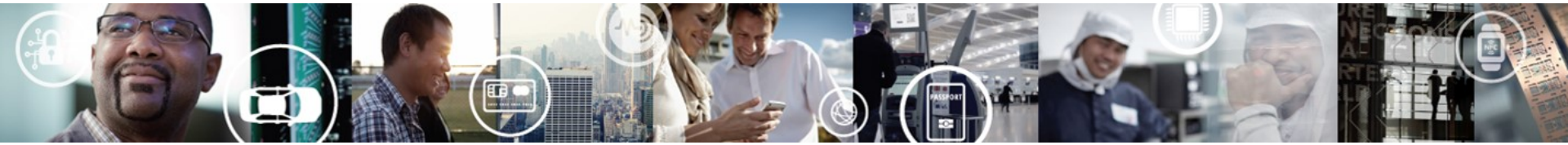
LPC82X 培训资料

中断的概念与使用

MAY, 2016



SECURE CONNECTIONS
FOR A SMARTER WORLD

内容

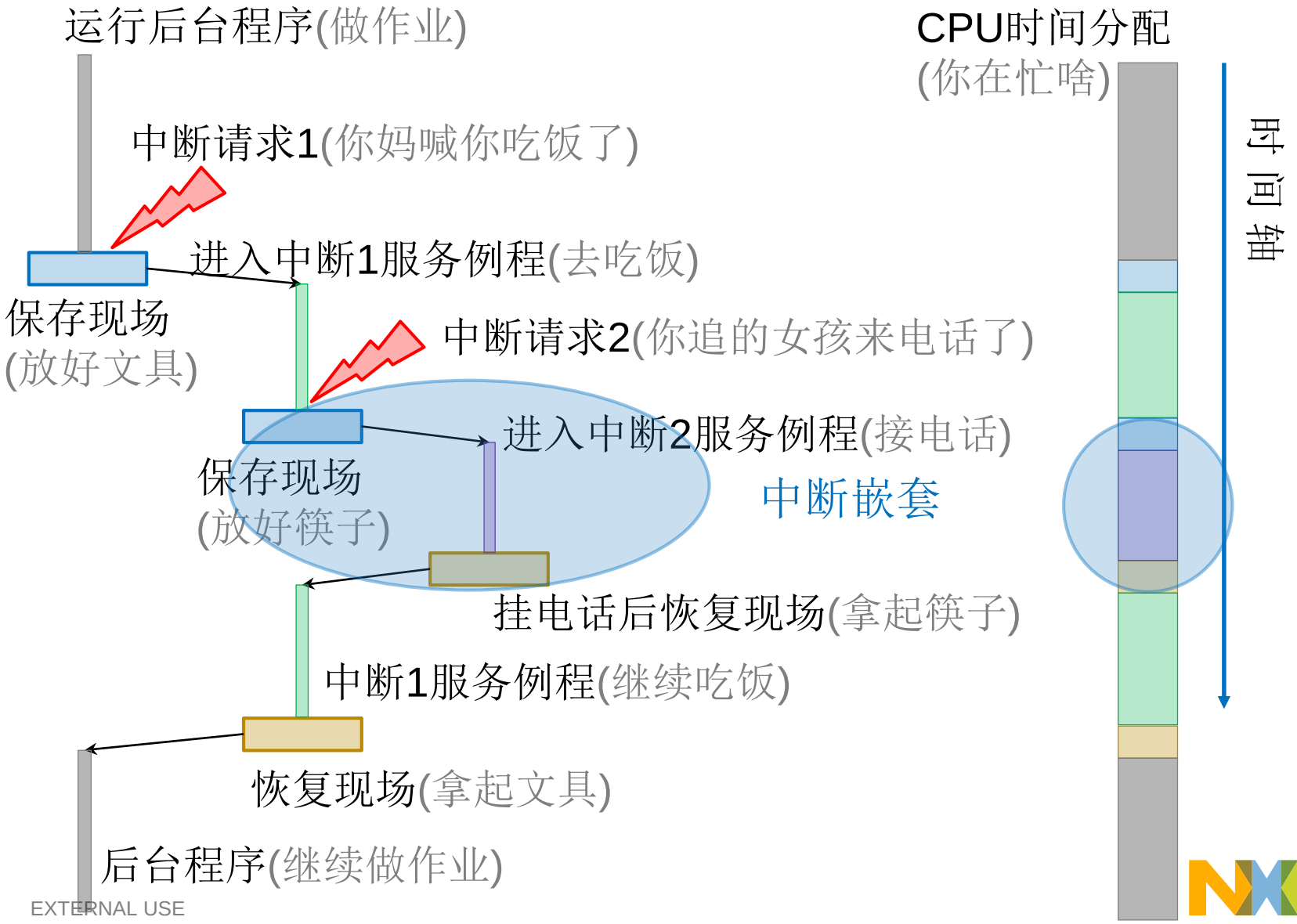
- 中断概述
- LPC82x 的中断系统概述

中断概述

中断概述

- “中断”就是当前的任务被更紧要的事件打断。这些事件如不及时处理可能导致系统故障：
  - 例如，UART收到数据后不及时取走导致数据丢失
  - 再来一个更严重的：电源故障中断不及时处理导致系统意外关机
  - 话说回来，并不是所有中断都必须实时响应。例如，UART发送完毕后会发中断通知CPU，但是晚些处理也只是导致吞吐能力降低。
- 中断也是一种CPU时间复用模型
  - CPU暂停当前正执行的程序（后台）并保存反映当前CPU处理状态的一些寄存器——“中断现场”(Context)，转而处理待响应的事件（前台），处理完成后恢复“现场”并继续原先执行的程序。
  - 因为CPU很快，所以看起来像是并行地处理了后台与前台任务
- 中断的概念源于生活，低于生活（见下页图示）：
  - 放学回家做作业->吃晚饭->接电话->继续吃晚饭->吃完晚饭继续做作业
  - 其中，“接电话”就是所谓的嵌套中断

中断响应全过程模式图



为什么需要中断

- 对突发情况紧急处理 (电话是你正在追的女孩打来的)
- 因为CPU相对人来说还是非常快的，实现宏观上的并行多任务。
  - (你在晚上做了作业，吃了饭，还接了电话。因为你的一天只是天上1秒，于是玉皇大帝很高兴他一眨眼的工夫你已经”并行”地做好了三件事)
  - 例如，一边计算一边采集一边执行
- 解决CPU与慢速外设和偶发事件速度不一致的矛盾(你不用流着口水在厨房等饭熟了)
  - CPU平时可处理其它工作
  - 仅当事件发生时再处理
  - 如果没有中断系统，CPU需要不停循环查询标志位/引脚电平，浪费大量处理资源

常见的使用中断的场合

- 引脚状态变化产生中断，也是常见的“外部中断”
  - 一般是反映外部产生的事件，例如按钮按下、片外外设通知事件等
- 通信接口的收到数据与发送完毕中断
  - 一个数据单位发送完毕时，产生中断，通知应用逻辑继续提供数据
    - 进一步地，发送器闲置中断可使应用逻辑更早地注入新数据
  - 接收器中有数据时，产生中断，通知应用逻辑取走数据
- 定时器溢出/倒数至0时，产生中断，通知时间到
- 模拟比较器结果翻转时产生中断
- 模数转换器采样完毕时产生中断
- 其它一些紧急的偶发事件：
  - 没有及时喂狗时的看门狗中断
  - 供电电压不足产生中断

中断服务例程 (ISR, Interrupt Service Routine)

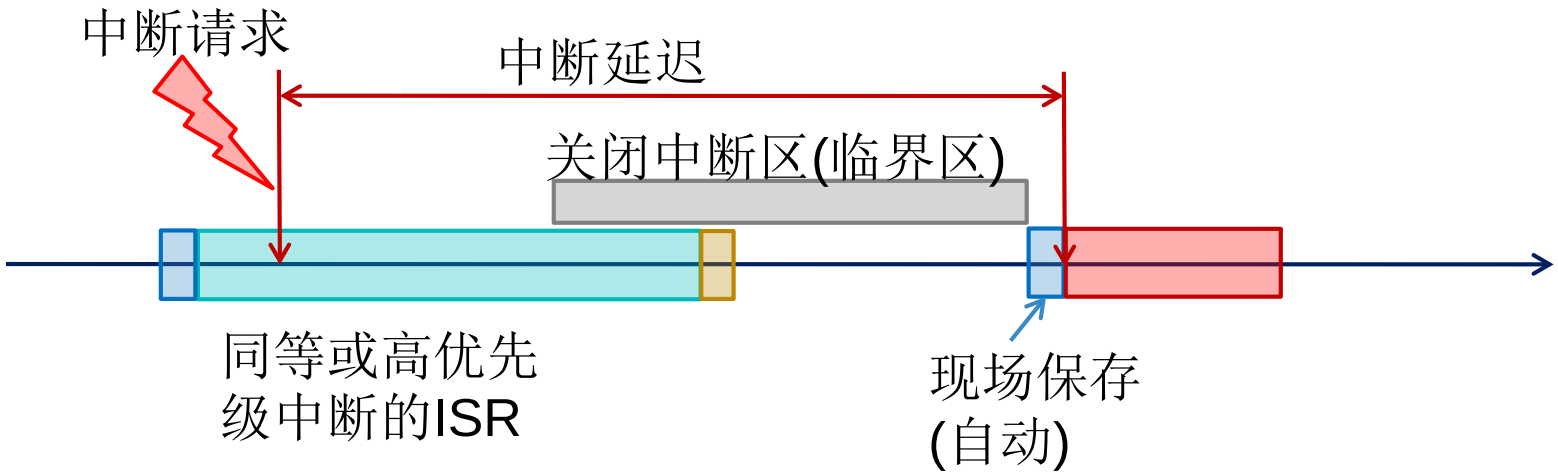
- 响应中断时，CPU将会例行地调用一段程序，用以执行中断发生后需要立即处理的事。这段程序即为“中断服务例程”。ISR常见的例行工作包括：
  - 清除中断事件标志
  - 如果与数据有关，处理数据
  - 作初步的软件层面上的处理
  - 按需设置软件层面上的标志位，供后台程序进一步处理
  - 按需访问硬件寄存器使硬件就绪后续工作
- ISR因为打断了当前的工作，所以要尽量短小精焊，只做必须马上处理的事
  - 例外：有些超低功耗的简单的系统平时一直待机，仅在中断产生时唤醒，并且在ISR中处理全部工作，中断返回就回到待机状态。例如，电池供电的烟雾报警器，气温采集器。

举例：I2C从模式中断

```
void I2C0_IRQHandler(void)
{
    清除I2C中断标志
    if (接收中断) {
        读取数据寄存器;
        写入缓冲队列;
        缓冲区快满时设置标志变量;
    }else{
        读取缓冲队列;
        写入数据寄存器;
        缓冲区快空时设置标志变量;
    }
    操作I2C外设释放SCL线;
}
```

中断延迟

- 从中断请求发起到ISR得到执行所花费的时间，称为中断延迟
  - 现场保存(最少24周期)
  - 如果此时中断被关闭(临界区)，需要待中断重新打开后才能响应
  - 如果此时正在服务同等或更高优先级的中断，需待其ISR退出



中断的代价

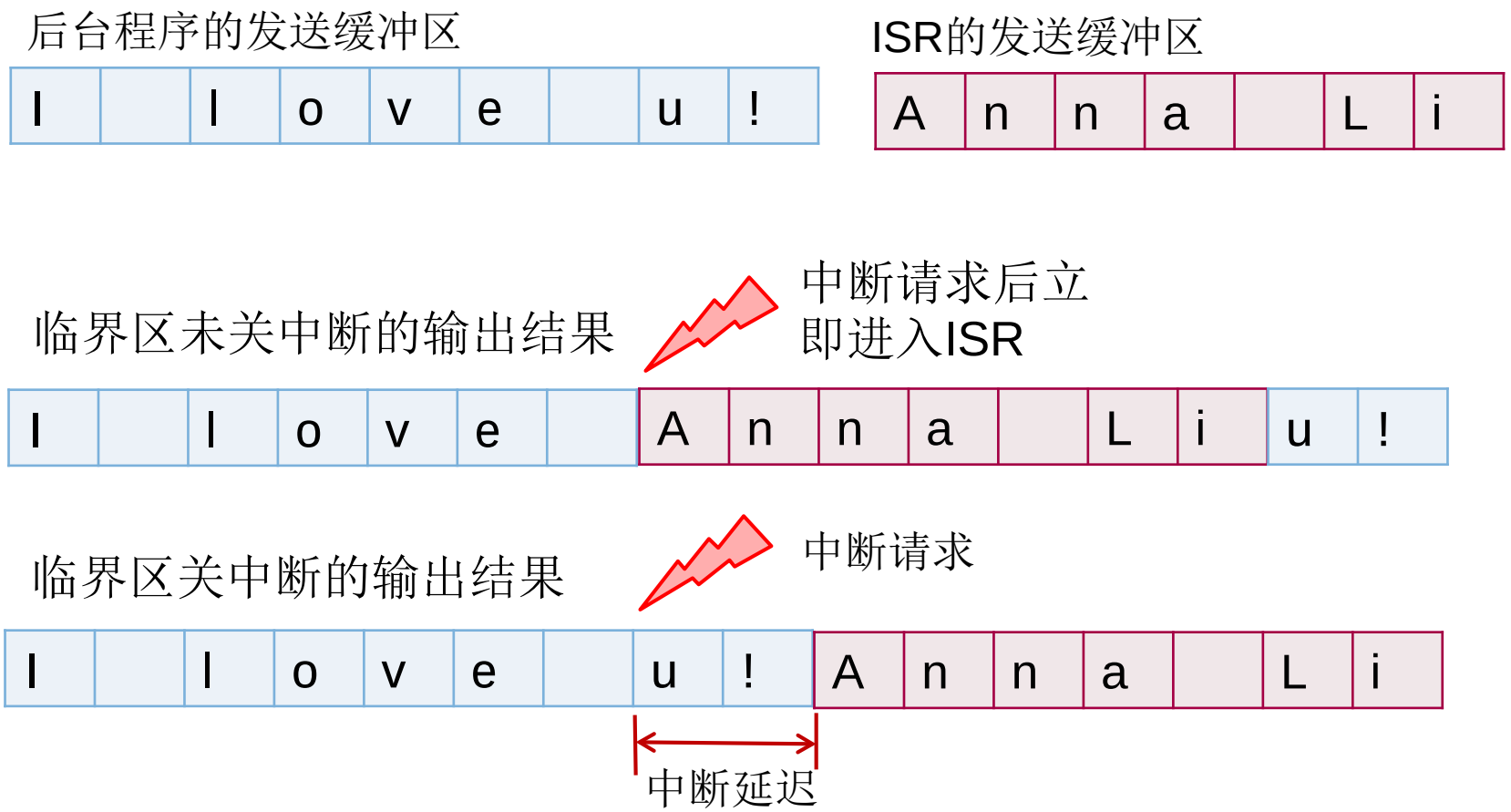
- 现场需要额外的栈空间来保存：LPC82x需要32字节保存现场
- 现场保存和恢复需要额外的时间:LPC82x需要约24周期
- 打乱了程序执行顺序，可能导致错综复杂的潜在问题。
  - 执行流程变得不可预测，变数增加，时序难保证，故障难复现。
  - 需要保证ISR与其它程序段互斥访问全局变量，尤其是复合型全局变量(struct)。如果处理不当会导致难以调试的竞争条件，可能导致数据紊乱的问题
    - 竞争条件举例：假如后台程序和ISR都要执行”a++;”语句，a初始值为0
      - 后台程序读取a到寄存器，得到0
      - 后台程序在寄存器中把0加到1，然后，早不来晚不来，中断偏偏这个时候来了：
      - ISR读取a到寄存器，也得到0
      - ISR在寄存器中把0加到1
      - ISR把寄存器中的1写回a，a现在等于1
      - 后台程序也把寄存器中的1写回a，a现在还等于1
    - 看到没，执行了两次a++后，a实际上只被++了一次！

如何解决这个问题？
答案在下一页

临界区与需要关闭中断的场合

- 有些场合必须关中断，必须在关中断时执行的代码又称为“临界区”(critical section), 有以下典型情况：
  - 1. 当后台程序与ISR均要更改同一位置的数据时，要先关中断，待改完后再开中断。否则会导致竞争冒险的情况，可能使数据紊乱
  - 2. 当后台程序与ISR均要操控同一外设时，要先关中断，待访问完毕后再开中断，否则轻则使外设输出混叠，重则使外设功能混乱。
    - 例如，后台程序和ISR都要使用同一个UART，后台程序要输出”I love u!”, ISR要输出”Anna Li”, 若后台程序在输出期间未关中断，可能输出“I love Anna Liu!”
    - 又如，在调用IAP函数擦写Flash时，如果未关中断，可能导致擦写期间又从Flash取指令(ISR的指令)的情况，导致死机
  - 3. 当控制对时序有严格要求的器件时，需要关中断以保证确定性
  - 4. 以上注意事项也适用于低优先级ISR与高优先级ISR之间。
  - 5. 使用RTOS时，多任务的调度修改了中断的返回地址，使得以上注意事项扩展到RTOS管理的任务间。

临界区未关中断的示意效果演示



Cortex-M0+上中断系统的增强功能

- 自动保存和恢复共计32字节的中断现场 (8个CPU寄存器)
  - 当前RAM和Flash的内容则不属于“现场”
    - 事实上，它们不是CPU的组成部分
- 使用普通的函数调用方式返回，不需要专用的返回指令
- 自带中断控制器NVIC管理外设和外部中断。

- 因为以上的优点，中断处理无需使用汇编，也无需使用特殊的编译器扩展功能，写法和普通的程序相同
  - 这并不意味着ISR变成普通程序了，中断处理的注意事项仍需遵守。

LPC82X 的中断系统概述

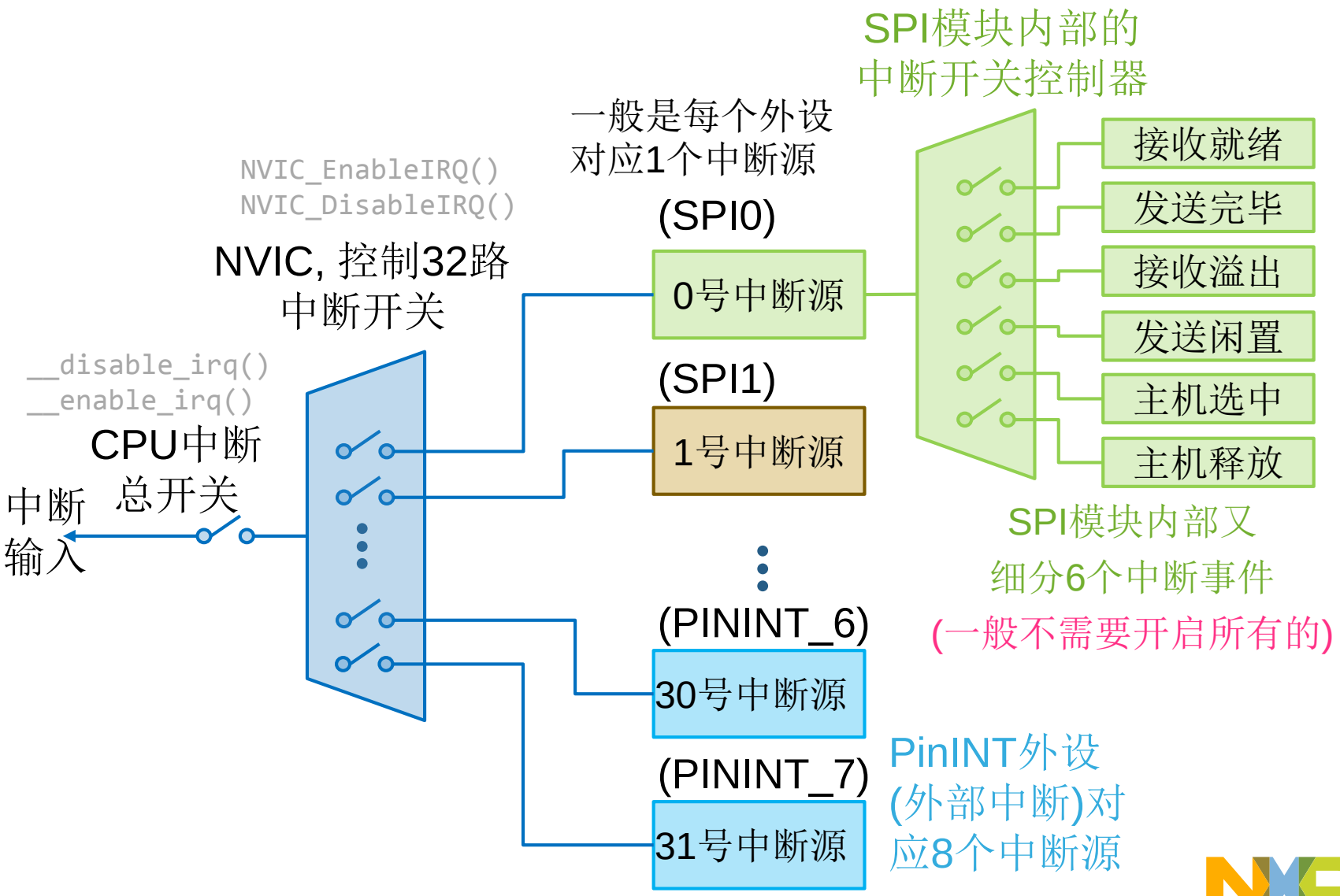
LPC82x (Cortex-M0+ CPU)的中断系统概述

- M0+内部集成了中断系统的核心功能：
  - 管理中断的进入、退出、嵌套，与优先级制度；自动保存与恢复现场
  - 这一切都是全自动的，软件看不见——也就是说不用写汇编代码！
- M0+自带了一个嵌套中断控制器，简称NVIC，支持32路中断输入
  - 提供开关控制
  - 反映和控制待决标志
  - 设置优先级
  - 反映ISR是否已进入
  - 如果发生嵌套，可以出现两个以上中断的ISR已进入的情况
- 除了来自NVIC的中断，M0+自身还定义了其它的特殊中断类型
  - 不可屏蔽中断(NMI), 硬故障(Hard_Fault), 系统服务(SVCall, PendSV), 系统节拍定时器(SysTick)

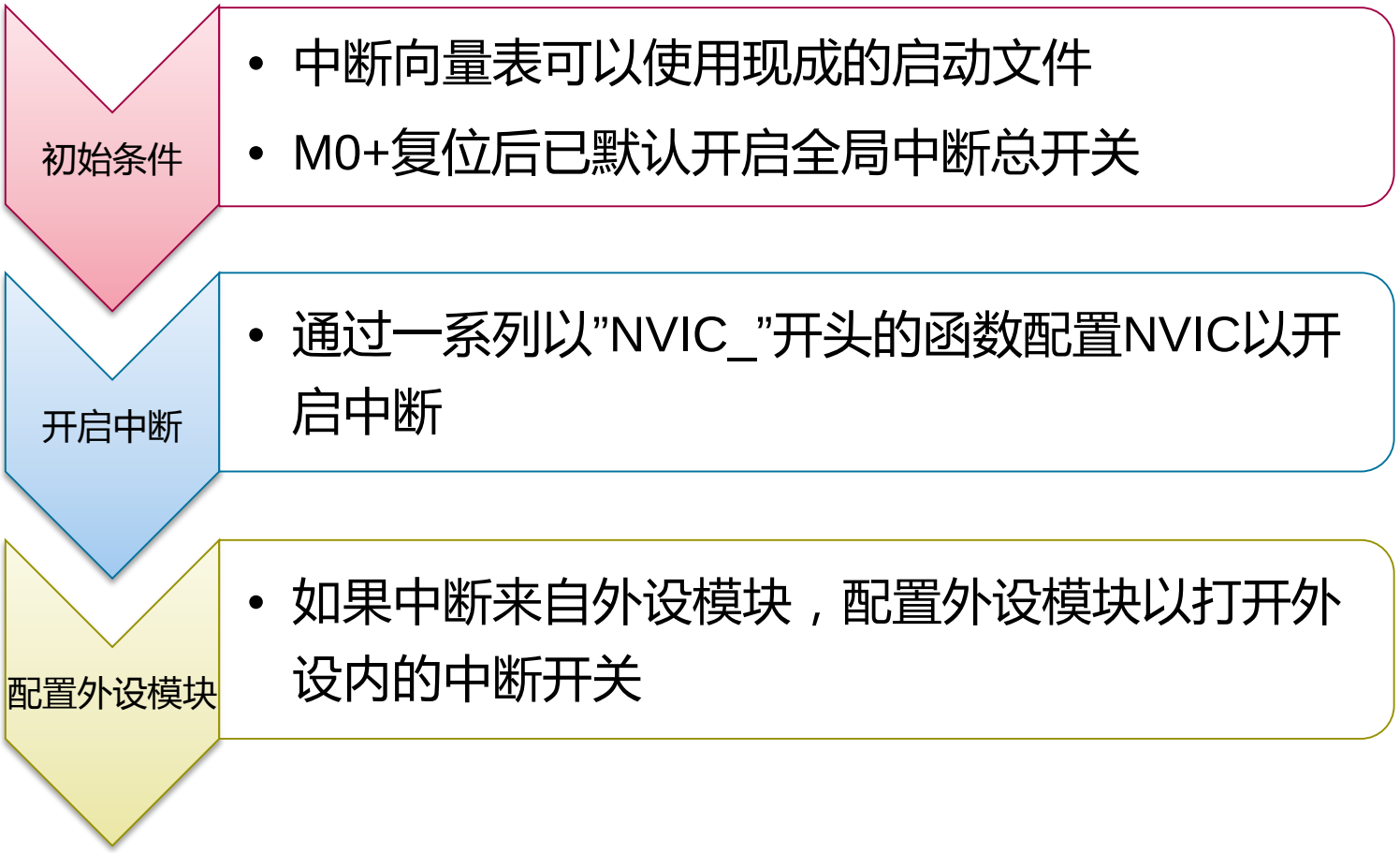
中断编号

- M0+ CPU为每个中断都分配了唯一识别号
  - 你可以把它当作是中断源的身份证号
- 中断号非常重要，它是识别和响应中断的凭据
  - 中断号用于寻找ISR的入口地址
  - M0+内核以变换后的中断号记录当前的中断
  - NVIC就是以中断号管理各路中断源的
- 在NVIC的API函数中也是以中断号作为参数
  - 这个和我们写程序时关系最密切了
  - 在cmsis.h中的”IRQn_Type”定义了LPC8xx的中断编号分配
    - NVIC管理的中断从0号开始
    - M0+直接管理的特殊中断，中断号为负

中断开关控制示意图


SPI模块内部的
中断开关控制器
一般是每个外设
对应1个中断源
NVIC_EnableIRQ()
NVIC_DisableIRQ()
NVIC, 控制32路
中断开关
(SPI0)
0号中断源
接收就绪
发送完毕
接收溢出
发送闲置
主机选中
主机释放
__disable_irq()
__enable_irq()
CPU中断
总开关
中断
输入
(SPI1)
1号中断源
SPI模块内部又
细分6个中断事件
(一般不需要开启所有的)
(PININT_6)
30号中断源
PinINT外设
(外部中断)对
应8个中断源
(PININT_7)
31号中断源

在LPC8xx上使用中断



LPC8xx的CPU内核(M0+)支持的其它中断

- 除了来自NVIC的中断，M0+还支持其它的特殊中断类型，用于任何以M0+为内核的MCU：
  - **NMI**: 不可屏蔽中断。LPC8xx可以任选一个中断源连接到NMI
    - 常用的是把看门狗中断或者欠压警告中断连接到NMI，它们的后果很严重。
  - **Hard_fault**: 软件或总线错误导致程序不能执行下去时，就会产生hard fault(硬故障)，并且走中断响应的流程进入Hard_Fault中断
    - 例如，软件存取了一个不存在的地址，或者地址没有对齐
    - 又如，CPU读取到的指令译码失败(如果M3的bin文件烧到了M0的器件上）
  - **SVCall和PendSV**: 给操作系统选用的系统服务中断（咱们一般不用）
    - 在严格划分特权级别以保护系统的RTOS上可能会用到SVCall
    - PendSV专用于上下文切换
  - **SysTick**: 这是M0+自带的一个简单的节拍定时器所对应的中断源。
    - 在一般情况下可以作为系统的时基，有超低功耗要求时不能用。

中断向量表

- M0+要求程序在CPU看到的0地址存储一张表，称为“中断向量表”
  - –每个表目存储一个ISR的入口地址（有些位置没有用到）
  - –每个中断源在表中有自己的位置
    - NVIC管理的中断，就是中断号+16
    - M0+自己管理的中断，另有编号机制
  - –0号表目有些特殊，它存储复位后的栈指针，会被装载到SP寄存器
  - –默认的中断向量表存储在启动代码中
- LPC8xx提供“0地址重映射”功能，把0地址开始的512字节范围映射到Flash, RAM, 或者Boot ROM
  - –用户程序执行时默认把Flash映射到0.
  - –用户程序可以今后再手工把向量表放到Flash或RAM
  - –映射成RAM后，Flash的前512字节内容无法被访问到！
- 启动后，M0+允许把向量表放到别处 (NVIC的VTOR寄存器)

LPC82x启动文件(keil_startup_lpc82x.s)中的中断向量表

```
                DCD     __initial_sp                ; Top of Stack
                DCD     Reset_Handler               ; Reset Handler
                DCD     NMI_Handler                 ; NMI Handler
                DCD     HardFault_Handler           ; Hard Fault Handler
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; checksum
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     SVC_Handler                 ; SVCall Handler
                DCD     0                           ; Reserved
                DCD     0                           ; Reserved
                DCD     PendSV_Handler              ; PendSV Handler
                DCD     SysTick_Handler             ; SysTick Handler
                ; External Interrupts
                DCD     SPI0_IRQHandler               ; SPI0 controller
                DCD     SPI1_IRQHandler               ; SPI1 controller
                DCD     0                             ; Reserved
                DCD     UART0_IRQHandler              ; UART0
                DCD     UART1_IRQHandler              ; UART1
                        . . .（此处略过）
                DCD     PIN_INT5_IRQHandler           ; PIO INT5
                DCD     PIN_INT6_IRQHandler           ; PIO INT6
                DCD     PIN_INT7_IRQHandler           ; PIO INT7
```

- 前16个是M0+内核自己管理的特殊向量，有两个比较特殊：
  - 0号：栈指针初值
  - 1号：复位后第1条指令的地址
  - 7号: 是前面7个向量之和的checksum补码由开发工具自动计算
- 后面32个是NVIC管理的最多32路中断未用到的填0

SECURE CONNECTIONS
FOR A SMARTER WORLD