

[toc]

嵌入式

概述

问答

什么是嵌入式系统

嵌入到应用对象中的专用的计算机系统

以应用为中心，以计算机技术为基础、软硬件可裁剪、功能、可靠性、成本、体积、功耗严格要求的专用计算机系统

为什么要提出嵌入式

嵌入式系统的特点

- 可以裁剪
- 小而实时
- 专用性 -- 为特定应用定制的计算机系统
- 裁剪性 -- 软、硬件小而精，够用即可
- 实用性 -- 程序和数据都在存储器中，既满足逻辑正确性，也要满足时间正确性
- 可靠性 -- 无人值守。自动化
- 低功耗 -- 便携式应用的要求
- 高性价比 -- 家用的应用要求

对于功能、可靠性、成本、体积、功耗严格要求的专用计算机系统

结构

一般由嵌入式微处理器、外围硬件设备、嵌入式操作系统以及用户应用程序等四个部分组成

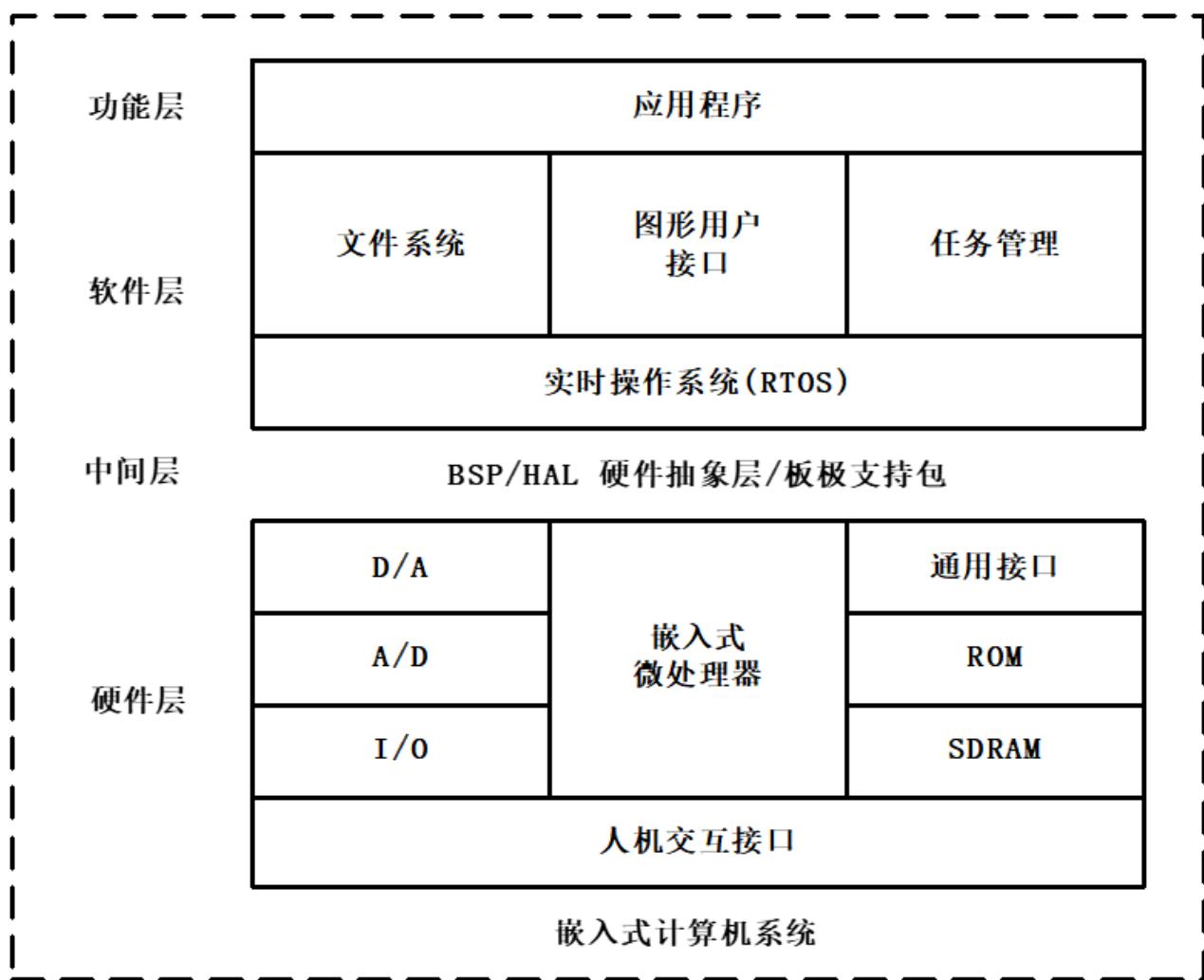
从系统的角度定义：

嵌入式系统是设计完成复杂功能的硬件和软件，并使其紧密耦合在一起的计算机系统 嵌入式反映了：这些系统通常是更大系统中的一个完整的部分，称为嵌入的系统，嵌入的系统重可以共存多个嵌入式系统

系统架构

- 硬件
- 驱动
- 程序
- 操作系统
- 中间件 -- 第三方设计的函数包，库，层次结构

- 应用程序



嵌入式系统编程技巧

1. 时间和空间的效率: 更快更省更好
 1. 硬件的致辞: 特殊的指令集 thumb-空间效率
 2. 软件的技巧: 查表 指针 初始化的提前定义
2. 省电
 1. 时钟的处理: 时钟的变频, 功能模块的时钟可选择关闭
 2. 操作系统的切换: 一个任务等待时另一个任务可以运行, 减少无意义的NOP指令的功率耗费
 3. idle任务可以进入CPU的低功耗模式
 4. 事件机制: 减少原轮询机制的运行功耗, 大部分时间可处于低功耗模式

从代码到可执行文件

以ARM工具链为例子(keil使用的工具链) 工具链有:

- armcc C/C++编译器
- armasm 汇编编译器
- armlink 链接器

1. 预处理

- 输入.c/h文件, 输出.i文件

- 工具：armcc

2. 编译

- 输入.i文件，输出 **.s文件**（汇编文件）

3. 汇编

- 输入.s文件，输出 **.o文件**（目标文件）

3. 链接

- 输入.o文件，输出 **.axf文件**（可执行文件）
- axf文件包含了程序代码，数据，符号表，**调试信息**，导入导出表，内存映射表，程序入口地址等信息，用于调试，可以直接烧录到嵌入式设备
- 将多个目标文件生成一个可执行文件

4. 生成烧录文件（有bin和hex两种固件烧录格式）

- 生成.bin文件
- 可选生成hex文件，hex文件带有地址信息
- bin即纯二进制文件，不带有地址信息

知识点

嵌入式微处理器

32位以上的处理器，在实际嵌入式应用中，只保留与嵌入式应用紧密相关的功能硬件，去除其他的冗余功能部分，这样就以最低的功耗和资源实现嵌入式应用的特殊要求。和工业控制计算机项相比，其体积小，重量轻，成本低，可靠性高的优点

实时系统概念

双核 实时操作系统：能够在指定或者确定的时间内完成系统功能以及对外部或内部事件在同步或异步时间内做出响应的系统

嵌入式系统发展与应用

发展趋势：

1. 嵌入式硬件集成化
2. 嵌入式环境网络化
3. 嵌入式软件构件化
4. 应用模式普适化
5. 行业应用标准化

嵌入式系统软硬件IP核

知识产权（IP） 电路或核是一种预先设计好的，甚至已经验证过的具有某种确定功能的集成电路、器件或部件。它有行为、结构、物理3种设计，对应的种类是软核、固核和硬核

操作系统分类

- 顺序执行系统 如DOS
- 分时操作系统 如UNIX
- 实时操作系统
 - 强实时系统，其系统响应时间在毫秒或微妙级
 - 一般实时系统，其系统响应时间在毫秒-几秒的数量级
 - 弱实时系统，其系统响应时间约为数十秒

软件结构

- 循环轮询系统
- 事件驱动系统

前后台系统

是中断驱动系统的一种

- 后台是一个轮询系统一直在运行
- 前台是由一些中断处理过程组成的
- 当有一前台事件（外部事件）发生时，引起中断，进行前台处理，处理完成后又回到后台（通常称为主程序）

需要考虑的是中断的现场保护和恢复，中断嵌套，中断处理过程与主程序的协调（共享资源）问题

后台循环，前台中断 系统的性能主要由中断延迟时间，响应时间和恢复时间来刻画

多任务

多任务运行的实现实际上是靠CPU(中央处理单元)在许多任务之间转换、调度 CPU只有一个，轮番服务于一系列任务中的某一个，多任务运行使CPU的利用率得到最大的发挥，并使应用程序模块化

任务优先级

每个任务都有优先级

- 静态优先级 应用程序执行过程中，任务的优先级是不变的，在编译时是已知的
- 动态优先级 任务的优先级是可变的，实时内核应当避免出现优先级反转问题

调度

调度是内核的主要职责之一，调度就是决定该轮到哪个人物运行了，多数实时内核是基于优先级调度法的，分为非占先式和占先式两种

非占先式/占先式

- 非占先式：也称为合作型多任务，中断服务可以使一个高优先级的任务由挂起状态变为就绪态，但中断服务以后控制权还是回到原来被中断了的那个任务，直到改任务主动放弃CPU的使用权，那个高优先级的任务才能获得CPU的使用权
 - 一个特点是不需要信号量保护共享数据，运行着的任务占有CPU

- **最大缺陷**是响应高优先级的任务慢，内核的响应时间也是不确定的，因为需要等任务执行完
- 占先式：最高优先级的任务一旦就绪，总能得到CPU的控制权
 - 当一个运行着的任务使一个比它优先级高的任务进入了就绪态，当前任务的CPU使用权就被剥夺了，或者挂起，高优先级的任务立马得到了CPU的控制权
 - 使用占先式内核时，应用程序应直接使用可重入型函数

ucos是可抢占的（占先式内核）

可重入函数

可以被一个以上的任务调用，而不必担心数据的破坏 可重入型函数或者只使用局部变量，即局部变量保存在CPU寄存器中或堆栈中

一个不可重入型函数的例子 一个可重入型函数的例子

```
int Temp;
Void swap (int *x,int*y)
Void swap (int *x,int*y)    {
{
Temp=*x;
*x=*Y;
*y=Temp;
}
}
int Temp;
Temp=*x;
*x=*Y;
*y=Temp;
```

课堂补充

嵌入式分层

APP	应用层
middle ware	中间件
OS	RTOS
BSP/HAL	驱动层/硬件抽象层
HW	硬件

内联函数和带参宏

带参宏：

没有现场切换，但是会代码膨胀。 **内联函数：**

ARM

指令

- 存储器访问指令
- ARM数据处理类指令
- 程序状态寄存器访问指令
- ARM分支转移类指令
- ARM协处理器类指令
- 软件中断和断点指令

ARM体系架构及指令集

指令集

CISC指令集 复杂指令集，指令条数多，是不等长的指令集，复杂功能有硬件实现，效率高，但是功耗巨大

- 具有大量的指令和寻址方式
- 8/2原则
- 大多数程序只使用少量的指令就能够运行

编码长度可变 1-15 字节 寻址方式多样

可以对存储器和寄存器进行算数和逻辑操作

RISC指令集 精简指令集，指令条数少，简单，是等长指令集，浮渣功能用多条指令组合完成（软件），再使用流水线降低指令的执行周期数

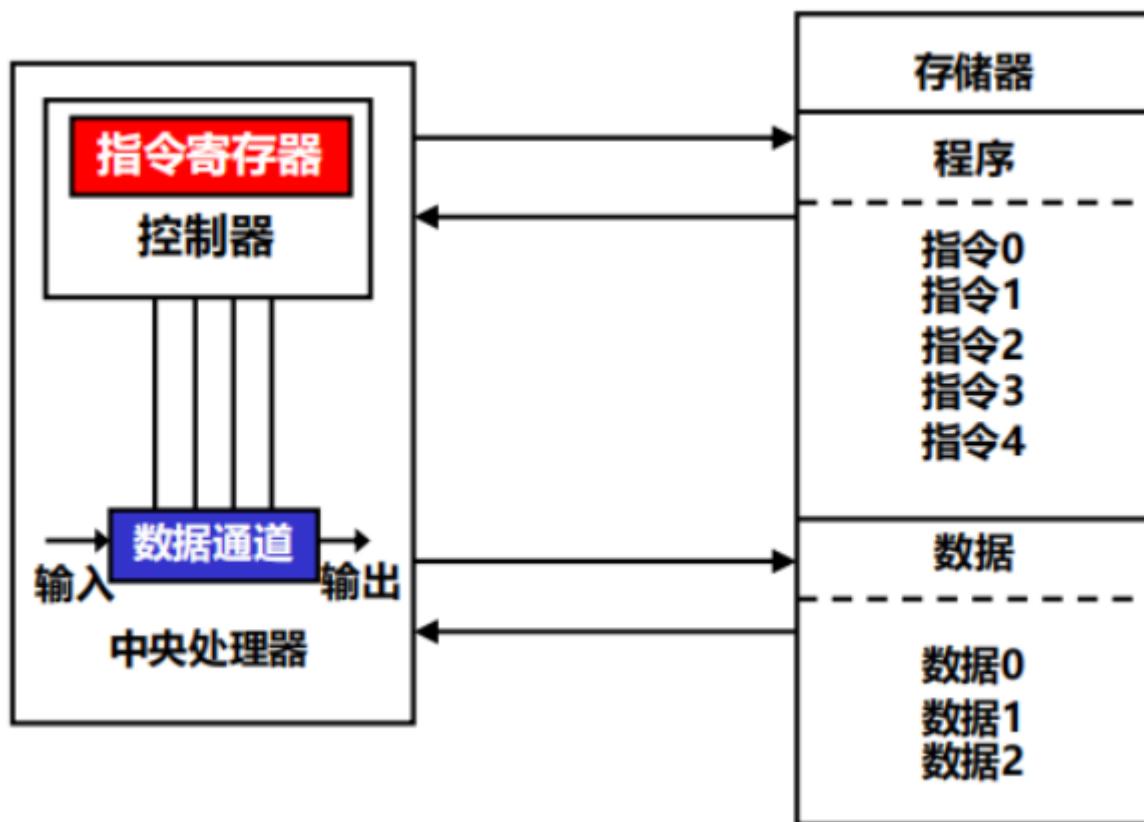
- 在通道中只包含最有用的指令
- 确保数据通道快速执行每一条指令
- 使CPU硬件结构设计变得更为简单

只能对寄存器进行算术和逻辑操作，Load/Store体系结构

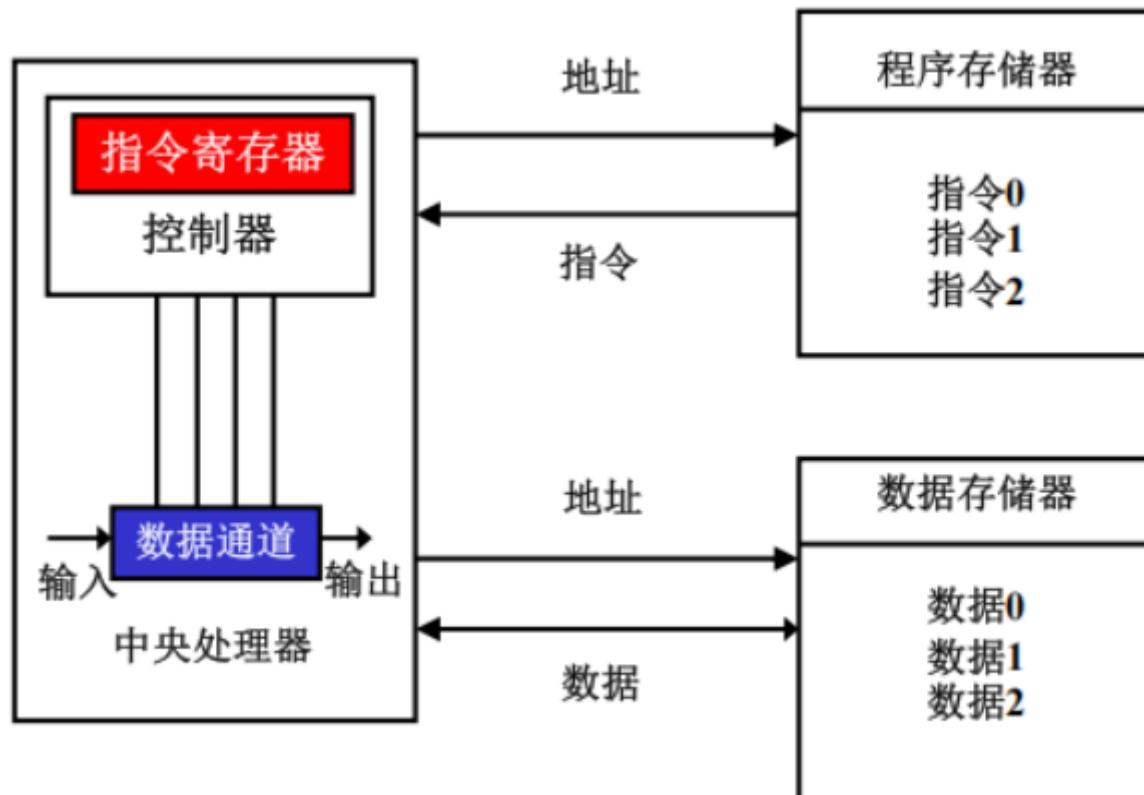
	CISC	RISC
价格	由硬件完成部分软件功能， 硬件复杂性增加 ，芯片成本高	由软件完成部分硬件功能， 软件复杂性增加 ，芯片成本低
性能	减少代码尺寸， 增加指令的执行周期数	使用流水线降低指令的执行周期数， 增加代码尺寸
指令集	大量的混杂型指令集，有简单快速的指令，也有复杂的多周期指令，符合HLL (high level language)	简单的单周期指令，在汇编指令方面有相应的CISC微代码指令
高级语言支持	硬件完成	软件完成
寻址模式	复杂的寻址模式，支持内存到内存寻址	简单的寻址模式，仅允许LOAD和STORE指令存取内存，其它所有的操作都基于寄存器到寄存器
控制单元	微码	直接执行
寄存器数目	寄存器较少	寄存器较多

架构

冯洛伊曼结构 指令和数据混合存储



哈佛结构 指令运算处理上和数据存储上采用并行结构



ARM指令集数据RISC指令，具有RISC指令的特点：指令少，且等长，标语充分利用流水线技术，使用多寄存器，且多为简单的Load/Store指令（内存取值，执行完再放回内存）

ARM和Thumb指令集

ARM7TDMI处理器中有两种工作状态：

- ARM - 32-bit 按字排列的ARM指令集
- Thumb - 16-bit 按半字排列的Thumb指令集

切换 使用BX指令（分支和交换指令）切换指令集状态。从ARM状态切换到Thumb状态

```
LDR R0, = Label +1  
BX R0
```

从Thumb状态切换到ARM状态

```
LDR R0, = Label  
BX R0
```

Thumb指令集可以看成是ARM指令压缩形式的子集，它是为减小代码量而提出的，提高代码密度，具有16位的代码密度

Thumb指令体系不完整，只支持通用功能，必要时仍需要ARM指令，如进入异常时。ARM指令为32位，Thumb指令为16位，thumb指令集与等价的ARM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

流水线 流水线技术：几个指令可以并行执行

ARM7系列使用3级流水线，允许多个操作同时处理，比逐条指令执行要快

PC指向正被取指的指令，而非正在执行的指令

大小端模式

- 大端模式
 - 字数据的高位字节存储在低地址中
 - 字数据的低位字节存储在高地址中
- 小端模式
 - 字数据的高位字节存储在高地址中
 - 字数据的低位字节存储在低地址中

模式

处理器模式

ARM体系架构支持7种处理器模式，分别为

- 用户模式
- 快中断模式
- 中断模式
- 管理模式
- 中止模式
- 未定义模式
- 系统模式 上电时候的模式为：管理模式 (SVC)

异常

异常 内部或外部中断源产生并引起处理器处理一个事件异常类型：

- **FIQ**快速中断请求， CPSR:I = 1
- **IRQ**外部中断请求， CPSR:F = 0 ,系统外设
- 未定义指令：ARM处理器或协处理器遇到不能处理的指令。
- 预取中止
- 数据中止
- 复位
- 软件中断：执行SWI指令产生，可用于用户模式下的程序调用特权操作指令，
 - 通过软件中断产生
 - 进行管理模式中获得
 - 通常要求特殊的管理功能，如操作系统支持

异常	发生条件	进入时的模式	返回指令
复位	复位信号有效时	管理	无
未定义指令	当遇到 ARM 处理器和协处理器都不能识别的指令时	未定义	MOVS PC, R14_und
软件中断异常	用户定义中断指令，用于用户模式下调用特权操作	管理	MOVS PC, R14_svc
中止（预取）	当 CPU 执行一条来自当前模式无法访问的地址的指令	中止	SUBS PC, R14_abt,#4
中止（数据）	当 CPU 对当前模式无法访问的地址进行数据传输时	中止	SUBS PC, R14_abt,#8
IRQ	外部中断请求信号有效且外部中断允许	中断	SUBS PC, R14_irq,#4
FIQ	快速中断请求信号有效且快速中断允许	快速中断	SUBS PC, R14_fiq,#4

在处理异常的时候，当前的处理器状态必须被保存，以便处理程序完成后，最后的程序可以被恢复。

说明

1. ARM文档中不严格区分中断和异常，都使用术语exception异常从CPU的角度描述，中断从外部事件的角度描述

2. 异常处理程序的重要性 设备驱动/中断处理 操作系统进程调度和切换

Load-Store指令

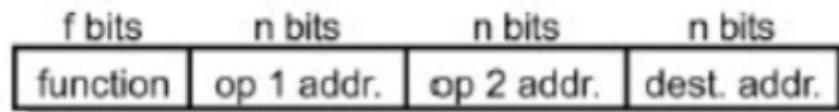
解释：从存储器中读某个值，操作完后再将其放回存储器中

指令分类

- 数据处理指令 - 使用和改变寄存器的值
 - 算术操作
 - 按位逻辑操作
 - 寄存器移位操作
 - 比较操作
- 数据传送指令 - 把存储器的值拷贝到寄存器中 **load** or 把寄存器中的值拷贝到存储器中 **store**
- 控制流指令
 - 分支
 - 分支和链接
 - 陷入系统代码

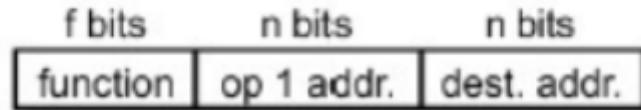
指令格式

➤ 在ARM状态中使用



- 3地址指令格式

➤ 在 ARM 和 THUMB 状态下使用



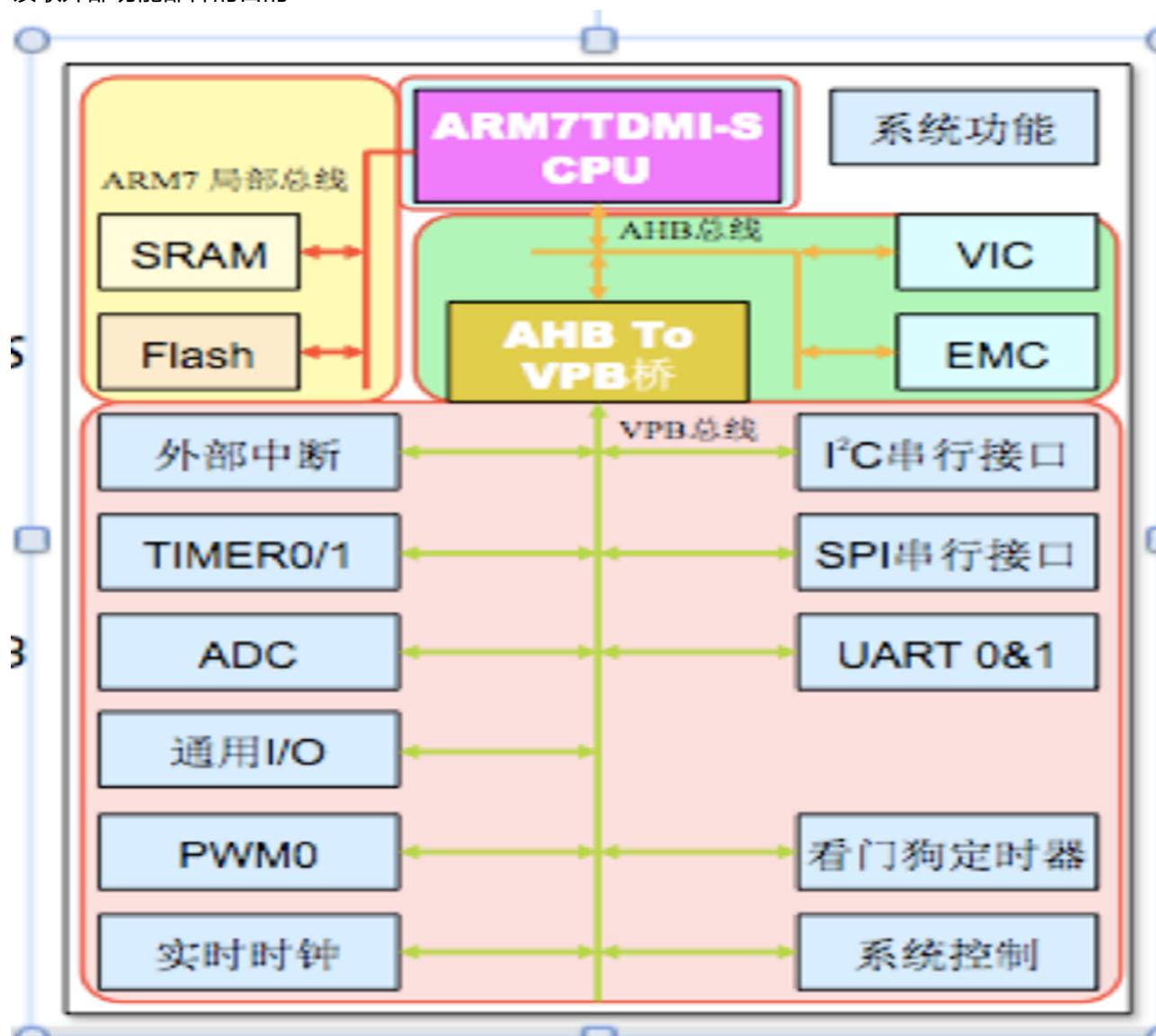
- 2地址指令格式

CPU如何扩展

数据交换（读/写）采用Load-store指令 由arm来扩展CPU的基本方法是把所有功能部件全部映射成内存，然后每个功能部件的寄存器映射成内存的地址（或一个变量）ARM最大只能对本地的寄存器进行操作，但是它可以通过Load-store指令让它自己的寄存器更功能部件的寄存器进行数据交换，交换完数据进行操作，操作完再交换回去，这样就可以对其他功能部件的数据进行操作了

怎么使用Load-store指令

- 第一部分：把外部功能部件的所有控制寄存器和状态寄存器都映射到RAM空间上统一编址
- 第二部分：ARM通过load或store指令实现内部寄存器和外部映射RAM的空间的数据交换，来达到控制和读取外部功能部件的目的



指令码及状态字

指令码

特点 每条语句都可以有一条键码，每条语句的第二操作数都可以是以为操作数，每条语句都可以有一个状态位

ARM的初始化

1. 中断向量表
2. 初始化存储器系统
 - 存储器的类型和时序配置
 - 存储器的地址分布
3. 初始化堆栈

- ARM有7种执行状态，每一种状态堆栈指针寄存器（SP）都是独立的，对每一种模式都要定义SP寄存器的堆栈地址

4. 初始化有特殊要求的端口、设备

5. 初始化应用程序执行环境

- 完成必要的从ROM到RAM的数据传输

6. 改变处理器模式

7. CALL主程序

- 系统初始化工作完成后，程序流程转入主应用程序执行
- 可以直接从启动代码跳到应用程序的主函数入口

启动代码

系统启动流程：

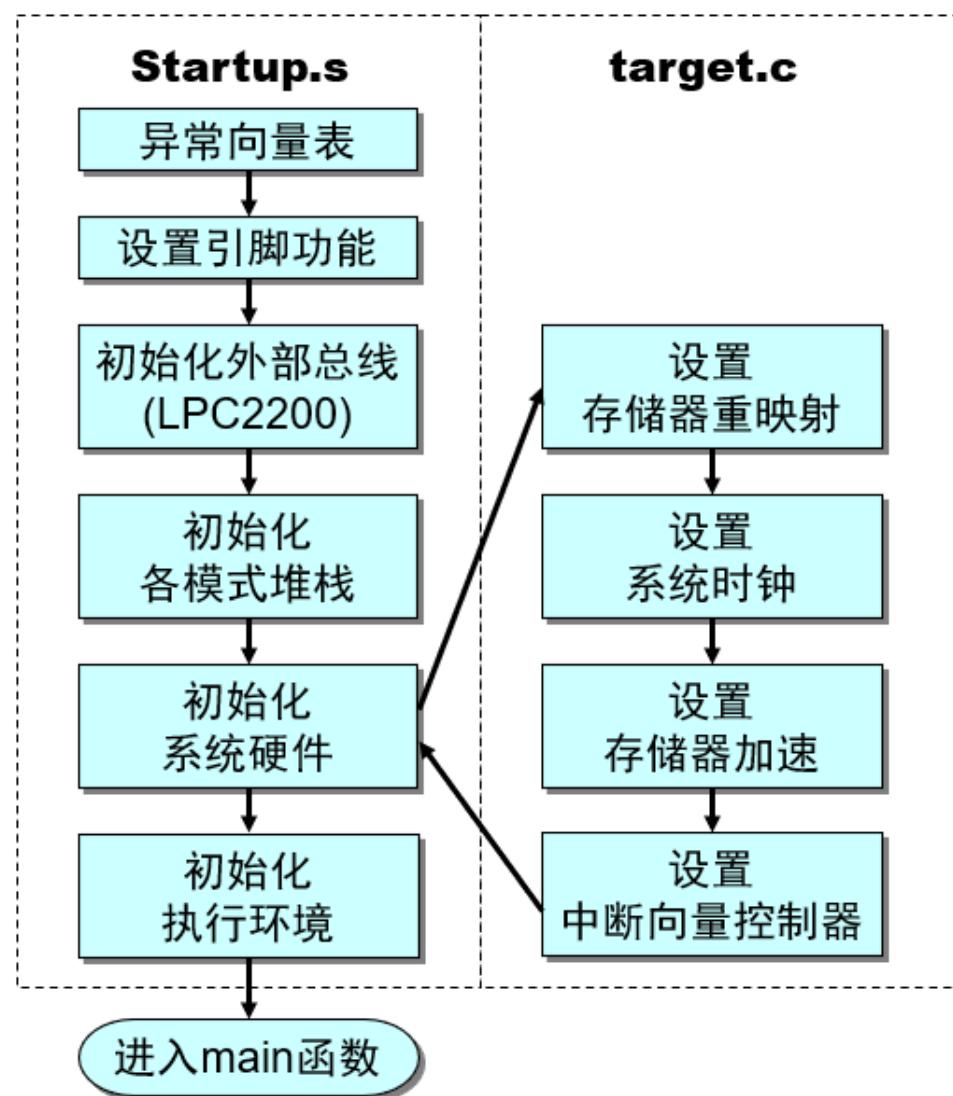
- 上电/复位
- 引导代码（bootblock）
- 启动代码
 - 向量表定义
 - 堆栈初始化
 - 系统变量初始化
 - 中断系统初始化
 - I/O初始化
 - 外围初始化
 - 地址重映射
- 用户main函数

Boot Block 引导块

boot block中的复位后运行的代码被称为引导代码

启动代码

• 启动代码流程图



初始化堆栈

ARM有7种执行状态，每一种状态堆栈指针寄存器（SP）都是独立的 对每一种模式都要定义SP寄存器的堆栈地址

初始化应用程序执行环境 完成必要的从ROM到RAM的数据传输

改变处理器模式 呼叫主应用程序

ARM核

请你用ARM和IP核设计一个CPU

由arm来扩展CPU的基本方法是把所有功能部件全部映射成内存（把挂在总线上的所有功能部件统一编址）然后每个功能部件的寄存器映射成内存的地址（或一个变量）ARM最多只能对本地的寄存器进行操作，但是它可以通过Load-store指令让自己的寄存器跟功能部件的寄存器进行数据交换，交换完数据进行操作，操作完再交换回去，这样就可以对其他地址部件的数据进行操作

```
LDR R0,[R1]
STR R0,[R1]
```

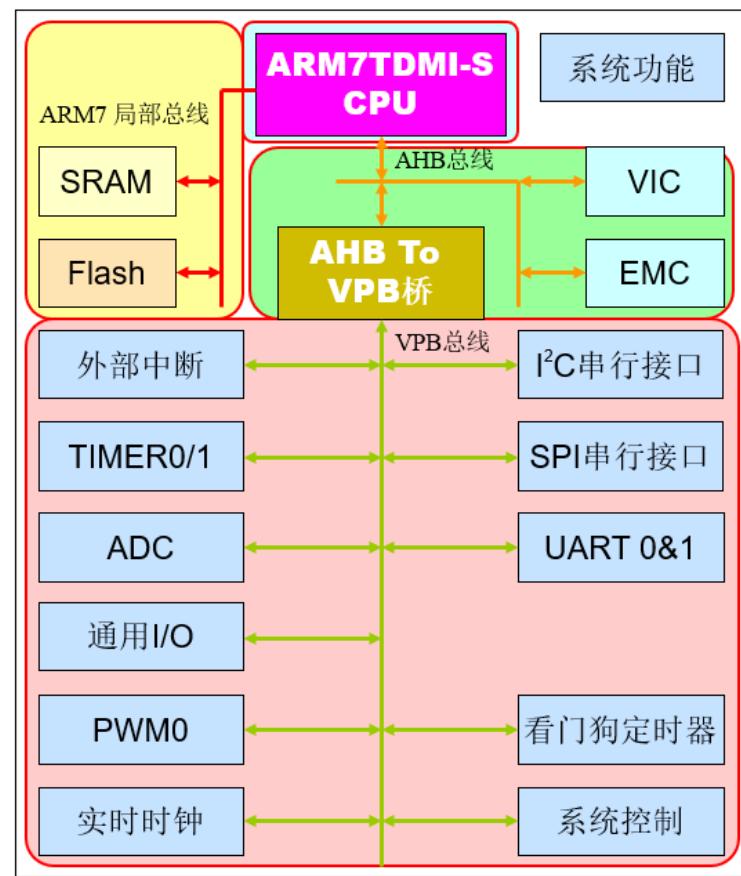
LPC芯片内部结构

内部总线、AHB总线、VPB总线 ARM局部总线、AHB高性能总线、VHB外设总线 AHB to VPB 的桥 将VPB总线和AHB总线相连，起缓冲作用（以空间换时间）

• 芯片内部结构

LPC2000系列微控制器包含
4大部分：

- ① 支持仿真的ARM7TDMI-S CPU
- ② 与片内存储器控制器接口的ARM7局部总线
- ③ 与中断控制器接口的AMB A高性能总线(AHB)
- ④ 连接片内外设功能的VLSI 外设总线(VPB)



VIC: 向量中断控制器

NVIC 嵌套向量中断控制器

EMC: 外部存储器控制器

LPC芯片的中断扩展

LPC的中断扩展，ARM只有两根中断线 IRQ中断、FIQ中断

需要用到硬件机构（电路），对外部的中断源进行仲裁，找出相应的中断源，再把该中断源的中断服务程序地址送给IRQ或FIR中断地址

存储器

存储器种类

SRAM：静态随机存取存储器 不需要刷新电路即能保存它内部存储的数据（体积大，容量小） 高速访问

DRAM :

- 只能将数据保持很短的时间 (大容量)
- 每个一段时间要刷新充电一次，否则内部数据会消失(周期性刷新)低速访问
- ROM 只读存储器
- PROM 可编程只读存储器
- EEPROM 可擦写可编程只读存储器
- EEPROM 电可擦写只读存储器
- OTPROM 一次可编程只读存储器
- flash 闪存
 - NOR-flash
 - NAND-flash

存储器映射

给物理地址分配逻辑地址的过程称为存储器映射

(上电时先运行Boot Block)

异常向量表从片外存储器中重映射

片内Flash编程

- JTAG下载程序
- 系统编程技术 (ISP) 通过串口下载程序
- 应用编程技术 (IAP) 在用户程序运行时对Flash进行擦除和/或编程操作，实现数据的存储和固件的现场升级

片外Flash映射

汇编使用

预定义寄存器

汇编器预定义的寄存器名称

R0~R15	ARM 处理器的通用寄存器
A1~A4	入口参数、处理结果、暂存寄存器；是 R0~R3 的同义词
V1~V8	变量寄存器，R4~R11
SB	静态基址寄存器，R9
SL	栈界限寄存器，R10
FP	帧指针寄存器，R11
IP	内部过程调用暂存寄存器，R12
SP	栈指针寄存器，R13
LR	链接寄存器，R14
PC	程序计数器，R15
CPSR	当前程序状态寄存器
SPSR	程序状态备份寄存器
F0~F7	浮点数运算加速寄存器
S0~S31	单精度向量浮点数运算寄存器
D0~D15	双精度向量浮点数运算寄存器
P0~P15	协处理器 0~15
C0~C15	协处理器寄存器 0~15

10

相互调用

ARM和Thumb的调用

ARM指令例程代码：

- 先对内存地址**0x3000**开始的**100**个字内存单元填入**0x10000001~0x10000064**字数据，然后将每个字单元进行**64**位累加结果保存于**[R9:R8]**。（**R9**中存放高**32**位）

- _start:
 - MOV R0 , #0X3000 @初始化寄存器
 - MOV R1 , #0X10000001
 - MOV R2 , #100
- loop_1: @第一次循环赋值
 - STR R1 , [R0],#4
 - ADD R1 , R1,#1
 - SUBS R2 , R2,#1
 - BNE loop_1

16

- MOV R0 , #0X3000
- MOV R2 , #100
- MOV R9 , #0
- MOV R8 , #0
- loop_2: @第二次循环累加
 - LDR R1 , [R0],#4
 - ADDS R8 , R1,R8@R8=R8+R1,进位影响标志位
 - ADC R9 , R9 , #0@R9=R9+C,C为进位
 - SUBS R2 , R2 , #1
 - BNE loop_2

17

汇编中调用C函数

声明:

在GUN ARM编译环境中，汇编程序中要使用.extern伪操作声明将要调用的C程序 在ARM开发工具编译环境下，

汇编程序中要使用IMPORT伪操作声明将要调用的C程序

PPT实例：

- 在**ARM**开发工具编译环境下设计程序，用**ARM**汇编语言调用**C**语言实现**20!** 的阶乘操作，并将**64**位结果保存到寄存器**R0、R1**中，其中**R1**中存放高**32**位结果。
- 首先建立汇编源文件**start.s**

```
/* start.s */  
IMPORT Factorial ;声明 Factorial 是一个外部函数  
Ni EQU 20 ;要计算的阶乘数  
AREA Fctrl, CODE, READONLY ; 声明代码 Fctrl  
ENTRY ; 标识程序入口  
  
start  
    MOV R0,#Ni ;将参数装入 R0  
    BL Factorial ;调用 Factorial， 并通过 R0 传递参数  
    /*注：在此处观察结果*/  
  
Stop  
    B Stop  
END ;文件结束
```

■ 然后建立C语言源文件**factorial.c**

```
/* factorial.c */
long long Factorial(char N)
{
    char i;
    long long Nx=1;
    for(i=1;i<=N;i++)Nx=Nx*i;
    return Nx; //通过 R0, R1 返回结果
}
```

■ 程序运行结果如下：

■ **R0 = 0x82B40000**

■ **R1 = 0x21C3677C**

个人代码实现：

```
-source group
-fatorial.c
-start.s
```

factorial.c

```
long long factorial(char N) {
    char i;
    long long Nx = 1;
    for(i = 1;i <= N;i++)
    {
        Nx = Nx * i;
    }
    return Nx;
}
```

start.s

```
IMPORT factorial ; 导入外部函数

Ni EQU 5 ; 定义常量

AREA fctrl, CODE, READONLY ; 定义代码段
ENTRY ; 声明程序入口点
EXPORT __main ; 导出入口符号 (如果需被链接器识别)

__main ; 标签必须顶格

MOV R0, #Ni ; 传递参数
BL factorial ; 调用阶乘函数

Stop ; 标签必须顶格
B Stop ; 无限循环

END ; 汇编结束
```

C语言中调用汇编

声明:

- 在GUN ARM编译环境下，在汇编程序中要使用.global伪操作声明汇编程序为全局的函数，可被外部函数调用，同时在C程序中要用关键字extern声明要调用的汇编语言程序。
- 在ARM开发工具编译环境上，汇编程序中要使用EXPORT伪操作声明本程序可以被其他程序调用。同时也要在C程序中要用关键字extern声明要调用汇编语言程序。

PPT例子:

■ (2) 在**ARM**开发工具编译环境下设计程序
，用**C**语言调用**ARM**汇编语言实现**20**的阶乘
(20！)操作，并将**64**位结果保存到
0xFFFFFFFF0开始的内存地址单元，按照小
端格式低位数据存放在低地址单元。

■ 每一步：建立启动C程序的代码，请读者参阅前面的章节自行建立。

■ 每二步：建立C语言源文件**main.c**，与**GNU ARM**编译环境下相同。

```
/* main.c */
extern void Factorial(char Nx); //声明 Factorial 是一个外部函数
__main()
{
    char N = 20;
    Factorial(N); //调用汇编文件实现 N!操作

    /*注：在此处观察结果*/
    while(1);
}
```

■ 每三步：建立汇编源文件**Factorial.s**

```
/* Factorial.s */
AREA Fctrl, CODE, READONLY ; 声明代码段 Fctrl
EXPORT Factorial

Factorial
    MOV     R8, R0          ;取参数
    MOV     R9, #0           ;高位初始化
    SUB     R0, R8, #1       ;初始化计数器

Loop
    MOV     R1, R9           ;暂存高位值
    UMULL   R8, R9, R0, R8   ;[R9:R8]=R0*R8
    MLA     R9, R1, R0, R9   ;R9=R1*R0+R9
    SUBS   R0, R0, #1        ;计数器递减
    BNE    Loop             ;计数器不为 0 继续循环
    LDR    R0, =0xFFFFFFFFF0 ;结果保存到 0xFFFFFFFFF0 开始的内存单元
    STMIA R0, {R8, R9}
    MOV    PC, LR
    MOV    PC, LR            ;子程序返回
```

■ 程序运行结果如下：

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0xFFFFFFF80	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0xFFFFFFF90	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0xFFFFFFF90	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0xFFFFFFF90	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0xFFFFFFFF0	00	00	B4	82	7C	67	C3	21	00	00	00	00	00	00	00	00

个人代码:

```
-source group
-fatorial.s
-main.c
```

main.c

```
extern void factorial(char Nx);
void __main()
{
    char N = 5;
    factorial(N);

    while (1)
    {
        /* code */
    }
}
```

fatorial.s

```
AREA FactorialCode, CODE, READONLY
EXPORT factorial ; 导出符号供外部调用

; 阶乘函数: 计算 R0 的阶乘, 结果通过 R0 返回 (32位)
; 输入: R0 = n (非负整数)
; 输出: R0 = n!
factorial
    PUSH {R4-R5, LR} ; 保存需要使用的寄存器和返回地址 (LR)

    ; 初始化变量
    MOVS R4, R0 ; R4 = n (当前乘数)
    MOVS R5, #1 ; R5 = 累积结果 (初始为1)

    ; 处理 n=0 的特殊情况
    CMP R4, #0
    BEQ return

loop
    ; 计算乘积: R5 = R5 * R4
    MUL R5, R5, R4 ; Thumb-2 指令 (等同于 UMULL 但更高效)

    ; 递减乘数并检查是否继续循环
    SUBS R4, R4, #1 ; R4 -= 1, 同时更新标志位
    BGT loop ; 如果 R4 > 0, 继续循环
```

```

return
    ; 返回结果到 R0
    MOVS    R0, R5

    ; 恢复寄存器并返回
    POP     {R4-R5, PC}    ; 直接弹出 PC (等效于 BX LR)

    END          ; 汇编结束

```

嵌入式C的编程规范

大致关键字：

排版

- 空格缩进
- 变量前后带空格
- 程序块前后带空行
- 长语句分多行书写，低优先级处划分新行，操作符在新行之首，新行要适当缩进
- 不能把多个短句写在一行
- 条件，循环语句应该单独在一行，而且无论其执行语句有多少都应有{}
- 函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格，case语句也要有缩进
- 一行程序应小于80字符为准，不要写的过长

注释

- 一般情况下，程序的有效注释量必须在20%以上
- 文件头部应该进行注释，注释必须列出：版权说明，版本号，生成日期，作者，内容，功能，修改日志等
- 函数头部应进行注释，列出；函数的目的/功能，输入参数，输出参数，返回值，调用关系等
- 保证注释与代码的一致性
- 防止注释二义性
- 注释应临近相关代码，放在上方或者右方
- 对于不能自注释的命名变量，常量，说明其物理含义，变量、常量、宏的注释应放在其上方相邻位置或右方
- 数据结构声明，同样应该给予注释
- 全局变量要有较详细的注释，包括功能，取值范围，哪些函数存取，注意事项等等
- 注释与所描述内容进行同样的缩排
- 避免在一行代码或表达式中间插入注释
- 通过对函数或过程。变量。结构的正确命名和结构使得代码自注释
- 注释格式尽量统一

标识符

- 标识符的命名要清晰，明了，有明确含义，使用公认缩写比如：
 - temp tmp
 - flag flg

- statistic stat
- increment inc
- message msg
- 命名中如果有特殊约定或缩写，则要有注释说明
- 自己特有的命名风格，要自始至终保持一致，不可来回变化
- 对于变量命名，禁止区单个字符（除非作局部循环命名）
- 非必要不要使用数字或奇怪的字符定义标识符
- 使用正确的反义词来命名互斥的变量动作
- 除了编译开关，头文件，应避免使用下划线开始结尾的定义

可读性

- 注意运算符的优先级，并用括号明确表达式的操作顺序
- 代码中尽量使用有意义的枚举或宏来代替
- 源程序中关系紧密的代码应可能相邻
- 不要使用难懂的写法

变量、结构

- 去掉没必要的公共变量
- 仔细定义并明确公共变量的含义，作用，取值范围，变量间关系
- 当向公共变量传递数据时，要十分小心，防止赋予不合理的值或越界
- 防止局部变量与公共变量
- 严禁使用未经初始化的变量作为右值
- 数据结构的功能要单一
- 不要涉及面面俱到，非常灵活的数据结构
- 不同结构的关系不要过于复杂
- 结构中的元素个数应适中，可以考虑使用子结构
- 设计结构中的元素的布局与排列顺序，使结构容易理解，节省占用空间
- 注意数据类型的强制转换
- 对编译系统默认的数据类型转换要有认识
- 合理设计自定义数据类型
- 合理命名以自注释

函数、过程

- 对所调用函数的错误返回码要仔细，全面的处理
- 明确函数功能，精确实现函数设计
- 编写可重入函数，注意局部变量（避免static）
- 编写可冲入函数时，使用中断，信号量等来使用全局变量
- 函数规模限制在200行以内
- 一个函数仅仅完成一个功能
- 函数的功能应该是可以预测的（尽量）
- 避免使用BOOL参数，因为返回值含义模糊且不利于扩充
- 对于有返回值的函数，引用时最好使用其返回值

嵌入式C语言相关知识点

volatile关键字

ucos操作系统

相关

高效

- 能做的事情先做（初始化，数据结构）
- 查表操作
- 大量位运算
- ucosii就绪表的三个算法

临界区

代码的临界段也称为临界区，指处理时不可分割的代码

一旦这部分代码开始执行，则不允许任何中断打断，为确保临界区段代码的执行，在进入临界区前需要关中断，而临界区代码执行完以后要立即开中断

RTOS

实时操作系统：能够在指定或者确定的时间内完成系统功能以及对外部或内部事件在同步或异步时间内做出响应的系统

分类

- 软实时系统 各个任务运行的越快越好，不要求限定某一任务必须在多长时间内完成
- 硬实时系统 各个任务不仅要执行无误而且要做到准时。系统对系统响应时间有严格的要求，如果系统响应时间不满足，就会引起系统崩溃或致命的错误
- 硬实时系统

初始化

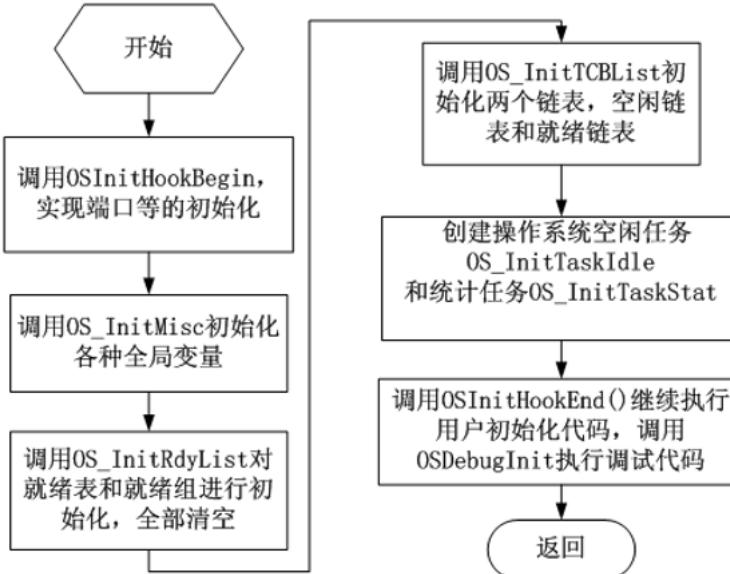
上电初始化工作：

1、操作系统上电初始化时作了哪些工作？（书本p271）

地址重映射、向量表定义、堆栈初始化、系统变量初始化、中断系统初始化



上电/复位 → 引导代码（Boot Block）→ 启动代码 → 用户 main 函数



答：OSInit() 初始化μC/OS-II 所有的变量和数据结构；
 OSInit() 建立空闲任务idle task，这个任务总是处于就绪态的；
 μC/OS-II 还初始化了4个空数据结构缓冲区，每个缓冲区都是单向链表，允许μC/OS-II 从缓冲区中迅速得到或释放一个缓冲区中的元素。

OSinit 初始化所有的变量和数据结构 OSinit 建立空闲任务idle task，这个任务总是处于就绪态 还初始化了4个空数据结构缓冲区，每个缓冲区都是单向链表，允许os从缓冲区中迅速得到或释放一个缓冲区中的元素

任务控制块TCB

任务控制块 OS_TCB 是一个数据结构，保存该任务的相关参数，包括任务堆栈指针，状态，优先级，任务表位置，任务链表指针等 所有任务控制块分为两条链表，空闲链表和使用链表

相关数据结构

包含了任务执行过程中所需要的所有信息

- 任务的名字
- 任务执行的起始地址
- 任务的优先级
- 任务的状态
- 任务的硬件上下文（堆栈指针，PC, 寄存器等）
- 任务的队列指针等

任务的组成：

- 代码 一段可执行的程序
- 数据 程序所需要的相关数据（变量、工作空间、缓冲区）
- 堆栈
- 程序执行的上下文环境
 - 包含了实时内核管理任务，以及处理器执行任务所需要的所有信息
 - 任务优先级

- 任务的状态等实时内核所需要的信息
- 以及处理器的各种寄存器的内容，程序计数器、堆栈指针、通用寄存器等的内容
- 任务的上下文环境通过任务控制块（TCB）来体现

优先级算法

```
OSTCBy      = priority >> 3;  
  
OSTCBBitY   = OSMapTbl[priority >> 3];  
  
OSTCBX      = priority & 0x07;  
  
OSTCBBitX   = OSMapTbl[priority & 0x07];
```

优先级反转

理想情况下，高优先级就绪后，能够立刻抢占低优先级任务而得到执行

但在有多个任务需要使用共享资源的情况下，可能会出现高优先级任务被低优先级任务阻塞，并等待低优先级任务执行完毕，从而导致高优先级任务无法得到执行

优先级反转：高优先级在等待低优先级释放资源，而低优先级任务又在等待中优先级任务的情况

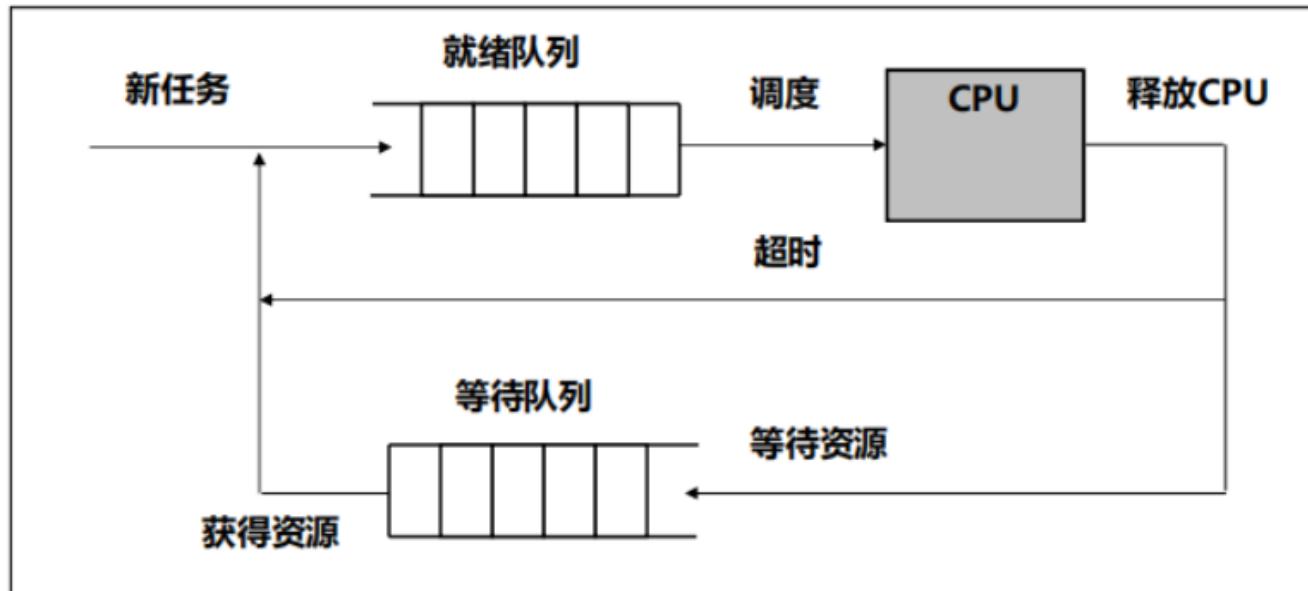
直接应用这些同步互斥机制将导致系统中出现不定时间长度的优先级反转和比较低的任务可调度性

解决：

1. 优先级继承
 1. 当一个任务阻塞了高优先级任务的时候，使用被阻塞的所有任务的最高优先级作为其执行临界区的优先级
 2. 他不能避免死锁问题
2. 优先级天花板
 1. 优先级天花板访问资源的信号量的优先级天花板（使用该信号量的任务的最高优先级）

任务队列

任务队列通过任务控制块实现对系统中所有任务的管理



单就绪队列和单等待队列

事件控制块 ECB

所有的通信信号都被看成是事件event,一个称为事件控制块ECB *event control block* 的数据结构来表征每一个具体事件。ECB的结构如下:

```

typedef struct {
    void *OSEventPtr;           /*指向消息或消息队列的指针*/
    INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; /*等待任务列表*/
    INT16U OSEventCnt;          /*计数器（当事件是信号量时）*/
    INT8U OSEventType;          /*事件类型：信号量、邮箱等*/
    INT8U OSEventGrp;           /*等待任务组*/
} OS_EVENT;
  
```

与TCB类似的结构，使用两个链表，空闲链表与使用链表

信号量实现机制

类似任务就绪表

将任务插入等待事件的任务列表中:

• 将一个任务插入到等待事件的任务列表中：

```
pevent->OSEventGrp  
|= OSMapTbl[prio >> 3];
```

```
pevent->OSEventTbl[prio >> 3]  
|= OSMapTbl[prio & 0x07];
```

从等待事件的任务列表中使任务脱离等待状态

```
if ((pevent->OSEventTbl[prio >> 3] &= ~OSMapTbl[prio  
& 0x07]) == 0) {  
    pevent->OSEventGrp &= ~OSMapTbl[prio >> 3];  
}
```

与将任务从就绪列表中清除的操作类似！

一些tips

什么是任务

任务包括：程序体、堆栈（内存区）、TCP（控制块）

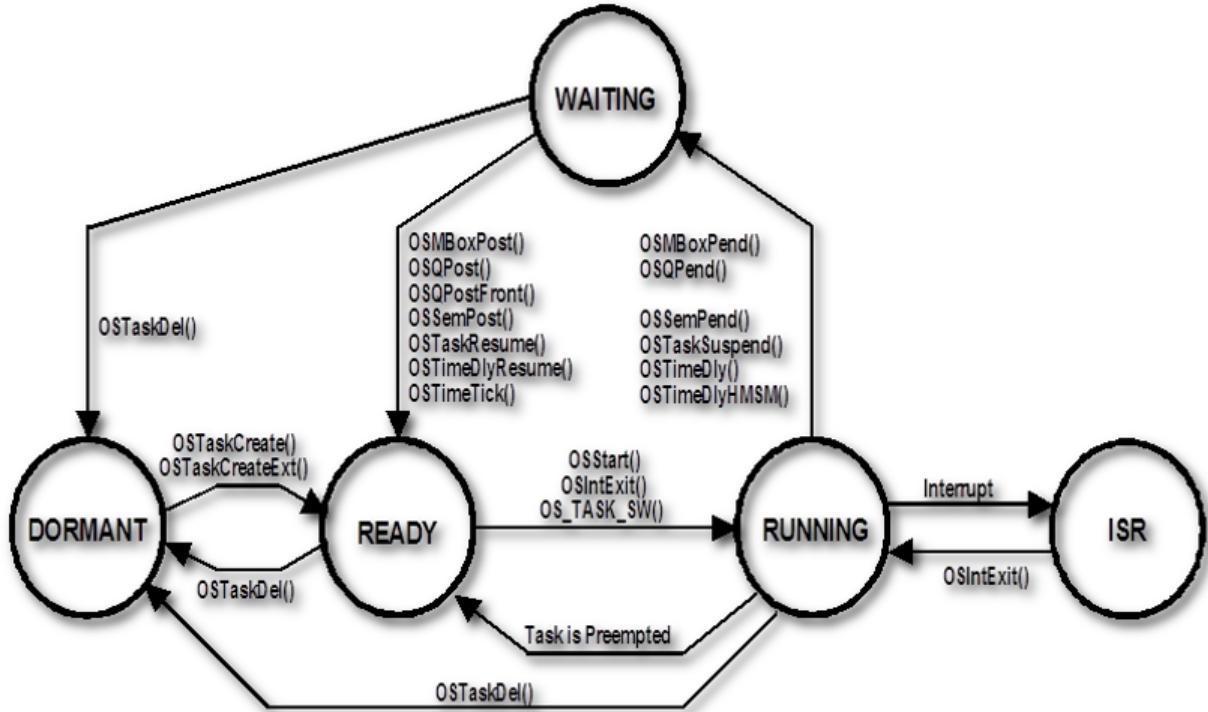
任务基本信息

- CPU中的PC寄存器：任务（程序）当前执行的位置
- CPU种的通用寄存器：惹怒我当前执行代码所涉及的临时数据
- CPU中的状态寄存器：存储当前CPU的状态

任务状态

- DORMANT(睡眠态)
- READY(就绪态)
- RUNNING(运行态)
- WAITING(等待态)

- ISR(中断服务态)



任务切换

1. 保存任务上下文
2. 更新当前运行任务的控制块内容，将其状态改为就绪或等待状态
3. 将任务控制块移到相应队列（就绪队列/等待队列）
4. 选择另一个任务进行执行（调度）
5. 改变另一个任务进行执行的控制块内容，将其状态变为运行状态
6. 恢复需投入运行任务的上下文环境

需要切换的情况：

- 中断、自陷
 - 如当IO中断发生时候
 - 如果IO活动是一个或多个任务正在等待的事件，内核就把相应的处于等待状态的任务转换为就绪状态
 - 同时内核还将确定是否继续执行当前处于运行状态的任务，或是用高优先级的就绪任务抢占该任务
 - 自陷
 - 由于执行任务中当前指令所引起，将导致实时内核处理相应的错误或异常事件，并根据事件类型，确定是否进行任务切换 查询信号量的当前状态（计数值、等待任务列表等）
- 运行任务因缺乏资源而被阻塞
 - 如有别的资源需要
- 时间片轮转调度时
 - 内核将在时钟中断处理程序中确定当前正在运行的任务的执行时间是否已经操作了设定的时间片
 - 如果超过了时间片，实时内核将停止当前任务的运行，把当前任务的状态变为继就绪状态，并把另一个任务投入运行

- 高优先级任务处于就绪时
 - 如果采用基于优先级的抢占式调度算法，将导致当前任务停止运行，使更高优先级的任务处于运行状态 **任务状态转换**：
- OSDly() 删除任务，进入休眠态
- OSTimeDly() OSMboxPend等进入等待态
- OSIntExit() 进入中断服务状态
- OSSStart() 让睡眠态的任务进入就绪态

事件

- 事件指的是一种表面预先定义的系统事件已经发生的机制
- 事件用于任务间，任务与ISR之间的同步
- 一个事件就是一个标志
- 一个及以上事件构成一个事件集

事件操作

- OSFlagCreate() 创建事件集
- OSFlagDel() 删除事件集
- OSFlagPend() 接受事件集
 - WAIT 接受事件集可等待
 - NO_WAIT 接受事件集不可等待
 - EVENT_ALL 与 事件集
 - EVENT_ANY 或 事件集
- OSFlagPost() 发送事件集
- OSFlagQuery() 查询事件集
- OSFlagAccept() 接受事件集

UCOS内核

内核

提供的基本服务是任务切换

调度

是内核的主要职责之一，决定该轮到哪个任务执行了。

基于优先级的调度法指的是，CPU总是让处在就绪态的优先级最高的任务先执行

任务级的任务调度：

- ucos是占先式实时多任务内核，优先级最高的任务一旦准备就绪，则拥有CPU的所有权开始投入运行
- ucos不支持时间片轮转法，每个任务的优先级要求不一样且是唯一的，所以任务调度的工作是：

查找准备就绪的最高优先级并进行上下文切换

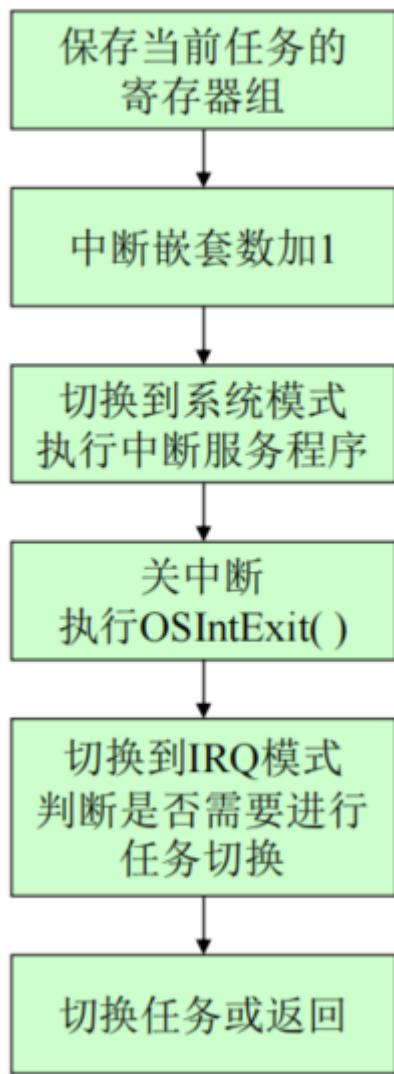
- ucos任务调度所花的时间为常数，与应用程序中建立的任务数无关

时间片轮转调度

当多个就绪任务具有相同优先级，则先后执行一段时间片，当时间片用完时，任务被调度到就绪队列的末尾，并重新开始执行

中断

流程图



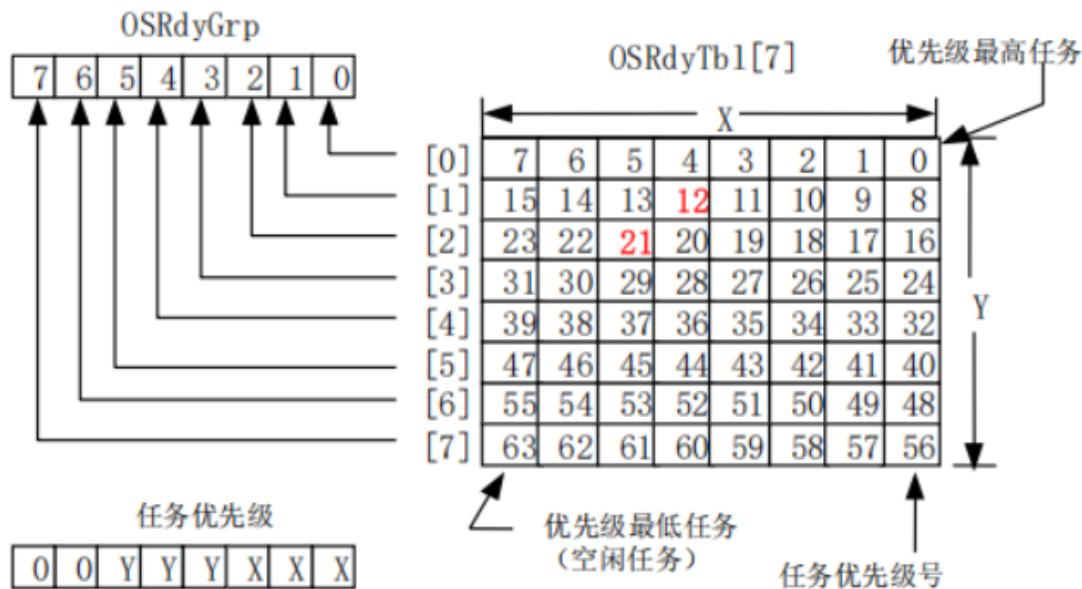
优先级算法

两个关键：

优先级分解为高三位和低三位分别确定 高优先级有着小的优先级号

就绪表：

每个就绪的任务都放入就绪表中 (ready list) 中，就绪表有两个变量：OSRdyGrp、OSRdyTbl[]



- 假设优先级为 12 的任务进入就绪状态， $12=1100b$, 则 OSRdyTbl[1]的第4位置1，且OSRdyGrp的第1位置1, 相应的数学表达式为:

OSRdyGrp |=0000 0010=0x02;

OSRdyTbl[1] |=0001 0000=0x10;

- 而优先级为21的任务就绪 $21=10\ 101b$, 则OSRdyTbl[2]的第5位置1, 且OSRdyGrp的第2位置1, 相应的数学表达式为:

OSRdyGrp |=0000 0100=0x04;

OSRdyTbl[2] |=0010 0000=0x20;

- 从上面的计算我们可以得到：若OSRdyGrp及OSRdyBbl[]的第n位置1，则应该把OSRdyGrp及OSRdyBbl[]的值与 2^n 相或。uC/OS中，把 2^n 的n=0-7的8个值先计算好存在数组OSMapTbl[]中，也就是：

$OSMapTbl[0] = 2^0 = 0x01 (0000 0001)$

$OSMapTbl[1] = 2^1 = 0x02 (0000 0010)$

.....

$OSMapTbl[7] = 2^7 = 0x80 (1000 0000)$

查表法：

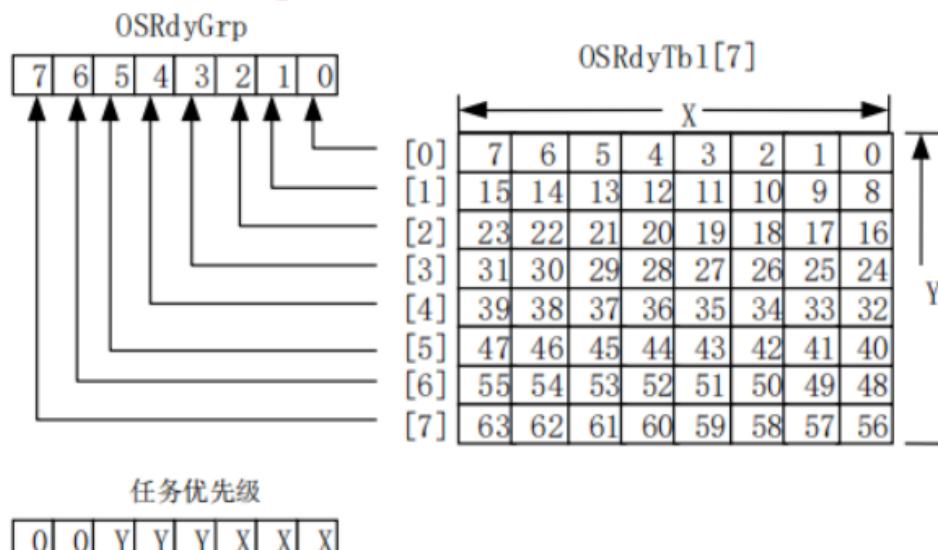
采用查表法确定高优先级任务

查表法具有确定的时间，增加了系统的可预测性，uC/OS中所有的系统调用时间都是确定的

$High3 = OSUnMapTbl[OSRdyGrp];$

$Low3 = OSUnMapTbl[OSRdyTbl[High3]];$

$Prio = (High3 << 3) + Low3;$



优先级判定表OSUnMapTbl [256]

```

INT8U const OSUnMapTbl[] = {
    0,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    6,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    7,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    6,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0
};


```

举例：

如OSRdyGrp的值为
01101000B，即0X68，则查
得
OSUnMapTbl[OSRdyGrp]
的值是3，它相应于
OSRdyGrp中的第3位置1；

如OSRdyTbl[3]的值是
11100100B，即0XE4，则查
OSUnMapTbl[OSRdyTbl[3]]
的值是2，则进入就绪态的最
高任务优先级

$$\text{Prio} = 3 * 8 + 2 = 26$$

代码：

```

char const table[] = {
    0,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    6,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    7,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    6,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0
};

/*

```

```

* @brief 获取就绪表中最高优先级任务的索引
* @param rdyTBL 就绪表数组，每个元素代表一个优先级组的就绪状态（非0表示有就绪任务）
* @return 最高优先级任务的索引（0~63），若无就绪任务返回-1
*/
int getHighestPriorityTable(char rdyTBL[8]) {
    // 初始化就绪组（RdyGrp），每个bit表示一个优先级组是否有就绪任务
    // 初始值0b11111111表示假设所有组都有任务（后续会修正）
    unsigned char RdyGrp = 0b11111111;

    // 遍历就绪表，构建真实的RdyGrp值
    // 从最高优先级组（rdyTBL[7]）开始检查
    for (int i = 7; i >= 0; i--) {
        if (rdyTBL[i] == 0)
            RdyGrp = (RdyGrp << 1) | 0; // 左移并补0：表示该组无就绪任务
        else
            RdyGrp = (RdyGrp << 1) | 1; // 左移并补1：表示该组有就绪任务
        // 等效于 RdyGrp = (RdyGrp*2 + 1);
    }

    // 通过查表法快速获取最高优先级组号（高3位）
    char high3 = table[RdyGrp]; // table为预定义的查找表，映射RdyGrp到最高优先级组号

    // 计算高优先级部分的基值（组号*8）
    char high = (char)(high3 << 3); // 左移3位等价于乘以8

    // 获取该组内最高优先级任务的偏移量（低3位）
    char low = table[rdyTBL[high3]]; // 再次查表，获取组内最高优先级位

    // 如果RdyGrp非零（表示存在就绪任务）
    if (RdyGrp)
        return high + low; // 组合高3位和低3位得到完整优先级索引
    else
        return -1; // 无就绪任务时返回-1
}

```

idle任务

空闲任务：OSTaskIdle,只通过计数器统计CPU时间

任务总是建立一个空闲任务，这个任务在没有其他任务进入就绪态的时候投入运行，这个空闲任务为最低优先级

信号量

信号量

相关函数：

```

OSSemCreate()
// 创建并初始化一个信号量

```

```
OSSemPend()
//请求信号量 (P操作) , 若信号量不可用则任务阻塞

OSSemPost()
//释放信号量 (V操作) , 唤醒等待任务

OSSemAccept()
//非阻塞方式尝试获取信号量, 立即返回结果。

OSSemQuery()
//查询信号量的当前状态 (计数值、等待任务列表等)
```

信号量种类:

1. Binary Semaphore 二进制信号量
2. Counting Semaphore 计数信号量
3. Mutex Semaphore 互斥信号量

死锁

多个任务在运行中因争夺资源而造成的一种僵局 原因：竞争资源，任务推进非法

举例：

P1:request(D1),request(D2); (1)
P1:release(D1),release(D2); (2)

P2:request(D2),request(D1); (3)
P2:release(D2),release(D1); (4)

D1、D2是临界资源。-互斥机制实现临界资源只有一个任务访问
**当运行到P1:request(D2)时， P1占用D1,但D2已经被P2占用，
P1阻塞；**
**当运行到P2:request(D1)时， 也将因为D1已被P1占用而阻塞，
产生死锁(条件： P1:request(D1), P2:request(D2))。**

说明：P1先占有D1资源，P2先占有D2资源，然后P1申请D2资源，P2申请D1资源，P1和P2都等待D1和D2资源，造成死锁

PV操作

PV操作的信号量是全局的

P:pend 等待

OSSemPend() 减少信号量的值 如果新的信号量的值不大于0，则操作阻塞

V:post 释放

OSSemPost() 增加信号量的值

互斥

一个任务持有信号量时候，其他不能获取该信号量，释放之后，其他任务可以获取该信号量

互斥机制

比较项目	关中断	使用测试并置位指令	禁止任务切换	使用信号量
锁定范围	互斥力度最强 , 锁定所有外部可屏蔽中断，凡是 以中断形式到达的外部事件以及 与之相关联的任务或处理过程均 得不到执行	凡是使用该 指令访问共 享资源的代 码	所有的任务	只影响竞争 共享资源的 任务
对系统响应时间的影响	如果关中断的时间较长，对系统的响应性能有很大影响	较小	如果禁止切换的时间过长，则影响系统的响应性能	对系统响应性能有一定影响，可能导致优先级反转
实现时的系统开销	小	小	小	较大 (创建信号量， 将等待信号量的任务加入 等待队列，优先级反转的策略)
注意事项	关中断时间要尽量短	不是所有的处理器都具备这种指令，影响可移植性	关调度的时间要尽量短	需采用一定的策略解决优先级反转问题

如何分辨

任务等待和发送的信号量是同一个，就是互斥

一般使用互斥信号量

同步

二值信号量主要用于任务与任务之间、任务与中断服务程序之间的同步

如何分辨

一个任务释放信号量，另外一个任务等待该信号量，就是同步

一般使用二值信号量

ucos中使用邮箱作为二值信号量

生产消费问题/进餐问题

指若干进程通过有限的共享缓冲区交换数据时的缓冲区资源使用问题

PPT举例：

Product: in(0—n-1)

repeat

```
produce an item in nextp;  
while counter=n do no-op;  
buffer [ in ]:=nextp;  
in:=in+1 mod n;  
counter:=counter+1;
```

until false;

Consumer: out (0—n-1)

repeat

```
while counter=0 do no-op;  
nextc:=buffer[ out ];  
out:=(out+1) mod n;  
counter:=counter-1;  
consumer the item in  
nextc;
```

until false;

```
Register1=counter;  
Register1=Register1+1;  
counter=Register1;
```

counter=5

Register1=counter;	(Register1=5)
Register1=Register1+1;	(Register1=6)
Register2=counter;	(Register2=5)
Register2=Register2-1;	(Register2=4)
counter=Register1;	(counter=6)
counter=Register2;	(counter=4)

```
Register2=counter;  
Register2=Register2-1;  
counter=Register2;
```

Product,Consumer分别执行或顺序执行是正确的

Product,Consumer并发执行出错了，交叉执行的顺序改变，counter的值又变化了，程序的执行失去了再现性

解决办法：

解决的办法：

- 1、将变量**mutex**作为临界资源处理，使得生产者任务和消费者任务互斥地访问变量**mutex**，初值为1。（互斥信号量）
- 2、设置生产者私用信号量**empty**，使得生产者任务之间共享缓冲区：初值为n，指示空缓冲块数目。（计数信号量）
- 3、设置消费者私用信号量**full**，使得消费者任务之间共享缓冲区：初值为0，指示满缓冲块数目。（计数信号量） **full+empty=n**

Product:

```
repeat
    produce an item in nextp;
    wait(empty); (wait(mutex);)
    wait(mutex); (wait(empty);)
    buffer [ in ]:=nextp;
    in:=in+1 mod n;
    singal(mutex);
    signal(full);
until false;
```

Consumer:

```
repeat
    wait(full);
    wait(mutex);
    nextc:=buffer[ out ];
    out:=(out+1) mod n;
    singal(mutex);
    signal(empty);
    consumer the item in nextc;
until false;
```

1、每个程序中**wait(mutex)** **singal(mutex)**必须成对出现

2、对资源信号量**empty**和**full**的**wait**和**signal**操作，同样要成对出现，分处在不同的程序中。

任务若因执行**wait(empty)**而阻塞，则可以通过**signal(empty)**将其唤醒。

3、多个**wait**操作顺序不能颠倒，应先执行资源信号量的**wait**操作，然后是互斥信号量的**wait**操作，否则可能引起死锁。出现死锁的条件是，申请到对整个缓冲区的互斥操作后，才发现自己对应的缓冲块资源，这时已不可能放弃对整个缓冲区的占用。

生产者任务	消费者任务
do	do
{	{
...	申请full
产生一个数据项	申请mutex
...	...
申请empty	从缓冲中移出一个数据项的内容
申请mutex	...
...	释放mutex
将新生成的数据项添加到缓冲中	释放empty
...	...
释放mutex	消费新获得的数据项内容
释放full	...
} while (1);	} while (1);

计数信号量full：已被填充的数据项数目，取值范围0 - n，初始值为0

计数信号量empty：空闲数据项数目，取值范围为0 - n，初始值为n；

互斥信号量mutex：控制生产者任务和消费者任务对有界缓冲的访问，初始值为1。

12

个人理解：首先，要注意每个数据是块，一次操作要完成整个数据的搬运，不能搬运到一半

消息机制

消息机制的意义

使得两个任务独立性增强，耦合比较松散 消息机制在任务和任务之间，任务和中断服务之间提供消息传送（通信）机制

应用可以只把消息当成一个标志，这时消息机制用于实现同步

邮箱仅能存放单条消息，它提供了一种低开销的机制来传送消息 消息队列可存放若干消息，提供了一种任务间缓冲通信的方法

概述

- 共享数据结构（全局变量，指针等）
- 消息系统
 - 消息队列
 - 邮箱
- 管道通信

邮箱

- 仅能存放单条消息
- 区别于信号量
 - 消息邮箱可以存放一条完整的内容信息
 - 信号量进行同步时候不能提供内容信息

函数

- OSMboxcreate()
- OSMboxPost()
- OSMboxPend()

消息队列

任务等待消息时的排列方式：FIFO或优先级

函数

- OSQAccept() 非阻塞方式检查消息队列中是否有消息，立即返回消息指针或 NULL
- OSQDel() 删除消息队列并释放资源
- OSQCreate() 初始化一个消息队列，返回事件控制块指针
- OSQpost() 将消息插入队列尾部（FIFO）
- OSQPend() 阻塞当前任务，直到队列中有消息或超时
- OSQPostFront() 将消息插入队列头部（LIFO），优先被接收
- OSQQuery() 获取队列信息（如消息数、等待任务数等），存入 OS_Q_DATA 结构体：
- OSQFlush() 清空队列中所有消息，**重置为空状态**

环形缓冲

内存管理

原始内存管理

C语言中,堆（heap）和栈（stack）：

- malloc() 分配
- free() 释放

内存碎片

- 频繁分配和释放不同大小的内存块后，剩余的内存块被分割为多个不连续的小块，无法满足较大内存块的分配需求，尽管总空闲内存足够
- 内存分配对其和固定块大小的浪费（强制内存对其，比如4字节对齐）

动态内存管理

ucos的内存管理以块为单位，一种大小的（1，2，4）个字节的数据分配到同一个块

垃圾回收

函数

- OSMemCreate() 动态创建一个内存分区，将其划分为多个固定大小的块
- OSMemPut() 将不再使用的内存块归还到分区，供后续分配
- OSMemGet() 从分区中获取一个空闲内存块，返回其指针
- OSMemQuery() 得到内存区的信息

ucos函数解析

时间管理

OSTimeDly()

任务调用 进入延时，同时发生任务调度 延时时间到 任务进入就绪状态 当前任务优先级为就绪态中最高 任务进入运行状态

OSTimeDlyHMSM()

小时H：分M：秒S：毫秒MS

OSTimeDlyResume()

让延时的任务结束延时 通过取消其他任务的延时来使自己处于就绪态 注意，因为OSTimeDlyHMSM()通过多次或一次调用OSTimeDly(),所以可能需要调用多次OSTimeDlyResume()才能恢复延时的任务

OSTimeGet()

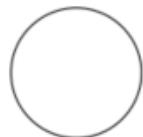
获取当前(系统)时间 ucos有一个32位的计数器，会按照时钟节拍递增

OSTimeSet()

设置当前(系统)时间

DARTS流图

基本图形符号解析



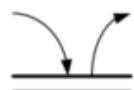
加工。输入数据在此进行变换产生输出数据，其中要标明加工的名字。



数据输入的源点或数据输出的终点。其中要标明源点或终点的名字。

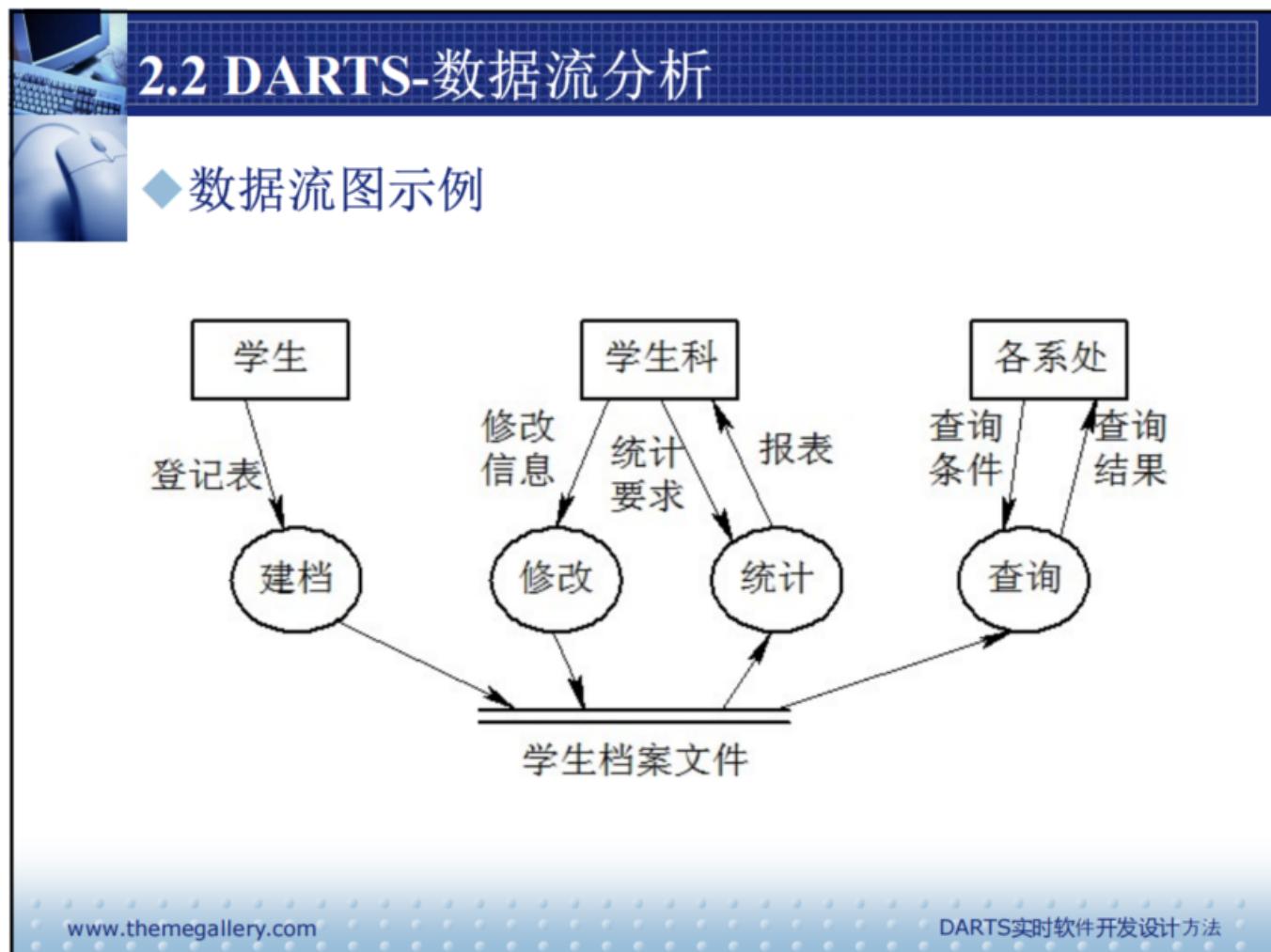


数据流。被加工的数据与流向，箭头边应给出数据流名字，可用名词或名词性短语命名。

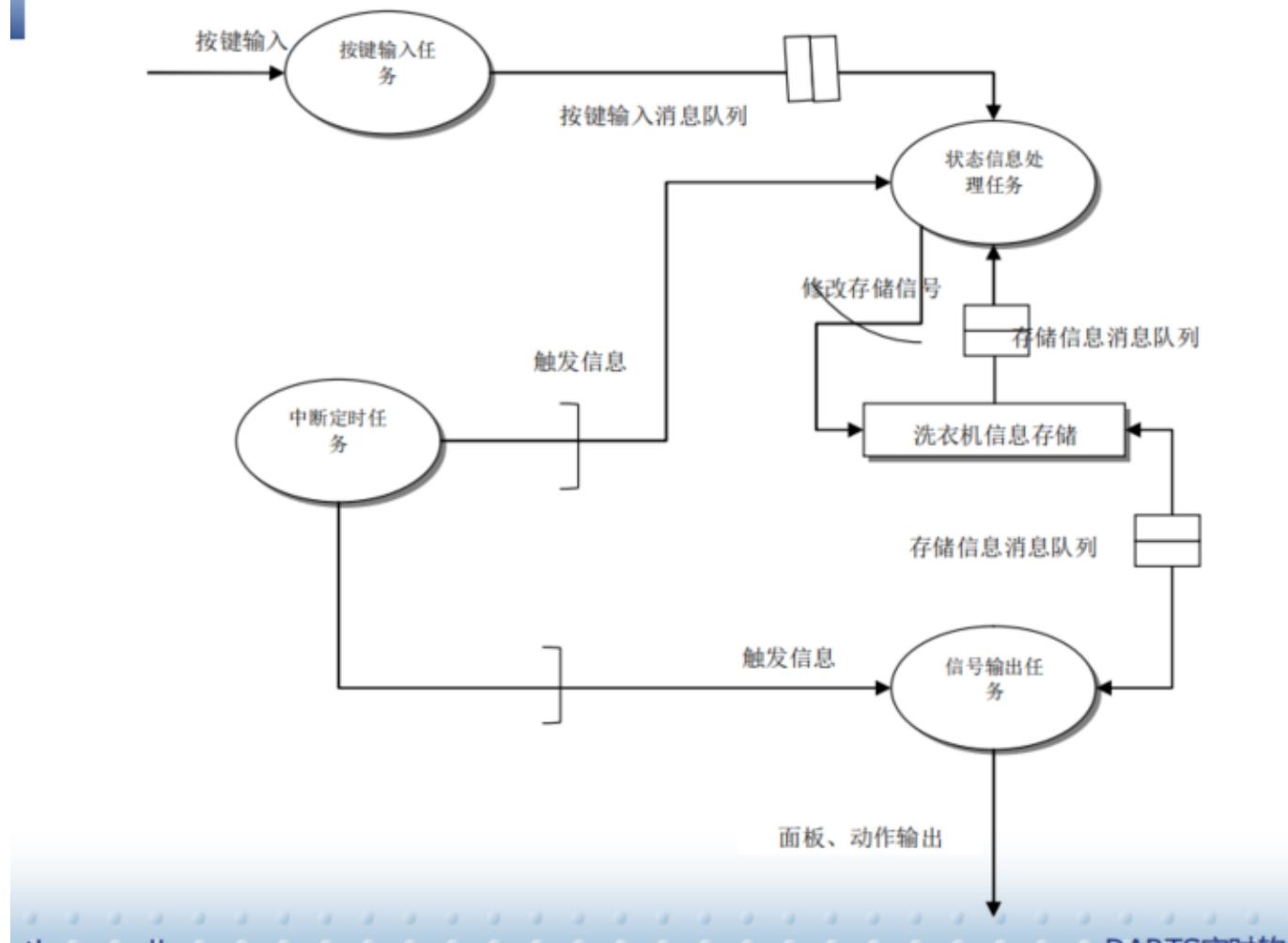


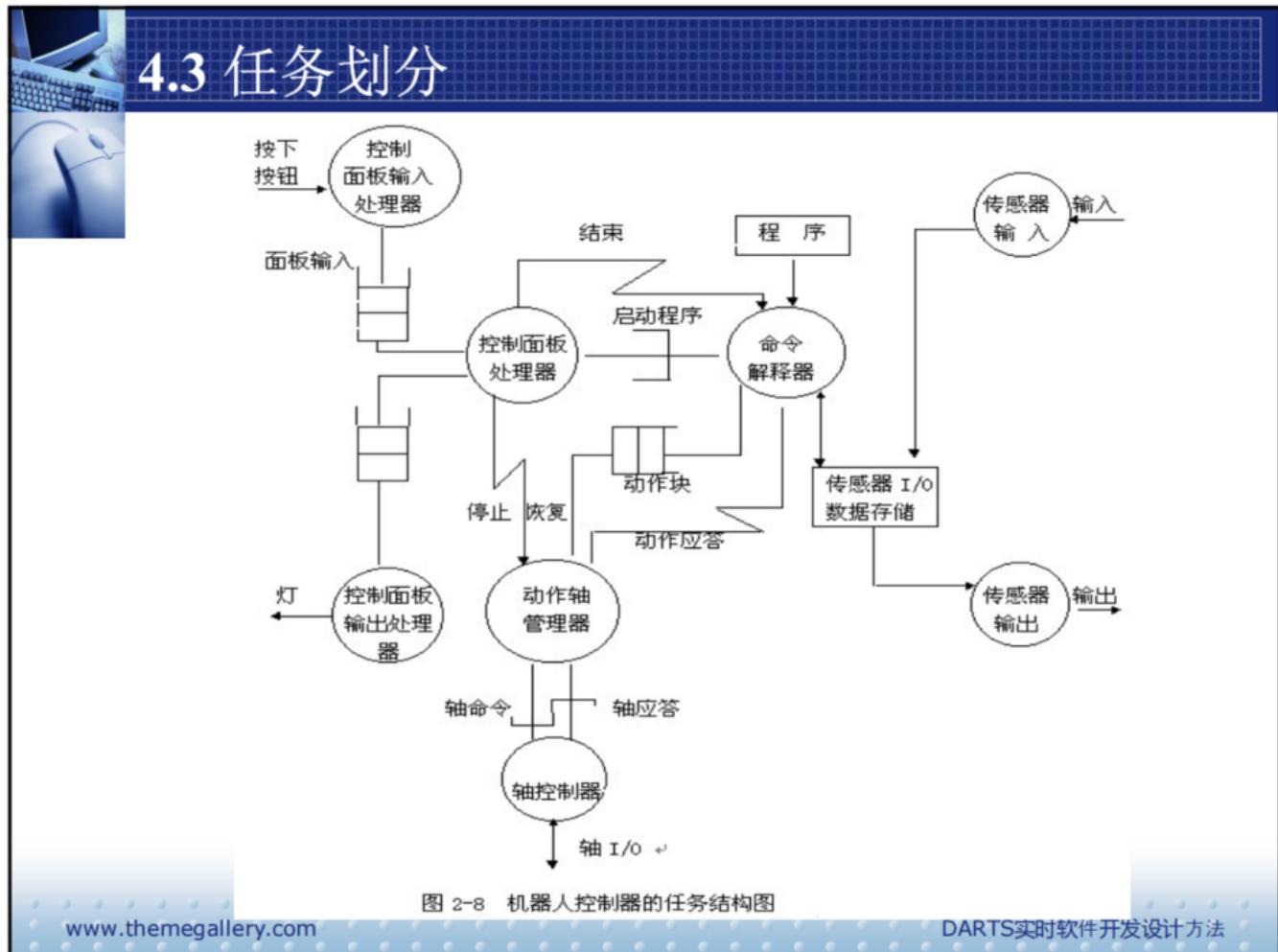
数据存储。必须加以命名，用名词或名词性短语命名。

数据流图示例



◆ 洗衣机控制软件任务接口



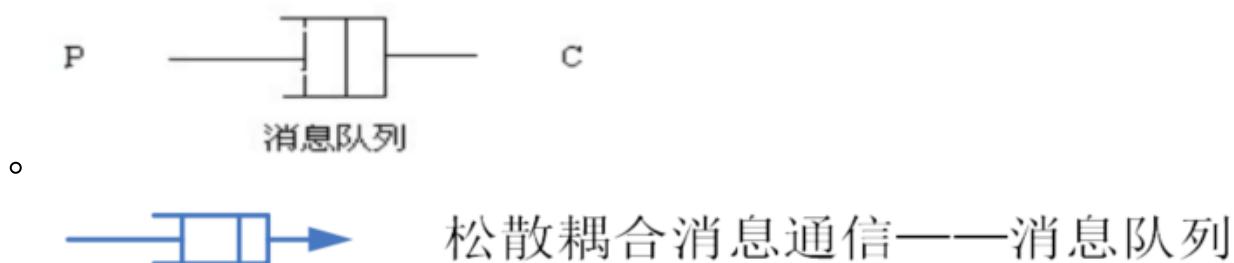


任务划分

- I/O 依赖性
 - 依赖，受限于 I/O，则独立成任务
 - I/O 设备创建等数量的 I/O 任务
 - I/O 任务只实现设备相关
 - 其速度取决于 I/O 设备速度，而非处理器
- 功能的时间关键性
- 计算需求
- 功能内聚
- 时间内聚
- 功能的周期执行

rtos 相关

- 松耦合（消息队列）



- 任务



- 消息



紧密耦合消息通信——消息/回复

- 总



任务



信息隐藏模块



松散耦合消息通信——消息队列



无回复紧密耦合消息通信



紧密耦合消息通信——消息/回复



事件

。

跳转

[跳转链接list点这里](#)