目录

1

2

实时操作系统基础知识

3

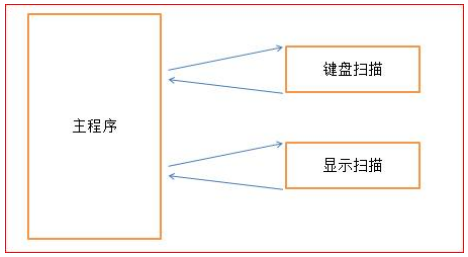
嵌入式软件设计的演变

- ■顺序程序设计
- ■基于状态机的程序设计
- ■基于简易任务调度器的程序设计
- ■基于操作系统的程序设计

4

顺序执行程序

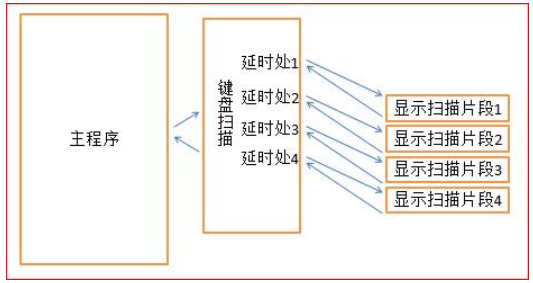
- ■顺序调用任务，执行完一个任务后再执行下一个任务。
- ■若任务长时间占用CPU，那么其它任务对外部事件的响应全部停止。



5

改进　将浪费的时间利用起来

- ■仔细观察可发现，其实任务并非一直运行，大部分时间是在延时。如果将任务**从延时处折断，分拆成小片段**后插入到另一个任务中，取代原有的延时程序，就可以提高系统资源的利用率。



6

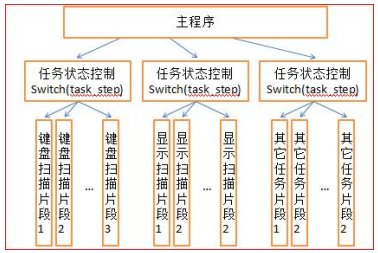
继续改进　实现流程控制---状态机

■ 在主程序与任务之间增加一个接口：任务状态控制器。主程序只与任务的状态控制器打交道，由状态控制器负责调用任务的片段以及控制阶段的变换。

■状态机：

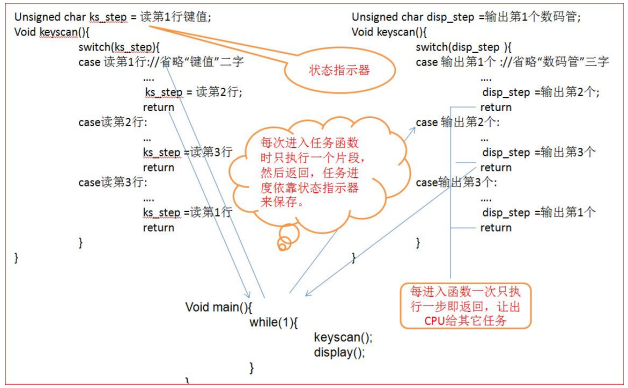
优点：占用资源少，执行效率高。

缺点：任务被拆得支离破碎，流程不直观。



7

状态机示例



8

实时系统概念

■实时系统特点：如果逻辑和时序出现偏差将会引起严重后果的系统。

■有两种类型的实时系统：软实时系统和硬实时系统。

■软实时系统：各个任务运行得越快越好，并不要求限定某一任务必须在多长时间内完成。

■硬实时系统：各任务不仅要执行无误而且要做到准时。

■大多数实时系统是**二者的结合**。

■实时应用软件的设计一般比非实时应用软件设计难。

9

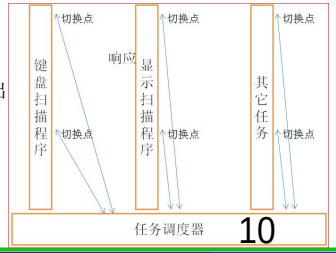
任务切换与任务调度

■**与顺序执行不同**的是：在执行完每个任务后，任务释放CPU，调度器分派下一个的任务接管CPU 。

■**与状态机不同的是**：状态机及主程序就是任务调用者，是主动调用者，任务片段是受调用者。而任务调度器中，任务调度者是被调用者。

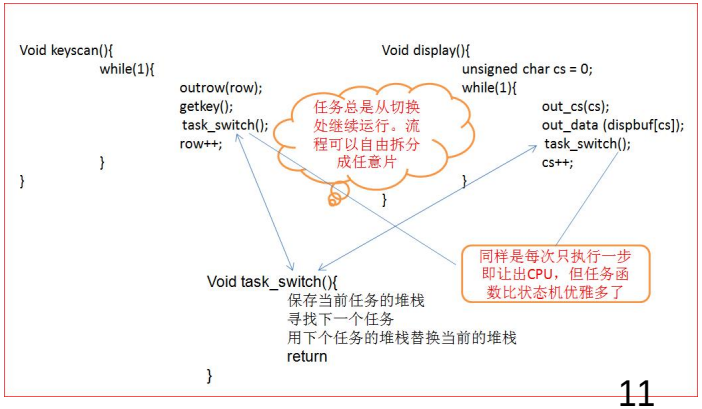
■这种调用关系决定了任务又可以像以前顺序流程那样写成直观的任务函数。

■任务执行完任务片段后到达切换点，此时调用任务调度函数将该任务流折断，调度器将该任务堆栈保存后将CPU交给下一个任务。当该任务下次重新获行运行机会时，只需取出之前保存的堆栈，即可从切换点处继续运行。



10

任务调度器示例



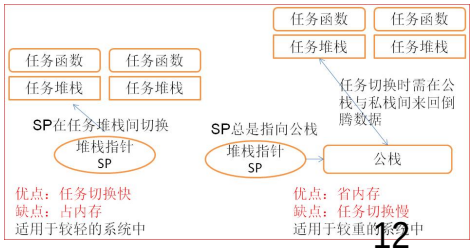
11

任务调度的核心：堆栈迁移

■有任务就有程序流，有程序流就需要堆栈，要折断任务流，就必须保存堆栈。每个任务都设有一个**私有任务堆栈**，用于保存任务被折断(任务切换)时的现场。

■堆栈是上下文切换时最重要的切换对象，这种对堆栈的切换叫作“**堆栈迁移**”。

■**堆栈迁移有两种方式**，一种方法是(左图)使用私栈作为堆栈，发生任务切换时，只需将栈指针即可到新任务的栈顶即可。另一种是(右图)使用**公栈**作为作堆栈，每切换一个任务，就将公栈的内容搬向私栈，并将新任务从私栈搬至公栈，然后修改栈指针指向新的栈底



12

栈指针切换VS堆栈搬移

■栈指针切换方式的优点是堆栈迁移速度快，修改SP即可完成迁移。

■堆栈搬移方式则较慢，每次还要计算栈深/栈顶位置，需搬若干字节。

■使用栈指针切换时，私栈需支撑调子函数、寄存器、局部变量、中断等。

■使用堆栈搬移时，私栈只支撑调子函数、寄存器、局部变量，与中断深度无关。

■栈指针切换的方式比堆栈搬移方式要多占用内存。

13

调度器的种类

■合作式调度器

■抢占式调度器

■混合式调度器

14

合作式调度器

合作式调度器
- 合作式调度器提供了一种单任务的系统结构

操作：
- 任务在特定的时刻被调度运行（以周期性或单次方式）
- 当任务需要运行时，被添加到等待队列
- 当 CPU 空闲时，运行等待任务中的下一个（如果有的话）
- 任务运行直到完成，然后由调度器来控制

实现：
- 这种调度器很简单，用少量代码即可实现
- 该调度器必须一次只为一个任务分配存储器
- 该调度器通常完全由高级语言（比如“C”）实现
- 该调度器不是一种独立的系统，它是开发人员的代码的一部分

性能：
- 设计阶段需要小心以快速响应外部事件

可靠性和安全性：
- 合作式调度简单、可预测、可靠并且安全

15

抢占式调度器

抢占式调度器
- 抢占式调度器提供了一种多任务的系统结构

操作：
- 任务在特定的时刻被调度运行（以周期性或单次方式）
- 当任务需要运行时，被添加到等待队列
- 等待的任务（如果有的话）运行一段固定的时间，如果没有完成，将被暂停并放回到等待队列。然后下一个等待任务将运行一段固定的时间，诸如此类等等

实现：
- 这种调度器相对复杂，因为必须实现诸如信号灯这样的特性，用来在“并行处理的”任务试图访问共用的资源时避免冲突
- 该调度器必须为抢占任务的所有中间状态分配存储器
- 该调度器通常将（至少是部分的）由汇编语言编写
- 该调度器通常作为一个独立的系统被创建

性能：
- 对外部事件的响应速度快

可靠性和安全性：
- 与合作式调度相比，通常认为其更不可预测，并且可靠性较低

16

混合式调度器

混合式调度器
- 混合式调度器提供了有限的多任务功能

操作：
- 支持多个合作式调度的任务
- 支持一个抢占式任务（可以中断合作式任务）

实现：
- 这种调度器很简单，用少量代码即可实现
- 该调度器必须一次为两个任务分配存储器
- 该调度器通常完全由高级语言（比如“C”）实现
- 该调度器不是一种独立的系统，它是开发人员的代码的一部分

性能：
- 对外部事件的响应速度快

可靠性和安全性：
- 只要小心设计，可以和单纯的合作式调度器一样可靠

17

1 嵌入式操作系统简介

实时多任务操作系统与分时多任务操作系统

■分时系统：软件的执行在时间上的要求并不严格，时间上的错误一般不会造成灾难性的后果。

■实时系统：虽然事件可能在无法预知的时刻到达，但是软件上必须在事件发生时能够在严格的时限内作出响应（系统响应时间），即使是在尖峰负荷下，也应如此，系统时间响应的超时就意味着致命的失败。另外，实时操作系统的重要特点是具有系统的可确定性，即系统能对运行情况的最好和最坏等的情况能做出精确的估计。

18

1 嵌入式操作系统简介(续)

- 基本概念 ——时钟节拍

时钟节拍是特定的周期性中断。这个中断可以看作是系统心脏的脉动。周期取决于不同应用，一般在10ms到200ms之间。时钟的节拍式中断使得内核可以将任务延时若干个整数时钟节拍，以及当任务等待事件发生时，提供等待超时的依据。时钟节拍率越快，系统的额外开销就越大。

1 嵌入式操作系统简介(续)

实时操作系统中的重要概念

■系统响应时间(System response time)

系统发出处理要求到系统给出应答信号的时间。

■任务切换时间(Context-switching time)

任务之间切换而使用的时间。

■中断延迟(Interrupt latency)

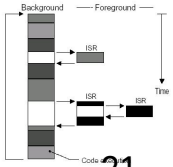
硬件接收到中断信号到操作系统作出响应，并转入中断服务程序的时间。

前后台系统 （Foreground/Background System）

■前后台系统或超循环系统(Super-Loops)：应用程序是一个无限的循环，循环中调用相应的函数完成相应的操作，这部分可以看成后台行为(background)。中断服务程序处理异步事件，这部分可以看成前台行为（foreground）。后台也可以叫做任务级。前台也叫中断级。适用于不复杂的小系统。

■时间相关性很强的关键操作靠中断服务来保证的。

■前后台系统在处理信息的及时性上，比实际可以做到的要差。这个指标称作任务级响应时间。最坏情况下的任务级响应时间取决于整个循环的执行时间。因为循环的执行时间不是常数，程序经过某一特定部分的准确时间也是不能确定的。进而，如果程序修改了，循环的时序也会受到影响。



代码的临界段

■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入。为确保临界段代码的执行，在进入临界段之前要关中断，而临界段代码执行完以后要立即开中断。

■数据撕裂（例如大于CPU位宽的数据、大于CPU位宽的定时器访问）

■临界段代码影响中断响应时间（因为要关中断）

资源

■任何为任务所占用的实体都可称为资源。

■资源可以是硬件设备，也可以是软件、存储空间。

■例如打印机、键盘、显示器、函数、变量、结构体或数组等都可以是资源。

多任务

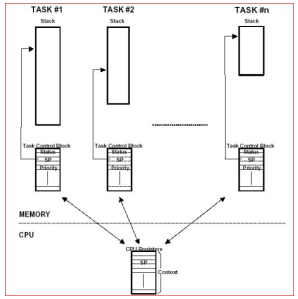
■多任务运行的实现实际上是靠CPU(中央处理单元)在许多任务之间转换、调度。CPU只有一个，轮番服务于一系列任务中的某一个。多任务运行很像前后台系统，但后台任务有多个。多任务运行使CPU的利用率得到最大的发挥，并使应用程序模块化。

■多任务化的最大特点是，开发人员可以将复杂的应用程序**层次化**。

■使用多任务，应用程序将更容易设计与维护。

任务

■ 任务，也称作一个线程，是一个简单的程序，该程序可以认为CPU完全只属于自己。实时应用程序的设计过程，包括如何把问题分割成多个任务，每个任务都是整个应用的某一部分，每个任务被赋予一定的优先级，有自己的CPU寄存器映射空间和栈空间。



25

任务状态

■5种状态

■休眠态：该任务驻留在内存中，但并不被多任务内核所调度。

■就绪态：任务准备运行，但优先级比运行任务低，暂时不能运行。

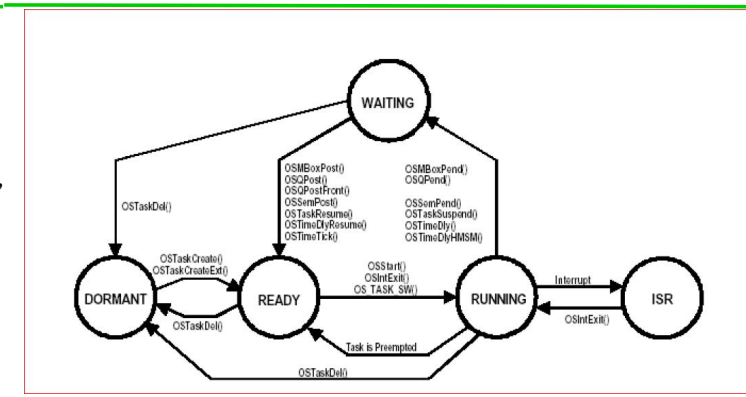
■运行态：任务是指该任务掌握了CPU的控制权，正在运行中。

■挂起态：也可以叫做等待事件态。指该任务在等待某一事件的发生，（例如等待某外设的I/O操作，等待某共享资源由暂不能使用变成能使用状态，等待定时脉冲的到来或等待超时信号的到来以结束目前的等待，等等）。

■被中断态：发生中断时，CPU提供相应的中断服务，正在运行的任务暂不能运行，就进入了被中断状态。

26

任务状态



27

任务切换(Context Switch or Task Switch)

■Context Switch 在有的书中翻译成上下文切换，实际含义是任务切换，或CPU寄存器内容切换。

■当多任务内核决定运行另外的任务时，它保存正在运行任务的当前状态（Context），即CPU寄存器中的全部内容。这些内容保存在任务的当前状况保存区（Task’s Context Storage area），也就是任务自己的栈区之中。入栈工作完成以后，就是把下一个将要运行的任务的当前状况从该任务的栈中重新装入CPU的寄存器，并开始下一个任务的运行。这个过程叫做任务切换。

■任务切换过程增加了应用程序的额外负荷。CPU的内部寄存器越多，额外负荷就越重。做任务切换所需要的时间取决于CPU有多少寄存器要入栈。实时内核的性能不应该以每秒钟能做多少次任务切换来评价。

28

内核（Kernel）

■多任务系统中，内核负责管理任务，或者说为任务分配CPU时间，并且负责任务之间的通讯。

■内核的基本服务是任务切换。之所以使用实时内核可以大大简化应用系统的设计，是因为实时内核允许将应用分成若干个任务，由实时内核来管理它们。

■内核本身也增加了应用程序的额外负荷，代码空间增加ROM的用量，内核本身的数据结构增加了RAM的用量。但更主要的是，每个任务要有自己的栈空间（占内存来是相当厉害的）。

■内核本身对CPU的占用时间一般在2-5%之间。

■单片机一般不能运行实时内核，因为单片机的RAM很有限。

■通过提供必不可缺少的系统服务，诸如信号量、邮箱、消息队列、延时等，实时内核使得CPU的利用更为有效。一旦用实时内核做过系统设计，将决不再想返回到前后台系统。

29

调度（Scheduler）

■调度（Scheduler、dispatcher）。这是内核的主要职责之一，就是要决定该轮到哪个任务运行了。多数实时内核是基于优先级调度法的。每个任务根据其重要程度的不同被赋予一定的优先级。

■基于优先级的调度法指，CPU总是让处在就绪态的优先级最高的任务先运行。

■何时让高优先级任务掌握CPU的使用权，有两种不同的情况，这要看用的是什么类型的内核，是不可剥夺型的还是可剥夺型内核。

30

不可剥夺型内核 （Non-Preemptive Kernel）

■不可剥夺型内核要求每个任务自我放弃CPU的所有权，也称作合作型多任务，各个任务彼此合作共享一个CPU。异步事件还是由中断服务来处理。中断服务以后控制权还是回到原来被中断的任务，直到该任务主动放弃CPU的使用权时，那个高优先级的任务才能获得CPU的使用权。

■不可剥夺型内核的一个优点是响应中断快。不可剥夺型内核允许使用不可重入函数。因为每个任务要运行到完成时才释放CPU的控制权。任务级响应时间取决于最长的任务执行时间。

■不可剥夺型内核的另一个优点是几乎不需要使用信号量保护共享数据。运行着的任务占有CPU，而不必担心被别的任务抢占。但这也不是绝对的。处理共享I/O设备时仍需要使用互斥型信号量。例如，在打印机的使用上，仍需要满足互斥条件。

■不可剥夺型内核的最大缺陷在于其响应时间。高优先级的任务已经进入就绪态，但还不能运行，要等，也许要等很长时间，直到当前运行着的任务释放CPU。与前后系统一样， 不可剥夺型内核的任务级响应时间是不确定的。

■商业软件几乎没有不可剥夺型内核。

31

可剥夺型内核

■可剥夺型内核总是让就绪态的高优先级的任务先运行，中断服务程序可以抢占CPU，到中断服务完成时，内核让优先级最高的就绪任务运行（不一定是被中断了的任务）。任务级系统响应时间得到了最优化且是可知的。使用可剥夺型内核使得任务级响应时间得以最优化。

■ 使用可剥夺型内核时，应用程序不应直接使用不可重入型函数。**调用不可重入型函数时，要满足互斥条件**，这一点可以用互斥型信号量来实现。如果调用不可重入型函数时，低优先级的任务CPU的使用权被高优先级任务剥夺，不可重入型函数中的数据有可能被破坏。

32

可重入性（Reentrancy）

■可重入型函数可以被一个以上的任务调用，而不必担心数据的破坏。可重入型函数任何时候都可以被中断，一段时间以后又可以运行，而相应数据不会丢失。可重入型函数或者只使用局部变量，即变量保存在CPU寄存器中或堆栈中。如果使用全局变量，则要对全局变量予以保护。

■应用程序中的不可重入函数引起的错误很可能在测试时发现不了，直到产品到了现场问题才出现。处理多任务的新手，使用不可重入型函数时，千万要当心。

33

时间片轮番调度法

■给任务分配优先级相当困难。

■当两个或两个以上任务有同样优先级，内核允许一个任务运行事先确定的一段时间，叫做时间额度（quantum），然后切换给另一个任务。也叫做时间片调度。内核在满足以下条件时，把CPU控制权交给下一个任务就绪态的任务：

- 当前任务已无事可做；
- 当前任务在时间片还没结束时已经完成；
- 当前任务时间片时间到。

34

任务优先级

■每个任务都有其优先级。任务越重要，赋予的优先级应越高。

■静态优先级：应用程序执行过程中诸任务优先级不变，则称之为静态优先级。在静态优先级系统中，诸任务以及它们的时间约束在程序编译时是已知的。

■动态优先级 ：应用程序执行过程中，任务的优先级是可变的，则称之为动态优先级。实时内核应当避免出现优先级反转问题。

■优先级反转 ：优先级反转问题是实时系统中出现最多的问题。为防止发生优先级反转,内核能自动变换任务的优先级,这叫做优先级继承(Priority inheritance) 。

35

任务优先级分配

■给任务定优先级是复杂的事，因为实时系统相当复杂。许多系统中，并非所有的任务都至关重要。不重要的任务自然优先级可以低一些。实时系统大多综合了软实时和硬实时这两种需求。软实时系统只是要求任务执行得尽量快，并不要求在某一特定时间内完成。硬实时系统中，任务不但要执行无误，还要准时完成。

■单调执行率调度法RMS(Rate Monotonic Scheduling),用于分配任务优先级。这种方法基于哪个任务执行的次数最频繁,执行最频繁的任务优先级最高。

36

互斥条件

■实现任务间通讯最简便到办法是使用共享数据。特别是当所有到任务都在一个单一地址空间下，能使用全程变量、指针、缓冲区、链表、循环缓冲区等，使用共享数据结构通讯就更为容易。虽然共享数据区法简化了任务间的信息交换，但是必须保证每个任务在处理共享数据时的排它性，以避免竞争和数据的破坏。

■与共享资源打交道时，使之满足互斥条件最一般的方法有：

1、关中断 ，关中断时间不能长，影响系统的中断响应时间

2、禁止做任务切换 ，给任务切换上锁和开锁

3、利用信号量

死锁（Deadlock （or Deadly Embrace））

■死锁，指两个任务无限期地互相等待对方控制着的资源。设任务T1正独享资源R1，任务T2在独享资源R2，而此时T1又要独享R2，T2也要独享R1，于是哪个任务都没法继续执行了，发生了死锁。

■最简单的防止发生死锁的方法是让每个任务都：

1、先得到全部需要的资源再做下一步的工作；

2、用同样的顺序去申请多个资源；

3、释放资源时使用相反的顺序；

4、内核大多允许在申请信号量时定义等待超时，以化解死锁。

同步

■信号量

■互斥量

■事件标志(Event Flags) ，与多个事件同步

任务间的通讯(Intertask Communication)

■任务间信息的传递有两个途径：通过全程变量或发消息给另一个任务。

■消息邮箱(Message Mail boxes)

■消息队列(Message Queue)

中断

■**中断响应时间** = 中断延迟 + 保存CPU内部寄存器的时间

■**中断处理时间**

■**中断恢复时间**(Interrupt Recovery)

■中断恢复时间 = 判定是否有优先级更高的任务进入了就绪态的时间 + 恢复那个优先级更高任务的CPU内部寄存器的时间 + 执行中断返回指令的时间。

■中断返回的处理：

前后台系统：程序回到后台程序；

不可剥夺型内核：程序回到被中断了的任务；

可剥夺型内核：就绪态的最高优先级任务运行；

中断延迟

■实时内核的一个重要性能指标是中断最长要关多长时间。所有实时系统在进入临界区代码段之前都要关中断，执行完临界代码之后再开中断。关中断的时间越长，中断延迟就越长。

■中断延迟 = 关中断的最长时间 + 开始执行中断服务子程序的第一条指令的时间

中断响应

■中断响应定义为从中断发生到开始执行用户的中断服务子程序代码来处理这个中断的时间。中断响应时间包括开始处理这个中断前的全部开销。

■中断响应是系统在最坏情况下的响应中断的时间。

■前后台系统中断响应时间 = 中断延迟 + 保存CPU寄存器的时间；

■不可剥夺型内核中断响应时间 = 中断延迟 + 保存CPU寄存器的时间；

■可剥夺型内核中断响应 = 中断延迟 + 保存CPU寄存器的时间 + 内核的进入中断服务函数的执行时间 ；

中断恢复时间(Interrupt Recovery)

■前后台系统中断恢复时间= 恢复CPU寄存器值时间 + 执行中断返回时间；

■不可剥夺型内核中断恢复时间= 恢复CPU寄存器值时间 + 执行中断返回时间；

■可剥夺型内核中断恢复时间= 判定是否有优先级更高的任务就绪态时间 + 恢复优先级更高任务CPU寄存器时间 + 执行中断返回时间。

中断处理时间

■中断服务的处理时间应该尽可能的短，并没有绝对的限制。

■在大多数情况下，中断服务子程序应识别中断来源，从中断源设备取得数据或状态，并通知真正做该事件处理的那个任务。

■通知（通过信号量、邮箱或消息队列）是需要时间的。

■如果事件处理需花的时间短于给一个任务发通知的时间，就应该考虑在中断服务子程序中做事件处理并在中断服务子程序中开中断，以允许优先级更高的中断打入并优先得到服务。

非屏蔽中断(NMI)

■有时内核引起的延时变得不可忍受。在这种情况下可以使用非屏蔽中断，绝大多数微处理器有非屏蔽中断功能。通常非屏蔽中断留做紧急处理用，如断电时保存重要的信息。然而，如果应用程序没有这方面的要求，非屏蔽中断可用于时间要求最苛刻的中断服务。

■非屏蔽中断的中断服务不能使用内核提供的服务，因为非屏蔽中断是关不掉的，故不能在非屏蔽中断处理中处理临界区代码。然而向非屏蔽中断传送参数或从非屏蔽中断获取参数还是可以进行的。参数的传递必须使用全程变量，全程变量的位数必须是一次读或写能完成的，即不应该是两个分离的字节，要两次读或写才能完成。

时钟节拍(Clock Tick)

■时钟节拍是特定的周期性中断。这个中断可以看作是系统心脏的脉动。中断之间的时间间隔取决于不同的应用，一般在10mS到200mS之间。时钟的节拍式中断使得内核可以将任务延时若干个整数时钟节拍，以及当任务等待事件发生时，提供等待超时的依据。时钟节拍率越快，系统的额外开销就越大。

■相对时间（Delay的抖动）

■绝对时间 （定时任务）

对存储器的需求

■前后台系统：对存储器容量的需求取决于应用程序代码。

■多任务内核：总代码量 = 应用程序代码 + 内核代码（1K-100K）

■每个任务需要单独的栈空间(RAM)。应用设计人员分配任务栈。

■有些内核任务栈大小可分别定义；有些内核要求任务栈相同。

■若内核不支持单独的中断栈：RAM总需求 = 应用程序RAM需求 +（任务栈需求 + 最大中断嵌套栈需求）* 任务数

■若内核支持中断栈，RAM总需求 = 应用程序的RAM需求 + 内核数据区的RAM需求 + 各任务栈需求之总和 + 最多中断嵌套之栈

■特别注意以下几点：

定义函数和中断服务的局部变量；函数(即子程序)的嵌套 ；

中断嵌套 ；库函数需要的栈空间 ；

内存管理

■在ANSI C中可以用malloc()和free()两个函数动态地分配内存和释放内存。但是，在嵌入式实时操作系统中，多次这样做会把原来很大的一块连续内存区域，逐渐地分割成许多非常小而且彼此又不相邻的内存区域，也就是内存碎片。由于这些碎片的大量存在，使得程序到后来连非常小的内存也分配不到。

■由于算法的原因，malloc()和free()函数执行时间是不确定的。

■常用方法：固定大小分区、可变长度分区。

使用实时内核的优缺点

■使得实时应用程序的设计和扩展变得容易，不需要大的改动就可以增加新的功能。通过将应用程序分割成若干独立的任务，RTOS使得应用程序的设计过程大为减化。

■使用实时内核额外的需求是：内核的价格、额外的ROM/RAM开销、2到4百分点的CPU额外负荷、 使用硬件成本。

■RTOS价格可能包括：

RTOS的基本价格；

RTOS的开发座席费；

RTOS版权使用费；

RTOS软件年维护费。

什么时候该使用OS？

1. 当多个任务都很耗时，而这些任务可以并发调用，为了提高任务之间的调度并发性，应该考虑使用OS；
2. 当业务逻辑过于复杂，而通过自设计的调度器来调度时，使得设计不能简单，相较于OS往往趋于复杂，不易维护，为了使系统设计更加简单可靠，可以考虑使用OS；
3. 当系统资源充足，开发团队熟悉OS，使用OS更加节省开发时间；
4. 对于低功耗设备，使用前后台系统比使用OS方法更易控制系统的功耗，因为 OS系统方式，为了使任务调度及时，往往定时器中断调度更频繁，CPU的唤醒频率更频繁。