第 3 章

嵌入式软件体系结构

软件体系结构是用来对程序逻辑功能结构进行规范化划分的一种方法，通过划分不同功能模块或构件，整个软件的结构更为清晰，有利于程序员的开发和复用。采用规范结构保障软件开发质量已经应用于嵌入式软件的开发过程，形成多种多样的体系结构模式，如客户端/服务器结构等。

此外，对于任何一个给定系统，特别是嵌入式系统，在决定最适合该系统的软件体系结构的众多因素中，最重要的是对系统响应时间进行控制的程度，不仅取决于对绝对响应时间的要求，而且也取决于采用的微处理器的速度和其他处理需求。对于一个功能有限的系统，如果该系统对响应时间的要求很低，那么该系统可以使用一种很简单的软件结构来完成。而对于一个能对许多任务作出快速响应，并且对任务截止时间和优先级有不同控制处理要求的系统，就需要较复杂的软件结构。

当前，软件体系结构设计已成为嵌入式软件开发过程中最关键的一步，也是设计嵌入式软件时的第一步。软件成本估算模型 COCOMO 之父 Barry Boehm 也明确指出“在没有设计出体系结构及其规则时，整个项目不能继续下去，而软件体系结构应该看作软件开发中可交付的中间产品”。

3.1 软件体系结构的概念

软件体系结构是具有一定形式的结构化元素，即**构件**的集合，包括**处理构件**、**数据构件**和**连接构件**。构件就是具备一定独立功能的程序。处理构件负责对数据进行加工，数据构件是被加工的信息，连接构件把体系结构的不同部分组合连接起来。这一定义注重区分处理构件、数据构件和连接构件，这种逻辑功能上的划分在实际应用中得到广泛支持。软件体系结构在 IEEE 中的定义为“一个系统的基础组织，包含各个构件、构件互相之间与环境的关系，还有指导其设计和演化的原则”。

3.2 软件体系结构的作用

从软件开发过程来看，所有被实现的软件系统都有一个软件结构。软件体系结构在软件开发中发挥着重要作用。在软件开发中，**软件工程**对软件体系结构设计支持的需求越来

越迫切。第一，通过认识和理解体系结构可以使系统的高层次关系得到全面表达和深刻理解；第二，获得正确的体系结构常常是软件系统设计成功的关键，否则可能导致灾难性的结果；第三，全面深入地理解软件体系结构，才可以使设计者在复杂的问题面前作出正确的抉择；第四，系统体系结构对于复杂系统的高层次性能的分析是至关重要的。如果原始的设计结构能够得到清楚和明确的表达，特别是高层次的表达，可大大减少软件维护相关的开销。

综上所述，软件体系结构是整个软件设计成功的基础和关键所在，它的作用在**软件生命周期**的各个阶段表现如下。

（1）在**项目规划**阶段，粗略的体系结构是进行**项目可行性**、**工程复杂性**、**工程进展**、**投资规模**、**风险预测**等的重要依据。

（2）在**项目需求分析**阶段，需要从需求出发建立更深入的体系结构描述。这时的体系结构是开发者和用户之间进行需求交互的表达形式，也是交互所产生的结果。通过它，可以准确地表达用户的需求，以及设计对应需求的解决方法，并考查总结系统的各项性能。

（3）在**项目设计**阶段，需要从实现的角度对体系结构进行更深入的分解和描述。

作为软件的一个分支，嵌入式软件的体系结构的作用与通用软件的体系结构的作用是一致的。但由于其运行环境的特点及对响应时间的不同要求，产生了许多经典的嵌入式软件体系结构，著名的有轮转结构、前后台结构和实时操作系统结构。下面对这 3 种结构进行详细介绍。

第 7 集
微课视频

3.3 轮转结构

轮转（Round-Robin）**结构**是一种最简单的**嵌入式实时软件体系结构**模型，它没有中断，无须考虑**延迟时间**，这些特点使得该结构成为所有结构中最具吸引力的一种，因此对于能用该结构成功解决问题的系统来说，这种结构是首选。

代码 3-1 是轮转结构的伪代码原型，在该结构中不存在中断，主循环只是简单地依次检查每个 I/O 设备，并且为每个需要服务的设备提供服务。

代码 3-1　轮转结构伪代码

```
int main()
{
    while(true)
    {
        if (I/O 设备 A 需要服务)
          处理 I/O 设备 A 的数据;
        if (I/O 设备 B 需要服务)
          处理 I/O 设备 B 的数据;
        …
        if (I/O 设备 Z 需要服务)
```

```
            处理 I/O 设备 Z 的数据;
        }
        return 0;
    }
```

每个设备访问完成之后，才将 CPU 移交给下一个设备使用。对于某个设备，当它提出执行请求后，必须等到它被 CPU 接管后才能执行。

3.3.1　运行方式

按照程序结构说明，轮转结构系统具有以下工作特点：系统完成一个轮转的时间取决于轮转环中需要执行的服务个数(即满足执行条件的外设个数)。此外，轮转的次序是静态固定的，在运行时是不能进行动态调整的。轮转结构的运行方式如图 3-1 所示，这里的任务对应于代码 3-1 中的 if 设备服务块。

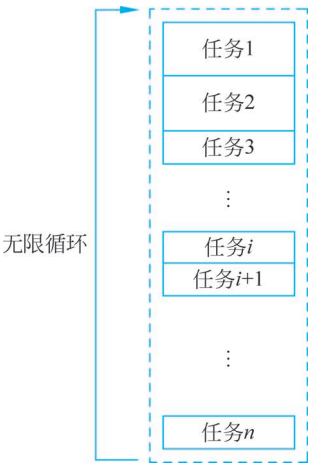

图 3-1　轮转结构的运行方式

轮转结构简单，可以在多路采样系统、实时监控系统等嵌入式应用中广泛使用，是最常用的软件结构之一。但它也存在无法忽略的弱点，主要如下。

(1) 如果一个设备需要比微处理器在最坏情况下完成一个循环的时间更短的响应时间，那么这个设备将无法工作。例如，在代码 3-1 中，如果设备 Z 需要在 6ms 之内获得服务，但为设备 A 和设备 B 服务的代码各需 4ms 执行，那么处理器就不能及时地响应设备 Z，可以通过将设备 Z 换到设备 A 后运行解决该问题。

(2) 即使设备需要的**响应时间**不是绝对的截止时间，当系统中有长时间服务设备时，设备的服务体验会很差。例如，设备 B 需要 3s 服务，那么设备 Z 的服务体验极差。

(3) 这种结构极其脆弱。即使能够提高系统的性能，如提高处理器速度，从而获得较小的处理循环时间，满足所有设备的服务需求，但一旦增加一个额外的设备，或者提出一个新的中断请求，就可能让一切破坏。

3.3.2　典型系统

轮转结构虽然简单，但是在一些对时间要求不高的场景下使用得很多。如图 3-2 所示的显示器调节器(类似空调遥控器)，可以通过不同按键调节显示器亮度、显示模式及功能选择等。

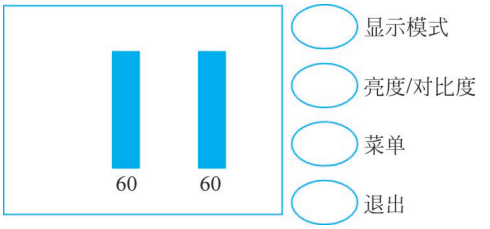

图 3-2　显示器调节器

代码 3-2 为显示器调节器的伪代码。

代码 3-2　显示器调节器的伪代码

```
void vDigitalController()
{
    enum choice{Mode,Brightness,Menu,Exit} eControllerItem;
    while(true)
    {
        eControllerItem = someChoice;
        switch (eControllerItem)
        {
            case Mode:
                显示器显示模式选项;
                使用对应按键选择一个模式;
                根据所选模式改变显示器的显示模式;
                break;
            case Brightness:
                显示器显示亮度和对比度控制选项;
                使用对应按键调整亮度和对比度值;
                根据所选亮度和对比度值改变显示器显示亮度;
                break;
            …
        }
    }
}
```

在每次循环中，代码检查按键的位置，然后执行对应的分支程序完成显示器显示设定读取，根据按键选择改变设定值，并完成显示器显示调整，即使系统使用的是一个非常低速的微处理器，也能在 1s 内完成多次该循环。

在该系统中，轮转结构可以良好工作，因为整个系统中只有两个 I/O 设备，并且没有特别耗时的处理任务，也没有紧迫的响应需求。微处理器可以随时读取按钮选择，并快速调整显示器显示。当用户正在调节显示器设定时，他不会关注到微处理器处理一次循环需要的几分之一秒，因此轮转结构足以胜任该需求。

3.4　前后台结构

前后台(Foreground-Background)**结构**其实是一种**带中断的轮转结构**，也称为**中断驱动结构**，其显著特点是运行的任务有**前台**和**后台**之分。后台是一个无限循环，循环中调用相应的函数完成相应的操作，这和轮转结构是一致的；在前台，中断服务程序(Interrupt Service Routines，ISR)用于处理系统的异步事件。因此，前台也称为中断级，而后台称为任务级。

3.4.1　运行方式

前后台结构运行方式如图 3-3 所示。

在前后台结构中，前台中断的产生与后台任务的运行是并行的：中断由外部事件随机

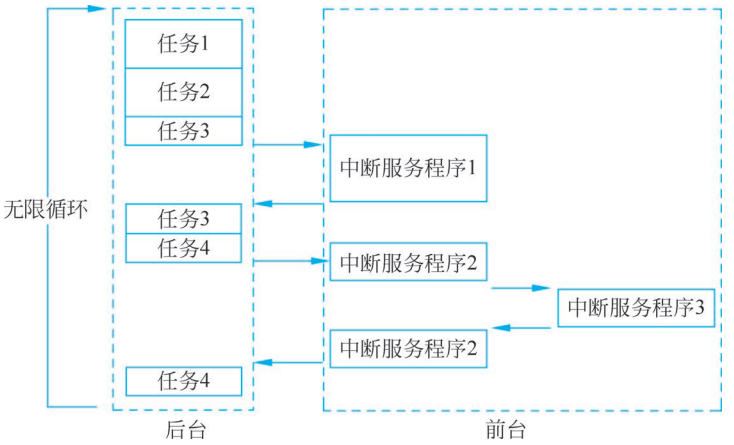

图 3-3　前后台结构运行方式

产生，而且绝大部分是不可预知的。此外，系统还必须解决前台与后台资源共享的问题。前后台结构伪代码如代码 3-3 所示。

代码 3-3　前后台结构伪代码

```
bool iDeviceA = false;
bool iDeviceB = false;
…
bool iDeviceZ = false;
void interruptServiceA()                    //中断服务程序
{
    iDeviceA = true;
}
void interruptServiceB()
{
    iDeviceB = true;
}
…
void interruptServiceZ()
{
    iDeviceZ = true;
}
int main()
{
    while(true)
    {
        if (iDeviceA)                       //任务 1
          处理 I/O 设备 A 输入或输出的数据;
        if (iDeviceB)
          处理 I/O 设备 B 输入或输出的数据;
        …
        if (iDeviceZ)
          处理 I/O 设备 Z 输入或输出的数据;
    }
```

```
    return 0;
}
```

与轮转结构相比，前后台结构可以对优先级进行更多的控制。中断服务程序可以获得很快的响应，因为硬件的中断信号可以使微处理器停止正在 main 函数中执行的任何操作，转而去执行中断服务程序。这是因为中断服务程序中的操作比主程序中的任务代码具备更高的优先级。实际上，不同中断具有不同的优先级，那么控制不同中断服务程序的优先级也是可行的。如图 3-3 所示，中断服务程序 3 的优先级高于中断服务程序 2，因此中断服务程序 2 被打断，中断服务程序 3 被优先执行。

图 3-4 展示了轮转结构和前后台结构对优先级进行控制的差异比较。轮转结构中没有优先级控制，因此对紧急设备的处理要求无法快速响应；而前后台结构由于引入中断，从而保障快速响应紧急任务。这种差异正是前后台结构相对于轮转结构的最大优势。前后台结构的缺点也在代码 3-3 中体现得很清晰，iDeviceA 等变量在中断服务程序和后台任务中进行了共享，从而带来潜在的威胁。

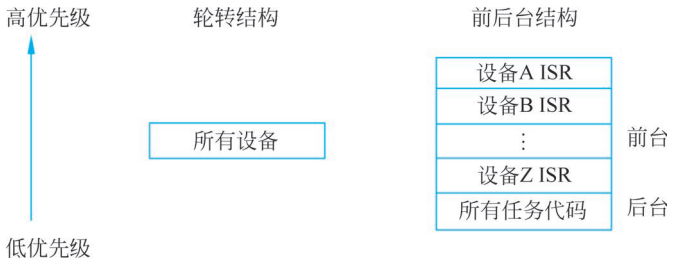

图 3-4　轮转结构和前后台结构的对比

3.4.2　系统性能

前后台结构的主要缺点是所有任务代码均以相同的**优先级**运行，假设代码 3-3 中的处理设备 A、B、C 的任务代码各需 300ms，如果在微处理器开始执行循环的起始部分，设备 A、B、C 都发出中断信号，那么设备 C 的任务代码可能需要等待 600ms 才能够开始执行。

一个可行的解决办法是将设备 C 的任务代码放到它的中断服务程序中。在前后台结构中，把设备服务代码放到中断服务程序中，从而使它可以以更高优先级执行是一个较合适的办法。但也会带来负面效果，改进的办法会使得设备 C 的中断服务程序执行的时间增加 300ms，从而导致低优先级设备 D、E 和 F 的中断服务程序的响应时间增加 300ms，这同样难以接受。

中断服务程序即使生成了特定的数据，后台程序也必须运行到对应的处理程序时才能进行处理，通常把后台完成数据处理的过程所需花费的时间称为**任务级（后台）响应时间**，而把中断发生到后台完成处理的时间称为**中断响应时间**。从代码 3-3 可以看出，最长的任务级响应时间取决于后台循环的运行时间。例如，循环刚通过了该设备的任务代码后发生了该设备的中断，并且其他设备也都发出了中断且需要进行服务，那么该设备的任务级响应时

间就十分漫长。

对该问题的解决思路是在主循环中加大某一任务所占的比重，在图 3-3 中，可以让后台任务处理顺序为：任务 1、任务 2、任务 3、任务 4、任务 3 等，增大任务 3 的个数，可以改善任务 3 的任务级响应时间。

除了较为复杂的实时应用之外，前后台系统能够满足几乎所有应用要求。绝大多数单用户计算机系统都采用前后台结构，后台是一个空循环，应用程序在该工作模式下通过中断方式得到 CPU 的服务。当前台没有中断请求时，后台按照轮转方式工作。当有新任务到达时，新任务能通过中断形式向系统提出请求，从而得到及时的响应，这样不会因系统响应不及时造成额外的损失。

由于中断发生时需要付出额外的开销（现场保护和恢复），因此在有较高吞吐量要求的场合，中断的事务处理是不合适的。此时，往往采用特殊的硬件（如 DM）进行处理，或采用轮转方式。

3.4.3　典型系统

很多低成本、大批量的微控制器应用，如家用微波炉等，采用的都是前后台结构。微波炉控制面板上包含 4 个控键按钮，依次是分、秒、开始和取消；还有一个显示屏，显示当前时间或微波时长，如图 3-5 所示。

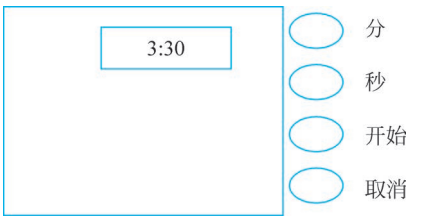

图 3-5　微波炉控制面板

微波炉应用伪代码如代码 3-4 所示。

代码 3-4　微波炉应用伪代码

```
bool isOn = false;
bool timeCounter = true;
int timer = 0;
int timeofWave = 0;
void interruptMinute()                    //中断服务程序
{
    timeofWave += 60;
    timeCounter = false;
}
void interruptSecond()
{
    timeofWave += 1;
    timeCounter = false;
}
void interruptStart()
{
    isOn = true;
    timeCounter = false;
}
void interruptCancel()
```

```
{
    isOn = false;
    timeCounter = true;
}
void interruptTimer()
{
    timeCounter = true;
    timer += 1;
}
int main()
{
    while(true)
    {
        if (timeCounter)                          //计时器
            print("%d",timer);                    //屏幕显示目前时间
        if (timeofWave)
            print("%d", timeofWave);              //屏幕显示 timeofWave
        if (timeofWave&&isOn)
        {
            屏蔽计时器和分秒按键中断
            while (timeofWave > 0)
            {
                print("%d", timeofWave);
                wavebegins();                     //开始微波
                timeofWave = timeofWave - 1;
            }
            开放计时器和分秒按键中断
            timeCounter = true;
            isOn = false;
            timeofWave = 0;
        }
    }
    return 0;
}
```

从代码 3-4 可以看出，该微波炉应用后台程序平时就是一个计时显示任务，没有按键按下时就显示当前时间；有按键按下时就显示微波时间。当用户设置微波时间并按下“开始”按键后就开始加热食物，加热过程中不响应计时器中断和时间按键按下中断，但可以响应“取消”按键按下中断。

3.5 实时操作系统结构

实时操作系统结构是用于管理微处理器资源和硬件资源的软件结构。设计实时操作系统时，可以把系统功能划分成多个任务，每个任务仅负责某一方面的功能。系统主要完成**任务切换**、**任务调度**和**中断管理**等基本工作，可以使用户把精力集中于应用系统功能的实现上，从而能高效地开发上层应用软件。

3.5.1　运行方式

每个程序执行时称为**任务**。因此，对于实时嵌入式操作系统，任务是一个能够拥有CPU、内存和I/O等硬件资源的基本单位，也是操作系统调度的基本单位。它在概念上基本等同于通用操作系统中的**进程**。系统中的每个任务通常都是一个死循环，CPU在任意时刻只能执行一个任务，但每个任务都以为自己在独占整个CPU。

实时操作系统结构伪代码如代码3-5所示。

代码3-5　实时操作系统结构伪代码

```
bool iDeviceA = false;
bool iDeviceB = false;
…
bool iDeviceZ = false;
void interruptServiceA()                    //中断服务程序
{
    iDeviceA = true;
}
void interruptServiceB()
{
    iDeviceB = true;
}
…
void task1()
{
    while(true)
    {
        if (iDeviceA)
            处理设备A的数据;
    }
}
void task2()
{
    while(true)
    {
        if (iDeviceB)
            处理设备B的数据;
    }
}
…
```

代码3-5中演示的实时操作系统运行方式如图3-6所示。

图3-6中的低优先级任务对应代码3-5中的task2，高优先级任务对应task1，等待事件就是iDeviceA和iDeviceB。运行过程如下。

(1) 低优先级任务task2正在运行。

(2) 此时产生中断，CPU转去中断源对应的中断服务程序。

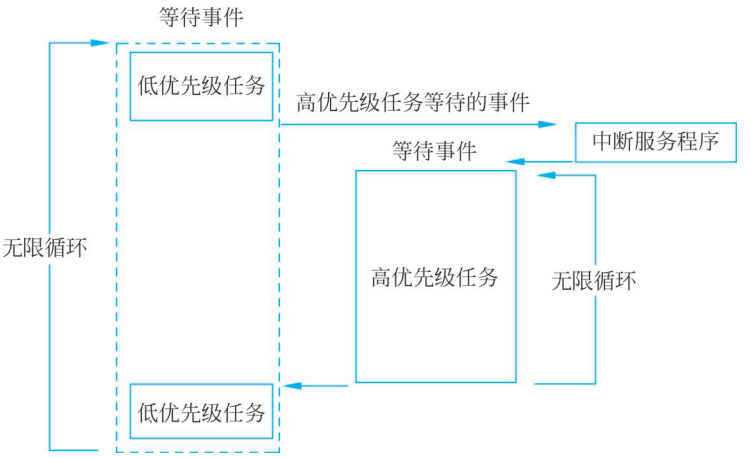

图 3-6　实时操作系统运行方式

(3) 中断服务程序非常简单,通常只是发出一个信号,如设置 iDeviceA＝true; 而由另一个高优先级的任务响应这个信号,并完成中断请求中的大部分工作。

(4) 当中断服务程序运行完毕,实时操作系统发现中断服务程序中发出的信号使某个高优先级任务具备了运行的条件,因此切换到高优先级任务 task1 运行。

(5) task1 开始运行,对中断事件进行进一步处理。

(6) task1 运行完毕,返回到低优先级任务 task2 起始点继续运行。

3.5.2　系统性能

在实时操作系统结构中,中断服务程序可以处理大多数的紧急情况,然后中断服务程序会发出“需要任务代码处理剩下的数据”的请求,该结构与前后台结构的区别如下。

(1) 中断服务程序和任务代码之间互相传递的信号是由实时操作系统处理的,并不需要前后台系统中的共享变量实现。

(2) 在任务代码中并没有使用循环决定下一步的操作。实时操作系统的系统代码可以决定什么任务代码可以运行。系统知道各种任务代码的优先级,由它决定优先运行哪个任务。

(3) 实时操作系统可以将当前正在执行的任务挂起,转去运行另外一个任务。

在实时操作系统的管理下,控制 CPU 在多个顺序执行的任务之间切换,实现对 CPU 资源利用的最大化,这个过程称为**多任务管理**。前两点有利于简化用户程序开发,后一点对应用性能的影响巨大: 使用实时操作系统结构的系统不仅可以控制任务代码的响应时间,还可以控制中断响应时间(发出中断到中断数据处理完毕)。

在代码 3-5 中,如果 task1 是最高优先级的任务,那么当中断服务程序将信号 iDeviceA 设置为 true 时,实时操作系统会立刻运行 task1。如果此时 task2 正在运行,实时操作系统会将 task2 挂起并且立刻运行 task1。因此,优先级最高的任务几乎没有延迟,中断响应时

间也几乎为对应的任务处理时间。

可以从图 3-7 看出各结构任务优先级关系。

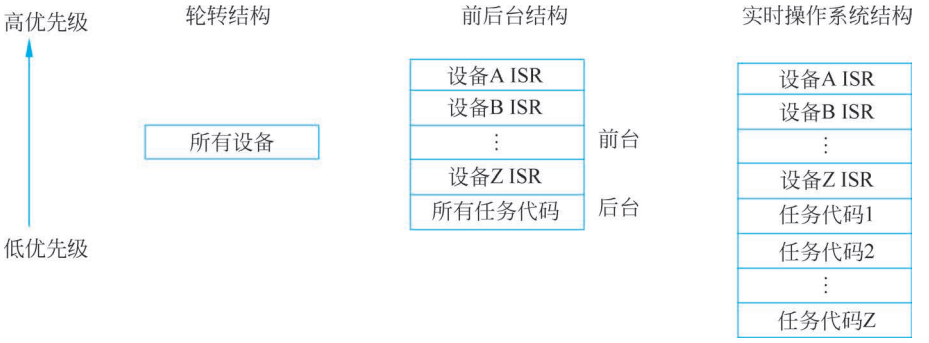

图 3-7　各结构任务优先级关系

使用实时操作系统提供的任务管理功能本身就会增加系统的开销，具体增加量取决于功能函数被调用的频率。因此，与前后台结构系统中前台和后台任务切换相比，实时多任务系统的任务切换需要更大的开销。一般来说，任务切换通常占用 2%～4%的 CPU 时间，时间大都限定在 2～10ms。而且由于操作系统是一套软件代码，因此需要额外的 ROM 和随机存储器(Random Access Memory，RAM)空间。

实时操作系统中，中断响应时间可以分为 3 种情况。

(1) 在中断服务程序中要进行任务切换，不过是切换到内核程序，由内核程序对中断请求事务进行处理。

(2) 在中断服务程序中不进行任务切换，中断请求事务完全由中断服务程序完成。

(3) 在中断服务程序中只设置标识、激活相应任务，所有中断事务处理由对应任务完成。

显然，第 2 种情况下的响应时间最短，但第 3 种情况的灵活度最高。因此，在对系统性能进行描述时，要区别对待。

3.5.3　典型系统

手机和平板电脑等都是实时操作系统的典型应用。以手机为例，其硬件环境多为 64 位微处理器，如 ARM，其上配置了专门的嵌入式操作系统，如 Android、HarmonyOS 等。众多的任务建立在操作系统的基础上，实现手机的功能，包括通信、游戏、短信、多媒体处理、办公、日程等。

显然，这些任务具有不同的优先级，系统也允许多任务抢占调度。图 3-8 给出了基于 HarmonyOS 的手机多任务功能结构框架。由图 3-8 可知，HarmonyOS 是一个实时操作系统，支持多个任务同时运行；高优先级任务能打断低优先级任务，如天气应用能被电话应用打断，保障高优先级任务快速得到响应；当没有任务运行时，后台处于空闲状态。

图 3-8　基于 HarmonyOS 的手机多任务功能结构框架