关于本书

这是一本工程师与工程师之间交流经验的书，本书的作者是一位嵌入式系统工程师，毕业以来，一直从事嵌入式系统开发，从未中断。本书中，作者将和广大读者共享其多年积累的经验。

当你捧起这本书，笔者为此感到荣幸，为有机会和你交流而兴奋，在工程技术领域，交流是提高的重要途径，希望能从你获得有益的建议。本书有经验的总结，也提出了一些自己的观点，就像上帝不会造出两张一摸一样的脸一样，也不可能有两个观点一摸一样的人，笔者不强求你认同本书的所有观点，但至少希望能拓展你的视野。如果你对本书涉及的问题有不同的看法，请务必给我写信，email:lstpr@21cn.com。

设计可靠的产品和可移植的技术是本书的两大主题。

怎样设计可靠的产品？笔者多年的职业生涯中，涉及研制的都是电力和电信行业相关的工业类产品，这种产品的特点是可靠性要求高，使用时效长。它不像消费类民用产品，更新淘汰得很快，在使用中，即使出现故障，也不会造成严重的损失，顶多是看不成电视，听不到音响而已，维修时也可以随时关闭电源、开机拆解，而且维修也是以更换部件为主，而不会深入分析故障原因。而许多工业监控类产品需要长期不间断运行，出现故障即有可能造成严重损失，设计中要防止局部故障导致严重损失。这些产品的服务期很长，动辄 5 年、8 年，甚至超过 10 年也很常见。在这么长的运行期中，生产厂家需要分析每一起故障，分析可能存在的隐患，不但要改进新产品，还要升级正在服役的产品，防止出现新的故障。更有甚者，在维修过程中可能还不能关闭电源，不能退出运行。从而使笔者能够长时间跟踪产品，充分暴露、发现隐含的问题，从中总结出宝贵的经验，笔者最近一年的主要工作职责之一，就是产品失效分析，分析产品的每一起售前或者售后故障，找出改进点。

设计可移植的技术，是本书的又一主题。由于工业现场条件复杂多变，使得工业类嵌入式产品的衍生型号很多，如何使产品能够适应不同的环境，或者使技术在不同产品间具有最大的可移植性，是一门艺术。有人说，“我的软件是用 C 语言写的，是可移植的。”这句话看起来很有道理，实际上谬之大矣！等于是说：“我用钢笔写字，我的书法具有可移植性。”灵活的系统结构，一致的模块接口，清晰的软硬件脉络，才是可移植的关键。也有人认为移植只是换个平台这么简单，其实能满足不同平台只是可移植软件的最低要求，可移植的软件最重要的是在当用户的需求发生变化时，能够迅速调整，重新组合，满足用户的需求，即使软件有较大的规模，船大也要好掉头才行。

要完成可靠性和可移植性目标，不是某一个模块性能优越能达到的，最优化设计的模块，不一定能组装出一个整体性能优越的产品；而整体性能优越的产品，其局部模块可能很粗糙，这是一个系统工程的。本书以笔者自己开发的 djyos 操作系统为范例讲述产品设计的系统性思想，djyos 是一个软件，但本书并不是一本软件教科书，djyos 专门为软硬件综合进行系统架构优化提供了支持。

计算机技术发展到现在，已经有很多技术积累，开发一个复杂的嵌入式产品，要懂得“拿来主义”，不要完全从 0 开发全部软件和硬件。但是，怎样拿来呢？开发高可靠的产品，对使用外来代码会非常谨慎，甚至只使用有商业支持的外来代码。任何外来代码，都要经过通读、仔细分析，完全测试过以后才允许使用，有商业支持的代码，供应商会提供支持，这个过程进行得就快些；而没有商业支持的所谓开源代码，这些代码甚至没有完整的注释，读懂这样的代码，所花费的时间甚至不会比自己开发少多少。所以，高可靠产品的“拿来主义”，主要靠继承企业自身积累的软件模块，来降低开发工作量。要使新产品开发时有东西可以拿来用，企业产品开发过程只就要非常注重“可拿去性”，一句话，不拿去，何以拿来，而可拿去性，实质上就是可移植性！djyos 系统支持下，在系统设计时强调“可拿来性”，即系

统融合现成的软件模块的能力；在模块设计方面，强调“可拿去性”，即模块应该能适应不同的运行环境。djyos 系统融合现成模块的能力，重点不在于支持并发运行，而在于支持程序设计者组合现有模块成为一个新产品，以及设计可组合模块的能力。

可靠性和可移植性并不是孤立的指标，他们是互相促进的。可移植性是可靠性的基石，产品的可靠性来自于 3 个方面：

1. 正确设计以减少缺陷。
2. 充分测试以修正缺陷。
3. 充分使用中暴露缺陷。

开发工作是一个创新性很强的工作，没有一个设计师敢声称自己设计的产品是没有缺陷的，即使有人这样说，也不会有人相信。同样，没有一个测试师敢声称能找出所有 bug。反复使用是暴露 bug 的一个重要手段，同时也是暴露设计缺陷的手段。只有可移植性强的模块，才有机会在不同的产品、不同的使用环境中反复使用。“拿来”一个可移植的、经过反复使用考验的模块，显然是设计高可靠性产品的最好的保证。

目录

第1章 设计嵌入式产品

本书讲述嵌入式产品的软硬件设计，开讲之前，我们先来认识一下什么是嵌入式系统的问题，做到有的放矢。给嵌入式系统下定义是个困难的事，不同的人对嵌入式系统有不同的理解，有很多讲嵌入式系统的书，对嵌入式系统的定义也各不相同，没有统一的定义。因此，本书不企求对嵌入式系统下一个统一的定义，只是要交代一下本书所讨论的中心议题。嵌入式产品是以嵌入式系统为核心的电子产品，本书讨论如何设计优秀的嵌入式系统并把它组成

优越的产品。嵌入式系统是电子产品的一个功能模块，图 1-1 所示是一个典型的嵌入式产品组成结构图，它典型地包含以下模块，

- 计算机核心系统模块，与通用计算机不同，这是为专门用途设计的计算模块。硬件设计时，也会根据用途采取专门的设计，例如使用 DSP 处理大量数学运算需求。
- 包含专用软件，并且可以通过修改软件适应新的功能。
- 人机界面模块，这个模块也是为专门用途设计。
- 输入和输出模块，与通用计算机的 I/O 不同，嵌入式系统的 I/O 没有确定的地址，而且与具体硬件紧密相关，还可能包括模拟量 I/O。
- 有许多嵌入式系统还包含通信模块，这些模块往往用来连接成一个专用现场网络，像汽车上常用的 CAN 网一样；也可能用来与 PC 机连接，用于设定工作参数，转移采集的数据等。

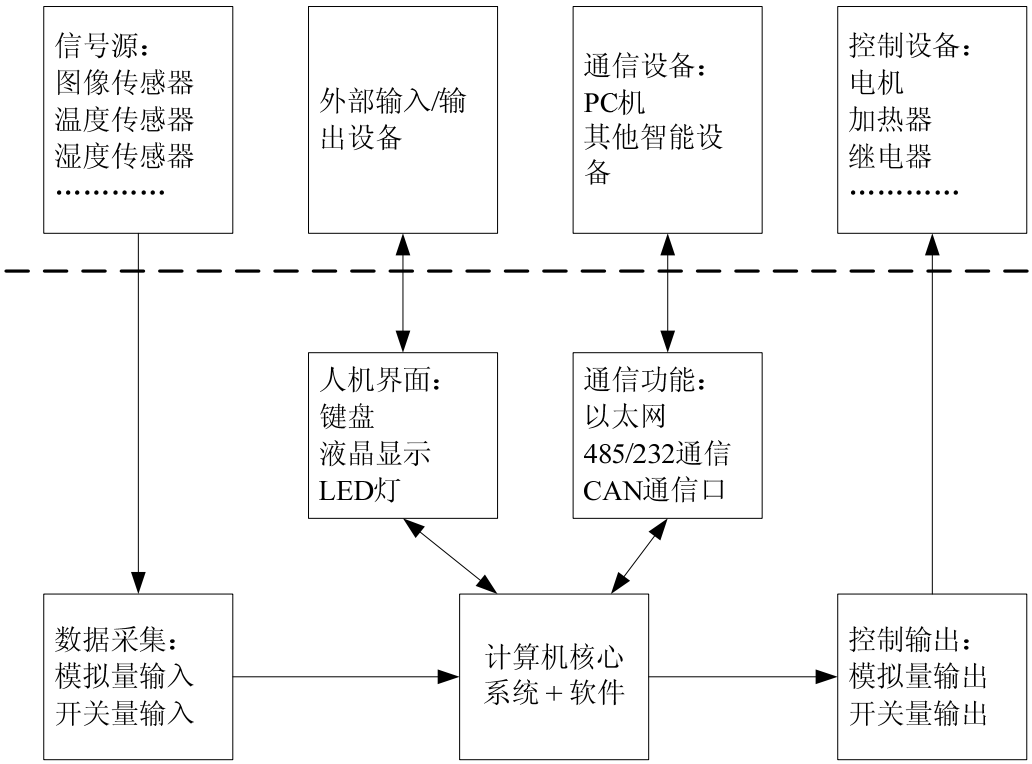


图 1-1 典型的嵌入式产品

1.1 嵌入式系统的特点

1.1.1 模拟电路和数字系统紧密结合

很多人都知道，传感器检测到的信号需要经过放大、滤波、A/D 转换后，才能被数字系统处理；CPU 输出的控制信号，也要经过与传感器检测相反的信号路径，才能实现功能。如何设计模拟电路与数字系统的实现方法息息相关，即使简单的模拟 RC 一阶低通滤波电路设计，使用精密电阻和电容，软件就不用进行复杂的补偿，而使用普通电阻和电容，降低了物料成本，但可能导致需要用软件做复杂的补偿。两种方案对软件运算的需求差异，甚至可以影响到 CPU 的选择，这单纯从软件设计和硬件设计方面都是很难想象的。

1.1.2 内存受限

桌面系统中，内存动辄几十几百 M，而且可以把磁盘作为虚拟内存交换使用，使程序员看起来拥有无限大的内存，一般的嵌入式系统，内存只有几十 K 至几 M，一般没有磁盘，即使有磁盘，也几乎没有硬盘页面交换能力，内存对系统的限制非常大。同时，为了保证系统的实时性，一般都没有内存碎片侦测和收集功能，所有内存使用都要由设计者直接干涉，一个由多人共同开发的系统，如何使用有限的内存，系统设计师应该在设计初期就做出决定，避免程序员在编程过程中随意申请内存，这样会导致混乱。

1.1.3 可靠性要求高

PC 软件是允许死机的，偶尔出点错，用户也不会较真，我们常戏称经常死机的软件为“盖茨标准”。但嵌入式控制系统经常用于关键的工业控制系统中，是不允许出错的，任何软件的或者是硬件的错误都可能是致命的。一个从技术角度讲并不算很严重的错误，如果发生在工业控制现场，就有可能是一场灾难性的事故。而很多工程师经常对一些小错误不以为然，比如当输出高电平时却输出了低电平，工程师常常一句“笔误而已”便敷衍过去了，但嵌入式工业控制系统中这种错误可能比系统崩溃更为严重。这样的“笔误”，如果出现在炼钢厂的高炉控制设备中，可能会造成一个价值数百万的设备损毁；如果出现在发电机保护设备中，可能使价值数亿的发电机报废并造成大面积停电；如果出现在生命维持设备中，那可真是人命关天的大事。

1.1.4 外设非标准

对于桌面系统，无论是台式机还是笔记本电脑，从早期的 XT 总线到后来的 ISA 总线到 PCI 总线，CPU 均用标准总线与外设相连。无论是串行口还是并行打印口还是 USB 总线，最后都转换成这些标准总线供 CPU 访问。PC 开发商甚至还对外设进行事先归类，不同种类的外设的接入地址和中断号都已经事先分配好，比如并口的 I/O 地址必然是 0x278 或 0x378，中断号必然是 IRQ5 或 IRQ7。外设供应商开发产品时，也总是使产品与某一种总线兼容，否则将无法接入计算机。或者说，桌面系统可以拒绝不符合自己要求的硬件接入。桌面系统的软件工程师，在开发硬件驱动程序或者硬件相关的产品时，只要使用操作系统提供的标准函数操作总线就可以了，剩下的就是按设定的格式解释数据。因此，桌面系统的硬件接口环境，对于嵌入式系统开发者来说，简直就是天堂！可怜的嵌入式开发工程师，他们面对的 CPU 种类超过 1000 种，操作系统也超过 1000 种。他们面前根本就没有总线的说法，要说有，也就是 CPU 的数据总线或者地址总线。嵌入式系统是用于解决某一个具体的问题而存在的，其硬件都是针对专用系统定制的，硬件的配置五花八门，它无需具备太多的通用性，自身浑然一体。只要你喜欢，你可以随意把一个外设安排在任意一个端口上，甚至可以用 IO 口线去操作内存，事实上，有些 MPU 根本就没有外部存储器接口总线，当内部存储器不够时，往往要设计专门的接口来扩充存储器，比如 I2C 接口的存储器。同时，软件需要处理各种硬件异常情况。如果软件处理不好，硬件的变化会对软件产生很大的干扰，使其升级维护极其痛苦。

还有一个容易被产品开发阶段忽略的问题，就是芯片停产改型带来的问题。工业及现场控制类的产品往往有较长的生命周期，一个产品开发定型后，连续销售 10 年甚至更长时

间都是有可能的。芯片厂商总是会不断地更新产品，停产老产品，工业控制类产品的产量往往不高，企业缺乏向芯片厂商叫板的筹码，遇到停产只能自己修改产品。在漫长的生命周期中，CPU 以及外围器件都有可能遭遇停产。人们往往只在硬件选型时考虑持续供货问题，而在软件开发中却缺乏考虑。当因器件供货问题局部修改设计时，往往会碰到新旧版本软件兼容升级的问题，不但增加了设计更新的工作量，也给软件版本管理带来极大的压力。

因此，嵌入式产品设计中一个很重要的问题就是统一外设，至少在一个企业范围内，要限制硬件设计的随意性，限制 cpu 种类，统一安排硬件地址，这样可以有效降解软件设计的复杂性，提高软件模块在一个企业内部不同产品之间的可移植性，要预估可能变化的硬件，对其相关代码组件化，当发生硬件配置变化时，单独修改相应的组件就可以了。在产品升级时，可以最大限度继承老版本的软件。

1.1.5 CPU 速度受限且非标准

嵌入式系统空间狭小，散热空间也小，出于可靠性的考虑，绝大多数没有散热风扇，这直接限制了 CPU 运算速度的提高。

所有的计算系统都一样，不同的配置间速度差别很大，但桌面计算机一般不应用于关键控制，速度快慢差别并不是致命的，顶多影响用户的等待时间；而大部分嵌入式系统用于实时控制环境，有些操作对定时要求相当严格，过快或者过慢都不行。CPU 运行快慢有可能对系统造成致命影响，编写软件时就应该考虑到运行速度的差别，防止留下隐患，优秀的软件，可以自动地适应不同的 CPU 运行速度。虽然同一个产品在硬件平台一般是不变的，但如果不注意系统速度差别的话，可以轻松地保证你的软件是不可移植的。

1.1.6 硬件平台不固定

嵌入式系统一般都是为专门用途设计，同一个产品在其生命周期内硬件平台一般是不变的，但是，一个企业一般都有系列化的产品，而且可能不止一个系列。同一个系列内的产品，可能会有一个公共的 CPU 平台，配以不同外设，衍生不同的产品；不同系列的产品，则 CPU 平台也可能不一样。更不幸的是，嵌入式系统工程师面对的还不仅仅是一个企业的产品，就算相似的应用，不同的公司使用的硬件配置也可能不一样，外设配置甚至 CPU 都会发生改变。为了避免重复劳动，特别是减少重复的 bug，嵌入式软件的跨软硬件平台移植的要求很高。设计者应该考虑到，当硬件和操作系统平台发生变化时，尽可能重用软件代码；当产品功能要求发生变化时，尽量保持硬件和操作系统平台的稳定。软件和硬件重用的意义主要有两方面，一方面可以缩短产品开发周期，加快产品推出时间，另一方面可以使产品越来越稳定，我们知道，软件是可能存在 bug 的，使用时间越长，则 bug 暴露得越充分，只要发现 bug 时及时修改，就能使 bug 越来越少，产品的可靠性也越来越高。

1.1.7 可确定的系统

计算机系统的可确定性包含两层意思，一是时间可确定，二是结果可确定。嵌入式系统的可确定性需求来自于两个方面，一是嵌入式系统可能用于实时控制，二是它是它可能用于无人值守或者无控制界面的环境。

实时系统对时间要求非常高，它要求计算机在限定的时间内完成计算任务并给出正确的

输出。通用计算机系统往往要求尽量高效地使用计算机，只要求计算机尽可能公平地干尽量多的活。嵌入式系统往往直接控制硬件，很多硬件有严格的定时要求，例如在自动剥线机控制中，要求精确控制刀口动作的时间以保证剥线的长度；在大米自动筛选机中，传感器首先感知到有不良大米颗粒到达筛选口，然后计算机通过采样和运算获知不良颗粒到达，再命令机械装置剔除不良颗粒，整个过程都要求快速而且精确定时，检测的速度和准确度直接影响筛选效果和生产效率。因此，许多嵌入式系统是时间关键系统，要求在指定的时间内完成指定的操作，而效率是否最高反而是次要的。在桌面系统中，对应用程序的定时需求，一般使用“尽量满足”的策略，而在嵌入式系统中，却要求“绝对满足”。这些区别使这嵌入式系统和桌面系统的设计观念上有很大的区别，他们的作业调度算法几乎完全不一样。

靠自己，别指望上帝！无人值守环境或者无控制界面环境意味着，如果计算机遇到异常情况下，这些异常情况包括外部错误的输入、计算机自身软件和硬件 bug 导致的错误，嵌入式系统应该自己做出决定，而不是等待人工干预。在桌面系统中，软件在遇到异常情况而不知道下一步该怎么做时，往往弹出对话框，让用户决定下一步该怎么做，把包袱扔给用户，而嵌入式系统却可能没有这种待遇，它必须自己做出决定。嵌入式系统的异常情况有两类，一类是可以预见但不可避免的，比如，硬件虚焊或者元器件损坏导致部分电路永久性失效，或者通信数据包由于干扰或者对方计算机出错导致错误的数据，这些都是可预见但不可避免的，嵌入式软件需要有确定的策略处理。另一类是可以避免的异常，比如内存不足，在支持动态内存分配的环境中，可能出现内存不足的情况，嵌入式系统也不例外。在桌面系统中，遇到内存不足时，操作系统可以提示用户关闭一些应用程序，或者关闭一些端口，而嵌入式环境呢，最好仔细计算好各计算任务的内存需求，按最大可能配置内存，或者在设计之初限制可能支持的通信端口数量、或者可打开的文件数量，以限制内存需求量，不要使关键任务在运行中得不到内存。

1.1.8 软件平台变化很大

与桌面平台 windows 系统和 unix 系操作系统一统天下相比，嵌入式系统的平台可谓是五花八门，全世界嵌入式操作系统有好几百种，流行较广的就不止 10 种，还有许多系统根本就不用操作系统。与嵌入式硬件和软件平台相对应，其开发平台也不一致，不同的平台有不同的编译器，这些编译器又大多根据特定平台有所扩展，以产生在特定平台上最优化的目标代码。但一般而言，我们尽量少使用编译器的特定扩展部分，最好只使用 C89 标准兼容的代码，在不得已使用特定扩展时，应该给出详细的注释，说明使用扩展功能的原因，最好还能给出标准 C 语言的实现方法。C99 标准虽然已经出台多年，但本书开始编写时，笔者广泛搜索，发现目前除 GNU C 涵盖了大部分 C99 标准特性以外，没有一个嵌入式开发工具宣布实现了 C99 标准。如果哪位读者发现了支持 C99 的嵌入式编译器，需请你慎重选用，不要让你的软件变得不可移植。

1.1.9 软件被固化

当然，嵌入式程序员得到的也不全是令人沮丧的消息，软件被固化在非易失存储器中的，只能由软件设计者修改，这就使嵌入式软件可以避开不怀好意者的搔扰。在桌面系统中，令人愤怒的病毒，令人讨厌的流氓软件、令人不快的垃圾邮件，使我们防不胜防，我们不得不安装各种防火墙，各种操作系统补丁，组成铜墙铁壁，把操作系统绑得严严实实，令计算机效率低下。即使如此，尚不能保证操作系统和应用程序正确运行，整天提心吊胆的生怕一不

小心中毒。在嵌入式环境中，一般不会执行外来程序，程序员所有工作就是使应用程序本身正确运行，而无需担心被外来程序破坏。

1.2 传统的设计方法

很长时间以来，软件和硬件分开设计被认为是天经地义的，硬件工程师只负责硬件，顶多再写一些驱动程序之类的简单代码，他们不知道软件特别是系统软件是如何使用 CPU 和内存的，他们遵循软件工程师的要求提供足够的内存和 CPU 运算速度，剩下的事情就不关他们事了；在他们看来，软件工作就是趴在电脑前写代码，把一堆 if-else 和 for（……）堆在一起就可以了。软件工程师完全不懂硬件也被认为是理所当然的，他们认为硬件设计就是选个 CPU，再给它添加一些内存，以及若干数量的外设，然后给画个 PCB 就可以了，谁都会做，他们不知道计算系统是如何工作的，也不知道如何编程才能使 CPU 系统发挥最大效用，才能降低硬件的复杂程度，从而降低成本。虽然有些企业会有一些嵌入式软件和硬件都熟悉的人才，他们在设计硬件时，就会考虑一些软件实现得问题，也会根据软件实现来调整硬件设计，但在传统设计方法下，本质上还是软硬件独立设计。这种设计方法将硬件和软件分为两个独立的部分。产品外特性确定以后，在整个设计过程中，通常软件和硬件方案会独立设计，具体实施则采用“硬件先行的原则”。这种设计方法存在许多弊端：

- 软件工程师做软件方案时，单纯使用软件方法，脱离硬件平台思考软件算法的问题。他们甚少考虑软件方案如何适应广泛的硬件环境，最终的结果也是软件在独特的硬件环境上调试通过，导致的后果是，硬件做任何细微的修改，都可能导致软件大幅度的修改，反之亦然。
- 硬件工程师做硬件方案设计时，会先向软件工程师征求意见，诸如需要多少位的 CPU，要多高主频，多少内存等，软件工程师则按照经验粗略估计硬件配置需求。硬件工程师获得配置需求后，独立进行硬件方案设计，方案设计过程中，硬件工程师主要从硬件的角度考虑问题，他们不熟悉软件方案，而且此时软件方案往往也还没有完成，他们甚少对软件方案提出建议。由于在硬件设计过程中缺乏对软件构架和实现机制的清晰了解，从软件工程师处获得的需求也很模糊，硬件设计工作带有一定的盲目性。软件工程师由于不熟悉硬件，估计硬件需求时，有经验的老工程师能给出一个比较准确的配置，一些年轻工程师便只能根据流行方案再结合宁多勿缺的原则给个配置，同样的功能，给出的配置可能差别很大，甚至会提出一些完全不切实际的要求。如果低估了系统配置要求，在软件开发过程中会要求硬件工程师修改设计，增加配置；如果高估了配置要求，冗余的硬件可能会增大体积，增加功耗，而且直接造成产品成本过高。无论硬件配置被高估还是低估，都会造成反复修改、反复试验，整个设计过程在很大程度上依赖于设计者的经验，在反复修改过程中，常常会在某些方面背离原始设计的要求。每一次硬件的修改都会付出大量的开发成本，更重要的是，每一次硬件修改都要经过原理图设计、PCB 设计，PCB 加工，元器件备料，板件焊接和调试这些工序，导致硬件改版的周期很长，这对项目进度的影响常常是致命的。由于开发进度的问题，使许多硬件的优化不能实现，反而要求软件削足适履去适应不太匹配的硬件。导致系统优化只能改善硬件软件各自的性能，不可能对系统做出较好的综合优化，得到的最终设计结果很难充分利用硬软件资源，难以适应现代复杂的、大规模的系统设计任务。
- 这种设计方法还不利于技术创新，嵌入式技术每天都在发展，新器件、新方案层出不穷。软硬件独立开发，由于专业方向的限制，硬件工程师可能不是太了解行

业软件发展水平，从而无法在硬件设计中予以支持，而最新的软件技术往往需要某些硬件方面的支持。例如，基于 RTOS 进行的软件设计，如果硬件能提供信号量指令，象 ARM 的 swp 指令，会大大提高系统的性能。而软件工程师则可能不了解最新的硬件技术发展趋势，在估计硬件需求时的依据也许是数年前的经验，不能根据当前技术发展水平提出恰当的需求。做软件方案时也不能很好地利用硬件技术进步带来的好处，使设计停留在落后的平台基础上。跟不上行业技术发展的节奏，你的产品就会缺乏竞争力，你所选用的器件可能很快停产，不从软硬件结合的角度考虑，你有可能会选择非主流的器件，这些器件的生命周期可能很短。嵌入式产品尤其是工业控制类产品往往有很长的生命周期，它要求厂家有持续供货和维护能力，而元器件的正常供应是最基本的要求，笔者所服务的公司就有一些 15 年前设计的产品，现在仍然在供货，现在生产的产品起码还要为用户维护 10 年，加起来至少 25 年的生命周期。你可能不得不使用老旧的开发工具，也可能得不到元器件厂家充分的技术支持，生产商一般对新器件提供最为充分的技术支持，而对老器件只提供最基本的支持。

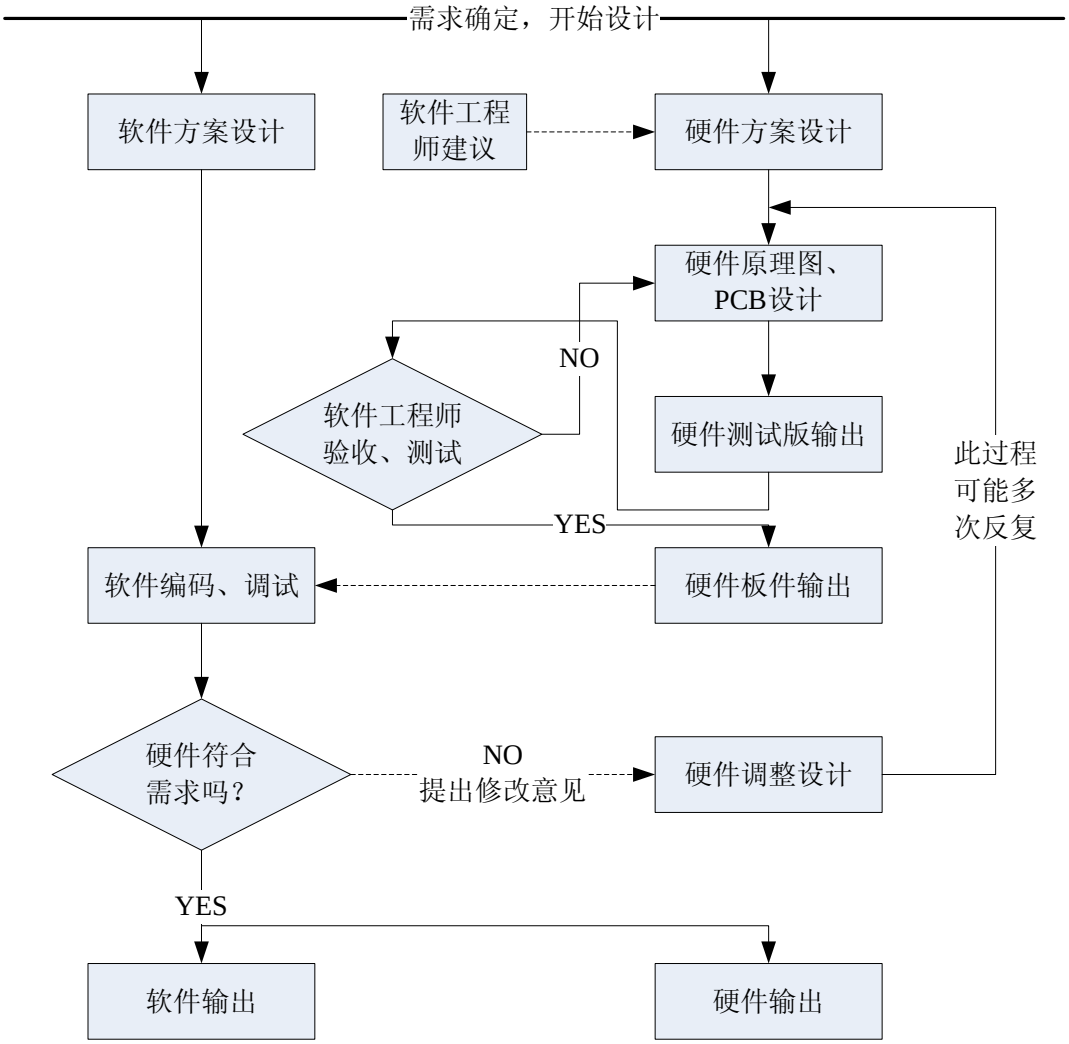


图 1-2 传统的、软硬件分开设计的流程

传统的、软硬件独立进行的设计，还易导致技术上“踢皮球”的现象。这里说的踢皮球，不是要讨论工作中可能出现的敷衍应付、推诿塞责的情况，这是企业管理的事，不是本书的主题。我们在这里要讨论的是，在所有开发者都认真负责的情况下，如何从技术组织和人员

分工角度上，防止出现在职责定位上模棱两可的问题。这个问题，在本书后续章节提到的djyos 操作系统中，也有考虑，其组件独立化的设计思想和支持，使问题局限在独立的组件内部，避免模糊。

我们在产品开发过程中，经常碰到分不清是软件问题还是硬件问题的情况，现在来看看笔者亲历的一个问题，为了避免涉嫌泄露笔者曾经服务的公司技术秘密，下面的描述是参照真实问题虚构的。有一个产品，开发时做了 10 台样机，该产品有通信功能，但调试中发现其中 9 台通信正常，1 台通信偶尔会少发送一些数据，该产品开发中，软件工程师与硬件工程师是分开的，项目经理也是软件出身，所有软件工程师包括项目经理都断定是硬件有问题，理由是相同的软件在其他机器上能正确，肯定是不正确的这台样机的硬件有故障，于是安排硬件工程师解决这个问题。硬件工程师用通信测试程序检测没有发现不正常的现象，检查各信号从通信线直到 CPU 引脚都是正常的，那些丢失的数据确实没有出现在 CPU 的输出端口上，确实是软件没有把数据发出来。而软件工程师用仿真器调试有故障的机器时，通信却又正常了。就这样，软件工程师和硬件工程师都很努力，都陷入苦闷之中，无从下手，他们怎么看都觉得问题与自己无关。时间在流逝，项目进度却不能拖延，硬件工程师实在没有办法的时候就开始啃软件代码，通过仔细分析发现了其中的问题，且看下面这段代码：

```
void func(void)
{
    ……
    if(status==true)
    {
        ……
    }else
    {
        ……
    }
    ……
}
```

这段代码中，如果if(status==true)条件成立，就会认为通信已经完成，不会再发送数据。status是一个综合条件，与CPU的一组IO端子上的信号有关。如 图 1-3 是其中一个IO端子示意图，实际电路中没有图中的R1。软件初始化CPU内部和外部硬件寄存器时，错误地把所有与该IO口相连的外设端口都设置成输入口，导致CPU的IO端子实际上处于类似悬空的状态。处于悬空状态的输入端子，读入的数据是不确定的，间接地导致if(status==true)语句错误判断，CPU就跳过这帧数据不发送。这本来是一个小问题，但查找的过程中走了许多弯路，花费了大量的时间，直接使开发进度后延了 10 天，为什么会出现这样的结果呢？

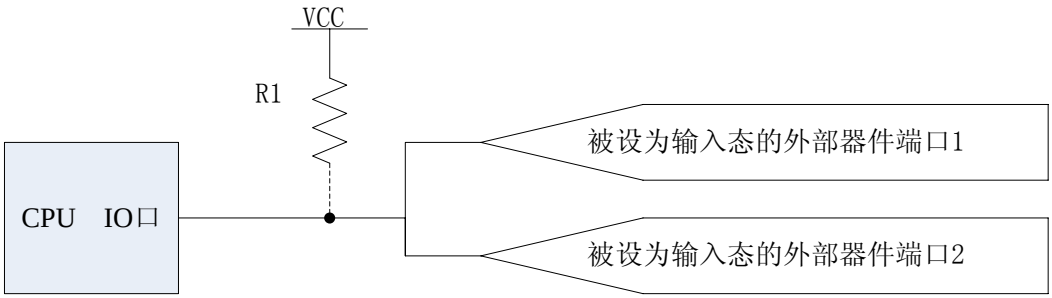


图 1-3 IO 端子设置不当引发软件 BUG

首先是这个项目的分工有问题，它把嵌入式系统的软件和硬件当成是两个独立的、不相关的模块，并且按照这种观点进行项目人员组织。这种技术观念和分工方法自然导致人才选择和培养上的缺陷，在建设软件工程师队伍时完全不考虑其是否掌握基本的硬件知识，硬件工程师也一样。这种分工组织最直接的后果就是没有人能够处理软件和硬件结合部分的问题，这个项目最后能把问题查出来，完全是侥幸，项目的硬件工程师恰好具备较强的软件知识，能读懂并分析代码。在实际工作中，这种分工方法也容易造成软件工程师和硬件工程师缺乏沟通，各自闭门造车。

其次是在技术本身，在软件工程师和硬件工程师独立分析问题的情况下，上述问题确实难于判断是硬件故障还是软件bug造成的。通信问题自然可能跟通信接口硬件相关，大家都知道，通信口与产品外部世界相连，容易受各种干扰甚至破坏性的干扰，比如静电放电，是容易出故障的部分，在只有一台样机异常的条件下，的确容易使人误判为硬件故障；该软件确实在大多数装置上运行流畅，只有个别装置上出问题，从日常经验上也容易判为硬件故障。但实际上这个故障是软件和硬件同时存在问题造成的，软件bug刚才已经讲过了，而硬件的问题在于没有从设计上避免CPU输入端子上数据不确定的情况，如果在设计时在CPU的IO上增加一个上拉电阻（如 图 1-3 中的R1），就可以是使输入信号在任何时候都是确定的，10台样机，要么全部通过，要么全部不通过，避免了随机性。顺便提一句，在设计输出控制电路时，也应该保证在CPU复位期间状态的确定性和安全性。

象这种和软件硬件都有关系的问题，如果不用软硬件综合的方法去分析，如果工程师不同时掌握软硬件知识，要排查这个问题确实不容易。在这个开发小组中，软件开发和硬件开发是严格分开的，硬件工程师只负责硬件，软件工程师只负责软件，软件和硬件的结合部就成了三不管地带，出了问题只能靠老天爷了。本例中，虽然最后由硬件工程师查出了问题，但却要付出加倍的代价，要知道，查别人写的半成品软件可不是一件容易的事。如果项目中不把软件和硬件割裂，而是有专人负责软件和硬件结合部，则情况可能会好很多。

那么，同样的软件，为什么会在 9 台样机上成功，只有一台不成功呢？问题就出在 IO 口悬空上，读入悬空的输入口上的数据，结果是不确定的，通常情况下，会得到一个固定的值，比如读入悬空的 TTL 输入口，一般得到 1，而 CMOS 口则大多数时候得到 0。但这样的 1 和 0 是不可靠的，他容易受各种干扰的影响，在一定条件下，就会满足本例中if(status==true)条件，不同的板子满足条件的情况出现几率不一样，恰好出故障那块出现的几率比较大，就出现了上述现象。而带上仿真器，又改变了电路的工作状态，使故障不再重现。其实，只要测试次数足够多，其他 9 块板子也会出现类似故障的，在后来的测试中验证了这个猜想。

1.3 软硬件联合设计方法

传统与联合的方法最大的差异在于软硬件等同的概念和软硬件模块间的接口设计上。

随着数字电子技术的发展，原来只有硬件才能解决的问题，越来越多地可以用软件方法实现；反过来，随着可编程逻辑器件功能的日益强大，硬件器件的可编程配置化，以及软件模块的硬件化，使得越来越多原来只能用软件实现的功能，可以用硬件实现。软硬件的界限已经不是非常清晰，很多问题可以从硬件方面去解决，也可以从软件的角度解决，需要根据实际情况灵活把握。这对嵌入式设计工程师提出了更高的要求，特别是系统设计工程师，需要同时掌握硬件的特征和软件的特征。

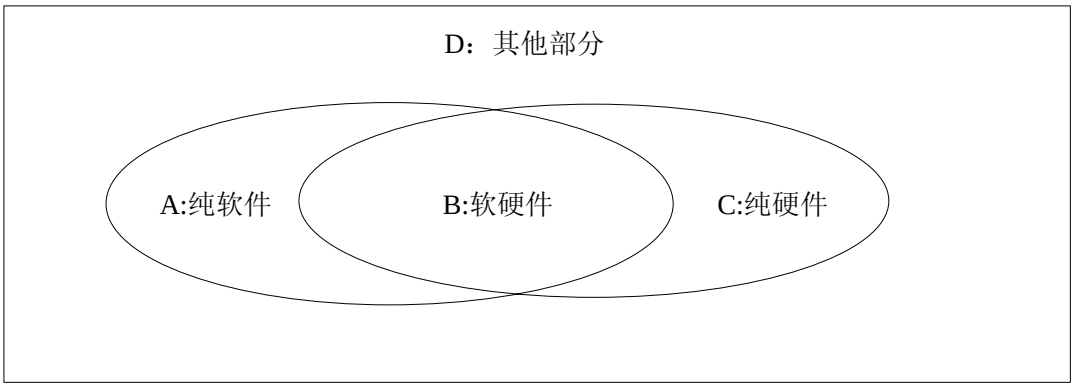


图 1-4 现代电子产品典型组成图

图 1-4 现代电子产品典型组成图，从图中我们看到，现代电子产品主要由 4 个部分组成，分别是：

A：纯软件部分，即只能用软件代码实现的部分，也有人说，从广义上说，是没有硬件实现不了的，也就没有纯软件，但这不免有些钻牛角尖了，这部分是软件工程师需要全面掌握的。

B：软硬件，即可用软件也可用硬件实现的部分，我们平常说的软硬件联合设计指的就是这部分。在这部分，软件工程师侧重于软件部分，需要了解硬件部分；硬件工程师侧重于硬件部分，但要了解软件部分；系统工程师则软件部分和硬件部分都需要熟练掌握。

C：纯硬件部分，即系统中只能用硬件实现的部分，例如显示器，射频模块，大功率电路等，这部分是硬件工程师需要熟练掌握的。

D：其他，即非电子部分，比如外壳，外观图案，商标等。

我们谈论软硬件联合设计，并不是不加区分地把硬件和软件撸在一起进行设计，这样你会无所适从。软硬件联合设计要解决两个问题，一是如何拆分系统需求，恰当地用软件和硬件实现系统部件，使系统综合性能最优化；二是要解决软硬件模块互相转化的问题，为了达到设计需求，软硬件联合设计要求部分软件模块和硬件模块具有互换性，在互换过程中不能大幅度修改设计，尤其不能修改与之无关的模块。第 9 章将提出一套解决软硬件互换问题的系统设计方法。要实现软硬件联合设计，要求系统设计师同时掌握软件和硬件知识，嵌入式软件工程师掌握基本的硬件知识，硬件工程师也要掌握基本的软件知识。不再把硬件和软件看成是有严格区分的领域，而应该是相互补充且相互可以替代的领域，把硬件功能模块和软件功能模块统一对待，统称做产品功能组件；在开发过程中，或者在维护过程中，排查故障和设计缺陷时，更不应该把硬件和软件割裂，这样会让你多走很多弯路。在 第 9 章中我们还会看到，软硬件联合设计的思想会使传统属于软件范畴的中断驱动程序和设备驱动程序的开发方法发生很大的变化。

图 1-5 是一个典型的软硬件联合设计的过程，整个产品开发过程中，都没有严格区分软件和硬件，而是以系统实现为中心，平衡开发进度、开发成本、生产成本、维护成本、产品性价比的条件下，软硬件择优使用。软硬件联合设计之于传统设计的区别，是一种设计思想上的转变，而不仅仅是软件设计和硬件设计谁先进行的差别。仅从项目进度上讲，软硬件联合设计也可能先执行硬件开发、再执行软件开发，特别是一些中小项目，需要部分工程师同时执行软件开发和硬件开发的工作。

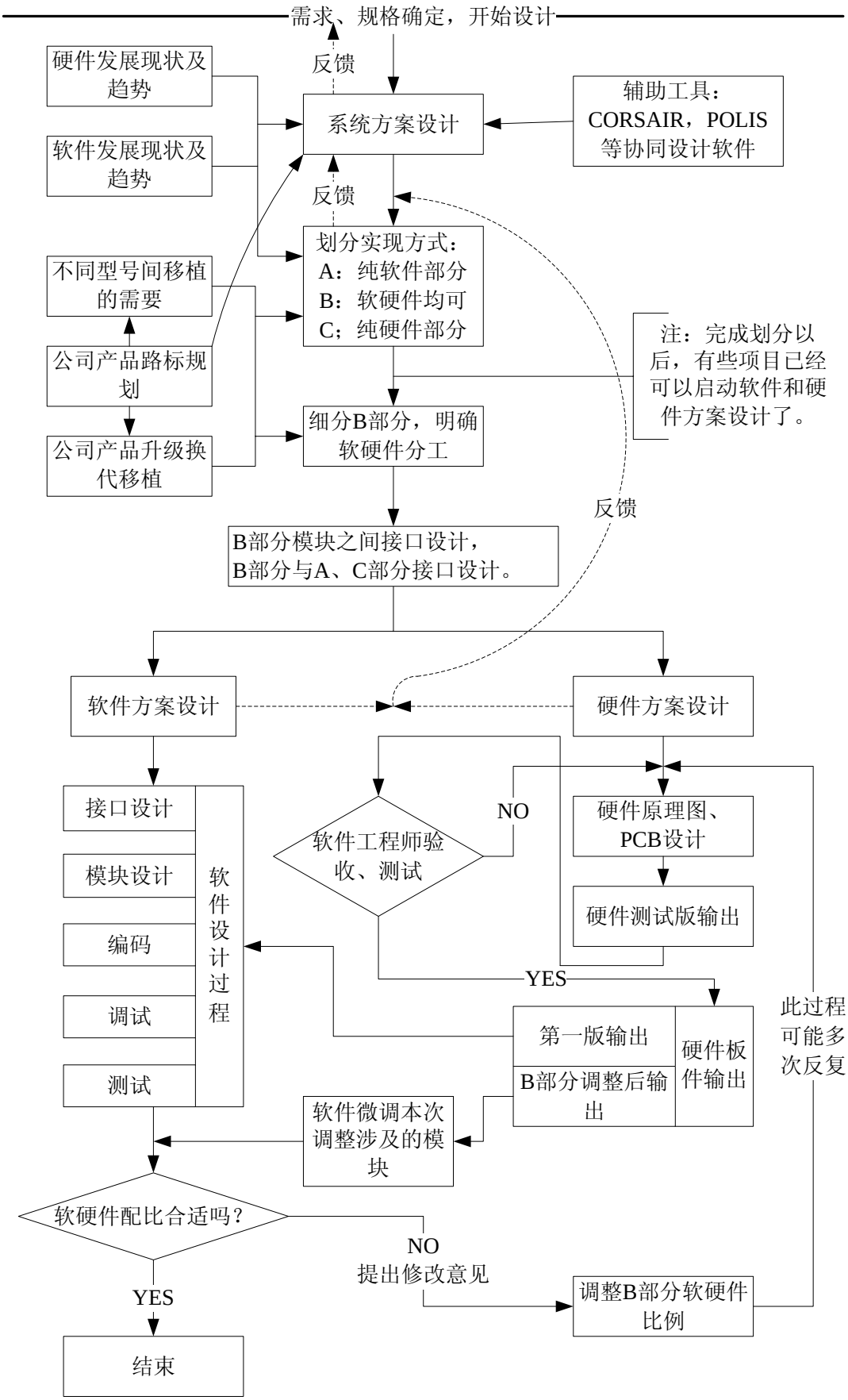


图 1-5 软硬件联合设计过程

1.3.1 系统方案设计阶段

需求和产品规格是本阶段最重要的输入数据，这里所说的需求是指经过提炼的、适合设计师阅读的需求，通常客户并不是嵌入式系统的专业设计人员，也不是产品设计人员，他们用自己的想象、自己的话而不是专业术语来表达需求，这种表达很不严谨，甚至会有一些不切实际的期望。对于用户描述的需求，不同的嵌入式设计人员会有不同的理解，直接作为设计输入会在设计团队中制造混乱。在开始设计之前，必须先把用户需求转化为用专业术语描述的需求和规格说明，这个说明必须是严谨的、一致的的说明，任何嵌入式设计人员阅读时都能得到一致的理解。

图 1-5 中，系统设计是产品设计最初的抽象阶段，在这个阶段，会把需求翻译成一个体系结构框图，框图中包含一个个相对独立的功能模块，规定了每个模块的主要功能和模块间的数据流。这个框图仍然很抽象，它没有设计任何实现细节，也没有划分软件部分和硬件部分。简化模块间数据流非常重要，过于复杂的数据流将增加系统的复杂性，以及模块间互相关联的可能性，甚至会使模块间产生过多的互相依赖性，我们称之为模块耦合，在 第 11 章有详细论述。模块间的耦合有两种，一种是运行时的耦合，两个模块争夺CPU，或者一个模块需要使用另一个模块输出的数据，就属于运行时耦合，同一个程序里的所有模块都或多或少地会有一定的运行耦合。运行时的耦合是不可避免的，只要合理分配好资源，这种耦合一般没有太大的危害。另一种是代码互相耦合，即一个模块的实现方法与另一个模块的实现方法有关，当修改一个模块内部实现时，可能导致其他模块也做相应的修改。模块间接口设计不当经常产生这种耦合，这种耦合就非常有害了。

在这个阶段中，企业产品路标规划起着举足轻重的作用，对企业产品开发来说，技术可移植性首要考虑的是企业内不同产品之间的移植（参见 11.7.3 节）。在系统设计阶段考虑引入产品路标，从全局角度进行抽象，会使产品具有最大的可移植性，使企业产品开发逐步走向标准化，开发工作越来越轻松，而大量模块的重复利用，将使企业产品越来越稳定。即使有些公司没有明确的产品路标规划，也要考虑企业产品可能的发展方向。有人会怀疑，这样会不会增加工作量，增加成本？这种担心是有道理的，在功能相同的情况下，可移植的模块需要考虑各种可能的情况，会比专用模块复杂一些，开发难度也相应会大些，导致开发成本的上升；更加复杂的模块会需要更多的资源和更快的CPU速度，这可能导致生产物料成本增加。但这也是相对的，因为你为别的产品提供可重用组件的同时，你也可以大量使用其他产品的组件，节省大量开发时间。因此，如果没有组件库或者库中的可用组件很少时，确实会增大开发难度；当产品组件库达到一定规模时，新产品开发就会如鱼得水，左右逢源。构件的重复使用，会大大增加产品的可维护性和可靠性，由此带来的生命周期成本降低，可能会弥补增加的生产成本甚至超过生产成本增加量。管理软件开发过程的 7 个主流模型中，就有 3 个是基于可重用组件的[5]，而最好的最可靠的可重用组件存在于企业自己的产品中。有许多企业都很重视货架技术建设，并有专门开发力量进行货架建设，“货架技术”实际上就是可重用的组件。根据产品路标规划，从每一个产品系统设计时提出，并在开发过程中实现的组件，具有无与伦比的实用性。

软硬件业界发展水平也是系统设计阶段首要考虑的因素，不能使用过于超前或者滞后的技术，孔子过犹不及的思想在这里得到了最好的诠释。系统方案需要用符合当前技术发展水平和趋势的、主流的技术方案去实现产品需求和规格书的要求，否则你会为此付出代价。为什么不能使用太超前的技术方案呢？

- 首先，会增加开发成本，超前的技术方案会使你的设计成为无根之木、无源之水，开发过程中能够的到的外部技术支持非常少，会使你的开发力量大量消耗在新技术

的探索中去。在这个技术分工日益细化的世界，嵌入式产品开发处于技术开发的下游，而处于上游的是芯片、开发工具厂商，中游厂商则开发许多中间件，比如嵌入式 gui 模块、TCP/IP 模块等，下游厂商应该使用上游厂商完成技术验证后的芯片和开发工具，再根据需要选用适当的中间件，然后构造自己的产品。系统方案如果过于超前，你将难于得到上游和中游厂商足够的支持，另外，在网络和 BBS 上讨论也是重要的开发支持资源，过于超前的技术在这里同样得不到支持。

- 其次，会增加生产成本，上游和中游厂商为了收回技术开发投资，在产品推出的初期，往往定价较高，然后逐步下降。
- 再次，会增加产品开发风险。电子技术发展的前端，总是百花齐放，百舸争流，各种先进技术激烈竞争，其中只有一小部分能够成为主流应用，非主流产品很快就会枯萎，相关芯片也会很快停产。超前的技术方案，如果没有侥幸地恰好选择了主流技术，将使我们所有的开发工作付之东流。

使用过于超前的技术方案不可取，那么，使用滞后的方案为什么也不对呢，这同样可以从技术、成本、风险角度进行分析。

- 现代电子产品开发中，上游厂商的技术支持非常重要，但是上游厂商的技术支持一般集中在其主流产品方面，对处于生命周期后期的产品，只提供基本的技术支持。如果系统方案滞后于当前主流技术，将迫使后续的硬件设计选择处于生命周期后期的芯片。不单是硬件，软件也同样面临问题，我们可能得不到合适的中间件，或者得到了中间件但是得不到充分的技术支持。
- 至于生产成本增加，这是显而易见的，所有厂商都会调高不再主推的芯片价格，迫使用户转移设计，直到停产。在开发过程中，选择一款芯片之前，要养成一个习惯，咨询芯片生产商，确认该芯片是否主流产品，也可以在网上查到该信息，大多数厂商会在网上列出产品目录，目录中，老产品上往往标明“不推荐用于新设计”，或者目录中根本就不列出即将淘汰的型号。
- 至于风险，是不言而喻的，处于生命周期后期的芯片，随时有停产的可能，芯片都停产了，还有什么说的。

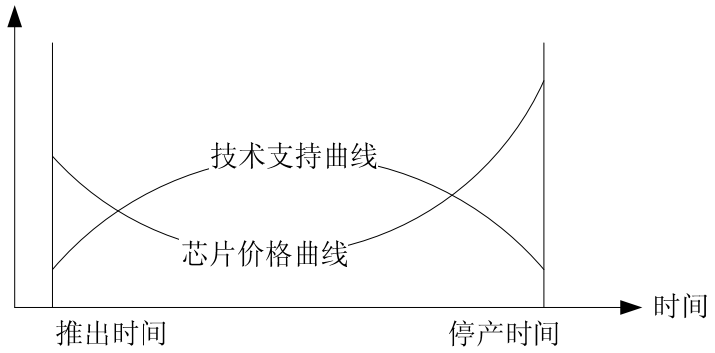


图 1-6 芯片的价格和厂商技术支持曲线

把握技术趋势是一项复杂艰巨的工作，而且存在一定的风险。一般来说，嵌入式系统的规格主要由计算机运行速度和存储器配置两方面决定，速度指标应该是目标系统相关领域的测试基准衡量的实际系统运行速度，而不是 cpu 主频，比如用于整数数字逻辑处理的计算机，可以使用 dhrystone 基准，用于浮点运算的计算机，用 whrystone 基准，用于数学分析的计算机，则应该使用 DBTImark2000 基准……。在系统设计阶段，需要确定嵌入式系统的规格，这要求系统设计师必需对整个系统的复杂性有充分的认识，需要大体确定哪些构件由硬件实现，哪些构件由软件实现，哪些构件由软硬件共同实现。嵌入式系统的规格也基本确定了其

造价和开发工作量，这又可以反过来核对需求和规格书是否合理。

在系统设计阶段，应用面向对象的方法非常重要，面向对象的设计方法虽然首先是从软件学中发展起来的，但它并不是软件设计的专利。设计师应该从需求和规格书中抽象出系统组件的工作，其实就是把需求对象化，而这些系统对象必需有严谨而完备的输入输出说明，但并不涉及其内部实现细节，也不关心组件对象的内部行为，更不用关心这个组件是用软件实现还是用硬件实现的。这样，既明确规定了下一步需要做的工作，又留下了广阔的灵活设计空间。

系统方案中还要考虑易于用相匹配的软件方案和硬件方案实现，如果在系统方案设计时考虑不周，那么在后续的软件方案设计和硬件方案设计中，就有可能迫使工程师选择不相匹配的方案。

1.3.2 软硬件分工设计阶段

完成系统设计后，接下来要做的就是软硬件划分，在这个阶段，系统设计工程师的实际灵感、知识结构和经验非常重要，一个项目，通常分为：

纯软件部分。

纯硬件部分。

软硬件均可以实现的部分。

软硬件共同实现的部分。

划分模块后，设计模块间和模块内的接口是本阶段最重要的工作。

有些模块，它并不是单纯的用软件或者用硬件实现，而是用软件接口和硬件共同实现，比如模拟量输入模块就是非常典型的一种。在这种模块就像跷跷板一样，既可以朝硬件倾斜，采用高性能的 A/D 转换器件，让硬件做更多的工作，以降低软件的负担；又可以朝软件倾斜，用软件的方法提高 A/D 转换性能，让软件做更多的工作，可以降低产品成本。

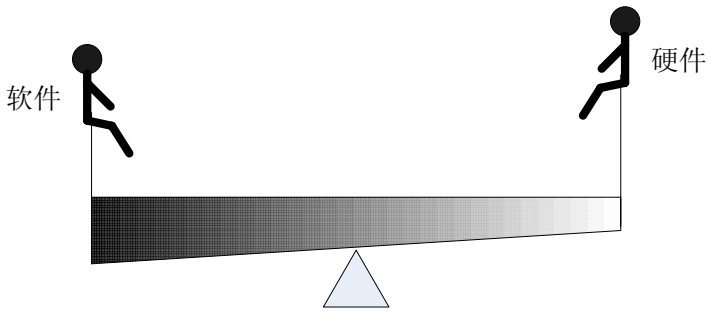


图 1-7 象跷跷板一样改变软硬件实现比例

以成本换进度，迅速占领市场，然后再进一步开发以降低成本，是企业竞争中常见的策略。在这种市场策略指导下，产品开发的初期是不会仔细计较成本的，表现在 图 1-7 中，就是硬件的分量大于软件的分量。此时，设计师不会仔细估计所需要的计算机性能，可能选择大大超出要求的CPU；不会仔细估计存储器速度和容量需要，片面地配置高速大容量存储器；外设配置方面，也会片面地配置高性能外设。总之，能实现需求和规格书的要求就好，成本不是最重要的。产品成功推向市场后，继续进行二次开发，发展功能相同的低成本版本，或者根据市场反应停止开发。这个时候，软件的分量就要加重，软件开发虽然要付出开发成本，但是制造成本会因此大大降低。

传统开发模式下，软件针对特定硬件开发，低成本方案开发时，如此大规模地修改硬件配置，很可能会引起软件大规模的修改，甚至重新开发。使用软硬件联合设计的方法，整个软硬件系统天生就有匹配不同软硬件配置的能力，修改硬件配置时，完全可以做到只修改与

被改动的硬件直接相关的代码，使“高成本版本”的软硬件模块可以最大限度地转移到“低成本版本”中，使两个版本保持高度一致性，这种一致性将带来莫大的好处。

首先，可以降低二次开发的工作量和测试工作量，加快低成本版本的上市时间，为企业节省大量开发和生产成本。

其次，可以降低维护成本，两个版本保持一致，可以用一致的方法维护，用户也可以用一致的方法养护产品，这两者都可以大大降低维护和培训成本。

1.3.3 反馈－软硬件分工调整阶段

在系统设计阶段，要准确估计软硬件需求是极其困难的，甚至是不可能的。现代嵌入式产品是如此复杂，准确估计单个模块的计算机运算能力需求和内存需求都是很困难的，不同的软件算法对硬件的需求都不一样。而且，嵌入式系统除了 CPU 运算能力和内存以外，还涉及到许多实时性的问题，模块间配合的问题，这些都对我们估计软硬件配置和比例提出了挑战。既然我们不能指望准确估计硬件需求，那么，我们最好避开它，不战而屈人之兵才是上上之策。避开并不是逃避，完成产品设计才是我们的最终目的，我们需要一种设计方法和技术，保证我们不需要在系统设计阶段准确估计软硬件配比，使我们既可以在设计过程中灵活调整软硬件配置和比例，又能够控制开发进度。

运用软硬件联合设计方法，我们可以在系统设计阶段粗略估计软硬件配置，并给软硬件可以互相转化实现的部分设计严格的接口，使其高度封装，我们姑且称这种模块为“协作模块”。其他的软件模块和硬件模块只能看见封装后的协作模块，而根本不知道其内部是由软件实现还是硬件实现。如果某协同模块用软件实现，它就必然占用cpu执行时间和内存资源，如果在软件设计过程中发现资源紧缺，就可以要求调整该协作模块的软硬件配比，并反馈给硬件设计，要求增加该协作模块的硬件功能，等硬件设计完成时删除该协作模块的冗余的软件部分。整个过程就像 图 1-7 所示的跷跷板一样，所有的修改工作均封闭在这个模块内部进行，不影响别的模块，也不影响系统运行，硬件开发工作和软件开发工作均按自己的程序进行，互不影响，反之亦然。

软件设计中反馈调整硬件设计，是传统设计方法和软硬件联合设计方法都需要的步骤。但是两者有着巨大的区别，对传统设计方法来说，软件要求调整硬件设计是一种设计错误导致的异常事件，会对开发进程造成冲击，有经验的工程师会为估计发生这种异常事件的可能性，并在计划中预留缓冲时间。但是，究竟保留多少时间、哪里可能出现异常、什么时候会发生异常，只有天知道！这种冲击严重的可能导致硬件整体被推翻，重新设计。现实中，许多项目延迟就是因为这种软硬件之间反复调整导致的。传统设计方法常常使软件针对特定硬件设计，没有对可能的硬件修改保留兼容接口，不具备可移植性，硬件的修改反过来影响软件设计，导致软件大规模修改。如果由于进度的问题，不允许硬件重新设计的话，那么软件就必须削足适履地适应不太合适的硬件，从而限制了产品性能。而对于软硬件联合设计来说，由于在系统设计阶段就已经考虑到了可能的软硬件配置调整，并且为软硬件协作完成的功能独立出来并做了完备的封装，软硬件可能出现交互的地方也设计了良好的接口，当软件设计要求调整硬件时，硬件稍作修改就可以；软件工作也无需因此而停下来等待硬件，因为软件并不会因硬件调整做大的修改，仅仅是局部模块的内部做些微调而已，开发过程不会依赖新版本软件，完全可以在硬件的调整过程中继续进行。由于在系统设计阶段就明确了协作模块的范围，硬件设计甚至可以预先估计可能的软硬件配比调整，预先完成多种版本的硬件，在 PCB 设计中预先留下位置，预先布线，使软件设计要求调整硬件时焊接或者取下抑或更换器件就立即得到新版本。有许多电子元器件有序列化型号，同一序列的不同型号的功能相同，性能不同，引脚兼容，可以直接互换，这也是动态调整软硬件配比的利器。例如串口驱动器，

ST16C754 和 ST16C554 的功能就完全一样，区别是 754 有更深的 fifo，大容量 fifo 可以降低软件开销，我们选择 754 还是 554 的过程，实际上就是调整串口通信模块软硬件配比的过程；又如 AD7656/57/58 序列的 ADC，功能完全一样，pin-to-pin 互换，不同的是精度不一样。采样理论告诉我们，过采样可以提高 ADC 精度，但过采样需要大量的软件开销，等于说，采样系统的精度要求已经确定的情况下，加大软件开销，可以降低 ADC 本身的精度要求，等同于降低了硬件成本和设计难度；采用高精度的 ADC 型号，可以降低软件开销，选择 ADC 的具体型号，实际上等效于调整模拟量转换模块的软硬件配比。软硬件联合设计，完全可以采用这种兼容器件来实现，那么，软硬件的调整就是更换器件而已。

总之，对软硬件联合设计方法来说，软硬件配比的调整是主动的而且是可控的过程，它不会打乱开发进程，并使最终产品获得最佳的性能；而传统设计方法中，软硬件调整是被动的，不可控的过程，它可能破坏整个开发计划，而且可能在最终产品中留下缺陷。

1.3.4 联合设计会影响进度吗？

软硬件联合设计使我们在开发的初始阶段需要做更多更细致的工作，具体软硬件启动时间肯定会比传统设计方法晚，有不少人不愿意花这个时间，他们认为这样会延误开发进度，他们更喜欢的是短时间内见到阶段性成果。然而，我要告诉你，一个较为复杂的产品，从开始到完成第一版样机的时间只占开发周期总时间很小的一部分。样机完成后，开发远未结束，还要花大量的时间进行软硬件的完善工作，测试工作所占的时间也不在少数。

产品开发过程中，第一步需要做的是需求和规格定义，此时，具体的设计工作还没有开始，需求个规格定义实际上是有一定的盲目性的，它需要根据后续开发工作中反馈的信息调整规格。同样，系统设计阶段的工作也是需要调整的，这个调整的过程可能出现反复，样机也许需要做许多版本。联合设计由于软硬件调整很方便，降低了在系统设计阶段准确估计硬件性能的要求，开发过程中硬件设计调整也不会影响软件设计的进度。联合设计虽然可能延迟第一版样机的完成时间，但会大大加快后续改进版本的开发进度，而且，由于在系统设计阶段做了更加细致的工作，还可以减少样机版本数量，实际上非但不会影响进度，反而会大大促进开发进度。

1.4 软硬件联合设计范例

1.4.1 需求

我们以一个嵌入式系统中最常见的串行通信模块为例，这是一个简单的工控产品，其系统结构图如 图 1-8 所示，我们要设计的是其中的总控机部分。

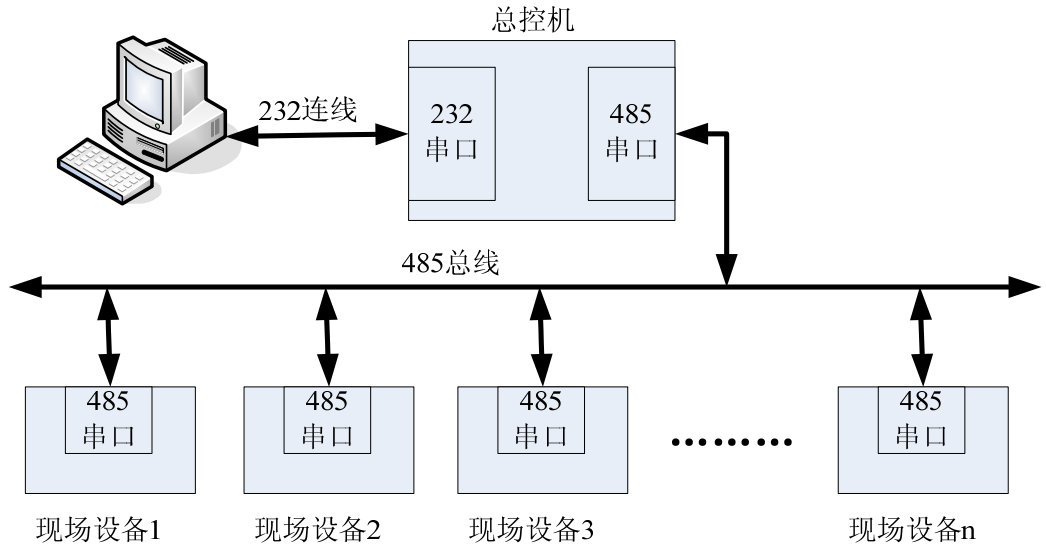


图 1-8 一个典型的现场控制系统

总控机功能需求如下：

- 用 485 总线与监控设备通信，采集现场设备的数据，向监控设备发送指令，控制监控设备工作。Baud＝9600。
- 在 PC 机读取数据之前暂存被采集的数据，并记录数据采集时间。
- 通过串口与 PC 机通信，把采集量上送；接受 PC 机的控制命令，改变对监控设备的控制方法。通信 Baud＝9600。
- 用 IO 口点 LED 灯，显示通信口状态和自身的工作状态。

我们来试着设计这个产品的电路及部分软件实现。

1.4.2 设计方案

本产品需要两个串口，但计算量不大，考虑选用MCS51 单片机，这个单片机公司原来用过，开发工具是现成的，软硬件开发工程师对这个单片机都比较熟悉，容易上手，可以加快开发进度。但该单片机只有一个串口，虽然也有一些拥有两个串口的型号，但选择范围窄且价格很高，现在 51 机的工作频率已经可以达到 40M，并且有双倍速甚至是 12 倍速的型号，CPU的运算资源还有冗余，因此决定使用CPU的IO口模拟一个串口，系统结构如 图 1-9。考虑到IO模拟串口的性能瓶颈问题，将来产品升级很可能被替换成硬件串口，故必须把它设计成一个软硬件协同模块，它虽然在CPU内部，但与其他软件隔离，只提供一个与硬件串口一致的接口与其他软件模块打交道。

内存方面，由于从机数据量不大，软件工程师认为外扩个 32K 就足够了，flash 有 4K 就足够了。

工业产品可靠性要求高，要外接一个看门狗电路，因为要记录时间，还要加一个实时时钟电路。

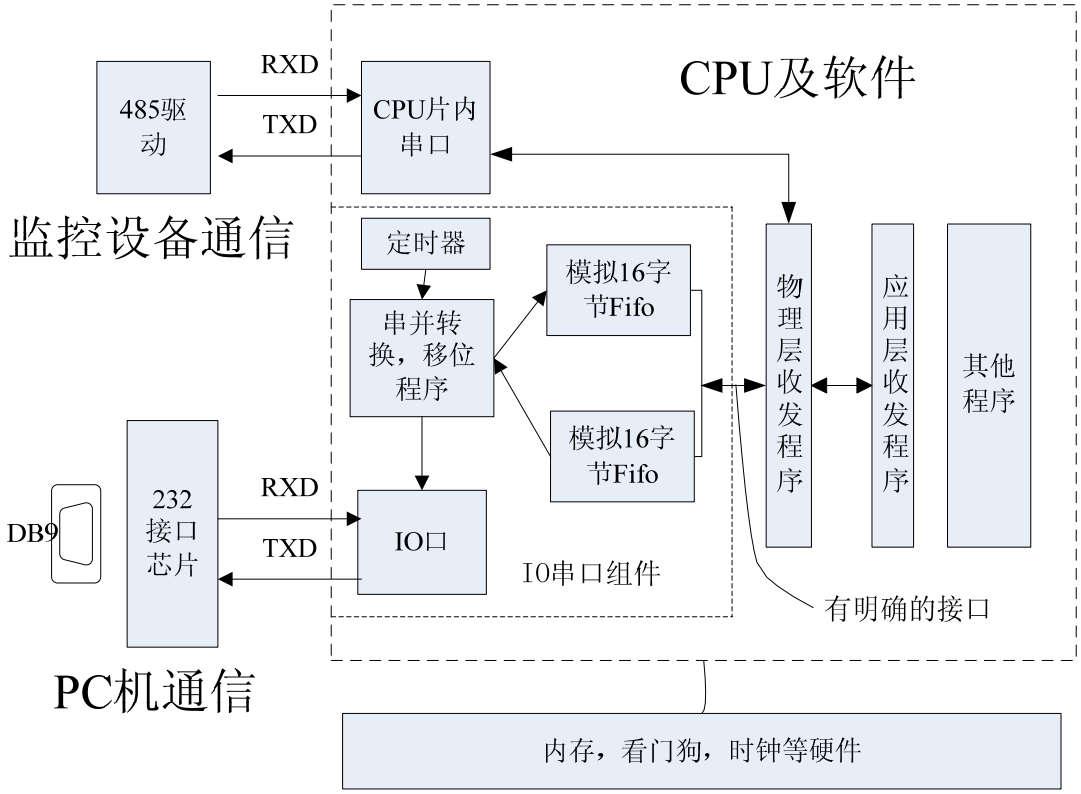


图 1-9 主控机软硬件结构

单片机本身的串口用于和监控设备的485总线通信，IO口模拟的串口用于和PC机通信，因为 Baud＝9600，每个 bit 收发需 104.2uS，考虑到要分辨脉冲宽度才能正确接收，因此在 104.2uS 里至少要查询 3 次 IO 口，因此启动一个 35uS 的定时中断，该中断不停地查询 IO 口状态，检查到有有效起始位时开始接收，并把接收到的位通过移位程序拼成完整的字节。发送时，则每 3 次中断往发送 IO 口移出一位数据，直到发送完成。下面是软件伪代码。

主程序：

初始化各种硬件；

while（1）{

SerialReceiver(com_IO); *//模拟串口接收*

SerialReceiver(com_Hard); *//硬件串口接收*

OtherModule( ); *//其他模块,包括串口应用层收发模块*

}

物理层接收程序

if (fifo 中有数据接收){

把数据读到接收缓冲区;

向应用层接收模块发信号;

}else

return;

中断服务程序：

中断程序开始;

保护现场;

if（状态变量＝＝等待起始位）{

读取 IO 状态;

```
    if（收到起始位）
        状态变量＝接收数据
}else if（状态变量＝＝接收数据）{
    接收 1 位数据；
    接收到完整数据位则移入接收寄存器；
    if（接收到完整字节）{
        数据存入模拟 fifo;
        if (fifo 达到触发条件)
            设置接收标志；          //用于中断外程序从 Fifo 读数据
        状态变量＝等待停止位；
    }
}else   if（置状态变量＝＝等待停止位）{
    if（收到停止位）{
        状态变量＝等待起始位；
    }
    if（超时）
        置错误状态；
}
if（fifo 中有数据待发送）
    每 3 次中断输出一位数据到 IO 口；直到发送完成；
恢复现场；
中断程序返回；
```

该中断程序需要 5～15uS 的执行时间，具体执行时间与串口工作状态有关。假设在开发中我们碰到几个问题，讨论一下处理办法。

1.4.3 产品功能升级

当产品使用一段时间后，由于通信数据量的增加，9600 Baud 不能满足需要，欲增加到 57600 时，怎么办？

Baud＝57600 时，时钟中断间隔时间只有 5.8uS，中断程序的执行时间却要 5～15uS，显然不能满足要求。怎么解决这个矛盾呢，通常会用以下几种方法解决问题

- 如果中断的执行时间足够短，系统是能够满足要求的，通过软件优化，用汇编重写，或者改进算法，加快中断程序执行时间，使其满足要求。这种方法的好处是不需要修改硬件，仅升级软件就可以了。
- 当升级软件不能达到要求时，可以通过提高主频，或者使用效率更高的 CPU，缩短中断程序的执行时间。这种方法在同系列 CPU 中有合适的型号时，硬件的改动最小，同时软件移植工作量也小。
- 当以上两种方法不能奏效时，我们分析，CPU用于串口通信的主要时间是花在一位一位处理数据上，那么使用专用硬件，比如改用内部带两个Uart的CPU，或者使用 16C554 等专用硬件，代替 图 1-9 中的“IO串口模块”。这样修改后，系统框图就变成了 图 1-10 的形式

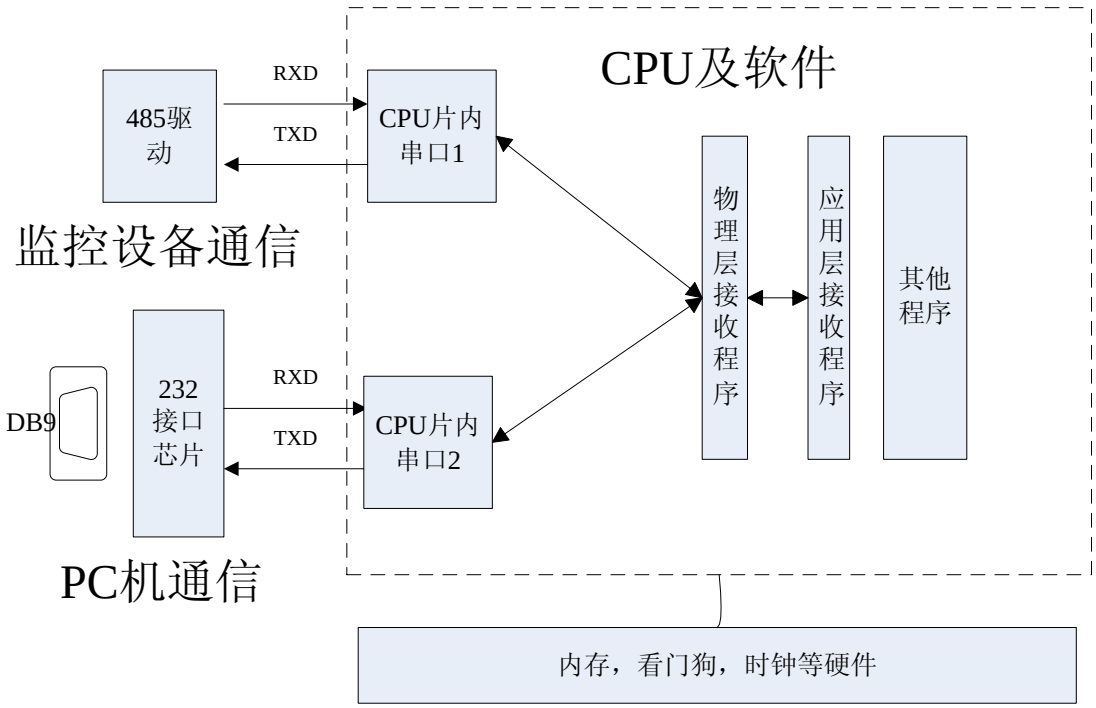


图 1-10 用硬件加速主控机串口

从问题二的解决方法中，我们可以看到，一个功能的实现，可以用硬件的手段，也可以用软件的手段，换一种思维，当用专用硬件实现串口接收时，我们可以看做是把 图 1-9 中的用于串口接收的部分软件和硬件都搬到了CPU外面，用硬件的方法实现了软件模块。实际上 16C554 是典型的软件硬化后的硬件产品。因此，我们应该改变固化的“硬件模块”和“软件模块”的观点，在系统设计初期，只有“功能模块”，具体一个功能模块在实现时究竟用软件还是硬件，是要综合衡量的。在解决问题时，也不能想当然地认为哪一类故障是软件故障，哪一类故障是硬件故障，而是应该从软硬件联合角度去分析。

1.5 软硬件协同分析问题

在产品开发过程中，或者是产品生命周期的维护过程中，可能会发生很多问题需要分析。用软硬件联合设计的产品，当然要用软硬件协同的方法来分析问题。

1.5.1 软硬件相互转化

我们经常难于分清软件故障还是硬件故障，是因为他们根本就是一致的东西，在一定的条件下，会互相转化的，硬是人为地严格区分软件故障还是硬件故障，只能使我们迷失方向，使本来就艰难的工作添加更多的障碍。根据设计方法和运行条件的不同，性质和外部表现都相同的问题，既有可能是软件 bug 造成的，也可能是硬件故障或者缺陷造成的。因此，我们在开发调试和维护产品时，一定要了解软件和硬件各自的特性，避免主观臆想的思维定势，不能抱定哪些问题是软件问题，哪些问题是硬件问题，记住软加问题和硬件问题有时候是很难区分且可能互相转化的。因此，需要用软硬件联合的角度，从故障特征判断是哪一个功能模块出了问题，然后再根据这个功能模块的实现方式进一步定位问题的位置，直到这个时候，我们才能真正区分问题出在软件还是硬件上，还是软硬件配合上。

调试第 1.4 节所设计的产品时，发现装置与PC机通信有时候会中断，经检查确认PC端

没有问题。检查此类问题时，我们一般会从以下几个方面查问题：

- 首先确定硬件有没有问题，用示波器检查 232 转换芯片输出是否正常，检查 232 接收信号是否正常。如果信号不正常，则检查连接线接触是否可靠，芯片有没有虚焊，232 电平转换芯片是否损坏。
- 如果上述检查通过，则怀疑软件有 Bug，用仿真器或者其他调试手段检查中断程序是否正确接收到 PC 机发送的数据，再检查发送给 PC 机的数据是否正确地送到发送 Fifo。就这样沿着程序执行时数据流向一步一步地查下去，直到查到 Bug 的位置。

我们看到，排查问题时，我们是同时从软件和硬件量两方面着手的，并不是孤立地怀疑是硬件故障还是软件故障。经过跟踪程序运行，我们发现不管有没有收到数据，“*fifo 中有数据接收*”的标志总是被置成 1，在主程序里查询该标志，查到有数据接收标志后就启动数据接收程序，如果中断总是设置该标志，软件就总是忙于接收数据，造成通信程序不正常。除通信程序工作不正常外，由于通信程序占用大量 CPU 时间，轻则会造成系统变慢，重则造成系统死机，使软件失去反应能力。这个问题的解决方法很多，在这里就不浪费篇幅了。

这是一个软件容错性方面的问题，一个强壮的软件，应该在一个模块出现错误时，故障限制在出错的模块内部，仅出错模块所在的功能部件工作不正常，其他模块应该不受影响而且能通过设定的方法报告错误并记录。在本案例中，一个模块工作异常就影响整机运行甚至造成整机死机是不允许的。

现在讲一个笔者亲身经历过的一个案例，该项目中软硬件也是分开设计的。某产品需要较多的串行通信口，使用了一片 16C554 用于串口通信扩展，16C554 的 4 个中断输出相或后连接到CPU的外部中断上，如 图 1-11 所示。16C554 的 4 根中断线分别对应 4 个串口，串口申请中断的方式是使相应的中断线变成高电平，中断状态被清除后，恢复低电平。16C554 的具体工作方式参见它的数据手册，本书不表。

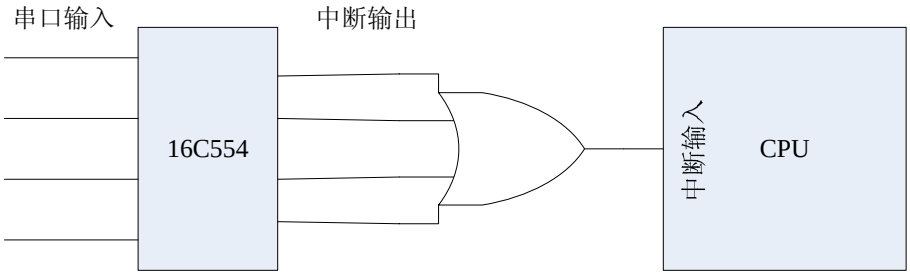


图 1-11 4 个中断信号相或后接到 CPU 的一个中断输入上

该例中，在系统设计时并没有规定 CPU 使用电平中断还是边沿中断方式，软件工程师按照自己的习惯选择电平中断方式，只要外部中断线输出高电平就进入中断。开始运行了很长时间都没有发生问题，终于有一天问题暴露了出来，一台产品上电不工作，键盘、显示、通信都没有任何反应，后来又陆续出现了几次。最后查出来的结果是 16C554 故障，有一个中断引脚总是高电平，造成 CPU 一直出于中断状态下。该产品使用的是 ARM 系列的 CPU，该 CPU 与 X86CPU 不一样，X86 系列在中断返回后至少要执行一条指令后才会再次进入中断，即使出现连续中断的情况，仍然会对外部操作做出蜗牛式的反应，而 ARM 的中断结构不一样，当中断函数返回后，如果中断信号仍然存在，就会立即进入新的中断服务函数，中断外程序一条指令都不会执行，导致产品没有任何反应。

本节描述的两个问题，其本质是相同的，都可以描述成“当系统出现一个异常信号时，使整个系统包括与该信号无关部分不工作”。但前者毫无疑问是软件 Bug，中断程序有一个明确直接的 Bug，主程序有容错性方面的 Bug，软件工程师责无旁贷。而后者却可以扯皮，硬件工程师会说“这是元器件故障，我不能完全杜绝”，而软件工程师也会说“这是硬件故障，跟软件没有关系，这不，换硬件就好了！”，最后是谁都没有责任，bug 随着产品一同发

布出去。这两个性质一致的问题，对 CPU 来说，都是判断一个标志量，根据这个标志量的值决定是否要读串口接收缓冲区，所不同的是，前者这个标志量由中断程序给出，后者通过 CPU 的中断引脚给出；而两者造成的后果都是一样的，就是造成 CPU 忙于执行串口接收工作，导致系统异常甚至死机；两者的解决方法也一样，当 CPU 检测到该标志总是存在时，屏蔽相应的中断，使错误仅局限于出错模块，使其他模块正常工作。在这里，同一个功能，根据实现方法的不同，软硬件问题相互转化了。

我们常常会碰到一些潜伏很深的故障，几小时，几天甚至几个月才出现一次，这说明，很多故障是要符合特定的条件才能暴露出来的。由于硬件的离散性，输入端子上的电压处于临界状态时，CPU 读该端子上的数据，有些器件可能读到 1，另一些器件可能读到 0，有些特定条件可能在这块板出现，而另一块板则不出现。而大量的可编程，可配置的硬件的出现，如果这些器件没有正确地编程或配置，往往会使程序运行出现截然不同的结果。我们常常听到软件工程师抱怨，一样的软件，在这块板上能运行，而在那块板上则出现错误，昨天能运行，今天就不能运行了，从而陷入迷茫中，其实出现这类故障，有可能是硬件有故障，也有可能是软件对硬件的离散性或者初始化状态考虑不足，或者没有对可配置的硬件正确初始化，例如未初始化的内存。嵌入式软件工程师应该对硬件尤其是可编程可配置的硬件有所了解。而硬件开发和底层驱动程序的开发，也要考虑尽可能地给软件工程师一个标准化的，一致的编程接口。

1.5.2 软硬件互存互容

既然有许多功能使用软件和硬件都可以达到，我们在设计中更倾向于把软件和硬件模块等同对待，在系统中互存互容，使系统功能高度积木化，这会对软件开发提出更高的要求。高度积木化的设计，使产品系列的开发也具有高度的弹性化，快速化。例如在产品开发初期，为了能快速推出占领市场，对开发进度的要求会远大于成本控制的要求。这时候，我们可以根据公司软件开发力量和硬件开发力量的对比，控制软件模块和硬件模块的选用，使开发工作同步快速进行，以期缩短开发周期；但随着产品产量的增加，我们可能很快就要面对成本的压力，这时候，可以把一些硬件模块转换成软件模块，从而降低成本。高度积木化和弹性化的设计，将大大降低移植的风险，加快移植的进度。

软硬件设计过程中应该互相配合，互为容错，以下几点一定要注意：

1、 硬件要容软件的错，在软件出错时，应尽可能减少硬件误动的可能。
2、 硬件要考虑软件死机，硬件不但要复位系统，而且要在复位期间尽可能保持系统安全，复位前，复位中，复位后都不能随便修改 IO。设计者还应确保不论软件做什么样的操作硬件都不应在短时间内发生永久性损坏。
3、 软件要容硬件出错，硬件出错时，软件应该使故障范围缩到最小，除出故障部分工作不正常以外，其他部分应该继续正确工作。绝对不能因部分硬件错误导致整机不工作。
4、 软件除了在模块之间要容错外，还应该有诊断硬件故障的能力，当硬件故障时，应记录并显示或通过通信送出故障状态。
5、 当确定是软件错误时，并不代表硬件设计者就完全没有责任了，应该检查硬件是否考虑了容错，同样，确定硬件故障时，也不代表软件工程师就可以高枕无忧了，也应该检查软件是否考虑了硬件容错。尤其要避免的是互相推诿扯皮，最后问题就挂在那谁也不理，等到产品大面积故障时再处理就晚了。

1.6 软硬件协作提高可靠性

1.6.1 硬件措施

- 在硬件设计中使重要的输出端子的状态可以回读，使软件可以做相应的容错检查。
- 假设软件可能存在严重的 Bugs，确保不论软件做什么样的操作硬件都不应在短时间内发生永久性损坏。
- 保证软件运行环境正确，寄存器、CPU 状态、内存以及固件数据不被意外修改。
- 软件出错时，尽可能降低硬件误动的可能，比如对一些重要的操作规定严格的时序或状态组合，使出错代码难于满足该时序或状态组合。重要的数据存储在串行操作的存储器里，如 I2C 总线，错误代码是很难满足 I2C 总线时序的。甚至还可以做双备份容错。
- 如果软件发生严重错误而死锁，要有复位计算机从而使软件恢复运行的机制，比如使用看门狗复位，硬件要保证复位期间系统安全，也不能随意改变系统的输出状态。

1.6.2 软件措施

软件措施虽然不能防止造成硬件永久性损坏的干扰，但只要处置得当，还是能防止很多不可恢复的故障（需复位解决）和可恢复的故障（造成短时间混乱）。要编写抗干扰能力强的软件，就要牢记“不确定性”四个字，一定不能认为端口、寄存器等的状态是确定的。在严重的干扰下，CPU、外设的寄存器可能会意外改变，这些改变可能会立即导致灾难性的后果，也可能不会立即表现出来，而且只要软件做适当的处理就能消除错误。

如同硬件上的抗干扰措施一样，软件抗干扰措施也是有代价的。通常，程序代码会更大一些，程序执行会慢一些，这意味着更多的存储器、更高的 CPU 速度，要权衡这些代价与使用硬件方法抗干扰的成本。

软件措施一般可以概括为：

定期检查状态

- 定期检查 IO 端子的配置和状态，使输入端子始终保持输入状态，而对于输出端子，则根据检查结果作出相应处理，如果输出端子所控制的设备是电平敏感的，则定期用正确的数据刷新输出端子的输出电平就可以了。如果输出端子所控制的设备是边沿（即电平的改变）敏感的，则不能简单地刷新状态了事，而是要根据设备的特性做出相应的处理。
- 上述 IO 端子包括 CPU 本身端子和扩展端子。
- 定期检查外设的状态，在出现异常时用正确数据重新初始化外设（包括 CPU 片内和片外扩展外设）。

陷阱

强烈的干扰或者是软件 bug 有可能使程序指针 PC 跑飞（即 PC 值不在正常代码范围以内），有一些 CPU 设置了非法指令地址捕获结构，当 PC 跑到非法程序地址时，就会发生中断，记下发生错误的指令地址，然后 PC 跳转到指令异常中断向量处。我们可以在非法地址中断处理程序中写入相应的处理程序，以降低干扰对系统的危害。使用非法指令地址异常设计陷阱的过程如下：

1、 正确设置 CPU，使合法地址为包含正常代码地址最小范围。

2、 编写指令异常中断处理程序，对错误做出相应的处理。

对于没有指令异常处理机构的CPU，处理起来就要困难一些，但还是有办法处理的，在空闲ROM地址处写入调用诊断函数的代码，出错处理函数的流程如 图 1-12：

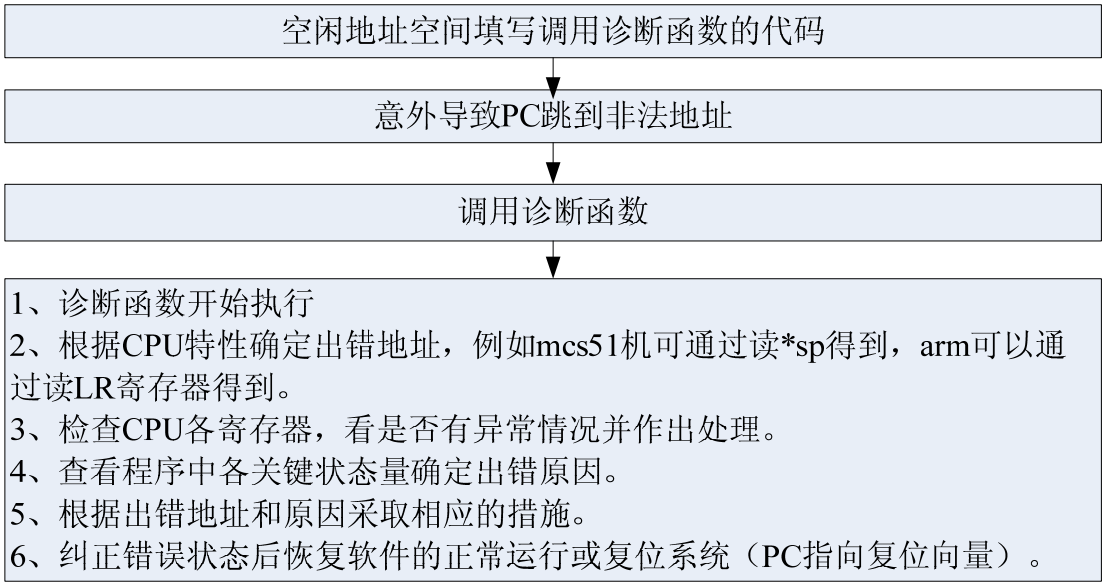


图 1-12 没有指令地址异常捕获功能时的陷阱设计

注意，在空闲地址空间要填写函数调用指令而不是跳转指令，因为使用跳转指令的话诊断程序无法确定是从什么地方跳过来的，而函数调用指令 CPU 会自动保存返回地址，有些 CPU 保存在栈里，有些 CPU 保存在特定寄存器里，这个返回地址对诊断函数正确诊断错误原因是非常重要的。

还要注意的是必须保证函数调用指令被正确解码，否则不能捕获跑飞的程序指针。对 RISC 处理器来说，直接填写调用指令就可以了，对于 CISC 处理器来说，函数调用指令往往是多字的，每条指令间就要填写若干条 nop 指令，以防止函数调用指令被中间截断，nop 指令的条数应该等于函数调用指令的字数。以最常用的 51 单片机为例，假定诊断函数的地址为 0x0075，从 0x1000 处开始是空闲地址，则从 0x1000 开始放置指令

Lcall ox0075

Lcall 指令的汇编码为 0x12，于是从 0x1000 开始的内存数据如下，

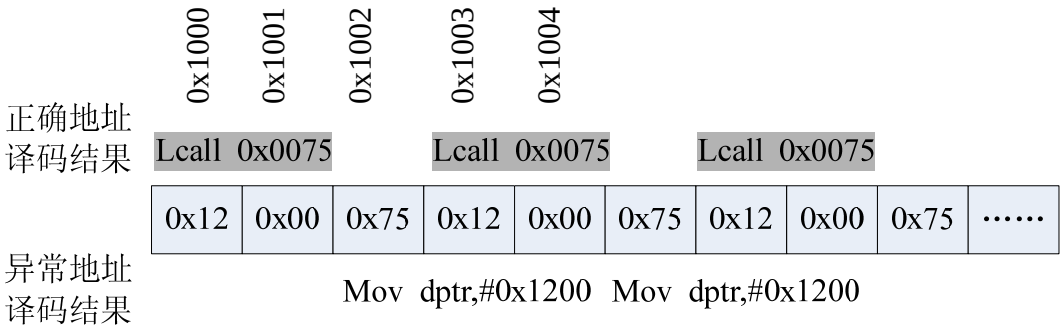


图 1-13 陷阱指令被错误译码

此时，如果在干扰条件下使 PC 跳到 0x1002 处，则 CPU 从 0x1002 处开始取指，不幸的是，0x75 刚好是指令 mov dptr,#data16 的汇编码，于是 CPU 便一直执行

mov dptr,#0x1200

直到内存终点，lcall 指令将不能得到执行，诊断函数自然也不起作用了，如果我们在每

条 lcall 指令后加入 2 条 nop 指令，则无论如何 lcall 指令都能执行到。

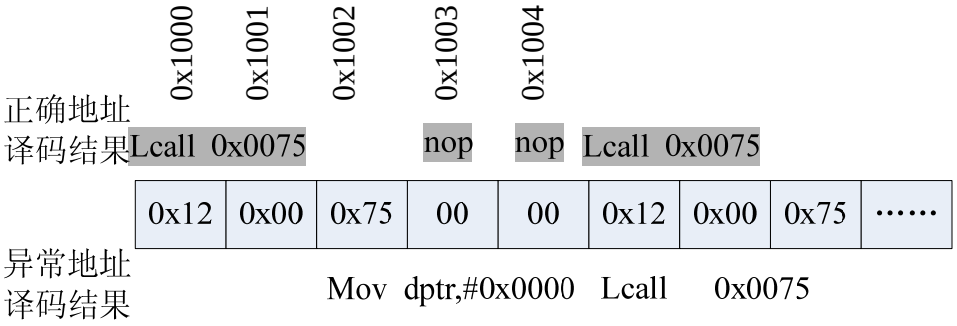


图 1-14 确保陷阱指令被正确执行

为了配合诊断函数的运行，软件应该在关键的地方留下脚印，诊断程序会根据这些脚印作出正确的诊断，记录这些信息也为事后的人工诊断提供信息，这对软件维护和消除 bug 都是至关重要的，许多系统都有一个日志文件，这个日志文件里就包含了许多诊断信息，嵌入式系统由于大多数时候都是在无人监控的条件下运行，而可靠性要求又那么高，因此保留诊断信息就显得特别重要。

1.7 可靠性分级设计

众所周知，功能相同的产品，如果可靠性要求不同，其成本和技术复杂度并不一样；同一个产品，也由多个模块共同构成，各个模块的可靠性要求也可能不一样。我们常说嵌入式系统的可靠性要求高，究竟高在哪里呢？是所有模块的要求都很高吗？显然不是，80/20 原则告诉我们，在绝大多数情况下，高可靠性产品，往往只有 20%的模块是要求高可靠性的，另外 80%的模块，可靠性降低一点是允许的。比如汽车电子系统，刹车控制部分的可靠性要求很高，必须做到万无一失，而汽车音响、车载 DVD 系统这些娱乐设施，其可靠性要求不会高于普通民用影音产品。按照功能的可靠性要求不同合理地分割模块，可以合理地分配有限的研发资源，降低整个系统的平均可靠性要求，从而加快研发速度，抢占市场；降低研发和生产成本，增加产品竞争力。

火星探路者号上的软件设计是一个很典型的可靠性分级设计的案例。探路者号刚上火星不久，就开始犯傻——开始无规律地重启，火星探路者号上使用 vxWorks 作为操作系统，WindRiver CTO David Wilner 先生在 IEEE 实时系统会议上分析说，由于优先级翻转导致巡航软件严重错误，正是这个错误导致了软件重启。火星探路者号的工程师们花费了极大的精力用于保证着陆软件部分的质量，使其几乎达到了完美的程度。因为如果着陆失败的话，整个火星探测计划就完全失败了，所以，完全可以理解工程师们为什么会对相比之下次重要的陆地巡航软件的小问题不那么在意，只是把简单的重启当作错误处理。问题分析清楚以后，工程师们用远程升级方法解决了上述问题。

1.7.1 防错还是容错

偏重防错的策略，该策略机制偏重于使用户容易编制无错的程序，以及出错后如何纠正错误使软件重回正确的轨道，一旦出现不可修正的错误可能导致严重的后果。

偏重容错的策略，该策略机制偏重于在应用程序出错的情况下隔离出错的程序，防止错误扩散，尽量保证系统的其他部分正确运行，只有出错模块自身不能正确地运行，或者得不

到所申请的服务。

在通用计算机的操作系统上，可能更加偏重于容错，因为桌面系统经常执行很多进程，由很多用户或者程序同时使用，一个不守规矩的程序出错时，其他用户和程序的安全运行全由操作系统来保证，操作系统自身必需足够健壮，在这样的系统中，操作系统自然是整个环境的核心模块，必需加以仔细保护，而与保护操作系统相比，用户程序反而显得没有那么重要了。当一个用户模块出现致命错误时，操作系统宁可杀死这个模块也不能让错误扩大而影响其他用户的安全，总之，安定团结的大局面是最重要的，对于少数破坏份子，就一个字：杀。所以，通用操作系统的出错处理可以比较简单，往往是保护自己第一，防止多用户间互相干扰第二，恢复出错的程序第三。

再看看嵌入式系统，虽然操作系统支持多任务，但往往只有一个用户一个程序在运行，这个程序虽然被分成多个线程，但其本质还是只有一个程序在运行，这个程序就是整个嵌入式系统的全部家当，如果这个程序被杀死，那么就算操作系统仍然健壮地运行，但只有操作系统在运行的计算机是毫无意义的。因此，嵌入式环境下编程，确保所有线程正确运行是最重要的，如果是不太重要的线程出错，可以把线程杀死再重新建立，甚至可以直接杀死；如果是核心线程出错，核心线程是无论如何都不能杀死的，就算杀死重新创建也不一定能保证正确运行，故核心线程出错时，即使操作系统本身运行正常，也会直接把整个系统重启。事实上，所有可靠性要求比较高的系统都配备了看门狗电路，看门狗由于守护核心线程正常工作，一旦核心线程出现异常，看门狗就会复位整个系统。如果核心线程出错，容错将变得毫无意义，不必等待看门狗，直接重新启动计算机才是最佳选择。因此，嵌入式操作系统比桌面系统在这方面有更高的要求，它不但要保持自身的健壮，而且处理用户程序 bug 的策略也比桌面系统更加复杂。

1.7.2 确定关键模块

关键模块是对产品质量有重大影响的模块，在系统设计阶段，应该把该产品的关键模块分解出来，并且采取特别的措施保证关键模块被正确地实现。有很多设计者有重技术轻应用的倾向，他们往往会根据技术实现的难度，把系统中各个模块进行排序，把最复杂最难实现的模块作为核心模块；许多时候，操作系统及相关模块会被当成核心模块，这在大多数情况下是正确的，因为操作系统崩溃的话将导致所有任务异常。这种分类方法是不对的，因为嵌入式系统是以开发产品为目标的，应该从使用者的角度，把实现产品目标最为关键的部分确定为关键模块。有些系统虽然由很多模块组成，但其中 1 个或少数几个模块起关键作用，这些模块可能比操作系统更加重要。这里介绍一种确定关键模块的 cehw 系数法，系统设计时，罗列出产品的所有功能模块（从使用角度），分别根据模块的特性确定各模块的 cehw 系数，各系数大于 0 小于 1，所有模块的同一个系数加起来等于 1。比如某产品可以分为 5 个模块，根据市场调查得出用户使用各模块的平均时间比例分别为 0.6，0.2，0.1，0.06，0.04，则各模块的使用频度 c 也分别就是上述系数，各模块的 c 系数的总和为 1。

- **使用频度 c**，经常使用的功能模块系数大，不经常使用的功能系数小。比如移动电话中实现通话功能的模块，c 系数就很大。
- **故障恶感度 e**，把用户对产品使用中出现的故障的反感程度排序，用户非常反感的故障 e 系数大，反之 e 系数小。还是移动电话的通话模块为例，如果正说到关键处，因手机故障断了线，你说有多扫兴，通话 e 系数自然很高。
- **好感度 h**，某功能模块正常使用能获得用户好评，该功能的微小改进能极大地增加用户的满意度，则调高本系数，反之降低本系数。比如在电话查号服务中，语音报电话号码时是一个一个数字蹦出来的，如果能实现连续流畅地报号，用户会感到很

亲切，好感度系数 h 就高。

- **危险度 w：**用于衡量一个模块正常与否与生命财产安全的关联程度，关联度高的系数也高，反之系数降低。比如汽车的刹车控制模块，w 系数将很大。

把某模块上述各系数相加即得到该模块的关键系数，系数越大说明该模块的关键程度越高，关键度特别高的模块应该确定为关键模块。就象修房子，房梁分主梁，次梁，墙面分承重墙和间隔墙，主梁和承重墙需要特别加固，嵌入式系统设计也是这样，关键模块的可靠性需要特别关注。从系统设计，到详细设计，到代码编写或者电路设计，再到测试都要进行重点照顾，这对提高产品的综合素质可以起到事半功倍的效果。

1.7.3 cehw 系数应用——电梯控制器设计

本节我们分别用两种方法模拟设计一种电梯控制器模型，这不是一个真实的产品设计，只是抽象出来的一个模型，用它只是为了阐明观点，可能与真实的电梯控制器相差很远。电梯控制器分轿厢内和楼层控制器以及总控制器，这里仅以轿厢内控制器为例。

我们略过规格和需求定义，直接进入系统阶段。一般来说，软件上，电梯控制器包含以下 5 个模块，分别是电梯运行调度模块、运行控制模块，轿厢门开关模块、按键处理模块、紧急制动模块。硬件也包含 5 个模块，分别是升降电机驱动模块、轿厢门开关驱动模块、按键模块、轿厢超速检测模块、紧急制动驱动模块。

1.7.3.1非 cehw 设计方案

每种电梯都有紧急制动功能，当发生意外事故时，轿厢超速下滑（如钢丝绳折断，轿顶滑轮脱离，牵引机蜗轮啮合失灵，电机下降转速过高等原因），或者乘员按紧急制动按钮时（有些电梯有这个按钮）。刹车装置就会立即启动，通过一系列电子或机械动作，使轿厢卡在导轨上缓慢下落。这个功能也许在电梯的生命周期内一次都不会遇到，但一旦发生事故，就是致命的。即使不使用 cehw 方法，人们也会想到，紧急制动功能是非常关键的模块，需要特别注意。

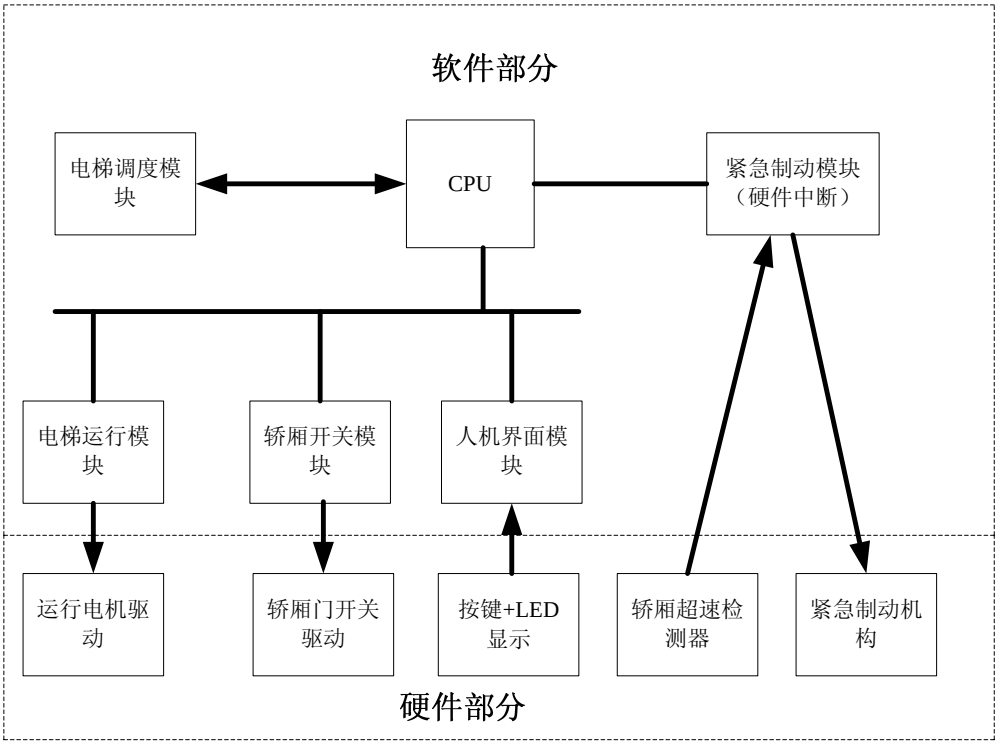


图 1-15 不使用 cehw 系数的电梯控制器方案

图 1-15 的控制器方案中，软件被分解为 5 个模块，各软件模块控制相应的硬件模块，其中除紧急制动模块外的其他模块在主程序中之行。考虑到轿厢超速检测及紧急制动模块的可靠性要求特别高，如果此模块出现错误导致紧急制动失败将直接关系到生命安全，所以本模块由硬件中断单独处理。这样，即使主程序出现故障处于异常状态，只要中断寄存器和代码没有被破坏，紧急制动机构还能正确动作。

我们来分析一下这个方案，方案中，显然已经考虑到了紧急制动模块的特殊性，并且做了相应的处理，使紧急制动模块的可靠性高于系统中其他软硬件模块，但是仍然存在下列问题，可能导致紧急制动失败：

1. 紧急制动功能由软件控制，该软件模块与其他软件模块在同一个 CPU 上运行，那么紧急制动模块就有受其他软件影响的可能，即使紧急制动模块本身没有 bug，但很难保证整个软件都没有 bug。就算紧急制动模块在单独的中断函数中实现，其他模块的 bug 一般不会影响到它，但是如果 bug 错误地修改了中断寄存器，或者改变了 CPU 的工作状态，那么紧急制动模块就可能会出错。
2. 工业计算系统一般都有看门狗电路，在看门狗复位其间发生的紧急制动中断将得不到响应。
3. CPU 系统的硬件可靠性问题也值得注意。CPU 由于内部结构复杂，往往是整个计算机系统的可靠性瓶颈。在外界干扰下，CPU 可能出现两种故障，一种是，在软件本身没有错误的情况下，外界干扰可能使 CPU 错误地执行指令，从而导致软件失效，在绝大多数情况下，如果看门狗电路和软件设计合理，将会由看门狗电路复位整个 CPU 系统从而恢复正常，但是在从软件失效到复位完成这段时间内，紧急制动模块将失效。另一种是，强烈的干扰导致硬件死锁。由于功耗问题，现代 CPU 大都采用 CMOS 电路，由于单向可控硅效应，CMOS 电路可能出现死锁的情况，这种状态下，要么直接烧毁 CPU，要么 CPU 进入一种既不执行程序，也不响应中断和复位，与软件死机对应，我们称这个状态为硬件死机。有一个简单的实验，可

以让我们切身感受一下干扰使 CPU 硬件发生故障的情况，用我们手头很容易找到的打火机，拆下其电火花芯，在 CPU 附近放几个火花，CPU 很容易就会死机复位，或者发生硬件死锁。软件死机是可以通过硬件看门狗复位来重新启动程序的，而硬件死锁只有断电冷启动一条路。电梯工作在工业环境，且直接控制电机，电机的火花有点象打火机的火花，是很严重的干扰源，电机起动和停止时也有很大的冲击电流，这都是潜在的严重干扰。紧急制动功能由软件控制，所以，它的可靠性不可能高于 CPU 系统硬件可靠性。

综上所述，由于 CPU 以及软件是紧急制动模块的一部分，我们只有把整个 CPU 硬件系统的可靠性提高到紧急制动模块的可靠性要求，才能满足紧急制动模块的可靠性需求。而且，其他软件模块的可靠性也随之提高，即使其他模块允许一定程度的 bug，但决不允许存在影响到紧急制动模块 bug。

1.7.3.2cehw 设计方案

表格 1-1 电梯控制器的 cehw 系数表

模块名	c 系数	e 系数	h 系数	w 系数	关键指数
电梯调度模块	0.25	0.25，如果调度错了楼层，多么尴尬	0.1	0	0.6
运行控制模块	0.25	0.3，电梯要是停在半空中，佛祖也会发火的	0.1	0	0.65
轿厢门开关模块	0.25	0.3	0.1	0.1	0.75
人机界面模块	0.25	0.15	0.1	0	0.5
紧急制动模块	忽略	忽略	0.6	0.9	1.5
备注	前 4 个功能每次坐电梯都需要使用，c 系数相同。	紧急制动功能平时根本就用不到，就算“正常值班”，用户也感觉不到	如果紧急制动有故障，即使没有造成伤害，也会造成严重的心里影响	轿厢门控制失灵可能会夹伤人，所以分配了一定的 w 系数	

在 cehw 系数表中，紧急制动模块的关键指数非常大，轿厢门开关模块次之，其他三个模块的关键指数是相同的。因此，紧急制动功能是电梯控制器的关键模块，它的关键系数是如此之高，让我们不得不用特别的手段保证其可靠性。

从软件程序员的角度来说，一个从设计到实现都完全没有缺陷的软件是足够可靠的（姑且不辩论这样的软件是否存在），然而，嵌入式系统是软硬件结合的产品，既然软件只是产品中的一个模块而已，那么，100%可靠的软件并不等于 100%可靠的产品，是的，任何产品的可靠性都不是由其中单个模块决定的！由于紧急制动模块的关键指数远高于其他四个模

块，如果该模块与其他模块一起用一个 CPU 控制，势必要求整个 CPU 系统都要达到紧急制动模块的可靠性要求。除了紧急制动软件模块的设计和实现过程要满足紧急制动模块的安全要求外，还要求在同一个 CPU 上运行的其他软件模块的可靠性达到很高要求，要求硬件保证 CPU 正确取指令和数据、保证 CPU 正确地执行指令。实际上，从纯技术角度讲，紧急制动模块是一个很小的模块，为了一个小模块的可靠性而把整个系统的可靠性要求升高数个等级，是否值得呢？如果把紧急制动模块与其他四个模块分离，独立实现，其他四个模块用 CPU 控制，则这个 CPU 系统的可靠性要求大大降低。降低可靠性要，伴随的是成本的大幅降低，开发时间也会大大缩短。紧急制动模块虽然可靠性要求非常高，但其功能实现方面其实非常简单，不使用 CPU，很容易直接用小规模硬件实现。这样的话，我们只要保证这一小块电路达到高可靠性就可以了。著名的 80/20 原则又一次得到了体现：分量只占很小一部分的紧急制动模块的可靠性决定了整个系统的可靠性。

其次，轿厢门开关模块也有较高的关键指数，它的关键指数较高是因为它比其他模块的危险系数 w 高一些，如果从机械结构上降低它的危险系数，那么控制电路部分的关键指数就与其他模块一致了。选择适当的轿厢门开关的电动机功率，使其在任何情况下都不能输出足以伤人的机械力，就可以达到控制危险性的目的，从这一点上讲，系统设计已经从软硬件结合发展到电子系统与机械控制结合。

至此，我们可以画出按 echw 系数设计的电梯控制器的系统结构图了。

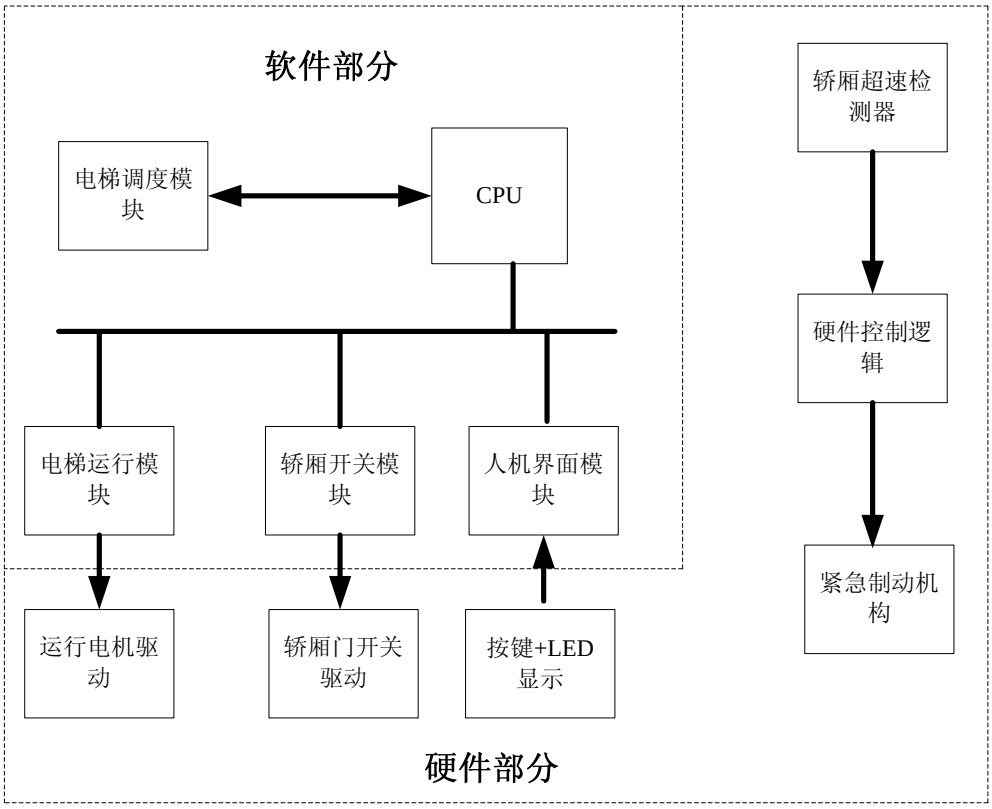


图 1-16 根据 echw 系数设计的电梯控制器

改进后的设计与原设计相比，仅仅把紧急制动的控制逻辑从 CPU 软件改为硬件，但是带来的好处是全方面的。

1. 增加了可靠性，小规模的电路本身比大规模的 CPU 电路具有较高的可靠性，对于性命悠关的功能，我们更倾向于使用小规模的电路直接实现。紧急制动模块是系统中可靠性要求最高的模块，它的可靠性就代表了整个控制器的可靠性，而用小规模

电路直接实现具有无可比拟的高可靠性。

2. 降低了成本，与非 echw 方案比，echw 方案增加了硬件，看起来似乎是增加了成本的，怎么会降低成本呢？这里顺便谈谈产品生产的成本因素，一个产品的成本应该包括生命周期的平均总成本，它包括物料成本、加工成本、生产调试成本，安装调试成本、营销成本、运行维护成本。更广义一点说，还包括由于产品质量问题导致品牌声望下降带来的无形资产损失的成本，更高的安全性将降低因紧急制动故障造成伤害乘员的可能性，而这种人身伤害对品牌的打击是致命的。电梯这种产品，注定是一个工程产品，他的产量不会很高，它对原材料的直接成本不敏感，但是安装和维护费用很高，对开发、调试、生产的成本更加敏感。而且电子控制部分的成本占材料成本的比例也很低，用硬件实现紧急制动控制逻辑增加的成本，几乎可以忽略不计。但是经这样修改，整个 CPU 系统的可靠性要求却降低了，这降低了 CPU 选型的要求，软硬件系统的测试费用也降低了。软件可靠性需求的降低还能大大缩短开发时间，加快产品研发进度。

1.7.4 echw 系数应用——手机通话模块

电梯控制器的特点是其中有一个高度危险性相关的模块，它的设计是围绕这个使用频度系数 c 非常低而危险性系数 w 非常高的模块展开的。我们再分析一个围绕使用频度展开设计的例子——移动电话。写到这里的时候，我想起了修手机的经历，到目前为止，笔者用过 5 款手机，其中有两款维修过，两者的故障很相似，一个手机是通话过程中会自动关机，另一个手机经常出现对方听不到我讲话而我能听到对方讲话。应该说明的是，这两个手机虽然不同品牌，但都出自全球性的大公司，牌子又硬又老。两次修手机的经历惊人地相似，保修中心的服务员态度很好，效率惊人，我还没有说完故障现象，她已经把维修单填好了，并且很有礼貌地递给我，说：

“一小时后来取机，给您带来不便，抱歉！”

“可是我还没有说完呢”我还想说下去。

“不用说了，我已经清楚了，谢谢”一样的笑容可掬。

毕竟是名牌服务，让你无可挑剔。低头看，在处理结果一栏赫然写着“软件升级”。维修中心环境很好，有沙发，还有报纸看，而且又没有什么事，于是就打算在里面看看报纸打发这一个小时的时间。当时保修中心还有几个要求修手机的，在等待的过程中，又有一些人进来。出于开发工程师的职业习惯，我就和待修手机的顾客以及店员聊天，了解手机的故障分布情况。故障原因中，有 1/2 是软件 bugs，升级软件即可，软件 bugs 中，有约 60％是通话问题，其次是电话本、短消息问题。这就奇怪了，一般来说，通话功能是一个不复杂的模块，与界面显示、游戏等模块的复杂度完全不是一个量级的，而且从手机问世起，通话功能就基本没有改变过，应该是经过了长时间使用验证的模块，而且是手机中最重要的功能，怎么会故障最多的功能模块呢？这些品牌又都有悠久的历史，应该积累了丰富的经验。俗话说，“不做不错，做多错多”，上述故障分布正好与用户的使用频度吻合，用户使用手机时，当然最主要的是打电话，其次是翻电话本，记录电话，发短消息。从开发工程师的角度，我们可以得出三个结论：

1. 故障涉及到多个型号与品牌手机，这些不同型号和品牌的手机的 bug 都集中在通话模块中是不可能的。而是因为通话功能经常使用，所以通话模块中 bug 的表现机会最多。
2. 手机软件中，虽然无法得知通话功能是否作为一个独立模块开发，但可以肯定的是，通话模块并没有作为核心模块受到特别的重视，而是与其他模块一视同仁。其可靠

性比其他模块高主要是因为这个模块比较简单。

3. 通话模块的故障会引起用户特别反感，而且记忆深刻，对品牌形象的影响非常大。

假设我们要设计一款手机，该手机由通话模块、界面模块、短消息模块、电话本模块以及 WEB 上网模块组成，我们用 echw 系数法分析一下各模块的重要性排队。

表格 1-2 手机的 echw 系数

模块名	c 系数	e 系数	h 系数	w 系数	关键指数
通话模块	0.5	0.5	0.8	忽略	1.8
界面模块	0.3	0.3	0.1	忽略	0.7
短消息模块	0.08	0.06	0.05	忽略	0.19
电话本模块	0.11	0.06	0.05	忽略	0.22
WEB 上网模块	0.01	0.08	0	忽略	0.09
备注		手机上网往往能给人意外惊喜。	打不了电话是多么恼火的事情。		

由于手机并没有多少危险性，故 w 系数被忽略。打电话是手机最重要的功能，故通话模块的关键指数远高于其他模块；除通话模块以外，其他模块都要依赖于界面模块，故界面模块也有较高的关键指数；电话号码本功能和短消息模块相差不大，大约并列第三；WEB 上网功能则属于锦上添花的功能。上述排序与人们日常使用电话的习惯是吻合的，想像一下，当你正在打一个重要的商务电话，谈到关键时候断了线，这时候你会发誓不再买这个牌子的手机，而如果是浏览器断了线，你只会抱怨网络不稳定，然后重新连接，过几天你就会忘记。因此，通话功能是手机的生命，而且通话模块很简单，与之相比，操作系统反而是次要的。因此，任何手机开发人员都不会不重视通话模块，否则，这款手机也不太可能与用户见面。通常会把通话模块当作最优先考虑的模块来开发，往往会安排最有经验的工程师执行通话模块的开发，而且在软硬件资源分配和测试时优先考虑通话模块。再者，正如前面所说的，通话模块自从 GSM 手机问世起就没有改变过，应该是最成熟且最久经考验的模块，其本身的 bug 应该会比其他模块少。那么为什么使用中会有那么高的故障率呢，可能的情况是：

1. 通话模块被当作重要模块设计，而且经过多年的验证，本身的 bug 是比较少的。事实上，手机通话模块的使用频度是超过 90%的，而表现出来的故障率只有 60%，也说明通话模块的可靠性确实有比较高的可靠性。
2. 通话模块虽然已经作为重要模块设计，但在软硬件系统上，并没有从系统的角度上使其具有特殊的位置，当操作系统出错时，或者其他模块出现严重错误时，通话模块就有可能被破坏。

通话模块是一个并不是很复杂但有特殊重要地位的模块，但其可靠性要求与上一节提到的电梯紧急制动模块相比，又要低得多，而且手机这种消费类产品，其硬件材料成本是相当敏感的，而且其节电与体积指标也很重要，决定了不可能添加独立的硬件来提高手机通话模块的可靠性，但在软件上，通过灵活的系统设计，我们可以把这个模块设计成在其他模块发生致命错误时，本模块能独善其身。中断就有这样的特性，如果把通话模块设计在独立于操作系统的中断里，那么，中断外的程序包括操作系统崩溃时，也只有很小的概率会破坏到中断寄存器，只要中断寄存器不被破坏，那么通话模块就可以确保无虞。图 1-15 所示的电梯设计方案中紧急制动模块的处理方式，其实可以用在手机的通话模块设计中。

第2章 认识内存

我们有时候会碰到这样的问题，程序在开始时一切正常，但运行数月后却出现错误，甚至整体崩溃，很有可能就是内存丢失或者碎片积累导致内存耗尽造成的。要开发高可靠性的嵌入式产品，就不得不深入的了解计算机系统的内存管理。由于嵌入式系统的特殊性，嵌入式系统的内存往往是非常紧凑而且有限的，不同的应用程序一般没有独立的地址空间，甚至连操作系统都没有独立的地址空间，所有的应用程序和操作系统一起挤在一个地址空间中运行，稍有不慎就会导致整个系统崩溃。而桌面系统不一样，桌面系统配置有很多物理内存且支持虚拟内存，且每个进程都有独立的地址空间，在 32 位机上，这个空间可达 2G 甚至更大，发生严重的内存碎片和内存泄漏时，程序仍然能正常运行，当程序退出时，泄漏的内存和碎片都将由操作系统统一收回，所以，桌面程序员几乎觉察不到内存限制的存在。随着嵌入式计算的日趋复杂和普及，对程序员的需求急剧增加，许多嵌入式程序员是从桌面系统转过来的，他们习惯了在桌面系统中自由自在的感觉。在嵌入式环境中，内存管理必需由程序员自己来完成，内存管理方面的 bug 是最容易发生，发生后最难定位和排除，很难通过测试和老化暴露 bug，甚至会在运行数月至数年以后才暴露出来，而且一旦发生就是致命的。因此，内存管理是最具挑战性的工作，本章讲述嵌入式系统内存使用的基础知识以及 djyos 系统的内存管理。

2.1 嵌入式系统的存储设备

嵌入式系统中的存储设备，按照其使用特点和可以分为以下几类：

1. 内存映射随机存储器 RAM，俗称内存，常见的有 SRAM，SDRAM 等设备，这些设备可以直接与系统总线相连，能对 CPU 的读写存储器的指令做出正确的反应。内存的访问速度最快，但断电后数据立即消失，上电后的数据是随机的。内存可以运行程序，但是运行程序之前必须先把代码和数据从 rom 中 copy 到内存中。可以用来存储变量、栈，可以动态分配（molloc）给用户使用，是嵌入式系统的内存管理的对象。
2. 设备式随机存储器，最常见的是包含在串行访问的 RTC（实时时钟芯片）内或者 MCU 监控芯片内的 RAM。这类存储器的内部存储阵列的组织也许与内存是一致的，但接口比较复杂，它不能直接挂在系统总线上，CPU 访问的时候，不能直接用访问存储器的指令进行读写，不能执行程序。这种存储器通常用在 MCU 没有外部总线接口而又需要扩展少量内存的应用，比如是利用 RTC 将你的 RAM 实现掉电保存。
3. cache，就是高速内存构成的一小块存储器，它的体积比主存小，但速度很快，可以直接与 CPU 的速度匹配。硬件以一定的算法把 CPU 早先访问过的数据暂存在 cache 中，CPU 再次访问相同的内存地址时，就直接访问 cache，减少等待慢速存储器的时间，提高整个系统的运行速度。cache 相对软件来说是透明的，也就是说，指令不能直接寻址 cache，也“看不见”cache 的存在。cache 的存在与当今电子制造工艺特点有关，CPU 运行速度可以做得很高，而且软件需要大量的内存，如果把所有内存的速度都制造成与 CPU 匹配的话，成本和功耗势必很高，体积也很大，

而一定数量的 cache 再配以高效的 cache 命中算法，系统的实际速度可以接近全高速内存的速度，而成本和功耗和体积则可以降低很多。

4. BROM（Bus ROM）是直接挂在 CPU 的总线上，CPU 可以像读内存一样读写的固态存储器，可以执行程序，掉电内容不丢失，但读 BROM 的速度一般会比读内存慢许多。BROM 分为可写类型和不可写入类型，flash 是典型的可写 BROM，工厂掩模的 ROM 则是典型的不可写入 BROM。即使是可写的 BROM，也不能像内存一样随机写入，而是要经过一系列复杂的操作才能实现写入，而且写入的速度也很慢。所以，BROM 里的数据不会被不经意的软件误操作改写，程序在 BROM 里执行会比在内存中安全一些，但缺乏灵活性，执行速度也可能会慢一些。有很多为嵌入式用途设计的 CPU，内嵌了一定数量的 flash 和 RAM，而且由于 ram 的数量很少，程序一般都不 copy 到 ram 而是直接在 flash 中运行。需要一些技巧来解决慢速存储器与快速 CPU 的矛盾，不同的芯片商有不同的做法，嵌入式系统设计师要仔细阅读相关资料。philips 的 LPC 系列处理器，使用 128 位的 flash 接口宽度配合指令预取；比如 Ti 的 TMS320F28x 系列 DSP 的 flash 访问速度仅为 36nS，只能把性能关键的代码和数据 copy 到 ram 中执行。
5. 设备式固态存储器，我们称之为 DROM（Device ROM），例如 nand flash，磁盘等。DROM 与 BROM 的不同之处在于 DROM 不是直接挂在 CPU 总线上，CPU 不能象读内存一样读 DROM，因此不能在 DROM 中执行程序。DROM 一般用于用少量的 IO 引脚扩展固态存储空间，比如 IIC 总线的 EEPROM，或者用有限的寻址空间扩展海量固态存储空间，比如 nand flash，只占用 2 个地址，但可以扩展几百兆的存储空间。

内存和BROM是所有计算机系统都有的，系统上电后，最初的一段代码必定在BROM中执行，有的嵌入式系统程序始终在BROM中执行，有些则在完成一系列初始化工作后，把代码copy到内存中，然后在内存中运行。而DROM和cache并不是所有嵌入式系统都有的，主频在数十Mhz以内的系统往往没有cache。也有一些CPU内部集成了nand flash硬件管理器，使其能够象访问BROM那样访问nand flash部分存储空间，例如三星的ARM9 系列的S3C2410，它可以象BROM一样访问nand flash的前 4K，从而可以不配备BROM。按照程序运行前和运行时代码和数据在存储器中保存的方式，表格 2-1 列出了嵌入式系统的几种常见的存储模式。由于嵌入式系统存储器配置多种多样，在此不能一一尽列，不尽之处，读者应根据自己的实际存储器配置灵活分析。

表格 2-1 嵌入式系统存储模式

代码全部在 BROM 中执行的模式						
特点：系统只配备了 BROM 和内存，且内存很小，不适宜用来执行代码。未配备 DROM（例如 IIC 总线的 flash），或 DROM 只作为存储数据。						
	初始化代码	其他代码	常数表格	初始化的全局变量	未初始化的的全局变量	局部变量
启动前	保存在 BROM 中的复位向量处	BROM 中	BROM 中	BROM 中	无	无
启动过程	执行代码	无操作	无操作	拷贝到内存中	内存区域清零	无操作
执行时	BROM 中，地址不变	BROM 中，地址	BROM 中，地址不变	内存中	内存中	内存（栈）中

		不变				
代码保存在 BROM 中，执行在内存中的模式（非重映射）						
特点：系统配备了 BROM 和内存，内存速度高且容量较大，足以用来执行代码。未配备 DROM（例如 IIC 总线的 flash），或 DROM 只作为存储运行结果用。						
	初始化代码	其他代码	常数表格	全局变量（初始化的）	变量（未初始化的）	局部变量
启动前	保存在 BROM 中的复位向量处	BROM 中	BROM 中	BROM 中	无	无
启动过程	执行代码	拷贝到内存	拷贝到内存	拷贝到内存中	内存区域清零	无操作
执行时	BROM 中，地址不变	内存中	内存中	内存中	内存中	内存（栈）中
代码保存在 BROM 中，执行在内存中的模式（重映射）						
特点：系统配备了 BROM 和内存，内存速度高且容量较大，足以用来执行代码。未配备 DROM（例如 IIC 总线的 flash），或 DROM 只作为存储运行结果用。						
	初始化代码	其他代码	常数表格	全局变量（初始化的）	变量（未初始化的）	局部变量
启动前	保存在 BROM 中的复位向量处	BROM 中	BROM 中	BROM 中	无	无
启动过程	执行代码	拷贝到内存	拷贝到内存	拷贝到内存中	内存区域清零	无操作
执行时	BROM 中，地址被重映射	内存中，重映射到执行地址	内存中	内存中	内存中	内存（栈）中
代码保存在 DROM 中（非重映射）						
特点：系统配备了 BROM、DROM 和内存，内存速度高且容量较大，足以用来执行代码。						
	初始化代码	其他代码	常数表格	全局变量（初始化的）	变量（未初始化的）	局部变量
启动前	保存在 BROM 中的复位向量处	DROM 中	BROM 中	BROM 中	无	无
启动过程	执行代码	拷贝到内存	拷贝到内存	拷贝到内存中	内存区域清零	无操作
执行时	BROM 中	内存中	内存中	内存中	内存中	内存（栈）中
代码保存在 DROM 中（重映射）						

特点：系统配备了 BROM、DROM 和内存，内存速度高且容量较大，足以用来执行代码。						
	初始化代码	其他代码	常数表格	全局变量（初始化的）	变量（未初始化的）	局部变量
启动前	保存在 BROM 中的复位向量处	DROM 中	BROM 中	BROM 中	无	无
启动过程	执行代码	拷贝到内存	拷贝到内存	拷贝到内存中	内存区域清零	无操作
执行时	BROM 中，地址被重映射	内存中，重映射到执行地址	内存中	内存中	内存中	内存（栈）中
存储在固态存储器（BROM 或 DROM）中，执行时动态加载到内存的模式						
特点：系统配备了固态存储器和内存，内存速度高但容量较小。这种情况在 DSP 应用中比较普遍，DSP 一般配备了容量小但速度非常高的内存。						
	初始化代码	其他代码	常数表格	全局变量（初始化的）	变量（未初始化的）	局部变量
启动前	保存在 BROM 中的复位向量处	固态存储器中	固态存储器中	固态存储器中	无	无
启动过程	执行代码	部分拷贝或不拷贝	拷贝到内存	拷贝到内存中	内存区域清零	无操作
执行时	BROM 中，地址被重映射	在固态存储器中，执行时动态拷贝到内存	在固态存储器中，执行时动态拷贝到内存	内存中	内存中	内存（栈）中
部分代码在高速内存、其他代码在普通内存中的模式						
特点：系统配备了少量高速内存和大量普通内存。高速内存只能容纳部分速度要求很高的程序，其他程序必须在普通内存中执行。						
	初始化代码	其他代码	常数表格	全局变量（初始化的）	变量（未初始化的）	局部变量
启动前	保存在 BROM 中的复位向量处	固态存储器中	固态存储器中	固态存储器中	无	无
启动过程	执行代码	高速代码拷贝到高速内存，其他拷贝到	按速度要求拷贝到高速内存或普通内	按速度要求拷贝到高速内存或普通内	内存区域清零	无操作

		普通内存	存。	存。		
执行时	BROM 中，	分别在高速和普通内存中	分别在高速和普通内存中	分别在高速和普通内存中	分别在高速和普通内存中	内存（栈）中
高速代码在内存、其他代码在 BROM 中的模式						
特点：系统配备了 BROM 和内存，内存速度很高，但容量小，内存只能容纳部分速度要求很高的程序，其他程序必须在 BROM 中执行。						
	初始化代码	其他代码	常数表格	全局变量（初始化的）	变量（未初始化的）	局部变量
启动前	保存在 BROM 中的复位向量处	BROM 中	BROM 中	BROM 中	无	无
启动过程	执行代码	高速代码拷贝到内存	高速访问部分拷贝到内存	拷贝到内存	内存区域清零	无操作
执行时	BROM 中，	分别在内存中和 BROM 中	分别在内存中和 BROM 中	内存	内存	内存（栈）中

2.2 存储器保护组件：MPU 简介

引子：非法指针

回顾一下，什么是指针？指针在内存中实际上是一个数，这个数的字宽与 CPU 的寻址空间匹配，在 32 位机上，这个数是 32 位的，在 16 位机上，这个数是 16 位的（也有例外，比如 keil c51）。但是这个数被赋予特殊的解释：表示变量或函数的地址。所以被形象地称为“指针”，因为它的的却象指针一样指向变量或者函数。在使用指针前，都必须先让它指向有意义的，并且允许由程序使用的实体——数据和代码。而所谓“非法指针”，就是指某个指针变量的值因故超出合法的范围，或者指向错误的地址，使其“枪口”乱指。程序逻辑错误、数组越界、堆栈溢出、指针未经初始化、非法赋值、多任务环境中的竞争，甚至是恶意地破坏等，都可以制造出非法指针。如果使用非法指针去读取或修改内存，则被读取或修改的位置是不可预料的。前者导致读回来的都是乱掉的数据，后者则会破坏未知用途的数据。这常常导致系统发生莫名其妙的功能紊乱，严重时会使系统毫无征兆，没有理由地失控、死机。非法指针就像大锅饭里的老鼠屎：一个非法指针就足以毁掉整个系统，而且极其隐蔽，很难通过症状来找出是哪里存在非法指针，甚至都不能判定症状是否因非法指针造成（程序大了其它 bug 也很多，并且也能导致相同的症状）。对于通常的单片机系统，如果没有硬件帮助保护存储器，是没有任何办法来防止非法指针的破坏的，完全靠程序员的素质和自律。但智者千虑，必有一失。尤其是当程序规模变得很大时，复杂度会呈指数上升，千头万绪纠缠不清，就算是谨慎如诸葛，聪明如司马，也不敢保证没有漏网之鱼。软件界有句名言“没有 bug 的软件是不存在的”，而指针（含数组下标越界）正是孳生 bug 的温床。

嵌入式系统常用于性命攸关的场合，我们称这种系统为关键使命系统。他们必须连续无故障地工作，比如，火车调度系统、生命维持系统、大型炼钢锅炉控制器、宇宙飞船控制器等。关键使命系统如果出现故障，将导致灾难性的后果，惨重的经济损失，甚至会使无数人死于非命。因此，决不允许这类系统因非法指针而导致系统紊乱。系统的可靠性将完全托付于开发人员，然而，这些系统的复杂度往往都非常高，几乎不可能由开发人员保证这种可靠性，防御性的系统设计方法应该承认程序员会出错，或者会遇到恶性破坏。因此，需要在硬件水平上加入一个防御性措施，当由于软件缺陷导致访问非法地址时，则强制改变执行流和处理器的工作状态，以便可以由防御软件做一些善后处理。这样，就可以为不同的程序限定一个内存使用范围，从而驳回因非法指针导致的对不允许访问的区域的访问。存储器保护单元（MPU）就是这种防御性的硬件措施，而事实上，目前配备 MPU 的 CPU 很少，但 MPU 的功能做为 MMU 的一个子集，弄清楚 MPU 对就容易理解下一节讲到的 MMU。

MPU 是这样的一种硬件：

1. MPU 需要配合 CPU 的不同状态才能发挥最大作用，简单的 MCU 只有一种运行状态，所有程序对所有资源有相同的访问权限。而复杂一些的 CPU 则提供特权状态和用户状态，特权状态用于执行操作系统，对系统资源有更高的访问权限。
2. 可以把 CPU 的存储空间划分为若干个区域，每个片区可以分别规定 CPU 在特权态和用户态下的访问权限，这些权限包括禁止访问、只读、可读写、可执行等，进一步地，还可以设置是否可以被 cache 缓存等。
3. 划分片区的方法可能有多组，每组可以独立配置访问权限。
4. MPU 配置只能由特权模式下的代码（一般是操作系统）来执行，应用程序只能被动接受操作系统给它安排的访问权限。

图 2-1 是一个典型的MPU原理模型以及典型的存储器分区设置，MPU的核心功能有 3 个：一是防止一个任务的代码和数据被其他任务修改破坏；二是防止用户任务修改破坏操作系统的代码和数据，三是防止程序访问非法区域，比如没有配备存储器的区域。MPU的次要功能包括：与cache和带缓冲的存储设备相配合，对专用存储区设置合适的cache和缓冲属性，比如DMA区域可以设为非cache区，阻止设备I/O区域被缓冲。

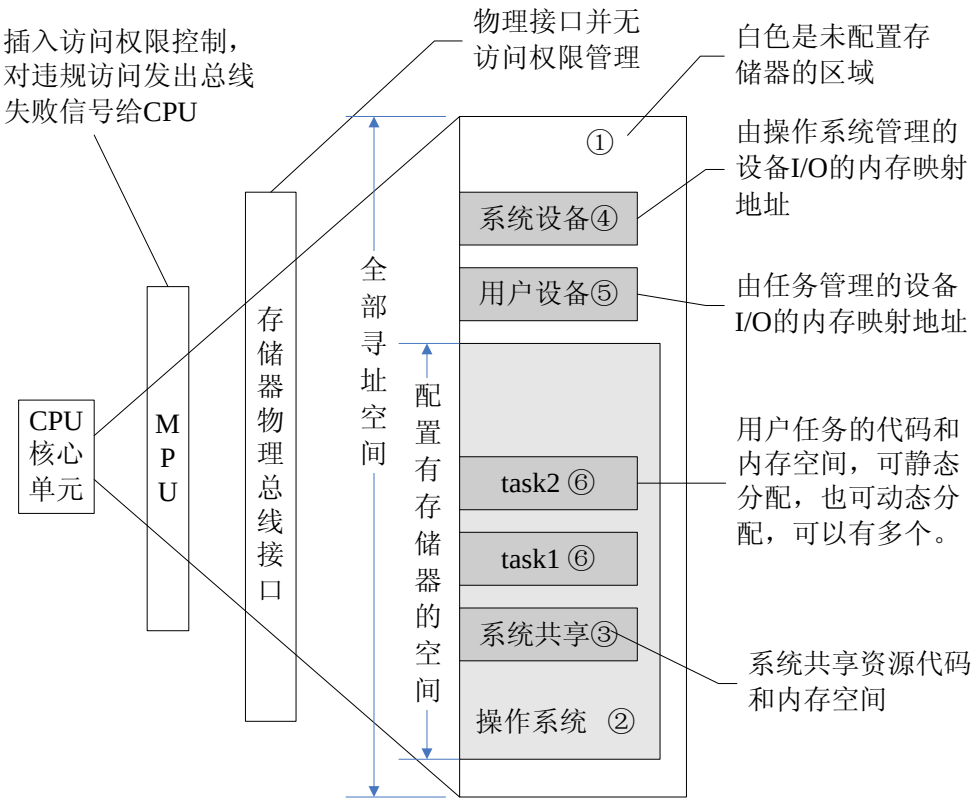


图 2-1 MPU 示意图

以下配合图中标号说明 图 2-1：

1. 因为 MPU 没有虚拟地址功能，没有配置物理存储器的区域是不可访问的，CPU 访问内存和内存映射 I/O 时，只能使用他们的物理地址。
2. 图中内存被分为 6 类片区，圆圈内的编号是各区的优先级，数越大优先级越高。
3. 各存储片区可以独立设置访问属性（通过配置 MPU），片区如果重叠，则在重叠区域优先级高的片区生效。
4. 各片区的访问属性如下：①区无条件禁止访问；②区则区别对待，如果 CPU 工作在特权级，则有完全的访问权限，如果工作在用户级，则禁止访问；③区是用户级和特权级都可以访问的区域；④区的访问权限与②区相同，但是不允许 cache（如果存在）和写缓冲（如果存在）；⑤区和③区的访问权限相同，但是不允许 cache（如果存在）和写缓冲（如果存在）；⑥区和③区的访问权限相同。
5. ⑥区的设置是实现用户级保护和多任务间保护的关键，该区的起始地址和区尺寸作为任务上下文的一部分，任务上下文切换时一并却换。当 task1 执行时，⑥区的起始地址和尺寸设置为 task1 所占据的存储区域，task2 所在的存储区是②区的一部分，task1 对它没有访问权限，task1 能访问的区域是③⑤⑥区。从 task1 上下文切换到 task2 时，⑥区的起始地址和尺寸就被设置为 task2 所在的区域。
6. 如果软件违反上述规则访问内存，比如企图访问①区，或者 task1 企图访问②区，则 MPU 将向 CPU 发出异常信号，中断当前软件执行流程，进行异常处理。

2.3 存储器多重映射组件：MMU 简介

与传统的、由 mcu 直接访问存储器的方式相比，在 MPU 的保护下，可以大大提高软件的可靠性，但它也有明显的缺陷。由于不能做地址变换，软件按照物理地址访问存储器，所

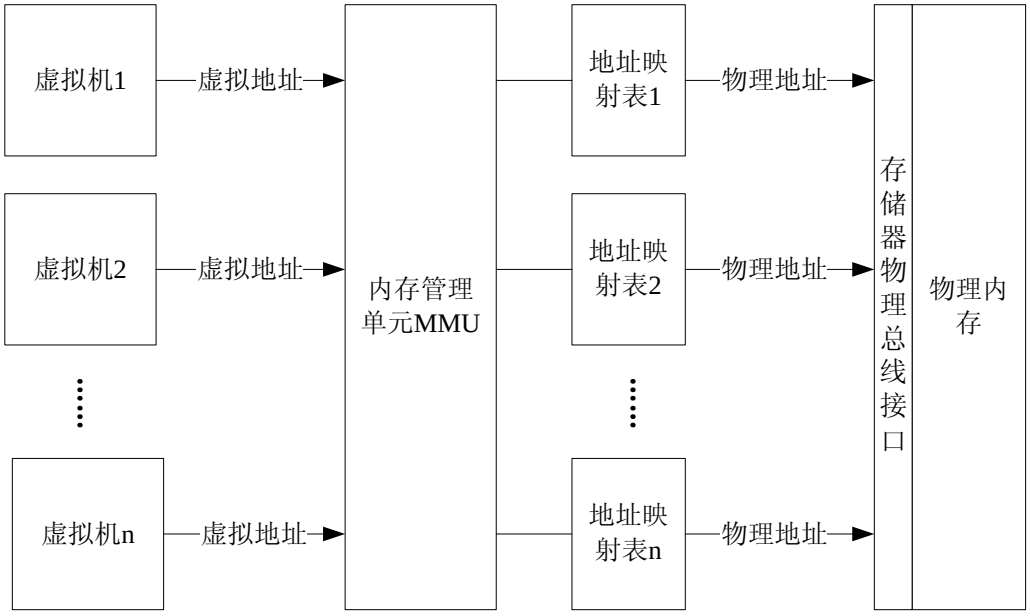
有程序只能共用一个平面内存空间，使任务和操作系统均不能拥有独立的寻址空间。因此，“存储器管理单元”（MMU）应运而生，它对存储器的管理更进一步，做到可以对地址执行变换的程度，在 MMU 的支持下，程序使用的地址未必是真实的存储器地址。在 MMU 支持下，不会因内存碎片而浪费内存，大大增强了软件抗内存碎片的能力，使应用程序拥有方便舒适的地址空间，从而使程序规模可以扩大甚至数百倍。与 MMU 相比，MPU 显得高不成低不就，功能上既没有 MMU 全面，系统又比传统直接访问的方式复杂得多，在 MMU 的挤压下，也就成了概念性的东西，极少有实际芯片加入了 MPU 组件。

MMU常用与“制造”虚拟机，虚拟机拥有独立的寻址空间，意味着在不同虚拟机上执行的程序的变量可以使用相同的地址而不互相影响，在MMU硬件支持下，不同的虚拟机内地址相同的变量，对应不同的物理存储单元。虚拟机的地址空间被称为虚拟空间，物理计算机的寻址空间被称为物理空间，存储器的真实地址称为物理地址。MMU实现虚拟寻址空间到物理寻址空间变换，或者说实现虚拟地址到物理地址的映射。如 图 2-2 所示，每个虚拟机拥有一个地址映射表，这个表由执行操作系统创建并维护。虚拟机是分时使用CPU的，操作系统在做上下文切换时也包括地址映射表的切换。MMU的地址变换过程如下：

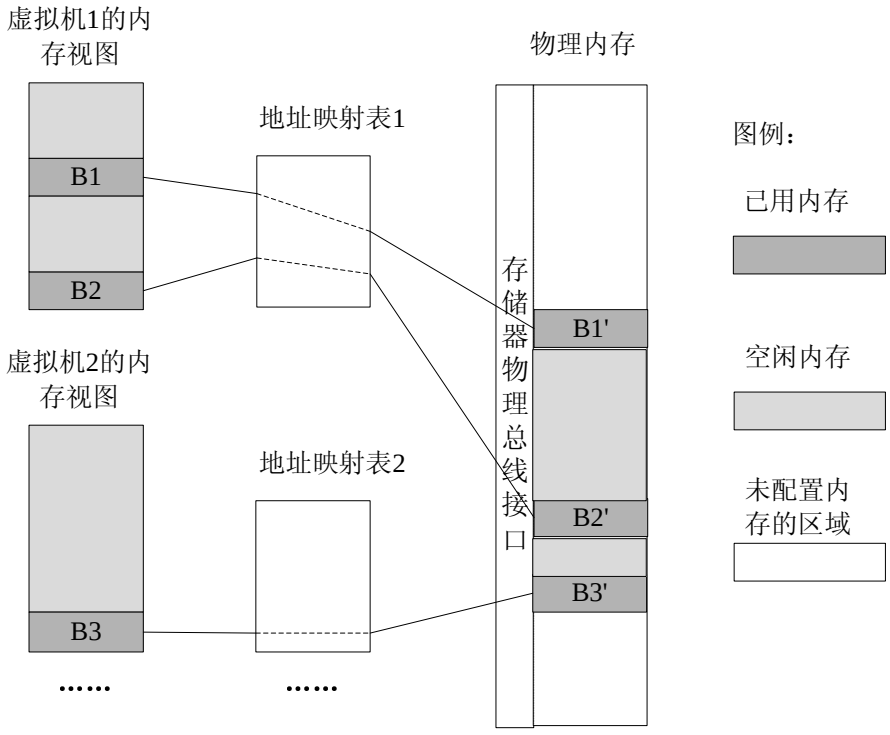
1. 虚拟机访问存储器，发出总线请求并输出虚拟地址。
2. MMU 收到虚拟地址后，查寻地址映射表确定这个虚拟地址对应的物理地址。
3. 把物理地址输出到地址总线上，完成 CPU 对存储器的访问。

使用MMU后，虚拟机可以访问的地址空间范围通常会比系统的物理内存大许多，在 32 位CPU上，虚拟机的地址空间可以达到 4G，但由于操作系统需要为各虚拟机提供一些公共服务和资源，这些资源必须所有虚拟机可见，且需要统一的地址，所以操作系统需要占用公共地址空间，从而限制虚拟机的地址空间范围。比如Linux系统下虚拟机的实际地址空间为 3G，djyos系统下进程地址空间为 2G。即使如此，一般的计算机尤其是嵌入式计算机也不会实际配置多达 3G内存。这里我们要搞清楚一个概念，就是寻址空间和实际拥有的物理内存的关系，当一个进程虚拟机刚创建时，操作系统会为其创建一个地址映射表，它就拥有大小为 2G的独立的寻址空间，但是它的“地址映射表”是空的，不占用一个字节的物理内存。直到这个虚拟机需要访问内存时，操作系统才会看到：哦，这家伙一穷二白呢，于是从物理内存中割一块内存给它。然后在该虚拟机的内存映射表中添加一个条目，把刚割下的内存的物理地址与进程所访问的地址对应起来。这个过程对虚拟机来说是透明的，虚拟机看到的是这样的假象：好像自己访问的地址上真的配备了存储器似的，而刚割下的那块存储器，就是虚拟机实际消耗的内存。无论虚拟机访问自己寻址空间内的哪一个地址，操作系统都用这种方法瞒天过海，所以，虚拟机在表面上看来拥有容量达 2G的存储器，这就是寻址空间的意思。而事实上，单个虚拟机实际上使用到的内存又会远小于系统物理内存，这是划分虚拟机的物质基础，显而易见，如果某一个虚拟机耗尽了计算机的物理内存，那么其他虚拟机继续存在是毫无必要的，没有内存，虚拟机什么也干不了。图 2-2(b)中，两个虚拟机使用了相同地址的虚拟内存，这是允许的，因为地址映射表将把他们映射到不同的物理地址上，通过 MMU，他们访问的的确是不同的物理地址。所以，只要各虚拟机（含执行操作系统内核的虚拟机）总共所需要的内存不超过系统的物理内存，虚拟机就可以独立运行。在桌面系统中，还使用内存交换技术，用辅助存储器（比如硬盘）来模拟物理内存，使计算机可以使用的物理内存也大大扩展，只是这个技术一般不会用在实时嵌入式系统中，但可能会用在一些非实时嵌入式系统中，比如PDA或者其他个人娱乐设备等。

MMU 除提供地址映射外，还提供存储器保护功能，CPU 可以通过 MMU 获得多个独立的寻址空间，每个寻址空间的访问控制能力都是 MPU 的超集，所有 MPU 能够提供的访问控制功能，均可以在任意寻址空间实现。



(a)



(b)

图 2-2 内存管理单元

虚拟机 1 和虚拟机 2 访问内存的过程如下：

1. 虚拟机 1 的 CPU 发出读/写地址 A 的指令，地址 A 在内存块 B2 中。
2. 该指令被 MMU 截获，MMU 查内存映射表，确定被寻址的内存单元的物理地址在物理内存的 B2’内存块中，地址为 A’。
3. MMU 把读/写 A’地址的指令发送到物理存储器总线接口单元，如果是读，把读的结果返回给 CPU。而 CPU 则仍然认为被操作的是地址 A。
4. 同样，虚拟机 2 也可以访问地址 A，但在 MMU 的安排下，实际访问的物理地址在

B3’内存块中，不会与虚拟机 1 相冲突。

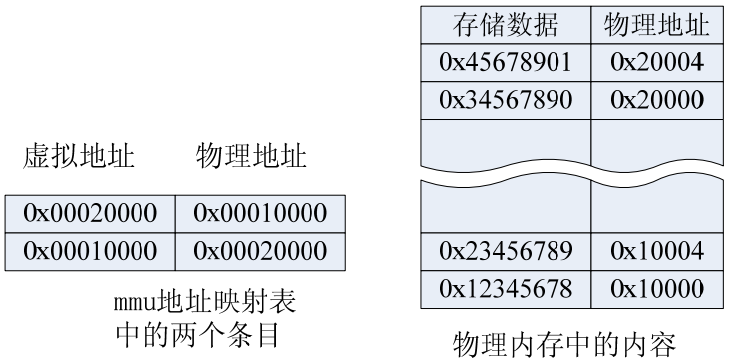


图 2-3mmu 内存映射

存储器中的内容如 图 2-3 所示，一ARM7 为例，执行如下CPU指令序列后，

```
mov     r0,#0x10000
ld      r0,[r0]
mov     r1,#0x20000
ld      r1,[r1]
```

在没有 MMU 的 CPU 上，或者没有启用 MMU 的情况下，执行结果是：r0=0x12345678,r1=0x34567890。

在启用 MMU 的情况下，执行结果是: r0=0x34567890,r1=0x12345678。

2.4 栈

在进一步讲述之前，我们还是先交代一下什么是堆，什么是栈，他们在使用中有什么区别。堆和栈是有严格区别的，应用程序、编译器、操作系统在管理堆和栈方面完全不一样。我们常说的堆栈，其实指的是栈，堆和栈的主要联系在于，在多线程环境下，线程的栈是从堆中分配的。在本书中，将严格区分堆和栈，不使用容易混淆的“堆栈”一词来描写内存，正确地使用堆和栈是一件非常有趣而又极具挑战的事，是影响软件质量的关键因素之一。

栈道，小道也，只容一人出入，绝无避让的余地，计算机中函数调用或者发生中断时保存局部变量和函数参数的地方称为栈，是再贴切不过的了。栈是一块 LIFO（后进先出）的存储区域，所有 CPU 都有一个栈指针寄存器 SP，栈按生长方向分向上生长和向下生长两种，按 SP 指针位置可分为满栈和空栈两种。满栈的 SP 总是指向最后分配的地址，空栈 SP 指向下一次分配的地址。向下生长栈的栈顶在栈的最高地址，栈底在最低地址，新数据入栈后栈指针减量，向上生长则反之，本书以向下生长的栈满栈为例进行叙述。栈由编译器决定如何使用，除非用嵌入汇编语言的方式，程序员是不能直接读写栈中数据的，也不能决定数据存在栈中什么位置。当然，有一些特殊技巧可以在特定编译器下做到用 C 语言读取栈中数据，这不属本书讨论的范围。如果你要编写 C 与汇编混合调用的程序，则必须了解所用的 C 编译器是如何使用栈的。那么，编译器又是如何利用栈的呢，哪些数据会在栈中保存呢？

1. 保存返回地址，当调用函数时，编译器把返回地址保存在栈中，使函数能返回到调用点的下一条指令处。
2. 如果函数有参数，编译器首先会使用寄存器传递参数，当寄存器不够时，就会用栈传递参数。arm 的 atpcs（子程序调用基本规则）规定，当参数小于 4 个时，用寄存器传递，超过 4 个的部分用栈传递。函数返回时，返回数据遵循同样的规则。

3. 分配局部变量，除静态局部变量外（静态变量以全局变量的方式分配内存），函数中用到的所有局部变量都从栈中分配内存。

代码 2-1 是一个用栈传递参数和分配局部变量的例子，函数exam_stack使用了 6 个参数，并且定义了 512 字节的数组，我们来通过查看调用函数和函数内部处的反汇编代码，来解释栈的使用过程。

代码 2-1 使用栈传递参数和分配局部变量

```
void exam_stack(uint32_t a,uint32_t b,uint32_t c,uint32_t d,uint32_t e,uint32_t f)
{
    uint32_t  ar[128];
    ar[0]=e;
    ar[1]=f;
    return;
}
```

代码 2-2 是调用该函数处代码的反汇编，编译器使用了 4 个寄存器保存参数，参数e和f则从栈中传递，他们分别被保存在当前栈指针（ARM中，r13 被用做栈指针）所在位置。大家可能已经注意到，在压栈过程中，不是每次压栈都移动栈SP指针，而是在压栈之前直接把指针移到栈目的地址，这与arm的指令系统有关，无论是压栈还是出栈，arm可以象访问普通内存一样访问栈，栈指针就象普通基址指针一样，可以用“栈指针＋偏移量”的方式访问参数，不一定要实时移动栈指针。有些cpu访问栈必须用专门的指令，例如mcs51 和x86 系列，当参数入栈时，就会一个一个字地移动栈指针。当参数保存完后，就跳转到函数入口地址。

代码 2-2：调用 exam_stack 函数处的反汇编结果

```
    0C000A54 E1A0C00D   MOV       r12,r13
    0C000A58 E92DD800   STMFD     r13!,{r11,r12,r14,pc}
    0C000A5C E24CB004   SUB       r11,r12,#4
    0C000A60 E24DD014   SUB       r13,r13,#0x14     //栈指针被移动到栈底地址
    0C000A40 E3A03005   MOV       r3,#5
    0C000A44 E58D3000   STR       r3,[r13,#0]   //参数 e 入栈
    0C000A48 E3A03006   MOV       r3,#6
    0C000A4C E58D3004   STR       r3,[r13,#4]   //参数 f 入栈
    0C000A50 E3A00001   MOV       r0,#1    //用寄存器 r0 传递 a
    0C000A54 E3A01002   MOV       r1,#2    //用寄存器 r1 传递 b
    0C000A58 E3A02003   MOV       r2,#3    //用寄存器 r2 传递 c
    0C000A5C E3A03004   MOV       r3,#4    //用寄存器 r3 传递 d
0C000A60 EBFFFFE8   BL          exam_stack    //调用函数
```

代码 2-3 是函数的反汇编代码，由于没有使用优化选项，反汇编代码有些繁琐。第 2 行保存寄存器和返回地址等，第 5 行移动栈指针，实际上就是在栈中为数组ar[128]分配内存空间，在函数中，参数也被当作局部变量看待，所以也同时为a、b、c、d四个参数分配了栈空间。ar[128]共需 512 字节，四个uint32_t型的参数需要 16 字节，加起来总共 528 字节（即 0x210），而参数e和f在函数调用前已经分配好了。当函数返回时，用保存起来的sp值恢复sp，前面分配的内存也就不复存在，被分配的栈空间内存被收回，当下次发生函数调用时，这块内存又可以被重复使用。整个内存分配和释放的过程都由编译器自动控制，应用程序无法干预，当函数执行时，内存就已经分配好了，就像静态分配一样。

代码 2-3 代码 2-1 的反汇编结果

```
1    exam_stack:
2    0C000A08 E1A0C00D    MOV          r12,r13
3    0C000A0C E92DD800    STMFD        r13!,{r11,r12,r14,pc}
4    0C000A10 E24CB004    SUB          r11,r12,#4
5    0C000A14 E24DDE21    SUB          r13,r13,#0x210  //移动栈指针，为数组分配内存
6    0C000A18 E50B0210    STR          r0,[r11,#-0x210]
7    0C000A1C E50B1214    STR          r1,[r11,#-0x214]
8    0C000A20 E50B2218    STR          r2,[r11,#-0x218]
9    0C000A24 E50B321C    STR          r3,[r11,#-0x21c]
10     ar[0]=e;
11     0C000A28 E59B3004    LDR          r3,[r11,#4]
12     0C000A2C E50B320C    STR          r3,[r11,#-0x20c]
13     ar[1]=f;
14     0C000A30 E59B3008    LDR          r3,[r11,#8]
15     0C000A34 E50B3208    STR          r3,[r11,#-0x208]
16     }
17     0C000A38 E24BD00C    SUB          r13,r11,#0xc
18     0C000A3C  E89DA800    LDMIA        r13,{r11,r13,pc}//恢复包含 sp 在内的寄存器，//隐含地收回了分配的内存。
```

代码 2-4 演示了反复调用函数过程中栈内存被重复使用的情况，图 2-4 则用图形化显示了函数反复调用过程中栈空间的使用过程。

代码 2-4　栈内存复用示意代码

```
void f1(void)
{
    //本函数需要 4k 空间用于存储局部变量和返回地址
    f2 ( );          // 2
    f3 ( );          // 4
}                    // 6
void f2(void)
{
    //本函数需要 8k 空间用于存储局部变量和返回地址
}                    // 3
void f3(void)
{
    //本函数需要 6k 空间用于存储局部变量和返回地址
}                    // 5
void main(void)
{
    f1 ( );          // 1
}
```

1. 调用 f1 后，系统在栈中为其分配 4K 字节内存，栈指针下移 4K。
2. f1 调用 f2 后，系统在栈中为其分配 8K 字节内存，栈指针再下移 8K。
3. f2 返回后，早先分配的 8K 字节内存将被收回，栈指针上移 8K。
4. f1 随后调用 f3，刚才收回的 8K 字节内存重新被利用，在上面分配了 6K 字节内存

给 f3，栈指针再次下移 6K。

5. f3 返回后，早先分配的 6K 字节内存将被收回，栈指针上移 6K.
6. f1 返回后，4K 字节内存被收回，栈指针恢复初始状态。

图 2-4 显示，随着函数的调用和返回，栈中的内存被重复使用，这个重复使用的过程是由编译器控制的，对应用程序是透明的。程序员必需为栈预先分配足够的空间，嵌入式程序员还要考虑中断问题，中断发生时，也需要使用堆栈。有些cpu有独立的中断栈空间，比如ARM，ARM支持 5 种中断源，各个中断源都有独立的栈指针。分别是软件中断SWI，栈指针R13_swi，中止模式中断Abort，栈指针R13_abt，未定义指令中断Undef，栈指针R13_und，外部普通中断IRQ，栈指针R13_irq，外部快速中断FIQ，栈指针R13_fiq。对于这种cpu，各种中断配有独立的栈空间，当中断发生时，中断处理函数被调用，所需要的栈空间从相应的栈中分配。另外一些cpu没有独立的栈空间，中断函数使用被中断的应用程序的栈，在分析栈溢出问题时，还要考虑中断栈开销，尤其是允许中断嵌套时。如果函数嵌套调用无休止地进行，比如递归调用层数过多，使栈的使用量超过预先分配的栈容量，将发生栈溢出，这是非常危险的。

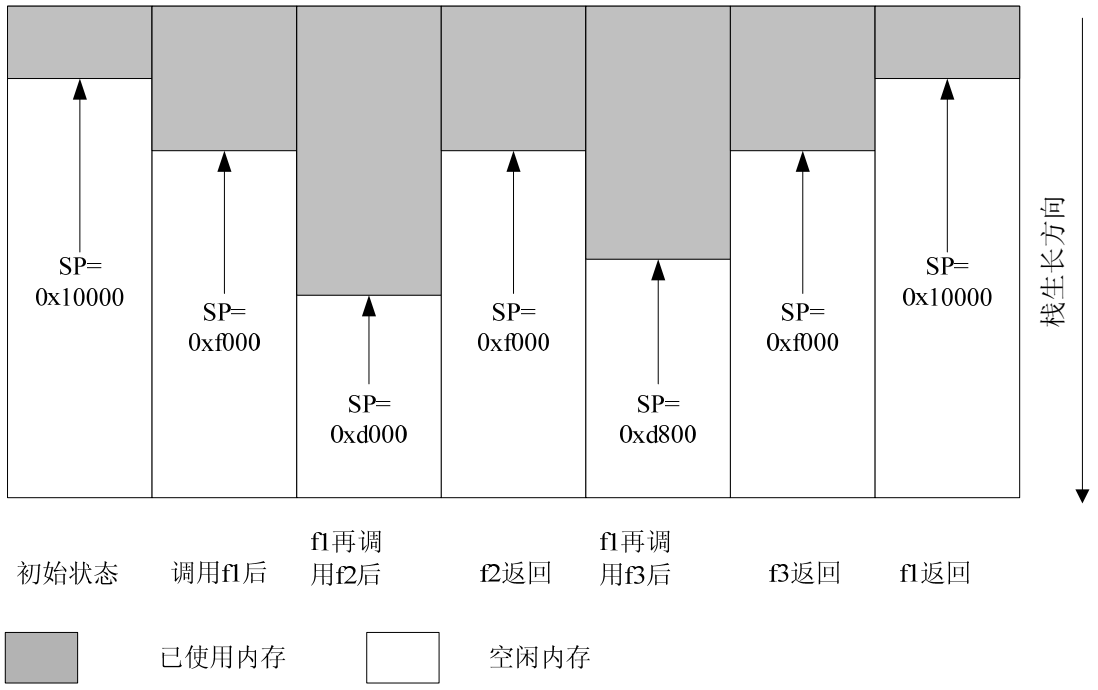


图 2-4 栈使用示意图

虽然应用程序无法直接干预栈中内存分配和释放的过程，但是，编译器使用栈内存时是根据SP寄存器的当前值作为基址指针进行分配的，应用程序可以通过改变SP值来间接改变栈的起始地址。事实上，这就是操作系统实现多线程的硬件基础。在嵌入式多任务环境下，操作系统为每个任务建立了独立的栈，否则，各任务的局部数据就会互相覆盖而使任务无法运行，每个任务栈的结构都与 图 2-4 相同。各任务栈以分时复用的方式共享一个SP指针，线程切换时，每个线程的SP指针被作为线程上下文加以保护。在建立线程时，程序员必须告诉操作系统新线程执行需要的栈空间尺寸，操作系统根据程序员的指示，再加上操作系统必要的栈开销，得到新任务实际栈需求，然后按照一定的策略（参见 2.5 节）从堆中分配一块适合的内存，作为新任务的栈。在发生多次任务建立和删除操作以后，整个系统的栈结构将变成 图 2-5 所示。

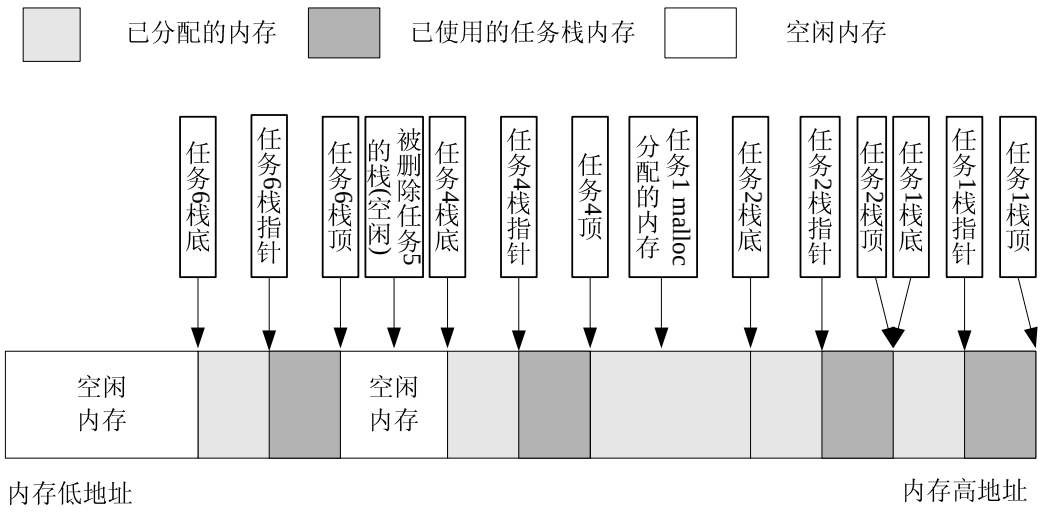


图 2-5 多任务环境下的栈结构

各任务的栈空间不能互相重叠，从 图 2-4 可以看出，当发生函数调用时，编译器从SP的当前值分配局部内存，无论是编译器还是操作系统都无从得知栈是否溢出（SP小于栈底地址），如果发生栈溢出，就可能破坏下一个线程的栈中的数据，或者破坏动态分配的内存，这就是臭名昭著的栈溢出错误，将出现无法预料的结果。当然，有些编译器或者操作系统会提供一些工具，让程序员在开发调试时能够观察栈的使用情况，及时发现并纠正潜在的栈溢出错误。确定一个线程需要多少栈空间的方法见第 2.8.2 节。

2.5 堆

可用于随机动态分配的内存的集合叫堆，栈虽然也可以动态分配，但它的分配过程完全取决于编译器，并不能随机分配。与栈一样，堆这个名字取得也非常形象，堆内存就像一个一堆沙子，可以东一瓢西一勺地取用，用完了又可以随时规还，应用程序在需要内存时，就可以向操作系统申请，操作系统就会从堆中取出一块内存给应用程序。与栈相比，堆最大的特点就是使用自由度大，有些系统为了减少内存碎片而做了一些限制，例如有些操作系统把堆内存分块管理，只能分配固定尺寸的内存块，而不能按字节随意分配，但堆内存可以自由使用的特点仍然没有改变。当然，自由是要付出代价的，从堆中分配内存要需要付出的代价主要在以下几个方面，详见 2.9.4 节

1. 操作系统管理堆需要软件开销，软件从堆中分配内存的执行效率比静态分配要低，实时性要求高的代码可能不允许。
2. 从堆中分配内存有分配不成功的风险，对可靠性要求高的操作是不允许的。
3. 频繁从堆中分配和释放内存可能会造成内存泄漏和内存碎片。

堆一般由操作系统管理，但是在一些没有操作系统的环境中，符合标准 C 规范的编译器也会在运行时库中提供 malloc 和 free 族函数。不管有没有操作系统管理，使用堆之前都必须先建立堆，建立堆的具体过程请参考所使用的开发工具的说明，并没有统一的方法。一般来说，下列步骤是必须的：

1. 提供堆的起始地址。有些系统支持不连续的内存作为堆空间，嵌入式系统可能存在这种情况，这时，就需要指定多个堆空间的起始地址。
2. 获得堆的结束地址，堆的结束地址和内存量有关，内存量可以自动探测，这需要相应的硬件支持，通用计算机通常用这个方法，而嵌入式系统的内存配置一般比较固定，很少用自动探测的。嵌入式系统的堆结束地址一般用配置文件确定，这跟具体

的开发工具有关。如果是不连续的堆空间，则每块内存的长度都要单独指定。有些系统堆的结束地址可能不是固定的。

3. 初始化相关数据结构。

djyos系统使用gcc编译器，在gcc的编译连接脚本文件里定义了两个常数，cn_heap_top表示堆顶地址，cn_heap_bottom表示堆底部地址，这两个常数是由编译器赋值的。数据结构则用m_init_heap函数初始化，详见 7.2 节。

2.6 汇编语言程序中使用栈

无论从什么角度看，栈都是一个伟大的发明，不管如何歌颂它都不过份，这里只讲述如何利用栈编写高可靠的嵌入式软件。看过一个大公司的统计数据，行数相同的C和汇编代码，其bug数量比是 1：4，而用汇编写程序，其代码两一般是C程序的 3 倍以上，也就是说，完成相同的功能，汇编的bug数量可能是C的 12 倍？为什么会有如此高的bug率，很多人归结为C语言的简洁、易读、一致上，确实这些都是降低bug的非常有效的因素，但是很多人都忽略了一点，就是C语言中栈的使用。众所周知，大量使用全局变量是造成意大利面条式程序的罪魁祸首，这些全局变量可能造成模块之间的紧耦合，很容易使程序陷入混乱，带来很多意想不到的bug。C语言允许函数定义局部变量，C编译器把局部变量存储在栈中，无论函数对局部变量做何种操作，都不会影响别的程序模块的运行。使函数的调用者和被调用者可以互相不认识，使软件的结构化和构件化成为可能。然而嵌入式系统中不可避免要用到汇编语言编程，尤其是一些比较低端的嵌入式系统，甚至会大量采用汇编语言编程。而汇编语言作为指令的符号表示，并不直接支持在栈中分配内存，定义局部变量后，还需要手工移动栈指针才能实现在栈中分配局部变量（见 2.6.2 节）。因此，很多汇编语言程序中，栈往往只是用来保存返回地址和寄存器的，不管是局部用途还是全局用途的变量，都被定义成全局变量。这就导致许多汇编语言写的程序里充斥这大量全局变量，这些大量使用的全局变量就是bug的温床，还会使程序变得难于维护。从 2.4 节可知，栈是可以重复利用的，重复利用就可以提高内存的利用率，而全局变量是静态分配的，不能重复利用，这样就会浪费内存，使用汇编语言编程，目的之一就是为了节省内存。

2.6.1 全局变量局部化

一般来说，存在大量汇编语言的程序很少工作在操作系统环境下的，不使用操作系统就不存在多个线程同时操作同一个全局变量的问题，一个函数就算与其他函数共享该变量，也不会互相影响，就像使用局部变量一般。我们可以利用这个特点，定义全局变量来当作局部变量使用，我们姑且称这种变量为“局部化的全局变量”，代码 2-5 演示了全局变量重复使用的过程：

```
// 两个范例函数。
//函数 global_local_var_example1 使用 4 个 32 位变量和 1 个 16 位变量
void global_local_var_example1(void)
{
    uint32_t i0,i1,i2,i3;
    uint16_t j;
    i = 10;
    j = 20;
```

```
}
//函数  global_local_var_example1  使用 4 个 8 位变量和 2 个 16 位变量
void global_local_var_example1(void)
{
    uint8_t i0,i1,i2,i3;
    uint16_t j0,j1;
    i = 40;
    j = 50;
}

@以下用汇编程序实现上述函数，重复使用全局变量替代局部变量
    .data               @  定义足够使用的全局变量          注 1
    .align 4
u32_var1:    .space 4    @第一个 32 位变量
u32_var2:    .space 4    @第二个 32 位变量
u32_var3:    .space 4    @第三个 32 位变量
u32_var4:    .space 4    @第四个 32 位变量

    .align 2
u16_var1:    .space 2    @第一个 16 位变量
u16_var2:    .space 2    @第二个 16 位变量

    .align 1
u8_var1:     .space  1    @第一个 8 位变量
u8_var2:     .space  1    @第二个 8 位变量
u8_var3:     .space  1    @第三个 8 位变量
u8_var4:     .space  1    @第四个 8 位变量

    .text
    .global global_local_var_example1
global_local_var_example1:
    STMFD    r13!,{r4,r5,r11,r14}
    .equ     i0,u32_var1       @  为使用到的变量取别名，增强可读性     注 2
    .equ     i1,u32_var2
    .equ     i2,u32_var3
    .equ     i3,u32_var4
    .equ     j,u16_var1
    ldr      r4,=i0
    mov      r5,#10
    str      r5,[r4]
    ldr      r4,=j
    mov      r5,#20
    strh     r5,[r4]
    LDMIA    r13!,{r4,r5,r11,pc}
```

```
    .global global_local_var_example2
global_local_var_example2:
    STMFD    r13!,{r4,r5,r11,r14}
    .equ     i0,u8_var1
    .equ     i1,u8_var2
    .equ     i2,u8_var3
    .equ     i3,u8_var4
    .equ     j0,u16_var1        @ 重复利用 global_local_var_example1 函数用过的变量。注 3
    .equ     j1,u16_var2
    ldr      r4,=i
    mov      r5,#40
    strb     r5,[r4]
    ldr      r4,=j0
    mov      r5,#50
    strh     r5,[r4]
    LDMIA    r13!,{r4,r5,r11,pc}
```

代码 2-5 重复使用全局变量

1. 定义足够的“全局局部变量”，程序只有两个函数，第一个函数用到 4 个 32 位变量和 1 个 16 位变量；第二个函数用到 2 个 16 位变量和 4 个 8 位变量，因此，定义 4 个 32 位变量、2 个 16 位变量、4 个 8 位变量就足够所有函数使用了。16 位变量只需要 2 个，而不是两个函数加起来的 3 个。
2. 为了增强程序的可读性，用伪指令给将要使用的全局变量起个别名，作为局部变量的名称，这个别名要有清晰的语义，象 C 语言的局部变量一样。
3. global_local_var_example1 函数用过的全局变量 u16_var1，在这里可以重复使用。

这种方法的优点是通用性强，几乎在所有汇编语言上都可以实现，使用上也很直观，易于理解。由于局部变量实际上是静态定义的，有固定的地址，在编写代码时可以使用多种寻址方式，在栈中定义的局部变量只能用相对寻址。与直接使用全局变量相比，这种重复使用局变量的方法节省了大量的内存。

这种方法也有明显的局限性，1 是只能用在单线程的场合，也不允许中断函数调用；2 是需要手工计算和维护全局变量数量。

2.6.2 在栈中定义局部变量

直接从栈中分配内存给局部变量，这也是绝大多数C编译器的标准用法。代码 2-6 演示了如何用汇编语言在栈中定义变量的例子，函数use_stack( )定义了两个局部变量，这两个局部变量的内存就是在栈中分配的。随后用汇编语言实现的use_stack( )函数，注意这是用汇编语言写的代码，不是C的反汇编。

```
    void   use_stack(void)
    {
        int vara,varb;
        vara=100;
        varb=1000;
    }
```

下面是用汇编写的实现上述函数的代码，

```
    .data
    .struct 0－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－①
vara:           //定义变量 vera,相对堆栈的偏移地址为 0
    .struct vara + 4
varb:       //定义变量 verb，相对堆栈的偏移地址为 4，也表示变量 vera 的长度为 4
    .struct varb + 4 //verb 的长度也是 4
end_of_use_stack:
    .text
    .global   use_stack
use_stack:
    STMFD      r13!,{r3,r11,r14}     //保护寄存器
    SUB        r11,r13,#4              //r11 为局部变量访问的基址寄存器－②
    SUB        r13,r13,#end_of_use_stack     //调整栈指针到栈底－－－－③
    MOV        r3,#0x64
    STR        r3,[r11,#-vara]        //执行 vara=100
    MOV        r3,#0x3e8
    STR        r3,[r11,#-varb]        //执行 varb=1000
    ADD        r13,r13,# end_of_use_stack     //恢复栈指针－－－－－－－－④
    LDMIA      r13!,{r3,r11,pc}       //恢复寄存器
     .data
    //下一个函数的局部数据定义
     .text
    //下一个函数的代码
    .end
```

代码 2-6　在栈中定义变量

1. struct 伪指令本来是用来定义结构的，当用指针访问结构变量时，和局部变量的寻址方式类似，这里借用来定义局部变量。每个局部变量就像结构成员一样，按每个局部变量的长度以累加的方式指定各变量的偏移地址，注意指定地址时必须满足 cpu 的对齐要求，否则可能出现灾难性的后果。在 ARM 公司的汇编器中，MAP 伪指令专门用来定义局部变量，且该伪指令能处理对齐问题，使用起来很方便。ARM 汇编器中用 LCLL、LCLA 和 LCLS 定义的“局部变量”实际上是“局部伪变量”，是给编译器用的，主要用于条件编译的逻辑表达式。
2. 由于每次调用函数时，栈指针是未知的，因此在栈中定义的变量只能指定其偏移地址，使用的时候必须用基于当前栈指针的[基址+偏移地址]的方式寻址，这里指定 r11 为相对寻址的基址。
3. 前面.struct 为每个变量定义了长度和偏移地址，但并没有分配存储空间，这里为局部变量分配了 8 个字节的存储空间。
4. 局部变量使用完毕，恢复栈指针，隐含地收回了分配给局部变量的内存。

如果用反汇编观察由 gcc 编译器产生的 use_stack( )函数的代码，我们会发现其处理方式与上述过程非常相似，编译时必须选 0 级优化，否则编译器有可能会直接分配寄存器给 vara 和 varb。如果用的是 ads1.2 编译器，即使选 0 级优化，编译器还是直接用寄存器，必须定义大量的局部变量迫使寄存器不够用才会把局部变量放在栈中。

绝大多数的嵌入式系统都可以模拟 C 语言的行为来实现在汇编语言中使用局部变量。

因为与 C 编译器的行为相似，这样编写的代码非常适合和 C 语言混合调用。

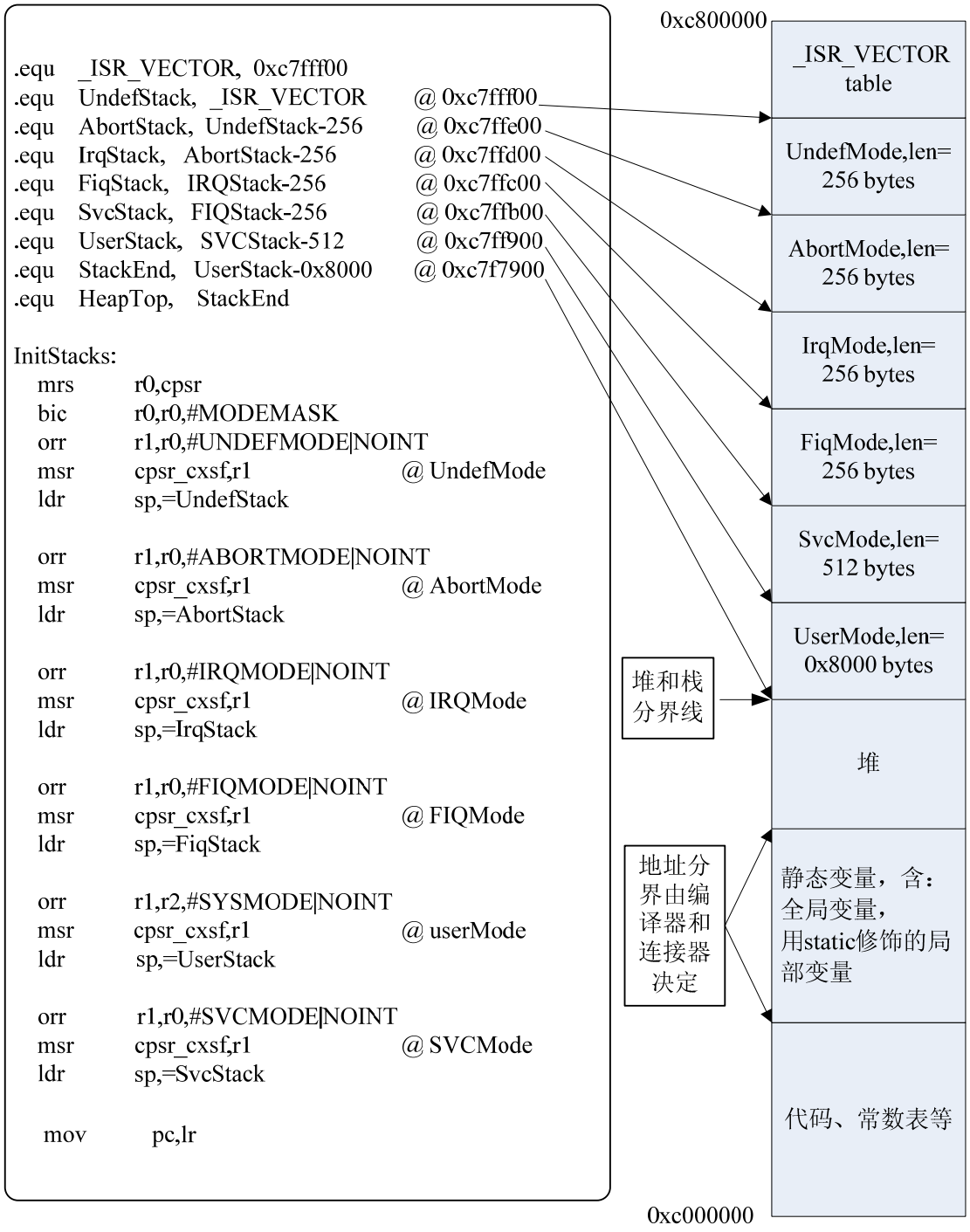
并不是所有 cpu 都支持这种方式的，比如 microchip 公司的 pic 系列 8 位单片机，这个系列 cpu 只提供硬件栈，只能用来保存返回地址，没有可以保存变量和参数的内存栈，需要人为地用内存模拟栈，使用这种方法就比较复杂，所以，用 pic 单片机时，最好还是使用“局部化全局变量”的方法。而 microchip 公司在其 16 位的 pic24 单片机中抛弃了这种设计。

2.7 堆与栈的关系

前面单独讲述了堆和栈的概念，实际上，堆和栈都是由计算机管理的一段内存构成的，他们的区别在于软件管理的方式不同。那么，软件是按照什么规则划分堆和栈的呢，堆和栈可以互相转化吗？任意指定一个内存单元，我们如何确定它属于堆空间和栈空间呢？在没有操作系统的情况下，这个问题是很明确的，但自从引入操作系统以后，这个问题就变得复杂起来。

2.7.1 单线程的堆和栈

图 2-6 演示了一个范例系统在没有操作系统情况下栈的初始化过程，以及初始化后这个系统的内存映像图。CPU是ARM7TDMI核的s3c44b0x，内存的起始地址是 0xC000000，共 8M。



某ARM系统 在无操作系统时的内存映像，堆栈分明

图 2-6 某ARM系统 在无操作系统时的内存映像，堆栈分明

1. ARM7 有 7 种运行模式，系统模式和用户模式共用用户模式栈以外，其他模式都有各自独立的栈指针寄存器，程序首先为 6 种模式分别定义栈的边界地址和栈长度，并且使该模式对应的栈指针指向栈顶。程序中使用到的模式必须为其分配栈，软件中没有用到的模式可以不分配栈空间，例如有些程序直接在特权模式（SvcMode）下运行，就不必为用户模式（UserMode）保留栈空间。
2. 0xc7fff00 以上是中断向量区域，UndefMode 的栈顶被定义在 0xc7fff00 处，随后是 AbortMode，栈顶地址为 0xc7ffe00，UndefMode 栈顶与 AbortMode 栈顶之间的 256

bytes 是分配给 UndefMode 的栈空间。其他模式所拥有的栈空间依次类推。

3. 每个栈的大小也是有程序员决定的，程序员应该准确估计各种模式下局部数据的总量（参见 2.8.2 节）。图 2-6 中给用户模式分配了 32K栈，是因为用户软件主要运行在用户模式（或系统模式），需要较多的栈空间。如果分配的栈小于实际要求的局部数据数量，在程序运行过程中就有可能破坏别的cpu模式的栈，可能会引发灾难性的后果。有些编译器允许产生带栈检查的代码，可以帮助用户在设计的时候发现栈溢出错误。
4. 从图 2-6 中可知，栈以下是堆空间，可用于动态分配内存。如果程序在ram中运行，则堆空间尺寸的计算方法为：堆尺寸＝总内存－所有栈的和－静态变量－代码（含常数表）。如果程序在ROM或flash中运行：堆尺寸＝总内存－所有栈的和－静态变量。静态变量、代码的尺寸可以从连接器的输出中得到，具体方法取决于所用的编译器。
5. 堆和栈共用内存，他们的总和是固定的，有些程序并不使用动态分配内存，可以把所有空闲内存都作为栈使用。

可见，没有操作系统时，堆和栈有非常清晰的界限，除全局变量、静态变量和代码外，任意一个内存单元，要么处于堆区，要么位于栈区，不会发生相护转换的情况。

2.7.2 多线程的堆和栈

当有操作系统支持时，由于有多个用户线程并发执行，为了保存各自的局部变量，每个线程都需要有独立的栈。2.7.1 节中可知，CPU各运行模式包括用户模式的栈是在系统初始化时设定的，在CPU初始化阶段，初始化程序并不知道这个系统是准备运行操作系统还是作为单线程运行，所以，它不会为需要操作系统支持的线程分配栈空间。事实上，操作系统内核和用户线程加在一起，可以当作一个大程序看待。在操作系统启动阶段，操作系统并不知道用户要建立多少线程，也不知道各线程需要多少栈空间，是不可能象单线程那样预先分配栈的。事实上，每个用户线程的栈空间是用动态分配内存的方法从堆中分配的，当需要建立新线程时，程序就执行“创建新线程”的系统调用，这个系统调用的其中一个参数是新线程需要的栈空间尺寸。操作系统从堆中分配一块合适的内存，作为新任务的栈，这块堆内存就摇身一变成为栈内存；相反地，当用户删除任务时，操作系统将收回早先分配的内存，重新放进堆区，这块内存又从栈内存变回堆内存。图 2-7 显式了一个有三个线程的系统，每个线程的栈，从操作系统角度看，它是已经分配的堆，从线程的角度看，它是程序运行的栈空间，同一块内存的堆和栈属性在程序运行过程中发生了转化。

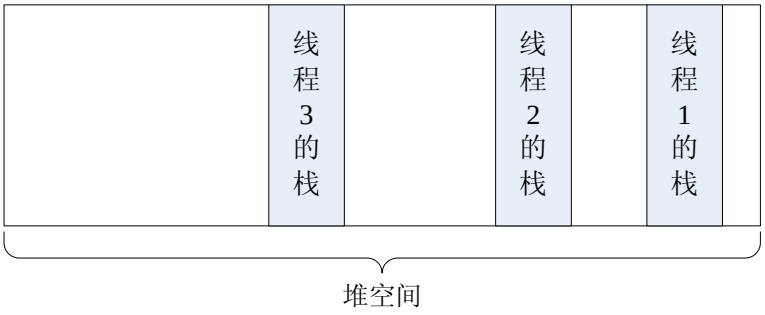


图 2-7 有三个线程的内存映像

2.8 该为系统配置多少内存

内存是嵌入式系统的关键部件，如何确定你的产品需要多少内存，这是个极其重要也极具挑战性的问题。内存配置多了，造成产品的成本升高；内存配置少了，产品功能不能实现。尤其是小规模的嵌入式系统，多采用内置存储器的 CPU，对内存需求的估算直接影响 CPU 选型，而 CPU 选型不当的话，可能导致整个硬件计划推倒重来，而且软件计划也可能会大受影响。对于大型嵌入式系统，内存一般会使用外置存储器芯片，如果内存配置不适当，经常更换内存芯片就可以解决，有时候连 PCB 都不用改，对项目进度的影响要比小型系统小些。

确定内存尺寸，可以分为两个阶段任务，首先是在项目系统设计时估算内存需求，其次是在项目基本完成后核对内存配置是否合适。在第一阶段，由于软件设计还没有开始，估算往往要靠设计师的经验，参考类似设计也是重要手段，系统设计师的经验在这一阶段无疑起决定性的作用。第二阶段检验内存配置却是有章可循的，为叙述方便，我们先以单线程情况为例。

2.8.1 图解多线程环境栈溢出

在多线程环境中，线程的栈是用malloc族函数从堆中分配的，程序运行过程中用到的动态内存也是用malloc从堆中分配的，动态内存和栈混杂在同一个地址空间内。如果某线程分配的栈不够，使栈指针越过栈边界，就发生了内存溢出。栈溢出的后果是致命的，因为你不知道栈相邻的内存的用途，它可能是代码、也可能是全局数据，也可能是其他线程的数据或栈，也可能你中了大奖，该内存恰好是空闲的。由于栈是从堆中分配的，可能性最大的是，栈相邻的内存也是堆的一部分，栈溢出破坏的是本线程或其他线程从堆中分配的内存块。图 2-8 显示了一个线程栈溢出破坏临近的动态分配的内存块的过程，图中thread1 发生栈溢出，最终导致无辜的thread3 莫名其妙地崩溃。栈溢出导致的故障往往是不可预见的，特别是，出问题的线程往往是本身没有任何缺陷的，排查这样的缺陷是非常困难的。

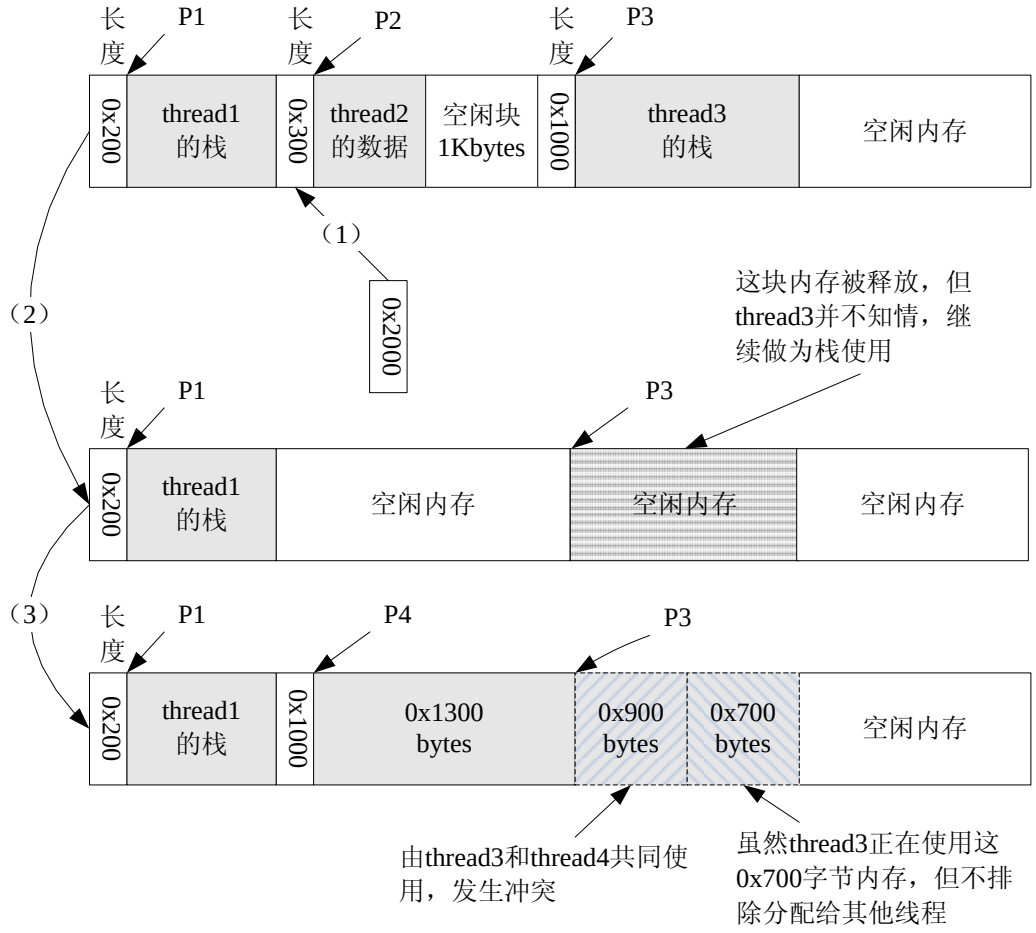


图 2-8 内存溢出的后果

1. 图中显示的是用 malloc 族函数从堆中分配的 thread1 和 thread3 的栈和 thread2 的数据，内存管理器在每个内存块相邻地址保存本内存块的尺寸。
2. thread1 的栈的相邻地址是 thread2 调用 malloc 函数分配的内存块，该内存块的尺寸是 0x300 字节。
3. thread1 发生内存溢出，覆盖了保存 P2 尺寸的存储单元，把 P2 的尺寸改为 0x2300。
4. thread2 执行 free(P2)，内存管理器就会把 P2 开始的 0x2000 字节的内存释放，导致 thread3 的栈也被释放掉。而 thread3 对此毫不知情，继续作为栈使用，而操作系统却已经准备好把 thread3 的栈作为下一次 malloc 的分配对象。thread3 被虽置于非常危险的境地，但仍然正常运转，直到 thread3 的栈被分配且被访问，危机才爆发。所以，从危机的产生到危机爆发，有一个不定时长的潜伏期，这更凸显了栈溢出错误的危险性。
5. thread4 执行 P4=malloc(0x1C00)，内存管理器从空闲内存中切割出 0x1C00 字节内存给 thread4。这 0x1000 字节内存中前面 0x1300 字节是安全的，后面 0x900 字节是从 thread3 的栈中切割出来的，典型的一女两嫁！

2.8.2 测算线程需要多大的栈

上一节可知，如果创建线程时分配的栈空间不够，将会发生非常严重且神秘莫测的问题。那么，如何计算一个线程需要多少栈空间呢？从前面的论述可知，栈是用来存储局部数据和

参数的，我们只要找出线入口函数以及调用各级子函数的可能路径，把该路径中每次调用函数时所需要的参数和局部变量（静态局部变量除外）所需要的内存相加得到该路径的栈需求，取栈需求最大的一条路径，再乘以一个安全系数就可以了。注意如果发生递归调用，则递归函数所需要的内存还应该乘以最大迭代次数；有些系统中断处理函数使用被中断线程的栈，则每个线程的栈都要加上中断嵌套最深时的中断栈需求。计算过程如下：

1. 列出函数的参数和局部变量（不含静态变量）所占的空间之和，即是该函数需要的栈空间，用这种方法计算除线程用到的所有函数需要的栈空间。
2. 列出线程入口函数调用各级子函数的所有可能路径。
3. 计算各调用路径所需要的栈空间，把该路径上所有调用的函数需要的栈空间相加即得，若有递归调用，递归函数应该重复计算最大可能的递归次数。
4. 所有路径的栈空间中，挑出最大的一个，再乘以一个安全系数，就是栈的总空间。该安全系数一般取 1.2 即可，要求特别高的系统，可取 1.5。
5. 如果中断 ISR 函数也使用被中断线程的栈，则用同样的方法计算所有中断入口函数需要的栈空间，如果允许中断嵌套，还要把可能互相嵌套的中断栈相加。取栈空间需求最大的一个中断源。
6. 由步骤 4 计算的线程入口函数栈需求，加上步骤 5 计算的最大中断栈需求，就是需要分配给该线程的栈空间。

有些编译器或者操作系统会提供一些工具，让程序员在开发调试时能够观察栈的使用情况，及时发现并纠正潜在的栈溢出错误。也可以用简单方法查看栈是否有溢出的危险，所谓的“内存着色”就是其中一种方法，在程序运行前，把整个栈空间填充成指定数据（着色），让程序运行一定时间，并且在运行时从外部输入各种可能的条件，让程序执行过程经历各种可能的路径，经过足够长时间后，把程序停下来，看栈的变色区是否接近或超过栈底。VxWorks 操作系统的 luostcheckStack( )函数和 checkStack( ) 函数就是用“内存着色”的方法检查栈溢出的。

如果是单线程环境，main函数的栈需求就是总的栈需求。有些CPU有多种状态，每种状态使用独立的栈，则还要为每种状态计算栈需求。比如ARM有SVC、IRQ、FIQ、UNDEF等 6 种状态，每种状态都有独立的栈指针，使用独立的栈，就要为每种状态的入口函数计算其栈需求，并分配足够的栈。如 图 2-6 所示的代码，就为ARM的各种状态分配了栈以及初始化了其栈指针。

2.9 内存分配

2.9.1 静态分配和动态分配

静态分配就是用硬编码的方式分配内存，程序编译时所需要的内存都已经分配好了，定义全局变量，全局数组、静态局部变量或静态局部数组都属于静态内存分配。静态分配的内存一经分配就永久占用，即使你从来不使用它，直到程序的运行期结束，而嵌入式系统的程序一般是连续运行，没有结束的时候。采用静态分配不可避免地使系统失去了灵活性，必须在设计阶段就预先知道所需要的内存并对之作出分配；必须在设计阶段就预先考虑到所有可能的情况，因为一旦出现没有考虑到的情况，系统就无法处理。这样的分配方案可能造成很大的浪费，因为内存分配必须按照最坏情况进行最大的配置，而实际上在运行中可能使用的只是其中的一小部分；如果硬件没有预留足够的备用内存空间，将不能灵活地为系统添加功能，从而使得系统的升级变得困难；而过多地配置备用内存，又会带来成本的上升。静态分

配也有其优点，它可以确保内存需求得到满足，能保证程序快速运行。静态内存分配主要用在以下几个场合：

1. 实时性和可靠性的要求极高（硬实时系统），不能容忍一点延时或者一次分配失败，比如汽车的 ABS 控制、生命维持系统、飞机的飞行控制系统等。
2. 系统内存十分充裕，使用静态分配可以降低软件设计难度，降低 bug 风险。
3. 固定的内存需求，例如嵌入式系统中常用的点阵字库，该字库在系统运行时自始至终必需保存在内存中，不会被释放。

动态内存分配即在运行时分配内存，在堆和栈中分配内存就属于动态分配。如果程序需要维护一个动态变化的链表或者树时，你可以用静态方法分配一块足够大的内存，这样做虽然可能会浪费一些内存，但可以降低风险和提高运行速度；你也可以使用从堆中动态分配的内存，这样可以提高内存的使用效率，但要忍受分配内存的开销，还有分配不成功的风险。内存管理器统一管理堆，当程序运行过程中需要使用内存时，就调用 malloc 族函数申请分配内存，或者建立新线程时，操作系统需要为新线程分配栈空间。如果有足够的连续空闲内存，内存管理器将从空闲内存中割下一块合适的内存给申请者，把该块内存的地址返回给申请者；否则返回 NULL，或者阻塞线程。嵌入式环境下的情况比较复杂，在有操作系统时，内存由操作系统管理，但有些嵌入式系统没有操作系统，或者是非常轻量级的操作系统，这种操作系统没有做内存管理，这种情况下的动态内存分配将依赖 C 运行库，标准 C 定义了库函数 malloc()族和 free()族函数，大多数嵌入式系统使用 C 运行库支持这些函数，他们可以为应用程序分配和释放任意大小的内存块。

从 2.4 可知，定义非静态局部数组是一种特殊的内存分配方式，局部变量在栈中分配内存，兼有静态分配和动态分配的特征。与静态变量一样，局部变量在编译时就已经确定，程序执行时是立即获得所需内存，无需额外的CPU时间，并且与静态变量一样可以直接用变量名而非指针进行访问，这样分配的内存虽然不象全局变量有一个唯一的地址，但其相对与栈指针的偏移地址却是在编译时就确定的。而动态分配的内存需要消耗额外的cpu时间，且动态内存只能用指针间接访问。与动态分配一样，栈中分配的内存可以重复使用。如果堆空间容量不足，就存在动态内存分配失败的风险，同样，如果栈空间容量不足，也存在局部变量分配内存失败的情况，这种情况叫“栈溢出”。栈溢出是一种非常严重的软件bug，堆空间不足时，只会导致分配失败，而栈溢出则没有任何提示，往往会导致软件整体崩溃，其后果远比堆内存分配失败严重。

且看一个栈溢出的案例，代码 2-7 是一个很典型的例子，在定时器中断函数中，如果adc模块被打开就启动ad转换。在adc_module_open=false的情况下，只要加上if部分代码，串口通信程序就出错，偶尔出现校验和不对的情况。只要把if部分代码注释掉，使if语句成为空语句，通信就正常。排查这个问题可谓是一波三折，由于这部分代码与出错的代码毫无关系，因此一直没有怀疑到这里，最后是在地毯式逐个函数逐个代码块地增减代码才把故障定位。由于if条件不成立，该段代码根本部会被执行，但为什么会影响到通信程序呢？原来是栈溢出惹的祸，该项目的连接配置文件中，给中断函数保留的栈空间是 512 字节，而数组adc_wave的大小是 1000 字节，除if部分代码外，没有其他代码访问adc_wave。当注释掉if部分代码时，编译器的优化器发现adc_wave是冗余的定义，没有在栈中为它分配内存，函数的其他局部变量总共只有 10 多字节，局部变量的总量没有超过连接配置文件规定的栈空间；当加入if部分代码块时，编译器就在中断栈中为adc_wave分配内存，所需要的栈空间就超过了 512 字节，发生栈溢出。栈溢出后，u16l_i以及其他局部变量就侵犯了其他内存空间，你根本不知道编译器会把这些空间用作什么用途。调试发现溢出的局部变量刚好覆盖了串行通信的发送缓冲区，导致串行通信出错。多么可怕，两段原本风马牛不相及的代码，由于栈溢出bug却互相破坏。你根本无法预计栈溢出会破坏哪些模块，也无法预计什么时候bug会爆发，

这是栈溢出bug最可怕的地方。

代码 2-7 中断服务函数栈溢出

```
void int_timer0(void)
{
  uint16_t   adc_wave[500];
  uint16_t   u16l_i,u16l_j;
  //在此省略了其他变量定义

  //省略部分代码
  if(adcmokule_open==true)
  {                                          //注释掉本代码块则故障消失
    //省略启动 adc 的操作
    adc_wave[u16l _i]=adc_channel[u16l _j];
  }
}
```

2.9.2 任意长度分配法

这种方法允许分配任意长度的内存（当然，受最大的空闲内存块限制），内存管理器无法知道应用程序何时申请内存，何时释放内存，每一次的申请和释放都有可能产生新的空闲内存块，也可能消除碎片，因此操作系统需要管理的空闲内存块的数量和内存块的大小是无法预料的，显而易见，链表可以用来管理空闲内存块。许多操作系统采用空闲内存链表的方式管理，链表的每个节点代表一个空闲内存块，节点记录该空闲块的大小，节点所需的内存直接取自空闲块，因此这种内存管理方法无需额外的开销。

任意长度内存分配的算法大致有如下几种。

① 最佳适合法（best fit）：存储管理器将寻找一个适合的且最小的空闲内存块，从中切割出所请求的内存块，比如内存中有 8KB、14KB、17KB、20KB、13KB 大小的 5 各空闲块，如果应用程序要求分配 13.5KB 的内存块，将从 14KB 的内存块中分割出 13.5KB，剩下的 0.5KB 作为一个新的空闲块保留在内存链表中。这种方法总能留下较大的内存块来满足某些需要大块内存的应用程序，但是很容易留下很多零碎的内存快。

② 最大适合（worst fit）：这种策略总是从最大的空闲内存块中切割出合适的尺寸给请求者，这样就可以避免留下很多零碎的内存块，但是会使大块内存需求得不到满足。最大适合和最佳适合法都要对链表进行排序，在排了序的链表中寻找合适的内存块比较快，但是释放内存时需要查找相邻内存块进行合并时，则几乎要遍历整个链表，速度非常慢。

③ 最快适合（first fit）：这种策略只要找到一个比所请求的内存大的空闲块，就从中切割出合适的块，把剩余的小块重新加入空闲链表。本策略的一个变种是“下一个适合（next fit）”法，即一次分配后，指针停在链表的下一个节点处，下次分配时从下一个节点开始搜索。

无论哪一种算法，都存在操作一个不确定长度的链表的问题，这种操作需要花费不确定的时间，这在实时系统中是不允许的。

2.9.3 固定块分配法

任意长度的内存分配要求处理一个不确定的链表，内存分配所花费的时间是不确定的，而且即使有足够的空闲内存量，内存分配能否成功也要取决于碎片状况，而不可预料的时间和执行结果正是实时系统的大忌。实时系统要求系统有确定而且快速的反应时间，还要求有确定的执行结果，任意长度分配内存的方法无法满足实时系统的需求，块分配法在实时系统中得到广泛的应用。

这种方法把内存分为固定大小的块，要求应用程序一次只能请求一块内存。它把所有空闲内存用链表连接起来，每次内存请求就从链表头部分配一块内存给请求者，应用程序归还的内存加入到链表中。这种策略来管理内存的最大优点是只要系统有空闲内存，内存分配花费的时间是确定的。但也有明显的缺点，应用程序的内存需求是多种多样的，用一种内存块尺寸可能无法满足应用程序的需求，因为块尺寸如果定义得小了，那些需要大块内存得需求便无法得到满足，如果块尺寸定义得太大，对小块内存需求来说则显得太浪费。改进的方法是定义多个内存池，每个内存池按不同的块尺寸进行分配。

2.9.4 块相联分配法

操作系统应该允许应用程序在运行中动态建立新线程和终止线程，在前面的章节中我们已经谈到，建立新线程就需要为其分配栈，线程栈由操作系统从堆空间分配。对于操作系统来说，线程栈的长度是任意的，另外，程序运行过程中也可能需要动态分配内存，操作系统并不知道应用程序需要分配多少内存，因此，操作系统需要一个允许应用程序分配不确定长度的内存的内存管理器。任意长度分配法由于其执行时间的不确定性，难于在实时操作系统中应用，而固定块分配法虽然有确定的执行时间，但却只适合于特定的应用，无法满足分配不确定长度内存的需求。于是，块相联分配方法兴起。这种策略把内存划分为固定大小的页，一次可以分配 1 或多个连续页，一次可以分配的连续页数可能不是任意的，与具体实现有关。块相联分配法实际上是固定块和任意尺寸分配法的折中，它兼顾了应用程序可能需要任意长度内存块和内存管理的时间开销可预见且快速的需求。

在实际应用中，块相联和固定块分配法可以联合使用，频繁地从系统堆中分配内存容易造成内存碎片，且执行效率也低，应用程序需要内存时，可以一次从系统堆中分配较大块的内存，把这块内存作为固定块分配法的内存池进行二次管理。这种方法特别适合于可配置的应用，例如嵌入式系统常见的 CAN 线程总线应用中，程序运行起来后，通过读配置文件获取总线连接的最大可能结点数，需要为总线上的每个结点分配一个结点控制块，我们可以用块相联法分配足以容纳最大节点数的一大块内存，再用固定块分配法把这块内存分配给各结点。

2.9.5 适时分配和释放堆内存

内存是共享的临界资源，同一块内存在同一时刻只能由一个线程使用，当没有足够的空闲内存时，申请内存的线程要么放弃操作，要么等待，直到持有内存的线程（进程）主动放弃（释放）内存为止。因此，持有内存的线程（进程）应该尽量缩短占用内存的时间，如 图 2-9 所示，在使用内存时才申请，使用完毕后立即释放，否则，操作系统并不知道内存已经使用完毕。内存使用完毕后及时释放，还有助于防止产生内存碎片。

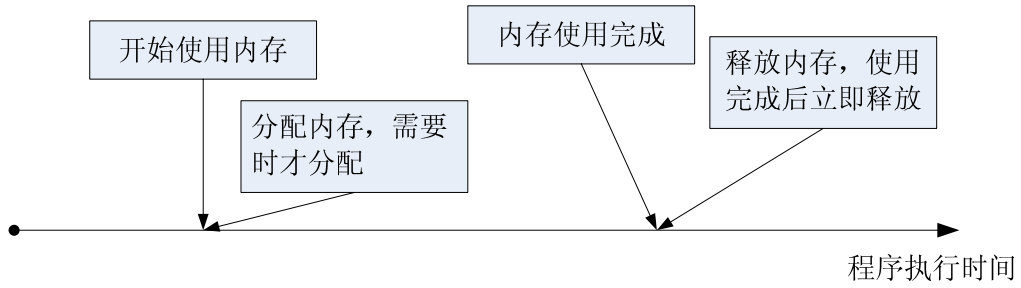


图 2-9 普通代码动态分配内存

然而，分动态配内存是十分费时且难于保证成功的，执行时间也难于预测，甚至会造成阻塞，因此，实时程序应该避免动态分配内存。如果你非要让时间关键代码使用动态分配，折衷的办法就是如 图 2-10 所示，在时间关键代码开始前分配好内存，内存使用完毕后，不要急着释放，而是在时间关键代码结束后才释放。

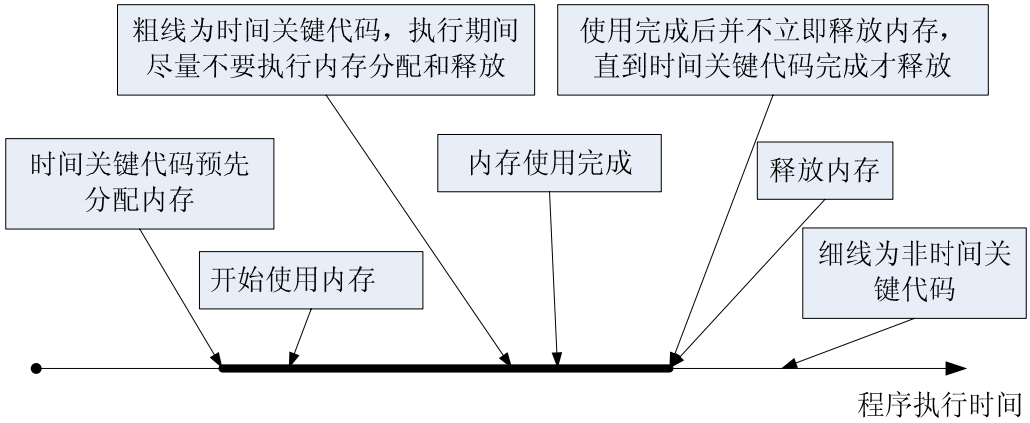


图 2-10 实时代码动态分配内存

2.10 动态分配与内存枯竭

程序执行一段时间以后，随着内存块的使用和释放，许多正在使用或已经释放的存储块将混杂在一起，未被使用的内存将分裂成许多小块内存，这就是内存碎片。内存碎片的数量取决于内存管理的实现策略，大量的内存碎片将带来三个后果：

1. 降低内存使用效率，虽然有大量的空闲内存，但分配一个并不是很大的连续内存空间却不可得。如果内存中最大的空闲内存块为 90 字节，即使空闲内存的总量超过 1000 字节，仍然无法满足 100 字节的内存分配请求。
2. 如果使用任意长度分配法，大量碎片会使内存管理的开销增大，大量的内存碎片使空闲内存链表的节点数量增加，延长了搜索和管理链表所需的时间，分配和释放内存所花费的时间变得很长且不确定。
3. 造成伪内存丢失，伪内存丢失指的是该内存虽然仍然被操作系统登记在案，但却永远不可能被使用。假如动态申请的内存块最小尺寸为 100 字节，如果内存区中存在一个 90 字节的内存碎片，只要这个碎片的相邻内存块不释放与本块合并，那这 90 字节内存永远无法使用。

动态内存分配的另一个可能的后果就是内存泄漏，也叫内存丢失，内存丢失是一块已经

分配但不再被任何人使用且永远不可能被释放的内存块，属于最严重的软件事故之一。如果一个内存块不再有指针指向它，这块内存就丢失了。内存丢失将直接使系统可用内存减少以致内存被逐步耗尽，对于要求高可靠性且长时间连续运行的嵌入式系统，程序员需要确保绝对没有内存丢失。内存丢失是软件使用 malloc 和 free 族函数不当的 bug 造成的，一个没有 bug 的程序是不会产生内存丢失的。所以，程序员虽然难于控制内存碎片的产生，但可以消除内存泄漏，但是消除这些 bug 的难度非常大。

避免内存丢失是每一个程序员都要特别注意的问题，经验丰富的程序员只要看到程序里出现 malloc（包括 calloc，realloc）时，就要象狼一般警觉，要避免内存丢失，就必须成对地使用 malloc 和 free 函数，内存使用完后及时释放。而且 malloc 和 free 要在同一层调用，尽量不要在函数里面调用 malloc，而在函数外面调用 free。更不要在这个线程调用 malloc，在另一个线程中调用 free，造成内存碎片的错误主要有两种，一是使用完毕后没有释放，如：

```
void      func0(void)
{
      char *ptr;
      ptr=malloc(1000);
      return;
}
```

代码 2-8 不释放导致内存泄漏

另一种情况 malloc 后是内存指针被改变，如 func1 所示，执行 free 时，将发生不可预料的情况，即使程序不崩溃，至少也会发生内存丢失。

```
void      func1(void)
{
      char *ptr;
      ptr=malloc(1000);
      ptr++;
      free(ptr);
      return;
}
```

代码 2-9 不匹配的 malloc 和 free

内存碎片和内存丢失都可能会导致内存枯竭，而且问题是慢慢积累发生的，内存资源越丰富，积累的时间就越长。这个问题功能测试往往发现不了，而是在软件运行一定时间后才表现出来，排查这类问题是非常困难的。由于堆的使用是应用程序的自主行为，操作系统无从知道应用程序要干什么，所以也无法进行干预。只要存在动态分配内存，任何高明的操作系统都没有办法防止内存丢失的发生，一切只有靠程序员自己保重。总而言之，应用程序员必须自己管理内存，操作系统只能提供有限的协助。有些操作系统会进行内存碎片整理，使小块内存聚合而成为一块完整的大块内存。但实时系统决不会这样做，内存是一种临界资源，整理内存时必需禁止多任务调度，此时，即使有高优先级的任务就绪，也没有办法运行，而且整理内存所花费的时间是不确定的，实时系统不可能花费很长且不确定的时间来整理内存。另一个原因是，整理内存就要移动内存，这样将迫使应用程序必需以间接指针来引用动态分配的内存，而且内存的读写都必需使用原子操作，这样势必大大降低程序的执行效率，这在实时系统中也是不允许的。

2.11 提高栈的复用率

从 2.4 可知，随着函数的调用和返回，使用栈是一种特殊的动态分配内存的方式，这种方式比在堆中分配快速且高效。在栈中分配内存还要注意均衡使用以达到高复用率，在第 2.8.2 节中讲述了计算线程所需栈深度的方法也适用于任意函数。我们把需要大量栈的函数称做高消费栈函数，定义了大的局部数组的、使用大数组作为参数、以及深度递归的函数就是典型的高消耗栈函数。如果应用程序中只有极少数高消耗栈函数，那么包含这些函数调用的路径需要的栈就会很深，而调用路径中不包含这些函数则只要很浅的栈，但是我们必须为最坏情况保留栈空间。假如很多线程使用了这些高消费栈函数，则每个线程都要分配很大的内存作为栈空间。这时候，栈空间实际上只供极少数函数使用，复用率非常低。把需要很深的栈空间的少数路径作为关键路径，对关键路径进行优化可以有效提高栈的复用率，降低程序的内存消耗。代码 2-10 中，反汇编代码显示函数wave_compress需要 12004 字节的栈空间，如果某线程中不包含wave_compress的调用路径所需的栈空间最大为 1000 字节，包含wave_compress的调用路径需要栈空间 12500 字节（因复用，总的栈空间会小于 12004+1000），则至少要为线程保留 12500 字节栈内存空间，可以认为其中 11500 字节内存是wave_compress函数专用的，栈的复用率非常低。如果有许多线程需要调用wave_compress函数，则每个线程需要的栈都可能很大，整个程序需要的内存就很大。

代码 2-10 非常消耗栈空间的代码

```
void wave_compress(void)
{
  uint32_t   u32l_voltage[1000];
  uint32_t   u32l_current[1000];
  uint32_t   u32l_power[1000];
  switch(flag)
  {
    case 1:
      //本部分代码使用 u32l_voltage，u32l_voltage 仅在此使用，具体略
      break;
    case 2:
      //本部分代码使用 u32l_current，u32l_current 仅在此使用，具体略
      break;
    case 3:
      //本部分代码使用 u32l_power，u32l_power 仅在此使用，具体略
      break;
    default:break;
  }
}
```

优化 wave_compress 函数有两种方法，一是在 case 2 和 case 3 所处的代码块均使用 u32l_voltage，这样就无需定义 u32l_current 和 u32l_power， 但各代码块的功能不一样，使用相同的变量名容易造成混淆，降低了代码的可读性。

二是改成 代码 2-11 中local_voltage的形式，u32l_voltage和u32l_current和u32l_power共用栈空间，反汇编表明函数local_voltage只需要 4004 字节栈空间，比优化前节省达 65%以上，而性能没有任何损失。

代码 2-11 减少消耗栈空间的代码

```
void wave_compress1(void)
{
  switch(flag)
  {
    case 1:
    {
      uint32_t   u32l_voltage[1000];
      //本部分代码使用 u32l_voltage，u32l_voltage 仅在此使用，具体略
      break;
    }
    case 2:
    {
      uint32_t   u32l_current[1000];
      //本部分代码使用 u32l_current，u32l_current 仅在此使用，具体略
      break;
    }
    case 3:
    {
      uint32_t   u32l_power[1000];
      //本部分代码使用 u32l_power，u32l_power 仅在此使用，具体略
      break;
    }
    default:break;
  }
}
```

解除高消费栈函数的嵌套调用也是提高栈复用率的有效手段，这些函数嵌套调用的路径一般就是栈消耗的关键路径。在 代码 2-12 中，函数的调用路径为run_control→motor_drive→get_position，高消费栈的函数发生了嵌套调用，他们的栈需求相加，本路径的栈需求超过4K字节。

代码 2-12 高消费栈的运动控制程序

```
uint16_t   get_position(void)
{
  sint16_t   s16l_motor[1000];    //定义了大数组,属高消费栈函数
  //代码被省略
}

void   motor_drive(void)
{
  uint16_t   u16l_buf[1000];       //定义了大数组,属高消费栈函数
  uint16_t   u16l_position;
  u16l_position=get_position();
  //省略了其他代码
}
```

```
void    run_control(void)
{
   //省略了其他代码
   motor_ drive ();
}
```

在 代码 2-13 中，motor_drive→get_position调用路径被拆分，改成在run_control函数中顺序调用这两个函数，把get_position的结果作为参数传递给motor_drive函数，这样，两个高消费栈函数共享了栈空间，这条路径所需要的栈空间降低到了 2K字节多一点点，只有 代码 2-12 的 50%。

代码 2-13 降低了栈消耗的运动控制程序

```
uint16_t    get_position(void)
{
   sint16_t    s16l_motor[1000];       //定义了大数组,属高消费栈函数
   //代码被省略
}

void    motor_drive(uint16_t    u16l_position)
{
   uint16_t    u16l_buf[1000];         //定义了大数组,属高消费栈函数
   //省略了具体代码
}

void run_control(void)
{
   uint16_t    u16l_position;
   //省略了其他代码
   u16l_position=get_position();
   motor_position(u16l_position);
}
```

2.12 windows 内存粉碎机

windows运行于x86 架构的计算机上，每个进程可以访问 4G地址空间，其中地址小于 0x80000000 的 2G空间是私有的，有完整的访问权限，只有进程自己能访问，其他进程不能访问。高 2G由操作系统使用，用于存放共享数据、代码、库以及操作系统代码，进程只能读。不管系统实际配置了多少内存，每个进程看起来都拥有 2G完整访问权限的内存空间，所需的内存都从这 2G空间范围内分配。代码 2-14 是一个典型的内存粉碎机程序，这段程序除了制造内存碎片和内存泄漏外，没有别的功能。本段代码在配置了 256M内存的P4 计算机上运行，用Borland C++ Builder6.0 编译，操作系统是windows2000。在随书光盘的winmem目录下有本程序的工程文件。

```
int main(int argc, char* argv[])
{
```

```
        unsigned int i=0,j=0,n=0;
        void * p1,* p2;
        char s1[20],s2[20];
        p2=(void* )malloc(0x20000000);              //1  试图分配 512M 内存
        if(p2!=NULL)
        {
                cout<<"可分配内存大于     0x"<<hex<<0x20000000<<" Bytes"<<endl;
                cout<<"释放掉刚分配的内存"<<endl;
                free((void*)p2);
        }
        p1=(void* )malloc(0x100000);                //2  分配 1M 内存
        while(p1!=NULL)
        {
                p2=(void* )malloc(0x10);            //3  分配小块内存
                cout<<"碎片  p1=0x"<<p1<<"--len=0x"<<hex<<(j)*100+0x100000<<
                        "--p2=0x"<<p2<<endl;
                free((void*)p1);                    //4  释放内存
                i+=0x100000+j*100;
                j++;
                p1=(void* )malloc(0x100000+j*100);  //5  申请一块更大的内存
        }
        cout<<"经过  " <<dec<< j <<"循环分配和释放内存后:"<<endl;
        cout << "应用程序共得到  0x10 Bytes 内存"<<endl;
        cout << "泄漏了 0x "<<hex<<(j-1)*0x10<<" Bytes 内存"<<endl;
        cout << "空闲内存总量约     0x"<<hex<<0x80000000-j*0x10<<" Bytes"<<endl;
        cout << "空闲内存被碎片化成" <<dec<< j-1 <<"块"<<endl;
        cout << "最大的碎片尺寸=0x"<<hex<< 0x100000+(j-1)*100<<"    Bytes"<<endl;
        return 0;
}                       //6  回收资源
碎片  p1=0x11a0004--len=0x100000--p2=0x12a0008
碎片  p1=0x12a001c--len=0x100064--p2=0x13a0084
……
碎片  p1=0x7fcde424--len=0x13a4d0--p2=0x7fe188f8
碎片  p1=0x7fe1890c--len=0x13a534--p2=0x7ff52e44
经过  2390 循环分配和释放内存后:
应用程序共得到  0x10 Bytes 内存
泄漏了 0x 9550 Bytes 内存
空闲内存总量约     0x7fff6aa0 Bytes
空闲内存被碎片化成 2389 块
最大的碎片尺寸=0x13a534    Bytes
```

代码 2-14　内存粉碎机程序

1. 程序的开头试图分配一块 512M 大小的内存，在仅配置 256M 内存的机器上一次分配 512M 内存并且分配成功。借助于 MMU 硬件功能，windows 分配内存时，并没有分配物理内存，仅仅是在进程的虚拟地址空间着色，表示被着色的地址空间已经

被使用。物理内存是在使用该块内存（读或者写）时才从物理空间映射到虚拟空间的，所以可以分配超过物理内存的内存块。

2. 申请一块 1M 的内存空间。
3. 申请一块小内存，下一次循环执行本行代码之前，并没有释放 p2，p2 又被一次新的分配覆盖，这就发生了内存丢失。
4. 释放上一次循环申请的一块大内存。
5. 再申请一块更大的内存，由于每次申请的内存都比刚释放的内存大，因此刚释放的内存总是不能被重新利用而成为内存碎片。碎片只存在于进程虚拟空间，而物理内存并没有因此而产生碎片。
6. 程序执行完成后，操作系统将回收所有资源，包括内存碎片、泄漏的内存和进程正在使用的内存。

图 2-11 显示了经内存粉碎机碾压后进程的内存视图，粉碎机进程实际只获得了 0x10 字节的有效内存，但是却轻而易举地制造了 0x9560 字节的内存泄漏，以及 2390 块内存碎片，使2G的进程空间消耗殆尽。这个例子告诉我们，一个不注意内存管理的程序的破坏力是如此的巨大，即使在每个进程拥有 2G私有地址空间的windows环境下，也如此轻而易举地把内存消耗殆尽，而最终只获得了区区 0x10 字节的内存。

windows支持独立进程空间，因此粉碎机进程耗尽的只是自己的私有内存，其他进程不受影响，操作系统也不受影响，粉碎机进程结束后，其所占用的所有内存都可以被回收再利用。在不支持进程独立地址空间的系统中，一个线程耗尽内存，就意味这整个系统的内存被耗尽，可能会造成系统瘫痪。djyos的内存管理在这方面做了一些改进，提供了一定的内存回收功能，参见第 4.3.11.3 节。

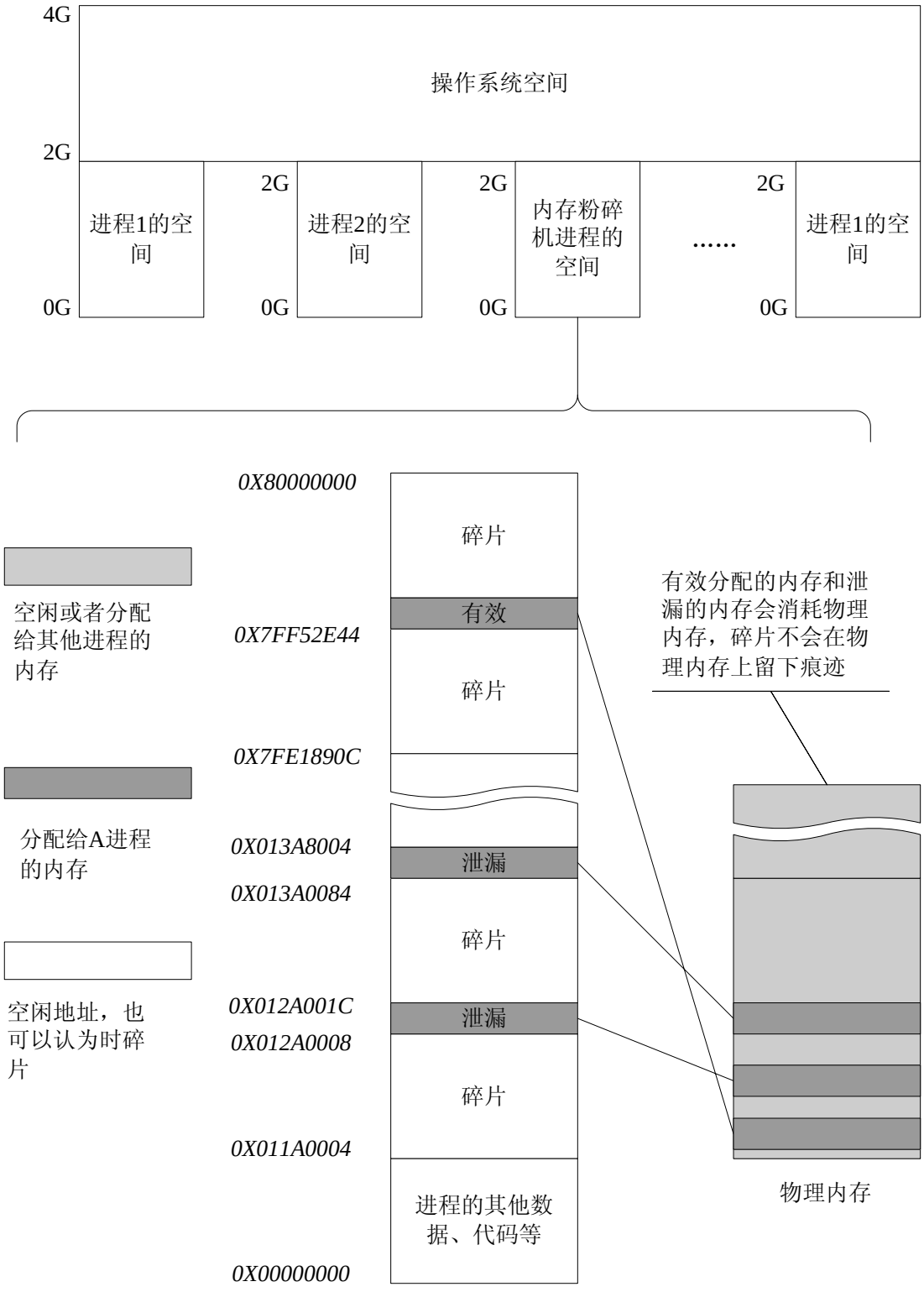


图 2-11 运行内存粉碎机后 windows 内存视图

2.13 数据对齐

数据对齐是对访问 C 语言基础类型的变量地址的一种约束，能被变量的尺寸（以字节数表示）整除的地址叫做对齐的地址，否则叫非对齐的地址。这是嵌入式系统开发中的一个

难点，也是一个高故障率点，再加上 C 语言无法无天的指针操作，它可能指向任何非对齐的地址，这个难点的难度被进一步放大，初学者甚至有一定经验的工程师很容易在这里出问题。特别是，x86 系 CPU 允许非对齐的地址，而大多数嵌入式 CPU 特别是 RISC 指令集的 CPU，访问非对齐的地址会发生致命错误，故从 PC 程序转型的嵌入式程序员，特别容易犯非对齐数据访问的错误。

约束源于计算机指令系统的要求，例如，ARM 加载或存储 32 位数据的指令

LDR　R1,[R0]　和

STR　R1,[R0]

包含在 R0 中的、存储一个 32 位数的地址应该是 4 的整数倍，否则就会导致数据访问出错，详情可参考 ARM 的指令手册，同样，LDRH 和 STRH 指令要求的地址是 2 的整数倍。

大多数 CPU 对访问多字节类型数据有对齐的要求，对程序员来说，具体对齐要求与所使用的 CPU 和编译器都有关系。值得一提的是动态分配内存的对齐，由于动态分配时，内存分配器并不知道用户怎样使用取得的内存，故需要做最严格的对齐，最严格的对齐与计算机体系结构有关，对于 ARM，最长的基础数据类型是 4 字节的，因此指令最严格的对齐要求是 4 字节对齐，但实际上 ARM 系统动态分配却是 8 字节对齐的，为什么呢？原来，ARM 的过程调用约定 ATPCS 中规定，子程序调用时，栈的起始地址应该是 8 字节对齐的，因此，如果动态分配的内存用于栈的话，则必须执行 8 字节对齐，所以，从整个体系结构来说，最严格的对齐要求不是 4 字节而是 8 字节。

大多数编译器提供 packed 关键字来改变编译器的默认对齐方式，若变量被定义到非对齐的地址，编译器就会额外产生大量的指令来拼装数据。比如一个 4 字节数据被定义在地址 101 处，当读这个数据时，是先读出地址 100 处的一个 4 字节数据右移 8 位，再读地址 104 处的一个字节扩展成 32 位后左移 24 位，然后把两次读出的经过移位后的数据相加，注意无论原来数据是有符号还是无符号，两次移位都用无符号移位（即不做符号扩展）。增加了 CPU 开销和代码尺寸，一般只在两种情况下使用非对齐数据：

1. 内存十分稀缺，而 CPU 的运算速度和代码存储器却有冗余，有些嵌入式系统尤其是成本敏感系统会有这种情况。
2. 访问固定格式的数据结构，比如通信数据包的解码、读取图形文件的数据。

有些程序员爱用集中分配内存然后再分配的方法优化程序，谨慎使用再分配算法，这样可能会因为对齐问题而导致bug或者潜在bug。看 代码 2-15，p8、p16 和p32 分别指向 3 块动态内存，get_size1/2/3 这三个函数能够计算三块动态内存的大小，然后连续调用malloc函数从堆中分配内存。

代码 2-15 初始化时分配永久内存

```
char *p8;
uint16_t *p162;
uint32_t *p32;
void main(void)
{
    size_t    m1,m2,m3;
    m1 = get_size1( );              //计算 m1 内存需求，结果等于 103；
    m2 = get_size2( );              //计算 m2 内存需求，结果等于 108；
    m3 = get_size3( );              //计算 m3 内存需求，结果等于 1000；
    p8 = malloc(m1);
    p16 = malloc(m2);
    p32 = malloc(m3);
```

```
        if((p8 == NULL) || (p16 == NULL) || (p32 == NULL))
            goto save_return;
        ....
save_return:
        if(p8 != NULL)      free(p8);
        if(p16 != NULL)     free(p16);
        if(p32 != NULL)     free(p32);
}
```

细心的读者一定会发现，连续三次调用malloc函数要花费可观的执行时间，能不能象 代码 2-16 那样优化呢，只调用一次malloc函数，从而节省程序执行时间。实际上是不允许的，代码 2-15 和 代码 2-16 的执行结果并不一致，malloc函数考虑了存储器访问边界对齐的问题，在gcc-arm编译环境中，它执行的是 8 字节对齐，即所有动态分配的内存的起始地址都是 8 的整数倍，假设p_heap_bottom＝0x10000，那么 代码 2-15 执行后的结果为：p8=0x10000，p16=0x10068， p32=0x100d8。而 代码 2-16 没有考虑对齐，执行后有p8=0x10000，p16=0x10067，p32=0x100d3，应用程序访问*p16 和*p32 时，就会发生意想不到的错误。

代码 2-16 不恰当地使用再分配方法

```
void main(void)
{
    size_t    m1,m2,m3;
    m1 = get_size1;                 //计算 m1 内存需求，假设等于 103；
    m2 = get_size2;                 //计算 m2 内存需求，假设等于 107；
    m3 = get_size3;                 //计算 m3 内存需求，假设等于 1000；
    p8 = m_static_malloc(m1+m2+m3);     //一次分配全部内存需求
    if(p8 == NULL)      free(p8);
    p16 = (size_t)p1+m1;
    p32 = (size_t)p2+m2;
}
```

另一种典型的再分配有关的bug产生在为结构类型分配内存中，我们来看看 代码 2-17，显然，sizeof(struct a)的值是 6，sizeof(struct b)的值是 12，malloc执行 8 字节对齐，故st6bytes指针的值是以 8 字节的整数倍，struct a的成员最大是 2 字节的，访问struct a要求使用 2 字节对齐的地址，所以st6bytes是符合要求的，通过st6bytes可以正确地读写struct a。而sturct b的成员是 4 字节的，要求使用 4 字节对齐的地址，st8bytes = st6bytes+sizeof(struct a)却注定不是 4 的整数倍，使用st8bytes指针并不能正确地访问sturct b。

更可怕的是，这种由内存对齐产生的bug往往只是一个病原体，而不是一个明确的bug，只要进行充分测试，明确的bug可以在产品测试中被发现并纠正。而病原体就不同，它只在特定条件下发病。如果把 代码 2-17 中被注释掉的语句加上，则sizeof(struct a)的值变为 12，从而使st8bytes的地址碰巧等于 4 的整数倍，struct b将可以正确地操作，除了仔细阅读代码外，其他任何测试都不能发现这个问题，这个病原体是否发病将与struct a尺寸有关。这样，有缺陷的程序将随产品一起发布、含病原体的代码将与项目其他文档一起存档。当你在维护产品或者升级产品、抑或下一个项目需要使用这个模块时，可能会修改sturct a的定义，潘多拉魔盒就可能被打开，这时候，你想哭都没有眼泪！首先，由于本次修改根本就没有涉及struct b，你根本不会去怀疑问题与本次修改有关；其次，这是一段已经经过测试并且应用于产品的模块，开始时你根本就不会想到问题是出在这个地方，你将为这个问题走一大圈子的弯路；第三，这个模块的原作者可能已经调职，谁都知道，分析别人写的代码，要走的路比由作者

自己分析要多得多。

代码 2-17 再分配与结构对齐错误

```
struct a
{
    uint16_t    a,b,c;
//      uint32_t    d;              //bug 是否发作依赖本句是否加上
} *st6bytes;
struct b
{
    uint32_t    a,b,c;
} *st8bytes;
void main(void)
{
    st6bytes = malloc(sizeof (struct a)+sizeof(struct b));
    if(st6bytes == NULL)        free(st6bytes);
    st8bytes = st6bytes + 1;
}
```

由于动态分配的内存一般是用指针操作，不严谨的指针操作是潜在的内存对齐错误。在C语言中，指针的使用是如此的广泛，一不注意就会陷入数据对齐错误的陷阱。这些错误源于对指针的直接赋值，如果赋给指针的地址与指针的类型所要求的对齐边界不相符的话，就会产生bug。这些bug是如此的难于捉摸，即使你打开所有的编译器告警选项，也不会帮助你在编译阶段发现错误。曾经有一个程序员声称他的程序得了“神经病”，同一段代码，有时候正常，有时候不正常，正常的时候总是正常，异常的时候总是异常。只要改变一下编译优化级别，或者把函数的书写顺序改变一下，或者添加和删除一些无关代码，程序就可能从正常变成异常，或者反之。经查发现程序在用指针访问变量时发生了对齐紊乱现象，代码 2-18 是笔者根据这个问题抽象的，在编译时不会有警告，但在执行时，可能会发生错误，由于数据对齐的问题，执行结果将不确定且是错误的。在嵌入式环境中，执行结果的不确定性将是致命的，而且不一致的执行结果也增加了排查bug的难度。由于排查这类bug非常困难，因此内存对齐相关的bug经常随着产品一起发布给用户。

代码 2-18 指针与数据对齐

```
int main(void)
{
    uint32_t    u32l_c,*pl_pt;
//  uint16_t    u16l_ab;                                    ①
    uint16_t    u16l_a=0,u16l_b=0;
    pl_pt=&u16l_a;                                          ②
    *pl_pt=0x1234;                                          ③
    u32l_c=*pl_pt;                                          ④
    pl_pt=&u16l_b;                                          ⑤
    *pl_pt=0x4567;                                          ⑥
    u32l_c=*pl_pt;                                          ⑦
//      pl_pt=&u16l_ab;     //使用一下 u16l_ab，以免给优化掉
    return 0;
}
```

执行步骤	u16l_a	&u16l_a	u16l_b	&u16l_b	u32l_c	pl_pt
②	0	0XC7FDDDE	0	0XC7FDDDC	0	0XC7FDDDE
③	不变	不变	0X1234	不变	不变	不变
	希望更新 u16l_a，却更新了 u16l_b。因为*pl_pt 是 32 位变量，CPU 在写 32 位数据时忽略最低两位地址，因此 0XC7FDDDE＝0XC7FDDDC，而 0XC7FDDDC 正好是变量 u16l_b 的地址。					
④	不变	不变	不变	不变	0X12340000	不变
	ARM7 CPU 在读 32 位数据时，对齐规则与写时不一样，是把当前地址处的 16 位数据作为结果的高 16 位（即 0X1234），再把倒数第二位地址取反后作为结果的低 16 位数据的地址(即 0000)。因此有 u32l_c＝0X12340000。					
⑤	不变	不变	不变	不变	不变	0XC7FDDDC
⑥	不变	不变	0X4567	不变	不变	不变
	这里 u16l_b 获得了期待的结果，因为 u16l_b 正好处在 4 字节对齐的边界上，按照对齐规则，确实如此.					
⑦	不变	不变	不变	不变	0X00004567	不变
	同样因为 pl_pt 的值正好处于 4 字节对齐的边界上，所以 u32l_c 获得了期待值。					
注释掉的代码加入后的结果如下						
	u16l_a	&u16l_a	u16l_b	&u16l_b	u32l_c	pl_pt
②	0	0XC7FDDDC	0	0XC7FDDDA	0	0XC7FDDDC
③	0X1234	不变	不变	不变	不变	不变
	u16l_a 刚好处于 4 字节对齐的边界。					
④	不变	不变	不变	不变	0X00001234	
⑤	不变	不变	不变	不变	不变	0XC7FDDDA
⑥	不变	不变	不变	**不变**	不变	不变
	仅仅是新定义了一个变量，就出现新问题了。执行本行后，所有变量均保持不变，0x4567 实际上被写入 0XC7FDDD8 地址处，该地址不知是何用途，意外的修改可能引发不可预料的后果。					
⑦	不变	不变	不变	不变	0X45670000	不变
	原来出现在④的问题现在出现在⑦，癌症转移了！					

表格 2-2 在ARM7 上执行 代码 2-18 的结果

表格 2-2 表明，同样的代码，变量的实际地址不一样的时候，程序执行结果是不一样的，的确有点象患了精神病的症状。在C语言环境中，函数定义的顺序，代码量的增减，编译优化级别的改变，以及变量定义的数量，均可以改变变量的实际地址，从而使程序执行结果变得捉摸不定。

解决上述对齐问题的方法其实很简单，只要正确地使用C语言的强制类型转换就可以了，把 代码 2-18 的 3、4、6、7 行做如下修改以后，就可以得到正确的结果。

```
*(uint16_t*)pl_pt=0x1234;                          ③
u32l_c=(uint32_t)*(uint16_t*)pl_pt;                ④
*(uint16_t*)pl_pt=0x4567;                          ⑥
u32l_c=(uint32_t)* (uint16_t*)pl_pt;               ⑦
```

当然，这仅仅是为了说明问题而假设的一段代码，烦琐的类型转换本身就是 bug 的来源，并且增加阅读难度，简洁易读的代码对减少 bug 数量非常有效。

这是在ARM7 上的执行结果，编译器是gcc4.10，在其他CPU或者编译器上，结果可能不一样，取决于CPU访问非对齐数据的具体方式，但无论如何，代码 2-18 肯定不会给出正确的运行结果。

动态内存分配时，由于内存分配器不知道用户使用动态分配的内存时要求的对齐方式，对被分配的内存地址要做最严格的对齐。djyos 系统提供了一组宏来实现对齐操作，gcc-arm 下，最严格的对齐是 8 字节对齐，移植 djyos 系统到其他“CPU-编译器”体系时，只需要修改 align_down 和 align_up 两个宏就可以了。

```
#define align_down_2(x)     ((x)&(~1))        //2 字节,向下对齐,
#define align_down_4(x)     ((x)&(~3))        //4 字节,向下对齐,
#define align_down_8(x)     ((x)&(~7))        //8 字节,向下对齐,
#define align_down_16(x)    ((x)&(~15))       //16 字节,向下对齐,
#define align_up_2(x)       (((x)+1)&(~1))    //2 字节,向上对齐,
#define align_up_4(x)       (((x)+3)&(~3))    //4 字节,向上对齐,
#define align_up_8(x)       (((x)+7)&(~7))    //8 字节,向上对齐,
#define align_up_16(x)      (((x)+15)&(~15))  //16 字节,向上对齐,

#define align_down(x)       align_down_8(x)   //gcc-arm 要求 8 字节对齐
#define align_up(x)         align_up_8(x)     //gcc-arm 要求 8 字节对齐
```

第3章 嵌入式实时操作系统基础

3.1 实时系统

都说嵌入式系统大多有实时性的要求，但什么是实时系统呢，业界对实时系统的定义有很多，IEEE(美国 电气工程师协会)对实时系统的定义是“那些正确性不仅取决于计算的逻辑结果也取决于产生结果所花费时间的系统”。

众所周知，应用很广的 uclinux、wince 都是非实时操作系统，而 vxworks、psos 等是实时操作系统，但不能安装第三方应用程序。本节还将探讨，前者为什么不能做成实时系统，而后者为什么不能安装第三方应用程序。

3.1.1 实时系统的实时性指标

Greg Bollella ，是 Sun 公司的一个杰出的工程师，实时 JAVA 规范的作者之一，他这样解释实时性“能够可靠的可预测的推测和控制程序逻辑的时间行为的能力。”实时并不像许多开发者想的那样，意味着速度快，而是意味着当需要对现实世界的事件作出反应时，它的行为是可预测的和可靠的，并且满足现实世界的时限要求的。实时的电脑总是在最后时限到来之前完成规定的操作，非实时系统则可能在大多数时候比实时系统快，却保证不了所有时候都能满足最后时限要求。通俗地说，实时性就是符合用户对正确地完成操作的时间要求的特性，实时性更高的系统能适应更苛刻的时间要求。

可见，衡量一个系统能不能达到产品实时性的要求，就要计算在最坏情况下，用户任务的完成时间能不能符合现实世界的要求，跟很多因素有关，操作系统、用户代码、硬件等都有关系。在操作系统支持下的嵌入式产品开发中，构成用户任务完成时间的主要因素如 表 3-1 所列。

表 3-1 影响任务完成时间的因素

实时需求 Trt	
任务要求完成时间 Trt	从任务发起时算起到计算机处理完事件并做出正确反应的时间，在嵌入式系统中，任务发起一般就是某事件发生。例如在机器人控制中，机器人行进时，从障碍物出现到实际作出避让动作的时间。
CPU 和操作系统因素 Tsys	
中断响应时间 Tii	从 PC 跳转到中断向量地址到开始执行用户定义的中断处理函数的时间间隔
中断返回时间 Tio	从用户的中断处理函数返回到 PC 返回到被中断的线程之间的时间间隔。
Tick 中断执行时间 Tt	基于 Tick 的操作系统都有 Tick 中断，为操作系统提供定时操作。
系统调度时间 Ts	高优先级的事件就绪到完成上下文切换的时间，例如控制机器人行进时自动避让障碍物的系统中，检测程序发现障碍物信息后将发出相应的事件，从发出事件到开始切换到避让程序的时间即 Ts。

操作系统连续禁止中断的时间 Tsi	多任务的并发执行会导致临界资源的保护问题，操作系统在访问临界资源时可能会禁止中断。
操作系统连续禁止调度的时间 Tsc	同上，也操作系统也可能会禁止线程调度。
系统调用的执行时间 Tsa	应用程序可能需要执行系统调用，操作系统提供的系统调用的执行时间也是影响实时性的重要因素。
运行环境切换时间 Tc	即保存正在执行的线程的上下文，然后恢复新线程的上下文所花的时间。
硬件因素 Thard	
硬件响应中断时间 Tih	中断模式下，从硬件发出中断请求到程序跳转到中断向量地址的时间
硬件执行时间 Te	从 CPU 发出执行指令到物理部件完成操作的时间，例如在机器人行进控制中，CPU 发出避让指令后转向电机实际完成转向动作的时间。
硬件检测时间 Tg	物理事件发生到传感器感知并完成数字转换的时间，在机器人自动避让障碍物的例子中，从障碍物出现到机器人“眼睛”拍摄到障碍物图像，然后转换成 CPU 可以读取图像数据并通知 CPU 读取数据的时间，
用户软件因素 Tuser	
软件灵敏度 Tm	从外部信号到达 CPU 到 CPU 读取到信号的时间。例如机器人控制系统中，障碍物信息到达 CPU 引脚到 CPU 采样到障碍物信息的时间为 Tm。如果是轮询实现，Tm=最长轮询时间；如果用中断实现，则 Tm＝Tih。
被高优先级线程抢占的时间 Tp	用户线程执行过程中如果被更高优先级的线程抢占，显然被抢占时间要算到任务的总完成时间里面。由于高优先级的线程是由用户启动的，应该算做用户软件因素。
用户连续禁止中断的时间 Tui	用户为保护临界资源也可能会禁止中断，由于中断服务期间也是禁止调度的，所以 Tui 还包括执行中断服务函数的持续时间。
用户连续禁止调度的时间 Tuc	用户为保护临界资源也可能需要禁止线程调度
用户线程被中断函数中断的时间 Ti	如果用户线程执行过程中发生了中断，显然执行中断函数会延缓用户任务的完成时间。
用户处理事件的代码执行时间 Tu	在不被高优先级线程抢占，也不被中断处理函数打断的条件下，用户的事件处理程序开始执行到完成任务所需要的时间，这个时间操作系统不能控制。如果代码中执行了系统调用，则 Tu 还包含了所有系统调用的执行时间之和。

注：表中有几个地方出现连续关中断的时间和连续禁止调度的时间，当高优先级线程就绪时，无论是关中断还是关调度，上下文都不会切换到高优先级线程，但只要有瞬间开放中断或者调度，就会引起上下文切换，所以必须强调“连续”两字。

实现用户任务的方法不同，计算任务完成时间的方法也不同，略举几种典型情况如下：

1. 用中断实现的任务，温度控制系统中，硬件检测到温度超限后，在 CPU 的外部中断引脚上产生一个中断信号，CPU 在中断函数中停止加热棒。用户任务的执行时间应该是从温度超限到加热棒停止工作的时间，计算方法如下：

   Time＝Tg+Tih+Tii+Tsi+Tsa+Tui+Ti+Te

- Tg 在这里是传感器检测温度超限的时间。
- Tih、Tii、Tsi、Tsa、Tui如 表 3-1 所示。
- Te 是 CPU 发出停止加热命令到加热棒实际停止工作的时间。

2. 与硬件信号无关的软件内部事件，用高优先级的线程实现的任务，例如通信程序接收到数据，发事件给通信协议处理程序，然后由协议处理程序处理该数据包。用户任务的执行时间应该是从发事件开始到协议处理程序完成数据处理之间的事件间隔。在整个过程不被中断程序（包括 Tick 中断）打断的情况下，计算方法：

   Time=Ts+Tsa+Tc+Tu+Tp

3. 由硬件检测外部事件，然后用线程处理的方式实现的任务，例如机器人行进中自动避让障碍物，障碍物出现后，硬件将给 CPU 的外部中断引脚一个中断信号，中断函数发出“发现障碍物”事件，事件处理函数识别障碍物的类型、方位，并给机器人运动装置发出转向命令。那么用户任务的执行时间是从障碍物出现到机器人实际做出避让的时间间隔。计算方法：

   Time= Tg+Tih+Tii+Tsi+Tsa+Tui+Ti+ Ts+Tc+Tu +Te+Tp

   - Tg 在这里是传感器检测障碍物的时间。
   - Te 是 CPU 发出转向命令给动力装置到机器人实际做出避让动作的时间。

嵌入式系统满足应用实时性要求的条件是Time<Trt。表 3-1 给出了计算操作系统和软硬件设计方案是否满足需求中的实时性要求的定量计算模型。

表 3-1 中所列出的各时间量，并不能简单地得到，这些时间与系统运行的实际状态有关，因此操作系统一般只会给出在特定平台上最坏情况和最好情况下的时间，以及平均时间。“实时操作系统”这个名字有时候会给人一个错误的印象，好像应用系统的实时性完全是由操作系统保证似的，而忽略用户软件和硬件的因素。事实上，使用RTOS仅仅意味着你可以预测在最坏情况下的响应，而使用非实时操作系统无法预测最坏情况下的响应。使用实时操作系统并不能保证你能实现一个实时系统，执行时间Time不但与操作系统有关，还与用户软件有关。如果你滥用关中断，或者不正确地使用中断的话，你的系统的实时性将非常糟糕。笔者曾经见过一个系统，这个系统在中断函数里使用循环等待一个硬件状态标志的出现，在正常情况下，该循环至少需要 1mS才能结束，如果硬件异常，循环等待时间将更长，我们知道，中断的优先级比最高优先级的线程还要高，在中断服务函数里做无谓的等待，等于降低里系统的实时性能。实时性好的操作系统Tsys较短且可预测，为用户因素Tuser和硬件因素Thard保留了较大的空间。实时系统的实现是操作系统和用户软件设计共同实现的，表 3-1 中即使是操作系统因素Tsys，也与用户设置有关。一般来说，操作系统操作事件队列的过程中必须关中断，而队列越长，关中断的时间就越长。用户应该小心翼翼地设计事件队列和分配优先级，使事件能够及时处理，避免积压在事件队列中。

为了满足用户实时性的要求，因此所有 RTOS 都小心翼翼地设计，尽量缩短操作系统运行中可以影响系统实时性的时间，所以 RTOS 有一些显著的特点：

1. RTOS 是抢占式的，高优先级的线程就绪时，只要调度是允许的，不管低优先级的线程是否愿意出让 CPU，都会被强制出让而把执行权利交给高优先级的线程。
2. 高优先级的线程正在处理时，除非自己主动放弃 CPU，是决不会被低优先级的任务打断的。
3. 线程是否会被同优先级的线程抢占与 RTOS 的具体实现有关。支持时间片轮转的 RTOS 时间到时会被抢占，如 Vxworks，不支持的不会被抢占，如 ucosii。
4. RTOS 忠实地执行程序员设定的线程优先级，绝不会擅自改变线程优先级。
5. 高优先级线程从就绪到正式投入运行的等待时间非常短，常常是微秒级的，这个等待时间的长短是衡量一个操作系统性能的重要指标。

6. RTOS 会竭尽所能地缩短连续关中断的时间 Tsi 和连续关调度的时间 Tsc，即使因此降低效率也在所不惜。例如要处理一个由多个结点组成的队列，处理结点时需要关中断，那么，在关闭中断的情况下连续处理整个队列是最高效的，但这样会增加 Tsi；如果每处理一个结点开一下中断，然后关闭中断继续处理后续结点，则会大大降低效率，但是连续关中断的时间 Tsi 却大大减小了，等同于提高了系统实时性。非实时操作系统往往采用前者以提高效率，RTOS 中往往采用后者以提高实时性，djyos 操作系统使用了后者来提高实时性。

实现嵌入式产品实时性需求，还跟用户系统软硬件设计有关，产品设计中还应该注意：

1. 中断服务函数和高优先级线程应该尽量精简，在中断服务函数或者优先级非常高的线程中只执行时间要求很紧迫的工作，不要做无关要紧的工作，尤其尽可能不要延时或者循环等待。
2. 仔细分配线程优先级，使真正紧迫的工作能够及时得到服务，如果系统比较复杂，这是一项具有相当挑战性的工作，合理的优先级分配能够提高系统响应高优先级线程的速度，还可以防止上下文反复切换造成系统性能下降，避免优先级翻转导致消耗大量系统资源。在实践中，优先级往往需要反复调整和实验才能确定。
3. 仔细设计临界资源以及访问临界资源的代码，尽量减少连续关中断和连续关调度的时间。即使关中断不可避免，也要尽量使用间歇性地开关中断，不要连续关中断，对关调度也是一样。
4. 设法降低系统实时性需求以降低设计难度。使用缓冲技术可以有效降低系统实时性要求，软件和硬件都可以实现缓冲。比如在通信程序中，必须在接收缓冲器溢出前读取数据，否则就会丢失数据。如果通信设备的硬件有比较大的缓冲器，显然接收任务的实时性要求就会降低；或者用在软件中设置大数组做缓冲区，在通信中断服务函数里把数据读到缓冲区中，也可以降低通信接收任务的实时性要求。
5. 合理分配软件和硬件实时性指标，使用高性能的硬件，可以加速硬件检测和执行时间，这样也可以使软件有更充足的时间处理，降低软件实时性要求；反过来，实时性更高的软件设计，可以降低硬件需求，从而降低硬件实现难度，降低成本。

3.1.2 非实时操作系统多道程序设计

多道程序设计技术是在计算机内存中同时存放几道相互独立的程序，使它们在管理程序控制下，相互穿插运行。当某道程序因某种原因不能继续运行下去时候，管理程序就将另一道程序投入运行，这样使几道程序在系统内并发工作，可使中央处理机及外设尽量处于忙碌状态，从而大大提高计算机使用效率。现代操作系统实现多道程序设计技术，为应用程序提供了一个虚拟环境，每个应用程序看起来都在独立的机器上执行，操作系统的几乎所有技术难点都与实现多道程序设计有关。整台计算机上的所有软件和操作系统加在一起，其实可以看做一个大程序，只是这个程序的不同部分可能由互相没有任何业务联系的公司或团体编写，而这些由不同团队开发的代码，在操作系统的协调下自由顺畅地运行，这就是多少软件精英为之着迷的神奇的操作系统!安装了 windows 操作系统的计算机上，来自澳大利亚 Protel 公司的 protel 电路设计软件和来自美国 Adobe 公司的 Acrobat 图文阅读软件在同一台计算机上同时运行，互不干扰，只要计算机足够快，内存足够多，这两个程序之间根本感觉不到对方的存在，这是因为操作系统分别为他们提供了一个虚拟环境，使他们各自看起来都“拥有”一台完整的计算机，其他程序的存在并不破坏这台“计算机”的完整性。在非实时系统中，多道程序设计多以多进程技术实现，操作系统管理各进程，根据进程的具体特征和执行状态，动态地调整进程使用 cpu 的时间，使各进程尽可能公平地享用 CPU，既不会使任意一个进

程长期处于饥饿状态，也不会让一个进程显失公平地占用大量 CPU 时间。一句话，尽可能公平地干尽可能多的活，就是非实时操作系统设计者追求的目标，一般来说，非实时系统可以用下面的 for 循环来描述其调度过程。

```
while(1)
 {
    for(系统中所有进程)
    {
        收集并分析进程的特征；
        重新分配 CPU 时间和进程的优先级；
    }
    进程调度；
}
```

不同的操作系统，无论其代码表达上有多大的差别，它的调度器逻辑上都会有上述执行过程，其差别在于重新分配各进程 CPU 时间的具体算法不同。

非实时操作系统的核心特征之一是，任何进程无权全部霸占 CPU 时间，即使你有十万火急的情况急需处理也得耐心等待。不管哪一道程序，只要胆敢长期霸占 CPU，操作系统调度程序就会毫不犹豫地把它推上绞架——剥夺它的 CPU 时间，以使其他道程序能够得到运行。这就使得任何软件设计者无须与其他设计者沟通，也无须担心自己设计的程序得不到运行时间。这也使得用户可以随时安装和卸载程序，随时启动一个新程序，或停止一个程序，只要计算机足够快，就无须担心这些程序在利用 cpu 时间方面会互相冲突。

3.1.3实时操作系统多道程序设计

从上节可知，多道程序设计实现的关键点是，操作系统的调度器拥有剥夺一道程序的 CPU 使用权的权力，不管该道程序是否情愿让出 CPU。然而，这种策略使得任何软件设计者都不知道自己的程序何时会被剥夺 CPU 使用权，以及被剥夺多长时间，导致实时性的核心——确定性就无从谈起，因此，实时操作系统应该使程序员能够控制和预测自己的程序何时获得 CPU，何时被剥夺 CPU。与非实时系统相比，RTOS 的最显著的区别在于实现多道程序设计上。非实时系统采用动态优先级管理，允许操作系统通过调整优先级来协调线程共享 CPU，在操作系统认为必要时强行中断一道程序，把 CPU 交给另一道程序，可以实现完全互相独立的多个程序同时运行，设计这些程序的团体无需为和平共享 CPU 而操心，因此互相之间可以没有任何联系。但是，这种调度策略在实时系统中将造成灾难性的后果，一个程序运行时必须保证以下条件，才能确保时间确定性：

1. 当任务发起后，应该在有限而且可知的时间内得到调度程序的关照，获得 CPU 的使用权。
2. 该道程序运行时不被打断，或者它能够预测到被打断的次数和每次时间长度，以期使它能够在设计时就可以确定自己被完成的最后时限。

实时 RTOS 为了实现可预测性指标，一般采用静态优先级策略，使得操作系统不能调整线程的优先级，否则将带来灾难性的后果，例如汽车的 ABS 控制系统中，控制程序必须有很高的优先级，当车轮打滑时，刹车控制程序不能因为优先级被调低而拒绝动作。当一个低优先级的线程处于严重饥饿状态时，就算高优先级的线程在空转，操作系统也决不会干预。实时系统的多道程序设计虽然允许同时运行多个程序，但是这些程序必须自己互相协调以共享 CPU 以及其他资源，合理分配线程优先级是最重要的协调手段。互相没有联系的软件开发团体，自然也不能在一起协商线程优先级分配，他们开发的软件即使能同时安装，也不能

有效地共享 CPU。

图 3-1 显示了两个互相独立的软件如果同时安装到实时操作系统中的情形。

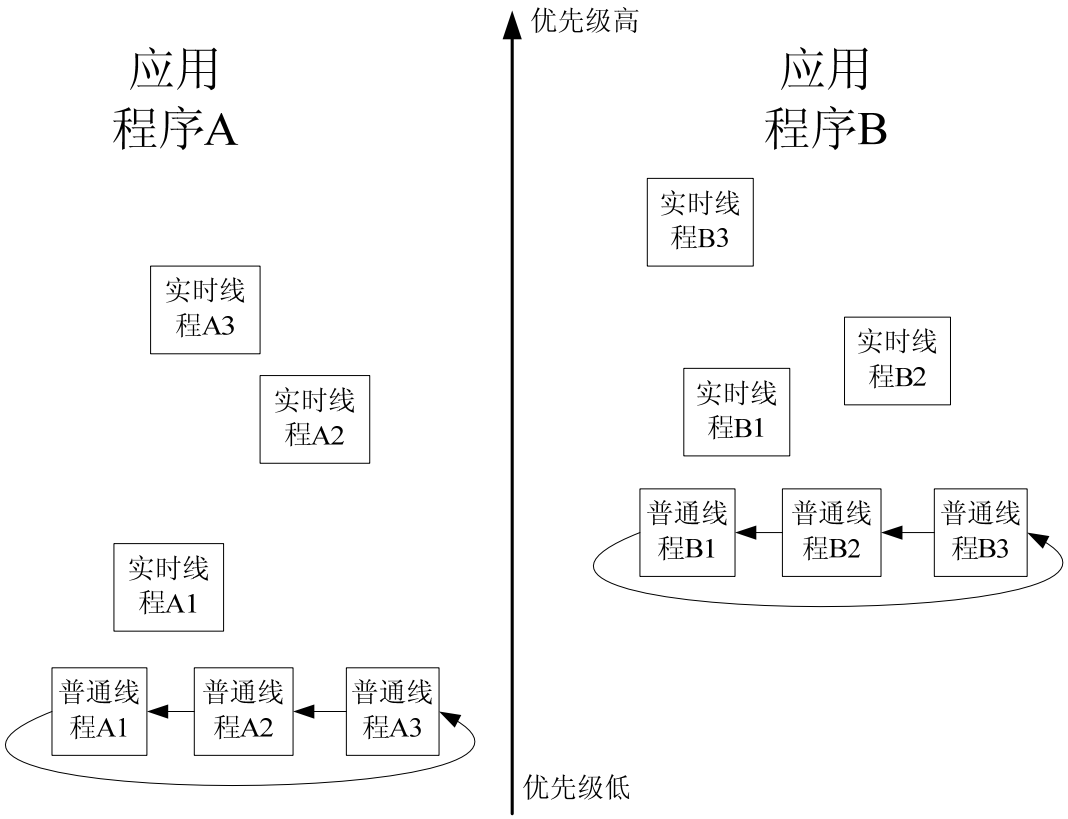


图 3-1 在实时操作系统中同时安装两个完全无关的应用程序

1. 图中左侧和右侧分别是程序 A 和 B 单独运行的情景。
2. 两个软件分别由 3 个普通线程和 3 个实时线程组成，普通线程使用有相同的优先级，它总是就绪，按时间片轮转调度，实时线程用于响应外部事件，优先级高于普通线程。
3. 两个软件独立开发，开发团队无法沟通，独立选择优先级，选择优先级的方法完全按自己的喜好进行，结果应用程序 B 的普通线程的优先级居然高于应用程序 A 的实时线程 A1。
4. 普通线程使用轮转调度，它总是就绪，也就是说，一旦应用程序 B 被启动，它的普通线程就会立即抢占应用程序 A 的普通线程和实时线程 A1，则应用程序 A 的普通线程和实时线程 1 完全没有机会运行。

因此，RTOS 的多道程序设计支持是有限制的，它虽然能实现多个程序并发地执行，但要求系统中所有应用程序互相协作，这就要求所有程序都是由同一个团体开发的，或者是团队之间有紧密沟通的情况。项目中可能会使用一些中间件，例如 GUI、TCP/IP 模块等，虽然这些中间件独立开发的，但他们并不是独立运行的程序，它只是项目中的功能模块而已。非实时的 linux 操作系统同时提供了非实时调度策略和软实时调度策略，在调度非实时优先级的进程时，内核会干预各进程的行为，惩罚疯狂占用 CPU 的进程，降低它的优先级，奖励不太占用 CPU 的 IO 类型进程，提高它的优先级，不会让高优先级的进程得到过多的运行时间，也不会让低优先级进程过于饥饿。但如果进程拥有实时优先级，它就会执行实时调度策略，内核就当起了撒手掌柜，只是简单地把 CPU 交给高优先级的就绪进程，直到地老天荒，只有天知道该进程什么时候会主动出让 CPU。可见，仅仅为了实现软实时，linux 就必

须使用静态优先级，更何况真正的实时系统呢？因此，在 linux 操作系统上编程时，赋予某进程实时优先级要非常谨慎，因为所有实时优先级都高于非实时优先级，只要有一个实时优先级的进程就绪且不主动让步（让出 CPU），后果只有一个——死机。实时优先级的进程要尽快完成任务，完成后及时出让 CPU。当然，也不能就此认为实时调度策略比非实时策略简单，为了使“完成操作的时间是可预测的和可靠的”，为了提高系统的实时性尤其是最坏情况下的性能，大大增加了实时操作系统的设计难度，即使野心勃勃的 Linux 开发者们只能在 Linux 中添加软实时调度算法。软实时算法可以形象地描述：让应用程序设计者决定谁能优先得到 CPU 以及占用多长 CPU 时间，但操作系统不保证你什么时候能够得到 CPU；应用程序运行过程中，操作系统尽量让你拥有完整的 CPU 时间，但不保证不打断你的执行，也不确保你被打断的时间长度；如果你需要系统服务，操作系统只能尽快为你服务，但不保证能及时提供服务，也不确保在最后时限到来之前完成服务。

在桌面系统上写程序时，我们常常使用下面的代码，在没有操作系统支持的嵌入式环境中，绝大多数程序也是这样写的。

```
int main(void)
{
    定义全局变量；
    初始化；
    while(1)
     {
          ……；
      }
}
```

在整个 while 循环中，没有任何可能引起阻塞的语句也是可以的，线程一旦获得 CPU，就会疯狂地抓住 CPU，即使没有任何有意义的事情可做也决不出让 CPU。这样的程序在非实时系统中是可以的，操作系统会惩罚这种不道德的行为，自动降低这个进程的优先级，并减少分配给它的 CPU 时间，从而让其他称序能得到运行。当然，即使在非实时操作系统中，我们也不提倡这样写程序，这样会大大降低计算机的效率，总之这是一个不好的编程习惯，谁也不愿意买台计算机摆在那里白跑。

而在 RTOS 中，这样的程序是致命的错误，会使比它低优先级的线程永远得不到运行的机会，因为 RTOS 中高优先级的线程在执行时，系统是绝对禁止低优先级的线程执行的。所以在 RTOS 中编写不同模块的程序员要互相协作，合理设定自己的线程优先级，高优先级的事件处理时要适时地让出 CPU，不要仗（优先级高之）势欺人。实际开发过程中中，一般由项目技术主管统一分配各任务的资源和优先级。正确的 RTOS 程序应该是这样的

```
int main(void)
{
    定义全局变量；
    初始化；
    while(1)
    {
          完成规定的事务；
          调用会引起阻塞的代码；
    }
}
```

线程阻塞时，就会退出运行，而无论线程以何种方式退出运行，必然是在等待某一个条

件的到来，该条件达到后，操作系统将会唤醒休眠的线程，一个没有结束条件的休眠虽然不会造成什么直接的恶果，但那是毫无意义的。在循环外，应该有在适当条件下重新唤醒线程的代码，或者在其他线程运行时，或者在中断处理程序里，必须保证唤醒线程的代码能执行到，否则，线程就会变成僵尸线程，白白占用内存空间。djyos 操作系统中，不允许无缘无故地阻塞线程，所有会引起阻塞的代码，都必须显式指定唤醒条件。

参考资料，摘自参考文献【7】第 46 页。

Linux 提供了两种实时调度策略：SCHED_FIFO 和 SCHED_RR。而普通的、非实时的调度策略是 SCHED_NORMAL。SCHED_FIFO 实现了一种简单的、先入先出的调度算法，它不使用时间片。SCHED_FIFO 级的进程会比任何 SCHED_NORMAL 级的进程都先得到调度。一旦一个 SCHED_FIFO 级进程处于可执行状态，就会一直执行，直到它自己受阻塞或显式地释放处理器位置；它不基于时间片，可以一直执行下去。只有较高优先级的 SCHED_FIFO 或者 SCHED_RR 任务才能抢占 SCHED_FIFO 任务。如果有两个或者更多的（同级的，本书作者注）SCHED_FIFO 级进程，他们会轮流执行，但是在他们愿意让出处理机时会再次让出。只要有 SCHED_FIFO 级进程在执行，其他级别较低的进程就只能等待它结束后才有机会执行。

SCHED_RR 与 SCHED_FIFO 大体相同，只是 SCHED_RR 级的进程在耗尽事先分配给它的时间后就不能再接着执行了。也就是说，SCHED_RR 是带有时间片的 SCHED_FIFO——这是一种实时轮流调度算法。当 SCHED_RR 任务耗尽它的时间片，在同一优先级的其他实时进程被轮流调度。时间片只用来重新调度同一优先级的进程。对于 SCHED_FIFO 进程，高优先级总是立即抢占低优先级，但低优先级进程决不能抢占 SCHED_RR 任务，即使它的时间片耗尽。

这两种实时算法实现的都是静态优先级。内核不为实施进程计算动态优先级。这能保证给定优先级别的实时进程总能抢占优先级比它低的进程。

Linux 的实时调度算法提供了一种软实时工作方式。软实时的含义是，内核调度进程，尽力使进程在它限定的时间到来前运行，但内核不保证总能满足这些进程的要求。相反，硬实时系统保证在一定条件下，可以满足任何调度的要求。Linux 对于实时任务的调度不做任何保证。虽然不能保证硬实时工作方式，但 Linux 的实时调度算法的性能还是很不错的。

3.1.4 在实时操作系统中运行多个应用程序

从前面的论述可知，由于 RTOS 要保证最高优先级的线程运行，且不能改变用户对线程优先级的设定，而独立应用程序的开发者又不能坐下来协商优先级，因此无法实现多个独立应用程序同时运行。VxWorks，Psos 等实时系统不能安装第三方应用程序，除了疑虑第三方软件的可靠性外，其调度器的实时特征也是决定性的。那是不是 RTOS 下就没有办法实现多个应用程序同时运行了呢？也不至于这么悲观，一些简单的约定可以帮助我们突破这些限制，只要独立应用程序的开发者同时遵守这些约定，完全可以使 RTOS 同时执行多个应用程序。

1. 规则 1：所有应用程序的轮转线程使用相同优先级。许多应用程序采用这样的策略：多个优先级相同的、一直就绪或者周期性就绪的线程轮流执行，处理程序的“日常事务”，这就是轮转线程。外加少量高优先级的线程用于响应突发事件或者 IO 操作。本规则要求所有的应用程序的轮转线程使用相同的优先级，djyos 操作系统中，约定优先级 200 为轮转优先级，并定义常量 cn_prio_cycle 表示。

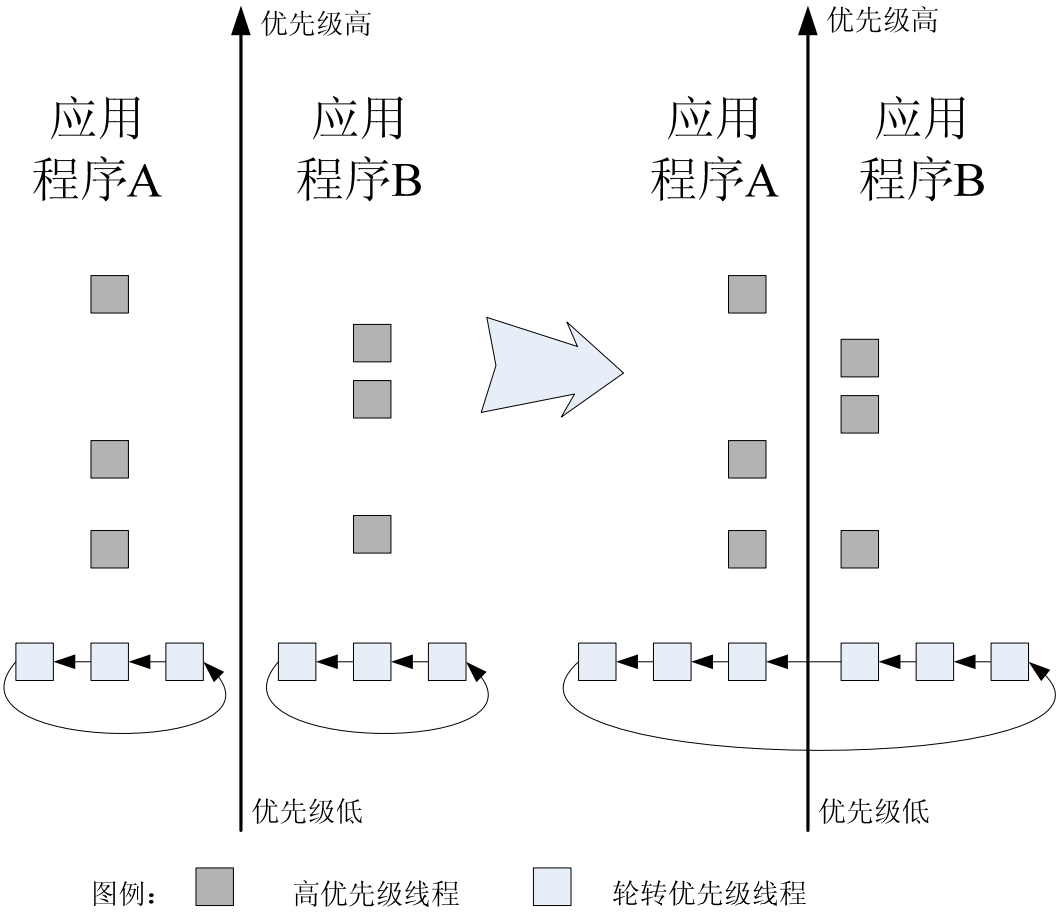


图 3-2 轮转线程使用相同优先级

a) 应用程序 A 和 B 是由独立的开发团队开发的没有关联的软件。但他们都遵守把轮转调度的任务的优先级定义为 cn_prio_cycle 建议，实时要求较高的线程则独立定义优先级，没有实时要求的线程使用轮转调度策略。

b) 当用户只启动其中一个程序时，运行方式如 图 3-2 左侧所示。当用户同时启动（或者先后启动）两个应用程序时，操作系统就把两个程序的轮转线程连接在同一个大的轮转调度表中。

c) 如果系统不支持多进程，轮转调度表中的的 6 个线程处于同一个进程中，如果在多进程系统中，这 6 个线程可以分属不同的进程。当然，系统会把属于同一个进程的线程放在轮转表的相邻位置，已减少进程切换的次数，降低开销。

2. 规则 2：所有线程干完活就找个理由休眠。

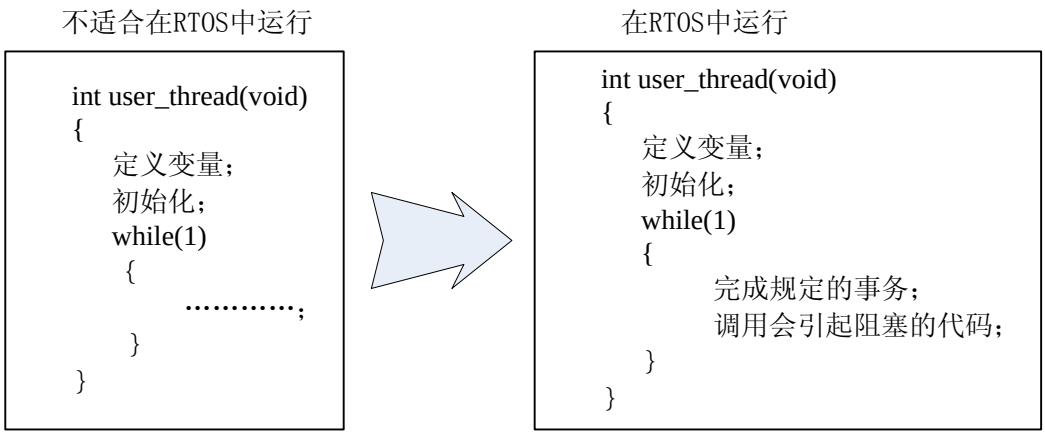


图 3-3 适合在 RTOS 中运行的代码

图 3-3 所示，即使是没有实时需求的轮转线程，也需要按右边的方法写，否则，即使线程在空转，操作系统必须等这个线程用完时间片才会把CPU分配给下一个轮转线程，白白地浪费了CPU时间。

3. 规则 3：禁止非实时任务使用紧急优先级。

在 第 4 章中讲到的djyos操作系统是一个典型的RTOS，的优先级共 255 级，共分 6 个优先级组，分组是一个建议性的分组，操作系统不干预用户对事件优先级的设定，在中断中弹出事件时，可以使用包括轮转优先级在内的任何优先级。但为了保证软件的可读性、可移植性等指标，用户最好遵守以下分组建议。

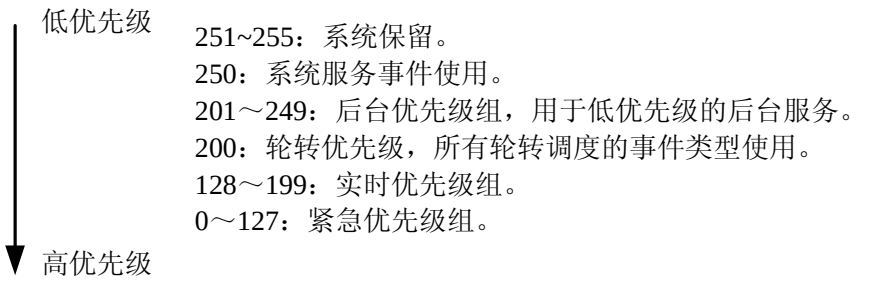


图 3-4 djyos 系统的优先级

后台优先级组用于要求很低的后台服务，比如后台打印服务和统计任务，djyos 把看门狗事件的优先级安排为 249；轮转优先级用于没有实时需求的线程；实时优先级用于各进程中有实时需求的线程，而紧急优先级用于实时要求很高的线程。编写软件时，不要随意使用紧急优先级，这样就能保证真正紧急的任务能够得到及时处理。

允许执行多个应用程序后，RTOS实际上蜕变成非实时系统，为什么呢？在 0 节中可知，计算机完成任务的时间包含Tp，在允许执行多个应用程序后，Tp可能是被别的应用程序的高优先级线程打断的时间。由于这个高优先级的线程是另一个毫无关联的程序建立的，无从得知它的执行时间，任务的完成时间就变得不可预测。根据实时系统的定义，无法预测最坏情况下执行时间的系统，显然不是实时系统。

既然这样，那在 RTOS 中执行多个应用程序还有意义吗？答案是肯定的，事实上我们的嵌入式系统经常用在非实时场合，数字电视的机顶盒、PDA、手机等都是典型的非实时系统，开发允许 RTOS 执行多应用程序技术，将使 RTOS 的应用范围更加广阔。而且，RTOS 中运行多个应用程序时其实时性能仍然高于普通非实时操作系统。在 RTOS 环境中，普通应用程序不使用高优先级，只启动一个高优先级的应用程序时，这个应用程序就是不折不扣的实时

应用程序。如今的嵌入式系统越来越复杂，功能上越来越面面俱到，大量传统上只有桌面系统上才有的文件系统、数据库管理、网络通信、复杂的显示界面等非实时应用软件，也出现在实时嵌入式系统中。现代的实时嵌入式系统，处于核心地位的实时控制任务，往往只占软件总量的很小一部分，而大量的增值功能却由非实时的软件提供。传统的开发方法中，由于要实现核心的实时控制，故只能选用实时操作系统，导致哪些非实时的软件也是不能由第三方独立开发，至多是利用第三方以源代码或者库函数方式提供的组件，继承到项目中一起编译，无法使用第三方提供的完整可执行程序。如果我们遵守规则 3，只有传统实时控制任务使用紧急优先级，就可以象在 wince 上一样，安装第三方独立开发的、不使用紧急优先级的软件，诸如数据库管理、数据分析显示等软件。

3.1.5 实时性、紧急性与可靠性

如果系统中有两个事件，事件 A 是一个随机事件，要求在事件发生后 1mS 内开始处理才能满足时限要求，CPU 需要开销 1mS，但是错过一两次事件并不会不可容忍的；事件 B 要求事件发生后 10mS 内开始处理即可以满足实现要求，CPU 需要开销 2mS，但是只要错过一次事件，将导致致命的后果。对于这样的两类事件，我们应该如何安排优先级呢？

显然，事件 B 比事件 A 重要，但事件 A 比事件 B 紧急，如果两个事件的实时性要求都得到满足，则必须使事件 A 的优先级高于事件 B 的优先级，这样的安排，则系统负荷不重的时候，可以很好地工作。但是，如果系统负荷很重，事件 A 发生的频率很高时，这样安排将导致事件 B 的实时性不能满足，**而事件 B 的实时性是产品可靠性的关键**，事件 A 的实时性与系统的可靠性发生了矛盾。理想的调度策略是，操作系统检测系统负荷，当系统负荷较轻时，事件 A 的优先级高于事件 B 的优先级，当过载或即将过载时，按事先设定调整优先级，使事件 B 的优先级高于事件 A 的优先级。这是一种弃车保帅的策略，POSIX 标准规定的“零星调度策略”就支持这种策略，有兴趣的读者可以参考 POSIX 的文档。

另一个典型的例子就是网络应用，许多应用于以太网通信的嵌入式产品提供 10M/100M 兼容接口，但是 CPU 的处理能力并不高，仅能处理数百 Kbps 甚至更低速率的数据流。当网络流量很低时，需要调高网络通信线程的优先级，防止丢包；当网络流量较高时，则要调低他的优先级，以适当丢包为代价使其他重要线程能够得到执行。

嵌入式条件下，事情还要更复杂些。可靠性和实时性要求高的系统，设计时应该严格控制 CPU 的负荷，所有的过载都应该在设计和测试阶段发现，解决的方式要么是减轻负荷，要么是增加 CPU 的处理能力，正常情况下运行时是不会过载的。所以，许多 RTOS 提供监视系统负荷的能力，虽然宣称支持 POSIX 标准，但并不实现“零星调度策略”。嵌入式系统需要面对各式各样的输入输出设备，许多输入设备是连接到中断上的，软件要对外部输入做出响应，外部输入有不可预测性，这些难于预测的外部输入可能引起系统过载。当然，设计者也应该保证，这些外部输入不应该使系统过载。然而，嵌入式系统尤其是工业控制系统运行条件恶劣，他们可能在强电磁干扰条件下工作，也可能在强烈机械振动、或者化学腐蚀环境下工作，这些都可能造成嵌入式系统硬件设备损坏。局部硬件设备损坏可能导致不可预测的大量突发输入，这些异常的、大量的信息还可能是通过中断输入的，面对这种“中断风暴”，操作系统的调度器根本无能为力！然而，局部硬件异常并不能成为系统可靠性降低的借口。

所以，设计高可靠性的实时控制系统，必须考虑因硬件故障或者异常输入而产生的非正常过载，应该保证，当硬件没有故障时，保证所有的任务的实时性，而在故障状态下，检测到并记录故障，关闭与故障硬件相关的线程（或者中断），或者调整优先级安排，保证其他软硬件模块正确运行。软件除了与故障部分硬件相关的功能外，其他功能应该保持完整。

3.1.6 案例：降低速度，提高实时性

乍一看，本节的标题很矛盾，降低了速度，怎么反而能提高实时性呢？这不忽悠人吗？且慢，我们看看实时性的定义，实时性讲的是最后时限，当然也包括在异常情况下的最后时限。如果计算机用 A 和 B 两种方法完成同一件任务，A 在绝大多数情况下 1mS 就可以完成，而最坏情况可能需要 100mS，平均 1.1mS 完成；而 B 在绝大多数情况下 10mS 完成，最坏情况下 20mS 完成，平均 12mS 完成，表面上，B 比 A 的执行速度要低得多，但是 B 的实时性却比 A 高得多。在 djyos 操作系统中，有不少地方是以降低速度为代价换取高实时性的，下面一 rtstrlen 函数为例说明一下。

ansi C 有一个标准库函数 strlen，用于检测一个字符串的长度，函数原型是

size_t strlen（const char *s）;

这个函数在经常被用于检测未知长度的字符串的实际长度，或者字符串长度是否非法，比如下列代码可以在遇到长度非法的字符串时返回错误：

len = strlen(name);

if(len > xxx) return error;

但这个函数本身有没有问题呢？要检测一个字符串长度，就得从 name 指针开始逐个比较字符，直到发现串结束符'\0'为止。如果由于指针错误，name 传进来一个无限长的字符串（当然，32 位机上，长度不会超过 4G），或者故障故障字符'\0'被意外修改，会是一个什么样的结果呢？要多长时间才能完成？不要忘了，name 是一个指针，C 语言里，指针可以为所欲为。不要心存侥幸，认为在内存中一定会出现字符'\0'，在嵌入式系统中，嵌入式 CPU 往往没有 MMU，不能提供存储器访问异常中断，更不要说虚拟内存技术了，访问没有配置存储器的空闲地址时，相当于读悬空的输入引脚，往往得到 0xFF。比如一个 32 位机，配置了 1M 存储器，地址范围是 0~0x100000，那么，在地址从 1M~4G 的范围内，读操作得到的可能都是 0xFF。如果因为指针错误 name=0x100000，那么这个字符串的长度至少等于 0xFFF00000，strlen 函数至少要做 4293918720 次比较才能停下来。

该函数在异常情况下的危险还在于，嵌入式系统中有不少设备是不能随便读的，象上述从 0x100000~0xffffffff 地址空间的疯狂的读，很可能会读这些不能读的设备的地址空间，导致误操作，有硬件设计经验的读者一定会有深刻的印象。

笔者实际测试了一下 strlen 函数的行为，CPU 是 44b0x，主频 64M，cache 全开，gcc 编译器，做了一个 4M 长的字符串，测试伪代码如下：

初始化 4M 字符串；

启动定时器；

调用 strlen；

读取定时器。

执行结果是 strlen 正确地报告了串长度，用时 315 毫秒，平均每个字符 75nS。如果串长度接近 4G，那执行时间岂不是要 300 多秒，要是用不带 cache 的 CPU，岂不要超过 10 分钟？速度再慢一些的 CPU 呢？而且，问题不在于具体要多长时间，而是 strlen 的执行时间的不确定性，实时系统拒绝不确定。本来是想用 strlen 函数防止字符串长度非法，以提高软件可靠性，就算是买个保险吧，谁知买来的是一个定时炸弹！

因此，djyos系统提供了一个rtstrlen函数（代码 3-1），用于替代strlen函数，该函数要求调用者提供一个合法上限，超过该上限即返回非法结果。一般来说，库函数中的函数都是经过高度优化甚至手工汇编优化的，其效率应该比rtstrlen高，再者，rtstrlen在每次比较字符是否等于'\0'的同时，还要比较长度是否超过over，其效率远比strlen函数低。但它的执行时间

上限是确定而且可控的，所以说它是符合实时系统要求的。所以说，这是典型的以低效率换取实时性的例子。

在 djyos 系统中，选择使用 strlen 和 rtstrlen 的原则是：

如果被测字符串有可能出现非法指针（比如由别的模块传入的），则必须使用 rtstrlen 函数。

如果被测字符串长度的上限已知且合法，只不过是需要知道其究竟有多长，则使用 strlen 函数。

代码 3-1 实时字符串函数

```
sint32_t rtstrlen(const char *s,sint32_t over)
{
    sint32_t len;
    if(over == 0)
        return strlen(s);
    else
    {
        for(len = 0; len < over; len++)
        {
            if(s[len] =='\0')
            {
                return len;
                break;
            }
        }
        return cn_limit_uint32;
    }
}
```

3.2 友好组——软件界的“智子疑邻”游戏

友好程序与可疑程序不仅是一个技术范畴的概念，更是一个工程范畴的概念，友好程序由友好、协作团队开发而得名；可疑程序由不协作的、非协作团队开发而得名，并不是因为它真的不怀好意。

本书初稿虽然在 2008 年下半年才成型，但是在 2004 年就开始写了，那时微软的 vista 上市还遥遥无期，本书早期使用的术语是更加准确的“可信组”和“可信程序”和“可疑程序”。但是 vista 中引入了“可信程序”的概念，而且它的“可信程序”术语的定义与 djyos 中的定义完全不一样，随着 vista 在 2007 年 1 月上市，这个术语被“抢注”了，为了避免不必要误会，本书改称“友好组”和“友好程序”和“可疑程序”。

3.2.1 友好组划分

所有的友好代码组成一个友好组，所有非友好代码都是可疑程序。“友好”有两层意思，一是这些代码是为一个共同的目标，密切协作地完成任务；二是这些代码不会主动地、恶意地互相干扰和破坏，也不会故意执行非法操作。

以设计软件的人为核心，是都江堰操作系统的核心思想之一。现代软件开发中，非常强调工程化开发的思想，在工程化开发中，任何软件策略都不能不考虑人的因素。划分友好程序和可疑程序是 djyos 操作系统设计中的一个根本性的策略，划分的依据是开发团队，而不是以技术特征，是都江堰操作系统当然的选择。把协同工作的团队开发的所有代码称作友好程序，划入一个友好组，而把所有不协同工作的团队以及个人开发的程序当成可疑程序，主要考虑的是人的因素。根据这个原则，操作系统自身的所有代码是由一个核心团队开发的，当然地成为一个友好代码组，操作系统开发团队无需与开发应用程序的团队和个人协作，它把所有应用程序都当作可疑程序；同样，一个应用程序是由一个人或者一个密切协作的团队开发的，该程序的所有代码形成一个友好组，而把所有其他应用程序以及操作系统都视为可疑程序。按照以上原则，我们可以这样直观地划分友好组：

1. 操作系统内核是一个友好组。
2. 多道程序设计中的每一道程序都构成一个友好组。

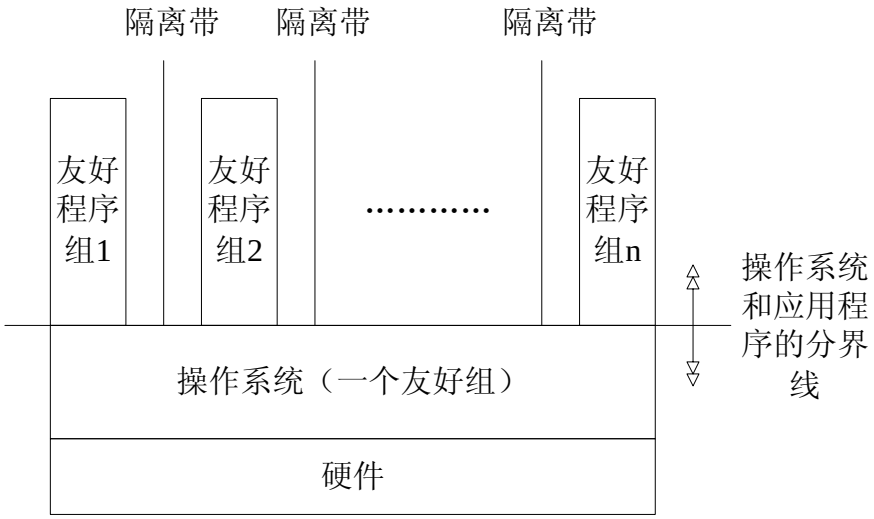


图 3-5 都江堰操作系统支持下的软件结构

既然友好组内程序是自己人开发的，当然是可信任的了，那么，友好程序是否一定比可疑程序可靠？我们说，不一定，比如，任何应用程序都把操作系统看做可疑程序，然而，因为操作系统是经过严格的开发测试，再经过众多用户和项目的应用，其 bugs 有充分的时间和机会暴露，进而加以解决。而应用程序则不同，许多应用程序的测试并不如操作系统充分，它的用户数量肯定比它所选用的操作系统少，通过用户使用暴露 bugs 的机会也就比操作系统少，所以，大多数较大规模的应用程序，缺陷的数量会比操作系统多。因此，应用程序遭作为可疑程序的操作系统干扰的可能性，很可能低于遭自己人开发的其他模块无意中破坏的可能性。这很有意思，不受信任的（操作系统开发）团队，其可靠性却高于自己人，真是家“贼”难防啊。

更有趣的是，并不是一个项目团队开发的代码就一定会被划分为一个友好组，大多数情况下，一个应用程序构成一个友好组，但并不尽然，参考 1.7 节，如果一个程序可以划为AB两部分，其中A部分的代码规模小，但可靠性要求很高，而B部分的程序规模很大，可靠性

要求却低得多。那我们就会花很多精力在A部分，确保A部分没有错误，而B部分的缺陷会远比A部分多而且严重。如果A部分和B部分作为一个友好组，那么B部分的缺陷可能会导致整个友好组崩溃，从而A部分也不能自保。这种情况下，就应该把A部分和B部分分别做一个友好组，利用操作系统提供提供的组间隔离和保护能力，确保A部分不因B部分bugs导致的有意或无意的破坏。

3.2.2 友好组保护

都江堰操作系统在友好组之间提供保护，以免软件被不明代码破坏，这包括恶意代码的故意破坏，也包括无恶意代码的无意破坏。在同一个友好组内，可以共享数据，使用全局变量等，而在不同的可信组之间，各自的数据和代码都是不可见的，要交换数据就必须通过操作系统中转，或者通过可以信赖的接口（例如泛设备接口（参见 第 9 章））。在命名空间方面，同一个友好组共享命名空间，而不同的友好组有互相独立的命名空间，也就是说，在同一个友好组内，你不能定义两个同名的全局变量，不能有同名函数，而在不同的友好组之间，就没有这个问题。操作系统本身是一个特殊的友好组，它需要为所有应用程序提供服务，这些服务是通过api函数提供的，因此，api函数的名字将占用应用程序的命名空间，也就是说，任何应用程序都没有完整的命名空间，因为它们不能定义与操作系统api函数同名的标识符。图 3-6 图示了都江堰操作系统命名空间的划分，

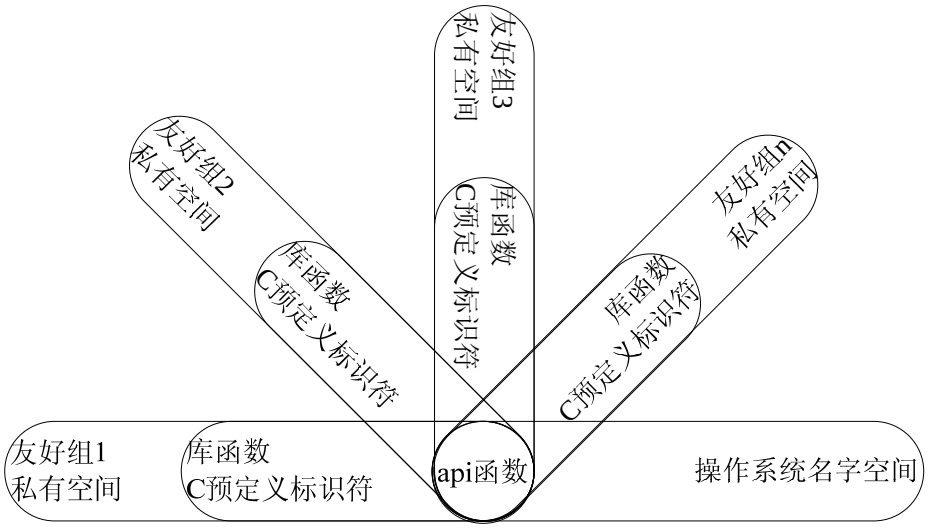
表格 3-1

	dlsp 模式		mp 模式	
	友好组之内	友好组之间	友好组之内	友好组之间
全局变量	可访问	不可访问	可访问	不可访问
函数	可互相调用	可用指针按回调函数互相调用	可互相调用	不可互相调用
内存空间	共享	共享	共享	独立，用指针也不能互相访问
代码	共享	独立	共享	独立
命名空间	共享	独立	共享	独立
连接（注）	目标机	目标机	后台机	后台机

注：1、si 模式不支持多可信组。

2、在目标机连接是指程序在开发后台只编译，不连接，由执行程序的目标机在加载前连接。

3、后台机连接是指开发后台提供经过连接的、具有绝对地址的代码，目标机只需要直接加载到目的地址就可以运行。



dlsp和mp模式的命名空间

si模式下，
操作系统和应用程序共用名字空间

图 3-6 都江堰操作系统命名空间

1. api 函数占据的名字空间是操作系统与所有应用程序的交集，任何应用程序都不可以定义与 api 函数同名的函数或其他标识符。
2. 各友好组内，库函数和 C 预定义标识符单独列出，是因为这些标识符是预定义的，应用程序员无法改变，但可以选择不使用，或覆盖掉。比如 memset 函数，如果你需要用这个函数，则必须使用 memset 这样的函数名，但你也可以选择不使用这个函数，这时候，你可以定义一个自己的 memset 函数（前提是你没有#include <string.h>文件）。
3. 在友好组之间，各友好组的私有命名空间是完全独立的。

都江堰操作系统把操作系统本身作为一个友好组，不允许任何应用程序作者开发的代码进入这个友好组，大多数情况下，应用程序的 bugs 密度会远大于操作系统，如果允许部分应用程序代码进入操作系统友好组的话，这些 bugs 发作时，可能会直接破坏操作系统而使之崩溃。软件开发过程中总会出现很多错误，要经过成百上千次的“运行－查错”循环才能完成，如果用户程序能轻易造成系统崩溃的话，那频繁的重新启动系统将极大地降低软件调试工作的效率，因此操作系统总是把用户程序看做可疑代码。笔者在写这本书的过程中，正好有一个项目需要做一个在 Linux 下的 PCI 卡驱动程序，由于对 Linux 不熟悉，所以花了大量时间熟悉 Linux 系统内核（我只是要写一个 driver，却要熟悉整个内核），在调试的时候，由于驱动程序运行在内核空间，驱动程序特别是中断里面的 bugs 造成反复的重启系统，而且这类错误往往无从着手查，因为系统崩溃了是不会给你输出任何信息的，只能重启，反复的重启差点没把我搞得精神分裂。这时候我就想，如果用户编写的 linux 设备驱动程序运行在用户空间，我根本不用花那么多时间和精力去熟悉 Linux 内核，也不用忍受那不停重启这种非人的折磨，这也使笔者深刻体会到，代码在哪个空间运行，不能由代码的功能属性来决定，而是应该由编写代码的团队来决定，由用户编写的代码应该运行在用户空间。把所有功能属性是设备 driver 的代码都放在内核空间运行，是一个面向过程的方法，而不是面向对象的思想，这点上，不得不说，是 linux 的一个败笔。——软件工程中，面向对象的思想无处不在，而不仅仅在于如何书写代码。当然，linux 最初设计在 X86 上运行，X86 的 IO 指

令是特权指令，只能在操作系统层操作，由于 X86 的先天特性，linux 的驱动程序运行在操作系统空间能提高执行效率。

综上所述，djyos 系统友好组的保护可分两个级别：

1. 在多进程环境下实现物理隔离，使不同友好组拥有独立的寻址空间，他们之间的代码、数据在物理上互不可见，也就无法互相破坏。实现物理隔离的手段，就是下一节讲述的虚拟机。
2. 在平板内存模式下实现逻辑隔离，在编写软件时，同一个友好组的代码独立编译，拥有独立的命名空间，友好组之间的函数、变量互不可见，使其不能显式地互相破坏，但在物理上，他们共享寻址空间，用绝对地址（比如通过指针）是可以访问其他友好组的数据和代码的。

3.3 友好组隔离的实现

3.3.1 虚拟机

我们看看“意大利面条”程序的困惑，一是大量“测试－分支”指令使大的程序模块缠绕不清，二是各模块使用共同的资源，使模块间相互干扰。因此，我们改进“面条”程序的方法，一是根据软件特性把大的软件分成一个个独立模块，各模块独立运行。二是使模块独立使用所需要的资源，不互相干扰。一个或多个相关模块就组成一个友好组。

有两个方法可以达到模块间不互相干扰，一是隔离，使模块相对之间不可见，一个模块既看不到别的模块存在，也不能破坏别的模块的代码和数据，也就达到不能相互破坏的目的。二是管理和监控，使用一组可靠的代码管理和监控各模块的行为，使之不能相互捣乱。

操作系统通过虚拟机实现了对用户程序的隔离和监控。虚拟机是一台看起来象是一台完整的计算机，它拥有执行程序所需的一切——自己的 CPU、寻址空间、存储器和寄存器，它的 CPU、存储器和寄存器分别是物理计算机的一个子集。都江堰操作系统中，主要有三种虚拟机：超级虚拟机、进程虚拟机和线程虚拟机，另外还有一个特殊的虚拟机“共享虚拟机”。超级虚拟机只有一个，就是执行操作系统内核的虚拟机，注意操作系统除了占据超级虚拟机以外，还可能创建并使用若干进程虚拟机。进程虚拟机由“部分 CPU 时间”+“独立寻址空间”+“物理计算机内存的子集”+“操作系统服务”四部分组成，而线程虚拟机则由“宿主进程虚拟机的部分 CPU 时间”+“宿主进程虚拟机的部分寻址空间”+“操作系统服务”三部分组成。“共享虚拟机”是为方便线程间共享数据而设的专用虚拟机，在此暂不讨论。在 si 和 dlsp 模式下，只支持线程虚拟机，线程虚拟机的宿主虚拟机就是物理计算机；在 mp 模式下，则三种虚拟机都支持。操作系统在超级虚拟机内执行，应用程序在进程虚拟机和线程虚拟机上运行，应用程序的每个进程占用一个进程虚拟机，在进程内，有可以创建多个线程虚拟机，每个线程虚拟机就是执行应用程序的一个或数个低昂对独立的执行单元。创建超级虚拟机需要硬件的支持，在没有特权级和特权指令的 CPU 上，超级虚拟机和进程虚拟机除寻址空间不同外，没有什么区别。操作系统依托超级虚拟机的强大能力，能够管理和监控进程和线程虚拟机，反之，应用程序则不能控制超级虚拟机。每个进程虚拟机都是一台独立的、被剥夺了部分 CPU 时间的计算机，他们之间是互不可见的。在 mp 模式下，友好组和进程虚拟机是一一对应的，借助线程虚拟机之间的隔离功能，实现友好组之间的隔离保护。

既然进程虚拟机不能直接访问超级虚拟机，那么，应用程序需要操作系统服务又是怎样实现的呢？都江堰操作系统工作在不同的模式下，有不同的方式实现。si 模式和 dlsp 模式中，

物理计算机并没有被划分为多进程，应用程序直接调用函数即可；在 mp 模式下，需要走严格的“法律程序”，首先，应用程序先把自己需要的服务写在便条上，然后把这个便条以专门的方式传送给操作系统，这种传送过程在计算机术语中成为“陷阱”或者“软中断”，最后，操作系统执行所请求的服务，并把执行结果返回给应用程序。

引入操作系统和虚拟机概念后，整台计算机的结构如 图 3-7 所示，它是一台通用的计算机结构图，在不同的cpu上会有细微差别。

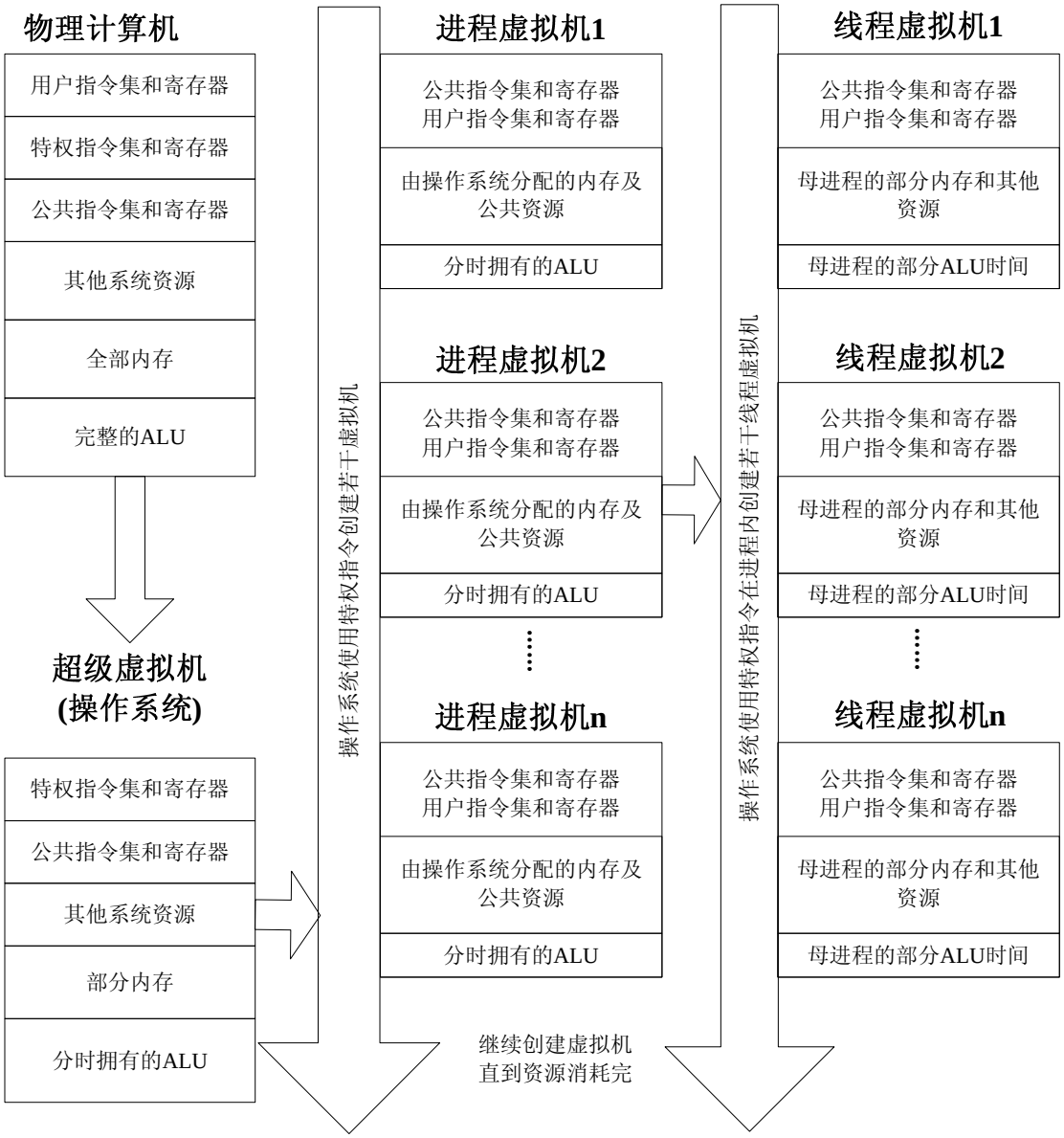


图 3-7　虚拟机

3.3.2 超级虚拟机

前面讲到，像警察一样，超级虚拟机肩负着管理和监控进程虚拟机和线程虚拟机的重任。为了完成这个重任，操作系统需要特权，这个特权就由操作系统所在的超级虚拟机提供。与进程和线程虚拟机相比，超级虚拟机拥有强大的特权，表现在：

1. 超级虚拟机可以创建和摧毁任意进程虚拟机和线程虚拟机。
2. 超级虚拟机拥有比进程虚拟机更加强大的 CPU，该 CPU 拥有特权指令，它随时可

以抢夺进程虚拟机的 CPU 使用权，也可以随时把 CPU 交给另外一个进程使用。

3. 超级虚拟机拥有更强大的寻址空间访问能力，超级虚拟机除了可以访问自己的寻址空间外，还可以访问任意进程虚拟机的寻址空间，反之则不行。当然，也不是像访问自己的寻址空间那样直接读写，而是要通过特别的“后门”才能访问进程虚拟机的寻址空间。

操作系统作掌握着程虚拟机和线程虚拟机的生死予夺大权，超级虚拟机的这种权力，完全仰赖 CPU 的特权模式，只有在提供特权模式的 CPU 中，才能创建成就虚拟机。依托超级虚拟机的强大能力，操作系统就能够为应用程序提供强大的管理、监控和服务：

1. 响应应用程序的需求，适时创建进程虚拟机以完成计算任务。
2. 根据调度算法，在恰当的时候剥夺一个进程虚拟机的 CPU 使用权，切换到另一个进程虚拟机，这就叫上下文切换。
3. 把物理资源分配给各进程虚拟机使用，协调各进程虚拟机协同使用公共资源，比如把所有申请使用串口的进程虚拟机排队，一个一个使用。
4. 接受进程的申请，在进程虚拟机内部创建线程虚拟机。

系统复位后，启动程序执行必要的初始化工作后，首先创建的就是超级虚拟机，然后由超级虚拟机接管 CPU，至此，启动程序的使命就完成了。

3.3.3 进程虚拟机

在 MMU 支持下，允许创建多个进程虚拟机的硬件环境称作“多虚拟机硬件平台”。拥有“多虚拟机硬件平台”且提供创建管理进程虚拟机服务的软件环境称作“**多进程系统**”。由于多进程系统需要切换地址映射表并提供进程间交换数据的功能，管理多个进程虚拟机也使系统变得更加复杂，在实时系统中可能成为性能瓶颈，所以是否使用“多进程系统”，取决于实际应用的需求。硬件不支持创建进程虚拟机或者虽然拥有“多虚拟机硬件平台”但是软件平台不支持创建进程虚拟机的环境，称作“**单进程系统**”。在单进程系统中，即使拥有 MMU，通常也只被用来扩大软件的寻址范围。

在多进程系统中，可以创建进程虚拟机，软件模块分别在一个个独立的线程或进程中运行，每个进程占用一个进程虚拟机，每个进程虚拟机都拥有自己的：

1. CPU，该 CPU 由物理 CPU 分时复用而成，与物理 CPU 极其相似，它与物理 CPU 的主要差异在于：
   a) 由于分时复用，它在宏观上要比物理 CPU“慢”一些。
   b) 被取消了特权指令，不能执行特权操作。
2. 存储器，该存储器有独立的私有寻址空间，该寻址空间除了可能被超级虚拟机访问外，对于其他进程虚拟机是不可见的。独立寻址空间并不意味这拥有与宿主物理计算机一样的寻址空间，而只能是它的一个子集。在都江堰操作系统支持下的虚拟机内存模式如 图 3-8 所示，超级虚拟机的寻址范围是 0~1G，同时超级虚拟机还可以通过特权指令，访问任意进程虚拟机的内存空间。共享虚拟机的地址范围是 1~2G，该虚拟机在用户模式运行，它的寻址空间对所有进程都是可读的，但只有超级虚拟机和共享虚拟机本身可以对它执行写操作，共享虚拟机是各进程虚拟机交换数据的重要途径，下文将讲到的独立泛设备driver也放在这里。每个进程虚拟机都有相同的寻址范围 2~4G，共 2G的空间。
3. 自己的输入输出系统，因硬件因素，进程虚拟机的输入输出系统的构成有两种情况，一种是由 x86 为代表，这类计算机的输入输出指令是特权指令，只有操作系统才能执行，进程虚拟机的输入输出系统由操作系统提供的系统调用组成；另一种是以

ARM 为代表的，这类 CPU 没有专门的输入输出指令，输入输出操作由普通指令完成，进程虚拟机与操作系统所在的超级虚拟机拥有完全相同的输入输出能力。

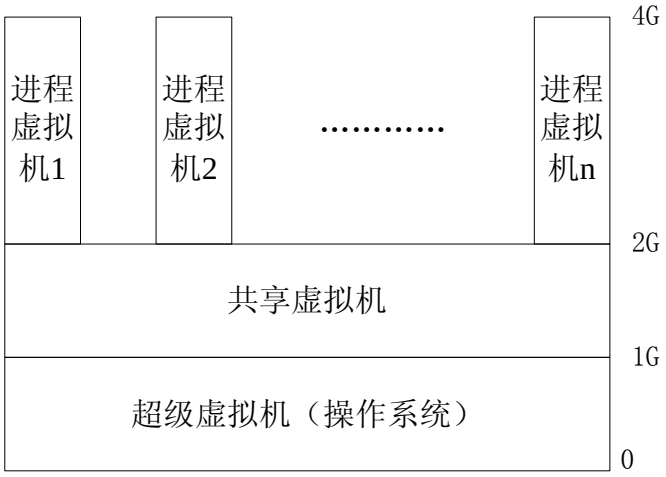


图 3-8 进程虚拟机内存空间

图 3-8 中，各种类型的虚拟机对存储器的访问权限规定如下：

内存空间	超级虚拟机	共享虚拟机	进程虚拟机 1	进程虚拟机 2	进程虚拟机 n
0~1G	读 写 执行	无	无	无	无
1~2G	读 写 执行	读 写 执行	可读	可读	可读
2~4G（1#）	读 写 执行	无	读 写 执行	无	无
2~4G（2#）	读 写 执行	无	无	读 写 执行	无
2~4G（n#）	读 写 执行	无	无	无	读 写 执行

回到第 3.2 节，操作系统提供友好组之间保护的方式，在mp模式下，就是用进程虚拟机来实现的，友好组与进程虚拟机一一对应，在友好组内的各模块之间，甚至线程虚拟机之间，代码和数据都是完全可见可访问的。而在友好组之间，通过进程虚拟机之间的篱笆墙，互相之间的代码和数据都是封闭的，完全看不见对方，更不用说修改了。无论是恶意的还是无意的，任意一个友好组都不可能干扰另一个友好组的运行。

再提一下“共享虚拟机”，它的作用协助进程间交换数据，如果没有它，进程间通过地址空间保护这个“篱笆墙”隔离后，交换数据就必须依靠超级虚拟机了。由于开发操作系统时不可能预测到以后的所有需要，它提供的手段是有限的，而且往往是低效的。由于进程无法访问超级虚拟机的地址空间，所以数据只能等待内核从源进程读出，再写入目标进程，过程不但很繁琐，而且还可能需要频繁的地址空间切换和同步（一般用锁）操作，开销很大。而拥有一个所有虚拟机都能读的地址空间，对数据交换无疑大有裨益。“共享虚拟机”作为一个虚拟机而不仅仅是一个地址空间，就赋予了它执行代码的能力，需用它交换数据的进程可以通过安装回调函数，使数据在“共享虚拟机”中做预处理，可以提供更加丰富的数据接口手段。该虚拟机不能访问进程虚拟机和超级虚拟机的内存空间，又确保了虚拟机之间的独立性。

读者注意到了，在 图 3-8 中，超级虚拟机的寻址空间反而比进程虚拟机小，这是合理的，一方面，对于执行操作系统内核来说，1G的寻址空间已经足够了；另一方面，在许多系统中，应用程序的规模可能远大于操作系统内核，特别是在数据库类型的应用中，更是如此；再者，大一点的寻址空间可以给应用程序程序员更自由的发挥空间，但操作系统自身就只好节俭一点了。

3.3.4 线程虚拟机

在多进程系统中，线程虚拟机是在进程虚拟机内创建的，该进程称作线程的宿主进程；单进程系统中，线程虚拟机直接在物理计算机上创建，如果把物理计算机看做系统中唯一进程的话，物理计算机就是线程的宿主进程。线程虚拟机使用宿主进程虚拟机的部分内存和CPU 时间，线程虚拟机与进程虚拟机的区别在于：

1. 寄生关系，线程虚拟机寄生在进程虚拟机内，如果进程虚拟机被销毁，该进程虚拟机的所有线程虚拟机自然消亡。
2. 存储器方面，线程虚拟机拥有从进程虚拟机内分配的“独立”的栈空间。进程虚拟机的所拥有的线程虚拟机共享进程虚拟机的寻址空间，线程虚拟机之间可以互相访问对方的存储器。也就是说，线程虚拟机的私有空间只是“君子协定”而已，要是碰上强盗，一点办法都没有。
3. 线程虚拟机使用宿主进程虚拟机的输入输出系统。

典型的多线程的进程的内存映像如 图 3-9 所示。

图 3-9 多线程系统中进程的内存映像

第4章 djyos—崭新的操作系统

djyos 操作系统是以事件为核心进行调度的，这种调度策略使程序员可以按人类认知事物的习惯而不是计算机的习惯来编程。

普通操作系统中，调度是以线程为核心的，事件被作为线程的数据，标榜为“事件触发”的软件模型，也是由线程在一旁候着，待特定事件发生时线程恢复运行并把它作为输入数据加以处理。

以事件为核心的调度，则像设备和内存一样，把线程虚拟机作为处理事件所需要的资源看待，当某事件需要处理时，分配或者创建一个线程虚拟机给该事件，并启动该线程虚拟机处理事件。

4.1 术语

这里介绍几个 djyos 系统引进的术语，

线程——即处理事件的线程，简称线程。在 djyos 中，线程是事件的资源，只有拥有了线程资源，事件才能被处理。

事件切换——也称线程切换、上下文切换，即把 CPU 从处理 A 事件的线程切换到处理 B 事件的线程，而 A 事件被挂起，若处理 B 事件的线程还不存在则先创建线程虚拟机再切换。在不引起歧义时，可简称为切换。

事件切入——也称线程切入，把处理某一个事件的线程的上下文从其栈中恢复到 CPU 上继续运行，线程还不存在则先创建线程虚拟机再切入。在不引起歧义时，可简称为切入。

事件切出——也称线程切出，把某事件的上下文从 CPU 中拷贝到处理该事件的线程虚拟机的栈中，然后事件被挂起，在不引起歧义时，可简称为切出。

4.2 djyos 系统的基本元素

事件、事件类型、线程是 djyos 系统的三个基本要素。事件表示发生了某件事情，需要处理，事件类型表明这个事件的基本属性，其中包含了处理事件的函数指针，线程是由操作系统建立来处理事件的。事件和事件类型是应用程序员定义的，而线程却对程序员却是不可见的。

4.2.1 事件

计算机处理的是现实世界中的具体任务，有因才有果，现实生活中的任务不会无缘无故地产生，人们做某一件事肯定是因为发生了某种事件。计算机中的事件与现实生活中的事件是一致的，CPU 不会无缘无故地执行某一段代码，就算是一段包含在一个 if 语句里的代码被执行，肯定是因为发生了使该条件成立的事件。人走到沙发前是一个事件，智能沙发上的计算机发现这个事件后才会处理这个事件，处理结果是执行调整坐垫到合适位置的操作；人转身面对电视机是一个事件，智能电视机里的计算机发现这个事件才会处理这个事件，处理结果是执行打开电视机的操作；人躺在床上并闭上眼睛，智能家居的计算机发现这个事件才会处理这个事件，处理的结果是执行关灯的操作。以上所述的事件，就是 djyos 操作系统中“事件”的原型。所有这些原型中，都有一个“发现”（或称“检测”）事件和执行一定操作以处理事件的过程，现实系统中，这两个过程可能非常复杂，甚至处于两个不同的学科，其软件实现模块可能会由两个不同专业方向的程序员编写。djyos 软件模型是：由一个软件模块专门用于监测人的行为，另外一些模块执行开关灯、开关电视机、调整沙发坐垫的操作。检测模块发现人靠近沙发的事件后，不是去调整沙发坐垫，而是把“事件”报告给操作系统了事。操作系统收到该事件后，把该事件记录在调度队列中，根据调度算法，决定要处理该事件时，就分配或创建用于处理该类型事件的线程虚拟机，并启动线程虚拟机，再由这个线程去执行调整沙发坐垫的操作。这样，就使“检测”和“执行”相互独立开来。进程、线程之类的东西只是操作系统内部的秘密，线程虚拟机作为一个资源，是创建新资源还是使用现有资源来处理事件，完全由操作系统自动完成，应用程序的程序员不知道也不需要关心这些。

djyos 的调度是基于事件的，这是 djyos 与传统操作系统最大的区别。传统的基于线程（进程）的调度模式下，用户只知道哪个线程（进程）正在占有 CPU，却不能知道该线程（进程）在干什么，操作系统调度器也只能针对线程（进程）分配 CPU，不能针对计算机所处理的具体事务分配 CPU。因此，传统操作系统下，程序员必须熟练掌握有关线程（进程）的知识，必须自己为程序需要处理的事务创建线程（进程），清楚在任何状态下哪些线程（进程）正在运行。而事件调度则不同，用户可能根本不知道线程（进程）的存在，以及谁正在运行，谁正在等待，实际上，程序员根本不需要关心这些。djyos 系统中，程序员只知道哪个事件正在被处理，哪些事件已经处理完成，哪些事件正在队列中等待处理。定义一个一个的事件类型并登记到系统中，为每类事件编写处理函数，便是编程的全部工作。当相同的事件，而具有不同的优先级时，在传统操作系统下，要么为每一种可能的优先级建立一个线程（进程）来实现，要么在执行中动态改变线程（进程）的优先级，不管哪种方式，程序员都需要花费大量的时间和精力，以确保事件按正确的优先级得到处理。而 djyos 不同，它先天就是以事件优先级作为调度的基础，只要在产生事件时，直接在事件中做优先级标志就可以了。例如一个串口通信程序，中断函数负责低层接收，当接收到完整数据包后，就发给上层应用程序处理，上层应用程序处理接收到的所有数据包，而数据包中有一个命令字域，不同的命令要求的优先级不同。在传统操作系统下，要么创建一个 comm_app 线程，在中断函数中把数据传送给该线程的同时根据命令字改变 comm_app 线程的优先级，这种在中断函数中改变线程优先级的做法，在许多操作系统中是不允许的；另一种方法是，为命令字对应的每一种可能的优先级，均创建一个相同的线程，这些线程除优先级不同外，其它部分完全相同，当中断函数接收到完整数据包时，就根据数据包中的命令字，发消息给相应优先级的线程。这种方法虽然可行，但是会消耗很多 CPU 资源，且难于在编码阶段枚举所有可能的优先级，一旦命令字发生变化，很可能就需要升级软件。而 djyos 系统不一样，程序员只需要定义一个事件类型，并为之编写事件处理函数 proc_uart()，当中断函数接收到完整数据包时，弹出

事件时直接以命令字对应的优先级作为参数就可以，djyos 的调度系统自动会根据事件优先级域进行调度。

另外，嵌入式系统多是反应性系统，反应性系统的主要任务是对外界的事件做出正确且及时的反应，从这点看，程序员根本就不用知道进程和线程这些东西，为处理外界事件而建立线程（进程）实际上是不得已而为之，传统操作系统下，你必须做这些工作，你的系统才能正确地为你做些事情。基于事件的调度非常适合这种反应性系统，被调度的目标就是反映外部刺激的事件，而不是处理这些外部刺激的线程，符合人们的习惯性思维。即使是非反应性系统，或者是非实时系统，基于事件的调度仍然有其优势，“有事就做，没事就坐”是人们最为习惯的思维方式，以事件为调度单位显然符合这种思维方式，而与人们习惯性思维相同的调度方式，又是避免人为错误，减少软件 bug 的有效方法。

现代计算机已经进入“ubiquitous/Pervasive Computing”时代，即普适计算。在普适计算时代，触手可及的计算产品里面也包含着触手可及的计算机程序，这些程序由大量的嵌入式程序员编写，是的，十年前的硬件工程师可以不懂软件，软件工程师可以不懂硬件，而今天的嵌入式技术，除了在一些很特殊的方向如射频设计，已经没有纯粹意义上的软件工程师和硬件工程师了。让这些队伍迅速壮大的、软硬兼施的“普适计算工程师”们去掌握晦涩难懂的进程和线程技术并灵活应用，恐怕要花费不少的人才培养成本，而使用传统的操作系统开发嵌入式产品，不理解这些复杂的概念根本就寸步难行，而人才的匮乏又将限制嵌入式产业的发展。djyos 操作系统不要求程序员操纵线程和进程，程序员只需把需要计算机处理的任务划分为一个个事件类型，并为各种不同类型的事件编写独立的事件处理函数，并且把它登记到系统中就可以了。当事件发生时，发现（检测到）该事件的程序员只要告诉操作系统“某类型事件发生了”，操作系统自动地创建或唤醒合适的线程去处理事件，而无须程序员亲自动手。当然，“普适计算工程师”即使是在 djyos 系统下编程，深入理解线程和进程技术，对开发工作也是很有帮助的。

djyos 下程序运行的过程，就是新事件不断发生，操作系统不断组织、创建、分配线程、进程以及其他资源去处理事件的过程。每弹出一条事件，djyos 操作系统就为它分配一个事件控制块，事件处理完毕后收回事件控制块。未处理完毕的事件就会堆积在队列中，操作系统对队列的容量有一定的限制，当队列中事件数量达到限制数时，操作系统将拒绝接受新事件。port_kernel.h 文件中的 cn_events_limit 常量用于确定队列容量，默认值=100，程序员可以修改它。

4.2.2 事件类型

程序运行期间会发生各种各样的事件，不同种类的事件由各自的线程处理，需要用事件类型去加以区分。djyos 操作系统为每一类事件分配一个唯一的事件类型 ID 号，并为每个事件类型分配一个事件类型控制块。事件类型的容量由 port_kernel.h 文件中的常量 cn_event_types 确定，它是用户可配置的选项，但不能在程序运行过程中动态改变，最多允许 32768 个事件类型，默认值是 256。系统会静态分配 cn_event_types 个事件类型控制块，用户在配置该参数时应该适量，尤其是内存紧张的系统。

事件类型必须经过登记才能使用，登记时要为该事件类型指定一个事件处理函数，该函数将成为线程虚拟机的入口函数，还要设定该类型的默认优先级，并且告诉操作系统该函数运行需要多少栈空间。在某些情况下，操作系统会为部分类型事件至少保留一个线程（参见 4.2.4 节），当有该类型的事件发生时，操作系统可能会自动创建线程处理该事件，也可能会指挥现有线程去处理该事件。

4.2.2.1mark 标记

若事件类型被打上 mark 标记，则该类型事件表示的是物理世界的一种状态，若此类型的事件重复发生，它也只代表系统处于某种状态，不是每次发生的事件都需要单独处理。因此，mark 型事件无论重复发生多少次，队列中都只会保留一条事件。而没有打上 mark 标记的事件类型则相反，该类型的每条事件都需要单独处理，非 mark 型的事件反复发生而又来不及处理时，事件队列中将积压多条同一类型的事件。这是一种很重要的事件，因为现实世界中有太多的 mark 型事件，试举数例如下：

1. 快件投递中，当客户有快件需要投递，就会给快递公司电话，同一个地址，无论重复多少次电话，只需派收件员上门一次把积累的快件全部取走就可以了。
2. 在 LCD 面板显示软件中，在内存中设计了一个显存镜像，应用程序修改显示内容时，修改的是镜像显存，然后发出“显示刷新”类型事件，处理该事件的过程就是把镜像显存中的图像搬到物理显示器上。这是一个典型的 mark 型事件，无论应用程序修改了多少次镜像显存，都表示“显存被修改”这一物理状态，处理一次“显示刷新”事件将把历史上积累的事件完全清理。
3. 串口通信软件中，缓冲区接收到数据，会发出“缓冲区有数据需处理”类型的事件，无论该类型事件重复发出多少次，都表示这样一个状态，事件处理时只要把缓冲区的所有数据取走，就可以了。

非 mark 型事件的例子也很多，例如：

1. 在百货商店，每进来一个顾客算发生了一个“顾客来了”类型的事件，由于每个顾客都是独特的，所以必须单独服务。
2. 同样是 LCD 面板显示软件中，当用户需要绘制时，就会发出“屏幕绘制”事件，因为每次绘制的内容都可能不同，所以每条事件都必须单独处理。
3. 串口通信软件中，应用程序需要发送数据，就把数据准备好，弹出“发送数据”类型的事件，并把数据缓冲区作为事件参数，由于每次事件的数据缓冲区都是独立的，不能把多条事件统一处理，而是每条事件都要单独处理。

4.2.3 线程

djyos以事件为调度单元，理论上，操作系统可以用任何方法处理事件，只要能够调用事件处理入口函数就可以了，当前版本的djyos操作系统使用的方案是，创建一个线程执行事件处理入口函数来实现事件处理。注意，线程是操作系统自行创建的、用于执行用户提供的事件处理入口函数的手段，对程序设计者是不可见的，这就隐含了一个事实：操作系统还可以选择其他方案代替线程方案，遗憾的是，笔者至今也没有想到可以替代线程的可行方案。关于线程虚拟机的知识，可参考第 3.3.4 节。要处理事件，操作系统就要为该事件创建线程，用户在登记事件类型时必须指定线程的入口函数，以及告诉操作系统执行该入口函数需要多少栈空间。测算栈空间时只须计算线程入口函数本身的需求，如果用户调用了操作系统的服务，无需计算该服务需要的栈，测算栈需求的方法参见第 2.8.2 节。在djyos系统中，线程虚拟机是处理事件的执行者，也作为事件的资源而存在——完成该事件需要许多资源，线程虚拟机是诸多资源之一。

4.2.4 事件、事件类型与线程

套用面向对象的方法，事件类型相当于 C++的类，登记事件类型相当于声明一个类数据类型；事件相当于对象，弹出事件相当于定义对象，同时做一些构造函数的工作；事件处理完成相当于撤销一个对象，同时做一些类似析构函数的工作；在初始化阶段弹出的事件相当于全局对象，在事件运行中弹出的事件相当于局部对象。事件和类最大的不同点是，事件控制块有一个成员——虚拟机，要到事件切入才初始化（mark 型可能在前一个事件处理完成后立即初始化）。

djyos 中应用程序的运行过程，就是不断地弹出新事件和处理事件的过程。每个事件都必须属于已经登记的事件类型，相同类型的事件使用相同的线程进行处理。

djyos 操作系统中，线程虚拟机作为事件的资源而存在，而该事件就是线程虚拟机的拥有者，因此，任何线程，都不能无缘无故地出生、存在和死亡，它必定与某一类型的某一条事件联系在一起。线程随事件的需要而生，随事件完成而消亡。线程无需登记，也无需有用户建立和启动，它的创建、启动和删除都是由操作系统自动完成的。这与传统操作系统不一样，传统操作系统可以由用户任意创建线程，创建一个毫无意义的线程是允许的。djyos 系统的调度依据是一个就绪事件队列和若干个同步事件队列，而不是线程队列（有的操作系统也称其为任务队列），djyos 中根本就没有线程队列。djyos 的调度针对事件而不是线程，创建一个线程虚拟机是因为它的拥有者需要处理，线程被切入是因为该线程的拥有者需要被切入，线程被切离是因为它的拥有者被挂起。把线程作为事件的资源的积极意义在于，当某类型的事件连续发生，操作系统将调集更多的资源，为其创建多个线程来处理该事件，如果在多处理机（多核）系统中，把这些线程分配给不同的处理器，处理器本身也就成为一种资源，将极大地方便多处理器系统的管理。而传统的编程方法中，程序员创建若干线程待机，每个线程对应一种或数种事件，待相应的事件发生后，唤醒线程予以处理，这种方式在管理多处理器方面要复杂得多。因此，djyos 在多处理器系统中，有先天的优越性。

线程的属性必须与事件类型对应，相同类型的事件使用相同的线程处理，不同类型的事件使用不同的线程处理。用户登记一个事件类型时，必须把操作系统创建用于处理该类型事件的线程所需要的两个关键参数：事件处理入口函数和该函数需要的栈空间。当某类型的事件发生后，操作系统就会在适当的时候创建线程（或分配存在的线程）执行该类型对应的事件处理函数。事件处理完成之后，操作系统会自动回收该线程所占用的资源，必要时还会删除线程（如何处置线程，参见第 4.3.11.3 节）。

每一条事件对应一个线程，如果有多条同一类型的事件需要处理，操作系统会创建多个相同的线程同时处理多个相同的事件。这可以更合理地使用计算机资源。虽然在单处理器的情况下，建立多个线程并不会比单个线程长期霸占处理器更充分利用处理器，但可以产生多个相同类型事件并行处理的效果，例如同时绘制多个窗口，特别是，在多处理机（或多核）系统中，可以把频繁发生的同一类型事件分配到不同的处理器上。而传统操作系统下，线程是由程序员创建的，如果程序员只为某项工作创建了一个线程，则该工作再繁忙也只有一个线程为它工作，该线程处理的多项任务只能串行执行。但是这样反复创建线程可能导致资源枯竭，比如处理某事件时需要使用串口，而串口又被其他线程占用，在串口被占用期间发生该事件，操作系统就会再次为其创建线程，该线程开始执行后会因串口资源繁忙而进入阻塞状态，如果事件反复发生，操作系统就会反复为其创建线程，直到消耗完所有内存，造成内存枯竭。为了防止发生资源枯竭事故，在事件类型控制块中提供了 vpus_limit 成员，表示该类型事件可以同时建立线程的个数，vpus_limit 的默认值是 10。

4.3 事件调度

4.3.1 调度的实质——分配 CPU 时间

我们在做系统设计的时候，为各功能模块分配资源是一个很重要的工作，说到资源，人们自然地想到内存、IO 口、通信端口、文件句柄等。往往会忽略一个最重要的资源——CPU 时间，如果软件模块得不到 CPU 时间，一切都是空的。操作系统的调度器就是用于分配 CPU 时间的，关于 CPU 时间，我们需要重视的问题包括：

1. CPU 系统处理速度，系统速度越快，等于处理单个事件所需要的时间越少，相当于有更多的 CPU 时间用于处理用户的运算任务，所以，配置适量的 CPU 系统速度，实际上就是为系统配置适量的 CPU 时间资源。因此，与用 bytes 表示系统的内存资源量一样，MIPS 是用来表示系统的时间资源量的。CPU 系统速度不能仅从主频分析，而要用标准测试向量来衡量，要根据具体应用的主要运算任务来选择标准测试向量。如果具体应用主要是整数运算，选择 dhrystone 是一个比较理想的测试向量，如果主要运算是浮点运算，则应该选择 whrystone 基准，等等。
2. 当多个模块争用 CPU 时，如何分配 CPU 时间，这实际上涉及的是操作系统内核调度算法的问题，许多人没有明确地意识到，这正是选择和配置操作系统的关键因素。

事件要得到处理，就必须得到 CPU 时间，如果只有一个事件就绪，它将毫无争议地得到 CPU 时间，如果有多个事件同时处于就绪态，表明这些事件在争用 CPU 时间，谁能得到 CPU 时间，调度器将做出裁决。传统的调度器是以线程为调度目标的，调度器裁决由哪个线程使用 CPU 的算法不外乎以下几种：

1. 轮转调度，即所有就绪态的线程一个一个地轮流使用 CPU，每个线程一次可以获得的时间应该是可配置的，一般是时钟嘀嗒的整数倍。
2. 非抢占优先级调度，为每个线程分配一个优先级，这些优先级可以相同，也可以不同。当多个线程就绪时，优先执行就绪线程中优先级高的线程，但高优先级的线程不能打断正在执行的低优先级线程，只能等低优先级线程时间到或者主动让出 CPU。同优先级的线程，则按进入就绪状态的时间顺序等待获得 CPU 时间。
3. 抢占式优先级调度，这种调度方式需要为每个线程分配一个优先级，当多个线程就绪时，最高优先级的线程将立即执行，它会强行剥夺正在执行的低优先级线程的 CPU 使用权，不管人家是否愿意。是否允许多个线程拥有相同的优先级，跟操作系统的实现有关，比如 UCOSII 是不允许的，而 VxWorks 在不使能轮转调度的情况下，也是允许多线程（任务）有相同优先级的。
4. 混合调度，这是一种优先级抢占和轮转调度相结合的一种调度方式，每个线程有一个优先级，这些优先级可以相同，也可以不同。当多个线程就绪时，最高优先级的线程将立即执行，它会强行剥夺正在执行的低优先级线程的 CPU 使用权，不管人家是否愿意。如果最高优先级有多个相同优先级的就绪线程，则按轮转调度的方式共享 CPU。

除轮转调度外，操作系统的其他调度方式都严重依赖线程的优先级，优先级是如何确定的呢？不外乎两种方法：

1. 固定优先级，即在线程诞生时，就拥有一个终生不变、由应用程序的程序员决定的优先级（防止优先级反转而需要临时调整优先级是一个例外）。一般实时操作系统会采用这种方式，详见第 3.1 节。
2. 动态优先级，线程诞生时，一般会拥有一个由程序员或者操作系统分配的优先级，

但是在运行过程中，调度程序可能会为了协调各线程的执行时间而动态地调整，不同的操作系统会有其独具特色的调整算法。一般支持多道程序设计的、采用占先式调度的非实时操作系统会采用这种调度算法，详见第 3.1.2 节。

把 CPU 时间当作资源进行调度的概念，更进一步的意义在于，把 djyos 系统移植到多核（多 CPU）体系中时，能够更加容易协调各 CPU。

djyos 系统中，调度是以事件为依据，线程是事件的资源，线程本身是没有优先级的，但上述理论依然适用，只要把其中的“线程”两字改为“事件”就可以了。

在实时系统中，事件的优先级是能否实现实时指标的关键，现实中，大多数事件会继承事件类型的默认优先级，因此确定每一类事件的默认优先级是系统设计的重中之重。

4.3.2 不允许直接控制线程和进程

djyos 不允许程序员直接控制线程和进程，为什么呢？我们把程序员比做汽车司机，汽车行进需要发动机推动，需要经过进气、给油、点火、排气这些过程，而驾驶员需要做的，不过是转动方向盘告诉汽车朝哪里走，踩油门告诉汽车开始走，熄火告诉汽车目的地已经到达，司机从来不需要亲自给气缸注油，也无需拿火柴去点火，更重要的是，汽车设计者也没有设计让司机拿着油壶给汽缸注油、用火柴给汽缸点火的机械装置。因此，发动机对司机来说是透明的，司机可以不知道发动机的存在，但必须知道目的地、油门和刹车踏板，以及方向盘；同样，在 djyos 系统中，线程和进程就像汽车的发动机，对于程序员是透明的，djyos 中虽然存在线程和进程，但从没有提供让程序员亲自操作线程和进程的机制。程序员可以不知道线程（进程）的存在，但必须告诉计算机你要干什么（弹出事件）和如何干（编写线程入口函数），什么时候开始干（启动汽车），什么状况要暂停（象塞车、红灯），什么状况表示工作已经完成了（熄火）。所以，程序员在 djyos 操作系统下编写程序时，只要登记事件类型告诉计算机有多少种类的事情可能要做，弹出事件告诉计算机开始做具体的某一个事件，使用同步机制在必要的时候暂停处理事件，事件处理函数返回或者调用 y_event_done 函数告诉操作系统事件已经处理完成。创建处理事件的线程，启动线程，最后杀死线程这些操作，是由 djyos 操作系统自动完成的。按照面向对象的概念，我们称创建线程和进程是操作系统内部的实现秘密，应该被封装在操作系统内部，无需应用程序的程序员了解，更禁止应用程序干涉的。因此，djyos 操作系统与传统操作系统最大的不同在于，它完全不提供线程控制函数。djyos 提供给用户使用的、跟调度有关的所有 api 函数，都是针对事件的，用户控制事件，操作系统操控线程或者进程以实现用户对事件的控制。下列操作是 djyos 系统不提供或禁止的：

1. 禁止手动创建线程，djyos 不提供创建线程的 api 调用。从另一个角度，工程化的软件开发要求保持模块的独立性，不允许一个模块直接控制另一个模块，如果线程 A 创建了线程 B，则 djyos 认为，线程 B 的诞生过程被线程 A 直接控制，线程 A 所属的模块与线程 B 所属的模块就不互相独立了。在用户登记事件类型和切入某事件时，由操作系统决定使用现有的线程还是创建新的线程来处理该事件。
2. 禁止自杀和他杀，djyos 不提供杀死线程（进程）的操作，因为这是操作系统的专利。那么，线程就长生不死了吗？也不是，djyos 是事件驱动的，如果一个事件已经完成，应用程序可以使用 y_event_done 系统调用来告诉操作系统，操作系统会决定是否要杀死线程。
3. djyos 也不提供直接使线程休眠或者进入等待状态的功能，使别的线程进入等待或者休眠状态肯定是被禁止的，为什么直接使自己休眠也被禁止呢？一是线程操作是操作系统的专利，应用程序既然不知道线程存在，又怎么使其休眠和唤醒？；二是

djyos 拒绝无缘无故的操作，这样可以防止程序员随意控制程序的执行流程，而留下 bug 隐患。线程放弃 CPU（可能是被迫的）或者占有 CPU 的原因不外乎以下几种，每一种原因 djyos 都提供了完善的支持，无需程序员手动创建、启动、停止或删除线程。

a) 有更高优先级的事件就绪，操作系统立即执行事件切换，正在处理的事件则在就绪队列中继续等待。

b) 如果允许轮转调度，则如果某线程的时间片用完，且就绪队列中有同优先级的线程在等待，就会发生事件切换。

c) 线程是为解决某一条事件由操作系统自动创建，如果该事件已经处理完毕，应用程序应该调用y_event_done（或者事件处理函数自然返回，但不推荐），操作系统将终止处理事件，而相应的线程虚拟机则可能被删除，也可能保留，参见第 4.3.11.2 节。

d) 事件需要延时指定时间，或者等待某特定时间才能继续处理，这在周期性执行的任务中最为常见。djyos为此提供了闹钟同步功能，参见第 5.1 节。

e) 等待某种条件达成，djyos 提供了事件同步功能、事件类型弹出同步功能、事件类型完成同步功能，等等。某事件使用同步功能后，如果条件还没有达成，操作系统会自动把同步事件置为阻塞状态，一旦同步条件达成，操作系统又会立即把同步事件置为就绪状态，如果该事件的优先级足够高，就会立即运行。

f) 处理事件所需要的资源被占用，djyos提供了内存同步和锁同步功能（第 5 章），锁同步有分为信号量同步和互斥量同步。需要注意的是，内存同步限于用块相联分配法（参见第 7.2 节）从系统堆分配内存，用固定块分配法（参见第 7.3 节）从特定内存池中分配的内存由信号量同步保护。

4. djyos不提供唤醒休眠或者等待状态的线程的api调用。首先，控制线程是操作系统的专利，应用程序并不知道线程是何物，既然不知道线程存在，又怎么唤醒？其次，djyos的事件没有休眠态，只有阻塞态和就绪态（参见第 4.3.7 节）；再次，阻塞状态的线程自身不拥有CPU，自然无法亲自调用这些api把自己唤醒，由其他线程调用这些api唤醒休眠中的线程，相当与一个线程控制另一个线程，违背模块独立性原则。

djyos 公开源代码，要增加象 process_idle 和 process_resume 之类函数是很容易的，作者告诫这些“黑客”们，千万不要这样做，djyos 内核从一开始设计就没有考虑这些功能，所有线程都自认为不会有任何人会干涉自己的内部事务，他们可能会被高优先级的事件剥夺 CPU，也可能会被外部中断打断，但绝不会被任何其他线程包括内核修改自己的内部数据结构和线程状态，所以不能保证增加这些功能不会带来其他副作用。

4.3.3 数据结构说明

事件调度需要的核心数据结构有 3 个，分别是：

线程虚拟机结构：`struct thread_vm`。

事件结构：struct event_script。

事件类型结构：struct event_type。

4.3.3.1事件类型控制块数据结构定义

代码 4-1 定义的struct event_type是事件类型控制块的数据结构，用于形成一张事件类型控制块表。在djyos.c文件中：

struct event_type tg_evtt_table[cn_evtts_limit];

用静态数组定义了事件类型控制块表所需要的内存池，因为事件类型控制块的初始化早于内存管理初始化，故该内存池不使用固定块内存分配策略分配，而是直接在内存块中用一个标志位表示该块内存是否空闲。port_kernel.h 文件中的 cn_evtts_limit 常量确定了表的大小，每个事件类型有一个类型号，类型号是该事件类型的唯一标识，就是相应的事件控制块在该数组中的偏移量。

代码 4-1 事件类型定义

```
struct evtt_property                //事件属性定义表
{
    uint16_t    mark:1;
    uint16_t    overlay:1;
    uint16_t    registered:1;
    uint16_t    in_use:1;
};

struct event_type
{
    struct evtt_property    property;       //事件类型属性
    struct  thread_vm   *my_free_vm;    //本事件类型拥有的空闲虚拟机
    char *evtt_name;    //事件类型允许没有名字，但只要有名字，就不允许同名
                        //如果一个类型不希望别的模块弹出本类型事件，可以不用名字。
                        //如模块间需要交叉弹出事件，用名字访问。
    ufast_t     default_prio;       //事件类型默认优先级.不同于事件优先级,0~249.
    uint16_t    pop_sum,done_sum;   //本类型已弹出事件总数，已完成事件总数
    uint16_t    repeats;    //未开始处理的该类型事件发生的次数
    uint16_t    vpus_limit; //本类型事件允许同时建立的线程虚拟机个数
    uint16_t    vpus;       //本类型事件已经拥有的线程虚拟机个数
    void (*thread_routine)(struct event_script *my_event);      //事件处理入口函数
    uint32_t stack_size;                    // thread _routine 所需的栈大小
    struct event_script *mark_unclear;      //未 clear 的 mark 型事件
    //这两队列都是以剩余次数排队的双向循环链表
    struct  event_script *done_sync,*pop_sync;      //弹出同步和完成同步队列头指针
};
```

事件类型属性 struct evtt_property 中各位的含义：

1. mark，overlay：mark型事件专用属性，见第 4.3.13 节。
2. registered：0 表示该事件类型控制块空闲，置 1 表示该事件类型控制块已经被登记。
3. in_use：1 表示该事件类型正在被使用，至少有一条该类型的事件在事件队列中未处理完毕。

下面说明事件类型控制块各成员含义，代码注释已经足够明确的成员就不再占用篇幅了：

1. evtt_name：事件类型名，为更好地提供组件化开发支持而设，参见 10.4.3。
2. default_prio：事件类型的默认优先级，如果弹出该类型事件时没有指定新事件优先级，新事件将继承事件类型的优先级。在实时系统中，事件的优先级是非常关键的参数，需要个别仔细设计。
3. repeats，该类型事件被重复弹出的次数，如果是 mark 型事件，则记录本类型事件上次执行 y_clear_mark 至今弹出的次数，如果不是 mark 型事件，则记录在事件队列中所有本类型事件（即未处理完成的事件）的总和。
4. vpus_limit：本类型事件可以创建的线程虚拟机总数。如果某类型的事件频繁发生，操作系统将可能创建多个线程来处理这些事件，djyos 用 vpus_limit 限制了同一种事件可以拥有的线程总数。
5. stack_size：处理本类型事件的线程虚拟机的入口函数thread_routine所需要的栈尺寸（栈尺寸计算方法参见第 2.8.2 节）。计算栈空间时需要考虑：用户自己写的代码和C库函数的需求，该函数还可能会调用操作系统提供系统服务函数，这些系统服务函数所需要的栈无需用户考虑。
6. mark_unclear：未clear的mark型事件，参见第 4.3.13 节。
7. pop_sync，done_sync：弹出同步和完成同步队列的头指针，参见第 5.2 节。

4.3.3.2登记事件类型

djyos 的事件类型采用先登记后使用的原则，每当应用程序弹出事件，必须指定事件类型 id，该类型应该是已经登记过的。在操作系统初始化函数__y_init_sys 中，初始化了事件类型和事件数据结构，之后，就可以调用 y_evtt_regist 函数登记事件类型了。在__y_init_sys 函数中，登记了系统的第一个默认优先级为 250 的事件类型：系统服务事件类型，把事件类型控制块表中其它单元初始化为空闲。

代码 4-2 登记事件类型

```
uint16_t y_evtt_regist(bool_t mark,bool_t overlay,
                       ufast_t default_prio,
                       uint16_t vpus_limit,
                       void (*thread_routine)(struct event_script *),
                       uint32_t stack_size,
                       char *evtt_name)
{
    uint16_t i,evtt_id;
    char *temp_name;
    if((default_prio >= cn_prio_sys_service) || (default_prio == 0))     //2
    {
        y_error_login(enum_knl_invalid_prio,"事件类型优先级非法");
        return cn_invalid_evtt_id;
    }
    int_save_asyn_signal();       //禁止调度也就是禁止异步事件
    for(evtt_id=0; evtt_id<cn_evtts_limit; evtt_id++)   //查找空闲的事件控制块
```

```
        if( tg_evtt_table[evtt_id].property.registered == 0)
            break;
    if(evtt_id == cn_evtts_limit)     //没有空闲事件控制块
    {
        y_error_login(enum_knl_no_free_etcb,"没有空闲事件控制块");
        int_restore_asyn_signal();
        return cn_invalid_evtt_id;
    }else if(evtt_name != NULL) //新类型有名字，需检查有没有重名
    {
        for(i=0; i<cn_evtts_limit; i++)
        {
            temp_name = tg_evtt_table[i].evtt_name;
            if(temp_name != NULL)           //事件类型重名
            {
                if(strcmp(temp_name,evtt_name) == 0)
                {
                    y_error_login(enum_knl_evtt_homonymy,"事件类型重名");
                    int_restore_asyn_signal();
                    return cn_invalid_evtt_id;
                }
            }
        }
    }
    tg_evtt_table[evtt_id].property = (struct evtt_property){0,0,1,0};
    tg_evtt_table[evtt_id].property.mark = mark;
    tg_evtt_table[evtt_id].property.overlay = overlay;
    tg_evtt_table[evtt_id].my_free_vm = NULL;
    tg_evtt_table[evtt_id].evtt_name = evtt_name;
    tg_evtt_table[evtt_id].default_prio = default_prio;
    tg_evtt_table[evtt_id].pop_sum = 0;
    tg_evtt_table[evtt_id].done_sum = 0;
    tg_evtt_table[evtt_id].repeats = 0;
    tg_evtt_table[evtt_id].vpus_limit = vpus_limit;
    tg_evtt_table[evtt_id].thread_routine = thread_routine;
    tg_evtt_table[evtt_id].stack_size = stack_size;
    tg_evtt_table[evtt_id].mark_unclear = NULL;
    tg_evtt_table[evtt_id].done_sync = NULL;
    tg_evtt_table[evtt_id].pop_sync = NULL;
    if((cn_run_mode!=cn_mode_mp)||(default_prio<0x80))          //3
    {//运行模式为 si 或 dlsp，或该事件类型拥有紧急优先级，需预先创建一个线程虚拟
机
        tg_evtt_table[evtt_id].my_free_vm =
                                __create_thread(&tg_evtt_table[evtt_id]);  //4
        if(tg_evtt_table[evtt_id].my_free_vm == NULL)
```

```
        {//内存不足，不能创建虚拟机
            y_error_login(enum_mem_tried,"创建虚拟机时内存不足");        //5
            return cn_invalid_evtt_id;
        }else
            tg_evtt_table[evtt_id].vpus = 1;
    }else
        tg_evtt_table[evtt_id].vpus = 0;
    int_restore_asyn_signal();
    return evtt_id;
}
```

代码 4-2 是登记事件类型的程序，说明如下：

1. 函数参数，参考 代码 4-1 和它的说明，很容易理解。
2. 优先级合法性判断，djyos 允许的优先级是 0~255，但这里只允许 1~249，是因为 0、250~255 是操作系统保留的，不允许用户使用。
3. 有些事件类型至少需要保留一个线程（参见第 4.3.5 节），必须在登记事件时为其创建第一个线程虚拟机，以期该类型事件只有一条事件被弹出时，可以立即获得虚拟机。在djyos中，线程虚拟机是事件的资源（参见第 4.2.4 节），使高优先级的事件快速获得虚拟机资源是必须的。
4. 此时创建的虚拟机必定是空闲虚拟机。
5. 登记事件类型时发现内存不足，只能返回错误，如果在为某具体事件创建虚拟机时发现内存不足，将把该事件挂到内存同步队列中。

4.3.3.3删除事件类型

当应用程序卸载，或者通过应用程序的动态配置功能取消软件的某些模块，又或者某类型事件不再需要时，就需要删除事件类型，回收其占用的事件类型控制块。删除事件类型由函数y_evtt_unregist函数完成，如 代码 4-3 所示。不能删除正在使用的事件类型，即事件队列中有本类型的事件尚未处理完（包括处理中、阻塞中、等待中），也不能删除事件类型的弹出同步队列和完成同步队列非空，即有事件在等待本类型事件弹出或完成。参见第 5.3 节和第 0 节。

代码 4-3 删除事件类型

```
bool_t y_evtt_unregist(uint16_t evtt_id)
{
    struct thread_vm *next_vm;
    struct event_type *evtt;
    bool_t result = true;
    if(evtt_id >= cn_evtts_limit)
        return false;
    evtt = &tg_evtt_table[evtt_id];
    if((evtt->property.in_use)||(evtt->done_sync != NULL)
                               || (evtt->pop_sync != NULL))
    {
```

```
        //事件类型正在使用或完成同步和弹出同步队列非空，不允许删除
        result = false;
    }else
    {
        next_vm = evtt->my_free_vm;
        //回收事件类型控制块，只需把 registered 属性清零。
        evtt->property.registered = 0;
        while(next_vm != NULL)   //释放该类型事件拥有的空闲虚拟机
        {
            m_free((void *)next_vm);
            next_vm = next_vm->next;
        }
    }
    return result;
}
```

4.3.3.4事件控制块数据结构定义

代码 4-4 定义的struct event_script是事件控制块的数据结构，是djyos操作系统调度的核心实体，在内存中，事件被组织成许多队列，调度器的核心工作就是决定哪一条事件被处理，哪一条事件应该等待，port_kernel.h文件中的cn_events_limit常量确定了系统中可以同时使用的最大事件数量，每个事件对应一个数据控制块，所需要的内存池由数组静态定义，与事件类型控制块一样，事件控制块内存池也不用固定块内存管理模块管理，而是用一个空闲队列把所有空闲块串起来（参见第 4.3.3.5 节）。每条事件都有一个ID号唯一标识，该ID号就是该事件控制块在数组中的偏移量。

代码 4-4 事件定义

```
struct event_status_bit
{    //所有位均为 0 表示事件刚弹出的原始状态,还没有进入调度
     uint16_t event_ready:1;              //就绪态
     uint16_t event_delay:1;              //闹钟同步
     uint16_t wait_overtime:1;            //超时等待
     uint16_t event_sync:1;               //事件同步
     uint16_t evtt_pop_sync:1;            //事件类型弹出同步
     uint16_t evtt_done_sync:1;           //事件类型完成同步
     uint16_t wait_memory:1;              //内存同步
     uint16_t wait_semp:1;                //信号量同步
     uint16_t wait_asyn_signal:1;         //异步信号同步
     uint16_t wait_mutex :1 ;             //互斥量同步
};
union event_status
{
     uint16_t all;
```

```
    struct event_status_bit bit;
};

struct event_script
{
    struct event_script *next,*previous;
    struct event_script *multi_next,*multi_previous;
    struct   thread_vm   *vm;        //处理本事件的线程虚拟机指针,
    struct   event_script *sync;     //同步事件队列,完成本事件后执行链表中的事件
    struct event_script **sync_head;     //记住自己在哪一个同步队列中，以便超时
                                         //返回时从该同步队列取出事件
    uint32_t      start_time;        //事件发生时间，tick 时钟
    uint32_t      consumed_time;   //事件消耗的时间
    uint32_t      delay_start;       //开始延时时间
    uint32_t      delay_end;        //延时结束时间
    uint32_t      error_no;          //本事件执行产生的最后一个错误号
    uint32_t      parameter0;        //事件参数 0,具体访问方式由程序员约定
    uint32_t      parameter1;        //事件参数 1,具体访问方式由程序员约定
    uint32_t      wait_mem_size;    //等待分配的内存数量.
    union event_status last_status;    //最后状态,用于查询事件进入运行态前的状态
    union event_status event_status;   //当前状态,本变量由操作系统内部使用,
    ufast_t       prio;                //事件优先级，取值 0~255
    uint16_t      evtt_id;             //事件类型 id
    uint16_t      sync_counter;        //同步计数
    //事件 id 范围:0~32767(cn_event_id_limit)
    uint16_t      event_id;    //事件序列编号,等同于事件在事件块数组中的偏移位置
    uint16_t      repeats;           //该事件开始执行前同类型事件积累发生的次数
    struct   dev_handle   *held_device; //本事件持有的设备指针，事件返回时强行释放设备
    struct  mem_record   *held_memory;  //动态分配尚未规还的内存,事件处理完成时不收回
                                        //将导致内存泄漏.
};
```

下面详细说明一下结构中各成员的用法，代码中注释很明确的就不重复了。

1. next，previous，multi_next，multi_previous四个指针，用于把事件连成各种队列，详见“第 4.3.3.5 节 空闲事件控制块队列”和“第 4.3.3.6就绪事件队列”和“第 5 章 同步”这几部分。
2. vm，指向分配给自己的线程，事件刚刚弹出时，vm=NULL，直到事件第一次切入时才分配虚拟机。
3. sync，同步指针，参见第 5.2 节。
4. sync_head，同步头指针的指针，事件因超时退出阻塞状态时，需要把事件从同步队列中取出，就必须知道它在哪一个同步队列中，参见第 5.8 节。
5. start_time，事件发生时间，在 y_event_pop 函数中取 32 位的时钟嘀嗒计数器的值。
6. consumed_time，处理该事件消耗的时间，操作系统在每次时钟嘀嗒异步信号（参见 第 6 章 中断）处理函数中，把当时正在处理的事件的consumed_time成员增量。

7. delay_start 和 delay_end，用于闹钟同步，delay_start 是调用 y_timer_sync 时时钟嘀嗒计数器的值，delay_end 是预设的闹铃响的时间。
8. wait_mem_size，用于内存同步，当事件申请内存不成功，就需要加入内存同步队列，wait_mem_size记录所申请的内存量。参见第 5.6 节。
9. last_status 记录事件的前一个状态，可以用来确认该事件的线程虚拟机是因什么原因激活的。如果 last_status==0，则说明是弹出后第一次切入。
10. sync_counter，在事件类型弹出同步和完成同步中充当同步条件计数器，参见第 5.2 节。
11. repeats，事件重复次数，本事件切入时事件类型控制块的 repeats 成员的值。
12. held_device，指向本事件处理中打开的设备的设备句柄队列，如果事件处理过程中总是能够及时关闭打开的设备，那么事件完成的时候，本指针应该指向 NULL，否则，将在 y_event_done 函数中强行关闭。
13. held_memory，指向本事件处理过程中申请的局部内存块的队列，作用同上。

4.3.3.5空闲事件控制块队列

空闲事件控制块队列把所有空闲事件控制块串在一起，如 图 4-1 所示。

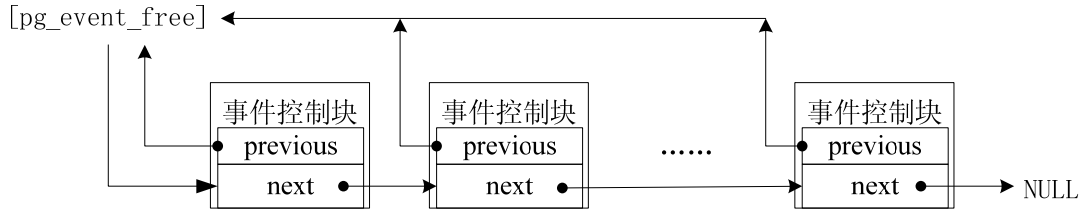


图 4-1 空闲事件控制块队列

事件控制块用 next 指针连接成一张单向不循环链表，而 previous 则指针指向&pg_event_free 表示该事件控制块是空闲的，pg_event_free 是一个全局变量，全局变量地址在整个运行期是唯一且不变的，可以用它来作为事件控制块空闲的标志。

if(event->previous == &pg_event_free)语句可以判断某事件控制块是否空闲。

下列代码片段可以从 pg_event_free 队列中取出一个事件控制块。

```
struct event_script *pl_ecb;
if(pg_event_free==NULL)                 //没有空闲的事件控制块
{
    y_error_login(en_knl_ecb_over,NULL);
    result = cn_invalid_id;
}else                                   //有空闲事件控制块
{

    pl_ecb = pg_event_free;             //从空闲链表中提取一个事件控制块
    if(pg_event_free->next == NULL)
        pg_event_free=NULL;             //这是最后一个事件控制块了
    else                                //不是最后一个事件控制块
    {
        pg_event_free = pg_event_free->next;  //空闲事件控制块数量减 1
```

```
    }
}
```

下列代码则用来把 pg_event_running 事件控制块加入空闲队列，当一条事件被处理完毕后，就需要这样做。

```
pg_event_running->previous = (struct  event_script*)&pg_event_free;
pg_event_running->next = pg_event_free;
pg_event_free = pg_event_running;
```

4.3.3.6就绪事件队列

顾名思义，就绪队列把所有处于就绪态（包括运行期和等待期）的事件串在一起，如 图 4-2 所示。系统中有一个优先级为 250 的系统服务事件，它总是处于就绪状态，因此，就绪队列用不会被置空。

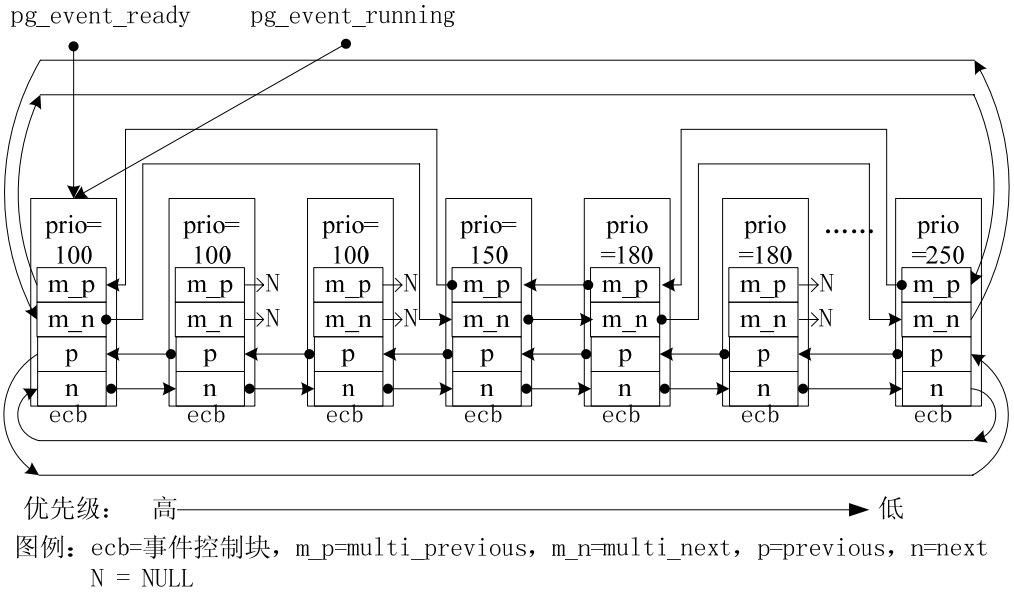


图 4-2 就绪事件队列

就绪队列其实是两个链表，一个链表由队列中所有事件的 previous 和 next 指针构成，以优先级排序，把所有事件串成一个双向循环链表。第二个链表只有部分事件参与，它用多功能指针 multi_previous 和 multi_next 把队列中相同优先级的首事件串成一个双向循环链表，例如优先级 100 的事件共有 3 个，只有第一个参与该队列，另外两个事件的多功能指针是空指针。

就绪队列是以优先级排序的队列，pg_event_ready 指针永远指向队列中最高优先级的事件。对相同优先级的事件，则按下列规则排列：

1. 新弹出的事件排在所有同优先级事件的最后。
2. 在运行过程中，非队列中最高优先级的、多个相同优先级事件的相对位置，不会发生改变。
3. 在运行过程中，若队列中最高优先级有多个相同优先级事件，则他们的排列顺序按下列规则决定：允许轮转调度，在轮转时间片到时旋转一位，原最前面的旋转到同优先级的最后，原第二位的转至队列首。无论禁止还是允许，正在处理的事件调用

y_timer_sync（0）（参见第 5.1 节），也旋转一位。

pg_event_ready 指针随着就绪队列的变化实时更新，它永远指向就绪队列首，根据 djyos 的调度算法，就绪队列首部的事件会立即切入，只要调度器发现 pg_event_ready 和 pg_event_running 不相等且允许调度，就会引发事件切换。因此，如果异步信号被允许（即调度被允许），pg_event_running 将一直跟踪 pg_event_ready 指针指向就绪队列首。但是如果异步信号被禁止期间有事件加入就绪队列，则可能会出现两个指针不等的情况。

把同优先级的首事件串起来有非常重要的意义，djyos 是一个实时操作系统，代码执行时间的确定性和快速性都非常重要，在__y_event_ready 函数中，需要遍历链表来查找新就绪事件的插入位置，如果没有首事件链表，则需要遍历整个就绪队列，有首事件链表则只遍历首事件链表即可；在__y_isr_tick 函数中，处理轮转调度时，需要找到 y_event_running 事件优先级相同的最后一个事件，然后把 pg_event_running 事件插入到它的后面。如果没有首事件链表，则在最坏情况下，同样需遍历整个就绪队列才能找到，如果有首事件链表，则 pg_event_running->multi_next->previous 就是同优先级的最后一个事件。

有两个函数用于操作就绪队列，分别是__y_event_ready和__y_cut_event，另外还可能因轮转调度而微调就绪队列（在__y_isr_tick函数和y_timer_sync函数中）。__y_event_ready函数用于把一个事件加入就绪队列，如 代码 4-5 所示。有两种情况需要使用本函数，一是新弹出事件，必然把新弹出的事件加入就绪队列，djyos不允许弹出事件时直接让事件进入某种阻塞状态，理由很简单，阻塞，不管是主动阻塞（比如闹钟同步）还是被动阻塞（申请资源得不到满足），都是被弹出的事件处理过程的一部分，原事件只能告诉操作系统发生了某某类型的事件，不能干涉新事件的处理过程，也就不能为新事件指定状态。二是事件脱离阻塞态，需要调用本函数把它加入到就绪队列。

代码 4-5 __y_event_ready

```
void __y_event_ready(struct    event_script *event_ready)
{
    struct    event_script *event;
    int_save_asyn_signal( );
    event_ready->event_status.bit.event_ready = 1;
    event = pg_event_ready;
    do
    { //找到一个优先级低于新事件的事件.由于系统服务事件总是 ready,因此总是能找到.
        if(event->prio <= event_ready->prio)
            event = event->multi_next;
        else
            break;
    }while(event != pg_event_ready);
    event_ready->next = event;
    event_ready->previous = event->previous;
    event->previous->next = event_ready;
    event->previous = event_ready;

    //新插入的事件在同优先级的最后，故这样能够判断新事件是否该优先级的唯一事件。
    //若是该优先级的唯一事件，也即使该优先级首事件，则需要将其插入首事件链表
    if(event_ready->prio != event_ready->previous->prio)
    {
```

```
            event = event_ready->next;
            event->multi_previous->multi_next = event_ready;
            event_ready->multi_previous = event->multi_previous;
            event->multi_previous = event_ready;
            event_ready->multi_next = event;
        }
        if(event_ready->prio < pg_event_ready->prio)
            pg_event_ready = event_ready;
        int_restore_asyn_signal();
}
```

__y_cut_event 函数把事件从就绪队列取出，当事件被删除时，将会调用本函数把事件从就绪队列取出然后释放之；当事件被阻塞时，调用本函数从就绪队列取出事件然后放进相应的同步队列。注意函数前面的双下划线，表明本函数是内部函数，只能由操作系统代码调用，应用程序是不能调用它的。djyos 不提倡应用程序直接操作这些核心队列，因此不提供控制核心队列的函数。

代码 4-6 __y_cut_ready_event 函数

```
void __y_cut_ready_event(struct event_script *event)
{
    struct event_script *pl_ecb;
    if(event != pg_event_ready)          //event 不是 pg_event_ready 队列头
    {
        if(event->multi_next == NULL)    //不是相应优先级的首事件
        {
            event->next->previous = event->previous;
            event->previous->next = event->next;
        }else                            //是相应优先级的首事件
        {
            pl_ecb = event->next;
            event->next->previous = event->previous;
            event->previous->next = event->next;
            if(pl_ecb->prio == event->prio)     //相应优先级不止一个事件
            {
                event->multi_previous->multi_next = pl_ecb;
                pl_ecb->multi_previous = event->multi_previous;
                event->multi_next->multi_previous = pl_ecb;
                pl_ecb->multi_next = event->multi_next;
            }else                                //相应优先级只有一个事件
            {
                //pl_ecb 即 event->multi_next.
                pl_ecb->multi_previous = event->multi_previous;
                event->multi_previous->multi_next = pl_ecb;
            }
        }
    }else                                //event 是 pg_event_ready 队列头
```

```
    {
        pg_event_ready = event->next;
        pl_ecb = event->next;
        event->next->previous = event->previous;
        event->previous->next = event->next;
        if(pl_ecb->prio == event->prio)     //相应优先级不止一个事件
        {
            event->multi_previous->multi_next = pl_ecb;
            pl_ecb->multi_previous = event->multi_previous;
            event->multi_next->multi_previous = pl_ecb;
            pl_ecb->multi_next = event->multi_next;
        }else                               //相应优先级只有一个事件
        {
            //pl_ecb 即 event->multi_next.
            pl_ecb->multi_previous = event->multi_previous;
            event->multi_previous->multi_next = pl_ecb;
        }
    }
}
```

4.3.3.7不能删除事件

前面提到，事件类型可以登记和删除，可能有些读者会认为，事件也能够弹出和删除，如果找不到“删除事件”这样的标题，一定会很失望的。且慢失望，djyos 确实不提供删除事件的功能，为什么呢？要删除某事件，唯一的原因就是该事件已经处理完毕了。按照模块独立性的原则，或者称为对象化的原则，事件一经弹出，这个事件就代表一个独立模块实体，或称对象实体，应该由事件处理程序自己判断自己是否已经处理完毕，对象只能自己决定自己的命运，别的模块无权干涉，也就不存在由其他模块删除它的可能性了。虽然在某些特定条件下，允许删除事件会给编程带来一时的便利，但从长远角度，这不是一个好习惯。而 djyos 根本就不提供这一功能，则从根子上杜绝了你随意弹出和删除事件的可能性。那么，如果因软件运行状态的变化，某些事件真的无需处理了，怎么办呢？没问题，只要在事件处理函数中判断自己是否需要进一步处理，如不需要，则直接调用 y_event_done 函数即可。

事件作为实体对象，不受其他实体控制，还表现在djyos中几乎没有以事件id、事件指针为参数的系统服务，完全杜绝了你通过事件指针或事件id去操控别的事件的可能。唯一的一个以事件id为参数的是y_event_sync函数（参见第 5.2 节），该函数也不是要操控目标事件，而是要把正在处理的事件挂到目标事件的同步队列中去，对目标事件本身完全没有影响。

4.3.3.8线程虚拟机数据结构定义

代码 4-7 定义的struct thread_vm是处理具体事件的线程虚拟机，事件切换的具体操作就是切换处理该事件的线程，有意思的是，struct thread_vm中并没有入口函数指针，这强烈地传达这样一个信息，线程虚拟机只是事件的资源，具体执行什么函数由事件类型说了算，

因此，线程入口函数的指针出现在事件类型控制块的thread_routine成员中。再者，虚拟机的栈尺寸也是由事件类型控制块的stack_size成员决定的。因此，线程虚拟机是一种专用资源，它只适用于执行特定事件类型的处理函数，只能分配给类型相匹配的事件。

代码 4-7 虚拟机类型定义

```
struct process_vm                   //进程虚拟机，用于 mp 模式，暂时空着
{
};
struct  thread_vm                   //线程虚拟机数据结构
{
    uint32_t    *stack;             //虚拟机栈指针,在虚拟机被抢占时保存 sp,
                                    //运行时并不动态跟踪 sp 变化
    uint32_t    *stack_top;         //虚拟机的栈顶指针
    uint32_t    stack_size;         //栈深度
    struct thread_vm *next;         //把 evtt 的所有空闲虚拟机连成一个单向开口链表
                                    //该链表由 evtt 的 my_free_vm 指针索引
    struct process_vm *host_vm;     //宿主进程虚拟机，在 si 和 dlsp 模式中为 NULL
};
```

各成员说明如下：

1. stack，虚拟机的栈指针，它在事件被切出时保存虚拟机当前栈指针，尔后事件被重新切入时，则从 stack 成员取得栈指针，从该指针所指位置恢复上下文。事件处理过程中，stack 成员并不跟踪栈指针变化。
2. stack_top，栈顶地址，为用户提供一些调试信息，在线程执行过程中并不使用。
3. stack_size成员，说明分配给本虚拟机的栈尺寸。虚拟机是用来处理某类型事件的，执行的是该事件的处理函数，在事件类型控制块中的stack_size成员说明了该函数需要的栈空间。创建虚拟机时，分配给虚拟机的栈空间与事件处理函数需要的栈空间有关联，但并不相同，而是比后者大。至于两者为什么会不等，参见第 11.3.4 节说明。
4. next，如代码注释。
5. host_vm，宿主进程虚拟机，在支持多进程的 mp 模式才有用。

4.3.3.9创建线程虚拟机

要使事件能够被处理，必须为其创建线程虚拟机，在cpu.c文件中的__create_thread函数用于创建线程虚拟机，如 代码 4-8 所示。创建虚拟机的过程相当简单，就是分配虚拟机栈然后初始化栈，最后把虚拟机复位。这是一个移植关键函数，与目标CPU的栈生长方向有关。

代码 4-8 创建线程虚拟机

```
struct    thread_vm *__create_thread(struct    event_type *evtt_id)              //1
{
    struct    thread_vm    *result;
    ptu32_t    len;
    //计算虚拟机栈:线程+最大单个 api 需求的栈
    len = evtt_id->stack_size+cn_kernel_stack +sizeof(struct    thread_vm);       //2
```

```
    //栈顶需要对齐，malloc 函数能保证栈底是对齐的，对齐长度可以使栈顶对齐
    len = align_up(len);                                                    //3
    result=(struct    thread_vm    *)m_malloc_gbl(len);                    //4
    if(result==NULL)
    {
        y_error_login(en_mem_tried,NULL);
        return result;
    }
    result->stack_top = (uint32_t*)((ptu32_t)result+len);                   //5
    result->next = NULL;                                                    //6
    result->stack_size = len - sizeof(struct thread_vm);
    result->host_vm = NULL;
    //复位虚拟机并重置线程
    __asm_reset_thread(evtt_id->thread_routine,result);                     //7
    return result;
}
```

代码说明如下：

1. 以事件类型 id 为参数，强调虚拟机是作为资源只能为该类型的事件服务的，它最终调用该类型事件的处理函数。
2. 计算线程所需要的栈空间，方法是：事件处理函数thread_routine需要的栈空间（用户部分）+线程可能调用操作系统服务需要的栈空间（系统部分），用户部分栈尺寸计算方法参见第 2.8.2 节，必须在登记事件类型时（参见第 4.3.3.2 节）写到事件类型控制块的stack_size成员中。
3. 对齐栈尺寸，栈是线程运行时用于保存上下文和局部数据的地方，C 编译器对此是有一定的对齐要求的，比如 ARM 的 ATPCS 就要求栈顶地址按 8 字节对齐。栈空间是从堆中动态分配的，可以保证其最低地址是对齐的，如果把栈尺寸也按要求对齐的话，就可以确保栈顶也是对齐的。align_up 是在 port_kernel.h 文件中定义的宏，特别提醒，移植的时候，一定要按照目标系统编译器的要求重写 align_up 宏。
4. 为栈分配内存，注意这里必须分配全局内存，以避免其在y_event_done函数中被自动回收，而必须等到删除虚拟机时显式调用m_free函数来回收。m_malloc_gbl函数返回的是内存块的低地址，栈底地址被赋予线程控制块，也就是说，本实例的目标系统栈是向下生长的。在源代码中，本函数被标注为“移植关键”，就是因为本行和注释 5 行，详见 第 15 章第 15.2.4.1 节。
5. 初始化线程栈顶地址，与栈结构有关。
6. 接下来几行初始化线程控制块，没什么好说的。
7. 复位线程虚拟机，这是一个有底层平台相关的汇编函数，参见 第 15 章。

执行本函数后，新创建的虚拟机的栈结构如 图 4-3 所示。

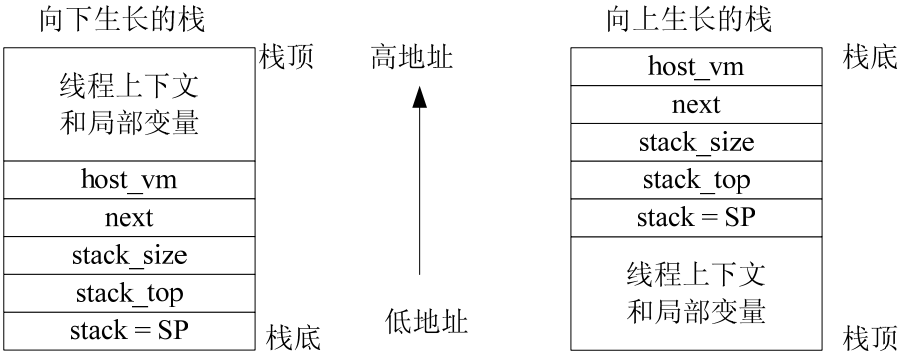


图 4-3 虚拟机的栈结构

4.3.3.10 删除线程虚拟机

由于 djyos 中，虚拟机只是事件的资源，所以，删除虚拟机也像删除其他资源一样简单，虚拟机由共享的代码、数据和私有的栈组成，删除虚拟机就是把栈空间占用的内存回收就可以了，只需一行语句：

```
m_free((void*)pg_event_running->vm);
```

就可以完成删除虚拟机，随后把该事件类型拥有的虚拟机数量减量就可以了。

```
pl_evtt->vpus--;
```

4.3.4 虚拟机分配与暂存

下列语句把一个虚拟机 vm 分配给事件 ev，这里略去了指针初始化的过程。

```
struct event_script    *ev;
struct thread_vm       *vm;
ev-> vm = vm;
```

而要把一个虚拟机 vm 暂存，则用下列语句，把 vm 挂到对应的事件类型控制块的 my_free_vm 队列就可以了，这里同样略去了指针初始化过程。

```
struct event_type *evtt;
vm->next = evtt->my_free_vm;
evtt->my_free_vm = vm;
```

暂存虚拟机发生在某事件处理完成后，虚拟机需要回收再利用的场合。

4.3.5 永久性线程虚拟机资源

线程虚拟机作为处理事件所需要的资源，任何事件在开始处理前必须先取得虚拟机，而创建线程虚拟机需要从堆中动态分配内存，虽然 djyos 的块相联内存分配法快速高效且高时间确定性，但还是要消耗不少时间，且还可能因内存不足而导致阻塞，甚至分配失败。默认优先级小于 128 的事件类型（数值越低，优先级越高）属于紧急优先类型，处理紧急事件需要快速可靠地获得所需要的资源，特别是线程虚拟机这种重要资源。因此，操作系统为此类事件至少保留一个线程，让其处于复位后随时可以启动的状态，但并不立即启动。待相应类

型的事件发生且被切入时，操作系统就把这个线程分配给该事件，然后立即启动。在登记这种类型的事件时，操作系统将创建一个线程虚拟机作为该类型事件的永久性资源，

创建和删除虚拟机需要分配动态内存，在si模式和dlsp模式下（运行模式参见第 4.5 节），由于这两种模式不支持虚拟地址映射，包含操作系统和应用程序在内的整个系统共用一个有限的地址空间，该地址空间与硬件所配备的物理存储器尺寸相同，因此容易受反复创建和删除线程虚拟机导致的内存碎片化。因此在si和dlsp模式下，包括优先级大于 127 在内的所有类型的事件，操作系统都至少为其保留一个线程虚拟机，因此在注册事件类型时操作系统会为其创建一个线程虚拟机作为其永久性资源。只要经过仔细设计，不要发生同一类型的前一事件未完成又发生新事件的情况，存在这个永久性资源使得每次事件处理都能够享用这个虚拟机，避免频繁创建和删除。支持虚拟内存的mp模式由于寻址空间可达 2G，对内存碎片没那么敏感，故登记优先级大于 127 的事件类型时不会为其创建线程虚拟机，只有在该类型事件发生后且被切入时才临时创建线程虚拟机，并且在事件处理完成后立即删除该虚拟机以回收资源。

永久性线程虚拟机资源是在登记事件类型时创建的，但永久性资源的含义，并不是指该虚拟机永不消灭，而是指该类型事件至少拥有一个虚拟机。如果某类型事件重复弹出，操作系统可能会为其创建多个虚拟机，当事件处理完毕，这些虚拟机将被删除，拥有永久性资源的事件类型，最后一个虚拟机而不是登记事件类型时创建的虚拟机，将被保留。

登记事件类型的代码参见本章 代码 4-2，其中第 3~5 条说明是关于创建永久线程虚拟机资源的。

4.3.6 线程虚拟机操作函数

前面讲到，djyos使用线程虚拟机来执行事件处理函数。调度器决定处理哪个事件，暂停处理哪个事件，是通过为事件创建线程虚拟机、把处理事件的线程切入和切出来实现的。几个线程虚拟机操作函数为调度器提供了至关重要的底层支持，这些函数与CPU寄存器和指令系统的关系非常密切，一般用汇编实现，属于移植敏感函数 ，在第 15.2.4 节有详细说明，这里仅简单列一下。

1. 复位线程虚拟机

__asm_reset_thread( );函数用于复位线程虚拟机的上下文到起始状态。

2. 复位老线程，切入新线程

__asm_reinit_to( );函数把一个线程虚拟机复位，然后切换到新线程。当一个事件处理完成，如果其虚拟机仍有保留价值，就应当调用本函数，把原虚拟机复位待用，再切入新线程。

3. 上下文切入

__asm_turnto_context ( );函数直接切入目标线程，当一个事件处理完成，如果不保留其虚拟机，便调用本函数直接切入新线程。初始化完成后启用多事件调度器时，也是调用本函数切入第一个线程的上下文。

4. 上下文切换

__asm_switch_context( );线程运行中被阻塞、或被抢占，就会调用本函数，把原线程的上下文保存到它的栈中，把新线程的上下文从其栈中弹出恢复到 CPU，新线程继续运行。

5. 从异步信号 ISR 中返回时的上下文切换

__asm_switch_context_int( )；如果在异步信号 ISR 中弹出比正在处理的事件更高优先级的事件，或者使更高优先级的事件就绪，就应该在中断返回主程序前调用本函数。本函数制造两个假象，一是被 ISR 中断的似乎是需要被切入的线程；二是原被 ISR 中断的线程似乎不是被 ISR 打断而是被阻塞或被抢占的。

4.3.7 事件状态

djyos 系统中事件状态只有两种：阻塞态和就绪态。

4.3.7.1就绪态

就绪态就是事件准备好可以被处理的状态，处于就绪态的事件，可能在就绪队列中等待调度器的眷顾，也可能已经被调度器切入正在处理。就绪态是事件的初始状态，事件一经弹出，就进入就绪队列。如果事件在线程中被弹出，事件被弹出后，如果该事件的优先级比正在处理的事件高，则立即切入 CPU，否则该事件在就绪队列中等待调度器的眷顾。如果在异步信号 ISR 中被弹出，且优先级足够高，则在 ISR 返回（若嵌套则在最后一级返回）时立即切入。处理中的事件，如果弹出比自己更高优先级的事件，或者有更高优先级的事件就绪，又或者轮转调度条件符合，也将立即切出，进入等待队列。

4.3.7.2阻塞态

阻塞态是指事件因同步原因而不能继续处理的状态， djyos把可能导致阻塞的操作称之为同步，意为阻塞自己以等待被同步的条件发生，然后再开始运行，使自己步调与该条件保持一致。djyos允许多种同步对象，根据被同步的对象不同，阻塞态有多种，但是同一时间只能有一种，各种阻塞态的进入和退出如 表格 4-1 所示。关于同步更进一步的说明参见第第 5 章。需要注意的是，事件退出阻塞态以后，只是进入了就绪队列，能否立即被切入还得看自己的造化（优先级是否足够高）。

表格 4-1 各种阻塞态的进入和退出条件

状态	描述	进入条件	退出条件
闹钟同步	事件被阻塞指定时间，时间到以后被叫醒，就像闹钟一样。闹钟同步队列是以结束时间由近到远排列的双向链表。	调用 y_timer_sync	定时时间到，在__y_isr_tick 函数中退出
异步信号同步	事件阻塞以等待某异步信号发生，该异步信号发生后 ISR 函数返回时被唤醒。每个异步信号只能被一个事件同步，不是队列。	调用 int_sync_asyn_signal	异步信号引擎中返回主程序前退出
互斥量同步	事件阻塞以等待所请求的互斥量有效。该同步队列是以优先级排列的双向链表，每次互斥量有效时就从队列头部取出一个事件，并把互斥量交给它。	调用 mutex_pend 函数，但没有获得互斥量	调用 mutex_post 函数使被请求的互斥量可用时退出。
信号量同步	事件阻塞以等待所请求的信号量有效。该同步队列是以优先级排列的双向链表，每次信号量有效时就从队列	调用 sem_pend 函数，但没有获得信号灯	被请求的信号灯可用时退出。

	头部取出一个事件，并把信号灯交给它。		
内存同步	从堆中分配内存时，若可用内存不足，就会阻塞当前线程，等待内存可用。同步队列是以所申请的内存块大小从小到大排列的双向链表。	调用 malloc 族函数时，内存不足	m_free 被调用且释放后最大内存块满足申请的内存。
事件类型弹出同步	阻塞直到某类型的事件发生设定次数。同步队列是一个不排序的双向链表。	调用 y_evtt_pop_sync	阻塞后，该类型事件弹出设定次数后退出
事件类型完成同步	阻塞直到某类型的事件完成设定次数，同步队列是一个不排序的双向链表。	调用 y_evtt_done_sync	阻塞后，该类型事件被完成设定次数后退出。
事件同步	阻塞以等待某一条事件处理完毕。同步队列是一个不排序的双向链表。	调用 y_event_sync	目标事件处理完毕
备注	1、在满足阻塞条件时，只有在允许调度的情况下才会引起阻塞，否则返回错误，malloc 族则返回 NULL 指针。 2、以上所有会引起阻塞的函数，均可以在异步信号 ISR 中调用，但在 ISR 中调用时，不会进入阻塞。比如在 ISR 中调用 malloc，如果分配成功则返回分配结果，如果分配不成功，则返回 NULL，不会阻塞。		

4.3.7.3状态变化

事件被弹出以后，就携带主人赋予它的使命，开始了其生命周期旅行，它的一生由此开始。事件一经弹出，它就有了生命，除非它已经完成了使命寿终正寝，否则没有任何人可以杀死它，操作系统也不会杀死它，因为djyos把每一个事件都视为独立模块，按照模块间互相独立的原则，没有任何其他模块可以操控它。事件的一生，可能异常顺利，一出生就得到操作系统的眷顾，立即切入CPU，直到处理完毕，“弹出——处理——完成——消亡”一气呵成，这只有高优先级的名门贵族才可能有的待遇。也可能命运多舛，经历反复多次阻塞、被抢占，历尽坎坷才最终完成任务。图 4-4 显示的是事件一生中可能经历的状态变迁。

事件弹出后，如果不是 mark 型事件，就会立即进入就绪队列，如果是 mark 型事件，则要看是否有相同类型的事件正在处理，如果有，则要等该事件处理完才能进入就绪队列。事件进入就绪队列后，并不意味着可以开始处理了，就绪队列是一个按优先级排列的队列，最高优先级的事件排在最前面，调度器总是把最前面的事件切入。事件被切入后，仍然留在就绪队列中。就事件状态本身来说，等待中和运行中并无区别，事件状态结构 struct event_status_bit 中，只有 event_ready 位，并没有标志位表明该事件处于等待中还是运行中。

线程入口函数返回，或者在事件处理过程中调用y_event_done函数，就表明该事件已经处理完毕，操作系统将回收该事件占用的资源，比如用malloc从堆中申请的内存、打开的设备。如果该线程需要保留（参见第 4.3.5 节），就把它挂在事件类型的my_free_vm指针下，否则，该虚拟机将被删除。

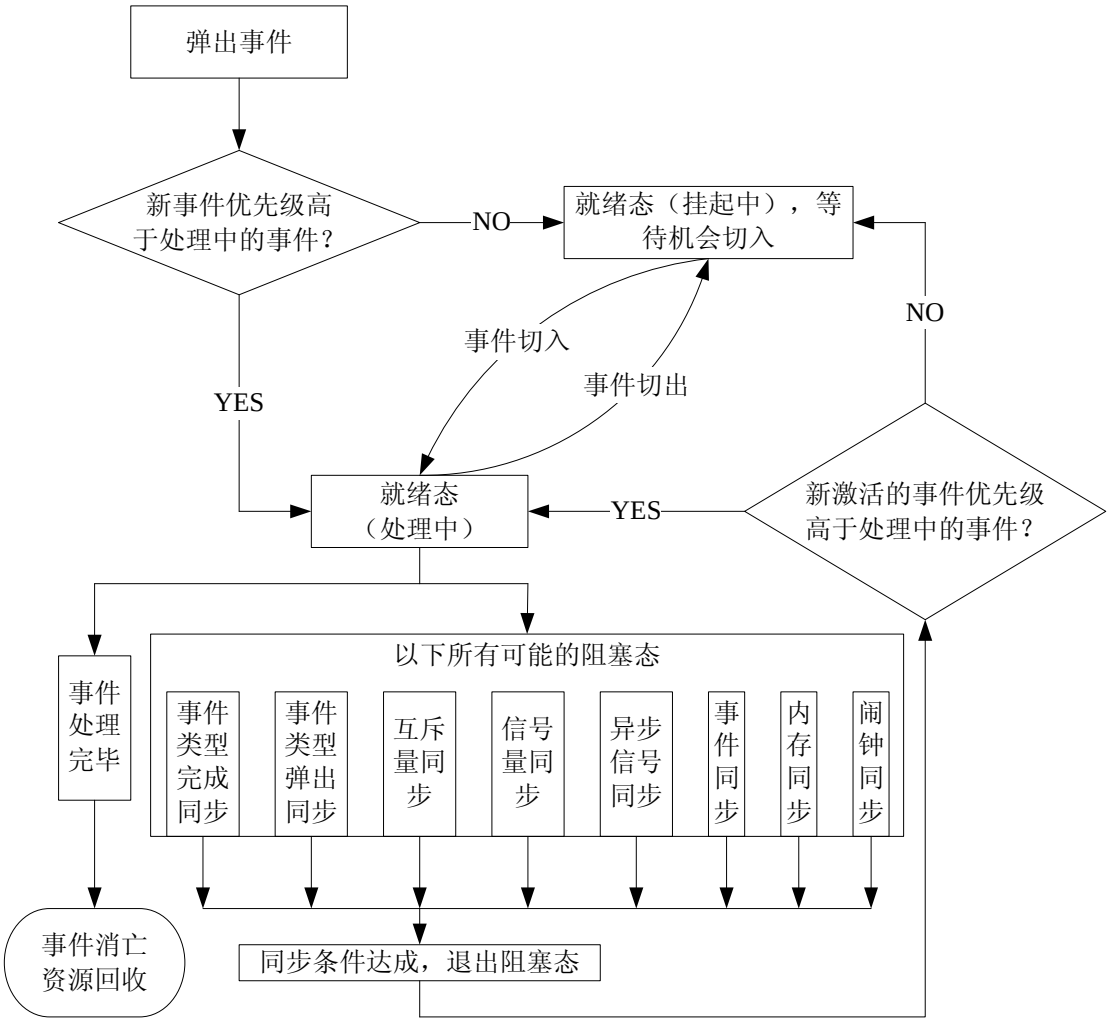


图 4-4 事件状态变迁图

4.3.8 事件优先级体系

djyos使用严格的优先级调度策略，它的优先级策略涉及到包括中断在内的整个软件系统，本节只讲述有关事件优先级的内容，要看到djyos优先级体系的全貌，请参阅第 6.2.1.1 节。

每个事件都有优先级，应用程序事件的优先级范围是 1~249，数值越低优先级越高，0 级和 250~255 之间的优先级由操作系统保留。事件的优先级是终身不变的，djyos 不提供半道修改事件优先级的功能，要求程序员在设计初期认真规划事件和事件类型的优先级。设定事件优先级的途径有两个：

1. 在弹出事件时指定优先级，调用 y_event_pop 弹出事件时，参数 prio 用于设定该事件的优先级，如果 prio=0，则使用事件类型的默认优先级。注意，default_prio=0 的事件类型并非不存在，而是由操作系统保留使用。
2. 继承事件类型的默认优先级，在调用 y_evtt_regist 时设定了默认优先级，正确设置事件类型优先级很重要，在绝大多数情况下，都会使用这个优先级。该优先级保存在 事件类型控制块的 default_prio 成员中。

每个类型的事件都有一个默认优先级，应用程序报告事件时，如果不设定事件优先级，新事件将继承事件类型的优先级。优先级最多为 256 级，但应用程序只能使用 1~249 级，djyos

对优先级做了如下约定，这些约定虽然不是强制性的，比如有 3 个优先级为 100 的事件同时就绪，虽然 100 不是建议的轮转优先级，但只要轮转调度被允许，他们还是会轮流执行。如果轮转调度被禁止，则虽有多条优先级为 200 的事件同时就绪，他们还是只能按照先到先处理的顺序处理，不会轮流执行。虽如此，为了移植方便，建议程序员遵守这个约定，尤其是支持多道程序设计时，必须遵守这个约定，参见第 3.1.4 节。

0，251~255：系统保留。

250：系统服务事件使用。

1~249：应用程序可以使用。

201～249：后台优先级组，用于低优先级的后台服务。

200：轮转优先级，所有轮转调度的事件类型使用。

128～199：实时优先级组。

1～127：紧急优先级组。

4.3.9 事件切换

4.3.9.1何时执行事件切换

djyos 的调度算法可简述为以下几点：

1. 永远只处理处于就绪态的最高优先级的事件，如果在事件运行中，弹出了比自身优先级高的事件，新弹出的事件将立即抢占自己；如果有比自己优先级高的事件进入就绪态，也将立即抢占。
2. 如果时间片轮转算法被允许，则处于相同最高优先级的若干个事件将轮流处理，每个事件占用 CPU 的时间由 y_set_slice 函数设定。时间片的大小只能全局设定，不允许每个事件单独设定，单位是 tick，默认值是 1 个 tick。
3. 如果时间片轮转被禁止，相同优先级的多个事件按先弹出先处理的顺序。
4. 调度器可以被禁止，禁止调度器和禁止异步信号是等同的，关于异步信号，详见 第 6 章。调度被禁止后，唯一能引起上下文切换的是，某事件处理完毕，处于就绪态的最高优先级线程将被切入。

从第 4.3.3.6 节可知，djyos的就绪队列是一个以以优先级排队的队列，pg_event_ready始终指向需要马上处理的事件，而pg_event_running始终指向正在处理的事件，查看是否需要切换事件，就是比较pg_event_running和pg_event_ready两个指针是否相等。那么，什么时候应该查看是否需要切换事件呢？在djyos操作系统中，导致需要切换事件的原因，必定是因为改变就绪队列，否则pg_event_running和pg_event_ready两个指针的关系不会发生变化，改变就绪队列的地方有三个：

1. 在事件处理线程中改变了就绪队列，操作就绪队列是需要关闭调度的，应该在调用重开调度前检查是否需要切换事件，参见“第 6 章 中断”的 代码 6-3；
2. 异步信号ISR中（异步信号ISR参见第 6.2.4 节）改变了就绪队列，应该在从异步信号引擎__int_engine_asyn_signal函数中、最后一级异步信号嵌套返回主程序之前判断是否需要切换事件，参见“第 6 章 中断”的 代码 6-6。
3. 某事件处理完成，当然不需要再占用 CPU 了，就应该把自己从就绪队列中取出，把 CPU 让给就绪队列头部的事件。这与事件处理中修改就绪队列不同，因为正在执行的事件即将被销毁，其虚拟机也即将被销毁或复位。

被切入的事件分两种情况，一是该事件是途中切入，即事件在处理过程中因阻塞或者优

先级抢占的原因被切离后重新切入；二是首次切入，即该事件尚未开始处理，是第一次切入。无论哪种情况，事件切换的调度过程是一样的，调度器并不知道事件是途中切入还是首次切入，调度器只有一个目标：使立即处理就绪队列头部的事件。再细化一些，首次切入涉及到分配虚拟机资源，又使得它与途中切入有些不同，具体表现为在__y_select_event_to_run函数中将执行不同的条件分支，详见第 4.3.11.2 节。

4.3.9.2事件到事件切换

如果在事件处理函数中改变了就绪队列，将导致事件切换，正在运行的事件上下文将保存在自己的栈中，新切入事件的上下文将从其栈中弹出恢复到CPU寄存器中，CPU将由处理旧事件转而处理新事件。切换由__schedule函数完成切换，切换流程如 图 4-5 所示，程序如代码 4-9 所示。函数返回一个布尔值，true表示成功切换到别的事件，false表示没有发生事件切换而直接返回。

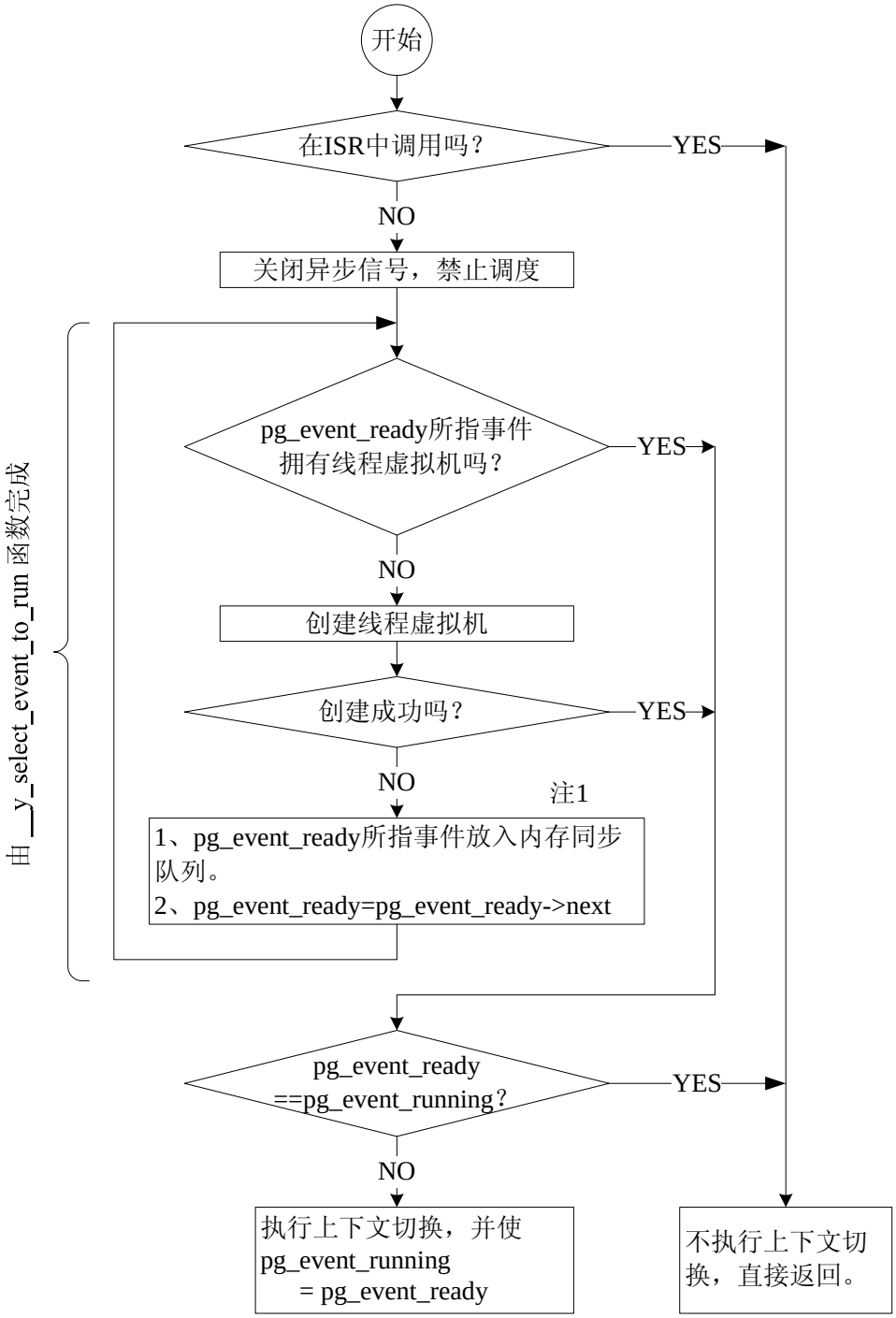


图 4-5 事件切换流程

注 1：创建线程虚拟机不不成功的原因只能是内存不足。内存同步队列参见第 第 5 章。

代码 4-9 调度函数

```
bool_t __schedule(void)
{
    struct    event_script *event;

    if(tg_int_global.nest_asyn_signal != 0)                    //1
        return false;
    int_save_asyn_signal();     //在上下文切换期间不能发生中断
```

```
    __y_select_event_to_run();                                          //2
    if(pg_event_ready != pg_event_running)                              //3
    {
        u32g_running_start=y_get_time();                                //4
        event = pg_event_running;
        pg_event_running=pg_event_ready;
        __asm_switch_context(pg_event_ready->vm ,event->vm);            //5
    }else
    {
        int_restore_asyn_signal();
        return false;
    }
    return true;
}
```

代码说明如下：

1. 本句是容错用途，如果在异步信号ISR内调用本函数，不允许继续执行，否则会产生不可预料的后果。这是可能发生的，因为djyos对异步信号ISR几乎没有限制，它确实可能会调用诸如y_event_pop之类函数，这些函数会成对调用int_save_asyn_signal和int_restore_asyn_signal。从第6.2.1.4节和图 6-7可知，如果在ISR中严格成对调用这两个函数，是不会调用__schedule的，如果因软件缺陷导致多调用int_restore_asyn_signal一次，则会调用__schedule。
2. 参见 代码 4-12。
3. 正常情况下，只有 pg_event_ready 和 pg_event_running 不相等时才会调用本函数，本条件句看起来似乎是多余的，但确实可能发生这样的情况，即 pg_event_running 指向的事件仍然在就绪队列中，但从 pg_event_ready 和 pg_event_running 之间的事件全部因内存不足不能创建虚拟机，就会使执行__y_select_event_to_run 后，pg_event_ready 又与 pg_event_running 指向同一事件。如果发生这样的情况，则无需切换。
4. u32g_running_start 是一个 32 位的全局变量，记录最后一次切换发生时间，也可以说是处理中的事件本次被切入的起始时间。
5. 把上下文从处理中的事件切换到新切入的事件，是一个汇编函数，移植关键，参见“第 15 章 djyos移植”。

4.3.9.3异步信号 ISR 中执行事件切换

如果在异步信号ISR中改变了就绪事件队列，参见“第 6 章 中断”的 图 6-6，由异步信号ISR引擎__int_engine_asyn_signal函数在即将返回主程序的时候执行事件切换。切换由__schedule_asyn_signal函数执行，对比 代码 4-10 和 代码 4-9，可以发现中断中切换的流程与 图 4-5 非常相似，不同点只有三个：

1. __schedule_asyn_signal 函数无需判断是否从 ISR 中调用，因为它必然会从 ISR 调用，且不会出现从非 ISR 中调用的危险。__schedule_asyn_signal 是内部函数，用户是不可能调用的，而__schedule 却可以通过 api 函数 int_restore_asyn_signal 间接调用，存在误调用的危险。

2. 无需关异步信号，__schedule_asyn_signal 函数本来就在异步信号 ISR 中调用。
3. 调用__schedule_asyn_signal 调用__asm_switch_context_int 执行事件切换，而__schedule调用__asm_switch_context执行事件切换，这两个切换函数有巨大的不同，参见第 4.3.6 节。

代码 4-10 异步信号中的事件切换

```
void __schedule_asyn_signal(void)
{
    struct  event_script *event;

    __y_select_event_to_run();
    if(pg_event_ready != pg_event_running)
    {
        u32g_running_start=y_get_time();
        event=pg_event_running;
        pg_event_running=pg_event_ready;
        __asm_switch_context_int(pg_event_ready->vm,event->vm);
    }else
    {
         //不同于__schedule 函数，这里无需做任何操作。
    }
    return;
}
```

4.3.9.4事件处理完成后切换到就绪事件

一旦事件处理完成，就应该调用 y_event_done 函数通知操作系统:“我的使命已经完成”，事件处理函数自然返回也可以起到相同的效果，但不推荐这样。事情已经办完了，自然也就不能再占着 CPU，需要切换到正需要处理的事件——pg_event_ready 正指着它呢。

事件完成切换是在y_event_done函数中直接实现的，该函数为已经完成的事件料理后事，然后切换到pg_event_ready事件所指的事件。参见 代码 4-14 中最后执行切换的部分。

4.3.10 事件重复弹出

事件重复弹出是指相同类型的事件，早先弹出的事件尚未处理完成（包括还没有开始处理、正在处理、被抢占、被阻塞），又有新事件弹出。事件和事件类型控制块中与重复弹出事件相关的成员有：

事件控制块的 repeats 成员：本事件首次切入 CPU 时同类型的事件尚未完成处理的数量，即本事件切入时事件类型控制块的 repeats 成员的值。

事件类型控制块的 repeats 成员：该类型事件被重复弹出的次数，如果是 mark 型事件，则记录本类型事件上次执行 y_clear_mark 至今弹出的次数，如果不是 mark 型事件，则记录在事件队列中所有本类型事件（即未处理完成的事件）的总和。

事件类型控制块的 vpus 成员：本事件类型所拥有的线程虚拟机总数，包括已经分配给被类型事件的虚拟机和挂在 my_free_vm 队列中的空闲虚拟机。

事件类型控制块的 vpus_limit 成员：虚拟机数量限制，本事件类型同时占有的虚拟机数量（含空闲的和已分配给事件的）不能超过此限制，默认是 10。本成员对 mark 型事件无效，mark 型事件只用一个虚拟机

事件类型控制块的属性成员 property 的 in_use 位：只要有本类型事件尚未处理完成（含处理中、阻塞中、等待中），就设为 1，否则为 0.

事件类型控制块的 mark_unclear 成员：mark 型事件弹出后，mark_unclear 指针将指向该事件，直到事件处理函数调用 y_clear_mark 函数才置为 NULL。

如果是 mark 型事件，则该类型的事件同一时间只能有一条正在处理，在清除 mark 标记前重复弹出，则只是使事件类型控制块 repeats 成员增量。如果在 mark 标记被清除之后重复发生，则新事件需要单独处理，但不能立即加入就绪队列，而是在事件类型控制块的 mark_unclear 指针下暂存，待先前弹出的事件处理完成后加入就绪队列。任何 mark 型事件完成后，发现 mark_unclear 指针非空，虚拟机将直接转交给新事件。

如果非 mark 型事件重复发生，则该事件会直接加入到就绪队列，但不会立即为其创建虚拟机。同一类型的事件，其同时可创建的事件类型是有上限的，事件类型控制块的 vpus_limit 成员表示该上限。如果数据控制块的 repeats 已经等于 vpus_limit，则不能弹出新事件。

事件处理完成后，其虚拟机是删除、保留还是转交给新事件，是在y_event_done函数中处理的，参见第 4.3.11.3 节。

4.3.11 事件生命周期

djyos系统下软件的运行轨迹是不断地弹出事件和处理事件的过程，一条事件从弹出到消亡的生命周期如 图 4-6 所示，事件从开始处理到处理完毕的过程如 图 4-7 所示。图中显示，如果事件处理函数永不返回，该事件就永远没有处理完毕的时候，相应的线程虚拟机也永不终结。这与传统操作系统的线程运行模式有些类似，djyos的键盘扫描事件就是按这种方式工作的典型事件，在系统初始化阶段执行的键盘模块初始化函数中，弹出了一条“键盘扫描”事件，多事件调度启动以后，该事件在适当的时候被切入，其事件处理函数永不返回，而是利用闹钟同步（参见第 5.1 节）每隔一定事件扫描一次键盘。

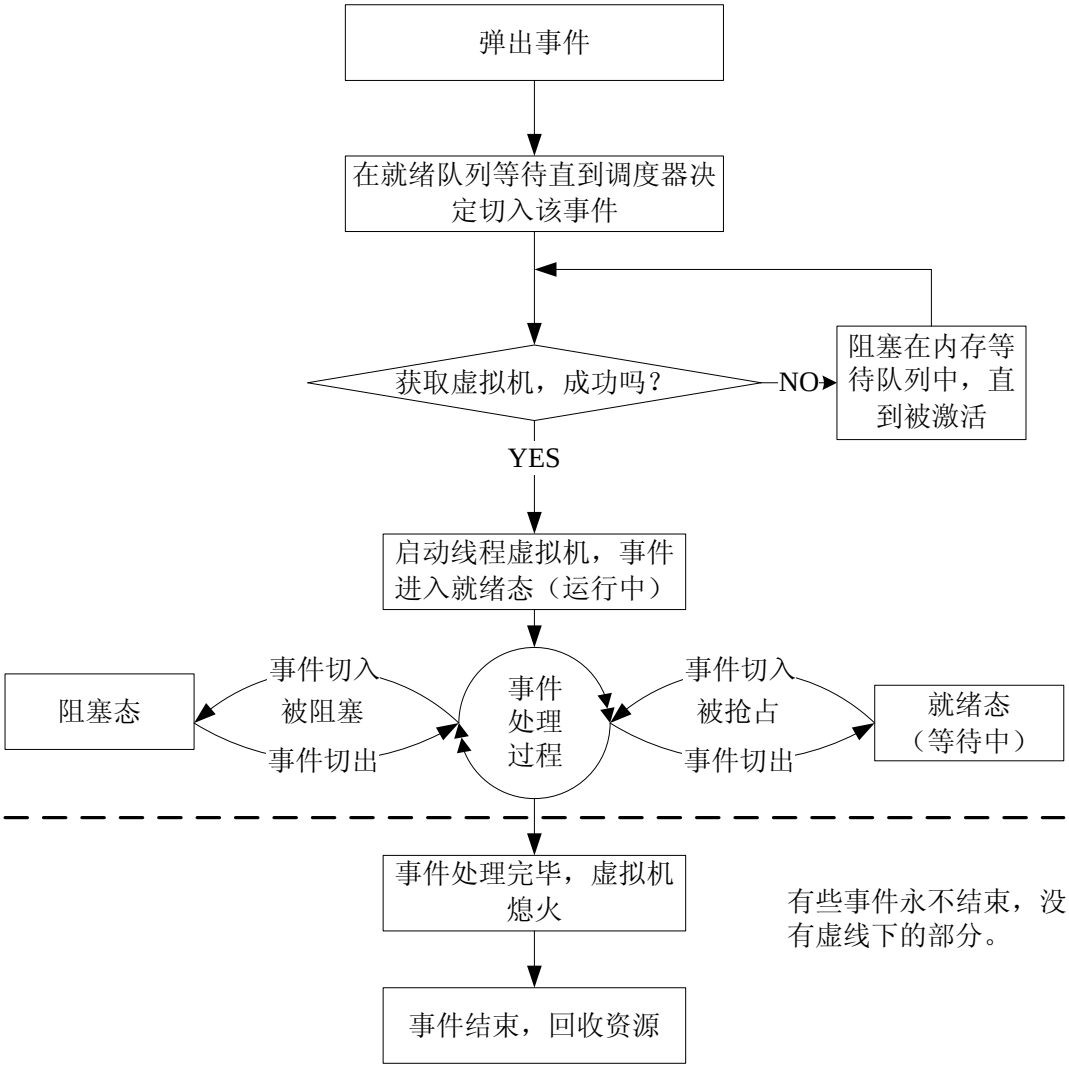


图 4-6 事件生命周期

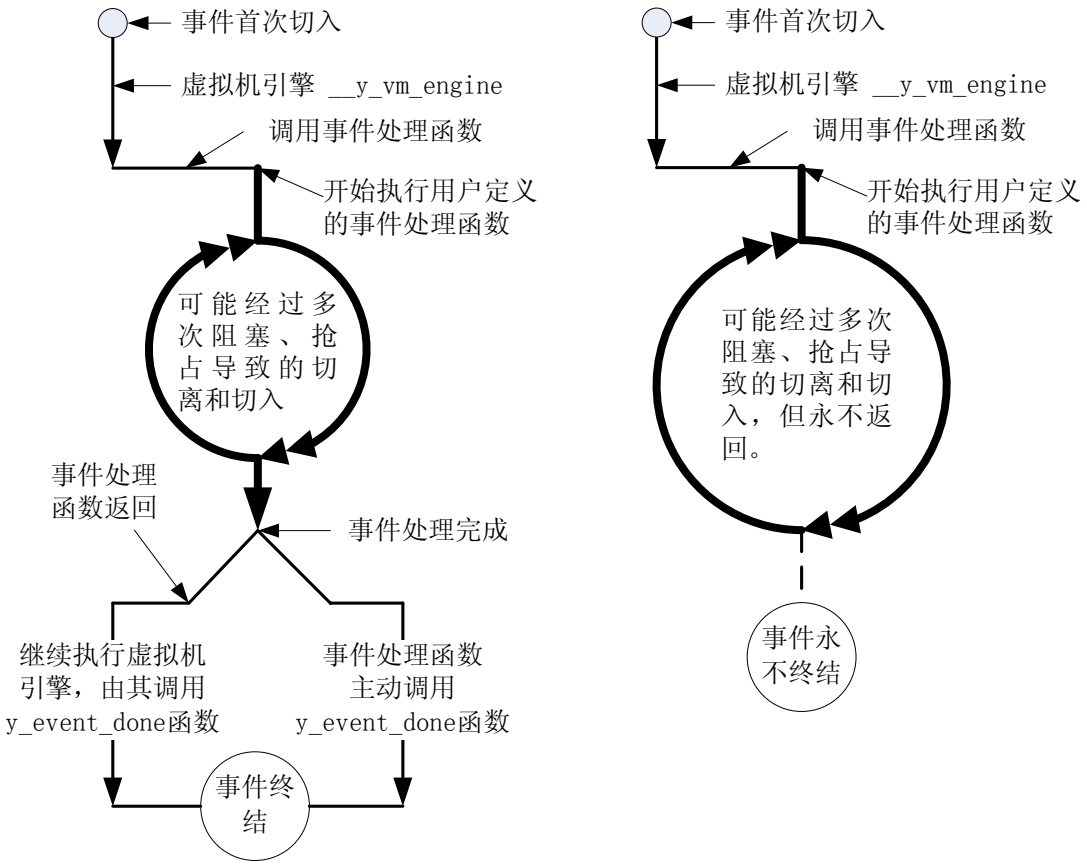


图 4-7 事件处理过程

4.3.11.1 弹出事件

在djyos系统下，应用程序完成一个完整的事件处理周期从弹出事件开始，函数的流程如 图 4-8 所示，图中和代码中对mark型事件有关的部分在第 4.3.13 节有更详细的说明。

事件被弹出后，除mark型事件可能特殊些外（mark型事件参见第 4.3.13 节），新事件将立即进入就绪队列，如果优先级高于正在处理的事件，将立即处理。

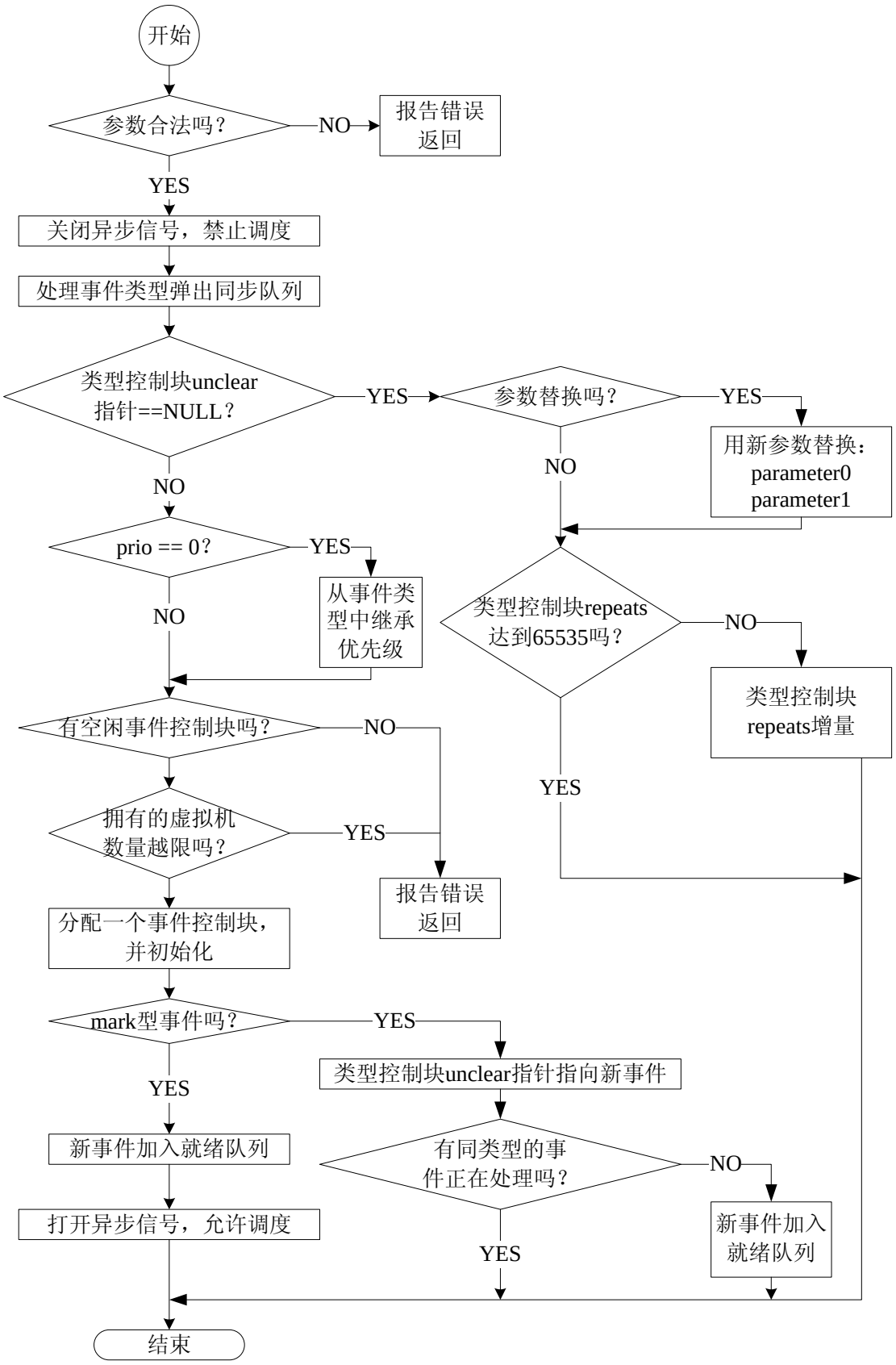


图 4-8 弹出事件流程

参数检查出错的条件有：

1. 事件类型未登记，这一条件检查除容错外，有更积极的意义。在积木化（或称组件

化）的开发中， 某组件需用到的事件类型由自己登记，其他组件如果有需要本组件处理的事件，直接弹出事件就可以了。如果要裁掉该组件，只要把该组件直接删去，自然该组件用到的事件类型就不会被登记了，而与该组件相关的其他组件可以完全不修改，只不过弹出被裁掉的组件相关的事件变成空操作罢了。例如某软件有两个版本，一个有 LCD 界面，另一个没有，则两个版本除界面组件外，其他组件完全相同，不但源代码相同，连二进制的可执行文件都可以做到相同。只要把界面模块单独编译成一个可执行文件，其余部分独立编译成另一个可执行文件，两个文件同时 copy 进去就成了有界面版本，删掉其中一个就成了无界面版本，极大地方便了产品维护和企业软件产品的版本管理。

2. 优先级小于系统服务优先级（cn_prio_sys_service=250），djyos 只允许用户事件的优先级在 1~249 之间。

事件类型弹出同步的处理方法参见第 第 5 章。

弹出事件函数见 代码 4-11。函数的注释应该比较明确，结合 图 4-8 应该很容易阅读，就不占用篇幅详细说明了。

代码 4-11 弹出事件

```
uint16_t y_event_pop(   uint16_t evtt_id,
                        uint32_t parameter0,
                        uint32_t parameter1,
                        ufast_t prio)
{
    struct  event_script *pl_ecb;
    struct  event_type *pl_evtt;
    uint16_t result;
    ufast_t my_prio;
    pl_evtt =&tg_evtt_table[evtt_id];
    if((pl_evtt->property.registered == 0)              //类型未登记
                || (prio >= cn_prio_sys_service))       //优先级非法
    {
        y_error_login(en_knl_etcb_error,NULL);
        return cn_invalid_evtt;
    }
```

检查参数

```
    int_save_asyn_signal(); //关异步信号(关调度)
    while(pl_evtt->pop_sync != NULL)
    {
        pl_ecb = pl_evtt->pop_sync;
        if(pl_ecb->sync_counter == 1)
        {
            pl_ecb->last_status.all = pl_ecb->event_status.all;
            pl_ecb->event_status.bit.evtt_pop_sync = 0;
            if(pl_ecb->event_status.bit.wait_overtime)  //指定的超时未到
            {
                __y_resume_delay(pl_ecb);       //从闹钟队列中移除事件
                pl_ecb->event_status.bit.wait_overtime = 0;
```

```
            }
            __y_event_ready(pl_ecb);
            if(pl_ecb->multi_next == pl_ecb->multi_previous)
            {//队列中只有一个事件
                pl_evtt->pop_sync = NULL;
            }else  //不是最后一个事件，把它从同步队列中取出
            {
                pl_evtt->pop_sync = pl_ecb->multi_next;
                pl_ecb->multi_next->multi_previous = pl_ecb->multi_next;
                pl_ecb->multi_previous->multi_next = pl_ecb->multi_previous;
            }
            pl_ecb->multi_next = NULL;
            pl_ecb->multi_previous = NULL;
        }else
        {
            pl_ecb->sync_counter--;
        }
    }
```

处理弹出同步，方法见第 5.3 节

```
//mark 型事件在队列中可能的组合为:
//  已 clear                    repeats        处理方式
//       未 clear  mark_unclear       in_use
//1  有     无          NULL   ==0     1     新事件由 mark_unclear 指针指向
//2  有     有         !NULL   !=0     1     repeats++,考虑参数替换
//3  无     有         !NULL   !=0     1     repeats++,考虑参数替换
//4  无     无          NULL   ==0     0     新事件加入就绪队列
    if(pl_evtt->mark_unclear != NULL)
    {// mark 型事件的第 2 和第 3 种情况
        pl_ecb = pl_evtt->mark_unclear;          //取事件控制块
        if(pl_evtt->property.overlay == 1)  //参数替换型,替换参数
        {
            pl_ecb->parameter0 = parameter0;
            pl_ecb->parameter1 = parameter1;
        }
        if(pl_evtt->repeats != cn_limit_uint16)
            pl_evtt->repeats++;          //事件发生次数增量
        result = pl_ecb->event_id;
        goto end_pop;                                        //注意 goto
    }
```

处理mark型有未应答事件的情况，参见第 4.3.13 节。

```
    if(prio != 0)
        my_prio = prio; //设置事件优先级,
    else
        my_prio = pl_evtt->default_prio;        //从事件类型中继承优先级
```

取新事件的优先级，参见第 4.3.8 节。

```
    //以下处理非 mark 型事件或者 mark 型事件的第 1、4 种情况
    if(pg_event_free==NULL)     //没有空闲的事件控制块
    {
        y_error_login(en_knl_ecb_over,NULL);
        result = cn_invalid_id;
    }else         //有空闲事件控制块，看能否创建虚拟机
    if((pl_evtt->property.mark == false)
                    && (pl_evtt->repeats >= pl_evtt->vpus_limit))
    {//非 mark 型事件，需要创建线程虚拟机，但该类型事件拥有的虚拟机数量已经超限
        y_error_login(en_knl_vpu_over,NULL);
        result = cn_invalid_id;
    }else                       //有空闲事件控制块且允许创建虚拟机
    {

        pl_ecb = pg_event_free;          //从空闲链表中提取一个事件控制块
        if(pg_event_free->next == NULL)
            pg_event_free=NULL;          //这是最后一块事件控制块了
        else                             //不是最后一个事件控制块
        {
            pg_event_free=pg_event_free->next;  //空闲事件控制块数量减 1
        }
        if(pl_evtt->repeats != cn_limit_uint16)
            pl_evtt->repeats++;          //repeats 增量
        //设置新事件的参数
        pl_ecb->next = NULL;
        pl_ecb->previous = NULL;
        pl_ecb->multi_next = NULL;
        pl_ecb->multi_previous = NULL;
        pl_ecb->evtt_id=evtt_id;
        pl_ecb->parameter0=parameter0;
        pl_ecb->parameter1=parameter1;
        pl_ecb->start_time=y_get_time();
        pl_ecb->consumed_time = 0;
        pl_ecb->sync = NULL;
        pl_ecb->delay_start = 0;
        pl_ecb->delay_end = 0;
        pl_ecb->vm = NULL;
        pl_ecb->held_device = NULL;
        pl_ecb->held_memory = NULL;
        pl_ecb->last_status.all = 0;
        pl_ecb->event_status.all = 0;
        pl_ecb->prio = my_prio;
        if(pl_evtt->property.mark == true)     //这是 mark 型事件
```

```
        {
            pl_evtt->mark_unclear = pl_ecb;
            if(pl_evtt->property.in_use == 1)    //第 1 种情况
            {
            }else                                //第 4 种情况
            {
                __y_event_ready(pl_ecb);
            }
        }else                                    //非 mark 型事件
            __y_event_ready(pl_ecb);
        pl_evtt->property.in_use = 1;
        pl_evtt->pop_sum++;
        result = pl_ecb->event_id;
    }

end_pop:
    int_restore_asyn_signal();  //恢复中断状态
    return result;
}
```

弹出的是非 mark 型事件或者 mark 型事件没有未 clear 的事件的情况。

4.3.11.2 事件开始处理

当事件弹出后，就在就绪队列中等待，调度器第一次决定把该事件切入CPU后，标志着该事件就要开始处理了。处理事件必须依靠线程，在弹出事件时，并没有为其分配线程，因而事件首次切入之前，可能并不拥有线程虚拟机资源。因此，调度器首先需要检查该事件是否拥有线程，如没有则检查该事件类型是否有空闲线程，有则直接分配，否则创建一个线程，这个过程请参见 图 4-9。

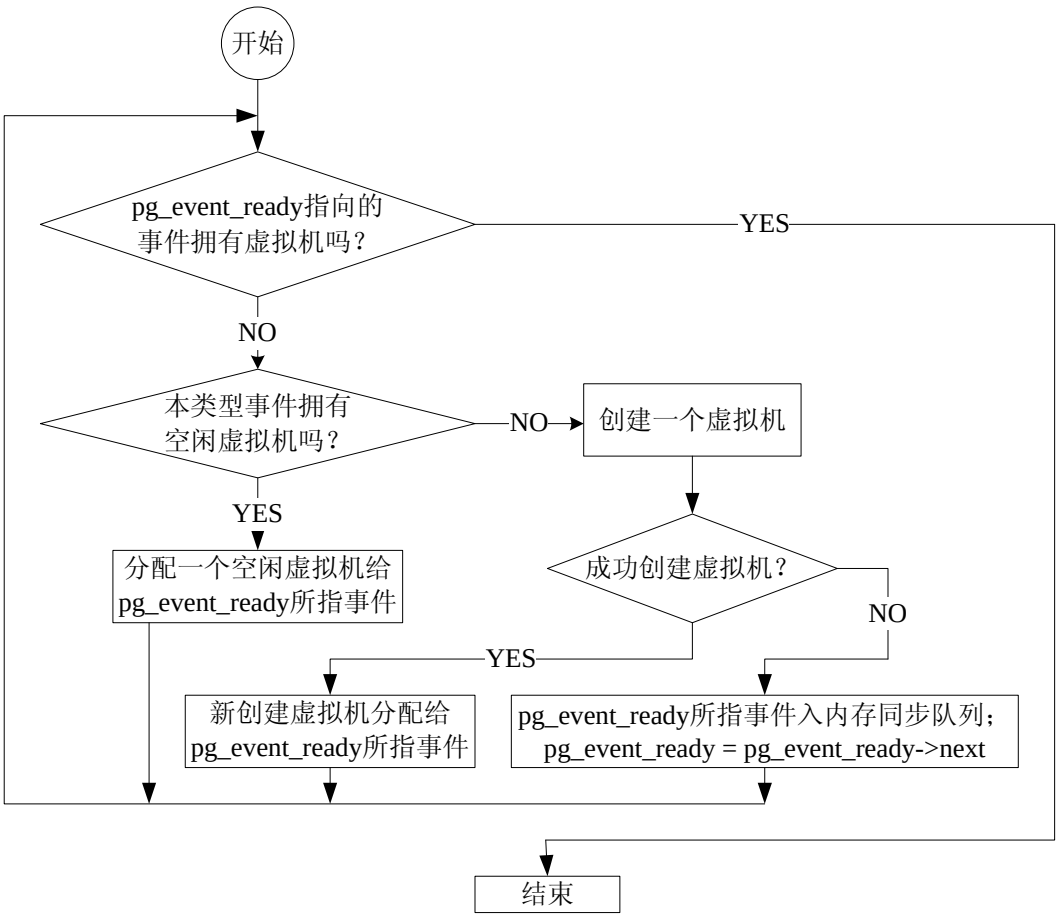


图 4-9 获取虚拟机

获取虚拟机的工作是在__y_select_event_to_run函数中完成的，如 代码 4-12 所示，无论是何种原因导致的事件切换，都必须在执行上下文切换之前调用本函数。有心的读者可能会担心，在 图 4-9 中，如果就绪队列中所有事件都不拥有虚拟机且不能成功创建虚拟机，而就绪队列是一个循环链表，会不会陷入死循环呢？不会的，因为整个系统中最低优先级的系统服务事件总是处于就绪状态，而且总是拥有虚拟机，所以循环到达队列尾部时循环退出条件必然成立。

代码 4-12 __y_select_event_to_run 函数

```
void __y_select_event_to_run(void)
{
    struct  event_script *event;
    struct  event_type *pl_evtt;  //被操作的事件的类型指针

    while(pg_event_ready->vm == NULL)                          //1
    {
        pl_evtt =& tg_evtt_table[pg_event_ready->evtt_id];
        if(pl_evtt->my_free_vm != NULL)
        {//有空闲的虚拟机资源,直接使用
            pg_event_ready->vm = pl_evtt->my_free_vm;          //2
        }else       //没有空闲的虚拟机资源,试图创建新虚拟机
        {
            pg_event_ready->vm = __create_thread(pl_evtt);
```

```
            if(pg_event_ready->vm == NULL)                          //3
            {
                event = pg_event_ready;
                __y_cut_event(event);
                __wait_memory(event);
            }else
            {
                pl_evtt->vpus++;
                //取此前被重复弹出事件的次数
                pg_event_ready->repeats = pl_evtt->repeats;
            }
        }
    }
}
```

注解：

1. 首次执行循环条件就成立的可能：一是本事件不是第一次切入；二是mark型事件接受了先前弹出的事件转交的虚拟机，参见第 4.3.6 节。
2. 空闲虚拟机资源的来源有两个可能性，一是在登记事件类型时创建的，参见第 4.3.5 节；另一个来源是暂存的，参见第 4.3.11.3 节。
3. 不能创建线程的原因只有一条：内存不足，需要把事件从就绪队列中取出，挂到内存同步队列中去，同步操作请参见第 第 5 章。

创建虚拟机时，复位函数__asm_reset_thread（参见第 4.3.6 节）把__vm_engine函数的入口地址设置到虚拟机的栈中程序指针的位置，使得该虚拟机首次被切入的时候，将立即调用__vm_engine函数。获取虚拟机后，便立即切入该虚拟机的上下文，执行虚拟机引擎函数__vm_engine。任何线程都是从__vm_engine函数开始的，诚如其名，这是一个发动虚拟机的引擎，用户定义的事件处理函数（也就是线程入口函数）将由__vm_engine调用，该函数就是当初登记事件类型时指定的、用于处理该类型事件的函数。

代码 4-13 虚拟机引擎

```
//----虚拟机引擎-----------------------------------------------------------------
//功能: 启动虚拟机,执行虚拟机入口函数,事件完成后执行清理工作
//参数: thread_routine  处理该事件类型的函数指针.
//返回: 无
//-----------------------------------------------------------------------------
void __vm_engine(void (*thread_routine)(struct event_script *my_event))
{
    int_restore_asyn_signal();                                  //1
    thread_routine(pg_event_running);                           // 2
    y_event_done();                                             //3
}
```

1. 开启调度使能（也即异步信号使能），在关闭调度使能期间是不可能发生事件切换的，因此，任何事件，在引擎刚启动的时候，调度使能必定是开启的。而执行__vm_engine 函数前程序必定执行了调度函数，调度函数需要访问操作系统的核心

数据，要关闭调度使能，调度使能只能在引擎中重开。

2. 这就是事件处理函数了，用户定义的，操作系统只管调用。
3. 本句实际上是容错用的，djyos 提倡用户在事件处理完成后，直接在事件处理函数中调用 y_event_done 函数，显式地告诉操作系统：我的使命结束了。如果用户没有这样做而是让事件处理函数直接返回，将在这里调用。

4.3.11.3 事件处理完成

y_event_done 函数被调用，标志着事件处理已经完成，应该从就绪队列中删除该事件，并回收资源，y_event_done 函数完成的工作共有：

1. 把事件同步队列中的所有事件加入就绪队列，该队列中的事件都是被阻塞以等待当前事件完成的事件，参见第 5.2 节。
2. 回收动态内存，事件处理中通过动态分配从系统堆中分配的、未释放的内存，djyos 的内存管理系统提供两个内存分配函数，分别用于分配局部内存（m_malloc）和全局内存（m_malloc_gbl），局部内存要在线程内及时释放，而全局内存则可以一直使用到使命完成，比如某事件处理时创建了一个设备，并为该设备分配了动态内存，如果该设备在事件处理结束后继续有效，则应该分配全局内存，直到该设备使命完成再释放内存。对于局部内存，djyos 系统提倡用户按照“申请——释放”成对出现的方式使用动态内存，这样可以减少内存碎片，如果用户严格按照这种方式编程，事件处理完成的时候，是没有未释放内存的。djyos 从可靠性出发，在 y_event_done 函数中检查用户是否有未释放的内存，如果有未释放的，则报告错误并强制释放。
3. 关闭打开的泛设备，泛设备是一个公共资源，使用完后应该及时关闭，与内存一样，在 y_event_done 函数中会检查事件是否有未关闭的设备，有则强制关闭之。djyos 要求事件处理完毕时必须关闭自己打开的设备，否则将强制关闭。
4. 释放事件控制块，事件完成，也就标志着该事件控制块的使命已经完成，把它从就绪队列中删除，放到空闲队列中去。
5. 检查事件类型是否应该删除，参考第 4.3.3.3 节。
6. 销毁、转交或暂存线程虚拟机，见代码中的说明。
7. 执行事件切换，把就绪队列头部的事件切入。

代码 4-14 y_event_done 函数

```
#define cn_deliver_to     0     //虚拟机已经转交
#define cn_keep           1     //虚拟机保留不删除
#define cn_delete         2     //虚拟机应该被删除
void y_event_done(void)
{
    struct   event_script *pl_ecb;
    struct   event_type     *pl_evtt;
    struct   event_script *pl_ecb_temp;
    struct   thread_vm *next_vm;
    ucpu_t   vm_final = cn_delete;

    __int_reset_asyn_signal();
```

关闭调度

以容错计，这里不象其他需要操作事件队列的地方一样用__int_save_asyn_signal 函数关闭调度，如果事件处理函数成对地调用__int_save_asyn_signal 函数和__int_restore_asyn_signal 函数，那么，调用 y_event_done 函数之前，调度应该处于允许状态，但是，如果应用程序有 bug，没有成对调用，使得调用 y_event_done 前调度处于禁止状态，就会出现不可挽回的错误。在其他可能引起切换的地方，由于事件还没有终结，禁止调度的后果只不过是切换不成功，仍然返回原事件，而 y_event_done 函数不同，由于正在处理的事件已经终结，必须要切换到新的事件，不能切换的后果不可预料。

调用__int_reset_asyn_signal 函数的结果，就象事件处理函数已经成对调用__int_save_asyn_signal 函数和__int_restore_asyn_signal 函数后再调用__int_save_asyn_signal 一样，修正了事件处理函数没有成对调用调度使能函数的错误。

```
pl_ecb = pg_event_running->sync;        //取同步队列头
while(pl_ecb != NULL)
{
    pl_ecb->last_status.all = pl_ecb->event_status.all;     //保存当前状态
    pl_ecb->event_status.bit.event_sync = 0;        //取消"同步中"状态
    if(pl_ecb->event_status.bit.wait_overtime)     //是否在超时队列中
    {
        __y_resume_delay(pl_ecb);       //结束超时等待
        pl_ecb->event_status.bit.wait_overtime = 0;   //取消"超时等待中"状态
    }
    pl_ecb_temp = pl_ecb;
    if(pl_ecb->multi_next == pg_event_running->sync)   //是最后一个事件
    {
        pg_event_running->sync = NULL;   //置空事件同步队列
        pl_ecb = NULL;
    }else
    {
        pl_ecb = pl_ecb->multi_next;          //取队列中下一个事件
    }
    __y_event_ready(pl_ecb_temp);                 //把事件加入到就绪队列中
}
```

处理事件同步，参见第 5.2 节

```
dev_cleanup(pg_event_running->held_device); //检查并关闭未关闭的设备
if(pg_event_running->held_memory != NULL)
{
    m_cleanup();
}
```

回收内存和设备资源，参见第 9.4.4.5 节和 7.3.3 节

```
__y_cut_ready_event(pg_event_running);
pg_event_running->previous
                    = (struct    event_script*)&pg_event_free;//表示本控制块空闲
pg_event_running->next = pg_event_free;        //pg_event_free 是单向非循环队列
pg_event_free = pg_event_running;
```

处理就绪事件队列

把事件从就绪队列取出，放进空闲事件控制块队列，参见第 4.3.3.5 节和第 4.3.3.6 节。

```
pl_evtt =&tg_evtt_table[pg_event_running->evtt_id];
pl_evtt->done_sum++;
while(pl_evtt->done_sync!= NULL)
{//链表中的事件都是要么没有指定超时，要么时限未到，或者时限虽到但还未开始
 //执行的.其他情况不会在此链表中留下痕迹，type_after_sync 位也已经清除
    pl_ecb = pl_evtt->done_sync;
    //同步条件达成的条件: 1、同步计数器为 1。
    //2、同步计数器为 0 且本类型最后一条事件已经处理完
    if((pl_ecb->sync_counter == 1)
            ||((pl_ecb->sync_counter == 0) &&(pl_evtt->repeats == 0)))
    {
        pl_ecb->last_status.all = pl_ecb->event_status.all;
        pl_ecb->event_status.bit.evtt_pop_sync = 0;
        if(pl_ecb->event_status.bit.wait_overtime)   //指定的超时未到
        {
            __y_resume_delay(pl_ecb);           //从闹钟队列中移除事件
            pl_ecb->event_status.bit.wait_overtime = 0;
        }
        __y_event_ready(pl_ecb);
        if(pl_ecb->multi_next == pl_ecb->multi_previous)
        {//是最后一个事件
            pl_evtt->done_sync = NULL;
        }else
        {
            pl_evtt->done_sync = pl_ecb->multi_next;
            pl_ecb->multi_next->multi_previous = pl_ecb->multi_next;
            pl_ecb->multi_previous->multi_next = pl_ecb->multi_previous;
        }
        pl_ecb->multi_next = NULL;
        pl_ecb->multi_previous = NULL;
    }else
    {
        pl_ecb->sync_counter--;
    }
}
```

处理事件类型完成同步，参见第 5.4 节

```
if(pl_evtt->property.mark == true)          //mark 型事件
{
    pl_ecb = pl_evtt->mark_unclear;
    if(pl_ecb == pg_event_running)   //说明用户没有清除 mark 标记       //1
    {
        pl_evtt->mark_unclear = NULL;
```

```
            pl_evtt->property.in_use = 0;
            pl_evtt->repeats = 0;
            y_error_login(en_knl_clear_mark,"未清除 mark 标记");
            if((cn_run_mode!=cn_mode_mp)||(pl_evtt->default_prio<0x80))
            {
                vm_final = cn_keep;
            }else
            {
                vm_final = cn_delete;
            }
        }else if(pl_ecb != NULL)       //有新事件发生,虚拟机可以转交
        {
            pl_ecb->vm = pg_event_running->vm;        //虚拟机转交（分配）给新事件
            vm_final = cn_deliver_to;
            __y_event_ready(pl_ecb);                  //新事件加入就绪链表
        }else           //没有新事件产生，看是否应该作为永久性资源予以保留
        {
            pl_evtt->property.in_use = 0;
            if((cn_run_mode!=cn_mode_mp)||(pl_evtt->default_prio<0x80))
            {
                vm_final = cn_keep;
            }else
            {
                vm_final = cn_delete;
            }
        }
    }else     //非 mark 型事件
    {
        if(pl_evtt->repeats > pl_evtt->vpus)       //有未得到虚拟机的事件，保留之
        {
            pl_evtt->repeats--;
            vm_final = cn_keep;
        }else     //没有未得到虚拟机的事件，再看是否常驻保留
        {
            if(pl_evtt->repeats == 1)
            {
                pl_evtt->repeats = 0;
                pl_evtt->property.in_use = 0;
                if((cn_run_mode!=cn_mode_mp)||(pl_evtt->default_prio<0x80))
              {
                    vm_final = cn_keep;
                }else
                {
                    vm_final = cn_delete;
```

```
                }
            }else
            {
                pl_evtt->repeats--;
                vm_final = cn_delete;
            }
        }
    }
```

决定虚拟机如何处理

1. 如果系统运行在si或dlsp模式，或者事件类型的默认优先级小于 128，无论mark型事件还是普通事件，最后一个虚拟机必须保留以做永久性资源，参见第 4.3.5 节。
2. mark 型事件本次运行被清除后，事件处理完毕前，又有新事件弹出，新事件将挂在 mark_unclear 指针下，而虚拟机也将转交给这个事件。
3. 非 mark 型事件，如果正在就绪队列中等待、尚未获得虚拟机的事件的数量比事件类型控制块的 my_free_vm 下挂的空闲虚拟机数量多，则虚拟机将插入到 my_free_vm 队列中。为什么不象 mark 型事件一样直接转交给就绪队列中的同类型事件呢？因为：事件队列中可能有多条同类型事件在等候虚拟机，或者当前事件完成后、下一次切入同类型事件前还可能会继续弹出同类型的新事件，我们希望虚拟机转交给这些等候虚拟机的事件中最先切入的事件，而当前事件并不知道谁会先得到调度器的眷顾。

注释 1：mark型事件要求事件处理函数调用y_clear_mark函数以清除标记，mark型事件切入时，pg_event_running指针与mark_unclear成员均指向正在处理的事件，如果用户清除标记，mark_unclear成员将置空，后续重复弹出的事件将挂在mark_unclear指针下，因此，如果mark_unclear等于pg_event_running则说明事件处理函数有bug，没有按要求调用y_clear_mark，需要善后，否则，若非空则有新事件重复弹出，需转交虚拟机，若空则没有新事件重复弹出。mark型事件在第 4.3.13 节有更详细的说明。

```
    __y_select_event_to_run();                          //获取待切入的事件
    if(vm_final == cn_delete)                           //删除虚拟机
    {
        m_free((void*)pg_event_running->vm);        //删除虚拟机
        pl_evtt->vpus--;
        pg_event_running = pg_event_ready;
        u32g_running_start = y_get_time();
        __asm_turnto_context(pg_event_running->vm);
    }else if(vm_final == cn_keep)       //保留虚拟机
    {
        pg_event_running->vm->next = pl_evtt->my_free_vm;
        pl_evtt->my_free_vm = pg_event_running->vm;
        pl_ecb = pg_event_running;
        pg_event_running = pg_event_ready;
        u32g_running_start = y_get_time();
        __asm_reset_switch(pl_evtt->thread_routine,
                           pg_event_running->vm,pl_ecb->vm);        //1
    }else                       //虚拟机已经转交给另一条事件
```

```
    {
        pl_ecb = pg_event_running;
        pg_event_running = pg_event_ready;
        u32g_running_start = y_get_time();
        __asm_reset_switch(pl_evtt->thread_routine,
                           pg_event_running->vm,pl_ecb->vm);
    }
}
```

决定线程去留和切换到新事件

注释 1、此函数复位当前虚拟机然后切入到新虚拟机，注意不能使用下列语句序列：

__asm_reset_thread(……);

__asm_turnto_context(……);

因为 y_event_done 在当前线程的上下文中执行，而第一个函数的目的是复位当前线程虚拟机，重置了当前线程的栈和栈指针，C 语言代码继续运行的环境已经破坏，调用第二个函数将出错。

4.3.12 允许和禁止调度

当程序要访问跟调度有关的核心数据结构，或者执行一些不容打断的操作时，需要禁止调度，djyos系统中，异步信号（异步信号是由中断触发的，参见 第 6 章）的优先级虽然比事件高，但只是量的不同而不是质的不同，**禁止/允许调度实际上等同于禁止/允许异步信号**。下列代码能够确保“临界代码块”不被异步信号打断和被高优先级事件抢占，用于保护访问跟调度有关的核心数据结构，比如各种事件队列。

```
int_save_asyn_signal（）;
临界代码块;
int_restore_asyn_signal（）;
```

但上述“临界代码块”还可能被实时中断（参见 第 6 章）打断，如果保护的目的是确保上述“临界代码块”连续执行，比如要执行有严格时序的硬件操作，就必须禁止实时中断，int_save_trunk( )函数禁止包括异步信号和实时中断在内的所有中断，自然也就禁止了高优先级的事件了。下列代码能够确保“临界代码块”在绝对“安静”的环境下运行。

```
int_save_trunk（）;
临界代码块;
int_restore_trunk（）;
```

4.3.13 mark 型事件

我们以简易串口接收程序来说明一下 mark 型事件的调度特点。

1、 注册一个 mark 型的“串口接收”事件类型。

2、 当串口收到数据时，将引发硬件中断，该中断被设置为“异步信号”，在异步信号 ISR 程序中，把数据从物理接收寄存器中读至接收 fifo 缓冲区，然后调用 y_event_pop 函数弹出一个“串口接收”类型的事件。

3、 “串口接收”事件处理程序把接收 fifo 中的数据全部取走。

4、 按约定的协议处理数据。

5、 处理完成后，调用 y_event_done 函数，告诉操作系统，事件处理完毕。

仔细分析一下 2~3 步之间，如果“串口接收”事件弹出后，由于事件优先级不够或者其他原因没有及时处理，后续接收到的数据会不断地压入接收 fifo 缓冲区，不断地弹出事件。然而，无论事件被弹出多少次，fifo 中被压入多少数据，第 3 步均一次全部取走。所以，在第 3 步之前发生的事件，无论调用 y_event_pop 多少次，实际上都只需要发出一条事件就可以了，这就是 mark 的真谛——它只是标记某种状态，在本例中，就是标记“串口接收 fifo 中有数据等待处理”这样一种状态。第 3 步以后，如再次接收到数据，新数据将不能在本次事件处理中得到处理，事件处理程序当在适当的时候调用 y_clear_mark 函数清除 mark 状态，使后续事件可以正常发出。等等！事情并没有这么简单，在第 3~5 步之间，如果再次发生该类型事件，新弹出的事件的优先级与正在处理的事件优先级是相同的，如果该事件直接进入就绪队列的话，因轮转调度，或者新事件弹出时指定了较高的优先级，或者原事件因故阻塞，这些原因都可能在原事件处理完之前新事件又开始处理。这不符合我们设立 mark 事件的初衷，我们希望，mark 新型事件应该在旧事件完成之后，再开始新事件处理。为解决这个问题，在事件类型控制块中特设了一个 struct event_script *mark_unclear 指针成员，用做临时驿站，在 mark 型事件已经被 clear 尚未完成的情况下，暂时保存新弹出的事件，在事件完成后再把该事件加入就绪队列。该指针指向该类型的尚未被 clear 的事件，如不是 mark 事件类型，该指针无效。mark 型事件必须在处理过程中由用户程序调用 y_clear_mark 函数进行 clear mark，如果用户没有这样做，将会在 y_event_done 函数中被检测出来，并且强制执行 clear mark 并且报告错误。

综上所述，mark型事件的弹出过程如 图 4-10 所示，这是从 图 4-8 中单独把mark型事件弹出过程抽取出来的，执行过程如 图 4-11 所示。

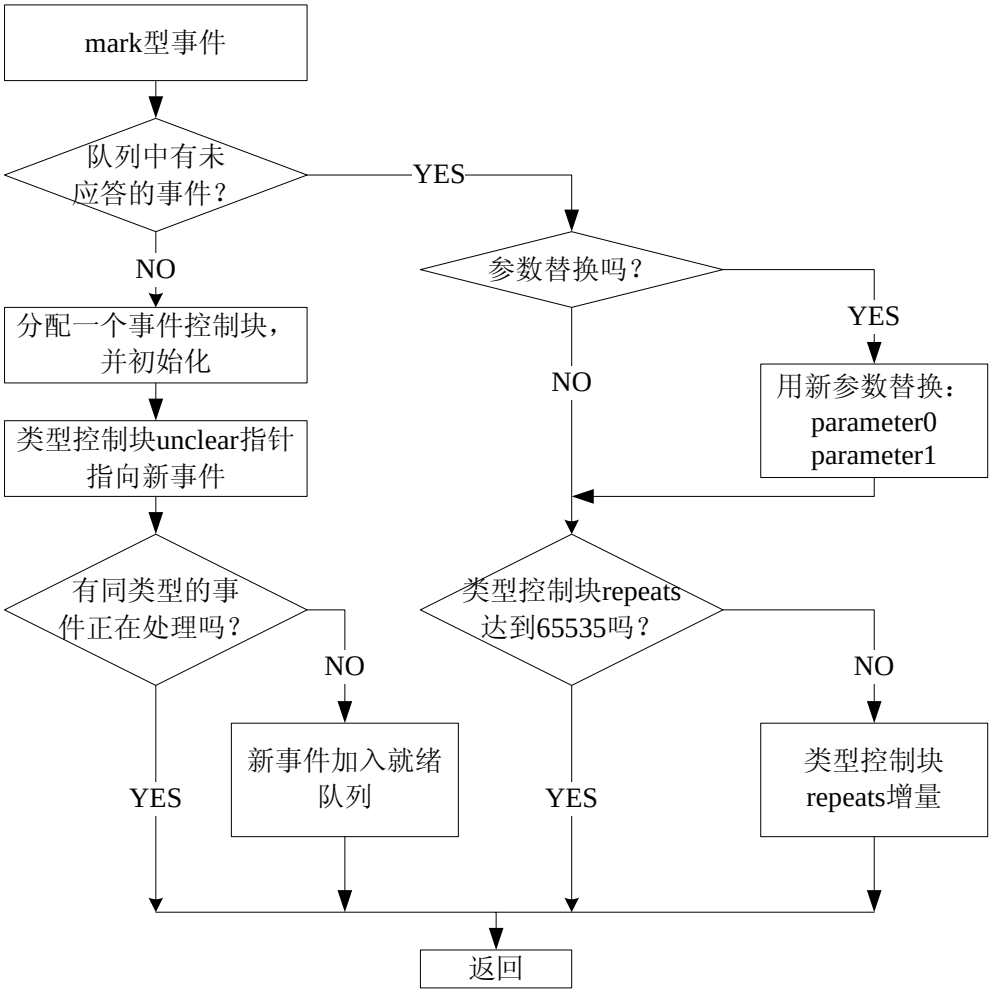


图 4-10 从图 4-8 中抽象出来的mark型事件弹出过程

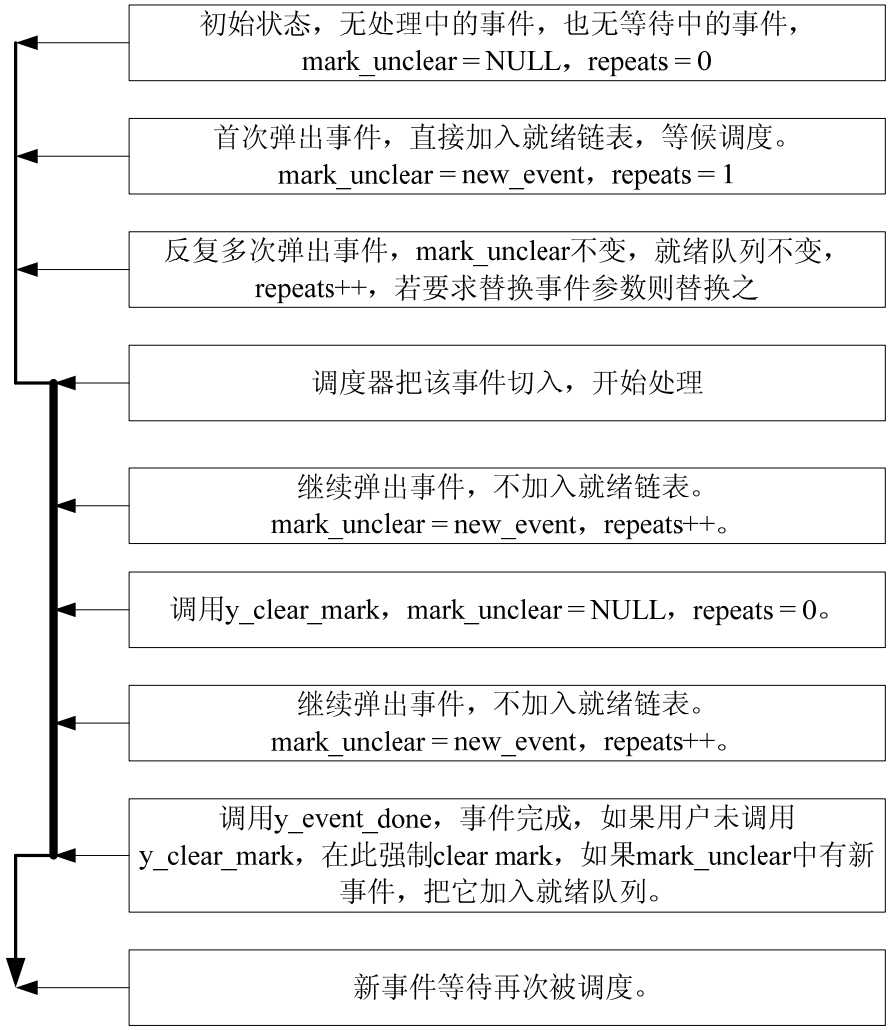


图 4-11 mark 型事件弹出与处理过程

4.3.14 事件与事件类型小结

前面讲了事件类型登记与删除、事件的生命周期以及虚拟机的创建与删除，这些内容分布在各小节中，在这里把操作系统各运行模式下这三要素的关系总结如 表格 4-2 所示。其中运行模式将在第 4.5 节中讲述。

表格 4-2 事件与虚拟机

	mark	优先级	si 和 dlsp 模式	mp 模式
注册事件类型	0	<128	创建一个线程虚拟机	同左
	0	≥128	创建一个线程虚拟机	无动作
	1	<128	创建一个线程虚拟机	同左
	1	≥128	创建一个线程虚拟机	无动作
首次弹出事件			加入就绪队列	同左
首次切入			事件类型有空闲虚拟机则直接分配，无则创建虚拟机	同左

重复弹出事件	0		如果事件队列中的事件总量少于vpus_limit，加入就绪队列	同左
	1		若mark标记未清除：事件类型中的repeats增量 若mark标记已清除：新事件暂存于mark_unclear指针下	同左
事件处理完成	0	<128	最后一个虚拟机：不删除。 如果队列中有同类型的、尚不拥有虚拟机的事件少于空闲虚拟机数量：不删除。 其他情况：删除虚拟机	同左
	0	≥128	最后一个虚拟机：不删除。 如果队列中同类型的、尚不拥有虚拟机的事件数量少于空闲虚拟机数量：不删除。 其他情况：删除虚拟机	最后一个虚拟机也可能被删除。 其他同左
	1	<128	mark_unclear==NULL：虚拟机放进事件类型的空闲队列。 mark_unclear!=NULL：虚拟机直接转交给新事件	同左
	1	≥128	mark_unclear==NULL：虚拟机放进事件类型的空闲队列。 mark_unclear!=NULL：虚拟机直接转交给新事件	同左

4.4 线程的栈

C语言函数调用时，需要在栈中保存局部变量和参数，系统调用和api函数（下称系统服务）也不例外。当应用程序调用系统服务时，所需的栈在哪里提供呢？一种方法是在内核空间创建栈，我们把这种栈叫做“内核代理栈”，意思是内核代理线程执行系统服务时所需要的栈。另一种方法是直接使用线程的栈，为了确保线程调用系统服务时栈不溢出，需要为线程提供比线程本身所需更多的内存做栈。djyos没有采用“内核代理栈”的方法，即使在mp模式下，系统服务所需要的栈也是由线程虚拟机提供。应用程序的作者需要负责告诉操作系统，该线程需要多大的栈空间，测算栈需求的方法参见第 2.8.2 节，在计算时无需考虑系统服务所需要的栈。操作系统在创建线程虚拟机时，在用户测算的栈需求的基础上，不管该线程是否调用了系统服务，都自动追加系统服务所需的栈，以防调用系统服务时栈溢出。在port_kernel.h文件中，有一个由移植者设置的常量cn_kernel_stack，表明该特定版本中栈需求最大的那个系统服务所需要的栈的数量。如下语句

```
u16g_evtt_flash_led = y_evtt_regist(false,false,100,10,flash_led,1024,NULL);
```

将登记一个事件类型，类型号保存在u16g_evtt_flash_led变量中，该类型的事件由flash_led函数处理，该函数运行需要的栈不超过 1Kbytes（不计算系统服务所需的栈）。当有该类型事件发生，并且该事件得到服务时，操作系统将创建一个栈尺寸为（cn_kernel_stack + 1024）字节的线程，该线程的栈将如 图 4-12 所示。

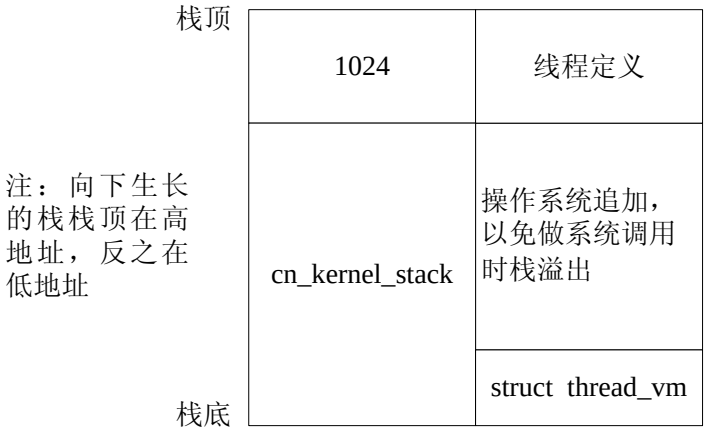


图 4-12 分配线程栈

系统服务合并使用线程的栈而不是在内核空间开辟“内核代理栈”，有许多好处，首先是调用系统服务时上下文切换更加快速，如果被调用的服务不使用特权指令，根本无需上下文切换，直接调用函数就可以了，快速而且高效；如果被调用的服务使用了特权指令，上下文切换也只是切换处理器状态，而无需切换栈。其次是这样的栈更加安全，不容易溢出，程序员测算栈需求时，一般都要加一个保险系数，且该线程未必会调用栈需求最大的系统服务，再附上 cn_kernel_stack “线程代理栈”，相当于有了双重保险。就算粗心的程序员测算栈需求时出错，实际需要 1.1Kbytes 而错误地测算为 1Kbytes，如果不是和系统服务合并栈，那么只要线程执行时栈越过 1Kbytes 的边界，就有可能破坏其他未知数据，造成不可预料的后果。如果与系统服务合并栈，则只要在实际运行中线程所消耗的栈和该线程实际调用的系统服务所消耗的栈不超过 1Kbytes + cn_kernel_stack，是完全安全的。

4.5 时钟嘀嗒

操作系统离不开时钟嘀嗒，它就像人的脉搏一样，永不停息。djyos中，时钟嘀嗒是通过一个定时器中断实现的，该中断被设置为异步信号（参见 第 6 章 中断）。该异步信号在启动多线程管理（参见第 4.7.2.5 节）时初始化并启动，初始化函数__y_init_tick是一个与体系结构相关的移植关键函数，详见“第 15 章 djyos移植”，该函数完成的工作有：设置定时器硬件工作方式、设定tick时间、把定时器中断线设为异步信号、连接__y_isr_tick为异步信号ISR、启动定时器开始计时、允许该定时器中断线。时钟嘀嗒是轮转调度和系统定时的最小时间粒度。时间粒度越小，意味着操作系统响应线程请求的实时性越高，但是要付出由于频繁中断和调用__y_isr_tick函数以及频繁上下文切换带来的高损耗的代价，因此，决定调度时间粒度应该是系统设计阶段需要仔细考量的问题。

操作系统提供两个函数供用户获取以时钟嘀嗒表示的当前时间：

uint32_t y_get_time(void);用于取得从系统启动始计算的时钟嘀嗒数，获得的是 32 位变量，若溢出则回绕到 0，如果时钟嘀嗒是 1mS，则每 49.7 天回绕到 0。

uint32_t y_get_fine_time(void);用于取得从上一次时钟嘀嗒数改变始计算的精密时钟嘀嗒数，也是 32 位的。

在 port_kernel.h 中有几个常量用于声明时钟嘀嗒和精密时钟嘀嗒的参数：

```
#define cn_tick_ms      1       //操作系统内核时钟脉冲长度，以毫秒为单位。
#define cn_tick_hz      1000    //内核时钟频率，单位为 hz。
#define cn_fine_us      1       //操作系统内核精密时钟脉冲长度，以微秒为单位。
```

#define cn_fine_hz　　1000000 //内核精密时钟频率，是 cn_fine_us 的倒数。

用 y_get_time 获得的时钟嘀嗒数是否与__y_isr_tick 函数被调用的次数相同呢？换句话说，是不是每调用一次__y_isr_tick 函数时钟嘀嗒就加 1 呢？不一定，要根据具体硬件。

根据 cn_tick_ms 的值和定时器的输入时钟周期，设定一个定时常数 K，定时器从 K 开始，每个输入时钟周期减 1，减到 0 为一个时钟嘀嗒，发出中断信号，或者反之，定时器从 0 开始计数，达到 K 为一个时钟嘀嗒。根据硬件的功能，时钟嘀嗒的走时方式有 3 种（以减计数的定时器为例）：

1. 手动走时自动重设定时器。

定时器发出中断信号时，自动重新加载 K 并重新运行，这叫自动走时。然而，没有专门的硬件记录时钟嘀嗒，需要设一个静态变量 u32g_os_ticks 记录，在每次调用__y_isr_tick 函数时执行“u32g_os_ticks++”。这种方式下 u32g_os_ticks 值与调用__y_isr_tick 函数的次数是绝对相等的，但并不能确保 u32g_os_ticks 绝对准确，在下列情况下，u32g_os_ticks 将会比实际时钟慢：

a) 连续关调度（也就是关异步信号）的时间超过 1 个 tick，将错过一次或多次__y_isr_tick 调用，相应地 u32g_os_ticks 也变慢。

b) 比定时器中断更高级的异步信号 ISR 连续执行，或者低级异步信号在不开中断嵌套的情况下连续执行，或者实时中断连续执行时间超过 1 个 tick，也将错过一次或多次__y_isr_tick 调用，相应地 u32g_os_ticks 也变慢。

2. 手动走时手动重设定时器。

这种情况与自动走时类似，不同的是，定时器发出中断信号时，常数 K 并不自动重新加载，需要在__y_isr_tick 中手动加载。由于从发出中断信号到__y_isr_tick 函数手动加载 K 之间有延时，且不是很确定，这种方式的工作的时钟嘀嗒是不准确的，需要做补偿。补偿的方法通常是，设法获得时钟中断发生到手动重加载之间的间隔时间 Et，加载 K 的时候减去 Et。

3. 自动走时方式。

与前两种情况时钟嘀嗒有误差不同，这种方式是没有误差的，虽然定时器硬件还是每个 tick 周期产生一个中断信号，但是时钟嘀嗒的增量是自动完成的。也就是说，不管__y_isr_tick 有没有被调用，时钟嘀嗒都自动增量，程序只要读取特定寄存器的值就可以得到准确的时钟嘀嗒值。

不管上述哪种方式下，精密时钟都是依靠硬件来实现的，y_get_fine_time 函数都是读硬件寄存器的方式取得时间值，因此肯定是准确的。

第三种方式获得了准确的时间，但也要付出代价，__y_isr_tick函数的要比前两种情况复杂些。在__y_isr_tick函数中，需要处理闹钟同步队列（参见第 5.1 节），因为前两种情况下，能确保每次检查闹钟同步队列时，u32g_os_ticks只增量 1，用语句：

if(pl_ecb->delay_end = u32g_os_ticks)

肯定可以准确判断闹铃 delay_end 时间是否到，设定闹铃时，闹铃时间肯定在 u32g_os_ticks 的前面，而 u32g_os_ticks 逐一增量的话，则肯定有 delay_end 与 u32g_os_ticks 相等的时刻。

而第三种情况则不知道增量是多少，有可能直接跨过 delay_end，即使用：

if(pl_ecb->delay_end = u32g_os_ticks)

为条件，也会因 32 位变量回绕到 0（如果 tick 为 1mS，每 49.7 天回绕一次）而不能准确判断，__y_isr_tick 需要记住上一次中断时的时钟嘀嗒，再看本次时钟嘀嗒，看两次__y_isr_tick 调用之间有没有跨越 delay_end 才能准确判断闹铃时间是否到。**把 djyos 移植到这样的硬件上时，需要特别注意。**

图 4-13 是__y_isr_tick函数的流程图，该流程并不复杂，代码就不列出了，有兴趣的读者可以参考随书源码。

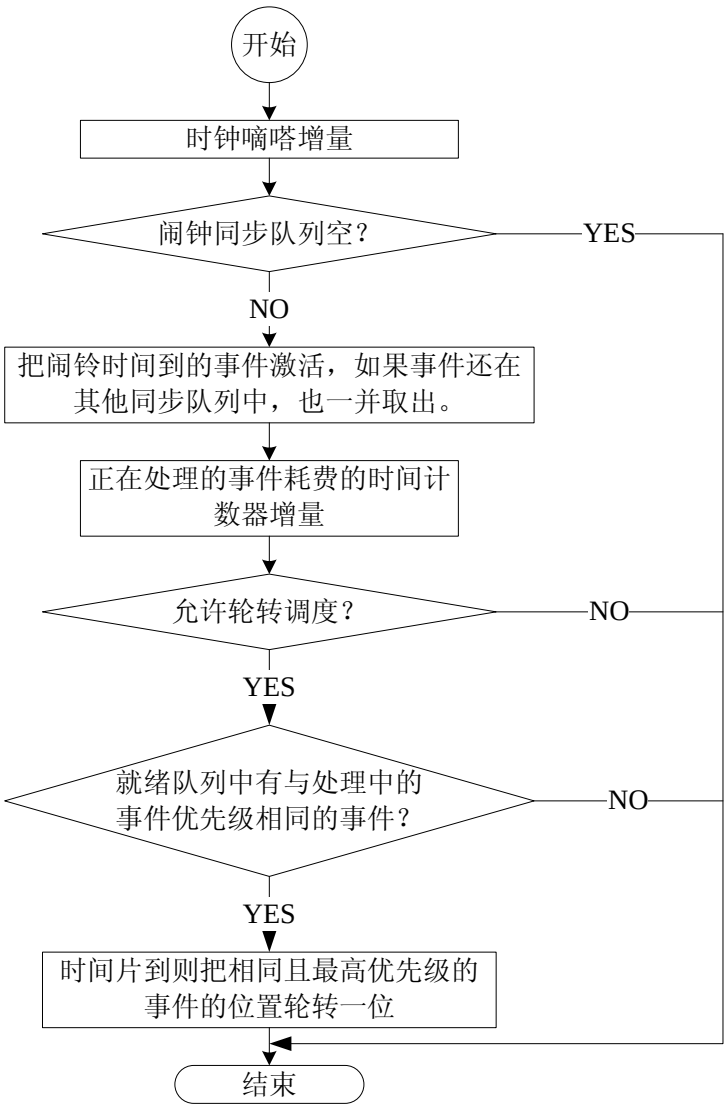


图 4-13 __y_isr_tick 函数流程

4.6 运行模式

djyos 有三种内存模式。

4.6.1 单映像模式（si 模式）

单映像模式（single image，缩写si），这种模式的可执行程序与没有操作系统支持下的前后台程序实际上并无多大的不同，操作系统和应用程序一起编译成一个大的可执行程序，运行时使用单一的地址空间，这是最简单的一种形式。这种模式缺点是只支持一个应用程序，不能形成固定的、编译好的操作系统可执行文件，而是每次修改应用程序都要连操作系统一起重新编译，用相似平台发展多种型号产品，或者产品维护修改用户代码时，虽然操作系统代码无需修改，但还是要与应用程序一起重新编译，存在操作系统被因笔误而误修改的可能。

si模式的优点是短小精悍，软件复杂度低，非常适合那些资源比较紧张的小型嵌入式系统。图 4-14 画出了两种典型的单映像模式代码和数据在内存中的映像图，嵌入式CPU以及内存配置千变万化，不能一一列举。图 4-14 适用于这样的系统：1、上电（复位）后从 0 地址取第一条指令；2、没有地址重映射机制。

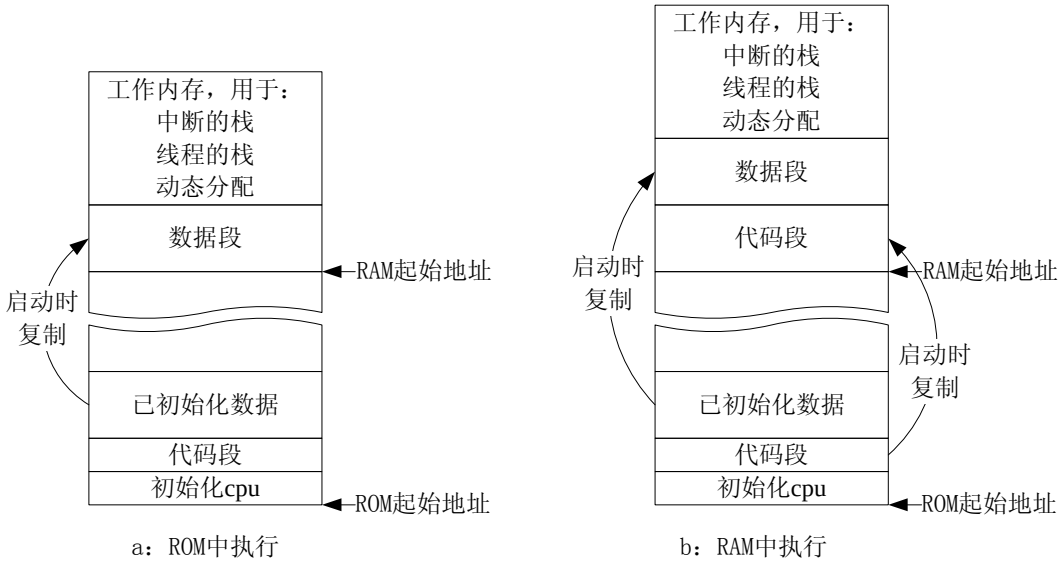


图 4-14 单映像模式典型的内存映像图

4.6.2 动态加载单进程模式（dlsp 模式）

动态加载单进程模式（dynamic load simplex process，缩写 dlsp），执行时，与 si 模式一样，操作系统与应用程序使用单一地址空间。编译和存储则不一样，操作系统、应用程序和第三方组件可以独立编译和存储，还支持多个应用程序分别独立编译和存储。应用程序可以由文件加载，也可以从串口、网口等通信口下载，可以在启动时加载，也可以在软件运行过程中动态加载和卸载。这种模式下，应用程序没有固定的运行地址，他们的运行地址由操作系统在加载时动态分配。嵌入式系统一般使用交叉开发环境开发，宿主计算机在编译和连接用于 dlsp 模式的应用程序和第三方组件时，由于执行地址还没有确定，只能生成可重定位的可执行文件，不能完成绝对地址连接。绝对地址连接的工作就只有留到目标嵌入式系统上执行了，操作系统加载器需要分析可执行文件，为其分配代码和数据空间后，才能知道应用程序运行时代码和数据的确切地址，然后把代码定位到这个地址，最后把应用程序拷贝到目标地址才可以执行。这种模式虽然比 si 模式复杂，但可以生成独立、固定的操作系统可执行程序，而且便于调试，操作系统的调试模式就是在这种模式的基础上实现的。更重要的是，如果你把项目分成多组件开发，每个组件独立编译的话，那么，每个组件可以拥有独立的符号命名空间，这给定义组件内可见的全局变量创造了条件。

众所周知，只要知道函数的地址以及类型声明，且该函数的地址在可访问的范围，就可以直接调用。dlsp 模式下，独立于应用程序编译的操作系统，api 函数的地址是确定的，应用程序与操作系统使用单一的、相同访问属性的地址空间，使得 api 函数的地址对于所有应用程序都是可以访问的。因此，只要在编译应用程序时引入操作系统的符号表，操作系统的 api 函数就可以直接调用，而无需使用软中断的方式实现。

如 图 4-15 所示，dlsp的内存映像与si模式有巨大的不同，它不再是统一的代码段和数据段，而是各组件（应用程序）有独立的代码段和数据段。dlsp模式一般用在较复杂的嵌入式系统，这种系统的内存配置比较充分，很少在ROM中运行的情况，图 4-15 只画出了在

RAM中运行的映像图，这并不是说，dlsp模式下操作系统不能在ROM中运行。

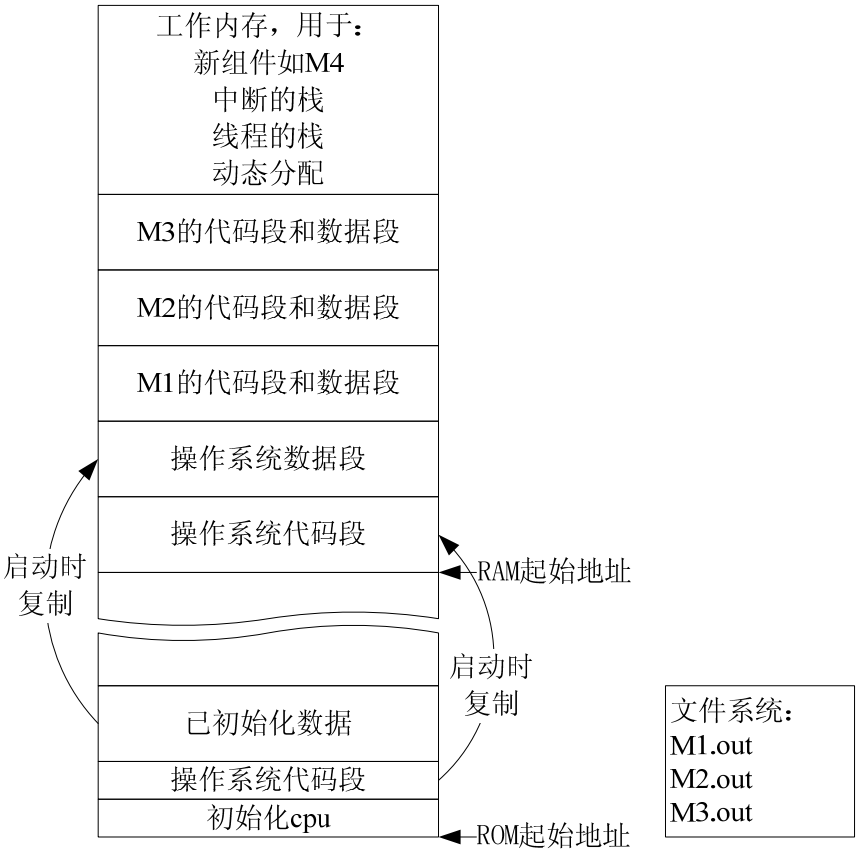


图 4-15 dlsp 模式典型的内存映像图

回顾第 3.2.2 节所述的友好组保护，si模式不能实现友好组划分，但dlsp模式可以实现逻辑隔离保护。

4.6.3 多进程模式（mp 模式）

多进程模式（multitude process，缩写mp）。在这种模式下，应用程序在进程虚拟机中运行，只有支持MMU的CPU可以实现这种模式。这种模式的每个应用程序在独立的进程中执行，他们拥有都有独立、固定而且相同的地址空间。mp模式的内存模式如 图 3-8 所示。

si模式只支持一个应用程序，这很好理解，si模式的所有代码包括操作系统一起编译成一个可执行文件，没听说过多个应用程序编译成一个可执行文件的，dlsp模式和mp模式都可以支持同时安装多个应用程序分时共享CPU（即同时执行），由于djyos是实时操作系统，因此多个应用程序同时运行需要符合 3.1.4 节的规定。本书成书时，si模式已经完成，dlsp模式正在开发中，mp模式尚未开始开发。

4.7 系统启动

硬件上电后，与任何操作系统一样，djyos操作系统也需要执行一系列的初始化过程，才能建立用户应用程序运行环境。图 4-16 显示了djyos操作系统的完整启动过程。本节后续部分将对 图 4-16 做详细的说明。

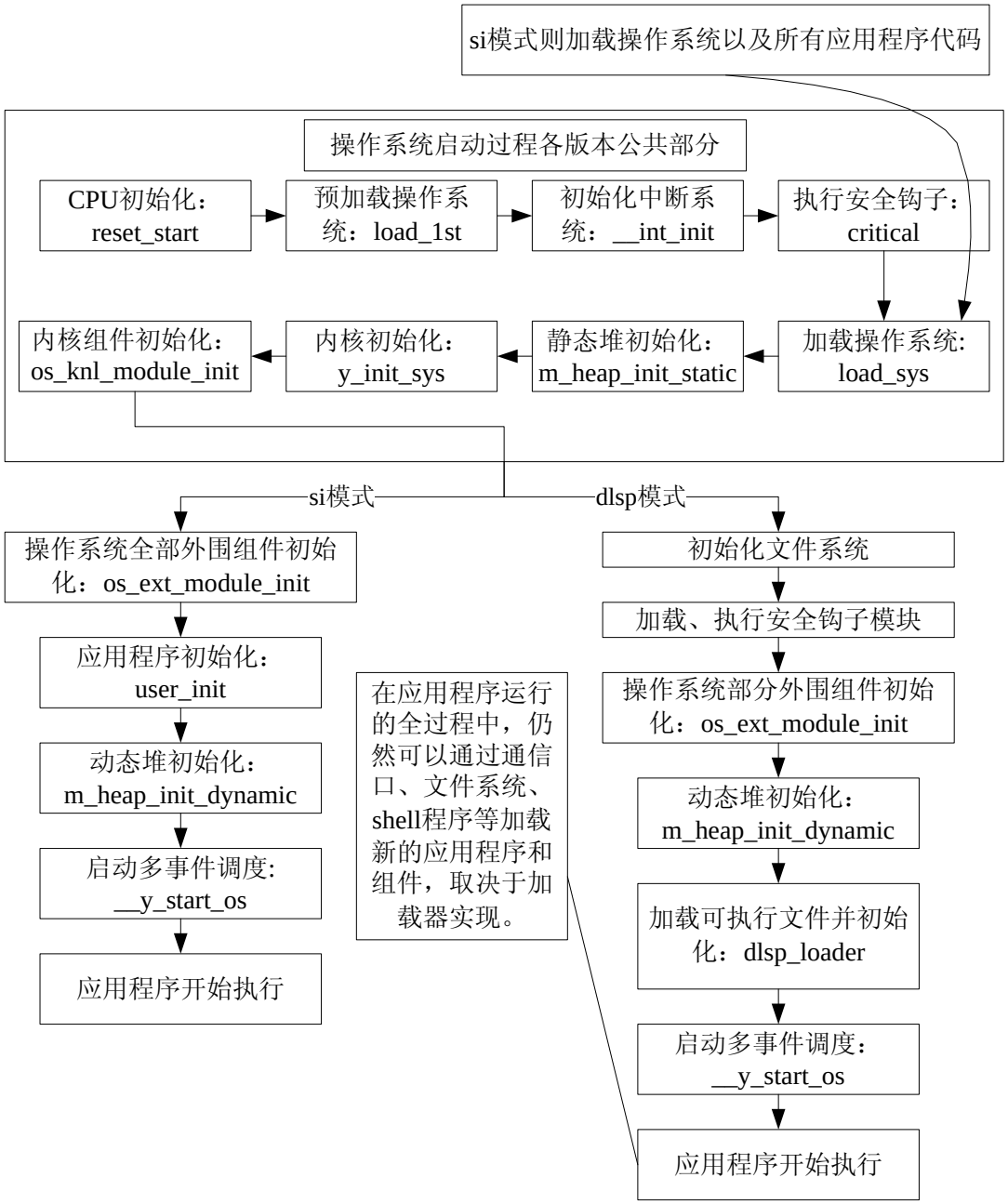


图 4-16 djyos 系统启动过程

4.7.1 系统启动过程各模式公共部分

4.7.1.1CPU 初始化

系统复位后，立即执行硬件初始化工作，硬件初始化需要访问 CPU 的特定资源，一般由汇编程序实现。硬件初始化工作需要完成：

1. 关闭所有中断，由于此时中断向量表尚未初始化，如果发生中断，将会出现不可预料的后果。一般来说，CPU 上电后会把中断初始化为禁止状态，但是，如果在运行过程中由于软件错误或者其他原因造成软复位的话，中断可能处于开启状态。比

如在 ARM 上，无论什么原因使 PC 赋值为 0，即产生一个软件复位。

2. 设置 CPU 时钟、CPU 工作模式，有许多 CPU 有多种工作模式，我们需要把 CPU 设定为特定的模式，一般是设为刚上电后的模式。一般来说，上电后，CPU 处于特权模式，但操作系统遇到严重错误而导致软件复位的话，CPU 模式可能是任意的，这就需要重新设置成上电后的模式，以防止因模式错误而导致出错。
3. 设置存储器工作参数，一般来说，CPU 上电后执行的第一条指令所在的地址区域的总线参数，可以通过硬件来设定，使该区域在上电后立即可以正确访问。其他存储区域，则要用软件正确设置了相应区域的总线参数后才能访问。为了 CPU 能正确访问内存以及内存映射的外设，上电后访问内存之前一定要初始化总线的工作参数。与实际存储器最匹配的总线参数能使系统发挥最大效能，提高软件运行速度。
4. 设置栈指针，这个过程必须在软件第一次使用栈之前进行，一般来说，大多数 CPU 调用函数时会自动把返回地址保存在栈中，对这类 CPU，栈指针必须在第一次调用函数之前设置好，因此，设置栈指针的工作也就不能用函数实现。ARM 比较特别，函数调用时返回地址保存在 LR 寄存器中，在设置栈指针之前允许子程序调用但不允许嵌套调用。故 ARM 允许使用子程序设置栈，但这不是通用的方法，有许多网上流传的 ARM 初始化代码和一些开发工具提供的 examples 中的初始化代码使用这种方式，特别在此交代一下。从严谨计，ARM 版本的 djyos 中初始化代码不使用函数设置栈。

4.7.1.2预加载操作系统

计算机上电或者复位后，所有代码和数据都保存在加载区中，需要加载到执行区才能执行，对于只读属性的代码和常数表，加载区和执行区可能是重合或者部分重合的，而读写属性的变量，则只能在内存中开辟执行区。在djyos操作系统为了实现安全钩子（紧急代码）先于操作系统执行（见 4.7.1.4），加载并启动操作系统内核前，需要执行一次预加载过程，预加载将加载以下模块：

1. 中断系统，djyos 允许在操作系统加载和初始化过程中使用实时中断（原因见 4.7.1.3），因此，在加载和初始化操作系统的主要部分之前，必须先加载和初始化中断系统。
2. 紧急代码，包括“工程目录\djyos\kernel\load1\port_my_os\critical.c”文件和用户编写的紧急代码，建议用户把紧急代码存储在“工程目录\usercode\critical\”目录中，
3. 主加载器，初始化中断系统和调用安全钩子后，就要执行加载器以加载整个操作系统，所以预加载时还应该把加载器加载到执行区中。

由于嵌入式系统存储器配置的多样性（参见 表格 2-1），加载过程也可能有多种形式。预加载器与被预加载的代码一起保存在复位向量所在的BROM中，预加载器也必定会在该BROM中执行。而被加载的模块则可能在BROM中或者在内存中执行，如果在BROM中执行，预加载执行操作：

1. 上述 3 个模块的 rw 段（初始化过的全局变量）拷贝到运行时的内存地址。
2. bss 段（未初始化的全局变量）清零。

如果在内存中执行，除以上两步操作外，还要执行：

1. 把 text 段（代码段）拷贝到运行时内存地址。
2. 把 rodata 段（常量表和字符串）拷贝到运行时内存地址。

如何让加载器识别需要预加载的代码呢？djyos 是使用段名来识别的，ARM 版本使用

GCC 编译，在连接脚本文件中，定义了一系列符号帮助预加载器识别，在 pre_load.c 文件中 extern 导入进来就可以了。该连接脚本文件是根据 makefile 在 make 时自动生成的，请参考本书所附代码的 makefile 文件。

```
extern ucpu_t text_preload_load_start[];      //预加载代码段的加载起始地址
extern ucpu_t text_preload_run_start[];       //预加载代码段的运行起始地址
extern ucpu_t text_preload_run_limit[];       //预加载代码段的运行终止地址
extern ucpu_t rodata_preload_load_start[];    //预加载只读数据段的加载起始地址
extern ucpu_t rodata_preload_run_start[];     //预加载只读数据段的运行起始地址
extern ucpu_t rodata_preload_run_limit[];     //预加载只读数据段的运行终止地址
extern ucpu_t rw_preload_load_start[];        //预加载读写数据段的加载起始地址
extern ucpu_t rw_preload_run_start[];         //预加载读写数据段的运行起始地址
extern ucpu_t rw_preload_run_limit[];         //预加载读写数据段的运行终止地址
extern ucpu_t zi_preload_start[];             //预加载清零段的起始地址
extern ucpu_t zi_preload_limit[];             //预加载清零段的终止地址
```

在导入上述常量后，函数load_preload函数（代码 4-15）执行加载操作。

代码 4-15 预加载操作系统

```
void load_preload(void)
{
    uint8_t *src,*des;

    if(text_preload_run_start != text_preload_load_start)          //拷贝代码段
    {
        for(src=text_preload_load_start,des=text_preload_run_start;
                                            des<text_preload_run_limit;src++,des++)
            *des=*src;
    }
    if(rodata_preload_run_start != rodata_preload_load_start)    //拷贝只读数据段
    {
        for(src=rodata_preload_load_start,des=rodata_preload_run_start;
                                          des<rodata_preload_run_limit;src++,des++)
            *des=*src;
    }
    if(rw_preload_run_start != rw_preload_load_start)              //拷贝读写数据段
    {
        for(src=rw_preload_load_start,des=rw_preload_run_start;
                                          des<rw_preload_run_limit;src++,des++)
            *des=*src;
    }
    for(src=zi_preload_start;src<zi_preload_limit;src++)           //清零 0 初始化段
        *src=0;
    sys_start();
}
```

以上是用 gcc 编译的设置，如果被移植到其他编译器环境下，可参考所选编译环境的说

明，许多编译器可以用编译指示 pragma 来指定段名，pragma 指示的用法可能因编译器的不同而有所不同，应该应该参考编译器的说明文档。

4.7.1.3初始化中断系统

从 第 6 章可知，djyos中中断类型分为异步信号和实时中断，异步信号可以使用绝大多数系统调用，与操作系统内核密切相关，必须在内核初始化完成并启动多事件调度后才可以使用。而实时中断则与操作系统内核关系不大，只要中断模块初始化好了，内核初始化完成之前就可以使用。而操作系统初始化过程和用户程序的loader过程的时间可能很长，有许多嵌入式系统，要求从上电后立即开始对外部事件做出反应，他们不能容忍长时间的等待。例如有些机电设备中，内嵌的嵌入式控制系统与电动机使用相同的电源，一上电电动机就开始工作，而嵌入式系统也必须立即开始控制，否则就可能导致电机失控。但是操作系统完成初始化之前，应用程序的进程/线程执行环境还没有建立，进程/线程级代码当然也不能执行，异步信号与操作系统内核密切相关，当然也不可能响应异步信号，此时，唯一能够供用户驱遣的只有实时中断。因此，在完成预加载后，首先调用的就是中断系统初始化，使用户可以使用实时中断。此后，安全钩子函数可以安装实时中断例程，并把相应的中断线属性设置为实施中断，就可以一边加载操作系统一边响应实时中断了。

4.7.1.4调用安全钩子函数

si模式下，只在这里执行一次安全钩子函数，dlsp模式下，除在这里调用安全钩子外，还可以在初始化内核后、加载操作系统外围组件和应用程序前，加载、执行另一个安全钩子模块，参见第 4.7.3.1 节。

嵌入式系统面对各种各样的现场情况，有许多设备要求其配套的嵌入式系统的关键部分在上电后立即开始工作，他们不能容忍失控状态持续到“漫长”的操作系统初始化和应用程序加载工作完成。也有一些系统在上电后某些状态是不确定的，现场设备又不容许不确定的状态持续持续很长时间，或者系统上电后虽然处于确定状态，但这个状态并不是现场设备所容许的。这些情况都要求 CPU 在上电后立即执行某些操作，把系统置于正确的状态。显然，这些工作不能由操作系统设计者来完成，因为他们并不知道使用操作系统的目标设备究竟要在上电伊始执行什么样的工作，这需要应用系统的设计者来完成，而 djyos 系统为完成这些紧急工作提供了接口，在“工程目录\djyos\kernel\load1\port_my_os\critical.c”文件中提供了一个空函数 critical，用户可以把目录中的“_my_os”改为自己喜欢的名字，初始化中断系统后，将调用这个函数。用户可以在这个函数中添加代码，完成必须在系统初始化前完成的功能，因为在这个函数中执行的一般是安全攸关的功能，故称“安全钩子”。需要特别注意的是，critical 函数只能调用本文件中的代码和中断模块中的代码，不能调用库函数、操作系统调用等。在实际的嵌入式系统中，下列情况可能需要使用 critical 函数：

1. 硬件上电后有确定状态，但是这个状态可能与实际要求不相符，不允许这个状态持续到操作系统初始化完成。典型的是三星的 ARM7TDMI 核的 S3C44B0X 的 LCD 驱动电路，复位后输出时序是确定的，但如果实际安装的 LCD 面板与默认时序不符，长时间持续错误的时序会损坏 LCD 面板。就需要在上电后立即关闭 LCD 面板，或者把 LCD 驱动时序设置为与 LCD 面板匹配的状态，这个操作只能使用 critical 函数来完成。
2. 硬件上电后状态不确定，有可能处于错误状态。许多 CPU 在复位后 IO 口都处于高

阻态，这是一种不确定的状态，外部与之相连的电路可能判为 1，也可能判为 0，甚至出现振荡，这是不希望出现的，应该在 critical 函数中设置 IO 口，尽快结束这种不确定状态。当然硬件应该避免发生这样的事情，一个上拉电阻或者下拉电阻就可以使 IO 口输出处于稳定状态。

3. 由错误导致异常复位，有些硬件复位不会改变状态。我们不知道复位前是否产生了错误操作，复位后，应该以最短时间纠正误操作产生的错误状态，而不应该使错误状态持续到操作系统启动完成。可以在 critical 函数中判断是否发生了异常复位，以及复位前是否有误操作，并做相应的处理。
4. 由错误导致异常复位，复位会使某些硬件恢复到初始化状态，而系统实际上希望维持复位前的状态。典型的是电源控制系统中，复位后的状态常常是关闭输出的，待系统启动后才开始输出。而在运行过程中，如果发生异常复位，却不允许因复位而长时间关闭电源，需要立即恢复输出。这就要在软件或者硬件设计中提供检查复位原因的接口，一般在指定的内存单元写上特定的数可以作为上电复位标志，在 critical 函数中检查这个接口，如果发现是异常复位，则立即恢复输出。
5. 有些嵌入式控制系统与被控制设备使用相同的电源，而被控制设备需要在上电后立即开始工作，嵌入式系统中相应的控制功能也就必须从上电伊始就不间断地工作。这常采用实时中断响应来实现，这就需要在安全钩子函数中安装中断服务例程并设置中断口的工作参数。由于在调用安全钩子函数前，中断系统被初始化，并且实时中断已经允许，用户可以在紧急代码中使用实时中断（中断用法参见 第 6 章）。
6. 嵌入式系统可能会因各种原因而复位（比如看门狗），有的系统不容许在复位后操作系统和应用程序重新加载和初始化期间完全失控，它要求系统的一小部分关键控制机能持续工作，这也要用到安全钩子。

编外：失控的 CPU 风扇

请读者做一个实验，打开你的计算机机箱，然后开机，你会发现开机后 CPU 风扇会发疯一样狂转好几秒钟，同时发出巨大的响声，然后才恢复正常，即使是在 CPU 全冷状态下（关机超过半小时）开机也是如此。众所周知，计算机软件会根据 CPU 的温度负载情况自动调节风扇的转速，以求在静音和散热之间找到一个平衡点。那么，刚开机的时候，CPU 是冷却的，风扇是不应该启动的，疯转的原因只有一个，就是计算机启动时，控制风扇转速的软件还没有执行，风扇处于失控状态。当然，计算机风扇多转一会除了会产生点噪音，多消耗一点电以外，不会有严重的后果，但是，如果心脏起搏器疯跳 5 秒钟，会有什么后果呢？

调用安全钩子，就可以在系统复位后极短时间内（毫秒甚至微秒级），启动装置关键的控制功能，使其从开机伊始就处于可控状态下。

4.7.1.5加载操作系统

到此时，操作系统的绝大部分代码还躺在加载区中，需要加载到执行区，加载器将把除预加载没有完成的操作系统代码加载到执行区。加载工作有 load_sys()函数完成，该函数在预加载时已经被拷贝到执行区。

与预加载一样，操作系统代码也可能在 BROM 中或者在内存中执行，如果操作系统在 BROM 中执行，加载器执行操作：

1. 把操作系统的 rw 段（初始化过的全局变量）拷贝到执行区的内存地址。
2. bss 段（未初始化的全局变量）清零。

如果操作系统在内存中执行，除以上两步操作外，还要执行：

1. 把 text 段（代码段）拷贝到执行区的内存地址。
2. 把 rodata 段（常量表和字符串）拷贝到执行区的内存地址。

如果操作系统代码保存在BROM中，则加载器的工作方式与预加载器一样，参见第 4.7.1.2 节，这里就不赘述了。

4.7.1.6准静态内存分配初始化

为准静态内存分配做好准备，参见第 7.1 节。

4.7.1.7操作系统内核初始化

__y_init_sys ()函数执行内核初始化，在这之前，还执行了指令延时常数初始化，参见第 11.3.2 节，__y_init_sys ()函数将执行如下操作：

1. 初始化事件类型控制块表，模拟登记事件类型，把第 0 个事件类型控制块初始化为系统服务事件类型，其余事件类型控制块初始化为空闲。
2. 初始化事件控制块表，模拟弹出系统服务事件并把它加入就绪队列，其他事件控制块则进入空闲队列。
3. 为系统服务事件创建线程虚拟机并分配给系统服务事件。

完成这一步后，虽然用户线程还不能执行，但所有的内核调度相关的系统调用都可以使用，允许使用的系统调用计有：

y_evtt_regist
y_evtt_unregist
y_get_evtt_id
y_event_pop
__y_event_ready
y_set_RRS_slice
y_get_RRS_slice

而下列服务因为它要求事件处理函数亲自调用，故只能在多事件调度开始后才可以使用：

y_timer_sync
y_event_sync
y_evtt_pop_sync
y_evtt_done_sync

4.7.1.8内核组件初始化

内核组件是指除调度器、事件队列、时间控制块表、内存管理以外的、内核运行必不可少的组件，这些组件计有：

资源队列管理器，这是一个无需初始化的函数库，参见 第 8 章。
锁模块，锁用于防止临界资源共享冲突，分信号量和互斥量两种，参见 第 5 章。
固定块内存分配模块，用于从内存池中按固定大小的块分配内存，参见第 7.3 节。

看门狗模块，参见 第 10 章。
泛设备驱动模块，参见 第 9 章。
os_knl_module_init 函数将完成上述初始化工作，读者可参看随书源代码。
经过这一步骤后，内核已经成型，

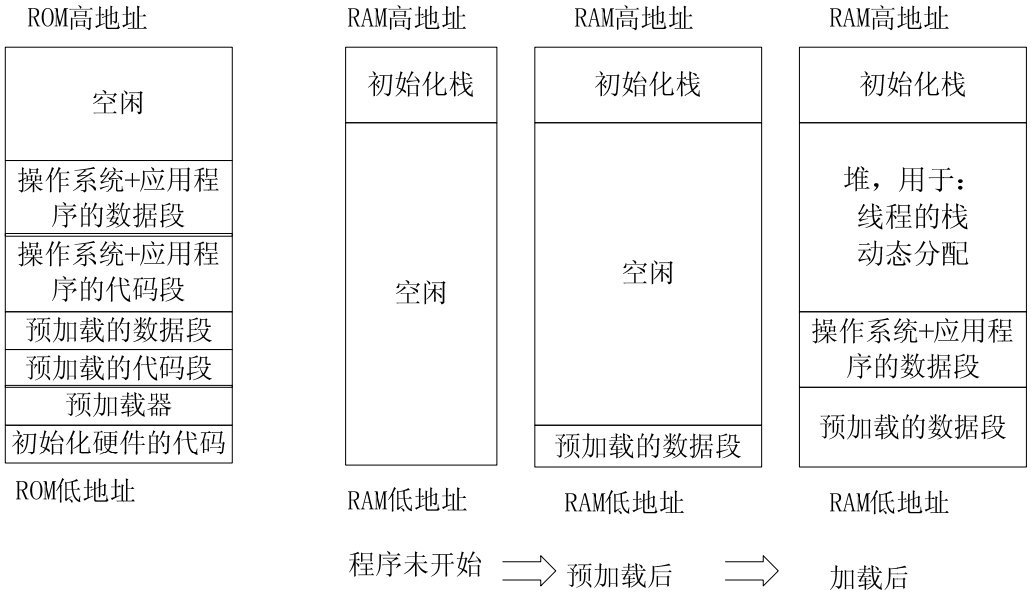
4.7.2 系统启动过程 si 模式部分

4.7.2.1操作系统外围组件初始化

os_ext_module_init ()函数完成操作系统外围组件初始化工作，这是一个与具体应用系统相关的个性化过程。如果你的系统配备了文件系统，则在这里调用文件系统初始化函数；如果配备了文本终端设备，在这里调用文本终端设备初始化函数。等等。

4.7.2.2加载应用程序

加载应用程序就是把应用程序的代码和数据拷贝到运行时地址。由于在si模式下，操作系统与全部应用程序在一起编译成一个可执行文件的模式，因此si模式的应用程序加载在第 4.7.1.5 节已经完成了。在这里讲述加载过程是为了凸显si模式与dlsp模式加载应用程序的不同。嵌入式系统的地址模式很多种，这里以平板地址且不重映射地址的情况为例说明si模式启动过程中代码加载过程。图 4-17 和 图 4-18 分别是程序在ROM中运行和程序在RAM中运行情况下，操作系统启动过程中内存视图的变迁情况

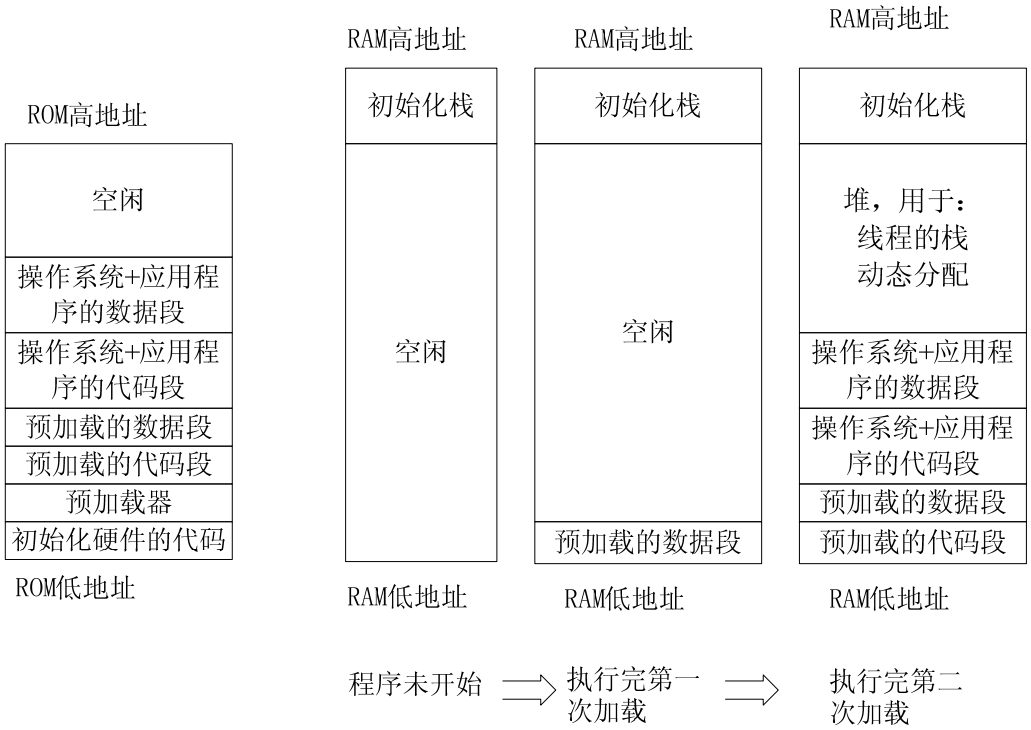


注：所有数据段包含RW段和BSS段

图 4-17 代码在 rom 中执行的模式

1. 系统刚上电时，只有 ROM 中有数据，RAM 时空的，此时首先执行 ROM 中的初始化 CPU 以及基本硬件的代码，这些代码主要设置 CPU 的主频、工作模式、内存控制器、中断控制器等。

2. 初始化CPU后，接着执行预加载器，参见第 4.7.1.2 节。
3. 接着调用中断模块初始化和主加载器，参见第 4.7.1.5 节。



注：所有数据段包含RW段和BSS段

图 4-18 代码在 ram 中执行的模式

图 4-18 和 图 4-17 相比，只多了一个拷贝代码段的过程，其余是一样的。

4.7.2.3用户应用程序初始化

这一步完全取决于用户，需要完成一系列跟具体应用相关的初始化。通常需要在这一步初始化专用硬件系统，建立用户级的泛设备 driver 结点。

4.7.2.4动态内存分配初始化

所有初始化过程完成以后，类静态分配内存策略即告完成（详见第 7.1 节），此后的内存分配应该使用块相联分配策略。m_heap_init_dynamic函数初始化动态内存分配，参见第 7.2 节。

4.7.2.5启动多线程管理

至此，在用户初始化过程中可能已经 pop 了许多事件，操作系统的多线程管理也已经做好了准备，可以说是万事俱备，只欠东风了，就等着启动多事件调度了，此后，应用程序将投入运行。__y_start_os 函数将完成这一步工作。

代码 4-16 启动多事件调度

```
void __y_start_os(void)
{
    __int_reset_asyn_signal();                          //1
    __y_init_tick();                                    //2
    __y_select_event_to_run();                          //3
    pg_event_running = pg_event_ready;
    __asm_turnto_context(pg_event_running->vm);       //转入线程上下文       //4
}
```

代码说明如下：

1. 复位异步信号管理器的状态到初始状态，属于容错性质，目的是防止用户初始化过程中没有成对调用 int_restore_asyn_signal 和 int_save_asyn_signal 函数，多事件调度开始前的正确状态。
2. 初始化时钟嘀嗒硬件，并启动tick中断，但此时异步信号总开关（参见 第 6 章）仍然是关闭的，故仍然不会响应tick中断，直到第一个线程开始执行后，在其虚拟机引擎函数__vm_engine中打开异步信号总开关，才能响应tick中断。
3. 为第一次切入的事件获取虚拟机，参见第 4.3.11.2 节。
4. 切入第一个处理的事件的虚拟机，参见第 4.3.6 节。

4.7.3 系统启动过程 dlsp 模式部分

4.7.3.1初始化文件系统

在si模式中，文件系统是作为操作系统外围组件与其他外围组件一起初始化的，dlsp模式则在其他外围组件初始化之前单独初始化，是因为dlsp模式下后续的加载过程需要文件系统的支持，只能单独初始化。本来，如果单独考虑dlsp模式，文件系统可以作为内核组件在第 4.7.1.8 步初始化，但该步骤是各模式的公共部分，而si模式下可能根本就没有文件系统。

4.7.3.2安全钩子模块

在加载操作系统内核之前，我们已经调用了一次安全钩子函数了，为什么在这里又有一个安全钩子模块呢？原来，dlsp 模式与 si 模式不同，dlsp 模式下操作系统内核、操作系统外围组件、独立组件、应用程序等都是单独编译的，安装安全钩子函数需要修改操作系统代码并重新编译。应用系统的作者使用的很可能是由第三方编译好的操作系统，或者虽是源代码，但应用系统项目组并没有打算修改其源代码，或者根本就没有配备能够修改操作系统源代码的人。还有就是，项目组虽然可以重编译操作系统，但企业内部更希望保持操作系统版本纯净而不允许修改。在这些情况下，项目组就无法利用安全钩子函数了。所幸的是，djyos 的核心部分很小，只有不到 100K，初始化核心部分所花的时间不长，djyos 为 dlsp 和 mp 模式保留了另一个快速通道，允许用户在完成初始化内核部分后，立即加载和执行安全钩子模块。无论是安全钩子函数还是安全钩子模块，都有相同的限制：不能使用事件调度，也不能使用异步信号中断，只能使用实时中断。

4.7.3.3操作系统外围组件初始化

本部分与第 4.7.2.1 节si模式的操作系统外围组件初始化类似，不同的是，si模式在此初始化了全部的外围组件，而dlsp模式只初始化了部分组件，原因是，dlsp模式下部分外围组件可能以独立组件的形式从文件系统单独加载。

4.7.3.4动态内存分配初始化

与si模式的第 4.7.2.4 节相同。

4.7.3.5加载可执行文件

djyos 认为，一个应用程序就是由一个或者多个独立和非独立组件构成的，加载应用程序就是把其所包含的组件一一加载，因此 djyos 从内核的角度看，不存在单纯的加载应用程序一说。

dlsp模式是操作系统、独立组件分开编译，操作系统核心部分存储在可执行的BROM（参见第 2.1 节）中，上电即开始执行，而独立组件则存储在文件系统或者其他存储介质中，也可以用通信线路下载。在dlsp模式，组件加载器dlsp_loader用于加载独立组件，图 4-19 显示了加载器的执行过程。根据dlsp_loader函数将决定如何获取组件的可执行文件，是通过文件系统加载、还是通过通信口加载，或者其他方式，或者多种方式的组合。

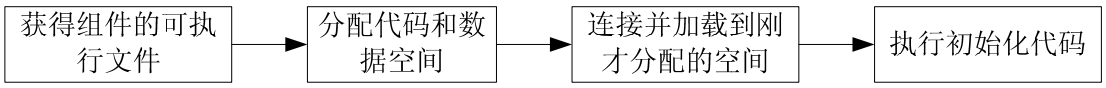


图 4-19 dl 模式加载器

4.7.3.6从文件系统加载可执行文件

从文件系统加载，应该事先把可执行文件保存到文件系统中，并把需要加载的可执行文件名保存到 autorun.cfg 文件中，autorun.cfg 是一个文本文件，假设 autorun.cfg 文件的内容如下：

```
demo1.out
demo2.out
demo3.out
```

表示操作系统启动后，需要加载 3 个可执行文件，这 3 个文件并可以是独立的 3 个应用程序，也可以是一个应用程序的 3 个功能模块。

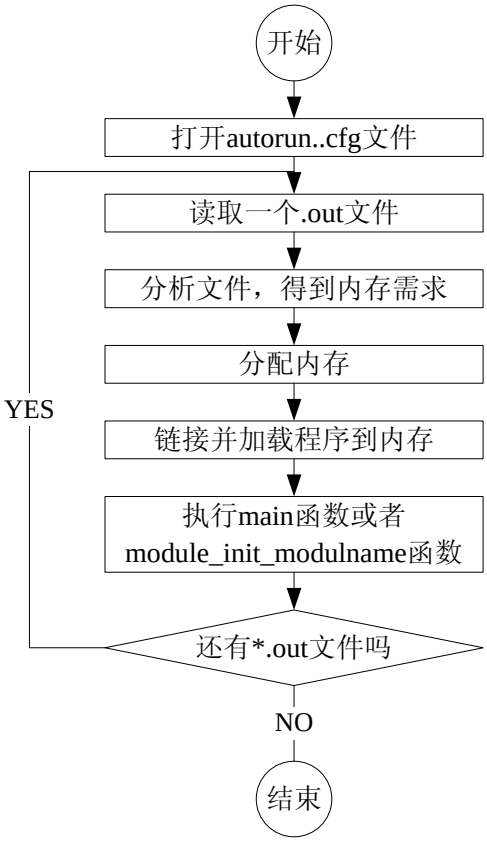


图 4-20 从文件系统加载应用程序的流程

1. 因为编译时并不知道应用程序的执行地址，不能执链接接操作，所以这里的所有*.out 文件都是未经绝对定位的文件。在主机上运行的交叉编译环境中，编译系统实际上是分两步工作的，首先是由编译器把许多 C 程序文件编译成目标代码文件，然后由链接器把这些目标文件转变为绝对地址文件。由于文件在主机上未经链接，因此在操作系统中需要包含一个链接器。
   a) 操作系统获得文件后，首先分析该程序需要多少内存空间来存放代码以及变量，然后从堆中分配相应的内存。如果分配失败，则返回错误，如果分配成功，则继续执行下一步工作。
   b) 分配好内存之后，也就知道了软件的执行地址，可以执行链接操作了。链接后，加载器再把代码和数据拷贝到目标内存，把 zi 段（有些编译器叫.bss 段）清零。至此，可以执行应用程序的代码了。
   c) 如果.out 文件包含需要加载后执行的初始化代码，就必须提供启动函数。如果该文件包含一个应用程序，启动函数就是 main；如果包含需初始化的功能模块，启动函数名以 module_init_开头。module_init 函数是一个没有参数也没有返回值的函数。初始化函数完成该模块必须的初始化操作，需要特别注意的是，该函数必须能够返回，如果有死循环，操作系统将不能正常启动。

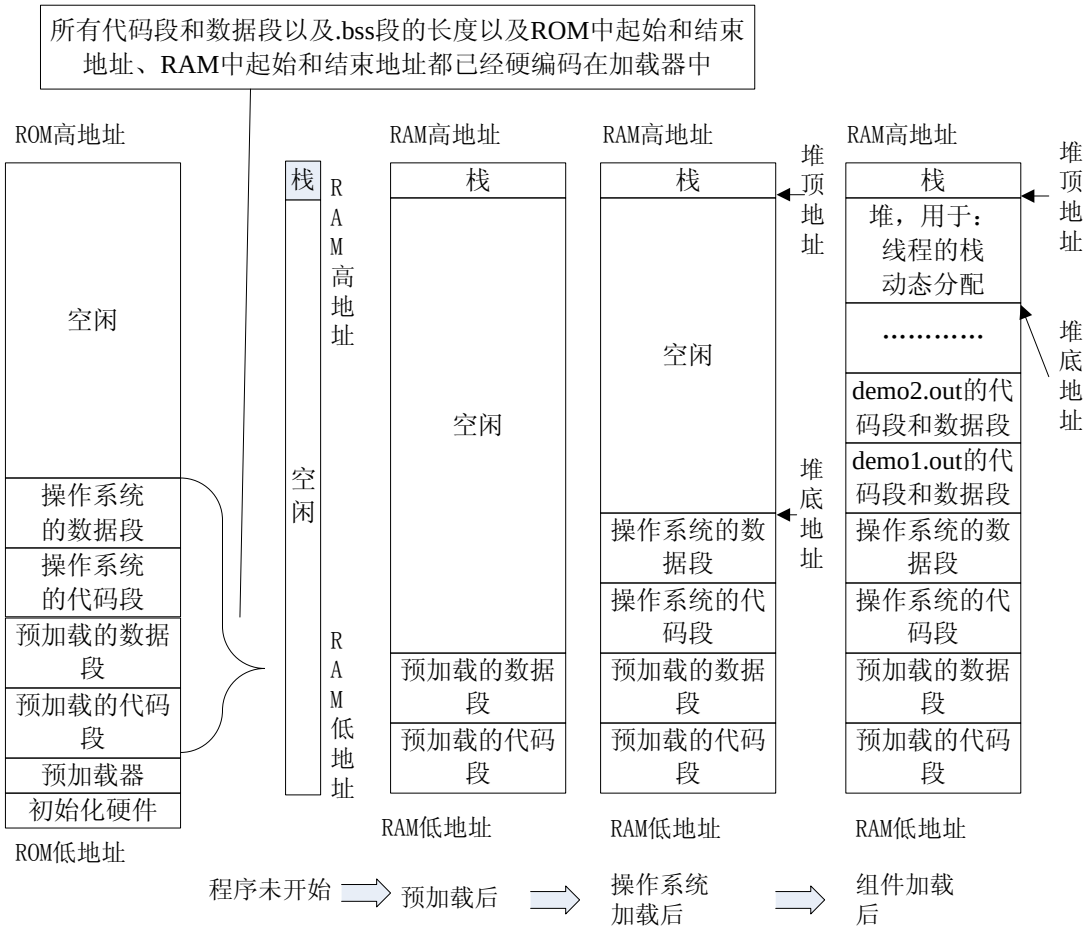


图 4-21 从文件系统加载应用程序的内存视图

1. 系统刚上电时，只有 ROM 中有数据，RAM 时空的，此时首先执行 ROM 中的初始化 CPU 代码，这些代码设置 CPU 的主频、工作模式以及内存控制器。
2. 初始化CPU后，接着是预加载和加载操作系统，参见第 4.7.1.2 节和第 4.7.1.5 节。
3. 加载完操作系统后，就可以逐个加载可执行文件了。

4.7.3.7从固定地址获取可执行文件

应用程序编译好以后，直接用外部工具把程序烧录到指定的地址。应用程序加载器直接到这个地址读取应用程序文件。加载过程与从文件中加载的方法基本相同，不同点是 autorun.cfg 文件的烧录地址硬编码在加载器中，应用程序文件的地址则可能硬编码在加载器中，也可能写在 autorun.cfg 文件中，可以在没有文件系统的平台上使用。

4.7.3.8从 DROM 中读取可执行文件

从DROM（见 2.1 节）设备中读取，这种方法和上一种方法类系，不同的是相应DROM的驱动才能才能读取数据，这样做的好处是允许使用多种介质。

4.7.3.9调试模式加载可执行文件

在调试模式下，目标机的运行受主机控制，主机可以发送各种调试命令，下载程序只是众多调试命令的一种。

在调试模式下，主机知道目标机的内存使用情况，也知道目标机空闲内存地址，因此，主机可以执行链接操作，从主机下发的程序是经过链接的绝对地址文件，加载器只需要按照主机的要求把程序写到指定地址就可以了，.bss 段清零也是按照主机命令进行。

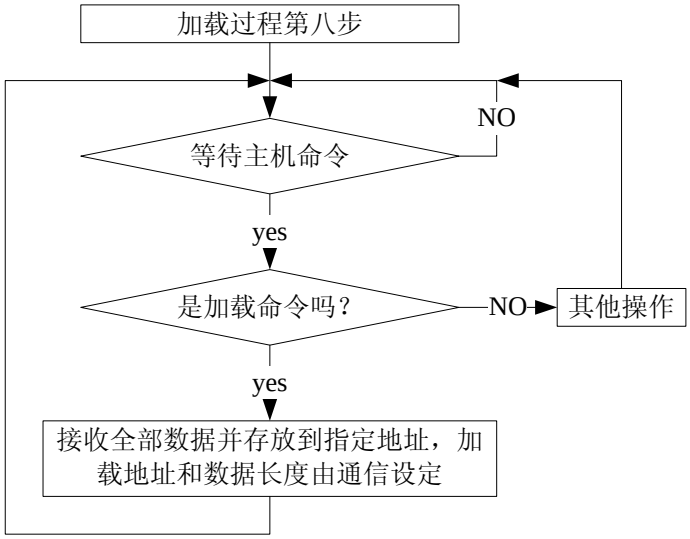


图 4-22 调试模式加载器执行流程

九九加一原则

“九九加一”原则是说，djyos 的设计力图使应用程序程序员的 99%的工作更加便利和简洁，而剩下的 1%的特别复杂的工作提供更大的解决的可能性。

“九九加一”原则本身并不像事件、线程一样是一个有形的实体，但确实贯穿 djyos 系统设计的总原则，djyos 系统无处不在地体现这个原则，可以说，“九九加一”是 djyos 系统的灵魂！

“九九加一”原则是计算机软件设计中简洁性和通用性的完美组合，99%的工作更加简洁使应用程序总体上更加简洁易读易懂易维护，展现了最大的简洁性；1%的可能性使用户能够实现其所有的目标，甚至有些传统认为只有裸机（没有操作系统支持的计算机）才能完成的工作，体现了最大的可用性。举例来说，在 djyos 系统的中断管理系统设计中，淋漓尽致地展现了“九九加一”原则，它把硬件中断划分为“实时中断”和“异步信号”两类，“实时中断”ISR 具有比一般操作系统更高的实时性，操作系统运行过程中永远不会主动关闭实时中断；而“异步信号”则比普通操作系统的中断更方便，异步信号的 ISR 可以从堆中分配内存，可以使用信号量，也可以打开、关闭和读写设备，甚至可以弹出事件在事件上下文中处理。

第5章 同步

同步的意思是在某些条件下协调多个线程之间步调，在多线程程序中，如果没有同步，个线程是自由运行、各行其是的，相互之间是异步的。同步主要用于：

1. 访问共享数据的多个线程之间需要协调步调，依次访问而不能同时访问。
2. 线程需要等待某些条件达成以后再继续运行，可以理解成线程需要与该条件同步，信号量和互斥量是一种很好的表示条件是否达成的载体。
3. 多线程共享资源，必须协调他们之间的步调，轮流使用资源，不能一窝蜂。
4. 定期执行的线程，时间是否到就成为它的同步条件，即闹钟同步。
5. ……

同步有主动同步和被动同步两种，主动同步指事件在处理过程中，主动停下来等待某条件达成，比如闹钟同步，可以这样理解，把闹铃响当作一个条件，正在处理的事件停下自己的脚步，直到闹铃响条件达成后继续处理，以使自己的步调与闹铃响条件保持同步；又比如事件同步，也是主动停下来等待被同步事件完成。事件被动同步则是被迫的，处理过程中，如果需要的资源得不到满足，则可能被动地进入阻塞态。比如申请内存得不到满足，则进入内存同步队列，请求信号量得不到满足，则进入信号量同步队列。

5.1 闹钟同步

操作系统很重要的功能之一——定时功能，就是由闹钟同步实现的。闹钟设定的同步条件是从当前时刻开始计算的时钟嘀嗒数，设置闹钟同步后，当前事件线程暂停执行，闹铃响后恢复到就绪队列，等待操作系统调度。

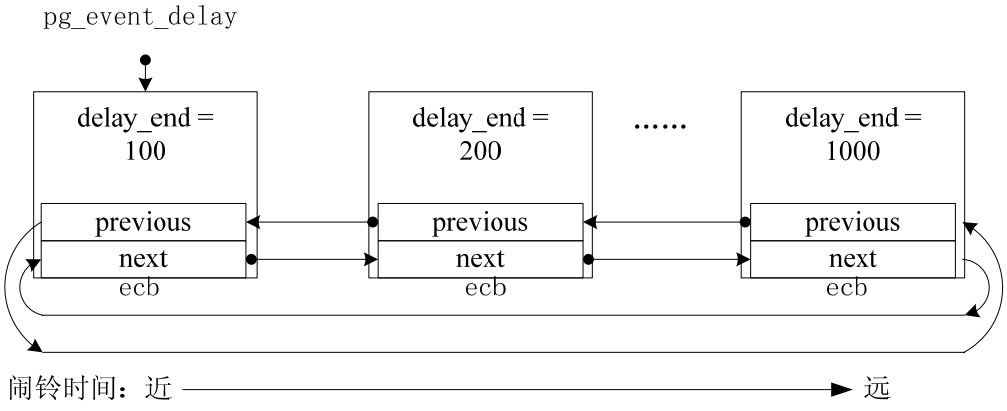


图 5-1 闹钟同步队列

图 5-1 是闹钟同步队列，图中delay_end就是闹铃时间。该队列是一个以闹铃时间排序的双向循环链表，全局指针pg_event_delay指向该队列的头部，pg_event_delay指针为NULL表示该队列不存在（没有事件正在使用闹钟同步）。系统中有一个自由运行的 32 位的全局时钟u32g_os_ticks，该变量每个时钟嘀嗒加 1，达到 0xffffffff后回绕到 0。当自由时钟达到闹铃响的时间，就把相应的事件从闹钟同步队列取出加入就绪队列。

在核心函数中，有 4 个函数和 1 个宏定义涉及闹钟同步，分别是：

__y_resume_delay 函数，从同步队列中取出一个事件。

__y_addto_delay 函数，把一个事件加入同步队列。

y_timer_sync 函数，本函数使正在处理的事件暂停设定的时钟嘀嗒数，然后再继续运行，参数是需要暂停的时钟嘀嗒数。执行本函数后，正在处理的事件将从就绪队列中取出，添加到闹钟同步队列中，而“闹铃响”时间就是当前自由时钟 u32g_os_ticks 与暂停嘀嗒数的和。

__y_isr_tick函数，时钟嘀嗒函数中有一部分代码是为闹钟同步服务的，它不断比较当前时钟是否与队列中事件的闹铃时间，把队列中闹铃时间已到的事件加入到就绪队列中。代码 5-1 是**__y_isr_tick**中节选的处理闹钟同步的部分。

#define y_event_delay(x) y_timer_sync(x)，在比较流行的操作系统中，实现跟 djyos 中闹钟同步相同功能的函数都叫任务延时，比如 vxworks 有 TaskDelay，ucosii 有 OSTimeDly 函数。定义 y_event_delay 完全是为了照顾程序员的习惯。

代码 5-1 **__y_isr_tick** 中闹钟同步相关代码

```
void   __y_isr_tick (ufast_t line)
{
    ……
    if(pg_event_delay != NULL)                                  //1
    {
        pl_ecb = pg_event_delay;
        while(1)
        {
            if(pl_ecb->delay_end = u32g_os_ticks)               //2
            {
                if(pl_ecb->sync_head != NULL)    //事件在某同步队列中    //3
                {
                    if(*pl_ecb->sync_head == pl_ecb)    //本事件是该同步队列的首事件
                    {
                        if(pl_ecb->multi_next == pl_ecb)      //队列中只有一个事件
                        {
                            *pl_ecb->sync_head = NULL;
                            pl_ecb->multi_next = NULL;
                            pl_ecb->multi_previous = NULL;
                        }else                          //队列中有多个事件
                        {
                            //头指针指向下一个事件
                            *pl_ecb->sync_head = pl_ecb->multi_next;
                            pl_ecb->multi_previous->multi_next
                                                = pl_ecb->multi_next;
                            pl_ecb->multi_next->multi_previous
                                                = pl_ecb->multi_previous;
                        }
                    }else               //本事件不是首事件
                    {
                        pl_ecb->multi_previous->multi_next
                                            = pl_ecb->multi_next;
```

```
                                pl_ecb->multi_next->multi_previous
                                                        = pl_ecb->multi_previous;
                        }
                        pl_ecb->sync_head = NULL;     //事件头指针置空
                    }
                    if(pl_ecb->next == pl_ecb)   //这是闹钟同步队列最后一个结点.     //4
                    {
                        pg_event_delay = NULL;
                        pl_ecb->last_status.all = pl_ecb->event_status.all;
                        pl_ecb->event_status.bit.event_delay = 0;
                        pl_ecb->event_status.bit.wait_overtime = 0;
                        __y_event_ready(pl_ecb);                                    //5
                        break;
                    }else
                    {
                        pg_event_delay = pl_ecb->next;
                        pl_ecb->next->previous = pl_ecb->previous;
                        pl_ecb->previous->next = pl_ecb->next;
                        pl_ecb->last_status.all = pl_ecb->event_status.all;
                        pl_ecb->event_status.bit.event_delay = 0;
                        pl_ecb->event_status.bit.wait_overtime = 0;
                        __y_event_ready(pl_ecb);
                        pl_ecb = pg_event_delay;
                    }
                }else
                    break;
            }
        }
        ……
}
```

代码 5-1 说明如下：

1. 代码首先查看闹钟同步队列是否空。
2. 从队列头部开始，找到所有闹铃时间已经到的事件，方法是把各事件的闹铃时间与全局时钟 u32g_os_ticks 比较。
3. 如果该事件是因为阻塞超时而进入闹钟同步队列的，要把它从同步队列中取出，参见第 0 节。
4. 把闹铃时间已到的事件从闹钟同步队列中取出，如果是队列中最后一个事件，则把全局指针 pg_event_delay 置空，否则继续查找。
5. 把取出的闹铃时间到的事件加入到就绪队列，该事件的优先级如果比正在处理的事件高，则在**__y_isr_tick** 中断返回时立即切入，否则在就绪队列中等候。

y_timer_sync 函数是闹钟同步的核心函数，在 djyos.c 文件中定义，函数原型是：

uint32_t y_timer_sync(uint32_t u32l_ticks);

参数 u32l_ticks 的含义是事件暂停处理的 ticks 数，函数的返回值是实际暂停的时间，该

时间不会比 u32l_ticks 短，但有可能比 u32l_ticks 长，这取决于软硬件平台设计。判断闹铃时间是否到是在**__y_isr_tick** 函数中进行的，如果平台设计能保证 u32g_os_ticks 变量加 1 与调用**__y_isr_tick** 函数的次数一一对应的话，暂停时间就严格等于 u32l_ticks；如果 u32g_os_ticks 由硬件自由增量，而闹铃时间到以后因关实时中断或者关异步信号(即关调度)又或者其他中断正在服务不能嵌套中断，而导致**__y_isr_tick** 函数不能进入且持续时间超过 1 个 ticks 的话，将导致暂停时间大于 u32l_ticks。前面所述的是以 ticks 数计量的暂停时间，而实际暂停时间呢？只要在闹铃时间到后因故不能进入**__y_isr_tick**，就会导致暂停时间越限。一如 djyos 的传统，该函数没有用事件指针为参数，也就是说，只能把正在处理的事件进入闹钟同步状态。

代码 5-2 y_timer_sync 函数

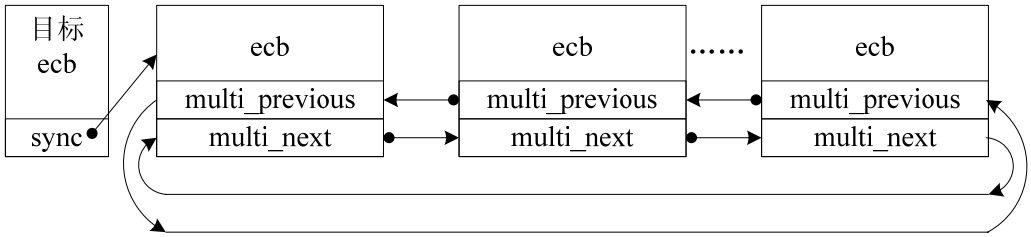
```
uint32_t y_timer_sync(uint32_t u32l_ticks)
{
    struct  event_script * event;
    uint32_t start;

    if( !y_query_sch())
    {   //禁止调度，不能进入闹钟同步状态。
        y_error_login(en_knl_cant_sched,NULL);
        return 0;
    }
    int_save_asyn_signal();
    //延时量为 0 的算法:就绪队列中有同优先级的，把本事件放到轮转最后一个，
    if(u32l_ticks == 0)
    {
        if((pg_event_running->prio == pg_event_running->next->prio)
                    && (pg_event_running != pg_event_running->next)   )
        {
            pg_event_running->delay_start = y_get_time(); //设定闹铃的时间    1
            __y_cut_ready_event(pg_event_running);       //从同步队列取出
            __y_event_ready(pg_event_running);             //放回同步队列尾部
        }else
        {
            int_restore_asyn_signal();
            return 0;   //延时量为 0，且不符合轮转条件
        }
    }else
    {
        pg_event_running->delay_start = y_get_time();   //设定闹铃的时间   1
        pg_event_running->delay_end = pg_event_running->delay_start
                    + (u32l_ticks + cn_tick_ms -1)/cn_tick_ms;  //闹铃时间

        __y_cut_ready_event(pg_event_running);

        pg_event_running->last_status.all = pg_event_running->event_status.all;
```

```
        pg_event_running->event_status.bit.event_delay=1;
        if(pg_event_delay==NULL)    //闹钟同步队列空
        {
            pg_event_running->next = pg_event_running;
            pg_event_running->previous = pg_event_running;
            pg_event_delay=pg_event_running;
        }else
        {
            event = pg_event_delay;
            start=pg_event_running->delay_start;
            do          //本循环找到第一个闹铃时间晚于新事件的事件.
            {
                if((event->delay_end - start)                    //2
                            <= (pg_event_running->delay_end - start))
                    event = event->next;
                else
                    break;
            }while(event != pg_event_delay);
            //下面把新事件插入前述找到的事件前面，如没有找到，则 event 将等于
            //pg_event_delay，双向循环队列，pg_event_delay 前面也就刚好是队列尾。
            pg_event_running->next = event;
            pg_event_running->previous = event->previous;
            event->previous->next = pg_event_running;
            event->previous = pg_event_running;
            if(pg_event_delay->delay_end -start
                    >pg_event_running->delay_end-start)
                //新事件延时小于原队列中的最小延时.
                pg_event_delay = pg_event_running;
        }
    }
    int_restore_asyn_signal();
    return y_get_time() - pg_event_running->delay_start;
}
```

对照 图 5-1，上述代码很容易理解，这里讲一下注释（2）处比较闹铃时间的代码，u32g_os_ticks是一个 32 位变量，运行到 4,294,967,295 后将环绕回 0。如果直接比较delay_end，则在一个发生环绕而另一个不发生环绕的话，比较结果将是错误的，用闹铃时间以无符号数减法减去当前时间，将得出从当前时间为基准的闹铃时间，从注释（1）处代码可知，当前时间就是pg_event_running->delay_start。如果时钟嘀嗒设为 1mS，那么 32 位变量能提供的最长定时是 49.7 天，djyos不能提供超过 49.7 天的延时。

5.2 事件同步

事件同步指当前正在处理的事件暂停处理，待指定的事件处理完成后再继续处理。同步中的事件组成一个双向循环链表，目标事件控制块的sync成员指向队列头部，如 图 5-2 所示。在传送带控制器中可以使用事件同步机制，传送带同时只能传送 1 个工件，负责放置工件的模块把工件放上传送带后，就弹出“工件已经放置”事件，然后调用y_event_sync函数，使自己与该事件同步。当“工件已经放置”事件处理完成后，负责放置工件的模块继续执行，安放下一个工件。



注：ecb是事件控制块

图 5-2 事件同步队列

y_event_sync 函数用于设定事件同步，其原型是：

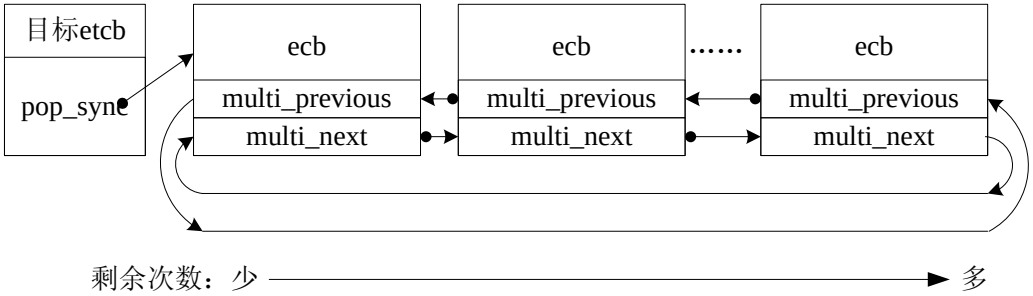
uint32_t y_event_sync(uint16_t event_id,uint32_t timeout);

函数把当前正在处理的事件加入到 id 号为 event_id 的事件同步队列中，目标事件执行 y_event_done 函数意味着同步条件达成，事件被恢复到就绪队列中。

timeout参数指定超时值，参见第 5.8 节。

5.3 事件类型弹出同步

事件类型弹出同步要求正在处理的事件暂停处理，待指定类型的事件弹出次数达到设定次数后继续处理。同步中的事件组成一个双向循环队列，该队列以剩余次数排序。目标事件类型控制块的pop_sync成员指向队列头部，如 图 5-3 所示。事件类型弹出同步应用在自动控制应用中，钻孔机每次可以钻 5 块板，每放进 1 块板，检测模块就弹出一条“放入板件”事件，使用弹出同步功能，钻孔事件每完成一次钻孔，就把自己放到放入板件事件类型的弹出同步队列中，只要把弹出次数设为 5，就可以很好地完成控制任务。



注：etcb是事件类型控制块

图 5-3 事件类型弹出同步队列

y_evtt_pop_sync 函数用于设定事件类型弹出同步，函数原型是：

uint32_t y_evtt_pop_sync(uint16_t evtt_id,uint16_t pop_times,uint32_t timeout);

函数把当前正在处理的事件加入到类型为 evtt_id 的事件类型弹出同步队列中，设定的弹出次数 pop_times 保存到事件控制块的 sync_counter 成员中。每调用 y_event_pop 函数弹出目标类型的事件一次，弹出同步队列中所有事件的 sync_counter 就减 1，在减 1 之前 sync_counter==1 表示相应事件的同步条件已经达成，事件被恢复到就绪队列中。

timeout参数指定超时值，参见第 5.8 节。

5.4 事件类型完成同步

事件类型完成同步把正在处理的事件暂停处理，待指定类型的事件完成次数达到设定次数后继续处理。设定次数的方法与弹出同步略有不同，弹出同步不允许设 0 次，完成同步设定为 0 次则有特殊含义，它表示不管完成多少次，只要最后一条事件被处理完就表示条件达成。如果目标事件类型的初始事件数（即设定完成同步那一刻的事件数）为 0，并不是表示同步条件立即达成，仍然要等至少处理完一条事件后才达成。完成同步队列和弹出同步队列完全一样，由目标事件类型控制块的done_sync成员指向队列头部，如 图 5-3 所示。事件类型完成同步可以用在废料清理中，假设每加工一个零件就会产生一些废料，需要启动废料清理模块来清理，但是，如果连续加工（未完成的零件数总大于 0），则每隔 30 分钟清理一次就可以了。设计一个加工零件事件类型和清理废料事件类型，把清理废料事件每清理完一次废料就把自己放在加工零件事件类型的完成同步队列中，完成次数设为 0，并且启动 30 分钟同步超时（同步超时见第 5.8 节）。如果连续加工，则总有加工零件事件在处理，队列中剩余事件总不等于 0，清理废料事件总不能启动，直到 30 分钟超时到。就可以达到这样的效果：加工工作一停歇，就清理废料一次，如果连续加工，则每 30 分钟清理一次废料，每清理一次废料，30 分钟便重新计算。

timeout 参数指定超时值，即事件被阻塞达到 timeout 值（ticks 数）同步条件仍然不成立的话，事件将强行结束阻塞状态。

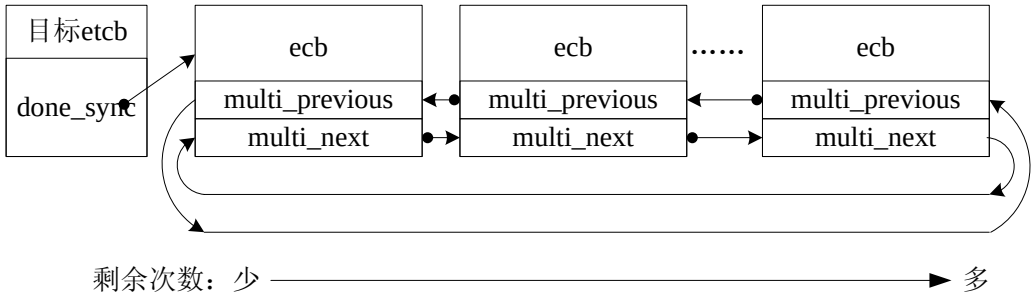


图 5-4 事件类型完成同步队列

y_evtt_done_sync 函数用于设定事件类型弹出同步，函数原型是：

uint32_t y_evtt_done_sync(uint16_t evtt_id,uint16_t done_times,uint32_t timeout);

函数把当前正在处理的事件加入到类型为 evtt_id 的事件类型完成同步队列中，设定的完成次数 done_times 保存到事件控制块的 sync_counter 成员中。完成同步和弹出同步都使用 sync_counter 成员，是因为两种同步不可能同时使用，djyos 不提供一个事件控制其他事件的功能，任何事件只有自己能使自己进入任何同步队列，而进入任意一个同步队列以后，本事件停止处理，也就不可能使自己进入其他同步队列。目标类型的任意事件调用 y_event_done 函数一次，其完成同步队列中所有事件的 sync_counter 就减 1，减 1 之前 sync_counter==1 表示相应事件的同步条件已经达成，事件被恢复到就绪队列中。sync_counter==0 则表示同步条件是目标类型的所有事件都处理完。

timeout参数指定超时值，参见第 5.8 节。

5.5 异步信号同步

异步信号（参见 第 6 章）是中断的一种，异步信号同步是说，当前正在处理的事件暂停处理，待设定的中断发生后继续处理。既然异步事件是事件的一种，一个中断源就代表一个异步事件的事件类型，与事件类型同步一样，异步信号同步对应的是异步事件类型同步。这与传统的“线程中等待信号量（或互斥量）——中断中释放信号量（或互斥量）”的方法不同，异步信号同步根本无需中断ISR配合，它完全是事件自主行为。这与传统方法相比，为系统工程师划分组件和分配任务提供了更强有力的支持，一般来说，中断ISR与硬件关系密切，往往会划入与硬件相关的组件里面（下称硬件组件），任务被划分给该组件的工程师完成；而事件处理却可能在另一个组件（下称事件组件）里面，可能会划分给另外一些工程师完成。用传统的方法，事件组件的工程师创建了信号量，获得了信号量句柄，这个句柄要么是全局变量使硬件组件可以访问，这要求两个组件在一起编译才可以，要么用其他方式传送给硬件组件，总之两个模块脱不了关系；用异步信号同步的方法，两个组件几乎完全没有耦合（参见第 11.1 节），两个组件甚至可以独立编译连接成可执行的二进制代码（不是目标文件，目标文件还需要放在一起连接才能成为可执行文件），两组工程师可以完全独立第完成自己的工作。

在 int.c 文件中的 int_asyn_signal_sync 函数把正在运行的事件挂在异步信号线 ufl_line 控制块的 sync_event 指针下，函数原型是：

bool_t int_asyn_signal_sync(ufast_t ufl_line);

ufl_line 必须被设置为异步信号，实时中断不能够被事件同步，而且，该中断线被事件同步后，同步条件达成前，该中断线不能被中心设置为实时中断。对应的中断线只要被允许，当中断发生后，同步条件就算达成，与该中断线是否与 ISR 程序连接无关，程序员可以根本不为该中断线准备 ISR 函数。同步条件达成后，目标事件将被加入就绪队列，如果其优先级足够高，则中断引擎返回主程序时，将直接返回到目标事件而不是被中断的事件。

与其他同步方法不同，异步信号同步每条异步信号中断线在同一时间只接收一个事件同步，而不是用队列的方式接收多个事件同步；也不象事件类型弹出同步一样，接受一个计数值，事件弹出若干次才算达成同步条件，异步信号一旦发生同步条件就算达成；不能设定超时，如果异步信号永不发生，将永远等待下去。这主要考虑到要简化异步信号引擎（参见 第 6 章），如果实现上述功能，引擎必然做得很复杂，执行时间也会很长。虽然异步信号不像实时中断那样紧急，但毕竟是在中断上下文中执行，在其中实现太复杂太消耗时间的功能是不明智的。

5.6 内存同步

参见第 7.5 节。

5.7 锁同步

5.7.1 数据结构定义

信号量和信号量同步用于确保安全访问共享资源，他们统称为锁，意即把被保护的资源放在一个带锁的箱子里，访问者需取得钥匙才能打开箱子，信号量有多把钥匙，允许多个访问者同时访问被保护的资源，而互斥量则只有一把钥匙，同一时间只允许一个用户访问被保护的资源。

信号量控制块定义如下：

```
struct semaphore_LCB
{
    struct    rsc_node node;
    uint32_t lamps_limit;
    uint32_t lamp_counter;
    uint32_t lamp_used;
    struct    event_script *semp_sync;
};
```

各成员解释如下：

node：信号量是在系统资源树中的一种资源，node 是资源结点。

lamps_limit：信号灯数量，被保护的资源可用数量的上限，0 表示不限数量。

lamp_counter：表示点亮的信号灯数量，表示该资源可用数量。

lamp_used：该信号量总计被请求的次数，溢出后将回绕到 0 重新开始计数。

semp_sync：信号量同步队列指针，申请本信号量不成功的事件将被在此排队等候信号灯被点亮。这是一个以优先级排队的队列，所以，当点亮一个信号灯时，被激活的是队列中优先级最高的事件，而不是等待时间最长的事件。

互斥量控制块的定义如下：

```
struct mutex_LCB
{
    struct    rsc_node node;
    bool_t    enable;
    ufast_t    prio_bak;
    struct    event_script *mutex_sync;
    struct    event_script *owner;
};
```

各成员解释如下：

node：互斥量是在系统资源树中的一种资源，node 是资源结点。

enable：true 表示被保护的资源可用，false 表示被保护的资源被占用。

prio_bak：如果发生优先级继承，保存拥有者的事件优先级，如果发生多次优先级继承，只保存事件的原始优先级。

mutex_sync：互斥量同步队列指针，与信号量同步指针相似。

owner：本互斥量的拥有者，即持有钥匙的事件。

锁控制块定义如下：

```
union lock_MCB
```

```
{
    struct semaphore_LCB sem;
    struct mutex_LCB    mut;
};
```

为了让信号量和互斥量共享内存池，这里使用了联合体，在嵌入式系统中，尤其是可能移植的系统中，需谨慎使用联合体，参见第 16.3 节。这里，联合体要么按信号量访问，要么让互斥量访问，不会同时既按互斥量又按信号量访问，因此是安全的。

在 port_kernel.h 文件中定义的常数

```
#define cn_locks_limit          100
```

表示程序最大允许创建的锁（含信号量和互斥量在内）的数量，这里的 100 指的是应用程序可使用的量，内核本身也要用锁，但不计算在内。在 lock.c 中定义锁控制块内存池，这个内存池是用固定块分配法分配的。

```
static struct    lock_MCB    tg_lock_mem_block[cn_locks_limit];
```

5.7.2 锁模块初始化

锁初始化的工作是，给信号量和互斥量在系统资源链表中找一个“家”和初始化锁内存池，创建两个根节点：semaphore 结点和 mutex 结点。此后创建的信号量都是 semaphore 的子结点，互斥量都是 mutex 的子结点。

锁初始化函数很不幸地被分离成两个函数：module_init_lock1 和module_init_lock2，前者完成创建根资源结点的工作，后者初始化锁内存池。这是因为，后者的工作必须在初始化固定块分配模块后才能进行，而初始化固定块分配模块（参见第 7.3 节）又要往资源队列中添加信号量结点，这必须在前者执行完后才能进行。在操作系统初始化过程中，将依次调用下列函数：

```
module_init_lock1();
module_init_memb();
module_init_lock2();
```

5.7.3 创建和删除锁

使用锁之前，必须先创建锁，锁模块提供了两套函数用于创建锁，这两组函数都在 lock.c 中，读者可参阅随书代码，由于代码比较简单，这里就不仔细说明了。

sem_create 函数：创建一个信号量，从池中分配信号量控制块。函数原型：

```
struct semaphore_LCB *sem_create(uint32_t lamps_limit, uint32_t init_lamp,char *name);
```

__sem_create_knl 函数：内核创建一个信号量，由调用者提供信号量控制块。函数原型：

```
void __sem_create_knl( struct semaphore_LCB *semp,
                       uint32_t lamps_limit,uint32_t init_lamp,char *name);
```

mutex_create 函数：创建一个互斥量，从池中分配互斥量控制块。函数原型：

```
struct mutex_LCB *mutex_create(bool_t init_lamp,char *name);
```

__mutex_create_knl 函数：内核创建一个互斥量，由调用者提供互斥量控制块。函数原型：

```
void __mutex_createe_knl( struct mutex_LCB *mutex,bool_t init_lamp,char *name);
```

为什么单独给操作系统一套函数呢？这是为了降低操作系统模块和其他模块之间的耦

合度（参见第 11.2 节），cn_locks_pools表示允许用户创建的锁数量，用户程序创建锁时，需要的锁控制块（信号量控制块或者互斥量控制块）从这个内存池中分配，如果操作系统模块从tg_lock_mem_block池中分配锁控制块，就占用了这个指标，应用程序与操作系统共享这个内存池，耦合随之形成。因此操作系统使用的锁不从这个内存池中分配，而是自己定义，在调用__sem_create_knl或__mutex_create_knl时传递地址。

删除锁用 sem_delete 和 mutex_delete 函数，不区分锁是由 sem_create 函数创建的还是由__sem_create_knl 函数创建的。这两个函数的原型如下：

bool_t sem_delete(struct semaphore_LCB *semp);

bool_t mutex_delete(struct mutex_LCB *mutex);

他们的返回值都是布尔变量，如果有人持有锁，该锁将不能删除，返回 false，否则从资源链表中删除锁结点，并释放锁控制块，返回 true。

5.7.4 请求和释放锁

请求一个信号量的流程如 图 5-5 所示，请求互斥量与此类似，不同的是互斥量要处理优先级继承（参见第 5.7.7 节）。请求信号量函数是sem_pend，函数原型为：

bool_t sem_pend(struct semaphore_LCB *semp,uint32_t timeout);

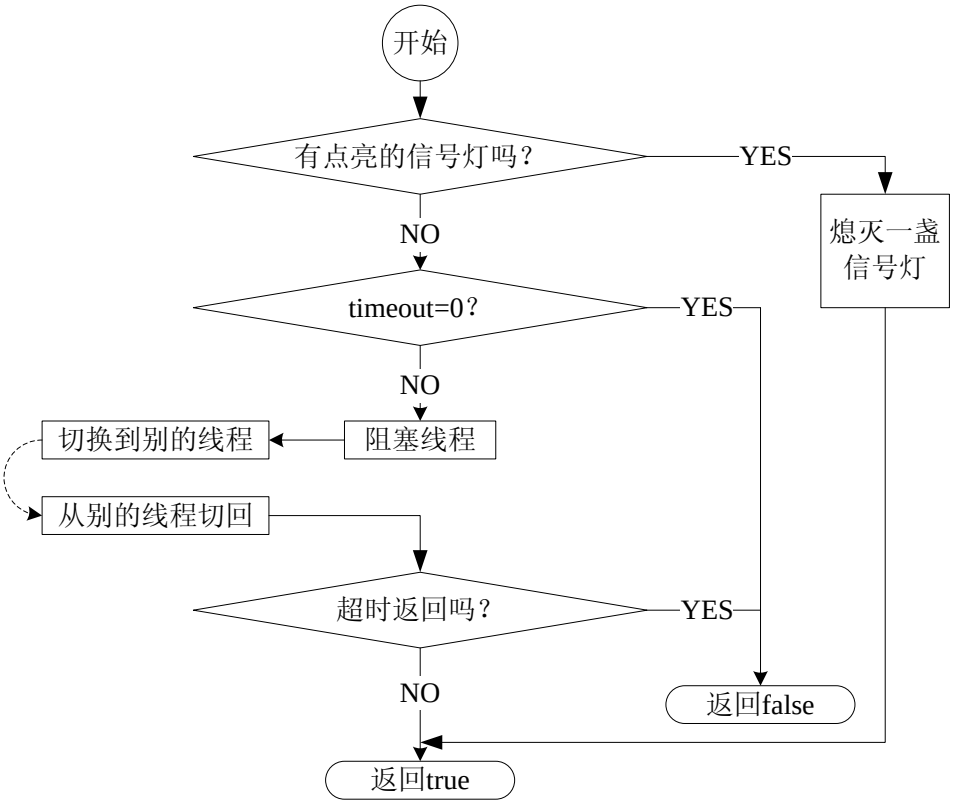


图 5-5 请求信号量

释放一个信号量的流程如 图 5-7 所示，释放互斥量与此类似，不同的是互斥量要处理优先级继承（参见第 5.7.7 节）。释放信号量函数是sem_post，函数原型为：

void sem_post(struct semaphore_LCB *semp);

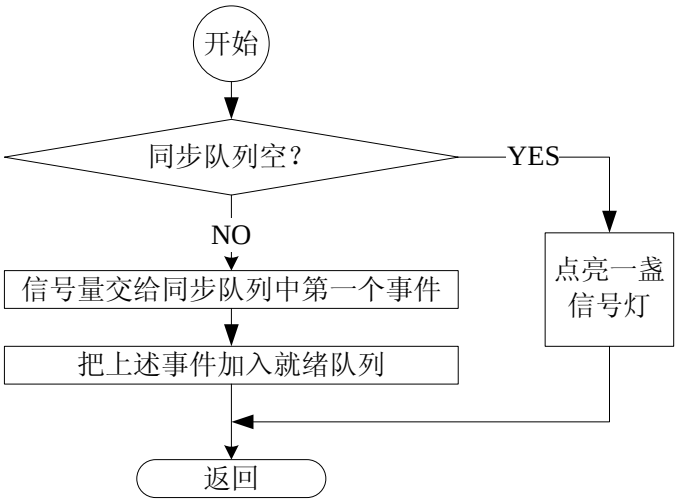


图 5-6 释放信号量

5.7.5 使用锁保护资源

图 5-7 显示的是用锁保护共享资源的示意图，如图，事件 1 的优先级高于事件 2，线程 1 处理事件 1，线程 2 处理事件 2，首先，线程 2 正在运行，在时刻A，线程 2 需要使用某共享资源，就提出申请并获得了锁，线程 2 得到锁以后开始使用共享资源。在时刻B，事件 1 就绪，线程 1 抢占了线程 2 并开始运行。在时刻C，线程 1 需要使用共享资源，它必须获得锁才能使用，但是锁被线程 2 占用，线程 1 申请不到资源，因此被阻塞，线程 2 得以继续运行。在时刻D，线程 2 使用共享资源完毕，释放了锁，锁同步管理器“看见”线程 1 正在等待该锁，就把锁交给了线程 1，线程 1 获得了锁，解除阻塞状态，因它的优先级高于线程 2，它立即抢占线程 2，进入持锁运行状态。

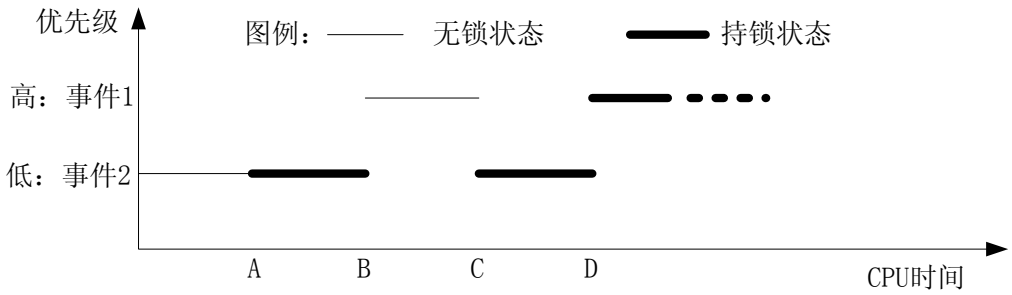


图 5-7 用锁保护资源

但 图 5-7 只是一个特例，使用的锁是互斥量或者只有 1 个信号灯的信号量的情况，而信号量可能有多个信号灯，只有在所有信号灯都被熄灭的情况下才会导致阻塞。

使用锁之前，先要创建锁，随后才能获取钥匙和释放钥匙。创建锁的过程如下（以键盘模块使用的信号量为例）：

```
static struct semaphore_LCB *pg_key_semp;      //定义信号量指针
pg_key_semp = sem_create(cn_key_buf_len,0,"read_key");   //创建信号量
```

创建信号量后，就可以用

sem_pend（pg_key_semp，0）或 sem_post（pg_key_semp）进行申请/释放信号量了。

下面以信号量为例，讲述锁的使用过程，互斥量的用法与此类似。信号量通常有 3 种用法，一是在线程内申请和释放，用来在并发访问中保证共享资源的安全；二是申请和释放分别在不同的线程中，同步资源的生产和消费行为；三是作为某资源的使用计数。

第一种用法以共享变量的读写为例，假定有一个共享变量 share，使用信号量保护并发访问 share 的过程如下：

sem_pend（……）；

访问 share；

sem_post（……）；

第二种用法以键盘模块为例，调用：

pg_key_semp = sem_create(cn_key_buf_len,0,"read_key");创建一个共有 16 盏信号灯，初始可用的信号灯为 0 的信号量。在键盘扫描线程中，执行下列伪代码过程：

1. 扫描到键值并写入键盘缓冲区；
2. 执行：sem_post（pg_key_semp）；
3. 返回 1.

在需读取按键的线程，执行下列伪代码过程：

1. 如果 sem_pend（pg_key_semp，x）==true
2. 返回读取的键值；
3. 否则：
4. 返回空键值。

第一句的意思是，如果键盘缓冲区有按键，就返回 true，否则就阻塞，若阻塞时间达到 x 毫秒按键扫描线程仍然没有执行 sem_post 的话，就返回 false；

信号量的第三种用法比较特殊，把信号灯总数设为 0，被保护的资源就没有信号灯数量限制，允许无限重并发访问，而信号量则演变为资源访问计数器，信号量控制块的 lamp_used 成员记录了该资源正在被多少个线程访问。互斥量则没有这种用法。

5.7.6 信号量和互斥量的区别

信号量和互斥量都是保护共享资源的，在使用方法上，信号量和互斥量非常相似，那么，他们有什么区别呢？

1. 信号量允许并发访问被保护的资源，并发数就是信号量控制块中的 lamps_limit 成员的值。而互斥量只允许独占访问被保护的资源。
2. 如果把信号量设置成允许无穷多重的并发访问，信号量就转化成共享资源访问计数器，它只记录被保护的共享资源正在被访问的次数。
3. 由于信号量允许并发访问，故没有“拥有者”的概念，而互斥量则有“拥有者”。
4. 信号量没有优先级继承机制，不能防止优先级反转；而互斥量则有优先级继承机制，可以防止发生优先级反转。

综上所述，信号量和互斥量的区别可如 表格 5-1 描述。

表格 5-1 信号量与互斥量的区别

	并发重数	拥有者	访问计数	优先级继承
信号量	1 至无穷多	无	有	无
互斥量	1 重，独占访问	有	无	有

5.7.7 优先级继承

互斥量同步支持优先级继承以解决优先级反转的问题。在实时内核中，优先级反转是一

个常见的问题，图 5-8 解释在不支持优先级继承的条件下，优先级反转是如何出现的。如图，事件 1 的线程 1 优先级高于事件我的线程 2，线程 2 优先级高于事件 3 的线程 3。首先，线程 3 正在运行，在A时刻，线程 3 要使用被锁保护的共享资源，就提出申请并得到了该锁。线程 3 得到了锁之后，就始使用该共享资源。在时刻B，事件 1 就绪，由于事件 1 优先级高，它就绪之后立即剥夺了线程 3 的CPU使用权，线程 1 开始运行。运行过程中线程 1 也要使用那个线程 3 正在使用着的资源，在时刻C申请保护该资源的锁，由于该资源的锁还被线程 3 占用着，线程 1 申请不成，只能进入阻塞状态，等待线程 3 释放该信号量线程 3 得以继续运行。在时刻D，事件 2 就绪，由于事件 2 的优先级高于事件 3，事件 2 就绪后线程 2 剥夺了线程 3 的CPU的使用权并开始运行。直到事件 2 处理完或因故阻塞之后，在时刻E将CPU控制权还给线程 3。线程 3 接着运行直到在时刻F释放保护那个共享资源的锁。此时，由于线程 1 在等待该锁，该锁将直接转交给线程 1，使线程 1 得到该信号量并接着运行。在这种情况下,事件 1 优先级实际上降到了事件 3 的优先级水平。因为事件 1 的线程 1 要等,直等到线程 3 释放占有的那个共享资源。由于线程 2 剥夺线程 3 的CPU使用权,使比事件 1 的优先级低的事件 2 实际上获得了比事件 1 更高的优先级，线程 2 使线程 1 增加了额外的延迟时间。事件 1 和事件 2 的优先级发生了反转。

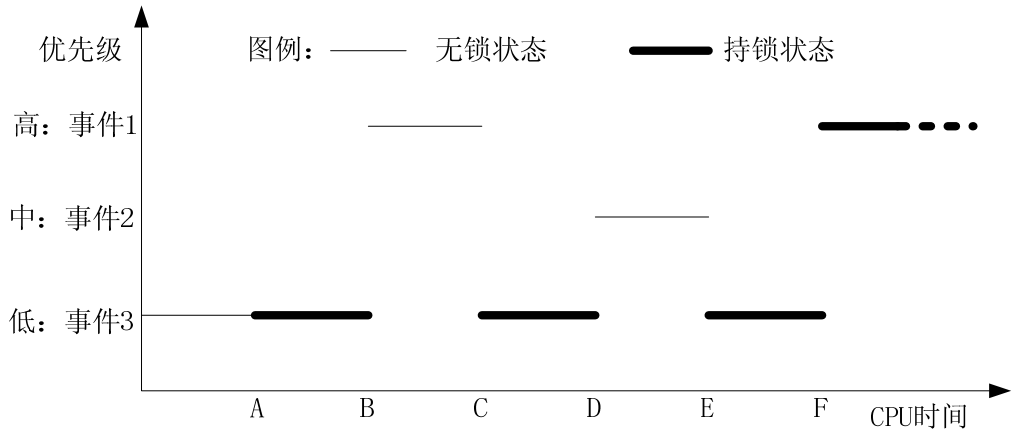


图 5-8 优先级反转

要纠正优先级反转,就必须在线程 3 使用共享资源时,提升事件 3 的优先级，线程使用完共享资源后予以恢复。事件 3 的优先级必须升至事件 1 的水平,以阻止优先级在事件 1 和事件 3 之间的线程抢占线程 3。如果在线程 3 以事件 1 的优先级处理的过程中，有比事件 1 更高优先级的事件X就绪抢占线程 3，如果事件X也申请该资源，还应该把事件 3 的优先级进一步提升到事件X相同。上述过程叫做优先级继承，事件 3 先继承事件 1 的优先级，再继承了事件X的优先级。djyos的锁有两种，其中信号量是不支持优先级继承的，而互斥量则支持优先级继承。图 5-9 说明了互斥量是如何实现优先级继承的，每个互斥量都有一个struct event_script类型的owner指针成员，该成员记录着互斥量正在被哪个事件占用。如果某线程申请互斥量时，因互斥量被低优先级事件占用而不得，操作系统就会考虑优先级调整。调整之前，需要检查拥有者是否在就绪状态，若否，即使调整了优先级也可能没有用，因为谁都不知道阻塞该事件的条件什么时候才能达成，因而不做优先级继承。若该事件在就绪态，则调整它的优先级，在调整之前，先要备份它的优先级，已备该事件使用完共享资源后恢复优先级。在备份之前，当然要判断一下该优先级是否已经备份，以免在多次继承中丢失原优先级。代码 5-3 则是mutex_pend和mutex_post函数中优先级继承部分代码。

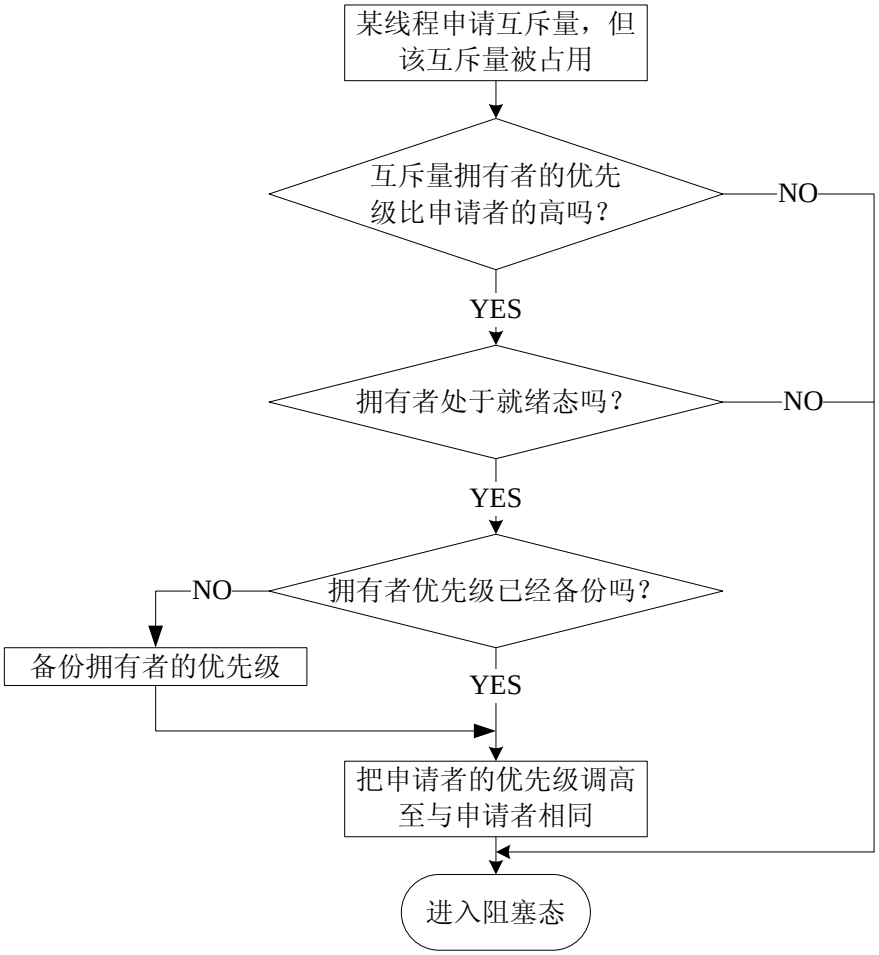


图 5-9 互斥量优先级继承

代码 5-3 请求和释放互斥量时优先级继承部分代码

```
bool_t mutex_pend(struct mutex_LCB *mutex,uint32_t timeout)
{
    ……;                    //省略部分代码
    //下面看看是否要做优先级继承
    pl_ecb = mutex->owner;
    if(pl_ecb->prio > pg_event_running->prio) //需要继承优先级
    {//1、处于就绪态，2、处于某种阻塞态。
        if(pl_ecb->event_status.bit.event_ready ==1)//占用互斥量的事件处于就绪态
        {
            __y_cut_ready_event(pl_ecb);
            //prio_bak 应该保留事件的原始优先级，防止出现多次继承中丢失原优先级
            if(mutex->prio_bak != cn_prio_invalid)
                mutex->prio_bak = pl_ecb->prio;
            pl_ecb->prio = pg_event_running->prio;
            __y_event_ready(pl_ecb);
        }else           //占用互斥量的事件处于某种阻塞态，暂不处理
        {
        }
    }
```

```
    ……            //省略的代码
}
void mutex_post(struct mutex_LCB *mutex)
{
    ……            //省略的代码
    if(mutex->prio_bak != cn_prio_invalid)   //该互斥量发生了优先级继承
    {
        __y_cut_ready_event(pg_event_running);       //取出 running 事件
        pg_event_running->prio = mutex->prio_bak;     //恢复优先级
        mutex->prio_bak = cn_prio_invalid;
        __y_event_ready(pg_event_running);     //重新把 running 插入就绪队列(新位置)
    }
    ……            //省略的代码
}
```

5.8 阻塞超时

除闹钟同步和异步信号同步外，其他同步方式都可以设定一个超时参数：timeout，它表示：线程被阻塞时间达到 timeout 将强制退阻塞状态，timeout 的单位是 mS，定时精度是 1 个 ticks，执行时 timeout 会被向上取整数个 ticks。

timeout 是一个 32 位无符号数，如果取 0，则表示如果同步条件不成立也立即返回 false，如果取 cn_timeout_forever，则表示如果同步条件一直不成立则无限期等待。

当调用会阻塞的函数，比如获取信号量，如果同步条件不成立（信号量不可用），且如果timeout不为 0，也不是cn_timeout_forever，则要执行阻塞超时等待，如 图 5-10 所示。

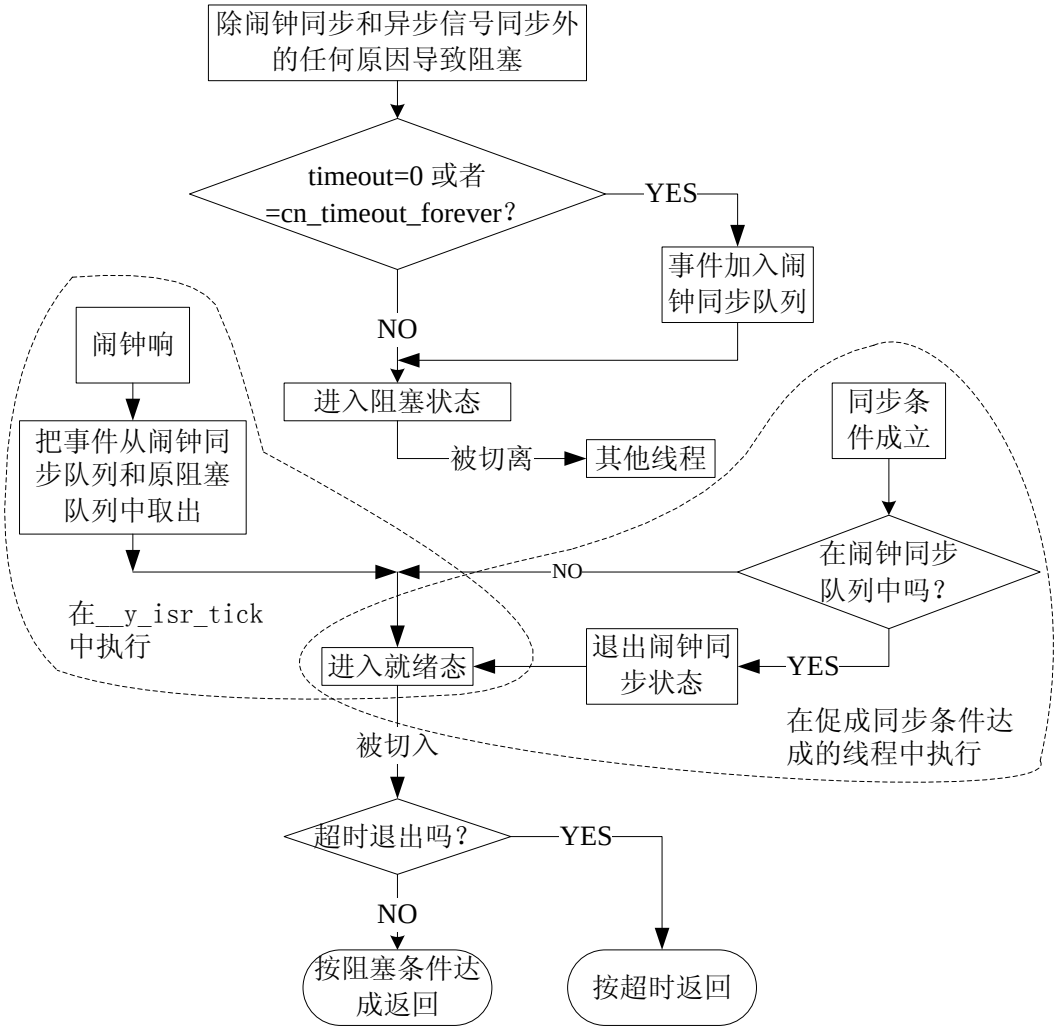


图 5-10 阻塞超时流程

事件进入阻塞状态之前，将被置入闹钟同步队列中。事件被切离后，闹铃时间到或者同步条件达成（比如有信号灯被点亮），都将导致该事件被加入就绪队列，继而被切入CPU。图中在__y_isr_tick函数中执行的部分参见 代码 5-1 的注释 3。

第6章 中断

6.1 中断与硬件设备

在面向对象的软硬件联合设计思想中，中断处理程序硬件操作函数既不是独立的软件模块，也不是设备驱动的一部分，而是以硬件为主体构成的功能模块延伸到 CPU 内部的一部分。

从软件构建的角度看，中断并无特殊之处，大多数只是异步通知的事件而已，可以纳入操作系统事件调度的轨道统一调度，只有少量中断是需要紧急处理的。

软硬件协作模块指的是跟软件和硬件都有联系的模块，即 图 1-4 中的B部分。这部分模块可能用纯硬件实现，也可能用纯软件实现，但最常见的是情况是硬件结合软件的方式实现，本节讲述的是用软硬件结合实现的情况。

6.1.1 硬件操作

本节提到的泛设备概念，将在 第 9 章讲述，这里只要知道他是工作在djyos操作系统之上的设备驱动程序就可以了。如果阅读有困难，读者可以先粗略浏览本章，待阅读完 第 9 章后，再回头细读。

有许多嵌入式系统被设计成“激励——响应”系统，而“激励”经常由中断输入，这样的系统的许多功能与中断相关，甚至会有一些实时性要求非常高的模块，系统设计时，能否把中断规划好，是项目完成质量的关键。

中断和中断响应函数是什么？硬件操作函数又是什么？他们应该如何组织，应该由谁开发，应该归属于哪一个模块？中断响应函数、硬件操作函数和泛设备 driver 以及使用设备的模块之间究竟是什么关系？带着这些问题，我们开始本章的讨论。本章不讲述中断产生的硬件机理，也不讲述 CPU 响应中断的过程，也不涉及硬件 IO 实现的机理，这不是本书的主题，有兴趣的读者可以自行找相关书籍了解。

本章后面将多次用到“硬件操作程序”这个概念，“硬件操作程序”包含以下三种操作之一种或几种。

1. CPU 芯片内部和外部产生的中断的响应函数，包括用查询方式响应的中断信号。
2. IO 读写操作，包括 CPU 自身提供的 IO 和片外扩展的 IO。
3. 硬件寄存器读写操作，包括 CPU 片内外设的寄存器和片外扩展的外设的寄存器。

毋庸置疑，中断必定跟某些硬件模块相关联，一个硬件模块与 CPU 交互，可以不使用中断，也可以使用一个或多个中断。通常，硬件模块需要通过泛设备 driver 接口与其他模块（一般是软件模块）交互工作。那么，“硬件操作程序”与泛设备 driver 究竟是什么关系呢？最为迷茫的莫过于和硬件接口的泛设备 driver 的作者了，既然叫设备 driver，那么与被驱动的硬件模块紧密结合的中断，总不能不闻不问吧，总得做点什么吧！可是能做什么呢，中断信号如闲云野鹤，来去无踪；CPU 响应中断后，会自动切换到特权级别，中断服务函数可以无法无天，根本不听使唤，所以，设备 driver 根本没有办法控制中断函数。另外，由于

djyos 的泛设备 driver 接口并不是专门为硬件操作准备的，控制硬件才需要的“硬件操作程序”究竟放在哪里呢？既然如此迷茫，我们为什么不反过来问问，虽然中断是由硬件模块产生的，泛设备 driver 就必须处理中断吗？“硬件操作程序”与泛设备 driver 有直接而且必然的联系吗？硬件的设备 driver 和对应设备的“硬件操作程序”必须由同一个人编写吗？从系统设计角度，djyos 系统认为，中断处理程序应该是和产生中断的硬件模块绑定在一起的，它和泛设备 driver 之间并没有直接的联系。除中断以外，“硬件操作程序”的其余部分与泛设备 driver 之间也是这种关系。“硬件操作程序”和相应的硬件模块共同组成一个软硬件协同模块，他们是一个整体。该模块的纯硬件部分和纯软件部分都和泛设备 driver 没有直接的关系，软硬件协作模块是作为一个整体与泛设备 driver 打交道的。虽然在大多数情况下，硬件设备 driver 与该硬件相对应的“硬件操作程序”是由同一个人或同一个团队开发的，但并非必须如此。泛设备 driver 只需给出明确的操作接口，软硬件协作模块作为实体与泛设备 driver 的操作接口（一般是右手接口）打交道，而泛设备 driver 完全不知道是其他软件模块在操作设备还是硬件控制函数在操作设备！泛设备 driver 很可能会调用“硬件操作程序”，这取决于如何实现模块。

要分析中断处理函数、硬件模块、硬件操作函数、泛设备driver以及其他软件模块的关系，就不得不提一下面向对象的设计方法。在 第 1 章中，我们看到，软硬件其实并没有明确的区分，是的，系统工程师在设计之初只需要指出“我需要某某功能模块”，软硬件协作模块的设计目标就是让设计者能够灵活选择用软件实现还是用硬件实现这个功能模块，这种可选择性可以在产品性能、产品生产成本和软件复杂度之间求取平衡点，大多数时候起决定作用的往往是设计师的习惯。面向对象的设计方法常常被片面地理解，很多人认为面向对象的方法仅仅适用于软件设计领域，更不幸的是，有些工程师甚至把面向对象的方法于面向对象的设计工具联系起来，好像只有用C++等面向对象的编程工具才能编写面向对象的程序似的，甚至有人专门研究用C语言按面向对象的格式书写代码的技巧。而在嵌入式领域在过去、现在以及以后相当长的时间内，主流编程语言是C语言而不是C++语言，遑论JAVA之类更抽象的面向对象语言了，但并不能因此而限制在嵌入式系统设计中应用面向对象的设计思想。就像大多数嵌入式系统一样，本书使用的所有范例都是用C编写的，至于为什么嵌入式多选用C而不是C++，已经超出本书讨论的范围了。面向对象是一种设计思想，这种设计思想可以贯穿从系统规划到系统总体设计到详细设计再到软件编码和硬件设计的全过程。我们完全可以在面向对象的设计思想指导下，用面向过程的工具，做出看起来象结构化的作品，而内涵却是对象化的。在系统规划阶段时，设计者面对的是宏观系统结构，这时候，与面向对象的编程工具没有直接的关系；当系统规划完成以后，大局已定，至于选择用C语言按面向过程的格式书写代码，还是用C++或者其他语言按面向对象的格式书写代码，并不是那么重要了，再强调一下，这里说的是按照面向对象的方法思考，而不是按面向对象的格式编写代码，非面向对象的设计语言一样可以贯彻面向对象的设计思想，选择编程语言取决于企业传统、你的习惯和你手头的开发工具。

在如 图 6-1 所示的系统结构中，如果按面向对象的角度，我们可以看到一个由软件和硬件共同组成的功能模块，该模块的功能不仅取决于硬件，还取决于软件。这个功能模块就是一个对象，系统设计阶段只需要设计这个对象的外特性，至于如何实现，选择软件实现还是硬件实现，如何平衡软硬件的比例，是下一步系统实现阶段的任务（参见第 1.3 节）。你可以创建一个设备来让这个模块与其他模块交换数据，或者控制硬件的行为。其他软件模块看见的只是泛设备driver，根本不知道该功能模块是用软件还是硬件实现的。如果把A模块中处于CPU内部的软件部分划入泛设备driver中，该设备driver立即就变成依赖于特定硬件的软件模块，即使用C++按面向对象的格式书写，也是个面向过程（A模块的硬件实现过程）的程序。丧失了在不同的硬件配置中的可移植性。而且，在系统的角度看，A模块也不再是

一个完整的面向对象的功能组件，而是一个随硬件修改而变化的面向过程的模块。一个企业稳定发展 3、5 年以后就会知道，由于产业发展、半导体工艺进步、原厂产品升级换代而停产、企业采购渠道的变化等因素，嵌入式电子产品的硬件模块也会变化，如果driver代码与特定硬件相关，就会从根本上影响企业软件体系的稳定。

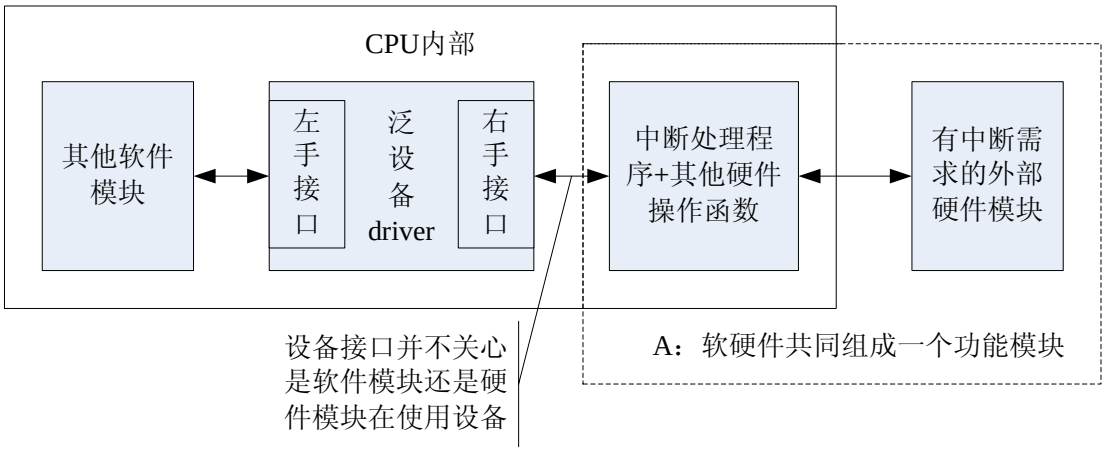


图 6-1 典型的软硬件模块接口

6.1.2 中断只是异步事件

事件（消息）触发是常见的一种软件策略，从广义上讲，所有软件都是按事件触发的方式工作的，例如一个条件语句，

```
if （conditional expression == true）
    子句 1；
else
    子句 2；
```

这是一种隐式的事件触发，我们可以理解为发生使条件为真的事件后，触发“子句 1”，当发生使条件为假的事件后，触发“子句 2”。显式的事件触发是，当检测到某事件发生时就弹出事件，明确地向操作系统报告“发生了某某事件，需要处理”，操作系统内核收到报告后，就会依据调度策略，在适当的时间调度分配现有的线程虚拟机或者创建新线程虚拟机处理该事件。在整个过程中，操作系统始终控制这程序的执行流向，事件的发出和处理和操作系统步调一致，可称之为同步事件，因“同步事件”容易与“事件同步”中的被同步事件混淆，下文中称“同步事件”为普通事件。

同样，中断信号通知系统发生了一个突发事件，与普通事件一样，都代表“发生了某某事，需要处理”，与普通事件不一样的是，中断是突然发生的，程序员不知道什么时候会发生中断，也不知道会发生什么中断，只要中断被允许，它就肯定会切换到中断上下文。从中断引脚发出通知，到切换到中断上下文的整个过程，操作系统完全不知情、不可控，因此，中断信号应该称作异步事件通知器。

中断信号是与突发事件相联系的，但突发事件不等于紧急事件，在 3.1 中告诉我们，紧急事件应该被赋予较高的优先级，在系统轻载时，紧急事件指实时性要求很高的事件，在系统过载的情况下，最重要的事件将上升为紧急事件。而嵌入式实时系统一般被设计成不会过载，其紧急程度一般从被通知的事件的实时性需求来判断。虽然系统设计中经常把紧急事件用中断的方式通知CPU，但并不总是这样，系统中有独立的标准确定事件的紧急程度，该标准与事件是否由中断信号通知并无关系。例如一个串口接收中断事件，如果没有硬件缓冲区，就应当作为紧急事件，如果有较大的缓冲区，则无需作为紧急事件。所以，中断仅仅是通知

发生异步事件的信号而已，异步事件可以是紧急事件，也可以非紧急事件。那种认为凡是中断服务就应该快速响应、快速完成的观点，是没有根据的。

进一步引申，对于直接连接到 CPU 的 IO 口的外部信号，虽然他们并没有连接到中断线上，但他们跟中断信号一样，同样是属于异步事件的一种，这种异步事件可以通过软件周期性查询来转化成普通事件。事实上，即使是连接在中断线上的异步信号，同样可以周期性地查询中断悬挂寄存器，把它转化成普通事件。

中断信号以及其关联的突发事件的软件模型，不外乎以下四种情况：

A. 紧急异步事件，该事件应该在严格的时限内完成。比如煤气告警器响了，无论你正在干什么，都要立即关闭煤气，以避免发生爆炸，并快速找到漏气的原因并修复之，否则烧水做饭取暖全完了。

B. 复合异步事件，邮递员给你送来一封挂号信，你需要立即签收，信里要求你准备许多材料，下月 10 日到北京开会。这类事件有一小部分操作是要求在严格的时限内完成的（邮递员还要送别的信，不能一直等你），但是会伴生大量的、紧急程度并不高的普通工作，你可以在合适的时间完成它就可以了。典型的通信中断，为了不致硬件缓冲区溢出，你需要在中断函数中迅速把数据从硬件缓冲区读到软件缓冲区，余下的工作可以慢慢完成，其优先级甚至低于普通事件。

C. 普通异步事件，这类事件虽然是由中断通知的突发事件，但其紧急程度可能很低，用中断实现往往只是为了避免周期性查询的开销，仅起异步通知器的作用。例如带缓冲区的键盘中断，CPU 对键盘操作的反应速度只需要与人按键操作的速度匹配就可以了，对快速的 CPU 来说，是几乎没有什么实时性可言的操作。

操作系统中，管理中断服务函数是一项极其挑战的任务，其复杂性来自于两个方面：异步性和高优先级。

1. 由于异步性，操作系统也好，应用程序也好，都完全不知道什么时候会发生中断，什么时候会进入中断服务函数。这对保护共享资源带来了挑战，要么在使用共享资源时关闭中断，要么不允许在中断函数中使用被保护的共享资源。在许多操作系统中，malloc 族函数以及其他使用共享资源的函数是不允许在中断函数中使用的。printf 等不可重入函数也是不允许在中断函数中使用。
2. 只要被允许，中断可以打断任意线程，也可以打断操作系统内核，中断服务函数由硬件直接调用的，其他程序根本无从控制。操作系统仅剩一点的微薄的控制能力，也就是关中断，让你闭嘴！——这简直是一种暴力行为，暴力不仅仅属于政治，也属于中断！如果中断函数被阻塞，操作系统将没有办法把上下文切换到其他任务，这样将导致整个系统死锁。因此，中断处理函数是不允许阻塞的。

既然中断函数如此难于驾驭，那么，最好的办法就是尽量减少中断服务函数的工作，80-20 原则在这里是适用的，也就是说，由中断异步通知的事件中，真正需要在严酷的时限内完成的工作，其实是很少的，大量的工作应该是普通工作，这些工作的优先级并不一定很高甚至低于普通实时事件。如果大量的、实时性要求不高的普通任务都在中断上下文中完成的话，就可能造成被中断的事件的实时性得不到满足。由于中断的随机性，被中断的事件的优先级可能很高，而如果在高优先级的事件中禁止中断，又可能导致错过紧急事件。为了解决这个矛盾，就必须让处理突发事件的程序按突发事件的实际实时性需求工作，实时性高的突发事件具有高优先级，实时性低的突发事件只有低优先级，使实时性需求高的事件不受或少受由中断触发的实时性需求低的突发事件影响。

既然这样，那么区别紧急异步事件和非紧急异步事件，为中断事件选择合适的优先级有重大意义，它牵涉到系统稳定性的问题。绝大多数情况下，CPU 的大部分负荷是周期性的，可以在开发初期预测 CPU 是否会过载，也可以通过测试确定是否过载。而中断事件是突发

性的，用户的误操作或者硬件故障可能会产生过多的中断事件而导致 CPU 过载，如果这些中断事件无论是否紧急异步事件，都以很高的优先级（中断的优先级高于所有事件优先级）运行的话，将导致被中断的、实际优先级很高的事件得不到 CPU。

6.2 中断的软件模型

6.2.1 实时中断与异步事件

djyos 系统设计中断管理模块时也体现了系统的“九九加一”原则：

1. 日常大量存在的、实时性并不是特别高的工作，系统提供最大的便利，让程序员能够简洁地实现。
2. 极少遇到的、高难度的、甚至挑战系统实时性和处理能力极限的工作，系统提供最大的灵活性，使问题的解决成为可能。

基于上述原则，在 djyos 系统中，中断被分为两大类，第一类是实时中断，对应现实世界中紧急程度非常高的中断信号，实时中断的响应与前后台系统无异，具有接近前后台系统的实时性，操作系统运行过程中，调度程序永远不会关闭实时中断，只是提供一个接口函数，使线程可以根据需要临时关闭实时中断。在实时中断里，程序员能够像在前后台系统中一样自由自在地编程，除了不能使用操作系统的系统调用外，没有太多的束缚，相应地，操作系统为实时中断提供的服务也最少。实时中断为用户的紧急突发事件提供绿色通道，它的实时性能要远高于其他操作系统的中断体系。

第二类是异步信号，对应紧急程度不是很高的中断信号，虽然异步信号的优先级高于所有普通事件，但在优先级上与他们并没有实质性的差别，djyos 的中断管理把他们等同看待。但因其异步性，与普通事件还是有区别的，他们的相同点表现在：

1. 禁止事件调度和禁止异步事件切换是等同的，也就是说，当禁止调度时，djyos 把异步信号当作事件一样也禁止了。
2. 异步信号 ISR 可以从堆中分配内存，可以释放和申请信号量，以及使用其他临界资源。
3. 异步信号 ISR 可以使用几乎所有的系统服务。

异步事件和普通事件的不同点表现在：

1. 异步信号没有独立的上下文，故 ISR 不可以被阻塞，因此：
   a) 不可以主动同步请求，当调用 y_timer_sync、y_event_sync、y_evtt_pop_sync、y_evtt_done_sync 这四个函数时，将立即返回，不会进入同步状态。
   b) 当申请临界资源不果，比如调用 m_malloc 分配内存，如果是事件处理函数调用，当前事件将可能进入阻塞状态，而异步信号 ISR 则会立即返回 NULL。
2. 由于异步信号优先级高于所有普通事件，故其 ISR 不能做成死循环的形式。

6.2.1.1异步事件的优先级模型

异步事件和普通事件在优先权上本质是相同的，不同的只是他们具体事件个例优先级量的不同，如何确定异步事件和普通事件的优先级呢？有两种方案可供选择。

6.2.1.1.1 理想模型：优先级混合

在这个模型中，中断的优先级与事件的优先级是完全混合的，所有中断事件和普通事件按照事件的实时性需求统一排定其优先级。当一个中断信号到达时，如果正在执行的事件的优先级高于或等于该中断的优先级，则该中断被挂起，直到比它优先级高的事件都已经处理完才得到服务。同样，一个实优先级高的普通事件就绪，它可以中断正在处理的、优先级比它低的中断事件。

例如在一个简化的自动控制装置中，优先级次序如 图 6-2 所示，在混合优先级模型下，调度特点：

当设备巡检事件就绪，无论通信中断或键盘中断还是显示刷新事件正在执行，都将被打断，直到设备巡检任务完成才能继续执行。

而通信中断尽管是中断，但由于优先级低，如果正在处理高优先级的设备巡检事件，则中断信号到达时并不会立即调用其 ISR 函数。优先级更低的键盘中断就更不要说了，它得给系统中所有的事件和中断让道。

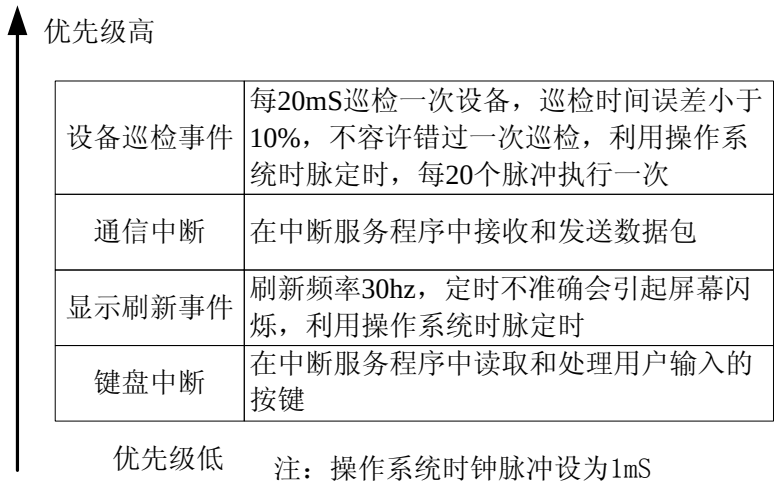


图 6-2 中断与事件优先级混合

这是一种完美的模型，但理想的东西，与现实总是有差距的，实现这样的模型，会遇到很多实际问题，而且对硬件的依赖也很强。djyos 在设计初期曾经视图实现这个模型，并且接近成功，但越接近成功，浮现出来的问题就越多，最终忍痛放弃了这个方案。

1. 中断能够被高优先级的事件打断，意味这中断 ISR 能够被阻塞，这样，ISR 就不能工作在传统的中断上下文中，而是必须使每个中断线有独立的栈空间，为其建立独立的上下文和上下文控制块，每次中断都要求执行上下文切换，将耗费许多 CPU 时间。每个中断线都会因此耗费大量的内存，这在内存比较充裕的系统还好，而靠片内继承 RAM 工作的 MCU，就望而生畏了。
2. 由于中断的异步性，调度器无法直接控制它，任何事件为了不被比它低优先级的中断打断，唯一的办法就是禁止所有比它的优先级低的中断线。如果硬件提供专门的支持，比如 cortex-m3，它可以设定允许的优先级门槛，比该门槛低的中断都被禁止。如果硬件不提供相应的支持，就的一个个地禁止比它优先级低的中断线，是一个非常耗时的工作。
3. 这个模型要求高优先级的中断能够打断低优先级的中断，反之则不行，意味着 CPU 必须提供中断嵌套机制，遗憾的是，有些 CPU 根本就不支持中断嵌套。对不支持

中断嵌套的 CPU，需要用汇编语言模拟中断返回，但依旧返回到该中断的 ISR 中，并且必须手工调整该 ISR 上下文的栈，使其返回时能够返回到被中断的事件中。这个过程非常繁琐不说，还不能确保你使用的 CPU 支持手工模拟中断返回。

4. 要确保高优先级的中断不被低优先级的中断打断，看似是个简单的问题，遗憾的是，这实际上不简单，有些 CPU 比如 S3C44B0X 的中断控制器，只要在中断 ISR 中打开中断使能，它就允许任何中断包括比它优先级低的中断嵌套。对这种 CPU，在中断 ISR 中必须逐个禁止优先级比它低的中断，这也是一个很耗时的工作。
5. 以上所述，都与目标硬件平台密切相关，甚至相同 CPU，只要中断控制器不同，就有巨大的不同，它使移植操作系统与硬件更加紧密关联，增加了移植工作量。

6.2.1.1.2 djyos 模型：ISR 引擎

这个方案中，中断事件的优先级高于所有事件，中断ISR能抢占包括最高优先级的所有事件。但中断作为异步事件，处理该事件的绝大多数工作，并不在ISR中完成，而是作为一个普通事件，在事件的上下文中完成。操作系统要实现的是，给于中断服务函数启动或激活相应的事件的手段。用户在编写中断服务函数时，如果处理中断事件的代码很简单，就直接在ISR中完成，如果很复杂，就只完成最紧急的工作，然后激活相应的事件，余下的非紧急工作由事件处理函数完成。使用这种策略，图 6-2 所示自动控制装置的优先级模型将变成如 图 6-3 所示。

这样处理的缺点是当低优先级的突发事件到达时，仍然会短暂中断正在执行的高优先级事件，但中断时间很短；而优点也是显而易见的，它简洁明了，上下文切换时间也很短，而且 ISR 模型本身与硬件无关，便于操作系统移植。许多 RTOS 都提供在中断服务函数中唤醒睡眠状态下的线程的功能， djyos 也不例外，不同的是 djyos 程序员直接面对的是事件而不是线程，详见下一节介绍。

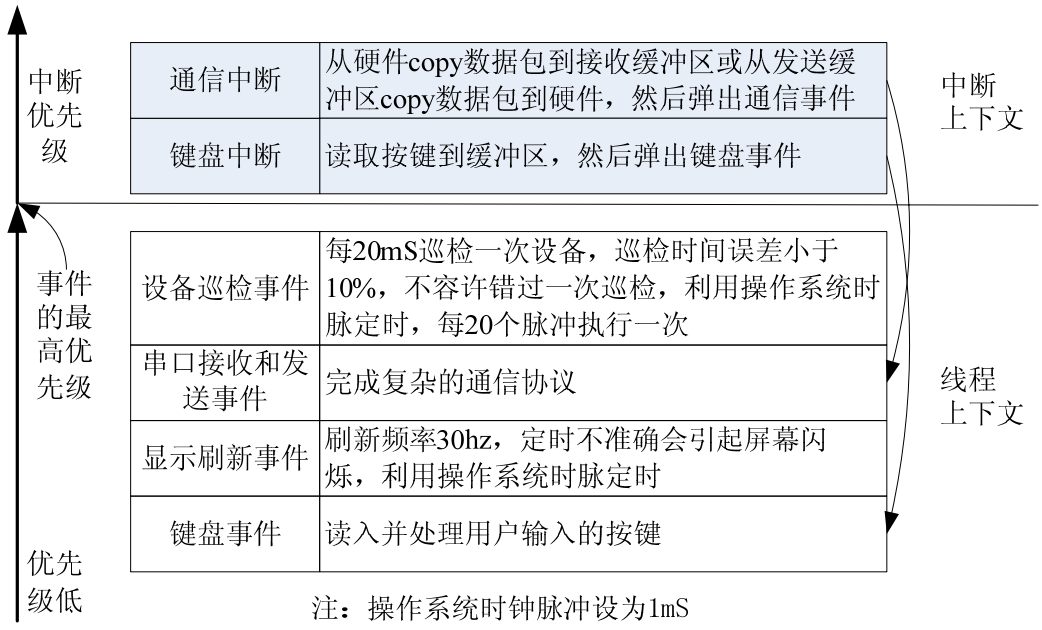


图 6-3 ISR 作为事件引擎

这种策略下，处理通信中断事件的软件模型有两种，第一种如下：

1. 通信口初始化时，登记“通信口接收”和“通信口发送”两个事件类型，并分别为此两类型事件编写事件处理函数，完成数据收发和通信协议处理。
2. 当中断到达时，将打断正在处理的事件，CPU 转而调用中断服务函数，但仅执行少量的紧急操作，然后弹出“通信口接收”或“通信口发送”事件，ISR 便返回。如果被打断的事件优先级高于或等于通信口收发事件，则继续执行，否则将立即切换到通信口收发事件。
3. 上述事件在就绪队列中等候，操作系统会在合适的时候把该事件切入，在处理事件的线程虚拟机中完成实际的数据收发工作。
4. 收发工作完成后，调用 y_event_done 退出。

第二种软件模型如下。

1. 初始化时登记“通信口接收”和“通信口发送”两个事件类型后，直接调用 y_event_pop 函数弹出事件，待启动多事件调度后，通信口收发事件将投入运行。
2. 通信口收发事件调用 int_asyn_signal_sync 函数，事件将被阻塞在“异步信号同步”状态，等待中断发生且 ISR 返回后继续执行。
3. 通信口中断发生后，将打断正在处理的事件，调用 ISR 函数，执行少量紧急操作后，直接返回，（这里少了弹出事件的步骤）。
4. 操作系统的同步机制将会把处于阻塞状态的通信口收发事件重新投入就绪队列，如果其优先级高于被中断的事件，将立即投入运行，否则就在就绪队列中等待调度器的眷顾。
5. 事件重新投入执行，完成收发工作后，函数并不返回或调用 y_event_done，而是重新调用 int_asyn_signal_sync，等待下次中断到来。

至此，我们终于揭开djyos优先级体系的全貌了，回顾一下第 4.3.8 节，在那里我们看到的是普通事件的优先级安排，而djyos中，是把整个软件系统的优先级同一管理的，这就必然涉及到中断优先级，以及中断优先级与事件优先级的关系。图 6-4 所示的优先级体系中，我们可以看到，无论是异步事件还是普通事件，其优先级只有量的不同，没有质的差别，都属于事件优先级组范围。**这反映在djyos的调度策略中，调度使能与禁止和异步信号使能与禁止是等同的。**

<table>
<tr><td colspan="2">优先级组别</td><td>优先级范围</td><td colspan="2">说明</td></tr>
<tr><td>绝对优先级组</td><td>实时中断</td><td>中断控制器决定</td><td colspan="2">操作系统提供服务，但不管理，其中断使能、禁止等完全由应用程序决定，用于实时性要求非常高的中断事件。</td></tr>
<tr><td rowspan="7">事件优先级</td><td>异步事件组</td><td>中断控制器决定</td><td>异步弹出，异步切入，不可阻塞，直到 ISR 函数返回才结束。</td><td>用于实时性要求稍低的中断事件，可使用几乎所有操作系统服务。</td></tr>
<tr><td rowspan="6">普通事件组</td><td>0～127</td><td rowspan="5">调用 y_event_pop 弹出、由调度器切换上下文、调用 y_event_done 函数结束</td><td>紧急事件优先级组，用于紧急事件。</td></tr>
<tr><td>128～199</td><td>实时优先级组，用于高实时性的事件。</td></tr>
<tr><td>200</td><td>轮转优先级，建议所有轮转调度的事件使用。</td></tr>
<tr><td>201～249</td><td>后台优先级组，用于提供后台服务的低优先级事件。</td></tr>
<tr><td>250</td><td>操作系统提供的后台服务事件</td></tr>
</table>

图 6-4 djyos 的优先级体系

djyos 系统的中断管理器实现如下特征：

1. 操作系统运行过程中不会禁止实时中断（但应用程序可以禁止），而异步信号可能在访问临界资源时被禁止。
2. 普通事件组的事件正在处理时，只要被使能，无论发生实时中断还是异步信号，都可以中断正在处理的事件。
3. 正在服务实时中断时关闭异步信号使能，确保实时中断不被异步信号中断。
4. 实时中断响应速度比异步信号快。
5. 中断引擎已经做好了支持嵌套中断的准备，只要目标系统支持嵌套，中断 ISR 就可以调用 int_enable_nest_asyn_signal 或 int_enable_nest_real 函数允许嵌套。

下列特征涉及到中断优先级和中断嵌套，是否支持及如何支持，由目标系统的软硬件设计者实现，核心系统不做硬性规定，留给移植者一块自由发挥的“自留地”。设计者可以在设计硬件的中断管理器时提供必要的支持，也可以在较弱的硬件上使用软件技巧实现，不提供任何支持（让所有函数空着）也是允许的。

1. 当异步信号正在被服务，实时中断信号到达能否打断正在服务的异步信号。
2. 在异步信号组内或实时中断组内，高优先级的中断线能否打断低优先级的中断线中断。
3. 在异步信号组内或实时中断组内，高优先级的中断线是否会被低优先级的中断线打断。
4. 是否支持用软件设置中断线的优先级，是支持所有中断线任意设置，还是局部任意设置，还是压根就不能设置。
5. 是否支持嵌套优先级和子优先级，关于嵌套优先级和子优先级，参见第 6.2.1.2 节。

上述功能，djyos 虽然没有统一实现，但提供统一的接口，即用户可以不实现，也可以实现一半，也可以全部实现，但只要实现，就必须使用规范的接口；即使不实现，也必须提供空函数，使使用这些功能的程序能够顺利编译。这些接口函数和数据结构有：

1. 中断线控制块数据结构struct int_line（见第 6.2.1.2 节）的成员nest_prio和sub_prio。
2. int_enable_nest_asyn_signal( );　　//允许异步信号嵌套
3. int_disable_nest_asyn_signal( );　　//禁止异步信号嵌套
4. int_enable_nest_real( );　　//允许实时中断嵌套
5. int_disable_nest_real( );　　//禁止实时中断嵌套
6. int_set_nest_prio(ufast_t ufl_line);　　//设定嵌套优先级
7. int_set_sub_prio(ufast_t ufl_line);　　//设定子优先级

这种安排实际上是一种妥协，由于中断 ISR 的特殊性，要求中断发生后尽快执行用户提供的 ISR 函数，且 ISR 函数执行完后能够尽快返回，因此不宜在中断引擎里做太多的事情。嵌入式操作系统面对的目标 CPU 种类非常多，每种 CPU 又可能面对不同的中断管理器，硬件实现的中断管理功能差异很大，这些差异主要集中在以下几个方面，移植操作系统时，务必仔细阅读你的硬件手册。

1. 不是所有的硬件都支持中断嵌套。
2. 支持中断嵌套的硬件，也不一定是高优先级中断能打断低优先级中断而相反则不能。有些硬件允许低优先级中断打断高优先级中断。比如 s3c44b0x，如果允许中断嵌套，则后到的中断可以打断先到而正在服务的中断，不管他们的优先级关系如何；而同是 ARM 系列的 cortex-m3，则严格地只有高优先级才能打断低优先级。
3. 不是所有的硬件都支持软件设定中断优先级，许多 CPU 其中断优先级是钉死的，不允许用户改变，或者是分组管理的，可以软件改变中断优先级，但不能任意改变。有些硬件只支持 nest_prio，有些硬件只支持 sub_prio，有些硬件两者都支持，还有些硬件根本就不支持优先级。

如果在这些差异如此之大的硬件上统一实现上述功能，势必造成中断管理器和中断引擎

过于复杂，执行时消耗过多的 CPU 时间和资源，这可能并不是用户所需要的。许多嵌入式系统并没有太严苛的时限要求，中断不分优先级或者不支持嵌套是允许的；在需要实现中断嵌套和严格优先级的场合，笔者建议系统设计者选用支持这些功能的硬件，而不要在不支持的硬件上用软件技巧强行实现。只有在硬件已经确定不能修改且不支持优先级或嵌套，而实际应用又必须实现的时候，才可考虑用软件技巧实现之。

6.2.1.2 中断控制数据结构

中断是凌驾于正常处理流程之上的暴力机构，中断管理功能的设置，也是各种CPU中最具特色的部分。由于中断是实现系统关键任务的核心硬件，各CPU设计厂商无不在中断功能上大做文章，使中断机制呈现百花齐放，各逞奇能的状态。操作系统对CPU中断功能的充分支持，对应用程序设计者来说至关重要。然而千姿百态的中断控制器，与追求版本统一、移植性强的操作系统来说，是相互矛盾的。都江堰操作系统采取的策略是，以CPU通用支持的中断机能为基础，制作统一的系统级的中断控制器架构，如 图 6-5 所示。图 6-5 是应用程序看到的中断结构，这个结构与具体硬件无关，也就是说，当移植操作系统时，无论目标硬件系统的中断管理器是怎样的，都必须实现图中的结构。唯一与目标系统相关的是中断线的数量，port_kernel.h文件中定义的常量cn_int_num即是目标系统中断线的数量。

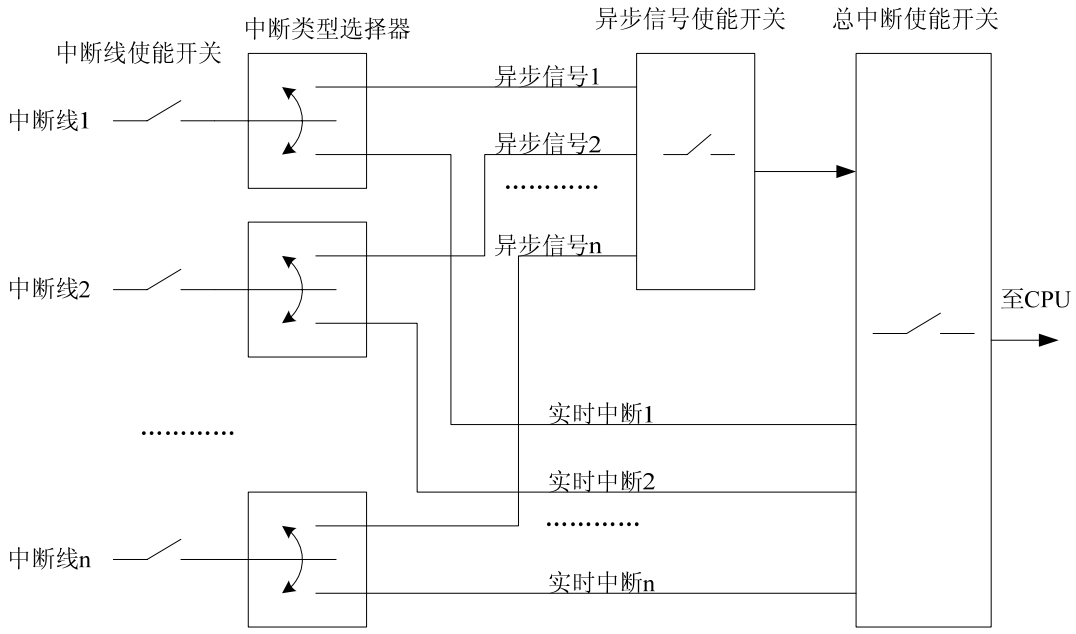


图 6-5 djyos 系统中断结构图

1. 任意中断源可以任意设定属于异步信号还是实时中断。
2. tick 中断属于异步信号中的一个。
3. 异步信号可以调用除 y_event_done 外所有操作系统服务，只是不会发生阻塞。
4. 实时中断只能调用部分操作系统函数，不会引起任务切换。

读者应该注意到了，图 6-5 中有异步信号使能开关，但没有实时中断使能开关，即实时中断不能统一关闭。我们认为，当要关闭实时中断时，必然会把异步信号也关掉，因此不存在单独关闭实时中断的需求，因此djyos不提供单独关闭实时中断的服务。如果用户实在要实现单独禁止实时中断，可在允许异步信号且允许总开关的情况下，逐个关闭被配置为实时中断的中断线来实现。

中断的数据结构在int.h文件中定义，如 代码 6-1 所示。

代码 6-1 中断数据结构

```
//表示各中断线状态的位图占 ucpu_t 类型的字数
#define cn_int_bits_words   ((cn_int_num+cn_cpu_bits-1)/cn_cpu_bits)
struct int_line         //中断线数据结构，每中断一个
{
    void    (*ISR)(ufast_t line);
    struct  event_script *sync_event;       //正在等待本中断发生的事件
    ucpu_t en_counter;     //禁止次数计数,等于 0 时表示允许中断
    ucpu_t int_type;         //1=实时中断,0=异步信号
    sint16_t  nest_prio;                //嵌套优先级，数越小，优先级越高
    sint16_t  sub_prio;                 //子优先级，数越小，优先级越高
};

struct int_master_ctrl          //中断总控数据结构。
{
    //中断线属性位图，0=异步信号，1=实时中断,数组的位数刚好可以容纳中断数量，与
    //中断线数据结构的 int_type 成员含义相同。
    ucpu_t  property_bit_map[cn_int_bits_words];
    ucpu_t nest;      //中断嵌套深度,主程序=0,第一次进入中断=1,依次递加
    //中断线使能位图,1=使能,0=禁止,反映相应的中断线的控制状态,
    //与总开关/异步信号开关的状态无关.
    ucpu_t  enable_bit_map[cn_int_bits_words];
    bool_t  en_trunk;           //1=总中断使能,  0=总中断禁止
    bool_t  en_asyn_signal;     //1=异步信号使能,0=异步信号禁止
    ucpu_t en_trunk_counter;    //全局中断禁止计数,=0 表示允许全局中断
    ucpu_t en_asyn_signal_counter; //异步信号禁止计数,=0 表示允许异步信号
};
```

数据结构的部分成员说明如下：

1. ISR，顾名思义，这是用户提供的中断服务函数指针，注意该函数是标准的 ANSI C 语言函数，无需使用编译器扩展的、用于支持中断的关键字修饰。
2. sync_event，同步事件指针，参见第 0 节。
3. en_counter，中断线使能计数，初始值为 1，每调用一次 int_save_line(my_line)函数，my_line 号中断线的 en_counter 增 1，int_restore_line 函数则相反。en_counter 减至 0 则该中断线被使能，否则禁止。en_trunk_counter、en_asyn_signal_counter 与之类似，分别对应总中断开关和异步信号开关。
4. nest_prio和sub_prio，nest_prio是嵌套优先级，nest_prio高的中断可以打断优先级低的中断，实现中断嵌套。sub_prio是子优先级，如果若干个nest_prio相同的中断，具有不同的sub_prio，则如果这写中断中的两个或更多中断信号同时到达的话，sub_prio优先级高的中断先服务。这两个成员只是系统提供的接口，是否实现由目标系统的软硬件设计者自行决定，参见第 6.2.1.1.2 节。

6.2.1.3初始化

djyos初始化中断系统的时间非常早，在CPU状态以及寄存器和内存初始化后，就立即初始化中断系统（参见第 4.7.1 节），初始化过程如 代码 6-2 所示，初始化后，中断状态为：

1. 全部中断被设定为异步信号。
2. 全部中断线的 en_counter = 1，即禁止中断，en_asyn_signal_counter=1，禁止异步信号；en_trunk_counter=0，允许实时中断。

tick 中断则在多事件调度被启动之前初始化，被设为异步信号，这是在__y_start_os 函数中调用__y_init_tick 函数完成的，因此__y_init_tick 函数是一个平台移植关键函数。然后，__y_start_os 将打开异步信号使能。

初始化后，异步信号是关闭的，直到在__y_start_os函数才打开，在此期间，应用程序可以连接ISR，也可以打开或关闭中断线，可以设定某中断线是异步信号还是实时中断，但绝对不能打开异步信号使能。初始化时可以使用int_save_asyn_signal和int_restore_asyn_signal两个函数，但必须成对使用，否则，如果int_save_asyn_signal调用次数多了，在__y_start_os函数中就不能成功打开异步信号，如果int_restore_asyn_signal函数调用次数多了，就会提前打开异步信号，使初始化工作无法完成。实时中断则不同，初始化后立即就可以使用，这使得部分打开电源就必须开始的工作能够在上电数毫秒内开始处理，例如在安全钩子（参见第 4.7.1.4 节）中使用实时中断。这不同于许多操作系统必须等操作系统加载和初始化全部完成才能打开中断。

代码 6-2 中断初始化

```
void __int_init(void)
{
    ufast_t ufl;
    __int_init_hard();
    __int_echo_all_line();
    for(ufl=0;ufl<cn_int_num;ufl++)
    {
        __int_cut_line(ufl);
        tg_int_table[ufl].en_counter = 1; //禁止中断,计数为 1
        tg_int_table[ufl].int_type = cn_asyn_signal;
        tg_int_table[ufl].ISR = int_empty; //所有中断函数指针指向空函数
        tg_int_table[ufl].sync_event = NULL;
    }
    for(ufl=0; ufl < cn_int_bits_words; ufl++)
    {
        //属性位图清零,全部置为异步信号方式
        tg_int_global.property_bit_map[ufl] = 0;
        //中断使能位图清 0,全部处于禁止状态
        tg_int_global.enable_bit_map[ufl] = 0;
    }
    tg_int_global.en_asyn_signal = false;
    tg_int_global.en_asyn_signal_counter = 1;
    __int_cut_asyn_signal();
```

```
    tg_int_global.en_trunk = true;
    tg_int_global.en_trunk_counter = 0;
    __int_contact_trunk();
}
```

注：cn_int_bits_words 是一个常量，表示属性和使能位图占用 ucpu_t 类型数组的尺寸。如果中断线数量是 20 个，cpu 字长是 16 位，则 cn_int_bits_words=2，如果字长是 32 位，则 cn_int_bits_words=1，类推之。

6.2.1.4异步信号

下列语句序列将把第 my_line 号中断线初始化成异步信号并使能，并把 my_isr 函数指定为中断服务函数：

```
int_isr_connect(my_line, my_isr);
int_evtt_connect(my_line, my_int_evtt);
int_setto_asyn_signal(my_line);
int_restore_line(my_line);
```

异步信号的执行过程如 图 6-6 所示。

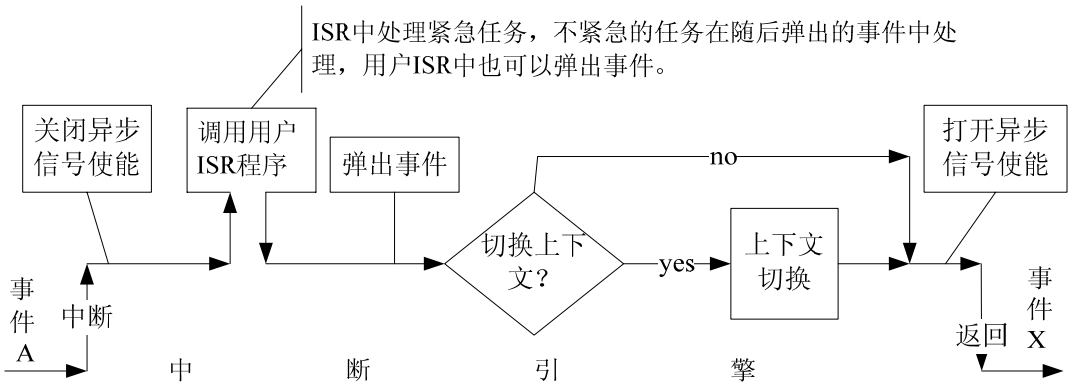


图 6-6 异步信号响应流程

CPU响应异步信号中断后，首先调用的是操作系统提供的中断引擎，如 代码 6-3 所示。引擎依次调用用户ISR和弹出中断线事件。需要说明的是，中断ISR和弹出事件的操作都是可选的，用户是否调用和如何调用int_isr_connect函数和int_evtt_connect函数，决定了用户处理该异步信号的策略：

函数原型：int_isr_connect(my_line, my_isr,my_int_evtt);

//不调用本函数则 my_isr=NULL。

void int_evtt_connect(ufast_t ufl_line,uint16_t my_evtt_id);

//不调用本函数则 my_int_evtt =NULL。

1. my_isr !=NULL，my_int_evtt = cn_invalid_evtt_id，说明用户只用在中断上下文中执行的用户 ISR 函数处理该异步事件。
2. my_isr =NULL，my_int_evtt != cn_invalid_evtt_id，说明用户使用在事件上下文中执行的事件来处理该异步事件，优先级是在登记事件时设定的。
3. my_isr !=NULL，my_int_evtt != cn_invalid_evtt_id，说明用户在中上下文中执行的 ISR 中完成少量紧急任务，把大量不是那么紧急的任务留给在事件上下文中执行的事件。
4. 还有一种更灵活的方式，djyos 允许在用户 ISR 中调用 y_event_pop 弹出事件，程序员可

以在 int_isr_connect 函数中只配置 my_isr。弹出事件和事件调度需要比较大的开销，用户可以在 ISR 中先做判断，有必要才调用 y_event_pop 弹出事件，而且，弹出的事件选择优先级就不限于事件类型的默认优先级，还可以弹出多条不同类型的事件。

用户的 ISR 函数是由中断引擎当作普通 C 语言函数调用的，“用户 ISR”和“中断 ISR”是两个不同的概念，“用户 ISR”是普通 C 函数，“中断 ISR”是专用于响应中断的 C 函数，这种函数是以中断返回指令结束的。一般来说，嵌入式 C 编译器都会提供扩展功能，以支持用 C 语言编写中断 ISR，以专门的关键字声明某函数是“中断 ISR”，比如在 gcc for arm 这样声明 f 为 IRQ 中断的“中断 ISR”函数：

```
void f () __attribute__ ((interrupt ("IRQ")));
```

中断引擎函数__int_engine_asyn_signal 就可能是“中断 ISR”函数，也可能不是，取决于具体实现。在 djyos for 44b0x 版本中，该函数就是普通函数，其调用路径是：“汇编实现的中断 ISR” —> __int_engine_al l—>__int_engine_asyn_signal。

代码 6-3 异步信号引擎

```
void __int_engine_asyn_signal(ufast_t ufl_line)
{
    struct event_script *event;
    tg_int_global.nest_asyn_signal++;
    //以下几句移植很关键，请用户根据自己的硬件仔细设计，需要使 CPU 进入这样的状态：
    //异步信号被禁止而总开关打开的状态，类似于依序调用 int_save_asyn_signal 和
    //__int_contact_trunk 两个函数，
    __int_cut_asyn_signal();            //移植提示:若硬件关闭了异步信号，则无需这句
    __int_echo_line(ufl_line);          //中断应答
    tg_int_global.en_asyn_signal = false;
    tg_int_global.en_asyn_signal_counter = 1;
    __int_contact_trunk();              //移植提示:若硬件没关闭总中断，则无需这句

    event = tg_int_table[ufl_line].sync_event;
    y_event_pop(tg_int_table[ufl_line].my_evtt_id,0,0,0);
    if(event != NULL)     //看同步指针中有没有事件
    {
        event->last_status.all = event->event_status.all;
        event->event_status.bit.wait_asyn_signal = 0;
        __y_event_ready(event);     //把该事件放到 ready 队列
        tg_int_table[ufl_line].sync_event = NULL;     //解除同步
    }
    //调用用户中断函数,此时嵌套中断是禁止的，用户如果需要允许嵌套，可以在
    //vec_func 函数中打开异步信号来达到。
    tg_int_table[ufl_line].ISR(ufl_line);
    if(tg_int_global.nest_asyn_signal == 1)                    //1
    {//已经是最后一级中断嵌套了,看看是否要调度
        if(pg_event_ready != pg_event_running)
            __schedule_asyn_signal();           //执行中断内调度     //2
    }
    //以下几句移植很关键，请用户根据自己的硬件仔细设计:
```

```
    //既要调用 int_restore_asyn_signal 使 en_asyn_signal_counter 归 0，又不能使
    //异步信号真的打开，而是要恢复到 CPU 响应中断后的状态，由中断返回指令打开。
    __int_cut_trunk();                    //移植提示:若硬件没关闭总中断，则无需这句
    tg_int_global.en_asyn_signal = true;
    tg_int_global.en_asyn_signal_counter = 0;
    __int_contact_asyn_signal();      //移植提示:若硬件关闭了异步信号，则无需这句

    tg_int_global.nest_asyn_signal--;
}
```

注：

1. 这里需要判断是否需要事件切换并执行切换，参见第 4.3.9 节说明。
2. 这是一个移植关键的汇编函数，参见第 5 节。

对照 图 6-6 和代码中的注释，不难理解 代码 6-3，下面重点说明一下异步信号ISR执行过程中断使能状态的变化。一般来说，CPU响应中断时会自动关闭中断使能，但这并不是我们想要的结果。djyos要求，在异步信号处理过程中，只要应用程序不主动打开异步信号中断嵌套，异步信号就始终维持禁止状态。djyos系统中，异步信号ISR允许调用几乎全部的操作系统服务，这些操作系统服务可能要访问临界资源而关闭调度，从第 4.3.9 节可知，禁止调度实际上就是使用int_save_asyn_signal函数禁止异步信号。

事件				en_asyn_signal_counter =0（注 1）
	引擎		禁止异步信号使能；	en_asyn_signal_counter =1（注 2）
		用 户 ISR	int_save_asyn_signal（）； 访问与调度相关的临界资源； int_restore_asyn_signal（）；	en_asyn_signal_counter =2（注 3） en_asyn_signal_counter =1 （注 4）
	引擎		恢复异步信号使能状态；	en_asyn_signal_counter =0（注 5）
事件				en_asyn_signal_counter =0

图 6-7 异步信号 ISR 执行过程中使能状态图

图 6-7 注解如下：

1. 在异步信号响应前，必然有 en_asyn_signal_counter =0，表示其是允许的。
2. 在引擎的初始阶段，即使 CPU 在响应异步信号时关闭了异步信号使能，还应该使 en_asyn_signal_counter=1 和 en_asyn_signal=false，否则，第 4 步就会使 en_asyn_signal_counter=0，int_restore_asyn_signal 函数将打开异步信号使能。
3. 用户使用与调度相关的临界资源前，必须关闭调度。
4. int_restore_asyn_signal 与 int_save_asyn_signal 成对使用。第 3 步到第 4 步之间还可能嵌套条用这对函数。
5. ISR 执行完成后，必须恢复异步信号使能状态，由于从此时到 CPU 实际执行返回指令之间，引擎需要执行一些敏感的操作，故不能使 CPU 处于允许异步信号嵌套的状态，故执行完本步骤后，应该处于这样一种状态：
   a) en_asyn_signal_counter=0；
   b) 异步信号开关和总中断开关处于 CPU 刚刚响应异步信号时的状态。

djyos 要求 CPU 响应异步信号不能影响实时中断的响应，故总中断必须维持进入中断前的状态（实际上就是允许状态，否则根本就进不了中断）。如果硬件在响应异步信号时连同总中断也关闭了，就需要在异步信号引擎中重新打开它，直接打开即可，无需管en_trunk_counter，这种情况，实时中断会受到短暂的影响，从响应异步信号到异步信号引擎中打开总中断期间，实时中断不能响应，所幸的是，这段时间非常短。这些操作与具体硬件

相关，因此中断引擎是一个移植关键的代码，这在代码注释中已经注明。

是否允许中断嵌套，djyos 让编写 ISR 的程序员自己决定，调用用户 ISR 之前，异步信号嵌套是禁止的，用户可以在 ISR 执行的任何地方启动或者关闭异步信号嵌套。

用户 ISR 完成后，将弹出一条事件，该事件类型是在 int_isr_connect 函数中设定的。并不是立即返回到主程序，而是继续执行中断引擎的其余部分，如果已经到了最后一级中断中断嵌套，返回主程序前就要判断是否需要切换上下文，需要切换上下文的条件是：

1. ISR 中弹出了事件，且该事件的优先级高于被中断的事件。
2. 执行 ISR 使一些原本在阻塞状态的事件变成就绪态，且新的就绪事件优先级高于被中断的事件。
3. 该 ISR 是 tick 中断，tick 中断使原来在闹钟同步状态的事件因闹铃响而变成就绪态，且新的就绪事件优先级高于被中断的事件；或者因轮转调度使就绪队列发生变化。

如果需要执行上下文切换，将完成以下几个步骤：

1. 响应中断时，被中断事件的上下文将保存到寄存器或中断栈中，究竟是在寄存器还是在栈中，是自动保存还是手动保存，这取决于目标硬件系统。
2. 从寄存器或中断的栈中取出被中断事件的上下文，压入到该事件自己的栈中。
3. 把新事件的上下文从其栈中弹出。
4. 把刚弹出的上下文写入到中断栈中，模拟被中断的是新事件的现场。

这样，中断返回时，不执行上下文切换则直接返回到原来被中断的事件，否则返回到新事件。与引擎开始时对应，返回前需要把异步信号重新打开。

6.2.1.5实时中断

下列语句序列将把第 my_line 号中断线初始化成实时中断并使能，并把 my_isr 函数指定为中断服务函数:

```
int_isr_connect(my_line, my_isr);
int_setto_asyn_signal(my_line);
int_restore_line(my_line);
```

实时中断的执行过程如 图 6-8 所示。

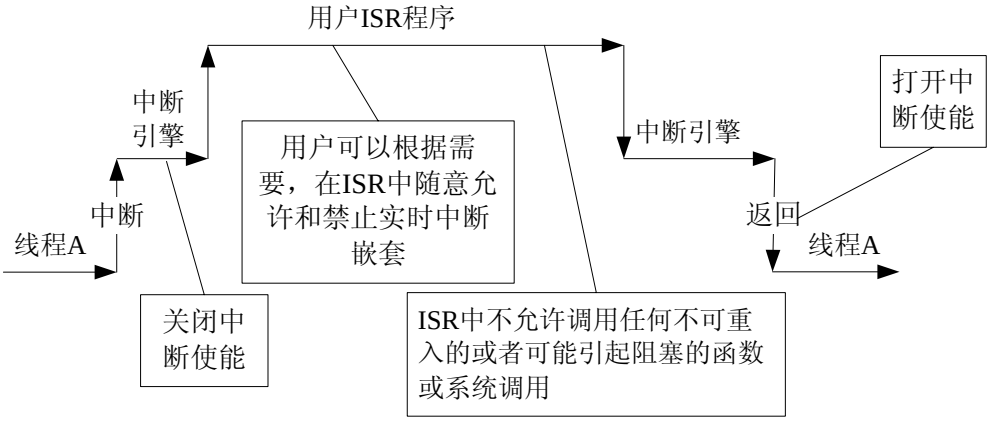


图 6-8 实时中断响应流程

实时中断引擎如 代码 6-4 所示，与异步信号相比，实时中断引擎则简单很多，执行速度也快很多，它实际上就是异步信号引擎去掉了有关同步和调度部分的简化版。此外，就是在中断使能状态的变化上略有不同，由于实时中断的优先级先天就比异步信号高，因此在其

中断引擎中无需考虑打开异步信号使能。
代码 6-4 实时中断引擎

```
void __int_engine_real(ufast_t ufl_line)
{
    tg_int_global.nest_real++;

    tg_int_global.en_trunk = false;
    tg_int_global.en_ trunk _counter = 1;
//     __int_cut_trunk();                  //移植提示:若硬件没有关闭总中断，需增加这句
    __int_echo_line(ufl_line);          //中断应答
    tg_int_table[ufl_line].ISR(ufl_line);   //调用用户中断函数

    tg_int_global.en_ trunk = true;
    tg_int_global.en_ trunk _counter = 0;
    //      __int_contact_trunk();      //移植提示:若硬件没有关闭总中断，需增加这句

    tg_int_global.nest_real--;
}
```

6.2.2 禁止和允许中断

参考 图 6-5 结构，djyos提供 3 对函数用于控制图中各中断使能开关：
int_save_line；　　保存某中断线使能状态并禁止。
int_restore_line；　　恢复该中断线使能状态。
int_save_asyn_signal；　　保存异步信号使能状态并禁止。
int_restore_asyn_signal；　　恢复异步信号使能状态。
int_save_trunk；　　保存总中断使能状态并禁止。
int_restore_trunk；　　恢复总中断使能状态。
这 3 对函数中，三个 int_save_xxx 的代码非常相似，这里单列出 int_save_line 函数的代码。至于异步信号和总中断的代码，有兴趣的读者可阅读随书代码。
代码 6-5 保存中断线状态并禁止

```
void int_save_line(ufast_t ufl_line)
{
    if(ufl_line>=cn_int_num)
        return;
    if(tg_int_table[ufl_line].en_counter!=cn_limit_ucpu)//达上限后再加会回绕到 0
        tg_int_table[ufl_line].en_counter++;
    //原算法是从 0->1 的过程中才进入，但如果在 en_counter != 0 的状态下
    //因故障使中断关闭，将使用户后续调用的 en_counter 起不到作用
    __int_cut_line(ufl_line);                                                    //1
    tg_int_global.enable_bit_map[ufl_line/cn_cpu_bits]
                    &= ~(1<<(ufl_line % cn_cpu_bits));
```

```
}
```

注释 1 处的代码值得注意，这里直接调用禁止中断的语句，而不是在条件：if(tg_int_table[ufl_line].en_counter ==1);成立才调用，虽然这样写效率更高一些，它只在 en_counter 从 0->时条件才成立并执行禁止中断语句，但这样安全性会低一些。如果在 en_counter>0 的时候，因以外使该中断线处于允许状态，若按注释 1 处写，再次调用 int_save_line 时将修复这个错误，而写成 if(tg_int_table[ufl_line].en_counter ==1)将不能修复错误。

从第 4.3.9 节可知，异步信号使能与调度使能是等价的，因此int_save_asyn_signal和int_restore_asyn_signal函数还肩负禁止和使能调度的责任，而且从第 4.3.9 节可知，才会改变就绪队列而引发事件切换，因此，int_restore_asyn_signal函数在结束前，需要判断是否需要切换并执行事件切换。另外，在禁止总中断期间，也可能发生改变就绪队列的操作，故在int_restore_trunk函数结束前也要判断是否需要切换并执行事件切换。因此，int_restore_trunk和int_restore_asyn_signal函数相似而与int_restore_line函数有很大不同，这在 代码 6-6 中可以看出他们的不同。这里只列出int_restore_asyn_signal，int_restore_trunk的代码请参考随书代码。

代码 6-6 恢复中断使能状态

```
void int_restore_line(ufast_t ufl_line)          //恢复中断线使能状态
{
    if(ufl_line>=cn_int_num)
        return;
    if(tg_int_table[ufl_line].en_counter != 0)
        tg_int_table[ufl_line].en_counter--;
    if(tg_int_table[ufl_line].en_counter==0)
    {
        tg_int_global.enable_bit_map[ufl_line/cn_cpu_bits]
                |= 1<<(ufl_line % cn_cpu_bits);
        __int_contact_line(ufl_line);
    }else
    {
        __int_cut_line(ufl_line);
    }
}
void int_restore_asyn_signal(void)               //恢复异步信号的使能状态
{
    if(tg_int_global.en_asyn_signal_counter != 0)
        tg_int_global.en_asyn_signal_counter--;
    if(tg_int_global.en_asyn_signal_counter==0)              //1
    {
        tg_int_global.en_asyn_signal = true;     //异步信号设为使能
        __int_contact_asyn_signal();
        if(tg_int_global.en_trunk == true)
        {
            if(pg_event_running != pg_event_ready)
                __schedule();
```

```
        }
    }else
    {
        __int_cut_asyn_signal();        //防止 counter>0 期间意外(bug)打开
    }
    return;
}
```

注 1：这里需要判断是否需要事件切换并执行切换，参见第 4.3.9 节说明。

无论是实时中断还是异步信号，或者具体中断线，djyos 都不提供无条件使能或禁止的服务，因为这是不安全的。禁止中断一般用于提供临界资源访问保护，或者保证硬件操作时序，其形式如下：

禁止中断；

访问临界资源或严格时序的硬件操作；

允许中断；

上述过程乍看上去是安全的，的确如此，但只是在临界资源不嵌套的情况下是安全的，如果发生嵌套访问临界资源呢？比如下面的伪代码：

禁止中断；

访问临界资源 1；

禁止中断；

访问临界资源 2；

允许中断；　　//1

继续访问临界资源 1；

允许中断；

上述代码是不安全的，因为注释 1 处允许了中断，将把临界资源 1 暴露在中断保护之外，继续访问临界资源 1 的时候就可能被中断干扰。系统中临界资源普遍存在，比如各种事件队列、事件类型队列，各种阻塞队列，内存堆等，都是临界资源。所以上述危险性也是普遍存在的。因此，djyos 不提供直接禁止或允许中断的服务，代之以安全的“save-restore”对。

6.2.3 建议的中断配置策略

在 djyos 下，对不同缓急程度的中断事件，建议采用的软件策略：

1. 与中断线连接的慢速设备，像键盘，这种情况可以关闭该中断，用周期性查询来把异步事件转换成普通事件；也可以把该中断配置成异步信号，使用ISR引擎模型（参见第 6.2.1.1.2 节）。
2. 有比较严格的时间要求、仍然在异步信号实时性指标范围内的、连接在中断线上的的事件，可以把配置成异步信号。异步信号的ISR几乎可以使用全部的操作系统服务（异步信号ISR使用操作系统服务的限制参见第 6.1.2 节），方便ISR函数的开发。如果从中断信号产生到开始处理异步事件的允许延迟时间小于最长连续关闭调度（异步信号）的时间，就表示异步信号可以满足该异步事件的实时性要求（详见 3.1 节）。
3. 有非常严格的实时性要求，异步信号难于满足的中断事件，可以配置成实时中断。
4. 中断事件的一小部分实时性要求较高，但尚在异步信号能够满足的范围内，而中断会引发大量实时性要求较低的任务，典型的是通信中断。可以把该中断配置成异步信号，使用ISR引擎模型（参见第 6.2.1.1.2 节）。

强烈建议，设计者审慎处理中断控制器！中断的功能很强大，是设计硬实时系统的利器，

但这是建立在暴力的基础上的，嵌入式系统中最常用的“法律”语言——ANSI C 官方语言中，根本就没有“中断”这一法律术语，所有 c 编译器支持中断的能力，都是各门派自己的武功秘笈，所以说，中断是凌驾于法律之上的。C 编译器既然不知道什么时候会发生中断，也就无法为中断做出任何保护措施，一切全靠程序员自己控制。随意地操纵中断控制器是在玩火——常常会改变系统的响应特性，导致某些关键任务有可能得不到及时响应，这种凶险的意外事故随时可能猛烈发作。推荐的做法是，中断控制器功能设置，比如优先级的分组、中断嵌套方式等，预先经过计算论证，并且在开机初始化时一次性地设置好，以后就再也不动它。只有在绝对需要且绝对有把握时，才小心地更改，并且要经过尽可能充分的测试。中断控制架构中，操作系统提供的服务部分，都江堰系统已经做过充分的测试，而自留地部分，程序员务必小心。

6.2.4 ISR 函数

在 djyos 中，用户的 ISR 函数是一个普通的 C 语言函数。

虽然从主程序的程序员看来，中断服务是透明的，进入中断时保护了现场，返回时又恢复原样，好像主程序连续运行似的，但执行中断服务毕竟要在 CPU 中留下脚印，有些共享数据可能被修改，由于 CPU 被剥夺，似乎 CPU 变慢了似的。因此，编写中断处理程序将受到以下条件的约束：

1. 由于中断程序执行是异步的，就有可能打断重要代码的执行，为避免被打断的代码停止时间过长，中断处理程序必需执行得越快越好。当然，重要代码也可以通过关中断的方式进行自我保护，根据 0 节的论述，这会降低系统的实时性能，设计者必需作出折衷。
2. 中断处理程序执行时可能会屏蔽部分或全部中断，还可能打断正在执行的另一个中断处理程序，这些都会影响系统的性能与反应能力。因此，中断执行时间必需很短。
3. 中断处理程序执行时间的长短直接关系到系统的实时性，在 0 节中讲到，实时性中用户因素包含用户程序的执行时间，如果用户程序正在执行的时候被中断打断，则中断处理程序的执行时间将直接累加在用户程序执行时间中，更糟的是，用户程序执行时可能被多次打断。这也决定了中断执行时间必需很短。
4. 中断往往要对硬件进行操作，必需配合特定硬件的时限要求。

因此，中断响应函数的执行时间应该尽量短，就意味着不能在中断函数中干太多的事情，毕竟，一次硬件中断可能引发很多事务，但并非所有事务都是十万火急非马上完成不可的，比如一次网络通信中断，需要立刻完成的工作仅仅是从网络设备中拷贝数据，至于什么时候执行网络通信协议，则是可以推后一些完成的。究竟什么事情应该在中断里完成，什么事情不应该在中断完成，不能在中断里完成的工作又应该在什么时候、什么地方完成，很遗憾，从来就没有这么一个标准，开发者应该自行做出判断。虽然不存在严格的对错之分，但决定中断完成什么工作还是有章可循的：

1. 实时性非常高的事务，应该在中断处理程序里执行。
2. 硬件相关且该硬件有严格的时序要求的操作，则在中断处理程序里执行比较合适。
3. 不能被相同中断或其他中断打断的工作，应该在中断处理程序里执行。
4. 中断执行时间的长短，直接影响被中断事务的实时性，如果有这样一个事务，若放在中断外执行，则不能保证自身的实时性；若放在中断里执行，则不能保证被中断事务的实时性。此时，就应该考虑升级系统了，要么釜底抽薪，重新设计软件体系，在新体系下不再需要上述实时性要求；要么提高硬件性能，用更快的执行速度实现实时性。

由于中断的异步性和不可阻塞性，在ISR中使用操作系统服务是有限制的，如 表格 6-1 所示，不遵守这些限制的结果是不可预料的。

表格 6-1 ISR 的限制

功能	异步信号 ISR	实时中断 ISR
弹出事件 y_event_pop	可以使用，没有限制	不能使用
事件完成 y_event_done	可以使用，没有限制	不能使用
从系统堆中分配内存	可以使用，如果内存不足则直接返回 NULL	不能使用
打开设备	可以使用，但若打开次数超过限制则直接返回 NULL，不会阻塞。	不能使用
读、写、控制设备	可以使用，执行过程中若遇到阻塞直接返回	视设备本身的读、写、控制函数而定，如果这些函数中没有使用本表中规定不能在实时中断 ISR 中使用的功能，则允许
关闭设备	可以使用，无限制	不能使用
锁（含计数信号量和互斥量）	可以使用，但不能取得钥匙时直接返回 NULL，不阻塞	不能使用
本地内存分配	可以使用，但不会阻塞	不能使用
所有同步函数（含闹钟同步、中断同步、事件同步、事件类型弹出同步，事件类型完成同步）	可以调用，但相当于调用空函数，无效。	不能使用

第7章 内存管理

内存是计算机最重要的资源之一，安全地使用内存是保证软件稳定可靠运行的基本条件，嵌入式系统也不例外。而内存相关的 bug 中，大多数跟堆和栈的使用有关，本节讲述 djyos 系统的内存管理，并以此为基础谈谈在高可靠软件设计中的内存分配策略。

内存是操作系统和应用程序运行所需要的核心资源，内存管理模块的初始化过程与操作系统初始化过程紧密相关。djyos系统中，在初始化和程序运行的不同阶段，将执行不同的内存分配策略，如 图 7-1 所示。

首先，操作系统初始化开始后到内存管理模块初始化之前，是不允许使用动态内存分配的，此时调用内存分配函数的后果是不可预料的。当然，除了安全钩子函数外（参见第 4.7.1.4 节），这期间应用程序并没有机会执行。

调用准静态分配堆初始化后，这是操作系统初始化的第 6 步，malloc 族函数可以使用了。

稍后，将调用固定块分配初始化函数，以允许接下来的初始化过程可以创建固定块池，并从池中分配内存。

最后，在完成相当部分的初始化过程以后，才调用动态分配堆初始化，以允许程序员自由使用 malloc 族函数。

内存管理模块使用动态分配还是准静态分配算法，对程序员来说是透明的，用户要从堆中分配 1000 字节的内存，无论是准静态分配阶段，还是块相联分配阶段，都是调用 m_maloc（1000，timeout）函数。

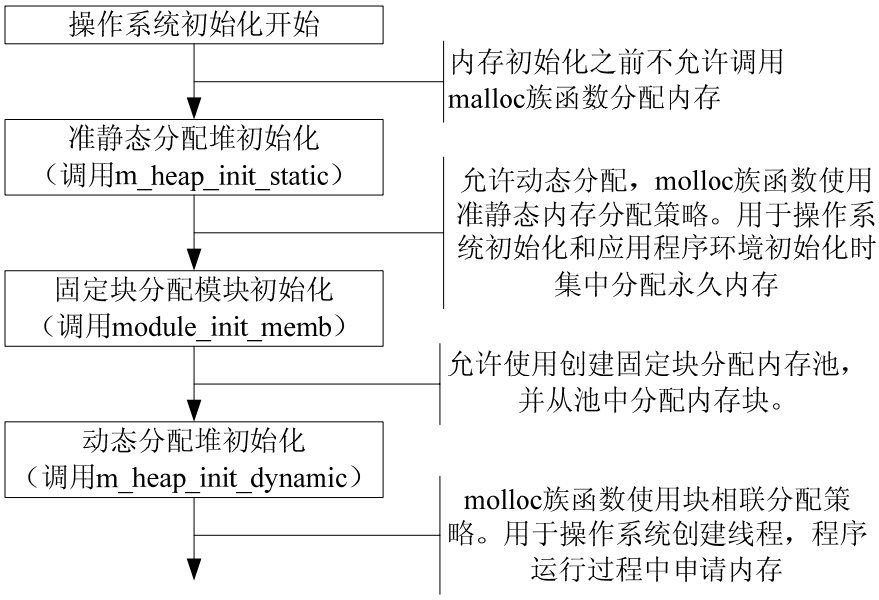


图 7-1 操作系统初始化与内存管理模块初始化

7.1 准静态内存分配

两个定义：

永久动态内存——使用动态分配但自分配以后直到程序运行结束都不释放的内存。

临时动态内存——程序运行过程中分配，使用完后就可以释放的内存。

一块“忙”内存把一块大块的内存从中间切开分割成两块较小的内存，就产生了内存碎片，内存碎片会严重降低内存的效能。内存碎片是软件运行过程中动态地执行 malloc 和 free 族的函数进行内存分配和释放后产生的，当插入在两块空闲内存中的“忙”内存释放时，这三块内存将聚合成一块大块内存，碎片也就随之消除。因此，制造碎片的“忙”内存块的生存期越长，碎片的生存期越长，由永不释放的内存块制造的碎片叫永久碎片。永久碎片的危害最大，但在大多数情况下，它是可以预防的，怎样才能避免产生永久碎片呢？只要我们尽可能地在第一次分配临时动态内存前，集中分配所有永久动态内存，就可以最大限度地使永久动态内存连续分配，使其不制造碎片。许多产品在编译时并不能确定内存需求，只有在运行时才可以确定。这种情况又可以细分为两种类型，一是完全无法预测什么时候需要使用永久内存，这种情况是无法集中分配的；二是在系统初始化的时候就可以确定需要多少内存，这是实际应用中更为普遍的情况，比如有些产品通过跳线来选择通信口的端口类型和端口数量，或者用配置文件确定模拟通道的数量，而且一旦选定就不会更改（不允许带电拔插跳线就属于这种情况），配置确定以后，内存需求也就随之确定。程序可以在开机后，在初始化阶段计算内存需求，在第一次分配临时动态内存之前，集中分配所有永久动态内存，就可以避免产生永久碎片。

djyos 的内存管理在初始化阶段使用准静态内存分配策略，准静态分配的特征是：

1. 按实际申请的内存量连续分配，而不像“块相联分配法”那样把内存格式化成标准尺寸来分配。这种分配方法的开销极小，除对齐开销外，只需要为每块内存提供 1 个 32 位整数存储该块内存的尺寸。分配内存的过程跟编译连接器为静态变量分配内存几乎是一样的，仅仅在对齐方式上有所差异，内存利用效率与编译器分配静态内存（如数组）几乎相同，“准静态”因此得名。
2. m_free 函数只能释放最后一次分配的内存块，否则 m_free 将是空操作。m_malloc 和 m_free 函数严格地必须成对调用，否则将产生内存泄漏。成对调用允许嵌套，即“分配 P1——分配 P2——释放 P2——释放 P1”的形式是允许的，但如果把上述过程变为“分配 P1——分配 P2——释放 P1——释放 P2”的话，P1 将丢失，造成内存泄漏。

这是一种特殊的内存分配策略。由__m_static_malloc和__m_free函数完成内存分配和释放，这两个函数并不提供给用户使用，而是由操作系统的内存分配接口函数间接调用。用户在执行m_init_heap_static进行内存堆初始化之前不能使用m_malloc族函数，调用m _heap_init _static之后调用m _heap_init _dynamic之前，允许调用m_malloc族函数，由malloc函数间接调用__static_malloc为用户分配内存。如 代码 7-1 所示，m_init_heap_static函数非常简单，就是设置堆底和堆顶指针。

代码 7-1 m_init_heap_static

```
void m_heap_init_static(void)
{
    tg_mem_global.dynamic = false;
    tg_mem_global.static_bottom = (uint8_t*)align_up((ptu32_t)cfg_heap_bottom);
    tg_mem_global.heap_bottom = tg_mem_global.static_bottom;
    tg_mem_global.heap_top=(uint8_t*)align_down((ptu32_t)cfg_heap_top);
}
```

类静态分配方法是效率最高，资源消耗最少、实现最简单的一种内存分配方式。但只能在程序初始化阶段可以使用，成功分配内存时，函数返回获得的内存块指针，如果内存分配

不成功，返回 NULL。

7.2 块相联分配法

7.2.1 内存组织

djyos操作系统内存管理器用块相联的方式管理系统堆内存（块相联方法见 2.9.4 节）。内存分配的最小单位是页，页尺寸的确定方法有两种，第一种是由硬件确定，在支持mmu或者mpu的系统上，由mmu或mpu确定；第二种是在不支持mmu和mpu的系统上，由用户设定，用户可以设定页的尺寸，但只能设定为 2 的整数次方幂个字节，不能任意设定。内存是以块为单位进行分配的，每个块包含若干页，块所包含的页数也不是任意的，只能是 2 的n次方幂个页，n为整数，如 1 页、2 页，4 页，8 页，16 页，……，n称为块的阶数（scale），n=0 为 0 阶块，n＝1 称为 1 阶块，依次类推。用户可以改变常量cn_block_limit的值，设定允许单次malloc分配的最大的内存块的字节数，当cn_block_limit＝0 时，由堆空间的尺寸限定，适当减小cn_block_limit的值可以加速内存分配。对页尺寸和块尺寸的限制是出于程序执行效率的考虑，djyos要动态地为事件创建线程，分配线程的执行栈，因此内存管理的执行效率将直接影响操作系统的调度效率，使用 2 的n次幂规格化的块和页尺寸使操作系统可以用移位代替大量的乘除运算。

内存管理器用位图金字塔来表示内存分配状况，系统维护两种位图金字塔：阶位图金字塔（下简称阶金字塔）和空闲位图金字塔（下简称空闲金字塔）。阶金字塔表示可供分配的块，金字塔的层数就是最大可分配的块的阶数；空闲金字塔表示内存块的分配情况，每阶一个。一个含 1026 页的内存堆，可被初始化成如 图 7-2 所示的阶位图金字塔，塔中每个格子代表一个可分配内存块：金字塔共 10 级，每级可用块数是上一级可用块数的一半（取整）。0 级有 1026 个块可供分配，第 10 级则只有 1 个块。

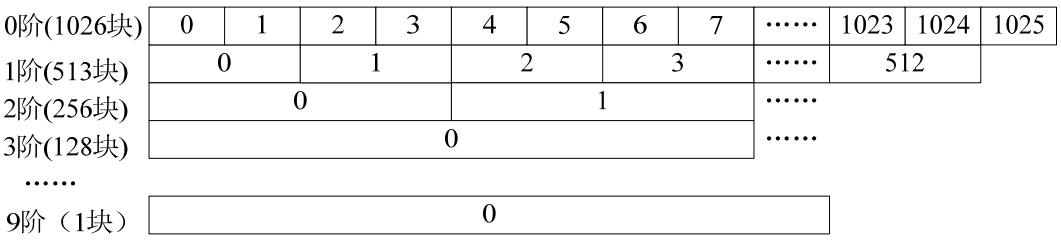


图 7-2 阶位图金字塔

空闲金字塔系统用数组位图表示内存块的分配情况，阶金字塔中的每一级对应一个空闲金字塔，表示该级内存块的分配情况。为什么不用简单位图而是用金字塔表示内存块的闲忙状态呢？原来，系统堆的尺寸可能很大，这样位图就会很大，硬件支持 mmu 时，寻址范围将达到 4G，以 4K 为一页，0 阶块位图将有 1048576 位，要用 32768 个 32 位字表示，1 阶块位图也有 524288 位合 16384 字。分配内存时，就是要从这个位图数组中找出一个空闲块，从如此大的数组织中搜索一个位，无疑是一件极其恐怖的事，其消耗的时间极长且非常不确定。金字塔就是用来解决这个问题的，金字塔的建立规则是（假设 CPU 字长是 n = 8）：

1. 金字塔第 0 级的像素数，就是该金字塔对应的内存块的总块数，比如 图 7-2 中 0 阶块有 1026 块，其对应的空闲位图的 0 阶就有 1026 个像素；1 阶有 513 块，其对应的空闲位图的 0 阶就有 513 个像素。如果 0 级的各个像素对应的内存块空闲，该位就为 0，否

则为 1。

2. 金字塔第 1 级或更高级的像素数为：（前 1 级的像素数 + n -1）/ n，舍弃全部余数，比如 0 级有 1026 位，n=8，则 1 级=（1026 + 8 -1）/ 8 = 129。即下一级的一个位，对应上一级的一个字（按 CPU 字长）。上一级字如果全 1，下一级对应的位就为 1，否则为 0。也就是说，任一级的某一位如果是 0 的话，其下一级肯定有 0，一直追溯的第 0 级，就可以找出空闲块对应的位，该位在位图中的偏移量就是该块内存在堆中的偏移量。
3. 重复上述步骤，直到某一级的位数小于等于 n。

根据上述规则，图 7-2 中 0 阶块的空闲金字塔的各级参数是：0 阶 1026 位，占用 129 字节；1 阶 129 位，占用 17 字节；2 阶 17 位，占用 3 个字节，3 阶 3 位，占用 1 个字节；3 阶的位数已经小于n，故无需算 4 阶。而 图 7-2 中 6 阶块共有 8 块，其空闲金字塔则只有 0 阶。

图 7-3 是一个空闲金字塔的示意图，0 级共有 70 块，CPU字长为 8。

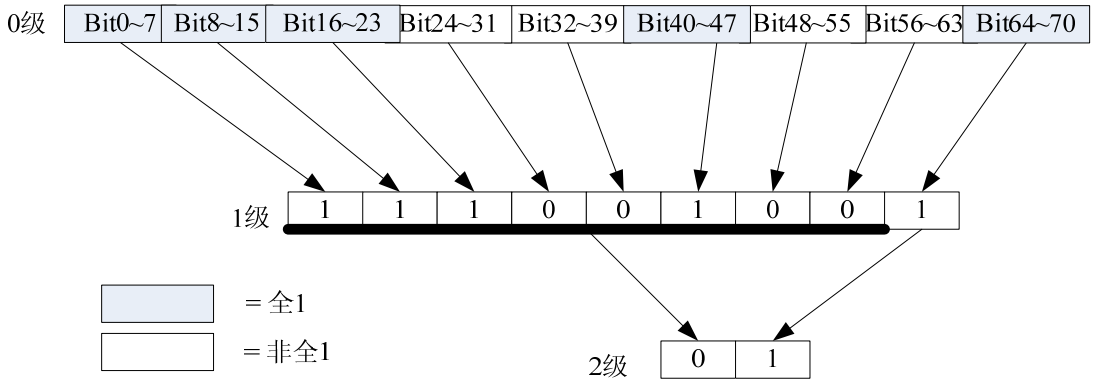


图 7-3 空闲位图金字塔

假设 图 7-3 中，Bit24~31 =0x5B，从图中搜索一个空闲块的过程如下：

1. 2 级的 Bit0 = 0，表示该 Bit 对应的上一级的 Bit0~7 中肯定有 1 个位是 0。
2. 1 级的 Bit3=0，表示其对应的上一级的第 3*8=24~31 位中至少有一个位是 0.
3. 0 级 Bit24~31 是该位图的第 3 个字节，该字节值是 0x5B，Bit1 = 0，Bit1 在整个位图中是第 25 位，表示第 25 块是空闲的。至此搜索完成。

用上述过程，在 4G 寻址空间，页尺寸为 1K，CPU 字长是 32 的系统中，只需要经过 5 次搜索（共 5 次比较、5 次乘法和 5 次加法）就可以定位一个空闲块，速度是相当快的。

位图映像建立起来以后，还要解决如何保存和访问的问题，djyos采用多级指针表来索引所有的位图，全局变量static mem_global_t tg_mem_global用于记录堆内存的信息，在 图 7-4 中有结构类型mem_global_t的定义。由于位图映像及指针表占用的内存是动态的，不能用全局数组的方式定义，所以要在内存堆中画出一块内存来保存，因此，实际可分配的内存要小于堆的尺寸，这就是内存管理的开销。注意内存管理开销是用来保存位图以及指向位图的指针表的，不是用来保存tg_mem_global的，它是一个静态变量，保存在数据段中。djyos的内存管理开销极为出色，它只占用很少的内存，管理一个 8M字节，页尺寸为 4K字节，不支持mmu的内存，内存管理的开销还不到 0.4%。下面我们详细描述内存管理数据结构的建立过程。

首先，需要获取堆空间的起始和结束地址，这与系统硬件和开发环境有关，一般来说编译连接系统会提供这两个地址，开发者要仔细熟悉自己的硬件和开发环境以获取这两个参数。djyos 是在 gcc 下编译和连接的，通过链接脚本文件可以获得堆底地 cn_heap_bottom 和堆顶地址 cn_heap_top。具体方法已超出本书讲述范围，有兴趣的读者可以参考 gcc 相关文档。djyos 并不直接使用这两个常量，而是把这两个常量经过对齐处理后保存在

tg_mem_global. p_heap_bottom 和 tg_mem_global. p_heap_top 两个指针中，是出于可移植性的考虑，虽然 gcc 的链接脚本文件可以使 cn_heap_top 和 cn_heap_bottom 符合系统的对齐要求，但是不能保证 djyos 移植到其他环境后，新的开发系统提供的参数也是经过对齐处理的。

页地址的对齐方式与系统硬件设计有关，当系统硬件包含内存管理单元（即 mmu）时，由内存管理单元的要求确定，往往要求进行页对齐，比如 mmu 的页尺寸是 4k 字节时，页地址必须是 4k 的整数倍。如果没有 MMU，则按 CPU 的要求对齐，大多数嵌入式 CPU 禁止访问非对齐的地址，内存管理器当然不能分配一块非对齐的内存给应用程序使用，由于内存管理器不知道分配的内存最终用途，故必须按最严格的方式对齐。对齐方式与 cpu 的体系结构有关，具体的对齐方式要参考目标 cpu 的文档，比如 ARM 就必须按 8 字节对齐。内存管理器通过调整堆底部地址使其对齐方式满足系统硬件的要求。

块的对齐方式则按页计算，0 阶块以 1 页对齐，可以是任意页；1 阶块以 2 页对齐，其起始页号必须是第 0 页、第 2 页、第 4 页……；2 阶块以 4 页对齐，其起始页号必须是第 0 页、第 4 页、第 8 页……依次类推到更高阶。用这种对齐方式简化了内存管理，大大加快了内存分配与释放内存的速度，也减小了内存管理的开销。但这种对齐也带来了负作用，可分配的最大块内存可能比实际存在的最大连续空闲内存小，比如第 4、5、6、7 页空闲的话，可以分配一个 2 阶块，但如果 3、4、5、6 页空闲而第 7 页忙的话，虽然有连续 4 页空闲，由于 3 不是 2 阶块的对齐边界，故只能分配 1 阶块。

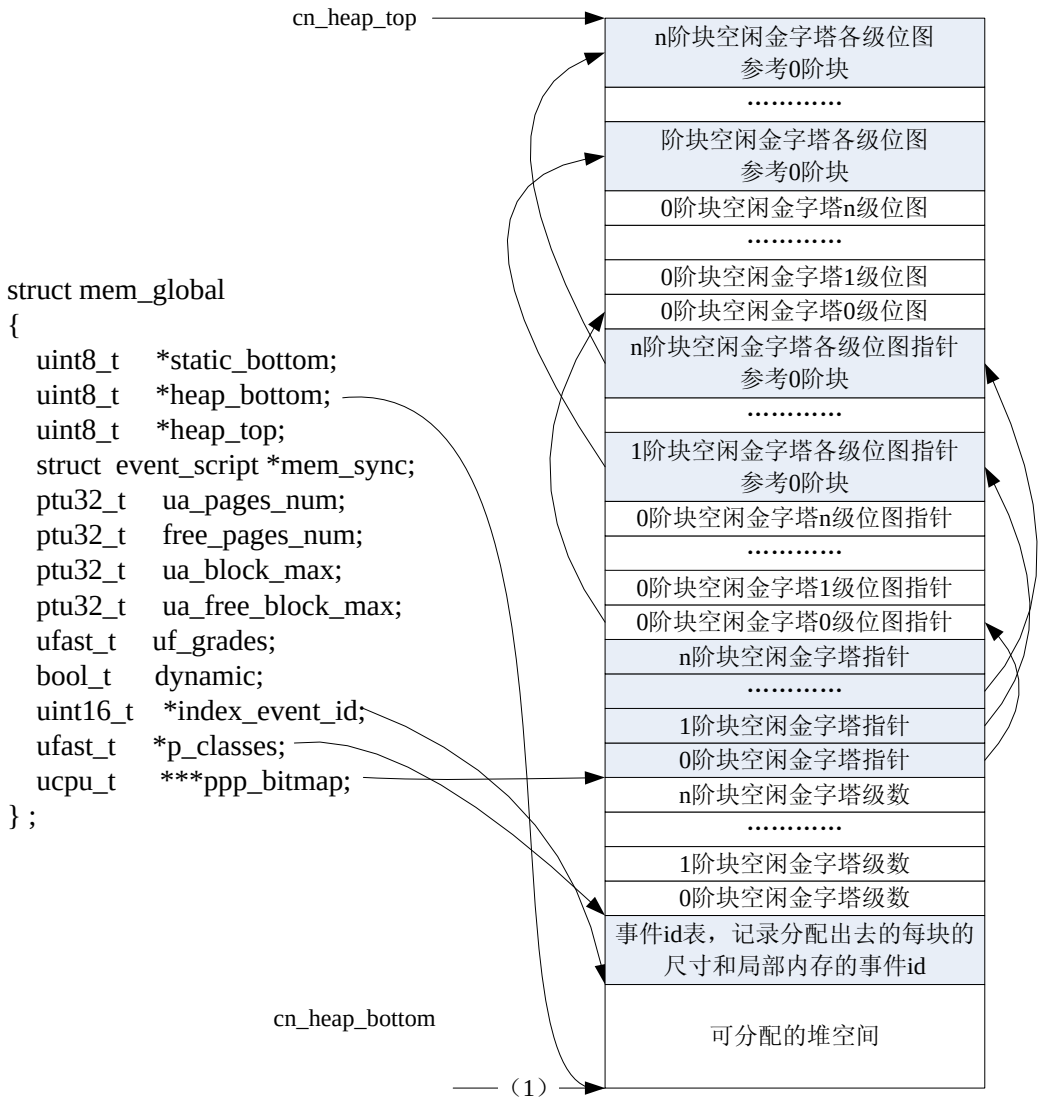


图 7-4 块相联内存分配数据结构

图 7-4 显示，除了按位访问的位图外，还有一个事件id表，该表记录的是每个内存块的拥有者以及该块内存的尺寸。事件id表是 16 位的，每页一项，记录格式与该块内存作用域有关（参见第 7.6 节），作用域有两种，一种是全局内存块，即该块内存不局限于一个事件的处理周期内访问，另一种是局部内存，局部内存由事件处理过程分配，只能在处理该事件时使用，事件处理完毕之前必须释放。如果线程没有主动释放，在事件处理完毕调用y_event_done函数时，操作系统将强制释放，这种强制释放是很消耗时间的，实时软件的设计者必须避免这种事情发生（参见第 7.6 节）。

如果是全局内存，不管是哪个事件处理过程中分配还是事件调度启动前分配，都无需记录拥有者，只需要记录该块内存的阶号，用阶号可以计算该块内存包含多少页，进而可以知道该块内存的尺寸。记录格式是：

1. 如果该块只含一页，记录为 0xfffd。释放内存时，读到被释放的地址所在页的事件 id 记录是 0xfffd，就知道该块只含 1 页。
2. 如果该块内存含 2 页或以上，则第一页的事件 id 记为 0xfffc，第二页记录该块的阶号，释放内存时，检查到被释放地址所在页事件 id 记录是 0xfffc，就读下一页的事件 id，得到该块的阶号。如果该块超过 2 页，剩余页的事件 id 被清零。

如果是局部内存，则除了要记录该块内存尺寸外，还要记录该块内存是被哪个事件申请的，以备该事件处理完毕时可以强制回收。记录格式是：

1. 如果该块只含 1 页，直接记录申请内存的事件 id。释放内存时，读到被释放的地址所在页的事件 id 记录<32768 则认为是块尺寸为 1 页的局部内存。
2. 如果该块内存含 2 页，则第一个页的事件 id 记为 0xffff，第二个页的事件 id 保存申请内存的事件 id。
3. 如果该块内存含 4 页或以上（不存在 3 页的块），则从第一个页的事件 id 起，依序记录：0xfffe+事件 id+阶号。

7.2.2 块相联分配模块初始化

参考 图 4-16，在操作系统和应用程序初始化完成后，启动多事件调度前，调用module_init_heap_dynamic（在mems.c文件中）初始化块相联分配模块，此后，用户再调用malloc族函数分配和释放内存，将不再按准静态分配方案而是按块相联方案分配内存。该函数的流程如 图 7-5 所示。

图 7-5 初始化块相联分配模块

7.2.3 分配一块内存

讲完存储器组织和初始化，就该到本章主题——分配内存了。分配内存的工作，简单地说，就是在如 图 7-2 所示的阶金字塔中找出一块符合要求的空闲内存，并返回给调用者。内存分配函数有两个，分别是m_malloc和m_malloc_gbl，前者分配局部内存，后者分配全局内存（参见第 7.6 节），两个函数除了记录事件id表的方法不同外，其他方面完全相同，这里以m_malloc函数为例讲讲实现过程。函数原型如下：

void *m_malloc(ptu32_t size,uint32_t timeout);

参数 size 是待分配的内存块尺寸，以字节为单位。

timeout 是超时时间，单位是毫秒，即如果没有合适的空闲内存块，则阻塞以等待内存可用，阻塞时间达到 timeout 仍然没有合适的内存块的话，就结束等待，返回 NULL。如果希望在分配失败时不等待直接返回，使 timeout=0 即可。

函数返回值是分配到的内存块指针。

函数代码如

代码 7-2 分配内存

```
void *m_malloc(ptu32_t size,uint32_t timeout)
{
```

```
uint8_t *ua_address;
ufast_t    uf_grade_th;
bool_t      en_scheduler;
void *result;
uint16_t *pl_id,id;
uint32_t page;

//启动多事件调度后，dynamic 成员不会再发生变化，即使并发访问也是安全的
if(tg_mem_global.dynamic==false)                                    //1
    return(__m_static_malloc(size));   //内存尚未初始化,执行准静态内存分配
//不能在此直接判断 size 是否满足,因为取得互斥量前可能发生切换而判断无效.
if(mutex_pend(&tg_mem_mutex,timeout) == false)                      //2
    return NULL;
en_scheduler = y_query_sch();
if((tg_mem_global.ua_free_block_max < size)
            && ((timeout == 0) || !en_scheduler))                   //3
{
    result = NULL;
}else
{
    if( !__check_memory(size,timeout) )          //没有合适的空闲内存块   //4
    {
        result = NULL;
    }else      //有合适的空闲内存块
    {
        uf_grade_th=__get_grade(size);          //取阶号                //5
        ua_address=__malloc_block(uf_grade_th);  //执行一块内存          //6
        pg_event_running->local_memory++;                               //7

        //阅读以下语句请结合 mem_global_t 中 index_event_id 成员定义的注释.
        pl_id = tg_mem_global.index_event_id;
        id = pg_event_running->event_id;
        page = (ptu32_t)(ua_address-tg_mem_global.heap_bottom)
                >>cn_page_size_suffix_zero;
        if(uf_grade_th==0)
        {//分配 1 页
            pl_id[page] = id;
        }else if(uf_grade_th==1)
        {//分配 2 页
            pl_id[page] = -1;
            pl_id[page+1] = id;
        }else
        {    //分配多页
            pl_id[page] = -2;
```

```
                    pl_id[page+1] = id;
                    pl_id[page+2] = uf_grade_th;
                }
                result = ua_address;
            }
        }
        mutex_post(&tg_mem_mutex);
        return result;
}
```

代码注释如下：

1. 如果动态分配（块相联方法）未初始化，执行准静态分配。在准静态分配程序中，如果判断到静态堆也没有初始化，将直接返回 NULL。
2. 请求互斥量，该互斥量不是用来保护堆的，而是用来保护用来管理堆的数据结构的。使用互斥量，说明内存分配支持优先级继承。
3. 如果没有合适的内存块，且不允许调度或者 timeout 为 0，则直接返回 NULL。
4. 检测并等待空闲内存块，如果有合适的空闲内存块，则本函数直接返回 true，如果没有，则阻塞直到有合适的空闲内存块或 timeout 时间到。
5. 用户提供的所申请的内存块尺寸是以字节为单位的、任意大小的，而块相联分配只能按块进行，这里把 size 向上调整为整块大小。
6. 分配一个规格化的块，这是内存分配的核心，本节后续有说明。
7. local_memory成员很重要，应用程序每申请一次局部内存，该成员就加 1，每释放一次，该成员就减 1，在y_event_done函数（参见第 4.3.11.3 节）中，正是判断这个成员是否为 0 以决定是否要强制释放内存的。

__malloc_block 函数是直接执行内存分配的，正是它从内存堆中挖出一块内存，如果说 module_init_heap_dynamic 函数 m_malloc 函数都是在排兵布阵的话，__malloc_block 是直入敌阵取其上将首级之举。函数原型：

void *__malloc_block(ufast_t grade);

grade是待分配的内存块的阶号。本函数代码比较长，函数流程如 图 7-6 所示，在这里就不列出代码了。函数在随书源代码中的mems.c文件中。

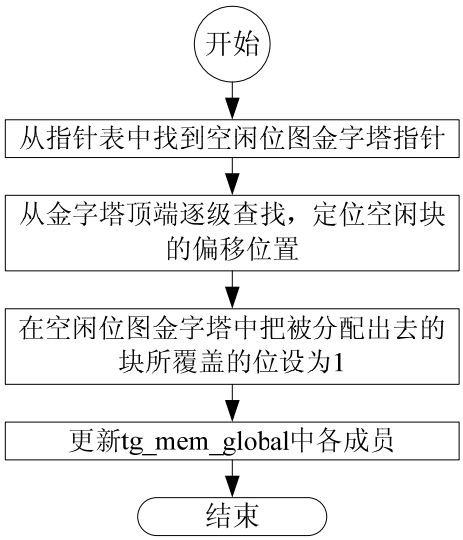


图 7-6 取一个空闲块

7.2.4 释放一块内存

无论是全局内存还是局部内存，都使用 m_free 函数释放，函数原型是：

bool_t m_free(void * pl_mem);

参数pl_mem是待释放的内存块地址，函数从内存控制块的事件id表中可以查到本块内存是全局内存还是局部内存以及内存块的尺寸，若是局部内存，还能查到拥有者的事件id，m_free函数据此进行释放内存的操作。函数流程如 图 7-7 所示，代码在mems.c中。

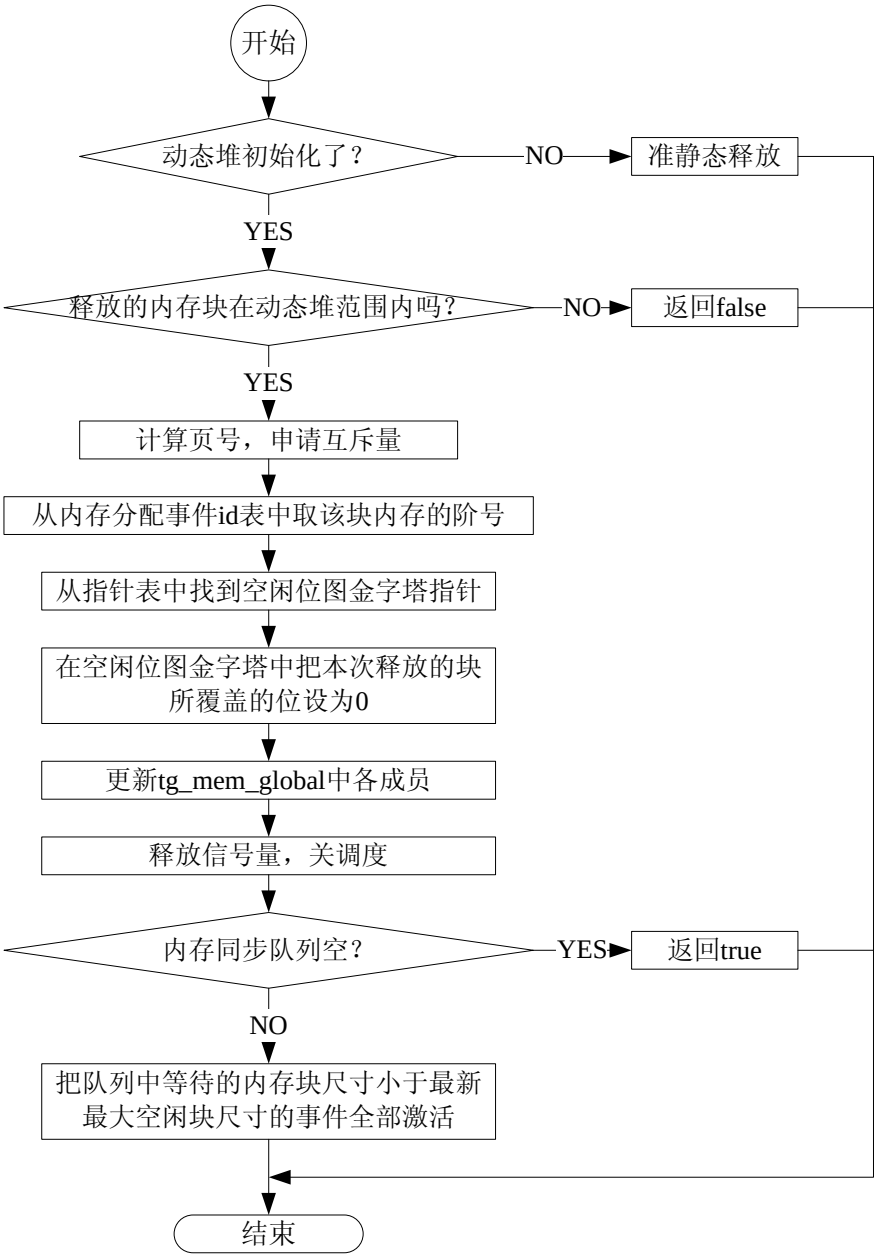


图 7-7 释放内存

7.3 固定块分配法

7.3.1 固定块分配模块初始化

djyos的内存管理系统支持 3 种内存分配策略，分别是“类静态内存分配”、“块相联分配”、“固定块分配”，3 种策略的使用时机条件各不相同。图 7-1 显式了不同时机执行的内存分配策略。“类静态内存分配”和“块相联分配”两种方法虽然相差甚远，但是对用户是透明的，用户将用同样的方法分配内存。

固定块分配法在第 2.9.3 节已经论及，这里讲述在djyos系统中是如何实现固定块分配法的，其代码在menb.c中。这种方法常用于链表结点的动态增加和删减，例如多机通信中动态地增加和减少子机数量。同一个内存池只能按一种块尺寸进行分配，但不同的内存池可以用不同的块尺寸分配，每个内存池由相应的内存池控制块记录，内存池控制块数据结构如下：

```
struct mem_cell_pool
{
    struct rsc_node memb_node;      //资源结点
    void    *continue_pool;         //连续内存池首地址，使用它可以增加实时性。
    void    *free_list;             //未分配块链表,单向,NULL 结尾
    struct semaphore_LCB memb_semp;
    ptu32_t pool_offset;            //连续池中的偏移量(当前地址)
    uint32_t cell_size;             //块大小,初始化时将按系统对其尺寸调整。
};
```

各成员解释如下：

memb_node：资源成员，内存池也是系统资源的一种，系统资源树中的名叫“固定块分配池”的根资源是内存池资源的根结点。

continue_pool：内存池管理的连续内存块首址。

free_list：空闲的内存块队列。

memb_semp：信号量控制块，每个内存池由一个信号量保护，内存池初始化后，该信号量的初始信号灯被设为内存池容量。

pool_offset：内存池下一次分配的偏移量，本成员专为优化 djyos 系统的实时性而设置，它使创建内存池函数的执行时间是确定的。

cell_size：内存池的块尺寸。

在 port_kernel.h 中定义的常量：

```
#define cn_mem_pools            (10)        //允许建立 10 个内存池
```

表示允许建立的内存池数量，在 memb.c 中定义了静态数组分配内存池控制块：

```
static struct mem_cell_pool tg_pool_of_cell_pool[cn_mem_pools];
static struct mem_cell_pool *pg_pool_of_cell_pool;
```

每创建一个内存池，就从 tg_pool_of_cell_pool 池中分配一个内存池控制块，内存池控制块本身也是按固定块分配法分配的。

module_init_memb函数初始化固定块分配法，程序如 代码 7-3 所示。

代码 7-3 固定块分配法初始化

```
bool_t module_init_memb(void)
{
```

```
    static struct mem_cell_pool cell_pool;                                  //1
    cell_pool.cell_size = sizeof(struct mem_cell_pool);                     //2
    cell_pool.continue_pool = tg_pool_of_cell_pool;
    cell_pool.free_list = NULL;
    cell_pool.pool_offset = (ptu32_t)&tg_pool_of_cell_pool[cn_mem_pools];
    pg_pool_of_cell_pool = &cell_pool;
    rsc_add_root_node(&cell_pool.memb_node,sizeof(struct mem_cell_pool),
                      "固定块分配池");                                       //3
    __sem_create_knl(&cell_pool.memb_semp,
                     cn_mem_pools,cn_mem_pools,"固定块分配池");             //4
    return true;
}
```

代码中各注释如下：

1. cell_pool 是一个内存池控制块静态局部变量，它既是内存池的根资源结点，又是用来管理内存池控制块池自身的内存池控制块。
2. 内存池控制块本身也是按固定块分配法管理的，但此时模块尚未完成初始化，故不能调用 mb_create 或者__mb_create_knl 函数来创建之，只能模仿__mb_create_knl 函数的行为初始化 cell_pool 的各成员。以下直到注释 3 完成模拟初始化工作。
3. 在系统资源链表中增加一个内存池控制块根结点。
4. 创建信号量，这个信号量保护的是内存池控制块池本身。

7.3.2 创建内存池

djyos 提供两个函数用于创建内存池：mb_create 和__mb_create_knl，函数原型是：

```
struct mem_cell_pool *mb_create(void *pool_original,uint32_t capacital,
                                uint32_t cell_size,char *name)
struct mem_cell_pool *__mb_create_knl(struct mem_cell_pool *pool,
                                void *pool_original,uint32_t capacital,
                                uint32_t cell_size,char *name)
```

这两个函数的区别在于内存池控制块的获取方法，前者从pg_pool_of_cell_pool内存池中分配，供一般程序设计者调用；后者内存池控制块由调用者提供，供操作系统模块调用。为什么要分成两个函数呢？这也是降低操作系统模块和其他模块之间的耦合度（参见第 11.2 节），cn_mem_pools 表示允许用户创建的内存池数量，如果操作系统模块从pg_pool_of_cell_pool池中分配，就占用了这个指标，应用程序与操作系统共享这个内存池，耦合随之形成。创建内存池的流程如 图 7-8 所示：

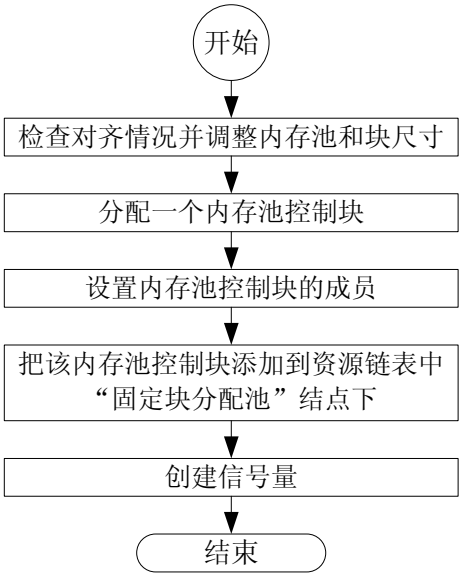


图 7-8 创建内存池流程

细心的读者一定已经注意到了，图 7-8 没有把内存池中的所有内存块用空闲队列串起来，而是保持内存池控制块的成员free_list=NULL。

7.3.3 分配内存

mb_malloc 函数从内存池中分配一块内存，函数原型：

void *mb_malloc(struct mem_cell_pool *pool,uint32_t timeout);

因考虑到实时性（不是快速性），mb_malloc函数比想象中的复杂一些，这种改变改善了实时性，但并没有改善执行速度，反而降低了一点点。程序如 代码 7-4 所示，代码中其他部分的注释已经很充分，这里着重讲一下else子句部分。这个子句是提高实时性的关键所在，在free_list空的情况下，直接从内存池连续字节池中取一块内存分配给调用者，因为前面已经取得了信号量，确保了内存池中至少有一块空闲内存，故无需判断pool_offset成员是否越界。这使得在创建内存池时无需用free_list队列把内存池中的内存块全部串起来，由于用户可能创建任意长度的内存池，把内存块串起来所花费的时间可能很长，且不确定，实时系统不喜欢不确定的操作。为什么说执行速度会降低呢？因为在前面已经获得信号量确保有空闲内存块的情况下，如果在创建内存池时建立了free_list队列，在mb_malloc函数中是无需进行if(pool->free_list != NULL)判断的。mb_malloc函数可能反复多次使用，而每次使用都多一次判断。所以说，确定性和快速性在这里是矛盾的，创建内存的函数获得了高确定性，付出的代价是，分配内存的执行时间稍微长一些。

代码 7-4 分配一块内存

```
void *mb_malloc(struct mem_cell_pool *pool,uint32_t timeout)
{
    void *result;

    if(pool == NULL)
        return NULL;
    //没有取得信号量，表明内存池空,这个信号量是保护内存池的，确保被分配的内存块
    //不超过内存池的容量
    if( ! sem_pend(&pool->memb_semp,timeout))
```

```
        return NULL;
    //下面的关调度(异步信号)是保护内存池控制块的，为什么不用互斥量呢？因为请求
    //互斥量的过程也要关调度，而且关调度的时间还比这里长。
    //注:从 sem_pend 到 int_save_asyn_signal 之间发生抢占是允许的，因为信号量已经
    //取得，其他事件不可能把内存块分配光。
    int_save_asyn_signal();
    if(pool->free_list != NULL)        //空闲队列中有内存块
    {
        result = pool->free_list;     //取空队列表头部的内存块
        pool->free_list = *(void**)(pool->free_list);   //空闲队列下移一格.
    }else                              //空闲队列中无内存块，从连续池中取
    {
        pool->pool_offset -= pool->cell_size;   //偏移地址调整
        //分配偏移地址处的内存块
        result = (void*)pool->pool_offset;
    }
    int_restore_asyn_signal();
    return result;
}
```

7.3.4 释放内存

mb_free 函数从内存池中分配一块内存，函数原型：

void mb_free(struct mem_cell_pool *pool,void *block);

参数 block 是被释放的内存块，pool 是释放到的目标内存池控制块。

本函数很简单，就是检查参数的合法性，然后把内存块挂在 free_list 队列中。

特别要注意的是，如果重复释放同一块内存，将可能造成 free_list 队列断裂，造成不可预测的后果。

7.4 堆和内存池

djyos系统中，堆是全局管理的，各线程向操作系统提出的内存申请都从堆中申请，所有线程的公用资源，内存池则由功能模块或者线程管理，操作系统的内存池管理模块提供从池中分配和释放内存的机制，比如泛设备驱动模块管理设备控制块内存池、看门狗模块管理看门狗控制块内存池。内存池的获得方式有两种，一是定义数组，由编译器分配；二是用m_malloc族函数从堆中分配。内存池被划分为一个个固定大小的块，使用固定块分配策略分配，而且每次只能分配一块内存。图 7-9 是djyos系统的动态内存管理示意图。

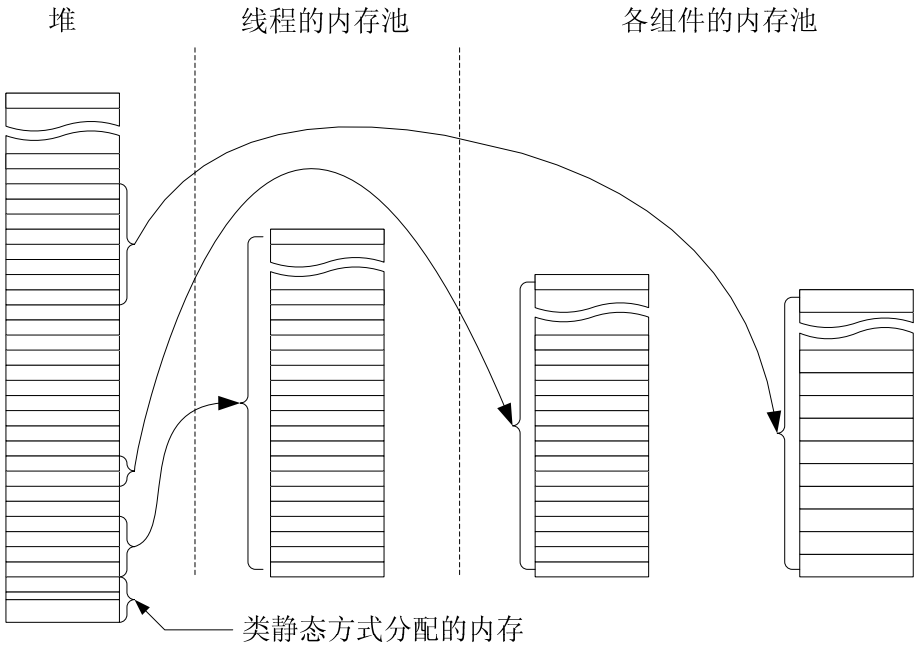


图 7-9 djyos 动态内存管理结构图

1. 当需要创建线程时，操作系统从全局堆中为线程分配线程运行所需要的栈；线程向操作系统申请内存时，操作系统也从全局堆中分配。由于操作系统事先不可能知道线程运行栈的大小，也不知道用户线程动态申请内存的大小，所以，操作系统必须满足任意尺寸的连续内存需求。全局堆使用块相联分配方法，全局内存被划分为固定大小的页，动态分配内存的最小单位是页，也可以分配连续页，连续页的页数必须是 2 的整数次幂。图 7-9 中有一个线程从全局堆中申请了一块大小为 4 页的内存，另外还有其他组件申请了两块内存，大小分别是 2 页和 8 页。
2. 线程从堆中申请内存后，可以当作本地内存池进行二次管理，djyos的内存池管理使用固定块分配方法（见第 7.3 节），每次只能分配 1 块内存。

7.5 内存同步

内存同步指的是事件从堆中分配内存时，由于找不到合适的内存块，事件暂停处理直到有合适的空闲内存块。

djyos系统提供两种内存管理手段，一是用于管理堆内存的，堆属于整个系统的核心资源，使用块相联分配策略管理；二是用于管理内存池的，使用固定块分配策略管理，属于组件级的管理，被管理的内存可以从堆中分配，也可以用数组静态分配。内存同步针对的是从堆中分配内存，从堆分配内存失败时，事件将加入内存同步队列，暂停以等待有其他事件释放内存。从内存池中分配内存失败导致的同步是通过信号量同步实现的（参见第 5.7 节）。内存同步队列如 图 7-10 所示，该队列是一个以所请求的内存尺寸排序的队列。

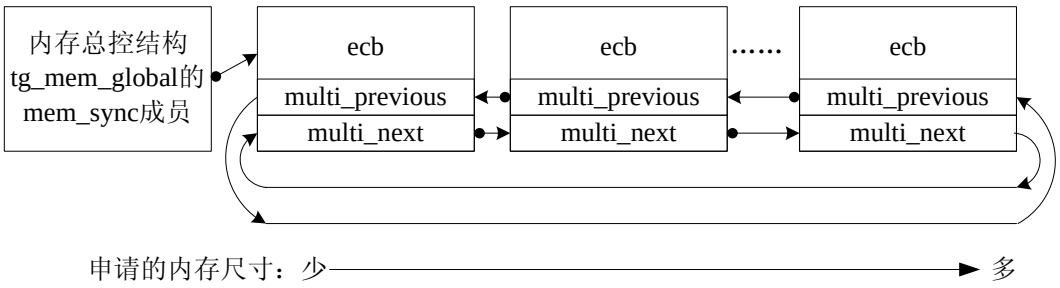


图 7-10 内存同步队列

创建虚拟机时被内存同步阻塞

如果在为某事件创建虚拟机时内存不足，该事件将被内存同步阻塞。这是一种特殊的同步，其他同步手段的被阻塞的事件都是正在处理的事件，换句话说，就是已经开始处理的事件。而内存同步的同步事件可能还没有开始处理，从第 4.3.11.2 节可知，事件必须获得虚拟机资源才能够被处理，而创建虚拟机需要从系统堆中分配内存，此时就可能被内存同步所阻塞。在__y_select_event_to_run函数（参见 代码 4-12）中，如果为就绪队列中的事件创建虚拟机失败，就将该事件从就绪队列取出，就调用__wait_memory函数直接把事件插入内存同步队列中，且不设置超时退出，只要内存一直不满足要求，就永不激活。在这个过程中，没有发生上下文切换，因为被阻塞的事件尚不拥有线程，根本就没有上下文，这也是唯一一个不发生上下文切换的阻塞过程。

事件处理中被内存同步阻塞

事件处理过程中从系统堆中分配内存不能满足，如果 timeout 不为 0，将被内存同步阻塞。djyos 有两个内存分配函数：m_malloc 和 m_malloc_gbl，前者分配局部内存，后者分配全局内存，两个函数均可能导致事件被内存同步阻塞。内存分配的结果可能有三种：一是成功分配内存，返回获得的内存块指针；二是调用时没有满足需求的内存块，经阻塞后，由于其他事件释放内存而成功分配，返回指针；三是被阻塞，超时返回 NULL，包括超时常数被设置为 0，因没有合适内存块，不经阻塞直接返回的情况。__check_memory 函数判断有没有合适的空闲内存块可供分配，没有的话就阻塞事件。

7.6 局部内存回收

djyos系统提供内存资源回收功能，可以在很大程度上防止内存泄漏。每个事件处理完毕之时，操作系统均要检查该事件是否有未释放的局部内存（参考第 7.2.1 节的最后部分可知什么是局部内存），如果有，则调用m_cleanup函数强行释放之。回顾一下第 7.2.1 节的最后部分，djyos的内存管理器用一个事件id表来描述每一页内存的分配情况。__m_cleanup函数的原型是：

void __m_cleanup(uint16_t event_id);

实现内存回收的方法，就是扫描整个事件 id 表，看有没有指定事件分配的局部内存，如果有，就登记“内存泄漏”错误，并释放之。所以，任何线程，调用 malloc 函数从堆中申请内存时，如果希望该内存在事件结束后仍然有效，则务必按全局内存申请。事件控制块中的 local_memory 成员，事件每申请一次局部内存，local_memory 就加 1，每释放一次就减 1，在 y_event_done 函数中，检查有没有内存泄漏的语句：

```
if(pg_event_running->local_memory != 0)
```

```
{
    __m_cleanup(pg_event_running->event_id);
}
```

因此，内存回收只针对堆，并不针对采用固定块策略分配的内存池。

7.7 djyos 内存管理的特点

1. 安全性高

有许多内存管理系统在动态内存分配时，需要在分配给用户的内存块的相邻位置保存一些数据，内存管理系统利用这些数据管理动态内存分配，如果这些数据被破坏，将导致系统崩溃。由于保存这些数据的地址正好与用户可使用的内存区域相邻，如果用户程序有有 bug，比如内存溢出，首先遭殃的是与之相邻的内存数据，这些数据被破坏后，释放内存时便不知道该释放多少内存，可能导致其他线程正在使用的内存被错误地释放掉，发生不可预知的后果，甚至造成系统崩溃。因此，这样分配分配方法可靠性不高，用户错误容易造成整个系统崩溃。djyos 系统的动态内存管理器则没有这个问题，djyos 的整个内存堆都是保留给用户使用的，它从不会在分配给用户的内存块前后存放数据，即使用户内存溢出，也不会破坏内存管理系统，free 时仍然能正确地释放内存。

2. 有防止内存泄漏机制

如果线程使用完动态内存后，不及时释放，就会导致内存泄漏。djyos 系统记录了每个线程使用动态内存的记录，djyos 中，每当事件处理完成，相应的线程就会被杀死。djyos 系统在杀死线程之前，会强行释放该线程占用的动态内存。这种内存管理方式有利于减少内存泄漏。

3. 实时性高

djyos的系统内存管理实用块相联分配法（见 2.9.4 节），内存被划分为页进行分配，允许分配连续的、大小为 2 的n次方页的内存块。djyos实用金字塔式的空闲块管理方式，查询任意尺寸的空闲内存，都可以在有限的时间内快速完成（见第 7.2.1 节），在 32 位机上，即使从 4G寻址空间搜索一个 4K的内存页，所花费的时间也能限定在（5 次比较+5 次乘法+5 次加法）以内。

4. 低碎片率

djyos 支持“类静态内存分配策略”，有效地防止了产生永久碎片。以单页和 2 的整数次幂个页为单位分配内存，有效地防止产生细小的碎片，当然，这也付出了浪费内存的代价。在搜索空闲内存时，总是从内存的一端开始，降低了内存分割的随意性。

第8章 资源管理

设计数据结构是软件中非常重要的一个环节，许多应用程序都面临如何组织和管理数据的问题，广义上，应用程序需要的一切数据都是资源，djyos 操作系统系统提供一个简单易用的资源管理模块，用于管理用户比较关注的、对程序运行意义重大的资源，并提供了丰富的资源管理 api 函数。在 djyos 资源管理模块的协助下，应用程序管理数据将变得非常简单。更主要的是，如果用户坚持用资源管理器管理所有数据，将使不同的软件模块实现细节上获得惊人的一致性。一个有一定规模的软件，必然要划分为若干模块才能顺利开发，如果不加约束，各模块可能各自设计自己喜欢的方法管理各自的数据。这种不一致性将增加软件的复杂度，增加阅读和修改软件的难度，而且，这样不可避免地会增加软件规模，增加编程和测试的工作量。若要在开发团队内和团队间人员调配，新人将花较多的时间熟悉新模块，而如果各模块代码有一致性，就会降低人员调配时交接工作的难度。

事件类型控制块表和事件控制块表是两类特殊的资源，dyos 操作系统的所有调度都是围绕这两类资源展开的，至于要不要把他们纳入资源链表进行管理，笔者曾有过激烈的思考，甚至代码也为此做过大规模的调整，最后权衡利弊后，还是决定放弃。事件类型等纳入资源链表的好处是显而易见的，事件和事件类型管理需要大量的链表操作，这些操作如果纳入资源链表模块进行，将使代码更加简洁，调试软件也可以用统一的方法浏览事件、事件类型和其他资源节点，也能简化调试软件的开发。所有和链表相关的操作按照相同的方法进行，还可以减少 bug 的可能性。坏处也是有的，因为事件类型和事件是操作系统的核心数据结构，这两个表任意一个被破坏就意味着系统崩溃，他们对保证系统安全有着举足轻重的作用。而资源链表是一个开放的链表，虽然操作系统不允许用户直接操作这个链表，只能通过操作系统提供的 api 调用操作，但毕竟，资源链表中的数据对操作系统安全所担负的责任远比事件控制块表、事件队列和事件类型控制块表低，把安全等级相差很大的数据放在一个链表中管理，总让人放心不下。权衡再三以后，还是把事件控制块表、事件队列和事件类型控制块表独立管理处理。

8.1 资源管理模型

djyos 推荐的软件模式是，通过良好的模块划分，使模块与模块之间完全不可见，软硬件系统中的任何一个功能模块，都看不到别的模块的实现细节，只看到一个个资源或者设备。我们可以这样描述 djyos 系统：操作系统就象大地，大地上是一个个的果园，果园里种了许多果树，果树上结了许多果实。根结点就是树根，资源树就是果树，而资源就是一个个果实。果园管理员相当于软件项目的系统工程师，他负责安排在果园里应该种些什么果树以及由谁（程序员）去种，统一安排采摘果实的标准方法（模块间的接口）；一个程序员就是一个果农，他们的工作就是用自己的方法浇灌自己的果树，他们不需要关心其他果农如何种树，只需要知道按照果园管理员的安排的标准方法送出自己的果实，也用同样的标准方法采摘其他果农的果实。djyos 一个资源链表管理系统的所有资源，该链表是开放的，操作系统提供了一组 api 调用，帮助用户实现自己的资源管理。用这组 api 调用，用户可以在大地上种植和浇灌资源树，修剪树枝、采摘果子。

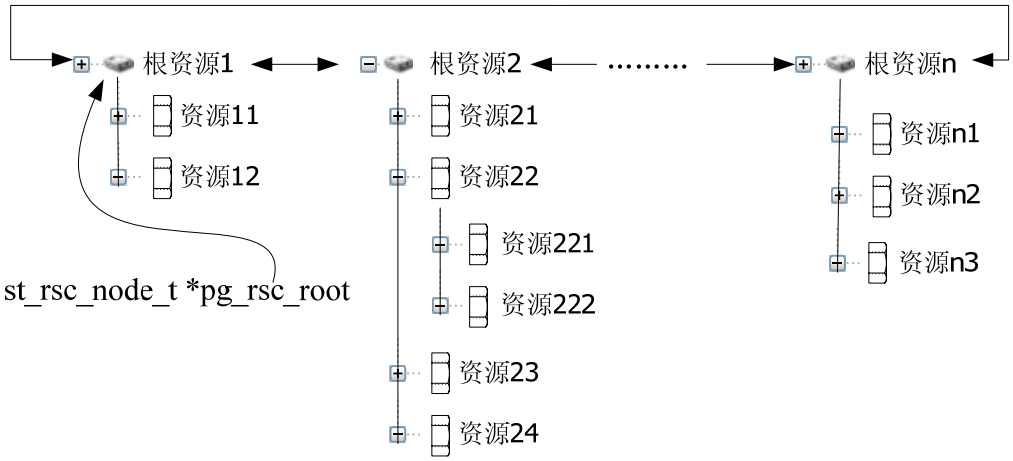


图 8-1 djyos 资源结点示意图

在 图 8-1 中的根结点和资源结点的数据结构是相同的，定义如下：

```
struct rsc_node
{
    struct rsc_node *next,*previous,*parent,*child;
    uint32_t    node_size;      //包含 node 的数据结构的尺寸，用于调试
    char *name;
};
```

next、previous、parent、child四个指针构成如 图 8-2 所示的资源链表节点，每个资源节点代表一个资源。node_size 是为调试准备的成员（详见第 9.5 节）。name是资源的名字，名字因资源用途不同而不同，用于文件系统则是文件名或者文件夹名，用于设备是设备名，用于gui则可能是窗口名……，当然，资源也可以没有名字。资源节点名字的命名规则：

1. 资源名可以是但不限于任何合法的 C 语言字符串，名字中不能包含回车符、换行符、字符'\'、字符\0。允许中文字符串，也允许中英文混合字符串。
2. 路径名是从根节点名称开始沿着资源链表逐级索引到目标资源的所有父节点的名字串联而成，各级资源之间用字符'\'隔开。例如 图 8-1 中“资源 222”的完整路径名是“根资源 2\资源 22\资源 222”。
3. 资源路径的组成与 windows 系统的文件路径有点相似，但 djyos 中路径本身也可以是完整的资源，例如图中的”资源 22”是一个资源实体，而 windows 文件系统的路径名只能是文件夹。

为便于分类管理，不同类型的资源应该有不同的根节点，并把同类的资源作为根节点的子树，例如，在设备管理中，所有设备组成一颗大树，这颗树的每个资源结点代表一个设备，pg_device_root 指针指向设备树的根节点；在 gui 设计中，同一个显示器下的所有窗口组成一颗大树，这颗树的每个资源结点代表一个窗口，pg_windows_root 指向窗口树的根节点。

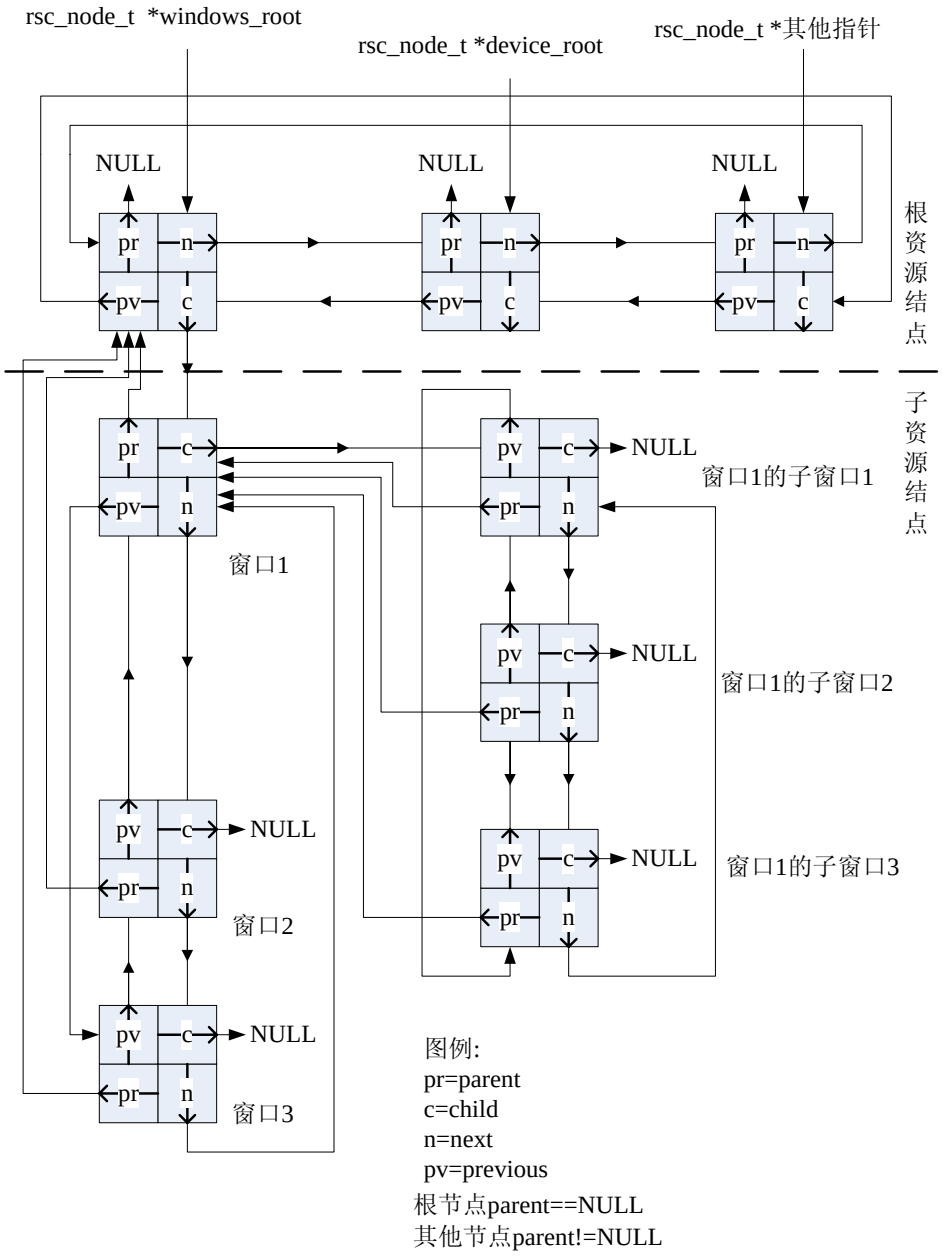


图 8-2 djyos 资源链表示意图

1. 在同辈结点中，较靠近父结点的为兄结点，远离的为弟结点。
2. 由父结点直接指向的为长子结点，在其同辈结点中也称为长兄结点。
3. 同辈结点的最后一个结点称为小弟结点，因是循环链表，满子结点同时也在长子结点的前面。这个结点也是父结点的满子结点
4. 所有结点的父指针均指向其父亲结点，如果没有子结点，则子结点指针为 NULL。根结点的父指针为 NULL。
5. 操作系统只提供 api 供调用，并不管理资源链，对资源的组织和管理完全由使用者负责。

8.2 宿主数据结构

就像一个单纯运行操作系统的计算机是毫无意义的一样，一个单纯的资源结点也是毫无

意义的，资源结点必须依附在宿主数据结构上，通过宿主数据结构实现该结点的功能。宿主数据结构也可以认为是资源结点的负载，通过资源链表，程序员可以实现对负载的有序化管理。例如操作系统的信号量管理就是按照资源链表进行管理的，信号量的定义如下：

```
struct semaphore_LCB
{
    struct    rsc_node node;    //第一个成员必须是资源结点成员
    uint32_t lamps_limit;     //信号灯数量上限，0 表示不限数量，即总不会阻塞
    uint32_t lamp_counter;    //可用信号灯数量。
    uint32_t lamp_used;       //该信号灯被请求的总次数，次数溢出时将环绕到 0
    struct    event_script *semp_sync;       //等候信号的事件队列，优先级排队
};
```

在都江堰操作系统中，所有信号量组成一棵信号树，每个信号量就是信号树上的一个结点，从而实现对信号量的集中管理。

在下一章中我们可以看到，作为都江堰操作系统最重要的模块之一——泛设备管理模块，就是通过资源链表进行管理的。

8.3 使用资源链表

djyos 的资源管理子系统可以帮助应用程序组织数据，下面以字符终端使用的字符串资源为例，说明如何使用资源链表。这个例子中要建立一棵资源树，该资源树包含 50 个字符串，每个字符数组的长度等于终端宽度。过程如下：

第1步：　定义资源结点数据结构

```
#define cn_terminal_lines       50
#define cn_terminal_width      40
struct text_line
{
    struct    rsc_node    node;
    char string[cn_terminal_width+1];
};
struct text_line tg_text_line[cn_terminal_lines];
```

程序定义了资源结点的 struct text_line 数据结构，数据结构包含一个 rsc_node 类型的数据数据和一个字符串，出于优化访问代码的需要，node 必须是数据结构的第一个成员。这样使 tg_terminal_line [n]与 tg_terminal_line [n].node 有相同的地址，可以直接用强制类型转换访问。

随后定义了构成该资源链的数据数组 tg_text_line。

第2步：　建立根结点

建立根结点的过程如下：

1. 定义文本串资源的根结点指针，这个指针对以后往这棵资源树上添加子结点（文本行）很有用：

   ```
   struct text_line * pg_text_line_root;
   ```

2. 定义文本串根结点数据结构

   ```
   static struct text_line    str_root;
   ```

3. 建立根资源节点，"terminal text resource"是根结点的资源名字。

```
pg_text_line_root = (struct text_line*)rsc_add_root(
                    &str_root.node,sizeof(struct terminal),"terminal text resource");
```

使用资源树前，先要把树根埋到地里，rsc_add_root函数可以完成这项工作。建立根结点就是在 图 8-2 所示的资源链表中增加一个根结点。这个结点可以是纯粹的struct rsc_node类型的数据，但推荐使用该资源树子结点相同的数据类型。

第3步： 使用资源树

建立根结点后，就可以使用资源树了，可以进行包括添加结点、删除结点、查询结点、遍历资源树等各种操作。下面这段代码把字符串资源数组中的所有元素连接到资源树的根结点 pg_text_line_root 下面。大家注意到了，所有的字符串资源都命名为"terminal text line"，这是资源名，而不是结点所包含的字符串数据，字符串数据由 tg_text_line [lines]. string 成员访问。操作系统并不禁止资源重名，你甚至可以不给资源取名字，资源命名方面的管理完全由使用资源的程序负责。字符串资源对命名没有限制，但是其他一些应用可能会有所限制，比如设备驱动程序中，就不允许兄弟设备同名，文件系统中，也不允许同一个文件夹下的文件重名。

```
for(lines = 0;lines < cn_terminal_lines; lines++)
{
    rsc_add_son((st_rsc_node_t*)pg_text_line_root,&tg_text_line[lines].node);
    tg_text_line[lines].node.name = "terminal text line";
}
```

第9章 泛设备驱动

现代嵌入式产品设计中，谁的可移植性做得好，谁就能在技术上占得先机，djyos 的泛设备驱动模块可以帮助工程师轻松构建可移植的软硬件功能模块。

9.1 模块泛设备化

传统的软件驱动硬件设备工作有两种方式，一种是由应用程序直接操作硬件，另一种是通过操作系统提供的设备驱动接口操作硬件。

应用程序直接操作硬件的方法相当灵活，所有的操作方式都由程序员决定，而且效率可以很高，缺点是设备的管理变得困难。当由多个软件模块同时需要使用设备时，需要由程序员处理所有可能的争用冲突问题。例如一个串口驱动程序，模块 A 发送“12345”，由于串口属于低速设备，模块 A 一般把字符串写入发送缓冲区，不会等待串口发送完成才继续运行，这时候如果系统切换到模块 B，而模块 B 不知道串口这在执行发送工作，又启动发送“ABCDE”，如果处理得不好，接收方很可能得到“123ABCDE”，这需要程序员极为小心地避免这样的问题。由于这种方式可能会使对硬件的操作直接混杂在应用程序代码里，而硬件的不确定性造成软件可移植性的降低，当硬件被修改时，可能被迫修改应用程序。但是在某些特殊条件下，这种方式也有其优势，在一些高速高精度的控制中，显然从中断服务函数直接控制硬件设备能够获得更高实时性。

另一种驱动方式与具体操作系统相关，由操作系统定义的一组标准接口和代码，用户按照操作系统要求编写代码控制硬件设备。不同的操作系统对设备驱动的模式可能完全不一样，但有一点是相通的，即所有的操作系统都会构造一个标准的驱动软件框架，提供了框架内共同的代码，根据设备类型的不同，框架也可能有所区别，但本质是一样的。在标准框架外面，还会提供两个接口，一个是与应用程序交互的接口，应用程序通过这个接口就可以实现对硬件的操作。另一个接口是驱动称序对硬件的接口，这个接口完成控制硬件和从硬件获取数据，操作系统一般会提供常见硬件的接口代码，如果用户的硬件比较特殊，就需要由用户编写接口代码，这也是整个驱动程序体系中唯一需要由用户编写的代码。通过这两个接口的包装，应用程序看见的和访问的都不是直接的硬件，而是操作系统提供的接口，程序员也就无须关心具体的硬件了。如果硬件被修改，只要修改驱动程序对硬件的接口程序就可以了，只要应用程序接口不变，那么应用程序可以不修改而直接移植。

djyos 系统则更进一步，设备驱动程序被赋予了更广泛的含义，它是被设计成软件功能模块间互相访问的接口，而不再仅仅是访问硬件的接口。从软硬件联合设计的角度，djyos 系统并不区分软件模块还是硬件模块，如果完整产品由多个模块组成，任意一个模块在别的模块“眼里”都是以设备的形式出现的，使用设备的模块并不知道该设备的实现细节，也不知道该设备是由硬件组成的还是由纯软件组成的。某一个功能由软件实现还是硬件实现并不重要，关键的是，它实现了需要的功能，并且为别的模块提供了相同的访问和操作接口。

我们来看看这样做有什么好处，就目前的技术而言，电子系统中有许多功能既可以直接用硬件实现，也可以用软件实现，系统设计时可以根据实际情况灵活选择。既然这样，我们还需要在设备驱动方面区别软件模块还是硬件模块吗？例如某产品中有一个数学模块，该模块需要一个计算行列式的值的功能，当 CPU 系统的速度和内存允许时，可以编写一个行列式求值的函数 determinant( )，数学模块直接调用这个函数就可以了；如果 CPU 负担很重，已经没有时间计算行列式，又受限于各种条件不能更换更快的 cpu，那么就可以用 FPGA 设计一个专门计算行列式的器件，这时它就是硬件。我们需要抛弃函数 determinant( )转而为这个设备编写一个驱动程序，然后修改数学模块，要求其通过驱动程序提供的接口去访问这个硬件吗？同样是完成计算行列式的功能，其实现方法不同时，按照模块独立且内闭的原则，不应该引起其他模块的变化，数学模块不应该因“行列式求值器”的实现方式不同而有所不同。而这个例子中，行列式模块的内部实现的变化确实使数学模块的代码发生了变化，这是不应该的。有什么办法可以使数学模块保持一致呢？djyos 系统的泛设备驱动为你提供了答案，数学模块只看到一个“行列式求值模块”的存在，并且可以通过泛设备驱动程序去访问它，无论该模块是用软件还是硬件实现，驱动程序接口都是一样的，数学模块根本就不知道行列式求值功能的实现细节。djyos 操作系统中设备驱动程序的定义是：操作系统为了实现自身和用户应用软件标准化，同时又能满足不同用户的使用习惯，用于操作系统模块和用户模块间、用户模块与用户模块之间互相访问的标准接口模块。djyos 的设备驱动程序模糊了软件和硬件的差别，不再专门为硬件服务，因此，我们把它叫做“泛设备驱动程序”接口。图 9-1 是djyos系统中泛设备driver模型的抽象图，图中两个模块互为设备，他们通过泛设备driver交换数据，泛设备有两套用户接口，我们形象地把他们命名为左手接口和右手接口。设备就象人一样有两只手，模块A跟设备人的左手接口打交道，他只看见设备人的左手，通过左手接口的left_read函数从设备获取数据，通过left_write函数把数据写入设备，left_ctrl函数则实现对设备的控制。模块B则与设备人的右手接口打交道，方法与模块A一样。两个模块都是设备的用户，与设备的左手接口打交道的模块叫左手用户，另一个模块称为右手用户，划分左右手用户由系统设计师统一管理，操作系统对此并没有严格的规定，通常按照如下约定划分，这是个非强制性的约定。

1. 如果设备被用于软件模块与硬件模块通信时，靠近软件模块的是左手接口，而靠近硬件模块的是右手接口。
2. 如果设备用于软件模块与软件模块双向通信，左右设备由系统设计师规定，由设备 driver 作者实现。
3. 如果设备用于软件模块与软件模块间单向通信，则输出数据的模块是左手用户，接收数据的模块是右手用户。硬件与硬件单向通信时也遵守这个规则。
4. 设备的左右手接口一经确定，就要维持稳定，不得再更改，即使将来由于系统优化使左右手划分不再符合以上约定（比如系统中的硬件模块被改为软件模块而使其不符合第一条约定）也不得修改。

这种划分习惯仅仅是习惯而已，并不具备约束性，由于 djyos 系统中软件和硬件模块是等价的，在实际应用中可能发生更改，比如原来由硬件实现的模块，可能改成用软件实现，或者反之。这时候，不应该更改相关设备 driver 代码的左右手定义。因此，程序员使用设备 driver 时，必须严格按照设备 driver 的说明书分辨左右手，不能想当然地认为硬件装置肯定是右手用户，或者见到软件模块就认为是左手用户。

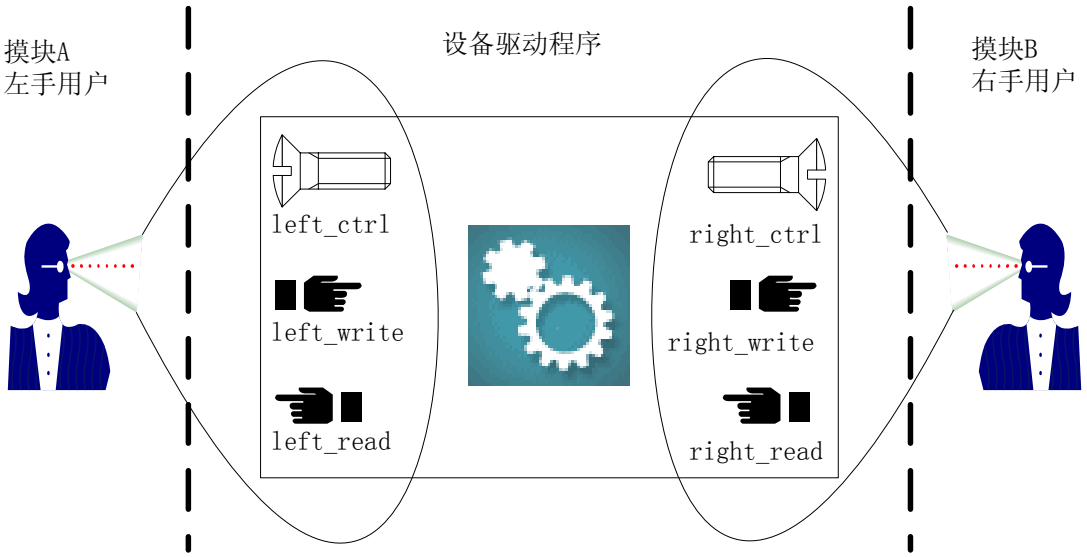


图 9-1 djyos 的设备模型

1. 系统设计确定了两个模块，模块 A 和模块 B 之间需要交换数据。
2. 系统设计还确定了一个设备 driver，这个设备负责两个模块间交换数据。
3. 两个模块可能由不同的团队开发，模块 A 和模块 B 的开发团队互相并不知道对方的存在，只要他们共同遵守 dirver 的接口规则，他们之间就能正常交换数据。
4. 模块 A 和模块 B 所面对的都是设备 driver，模块 A 面对设备左手接口，模块 B 面对右手接口，左右手的设定是由系统设计师按照一定的规则规定的。两个模块按照设备 driver 说明书规定的格式输入和输出数据。

图 9-2 显示了设备driver在djyos应用程序中的使用情况，设备driver的用途非常广泛，它甚至允许driver作为硬件模块与硬件模块之间交换数据的桥梁。

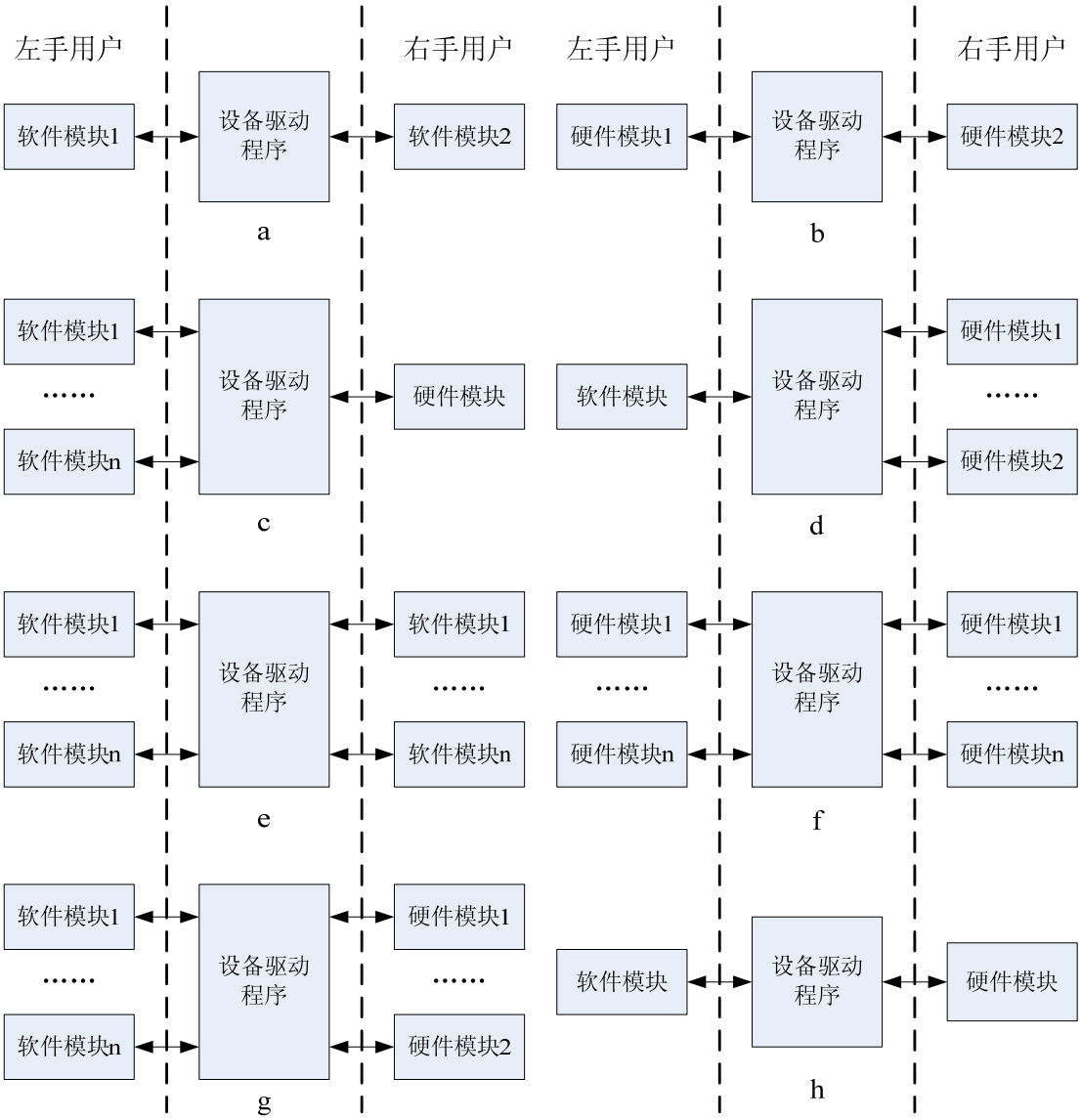


图 9-2 泛设备 driver 与产品功能组件的组织结构

1. 在设计之初，除硬件与硬件交换数据的情况外，硬件模块总是右手用户。除软件与软件交换数据的情况外，软件总是左手用户。但并不绝对，如果系统设计完成后，更改模块软硬件实现方式的配置，设备的左右手设定一般不再更改。
2. （a）是软件模块与软件模块之间交换数据的模型，用于降解两个模块之间的相关性，降低他们之间的耦合度。
3. （b）是设备 driver 用于硬件模块与硬件模块交换数据的情形，这种模式特别适合于做通信转发器。
4. （c）中有多个软件模块需要操纵硬件模块，设备 driver 应该协调各软件模块，保证硬件模块的时序是完整性。

设备driver的功能从软硬件接口模块变成产品功能组件之间的接口模块，这个小小的观念转变，不就是换个说法嘛，这样做有什么意义吗？我们先放松一下，看个真实的笑话：1999年的时候，香港有一个小伙子，一手拿着众多报道“千年虫”危害性的报纸，一手拿着一包白色粉末，向一些老太太兜售“千年虫”特效杀虫剂，还煞有介事地说全香港就他有这个药，有一个老太太就这样给骗走了 10 多万港币，后来证实这些白色粉末只是面粉而已。这就是著名的计算机“千年虫”问题，它不仅使全世界计算机界蒙受天文数字般的损失，还有一个

无辜老太太的养老金！“历史有无穷的力量”，“千年虫”是计算机发展早期为节省昂贵的存储器而留下的历史问题。然而，生产嵌入式产品企业，在扩展产品系列和产品升级换代的过程中，要面对许多历史问题，这些问题大多数是因为设计之初考虑不周造成的，一个存在问题的软件和硬件模块，经过广泛发布后才发现错误，你将欲改不能，改正它，将使所有用户手册、设计手册、用户服务经验等全部作废，还可能使与发生故障的模块互相关联的其他模块不能工作。你将眼睁睁地看着自己一手制造的错误日益扩散，你的老用户需要继续忍耐，你还要忍受日益增加的新用户的指责，一切你都无能为力！怎样才能改善这种状况呢？姜太公的鱼钩是钓不到鱼的，要求软件和硬件设计没有缺陷也是不可能的，即使今天看起来没有缺陷的设计，难保明天就会发现缺陷。这时候，你一定会想，如果各个模块能够互相独立修改和升级，而不影响其他与之交互的模块，那该多好啊！djyos系统希望帮助你摆脱这样的困境，回忆一下 第 11 章的积木化支持，模块结合泛设备管理功能，把各独立的模块用泛设备封装起来，就能使各模块可以独立修改。

在软件开发领域，我们还会碰到以下 2 种尴尬：

1. 许多企业起步时往往都没有什么编程规范，软件一般由几个高水平的天才完成，往往是软件实现技巧非常高明，但由于缺乏规范和标准，但软件的接口往往不好。当企业发展壮大后，软件开发就会规范化，这些早期软件就显得有些非驴非马了。它一方面是老一代程序员的心血结晶，而且确实有很大的价值；另一方面，它又很难与新软件配合使用。把他们作为包袱背上吧，软件的规范性就会受到破坏，让系统很别扭；如果舍弃这些代码，实在有些可惜。
2. 很多企业开发产品时会利用开源代码或者购买商业化的中间件来加快产品开发，也确实有许多开源代码是非常优秀的，许多开源代码有一些组织在维护并不断升级。但是这些代码的书写格式以及编程规范往往与企业的规范和程序员的习惯不一致，如果直接与公司的其他代码揉在一起，势必会破坏代码的一致性，轻则导致书写风格的不统一，重则会使代码的接口规范遭到破坏，使其他代码削足适履地迎合这些开源代码。重写这些代码以使其符合规范也是不明智的，因为重写会导致潜伏 bug，而且不能享受开源代码维护者升级的好处。

在 djyos 操作系统中，利用泛设备驱动程序概念，很好地解决了上述问题。只要把这些“老”程序和开源程序统称为外来程序，象对待硬件模块一样，做个 driver 把这些模块封装起来，把这些外来程序当作设备来访问，其他模块就可以用标准的符合规范的方法访问这些“老”代码了。当然，软件模块之间使用 driver，效率会有所降低，但当今电子技术的发展，一个产品中计算部分所占的成本已经越来越低了，提高运算系统的速度只需增加很少的成本，甚至有许多嵌入式系统根本就没有充分利用 cpu 的计算能力。

呵呵，新问题又出来了，计算行列式的功能要用 driver 封装起来，那么三角函数要不要封装？浮点加减乘除运算要不要封装？在程序设计时，如何判断哪些功能要用 driver 封装是一个棘手的问题，毕竟，用 driver 封装比直接用函数调用需要多得多的代码和 cpu 执行时间的开销，滥用 driver 可能导致程序的极端低效。一般来说，以下几个原则可以帮助我们确定哪些功能需要用 driver 封装。适合用 driver 封装的情况有：

1. 模块的运算密度很高，使用 driver 带来的额外开销于其运算密度相比微不足道时，建议用 driver 封装运算模块使其与其他模块隔离。这样有利于运算模块单独改进算法而不影响使用该运算模块的其他代码。
2. 某一个功能的专业性很强，必须由掌握专门技能的人来设计算法时，最好用 driver 把这部分代码与系统的其他部分隔离。
3. 决定产品差异性和个性化的代码，建议用 driver 隔离，这样，当发展系列化产品型号时，只需要修改这些系列化相关的模块就可以了。

4. 在可预见的将来，有可能会使用硬件加速模块实现的功能。比如有些视频处理产品，企业可能同时生产高低档次不同的产品，在抵挡产品中，由于其色彩和分辨率可能比较低，用 cpu 直接计算可以满足要求；而在高档产品中，需要添加一个视频加速硬件来处理大规模的图形处理运算。使用 driver 封装显示模块，可以使产品中除显示模块以外的其他代码保持一致，使不同档次的产品最大限度地共享代码。
5. 在多 cpu 的情况下，可能在 cpu 之间转移的模块，需要用 driver 封装。例如一个数据采集系统，dsp 进行数据采集和计算，然后用串口传送到 mcu 上进行数据库存储与其他操作，如果计算结果的数据量大于原始数据而导致通信不堪重负，其中的计算模块就有可能被转移到 mcu 上执行。用 driver 封装计算模块后，该模块在 cpu 之间转移就很方便。

不适合用 driver 封装的情况有：

1. 人们已经习惯直接用函数实现的功能，比如三角函数等，尽量不要使用 driver 封装。改变程序员的编程习惯往往是产生软件 bug 的温床。
2. 对执行效率非常敏感的功能，用 driver 封装后可能达不到效率需求，就不要使用 driver 封装。
3. 功能非常简单，不需要复杂操作就可以完成的功能，例如点亮一个直接用 CPU 的 IO 口驱动的发光二极管，虽然是硬件驱动，但最好直接用函数实现。
4. 不具有完整而独立功能的代码，例如一个 16 进制数转 BCD 码的功能，应该用函数实现，不要用设备 driver 封装。

9.2 泛设备 driver 分类

9.2.1 设备分类原则

有许多操作系统的设备驱动架构对设备分类，分类的方法主要依据设备本身的技术特征，比如分成字符设备、块设备、网络设备等。djyos 系统中，设备并不以技术特征分类，也不以用途分类，泛设备驱动程序的代码与该设备的类型无关，也就是说，你不能通过阅读一个泛设备驱动程序代码识别该设备的类型。不仅如此，djyos 泛设备管理还不区分一个设备是硬件设备还是软件设备，只要设备能实现模块接口规定的功能就可以了。也就是说，在设计的前端，你根本看不出某设备是用软件实现的还是硬件实现的，一直到设计的后端才能分辨某设备是软件还是硬件。djyos 程序员可以专注于本模块的实现，在他看来，项目中除自己编写的模块以外的一切功能模块都是设备，或者是资源，这就是泛设备的魅力！

djyos的泛设备driver的分类以方便软件工程管理、方便系统架构师做系统架构设计为目标，按照友好程序分组的原则进行划分。我们在 3.2 节中讲到友好组管理，也讲到区分友好代码与可疑代码的基本方法，我们在这里讨论一下泛设备driver的友好组划分方法。

现代操作系统一般都包含网络功能，这就需要一个网络驱动程序，如果操作系统包含文件系统，就需要文件存储介质的硬件驱动程序，这些驱动程序都是操作系统的一部分，从这一方面讲，driver 应该毫无疑问地划为操作系统的友好代码。嵌入式产品千变万化，很多用户都有自己特有的外围设备，这些设备需要专用的驱动程序，这样，对于嵌入式系统应用程序的开发者来说，编写驱动程序就是非常普遍的行为了，而这些代码是由应用程序团队开发的，应与该团队开发的其他应用程序代码分在同一个友好组中；如果某设备为特定的应用程序服务，无论该设备驱动程序和应用程序是否由同一个团队开发的，应该有相同的友好属性，所以，这个泛设备 driver 应该是相应的应用程序的友好代码。如果某设备 dirver 是由第三方

（往往是硬件设备提供方）开发，并且可能服务于多个应用程序模块，则该 driver 可能独立成为一个友好程序组，或者多个这样的 driver 共同组成一个友好程序组。把 driver 统一划分为操作系统的友好代码，或者是某特定应用程序的友好代码，都是不合理的，它不符合软件分工原则，在软件的开发过程中和执行过程中都可能付出代价。linux 就是典型的把所有 driver 都划归操作系统友好程序组的例子（linux 并没有友好组的概念，权且这样说吧），linux 把所有驱动程序都划分为内核的一部分，与内核一起运行。linux 为此还提供了专门的工具帮助应用程序开发者把自己开发的 driver 加入到内核中，引狼入室还给狼绘制详细的地图，极大地方便了不怀好意者或者初学者破坏内核。笔者在开发一个 linux 下的 pci 设备驱动程序时非常深刻的体会，在开发的初期，程序中有许多 bug，由于设备 driver 与内核一起运行，这些 bug 也就直接污染了操作系统，动不动就造成死机，导致调试过程中频繁重启计算机，每次启动都要等待好几分钟，气得我差点把计算机给砸了。开发调试结束以后，并不能说明程序中就不存在错误了，由于操作系统和 driver 绑在一起，在同一个虚拟机空间运行，操作系统中的 bug 和 driver 中的 bug 是会互相污染的，将来执行过程中如果出现错误，没有人能说清楚是因为操作系统的 bug 导致的，还是 driver 的 bug 导致的错误，这势必给彻查问题带来不必要的麻烦。

因此，设备 driver 既不能单纯划为操作系统的友好代码，也不能单纯划为某个应用程序友好代码，而是要根据 driver 的开发团队区别对待，这是 djyos 系统设备 driver 分类的理论基础。djyos 把设备分为 3 中类型，分别是内核设备、独立设备、私有设备。分类的标准与设备本身的特点无关，而是按设备的驱动程序是由谁开发和谁使用为标准划分，原则是同一个团队开发的所有程序（包括应用程序和 driver）应该有相同友好属性，由操作系统提供相同等级的保护，不同团队开发的软件不能互相污染。这种划分方式有利于项目组织和管理，有利于人员分工，有利于合理分配产品不同模块的可靠性指标。

9.2.2 内核泛设备 driver

由操作系统开发团队开发、经过严格测试的、操作系统执行所必须的泛设备 driver，或者是由第三方开发商开发，经过操作系统开发团队严格认证的泛设备 driver，称为内核泛设备 driver。内核泛设备 driver 与操作系统一起编译，在操作系统启动时和操作系统一起加载到操作系统空间，操作系统和用户程序都可以使用他们。网络泛驱动 dirver、文件系统泛设备 driver 等是典型的内核泛设备 driver，这些泛设备 driver 与操作系统调度程序共同组成操作系统内核。

9.2.3 独立泛设备 driver

由独立团队但不是操作系统开发团队开发的泛设备 driver 称为独立泛设备 driver；最常见的独立泛设备 driver 是由设备厂商提供，未经操作系统开发团队认证的泛设备 driver；或者由应用程序开发团队开发的，可能有多个应用程序需要使用这些泛设备 driver。在 djyos si 模式中，独立泛设备 driver 事实上和操作系统编译成一个单一的可执行文件；djyos dlsp 模式中，操作系统单独加载和初始化独立泛设备 driver；djyos mp 模式中，操作系统为独立泛设备 driver 单独建立了一个进程，所有独立泛设备 driver 都在这个进程中运行。

9.2.4 私有泛设备 driver

私有泛设备 driver 是指与应用程序一起开发和编译，由该应用程序专用的泛设备 driver，该 driver 可以与相应的应用程序一起加载到同一个进程运行，该应用程序则称作设备的宿主应用程序。私有泛设备 driver 与宿主应用程序有相同的友好属性，同属一个友好程序组。

9.3 谈谈硬件驱动

都江堰操作系统的泛设备驱动并不是以管理硬件为设计目标，但不可否认，许多driver是用来管理硬件的，所以还是要谈谈硬件驱动。许多操作系统理论认为，所有硬件功能组件都应该由操作系统进行管理，应用程序要使用硬件的话，就只有向操作系统提出申请，得到操作系统批准并且由操作系统代为执行，才能防止一些未授权的代码错误地操作硬件，保证系统安全。但都江堰操作系统却认为，产品中由CPU外的硬件和CPU内的软件共同组成的功能模块，其中软件部分放在什么地方，由谁负责设计，应该由产品系统架构决定。如果系统架构决定把某包含硬件设备的功能模块作为操作系统的一部分，就应该由操作系统部分实现；如果该功能模块被划到一个应用程序组件中，则driver应与相应的应用程序组件绑定在一起。还有人会认为，硬件设备与driver是一一对应的，有与CPU连接的硬件实体就应该为其建立设备，在系统设备链表中占一个位。这些刻板的观点是有害的，无论是由硬件实现的功能模块，还是由软件实现的功能模块，是否把它作为一个设备来管理，应该根据整体系统的实现策略来决定。如 图 9-3（b） 所示的硬件结构中，我们可以为IIC总线驱动建立一个设备，其上层设备比如电源管理设备通过设备接口来访问IIC总线设备，也可以只提供几个IIC总线的读写函数，由上层设备直接调用。反而，象电源管理芯片、温度测控芯片这些没有与CPU直接连接的实体，都一一建立了设备。

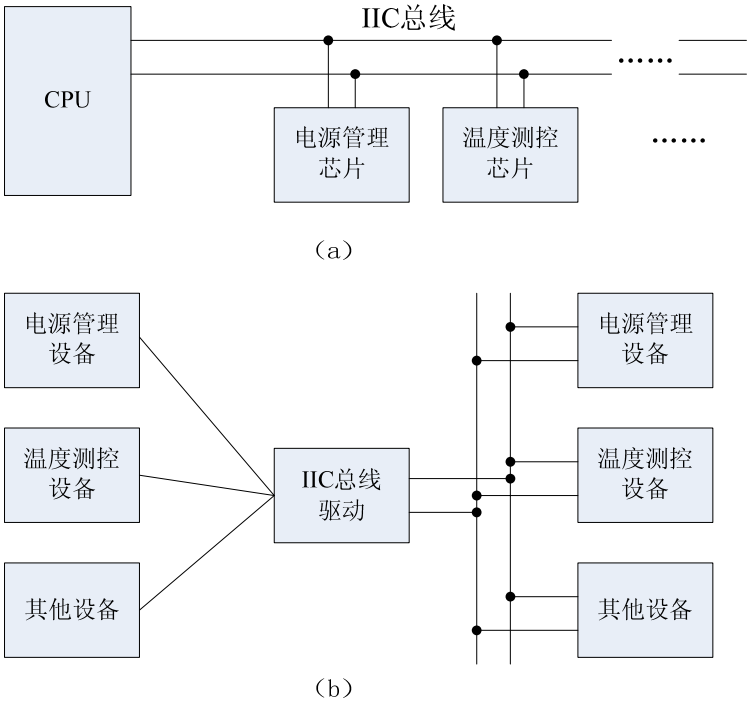


图 9-3 IIC 总线连接着多个实体

为什么要这样做呢？打个比方吧，在军队编制中，陆军师有司令部，下辖有直属炮兵营

和和普通的团，团里的各连队还有炊事班。假如在作战中，团长需要炮兵支持，则只有请求师司令部给炮兵营下令，或者获得师长授权只会炮兵营，至于如何设置炮兵阵地，往哪里打炮，什么时候打，打多少炮弹等，是由团长说了算。而团长要吃饭时，则没有这么啰唆，直接下命令本团的炊事兵做饭即可。师司令部相当于操作系统，团相当于一个用户进程，炮兵营和炊事班是专业兵种，相当于硬件设备。团长使用专业兵种的方法取决于师的兵力、火力编成。设计部队的兵力和火力编成是非常有学问的，一个师需要配多少炮兵、多少侦察兵、多少工兵，这些专业兵种配属为师直属部队，还是团属，或者营属，需要根据武器装备的发展和敌情的变化来决定。类比到设计嵌入式产品的系统架构，产品中需要配备多少硬件，硬件设备的驱动程序放在哪里，应用程序如何使用硬件设备，也应该由系统架构设计决定，而不是由操作系统设计师规定。操作系统规定只有通过它才能访问硬件，这完全是越俎代庖的行为，相当于不顾敌情硬性把专业兵种全部划归师直属。团长需要通过师长批准才能调动炮兵营，与直接指挥炊事班相比，有什么不同呢？不同点是，首先，如果团长要造反，因为有师长把关，他肯定不敢请求师长命令炮兵营炮击师司令部，这样做是自寻死路，而炊事班则不同，他们照样会给叛军做饭；其次，如果有别的团也要使用炮兵营，得由师长拍板谁先用，谁后用。相同点是，他们的战斗效果如何，全在乎团长指挥得好不好。软件也一样，由操作系统管理的硬件，当有应用程序争用时，由操作系统排队使用；操作系统只能保证应用程序使用硬件设备的方法是正确的，但不能保证应用程序用这些硬件做正确的动作。比如一个由操作系统管理的串口，它能保证不被设置为非法的 baud，但不能防止应用程序发送非法字符。一句话，它能确保你不用刀往石头上砍，这样会卷刃，但你用刀来行侠仗义还是杀人越货，一概不管。

9.4 实现泛设备驱动程序

9.4.1 辅助组件

djyos 的泛设备 driver 是用于不同的功能模块之间交换数据的，操作系统提供了一些常用的辅助组件，帮助用户编写设备 driver。这些组件是用户编写 driver 的有力武器，但他们的作用并不局限于编写 driver，在模块内部的功能块之间，使用这些组件也十分有用，善用这些组件，对规范功能块之间的数据交换，理顺功能块之间的数据流，降解功能块之间的耦合，使程序的逻辑关系更加清晰，都很有帮助。在 djyos 系统内部，键盘驱动、异步串口驱动使用了环形缓冲区，而文件系统则使用环形缓冲区作为读文件缓冲区，使用了线性缓冲区作为写文件缓冲区。

9.4.1.1环形缓冲区

环形缓冲区是类似fifo的缓冲区，用于模块间数据通信时提供一个缓冲存储区域，图 9-4显示了环形缓冲区的典型结构。

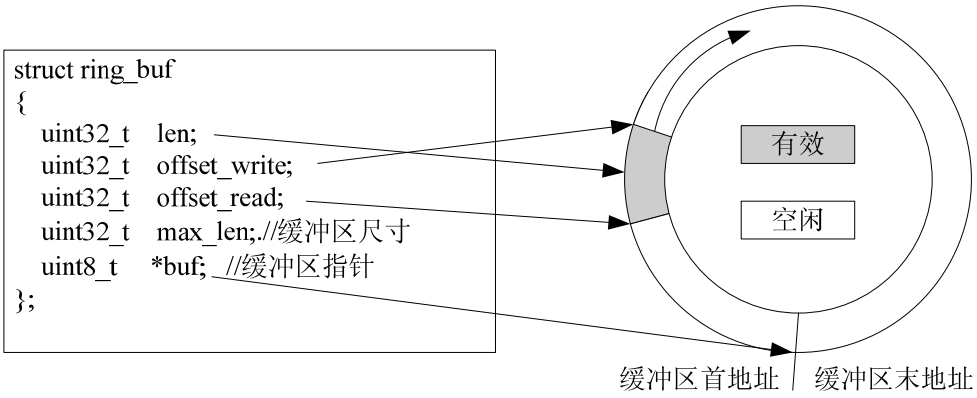


图 9-4 环形缓冲区结构

图 9-4 的说明如下：

1. 缓冲区数据结构定义了一个字节池指针，并没有为其分配内存，缓冲区初始化以后，这个指针指向实际的字节池首地址。
2. offset_write 和 offset_read 记录的是下一个读（写）位置，是从缓冲区首地址开始的偏移量。
3. len 记录了缓冲区中的数据量，从 offset_write 和 offset_read 也可以获得 len 参数，但是这样的话，当 offset_write 和 offset_read 相等时，就无法判断是缓冲区满还是缓冲区空。使用 len 参数也可以加速缓冲区操作。

有一个值得讨论的问题，就是 图 9-4 中的缓冲区有谁分配的问题，一种观点是，环形缓冲区管理模块应该负责分配内存，并且连“struct ring_buf”结构所需要的内存也应该有环形缓冲区管理模块分配，这样，环形缓冲区管理模块需提供ring_create函数和ring_free函数，用户使用环形缓冲区时将非常简单，建立一个含有 1000 个字节的环形缓冲区，代码如下：

```
struct ring_buf    *my_ring;                    //定义环形缓冲区指针
my_ring = ring_create(1000);                    //建立环形缓冲区
……;                                            //使用环形缓冲区
ring_free(my_ring);                             //释放环形缓冲区
```

另一种观点是，环形缓冲区管理模块不应该负责分配内存，应用程序需要使用的内存由应用程序负责分配，环形缓冲区管理模块不提供 ring_create 函数，而是提供 ring_init 函数，用户使用环形缓冲区比提供 ring_create 复杂些，建立一个含有 1000 个字节的环形缓冲区的代码如下：

```
struct ring_buf    my_ring;                     //定义环形缓冲区数据结构
my_buf = m_malloc(1000);                        //为环形缓冲区分配内存
ring_creat(&my_ring, my_buf, 1000);             //建立环形缓冲区
……;                                            //使用环形缓冲区
m_free(my_buf);                                 //释放环形缓冲区内存
```

其中，my_buf 可以全局数组静态分配，也可以调用 m_malloc（动态堆）函数或者 mb_malloc（固定块）函数分配，如果只在函数内部使用，还可以用局部数组在栈中分配。

以上两种方法区别，表面上是环形缓冲区实现方法的区别，但实质是内存管理策略的区别。回顾 2.9 节，在该节中，我们讨论了静态分配和动态分配内存两种内存分配策略的特点，而环形缓冲区初始化方法，则是两种内存分配策略在实际编程中如何选择的典型案例。

从计算机执行的角度看，第一种方法，内存由环形缓冲区管理模块分配，直到调用时，ring_create 函数才可能获知用户究竟需要多少内存，因此，只能选择任意长度或者块相联动态分配，即使用户事先知道所需缓冲区大小也无济于事，不可能使用静态分配。而第二种方

法则不同，用户可以根据需要灵活使用各种方法分配内存，尤其在编译前就能知道所需缓冲区大小的情况，这也是实际编程中最常见的情况，可以用定义数组的方式静态分配内存。动态分配内存是软件不确定性的重要来源，它的执行时间长且难于意料，也不一定可以成功得到所需的内存，最重要的是内存分配失败的噩耗必须在执行时才能暴露，这在有些嵌入式系统中是不允许的。显然，第二种方法有无可比拟的灵活性和实时性，在实时系统中，时间关键的模块、或者不容失败的模块，必须使用静态分配缓冲区，这只有第二种方法可以实现。

从代码实现角度看，显然第一种方法更为优越，首先，如果把环形缓冲区看做一个对象，那么，它所使用的字节池的地址，应该是这个对象的内部“秘密”，属于对象内部实现方法的一部分，从面向对象编程的角度出发，这个“秘密”应该是对使用对象的程序员“保密”的，第一种方法显然能够做到这点。而第二种方法呢，因为字节池是由用户提供的，先天缺陷导致用户可以对该缓冲区做任何操作，不能排除某些偷懒的程序员会贪图方便直接操作字节池，这就可能埋下 bug 的种子。其次，按第一种方法，用户代码将更加简洁，易读。

综上所述，第一种方法显然比较适合于通用操作系统，以及实时系统中的非实时部分（所有的实时系统，并不是所有模块都有实时性需求的）；而第二种方法呢，特别适合高可靠性的系统以及实时性要求很高的场合。既然两种方法各有千秋，那么，一个折中的办法，就是同时提供 ring_create 函数和 ring_init 函数，前者由环形缓冲区管理模块分配内存，后者由用户分配内存，djyos 系统采用两种方法并存的方案。

环形缓冲区操作函数：

1. 建立环形缓冲区

函数原型：

```
struct ring_buf    *ring_create(uint32_t len);
```

功能：建立环形缓冲区并初始化，缓冲区使用的字节池有该函数分配。如果没有足够的内存用于建立缓冲区，则任务进入等待状态。

参数：len，缓冲区长度.单位是字节数

返回：得到的环形缓冲区指针

2. 初始化环形缓冲区

函数原型：

```
void ring_init(st_ring_t *ring, uint8_t *buf, uint32_t len);
```

功能：初始化环形缓冲区，使用这个函数之前，用户应该定义缓冲区内存块和缓冲区数据结构。

参数：ring，目标环形缓冲区结构指针

buf，缓冲区起始地址

len，缓冲区长度.单位是字节数

返回：无

Example：

```
st_ring_t    my_ring;                    //定义环形缓冲区数据结构
uint8_t      my_buf[1000];               //定义缓冲区内存块
ring_creat(&my_ring, my_buf, 1000);  //建立环形缓冲区
```

3. 删除环形缓冲区

函数原型：

```
void ring_free(struct ring_buf *ring);
```

功能：删除用 ring_create 函数建立的环形缓冲区，释放其中的字节池。如果该缓冲区不是用 ring_create 函数建立的，则执行空操作。

参数：ring，被删除的缓冲区。

返回：无

4. 环形缓冲区写入

函数原型：

uint32_t ring_write(st_ring_t *ring,uint8_t *buffer,uint32_t len);

功能：往环形缓冲区写入若干个字节，返回实际写入的数据量，并移动写指针。如果环形缓冲区没有足够的空间,按实际剩余空间写入。

参数：ring，目标环形缓冲区结构指针。

buffer，待写入的数据指针。

len，待写入的数据长度。单位是字节数。

返回：实际写入的字节数，如果缓冲区有足够的空间，=len。

5. 环形缓冲区读

函数原型：

uint32_t ring_read(st_ring_t *ring,uint8_t *buffer,uint32_t len);

功能：从环形缓冲区读出若干个字节，返回实际读出的数据量，并且移动读指针，如果缓冲区内数据不足，按实际数量读取。

参数： ring，目标环形缓冲区结构指针。

buffer，接收数据的缓冲区指针。

len，待读出的数据长度。单位是字节数。

返回：实际读出的字节数，如果缓冲区有足够的数据，=len。

6. 检查环形缓冲区中数据量

函数原型：

uint32_t ring_check(st_ring_t *ring);

功能：检查指定的环形缓冲区中的数据量,返回字节数。

参数：ring,目标环形缓冲区指针。

返回：缓冲区中的数据量

7. 检查缓冲区是否满

函数原型：

bool_t ring_if_full(st_ring_t *ring);

功能：检查指定的环形缓冲区中是否已经满。

参数： ring，目标环形缓冲区指针。

返回：满则返回 true，非满返回 false。

8. 检查缓冲区是否空

函数原型：

bool_t ring_if_empty(st_ring_t *ring);

功能：这是 ring_is_full 的姊妹函数，虽然功能与 ring_check 函数似乎有些重复，但可以与 ring_is_full 函数保持对称完整。否则，用户程序在环形缓冲区查空时使用 ring_check，而在查满时使++- 吧用 ring_is_full，显得有些别扭。

9. 清空环形缓冲区

函数原型：

void ring_flush(st_ring_t *ring);

功能：清除环形缓冲区中所有数据，本函数只是把环形缓冲区的读写指针归零，并没有实际覆盖缓冲区中的数据。

参数：ring，目标环形缓冲区指针。

返回：无

10. 释放若干数据

函数原型：

uint32_t　　ring_dumb_read(st_ring_t *ring,uint32_t len)；

功能：读指针从当前位置开始，向前移动指定长度，释放掉被跳过的数据，相当于哑读了 len 个字节。如果缓冲区中的数据量不足，则全部释放。

参数：ring，目标环形缓冲区指针。

　　len，释放的数据数量。

返回：实际释放的数据量。

11. 查找字符

函数原型：

uint32_t ring_search_ch(st_ring_t *ring, char c)；

功能：从环形缓冲区的当前读位置开始，查找字符 c 的位置。

参数：ring，目标环形缓冲区指针。

　　c，需查找的字符。

返回：如果找到，返回 c 出现的位置（相对于当前读指针的增量），如果没有找到则返回 cn_limit_uint32。

12. 查找字符序列（串）

函数原型：

uint32_t ring_search_str(st_ring_t *ring, char *string,uint32_t str_len);

功能：从 ring 当前读位置开始查找字符序列的位置，字符序列可以包含字符 0，不是以 0 结束的字符串，而是指定序列长度。

参数：ring，目标环形缓冲区指针。

　　string，需查找的字符序列。

　　str_len，字符序列长度。

返回：如果找到，返回 str 出现的位置（相对于当前读指针的增量），如果没有找到则返回 cn_limit_uint32。

9.4.1.2线性缓冲区

线性缓冲区也是一种类似于 fifo 的缓冲区，它实质上是环形缓冲区的简化版。它与环形缓冲区的区别在于数据不发生环绕，它的读指针总是在缓冲区的起始地址。在实际应用中，经常会有这样的需求，用一个缓冲区收集数据，数据量积累到一定数量或者超时时间到后，再一次处理完所有数据。对于这种应用，使用环形缓冲区虽然也可以满足要求，但环形缓冲区有数据环绕的问题，环绕就是指针到达缓冲区末地址后又从缓冲区首地址的过程。读写环形缓冲区都需要判断数据是否发生环绕，这是很消耗时间的。而使用线性缓冲区则没有这个问题，速度比环形缓冲区快很多。

线性缓冲区每次读都要求完全拷贝缓冲区中所有数据，然后把读指针清零，不支持读任意长度数据，读和写不能异步进行。如果允许不完全读取，读函数把剩余的数据拷贝到缓冲区头部，是一种更安全更通用的做法，但是，这就违背了建立线性缓冲区的初衷。线性缓冲区就是作为一个快速缓冲区来设计的，它省略了一切会影响效率的繁文缛节，拷贝剩余数据的过程将使线性缓冲区的速度比缓存缓冲区还慢，如果需要完全功能的缓冲区，使用环形缓冲区更加理想。

与环形缓冲区一样，线性缓冲区也存在内存由谁分配的问题，其分析过程和处理方法与

环形缓冲区是一致的。

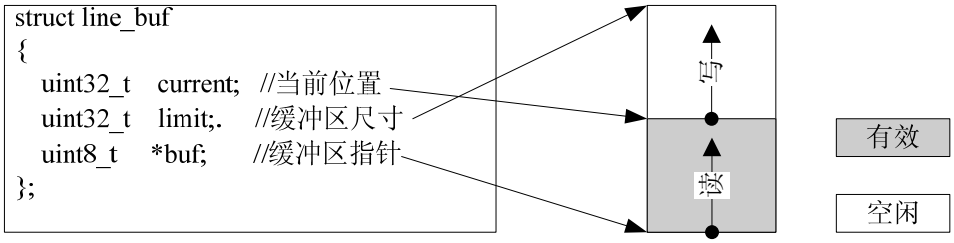


图 9-5 线性缓冲区结构

表格 9-1 环形缓冲区与线性缓冲区特性比较

项目	环形缓冲区	线性缓冲区
读写方式	先进先出	先进先出
读/写速度	慢	块
异步读写	支持	不支持
任意长度读	支持	必须一次读完所有数据
任意长度写	支持	支持

9.4.1.3栈缓冲区

栈是一种后进先出的缓冲区，在做数据结构编程时常用。尤其做递归迭代运算时，常用栈保存临时结果。djyos 系统提供一个简易的栈管理模块。

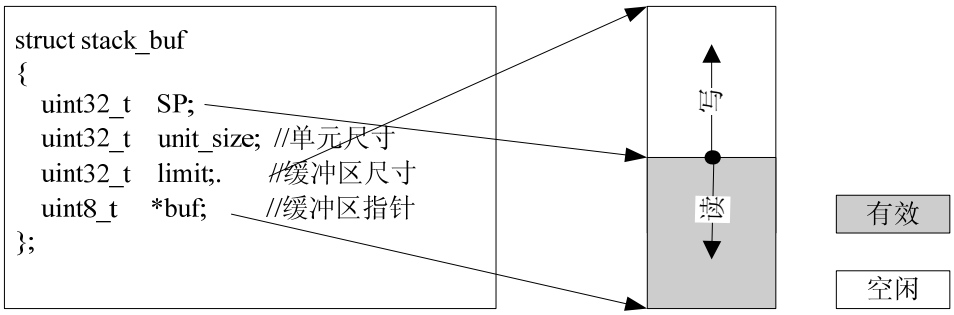


图 9-6 栈缓冲区结构

9.4.2 泛设备驱动程序数据结构

在djyos\kernel\include\dirver.h文件中，定义了实现泛设备driver管理所必需的数据结构，实现泛设备管理涉及到比较复杂的数据结构，图 9-7~图 9-9 层层展开了这些数据结构是如何组织的。

系统资源链表，管理系统中所有资源

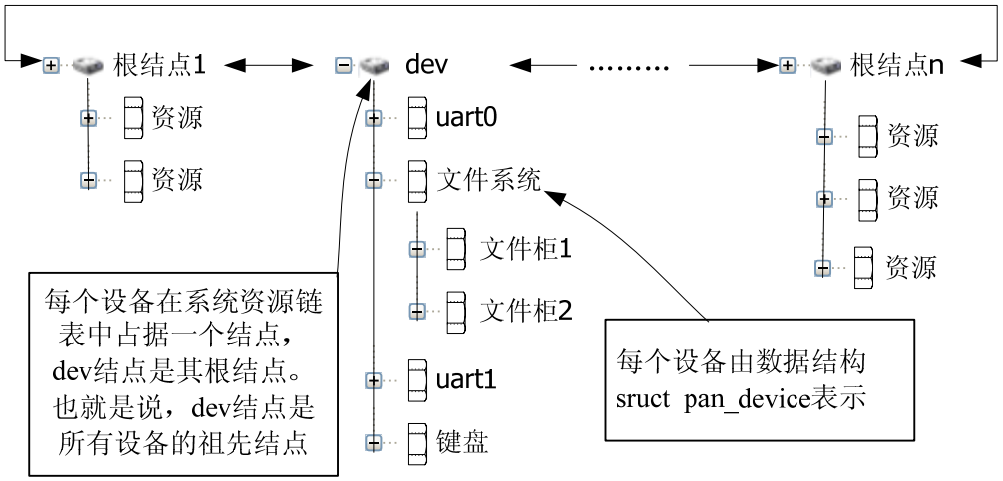


图 9-7 泛设备数据结构图之一（共三）：资源视图

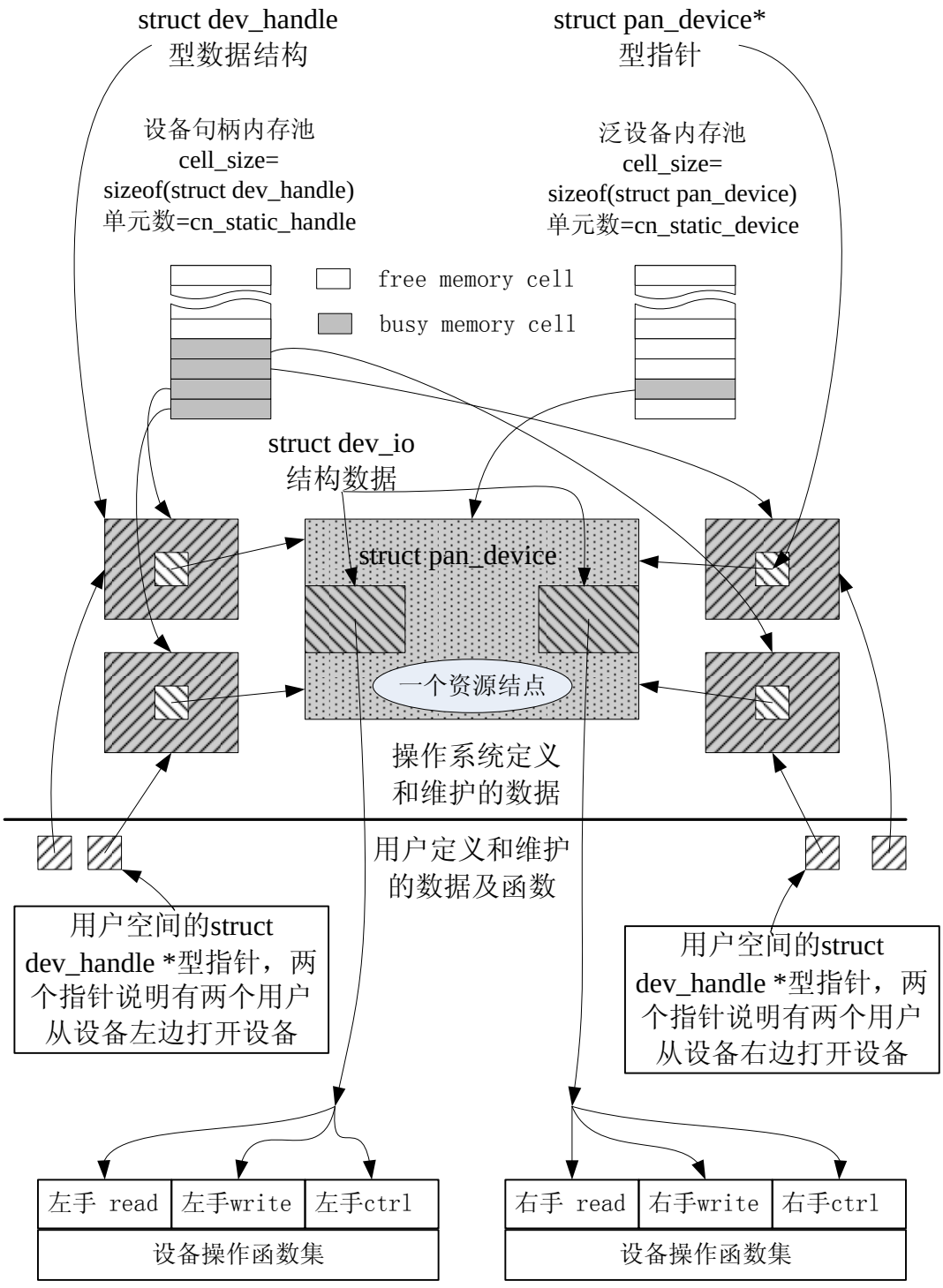


图 9-8 泛设备数据结构图之二（共三）：逻辑结构

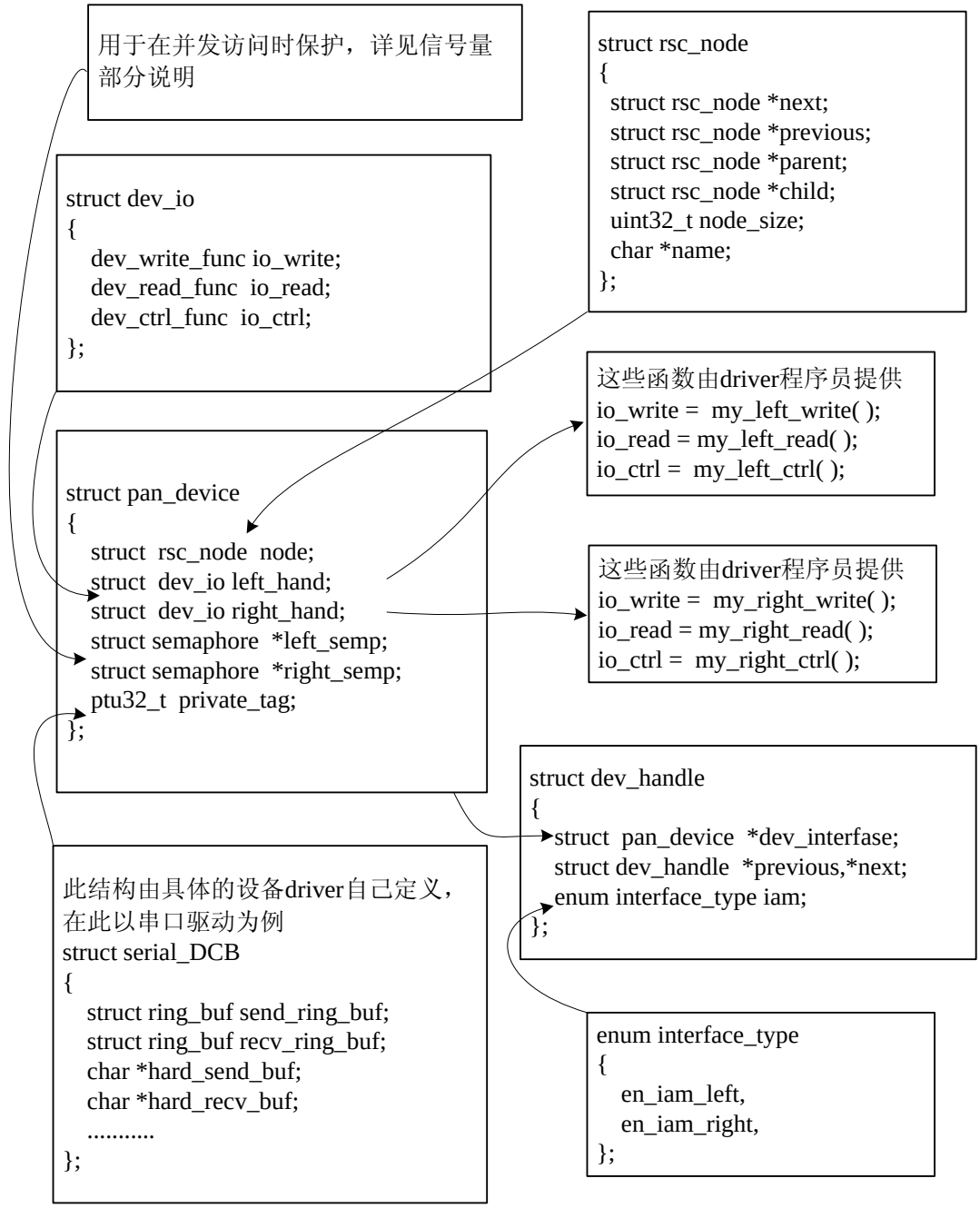


图 9-9 泛设备数据结构图之三（共三）：展开

图 9-7 显示了泛设备管理对系统资源树的依赖关系，每个设备都是系统中的一个资源，“dev”资源结点是所有设备的祖先结点，静态全局指针struct pan_device *pg_device_root指向该结点。细心的读者已经看到了，pg_device_root并不是struct rsc_node* 类型，是的，资源链表中的结点可以是纯struct rsc_node类型的数据，也可以由struct rsc_node类型的数据加上结点负载构成的新数据结构，方法是在负载数据结构中必须嵌入struct rsc_node类型的成员，且是第一个成员，比如，struct pan_device 结构的第一个成员就是struct rsc_node node。node是负载数据结构的第一个成员。关于资源链表的详细说明，参见“第 8 章”。

图 9-8 显示了各种数据结构的逻辑关系，可以看出，struct pan_device是泛设备管理的核心数据结构，每个设备占用一个，利用内嵌的struct rsc_node node成员把设备连接到系统资源链表中。当建立设备时，泛设备管理模块从pg_pan_device_pool内存池中分配一块内存，分配算法见“7.3固定块分配法

”。在 kernel.h 文件中定义的常量 cn_device_limit 规定了 pg_pan_device_pool 内存池中静态分配的内存块数量。

每当应用程序打开设备，泛设备管理模块将从 pg_device_handle_pool 内存池中分配一块内存，分配方法与 pg_pan_device_pool 内存池相同。应用程序得到的是 struct dev_handle*型指针。无论是从左手打开还是从右手打开，都得到相同类型的指针，有了这个指针，应用程序就可以实现对设备的读、写、控制操作。

图 9-9 描述了各数据结构展开后的相互关系，其中由用户实现的的函数有两组共 6 个，分别是左手接口的my_left_write、my_left_read、my_left_ctrl和右手接口的my_right_write、my_right_read、my_right_ctrl函数。操作系统提供 3 个系统调用分别对应这 6 个函数，他们的对应关系是

dev_write 对应 my_left_write 和 my_right_write。

dev_read 对应 my_left_read 和 my_right_read。

dev_ctrl 对应 my_left_ctrl 和 my_right_ctrl。

如果用户打开设备的左手接口，就被称为左手用户，当左手用户调用dev_write函数时，在泛设备管理模块的引导下，最终将调用my_left_write函数，若是右手用户则调用my_right_write函数，其他两个函数也一样。以左手用户为例，实现过程如下（参考 图 9-9 以理解下述过程）。

1. 用户调用 dev_open_left 函数打开设备的左手句柄，获得 struct dev_handle 类型的指针 my_left_handle。dev_open_right 打开的右手句柄的类型也是 struct dev_handle。
2. 调用 dev_write 函数，my_left_handle 作为该函数的第一个参数。
3. 泛设备管理模块从 my_left_handle 参数中得到该设备的 struct pan_device 类型的核心数据结构地址。
4. 再根据 my_left_handle-> iam 成员确定应该访问 struct pan_device 的 left_hand 成员。
5. left_hand 是 struct dev_io 类型的数据结构，该结构的 io_write 成员便是指向 my_left_write 函数的指针。
6. 通过 io_write 调用 my_left_write 函数。

struct pan_device中有两个需要特别注意的成员left_semp和right_semp，他们是信号量指针，为什么他们被设计成指针而不是实体呢？我们知道，信号量可以在某些临界资源提供访问保护，有许多设备，特别是硬件设备，同一时间只能为有限个用户提供服务，用信号量可以保护该设备免受超过限制的并发访问（信号量的详情参见第 5.7 节）。如果在pan_device中使用信号量实体而不是指针，就隐含了这样一个规则：每个泛设备对应一对信号量，信号量对与泛设备是一一对应的，它使多个设备使用同一个信号量保护成为不可能实现的事情。这就要求程序员在一个不能并发访问或者只支持有限并发访问的实体（比如一个IIC总线）上，只能建立一个泛设备。而我们又知道，在一个实体上究竟建立几个泛设备，应该是应用程序的实现策略，限制用户只能建立一个泛设备，也就是限制了用户选择软件策略，或者说代替用户决定策略——这个策略只允许用户建立一个设备，操作系统只应该提供用户实现其策略的机制，而不应该越俎代庖。在 图 9-3 的案例中，一条IIC总线连接这多个实体，显然，我们在为这些实体建立泛设备driver时，至少有两种策略是可供选择的：

第一：为这条 IIC 总线上的所有实体建立一个设备。

第二：为总线上的每个器件单独建立设备。建立如 图 9-3（b）所示的设备结构。

显然，第一种方案有着明显的缺陷，IIC 总线上增加和减少芯片，或者改变任意一个芯片的驱动方法，都会造成整个 driver 的改变，任何错误都可能危及到总线上的其他器件，同时还要求总线上的所有器件的 driver 必须由同一个人（或者小组）编写。而如果把信号量定义为实体而不是指针，则只能用这种策略。

第二种方案呢，因为所有器件挂在同一个 IIC 总线上，而 IIC 总线是不允许并发操作的，也就是说，挂在这条总线上的所有设备必须用同一个信号量进行保护。如果 pan_device 中的信号量使用实体，电源管理设备和温度测控设备都有自己的信号量，电源管理设备的信号量只能保证其本身的并发访问安全，而不能保证 IIC 总线的并发访问安全，反过来又使得电源管理设备本身的多线程安全得不到保证。如果使用指针，则可以在实现 IIC 总线驱动时建立一个信号量实体，所有挂在这条总线上的设备都把指针指向这个信号量即可，这就保证了挂在这条总线上的所有设备的多线程安全。

大家可能已经注意到了，虽然设备是按名字访问的，但是 struct pan_device 结构中，并没有出现 name 成员，这是因为 djyos 系统借助资源管理模块组织泛设备的数据结构，与其他使用资源管理模块的模块一样，资源名（struct rsc_node 的 name 成员）就是设备名。

9.4.2.1 struct dev_handle 结构

这是应用程序唯一可以接触到的数据结构，如果你的目的只是使用设备，根本无需知道这个结构的任何细节即可正常使用设备。如果你是泛设备 driver 的设计者，就应该深入了解这个结构了。当用户 open 设备时，无论是从左手侧打开，还是从右手侧打开，将获得一个 struct dev_handle * 类型的设备句柄指针，用户使用 dev_read、dev_write、dev_ctrl 三个系统调用读写和控制设备时，需要提供这个指针，操作系统在这个指针的指引下，找到用户需要访问的具体设备，这个结构还记录着用户要使用左手接口还是右手接口，以及用于垃圾回收的指针。

这个数据结构完全由操作系统管理，用户获得指针后，切勿修改指针，也不要修改数据结构内各成员的值，否则将有不可预料的后果。

```
enum interface_type
{
    en_iam_left,
    en_iam_right,
};
struct dev_handle
{
    struct    pan_device *dev_interfase;
    struct dev_handle *previous,*next;
    enum interface_type    iam;
};
```

1. dev_interfase，这是一个指向实体设备的设备控制块的指针，这个设备就是设备句柄所代表的设备。
2. previous，next 两个指针可以用于构成一个双向循环链表，每个事件控制块有一个 held_device 指针，所有该事件线程打开且未关闭的设备 handle 组成一个双向循环链表，held_device 指针指向这个链表。当事件处理完（线程调用 y_event_done 函数或者线程函数返回）后，操作系统将检查该线程是否有未关闭的设备，如果有，则强行关闭之，以降低资源泄漏的风险。
3. iam，这个句柄的类型是左手类型还是右手类型。

9.4.2.2 struct pan_device 结构

这个结构是应用程序无需也不能直接接触的，只有泛设备 driver 的设计者才需要关心这个结构的细节。 “pan-”在英语中是表示“泛”的前缀，struct pan_device 是实现泛设备管理的核心数据类型，它包括操作系统管理设备所需的数据成员，也就是所有设备共同拥有的成员。而操作系统并不知道设备要实现的功能，这是设备的秘密，是设备设计者所要关注的，private_tag 成员指向具体设备独有的数据。

```
struct pan_device
{
    struct   rsc_node   node;
    struct   dev_io left_hand;
    struct   dev_io right_hand;
    struct semaphore *left_semp;   //左手信号量
    struct semaphore *right_semp; //右手信号量
    ptu32_t   private_tag;              //具体设备专用数据标签
};
```

1. st_rsc_node_t node，资源结点，用于把设备作为一个结点接入系统资源链表，参见 8.3 节。
2. left_hand,right_hand，是设备的应用程序接口，包括左右手接口的读、写和控制函数以及同步队列。
3. left_semp和right_semp是信号量指针，用于提供设备多线程并发访问保护的。详情参见第 5.7 节。
4. private_tag 是具体设备的专用标签，它的数据类型 ptu32_t 是一个特别的数据类型，它定义成一个至少 32 位（4 字节）长的、并且长度大于或者等于指针类型长度的整数数据类型，它有 3 层含义：
   a) 如果用户把它当整数使用，那么它至少可以得到一个 32 位字长的整数。
   b) 用户可以安全地把它强制转化为指针使用；
   c) 如果 djyos 被移植到一个指针长度小于 32 位的计算机中，那么 ptu32_t 就被定义成 uint32_t 型，如果指针的长度是 64 位的，那么 ptu32_t 就定义成 uint64_t 型数据。

   用户为自己的设备编写 driver 时，可以灵活利用 private_tag 成员，根据具体设备的复杂程度，它既可以是一个指针，指向由具体设备定义的复杂的数据结构，又可以根本不用。例如在文件系统中，文件柜设备的 private_tag 成员就是指向 struct st_DBX_device_tag 类型数据的指针；而在文件系统设备（设备名是“fs”）中，则直接把 private_tag 初始化为 0 了事。

9.4.2.3 struct dev_io 结构

与 struct pan_device 一样，应用程序程序员也无需关心这个结构，它是设备控制块中 left_hand 和 right_hand 成员的原型，用户通过这个两个成员可以分别实现设备左手接口读、写、控制和右手接口读、写、控制操作。数据结构定义如下：

```
struct dev_io
{
```

```
    dev_write_func    io_write;
    dev_read_func    io_read;
    dev_ctrl_func     io_ctrl;
};
```

1. io_read、io_write 函数指针，这两个指针指向该设备 driver 提供的设备读和写函数。函数指针类型 dev_write_func 和 dev_write_func 的定义如下：

```
typedef ptu32_t (*dev_write_func) (struct dev_handle * dev_hdl, ptu32_t src_buf,
                                   ptu32_t des_buf, ptu32_t len);
typedef ptu32_t (*dev_write_func) (s struct dev_handle * dev_hdl, ptu32_t src_buf,
                                   ptu32_t des_buf, ptu32_t len);
```

   2 个函数各有 4 个参数，dev_hdl 是设备句柄，操作系统不限制其他 3 个参数的用途。如何使用这几个参数由泛设备 driver 的作者确定，并通过说明书明确，使用者严格按照说明书进行操作设备。len 的类型是 ptu32_t，推荐把它做读写的数据长度；src_buf 和 des_buf 的类型是 ptu32_t，可用强制转换成指针使用。

2. io_ctrl 函数指针，用户用这个函数控制设备，参数 dev_hdl 是设备句柄，cmd 是操作功能号，另外两个参数的含义由设备 driver 的作者确定。dev_ctrl_t 类型定义如下：

```
typedef ptu32_t (*dev_ctrl_func) (st_dev_handle_t * dev_hdl,uint32_t cmd,
                                  ptu32_t data1, ptu32_t data2);
```

9.4.2.4内存池

创建一个设备，需要分配一个struct pan_device类型的数据结构，删除设备时则需要把这块内存释放；任何设备每打开一次，都需要分配一个struct dev_handle型数据结构，关闭时也需要释放内存。泛设备驱动的内存管理使用固定块内存分配策略，详情参考 第 7 章。

在 djyos\config\kernel.h 文件中定义了两个常量，用户可以改变这两个常量。

```
#define cn_device_limit       100   //定义设备数量，
#define cn_handle_limit       100   //定义可同时打开的设备人次
```

在 djyos\kernel\load2\driver.c 文件中，定义了几个全局变量和内存池。

```
static struct pan_device tg_mem_of_device[cn_device_limit];    //泛设备控制块内存池
static struct dev_handle tg_mem_of_handle[cn_handle_limit];   //泛设备句柄内存池
static struct mem_cell_pool *pg_pan_device_pool;         //设备控制块内存池头指针
static struct mem_cell_pool *pg_device_handle_pool;   //设备句柄内存池头指针
```

cn_device_limit 常量表示允许创建的设备数量，tg_mem_of_device [cn_device_limit]是为 cn_device_limit 个设备控制块保留的内存，当用户建立新设备时，就从内存池中取出一块内存用作新设备控制块，用户删除一个设备则把该块内存送回内存池。

cn_device_limit 常量表示默认最大可以同时打开的设备次数，设备被多次打开则重复计数，打开设备的左手和右手接口也要单独计数。tg_mem_of_handle [cn_handle_limit]是为 cn_handle_limit 个设备句柄保留的内存池，当用户打开设备，就从内存池中取出一块内存用于引用设备，并把内存地址返回给用户；当用户关闭设备，就把该设备句柄的内存块送回到内存池中。

9.4.3 初始化泛设备管理模块

在建立任何泛设备driver之前，先要初始化操作系统的设备管理器。与其他依托资源链表的模块一样，初始化泛设备管理模块的过程非常简单。代码 9-1 所示的dev_init函数是用来初始化设备管理器的，该函数是在操作系统加载过程的“4.7.1.8 节 内核组件初始化”步骤调用的。初始化工作包含 3 个步骤：

1. 在操作系统资源链表中增加了一个设备根结点，然后使根设备指针pg_device_root指向根结点，参见 图 9-7 可以更好地理解。
2. 建立设备控制块内存池和设备句柄内存池，这是两个“固定块分配法”的内存池，参见第 7.3 节。设备控制块池的块尺寸是sizeof(struct pan_device)，默认容量是cn_device_limit块；设备句柄池与此类似。
3. 创建一个互斥量，用于保护资源树中设备子树的并发访问，请读者注意区分该互斥量与保护设备本身的信号量。

初始化完成后，就可以建立泛设备了。泛设备管理模块并不是操作系统运行所必须的，如果项目中没有使用泛设备驱动，则可以在配置文件中禁止泛设备管理模块。

代码 9-1 初始化操作系统的泛设备驱动模块

```
bool_t module_driver_init(void)
{
    static struct    pan_device root;

    pg_device_root = (struct    pan_device *)
            rsc_add_root(&root.node,sizeof(struct    pan_device),"dev");
    //初始化泛设备控制块内存池
    pg_pan_device_pool = mb_create((void*)tg_mem_of_device,
                                   cn_device_limit,
                                   sizeof(struct    pan_device),
                                   "泛设备控制块池");
    //初始化泛设备句柄内存池.
    pg_device_handle_pool = mb_create((void*)tg_mem_of_handle,
                                   cn_handle_limit,
                                   sizeof(struct    dev_handle),
                                   "泛设备句柄池");
    __mutex_createe_knl(&tg_dev_mutex,true,"device driver");
    return true;
}
```

9.4.4 建立和使用设备

如 图 9-10 所示，使用djyos泛设备驱动程序非常简单，只需要少数几个步骤就可以完成，详细使用过程请参考api介绍章节。

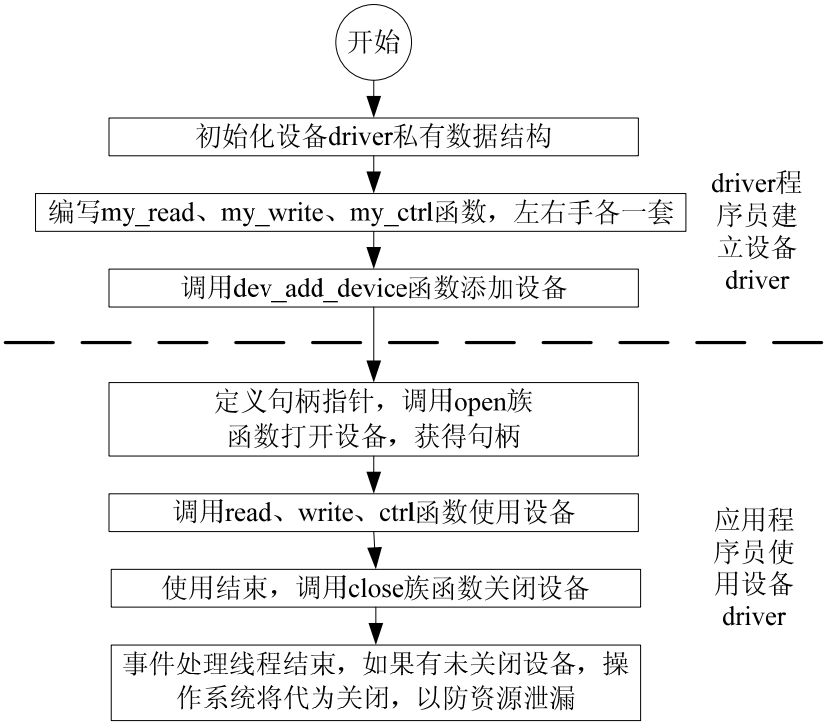


图 9-10 建立和使用设备流程

9.4.4.1创建新设备

创建设备的过程非常简单，dev_add_device函数用于创建设备，创建设备就是在 图 9-7 所示的设备资源树中添加一个节点，代码 9-2 是创建新设备的代码，图 9-11 则以uart1 设备为例图示了创建设备的过程。

代码 9-2 创建新设备

```
struct    pan_device *dev_add_device(struct    pan_device *parent_device,
                                     char                    *name,
                                     struct semaphore_LCB    *right_semp,
                                     struct semaphore_LCB    *left_semp,
                                     dev_write_func          right_write ,
                                     dev_read_func           right_read,
                                     dev_ctrl_func           right_ctrl ,
                                     dev_write_func          left_write ,
                                     dev_read_func           left_read ,
                                     dev_ctrl_func           left_ctrl )
{
    struct    pan_device    *new_device,*result;
    if((parent_device == NULL) || (name == NULL))
        return NULL;              //设备不能没有名字
    if(strchr(name,'\\'))     //名字中不能包含字符 \.
        return NULL;
    mutex_pend(&tg_dev_mutex,cn_timeout_forever);                    //1
```

```
    if(rsc_search_son(&parent_device->node,name) != NULL)
    {
        y_error_login(en_drv_homonymy,"设备重名");
        result = NULL;
    }else
    {
        if(parent_device != NULL)
        {
            //分配泛设备控制块给新设备
            new_device = mb_malloc(pg_pan_device_pool,0);          //2
            if(new_device != NULL)
            {
                //新设备添加到父设备下成为满子结点
                rsc_add_son(&parent_device->node,&new_device->node,
                                                    sizeof(struct    pan_device),name);      //3
                new_device->left_semp = left_semp;
                new_device->right_semp = right_semp;
                if(right_write != NULL)
                    new_device->right_hand.io_write = right_write;
                else
                    new_device->right_hand.io_write = NULL_func;
                if(right_read != NULL)
                    new_device->right_hand.io_read = right_read;
                else
                    new_device->right_hand.io_read = NULL_func;
                if(right_ctrl != NULL)
                    new_device->right_hand.io_ctrl = right_ctrl;
                else
                    new_device->right_hand.io_ctrl = NULL_func;
                if(left_write != NULL)
                    new_device->left_hand.io_write = left_write;
                else
                    new_device->left_hand.io_write = NULL_func;
                if(left_read != NULL)
                    new_device->left_hand.io_read = left_read;
                else
                    new_device->left_hand.io_read = NULL_func;
                if(left_ctrl != NULL)
                    new_device->left_hand.io_ctrl = left_ctrl;
                else
                    new_device->left_hand.io_ctrl = NULL_func;
                result = new_device;
            }
            {
```

```
                    y_error_login(en_mem_tried,"内存不足");
                    result = NULL;
                }
            }else
                result = NULL;
    }
    mutex_post(&tg_dev_mutex);
    return result;
}
```

1. 创建设备需要操作资源链表，资源链表是一个全局数据结构，必须用锁保护，这里用了互斥量做锁，支持优先级继承。
2. 参见 图 9-11 中 2 号方向箭头指示，添加设备之前必须先从设备控制块内存池中分配一块内存。
3. 参见 图 9-11 中 3 号方向箭头指示，把上一步分配的内存块连到资源链表相应位置，该位置由parent_device参数确定。连接是利用的是struct pan_device类型的node成员进行的（参见 8.3 节）。
4. 资源节点建立后，需要初始化泛设备数据结构的各成员，详见代码注释。

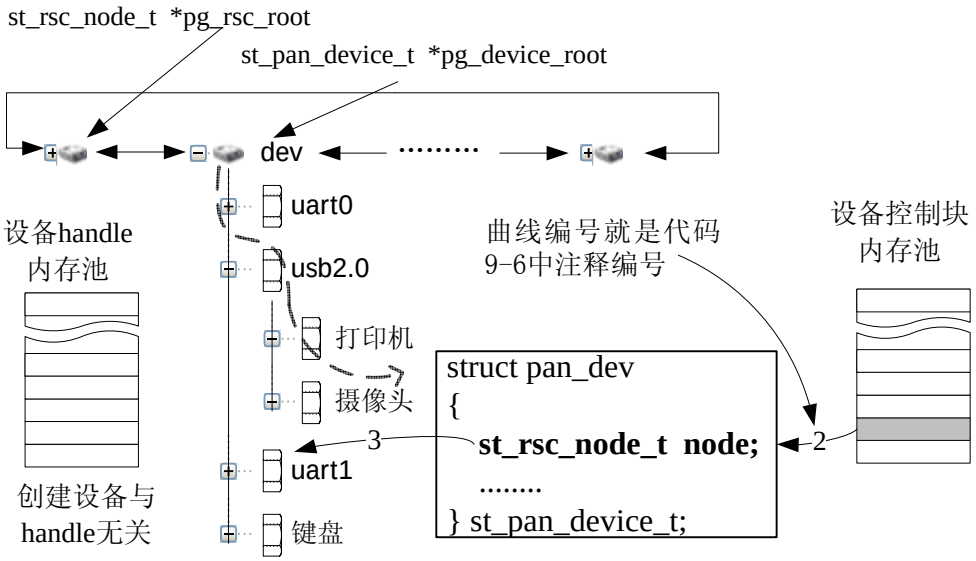


图 9-11 创建设备，以 uart1 为例

9.4.4.2打开设备

创建设备后，在使用设备前，还应该打开设备，与其他操作系统一样，打开设备时将获得一个设备 handle 指针。djyos 提供了 12 个 api 函数用于打开设备，分别是

1. 标准方式打开设备，dev_open_left，dev_open_right，这两个函数原型是：

struct dev_handle *dev_open_left(char *name,uint32_t timeout);

struct dev_handle *dev_open_right(char *name,uint32_t timeout);

本函数返回设备 handle 指针，该指针只能用于从设备的左手接口执行读写和控制。name 是被打开设备的完整路径名+设备名。timeout 是超时嘀嗒数，表示如果设备正忙，dev_open_left 函数最多被阻塞 timeout 个时钟嘀嗒。

2. 快速打开设备，dev_open_left_again、dev_open_right_again，曾经打开的设备关闭后再次打开时，使用本函数可以加速执行，详见 9.4.4.4 节。函数原型如下：

bool_t dev_open_left_again(struct dev_handle *handle,uint32_t timeout);

bool_t dev_open_right_again(struct dev_handle *handle,uint32_t timeout);

3. 在知道设备的任意一个祖先设备的情况下，可以调用 dev_open_left_scion、dev_open_right_scion 这 2 个函数，dev_open_left_scion 函数的原型是：

struct dev_handle *dev_open_left_scion(struct dev_handle *ancestor,
char *scion_name, uint32_t timeout);

struct dev_handle *dev_open_right_scion(struct dev_handle *ancestor,
char *scion_name, uint32_t timeout);

与大多数操作系统一样，djyos系统的设备是按名字访问的，djyos的设备驱动模块是在资源管理模块的基础上实现的，设备名就是资源链表中的资源名。访问设备要使用包含完整路径的设备名，打开 图 9-11 所示的“摄像头”设备的左手接口代码如下：

struct dev_handle *pg_handle_camera；

pg_handle_camera = dev_open_left(“usb2.0\\摄像头”,10);

表示用标准方式打开“摄像头”设备的左手接口，10 表示如果设备忙，最多被阻塞 10 毫秒（注意不是时钟嘀嗒）。

dev_close_left（pg_handle_camera）；函数用于关闭设备，当设备被打开然后关闭后，再次打开可用快速函数：

dev_open_left_again(pg_handle_camera,10);

打开设备应该包含设备的全路径名，但不包含根结点“dev”，不能只用设备名，下列代码是错误的

pg_handle_camera = dev_open_left(“摄像头”);

设备命名的规则与资源命名规则是相同的，详见 8.1 节。

我们以dev_open_left和dev_open_left_again函数为例，说明操作系统是如何实现打开设备的，图 9-12 演示了这两个函数的实现流程，代码 9-3 是这两个函数的实现代码。

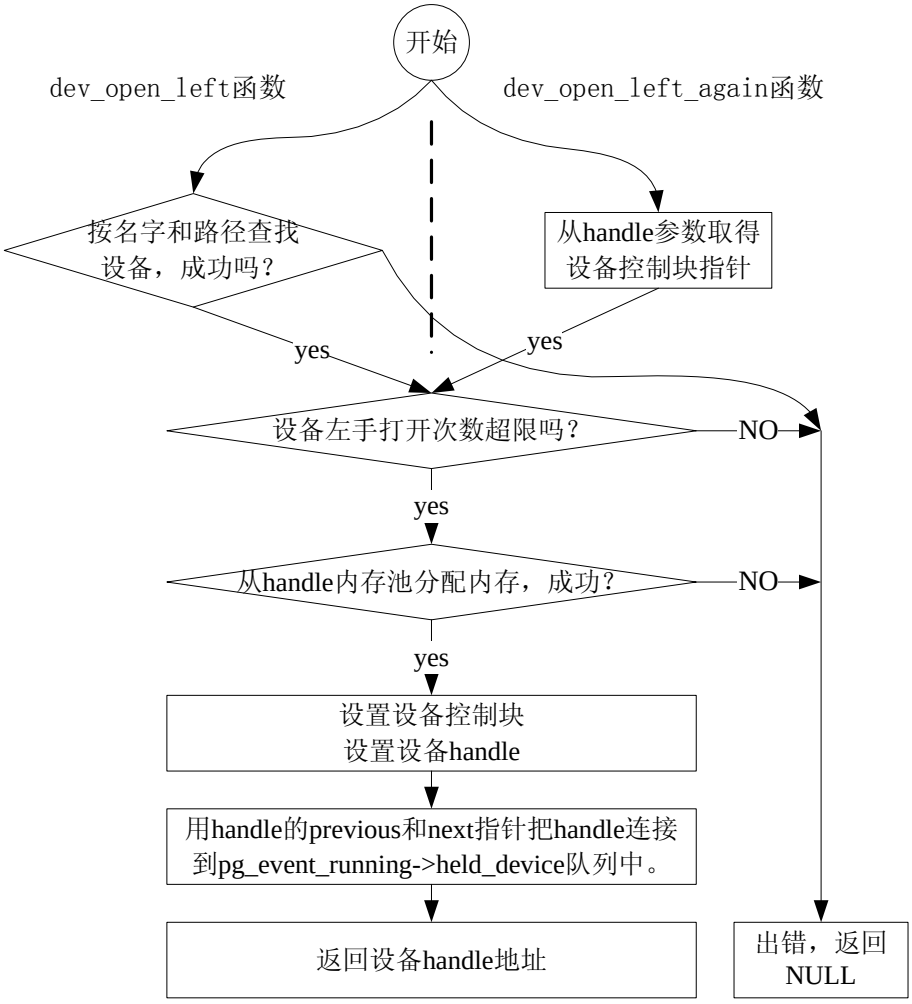


图 9-12 打开设备左手接口流程

1. 这两个函数非常相似，不同的仅仅是得到设备控制块的方法不同，dev_open_left 函数用字符串比较的方式在系统资源链表中查找名字匹配的资源结点，继而定位设备控制块，而 dev_open_left_again 函数则要求调用者提供被打开的设备控制块指针。用 dev_open_left 函数打开“摄像头”设备至少需要比较字符串“uart0”、“打印机”、“摄像头”3 个字符串，字符串比较是非常消耗时间的操作，如果设备树很高很大，需要比较的字符串更多。系统中可能有许多设备是多用户共享的，但同一时间只允许一个用户打开，为了实现更好地共享，要求用户使用设备后及时关闭设备。因此，软件中执行过程中通常会存在反复打开和关闭设备的操作，设备被关闭后，如要重新打开，使用快速函数 dev_open_left_again 可以跳过上述字符串比较过程，可以极大地提高软件的执行效率。
2. 快速函数的实现与关闭设备函数dev_close_left有密切关系，关闭设备时没有使handle指针指向NULL，而是指向被关闭的设备控制块（详见 9.4.4.3 节）。dev_open_left_again函数的参数类型是struct dev_handle*，但其包含的内容是目标设备的设备结点指针，类型是struct pan_device*。关闭设备后再次打开时dev_open_left_again函数把handle参数的类型强制转换成struct pan_device*进行访问。这是泛设备管理的实现细节，用户不必关心，这样可以使泛设备api接口更加一致，使用户不致混淆。
3. dev_open_left_again 函数里面，从 handle 取设备控制块指针时，无法做指针合

法性检查，用户应该保证在关闭和再次打开之间不修改 handle 指针，且设备没有被删除，否则将会有不可预料的后果。

代码 9-3 dev_open_left 函数和 dev_open_left_again 函数的代码。

```
struct  dev_handle *dev_open_left(char *name,uint32_t timeout)
{
    struct  pan_device     *pan;
    struct  dev_handle     *handle;
    struct  dev_handle     *result;
    if( ! mutex_pend(&tg_dev_mutex,timeout))    //这是保护设备树的互斥量
        return NULL;
    //在设备树中搜索设备
    pan = (struct  pan_device *)rsc_search(&pg_device_root->node,name);
    if(pan == NULL)     //如果没有找到 name 设备,返回空
    {
        result = NULL;
        goto end_of_dev_open_left;
    }
    if(sem_pend(pan->left_semp,timeout)==false)//获取信号量,这是保护设备的
    {
        result = NULL;
        goto end_of_dev_open_left;
    }
    //在分配内存之前判断信号量，可避免在没有信号的情况下长时间占用内存
    handle = mb_malloc(pg_device_handle_pool,0);  //从池中分配句柄内存块
    if(handle != NULL)
    {//成功分配句柄
        handle->dev_interfase = pan;
        handle->iam = en_iam_left;              //我是左手接口句柄
        if(pg_event_running->held_device == NULL)   //事件还没有打开过设备
        {
            handle->next = handle;      //句柄指针自成双向循环链表
            handle->previous = handle;
            pg_event_running->held_device = handle; //事件指针指向设备链表
        }else                                       //事件已经有打开的设备
        {
            //以下把新设备插入到事件打开的设备队列头部
            handle->next = pg_event_running->held_device;
            handle->previous = pg_event_running->held_device->previous;
            pg_event_running->held_device->previous->next = handle;
            pg_event_running->held_device->previous = handle;
        }
        result = handle;
    }else
        result = NULL;
```

```
end_of_dev_open_left:
    mutex_post(&tg_dev_mutex);
    return result;
}
bool_t dev_open_left_again(struct  dev_handle *handle,uint32_t timeout)
{
    struct  pan_device     *pan;
    bool_t result;
    if (handle == NULL) //句柄空
        return false;
    if( ! mutex_pend(&tg_dev_mutex,timeout))    //这是保护设备树的互斥量
        return NULL;

    pan = (struct  pan_device *)handle;    //句柄里存的实际是泛设备指针
    if(sem_pend(pan->left_semp,timeout)==false)//获取信号量
    {
        result = false;
        goto end_of_dev_open_left_again;
    }
    handle = mb_malloc(pg_device_handle_pool,0);  //分配泛设备句柄控制块
    if(handle == NULL)
    {//分配不成功
        handle = pan;    //恢复句柄指针
        result = false;
    }else
    {
        handle->dev_interfase = pan;
        handle->iam = en_iam_right;              //我是左手接口句柄
        if(pg_event_running->held_device == NULL)   //事件还没有打开过设备
        {
            handle->next = handle;
            handle->previous = handle;
            pg_event_running->held_device = handle;
        }else                                       //事件已经有打开的设备
        {
            //以下把新设备插入到事件打开的设备队列头部
            handle->next = pg_event_running->held_device;
            handle->previous = pg_event_running->held_device->previous;
            pg_event_running->held_device->previous->next = handle;
            pg_event_running->held_device->previous = handle;
        }
        result = true;
    }
```

```
end_of_dev_open_left_again:
    mutex_post(&tg_dev_mutex);
    return result;
}
```

代码中有比较明确的注释，在这里就不详细解释代码了，程序中用到的资源管理函数 rsc_search_path 和内存管理函数 mb_malloc，请读者自行参考相关章节。

9.4.4.3关闭设备

使用设备应该养成使用完毕后及时关闭设备的习惯，djyos 共有两个 api 函数用于关闭设备，dev_close_left 函数关闭设备左手接口，dev_close_right 函数关闭设备右手接口，函数原型如下：

bool_t dev_close_left(struct dev_handle *handle);　　//handle 是打开设备时获得的指针

bool_t dev_close_right(struct dev_handle *handle);

图 9-13 以uart1 设备为例，说明了关闭设备过程中关键数据结构的主要变化，代码 9-4 则是关闭设备左手接口的代码。 图中有几点需要注意：

1. pg_handle_uart1 指针很特殊，关闭设备后，原来该指针指向的 handle 结构被释放回内存池，而 pg_handle_uart1 指向原来 pg_handle_uart1->dev_interface 指向的设备控制块。我们知道，打开设备的第一步搜索设备树的目的就是要定位设备控制块的地址，而 pg_handle_uart1 保存该地址为以后快速打开设备提供了可能。
2. 关闭设备只释放设备 handle 结构占用的内存，设备控制块并不受影响，只有删除设备才会释放设备控制块。

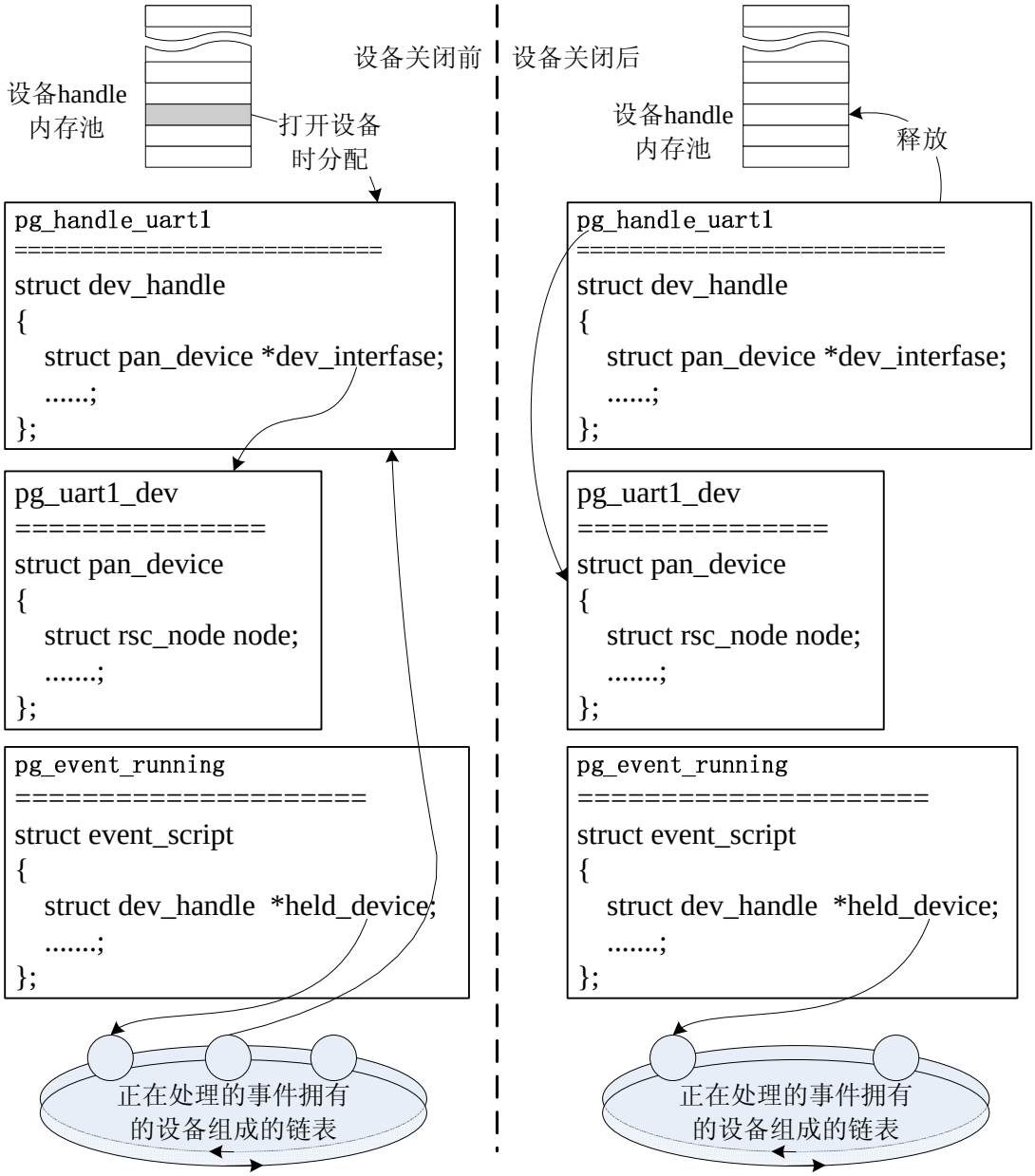


图 9-13 关闭设备过程中数据结构的变化

代码 9-4 关闭设备左手接口函数

```
bool_t dev_close_left(struct    dev_handle *handle)
{
    struct    pan_device *pan;
    if (handle == NULL)
        return false;
    if(handle->iam != en_iam_left)
        return false;
    mutex_pend(&tg_dev_mutex,cn_timeout_forever);
    pan = handle->dev_interfase;
    if(handle->previous == handle)
```

```
    {//事件的 held_device 队列中只有一个设备
        pg_event_running->held_device = NULL;
    }else
    {//held_device 队列中有多个设备,把本设备从队列中取出.
        handle->next->previous = handle->previous;
        handle->previous->next = handle->next;
    }
    mutex_post(&tg_dev_mutex);
    mb_free(pg_device_handle_pool,handle);   //释放内存
    //句柄指针指向原打开设备的泛设备控制块,以备快速打开
    handle = (struct    dev_handle *)pan;
    sem_post(pan->left_semp);
    return true;
}
```

读者可以看到，关闭设备时释放了struct dev_handle结构占用的内存，联系 9.4.4.2 节，在快速打开设备函数 dev_open_left_again 和 dev_open_right_again 中，又重新为 struct dev_handle申请了内存，聪明的读者一定会问，如果在关闭设备是不释放内存，那快速打开函数不是可以更快速地执行吗？我们把在关闭设备是释放struct dev_handle，下次调用快速打开设备时重新申请dev_handle的方案称为方案A，在关闭设备时不释放dev_handle待下次快速打开时再次使用的方案称为方案B。其实djyos开发初期的确是采用方案B的，后来放弃了，也是有其原因的。

方案 A 的缺点是，在快速打开设备时仍然要花时间申请内存，如果用户的内存资源不够充足，还有申请不成功的可能。但是这两方面的缺点并不是致命的，一是 djyos 管理设备 handle 内存池使用固定块分配策略，只要内存池中有内存块可资分配，这种策略分配和释放内存都是快速而且有确定的执行时间；二是只要用户在系统设计时合理分配内存资源，是可以避免 struct dev_handle 内存池容量不足的，完全可以避免从系统堆中增加内存或者分配失败的情况出现。

方案B的缺点比较隐蔽，而就是这种隐蔽的缺陷，可能带来隐藏很深的bug，这种深藏不露的bug是最为可怕的。在 9.4.2.4 节我们已经知道，用户在初始化阶段要为设备handle分配初始化内存池，内存池的容量是cn_handle_limit块st_dev_handle_t类型的结构。最直观地理解，就是无需从系统堆增加内存，这个内存池就可以满足用户同时打开cn_handle_limit个设备。但如果在关闭设备时不释放内存，则使已经释放的设备也占用设备句柄内存块，使允许用户同时打开的设备数减少而且不确定减少多少。由于关闭的设备仍然占用内存是设备管理的内部操作，用户并不知情，他仍然认为自己有cn_handle_limit个设备句柄可以使用，这就可能导致非预期的打开设备失败。在通用计算机上，打开设备失败并不是致命错误，顶多弹出一个信息框，用户大不了关闭某些设备，最多也是重启计算机了事。在嵌入式产品中，可能无法进行人工干预，不能打开设备就意味着产品的部分或者全部功能实现不了，与系统崩溃无异！

方案 B 可能会导致内存泄漏，如果用户反复使用 dev_open_left 打开和用 dev_close_left 关闭设备，而不用 dev_open_left_again 函数的话，struct dev_handle 内存池将很快被耗尽，这就是声名狼藉的内存泄漏。内存泄漏可不是堆的专利，在内存池中也有可能！

“让想当然就是正确的”是 djyos 系统的基本信条之一，用户可能想当然地认为 cn_handle_limit 等于允许同时打开的设备数，djyos 必须使它是正确的。而方案 B 恰恰违反了这一信条。

综上所述，两个方案比较，方案 A 降低了执行效率，但避免了不确定性——cn_handle_limit 就是用户能打开的设备数量；方案 B 速度更快，但可能导致一个隐藏很深的不确定性——用户不知道究竟能打开多少设备。djyos 是一个实时系统，实时系统是非常强调确定性的，因此选择了释放内存的方案。

9.4.4.4快速打开设备在实时系统中的应用

djyos的泛设备管理提供快速打开设备的功能（详见 9.4.4.2 节），可不仅仅是为了提高代码的执行效率，在实时系统中，追求确定性是非常重要的，用标准方式打开设备时，需要沿设备树进行链表搜索，搜索过程中还要比较字符串，整个搜索时间与设备树的大小、目标设备在设备树中的位置、以及各设备名称长度都有关系，执行时间程冗长而且不可预知。这种不确定的执行时间可能会给系统实时性带来致命的伤害。反观快速打开设备函数，无论设备树结构如何，他的执行时间是快速而且确定的。常识告诉给我们，软件领域里确定性和高效性经常是对立的，在这里却同时实现了高效性和高确定性。

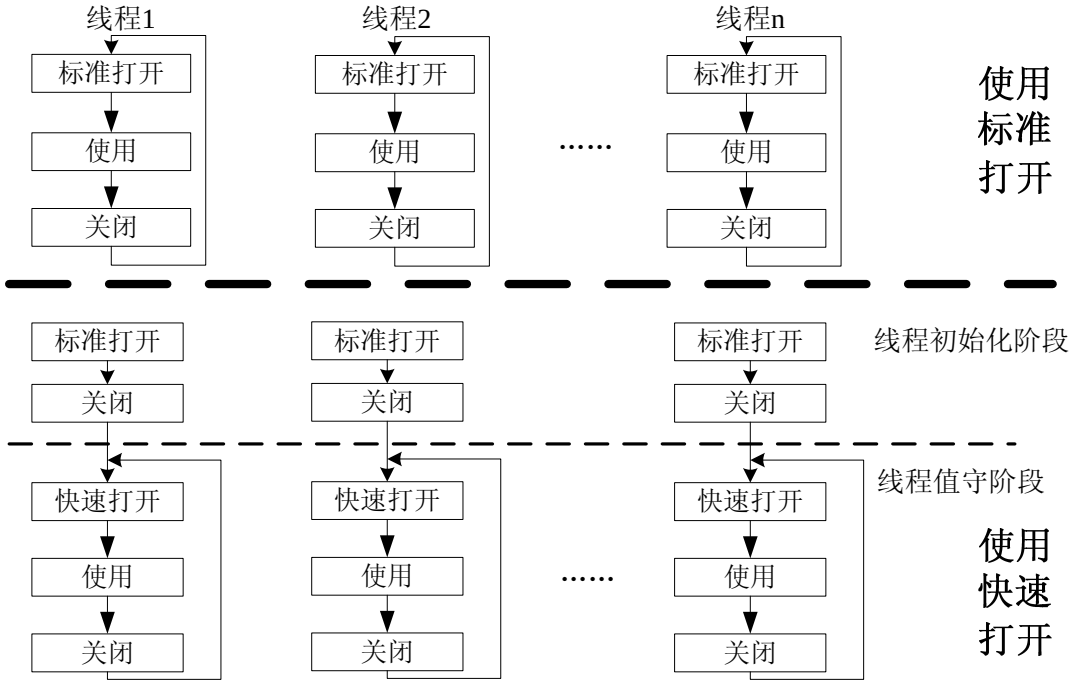


图 9-14 多线程共享一个单用户设备

使用完共享资源后马上释放掉是一个良好的变成习惯，如 图 9-14 所示，当有多个线程使用相同的设备，而该设备又只允许一个或有限个用户打开时（受设备控制块中的信号量的limit_lamp成员限制），则及时释放设备就不仅仅是习惯问题了，而是必须这样做！如果每次打开都使用标准打开，则软件效率大大降低而且执行时间不确定，而且每次打开设备的执行时间都是不确定的，因为两次打来之间可能会有别的线程创建了新设备、或者删除了设备。而使用快速打开则不同，线程只在初始化阶段执行一次标准打开，在值守阶段执行的是快速打开，快速打开是高效而且执行时间确定。

9.4.4.5防止资源泄漏

djyos 系统中设备并非只为硬件模块服务，它可能用于软件模块与软件模块之间交换数

据，而且操作系统鼓励程序员在模块间使用设备来交换数据，所以 djyos 系统中编程会比在其他系统中更加频繁使用设备。djyos 系统的调度是以事件为基本单元的，当事件发生时，临时创建执行线程，一旦事件处理结束，线程的生命周期也就结束了。如果事件处理线程需要使用设备，就要求在执行线程中打开设备，在线程结束前关闭设备。打开设备和关闭设备的操作应该成对出现，如果打开次数多于关闭次数，就会产生资源泄漏，设备句柄控制块无法收回，设备的信号量也被无端消耗，严重时会导致系统崩溃。

下列代码将会导致指针丢失，进而导致不可预料的后果。

struct dev_handle　my_handle;

my_handle　= dev_open_left（“uart0”，0）;

由于程序错误，my_handle 被意外修改;

dev_close_left（my_handle）　　//意欲关闭 uart0。

如果应用程序中出现以上执行序列，关闭设备时将导致不可预知的错误。函数 dev_close_left 中，只检查也只能检查 my_handle 是否空指针，只要不是 NULL，都会当作合法指针进行访问，我们知道，访问空指针会导致不可预测的后果。

另一种典型错误是，线程结束时仍有未关闭的设备，djyos系统对这种错误做了保护，事件处理完成后，线程结束前，操作系统会检查该线程是否有未关闭的设备（包括指针丢失的设备），如果有，则强行关闭之。事件控制块有一个成员 struct dev_handle *held_device，held_device指向一个由设备handle组成的双向循环链表，该链表是用struct dev_handle结构的previous成员和next成员组成的。参见 图 9-13，线程每次打开设备，就把新的handle结构插入到该链表中，而关闭设备则把它从链表中删除。如果一个设备被多次打开，就会多次加入，即使出现指针丢失的现象，链表中的节点也仍然存在。线程结束前，操作系统将查看held_device队列，如果非空，则强制关闭该队列中的所有设备，由dev_cleanup函数完成设备清理工作。

代码 9-5 设备清理函数

```
void dev_cleanup(struct　event_script *event)
{
    if(event == NULL)
        return;
    while(event->held_device != NULL)
    {
        if(event->held_device->iam == en_iam_left)
            dev_close_left(event->held_device);
        else
            dev_close_right(event->held_device);
    };
}
```

9.4.4.6设备的读、写、控制

读、写、控制函数是设备实现功能的关键，参考 图 9-1 djyos的设备模型，完整的设备接口需要实现左手接口和右手接口两套共 6 个接口函数，设备对外提供的功能全部由这两组函数实现。实际设备不一定需要全部实现这些接口函数，比如终端显示设备就只需要实现左手接口，右手接口的所有函数都用y_empty_func函数替代，注意不要使用NULL替代空函

数，在有些嵌入式系统中，调用NULL函数指针会导致系统复位；还有些设备左右手接口共用一个dev_ctrl函数。图 9-15 和 图 9-16 显示了从driver设计者实现设备的my_left_read函数到用户通过dev_read系统调用执行my_left_read函数的全过程，dev_write和dev_ctrl函数操作与此类似。代码 9-6 则展示了操作系统如何实现dev_read的，代码 9-7 则选取了一个非常简单的uart设备的左手读函数，作为设备driver实现my_left_read函数的典型范例。

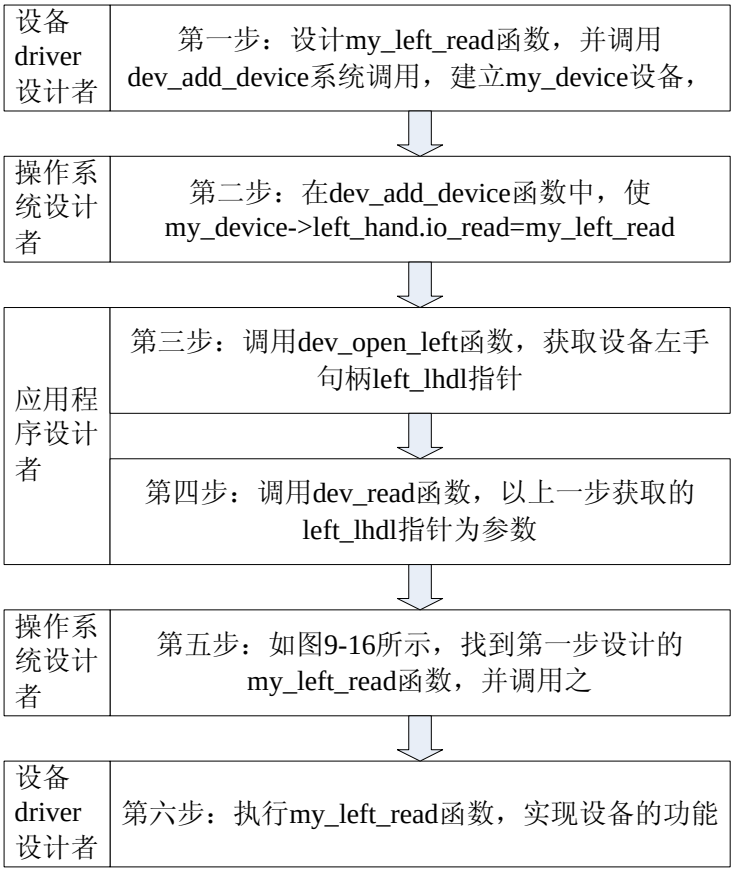


图 9-15 实现左手读设备

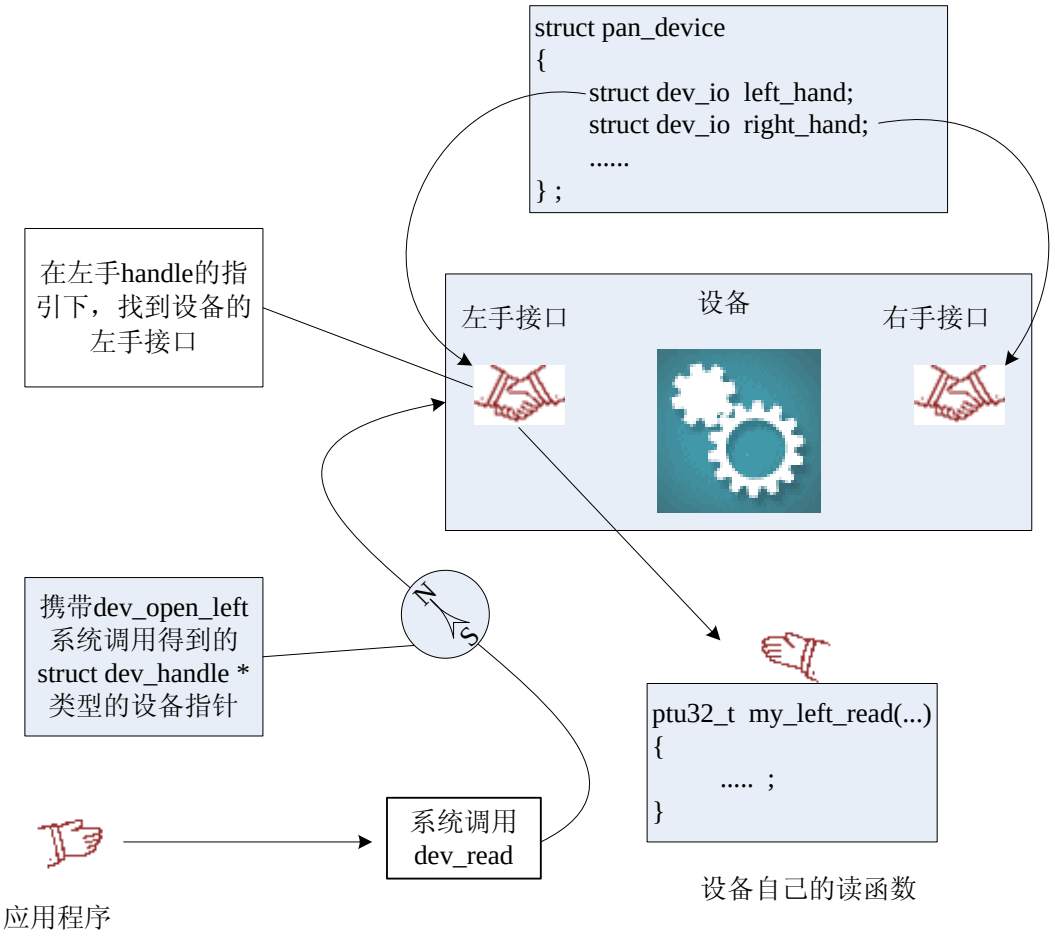


图 9-16 左手读设备的调用路径

代码 9-6 dev_read 系统调用

```
ptu32_t dev_read(struct    dev_handle *handle,ptu32_t src_buf, ptu32_t des_buf,ptu32_t len)
{
    if (handle == NULL)
        return 0;
    if(handle->iam == en_iam_left)
        return (handle->dev_interfase->left_hand.io_read (handle,src_buf,des_buf,len));
    else if(handle->iam == en_iam_right)
        return (handle->dev_interfase->right_hand.io_read (handle,src_buf,des_buf,len));
    return 0;
}
```

1. 代码 9-6 中的参数handle是由打开设备函数（dev_open_left或者dev_open_right）返回的指针。
2. dev_read 函数在 handle 参数的指引下，找到被读的实际设备 dev_interfase。如果该 handle 是左手接口 handle，则调用左手接口 io_read 函数指针，该指针指向由设备 driver 设计者实现的 my_left_read 函数，否则调用右手接口的 my_right_read 函数。
3. handle 仍然作为参数传递给 my_left_read 函数，否则，my_left_read 函数就可能不知道自己被哪个设备调用。也许会有人问，my_left_read 是泛设备 driver 的作者实现的，怎么会不知道自己是哪个设备的 my_left_read 函数呢？这种可能性是存在的，

可能有多个设备的左手接口 io_read 指针指向同一个 my_left_read 函数，这种情况下，my_left_read 函数被调用时，函数本身是不知道自己被谁调用的，所以也就不知道该从哪个设备读数据了。典型的案例是，系统有多个网络口，对应有多个网络设备，但是从网络设备读数据的函数 my_left_read_net 函数只有一个，所有网络设备的读函数 io_read 指针都指向这个函数，my_left_read_net 函数被调用时，只有借助 handle 指针才能确定具体读哪一个网络设备。

代码 9-7 uart 设备的左手读函数

```
ptu32_t uart_left_read(struct    dev_handle *uart_ldev,ptu32_t buf, ptu32_t res,ptu32_t len)    //1
{
    struct    pan_device *uart_dev;
    struct serial_DCB *serial_uart;
    uart_dev = uart_ldev->dev_interfase;                                    //2
    serial_uart = (struct serial_DCB*)uart_dev->private_tag;        //3
    return ring_read(&serial_uart->recv_ring_buf,(uint8_t*)buf,len);        //4
}
```

1. uart_left_read 函数的原型必须与 driver.h 文件中定义的函数指针 dev_read_func 的类型一致，才能正确地实现参数传递。
2. 注释 2，从设备 handle 指针中取得设备控制块，本函数由多个串口设备共用，必须以这种方式才能确定用户读的是哪一个串口。
3. 除 handle 参数以外，其他参数都是 ptu32_t 类型的，用户可以作为 32 位变量使用，也可以转换成指针使用，其实际类型由 driver 的设计者确定，在 uart_left_read 函数中，buf 参数的用途是用户提供的缓冲区指针，len 的含义是读数据的长度，0 表示全部读取，res 参数保留，这些约定应该在软件说明书中详细说明，用户使用设备时应该遵循设备的使用说明文档。
4. 注释 3，从设备控制块中取得特定串口的专用数据结构。
5. 注释 4，从串口的环形缓冲区中读取数据。

9.5 内窥镜泛设备

乍一看本节标题，似乎笔者要设计医疗设备，其实不然，造成这样的混乱，实非本意，因为我实在想不到有什么比这更贴切的词汇了。

为便于调试，djyos 设计了一个内窥镜模块，用来窥探内核的秘密。该模块使开发者能够通过 PC 机显示内核的信息，像打开的文件，用户注册的事件类型，就绪队列中的事件，闹钟同步队列中的事件，系统中存在的信号量，各信号量同步队列中的事件等等信息，都可以通过内窥镜模块发送到 PC 机，只要在 PC 端设计适当的软件，就可以由 PC 机显示。

内窥镜模块由一个内窥镜泛设备为核心组成，PC 下发的命令通过通信设备传送到内窥镜设备，而内核数据则由内窥镜设备发送给通信设备，再由通信设备发到 PC 机。

第10章　看门狗

10.1　看门狗的硬件

为了提高系统可靠性，嵌入式系统一般都会设计看门狗电路，在软件严重错误、或者硬件受干扰而使系统运行出错的时候，看门狗电路将输出复位信号使系统恢复正常运行。看门狗有CPU内嵌和外置两种，无论哪一种看门狗电路，其原理不外乎如 图 10-1 所示。软件正常运行时，定期执行看门狗时钟的清零（俗称喂狗），喂狗信号因软件异常而停止后，看门狗时钟在设定的时间到来时，发出超时信号，这个信号一般作为复位信号。

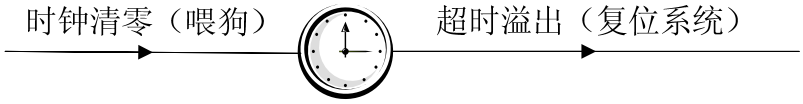


图 10-1 看门狗电路设计原理

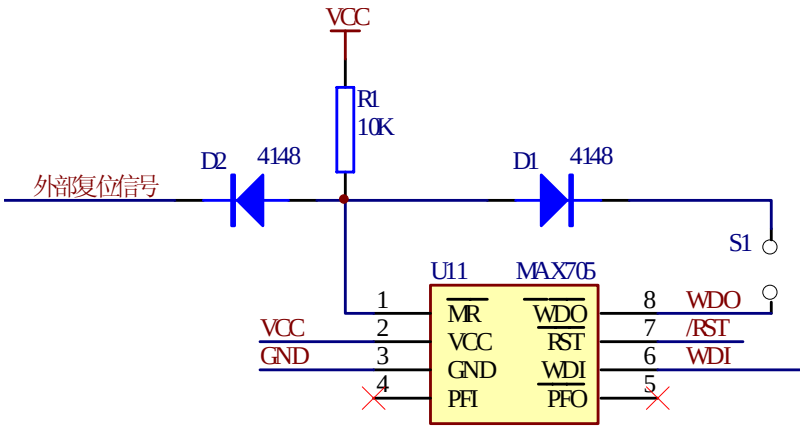


图 10-2 典型外置看门狗电路图

图 10-2 是典型的外置看门狗电路图，MAX705 内置两个定时器，一个用于控制看门狗溢出时间，时间常数固定为 1.6 秒，WDI信号的每次改变都会清零定时器，如果wdt信号连续 1.6 秒没有变化时，MAX705 的 7 脚就输出一个低电平脉冲表示看门狗溢出，该信号用于复位计算机。另一个定时器控制看门狗溢出脉冲的宽度，时间常数固定为 200mS。与大多数看门狗芯片一样，MAX705 兼具电源监测功能，图中没有画出。也有一些看门狗芯片的定时时间是可以调整的，设计时可以根据需要选用。断开掉线S1 可以禁止看门狗复位，用于调试。

10.2　看门狗的软件原理

看门狗是典型的软件和硬件协作的电路，在系统发生严重错误不能自我恢复的时候，复位整个系统。在是否需要复位系统的判据方面，软硬件是独立的，硬件判据比较简单，饿急了就咬，而软件判据就比较复杂了。不管是高贵的钻石还是丑陋的石墨，都是由一个个的炭原子组成，在原子级别，最高贵的钻石和最丑陋的石墨并无二致。计算机系统的运行过程也一样，不管软硬件的复杂度如何，无论是操作系统支持下还是直接在裸机下运行的程序，都

可以抽象成 图 10-3 的形式，不同的是分支的多少、条件判断的复杂度、各种循环执行的次数和频度，以及中断发生的频度等。

图 10-3 中可见，为了确保系统的正常运行，我们要找出软件中的若干个关键路径，只要这些关键路径正常执行到，基本上可以保证软件是正常的。看门狗软件就是在这些关键路径上设置适当的判定条件或标志，通过分析这些判定条件，可以判断各关键路径是否正常执行。通常由一个看门狗监察模块分析这些条件标志，当软件正常运行时，由监察模块执行喂狗，当监察模块分析到某些关键路径执行不正常时，就会调用相应的善后处理程序将作出相应的处理，使软件恢复正常运行。如果出现不可恢复错误，监察模块就会停止喂狗，看门狗硬件在看门狗定时器溢出时，输出复位信号，强行复位系统。如果监察模块本身不能正常执行，自然也就无法正常喂狗，也会引起硬件复位。

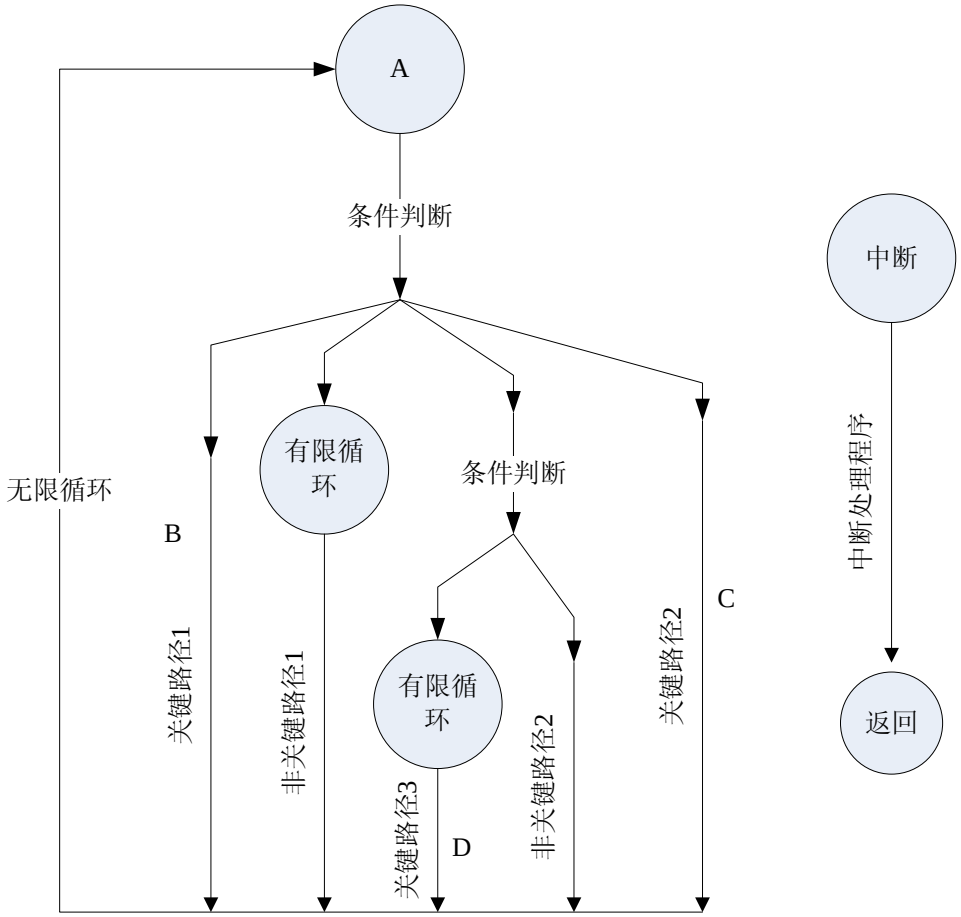


图 10-3 软件典型抽象

在很多人的印象中，看门狗是用来防死机的，死机就是软件意外进入死循环，不执行任何用户程序，也不响应用户的操作，在 图 10-3 中表现为A点执行不到，导致其他所有关键路径都执行不到。最简单的防死机的方法是在 图 10-3 的A点放置一条喂狗指令，只要主循环执行一圈的时间在看门狗定时器时间常数之内，就会照常喂狗，看门狗电路就不会输出复位脉冲。如果A点执行不到，就意味着无法喂狗，看门狗电路在定时时间到就会复位系统，很多看门狗设计用的就是这种方法。

从 图 10-3 中我们还可以看到，仅在A点清看门狗还不足以保证系统的可靠性，由于软硬件意外，还可能导致一些条件标志被意外修改，致使某些关键路径不能执行到或者不能按设定的条件执行，完善的看门狗处理称序应该能兼顾所有的关键路径。关键路径的判定条件

可以因地制宜，灵活制定，例如，图 10-3 中B点与D点有一定的依赖关系，程序每执行D路径 10～20 次就必然会执行B路径一次，那这种关系就可以作为判定条件，看门狗监察模块分析这个条件是否满足就可以确认B路径是否正常。图 10-4 是一个比较完整的看门狗监察模块的流程图。

对于没有操作系统支持的嵌入式软件，看门狗监察模块可以放在主循环中调用，但必须确保主循环的执行周期小于看门狗定时器的溢出周期。如果主循环执行时间长于看门狗溢出周期，也可以在周期性的定时器中断中执行。在有操作系统支持下的嵌入式软件，则可以建立一个看门狗守护事件，该事件周期性地执行看门狗监察程序。需要注意的是，无论是定时器中断还是中期性执行的事件中，都是执行看门狗监察程序，而不是直接喂狗。

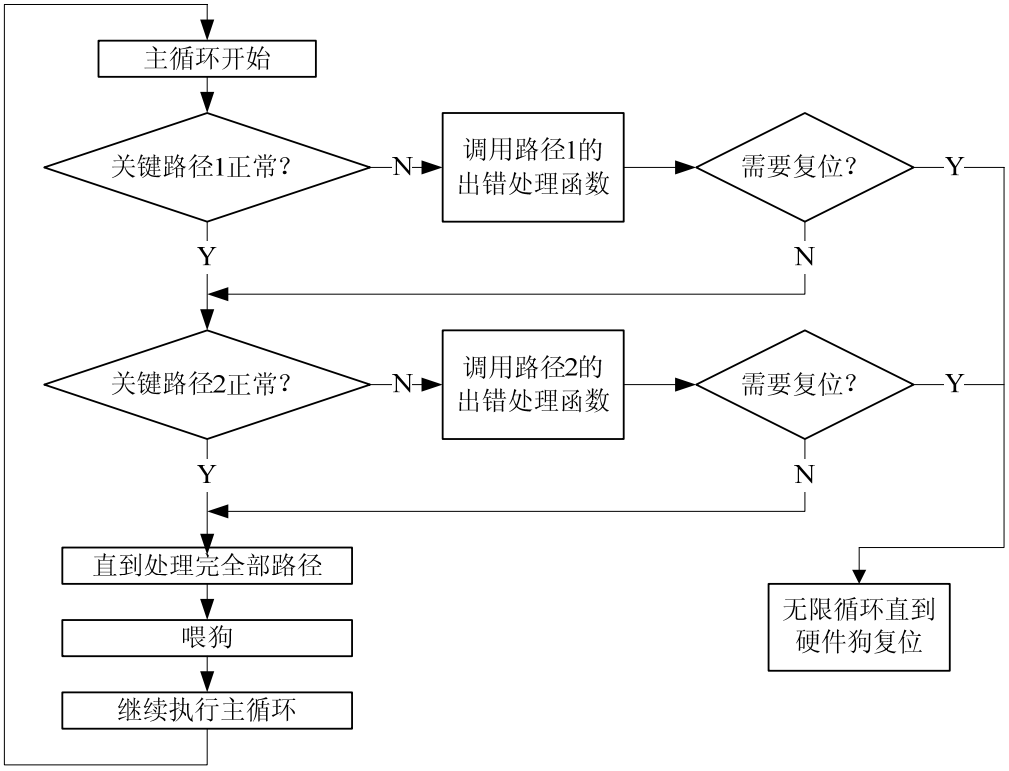


图 10-4 抽象的看门狗软件结构

10.3 看门狗设计的常见误区

10.3.1 为了喂狗而喂狗

这是笔者亲眼所见的一个例子，由于某产品在现场运行时出现死机现象，通信、显示和键盘操作均没有任何反应，但是看门狗却不动作。在实验室中对该产品施加强烈干扰，则反复出现这种现象，在死机的时候，测得看门狗的喂狗信号是正常的。家里失窃了，狗却欢快地啃着骨头，对窃贼视而不见！在读代码时，发现该产品竟然专门使用一个定时器用于喂狗，在周期性的定时器中断里只干一件事：喂狗。于是就问软件工程师：

“timer1 中断是用来干什么的？”

“喂狗啊。”

“为什么这样喂狗？”

“李工说要保证每 1.6 秒喂一次狗，否则看门狗就会复位 CPU。”（李工是该产品硬件工

程师）。

在这里，看门狗没起作用也就罢了，还纯粹成了负担！要知道，与庞大的应用程序相比，定时器中断函数的代码量是微不足道的，在绝大多数情况下，应用程序出现严重故障而必须复位 cpu 时，定时器中断代码仍然完好，定时器相关寄存器的设置仍然正确，定时器中断程序仍然会准时喂狗，看门狗将因此失效！打个比方，这种喂狗方式相当于专门修一个狗屋把狗圈起来，并且专门为狗准备充足的骨头，目的只有一个，就是无论发生任何事情，只要这个小屋子不被砸掉，狗就不能叫。即使盗贼把整个屋子都搬空了，只要不去抢狗屋的骨头，看门狗就会有滋有味地啃它的骨头。

10.3.2 过度喂狗

猫吃得太饱就不会去抓老鼠，同样，狗吃得太饱了也会丧失看门的本领。有些程序员编程时在代码中间随意使用喂狗指令，他们的目的只有一个，就是让看门狗闭嘴，好让蹩脚的软件运行下去。这样，即使程序在某些地方陷入了死循环，但由于该循环处有喂狗指令，看门狗也是不会动作的；或者程序虽然没有陷入死循环，但是需要看护的关键点没有运行到，而由于在非关键点处把狗喂饱了，也是不会动作的。

10.3.3 过分倚重看门狗

有些设计者以为设置看门狗以后就可以万事大吉了，其实并不是这样的。

看门狗就像汽车保险，买了保险后，只能保证发生事故后修车有人出钱，疗伤有人买单，但是并不能保证你不出事故，也不能保证你在事故中不受伤害，重要的还是自己要遵守交通规则，小心驾驶。看门狗也一样，它只能保证软件中所有关键路径按设定的方式执行到，但这并不代表软件或者硬件正确地完成了工作。看门狗看家护院的本领依赖与关键路径，严密的关键路径和恰当的判断条件可以使看门狗发挥最大效能。

看门狗只能在软件出现严重错误时复位系统，但本身不能善后因错误产生的后果。一个严谨的嵌入式系统设计，应该在 CPU 复位后，软件能够诊断复位原因，软件也要根据诊断结果做相应的处理。

总之，看门狗只能是飞机的黑匣子，它虽然能够提供信息以帮助分析事故和改进飞机以及空管，但本身不能对飞行安全提供任何帮助。看门狗也一样，设计良好的看门狗可以提供信息以帮助设计者分析故障和改进设计。

10.3.4 定时器中断喂狗的特例

看门狗是通过保证程序执行流程能够正确地经过设定的关键点来保证系统可靠性的，从 10.3.1 节可知，在一般情况下，是不能在定时器中断响应函数中无条件喂狗的，推而广之，任何会周期性发生的中断响应函数里，均不能无条件喂狗。

是不是任何情况下都要禁止在周期性中断中喂狗呢？也不尽然，如果定时器中断处理函数就是整个系统最关键的部分，那么中断函数就成了 图 10-3 中的关键路径了。如果这是整个系统的唯一关键路径，就可以在此直接喂狗；如果只是多条关键路径中的一条，就应该在此放置一个喂狗判断条件，使之成为 图 10-4 中监控的关键点。在陀螺仪控制器中，需要周期性地检查目标倾斜度，并相应地修正陀螺仪的姿态。启动一个硬件定时器，姿态监控模

块在定时器中断服务函数中执行，只要保证中断服务例程正确执行，就可以保证整个系统的关键部分是正常的，在这种情况下，就应该把喂狗指令直接放在定时器中断服务函数中。

10.4 djyos 系统的看门狗模块

10.4.1 看门狗模块初始化

djyos系统提供了看门狗模块，应用程序调用看门狗服务可以添加看门狗，看住程序中的关键路径，使用看门狗的模块称为狗主人。看门狗数据结构定义和初始化过程如 代码 10-1 所示。

代码 10-1 看门狗数据结构与初始化

```
struct wdt_rsc                                                          //1
{
    struct rsc_node wdt_node;
    struct semaphore wdt_semp;                                          //2
    bool_t (* judge)(void);             //用户提供的判断狗叫的函数      //3
    uint32_t (* yip_remedy)(void);      //狗叫后的补救措施，用户提供    //4
    uint32_t    timeout;                //调用 judge 的时间间隔         //5
};
bool_t module_init_wdt(void)
{
    static struct wdt_rsc wdt_root;
    uint16_t wdt_evtt;
    __wdt_init_hard();
    pg_wdt_rsc_pool = mb_create(&tg_wdt_rsc_pool,cn_wdt_limit,
                                sizeof(struct wdt_rsc),"wdt pool");     //6
    pg_wdt_rsc = (struct wdt_rsc*)
        rsc_add_root(&wdt_root.wdt_node,sizeof(struct wdt_rsc),"watch dog");    //7
    __sem_create_knl(&wdt_root.wdt_semp,1,1,"watch dog");
    u16g_wdt_evtt = y_evtt_regist(true,false,cn_prio_wdt,1,wdt_check, 100,"wdt service"); //8
    if(u16g_wdt_evtt != cn_invalid_evtt_id)
        return true;
    else
        return false;
}
```

代码注释如下：

1. 诚如其名，看门狗是一种资源，每只看门狗都是系统资源链表中的一个结点，所有看门狗都是根节点“watch dog”的子结点。
2. 看门狗以队列方式管理，该队列属于公共数据，访问时需要用信号量保护。
3. 每个狗主人应该为他的看门狗提供判断看门狗所看护的代码是否正常的回调函数，看门狗模块本身不判断看门狗是否正常，而是按 timeout 设定的时间间隔调用该函数，由该函数告诉看门狗模块本狗看护的代码是否正常。如果不正常，看门狗将鸣叫（yip）。
4. 狗主人还应该提供狗叫后的善后回调函数，如果看门狗叫（judge 函数返回 false），善

后函数将被调用。这个函数执行必要的善后工作，返回值有两种可能：cn_wdt_action_none 表示无动作，cn_wdt_action_reset 表示要复位计算机，一般是看门狗模块故意停止喂狗，等待看门狗定时器溢出。

5. 调用 judge 函数的时间间隔，这个间隔很重要，有许多看门狗是用时间作为判定条件的，即只要执行关键点的时间间隔在允许范围以内，就认为正常，狗就不叫。用户创建看门狗时，设定时间间隔用 mS 为单位，该时间会自动被向上调整为整数个 tick。
6. 创建看门狗控制块内存池，该内存池是静态定义的数组：
   static struct wdt_rsc tg_wdt_rsc_pool[cn_wdt_limit];
   上创建的，按固定块方式进行分配（参见第 7.3 节）。cn_wdt_limit是在port_kernel.h文件中定义的常量，规定系统最多允许创建看门狗的数量。
7. 添加看门狗资源根节点，用户养的看门狗都是它的子节点。
8. 在这里登记一个默认优先级 cn_prio_wdt=1 的“wdt service”事件类型。创建第一个看门狗的时候会弹出该类型事件，该事件周期性执行，定时常数是所有看门狗的溢出时间常数的最大公约数。软件设计时应该注意，由于看门狗事件的优先级是 1，属所有用户可使用的优先级（1~249）的最高级别，执行不宜过于频繁。所以，timeout 不建议设置太小的值，最好是整十或整百的倍数，这样他们将有较大的公约数。

10.4.2 看门狗服务执行流程

在创建第一只看门狗以后（包括删除前部看门狗后再次创建看门狗），看门狗模块将弹出看门狗事件，该事件的线程周期性地执行直到最后一只看门狗被删除。看门狗服务只管按每只看门狗的 timeout 时间周期性地调用该看门狗的 judge 回调函数，如果狗叫（judge 返回 false），再调用善后回调函数，最后按善后函数的返回结果，决定是否继续喂狗。

线程利用闹钟同步实现周期性执行，执行周期是所有看门狗 timeout 时间的最大公约数，这样就可以使每一只看门狗定时周期到的时候，线程都会被唤醒。

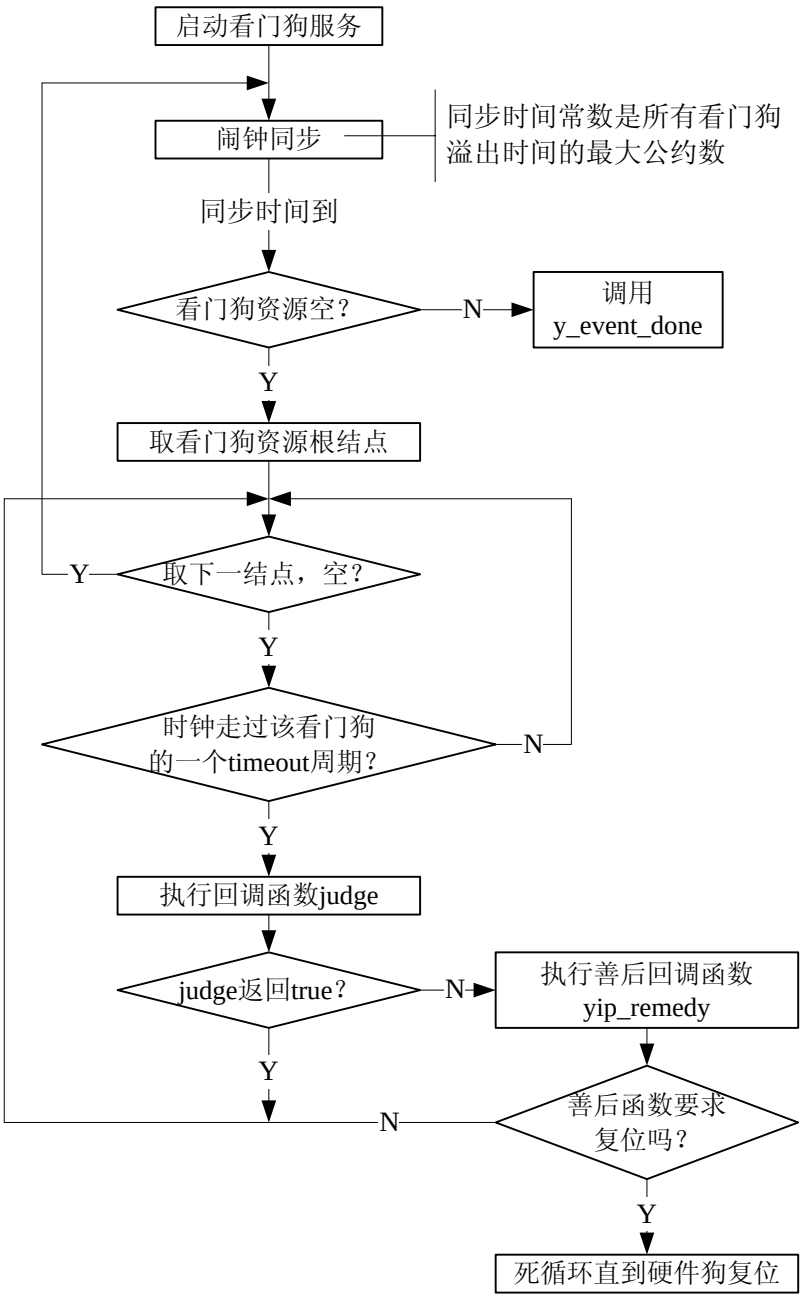


图 10-5 看门狗事件处理流程

10.4.3 创建和删除看门狗

wdt_create 函数用于创建一只看门狗并加入到资源队列中，函数原型如下：

```
struct wdt_rsc * wdt_create(bool_t (*judge)(void),uint32_t (*yip_remedy)(void),
                            uint32_t timeout,char *wdt_name);
```

详细参数说明参见 第 17 章 api参考手册，该函数的流程如 图 10-6 所示。

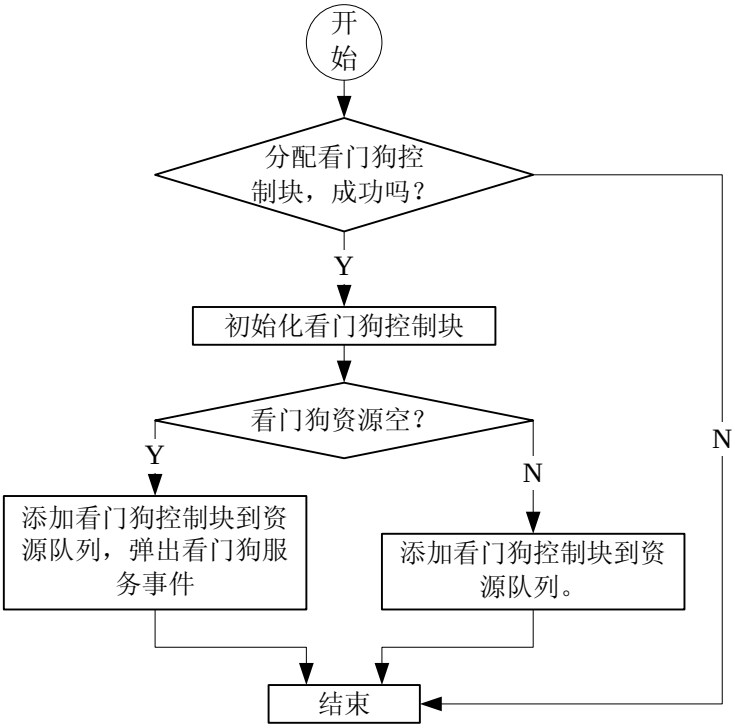


图 10-6 创建看门狗流程

wdt_delete 函数用于删除一只看门狗，该函数原型是：

void wdt_delete(struct wdt_rsc *wdt);

该函数相当简单，这里就不画流程和列代码了，该函数完成的工作就是：从看门狗资源队列中删除该看门狗结点，重新计算溢出周期的最大公约数，释放看门狗控制块内存。

10.4.4 使用看门狗示例

第11章　组件化开发（本章未完成）

11.1　可移植是软件的灵魂

软件的可移植性指软件在其运行的软硬件环境发生变化时，不加修改或者经过简单修改就能适应新环境的能力。这种能力体现在两个方面：

1. 当硬件和软件环境发生变化时，软件无须修改或者少量修改，重新编译就能够适应的能力。
2. 当软件的需求发生局部变化时，软件经过修改少量代码，或者修改配置文件或硬件跳线就能满足新需求的能力。这一点尤为重要，因为嵌入式产品都是个性化产品，甚至是工程化的产品，常常会根据用户现场配置的不同而调整产品功能，而这些调整大多需要软件配合。不但如此，软件还得做好不断应付客户提出的新需求的能力。

可移植的代码意味着功能模块可以被重复利用，实在是软件开发进步的灵魂，程序员编写的程序被反复移植利用，就象文人有了传世之作，是非常值得荣耀的事。从计算机发明至今，不，应该是从人类进化至今，重复利用资源的能力始终是人类进步的重要推动力，软件的可移植性自然也是推动计算机应用进步的最重要的动力。

不知多少年前，人类的祖先学会了挑选、收集和使用工具，重复使用工具使得自然界的天然资源得以重复利用，使其在与其他更加强壮的动物竞争中占得先机，迈出了向人类进化的第一步。

又过了不知多少年，大约在数十万年前，人类祖先学会了制造和使用工具，使得可以重复利用的资源的范围极大地扩大，这反过来又大大地促进了人类的进化。自此至今，虽然从生物学的角度上讲，几十万年间人类的进化是很少的，但人类文明的发展却突然加速，而且加速度越来越快。

随后，农作物的种植、动物的驯化、人畜动力机器、机械动力机器的发明，无不是在扩充重复利用资源的范围，而这些资源的扩充反过来又在加速了人类社会的发展。

人类语言和文字的发明，使人类积累的知识通过传播得以重复利用，使人类在万物之灵的道路上越走越远。

古老中国发明的筹算和珠算，以及西方 17 世纪发明的机械计算机，是可重复使用的计算工具，对当时的商业贸易、生产统计工作起了不可估量的作用。

18 世纪到 20 世纪中期发明的穿孔机、分析机、差分机，以及后期出现的更加自动化的机械计算机，同时期布尔代数理论的大发展，进一步促进了可以重复使用的计算机械的发展，也为以后电子计算机的出现奠定了坚实的理论和实验基础。

20 世纪中期到现在电子计算机的发展，促进了人类信息技术和工业技术的空前繁荣，人们可以编制程序，利用计算机高速重复运算的能力，完成人类靠自身 1 万年也完成不了的运算。

早期的电子计算机使用二进制（汇编语言是二进制的可阅读版本）编制程序，这样编写的程序只能在相同指令集且配置相似的机器之间重复利用。

很快，人们就发现了用汇编语言编程的缺陷，发明了各种各样的高级编程语言，高级编程语言的出现，使得软件通过重新编译就可以在不同指令集的机器之间移植，使代码的可重复利用率一下子扩大了许多。

这还不够，革命性的操作系统的出现，它通过硬件抽象屏蔽了硬件的差异性，提供给程序员标准的接口，使编写可跨平台移植的程序变得非常容易。同时，操作系统提供了大量的标准化的系统调用和库函数，这些代码都是可以被所有使用该操作系统重复使用的代码，使得程序员可以专心于解决具体问题的算法。

代码可移植性的好坏，是衡量代码质量的重要标准，是程序员智慧的集中体现。一个优秀的操作系统，应该能够帮助程序员编写可移植的代码，并且能最大限度地兼容个性化的代码，djyos 通过泛设备驱动程序的概念协助程序员设计可移植性的程序，并且用泛设备驱动程序结构容纳个性化的代码。

拥有一个稳定的系统架构，大量经过测试的、可移植性强的代码是一个有长远目标的企业的巨大财富。可移植性强的代码可以轻松地被新产品继承，而代码的继承可以加快产品设计速度，减少测试时间，减少 bug 存在的机会。通常，如果一个 bug 在同一个企业的不同产品上重现的话，该企业软件的可移植性一定不佳。

11.2 组件化的困惑

传统的软件开发是垂直型的，整个软件由一个独立的团队开发，从软件规格书的指定到框架设计到详细设计到代码编写到测试定版，都由这个团队完成。在传统模式下，应用程序的开发一般是从“零”开始的，开发中最多也就是引用一些现成的函数的代码而已。组件化开发涉及到如何把一个完整软件拆散分配给不同团队并把各个模块如何组装在一起的问题，这就需要一套比较完整的体系框架和接口做保证。这种搭积木的方式要求我们把项目中可以重用的部分识别出来并且单独加以封装，能够用到今后项目中，但传统操作系统不提供直接支持，定制重用模块接口的成本太大，对程序员的经验和水平的依赖性高，往往最后导致愿望很好、实现起来很困难。如果没有操作系统的特别支持和引导，容易导致了系统开发周期长，开发费用大，可移植性差，可靠性低等问题，只有在经验特别丰富的系统设计师的引导和干涉下才能避免这种问题，但是，“天才是可遇不可求的，依赖天才的研发是失败的”。

Ivica Crnkovic 在《基于组件的软件工程——软件开发中的新挑战》一文中认为开发可重用组件的困境在于：

“·开发组件所需要的时间和精力。在所有阻碍可重用组件开发的因素中，比较重要的是对时间和精力增长的需求。构建一个可重用的组件需要三至五倍于开发满足特定需要组件的时间和精力。”

“·不确定和模糊的需求。通常，需求管理是开发过程中一个重要的方面。其主要目的是定义一个一致和完整的组件需求。可重用组件被定义，然后应用在项目中，其中一些应用和需求难以预见，包括功能性和非功能性的。”

“·可用性和可重用性的冲突。为了更为广泛的应用，一个组件必须足够的全面综合、可升级和易于维护，这导致了组件更为复杂（给使用也带来了一定的难度），对计算资源更多的消耗需求（使用所需要的花费也上升了）。对可重用性的需求可能导致转向其他的开发途径，如采用一个新的较为抽象的开发层次，这会减少弹性和可微调性，但更好的实现了简洁性。”

“·组件维护所需的花销。应用软件维护所需要的花费可能会很低，但组件维护所需要的花费会很高。这是因为组件必须和在不同环境下的不同应用的不同需求相一致，这包括不同的可依赖性的需求和可能需要不同级别维护支持的需求。”

“·可靠性和对挑战的灵敏性。由于组件和应用软件有独立的生命周期和不同种类的需求，所以存在组件不能完全满足应用软件需求、或可能具有某些应用软件开发者也不确定的隐藏特征的风险。在介绍应用程序级别的挑战中（比如操作系统的更新，其他组件的更新，应用软件的挑战等等），以上介绍的挑战可能导致系统崩溃的风险。”

综合来说，组件开发的困难在于需求的不明确性和组件功能的复杂性。明确需求的困难在于，定义可重用组件有定义未来需求的意义，而未来需求究竟是什么样子，只有天知道。组件功能的复杂性在于，组件不是为特定的需求而开发，它要适应所有应用本组件的应用程序的所有变化，既有应用软件的客户需求的变化，又有软件运行平台的变化。让我们听听开发很多项目之后的、有经验的研发工程师或者研发主管的声音吧：“这么多项目，很多功能模块都是重复开发，浪费了大量人力物力，如果类似功能的代码能够重用就好了”。也有说：“这个模块，客户的需求只是变化了一点点，我们却要同时修改那么多的模块，工作量那么大，还容易出错，要是各个模块能够独立些就好了”。要求模块重用和独立，这不是组件化研发是什么？与第三方的、商用或者开源的组件相比，这可以称作企业级组件，是更为普遍的、广泛存在的组件化开发。因此，组件化的开发已经渗透到软件开发的方方面面，而不限于商用组件了。

与通用组件相比，企业级组件需求不明确性和功能的复杂性均大大降低，

作为软件开发基础平台的操作系统，为组件化的开发提供支持，是责无旁贷的。有鉴于此，djyos 操作系统从内核开始提供组件化开发的支持。组件的功能是否能重用，这是组件需求制定者的责任，这与操作系统实际上并无多大关系，操作系统的责任主要在于为组件的独立性提供支持。djyos 系统对模块的独立性做了充分而全面的支持，引导不是非常有经验的程序员写出独立的代码，在普适计算时代，这点尤为突出。未来需求的不明确性和多样性始终存在，制定需求时总不能做到面面俱到，因此独立性就显得尤为重要，独立性做好了，组件就可以独立升级。

从技术研发和版本控制以及生产管理的角度，组件的独立性分几个层次：

独立编译独立连接。

独立编译一起连接。

独立修改配置、独立编译连接。

独立修改配置、共同编译连接。

一起修改配置、一起编译连接。

独立修改代码、独立编译连接

独立修改代码、一起编译连接

非独立修改代码、当然一起编译连接。

11.3 耦合

硬件电路有耦合的概念，包括确保有用信号的顺利耦合和阻止无用信号的耦合，比如多级放大器的前级把信号耦合到后级，以及高速电路数据信号、地址信号、控制信号的耦合，都是有用信号的耦合，是良性的，设计电路就要考虑让信号尽可能完整地耦合过去；而干扰信号（包括外部干扰和单元电路本身产生的干扰）也可能相互耦合传递，这就是恶性耦合，

我们需要设计去耦电路来阻止恶性耦合，并提高电路的抗干扰性能来消除恶性耦合产生的后果。以上两者，在电子学中是相当复杂的问题，“信号完整性”是研究良性耦合的专门学科，“电磁兼容”是研究恶性耦合的一门学科，这两门学科都是相当复杂而且仍在发展中的学科。耦合的概念也可以用在软件中，在同一个计算机系统中运行的程序，将不可避免地发生互相作用而互相影响，我们称这种影响为软件模块间耦合。与硬件相呼应，耦合分良性耦合和恶性耦合。一个模块运行可能产生一些数据、或者操作某些硬件，而这些数据或者操作硬件的结果，又成为另一个模块的输入数据，从而间接地影响了第二个模块。而两个模块互相之间却并不知道这种耦合的存在，第一个模块只管输出数据，它并不知道数据给了谁，第二个模块则只管取数据，它并不知道数据是谁给的。这种耦合就是良性耦合，因此，良性耦合一般来说都是间接的耦合。而模块间互相使用对方的数据，或者要求对方必须如何如何做，从而使模块间互相依赖，这种耦合叫做恶性耦合，恶性耦合一般都是直接的，下文中说到的耦合均指恶性耦合。系统模块级设计的目的就是使各模块间尽可能地互相独立，尽量减少彼此之间的耦合，即使不可避免耦合，也要仔细设计模块间的接口，使之成为良性耦合。这样，各模块就能够独立开发、独立维护、独立生存，能够通过动态裁剪来适应新的需求，使企业能够制定灵活的产品战略，当经营环境发生变化时，能够降低转型的成本。中国有句古话，叫“船大掉头难”，如果把大船变成一批小船，掉头就容易多了，一个企业的产品规模和大了，如果各产品以及产品内部各模块之间互相耦合，修改任何一个产品或产品的一部分，都牵一发动全身的话，产品的转型和调整当然相当困难了。如果能够通过合理划分模块，并使各模块之间无耦合或者低耦合的话，就相当于一群小船掉头了。

技术路标设计的目的，是设计航道，让无数小船能够各行其道，避免发生挤占、摩擦、撞击等事故；设计指挥中心，用来协调指挥各船只停止、行进、转弯、掉头等动作；设计良性耦合接口，让各船只能够协同完成任务。

项目系统设计的目的，就是弄清楚完成项目需要多少小船，每条小船都用来干什么，为每条小船选择航道，选择小船之间的耦合通道，使之既能独立航行又能互相协作。

为支持低耦合和无恶性耦合的技术路标设计和系统设计，djyos 系统在以下几个方面做了优化：

1. 加载器设计方面，允许把一个应用程序分成若干模块独立加载，然后有机地融合在一起运行。
2. 事件类型 id 号由人工分配，而不是项事件 id 号一样由操作系统动态分配。
3. 泛设备驱动模块，泛设备接口使得系统工程师能够很方便地设计子系统以及子模块之间的接口。

有一个相当典型的关于耦合例子，著名的 Maxthon(傲游浏览器)的作者兼首席执行官陈明杰先生说，傲游选择 IE 为内核开发，最重要的原因还是基于 IE 做二次开发相对比较容易。当时也曾考虑过用 Firefox 的内核，但经研究后发现 Firefox Gecko 代码的逻辑结构很混乱，内部耦合性又很强，在上面做二次开发还是很难，才最终选择了 IE 内核。

耦合还是导致软件缺陷的温床，这种缺陷也表现在两个方面，一是相互耦合的模块本身是没有逻辑问题的，也就是说，作为单个模块是正确的。但是，单独运行正确的模块互相耦合也可能引发整体故障，这种故障不仅仅在模块间存在，在模块内部也广泛存在。这种耦合危害在于它作为一种全局 bug 而存在，查找这类 bug 通常是非常困难的，它需要对整个系统做全面的分析。要消除这种 bug 需要在系统设计初期就考虑软件模块间的全局配合。

二是互相耦合的模块中，如果一个模块内部有 bug，那么很可能会引起被耦合的模块也不能正常运行。如果模块对其他模块异常产生的错误做了充分的保护，通常会降低这种耦合的危害，当然，这种保护通常会降低软件的执行效率，但是从全局的角度来将，还是值得的。

11.3.1 时间耦合

如果一个模块的执行时间长短会影响另一个模块的功能，就认为这两个模块存在时间耦合。当软件各模块所要求的总运算量超过 CPU 的处理能力时，时间耦合是绝对性的，任何一个模块，都会因为消耗了 CPU 时间而影响别的模块，需要升级硬件以提高 CPU 平台的运算能力，或改进数学算法降低运算需求才能解决。更普遍的时间耦合是，CPU 运算能力本身有富余，但由于设计者考虑不周导致的非绝对时间耦合，这种耦合可以通过改进软件结构来予以解除。例如在一个扫描打印两用机控制程序中，程序员在主循环中依次调用各模块，包括传送打印数据模块和读取扫描数据的模块。打印数据传送程序检查打印缓冲区数据，若发现缓冲区非空就把需要打印的数据写入打印缓冲区；扫描数据读取模块检查扫描缓冲区，若有图像数据读出并清空。打印缓冲区能存储连续打印 1 秒的数据，扫描缓冲区也可以存储连续扫描 1 秒图像数据，该程序员认为主循环运行一圈的时间肯定小于 1 秒，只要每次循环把打印缓冲区填满，把扫描缓冲区数据全部取走，就能保证打印连续进行，扫描的数据也不会丢失，程序运行一圈实测的执行时间小于 500mS，有 50％安全裕量，因此设计者认为该软件是正确的。这是典型的时间耦合程序，打印模块的正常运行依赖于其他所有模块的执行时间，系统的软硬件及外特性的升级以及修改会受到极大的制约，表现为：

1. 如果产品升级，或者需要发展更高性能的型号，更换了更高速度的打印头，速度提高到 2 倍以上，那么发送到打印缓冲区的数据就不能维持连续打印，打印头将只能断断续续地工作，新打印头的性能将得不到发挥，无论使用多快的打印头，打印速度都不能提高。如果更换更快的扫描头，后果将更加严重，没有及时读出的图像数据将丢失。
2. 程序中其他模块修改导致执行时间延长时，主循环执行时间也相应地延长，当主循环延长到超过 1 秒，即使保持打印头不变，也会出现类似①的情况。可以认为打印和扫描模块与其他所有模块发生了时间耦合。
3. 程序员虽然考虑了安全裕量，但准确测试主循环执行时间是非常困难的，程序中会有很多的条件判断和分支，不同的分支执行时间是不一致的，很难测到最长时间的执行路径，即使同一个函数，随输入参数的不同执行时间也布一致，因此 500mS 的执行时间的可信度值得怀疑。而且，软件任何模块的修改都需要重新测试主循环执行时间，必须严格保证主循环执行时间小于 1 秒，使软件的可维护性极差。
4. 如果程序在某些极罕见的特定条件下执行时间不满足要求，这种 bug 是很难定位的。因为故障状态将难于重现，由于所有代码本身都是正确的，做代码的白盒分析也难于找到故障点。
5. 硬件平台发生变化时，同样难于保证软件的正确运行，需要重新测试软件在新平台下的执行时间。

要解决时间耦合的问题，在软件系统设计阶段就要避免任意一个模块对其他模块或者模块内部的执行时间发生依赖关系。本例的改进方法很简单，比如我们可以在打印缓冲区即将变空时发送事件，打印模块被事件触发把打印缓冲区填满；扫描缓冲区快满时发送事件，图像处理模块被事件触发，取走缓冲区中的全部数据，就可以完全解除时间耦合。

11.3.2 CPU 运行速度耦合

嵌入式程序经常需要在平台间移植，特别是，随着电子工艺水平的提高，产品的硬件平台也会升级，而软件则希望能够适应新的平台。不同的平台上，CPU 执行速度可能有很大

的差别，即使相同的 CPU 相同主频，也会因总线以及内存配置不同而运行速度有很大的差别，如果软件因 CPU 速度改变而执行结果不同，就认为这个软件发生了 CPU 速度耦合。当然，如果目标平台运行速度太慢而不能满足要求的除外。

上一节讲到的时间耦合，实际上也与 CPU 速度相关，但时间上的特征更为显著，故单列之。CPU 速度耦合最常见于与严格的硬件时序相匹配的程序中，如果硬件要求微秒级的时序匹配，通常的做法是用 for 循环做指令延时。但指令延时恰好与 CPU 运行速度关系极大，本节介绍如何解除指令延时和 CPU 速度之间的耦合。

指令延时的 C 语句形式：

```
volatile uint32_t    delay_var;
for(delay_var =j; delay_var >0; delay_var --);
```

注意，volatile 必不可少，有些编译器在优化代码时，可能会认为该 for 循环是“垃圾”代码而予以清除，加上 volatile 指示后编译器无论如何都会老老实实地生成代码。那么，需要循环多少次才能正确得到需要的延时时间呢？djyos 提供了几个全局参数，可以帮助程序员精确定时，y_delay_10us 则提供 10uS 粒度的延时。

1. uint32_t u32g_ns_of_u32for;
   当 delay_var 是 32 位无符号整数时，每个循环消耗的纳秒数。
2. uint32_t u16g_ns_of_u32for;
   当 delay_var 是 16 位无符号整数时，每个循环消耗的纳秒数。
3. uint32_t u8g_ns_of_u32for;
   当 delay_var 是 8 位无符号整数时，每个循环消耗的纳秒数。
4. volatile uint32_t u32g_delay_10uS;
   用 32 位无符号数做循环变量，延时 10uS 需要的循环次数。

为什么上述参数不是用符号常量定义而用变量呢？因为延时是用 for 循环产生的指令执行所需时间决定的，CPU 主频发生变化时，for 循环执行时间随之改变；即使硬件不做任何改变，编译器版本不同、或者编译选项发生改变，都可能造成代码质量发生变化，也会改变 for 循环执行时间，需要反复调节循环次数才能准确定时。导致的后果是，你的软件与编译器设置耦合，而且每次改变工程设置都要重新测定各常量值定义并重新编译，使软件的版本很多很繁琐，任何小小的疏忽都可能导致错误。解除这种耦合的方法是，用变量定义这些参数，在操作系统初始化阶段，调用 y_set_delay 函数动态测量不同情况下 for 循环的执行时间并初始化上述变量。这样，不管是升级编译器，还是改变编译优化级别或者改变其他工程设置，都无需修改使用 for 循环延时的代码，也无需重新编译。y_delay_10us()函数就总是能准确地进行微秒级延时。即使如此，for 循环准确延时是有条件的，嵌入式系统的运行速度由 CPU 和内存速度共同决定，也受 cache 是否命中的影响，下列情况下，for 循环不能准确延时：

1. y_set_delay 函数的执行地址所在的内存速度和 for 循环延时的执行地址所在的内存速度不同。
2. 行 y_delay_us 函数和执行 for 循环时 cache 的使能状态不一致。
3. 调用 y_set_delay 函数时中断仍处于关闭状态，且不允许事件调度，可以准确测量 fou 循环速度，但如果执行 for 循环时线程被中断打断或高优先级线程抢占，将不能准确延时。

代码 11-1 是ARM7 内核的S3C44B0X上初始化上述参量的示例，这是一个移植关键函数，用户在移植操作系统时，必须按照目标硬件系统修改这个函数。在这个函数中，利用定时器产生精确时间来测量for循环的执行时间，该定时器的时间基准来自CPU主频，故CPU主频改变时，需要修改cn_mclk常量后，重新编译才行，所以，这段代码是主频耦合的。但执行这

段代码后，所有用指令延时的代码都与主频无关。如果目标系统上有不依赖于CPU主频的独立时间基准，使用这样的时间基准编写的函数，这段代码将可以在不同主频的CPU系统之间自由移动，无需重新编译，也就是说，解除了主频耦合。

代码 11-1 时基测量函数。

```
void y_set_delay(void)
{
    uint32_t counter,u32_fors=64000;
    uint16_t u16_fors=64000;
    uint8_t  u8_fors=250,i;
    uint32_t u32_u8 = 256;
    volatile uint32_t u32loops;
    volatile uint16_t u16loops;
    volatile uint8_t  u8loops;

    timer_set_clk_source(5,0);          //timer5 的时钟源=mclk/2，最高时钟。
    timer_set_precale(2,0);             //预分频数为 1(不预分频)
    timer_set_type(5,1);                //自动加载连续工作
    timer_set_counter(5,cn_mclk/1000);  //2000uS，减计数
    timer_reload(5);
    timer_start(5);
    do  //测量 32 位变量循环时间(nS)
    {
        u32_fors >>= 1;                         //k 减半
        pg_int_reg->INTPND &= ~0x100;           //清中断标志
        while((pg_int_reg->INTPND & 0x100)==0); //直到发生中断，重新计数
        pg_int_reg->INTPND &= ~0x100;           //清中断标志
        for(u32loops=u32_fors;u32loops>0;u32loops--); //循环 u32_fors 次
        counter = timer_read(5);                      //读取循环 k 次所需时间
   }while(pg_int_reg->INTPND & 0x100);  //如果中断已经发生，说明 k 次循环大于 2000
                                        //uS，k 减半，再次循环，直到中断不发生
    counter = cn_mclk/1000 -counter;
    //扩大 1000，可提高精度
    counter = 2000000000/(cn_mclk/1000) * counter;
    //调试时注意，本行除 1000 与上 1 行扩大 1000 会不会给优化掉
    u32g_ns_of_u32for = counter /u32_fors/1000;

    do  //测量 16 位变量循环时间(nS)
    {
        u16_fors >>= 1;                         //k 减半
        pg_int_reg->INTPND &= ~0x100;           //清中断标志
        while((pg_int_reg->INTPND & 0x100)==0); //直到发生中断，重新计数
        pg_int_reg->INTPND &= ~0x100;           //清中断标志
        for(u16loops=u16_fors;u16loops>0;u16loops--); //循环 u16_fors 次
        counter = timer_read(5);                      //读取循环 k 次所需时间
```

```
    }while(pg_int_reg->INTPND & 0x100);  //如果中断已经发生,说明 k 次循环大于 2000
                                         //uS，k 减半，再次循环，直到中断不发生
    counter = cn_mclk/1000 -counter;
    //扩大 1000，可提高精度
    counter = 2000000000/(cn_mclk/1000) * counter;
    //调试时注意，本行除 1000 与上 1 行扩大 1000 会不会给优化掉
    u32g_ns_of_u16for = counter /u16_fors/1000;

    do  //测量 8 位变量循环时间(nS)
    {
        u32_u8 >>= 1;                           //k 减半
        pg_int_reg->INTPND &= ~0x100;           //清中断标志
        while((pg_int_reg->INTPND & 0x100)==0); //直到发生中断，重新计数
        pg_int_reg->INTPND &= ~0x100;           //清中断标志
        for(i = (uint8_t)u32_u8; i > 0; i--)
            for(u8loops=u8_fors;u8loops>0;u8loops--); //循环 u8_fors 次
        counter = timer_read(5);                      //读取循环 k 次所需时间
    }while(pg_int_reg->INTPND & 0x100);  //如果中断已经发生,说明 k 次循环大于 2000
                                         //uS，k 减半，再次循环，直到中断不发生
    counter = cn_mclk/1000 -counter;
    //扩大 1000，可提高精度
    counter = 2000000000/(cn_mclk/1000) * counter;
    //调试时注意，本行除 1000 与上 1 行扩大 1000 会不会给优化掉
    u32g_ns_of_u8for = counter /(u8_fors+1) * u32_u8/1000;

    u32g_delay_10uS = 10000/u32g_ns_of_u32for;
}
```

此外，djyos 还提供了一个时间粒度很小的指令延时函数，该函数的延时精度是 10uS，就是利用以上变量完成的。函数原型是：

void y_delay_10us(volatile uint16_t time);

延时的时间量是 time*10uS。本函数的延时精度与目标 CPU 的速度密切相关，CPU 速度越快，精度越高，如果目标 CPU 比较慢，指令时间达到 100nS 级，延时精度将大大降低。在 S3C44BX 上，主频 64Mhz，打开 cache 的情况下，延时误差<2% 。

注意，使用指令延时，CPU 完全在空转，效率是很低的，只适用于时间很短的延时，长时间延时应该使用闹钟同步,这也是time的类型限定为uint16_t的原因,最多只提供655mS延时。另外，如果指令延时执行过程中被高优先级事件抢占，或者被中断，延时时间将大大延长。

11.3.3 临界资源耦合

临界资源是指只允许有限重并发访问的资源，典型的是串行通信设备。当系统中一个进程正在使用串口进行通信时，其他需要使用串口的进程要么放弃通信要求，要么等待，就算该进程有较高的优先级也不例外。可能产生同时操作临界设备的模块，称做临界设备耦合模块。如果该设备性能不足，比如串行通信速度只有 10k 字节/S，而使用串口通信的所有模

块的通信需求之和为 20k 字节/S，则只能升级硬件才能解决。在设备性能足够的情况下，我们可以改进软件来降低或者消除这种耦合，使用合适大小的缓冲区就是一种可行的方法。假设：

1. 模块 A 每 100mS 最多发送 200 字节数据，其中有 64 字节需连续发送。
2. 模块 B 每 120mS 最多发送 300 字节数据，其中有 96 字节需连续发送。
3. 通信设备的吞吐量为每 100mS 发送 1K 字节数据。

很明显，模块 A 和模块 B 的的总通信容量在设备吞吐量之内，设备可以满足要求。如果模块 A 正在发送，模块 B 就必需等待，模块 A 最长连续占用设备的时间就是模块 B 的最长等待时间，反之亦然。所以，模块 B 的最长等待时间是 100mS×64/1000＝6.4mS，模块 A 的最长等待时间是 100mS×96/1000＝9.6mS。如果串口设备有足够的缓冲区，用户需要使用串口时，只要把待发送的数据写入缓冲区即可，无需等待，在本例中，缓冲区长度只要大于（96＋64）＝160 字节，模块 A 和模块 B 就无需等待，也就是说，两个模块通过串口设备产生的耦合被解除了。推而广之，如果有 n 个模块使用串口设备，各设备需连续发送的数据包长度是 Ln，要完全解除 n 个模块间的耦合，缓冲区长度要求为：

Buffer length＝（L1+L2+L3+……Ln）字节。

缓冲区的设置有硬件和软件两种方法，需要从软硬件联合设计的角度进行选择。硬件方法就是加大串口设备的内部缓冲区，通常要付出硬件设计和材料成本；软件方法是在内存中开辟缓冲区，这样会增加软件设计难度和 CPU 的负担。从软件的可移植性考虑，无论哪一种方法，在编写接口程序时都需要提供给应用程序一致的接口，使应用程序无需考虑是软件实现还是硬件实现。

操作系统提供了多种手段来保护临界设备的正确使用，如果不解除模块间的耦合，我们就必须依赖操作系统的保护，但我们并不提倡这样做。首先，操作系统保护功能一般会导致不期望的任务切换、优先级反转等，这些操作要消耗大量的 CPU 时间，降低了软件的执行效率；其次，这可能会导致高优先级的任务等待设备，会降低软件的实时性，对于嵌入式系统是致命的；第三，操作系统也可能会有 bug，操作系统功能执行的次数越多，bug 表现的机会也就越多，优秀的软件应尽量减少操作系统介入。

11.3.4 内存耦合

内存是临界设备的一种，但又区别于一般的临界设备，它是程序得以运行的核心资源。内存有静态分配和动态分配两种，静态分配的内存在软件编译时就已经分配好了，属于私有资源而不是临界资源。内存耦合是由于动态内存分配引发的，一个不使用动态内存分配的程序，就不存在内存耦合的可能。如果一个模块申请内存而不可得，就认为占用内存的模块与这个模块之间产生了耦合。要降低内存耦合程度，需要注意以下几点：

1. 尽量避免使用动态内存分配，任何在编译时可以确定的内存，都要采用静态分配。
2. 坚持用时分配，使用完毕后立即释放的原则。
3. 尽量不要频繁分配小块内存，如果需要大量小块内存，可以一次分配大块内存，在模块内部再划分为小块内存使用的原则，具体方法参见 第 2 章。

栈也是一种特殊的动态内存分配方式（参见第 2.4 节），同样会因此产生内存耦合。当一个线程调用操作系统的系统调用，系统调用会使用当前线程的栈（部分或全部），如果调用第三方开发的中间件的函数库，这些函数同样会使用当前线程的栈。如果你不为这些调用准备足够的栈空间，就可能因此造成栈溢出，产生不可预知的故障。djyos系统在实现时，无论系统调用还是库函数，均使用用户线程的栈，djyos确保每个函数（或系统调用）使用的栈不超过 8K，且在创建线程时，就直接在用户声明的栈尺寸的基础上增加了 8K，因此，

不会因调用操作系统服务产生有害的耦合。但是在调用第三方组件时，就应该格外小心，查清楚每个函数的栈需求，创建线程时为它保留足够的栈空间。

11.3.5 全局变量耦合

全局变量是多模块软件设计中一个极其危险的份子，是不折不扣的软件恐怖分子，它对所有模块都是可见的，任何模块都可以悄无声息地修改它，而使用这个全局变量的其他模块却一无所知。任何使用公共全局变量的模块之间，都通过这个全局变量构成了耦合，过多的全局变量互相耦合会使程序变成一张复杂的网，使程序员困在网中难于自拔。在通用计算机中，这种耦合关系往往会成为恶意入侵者的武器。在嵌入式环境中，一般没有恶意入侵者，但是软件 bug 不可避免，我们无法预知包含 bug 的模块会怎样破坏全局变量，会使软件的行为变得不可捉摸。

1. 当一个模块打算使用全局变量时，你必须了解使用了这个全局变量的其他模块的行为，以确保自己不会干扰其他模块以及自己不被其他模块干扰。
2. 当你调试软件时需要跟踪一个与全局变量相关的问题时，你可能需要会同使用该全局变量的其他模块的开发小组共同进行。
3. 当因某些原因需要修改一个模块而且涉及到全局变量时，可能要同时修改与本模块共用全局变量的其他模块，更要命的是，此时项目组可能已经解散，你难于召集所有人员。如果两个（或多个）模块分属不同的工程师编写，找不到相关模块的开发人员将是致命的，用户的期望可能无法实现。
4. 在开发和维护过程中，如果你发现一个 bug 与某个全局变量有关，可能需要在所有使用该变量的模块中查找 bug。不幸的是，很多 bug 隐藏很深，即使 bug 跟全局变量无关，但由于全局变量可能在被其他模块操作，具有“不透明性”，因此总是首先被怀疑，使查找 bug 的范围扩大到多个模块。

可见，过多地使用全局变量将严重威胁软件质量，它破坏了软件模块之间的独立性。下面举一个典型的全局变量耦合的例子。在某项目中，现场控制模块完成温度、湿度测量和控制功能，其中温度控制函数是 temperature_up 和 temperature_down 和 temperature_setup，这三个函数的名称已经很明白地表明了他们的功能，就不啰嗦了。数据采集管理模块则需要保存包括现场温度、湿度在内的许多参数，于是，数据采集管理模块定义了一个结构：

```
struct run_parameter
{
    int temperature;
    int humidity;
    .....;          //结构的其他数据域
};
```

并声明了一个全局变量：

```
struct run_parameter tag_run_parameter;
```

现场控制模块的 temperature_up 函数如下：

```
//参数: scalar，温度升高的度数
//          run_para，保存现场环境的结构指针
void temperature_up(int scalar,struct run_parameter *run_para)
{
    int fact_temperature;     //实际温度
    ......;
```

```
    run_para.temperature = fact_temperature;
    return true;
}
```

temperature_down 函数和 temperature_setup 函数与此类似，就不再重复了。

数据采集管理模块使用以下方法调用温度控制函数：

```
    temperature_up(5,&tag_run_parameter);
    save_run_para( );   //保存现场环境到文件。
```

系统上电初始化时，数据采集管理模块使用下列代码恢复现场环境：

```
    tag_run_parameter = read_run_para( ); //从文件中读出现场环境
    temerature_setup(tag_run_parameter.temperature);
```

在这里，我要说，这个项目是存在严重的全局变量耦合问题的，

下列方法可以在一定程度上解除全局变量耦合：

1. 限制使用全局变量，最好不使用全局变量。全局变量往往作为系统的状态量或者在模块之间充当信使，如果在操作系统支持下，这些功能完全可以用操作系统提供的消息或者邮箱等机制完成；如果没有操作系统，也可以使用环形缓冲区来实现模块间交换信息，部分取代全局变量。
2. 对全局变量严格管理，系统中可能由多个模块公用的的全局变量宜保存在 globe_var.c 中，在其他文件中不定义全局变量而用静态变量代替。如果是由多人或小组开发的项目，globe_var.c 宜由专人维护，由系统设计师统一控制，不允许项目组其他人员擅自定义全局变量。
3. 使用全局变量时宜做一定的保护，判断变量值是否在合理范围，若在多任务环境，访问时还应该用信号量进行适当的保护。有些操作系统支持“原子变量”，把全局变量定义为原子变量是一种很好的保护手段。
4. 如果全局变量在同一个模块内部使用，可以使用静态（static）变量实现。

11.3.6 功能性耦合

功能性耦合通常是模块功能定义不合理造成的，它使本来应该由某一个模块实现的功能，却由另一个模块实现，导致模块功能不完整。在一个设计合理的嵌入式系统中，每一个模块都应该能够独立完成赋予自己的任务，它在完成自己的任务的过程中，也许需要其他模块的帮助，但是，不应该依赖于其他模块的实现方式。举一个键盘驱动的例子，这是一个非常典型的、初级程序员容易犯的、且在嵌入式系统设计中广泛存在的错误。在一个典型的嵌入式系统中，界面由液晶显示器和键盘组成，需要实现一个编辑数据的功能，当用户按一下“↑”键时，选中的数字加 1，按住“↑”键不放时，选中的数字按每 200 豪秒加 1，“↓”键则反之。这里涉及到两个模块，一个是数据处理模块，该模块提供被编辑的数据和接收编辑结果；另一个是键盘驱动模块，该模块负责管理键盘。在这个系统中，数据处理模块被涉及成每次收到“↑”键就把数字加 1，收到“↓”键就减 1；键盘驱动模块则被涉及成：用户每按一次“↓”键便发出一个“↓”键，连续按住则每 200 毫秒发出 1 个“↓”键，“↑”键也是这样。这是典型的键盘模块与编辑模块间的功能性耦合，会引起如下问题：

1. 当用户需求发生变化，编辑功能被修改，持续按住按键时变量的增量（减量）速度由每次/200 毫秒改为每次/300 毫秒时，将要求修改键盘驱动程序。真有意思，数据处理模块的需求被修改，但实际修改的却是键盘驱动程序，这算哪门子事，键盘驱动程序的作者也够冤的。如果你是产品经理，就不会觉得有意思了，项目完成后，数据处理模块的作者已经被调到另一个部门了，当你厚着脸皮把他请回来后，才发

现真正要修改的是键盘驱动程序，只好再卖一把老脸了。

2. 这个程序的可移植性非常差，尤其是键盘驱动，当软件被移植到另一个产品上，该产品硬件与原产品相同，但是，软件却有多个地方需要使用类似按住按键自动连续增减的功能，且每个地方连续输入的速率不一样，驱动程序的作者真是欲哭无泪啊。

造成这种后果的原因很简单，连续按键时按照设定的频率给变量增量（减量）的工作本因该由数据处理模块完成，却由键盘驱动程序越俎代庖了，造成数据处理模块功能不完整，且严重依赖键盘驱动模块。而键盘驱动模块则相反，该模块的任务本应该是按键输入，却无缘无故地给加了一条“尾巴”——按数据处理模块要求的频率发出连续按键。判断连续按键应该是上层程序的功能，读者也许要问，不按照应用程序要求的频率连续发出按键，但在检测到连续按键时，就向应用程序发一次“连续按键”事件可以吗？这也没有破坏键盘驱动的完整性吧！这样也是有问题的，连续按住多长时间算连续按键，不同的应用程序有不同的标准，如果要做一个适应性强的键盘驱动程序，最好还是把判断连续按键的标准还给应用程序。改正上述错误的方法其实很简单，只要键盘驱动不判断连续按键就可以了，当用户按下按键时，便发出“按下按键”事件，当用户松开按键时，就发出“按键松开”事件，是否连续按键则由编辑模块自己判断，要改变增量（减量）的速度时，只修改数据处理模块即可，键盘驱动不需要修改。这样的键盘驱动程序可以不加任何修改移植到任何产品上都不成问题。事实上，这就是 PC 键盘采用的方案。

再举一个更加隐蔽的功能性耦合的例子，有一个数据采集系统，它主要由 3 个模块组成，分别是模拟量采集模块、数据运算处理模块、简单界面模块。数据运算处理模块需要从界面输入一个 16 位整型系数，用于计算 AD 转换的到的数据。于是这个系数的产生过程为：键盘输入数字字符—>界面模块把字符串转换成 16 位整数—>用消息把系数传给数据运算模块。这个过程看起来没有什么问题，但仔细想一想，它存在如下问题：

1. 数据运算处理模块需要系数是 16 位整数而不是浮点数，是模块内部的实现细节，项目设计要求界面模块把字符串转换成整数，实际上暴露了该模块的内部细节。
2. 如果数据运算处理模块被修改，16 位整型系数不能满足要求，需要使用浮点数或者 32 位整数，将导致界面模块随之修改，调整数据运算处理模块却连累界面模块也要修改，这是非常不合适的。

有两种方法可以消除这种耦合，分别是：

1. 界面模块把键盘输入的字符串直接传给数据运算模块，由数据运算模块把输入的数据转换成整数或者浮点数，界面模块只是显示输入数据和回显计算结果，这样，无论界面模块需要什么样的数据，都跟界面模块无关，所有的修改都局限在数据运算处理模块内部。
2. 界面模块提供充分的输入框组件，比如字符串输入组件、整数输入组件、浮点数输入组件等。当数据运算模块需要的系数类型改变时，只需要修改数据运算模块，重选组件即可，界面模块本身不需要修改。

11.3.7 解释性耦合

这种耦合通常发生在不同的计算机之间交换数据的过程中。（举一个在文件中存储整数类型数据，然后移植到大端系统中的例子）

11.3.8 编译器耦合

11.4 不要图一时之快

可以说，程序员偷懒、图一时之快是造成软件模块间严重耦合，使软件升级、修改、移植异常困难的主要原因之一，这种例子不胜枚举。许多程序员会为了少写两行代码、为了函数少传递一两个参数、为了提高一点点执行效率，而不顾模块的独立性。

11.5 组件的接口与实现

组件划分以后，自然就到实现组件了。只有降低组件之间的耦合度，提高组件的移植性和独立性，使之易于维护，才是组件化开发的精髓。因此，在实现组件时，应该遵循“输入接口－－实现策略－－输出接口”结构，如 图 11-1 所示。

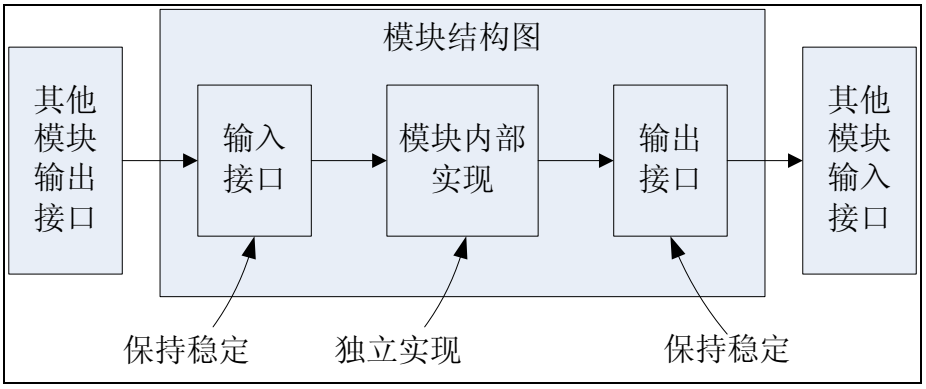


图 11-1 模块实现典型结构

组件内部实现部分代码要独立，与组件之外的代码无关，它只依赖于本组件事先定义好的输入和输出接口，不管是明确的还是隐含的，都不依赖与系统中其他组件。“输入接口”和“输出接口”包括明确的接口和隐含的相关性，明确的接口就是规定模块与别的模块交换数据的格式和方法，请求别的模块处理数据的方法，以及如何接受别的模块的请求等，只要稍加注意不难实现。隐含的相关性是指组件之间表面上看不到的，但实际上互相耦合的地方，经验不足的工程师很容易上当，这些错误经常来自想当然式的编程，或者称懒汉编程。设计一个组件时想当然地认为别的组件会如何如何，而这个“想当然”实际上只是一种特殊情况。例如一个通信接收程序，根据通信协议，最长的数据包只有 20 字节，从系统的角度讲，本机的通信模块是一个组件，通信的对侧机就是另外一个组件，本机组件的程序员想当然地认为对侧机组件不会发送超过 20 字节的数据包，于是就只开了一个 20 字节的缓冲区保存数据，且不作数据包长度检查。一旦对方出错，或者通信线路受到干扰而出现误码，很可能会出现超过 20 字节的数据包，就会铸成一个经典错误“数组下标越界”，后悔都来不及！如果说这是一个容错性的问题，只要稍有防御性编程经验的工程师都不会犯这样的错误。那我们再来看一个更加隐蔽的例子，在一个包含 ADC 转换组件、数据计算组件、界面组件三个组件的系统中，数据计算需要通过界面组件获取一个整型系数，用来和 ADC 模块的数据一起计算。于是界面模块实现了一个对话框，用户输入数据后，界面组件把用户输入的数据转换成整数传给计算模块。大家看看这个过程有没有问题？这个设计暴露了数据计算模块需要整型系数的细节，而且这个细节依赖于界面模块，它要求界面模块必须传递整型系数给自己。一旦数

据计算模块被修改，需要使用浮点型系数，将导致界面模块连带修改。改进的方法其实很简单，只要界面组件在弹出对话框之前，

为了适应标准化的输入和输出接口，是要付出代价的，要么软件执行效率降低，要么编程变得复杂，要么代码或者内存需求增加。有些程序员为了简单，或者为了提高效率，会在编程过程中不断修改接口，等软件完成了，原先定义的接口也就面目全非了，是得不偿失的。当然，编程过程中也不是绝对禁止修改接口，软件工程是如此复杂，开发初期是不可能完整且正确地定义出全部接口的，但修改接口是有原则的：

- 早期开发调试过程中，同时开发的同一个项目的各组件之间的接口，可以在开发过程中根据编码需要修改。
- 版本发布后，在软硬件维护升级过程中不能修改接口，除非能确定被修改部分全部局限在本次升级过程中，且不会与已经发行的文档资料相冲突。
- 如果开发中需要继承其他产品的组件，在移植过程中不允许修改组件接口。

11.6 组件化的好处

11.6.1 解放专业设计者

本书写到现在，还没有对嵌入式产品设计中的特殊群体——行业专家——予以特别关注。单纯的嵌入式系统并没有多少应用价值，它必须嵌入到实际产品中，比如数码摄像机、心电分析仪、继电保护设备等，嵌入式系统的硬件与各行业专用硬件结合，嵌入式系统中运行各行业专业软件，嵌入式系统才有意义。

然“闻道有先后，术业有专攻”，行业专家未必是计算机专家，而计算机专家也未必能看明白行业专家的理论。这种情况下，理想的开发方式是行业专家只用文字把算法详尽地描述，由计算机专家将其实现，但现实情况是，在大多数企业或者研发机构中，分工并没有这么细致和明确，计算机专家往往只搭一个台子，行业专家既要负责算法又要负责软件实现。行业专家由于缺乏专业软件训练，往往只能用程序设计语言写出由一个个函数组成的计算模块，对软件结构方面的了解比较少，他们能保证算法的正确性，却不知道如何使自己写的计算模块与系统中的其他模块协作运行。计算机软件工程师就应该设计程序架构（搭台子），划分产品构成组件，并规定组件间如何接口，以及通过该接口需要交换的数据和指令。使行业工程师能够在划定的框架内，充分发挥自己的专业才能。台子的坚固性、组件划分的合理性、组件接口的正确性将直接左右产品的各方面的性能。

所以，划分模块就是把整个嵌入式系统的功能划分为若干个相对独立的、针对具体问题的模块，这些模块可以独立实现，模块间没有或很少关联，尽量降低模块间交换的数据量和指令量，降解模块间协作的复杂度，使各模块可以由各自的行业专业小组独立攻关，便于分割包围、各个击破。

11.6.2 升级维护，得心应手

很多程序员都会有这样的经验，如果在软件设计初期没有规划好系统结构，很容易造成模块与模块之间互相关联，有些关联还可能相当复杂，数据与数据绞在一起，模块与模块功能交错。当你修改一个模块时，可能会引起多个模块的连锁修改，这些连锁修改又可能引发

更多的连锁修改，当软件规模很小的时候，整个软件你一览无余，可以随心所欲地控制，当软件规模越来越大的时候，就会觉得到处机关重重，犹如迷宫一般，让人不知所措，无所适从。要是想向这个软件增加一些功能的话，简直是个噩耗！嗯，够吓人的，但是，我告诉你，这只是个开始，这仅仅是软件方面的问题，嵌入式系统设计中，通常是软硬件一体化的设计，再考虑到软硬件互相配合的问题，问题将更加复杂化。如果软件和硬件模块没有从系统的角度规划好，软硬件可能会互相制约，谁也动不了。没有进行认真规划的系统，系统的复杂度与系统的软硬件规模呈指数增长，各个功能模块的复杂度也随系统规模增大而显著增大。这样的系统，无论时优化、维护还是移植都会困难重重。当你想优化升级硬件来提高性能时，可能会遇到来自于软件的巨大阻力；相反，如果你想升级软件以增加功能，又会遇到来自硬件的阻力。

笔者亲眼见过的一个产品，使用 1 片 16C554 扩展串口，16C554 的四个串口分别属于 4 个软件模块，而且这 4 个模块由不同的人编写。由于在系统设计时没有认真地从软硬件综合进行设计，没有评估 16C554 是否满足软件的要求，也没有很好地规划软件模块。结果 4 个人各自独立编写软件，这 4 个串口没有 1 行代码时共用的，就连初始化函数也是完全独立的 4 个函数。软件编写完成后，系统联调过程中发现，由于 16C554 的fifo只有 16bytes，A模块因硬件fifo深度不够而经常出现丢数据的现象，于是提出增加使用具有 64bytes fifo的 16C754 代替。等硬件修改完以后，调试模块B时，又出事了，原来B模块的Baud是 9600，它把 16C554 的接收fifo设为 4bytes触发，即fifo中有 4bytes数据时触发中断。串口接收 4bytes的时间最短为 4mS，模块把 4mS作为一个时标，许多功能依赖于这个时标。16C754 的fifo没有 4bytes 触发这个选项，于是，模块B需要做大规模的修改，而且该模块已经完成的大量测试全部重新测一遍。模块C和D还好，只需要修改各自的硬件初始化函数即可。注：模块B内部存在典型的时间耦合问题，本书第 11.3.1 节对时间耦合有详细的描述。

而一个结构规划得很好的软件，从系统的任意一点看过去，呈现在你眼前的是一个清晰的层次结构，所有模块的功能和数据都是内聚的，模块之间用标准的方法互相连接成一个有机的整体。任何一个模块的复杂度将只与本模块内部实现有关，当系统规模增大时，模块的复杂度不会增大，系统的复杂度也不会显著增大，一个模块甚至可以从已经存在的模块中获得帮助，从而简化自身的设计难度。系统规模增大时，对单个模块来说，看见的只是资源增加，设计时越是得心应手，左右逢源，经常会有一些意外的收获，让本来枯燥乏味的软件编程工作变成有趣的淘宝行动，这种感觉就象在沙漠里艰难前行中意外地发现一壶甘泉一样。Linux2.6 版本设计了设备模型，这本来是为方便电源管理而设计的，得益于 linux 的良好的软件结构，却意外地收获了 sysfs 文件系统，是一个典型的范例。这样的产品，即使在产品发布以后，增加、删除或者修改一个模块，都是轻而易举的事。操作系统的进程和线程管理对积木化提供了强有力的支持，但只依赖操作系统还远远不够，在以后的章节中，会陆续讲到这些问题。

良好的模块划分就需要为模块之间交互数据和操作设计接口，这些接口是标准化和规范化了的，为这些接口需要付出额外的工作，模块为了配合这些接口也会有所限制。虽然这些措施是保证项目质量和进度的重要手段，但这会给工程师带来额外的工作，延缓工程师们首次享受成功喜悦的时间，工程师们可能会因此感到厌烦，因为他们崇尚自由，喜欢很快进入工作状态，马上看到屏幕上有所反应，他们为此感到欢欣鼓舞。而没有进行模块规划的项目，在设计初期会很自由，程序员可以随意添加全局变量、标志位，可以随意改变数据在内存和文件中的保存格式，硬件工程师也可以随便地以自己最熟悉、最方便的方式设计硬件。工程师们会很乐意用这种方式开发，他们享受着自由自在的乐趣，他们享受着在项目初期快速进展的喜悦，他们快乐地结起了一个一个的圈套，最后深陷其中不能自拔。如果是一个比较大的项目，等他们感觉到不自由的时候，为时已晚，市场的压力已经不允许他们回头了；如果

项目不是太大，一直到项目结束，一直到额手庆功，他们都不会遇到太多的麻烦。且慢高兴！如果你的产品还需要维护，需要升级，需要面对用户新的需求，需要细分市场版本，需要为特定用户定制版本，你的恶梦就要开始了！即使一次小小的修改，你都要检查许多本可以不相关的代码甚至是全部的代码，你都要对整个软件做一次全面的测试，你会发现，你的软件做一次兼容升级是多么困难的事，非兼容的升级使你必须维护越来越多的版本。如果你是老板，你会发现你的员工总是被一些老产品拖住，即使这些产品已经不再产生利润，不能抽身出来做新项目的开发。

11.6.3 系统规划，性能优越

总之，如果系统设计时有科学合理得模块划分，即使模块内部设计得很蹩脚，但仍然有可能组成一个优秀的产品；如果模块划分得不合理，即使每一个模块内部设计都非常优秀，却有可能不能构造出合格得产品来。前苏联武器装备的系统集成能力是及其出色的，七十年代前苏联的米格 25 是当时世界上最先进得战斗机之一，别连科上尉驾驶一架米格 25 叛逃日本，美国大喜过望地进行了拆解，结果发现这架当时世界最先进的战斗机，其内部各子系统的工艺水准都很差，但组合起来起来却性能颇优。80 年代时，日本人嘲笑中国长征火箭工艺落后，粗制滥造，的确，日本的工业加工水平很高，日本 H2A 火箭每一个零件的制作水平都比长征火箭高很多，性能指标也很高，但是直到今天，H2A 火箭的发生成功率都远远低于长征火箭。〔参考：中国青年报 2005 年 2 月 28 日〕。

嵌入式系统也一样，每一个模块都实现单项最优是不切实际的，但通过系统设计，实现整体最优却是可能的，这与软硬件的系统架构有关，合理的模块划分是关键！追求整体最优能缩短开发周期，节省开发成本，还能降低产品的生产成本。

11.7 划分模块的原则

项目开始的时候，系统设计员可能面对的是一堆杂乱无章的功能、需求，几乎没有章法可循，需要使用综合、抽象等手段，对各种需求进行归纳、拆分和合并，使项目成为一系列模块的有序结合。如何划分模块，往往体现一种软件设计策略，这种策略运用，是设计者智慧之光的关键所在，它从一开始就决定了软件的质量。模块划分是否合理，不但是一个项目能否成功的关键，还决定了项目对企业研发能力的持续增长提供多少增益，模块划分得好，可以有效地降解项目的复杂度，可以更加准确地估计研发工作量，提高项目研发的可计划性，促进项目顺利完成，提高产品生命周期的维护工作量，并且在项目开发完成后还可以为企业留下大量可重用的组件，这些已经测试和验证的组件是企业研发能力持续提高的源泉。划分模块属于系统设计阶段的任务，此时，一切都还处于混沌状态，没有参考，也没有任何约束。然而，越是处于自由状态，就越是无从下手，模块划分的方法就没有什么规律可寻吗？不是的，下面我们探讨一下划分模块需要遵循的基本原则。再强调一次，这些原则可以用于软件开发但不仅仅用于软件开发，在系统设计阶段，软硬件是一体的，这个阶段划分出来的模块并不区分是用软件实现还是硬件实现。遵循这些原则，我们就可以做到：

1. 高可维护性，产品开发完成后，就进入一直延续到产品生命周期结束的维护周期，维护工作主要是故障维修，有些产品可能还需要根据用户需求微调产品的功能。模块划分清晰合理的项目，有助于迅速定位故障位置，有助于缩小故障范围，有助于使用简单的措施消灭故障。
2. 高可移植性，嵌入式产品是工程性很强的产品，尤其是在工业控制方面，用户的需

求五花八门，产品需要很多系列化的型号去面对细分的用户需求，每个型号的功能大同小异，最坏的情况是付运的每一件产品都需要根据工程需求做或多或少的调整。模块的合理划分，使产品可以用构建重组的方式实现产品功能的重新定义，用简单的工程配置方式使产品适合特定工程，还有助于减少产品型号数量，方便生产和库存管理。

3. 加快研发进度，合理划分模块使各模块的内部功能和外部接口清晰而且明确，使各模块可以独立开发，开发进度不受或少受其他模块进度的影响，避免模块间互相掣肘的情形，使开发工作可以持续进行。明确的外部接口还使模块在一定程度上可以独立测试，减少因不同模块的开发人员之间沟通不畅造成的返工，因此能加快研发进度。
4. 提高可靠性，合理模块划分可以降解复杂度，如果模块划分不清晰，导致模块间互相关联，那么模块开发人员就要面对系统间的复杂度；反之，模块开发人员只面对本模块的复杂度。常识告诉我们，越复杂的东西，出错的概率越高。按可靠性分级划分模块的原则也使得采取专门措施来提高关键模块的可靠性变得简单。按实时性分级的原则还使得可以采取专门措施提高时间关键模块的实时性。所有这些都是提高产品可靠性的有效手段。

图 11-2 是一个完整模块应该包含的要素，与传统设计方法相比，图 11-2 增加了实时性和可靠性元素，也就是说，可以用相同方法实现但可靠性要求差别很大的功能块，放在一个模块中时如果低要求的功能块有可能影响高要求的功能块的实现，就不能放在一个模块中。实时性要求也是如此。

图 11-2 模块包含的要素

11.7.1 一致性原则

一致性原则指的是相同的功能用相同的方式实现，而且是用独立的单个模块中实现。单个模块的意思是实现该功能的代码只出现在一个地方，所有需要使用该功能的需求都由这个独立的模块提供服务。避免用不同的重复实现相同的功能，是优秀的系统设计的基本要求。而重复实现相同的功能，是嵌入式系统开发尤其是软件开发中常见的错误，原因是在系统设计时没有合理划分模块和对模块的功能做出明确的规定，模块设计人员就会按照自己的喜好随意设计模块；以及在详细设计时没有合理设计子模块，程序员在实现子模块时就无章可循，完全凭自己喜好行事；最后是程序员没有编写一致性代码的意识，在写代码时没有时时警觉：这个功能我在 xx 地方实现过了，不应该图一时方便信手重写。面向对象的设计中反而比面向过程的设计容易犯这种错误，如果你把 485 通信口和光纤串行通信口当作两种对象分配给不同的人来设计的话，他们往往会完全独立地实现所有功能，然而，由于他们的公共特性：串行字符流，他们必然会有许多相同的特性，比如可能都采用 CRC 校验。两个对象实现后，结果可能会两个程序员分别独立写两个 CRC 校验函数，独立实现接收和发送队列，软件测试时就需要独立测试两套功能几乎完全相同的代码，软件维护时也要独立维护两个版本。

笔者看到过这样一个产品，该产品使用一片 16C554 扩展串口通信，16C554 的四个串口分属 4 个模块，分别由 4 个工程师设计。大家知道，串口需要一个把数据从物理接口和应用层的缓冲区之间传送的功能，通常被叫做链路层。无论通信的上层协议怎么样，链路层的功能是一样的，应该当作独立的模块予以设计，这个独立的链路层模块为上层协议提供传输服务。但笔者看到的这个产品，系统工程师只描述了使用串口的 4 个模块的功能，没有系统性地划分模块，结果这 4 个工程师各自按照自己习惯的方法设计串口传输部分，4 个串口从初始化到中断收发函数再到数据处理过程完全不一样，增加了 3 倍的开发、测试和维护工作量。

而且这 4 个功能模块还被牢牢绑定在固定的串口上，在发展该产品系列化型号的过程中，这种绑定带来了极大的麻烦。有些新型号在硬件方面删除了串口 A，保留了串口 B；在软件上则保留了模块 A，删除了模块 B，因此，需要让模块 A 从使用串口 A 改为使用串口 B 进行通信，由于模块 A 直接操作串口 A 的寄存器，需要修改大量代码才能完成。由于这种问题频繁出现，最后才下定决心把链路层独立出来，4 个模块的串口接口部分也做了很大的修改。虽说亡羊补牢，未为晚也，但是已经浪费了大量的开发时间，而且在用户那里留下了许多早期版本的软件，这些已经卖出的产品都是需要维护的，而嵌入式系统的维护可不像 windows 那样，发个补丁给用户下载就可以的。

另一个例子是关于嵌入式编译器的，我在文件 B 中 inlcude 了文件 A，编译时，编译器输出了两条互相矛盾的出错信息，第一条出错信息说文件 A 不存在，第二条出错信息说文件 A 的某行语法错误。既然文件不存在，又怎么说文件里面有语法错误的？这不是自相矛盾吗？经过反复试验，发现问题与文件 A 的存放位置有关，文件 A 放在配置目录中，文件 B 放在工程目录中。猜想是编译器有两个独立的功能模块先后使用了文件 A，第一次打开文件 A 时搜索工程目录和配置文件指定的目录，所以能找到文件；第二次打开文件 A 时仅搜索工程目录，就找不到文件了。我把文件 A 拷贝到工程目录后，问题就解决了。这是显然一个 bug，出现这种自相矛盾的信息，用户是非常反感的，而且使用户无所适从。相反，如果两个模块都有 bug，直接报告找不到文件 A，用户可能还没有那么困惑，很容易就怀疑到文件 A 的存放位置问题，并加以解决。我们知道，用户的满意度就是衡量产品品质的好坏的标准，排一下用户满意度与产品中 bug 数量的关系，我们会发现一个有趣的现象：

最满意：两个模块都没有 bug，这没有什么好说的。

不满意：两个模块都有 bug，用户虽然不快，用户可能会很快想到是文件目录的问题，不至于无从下手而抓狂。

最不满意：只有一个模块有 bug，用户为自相矛盾的信息抓狂而无从下手。

这是非常有代表性的系统设计问题，用户的满意度没有随 bug 的减少而提高。这就不是程序员的问题了，也不是测试员的问题，程序员和测试员的目标都是忠实地实现系统设计的要求，尽量减少 bug 的数量，对于他们来说，少一个 bug 就多一份成就。但少一个 bug，相当于产品的局部改善了，却降低了产品的总体品质，这难道不是一个系统设计的问题吗？就像每一个零件的制造工艺都更加精良的 H2A 火箭，发射的时候却一个劲地往地上掉，只能眼睁睁地看着“粗制滥造”的长征火箭翱翔九天！

出现这样滑稽的结果，显然是模块划分时出了问题，它使“查找并读入文件”这样的功能在两个模块中重复实现，两个模块实现这个功能时使用了不同的方法，导致了一个模块有 bug，而另一个模块没有 bug 的现象。我们不能责怪模块模块设计者使用了两种不同的方法实现该功能，因为模块是独立的，作者只要保证模块外部接口的一致性，内部如何实现，完全是模块“隐私”，别人无权干涉。

相同的功能统一实现，不但能降低开发工作量，还可以提高产品质量，实在时一举两得的事情。但由于嵌入式系统的复杂性，很多功能往往出现交叉，要合理第划分软件模块使其不重复，是一件极具挑战的工作。

11.7.2 可维护性原则

对于嵌入式产品尤其是工业嵌入式产品，他们的维护周期很长，而且维护工作很复杂，有些产品根本就不允许停止运行。这种情况下，良好的模块划分尤其重要，划分模块时，应该充分估计产品中哪些部分是可能根据工程修改的，哪些部分是不需要修改的。把需要修改的部分独立成模块，使这些模块的修改、增加、删除均不影响产品中其他模块的运行。而把

不需要修改的部分做成公共模块，并且尽量使公共模块最大化。可维护性原则与前面讲的一致性原则相结合，可以实现如下目标：

1.

使用配置文件也是一个很好的方法，这样可以使工程服务人员直接用文本编辑工具修改产品的功能，使之适应工程需要。但是，使用配置文件只能满足已知的需求，当工程现场出现原先没有预料到的状况时，还是需要修改软件，科学的模块划分仍然是最重要的。

11.7.3 可移植性原则

实现高可移植性的模块，需要在系统设计的时尤其是模块划分的时候做出妥善的设计，既要使模块很容易被摘下来，不影响软件其他功能，又要很容易把一个新模块拼上去，立即增加新功能。一般来说，一个企业不会只有一个产品，而往往有若干个产品系列。每个系列可能有许多细分型号，每个型号的功能只有细微的差异，有些工程产品甚至付运的每一个产品都有所不同。不同系列的产品，在功能和组成上也有所联系，往往是同一行业的产品。而且，同一个企业的产品，往往有相同的软件和硬件开发平台。我们常常讲拿来主义，拿来的方式有很多种，可以购买商业化的软硬件模块，要么利用开源代码或者网络上的资源，但最重要的是从自己的产品中继承。多数嵌入式系统是个性化的系统，当我们开始工作并想去拿些前人积累的东西来用时，往往会找不到什么可拿的，你会发觉相似的东西不少，但完全符合需要的没有。最关键的是，从网络或者文献上找到的资料，多数是未经验证的，你需要花很多时间去通读它，然后移植并验证它，所花的时间并不一定就比自己重新开发少。因此，我们大多数时候只能拿自己积累的东西，由于企业内不同系列产品或者同系列不同型号产品的相似性，你可以从中拿到很多东西。为了后续开发产品时有东西可以拿，我们开发任何一款产品时都要非常重视“可拿去性”，这是一个企业可持续发展的动力所在。未了实现可拿去性，设计时，要从公司全线产品的角度考虑哪些功能是可能在不同产品之间互相转移的，也就是说要拿给其他产品使用的，这些模块要划分为独立的模块进行设计。哪些功能模块是将来产品升级或开发新的系列型号时可能发生变化的，把这些模块独立设计，也会使新型号开发时需要修改的代码最小化，绝大多数不需要修改的部分可以直接拿到新型号上使用，这同样是出于可拿去性的考虑。

一句话，不拿去，何以拿来？

11.7.4 功能独立实现原则

与面向对象软件设计中所说的对象封装概念相似，这里所要讲的模块范围比软件设计中的对象更广泛更抽象一些，它包含软件中的对象，还包含硬件模块以及软硬件协同模块在内。

数字编辑框的案例。

11.7.5 可靠性分级原则

一款完整的产品往往由许多功能模块组成，每个模块的可靠性需求是不一样的，根据现实生活正著名的 80-20 原则，产品中高可靠性需求的模块可能不会超过 20%，甚至只有极个别的模块需要很高的可靠性。例如在汽车集中控制系统中，刹车控制模块的可靠性要求可能

比其他模块的可靠性要求加起来还高。

系统的可靠性需求由可靠性需求最高的模块决定，比如一台机床控制系统中，包含防止机床过热损坏的功能，由于机床非常昂贵，要求过热保护功能非常可靠。如果过热保护功能与其他控制功能由同一个 CPU 执行，则 CPU 硬件系统的可靠性要求等同于过热保护功能；如果把过热保护功能从主 CPU 分离，改由一个功能单一的小规模 CPU 实现的话，则只需要保证小 CPU 的可靠性，而主 CPU 系统的可靠性需求则可以降低。

而一个模块的可靠性需求又由该模块中可靠性需求最高的功能部件决定。

不要把可靠性要求相差较大的几个功能放在同一个模块之中，如果一个模块包含几个功能，每个功能的可靠性要求并不一致，那么，整个模块的可靠性要求取决于要求最高的模块。我们知道，即使相同的功能，因为可靠性要求不同实现成本也是不一样的。把可靠性要求高的模块独立出来，有利于开发和测试时采取特殊手段单独强化这些模块，这些特殊手段的成本可能会很高，因此，缩小特殊手段的应用范围很重要。

即使有些看起来完全一样的功能，如果其可靠性要求相差很大，也要分别放在不同的模块中，例如：

11.7.6 实时性分级原则

11.8 djyos 为组件化开发提供的支持

设置事件类型名字，不光是便于调试。
弹出事件时类型是否登记的判断。
泛设备组件。
异步信号同步功能使得使用事件处理函数无需与 ISR 互动。
事件类型同步使得模块间无需互动。
事件是独立对象，事件处理函数之间无法直接交互。
模块独立加载器，独立模块等的定义。

第12章　文件系统

和其他许多操作系统一样，djyos提供文件系统支持，使应用程序能够按文件组织和访问系统中的存储器，文件系统由公共部分和存储介质驱动部分组成，公共部分简称djyfs（djyos file system）模块，存储介质驱动模块简称dfsmd（djyos file system medium driver）模块，dfsmd是统称，具体介质的驱动程序可以有自己的名字，比如flash文件系统驱动程序就叫DFFSD。无论使用何种物理存储介质，djyfs模块都是一致的，djyfs模块是依托操作系统的资源管理模块和泛设备驱动模块实现的，如果你希望深入了解djyfs，那么，阅读本章之前，请先阅读“第 8 章 资源管理”和“第 9 章 泛设备驱动”。无论系统中存在多少种存储介质，都使用同一份djyfs模块，同一份的含义是，它不是相同代码和数据的多份拷贝，而是只有一份拷贝。djyfs模块提供应用程序接口函数，管理文件系统设备和文件柜设备，通过这两个设备与dfsmd模块驱动的物理存储介质连接。这一级不面对物理存储介质，不对文件分配表、扇区、以及目录表等进行操作。曾经考虑在djyfs这一级直接支持广泛使用的分块存储策略，但最终还是放弃了这个想法，djyfs认为，存储器是否被划分扇区和块、扇区的尺寸、块的尺寸、是否需要擦除操作、文件在存储器中如何组织、目录表如何组织等，都是文件存储介质的特征，直接面向操作系统和应用程序的文件系统公共部分应该是与存储介质无关的，它不应该与具体存储介质扯上哪怕一丁点的关系。文件在存储介质中如何存入和读出，应该由dfsmd模块实现。另外，操作系统中djyfs模块与dfsmd模块是两个独立的模块，**djyos系统信条之一是，任何一个模块不应该对其他模块的内部实现机理枉自揣测，**所以，djyfs模块不应该对dfsmd模块的行为做任何假设，更不应该自作主张地认为存储器是按扇区或者块存储的，进而对dfsmd模块发出读写扇区之类的指令，如果该存储器压根就没有扇区，你这不是瞎指挥吗？djyfs完成以下几件工作：

1. 组织和管理文件柜（相当于 win/dos 系统的磁盘），在系统资源链表的设备树下面建立了一个文件系统结点，所有文件柜都建立在文件系统结点下面，组成一颗文件系统树。
2. 提供面向应用程序的、与存储介质无关的目录管理，在系统资源链表中维护了一棵由打开的文件和目录以及当前工作目录组成的目录树。
3. 把应用程序对文件的存取操作传递给相应的 dfsmd 模块，由 dfsmd 模块最终完成实际的读写操作。

12.1 djyfs 与 ANSI C 文件 IO 的差异

与 ANSI C89 一样，djyfs 支持流式文件访问，但并不支持 ANSI C89 标准要求的与文本文件相关的操作。除此之外，djyfs 与 ANSI C89 完全兼容。与 unix 文件系统一样，djyfs 只支持二进制数据存储，也就是说，**djyfs 把所有读写操作都看做二进制码流，不对其内容做任何假设和解析**。我们知道，windows 和 dos 都支持文本流的操作，并且得到 ANSI C 标准的支持，如果用户用“r”“w”“a”“r+”“w+”“a+”模式打开文件，文件系统将把用户对文件的读写数据流自动按文本文件格式解析，可以用文本文件的方式打开、读取和保存文件，必要时会自动插入和删除回车和换行符等文本文件特有的操作。如果用“rb”“wb”“ab”“rb+”“wb+”“ab+”这些模式打开文件，文件系统就不对读写数据做任何解析。在 djyfs 中，虽

然不支持文本文件操作，但是仍然可以识别 ANSI C 格式的所有模式字符串，“r”与“rb”等价，“w”与“wb”等价，……。

fp = djyfs_fopen(“myfile”, “r”);与

fp = djyfs_fopen(“myfile”, “rb”);将执行完全一致的操作。

上述差异可能会给一些习惯在windows下编程的用户带来些许不习惯，并且造成djyfs与ANSI C不完全兼容，djyfs的下一步开发计划会在二进制流的基础上增加一个外壳以符合ANSI C的要求。可能有些读者要问，如果直接在文件系统的底层直接实现文本流的操作，文件系统看起来将更加简洁高效，为什么要不胜其烦地用增加外壳的方式实现呢？笔者认为，软件模块应该保持独立性和完整性，避免模块间在功能上互相交叉，这样会导致模块间的功能性耦合（模块间耦合参见 11.1）。文件系统是一个有组织地存储数据的模块，它应该专守数据存储的职责，而不应该过问所保存的数据的含义。应用程序从文件中读取数据后，将按其需要对数据进行解析，按文本解析是其中一种可能。支持文本流实际上就要求文件系统要把存储在文件中的数据按照文本格式解析，而解析文件内容应该是文件系统使用者的事，这样做将造成文件系统模块和使用者程序模块间功能性的交叉。现实中，ANSI C支持文本文件操作是有明显的副作用的：

1. 造成了理解上的歧义，从使用者的角度去思考，不把 word 文档认为是文本文件是不适宜的，而计算机术语中的“文本文件”的却不包含 word 文档，以至于“什么是文本文件？”会成为一道计算机专业考试题。
2. 好像有点不公平，文本文件是一种文件格式，同样，还有许多文件类型如 pdf、doc、html 等同样也得到广泛应用，ANSI C 却只对文本文件提供编程语言级的支持，对其他文件格式却视如不见，奈何厚此薄彼？
3. 从软件维护和管理角度上，我们希望用相同的方法实现相同的功能，但是，为了支持文本文件却使这成为一种奢望。以 seek 函数为例，当用户执行 fseek(fp,100000, SEEK_CUR)函数，文本方式和非文本文件的执行方式是完全不同的。如果是非文本文件，则只要把文件指针在存储器中的偏移量移动 100000 就可以了，而文本文件却不行，因为文本文件的回车和换行是按照一个字符解析的，因此，seek 文本文件就需要从当前文件指针开始连续读取超过 100000 字节数据，合并连续的回车字符和换行字符为一个字符，才能确定文件指针在存储器中究竟要移动多少。同样是 fseek 操作，两种文件的实现方式完全不一样。
4. 本来，按文件方式存储数据有利于在不同的计算机系统之间交换和共享数据，但由于 ANSI C 文件系统对文本文件的内容进行解析，给这种数据共享造成了不必要的障碍，所有处理文本文件的程序，都要对 dos 文件系统和 unix 文件系统区别处理。

因此，为了维护软件模块的独立性，不造成模块间耦合，djyfs 的文件系统将只支持二进制流，在这之上，增加一个文本文件读写模块，完成对文本文件的解析工作，以迎合 ANSI C 标准。

12.2 约定规范

使用 djyfs 必须遵循以下几个规范：

1. 文件柜（相当于 win/dos 的磁盘）名的长度不得超过 255 字符，汉字按两个字符计算，具体的 dfsmd 模块所支持的文件柜名的长度可能小于 255 字符，比如 flash 存储介质驱动模块 DFFSD 的文件柜名就只支持 31 字符。
2. 文件名长度也不得超过 255 字符，与文件柜名的规定一样，比如 flash 存储介质驱动模块 DFFSD 的文件名长度只支持 215 字符。

3. 文件夹名、文件名和文件柜名均不能包含“*\?/:|<>“”这 9 个字符。
4. 字符‘:’是文件柜名分隔符，在一个完整的路径字符串中，‘:’前的部分被视为文件柜名。
5. ‘\’是目录分隔符，包含多级目录的字符串被‘\’分割为一个个独立的文件柜名、目录名或文件名。
6. 不含文件柜名的目录和文件串中，如果以‘\’开头则认为从当前文件柜的根目录为父目录，如果不是以‘\’开头则认为当前路径 pg_work_path 为父目录；如果字符串以‘\’结尾则认为最后一项是目录，否则认为是文件。
7. 目录嵌套最深不超过 20 级。
8. 单个文件的长度允许达到 2^{63} 字节，一次读或者写的长度不得超过 2^{32}-1 字节。

djyfs 文件系统限定了文件名长度和目录嵌套深度，作者并不愿意限制用户的自由，但作为与实时操作系统 djyos 配套的文件系统，却又不得不这样做。我们知道，文件和目录是按照名字操作的，访问文件首先要提供文件名和它的存放位置，文件系统通过目录路径逐级比较字符串定位文件的物理存储位置。如果文件名长度和目录嵌套深度不做限制，就有可能出现不可预估长度的文件名+路径名字符串，使得处理时间变得不确定，而**不确定的执行时间是实时系统的大忌**。另外，高可靠系统必须分析用户提供的名字串是否合法，其中就包含判断名字的长度是否合法，允许任意长的名字串长度使合法性判断无从下手。因此必须限定名字串的总长度，限定的方法有两种，第一种方法是每个文件或者目录的名字长度均可以达到 255 字节，但限制目录深度不能超过某一数值。第二种方法是限制包含路径名在内的整个名字串的总长度不得超过某一数值，单个文件或目录的名字长度以及允许的目录深度则不计较。djyfs 选择了前者，不选择后者的原因是：

1. 如果用户修改一个目录名，那么文件系统就必须扫描这个目录的上级目录和所有下级目录以及文件，其名字加起来不越界，这是一种执行时间开销极大而且不确定究竟需要多长时间的操作，实时系统要尽量避免这种不确定执行时间的操作。
2. **djyos 的信条之一，软件中的不同元素应该互相独立，尽量减少模块（元素）之间的耦合**。文件系统中的一个个的文件和目录可能分属不同的模块，应该作为独立元素，应该有独立的名字空间，限制名字的总长度将导致不同的目录和文件的名字长度互相牵制，任何一个目录或者文件名的允许长度，都与其上下级目录（文件）名字的长度相关，使其命名空间不独立。

dfsmd 模块所支持的文件名和文件柜名长度可能小于 djyfs 模块，那么，因两者不一致而发生冲突的时候如何处理呢？djyfs 规定如下：

1. dfsmd 模块支持的名字不得超过 djyfs 模块，即文件柜名和文件名（目录名）长度均不得超过 255 字符。
2. 如果名字的实际长度不超过 dfsmd 模块所支持的长度，直接使用原字符串。
3. 如果 djyfs 模块传递给 dfsmd 模块的名字长度超过 dfsmd 模块的限定值 n，则只有前 n 个字符有效，在同一命名空间内，前 n 个字符如果相同，则构成名字冲突的条件。比如在 flash 芯片内创建两个文件柜，如果用户提供的名字的前 31 个字符相同而第 32 字符以后不同，则 djyfs 模块认为是合法的字符串，而 dfsmd 模块则认为发生了字符串重名错误。

12.3 文件系统整体结构

承接上一章，djyfs依托泛设备驱动模块而构建。在操作系统的数据结构资源视图中，文件系统包括文件系统设备（设备名是“fs”），文件柜设备（设备名是文件柜名），文件资源

（以文件名为资源名），路径资源（以路径名为资源名），图 12-1 显示了文件系统各种元素在系统资源管理体系中的位置。当用户执行文件系统调用时，操作系统通过“文件系统设备—>文件柜设备—>各级路径资源—>文件资源”逐级找到被访问的文件。

文件系统设备只有一个，名为“fs”；文件柜设备有若干个，是“fs”设备的子设备，名字由 dfsmd 模块设定，文件资源由打开的文件和目录组成，文件系统的所有功能都是围绕这些设备和资源实现的，在这些设备和资源中，文件柜设备是核心中的核心。

文件系统设备虽然处于最顶层，但主要起统领作用，本身比较简单，就是一个 struct pan_device 结构，没有专门的数据结构。

文件柜设备的核心数据结构是 struct st_DBX_device_tag，文件柜设备的 struct pan_device 结构的 private_tag 成员作为指针指向这个结构，它详细描述了一个文件柜设备的信息，比如文件柜的尺寸、文件数量、以及一些列文件柜接口函数的指针，文件系统利用这个结构管理文件柜本身和访问这个文件柜下的文件和目录。

文件资源指的是被打开的文件或目录，而不是静静地躺在文件柜里面的所有文件和目录。djyfs 模块初始化的时候，在系统资源树上建立了一个名为“opened file”的结点，所有文件资源都是该结点的后代结点，他们在资源树中的组织关系一如其在存储介质中的形式。数据结构 struct file_rsc 是文件资源结点的数据类型。它详细描述了一个文件（目录）资源的信息，比如文件属性、文件尺寸、建立和访问日期等。

所有文件柜设备都是文件系统设备的子设备，不同的文件柜可能建立在相同类型的存储媒体上，也可能建立在不同类型的存储媒体上，一切取决于它的驱动程序，在文件系统这一级，并不涉及到具体的存储媒体。

djyfs按明显的层次分布，如 图 12-2 显示的整体组织结构和 图 12-3 显示的访问层次结构所示，最顶层当然是文件系统使用者。应用程序执行文件系统相关的系统调用函数请求文件系统服务，文件柜作为一个设备，其所有功能都是通过设备接口来实现的。文件柜的左手接口用于提供其上层的系统调用服务，而右手接口为dfsmd提供服务。值得说明的是，dfsmd模块并没有作为泛设备来组织，而是提供了一组读写函数供文件系统模块访问，文件柜访问存储介质就是通过这组函数来实现的。这样做并不会破坏软件结构，反而会使其更加紧凑灵活。首先，存储介质驱动模块要访问文件柜时，是通过文件柜设备的右手接口来实现的，符合设备不被直接访问的原则。其次，存储介质驱动模块本来就是专门为文件系统模块服务的，没有其他模块可能访问它，文件系统严格按照存储介质驱动模块提供的函数指针访问它，并不会破坏其模块独立性。

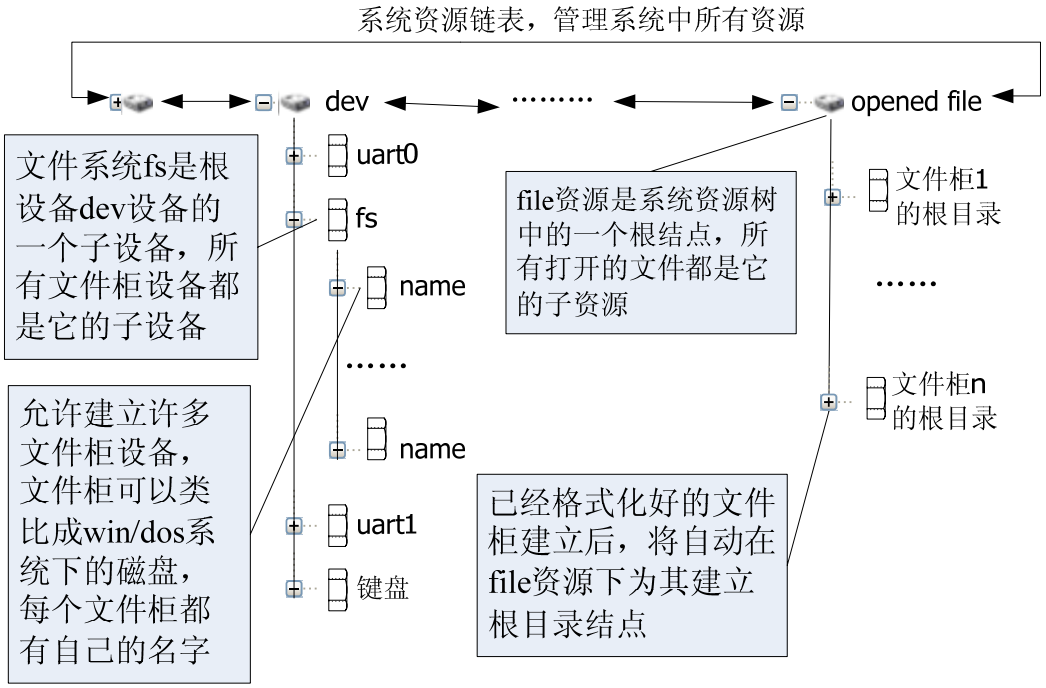


图 12-1 文件系统的资源视图

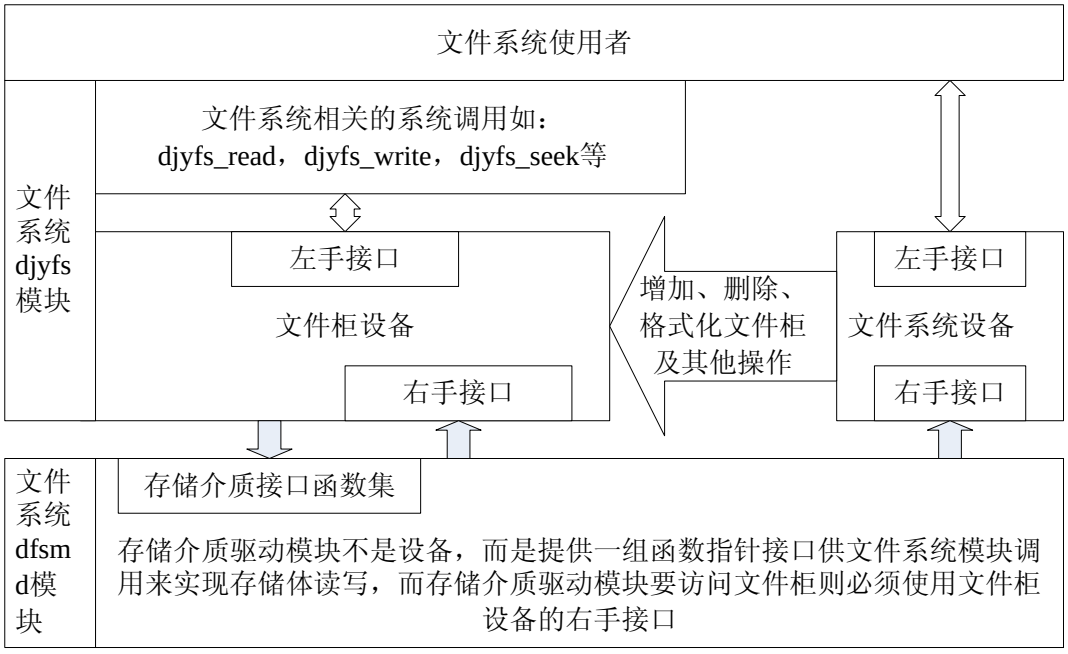


图 12-2 文件系统的整体结构

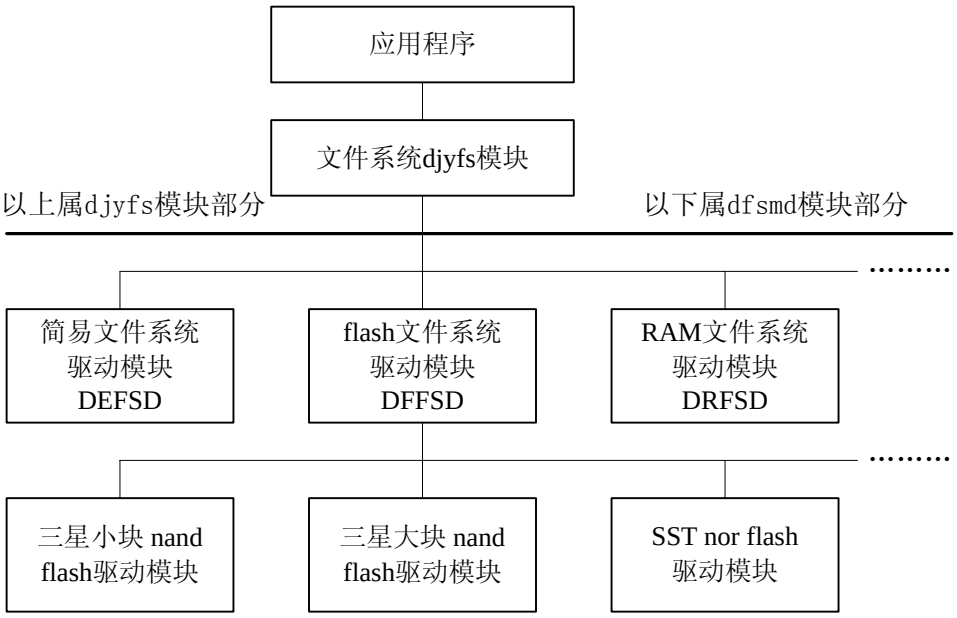


图 12-3 文件系统层次图

12.4 文件系统设备和初始化文件系统

文件系统设备是根设备“dev”的一个子设备，设备名称是“fs”，它是所有文件柜设备的父设备(是父设备而不是祖先设备)。“fs”设备并不是一个实体，真正的实体设备是它的子设备——文件柜设备。“fs”设备的接口可以用于管理文件柜，文件柜可以类比成 win/dos 系统的磁盘来理解，真正的文件和文件夹是保存在文件柜里面的。

如果你在工程配置时选择包含文件系统，在操作系统初始化时将调用module_init_djyfs函数，创建“fs”设备，并且打开一个指向“fs”设备的全局设备句柄pg_fs_handle_left。关于文件系统设备在系统设备树上的位置，参见第 12.3 节的 图 12-1 文件系统的资源视图。

代码 12-1 文件系统初始化函数

```
bool_t module_init_djyfs(void)
{
    pg_fs_dev = dev_add_root_device("fs",
                                    NULL,NULL,   //无信号量保护
                                    (dev_write_func) y_empty_func ,
                                    (dev_read_func ) y_empty_func,
                                    (dev_ctrl_func ) fs_right_ctrl ,
                                    (dev_write_func ) y_empty_func ,
                                    (dev_read_func   ) y_empty_func ,
                                    (dev_ctrl_func   ) fs_left_ctrl
                                    );      //"fs"是一个根设备
    if(pg_fs_dev == NULL)
        return false;     //建立文件系统设备失败
    pg_fs_dev->private_tag = 0;         //不使用私有标签
    rsc_add_root(&tg_opened_file_root,sizeof(struct   file_rsc),"opened file");
    pg_content_pool = mb_creat(tg_content_buf,sizeof(tg_content_buf),
                               cn_opened_fsnode_limit,sizeof(struct file_rsc),0);
```

```
    pg_fs_lhdl = dev_open_left_nsch("fs");    //打开文件系统设备左手接口
    return true;
}
```

1. fs 设备没有信号量保护，说明它是一个可以并发访问的设备，但并不是说，访问 fs 设备就不会被阻塞，因为 fs 设备的主要功能就是管理文件柜，是访问文件柜的桥梁。而文件柜是否可以并发访问，以及允许多少重并发访问，是由文件柜自己决定的，在绝大多数情况下，文件柜是单线程设备。如果某文件柜是单线程设备，当用户连续两次利用 fs 设备的 fs_left_ctrl 或 fs_right_ctrl 函数访问该文件柜时，第二次访问将被阻塞，如果连续两次访问的不是同一个单线程文件柜，则不会阻塞。
2. fs 设备只实现了 fs_left_ctrl 和 fs_right_ctrl 两个接口函数，其他 4 个都是空函数指针，再次强调一下，嵌入式环境不要使用 NULL 作为空函数指针，因为在嵌入式环境下，0 地址往往是复位向量地址，且没有存储器异常保护。
3. 文件系统中，fs 设备的功能比较简单，文件系统的主要功能是在文件柜设备中实现的，因此 fs 设备不使用 struct pan_device 结构的 private_tag 成员，就算把它初始化为 0 也不是必须的，只是一种编码习惯而已，你最好不要留下任何未初始化的数据。
4. 初始化文件系统的同时，还应该在系统资源链表中增加一个根结点，所有打开的文件和目录均作为一个资源挂在这个结点下，成为它的后代结点。
5. 每打开一个文件，都要分配一个struct file_rsc类型的数据结构，返回该结构的指针作为文件句柄。struct file_rsc是按照固定块内存分配的方法进行的，详见 7.3 节。初始化文件系统时调用mb_create函数初始化struct file_rsc内存池，该内存池是静态分配的，含 cn_opened_fsnode_limit 内存单元。应该注意的是，cn_opened_fsnode_limit并不等同于用户可以打开的文件数，请参见第 12.7 节了解详情。
6. 最后，打开“fs”设备的左手接口，因为 djyfs 模块本身要使用“fs”设备的左手接口。

12.5 文件柜设备

“文件柜”相当于 windows 系统的“磁盘”，实际上 djyfs 的早期版本中，使用的就是“磁盘”这个名字。并且在“磁盘”和“文件柜”这两个名称中摇摆了好一阵子，最终还是抛弃了“磁盘”而使用“文件柜”这个名字。win/dos 的 fat 文件系统的使用范围是如此广泛，“磁盘”这个名称可以顺应历史沿革和人们使用习惯，但它并不符合 djyos 的设计思想。因为“磁盘”这个名称隐含了特定存储介质——磁性存储的特征，而 djyfs 在文件柜这一层是与存储设备无关的，即使没有“磁盘”这一传统名称，它也不会以任何明确或隐含地与具体存储介质相关的名字。事实上，disk 的字面含义是存储数据的盘状物，微软也没有以存储介质为它命名，是中文翻译时译成跟磁性存储介质相关的名字。文件柜是文件系统和具体的 dfsmd 模块之间的设备接口，通过这个设备接口，操作系统可以访问到存储介质，可以对介质进行读写操作。从文件柜左手看过去，我们看到的是一个个文件和文件夹，因此，文件系统对文件柜的操作都是以文件名（文件夹名）为目标的。文件系统通过文件柜访问文件时，对存储模块发出类似“从某文件的某位置开始读出多少数据”的指令，而存储模块根据其对文件的组织方式，把数据从文件读出并写入到给定的缓冲区中。

划分文件柜有利于对对存储介质分组管理，djyfs 提供把存储介质划分为一个个文件柜的支持，但是否划分文件柜，以及如何划分，都是 dfsmd 模块的职责范围。一般来说，不同的存储介质需要建立不同的文件柜来管理其文件，而同一个物理存储介质也可以划分为多个文件柜。每个文件柜有单独的名字，从文件系统使用者的角度看，并不能区分文件柜与存储介质的关系。

代码 12-2 是struct st_DBX_device_tag的定义，每个文件柜设备都有一个这样的数据结构，其struct pan_device的private_tag指针指向这个结构。

代码 12-2 struct st_DBX_device_tag 结构定义

```
struct st_DBX_device_tag
{
    uint32_t    opened_sum;          //打开的文件(目录)总数
    bool_t      formatted;           //文件柜格式化标志
    char        name[cn_DBX_name_limit+1];    //文件柜名，相当于 dos 的卷标
    struct file_rsc *opened_root;
    ptu32_t DBX_medium_tag;     //特定文件柜的特征数据结构指针，由存储模块使用

    //文件柜功能函数
    //格式化文件柜，在 DBX_device_tag 中返回格式化后文件柜的参数，成功则 true，否则 false
    bool_t (*format)(uint32_t cmd,struct st_DBX_device_tag *DBX_device_tag);

    //文件柜文件读写函数，这些函数由 left_read left_write 函数使用
    //写文件，把 buf 中长度为 len 的数据写入到 file 的当前指针处
    uint32_t (*write)(struct file_rsc *fp,uint8_t *buf,uint32_t len);
    //读文件，从 file 当前指针处读取 len 长度的数据到 buf 中
    uint32_t (*read)(struct file_rsc *fp,uint8_t *buf,uint32_t len);
    //把写缓冲区的数据写入 flash
    uint32_t (*flush)(struct file_rsc *fp);
    //查询有多少可读数据，对于实质文件来说，就是(文件长度-当前指针)，但流的标准
    //定义并非一定如此，比如通信端子。
    uint32_t (*query_file_stocks)(struct file_rsc *fp);
    //查询目标文件还可以写入多少数据，一般等于文件柜的剩余空间，但流的标准
    //定义并非一定如此。
    sint64_t (*query_file_cubage)(struct file_rsc *fp);
    void (*check_DBX)(struct st_DBX_device_tag *DBX_device_tag,
                      sint64_t sum_size,sint64_t valid_size,sint64_t free_size);
    //以下是目录(文件)操作函数,这些函数由 left_ctrl 函数调用
    //设置文件长度，短于实际长度将截尾，长于实际长度将追加空间。
    sint64_t (*set_file_size)(struct file_rsc *fp,sint64_t new_len);
    //设置文件指针
    bool_t (*seek_file)(struct file_rsc *fp,struct seek_para *pos);
    //建立文件(目录)，attr 参数将表明建立目录还是文件。
    bool_t (*create_item)(char *name,struct file_rsc *parent,
                          union file_attrs attr);
```

```
    //删除一个文件(目录)，只能删除空目录
    bool_t (*remove_item)(struct file_rsc *fp);
    //打开文件(目录)，result 返回 item 信息，函数返回结果:
    //cn_fs_open_success,成功打开文件(目录)
    //cn_fs_item_exist,文件(目录)存在但不能打开(一般是模式不兼容)
    //cn_fs_item_inexist,文件(目录)不存在
    uint32_t (*open_item)(char *name,struct file_rsc *parent,
                          struct file_rsc *result,enum file_open_mode mode);
    //关闭目录(文件)，true = 成功，false = 失败(因无需分配资源，一般不会失败)
    bool_t (*close_item)(struct file_rsc *fp);
    //查找文件(目录)，result 返回 item 信息，但不分配
    //缓冲区，成功找到 item 返回 true，否则返回 false
    bool_t (*lookfor_item)(char *name,struct file_rsc *parent,
                           struct file_rsc *result);
    bool_t (*rename_item)(struct file_rsc *fp);
    //目录表操作函数
    //查询目录表尺寸
    uint32_t (*check_fdt_size)(struct st_DBX_device_tag *DBX_device_tag);
    //读目录表
    void (*read_fdt)(struct st_DBX_device_tag *DBX_device_tag,uint8_t *buf);
    //检查文件夹下子目录和文件的数量，不包含子目录下的文件。
    uint32_t (*check_folder)(struct file_rsc *folder);
};
```

struct st_DBX_device_tag结构定义了一系列的函数指针，这些函数是由文件系统公共部分使用，dfsmd模块实现的。更进一步地，它是文件柜设备的内部数据结构，只能由文件柜接口函数访问，文件系统的公共部分也不是直接调用这些函数的，而是通过文件柜设备的左手接口间接调用的。这些函数指针的详细说明，参见第 12.9 节。

代码 12-3 显示了djyfs中是如何创建文件柜并加入到系统设备树中的，代码 12-4 则演示了dfsmd模块如何为自己所属的存储器创建一个文件柜。

关于文件柜设备在系统设备树上的位置，参见第 12.3 节的 图 12-1 文件系统的资源视图。文件柜设备是文件系统设备“fs”的子设备，“fs”设备提供一个控制命令用于添加文件柜。文件柜由存储介质组成，应由dfsmd模块调用dev_ctrl函数把文件柜设备添加到设备树中，根据泛设备driver的约定（该约定不是强制性约定，参见第 9.1 节），dfsmd模块应该使用“fs”设备的右手接口，故右手控制函数fs_right_ctrl执行添加文件柜的命令。应该代码 12-3 节选了“fs”设备的右手控制函数中添加文件柜相关代码。代码 12-4 则演示了dfsmd模块如何添加文件柜。

创建文件柜后，用 dev_open_left(“fs\\a”)可以打开文件柜 a（相当于 win/dos 系统的 a 盘），也可以用 dev_open_left_son(pg_fs_lhdl,”a”);打开文件柜 a。当然，这些工作都是由操作系统的文件系统代码完成的，应用程序员无需关心。如果用户需要打开文件柜 a 下面的文件 abc.txt，只需要使用 djyfs_fopen(“a:\abc.txt”)就可以了。

代码 12-3 创建文件柜设备的代码

```
ptu32_t fs_right_ctrl(struct dev_handle *fs_rhdl,uint32_t right_cmd, ptu32_t data1,ptu32_t data2)
{
    switch(right_cmd)
```

```
    {
        case en_fs_add_dbx:
        {
            DBX_device_tag = (struct st_DBX_device_tag*)data1;
            DBX_device = dev_add_device(pg_fs_dev,DBX_device_tag->name,
                                        ((struct DBX_semp_para *)data2)->left,
                                        ((struct DBX_semp_para *)data2)->right,
                                        (dev_write_func) NULL,
                                        (dev_read_func ) NULL,
                                        (dev_ctrl_func ) DBX_right_ctrl,
                                        (dev_write_func ) DBX_left_write,
                                        (dev_read_func    ) DBX_left_read,
                                        (dev_ctrl_func    ) DBX_left_ctrl
                                        );
            if(DBX_device == NULL)
                return en_fs_creat_dbx_error;
            //文件柜设备私有标签指向文件柜结构专用数据结构
            DBX_device->private_tag = (ptu32_t)DBX_device_tag;
            if(DBX_device_tag->formatted)                                   //4
            {
                //申请根目录资源结点内存
                root_folder = mb_malloc(pg_content_pool,0);
                if(root_folder == NULL)
                {    //申请不到内存，释放早先建立的设备
                    dev_delete_device(DBX_device);
                    return en_fs_creat_dbx_error;
                }
                memset(root_folder,0,sizeof(struct file_rsc));
                //把新文件柜的根目录资源节点加入到文件根资源结点下，成为满子结点
                rsc_add_son(&tg_opened_file_root,&root_folder->file_node,
                                    sizeof(struct file_rsc),DBX_device_tag->name);
                //文件柜设备的打开文件的根结点指向该根资源结点。
                DBX_device_tag->opened_root = root_folder;
                root_folder->home_DBX = DBX_device;
            }else
                DBX_device_tag->opened_root = NULL;
            return en_fs_no_error;
        }break; //for en_fs_add_DBX
         ……        //fs_right_ctrl 函数的其他功能略。
        default:break;
    }
    return en_fs_no_error;
}
```

1. 要添加文件柜，应该使 fs_right_ctrl 函数的第二个参数 cmd = en_fs_add_dbx，

en_fs_add_dbx 是一个枚举常数，表示要执行在 fs 设备下添加文件柜设备。

2. data1 是struct st_DBX_device_tag* 类型的指针，将被赋值给文件柜设备struct pan_device结构的private_tag成员（参见第 9.4.2.2 节）。它是文件柜设备的核心数据结构，详见第 12.3 节。
3. data2 是 struct DBX_semp_para*类型的指针，提供该文件柜多线程并发访问的安全保护，struct DBX_semp_para 结构包含两个信号量，分别提供 struct pan_device 结构中的 left_semp 和 right_semp，因为 dev_ctrl 函数的参数不够，所以把两个指针组合成一个结构指针传递。文件柜的多线程安全保护是由 dfsmd 模块实现的，文件系统本身并不提供信号量保护机制。
4. 如果新加入的是已经格式化的文件柜，则要为其建立根目录资源结点，以后该文件柜所有打开的文件和目录都成为该结点的后代结点。所有文件柜的根目录结点都是“opened file”结点的子结点，关于“opened file”结点以及其后代结点组成的文件资源树，参见下一节。

代码 12-4 dfsmd 模块创建文件柜设备

```
struct dev_handle *fs_handle_right;
if((fs_handle_right = dev_open_right_nsch("fs")) == NULL)                    //1
    return result;
……        //省略的一系列与具体存储介质相关的代码
if(dev_ctrl(fs_handle_right,en_fs_add_DBX,
                    (ptu32_t)DBX_device_tag,(ptu32_t)&semp_dbx)
                    != en_fs_no_error)                                       //2
{
    goto goto_exit_install_chip;
}
dev_close_right(fs_handle_right);        //关闭文件系统设备
……
```

1. 新文件柜作为“fs”设备的子设备，添加设备前需要打开“fs”设备，添加文件柜的功能由“fs”设备的右手接口提供，故打开右手接口。
2. “fs”设备右手 dev_ctrl 的 en_fs_add_dbx 命令用于添加文件柜，需要提供 DBX_device_tag 的地址和信号量的地址。其中 DBX_device_tag 地址将保存在新文件柜的 struct pan_device 结构的 private_tag 成员中，这是实现文件系统的核心数据结构，它包含了本文件柜的所有特征数据。信号量可以为多线程同时访问文件柜时提供保护。

12.6 格式化文件柜

上一节把文件柜添加到系统设备树上后，在正式往文件柜中建立文件和存取数据之前，还需要格式化文件柜。djyfs 系统中，公共部分代码并不执行真正的格式化工作，而是把用户的格式化命令重定向到 dfsmd 模块提供的格式化命令上。这是因为，djyfs 只把文件柜和文件当作数据流的源和目的地，数据在存储介质上如何组织应该与存储器本身的特征相关，而与存储器本身特征相关的功能，自然应该由该存储介质的驱动程序完成，文件系统的公共部分不能为了降低 dfsmd 模块作者的工作量而越俎代庖。

代码 12-5 文件系统公共部分的格式化文件柜代码

```
bool_t djyfs_format(uint32_t cmd,char *dbx_name)
{
    uint32_t name_len,loop;
    bool_t result;
    struct dev_handle *DBX_lhdl;
    struct st_DBX_device_tag *DBX_device_tag;
    if(dbx_name == NULL)
        return 0;
    name_len = rtstrlen(dbx_name,cn_DBX_name_limit);            //1
    if((name_len == 0) || (name_len == cn_limit_uint32))
        return false;
    for(loop = 0;loop < name_len; loop++)
    {
        if((dbx_name[loop]=='*')||(dbx_name[loop]=='/')||(dbx_name[loop]=='?')
            ||(dbx_name[loop]=='<')||(dbx_name[loop]=='>')||(dbx_name[loop]=='|')
            ||(dbx_name[loop]=='"')||(dbx_name[loop]==':'))
              return false;     //名称串中不能出现这几个字符的
    }
    DBX_lhdl = dev_open_left_scion_nsch(pg_fs_lhdl,dbx_name);       //2
    if(DBX_lhdl == NULL)
        return false;
    DBX_device_tag = (struct st_DBX_device_tag *)
                                        DBX_lhdl->dev_interfase->private_tag;
    if(DBX_device_tag->opened_sum != 0)
        result = false;
    else
        result =    dev_ctrl(DBX_lhdl,en_DBX_format,cmd,0);
    dev_close_left(DBX_lhdl);
    return result;
}
//下面是文件柜的左手控制函数
ptu32_t DBX_left_ctrl(struct dev_handle *DBX_lhdl,uint32_t left_cmd,
                                        uint32_t data1,uint32_t data2)
{
    struct st_DBX_device_tag *DBX_device_tag = (struct st_DBX_device_tag *)
                                                (DBX_lhdl->dev_interfase->private_tag);
    switch((enum DBX_left_cmd)left_cmd)
    {
        case en_DBX_format:
        {//格式化文件柜，data1 是格式化参数
            return (ptu32_t)DBX_device_tag->format(data1,DBX_device_tag);       //3
        }break; //for en_DBX_format
          ……          // DBX_left_ctrl 函数的其他功能略。
    }
```

```
    return 0;
}
```

1. 注意这里使用rtstrlen函数而不是strlen函数（参见第 3.1.6 节）。
2. pg_fs_lhdl 是一个静态全局变量，除文件系统公共部分代码外，其他地方不得使用本变量。它保存的是初始化文件系统时打开的文件系统设备（“fs”设备）的左手句柄。
3. 文件柜左手控制函数 DBX_left_ctrl 接到格式化命令 en_DBX_format 后，简单地直接调用有 dfsmd 模块提供的格式化函数，所有实质操作都是有 dfsmd 模块完成的。

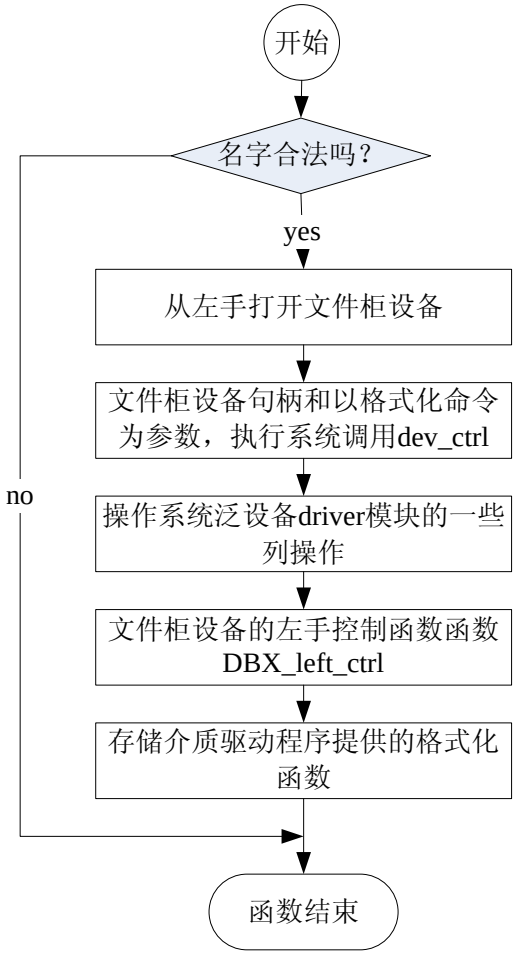


图 12-4 文件系统公共部分的格式化命令的执行流程

12.7 文件资源

12.7.1 文件资源树

在 djyfs 中，文件与文件柜不同，文件柜是作为设备出现在系统资源树中的，而打开的文件是作为普通资源出现在系统资源树中，文件柜中未打开的文件并不会出现在资源树中。区分两个重要的概念非常重要，即存储在文件存储介质中的文件和打开的文件，从广义上来说，他们都是“资源”，但是只有打开的文件出现在系统资源树中。从内存中的系统资源树

中查找文件显然比在文件存储介质中快，为什么不把所有文件都放在资源树中呢？有两个原因：

1. 即使不算文件缓冲区，在内存中保存一个文件结点需要超过 350 字节的内存，如果存储设备容量很大，保存的文件很多，就有撑破内存的可能。
2. 这样做会引入两个不确定性：内存需求不确定和执行时间不确定，随着保存在文件柜中的文件数量不断增多，系统资源树所需要的内存量也不断增多，而且没有明确的上限；初始化文件系统时势必要扫描整个目录表，导致执行时间不确定。我们知道，嵌入式系统经常用于无人值守的环境和实时环境，无人值守环境和实时环境都是拒绝不确定性的。

打开的文件资源如 图 12-5 所示，该图显示，系统中建立了已经格式化的文件柜“nand”和“nor”，以及未格式化的文件柜“nand1”。只要是已经格式化的文件柜，其根目录就会被打开并出现在“打开的文件”树中，所以nand1 是未格式化的。从“nand”和“nor”文件柜中总共打开了四个文件，他们是：

“nand:\folder1\\file3”

“nand:\folder1\folder2\file1”

“nand:\folder1\folder2\file2”

“nor:\folder1\file1”

“nor:\folder1\file2”

当前工作路径是“nor:\folder1\folder2\”。

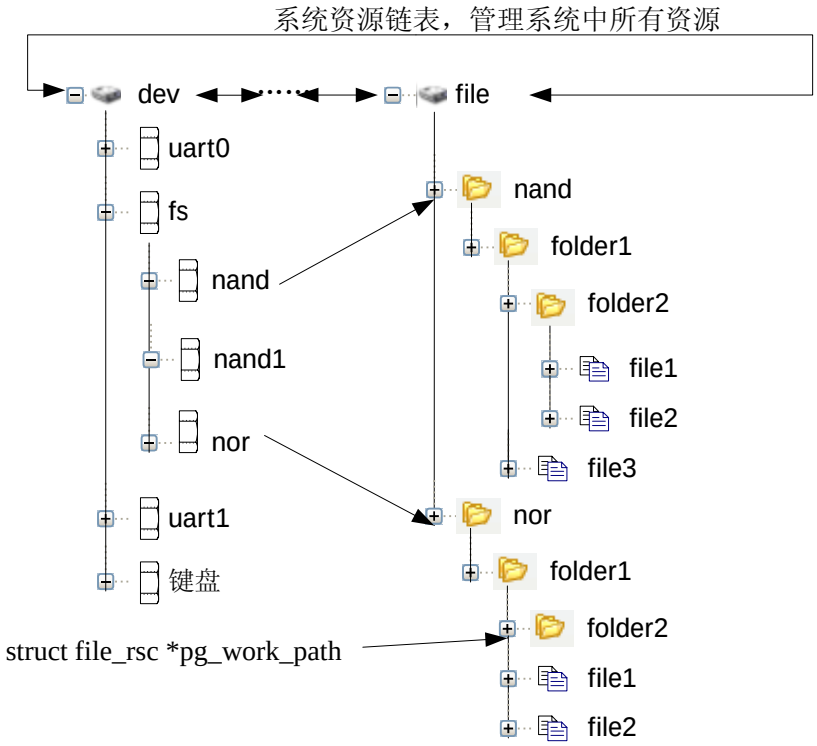


图 12-5 打开的文件资源

1. 文件系统中所有打开的文件和目录都将在系统资源链表中占据一个结点，“opened file”结点是 struct node 类型的结构，其他所有结点都是 struct file_rsc 类型的结构，而“opened file”结点是他们的共同祖先。
2. 每个已经格式化的文件柜的根目录都会被自动打开，并且成为“opened file”资源结点的子结点，而未格式化的文件柜则没有根目录。

3. 每个打开的文件都会在资源链表中占据一个结点，并且其每级路径都会作为独立结点被打开，文件和目录在资源链表中的相互关系一如其在存储介质中的相互关系。cn_opened_fsnode_limit 常数限制的是“opened file”的所有后代结点的总和（不包含“opened file”结点本身），并不严格等于允许打开的文件数量，这也是没有被定义为 cn_opened_file_limit 的原因。
4. 如果两个或以上被打开的文件的路径有重叠，重叠部分不会重复作为独立结点打开。
5. 文件系统维护一个当前工作路径指针 pg_work_path，当前工作路径是由文件系统的上层管理工具管理的一个环境变量。文件系统打开当前工作路径并使 pg_work_path 指针指向该路径的最后一个文件夹结点。
6. 除当前工作路径和根目录外，如果某文件夹下所有打开的文件都已经关闭，则该文件夹结点自动删除。例如“nand”文件柜的 file1 和 file2 都已经关闭后，folder2 结点自动被删除，因 folder1 下尚有 file3 还没有关闭，故 folder1 结点将会被保留。

代码 12-6 是struct file_rsc的定义。当用户打开文件时，得到的是struct file_rsc*类型的文件句柄。无论目录还是文件，都使用struct file_rsc类型。

代码 12-6 struct file_rsc 结构定义

```
struct file_rsc
{
    struct rsc_node file_node;       //把文件连接到系统资源树
    struct semaphore *file_semp;     //为文件访问提供信号量保护
    union file_attrs attr;           //文件属性
    enum file_open_mode open_mode;   //文件打开模式
    uint8_t  second_create;          //文件建立时间:second,2 位组合 BCD 码
    uint8_t  minute_create;          //文件建立时间:minute,2 位组合 BCD 码
    uint8_t  hour_create;            //文件建立时间:hour,2 位组合 BCD 码
    uint8_t  date_create;            //文件建立日期,2 位组合 BCD 码
    uint8_t  month_create;           //文件建立月份,2 位组合 BCD 码
    uint8_t  year_low_create;        //文件建立年度,4 位组合 BCD 码
    uint8_t  year_high_create;       //文件建立年度,4 位组合 BCD 码
    uint8_t  second_modify;          //文件修改时间:second,2 位组合 BCD 码
    uint8_t  minute_modify;          //文件修改时间:minute,2 位组合 BCD 码
    uint8_t  hour_modify;            //文件修改时间:hour,2 位组合 BCD 码
    uint8_t  date_modify;            //文件修改日期,2 位组合 BCD 码
    uint8_t  month_modify;           //文件修改月份,2 位组合 BCD 码
    uint8_t  year_low_modify;        //文件修改年度,4 位组合 BCD 码
    uint8_t  year_high_modify;       //文件修改年度,4 位组合 BCD 码
    struct pan_device *home_DBX;     //本文件所属的文件柜
    ptu32_t file_medium_tag;         //与媒体相关的标记，其含义由特定文件系统
                                     //driver 解释，如 flash file driver)中，保存
                                     //该文件(目录)的 FDT 项目号。
    struct ring_buf read_rw_buf;     //独立读写文件的读缓冲区，或合并读写文件的
                                     //读写缓冲区
    struct ring_buf write_buf;       //文件写缓冲区
    struct ring_buf *p_read_buf;     //读缓冲区指针
```

```
    struct ring_buf *p_write_buf;    //写缓冲区指针
    uint32_t open_counter;           //文件打开次数计数
    sint64_t file_size;              //文件尺寸,字节数，不含仍在缓冲区的数据
    sint64_t read_ptr;               //当前读指针
    sint64_t write_ptr;              //当前写指针
    uint32_t eno;           //最近一次出错类型，en_fs_no_error 表示从来没有出错
    char     name [cn_file_name_limit+1];  //文件名（目录名）
};
```

struct file_rsc 结构定义注释很详细，这里就部分成员补充说明一下：

struct semaphore *file_semp;

是否使用信号量保护文件访问由 dfsmd 模块决定，文件系统的公共部分代码不理会这个成员，比如 flash 文件系统就不使用这个成员。

union file_attrs attr;

文件属性，定义如下：

```
struct file_attr_bits
{
    uint8_t read_only:1;     //只读文件
    uint8_t hidden:1;        //隐藏文件
    uint8_t folder:1;        //目录
    uint8_t archive:1;       //档案文件
    uint8_t deleted:1;       //文件被删除
};
union file_attrs
{
    struct file_attr_bits bits;
    uint8_t all;
};
```

需要特别说明的是，union和C语言的位域变量如何存储与编译系统紧密相关，属于移植不安全的数据类型。而文件系统应该是跨平台使用的，因此，attr成员不能按其在内存中的格式直接保存到存储介质中，而是应该转换成统一的格式存储。代码 12-7 是djyos flash 文件系统driver中fdt.c中的代码片段，它把struct file_rsc中的attr成员翻译成可以直接存储在flash中的变量attr_mirror。

代码 12-7 翻译文件属性

```
    uint8_t attr_mirror;
    //因存在可能的移植问题，不宜直接使用位联合结构的 attr 参数
    attr_mirror = 0;
    if(attr.bits.read_only)
        attr_mirror +=cn_FS_ATTR_READ_ONLY;
    if(attr.bits.hidden)
        attr_mirror +=cn_FS_ATTR_HIDDEN;
    if(attr.bits.folder)
        attr_mirror +=cn_FS_ATTR_DIRECTORY;
    if(attr.bits.archive)
        attr_mirror +=cn_FS_ATTR_ARCHIVE;    //以上只初始化 4 个属性，还两个暂不用
```

enum file_open_mode open_mode;

这是一个包含 6 个值的枚举变量，与 ansi 标准文件系统的模式字符串对应，如下：

en_r_rb,　　对应“r”和“rb”

en_w_wb,　　对应“w”和“wb”

en_a_ab,　　对应“a”和“ab”

en_r_rb_plus, 对应“r+”和“rb+”

en_w_wb_plus, 对应“w+”和“wb+”

en_a_ab_plus, 对应“a+”和“ab+”

struct pan_device *home_DBX;

打开的文件（目录）的文件柜指针，指向这个文件（目录）所属的文件柜设备指针。每个打开的文件（目录）都应该知道自己所属的文件柜，用户操作该文件时文件系统才知道被操作的目标文件柜。

ptu32_t file_medium_tag;

文件的存储介质签章，在 dfsmd 模块中需要用到它，文件系统的公共部分并不使用这个成员，文件柜设备提供一个右手控制命令，可以让 dfsmd 模块凭这个签章找到“opened file”资源树中相应的文件。

struct ring_buf read_rw_buf;

struct ring_buf write_buf;

struct ring_buf *p_read_buf;

struct ring_buf *p_write_buf;

这几个是由 dfsmd 模块使用的缓冲区，用于可缓冲的存储介质，把缓冲区放在 dfsmd 模块中管理，是因为数据是否可以被缓冲以及缓冲区如何使用，与实际使用的存储介质相关，文件系统的公共部分不能对它做出假设。

12.7.2 创建文件

unix中creat函数是专门用于创建文件的，但是ansi C中没有专门用于创建文件的函数，创建文件是在fopen函数中间接实现的。djyfs与ansi C一致，执行djyfs_fopen系统调用时，如果使用的模式字符串不是“r”，“rb”，“r+b”，“r+”，“rb+”中的任意一个，则目标文件不存在时会创建新文件。详见第 12.7.3 节。

12.7.3 打开文件

系统调用djyfs_fopen用于打开现有文件，或者创建并打开新文件，函数最终将调用lookfor_item函数查找目标文件是否存在，如存在则调用open_item函数打开，若文件（目录）不存在，且mode允许创建新文件，则调用create_item函数创建新文件（目录），其中lookfor_item、open_item、create_item函数都是由dfsmd模块实现的。至于从系统调用开始如何辗转调用dfsmd模块的流程，参见“图 12-4 文件系统公共部分的格式化命令的执行流程”。

mode参数规定了被打开文件的访问模式，djyfs_fopen函数能接受ansi C规定的所有模式字符串，但含义并不完全一致，见 表 12-1 所示。

表 12-1 mode 含义对照表

模式	ansi C 的 fopen 中的含义	djyfs_fopen 中的含义
r	打开一个文本文件用于读	打开一个文件用于读。

rb	打开一个二进制文件用于读	
w	以只写方式打开一个文本文件，打开后立即清空文件，若目标文件不存在则创建新文件并以只写方式打开。	以只写方式打开一个文件，打开后立即清空文件，若目标文件不存在则创建新文件并以只写方式打开。
wb	以只写方式打开一个二进制文件，打开后立即清空文件，若目标文件不存在则创建新文件并以只写方式打开。	
a	按追加数据方式打开一个文本文件，若目标文件不存在则创建新文件并以追加数据方式打开。	按追加数据方式打开一个文件，若目标文件不存在则创建新文件并以追加数据方式打开。
ab	按追加数据方式打开一个二进制文件，若目标文件不存在则创建新文件并以追加数据方式打开。	
r+	打开一个文本文件用于读写，若目标文件不存在则失败。	打开一个文件用于读写，若目标文件不存在则失败。
r+b 或 rb+	打开一个二进制文件用于读写，若目标文件不存在则失败。	
w+	以可读写方式打开一个文本文件，打开后立即清空文件，若目标文件不存在则创建新文件并以可读写方式打开。	以可读写方式打开一个文件，打开后立即清空文件，若目标文件不存在则创建新文件并以可读写方式打开。
w+b 或 wb+	以可读写方式打开一个二进制文件，打开后立即清空文件，若目标文件不存在则创建新文件并以可读写方式打开。	
a+	按追加数据且可读的方式打开一个文本文件，若目标文件不存在则创建新文件并以追加数据且可读的方式打开。	按追加数据且可读的方式打开一个文件，若目标文件不存在则创建新文件并以追加数据且可读的方式打开。
a+b 或 ab+	按追加数据且可读的方式打开一个二进制文件，若目标文件不存在则创建新文件并以追加数据且可读的方式打开。	

在文件系统内部，则把模式字符串转换成具有相同含义的枚举变量访问，共有 6 个枚举值，枚举值与模式字符串的对照关系如 表 12-2 所示。

表 12-2 枚举值与模式字符串对照表

枚举变量	对应的模式字符串
en_r_rb	“r”，“rb”
en_w_wb	“w”，“wb”
en_a_ab	“a”，“ab”
en_r_rb_plus	“r+”，“rb+”，“r+b”
en_w_wb_plus	“w+”，“wb+”，“w+b”
en_a_ab_plus	“a+”，“ab+”，“a+b”

每个文件都有一个读写指针表明当前读/写位置，文件资源数据结构struct file_rsc中的成员sint64_t read_ptr保存的是下一次读数据的位置，成员sint64_t write_ptr保存的是下一次写数据的位置。文件刚打开时，读写指针有一个初始位置，这个初始位置与文件的打开方式有关，对文件的读、写、seek操作也都可能会影响文件读写指针的位置，那么，读写指针是如何确定的呢？表 12-3 显示了在各种文件操作后读写指针以及文件长度的变化表。

表 12-3 文件读写指针位置表

模式	项目	打开文件	读 n 字节后	写 n 字节后	seek 到新位置
en_r_rb	文件尺寸	原长度	不变	原长度	原长度
	read_ptr	0	+n	—①	=新位置②
	write_ptr	无效	无效	—	无效
en_w_wb	文件尺寸	0	不变	+n	若新位置大于文件尺寸，则延长文件，延长部分填充数据由 dfsmd 模块决定。若新位置小于文件尺寸，则不变。
	read_ptr	无效	—	无效	无效
	write_ptr	0	—	+n	=新位置
en_a_ab	文件尺寸	原长度	不变	+n	无效
	read_ptr	无效	—	无效	无效
	write_ptr	文件末	—	文件末	文件末
en_r_rb_plus	文件尺寸	原长度	不变	若新写指针大于文件尺寸则，则新文件尺寸=新写指针，否则不变。	不变
	read_ptr	0	+n	+n	=新位置
	write_ptr	0	+n	+n	=新位置
en_w_wb_plus	文件尺寸	0	不变	若新写指针大于文件尺寸则，则新文件尺寸=新写指针，否则不变。	不变
	read_ptr	0	+n	+n	=新位置
	write_ptr	0	+n	+n	=新位置
en_a_ab_plus	文件尺寸	原长度	不变	+n	不变
	read_ptr	0	+n	不变	=新位置
	write_ptr	文件末	文件末	文件末	文件末

注①：“—”表示该模式不支持相应操作。

注②：本表中所有“新位置”均不能超过文件长度。

代码 12-8 打开文件系统调用

```
djyfs_file *djyfs_fopen(char *fullname, char *mode)
{
    struct file_rsc *result;
    struct    dev_handle *DBX_lhdl;
    struct st_DBX_device_tag *DBX_device_tag;
    char *synname;

    if(__check_fullname(fullname) == false)        //fullname 不是一个合法的字符串      1
```

```
        return NULL;
    DBX_lhdl = __open_DBX_left(fullname);     //2
    if(DBX_lhdl == NULL)
        return NULL;      //文件柜不存在，或 fullname 和当前路径均没有指定文件柜名
    DBX_device_tag = (struct st_DBX_device_tag *)
                        DBX_lhdl->dev_interfase->private_tag;
    if(DBX_device_tag->formatted)
    {
        synname = __pick_synname(fullname);                    //3
        //调用文件柜的左手控制函数
        result = (struct file_rsc *)dev_ctrl(DBX_lhdl,en_DBX_open,
                            (ptu32_t)synname,(ptu32_t)mode);
    }else
        result = NULL;
    dev_close_left(DBX_lhdl);
    return result;
}
```

1. fullname的合法性判断请参见第 12.2 节的约定。
2. __open_DBX_left 函数用于打开文件柜，规则如下：
   a) fullname 中包含文件柜名，则打开该文件柜名。
   b) fullname 不包含文件柜名，则打开当前路径所在的文件柜。
3. __pick_synname 函数用于去掉 fullname 中可能的文件柜名，因为 DBX_lhdl 已经是文件柜句柄了。

__DBX_open_file是文件系统公共部分的函数，djyfs_fopen经过类似 图 12-4 的流程后，调用文件柜左手控制函数DBX_left_ctrl，DBX_left_ctrl函数调用__DBX_open_file。代码 12-9 显示了该函数的执行过程。

代码 12-9 打开文件

```
djyfs_file *__DBX_open_file(struct dev_handle *DBX_lhdl,
                                    char *synname,char *mode)
{
    struct st_DBX_device_tag *DBX_device_tag;
    struct file_rsc *parent,*son;
    struct file_rsc *opened = NULL;     //保存最后一个已经打开的目录项
    uint32_t open_result;
    uint16_t next_char_off;
    char       name[256];
    union file_attrs attr;
    enum file_open_mode my_mode;
    bool_t need_to_creat = true;       //表示当文件不存在时，是否需要创建
    DBX_device_tag = (struct st_DBX_device_tag*)
                        (DBX_lhdl->dev_interfase->private_tag);
    if((strcmp(mode,"r")==0) || (strcmp(mode,"rb")==0))
    {
        need_to_creat = false;
```

```
        my_mode = en_r_rb;
    }else if((strcmp(mode,"w")==0) || (strcmp(mode,"wb")==0))
        my_mode = en_w_wb;
    else if((strcmp(mode,"a")==0) || (strcmp(mode,"ab")==0))
        my_mode = en_a_ab;
    else if((strcmp(mode,"r+")==0)||(strcmp(mode,"rb+")==0)
                                    ||(strcmp(mode,"r+b")==0))
    {
        need_to_creat = false;
        my_mode = en_r_rb_plus;
    }else if((strcmp(mode,"w+")==0) || (strcmp(mode,"wb+")==0)
                                    || (strcmp(mode,"w+b")==0))
        my_mode = en_w_wb_plus;
    else if((strcmp(mode,"a+")==0) || (strcmp(mode,"ab+")==0)
                                    || (strcmp(mode,"a+b")==0))
        my_mode = en_a_ab_plus;
    else
        return NULL;
    if(__if_abs_path(synname))    //synname 中已经去掉了文件柜名，无需再判断
    {
        parent = DBX_device_tag->opened_root;   //从根目录开始操作
        next_char_off = 1;
    }else
    {
        parent = pg_work_path;   //从当前路径开始操作
        next_char_off = 0;
    }

    while(__pick_path_word(synname,next_char_off,name))
    {//沿路径逐级打开目录，但不会打开文件
        //name 是模块内部提供的字符串指针，已经经过字符串长度合法性检查
        next_char_off += strlen(name) +1;       //+1 是为跳过字符'\'
        if((son = (struct file_rsc *)rsc_search_son(&parent->file_node,name))
                        != NULL)
        {//目标已经打开
            opened = son;
            if(son->open_counter != cn_limit_uint32)
                son->open_counter ++;
        }else
        {//目标文件(目录)尚未打开
            if(!(son = mb_malloc(pg_content_pool,0)))
                goto exit_open_err;       //分配内存失败
            son->home_DBX = parent->home_DBX;
            //查找目录并初始化文件数据结构
```

```
open_result = DBX_device_tag->open_item(name,parent,son,my_mode);
if(open_result == cn_fs_open_success)
{//目录存在，并且已经初始化目录信息
    //打开目录，实际上就是把目录结点挂到打开的文件资源树上。
    rsc_add_eldest_son(&parent->file_node,&son->file_node,
                       sizeof(struct file_rsc),son->name);
    if(DBX_device_tag->opened_sum != cn_limit_uint32)
        DBX_device_tag->opened_sum ++;
    son->open_counter = 1;
    son->eno = en_fs_no_error;
}else if(open_result == cn_fs_item_exist)     //目录存在，但模式不匹配
{
    goto exit_open_err;
}else
{//目录不存在，看是否需要创建。
    if(need_to_creat)
    {//需要创建
        attr.all = 0;          //先把所有属性初始化为 false
        attr.bits.folder = 1;  //创建的是目录
        if(DBX_device_tag->create_item(name,parent,attr) == false)
        {//创建目录失败
            mb_free(pg_content_pool,son);
            goto exit_open_err;
        }else
        {
            if(DBX_device_tag->open_item(name,parent,son,my_mode)
                                    == cn_fs_open_success)
            {
                //打开目录实际上就是把目录结点挂到打开的文件资源树。
                rsc_add_eldest_son(&parent->file_node,
                                   &son->file_node,
                                   sizeof(struct file_rsc),
                                   son->name);
                if(DBX_device_tag->opened_sum != cn_limit_uint32)
                    DBX_device_tag->opened_sum ++;
                son->open_counter = 1;
                son->eno = en_fs_no_error;
            }else
            {
                mb_free(pg_content_pool,son);
                goto exit_open_err;
            }
        }
    }else
```

```
                {//不需要创建，当以 r 或 r+方式打开文件时，不存在也不创建。
                    mb_free(pg_content_pool,son);
                    goto exit_open_err;
                }
            }
            //项目所属文件柜
            son->home_DBX = (struct pan_device*)DBX_lhdl->dev_interfase;
        }
        parent = son;     //以当前打开的目录为下次使用的父目录
    }
    //至此，目录已经全部打开(或创建)，下面打开文件
    if(__pick_filename_word(synname,name))   //synname 串中包含文件名吗?
    {//打开文件
        if((son = (struct file_rsc *)rsc_search_son(&parent->file_node,name))
                                            == NULL)
        {//文件尚未打开
            if(!(son = mb_malloc(pg_content_pool,0)))
                goto exit_open_err;       //分配内存失败
            son->home_DBX = parent->home_DBX;
            //查找并打开文件
            open_result = DBX_device_tag->open_item(name,parent,son,my_mode);
            if(open_result == cn_fs_open_success)
            {//文件存在并且可以按 my_mode 模式打开，已经初始化文件信息
                rsc_add_eldest_son(&parent->file_node,&son->file_node,
                                    sizeof(struct file_rsc),son->name);
                if(DBX_device_tag->opened_sum != cn_limit_uint32)
                    DBX_device_tag->opened_sum ++;
                son->open_counter = 1;
                son->eno = en_fs_no_error;
            }else if(open_result == cn_fs_item_exist)     //文件存在，但模式不匹配
            {
                goto exit_open_err;
            }else
            {//文件不存在，看是否需要创建。
                if(need_to_creat)
                {//需要创建
                    attr.all = 0;
                    attr.bits.archive = 1;        //是文件
                    if(DBX_device_tag->create_item(name,parent,attr) == false)
                    {//创建文件失败
                        mb_free(pg_content_pool,son);
                        goto exit_open_err;
                    }else
                    {
```

```
                        if(DBX_device_tag->open_item(name,parent,son,my_mode)
                                == cn_fs_open_success)
                        {
                            rsc_add_eldest_son(&parent->file_node,
                                               &son->file_node,
                                               sizeof(struct file_rsc),
                                               son->name);
                            son->open_counter = 1;
                            son->eno = en_fs_no_error;
                            if(DBX_device_tag->opened_sum != cn_limit_uint32)
                                DBX_device_tag->opened_sum ++;
                        }else
                        {
                            mb_free(pg_content_pool,son);
                            goto exit_open_err;
                        }
                    }
                }else
                {//r 或 r+方式，不需要创建。
                    mb_free(pg_content_pool,son);
                    goto exit_open_err;
                }
            }
        }else
        {//文件已经打开
            if(my_mode == son->open_mode)
            {//已经打开的文件，只能以相同的方式再次打开。并且读写指针不改变
                if(son->open_counter != cn_limit_uint32)
                    son->open_counter ++;
                return son;
            }else
                return NULL;
        }
    }else    //打开目录，目录在 if(index != 0)语句前已经打开，且无需计打开次数
        return son;
    son->home_DBX = (struct pan_device*)DBX_lhdl->dev_interfase;
    return son;
exit_open_err:
    //删除已经添加的资源节点和释放分配的内存,opened 保存的是本次增加的
    //第一个资源节点的上一级节点。
    if(opened == NULL)
        return NULL;
    son = (struct file_rsc *)rsc_get_twig(&opened->file_node);
    while(son != NULL)
```

```
    {
        rsc_del_node(&son->file_node);
        mb_free(pg_content_pool,son);
        son = (struct file_rsc *)rsc_get_twig(&opened->file_node);
    }
    rsc_del_node(&opened->file_node);
    mb_free(pg_content_pool,opened);
    return NULL;
}
```

函数中有详细的注释，就不再占用篇幅详细解释了。

12.7.4 删除文件（目录）

与创建文件对应，文件系统还应该提供删除文件（目录）的系统调用。本函数的系统调用部分代码与 代码 12-9 几乎一样，唯一的不同就是调用文件柜设备的dev_ctrl函数时使用的命令字不一样，在这里就不重复占用篇幅了。当以en_DBX_remove命令调用文件柜的左手控制函数DBX_left_ctrl时，实际调用__djyfs_DBX_remove_item函数执行删除操作。

代码 12-10 删除文件

```
uint32_t __djyfs_DBX_remove_item(struct dev_handle *DBX_lhdl,char *synname)
{
    struct file_rsc item;
    struct st_DBX_device_tag *DBX_device_tag;
    DBX_device_tag = (struct st_DBX_device_tag *)
                                        (DBX_lhdl->dev_interfase->private_tag);
    if(__item_if_opened(DBX_device_tag,synname) == true)
        return (uint32_t)en_fs_object_opened;           //不能删除已经打开的对象
    if(__if_abs_path(synname) == true)
    {//synname 是当前文件柜的绝对路径
        if(!DBX_device_tag->lookfor_item(synname+1,
                                        DBX_device_tag->opened_root,&item))
            return (uint32_t)en_fs_object_nonentity;    //被删除的对象不存在
    }else
    {//synname 是相对与 pg_work_path 的相对路径
        if(!DBX_device_tag->lookfor_item(synname,pg_work_path,&item))
            return (uint32_t)en_fs_object_nonentity;    //被删除的对象不存在
    }
    item.open_counter = 0;          //打开次数清零
    item.home_DBX = (struct pan_device*)DBX_lhdl->dev_interfase;
    if(item.attr.bits.read_only == true)
        return (uint32_t)en_fs_object_readonly;         //不能删除只读对象
    if(item.attr.bits.folder)
    {//是个目录，需要判断是否空
        if(DBX_device_tag->check_folder(&item) != 0)
```

```
                return (uint32_t)en_fs_folder_unblank;
    }
    if(DBX_device_tag->remove_item(&item))
        return en_fs_no_error;
    else
        return en_fs_remove_error;
}
```

代码中有完整的注释，值得注意的地方是，从存储介质中删除文件的操作最终是由dfsmd模块提供的remove_item函数指针完成的。

12.7.5 读、写文件

文件系统公共部分读文件和写文件的过程非常相似，这里仅以写文件为例说明之，至于读文件的实现过程详参随书代码。代码 12-11 是写文件的系统调用代码，size和nmemb两个参数是非常有趣的。按照ansi C的规定，size不是需要写入的字节数，而是记录数，每个记录的长度由nmemb确定，两个参数的乘积size*nmemb才是实际需要写入的字节数。这样设置参数实际上规定了数据是按照一个一个记录保存在文件中的，每个记录的长度都是nmemb字节，并且要求文件系统按整数个记录读取和写入数据，并不符合djyos系统的设计思想，djyos系统认为，文件仅仅是一个数据源而已，至于这些数据是否按记录存储，以及每个记录的长度，应该是数据使用者的事，不予关心。那么，ansi C这样规定是错误的吗？djyfs_fwrite函数原型应该改为

size_t djyfs_fwrite2（void *buf，size_t size，djyfs_file *fp）；吗？

显然不能这么武断，我们的前辈学者这样规定，自然有他的道理，我们应该给予更多的尊重。我们看到，djyfs_fwrite 函数的返回值是实际写入的记录数，也就是说，这个函数考虑到了物理存储设备可能因某些原因不能完整地写入数据，实际写入的数据可能比要求的少！在大多数应用中，数据文件可能确实是按记录组织的，如果真的按 djyfs_fwrite2 的格式写入数据的话，文件系统无法确保写入整数个记录，而按照 djyfs_fwrite 的格式，则文件系统可以保证写入整数个记录。读取数据的时候也一样，按整数个记录读取，避免读指针停留在一个记录的中间。遗憾的是，ansi C 规定的 fseek 函数的参数 offset 是绝对字节数，而不是按照记录移动指针的，这是否 ansi C 的疏忽呢？如果推测 fwrite 函数的参数按记录写入是为了保证记录的完整性，那么 fseek 的 offset 参数和 fwrite 的参数是矛盾的。

在 djyfs 中，在这方面做了一个折中，一方面，djyfs_fwrite 和 djyfs_fread 系统调用遵守 ansi C 的规定，而在 dfsmd 模块，取消了记录格式支持，文件柜设备提供的读写数据函数指针的原型分别是：

```
uint32_t (*write)(struct file_rsc *fp,uint8_t *buf,uint32_t len);
uint32_t (*read)(struct file_rsc *fp,uint8_t *buf,uint32_t len);
```

为了确保整数个记录读写，djyfs_fwrite 函数和 djyfs_fread 函数利用了另外两个函数：

```
sint64_t (*query_file_stocks)(struct file_rsc *fp);
sint64_t (*query_file_cubage)(struct file_rsc *fp);
```

query_file_stocks 函数用于查询文件中从当前指针开始还有多少可读数据，query_file_cubage 函数用于查询文件还能写入多少数据，在读写之前先调用这两个函数，可以确保调用 write 和 read 函数时，按整数个记录进行。

代码 12-11 写文件系统调用

```
size_t djyfs_fwrite(void *buf,size_t size, size_t nmemb,djyfs_file *fp)
{
    size_t result;
    struct pan_device *DBX_pan_device;
    struct dev_handle *DBX_lhdl;
    struct rw_para    para;
    if(fp == NULL)
        return 0;
    if((size == 0) || (nmemb == 0))
        return 0;     //写入的数据量是 0
    DBX_pan_device = fp->home_DBX;
    DBX_lhdl = dev_open_left_scion(pg_fs_lhdl,DBX_pan_device->node.name,0);
    if(DBX_lhdl == NULL)
        return false;
    if(fp->attr.bits.folder)
    { //目录不能写
        result = 0;
        goto goto_end;
    }
    if(fp->open_mode == en_r_rb)
    {//只读文件，不能写
        result = 0;
        goto goto_end;
    }
    para.nmemb = nmemb;
    para.size = size;
    result = dev_write(DBX_lhdl,(ptu32_t)buf,
                                        (ptu32_t)&para,(ptu32_t)fp);
goto_end:
    dev_close_left(DBX_lhdl);
    return result;
}
```

函数代码比较简单，就不逐行解释了，djyfs_fwrite经过类似 图 12-4 的流程后，调用文件柜左手写函数DBX_left_write，代码 12-12 是DBX_left_write函数的代码。

代码 12-12 文件柜设备左手写接口函数

```
uint32_t DBX_left_write(struct dev_handle *DBX_lhdl,ptu32_t buf,
                                    ptu32_t write_para,ptu32_t file)                //1
{
    struct rw_para *pl_write_para = (struct rw_para *)write_para;
    uint32_t write_len;   //需要写入的数据长度
    sint64_t rest_len;
    uint32_t result;
    struct file_rsc *fp = (struct file_rsc *)file;   //取得文件指针
    struct st_DBX_device_tag *DBX_device_tag = (struct st_DBX_device_tag *)
```

```
                                DBX_lhdl->dev_interfase->private_tag;
    write_len = pl_write_para->nmemb * pl_write_para->size;   //计算需写入的长度
    //查询物理设备空闲空间
    rest_len = DBX_device_tag->query_file_cubage(fp);
    if(rest_len >= write_len)
    {//物理设备上有足够的空间
        result=DBX_device_tag->write(fp,(uint8_t*)buf,write_len);          //2
        result = result / pl_write_para->nmemb;
    }else
    {//物理设备上没有足够的空间，写入整数个完整记录
        write_len = (rest_len / pl_write_para->nmemb) * pl_write_para->nmemb;
        result=DBX_device_tag->write(fp,(uint8_t*)buf,write_len);
        result = result / pl_write_para->nmemb;
    }
    return result;
}
```

1. 除第一个参数的类型是设备句柄“struct dev_handle *”外，其他 3 个参数的类型都是ptu32_t，ptu32_t类型的说明见第 9.4.2.2 节关于struct pan_device结构的说明。
2. write函数指针指向由dfsmd模块定义的写函数，参见第 12.9 节。

12.7.6 关闭文件

当文件使用完毕后，应当及时关闭，一方面可以回收有限的文件句柄资源，防止发生资源泄漏；另一方面可以使数据及时写入到存储介质中，djyfs 支持的是带缓冲的流式文件，写入数据时，文件系统尽量在缓冲区中收集数据，在必要的时候（缓冲区满、调用 flush 或关闭文件）写入到存储介质中，如果文件操作已经完成，就应该及时关闭以使数据写入到存储设备，以防断电丢失。

代码 12-13 关闭文件系统调用。

```
uint32_t djyfs_fclose(djyfs_file *fp)
{
    bool_t result;
    struct pan_device *DBX_pan_device;
    struct dev_handle *DBX_lhdl;
    if(fp == NULL)
        return cn_limit_uint32;
    DBX_pan_device = fp->home_DBX;
    DBX_lhdl = dev_open_left_scion(pg_fs_lhdl,DBX_pan_device->node.name,0);
    if(DBX_lhdl == NULL)
        return cn_limit_uint32;
    result = (bool_t)dev_ctrl(DBX_lhdl,en_DBX_close,(ptu32_t)fp,0);
    dev_close_left(DBX_lhdl);
    if(result)
        return 0;
```

```
    else
        return cn_limit_uint32;
}
```

本函数比较简单，就不再逐句解释了。在经过类似 图 12-4 所示的流程后，dev_ctrl函数将辗转调用__DBX_close_item函数，该函数将执行实际的关闭操作。关闭文件时将会把被关闭的文件结点从资源链表中删除，结点删除以后，如果上级目录结点不再有子结点，那么，上级目录结点也要被删除，所有删除资源结点的操作并不涉及到物理存储器

代码 12-14 关闭文件

```
bool_t __DBX_close_item(struct dev_handle *DBX_lhdl,struct file_rsc *fp)
{
    struct st_DBX_device_tag *DBX_device_tag;
    struct file_rsc *parent,*son;
    DBX_device_tag =(struct st_DBX_device_tag *)
                                (DBX_lhdl->dev_interfase->private_tag);
    if(fp->attr.bits.folder)
    {
        if(rsc_get_son(&fp->file_node) != NULL)
            return false;           //被关闭的目录仍有已经被打开的子项，不能关闭
    }
    //下面把文件从资源链表中删除，包括已经空的目录
    son = fp;
    parent = (struct file_rsc *)rsc_get_parent(&son->file_node);
    do
    {//从被删除结点开始，逐级向上删除空目录。空目录是指资源链表上的空目录结点，
     //而物理存储器上该目录可能不是空的
        if(son == pg_work_path)   //当前工作路径不能被删除
            break;
        DBX_device_tag->close_item(son);
        rsc_del_node(&son->file_node);    //从打开文件资源链表中删除结点
        mb_free(pg_content_pool,son);                    //释放结点内存
        son = parent;                    //下一循环释放父结点
        parent=(struct file_rsc*)rsc_get_parent(&parent->file_node);//父结点上移
        if(DBX_device_tag->opened_sum != 0)
            DBX_device_tag->opened_sum --;
    }while(son != DBX_device_tag->opened_root);      //直到根目录
    return true;
}
```

函数体内已经有充分的注释说明，不再逐行解释了。

12.7.7 其他文件操作

其他操作包括 seek 文件指针、文件改名等等，其实现过程与读写文件类似，有兴趣的读者可以自行参考随书光盘中的代码，在这里就不占用篇幅了。

12.8 数据结构访问权限

文件系统由 djyfs 模块和 dfsmd 模块组成，涉及到复杂的数据结构，而这些数据结构中有相当部分是两者共享的，需要界定 djyfs 模块和 dfsmd 模块各自的访问权限。

表 12-4 数据结构访问权限

大类代号： FSD = 文件系统设备。
DBXD = 文件柜设备。
FRN = 文件资源结点。
DDT = struct st_DBX_device_tag
FR = struct file_rsc

大类	成员/操作	说明	djyfs 模块	dfsmd 模块
FSD	初始化		实现	
	读写函数	空		
	左手控制函数		实现，可使用	不可访问
	右手控制函数		实现，不可使用	可使用
	private_tag	空		
DBXD	初始化		实现	
	左手接口函数		实现，可使用	不可访问
	右手接口函数		实现，不可使用	可使用
	private_tag		赋值为 DDT	
FRN			可使用	不可访问
DDT	opened_sum	打开的文件数	打开、关闭文件时改写	只读，建议不要访问
	formatted	文件柜格式化否	只读	上电扫描时赋值 创建文件柜时赋值 格式化时赋值
	name	文件柜名	只读	创建文件柜时赋值 修改文件柜名时赋值 上电扫描时赋值
	opened_root	本文件柜被打开的文件资源树根目录结点	创建文件柜设备时赋值 格式化文件柜时赋值	不可访问
	DBX_medium_tag	存储介质标签	只读	自由使用
	所有函数指针		只读（即调用）	编写函数，赋值指针
FR	file_node	见第 8 章 资源管理		
	file_semp	文件并发访问保护	不可访问	根据具体需要使用，也可不使用
	attr	文件属性	只读	打开文件时赋值，其他情况只读
	open_mode	文件打开模式	只读	可写

	second_create~year_high_modify	文件时间	只读	打开文件时赋值，其他时候只读
	home_DBX	所属文件柜	打开文件时赋值	只读
	file_medium_tag	存储介质标签	只读	根据实际需要使用
	read_rw_buf	读缓冲区或综合读写缓冲区	不可访问	可写
	write_buf	写缓冲区	不可访问	可写
	p_read_buf	读缓冲区指针	不可访问	可写
	p_write_buf	写缓冲区指针	不可访问	可写
	open_counter	文件打开次数	打开文件时赋值	不可访问
	file_size	文件尺寸	只读	可写
	read_ptr	文件读位置	不可访问	可写
	write_ptr	文件写位置	不可访问	可写
	eno	最近一次出错代码	出错时写入	只读
	name	文件名	只读	可写

注：

1. djyfs 模块和 dfsmd 模块都可以读 DDT 和 FR 的所有成员而不会导致副作用，写操作受上表限制。所谓的不可访问只是从模块独立性出发，要求某模块对这些成员“视而不见”。
2. djyos 由 C 语言编写，C 语言没有访问权限控制，如果用户违反上表，编译器不能限制，也不会告警。

12.9 存储介质接口函数

12.9.1 format 格式化文件柜

函数指针的原型为

bool_t (*format)(uint32_t cmd,struct st_DBX_device_tag *DBX_device_tag);

参数：

cmd：格式化选项。

DBX_device_tag：被格式化的文件柜的设备标签。

返回值：文件柜的格式化状态，无论是文件柜无需格式化还是本来已经格式化，只要函数执行的结果是文件柜已经格式化，返回 true，文件柜未格式化则返回 false。

这是文件柜格式化函数，功能是按目标存储器特定的格式初始化存储器，并在系统资源树中建立该文件柜的根目录结点。本函数与具体存储介质类型密切相关，可能很复杂，也可能只是一个空函数。

12.9.2 write 写文件

函数指针的原型为

uint32_t (*write)(struct file_rsc *fp,uint8_t *buf,uint32_t len);

参数：

buf：写入的数据源缓冲区指针。

len：写入的数据长度，与 djyfs_fwrite 系统调用不同，len 是以字节为单位的。

fp：写入的目标文件。

返回值：实际写入的数据量。

本函数把以 buf 为首地址、长度为 len 的数据写入文件中，本函数返回实际写入存储介质的数据量，如果该驱动程序支持数据缓冲，则写入的数据可能先保存到缓冲区中，计算返回值时，缓冲区和物理存储器是等同对待的。本函数是有 dfsmd 模块实现的，如何使用缓冲区，由 dfsmd 模块决定，根据处理方法不同，返回值的计算方法分以下几种情况：

1. 缓冲区中原有 100bytes，len =1000bytes，缓冲区足够大，新的 1000bytes 数据全部保存在缓冲区中，返回值是 1000；
2. 如果缓冲区空间不够了，写入时把 1000bytes 连同缓冲区原有的 100bytes 一起，共 1100bytes 一起写入存储器中，返回值不是 1100，而是 1000。
3. 又或者，缓冲区长度是 500，写入时把原有的 100bytes 写入存储器，新写入的 1000 字节则 500 字节写入存储器，另外 500 字节保存在缓冲区中，返回值仍然是 1000。

注意了，len 的类型是 uint32_t，它隐含着一个事实是，虽然单个文件的长度可以超过 4G 字节，但单次执行写操作写入的数据量不能超过 4G 字节。

12.9.3 read 读文件

函数指针的原型为

uint32_t (*read)(struct file_rsc *fp,uint8_t *buf,uint32_t len);

参数：

buf：保存读出的数据的缓冲区指针。

len：读出数据长度，与 djyfs_fread 系统调用不同，len 是以字节为单位的。

fp：读数据的源文件。

返回值：实际读出的数据量。

本函数与 write 函数正好相反从 DBX_device_tag 文件柜中的 fp 文件中读取数据。当存储设备驱动模块支持缓冲读时，与 write 函数一样，无论数据来自物理存储器还是缓冲区，都返回用户实际得到的数据量。

12.9.4 flush 刷新文件

函数指针的原型为

uint32_t (*flush)(struct file_rsc *fp);

参数：

fp：需刷新的目标文件。

返回：实际写入存储介质的数据量

本函数对支持写缓冲的 dfsmd 模块有效，把滞留在缓冲区的数据写入存储介质中，关闭文件时也会执行同样的操作。如果驱动程序不支持写缓冲，则本函数空。

12.9.5 query_file_stocks 查文件库存

函数指针的原型为

uint32_t (*query_file_stocks)(struct file_rsc *fp);

参数：

fp：被查询的目标文件。

返回：该文件有多少可读数据，-1 表示未知或不限量，通信端子可能返回-1。

查询有多少可读数据，对于静态数据文件来说，就是(文件长度-当前读指针)，djyfs 支持的是流式文件，流的标准定义并非一定如此。返回值应该是从现在起还能从该文件获取的数据的总和，对静态数据文件来说，不难理解，但对于按文件访问的通信口，则容易与串口接收缓冲区中暂存的数据量混淆。通信口可能继续传输，它的返回结果应该是-1，而不是接收缓冲区中的数据量。

12.9.6 query_file_ cubage 查文件库容

函数指针的原型为

sint64_t (*query_file_cubage)(struct file_rsc *fp);

参数：

fp：被查询的目标文件。

返回：该文件还可写入多少数据，-1 表示未知或不限量，通信端子可能返回-1。

查询目标文件还可以写入多少数据，对于静态数据文件，等于文件柜的剩余空间，djyfs 支持的是流式文件，流的标准定义并非一定如此。与 query_file_stocks 函数一样，按文件访问的通信口，应返回-1。

12.9.7 set_file_size 设置文件长度

函数指针的原型为

sint64_t (*set_file_size)(struct file_rsc *fp,sint64_t new_len);

参数：

fp：被设置的目标文件。

new_len：新的文件长度。

返回：新的文件长度，若与 new_size 参数不等，一般是因为文件柜没有足够的容量。

对于按文件访问的通信口这样的文件系统，或者是只读文件系统，本函数无效；对于静态数据文件，如果 new_len 大于文件尺寸，则需要延长文件，延长部分填入随机内容；如果 new_len 小于文件尺寸，则需要截断文件，截掉的部分数据将丢失。本函数一般用于在数据写入之前预先分配存储器。

需要注意的是，本函数只关心文件本身的属性，并不理会文件的打开模式，如果是只读文件，不执行任何操作；但如果是按只读模式打开的可读写文件，根据模块职能分工，dfsmd 模块不知道也不应该关心上层按什么模式打开文件，故本函数将正常操作，保证不针对这样的文件调用本函数应该是文件系统公共部分的职责。

12.9.8 seek_file 设置文件指针

函数指针的原型为

bool_t (*seek_file)(struct file_rsc *fp,struct seek_para *pos);

参数：

fp：需设置文件指针的目标文件。

pos：文件指针的新位置。

返回：true=操作成功，false=失败

对于按文件访问的通信口这样的文件系统，seek操作无效。对于静态数据文件，seek执行的操作参见 表 12-3。

12.9.9 create_item 创建文件或目录

函数指针的原型为

bool_t (*create_item)(char *name,struct file_rsc *parent, union file_attrs attr);

参数：

name：新文件、目录的名字

parent：新创建文件的父目录

attr：文件属性，其内容与保存在文件存储介质中的文件属性的内容相同，但该参数使用了位域，位域在内存中的格式与编译器相关，应该转换成固定的格式保存在存储介质中。

返回：true=成功创建，false=失败。

建立目录和建立文件都使用这个函数，attr 参数中的 folder 位将表明要建立的是目录还是文件。本函数只负责创建，创建后不会自动立即打开，如果需要打开，还需单独调用 open_item 函数。

12.9.10 remove_item 删除文件或目录

函数指针的原型为

bool_t (*remove_item)(struct file_rsc *item);

参数：

item：待删除的文件或目录

返回：true=成功删除，false=失败。

删除一个文件，或者目录。djyfs规定，不能删除已经被打开的文件，但remove_item函数并不判读文件（目录）是否被打开，这个工作是由djyfs模块完成的，djyfs模块调用本函数前，就应该确保被删除的目录（文件）是未打开的。如果删除的是目录，则只能删除空目录，如果要删除整个目录，需要与rsc_get_twig函数（参见第 13 节）配合，逐个调用本函数把目录内的所有文件和子目录删除，然后才能删除本目录。如果被删除的是只读文件（目录），则需要先修改文件（目录）的只读属性。

12.9.11 open_item 打开文件或目录

函数指针的原型为

bool_t (*open_item)(char *name,struct file_rsc *parent,
struct file_rsc *result,enum file_open_mode mode);

参数：

name：待打开的文件或目录名字，是一个单一的名字，不能包含路径。

parent：待打开的文件或目录的父目录

result：用于返回被打开的文件或目录的各种参数。

mode：文件打开模式，是一个枚举变量，其可能值以及含义详见 表 12-1、表 12-2 和表 12-3 所示。

返回：true=成功删除，false=失败。

本函数是 struct st_DBX_device_tag 结构的各函数指针中比较复杂的一个函数，它的工作包括：

1. 在所属的文件柜中查找目标文件是否存在。
2. 如果存在，则从目录表中读取该文件的信息，比如文件创建时间、文件属性、文件尺寸等。
3. 把上述数据转换成内存中 struct file_rsc 的格式。目录表在存储器中是按固定格式存储的，可以在不同的计算机系统中交换数据。而随着 CPU 系统和编译系统的改变，在不同环境下本地的 struct file_rsc 资源的格式可能并不一致。
4. 根据 mode 参数，如果该驱动程序支持缓冲读写功能，则可能需要为其分配读写缓冲区，并设置各相关读写文件指针。

12.9.12 close_item 关闭文件或目录

函数指针的原型为

bool_t (*close_item)(struct file_rsc *fp);

参数：

fp：待关闭的文件或目录指针。

返回：true=成功关闭，false=失败。

本函数比较简单，由于文件柜设备和文件资源链表由 djyfs 模块管理，关闭文件的大部分工作都由 djyfs 模块完成了，dfsmd 部分只是做一些善后工作。例如在 flash 文件系统中，驱动程序做的工作仅仅是把逗留在缓冲区中的数据写入 flash 中，然后释放缓冲区。

12.9.13 lookfor_item 查找文件或目录

函数指针的原型为

bool_t (*lookfor_item)(char *name,struct file_rsc *parent, struct file_rsc *result);

参数：

name：待查找的文件或目录名字，是一个单一的名字，不能包含路径。

parent：待查找的文件或目录的父目录

result：用于返回被打开的文件或目录的各种参数。如果=NULL，则只返回文件（目录）

是否存在的结果，不返回文件相关信息。

返回：true=找到文件或目录，false=文件或目录不存在。

本函数实际上是 open_item 函数的一个子集，有两处不一致：

1. 没有 mode 参数，不管是否支持缓冲读写，都无需为文件分配缓冲区。
2. 允许 result=NULL，若 result=NULL，不读取和转换文件或目录的信息。

12.9.14 rename_item 修改文件或目录的名字

函数指针的原型为

```
bool_t (*rename_item)(struct file_rsc *fp);
```

参数：

fp：待改名的文件或目录指针，新的文件名包含在这个指针指向的结构里面。

返回：true=成功改名，false=失败。

本函数修改 fp 指定的文件的名字，在目录表中找到 fp 指针对应的目录表项，修改名字后写回到存储介质中。需注意的是，djyfs 支持长度达 255 字符的文件（目录）名，但具体的驱动程序可能不支持这么长的文件名。当用户传入的文件名长度超过驱动程序支持的长度时，文件名将会被截短，至于如何截取文件名，将由驱动程序自己决定，但把截断后的文件名写回 fp 中。

12.9.15 check_folder 查询子项目数量

函数指针的原型为

```
uint32_t (*check_folder)(struct file_rsc *folder);
```

参数：

check_folder：被检查的文件夹。

返回：文件夹中包含的文件和文件夹数量，不含子文件夹下的文件和文件夹。

12.10 dfsmd 部分

从 图 12-2 所示的文件系统层次结构中，我们看到，djyos的文件系统由公共部分（djyfs模块）和dfsmd模块组成，其中djyfs模块对上直接面对应用程序，对下面对dfsmd模块，不针对特定存储介质；而dfsmd模块面对djyfs模块和特定的物理存储介质，为djyfs模块提供用于操作存储器的接口函数，组织管理存储介质并从中读写文件，从djyfs模块角度看过去，dfsmd模块的所有工作就是：

1. 初始化struct st_DBX_device_tag数据结构，该结构见第 12.5 节。
2. 实现 struct st_DBX_device_tag 结构中各接口函数指针。
3. 通过文件系统设备“fs”的右手接口把文件柜安装到设备树。
4. 如果有目录表，也由驱动程序管理。djyfs 认为，目录表的组织形式与存储介质的特性密切相关，应该由它自己的驱动程序管理。
5. 组织存储介质，数据在存储器中的存储方式完全由 dfsmd 模块决定。
6. 实现从存储介质的读、写文件等逻辑操作，它可能直接操作物理存储器，也可能进一步调用存储器的物理驱动实现。

7. 管理数据流缓存，包括申请缓冲内存、读写缓冲区等。

在整个文件系统的组成中，dfsmd 模块是弹性最大的部件，由于涉及到存储介质的组织和管理，驱动程序可能很庞大，也可能很短小。

第13章 flash 文件系统驱动程序

flash 文件系统驱动程序简称 DFFSD（djyos flash file system driver），是用于 djyos 文件系统的一种“dfsmd 模块”，它以嵌入式系统中广泛使用的 flash 存储器作为存储介质，凡是具有下列特征的物理存储器都适合用 DFFSD 作为驱动程序与 djyfs 连接：

1. 针对不可靠存储器，该存储器在出厂时就可能存在坏块，并且在使用过程中还会陆续产生新的坏块。
2. 分块存储，块尺寸至少 256 字节，且必须是 8 的整数倍。
3. 必须先擦除才可以写入，擦除以整块为单位进行，不限制有更小的擦除单位。在已经被擦除的区域，可以随机写入。
4. 写入次数有限而读取次数无限。

flash 存储器分为两种大类，一类是 nor flash，另一类是 nand flash，DFFSD 不区分所用 flash 的类型。市场上销售的 flash 芯片这类繁多，而且在不断增多，DFFSD 不可能涵盖所有芯片，因此，DFFSD 把芯片驱动代码独立出来，芯片驱动代码由应用系统设计者编写，DFFSD 通过标准格式的函数指针调用芯片驱动代码，实现对 flash 芯片读写。值得注意的是，针对不可靠存储器而作的数据校验也是有芯片驱动完成的，DFFSD 提供了几个校验函数供芯片驱动工程师选用，芯片驱动工程师也可以自己实现校验算法，djyos 系统认为，校验算法和校验码的存储方法都是与芯片密切相关的特性，应该由最直接访问芯片的模块完成，这样才能保证模块的独立性。

DFFSD 有如下特点：

1. 有完善的掉电保护措施，设掉电恢复块，修改一块时，该块内容先拷贝到掉电恢复块，然后才改写原块。
2. 文件系统的重要信息双备份存储。
3. 灵活的擦除平衡管理，保证存储器均衡磨损，保证一个文件柜内擦写次数最高和最低之差不超过 1024 次。
4. flash 驱动的公共部分和芯片驱动部分合理划分，层次分明，接口明确，便于用户在自己的系统中实现 flash 文件系统驱动。
5. 可选的 ECC 校验，当允许 ECC 校验时，可以在不可靠存储器上可靠地存储数据，当文件柜用于存储多媒体文件时，可选择取消 ECC 校验以加快存取速度。

13.1 芯片存储体布局

图 13-1 是flash芯片的整体布局图，MDR表是位于芯片头部的一个数据区域，它保存着该芯片的基本数据，芯片初始化代码读取MDR表以建立该芯片的内存视图——struct flash_chip类型的数据结构，读取MDR_DBX中保存的数据，找到文件柜的位置，读取文件柜的DDR记录，建立完整的文件系统数据结构。

MDR 表的长度为 4Kbytes，它占用整数块空间，如果块尺寸小于 4K，则取块尺寸的整数倍向上取，直到大于等于 4096，如果块尺寸大于 4096，则等于一块。

MDR 表共两份，分别是 MDR_main 和 MDR_bak，当 MDR_main 需要擦除重写时，就把 MDR_main 中的有效内容读出写到 MDR_bak 中，除滚动记录区外的其他部分，MDR_bak

和 MDR_main 是完全一致的，滚动记录区则舍弃老旧的数据，只把最终数据写入到 MDR_bak 中。

用户可以设定若干块为保留区，保留区由用户自行管理，用户可以根据自己的特定应用决定是否设置保留区，以及决定保留块的数量。MDR_main 从保留区后的第一个有效块开始存储，MDR_bak 则从 MDR_main 后第一个有效块开始存储。MDR 表的存储位置一经确定就不再改变，即使该块产生了瑕点，MDR 的磨损次数被严格控制，还使用增强的校验来保证可靠性。MDR 表不是整块做校验，而是各部分内容单独校验，CDR（32 字节）表、DBX（52 字节）表都有自己的校验码，滚动记录区则每条记录（8 字节）都有自己的校验码，校验码采用可以发现 2 位错误纠正 1 位错误的 ECC 码，以增强可靠性。芯片每 100 万次文件擦写操作，滚动记录区才增加 2 条记录，滚动记录区可以存储 426 条记录，而只有滚动记录区填满了，才会把 MDR 表整体擦除一次，同时 MDR_bak 上增加 3 条记录。这样，就很好地控制了 MDR 表的擦除次数。

在 MDR 表后面，就是文件柜的实际存储位置了，用户可以在每片芯片上建立 1~3 个文件柜，每个文件柜的大小也可以由用户任意指定，DFFSD 会把用户设定的文件柜尺寸自动向上调整为块尺寸的倍数。

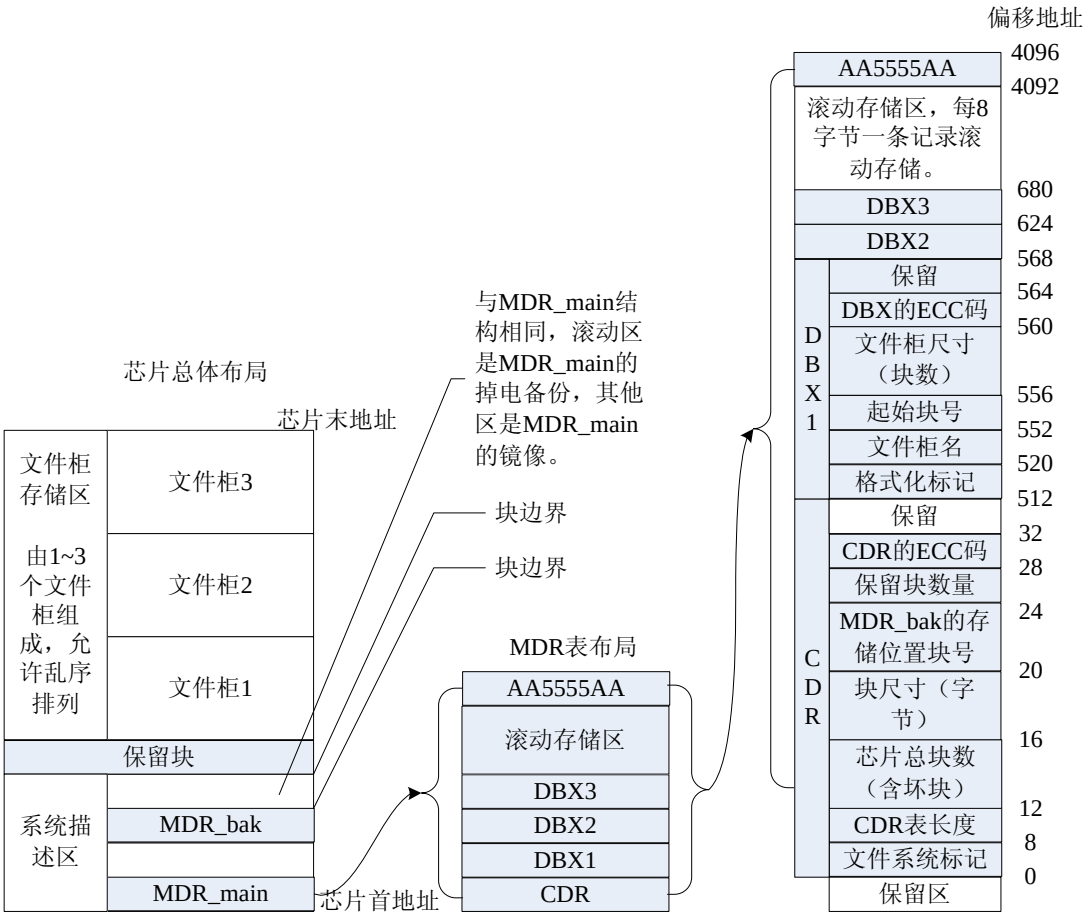


图 13-1 芯片布局图

图 13-1 的说明：

1. 多字节数据全部按照小端方式存储。
2. 作为数据完整性标记的 AA5555AA，允许有一位不相符。

文件柜在芯片内占用连续的存储区域，图 13-2 是一个文件柜的布局图。

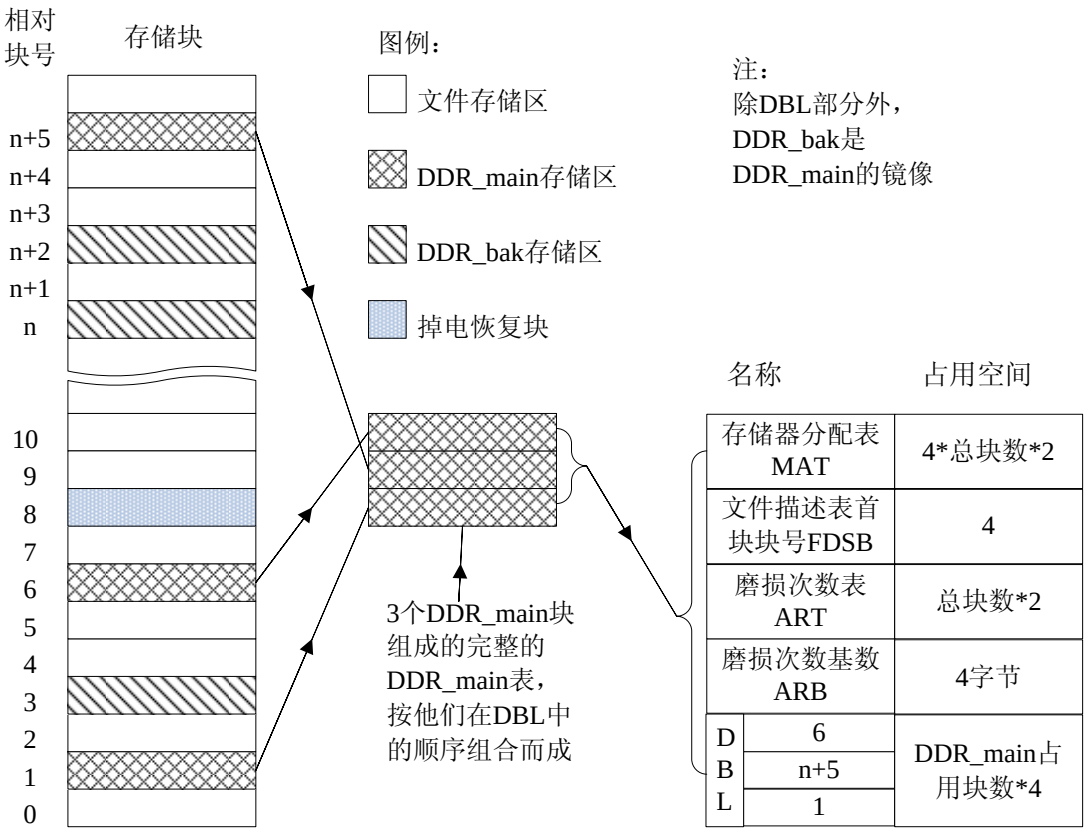


图 13-2 文件柜布局图

图 13-2 说明如下：

1. 管理一个文件柜需要的数据由 DDR 表（双备份）和一个掉电恢复块组成，这些数据存储在文件柜内，其中 DDR 表的长度要根据文件柜占用的块数和块尺寸计算得到，图中所示的文件柜的 DDR 表占用 3 个存储块。
2. DDR 表的存储位置不固定，可能位于文件柜内的任意地方，且顺序也不确定。保存 DDR 表本身的块号表是 DDR 表中的 DBL 区域，散布在整个文件柜中的 DDR 块按其在 DBL 表中的顺序组合成连续的 DDR 表。
3. DDR 表在文件柜中双备份保存。
4. 主 DDR 表和备份 DDR 表的首块块号保存在芯片 MDR 表的滚动记录区中。DDR 表内部，块号用相对于文件柜的块偏移表示，在滚动记录区中是按照其在芯片中的绝对块号表示的。
5. 文件柜布局图中没有出现文件描述表（FDT 表，相当于 win/dos 的目录表），因为 DFFSD 把目录表当作特殊文件，保存在文件存储区中。
6. 掉电恢复块是一个特殊的块，是为了防止文件操作过程中掉电损坏数据而设置的。

DDR 表个分量说明：

1. DBL 表依序保存本 DDR 表存储在文件柜内部的块号，构成 DDR 表的各存储块可以独立出现在文件柜内的任意位置，按 DBL 中的顺序可以组合除一份完整的 DDR 表出来。
2. 磨损次数基数 ARB 和磨损次数表 ART，DFFSD 的磨损平衡算法可以确保文件柜内各块的磨损次数之差不超过 1024，ART 表用 2 字节表示各块相对于 ARB 的磨损次数，该值与 ARB 相加即得到相应块的实际磨损次数。
3. 文件描述表（FDT）首块块号 FDSB，FDT 表描述文件柜内所有目录和文件的组织

结构以及其他信息，随着文件柜的使用，FDT 表的长度是动态的，DFFSD 象文件一样管理 FDT 表，FDSB 就是 FDT 表的首块，定位首块后，再根据 MAT 表就可以读出完整的 FDT 表。

4. 存储器分配表 MAT，该表把所有文件占用的存储块用链表串起来，每个文件一个独立的链表，其功能类似于 FAT 文件系统的 FAT 表。

图 13-3 显示的是MDR表中滚动记录区的结构图，从 图 13-1 中可以看到，一个芯片可以划分为 3 个文件柜，每个文件柜的起始块号和占用的块数可以从MDR_DBXx得到，正如 图 13-2 显示的那样，文件柜的内部信息是保存在DDR表中的，只有读取DDR表才能正确访问文件柜，而DDR表在文件柜内的保存位置并不确定。DFFSD把每个文件柜的DDR表和备份DDR表首块块号以及掉电恢复块块号保存在MDR表中，当这 3 个记录改变时，需要写入MDR表中，为了降低MDR表的磨损次数，MDR表的最后开辟了一个滚动记录区用于保存这些记录。至于掉电恢复块为什么要保存在MDR中而不是保存在相应文件柜的DDR中，参见第 13.4 节。

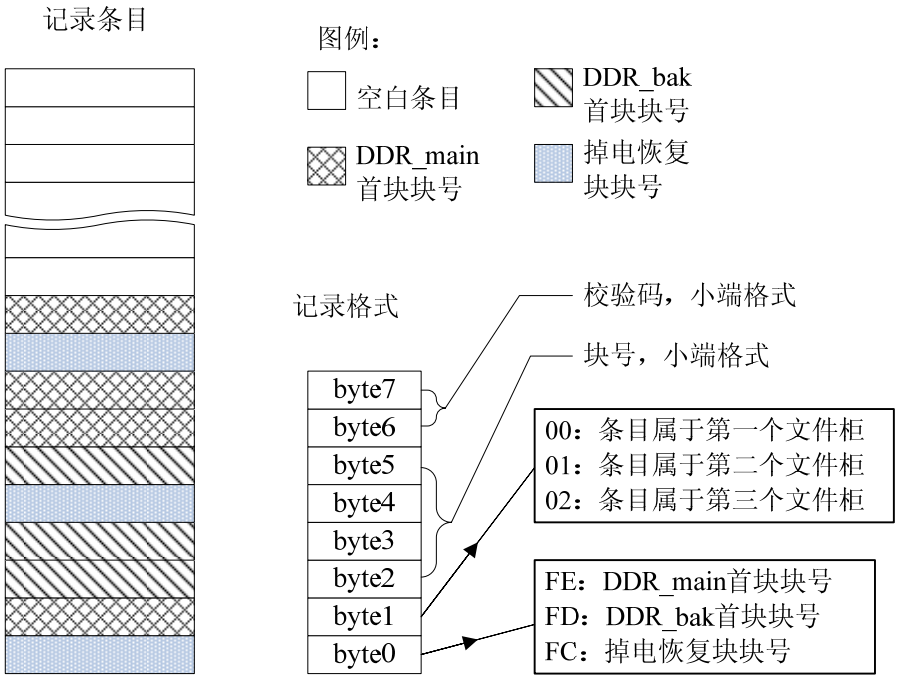


图 13-3 MDR 表的滚动记录区

1. 滚动记录区按 8 字节一条记录组织。
2. 当有新记录产生时，把新记录写在滚动记录区的最后，就像更新日志一样，并不改写和擦除滚动区原来的记录，相同类型记录的最后一条记录即有效记录。
3. 一个芯片上最多可以建立 3 个文件柜，每个文件柜有 3 种记录，所以滚动记录区中的有效记录最多有 9 条。
4. 上电（复位）后，文件系统加载程序扫描整个滚动记录区，取出有效记录。
5. 每条记录都有单独的校验码，使用能够纠正错误的校验算法，以确保记录的有效性。
6. 文件柜的磨损平衡算法可以有效控制各记录的写入次数，保证滚动记录区的寿命。
7. 当滚动区写满以后，取出滚动区的所有有效记录，写入 MDR_bak 的滚动区中，然后擦除整个滚动区，从头开始重新记录。

13.2 数据结构

DFFSD模块主要数据结构有两个，一是struct st_DBX_flash_tag，文件柜设备的私有成员sruct st_DBX_device_tag的DBX_medium_tag成员作为指针指向这个结构；另一个struct flash_chip结构，它描述的是flash芯片，它在系统资源树上构成一棵芯片资源树。这两个结构与“图 12-1 文件系统的资源视图”相结合，可形成图 13-4 所示的一个完整的文件系统数据结构视图。

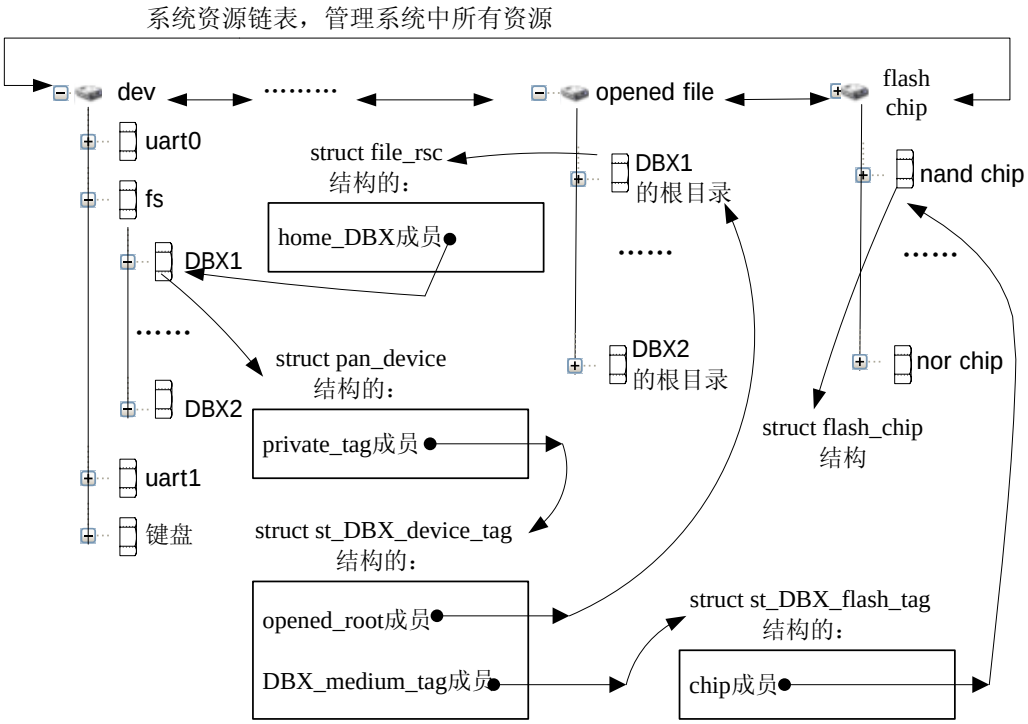


图 13-4 DFFSD 数据结构视图

图 13-4中出现的数据结构，struct pan_device见第9.4.2.2节，struct st_DBX_device_tag见第 12.5 节，struct file_rsc见 代码 12-6。本节着重介绍struct st_DBX_flash_tag和struct flash_chip，他们的定义见 代码 13-1。代码中有注释可以帮助理解，其具体应用见后续章节介绍。

代码 13-1 DFFSD 数据结构定义

```
struct flash_chip
{
    struct rsc_node chip_node;
    char *chip_name;
    uint32_t block_sum;              //总块数
    uint32_t block_size;             //块尺寸
    uint32_t valid_sum;              //可用块数
    uint32_t rev_blocks;             //保留块数量
    uint32_t rev_start;              //保留区首块块号，保留区前是芯片描述块。
    uint32_t DBX_start;              //用于文件柜的首块块号
    uint32_t DBX_read_buf_size; //读文件缓冲区尺寸
    uint32_t DBX_write_buf_size;//写文件缓冲区尺寸
```

```
struct st_MDR_flag MDR;       //MDR 表相关参数
uint8_t    *block_buf;        //块缓冲区
struct semaphore_LCB *left_semaphore;
struct semaphore_LCB *right_semaphore;

//以下是典型的 flash 操作函数指针，编写一个具体芯片的驱动时，设为具体芯片的
//操作函数,这些函数并不是每一种芯片都具有，没有的用 NULL。

//检查写入参数规定的内容时，是否需要擦除。如果 buf=NULL 则检查由 offset 和 size
//设定的区域是否被擦除
//true = 已准备好，不需要擦除(或空块),false = 需要擦除；
bool_t     (*query_block_ready_with_ecc)(uint32_t block,uint32_t offset,
                                          uint8_t *buf,uint32_t size);
bool_t     (*query_block_ready_no_ecc)(uint32_t block,uint32_t offset,
                                          uint8_t *buf,uint32_t size);
//在已知存储器中原来的内容的条件下，检查写入规定的内容时，是否需要擦除。
//new_data==NULL 则检查从 org_data 改为任何数据时是否被擦除,没有带 ecc 版本。
bool_t     (*query_ready_with_data)(uint8_t *new_data,uint8_t *org_data,
                                                uint32_t size);
//返回: true = 成功擦除，false = 坏块
bool_t (*erase_block)(uint32_t block);
bool_t     (*check_block)(uint32_t Block);   //返回: true=好块，false=坏块。

//读一块之内的任意长度数据，返回校验过的正确数据
//返回：cn_limit_uint32=参数错误，其他值：实际写入数量
uint32_t (*read_data_with_ecc)(uint32_t block,uint32_t offset,
                               uint8_t *buf,uint32_t size);
//写入一块之内的任意长度数据，校验方法取决于具体器件，无校验也是一种可能。
//返回：cn_limit_uint32=参数错误，其他值：实际写入数量
uint32_t (*write_data_with_ecc)(uint32_t block,uint32_t offset,
                               uint8_t *buf,uint32_t size);
//不带校验的读函数，返回 true = 成功，false = 失败
uint32_t (*read_data_no_ecc)(uint32_t block,uint32_t offset,
                               uint8_t *buf,uint32_t size);
//返回：cn_limit_uint32=参数错误，其他值：实际写入数量
uint32_t (*write_data_no_ecc)(uint32_t block,uint32_t offset,
                               uint8_t *buf,uint32_t size);
//写掉电保护块,PCR_block 为保存掉电保护信息的块号，protected_block 是被保护的
//块号，均是芯片的绝对块号。
//返回: true = 成功，false = 写入错误 PCRB_block 是坏块
bool_t (*write_PCRB)(uint32_t PCRB_block,
                              uint32_t protected_block,uint8_t *buf);
//把掉电恢复块的数据恢复到目标块，这样可以使掉电恢复算法完全由芯片驱动提供
//PCRB_block 是掉电恢复块的绝对块号。
```

```
    //返回: true=成功恢复或无需恢复，false=恢复失败
    bool_t (*restore_PCRB)(uint32_t PCRB_block,uint32_t *restored);
};
struct st_DBX_flash_tag
{
    struct flash_chip *chip;      //文件柜所属芯片
    bool_t     nand_ecc;          //是否做 ecc 校验，一般来说，nand 类型 flash 做多媒体
                                  //存储器时，可不做 ecc 校验。
    uint32_t DBX_no;              //在该芯片中的文件柜编号
    uint32_t start_block;         //起始块号，相对于芯片的绝对块号
    uint32_t block_sum;           //总块数
    uint32_t valid_sum;           //有效块数
    uint32_t free_sum;            //空闲块数
    uint32_t PCRB_no;             //掉电恢复块的块号，相对于文件柜的相对块号
    uint32_t DDR_main;            //首个 DDR 表块号，相对于文件柜的相对块号
    uint32_t DDR_bak;             //第二个 DDR 表块号，相对于文件柜的相对块号
    uint32_t ero;                 //文件柜状态
    uint32_t FDT_capacity;        //已经分配的 FDT 块的 FDT_item 容量
    uint32_t FDT_free_num;        //已经使用的 FDT 块数量，这两个参数用于启动 FDT 释放
    uint32_t DDR_size;            //DDR 表尺寸
    uint32_t *DDR_DBL;            //DBL 表，先 main 后 bak
    uint32_t DDR_ARB;             //磨损次数基数，加磨损次数表(16bit)得实际磨损次数
    uint32_t DDR_FDSB;            //目录表首块块号
    struct MAT_table *DDR_MAT;                  //MAT 表
    //主 DDR 表写入标志数组，true=已写入 flash，false=未写，数组大小与 DDR 表占用的
    //块数相同，主备各一个。
    bool_t    *writed_DDR_main;
    bool_t    *writed_DDR_bak;
    //以下两个表，在分配内存时须按编译器要求保证边界对齐(使用 djyos 动态内存分配
    //可以保证)
    uint32_t *ART_position;          //磨损位置表，该块在磨损顺序表中的位置，
    uint32_t *ART_block_no; //磨损块号表，与次数表同列，表明次数表中对应项的块号
    uint16_t *ART_times;      //磨损次数表，忙块和闲块分别按磨损次数排序
    uint32_t balance_region[19];//磨损分区表，忙块闲块各 9 个分区，最后是坏块分区
};
```

13.3 DFFSD 初始化与挂载芯片

和所有模块一样，DFFSD 模块在使用前，必须先初始化，module_init_DFFSD 函数执行初始化工作。该函数非常简单，仅仅在系统资源链表中增加了一个名叫“flash chip”的根结点。系统中所有 flash 芯片均挂载在该结点下，注意，该结点的数据类型是 struct rsc_node，它的子结点——具体芯片的数据类型是 struct flash_chip。

代码 13-2 初始化 DFFSD 模块

```
static struct rsc_node tg_flash_chip_root_rsc;
bool_t module_init_DFFSD(void)
{
    //在资源链表中建立一个根结点，系统中所有芯片都挂在这个结点下
    rsc_add_root(&tg_flash_chip_root_rsc,sizeof(struct  rsc_node),"flash chip");
    return true;
}
```

要把一个flash芯片接入文件系统，就必须先挂载芯片，图 13-5 所示的是挂载芯片的简化过程。由于涉及到在异常掉电、芯片有瑕点下的可靠存储，以及磨损平衡算法，使加载的过程很复杂，DFFSD_install_chip函数远不是 图 13-5 显示的那么简单，在这里只是建立一个对加载过程的总体印象，详细的算法将在后面章节逐步展开。

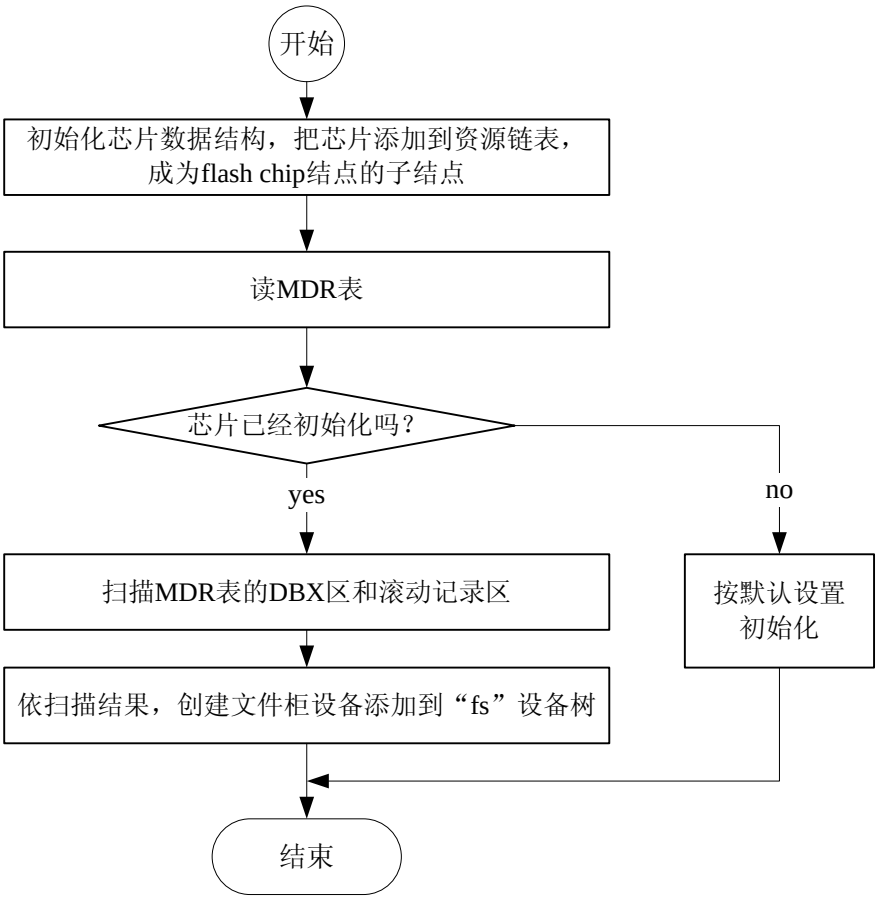


图 13-5 挂载芯片流程

13.4 掉电恢复块

13.5 磨损平衡算法

flash 是一种特殊的存储器，它要先擦除后才能写入，擦数是以块为单位进行的，读操作则可以在随机地址读，但是写操作只能在已经擦除的区域进行。flash 的擦写次数也是有限的，一般 nor flash 不能超过 10 万次，nand flash 不能超过 100 万次。虽然没有官方确认，但根据笔者经验，再用几个不同型号的 flash 芯片验证，目前市面上几乎所有的 flash 芯片，

其擦写特性都可以归结为：

1. 擦除操作的结果是把整块所有的位恢复为 1，包括 nand flash 的 OOB 块，如果擦除后不是 1，则说明它是一个坏块。根据笔者经验，当一个块磨损过度变成坏块时，首先表现为不能擦除干净。
2. 写入操作可以在已擦除块内的随机地址进行，但不能重复进行。写入可以把任意为 1 的位变为 0，但不能把为 0 的位变为 1。只有擦除操作才能把 0 变为 1，即使你只是想把一块中的一位由 0 变为 1，也只能使用整块擦除操作。所以，所谓写入操作不能重复进行，指的是不能把一个位在 0 和 1 之间反复改变。对一个区域，第一次写入把部分位改为 0，第二次写再把另外一部分位改为 0，则是允许的，而且，只要没有把任何位从 0 改为 1 的操作，都是允许的。
3. 每个块有独立的寿命周期，寿命由擦除次数决定。如果某一块过度使用而损坏，不会影响到其他块的正常操作，芯片的电气性能和读、擦、写速度均不受影响。
4. nand flash 出厂时就可能有坏块，在使用过程中可能还会产生坏块（非因整片寿终），但保证首块不是坏块，首块只有过度磨损才会损坏。nor flash 则保证所有块都能达到手册声称的寿命周期。

因此，在 flash 上实现文件系统需要磨损平衡管理，否则会出现部分块过度使用而损坏，部分块则远远没有达到寿命周期的情况。DFFSD 磨损平衡算法可保证各块的磨损次数相差不超过 1024 次，为什么是 1024 次而不是保证绝对平衡呢？因为保证绝对平衡，一是会导致频繁的块交换，严重地影响文件系统的性能和 flash 的寿命，二是磨损次数表要保存在 flash 中，为了防止嵌入式系统突然掉电丢失数据，必须在任意块的磨损次数改变时，都要写入 flash，极大地降低了文件系统的性能，加速 flash 的磨损。如果允许一定程度的不平衡，可以极大地降低块交换几率，降低磨损次数表的改写次数，提高文件系统的性能和 flash 的寿命。相对于 flash 的寿命来说，1024 次磨损次数差是微不足道的，它仅仅是 nor flash 寿命的 1%，nand flash 寿命的 0.1%。

13.5.1 数据结构设计

在 图 13-2 所示的文件柜布局图中显示，DDR表的ARB和ART区域保存了文件柜的所有块的磨损次数，系统启动时，磨损平衡管理器将扫描ART表，建立在内存中的数据结构映像，内存中的磨损次数表要比flash中保存的要复杂得多。在flash中，由于寿命有限和整块擦除特性，且擦写操作很慢，要考虑的是避免频繁写入。再者，磨损次数表并不需要绝对准确，所以磨损次数表的修改次数积累到一定程度后，才一次性写入flash中是允许的。

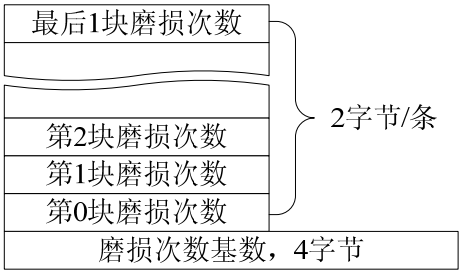


图 13-6 flash 中的磨损次数表结构

内存中的磨损次数表将用于：

1. 收集每次磨损次数改变的数据，当积累到一定程度时，写入 flash 中。
2. 文件操作需要分配一个新存储块时，通过内存中的磨损平衡表查找磨损程度最轻的空闲块。

3. 某块被改写时，其磨损次数将增加，要根据磨损平衡表判断其磨损次数是否越限，如果越限则把内容交换到磨损程度最轻的块上，并让原块“休息”。

如果内存中直接使用跟flash中一样的磨损次数表，则每次申请新存储块，都要扫描整个磨损次数表，才能找到最轻磨损的块，这是一个时间复杂度为O(n)的操作，它所需要的执行时间随着文件柜尺寸（用块数表示）线性增加。如果你不能确定文件柜的尺寸，也就不能估算分配一个存储块需要多少时间，这在实时系统中应该是尽量避免的。如 图 13-7 所示，在内存中，有 3 个表用于磨损平衡管理，分别是位置表、块号表、磨损次数表，位置表记录各块在磨损次数表中的偏移量，磨损次数表则记录各块的磨损次数，而块号表是各块的块号按该块的磨损次数排序的一个表。其中位置表用于在知道块号的情况下，查看该块的擦除次数以及其在块号表中的排列位置；块号表用于确定块号表中某排列位置的块的块号；磨损次数表用于确定某块的磨损程度。磨损次数表每条记录 2 字节，位置表和块号表每条记录 4 字节。

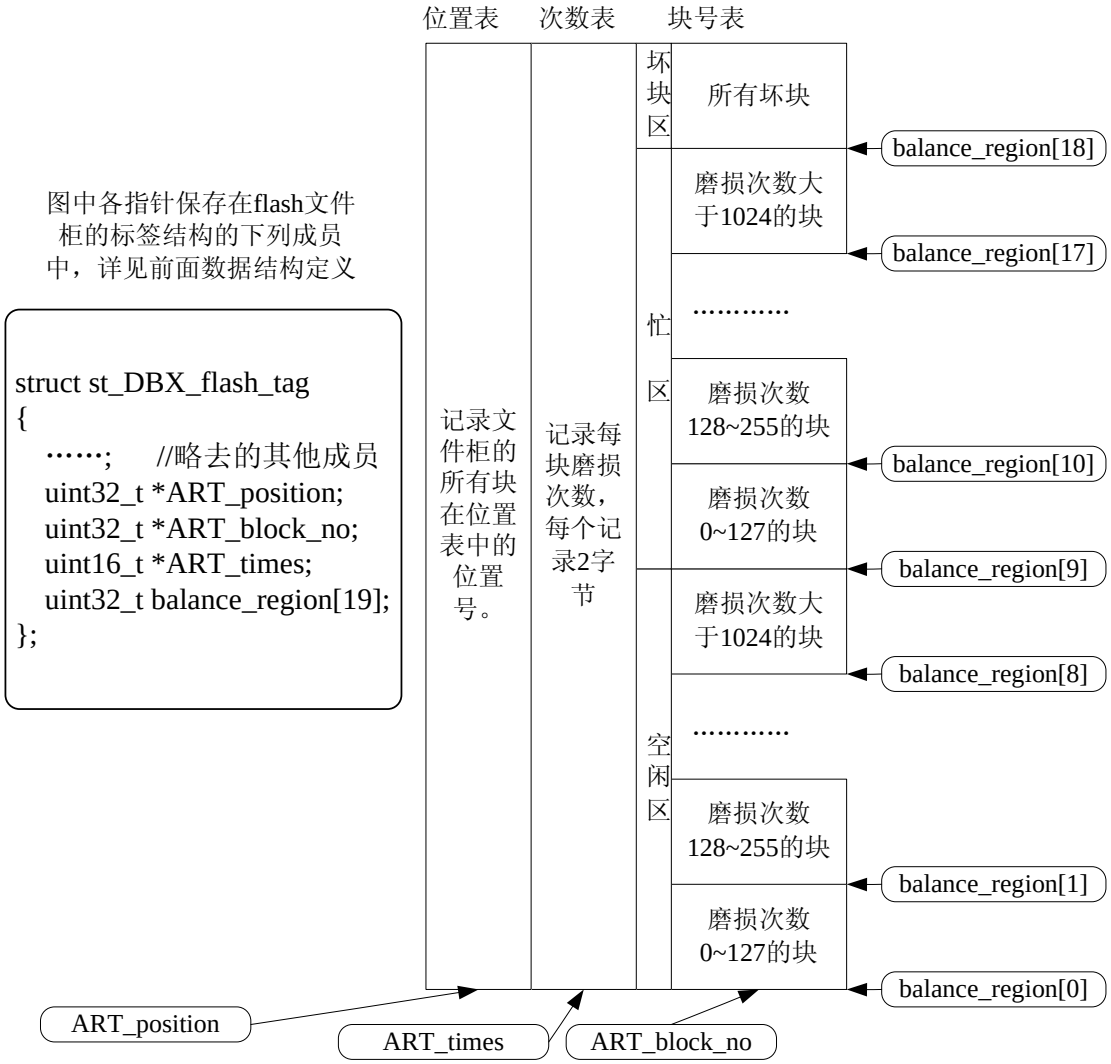


图 13-7 内存中的磨损次数表结构

图的说明：

1. 磨损次数表分忙块区、空闲块区、坏块区三个大区，其中忙和空闲块区又按磨损程度各分 8 个磨损区，按顺序排列。
2. balance_region 表中保存的是各分区首块相对于指针 ART_block_no 的偏移量，偏移量的计算标准是以 0 起计的块数，balance_region[0]肯定是 0，如果 balance_region[0]

区有 10 块（即磨损次数在 0~127 之间的块），则 balance_region[1]=10，类推之。

3. 磨损次数表与块号表并列，而不是与位置表并列。如果文件柜中第 8 块的磨损次数是 500，在块号表中排在第 20 位，则有 ART_times[20]=8 而不是 ART_times[8]=500。

磨损平衡表的排序并不是绝对排序，而是把空闲块和忙块按每 128 次为一组分组排序，磨损次数 0~127 次为一组，128~255 次为一组，依次类推，896~1023 次为一组，超过 1024 次为一组，坏块集中为一组，组内不排序。这样分组后，大大简化了磨损平衡表的管理，提高了执行效率，表现在以下几个方面：

1. 磨损次数表重新排序的几率降低到 1/128，如果某块的磨损次数增加，只要没有跨过组边界，就无需重新排序。
2. 加快分配和释放一块的时间，不分组的话，新分配一块时，最年轻的空闲块将从闲区转移到忙区，那么整个空闲表要逐条记录下移一格，如果文件柜很大，块数很多，是相当消耗时间的，分组排序后，使用下一节的跳跃换区法，移动的记录数不会超过 18 条。释放一块的过程也一样。

如果说 图 13-7 还比较抽象难于理解的话，图 13-8 所示的范例将帮助你理解。该图描述了一个只拥有 20 个存储块的文件柜的磨损次数表，

表格 13-1 磨损表实例

块号	磨损次数	状态
0	370	忙
1	580	忙
2	160	空闲
3	100	坏块
4	250	空闲
5	700	空闲
6	10	忙
7	755	空闲
8	120	忙
9	300	忙
10	58	忙
11	200	空闲
12	700	空闲
13	330	忙
14	522	忙
15	680	空闲
16	1024	忙
17	560	忙
18	900	空闲
19	1000	空闲

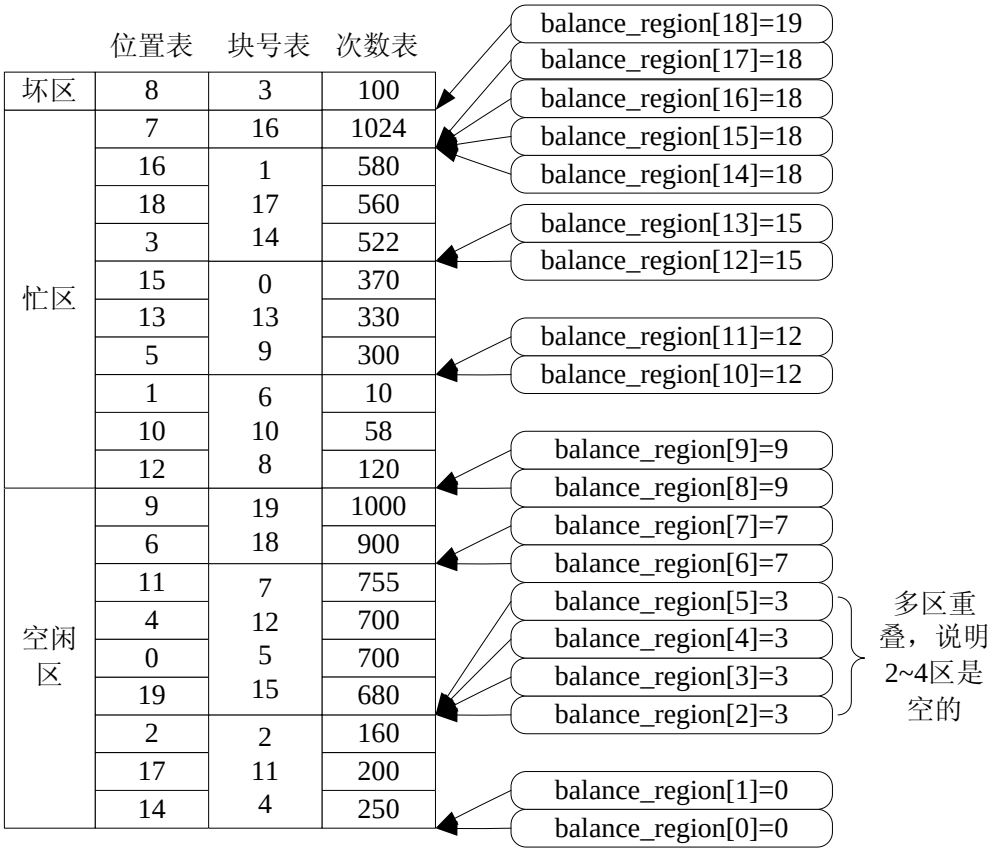


图 13-8 磨损表实例

13.5.2 跳跃换区法

上一节讲到，磨损平衡表是以每 128 次为一个区域分区管理的，当某一块从一个区转移到另一个区时，就需要使用跳跃换区。在下列情况下，需换区：

1. 某块磨损次数增加使其跨入另一个区，该块将转移到邻近的高一区。
2. 释放块将使该块从忙区转移到空闲区，该块将被转移到比原来低 9 区的位置。
3. 分配块将使一块从空闲区转移到忙区，该块将被转移到比原来高 9 区的位置。空闲变忙要增加磨损次数，如果因此而使磨损次数跨越分区线，则被移动到比原来高 10 区的位置。
4. 发现坏块时，将使该块转移到坏块区。

图 13-9 是一个典型的跳跃换区过程，它描述了一个原来位于第n区的块，转移到第n+3 区的过程。__DFFSD_jump_region函数实现跳跃换区，详见随书代码。

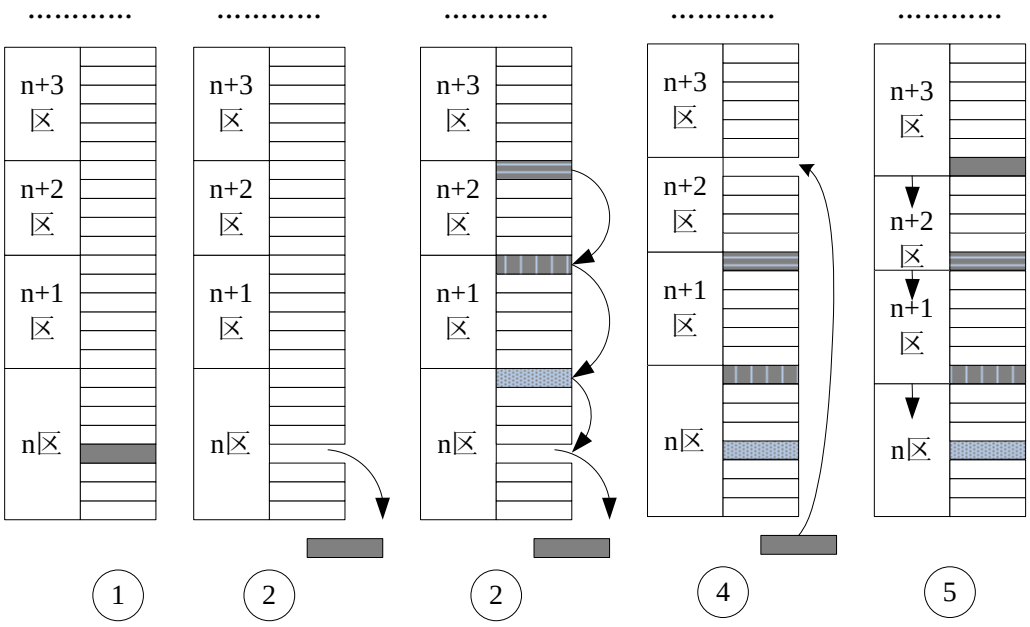


图 13-9 跳跃换区过程

图解：

1. 磨损平衡表原来的样子，灰色条目即将被调整到 n+3 区。
2. 先把目标块取出备份。
3. 把第 n 区的最后一块转移到目标块原位置，该位置便成了空位，n+1 块开始，逐区把该区的最后一块转移到前一个区先前空出的位置。
4. 把备份的目标块拷贝到 n+2 区的空位。
5. 从当前区开始，逐区边界下移，得到⑤的最终结果。

13.5.3 分配一块

13.6 对 djyfs 的接口函数

DFFSD需要实现第 12.9 节所述的所有接口函数。

13.7 芯片驱动接口

关于 mark_invalid_block 函数由谁调用的问题，在设计 djyffs 时也出现过反复。有两种方案可供选择，一是把标记坏块纯粹作为特定存储介质的特性，在芯片 driver 中完成，即当芯片读写、擦除等操作发现某一块是坏块时，调用该函数直接标记之；另一种选择是，由 djyffs 决定，芯片 driver 提供一个标记函数。

13.8 nand flash 芯片范例：2808U0B

13.9 nor flash 芯片范例：SST39VF1601

第14章　其他文件系统驱动程序

14.1 RAM 文件系统驱动程序

14.2 串口文件系统驱动程序

14.3 简易文件系统驱动程序

这是在 AT24C128 上实现的只读文件系统，简称 EFS，AT24C128 是一个 IIC 接口的 EEPROM，容量为 16Kbytes。通过 EFS 可以把 AT24C128 当作一个文件柜，按读文件的方式来从 AT24C128 中读取数据。EFS 相当简单，它只支持了 djyfs_fopen、djyfs_fread、djyfs_fsearch 和 djyfs_seek 四个系统调用，其他系统调用将是空操作。那么，EFS 有什么用呢？从孤立的一个产品来看，没什么用，徒然使软件变得更复杂，消耗更多的资源。但如果站得高一点，从软件的可靠性、兼容性和可移植性来看，无疑使应用程序有更高的质量！有许多产品用小容量的串行 EEPROM 来配置产品的功能和参数，而这些参数是产品出厂时统一烧录的，或者在使用过程中由专用工具配置的。如果不使用文件接口来访问 EEPROM，用户就必须直接调用读写 AT24C128 的函数访问 EEPROM，这样会产生下列问题：

1. 这样的小型存储器种类繁多，当芯片型号更替时，需要修改芯片驱动程序，由于应用程序直接调用芯片驱动程序，可能导致应用程序也需要修改，至少也需要重新编译包括应用程序在内的整个软件。
2. 如果某企业有丰富的产品线，其中许多软件模块需要跨产品共享，而这些共享模块的产品可能使用不同的存储芯片，就会导致这些共享模块在不同产品中不一致。当发现 bug 或者功能升级而需要修改代码时，这些不一致的、功能相同的模块需要逐一修改，这种修改很容易由于人为出错而导致安全隐患，即使重新测试也不可能完全消除隐患。

而如果使用文件系统来读取配置，由于文件系统是一个接口统一、可以独立编译和加载的模块，当存储器芯片发生变化时，只需要重新编写和加载文件系统就可以了。应用程序可以在不同产品之间移植而无需任何修改，甚至无需重新编译，由于无需修改，故不会产生新的 bug，提高可移植性的同时提高了可靠性。新的文件系统 dfsmd 模块可以独立测试，因无需测试应用程序部分而减少了工作量，同时还提高了开发工作的并行度，进一步提高了研发效率。

第15章　djyos 移植（本章未完成）

15.1　可移植性的考虑

都江堰操作系统作为一个操作系统，当然要考虑被移植到其他 CPU 平台上运行，系统在设计过程中，就充分考虑到移植的可行性和便利性。可移植性主要考虑两个方面，一是操作系统本身的可移植性，二是考虑应用程序迁移的方便性。

我们首先要看到，开发高可靠性的、含嵌入式系统的产品，软件测试非常重要，把操作系统从一个 CPU 平台迁移到另一个 CPU 平台，需要做非常多的测试工作，测试的工作量远远多于修改程序文件的工作量。而测试的工作量又和软件中的 bugs 数量成比例的，bugs 数量越多，则“测试——修改——再测试”的循环就越多。因此，djyos 在考虑可移植性方面，主要着力于协助程序员减少 bugs 数量，并不以减少需要修改的代码行数和函数个数以及文件数为目标，不用高度技巧化的手段降低修改程序文件的工作量。

15.1.1 充分注释

笔者认为，在充分理解代码的基础上进行移植，是减少移植 bugs 的有效手段。djyos 几乎所有代码都是自注释的，此外还有大量的文字注释，把每个函数、每段代码的含义和相关的上下文均解释清楚，特别是在移植时需要修改的函数、代码、定义、配置，均注明“移植关键”字样。下面是一个平台相关的 y_get_fine_time 函数的书写示例：

```
//----读取精密时间----------------------------------------------------------
//功能：读取操作系统精密时钟，该时钟是一个自由运行的定时器，表示从上次 tick 中断开
//          始至今流逝的时间，时间粒度在配置文件 port_kernel.h 中设置，与硬件相关，一般
//          设为系统 tick 的百分之一或者千分之一，S3C44B0X 中设为千分之一.
//输入：无
//返回：精密时钟，不包含 ticks，如果用户需要得到完整时钟还需要调用 y_get_time
//备注: 本函数是移植关键函数.
//-----------------------------------------------------------------------------
uint32_t y_get_fine_time(void)
{
    return 1000-pg_timer_reg->TCNTO5;
}
```

15.1.2 平台无关

充分考虑了 C 语言的平台相关性，避免使用 C 语言与平台相关的特性。C 语言有许多行为是没有明确定义的，相同的代码，在不同的硬件平台上，或者相同的硬件平台但不同的编译器上，运行的结果可能不相同。为了避免这样的结果，有许多代码做了相应的调整，其结果是，执行效率可能会降低，产生的目标代码可能更大。对此，在源代码中都有相关注释

说明。

例如，在 flash 文件系统中，把本地文件属性拷贝到文件存储存储介质中时，用下列代码逐位拷贝：

```
attr_mirror = 0;
if(attr.bits.read_only)
    attr_mirror +=cn_FS_ATTR_READ_ONLY;
if(attr.bits.hidden)
    attr_mirror +=cn_FS_ATTR_HIDDEN;
if(attr.bits.folder)
    attr_mirror +=cn_FS_ATTR_DIRECTORY;
if(attr.bits.archive)
    attr_mirror +=cn_FS_ATTR_ARCHIVE;
FDT_item->mattr = attr_mirror;
```

而不是用效率最高的单条语句一次性拷贝：

```
FDT_item->mattr = attr.all;
```

这是因为 attr 结构是本地数据结构，其位域成员的排列是由目标机编译器决定的，不同的开发平台可能不一致，而保存在文件存储介质中的文件属性的格式是固定的，在所有平台上都是固定的，它的格式可能与你正在使用的编译器不一致。

15.1.3 严格遵守 ANSI C

djyos 虽然用 gcc 开发，但是没有使用任何 gcc 对 C 语言的扩展，完全遵守 ANSI C（对应的国际标准是 ISO/IEC 9899：1994）。即使是从 linux 中引进的 container_of 宏，也被修改成符合 ANSI C 标准的格式。

linux 中 container_of 宏的定义：

```
#define container_of(ptr, type, member) ({              \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
        (type *)( (char *)__mptr - offsetof(type,member) );})
```

由于 typeof 是 gcc 的扩展，ANSI C 不支持，在 djyos 中，container_of 宏被修改成：

```
#define container_of(container,ptr, type, member) do{               \
        const void *__mptr = (void*)(ptr);\
        container = (type *)( (char *)__mptr - offsetof(type,member) );}while(0)
```

这样修改以后，使用起来虽然稍微麻烦一点，但是可以在任何支持 ANSI C 的编译器上编译通过。

15.1.4 慎用汇编

操作系统需要控制计算机的所有资源，从复位后的第一条指令开始就要控制计算机，有部分代码是必须用汇编实现的。一般来说，可能需要用汇编实现的部分有下列几个方面，需要注意的是，这只是可能性而已，在有些平台上需要用汇编实现的东西，在另一些平台上可能用 C 语言就直接可以实现。为此，djyos 中凡是用汇编实现的部分，都会有更详细的注释说明，注明为什么用汇编实现，以及相应的 C 伪代码，方便大家移植。

1、 启动代码

从 CPU 上电（复位）后，设置 CPU 工作方式，准备 C 语言程序的执行环境，然后转向 C 代码的部分，称作 CPU 的启动代码。这些代码可能需要操作 CPU 的专有寄存器，或者把特定的指令放置到特定的位置，需要用汇编语言才能实现。

启动代码不但与 CPU 有关系，还与编译器和连接器有关系，典型的是存储器初始化部分，有些编译/连接器能够输出各段的起始和结束地址并让 C 语言捕获，这样就允许用 C 语言初始化，比如 gcc；还有一些编译/链接器不提供这样的功能，就无法用 C 代码去初始化 C 语言的运行环境，而必须使用汇编。

细心的读者可能会问，既然启动代码的功能之一就是“准备 C 语言程序的执行环境”，那就是说，启动代码运行的时候，C 语言的运行环境还没有建立起来，为什么还可能会允许使用 C 语言编写启动代码呢？的确，此时的 C 语言代码是有限制的，这种限制表现在：

a) 只能调用启动代码模块的函数，不能调用 C 库函数。

b) 不能使用定义时初始化的变量，例如代码：

```
uint32_t    aa = 100;
```

是不允许的，而是应该写成：

```
uint32_t    aa;
aa = 100;
```

2、 上下文切换

如果说启动代码还有可能用 C 实现的话，上下文切换是不折不扣的汇编语言的天下，上下文切换要操作所有 CPU 寄存器、栈指针、程序指针，是必须使用汇编才能实现的。

3、 部分异常和中断控制

当中断发生转入中断向量后的最初一段代码，往往要用汇编才能控制。

15.1.5 便于应用程序迁移

在为设计产品准备操作系统时，一般会参考一个已经移植好的操作系统，再根据项目硬件平台特征做少量的调整，只需要很少的工作。即使没有现成的移植可以参考，把操作系统移植到一个全新的硬件平台上，移植操作系统上的工作量，与应用程序的工作量比起来，还是只占极少的部分。对一个企业来说，硬件平台一般都会比较固定，操作系统只需要移植一次，经过充分测试后即可成为可靠的软件模块存档，切换产品时，一般更新配置即可。一次移植，终身使用，因此，即使在移植操作系统上的工作量增加一些，但这是一次性的工作，对一个企业或者开发团队来说，移植操作系统所花费的时间占总开发时间的比例很小。

对于一个操作系统来说，大量的嵌入式最终产品才是最重要的，如果不应用于产品，操作系统本身毫无意义，产品也只有可靠的产品才有意义。产品的可靠性由可靠的应用程序和可靠的操作系统共同组成，应用程序的 bug 只会存在单个产品中，而操作系统的 bug 却是潜藏在所有产品中的定时炸弹，而且是最顽固的定时炸弹，因此，当可靠性与可移植性发生矛盾时，djyos 系统优先保证可靠性。

不要为了兼容更多的平台或者提高效率而使代码晦涩难懂，妨碍用户阅读，使用户难于充分掌握操作系统。过分地强调兼容性和代码效率，你可能需要用大量的#if 来控制条件编译，使大量跟你的实际系统不相关的代码充斥在你的源码文件中，你还可能需要使用层层嵌套的宏定义，阅读时便需要层层展开，要看到代码的真是面目非常不容易。这真的很可怕，二战时美军训练印第安人充当通信兵，使用的全是明码语音通信，效率极高，译码的时间都省了，日本人虽然轻而易举地截获了电码，却如同天书一样无法破译。过多使用宏嵌套的代码就像印第安语电码一样，代码可以轻易得到，可读懂它就不是件容易的事情了。djyos 系

统约定，除了一些简单的类型定义外，包含代码的宏嵌套最多两级；条件编译嵌套最多两级，并且严格限制#if 和#endif 之间语句的行数。

15.2 需要移植的部分

15.2.1 CPU 初始化文件

初始化文件的写法既与 CPU 有关，又与开发工具有关，不同的平台差异很大，是启动代码的核心文件。在大多数平台上，初始化文件必须使用汇编语言编写，或者汇编和 C 相结合的方式。在 S3C44B0X 版本上，全部 CPU 初始化代码在 initcpu.s 文件中。

CPU 初始化一般要完成一下步骤（不是所有硬件平台都需要所有步骤）：

1、 在复位向量、异常、中断向量处放置跳转语句，使 CPU 在复位或者异常或者中断发生时能够跳转到相应的位置执行程序。
2、 设置 CPU 的工作模式，关闭中断。有许多 CPU 区分用户态和多种不同权限的特权状态，在这里必须设置为最高特权级。
3、 时钟设置，主要是设置 CPU 的工作频率。至此，存储器总线还没有配置好，并不是所有平台都配置有无需配置直接可以访问的存储器，故不可使用 RAM，也就是说不能使用变量。
4、 配置存储系统，包括存储器总线参数设置，配置 cache 和 MMU，映射存储器地址，完成后就可以使用变量了。
5、 如果有硬件看门狗，关闭之。
6、 初始化栈指针，由于需要在栈中保存返回地址和传递参数，以及保存局部变量，故在本步骤之前，不可调用函数。在确保不使用栈的情况下，可以调用汇编函数；对于 C 函数，即使你知道不会使用栈，也不要使用函数，因为是否使用栈，取决于编译器，而你不可能了解所有不同编译器的全部行为，使你的代码的功能独立实现，不依赖于某些你不了解或者不能控制的条件，是一个良好的习惯。
7、 跳转到加载程序，加载程序将加载操作系统到内存中，至此，CPU 初始化完成。

15.2.2 配置文件

移植操作系统时与内核相关的参数全部在port_kernel.h文件中，需要配置的硬件参数包括CPU字长、编译器字长，CPU主频，存储器的大小端，CPU访问存储器要求的对齐方式等。软件参数配置包括事件类型数量、事件数量等参数。这些常量大部分与CPU和编译系统有关，也有少量纯粹是设定操作系统工作模式的，如 表 15-1 所示。

表 15-1 port_kernel.h 中需配置的常量

常量	说明
以下常量与 CPU 和编译环境有关	
cn_mclk	cpu 主频
cn_cpu_endian	存储器大小端配置，与 CPU 有关，有些 CPU 大小端都支持，cn_cpu_endian 须与编译系统关于大小端的选项一致，可选值有： #define cn_little_endian 0 #define cn_big_endian 1

cn_cpu_bits	CPU 字长，即 ALU 的字长
cn_point_bits	指针位数，一般等于 CPU 字长，但有例外，尤其是 16 位 CPU。
cn_cpu_bits_suffix_zero	CPU 字长 2 进制表示的后缀 0 个数，如 32 的二进制为 00100000B，后面是 5 个 0，本参数用与存储器管理模块
cn_char_bytes	字符长度（8 的倍数）,字符是最基本的存储单元，大多数 CPU 的字符是 8 位的，也有一些是 16 位的（比如 TMS320F28xx 系列），也有一些 CPU 的字符是 32 位的（如 TMS320C33），这个常量对编写可在不同 CPU 间移植的软件非常重要。
uint64_t/sint64_t	64 位无/有符号整数类型定义
uint32_t/sint32_t	32 位无/有符号整数类型定义
uint16_t/sint16_t	16 位无/有符号整数类型定义
uint8_t/sint8_t	8 位无/有符号整数类型定义
ucpu_t/scpu_t	无/有符号cpu字长类型，cpu可以用一条指令读取或写入ucpu_t类型的数据，可以保证“树状共享变量”（参见 16.7 节）的数据完整性，也是操作速度最快的一种数据类型。
ptu32_t	可转换为指针的、长度最少为 32 位的无符号整数，与 cpu 的寻址范围有关，也与 CPU 的结构有关，如果该系统中指针长度小于或等于 32 位，则 ptu32_t 被定义为 32 位，否则与指针等长，它的用途有二：一是用于内存分配的指针限定,或者用于描述内存长度；二是用于可能需要转换成指针使用的整形数据，比如 struct pan_device 结构中的 ptu32_t private_tag 成员。
cn_stack_type	CPU 系统的栈类型常数，可选值为以下 4 个： #define cn_full_down_stack 0 //向下生长的满栈 #define cn_empty_down_stack 1 //向下生长的空栈 #define cn_full_up_stack 2 //向上生长的满栈 #define cn_empty_up_stack 3 //向上生长的空栈 栈的类型相对于编译器是透明的，它只与 CPU 相关，与编译器无关。

align_down(x)/
align_up(x)
	内存对齐设定，up 和 down 各有 4 个可选项：
	#define align_down_2(x) ((x)&(~1)) //2 字节,向下对齐
	#define align_down_4(x) ((x)&(~3)) //4 字节,向下对齐
	#define align_down_8(x) ((x)&(~7)) //8 字节,向下对齐
	#define align_down_16(x) ((x)&(~15)) //16 字节,向下对齐
	#define align_up_2(x) (((x)+1)&(~1)) //2 字节,向上对齐
	#define align_up_4(x) (((x)+3)&(~3)) //4 字节,向上对齐
	#define align_up_8(x) (((x)+7)&(~7)) //8 字节,向上对齐
	#define align_up_16(x) (((x)+15)&(~15)) //16 字节,向上对齐

cn_int_num	目标系统中断线的数量
下面这些常数控制操作系统的工作参数，与 CPU 和编译工具无关	
cn_tick_ms	操作系统内核时钟脉冲长度，以毫秒为单位。
cn_tick_hz	内核时钟频率，单位为 hz。
cn_fine_us	操作系统内核精密时钟脉冲长度，以微秒为单位。它表示从上一次 tick 脉冲开始至今流逝的时间量。
cn_fine_hz	内核精密时钟频率，是 cn_fine_us 的倒数。
	以上几个常数要与硬件相匹配，也要与 CPU 的运算能力、应用软件要

	求的定时精度有关，绝对不能超过硬件定时器的能力，原则上，CPU 的速度快、或者定时精度要求高的可以把 tick 间隔定得小一些，但这样会增加 CPU 的负担。
cn_run_mode	cpu 运行模式定义，可选值有： #define cn_mode_si 0 //单映像模式 #define cn_mode_dlsp 1 //动态加载单进程模式 #define cn_mode_mp 2 //多进程模式，只有 mmu=true 才可//选择此模式
cn_device_limit	允许用户创建的设备数量。
cn_handle_limit	允许打开的设备数量
cn_locks_limit	定义程序中可用的锁的数量。指用户调用 semp_creat 和 mutex_create 创建的锁，不包括嵌入到设备中的锁，也不包括操作系统自身使用的锁。
cn_events_limit	事件控制块数量
cn_evtts_limit	事件类型控制块数量
cn_wdt_limi	允许养狗数量
下列常量用户内存管理	
cn_page_size	页尺寸（若用 MMU，必须与 MMU 一致）
cn_block_limit	允许一次分配的最大内存块尺寸
cn_mem_recycle	是否支持内存回收
cn_mem_pools	允许创建由固定块策略管理的内存池的数量

15.2.3 makefile 连接脚本文件

makefile 是什么？有许多习惯了想 keil、iar、ARM realview 等 IDE（集成开发环境）或者开发 PC 软件的人，可能不知道 makefile 是什么。他们习惯了“创建工程——简单配置——编译（甚至不知道有连接）——调试——产生烧录文件”这样的过程，IDE 为你做了其他的一切。但是要开发操作系统，我们需要精确地控制编译和连接过程，这些更高阶的控制，IDE 的傻瓜化的工具就显得力不从心了，就只能依靠 makefile 文件了。简单地说，makefile 是告诉 make 命令需要怎么样的去编译和链接程序的文本文件。makefile 定义了一系列的规则来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，还能独立地控制每个文件的编译选项，以及一些更智能更复杂的功能。把 makefile 文件编写好以后，就可以用 make 命令全自动化地完成整个工程的编译和连接了。djyos 使用 gnu 的 gcc 编译器，关于 make 和 makefile 的更进一步的知识，请读者参考 gnu 的相关文档。在随书的工程代码中，可以找到 makefile 文件和连接脚本文件，在这里就不列出其内容了。

程序编译后，还要通过连接才能生成可执行的文件，连接脚本文件用于引导连接程序如何确定代码的存储地址和运行地址。它告诉链接器，目标板的 rom 和 ram 的起始地址和尺寸，它允许定义不连续的多块 rom 和 ram，哪些代码需要连接，代码在 rom 中的存储位置和运行时地址，每段代码都可以单独指定，以及运行时使用的内存地址。另外，连接脚本文件还能输出程序的定位信息，在 C 源程序中作为外部符号引用，以完成程序加载功能。在 djyos 的初始化代码中，用下列语句获取预加载器的加载和执行地址：

```
extern ucpu_t text_preload_load_start[];
```

```
extern ucpu_t text_preload_run_start[];
extern ucpu_t text_preload_run_limit[];
```

其中 text_preload_load_start 等三个符号是在连接脚本文件中定义的，下列语句用于把预加载器的代码段从加载地址考北到执行地址

```
if(text_preload_run_start != text_preload_load_start)          //拷贝代码段
{
    for(src=text_preload_load_start,des=text_preload_run_start;
                                                des<text_preload_run_limit;src++,des++)
        *des=*src;
}
```

15.2.4 线程相关函数

需要移植的线程相关函数包括：

15.2.4.1 创建线程虚拟机

原型：struct thread_vm *__create_thread(struct event_type *evtt_id)

说明：创建一个线程虚拟机用于处理事件类型 id 号为 evtt_id 的事件，创建虚拟机所需要的参数：入口函数和栈尺寸都从事件类型控制块中获取，因此调用本函数前，必须正确初始化事件类型控制块。在 cpu.c 文件中可以找到本函数。

移植要点：在 代码 15-1 中注释 1 和 2 两行是需要移植的。

代码 15-1 __create_thread 函数需移植部分

```
struct thread_vm *__create_thread(struct event_type *evtt_id)
{
    ……
    result=(struct thread_vm *)m_malloc_gbl(len);                    //1
    ……
    result->stack_top = (uint32_t*)((ptu32_t)result+len);            //2
    ……
}
```

这两行代码与满栈还是空栈，向上生长还是向下生长有关，上述代码用于向下生长满栈的情况，其他情况应分别改为：

1. 向下生长空栈：

```
result=(struct thread_vm *)m_malloc_gbl(len);
result->stack_top = (uint32_t*)((ptu32_t)result + len) - 1;
```

2. 向上生长满栈：

```
result=(struct thread_vm *)m_malloc_gbl(len) + len – sizeof(struct thread_vm);
result->stack_top = (uint32_t*)((ptu32_t)result) - 1;
```

3. 向上生长空栈：

```
result=(struct thread_vm *)m_malloc_gbl(len) + len – sizeof(struct thread_vm);
result->stack_top = (uint32_t*)((ptu32_t)result);
```

15.2.4.2 复位线程虚拟机

原型：void * __asm_reset_thread(void (*thread_routine)(struct event_script *),
struct thread_vm *vm);

说明：thread_routine 是该线程的入口函数，也就是线程对应的事件类型的事件处理函数，vm 是线程虚拟机指针。本函数用于复位线程虚拟机的上下文到起始状态，一般用汇编实现。复位后，线程虚拟机的栈处于这样一种状态：当线程被再次切入，从栈中恢复的上下文后，程序处于即将要调用虚拟机引擎函数__vm_engine 的状态，参数 thread_routine 就是作为__vm_engine 函数的参数。

移植要点：当以下三个要素改变时，可能需要修改本函数：

1. 更换 CPU、或者更换指令系统。比如 ARM 系列 CPU 有 ARM 和 thumb 两套指令系统。
2. 更换编译器，本函数由 C 代码调用，需要知道 C 代码是如何传递 thread_routine 参数和 vm 参数的，还要把 thread_routine 函数指针当作 C 语言函数__vm_engine 的参数，也要知道 C 的参数传递规则。由于 ANSI 没有规定函数参数传递规则，各编译器可以自己定义，一般来说，不更换 CPU 和指令系统的情况下，是无需修改本函数的，但也可能有例外，请仔细阅读编译器的说明书。
3. 修改栈类型（栈结构参见第 2.4 节），本函数初始化线程栈，栈类型牵涉到压栈和弹栈的次序和位置。

15.2.4.3 复位老线程，切入新线程

原型：void __asm_reinit_to(void (*thread_routine)(struct event_script *),
struct thread_vm *new_vm, struct thread_vm *old_vm);

参数：thread_routine：是被复位线程入口函数，由于入口函数是事件类型的属性，而不是虚拟机的属性，不能通过 old_vm 获得，因此必须通过参数来传递。
new_vm：被切入的新事件的线程虚拟机指针。
old_vm：被复位的线程虚拟机指针。

说明：本函数把老线程虚拟机 old_vm 上下文重新初始化到新创建的状态，然后切入新线程 new_vm 的上下文，使 new_vm 开始运行。当一个事件完成（线程入口函数返回或主动调用 y_event_done 函数），如果根据调度算法该虚拟机应当保留待用，就应该调用本函数来复位它并进行上下文切换。

15.2.4.4 上下文切入

原型：void __asm_turnto_context (struct thread_vm *new_vm);

参数：new_vm：被切入的新事件的线程虚拟机指针。

说明：不保存当前线程的上下文，直接把新线程的上下文切入并开始运行。当事件处理完成（线程入口函数返回或者显式调用 y_event_done 函数），如果处理该事件的虚拟机无需保留，则该虚拟机将消亡，其资源也将被操作系统收回，此时就应该调用本函数切换到新虚拟机中运行。另外，操作系统启动第一个线程虚拟机也是用这个函数。

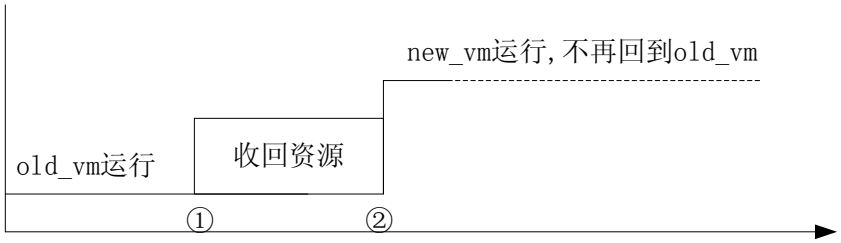


图 15-1 上下文切入

1. old_vm 运行至此，线程函数自然返回，或者显式调用 y_event_done，如果 old_vm 无需保留，则操作系统将收回 old_vm 所占用的资源，并使之消亡。
2. 操作系统完成删除虚拟机后，将调用__asm_turnto_context 函数切换到新的就绪线程中。

移植要点：

15.2.4.5 上下文切换

原型：void __asm_switch_context(struct thread_vm *new_vm,
struct thread_vm *old_vm);

说明：本函数与“上下文切入”只有一字之差，正如其字面含义，“切入”直接把CPU交给新线程，不管当前线程死活（事实上老当前线程已经“死亡”），而“切换”则是把CPU从执行老线程切换到执行新线程，要保存当前线程虚拟机的上下文。这是最基本最常用的线程上下文切换函数，当一个正在运行的线程被阻塞，或者有更高优先级的线程就绪，就会调用本函数进行线程切换。其过程如图 15-2 所示。

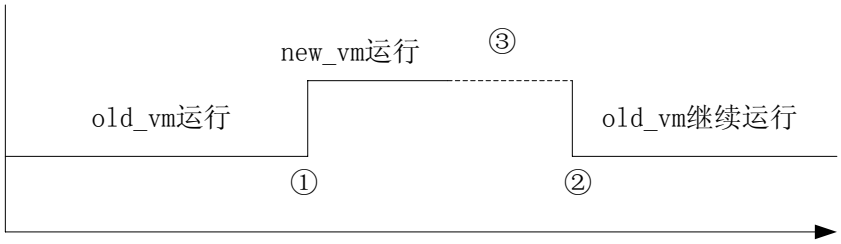


图 15-2 上下文切换

1. 线程在此时调用__asm_switch_context 函数，切换到 new_vm 中运行。
2. 线程返回到 old_vm 中运行，在 old_vm 线程看来，从①～②的过程是透明的，它并不知道曾经“丢失”过 CPU，好像只是__asm_switch_context 函数花了比较长的时间来执行一样。
3. new_vm 运行过程中也可能被别的线程虚拟机打断，可能经过了复杂的线程切换后才回到②，可能是从 new_vm 直接返回②，也可能从别的线程返回②，对此，old_vm 毫不知情。

移植要点：本函数要制造 old_vm 好像从__asm_switch_context 返回的假象，理解这一点很重要。

15.2.4.6 从异步信号 ISR 中返回时的上下文切换

原型：void __asm_switch_context_int(struct thread_vm *new_vm,
struct thread_vm *old_vm);

说明：参考 图 6-6，如果在异步信号ISR中使比被中断的事件更高优先级的事件就绪，从异步信号ISR返回主程序时，将调用本函数，完成上下文切换，使中断直接返回到新线程中开始执行。

1. CPU 响应异步信号时，将把正在处理的事件的上下文保存到中断栈中，执行中断服务函数。
2. 本函数在中断上下文中执行，目的是制造一种假象，似乎原事件不是被中断打断而是被新事件抢占的，而中断打断的不是原事件而是新事件。

15.2.5 中断系统

中断系统是一个个性十足的设计，每种类型的CPU都有其独特的中断控制硬件结构，即使在相同的CPU，中断模块软件也因程序员的习惯不同而不同，难于刻画出统一的软件模板，读者应在读懂本书第 第 6 章的基础上自行设计。

15.2.6 系统定时

系统定时有两类，一种是通过操作系统的时钟嘀嗒为应用程序提供服务，另一种是用软件指令循环为用户提供微小延时服务，与此相关的、需要移植的函数有：

1. 设定指令延时常数

   原型：void __y_set_delay (void)；

   说明：设定指令延时所需的几个变量，参见第 11.3.2 节。

   移植要点：

2. 初始化 tick

   原型：void __y_init_tick(void)；

   说明：连接 isr_tick 函数到提供时钟嘀嗒的定时器，把该中断源设置为异步信号，根据 port_kernel.h 文件中定义的常数 cn_tick_ms 和 cn_fine_us 初始化该定时器，启动该定时器，并打开定时器中断。（注：此时总中断开关尚未打开）。

   移植要点： 本函数的执行与硬件定时器设置模块和中断控制模块有关，这两个模块应提供相应的功能函数。

3. 读取当前 ticks

   原型：uint32_t y_get_time(void)；

   说明：ticks 是一个持续递增的 32 位无符号整数，超过 0xffffffff 后，将回绕到 0。除实时中断服务函数外，应用程序可以在任何时刻读取该值。

   移植要点： 只有把系统移植到 16 位或以下的 CPU 中时才需要修改本函数，把函数中关闭和打开定时器中断的中断号替换成实际使用的中断号。

4. 读取当前精密时间

   原型：uint32_t y_get_fine_time (void)；

   说明：读取精密时间得到从上次 tick 中断至今流逝的时间量，单位是微秒。

移植要点： 本函数完全依赖于具体硬件。

15.2.7 CPU 的自留地

在 CPU 的自留地里，主要收集一些与具体的 CPU 相关的代码，这些特性并不是实现操作系统所必须的，如果利用得好，可以大大提高系统的健壮性，并给用户提供友好的诊断信息。比如 ARM 的异常，当系统当机时，往往可以告诉用户当机的原因。

15.3 ARM7 版本—S3C44B0X

15.3.1 ARM 的自留地

本书成书时，这部分还没有写呢。

15.3.2 中断设计

15.3.3 线程栈结构设计

线程的栈是线程保存局部变量和线程上下文的地方，所有线程切换有关的操作都围绕线程的栈展开，所以，设计和移植操作系统，首要的任务就是设计目标系统的栈，包括线程栈、中断栈和内核栈。djyos系统在si模式和dlsp模式下，不存在内核栈，在mp模式下需要内核栈。设计线程栈的核心在于设计线程上下文区，S3C44B0X的线程栈如 图 15-3 所示。

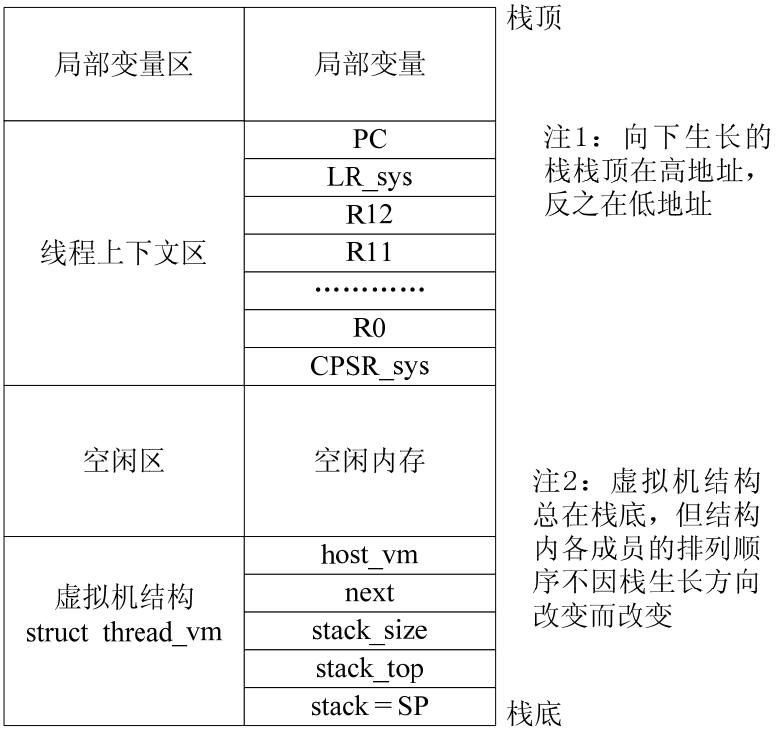


图 15-3 44B0X 的线程栈

在 图 15-3 中，当线程被调度时，线程上下文区是不存在的，局部变量区随线程运行，函数的调用与返回动态地变化，当线程被剥夺CPU时，才临时把各寄存器保存到栈中，从而生成“线程上下文区”，同时把栈指针SP的最终地址写入到虚拟机结构的stack成员中。注意，SP保存在栈底而不是保存在上下文区中，否则，当需要切入本线程时，调度器无法获得线程上下文区的起始地址，也就无法取出线程上下文。虚拟机结构总是位于线程栈的底部，其内部成员的排列顺序与其在struct thread_vm结构内部定义的顺序相同。stack成员保存的是线程被切离时的SP值，并不跟踪程序执行中SP的动态变化。

15.3.4 常量配置

在gcc+44b0x环境下，port_kernel.h文件中各常量的配置如 表 15-2 所示。

表 15-2 gcc+44B0X 环境下 port_kernel.h 中常量设置

常量	值
cn_mclk	64000000
cn_cpu_endian	cn_little_endian
cn_cpu_bits	32
cn_point_bits	32
cn_cpu_bits_suffix_zero	5
uint64_t/sint64_t	unsigned long long/signed long long
uint32_t/sint32_t	unsigned int/signed int
uint16_t/sint16_t	unsigned short/signed short
uint8_t/sint8_t	unsigned char/signed char
ucpu_t/scpu_t	uint32_t/sint32_t

ptu32_t	uint32_t
cn_stack_type	cn_full_down_stack
align_down(x)/ align_up(x)	align_down_8(x)/align_up_8(x)
cn_int_num	26，因为 S3C44B0X 有 26 个中断源
cn_tick_ms	1
cn_tick_hz	1000
cn_fine_us	1
cn_fine_hz	1000000
cn_run_mode	44B0X 可运行在 cn_mode_si 模式和 cn_mode_dlsp 模式下，移植者根据需要配置此常量。

15.3.5 线程相关函数

在 44b0x的实现版本中，需移植的线程相关函数全部用汇编实现，在cpus.s文件中。这些函数都是围绕线程栈展开，阅读本节需要结合第 15.3.3 节帮助理解。

15.3.5.1 复位线程虚拟机

本函数的原型是：

void * __asm_reset_thread(void (*thread_routine)(struct event_script *),
struct thread_vm *vm);

按照 ARM 的 atpcs（子程序调用基本规则），r0 = thread_routine，R1 = sp_top。

代码 15-2 __asm_reset_thread 函数

```
.global   __asm_ reset _thread
__asm_ reset _thread:
    stmfd     sp!,{lr}                  //1
    ldr       r2,[r1,#4]                //2
    ldr       r3,=__vm_engine
    stmfd     r2!,{r3}                  //3
    sub       r2,r2,#13*4               //4
    stmfd     r2!,{r0}                  //5
    mrs       r0,CPSR
    stmfd     r2!,{r0}                  //6
    str       r2,[r1]                   //7
    ldmfd     sp!,{pc}                  //8
```

1. 把返回地址压栈，按照 atpcs 规则，r2 和 r3 无需压栈。
2. 取虚拟机栈顶指针到 r2
3. 取__vm_engine 函数的地址，并把它保存到栈中上下文区的 PC 的位置。
4. 初始状态时，就像刚上电是物理计算机的寄存器值是随机的一样，线程虚拟机的的 r1~r12 也是随机的，无需保存，直接维持内存中的随机值即可。由于__vm_engine 函数并不返回，故 LR 在这里并无意义，也无需保存，所以栈指针直接下移 13 个

单元。特别要注意的是，这里说的栈和第 5、6 步所说的栈，都是目标虚拟机的栈，而不是当前程序运行的栈，当前栈指针用 SP 表示。

5. 把 r0（即线程入口函数 thread_routine）保存到栈中，当线程第一次获得操作系统照顾时，__vm_engine 的地址将被恢复到 PC 中，根据 atpcs 规则，r0 将作为 __vm_engine 的参数。注意，thread_routine 函数本身的参数无需在此提供，而是在 __vm_engine 函数内调用 thread_routine 函数时提供。
6. 把 CPSR_sys 压入栈中。
7. 把虚拟机 wm 的当前栈指针保存到 vm->stack 中。
8. 从栈中恢复 PC，函数返回。

15.3.5.2 重置老线程，切换到新线程

本函数的原型是：

```
void    __asm_reset_switch (void (*thread_routine)(struct event_script *),
                                                    void *new_vm,void *old_vm);
```

按照 ARM 的 atpcs（子程序调用基本规则），r0 = thread_routine，r1 = new_vm，r2 = old_vm。本函数在被重置的线程上下文中执行，由于该线程即将被复位，保存函数调用现场已没有意义，故开头并没有出现压栈指令。

代码 15-3 __asm_reinit_to 函数

```
.global    __asm_reset_switch
__asm_reset_switch:
        ldr         sp,[r2,#4]                  //1
        ldr         r12,=__vm_engine
        stmfd       sp!,{r12}                   //2
        sub         sp,sp,#13*4
        stmfd       sp!,{r0}
        mrs         r0,CPSR
        stmfd       sp!,{r0}
        str         sp,[r2]
        ldr         sp,[r1]                     //3
        bl          int_restore_asyn_signal     //4
        ldmfd       sp!,{r1}
        msr         CPSR_cxsf,r1                //5
        ldmfd       sp!,{r0-r12,lr,pc}          //6
```

1. 取老线程栈顶指针，请参考 图 15-3 中虚拟机结构部分。
2. 从本步骤直到第 3 步（不含）之前，请参考 代码 15-2 的第 3 至第 7 步。
3. 取新线程的当前栈指针到SP寄存器，也就是 图 15-3 中“线程上下文区”的指针。
4. 本函数被 y_event_done 函数调用，由于在 y_event_done 函数的开头，调用了 int_save_asyn_signal 函数关闭中断（亦即关闭调度），须再次打开中断，使 new_vm 在开中断的状态下从新开始运行。
5. 从栈中恢复 new_vm 虚拟机的 CPSR 寄存器。
6. 把 new_vm 的栈中保存的 r0~r12，lr 和 PC 恢复到寄存器中，新 new_vm 从恢复的 PC 处恢复运行。

15.3.5.3 直接切入上下文

本函数的原型是：

void __asm_turnto_context (struct thread_vm *new_vm);

根据 atpcs 规则，new_sp 参数在 r0 中传递，调用本函数时，当前线程即将消亡，因此当前线程的上下文无需保存到栈中。

代码 15-4 __asm_turnto_context 函数

```
    .global __asm_turnto_context
__asm_turnto_context:
    ldr     sp,[r0]                         //1
    bl      int_restore_asyn_signal             //2
    ldmfd   sp!,{r0}
    msr     CPSR_cxsf,r0                    //3
    ldmfd   sp!,        {r0-r12,lr,pc}      //4
```

1. 取新虚拟机上下文区当前栈指针请参考 图 15-3 中虚拟机结构部分。
2. 2~4 步与 代码 15-3 的 4~6 步是相同的。

15.3.5.4 上下文切换

本函数的原型是：

void __asm_switch_context(struct thread_vm *new_vm,struct thread_vm *old_vm);

根据 atpcs 规则，r0 = new_vm，r1 = old_vm。

代码 15-5 __asm_switch_context 函数

```
    .global __asm_switch_context
__asm_switch_context:
    stmfd   sp!,{lr}                        //1
    stmfd   sp!,{r0-r12,lr}                 //2
    mrs     r4,CPSR
    stmfd   sp!,{r4}                        //3
    str     sp,[r1]                         //4
    ldr     sp,[r0]                         //5
    bl      int_restore_asyn_signal             //6
    ldmfd   sp!,        {r0}
    msr     CPSR_cxsf,  r0                  //7
    ldmfd   sp!,        {r0-r12,lr,pc}      //8
```

1. 保存LR寄存器到栈中PC的位置（参考 图 15-3 中线程上下文部分），如果__asm_switch_context 是个普通函数，LR是调用此函数的返回地址。把它放在PC所在的位置，是制造“当线程重新被调度时，在线程看来是从__asm_switch_context函数返回”的假象的关键。
2. 把 r0~r12，LR 寄存器保存到栈中。
3. 保存 CPSR 到栈中。
4. 保存当前栈指针到 old_vm->stack 中。

5. 5~8 步与 代码 15-3 的 3~6 步是相同的。

15.3.5.5 从异步信号 ISR 返回时的线程切换

本函数的原型是：

void __asm_switch_context_int(struct thread_vm *new_vm,struct thread_vm *old_vm);

根据 atpcs 规则，r0 = new_vm，r1 = old_vm。本函数被__schedule_int 函数调用，在__schedule_int 函数中把 pg_event_running 当作 old_vm。

这是线程相关函数中最复杂的一个，为了支持中断嵌套，ARM的中断处理函数要切换到svc状态下执行，因此，与前面 3 个切换函数不同，本函数调用完之后，并不是直接进入新线程中执行，而是仍然返回到中断函数处。本函数的作用是，然后瞒天过海，制造出两个假象，一是old_vm似乎是被new_vm抢占而不是被中断打断的，二是似乎被中断的线程是new_vm而不是old_vm。这样，接下来中断返回进入主程序时，将直接进入new_vm。图 15-4~图 15-6 显示了调用该函数前后以及过程中栈中数据转移图，代码 15-6 则是__asm_switch_context_int函数的实现。经过这一系列的寄存器换位以后，中断服务例程返回主程序时，就直接返回到new_vm中，而old_vm则被挂起。

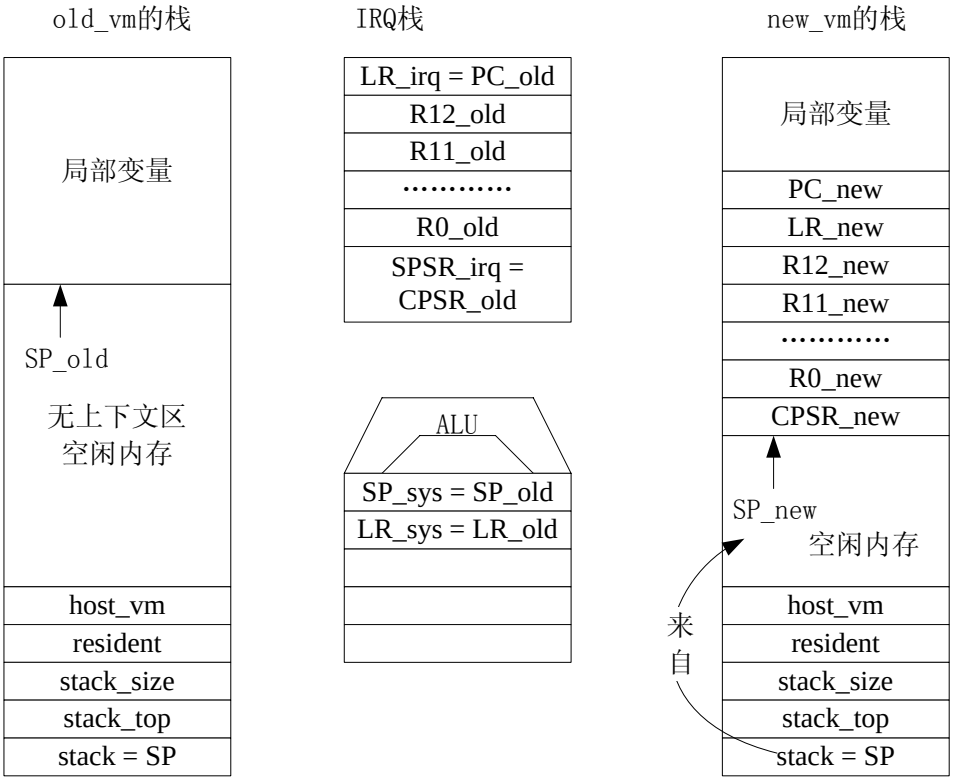


图 15-4 调用__asm_switch_context_int 前的寄存器状态

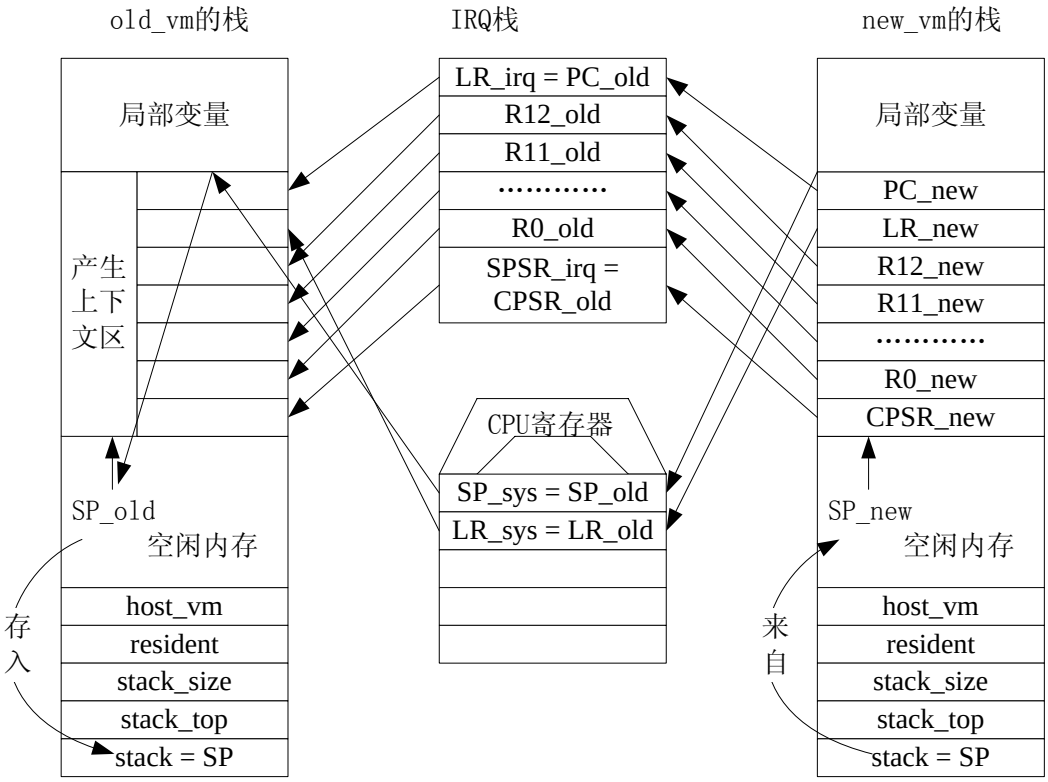


图 15-5 寄存器转移图

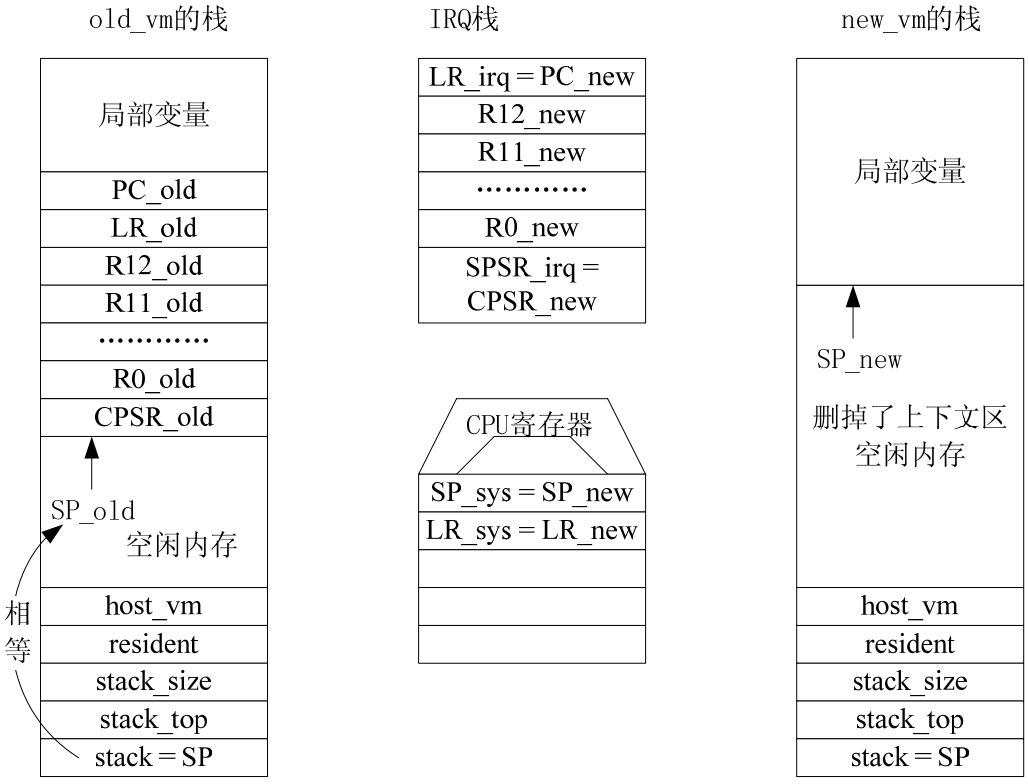


图 15-6 调用__asm_switch_context_int 后的寄存器状态

代码 15-6 __asm_switch_context_int 函数

```
        .global __asm_switch_context_int
```

```
__asm_switch_context_int:
    stmfd    sp!,{r2-r12,lr}    @保存正在服务的中断上下文

    @以下几行把 old_vm 的上下文从正在服务的中断栈顶转移到虚拟机栈中
    ldr      r2,=IRQ_stack      @取 irq 栈基址,这里存放着被中断线程的上下文
    ldmea    r2!,{r3-r10}       @按递增式空栈方式弹栈，结果:
                                @[r2-1]=LR_irq->r10，被中断线程的 PC+4
                                @[r2-2]=r12->r9，被中断线程的 r12
                                @[r2-3]=r11->r8，被中断线程的 PC
                                @类推之……
                                @[r2-8]=r6->r3，被中断线程的 r6
    sub      r10,r10,#4         @中断栈中的 LR_irq-4=PC

    @以下三句就是取出 old_vm 的 SP_sys，只能通过 stmfd 指令间接取
    mov      r11,sp             @下一句不能用 SP，故先拷贝到 r11
    stmfd    r11!,{sp}^         @被中断线程的 SP_sys 压入正在服务的中断栈中
    ldmfd    r11!,{r12}         @从正在服务的中断栈中读取 SP_sys->R12

    stmfd    r12!,{r10}         @保存 PC_sys
    stmfd    r12!,{lr}^         @保存 lr_sys
    stmfd    r12!,{r3-r9}       @保存被中断线程的 r12-r6 到它的栈中
    ldmea    r2!,{r3-r9}        @读被中断线程的 r5-r0->r9-r4,SPSR_irq->r3,递增式空栈
    stmfd    r12!,{r3-r9}       @保存被中断线程的 r5-r0,CPSR_sys 到它的栈中

    str      r12,[r1]           @换出的上下文的栈指针-->old_sp

    @以下几行把 new_vm 的上下文 copy 到 IRQ 栈顶，
    @与递减式满栈对应,此时 IRQ 栈用递增式空栈的方式访问
    ldr      r12,[r0]           @读取需换入的栈指针
    ldmfd    r12!,{r3-r11}      @读取换入线程的 CPSR_sys->r3
                                @读取换入线程的 r0-r7->r4-r11
    stmea    r2!,{r3-r11}       @保存换入线程的 CPSR_sys->SPSR_irq, r0-r7 到 IRQ 栈
    ldmfd    r12!,{r3-r7}       @读取换入线程的 r8-r12->r3-r7
    stmea    r2!,{r3-r7}        @保存换入线程的 r8-r12 到 IRQ 栈
    ldmfd    r12!,{lr}^         @恢复换入线程的 LR_sys 到寄存器中
    ldmfd    r12!,{r3}          @读取换入线程的 PC->r3
    add      r3,r3,#4           @模拟 IRQ 保存被中断上下文 PC 的方式：PC+4->LR_irq
    stmea    r2!,{r3}           @保存换入线程的 LR_irq 到 IRQ 栈
    stmfd    r12!,{r12}         @读取 SP_sys 到 r12
    ldmfd    r12!,{sp}^         @恢复 SP_sys
    mov      r0,r0              @无论是否操作当前状态的 SP，操作 sp 后，不能立即
                                @执行函数返回指令，否则返回指令的结果不可预知。
    ldmfd    sp!,{r2-r12,pc}
```

结合 图 15-4~图 15-6，代码中的注释已经非常明了，就不再解释

15.3.6 中断系统

毫无疑问，操作系统在管理中断时，必须考虑中断嵌套的需求，在 ARM 上实现中断嵌套是一件很繁琐的事情，笔者一直认为这是 ARM 设计中的一个瑕疵，因为它无端使中断服务例程变得晦涩难懂，而且降低中断响应速度——它不得不花费许多条指令用于和用户需求无关的状态切换。

15.4 cortex-M3 版本

15.5 DSP 版本—TMS320F2808

第16章　嵌入式 C 语言编程杂谈

有许多人认为，用 C 语言写的程序，从这个平台搬到那个平台时，只要重新编译就可以了，其实不是这样的，C 语言的许多要素是跟具体硬件平台和软件编译器相关的，同样的 C 代码，在不同的平台上的运行结果不一定相同。而我们把一个产品上的成熟代码移植到别的产品时，调试和测试中都或多或少地会麻痹大意，简单地试一下能不能运行就定版，如果进度要求很紧张的话，尤其如此。如果原来的代码没有注意 C 语言与平台相关的差异性，往往容易留下隐患，成为产品故障的诱因。这也是在一个企业内要求严格限定平台种类数量的原因之一。

了解一下 C 的平台差异性产生的原因，对识别 C 语法中的哪些要素是平台相关的有很大帮助，在日常编程工作中，用到到这些可能平台相关的要素时，就会自然地产生象狼一般的警觉，避开这些差异，将会使你的程序有更好的移植健壮性。70 年代，一群计算机科学家正在做 unix，他们不能忍受汇编语言的笨拙与繁琐，也不愿意使用 PL/I 和 Fortran，于是创造了 C 语言（严格来说是先 B 语言而后 C 语言）。他们发明了 C 语言，并制作了第一代的、运行于 unix 的 C 编译器，然后再用 C 语言以及这个编译器继续完成 unix 操作系统。有趣的事情发生了，C 语言的发明者又是 C 编译器的作者，当时机器速度慢、内存少，C 语言的语法总是优先照顾 CPU 和编译器作者，语法怎么定义可以简单高效地实现就怎么定义，而当时 CPU 的种类又比较单一，CPU 间移植的问题当然也就很少考虑了。因此，C 语言的许多特性是为了方便编译器作者，而程序员的利益考虑得比较少，编译器作者直接面对目标硬件平台，这迫使程序员也要熟悉目标硬件平台，否则很难写出优质高效的代码。纵观 C 语言所有未严格定义的要素，都与方便编译器作者在硬件上实现，以及提高执行效率有关。这使 C 的语法打上了严重的硬件相关烙印，试举数例：

1. 数组下表从计算机习惯的 0 开始，而不是从人类思维习惯的从 1 开始。
2. C 语言的基本数据类型与硬件支持类型一致，早期的 C 语言没有浮点类型，直到机器支持浮点后才支持。int 类型的长度依机器字长而定，甚至出现过 26 位长度的 int 类型。在有些机器上，float 与 double 类型是一致的，因为这些机器的硬件就是这样实现的。
3. register 关键字，典型的硬件标记。
4. char 类型是否带符号，由机器决定。同样由机器决定的还有多字节数据的大小端。
5. ……

同样，C 语言为了照顾编译器作者产生简短快速高效的目标代码，而无视程序员的利益，C 语言的设计理念认为，程序员必须自己知道自己在干什么，自己保证自己的所有行为是正确的，例如：

1. C 语言从来不做数组下表检查，数组下标溢出会有什么结果，只有天知道。
2. switch-case 语句必须由程序员显式书写 default，否则，在找不到匹配项时，不会有任何提示。
3. 从来不做空指针引用检查。在 windows 下，是操作系统帮你检查，在 dos 下，只能求菩萨保佑，在 vxworks for arm7tdmi 下，只能听天由命，在 vxworks for arm9 下，因有 MMU 协助，如果你的 bsp 足够高明，也可以仰赖操作系统帮你检查。
4. ……

C 语言给了编译器作者无限的自由，有许多语法要素是没有明确定义的，编译器作者可

以根据自己的喜好或者目标 CPU 指令特点实现，导致相同的代码在不同的 CPU 环境下，甚至是相同 CPU 不同编译器环境下，执行的结果是不相同的。我们选取了具体的几个平台，看看没有明确定义的 C 语言要素在这些平台上的表现。只能略举一二，不能穷尽，意在抛砖引玉，有心的读者可以继续挖掘。

1. 编译器：turbo C3.0，CPU：x86。
2. 编译器：Borland C++ builder 6，CPU：x86。
3. 编译器：ARM-elf-gcc，CPU：ARM7TDMI（存储器配置为小端）。
4. 编译器：ARM-elf-gcc，CPU：ARM7TDMI（存储器配置为大端）。
5. 编译器：Texas Instruments Incorporated 公司的 CCS 编译工具，CPU：TMS320C33。

16.1 对齐及数据长度

有下列定义：

```
struct size{
        char a;
        int  b;
        }
```

sizeof(struct size)的值是什么？请看下表

CPU	编译器	结果
x86	turbo C3.0	sizeof(struct size) = 3
x86	Borland C++ builder 6	sizeof(struct size) = 8
arm-elf-gcc	ARM7TDMI（大端和小端）	sizeof(struct size) = 8
TMS320C33	Texas Instruments Incorporated 公司的 CCS 编译工具	sizeof(struct size) = 2

点评：turboC 的结果是 3，char 1 字节，int 2 字节，int 没有按偶地址对齐，这个不难理解，看看 turboC 多大岁数就知道了，它出生时 x86 就是 16 位机嘛。那时内存比 CPU 贵得多，x86 又支持非对齐访问（慢一些而已），故 char 和 int 压得紧紧的存储。

BorlandC++ builder 就不同了，它的 int 是 4 字节的，并且结构内做了对齐，使成员 a 和成员 b 直接插入了 3 个填充字节，所以 size=8。gcc for arm 也是这个道理。

大家注意到了，即使相同的硬件平台，不同的编译器会给出不同的结果，这充分说明了 C 语言标准定义的不严谨，编译器作者可以按照自己的理解行事而不用担心会违反 C 标准。编译器作者获得了充分的自由，却不把程序员当人看，呵呵，你不但要清楚自己在干什么，还要猜测编译器会怎么干，你说冤不冤！

ccs for TMS320C33 的结果比较奇怪，但仔细分析一下 TMS320C33 的硬件，你就不会觉得怪了。sizeof 是以字符为单位计算 size 的，TMS320C33 是纯 32 位机，它的 char 是 32 位的，int 也是 32 位的，2 个 32 位成员加在一起，size 当然是 2 了，刚好两个字符宽度。需要特别说明的是，C33 的 char 虽然占据 32 位存储器，但是其值域仍然是 0~255，多出来的 3 个字节属于没用的填充物。

sizeof(struct size)的值不但受编译工具和 CPU 的影响，而且在结构内成员的排列顺序也有影响，一个成员较多的结构，把相同长度的成员集中排序，先把所有 8 位成员放在一起，16 位次之，32 位再次，或者反之，往往能节省许多内存，当然，这是不太多地影响可读性为前提。有些经验不足的程序员在编写通信程序时，可能会因为对齐而留下 bug，一个由嵌

入式系统组成的网络，往往由多种 CPU，问题就出来了。假设通信发送和接收函数的原型如下：

uint32_t send_data(uint8_t *send,uint32_t len);

uint32_t recv_data(uint8_t *send,uint32_t len);

发送方调用 send_data 函数发送：

struct struct_a;

send_data((uint8_t*)&struct_a,sizeof(struct size));

用如下调用接收数据：

recv_data((uint8_t*)&struct_a,sizeof(struct_size));

从前面的分析可以知道，该函数实际发送和接收的数据量将依 CPU 和编译环境不同而不同，如果收发在相同的 CPU 中执行，还能凑巧正确，否则就是鸡言对鸭语。特别严重的是，在一个企业内，可能只用一种 CPU，上述问题可能测试不出来，而产品卖到用户手中的后，用户可能采购了多家企业的产品联网，问题就出来了。这种事后补救工作，成本比在开发中发现问题要高得多。

16.2 强制类型转换

代码：

```
char ca,cb,cc,cd;
int ia;
ca = 1; cb = 2; cc = 3; cd = 4;
ia = 0x12345678;
```

问题：

（int）ca = ？

（char）ia = ？

在不同的环境下，结果如下：

CPU	编译器	结果
x86	turbo C3.0	（int）ca = 0x1 （char）ia =0x78
x86	Borland C++ builder 6	（int）ca = 0x1 （char）ia =0x78
arm-elf-gcc	ARM7TDMI(小端)	（int）ca = 0x1 （char）ia =0x78
arm-elf-gcc	ARM7TDMI(大端)	（int）ca = 0x1 （char）ia =0x78
TMS320C33	Texas Instruments Incorporated 公司的 CCS 编译工具	（int）ca = 0xxxxxxx01 （char）ia =0x78

点评：除 TMS320C33 以外，其他 4 种环境均顺利转换。TMS320C33 比较特殊，char 本来占据 32 位存储器，但只使用了最低 8 位，若转换成 int 访问，则把原来没用的 24 位填充物也用上了，我们并不知道这 24 位原来是什么值，只好用 xxxxxx 来表示结果，x 代表任意值。

16.3 慎用 union

在通信程序中，经常可以看到用联合体进行数据组装的代码，请看：

```
union data_pack
{
```

```
    unsigned int    word;
    unsigned char bytes[4];
};
union data_pack tmp;
tmp.bytes[0] = 0x12;
tmp.bytes[1] = 0x34;
tmp.bytes[2] = 0x56;
tmp.bytes[3] = 0x78;
```

问题：

tmp.word = ?

在不同的环境下，结果如下：

CPU	编译器	结果
x86	turbo C3.0	0x3412
x86	Borland C++ builder 6	0x78563412
arm-elf-gcc	ARM7TDMI(小端)	0x78563412
arm-elf-gcc	ARM7TDMI(大端)	0x12345678
TMS320C33	Texas Instruments Incorporated 公司的 CCS 编译工具	0x00000012

点评：有了前面的基础，这个应该很容易理解了，turbo C3.0 中整数是 16 位的，当然是 0x3412 了。按大端配置的 ARM 中，多字节数据的高字节放在低地址，bytes 数组中，显然 bytes[0]会出现在低地址，它自然成了 32 为的整数的最高字节。而小端配置则正好相反 bytes[3]就成了最高字节。C33 的字符是 32 位的，bytes[0]与 word 的地址相同，bytes[1]~bytes[3]单独占用 3 个 32 位数，故有 tmp.word 与 tmp.bytes[0]相等。

牺牲一点效率，把代码写成这样会更安全：

```
    typedef unsigned char uint8;
    tupedef unsigned int    uint32;
//在 turbo C 或其他 16 位环境则改为：
    tupedef unsigned    long int    uint32;
    uint8 bytes4];
    uint32 word;
    word=(unit32_t)bytes[3]<<24;
    word=word|((unit32_t)bytes[2]<<16);
    word=word|((unit32_t)bytes[1]<<8);
    word=word|((unit32_t)bytes[0]);
```

同样，由于 C 标准没有明确指定位域的数据格式，C 编译器可以任意安排，使用位联合体也是要相当小心的。在汽车工业软件可靠性联合会制定的 C 语言编程规范中，根本就是禁止使用 union 关键字的。

union 的困惑是由于 C 对 union 各成员的存储、对齐、排列、填充的方式没有限定造成的，只要避开了这些不确定因素，使用 union 是安全的。在 djyos 系统中，在下列两种情况下，使用了联合体。

1. 联合体内各成员不交叉访问的情况，即某联合变量，要么只按成员甲访问，要么只按成员乙访问，中间没有交叉。例如：

```
union lock_MCB
{
```

```
    struct semaphore_LCB sem;
    struct mutex_LCB    mut;
};
```

如果某个锁被作为信号量创建，则永远按信号量访问，如果作为互斥量创建，则永远按互斥量访问。

2. 数据的内容只按其中一个成员解释，包括隐含的和明示的解释。例如：

```
union file_attrs
{
  uint8_t all;
  struct file_attr_bits bits;
};
```

其中 bits 是一个位域结构，对某具体位的置位、清零、条件判断完全在 bits 成员上进行，all 成员仅仅用于把所有位清零或者置位，不对位成员的存储位置做任何条件判断和解释。

16.4 适应运行环境

程序员不是在臆想的环境下工作，而是受 CPU、硬件配置以及软件开发系统共同组成的开发环境制约，在动手设计之前，先要熟悉软件的运行环境，才能保证你的软件在目标环境下的高效性和移植正确性。这有两层意思，一是你开发产品时肯定是针对某种具体的软硬件环境，你开发的软件可能一辈子就在这个环境下工作，应该优先考虑在这个环境下高效运行；二是需要考虑到软件的可移植性，要假设你的软件会被移植到别的软硬件环境中运行，你应该确保你的软件不加修改直接在新环境中重新编译即可正确运行，当然效率可能会比你的初始环境低点。djyos 系统定义了一个很特别的数据类型 ucpu_t 和 scpu_t，含义是字长等于 CPU 字长的有符号和无符号整数类型，不用关心数据字长的变量可以使用这个数据类型。不关心数据字长的类型可以理解为 8 位就可以满足要求的、不受其他协议限定类型的整数类型。djyos 的 ARM7 移植版本上，由于 ARM7 是 32 位 CPU，故 ucpu_t 被定义为 uint32_t。下列代码：

```
ucpu_t    loop;
for (loop = 0; loop < 100; loop++)
{    ;    }
```

在 32 位机上，loop 将被定义为 32 位无符号数，其执行速度比定义为 8 位无符号数还快，但在 8 位机器上编译时，loop 将作为 8 位无符号数处理，同样可以得到最优化的运行速度。但是，如果循环语句变成 for (loop = 0; loop < 1000; loop++)，则 loop 不能定义成 ucpu_t，如果编程时目标系统是 32 位的，就应该定义为 uint32_t，以期得到最优化的运行速度，当移植到 8 位或者 16 位系统时，软件也能正确运行，只是比 uint16_t 的执行效率低一些。反之，如果在 8 位或 16 位系统上编程，就应该定义为 uint6_t，可以得到最优化的效率，移植到 32 位系统时也能正确运行，只是比定义为 uint32_t 效率低一些。

如果说上述例子影响的是软件执行效率的话，下面的例子将揭示一个与运行环境有关的、隐藏着一个严重的潜在 bug 的案例。

```
char latent_bug=0;
int j;
char buf[256];
for(j =0; j <256; j ++)
```

```
buf[latent_bug ++]= latent_bug;
```

正如同变量命名为 latent_bug 所暗示的，这是一段隐含了 bug 的程序，如果你正在使用的编译器把 char 当无符号数处理，那么程序可能正确执行，你无论如何也测试不到其中有 bug，但是，一旦你更换了编译器，新编译器把 char 当有符号数处理，则可能会发生灾难性的后果。为什么会这样呢？因为 ansi c 没有规定 char 数据类型作为整数使用的时候的符号，当编译器把 char 当无符号数使用的时候，OK，恭喜你，这段代码能得到正确的结果，你成功地把 buf[0]~buf[255]这段内存修改掉；反之，如果你更换了编译器，而新编译器把 char 当作有符号数处理，很不幸，你中招了！你改写的实际上是 buf[-128] ~buf[127]这些存储单元的数据，而你真正要修改的 buf[128]~buf[255]却纹丝不动。至于 buf[-128] ~buf[-1]里究竟存放这什么数据，只有天知道，你能做的，就是祈求上苍保佑，在 buf[-128] ~buf[-1]这些单元放的不是什么重要的数据，而是一段空闲内存。

16.5 软件效率

程序员经常会讨论软件效率的问题，软件效率是什么？只有把这个问题定义清楚了，讨论软件效率才有意义。许多人认为，软件效率是以执行速度、存储器消耗来衡量的，这在当年存储器价格按 bit 计算，CPU 速度只有数 Mhz 或更低的年代，这确实占据非常重要的地位。而如今，计算机系统的硬件成本日益下降，性能日益提升，软件复杂度凸现的时代，这种看法显得有些片面了，在产业化的软件开发实践中，软件效率至少包含以下几个方面：

1、 运行效率，也就是单位时间内软件能够完成多少工作。
2、 存储效率，存储效率包含两个方面，一是代码需要多少存储器来存储，二是软件运行过程中需要消耗多少内存。
3、 研发效率，即开发一个软件需要消耗的时间，这个时间与开发工具，软件系统架构、以及硬件系统支持程度都有关系。
4、 移植效率，顾名思义，当软件的运行平台发生变化，或者软件的需求发生变化时，调整软件适应新的环境所需要消耗的时间。
5、 阅读效率，软件代码本书就是一个原始的技术文档，有时候，需要阅读代码才能了解软件的全部技术细节。另外，软件测试中，有一种白盒测试方法，也要求测试员阅读代码，高效的软件必须是高阅读效率的。

以上列出的几条效率中，有些方面可能是相互制约的，比如要得到高的研发效率，可能就要牺牲程序的运行效率和存储效率，因为为了达到高的运行速度或者节省存储器，就要使用各种技巧去优化代码，而这些工作肯定会降低研发效率。

除软件本身的设计外，开发工具也是影响软件研发效率的重要因素。嵌入式操作系统身兼两重身份，它既是和应用程序一起运行的代码，同时又是开发工具，因此，嵌入式操作系统不仅仅要考虑为应用程序提供稳定高效的运行环境，还要考虑它作为开发工具和开发环境的方便性和易用性。

16.6 定长数据类型

本章前面的论述中我们知道，C 语言中许多数据类型并没有规定字长，我们建议程序员尽量使用确定长度的数据类型，因为这样的程序有最好的可移植性，而且能很好地满足特定协议和特定外设的处理。

1. 定长数据有利于改善可移植性

标准 C 语言和 C90 定义了许多数据类型，但其中只有 char 类型的长度是固定为 8 位的，其他数据类型的长度是不确定的，它可以根据体系结构和具体编译器的变化而变化，这样做的好处就是编译器可以根据体系结构确定数据长度以产生高效的代码。坏处就是会使程序员无所适从，没有人能保证 int 是 16 位的还是 32 位的，如果你使用了一个 int 变量，并且给它赋值 100000，这在 32 位机器上，int 一般是 32 位的，不会有问题，但如果把软件移植到 16 位机器上，这时 int 一般是 16 位的，数据就会被截断成-31072，而且这样的 bug 往往很难查。嵌入式系统的跨度太大，这样跨 CPU 字长移植程序的事情是经常发生的。使用 C 语言的标准数据类型往往会带来可移植性问题，从 PC 软件转型写嵌入式软件的程序员容易犯下列错误：

1、 总认为 int 类型是 32 位的，事实上，在 16 位机上，int 多数是 16 位的，在 64 位机上，int 有可能是 64 位的。唯一可以确定的是，ANSI C 规定了 int 的最小字长是 16 位，因此在 8 位机上，int 也不会是 8 位的。
2、 总认为指针是 32 位的，事实上指针的字长变化是最大的，从 8 位到 64 位都有可能。
3、 总认为 char 是有符号的，因为 PC 上 char 就是被当作有符号数使用的，但在嵌入式系统的编译器上，却有些系统是有符号的，有些是无符号的，也有些是通过编译选项由用户确定的。

2. 处理特定外设需要定长数据

外设的字长是由外设本身的特性决定的，编写访问外设的程序时，应该使用与外设长度相等的变量，否则会带来意想不到的后果。例如一个 8 位的外设被安排在 0x1000 开始的一段内存区间，该段存储器总线就应该被配置成 8 位存储器的方式访问，如果我们用下面的代码往 0x1000 写一个字节数据再在同一个地址读入一个字节：

```
int   data=0;
int   *p=0x1000;
*p=data; //     （3）
data=*p; //     （4）
```

如果该程序运行的目标硬件中，int 是 32 位的变量，那么第（3）行代码会依次往 0x1000、0x1001、0x1002、0x1003 写入 0，可能造成对设备的误操作。第（4）行代码会依次从 0x1000、0x1001、0x1002、0x1003 读出一个字节数据数据，我们知道，有些硬件设备是读敏感的，多余的读操作也会导致误操作。只要用定长数据类型把 data 和 p 的定义改为：

```
uint8_t data = 0;
uint8_t *p = 0x1000;
```

就可以避免上述问题。

3. 处理特定协议时需要定长数据

典型的是在通信系统中，数据报文中的所有成员都是确定了长度的，才能保证通信双方一致。软件在处理这些数据时，就必须使用定长的数据类型。

C99 使用 typedef 定义了一系列的定长数据类型，这些定长类型并不是新数据类型，而是相应字长的整数类型的别名，例如 int16_t 表示 16 位有符号整数，uint32_t 表示 32 位无符号整数，在不同的系统中，他们的实际类型不一样，例如在 16 位的系统中，uint32_t 可能被定义为 unsigned long，而在 32 位的系统中，则可能是 unsigned int，定长类型的长度的正确性由编译器的发行商保证，用户无需关心。但目前我们使用的很多编译器尤其是嵌入式系统编译器都还没有支持 C99 标准，因此必须在我们自己的代码里定义定长数据类型。djyos 系统基本上不使用非定长数据类型，只在特定的条件下使用的 ucpu_t 和 scpu_t 是非定长的，下列是 djyos 系统在 ARM 平台上使用的定长数据类型定义。

```
#define u32_t                unsigned int
```

```
#define u16_t          unsigned short
#define s32_t          int
#define s16_t          short int
#define u8_t           unsigned char
#define s8_t           signed char
```

定长数据类型的长度是否正确，还应该经过检查才能确定，下节会介绍检查方法。

4. 定长数据类型检查

以下代码可以检查定义的数据类型在所使用的硬件以及软件开发环境下是否符合要求，在系统设计的初期，应该调用这个函数检查数据类型的正确性。

```
bool_t CheckType(void)
{
   bool_t result=true;
   //并无小于 8 位的数据（位域除外），故只须检查是否大于 8 位
   if( (uint8_t)0x100 != 0)
   {
      printf("uint8_t type error\n");
      result=flase;
   }
   //下式第一个条件检查是否大于 16 位，第二个条件检查是否小于 16 位
   if(( (uint16_t)0x10000 != 0) || ((uint16_t)0xffff != 0xffff))
   {
      printf("uint16_t type error\n");
      result=flase;
   }
   //用相似的方法可以检验其他类型，在此略去。
   return(result);
}
```

注意，不能用 if(sizeof(uint16_t) == 2)来检查 uint16_t 类型是否 16 位数据，这只在最小总线单位是 8 位（即有字节访问指令）的机器上是正确的，在最小访问单位是 16 位的机器上（例如 TMS320F28xx 系列），sizeof(uint8_t) = sizeof(uint16_t) = 1，sizeof(uint32_t) = 2；在最小访问单位是 32 位的机器上（例如 TMS320C33 上），sizeof(uint8_t) = sizeof(uint16_t) = sizeof(uint32_t) = 1。

16.7 数据完整性

在多线程程序设计中，可能会发生并发执行的多个线程“同时”访问某个数据。并发访问中要保证数据完整性，确保任何写操作能够把正确而且完整的数据写入内存，读操作也能够得到正确而且完整的数据。原子原本是指不可分割的粒子，原子操作指的是不能再被分隔的指令或指令序列，该指令序列看起来就像单条指令一样，在其执行过程中不可中断也不可切换到其他线程上下文。原子操作从微观上根本阻止了访问数据的并发操作，使宏观生同时发生的访问在微观上按顺序一个一个地执行，使共享变量的访问从宏观上看起来是并发访问的，但从微观上却是顺序访问的。

一般来说，复杂的结构数据类型是只能通过原子操作来保证数据完整性的，实现原子操作的措施不外乎关中断、停止调度、或者用信号量保护，这几种方式的时间和空间开销都很

大。对于简单的基础数据类型来说，不使用原子变量能保证数据完整性吗？一个优秀的软件不应该全局变量漫天飞舞，即使要用全局变量，也应该限制全局变量的访问，不应该使多个线程同时具备读写权限，而是只有一个线程（或初始化代码）可写，其他线程只读。我们把这种一个地方可修改多线程可读的变量称做“树状共享变量”，这种变量像树一样，只有一个根部，可以吸收水和养料，可以有无数的树枝，树枝索取从根部吸收的养料，散页开花结果。对“树状共享变量”，无论并发读还是写，并不需要使用开销很大的原子操作，只需要保证数据是完整的就行了。只要保证变量的值从 A 改变到 B 的过程中，任何时候发生线程抢占，读取的数据要么是 A，要么是 B，绝对不会是 C 就可以了，且数据 B 能够正确地写入存储器；同样，在读的过程中任何时候被写操作抢占，把变量从 A 改为 B，也能够确保读取的数据非 A 即 B，B 能够正确地写入变量。那么，怎样才能确保数据完整性呢？我们看看下列代码在 32 位 CPU 和 16 位 CPU 上的执行情况。

```
uint32_t       b32;
b32 +=100;   //假设 b32 的原值是 0x0000ffff
```

在 32 位 risc 机上，执行过程为：

1、 取 b32 地址到寄存器 reg0

2、 取 reg0 地址的数值到寄存器 reg1，reg1 的值为 0x0000ffff

3、 reg1 = reg1+1，reg1 的值更新为 0x00010063

4、 把 reg1 存入 reg0 地址，本步骤单机器周期完成，不可中断。

上述过程的 4 个步骤无论在哪里被中断或者被高优先级线程打断，在中断或高优先级线程中读 b32，要么得到 0x0000ffff，要么得到 0x00010063，不会出现其他数值，而且完成第 4 步后，b32 的值也确实能够正确地被修改为 0x00010063。

而在 16 位机上，执行过程将是（假设 CPU 有足够的寄存器，小端配置）

1、 取 b32 地址到寄存器 reg0

2、 取 reg0 地址的数值到寄存器 reg1，reg0+2 地址的数值到寄存器 reg2，则 reg1 的值为 0xffff，reg2 的值是 0.

3、 [reg2：reg1] = [reg2：reg1]+100，[reg2：reg1]的值更新为 0x00010063

4、 把 reg1 存入 reg0 地址，本步骤单机器周期完成，不可中断。

5、 把 reg2 存入 reg0+2 地址，本步骤单机器周期完成，不可中断。

如果在第 4 和第 5 步骤之间被中断打断，在中断里读 b32，将得到错误的 0x00000063，而不是正确的 0x00010063。同样，如果在读的过程中读高 16 位和低 16 位数据之间发生被抢占并修改了变量，也不能读取正确的值，读者可以自行分析。

从上述例子可以看出，不使用原子操作而实现“树状共享变量”数据完整性的关键在于，无论完整的存取操作需要多少步骤，只要保证直接把数据存入存储器或从存储器中把数据取出的过程可以用单 CPU 指令完成就可以了。我们知道，无论哪一种 CPU，宽度小于或者等于 CPU 字宽的变量，是可以用单机器指令完成读或者写的，而等于 CPU 字长的整型变量的操作速度又是最快的。djyos 的 ucpu_t 和 scpu_t 类型变量就是等于 CPU 字长的有符号和无符号整数，读者可以放心地用这两个类型的变量用作“树状共享变量”。

“树状共享变量”在 djyos 系统中的典型应用是读取系统 tick 的函数，如下：

```
uint32_t y_get_time(void)
{
    uint32_t time;
#if (32 > cn_cpu_bits)
    int_save_line(cn_irq_line_timer5);
#endif
```

```
    time = u32g_os_ticks;
#if (32 > cn_cpu_bits)
    int_restore_line(cn_irq_line_timer5);
#endif
    return time;
}
```

变量 u32g_os_ticks 在 isr_tick 函数中不断增量，在应用程序的任何地方都可以读取这个变量，因此，如果 CPU 字长小于 32 位，则需要在读取的过程中关闭 tick 中断，以确保读的过程不会被 isr_tick 中断打断。如果 CPU 字长大于等于 32 位，则即使读过程中被 isr_tick 中断，也是安全的。

16.8 精简代码与阅读困难

高度技巧化的、精简的代码，阅读起来往往很困难，而浅显易读的代码，又可能要牺牲效率，程序设计者应该在两者中取得平衡。在少数效率至关重要的地方，使用技巧以照顾代码效率，而大量的、效率并不是那么重要的地方，还是要兼顾一致易读性。

djyos 系统大量链表操作，不管实际问题是否需要使用双向循环链表，所有这些链表都设计成循环链表，为什么要这样做呢？

假如不这么做，那么在代码的注释和软件文档中，所有链表都需要说明是循环链表还是线性链表，在处理每一个链表时，都需要关注该链表是循环的还是开口的，即使是内核开发者，也难于做到一一明辨，这就容易成为隐藏 bug 的温床。对于代码阅读者来说，情况会更加严重，他们每当读到使用链表的地方，往往需要回去看看文档，以确定链表是循环的还是开口的。因此，在绝大多数操作使用循环链表的情况下，我们宁可使代码稍微复杂些，从而使所有使用链表的操作一致。

还有一种广泛使用的做法，通过多次嵌套宏定义的方式，使最终代码看起来很简洁，也很直观，也很高效，但当读者阅读此代码时，需要逐个回溯宏定义以便将其展开，而且这种回溯可能在多个文件中进行，将是一件令人心烦的事情。

16.9 用户习惯

就像 unix 拒绝为用户提供友好的操作界面一样（参考著名的《unix 痛恨者手册》，从 unix 转型到嵌入式行业的程序员，也一样习惯于按照程序员的习惯行事。对他们来说，照顾用户使用习惯、为那些计算机呆子浪费时间不符合他们作为计算机软件高手的身份，他们会不屑地问：“为什么要照顾那些傻瓜”。然而，恰恰按照用户使用习惯来设置软件的人机界面，是提高软件可靠性的有效手段，而不是一句华丽的口号。记得不，**“想当然就是正确的”是 djyos 的信条之一**，所谓“想当然”就是用户可以按照自己的习惯去理解和使用软件。

笔者认为，操作系统的可靠性来自于两个方面，一方面是绝对可靠性，即操作系统本身应该没有 bug，只要用户正确使用，则操作系统的执行结果肯定是预期的。第二个方面在于应用程序的正确性，djyos 操作系统认为，编写正确的应用程序不仅仅是应用程序员的责任，也是操作系统的责任。操作系统和普通产品一样，使用方法应该符合人们的日常思维习惯，日常生活中有很多错误是由于人们按照“想当然”的方法去理解和行动造成的。我们不能设计一辆左脚刹车右脚离合的汽车，就算你用斗大的字把“左脚刹车”印在方向盘上，司机仍然会想当然地用右脚刹车，这不能怪司机没有遵守使用说明书。djyos 以“想当然就是正确

的”为设计原则，这样可以最大限度地降低程序员因不熟练或者没有仔细阅读说明书而导致错误，过分强调高效性可能会限制用户的行为，使一些符合日常思维习惯的操作变成非法的操作，应用程序员误操作导致错误时，我们不能一味责怪用户，操作系统本身应该保持易学易用，并且符合人们的日常思维习惯，保证用户编写正确的程序的重要性远胜于操作系统本身的执行效率。记住以下几条很重要：

- 降低低误操作可以减少 bug 爆发的机会。
- 遵循用户使用习惯可以减少误操作。
- 多数 bug 隐藏在非预期的操作中，因为这常常成为测试盲点，误操作很容易引爆一个 bug。
- 减少 bug 的绝对数量和减少 bug 爆发机会同等重要。
- 减少含 bug 的代码的执行次数可以有效防止 bug 爆发，从而提高可靠性。

16.10 空函数指针

C 语言编程，免不了使用函数指针，许多教科书和参考资料都教导程序员：不要把未初始化的变量留在内存中，要求把未初始化的指针变量赋值为 NULL，这当然也包括函数指针。NULL 是一个由预定义的常量，在绝大多数系统上，声明为：

```
#define NULL ((void *)0)  或
#define NULL 0
```

在桌面系统下，未初始化的函数被初始化为 NULL 是安全的，因为桌面 PC 在操作系统支持下，0 地址受到严密保护，应用程序如果调用地址为 0 的函数，操作系统会释出异常，告诉用户：你错了，因此不会导致严重问题。嵌入式环境下，许多情况下 0 地址并不受特别保护，特别是没有操作系统支持的环境尤其如此，调用值为 NULL 的函数指针可能导致系统复位。因此，把空函数指针初始化为 NULL 并不安全。

在 djyos 中，定义了一个空函数：

```
void NULL_func(void);
```

作者建议所有的在 djyos 操作系统下编程的程序员，把未初始化的函数指针赋值为 NULL_func，下列语句来自 djyos 泛设备驱动模块中 dev_add_device 函数：

```
if(right_write != NULL)
    new_device->right_hand.io_write = right_write;
else
    new_device->right_hand.io_write = NULL_func;
```

当用调试器调试软件时，可以在 NULL_func 函数处放置一个调试断点，也可以在该函数处编写一些调试代码，输出调试信息，一旦程序走到这里，就说明软件的某个地方调用了空函数指针，让程序停下来并收集记录有利于诊断的信息。这样做可以加快调试，也可以发现一些隐藏的 bug。

16.11 BOOL 变量的第三值

顾名思义 boolean 变量是只有真假两个值的，在 pascal 语言中也确实是这样的，但在 C 语言中也真的是这样吗？在 ANSI C 标准中，boolean 与 int 等类型不一样，并不是一个基础类型，也没有规定 boolean 变量是如何实现的，在具体实现中，编译器一般用宏定义的方法，把 boolean 类型宏展开成其他类型，一般地定义为：

typedef unsigned int　　bool;　或者

typedef enum { FALSE = 0, TRUE　= !FALSE } bool;

前者被定义为 int，显见它不止 2 个值，后者呢？难道也不止 2 个值吗？的确如此，下列代码可以顺利通过编译 gcc，编译器连警告都懒得给：

typedef enum { aa = 0,bb = 1} entest;

entest eee;

eee = 100;

为什么会这样呢，原来 C 语言在实现 enum 时，是由编译器自信选择合适的、可以容纳所有 enum 值的基础数据类型来实现的，也就是说，它可能是 char，也可能是 int，一切看写编译器作者的心情而定。

比如在 keil for ARM 中，bool 类型则五花八门，austriamicrosystems AG 公司的 ams 系列准备的头文件中，有

typedef sint bool_t; 和

typedef bool_t　BOOL;

给 STMicroelectronics 公司的 STR7 系列 ARM7 芯片准备的头文件中则有：

typedef enum { FALSE = 0, TRUE　= !FALSE } bool;

而同样是 STR7 系列 ARM7 芯片的 USB 应用库中，bool 又被定义为：

typedef unsigned char　BOOL;

当然，也有真的使用 1bit 来定义 bool 类型的，比如在 keil C51 编译器中，就有：

typedef　bit　BOOL;

所以，除非用 keil C51 编译，否则 if(bool_var == true)和 if（bool_var）是完全不同的，前者仅在 bool_var 值为 true（就是 1）的时候条件成立，而后者则只要 bool_var 不等于 0 就成立。

所以，正确无误的软件在正常运行时，boolean 变量确实只会有 true 和 false 2 个可能值，但是，如果软件有 bug，boolean 变量被非法指针修改，抑或被不怀好意的恶意代码修改，则可能出现 true 和 false 以外的值。事实上，我们可以利用 boolean 变量可能有第三值的特点，在软件中设置陷阱，捕捉 boolean 变量被非法修改的异常情况。由于 true 和 false 往往只是 boolean 类型变量值域的很小一个子集，被非法修改后其值仍然是 true 或 false 之一的可能性不大，这种捕捉方法的准确度是很高的。陷阱代码如下：

```
if(bool_var == true)
    ……;
else if(bool_var == false)
    ……;
else
    出错处理代码;
```

第17章　api 参考手册

17.1 事件与事件类型

1. 设置轮转调度时间片

函数原型：

void y_set_RRS_slice(uint32_t slices);

功能：设置轮转调度的时间片，所有轮流执行的线程的时间片都是相同的。slices 的单位是毫秒，将被向上取整为 ticks 时间的整数倍。如果设为 0 表示禁止轮转调度。

参数：slices，时间片，单位是毫秒。

返回：无。

2. 读取轮转调度时间片

函数原型：

uint32_t y_get_RRS_slice(void);

功能：读取轮转调度时间片，单位是毫秒。返回值可能与调用 y_set_RRS_slice 时的参数 slices 不一致，除非 ticks 正好是整数毫秒。

参数：无。

返回：轮转调度时间片，单位是毫秒。

3. 获取最后一次出错信息

函数原型：

uint32_t y_get_last_error(char *text);

功能：操作系统将记录软件运行过程中产生的出错信息，本函数返回最后一条信息，要取得更多的出错信息，可访问 log 文件或出错信息资源链表。

参数：text，用来返回出错信息字符串的指针。

返回：错误号。

4. 登记事件类型

函数原型：

```
uint16_t y_evtt_regist(bool_t mark,bool_t overlay,
                       ufast_t default_prio,
                       uint16_t vpus_limit,
                       void (*thread_routine)(struct event_script *),
                       uint32_t stack_size,
                       char *evtt_name);
```

功能： 登记一个事件类型到事件类型表中，djyos 系统只接受弹出登记过的事件类型。

参数：mark，true = 登记的是 mark 型事件，否则不是。

overlay，如果是 mark 型事件，true=该类型事件重复弹出时用新事件的参数覆盖老事件的参数，false = 不覆盖。

default_prio，事件类型的默认优先级，弹出该类型事件时，如果不指定优先级，则使用默认优先级。

vpus_limit，允许为本类型事件创建的线程总数。

thread_routine，用来处理本类型事件的函数。

stack_size，调用thread_routine函数所需要的栈，操作系统创建的用来执行thread_routine函数的线程的栈将比stack_size大，参见第 11.3.4 节。

evtt_name，给刚登记的事件类型起个名字，用于为不同的组件登记的事件类型能互相弹出事件。指针可以是 NULL，但只要不是 NULL，就不能与系统中存在的事件类型同名。

返回：成功则返回新登记的事件类型，失败则返回 cn_invalid_evtt_id。

5. 取事件类型 id

函数原型：

uint16_t y_get_evtt_id(char *evtt_name);

功能：在系统事件类型表中搜索名字是 evtt_name 的事件类型，返回其 id。不查找没名字的事件类型，如果事件类型表中有名字成员是 NULL 指针的事件类型，且 evtt_name 也等于 NULL，但不算匹配。

参数：evtt_name，搜索的目标事件类型名。

返回：找到则返回其事件类型，否则返回 cn_invalid_evtt_id。

6. 删除（取消登记）事件类型

函数原型：

bool_t y_evtt_unregist(uint16_t evtt_id);

功能： 把 id 号是 evtt_id 的事件类型注销，如果队列中还有该类型事件(in_use == 1)，只标记该类型为被注销，真正的删除工作是在 done 函数里完成的。如果队列中已经没有该类型事件了，将会执行真正的删除操作。无论哪种情况，此后系统均会拒绝弹出该类型的新事件。

参数：evtt_id，欲注销的事件类型 id。

返回：true=成功注销（含标记为注销），false=失败。

7. 检查是否允许调度

函数原型：

bool_t y_query_sch(void);

功能：异步信号、总中断开关均允许且运行于线程山下文时，允许调度。

参数：无。

返回：true = 允许调度，false = 禁止调度。

8. 清除 mark 状态

函数原型：

void y_clear_mark(void);

功能：参见 第 4 章第 4.3.13 节说明

参数：无。

返回：无。

9. 事件同步

函数原型：

uint32_t y_event_sync(uint16_t event_id,uint32_t timeout);

功能：把正在处理的事件置入 id 号为 event_id 的事件的同步队列中，随后进入阻塞态，待 event_id 事件处理完成后才激活。

参数：event_id，同步的目标事件。

timeout，如果在 timeout 毫秒后，event_id 事件仍不完成，将强行退出阻塞态。阻塞时间将被向上取整为整数个 ticks。

返回：cn_sync_success=同步条件成立返回。
cn_sync_timeout=超时返回。
cn_sync_error=出错。

10. 事件类型完成同步

函数原型：

uint32_t y_evtt_done_sync(uint16_t evtt_id,uint16_t done_times,uint32_t timeout);

功能：参见第 5.4 节。

参数：evtt_id，被同步的事件类型 id。
done_times，设定完成次数。
timeout，如果在 timeout 毫秒后，同步条件仍不达成，将强行退出阻塞态。阻塞时间将被向上取整为整数个 ticks。

返回：cn_sync_success=同步条件成立返回。
cn_sync_timeout=超时返回。
cn_sync_error=出错。

11. 事件类型弹出同步

函数原型：

uint32_t y_evtt_pop_sync(uint16_t evtt_id,uint16_t pop_times,uint32_t timeout);

功能：参见第 5.3 节。

参数：evtt_id，被同步的事件类型 id。
pop_times，设定完成次数。
timeout，如果在 timeout 毫秒后，同步条件仍不达成，将强行退出阻塞态。阻塞时间将被向上取整为整数个 ticks。

返回：cn_sync_success=同步条件成立返回。
cn_sync_timeout=超时返回。
cn_sync_error=出错。

12. 弹出事件

函数原型：

```
uint16_t y_event_pop(       uint16_t evtt_id,
                            uint32_t parameter0,
                            uint32_t parameter1,
                            ufast_t prio);
```

功能：弹出类型 id=evtt_id 的事件。弹出事件需要分配事件类型控制块，如果因没有空闲的事件类型控制块，本函数不会导致阻塞，直接返回 cn_invalid_event_id。

参数：evtt_id，弹出的事件类型 id，该类型必须是登记过的，否则空操作。
parameter0，事件参数 0。
parameter1，事件参数 1。
prio，事件的优先级，如果设为 0，则使用事件类型的默认优先级。

返回：成功弹出事件则返回事件 id，否则返回 cn_invalid_event_id。

13. 事件完成

函数原型：

void y_event_done(void);

功能：事件处理完成后，调用本函数，如果程序设计者没有调用本函数，则在线程入口函数返回后由系统代劳，参见第 4.3.11.3 节。

参数：无。

返回：无。

14. 查询唤醒原因

函数原型：

union event_status y_wakeup_from(void)；

功能：返回事件最后一次从阻塞状态的阻塞原因，如果事件从未阻塞，则返回 0。

参数：无。

返回：事件状态变量。

17.2 资源管理

系统提供了一系列的 api 函数用于资源结点的操作：

1. 添加根结点

函数原型：

```
struct  rsc_node * rsc_add_root_node(struct  rsc_node *node, uint32_t size,char *name)；
```

功能：在系统资源链表中添加一个根结点。

参数：node，待添加的结点，数据结构由调用者提供，但调用者无需初始化它。

name，待添加的结点名字。

返回：新添加的结点指针。

2. 插入资源结点

函数原型：

```
struct  rsc_node * rsc_insert_node(struct  rsc_node *node,
                                   struct rsc_node *new_node,
                                   uint32_t size,char *name)；
```

功能：在资源链表中指定项前面插入一个结点，若指定结点是长子，则成为新的长子。

参数：node，插入位置结点，新结点插入到 node 的前面。

new_node，待插入的结点，数据结构由调用者提供，但调用者无需初始化它。

name，待添加的结点名字。

返回：新插入的结点指针。

3. 增加资源结点

函数原型：

```
struct  rsc_node * rsc_add_node(struct  rsc_node *node,
                                struct  rsc_node *new_node,
                                uint32_t size,char *name)；
```

功能：在资源链表中指定项后面增加一个结点，若指定结点是满子,则成为新的满子。

参数：node，插入位置结点，新结点插入到 node 的后面。

new_node，待插入的结点，数据结构由调用者提供，但调用者无需初始化它。

name，待添加的结点名字。

返回：新添加的结点指针。

4. 增加子资源结点

函数原型：

```
struct  rsc_node * rsc_add_son(struct  rsc_node *parent_node,
                               struct  rsc_node *new_node,
                               uint32_t size,char *name);
```

给指定结点增加一个子结点，该结点将加入到所有子结点的最后面，即满子结点,

5. 插入长子资源结点

函数原型：

```
struct  rsc_node * rsc_add_eldest_son(struct  rsc_node *parent_node,
                                      struct  rsc_node *new_node,
                                      uint32_t size,char *name);
```

功能：给指定结点增加一个子结点，并且使该结点成为长子结点。

参数：parent_node，插入位置的父结点。

new_node，待插入的结点，数据结构由调用者提供，但调用者无需初始化它。

name，待添加的结点名字。

返回：新添加的结点指针。

6. 删除一个结点

函数原型：

```
bool_t rsc_del_node(struct  rsc_node *node);
```

功能：把一个结点从资源链表中断开结点，该结点不能有子结点。注意只是断开链表，并不释放内存，因为资源管理模块并不分配和释放结点内存。

参数：node，待删除的结点。

返回：true=成功删除，false=有子结点，不能删除。

7. 移动一个资源树枝

函数原型：

```
void rsc_moveto_tree(struct rsc_node *parent,struct  rsc_node *node);
```

功能：把一个树枝从资源结点队列中移除，然后添加到另一个结点下成为其子树。

参数：node，待移动的子树。

parent_node，插入目标位置的父结点。

返回：无。

8. 移动结点成为小弟结点

函数原型：

```
void rsc_moveto_least(struct  rsc_node *node);
```

功能：移动一个结点，使该结点成为小弟结点，该结点本来应在链表中。

参数：node，被移动的结点。

返回：无。

9. 移动结点成为长兄结点

函数原型：

```
void rsc_movto_eldest(struct  rsc_node *node);
```

功能：移动一个结点，使该结点成为长兄结点，该结点本来应在链表中。

参数：node，被移动的结点。

返回：无。

10. 移动结点到某结点前面

函数原型：

```
void rsc_movto_elder(struct  rsc_node *lesser,struct  rsc_node *node);
```

功能：移动一个结点到某结点前边成为兄结点，该结点本来应在链表中。

参数：node，被移动的结点。

lesser，目标位置的参考结点。

返回：无。

11. 移动结点到某结点后面

函数原型：
void rsc_movto_lesser(struct rsc_node * elder,struct rsc_node *node)；
功能：移动一个结点到某结点后面成为弟结点，该结点本来应在链表中。
参数：node，被移动的结点。
elder，目标位置的参考结点。
返回：无。
12. 长子位置后移
函数原型：
void rsc_round_back(struct rsc_node *parent)；
功能：同辈结点的相对位置不变，但是长子位置后移一格，即把满子变成长子。
参数：parent，被移动的结点的父指针。
返回：无。
13. 长子位置前移
函数原型：
void rsc_round_forward(struct rsc_node *parent)；
功能：同辈结点的相对位置不变，但是长子位置前移一格，即把次子变成长子。
参数：parent，被移动的结点的父指针。
返回：无。
14. 资源结点改名
函数原型：
void rsc_rename_node(struct rsc_node *node,char *new_name)；
功能：修改一个资源结点的名字。
参数：node，被修改的结点。
new_name，新名字，允许是空串或空指针。
返回：无。
15. 返回根结点
函数原型：
struct rsc_node *rsc_get_root(struct rsc_node * scion_node)；
功能：查询指定结点的根结点，返回根结点指针。
参数：scion_node，被查询的任一后代结点指针。
返回：根结点指针，如果 son_node 已经是根结点，则返回 NULL。
16. 查询父结点
函数原型：
struct rsc_node *rsc_get_parent(struct rsc_node *son_node)；
功能：查询指定结点的父结点，返回父结点指针。
参数：son _node，被查询的子结点指针。
返回：父结点指针。
17. 查询长子结点
函数原型：
struct rsc_node *rsc_get_son(struct rsc_node *parent_node)；
功能：查询指定结点的直接子节点，即长子结点，返回长子结点指针。
参数：parent _node，被查询结点的父结点指针。
返回：长子结点指针。
18. 查询弟结点

函数原型：

struct　rsc_node *rsc_get_next(struct　rsc_node *elder_node)；

功能：查询指定结点的弟结点，返回弟结点指针。

参数：elser _node，被查询结点的兄结点指针。

返回：弟结点指针。

19. 查询树枝一个末梢结点

函数原型：

struct　rsc_node *rsc_get_twig(struct　rsc_node *parent_node)；

功能：返回某树枝的一个末梢结点，末梢结点是指没有子结点的结点。当需要删除整个树枝时，本函数很有用，结合 rsc_del_node 函数，反复调用本函数并把返回的结点删除，直到本函数返回 NULL，就可以删除一个资源树枝。需要删除一个文件夹或者删除一个存在子窗口的 gui 窗口时，就需要用到删除整个树枝的操作。

参数：parent _node，此结点的子结点组成的资源树中查询。

返回：被查询树枝的第一个末梢结点。

20. 遍历一个树枝

函数原型：

struct　rsc_node *rsc_trave_scion(struct　rsc_node *parent_node,
　　struct　rsc_node *current_node)；

功能：从当前结点开始，获取下一个结点的指针，沿着搜索路线，直到搜索完整个树枝。搜索顺序：当前结点的子结点，如果子结点为空则搜索弟结点，弟结点又为空则搜索父结点的弟结点,直到搜索完成。当需要对资源链表中某一个树枝或者整个链表中的结点逐一进行某中操作时，可反复调用本函数，函数的返回指针作为下一次调用的当前结点，直到返回空。

参数：parent _node，此结点的子结点组成的资源树中遍历。

　　current_node，遍历过程的当前结点。

返回：被查询树枝中 current_node 的下一个结点。

21. 在兄弟结点中搜索资源

函数原型：

struct　rsc_node *rsc_search_sibling(struct　rsc_node *brother,char *name)；

功能：在某一个资源结点的所有兄弟结点中搜索给定名字的资源。

参数：brother，在此结点的兄弟结点中搜索名为 name 的资源。

　　name，被查询的结点名字，允许是空字符串，但不能是 NULL。

返回：找到的第一个结点指针，否则返回 NULL。

22. 某结点的子结点中搜索资源名

函数原型：

struct rsc_node *rsc_search_scion(struct rsc_node *ancestor_node,char *name)；

功能：在某一个资源结点的所有子结点（不包含孙结点及更低级的结点）中搜索给定名字的资源。

参数：ancestor，在此结点的子结点中搜索名为 name 的资源。

　　name，被查询的结点名字，允许是空字符串，但不能是 NULL。

返回：找到的第一个结点指针，否则返回 NULL。

23. 在整个树枝中搜索资源名

函数原型：

struct rsc_node *rsc_search_scion(struct rsc_node *ancestor_node,char *name)；

功能：在某一个资源结点的所有后代结点中搜索给定名字的资源。

参数：ancestor，在此结点的后代结点中搜索名为 name 的资源。
name，被查询的结点名字，允许是空字符串，但不能是 NULL。
返回：找到的第一个结点指针，否则返回 NULL。
24. 沿路径搜索资源名
函数原型：
struct rsc_node *rsc_search(struct rsc_node *ancestor_node,char *path);
功能：path 是含路径名的资源名，从 ancestor 结点开始，沿着 path 参数指定的路径搜索，直到到达资源名。
参数：ancestor，在此结点的后代结点中搜索名为 name 的资源。
path，被查询的路径名字，允许是空字符串，但不能是 NULL。
返回：找到的资源结点指针，否则返回 NULL。

17.3 泛设备管理

1. 添加设备
函数原型：
```
struct  pan_device *dev_add_device(struct  pan_device *parent_device,
                                   char                  *name,
                                   struct semaphore_LCB  *right_semp,
                                   struct semaphore_LCB  *left_semp,
                                   dev_write_func        right_write ,
                                   dev_read_func         right_read,
                                   dev_ctrl_func         right_ctrl ,
                                   dev_write_func        left_write ,
                                   dev_read_func         left_read ,
                                   dev_ctrl_func         left_ctrl );
```
功能：新添加的设备成为 parent_device 设备的子设备，如果 parent_device 已经有子设备，则新设备添加在兄弟设备的尾部称为满子设备。6 个函数指针是设备的全部对外接口，并不是所有接口都需要实现，有些设备仅用了其中少数接口。许多嵌入式系统中 NULL 指针是合法的函数地址，但是调用 NULL 函数指针可能引起不可预料的操作，最可能的是复位。因此，没有实现的函数请不要用 NULL 指针代替，而是用操作系统提供的空函数 api 调用 NULL_func。
参数：parent_device，待添加设备的父设备指针
name，新设备的名字，设备名不能为空，也不能包含字符'\'，因为'\'是资源路径（这里是设备路径）分隔符。
right_semp，设备右手接口信号量指针
left_semp，设备左手接口信号量指针
right_write，右手写函数指针
right_read，右手读函数指针
right_ctrl，右手控制函数指针
left_write，左手写函数指针
left_read，左手读函数指针
left_ctrl，左手控制函数指针
返回：本函数返回新增加的设备的指针，如果资源不足不能添加设备，则返回 NULL。

2. 添加根设备

函数原型：

```
struct  pan_device *dev_add_root_device(    char            *name,
                                            struct semaphore_LCB *right_semp,
                                            struct semaphore_LCB *left_semp,
                                            dev_write_func   right_write ,
                                            dev_read_func    right_read,
                                            dev_ctrl_func    right_ctrl ,
                                            dev_write_func   left_write ,
                                            dev_read_func    left_read ,
                                            dev_ctrl_func    left_ctrl );
```

功能：本函数直接调用 dev_add_device 函数添加一个以 pg_device_root 为父设备的设备，即根设备。为什么要设计这个函数呢？ 由用户直接调用 dev_add_device 函数不是更快捷吗？因为 pg_device_root 虽是全局变量，但被定义为在 driver 模块内使用的静态变量，其他模块是不能访问的，否则，就违背了模块独立性的原则。

参数：参见 dev_add_device 函数。

返回：本函数返回新增加的设备的指针，如果资源不足不能添加设备，则返回 NULL。

3. 卸载设备

函数原型：

```
bool_t dev_delete_device(struct  pan_device * device);
```

功能: 把设备从设备链表中删除，并释放其占用的内存，待释放的设备需符合以下条件：

a.该设备没有子设备。

b.正在使用该设备的用户数为 0。

参数：device，待释放的设备

返回: 成功释放返回 true，否则返回 false。

4. 打开设备左手接口

函数原型：

```
struct  dev_handle *dev_open_left(char *name,uint32_t timeout);
```

功能：打开名为 name 的设备的左手接口，搜索整个设备资源树，找到名称与 name 匹配的资源结点，含该结点的设备即是目标设备，如果该设备的左手用户数已经达到最大限值，则不允许打开。设备打开后，左手用户数加 1。

参数：name，设备名字符串，包含路径名，但不包含'dev\'这 4 个字符，沿设备资源树逐级索引值到目标设备，各级路径之间用字符’\’隔开。

timeout，遇忙等待毫秒数，若设备被占用则阻塞线程，如果 timeout 毫秒以后仍然忙，则返回 false。阻塞时间将被向上取整为整数个 ticks。

返回：成功打开设备返回设备句柄，否则返回 NULL。

5. 快速打开设备的左手接口

函数原型：

```
bool_t dev_open_left_again(struct  dev_handle *handle,uint32_t timeout);
```

功能: 在已知设备的左手句柄的情况下，快速打开设备的左手接口。适用于设备曾经被打开左手接口，设备被关闭但一直没有被删除且用户没有修改 handle 指针的情况。在设备资源树中查找路径和名字匹配的设备需要多次比较字符串，是非常费时的操作，在已知句柄的情况下，不用按设备名字符串查找设备，速度很快。如果有多个用户共享某设备，为了充分共享，每次使用完毕后应该关闭设备，但是不设备 handle 指针，也不删除设备，当需要

再次打开时，可以使用本函数。

参数：handle，设备左手 handle 指针，句柄不分左右手。

timeout，遇忙等待毫秒数，若设备被占用则阻塞线程，如果 timeout 毫秒以后仍然忙，则返回 false。阻塞时间将被向上取整为整数个 ticks。

参数: left，设备左手 handle 指针，句柄不分左右手。

返回: 成功打开设备则返回 true，否则返回 flase。

6. 打开后代设备左手接口

函数原型：

```
struct    dev_handle *dev_open_left_scion(struct    dev_handle *ancestor,
                                          char *scion_name, uint32_t timeout);
```

功能：打开后代设备的左手接口，搜索 ancestor 设备的整个子设备树，找到名称与 scion_name 匹配的资源结点，然后从泛设备句柄池中分配控制块，把左手接口指针赋值给句柄后，返回该句柄指针。

参数：ancestor，被搜索的设备树的祖先设备句柄，句柄不分左右手。

name，设备名字符串，包含路径名，但不包含'dev\'这 4 个字符，沿设备资源树逐级索引值到目标设备，各级路径之间用字符'\'隔开。

timeout，遇忙等待毫秒数，若设备被占用则阻塞线程，如果 timeout 毫秒以后仍然忙，则返回 false。阻塞时间将被向上取整为整数个 ticks。

返回：成功打开设备返回设备句柄，否则返回 NULL。

7. 打开设备右手接口

函数原型：

```
struct    dev_handle *dev_open_right(char *name,uint32_t timeout);
```

参见“打开设备左手接口”的说明。

8. 快速打开设备右手接口

函数原型：

```
bool_t dev_open_right_again(struct    dev_handle *handle,uint32_t timeout);
```

参见“快速打开设备左手接口”的说明。

9. 打开后代设备右手接口

函数原型：

```
struct    dev_handle *dev_open_left_scion(struct    dev_handle *ancestor,
                                          char *scion_name, uint32_t timeout);
```

参见“打开后代设备左手接口”的说明

10. 关闭设备左手接口

函数原型：

```
bool_t dev_close_left(struct    dev_handle *handle);
```

功能：关闭设备左手接口，把设备的用户数减 1，其他不变，本函数只关闭设备接口，如果具体设备要求额外的操作清理设备状态，应用程序应该在关闭设备前调用设备的 ctrl 函数。为了将来可以使用 dev_open_left_again 函数，本函数并没有把 left 指针置空，而是使其指向被关闭的设备的设备控制块。

参数：handle，被关闭的设备句柄，虽然设备类型不分左右手，但该句柄须是调用打开左手设备的函数返回的，否则本函数不执行任何操作。

返回：成功关闭返回 true，否则返回 false。

11. 关闭右手设备

函数原型：

bool_t dev_close_right(struct dev_handle *handle)；

说明：参见“关闭设备左手接口”的说明。

12. 读设备函数

函数原型：

ptu32_t dev_read(struct dev_handle *handle,ptu32_t src_buf, ptu32_t des_buf,ptu32_t len);

功能：从设备读数据的接口函数，无论是左手设备还是右手设备，都调用统一的 dev_read 函数。在设备句柄的成员 iam 的引导下，确定调用设备 driver 提供的 left_read 函数还是 right_read 函数。如果 handle 是调用打开左手接口的函数打开的设备句柄，就调用 left_read 函数，否则调用 right_read 函数。

参数：handle，被读的设备的句柄，句柄不分左右手。

src_buf，参数 1，用途由 driver 决定，并在设备说明书中说明。

des_buf，参数 2，用途由 driver 决定，并在设备说明书中说明。

len，参数 3，用途由 driver 决定，并在设备说明书中说明。

返回：读取操作的结果，含义由 driver 定义，并在设备说明书中说明。

13. 写设备函数

函数原型：

ptu32_t dev_write(struct dev_handle *handle,ptu32_t src_buf, ptu32_t des_buf,ptu32_t len);

功能：写数据到设备的接口函数，无论是左手设备还是右手设备，都调用统一的 dev_write 函数。在设备句柄的成员 iam 的引导下，确定调用设备 driver 提供的 left_write 函数还是 right_write 函数。如果 handle 是调用打开左手接口的函数打开的设备句柄，就调用 left_write 函数，否则调用 right_write 函数。

参数：handle，被读的设备的句柄，句柄不分左右手。

src_buf，参数 1，用途由 driver 决定，并在设备说明书中说明。

des_buf，参数 2，用途由 driver 决定，并在设备说明书中说明。

len，参数 3，用途由 driver 决定，并在设备说明书中说明。

返回：写入操作的结果，含义由 driver 定义，并在设备说明书中说明。

14. 设备控制函数

函数原型：

ptu32_t dev_ctrl(struct dev_handle *handle,uint32_t cmd, ptu32_t data1,ptu32_t data2)；

功能：控制设备的接口函数，无论是左手设备还是右手设备，都调用统一的 dev_ ctrl 函数。在设备句柄的成员 iam 的引导下，确定调用设备 driver 提供的 left_ ctrl 函数还是 right_ ctrl 函数。如果 handle 是调用打开左手接口的函数打开的设备句柄，就调用 left_ ctrl 函数，否则调用 right_ ctrl 函数。

参数：handle，被读的设备的句柄，句柄不分左右手。

cmd，参数 1，用途由 driver 决定，并在设备说明书中说明。

data1，参数 2，用途由 driver 决定，并在设备说明书中说明。

data2，参数 3，用途由 driver 决定，并在设备说明书中说明。

返回：控制操作的结果，含义由 driver 定义，并在设备说明书中说明。

15. 清理设备

函数原型：

void dev_cleanup(struct event_script *event)；

功能： 关闭事件 event 的 held_device 指针成员指向的所有设备，这是一个安全函数，正常情况下，用户使用设备时应该打开和关闭设备成对使用，否则可能造成资源泄漏。在事件处理完成线程结束前，操作系统检查事件控制块的 held_device 操作系统将调用本函数。

参数：handle，被操作的设备句柄，句柄不分左右手。
返回：无

17.4 中断

1. 保存总中断开关状态并禁止总中断
函数原型：
void int_save_trunk(void);
功能：把总中断使能计数器加 1，并禁止总中断。
参数：无。
返回：无。
2. 恢复保存的总中断开关状态
函数原型：
void int_restore_trunk(void);
功能: 把总中断使能计数器减 1，减到 0 则不能再减。如果减量后使能计数器=0，则允许总中断开关，如果此时异步信号也处于使能状态，将可能引发一次上下文切换，视在中断禁止期间有没有改变就绪事件队列。
参数：无。
返回：无。
3. 查看总中断是否允许
函数原型：
bool_t int_check_trunk(void);
功能: 查看总中断是否允许
参数：无
返回: 总中断被允许则返回 true，否则返回 false。
4. 保存异步信号开关状态并禁止异步信号
函数原型：
void int_save_asyn_signal(void);
功能：异步信号开关的 en_asyn_signal_counter 加 1，并禁止异步信号。
参数：无。
返回：无。
5. 恢复保存的异步信号开关状态
函数原型：
void int_restore_asyn_signal(void);
功能: 把异步信号使能计数器减 1，减到 0 则不能再减。如果减量后使能计数器=0，则允许异步信号开关，如果此时总中断也处于使能状态，将可能引发一次上下文切换，视在中断禁止期间有没有改变就绪事件队列。
参数：无。
返回：无。
6. 查看异步信号是否允许
函数原型：
bool_t int_check_asyn_signal (void);
功能: 查看异步信号是否允许

参数：无

返回: 异步信号被允许则返回 true，否则返回 false。

7. 保存某中断线开关状态并禁止该中断线

函数原型：

void int_save_line(ufast_t ufl_line);

功能：把中断线使能开关的使能计数器加 1，并禁止该中断线。本函数不区分该中断线是异步信号还是实时中断。

参数：ufl_line，被操作的中断线。

返回：无。

8. 恢复保存的中断线开关状态

函数原型：

void int_restore_line(ufast_t ufl_line);

功能: 把中断线使能计数器减 1，减到 0 则不能再减。如果减量后使能计数器=0，则允许该中断线。

参数：ufl_line，被操作的中断线。

返回：无。

9. 直接禁止某中断线

函数原型：

void int_disable_line(ufast_t ufl_line);

功能：本函数是 int_enable_line（）的姊妹函数，调用本函数使中断线的使能计数器置位，并掐断中断线。

参数：ufl_line，被操作的中断线。

返回：无。

特别备注：总中断开关和异步信号开关不提供直接使能和禁止的功能。

10. 恢复保存的中断线开关状态

函数原型：

void int_enable_line(ufast_t ufl_line);

功能: 本函数是 int_ disable _line（）的姊妹函数，调用本函数使中断线的使能计数器清零，并允许中断线。

参数：ufl_line，被操作的中断线。

返回：无。

特别备注：总中断开关和异步信号开关不提供直接使能和禁止的功能。

11. 查看一个中断线是否允许

函数原型：

bool_t int_check_line(ufast_t ufl_line);

功能: 查看一个中断线是否允许，无论该中断线被设为异步信号还是实时中断，都可以用这个函数查询。

参数：ufl_line，被查询的中断线编号

返回: 中断线被允许则返回 true，否则返回 false。

12. 查看一个中断线状态

函数原型：

bool_t int_query_line(ufast_t ufl_line);

功能: 查看一个中断线的中断信号是否悬挂，若悬挂则返回 true，并清除中断信号，无论该中断线被设为异步信号还是实时中断，都可以用这个函数查询。

参数：ufl_line，被查询的中断线编号

返回: 中断信号悬挂则返回 true，否则返回 false。

13. 允许异步信号嵌套

函数原型：

void int_enable_nest_asyn_signal (void)

功能: 异步信号响应后，操作系统默认不允许异步信号嵌套，如果用户需要允许异步信号嵌套，可以调用本函数打开异步信号嵌套功能。允许异步信号嵌套，请注意中断栈空间是否足够，异步信号栈空间应该大于所有可能嵌套发生的异步信号服务函数的栈需求的总和。中断嵌套是否能够实现，有赖于硬件支持，这个函数和禁止中断嵌套的函数 int_disable _nest_asyn_signal 无需成对使用。

参数：无。

返回：无。

14. 禁止异步信号嵌套

函数原型：

void int_disable _nest_asyn_signal (void)

功能：异步信号响应后，操作系统默认不允许异步信号嵌套，用户调用 int_enable_asyn_signal 函数打开异步信号嵌套功能后。要重新关闭异步信号嵌套，才需要使用本函数。中断嵌套是否能够实现，有赖于硬件支持，这个函数和允许中断嵌套的函数 int_enable _nest_asyn_signal 无需成对使用。

参数：无。

返回：无。

15. 允许实时中断嵌套

函数原型：

void int_enable_nest_real(void)

功能: 参见允许异步信号嵌套 int_enable_nest_asyn_signal。

参数：无。

返回：无。

16. 禁止异步信号嵌套

函数原型：

void int_disable _nest_real (void)

功能：参见禁止异步信号嵌套 int_disable_nest_asyn_signal。

参数：无。

返回：无。

17. 模拟触发中断

函数原型：

bool_t int_tap_line(ufast_t ufl_line);

功能：模拟一个中断线发出中断请求的情形，这个功能依赖于具体硬件，在某些硬件平台上可能不允许软件触发中断，或者只有部分中断线允许软件触发。

参数：ufl_line，被操作的中断线。

返回：如果该中断线允许软件触发并成功触发，返回 true，否则返回 flase。

18. 把某中断线设为异步信号

函数原型：

void int_setto_asyn_signal(ufast_t ufl_line));

功能: 看 图 6-5，把某中断线从设定到异步信号组中。

参数：ufl_line，被操作的中断线。

返回：无。

19. 把某中断线设为实时中断

函数原型：

void int_setto_real(ufast_t ufl_line))；

功能: 看 图 6-5，把某中断线从设定到实时中断组中。

参数：ufl_line，被操作的中断线。

返回：无。

20. 设定中断线的嵌套优先级

函数原型：

void int_set_nest_prio(ufast_t ufl_line)；

功能: 嵌套优先级是指：当允许嵌套时，嵌套优先级高的中断可以抢占正在服务的嵌套优先级低的中断。

参数：ufl_line，被操作的中断线。

返回：无。

21. 设定中断线的子优先级

函数原型：

void int_set_sub_prio(ufast_t ufl_line)；

功能: 子优先级是指：当嵌套优先级相同的多个中断同时悬挂时，先服务子优先级高的中断。

参数：ufl_line，被操作的中断线。

返回：无。

22. 关联 ISR 和中断线

函数原型：

void int_isr_connect(ufast_t ufl_line, void (*isr)(ufast_t))；

功能: 系统初始化后，除 tick 中断外，中断服务函数指针全部指向空函数。本函数为中断线设定中断服务函数，当中断发生后，如果中断被允许，操作系统将自动调用该中断服务函数。无论实时中断还是异步信号，都用相同的方法设定。isr 是普通的 C 函数指针。

参数：uflline，被连接的中断线，

isr，中断服务函数指针。

返回：无。

23. 解除 ISR 和中断线的关联

函数原型：

void int_isr_disconnect(ufast_t ufl_line)；

功能: 解除中断线与中断服务函数的关联，使该中断线的服务函数指针指向空函数，事件类型 id 成员设为 cn_invalid_evtt_id。

参数：uflline，被操作的中断线，

返回：无。

24. 关联事件类型和中断线

函数原型：

void int_evtt_connect(ufast_t ufl_line,uint16_t my_evtt_id)；

功能: 系统初始化后，任何中断线在中断返回前，系统不会自动弹出事件，调用本函数把事件类型与中断线相关联后，在该中断线中断返回前，中断管理程序会自动弹出该类型的事件。只有被设置为异步信号的中断线才会弹出事件，若调用本函数时，该中断线被设置成

实时中断，则只有在后续操作把它设置成异步信号时，事件类型才生效。

参数：uflline，被连接的中断线，

my_evtt_id，被关联的事件类型 id

返回：无。

25. 解除事件类型和中断线的关联

函数原型：

void int_evtt_disconnect(ufast_t ufl_line);

功能: 解除中断线与事件类型的关联，是该中断线的中断服务函数返回前，系统不会自动弹出原来所关联的事件类型。

参数：uflline，被操作的中断线，

返回：无。

26. 设定异步信号同步

函数原型：

bool_t int_asyn_signal_sync(ufast_t ufl_line);

功能: 阻塞正在处理的事件的线程，直到指定的中断线的中断发生、响应并返回后才激活，该中断线必须是异步信号，否则无效，每个中断线只接受一个事件同步。注意，该中断线只要设为异步信号并被允许就可以，允许不经 int_isr_connect 连接 ISR 函数。

参数：uflline，被操作的中断线，

返回：true=设置同步成功，false=失败，失败的原因一般是因为该中断线已经被别的事件同步，或者该中断线是实时中断。

17.5 内存分配（从堆中分配）

1. 分配一个局部内存块

函数原型：

void *m_malloc(ptu32_t size,uint32_t timeout);

功能: 从堆中分配内存块，size是字节数，由于djyos系统使用块相联分配算法，size将被向上取整为一个规格化的块，规格化方法参见第 7.2.1 节。本函数分配的内存是局部内存，在调用者事件的处理期内有效，程序设计者应该及时调用m_free函数释放内存，否则，事件处理完毕时将由操作系统强行释放以避免内存泄漏。这种强制释放是很消耗时间的，实时软件的设计者必须避免这种事情发生。欲在事件生存期外使用，请调用m_malloc_gbl函数分配全局内存块。如果没有合适的内存块可供分配，且timeout不等于 0，则阻塞线程，知道timeout时间到或者有其他线程释放内存致使有合适的内存块。

参数：size，欲分配的内存尺寸。

timeout，如果在 timeout 毫秒后，仍无合适的内存块，将强行退出阻塞态。阻塞时间将被向上取整为整数个 ticks。

返回：获得的内存块指针，分配失败则返回 NULL。

2. 分配一个全局内存块

函数原型：

void *m_malloc_gbl(ptu32_t size,uint32_t timeout);

功能: 分配一个全局内存块，该内存块不限于在申请者事件处理过程内使用，操作系统不对该内存块做强行释放操作，其余同 m_malloc 函数。

参数：size，欲分配的内存尺寸。

timeout，如果在 timeout 毫秒后，仍无合适的内存块，将强行退出阻塞态。阻塞时间将被向上取整为整数个 ticks。

返回：获得的内存块指针，分配失败则返回 NULL。

3. 查询给定的内存块的实际尺寸

函数原型：

ptu32_t m_check_size(void * mp);

功能: 由于内存分配函数要对 size 参数做规格化调整，malloc 函数实际获得的内存块可能比申请量大许多，本函数查询获得的内存块的实际可用尺寸。

参数：mp，被查询的内存块指针。

返回：内存块的实际尺寸。

4. 释放一块内存

函数原型：

bool_t m_free(void * pl_mem);

功能: 把 pl_mem 指向的内存块释放会堆中。

参数：pl_mem，待释放的内存块指针。

返回：true = 成功释放，false=失败。

5. 查询最大空闲内存块尺寸

函数原型：

ptu32_t m_get_max_free_mem(void);

功能: 查询最大空闲内存块尺寸

参数：无。

返回：最大空闲内存块尺寸。

6. 查询总堆空间大小

函数原型：

ptu32_t m_get_heap_size(void);

功能: 查询系统堆空间大小

参数：无。

返回：对尺寸字节数。

7. 查询空闲内存尺寸

函数原型：

ptu32_t m_get_free_mem(void);

功能: 查询空闲内存尺寸，由于可能存在碎片，最大空闲内存块可能要小些

参数：无。

返回：空闲内存字节数。

8. 查询页尺寸呢

函数原型：

ptu32_t m_get_page_size(void);

功能: 实际内存分配的最小单位是页。本函数返回页尺寸。

参数：无。

返回：页字节数。

17.6 内存分配（从内存池中分配）

1. 创建内存池

函数原型：

struct mem_cell_pool *mb_create(void *pool_original,uint32_t capacital,
uint32_t cell_size,char *name)；

功能：创建一个内存池，以备随后从 pool_original 池中分配内存。如果调用者提供的 cell_size 和 pool_original 不符合系统的对齐要求，将不执行创建操作。

参数：pool_original，字节池。

capacital，内存池包含的块数。

cell_size，内存池的块尺寸（字节数）。

name，内存池的名字，创建后，内存池将出现在系统资源链表中，name 就是链表结点名字。

返回：新创建的内存池指针。

2. 分配一块内存

函数原型：

void *mb_malloc(struct mem_cell_pool *pool,uint32_t timeout)；

功能：从内存池中分配一块内存，一次只能分配一块。如果没有空闲内存块，则阻塞知道 timeout 时间到或别的线程释放内存块。

参数：pool，从 pool 池中分配，pool 应该是 mb_create 函数返回的指针。

timeout，如果在 timeout 毫秒后，仍无空闲内存块，将强行退出阻塞态。阻塞时间将被向上取整为整数个 ticks。

返回：分配的内存块指针，分配失败则返回 NULL。

3. 释放内存块

函数原型：

void mb_free(struct mem_cell_pool *pool,void *block)；

功能：把 block 指向的内存块返还到内存池 pool 中。

参数：pool，接收被释放内存块的内存池。

block，被释放的内存池。

返回：无。

17.7 定时

1. 10 微秒粒度延时

函数原型：

void y_delay_10us(volatile uint16_t time)；

功能：延时 time×10 微秒，最多延时约 655 毫秒。本函数用指令循环实现的微秒分辨率延时，首次使用本函数前,请先调用 SetDelay 函数，否则在不同优化级别和不同编译器下，延时数不同。本函数执行时，处理器在空转，不建议用本函数做长延时，长延时用闹钟同步函数 y_timer_sync。

参数：time，延时时间量，单位是 10 微秒。

返回：无。

2. 闹钟同步

函数原型：

uint32_t y_timer_sync(uint32_t u32l_mS)；

功能：把正在处理的事件延迟 u32l_mS 毫秒后继续执行，延迟时间将向上调整为 tick 时间的整数倍。把 u32l_mS 设为 0 有特殊的意义，如果就绪队列中还有与正在处理的事件

优先级相同的其他事件，将按轮转调度的规则旋转一圈，正在处理的事件在就绪队列中的位置将调整到同优先级的最后，原处第二位置的将上升到第一位置，依次递推之。

参数：u32l_mS，延迟时间，毫秒数。

返回：实际延迟时间，一般来说应该等于 u32l_mS 按 tick 时间取整后的毫秒数，但是，因优先级抢占或关中断等因素，可能会附加额外的延时。

3. 获取当前 ticks 时钟

函数原型：

uint32_t y_get_time(void);

功能：操作系统运行时，有一个自由运行的 32 位时钟，该时钟达到 0xffffffff 变转回 0，本函数就是获取该时钟的当前值。

参数：无。

返回：当前时钟值。

4. 获取当前精密时钟

函数原型：

uint32_t y_get_fine_time(void);

功能：该时钟也是一个 32 位自由运行的时钟，定时精度由 port_kernel.h 中的常量 cn_fine_us 和 cn_fine_hz 表示。该时钟在每次 ticks 时钟改变的时候清零，重新计算。

参数：无。

返回：当前精密时钟值。

17.8 锁（信号量和互斥量）

1. 创建一个信号量

函数原型：

```
struct semaphore_LCB *sem_create(uint32_t lamps_limit,
                                 uint32_t init_lamp,char *name);
```

功能：从锁控制块池中分配一个锁控制块作为新创建的信号量控制块，然后初始化并加入系统资源队列中。如果在分配锁控制块失败将直接返回 NULL。

参数：lamps_limit，信号量包含的信号灯总数。

init_lamp，初始点亮的信号灯数量。

name，信号量的名字，信号量是系统资源队列中的一个资源，name 就是资源结点的名字。

返回：刚创建的信号量指针，失败则返回 NULL。

2. 点亮一盏信号灯

函数原型：

void sem_post(struct semaphore_LCB *semp);

功能：如果没有事件在等待该信号量，就点亮一盏信号灯，表示可用资源数加 1，否则不点亮信号灯而是直接把信号灯转交给等待中的事件，并激活该事件。

参数：semp，被点亮的信号灯。

返回：无。

3. 请求一盏信号灯

函数原型：

bool_t sem_pend(struct semaphore_LCB *semp,uint32_t timeout);

功能：请求一盏信号灯，如果有点亮的信号灯，就熄灭一盏信号灯，表示可用资源数减

1，返回 true，否则阻塞线程，直到有其他线程释放该信号灯或超时。

参数：semp，被请求的信号量。

timeout，如果在 timeout 毫秒后，仍无空闲内存块，将强行退出阻塞态。阻塞时间将被向上取整为整数个 ticks。。

返回：true = 成功获取信号灯（含阻塞后获取），false = 没有获取信号灯。

4. 删除一个信号量

函数原型：

bool_t sem_delete(struct semaphore_LCB *semp);

功能：当一个信号量不再需要时，调用本函数删除之。把信号量占用的锁控制块归还锁控制块内存池，把信号量从系统资源链表中删除。如果有事件在该信号量的同步队列中，则不能删除。

参数：semp，被删除的信号量。

返回：true = 成功删除，false = 失败，。

5. 查询信号灯已用数量

函数原型：

uint32_t sem_query_used(struct semaphore_LCB *semp);

功能：查询信号灯已用数量，也就是处于熄灭状态的盏数。

参数：semp，被查询的信号量。

返回：已用信号灯数量。

6. 查询信号灯总量

函数原型：

uint32_t sem_query_capacital(struct semaphore_LCB *semp);

功能：查询信号量包含的信号灯总量。

参数：semp，被查询的信号量。

返回：信号灯总量。

7. 查询信号灯可用数量

函数原型：

uint32_t sem_query_free(struct semaphore_LCB *semp);

功能：查询信号灯可用数量，也就是处于点亮状态的盏数。

参数：semp，被查询的信号量。

返回：可用信号灯数量。

8. 创建一个互斥量

函数原型：

struct mutex_LCB *mutex_create(bool_t init_lamp,char *name);

功能：从锁控制块池中分配一个锁控制块作为新创建的互斥量控制块，然后初始化并加入系统资源队列中。如果在分配锁控制块失败将直接返回 NULL。

参数：init_lamp，初始化互斥量状态，true = 可用，false = 不可用。

name，信号量的名字，信号量是系统资源队列中的一个资源，name 就是资源结点的名字。

返回：刚创建的互斥量指针，失败则返回 NULL。

9. 释放一个互斥量

函数原型：

void mutex_post(struct mutex_LCB *mutex);

功能：如果没有事件在等待该互斥量，就把互斥量设为可用，否则维持互斥量不可用，

直接把互斥量转交给等待中的事件，并激活该事件。。

参数：mutex，释放的互斥量。

返回：无。

10. 请求一个互斥量

函数原型：

bool_t mutex_pend(struct mutex_LCB *mutex,uint32_t timeout);

功能：请求一个互斥量，如果有互斥量可用，就拿走并设为不可用，否则阻塞线程，直到有其他线程释放该互斥量或超时。

参数：mutex，被请求的互斥量。

timeout，如果在 timeout 毫秒后，仍无空闲内存块，将强行退出阻塞态。阻塞时间将被向上取整为整数个 ticks。

返回：true = 成功获取互斥量（含阻塞后获取），false = 没有获取互斥量。

11. 删除一个互斥量

函数原型：

bool_t mutex_delete(struct mutex_LCB *mutex);

功能：当一个互斥量量不再需要时，调用本函数删除之。把互斥量占用的锁控制块归还锁控制块内存池，把互斥量从系统资源链表中删除。如果有事件在该互斥量的同步队列中，则不能删除。

参数：mutex，被删除的互斥量。

返回：true = 成功删除，false = 失败，。

12. 查询互斥量的使用状态

函数原型：

bool_t mutex_query_used(struct mutex_LCB *mutex);

功能：查询一个互斥量是可用还是不可用。

参数：mutex，被查询的互斥量。

返回：true = 可用，false = 不可用。

17.9 看门狗

1. 创建一只看门狗

函数原型：

struct wdt_rsc * wdt_create(bool_t (*judge)(void),
uint32_t (*yip_remedy)(void),
uint32_t timeout,char *wdt_name);

功能：创建一只看门狗，如果之前没有看门狗，则还要弹出看门狗服务事件。在创建看门狗之前，用户还要准备好判断是否狗叫的 judge 函数，以及狗叫以后的善后函数 yip_remedy。看门狗模块本身不做裁判，只是调用用户提供的 judge 函数判断是否狗叫，狗叫后，看门狗模块也不自作主张地直接复位系统，而是由善后函数 yip_remedy 的返回值决定。该函数的返回值有两个：cn_wdt_action_none=无任何操作，一般是因为善后函数已经处理好一切；cn_wdt_action_reset=要求复位，看门狗模块将停止喂狗，直到硬件复位。

参数：judge，用户提供的判断狗叫的函数。

yip_remedy，用户提供的狗叫后的善后函数。

timeout，看门狗溢出周期，单位是毫秒，将被向上取整为整数个 ticks。看门狗模块每各 timeout 毫秒调用一次该看门狗的 judge 函数。

name，看门狗的名字，看门狗是系统资源队列中的一个资源，name 就是资源结点的名字。

返回：刚创建的看门狗。

2. 删除一只看门狗

函数原型：

void wdt_delete(struct wdt_rsc *wdt);

功能：当某看门狗不再需要，调用本函数删除之，把看门狗从系统资源链表中删除，并回收看门狗控制块锁占用的内存块。

参数：wdt，被删除的看门狗。

返回：无。

17.10 文件系统

1. 打开文件

函数原型：

djyfs_file *djyfs_fopen(char *fullname, char *mode);

功能：打开一个文件。

参数：fullname，包含全路径（含文件柜名）的文件名。

mode，模式字符串，参见 第 12 章。

返回：文件描述符。

2. 查找文件

函数原型：

bool_t djyfs_fsearch(char *fullname);

功能：在文件柜中查找一个文件是否存在。

参数：fullname，包含全路径（含文件柜名）的文件名。

mode，模式字符串，参见 第 12 章。

返回：true = 文件存在，false = 文件不存在。

3. 删除文件

函数原型：

bool_t djyfs_remove(char *fullname);

功能：删除一个文件，该文件必须未打开，如果是目录则必须是空目录。

参数：fullname，包含全路径（含文件柜名）的文件名。

返回：。

4. 文件（目录）改名

函数原型：

bool_t djyfs_rename(char *old_fullname,char *new_filename);

功能：把一个文件或者目录的名字改掉。

参数：old_fullname，包含全路径（含文件柜名）的文件名。

new_filename，把 old_fullname 中以字符'\'分割的最后一个文件（或目录）的名字改为 new_filename。

返回：true = 成功修改，false = 失败。

5. 关闭文件（目录）

函数原型：

uint32_t djyfs_fclose(djyfs_file *fp);

功能：关闭一个文件（或目录），如果是目录，则该目录不能有打开的下级目录或文件。

参数：fp，被关闭的文件（目录）。

返回：0 = 成功，cn_limit_uint32 = 失败。

6. 从文件读数据

函数原型：

size_t djyfs_fread(void *buf,size_t size, size_t nmemb,djyfs_file *fp);

功能：从文件中读出数据。

参数：buf，保存读出的数据的缓冲区。

size，待读出的单元数。

nmemb，每个单元的字节数。

fp，被读的文件

返回：读出的数据数量（单元数）。

7. 写数据到文件

函数原型：

size_t djyfs_write(void *buf,size_t size, size_t nmemb,djyfs_file *fp);

功能：写数据到文件中。

参数：buf，保存写入的数据的缓冲区。

size，欲写入的单元数。

nmemb，每个单元的字节数。

fp，被写的文件

返回：实际写入的数据数量（单元数）。

8. 刷新文件

函数原型：

uint32_t djyfs_fflush(djyfs_file *fp);

功能：djyos 提供的是带缓冲的文件系统，只要缓冲区有空位置，平时写入操作就把数据写到缓冲区中，调用本函数把缓冲区中的数据写入到存储介质中。

参数：fp，被操作的文件。

返回：刷新操作写入存储介质的数据量。

9. 设置文件指针位置

函数原型：

uint32_t djyfs_fseek(djyfs_file *fp, sint64_t offset, int whence);

功能：把文件指针从 whence 指定的位置开始移动 offset 长度，whence 含义为：SEEK_SET=从 0 开始计算，SEEK_CUR=从当前位置开始计算，SEEK_END=从文件结束位置计算。

参数：fp，被操作的文件。

offset，移动长度。

whence，移动的参考位置

返回：1=成功移动，0=失败。

10. 格式化文件柜

函数原型：

bool_t djyfs_format(uint32_t cmd,char *dbx_name);

功能：格式化文件柜，初始化存储介质，建立目录表等。

参数：cmd，格式化命令。

dbx_name，被格式化的文件柜名。

返回：true = 格式化成功，false = 失败。

● 附录一　编码规范

一、 排版

1. 程序块采用缩进风格编写，缩进的空格数为 4 个。
2. 相对独立的程序块之间、变量说明之后必须加空行。
3. 在不影响阅读与理解的前提下，使代码紧凑，一页能显示尽量多的内容，
4. 较长的语句（行末>80 字符）分成多行书写，长表达式在低优先级操作符处划分新行，操作符放在新行之首；函数参数多要分行书写的，不允许参数骑行（即同一个参数内分行），划分出的新行要进行适当的缩进，使排版整齐，语句可读。
5. 运算符号前后留空格，但因这些空格导致行长超过 80 字符的，根据规则 3，可以删除空格。
6. 不允许把多个短语句写在一行中，即一行只写一条语句。根据规则 3 产生的特例是，“if (a>b) return;”可以写在一行，但不允许 “if(a > b) b++;”写在一行，因为这样写将导致逐行调试时不知道 b++执行了没有。
7. 缩进以及对齐只使用空格键，不使用 TAB 键。这样在所有编辑器上均能够排版整齐。
8. 程序块的分界符（如 C/C++语言的大括号‘{’和‘}’）应各独占一行并且位于同一列，同时与引用它们的语句左对齐。
9. 格式整齐与信息丰富同等重要，让一屏（一页）显示尽量多的内容，对阅读程序也很有益处。
10. 函数参数多但不超过 80 字符的，不影响阅读前提下，尽量不分行；

二、 注释

1. 注释语言，应尽量使用中文。
2. 注释格式
   a) 源程序中全部使用行注释，即“//注释内容”。
   b) 当调试中需要临时注释掉部分代码时，使用“// db ”。
      调试中需要增加用于调试的代码时，应使用如下格式
      printf(“输出调试信息”);　　// db
      db 的前后必须是英文符号或空格，以便批量搜索。
   c) 注释应该对齐以保持行文美观，但应让步于排版规则 3，即如果因对齐而导致分行，在不特别影响阅读的情况下，允许不对齐注释，例如下列代码中，除 abrasion 外其他变量的注释是对齐的，但对齐变量 abrasion 的注释将导致分行，可不对齐。
      ```
      uint32_t sector_per_block;   //每块扇区数
      uint32_t sector_size;        //每扇区字节数
      uint32_t abrasion;    //磨损次数基数，加磨损次数表(16bit)得实际磨损次数
      bool_t    formatted;         //文件柜格式化标志
      uint32_t start_block;        //起始块号
      ```
3. 注释内容应该简洁明确，没有歧义，避免过度注释，如：

a = 100;　　//a 赋值为 100

就属于过度注释，这样是侮辱阅读程序的人的智商。

4. 除了已经成为该软件所属行业内通用语的缩写外，比如用 DC 和 AC 表示直流和交流，注释中不要使用缩写。
5. 单行语句的注释放在语句的右方，依排版规则 3，若注释使行超长而导致分行书写则放在语句的上方，与被注释的语句对齐。对语句块的注释紧跟在“{”的右方。
6. 在较长的程序块结束时，“}”右方加入注释，说明是哪一个程序块结束，例如：

```
if(a==b)
{//语句块的注释，紧跟“{”
    //“a=0”的注释，与语句对齐
    ap=”本行很长，在行尾增加注释会使行超长而导致分行书写”;
    a = 0;                  //行不超长,直接放在语句后面.
    b = 1+2+3+4+5;    //注释与上一行对齐.
    bar = 12; //本行注释若与上一行对齐可导致行超长而分行书写，可不对齐。
}    // end of   if(a==b)
```

7. 变量、常量、数据结构，如果命名没有充分注释，应该加注释。
8. 全局变量要做特别的注释，说明在哪些模块使用它，哪些模块可以修改它，取值范围，访问时的注意事项等。
9. 对分支语句要对其所有分支程序块单独编写注释，说明分支条件，case 语句如果没有 break，应使用独占一行的　“// break＋不使用 break 的原因”。
10. 禁止在一行代码的中间插入注释。
11. 文件头和函数头注释

说明性文件（如头文件.h 文件、.inc 文件、.def 文件、编译说明文件.cfg 等）头部应进行注释，注释必须列出：版权说明、版本号、生成日期、作者、内容和功能描述、与其它文件的关系、修改日志等，头文件的注释中还应有函数功能简要说明。除版权信息用中英文对照外，所有内容均使用中文。

源代码文件和头文件的头部注释如下：

```
//-------------------------------------------------
//Copyright (C), 2005-2008,   lst.
//版权所有 (C), 2005-2008,    lst.
//所属模块:
//作者：
//版本：
//文件描述:
//其他说明:
//修订历史:    //按时间先后倒叙，最上面是最新修订的说明
//     2. ...
//     1. 日期:
//        作者:
//        新版本号：
//        修改说明:
//-------------------------------------------------
```

函数头部注释如下：

```
//----恢复保存的中断线状态----------------------------
//功能：函数功能、性能等的描述
//参数：输入参数说明，包括每个参数的作用、取值说明及参数间关系。
//返回：函数返回值的说明
//其他：其它说明
//-----------------------------------------------
```

三、 命名规则

1. 标识符的命名要清晰、明了，有明确含义，使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。
2. 任何时候，禁止取单字符变量。标识符命名中尽量不使用数字 0 和 1，以避免和英文字符 O 和 l 混淆。标识符中有数字串或者数字序列时例外，比如 chanel3021，int0,int1,int2。
3. djyos 标识符采用全小写＋下划线分割的方式书写。
4. 变量命名规则

变量格式：<变量类型><访问类型><下划线><变量描述>

变量类型表：

名称	类型名	描述
uxx	uint8_t，uint16_t 等	确定长度的无符号数，xx 表示变量长度，以位表示，如 u8，u16 等
sxx	sint8_t，sint16_t 等	确定长度的有符号数，xx 表示变量长度，以位表示，如 u8，u16 等
uc 和 sc	ucpu_t 和 scpu_t	读取该类型变量可以保证原子性
uf 和 sf	ufast_t 和 sfast_t	长度等于存储器位宽，可保证读写操作快速和读操作的原子性。
t	结构类型	
p	指针类型，	指针的目标类型不再细分
pp	指向指针的指针	包含多级指针,如***pp
ch	字符类型	只用于存储字符，不能进行数学运算
fs	单精度浮点数	
fd	双精度浮点数	
b	boolean 类型	
q	其他类型	比如 size_t 等
cn	常量	常量可以忽略访问类型
注：数组用其元素名表示		

访问类型表：

名称	描述	范例
l	局部变量	ls 表示局部静态变量
g	全局变量	gv 表示全局 volatile 变量
s	静态变量	
v	volatile 变量	

r	只读变量	
w	只写变量	
注：结构成员可省略		

5. 函数命名规则：<模块名前缀><下划线><函数描述>，例如 int_save_line

模块名前缀	模块名称
y	操作系统内核调度
int	中断模块
m	内存管理模块

6. 数据类型命名规则：<类型描述><下划线><字符 t>，例如 uint16_t

四、 代码约定

1. 禁止函数的参数作为工作变量，用于返回结果的指针例外。
2. 函数的规模尽量限制在 200 行以内。
3. 一个函数仅完成一件功能，不设计多用途面面俱到的函数。
4. 不使用难懂的技巧性很高的语句，在有特殊需要时，应伴随详细的注释出现。
5. 不要有多用途的复合表达式；

   例如：

   d = (a = b + c) + r ; //应拆分为两个语句：

   a = b + c;

   d = a + r;
6. 不使用太复杂的表达式，除加减乘除外，关系运算一律用括号确定运算顺序。
7. 布尔变量的赋值和比较都统一使用 true 和 false，不使用 1 和 0，因为我们不知道开发系统如何定义这两个常量。

参考文献

[1] Andrew N.Solss ,Dominic Symes ,Chris Wright . ARM 嵌入式系统开发－软件设计与优化 .沈建华译 .北京 ： 北京航空航天大学出版社 ,2005

[2] 杜春雷 . ARM 体系结构与编程 . 北京 ： 清华大学出版社 ,2003

[3] Wayne Wolf . 嵌入式计算系统设计原理 .孙玉芳 ,梁彬等译 .北京 :机械工业出版社 ,2002

[4] Gary Nutt .操作系统 .罗宇 ,吕硕译 .北京 :机械工业出版社 ,2005

[5] Raj Kamal . 嵌入式系统－体系结构、编程与设计 . 陈曙晖 译 .北京 :清华大学出版社 ,2005

[6] Bonnie Baker . 嵌入式系统中的模拟设计 . 李喻奎 译 .北京 :北京航空航天大学出版社 ,2006

[7] Robert Love . Linux 内核设计与实现 . 陈莉君，康华等译.北京 :机械工业出版社 ,2006