前言

个人电脑已经是 64 位了，您还在使用 8 位微控制器吗？

尽管一般情况下嵌入式系统对 CPU 处理能力的要求比个人电脑（对 CPU 处理能力的要求）低，但随着人们生活的提高和技术的进步，嵌入式系统对 CPU 处理能力的要求也稳步的提高，大量高速的与 MCS51 体系结构兼容的微控制器的出现就证明了这一点。但 8 位微控制器受限于体系结构，处理能力的提高始终有限。而 16 位系统在性能上与 8 位机相比始终没有太大优势，成本上与 32 位系统相比也没有什么优势，未来一段时间嵌入式微控制器的发展方向必然是 32 位系统。

基于 ARM 体系结构的 32 位系统占领了 32 位嵌入式系统的大部分分额，但长期以来，基于 ARM 体系结构的 32 位系统仅在嵌入式式系统的高端（通讯领域、PDA）等场合使用，要么以专用芯片的面貌出现，要么以位处理器的庙貌出现，并没有出现性价比高的通用的微控制器。PHILIPS 发现了这个空当，推出了性价比很高 LPC2000 系列微控制器，让更多的嵌入式系统具有 32 位的处理能力。这也预示着 32 位系统即将成为嵌入式系统的主流。

基于 ARM 体系结构的芯片在中国推广已经有好几年了，关于 ARM 的图书也出了不少。关于 ARM 的图书主要有以下几类：

1．关于 ARM 内核的图书，主要读者是芯片设计者，内容主要是介绍芯片设计的。

2．芯片应用类图书，主要是芯片的生产商或代理商编写，主要读者为应用工程师。

3．开发板类图书，主要介绍相应的 ARM 开发板，给应用者一些参考。

以上 3 类图书的侧重点都不是 ARM 应用开发教学，用于大学本科教学不太适合。为了方便高等院校教学方便，笔者编写了这本教材。不过，因为嵌入式系统牵涉的知识太广，一本教材无法深入论述。为此，笔者还会推出多本被套图书以便学生知识扩展。

目录

第1章 嵌入式系统概述

1.1 嵌入式系统

经过几十年的发展，嵌入式系统已经在很大程度改变了人们的生活、工作和娱乐方式，而且这些改变还在加速。嵌入式系统具有无数的种类，每类都具有自己独特的个性。例如，MP3、数码相机与打印机就有很大的不同。汽车中更是具有多个嵌入式系统，使汽车更轻快、更干净、更容易驾驶。

尽管嵌入式系统极大地改变了人们的生活、工作和娱乐，但要定义嵌入式系统的概念却不容易，下面先介绍一些生活中常见的嵌入式系统。

1.1.1 现实中的嵌入式系统

即使不可见，嵌入式系统也无处不在。嵌入式系统在很多产业中得到了广泛的应用并逐步改变着这些产业，包括工业自动化、国防、运输和航天领域。例如神州飞船和长征火箭中肯定有很多嵌入式系统，导弹的制导系统也是嵌入式系统，高档汽车中也有多达几十个嵌入式系统。

在日常生活中，人们使用各种嵌入式系统，但未必知道它们。图 1.1 就是一些比较新的、生活中比较常见的嵌入式系统。事实上，几乎所有带有一点“智能”的家电（全自动洗衣机、电脑电饭煲…）都是嵌入式系统。嵌入式系统广泛的适应能力和多样性，使得视听、工作场所甚至健身设备中到处都有嵌入式系统。

图 1.1 常见的嵌入式系统应用实例

1.1.2 嵌入式系统的概念

目前，对嵌入式系统的定义多种多样，但没有一种定义是全面的。下面给出两种比较合理定义：

- 嵌入式系统：以应用为中心、以计算机技术为基础、软件硬件可裁剪、适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。
- 嵌入式系统：嵌入式系统是设计完成复杂功能的硬件和软件，并使其紧密耦合在一起的计算机系统。术语嵌入式反映了这些系统通常是更大系统中的一个完整的部分，称为嵌入的系统。嵌入的系统中可以共存多个嵌入式系统。

两种定义的出发角度不同，一个是从技术的角度来定义的，另一个是从系统的角度来定义的。事实上，在大多数情况下，嵌入式系统是真正的被嵌入，即它们是“系统中的系统”。它们不能够或没有自身的功能。例如，数字机顶盒 DST(Digital Set -Top box)可以在许多家庭娱乐场所中找到。数字音频 / 视频解码系统，称为 A/V 解码器（A/V decoder），是 DST 的一个完整部分，是一个嵌入式系统。A/V 解码器接收单个的多媒体流，并且产生声音和视频帧作为输出。DST 从卫星接收的信号中包含多个流或频道，因此，A/V 解码器与传输流解码器连接工作。传输流解码器也是一个嵌入式系统。传输流解码器解调收到的多媒体流到分离的频道上，并且只将所选的频道送给 A/V 解码器。

某些情况下，嵌入式系统在功能上是独立的系统。例如网络路由器是独立的嵌入式系统。它由特殊的通信处理器、内存、许多网络访问接口（称为网络端口）以及实现包的路由算法的特殊软件组成。换句话说，网络路由器是一个独立的嵌入式系统，路由包从一个端口到另一个端口，实现程序化的路由算法。

1.1.3 嵌入式系统的未来

早在 1990 年之前，嵌入式系统通常是很简单的且具有很长的产品生命周期的自主设备。近些年来，嵌入式工业经历了巨大的变革。

- 产品市场窗口现在预计翻番的周期狂热到 6～9 个月。
- 全球重新定义市场的机会和膨胀的应用空间。
- 互联现在是一个需求而不是辅助性的，包括用有线和刚刚显露头角的无线技术。
- 基于电子的产品更复杂化。
- 互联嵌入式系统产生新的依赖网络基础设施的应用。
- 微处理器的处理能力按莫尔定律（Moore’s L aw）预计的速度在增加。该定律认为集成电路和晶体管个数每 18 个月翻一番。

如果说过去的趋势能指明未来，那么随着技术的革新，嵌入式软件将继续增加新的应用，并产生更加灵巧的产品种类。根据人们对于自身虚拟运行设备的消费要求增加而不断壮大的市场，以及由 Internet 创造的无限的机会，嵌入式系统将不断地重新塑造未来的世界。

1.2 嵌入式处理器

1.2.1 简介

普通个人计算机（PC）中的处理器是通用目的的处理器。它们的设计非常丰富，因为这些处理器提供全部的特性和广泛的功能，故可以用于各种应用中。使用这些通用处理器的系统有大量的应用编程资源。例如，现代处理器具有内置的内存管理单元（MMU），提供内存保护和多任务能力的虚存和通用目的的操作系统。这些通用的处理器具有先进的高速缓存逻辑。许多这样的处理器具有执行快速浮点运算的内置数学协处理器。这些处理器提供接口，支持各种各样的外部设备。这些处理器能源消耗大，产生的热量高，尺寸也大。其复杂性意

味着这些处理器的制造成本昂贵。在早期，嵌入式系统通常用通用目的的处理器建造。

近年来，随着大量先进的微处理器制造技术的发展，越来越多的嵌入式系统用嵌入式处理器建造，而不是用通用目的的处理器。这些嵌入式处理器是为完成特殊的应用而设计的特殊目的的处理器。关键是应用意识，即知道应用的自然规律并满足这些应用的需求。

一类嵌入式处理器注重尺寸、能耗和价格。因此，某些嵌入式处理器限定其功能，即处理器对于某类应用足够好，而对于其他类的应用可能就不够好了。这就是为何许多的嵌入式处理器没有太高的 CPU 速度的原因。例如，为个人数字助理（PDA）设备选择的就没有浮点协处理器，因为浮点运算没有必要，或用软件仿真就足够了。这些处理器可以是 16－bit 地址体系结构，而不是 32－bit 的，因为受内存储器容量的限制；可以是 200MHz CPU 频率，因为应用的主要特性是交互和显示密集性的，而不是计算密集性的。这类嵌入式处理器很小，因为整个 PDA 装置尺寸很小并能放在手掌上。限制功能意味着降低能耗并延长电池供电时间。更小的尺寸可降低处理器的制造成本。

另一类嵌入式处理器更关注性能。这些处理器功能很强，并用先进的芯片设计技术包装，如先进的管道线和并行处理体系结构。这些处理器设计满足那些用通用目的处理器难以达到的密集性计算的应用需求。新出现的高度特殊的高性能的嵌入式处理器，包括为网络设备和电信工业开发的网络处理器。总之，系统和应用速度是人们关心的主要问题。

还有一类嵌入式处理器关注全部 4 个需求——性能、尺寸、能耗和价格。例如，蜂窝电话中的嵌入式数字信号处理器（DSP）具有特殊性的计算单元、内存中的优化设计、寻址和带多个处理能力的总线体系结构，这样 DSP 可以非常快地实时执行复杂的计算。在同样的时钟频率下，DSP 执行数字信号处理要比通用目的的处理器速度快若干倍，这就是在蜂窝电话的设计上用 DSP 而不用通用目的的处理器的原因。更甚之，DSP 具有非常快的速度和强大的嵌入式处理器，其价格是相当合适的，使得蜂窝电话的整体价格具有相当的竞争力。使用 DSP 的供电电池可以持续几十小时。

片上系统 SoC（System –on-a- Chip）处理器对嵌入式系统具有特别的吸引力。SoC 处理器具有 CPU 内核并带内置外设模块，如可编程通用目的计时器、可编程中断控制器、DMA 控制器和以太网接口。这样的自含设计使嵌入式设计可以用来建造各种嵌入式应用，而不需要附加外部设备，再次减少了最终产品的整个费用和尺寸。

1.2.2 分类

- **嵌入式微处理器(Embedded Microprocessor Unit, EMPU)**

嵌入式微处理器的基础是通用计算机中的 CPU。在应用中，将微处理器装配在专门设计的电路板上，只保留和嵌入式应用有关的母板功能，这样可以大幅度减小系统体积和功耗。为了满足嵌入式应用的特殊要求，嵌入式微处理器虽然在功能上和标准微处理器基本是一样的，但在工作温度、抗电磁干扰、可靠性等方面一般都做了各种增强。

和工业控制计算机相比，嵌入式微处理器具有体积小、重量轻、成本低、可靠性高的优点，但是在电路板上必须包括 ROM、RAM、总线接口、各种外设等器件，从而降低了系统的可靠性，技术保密性也较差。嵌入式微处理器及其存储器、总线、外设等安装在一块电路板上，称为单板计算机。如 STD-BUS、PC104 等。近年来，德国、日本的一些公司又开发出了类似“火柴盒”式名片大小的嵌入式计算机系列 OEM 产品。

嵌入式处理器目前主要有 Am186/88、386EX、SC-400、Power PC、68000、MIPS、ARM 系列等。

- **嵌入式微控制器(Microcontroller Unit, MCU)**

嵌入式微控制器又称单片机，顾名思义，就是将整个计算机系统集成到一块芯片中。嵌入式微控制器一般以某一种微处理器内核为核心，芯片内部集成 ROM/EPROM、RAM、总线、总线逻辑、定时/计数器、WatchDog、I/O、串行口、脉宽调制输出、A/D、D/A、Flash RAM、

EEPROM 等各种必要功能和外设。为适应不同的应用需求，一般一个系列的单片机具有多种衍生产品，每种衍生产品的处理器内核都是一样的，不同的是存储器和外设的配置及封装。这样可以使单片机最大限度地和应用需求相匹配，功能不多不少，从而减少功耗和成本。

和嵌入式微处理器相比，微控制器的最大特点是单片化，体积大大减小，从而使功耗和成本下降、可靠性提高。微控制器是目前嵌入式系统工业的主流。微控制器的片上外设资源一般比较丰富，适合于控制，因此称微控制器。

嵌入式微控制器目前的品种和数量最多，比较有代表性的通用系列包括 8051、P51XA、MCS-251、MCS-96/196/296、C166/167、MC68HC05/11/12/16、68300、数目众多 ARM 芯片等。目前 MCU 占嵌入式系统约 70%的市场份额。

- **嵌入式 DSP 处理器(Embedded Digital Signal Processor, EDSP)**

DSP 处理器对系统结构和指令进行了特殊设计，使其适合于执行 DSP 算法，编译效率较高，指令执行速度也较高。在数字滤波、FFT、谱分析等方面 DSP 算法正在大量进入嵌入式领域，DSP 应用正从在通用单片机中以普通指令实现 DSP 功能，过渡到采用嵌入式 DSP 处理器。

嵌入式 DSP 处理器比较有代表性的产品是 Texas Instruments 的 TMS320 系列和 Motorola 的 DSP56000 系列。TMS320 系列处理器包括用于控制的 C2000 系列，移动通信的 C5000 系列，以及性能更高的 C6000 和 C8000 系列。DSP56000 目前已经发展成为 DSP56000，DSP56100，DSP56200 和 DSP56300 等几个不同系列的处理器。另外 PHILIPS 公司近年也推出了基于可重置嵌入式 DSP 结构低成本、低功耗技术上制造的 R. E. A. L DSP 处理器，特点是具备双 Harvard 结构和双乘/累加单元，应用目标是大批量消费类产品。

- **嵌入式片上系统(System On Chip)**

随着 EDI 的推广和 VLSI 设计的普及化及半导体工艺的迅速发展，在一个硅片上实现一个更为复杂的系统的时代已来临，这就是 System On Chip(SOC)。各种通用处理器内核将作为 SOC 设计公司的标准库，和许多其它嵌入式系统外设一样，成为 VLSI 设计中一种标准的器件，用标准的 VHDL 等语言描述，存储在器件库中。用户只需定义出其整个应用系统，仿真通过后就可以将设计图交给半导体工厂制作样品。这样除个别无法集成的器件以外，整个嵌入式系统大部分均可集成到一块或几块芯片中去，应用系统电路板将变得很简洁，对于减小体积和功耗、提高可靠性非常有利。

SOC 可以分为通用和专用两类。通用系列包括 Infineon 的 TriCore，Motorola 的 M-Core，某些 ARM 系列器件，Echelon 和 Motorola 联合研制的 Neuron 芯片等。专用 SOC 一般专用于某个或某类系统中，不为一般用户所知。一个有代表性的产品是 Philips 的 Smart XA，它将 XA 单片机内核和支持超过 2048 位复杂 RSA 算法的 CCU 单元制作在一块硅片上，形成一个可加载 JAVA 或 C 语言的专用的 SOC，可用于公众互联网如 Internet 安全方面。

1.3 嵌入式操作系统

1.3.1 简介

在计算机技术发展的初期阶段，计算机系统中没有操作系统这个概念。为了给用户提供一个与计算机之间的接口，同时提高计算机的资源利用率便出现了计算机监控程序(Monitor)，使用户能通过监控程序来使用计算机。随着计算机技术的发展，计算机系统的硬件、软件资源也愈来愈丰富，监控程序已不能适应计算机应用的要求。于是在六十年代中期监控程序又进一步发展形成了操作系统(Operating System)。发展到现在，广泛使用的有三种操作系统即多道批处理操作系统、分时操作系统以及实时操作系统。

多道批量处理系统一般用于计算中心较大的计算机系统中。由于它的硬件设备比较全、

价格较高，所以此类系统十分注意 CPU 及其它设备的充分利用，追求高的吞吐量，不具备实时性。

分时系统的主要目的是让多个计算机用户能共享系统的资源，能及时地响应和服务于联机用户，只具有很弱的实时功能，但与真正的实时操作系统仍然有明显的区别。

那么什么样的操作系统才能称为实时操作系统呢? IEEE 的实时 UNIX 分委会认为实时操作系统应具备以下的几点:

1. 异步的事件响应

实时系统为能在系统要求的时间内响应异步的外部事件，要求有异步 I/O 和中断处理能力。I/O 响应时间常受内存访问、盘访问和处理机总线速度所限制。

2. 切换时间和中断延迟时间确定

3. 优先级中断和调度

必须允许用户定义中断优先级和被调度的任务优先级并指定如何服务中断。

4. 抢占式调度

为保证响应时间，实时操作系统必须允许高优先级任务一旦准备好运行马上抢占低优先级任务的执行。

5. 内存锁定

必须具有将程序或部分程序锁定在内存的能力，锁定在内存的程序减少了为获取该程序而访问盘的时间，从而保证了快速的响应时间。

6. 连续文件

应提供存取盘上数据的优化方法，使得存取数据时查找时间最少。通常要求把数据存储在连续文件上。

7. 同步

提供同步和协调共享数据使用和时间执行的手段。

总的来说实时操作系统是事件驱动的(event_driven)，能对来自外界的作用和信号在限定的时间范围内作出响应。它强调的是实时性、可靠性和灵活性，与实时应用软件相结合成为有机的整体起着核心作用，由它来管理和协调各项工作，为应用软件提供良好的运行软件环境及开发环境。

从实时系统的应用特点来看实时操作系统可以分为两种：一般实时操作系统和嵌入式实时操作系统。

一般实时操作系统与嵌入式实时操作系统都是具有实时性的操作系统，它们的主要区别在于应用场合和开发过程。

一般实时操作系统应用于实时处理系统的上位机和实时查询系统等实时性较弱的实时系统，并且提供了开发、调试、运用一致的环境。

嵌入式实时操作系统应用于实时性要求高的实时控制系统，而且应用程序的开发过程是通过交叉开发来完成的，即开发环境与运行环境是不一致。嵌入式实时操作系统具有规模小(一般在几 K-几十 K 内)、可固化使用实时性强(在毫秒或微秒数量级上)的特点。

1.3.2 基本概念

- **前后台系统**

对于基于芯片开发来说，应用程序一般是一个无限的循环，可称为前后台系统或超循环系统。循环中调用相应的函数完成相应的操作，这部分可以看成后台行为。中断服务程序处理异步事件，这部分可以看成前台行为。后台也可以叫做任务级，前台也叫中断级。时间相关性很强的关键操作一定是靠中断服务程序来保证的。因为中断服务提供的信息一直要等到后台程序走到该处理这个信息这一步时才能得到进一步处理，所以这种系统在处理的及时性上比实际可以做到的要差。这个指标称作任务级响应时间。最坏情况下的任务级响应时间取

决于整个循环的执行时间。因为循环的执行时间不是常数，程序经过某一特定部分的准确时间也不能确定。进而，如果程序修改了，循环的时序也会受到影响。

很多基于微处理器的产品采用前后台系统设计，例如微波炉、电话机、玩具等。在另外一些基于微处理器应用中，从省电的角度出发，平时微处理器处在停机状态，所有事都靠中断服务来完成。

- **操作系统**

操作系统是计算机中最基本的程序。操作系统负责计算机系统中全部软硬资源的分配与回收、控制与协调等并发的活动；操作系统提供用户接口，使用户获得良好的工作环境；操作系统为用户扩展新的系统功能提供软件平台。

- **实时操作系统（RTOS）**

实时操作系统是一段在嵌入式系统启动后首先执行的背景程序，用户的应用程序是运行于 RTOS 之上的各个任务，RTOS 根据各个任务的要求，进行资源(包括存储器、外设等)管理、消息管理、任务调度、异常处理等工作。在 RTOS 支持的系统中， 每个任务均有一个优先级，RTOS 根据各个任务的优先级，动态地切换各个任务， 保证对实时性的要求。 工程师在编写程序时，可以分别编写各个任务，不必同时将所有任务运行的各种可能情况记在心中，大大减小了程序编写的工作量，而且减小了出错的可能，保证最终程序具有高可靠性。实时多任务操作系统，以分时方式运行多个任务，看上去好象是多个任务“同时”运行。任务之间的切换应当以优先级为根据，只有优先服务方式的 RTOS 才是真正的实时操作系统，时间分片方式和协作方式的 RTOS 并不是真正的“实时”。

- **代码的临界区**

代码的临界区也称为临界区，指处理时不可分割的代码，运行这些代码不允许被打断。一旦这部分代码开始执行，则不允许任何中断打入（这不是绝对的，如果中断不调用任何包含临界区的代码，也不访问任何临界区使用的共享资源，这个中断可能可以执行）。为确保临界区代码的执行，在进入临界区之前要关中断，而临界区代码执行完成以后要立即开中断。

- **资源**

程序运行时可使用的软、硬件环境统称为资源。资源可以是输入输出设备，例如打印机、键盘、显示器。资源也可以是一个变量、一个结构或一个数组等。

- **共享资源**

可以被一个以上任务使用的资源叫做共享资源。为了防止数据被破坏，每个任务在与共享资源打交道时，必须独占该资源，这叫做互斥。至于在技术上如何保证互斥条件，本章会做进一步讨论。

- **任务**

一个任务，也称作一个线程，是一个简单的程序，该程序可以认为 CPU 完全属于该程序自己。实时应用程序的设计过程，包括如何把问题分割成多个任务，每个任务都是整个应用的某一部分，每个任务被赋予一定的优先级，有它自己的一套 CPU 寄存器和自己的栈空间。

- **任务切换**

当多任务内核决定运行另外的任务时，它保存正在运行任务的当前状态，即 CPU 寄存器中的全部内容。这些内容保存在任务的当前状态保存区，也就是任务自己的栈区之中。入栈工作完成以后，就把下一个将要运行的任务的当前状态从任务的栈中重新装入 CPU 的寄予存器，并开始下一个任务的运行。这个过程就称为任务切换。这个过程增加了应用程序的额外负荷。CPU 的内部寄存器越多，额外负荷就越重。做任务切换所需要的时间取决于 CPU 有多少寄存器要入栈。实时内核的性能不应该以每秒钟能做多少次任务切换来评价。

- **内核**

多任务系统中，内核负责管理各个任务，或者说为每个任务分配 CPU 时间，并且负责任务之间的通信。内核提供的基本服务是任务切换。之所以使用实时内核可以大大简化应用系统的设计，是因为实时内核允许将应用分成若干个任务，由实时内核来管理它们。内核本身也增加了应用程序的额外负荷。代码空间增加 ROM 的用量，内核本身的数据结构增加了 RAM 的用量，但更主要的是，每个任务要有自己的栈空间，这一块占起内存来是相当厉害的。内核本身对 CPU 的占用时间一般在 2 到 5 个百分点之间。

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

- **调度**

调度是内核的主要职责之一。调度就是决定该轮到哪个任务运行了。多数实时内核是基于优先级调度法的。每个任务根据其重要程序的不同被赋予一定的优先级。基于优先级的调度法指 CPU 总是让处在就绪态的优先级最高的任务先运行。然而究竟何时让高优先级任务掌握 CPU 的使用权，有两种不同的情况，这要看用的是什么类型的内核，是非占先式的还是占先式的内核。

- **非占先式内核**

非占先式内核要求每个任务自我放弃 CPU 的所有权。非占先式调度法也称作合作型多任务，各个任务彼此合作共享一个 CPU。异步事件还是由中断服务来处理。中断服务可以使一个高优先级的任务由挂起状态变为就绪状态。但中断服务以后控制权还是回到原来被中断了的那个任务，直到该任务主动放弃 CPU 的使用权时，那个高优先级的任务才能获得 CPU 的使用权。

- **占先式内核**

当系统响应时间很重要时，要使用占先式内核。因此绝大多数商业上销售的实时内核都是占先式内核。最高优先级的任务一旦就绪，总能得到 CPU 的控制权。当一个运行着的任务使一个比它优先级高的任务进入了就绪状态，当前任务的 CPU 使用权就被剥夺了，或者说被挂起了，那个高优先级的任务立刻得到了 CPU 的控制权。如果是中断服务子程序使一个高优先级的任务进入就绪态，中断完成时，中断了的任务被挂起，优先级高的那个任务开始运行。

- **任务优先级**

任务的优先级是表示任务被调度的优先程度。每个任务都具有优先级。任务越重要，赋予的优先级应越高，越容易被调度而进入运行态。

- **中断**

中断是一种硬件机制，用于通知 CPU 有个异步事件发生了。中断一旦被识别，CPU 保存部分（或全部）上下文即部分或全部寄存器的值，跳转到专门的子程序，称为中断服务子程序（ISR）。中断服务子程序做事件处理，处理完成后，程序回到：

1. 在前后台系统中，程序回到后台程序；
2. 对非占先式内核而言，程序回到被中断了的任务；
3. 对占先式内核而言，让进入就绪态的优先级最高的任务开始运行。

中断使得 CPU 可以在事件发生时才予以处理，而不必让微处理器连续不断地查询是否有事件发生。通过两条特殊指令：关中断和开中断可以让微处理器不响应或响应中断。在实时环境中，关中断的时间应尽量的短。

关中断影响中断延迟时间。关中断时间太长可能会引起中断丢失。微处理器一般允许中断嵌套，也就是在中断服务期间，微处理器可以识别另一个更重要的中断，并服务于那个更重要的中断。

- **时钟节拍**

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

1.3.3 使用实时操作系统的必要性

嵌入式实时操作系统在目前的嵌入式应用中用得越来越广泛，尤其在功能复杂、系统庞大的应用中显得愈来愈重要。

首先，嵌入式实时操作系统提高了系统的可靠性。在控制系统中，出于安全方面的考虑，要求系统起码不能崩溃，而且还要有自愈能力。不仅要求在硬件设计方面提高系统的可靠性和抗干扰性，而且也应在软件设计方面提高系统的抗干扰性，尽可能地减少安全漏洞和不可靠的隐患。长期以来的前后台系统软件设计在遇到强干扰时，使得运行的程序产生异常、出错、跑飞，甚至死循环，造成了系统的崩溃。而实时操作系统管理的系统，这种干扰可能只是引起若干进程中的一个被破坏，可以通过系统运行的系统监控进程对其进行修复。通常情况下，这个系统监视进程用来监视各进程运行状况，遇到异常情况时采取一些利于系统稳定可靠的措施，如把有问题的任务清除掉。

其次，提高了开发效率，缩短了开发周期。在嵌入式实时操作系统环境下，开发一个复杂的应用程序，通常可以按照软件工程中的解耦原则将整个程序分解为多个任务模块。每个任务模块的调试、修改几乎不影响其他模块。商业软件一般都提供了良好的多任务调试环境。

再次，嵌入式实时操作系统充分发挥了 32 位 CPU 的多任务潜力。32 位 CPU 比 8、16 位 CPU 快，另外它本来是为运行多用户、多任务操作系统而设计的，特别适于运行多任务实时系统。32 位 CPU 采用利于提高系统可靠性和稳定性的设计，使其更容易做到不崩溃。例如， CPU 运行状态分为系统态和用户态。将系统堆栈和用户堆栈分开，以及实时地给出 CPU 的运行状态等，允许用户在系统设计中从硬件和软件两方面对实时内核的运行实施保护。如果还是采用以前的前后台方式，则无法发挥 32 位 CPU 的优势。

从某种意义上说，没有操作系统的计算机(裸机)是没有用的。在嵌入式应用中，只有把 CPU 嵌入到系统中，同时又把操作系统嵌入进去，才是真正的计算机嵌入式应用。

1.3.4 实时操作系统的优缺点

在嵌入式实时操作系统环境下开发实时应用程序使程序的设计和扩展变得容易，不需要大的改动就可以增加新的功能。通过将应用程序分割成若干独立的任务模块，使应用程序的设计过程大为简化；而且对实时性要求苛刻的事件都得到了快速、可靠的处理。通过有效的系统服务，嵌入式实时操作系统使得系统资源得到更好的利用。

但是，使用嵌入式实时操作系统还需要额外的 ROM/RAM 开销，2~5%的 CPU 额外负荷，以及内核的费用。

1.3.5 常见的嵌入式操作系统

1. 嵌入式 Linux

uClinux 是一个完全符合 GNU/GPL 公约的操作系统，完全开放代码，现在由 Lineo 公司支持维护。uClinux 的发音是“you-see-linux”,它的名字来自于希腊字母“mu”和英文大写字母“C”的结合。“mu”代表“微小”之意，字母“C”代表“控制器”，所以从字面上就可以看出它的含义，即“微控制领域中的 Linux 系统”。

为了降低硬件成本及运行功耗，很多嵌入式 CPU 没有设计内存管理单元（Memory Management Unit，以下简称 MMU）功能模块。最初，运行于这类没有 MMU 的 CPU 之上的都是一些很简单的单任务操作系统，或者更简单的控制程序，甚至根本就没有操作系统而

直接运行应用程序。在这种情况下，系统无法运行复杂的应用程序，或者效率很低，而且，所有的应用程序需要重写，并要求程序员十分了解硬件特性。这些都阻碍了应用于这类 CPU 之上的嵌入式产品开发的速度。

uClinux 从 Linux 2.0/2.4 内核派生而来，沿袭了主流 Linux 的绝大部分特性。它是专门针对没有 MMU 的 CPU，并且为嵌入式系统做了许多小型化的工作。适用于没有虚拟内存或内存管理单元（MMU）的处理器，例如 ARM7TDMI。它通常用于具有很少内存或 Flash 的嵌入式系统。uClinux 是为了支持没有 MMU 的处理器而对标准 Linux 作出的修正。它保留了操作系统的所有特性，为硬件平台更好的运行各种程序提供了保证。在 GNU 通用公共许可证（GNU GPL）的保证下，运行 uClinux 操作系统的用户可以使用几乎所有的 Linux API 函数，不会因为没有 MMU 而受到影响。由于 uClinux 在标准的 Linux 基础上进行了适当的裁剪和优化，形成了一个高度优化的、代码紧凑的嵌入式 Linux，虽然它的体积很小，uClinux 仍然保留了 Linux 的大多数的优点：稳定、良好的移植性、优秀的网络功能、完备的对各种文件系统的支持、以及标准丰富的 API 等。

2. Win CE

Windows CE 是微软开发的一个开放的、可升级的 32 位嵌入式操作系统，是基于掌上型电脑类的电子设备操作。它是精简的 Windows 95。Windows CE 的图形用户界面相当出色。其中 CE 中的 C 代表袖珍（Compact）、消费（Consumer）、通信能力（Connectivity）和伴侣（Companion）；E 代表电子产品（Electronics）。与 Windows 95/98、Windows NT 不同的是，Windows CE 是所有源代码全部由微软自行开发的嵌入式新型操作系统，其操作界面虽来源于 Windows 95/98，但 Windows CE 是基于 Win32 API 重新开发的、新型的信息设备平台。Windows CE 具有模块化、结构化和基于 Win32 应用程序接口以及与处理器无关等特点。Windows CE 不仅继承了传统的 Windows 图形界面，并且在 Windows CE 平台上可以使用 Windows 95/98 上的编程工具（如 Visual Basic、Visual C++等）、使用同样的函数、使用同样的界面网格，使绝大多数的应用软件只需简单的修改和移植就可以在 Windows CE 平台上继续使用。

3. VxWorks

VxWorks 操作系统是美国 WindRiver 公司于 1983 年设计开发的一种嵌入式实时操作系统（RTOS），是嵌入式开发环境的关键组成部分。良好的持续发展能力、高性能的内核以及友好的用户开发环境，在嵌入式实时操作系统领域占据一席之地。它以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中，如卫星通讯、军事演习、弹道制导、飞机导航等。在美国的 F-16、FA-18 战斗机、B-2 隐形轰炸机和爱国者导弹上，甚至连 1997 年 4 月在火星表面登陆的火星探测器上也使用到了 VxWorks。

VxWorks 具有以下特点：

可靠性

操作系统的用户希望在一个工作稳定，可以信赖的环境中工作，所以操作系统的可靠性是用户首先要考虑的问题。而稳定、可靠一直是 VxWorks 的一个突出优点。自从对中国的销售解禁以来，VxWorks 以其良好的可靠性在中国赢得了越来越多的用户。

实时性

实时性是指能够在限定时间内执行完规定的功能并对外部的异步事件作出响应的能力。实时性的强弱是以完成规定功能和作出响应时间的长短来衡量的。

VxWorks 的实时性做得非常好，其系统本身的开销很小，进程调度、进程间通信、中断处理等系统公用程序精练而有效，它们造成的延迟很短。VxWorks 提供的多任务机制中对任务的控制采用了优先级抢占（Preemptive Priority Scheduling）和轮转调度（Round-Robin

Scheduling）机制，也充分保证了可靠的实时性，使同样的硬件配置能满足更强的实时性要求，为应用的开发留下更大的余地。

可裁减性

用户在使用操作系统时，并不是操作系统中的每一个部件都要用到。例如图形显示、文件系统以及一些设备驱动在某些嵌入系统中往往并不使用。

VxWorks 由一个体积很小的内核及一些可以根据需要进行定制的系统模块组成。VxWorks 内核最小为 8kB，即便加上其它必要模块，所占用的空间也很小，且不失其实时、多任务的系统特征。由于它的高度灵活性，用户可以很容易地对这一操作系统进行定制或作适当开发，来满足自己的实际应用需要。

4. OSE

OSE 主要是由 ENEA Data AB 下属的 ENEA OSE Systems AB 负责开发和技术服务的，一直以来都充当着实时操作系统以及分布式和容错性应用的先锋。公司建立于 1968 年，由大约 600 名雇员专门从事实时应用的技术支持工作。ENEA OSE Systems AB 是现今市场上一个飞速发展的 RTOS 供应商，在过去三年中，该公司的税收以每年 70％的速度递增。

该公司开发的 OSE 支持容错，适用于可从硬件和软件错误中恢复的应用，它的独特的消息传输方式使它能方便地支持多处理机之间的通信。它的客户深入到电信，数据，工控，航空等领域，尤其在电信方面，该公司已经有了十余年的开发经验，ENEA Data AB 现在已经成为日趋成熟，功能强大，经营灵活的 RTOS 供应商，也同诸如爱立信，诺基亚，西门子等知名公司确定了良好的关系。

OSE 操作系统的特点

- 高处理能力

内核中实时性严格的部分都由优化的汇编来实现，特别是使用信号量指针，使数据处理非常快。

- 真正适合开发复杂（包括多 CPU 和多 DSP）的分布式系统

OSE 为解决不间断运行和多 CPU 的分布式系统的需求而进行了专门设计，为开发商开发不同种处理器组成的分布式系统提供了最快捷的方式。对于复杂的并行系统来说，OSE 提供了一种简单的通信方式，简化了多 CPU 的处理。

- 广泛的应用

已经在电信、无线通信、数据通信、工业、航空、 汽车工业、石油化工、医疗和消费类电子等领域获得广泛应用。

- 认证

OSE 获得了 IEC 61508, SIL3、 DO-178B (levels A-D)、 EN60601-4 等认证。

- 第三方

ENEA 有强大的第三方，可以为嵌入式系统的用户提供基于完整和有效的解决方案，包括：ARM、Green Hill Software、Harris & Jeffries、Lucent Technologies、Motorola、Rational Software、Sun Microsystems、Telelogic、Texas Instruments、Trillium Digital System 等。

5. Nucleus

Nucleus PLUS 是为实时嵌入式应用而设计的一个抢先式多任务操作系统内核，其 95％的代码是用 ANSIC 写成的，因此非常便于移植并能够支持大多数类型的处理器。从实现角度来看，Nucleus PLUS 是一组 C 函数库，应用程序代码与核心函数库连接在一起，生成一个目标代码，下载到目标板的 RAM 中或直接烧录到目标板的 ROM 中执行。在典型的目标环境中，Nucleus PLUS 核心代码区一般不超过 20K 字节大小。

Nucleus PLUS 采用了软件组件的方法。每个组件具有单一而明确的目的，通常由几个 C 及汇编语言模块构成，提供清晰的外部接口，对组件的引用就是通过这些接口完成的。除

了一些少数特殊情况外，不允许从外部对组件内的全局进行访问。由于采用了软件组件的方法，Nucleus PLUS 各个组件非常易于替换和复用。

Nucleus PLUS 的组件包括任务控制、内存管理、任务间通信、任务的同步与互斥、中断管理、定时器及 I/O 驱动等。

Nucleus 具有如下特点：

- 提供源代码

Nucleus PLUS 提供注释严格的 C 源级代码给每一个用户。这样，用户能够深入地了解底层内核的运作方式，并可根据自己的特殊要求删减或改动系统软件，这对软件的规范化管理及系统软件的测试都有极大的帮助。另外，由于提供了 RTOS 的源级代码，用户不但可以进行 RTOS 的学习和研究，而且产品在量产时也不必支付 License，可以省去大量的费用。对于军方来说，由于提供了源代码，用户完全可以控制内核而不必担心操作系统中可能会存在异常任务导致系统崩溃。

- 性价比高

Nucleus PLUS 由于采用了先进的微内核（Micro-kernel）技术，因而在优先级安排，任务调度，任务切换等各个方面都有相当大的优势。另外，对 C++语言的全面支持又使得 Nucleus PLUS 的 Kernel 成为名副其实的面向对象的实时操作系统内核。然而，其价格却比较合理。所以，容易被广大的研发单位接受。

- 易学易用

Nucleus PLUS 能够结合 Paradigm，SDS 以及 ATI 自己的多任务调试器组成功能强大的集成开发环境，配合相应的编译器和动态联结库以及各类底层驱动软件，用户可以轻松地进行 RTOS 的开发和调试。另外，由于这些集成开发环境（IDE）为所有的开发工程师所熟悉，因而，容易学习和使用。

- 功能模块丰富

Nucleus PLUS 除提供功能强大的内核操作系统外，还提供种类丰富的功能模块。例如用于通讯系统的局域和广域网络模块，支持 图形应用的实时化 Windows 模块，支持 Internet 网的 WEB 产品模块，工控机实时 BIOS 模块，图形化用户接口以及应用软件性能分析模块等。用户可以根据自己的应用来选择不同的应用模块。

Nucleus PLUS 支持的 CPU 类型：

Nucleus PLUS 的 RTOS 内核可支持如下类型的 CPU：x86,68xxx,68HCxx,NEC V25, ColdFire, 29K,i960, MIPS, SPARClite, TI DSP, ARM6/7, StrongARM, H8/300H, SH1/2/3, PowerPC, V8xx, Tricore, Mcore, Panasonic MN10200, Tricore, Mcore 等。可以说 NUCLEUS 是支持 CPU 类型最丰富的实时多任务操作系统。

针对各种嵌入式应用，Nucleus PLUS 还提供相应的网络协议（如 TCP/IP，SNMP 等），以满足用户对通讯系统的开发要求。另外，可重入的文件系统、可重入的 C 函数库以及图形化界面等也给开发者提供了方便。

来配置用户的开发环境。值得提出的是 ATI 公司最近还发表了基于 Microsoft Developers Studio 的嵌入式集成开发环境－NUCLEUS EDE。从而率先将嵌入式开发工具与 Microsoft 的强大开发环境结合起来，提供给工程师们强大的开发手段。

6. eCos

eCos 是 RedHat 公司开发的源代码开放的嵌入式 RTOS 产品，是一个可配置、可移植的嵌入式实时操作系统，设计的运行环境为 RedHat 的 GNUPro 和 GNU 开发环境。eCOS 的所有部分都开放源代码，可以按照需要自由修改和添加。eCOS 的关键技术是操作系统可配置性，允许用户组和自己的实时组件和函数以及实现方式，特别允许 eCOS 的开发则定制自己的面向应用的操作系统，使 eCos 能有更广泛的应用范围。eCOS 本身可以运行在 16、32 和

64 位的体系结构、微处理器（MPU）、微控制器（MCU）以及 DSP 上，其内核、库以及运行时逐渐是建立在硬件抽象层 HAL（Hardware Abstraction Layer）上的，只要将 HAL 移植到目标硬件上，整个 eCos 就可以运行在目标系统之上了。目前 eCos 支持的系统包括 ARM、Hitachi SH3、Intel X86、MIPS、PowerPC 和 SPARC 等。eCos 提供了应用程序所需的实时要求，包括可抢占性、短的中断延时、必要的同步机制、调度规则、中断机制等。eCos 还提供了必要的一般嵌入式应用程序所需的驱动程序、内存管理、异常管理、C 语言库和数学库等。

7. μC/OS-II

一个源码公开、可移植、可固化、可裁剪、占先式的实时多任务操作系统。其绝大部分源码是用 ANSI C 写的，世界著名嵌入式专家 Jean J.Labrosse（μC/OS-II 的作者）出版了多本图书详细分析了该内核的几个版本。μC/OS-II 通过了联邦航空局（FAA）商用航行器认证，符合 RTCA(航空无线电技术委员会)DO-178B 标准，该标准是为航空电子设备所使用软件的性能要求而制定的。自 1992 年问世以来，μC/OS-II 已经被应用到数以百计的产品中。uC/OS-II 在高校教学使用是不需要申请许可证的，但将μC/OS-II 的目标代码嵌入到产品中去，应当购买目标代码销售许可证。

μC/OS-II 的特点

- 提供源代码：购买《嵌入式实时操作系统μC/OS-II(第 2 版)》可以获得μC/OS-II V2.52 版本的所有源代码，购买此书的其它版本可以获得相应版本的全部源代码。
- 可移植性（portable）：μC/OS-II 的源代码绝大部分是使用移植性很强的 ANSI C 编写，与微处理器硬件相关的部分是使用汇编语言编写。汇编语言写的部分已经压缩到最低的限度，以使μC/OS-II 便于移植到其它微处理器上。目前，μC/OS-II 已经被移植到多种不同架构的微处理器上。
- 可固化(ROMmable)：只要具备合适的软硬件工具，就可以将μC/OS-II 嵌入到产品中成为产品的一部分。
- 可剪裁(scalable)：μC/OS-II 使用条件编译实现可剪裁，用户程序可以只编译自己需要的（μC/OS-II 的）功能，而不编译不要需要的功能，以减少μC/OS-II 对代码空间和数据空间的占用。
- 可剥夺(preemptive)：μC/OS-II 是完全可剥夺型的实时内核，μC/OS-II 总是运行就绪条件下优先级最高的任务。
- 多任务：μC/OS-II 可以管理 64 个任务，然而，μC/OS-II 的作者建议用户保留 8 个给μC/OS-II。这样，留给用户的应用程序最多可有 56 个任务。
- 可确定性：绝大多数μC/OS-II 的函数调用和服务的执行时间具有确定性，也就是说，用户总是能知道μC/OS-II 的函数调用与服务执行了多长时间。
- 任务栈：μC/OS-II 的每个任务都有自己单独的栈，使用μC/OS-II 的占空间校验函数，可确定每个任务到底需要多少栈空间。
- 系统服务：μC/OS-II 提供很多系统服务，例如信号量、互斥信号量、时间标志、消息邮箱、消息队列、块大小固定的内存的申请与释放及时间管理函数等。
- 中断管理：中断可以使正在执行的任务暂时挂起，如果优先级更高的任务被中断唤醒，则高优先级的任务在中断嵌套全部退出后立即执行，中断嵌套层数可达 255 层。
- 稳定性与可靠性：μC/OS-II 是基于μC/OS 的，μC/OS 自 1992 年以来已经有数百个商业应用。μC/OS-II 与μC/OS 的内核是一样的，只是提供了更多的功能。另外，2000 年 7 月，μC/OS-II 在一个航空项目中得到了美国联邦航空管理局对商用飞机的、符合 RTCA DO – 178B 标准的认证。这一结论表明，该操作系统的质量得到了认证，可以在任何应用中使用。

思考与练习

1．举出 3 个本书中未提到的嵌入式系统的例子。

2．什么叫嵌入式系统？

3．什么叫嵌入式处理器？嵌入式处理器分那几类？

4．什么是嵌入式操作系统？为何要使用嵌入式操作系统？

第2章 嵌入式系统工程设计

2.1 嵌入式系统项目开发生命周期

2.1.1 概述

嵌入式系统的开发实际可以看作对一个项目的实施。项目的生命周期一般分为识别需求、提出解决方案、执行项目和结束项目 4 个阶段。嵌入式系统项目开发也是如此。

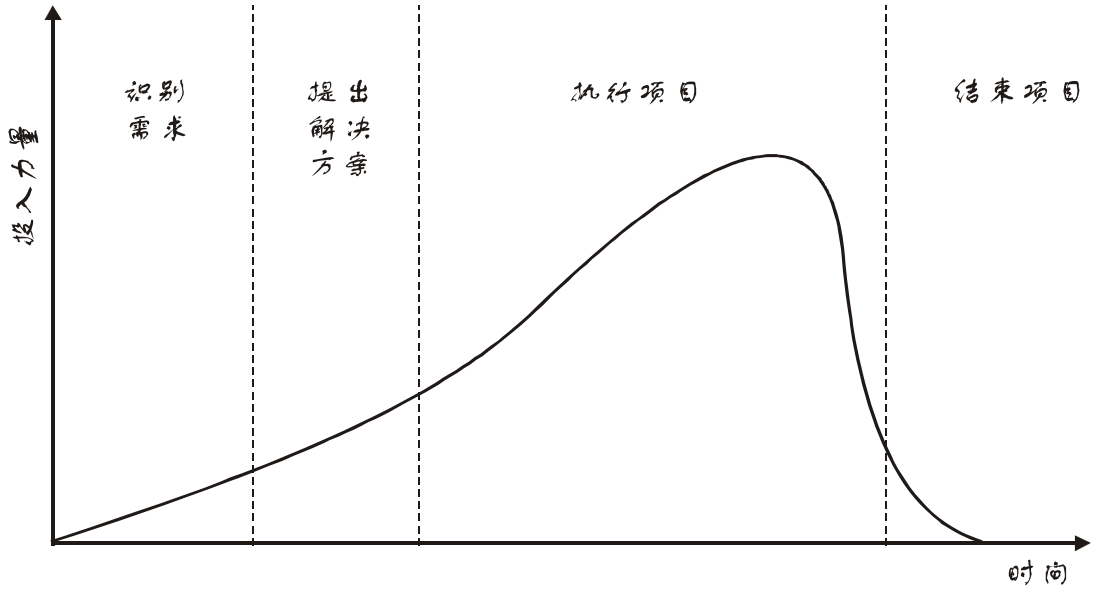


图 2.1 项目的生命周期

下面论述的项目的需求方和承包方为不同的公司。事实上，如果它们属于一个公司（组织）也是适用的，只是“公司”变成了“部门”或“小组”，或者更一般的“团队”。

1. 识别需求

识别需求是项目生命周期的最初阶段。当需求被客户确定时，项目就产生了。这个阶段的主要任务是确认需求，分析投资收益比，研究项目的可行性，分析厂商所应具备的条件。商务上这个阶段以客户提出明确的《需求建议书》或《招标书》为结束标志。这个阶段尽管可以由客户单独完成，但如果厂商介入则非常有利：一方面可了解客户真正需要什么；另一方面早期的交流可建立良好客户关系，为后续的投标和合同奠定基础。

2. 提出方案

主要由各厂商向客户提交标书、介绍解决方案。这个阶段是赢得项目的关键，公司既要展示实力又要合理报价。如果竞标成功则签定合同，厂商开始承担项目成败的责任。这个阶段容易出的问题是：因看不见最终产品，销售人员可以“随便说”，甚至过度承诺（因不用他们去执行），由此会造成公司的损失。防治的方法是一方面在合同中明确定义项目的目标和工作范围，另一方面在公司一层建立合同审核机制。

3. 执行项目

从公司角度来看这才是项目的开始。这个阶段项目经理和项目组将代表公司完全承担合同规定的任务。一般需要细化目标，制定工作计划，协调人力和其他资源；定期监控进展，分析项目偏差，采取必要措施以实现目标。

4. 结束项目

主要包括移交工作成果，帮助客户实现商务目标；系统交接给维护人员；结清各种款项。完成这些工作后一般进行项目评估。评估可以请客户参加，让其表达意见，并争取下一个商业机会，或请求将项目作为灯塔向其他客户展示。最后，举行庆祝仪式，让项目成员释放心理压力、享受成果。

2.1.2 识别需求

识别需求对于嵌入式系统项目开发是很重要的。这是因为嵌入式系统往往需要嵌入到其它产品中，不能独立工作，而这个产品往往不是嵌入式开发承包商（部门）所熟悉的，不了解需求做成的产品往往是失败的。

对于项目开发团队来说，这个阶段主要的工作就是风险分析和制订系统规范。风险分析的目的在于：在一个团队接受一个嵌入式项目之前，需要由多个层次来评估项目的可行性，如果项目团队发现项目的风险太大，就不宜再进行下去。

系统的规格是数字化的系统需求，系统规格的制定是项目进行中的最重要的一个阶段。制订系统规范实质是项目团队和系统委托客户一起讨论制订双方都可以接受的最终交货标准。系统规格将会是以后系统开发的规范，也会是系统结案的标准。

- **风险分析**

只要有项目存在，就有风险存在。

风险分析的目的在于评估项目的进行是否分出现变数。在一个项目中，有许多的因素会影响到项目进行，因此在项目进行的初期，在客户和开发团队都还未投入大量资源之前，风险的评估可以用来预估项目进行可能会遭遇的难题。如果能在项目进行的前期找出可能会发生的问题，可以决定项目是要继续下去，还是就此打住不继续往下进行。

项目的风险分析可以朝这几个方向进行思考：

需求风险

项目的目的在于制作一个可以满足需求的产品，如果需求消失，当然由项目产出的产品也将一无是处。

当客户在委托开发团队时，双方就同时担负需求风险。客户需要在项目开始进行之前，先行评估产品的市场。就可能的竞争者、使用范围或潜在的市场进行调查。而开发团队则需要对产品的属性进行需求风险评估，例如开发团队开发出来的产品是否为客户所要的？开发团队有无相关的技术？开发团队是否能在客户要求的时间内完成该项目？进行开发项目对往后的项目有无帮助？在完成项目后需要再花多少人力来进行售后服务？等等。

时间风险

在经过需求风险的评估后，项目开发团队就需针对开发所需要的时间进行评估。一般而言，客户都希望产品越快上市越好。但是在实际的项目开发上，有许多不可预期因素潜藏其中。如何在客户所要求的项目开发时间和开发团队所需要的时间中达到妥协，这是需要实际的经验去确认的。

很多的合约都是签订在整数的月份内完成的，比方说一年，或者六个月。虽然从来没有听过这是怎么推算出来的，但 Time to Market 一直是消费性产品取得市场优势的关键。因此在开发团队确定开始进行项目前，应就开发团队技术与整体的项目资源来评估，是否要接受这一项目。

资金风险

资金是项目的血液，没有资金支持的项目将快速地瓦解。除了产品开发所需要的资金外，人事、场地、设备、系统维护都需要资金。

有些计划在项目一开始，就有固定的资金来进行项目开发。在有些项目中，资金则是渐次地加入。所谓的资金风险即是，资金短缺对系统开发造成的冲击。资金风险将会影响项目

的质量，甚至影响项目是否可以继续进行。

项目管理风险

一个项目需要需要有许多专业人士的参与：需要有技术人员从事技术开发，需要有业务人员和客户进行沟通，需要有行政人员负责行政业务，需要有项目经理来带领开发团队。如果在项目中缺乏相关的管理人才，项目将无法顺利执行。在客户委托开发团队前，需要就开发团队的技术与市场口碑进行调查，视其是否有能力可以完成此项目。而开发团队需要在接下项目之前，考虑自己的开发团队是否有能力接受此项目。

有项目执行的过程中，风险处处存在。如何在风险变成悲剧之前，提早找出问题点，即是风险分析的功能。

有鉴于嵌入式项目的多变性，在不同的项目中，即会有不同的风险产生。在风险分析阶段，应就各层面可能发生的问题，集思广益，并进行自问自答。所谓当局者迷，当身陷困局之中，可能就没有办法冷静下来进行思考。

解决问题最佳方式，就是不要让问题发生。以系统工程角度而言，在项目进行的前期即进行项目的风险分析，将有效地评估项目初步可行性，并发现项目进行中可能会出现的问题。通过风险分析，及早地理清问题点，提出配套方案，将可节省许多项目资源。

- **制订系统规范**

规格制定阶段的目的在于将客户的需求，由模糊的描述，转换成有意义的量化数据。对一两个新系统的开发而言，规格的制定需要花许多的时间进行沟通。因为在两个团队开始进行合作的时候，客户端可能不清楚自己的需求是不是可以被实现出来，而开发团队这一边则不清楚客户端真正的需求。

规格制定的好处在于理清系统的边界。需求是一个模糊的概念，一直要等到有真实的数字出来后，系统实现才能有依据。比方说，客户需要一个可以测量、记录湿度的设备。在还没确定这个设备要测量多大范围的湿度及可以记录多久的数据之前，开发团队是无法进行下一步骤的。

制订系统规范主要从下面几个方面着手：

系统功能

就系统功能而言，这个系统可能会接收哪些输入？输入数理量为何？以什么方式进行输入？需要进行前期处理吗？物理量的范围是否确定？有没有什么特别的需求？诸如取样频率、放大等等。

系统需要哪些输出？是否需要进行类比数字转换？可能需要驱动哪些外设？输出范围为何?有没有特殊的需求?如输出频率、输出信号种类等等。

通过输入端取得的数据是否需要经过处理？数据需不需要进行存储？要不要进行特殊的数据处理后，再送到输出端（如处理多媒体的 Codec）？

系统限制

系统限制在于发现系统使用上或开发上的限制，可能的限制如下：

嵌入式系统可能被部署在各种环境中。温度、湿度、震动、电磁波干扰、电源供应、工业安全标准，以及是否要在特定时间内完成某项任务等等都是嵌入式系统可能会遇到的工作环境限制，由于这和系统所处的环境有关，所以要和专业的人士做进一步的确认。

另外，价格的限制会影响到系统的设计与组成组件。在价格成本的限制下，开发团队需要寻找适当的方案来应付。

系统开发资源

经费是系统开发的血液，没有经费，项目就无法正常运行。经费也直接影响到开发所需要的人力。项目的开发展讲究时间的掌控，客户需要在一定的时间内将产品推至生产线才有

可能得到预期中的效果。因此在项目进行的过程中，时间也是一个重要的资源。

专业人力的质量决定系统的质量与所需开发时间，如果在团队中没有相关的专业人才，就需要以外包的方式维护系统开发的顺畅。

2.1.3 提出方案

对于嵌入式系统项目来说，这个阶段的主要工作是系统规划与设计。在设计规划阶段中，开发团队需要分析所有可行的解决方案，并拟定进程，使项目在合理的进程范围中逐渐建构完成。在系统的设计方面最重要的一件事就是确定系统的框架。

- **系统规划**

规划阶段是项目进行的第一个重要的决策点。在与客户进行完系统规格制定后，项目团队需要对系统规格作更进一步的分析，来决定是否以这一个版本的系统规格进行下一阶段的工作。如果确定系统规格可行的话，项目团队需要针对系统的开发拟出发展进程。

在系统规划阶段中，项目团队需要从系统开发的专业角度，评估现行的系统规格是否合理，是否可以在现有资源下完成，需不需要再进行修改等。如果发现目前的系统规格无法完成或部分无法完成，就必须回到系统规格阶段，重新讨论出双方皆可接受的新系统规格。

在确定系统规格可行后，项目状态将会从原本由客户与开发团队的合作关系。转换成由开发团队主导开发，客户进行项目追踪的状态。因此项目团队需要对系统的开发进行预估，让客户可以进一步地掌握系统开发的进程，并确定检查点，以让双方确定项目是否如预期中的进度完成。下面就系统规划的两个阶段进行讨论：

规格分析

规格分析的目的在于给开发团队一个机会去检查系统规格的可行性。

在客户与开发团队完成系统规格后，并不能确保系统规格一定可以被完全实现。由于在制定系统规格阶段中，领域专家已经将一些专业的术语与数据转换成工程人员可以接受的词汇，因此开发团队可以就此进行深入的评估，在系统还未进入真正的设计与实现阶段前，开发团队可凭以往的经验、现行的成熟技术、研发的能力、项目资源等等信息来评估这一版本的系统规格是否可行。

预估项目进程

预估项目进程是需要经验累积的一件工作。多数的嵌入项目总是由一些旧的经验与新的设计组成。对之前有经验的部分，开发团队或许会有一个参考的进程数据，但是对于开发团队而言，如果有新的技术导入项目中，那就难以去预估出一个正确的时间。在进程的预估上，开发团队也只能视以往的经验来预估某个阶段的开发需要花多少时间，至于某某项目合理来说要花多少时间，可真是依项目而异的。

一般来说，如果开发团队有良好的项目记忆（文件管理，软件版本控制）,旧有的经验就很容易在新的项目中重新使用。但是对新导进技术而言，不管是新技术、新界面、新工具，工程人员都要花时间去研读规格、去尝试、去验证。因此很难预估一个新的东西要多久才能就绪，也就使得项目进程更难以预估。

不过不管项目进程预估得是否准确，项目进程完成预估后，就会对项目团队施加一种无形的推力。它让项目人员有一个需要完成某个任务的时间感（或者，可以说是压迫感），也让客户可以通过这个预估的进程来确保项目正朝完成的目标前进。

在项目进程的预估上，需要设置适当的检查点。如果只有预估项目完成的时间点，那项目有很大的可能性会有进程上的延迟。在整个系统的开发进程预估上，需要加入许多中途的检查点（也有书上说是里程碑，milestone）让双方的人马可以确定每个阶段的进程，再适度地调整系统开发的进程。如果进度超前，那当然可以缩短预估进程，如果进度落后就必须提出解决方案，或者提出延后产品完成时间的建议。

- **系统设计**

在系统设计的阶段中，开发团队需要寻找适当的组件组成系统，以达到在系统规格阶段所制定的系统规格。在决定了系统的关键组件后，必须由系统的架构开始设计，然后再进行系统的细节设计。

嵌入式系统的核心在于控制全局的智能组件，这个智能组件可能是微控器，也可能是数字信号处理器（DSP）或是可现场编程门数组（FPGA）、可编程逻辑组件（CPLD）。不过这些智能组件的存在都是为了达到系统规格的标准，也因些系统规格是系统设计的目标与标准。

在这里，就系统设计的几个大方向作一说明：

设计系统架构

许多的嵌入式系统都可以画出系统的系统功能块图（Function Block Diagram）来描述系统。这一张系统功能块图可能大致说明系统的功能分配。在系统还未正式开始设计之前，有一个明确的架构是必需的。就像一个房子，在还没开始进行搭建之前，需要有一张明确的结构设计蓝图，才得以让其他的工作开始进行。

系统架构的目的在于符合系统规格中的“功能”，但对于使用哪一种组件并无进一步的规范。组件的选择是为了达到系统规范的“范围”。

寻找适当方案

有了系统架构后，系统开发团队就可以再进一步地去讨论使用哪一个适当的方案来达到系统规格的要求。为了达到系统规格，开发团队可能需要去寻找不同的方案，从智能组件的运算能力到提供的外设都需要审慎地选择。

系统设计

嵌入式系统和一般信息系统最大的不同在于嵌入式系统所使用的硬件与所使用的软件流程可能是独一无二的。由于嵌入式系统是由软件与硬件所组成，因此在这里就这两方面进行说明：

在硬件的初步估计中，可以将系统的功能各个击破，一个一个去评估所使用的硬件是否可以达到系统规格的要求。在初步设计阶段中，也有可能使用现成的开发板（EVB）来进行特殊外设与主动组件之间的配合度测试。并搭配适当的测试程序，证实硬件系统搭配动作正确无误码。

由于智能组件本身的限制，系统在外设的外接上可能受到种种的限制。如果发现智能组件无法完成系统需求时，就应及时更改硬件组件方案。在另一方面，由于各种外设在系统上是以不同的总线进行连接，所以对不同的硬件的接线也要有进一步的了解。

现在多数的智能组件厂商会提供所谓的公板，并有逻辑线路图与评估板可供参考，在设计阶段不妨多多利用这些资源，减少硬件设计上的错误。

在细节设计阶段中，需要就之前的雏形作进一步的加强，如设计属于自家的 PCB，外设与外设的配合，外设与主要主动组件间的集成是这个阶段的重点。

对于执行单一工作的嵌入式系统，开发团队可以先行制作出系统流程图来描述软件应有的功能。而复杂的嵌入式系统就应该以更高级的方式去描绘出系统的行为。

在确定软件功能分割后，接下来要针对不同的软件功能进行设计。由于嵌入式系统可能会接触到许多外设，因此在设计软件的时候，需要将不同的界面要用到的资源先行预留（需和硬件设计团队的做事前沟通）。在目前的微控器设计中由于总线的地址范围等资源受限，所以需要在系统设计阶段先行讨论各外设可以占用的系统资源为何，例如内存映射空间、中断服务向量、DMA 的分配。

2.1.4 执行项目

这个阶段的主要工作就是系统的实现和系统的测试。由于嵌入式系统的特殊性，嵌入式系统项目的实现一般实现系统的硬件，又需要在硬件上实现相应的软件。这就牵涉到硬件的实现和软件的实现，而这两方面又互相牵扯。系统实现完成后，需要测试其是否符合我们的要求，这就需要系统测试。依据不同的系统层次及集成程度，测试团队需要对系统作不同程度的测试。测试过程中如果发现问题，需要通过调试找出问题的所在并解决它。事实上，系统的实现、测试和调试贯穿整个“执行项目”阶段。

- **系统实现**

由于不同的嵌入式系统会有不同的设计考虑，在实现阶段就需要不同的系统架构来进行系统实现。

对嵌入式系统而言，大体上可以将架构分为两类。一种是没有操作系统的嵌入式系统。相对的，另一种则是有使用操作系统的嵌入式系统。

对单纯的系统来说，只要是输出、输入与运算较为单纯，或者整个系统可以利用前景背景式的方式描述出来，都可以考虑在没有操作系统的协助之下，完成系统所需要的工作。对于较复杂的嵌入式系统而言，有一个操作系统来提供基本的操作需求是必须的。

下面就针对不同的系统架构所对应的开发程序作一些说明。

从硬件做起

新一代的消费性电子产品或是嵌入式系统产品中，硬件很少是需要自己从头开始做起的。很多的微控器厂商会提供所谓的公板。这个公板会将该微控器可以做到的功能尽量地放到一个参考板上来。使用该微控器的厂商依据这个参考设计，再配合自己的需求，将公板的设计转换成自己的设计。这样不但节省设计的时间，也可以确保系统硬件的可靠性。

在另一方面，由于参考板都会有一些基本的驱动程序范例可以参考，所以厂商在驱动程序移植的进程上可以缩短，并减少实现人员的负担。

驱动程序是硬件和软件之间的桥梁。由于驱动程序的移植牵涉到不同硬件的规格，并且需要提供合适的 API 供操作系统调用，所以在实现驱动程序上有其困难性。

在基本的驱动程序移植完成后，开发团队会开始进行操作系统的移植。由于现在商用的操作系统都有不错的设计，并对普及的微控器有很好的支持，所以在经过一段时间的研究后，就可以将操作系统移植到平台上。

在操作系统移植完成后，应用程序就可以在目标平台上进行验证，到此为止，系统的开发才真正开始。

从驱动程序移植开始做起

为了软硬件同步的开发，许多的项目会从使用微控器合作厂商的参考平台（EVB）开始进行软件的开发。

使用合作厂商出品的评估板的好处在于有一个稳定的开始。EVB 提供了验证过的硬件与简单的驱动程序。开发团队可以利用这些基础，来进行客制的驱动程序。相同的，在特定的驱动程序开发完成后（通常为 UART，FLASH，LCD 等等）就可以开始进行操作系统的移植，等操作系统移植完成后，系统的开发才算正式开始。

某些操作系统厂商会提供给开发厂商更方便的工具与操作系统移植的方式，让厂商只需要针对他们的目标平台补上一层 HAL（Hardware Abstraction Layer）后，就可以让操作系统和驱动程序接轨，所以很快就可以进行应用程序的开发。

从现成平台开始做起

曾经使用过 x86 平台的人大概都能体会到这样的开发流程。从项目一开始，开发团队就已经拿到有操作系统在上面的目标板。这时候开发团队只需要针对应用程序进行开发就可以。有些时候开发团队也需要撰写驱动程序，不过这时候就需要针对不同的操作系统来进行

驱动程序的撰写。

- **系统测试**

测试的目的在于提早找出问题所在，并验证系统设计符合系统规格。

在整个开发的历程中，一开始，许多的小模块会被先建立，然后慢慢地组合成大的子系统，接着再组成系统。在不同的开发阶段中都需要相对应的测试来检验每个阶段的工作是否被正确地执行、有没有潜藏的问题。而检查这些问题，就是系统测试的目的。

就和系统工程的流程一样，有明确的项目目标才有明确的产品。进行测试也需要有明确的目标，才会有明确的结果，根据不同的项目，在测试进行之前，开发团队需要了解并完成下列的工作：

确认测试目标，就是找出系统可能的问题，但是系统在什么情况下有可能会发生什么问题就需要先行确定。漫无目标的测试不具有任何的意义，有明确的测试目标才能发挥系统测试的功能。

有了测试的目标之后，就要确定测试的标准。测试的标准在于设置资格，如果通过测试标准，就是 OK，如果无法通过这个标准，就需要再改进。

由于一个嵌入式系统里，有硬件也有软件，要如何就测试该项目设置测试的标准是需要测试团队根据系统的规格来加以规范的。

有了测试的标准之后，接下来就要建立测试的项目。在测试阶段中，想到什么就测什么是没有办法系统地对系统进行测试的。事先拟定测试项目不但可以让测试的过程更加顺利，也让后面的测试报告更有说服力。

好不容易进行完测试，如果没有记录下来而忘记了重要的测试数据岂不是做白工？在进行测试的同时就将测试的结果记录下来，以供后人参考。测试的目的在于找出系统潜在的问题并确认系统是否符合系统规格，因此测试报告可说是一份系统的体检表。不但可以用来检验系统是否正常运行，也可以提供客户一个产品的快照，让客户对项目的进程有进一步的了解。

在嵌入式系统中测试一般以下种类：

功能测试

功能测试的目的在于检验一个小的程序模块（如副程序，对象等）或硬件的子系统（内存，周边等）是否如预期地运行。功能测试是系统测试中的最小单位，也是系统测试的基础。在软件的测试方面，利用建立好的功能测试程序，可以有效的检查系统是否符合设计需求。在另一方面，使用设计良好的硬件测试程序也可以确认底层硬件是否正常地运作。

集成测试

在完成功能模块的实现与测试后，开发团队需要对系统进行集成测试。系统中的子系统是一些功能模块的组合，比方说嵌入式系统的文件系统就需要操作系统、文件系统、内存驱动程序互相配合来完成存取文件的功能。在功能测试阶段，每一个模块理论上都已经通过了基础的测试，但是这并不代表此功能模块可以和别和模块完美地集成。集成测试的目的在于将模块与模块之间的问题找出来，让系统在子系统层次上的表现可以得到一个质量的认证。

系统测试

在所有的软硬件模块组合成系统之后，就需要对系统进行系统测试。系统测试的目的在于测试系统是否达到系统规格的标准。开发团队需要用不同的测试程序或方法来测试系统的行为是否和预期中一样，是否可以通过所有的测试向量。在系统测试中，可能会发现在系统集成过后才会发生的问题。进行到系统测试阶段，问题会变得更加复杂，因此，更显出前一个步骤中集成测试的重要性。

环境测试

环境测试的目的在于使系统可以在真实的操作情况下运行，让系统在正式出货之前，还有机会可以进行系统稳定性的确认。

由于嵌入式系统需在不同的环境中使用，因此针对系统的需求，开发完成的产品在交给客户之前，需要进行环境测试，以确定整个系统可以在其操作环境下顺利运行。不同的环境，可能会带给系统不一样的冲击，这在系统设计时，开发团队就应该将之列入设计考虑。但是由于使用环境或系统操作者的多变性，使得系统的稳定程序倍受挑战。在环境测试中，由于系统处在一个真实的操作环境，所以可以进一步地找出例外状况。

为了加快环境测试的进度，有一些系统会进行加速测试，如卫星系统会放入模拟的环境中，以较真实环境和恶劣的状况测试卫星，让可能的问题提早出现。

在通过环境测试后，整个系统的功能与稳定性皆能为客户所接受，就可以开始进行移交了。

出货测试

出货测试（或是移交测试）就是让用户以用户的角度，来验收这一个系统。由于用户和工程员的观点是不一样的，所以他可能会以各种工程师事先没有预想到的方式对系统进行操作，如果这一阶段，系统顺利通过用户的测试，系统的发展可以算是告一段落。当然，随着系统使用时间的增加，会衍生出许多额外的问题，关于这一点，就是为何项目需要有售后服务阶段的原因了。

- **系统调试**

在嵌入式项目进行到实施阶段后，就有可能会出现一些非预期的结果，这时候就需要对有问题的地方进行调试。系统测试与系统调试是孪生姐妹，开发团队利用系统测试找出可能的系统问题，再利用系统调试将问题找出并解决。与其说调试是一门技术，不如把说它是一门艺术。由于大型的嵌入式项目中有许多的模块相互牵连，并存在许多不同的不确定性，要如何从这些盘根粗节中找出真正的问题，需要相当的背景知识与经验和一些想象力。

由于嵌入式系统是由软件和硬件所组成，所以把问题分层软件问题和硬件问题是合乎逻辑的。除此以外，还有一种“软硬件”问题，也就是无法确定是硬件问题还是软件问题的问题。这种“软硬件”问题可能是整个系统中最难找出原因的问题。

很少人喜欢调试，所以最好的办法就是在错误发生之前就发现它，或者根本不要让错误隐藏在设计中。

调试可以分成软件调试和硬件调试，除了使用适当的工具进行调试之外，开发人员需要有相对的知识和经验才能把真正的问题找出来，而不是利用别的方法绕过去。以短时间的观点来看，利用不动的方式进行调试虽然很方便，但以长期的观点来说，为了找出因为补洞而产生的错误所花的时间，可能比当初用心去找出真正的错误所花的时间还多地多。因此，对于嵌入式开发人员最好的一句建议是：问题不解决，永远存在。

2.1.5 结束项目

产品开发完毕并移交给客户并不等于项目已经结束。客户在使用产品的过程中还会发现一连串的问题，此时开发团队还需要服务客户，这就是售后服务。售后服务是一种保障客户权利的措施，相对的也是开发团队的义务。当售后服务也结束，项目结案了，项目也没有结束，这时需要项目讨论来总结、学习一些东西。项目讨论是一个项目的反馈机制，通过这一程序，项目团队的经验才可以被记录下来，也就是说，这是一个撰写项目历史的过程。

- **售后服务**

虽然项目的产出经过了无数的测试，但是系统执行的环境却一直在改变。一个设计良好的系统当然可以在设计的环境中顺利运行，但是难免会遇到设计点之外的状况。除此之外，厂商可以根据用户的需求要求开发团队加入新的功能，这也是在原先的设计中无法完全掌握的。

一般来说，在系统出货后产生的问题，会比在设计阶段中预想到的问题更加复杂，主要的原因是系统所处的环境远比在实验室中多变。也因此，才能进一步解决问题。

另一方面，一个嵌入式系统可能在市场上大受好评，而需要加入新的功能，开发团队必须根据现在的系统进行升级，或者重新设计系统，这些都在售后服务的范畴中。

系统生命周期小如消费性电子商品（如电子宠物），大如飞机上的航电系统，在系统正式转交给客户后，一直到系统“退役”之前，都需要有人去维护它，甚至去升级它。因此根据不同的系统生命周期，相关的维护计划就需要被拟定。对于生命周期小的产品，可能还没用到系统故障，就被消费者丢置一旁。而生命周期长的系统，在其使用年限中，都需要有后备的零件可以替换。

- **项目讨论**

项目的讨论一个项目进行的反馈机制。通过这一个程序，项目团队的经验才可以被记录下来，也就是说，这是一个撰写项目历史的过程。

事实上，项目在进行的同时，项目的相关文件即是一个项目的历史。而在项目完成后的项目讨论，将对整个项目做一个整体的检讨，看看哪里做得好，哪里值得改进，这些经验都是下一个项目的踏脚石，也是团队最重要的资产。记录下来的项目经验，不会随着人员的流动而消失，但是项目团队的数据如果未经整理与分类，等时间一久，数据渐渐变多了以后，对于团队与新进的人员来说，不但不是一种好的经验来源，反而会变成一种额外的负担。

因此，现在很多公司提倡所谓的知识管理（Knowledge Management, KM）。知识管理本身的立意即在于保存众人的知识与经验，并经有效地分类与整理后，让体制内的人都可以享受到这些宝贵的经验，并将产出再次放进知识库中。对于嵌入式系统的开发团队来说，这些东西可能是大到像是项目系统工程的经验，也可能是次系统调试的程序。换句话说，知识库将项目团队的知识保存的电子媒介中，等待有一天，某些人在碰到某一些之前曾经遇到的问题时，可以在有参考的基准上，不需再重蹈覆辙，即可享有前人流血流汗所得的宝贵经验。而对一个公司或团体来说，知识管理系统的建构确保了公司投资可以完整地留在公司内，不会因为一两个关键人物的离职或跳槽，就把相关的知识一并带走。

2.2 嵌入式系统工程设计方法简介

2.2.1 由上而下与由下而上

由上而下（Top Down Approach）是一个正统的设计方式，也就是说，所有的设计皆是遵循系统工程的流程来进行，确定需求、制订系统规格、设计、实现、测试、皆是一步一步、按部就班地进行。

相对的就是由下而上（Button Up Approach）。由下而上的意思就是说，一个系统是由已经有的基础（或组件）为起点，开始往上延伸，最后将系统完成。所以在先天上已经有所限制。

其实大部分的项目都是这两种方式的混合体，很少有整个项目都是从上而下的，相同的，也很少有整个项目都是都是由下而上的。在产品的设计上，即使是由上而下的设计方式也需要考虑到现实因素。刚开始设计的人也许就会设计出他心中“完美”的系统。例如说，他会使用非常特殊的电阻来匹配电路，而使用一个全世界都没有人生产的螺丝来固定系统。如果在系统的设计阶段就考虑这些问题，就不会闹出刚才的笑话了。

2.2.2 UML 系统建模

UML（Unified Modeling Language）是一种原本设计用来描述对象导向程序语言开发的图形化语言。由于它具有描述事物的多重性，所以理论上也可以被拉到其他领域使用。

在实际使用上，根据不同的使用情况，UML 提供了不同的图形来描述系统。在 UML 中，包括了下面几种图形：

1．类图（Class Diagram）
2．对象图（Object Diagram）
3．用例图（Use Case Diagram）
4．顺序图（Sequence Diagram）
5．协作图（Collaboration Chart Diagram）
6．状态图（State Chart Diagram）
7．活动图（Activity Diagram）
8．组件图（Component Diagram）
9．部署图（Deployment Diagram）

虽然 UML 的初始目的在于描述软件系统，尤其是对象导向项目的设计与规划。但由于 UML 本身包括了许多前人的智慧，因此 UML 也就更具有变形性和应用性，可以应用在不同的问题领域上，当然可以在嵌入式系统设计过程中应用。

使用 UML 的好处

语言的用处在于沟通。UML 也是一种语言，它利用视觉化的方法来制定、构建以及记录对象导向系统。因此，可以把 UML 当作一种软件工程用的语言。

使用 UML 的好处在于可以在短时间内了解别人要传达的消息，而不是花时间在了解消息本身如何解读。UML 提供给用户基础的工具与基本的规范，在这个基础上，用户可以利用这个语言去描述他所想要描述的系统，用不同的界面去描绘出系统的不同方面。

就嵌入式项目系统而言，从不同的视角有不同的表现，需要不同的方法去记录与描述。传统的程序流程图无法详细描述到系统的每一个细节，只有利用适当的方法，才能将一个系统的每一个功能在设计阶段就被审慎地考虑。在系统的设计阶段将系统的构架稳定下来，再不会在系统完成实现时才发现系统有潜在的问题。

语言发明的本意在于沟通，而不在于制造误解。虽然 UML 本身提供了丰富的词汇，但并不表示项目成员需要学习每一个 UML 细节。适当的 UML 界面可以让参与项目的人员对系统更加了解、更容易互相讨论、对系统进行修改，以及保存项目的历史。

在嵌入式系统项目中，许多的情况都需要事先预想到的。嵌入式系统本质上就是一种强壮性设计，加上许多系统用封闭型设计，无法像开放系统一样能轻易地进行维护。在设计阶段使用 UML 来描述系统的模型，可以及早确定系统的方向、规划系统的功能，并提早发现问题。更能记忆团队的项目历程（或智慧），提供给下一次项目中使用。

2.2.3 面向对象 OO 的思想

随着系统的需求日益增加，系统的功能及复杂程度不断增大，为了使系统开发变得容易，我们要逐步改进我们对系统的思考方式以及我们开发系统的的方式，这项新技术我们称之为面向对象的的开发。

对象是客观世界中具有独立属性及能力的实体，有着某种特征(状态)和行为。在面向对象的开发中我们常常遇到面向对象的分析(Object-Oriented Analysis)、面向对象的设计(Object-Oriented Design)和面向对象的编程(Object-Oriented Programming)等。面向对象的分析是所有软件分析活动的第一步，仔细的划分系统的各个部分，然后将各个部分作为一个对象进行功能或行为上的分析和定义；面向对象的设计是将面向对象分析所建立的分析模型转变为软件构造蓝图的设计模型，即在预定义的基本类框架上构建一个系统，这个阶段中只要进一步确定各个对象的功能以及各个对象之间的关系；面向对象的编程是指使用面向对象的设计语言(如 JAVA、C++、Ada 等)把面向对象设计的系统模型程序化，亦即是完成具体实现。编码是软件开发过程中最基本、最底层的需要，它强调的是一种分析及解决问题的

思路，而不在乎他所使用的语言工具。

在传统的结构化方法看来，它是将系统分解为很多基本函数的集合，数据被孤立分离，并且不考虑并发。而面向对象方法则不同，它的基本分解单位为对象。在面对较复杂的系统设计时，我们可以将它作为一个对象来进行分析。一个系统作为一个对象，它可以由多个部分组成。同样，这个对象也可以分解为多个对象；若从代码实现的视角分析，面向对象代码侧重于对象之间的交互，多个对象各司其职，相互协作以完成目标。

思考与练习

1．嵌入式系统项目开发的生命周期分那几个阶段？各自的具体任务是什么？
2．为何要进行风险分析？嵌入式项目主要有哪些方面的风险？
3．何谓系统规范？制定系统规范的目的是什么？
4．何谓系统规划？为何要做系统规划？
5．为什么在项目结束前需要进行项目讨论？

第3章 ARM7体系结构

3.1 简介

3.1.1 ARM

ARM 是 Advanced RISC Machines 的缩写，是微处理器行业的一家知名企业，该企业设计了大量高性能、廉价、耗能低的 RISC 处理器、相关技术及软件。技术具有性能高、成本低和能耗省的特点。适用于多种领域，比如嵌入控制、消费/教育类多媒体、DSP 和移动式应用等。

ARM 将其技术授权给世界上许多著名的半导体、软件和 OEM 厂商，每个厂商得到的都是一套独一无二的 ARM 相关技术及服务。利用这种合伙关系，ARM 很快成为许多全球性 RISC 标准的缔造者。

目前，总共有 30 家半导体公司与 ARM 签订了硬件技术使用许可协议，其中包括 Intel、IBM、LG 半导体、NEC、SONY、菲利浦和国家半导体这样的大公司。至于软件系统的合伙人，则包括微软、升阳和 MRI 等一系列知名公司。

ARM 架构是面向低预算市场设计的第一款 RISC 微处理器。

3.1.2 ARM 的体系结构

ARM 的设计实现了非常小，但是高性能的结构。ARM 处理器结构的简单使 ARM 的内核非常小，这样使器件的功耗也非常低。

ARM 是精简指令集计算机（RISC），因为它集成了非常典型的 RISC 结构特性：

- 一个大的、统一的寄存器文件
- 装载/保存结构，数据处理的操作只针对寄存器的内容，而不直接存储器进行操作。
- 简单的寻址模式，所有装载/保存的地址都只由寄存器内容和指令域决定。
- 统一和固定长度的指令域，简化了指令的译码。

此外，ARM 体系结构还提供：

- 每一条数据处理指令都对算术逻辑单元（ALU）和移位器控制，以实现对 ALU 和移位器的最大利用。
- 地址自动增加和自动减少的寻址模式实现了程序循环的优化。
- 多寄存器装载和存储指令实现最大数据吞吐量。
- 所有指令的条件执行实现最快速的代码执行。

这些在基本 RISC 结构上增强的特性使 ARM 处理器在高性能、低代码规模、低功耗和小的硅片尺寸方面取得良好的平衡。

体系结构板本

从最初开发到现在，ARM 指令集体系结构有了巨大的改进，并在不断完善和发展。为了清楚地表达每个 ARM 应用实例所使用的指令集，ARM 公司定义了 5 种主要的 ARM 指令集体系结构版本，以版本号 v1~v5 表示。

1. 版本 1（v1）在 ARM1 使用，由于只有 26 位的寻址空间（现已废弃不用），从未商业化，该版本包括：

- 基本的数据处理指令（不包括乘法）；
- 字节、字和半字加载 / 存储指令（load/store）；
- 分支指令（branch），包括在子程序调用中使用的分支和链接指令（branch-and-link）；

- 在操作系统调用中使用的软件中断指令（software interrupt）。

2. 版本 2（v2）仍然只有 26 位寻址空间（现已废弃不用），但相对版本 1 增加了以下内容：

- 乘法和乘加指令；
- 协处理器支持；
- 快速中断模式中的两个以上的分组寄存器；
- 原子性（atomic）加载 / 存储指令 SWP 和 SWPB（稍后版本中称作 v2a）。

3. 版本 3（v3）将寻址范围扩展到 32 位；先前存储于 R15 的程序状态信息存储在新的当前程序状态寄存器（CPSR）中，且增加了程序状态保存寄存器（SPSR），以便出现异常时保存 CPSR 中的内容。此外，版本 3 还增加了两种处理器模式，以便在操作系统代码中有效地使用数据中止异常、取指中止异常和未定义指令异常。相应地，版本 3 指令集发生如下改变：

- 增加了两个指令 MRS 和 MSR，允许访问新的 CPSR 和 SPSR 寄存器：
- 修改过去用于异常返回指令的功能，以便继续使用。

4. 版本 4（v4）不再强制要求与以前的版本兼容以支持 26 位体系结构，清楚地指明哪个指令会引起未定义指令异常发生。版本 4 在版本 3 的基础上增加了如下内容：

- 半字加载 / 存储指令：
- 字节和半字的加载和符号扩展（sign-extend）指令；
- 在 T 变量中，转换到 Thumb 状态的指令；
- 使用用户（User）模式寄存器的新的特权处理器模式。

5. 版本 5（v5）在版本 4 的基础上，对现在指令的定义进行了必要的修正，对版本 4 体系结构进行了扩展，并增加了指令，具体如下：

- 改进在 T 变量中 ARM/Thumb 状态之间的切换效率：
- 允许非 T 变量和 T 变量一样，使用相同的代码生成技术：
- 增加计数前导零(count leading zeros)指令，允许更有效的整数除法和中断优先程序；
- 增加软件断点（software breakpoint）指令；
- 对乘法指令如何设置标志进行了严格的定义。

3.1.3 ARM 处理器核简介

ARM 公司开发了很多系列的 ARM 处理器核，目前最新的系列已经是 ARM11 了，但 ARM6 核及更早的系列已经很罕见了，ARM7 以后的核也不是都获得广泛应用。目前，应用比较多的是 ARM7 系列、ARM9 系列、ARM9E 系列、ARM10 系列、SecurCore 系列和 Intel 的 StrongARM、Xscale 系列，下面简单介绍一下这几个系列。

1. ARM7 系列

ARM7 系列包括 ARM7TDMI、ARM7TDMI－S、带有高速缓存处理器宏单元的 ARM720T 和扩充了 Jazelle 的 ARM7EJ－S。该系列处理器提供 Thumb 16 位压缩指令集和 EmbeddedICEJTAG 软件调试方式，适合应用于更大规模的 SoC 设计中。其中 ARM720T 高速缓存处理宏单元还提供 8KB 缓存、读缓冲和具有内存管理功能的高性能处理器，支持 Linux、Symbian OS 和 Windows CE 等操作系统。

ARM7 系列广泛应用于多媒体和嵌入式设备，包括 Internet 设备、网络和调制解调器设备，以及移动电话、PDA 等无线设备。无线信息设备领域的前景广阔，因此，ARM7 系列也瞄准了下一代智能化多媒体无线设备领域的应用。

2. ARM9 系列

ARM9 系列有 ARM9TDMI、ARM920T 和带有高速缓存处理器宏单元的 ARM940T。所有的 ARM9 系列处理器都具有 Thumb 压缩指令集和基于 EmbeddedICE JTAG 的软件调试方

式。ARM9 系列兼容 ARM7 系列，而且能够比 ARM7 进行更加灵活的设计。

ARM9 系列主要应用于引擎管理、仪器仪表、安全系统、机顶盒、高端打印机、PDA、网络电脑以及带有 MP3 音频和 MPEG4 视频多媒体格式的智能电话中。

3. ARM9E 系列

ARM9E 系列为综合处理器，包括 ARM926EJ－S 和带有高速缓存处理器宏单元的 ARM966E－S、ARM946E－S 和带有高速缓存处理器宏单元的 ARM966E－S。该系列强化了数字信号处理（DSP）功能，可应用于需要 DSP 与微控制器结合使用的情况，将 Thumb 技术和 DSP 都扩展到 ARM 指令集中，并具有 EmbeddedICE-RT 逻辑（ARM 的基于 EmbeddedICE JTAG 软件调试的增强版本），更好地适应了实时系统的开发需要。同时其内核在 ARM9 处理器内核的基础上使用了 Jazelle 增强技术，该技术支持一种新的 Java 操作状态，允许在硬件中执行 Java 字节码。

4. ARM10 系列

ARM10 系列包括 ARM1020E 和 ARM1020E 微处理器核。其核心在于使用向量浮点（VFP）单元 VFP10 提供高性能的浮点解决方案，从而极大提高了处理器的整型和浮点运算性能，为用户界面的 2D 和 3D 图形引擎应用夯实基础，如视频游戏机和高性能打印机等。

5 SecurCore

SecurCore 系列涵盖了 SC100、SC110、SC200 和 SC210 处理核。该系列处理器主要针对新兴的安全市场，以一种全新的安全处理器设计为智能卡和其他安全 IC 开发提供独特的 32 位系统设计，并具有特定的反伪造方法，从而有助于防止对硬件和软件的盗版。

6. StrongARM 和 Xscale

StrongARM 处理器将 Intel 处理器技术和 ARM 体系结构融为一体，致力于为手提式通信和消费电子类设备提供理想的解决方案。Intel Xscale 微体系结构则提供全性能、高性价比、低功耗的解决方案，支持 16 位 Thumb 指令和集成数字信号处理（DSP）指令。

3.2 ARM7TDMI

3.2.1 简介

ARM7TDMI 基于 ARM 体系结构 V4 板本，是目前低端的 ARM 核（并非芯片，ARM 核与其它部件如 RAM、ROM、片内外设组合在一起才构成现实的芯片），具有广泛的应用，其最显著的应用为数字移动电话。

ARM7TDMI 是从 ARM6 核发展而来的。ARM6 核（在 ARM 体系结构中）最早实现了 32 地址空间编程模式（早期的 ARM 为 26 位地址），但现在已经被取代。ARM6 所使用的电路技术使它很难稳定的在低于 5V 的电源电压下工作。ARM7 弥补了这一不足，且在短时间增加了 64 位乘法指令（带 M 后缀的）、支持片上调试（带 D 后缀的）、高密度 16 位的 Thumb 指令机扩展（带 T 后缀的）和 EmbededICE 观察点硬件（带 I 后缀的），形成 ARM7TDMI。

ARM7TDMI-S 是 ARM7TDMI 的可综合（synthesizable）版本（软核）。对应用工程师来说，除非芯片生产厂商对 ARM7TDMI-S 进行了裁减，否则在逻辑上 ARM7TDMI-S 与 ARM7TDMI 没有太大区别，其编程模型与 ARM7TDMI 一致。如果没有特殊说明，本章对 ARM7TDMI 和 ARM7TDMI-S 不加区分，通称为 ARM7TDMI。

ARM7TDMI 处理器是 ARM 通用 32 位微处理器家族的成员之一。ARM 处理器具有优异的性能，但功耗却很低，使用门的数量也很少。ARM 结构是基于精简指令集计算机(RISC)原理而设计的。指令集和相关的译码机制比复杂指令集计算机要简单得多。这样的简化实现了：

- 高的指令吞吐量

- 出色的实时中断响应
- 小的、高性价比的处理器宏单元

3.2.2 三级流水线

ARM7TDMI 处理器使用流水线来增加处理器指令流的速度。这样可使几个操作同时进行，并使处理和存储器系统连续操作，能提供 0.9MIPS/MHz 的指令执行速度。

流水线使用 3 个阶段，因此指令分 3 个阶段执行：

- 取指
- 译码
- 执行

3 阶段流水线如图 3.1 所示。

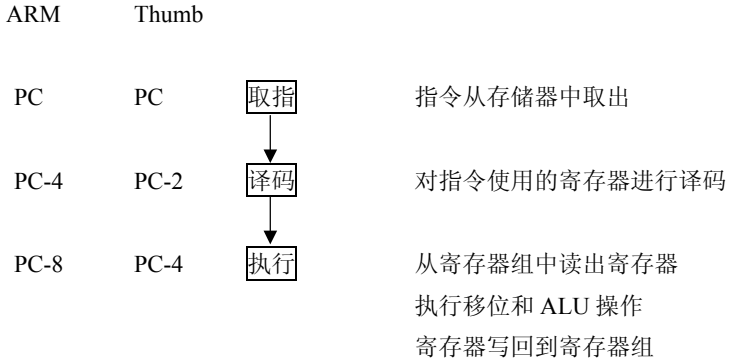


图 3.1 指令流水线

注：程序计数器(PC)指向被取指的指令，而不是指向正在执行的指令。

在正常操作过程中，在执行一条指令的同时对下一条指令进行译码，并将第三条指令从存储器中取出。

3.2.3 存储器访问

ARM7TDMI 处理器使用了冯•诺依曼（Von Neumann）结构，指令和数据共用一条 32 位总线。只有装载、存储和交换指令可以对存储器中的数据进行访问。

数据可以是 8 位字节、16 位半字或者 32 位字。字必须分配为占用 4 字节，而半字必须分配为占用 2 字节。

3.2.4 存储器接口

ARM7TDMI 处理器的存储器接口可以使潜在的性能得到实现，这样减少了存储器的使用。对速度有严格要求的控制信号使用流水线，这样使系统控制功能以标准的低功耗逻辑实现。这些控制信号使许多片内和片外存储器技术所支持的“快速突发访问模式”得到充分利用。

ARM7TDMI 处理器的存储器周期有 4 种基本类型：

- 内部周期
- 非连续的周期
- 连续的周期
- 协处理器寄存器传输周期

3.3 ARM7TDMI 的模块和内核框图

ARM7TDMI 的模块框图见图 3.2，内核框图见图 3.3，功能框图见图 3.4。

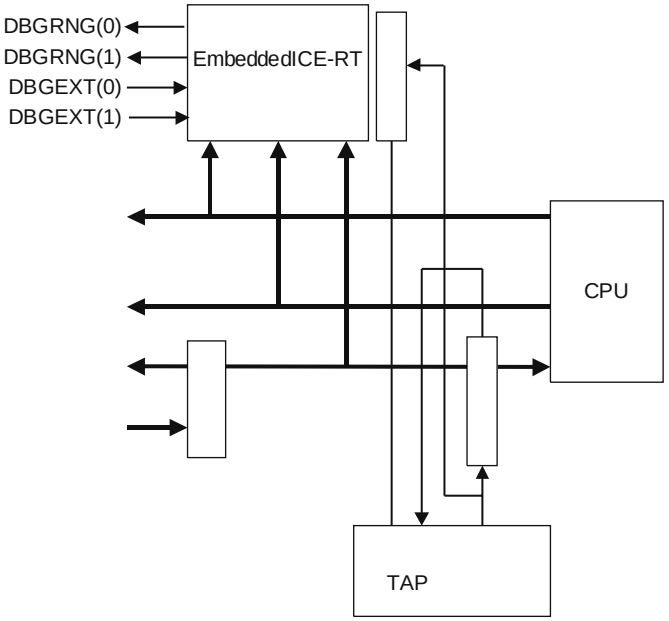


图 3.2 ARM7TDMI 模块

注：数据总线上没有双向路径。图 3.2 对这些作了简化。

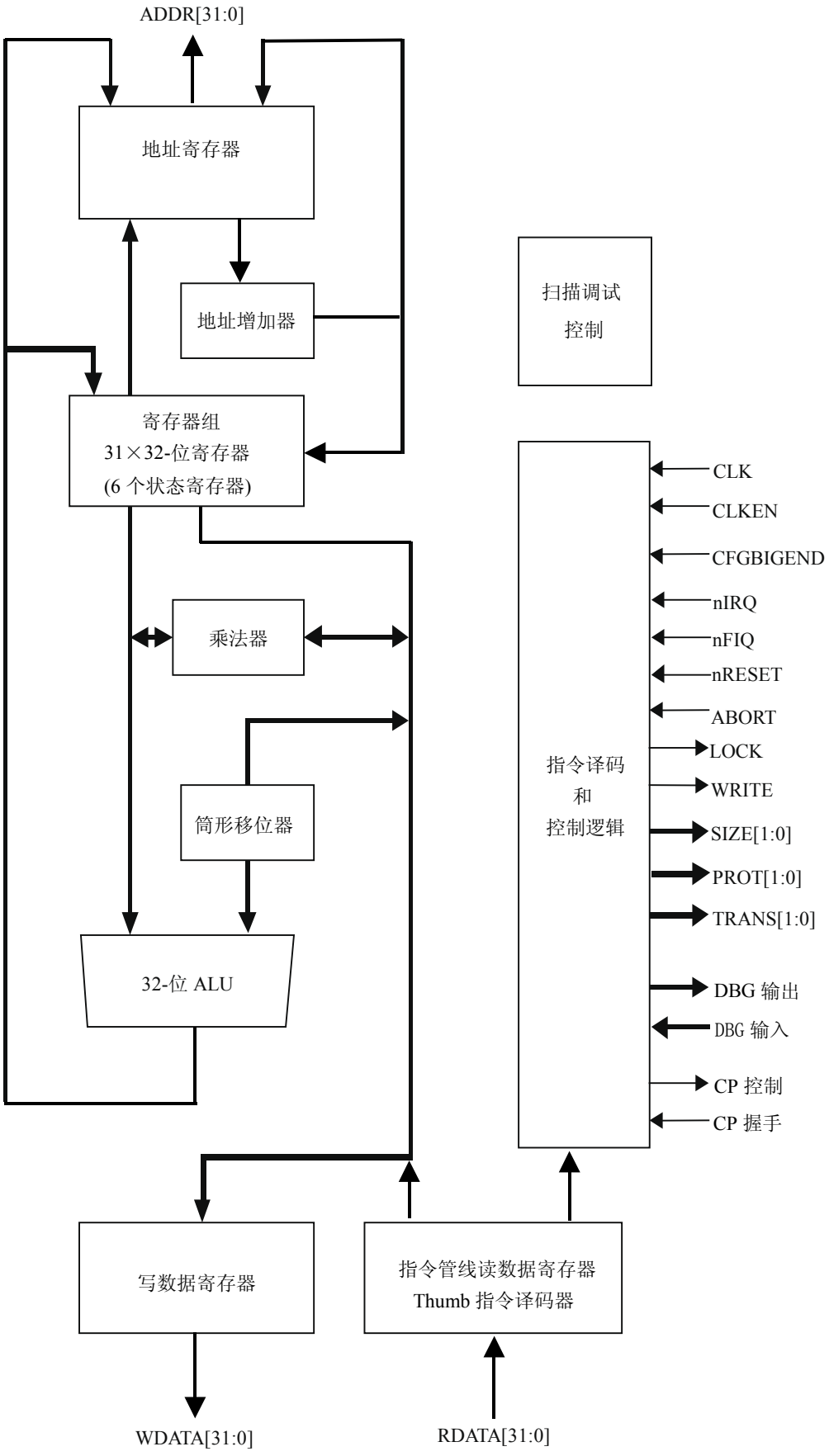


图 3.3 ARM7TDMI 内核

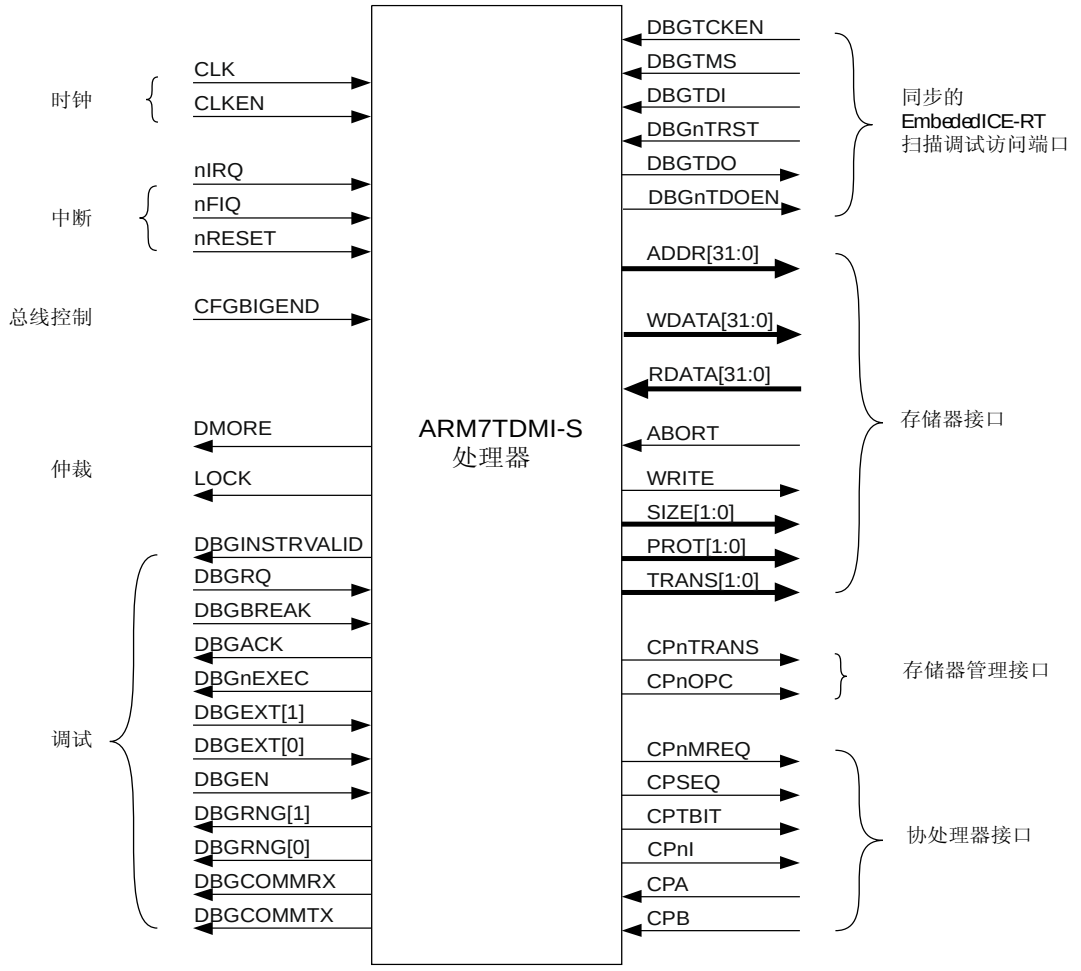


图 3.4 ARM7TDMI 功能框图

3.4 体系结构直接支持的数据类型

ARM 处理器支持下列数据类型：

字节 8 位

半字 16 位（必须分配为占用 2 个字节）

字 32 位（必须分配为占用 4 个字节）

注：

- ARM 结构第 4 版及以上版本都支持这 3 种数据。ARM 结构第 4 版之前的版本只支持字节和字（ARM7TDMI 是基于 ARM 结构第 4 版的）。
- 当任意一种类型描述为 unsigned 时，N 位数据值使用正常的二进制格式表示范围为 0 ~ $+2^N$-1 的非负整数。
- 当任意一种类型描述为 signed 时，N 位数据值使用 2 的补码格式表示范围为-2^{N-1} ~ $+2^{N-1}$-1 的整数。
- 所有数据操作，例如 ADD，都以字为单位。
- 装载和保存指令可以对字节、半字和字进行操作，当装载字节或半字时自动实现零扩展或符号扩展。
- ARM 指令的长度刚好是 1 个字（分配为占用 4 个字节）。Thumb 指令的长度刚好是一个半字（占用 2 个字节）。

3.5 处理器状态

ARM7TDMI 处理器内核使用 ARM v4T 结构实现，该结构包含 32 位 ARM 指令集和 16 位 Thumb 指令集。因此 ARM7TDMI 处理器有两种操作状态：

ARM 状态　32 位，这种状态下执行的是字方式的 ARM 指令

Thumb 状态　16 位，这种状态下执行半字方式的 Thumb 指令

在 Thumb 状态中，程序计数器（PC）使用 bit1 来选择切换半字。

注：ARM 和 Thumb 状态间的切换并不影响处理器模式或寄存器内容。

您可以使用 BX 指令将 ARM7TDMI 内核的操作状态在 ARM 状态和 Thumb 状态之间进行切换，其例子见程序清单 3.1。详见第 4 章　。

所有的异常处理都在 ARM 状态中执行。如果异常发生在 Thumb 状态中，处理器会切换 ARM 状态。在异常处理返回时自动切换回 Thumb 状态。详细情况参考 3.9.3 小节。

程序清单 3.1　态切换的例子

```
;从 ARM 状态转变为 Thumb 状态
        LDR     R0, =Lable+1
        BX      R0
;从 Thumb 状态转变为 ARM 状态
        LDR     R0, =Lable
        BX      R0
```

3.6 处理器模式

ARM 体系结构支持 7 种处理器模式：用户模式、快中断模式、中断模式、管理模式、中止模式、未定义模式和系统模式。ARM7TDMI 完全支持这七种模式，具体参考表 3.1。除用户模式外，其它模式均为特权模式。ARM 内部寄存器和一些片内外设在硬件设计上只允许（或可选为只允许）特权模式下访问。此外，特权模式可以自由的切换处理器模式，而用户模式不能直接切换到别的模式。

表 3.1　处理器模式

处理器模式	说明	备注
用户　（usr）	正常程序工作模式	不能直接切换到其它模式
快中断　（fiq）	支持高速数据传输及通道处理	FIR 异常响应时进入此模式
中断　（irq）	用于通用中断处理	IRQ 异常响应时进入此模式
管理　（svc）	操作系统保护代码	系统复位和软件中断响应时进入此模式
中止　（abt）	用于支持虚拟内存和/或存储器保护	在 ARM7TDMI 没有大用处
未定义　（und）	支持硬件协处理器的软件仿真	未定义指令异常响应时进入此模式
系统　（sys）	用于支持操作系统的特权任务等	与用户类似，但具有可以直接切换到其它模式等特权

有五种处理器模式称为异常模式，它们是：快中断模式、中断模式、管理模式、中止模式和未定义模式。它们除了可以通过程序切换进入外，也可以由特定的异常进入。当特定的异常出现时，处理器进入相应的模式。每种模式都有某些附加的寄存器，以避免异常退出时用户模式的状态不可靠。详细寄存器的说明见 3.7 节。

至于系统模式，它与用户模式一样，不能由异常进入，且使用与用户模式完全相同的寄存器。然而它是特权模式，不受用户模式的限制。有这个模式，操作系统要访问用户模式的寄存器就比较方便。同时，操作系统的一些特权任务可以使用这个模式以访问一些受控的资源而不必担心异常的出现时的任务状态变得不可靠。

3.7 内部寄存器

3.7.1 简介

在 ARM7TDMI 处理器内部有 37 个寄存器用户可见的寄存器。

- 31 个通用 32 位寄存器，在 ARM 公司文件中它们的名称为：R0~R15、R13_svc、R14_svc、R13_abt、R14_abt、R13_und、R14_und、R13_irq、R14_irq 和 R8_fiq~R14_fiq。
- 6 个状态寄存器，在 ARM 公司文件中它们的名称为：CPSR、SPSR_svc、SPSR_abt、SPSR_und、SPSR_irq 和 SPSR_fiq。

这些寄存器并不是在同一时间全都可以被访问的。处理器状态和操作模式决定了程序员可以访问哪些寄存器。

3.7.2 ARM 状态寄存器集

(1) 各模式可访问的寄存器

在 ARM 状态中，16 个通用寄存器和 1 个或 2 个状态寄存器可在任何时候同时被访问。在特权模式中，与模式相关的分组寄存器可以被访问。表 3.2 所示为每种模式所能访问的寄存器。

表 3.2 **ARM 状态各模式下的寄存器**

<table>
<tr><th rowspan="2">寄存器类别</th><th rowspan="2">寄存器在汇编中的名称</th><th colspan="7">各模式实际访问的寄存器</th></tr>
<tr><th>用户</th><th>系统</th><th>管理</th><th>中止</th><th>未定义</th><th>中断</th><th>快中断</th></tr>
<tr><td rowspan="17">通用寄存器和程序计数器</td><td>R0(a1)</td><td colspan="7">R0</td></tr>
<tr><td>R1(a2)</td><td colspan="7">R1</td></tr>
<tr><td>R2(a3)</td><td colspan="7">R2</td></tr>
<tr><td>R3(a4)</td><td colspan="7">R3</td></tr>
<tr><td>R4(v1)</td><td colspan="7">R4</td></tr>
<tr><td>R5(v2)</td><td colspan="7">R5</td></tr>
<tr><td>R6(v3)</td><td colspan="7">R6</td></tr>
<tr><td>R7(v4)</td><td colspan="7">R7</td></tr>
<tr><td>R8(v5)</td><td colspan="6">R8</td><td>R8_fiq</td></tr>
<tr><td>R9(SB,v6)</td><td colspan="6">R9</td><td>R9_fiq</td></tr>
<tr><td>R10(SL,v7)</td><td colspan="6">R10</td><td>R10_fiq</td></tr>
<tr><td>R11(FP,v8)</td><td colspan="6">R11</td><td>R11_fiq</td></tr>
<tr><td>R12(IP)</td><td colspan="6">R12</td><td>R12_fiq</td></tr>
<tr><td>R13(SP)</td><td colspan="2">R13</td><td>R13_svc</td><td>R13_abt</td><td>R13_und</td><td>R13_irq</td><td>R13_fiq</td></tr>
<tr><td>R14(LR)</td><td colspan="2">R14</td><td>R14_svc</td><td>R14_abt</td><td>R14_und</td><td>R14_irq</td><td>R14_fiq</td></tr>
<tr><td>R15（PC）</td><td colspan="7">R15</td></tr>
<tr><td rowspan="2">状态寄存器</td><td>CPSR</td><td colspan="7">CPSR</td></tr>
<tr><td>SPSR</td><td colspan="2">无</td><td>SPSR_svc</td><td>SPSR_abt</td><td>SPSR_und</td><td>SPSR_irq</td><td>SPSR_fiq</td></tr>
</table>

注意：括号内为 ATPCS 中寄存器的命名，可以使用 RN 汇编伪指令将寄存器定义多个名字。其中，

ADS1.2 的汇编程序直接支持这些名称，但注意 a1~a4，v1~v8 必须用小写字母。详细请参见参考文献[1]。

(2) 一般的通用寄存器

在汇编语言中寄存器 R0～R13 为保存数据或地址值的通用寄存器。其中寄存器 R0~R7 为未分组的寄存器。这意味着对于任何处理器模式，它们中的每一个都对应于相同的 32 位物理寄存器。它们是完全通用的寄存器，不会被体系结构作为特殊的用途，并且可用于任何使用通用寄存器的指令。

寄存器 R8~R14 为分组寄存器。它们所对应的物理寄存器取决于当前的处理器模式。几乎所有允许使用通用寄存器的指令都允许使用分组寄存器。

寄存器 R8~R12 有两个分组的物理寄存器。一个用于除 FIQ 模式之外的所有寄存器模式(R8~R12)，另一个用于 FIQ 模式(R8_fiq~R12_fiq)。

寄存器 R8~R12 在 ARM 体系结构中没有特定的用途。不过对于那些只使用 R8~R14 就足够处理的简单中断来说，FIQ 所单独使用的这些寄存器可实现快速的中断处理。

寄存器 R13 和 R14 分别有 6 个分组的物理寄存器。一个用于用户和系统模式，其余 5 个分别用于 5 种异常模式。

(3) 堆栈指针 R13

寄存器 R13 通常作为堆栈指针（SP）。在 ARM 指令集当中，由于没有以特殊方式使用 R13 的指令或其它功能，只是习惯上都这样使用。但是在 Thumb 指令集中存在使用 R13 的指令，详细参考 3.7.3 小节。

每个异常模式都有其自身的 R13 分组版本，它通常指向由异常模式所专用的堆栈。在入口处，异常处理程序通常将其它要使用的寄存器值保存到这个堆栈。通过返回时将这些值重装到寄存器中，异常处理程序可确保异常发生时的程序状态不会被破坏。

(4) 链接寄存器 R14

寄存器 R14（也称为链接寄存器或 LR）在结构上有两个特殊功能：

- 在每种模式下，模式自身的 R14 版本用于保存子程序返回地址。当使用 BL 或 BLX 指令（注意：ARM7TDMI 没有 BLX 这条指令）调用子程序时，R14 设置为子程序返回地址。子程序返回通过将 R14 复制到程序计数器来实现。通常有下列两种方式：
  执行下列指令之一：
  ```
  MOV   PC,LR
  BX    LR
  ```
  或是在子程序入口，使用下列形式的指令将 R14 存入堆栈：
  ```
  STMFD   SP!,{<registers>,LR}
  ```
  并使用匹配的指令返回：
  ```
  LDMFD   SP!,{<registers>,PC}
  ```
- 当发生异常时，将 R14 对应的异常模式版本设置为异常返回地址（有些异常有一个小常量的偏移）。异常返回的执行类似于子程序返回，只是使用稍微不同的指令来确保被异常中断的程序状态能够完全恢复。

寄存器 R14 在其它任何时候可作为一个通用寄存器。

注意：当嵌套异常发生时，这两个异常可能会发生冲突。例如，如果用户在用户模式下执行程序时发生了 IRQ 中断，用户模式寄存器不会被破坏。*但如果运行在 IRQ 模式下的中断处理程序重新使能 IRQ 中断，并且发生了嵌套的 IRQ 中断时，外部中断处理程序保存在 R14_irq 中的任何值都将被嵌套中断的返回地址所覆盖。*

系统程序员应当小心处理这样的事件，通常处理的方法是确保 R14 的对应版本在发生嵌套中断时不再保存任何有意义的值（可行的方法：将 R14 入栈）。当使用直接的方法难于

处理时，最好在进入异常处理程序后，重新使能中断或允许嵌套异常发生之前，切换到其它处理器模式。（在 ARM 结构第 4 版和以上版本，系统模式通常是这种情况下最好的模式。）

(5) 程序计数器 R15

寄存器 R15 保存程序计数器（PC），它总是用于特殊的用途。它经常可用于通用寄存器 R0~R14 所使用的位置（即在指令编码中 R15 与 R0~R14 的地位一样，只是指令执行的结果不同），因此可以认为它是一个通用寄存器。但是对于它的使用还有许多与指令相关的限制或特殊情况。这些将在具体的指令描述中提到。通常，如果 R15 使用的方式超出了这些限制，那么指令将是不可预测的。

读取程序计数器的一般限制

当指令对 R15 的读取没有超过任何对 R15 使用的限制时，读取的值是指令的地址加上 8 个字节。由于 ARM 指令总是以字为单位，结果的 bit[1:0]总是为 0。

这种读取 PC 的方式主要用于对附近的指令和数据进行快速、与位置无关的寻址，包括程序中与位置无关的转移。

当使用 STR 或 STM 指令保存 R15 时，出现了上述规则的一个例外。这些指令可将指令地址加 8 字节保存（和其它指令读取 R15 一样）或将指令自身地址加 12 字节（将来还可能出现别的数据）。偏移量 8 还是 12（或是其它数值）取决于 ARM 的实现（也就是说，与芯片有关）。对于某个具体的芯片，它是个常量。这样使用 STR 和 STM 指令是不可移植的。

由于这个例外，最好避免使用 STR 和 STM 指令来保存 R15。如果很难做到，那么应当在程序中使用合适的指令序列确定当前使用的芯片所使用的偏移量。例如，使用程序清单 3.2 所示的指令序列将这个的偏移量存入 R0 中。

程序清单 3.2 取具体芯片关于存储 PC 时的偏移量

```
SUB     R1, PC, #4          ; R1 = 下面 STR 指令的地址
STR     PC, [R0]            ; 保存 STR 指令地址+偏移量
LDR     R0, [R0]            ; 然后重装
SUB     R0, R0, R1          ; 计算偏移量
```

写程序计数器的一般限制

当执行一条执行写 R15 的指令没有超出任何对它使用的限制时，写入 R15 的正常结果值被当成一个指令地址，程序从这个地址处继续执行（相当于执行一次无条件跳转）。

由于 ARM 指令以字为边界，因此写入 R15 的值的 bit[1:]通常为 0b00。具体的规则取决于所使用的结构的版本：

- 在 ARM 结构 V3 版及以下版本中，写入 R15 的值的 bit[1:0]被忽略，因此指令的实际目标地址（写入 R15 的值）和 0xFFFFFFFC 相与。
- 在 ARM 结构 V4 版（ARM7TDMI 基于 V4 版）及以上版本中，写入 R15 的值的 bit[1:0]必须为 0b00。如果不是，结果将不可预测。

(6) 程序状态寄存器

所有模式共享一个程序状态寄存器（CPSR），在异常模式中，另外一个寄存器程序状态保存寄存器（SPSR）可以被访问。每种异常具有自己的 SPSR，在进入异常时它保存 CPSR 的当前值，异常退出时可同它恢复 CPSR。关于程序状态寄存器的描述见 3.8 节。

3.7.3 Thumb 状态寄存器集

(1) 各模式可访问的寄存器

Thumb 状态寄存器集是 ARM 状态集的子集。程序员可直接访问：

- 8 个通用寄存器 R0～R7

- PC
- 堆栈指针（SP）
- 连接寄存器（LR）
- CPSR（有条件的访问）

每个特权模式都有分组的 SP 和 LR。Thumb 寄存器的详细情况见表 3.3。

表 3.3 Thumb 状态各模式下的寄存器

寄存器类别	寄存器在汇编中的名称	各模式实际访问的寄存器						
		用户	系统	管理	中止	未 定义	中断	快中断
通用寄存器和程序计数器	R0(a1)	R0						
	R1(a2)	R1						
	R2(a3)	R2						
	R3(a4)	R3						
	R4(v1)	R4						
	R5(v2)	R5						
	R6(v3)	R6						
	R7(v4,WR)	R7						
	SP	R13		R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
	LR	R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
	PC	R15						
状态寄存器	CPSR	CPSR						

注意：括号内为 ATPCS 中寄存器的命名，可以使用 RN 汇编伪指令将寄存器定义多个名字。其中，ADS1.2 的汇编程序直接支持这些名称，但注意 a1~a4，v1~v4 必须用小写字母。详细请参考参考文献[1]。

(2) 一般的通用寄存器

在汇编语言中寄存器 R0～R7 为保存数据或地址值的通用寄存器。对于任何处理器模式，它们中的每一个都对应于相同的 32 位物理寄存器。它们是完全通用的寄存器，不会被体系结构作为特殊的用途，并且可用于任何使用通用寄存器的指令。

(3) 堆栈指针 SP

堆栈指针 SP 对应 ARM 状态的寄存器 R13。每个异常模式都有其自身的 SP 分组版本，它通常指向由异常模式所专用的堆栈。在入口处，异常处理程序通常将其它要使用的寄存器值保存到这个堆栈。通过返回时将这些值重装到寄存器中，异常处理程序可确保异常发生时的程序状态不会被破坏。要注意的是某种原因使处理器进入异常时，处理器自动进入 ARM 状态。

(4) 链接寄存器 LR

链接寄存器 LR 对应 ARM 状态寄存器 R14，在结构上有两个特殊功能，详细情况参考 3.7.2 小节关于*链接寄存器 R14 部分*。唯一要注意的是某种原因使处理器进入异常时，处理器自动进入 ARM 状态。

(5) ARM 状态寄存器和 Thumb 状态寄存器之间的关系

Thumb 状态寄存器与 ARM 状态寄存器有如下的关系：

- Thumb 状态 R0～R7 与 ARM 状态 R0～R7 相同
- Thumb 状态 CPSR 和 SPSR 与 ARM 状态 CPSR 和 SPSR 相同
- Thumb 状态 SP 映射到 ARM 状态 R13

- Thumb 状态 LR 映射到 ARM 状态 R14
- Thumb 状态 PC 映射到 ARM 状态 PC(R15)

这些关系如图 3.5 所示。

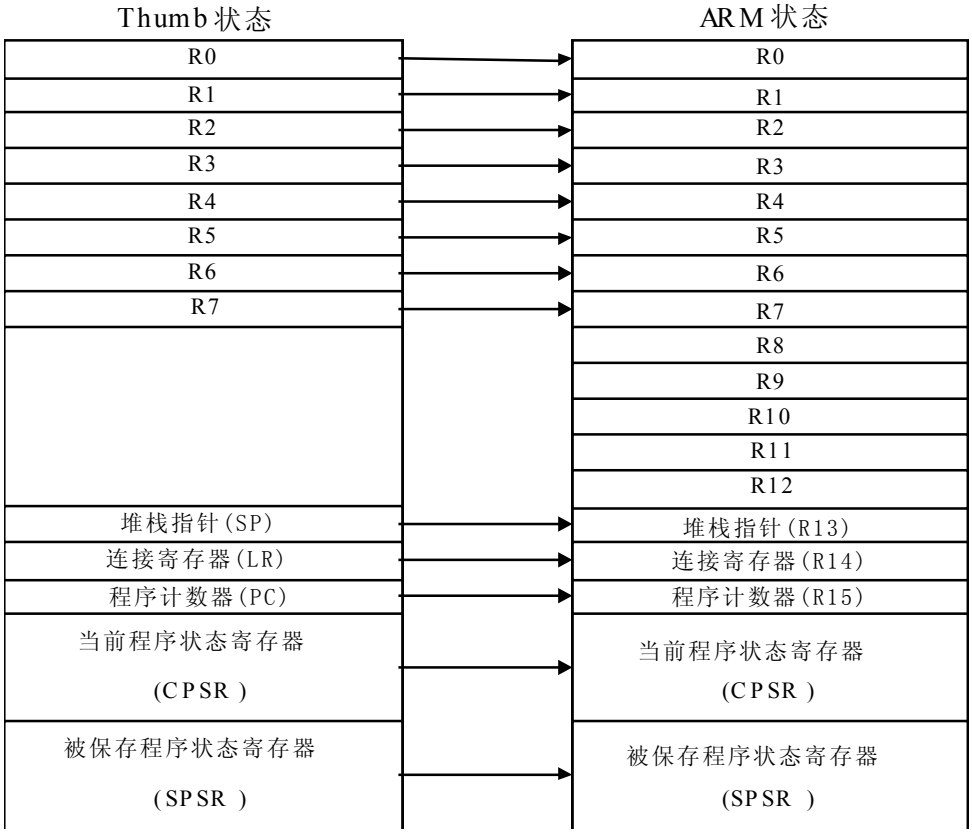


图 3.5 Thumb 寄存器在 ARM 状态寄存器上的映射

注：寄存器 R0～R7 为低寄存器。寄存器 R8～R15 为高寄存器。

(6) 在 Thumb 状态中访问高寄存器

在 Thumb 状态中，高寄存器（R8～R15）不是标准寄存器集的一部分。汇编语言程序员对它们的访问受到限制，但可以将它们用于快速暂存。

可以使用 MOV 指令的特殊变量将一个值从低寄存器（R0～R7）转移到高寄存器，或者从高寄存器转移到低寄存器。CMP 指令可用于比较高寄存器和低寄存器的值。ADD 指令可用于将高寄存器的值与低寄存器的值相加。详细信息请参考第 4 章 。

3.8 程序状态寄存器

3.8.1 简介

ARM7TDMI 内核包含 1 个 CPSR 和 5 个 SPSR 供异常处理程序使用。ARM7TDMI 内核所有处理器状态都保存在 CPSR 中。当前的操作处理器状态位于程序状态寄存器（CPSR）当中。CPSR 包含：

- 4 个条件代码标志（负(N)、零(Z)、进位(C)和溢出(V)）
- 2 个中断禁止位，分别用于一种类型的中断
- 5 个对当前处理器模式进行编码的位
- 1 个用于指示当前执行指令(ARM 还是 Thumb)的位

每个异常模式还带有一个程序状态保存寄存器(SPSR)，它用于保存任务在异常发生之前的 CPSR。CPSR 和 SPSR 通过特殊指令进行访问。详细信息请参阅第二章。

CPSR 的各位的分配如图 3.6 所示。

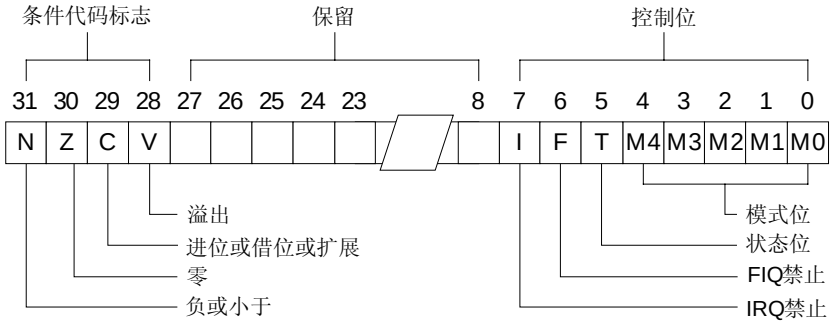


图 3.6 程序状态寄存器的格式

注：为了保持与将来的 ARM 处理器兼容，并且作为一种良好的习惯，在更改 CPSR 时，我们强烈建议您使用读－修改－写的方法。

3.8.2 条件代码标志

大多数数值处理指令可以选择是否修改条件代码标志。一般的，如果指令带 S 后缀，则指令会修改条件代码标志；但有一些指令总是改变条件代码标志。

N, Z, C 和 V 位都是条件代码标志。可以通过算术和逻辑操作来设置这些位。这些标志还可通过 MSR 和 LDM 指令进行设置。ARM7TDMI 处理器对这些位进行测试以决定是否执行一条指令。

各标志位的含义如下：

N 运算结果的 b31 位值。对于有符号二进制补码，结果为负数时 N=1，结果为正数或零时 N=0；

Z 指令结果为 0 时 Z=1(通常表示比较结果“相等”)，否则 Z=0；

C 使用加法运算(包括 CMN 指令)，b31 位产生进位时 C=1，否则 C=0。使用减法运算(包括 CMP 指令)，b31 位产生借位时 C=0，否则 C=1。对于结合移位操作的非加法/减法指令，C 为 b31 位最后的移出值，其它指令 C 通常不变；

V 使用加法/减法运算，当发生有符号溢出时 V=1，否则 V=0，其它指令 V 通常不变。

在 ARM 状态中，所有指令都可按条件来执行。在 Thumb 状态中，只有分支指令可条件执行。更详细的信息请参考第 4 章 。

3.8.3 控制位

CPSR 的最低 8 位为控制位。它们分别是：

- 中断禁止位
- T 位
- 模式位

当发生异常时，控制位改变。当处理器在一个特权模式下操作时，可用软件操作这些位。

中断禁止位

I 和 F 位都是中断禁止位：

- 当 I 位置位时，IRQ 中断被禁止
- 当 F 位置位时，FIQ 中断被禁止

T 位

T 位反映了正在操作的状态：

- 当 T 位置位时，处理器正在 Thumb 状态下运行
- 当 T 位清零时，处理器正在 ARM 状态下运行

警告：绝对不要强制改变 CPSR 寄存器中的 T 位。如果这样做，处理器会进入一个无法预知的状态。

模式位

M4, M3, M2, M1 和 M0 位（M[4:0]）都是模式位。这些位决定处理器的操作模式，见表 3.4。不是所有模式位的组合都定义了有效的处理器模式，因此请小心不要使用表中所没有列出的组合。

表 3.4 **CPSR 模式位值**

M[4:0]	模式	可见的 Thumb 状态寄存器	可见的 ARM 状态寄存器
10000	用户	R0~R7, SP, LR, PC, CPSR	R0~R14,PC, CPSR
10001	快中断	R0~R7,SP_fiq,LR_fiq,PC,CPSR, SPSR_fiq	R0~R7,R8_fiq~R14_fiq,PC,CPSR, SPSR_fiq
10010	中断	R0~R7,SP_irq,LR_irq,PC,CPSR, SPSR_fiq	R0~R12,R13_irq,R14_irq,PC,CPSR, SPSR_irq
10011	管理	R0~R7,SP_svc,LR_svc,PC,CPSR, SPSR_svc	R0~R12,R13_svc,R14_svc,PC,CPSR, SPSR_svc
10111	中止	R0~R7,SP_abt,LR_abt,PC,CPSR, SPSR_abt	R0~R12,R13_abt,R14_abt,PC,CPSR, SPSR_abt
11011	未定义	R0~R7,SP_und,LR_und,PC,CPSR, SPSR_und	R0~R12,R13_und,R14_und,PC,CPSR, SPSR_und
11111	系统	R0~R7,SP,LR,PC,CPSR	R0~R14,PC, CPSR

注：如果将非法值写入 M[4:0]中，处理器将进入一个无法恢复的模式。

3.8.4 保留位

CPSR 中的保留位被保留将来使用。当改变 CPSR 标志和控制位时，请确认没有改变这些保留位。另外，请确保您的程序不依赖于包含特定值的保留位，因为将来的处理器可能会将这些位设置为 1 或者 0。

3.9 异常

3.9.1 简介

只要正常的程序流被暂时中止，处理器就进入异常模式。例如响应一个来自外设的中断。在处理异常之前，ARM7TDMI 内核保存当前的处理器状态，这样当处理程序结束时可以恢复执行原来的程序。

如果同时发生两个或更多异常，那么将按照固定的顺序来处理异常，见 3.9.11 小节。

3.9.2 异常入口/出口汇总

表 3.5 所示为异常入口处变量 R14 所保存的 PC 值以及退出异常处理程序所推荐使用的指令。

表 3.5 异常入口/出口

异常或入口	返回指令	之前的状态		备注
		ARM R14_x	Thumb R14_x	
BL	MOV PC,R14	PC+4	PC+2	此处 PC 为 BL, SWI, 未定义的指令取指或者预取中止指令的地址。
SWI	MOVS PC,R14_svc	PC+4	PC+2	
未定义的指令	MOVS PC,R14_und	PC+4	PC+2	
预取中止	SUBS PC,R14_abt,#4	PC+4	PC+4	
快中断	SUBS PC,R14_fiq,#4	PC+4	PC+4	此处 PC 为由于 FIQ 或 IRQ 占先而没有被执行的指令的地址
中断	SUBS PC,R14_irq,#4	PC+4	PC+4	
数据中止	SUBS PC,R14_abt,#8	PC+8	PC+8	此处 PC 为产生数据中止的装载或保存指令的地址。
复位	无	—	—	复位时保存在 R14_svc 中的值不可预知。

注意：”MOVS PC,R14_svc” 是指在在管理模式执行 MOVS PC,R14 指令。”MOVS PC,R14_und”、”SUBS PC,R14_abt,#4”等指令也是类似的。

如果异常处理程序已经把返回地址拷贝到堆栈，可以使用一条多寄存器传送指令来恢复用户寄存器并实现返回。程序清单 3.3 以普通中断为例说明这种情况。

程序清单 3.3 断处理代码的开始部分与退出部分

```
SUB         LR,LR, #4              ; 计算返回地址
STMFD       SP!,{R0-R3,LR}         ; 保存使用到的寄存器
….
LDMFD       SP!,{R0-R3,PC}^        ; 中断返回
```

这里，中断返回指令的寄存器列表（其中必须包括 PC）后的“^”符号表示这是一条特殊形式的指令。这条指令在从存储器中装载 PC 的同时（PC 是最后恢复的），CPSR 也得到恢复。这里使用的堆栈指针 SP（R13）是属于异常模式的寄存器，每个异常模式有自己的堆栈指针。这个堆栈指针应必须在系统启动时初始化。

3.9.3 进入异常

当处理异常时，ARM7TDMI 内核会：

1. 在适当的 LR 中保存下一条指令的地址。当异常入口来自：
   - ARM 状态，ARM7TDMI 将下一条指令的地址复制到 LR 中（当前 PC+4，或 PC+8，取决于异常的类型）
   - Thumb 状态，ARM7TDMI 将 PC 加偏移值（PC+4 或 PC+8，取决于异常的类型）写入 LR 当进入异常时，异常处理程序不必确定状态。例如在 SWI 情况下，MOVS PC,R14_svc 总是返回到下一条指令，而不管 SWI 是在 ARM 还是在 Thumb 状态下执行。
2. 将 CPSR 复制到适当的 SPSR。
3. 根据异常将 CPSR 模式强制设为某一值。
4. 强制 PC 从相关的异常向量处取指。

ARM7TDMI 内核在中断异常时置位中断禁止标志，这样可防止不受控制的异常嵌套。

注：异常总是在 ARM 状态中进行处理。当处理器处于 Thumb 状态时发生了异常，在异

常向量地址装入 PC 时，会自动切换到 ARM 状态。

3.9.4 退出异常

当异常结束时，异常处理程序必须：

1. 将 LR 中的值减去偏移量后移入 PC。偏移量根据异常的类型而有所不同，见表 1.5。
2. 将 SPSR 的值复制回 CPSR。
3. 清零在入口置位的中断禁止标志。

注：恢复 CPSR 的动作会将 T, F 和 I 位自动恢复为异常发生前的值。

3.9.5 快速中断请求

快速中断请求(FIQ)异常支持数据转移或通道处理。在 ARM 状态中，快中断模式有 8 个专用的寄存器可用来满足寄存器保护的需要（这是上下文切换的最小开销）。

将 nFIQ 信号拉低可实现外部产生 FIQ（在具体的芯片中，nFIQ 由片内外设拉低，nFIQ 是内核的一个信号，对用户不可见）。

不管异常入口是来自 ARM 状态还是 Thumb 状态，FIQ 处理程序都会通过执行下面的指令从中断返回：

SUBS PC,R14_fiq,#4 （即在快中断模式执行 SUBS PC,R14,#4 指令）

在一个特权模式中，可通过置位 CPSR 中的 F 标志来禁止 FIQ 异常。当 F 标志清零时，ARM7TDMI 在每条指令结束时检测 FIQ 同步器输出端的低电平。

3.9.6 中断请求

中断请求（IRQ）异常是一个由 nIRQ 输入端的低电平所产生的正常中断（在具体的芯片中，nIRQ 由片内外设拉低，nIRQ 是内核的一个信号，对用户不可见）。IRQ 的优先级低于 FIQ。对于 FIQ 序列它是被屏蔽的。任何时候在一个特权模式下，都可通过置位 CPSR 中的 I 位来禁止 IRQ,。

不管异常入口是来自 ARM 状态还是 Thumb 状态，IRQ 处理程序都会通过执行下面的指令从中断返回：

SUBS PC,R14_irq,#4 （即在中断模式执行 SUBS PC,R14,#4 指令）

3.9.7 中止

中止表示当前对存储器访问不能被完成。这是通过外部 ABORT 输入指示的（在具体的芯片中，ABORT 信号由片内存储器管理部件控制，ABORT 是内核的一个信号，对用户不可见）。不管异常入口是来自 ARM 状态还是 Thumb 状态，中止异常处理程序都会通过执行下面的指令从中断返回：

SUBS PC,R14_fiq,#4 （即在中止模式执行 SUBS PC,R14,#4 指令）

处理器在存储器访问周期结束时检测中止异常。

有两种类型的中止：

- 预取中止 发生在指令预取过程中，
- 数据中止 发生在对数据访问时。

预取中止

当发生预取中止时，ARM7TDMI 内核将预取的指令标记为无效，但在指令到达流水线的执行阶段时才进入异常。如果指令在流水线中因为发生分支而没有被执行，中止将不会发生。

在处理中止的原因之后，不管处于哪种处理器操作状态，处理程序都会执行下面的指令：

SUBS PC,R14_abt,#4 （即在中止模式执行 SUBS PC,R14,#4 指令）

这个动作恢复了 PC 和 CPSR 并重试被中止的指令。

数据中止

当发生数据中止时，根据指令的类型产生不同的动作：

- 数据转移指令（LDR,STR）回写到被修改的基址寄存器。中止处理程序必须注意这一点。
- 交换指令（SWP）中止好像没有被执行过一样（中止必须发生在 SWP 指令进行读访问时）。
- 块数据转移指令（LDM,STM）完成。 当回写被设置时，基址寄存器被更新。在指示出现中止后，ARM7TDMI 内核防止所有寄存器被覆盖。这意味着 ARM7TDMI 内核总是会保护被中止的 LDM 指令中的 R15（总是最后一个被转移的寄存器）。

中止的机制使指令分页的虚拟存储器系统能够被实现。在这样一个系统中，处理器允许产生仲裁地址。当某一地址的数据无法访问时，存储器管理单元（MMU）通知产生了中止。中止处理程序必须找出中止的原因，使请求的数据可以被访问并重新执行被中止的指令。应用程序不必知道可用存储器的数量，也不必知道它的被中止时所处的状态。

在修复产生中止的原因后，不管处于哪种处理器操作状态，处理程序都必须执行下面的返回指令：

```
SUBS    PC,R14_abt,#8          （即在中止模式执行 SUBS  PC,R14,#8 指令）
```

这个动作恢复了 PC 和 CPSR 并重试被中止的指令。

3.9.8 软件中断指令

软件中断(SWI)用于进入管理模式，通常用于请求一个特定的管理函数。SWI 处理程序通过执行下面的指令返回：

```
MOVS    PC,R14_svc             （即在管理模式执行 MOVS  PC,R14 指令）
```

这个动作恢复了 PC 和 CPSR 并返回到 SWI 之后的指令。SWI 处理程序读取操作码以提取 SWI 函数编号。

3.9.9 未定义的指令

当 ARM7TDMI 处理器遇到一条自己和系统内任何协处理器都无法处理的指令时，ARM7TDMI 内核执行未定义指令陷阱。软件可使用这一机制通过仿真未定义的协处理器指令来扩展 ARM 指令集。

注：ARM7TDMI 处理器完全遵循 ARM 结构 v4T，可以捕获所有分类未被定义的指令位格式。

在防止失败的指令后，捕获处理器执行下面的指令：

```
MOVS    PC,R14_und             （即在未定义模式执行 MOVS  PC,R14 指令）
```

这个动作恢复了 PC 和 CPSR 并返回到未定义指令之后的指令。

关于未定义指令更详细的信息请参考参考文献[2]。

3.9.10 异常向量

表 3.6 所示位异常向量地址。在表中，I 和 F 表示先前的值。

表 3.6 异常向量

地址	异常	进入时的模式	进入时 I 的状态	进入时 F 的状态
0x00000000	复位	管理	禁止	禁止
0x00000004	未定义指令	未定义	I	F
0x00000008	软件中断	管理	禁止	F

接上表

地址	异常	进入时的模式	进入时 I 的状态	进入时 F 的状态
0x0000000C	中止（预取）	中止	I	F
0x00000010	中止（数据）	中止	I	F
0x00000014	保留	保留	—	—
0x00000018	IRQ	中断	禁止	F
0x0000001C	FIQ	快中断	禁止	禁止

3.9.11 异常优先级

当多个异常同时发生时，一个固定的优先级系统决定它们被处理的顺序：

1. 复位（最高优先级）
2. 数据中止
3. FIQ
4. IRQ
5. 预取中止
6. 未定义指令
7. SWI（最低优先级）

有些异常不能一起发生：

- 未定义的指令和 SWI 异常互斥。它们分别对应于当前指令的一个特定（非重叠）译码。
- 当 FIQ 使能，并且在发生 FIQ 的同时产生了一个数据中止，ARM7TDMI 内核进入数据中止处理程序，然后立即转到 FIQ 向量。从 FIQ 的正常返回使数据中止处理程序恢复执行。数据中止的优先级必须高于 FIQ 以确保数据转移错误不会被漏过。必须将异常入口的时间增加到系统中最坏情况下 FIQ 的延迟时间。

3.10 中断延迟

3.10.1 最大中断延迟

当 FIQ 使能时，最坏情况下 FIQ 的延迟时间包含：

- Tsyncmax，请求通过同步器的最长时间。Tsyncmax 为 2 个处理器周期（由内核决定）。
- Tldm，最长的指令执行需要的时间（最长的指令是装载包括 PC 在内所有寄存器的 LDM 指令）。Tldm 在零等待状态系统中的执行时间为 20 个周期。**注意，是在零等待状态系统中。**一般的基于 ARM7 核的芯片的存储器系统比内核速度慢，造成其不是零等待的。
- Texc，数据中止入口的时间。Texc 为 3 个周期（由内核决定）。
- Tfiq，FIQ 入口的时间。Tfiq 为 2 个周期（由内核决定）。

因此总的延迟时间为 27 个周期，在系统使用 40MHz 处理器时钟时，略微小于 0.7 微秒。在此时间结束后，ARM7TDMI 执行位于 0x1c 处的指令。

最大的 IRQ 延迟时间与之相似，但必须考虑到这样一个事实，即有更高优先级的 FIQ，当 FIQ 和 IRQ 同时申请时，IRQ 要延迟到 FIQ 处理程序允许 IRQ 中断时才处理（可能需要对中断控制器进行相应的操作）。IRQ 延迟时间也要相应增加。

3.10.2 最小中断延迟

FIQ 或 IRQ 的最小中断延迟是请求通过同步器的时间 Tsyncmin 加上 Tfiq（共 4 个处理器周期）。

3.11 复位

当 nRESET 信号被拉低时（一般外部复位引脚电平的变化和芯片的其它复位源会改变这个内核信号），ARM7TDMI 处理器放弃正在执行的指令。

当 nRESET 信号再次变为高电平时，ARM 处理器执行下列操作：

1. 强制 M[4:0]变为 b10011（管理模式）
2. 置位 CPSR 中的 I 和 F 位
3. 清零 CPSR 中的 T 位
4. 强制 PC 从地址 0x00 开始对下一条指令进行取指。
5. 返回到 ARM 状态并恢复执行

在复位后，除 PC 和 CPSR 之外的所有寄存器的值都不确定。

3.12 存储器及存储器映射 I/O

3.12.1 简介

ARM7TDMI 处理器采用冯·诺依曼（Von Neumann）结构，指令和数据共用一条 32 位数据总线。只有装载、保存和交换指令可访问存储器中的数据。

ARM7 的规范仅定义了处理器核与存储系统之间的信号及时序（局部总线），而现实的芯片一般在外部总线与处理器核的局部总线之间有一个存储器管理部件将局部总线的信号和时序转换为现实的外部总线信号和时序。因此，外部总线的信号和时序与具体的芯片相关，不是 ARM7 的标准。具体到某个芯片的外部存储系统的设计需要参考其芯片的数据手册或使用手册等资料。

ARM7TDMI 处理器将存储器看作是一个从 0 开始的线性递增的字节集合：

- 字节 0 到 3 保存第 1 个存储的字
- 字节 4 到 7 保存第 2 个存储的字
- 字节 8 到 11 保存第 3 个存储的字

ARM7TDMI 处理器可以将存储器中的字以下列格式存储：

- 大端（Big-endian）格式
- 小端（Little-endian）格式

3.12.2 地址空间

ARM 结构使用单个平面的 2^{32} 个 8 位字节地址空间。字节地址按照无符号数排列，从 0 到 $2^{32}-1$。

地址空间可以看作是包含 2^{30} 个 32 位字，地址以字为单位进行分配。也就是将地址除以 4。地址为 A 的字包含 4 个字节，地址分别为 A, A+1, A+2 和 A+3。

在 ARM 结构 v4 及以上版本中（ARM7TDMI 基于 v4 版本），地址空间还可被看作包含 2^{31} 个 16 位半字。地址按照半字进行分配。地址为 A 的半字包含 2 个字节，地址分别为 A 和 A+1。

地址计算通常通过普通的整数指令来实现。这意味着如果地址向上或向下溢出地址空间，通常会发生翻转。也就是说计算的结果以 2^{32} 为模。但是如果地址空间在将来进行扩展，

为了降低不兼容性，程序不应依赖于该特性进行编写。如果地址的计算没有发生翻转，那么结果仍然位于范围 0~2^{31}-1 内。

大多数指令通过指令所指定的偏移量与 PC 值相加并将结果写入 PC 来计算目标地址。如果下面的计算：

(当前指令的地址) + 8 + 偏移量

溢出地址空间，那么该指令依赖于地址的翻转，因此在技术上是不可预测的。因此穿过地址 0xFFFFFFFF 的向前转移和穿过地址 0x00000000 的向后转移都不应使用。

另外，正常连续执行的指令实际上是通过计算：

(当前指令的地址) + 4

来确定下一条要执行的指令。如果该计算溢出了地址空间的顶端，结果同样不可预测。换句话说，程序不应信任在地址 0xFFFFFFFC 处的指令之后连续执行的位于地址 0x00000000 的指令。

注意：上述原则不只适用于执行的指令，还包括指令条件代码检测失败的指令。大多数 ARM 在当前执行的指令之前执行预取指令。如果预取操作溢出了地址空间的顶端，则不会产生执行动作并导致不可预测的结果，除非预取的指令实际上已经执行。

LDR, LDM, STR 和 STM 指令在增加的地址空间访问一连串的字，每次装载或保存，存储器地址都会加 4。如果计算溢出了地址空间的顶端，结果是不可预测的。换句话说，程序在使用这些指令时不应使其溢出。

3.12.3 存储器格式

地址空间的规则要求字地址 A：

- 位于地址 A 的字包含的字节位于地址 A, A+1, A+2 和 A+3；
- 位于地址 A 的半字包含的字节位于地址 A 和 A+1；
- 位于地址 A+2 的半字包含的字节位于地址 A+2 和 A+3；
- 位于地址 A 的字包含的半字位于地址 A 和 A+2；

但是这样并不能完全定义字、半字和字节之间的映射。

存储器系统使用下列两种映射机制中的一种。

- 对于小端（little-endian）存储器系统：

在小端格式中，一个字当中最低地址的字节被看作是最低位字节，最高地址字节被看作是最高位字节。因此存储器系统字节 0 连接到数据线 7～0。如图 3.7 所示。

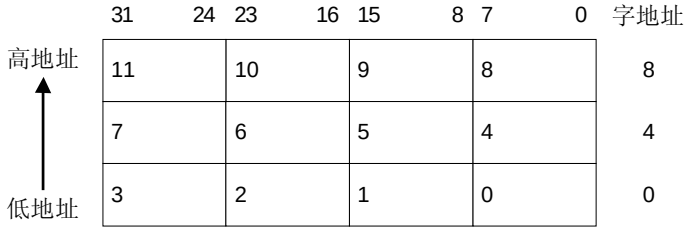


图 3.7 字内字节的小端地址

- 大端（big-endian）存储器系统

在大端格式中，ARM7TDMI 处理器将最高位字节保存在最低地址字节，最低位字节保存在最高地址字节。因此存储器系统字节 0 连接到数据线 31～24。如图 3.8 所示。

一个具体的基于 ARM 芯片可能只支持小端存储器系统，也可能只支持大端存储器系统，还可能两者都支持。

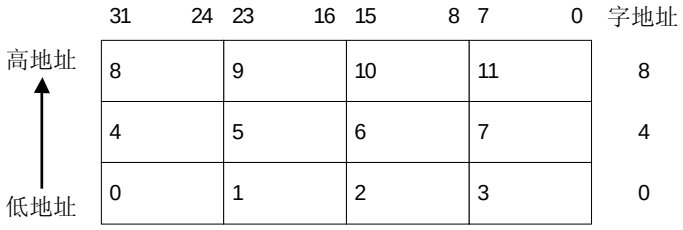


图 3.8 字内字节的大端地址

ARM 指令集不包含任何直接选择大小端的指令。但是一个同时支持大小端的基于 ARM 芯片可以在硬件上配置（一般使用芯片的引脚来配置）来匹配存储器系统所使用的规则。如果芯片有一个标准系统控制协处理器，系统控制协处理器的寄存器 1 的 bit7 可用于改变配置输入。

如果一个基于 ARM 芯片将存储器系统配置为其中一种存储器格式（如小端），而实际连接的存储器系统配置为相反的格式（如大端），那么只有以字为单位的指令取指、数据装载和数据保存能够可靠实现。其它的存储器访问将出现不可预期的结果。

当标准系统控制协处理器连接到支持大小端的 ARM 处理器时，协处理器寄存器 1 的 bit7 在复位时清零。这表示 ARM 处理器在复位后立即配置为小端存储器系统。如果它连接到一个大端存储器系统，复位处理程序所要尽早做的事情之一就是切换到大端存储器系统，并必须在任何可能的字节或半字数据访问发生或 Thumb 指令执行之前执行。

注意：存储器格式的规则意味着字的装载和保存并不受配置的大小端的影响。因此不可能通过保存一个字，改变存储器格式，然后重装已保存的字使该字当中字节的顺序翻转。

一般来说，改变 ARM 处理器配置的存储器格式，使其不同于连接的存储器系统并没有什么用处，因为这样做的结果并不会产生一个额外的结构定义的操作。因此通常只在复位时改变存储器格式的配置使其匹配存储器系统的存储器格式。

3.12.4 未对齐的存储器访问

ARM 结构通常期望所有的存储器访问都合理的对齐。具体来说就是字访问的地址通常是字对齐的，而半字访问使用的地址是半字对齐的。不按这种方式对齐的存储器访问称为非对齐的存储器访问。

非对齐的指令取指

如果在 ARM 状态下将一个非字对齐的地址写入 R15，结果通常不可预测。如果在 Thumb 状态下将一个非半字对齐的地址写入 R15，地址位 bit[0]通常被忽略（见 3.7 节和第 4 章 各个具体指令的详细描述）。结果在 ARM 状态下有效代码从 R15 读出值的 bit[1:0]为 0，而在 Thumb 状态下读出的 R15 值的 bit0 为 0。

当规定忽略这些位时，ARM 实现不要求在指令取指时将这些位清零。可以将写入 R15 的值不加改变地发送到存储器，并在 ARM 或 Thumb 指令取指时请求系统忽略地址位 bit[1:0]或 bit[0]。

非对齐的数据访问

产生非对齐访问的装载/保存指令会出现下列定义的动作之一：

- 不可预测
- 忽略造成访问不对齐的低地址位。这意味着在半字访问时使用公式（地址 AND 0xFFFFFFFE），而在字访问时使用公式（地址 AND 0xFFFFFFFC）。
- 对存储器访问本身忽略造成访问不对齐的低地址位，然后使用这些低地址位控制装载数据的循环（该动作只适用于 LDR 和 SWP 指令）。

这 3 个选项中的哪一个适用于装载/保存指令取决于具体的指令（参考第 4 章 ）。

在将地址发送到存储器时，ARM 实现不要求将造成不对齐的低地址位清零。可以将装载/保存指令计算出的地址不加改变地发送到存储器，并在半字访问或字访问时请求存储器系统忽略地址位 bit[0]或 bit[1:0]。

3.12.5 指令的预取和自修改代码

许多 ARM 实现在前一条指令的执行尚未完成时将指令从存储器中取出。这个动作称为指令的预取。指令的预取并不是实际执行指令。指令后来没有被执行有两种典型的情况：

- 当发生异常时，当前指令执行完毕，所有预取的指令都被丢弃，指令的执行从异常向量开始。
- 当发生跳转时，预取的在分支指令后的指令将被丢弃。

ARM 实现可以自由选择预取指令比当前执行点提前多少（即半导体厂商在设计具体的芯片时可以自由选择预取指令比当前执行点提前多少），甚至可以动态改变预取指令的数目。最初的 ARM 实现在当前执行的指令之前预取两条指令，不过现在可以选择多于或少于两条指令。

注意：当指令读取 PC 时，它得到的指令地址比它自身地址落后了两条指令：

- 对于 ARM 指令，得到的地址是它自身地址+8；
- 对于 Thumb 指令，得到的地址是它自身地址+4。

最初的 ARM 实现曾经在 PC 读取的两指令偏移量和两指令预取之间存在关联。但这一关联不是结构上的。一个预取不同数目指令的 ARM 实现仍能保证读取 PC 所得到的地址比它自身地址落后两条指令。

和自由选择多少条预取指令一样，ARM 实现可选择沿着哪条可能的执行路径进行预取。例如，在一条分支指令之后，它可选择预取分支指令之后的指令或者是转移目标地址的指令。这称为*转移预测*。

所有形式的指令预取都有一个潜在的问题，即存储器中的指令可能在它被预取之后，被执行之前发生改变。如果发生这种情况，对存储器中的指令进行修改通常并不妨碍已取指的指令备份执行完毕。

例如，在下面的代码序列中，STR 指令使用 ADD 指令的备份取代了它后面的 SUB 指令：

```
    LDR     r0, AddInstr
    STR     r0, NextInstr
NextInstr
    SUB     r1, r1, #1
    .
    .
    .
AddInstr
    ADD r1, r1, #1
```

但是当代码第一次执行时，STR 指令之后执行的指令通常是 SUB 指令，因为 SUB 指令在存储器中的指令发生改变之前已经被预取了。ADD 指令不会被执行，除非第二次执行该代码序列。

其实，处理器不能保证按照上面所述的方式执行，因为：

- 当代码第一次执行时，在 STR 指令之后有可能立即产生一个中断，如果这样，已经预取的 SUB 指令将被丢弃。当中断处理程序返回时，位于 NextInstr 处的指令被再次预取，而这次则执行 ADD 指令。因此虽然 SUB 指令通常最有可能被执行，但也有可能执行 ADD 指令。

- 如果指令被再次执行，ARM 处理器或存储器系统允许保持预取指令的备份并使用这些备份而不是重新预取。如果发生这种情况，在代码序列按照第二及以下可能性执行时，SUB 指令可被执行。

发生这种情况的主要原因是存储器系统包含独立的指令和数据缓存。但是也存在其它可能性。例如，一些分支预测的硬件保存了分支后的指令。

总之，应当尽可能避免使用涉及自修改代码的编程技术。

指令存储器屏障（IMB）

在许多系统中，几乎不可能完全避免自修改代码的使用。例如，任何一个允许将程序装入存储器然后执行的系统都使用自修改代码。

因此每个 ARM 实现（可以理解为具体芯片）都定义了一系列的操作使自修改代码序列可以可靠地执行。这一串代码称为指令存储器屏障（IMB），它通常同时取决于 ARM 处理器的实现和存储器系统的实现（可以理解为具体的芯片）。

IMB 序列必须在新的指令已经保存到存储器之后而尚未执行时执行。例如，在程序被装载之后并且在转移到它的入口之前。任何不以这种方式使用 IMB 的自修改代码序列都可能会执行不确定的动作。

根据 IMB 所执行的确定的操作顺序取决于 ARM 和存储器系统的实现（可以理解为具体的芯片）。建议在软件设计时使 IMB 序列作为一个调用程序来替换与系统相关的模块，而不是直接插入到需要的地方。这样易于移植到其它 ARM 处理器和存储器系统。

另外在许多实现当中，IMB 序列包含了只能在特权模式下使用的操作，例如标准系统控制协处理器提供的缓存清零和无效操作。为了允许用户模式程序使用 IMB 序列，推荐将其作为一个操作系统调用程序，由 SWI 指令调用。

在 SWI 指令使用 24 位立即数的系统中指定所要求的系统服务，推荐通过下面的指令来请求 IMB 序列：

```
SWI   0xF00000
```

这是一个无参调用，不返回结果，应当使用与带原型的 C 函数调用相同的调用约定：

```
void   IMB(void) ;
```

区别在于使用 SWI 指令调用而不是 BL 指令。

有些实现可对已保存的新指令使用地址范围的知识来减少 IMB 执行的时间。因此还推荐执行第二个操作系统调用程序，该调用程序只根据指定的地址范围执行 IMB。在 SWI 指令使用 24 位立即数的系统中指定所要求的系统服务，推荐通过下面的指令来请求：

```
SWI   0xF00001
```

应当使用与带原型的 C 函数调用相似的调用约定：

```
void IMB_Range(unsigned long start_addr, unsigned long end_addr);
```

此处，地址范围从 start_addr(包含)到 end_addr(不包含)。

注意：

- 当使用标准的 ARM 过程调用标准时，start_addr 在 R0 中传递，而 end_addr 则在 R1 中传递。
- 对于某些 ARM 实现来说，即使使用小地址范围，IMB 执行的时间也可能非常长（数千个时钟周期）。对于自修改代码的小规模使用，这样很可能使性能上受到较大损失。因此建议自修改代码只用于不可避免和/或有足够的执行时间裕量的情况。

IMB 的其它用途

有些存储器系统允许虚拟－物理的地址映射，其中物理存储器位置对应于 ARM 处理器产生的地址，它可以被改变。如果该地址映射在指令预取之后，执行之前被更改，指令的地址会受到地址映射更改的影响，那么将执行错误的指令。

这非常类似于在指令被预取但尚未执行时在指令地址发生保存的情况。在这两种情况下，由于有一个值保存到该地址或者该地址关联到一个不同的物理存储器位置，保存在存储器地址的指令被改变。当虚拟－物理地址映射发生改变时可以使用相同的解决办法。IMB 序列必须在虚拟－物理地址映射改变之后，并在指令执行之前执行。

另一种相似的情况是在指令预取和指令执行之间这段时间内发生存储器访问许可的变更。如果在指令预取时不允许访问，而在指令执行时允许访问，可能会发生不期望的预取中止异常。在相反的情况，即指令预取时允许访问而在指令执行时禁止访问，系统中可能存在安全漏洞。

存储器访问许可的改变通常是因为将新的访问许可设定写入存储器或是因为存储器系统对于用户模式、特权模式以及发生的下列情况支持不同的访问许可：

- 在用户模式下产生一次异常，导致处理器切换到特权模式。
- 特权代码将模式切换到用户模式

所有 ARM 的实现都确保了下列事件不会导致在错误的访问许可下预取之后指令的执行：

- 在用户模式下产生一次异常
- 当异常返回时使特权模式切换到用户模式的指令的执行。这些指令有一个副作用，就是将当前模式的 SPSR 内容复制到 CPSR，它们是：

  － 目标寄存器为 R15 的数据处理指令 ADCS, ADDS, ANDS, BICS, EORS, MOVS, MVNS, ORRS, RSBS, RSCS, SBCS 和 SUBS（不过通常只有 MOVS 和 SUBS 指令用于异常返回）。

  － 3.9.2 节（程序清单 3.3）介绍的 LDM 指令的特殊形式。

其余的情况则不能保证在错误的访问许可下预取之后指令不会被执行。这些情况是：

- 向存储器系统明确地写入新的访问许可设定。
- 通过 MSR 指令从特权模式切换到用户模式。

在这些情况下，在访问许可发生改变后需要立即执行 IMB 序列，并且在访问许可改变之后，指令存储器屏障被访问许可改变之前没有执行任何指令。

但是在这些情况下通常可以避免整个 IMB 的开销。特别地，与任何特定地址相关联的指令字并未改变，因此通常可以避免清空缓存。因此一个实现可以定义受限版的 IMB 序列在这些情况下使用。

3.12.6 存储器映射的 I/O

执行 ARM 系统 I/O 功能的标准方法是使用存储器映射的 I/O。装载或保存 I/O 值时，使用提供给 I/O 功能的特殊存储器地址。通常，从存储器映射的 I/O 地址装载用于输入，而保存到存储器映射的 I/O 地址则用于输出。装载和保存都可用于执行控制功能，用于取代它们正常的输入或输出功能。

存储器映射的 I/O 位置的动作通常不同于正常的存储器位置的动作。例如，正常存储器位置的两次连续装载每次都会返回相同的值，除非中间插入了保存的操作。对于存储器映射的 I/O 位置，第二次装载返回的值可以不同于第一次返回的值。因为第一次装载的副作用（例如从缓冲区移走已装载的值）或是因为插入另一个存储器映射 I/O 位置的装载和保存的副作用。

这些区别主要影响高速缓存的使用和存储器系统的写缓冲区，具体信息请参考相关资料。一般来说，存储器映射的 I/O 位置通常标示为无高速缓存和无缓冲区，以避免对它们进行访问的次数、类型、顺序或时序发生改变。

从存储器映射的 I/O 取指

在 1.11.5 节中讲到，不同 ARM 的实现（可以理解为不同的芯片）在存储器指令取指时

会有相当大的区别。因此强烈建议存储器映射的 I/O 位置只用于数据的装载和保存，不用于指令取指。任何依赖于从存储器映射 I/O 位置取指的系统设计都可能难于移植到将来的 ARM 实现。

对存储器映射 I/O 的数据访问

一个指令序列在执行时会在不同的点访问数据存储器，产生装载和保存访问的时序。如果这些装载和保存访问的是正常的存储器位置，那么在它们在访问相同的存储器位置时只是执行交互操作。结果，对不同存储器位置的保存和装载可以按照不同于指令的顺序执行，但不会改变最终的结果。这种改变存储器访问顺序的自由可被存储器系统用来提高性能（例如通过使用高速缓存和写缓冲区）。

此外，对同一存储器位置的访问还拥有其它可用于提升性能的特性，其中包括：

- 从相同的位置连续加载（没有插入存储）产生相同的结果。
- 从一个位置执行加载操作，将返回最后保存到该位置的值。
- 对某个数据规格的多次访问有时可合并成单个的更大规模的访问。例如，分别存储一个字所包含的两个半字可合并成单个字的存储。

但是如果存储器字、半字或字节访问的对象是存储器映射的 I/O 位置。一次访问会产生副作用，使后续访问改变成一个不同的地址。如果是这样，那么不同时间顺序的访问将会使代码序列产生不同的最终结果。因此访问存储器映射的 I/O 位置时不能进行优化，它们的时间顺序绝对不能改变。

对于存储器映射的 I/O，另外还有很重要的一点。那就是每次存储器访问的数据规格都不会改变。例如，在访问存储器映射的 I/O 时，一个指定从 4 个连续字节地址读出数据的代码序列决不能合并成单个字的读取，否则会使代码序列的最终执行结果不同于期望的结果。相似地，将字的访问分解成多个字节的访问可能会导致存储器映射的 I/O 设备无法按照预期进行操作。

每个 ARM 实现（可以理解为具体的基于 ARM 的芯片）都提供一套机制来保证在数据存储器访问时不会改变访问的次数、数据的规格或时间顺序。该机制包含了实现定义的要求，在存储器访问时保护访问的次数、数据规格和时间顺序。如果在访问存储器映射的 I/O 时不符合这些要求，可能会发生不可预期的动作。

典型的要求包括：

- 限制存储器映射 I/O 位置的存储器属性。例如，在标准存储器系统结构中，存储器位置必须是无高速缓存和无缓冲区的。
- 限制访问存储器映射 I/O 位置的规格或对齐方式。例如，如果一个 ARM 实现带有 16 位外部数据总线，它可以禁止对存储器映射的 I/O 使用 32 位访问，因为 32 位访问无法在单个总线周期内执行。
- 要求额外的外部硬件。例如，带 16 位外部数据总线的 ARM 实现可以允许对存储器映射的 I/O 使用 32 位访问，但要求外部硬件将两个 16 位总线访问合并成对 I/O 设备的单个 32 位访问。

如果数据存储器访问序列包含一些符合要求的访问和一些不符合要求的访问，那么：

- 符合要求的访问其数据规格和数目都被保护，没有互相合并或，也没有与不符合要求的访问以任何方式合并。不符合要求的访问可以互相合并。
- 符合要求的访问彼此的时间顺序被保护，但它们相对于那些不符合要求访问的时间顺序不能保证。

LDM 和 STM 指令的时间顺序

LDM 指令对存储器中的连续字执行连续的加载。STM 指令将多个数据存储到存储器的连续字单元中。上面所描述的访问存储器映射 I/O 的规则适用于这些指令当中连续字的访

问，和在一串单个存储器访问指令序列中应用的方式一样。

LDM 或 STM 指令所执行的存储器访问序列的时间顺序只在有限的环境下由结构所定义。这些规则包括：

- 如果指令中所列寄存器包含 PC，存储器访问的序列并未定义（意味着这样的 LDM 和 STM 指令不适合访问存储器映射的 I/O）。
- 如果指令中所列寄存器不包含 PC，存储器访问序列的时间顺序按照存储器地址排列，从最低地址开始，到最高地址结束。（该顺序与加载和存储寄存器列表里的寄存器升序排列相同。）
- 如果所有由 LDM 或 STM 产生的存储器访问都符合实现定义的对待存储器映射 I/O 位置的要求，那么它们的数目、数据规格和时间顺序都被保护。
- 如果有些由 LDM 或 STM 产生的存储器访问符合实现定义的对待存储器映射 I/O 位置的要求，而有些不符合，那么它们的数目、数据规格和时间顺序不能确保被保护。ARM 处理器和存储器系统甚至不必保护符合该要求访问的时间顺序。这是正常规则的一个例外，它适用于有些访问符合要求而有些访问不符合要求的情况。
  例如，使用标准存储器系统时，如果 LDM 或 STM 指令穿过了存储器带高速缓存的区域和无高速缓存、无缓冲区的区域之间的边界，存储器访问的时间顺序将无法确保得到保护。因此这样的 LDM 和 STM 指令不适用于存储器映射的 I/O。

3.13 寻址方式简介

寻址方式是处理器执行指令时寻找真实操作数地址的方式。ARM 处理器支持 9 种基本寻址方式，下面一一介绍。

寄存器寻址：所需要的操作数在寄存器中，即寄存器的内容为操作数。

立即寻址：操作数包含在指令当中，读取了指令就读取了操作数。

寄存器移位寻址：ARM 指令集特有的寻址方式。操作数在寄存器中，但寄存器中保存的数并不是操作数本身。真实操作数由寄存器移动一定的位数得来（即乘以 2^n 或除以 2^n，n 为左移或右移的位数）。

寄存器间接寻址：操作数在内存中，但指令中并没有包含操作数在内存中的地址，而是指定一个寄存器，这个寄存器的内容为操作数在内存中的地址（即用寄存器作为指针访问内存）。

基址寻址：与寄存器间接寻址类似，但寄存器保存的不是操作数在内存中的地址。操作数在内存中的地址由寄存器的值加上指令指定的一个偏移得到。

多寄存器寻址：一次可以把内存中的多个值传送到多个寄存器或是把多个寄存器的值一次传递到内存中。这种寻址允许一条指令传送 16 个寄存器的任何子集（或是所有 16 个寄存器）。

堆栈寻址：多寄存器寻址的特殊形式，是按照堆栈的约束条件工作的成对使用的多寄存器寻址。ARM 处理器支持所有类型的堆栈。

块拷贝寻址：多寄存器寻址的特殊形式，是（按一定的规则）成对使用的多寄存器寻址，一般用于内存拷贝。

相对寻址：基址寻址的特殊形式：这个基址必须由程序计数器 PC(R15)提供。这样，操作数就在指令本身所在的内存地址+8（参考 3.7 关于程序计数器 R15 的描述）为基址了，而指令中指出的偏移量实质就是操作数与这条指令的相对位置(应当还要加上 8 才是真正的偏移，但汇编程序会处理这些差异)。

3.14 ARM7 指令集简介

3.14.1 简介

ARM7TDMI 处理器有两个指令集：

- 32 位 ARM 指令集
- 16 位 Thumb 指令集

每种指令集有自己的优缺点和使用范围。

3.14.2 ARM 指令集

ARM 指令集可分为 5 大类指令：

- 分支指令
- 数据处理指令
- 加载和存储指令
- 协处理器指令
- 杂项指令

大多数的数据处理指令和一种类型的协处理器指令可以根据它们的结果使 CPSR 寄存器当中的 4 个条件代码标志（N、Z、C 和 V）更新。注意是“可以”而不是“一定”。当指令带 S 后缀时一般要更新条件代码标志。否则一般不更新。不过有例外的情况。详细情况参考第 4 章 。

几乎所有的 ARM 指令都包含一个 4 位的条件域。如果条件代码标志在指令开始执行时指示条件为真，那么指令正常执行，否则指令什么也不做。14 个可用的条件允许：

- 测试相等或不相等
- 测试不相等调节<, <=, >和>=，包括有符号和无符号运算
- 单独测试每个条件代码标志

条件域的第 16 个值用于那些不允许条件执行的指令。

这个条件域在指令中由指令的条件码后缀指定，例子见程序清单 3.4。详细情况参考第 4 章 。

程序清单 3.4 令的条件执行

```
正常指令(总是执行)：
        B       Lable
相等执行：
        BEQ     Lable
```

(1) 分支指令

标准分支指令除了允许数据处理或加载指令通过写 PC 来改变控制流以外，还提供了一个 24 位有符号偏移，可实现最大 32MB 向前或向后的转移。

转移和连接（BL）选项在跳转后将指令地址保存在 R14（LR）当中。这样通过将 LR 复制到 PC 可实现子程序的返回。

另外有的分支指令可在指令集之间进行切换，此时，分支指令执行完成后处理器继续执行 Thumb 指令集的指令。这样就允许 ARM 代码调用 Thumb 子程序，而 ARM 子程序也可返回到 Thumb 调用程序。Thumb 指令集中相似的指令可实现对应的 Thumb→ARM 的切换。

(2) 数据处理指令

数据处理指令在通用寄存器上执行计算。ARM7TDMI 的数据处理指令分为 3 种类型：

- 算术/逻辑指令
- 比较指令
- 乘法指令

算术/逻辑指令

算术/逻辑指令一共有 12 条，它们使用相同的指令格式。它们最多使用两个源操作数来执行算术或逻辑操作，并将结果写入目标寄存器。也可选择根据结果更新条件代码标志。

两个源操作数为：

- 其中一个一定是寄存器
- 另一个有两种基本形式：

  立即数或是寄存器值，可选择移位。

如果操作数是一个移位寄存器，移位计数可以是一个立即数或另一个寄存器的值。可以指定 4 种移位的类型。每一条算术/逻辑指令都可以执行算术/逻辑和移位操作。这样就可轻松实现各种不同的分支指令。

比较指令

比较指令有 4 条，它们使用与算术/逻辑指令相同的指令格式。比较指令根据两个源操作数执行算术或逻辑操作，但不将结果写入寄存器。它们总是根据结果更新条件代码标志。

比较指令源操作数的格式与算术/逻辑指令相同，包括移位操作的功能。

乘法指令

乘法指令分成两类。这两类指令都将 32 位寄存器值相乘并保存结果：

32 位结果　　正常 在一个寄存器中保存 32 位结果。

64 位结果　　长 在两个独立的寄存器中保存 64 位结果。

乘法指令的这两种类型都可选择执行累加操作。

(3) 加载和存储指令

装载和保存指令包括：

- 装载和保存寄存器
- 装载和保存多个寄存器
- 交换寄存器和存储器内容

加载和存储寄存器

加载寄存器指令可将一个 32 位字、一个 16 位半字或一个 8 位字节从存储器装入寄存器。字节和半字在加载时自动实现零扩展和符号扩展。保存寄存器指令可以将一个 32 位字、一个 16 位半字或一个 8 位字节从寄存器保存到存储器。

加载和存储寄存器指令有 3 种主要的寻址模式，这 3 种模式都使用指令所指定的基址寄存器和偏移量：

- 在偏移寻址模式中，将基址寄存器值加上或减去一个偏移量得到存储器地址。
- 在前变址寻址模式中，存储器地址的构成方式和偏移寻址模式相同，但存储器地址会回写到基址寄存器。
- 在后变址寻址模式中，存储器地址为基址寄存器的值。基址寄存器的值加上或减去偏移量的结果写入基址寄存器。

在每种情况下，偏移量都可以是一个立即数或是一个变址寄存器的值。基于寄存器的偏移量也可使用移位操作来调整。

由于 PC 是一个通用寄存器，可以通过将 32 位值直接装入 PC 跳转到 4GB 存储器空间的任何地址。

加载和存储多个寄存器

加载多个寄存器（LDM）和存储多个寄存器（STM）指令可以对任意数目的通用寄存

器执行块转移。支持下列 4 种寻址模式：

- 前递增
- 后递增
- 前递减
- 后递减

基地址由一个寄存器值指定，它在转移后可选择更新。由于子程序返回地址和 PC 值位于通用寄存器当中，使用 LDM 和 STM 可构成非常高效的子程序入口和出口：

- 子程序入口处的单个 STM 指令可将寄存器内容和返回地址压入堆栈，在处理中更新堆栈指针。
- 子程序出口处的单个 LDM 指令可将寄存器内容从堆栈恢复，将返回地址装入 PC 并更新堆栈指针。

LDM 和 STM 指令还可用于实现非常高效的块复制和相似的数据移动算法。

交换寄存器和存储器内容

交换指令（SWP）执行下列操作：

1. 从寄存器指定的存储器位置装入一个值；
2. 将寄存器内容保存到同一个存储器位置；
3. 将步骤 1 装入的值写入一个寄存器。

如果步骤 2 和 3 指定同一个寄存器，那么存储器和寄存器的内容就实现了互换。

交换指令执行一个特殊的不可分割的总线操作，该操作允许信号量的原子更新并支持 32 位字和 8 位字节信号量。

(4) 协处理器指令

协处理器指令有 3 种类型：

1. 数据处理指令

启动一个协处理器专用的内部操作。

2. 数据转移指令

将数据在协处理器和存储器之间进行转移。转移的地址由 ARM 处理器计算。

3. 寄存器转移指令

允许协处理器值转移到 ARM 寄存器或将 ARM 寄存器值转移到协处理器。

(5) 杂项指令

杂项指令包括状态寄存器转移指令和异常产生指令。

状态寄存器转移指令将 CPSR 或 SPSR 的内容转移到一个通用寄存器，或者反过来将通用寄存器的内容写入 CPSR 或 SPSR 寄存器。写 CPSR 会：

- 设定条件代码标志的值
- 设定中断使能位的值
- 设定处理器模式

有两种类型的指令用于产生特定的异常，但在 ARM7TDMI 仅实现了一种，它就是软件中断指令。

SWI 指令导致产生软件中断异常。它通常用于向操作系统请求调用 OS 定义的服务。SWI 指令导致的处理器进入管理模式（一种特权模式）。这样一个非特权任务就能对特权的功能进行访问，但是只能以 OS 所允许的方式访问。

3.14.3 Thumb 指令集

传统的微处理器结构对于指令和数据有相同的带宽。因此，和 16 位结构相比，32 位结构处理 32 位数据具有更高的性能，并且在寻址更大的地址空间时要有效得多。

16 位结构比 32 位结构具有更高的代码密度，并且超过 32 位结构 50%的性能。Thumb

在 32 位结构上实现了 16 位的指令集，这样可提供：

- 比 16 位结构更高的性能
- 比 32 位结构更高的代码密度

Thumb 指令集不是一个完整的指令集，它仅仅是最通用的 ARM 指令的子集，不能指望处理器只执行 Thumb 指令而不支持 ARM 指令。Thumb 指令长度为 16 位，每条指令都对应一条 32 位 ARM 指令，它对处理器模型有相同的效果。

Thumb 指令使用标准的 ARM 寄存器配置进行操作（Thumb 指令对寄存器访问的限制参考 1.6.3 节），这样 ARM 和 Thumb 状态之间具有极好的互用性。在执行方面，Thumb 具有 32 位内核所有的优点：

- 32 位地址空间
- 32 位寄存器
- 32 位移位器和算术逻辑单元(ALU)
- 32 位存储器传输

Thumb 因此提供了长的分支范围、强大的算术操作和巨大的地址空间。

Thumb 代码仅为 ARM 代码规模的 65%，但其性能却相当于连接到 16 位存储器系统的 ARM 处理器性能的 160%。因此 Thumb 使 ARM7TDMI 处理器非常适用于那些只有有限的存储器带宽并且代码密度很高的嵌入式应用。

16 位 Thumb 和 32 位 ARM 指令集使设计者极大的灵活性，使他们可以根据各自应用的需求，在子程序一级上实现对性能或者代码规模的优化。例如，应用中的快速中断和 DSP 算法可使用完全的 ARM 指令集编写并与使用 Thumb 代码连接。

为了实现 16 位指令长度，Thumb 指令丢弃 ARM 指令集的一些特性：

- 大多数指令是无条件执行的（所有 ARM 指令是条件执行的）
- 许多 Thumb 指令采用 2 地址格式（除 64 位乘法外，ARM 数据处理指令采取 3 地址格式）
- Thumb 指令没有 ARM 指令规则

Thumb 指令集可分为 4 大类指令：

- 分支指令
- 数据处理指令
- 寄存器加载与存储指令
- 异常产生指令

(1) 分支指令

与 ARM 分支指令不同，Thumb 的分支指令 B、BX 和 BL 中的偏移域没有固定的位数，不过应用工程师不必关心它，汇编程序会自动处理。其中指令 B 是 Thumb 指令中唯一条件执行的指令。

转移和连接（BL）选项在跳转后将指令地址保存在 R14（LR）当中。这样通过将 LR 复制到 PC 可实现子程序的返回。

另外有的分支指令可在指令集之间进行切换。这样就允许 Thumb 子程序和 ARM 子程序可以相互调用。

(2) 数据处理指令

这些指令都能够映射到相应的 ARM 数据处理指令（包括乘法指令）。尽管 ARM 指令支持在单条指令中完成一个操作数的移位和 AUL 操作，但 Thumb 指令集将移位操作和 ALU 操作分离为不同的指令。

Thumb 指令对 8 个寄存器的操作的数据处理指令都更新条件码标志（同功能的 ARM 指令仅在带 S 后缀时更新条件码标志位）。除 CMP 指令外，对高 8 个寄存器操作的指令不改

变条件码标志（CMP 指令的用途就是改变条件码标志）。

(3) 加载和存储指令

加载和存储指令包括加载和存储单寄存器和加载和存储多个寄存器两类。

加载和存储单寄存器的指令是从 ARM 的加载和存储单寄存器指令集中精选出来的子集，并且与等价的 ARM 指令有严格相同的语义和完全相同的汇编格式。

Thumb 只有六条加载和存储多个寄存器的指令，分别为：PUSH {reglist}、POP {reglist}、PUSH {reglist，LR}、POP {reglist,PC}、LDMIA Rn,{reglist}和 STMIA Rn,{reglist}。这些指令有很多使用限制，具体情况参考第二章。

(4) 异常产生指令

有两种类型的指令用于产生特定的异常，但在 ARM7TDMI 仅实现了一种，它就是软件中断指令。

SWI 指令导致产生软件中断异常。它通常用于向操作系统请求调用 OS 定义的服务。SWI 指令导致的处理器进入管理模式（一种特权模式）和进入 ARM 状态。这样一个非特权任务就能对特权的功能进行访问，但是只能以 OS 所允许的方式访问。

3.15 协处理器接口

3.15.1 简介

ARM7TDMI 处理器指令集使您可以通过协处理器来实现特殊的附加指令。这些协处理器是与 ARM7TDMI 内核相结合的单独的处理单元。一个典型的协处理器包括：

- 指令流水线
- 指令译码逻辑
- 寄存器分组
- 带独立数据通路的特殊处理逻辑

协处理器和 ARM7TDMI 处理器连接到同一个数据总线，这意味着协处理器可以对指令流中的指令进行译码并执行那些它所支持的指令。每条指令的处理都沿着 ARM7TDMI 处理器流水线和协处理器流水线同时进行。

指令的执行由 ARM7TDMI 内核与协处理器共同实现。

ARM7TDMI 内核：

1. 求出条件代码的值以确定指令是否必须由协处理器执行，然后使用 CPnI（内核与协处理器握手的信号）通知系统中的所有协处理器。
2. 产生指令所要求的地址（包括下一条指令的预取）来填充流水线。
3. 如果出现协处理器不接受的指令，则执行未定义指令陷阱。

协处理器：

1. 对指令进行译码以确定是否接受。
2. 通过 CPA 和 CPB（内核与协处理器握手的信号）指示它是否接受这一指令。
3. 从自身的寄存器组当中取出任何需要的值。
4. 执行指令所要求的操作。

如果协处理器无法执行某条指令，则执行未定义指令陷阱。您可以选择在软件中仿真协处理器功能或设计一个专用的协处理器。

3.15.2 可用的协处理器

一个系统中最多可连接 16 个协处理器，每个协处理器都通过唯一的 ID 号识别。ARM7TDMI 处理器包含两个内部协处理器：

- CP14 通信通道协处理器
- CP15 为 cache 和 MMU 功能提供的系统控制协处理器

因此，您不能将外部协处理器的编号分配为 14 和 15。ARM 还保留了其它的协处理器编号，见表 3.7。

表 3.7 可用的协处理器

协处理器编号	分配
15	系统控制
14	调试控制器
13:8	保留
7:4	可供芯片设计者使用
3:0	保留

3.15.3 关于未定义的指令

ARM7TDMI 处理器执行完全的 ARM 结构 v4T 未定义指令的处理。这意味着 ARM 体系结构参考手册中定义为 UNDEFINED 的任何指令都会使 ARM7TDMI 处理器执行未定义指令陷阱。任何一个不被协处理器接受的指令也会使 ARM7TDMI 处理器执行未定义指令陷阱。

3.16 调试接口简介

ARM7TDMI 处理器的高级调试特性使应用程序、操作系统和硬件的开发变得更加容易。

3.16.1 典型调试系统

ARM7TDMI 处理器构成了调试系统的一个部件，它作为您执行的高级调试与 ARM7TDMI 所支持的低级调试之间的接口。图 3.9 所示为一个典型的调试系统。

一个调试系统通常具有 3 个部分：

调试主机

一台运行调试软件（例如 ARM 的 Windows 版调试器 AXD）的计算机。调试主机使您可以使用设置断点或检查存储器内容这些高级命令。

协议转换器

调试主机发出的高级命令与 ARM7TDMI 处理器 JTAG 接口的低级命令之间的接口。典型地，它通过一个接口（例如增强型并口）与主机相连。

调试目标

ARM7TDMI 处理器具有便于进行底层调试的硬件扩展。这些扩展可使您：

- 暂停程序的执行
- 检查和修改内核的内部状态
- 检查存储器系统的状态
- 执行中止异常，允许实时监控内核
- 恢复程序执行

调试主机和协议转换器与系统有关。

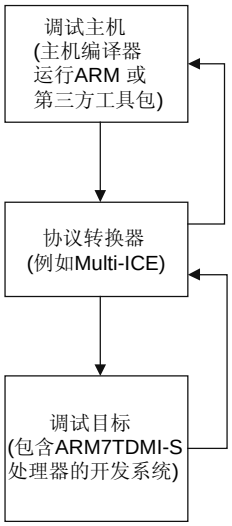


图 3.9 典型的调试系统

3.16.2 调试接口

ARM7TDMI 处理器的调试接口是建立在 IEEE 1149.1-1990 标准、*标准测试访问端口和边界扫描结构*基础之上的。

3.16.3 EmbeddedICE-RT

ARM7TDMI 处理器 EmbeddedICE-RT 模块为 ARM7TDMI 内核提供集成的片内调试支持。EmbeddedICE-RT 通过 ARM7TDMI 处理器 TAP 控制器串行编程。图 3.10 为内核、EmbeddedICE-RT 与 TAP 控制器之间的关系，图中只显示了与 EmbeddedICE-RT 有关的信号。

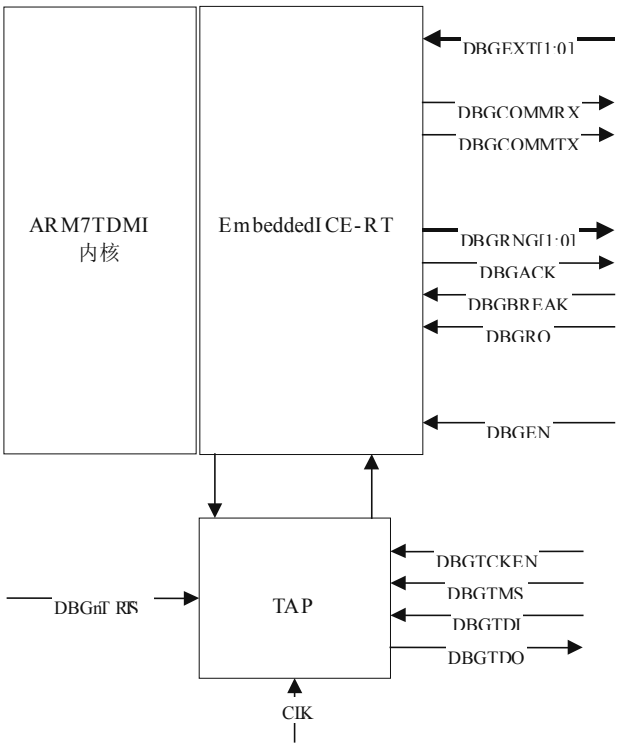


图 3.10 ARM7TDMI 内核、TAP 控制器和 EmbeddedICE-RT 宏单元

EmbeddedICE-RT 逻辑包含下面这些部分：

两个实时观察点单元

可以编程这两个观察点或其中一个使内核暂停指令的执行。当编程到 EmbeddedICE-RT 的值与地址总线、数据总线和不同的控制信号上出现的值相匹配时，指令的执行会被暂停。可以屏蔽其中任何一位使它的值不影响比较。

每个观察点单元都可配置为观察点（监视数据的访问）或断点（监视指令取指）

中止状态寄存器

该寄存器用于识别产生异常中止的原因。

调试通信通道（DCC）

DCC 在目标系统与主机调试程序之间传递信息。

3.16.4 扫描链和 JTAG 接口

在 ARM7TDMI 处理器中有两个扫描链用于实现调试和 EmbeddedICE-RT 编程。JTAG 类型的测试访问端口（TAP）控制器控制扫描链。关于 JTAG 规范的详细信息请参阅 IEEE 标准 1149.1-1990。

3.17 ETM 接口简介

外部嵌入式跟踪宏单元（ETM）连接到 ARM7TDMI 处理器，这样就能够实现对正在执行的处理器进行代码的实时跟踪。

ETM 直接连接到 ARM 内核而不是主 AMBA 系统总线。它将跟踪信息压缩并通过一个窄带跟踪端口输出。外部跟踪端口分析仪在软件调试器的控制下捕获跟踪信息。跟踪端口可以广播指令跟踪信息。指令跟踪(或 PC 跟踪)显示了处理器的执行流程并提供所有已执行指令的列表。指令跟踪被显著压缩为广播分支地址和一套用于指示流水线状态的状态信号。跟踪信息的产生可通过选择触发源进行控制。触发源包括地址比较器、计数器和序列发生器。由于跟踪信息被压缩，软件调试器需要一个执行代码的静态映像。由于这个限制，自修改代码无法被跟踪。

思考与练习

1 基础知识

a) ARM7TDMI 中的 T、D、M、I 的含义是什么？
b) ARM7TDMI 采用几级流水线?使用何种存储器编址方式？
c) ARM 处理器模式和 ARM 处理器状态有何区别？
d) 分别列举 ARM 的处理器模式和状态。
e) PC 和 LR 分别使用哪个寄存器？
f) R13 寄存器的通用功能是什么？
g) CPSR 寄存器中哪些位用来定义处理器状态？
h) ARM 和 Thumb 指令的边界对齐有何不同？
i) 描述一下如何禁止 IRQ 和 FIQ 的中断？

2 存储器格式

定义 R0＝0x12345678，假设使用存储指令将 R0 的值存放在 0x4000 单元中（有关 ARM 指令将在第二章详细介绍）。如果存储器格式为大端格式，请写出在执行加载指令将存储器 0x4000 单元的内容取出存放到 R2 寄存器操作后所得 R2 的值。如果存储器格式改为小端格式，所得的 R2 值又为多少？低地址 0x4000 单元的字节内容分别是多少？

3 处理器异常

请描述一下 ARM7TDMI 的产生异常的条件分别是什么?各种异常会使处理器进入哪种模式?进入异常时内核有何操作，各种异常的返回指令又是什么?

第4章 ARM7TDMI(-S)指令系统

ARM 处理器是基于精简指令集计算机(RISC)原理设计的，指令集和相关译码机制较为简单。ARM7TDMI(-S)具有 32 位 ARM 指令集和 16 位 Thumb 指令集，ARM 指令集效率高，但是代码密度低;而 Thumb 指令集具有较高的代码密度，却仍然保持 ARM 的大多数性能上的优势，它是 ARM 指令集的子集。所有的 ARM 指令都是可以有条件执行的，而 Thumb 指令仅有一条指令具备条件执行功能。ARM 程序和 Thumb 程序可相互调用，相互之间的状态切换开销几乎为零。

说明：本章中的 ARM7TDMI(-S)表示 ARM7TDMI 或 ARM7TDMI-S。

4.1 ARM 处理器寻址方式

寻址方式是根据指令中给出的地址码字段来实现寻找真实操作数地址的方式。ARM 处理具有 9 种基本寻址方式。

1. 寄存器寻址

操作数的值在寄存器中，指令中的地址码字段指出的是寄存器编号，指令执行时直接取出寄存器值来操作。寄存器寻址指令举例如下：

```
MOV     R1,R2          ; 将 R2 的值存入 R1
SUB     R0,R1,R2       ; 将 R1 的值减去 R2 的值，结果保存到 R0
```

2. 立即寻址

立即寻址指令中的操作码字段后面的地址码部分即是操作数本身，也就是说，数据就包含在指令当中，取出指令也就取出了可以立即使用的操作数(这样的数称为立即数)。立即寻址指令举例如下：

```
SUBS    R0,R0,#1       ; R0 减 1，结果放入 R0，并且影响标志位
MOV     R0,#0xFF000 ; 将立即数 0xFF000 装入 R0 寄存器
```

立即数要以“#”号为前缀，表示 16 进制数值时以“0x”表示。

3. 寄存器移位寻址

寄存器移位寻址是 ARM 指令集特有的寻址方式。当第 2 个操作数是寄存器移位方式时，第 2 个寄存器操作数在与第 1 个操作数结合之前，选择进行移位操作。寄存器移位寻址指令举例如下：

```
MOV     R0,R2,LSL #3       ; R2 的值左移 3 位，结果放入 R0，即是 R0=R2×8
ANDS    R1,R1,R2,LSL R3  ; R2 的值左移 R3 位，然后和 R1 相“与”操作，结果放
                           ; 入 R1
```

可采用的移位操作如下：

- LSL： 逻辑左移(Logical Shift Left)，寄存器中字的低端空出的位补 0。
- LSR： 逻辑右移(Logical Shift Right)，寄存器中字的高端空出的位补 0。
- ASR： 算术右移(Arithmetic Shift Right)，移位过程中保持符号位不变，即若源操作数为正数，则字的高端空出的位补 0，否则补 1。
- ROR： 循环右移(Rtate Right)，由字的低端移出的位填入字的高端空出的位。
- RRX： 带扩展的循环右移(Rotate Right eXtended by 1 place)，操作数右移一位，高端空出的位用原 C 标志值填充。

各种移位操作如图 4.1 所示。

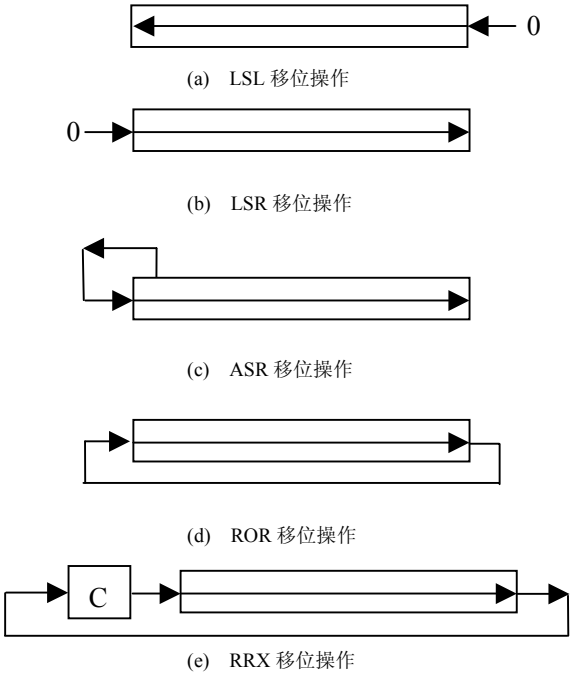


图 4.1 移位操作示意图

4. 寄存器间接寻址

寄存器间接寻址指令中的地址码给出的是一个通用寄存器的编号，所需的操作数保存在寄存器指定地址的存储单元中，即寄存器为操作数的地址指针。寄存器间接寻址指令举例如下：

```
LDR     R1,[R2]          ; 将 R2 指向的存储单元的数据读出，保存在 R1 中
SWP     R1,R1,[R2]       ; 将寄存器 R1 的值和 R2 指定的存储单元的内容交换
```

5. 基址寻址

基址寻址就是将基址寄存器的内容与指令中给出的偏移量相加，形成操作数的有效地址。基址寻址用于访问基址附近的存储单元，常用于查表、数组操作、功能部件寄存器访问等。基址寻址指令举例如下：

```
LDR     R2,[R3,#0x0C]        ; 读取 R3+0x0C 地址上的存储单元的内容，放入 R2
STR     R1,[R0,#-4]!         ; 先 R0=R0-4，然后把 R1 的值寄存到保存到 R0 指定
                             ; 的存储单元
LDR     R1,[R0,R3,LSL #1]    ; 将 R0+R3×2 地址上的存储单元的内容读出，存
                             ; 入 R1
```

6. 多寄存器寻址

多寄存器寻址即是一次可传送几个寄存器值，允许一条指令传送 16 个寄存器的任何子集或所有寄存器。多寄存器寻址指令举例如下：

```
LDMIA   R1!,{R2-R7,R12}      ; 将 R1 指向的单元中的数据读出到 R2～R7、R12 中
                             ; (R1 自动加 1)
STMIA   R0!,{R2-R7,R12}      ; 将寄存器 R2～R7、R12 的值保存到 R0 指向的存储
                             ; 单元中(R0 自动加 1)
```

使用多寄存器寻址指令时，寄存器子集的顺序是由小到大的顺序排列，连续的寄存器可用“-”连接，否则用“,”分隔书写。

7. 堆栈寻址

堆栈是一种按特定顺序进行存取的存储区，操作顺序分为“后进先出”或“先进后出”。堆栈寻址是隐含的，它使用一个专门的寄存器(堆栈指针)指向一块存储区域(堆栈)，指针所指向的存储单元即是堆栈的栈顶。存储器堆栈可分为两种：

- 向上生长：向高地址方向生长，称为递增堆栈。
- 向下生长：向低地址方向生长，称为递减堆栈。

堆栈指针指向最后压入的堆栈的有效数据项，称为满堆栈；堆栈指针指向下一个待压入数据的空位置，称为空堆栈。这样就有 4 种类型的堆栈表示递增和递减的满和空堆栈的各种组合。

- 满递增：堆栈通过增大存储器的地址向上增长，堆栈指针指向内含有效数据项的最高地址。指令如 LDMFA、STMFA 等。
- 空递增：堆栈通过增大存储器的地址向上增长，堆栈指针指向堆栈上的第一个空位置。指令如 LDMEA、STMEA 等。
- 满递减：堆栈通过减小存储器的地址向下增长，堆栈指针指向内含有效数据项的最低地址。指令如 LDMFD、STMFD 等。
- 空递减：堆栈通过减小存储器的地址向下增长，堆栈指针向堆栈下的第一个空位置。指令如 LDMED、STMED 等。

堆栈寻址指令举例如下：

```
STMFD  SP!,{R1-R7,LR}   ; 将 R1～R7、LR 入栈。满递减堆栈。
LDMFD  SP!,{R1-R7,LR}   ; 数据出栈，放入 R1～R7、LR 寄存器。满递减堆栈。
```

8. 块拷贝寻址

多寄存器传送指令用于将一块数据从存储器的某一位置拷贝到另一位置。块拷贝寻址指令举例如下：

```
STMIA   R0!,{R1-R7}     ; 将 R1～R7 的数据保存到存储器中。存储指针在保存
                        ; 第一个值之后增加，增长方向为向上增长。
STMIB   R0!,{R1-R7}     ; 将 R1～R7 的数据保存到存储器中。存储指针在保存
                        ; 第一个值之前增加，增长方向为向上增长。
STMDA  R0!,{R1-R7}      ; 将 R1～R7 的数据保存到存储器中。存储指针在保存
                        ; 第一个值之后增加，增长方向为向下增长。
STMDB  R0!,{R1-R7}      ; 将 R1～R7 的数据保存到存储器中。存储指针在保存
                        ; 第一个值之前增加，增长方向为向下增长。
```

9. 相对寻址

相对寻址是基址寻址的一种变通。由程序计数器 PC 提供基准地址，指令中的地址码字段作为偏移量，两者相加后得到的地址即为操作数的有效地址。相对寻址指令举例如下：

```
    BL      SUBR1       ; 调用到 SUBR1 子程序
    BEQ     LOOP        ; 条件跳转到 LOOP 标号处
    …
```

```
LOOP    MOV     R6,#1
        …
SUBR1   …
```

4.2 指令集介绍

本节介绍的是 ARM7TDMI(-S) 的指令集，包括 ARM 指令集和 Thumb 指令集。首先介绍 ARM 指令的基本格式及灵活的操作数，然后介绍条件码，再把 ARM 指令集、Thumb 指令集按类别分别说明。

在介绍 ARM 指令集之前，我们先看一个简单的 ARM 汇编程序，通过这一段程序读者可以了解 ARM 汇编指令格式、程序结构及基本风格，完整代码如程序清单 4.1 所示。

程序清单 4.1 寄存器相加

```
; 文件名：TEST1.S                                                    (1)
; 功能：实现两个寄存器相加                                           (2)
; 说明：使用 ARMulate 软件仿真调试                                   (3)
        AREA    Example1,CODE,READONLY  ; 声明代码段 Example1       (4)
        ENTRY                           ; 标识程序入口              (5)
        CODE32                          ; 声明 32 位 ARM 指令       (6)
START   MOV     R0,#0                   ; 设置参数                  (7)
        MOV     R1,#10                                              (8)
LOOP    BL      ADD_SUB                 ; 调用子程序 ADD_SUB        (9)
        B       LOOP                    ; 跳转到 LOOP               (10)
                                                                    (11)
ADD_SUB                                                             (12)
        ADDS    R0,R0,R1                ; R0 = R0 + R1              (13)
        MOV     PC,LR                   ; 子程序返回                (14)
                                                                    (15)
        END                             ; 文件结束                  (16)
```

第 1、2、3 行为程序说明，使用“;”进行注释，“;”号后面至行结束均为注释内容；

第 4 行声明一个代码段，ARM 汇编程序至少要声明一个代码段；

第 5 行标识程序入口，在仿真调试时会从指定入口处开始运行程序；

第 6 行声明 32 位 ARM 指令，ARM7TDMI(-S)复位后是 ARM 状态；

第 7～14 行为实际代码，标号要顶格书写(如 START、LOOP、ADD_SUB)，而指令不能顶格书写。BL 为调用子程序指令，它会把返回地址(即下一条指令的地址)存到 LR，然后跳转到子程序 ADD_SUB。子程序 ADD_SUB 处理结束后，将 LR 的值装入 PC 即可返回；(第 11、15 行为空行，目的在于增强程序的可读性)

第 16 行用于指示汇编源文件结束，每一个 ARM 汇编文件均要用 END 声明结束。

4.2.1 ARM 指令集

1. 指令格式

ARM 指令的基本格式如下：

<opcode> {<cond>} {S} <Rd> ,<Rn>{,<operand2>}

其中，<>号内的项是必需的，{}号内的项是可选的。如<opcode>是指令助记符，这是

必须书写的，而{<cond>}为指令执行条件，是可选项。若不书写则使用默认条件 AL(无条件执行)。

opcode	指令助记符，如 LDR、STR 等。
cond	执行条件，如 EQ、NE 等。
S	是否影响 CPSR 寄存器的值，书写时影响 CPSR。
Rd	目标寄存器。
Rn	第 1 个操作数的寄存器。
operand2	第 2 个操作数。

指令格式举例如下：

```
LDR     R0,[R1]         ; 读取 R1 地址上的存储器单元内容，执行条件 AL
BEQ     DATAEVEN        ; 分支指令，执行条件 EQ，即相等则跳转
                        ; 到 DATAEVEN
ADDS    R1,R1,#1        ; 加法指令，R1+1=>R1，影响 CPSR 寄存器(S)
SUBNES  R1,R1,#0x10     ; 条件执行减法运算(NE)，R1-0x10=>R1，影响
                        ; CPSR 寄存器(S)
```

第 2 个操作数

在 ARM 指令中，灵活地使用第 2 个操作数能够提高代码效率。第 2 个操作数的形式如下：

- #immed_8r——常数表达式

该常数必须对应 8 位位图(pattern)，即常数是由一个 8 位的常数循环移位偶数位得到。

合法常量：0x3FC（0xFF<<2）、0、0xF0000000(0xF0<<24)、200(0xC8)、0xF0000001(0x1F<<28)。

非法常量：0x1FE、511、0xFFFF、0x1010、0xF0000010。

常数表达式应用举例：

```
MOV     R0,#1           ; R0=1
AND     R1,R2,#0x0F     ; R2 与 0x0F，结果保存在 R1
LDR     R0,[R1],#-4     ; 读取 R1 地址上的存储器单元内容，且 R1=R1-4
```

- Rm——寄存器方式

在寄存器方式下，操作数即为寄存器的数值。

寄存器方式应用举例：

```
SUB     R1,R1,R2        ; R1-R2=>R1
MOV     PC,R0           ; PC=R0，程序跳转到指定地址
LDR     R0,[R1],-R2     ; 读取 R1 地址上的存储器单元内容并存入 R0，且 R1=R1-R2
```

- Rm,shift——寄存器移位方式。

将寄存器的移位结果作为操作数，但 Rm 值保存不变，移位方法如下：

ASR #n	算术右移 n 位(1≤n≤32)。
LSL #n	逻辑左移 n 位(1≤n≤31)。
LSR #n	逻辑右移 n 位(1≤n≤32)。
ROR #n	循环右移 n 位(1≤n≤31)。
RRX	带扩展的循环右移 1 位。
type Rs	其中，type 为 ASR、LSL、LSR 和 ROR 中的一种；Rs 偏移量寄存器，低 8 位有效。若其值大于或等于 32，则第 2 个操作数的结果为 0(ASR、ROR 例外)。

寄存器偏移方式应用举例：

```
ADD     R1,R1,R1,LSL #3   ; R1=R1X9
SUB     R1,R1,R2,LSR #2   ; R1=R1-R2/4
```

R15 为处理器的程序计数器 PC，一般不要对其进行操作，而且有些指令是不允许使用 R15 的，如 UMULL 指令。

2. 条件码

使用指令条件码可实现高效的逻辑操作，提高代码效率。指令条件码表如表 4.1 所示。

表 4.1 指令条件码表

操作码	条件码助记符	标志	含义
0000	EQ	Z=1	相等
0001	NE	Z=0	不相等
0010	CS/HS	C=1	无符号数大于或等于
0011	CC/LO	C=0	无符号数小于
0100	MI	N=1	负数
0101	PL	N=0	正数或零
0110	VS	V=1	溢出
0111	VC	V=0	没有溢出
1000	HI	C=1，Z=0	无符号数大于
1001	LS	C=0，Z=1	无符号数小于或等于
1010	GE	N=V	有符号数大于或等于
1011	LT	N!=V	有符号数小于
1100	GT	Z=0,N=V	有符号数大于
1101	LE	Z=1,N!=V	有符号数小于或等于
1110	AL	任何	无条件执行 (指令默认条件)
1111	NV	任何	从不执行(不要使用)

对于 Thumb 指令集，只有 B 指令具有条件码执行功能。此指令的条件码同表 4.1。但如果为无条件执行时，条件码助记符 AL 不能在指令中书写。

条件码应用举例如下：

比较两个值大小，并进行相应加 1 处理，C 代码为

```
if(a>b)    a++;
  else     b++;
```

对应的 ARM 指令如下（其中 R0 为 a，R1 为 b）：

```
CMP         R0,R1         ; R0 与 R1 比较
ADDHI       R0,R0,#1      ; 若 R0>R1，则 R0=R0+1
ADDLS       R1,R1,#1      ; 若 R0≤1，则 R1=R1+1
```

若两个条件均成立，则将这两个数值相加，C 代码为

```
if( (a!=10)&&(b!=20) )    a = a+b;
```

对应的 ARM 指令如下。其中 R0 为 a，R1 为 b。

```
CMP         R0,#10        ; 比较 R0 是否为 10
```

```
CMPNE    R1,#20      ; 若 R0 不为 10，则比较 R1 是否为 20
ADDNE    R0,R0,R1    ; 若 R0 不为 10 且 R1 不为 20，指令执行，R0=R0+R1
```

3. ARM 存储器访问指令

ARM 处理器是加载/存储体系结构的典型的 RISC 处理器，对存储器的访问只能使用加载和存储指令实现。ARM 的加载/存储指令实现字、半字、无符号/有符号字节操作；多寄存器加载/存储指令可实现一条指令加载/存储多个寄存器的内容，大大提高效率；SWP 指令是一条寄存器和存储器内容交换的指令，可用于信号量操作等。ARM 处理器是冯•诺依曼存储结构，程序空间、RAM 空间及 I/O 映射空间统一编址，除对 RAM 操作以外，对外围 IO、程序数据的访问均要通过加载/存储指令进行。

ARM 存储器访问指令见表 4.2。

表 4.2 ARM 存储器访问指令

助记符	说明	操作	条件码位置
LDR Rd,addressing	加载字数据	Rd←[addressing]，addressing 索引	LDR{cond}
LDRB Rd,addressing	加载无符号字节数据	Rd←[addressing]，addressing 索引	LDR{cond}B
LDRT Rd, addressing	以用户模式加载字数据	Rd←[addressing]，addressing 索引	LDR{cond}T
LDRBT Rd, addressing	以用户模式加载无符号字节数据	Rd←[addressing]，addressing 索引	LDR{cond}BT
LDRH Rd, addressing	加载无符号半字数据	Rd←[addressing]，addressing 索引	LDR{cond}H
LDRSB Rd, addressing	加载有符号字节数据	Rd←[addressing]，addressing 索引	LDR{cond}SB
LDRSH Rd, addressing	加载有符号半字数据	Rd←[addressing]，addressing 索引	LDR{cond}SH
STR Rd, addressing	存储字数据	[addressing] ←Rd，addressing 索引	STR{cond}
STRB Rd,addressing	存储字节数据	[addressing] ←Rd，addressing 索引	STR{cond}B
STRT Rd,addressing	以用户模式存储字数据	[addressing] ←Rd，addressing 索引	STR{cond}T
STRBT Rd,addressing	以用户模式存储字节数据	[addressing] ←Rd，addressing 索引	STR{cond}BT
STRH Rd,addressing	存储半字数据	[addressing] ←Rd，addressing 索引	STR{cond}H
LDM{mode} Rn{!},reglist	多寄存器加载	reglist←[Rn...]，Rn 回写等	LDM{cond}{mode}
STM{mode} Rn{!},reglist	多寄存器存储	[Rn...]←reglist，Rn 回写等	STM{cond}{mode}
SWP Rd,Rm,Rn	寄存器和存储器字数据交换	Rd←[Rn]，[Rn]←Rm (Rn≠Rd 或 Rm)	SWP{cond}
SWPB Rd,Rm,Rn	寄存器和存储器字节数据交换	Rd←[Rn]，[Rn]←Rm (Rn≠Rd 或 Rm)	SWP{cond}B

- **LDR 和 STR——加载存储指令**

加载/存储字和无符号字节指令

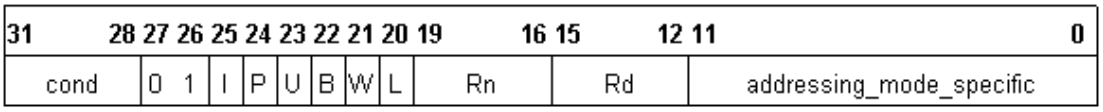
使用 STR 指令将单一字节或字存储到内存，使用 LDR 指令从内存中加载单一字节或字到寄存器。LDR 指令用于从内存中读取数据放入寄存器中；STR 指令用于将寄存器中的数据保存到内存。指令格式如下：

```
LDR{cond}{T}     Rd,<地址>      ; 加载指定地址上的数据(字)，放入 Rd 中
STR{cond}{T}     Rd,<地址>      ; 存储数据(字)到指定地址的存储单元，要存储
                                ; 的数据在 Rd 中
LDR{cond}B{T}    Rd,<地址>      ; 加载字节数据，放入 Rd 中，即 Rd 最低字节
                                ; 有效，高 24 位清零
STR{cond}B{T}    Rd,<地址>      ; 存储字节数据，要存储的数据在 Rd，最低字
                                ; 节有效
```

其中，T 为可选后缀。若指令有 T，那么即使处理器是在特权模式下，存储系统也将访

问看成是处理器是在用户模式下。T 在用户模式下无效，不能与前索引偏移一起使用 T。

指令编码格式：



I，P，U，W	用于区别不同的地址模式(偏移量)。I 为 0 时，偏移量为 12 位立即数；I 为 1 时，偏移量为寄存器移位。P 表示前/后变址，U 表示加/减，W 表示回写。
L	用于区别加载(L 为 1)或存储(L 为 0)。
B	用于区别字节访问(B 为 1)或字访问(B 为 0)。
Rn	基址寄存器。
Rd	源/目标寄存器。

LDR/STR 指令寻址是非常灵活的，由两部分组成，一部分为一个基址寄存器，可以为任一个通用寄存器；另一部分为一个地址偏移量。地址偏移量有以下 3 种格式：

(1) 立即数。立即数可以是一个无符号的数值。这个数据可以加到基址寄存器，也可以从基址寄存器中减去这个数值。指令举例如下：

```
LDR     R1,[R0,#0x12]    ; 将 R0+0x12 地址处的数据读出，保存到 R1 中
                         ; (R0 的值不变)
LDR     R1,[R0,#-0x12]   ; 将 R0-0x12 地址处的数据读出，保存到 R1 中
                         ; (R0 的值不变)
LDR     R1,[R0]          ; 将 R0 地址处的数据读出，保存到 R1 中 (零偏移)
```

(2) 寄存器。寄存器中的数值可以加到基址寄存器，也可以从基址寄存器中减去这个数值。指令举例如下：

```
LDR     R1,[R0,R2]   ; 将 R0+R2 地址处的数据读出，保存到 R1 中 (R0 的值不变)
LDR     R1,[R0,-R2]  ; 将 R0-R2 地址处的数据读出，保存到 R1 中 (R0 的值不变)
```

(3) 寄存器及移位常数。寄存器移位后的值可以加到基址寄存器，也可以从基址寄存器中减去这个数值。指令举例如下：

```
LDR     R1,[R0,R2,LSL #2]    ; 将 R0+R2×4 地址处的数据读出，保存到 R1 中
                             ; (R0、R2 的值不变)
LDR     R1,[R0,-R2,LSL #2]   ; 将 R0-R2×4 地址处的数据读出，保存到 R1 中
                             ; (R0、R2 的值不变)
```

从寻址方式的地址计算方法分，加载/存储指令有以下 4 种形式：

(1) 零偏移。Rn 的值作为传送数据的地址，即地址偏移量为 0。指令举例如下：

```
LDR     Rd,[Rn]
```

(2) 前索引偏移。在数据传送之前，将偏移量加到 Rn 中，其结果作为传送数据的存储地址。若使用后缀“!”，则结果写回到 Rn 中，且 Rn 的值不允许为 R15。指令举例如下：

```
LDR     Rd,[Rn,#0x04]!
LDR     Rd,[Rn,#-0x04]
```

(3) 程序相对偏移。程序相对偏移是前索引形式的另一个版本。汇编器由 PC 寄存器计算偏移量，并将 PC 寄存器作为 Rn 生成前索引指令。不能使用后缀“!”。指令举例如下：

```
LDR     Rd,label     ; label 为程序标号，label 必须是在当前指令的±4KB 范围内
```

(4) 后索引偏移。Rn 的值用做传送数据的存储地址。在数据传送后，将偏移量与 Rn 相加，结果写回到 Rn 中。Rn 不允许是 R15。指令举例如下：

```
LDR     Rd,[Rn],#0x04
```

地址对齐——大多数情况下，必须保证用于 32 位传送的地址是 32 位对齐的。

加载/存储字和无符号字节指令举例如下：

```
LDR     R2,[R5]         ; 加载 R5 指定地址上的数据(字)，放入 R2 中
STR     R1,[R0,#0x04]   ; 将 R1 的数据存储到 R0+0x04 的存储单元，R0 值不变
LDRB    R3,[R2],#1      ; 读取 R2 地址上的一字节数据，并保在到 R3 中，R2=R2+1
STRB    R6,[R7]         ; 将 R6 的数据保存到 R7 指定的地址中，只存储一字节
                        ; 数据
```

加载/存储半字和有符号字节

这类 LDR/STR 指令可加载有符号字节、加载有符号半字、加载/存储无符号半字。偏移量格式、寻址方式与加载/存储字和无符号字节指令相同。指令格式如下：

```
LDR{cond}SB     Rd,<地址>   ; 加载指定地址上的数据(有符号字节)，放入 Rd 中
LDR{cond}SH     Rd,<地址>   ; 加载指定地址上的数据(有符号半字)，放入 Rd 中
LDR{cond}H      Rd,<地址>   ; 加载半字数据，放入 Rd 中，即 Rd 最低 16 位有效，
                            ; 高 16 位清零
STR{cond}H      Rd,<地址>   ; 存储半字数据，要存储的数据在 Rd，最低 16 位有效
```

说明：有符号位半字/字节加载是指用符号位加载扩展到 32 位；无符号位半字加载是指零扩展到 32 位。

指令编码格式：

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0 0 0	P	U	I	W	L	Rn	Rd	addr_mode	1	S	H	1	addr_mode

I，P，U，W	用于区别不同的地址模式(偏移量)。I 为 0 时，偏移量为 8 位立即数；I 为 1 时，偏移量为寄存器偏移。P 表示前/后变址，U 表示加/减，W 表示回写。
L	用于区别加载(L 为 1)或存储(L 为 0)。
S	用于区别有符号访问(S 为 1)和无符号访问(S 为 0)。
H	用于区别半字访问(H 为 1)或字节访问(H 为 0)。
Rn	基址寄存器。
Rd	源/目标寄存器。

地址对齐——对半字传送的地址必须为偶数。非半字对齐的半字加载将使 Rd 内容不可靠；非半字对齐的半字存储将使指定地址的 2 字节存储内容不可靠。

加载/存储半字和有符号字节指令举例如下：

```
LDRSB   R1,[R0,R3]  ; 将 R0+R3 地址上的字节数据读出到 R1，高 24 位用符号位
                    ; 扩展
LDRSH   R1,[R9]     ; 将 R9 地址上的半字数据读出到 R1，高 16 位用符号位扩展
LDRH    R6,[R2],#2  ; 将 R2 地址上的半字数据读出到 R6，高 16 位用零扩展，
                    ; R2=R2+2
STRH    R1,[R0,#2]! ; 将 R1 的数据保存到 R0+2 地址中，只存储低 2 字节数据，
```

```
                                ; R0=R0+2
```

LDR/STR 指令用于对内存变量的访问、内存缓冲区数据的访问、查表、外围部件的控制操作等等。若使用 LDR 指令加载数据到 PC 寄存器，则实现程序跳转功能，这样也就实现了程序散转。

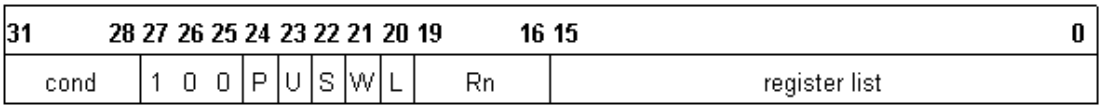
- **LDM 和 STM——多寄存器加载/存储指令**

多寄存器加载/存储指令可以实现在一组寄存器和一块连续的内存单元之间传输数据。LDM 为加载多个寄存器；STM 为存储多个寄存器。允许一条指令传送 16 个寄存器的任何子集或所有寄存器。指令格式如下：

LDM{cond}<模式> Rn{!},reglist{^}

STM{cond}<模式> Rn{!},reglist{^}

指令编码格式：



register list	寄存器列表，b0 位与 R0 对应，b15 位与 R15 对应。
P，U，W	用于区别不同的地址模式。P 表示前/后变址，U 表示加/减，W 表示回写。
S	恢复 CPSR 和强制用户位。当 PC 寄存器包含在 LDM 指令的 reglist 中，且 S 为 1 时，则当前模式的 SPSR 将被拷贝到 CPSR，成为一个原子的返回和恢复状态指令。若 reglist 不包含 PC 寄存器，S 为 1，则加载/存储的是用户模式的寄存器。
L	用于区别加载(L 为 1)或存储(L 为 0)。
Rn	基址寄存器。

LDM/STM 的主要用途是现场保护、数据复制、参数传送等。其模式有如下 8 种(前面 4 种用于数据块的传输，后面 4 种是堆栈操作)：

(1) IA：每次传送后地址加 4；
(2) IB：每次传送前地址加 4；
(3) DA：每次传送后地址减 4；
(4) DB：每次传送前地址减 4；
(5) FD：满递减堆栈；
(6) ED：空递减堆栈；
(7) FA：满递增堆栈；
(8) EA：空递增堆栈。

指令格式中，寄存器 Rn 为基址寄存器，装有传送数据的初始地址，Rn 不允许为 R15；后缀“!”表示最后的地址写回到 Rn 中。寄存器列表 reglist 可包含多于一个寄存器或包含寄存器范围，使用“,”分开，如{R1,R2,R6-R9}，寄存器由小到大排列；“^”后缀不允许在用户模式或系统模式下使用，若在 LDM 指令且寄存器列表中包含有 PC 时使用，那么除了正常的多寄存器传送外，将 SPSR 也拷贝到 CPSR 中，这可用于异常处理返回。使用“^”后缀进行数据传送且寄存器列表不包含 PC 时，加载/存储的是用户模式的寄存器，而不是当前模式的寄存器。

当 Rn 在寄存器列表中且使用后缀“!”时，对于 STM 指令，若 Rn 为寄存器列表中的最

低数字的寄存器，则会将 Rn 的初值保存；其它情况下 Rn 的加载值和存储值不可预知。

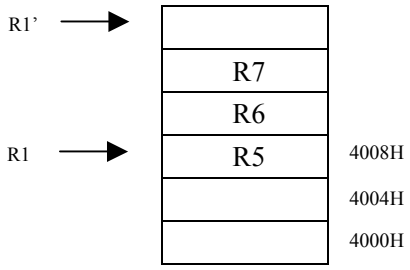
地址对齐——这些指令忽略地址的位[1:0]。

多寄存器加载/存储指令举例如下：

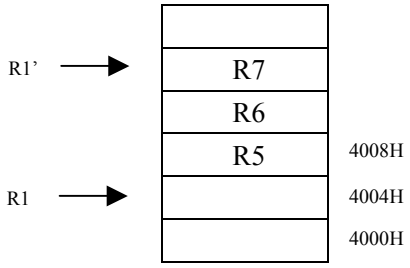
```
LDMIA   R0!,{R3-R9}       ; 加载 R0 指向的地址上的多字数据，保存到 R3～R9 中，
                          ; R0 值更新
STMIA   R1!,{R3-R9}       ; 将 R3～R9 的数据存储到 R1 指向的地址上，R1 值更新
STMFD   SP!,{R0-R7,LR}    ; 现场保存，将 R0～R7、LR 入栈
LDMFD   SP!,{R0-R7,PC}^   ; 恢复现场，异常处理返回
```

在进行数据复制时，先设置好源数据指针和目标指针，然后使用块拷贝寻址指令 LDMIA/STMIA、LDMIB/STMIB、LDMDA/STMDA、LDMDB/STMDB 进行读取和存储。而进行堆栈操作时，则要先设置堆栈指针，一般使用 SP，然后使用堆栈寻址指令 STMFD/LDMFD、STMED/LDMED、STMFA/LDMFA 和 STMEA/LDMEA 实现堆栈操作。

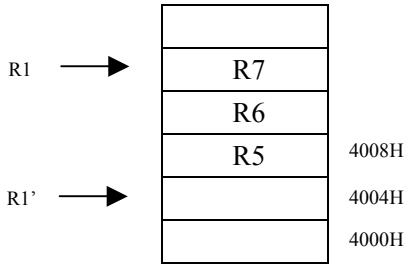
多寄存器传送指令示意图如图 4.2 所示，其中 R1 为指令执行前的基址寄存器，R1’则为指令执行完后的基址寄存器。



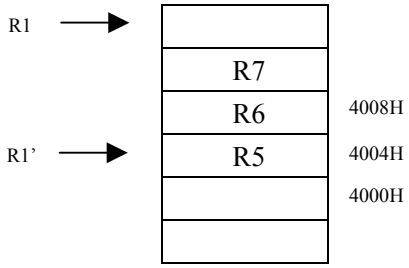
(a) 指令 STMIA R1!,{R5-R7}



(b) 指令 STMIB R1!,{R5-R7}



(c) 指令 STMDA R1!,{R5-R7}



(d) 指令 STMDB R1!,{R5-R7}

图 4.2 多寄存器传送指令示意图

使用多寄存器传送指令时，基址寄存器的地址是向上增长还是向下增长，地址是在加载/存储数据之前还是之后增加/减少，其对应关系如表 4.3 所示。

表 4.3 多寄存器传送指令映射

增长的方向 / 增长的先后		向上生长		向下生长	
		满	空	满	空
增加	之前	STMIB STMFA			LDMIB LDMED
	之后		STMIA STMEA	LDMIA LDMFD	
减少	之前		LDMDB LDMEA	STMDB STMFD	
	之后	LDMDA LDMFA			STMDA STMED

- **SWP——寄存器和存储器交换指令**

SWP 指令用于将一个内存单元(该单元地址放在寄存器 Rn 中)的内容读取到一个寄存器 Rd 中，同时将另一个寄存器 Rm 的内容写入到该内存单元中。使用 SWP 可实现信号量操作。指令格式如下：

SWP{cond}{B} Rd,Rm,[Rn]

其中，B 为可选后缀，若有 B，则交换字节，否则交换 32 位字；Rd 为数据从存储器加载到的寄存器；Rm 的数据用于存储到存储器中，若 Rm 与 Rn 相同，则为寄存器与存储器内容进行交换；Rn 为要进行数据交换的存储器地址，Rn 不能与 Rd 和 Rm 相同。

指令编码格式：

31 28	27 26 25 24 23	22	21 20	19 16	15 12	11 4	3 0
cond	0 0 0 1 0	B	0 0	Rn	Rd	0 0 0 0 1 0 0 1	Rm

B 用于区别无符号字节(B 为 1)或字(B 为 0)。

Rm 源寄存器。

Rd 目标寄存器。

Rn 基址寄存器。

SWP 指令举例如下：

```
SWP     R1,R1,[R0]    ; 将 R1 的内容与 R0 指向的存储单元的内容进行交换
```

```
SWPB    R1,R2,[R0]   ; 将 R0 指向的存储单元内的容读取一字节数据到 R1 中(高 24
                     ; 位清零)，并将 R2 的内容写入到该内存单元中(最低字节有效)
```

4. ARM 数据处理指令

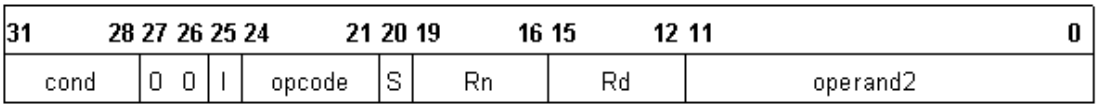
数据处理指令大致可分为 3 类：数据传送指令(如 MOV、MVN)、算术逻辑运算指令(如 ADD、SUB、AND)，比较指令(如 CMP、TST)。数据处理指令只能对寄存器的内容进行操作。**所有 ARM 数据处理指令均可选择使用 S 后缀，并影响状态标志。**比较指令 CMP、CMN、TST 和 TEQ 不需要后缀 S，它们会直接影响状态标志。

ARM 数据处理指令见表 4.4。

表 4.4 ARM 数据处理指令

助记符	说明	操作	条件码位置
MOV Rd,operand2	数据传送	Rd←operand2	MOV{cond}{S}
MVN Rd,operand2	数据非传送	Rd←(~operand2)	MVN{cond}{S}
ADD Rd, Rn, operand2	加法运算指令	Rd←Rn+operand2	ADD{cond}{S}
SUB Rd, Rn, operand2	减法运算指令	Rd←Rn-operand2	SUB{cond}{S}
RSB Rd, Rn, operand2	逆向减法指令	Rd←operand2-Rn	RSB{cond}{S}
ADC Rd, Rn, operand2	带进位加法	Rd←Rn+operand2+Carry	ADC{cond}{S}
SBC Rd, Rn, operand2	带进位减法指令	Rd←Rn-operand2-(NOT)Carry	SBC{cond}{S}
RSC Rd, Rn, operand2	带进位逆向减法指令	Rd←operand2-Rn-(NOT)Carry	RSC{cond}{S}
AND Rd, Rn, operand2	逻辑与操作指令	Rd←Rn & operand2	AND{cond}{S}
ORR Rd, Rn, operand2	逻辑或操作指令	Rd←Rn | operand2	ORR{cond}{S}
EOR Rd, Rn, operand2	逻辑异或操作指令	Rd←Rn ^ operand2	EOR{cond}{S}
BIC Rd, Rn, operand2	位清除指令	Rd←Rn & (~operand2)	BIC{cond}{S}
CMP Rn, operand2	比较指令	标志 N、Z、C、V←Rn-operand2	CMP{cond}
CMN Rn, operand2	负数比较指令	标志 N、Z、C、V←Rn+operand2	CMN{cond}
TST Rn, operand2	位测试指令	标志 N、Z、C、V←Rn & operand2	TST{cond}
TEQ Rn, operand2	相等测试指令	标志 N、Z、C、V←Rn ^ operand2	TEQ{cond}

ARM 数据处理指令编码格式：



opcode	数据处理指令操作码。
I	用于区别立即数(I 为 1)或寄存器移位(I 为 0)。
S	设置条件码。
Rn	第一操作数寄存器。
Rd	目标寄存器。
operand2	第二个操作数。

若指令不需要全部的可用操作数时(如 MOV 指令的 Rn)，不用的寄存器域应设置为 0(由编译器自动完成)。对于比较指令，b20 位固定为 1。ARM 数据处理指令操作码见表 4.5。

表 4.5 ARM 数据处理指令操作码

操作码	指令助记符	说明
0000	AND	逻辑与操作指令
0001	EOR	逻辑异或操作指令
0010	SUB	减法运算指令
0011	RSB	逆向减法指令
0100	ADD	加法运算指令
0101	ADC	带进位加法
0110	SBC	带进位减法指令
0111	RSC	带进位逆向减法指令
1000	TST	位测试指令
1001	TEQ	相等测试指令
1010	CMP	比较指令
1011	CMN	负数比较指令
1100	ORR	逻辑或操作指令
1101	MOV	数据传送
1110	BIC	位清除指令
1111	MVN	数据非传送

数据传送指令

- **MOV——数据传送指令**

MOV 将 8 位图(pattern)立即数或寄存器(operand2)传送到目标寄存器(Rd)，可用于移位运算等操作。指令格式如下：

MOV{cond}{S} Rd,operand2

MOV 指令举例如下：

```
MOV     R1,#0x10          ; R1=0x10
MOV     R0,R1             ; R0=R1
MOVS    R3,R1,LSL #2      ; R3=R1<<2，并影响标志位
MOV     PC,LR             ; PC=LR，子程序返回
```

- **MVN——数据非传送指令**

MVN 指令将 8 位图(pattern)立即数或寄存器(operand2)按位取反后传送到目标寄存器(Rd)，因为其具有取反功能，所以可以装载范围更广的立即数。指令格式如下：

MVN{cond}{S} Rd,operand2

MVN 指令举例如下：

```
MVN         R1,#0xFF          ; R1=0xFFFFFF00
MVN         R1,R2             ; 将 R2 取反，结果存到 R1
```

算术逻辑运算指令

- **ADD——加法运算指令**

ADD 指令将 operand2 的值与 Rn 的值相加，结果保存到 Rd 寄存器。指令格式如下：

ADD{cond}{S} Rd,Rn, operand2

ADD 指令举例如下：

```
ADDS    R1,R1,#1          ; R1=R1+1
ADD     R1,R1,R2          ; R1=R1+R2
ADDS    R3,R1,R2,LSL #2   ; R3=R1+R2<<2
```

- **SUB——减法运算指令**

SUB 指令用寄存器 Rn 减去 operand2，结果保存到 Rd 中。指令格式如下：

SUB{cond}{S} Rd,Rn, operand2

SUB 指令举例如下：

```
SUBS    R0,R0,#1          ; R0=R0-1
SUBS    R2,R1,R2          ; R2=R1-R2
SUB     R6,R7,#0x10       ; R6=R7-0x10
```

- **RSB——逆向减法指令**

RSB 指令将 operand2 的值减去 Rn，结果保存到 Rd 中。指令格式如下：

RSB{cond}{S} Rd,Rn, operand2

RSB 指令举例如下：

```
RSB     R3,R1,#0xFF00     ; R3=0xFF00-R1
RSBS    R1,R2,R2,LSL #2   ; R1=R2<<2-R2=R2×3
RSB     R0,R1,#0          ; R0=-R1
```

- **ADC——带进位加法指令**

将 operand2 的值与 Rn 的值相加，再加上 CPSR 中的 C 条件标志位，结果保存到 Rd 寄存器。指令格式如下：

ADC{cond}{S} Rd,Rn, operand2

ADC 指令举例如下：

```
ADDS    R0,R0,R2
ADC     R1,R1,R3          ; 使用 ADC 实现 64 位加法，(R1、R0)=(R1、R0)+(R3、R2)
```

- **SBC——带进位减法指令**

SBC 指令用寄存器 Rn 减去 operand2，再减去 CPSR 中的 C 条件标志位的非(即若 C 标志清零，则结果减去 1)，结果保存到 Rd 中。指令格式如下：

SBC{cond}{S} Rd,Rn, operand2

SBC 指令举例如下：

```
SUBS    R0,R0,R2
SBC     R1,R1,R3          ; 使用 SBC 实现 64 位减法，(R1、R0)=(R1、R0)-(R3、R2)
```

- **RSC——带进位逆向减法指令**

RSB 指令用寄存器 operand2 减去 Rn，再减去 CPSR 中的 C 条件标志位，结果保存到 Rd 中。指令格式如下：

RSC{cond}{S} Rd,Rn, operand2

RSC 指令举例如下：

```
RSBS    R2,R0,#0
RSC     R3,R1,#0    ; 使用 RSC 指令实现求 64 位数值的负数
```

- **AND——逻辑“与”操作指令**

AND 指令将 operand2 的值与寄存器 Rn 的值按位作逻辑“与”操作，结果保存到 Rd 中。指令格式如下：

AND{cond}{S} Rd,Rn, operand2

AND 指令举例如下：

```
ANDS    R0,R0,#0x01  ; R0=R0&0x01，取出最低位数据
AND     R2,R1,R3     ; R2=R1&R3
```

- **ORR——逻辑“或”操作指令**

ORR 指令将 operand2 的值与寄存器 Rn 的值按位作逻辑“或”操作，结果保存到 Rd 中。指令格式如下：

ORR{cond}{S} Rd,Rn, operand2

ORR 指令举例如下：

```
ORR     R0,R0,#0x0F     ; 将 R0 的低 4 位置 1
MOV     R1,R2,LSR #24
ORR     R3,R1,R3,LSL #8 ; 使用 ORR 指令将 R2 的高 8 位数据移入到 R3 低 8 位中
```

- **EOR——逻辑“异或”操作指令**

EOR 指令将 operand2 的值与寄存器 Rn 的值按位作逻辑“异或”操作，结果保存到 Rd 中。指令格式如下：

EOR{cond}{S} Rd,Rn, operand2

EOR 指令举例如下：

```
EOR     R1,R1,#0x0F ; 将 R1 的低 4 位取反
EOR     R2,R1,R0    ; R2=R1^R0
EORS    R0,R5,#0x01 ; 将 R5 和 0x01 进行逻辑异或，结果保存到 R0，并影响标志位
```

- **BIC——位清除指令**

BIC 指令将寄存器 Rn 的值与 operand2 的值的反码按位作逻辑“与”操作，结果保存到 Rd 中。指令格式如下：

BIC{cond}{S} Rd,Rn, operand2

BIC 指令举例如下：

```
BIC     R1,R1,#0x0F ; 将 R1 的低 4 位清零，其它位不变
```

```
BIC     R1,R2,R3    ; 将 R3 的反码和 R2 相逻辑“与”，结果保存到 R1 中
```

比较指令

- **CMP——比较指令**

CMP 指令将寄存器 Rn 的值减去 operand2 的值，根据操作的结果更新 CPSR 中的相应条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。指令格式如下：

```
CMP{cond}   Rn, operand2
```

CMP 指令举例如下：

```
CMP     R1,#10  ; R1 与 10 比较，设置相关标志位
CMP     R1,R2   ; R1 与 R2 比较，设置相关标志位
```

CMP 指令与 SUBS 指令的区别在于 CMP 指令不保存运算结果。在进行两个数据的大小判断时，常用 CMP 指令及相应的条件码来操作。

- **CMN——负数比较指令**

CMN 指令使用寄存器 Rn 的值加上 operand2 的值，根据操作的结果更新 CPSR 中的相应条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。指令格式如下：

```
CMN{cond}   Rn, operand2
```

CMN 指令举例如下：

```
CMN     R0,#1   ; R0+1，判断 R0 是否为 1 的补码。若是，则 Z 置位
```

CMN 指令与 ADDS 指令的区别在于 CMN 指令不保存运算结果。CMN 指令可用于负数比较，比如 CMN R0,#1 指令则表示 R0 与-1 比较，若 R0 为-1(即 1 的补码)，则 Z 置位；否则 Z 复位。

- **TST——位测试指令**

TST 指令将寄存器 Rn 的值与 operand2 的值按位作逻辑“与”操作，根据操作的结果更新 CPSR 中的相应条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。指令格式如下：

```
TST{cond}       Rn, operand2
```

TST 指令举例如下：

```
TST     R0,#0x01    ; 判断 R0 的最低位是否为 0
TST     R1,#0x0F    ; 判断 R1 的低 4 位是否为 0
```

TST 指令与 ANDS 指令的区别在于 TST 指令不保存运算结果。TST 指令通常与 EQ、NE 条件码配合使用，当所有测试位均为 0 时，EQ 有效，而只要有一个测试位不为 0，则 NE 有效。

- **TEQ——相等测试指令**

TEQ 指令将寄存器 Rn 的值与 operand2 的值按位作逻辑“异或”操作，根据操作的结果更新 CPSR 中的相应条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。指令格式如下：

TEQ{cond} Rn, operand2

TEQ 指令举例如下：

```
TEQ     R0,R1   ; 比较 R0 与 R1 是否相等 (不影响 V 位和 C 位)
```

TEQ 指令与 EORS 指令的区别在于 TEQ 指令不保存运算结果。使用 TEQ 进行相等测试时，常与 EQ、NE 条件码配合使用。当两个数据相等时，EQ 有效；否则 NE 有效。

5. 乘法指令

ARM7TDMI(-S)具有 32×32 乘法指令，32×32 乘加指令，32×32 结果为 64 位的乘/乘加指令。

ARM 乘法指令见表 4.6。

表 4.6 ARM 乘法指令

助记符	说明	操作	条件码位置
MUL Rd,Rm,Rs	32 位乘法指令	Rd←Rm*Rs (Rd≠Rm)	MUL{cond}{S}
MLA Rd,Rm,Rs,Rn	32 位乘加指令	Rd←Rm*Rs+Rn (Rd≠Rm)	MLA{cond}{S}
UMULL RdLo,RdHi,Rm,Rs	64 位无符号乘法指令	(RdLo,RdHi) ←Rm*Rs	UMULL{cond}{S}
UMLAL RdLo,RdHi,Rm,Rs	64 位无符号乘加指令	(RdLo,RdHi) ←Rm*Rs+(RdLo,RdHi)	UMLAL{cond}{S}
SMULL RdLo,RdHi,Rm,Rs	64 位有符号乘法指令	(RdLo,RdHi) ←Rm*Rs	SMULL{cond}{S}
SMLAL RdLo,RdHi,Rm,Rs	64 位有符号乘加指令	(RdLo,RdHi) ←Rm*Rs+(RdLo,RdHi)	SMLAL{cond}{S}

ARM 乘法指令编码格式：

31 28	27 24	23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0	opcode	S	Rd/RdHi	Rn/RdLo	Rs	1 0 0 1	Rm

opcode 乘法指令操作码。

S 设置条件码。

Rm 被乘数寄存器。

Rs 乘数的寄存器。

Rn/RdLo MLA 指令相加的寄存器或 64 位乘法指令的目标寄存器(低 32 位)。

Rd/RdHi 目标寄存器或 64 位乘法指令的目标寄存器(高 32 位)。

若指令不需要全部的可用操作数时(如 MUL 指令的 Rn)，不用的寄存器域应设置为 0(由编译器自动完成)。ARM 乘法指令操作码见表 4.7。

表 4.7 ARM 乘法指令操作码

操作码	指令助记符	说明
000	MUL	32 位乘法指令
001	MLA	32 位乘加指令
100	UMULL	64 位无符号乘法指令
101	UMLAL	64 位无符号乘加指令
110	SMULL	64 位有符号乘法指令
111	SMLAL	64 位有符号乘加指令

- **MUL——32 位乘法指令**

MUL 指令将 Rm 和 Rs 中的值相乘，结果的低 32 位保存到 Rd 中。指令格式如下：

MUL{cond}{S} Rd,Rm,Rs

MUL 指令举例如下：

```
MUL     R1,R2,R3      ; R1=R2×R3
MULS    R0,R3,R7      ; R0=R3×R7，同时设置 CPSR 中的 N 位和 Z 位
```

- **MLA——32 位乘加指令**

MLA 指令将 Rm 和 Rs 中的值相乘，再将乘积加上第 3 个操作数，结果的低 32 位保存到 Rd 中。指令格式如下：

MLA{cond}{S} Rd,Rm,Rs,Rn

MLA 指令举例如下：

```
MLA     R1,R2,R3,R0  ; R1=R2×R3+R0
```

- **UMULL——64 位无符号乘法指令**

UMULL 指令将 Rm 和 Rs 中的值作无符号数相乘，结果的低 32 位保存到 RdLo 中，而高 32 位保存到 RdHi 中。指令格式如下：

UMULL{cond}{S} RdLo,RdHi,Rm,Rs

UMULL 指令举例如下：

```
UMULL   R0,R1,R5,R8  ; (R1、R0)=R5×R8
```

- **UMLAL——64 位无符号乘加指令**

UMLAL 指令将 Rm 和 Rs 中的值作无符号数相乘，64 位乘积与 RdHi、RdLo 相加，结果的低 32 位保存到 RdLo 中，而高 32 位保存到 RdHi 中。指令格式如下：

UMLAL{cond}{S} RdLo,RdHi,Rm,Rs

UMLAL 指令举例如下：

```
UMLAL   R0,R1,R5,R8  ; (R1、R0)=R5×R8+(R1、R0)
```

- **SMULL——64 位有符号乘法指令**

SMULL 指令将 Rm 和 Rs 中的值作有符号数相乘，结果的低 32 位保存到 RdLo 中，而高 32 位保存到 RdHi 中。指令格式如下：

SMULL{cond}{S} RdLo,RdHi,Rm,Rs

SMULL 指令举例如下：

```
SMULL   R2,R3,R7,R6  ; (R3、R2)=R7×R6
```

- **SMLAL——64 位有符号乘加指令**

SMLAL 指令将 Rm 和 Rs 中的值作有符号数相乘，64 位乘积与 RdHi、RdLo 相加，结果的低 32 位保存到 RdLo 中，而高 32 位保存到 RdHi 中。指令格式如下：

SMLAL{cond}{S} RdLo,RdHi,Rm,Rs

SMLAL 指令举例如下：

SMLAL R2,R3,R7,R6 ; (R3、R2)=R7×R6+(R3、R2)

6. ARM 分支指令

在 ARM 中有两种方式可以实现程序的跳转，一种是使用分支指令直接跳转，另一种则是直接向 PC 寄存器赋值实现跳转。分支指令有分支指令 B、带链接的分支指令 BL、带状态切换的分支指令 BX。

ARM 分支指令见表 4.8。

表 4.8 ARM 分支指令

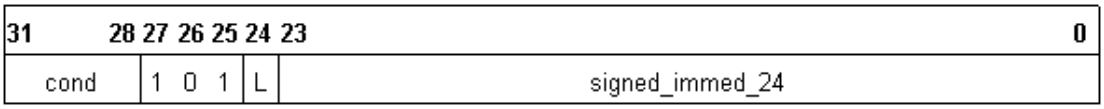
助记符	说明	操作	条件码位置
B label	分支指令	PC←label	B{cond}
BL label	带链接的分支指令	LR←PC-4，PC←label	BL{cond}
BX Rm	带状态切换的分支指令	PC←label，切换处理器状态	BX{cond}

- **B——分支指令**

B 指令跳转到指定的地址执行程序。指令格式如下：

B{cond} label

指令编码格式：



signed_immed_24 24 位有符号立即数(偏移量)。

L 区别分支(L 为 0)或带链接的分支指令(L 为 1)。

分支指令 B 举例如下：

```
B       WAITA       ; 跳转到 WAITA 标号处
B       0x1234      ; 跳转到绝对地址 0x1234 处
```

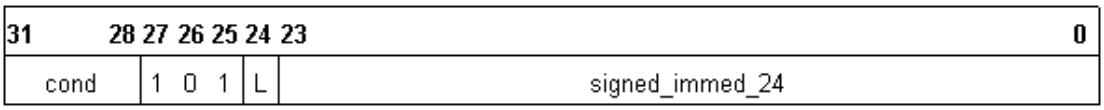
分支指令 B 限制在当前指令的±32M 字节地址范围内(ARM 指令为字对齐，最低 2 位地址固定为 0)。

- **BL——带连接的分支指令**

BL 指令先将下一条指令的地址拷贝到 R14(即 LR) 连接寄存器中，然后跳转到指定地址运行程序。指令格式如下：

BL{cond} label

指令编码格式：



signed_immed_24 24 位有符号立即数(偏移量)。

L 区别分支(L 为 0)或带连接的分支指令(L 为 1)。

带连接的分支指令 BL 举例如下：

```
BL      DELAY
```

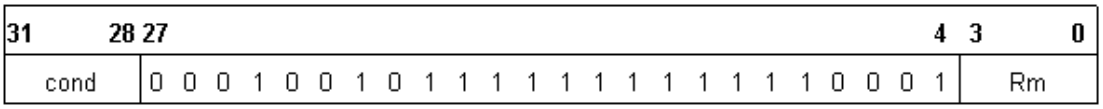
分支指令 BL 限制在当前指令的±32M 字节地址范围内。BL 指令用于子程序调用。

- **BX——带状态切换的分支指令**

BX 指令跳转到 Rm 指定的地址执行程序，若 Rm 的位[0]为 1，则跳转时自动将 CPSR 中的标志 T 置位，即把目标地址的代码解释为 Thumb 代码；若 Rm 的位[0]为 0，则跳转时自动将 CPSR 中的标志 T 复位，即把目标地址的代码解释为 ARM 代码。指令格式如下：

BX{cond}　　Rm

指令编码格式：



Rm　　目标地址寄存器。

带状态切换的分支指令 BX 举例如下：

```
ADRL    R0,ThumbFun+1
BX      R0              ; 跳转到 R0 指定的地址，并根据 R0 的最低位来切换处
                        ; 理器状态
```

7. ARM 协处理器指令

ARM 支持协处理器操作，协处理器的控制要通过协处理器命令实现。

ARM 协处理器指令见表 4.9。

表 4.9　ARM 协处理器指令

助记符	说明	操作	条件码位置
CDP　coproc,opcode1,CRd,CRn,CRm{,opcode2}	协处理器数据操作指令	取决于协处理器	CDP{cond}
LDC{L}　coproc, CRd,<地址>	协处理器数据读取指令	取决于协处理器	LDC{cond}{L}
STC{L}　coproc, CRd,<地址>	协处理器数据写入指令	取决于协处理器	STC{cond}{L}
MCR　coproc,opcode1,Rd,CRn,CRm{,opcode2}	ARM 寄存器到协处理器寄存器的数据传送指令	取决于协处理器	MCR{cond}
MRC　coproc,opcode1,Rd,CRn,CRm{,opcode2}	协处理器寄存器到 ARM 寄存器到的数据传送指令	取决于协处理器	MCR{cond}

- **CDP——协处理器数据操作指令**

ARM 处理器通过 CDP 指令通知 ARM 协处理器执行特定的操作。该操作由协处理器完成，即对命令的参数的解释与协处理器有关，指令的使用取决于协处理器。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。指令格式如下：

CDP{cond}　　coproc,opcode1,CRd,CRn,CRm{,opcode2}

其中：　coproc　指令操作的协处理器名。标准名为 pn，n 为 0～15。

opcode1　协处理器的特定操作码。

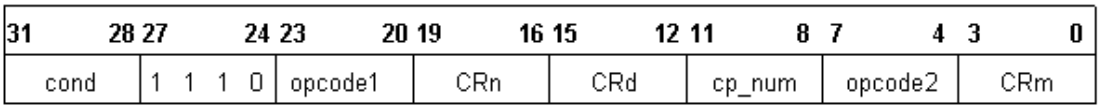
CRd　作为目标寄存的协处理器寄存器。

CRn　存放第 1 个操作数的协处理器寄存器。

CRm　存放第 2 个操作数的协处理器寄存器。

Opcode2 可选的协处理器特定操作码。

指令编码格式：



cp_num 为协处理器编号。

CDP 指令举例如下：

```
CDP     p7,0,c0,c2,c3,0     ; 协处理器 7 操作，操作码为 0，可选操作码为 0
CDP     p6,1,c3,c4,c5       ; 协处理器 6 操作，操作码为 1
```

- **LDC——协处理器数据读取指令**

LDC 指令从某一连续的内存单元将数据读取到协处理器的寄存器中。进行协处理器数据的传送，由协处理器来控制传送的字数。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。指令格式如下：

LDC{cond}{L} coproc, CRd,<地址>

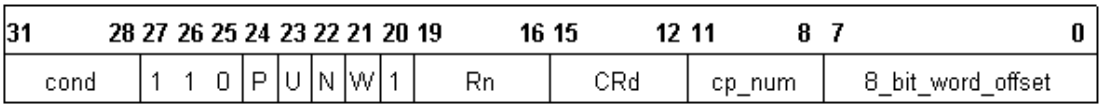
其中： L 可选后缀，指明是长整数传送。

coproc 指令操作的协处理器名。标准名为 pn，n 为 0～15。

CRd 作为目标寄存的协处理器寄存器。

<地址> 指定的内存地址。

指令编码格式：



cp_num 为协处理器编号。

8_bit_word_offset 8 位立即数偏移。

P，U，W 用于区别不同的地址模式。P 表示前/后变址，U 表示加/减，W 表示回写。

N 数据大小(依赖于协处理器)。

LDC 指令举例如下：

```
LDC     p5,c2,[R2,#4]     ; 读取 R2+4 指向的内存单元的数据，传送到协处理器
                          ; p5 的 c2 寄存器中
LDC     p6,c2,[R1]        ; 读取 R1 指向的内存单元的数据，传送到协处理器 p6
                          ; 的 c2 寄存器中
```

- **STC——协处理器数据写入指令**

STC 指令将协处理器的寄存器数据写入到某一连续的内存单元中。进行协处理器数据的数据传送，由协处理器来控制传送的字数。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。指令格式如下：

STC{cond}{L} coproc, CRd,<地址>

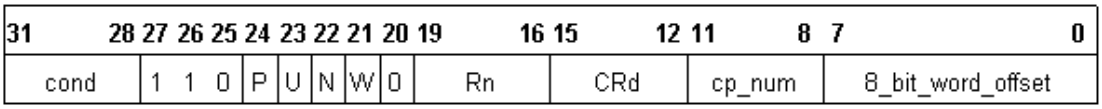
其中： L 可选后缀，指明是长整数传送。

coproc 指令操作的协处理器名。标准名为 pn，n 为 0～15。

CRd 作为目标寄存的协处理器寄存器。

<地址> 指定的内存地址。

指令编码格式：



cp_num	为协处理器编号。
8_bit_word_offset	8 位立即数偏移。
P，U，W	用于区别不同的地址模式。P 表示前/后变址，U 表示加/减，W 表示回写。
N	数据大小(依赖于协处理器)。

STC 指令举例如下：

```
STC     p5,c1,[R0]
STC     p5,c1,[R0,#-0x04]
```

- **MCR——ARM 寄存器到协处理器寄存器的数据传送指令**

MCR 指令将 ARM 处理器的寄存器中的数据传送到协处理器的寄存器中。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。指令格式如下：

MCR{cond} coproc,opcode1,Rd,CRn,CRm{,opcode2}

其中：

coproc 指令操作的协处理器名。标准名为 pn，n 为 0～15。

opcode1 协处理器的特定操作码。

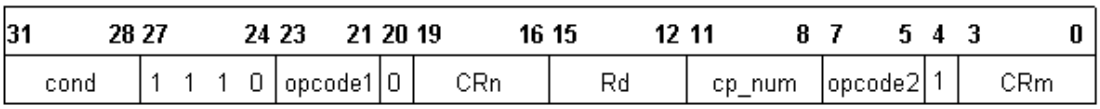
Rd 作为目标寄存的协处理器寄存器。

CRn 存放第 1 个操作数的协处理器寄存器。

CRm 存放第 2 个操作数的协处理器寄存器。

opcode2 可选的协处理器特定操作码。

指令编码格式：



cp_num	为协处理器编号。

MCR 指令举例如下：

```
MCR     p6,2,R7,c1,c2
MCR     p7,0,R1,c3,c2,1
```

- **MRC——协处理器寄存器到 ARM 寄存器到的数据传送指令**

MRC 指令将协处理器寄存器中的数据传送到 ARM 处理器的寄存器中。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。指令格式如下：

MRC{cond} coproc,opcode1,Rd,CRn,CRm{,opcode2}

其中：

coproc 指令操作的协处理器名。标准名为 pn，n 为 0～15。

opcode1 协处理器的特定操作码。

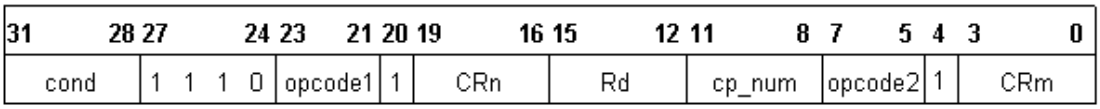
Rd 作为目标寄存的协处理器寄存器。

CRn 存放第 1 个操作数的协处理器寄存器。

CRm 存放第 2 个操作数的协处理器寄存器。

Opcode2 可选的协处理器特定操作码。

指令编码格式：



31 28	27 24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
cond	1 1 1 0	opcode1	1	CRn	Rd	cp_num	opcode2	1	CRm

cp_num 为协处理器编号。

MRC 指令举例如下：

```
MRC     p5,2,R2,c3,c2
MRC     p7,0,R0,c1,c2,1
```

8. ARM 杂项指令

ARM 杂项指令见表 4.10。

表 4.10 ARM 杂项指令

助记符	说明	操作	条件码位置
SWI immed_24	软中断指令	产生软中断，处理器进入管理模式	SWI{cond}
MRS Rd,psr	读状态寄存器指令	Rd←psr，psr 为 CPSR 或 SPSR	MRS{cond}
MSR psr_fields,Rd/#immed_8r	写状态寄存器指令	psr_fields←Rd/#immed_8r，psr 为 CPSR 或 SPSR	MSR{cond}

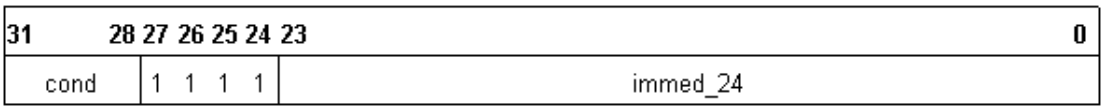
- **SWI——软中断指令**

SWI 指令用于产生软中断，从而实现在从户模式变换到管理模式，CPSR 保存到管理模式的 SPSR 中，执行转移到 SWI 向量。在其它模式下也可使用 SWI 指令，处理器同样地切换到管理模式。指令格式如下：

SWI{cond} immed_24

其中： immed_24 24 位立即数，值为 0～16777215 之间的整数。

指令编码格式：



31 28	27 26 25 24	23 0
cond	1 1 1 1	immed_24

SWI 指令举例如下：

```
SWI     0               ; 软中断，中断立即数为 0
SWI     0x123456        ; 软中断，中断立即数为 0x123456
```

使用 SWI 指令时，通常使用以下两种方法进行传递参数，SWI 异常中断处理程序就可以提供相关的服务，这两种方法均是用户软件协定。SWI 异常中断处理程序要通过读取引起软中断的 SWI 指令，以取得 24 位立即数。

1. 指令中 24 位的立即数指定了用户请求的服务类型，参数通过通用寄存器传递。

```
MOV     R0,#34          ; 设置子功能号为 34
SWI     12              ; 调用 12 号软中断
```

2. 指令中的 24 位立即数被忽略，用户请求的服务类型由寄存器 R0 的值决定，参数通过其它的通用寄存器传递。

```
MOV     R0,#12          ; 调用 12 号软中断
MOV     R1,#34          ; 设置子功能号为 34
SWI     0
```

在 SWI 异常中断处理程序中，取出 SWI 立即数的步骤为：首先确定引起软中断的 SWI 指令是 ARM 指令还时 Thumb 指令，这可通过对 SPSR 访问得到；然后要取得该 SWI 指令的地址，这可通过访问 LR 寄存器得到；接着读出指令，分解出立即数。如程序清单 4.2 所示。

程序清单 4.2 读取 SWI 立即数

```
T_bit       EQU     0x20
SWI_Handler
            STMFD   SP!, {R0-R3, R12, LR}     ; 现场保护
            MRS     R0, SPSR                  ; 读取 SPSR
            STMFD   SP!, {R0}                 ; 保存 SPSR
            TST     R0, #T_bit                ; 测试 T 标志位
            LDRNEH  R0, [LR,#-2]              ; 若是 Thumb 指令，读取指令码(16 位)
            BICNE   R0, R0, #0xFF00           ; 取得 Thumb 指令的 8 位立即数
            LDREQ   R0, [LR,#-4]              ; 若是 ARM 指令，读取指令码(32 位)
            BICEQ   R0, R0, #0xFF000000       ; 取得 ARM 指令的 24 位立即数
            …
            LDMFD    SP!, {R0-R3, R12, PC}^   ; SWI 异常中断返回
```

- **MRS——读状态寄存器指令**

在 ARM 处理器中，只有 MRS 指令可以将状态寄存器 CPSR 或 SPSR 读出到通用寄存器中。指令格式如下：

```
MRS{cond}       Rd,psr
```

其中： Rd 目标寄存器。Rd 不允许为 R15。

psr CPSR 或 SPSR。

指令编码格式：

31 28	27 23	22	21 20	19 16	15 12	11 0
cond	0 0 0 1 0	R	0 0	1 1 1 1	Rd	0 0 0 0 0 0 0 0 0 0 0 0

R 用于区别 CPSR(R 为 0)或 SPSR(R 为 1)。

MRS 指令举例如下：

```
MRS     R1,CPSR         ; 将 CPSR 状态寄存器读取，保存到 R1 中
MRS     R2,SPSR         ; 将 SPSR 状态寄存器读取，保存到 R2 中
```

MRS 指令读取 CPSR，可用来判断 ALU 的状态标志，或 IRQ、FIQ 中断是否允许等。在异常处理程序中，读 SPSR 可知道进行异常前的处理器状态等。MRS 与 MSR 配合使用，实现 CPSR 或 SPSR 寄存器的读－修改－写操作，可用来进行处理器模式切换、允许/禁止 IRQ/FIQ 中断等设置，如程序清单 4.3、程序清单 4.4 所示。另外，进程切换或允许异常中断嵌套时，也需要使用 MRS 指令读取 SPSR 状态值，并保存起来。

程序清单 4.3 使能 IRQ 中断

```
ENABLE_IRQ
        MRS     R0, CPSR
        BIC     R0, R0, #0x80
        MSR     CPSR_c, R0
        MOV     PC,LR
```

程序清单 4.4 禁能 IRQ 中断

```
DISABLE_IRQ
        MRS     R0 CPSR
        ORR     R0, R0, #0x80
        MSR     CPSR_c, R0
        MOV     PC,LR
```

- **MSR——写状态寄存器指令**

在 ARM 处理器中，只有 MSR 指令可以直接设置状态寄存器 CPSR 或 SPSR。指令格式如下：

MSR{cond}　　psr_fields,#immed_8r
MSR{cond}　　psr_fields,Rm
其中：psr　　CPSR 或 SPSR。
　　fields　　指定传送的区域。fields 可以是以下的一种或多种(字母必须为小写)：
　　c　控制域屏蔽字节(psr[7…0])；
　　x　扩展域屏蔽字节(psr[15…8])；
　　s　状态域屏蔽字节(psr[23…16])；
　　f　标志域屏蔽字节(psr[31…24])。
　　immed_8r　要传送到状态寄存器指定域的立即数，8 位。
　　Rm　　要传送到状态寄存器指定域的数据的源寄存器。

指令编码格式(操作数为立即数)：

31 28	27 23	22	21 20	19 16	15 12	11 8	7 0
cond	0 0 1 1 0	R	1 0	field_mask	1 1 1 1	rotate_imm	8_bit_immediate

指令编码格式(操作数为寄存器)：

31 28	27 23	22	21 20	19 16	15 12	11 4	3 0
cond	0 0 0 1 0	R	1 0	field_mask	1 1 1 1	0 0 0 0 0 0 0 0	Rm

R　　用于区别 CPSR(R 为 0)或 SPSR(R 为 1)。
field_mask　　域屏蔽。
rotate_imm　　立即数对齐。
8_bit_immediate　　8 位立即数。
Rm　　操作数寄存器。

MSR 指令举例如下：

```
MSR     CPSR_c,#0xD3        ; CPSR[7…0]=0xD3，即切换到管理模式
MSR     CPSR_cxsf,R3        ; CPSR=R3
```

只有在特权模式下才能修改状态寄存器。

程序中不能通过 MSR 指令直接修改 CPSR 中的 T 控制位来实现 ARM 状态/Thumb 状态的切换，必须使用 BX 指令完成处理器状态的切换(因为 BX 指令属分支指令，它会打断流水线状态，实现处理器状态切换)。MRS 与 MSR 配合使用，实现 CPSR 或 SPSR 寄存器的读－修改－写操作，可用来进行处理器模式切换、允许/禁止 IRQ/FIQ 中断等设置，如程序清单 4.5 所示。

程序清单 4.5 堆栈指令初始化

```
INITSTACK
        MOV     R0, LR          ; 保存返回地址
; 设置管理模式堆栈
        MSR     CPSR_c, #0xD3
        LDR     SP, StackSvc
; 设置中断模式堆栈
        MSR     CPSR_c, #0xD2
        LDR     SP, StackIrq
        …
```

9. ARM 伪指令

ARM 伪指令不是 ARM 指令集中的指令，只是为了编程方便编译器定义了伪指令，使用时可以像其它 ARM 指令一样使用，但在编译时这些指令将被等效的 ARM 指令代替。ARM 伪指令有四条，分别为 ADR 伪指令、ADRL 伪指令、LDR 伪指令、NOP 伪指令。

- **ADR——小范围的地址读取伪指令**

ADR 指令将基于 PC 相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中。在汇编编译源程序时，ADR 伪指令被编译器替换成一条合适的指令。通常，编译器用一条 ADD 指令或 SUB 指令来实现该 ADR 伪指令的功能，若不能用一条指令实现，则产生错误，编译失败。ADR 伪指令格式如下：

ADR{cond} register,expr

其中：
- register 加载的目标寄存器。
- expr 地址表达式。当地址值是非字对齐时，取值范围-255～255 字节之间；当地址值是字对齐时，取值范围-1020～1020 字节之间。对于基于 PC 相对偏移的地址值时，给定范围是相对当前指令地址后两个字处(因为 ARM7TDMI 为三级流水线)。

ADR 伪指令举例如下：

```
LOOP    MOV     R1,#0xF0
        …
        ADR     R2,LOOP         ; 将 LOOP 的地址放入 R2
        ADR     R3,LOOP+4
```

可以用 ADR 加载地址，实现查表，如程序清单 4.6 所示。

程序清单 4.6 小范围地址的加载

```
        …
        ADR     R0,DISP_TAB             ; 加载转换表地址
        LDRB    R1,[R0,R2]              ; 使用 R2 作为参数，进行查表
        …
DISP_TAB
        DCB     0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8, 0x80, 0x90
```

- **ADRL——中等范围的地址读取伪指令**

ADRL 指令将基于 PC 相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中，比 ADR 伪指令可以读取更大范围的地址。在汇编编译源程序时，ADRL 伪指令被编译器替换成两条合适的指令。若不能用两条指令实现 ADRL 伪指令功能，则产生错误，编译失败。ADRL 伪指令格式如下：

ADRL{cond} register,expr

其中： register 加载的目标寄存器。

Expr 地址表达式。当地址值是非字对齐时，取值范围-64KB～64KB 之间；当地址值是字对齐时，取值范围-256KB～256KB 之间。

ADRL 伪指令举例如下：

```
        ADRL    R0,DATA_BUF
        …
        ADRL    R1,DATA_BUF+80
        …
DATA_BUF
        SPACE   100             ; 定义 100 字节缓冲区
```

可以用 ADRL 加载地址，实现程序跳转，如程序清单 4.7 所示。

程序清单 4.7 中等范围地址的加载

```
        …
        ADR     LR,RETURN1              ; 设置返回地址
        ADRL    R1,Thumb_Sub+1          ; 取得 Thumb 子程序入口地址，且 R1 的 0 位置 1
        BX      R1                      ; 调用 Thumb 子程序，并切换处理器状态
RETURN1
        …
        CODE    16
Thumb_Sub
        MOV     R1,#10
        …
```

- **LDR——大范围的地址读取伪指令**

LDR 伪指令用于加载 32 位的立即数或一个地址值到指定寄存器。在汇编编译源程序时，LDR 伪指令被编译器替换成一条合适的指令。若加载的常数未超出 MOV 或 MVN 的范围，则使用 MOV 或 MVN 指令代替该 LDR 伪指令，否则汇编器将常量放入文字池，并使用一

条程序相对偏移的 LDR 指令从文字池读出常量。LDR 伪指令格式如下：

LDR{cond} register,=expr/label-expr

其中： register 加载的目标寄存器。
expr 32 位立即数。
label-expr 基于 PC 的地址表达式或外部表达式。

LADR 伪指令举例如下：

```
LDR     R0,=0x12345678      ; 加载 32 位立即数 0x12345678
LDR     R0,=DATA_BUF+60     ; 加载 DATA_BUF 地址+60
…
LTORG                       ; 声明文字池
…
```

伪指令 LDR 常用于加载芯片外围功能部件的寄存器地址(32 位立即数)，以实现各种控制操作，如程序清单 4.8 所示。

程序清单 4.8 加载 32 位立即数

```
…
LDR     R0,=IOPIN           ; 加载 GPIO 的寄存器 IOPIN 的地址
LDR     R1,[R0]             ; 读取 IOPIN 寄存器的值
…
LDR     R0,=IOSET
LDR     R1,=0x00500500
STR     R1,[R0]             ; IOSET=0x00500500
…
```

从 PC 到文字池的偏移量必须小于 4KB。
与 ARM 指令的 LDR 相比，伪指令的 LDR 的参数有“=”号。

- **NOP——空操作伪指令**

NOP 伪指令在汇编时将会被代替成 ARM 中的空操作，比如可能为“MOV R0,R0”指令等。NOP 伪指令格式如下：

NOP

NOP 可用于延时操作，如程序清单 4.9 所示。

程序清单 4.9 软件延时

```
        …
DELAY1
        NOP
        NOP
        NOP
        SUBS    R1,R1,#1
        BNE     DELAY1
```

```
…
```

4.2.2 Thumb 指令集

Thumb指令集可以看作是ARM指令压缩形式的子集，是针对代码密度的问题而提出的，它具有 16 位的代码密度。Thumb 不是一个完整的体系结构，不能指望处理器只执行 Thumb 指令而不支持 ARM 指令集。因此，Thumb 指令只需要支持通用功能，必要时可以借助于完善的 ARM 指令集，比如，所有异常自动进入 ARM 状态。

在编写 Thumb 指令时，先要使用伪指令 CODE16 声明，而且在 ARM 指令中要使用 BX 指令跳转到 Thumb 指令，以切换处理器状态。编写 ARM 指令时，则可使用伪指令 CODE32 声明。由 ARM 状态切换到 Thumb 状态的代码，如程序清单 4.10 所示。

程序清单 4.10 ARM 到 Thumb 的状态切换

```
; 文件名：TEST8.S
; 功能：使用 BX 指令切换处理器状态
; 说明：使用 ARMulate 软件仿真调试
                AREA    Example8,CODE,READONLY
                ENTRY
                CODE32
ARM_CODE        ADR     R0,THUMB_CODE+1
                BX      R0                      ; 跳转并切换处理器状态

                CODE16
THUMB_CODE
                MOV     R0,#10                  ; R0 = 10
                MOV     R1,#20                  ; R1 = 20
                ADD     R0,R1                   ; R0 = R0+R1
                B       .
                END
```

程序首先在 ARM 状态下使用“ADR R0,THUMB_CODE+1 ”伪指令装载 THUMB_CODE 的地址，为了使 R0 的位[0]为 1，所以使用了“THUMB_CODE+1”，这样使用 BX 即可把处理器状态切换到 Thumb 状态。

1. Thumb 指令集与 ARM 指令集的区别

Thumb 指令集没有协处理器指令、信号量指令以及访问 CPSR 或 SPSR 的指令，没有乘加指令及 64 位乘法指令等，且指令的第二操作数受到限制；除了分支指令 B 有条件执行功能外，其它指令均为无条件执行；大多数 Thumb 数据处理指令采用 2 地址格式。Thumb 指令集与 ARM 指令集的区别一般有如下几点：

- 分支指令

程序相对转移，特别是条件跳转与 ARM 代码下的跳转相比，在范围上有更多的限制，转向子程序是无条件的转移。

- 数据处理指令

数据处理指令是对通用寄存器进行操作，在大多数情况下，操作的结果须放入其中一个操作数寄存器中，而不是第 3 个寄存器中。

数据处理操作比 ARM 状态的更少。

访问寄存器 R8～R15 受到一定限制。

除 MOV 和 ADD 指令访问寄存器 R8～R15 外，其它数据处理指令总是更新 CPSR 中的 ALU 状态标志。

访问寄存器 R8～R15 的 Thumb 数据处理指令不能更新 CPSR 中的 ALU 状态标志。

- **单寄存器加载和存储指令**

在 Thumb 状态下，单寄存器加载和存储指令只能访问寄存器 R0～R7。

- **多寄存器加载和多寄存器存储指令**

LDM 和 STM 指令可以将任何范围为 R0～R7 的寄存器子集加载或存储。

PUSH 和 POP 指令使用堆栈指令 R13 作为基址实现满递减堆栈。除 R0～R7 外，PUSH 指令还可以存储连接寄存器 R14，并且 POP 指令可以加载程序指令 PC。

2. Thumb 存储器访问指令

Thumb 指令集的 LDM 和 STM 指令可以将任何范围为 R0～R7 的寄存器子集加载或存储。多寄存器加载和多寄存器存储指令只有 LDMIA、STMIA 指令，即每次传送先加载/存储数据，然后地址加 4。对堆栈处理只能使用 PUSH 及 POP 指令。

Thumb 存储器访问指令见表 4.11。

表 4.11 Thumb 存储器访问指令

助记符	说明	操作	影响标志
LDR Rd,[Rn,#immed_5×4]	加载字数据	Rd←[Rn,#immed_5×4]，Rd、Rn 为 R0～R7	无
LDRH Rd,[Rn,#immed_5×2]	加载无符号半字数据	Rd←[Rn,#immed_5×2]，Rd、Rn 为 R0～R7	无
LDRB Rd,[Rn,#immed_5×1]	加载无符号字节数据	Rd←[Rn,#immed_5×1]，Rd、Rn 为 R0～R7	无
STR Rd,[Rn,#immed_5×4]	存储字数据	[Rn,#immed_5×4]←Rd，Rd、Rn 为 R0～R7	无
STRH Rd,[Rn,#immed_5×2]	存储无符号半字数据	[Rn,#immed_5×2]←Rd，Rd、Rn 为 R0～R7	无
STRB Rd,[Rn,#immed_5×1]	存储无符号字节数据	[Rn,#immed_5×1]←Rd，Rd、Rn 为 R0～R7	无
LDR Rd,[Rn,Rm]	加载字数据	Rd←[Rn,Rm]，Rd、Rn、Rm 为 R0～R7	无
LDRH Rd,[Rn,Rm]	加载无符号半字数据	Rd←[Rn,Rm]，Rd、Rn、Rm 为 R0～R7	无
LDRB Rd,[Rn,Rm]	加载无符号字节数据	Rd←[Rn,Rm]，Rd、Rn、Rm 为 R0～R7	无
LDRSH Rd,[Rn,Rm]	加载有符号半字数据	Rd←[Rn,Rm]，Rd、Rn、Rm 为 R0～R7	无
LDRSB Rd,[Rn,Rm]	加载有符号字节数据	Rd←[Rn,Rm]，Rd、Rn、Rm 为 R0～R7	无
STR Rd,[Rn,Rm]	存储字数据	[Rn,Rm]←Rd，Rd、Rn、Rm 为 R0～R7	无
STRH Rd,[Rn,Rm]	存储无符号半字数据	[Rn,Rm]←Rd，Rd、Rn、Rm 为 R0～R7	无
STRB Rd,[Rn,Rm]	存储无符号字节数据	[Rn,Rm]←Rd，Rd、Rn、Rm 为 R0～R7	无
LDR Rd,[PC,#immed_8×4]	基于 PC 加载字数据	Rd←[PC,#immed_8×4]，Rd 为 R0～R7	无
LDR Rd,label	基于 PC 加载字数据	Rd←[label]，Rd 为 R0～R7	无
LDR Rd,[SP,#immed_8×4]	基于 SP 加载字数据	Rd←[SP,#immed_8×4]，Rd 为 R0～R7	无
STR Rd,[SP,#immed_8×4]	基于 SP 存储字数据	[SP,#immed_8×4]←Rd，Rd 为 R0～R7	无
LDMIA Rn{!},reglist	多寄存器加载	reglist←[Rn…]，Rn 回写等 (R0～R7)	无
STMIA Rn{!},reglist	多寄存器存储	[Rn…]←reglist，Rn 回写等 (R0～R7)	无
PUSH {reglist[,LR]}	寄存器入栈指令	[SP…]←reglist[,LR]，SP 回写等 (R0～R7、LR)	无
POP {reglist[,PC]}	寄存器出栈指令	reglist[,PC]←[SP…] ，SP 回写等 (R0～R7、PC)	无

- **LDR 和 STR——加载/存储指令**

立即数偏移的 LDR 和 STR 指令。

存储器的地址以一个寄存器的立即数偏移指明。指令格式如下：

```
LDR     Rd,[Rn,#immed_5×4]  ; 加载指定地址上的数据(字)，放入 Rd 中
STR     Rd,[Rn,#immed_5×4]  ; 存储数据(字)到指定地址的存储单元，要存储的数据在 Rd 中
LDRH    Rd,[Rn,#immed_5×2]  ; 加载半字数据，放入 Rd 中，即 Rd 最低 16 位有效，高 16 位清零
STRH    Rd,[Rn,#immed_5×2]  ; 存储半字数据，要存储的数据在 Rd，最低 16 位有效
LDRB    Rd,[Rn,#immed_5×1]  ; 加载字节数据，放入 Rd 中，即 Rd 最低字节有效，高 24 位清零
STRB    Rd,[Rn,#immed_5×1]  ; 存储字节数据，要存储的数据在 Rd，最低字节有效
```

其中： Rd 加载或存储的寄存器。必须为 R0～R7。

Rn 基址寄存器。必须为 R0～R7。

immed_5×N 偏移量。它是一个无符号立即数表达式，其取值为(0～31)×N。

立即数偏移的半字和字节加载是无符号的。数据加载到 Rd 的最低有效半字或字节，Rd 的其余位补 0。

指令编码格式(LDR/STR Rd,[Rn,#immed_5×4])：

15			12	11	10 6	5 3	2 0
0	1	1	0	L	immed_5	Rn	Rd

指令编码格式(LDRH/STRH Rd,[Rn,#immed_5×2])：

15			12	11	10 6	5 3	2 0
1	0	0	0	L	immed_5	Rn	Rd

指令编码格式(LDRB/STRB Rd,[Rn,#immed_5×1])：

15	14	13	12	11	10 6	5 3	2 0
0	1	1	1	L	immed_5	Rn	Rd

L 用于区别加载(L 为 1)或存储(L 为 0)。

immed_5 5 位无符号立即数偏移。

地址对齐——字传送时，必须保证传送地址为 32 位对齐。半字传送时，必须保证传送地址为 16 位对齐。

立即数偏移的 LDR 和 STR 指令举例如下：

```
LDR     R0,[R1,#0x4]
STR     R3,[R4]
LDRH    R5,[R0,#0x02]
STRH    R1,[R0,#0x08]
LDRB    R3,[R6,#20]
STRB    R1,[R0,#31]
```

寄存器偏移的 LDR 和 STR 指令。存储器的地址以一个寄存器的寄存器偏移来指明。指令格式如下：

```
LDR     Rd,[Rn,Rm]      ; 加载一个字数据
STR     Rd,[Rn,Rm]      ; 存储一个字数据
LDRH    Rd,[Rn,Rm]      ; 加载一个无符号半字数据
```

```
STRH    Rd,[Rn,Rm]      ; 存储一个无符号半字数据
LDRB    Rd,[Rn,Rm]      ; 加载一个无符号字节数据
STRB    Rd,[Rn,Rm]      ; 存储一个无符号字节数据
LDRSH   Rd,[Rn,Rm]      ; 加载一个有符号半字数据
LDRSB   Rd,[Rn,Rm]      ; 存储一个有符号半字数据
```

其中： Rd 加载或存储的寄存器。必须为 R0～R7。

Rn 基址寄存器。必须为 R0～R7。

Rm 内含偏移量的寄存器。必须为 R0～R7。

寄存器偏移的半字和字节加载可以是有符号或无符号的，数据加载到 Rd 的最低有效半字或字节。对于无符号半字或字节加载，Rd 的其余位补 0；对于有符号半字或字节加载，Rd 的其余位拷贝符号位。

指令编码格式(LDR/STR Rd,[Rn,Rm])：

15			12	11	10	9	8	6	5	3	2	0
0	1	0	1	L	0	0	Rm		Rn		Rd	

指令编码格式(LDRH/STRH Rd,[Rn,Rm])：

15			12	11	10	9	8	6	5	3	2	0
0	1	0	1	L	0	1	Rm		Rn		Rd	

指令编码格式(LDRB/STRB Rd,[Rn,Rm])：

15	14	13	12	11	10	9	8	6	5	3	2	0
0	1	0	1	L	1	0	Rm		Rn		Rd	

指令编码格式(LDRSH Rd,[Rn,Rm])：

15	14	13	12	11	10	9	8	6	5	3	2	0
0	1	0	1	1	1	1	Rm		Rn		Rd	

指令编码格式(LDRSB Rd,[Rn,Rm])：

15	14	13	12	11	10	9	8	6	5	3	2	0
0	1	0	1	0	1	1	Rm		Rn		Rd	

L 用于区别加载(L 为 1)或存储(L 为 0)。

地址对齐——字传送时，必须保证传送地址为 32 位对齐。半字传送时，必须保证传送地址为 16 位对齐。

寄存器偏移的 LDR 和 STR 指令举例如下：

```
LDR     R3,[R1,R0]
STR     R1,[R0,R2]
LDRH    R6,[R0,R1]
STRH    R0,[R4,R5]
LDRB    R2,[R5,R1]
STRB    R1,[R3,R2]
LDRSH   R7,[R6,R3]
LDRSB   R5,[R7,R2]
```

PC 或 SP 相对偏移的 LDR 和 STR 指令。用 PC 或 SP 寄存器中的值的立即数偏移来指明存储器中的地址。指令格式如下：

```
LDR     Rd,[PC,#immed_8×4]
LDR     Rd,label
LDR     Rd,[SP,#immed_8×4]
STR     Rd,[SP,#immed_8×4]
```

其中： Rd 加载或存储的寄存器。必须为 R0～R7。
immed_8×4 偏移量。它是一个无符号立即数表达式，其取值为(0～255)×4。
label 程序相对偏移表达式。label 必须在当前指令之后 1KB 范围内。

指令编码格式(PC 相对偏移 LDR 指令)：

15			12	11	10	8	7	0
0	1	0	0	1	Rd		immed_8	

指令编码格式(SP 相对偏移 LDR/STR 指令)：

15			12	11	10	8	7	0
1	0	0	1	L	Rd		immed_8	

L 用于区别加载(L 为 1)或存储(L 为 0)。
immed_8 8 位无符号立即数偏移。

地址对齐——地址必须是 4 的整数倍。

PC 或 SP 相对偏移的 LDR 和 STR 指令举例如下：

```
LDR     R0,[PC,#0x08]    ; 读取 PC+0x08 地址上的字数据，保存到 R0 中
LDR     R7,LOCALDAT      ; 读取 LOCALDAT 地址上的字数据，保存到 R7 中
LDR     R3,[SP,#1020]    ; 读取 SP+1020 地址上的字数据，保存到 R3 中
STR     R2,[SP]          ; 存储 R2 寄存器的数据到 SP 指向的存储单元 (偏移量为 0)
```

- **PUSH 和 POP——寄存器入栈及出栈指令**

实现低寄存器和可选的 LR 寄存器入栈及低寄存器和可选的 PC 寄存器出栈操作。堆栈地址由 SP 寄存器设置，堆栈是满递减堆栈。指令格式如下：

```
PUSH    {reglist[,LR]}
POP     {reglist[,PC]}
```

其中： reglist 入栈/出栈低寄存器列表，即 R0～R7。
LR 入栈时的可选寄存器。
PC 出栈时的可选寄存器。

指令编码格式：

15			12	11	10	9	8	7	0
1	0	1	1	L	1	0	R	register_list	

L 用于区别出栈(L 为 1)或入栈(L 为 0)。
R 区别操作寄存器中是否有 LR/PC(有，则 R 为 1，否则为 0)。

寄存器入栈及出栈指令举例如下：

```
PUSH    {R0-R7,LR}      ; 将低寄存器 R0～R7 全部入栈，LR 也入栈
POP     {R0-R7,PC}      ; 将堆栈中的数据弹出到低寄存器 R0～R7 及 PC 中
```

- **LDMIA 和 STMIA——多寄存器加载/存储指令**

可以实现在一组寄存器和一块连续的内存单元之间传输数据。Thumb 指令集多寄存器加载/存储指令为 LDMIA 和 STMIA，LDMIA 为加载多个寄存器；STM 为存储多个寄存器。允许一条指令传送 8 个低寄存器 R0～R7 的任何子集。指令格式如下：

```
LDMIA   Rn!,reglist
STMIA   Rn!,reglist
```

其中： Rn 加载/存储的起始地址寄存器。Rn 必须为 R0～R7。

reglist 加载/存储的寄存器列表。寄存器必须为 R0～R7。

指令编码格式：

15			12	11	10	8	7	0
1	1	0	0	L	Rn		register_list	

L 用于区别出栈(L 为 1)或入栈(L 为 0)。

LDMIA/STMIA 的主要用途是数据复制、参数传送等。进行数据传送时，每次传送后地址加 4。若 Rn 在寄存器列表中，对于 LDMIA 指令，Rn 的最终值是加载的值，而不是增加后的地址；对于 STMIA 指令，若 Rn 是寄存器列表中的最低数字的寄存器，则 Rn 存储的值为 Rn 在初值，其它情况不可预知。

多寄存器加载/存储指令举例如下：

```
LDMIA   R0!,{R2-R7}  ; 加载 R0 指向的地址上的多字数据，保存到 R2～R7 中，
                     ; R0 的值更新。
STMIA   R1!,{R2-R7}  ; 将 R2～R7 的数据存储到 R1 指向的地址上，R1 值更新
```

3. Thumb 数据处理指令

大多数 Thumb 数据处理指令采用 2 地址格式，数据处理操作比 ARM 状态的更少，访问寄存器 R8～R15 受到一定限制。

Thumb 数据处理指令见表 4.12。

表 4.12 Thumb 数据处理指令

助记符	说明	操作	影响标志
MOV Rd,#expr	数据传送指令	Rd←expr，Rd 为 R0～R7	影响 N、Z
MOV Rd,Rm	数据传送指令	Rd←Rm，Rd、Rm 均可为 R0～R15	Rd 和 Rm 均为 R0～R7 时，影响 N、Z，清零 C、V
MVN Rd,Rm	数据非传送指令	Rd←(~Rm) ，Rd、Rm 均为 R0～R7	影响 N、Z
NEG Rd,Rm	数据取负指令	Rd←(-Rm) ，Rd、Rm 均为 R0～R7	影响 N、Z、C、V
ADD Rd,Rn,Rm	加法运算指令	Rd←Rn+Rm，Rd、Rn、Rm 均为 R0～R7	影响 N、Z、C、V
ADD Rd,Rn,#expr3	加法运算指令	Rd←Rn+expr3，Rd、Rn 均为 R0～R7	影响 N、Z、C、V
ADD Rd,#expr8	加法运算指令	Rd←Rd+expr8，Rd 为 R0～R7	影响 N、Z、C、V
ADD Rd, Rm	加法运算指令	Rd←Rd+Rm，Rd、Rm 均可为 R0～R15	Rd 和 Rm 均为 R0～R7 时，影响 N、Z、C、V
ADD Rd,Rp,#expr	SP/PC 加法运算指令	Rd←SP+expr 或 PC+expr，Rd 为 R0～R7	无

接上表

助记符	说明	操作	影响标志
ADD SP,#expr	SP 加法运算指令	SP←SP+expr	无
SUB Rd,Rn,Rm	减法运算指令	Rd←Rn-Rm，Rd、Rn、Rm 均为 R0～R7	影响 N、Z、C、V
SUB Rd,Rn,#expr3	减法运算指令	Rd←Rn-expr3，Rd、Rn 均为 R0～R7	影响 N、Z、C、V
SUB Rd,#expr8	减法运算指令	Rd←Rd-expr8，Rd 为 R0～R7	影响 N、Z、C、V
SUB SP,#expr	SP 减法运算指令	SP←SP-expr	无
ADC Rd,Rm	带进位加法指令	Rd←Rd+Rm+ Carry，Rd、Rm 为 R0～R7	影响 N、Z、C、V
SBC Rd,Rm	带进位减法指令	Rd←Rd-Rm-(NOT)Carry，Rd、Rm 为 R0～R7	影响 N、Z、C、V
MUL Rd,Rm	乘法运算指令	Rd←Rd*Rm，Rd、Rm 为 R0～R7	影响 N、Z
AND Rd,Rm	逻辑与操作指令	Rd←Rd & Rm，Rd、Rm 为 R0～R7	影响 N、Z
ORR Rd,Rm	逻辑或操作指令	Rd←Rd | Rm，Rd、Rm 为 R0～R7	影响 N、Z
EOR Rd,Rm	逻辑异或操作指令	Rd←Rd ^ Rm，Rd、Rm 为 R0～R7	影响 N、Z
BIC Rd,Rm	位清除指令	Rd←Rd & (~Rm)，Rd、Rm 为 R0～R7	影响 N、Z
ASR Rd,Rs	算术右移指令	Rd←Rd 算术右移 Rs 位，Rd、Rs 为 R0～R7	影响 N、Z、C
ASR Rd,Rm,#expr	算术右移指令	Rd←Rm 算术右移 expr 位，Rd、Rm 为 R0～R7	影响 N、Z、C
LSL Rd,Rs	逻辑左移指令	Rd←Rd << Rs，Rd、Rs 为 R0～R7	影响 N、Z、C
LSL Rd,Rm,#expr	逻辑左移指令	Rd←Rm << expr，Rd、Rm 为 R0～R7	影响 N、Z、C
LSR Rd,Rs	逻辑右移指令	Rd←Rd >> Rs，Rd、Rs 为 R0～R7	影响 N、Z、C
LSR Rd,Rm,#expr	逻辑右移指令	Rd←Rm >> expr，Rd、Rm 为 R0～R7	影响 N、Z、C
ROR Rd,Rs	循环右移指令	Rd←Rm 循环右移 Rs 位，Rd、Rs 为 R0～R7	影响 N、Z、C
CMP Rn,Rm	比较指令	状态标志←Rn-Rm，Rn、Rm 均可为 R0～R15	影响 N、Z、C、V
CMP Rn,#expr	比较指令	状态标志←Rn-expr，Rn 为 R0～R7	影响 N、Z、C、V
CMN Rn,Rm	负数比较指令	状态标志←Rn+Rm，Rn、Rm 为 R0～R7	影响 N、Z、C、V
TST Rn,Rm	位测试指令	状态标志←Rn & Rm，Rn、Rm 为 R0～R7	影响 N、Z、C、V

数据传送指令

- **MOV——数据传送指令**

MOV 指令将 8 位立即数或寄存器(operand2)传送到目标寄存器(Rd)。指令格式如下：

MOV Rd,#expr

MOV Rd,Rm

其中： Rd 目标寄存器。MOV Rd,#expr 时，Rd 必须在 R0～R7 之间。

exper 8 位立即数，即 0～255。

Rm 源寄存器。为 R0～R15。

指令编码格式(立即数转送)：

15				11	10		8	7							0
0	0	1	0	0	Rd			immed_8							

指令编码格式(寄存器转送)：

15						9	8		6	5		3	2		0
0	0	0	1	1	1	0	0	0	0	Rn			Rd		

条件码标志：

MOV Rd,#expr 指令会更新 N 和 Z 标志，对标志 C 和 V 无影响。而 MOV Rd,Rm 指令，若 Rd 或 Rm 是高寄存器(R8～R15)，则标志不受影响，若 Rd 或 Rm 都是低寄存器(R0～R7)，则会更新标志 N 和 Z，且清除标志 C 和 V。

MOV 指令举例如下：

```
MOV         R1,#0x10      ; R1=0x10
MOV         R0,R8         ; R0=R8
MOV         PC,LR         ; PC=LR，子程序返回
```

- **MVN——数据非传送指令**

MVN 指令将寄存器 Rm 按位取反后传送到目标寄存器(Rd)。指令格式如下：

MVN Rd,Rm

其中： Rd 目标寄存器。必须在 R0～R7 之间。

Rm 源寄存器。必须在 R0～R7 之间。

指令编码格式：

15					10	9			6	5	3	2	0
0	1	0	0	0	0	1	1	1	1	Rm		Rd	

条件码标志：

指令会更新 N 和 Z 标志，对标志 C 和 V 无影响。

MVN 指令举例如下：

```
MVN         R1,R2         ; 将 R2 取反，结果存到 R1
```

- **NEG——数据取负指令**

NEG 指令将寄存器 Rm 乘以-1 后传送到目标寄存器(Rd)。指令格式如下：

NEG Rd,Rm

其中： Rd 目标寄存器。必须在 R0～R7 之间。

Rm 源寄存器。必须在 R0～R7 之间。

指令编码格式：

15									6	5	3	2	0
0	1	0	0	0	0	1	0	0	1	Rm		Rd	

条件码标志：

指令会更新 N、Z、C 和 V 标志。

NEG 指令举例如下：

```
NEG         R1,R0         ; R1-R0
```

算术逻辑运算指令

- **ADD——加法运算指令**

ADD 指令将两个数据相加，结果保存到 Rd 寄存器。

低寄存器的 ADD 指令的指令格式如下：

ADD Rd,Rn, Rm

ADD Rd,Rn,#expr3

ADD Rd,#expr8

其中： Rd 目标寄存器。必须在 R0～R7 之间。

Rn 第 1 个操作数寄存器。必须在 R0～R7 之间。

Rm 第 2 个操作数寄存器。必须在 R0～R7 之间。

expr3 3 位立即数，即 0～7。

expr8 8 位立即数，即 0～255。

指令编码格式(ADD Rd,Rn, Rm)：

15						9	8		6	5		3	2		0
0	0	0	1	1	0	0	Rm			Rn			Rd		

指令编码格式(ADD Rd,Rn,#expr3)：

15						9	8		6	5		3	2		0
0	0	0	1	1	1	0	immed_3			Rn			Rd		

指令编码格式(ADD Rd,#expr8)：

15					10	8	7							0
0	0	1	1	0	Rd		immed_8							

条件码标志：

指令会更新 N、Z、C 和 V 标志。

高或低寄存器的 ADD 指令的指令格式如下：

ADD Rd, Rm

其中： Rd 目标寄存器，也是第一个操作数寄存器。

Rm 第二个操作数寄存器。

指令编码格式：

15							8	7	6	5		3	2		0
0	1	0	0	0	1	0	0	H1	H2	Rm			Rd		

H1 用于指示 Rd 是否为高寄存器。

H2 用于指示 Rm 是否为高寄存器。

条件码标志：

若 Rd 或 Rm 都是低寄存器(R0～R7)，指令会更新 N、Z、C 和 V 标志。其它情况不影响条件码标志。

PC 或 SP 相对偏移的 ADD 指令的指令格式如下：

ADD Rd,Rp,#expr

其中： Rd 目标寄存器。必须在 R0～R7 之间。

Rp PC 或 SP，第一个操作数寄存器。
expr 立即数，在 0～1020 范围内。

指令编码格式(ADD Rd,PC,#expr)：

15				11	10		8	7							0
1	0	1	0	0	Rd			immed_8							

指令编码格式(ADD Rd,SP,#expr)：

15				11	10		8	7							0
1	0	1	0	1	Rd			immed_8							

条件码标志：
不影响条件码标志。

SP 操作的 ADD 指令的指令格式如下：

ADD SP,#expr
其中： SP 目标寄存器，也是第一个操作数寄存器。
expr 立即数，在-508～+508 之间的 4 的整数倍的数。

指令编码格式：

15							8	7							0
1	0	1	1	0	0	0	0	signed_immed_8							

条件码标志：
不影响条件码标志。

ADD 指令举例如下：

```
ADD     R1,R1,R0      ; R1=R1+R0
ADD     R1,R1,#7      ; R1=R1+7
ADD     R3, #200      ; R3=R3+200
ADD     R3,R8         ; R3=R3+R8
ADD     R1,SP,#1000   ; R1=SP+1000
ADD     SP,#-500      ; SP=SP-500
```

- **SUB——减法运算指令**

SUB 指令将两个数相减，结果保存到 Rd 中。

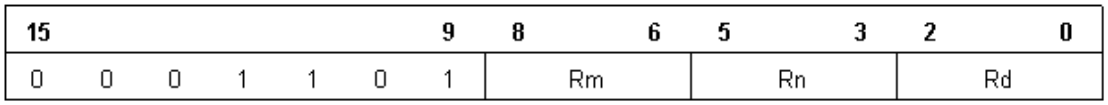
低寄存器的 SUB 指令的指令格式如下：

```
SUB     Rd,Rn, Rm
SUB     Rd,Rn,#expr3
SUB     Rd,#expr8
```

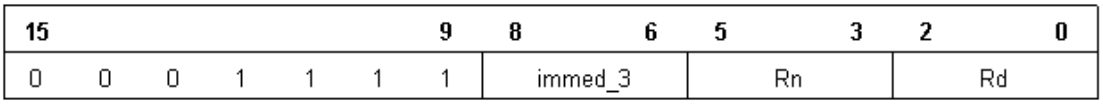
其中： Rd 目标寄存器，必须在 R0～R7 之间。
Rn 第 1 个操作数寄存器，必须在 R0～R7 之间。
Rm 第 2 个操作数寄存器，必须在 R0～R7 之间。
expr3 3 位立即数，即 0～7。

expr8 8 位立即数，即 0～255。

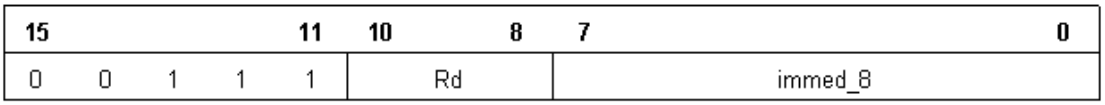
指令编码格式(SUB Rd,Rn, Rm)：



指令编码格式(SUB Rd,Rn,#expr3)：



指令编码格式(SUB Rd,#expr8)：



条件码标志：

指令会更新 N、Z、C 和 V 标志。

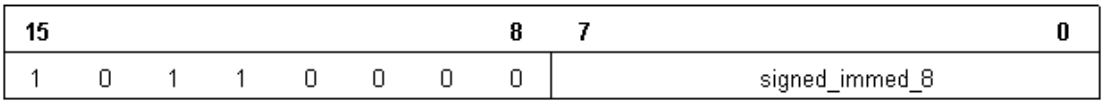
SP 操作的 SUB 指令的指令格式如下：

SUB SP,#expr

其中： SP 目标寄存器，也是第一个操作数寄存器。

expr 立即数，在-508～+508 之间的 4 的整数倍的数

指令编码格式：



条件码标志：

不影响条件码标志。

SUB 指令举例如下：

```
SUB     R0,R2,R1      ; R0=R2-R1
SUB     R2,R1,#1      ; R2=R1-1
SUB     R6,#250       ; R6=R6-250
SUB     SP,#380       ; SP=SP-380
```

- **ADC——带进位加法指令**

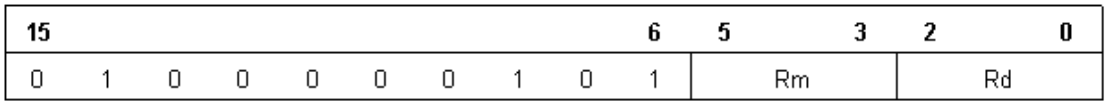
ADC 指令将 Rm 的值与 Rd 的值相加，再加上 CPSR 中的 C 条件标志位，结果保存到 Rd 寄存器。指令格式如下：

ADC Rd, Rm

其中： Rd 目标寄存器，也是第一个操作数寄存器。必须在 R0～R7 之间。

Rm 第二个操作数寄存器。必须在 R0～R7 之间。

指令编码格式：



条件码标志：
指令会更新 N、Z、C 和 V 标志。

ADC 指令举例如下：
```
ADD     R0,R2
ADC     R1,R3          ; 使用 ADC 实现 64 位加法，(R1、R0)=(R1、R0)+(R3、R2)
```

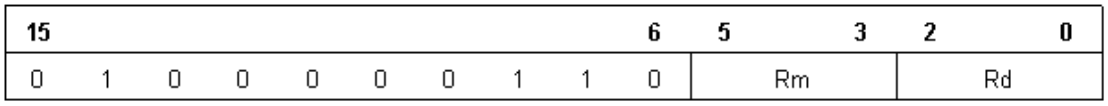
- **SBC——带进位减法指令**

SBC 指令用寄存器 Rd 减去 Rm，再减去 CPSR 中的 C 条件标志位的非(即若 C 标志清零，则结果减去 1)，结果保存到 Rd 中。指令格式如下：

```
SBC     Rd,Rm
```

其中： Rd 目标寄存器，也是第一个操作数寄存器。必须在 R0～R7 之间。
Rm 第二个操作数寄存器。必须在 R0～R7 之间。

指令编码格式：



条件码标志：
指令会更新 N、Z、C 和 V 标志。

SBC 指令举例如下：
```
SUB     R0,R2
SBC     R1,R3          ; 使用 SBC 实现 64 位减法，(R1、R0)=(R1、R0)-(R3、R2)
```

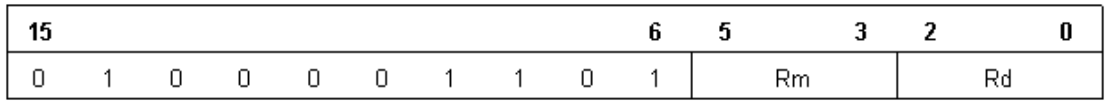
- **MUL——乘法运算指令**

MUL 指令用寄存器 Rd 乘以 Rm，结果保存到 Rd 中。指令格式如下：

```
MUL     Rd,Rm
```

其中： Rd 目标寄存器，也是第一个操作数寄存器。必须在 R0～R7 之间。
Rm 第二个操作数寄存器，必须在 R0～R7 之间。

指令编码格式：



条件码标志：
指令会更新 N 和 Z 标志。

MUL 指令举例如下：
```
MUL     R0,R1          ; R0=R0×R1
```

- **AND——逻辑“与”操作指令**

AND 指令将寄存器 Rd 的值与寄存器 Rm 的值按位作逻辑“与”操作，结果保存到 Rd 中。指令格式如下：

AND Rd,Rm

其中： Rd 目标寄存器，也是第一个操作数寄存器。必须在 R0～R7 之间。

Rm 第二个操作数寄存器。必须在 R0～R7 之间。

指令编码格式：

15									6	5	3	2	0
0	1	0	0	0	0	0	0	0	0	Rm		Rd	

条件码标志：

指令会更新 N 和 Z 标志。

AND 指令举例如下：

```
MOV     R1,#0x0F
AND     R0,R1          ; R0=R0&R1
```

- **ORR——逻辑“或”操作指令**

ORR 指令将寄存器 Rd 与寄存器 Rn 的值按位作逻辑“或”操作，结果保存到 Rd 中。指令格式如下：

ORR Rd,Rm

其中： Rd 目标寄存器，也是第 1 个操作数寄存器，必须在 R0～R7 之间。

Rm 第 2 个操作数寄存器，必须在 R0～R7 之间。

指令编码格式：

15									6	5	3	2	0
0	1	0	0	0	0	1	1	0	0	Rm		Rd	

条件码标志：

指令会更新 N 和 Z 标志。

ORR 指令举例如下：

```
MOV     R1,#0x03
ORR     R0,R1          ; R0=R0 | R1
```

- **EOR——逻辑“异或”操作指令**

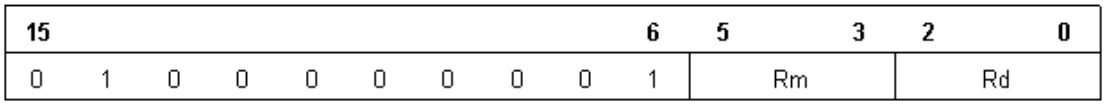
EOR 指令将寄存器 Rd 的值与寄存器 Rn 的值按位作逻辑异或操作，结果保存到 Rd 中。指令格式如下：

EOR Rd,Rm

其中： Rd 目标寄存器，也是第 1 个操作数寄存器，必须在 R0～R7 之间。

Rm 第 2 个操作数寄存器，必须在 R0～R7 之间。

指令编码格式：



15									6	5		3	2		0
0	1	0	0	0	0	0	0	0	1	Rm			Rd		

条件码标志：
指令会更新 N 和 Z 标志。

EOR 指令举例如下：

```
MOV     R2,#0xF0
EOR     R3,R2          ; R3=R3 ^ R2
```

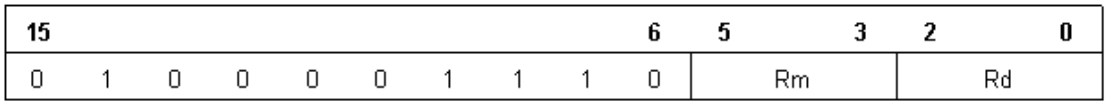
- **BIC——位清除指令**

BIC 指令将寄存器 Rd 的值与寄存器 Rm 的值的反码按位作逻辑 “与”操作，结果保存到 Rd 中。指令格式如下：

```
BIC     Rd,Rm
```

其中： Rd 目标寄存器，也是第 1 个操作数寄存器，必须在 R0～R7 之间。
Rm 第 2 个操作数寄存器，必须在 R0～R7 之间。

指令编码格式：



15									6	5		3	2		0
0	1	0	0	0	0	1	1	1	0	Rm			Rd		

条件码标志：
指令会更新 N 和 Z 标志。

BIC 指令举例如下：

```
MOV     R1,#0x80
BIC     R3,R1          ; 将 R1 的最高位清零，其它位不变
```

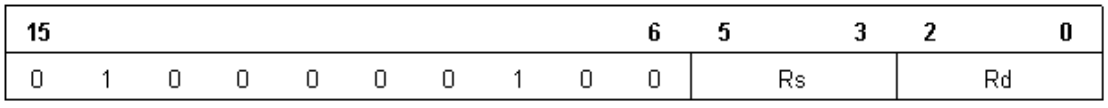
- **ASR——算术右移指令**

ASR 指令将数据算术右移，将符号位拷贝到空位，移位结果保存到 Rd 中。指令格式如下：

```
ASR     Rd,Rs
ASR     Rd,Rm,#expr
```

其中： Rd 目标寄存器，也是第 1 个操作数寄存器，必须在 R0～R7 之间。
Rs 寄存器控制移位中包含移位位数的寄存器，必须在 R0～R7 之间。
Rm 立即数移位的源寄存器，必须在 R0～R7 之间。
expr 立即数移位位数，值为 1～32。

指令编码格式(ASR Rd,Rs)：



15									6	5		3	2		0
0	1	0	0	0	0	0	1	0	0	Rs			Rd		

指令编码格式(ASR Rd,Rm,#expr)：

15				11	10	6	5	3	2	0
0	0	0	1	0	immed_5		Rm		Rd	

条件码标志：
指令会更新 N、Z 和 C 标志(若移位位数为零，则不影响 C 标志)。

ASR 指令举例如下：
ASR R1,R2
ASR R3,R1,#2

若移位位数为 32，则 Rd 清零，最后移出的位保留在标志 C 中；若移位量大于 32，则 Rd 和标志 C 均被清零；若移位量为 0，则不影响 C 标志。

- **LSL——逻辑左移指令**

LSR 指令将数据逻辑左移，空位清零，移位结果保存到 Rd 中。指令格式如下：
LSL Rd,Rs
LSL Rd,Rm,#expr
其中： Rd 目标寄存器，也是第 1 个操作数寄存器，必须在 R0～R7 之间。
Rs 寄存器控制移位中包含移位位数的寄存器，必须在 R0～R7 之间。
Rm 立即数移位的源寄存器，必须在 R0～R7 之间。
expr 立即数移位位数，值为 1～31。
指令编码格式(LSL Rd,Rs)：

15				11	10	6	5	3	2	0
0	0	0	0	0	immed_5		Rm		Rd	

指令编码格式(LSL Rd,Rm,#expr)：

15				11	10	6	5	3	2	0
0	0	0	0	0	immed_5		Rm		Rd	

条件码标志：
指令会更新 N、Z 和 C 标志(若移位位数为零，则不影响 C 标志)。

LSL 指令举例如下：
LSL R6,R7
LSL R1,R6,#2

若移位位数为 32，则 Rd 清零，最后移出的位保留在标志 C 中；若移位位数大于 32，则 Rd 和标志 C 均被清零；若移位位数为 0，则不影响 C 标志。

- **LSR——逻辑右移指令**

LSR 指令将数据逻辑右移，空位清零，移位结果保存到 Rd 中。指令格式如下：

LSR Rd,Rs
LSR Rd,Rm,#expr
其中： Rd 目标寄存器，也是第 1 个操作数寄存器，必须在 R0～R7 之间。
Rs 寄存器控制移位中包含移位位数的寄存器，必须在 R0～R7 之间。
Rm 立即数移位的源寄存器，必须在 R0～R7 之间。
expr 立即数移位位数，值为 1～32。

指令编码格式(LSR Rd,Rs)：

15									6	5		3	2		0
0	1	0	0	0	0	0	0	1	1	Rs			Rd		

指令编码格式(LSR Rd,Rm,#expr)：

15				11	10				6	5		3	2		0
0	0	0	0	1	immed_5					Rm			Rd		

条件码标志：
指令会更新 N、Z 和 C 标志(若移位位数为零，则不影响 C 标志)。

LSR 指令举例如下：
LSR R3,R0
LSR R5,R2,#2

若移位位数为 32，则 Rd 清零，最后移出的位保留在标志 C 中；若移位位数大于 32，则 Rd 和标志 C 均被清零；若移位位数为 0，则不影响 C 标志。

- **ROR——循环右移指令**

ROR 指令将数据循环右移，寄存器右边移出的位循环移回到左边，移位结果保存到 Rd 中。指令格式如下：

ROR Rd,Rs
其中： Rd 目标寄存器，也是第 1 个操作数寄存器，必须在 R0～R7 之间。
Rs 寄存器控制移位中包含移位位数的寄存器，必须在 R0～R7 之间。

指令编码格式：

15									6	5		3	2		0
0	1	0	0	0	0	0	1	1	1	Rs			Rd		

条件码标志：
指令会更新 N、Z 和 C 标志(若移位位数为零，则不影响 C 标志)。

ROR 指令举例如下：
ROR R2,R3

比较指令

- **CMP——比较指令**

CMP 指令使用寄存器 Rn 的值减去第二个操作数的值，根据操作的结果更新 CPSR 中的相应条件标志位。指令格式如下：

CMP Rn, Rm

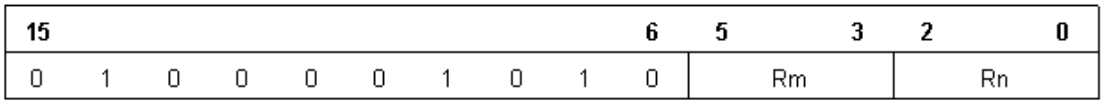
CMP Rn,#expr

其中：Rn 第一个操作数寄存器。对于 CMP Rn,#expr 指令，Rn 在 R0～R7 之间；对于 CMP Rn,Rm 指令，Rn 在 R0～R15 之间。

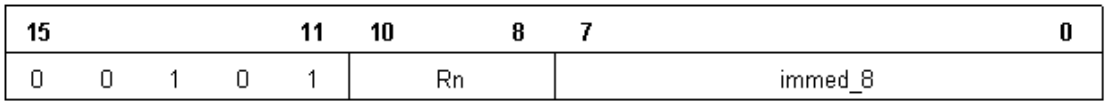
Rm 第二个操作数寄存器。Rm 在 R0～R15 之间。

expr 立即数，值为 0～255。

指令编码格式(CMP Rn, Rm)：



指令编码格式(CMP Rn,#expr)：



条件码标志：

指令会更新 N、Z、 C 和 V 标志。

CMP 指令举例如下：

```
CMP     R1,#10   ; R1 与 10 比较，设置相关标志位
CMP     R1,R2    ; R1 与 R2 比较，设置相关标志位
```

- **CMN——负数比较指令**

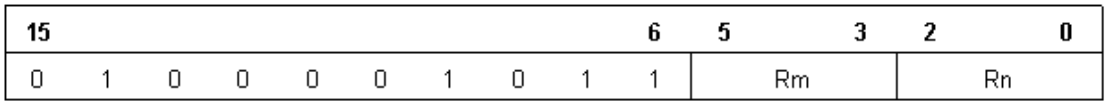
CMN 指令使用寄存器 Rn 的值加上寄存器 Rm 的值，根据操作的结果更新 CPSR 中的相应条件标志位。指令格式如下：

CMN Rn,Rm

其中：Rn 第 1 个操作数寄存器，必须在 R0～R7 之间。

Rm 第 2 个操作数寄存器，必须在 R0～R7 之间。

指令编码格式：



条件码标志：

指令会更新 N、Z、 C 和 V 标志。

CMN 指令举例如下：

```
CMN     R0,R2    ; R0 与-R2 进行比较
```

- **TST——位测试指令**

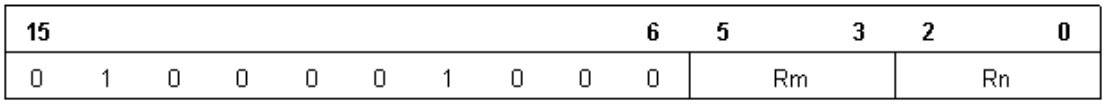
TST 指令将寄存器 Rn 的值与寄存器 Rm 的值按位作逻辑“与”操作，根据操作的结果更新 CPSR 中的相应条件标志位。指令格式如下：

TST Rn,Rm

其中： Rn 第 1 个操作数寄存器，必须在 R0～R7 之间。

Rm 第 2 个操作数寄存器，必须在 R0～R7 之间。

指令编码格式：



条件码标志：

指令会更新 N、Z、 C 和 V 标志。

TST 指令举例如下：

```
MOV     R0,#0x01
TST     R1,R0    ; 判断 R1 的最低位是否为 0
```

4. Thumb 分支指令

Thumb 分支指令见表 4.13。

表 4.13 Thumb 分支指令

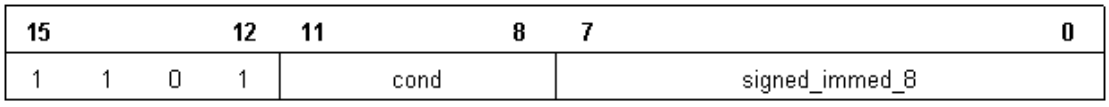
助记符	说明	操作	条件码位置
B label	分支指令	PC←label	B{cond}
BL label	带链接的分支指令	LR←PC-4，PC←label	无
BX Rm	带状态切换的分支指令	PC←label，切换处理器状态	无

- **B——分支指令**

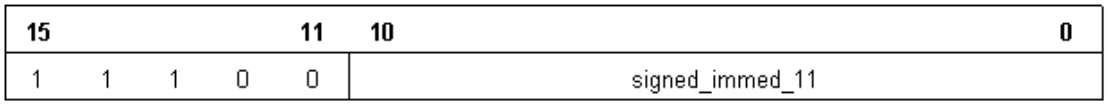
B 指令跳转到指定的地址执行程序。这是 Thumb 指令集中的惟一有条件执行指令。指令格式如下：

B{cond} label

指令编码格式(有条件执行)：



指令编码格式(无条件执行)：



分支指令 B 举例如下：

```
B       WAITB
BEQ     LOOP1
```

若使用 cond，则 label 必须在当前指令的-252～+256 字节范围内；若指令是无条件的，则分支指令 label 必须在当前指令的±2KB 范围内。

- **BL——带连接的分支指令**

BL 指令先将下一条指令的地址拷贝到 R14(即 LR)链接寄存器中，然后跳转到指定地址运行程序。指令格式如下：

BL label

指令编码格式：

15			12	11	10	0
1	1	1	1	H	offset_11	

H 区别±4MB 范围的高 11 位偏移(H 为 0)或低 11 位偏移(H 为 1)。

由于 BL 指令通常需要大的地址范围，很难用 16 位指令格式实现，为此，Thumb 采用两条这样的指令组合成 22 位半字偏移(符号扩展为 32 位)，使指令转移范围为±4MB。

带链接的分支指令 BL 举例如下：

```
BL      DELAY1
```

机器级分支指令 BL 限制在当前指令的±4MB 的范围内，必要时，ARM 链接器插入代码以允许更长的转移。

- **BX——带状态切换的分支指令**

BX 跳转到 Rm 指定的地址执行程序。若 Rm 的位[0]为 0，则 Rm 的位［1］也必须为 0。跳转时自动将 CPSR 中的标志 T 复位，即把目标地址的代码解释为 ARM 代码。指令格式如下：

BX Rm

指令编码格式：

15								7	6	5	3	2	0
0	1	0	0	0	1	1	1	0	H	Rm		0 0	0

H 用于区别高寄存器(H 为 1)或低寄存器(H 为 0)。

带状态切换的分支指令 BX 举例如下：

```
ADR     R0,ArmFun
BX      R0      ; 跳转到 R0 指定的地址，并根据 R0 的最低位来切换处理器状态
```

5. Thumb 杂项指令

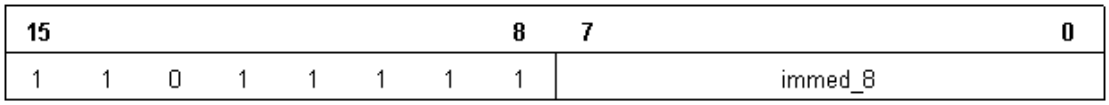
- **SWI——软中断指令**

SWI 指令用于产生软中断，从而实现从用户模式变换到管理模式，CPSR 保存到管理模式的 SPSR 中，执行转移到 SWI 向量。在其它模式下也可使用 SWI 指令，处理器同样地切换到管理模式。指令格式如下：

SWI immed_8

其中： immed_8 8 位立即数，值为 0～255 之间的整数。

指令编码格式：



SWI 指令举例如下：

```
SWI     1               ; 软中断，中断立即数为 0
SWI     0x55            ; 软中断，中断立即数为 0x55
```

使用 SWI 指令时，通常使用以下两种方法进行传递参数，SWI 异常中断处理程序就可以提供相关的服务。这两种方法均是由用户自己协定。SWI 异常中断处理程序要通过读取引起软中断的 SWI 指令，以取得 8 位立即数。

1. 指令中的 8 位立即数指定了用户请求的服务类型，参数通过通用寄存器传递。

```
MOV     R0,#34          ; 设置子功能号为 34
SWI     18              ; 调用 18 号软中断
```

2. 指令中的 8 位立即数被忽略，用户请求的服务类型由寄存器 R0 的值决定，参数通过其它的通用寄存器传递。

```
MOV     R0,#18          ; 调用 18 号软中断
MOV     R1,#34          ; 设置子功能号为 34
SWI     0
```

6. Thumb 伪指令

- **ADR——小范围的地址读取伪指令**

ADR 指令将基于 PC 相对偏移的地址值读取到寄存器中。ADR 伪指令格式如下：

```
ADR     register,expr
```

其中：
- register 加载的目标寄存器。
- expr 地址表达式。偏移量必须是正数并小于 1KB。Expr 必须局部定义，不能被导入。

ADR 伪指令举例如下：

```
        ADR     R0,TxtTab
        …
TxtTab
        DCB     "ARM7TDMI",0
```

- **LDR——大范围的地址读取伪指令**

LDR 伪指令用于加载 32 位的立即数或一个地址值到指定寄存器。在汇编编译源程序时，LDR 伪指令被编译器替换成一条合适的指令。若加载的常数未超出 MOV 的范围，则使用 MOV 或 MVN 指令代替该 LDR 伪指令，否则汇编器将常量放入文字池，并使用一条程序相对偏移的 LDR 指令从文字池读出常量。LDR 伪指令格式如下：

```
LDR     register,=expr/label-expr
```

其中：
- register 加载的目标寄存器。
- expr 32 位立即数。
- label-expr 基于 PC 的地址表达式或外部表达式。

LDR 伪指令举例如下：

```
LDR     R0,=0x12345678        ; 加载 32 位立即数 0x12345678
LDR     R0,=DATA_BUF+60       ; 加载 DATA_BUF 地址+60
…
LTORG                         ; 声明文字池
…
```

从 PC 到文字池的偏移量必须是正数并小于 1KB。
与 Thumb 指令的 LDR 相比，伪指令的 LDR 的参数有“=”号。

- **NOP——空操作伪指令**

NOP 伪指令在汇编时将会被代替成 ARM 中的空操作，比如可能为 MOV R0,R0 指令等。NOP 伪指令格式如下：

```
NOP
```

NOP 可用于延时操作。

2.6 本章小结

本章详细地价绍了 ARM 指令集、Thumb 指令集，并列出了各条指令的编码格式及相关应用举例，使读者对 ARM7TDMI(-S)的指令系统有全面的了解。

思考与练习

1 基础知识

a) ARM7TDMI(-S)有几种寻址方式？LDR R1,[R0,#0x08]属于哪种寻址方式？
b) ARM 指令的条件码有多少个?默认条件码是什么?
c) ARM 指令中第二个操作数有哪几种形式?列举 5 个 8 位图立即数。
d) LDR/STR 指令的偏移形式有哪 4 种？LDRB 和 LDRSB 有何区别？
e) 请指出 MOV 指令与 LDR 加载指令的区别及用途。
f) CMP 指令的操作是什么？写一程序，判断 R1 的值是否大于 0x30，是则将 R1 减去 0x30。
g) 调用子程序是用 B 还是用 BL 指令？请写出返回子程序的指令？
h) 请指出 LDR 伪指令的用法。指令格式与 LDR 加载指令的区别是什么？
i) ARM 状态与 Thumb 状态的切换指令是什么？请举例说明。
j) Thumb 状态与 ARM 状态的寄存器有区别吗？Thumb 指令对哪些寄存器的访问受到一定即制？
k) Thumb 指令集的堆栈入栈、出栈指令是哪两条？
l) Thumb 指令集的 BL 指令转移范围为何能达到±4MB？其指令编码是怎样的？

2 有符号和无符号加法

下面给出 A 和 B 的值，您可先手动计算 A+B，并预测 N、Z、V 和 C 标志位的值。然后修改程序清单 4.1 中 R0、R1 的值，将这两个值装载到这两个寄存器中(使用 LDR 伪指令，如 LDR R0,=0x FFFF0000)，使其执行两个寄存器的加法操作。调试程序，每执行一次加法操作就将标志位的状态记录下来，并将所得结果与您预先计算得出的结果相比较。如果两个操作数看作是有符号数，如何解释所得标志位的状态？同样，如果这两个操作数看作是无符

号数，所得标志位又当如何理解？

	0xFFFF000F	0x7FFFFFFF	67654321 (A)
+	0x0000FFF1	+ 0x02345678	+ 23110000 (B)
结果：	( )	( )	( )

3 数据访问

把下面的 C 代码转换成汇编代码。数组 a 和 b 分别存放在以 0x4000 和 0x5000 为起始地址的存储区内，类型为 long(即 32 位)。把编写的汇编语言进行编译连接，并进行调试。

```
for (i=0; i<8; i++)
{   a[i] = b[7-i];
}
```

4 阶乘计算

计算一个数 n 的阶乘，即 n!=n*(n-1)*(n-2)…(1)。

给定 n 的值后，整个算法就是不断使当前值与前一次乘数减一所得值相乘，这里所说的当前值即是乘法运算的结果。程序不断循环执行乘法操作，每次循环先将乘数减一，若所得值为 0 则循环结束。在程序中，使用条件执行的思想来做乘法。在编写含有循环和转移指令的程序时，由于可以用 Z 标志来迅速判断是否到达循环次数，很多编程者通常使用一个非零数向下计数而不是向上计数的方法来起动程序。

请填充下面的代码段，并加入相应的段声明信息，然后调试程序的正确性。设定 n 的值为 10，说明程序执行结果，并观察程序运行之前和之后寄存器的内容。

```
FACTORIAL       MOV     R6,#10          ; 将 10 存放到 R6 (n)
                MOV     R4,R6           ; 初始化保存结果的寄存器 R4 (n 的结果)
LOOP            SUBS    _____           ; 本次乘数减一
                MULNE   _____           ; 乘法运算
                BNE     LOOP            ; 如果循环未结束，转去执行下次循环
```

第5章 LPC2000系列ARM硬件结构

5.1 简介

5.1.1 描述

LPC2114/2124/2210/2212/2214 是基于一个支持实时仿真和跟踪的 16/32 位 ARM7TDMI-S^{TM} CPU 的微控制器，并带有 0/128/256 K 字节嵌入的高速片内 Flash 存储器。片内 128 位宽度的存储器接口和独特的加速结构使 32 位代码能够在最大时钟速率下运行。对代码规模有严格控制的应用可使用 16 位 Thumb 模式将代码规模降低超过 30%，而性能的损失却很小。

由于 LPC2114/2124/2210/2212/2214 较小的 64 和 144 脚封装、极低的功耗、多个 32 位定时器、4 路 10 位 ADC 或 8 路 10 位 ADC（64 脚和 144 脚封装）以及多达 9 个外部中断使它们特别适用于工业控制、医疗系统、访问控制和 POS 机。

在 64 脚的封装中，最多可使用 46 个 GPIO。在 144 脚的封装中，可使用的 GPIO 高达 76（使用了外部存储器）～112 个（单片应用）。由于内置了宽范围的串行通信接口，它们也非常适合于通信网关、协议转换器、嵌入式软 MODEM 以及其它各种类型的应用。

5.1.2 特性

- 16/32 位 64/144 脚 ARM7TDMI-S 微控制器。
- 16K 字节静态 RAM。
- 0/128/256K 字节片内 Flash 程序存储器。128 位宽度接口/加速器实现高达 60MHz 的操作频率。
- 外部 8、16 或 32 位总线（144 脚封装）。
- 通过外部存储器接口可将存储器配置成 4 组，每组的容量高达 16M 字节。
- 片内 Boot 装载程序实现在系统编程（ISP）和在应用中编程（IAP）。Flash 编程时间：1ms 可编程 512 字节，扇区擦除或整片擦除只需 400ms。（针对片内有 Flash 的型号）。
- 串行 Boot 装载程序通过 UART0 将应用程序装入器件的 RAM 中并使其在 RAM 中执行（针对 LPC2210）。
- EmbeddedICE-RT 接口使能断点和观察点。当前台任务使用片内 RealMonitor 软件调试时，中断服务程序可继续执行。
- 嵌入式跟踪宏单元（ETM）支持对执行代码进行无干扰的高速实时跟踪。
- 4/8 路（64/144 脚封装）10 位 A/D 转换器，转换时间低至 2.44ms。
- 2 个 32 位定时器（带 4 路捕获和 4 路比较通道）、PWM 单元（6 路输出）、实时时钟和看门狗。
- 多个串行接口，包括 2 个 16C550 工业标准 UART、高速 I^2C 接口（400 kbit/s）和 2 个 SPI 接口。
- 通过片内 PLL 可实现最大为 60MHz 的 CPU 操作频率。
- 向量中断控制器。可配置优先级和向量地址。
- 多达 46 个（64 脚封装）或 112 个（144 脚封装）通用 I/O 口（可承受 5V 电压），12 个独立外部中断引脚（EINT 和 CAP 功能）。
- 晶振频率范围：1~30 MHz，若使用 PLL 或 ISP 功能为：10～25MHz。

- 2 个低功耗模式：空闲和掉电。
- 通过外部中断将处理器从掉电模式中唤醒。
- 可通过个别使能/禁止外部功能来优化功耗。
- 双电源
  －CPU 操作电压范围：1.65~1.95 V(1.8 V ±8.3%)
  －I/O 操作电压范围：3.0~3.6 V(3.3 V ±10%)

5.1.3 器件信息

器件信息见表 5.1。

表 5.1 LPC2114/2124/2210/2212/2214 器件信息

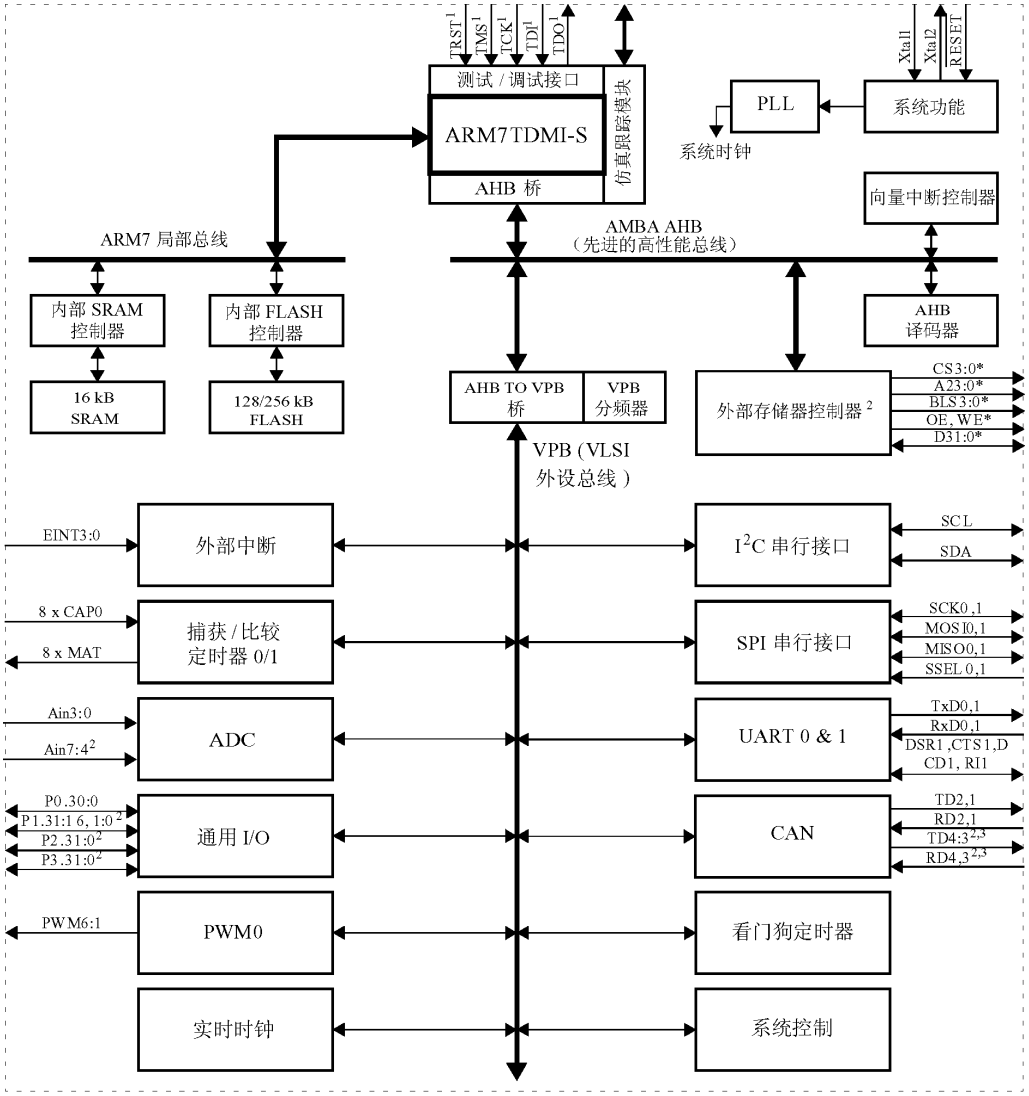
器件	引脚数	片内 RAM	片内 FLASH	10 位 A/D 通道数	注
LPC2114	64	16 kB	128 kB	4	—
LPC2124	64	16 kB	256 kB	4	—
LPC2210	144	16 kB	—	8	带外部存储器接口
LPC2212	144	16 kB	128 kB	8	带外部存储器接口
LPC2214	144	16 kB	256 kB	8	带外部存储器接口

5.1.4 结构概述

LPC2114/2124/2210/2212/2214 的结构见图 5.1，它们包含一个支持仿真的 ARM7TDMI-S CPU、与片内存储器控制器接口的 ARM7 局部总线、与中断控制器接口的 AMBA 高性能总线（AHB）和连接片内外设功能的 VLSI 外设总线（VPB，ARM AMBA 总线的兼容超集）。LPC2114/2124/2210/2212/2214 将 ARM7TDMI-S 配置为小端（little-endian）字节顺序。

AHB 外设分配了 2M 字节的地址范围，它位于 4G 字节 ARM 存储器空间的最顶端。每个 AHB 外设都分配了 16K 字节的地址空间。LPC2114/2124/2210/2212/2214 的外设功能（中断控制器除外）都连接到 VPB 总线。AHB 到 VPB 的桥将 VPB 总线与 AHB 总线相连。VPB 外设也分配了 2M 字节的地址范围，从 3.5GB 地址点开始。每个 VPB 外设在 VPB 地址空间内都分配了 16K 字节地址空间。

片内外设与器件引脚的连接由引脚连接模块控制。软件可以通过控制该模块让引脚与特定的片内外设相连接。



* 与 GPIO 共用

[1] 当使用测试/调试接口时，共用这些引脚的 GPIO/其它功能都不可用。

[2] 仅对 LPC2210/2212/2214 有效

图 5.1 LPC2114/2124/2210/2212/2214 方框图

5.2 引脚配置

5.2.1 引脚排列及封装信息

LPC2114/2124 的引脚分布见图 5.2。

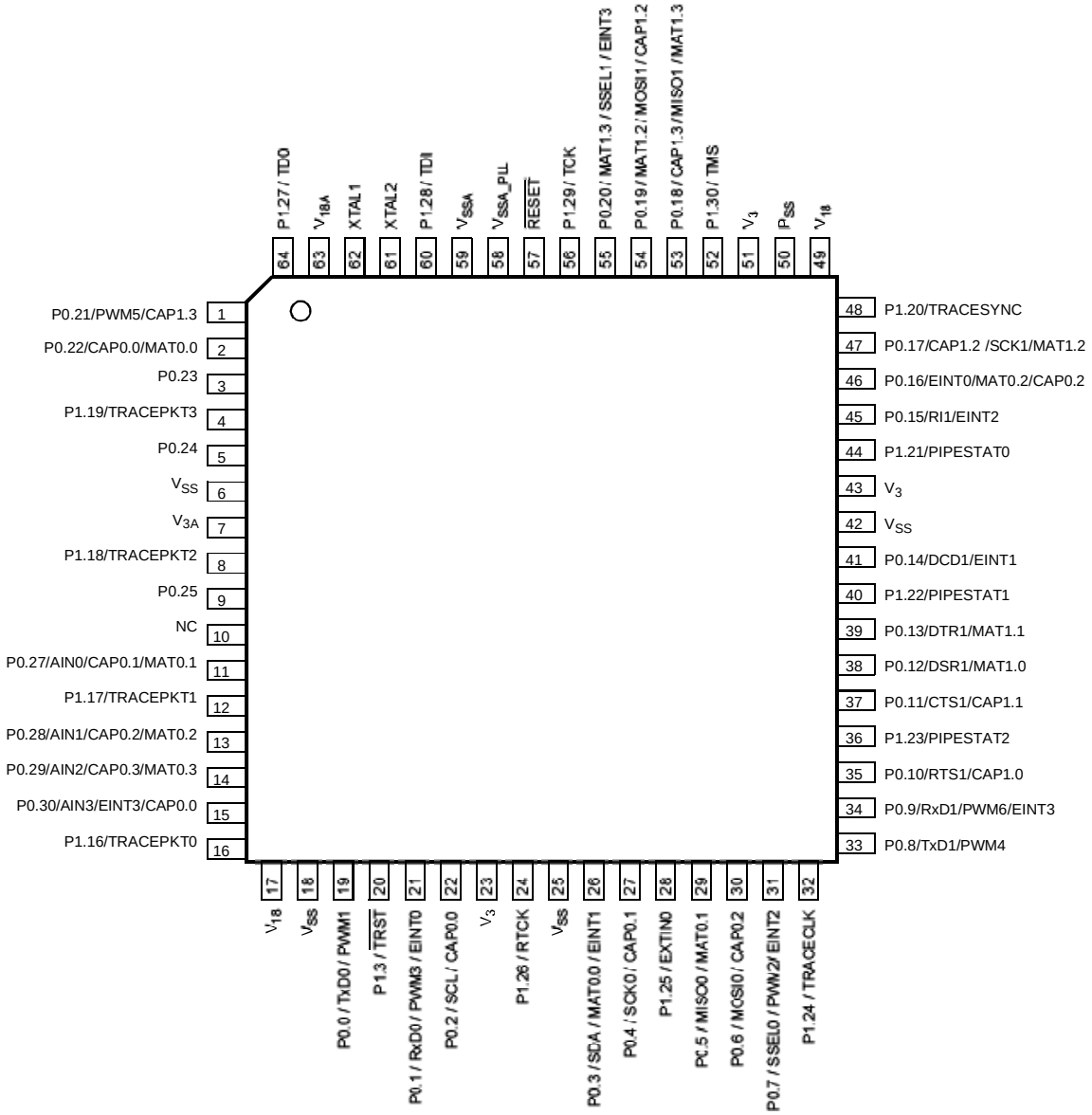


图 5.2 LPC2114/2124 的 64 脚封装

LPC2210/2212/2214 的引脚分布见图 5.3。

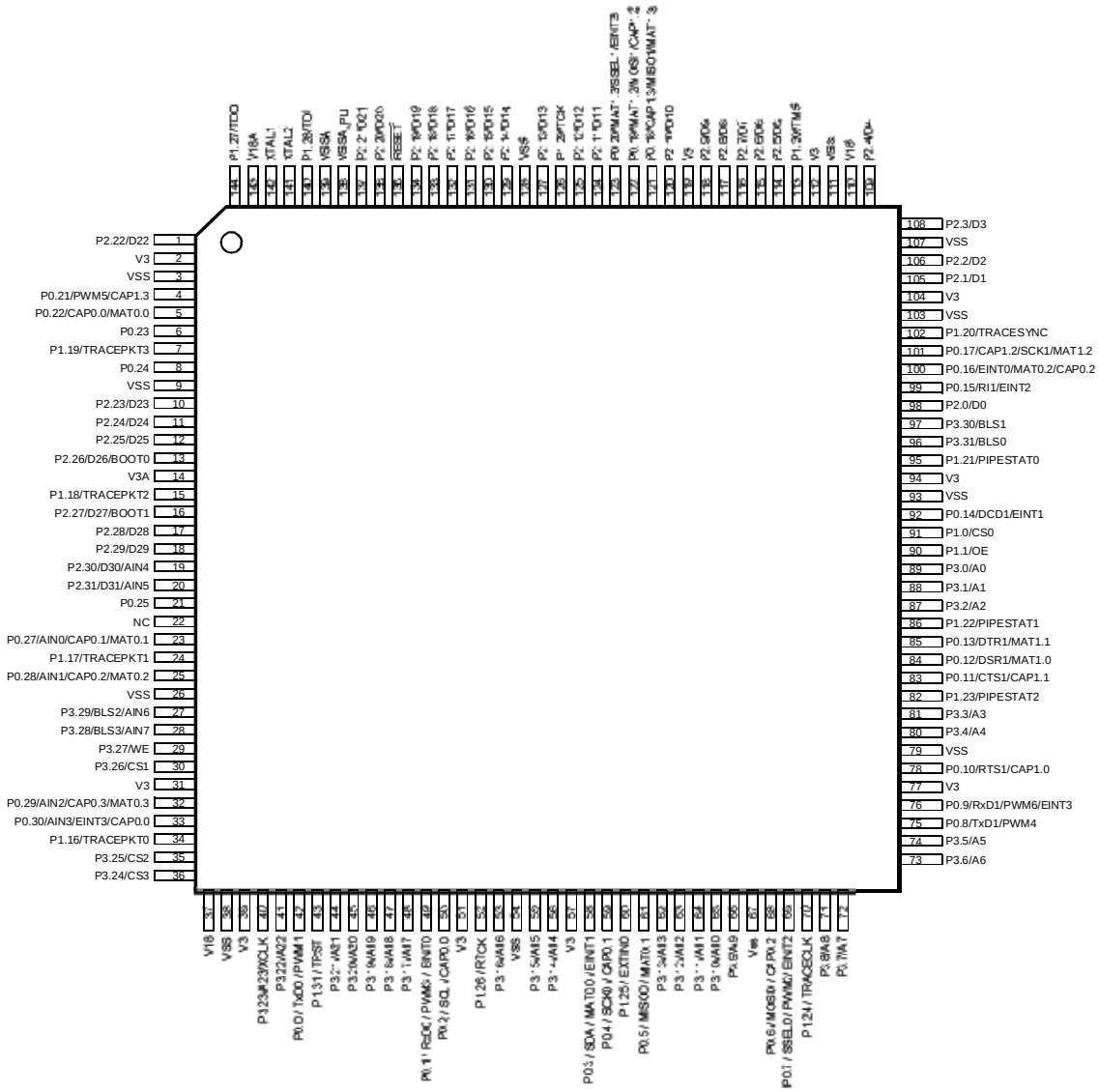


图 5.3 LPC2210/2212/2214 的 144 脚封装

5.2.2 LPC2114/2124 的引脚描述

LPC2114/2124 的引脚描述及其主要功能见表 5.2。

表 5.2 LPC2114/2124 的引脚描述

引脚名称	LQFP64 引脚#	类型	描述		
P0.0~P0.1		I/O	**P0 口：**P0 口是一个 32 位双向 I/O 口，每位的方向可单独控制。P0 口的功能取决于引脚连接模块的引脚功能选择。P0.26 和 P0.31 脚未用。		
	19	O	**P0.0**	**TxD0**	UART0 发送输出端。
		O		**PWM1**	脉宽调制器输出 1。
	21	I	**P0.1**	**RxD0**	UART0 接收输入端。
		O		**PWM3**	脉宽调制器输出 3。
		I		**EINT0**	外部中断 0 输入。

接上表

引脚名称	LQFP64 引脚#	类型	描述		
P0.2~P0.16	22	I/O	**P0.2**	**SCL**	I^2C 时钟输入/输出，开漏输出。
		I		**CAP0.0**	TIMER0 的捕获输入通道 0。
	26	I/O	**P0.3**	**SDA**	I^2C 数据输入/输出，开漏输出。
		O		**MAT0.0**	TIMER0 的匹配输出通道 0。
				EINT1	外部中断 1 输入。
	27	I/O	**P0.4**	**SCK0**	SPI0 的串行时钟。SPI 时钟从主机输出，从机输入。
		I		**CAP0.1**	TIMER0 的捕获输入通道 1。
	29	I/O	**P0.5**	**MISO0**	SPI0 主机输入从机输出端。数据输入到 SPI 主机或从 SPI 从机输出。
		O		**MAT0.1**	TIMER0 的匹配输出通道 1。
	30	I/O	**P0.6**	**MOSI0**	SPI0 主机输出从机输入端。数据从 SPI 主机输出或输入到 SPI 从机。
		I		**CAP0.2**	TIMER0 的捕获输入通道 2。
	31	I	**P0.7**	**SSEL0**	SPI0 从机选择。选择 SPI 接口用作从机。
		O		**PWM2**	脉宽调制器输出 2。
		I		**EINT2**	外部中断 2 输入。
	33	O	**P0.8**	**TxD1**	UART1 发送输出端。
		O		**PWM4**	脉宽调制器输出 4。
	34	I	**P0.9**	**RxD1**	UART1 接收输入端。
		O		**PWM6**	脉宽调制器输出 6。
		I		**EINT3**	外部中断 3 输入。
	35	O	**P0.10**	**RTS1**	UART1 请求发送输出端。
		I		**CAP1.0**	TIMER1 的捕获输入通道 0。
	37	I	**P0.11**	**CTS1**	UART1 清除发送输入端。
		I		**CAP1.1**	TIMER1 的捕获输入通道 1。
	38	I	**P0.12**	**DSR1**	UART1 数据设置就绪端。
		O		**MAT1.0**	TIMER1 的匹配输出通道 0。
	39	O	**P0.13**	**DTR1**	UART1 数据终止就绪端。
		O		**MAT1.1**	TIMER1 的匹配输出通道 1。
	41	I	**P0.14**	**DCD1**	UART1 数据载波检测输入端。
		I		**EINT1**	外部中断 1 输入。
				重点：$\overline{\text{RESET}}$ 为低时，P0.14 的低电平将强制片内引导装载程序复位后控制器件的操作，即进入 ISP 状态。	
	45	I	**P0.15**	**RI1**	UART1 铃响指示输入端。
		I		**EINT2**	外部中断 2 输入。
	46	I	**P0.16**	**EINT0**	外部中断 0 输入。
		O		**MAT0.2**	TIMER0 的匹配输出通道 2。
		I		**CAP0.2**	TIMER0 的捕获输入通道 2。

接上表

引脚名称	LQFP64 引脚#	类型	描述		
P0.17~P0.30	47	I	**P0.17**	**CAP1.2**	TIMER1 的捕获输入通道 2。
		I/O		**SCK1**	SPI1 串行时钟。SPI 时钟从主机输出或输入到从机。
		O		**MAT1.2**	TIMER1 的匹配输出通道 2。
	53	I	**P0.18**	**CAP1.3**	TIMER1 的捕获输入通道 3。
		I/O		**MISO1**	SPI1 主机输入从机输出端。数据输入到 SPI 主机或从 SPI 从机输出。
		O		**MAT1.3**	TIMER1 的匹配输出通道 3。
	54	O	**P0.19**	**MAT1.2**	TIMER1 的匹配输出通道 2。
		I/O		**MOSI1**	SPI1 主机输出从机输入端。数据从 SPI 主机输出或输入到 SPI 从机。
		O		**CAP1.2**	TIMER1 的捕获输入通道 2。
	55	O	**P0.20**	**MAT1.3**	TIMER1 的匹配输出通道 3。
		I		**SSEL1**	SPI1 从机选择。选择 SPI 接口用作从机。
		I		**EINT3**	外部中断 3 输入。
	1	O	**P0.21**	**PWM5**	脉宽调制器输出 5。
		I		**CAP1.3**	TIMER1 的捕获输入通道 3。
	2	I	**P0.22**	**CAP0.0**	TIMER0 的捕获输入通道 0。
		O		**MAT0.0**	TIMER0 的匹配输出通道 0。
	3	I/O	**P0.23**		通用双向数字端口。
	5	I/O	**P0.24**		通用双向数字端口。
	9	I/O	**P0.25**		通用双向数字端口。
	11	I	**P0.27**	**AIN0**	A/D 转换器输入 0。该模拟输入总是连接到相应的引脚上。
		I		**CAP0.1**	TIMER0 的捕获输入通道 1。
		O		**MAT0.1**	TIMER0 的匹配输出通道 1。
	13	I	**P0.28**	**AIN1**	A/D 转换器输入 1。该模拟输入总是连接到相应的引脚上。
		I		**CAP0.2**	TIMER0 的捕获输入通道 2。
		O		**MAT0.2**	TIMER0 的匹配输出通道 2。
	14	I	**P0.29**	**AIN2**	A/D 转换器输入 2。该模拟输入总是连接到相应的引脚上。
		I		**CAP0.3**	TIMER0 的捕获输入通道 3。
		O		**MAT0.3**	TIMER0 的匹配输出通道 3。
	15	I	**P0.30**	**AIN3**	A/D 转换器输入 3。该模拟输入总是连接到相应的引脚上。
		I		**EINT3**	外部中断 3 输入。
		I		**CAP0.0**	TIMER0 的捕获输入通道 0。

接上表

引脚名称	LQFP64 引脚#	类型	描述
P1.16~P1.31		I/O	**P1 口**：P1 口是一个 32 位双向 I/O 口，每位的方向可单独控制。P1 口的功能取决于引脚连接模块的引脚功能选择。只有 P1.16～P1.31 脚可用。
	16	O	**P1.16 TRACEPKT0** 跟踪包位 0。带内部上拉的标准 I/O 口。
	12	O	**P1.17 TRACEPKT1** 跟踪包位 1。带内部上拉的标准 I/O 口。
	8	O	**P1.18 TRACEPKT2** 跟踪包位 2。带内部上拉的标准 I/O 口。
	4	O	**P1.19 TRACEPKT3** 跟踪包位 3。带内部上拉的标准 I/O 口。
	48	O	**P1.20 TRACESYNC** 跟踪同步。标准 I/O 口带内部上拉。$\overline{\text{RESET}}$ 为低时，该引脚线上的低电平使 P1.25～P1.16 复位后用作跟踪端口。 **重点：$\overline{\text{RESET}}$ 为低时，P1.20 的低电平使 P1.25～P1.16 复位后用作跟踪端口。**
	44	O	**P1.21 PIPESTAT0** 流水线状态位 0。带内部上拉的标准 I/O 口。
	40	O	**P1.22 PIPESTAT1** 流水线状态位 1。带内部上拉的标准 I/O 口。
	36	O	**P1.23 PIPESTAT2** 流水线状态位 2。带内部上拉的标准 I/O 口。
	32	O	**P1.24 TRACECLK** 跟踪时钟。带内部上拉的标准 I/O 口。
	28	I	**P1.25 EXTIN0** 外部触发输入。带内部上拉的标准 I/O 口。
	24	I/O	**P1.26 RTCK** 返回的测试时钟输出。它是加载在 JTAG 接口的额外信号。辅助调试器与处理器频率的变化同步。双向引脚带内部上拉。$\overline{\text{RESET}}$ 为低时，该引脚线上的低电平使 P1.31～P1.26 复位后用作一个调试端口。 **重点：$\overline{\text{RESET}}$ 为低时，P1.26 的低电平使 P1.31～P1.26 复位后用作一个调试端口。**
	64	O	**P1.27 TDO** JTAG 接口的测试数据输出。
	60	I	**P1.28 TDI** JTAG 接口的测试数据输入。
	56	I	**P1.29 TCK** JTAG 接口的测试时钟。
	52	I	**P1.30 TMS** JTAG 接口的测试方式。
	20	I	**P1.31 $\overline{\text{TRST}}$** JTAG 接口的测试复位。
NC	10		引脚悬空。
$\overline{\text{RESET}}$	57	I	**外部复位输入**：当该引脚为低电平时，器件复位，I/O 口和外围功能进入默认状态，处理器从地址 0 开始执行程序。复位信号是具有迟滞作用的 TTL 电平。引脚可承受 5V 电压。
XTAL1	62	I	振荡器电路和内部时钟发生电路的输入。
XTAL2	61	O	振荡放大器的输出。
Vss	6,18,25, 42,50	I	**地**：0V 电压参考点。
V_{SSA}	59	I	**模拟地**：0V 电压参考点。它与 Vss 的电压相同，但为了降低噪声和出错几率，两者应当隔离。

接上表

引脚名称	LQFP64 引脚#	类型	描述
V_{SSA_PLL}	58	I	**PLL 模拟地**：0V 电压参考点。它与 Vss 的电压相同，但为了降低噪声和出错几率，两者应当隔离。
V_{18}	17,49	I	**1.8V 内核电源**：内部电路的电源电压。
V_{18A}	63	I	**模拟 1.8V 内核电源**：内部电路的电源电压。它与 V_{18} 的电压相同，但为了降低噪声和出错几率，两者应当隔离。
V_3	23,43, 51	I	**3.3V 端口电源**：I/O 口电源电压。
V_{3A}	7	I	**模拟 3.3V 端口电源**：它与 V_3 的电压相同，但为了降低噪声和出错几率，两者应当隔离。

5.2.3 LPC2210/2212/2214 的引脚描述

LPC2210/2212/2214 的引脚描述及其主要功能见表 5.3。

表 5.3 LPC2210/2212/2214 的引脚描述

引脚名称	LQFP144 引脚#	类型	描述		
P0.0~P0.6		I/O	**P0 口**：P0 口是一个 32 位双向 I/O 口，每位的方向可单独控制。P0 口的功能取决于引脚连接模块的引脚功能选择。P0.26 和 P0.31 脚未用。		
	42	O	**P0.0**	**TxD0**	UART0 发送输出端。
		O		**PWM1**	脉宽调制器输出 1。
	49	I	**P0.1**	**RxD0**	UART0 接收输入端。
		O		**PWM3**	脉宽调制器输出 3。
		I		**EINT0**	外部中断 0 输入。
	50	I/O	**P0.2**	**SCL**	I^2C 时钟输入/输出。开漏输出。
		I		**CAP0.0**	TIMER0 的捕获输入通道 0。
	58	I/O	**P0.3**	**SDA**	I^2C 数据输入/输出。开漏输出。
		O		**MAT0.0**	TIMER0 的匹配输出通道 0。
		I		**EINT1**	外部中断 1 输入。
	59	I/O	**P0.4**	**SCK0**	SPI0 的串行时钟。SPI 时钟从主机输出，从机输入。
		I		**CAP0.1**	TIMER0 的捕获输入通道 1。
	61	I/O	**P0.5**	**MISO0**	SPI0 主机输入从机输出端。数据输入到 SPI 主机或从 SPI 从机输出。
		O		**MAT0.1**	TIMER0 的匹配输出通道 1。
	68	I/O	**P0.6**	**MOSI0**	SPI0 主机输出从机输入端。数据从 SPI 主机输出或输入到 SPI 从机。
		I		**CAP0.2**	TIMER0 的捕获输入通道 2。

接上表

引脚名称	LQFP144 引脚#	类型	描述		
P0.7~P0.20	69	I	**P0.7**	**SSEL0**	SPI0 从机选择。选择 SPI 接口用作从机。
		O		**PWM2**	脉宽调制器输出 2。
		I		**EINT2**	外部中断 2 输入。
	75	O	**P0.8**	**TxD1**	UART1 发送输出端。
		O		**PWM4**	脉宽调制器输出 4。
	76	I	**P0.9**	**RxD1**	UART1 接收输入端。
		O		**PWM6**	脉宽调制器输出 6。
		I		**EINT3**	外部中断 3 输入。
	78	O	**P0.10**	**RTS1**	UART1 请求发送输出端。
		I		**CAP1.0**	TIMER1 的捕获输入通道 0。
	83	I	**P0.11**	**CTS1**	UART1 清除发送输入端。
		I		**CAP1.1**	TIMER1 的捕获输入通道 1。
	84	I	**P0.12**	**DSR1**	UART1 数据设置就绪端。
		O		**MAT1.0**	TIMER1 的匹配输出通道 0。
	85	O	**P0.13**	**DTR1**	UART1 数据终止就绪端。
		O		**MAT1.1**	TIMER1 的匹配输出通道 1。
	92	I	**P0.14**	**DCD1**	UART1 数据载波检测输入端。
		I		**EINT1**	外部中断 1 输入。
			重点：$\overline{\text{RESET}}$ 为低时， P0.14 的低电平强制片内引导装载程序复位后控制器件的操作，即进入 ISP 状态。		
	99	I	**P0.15**	**RI1**	UART1 铃响指示输入端。
		I		**EINT2**	外部中断 2 输入。
	100	I	**P0.16**	**EINT0**	外部中断 0 输入。
		O		**MAT0.2**	TIMER0 的匹配输出通道 2。
		I		**CAP0.2**	TIMER0 的捕获输入通道 2。
	101	I	**P0.17**	**CAP1.2**	TIMER1 的捕获输入通道 2。
		I/O		**SCK1**	SPI1 串行时钟。SPI 时钟从主机输出或输入到从机。
		O		**MAT1.2**	TIMER1 的匹配输出通道 2。
	121	I	**P0.18**	**CAP1.3**	TIMER1 的捕获输入通道 3。
		I/O		**MISO1**	SPI1 主机输入从机输出端。数据输入到 SPI 主机或从 SPI 从机输出。
		O		**MAT1.3**	TIMER1 的匹配输出通道 3。
	122	O	**P0.19**	**MAT1.2**	TIMER1 的匹配输出通道 2。
		I/O		**MOSI1**	SPI1 主机输出从机输入端。数据从 SPI 主机输出或输入到 SPI 从机。
		O		**CAP1.2**	TIMER1 的捕获输入通道 2。
	123	O	**P0.20**	**MAT1.3**	TIMER1 的匹配输出通道 3。
		I		**SSEL1**	SPI1 从机选择。选择 SPI 接口用作从机。
		I		**EINT3**	外部中断 3 输入。

接上表

引脚名称	LQFP144 引脚#	类型	描述		
P0.21~P0.30	4	O	**P0.21**	**PWM5**	脉宽调制器输出 5。
		I		**CAP1.3**	TIMER1 的捕获输入通道 3。
	5	I	**P0.22**	**CAP0.0**	TIMER0 的捕获输入通道 0。
		O		**MAT0.0**	TIMER0 的匹配输出通道 0。
	6	I/O	**P0.23**		通用双向数字端口。
	8	I/O	**P0.24**		通用双向数字端口。
	21	I/O	**P0.25**		通用双向数字端口。
	23	I	**P0.27**	**AIN0**	A/D 转换器输入 0。该模拟输入总是连接到相应的引脚上。
		I		**CAP0.1**	TIMER0 的捕获输入通道 1。
		O		**MAT0.1**	TIMER0 的匹配输出通道 1。
	25	I	**P0.28**	**AIN1**	A/D 转换器输入 1。该模拟输入总是连接到相应的引脚上。
		I		**CAP0.2**	TIMER0 的捕获输入通道 2。
		O		**MAT0.2**	TIMER0 的匹配输出通道 2。
	32	I	**P0.29**	**AIN2**	A/D 转换器输入 2。该模拟输入总是连接到相应的引脚上。
		I		**CAP0.3**	TIMER0 的捕获输入通道 3。
		O		**MAT0.3**	TIMER0 的匹配输出通道 3。
	33	I	**P0.30**	**AIN3**	A/D 转换器输入 3。该模拟输入总是连接到相应的引脚上。
		I		**EINT3**	外部中断 3 输入。
		I		**CAP0.0**	TIMER0 的捕获输入通道 0。
P1.0~P1.23		I/O	**P1 口**：P1 口是一个 32 位双向 I/O 口，每位的方向可单独控制。P1 口的功能取决于引脚连接模块的引脚功能选择。P1.2～P1.15 脚未用。		
	91	O	**P1.0**	**CS0**	低有效片选 0 信号。（Bank 0 地址范围为 8000 0000 – 80FF FFFF）
	90	O	**P1.1**	**OE**	低有效输出使能信号。
	34	O	**P1.16**	**TRACEPKT0**	跟踪包位 0。带内部上拉的标准 I/O 口。
	24	O	**P1.17**	**TRACEPKT1**	跟踪包位 1。带内部上拉的标准 I/O 口。
	15	O	**P1.18**	**TRACEPKT2**	跟踪包位 2。带内部上拉的标准 I/O 口。
	7	O	**P1.19**	**TRACEPKT3**	跟踪包位 3。带内部上拉的标准 I/O 口。
	102	O	**P1.20**	**TRACESYNC**	跟踪同步。带内部上拉的标准 I/O 口。$\overline{\text{RESET}}$ 为低时，该引脚线上的低电平使 P1.25～P1.16 复位后用作跟踪端口。 **重点：$\overline{\text{RESET}}$ 为低时，P1.20 的低电平使 P1.25～P1.16 复位后用作跟踪端口。**
	95	O	**P1.21**	**PIPESTAT0**	流水线状态位 0。带内部上拉的标准 I/O 口。
	86	O	**P1.22**	**PIPESTAT1**	流水线状态位 1。带内部上拉的标准 I/O 口。
	82	O	**P1.23**	**PIPESTAT2**	流水线状态位 2。带内部上拉的标准 I/O 口。

接上表

引脚名称	LQFP144 引脚#	类型	描述		
P1.24~P1.31	70	O	**P1.24**	**TRACECLK**	跟踪时钟。带内部上拉的标准 I/O 口。
	60	I	**P1.25**	**EXTIN0**	外部触发输入。带内部上拉的标准 I/O 口。
	52	I/O	**P1.26**	**RTCK**	返回的测试时钟输出。它是加载在 JTAG 接口的额外信号。辅助调试器与处理器频率的变化同步。带内部上拉的双向引脚。$\overline{\text{RESET}}$ 为低时，该引脚线上的低电平使 P1.31～P1.26 复位后用作一个调试端口。 **重点：$\overline{\text{RESET}}$ 为低时，P1.26 的低电平使 P1.31～P1.26 复位后用作一个调试端口。**
	144	O	**P1.27**	**TDO**	JTAG 接口的测试数据输出。
	140	I	**P1.28**	**TDI**	JTAG 接口的测试数据输入。
	126	I	**P1.29**	**TCK**	JTAG 接口的测试时钟。
	113	I	**P1.30**	**TMS**	JTAG 接口的测试方式。
	43	I	**P1.31**	**$\overline{\text{TRST}}$**	JTAG 接口的测试复位。
P2.0~P2.22		I/O	**P2 口：**P2 口是一个 32 位双向 I/O 口，每位的方向可单独控制。P2 口的功能取决于引脚连接模块的引脚功能选择。		
	98	I/O	**P2.0**	**D0**	外部存储器数据线 0。
	105	I/O	**P2.1**	**D1**	外部存储器数据线 1。
	106	I/O	**P2.2**	**D2**	外部存储器数据线 2。
	108	I/O	**P2.3**	**D3**	外部存储器数据线 3。
	109	I/O	**P2.4**	**D4**	外部存储器数据线 4。
	114	I/O	**P2.5**	**D5**	外部存储器数据线 5。
	115	I/O	**P2.6**	**D6**	外部存储器数据线 6。
	116	I/O	**P2.7**	**D7**	外部存储器数据线 7。
	117	I/O	**P2.8**	**D8**	外部存储器数据线 8。
	118	I/O	**P2.9**	**D9**	外部存储器数据线 9。
	120	I/O	**P2.10**	**D10**	外部存储器数据线 10。
	124	I/O	**P2.11**	**D11**	外部存储器数据线 11。
	125	I/O	**P2.12**	**D12**	外部存储器数据线 12。
	127	I/O	**P2.13**	**D13**	外部存储器数据线 13。
	129	I/O	**P2.14**	**D14**	外部存储器数据线 14。
	130	I/O	**P2.15**	**D15**	外部存储器数据线 15。
	131	I/O	**P2.16**	**D16**	外部存储器数据线 16。
	132	I/O	**P2.17**	**D17**	外部存储器数据线 17。
	133	I/O	**P2.18**	**D18**	外部存储器数据线 18。
	134	I/O	**P2.19**	**D19**	外部存储器数据线 19。
	136	I/O	**P2.20**	**D20**	外部存储器数据线 20。
	137	I/O	**P2.21**	**D21**	外部存储器数据线 21。
	1	I/O	**P2.22**	**D22**	外部存储器数据线 22。

接上表

引脚名称	LQFP144 引脚#	类型	描述		
P2.23~P2.31	10	I/O	**P2.23**	**D23**	外部存储器数据线 23。
	11	I/O	**P2.24**	**D24**	外部存储器数据线 24。
	12	I/O	**P2.25**	**D25**	外部存储器数据线 25。
	13	I/O	**P2.26**	**D26**	外部存储器数据线 26。
		I		**BOOT0**	当 RESET 为低时，BOOT0 与 BOOT1 一同控制引导和内部操作。引脚的内部上拉确保了引脚未连接时呈现高电平。
	16	I/O	**P2.27**	**D27**	外部存储器数据线 27。
		I		**BOOT1**	当 RESET 为低时，BOOT1 与 BOOT0 一同控制引导和内部操作。引脚的内部上拉确保了引脚未连接时呈现高电平。 **BOOT1:0=00 选择引导 CS0 控制的 8 位存储器。** **BOOT1:0=01 选择引导 CS0 控制的 16 位存储器。** **BOOT1:0=10 选择引导 CS0 控制的 32 位存储器。** **BOOT1:0=11 选择内部 Flash 存储器。**
	17	I/O	**P2.28**	**D28**	外部存储器数据线 28。
	18	I/O	**P2.29**	**D29**	外部存储器数据线 29。
	19	I/O	**P2.30**	**D30**	外部存储器数据线 30。
		I		**AIN4**	A/D 转换器输入 4。该模拟输入总是连接到相应的引脚上。
	20	I/O	**P2.31**	**D31**	外部存储器数据线 31。
		I		**AIN5**	A/D 转换器输入 5。该模拟输入总是连接到相应的引脚上。
P3.0~P3.11		I/O	**P3 口：**P3 口是一个 32 位双向 I/O 口，每位的方向可单独控制。P3 口的功能取决于引脚连接模块的引脚功能选择。		
	89	O	**P3.0**	**A0**	外部存储器地址线 0。
	88	O	**P3.1**	**A1**	外部存储器地址线 1。
	87	O	**P3.2**	**A2**	外部存储器地址线 2。
	81	O	**P3.3**	**A3**	外部存储器地址线 3。
	80	O	**P3.4**	**A4**	外部存储器地址线 4。
	74	O	**P3.5**	**A5**	外部存储器地址线 5。
	73	O	**P3.6**	**A6**	外部存储器地址线 6。
	72	O	**P3.7**	**A7**	外部存储器地址线 7。
	71	O	**P3.8**	**A8**	外部存储器地址线 8。
	66	O	**P3.9**	**A9**	外部存储器地址线 9。
	65	O	**P3.10**	**A10**	外部存储器地址线 10。
	64	O	**P3.11**	**A11**	外部存储器地址线 11。

接上表

引脚名称	LQFP144 引脚#	类型	描述		
P3.12~P3.31	63	O	**P3.12**	**A12**	外部存储器地址线 12。
	62	O	**P3.13**	**A13**	外部存储器地址线 13。
	56	O	**P3.14**	**A14**	外部存储器地址线 14。
	55	O	**P3.15**	**A15**	外部存储器地址线 15。
	53	O	**P3.16**	**A16**	外部存储器地址线 16。
	48	O	**P3.17**	**A17**	外部存储器地址线 17。
	47	O	**P3.18**	**A18**	外部存储器地址线 18。
	46	O	**P3.19**	**A19**	外部存储器地址线 19。
	45	O	**P3.20**	**A20**	外部存储器地址线 20。
	44	O	**P3.21**	**A21**	外部存储器地址线 21。
	41	O	**P3.22**	**A22**	外部存储器地址线 22。
	40	I/O	**P3.23**	**A23**	外部存储器地址线 23。
		O		**XCLK**	时钟输出。
	36	O	**P3.24**	**CS3**	低有效片选 3 信号。（Bank 3 地址范围为 8300 0000 – 83FF FFFF）
	35	O	**P3.25**	**CS2**	低有效片选 2 信号。（Bank 2 地址范围为 8200 0000 – 82FF FFFF）
	30	O	**P3.26**	**CS1**	低有效片选 1 信号。（Bank 1 地址范围为 8100 0000 – 81FF FFFF）
	29	O	**P3.27**	**WE**	低有效写使能信号。
	28	O	**P3.28**	**BLS3**	字节定位选择信号（Bank 3），低有效。
		I		**AIN7**	A/D 转换器输入 7。该模拟输入总是连接到相应的引脚上。
	27	O	**P3.29**	**BLS2**	字节定位选择信号（Bank 2），低有效。
		I		**AIN6**	A/D 转换器输入 6。该模拟输入总是连接到相应的引脚上。
	97	O	**P3.30**	**BLS1**	字节定位选择信号（Bank 1），低有效。
	96	O	**P3.31**	**BLS0**	字节定位选择信号（Bank 0），低有效。
NC	22		引脚悬空。		
$\overline{\text{RESET}}$	135	I	**外部复位输入**：当该引脚为低电平时，器件复位，I/O 口和外围功能进入默认状态，处理器从地址 0 开始执行程序。复位信号是具有迟滞作用的 TTL 电平。引脚可承受 5V 电压。		
XTAL1	142	I	振荡器电路和内部时钟发生电路的输入。		
XTAL2	141	O	振荡放大器的输出。		
Vss	3,9,26, 38,54,67,7 9,93, 103,107,1 11,128	I	**地**：0V 电压参考点。		

接上表

引脚名称	LQFP144 引脚#	类型	描述
V_{SSA}	139	I	**模拟地**：0V 电压参考点。它与 Vss 的电压相同，但为了降低噪声和出错几率，两者应当隔离。
V_{SSA_PLL}	138	I	**PLL 模拟地**：0V 电压参考点。它与 Vss 的电压相同，但为了降低噪声和出错几率，两者应当隔离。
V_{18}	37,110	I	**1.8V 内核电源**：内部电路的电源电压。
V_{18A}	143	I	**模拟 1.8V 内核电源**：内部电路的电源电压。它与 V_{18} 的电压相同，但为了降低噪声和出错几率，两者应当隔离。
V_3	2,31,39,51,57,77,94,104,112,119	I	**3.3V 端口电源**：I/O 口电源电压。
V_{3A}	14	I	**模拟 3.3V 端口电源**：它与 V_3 的电压相同，但为了降低噪声和出错几率，两者应当隔离。

5.2.4 引脚功能选择使用示例

LPC2114/2124/2210/2212/2214 的引脚一般是多个功能复用的，比如 P0.0 口，它可以作为 GPIO(通用 I/O)功能，可以作为 UART0 的 TxD0，还可以作脉宽调制器输出 PWM1。但同一引脚在同一时刻只能使用其中的一个功能，这是通过设置 PINSEL0、PINSEL1 或 PINSEL2 来进行选择，详细介绍请参考本书的第 5.7 节。芯片复位时 PINSEL0、PINSEL1 和 PINSEL2 会自动设置为默认值，所以复位后芯片引脚的功能是确定的。

应该注意的是 P0.2、P0.3 口是硬件 I^2C 接口，为开漏输出口，即使将它们设置为 GPIO 功能。

1. 设置 P0.0 为个 GPIO 功能

```
PINSEL0 = PINSEL0 & 0xFFFFFFFC;
```

2. 设置 P0.0 为 TxD0 功能

```
PINSEL0 = (PINSEL0 & 0xFFFFFFFC) | 0x01;
```

5.3 存储器寻址

5.3.1 片内存储器

1. 片内 FLASH 程序存储器

LPC2114/2212 集成了一个 128K，而 LPC2124/2214 集成了 256K 的 FLASH 存储器系统。该存储器可用作代码和数据的存储。对 FLASH 存储器的编程可通过几种方法来实现：通过内置的串行 JTAG 接口，通过在系统中编程（即 ISP，使用 UART0 通讯），或通过在应用编程（IAP）。使用在应用编程的应用程序，可以在应用程序运行时对 FLAH 进行擦除和/或编程，这样就为数据存储和现场固件的升级都带来了极大的灵活性。

2. 片内静态 RAM

LPC2114/2124/2210/2212/2214 含有 16KB 的静态 RAM，可用作代码和/或数据的存储。SRAM 支持 8 位、16 位和 32 位访问。

SRAM 控制器包含一个回写缓冲区，它用于防止 CPU 在连续的写操作时停止运行。回

写缓冲区总是保存着软件发送到 SRAM 的最后一个字数据。该数据只有在软件请求下一次写操作时才写入 SRAM（即：数据只有在软件执行另外一次写操作时被写入 SRAM）。如果发生芯片复位，实际的 SRAM 内容将不会反映最近一次的写请求（即：在一次“热”复位后，SRAM 不会反映最后一次写入的内容）。任何在复位后检查 SRAM 内容的程序都必须注意这一点。若用户程序关心“热”复位后 RAM 的内容，就还需要多执行一次写操作。

5.3.2 片外存储器

LPC2114/2124 不具备外部存储器接口，所以扩展片外存储器只能通过 I/O 口模拟总线操作，或者使用 I^2C、SPI 接口连接。

LPC2210/2212/2214 具备外部存储器接口，通过外部存储器控制器(EMC)可以扩展 4 个 Bank 的存储器组(Bank0 ～ Bank3)，每个存储器组的空间大小为 16M 字节。LPC2210/2212/2214 的 EMC 符合 ARM 公司的 PL090 标准，总线宽度可设置为 8 位、16 位、32 位，通常使用 16 位总线宽度的存储器具有较高的性价比。

对于外扩的 SRAM 存储器，使用 ARM 的 LDR/STR 指令即可进行数据的读写操作；而对于外扩的 FLASH（NOR 型），可以使用 LDR 指令读取数据，但是不能使用 STR 指令直接写数据，而是根据 FLASH 芯片写操作时序进行控制，实现 FLASH 的擦除编程。如果需要将程序代码烧写到扩展的 FLASH，则需要动行一个装载程序(Loder 程序，一般由用户自行编写)，这个程序的功能是通过串口接收要烧写的数据，然后擦除编程 FLASH，连接示意图如图 5.4 所示。

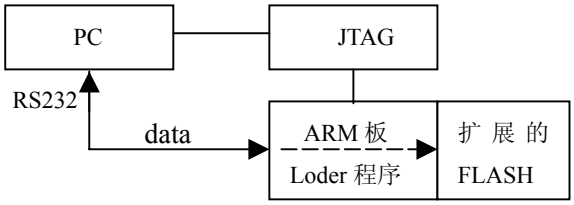


图 5.4 片外 FLASH 存储器 Loader 示意图

5.3.3 存储器映射

LPC2114/2124/2210/2212/2214 包含几个不同的存储器组，见图 5.5～图 5.8。图 5.5 所示为复位后从用户角度所看到的整个地址空间映射。中断向量支持地址的重新映射，详见第 5.3.5 小节。

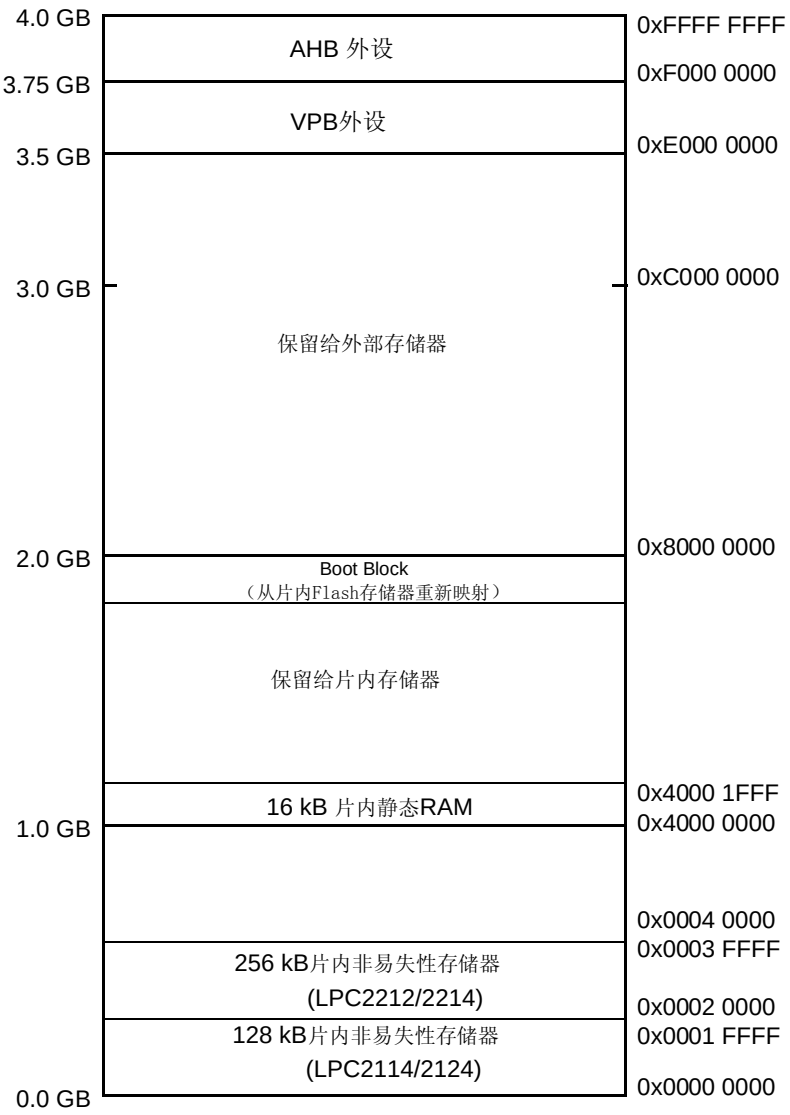
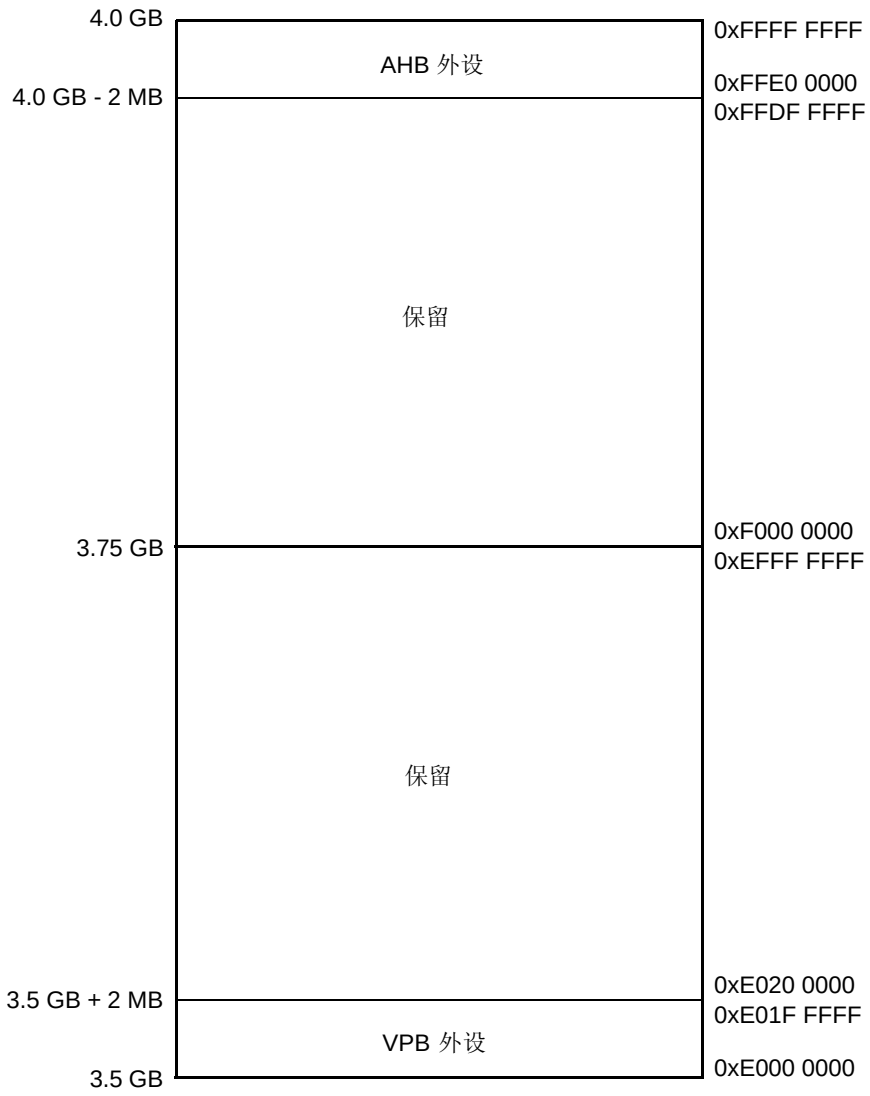


图 5.5 系统存储器映射

图 5.6～图 5.8 显示了从不同角度所观察到的外设地址空间。AHB 和 VPB 外设区域都为 2M 字节，可各自分配最多 128 个外设。每个外设空间的规格都为 16k 字节，这样就简化了每个外设的地址译码。所有外设寄存器不管规格大小，都按照字地址进行分配（32 位边界）。这样就不再需要使用字节定位的硬件来进行小边界的字节（8 位）或半字（16 位）访问。不管字还是半字寄存器都是一次性访问，例如，不可能对一个字寄存器的最高字节执行单独的读或写操作。



注：

－AHB 部分是 128×16kB 的范围（共 2MB）。

－VPB 部分是 128×16kB 的范围（共 2MB）。

图 5.6 外设存储器映射

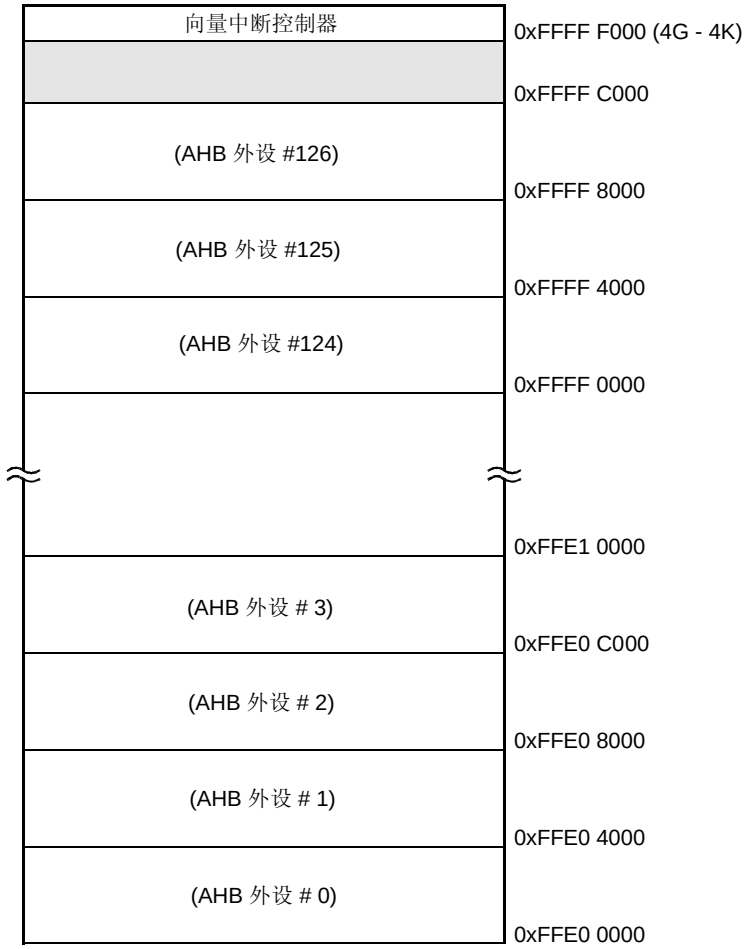

向量中断控制器
0xFFFF F000 (4G - 4K)
0xFFFF C000
(AHB 外设 #126)
0xFFFF 8000
(AHB 外设 #125)
0xFFFF 4000
(AHB 外设 #124)
0xFFFF 0000
0xFFE1 0000
(AHB 外设 # 3)
0xFFE0 C000
(AHB 外设 # 2)
0xFFE0 8000
(AHB 外设 # 1)
0xFFE0 4000
(AHB 外设 # 0)
0xFFE0 0000

图 5.7 AHB 外设映射

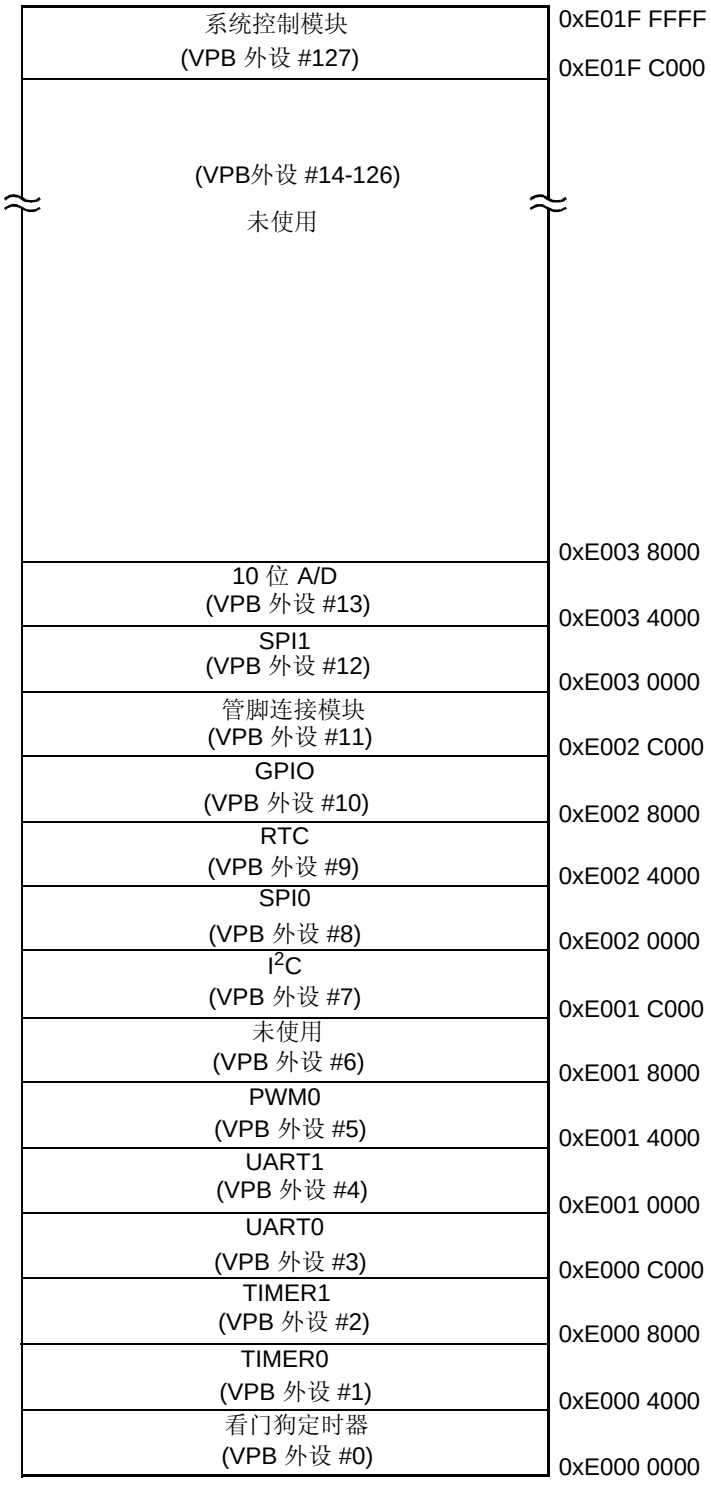


图 5.8 VPB 外设映射

5.3.4 预取指中止和数据中止异常

如果试图访问一个保留地址或未分配区域的地址，LPC2114/2124/2210/2212/2214 将产生预取指中止或数据中止异常。这些区域包括：

- 特定的 ARM 器件所没有的存储器映射区域。对于 LPC2114/2124/2210/2212/2214，它们是：

  ——片内非易失性存储器与片内 SRAM 之间的地址空间，在图 5.5 和图 5.9 中标为

“保留给片内存储器”。对于无 Flash 的器件来说，它们是地址范围从 0x00000000 到 0x3FFFFFFF。对于 128kB Flash 器件来说，它们是 0x00020000 到 0x3FFFFFFF 的存储器地址空间；而对于 256kB Flash 器件来说，它们是 0x00040000 到 0x3FFFFFFF 的存储器地址空间。

——片内静态 RAM 与外部存储器之间的地址空间，在图 5.5 中标为“保留给片内存储器”。地址范围从 0x40003FFF 到 0x7FFFDFFF。

——外部存储器，但由 LPC2210/2212/2214 中 EMC 提供的除外。

——AHB 和 VPB 空间的保留区域，见图 5.6。

- 未分配的 AHB 外设空间，见图 5.7。
- 未分配的 VPB 外设空间，见图 5.8。

对于这些区域，对数据的访问和对指令的取指都会产生异常。此外，对 AHB 或 VPB 外设地址执行任何指令取指都会导致产生预取指中止异常。

在现有的 VPB 外设地址空间内，对未定义地址的访问不会产生数据中止异常。每个外设内的地址译码被限制为外设内部需要判别的已定义寄存器。例如，对地址 0xE000D000（UART0 空间内一个未定义的地址）的访问可能导致对定义在地址 0xE000C000 处的寄存器进行访问。一个外设内的这样一种地址混淆在 LPC2114/2124/2210/2212/2214 文档中没有定义，并且它也不是一个被 LPC2114/2124/2210/2212/2214 支持的特性。

需要注意的是，只有在试图执行从非法地址取指的指令时，ARM 才会将预取指中止标志与相关的指令（没有意义的指令）一起保存到流水线并对中止进行处理。当代码在非常靠近存储器边界执行时，这样防止了由预取指所导致的意外中止。

5.3.5 存储器重映射及引导块

1. 存储器映射概念和操作模式

LPC2114/2124/2210/2212/2214 的存储器映射的基本概念是：每个存储器组在存储器映射中都有一个“物理上的”位置。它是一个地址范围，该范围内可写入程序代码。每一个存储器空间的容量都永久固定在同一个位置，这样就不需要将代码设计成在不同地址范围内运行。

由于 ARM7 处理器上的中断向量位置（地址 0x00000000~0x0000001C，见表 5.4），Boot Block 和 SRAM 空间的一小部分需要重新映射来实现在不同操作模式下对中断的使用，见表 5.5。中断的重新映射通过存储器映射控制特性来实现。

表 5.4 ARM 异常向量位置

地址	异常
0x0000 0000	复位
0x0000 0004	未定义指令
0x0000 0008	软件中断
0x0000 000C	预取指中止（从存储器取指出错）
0x0000 0010	数据中止（访问存储器数据出错）
0x0000 0014	保留 *
0x0000 0018	IRQ
0x0000 001C	FIQ

*: 在 ARM 文档中标识为保留，该位置被 Boot 装载程序用作有效的用户程序关键字。通过定义此保留字的值(使用 DCD 指令定义)，使向量表所有数据 32 位累加和为零(0x00000000～0x0000001C 的 8 个字的机器码累加)，才能脱机运行用户程序，这是 LPC2114/2124/2212/2214 的特性。

表 5.5 LPC2114/2124/2210/2212/2214 存储器映射模式

模式	激活	用途
Boot 装载程序模式	由任何复位硬件激活	在任何复位后都会执行 Boot 装载程序。Boot Block 中断向量映射到存储器的底部以允许处理异常并在 Boot 装载过程中使用中断。
用户 Flash 模式	由 Boot 代码软件激活	当在存储器中识别了一个有效的用户程序标识并且 Boot 装载操作未被执行时，由 Boot 装载程序启动。中断向量没有重新映射，它位于 Flash 存储器的底部。
用户 RAM 模式	由用户程序软件激活	由用户程序激活。中断向量重新映射到静态 RAM 的底部。
用户外部模式	复位时 BOOT1:0 不为 11 时激活	当一个或两个 BOOT 引脚在 RESET 低电平结束时为低时由 Boot 装载程序激活。中断向量从外部存储器映射的底部重新映射。 注：该模式只适用于 LPC2210/2212/2214！

对于 LPC2210/2212/2214，当 $\overline{\text{RESET}}$ 为低时，BOOT1:0 脚的状态控制着引导和初始操作。如果引脚悬空，引脚的内部上拉可保证它的高电平状态。设计者可通过连接一些弱下拉电阻（4.7kΩ）或晶体管（ $\overline{\text{RESET}}$ 为低时可驱动为低电平）到 BOOT1:0 脚来选择引导方式，如表 5.6 所示。

表 5.6 BOOT1:0 的引导控制

P2.27/D27/BOOT1	P2.26/D26/BOOT0	引导方式
0	0	CS0 控制的 8 位存储器
0	1	CS0 控制的 16 位存储器
1	0	CS0 控制的 32 位存储器
1	1	内部 Flash 存储器

2. 存储器的重新映射

为了与将来器件相兼容，整个 Boot Block 都被映射到片内存储器空间的顶端。在这种方式下，使用较大或较小的 Flash 模块都不需要改变 Boot Block 的位置或改变 Boot Block 中断向量的映射。**除了中断向量之外的存储器空间都保持固定的位置。**图 5.9 所示为使用上述定义的模式映射的片内存储器。

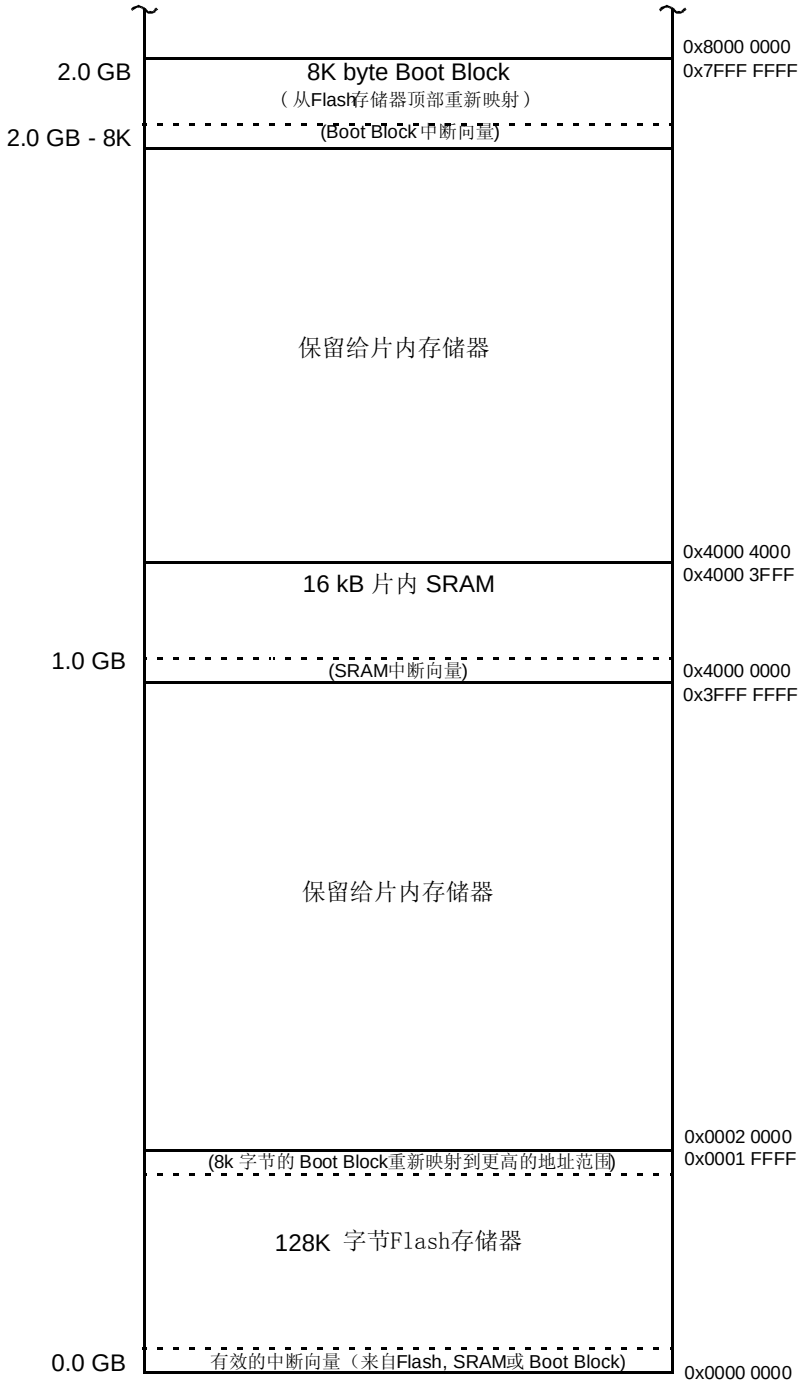
存储器重新映射的部分允许在不同模式下处理中断，它包括中断向量区（32 字节）和额外的 32 字节，一共是 64 字节。重新映射的代码位置与地址 0x00000000~0x0000003F 重叠。一个位于 Flash 存储器中的典型用户程序可以将整个 FIQ 处理程序放置在地址 0x0000001C 而不需要考虑存储器的边界。包含在 SRAM、外部存储器和 Boot Block 中的向量必须包含跳转到实际中断处理程序的分支或者其它执行跳转到中断处理程序的转移指令。

使用存储器的重新映射有以下三个原因：

- 使 Flash 存储器中的 FIQ 处理程序不必考虑因为重新映射所导致的存储器边界问题。
- 用来处理代码空间中段边界仲裁的 SRAM 和 Boot Block 向量的使用大大减少。
- 为超过单字转移指令范围的跳转提供空间来保存常量

重新映射的存储器组，包括 Boot Block 和中断向量，除了重新映射的地址外，仍然继续出现在它们最初的位置。

有关重新映射及其举例详见第 5.4 节。



注：存储器组并不是按比例绘制的。

图 5.9 显示已重新映射和可重新映射区域的低端存储器空间

5.3.6 启动代码相关部分

在一般 32 位 ARM 应用系统中，软件大多数采用 C 语言进行编程，并且以嵌入式操作系统为开发平台，这样就大大的提高了开发效率及软件性能。为了能够进行系统初始化，采用一个汇编文件作启动代码是常用的做法，它可以实现向量表定义、堆栈初始化、系统变量初始化、中断系统初始化、I/O 初始化、外围初始化、地址重映射等操作。ARM 公司只设计核心，不自己生产芯片，只是把核心授权给其它厂商，其它厂商购买了授权后加入自己的外设后生产出各具特色的芯片。这样促进了基于 ARM 处理器核的芯片就多元化，但也使得

每一种芯片的启动代码差别很大，不易编写出统一的启动代码。ADS 的策略是不提供完整的启动代码。启动代码不足部分或者由厂商提供，或者自己编写。启动代码与芯片的特性有着紧密的联系，后面的章节中将根据芯片的特性分别介绍 LPC2100、LPC2200 的启动代码。

ARM 芯片复位后，系统进入管理模式，ARM 状态，PC(R15)寄存器值为 0x00000000，所以必须保证用户的向量表代码定位在 0x00000000 处，或者映射到 0x00000000 处(比如，向量表代码在 0x80000000 处，通过存储器映射，访问 0x0000000 就是访问 0x800000000)。LPC2114/2124/2210/2212/2214 的启动代码中，向量表的定义如程序清单 5.1 所示(在 startup.s 文件中)。

程序清单 5.1 异常向量表

```
                CODE32
                AREA    vectors,CODE,READONLY
                ENTRY
Reset
                LDR     PC, ResetAddr                   (1)
                LDR     PC, UndefinedAddr               (2)
                LDR     PC, SWI_Addr                    (3)
                LDR     PC, PrefetchAddr                (4)
                LDR     PC, DataAbortAddr               (5)
                DCD     0xb9205f80                      (6)
                LDR     PC, [PC, #-0xff0]               (7)
                LDR     PC, FIQ_Addr                    (8)

ResetAddr       DCD     ResetInit                       (9)
UndefinedAddr   DCD     Undefined                       (10)
SWI_Addr        DCD     SoftwareInterrupt               (11)
PrefetchAddr    DCD     PrefetchAbort                   (12)
DataAbortAddr   DCD     DataAbort                       (13)
Nouse           DCD     0                               (14)
IRQ_Addr        DCD     0                               (15)
FIQ_Addr        DCD     FIQ_Handler                     (16)
```

向量从上到下依次为复位（程序清单 5.1 (1)）、未定义指令异常（程序清单 5.1 (2)）、软件中断（程序清单 5.1 (3)）、预取令中止（程序清单 5.1 (4)）、预取数据中止（程序清单 5.1 (5)）、保留的异常（程序清单 5.1 (6)）、IRQ（程序清单 5.1 (7)）和 FIQ（程序清单 5.1 (8)）。使用 LDR 指令跳转而不使用 B 指令跳转的原因有如下 2 个：

- LDR 指令可以全地址范围跳转而 B 指令不行；
- 芯片具有 ReMap 功能。当向量表位于 RAM 中时，用 B 指令不能跳转到正确的位置。

LPC2100、LPC2200 的工程模板(工程模板包含了启动代码及编译连接配置，适用于 ADS1.2 集成开发环境)使用了 ADS 的分散加载机制，只要编辑相应的分散加载描述文件(比如：mem_a.scf，mem_b.scf，mem_c.scf)，即可将代码段、数据段分别定位到指定地址上。程序清单 5.2 是 LPC2200 的工程模板中，使用片外 FLASH 启动程序的分散加载描述文件(在 mem_a.scf 文件中)。

程序清单 5.2 片外 FLASH 启动程序的分散加载描述文件

```
ROM_LOAD   0x80000000                                    (1)
{
    ROM_EXEC 0x80000000                                  (2)
    {
        Startup.o (vectors, +First)                      (3)
        * (+RO)                                          (4)
    }
    …
}
```

其中，程序清单 5.2 (1)，ROM_LOAD 为加载区的名称，其后面的 0x80000000 表示加载区的起始地址（存放程序代码），也可以在后面添加其空间大小，如“ROM_LOAD 0x80000000 0x20000”。程序清单 5.2 (2)，ROM_EXEC 描述了执行区的地址，放在第一块定义，其起始地址、空间大小与加载区起始地址、空间大小要一致。程序清单 5.2 (3、4)，从起始地址开始放置向量表(即“Startup.o (vectors, +First)”，其中 startup.o 为 startup.s 的目标文件)，接着放置其它代码(即“* (+RO)”)。这样即可把向量表定义到 0x80000000 处，LPC2210/2212/2214 芯片复位时若 BOOT1:0 引脚不为 11，则会自动 0x80000000～0x8000003F 映射到 0x00000000～0x0000003F，实现程序的启动。

复位初始化程序 ResetInit(在 startup.s 文件中)如程序清单 5.3 所示，调用 InitStack 子程序(在 startup.s 文件中)来初始化各个模式下的堆栈，调用 TargetResetInit()函数(在 target.c 文件中)来初始化与目标系统相关的设置，最后调用 ADS 提供的__main，初始化运行时库并进入用户的 main()函数。

程序清单 5.3 复位初始化程序

```
ResetInit
            …
            BL      InitStack
            BL      TargetResetInit

            B       __main
```

5.4 系统控制模块

5.4.1 系统控制模块功能汇总

系统控制模块包括几个系统构件和控制寄存器，这些寄存器具有众多与特定外设器件无关的功能。系统控制模块包括：

- 晶体振荡器
- 复位
- 外部中断输入
- 存储器映射控制
- PLL
- VPB 分频器

- 功率控制
- 唤醒定时器

每种类型的功能都有其自身的寄存器，不需要的位则定义为保留位。为了满足将来扩展的需要，无关的功能不共用相同的寄存器地址。

5.4.2 引脚描述

表 5.7 所示为系统控制模块功能相关的引脚。

表 5.7 系统控制模块引脚汇总

引脚名称	引脚方向	引脚描述
X1	输入	**晶振输入** 振荡器和内部时钟发生器电路的输入
X2	输出	**晶振输出** 振荡器放大器的输出
$\overline{\text{RESET}}$	输入	**外部复位输入** 该引脚上的低电平将使芯片复位，使 I/O 口和外设恢复其默认状态，并使处理器从地址 0 开始执行程序。
EINT0	输入	**外部中断 0 输入** 该引脚可用于将处理器从空闲或掉电模式中唤醒。P0.1、P0.16 引脚可设置为 EINT0 功能。
EINT1	输入	**外部中断 1 输入** 该引脚可用于将处理器从空闲或掉电模式中唤醒。P0.3、P0.14 引脚可设置为 EINT1 功能。
EINT2	输入	**外部中断 2 输入** 该引脚可用于将处理器从空闲或掉电模式中唤醒。P0.7、P0.15 引脚可设置为 EINT2 功能。
EINT3	输入	**外部中断 3 输入** 该引脚可用于将处理器从空闲或掉电模式中唤醒。P0.9、P0.20、P0.30 引脚可设置为 EINT3 功能。

5.4.3 寄存器描述

系统控制模块寄存器汇总见表 5.8，所有寄存器不管规格大小都以字地址作为边界。这些寄存器的详细信息见相关功能的描述。

表 5.8 系统控制寄存器汇总

名称	描述	访问	复位值*	地址
外部中断				
EXTINT	外部中断标志寄存器	R/W	0	0xE01FC140
EXTWAKE	外部中断唤醒寄存器	R/W	0	0xE01FC144
EXTMODE	外部中断方式寄存器	R/W	0	0xE01FC148
EXTPOLAR	外部中断极性寄存器	R/W	0	0xE01FC14C
存储器映射控制				
MEMMAP	存储器映射控制	R/W	0	0xE01FC040
锁相环				
PLLCON	PLL 控制寄存器	R/W	0	0xE01FC080
PLLCFG	PLL 配置寄存器	R/W	0	0xE01FC084
PLLSTAT	PLL 状态寄存器	RO	0	0xE01FC088
PLLFEED	PLL 馈送寄存器	WO	NA	0xE01FC08C
功率控制				
PCON	功率控制寄存器	R/W	0	0xE01FC0C0

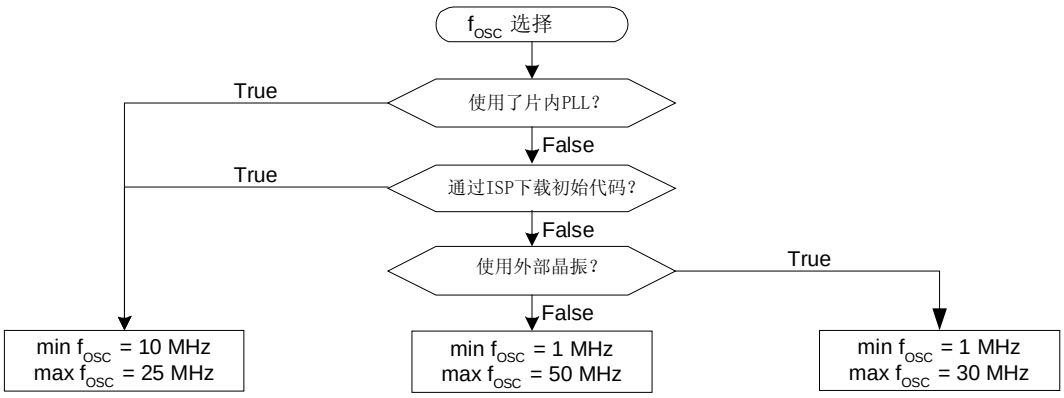
接上表

名称	描述	访问	复位值*	地址
PCONP	外设功率控制	R/W	0x3BE	0xE01FC0C4
VPB 分频器				
VPBDIV	VPB 分频器控制	R/W	0	0xE01FC100

*: 复位值仅指已使用位中保存的数据，不包括保留位的内容。

5.4.4 晶体振荡器

对于 LPC2114/2124/2210/2212/2214，如果是从 XTAL1 脚输入占空比因数为 50-50 的时钟信号，时钟频率在 1MHz～50MHz 范围内；如果是使用外部晶体，微控制器的内部振荡电路仅支持 1MHz～30MHz 的外部晶体。如果需要使用片内 PLL 系统或引导装载程序(即 ISP 功能)，输入时钟频率将被限制到 10MHz～25MHz。图 5.10 为选择振荡器的流程图。



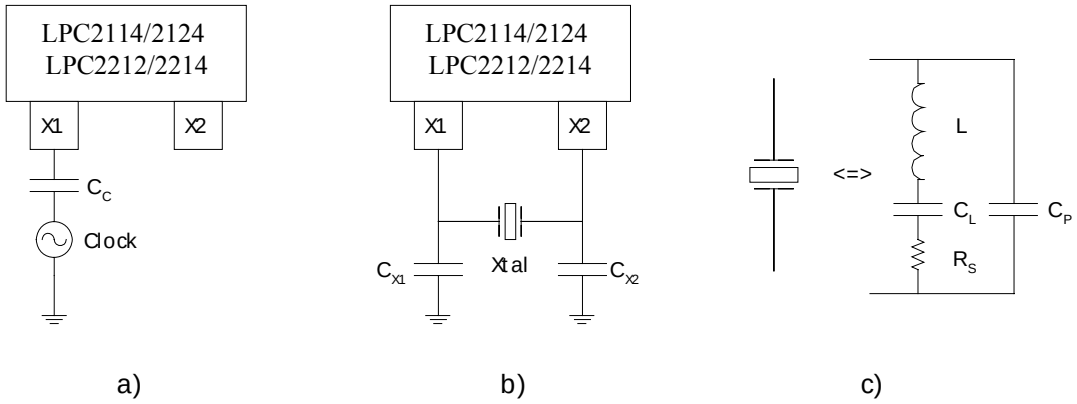
(图 5.11，模式 a 或 b) (图 5.11，模式 a) (图 5.11，模式 b)

图 5.10 Fosc 的选择

振荡器输出频率称为 F_{OSC}。为了便于频率等式的书写及本文档的描述，ARM 处理器时钟频率称为 cclk。除非 PLL 运行并连接，否则 F_{OSC} 和 cclk 的值相同。

LPC2114/2124/2210/2212/2214 的振荡器可工作在两种模式下：从属模式和振荡模式。

- **从属模式**，如图 5.11 中 a 图所示，输入时钟信号与一个 100pF（图 5.11 的 Cc）相连，其幅值不少于 200mVrms，X2 引脚不连接。如果选择从属模式，Fosc 信号（占空比因数为 50-50）的频率被限制在 1MHz～50MHz。
- **振荡模式**，使用的外部元件和模型见图 5.11 中的 b 和 c 图以及表 5.9。由于片内集成了反馈电阻，只需在外部连接一个晶体和电容 Cx1、Cx2 就可形成基本模式的振荡（基本频率用 L、C_L 和 Rs 来表示）。图 5.11 中 c 图的电容 Cp 是并联封装电容，其值不能大于 7pF。参数 Fc、C_L、Rs 和 Cp 都由晶体制造商提供。



a)从属模式，b)振荡模式，c）外部晶体模型（用来评估 $C_{X1/X2}$ 的值）

图 5.11 振荡器模式和模型

表 5.9 振荡模式下 $C_{X1/X2}$ 的建议取值（晶体和外部元件参数）

基本振荡频率 Fc	晶体负载电容 CL	最大晶体串联电阻 Rs	外部负载 电容 Cx1，Cx2
1～5MHz	10pF	n.a.	n.a.
	20pF	n.a.	n.a.
	30pF	<300Ω	58pF，58pF
5～10MHz	10pF	<300Ω	18pF，18pF
	20pF	<300Ω	38pF，38pF
	30pF	<300Ω	58pF，58pF
10～15MHz	10pF	<300Ω	18pF，18pF
	20pF	<220Ω	38pF，38pF
	30pF	<140Ω	58pF，58pF
15～20MHz	10pF	<220Ω	18pF，18pF
	20pF	<140Ω	38pF，38pF
	30pF	<80Ω	58pF，58pF
20～25MHz	10pF	<160Ω	18pF，18pF
	20pF	<90Ω	38pF，38pF
	30pF	<50Ω	58pF，58pF
25～30MHz	10pF	<130Ω	18pF，18pF
	20pF	<50Ω	38pF，38pF
	30pF	n.a.	n.a.

5.4.5 复位

1. 描述

LPC2114/2124/2210/2212/2214 有两个复位源：$\overline{\text{RESET}}$ 引脚和看门狗复位。$\overline{\text{RESET}}$ 引脚为施密特触发输入引脚，带有一个额外的干扰滤波器。任何复位源引起的芯片复位都会启动唤醒定时器（详见第 5.4.12 小节的唤醒定时器的描述），复位将保持有效直至外部复位撤除，此时振荡器已正常工作。当计数达到一个固定个数的时钟时，Flash 控制器完成其初始化。

复位、振荡器以及唤醒定时器之间的关系见图 5.12。LPC2114/2124 复位处理流程参考图 5.13，LPC2210/2212/2214 复位处理流程参考图 5.14。

复位干扰滤波器使处理器可以忽略非常短的外部复位脉冲，它决定了 $\overline{\text{RESET}}$ 保证芯片复位所必须保持的最短时间。$\overline{\text{RESET}}$ 一旦有效，只有当晶振运行稳定并且 LPC2114/2124/2210/2212/2214 的 X1 脚上出现适当的信号时才能撤除。**如果晶振子系统使用的是外部晶体，上电后 $\overline{\text{RESET}}$ 脚的信号必须保持 10ms。对于晶振已经稳定运行且 X1 脚上已出现稳定信号时出现的复位，$\overline{\text{RESET}}$ 脚的信号只需保持 300ns。**

当内部复位撤除时，处理器从地址 0 开始运行，此处为从 Boot Block 映射的复位向量。此时所有的处理器和外设寄存器都恢复为默认状态。

芯片复位可以发生在 Flash 编程或擦除操作过程中。Flash 存储器会中断正在进行的操作并使 CPU 复位延迟到内部 Flash 高电压降低后才完成。

2. 复位与电源上电次序

一般说来，各个电源引脚（V_{18},V_3,V_{18A} 和 V_{3A}）的上电是无顺序的。但是，**为了正确地处理复位，所有 V_{18} 脚必须给定有效的电压。**这是因为片内复位电路和振荡器的相关硬件都由它们供电。V_3 脚通过其数字引脚来使能微控制器与外部功能的接口。所以，不供给 V_3 电源并不会影响复位序列，但会妨碍微控制器与外部器件的通信。

3. 外部复位和内部 WDT 复位

外部复位和内部 WDT 复位有一些小的区别。外部复位使特定引脚的值被锁存以实现配置，内部 WDT 复位则无此功能。**在外部复位时对引脚 P1.20/TRACESYNC，P1.26/RTCK，BOOT1 和 BOOT0**（见第 5.2 节、第 5.7 节和第 5.6 节的描述）的状态进行判断，以实现不同的目的。**当复位后执行引导装载程序时，片内引导装载程序将对 P0.14 进行检测**（判断是否运行 ISP 服务程序）。

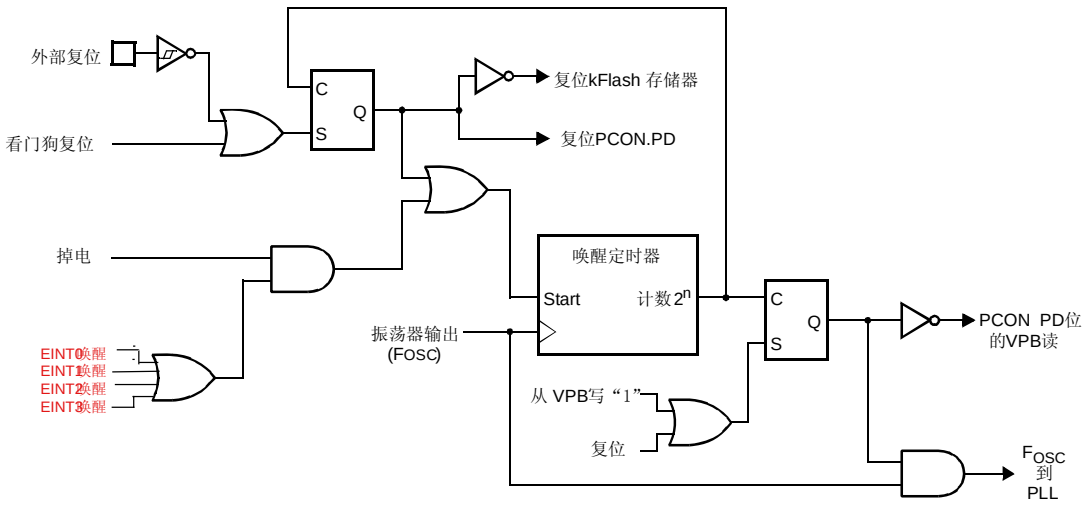


图 5.12 包括唤醒定时器的复位方框图

4. 有效的用户代码

PC2114/2124/2212/2214 规定了“向量表所有数据 32 位累加和为零”作为有效代用户代码条件，也就是说，只有当向量表所有数据 32 位累加和为零时，用户的程序才能脱机运行。通过定义向量表中的保留字(0x00000014 地址)的值(使用 DCD 指令定义)，使向量表所有数据 32 位累加和为零(0x00000000～0x0000001C 的 8 个字的机器码累加)。LPC2100、LPC2200 启动代码的向量表及指令机器码如程序清单 5.4 所示。

程序清单 5.4 向量表及指令机器码

```
Reset
[0xe59ff018]        LDR     PC, ResetAddr
[0xe59ff018]        LDR     PC, UndefinedAddr
[0xe59ff018]        LDR     PC, SWI_Addr
[0xe59ff018]        LDR     PC, PrefetchAddr
[0xe59ff018]        LDR     PC, DataAbortAddr
[0xb9205f80]        DCD     0xb9205f80
[0xe51ffff0]        LDR     PC, [PC, #-0xff0]
[0xe59ff018]        LDR     PC, FIQ_Addr
```

向量表所有数据 32 位累加和：

0xe59ff018 + 0xe59ff018 +0xe59ff018 + 0xe59ff018 + 0xe59ff018 + 0xb9205f80 + 0xe51ffff0 + 0xe59ff018 = 0x00000000

向量表中的保留字的值计算(其中“~”为取反码)：

~(0xe59ff018 + 0xe59ff018 +0xe59ff018 + 0xe59ff018 + 0xe59ff018 + 0xe51ffff0 + 0xe59ff018) + 1 = 0xb9205f80

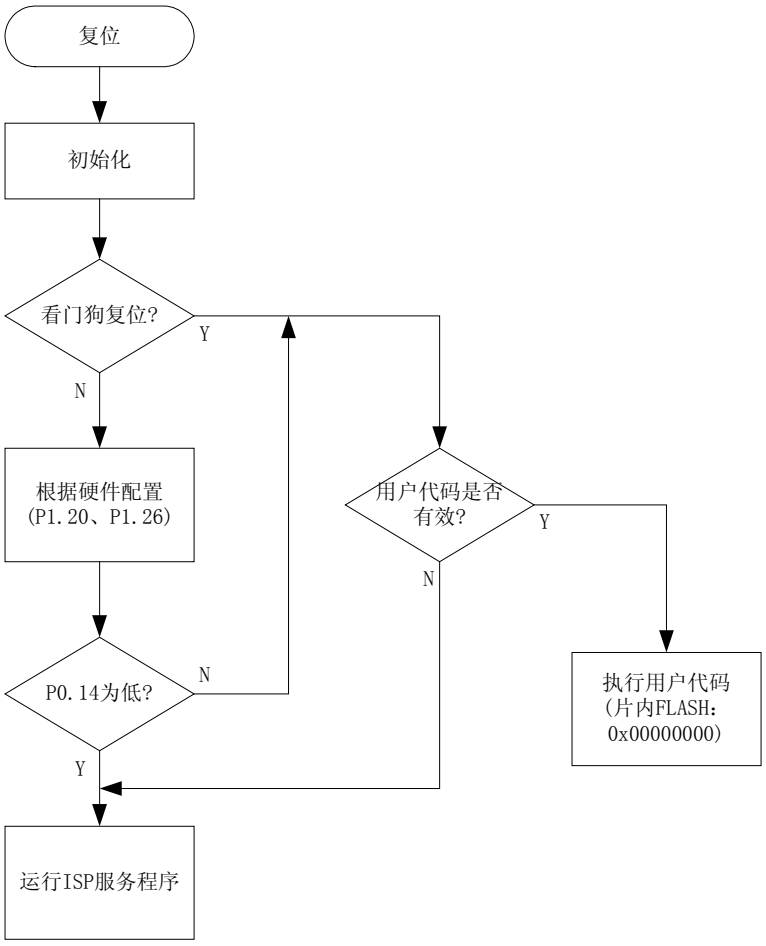


图 5.13 LPC2114/2124 复位处理流程

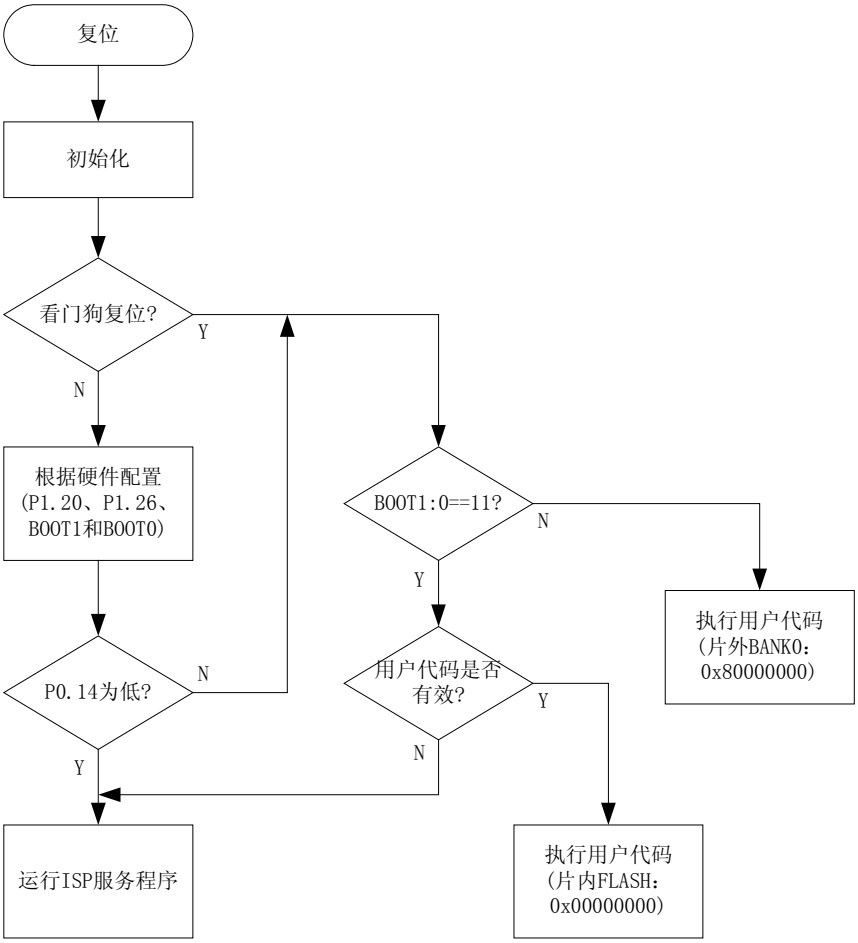


图 5.14 LPC2210/2212/2214 复位处理流程

5.4.6 外部中断输入

LPC2114/2124/2210/2212/2214 含有 4 个外部中断输入（作为可选的引脚功能，即可以通过 PINSEL0/1 寄存器设置相应管脚为外部中断功能）。外部中断输入可用于将处理器从掉电模式唤醒。

逻辑结构

外部中断的逻辑原理图见图 5.15。外部中断逻辑获取的 EINTi 信号，用来控制处理器从掉电模式唤醒。

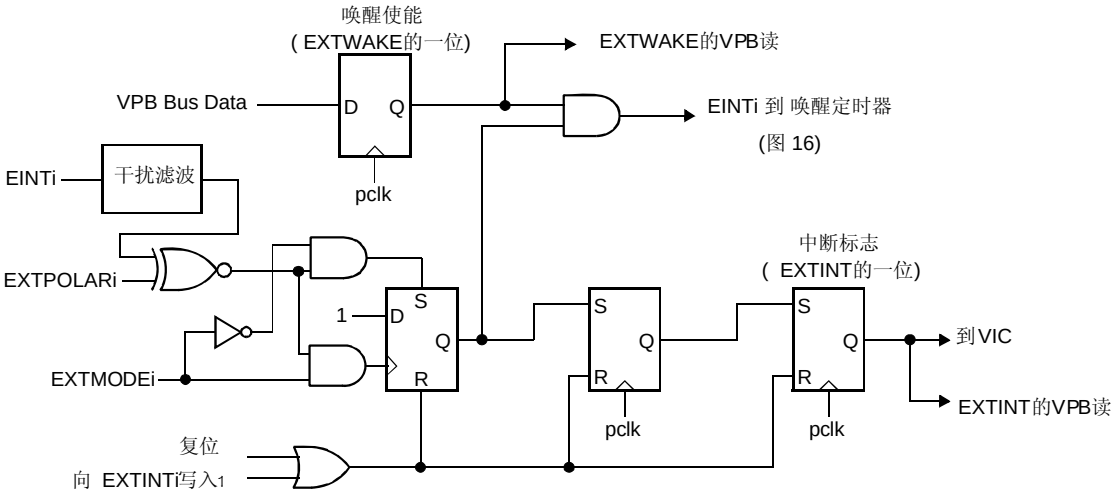


图 5.15 外部中断逻辑

寄存器汇总

外部中断功能具有 4 个相关的寄存器，如表 5.10 所示。EXTINT 寄存器包含中断标志；EXTWAKEUP 寄存器包含使能唤醒位，可使能独立的外部中断输入将处理器从掉电模式唤醒；EXTMODE 和 EXTPOLAR 寄存器用来指定引脚使用电平或边沿触发方式。

表 5.10 外部中断寄存器

地址	名称	描述	访问
0xE01FC140	EXTINT	外部中断标志寄存器包含 ENIT0,EINT1,EINT2 和 EINT3 的中断标志。见表 5.11。	R/W
0xE01FC144	EXTWAKE	外部中断唤醒寄存器包含 3 个用于控制外部中断是否将处理器从掉电模式唤醒的使能位，见表 5.12。	R/W
0xE01FC148	EXTMODE	外部中断方式寄存器控制每个引脚的边沿或电平触发中断。	R/W
0xE01FC14C	EXTPOLAR	外部中断极性寄存器控制由每个引脚的哪种电平或边沿来触发中断。	R/W

外部中断标志寄存器（EXTINT - 0xE01FC140）

当一个引脚选择使用外部中断功能时(通过设置 PINSEL0/1 寄存器实现)，若引脚上出现了对应于 EXTPOLAR 和 EXTMODE 寄存器设置的电平或边沿信号，EXTINT 寄存器中的中断标志将被置位。然后向 VIC 提出中断请求，如果这个外部中断已使能，则产生中断。

通过向 EXTINT 寄存器的 EINT0～EINT3 位写入 1 来将其清零。电平触发方式下，该操作只有在引脚处于无效状态时才有效，比如设置为低电平中断，则只有在中断引脚恢复为高电平后才能清除中断标志。

EXTINT 寄存器描述见表 5.11。

表 5.11 外部中断标志寄存器

EXTINT	功能	描述	复位值
0	EINT0	电平触发方式下，如果引脚的 EINT0 功能被选用且引脚处于有效状态时，该位置位；边沿触发方式下，如果引脚的 EINT0 功能被选用且引脚上出现所选边沿，该位置位。 该位通过写入 1 清除，但电平触发方式下引脚处于有效状态的情况除外。	0

接上表

EXTINT	功能	描述	复位值
1	EINT1	电平触发方式下，如果引脚的 EINT1 功能被选用且引脚处于有效状态时，该位置位；边沿触发方式下，如果引脚的 EINT1 功能被选用且引脚上出现所选边沿，该位置位。 该位通过写入 1 清除，但电平触发方式下引脚处于有效状态的情况除外。	0
2	EINT2	电平触发方式下，如果引脚的 EINT2 功能被选用且引脚处于有效状态时，该位置位；边沿触发方式下，如果引脚的 EINT2 功能被选用且引脚上出现所选边沿，该位置位。 该位通过写入 1 清除，但电平触发方式下引脚处于有效状态的情况除外。	0
3	EINT3	电平触发方式下，如果引脚的 EINT3 功能被选用且引脚处于有效状态时，该位置位；边沿触发方式下，如果引脚的 EINT3 功能被选用且引脚上出现所选边沿，该位置位。 该位通过写入 1 清除，但电平触发方式下引脚处于有效状态的情况除外。	0
7:4	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

外部中断唤醒寄存器（EXTWAKE - 0xE01FC144）

EXTWAKE 寄存器中的使能位允许相应的外部中断将处理器从掉电模式唤醒。相关的 EINTn 功能必须连接到引脚才能实现掉电唤醒功能。实现掉点唤醒不需要(在向量中断控制器中)使能相应的中断，这样做的好处是允许外部中断输入将处理器从掉电模式唤醒，但不产生中断（只是简单地恢复操作）。

EXTWAKE 寄存器描述见表 5.12。

表 5.12 外部中断唤醒寄存器

EXTWAKE	功能	描述	复位值
0	EXTWAKE0	该位为 1 时，使能 $\overline{\text{EINT0}}$ 将处理器从掉电模式唤醒。	0
1	EXTWAKE1	该位为 1 时，使能 $\overline{\text{EINT1}}$ 将处理器从掉电模式唤醒。	0
2	EXTWAKE2	该位为 1 时，使能 $\overline{\text{EINT2}}$ 将处理器从掉电模式唤醒。	0
3	EXTWAKE3	该位为 1 时，使能 $\overline{\text{EINT3}}$ 将处理器从掉电模式唤醒。	0
7:4	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

外部中断方式寄存器（EXTMODE – 0xE01FC148）

EXTMODE 寄存器中的位用来选择每个 EINT 脚是电平或边沿触发。只有选择用作 EINT 功能（见第 5.7 节）的引脚，并已通过 VICIntEnable（见第 5.8 节）使能相应中断，才能产生外部中断。

EXTMODEM 寄存器描述见表 5.13。

表 5.13 外部中断方式寄存器

EXTMODE	功能	描述	复位值
0	EXTMODE0	该位为 0 时，EINT0 使用电平触发；该位为 1 时，EINT0 使用边沿触发。	0
1	EXTMODE1	该位为 0 时，EINT1 使用电平触发；该位为 1 时，EINT1 使用边沿触发。	0

接上表

EXTMODE	功能	描述	复位值
2	EXTMODE2	该位为 0 时，EINT2 使用电平触发；该位为 1 时，EINT2 使用边沿触发。	0
3	EXTMODE3	该位为 0 时，EINT3 使用电平触发；该位为 1 时，EINT3 使用边沿触发。	0
7:4	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

外部中断极性寄存器（EXTPOLAR – 0xE01FC14C）

在电平触发方式中，EXTPOLAR 寄存器用来选择相应引脚是高电平或低电平有效。在边沿触发方式中，EXTPOLAR 寄存器用来选择引脚上升沿或下降沿有效。只有选择用作 EINT 功能的引脚，并已通过 VICIntEnable 使能相应中断，才能产生外部中断。

EXTPOLAR 寄存器描述见表 5.14。

表 5.14 外部中断极性寄存器

EXTPOLAR	功能	描述	复位值
0	EXTPOLAR0	该位为 0 时，EINT0 低电平或下降沿有效（由 EXTMODE0 决定）。 该位为 1 时，EINT0 高电平或上升沿有效（由 EXTMODE0 决定）。	0
1	EXTPOLAR1	该位为 0 时，EINT1 低电平或下降沿有效（由 EXTMODE1 决定）。 该位为 1 时，EINT1 高电平或上升沿有效（由 EXTMODE1 决定）。	0
2	EXTPOLAR2	该位为 0 时，EINT2 低电平或下降沿有效（由 EXTMODE2 决定）。 该位为 1 时，EINT2 高电平或上升沿有效（由 EXTMODE2 决定）。	0
3	EXTPOLAR3	该位为 0 时，EINT3 低电平或下降沿有效（由 EXTMODE3 决定）。 该位为 1 时，EINT3 高电平或上升沿有效（由 EXTMODE3 决定）。	0
7:4	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

外部中断引脚设置

可以通过软件设置引脚选择寄存器来选择多个引脚为 EINT3～EINT0 功能，每个 EINT3～EINT0 的外部中断逻辑接收与之相连的所有引脚的状态和信号。当多个引脚同时设置为相同外部中断时(比如 P0.1、P0.16 引脚均设置为 EINT0 功能)，根据其方式位和极性位的不同，外部中断逻辑处理如下：

- **低电平触发方式中**，选用 EINT 功能的全部引脚的状态都连接到一个正逻辑与门。
- **高电平触发方式中**，选用 EINT 功能的全部引脚的状态都连接到一个正逻辑或门。
- **边沿触发方式中**，使用 GPIO 端口号最低的引脚，与引脚的极性无关。（边沿触发方式中选择使用多个 EINT 引脚被看作编程出错。）

当多个 EINT 引脚逻辑或时，可在中断服务程序中通过 IO0PIN 和 IO1PIN 寄存器从 GPIO 端口读出引脚状态来判断产生中断的引脚。

5.4.7 外部中断应用示例

把相应引脚设置为外部中断功能时，引脚为输入模式，由于没有内部上拉电阻，用户需要外接一个上拉电阻，确保引脚不会悬空。

1. 初始化 EINT0 为电平中断

设置 EINT0 为电平中断的初始化程序如程序清单 5.5 所示。

程序清单 5.5 EINT0 电平中断初始化

```
PINSEL1 = (PINSEL1&0xFFFFFFFC) | 0x01;
EXTMODE = EXTMODE & 0x0E;
```

2. 初始化 EINT0 为下降沿中断

设置 EINT0 为下降沿中断的初始化程序如程序清单 5.6 所示。

程序清单 5.6 EINT0 下降沿中断初始化

```
PINSEL1 = (PINSEL1&0xFFFFFFFC) | 0x01;
EXTMODE = EXTMODE | 0x01;
EXTPOLAR = EXTPOLAR & 0x0E;
```

3. 清除所有外部中断标志

```
EXTINT = 0x0F;
```

5.4.8 存储器映射控制

存储器映射控制用于改变从地址 0x00000000 开始的中断向量的映射，这就允许了运行在不同存储器空间中的代码对中断进行控制。

存储器映射控制寄存器（MEMMAP – 0xE01FC040）

存储器映射控制寄存器见表 5.15、表 5.16。

表 5.15 MEMMAP 寄存器

地址	名称	描述	访问
0xE01FC040	MEMMAP	存储器映射控制。选择从 Flash Boot Block、用户 Flash 或 RAM 中读取 ARM 中断向量。	R/W

表 5.16 存储器映射控制寄存器

MEMMAP	功能	描述	复位值
1:0	MAP1:0	00: Boot 装载程序模式。中断向量从 Boot Block 重新映射。 01: 用户 Flash 模式。中断向量不重新映射，它位于 Flash 中。 10: 用户 RAM 模式。中断向量从静态 RAM 重新映射。 11: 用户外部存储器模式。中断向量从外部存储器重新映射。该模式仅适用于 LPC2210/2212/2214，LPC2114/2124 使用此项功能时未设定该模式。 **警告**：不正确的设定会导致器件的错误操作。	0
7:2	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

* LPC2114/2124/2210/2212/2214 的 MAP 位的硬件复位值为 00。Boot 装载程序总是在复位后立即运行，该程序会将用户看到的复位值更改。

系统引导与存储器映射

对于 LPC2114/2124，因为没有外部存储器接口，所以只能从片内 FLASH 引导程序运行，所以 MAP1:0 的值为 01。

对于 LPC2210/2212/2214，当 $\overline{\text{RESET}}$ 为低时，BOOT1:0 脚的状态控制着引导方式，见表 5.17。如果某个引脚不连，接收器的内部上拉可保证它的高电平状态。设计者可通过连接一些弱下拉电阻（4.7kΩ）或晶体管（ $\overline{\text{RESET}}$ 为低时可驱动为低电平）到 BOOT1:0 脚来选择相应的引导方式。

表 5.17 BOOT1:0 的引导控制(LPC2210/2212/2214)

P2.27/D27/BOOT1	P2.26/D26/BOOT0	引导方式	MAP1:0
0	0	CS0 控制的 8 位存储器	11
0	1	CS0 控制的 16 位存储器	11
1	0	CS0 控制的 32 位存储器	11
1	1	内部 Flash 存储器	01

存储器映射控制的使用注意事项

存储器映射控制只是从处理 ARM 异常必需的 3 个数据源（即异常向量表，64 字节）中选择一个使用，对于 LPC2210/2212/2214 则有 4 个数据源，如图 5.16 所示。

例如，每当产生一个软件中断请求，ARM 内核就从 0x00000008 处取出 32 位数据。这就意味着当 MEMMAP[1:0]＝10(用户 RAM 模式)时，从 0x00000008 的读数/取指是对 0x40000008 单元进行操作。如果 MEMMAP[1:0]=01（用户 Flash 模式），从 0x00000008 的读数/取指是对片内 Flash 单元 0x00000008 进行操作。当 MEMMAP[1:0]=00（Boot 装载程序模式）时，从 0x00000008 的读数/取指是对 0x7FFFE008 单元的数据进行操作（Boot Block 从片内 Flash 存储器重新映射）。

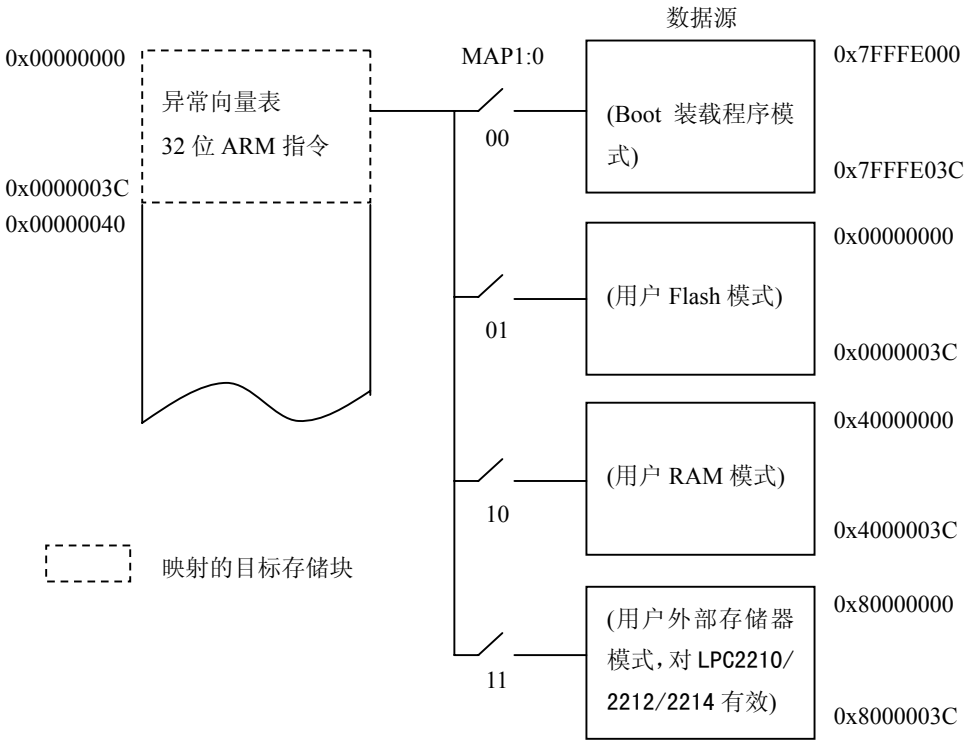


图 5.16 存储器映射控制示意图

REMAP 应用操作

芯片复位时 MEMMAP=0，启动 Boot 装载程序，Boot 装载程序检查 P0.14 口的状态和用户的异常向量表，判断是进入 ISP 状态还是启动用户程序，若启动用户程序，则自动设置

MEMMAP=1(片内 FLASH 启动)或 3(片外程序存储器启动)。若用户程序需要随时更改异常向量表，可以将异常向量表(64 字节)复制片内 RAM 的 0x40000000 地址上，然后设置 MEMMAP=2 进行重新映射，0x40000000 地址上的向量表就可以更改了，复制向量表程序如程序清单 5.7 所示。

当使用片内 RAM 进行调试时，需要设置 MEMMAP=2，使保存在 0x40000000 地址处的异常向量表映射到 0x00000000 地址上。

程序清单 5.7 复制向量表到片内 RAM

```
…
uint8   i;
volatile uint32   *cp1;
volatile uint32   *cp2;

cp1 = uint32(Vectors);
cp2 = 0x40000000;
for(i=0; i<16; i++)
{   *cp2++ = *cp1++;
}
MEMMAP = 2;
…
```

5.4.9 PLL（锁相环）

LPC2114/2124/2210/2212/2214 均具有 PLL 电路，通过 PLL 升频，可以获得更高的系统时钟(cclk)，PLL 的方框图见图 5.17。

PLL 接受的输入时钟频率范围为 10MHz~25MHz，输入频率通过一个电流控制振荡器（CCO）倍增到范围 10MHz~60MHz。倍频器可以是从 1 到 32 的整数（实际上，由于 CPU 最高频率的限制，LPC2114/2124/2210/2212/2214 的倍频值不能高于 6）。CCO 的操作频率范围为 156MHz~320MHz，因此在环中有一个额外的分频器使 PLL 提供所需要的输出频率时 CCO 保持依然在允许频率范围内。输出分频器可设置为 2，4，8 或 16 分频。输出分频器的最小值为 2，保证了 PLL 输出有 50%的占空比。

PLL 的激活由 PLLCON 寄存器控制。PLL 倍频器和分频器的值由 PLLCFG 寄存器控制。为了防止 PLL 参数发生意外改变或 PLL 失效，对这两个寄存器进行了保护。当 PLL 提供芯片时钟时，由于芯片的所有操作，包括看门狗定时器在内都依赖于它，因此 PLL 设置的意外改变将导致 CPU 执行不期望的动作。对它们的保护是由一个类似于操作看门狗定时器的代码序列来实现。详情请参阅 PLLFEED 寄存器的描述。

PLL 在芯片复位和进入掉电模式时被关闭并旁路。PLL 只能通过软件使能。程序必须在配置并激活 PLL 后等待其锁定，然后再连接 PLL。

警告：PLL 值的不正确设定会导致芯片的错误操作。

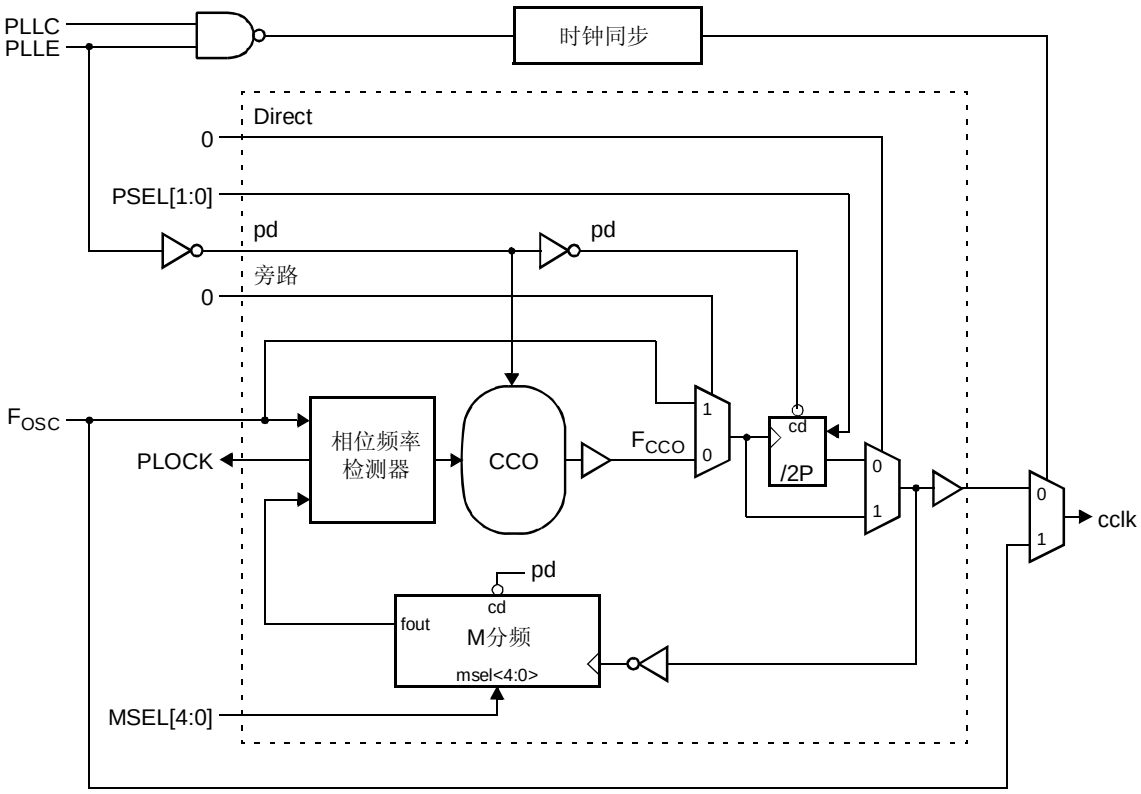


图 5.17 PLL 方框图

1. 寄存器描述

PLL 由表 5.18 所示的寄存器进行控制。

表 5.18 PLL 寄存器

地址	名称	描述	访问
0xE01FC080	PLLCON	PLL 控制寄存器。最新的 PLL 控制位的保持寄存器。写入该寄存器的值在有效的 PLL 馈送序列执行之前不起作用。	R/W
0xE01FC084	PLLCFG	PLL 配置寄存器。最新的 PLL 配置值的保持寄存器。写入该寄存器的值在有效的 PLL 馈送序列执行之前不起作用。	R/W
0xE01FC088	PLLSTAT	PLL 状态寄存器。PLL 控制和配置信息的读回寄存器。如果曾对 PLLCON 或 PLLCFG 执行了写操作，但没有产生 PLL 馈送序列，这些值将不会反映 PLL 的当前状态。**读取该寄存器提供了控制 PLL 和 PLL 状态的真实值。**	RO
0xE01FC08C	PLLFEED	PLL 馈送寄存器。该寄存器使能装载 PLL 控制和配置信息，该配置信息从 PLLCON 和 PLLCFG 寄存器装入实际影响 PLL 操作的映像寄存器。	WO

PLL 控制寄存器（PLLCON – 0xE01FC080）

PLLCON 寄存器包含使能和连接 PLL 的位。使能 PLL 将锁定到当前倍频器和分频器值的设定频率上。连接 PLL 将使处理器和所有片内功能都根据 PLL 输出时钟来运行。对 PLLCON 的更改只有在对 PLLFEED 寄存器执行了正确的 PLL 馈送序列后才生效（见 PLL 馈送寄存器的描述）。

PLLCON 寄存器描述见表 5.19。

表 5.19 PLL 控制寄存器

PLLCON	功能	描述	复位值
0	PLLE	**PLL 使能**。当该位为 1 并且在有效的 PLL 馈送之后，该位将激活 PLL 并允许其锁定到指定的频率。见表 5.21 的 PLLSTAT 寄存器。	0
1	PLLC	**PLL 连接**。当 PLLC 和 PLLE 都为 1 并且在有效的 PLL 馈送后，将 PLL 作为时钟源连接到 CPU。否则，CPU 直接使用振荡器时钟。见表 5.21 的 PLLSTAT 寄存器描述。	0
7:2	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

PLL 在作为时钟源之前必须进行设置、使能并锁定。将振荡器时钟切换到 PLL 输出或反过来操作时，内部电路对操作进行同步以确保不会产生干扰。硬件不能确保 PLL 在连接之前锁定或在 PLL 在失去锁定时自动断开连接。在 PLL 失去锁定的情况下，振荡器很可能已经变得不稳定，这样断开 PLL 也挽救不了这种状况。

PLL 配置寄存器（PLLCFG – 0xE01FC084）

PLLCFG 寄存器包含 PLL 倍频器和分频器值。在执行正确的 PLL 馈送序列之前改变 PLLCFG 寄存器的值不会生效（见 PLL 馈送寄存器 PLLFEED 的描述）。PLL 频率和倍频器以及分频器值的计算详见“PLL 频率计算”一节。

PLLCFG 寄存器描述见表 5.20。

表 5.20 PLL 配置寄存器

PLLCFG	功能	描述	复位值
4:0	MSEL4:0	PLL 倍频器值。在 PLL 频率计算中其值为 M。 注：有关 MSEL4:0 值的正确选取见“PLL 频率计算”。	0
6:5	PSEL1:0	PLL 分频器值。在 PLL 频率计算中其值为 P。 注：有关 PSEL1:0 值的正确选取见“PLL 频率计算”。	0
7	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

PLL 状态寄存器（PLLSTAT - 0xE01FC088）

从 PLLSTAT 寄存器读出的是正在使用的真实 PLL 参数和状态。PLLSTAT 可能与 PLLCON 和 PLLCFG 中的值不同，这是因为没有执行正确的 PLL 馈送序列，这两个寄存器中的值并未生效。

PLLSTAT 寄存器描述见表 5.21。

表 5.21 PLL 状态寄存器

PLLSTAT	功能	描述	复位值
4:0	MSEL4:0	读出的 PLL 倍增器值。这是 PLL 当前使用的值。	0
6:5	PSEL1:0	读出的 PLL 分频器值。这是 PLL 当前使用的值。	0
7	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
8	PLLE	读出的 PLL 使能位。当该位为 1 时，PLL 处于激活状态；为 0 时，PLL 关闭。当进入掉电模式时，该位自动清零。	0

接上表

PLLSTAT	功能	描述	复位值
9	PLLC	读出的 PLL 连接位。当 PLLC 和 PLLE 都为 1 时，PLL 作为时钟源连接到 CPU；当 PLLC 或 PLLE 位为 0 时，PLL 被旁路，CPU 直接使用振荡器时钟。当进入掉电模式时，该位自动清零。	0
10	PLOCK	反映 PLL 的锁定状态。为 0 时，PLL 未锁定；为 1 时，PLL 锁定到指定的频率。	0
15:11	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

PLL 中断：PLLSTAT 寄存器中的 PLOCK 位连接到中断控制器。这样可使用软件打开 PLL，然后继续运行其它程序，不需要等待 PLL 锁定。当发生中断时（PLOCK=1），就可以连接 PLL，再禁止 PLL 中断。

PLL 模式：PLLE 和 PLLC 的组合见表 5.22。

表 5.22 PLL 控制位组合

PLLC	PLLE	PLL 功能
0	0	PLL 被关闭并断开连接。系统使用未更改的时钟输入。
0	1	PLL 被激活但是尚未连接。PLL 可在 PLOCK 置位后连接。
1	0	与 00 组合相同。这样消除了 PLL 已连接但没有使能的可能性。
1	1	PLL 已使能并连接到处理器作为系统时钟源。

PLL 馈送寄存器（PLLFEED – 0xE01FC08C）

必须将正确的馈送序列写入 PLLFEED 寄存器才能使 PLLCON 和 PLLCFG 寄存器的更改生效。馈送序列如下：

1. 将值 0xAA 写入 PLLFEED
2. 将值 0x55 写入 PLLFEED

这两个写操作的顺序必须正确，而且必须是连续的 VPB 总线周期。后面一个要求表明在执行 PLL 馈送操作时必须禁止中断。不管是写入的值不正确还是没有满足前两个条件，对 PLLCON 或 PLLCFG 寄存器的更改都不会生效。

PLLFEED 寄存器描述见表 5.23。

表 5.23 PLL 馈送寄存器

PLLFEED	功能	描述	复位值
7:0	PLLFEED	PLL 馈送序列必须写入该寄存器才能使 PLL 配置和控制寄存器的更改生效。	未定义

2. PLL 和掉电模式

掉电模式会自动关闭并断开 PLL。从掉电模式唤醒不会自动恢复 PLL 的设定，PLL 的恢复必须由软件来完成。通常，首先将 PLL 激活并等待锁定，然后再将 PLL 连接。**有一点非常重要，那就是不要试图在掉电唤醒之后简单地执行馈送序列来重新启动 PLL，因为这会在 PLL 锁定建立之前同时使能并连接 PLL。**

3. PLL 频率计算

PLL 等式使用下列参数：

F_{OSC} 晶振频率

F_{CCO} PLL 电流控制振荡器的频率

cclk PLL 输出频率（也是处理器的时钟频率）

M PLLCFG 寄存器中 MSEL 位的倍增器值

P PLLCFG 寄存器中 PSEL 位的分频器值

PLL 输出频率（当 PLL 激活并连接时）由下式得到：

$cclk = M * F_{OSC}$ 或 $cclk = F_{CCO} / (2*P)$

CCO 频率可由下式得到：

$F_{CCO} = cclk * 2 * P$ 或 $F_{CCO} = F_{OSC} * M * 2 * P$

PLL 输入和设定必须满足下面的条件：

- **F_{OSC} 的范围：10MHz~25MHz**
- **cclk 的范围：10MHz~F_{max}（LPC2114/2124/2210/2212/2214 的最大允许频率）**
- **F_{CCO} 的范围：156MHz~320MHz**

4. 确定 PLL 设定的过程

如果一个特定的应用使用 PLL，它的配置必须依照下面的原则：

- **选择处理器的操作频率（cclk）。**这可以根据处理器的整体要求、UART 波特率的支持等因素来决定。外围器件的时钟频率可以低于处理器频率。
- **选择振荡器频率（F_{OSC}）。**cclk 一定要是 F_{OSC} 的整数倍。
- **计算 M 值以配置 MSEL 位。**M = cclk/F_{OSC}，M 的取值范围为 1~32。写入 MSEL 位的值为 M-1(见表 5.25)。
- **选择 P 值以配置 PSEL 位。**通过设置 P 值，使 F_{CCO} 在定义的频率限制范围内，F_{CCO} 可通过前面的等式计算。P 必须是 1, 2, 4 或 8 其中的一个。写入 PSEL 位的值对应的 P 值见表 5.24。

表 5.24 PLL 分频器值

PSEL 位 PLLCFG[6:5]	P 值
00	1
01	2
10	4
11	8

表 5.25 PLL 倍增器值

MSEL 位 PLLCFG[4:0]	M 值
00000	1
00001	2
00010	3
00011	4
...	...
11110	31
11111	32

5. PLL 设置举例

例如系统要求 Fosc＝10MHz，cclk＝60MHz。

根据这些要求，可得出 M=cclk/Fosc＝60MHz/10MHz＝6。因此，M-1＝5 写入 PLLCFG4:0。

P 值可由 P=Fcco/(cclk*2)得出，Fcco 必须在 156MHz~320MHz 内。假设 Fcco 取最低频率 156MHz，则 P=156MHz/(2*60MHz)＝1.3。Fcco 取最高频率可得出 P=2.67。因此，同时满足 Fcco 最低和最高频率要求的 P 值只能为 2，见表 5.24。所以，PLLCFG[6:5]=01。

5.4.10 VPB 分频器

1. 描述

VPB 分频器决定处理器时钟（cclk）与外设器件所使用的时钟（pclk）之间的关系。VPB 分频器有两个用途，第一个是通过 VPB 总线为外设提供所需的 pclk 时钟以便外设在合适的速度下工作。为了实现此目的，VPB 总线可以降低到 1/2 或 1/4 处理器时钟速率。由于 VPB 总线必须在上电后正常工作（并且如果由于 VPB 分频器控制器位于 VPB 总线上而使上电时 VPB 总线不工作，其时序就不能改变），VPB 总线在复位后默认的状态是以 1/4 速度运行。VPB 分频器的第二个用途是在应用不需要任何外设全速运行时使功耗降低。

VPB 分频器与振荡器和处理器时钟的连接见图 5.18。由于 VPB 分频器连接到 PLL 输出，PLL 在空闲模式下保持有效（如果 PLL 处于运行状态）。

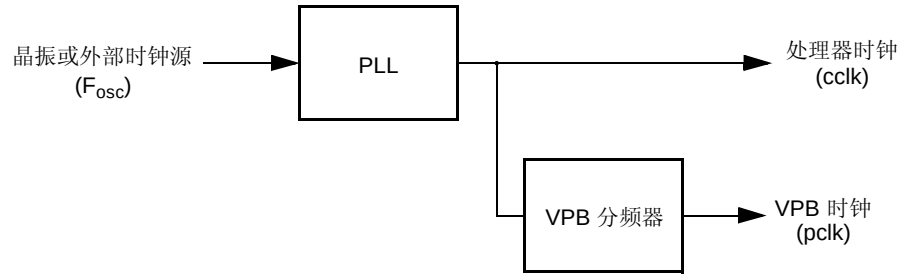


图 5.18 VPB 分频器连接

2. VPBDIV 寄存器（VPBDIV - 0xE01FC100）

VPB 分频器寄存器描述见表 5.26。VPBDIV[1:0]两个位可以设定 3 个分频值，详见表 5.27。XCLKDIV 在 LPC2210/2212/2214 中有效。

表 5.26 VPBDIV 寄存器映射

地址	名称	描述	访问
0xE01FC100	VPBDIV	控制 VPB 时钟速率与处理器时钟之间的关系	R/W

表 5.27 VPBDIV 寄存器

VPBDIV	功能	描述	复位值
1:0	VPBDIV	VPB 时钟速率如下： 00：VPB 总线时钟为处理器时钟的 1/4。 01：VPB 总线时钟与处理器时钟相同。 10：VPB 总线时钟为处理器时钟的 1/2。 11：保留。将该值写入 VPBDIV 寄存器无效（保留原来的设定）。	0

接上表

VPBDIV	功能	描述	复位值
3:2	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	0
5:4	XCLKDIV	这些位仅用于 LPC2210/2212/2214（144 脚封装）中，它们控制着 A23/XCLK 脚上的时钟驱动，取值编码方式与 VPBDIV 相同。由 PINSEL2 寄存器中的一位来控制选择引脚用作 A23 还是 XCLKDIV 选择的时钟功能。 **注：如果 XCLKDIV 和 VPBDIV 取值相同，则 VPB 和 XCLK 使用相同的时钟**。（这在处理 VPB 外设的外部逻辑时可能有用。）	0
7:6	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	0

5.4.11 功率控制

1. 描述

LPC2114/2124/2210/2212/2214 支持两种节电模式：**空闲模式和掉电模式。**

a) 在空闲模式下，指令的执行被挂起直到发生复位或中断为止，但系统时钟 cclk 一直有效。外设功能在空闲模式下继续保持并可产生中断使处理器恢复运行。空闲模式使处理器、存储器系统和相关控制器以及内部总线不再消耗功率。

b) 在掉电模式下，振荡器关闭，这样芯片没有任何内部时钟。处理器状态和寄存器、外设寄存器以及内部 SRAM 值在掉电模式下被保持。芯片引脚的逻辑电平保持静态。复位或特定的不需要时钟仍能工作的中断可终止掉电模式并使芯片恢复正常运行。由于掉电模式使芯片所有的动态操作都挂起，因此芯片的功耗降低到几乎为零。

掉电或空闲模式的进入是与程序的执行同步进行。通过中断唤醒掉电模式不会使指令丢失、不完整或重复。从掉电模式唤醒将在 5.4.12 小节中作进一步讨论。

外设的功率控制特性允许独立关闭应用中不需要的外设，这样进一步降低了功耗。

2. 寄存器描述

功率控制功能包含两个寄存器，如表 5.28 所示。

表 5.28 功率控制寄存器

地址	名称	描述	访问
0xE01FC0C0	PCON	功率控制寄存器。该寄存器包含 LPC2114/2124/2210/2212/2214 两种节电模式的控制位。	R/W
0xE01FC0C4	PCONP	外设功率控制寄存器。该寄存器包含使能和禁止单个外设功能的控制位。该寄存器可使未被使用的外设不消耗功率。	R/W

功率控制寄存器（PCON – 0xE01FC0C0）

PCON 寄存器包含两个位。置位 IDL 位，将会进入空闲模式；置位 PD 位，将会进入掉电模式。如果两位都置位，则进入掉电模式。PCON 寄存器描述见表 5.29。

表 5.29 功率控制寄存器

PCON	功能	描述	复位值
0	IDL	空闲模式－当该位置位时，处理器停止执行程序，但外围功能保持工作状态。外设或外部中断源所产生的任何中断都会使处理器恢复运行。	0

接上表

PCON	功能	描述	复位值
1	PD	掉电模式－当该位置位时，振荡器和所有片内时钟都停止。外部中断所产生的唤醒条件可使振荡器重新启动并使 PD 位清零，处理器恢复运行。	0
7:2	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

外设功率控制寄存器（PCONP – 0xE01FC0C4）

PCONP 寄存器允许将所选的外设功能关闭以实现节电的目的。有少数外设功能不能被关闭（看门狗定时器、GPIO、引脚连接模块和系统控制模块）。PCONP 中的每个位都控制一个外设。每个位所对应的外设编号见 5.3.3 小节的 VPB 外设映射部分。

由于 LPC2210/2212/2214 具有 EMC 模块，而 LPC2112/2114 没有，所以它们的 PCONP 寄存器有点区别，见表 5.30 和表 5.31。

表 5.30 LPC2112/2114 外设功率控制寄存器

PCONP	功能	描述	复位值
0	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	0
1	PCTIM0	该位为 1 时，定时器 0 使能。为 0 时，定时器 0 被关闭以实现节电。	1
2	PCTIM1	该位为 1 时，定时器 1 使能。为 0 时，定时器 1 被关闭以实现节电。	1
3	PCURT0	该位为 1 时，UART0 使能。为 0 时，UART0 被关闭以实现节电。	1
4	PCURT1	该位为 1 时，UART1 使能。为 0 时，UART1 被关闭以实现节电。	1
5	PCPWM0	该位为 1 时，PWM0 使能。为 0 时，PWM0 被关闭以实现节电。	1
6	保留	用户软件不要向其写入 1。从保留位读出的值未被定义。	0
7	PCI2C	该位为 1 时，I^2C 接口使能。为 0 时，I^2C 接口被关闭以实现节电。	1
8	PCSPI0	该位为 1 时，SPI0 接口使能。为 0 时，SPI0 接口被关闭以实现节电。	1
9	PCRTC	该位为 1 时，RTC 使能。为 0 时，RTC 被关闭以实现节电。	1
10	PCSPI1	该位为 1 时，SPI1 接口使能。为 0 时，SPI1 接口被关闭以实现节电。	1
11	保留	用户软件写入 0 来实现节电。	1
12	PCAD	该位为 1 时，A/D 转换器使能。为 0 时，A/D 转换器被关闭以实现节能。	1
31:13	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

表 5.31 LPC2210/2212/2214 的外设功率控制寄存器（PCONP – 0xE01FC0C4）

PCONP	功能	描述	复位值
0	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	0
1	PCTIM0	该位为 1 时，定时器 0 使能。为 0 时，定时器 0 被关闭以实现节电。	1
2	PCTIM1	该位为 1 时，定时器 1 使能。为 0 时，定时器 1 被关闭以实现节电。	1
3	PCURT0	该位为 1 时，UART0 使能。为 0 时，UART0 被关闭以实现节电。	1
4	PCURT1	该位为 1 时，UART1 使能。为 0 时，UART1 被关闭以实现节电。	1
5	PCPWM0	该位为 1 时，PWM0 使能。为 0 时，PWM0 被关闭以实现节电。	1
6	保留	用户软件不要向其写入 1。从保留位读出的值未被定义。	0
7	PCI2C	该位为 1 时，I^2C 接口使能。为 0 时，I^2C 接口被关闭以实现节电。	1
8	PCSPI0	该位为 1 时，SPI0 接口使能。为 0 时，SPI0 接口被关闭以实现节电。	1
9	PCRTC	该位为 1 时，RTC 使能。为 0 时，RTC 被关闭以实现节电。	1
10	PCSPI1	该位为 1 时，SPI1 接口使能。为 0 时，SPI1 接口被关闭以实现节电。	1

接上表

PCONP	功能	描述	复位值
11	PCEMC	该位为 1 时，外部存储器控制器被使能。为 0 时，EMC 被关闭以实现节电。	1
12	PCAD	该位为 1 时，A/D 转换器使能。为 0 时，A/D 转换器被关闭以实现节能。	1
31:13	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

注意：若当前运行的是片外存储器中的程序，不要设置 PCEMC 为 0，否则由于 EMC 关闭，会导致程序运行错误。

3. 功率控制注意事项

复位后，PCONP 的值设置成使能所有接口和外围功能。除了对外围功能相关的寄存器进行配置外，用户应用程序不要访问 PCONP 寄存器以便启动使用片内的任何外围功能。

在需要控制功率的系统中，只要将应用中用到的外围功能的对应在 PCONP 寄存器的位置 1，寄存器的其它“保留”位或当前无需使用的外围功能对应在寄存器中的位都必须清零。

5.4.12 唤醒定时器

描述

唤醒定时器的用途是：确保振荡器和芯片所需要的其它模拟电路在处理器开始执行指令之前能够正确工作。这在所有类型的复位以及任何原因所导致上述功能关闭时非常重要。由于振荡器和其它功能在掉电模式下关闭了，所以处理器从掉电模式中唤醒时必须使用唤醒定时器。

唤醒定时器通过检测晶振是否能可靠地开始代码的执行来对其进行监视。当给芯片加电或某个事件使芯片退出掉电模式，振荡器需要一段时间来产生足够振幅的信号驱动时钟逻辑。时间的长度取决于许多因素，包括 Vdd 的上升速率（上电时）、晶振的类型及其电气特性（如果使用石英晶振）、任何其它外部电路（例如电容）和振荡器在现有环境下自身的特性。

唤醒定时器与时钟的关系

一旦检测到一个时钟，唤醒定时器则对 4096 个时钟计数，这段时间可使 Flash 进行初始化。当 Flash 存储器初始化完毕时，如果外部复位已撤除，处理器开始执行指令。当系统使用外部时钟源时，需要考虑的振荡器的启动延时可能很短甚至没有。唤醒定时器的设计确保了芯片所需要的任何其它功能在程序运行之前都能够进行操作。

总之，LPC2114/2124/2210/2212/2214 唤醒定时器是根据晶振的情况来执行最短时间的复位，它在从掉电模式中唤醒或任何复位产生时激活。

外部中断与唤醒定时器

如果使能了外部中断唤醒功能，并且所选中断事件出现，那么唤醒定时器将被启动。实际的中断（如果有）在唤醒定时器停止后产生，由向量中断控制器（VIC）进行处理。

要使器件进入掉电模式并通过外部中断唤醒，软件应该对引脚的外部中断功能重新编程，选择中断合适的方式和极性，再进入掉电模式。唤醒时软件应恢复引脚复用的外围功能。

如果软件要使器件退出掉电模式来响应多个引脚共用的同一个 EINTi 通道的事件，中断通道必须编程设定为低电平激活方式，因为只有在电平方式中通道才能使信号逻辑“或”来唤醒器件。

5.4.13 启动代码相关部分

在 LPC2100、LPC2200 的启动代码中，target.c 文件包含目标板特殊的代码，包括异常

处理程序和目标板初始化程序，此文件用户要根据程序的需要修改。

为了使系统基本能够工作，必须在进入 main()函数前对系统进行一些基本的初始化工作，这些工作由函数 TargetResetInit()完成(在 target.c 文件中)。LPC2200 的启动代码的 TargetResetInit()的示例见程序清单 5.8。

程序清单 5.8 TargetResetInit()示例—LPC2200

```
void  TargetResetInit(void)
{
#ifdef      __DEBUG
    MEMMAP = 0x3;                    //remap                              (1)
#endif

#ifdef      __OUT_CHIP
    MEMMAP = 0x3;                    //remap                              (2)
#endif

#ifdef      __IN_CHIP
    MEMMAP = 0x1;                    //remap                              (3)
#endif

    /* 设置系统各部分时钟 */
    PLLCON = 1;                                                           (4)
#if  (Fpclk / (Fcclk / 4)) == 1
    VPBDIV = 0;                                                           (5)
#endif
#if  (Fpclk / (Fcclk / 4)) == 2
    VPBDIV = 2;                                                           (6)
#endif
#if  (Fpclk / (Fcclk / 4)) == 4
    VPBDIV = 1;                                                           (7)
#endif

#if   (Fcco / Fcclk) == 2
    PLLCFG = ((Fcclk / Fosc) - 1) | (0 << 5);                             (8)
#endif
#if   (Fcco / Fcclk) == 4
    PLLCFG = ((Fcclk / Fosc) - 1) | (1 << 5);                             (9)
#endif
#if   (Fcco / Fcclk) == 8
    PLLCFG = ((Fcclk / Fosc) - 1) | (2 << 5);                             (10)
#endif
#if   (Fcco / Fcclk) == 16
    PLLCFG = ((Fcclk / Fosc) - 1) | (3 << 5);                             (11)
#endif
```

```
    PLLFEED = 0xaa;                                          (12)
    PLLFEED = 0x55;                                          (13)
    while((PLLSTAT & (1 << 10)) == 0);                       (14)
    PLLCON = 3;                                              (15)
    PLLFEED = 0xaa;                                          (16)
    PLLFEED = 0x55;                                          (17)
    …
}
```

LPC2210/2212/2214具有不同的存储器映射方式，必须根据硬件设置。程序清单5.8 (1)～(3)就是设置存储器映射方式的。当使用我们提供的LPC2200工程模板（for ADS1.2）建立工程时，编译器会根据用户选择的Target项来预定义__DEBUG、__OUT_CHIP和__IN_CHIP三个宏中的一个，而不同的Target代表着不同的工程配置。这样，当配置改变时就无需改动代码。

时钟是芯片各部分正常工作的基础，虽然时钟可以在任何时候设置，但为了避免混乱，最好在进入main()函数前设置（程序清单5.8 (4)～(17)）。这段代码使用友好的接口正确的设置系统个部分时钟，设置方法是在系统配置文件config.h中定义个部分的时钟,例子代码见程序清单5.9。用户按照注释说明的要点设置即可，它们都是芯片的要求。程序首先允许PLL但不连接PLL（程序清单5.8(4)），然后设置外设时钟（VPB时钟pclk）与系统时钟（cclk）的分频比（程序清单5.8 (5)或(6)或(7)）。接着设置PLL的乘因子和除因子（程序清单5.8 (8)或(9)或(10)或(11)）。设置完成后使用（程序清单5.8 (12)、(13)）芯片要求的访问序列把数据确实写入硬件，并等待PLL跟踪完成（程序清单5.8 (14)）。最后，把使能PLL并使PLL联上系统（程序清单5.8 (15)～(17)）。

程序清单 5.9 设置系统时钟

```
/* 系统设置, Fosc、Fcclk、Fcco、Fpclk 必须定义*/
#define   Fosc      11059200        //晶振频率,10MHz~25MHz，应当与实际一至
#define   Fcclk     (Fosc * 4)      //系统频率，必须为 Fosc 的整数倍(1~32)，且<=60MHZ
#define   Fcco      (Fcclk * 4)     //CCO 频率，必须为 Fcclk 的 2、4、8、16 倍，范围为 156MHz~320MHz
#define   Fpclk     (Fcclk / 4) * 1 //VPB 时钟频率，只能为(Fcclk / 4)的 1、2、4 倍
```

值得注意的是 Fcco 并没有联上内核，仅仅是 PLL 的频率，156MHz～320MHz 是 PLL 硬件的振荡频率范围。

5.5 存储器加速模块（MAM）

5.5.1 描述

存储器加速模块（MAM）将下一个需要的 ARM 指令锁存起来，以防止 CPU 取指暂停。MAM 所使用的方法是将 Flash 存储器分成两组，每一组都可独立进行访问，这两个 Flash 组都有自己的预取指缓冲区和分支跟踪缓冲区，如图 5.19 所示。当一个组的预取指缓冲区和分支跟踪缓冲区不能满足指令取指的需要，并且预取指还没有启动时，两个组的分支跟踪缓冲区捕获两个 128 位的 Flash 数据行。在 MAM 启动的预取指周期的结束时，每个预取指缓冲区从它自身的 Flash 组捕获一个 128 位指令行。若关闭 MAM，所有存储器请求都会直接对 Flash 操作。

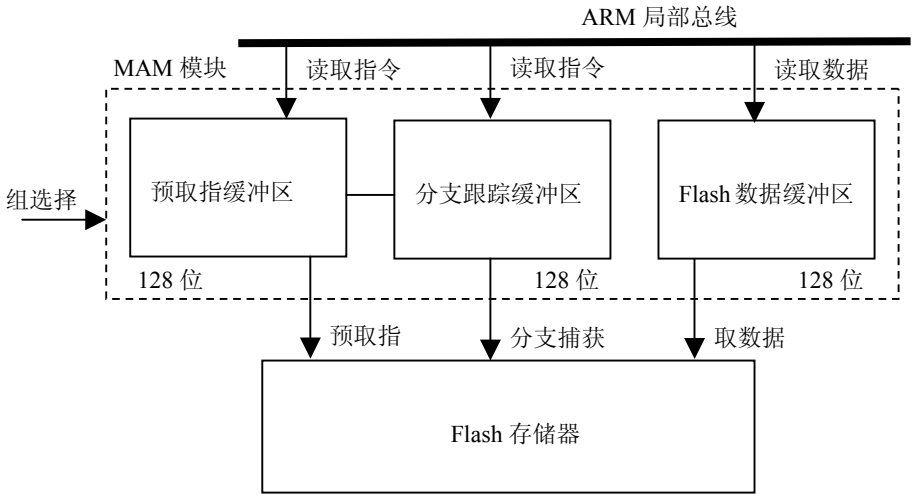


图 5.19 MAM 的一个存储器组连接示意图

每个 128 位值包括了 4 个 32 位 ARM 指令或 8 个 16 位 Thumb 指令。**在连续执行代码时，通常一个 Flash 组包含或者当前正在取指的指令和包含该指令的整个 Flash 行。另一个 Flash 组则包含或正在预取指下一个连续的代码行。**当一个代码行传送完最后一条指令时，包含它的 Flash 组开始对下一行进行取指。

分支和其它程序流的变化会导致前面所讲述的连续指令取指出现中断。当发生回溯分支时，表示很有可能正在执行一个循环，分支跟踪缓冲区有可能已经包含了目标指令。如果是，不需要执行 Flash 读周期就可执行指令。对于一个前向分支，新的地址也有可能包含在其中一个预取指缓冲区中。如果是，那么分支的执行不会有任何延迟。

当分支不在分支跟踪和预取指缓冲区当中时，则需要一个 Flash 访问周期来装载分支跟踪缓冲区。接下来将不再有取指的延迟，除非发生了另一个这样的“指令丢失”。

Flash 存储器控制器检测访问 Flash 存储器的数据并使用一个单独的缓冲区保存结果，采用的方式类似于代码取指时使用的方式。这样就加快了按顺序访问数据的速度。数据访问使用一个单行的缓冲区，和访问代码时提供两个缓冲区不同，因为数据访问不需要预取指功能。

5.5.2 MAM 结构

存储器加速器模块分成以下几个功能块：

- 为每个存储器组提供 Flash 地址锁存。用于 Flash 组 0 地址锁存的增量器功能。
- 两个 Flash 存储器组
- 指令锁存，数据锁存，地址比较锁存
- 等待逻辑

图 5.20 所示为存储器加速器模块数据通路的一个简化框图。

在下面的描述中，“取指”一词表示 ARM 发出的一个直接的 Flash 读请求。“预取指”一词表示对当前处理器取指地址之后的地址执行 Flash 读操作。

1. Flash 存储器组

两个 Flash 存储器组实现了并行访问并消除了连续访问时的延迟。

Flash 编程功能不受存储器加速器模块的控制，而是作为一个独立的功能进行处理。“Boot block”扇区包含作为应用程序的一部分调用的 Flash 编程算法(即 IAP 代码)和一个可对 Flash 存储器进行串行编程的装载程序(即 ISP 代码)。

Flash 存储器的布线使其每个扇区同时存在于两个组当中，这样扇区擦除操作可同时对两个组执行。实际上，两个组的实体对于编程功能是透明的。

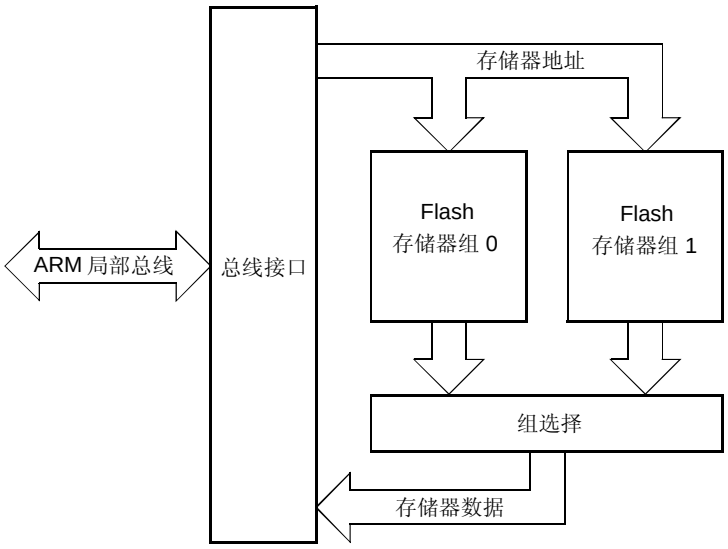


图 5.20 存储器加速器模块的简化框图

2. 指令锁存和数据锁存

代码和数据的访问由存储器加速器模块分别进行处理。每个 Flash 组都由两套 128 位指令锁存和 12 位比较地址锁存。其中一套称为分支跟踪缓冲区，用于保存最近一次指令丢失以来的数据和比较地址。另一套称为预取指缓冲区，用于保存预取指的数据和比较地址。每个指令锁存保存 4 个代码字（4 条 ARM 指令或 8 条 Thumb 指令）。

与之相似，在数据访问中使用了一个 128 位数据锁存和 13 位数据地址锁存。两个 Flash 组共用这一套锁存。对数据锁存中没有的数据进行访问会导致读取 Flash 的 4 个数据字，它们由数据锁存所捕获。使用数据锁存加速了连续数据的访问，但对于随机数据访问几乎没什么效果。

5.5.3 MAM 的操作模式

MAM 定义了 3 种操作模式，您可以在性能和可预测性之间进行选择：

- **MAM 关闭。**所有存储器请求都会导致 Flash 的读操作（表 5.32、表 5.33 的注 2）。无指令预取指。
- **MAM 部分使能。**如果数据可用，则从保持锁存区执行连续的指令访问。指令预取指使能。非连续的指令访问启动 Flash 读操作（表 5.32、表 5.33 的注 2）。这意味着所有的转移指令都会导致对存储器的取指。由于缓冲的数据访问时序很难预测并且非常依赖于所处的状况，因此所有数据操作都会导致 Flash 读操作。
- **MAM 完全使能。**任何存储器请求（代码或数据），如果其值已经包含在其中一个保持锁存当中，那么从缓冲区执行该代码或数据的访问。指令预取指使能。Flash 读操作用于指令的预取指和当前缓冲区所没有的代码或数据的访问。

表 5.32 MAM 响应的不同类型的程序访问

程序存储器请求类型	MAM 模式		
	0	1	2
连续访问，数据位于 MAM 锁存当中	启动取指 [2]	使用锁存的数据 [1]	使用锁存的数据 [1]
连续访问，数据不在 MAM 锁存当中	启动取指	启动取指 [1]	启动取指 [1]
非连续访问，数据位于 MAM 锁存当中	启动取指 [2]	启动取指 [1,2]	使用锁存的数据 [1]
非连续访问，数据不在 MAM 锁存当中	启动取指	启动取指 [1]	启动取指 [1]

表 5.33 MAM 响应的不同类型的数据和 DMA 访问

数据存储器请求类型	MAM 模式		
	0	1	2
连续访问，数据位于 MAM 锁存当中	启动取指 [2]	启动取指 [2]	使用锁存的数据
连续访问，数据不在 MAM 锁存当中	启动取指	启动取指	启动取指
非连续访问，数据位于 MAM 锁存当中	启动取指 [2]	启动取指 [2]	使用锁存的数据
非连续访问，数据不在 MAM 锁存当中	启动取指	启动取指	启动取指

1. 指令预取指在模式 1 和 2 中使能。
2. 只要锁存的数据可用，MAM 则使用锁存的数据，但模仿 Flash 读操作的时序。这样虽然使用相同的执行时序，但却降低了功耗。将 MAMTIM 中的取指时间设置为 1 个时钟可关闭 MAM。

5.5.4 MAM 配置

在复位后，MAM 默认为禁止状态。软件可以随时将存储器访问加速打开或关闭。这样就可使大多数应用程序以最高速度运行，而某些要求更精确定时的功能可以较慢但更可预测的速度运行。

5.5.5 寄存器描述

寄存器汇总

MAM 模块寄存器汇总见表 5.34。

表 5.34 MAM 模块寄存器汇总

名称	描述	访问	复位值*	地址
MAMCR	存储器加速器模块控制寄存器。决定 MAM 的操作模式。也就是说 MAM 性能增强的程度，见表 5.35。	R/W	0	0xE01FC000
MAMTIM	存储器加速器定时控制。决定 Flash 存储器取指所使用的时钟个数(1 到 7 个处理器时钟)。	R/W	0x07	0xE01FC004

* 复位值仅指已使用位中保存的数据，不包括保留位的内容。

MAM 控制寄存器（MAMCR - 0xE01FC000）

两个配置位选择 MAM 的 3 种操作模式，见表 5.35。在复位后，MAM 功能被禁止。改变 MAM 操作模式会导致 MAM 所有的保持锁存内容无效，因此需要执行新的 Flash 读操作。

表 5.35 MAM 控制寄存器

MAMCR	功能	描述	复位值
1:0	MAM 模式控制	这两个位决定 MAM 的操作模式： 00－MAM 功能被禁止 01－MAM 功能部分使能 10－MAM 功能完全使能 11－保留	0
7:2	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

MAM 定时寄存器（MAMTIM - 0xE01FC004）

MAM 定时寄存器决定使用多少个 cclk 周期访问 Flash 存储器，见表 5.36。这样可调整 MAM 时序使其匹配处理器操作频率。Flash 访问时间可以从 1 到 7 个时钟。单个时钟的 Flash 访问实际上关闭了 MAM。这种情况下可以选择 MAM 模式对功耗进行优化。

表 5.36 MAM 定时寄存器

MAMTIM	功能	描述	复位值
2:0	MAM 取指周期	这几个位决定 MAM Flash 取指操作的时间： 000＝0，保留 001＝1，MAM 取指周期为 1 个处理器时钟（cclk）。 010＝2，MAM 取指周期为 2 个处理器时钟（cclk）。 011＝3，MAM 取指周期为 3 个处理器时钟（cclk）。 100＝4，MAM 取指周期为 4 个处理器时钟（cclk）。 101＝5，MAM 取指周期为 5 个处理器时钟（cclk）。 110＝6，MAM 取指周期为 6 个处理器时钟（cclk）。 111＝7，MAM 取指周期为 7 个处理器时钟（cclk）。 警告：不正确的设定会导致器件的错误操作。	0x07
7:3	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

5.5.6 MAM 使用注意事项

MAM 定时值问题

当改变 MAM 定时值时，必须先通过向 MAMCR 写入 0 来关闭 MAM，然后将新值写入 MAMTIM。最后，将需要的操作模式的对应值（1 或 2）写入 MAMCR，再次打开 MAM。

对于低于 20MHz 的系统时钟，MAMTIM 设定为 001。对于 20MHz 到 40MHz 之间的系统时钟，建议将 Flash 访问时间设定为 2cclk，而在高于 40MHz 的系统时钟下，建议使用 3cclk。

Flash 编程问题

在编程和擦除操作过程中不允许访问 Flash 存储器。如果在 Flash 模块忙时存储器请求访问 Flash 地址，MAM 就会强制 CPU 等待（这通过声明 ARM7TDMI-S 局部总线信号 CLKEN 来实现）。在某些情况下，代码执行的延迟会导致看门狗超时。用户必须注意到这种可能性，并采取措施来确保在编程或擦除 Flash 存储器时不会出现非预期的看门狗复位，从而导致系统故障。

为了防止从 Flash 存储器中读取无效的数据，MAM 使锁存在 Flash 编程或擦除操作的开始自动失效。在 Flash 操作结束后，任何对 Flash 地址的读操作将启动新的取指操作。

5.5.7 启动代码相关部分

LPC2100、LPC2200 的启动代码中，会根据 Fcclk 的大小来自动设置 MAM，如程序清单 5.10 所示(在 target.c 文件中)。由于 LPC2210 没有片内 FLASH，MAM 设置无效。

如程序清单 5.10 (1)，首先要将 MAM 功能禁止，然后根据 Fcclk 的大小来设置 MAM 定时寄存器(这是通过条件编译实现，Fcclk 的定义在 config.h 文件中)，最后再使能 MAM(程序清单 5.10 (5))。

程序清单 5.10 TargetResetInit ()—MAM 初始化

```
void   TargetResetInit(void)
{    …
     /* 设置存储器加速模块 */
     MAMCR = 0;                                                    (1)
#if   Fcclk < 20000000
     MAMTIM = 1;                                                   (2)
#else
```

```
#if   Fcclk < 40000000
      MAMTIM = 2;                                                    (3)
#else
      MAMTIM = 3;                                                    (4)
#endif
#endif
      MAMCR = 2;                                                     (5)
      …
}
```

5.6 外部存储器控制器（EMC）

只有 LPC2210、LPC2212 和 LPC2214 含有该模块。

5.6.1 特性

- 支持静态存储器映射器件，包括 RAM、ROM、Flash、Burst ROM 和一些外部 I/O 器件。
- 可对异步(non-clocked)存储器子系统进行异步页模式读操作。
- 可对 Burst ROM 器件进行异步突发模式读访问。
- 4 个存储器组(Bank0～Bank3)可单独配置，每个存储器组可访问 16M 字节空间。
- 总线切换（空闲）周期（1～16 个 CCLK 周期）可编程。
- 可对静态 RAM 器件的读和写 WAIT 状态（高达 32 个 CCLK 周期）进行编程。
- 可编程 Burst ROM 器件的初始和连续读 WAIT 状态。
- 可编程写保护。
- 可编程外部数据总线宽度（8、16 或 32 位）。
- 可编程读字节定位使能控制。

5.6.2 概述

外部静态存储器控制器是一个 AMBA AHB 总线上的从模块，它为 AMBA AHB 系统总线和外部（片外）存储器器件提供了一个接口。该模块可同时支持多达 4 个单独配置的存储器组，每个存储器组都支持 SRAM、ROM、Flash EPROM、Burst ROM 存储器或一些外部 I/O 器件，EMC 与外部存储器连接示意图见图 5.21。每个存储器组的总线宽度为 8、16 或 32 位，但是同一个存储器组不要使用两个不同宽度的器件。

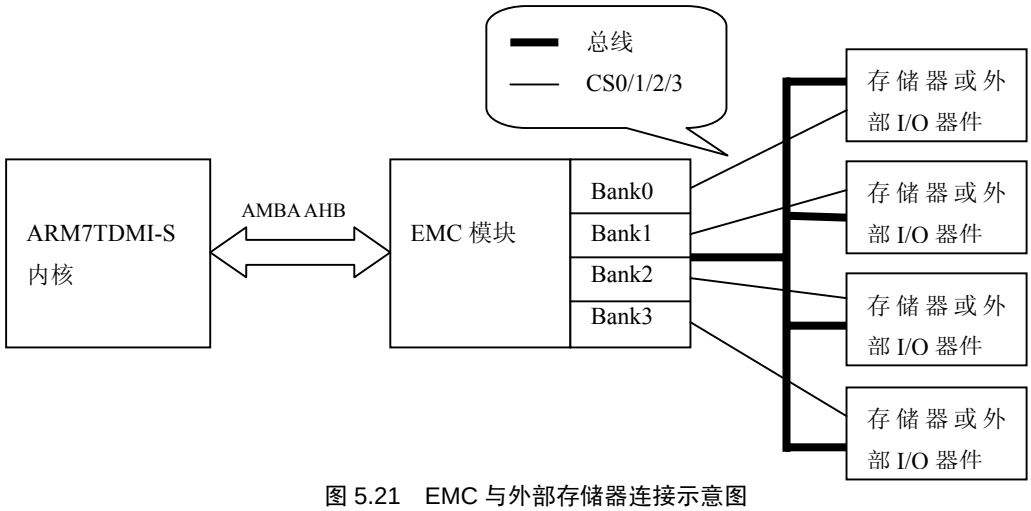


图 5.21 EMC 与外部存储器连接示意图

LPC2200 系列微控制器的引脚地址输出线是 A[23:0]，其中地址位 A[25:24]用于 4 个存储器组的译码。4 个存储器组的有效区域位于外部存储器的起始部分，地址如表 5.37 所示。**在引脚 BOOT1:0 的状态控制下，Bank 0 可用于引导程序运行。**

表 5.37 外部存储器组的地址范围

Bank	地址范围	配置寄存器
0	8000 0000－80FF FFFF	BCFG0
1	8100 0000－81FF FFFF	BCFG1
2	8200 0000－82FF FFFF	BCFG2
3	8300 0000－83FF FFFF	BCFG3

Bank0～Bank3 的片选信号分别为 CS0～CS3，如果片外存储器或 I/O 器件是通过 CS0 进行片选，或者由 CS0 与地址线进行译码来片选，则此片外存储器或 I/O 器件属于 Bank0 组，地址 0x80000000～0x80FFFFFF。

5.6.3 引脚描述

外部存储器控制器引脚描述见表 5.38。**这些引脚是与 P1、P2 和 P3 口 GPIO 功能复用，所以在使用外部总线前首先要正确配置 PINSEL2 寄存器**(可通过硬件上对引脚 BOOT1:0 设定，复位时微处理器自动初始化 PINSEL2；或者软件上直接初始化 PINSEL2，这只适用于片内 FLASH 引导程序运行的系统中)。

表 5.38 外部存储器控制器引脚描述

引脚名称	类型	引脚描述
D[31:0]	输入/输出	外部存储器数据线
A[23:0]	输出	外部存储器地址线
OE	输出	输出使能信号，低有效
BLS[3:0]	输出	字节定位选择信号，低有效
WE	输出	写使能信号，低有效
CS[3:0]	输出	芯片选择信号，低有效

5.6.4 寄存器描述

寄存器总汇

外部存储器控制器包含 4 个寄存器，如表 5.39 所示。

表 5.39 外部存储器控制器寄存器

名称	描述	访问	复位值	地址
BCFG0	存储器组 0 的配置寄存器	读/写	0x2000 FBEF	0xFFE00000
BCFG1	存储器组 1 的配置寄存器	读/写	0x2000 FBEF	0xFFE00004
BCFG2	存储器组 2 的配置寄存器	读/写	0x1000 FBEF	0xFFE00008
BCFG3	存储器组 3 的配置寄存器	读/写	0x0000 FBEF	0xFFE0000C

注：由于 Bank 0 可用于引导程序运行，所以 BCFG0 的复位值与引脚 BOOT1:0 的设定有关，见表 5.41。

每个寄存器为对应的存储器组配置了以下选项：

- 一个存储器组内部的读写访问之间以及访问一个存储器组和访问另一个存储器组之间需要间隔的空闲时钟周期个数（1～16 个 CCLK 周期），以避免器件间的总线竞争。
- 读访问长度(即等待周期+操作周期，3～34 个 CCLK 周期)，但对 Burst ROM 的连续读访问除外。
- 写访问长度(即等待周期+操作周期，1～32 个 CCLK 周期)。
- 存储器组是否写保护
- 存储器组的总线宽度：8、16 或 32 位

存储器组配置寄存器 0-3（BCFG0-3 － 0xFFE00000-0C）

对于 BCFG 寄存器，我们要根据实际连接的存储器或外设进行设置。如果使用的是 Burst ROM，则设置 BM 位为 1，否则设置为 0；对于不同宽度的存储器，设置 MW 的值；若是带有字节选择输入的 16/32 位宽度的器件，需要设置 RBLE 位为 1；然后设置总线切换的空闲周期 IDCY，读访问长度 WST1，写访问长度 WST2。

要根据存储器/外部 I/O 器件的速度来设置 WST1、WST2 的值，若存储器/外部 I/O 器件的速度较慢，还可以通过降低 CCLK 的频率确保正确的总线操作。

BCFG 寄存器描述见表 5.40。

表 5.40 存储器组配置寄存器 0-3

BCFG0-3	名称	功能	复位值
3:0	IDCY	该域控制着一个存储器组内部的读写访问之间，以及访问一个存储器组和访问另一个存储器组之间 EMC 需要给定的“空闲”CCLK 周期最小数目，以避免器件间的总线竞争。 **空闲 CCLK 周期数 = IDCY + 1**	1111
4	保留	保留，用户软件不应向其写入 1。从保留位读出的值未被定义。	NA
9:5	WST1	该域控制着读访问的长度（对 Burst ROM 的连续读访问除外）。读访问的长度以 CCLK 周期来计量。 **读访问的长度 = WST1 + 3**	11111
10	RBLE	当存储器组由字节宽度或未按字节区分的器件组成时该位为 0，这时在读访问时 EMC 将 BLS3:0 输出拉高； **当存储器组由含有字节选择输入的 16 位和 32 位宽器件组成时该位为 1**，这时在读访问时 EMC 将 BLS3:0 输出拉低。	0
15:11	WST2	该域控制着写访问的长度，写访问长度由以下几部分组成： ● 1 个 CCLK 周期（地址建立，CS、BLS 和 WE 为高） ● WST2+1 个 CCLK 周期（地址有效，CS、BLS 和 WE 为低） ● 1 个 CCLK 周期（地址有效，CS 为低，BLS 和 WE 为高） 对于 Burst ROM，该域控制着连续访问的长度，其值为 WST2+1 个 CCLK 周期。	11111
16:23	保留	保留，用户软件不应向其写入 1。从保留位读出的值未被定义。	NA
24	BUSERR	总线错误状态位。如果 EMC 检测到一个大于 32 位数据访问的 AMBA 请求时该位被置位。ARM7TDMI-S 不会出现这样的请求。	0
25	WPERR	错误写状态位。如果试图对一个 WP 位为 1 的存储器组进行写操作，该位置位。通过写入 1 将该位清零。	0

接上表

BCFG0-3	名称	功能	复位值
26	WP	该位为 1 时，表明存储器组写保护。	0
27	BM	该位为 1 时，表明存储器组使用的是 Burst ROM。	0
29:28	MW	该域控制着存储器组数据总线的宽度：00＝8 位，01＝16 位，10＝32 位，11＝保留。	见表 5.41
31:30	AT	该域通常写入 00	00

由于 Bank 0 可用于引导程序运行，所以 BCFG0[29:28]的复位值与引脚 BOOT1:0 的设定有关，见表 5.41。说明：当 BOOT1:0=11 时，从片内 FLASH 引导程序运行。

表 5.41 复位时默认的存储器宽度

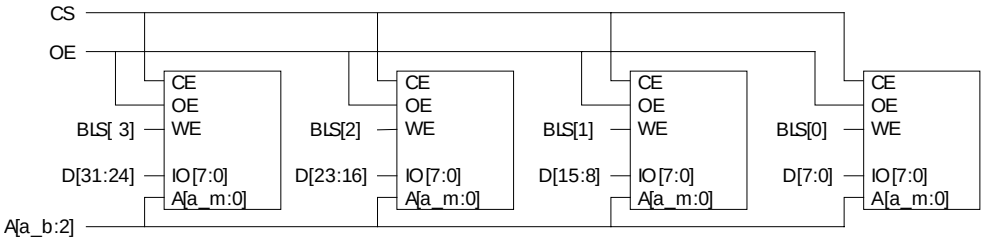
Bank	复位时 BOOT1:0 的状态	BCFG[29:28]复位值	存储器宽度
0	LL	00	8 位
0	LH	01	16 位
0	HL	10	32 位
0	HH	10	32 位
1	XX	10	32 位
2	XX	01	16 位
3	XX	00	8 位

5.6.5 外部存储器接口

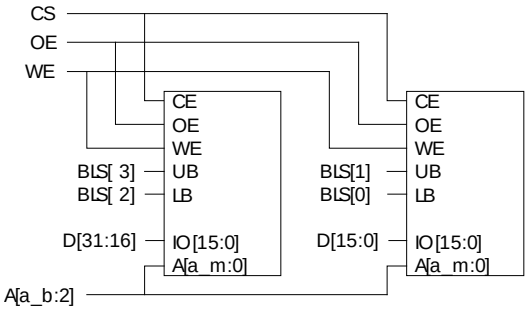
外部存储器接口取决于存储器组的宽度（32、16 或 8 位，由 BCFG 寄存器的 MW 位选择）。而且，存储器芯片的选择也需要对 BCFG 寄存器的 RBLE 位进行适当的设置。RBLE=0 时选择 8 位宽度的外部存储器；RBLE=1 时选择 16/32 位宽度的外部存储器。

如果存储器组配置成 32 位宽度，地址线 A0 和 A1 无用。如果存储器组配置成 16 位宽，则不需要 A0；8 位宽的存储器组需要使用 A0。使用各种宽度存储器与总线的连接参考图 5.22、图 5.23 和图 5.24，图中的符号“a_b”表示地址总线的最高位地址线，符号“a_m”表示存储器芯片的最高位地址线。

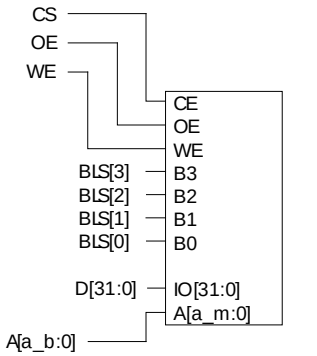
倘若所有使用的存储器组均配置为 32 位宽度，A0 和 A1 引脚可用作 GPIO。通过引脚功能选择寄存器 2（PINSEL2 寄存器）的位 23 和 24 来对 A1 和/或 A0 线来进行配置，从而实现 A0/A1 的地址或 GPIO 功能。



a) 32 位宽存储器组连接8位的存储器芯片

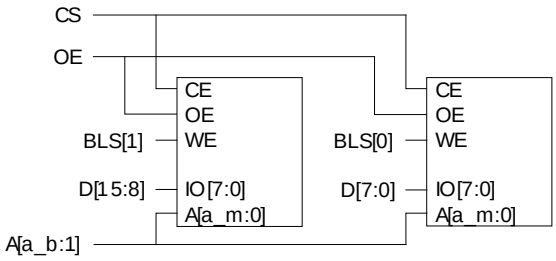


b) 32位宽存储器组连接16位的存储器芯片

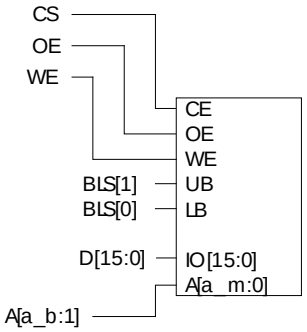


c) 32 位宽存储器组连接32位的存储器芯片

图 5.22 32 位存储器组的外部存储器接口



a) 16 位宽存储器组连接8位的存储器芯片



a) 16位宽存储器组连接16位的存储器芯片

图 5.23 16 位存储器组的外部存储器接口

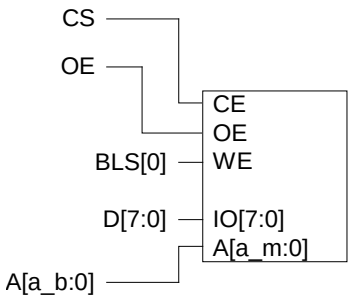


图 5.24 8 位存储器组的外部存储器接口

5.6.6 典型总线时序

图 5.25、图 5.26 所示为典型的外部存储器读/写访问时序。XCLK 是 P3.23 上的时钟信号。当 P3.23 脚的信号不被外部存储器用作时钟信号时，在典型的外部存储器读/写访问中，它还可用作时间基准（**XCLK 和 CCLK 必须设定成相同的频率**）。

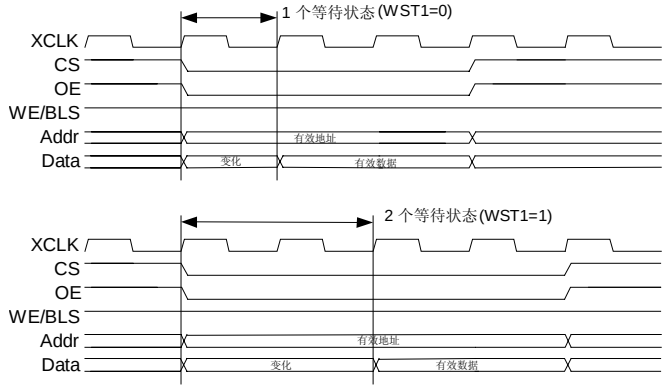


图 5.25 外部存储器读访问（WST1=0 和 WST1=1 两种情况）

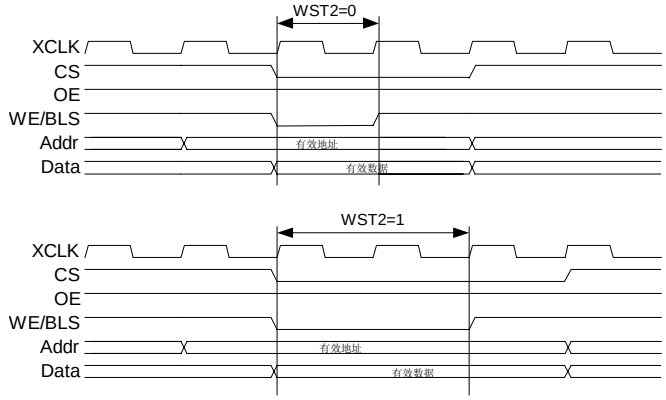


图 5.26 外部存储器写访问（WST2=0 和 WST2=1 两种情况）

图 5.25 和图 5.26 所示是典型的外部存储器读/写访问时序。因而，在某些特殊情况下会有所变化。例如，当对刚被选中的存储器组执行首次读访问时，CS 和 OE 线的低电平可能比图 5.25 中早出现 1 个 XCLK 周期。

同样，在对 SRAM 的几次连续写访问时序中，最后一次写访问的时序与图 5.26 给出的相同。但另一方面，前导写周期的数据有效时间会长 1 个周期。单个的写访问时序也将与图 5.26 的其中一个相同。

5.6.7 外部存储器选择

根据 EMC 操作的描述和常用的外部存储器（合适的读和写访问时间 t_{RAM} 和 t_{WRITE}），可构造出

表 5.42 并用于外部存储器的选择。t_{CYC} 表示单个 XCLK 周期（见图 5.25 和图 5.26）。F_{max} 表示选用了外部存储器的系统能得到的最大 CCLK 频率。

表 5.42 外部存储器和系统的性能指标

访问时序	最大频率	WST 设置 （WST>=0；取整数）	所需的存储器访问时间
标准读	$F_{max}<=\frac{2+WST1}{t_{RAM}+20ns}$	$WST1>=\frac{t_{RAM}+20ns}{t_{CYC}}-2$	$t_{RAM}<=t_{CYC}*(2+WST1)-20ns$
标准写	$F_{max}<=\frac{1+WST2}{t_{RAM}+5ns}$	$WST2>=\frac{t_{WRITE}-t_{CYC}+5ns}{t_{CYC}}$	$t_{WRITE}<=t_{CYC}*(1+WST2)-5ns$

5.6.8 启动代码相关部分

由于 LPC2210/2212/2214 是总线开放型芯片，具有 4 个 Bank 的存储器组，总线宽度可设置为 8 位、16 位或 32 位。LPC2200 的启动代码中包总线的初始化设置，如程序清单 5.11 所示(在 startup.s 文件中)。

程序清单 5.11 总线配置初始化

```
ResetInit
        LDR     R0, =PINSEL2                                    (1)
    IF  :DEF: EN_CRP
        LDR     R1, =0x0f814910                                 (2)
    ELSE
        LDR     R1, =0x0f814914                                 (3)
    ENDIF
        STR     R1, [R0]                                        (4)
        LDR     R0, =BCFG0                                      (5)
        LDR     R1, =0x1000ffef                                 (6)
        STR     R1, [R0]                                        (7)
        LDR     R0, =BCFG1                                      (8)
        LDR     R1, =0x1000ffef                                 (9)
        STR     R1, [R0]                                        (10)
;       LDR     R0, =BCFG2                                      (11)
;       LDR     R1, =0x2000ffef                                 (12)
;       STR     R1, [R0]                                        (13)
;       LDR     R0, =BCFG3                                      (14)
;       LDR     R1, =0x2000ffef                                 (15)
;       STR     R1, [R0]                                        (16)
        …
```

由程序清单 5.1 可知，在芯片复位时会程序跳转到标号 ResetInit 处。首先，程序清单 5.11 (1)～(4)设置 PINSEL2 寄存器的值，即设置总线的 IO 引脚，详细介绍参考 5.9 节；

然后分别配置外部第 0 个存储区（程序清单 5.11 (5)~(7)）、第 1 个存储区（程序清单 5.11 (8)~(10)）、第 2 个存储区（程序清单 5.11 (11)~(13)）和第 3 个存储区（程序清单 5.11 (14)~(16)）的时序和总线宽度。程序默认设置 Bank0、Bank1 为 16 位总线宽度，总线速度调到最慢，

用户可根据实际的目标系统更改设置值。

5.7 引脚连接模块

5.7.1 介绍

引脚连接模块使同一个管脚可以具有多种功能，即管脚复用，这是通过配置相关寄存器控制多路开关来连接引脚与片内外设。

外设在激活和任何相关中断使能之前必须连接到适当的引脚。任何使能的外设功能如果没有映射到相关的引脚，则被认为是无效的。

5.7.2 寄存器描述

寄存器汇总

引脚连接模块包含 3 个寄存器，见表 5.43。

表 5.43 引脚连接模块寄存器映射

名称	描述	访问	复位值	地址
PINSEL0	引脚选择寄存器 0	读/写	0x0000 0000	0xE002C000
PINSEL1	引脚选择寄存器 1	读/写	0x1540 0000	0xE002C004
PINSEL2	引脚选择寄存器 2	读/写	见表 5.47 和表 5.48	0xE002C014

引脚功能选择寄存器 0（PINSEL0 - 0xE002C000）

PINSEL0 寄存器按照表 5.44 当中的设定来控制引脚的功能。IODIR 寄存器中的方向控制位只有在引脚选择 GPIO 功能时才有效。对于其它功能，方向是自动控制的。

表 5.44 引脚选择寄存器 0

PINSEL0	引脚名称	00	01	10	11	复位值
1:0	P0.0	GPIO P0.0	TxD(UART0)	PWM1	保留	00
3:2	P0.1	GPIO P0.1	RxD(UART0)	PWM3	EINT0	00
5:4	P0.2	GPIO P0.2	SCL(I^2C)	捕获 0.0(TIMER0)	保留	00
7:6	P0.3	GPIO P0.3	SDA(I^2C)	匹配 0.0(TIMER0)	EINT1	00
9:8	P0.4	GPIO P0.4	SCK(SPI0)	捕获 0.1(TIMER0)	保留	00
11:10	P0.5	GPIO P0.5	MISO(SPI0)	匹配 0.1(TIMER0)	保留	00
13:12	P0.6	GPIO P0.6	MOSI(SPI0)	捕获 0.2(TIMER0)	保留	00
15:14	P0.7	GPIO P0.7	SSEL(SPI0)	PWM2	EINT2	00
17:16	P0.8	GPIO P0.8	TxD UART1	PWM4	保留	00
19:18	P0.9	GPIO P0.9	RxD(UART1)	PWM6	EINT3	00
21:20	P0.10	GPIO P0.10	RTS(UART1)	捕获 1.0(TIMER1)	保留	00
23:22	P0.11	GPIO P0.11	CTS(UART1)	捕获 1.1(TIMER1)	保留	00
25:24	P0.12	GPIO P0.12	DSR(UART1)	匹配 1.0(TIMER1)	保留	00
27:26	P0.13	GPIO P0.13	DTR(UART1)	匹配 1.1(TIMER1)	保留	00
29:28	P0.14	GPIO P0.14	CD(UART1)	EINT1	保留	00
31:30	P0.15	GPIO P0.15	RI(UART1)	EINT2	保留	00

说明：表 5.44 中的“PINSEL0”栏表示 PINSEL0 寄存器的控制位，“引脚名称”栏表示控制位所控制的引脚，“00/01/10//11”栏表示控制位在这些设定值时的引脚功能。比如，P0.0 脚，控制位为 PINSEL0[1:0]，

当 PINSEL0[1:0]=00 时引脚为 GPIO 功能(即 P0.0)，当 PINSEL0[1:0]=01 时引脚为 UART0 的 TxD 功能脚，当 PINSEL0[1:0]=10 时引脚为 PWM1 功能脚。

引脚功能选择寄存器 1（PINSEL1 - 0xE002C004）

PINSEL1 寄存器按照表 5.45 中的设定来控制引脚的功能。IODIR 寄存器中的方向控制位只有在引脚选择 GPIO 功能时才有效。对于其它功能，方向是自动控制的。

表 5.45 引脚选择寄存器 1

PINSEL1	引脚名称	00	01	10	11	复位值
1:0	P0.16	GPIO P0.16	EINT0	匹配 0.2(TIMER0)	保留	00
3:2	P0.17	GPIO P0.17	捕获 1.2(TIMER1)	SCK(SPI1)	匹配 1.2(TIMER1)	00
5:4	P0.18	GPIO P0.18	捕获 1.3(TIMER1)	MISO(SPI1)	匹配 1.3(TIMER1)	00
7:6	P0.19	GPIO P0.19	匹配 1.2(TIMER1)	MOSI(SPI1)	匹配 1.3(TIMER1)	00
9:8	P0.20	GPIO P0.20	匹配 1,3(TIMER1)	SSEL(SPI1)	EINT3	00
11:10	P0.21	GPIO P0.21	PWM5	保留	捕获 1.3(TIMER1)	00
13:12	P0.22	GPIO P0.22	保留	捕获 0.0(TIMER0)	匹配 0.0(TIMER0)	00
15:14	P0.23	GPIO P0.23	保留	保留	保留	00
17:16	P0.24	GPIO P0.24	保留	保留	保留	00
19:18	P0.25	GPIO P0.25	保留	保留	保留	00
21:20	P0.26	保留				00
23:22	P0.27	GPIO P0.27	AIN0(A/D 转换器)	捕获 0.1(TIMER0)	匹配 0.1(TIMER0)	01
25:24	P0.28	GPIO P0.28	AIN1(A/D 转换器)	捕获 0.2(TIMER0)	匹配 0.2(TIMER0)	01
27:26	P0.29	GPIO P0.29	AIN2(A/D 转换器)	捕获 0.3(TIMER0)	匹配 0.3(TIMER0)	01
29:28	P0.30	GPIO P0.30	AIN3(A/D 转换器)	EINT3	捕获 0.0(TIMER0)	01
31:30	P0.31	保留				00

PINSEL 寄存器控制表 5.46 中器件引脚的功能。每 2 个寄存器位对应 1 个特定的器件引脚。PINSEL1 中的[23:22]、[25:24]、[27:26]、[29:28]位复位值为 01。

表 5.46 引脚功能选择寄存器位

PINSEL0 和 PINSLE1 的值		功能	复位值
0	0	首选（默认）功能，通常为 GPIO 口	00
0	1	第一可选功能	
1	0	第二可选功能	
1	1	保留	

引脚功能选择寄存器 2（PINSEL2 - 0xE002C014）

PINSEL2 寄存器按照表 5.47、表 5.48 当中的设定来控制引脚的功能。IODIR 寄存器中

的方向控制位只有在引脚选择 GPIO 功能时才有效。对于其它功能，方向是自动控制的。

在表 5.47、表 5.48 中的“复位值”这一栏表示微控制器复位时对应位的值；对于 PINSEL2 的 bit2、bit3，复位时由 P1.26、P1.20 引脚的电平决定，倘若引脚接有上拉电阻(如 10KΩ上拉电阻)，相应位的值被设置为 0，倘若引脚接有下拉电阻(如 4.7KΩ下拉电阻)，相应位的值被设置为 1；对于 PINSEL2 的 bit23、bit24、bit25～bit27，复位时 BOOT1 和 BOOT0 引脚的电平决定。

警告：使用读－修改－写的方法来访问 PINSEL2 寄存器，例如 PINSEL2 = (PINSEL2&0xFFFFFFCF) | (2<<4)。对 bit0～bit2 和/或 bit3 的意外写操作会造成调试和/或跟踪功能的丢失！

表 5.47 LPC2114/2124 引脚功能选择寄存器 2

PINSEL2	描述	复位值
1:0	保留。	00
2	该位为 0 时，P1.31:26 用作 GPIO。该位为 1 时，P1.31:26 用作一个调试端口。	$\overline{\text{P1.26/RTCK}}$
3	该位为 0 时，P1:25:16 用作 GPIO。该位为 1 时，P1.25:16 用作一个跟踪端口。	$\overline{\text{P1.20}}$ / $\overline{\text{TRACESYNC}}$
4:31	保留。	00

表 5.48 LPC2210/2212/2214 引脚功能选择寄存器 2

PINSEL2	描述	复位值
1:0	保留。	00
2	该位为 0 时，P1.36:26 用作 GPIO。该位为 1 时，P1.31:26 用作一个调试端口。	$\overline{\text{P1.26/RTCK}}$
3	该位为 0 时，P1:25:16 用作 GPIO。该位为 1 时，P1.25:16 用作一个跟踪端口。	$\overline{\text{P1.20}}$ / $\overline{\text{TRACESYNC}}$
5:4	控制数据总线和选通引脚的使用： 引脚 P2.7:0 11=P2.7:0 0x 或 10＝D7:0 引脚 P1.0 11=P1.0 0x 或 10＝CS0 引脚 P1.1 11=P1.1 0x 或 10＝OE 引脚 P3.31 11=P3.31 0x 或 10＝BLS0 引脚 P2.15:8 00 或 11＝P2.15:8 01 或 10＝D15:8 引脚 P3.30 00 或 11＝P3.30 01 或 10＝BLS1 引脚 P2.27:16 0x 或 11＝P2.27:16 10=D27:16 引脚 P2.29:28 0x 或 11＝P2.29:28 10=D29:28 引脚 P2.31:30 0x 或 11＝P2.31:30 或 AIN5:4 10=D31:30 引脚 P3.29:28 0x 或 11＝P3.29:28 或 AIN6:7 10=BLS2:3	BOOT1:0 (如 BOOT1:0=01，该域的复位值就为 01)
6	如果位 5:4 不为 10，由该位控制 P3.29 脚的使用：为 0 时使能 P3.29，为 1 时使能 AIN6。	1
7	如果位 5:4 不为 10，由该位控制 P3.28 脚的使用：为 0 时使能 P3.28，为 1 时使能 AIN7。	1
8	该位控制 P3.27 脚的使用：为 0 时使能 P3.27，为 1 时使能 WE。	0
10:9	保留。	-
11	该位控制 P3.26 脚的使用：为 0 时使能 P3.26，为 1 时使能 CS1。	0
12	保留。	-

接上表

PINSEL2	描述	复位值
13	如果位 25:23 不为 111，由该位控制 P3.23/A23/XCLK 脚的使用：为 0 时使能 P3.23，为 1 时使能 XCLK。	0
15:14	控制 P3.25 脚的使用：00 使能 P3.25，01 使能 CS2，10 和 11 保留。	00
17:16	控制 P3.24 脚的使用：00 使能 P3.24，01 使能 CS3，10 和 11 保留。	00
19:18	保留。	-
20	如果位 5:4 不为 10，由该位控制 P2.29:28 的使用：0 使能 P2.29:28，1 保留。	0
21	如果位 5:4 不为 10，由该位控制 P2.30 的使用：0 使能 P2.30，1 使能 AIN4。	1
22	如果位 5:4 不为 10，由该位控制 P2.31 的使用：0 使能 P2.31，1 使能 AIN5。	1
23	控制 P3.0/A0 用作端口引脚（0）或地址线（1）。	如果 $\overline{\text{RESET}}$ ＝0 时 BOOT1:0=00，该位的复位值为 1。反之为 0。
24	控制 P3.1/A1 用作端口引脚（0）或地址线（1）。	如果复位时 BOOT1=0，该位的复位值为 1，反之为 0。
27:25	控制 P3.23/A23/XCLK 和 P3.22:2/A2.22:2 中地址线的数目： 000＝无地址线　　100＝A11:2 为地址线 001＝A3:2 为地址线　　101＝A15:2 为地址线 010＝A5:2 为地址线　　110＝A19:2 为地址线 011＝A7:2 为地址线　　111＝A23:2 为地址线	如果复位时 BOOT1:0=11，该域的复位值为 000。反之为 111。
31:28	保留。	-

5.7.3 引脚功能控制

1. 将 P0.8、P0.9 设置为 TxD1、RxD1 功能

```
    PINSEL0 = 0x00050000;
或  PINSEL0 = 0x05<<16;
```

PINSEL0、PINSEL1 和 PINSEL2 寄存器是可读可写的，为了不更改原先的引脚功能设置，可以先读取寄存器值，然后进行逻辑“与”、“或”操作，再回写到此寄存器。

```
    PINSEL0 = ( PINSEL0 & 0xFFF0FFFF ) | (0x05<<16);
```

2．PINSEL2 与芯片加密

LPC2114/2124/2212/2214 片内 FLASH 是可以加密的，进行加密设置后，JTAG 调试接口无效，ISP 功能只提供读 ID 及全片擦除功能。

但是要注意，PINSEL2 的 bit2 是 JTAG 接口使能的控制位，若用户程序将此位设置 1 时，则会强行使能 JTAG 接口。LPC2100、LPC2200 的启动代码支持芯片加密，已对 PINSEL2 正确设置，一般用户程序不需要再对 PINSEL2 操作。

5.7.4 启动代码相关部分

由于 LPC2210/2212/2214 是总线开放型芯片，其总线宽度可设置为 8 位、16 位或 32 位，

对于没有使用到的总线引脚(比如 16 位总线宽度时，D16～D31 没有使用)，可以作为 GPIO 使用。LPC2200 的启动代码中包 P1、P2 和 P3 口的初始化设置，如程序清单 5.12 所示(在 startup.s 文件中)。

程序清单 5.12 总线引脚设置

```
ResetInit
        LDR     R0, =PINSEL2                                    (1)
    IF  :DEF: EN_CRP
        LDR     R1, =0x0f814910                                 (2)
    ELSE
        LDR     R1, =0x0f814914                                 (3)
    ENDIF
        STR     R1, [R0]                                        (4)
         …
```

程序清单 5.12 中，当预定义有 EN_CRP 宏时，将会编译程序清单 5.12 (2)，PINSEL2 被设置为 0x0f814910，禁止 JTAG 口调试；若没有定义 EN_CRP 宏，将会编译程序清单 5.12 (3)，PINSEL2 被设置为 0x0f814914，使用 JTAG 口调试。

在 startup.s 文件中，用户并不能找到定义 EN_CRP 宏的代码，也不需要直接去定义 EN_CRP 宏，因为使用 LPC2200 的工程模板时，只要选 RelInChip 目标，编译器将会预定义 EN_CRP 宏，而选其它目标则不会预定义 EN_CRP 宏。

没有定义 EN_CRP 宏时，PINSEL2 被设置为 0x0f814914，含义如下：

1:0 位为 00，保留；

2 位为 1，P1.31～P1.26 作为 JTAG 调试端口；

3 位为 0，P1.25～P1.16 作为 GPIO；

5:4 位为 01，总线引脚使用 D0～D15、CS0、OE、BLS0 和 BLS1；

6 位为 0，P3.29 作为 GPIO；

7 位为 0，P3.28 作为 GPIO；

8 位为 1，P3.27 作为 WE；

10:9 位为 00，保留；

11 位为 1，P3.26 作为 CS1；

12 位为 0，保留；

13 位为 0，P3.23 作为 GPIO(当 27:25 不为 111 时)；

15:14 位为 01，P3.25 作为 CS2；

17:16 位为 01，P3.24 作为 CS3；

19:18 位为 00，保留；

20 位为 0，P2.29、P2.28 作为 GPIO；

21 位为 0，P2.30 作为 GPIO；

22 位为 0，P2.31 作为 GPIO；

23 位为 1，P3.0 作为 A0；

24 位为 1，P3.1 作为 A1；

27:25 位为 111，P3.23～P3.2 作为 A23～A2；

31:28 位为 0000，保留。

5.8 向量中断控制器（VIC）

5.8.1 特性

- ARM PrimeCell ™ 向量中断控制器
- 最多 32 个中断请求输入
- 16 个向量 IRQ 中断
- 16 个优先级，可动态分配给中断请求
- 可产生软件中断

5.8.2 描述

向量中断控制器（Vectored Interrupt Controller，简写为VIC）具有32个中断请求输入（注意：这是本模块具有这么多中断请求输入，而不是芯片具有这么多中断请求连接到本模块），可将其编程分为3类：FIQ、向量IRQ和非向量IRQ。可编程分配机制意味着不同外设的中断优先级可以动态分配及调整。

- **快速中断请求**（FIQ）要求具有最高优先级。如果分配给 FIQ 的请求多于 1 个，VIC 将中断请求“相或”后向 ARM 处理器产生 FIQ 信号。当只有一个中断被分配为 FIQ 时可实现最短的 FIQ 等待时间，因为 FIQ 服务程序只要简单地启动对该中断处理就可以了。但如果分配给 FIQ 的中断多于 1 个，FIQ 服务程序从 VIC 中读出 FIQ 状态寄存器来识别产生中断请求的 FIQ 中断源是哪一个。
- **向量 IRQ 中断**具有中等优先级。该级别可分配 32 个请求中的 16 个。32 个请求中的任意一个都可分配到 16 个向量 IRQ slot 中的任意一个，其中 slot0 具有最高优先级，而 slot15 则为最低优先级。
- **非向量 IRQ 中断**的优先级最低。如果分配给非向量 IRQ 的中断多于 1 个，默认中断服务程序要从 VIC 中读出 IRQ 状态寄存器来识别产生中断请求的 IRQ 中断源是哪一个。

VIC 将所有向量和非向量 IRQ 相“或”向 ARM 处理器产生 IRQ 信号。如果有任意一个向量 IRQ 发出请求，VIC 则提供最高优先级请求 IRQ 服务程序的地址；若是非向量 IRQ 中断，则提供所默认服务程序的地址。IRQ 中断入口程序可通过读取 VIC 的向量地址存器(VICVectAddr)来取得该地址，然后跳转到相应地址即可执行相应中断的服务程序。该默认服务程序由所有非向量 IRQ 共用，默认服务程序可读取 IRQ 状态寄存器以确定哪个 IRQ 被激活。

VIC 的 IRQ 中断优先级只是在同时产生多个中断时，VIC 会将最高优先级请求的 IRQ 服务程序地址存入向量地址寄存器 VICVectAddr，没有限制低优先级中断产生的中断逻辑机制。关于向量中断控制器的其它信息请参阅 *ARM PrimeCell ™ 向量中断控制器（PL190）*的相关文档。

使用 VIC 的 IRQ 中断处理过程如图 5.27 所示，用户程序首先要初始化 VIC 使能相关中断，然后正常运行用户程序(如图 5.27 中的①)；当有 IRQ 中断产生时，VIC 将会根据中断源设置 VICVectAddr 寄存器为相应中断服务程序的地址(如图 5.27 中的②)，切换处理器工作模式为 IRQ 模式，并跳转到 IRQ 中断入口 0x00000018 处(如图 5.27 中的③)；异常向量表中 0x00000018 处使用一条“LDR PC, [PC, #-0xff0]”指令，这条指令将会读取 VICVectAddr 寄存器的值然后放入 PC 程序指针，即跳转到相应中断服务程序(如图 5.27 中的④)；中断服务中执行相应的中断处理，清除中断标志，如图 5.27 中的⑤；中断服务完成后，即可返回原中断点(如图 5.27 中的⑥)，注意返回时要同时切换处理器工作模式。

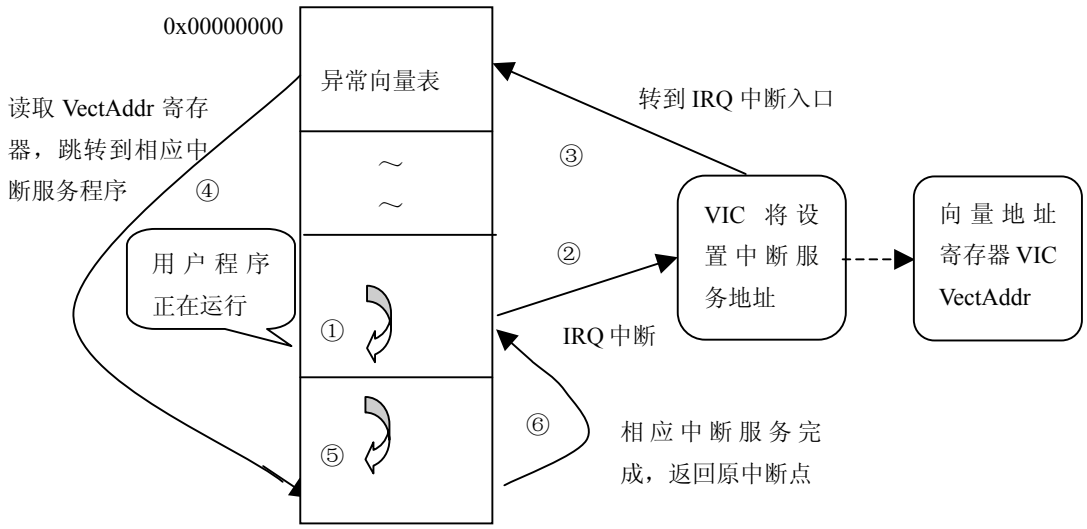


图 5.27 使用 VIC 的 IRQ 中断处理过程

5.8.3 结构

向量中断控制器方框图见图 5.28。

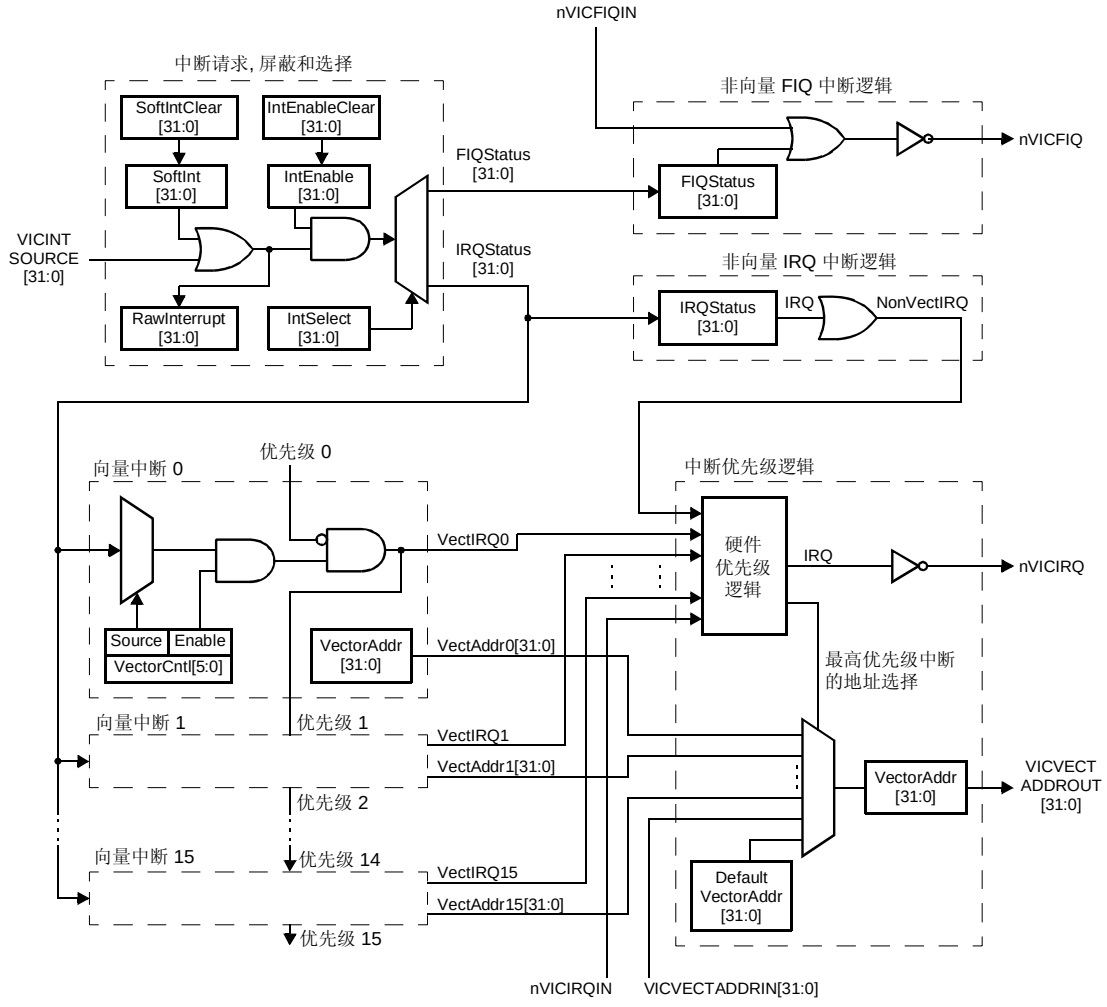


图 5.28 向量中断控制器方框图

5.8.4 寄存器描述

寄存器汇总

VIC 所包含的寄存器如表 5.49 所示。VIC 中所有的寄存器都为字寄存器。不支持字节和半字的读和写操作。

表 5.49 VIC 寄存器映射

名称	描述	访问	复位值*	地址
VICIRQStatus	IRQ 状态寄存器。该寄存器读出定义为 IRQ 并使能的中断的状态。	RO	0	0xFFFF F000
VICFIQStatus	FIQ 状态寄存器。该寄存器读出定义为 FIQ 并使能的中断的状态。	RO	0	0xFFFF F004
VICRawIntr	所有中断的状态寄存器。该寄存器读出 32 个中断请求/软件中断的状态，不管中断是否使能或分类。	RO	0	0xFFFF F008
VICIntSelect	中断选择寄存器。该寄存器将 32 个中断请求的每个都分配为 FIQ 或 IRQ。	R/W	0	0xFFFF F00C
VICIntEnable	中断使能寄存器。该寄存器控制 32 个中断请求和软件中断的使能。	R/W	0	0xFFFF F010
VICIntEnClr	中断使能清零寄存器。该寄存器允许软件将中断使能寄存器中的一个或多个位清零。	W	0	0xFFFF F014
VICSoftInt	软件中断寄存器。该寄存器的内容与 32 个不同外设的中断请求“相或”来产生中断。	R/W	0	0xFFFF F018
VICSoftIntClear	软件中断清零寄存器。该寄存器允许软件将软件中断寄存器中的一个或多个位清零。	W	0	0xFFFF F01C
VICProtection	保护使能寄存器。该寄存器可以限制非特权模式下的软件对 VIC 寄存器的访问。	R/W	0	0xFFFF F020
VICVectAddr	向量地址寄存器。当发生一个 IRQ 中断时，IRQ 服务程序可读出该寄存器并跳转到读出的地址。	R/W	0	0xFFFF F030
VICDefVectAddr	默认向量地址寄存器。该寄存器保存了非向量IRQ的中断服务程序（ISR）地址。	R/W	0	0xFFFF F034
VICVectAddr0	向量地址 0 寄存器。向量地址寄存器 0-15 保存了 16 个向量 IRQ slot 的中断服务程序地址。	R/W	0	0xFFFF F100
VICVectAddr1	向量地址 1 寄存器	R/W	0	0xFFFF F104
VICVectAddr2	向量地址 2 寄存器	R/W	0	0xFFFF F108
VICVectAddr3	向量地址 3 寄存器	R/W	0	0xFFFF F10C
VICVectAddr4	向量地址 4 寄存器	R/W	0	0xFFFF F110
VICVectAddr5	向量地址 5 寄存器	R/W	0	0xFFFF F114
VICVectAddr6	向量地址 6 寄存器	R/W	0	0xFFFF F118
VICVectAddr7	向量地址 7 寄存器	R/W	0	0xFFFF F11C
VICVectAddr8	向量地址 8 寄存器	R/W	0	0xFFFF F120
VICVectAddr9	向量地址 9 寄存器	R/W	0	0xFFFF F124
VICVectAddr10	向量地址 10 寄存器	R/W	0	0xFFFF F128
VICVectAddr11	向量地址 11 寄存器	R/W	0	0xFFFF F12C
VICVectAddr12	向量地址 12 寄存器	R/W	0	0xFFFF F130

接上表

名称	描述	访问	复位值*	地址
VICVectAddr13	向量地址 13 寄存器	R/W	0	0xFFFF F134
VICVectAddr14	向量地址 14 寄存器	R/W	0	0xFFFF F138
VICVectAddr15	向量地址 15 寄存器	R/W	0	0xFFFF F13C
VICVectCntl0	向量控制 0 寄存器。向量控制寄存器 0-15 分别控制 16 个向量 IRQ slot 中的一个。Slot0 优先级最高，而 Slot15 优先级最低。	R/W	0	0xFFFF F200
VICVectCntl1	向量控制 1 寄存器	R/W	0	0xFFFF F204
VICVectCntl2	向量控制 2 寄存器	R/W	0	0xFFFF F208
VICVectCntl3	向量控制 3 寄存器	R/W	0	0xFFFF F20C
VICVectCntl4	向量控制 4 寄存器	R/W	0	0xFFFF F210
VICVectCntl5	向量控制 5 寄存器	R/W	0	0xFFFF F214
VICVectCntl6	向量控制 6 寄存器	R/W	0	0xFFFF F218
VICVectCntl7	向量控制 7 寄存器	R/W	0	0xFFFF F21C
VICVectCntl8	向量控制 8 寄存器	R/W	0	0xFFFF F220
VICVectCntl9	向量控制 9 寄存器	R/W	0	0xFFFF F224
VICVectCntl10	向量控制 10 寄存器	R/W	0	0xFFFF F228
VICVectCntl11	向量控制 11 寄存器	R/W	0	0xFFFF F22C
VICVectCntl12	向量控制 12 寄存器	R/W	0	0xFFFF F230
VICVectCntl13	向量控制 13 寄存器	R/W	0	0xFFFF F234
VICVectCntl14	向量控制 14 寄存器	R/W	0	0xFFFF F238
VICVectCntl15	向量控制 15 寄存器	R/W	0	0xFFFF F23C

*：复位值仅指已使用位中保存的数据，不包括保留位的内容。

以下将按照 VIC 逻辑中的使用顺序对 VIC 寄存器进行描述，该顺序为从那些与中断请求输入最密切的寄存器到那些由软件所使用的最抽象的寄存器。对大多数人来说，这也是在学习 VIC 时读取寄存器的最佳顺序。

软件中断寄存器（VICSoftInt - 0xFFFFF018，读/写）

VIC 在执行任何逻辑之前，将该寄存器的内容与 32 个不同外设的中断请求相“或”来产生中断。

VICSoftInt 寄存器描述见表 5.50。

表 5.50 软件中断寄存器

VICSoftInt	功能	复位值
31:0	1：强制产生与该位相关的中断请求。 0：不强制产生中断请求。 向 VICSoftInt 写入 0 无效，通过写 VICSoftIntClear 清零相应位。	0

软件中断清零寄存器（VICSoftIntClear - 0xFFFFF01C，只写）

可用软件清零软件中断寄存器(VICSoftInt)中的一个或多个位，即清除相应中断输入的 VIC 软件中断。

VICSoftIntClear 寄存器描述见表 5.51。

表 5.51 软件中断清零寄存器

VICSoftIntClear	功能	复位值
31:0	1：写入 1 清零软件中断寄存器的相应位，并解除强制的中断请求。 0：写入 0 不会影响 VICSoftInt 中的相应位。	0

所有中断状态寄存器（VICRawIntr - 0xFFFFF008，只读）

该寄存器读取所有 32 个中断请求和软件中断的状态，不管中断是否使能或分类(IRQ 或 FIQ)。

VICRawIntr 寄存器描述见表 5.52。

表 5.52 所有中断状态寄存器

VICRawIntr	功能	复位值
31:0	1：对应位的中断请求或软件中断声明(即有中断)。 0：对应位的中断请求或软件中断未声明。	0

中断使能寄存器（VICIntEnable - 0xFFFFF010，读/写）

该寄存器使能分配为 FIQ 或 IRQ 的中断请求或软件中断。

VICIntEnable 寄存器描述见表 5.53。

表 5.53 中断使能寄存器

VICIntEnable	功能	复位值
31:0	当写该寄存器时，1 使能中断请求或软件中断，写入 0 无效，通过写 VICIntEnClr 清零相应位(禁止中断)。 当读取该寄存器时，1 表示中断请求使能为 FIQ 或 IRQ。	0

中断使能清零寄存器（VICIntEnClear - 0xFFFFF014，只写）

可用软件清零中断使能寄存器(VICIntEnable)中的一个或多个位，即禁止相应中断输入的使能。

VICIntEnClr 寄存器描述见表 5.54。

表 5.54 中断使能清零寄存器

VICIntEnClr	功能	复位值
31:0	1：写入 1 清零中断使能寄存器中的对应位并禁止对应的中断请求。 0：写入 0 不影响中断使能寄存器中的位。	0

中断选择寄存器（VICIntSelect - 0xFFFFF00C，读/写）

该寄存器将 32 个中断请求分别分配为 FIQ 或 IRQ。

VICIntSelect 寄存器描述见表 5.55。

表 5.55 中断选择寄存器

VICIntSelect	功能	复位值
31:0	1：对应的中断请求分配为 FIQ。 0：对应的中断请求分配为 IRQ。	0

IRQ 状态寄存器（VICIRQStatus - 0xFFFFF000，只读）

该寄存器保存了已使能的 IRQ 中断请求的状态，不管是向量和非向量 IRQ。

VICIRQStatus 寄存器描述见表 5.56。

表 5.56 IRQ 状态寄存器

VICIRQStatus	功能	复位值
31:0	1：对应位的中断请求使能并分配为 IRQ 并且声明。	0

FIQ 状态寄存器（VICFIQStatus - 0xFFFFF004，只读）

该寄存器保存了已使能的 FIQ 中断请求的状态。如果有超过一个请求分配为 FIQ，FIQ 服务程序可读取该寄存器来确定是哪一个（几个）请求被激活。

VICFIQStatus 寄存器描述见表 5.57。

表 5.57 FIQ 状态寄存器

VICFIQStatus	功能	复位值
31:0	1：对应位的中断请求使能并分配为 FIQ 并且声明。	0

向量控制寄存器 0-15（VICVectCnt l0-15 - 0xFFFFF200-23C，读/写）

每一个寄存器控制 16 个向量 IRQ slot 中的一个。Slot0 优先级最高，Slot15 优先级最低。在 VICVectCntl 寄存器中禁止一个向量 IRQ slot 不会禁止中断本身，只是中断变为了非向量的形式。**VICVectCntl [4:0]是为此 IRQ slot 所分配中断源的编号，中断源的编号见表 5.63。**

VICVectCntl0-15 寄存器描述见表 5.58。

表 5.58 向量控制寄存器 0-15

VICVectCntl0-15	功能	复位值
5	1：向量 IRQ 使能，当分配的中断请求或软件中断使能，被分配为 IRQ 并声明时，可产生一个唯一的 ISR 地址(读 VICVectAddr 寄存器)。	0
4:0	分配给此向量 IRQ slot 的中断请求或软件中断的编号。 说明，不要将把相同的中断编号分配给多个使能的向量 IRQ slot。但如果这样做了，当中断请求或软件中断使能，且被分配为 IRQ 并声明时，会使用最低编号的 slot。	0

向量地址寄存器 0-15（VICVectAddr0-15 - 0xFFFFF100-13C，读/写）

这些寄存器保存 16 个向量 IRQ slot 中断服务程序（ISR）的地址。

VICVectAddr0-15 寄存器描述见表 5.59。

表 5.59 向量地址寄存器 0-15

VICVectAddr0-15	功能	复位值
31:0	当一个或多个分配为向量 IRQ slot 的中断请求使能，分配为 IRQ 并声明时，IRQ 服务程序读取向量地址寄存器（VICVectAddr）时会得到最高优先级 slot 寄存器的值。	0

默认向量地址寄存器（VICDefVectAddr - 0xFFFFF034，读/写）

该寄存器保存了非向量 IRQ 中断服务程序（ISR）的地址。

VICDefVectAddr 寄存器描述见表 5.60。

表 5.60 默认向量地址寄存器

VICDefVectAddr	功能	复位值
31:0	当一个 IRQ 服务程序读取向量地址寄存器（VICVectAddr），并且没有 IRQ slot 响应时，则返回该寄存器中的地址。	0

向量地址寄存器（VICVectAddr - 0xFFFFF030，读/写）

当发生一个 IRQ 中断时，VIC 会将对应的 IRQ 服务程序地址存入该寄存器，IRQ 中断入口处的程序可读取该寄存器并跳转到读出的地址，执行相应的中断服务程序。

注意，该寄存器应该在 ISR 快结束时执行一次写操作(写入的值一般为 0)，以便更新优先级硬件。

VICVectAddr 寄存器描述见表 5.61。

表 5.61 向量地址寄存器

VICVectAddr	功能	复位值
31:0	当任何分配给向量 IRQ slot 的中断请求或软件中断使能，分配为 IRQ 并声明时，读取该寄存器将返回最高优先级 slot（最低编号）在向量地址寄存器中的地址。否则返回默认向量地址寄存器中的地址。	0

保护使能寄存器（VICProtection - 0xFFFFF020，读/写）

该寄存器的 bit0 用来控制运行在用户模式下的软件对 VIC 寄存器的访问。

VICProtection 寄存器描述见表 5.62。

表 5.62 保护使能寄存器

VICProtection	功能	复位值
0	1：VIC 寄存器只能在特权模式下访问。 0：VIC 寄存器可在用户模式或特权模式下访问。	0

5.8.5 中断源

表 5.63 列出了每一个外设功能的中断源。每个外围设备都有一条中断线连接到向量中断控制器，但有些中断源可能拥有几个内部中断标志(比如 RTC 中断，就有 RTCCIF 和 RTCALF 两个中断标志)，或者单个中断标志也有可能代表一个以上的中断(比如 I^2C 中断，中断标志为 SI，包括了起始信号、发送数据和接收数据等等中断)。

中断源与 VIC 连接示意图见图 5.29。

表 5.63 连接到向量中断控制器的中断源

模块	标志	VIC 通道号
WDT	看门狗中断（WDINT）	0
-	保留给软件中断	1
ARM 内核	EmbeddedICE, DbgCommRx	2
ARM 内核	EmbeddedICE, DbgCommTx	3

接上表

模块	标志	VIC 通道号
定时器 0	匹配 0-3（MR0, MR1, MR2, MR3） 捕获 0-3（CR0, CR1, CR2, CR3）	4
定时器 1	匹配 0-3（MR0, MR1, MR2, MR3） 捕获 0-3（CR0, CR1, CR2, CR3）	5
UART0	Rx 线状态（RLS） 发送保持寄存器空（THRE） Rx 数据可用（RDA） 字符超时指示（CTI）	6
UART1	Rx 线状态（RLS） 发送保持寄存器空（THRE） Rx 数据可用（RDA） 字符超时指示（CTI） Modem 状态中断（MSI）	7
PWM0	匹配 0-6（MR0, MR1, MR2, MR3, MR4, MR5, MR6）	8
I^2C	SI（状态改变）	9
SPI0	SPI 中断标志（SPIF） 模式错误（MODF）	10
SPI1	SPI 中断标志（SPIF） 模式错误（MODF）	11
PLL	PLL 锁定（PLOCK）	12
RTC	计数器增加（RTCCIF） 报警（RTCALF）	13
系统控制	外部中断 0（EINT0）	14
系统控制	外部中断 1（EINT1）	15
系统控制	外部中断 2（EINT2）	16
系统控制	外部中断 3（EINT3）	17
A/D	A/D 转换器	18

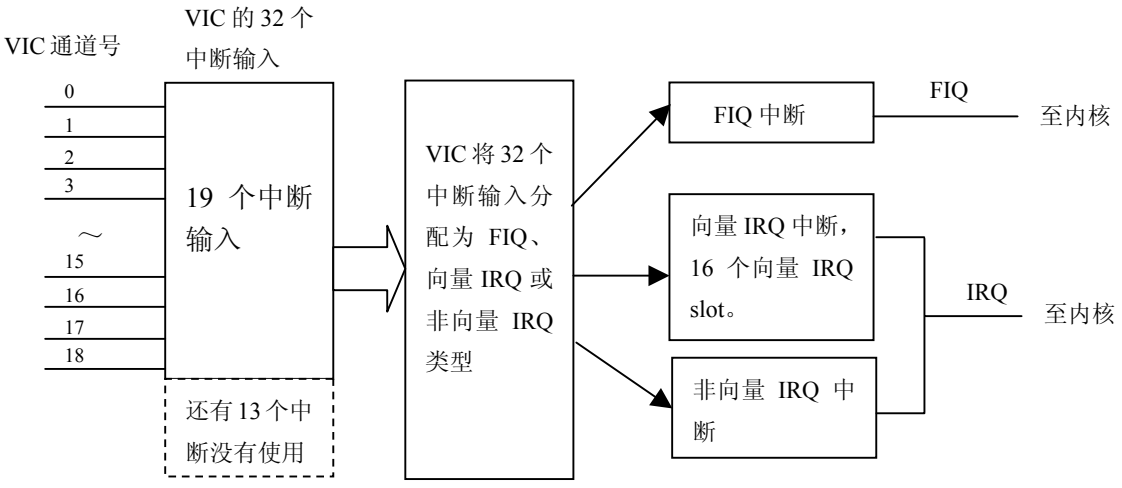


图 5.29 中断源与 VIC 连接示意图

5.8.6 VIC 使用事项

- **VIC 中断与片内 RAM 调试。**如果在片内 RAM 中调试程序(JTAG 调试)时需要使用中断，那么必须将中断向量重新映射到地址 0x00000000。这样做是因为所有的异常向量都位于地址 0x00000000 及以上。通过将寄存器 MEMMAP（位于系统控制模块当中）配置为用户 RAM 模式来实现这一点。另外，用户代码编译连接时应该使中断向量表装载到地址 0x40000000。
- **多个 FIQ 中断。**虽然可以选择多个中断源（通过 VICIntSelect）来产生 FIQ 请求，但是只有一个专门的中断服务程序来响应所有出现的 FIQ 请求。因此，如果分配为 FIQ 的中断多于一个，FIQ 中断服务程序就必须读取 VICFIQStatus 的内容来识别产生中断请求的 FIQ 中断源是哪一个，然后再进行相应中断处理。不过我们还是建议只将一个中断分配为 FIQ。多个 FIQ 中断源会增加中断处理程序的延迟。
- **IRQ 中断服务程序与 VIC 寄存器。**在中断服务程序执行完毕后，对外设中断标志的清零将会对 VIC 寄存器（VICRawIntr, VICFIQStatus 和 VICIRQStatus）当中的对应位产生影响。另外，为了能够服务下次中断，必须在中断返回之前对 VICVectAddr 寄存器执行一次写操作(写入的值一般为 0)，该写操作将清零内部中断优先级硬件当中对应的标志。
- **VIC 中断禁能操作。**若要禁止 VIC 中断，必须清零 VICIntEnable 寄存器中的对应位，这可以通过写 VICIntEnClr 寄存器实现。这同样应用于 VICSoftInt 和 VICSoftIntClear，VICSoftIntClear 将会使 VICSoftInt 中的对应位清零。例如，如果 VICSoftInt=0x00000005，需要将其 bit0 清零，那么 VICSoftIntClear=0x00000001 可实现该操作。向 VICSoftIntClear 寄存器任何位写入 1 对目标寄存器都是一次有效。
- **看门狗中断。**如果看门狗在溢出或无效喂狗时产生中断，那么无法清除中断。唯一的方法是通过 VICIntEnClr 禁止 VIC 中断，然后再中断返回。

举例：

假设 UART0 和 SPI0 产生中断请求，它们被分配为向量 IRQ（UART0 的优先级高于 SPI0），而 UART1 和 I^2C 产生非向量 IRQ，下面就是 VIC 一种初始化的示例：

VICIntSelect = 0x00000000	(SPI0, I^2C, UART1 和 UART0 为 IRQ => bit10, bit9, bit7 和 bit6=0)
VICIntEnable = 0x000006C0	(SPI0, I^2C, UART1 和 UART0 中断使能 => bit10, bit9, bit 7 和 bit6=1)
VICDefVectAddr = 0x…	(保存服务非向量 IRQ 的程序地址 ，即 UART1 和 I^2C 服务程序的起始地址)
VICVectAddr0 = 0x…	(保存 UART0 IRQ 服务程序的起始地址)
VICVectAddr1 = 0x…	(保存 SPI0 IRQ 服务程序的起始地址)
VICVectCntl0 = 0x00000026	(VIC 通道号为 6（UART0）的中断源使能为优先级 0 (最高优先级))
VICVectCntl1 = 0x0000002A	(VIC 通道号为 10（SPI0）中断源使能为优先级 1)

在任何 IRQ 请求（SPI0, I^2C, UART0 或 UART1）产生之后，微控制器跳转到地址 0x00000018 执行代码。对于向量和非向量 IRQ，可在地址 0x18 放入下面指令：

```
LDR    pc, [ pc, #-0xFF0 ]
```

该指令将 VICVectAddr 寄存器中保存的地址装入 PC。

一旦产生 UART0 请求，VICVectAddr 和 VICVectAddr0 相同。如果产生 SPI 请求，VICVectAddr 等于 VICVectAddr1。如果 UART0 和 SPI 都没有产生 IRQ 请求，而 UART1 和

/或 I^2C 产生请求，那么 VICVectAddr 的内容与 VICDefVectAddr 相同。

5.8.7 VIC 应用示例

1. VIC 基本操作方法

设置 IRQ/FIQ 中断，若是 IRQ 中断则可以设置为向量中断并分配中断优先级，否则为非向量 IRQ。然后可以设置中断允许，以及向量中断对应地址或非向量中断默认地址。当有中断后，若是 IRQ 中断，则可以读取向量地址寄存器，然后跳转到相应代码。当要退出中断时，对向量地址寄存器写 0，通知 VIC 中断结束。当发生中断时，处理器将会切换处理器模式，同时相关的寄存器也将会映射(如 R13、R14)。

对于中断源(VIC 通道)的 IRQ/FIQ 选择，由 VICIntSelect 寄存器控制，每一个中断源与 VICIntSelect 的各个位一一对应，比如 VIC 通道号 9(I^2C 中断)与 VICIntSelect 的 d9 位对应，设置该位为 1，则分配为 FIQ 中断，否则分配为 IRQ 中断。

2. 向量/非向量 IRQ 中断

如程序清单 5.13 所示，这是一个 IRQ 中断初始化的程序，程序设置了 EINT0 为 FIQ 中断，EINT1 为向量 IRQ 中断，EINT2 为非向量 IRQ 中断，并且设置了 IRQ 中断服务程序入口地址。

程序清单 5.13 向量/非向量 IRQ 中断初始化

```
VICIntSelect = 0x00004000;              // 设置 EINT0 为 FIQ 中断，其它中断为 IRQ 中断
VICVectCntl0 = 0x20 | 15;               // 设置 EINT1 为向量 IRQ 中断，使用 Slot0
VICVectAddr0 = (uint32) Eint1_IRQ;      // 设置 EINT1 中断服务程序入口地址
VICDefVectAddr = (uint32) Eint2_IRQ;    // 设置非向量 IRQ 中断服务程序入口地址
VICIntEnable = 0x0001C000;              // 使能 EINT0、EINT1 和 EINT2 中断允许
```

3. 中断服务程序

对于向量/非向量 IRQ 中断服务程序，需要在函数定义时加入__irq 关键字(这是 ADS1.2 编译器的关键字)，以保证函数返回时会切换处理器模式，如程序清单 5.14 所示。

程序清单 5.14 IRQ 中断服务程序编写

```
void  __irq  Eint1_IRQ(void)
{  …
   EXTINT = 0x02;
   VICVectAddr = 0;
}

void  __irq  Eint2_IRQ(void)
{  …
   EXTINT = 0x04;
   VICVectAddr = 0;
}
```

4. 使用 VIC 产生软件中断

VIC 提供了软件中断功能，第一个中断均可以使用软件中断产生，软件中断与对应通道上的硬件中断是逻辑“或”的关系。VIC 产生软件中断是进入到 IRQ/FIQ 模式，是进入 IRQ

模式还是 FIQ 模式与该通道中断设置有关(VICIntSelect 寄存器设置)。VIC 软件中断初始化例子如程序清单 5.15 所示。

VIC 软件中断是通过设置 VICSoftInt 发生，在中断处理程序中需要操作 VICSoftIntClear 来清除中断标志。

程序清单 5.15 VIC 软件中断初始化

```
VICIntSelect = 0x00000000;                  // 设置所有中断均为 IRQ 中断
VICVectCntl15 = 0x20 | 31;                  // 中断通道号 31 分配到 Slot15
VICVectAddr15 = (uint32) Soft_IRQ;          // 设置 VIC 软件中断服务程序入口地址
VICIntEnable = 1<<31;
```

5.8.8 启动代码相关部分

LPC2100、LPC2200 的启动代码中包含有 VIC 初始化程序，如程序清单 5.16 所示(在 target.c 文件中)。程序首先禁止所有中断(程序清单 5.16 (1))，设置 VICVectAddr 寄存器的值为 0(程序清单 5.16 (2))，最后将所有中断设置为 IRQ 中断(程序清单 5.16 (3))。

禁止所有中断是避免调试时一个中断没有响应就再次载入程序，因向量中断控制器状态错误而不能正确识别中断。

程序清单 5.16 TargetResetInit ()—VIC 初始化

```
void   TargetResetInit(void)
{
    …
    /* 初始化 VIC */
    VICIntEnClr = 0xffffffff;                                    (1)
    VICVectAddr = 0;                                             (2)
    VICIntSelect = 0;                                            (3)
    …
}
```

当用户使用 IRQ/FIQ 中断时，需要设置 CPSR 寄存器的 I 位或 F 位，并在用户主程序中设置 VIC 来使能相应片内外设的中断，设置片内外设中断使能。

一旦产生 IRQ 中断，微控制器即会切换到 IRQ 模式，并且跳转到向量表 0x00000018 地址执行程序。如程序清单 5.17 (7)所示，在 IRQ 向量使用的指令与其它向量不同，当 CPU 执行这条指令但还没有跳转时，PC 的值为 0x00000020(因为 ARM7TDMI 内核是三级流水线结构)，0x00000020 减去 0x00000ff0 为 0xFFFFF030，这是向量中断控制器（VIC）的特殊寄存器 VICVectAddr，这个寄存器保存当前将要服务的 IRQ 的中断服务程序的入口，用这一条指令就可以直接跳转到需要的中断服务程序中。

一旦产生 FIQ 中断，处理器即会切换到 FIQ 模式，并且跳转到向量表 0x0000001C 地址执行程序。如程序清单 5.17 (8、16)所示，程序将跳到 FIQ_Handler 标号处，处理 FIQ 中断服务程序。

程序清单 5.17 异常向量表—IRQ/FIQ 中断

```
Reset
        LDR     PC, ResetAddr                                    (1)
```

```
        LDR     PC, UndefinedAddr                       (2)
        LDR     PC, SWI_Addr                            (3)
        LDR     PC, PrefetchAddr                        (4)
        LDR     PC, DataAbortAddr                       (5)
        DCD     0xb9205f80                              (6)
        LDR     PC, [PC, #-0xff0]                       (7)
        LDR     PC, FIQ_Addr                            (8)

ResetAddr       DCD     ResetInit                       (9)
UndefinedAddr   DCD     Undefined                       (10)
SWI_Addr        DCD     SoftwareInterrupt               (11)
PrefetchAddr    DCD     PrefetchAbort                   (12)
DataAbortAddr   DCD     DataAbort                       (13)
Nouse           DCD     0                               (14)
IRQ_Addr        DCD     0                               (15)
FIQ_Addr        DCD     FIQ_Handler                     (16)
```

设置 CPSR 寄存器的 I 位或 F 位，需要在特权模式下进行，最简单的方法就是在启动代码中设置。如程序清单 5.18 所示(在 startup.s 文件中)，把程序清单 5.18 (2)从原来的 0xdf 更改为 0x5f，即清零 CPSR 寄存器的 I 位，使能 IRQ 总的中断允许。

程序清单 5.18 设置 CPSR 寄存器的 I 位

```
InitStack
        MOV     R0, LR                                  (1)
        …
;设置系统模式堆栈
        MSR     CPSR_c, #0x5f                           (2)
        LDR     SP, =StackUsr                           (3)

        MOV     PC, R0                                  (4)
```

5.9 GPIO

5.9.1 特性

- 单独位的方向控制，即每一个 I/O 口线可单独设置为输入/输出模式
- 单独控制 I/O 口输出的置位或清零
- 所有 I/O 口在复位后默认为输入

5.9.2 应用

- 通用 I/O 口
- 驱动 LED 或其它指示器
- 控制片外器件
- 检测数字输入，如键盘或开关信号

5.9.3 引脚描述

GPIO 引脚描述见表 5.64，这是 P0 和 P1 端口的 GPIO 引脚。除 P0 和 P1 外，LPC2210/2212/2214 还包含另外两个端口——P2 和 P3，这两个端口与外部存储器总线复用，只有在不用作外部存储器总线时(通过配置 PINSEL2 寄存器实现)，才可以作为 GPIO 使用，比如配置为 16 位总线接口时，D15～D31 可作为 GPIO。

表 5.64 GPIO 引脚描述

引脚名称	类型	描述
P0.0 – P0.31 P1.16 － P1.31	输入/输出	通用 I/O 口。实际可用的 GPIO 数量取决于可选功能的使用，即管脚连接模块的设置。

5.9.4 寄存器描述

寄存器汇总

LPC2114/2124 有 2 个 32 位的通用 I/O 口。P0 口使用了 30 个引脚(P0.0～P0.25，P0.27～P0.30)，P1 口有 16 个引脚可用作 GPIO 功能(P1.16～P1.31)。P0 口和 P1 口由 2 组 （每组 4 个） 寄存器控制，如表 5.65 所示。对于 LPC2210/2212/2214，P2 口的寄存器起始地址为 0xE0028020，P3 口的寄存器起始地址为 0xE0028030，如表 5.66 所示，各寄存器的功能与 P0 和 P1 的寄存器是一致的。

表 5.65 GPIO 寄存器映射—P0 和 P1

通用名称	描述	访问	复位值	PORT0 地址&名称	PORT1 地址&名称
IOPIN	GPIO 引脚值寄存器。不管方向和模式如何设定，引脚的当前状态都可从该寄存器中读出。	只读	NA	0xE0028000 IO0PIN	0xE0028010 IO1PIN
IOSET	GPIO 输出置位寄存器。该寄存器和 IOCLR 寄存器一起控制输出引脚的状态。写入 1 使对应引脚输出高电平。写入 0 无效。	读/置位	0x0000 0000	0xE0028004 IO0SET	0xE0028014 IO1SET
IODIR	GPIO 方向控制寄存器。该寄存器单独控制每个 I/O 口的方向。	读/写	0x0000 0000	0xE0028008 IO0DIR	0xE0028018 IO1DIR
IOCLR	GPIO 输出清零寄存器。该寄存器控制输出引脚的状态。写入 1 使对应引脚输出低电平并清零 IOSET 寄存器中的对应位。写入 0 无效。	只清零	0x0000 0000	0xE002800C IO0CLR	0xE002801C IO1CLR

表 5.66 GPIO 寄存器映射—P2 和 P3

通用名称	描述	访问	复位值	PORT2 地址&名称	PORT3 地址&名称
IOPIN	GPIO 引脚值寄存器。不管方向和模式如何设定，引脚的当前状态都可从该寄存器中读出。	只读	NA	0xE0028020 IO0PIN	0xE002830 IO1PIN
IOSET	GPIO 输出置位寄存器。该寄存器和 IOCLR 寄存器一起控制输出引脚的状态。写入 1 使对应引脚输出高电平。写入 0 无效。	读/置位	0x0000 0000	0xE0028024 IO0SET	0xE0028034 IO1SET
IODIR	GPIO 方向控制寄存器。该寄存器单独控制每个 I/O 口的方向。	读/写	0x0000 0000	0xE0028028 IO0DIR	0xE0028038 IO1DIR

接上表

通用名称	描述	访问	复位值	PORT2 地址&名称	PORT3 地址&名称
IOCLR	GPIO 输出清零寄存器。该寄存器控制输出引脚的状态。写入 1 使对应引脚输出低电平并清零 IOSET 寄存器中的对应位。写入 0 无效。	只清零	0x0000 0000	0xE002802C IO0CLR	0xE002803C IO1CLR

注意：LPC2210/2212/2214 才有 P2 和 P3 口。

GPIO 引脚值寄存器（IO0PIN - 0xE0028000,IO1PIN – 0xE0028010）

该寄存器提供 GPIO 引脚的值。它反映了外部环境对引脚的影响。

写该寄存器会将值保存到输出寄存器，可用于 I/O 测试。该特性在应用中几乎毫无用处，因为不可能对该寄存器中单个字节执行写操作。

IOPIN 寄存器描述见表 5.67。

表 5.67 GPIO 引脚值寄存器

IOPIN	描述	复位值
31:0	GPIO 引脚值。IO0PIN 的位 0 对应于 P0.0 ... 位 31 对应于 P0.31	未定义

GPIO 输出置位寄存器（IO0SET - 0xE0028004，IO1SET – 0xE0028014）

当引脚配置为 GPIO 输出模式时，可使用该寄存器从引脚输出高电平。写入 1 使对应引脚输出高电平。写入 0 无效。如果一个引脚被配置为输入或第二功能，写 IOSET 无效。

读 IOSET 寄存器返回 GPIO 输出寄存器中的值。该值由前一次对 IOSET 和 IOCLR（或前面提到的 IOPIN）的写操作决定。该值不反映任何外部环境对引脚的影响。

IOSET 寄存器描述见表 5.68。

表 5.68 GPIO 输出置位寄存器

IOSET	描述	复位值
31:0	输出置位。IO0SET 的位 0 对应于 P0.0 ... 位 31 对应于 P0.31	0

GPIO 输出清零寄存器（IO0CLR - 0xE002800C，IO1CLR – 0xE002801C）

当引脚配置为 GPIO 输出模式时，可使用该寄存器从引脚输出低电平。写入 1 使对应引脚输出低电平并清零 IOSET 寄存器中相应的位。写入 0 无效。如果一个引脚被配置为输入或第二功能，写 IOCLR 无效。

IOCLR 寄存器描述见表 5.69。

表 5.69 GPIO 输出清零寄存器

IOCLR	描述	复位值
31:0	输出清零。IO0CLR 的位 0 对应于 P0.0 ... 位 31 对应于 P0.31	0

GPIO 方向寄存器（IO0DIR - 0xE0028008，IO1DIR – 0xE0028018）

当引脚配置为 GPIO 模式时，可使用该寄存器控制引脚的方向。任意引脚的方向位的设置必须与引脚功能一致，比如某引脚用作输出功能，IODIR 寄存器的相应位必须设置为 1。

IODIR 寄存器描述见表 5.70。

表 5.70 GPIO 方向寄存器

IODIR	描述	复位值
31:0	方向控制位（0＝输入，1＝输出）。IO0DIR 的位 0 控制 P0.0 ... 位 31 控制 P0.31	0

5.9.5 GPIO 使用注意事项

如果指定输出引脚在 GPIO 输出置位寄存器（IOnSET）和 GPIO 输出清零寄存器（IOnCLR）中的对应位都置位，那么引脚的输出电平取决于后写入的寄存器的值。例如：

```
IO0SET = 0x0000 0080
IO0CLR = 0x0000 0080
```

P0.7 输出电平为低，因为写 GPIO 清零寄存器在写置位寄存器之后。

5.9.6 GPIO 应用示例

1. 设置 P0.0 口为输出模式

如程序清单 5.19 所示为设置 P0.0 口为 GPIO 输出模式的代码。

程序清单 5.19 设置 P0.0 为输出模式

```
PINSEL0 = 0x00000000;
IO0DIR = 0x00000001;
```

2. GPIO 读写操作

如程序清单 5.20 所示，程序将会读取 P0.7～P0.4 脚值，然后从 P0.3～P0.0 输出。

程序清单 5.20 GPIO 读写操作

```
bak = IO0PIN;                          // 读取引脚上的值
IO0CLR = 0x0000000F;                   // 将 P0.0～P0.3 输出 0
IO0SET = (bak&0x000000F0) >> 4;        // 设置 P0.0～P0.3 输出(为 1 的位输出 1)
```

3. 取反 P0.0 的输出

如所示，程序先从 IO0SET 读取当前输出寄存器的值，而不是去读引脚上的电平值(即读 IO0PIN)，然后判断 P0.0 的是输出高电平还是低电平，再控制输出相反。

程序清单 5.21 取反 P0.0 的输出

```
if((IO0SET&0x00000001) == 0)   IO0SET = 0x00000001;
   else   IO0CLR = 0x00000001;
```

5.10 UART 0

5.10.1 特性

- 16 字节接收 FIFO 和 16 字节发送 FIFO
- 寄存器位置符合 16C550 工业标准
- 接收器 FIFO 触发点可为 1, 4, 8 和 14 字节
- 内置波特率发生器

5.10.2 引脚描述

UART0 引脚描述见表 5.71。

表 5.71 UART0 引脚描述

引脚名称	类型	描述
RxD0	输入	**串行输入** 串行接收数据
TxD0	输出	**串行输出** 串行发送数据

5.10.3 应用

- 使用 UART0 与其它控制器进行数据交换，如图 5.30 所示。由于 LPC2000 的 I/O 电压为 3.3V（但 I/O 口可承受 5V 电压），所以连接时注意电平的匹配。
- 使用 UART0 与 PC 机通讯，如图 5.31 所示。由于 PC 机串口是 RS232 电平，所以连接时需要使用 RS232 转换器。LPC2000 就是通过 UART0 进行 ISP 操作的。

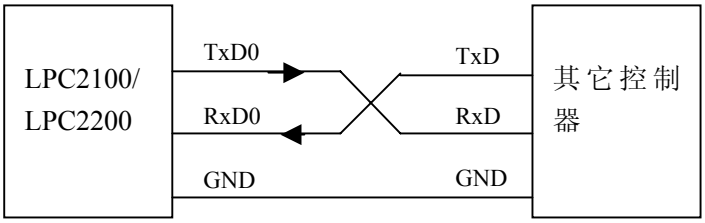


图 5.30 使用串口进行数据交换

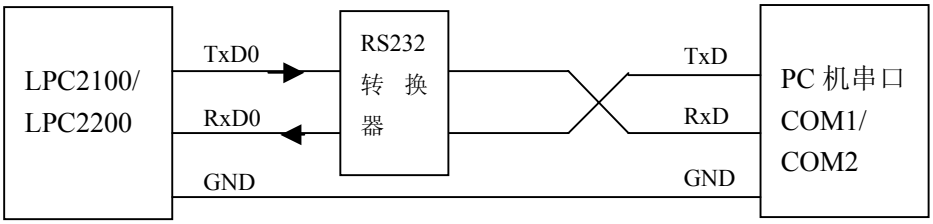


图 5.31 使用串口与 PC 机通讯

5.10.4 结构

UART0 的结构如图 5.32 所示。

VPB 接口提供 CPU 与 UART0 之间的通信连接。

UART0 接收器模块 U0Rx 监视串行输入线 RxD0 的有效输入。UART0 Rx 移位寄存器（U0RSR）通过 RxD0 接收有效的字符。当 U0RSR 接收到一个有效字符时，它将该字符传送到 UART0 Rx 缓冲寄存器 FIFO 中，等待 CPU 通过 VPB 接口进行访问。

UART0 发送器模块 U0Tx 接收 CPU 或主机写入的数据并将数据缓存到 UART0 Tx 保持寄存器 FIFO（U0THR）中。UART0 Tx 移位寄存器（U0TSR）读取 U0THR 中的数据并将数据通过串行输出引脚 TxD0 发送。

U0Tx 和 U0Rx 的状态信息保存在 U0LSR 中。U0Tx 和 U0Rx 的控制信息保存在 U0LCR 中。

UART0 波特率发生器模块 U0BRG 产生 UART0 Tx 模块所使用的定时。U0BRG 模块时钟源为 VPB 时钟（pclk）。主时钟与 U0DLL 和 U0DLM 寄存器所定义的除数相除得到 UART0 Tx 模块使用的时钟。该时钟必须为波特率的 16 倍。

中断接口包含寄存器 U0IER 和 U0IIR。中断接口接收几个由 U0Tx 和 U0Rx 发出的单时钟宽度的使能信号。

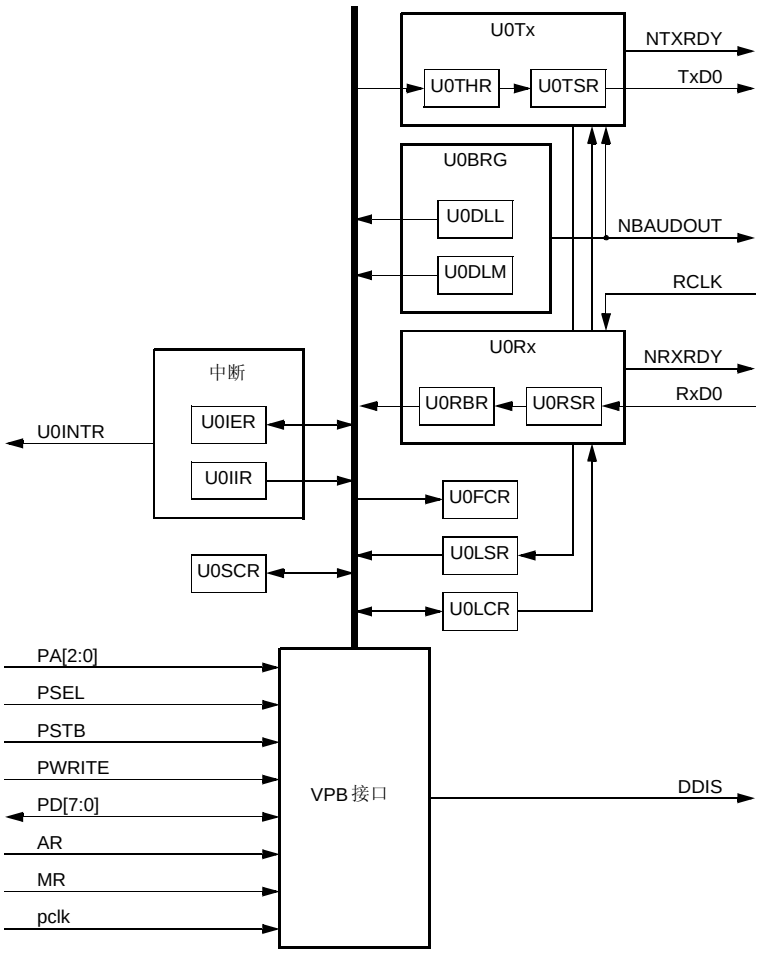


图 5.32 UART0 方框图

5.10.5 寄存器描述

寄存器汇总

UART0 包含 10 个 8 位寄存器，见表 5.72。除数锁存访问位（DLAB）位于 U0LCR 的 bit7，它使能对除数锁存的访问。

表 5.72 UART0 寄存器映射

名称	描述	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0	访问	复位值*	地址
U0RBR	接收缓冲	MSB			读数据				LSB	RO	未定义	0xE000C000 DLAB=0
U0THR	发送保持	MSB			写数据				LSB	WO	NA	0xE000C000 DLAB=0
U0IER	中断使能	0	0	0	0	0	使能 Rx 线状态中断	使能 THRE 中 断	使能 Rx 数据可用中断	R / W	0	0xE000C004 DLAB=0
U0IIR	中断 ID	FIFO 使能		0	0	IIR3	IIR2	IIR1	IIR0	RO	0x01	0xE000C008

接上表

名称	描述	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0	访问	复位值*	地址
U0FCR	FIFO 控制	Rx 触发		保留		-	Tx FIFO 复位	Rx FIFO 复位	FIFO 使能	WO	0	0xE000C008
U0LCR	线控制	DLAB	设置间隔	奇偶固定	偶选择	奇偶使能	停止位个数	字长度选择		R/W	0	0xE000C00C
U0LSR	线状态	Rx FIFO 错误	TEMT	THRE	BI	FE	PE	OE	DR	RO	0x60	0xE000C014
U0SCR	高速缓存	MSB							LSB	R/W	0	0xE000C01C
U0DLL	除数锁存 LSB	MSB							LSB	R/W	0x01	0xE000C000 DLAB=1
U0DLM	除数锁存 MSB	MSB							LSB	R/W	0	0xE000C004 DLAB=1

*：复位值仅指已使用位中保存的数据，不包括保留位的内容。

UART0 接收器缓存寄存器（U0RBR - 0xE000C000，DLAB=0，只读）

U0RBR 是 UART0 Rx FIFO(即接收 FIFO)的最高字节。它包含了最早接收到的字符，可通过总线接口读出。串口接收数据时低位在先，即 LSB(bit0)为最早接收到的数据位。如果接收到的数据小于 8 位，未使用的 MSB 填充为 0。

如果要访问 U0RBR，U0LCR 的除数锁存访问位（DLAB）必须为 0。U0RBR 为只读寄存器。

U0RBR 寄存器描述见表 5.73。

表 5.73 UART0 接收器缓存寄存器

U0RBR	功能	描述	复位值
7:0	接收器缓存	接收器缓存寄存器包含 UART0 Rx FIFO 当中最早接收到的字节	未定义

UART0 发送器保持寄存器（U0THR - 0xE000C000，DLAB=0，只写）

U0THR 是 UART0 Tx FIFO(即发送 FIFO)的最高字节。它包含了 Tx FIFO 中最新的字符，可通过总线接口写入。串口接收数据时低位在先，LSB(bit0)代表最先发送的位。

如果要访问 U0THR，U0LCR 的除数锁存访问位（DLAB）必须为 0。U0THR 为只写寄存器。

U0THR 寄存器描述见表 5.74。

表 5.74 UART0 发送器保持寄存器

U0THR	功能	描述	复位值
7:0	发送器保持	写 UART0 发送器保持寄存器使数据保存到 UART0 发送 FIFO 当中。当字节到达 FIFO 的最底部并且发送器就绪时，该字节将被发送。	N/A

UART0 除数锁存 LSB 寄存器（U0DLL - 0xE000C000，DLAB=1）

除数锁存是波特率发生器的一部分，它保存了用于产生波特率时钟的 VPB 时钟（pclk）分频值，波特率时钟必须是波特率的 16 倍。U0DLL 和 U0DLM 寄存器一起构成一个 16 位

除数，U0DLL 包含除数的低 8 位，U0DLM 包含除数的高 8 位。值 0x0000 被看作是 0x0001，因为除数是不允许为 0 的。由于 U0DLL 与 U0RBR/U0THR 共用同一地址，U0DLM 与 U0IER 共用同一地址，所以访问 UART0 除数锁存寄存器时，除数锁存访问位（DLAB）必须为 1，以确保寄存器的正确访问。

U0DLL 寄存器描述见表 5.75。

表 5.75 UART0 除数锁存 LSB 寄存器

U0DLL	功能	描述	复位值
7:0	除数锁存 LSB 寄存器	UART0 除数锁存 LSB 寄存器与 U0DLM 寄存器一起决定 UART0 的波特率。	0x01

UART0 除数锁存 MSB 寄存器（U0DLM - 0xE000C004，DLAB=1）

U0DLL 和 U0DLM 寄存器一起构成一个 16 位除数，用于产生波特率。

U0DLM 寄存器描述见表 5.76。

表 5.76 UART0 除数锁存 MSB 寄存器

U0DLM	功能	描述	复位值
7:0	除数锁存 MSB 寄存器	UART0 除数锁存 MSB 寄存器与 U0DLL 寄存器一起决定 UART0 的波特率。	0

UART0 中断使能寄存器（U0IER - 0xE000C004，DLAB=0）

U0IER 用于使能 4 个 UART0 中断源。**说明，RBR 中断包含了两个中断源，一是接收数据可用(RDA)中断，即正确接收到数据；二是接收超时中断(CTI)。**

U0IER 寄存器描述见表 5.77。

表 5.77 UART0 中断使能寄存器

U0IER	功能	描述	复位值
0	RBR 中断使能	0：禁止 RDA 中断 1：使能 RDA 中断 U0IER0 使能 UART0 接收数据可用中断。它还控制字符接收超时中断。	0
1	THRE 中断使能	0：禁止 THRE 中断 1：使能 THRE 中断 U0IER1 使能 UART0 THRE 中断。该中断的状态可从 U0LSR5 读出。	0
2	Rx 线状态中断使能	0：禁止 Rx 线状态中断 1：使能 Rx 线状态中断 U0IER2 使能 UART0 Rx 线状态中断。该中断的状态可从 U0LSR[4:1] 读出。	0
7:3	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

UART0 中断标识寄存器（U0IIR - 0xE000C008，只读）

U0IIR 提供状态代码用于指示一个挂起中断的中断源和优先级。在访问 U0IIR 过程中，中断被冻结。如果在访问 U0IIR 时产生了中断，该中断被记录，下次 U0IIR 访问可读出。

U0IIR 寄存器描述见表 5.78。

表 5.78 UART0 中断标识寄存器

U0IIR	功能	描述	复位值
0	中断挂起	0：至少有 1 个中断被挂起 1：没有挂起的中断 U0IIR0 为低有效。挂起的中断可通过 U0IER3:1 确定。	1
3:1	中断标识	011：1. 接收线状态（RLS） 010：2a. 接收数据可用（RDA） 110：2b. 字符超时指示（CTI） 001：3. THRE 中断 U0IER 的 bit3 指示对应于 UART0 Rx FIFO 的中断。上面未列出的 U0IER[3:1]的其它组合都为保留值（000, 100, 101, 111）	0
5:4	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
7:6	FIFO 使能	这些位等效于 U0FCR 的 bit0	0

UART0 中断源和中断使能关系如图 5.33 所示。

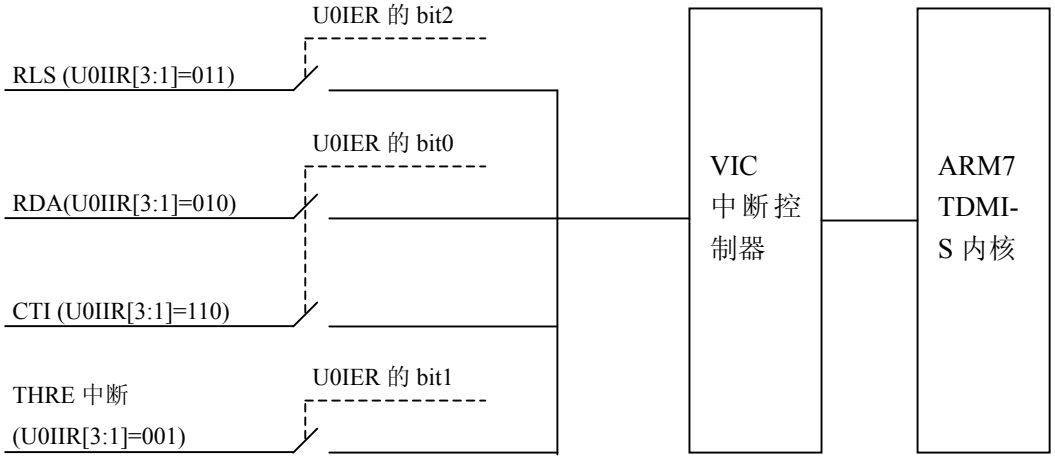


图 5.33 UART0 中断源和中断使能关系图

中断的处理见表 5.79。给定了 U0IIR[3:0]的状态，中断处理程序就能确定中断源以及如何清除激活的中断。**在退出中断服务程序之前，必须读取 U0IIR 来清除中断。**

表 5.79 UART0 中断处理

U0IIR[3:0]	优先级	中断类型	中断源	中断复位
0001	—	无	无	—
0110	最高	Rx 线状态/错误	OE, PE, FE, 或 BI	U0LSR 读操作
0100	第二	Rx 数据可用	Rx 数据可用或 FIFO 模式下（U0FCR0=1）到达触发点	U0RBR 读或 UART0 FIFO 低于触发值

接上表

U0IIR[3:0]	优先级	中断类型	中断源	中断复位
1100	第二	字符超时指示	Rx FIFO 包含至少 1 个字符并且在一段时间内无字符输入或移出，该时间的长短取决于 FIFO 中的字符数以及在 3.5 到 4.5 字符的时间内的触发值。 实际的时间为： [(字长度)×7-2]×8+[(触发值-字符数)×8+1]PCLK	U0RBR 读操作
0010	第三	THRE	THRE	U0IIR 读或 THR 写操作
注：“0000”, “0011”, “0101”, “0111”, “1000”, “1001”, “1010”, “1011”, “1101”, “1110”, “1111”为保留值。				

- **UART0 RLS 中断**（U0IIR[3:1]=011）是最高优先级的中断。只要 UART0 Rx 输入产生 4 个错误条件（溢出错误（OE）、奇偶错误（PE）、帧错误（FE）和间隔中断（BI））中的任意一个，该中断标志将置位。产生该中断的 UART0 Rx 错误条件可通过查看 U0LSR[4:1]得到。当读取 U0LSR 时清除中断。
- **UART0 RDA 中断**（U0IIR[3:1]=010）与 CTI 中断（U0IIR[3:1]=110）共用第二优先级。当 UART0 Rx FIFO 到达 U0FCR7:6 所定义的触发点时，RDA 被激活。当 UART0 Rx FIFO 的深度低于触发点时，RDA 复位。当 RDA 中断激活时，CPU 可读出由触发点所定义的数据块。
- **UART0 CTI 中断**（U0IIR[3:1]=110）为第二优先级中断。当 UART0 Rx FIFO 包含至少 1 个字符并且在接收 3.5 到 4.5 字符的时间内没有发生 UART0 Rx FIFO 动作时，产生该中断。UART0 Rx FIFO 的任何动作（读或写 UART0 RBR）都将清除该中断。当接收到的信息不是触发值的倍数时，CTI 中断将会清空 UART0 RBR。例如，如果一个外设想要发送一个 105 个字符的信息，而触发值为 10 个字符，那么前 100 个字符将使 CPU 接收 10 个 RDA 中断，而剩下的 5 个字符使 CPU 接收 1 到 5 个 CTI 中断（取决于服务程序）。
- **UART0 THRE 中断**(U0IIR[3:1]=001)为第三优先级中断。当 UART0 THR FIFO 为空并且满足特定的初始化条件时，该中断激活。这些初始化条件将使 UART0 THR FIFO 被数据填充，以免在系统启动时产生许多 THRE 中断。初始化条件在 THRE=1 时实现了一个字符的延时减去停止位并在上一次 THRE=1 事件之后没有在 U0THR 中存在至少 2 个字符。在没有译码和服务 THRE 中断时，该延迟为 CPU 提供了将数据写入 U0THR 的时间。如果 UART0 THR FIFO 中曾经有两个或更多字符，而当前 U0THR 为空时，THRE 中断立即设置。当发生 U0THR 写操作或 U0IIR 读操作并且 THRE 为最高优先级中断（U0IIR3:1=001）时，THRE 中断复位。

UART0 FIFO 控制寄存器（U0FCR - 0xE000C008）

U0FCR 控制 UART0 Rx 和 Tx FIFO 的操作。

U0FCR 寄存器描述见表 5.80。

表 5.80 UART0 FIFO 控制寄存器

U0FCR	功能	描述	复位值
0	FIFO 使能	为 1 时使能对 UART0 Rx 和 Tx FIFO 以及 U0FCR[7:1]的访问。该位必须置位以实现正确的 UART0 操作。该位的任何变化都将使 UART0 FIFO 清空。	0

接上表

U0FCR	功能	描述	复位值
1	Rx FIFO 复位	该位置位会清零 UART0 Rx FIFO 中的所有字节并复位指针逻辑。该位自动清零。	0
2	Tx FIFO 复位	该位置位会清零 UART0 Tx FIFO 中的所有字节并复位指针逻辑。该位自动清零。	0
5:3	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
7:6	Rx 触发选择	00：触发点 0（默认 1 字节） 01：触发点 1（默认 4 字节） 10：触发点 2（默认 8 字节） 11：触发点 3（默认 14 字节） 这两个位决定在激活中断之前，接收 UART0 FIFO 必须写入多少个字符。4 个触发点由用户在编译时定义，可以选择所需要的触发深度。	0

UART0 线控制寄存器（U0LCR - 0xE000C00C）

U0LCR 决定发送和接收数据字符的格式。

U0LCR 寄存器描述见表 5.81。

表 5.81 UART0 线控制寄存器

U0LCR	功能	描述	复位值
1:0	字长度选择	00：5 位字符长度 01：6 位字符长度 10：7 位字符长度 11：8 位字符长度	0
2	停止位选择	0：1 个停止位 1：2 个停止位（如果 U0LCR[1:0]=00 则为 1.5）	0
3	奇偶使能	0：禁止奇偶产生和校验 1：使能奇偶产生和校验	0
5:4	奇偶选择	00：奇数 01：偶数 10：强制为 1 11：强制为 0	0
6	间隔控制	0：禁止间隔发送 1：使能间隔发送 当 U0LCR 的 bit6 为 1 时，输出引脚 UART0 TxD 强制为逻辑 0。	0
7	除数锁存访问位	0：禁止访问除数锁存寄存器 1：使能访问除数锁存寄存器	0

UART0 线状态寄存器（U0LSR - 0xE000C014，只读）

U0LSR 为只读寄存器，它提供 UART0 Tx 和 Rx 模块的状态信息。

U0LSR 寄存器描述见表 5.82。

表 5.82 线状态寄存器

U0LSR	功能	描述	复位值
0	接收数据就绪（RDR）	0：U0RBR 为空 1：U0RBR 包含有效数据 当 U0RBR 包含未读取的字符时，RDR 位置位；当 UART0 RBR FIFO 为空时，RDR 位清零。	0
1	溢出错误（OE）	0：溢出错误状态未激活 1：溢出错误状态激活 溢出错误条件在错误发生后立即设置。U0LSR 读操作清零 OE 位。当 UART0 RSR 已经有新的字符就绪而 UART0 RBR FIFO 已满时，OE 位置位。此时 UART0 RBR FIFO 不会被覆盖，UART0 RSR 中的字符将丢失。	0
2	奇偶错误（PE）	0：奇偶错误状态未激活 1：奇偶错误状态激活 当接收字符的奇偶位处于错误状态时产生一个奇偶错误。U0LSR 读操作清零 PE 位。奇偶错误检测时间取决于 U0FCR 的 bit0。奇偶错误与 UART0 RBR FIFO 中读出的字符相关。	0
3	帧错误（FE）	0：帧错误状态未激活 1：帧错误状态激活 当接收字符的停止位为 0 时，产生帧错误。U0LSR 读操作清零 FE 位。帧错误检测时间取决于 U0FCR 的 bit0。帧错误与 UART0 RBR FIFO 中读出的字符相关。当检测到一个帧错误时，Rx 将尝试与数据重新同步并假设错误的停止位实际是一个超前的起始位。但即使没有出现帧错误，它也不能假设下一个接收到的字节是正确的。	0
4	间隔中断（BI）	0：间隔中断状态未激活 1：间隔中断状态激活 在发送整个字符（起始位、数据、奇偶位和停止位）过程中 RxD0 如果都保持逻辑 0，则产生间隔中断。当检测到中断条件时，接收器立即进入空闲状态直到 RxD0 变为全 1 状态。U0LSR 读操作清零该状态位。间隔检测的时间取决于 U0FCR 的 bit0。间隔中断与 UART0 RBR FIFO 中读出的字符相关。	0
5	发送保持寄存器空（THRE）	0：U0THR 包含有效数据 1：U0THR 空 当检测到 UART0 THR 空时，THRE 置位，U0THR 写操作清零该位。	1
6	发送器空（TEMT）	0：U0THR 和/或 U0TSR 包含有效数据 1：U0THR 和 U0TSR 空 当 U0THR 和 U0TSR 都为空时，TEMT 置位。当 U0TSR 或 U0THR 包含有效数据时，TEMT 清零。	1
7	Rx FIFO 错误（RXFE）	0：U0RBR 中没有 UART0 Rx 错误，或 U0FCR 的 bit0 为 0 1：U0RBR 包含至少一个 UART0 Rx 错误 当一个带有 Rx 错误（例如帧错误、奇偶错误或间隔中断）的字符装入 U0RBR 时，RXFE 位置位。当读取 U0LSR 寄存器并且 UART0 FIFO 中不再有错误时，RXFE 位清零。	0

UART0 高速缓存寄存器（U0SCR – 0Xe000C01C）

在 UART0 操作时 U0SCR 无效。用户可自由对该寄存器进行读或写。不提供中断接口向主机指示 U0SCR 所发生的读或写操作。

U0SCR 寄存器描述见表 5.83。

表 5.83 UART0 高速缓存寄存器

U0SCR	功能	描述	复位值
7:0	-	一个可读可写的字节	0

5.10.6 使用示例

LPC2114/2124/2210/2212/2214 的两个 UART，均具有 16 字节的收发 FIFO，寄存器位置符合 16C550 工业标准，内置波特率发生器，两个串口具有基本相同的寄存器，其中 UART1 带有完全的调制解调器控制握手接口。在使用 UART 与上位机 PC 通讯时，需要一个 RS232 电平转换电路，如 SP3243ECA(或 MAX3243ECA)芯片等。UART0 的基本寄存器功能框图如图 5.34 所示。

其中，寄存器 U0RBR 与 U0THR 是同一地址，但物理上是分开的，读操作时为 U0RBR，而写操作时为 U0THR；寄存器 U0DLL 与 U0RBR/U0THR、U0DLM 与 U0TER 具有同样的地址，如果要访问 U0DLM、U0DLL，除数访问位 DLAB 必须为 1，若要访问 U0RBR/U0THR、U0IER，则除数访问位 DLAB 必须为 0。图 5.34 中，U0DLM 和 U0DLL 寄存器是波特率发生器的除数锁存寄存器，用于设置合适的串口波特率；U0RBR 为数据接以缓冲，用于读取接收到的数据，若 FIFO 使能，串口接收到的数据会压入 FIFO 缓冲；U0THR 为发送保存，向此寄存器写入数据时，将会引起串口数据发送，若 FIFO 使能，数据会压入 FIFO 缓冲。

波特率的除数计算如下：

$$UxDLM、UxDLL = \frac{F_{PCLK}}{16 \times baud}$$

其中，baud 为所需要的波特率。

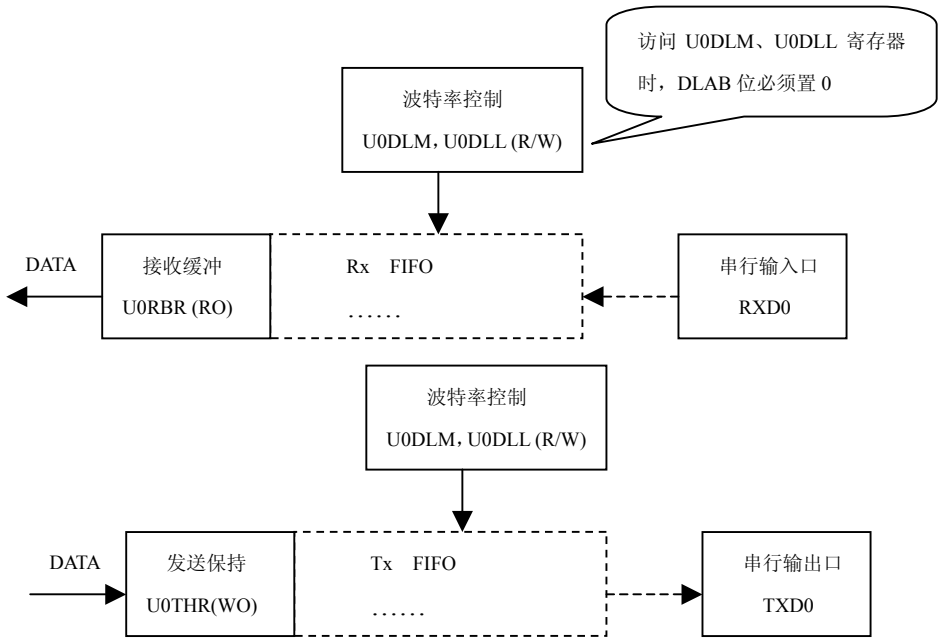


图 5.34 UART 的基本寄存器功能框图

如图 5.35 所示，通过线控制寄存器 U0LCR 设置串口的工作模式，而 U0FCR 则用于 FIFO 使能或复位操作；当接收或发送数据时，会产生相应的状态标志位(U0LSR)；通过对 U0IER 进行设置，可实现串口的发送、接收、出错中断等。注意，U0IER 中的位 0 为接收中断使能，位 1 为发送中断使能，位 2 为线状态中断使能(通讯出错中断使能)，若不使能相应的中断，对应的中断标志是不会产生，此时可以通过 U0LSR 读取串口的状态判断串口操作是否完成或是否成功。

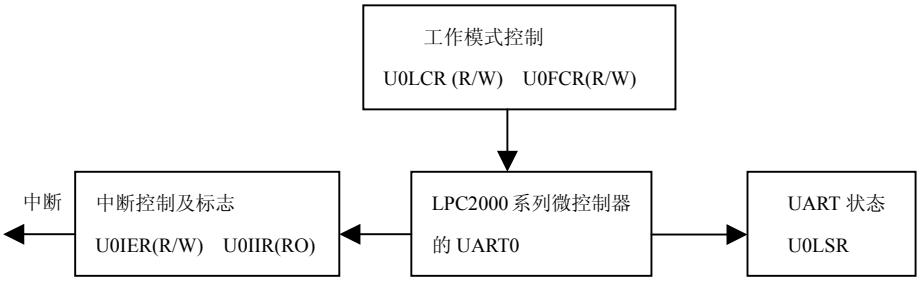


图 5.35 UART0 的模式寄存器功能框图

UART0 的基本操作方法：

- 设置 I/O 连接到 UART0；
- 设置串口波特率 (U0DLM、U0DLL)；
- 设置串口工作模式 (U0LCR、U0FCR)；
- 发送或接收数据 (U0THR、U0RBR)；
- 检查串口状态字或等待串口中断 (U0LSR)。

1. UART0 初始化设置

程序清单 5.22 为 UART0 初始化示例，程序将串口波特率设置为 UART_BPS(如 115200)，8 位数据长度，1 位停止位，无奇偶校验。

程序清单 5.22 UART0 初始化示例

```
#define   UART_BPS       115200          /* 定义通讯波特率 */
/*****************************************************************************
* 名称：UART0_Ini()
* 功能：初始化串口 0。设置为 8 位数据位，1 位停止位，无奇偶校验，波特率为 115200
* 入口参数：无
* 出口参数：无
*****************************************************************************/
void   UART0_Ini(void)
{   uint16 Fdiv;
    U0LCR = 0x83;                       // DLAB = 1，可设置波特率
    Fdiv = (Fpclk / 16) / UART_BPS;     // 设置波特率
    U0DLM = Fdiv / 256;
    U0DLL = Fdiv % 256;
    U0LCR = 0x03;
}
```

2. 发送数据

采用查询方式发送一字节数据，如程序清单 5.23 所示。

程序清单 5.23 UART0 查询方式发送数据

```
/*****************************************************************************
* 名称：UART0_SendByte()
* 功能：向串口发送字节数据，并等待发送完毕。
* 入口参数：data            要发送的数据
* 出口参数：无
*****************************************************************************/
void   UART0_SendByte(uint8 data)
{   U0THR = data;                              // 发送数据
    while( (U0LSR&0x40)==0 );                  // 等待数据发送完毕
}
```

3. 接收数据

采用查询方式接收一字节数据，如程序清单 5.24 所示。

程序清单 5.24 UART0 查询方式接收数据

```
/*****************************************************************************
* 名称：UART0_RcvByte()
* 功能：从串口接收字节数据。使用查询方式。
* 入口参数：无
* 出口参数：返回接收到的数据
*****************************************************************************/
uint8   UART0_RcvByte(void)
{    uint8   rcv_data;
     while((U0LSR&0x01) == 0);
     rcv_data = U0RBR;
     return(rcv_data);
}
```

5.11 UART1

5.11.1 特性

- UART1 与 UART0 相同，只是增加了一个调制解调器（Modem）接口
- 16 字节接收 FIFO 和 16 字节发送 FIFO
- 寄存器位置符合 16C550 工业标准
- 接收器 FIFO 触发点可为 1, 4, 8 和 14 字节
- 内置波特率发生器
- 包含标准调制解调器接口信号

5.11.2 引脚描述

UART1 引脚描述见表 5.84。

表 5.84 UART1 引脚描述

引脚名称	类型	描述
RxD1	输入	**串行输入** 串行接收数据
TxD1	输出	**串行输出** 串行发送数据
CTS1	输入	**清除发送** 指示外部 modem 的接收是否已经准备就绪，低电平有效，UART1 数据可通过 TxD1 发送。在 modem 的正常操作中(U1MCR 的 bit4 为 0)，该信号的补码保存在 U1MSR 的 bit4 中。状态改变信息保存在 U1MSR 的 bit0 中，如果第 4 优先级中断使能(U1IER 的 bit3 为 1)，该信息将作为中断源。
DCD1	输入	**数据载波检测** 指示外部 modem 是否已经与 UART1 建立了通信连接，低电平有效，可以进行数据交换。在 modem 的正常操作中(U1MCR 的 bit4 为 0)，该信号的补码保存在 U1MSR 的 bit7 中。状态改变信息保存在 U1MSR 的 bit3 中，如果第 4 优先级中断使能(U1IER 的 bit3 为 1)，该信息将作为中断源。
DSR1	输入	**数据设备就绪** 指示外部 modem 是否准备建立与 UART1 的连接，低电平有效。在 modem 的正常操作中(U1MCR 的 bit4 为 0)，该信号的补码保存在 U1MSR 的 bit5 中。状态改变信息保存在 U1MSR 的 bit1 中，如果第 4 优先级中断使能(U1IER 的 bit3 为 1)，该信息将作为中断源。
DTR1	输出	**数据终端就绪** 有效低电平指示 UART1 准备建立与外部 modem 的连接。该信号的补码保存在 U1MCR 的 bit0 中。
RI1	输入	**铃响指示** 指示 modem 检测到电话的响铃信号，低电平有效。在 modem 的正常操作中(U1MCR 的 bit4 为 0)，该信号的补码保存在 U1MSR 的 bit6 中。状态改变信息保存在 U1MSR 的 bit2 中，如果第 4 优先级中断使能(U1IER 的 bit3 为 1)，该信息将作为中断源。
RTS1	输出	**请求发送** 指示 UART1 打算向外部 modem 发送数据，低电平有效。该信号的补码保存在 U1MCR 的 bit1 中。

5.11.3 应用

通过对 PINSEL0 寄存器设置来决定是否使用 UART1 的 MODEM 接口，当使用 MODEM 接口时，需要一个 RS232 转换器将信号转换为 RS232 电平后，才能与 MODEM 连接，如图 5.36 所示。

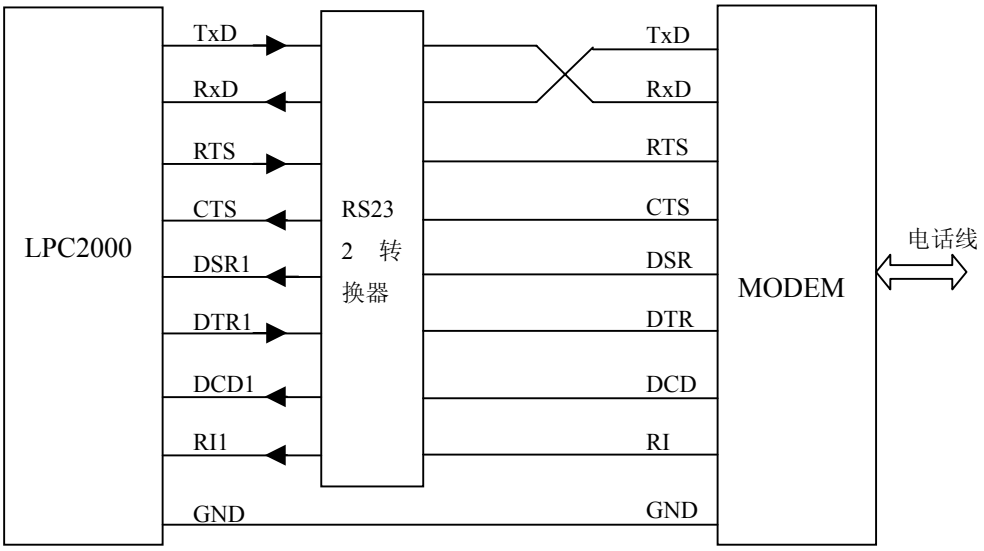


图 5.36 UART1 与 MODEM 接口电路

当不使用 MODEM 接口功能时，UART1 与 UART0 是一样，只需要 TxD1、RxD1 和 GND 引脚即可进行串口通讯，此时 UART1 的其它口线可以作为 GPIO 使用。

5.11.4 结构

UART1 的结构如图 5.37 所示。

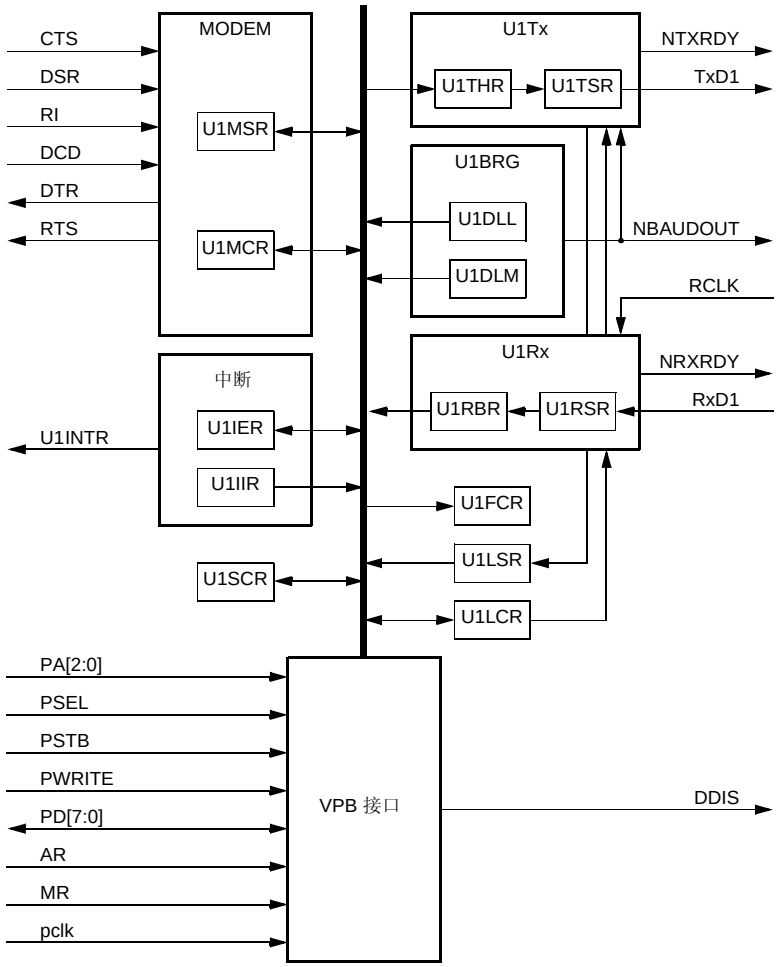


图 5.37 UART1 方框图

VPB 接口提供了 CPU 与 UART1 之间的通信连接。

UART1 接收器模块 U1Rx 监视串行输入线 RxD1 的有效输入。UART1 Rx 移位寄存器（U1RSR）通过 RxD1 接受有效的字符。当 U1RSR 接收到一个有效字符时，它将该字符传送到 UART1 Rx 缓冲寄存器 FIFO 中，等待 CPU 通过 VPB 接口进行访问。

UART1 发送器模块 U1Tx 接受 CPU 写入的数据并将数据缓存到 UART1 Tx 保持寄存器 FIFO (U1THR)中。UART1 Tx 移位寄存器（U1TSR）读取 U1THR 中的数据并将数据通过串行输出引脚 TxD1 发送。

U1Tx 和 U1Rx 的状态信息保存在 U1LSR 中。U1Tx 和 U1Rx 的控制信息保存在 U1LCR 中。

UART1 波特率发生器模块 U1BRG 产生 UART1 Tx 模块所使用的定时时间。U1BRG 模块时钟源为 VPB 时钟（pclk）。主时钟与 U1DLL 和 U1DLM 寄存器所定义的除数相除得到 Tx 模块使用的时钟。该时钟必须为波特率的 16 倍。

Modem 接口包含寄存器 U1MCR 和 U1MSR。该接口负责一个 Modem 外设与 UART1 之间的握手。

中断接口包含寄存器 U1IER 和 U1IIR。中断接口接收几个由 U1Tx，U1Rx 和 Modem 模块发出的单时钟宽度的使能信号。

5.11.5 寄存器描述

寄存器汇总

UART1 包含 12 个 8 位寄存器，见表 5.85。除数锁存访问位（DLAB）位于 U0LCR 的 bit7，它使能对除数锁存的访问。

表 5.85 UART1 寄存器映射

名称	描述	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0	访问	复位值*	地址偏移
U1RBR	接收缓冲	MSB				读数据			LSB	RO	未定义	0xE0010000 DLAB=0
U1THR	发送保持	MSB				写数据			LSB	WO	NA	0xE0010000 DLAB=0
U1IER	中断使能	0	0	0	0	使能 Modem 状态中断	使能 Rx 线状态中断	使能 THRE 中断	使能 Rx 数据可用中断	R/W	0	0xE0010004 DLAB=0
U1IIR	中断 ID	FIFO 使能		0	0	IIR3	IIR2	IIR1	IIR0	RO	0x01	0xE0010008
U1FCR	FIFO 控制	Rx 触发		保留		-	Tx FIFO 复位	Rx FIFO 复位	FIFO 使能	WO	0	0xE0010008
U1LCR	线控制	DLAB	设置间隔	奇偶固定	偶选择	奇偶使能	停止位个数	字长度选择		R/W	0	0xE001000C
U1MCR	Modem 控制	0	0	0	回送	0	0	RTS	DTR	R/W	0	0xE0010010
U1LSR	线状态	Rx FIFO 错误	TEMT	THRE	BI	FE	PE	OE	DR	RO	0x60	0xE0010014
U1MSR	Modem 状态	DCD	RI	DSR	CTS	Delta DCD	后沿 RI	Delta DSR	Delta CTS	RO	0	0xE0010018
U1SCR	高速缓存	MSB							LSB	R/W	0	0xE001001C
U1DLL	除数锁存 LSB	MSB							LSB	R/W	0	0xE0010000 DLAB=1
U1DLM	除数锁存 MSB	MSB							LSB	R/W	0	0xE0010004 DLAB=1

* 复位值仅指已使用位中保存的数据，不包括保留位的内容。

UART1 接收器缓存寄存器（U1RBR - 0xE0010000，DLAB=0，只读）

U1RBR 是 UART1 Rx FIFO(即接收 FIFO)的最高字节，见表 5.86。它包含了最早接收到的字符，可通过总线接口读出。串口接收数据时低位在先，即 LSB(bit0)为最早接收到的数据位。如果接收到的字符小于 8 位，未使用的 MSB 填充为 0。

如果要访问 U1RBR，U1LCR 的除数锁存访问位（DLAB）必须为 0。U1RBR 为只读寄存器。

U1RBR 寄存器描述见表 5.86。

表 5.86 UART1 接收器缓存寄存器

U1RBR	功能	描述	复位值
7:0	接收器缓存	接收器缓存寄存器包含 UART1 Rx FIFO 当中最早接收到的字节	未定义

UART1 发送器保持寄存器（U1THR - 0xE0010000，DLAB=0，只写）

U1THR 是 UART1 Tx FIFO(即发送 FIFO)的最高字节，见表 5.87。它包含了 Tx FIFO 中最新的字符，可通过总线接口写入。串口接收数据时低位在先，LSB(bit0)代表最先发送的位。

如果要访问 U1THR，U1LCR 的除数锁存访问位（DLAB）必须为 0。U1THR 为只写寄存器。

U1THR 寄存器描述见表 5.87。

表 5.87 UART1 发送器保持寄存器

U1THR	功能	描述	复位值
7:0	发送器保持	写发送器保持寄存器使数据保存到 UART1 发送 FIFO 当中。当字节到达 FIFO 的最低部并且发送器就绪时，该字节将被发送。	N/A

UART1 除数锁存 LSB 寄存器（U1DLL - 0xE0010000，DLAB=1）

UART1 的除数锁存是波特率发生器的一部分，它保存了用于产生波特率时钟的 VPB 时钟（pclk）分频值，波特率时钟必须是波特率的 16 倍，见表 5.88、表 5.89。U1DLL 和 U1DLM 寄存器一起构成一个 16 位除数，U1DLL 包含除数的低 8 位，U1DLM 包含除数的高 8 位。值 0x0000 被看作是 0x0001，因为除数是不允许为 0 的。由于 U0DLL 与 U0RBR/U0THR 共用同一地址，U0DLM 与 U0IER 共用同一地址，所以访问 UART1 除数锁存寄存器时，U1LCR 的除数锁存访问位（DLAB）必须为 1。以确保寄存器的正确访问。

U1DLL 寄存器描述见表 5.88。

表 5.88 UART1 除数锁存 LSB 寄存器

U1DLL	功能	描述	复位值
7:0	除数锁存 LSB 寄存器	UART1 除数锁存 LSB 寄存器与 U1DLM 寄存器一起决定 UART1 的波特率。	0x01

UART1 除数锁存 MSB 寄存器（U1DLM - 0xE0010004，DLAB=1）

U1DLL 和 U1DLM 寄存器一起构成一个 16 位除数，用于产生波特率。

U1DLM 寄存器描述见表 5.89。

表 5.89 UART1 除数锁存 MSB 寄存器

U1DLM	功能	描述	复位值
7:0	除数锁存 MSB 寄存器	UART1 除数锁存 MSB 寄存器与 U1DLL 寄存器一起决定 UART1 的波特率。	0

UART1 中断使能寄存器（U1IER - 0xE0010004，DLAB=0）

U1IER 用于使能 5 个中断源，见表 5.90。**说明，RBR 中断包含了两个中断源，一是接收数据可用(RDA)中断，即正确接收到数据；二是接收超时中断(CTI)。**

U1IER 寄存器描述见表 5.90。

表 5.90 UART1 中断使能寄存器

U1IER	功能	描述	复位值
0	RBR 中断使能	0：禁止 RDA 中断 1：使能 RDA 中断 U1IER0 使能 UART1 的接收数据可用中断。它还控制着接收超时中断。	0
1	THRE 中断使能	0：禁止 THRE 中断 1：使能 THRE 中断 U1IER1 使能 UART1 的 THRE 中断。该中断的状态可从 U1LSR5 读出。	0
2	Rx 线状态中断使能	0：禁止 Rx 线状态中断 1：使能 Rx 线状态中断 U1IER2 使能 UART1 Rx 线状态中断。该中断的状态可从 U1LSR[4:1]读出。	0
3	Modem 状态中断使能	0：禁止 Modem 中断 1：使能 Modem 中断 U1IER3 使能 modem 中断。中断的状态可从 U1MSR[3:0]读取。	0
7:4	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

UART1 中断标识寄存器（U1IIR - 0xE0010008，只读）

U1IIR 提供状态代码用于指示一个挂起中断的中断源和优先级，见表 5.91。在访问 U1IIR 过程中，中断被冻结。如果在访问 U1IIR 时产生了中断，该中断被记录，下次 U1IIR 访问可读出。

表 5.91 UART1 中断标识寄存器

U1IIR	功能	描述	复位值
0	中断挂起	0：至少有 1 个中断被挂起 1：没有挂起的中断 U1IIR0 为低有效。挂起的中断可通过 U1IIR3:1 确定。	1

接上表

U1IIR	功能	描述	复位值
3:1	中断标识	011：1. 接收线状态（RLS） 010：2a. 接收数据可用（RDA） 110：2b. 字符超时指示（CTI） 001：3. THRE 中断 000：4. Modem 中断 U1IER 的 bit3 指示对应于 UART1 Rx FIFO 的中断。U1IER3 指示对应于 UART1 Rx FIFO 的中断。上面未列出的 U0IER[3:1]的其它组合都为保留值（100, 101, 111）	0
5:4	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
7:6	FIFO 使能	这些位等效于 U1FCR0	0

UART1 中断源和中断使能关系如图 5.38 所示。

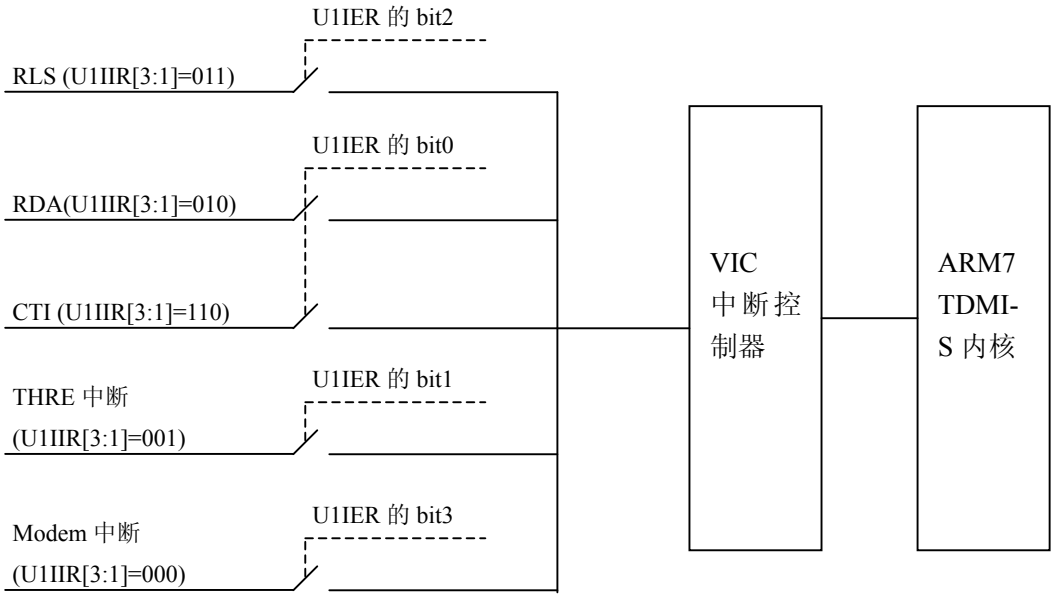


图 5.38 UART1 中断源和中断使能关系图

中断的处理见表 5.92。给定了 U1IIR[3:0]的状态，中断处理程序就能确定中断源以及如何清除激活的中断。**在退出中断服务程序之前，必须读取 U1IIR 来清除中断。**

表 5.92 UART1 中断处理

U1IIR[3:0]	优先级	中断类型	中断源	中断复位
0001	—	无	无	—
0110	最高	Rx 线状态/错误	OE, PE, FE, 或 BI	U1LSR 读操作
0100	第二	Rx 数据可用	Rx 数据可用或 FIFO 模式下（FCR0=1）到达触发点	U1RBR 读或 FIFO 低于触发值

接上表

U1IIR[3:0]	优先级	中断类型	中断源	中断复位
1100	第二	字符超时指示	Rx FIFO 包含至少 1 个字符并且在一段时间内无字符输入或移出，该时间的长短取决于 FIFO 中的字符数以及在（3.5 到 4.5 字符的时间内）的触发值。实际的时间为： [(字长度)×7-2]×8+[(触发值-字符数)×8+1]PCLK	U1RBR 读操作
0010	第三	THRE	THRE	U1IIR 读 （如果是中断源）或 THR 写操作
0000	第四	Modem 状态	CTS, DSR, RI 或 DCD	MSR 读操作
注：“0011”, “0101”, “0111”, “1000”, “1001”, “1010”, “1011”, “1101”, “1110”, “1111”为保留值。				

- **UART1 RLS 中断**（U1IIR[3:1]=011）是最高优先级的中断。只要 UART1 RX 输入产生 4 个错误条件（溢出错误（OE）、优先级错误（PE）、帧错误（FE）和间隔中断（BI））中的任意一个，该中断标志将置位。产生该中断的 UART1 Rx 错误条件可通过查看 U1LSR4:1 得到。当读取 U1LSR 时清除中断。
- **UART1 RDA 中断**（U1IIR[3:1]=010）与 CTI 中断（U1IIR3:1=110）共用第二优先级。当 UART1 Rx FIFO 到达 U1FCR7:6 所定义的触发点时，RDA 被激活。当 UART1 Rx FIFO 的深度低于触发点时，RDA 复位。当 RDA 中断激活时，CPU 可读出由触发点所定义的数据块。
- **CTI 中断**（U1IIR[3:1]=110）为第二优先级中断。当 UART1 Rx FIFO 包含至少 1 个字符并且在接收 3.5 到 4.5 字符的时间内没有发生 UART1 Rx FIFO 动作时，产生该中断。UART1 Rx FIFO 的任何动作（读或写 UART1 RBR）都将清除该中断。当接收到的信息不是触发值的倍数时，CTI 中断将会清空 UART1 RBR。例如，如果一个外设想要发送一个 105 个字符的信息，而触发值为 10 个字符，那么前 100 个字符将使 CPU 接收 10 个 RDA 中断，而剩下的 5 个字符使 CPU 接收 1 到 5 个 CTI 中断（取决于服务程序）。
- **UART1 THRE 中断**(U1IIR[3:1]=001)为第三优先级中断。当 UART1 THR FIFO 为空并且满足特定的初始化条件时，该中断激活。这些初始化条件将使 UART1 THR FIFO 被数据填充，以免在系统启动时产生许多 THRE 中断。初始化条件在 THRE=1 时实现了一个字符的延时减去停止位并在上一次 THRE=1 事件之后没有在 U1THR 中存在至少 2 个字符。在没有译码和服务 THRE 中断时，该延迟为 CPU 提供了将数据写入 U1THR 的时间。如果 UART1 THR FIFO 中曾经有两个或更多字符，而当前 U1THR 为空时，THRE 中断立即设置。当发生 U1THR 写操作或 U1IIR 读操作并且 THRE 为最高优先级中断（U1IIR3:1=001）时，THRE 中断复位。
- **Modem 中断**（U1IIR[3:1]=000）是最低优先级中断，只要在 modem 输入引脚 DCD、DSR 或 CTS 上发生任何状态变化，该中断就会激活。此外，modem 输入口 RI 上低到高电平的跳变会产生一个 modem 中断。modem 中断源可通过检查 U1MSR[3:0] 得到。读取 U1MSR 将清除 modem 中断。

UART1 FIFO 控制寄存器（U1FCR - 0xE0010008）

U1FCR 控制 UART1 Rx 和 Tx FIFO 的操作，见表 5.93。

表 5.93 UART1 FIFO 控制寄存器

U1FCR	功能	描述	复位值
0	FIFO 使能	为 1 时使能对 UART1 Rx 和 Tx FIFO 以及 U1FCR7:1 的访问。该位必须置位以实现正确的UART1 操作。该位的任何变化都将使UART1 FIFO 清空。	0
1	Rx FIFO 复位	该位置位会清零 UART1 Rx FIFO 中的所有字节并复位指针逻辑。该位自动清零。	0
2	Tx FIFO 复位	该位置位会清零 UART1 Tx FIFO 中的所有字节并复位指针逻辑。该位自动清零。	0
5:3	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
7:6	Rx 触发选择	00：触发点 0（默认 1 字节） 01：触发点 1（默认 4 字节） 10：触发点 2（默认 8 字节） 11：触发点 3（默认 14 字节） 这两个位决定在激活中断之前，UART1 FIFO 必须写入多少个字符。4 个触发点由用户在编译时定义，可以选择所需要的触发深度。	0

UART1 线控制寄存器（U1LCR - 0xE001000C）

U1LCR 决定发送和接收数据字符的格式，见表 5.94。

表 5.94 UART1 线控制寄存器

U1LCR	功能	描述	复位值
1:0	字长度选择	00：5 位字符长度 01：6 位字符长度 10：7 位字符长度 11：8 位字符长度	0
2	停止位选择	0：1 个停止位 1：2 个停止位（如果 U1LCR[1:0]=00 则为 1.5）	0
3	奇偶使能	0：禁止奇偶产生和校验 1：使能奇偶产生和校验	0
5:4	奇偶选择	00：奇数 01：偶数 10：强制为 1 11：强制为 0	0
6	间隔控制	0：禁止间隔发送 1：使能间隔发送 当 U1LCR6 的 bit6 为 1 时，输出引脚 UART1 TxD 强制为逻辑 0。	0
7	除数锁存访问位	0：禁止访问除数锁存寄存器 1：使能访问除数锁存寄存器	0

UART1 Modem 控制寄存器（U1MCR - 0xE0010010）

U1MCR 使能 modem 的回写模式并控制 modem 的输出信号，见表 5.95。

表 5.95 U1RT1 Modem 控制寄存器

U1MCR	功能	描述	复位值
0	DTR 控制	选择 modem 输出引脚 DTR。该位在回写模式激活时读出为 0。	0
1	RTS 控制	选择 modem 输出引脚 RTS。该位在回写模式激活时读出为 0。	
2	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
3	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
4	回写模式选择	0：禁止 modem 回写模式 1：使能 modem 回写模式 modem 回写模式提供了一个执行回写测试的诊断机制。发送器输出的串行数据在内部连接到接收器的串行输入端。输入脚 RxD1 对回写模式无影响，输出脚 TxD1 保持总为 1 的状态。4 个 modem 输入（CTS, DSR, RI 和 DCD）与外部断开。从外部来看，modem 的输出端（RTS, DTR）无效。在内部，4 个 modem 输出连接到 4 个 modem 输入。这样连接的结果是 U1MSR 的高 4 位由 U1MCR 的低 4 位驱动，而不是在正常模式下由 4 个 modem 输入驱动。这样在回写模式下，写 U1MCR 的低 4 位就可产生 modem 状态中断。	0
7:5	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

UART1 线状态寄存器（U1LSR - 0xE0010014，只读）

U1LSR 为只读寄存器，它提供 UART1 Tx 和 Rx 模块的状态信息，见表 5.96。

表 5.96 UART1 线状态寄存器

U1LSR	功能	描述	复位值
0	接收数据就绪（RDR）	0：U1RBR 为空 1：U1RBR 包含有效数据 当 U1RBR 包含未读取的字符时，RDR 位置位；当 UART1 RBR FIFO 为空时，RDR 位清零。	0
1	溢出错误（OE）	0：溢出错误状态未激活 1：溢出错误状态激活 溢出错误条件在错误发生后立即设置。U1LSR 读操作清零 OE 位。当 UART1 RSR 已经有新的字符就绪而 UART1 RBR FIFO 已满时，OE 位置位。此时 UART1 RBR FIFO 不会被覆盖，UART1 RSR 中的字符将丢失。	0
2	奇偶错误（PE）	0：奇偶错误状态未激活 1：奇偶错误状态激活 当接收字符的奇偶位处于错误状态时产生一个奇偶错误。U1LSR 读操作清零 PE 位。奇偶错误检测时间取决于的 bit0。奇偶错误与 UART1 RBR FIFO 中读出的字符相关。	0
3	帧错误（FE）	0：帧错误状态未激活 1：帧错误状态激活 当接收字符的停止位为 0 时，产生帧错误。U1LSR 读操作清零 FE 位。帧错误检测时间取决于 U1FCR 的 bit0。帧错误与 UART1 RBR FIFO 中读出的字符相关。当检测到一个帧错误时，Rx 将尝试与数据重新同步并假设错误的停止位实际是一个超前的起始位。	0

接上表

U1LSR	功能	描述	复位值
4	间隔中断（BI）	0：间隔中断状态未激活 1：间隔中断状态激活 在发送整个字符（起始位、数据、奇偶位和停止位）过程中 RxD1 如果都保持逻辑 0，则产生间隔中断。当检测到间隔条件时，接收器立即进入空闲状态直到 RxD1 变为全 1 状态。U1LSR 读操作清零该状态位。间隔检测的时间取决于 U1FCR 的 bit0。间隔中断与 UART1 RBR FIFO 中读出的字符相关。	0
5	发送保持寄存器空（THRE）	0：U1THR 包含有效数据 1：U1THR 空 当检测到 U1THR 空时，THRE 置位，U1THR 写操作清零该位。	1
6	发送器空（TEMT）	0：U1THR 和/或 U1TSR 包含有效数据 1：U1THR 和 U1TSR 空 当 THR 和 TSR 都为空时，TEMT 置位。当 U1TSR 或 U1THR 包含有效数据时，TEMT 清零。	1
7	Rx FIFO 错误（RXFE）	0：U1RBR 中没有 UART1 Rx 错误，或 U1FCR 的 bit0 为 0 1：U1RBR 包含至少一个 UART1 Rx 错误 当一个带有 Rx 错误（例如帧错误、奇偶错误或间隔中断）的字符装入 U1RBR 时，RXFE 位置位。当读取 U1LSR 寄存器并且 UART1 FIFO 中不再有错误时，RXFE 位清零。	0

UART1 Modem 状态寄存器（U1MSR - 0x0E0010018）

U1MSR 是一个只读寄存器，它提供 modem 输入信号的状态信息，见表 5.97。U1MSR[3:0]在读取 U1MSR 时清零。需要注意的是，modem 信号对 UART1 的操作没有直接影响，modem 信号的操作是通过软件来实现的。

表 5.97 UART1 Modem 状态寄存器

U1MSR	功能	描述	复位值
0	Delta CTS	0：没有检测到 modem 输入 CTS 上的状态变化 1：检测到 modem 输入 CTS 上的状态变化 当输入 CTS 状态发生变化时，该位置位。读取 U1MSR 时清零。	0
1	Delta DSR	0：没有检测到 modem 输入 DSR 上的状态变化 1：检测到 modem 输入 DSR 上的状态变化 当输入 DSR 状态发生变化时，该位置位。读取 U1MSR 时清零。	0
2	后沿 RI	0：没有检测到 modem 输入 RI 上的状态变化 1：检测到 modem 输入 RI 上的状态变化 当输入 RI 状态发生变化时，该位置位。读取 U1MSR 时清零。	0
3	Delta DCD	0：没有检测到 modem 输入 DCD 上的状态变化 1：检测到 modem 输入 DCD 上的状态变化 当输入 DCD 状态发生变化时，该位置位。读取 U1MSR 时清零。	0
4	CTS	清零以发送状态 输入信号 CTS 的补码。在回写模式下，该位连接到 U1MCR 的 bit1。	0

接上表

U1MSR	功能	描述	复位值
5	DSR	数据设置就绪状态 输入信号 DSR 的补码。在回写模式下，该位连接到 U1MCR 的 bit0。	0
6	RI	响铃指示状态 输入信号 RI 的补码。在回写模式下，该位连接到 U1MCR 的 bit2。	0
7	DCD	数据载波检测状态 输入信号 DCD 的补码。在回写模式下，该位连接到 U1MCR 的 bit3。	0

UART1 高速缓存寄存器（U1SCR - 0xE001001C）

在 UART1 操作时 U1SCR 无效。用户可自由对该寄存器进行读或写。不提供中断接口向主机指示 U1SCR 所发生的读或写操作。

U1SCR 寄存器描述见表 5.98。

表 5.98 高速缓存寄存器

U1SCR	功能	描述	复位值
7:0	-	一个可读可写的字节	0

5.12 I²C 接口

5.12.1 特性

- 标准的 I²C 总线接口
- 可配置为主机、从机或主/从机
- 可编程时钟可实现通用速率控制
- 主机从机之间双向数据传输
- 多主机总线(无中央主机)
- 同时发送的主机之间进行仲裁，避免了总线数据的冲突

5.12.2 应用

与外部标准 I²C 部件接口，例如串行 E²PROM、RAM、RTC、LCD、音调发生器等等。

5.12.3 引脚描述

I²C 引脚描述见表 5.99。

表 5.99 I²C 引脚描述

引脚名称	类型	描述
SDA	输入/输出	**串行数据** I²C 数据输入和输出。相关端口为开漏输出以符合 I²C 规范。
SCL	输入/输出	**串行时钟** I²C 时钟输入和输出。相关端口为开漏输出以符合 I²C 规范。

5.12.4 I²C 接口描述

1. I²C 总线简要描述

I²C 总线的典型应用电路原理如图 5.39 所示。根据方向位(R/W)状态的不同，I²C 总线上

存在以下两种类型的数据传输：

- 主发送器向从接收器发送数据，即主发送，数据传输方向如图 5.40 所示。主机发送的第一个字节是从机地址，接下来是数据字节流。从机每接收一个字节返回一个应答位。
- 从发送器向主接收器发送数据，即主接收，数据传输方向图 5.41 所示。第一个字节(从地址)由主机发送，然后从机返回一个应答位，接下来从机向主机发送数据字节。主机每接收一个字节返回一个应答位，接收完最后一个字节，主机返回一个“非应答位”。

当主机产生起始条件或重新起始条件，发送从机寻址字节(从机地址+读写操作位)之后，即开始一次串行数据发送/接收。当出现停止条件时，此次数据传输结束。

I^2C 数据传送速度：标准模式为 100Kbit/s；高速模式为 400Kbit/s。总线速率为 100Kbit/s，即是在数据传送时 SCL 上的时钟信号频率约为 100KHz。一般 I^2C 器件均可达到 100Kbit/s 的总线速率。

总线速率与总线上拉电阻的关系：总线速率越高，总线上拉电阻要越小。100Kbit/s 总线速率，通常使用 5.1KΩ的上拉电阻。

注意：无论是主发送还是主接收，均由主器件产生所有串行时钟脉冲和起始以及停止条件。

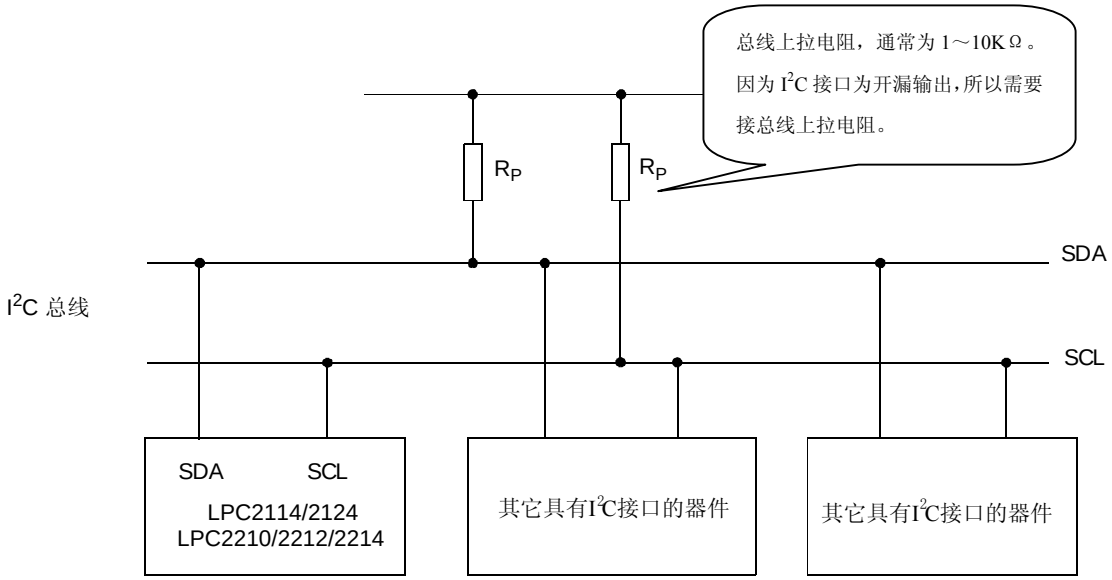


图 5.39 I^2C 总线典型应用电路原理

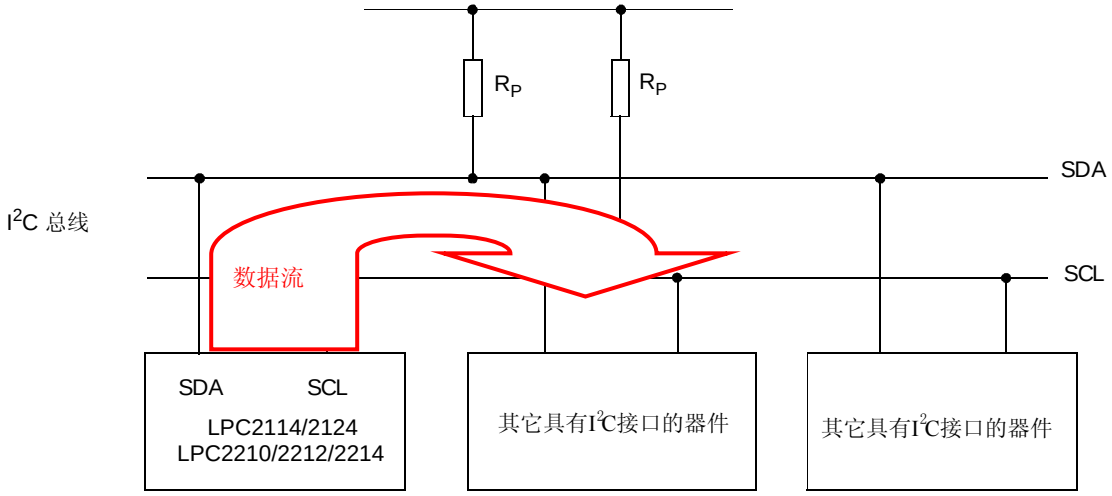


图 5.40 主发送时数据传输方向

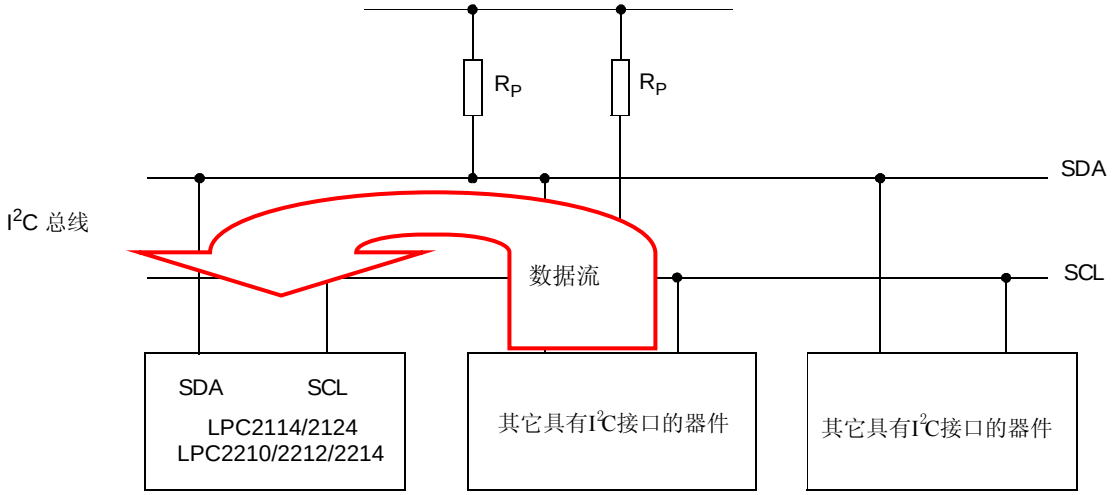


图 5.41 主接收时数据传输方向

2. LPC2000 的 I²C 接口简要描述

LPC2000 的 I²C 结构图见图 5.42。

LPC2000 是**字节方式的 I²C 接口**，简单的说就是把一个字节数据写入 I²C 数据寄存器 I2DAT 后，即可由 I²C 接口自动完成所有数据位的发送。补充说明，位方式的 I²C 接口需要用户程序控制每一位数据的发送/接收，比如 PHILIPS 公司的 LPC700 系列微控制器就是位方式的 I²C 接口。

该系列器件可以配置为 I²C 主机，也可以配置为 I²C 从机(比如，可以用该系列器件模拟一个 CAT24WC02)，所以具有 4 种操作模式：主发送模式、主接收模式、从发送模式和从接收模式。

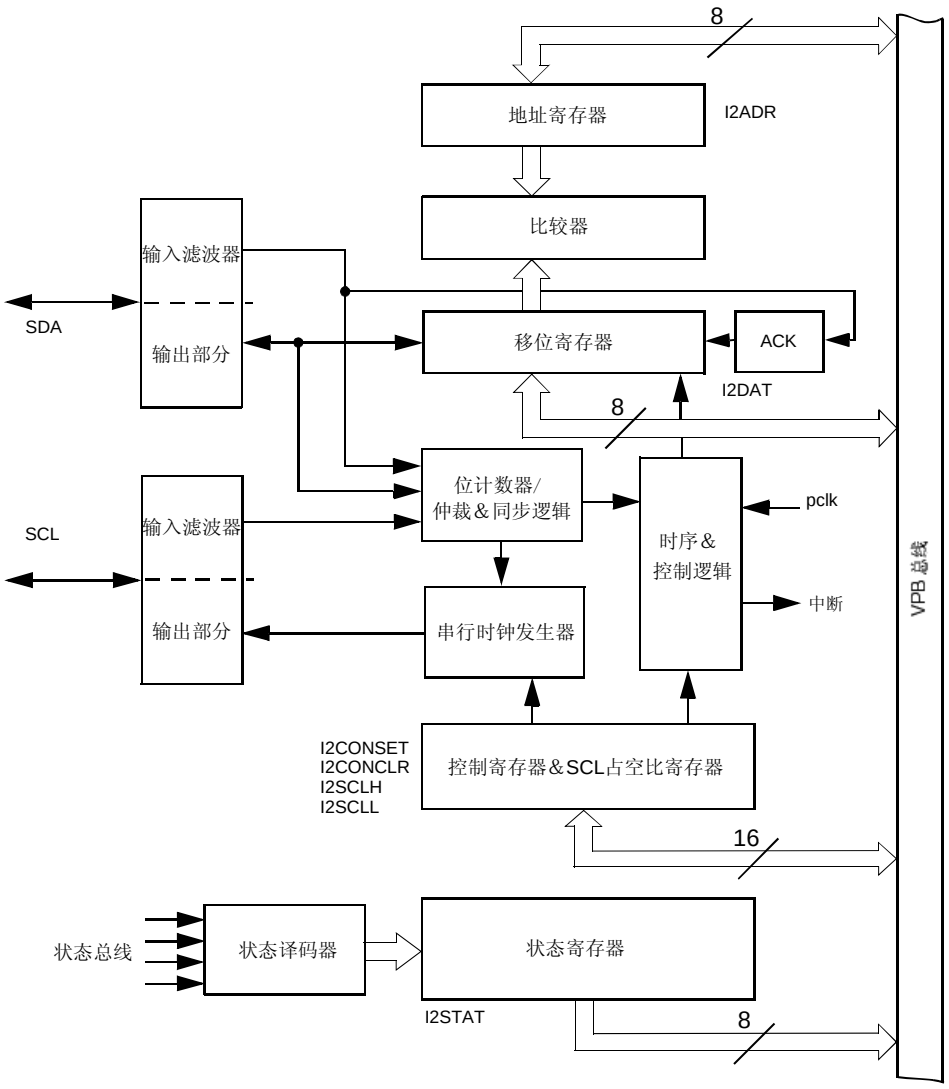


图 5.42 I^2C 结构

5.12.5 I^2C 操作模式

1. 主模式 I^2C

在该模式中，LPC2000 作为主控器，向从机发送数据(即是主发送模式)及接收从机的数据(即是主接收模式)。在进入主模式 I^2C，I2CONSET 必须按照图 5.43 进行初始化。**LPC2000 的 I^2C 寄存器的详细说明见第 5.12.6 小节。**

I2EN 置 1 操作是通过向 I2CONSET 写入 0x40 实现；AA，STA 和 SI 置 0 操作是通过向 I2CONCLR 写入 0x2C 实现；当总线产生了一个停止条件时，STO 位由硬件自动置 0。

	7	6	5	4	3	2	1	0
I2CONSET	—	I2EN	STA	STO	SI	AA	—	—
	—	1	0	0	0	0	—	—

图 5.43 主模式配置

说明：I2EN = 1，使能 I^2C 接口；

AA = 0，不产生应答信号，即不允许进入从机模式；

SI = 0，I^2C 中断标志为 0；
STO=0，起始标志为 0；
STA=0，停止标志为 0；

主模式 I^2C 的初始化

使用主模式 I^2C 时，先设置 I/O 口功能选择，然后设置总线的速率，再使能主 I^2C，即可开始发送/接收数据。主模式 I^2C 初始化示例如程序清单 5.25 所示。实际应用中，通常会使用中断方式进行 I^2C 的操作，所以初始化程序中加入了中断的初始化。

程序清单 5.25 主模式 I^2C 初始化示例

```
/*****************************************************************************
* 名称：I2C_Init()
* 功能：I2C 初始化，包括初始化其中断为向量 IRQ 中断。
* 入口参数：fi2c            初始化 I2C 总线速率，最大值为 400K
* 出口参数：无
*****************************************************************************/
void   I2C_Init(uint32 fi2c)
{   if(fi2c>400000) fi2c = 400000;

    PINSEL0 = (PINSEL0&0xFFFFFF0F) | 0x50;  // 设置 I2C 控制口有效

    I2SCLH = (Fpclk/fi2c + 1) / 2;          // 设置 I2C 时钟为 fi2c
    I2SCLL = (Fpclk/fi2c) / 2;
    I2CONCLR = 0x2C;
    I2CONSET = 0x40;                        // 使能主 I2C

    /* 设置 I2C 中断允许 */
    VICIntSelect = 0x00000000;              // 设置所有通道为 IRQ 中断
    VICVectCntl0 = 0x29;                    // I2C 通道分配到 IRQ slot 0，即优先级最高
    VICVectAddr0 = (int)IRQ_I2C;            // 设置 I2C 中断向量地址
    VICIntEnable = 0x0200;                  // 使能 I2C 中断
}
```

主模式 I^2C 的数据发送

主模式 I^2C 的数据发送格式见图 5.44，起始和停止条件用于指示串行传输的起始和结束。第一个发送的数据包含接收器件的从地址（7 位）和读写操作位。在此模式下，读写操作位（R/W）应该为 0，表示执行写操作。数据的发送每次为 8 位，即一字节，每发送完一个字节，主机都接收到一个应答位(是由从机回发的)。

主模式 I^2C 的数据发送波形图如图 5.45 所示。

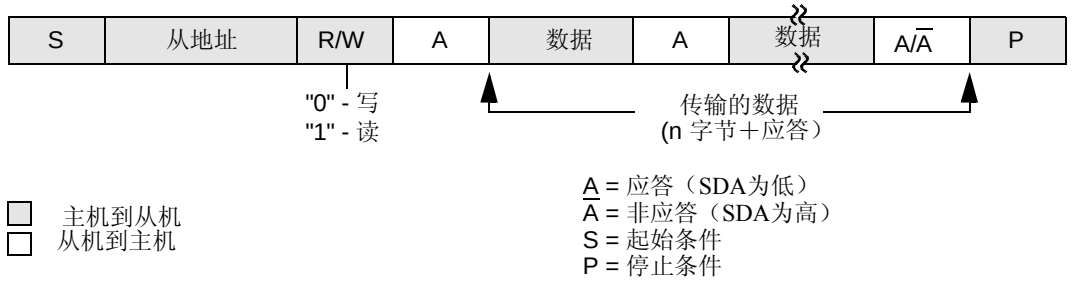


图 5.44 主发送模式中的格式

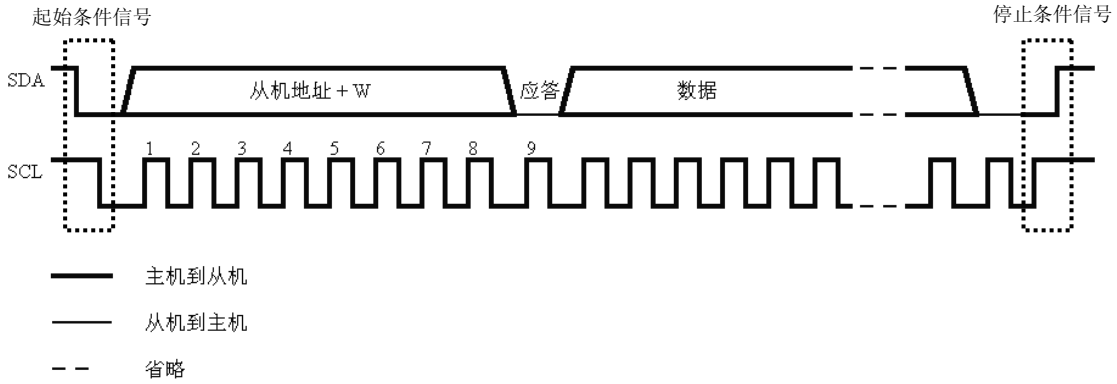


图 5.45 主模式 I^2C 的数据发送波形

主模式 I^2C 的数据发送操作步骤如下：

- 通过软件置位 STA 进入 I^2C 主发送模式，I^2C 逻辑在总线空闲后立即发送一个起始条件。
- 当发送完起始条件后，SI 会置位，此时 I2STAT 中的状态代码为 08H。该状态代码用于中断服务程序的处理。
- 把从地址和读写操作位装入 I2DAT（数据寄存器），然后清零 SI 位，开始发送从地址和 W 位。
- 当从地址和 W 位已发送且接收到应答位之后，SI 位再次置位，可能的状态代码为 18H，20H 或 38H。每个状态代码及其对应的执行动作见表 5.100。
- 若状态码为 18H，表明从机已应答，则可以将数据装入 I2DAT，然后清零 SI 位，开始发送数据。
- 当正确发送数据，SI 位再次置位，可能的状态代码为 28H 或 30H，此时可以再次发送数据，或者置位 STO 结束总线。每个状态代码及其对应的执行动作见表 5.100。

表 5.100 主发送模式状态

状态代码 (I2STAT)	I^2C 总线硬件状态	应用软件的响应					I^2C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
08H	已发送起始条件	装入 SLA+W	x	0	0	x	将发送 SLA+W，接收 ACK 位
10H	已发送重复起始条件	装入 SLA+W	x	0	0	x	同上
		装入 SLA+R	x	0	0	x	将发送 SLA+W，I^2C 将切换到主接收模式
18H	已发送 SLA+W；已接收 ACK	装入数据字节	0	0	0	x	将发送数据字节，接收 ACK 位
		无 I2DAT 动作	1	0	0	x	将发送重复起始条件
		无 I2DAT 动作	0	1	0	x	将发送停止条件；STO 标志将复位
		无 I2DAT 动作	1	1	0	x	将发送停止条件，然后发送起始条件；STO 标志将复位
20H	已发送 SLA+W；已接收非 ACK	装入数据字节	0	0	0	x	将发送数据字节，接收 ACK 位
		无 I2DAT 动作	1	0	0	x	将发送重复起始条件
		无 I2DAT 动作	0	1	0	x	将发送停止条件；STO 标志将复位
		无 I2DAT 动作	1	1	0	x	将发送停止条件，然后发送起始条件；STO 标志将复位
28H	已发送 I2DAT 中的数据字节；已接收 ACK	装入数据字节	0	0	0	x	将发送数据字节，接收 ACK 位
		无 I2DAT 动作	1	0	0	x	将发送重复起始条件
		无 I2DAT 动作	0	1	0	x	将发送停止条件；STO 标志将复位
		无 I2DAT 动作	1	1	0	x	将发送停止条件，然后发送起始条件；STO 标志将复位
30H	已发送 I2DAT 中的数据字节；已接收非 ACK	装入数据字节	0	0	0	x	将发送数据字节，接收 ACK 位
		无 I2DAT 动作	1	0	0	x	将发送重复起始条件
		无 I2DAT 动作	0	1	0	x	将发送停止条件；STO 标志将复位
		无 I2DAT 动作	1	1	0	x	将发送停止条件，然后发送起始条件；STO 标志将复位
38H	在 SLA+R/W 或数据字节中丢失仲裁	无 I2DAT 动作	0	0	0	x	I^2C 总线将被释放；进入不可寻址从模式
		无 I2DAT 动作	1	0	0	x	当总线变为空闲时发送起始条件

主模式 I^2C 的数据发送(中断方式)程序原理示意图如图 5.46 所示。

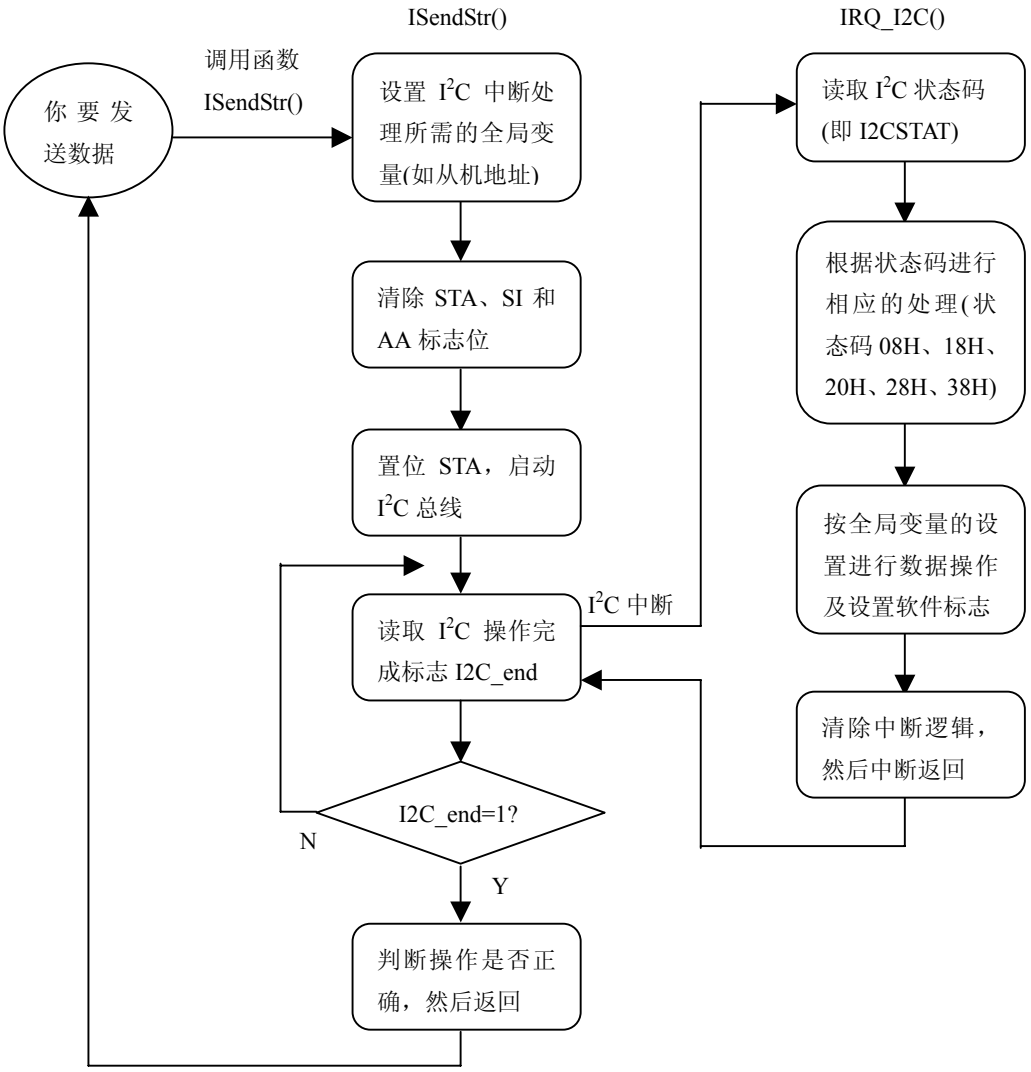


图 5.46 主模式 I²C 的数据发送程序原理示意图

主模式 I²C 的数据接收

在主接收模式中，主机所接收的数据字节来自从发送器(即从机) ，主模式 I²C 的数据接收格式见图 5.47。起始和停止条件用于指示串行传输的起始和结束。第一个发送的数据包含接收器件的从地址（7 位）和读写操作位。在此模式下，读写操作位（R/W）应该为 1，表示执行读操作。

主模式 I²C 的数据接收波形图如图 5.48 所示。

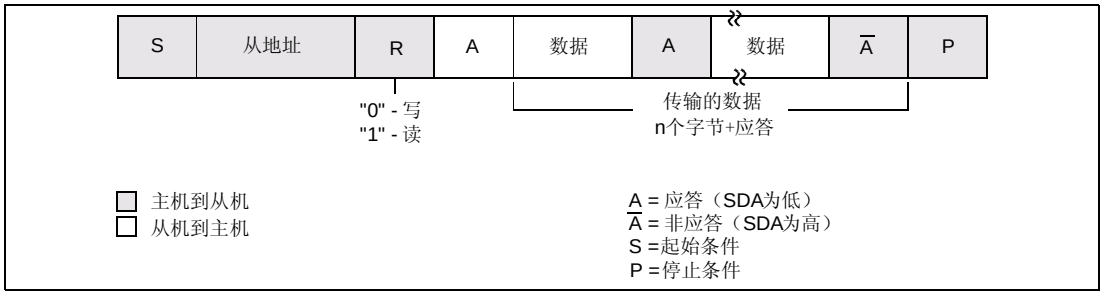


图 5.47 主接收模式的格式

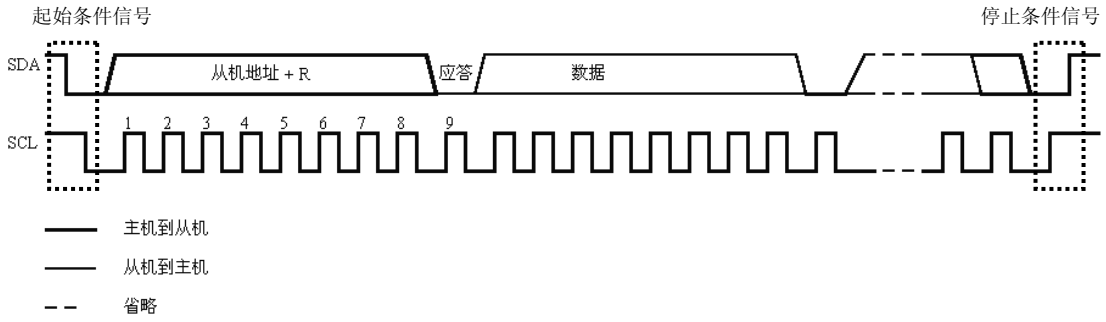


图 5.48 主模式 I²C 的数据接收波形

主模式 I²C 的数据发送操作步骤如下：

- 通过软件置位 STA 进入 I²C 主发送模式，I²C 逻辑在总线空闲后立即发送一个起始条件。
- 当发送完起始条件后，SI 会置位，此时 I2STAT 中的状态代码为 08H。该状态代码用于中断服务程序的处理。
- 把从地址和读写操作位装入 I2DAT（数据寄存器），然后清零 SI 位，开始发送从地址和 R 位。
- 当从地址和 R 位已发送且接收到应答位之后，SI 位再次置位，可能的状态代码为 38H，40H 或 48H。每个状态代码及其对应的执行动作见表 5.101。
- 若状态码为 40H，表明从机已应答。设置 AA 位，用来控制接收到数据后是产生应答信号，还是产生非应答信号，然后清零 SI 位，开始接收数据。
- 当正确接收到一字节数据后，SI 位再次置位，可能的状态代码为 50H 或 58H，此时可以再次接收数据，或者置位 STO 结束总线。每个状态代码及其对应的执行动作见表 5.101。

表 5.101 主接收模式状态

状态代码 (I2STAT)	I²C 总线硬件状态	应用软件的响应					I²C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
08H	已发送起始条件	装入 SLA+R	x	0	0	x	将发送 SLA+R，接收 ACK 位
10H	已发送重复起始条件	装入 SLA+R	x	0	0	x	同上
		装入 SLA+W	x	0	0	x	将发送 SLA+W，I²C 将切换到主发送模式
38H	在发送 SLA+R 时丢失仲裁	无 I2DAT 动作	0	0	0	x	I²C 总线将被释放；I²C 将进入从模式
		无 I2DAT 动作	1	0	0	x	当总线恢复空闲后发送起始条件
40H	已发送 SLA+R；已接收 ACK	无 I2DAT 动作	0	0	0	0	将接收数据字节；返回非 ACK 位
		无 I2DAT 动作	0	0	0	1	将接收数据字节；返回 ACK 位
48H	已发送 SLA+R；已接收非 ACK	无 I2DAT 动作	1	0	0	x	将发送重复起始条件
		无 I2DAT 动作	0	1	0	x	将发送停止条件；STO 标志将复位
		无 I2DAT 动作	1	1	0	x	将发送停止条件，然后发送起始条件；STO 标志将复位
50H	已接收数据字节；已返回 ACK	读数据字节	0	0	0	0	将接收数据字节，返回非 ACK 位
		读数据字节	0	0	0	1	将接收数据字节；返回 ACK 位

接上表

状态代码 (I2STAT)	I^2C 总线硬件状态	应用软件的响应					I^2C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
58H	已接收数据字节；已返回非 ACK	读数据字节	1	0	0	x	将发送重复起始条件
		读数据字节	0	1	0	x	将发送停止条件；STO 标志将复位
		读数据字节	1	1	0	x	将发送停止条件，然后发送起始条件；STO 标志将复位

主模式 I^2C 的数据接收(中断方式)程序原理示意图如图 5.49 所示。

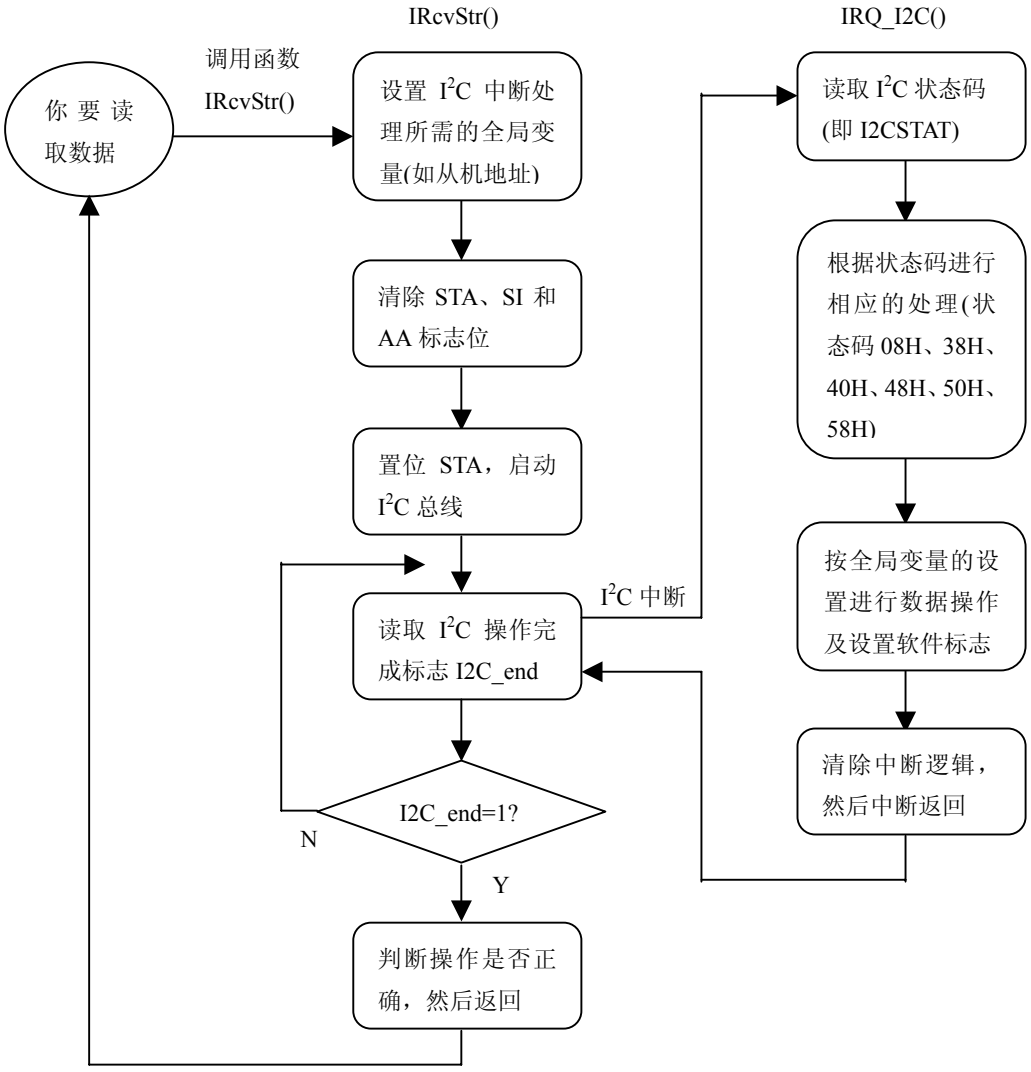


图 5.49 主模式 I^2C 的数据接收程序原理示意图

2. 从模式 I^2C

LPC2000 系列器件配置为 I^2C 从机时，I^2C 主机可以对它进行读/写操作，此时从机处于从发送/接收模式。要初始化从接收模式，用户必须将从地址写入从地址寄存器（I2ADR）并按照图 5.50 配置 I^2C 控制置位寄存器（I2CONSET）。I2CONSET 寄存器的详细说明见第 5.12.6 小节。

I2EN 和 AA 置 1 操作是通过向 I2CONSET 写入 0x44 实现；STA 和 SI 置 0 操作是通过向 I2CONCLR 写入 0x28 实现；当总线产生了一个停止条件时，STO 位由硬件自动置 0。

	7	6	5	4	3	2	1	0
I2CONSET	—	I2EN	STA	STO	SI	AA	—	—
	—	1	0	0	0	1	—	—

图 5.50 从模式配置

说明：I2EN = 1，使能 I^2C 接口；

AA = 1，应答主机对本从机地址的访问；

SI = 0，I^2C 中断标志为 0；

STO=0，起始标志为 0；

STA=0，停止标志为 0；

从模式 I^2C 的初始化

使用从模式 I^2C 时，先设置 I/O 口功能选择，再设置从机地址，然后使能 I^2C(配置为从模式)，即可等待主机访问。从模式 I^2C 初始化示例如程序清单 5.26 所示。实际应用中，通常是使用中断方式进行 I^2C 的操作，所以初始化程序中加入了中断的初始化。

因为 I^2C 总线时钟信号是由主机产生，所以从机不用初始化 I2SCLH 和 I2SCLL 寄存器。

程序清单 5.26 从模式 I^2C 初始化示例

```
/*****************************************************************************
* 名称：I2C_SlaveInit()
* 功能：从模式 I2C 初始化，包括初始化其中断为向量 IRQ 中断。
* 入口参数：adr              本从机地址
* 出口参数：无
*****************************************************************************/
void   I2C_SlavInit(uint8 adr)
{   PINSEL0 = (PINSEL0&0xFFFFFF0F) | 0x50;   // 设置 I2C 控制口有效

    I2ADR = adr&0xFE;                  // 设置从机地址
    I2CONCLR = 0x28;
    I2CONSET = 0x44;                   // I2C 配置为从机模式

    /* 设置 I2C 中断允许 */
    VICIntSelect = 0x00000000;         // 设置所有通道为 IRQ 中断
    VICVectCntl0 = 0x29;               // I2C 通道分配到 IRQ slot 0，即优先级最高
    VICVectAddr0 = (int)IRQ_I2C;       // 设置 I2C 中断向量地址
    VICIntEnable = 0x0200;             // 使能 I2C 中断
}
```

从模式 I^2C 的数据接收

当主机访问从机时，若读写操作位为 0（W），则从机进入从接收模式，接收主机发送过来的数据，并产生应答信号。从模式 I^2C 的数据接收格式见图 5.51，从接收模式中，总线

时钟、起始条件、从机地址、停止条件仍由主机产生。

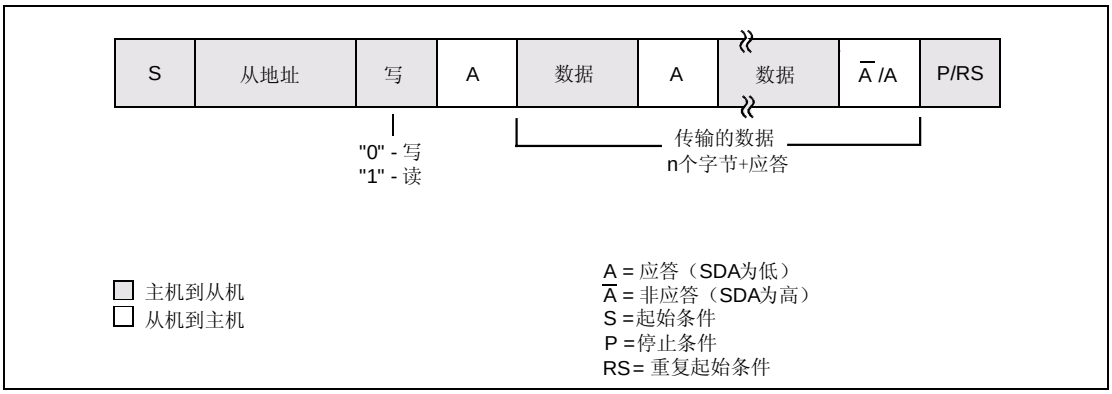


图 5.51 从接收模式的格式

使用从模式 I^2C 时，用户程序只需要在 I^2C 中断服务程序完成各种数据操作，即是根据各种状态码作出相应的操作。从接收模式的每个状态代码及其对应的执行动作见表 5.102。

表 5.102 从接收模式状态

状态代码 (I2STAT)	I^2C 总线硬件状态	应用软件的响应					I^2C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
60H	已接收自身 SLA+W; 已返回 ACK	无 I2DAT 动作	x	0	0	0	将接收数据字节并返回非 ACK 位
		无 I2DAT 动作	x	0	0	1	将接收数据字节并返回 ACK 位
68H	主控器时在 SLA+W 中丢失仲裁;已接收自身 SLA+W, 已返回 ACK	无 I2DAT 动作	x	0	0	0	将接收数据字节并返回非 ACK 位
		无 I2DAT 动作	x	0	0	1	将接收数据字节并返回 ACK 位
70H	已接收通用调用地址 (00H); 已返回 ACK	无 I2DAT 动作	x	0	0	0	将接收数据字节并返回非 ACK 位
		无 I2DAT 动作	x	0	0	1	将接收数据字节并返回 ACK 位
78H	主控器时在 SLA+R/W 中丢失仲裁; 已接收通用调用地址;已返回 ACK	无 I2DAT 动作	x	0	0	0	将接收数据字节并返回非 ACK 位
		无 I2DAT 动作	x	0	0	1	将接收数据字节并返回 ACK 位
80H	前一次寻址使用自身从地址; 已接收数据字节; 已返回 ACK	读数据字节	x	0	0	0	将接收数据字节并返回非 ACK 位
		读数据字节	x	0	0	1	将接收数据字节并返回 ACK 位

接上表

状态代码(I2STAT)	I^2C 总线硬件状态	应用软件的响应					I^2C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
88H	前一次寻址使用自身从地址；已接收数据字节；已返回非 ACK	读数据字节	0	0	0	0	切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址
		读数据字节	0	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址
		读数据字节	1	0	0	0	切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址；当总线空闲后发送起始条件
		读数据字节	1	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址；当总线空闲后发送起始条件
90H	前一次寻址使用通用调用；已接收数据字节；已返回 ACK	读数据字节	x	0	0	0	将接收数据字节并返回非 ACK 位
		读数据字节	x	0	0	1	将接收数据字节并返回 ACK 位
98H	前一次寻址使用通用调用；已接收数据字节；已返回非 ACK	读数据字节	0	0	0	0	切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址
		读数据字节或	0	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址
		读数据字节或	1	0	0	0	切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址；当总线空闲后发送起始条件
		读数据字节	1	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址；当总线空闲后发送起始条件
A0H	当使用 SLV/REC 或 SLV/TRX 静态寻址时,接收到停止条件或重复的起始条件	无 I2DAT 动作或	0	0	0	0	切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址
		无 I2DAT 动作或	0	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址
		无 I2DAT 动作或	1	0	0	0	切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址；当总线空闲后发送起始条件
		无 I2DAT 动作	1	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址；当总线空闲后发送起始条件

从模式 I^2C 的数据发送

当主机访问从机时，若读写操作位为 1（R），则从机进入从发送模式，向主机发送数据，并等待主机的应答信号。从模式 I^2C 的数据接收格式见图 5.52，从发送模式中，总线时钟、起始条件、从机地址、停止条件仍由主机产生。

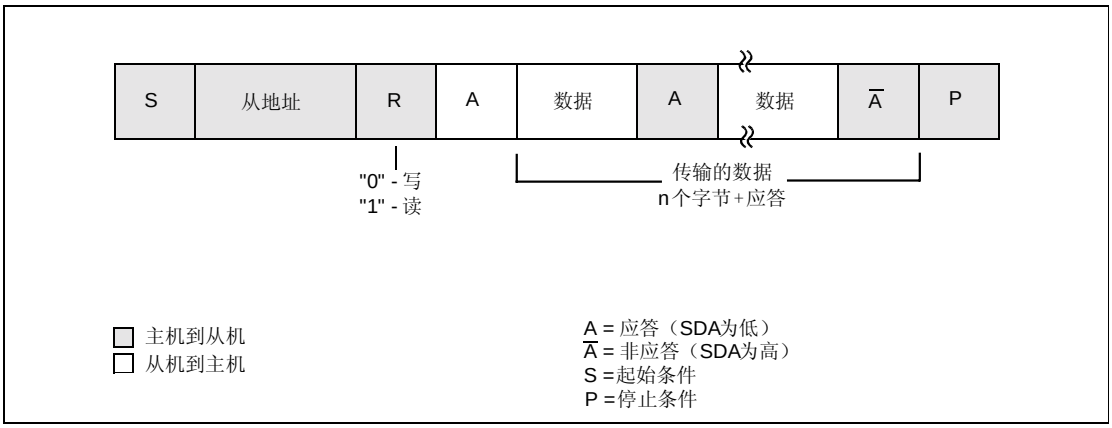


图 5.52 从发送模式的格式

使用从模式 I^2C 时，用户程序只需要在 I^2C 中断服务程序完成各种数据操作，即是根据各种状态码作出相应的操作。从发送模式的每个状态代码及其对应的执行动作见表 5.103。

表 5.103 从发送模式状态

状态代码 (I2STAT)	I^2C 总线硬件状态	应用软件的响应					I^2C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
A8H	已接收自身 SLA+R；已返回 ACK	装入数据字节或	x	0	0	0	将发送最后的数据字节并接收 ACK 位
		装入数据字节	x	0	0	1	将发送数据字节并接收 ACK 位
B0H	主控器时在 SLA+R/W 中丢失仲裁；已接收自身 SLA+R，已返回 ACK	装入数据字节或	x	0	0	0	将发送最后的数据字节并接收 ACK 位
		装入数据字节	x	0	0	1	将发送数据字节并接收 ACK 位
B8H	已发送 I2DAT 中数据字节；已返回 ACK	装入数据字节或	x	0	0	0	将发送最后的数据字节并接收 ACK 位
		装入数据字节	x	0	0	1	将发送数据字节并接收 ACK 位
C0H	已发送 I2DAT 中数据字节；已返回非 ACK	无 I2DAT 动作或	0	0	0	0	切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址
		无 I2DAT 动作或	0	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址
		无 I2DAT 动作或	1	0	0	0	切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址；当总线空闲后发送起始条件
		无 I2DAT 动作	1	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址；当总线空闲后发送起始条件

接上表

状态代码 (I2STAT)	I^2C 总线硬件状态	应用软件的响应					I^2C 硬件执行的下一个动作
		读/写 I2DAT	写 I2CON				
			STA	STO	SI	AA	
C8H	已发送 I2DAT 中最后的数据字节(AA=0)；已返回 ACK	无 I2DAT 动作或	0	0	0	0	切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址
		无 I2DAT 动作或	0	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址
		无 I2DAT 动作或	1	0	0	0	切换到不可寻址 SLV 模式；不识别自身 SLA 或通用调用地址；当总线空闲后发送起始条件
		无 I2DAT 动作	1	0	0	1	切换到不可寻址 SLV 模式；识别自身 SLA；如果 S1ADR.0=1，将识别通用调用地址；当总线空闲后发送起始条件
F8H	无可用相关信息；SI=0	无 I2DAT 动作	无 I2DAT 动作				等待或进行当前的传输
00H	在 MST 或选择的从模式中，由于非法的起始或停止条件，使总线发生错误。当干扰导致 I^2C 进入一个未定义的状态时，也可产生状态 00H	无 I2DAT 动作	0	1	0	x	在 MST 或寻址 SLV 模式中只有内部硬件受影响。在所有情况下，总线被释放，而 I^2C 切换到不可寻址 SLV 模式。STO 复位。

5.12.6 寄存器描述

I^2C 接口包含 7 个寄存器，如表 5.104 所示。

表 5.104 I^2C 寄存器汇总

名称	描述	访问	复位值*	地址
I2CONSET	I^2C 控制置位寄存器	读/置位	0	0xE001C000
I2STAT	I^2C 状态寄存器	只读	0xF8	0xE001C004
I2DAT	I^2C 数据寄存器	读/写	0	0xE001C008
I2ADR	I^2C 从地址寄存器	读/写	0	0xE001C00C
I2SCLH	SCL 占空比寄存器高半字	读/写	0x04	0xE001C010
I2SCLL	SCL 占空比寄存器低半字	读/写	0x04	0xE001C014
I2CONCLR	I^2C 控制清零寄存器	只清零	NA	0xE001C018

* 复位值仅指已使用位中保存的数据，不包括保留位的内容。

I^2C 控制置位寄存器（I2CONSET-0xE001C000）

I2CONSET 寄存器的描述见表 5.105。对此寄存器的某个位写入 1，相应位即被设置为 1；**但需要注意，对 I2CONSET 寄存器的某个位写入 0，相应位并不能被设置为 0，置 0 操作只能通过 I2CONCLR 寄存器实现。**

表 5.105 I^2C 控制置位寄存器 I2CONSET

I2CONSET	功能	描述	复位值
0	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
1	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

接上表

I2CONSET	功能	描述	复位值
2	AA	应答标志	0
3	SI	I^2C 中断标志	0
4	STO	停止标志	0
5	STA	起始标志	0
6	I2EN	I^2C 接口使能	0
7	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

AA 为声明应答标志。当该位置位时，在 SCL 线的应答时钟脉冲内，出现下面的任意条件之一将产生一个应答信号（SDA 线为低电平）：

- 接收到从地址寄存器中的地址。
- 当 I2ADR 中的通用调用位（GC）置位时，接收到通用调用地址。
- 当 I^2C 接口处于主接收模式时，接收到一个数据字节。
- 当 I^2C 接口处于可寻址的从接收模式时，接收到一个数据字节。

向 I2CONCLR 寄存器中的 AAC 位写入 1 会使 AA 位清零。当 AA 为零时，在 SCL 线的应答时钟脉冲内，出现下列情况将返回一个非应答信号（SDA 线为高电平）：

- 当 I^2C 接口处于主接收模式时，接收到一个数据字节。
- 当 I^2C 接口处于可寻址的从接收模式时，接收到一个数据字节。

SI 为 I^2C 中断标志。当进入 25 种可能的 I^2C 状态中的任何一个后，该位置位。通常，I^2C 中断只在空闲的从器件中用于指示一个起始条件，或在一个空闲的主器件（如果它等待使用 I^2C 总线）中指示一个停止条件。向 I2CONCLR 寄存器中的 SIC 位写入 1 使 SI 位清零。

STO 为停止标志。当 STO 为 1 时，在主模式中，向 I^2C 总线发送一个停止条件或在从模式中使总线从错误状态中恢复。当主模式中 STO=1 时，向总线发送停止条件。当总线检测到停止条件时，STO 自动清零。

在从模式中，置位 STO 位可从错误状态中恢复。这种情况下不向总线发送停止条件。硬件的表现就好像是接收到一个停止条件并切换到不可寻址的从接收模式。STO 标志由硬件自动清零。

STA 为起始标志。当 STA=1 时，I^2C 接口进入主模式并发送一个起始条件，如果已经处于主模式，则发送一个重复起始条件。

当 STA=1 并且 I^2C 接口还没进入主模式时，I^2C 接口将进入主模式，检测总线并在总线空闲时产生一个起始条件。如果总线忙，则等待一个停止条件（释放总线）并在延迟半个内部时钟发生器周期后发送一个起始条件。当 I^2C 接口已经处于主模式中并发送或接收了数据时，I^2C 接口会发送一个重复的起始条件。STA 可在任何时候置位，当 I^2C 接口处于可寻址的从模式时，STA 也可以置位。

向 I2CONCLR 寄存器中的 STAC 位写入 1 使 STA 位清零。当 STA=0 时，不会产生起始或重复起始条件。

当 STA 和 STO 都置位时，如果 I^2C 接口处于主模式，I^2C 接口将向总线发送一个停止条件，然后发送一个起始条件。如果 I^2C 接口处于从模式，则产生一个内部停止条件，但不发送到总线上。

I2EN 为 I^2C 接口使能。当该位置位时，使能 I^2C 接口。向 I2CONCLR 寄存器中的 I2ENC 位写入 1 将使 I2EN 位清零。当 I2EN 位为 0 时，I^2C 功能被禁止。

I^2C 控制清零寄存器（I2CONCLR – 0xE001C018）

I2CONCLR 寄存器描述见表 5.106。

表 5.106 I^2C 控制清零寄存器（I2CONCLR - 0xE001C018）

I2CONCLR	功能	描述	复位值
0	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
1	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
2	AAC	应答标志清零位。向该位写入 1 清零 I2CONSET 寄存器中的 AA 位。写入 0 无效。	NA
3	SIC	I^2C 中断标志清零位。向该位写入 1 清零 I2CONSET 寄存器中的 SI 位。写入 0 无效。	NA
4	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
5	STAC	起始标志清零位。向该位写入 1 清零 I2CONSET 寄存器中的 STA 位。写入 0 无效。	NA
6	I2ENC	I^2C 接口禁止。向该位写入 1 清零 I2CONSET 寄存器中的 I2EN 位。写入 0 无效。	NA
7	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

I^2C 状态寄存器（I2STAT - 0xE001C004）

这是一个只读寄存器，它包含 I^2C 接口的状态代码，见表 5.107。最低 3 位总是为 0。一共有 26 种可能存在的状态代码。当代码为 F8H 时，无可用的相关信息，SI 位不会置位。所有其它 25 种状态代码都对应一个已定义的 I^2C 状态。当进入其中一种状态时，SI 位将置位。所有状态代码的描述表 5.100、表 5.101、表 5.102 和表 5.103。

表 5.107 I^2C 状态寄存器 I2STAT

I2STAT	功能	描述	复位值
2:0	状态	这 3 个位总是为 0	0
7:3	状态	状态位	1

I^2C 数据寄存器（I2DAT - 0xE001C008）

该寄存器包含要发送或刚接收的数据，见表 5.108。当它没有处理字节的移位时，CPU 可对其进行读写。该寄存器只能在 SI 置位时访问。在 SI 置位期间，I2DAT 中的数据保持稳定。I2DAT 中的数据移位总是从右至左进行：第一个发送的位是 MSB（位 7），在接收字节时，第一个接收到的位存放在 I2DAT 的 MSB。

表 5.108 I^2C 数据寄存器 I2DAT

I2DAT	功能	描述	复位值
7:0	数据	发送/接收数据位	0

I^2C 从地址寄存器（I2ADR - 0xE001C00C）

该寄存器可读可写，但只能在 I^2C 设置为从模式时才能使用，见

表 5.109。在主模式中，该寄存器无效。I2ADR 的 LSB 为通用调用位。当该位置位时，通用调用地址（00h）被识别。

表 5.109 I^2C 从地址寄存器 I2ADR

I2ADR	功能	描述	复位值
0	GC	通用调用位	0
7:1	地址	从模式地址	0

I^2C SCL 占空比寄存器（I2SCLH - 0xE001C010 和 I2SCLL - 0xE001C014）

软件必须通过对 I2SCLH（表 5.110）和 I2SCLL（表 5.111）寄存器进行设置来选择合适的波特率。I2SCLH 定义 SCL 高电平所保持的 pclk 周期数，I2SCLL 定义 SCL 低电平的 pclk 周期数。位频率(即总线速率)由下面的公式得出：

$$\text{位频率} = F_{pclk} / (I2SCLH + I2SCLL)$$

I2SCLL 和 I2SCLH 的值不一定要相同。通过设定这两个寄存器可得到 SCL 的不同占空比。但寄存器的值必须确保 I^2C 数据通信速率在 0 到 400KHz 之间。这样对 I2SCLL 和 I2SCLH 的值就有一些限制。**I2SCLL 和 I2SCLH 寄存器的值都必须大于等于 4**。

表 5.110 I^2C SCL 高电平占空比寄存器 I2SCLH

I2SCLH	功能	描述	复位值
15:0	计数值	SCL 高电平周期选择计数	0x0004

表 5.111 I^2C SCL 低电平占空比寄存器 I2SCLL

I2SCLL	功能	描述	复位值
15:0	计数值	SCL 低电平周期选择计数	0x0004

5.13 SPI 接口

5.13.1 特性

- 具有两个完全独立的 SPI 控制器
- 遵循同步串行接口(SPI)规范
- 全双工数据通信
- 可配置为 SPI 主机或从机
- **最大数据位速率为外设时钟 Fpclk 的 1/8**

5.13.2 引脚描述

SPI 引脚描述见表 5.112。

表 5.112 SPI 引脚描述

引脚名称	类型	描述
SCK1,SCK0	输入/输出	**串行时钟** 用于同步 SPI 接口间数据传输的时钟信号。该时钟信号总是由主机输出。时钟可编程为高有效或低有效。它只在数据传输时才被激活，其它任何时候都处于非激活状态或三态。

接上表

引脚名称	类型	描述
SSEL1,SSEL0	输入	**从机选择** SPI 从机选择信号是一个低有效信号，用于指示被选择参与数据传输的从机。每个从机都有各自特定的从机选择输入信号。在数据处理之前，SSEL 必须为低电平并在整个处理过程中保持低电平。如果在数据传输中 SSEL 信号变为高电平，传输将被中止。这种情况下，从机返回到空闲状态并将任何接收到的数据丢弃。对于这样的异常没有其它的指示。在主 SPI 模式下，该信号不能用作 GPIO。 **注：配置为 SPI 主机的 LPC2114/2124/2210/2212/2214 必须选择相应的引脚用作 SSEL 功能并使其保持高电平，只有这样，器件才能真正执行主机的功能。**
MISO1,MISO0	输入/输出	**主入从出** MISO 信号是一个单向的信号，它将数据由从机传输到主机。当器件为从机时，串行数据从该端口输出。当器件为主机时，串行数据从该端口输入。当从机没有被选择时，将该信号输出为高阻态。
MOSI1,MOSI0	输入/输出	**主出从入** MOSI 信号是一个单向的信号，它将数据从主机传输到从机。当器件为主机时，串行数据从该端口输出。当器件为从机时，串行数据从该端口输入。

5.13.3 描述

1. SPI 总线概述

SPI0 和 SPI1 是一个全双工的同步串行接口，如图 5.53 所示为基于 LPC2000 主机的 SPI 总线配置。一个 SPI 总线可以连接多个主机和多个从机，但是在同一时刻只允许有一个主机操作总线。在数据传输过程中，总线上只能有一个主机和一个从机通信。在一次数据传输中，主机总是向从机发送一个字节数据(主机通过 MOSI 输出数据)，而从机也总是向主机发送一个字节数据(主机通过 MISO 接收数据)。

SPI 总线时钟总是由主机产生的。

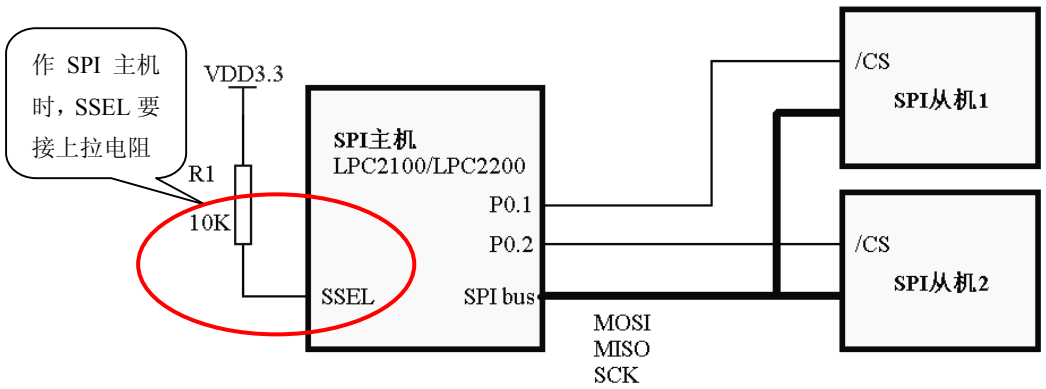


图 5.53 SPI 总线配置

2. SPI 数据传输

图 5.54 所示为 SPI 的 4 种不同数据传输格式的时序，该时序图描述的是 8 位数据的传输。需要注意的是，该时序图按水平方向可分成了 3 个部分，第一部分描述 SCK 和 SSEL 信号；第二部分描述了 CPHA=0 时的 MOSI 和 MISO 信号；第三部分描述了 CPHA＝1 时的

MOSI 和 MISO 信号。

在时序图的第一部分需要注意以下两点：

- 时序图包含了 CPOL 设置为 0 和 1 的情况。
- SSEL 信号的激活和未激活，SPI 从机时用作器件的片选信号。

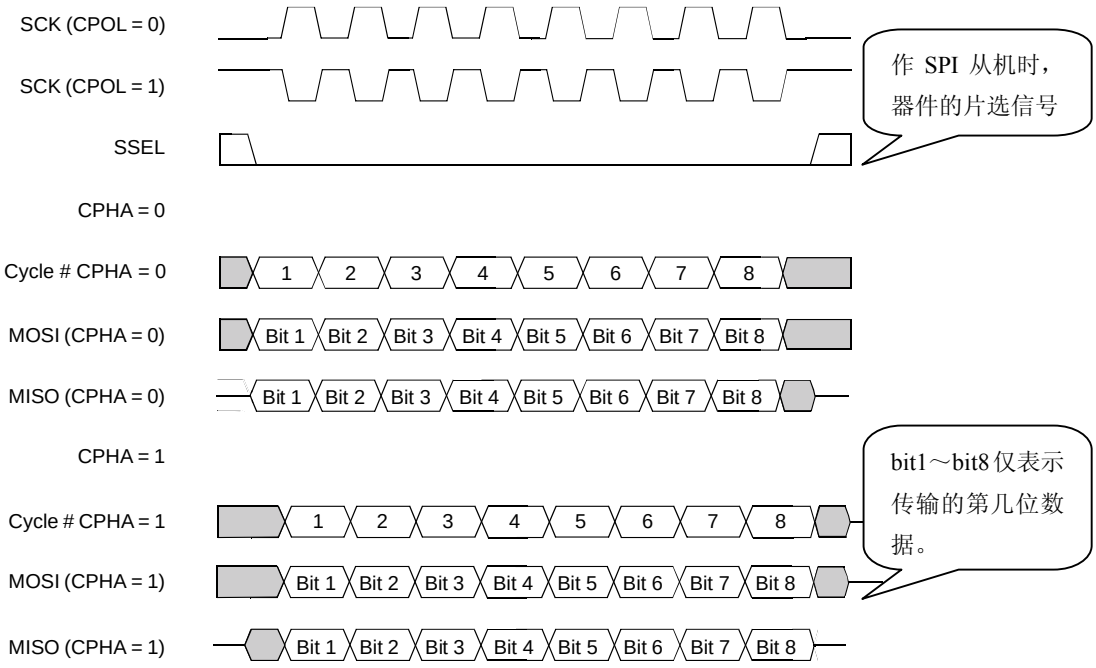


图 5.54 SPI 数据传输格式（CPHA=0 和 CPHA=1）

数据和时钟的相位关系在表 5.113 中描述，该表汇集了 CPOL 和 CPHA 的每一种设定。其中“第一个数据的输出”和“其它位数据的输出”栏是表示数据在什么时刻更新输出，这是由硬件 SPI 接口自动操作，用户一般不需理会。**用户需要注意的是“数据的采样”这一栏，这代表数据是 SCK 上升沿有效，还是下降沿有效。**

细心的读者会发现，在表 5.113 中的“数据的采样”栏中“SCK 上升沿”和“SCK 下降沿”都分别有两种设置，到底使用哪一种设置呢？**这就取决于在总线空闲时需要 SCK 信号为高电平还是低电平，CPOL 为 0 时表示为低电平，CPOL 为 1 时表示为高电平。**

表 5.113 SPI 数据和时钟的相位关系

CPOL 和 CPHA 的设定	第一位数据的输出	其它位数据的输出	数据的采样
CPOL=0, CPHA=0	在第一个 SCK 上升沿之前	SCK 下降沿	**SCK 上升沿**
CPOL=0, CPHA=1	第一个 SCK 上升沿	SCK 上升沿	**SCK 下降沿**
CPOL=1, CPHA=0	在第一个 SCK 下降沿之前	SCK 上升沿	**SCK 下降沿**
CPOL=1, CPHA=1	第一个 SCK 下降沿	SCK 下降沿	**SCK 上升沿**

当器件为主机时，传输的起始由主机发送数据来启动，此时，主机可激活时钟并开始传输。当传输的最后一个时钟周期结束时，传输结束。

当器件为从机并且 CPHA=0 时，传输在 SSEL 信号激活时开始，并在 SSEL 变为高电平时结束。当器件为从机且 CPHA=1 时，如果该器件被选择，传输从第一个时钟沿开始，并在数据采样的最后一个时钟沿结束。

说明：对于 16 位的数据发送，发送完第 1 个字节数据之后，接着再发送第 2 字节数据

即可，在 2 字节发送过程中，从机片选要保持有效；对于非全双工(接收与发送不能同时进行)的从机，主机读取从机数据时也需要发送一字节数据(可以任意数据)，以便主机产生 SPI 时钟信号，从而读取到从机发送的数据。

3. SPI 功能模块描述

概述

有 5 个寄存器控制 SPI 功能模块，这里只作如下简单的描述，在第 5.13.5 节中将详细讲述。

- SPCR——SPI 控制寄存器包含一些可编程位来控制 SPI 功能模块的功能。该寄存器必须在数据传输之前进行设定。
- SPSR——SPI 状态寄存器为只读的寄存器，用于监视 SPI 功能模块的状态，包括一般性功能和异常状况。该寄存器的主要用途是检测数据传输的完成，这通过判断能 SPIF 位来实现；其它位用于指示异常状况。
- SPDR——SPI 数据寄存器用于发送和接收数据，在发送时向 SPI 数据寄存器写入数据。串行数据的发送和接收通过内部移位寄存器来实现。数据寄存器和内部移位寄存器之间没有缓冲区，写 SPDR 会使数据直接进入内部移位寄存器，因此数据只能在上一次数据发送完成之后写入该寄存器。读数据是带有缓冲区的，当传输结束时，接收到的数据转移到一个单字节的数据缓冲区，读 SPI 数据寄存器将返回读缓冲区的值。SPI 数据传输方向如图 5.53 所示。
- SPCCR——SPI 时钟计数器寄存器，用于设置 SPI 时钟分频值。当 SPI 功能模块处于主模式时，SPCCR 寄存器用于控制时钟速率，即 SPI 总线速率。该寄存器必须在数据传输之前设定。**当 SPI 功能模块处于从模式时，该寄存器无效。**
- SPINT——SPI 中断标志寄存器，该寄存器包含了 SPI 的中断标志位。

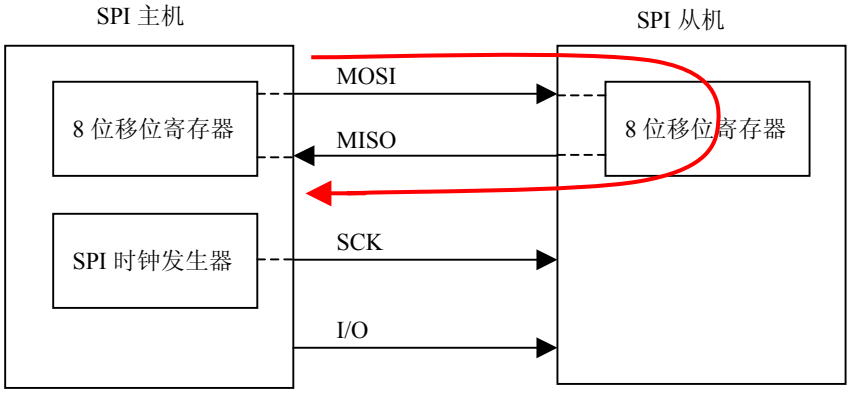


图 5.55 SPI 数据传输方向

主机操作

下面的步骤描述了 SPI 设置为主机时如何 处理数据传输。该处理假设上一次的数据传输已经结束。

1. 设置 SPCCR 寄存器，得到相应的 SPI 时钟。
2. 设置 SPCR 寄存器，控制 SPI 为主机。
3. 控制片选信号，选择从机。
4. 将要发送的数据写入 SPDR 寄存器，即启动 SPI 数据传输。
5. 读取 SPSR 寄存器，等待 SPIF 位置位。SPIF 位将在数据传输的最后一个周期之后由硬件自动置位。
6. 从 SPI 数据寄存器中读出接收到的数据（可选）。

7. 如果有更多数据需要发送，则跳到第 3 步，否则取消对从机的选择。

主机初始化示例程序见程序清单 5.27，程序首先判断需要设置的 SPI 时钟分频值是否合法，如果设置值小于 8 则强行设置为 8。由于 SPCCR 寄存器的值必须为偶数，所以将分频值和 0xFE 进行“与”操作。

程序清单 5.27 SPI 主机初始化示例

```
#define   MSTR            (1<<5)
#define   CPOL            (1<<4)
#define   CPHA            (1<<3)
#define   LSBF            (1<<6)

#define   SPI_MODE        (MSTR | CPOL)   /* SPI 接口模式，MSTR=1，CPOL=1，CPHA=0，LSBF=0 */

/*****************************************************************************
* 名称：MSpiIni()
* 功能：初始化 SPI 接口，设置为主机。
* 入口参数：fdiv            SPI 时钟分频值，大于 8 的偶数
* 出口参数：无
*****************************************************************************/
void   MSpiIni(uint8 fdiv)
{   if(fdiv<8) fdiv = 8;
    S0PCCR = fdiv&0xFE;                    // 设置 SPI 时钟分频
    S0PCR = SPI_MODE;
}
```

SPI 主机数据发送和接收示例程序见程序清单 5.28，向 S0PDR 寄存器(即 SPI0 的 SPDR)写入一字节数据后，即可启动数据发送。SPI 功能模块在发送数据时同时接收一字节数据，并将接收到的数据返回。

程序清单 5.28 SPI 主机数据发送和接收程序

```
/*****************************************************************************
* 名称：MSendData()
* 功能：向 SPI 总线发送数据，并接收从机发回的数据。
* 入口参数：data            待发送的数据
* 出口参数：返回值为接收到的数据
*****************************************************************************/
uint8   MSendData(uint8 data)
{   IO0CLR = HC595_CS;                    // 片选

    S0PDR = data;
    while( 0==(S0PSR&0x80) );              // 等待 SPIF 置位，即等待数据发送完毕

    IO0SET = HC595_CS;
```

```
    return(S0PDR);
}
```

从机操作

下面的步骤描述了 SPI 设置为从机时如何处理数据传输。该处理假设上一次的数据传输已经结束。要求驱动 SPI 逻辑的系统时钟速度至少 8 倍于 SPI。

1. 设置 SPCR 寄存器，控制 SPI 为从机。
2. 将要发送的数据写入 SPI 数据寄存器（可选）。注意，这只能在从 SPI 传输没有进行时执行。
3. 读取 SPSR 寄存器，等待 SPIF 位置位。SPIF 位将在 SPI 数据传输的最后一个采样时钟沿后由硬件自动置位。
4. 从 SPI 数据寄存器中读出接收到的数据（可选）。
5. 如果有更多数据需要发送，则跳到第 2 步。

从机初始化示例程序见程序清单 5.29，程序只对 S0PCR 进行设置，控制 SPI 为从机。为了能够上 SPI 主机进行通讯，需要正确设置 CPOL、CPHA、LSBF 控制位。注意，SPI 时钟脉冲是由主机产生，所以从机无需初始化 S0PCCR 寄存器。

程序清单 5.29 SPI 从机初始化示例

```
#define   CPOL              (1<<4)
#define   CPHA              (1<<3)
#define   LSBF              (1<<6)

#define   SPI_MODE          (CPOL)          /* SPI 接口模式，MSTR=0，CPOL=1，CPHA=0，LSBF=0 */

/*****************************************************************************
* 名称：SSpiIni()
* 功能：初始化 SPI 接口，设置为从机。
* 入口参数：无
* 出口参数：无
*****************************************************************************/
void   SSpiIni(void)
{   S0PCR = SPI_MODE;
}
```

SPI 从机数据发送示例程序见程序清单 5.30，向 S0PDR 寄存器(即 SPI0 的 SPDR)写入一字节数据，然后等待主机读数据操作。当然，用户可以使用中断形式进行数据的发送，这样有助于提高整个系统程序的效率。

程序清单 5.30 SPI 从机数据发送程序

```
/*****************************************************************************
* 名称：SSendData()
* 功能：SPI 从机发送数据。
* 入口参数：data            待发送的数据
```

```
* 出口参数：无
*******************************************************************************/
void   SSendData(uint8 data)
{   S0PDR = data;
    while( 0==(S0PSR&0x80) );                  // 等待 SPIF 置位，即等待数据发送完毕
}
```

SPI 从机数据接收示例程序见程序清单 5.31，程序首先等待 S0PSR 寄存器的 SPIF 位为 1，然后读取 S0PDR 寄存器内的数据（即是接收到的数据）。当然，用户可以使用中断形式进行数据的接收，这样有助于提高整个系统程序的效率。

程序清单 5.31 SPI 从机数据接收程序

```
/*******************************************************************************
* 名称：SRcvData()
* 功能：SPI 从机接收数据。
* 入口参数：无
* 出口参数：返回值为读取到的数据。
*******************************************************************************/
uint8   SRcvData(void)
{   while( 0==(S0PSR&0x80) ) ;
    return(S0PDR);
}
```

异常状况

读溢出——当 SPI 功能模块内部读缓冲区包含没有读出的数据，而新的传输已经完成，那么这时候就会发生读溢出。SPIF 位置位表示读缓冲区包含了有效数据。当一次传输结束时，SPI 功能模块需要将接收到的数据移到读缓冲区。如果 SPIF 位置位 (读缓冲区已满)，新接收到的数据将会丢失，而状态寄存器的读溢出 (ROVR)位将置位。

写冲突——我们在前面提到过，在 SPI 总线接口与内部移位寄存器之间没有写缓冲区。这样在 SPI 数据传输过程当中不应向 SPI 数据寄存器写入数据。不能向 SPI 数据寄存器写入数据的时间从传输启动时开始，直到 SPIF 置位时读取状态寄存器为止。如果在这段时间内写 SPI 数据寄存器，写入的数据将会丢失，状态寄存器中的写冲突位（WCOL）置位。

模式错误——SSEL 信号在 SPI 功能模块为主机时必须无效，不能用作 GPIO。当 SPI 功能模块为主机时，如果 SSEL 信号被激活(将 SSEL 变为低电平)，表示有另外一个主机将该器件选择为从机。这种状态称为模式错误。当检测到一个模式错误时，状态寄存器的模式错误位（MODF）位置位，SPI 信号驱动器关闭，而 SPI 模式转换为从模式。

从机中止——如果 SSEL 信号在传输结束之前变为高电平，从传输将被认为中止。此时，正在处理的发送或接收数据都将丢失，状态寄存器的从机中止（ABRT）位置位。

5.13.4 结构

SPI0 和 SPI1 接口中的 SPI 方框图见图 5.56。

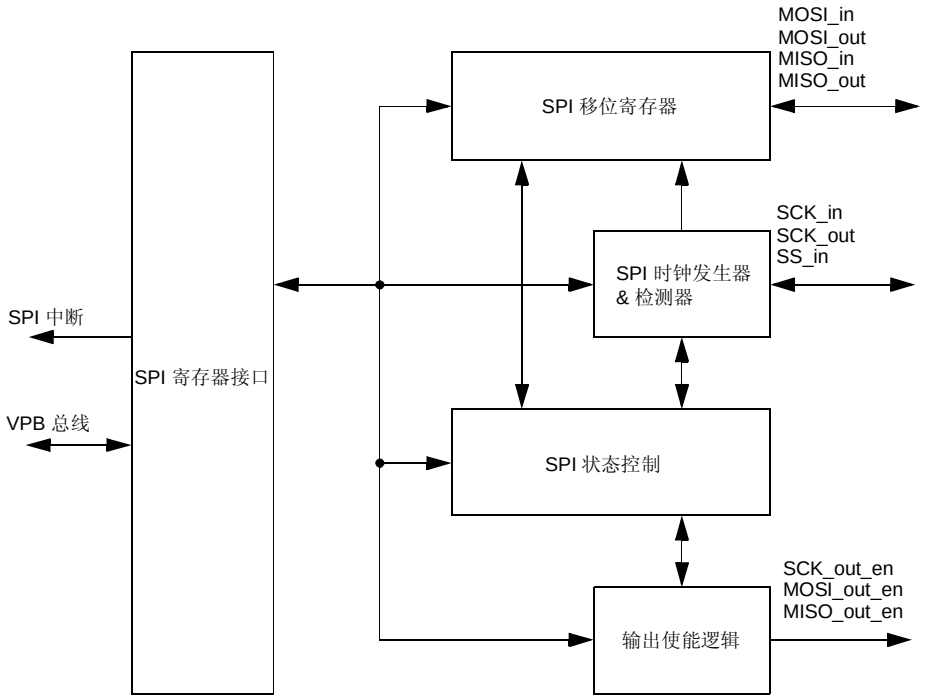


图 5.56 SPI 方框图

5.13.5 寄存器描述

寄存器汇总

SPI 包含 5 个寄存器，见表 5.114。所有寄存器都可以字节、半字和字的形式访问。

表 5.114 SPI 寄存器映射

名称	描述	访问	复位值*	SPI0 地址&名称	SPI1 地址&名称
SPCR	SPI 控制寄存器。该寄存器控制 SPI 的操作模式。	R/W	0	0xE0020000 S0SPCR	0xE0030000 S1SPCR
SPSR	SPI 状态寄存器。该寄存器显示 SPI 的状态。	RO	0	0xE0020004 S0SPSR	0xE0030004 S1SPSR
SPDR	SPI 数据寄存器。该双向寄存器为 SPI 提供发送和接收的数据。发送数据通过写该寄存器提供。SPI 接收的数据可从该寄存器读出。	R/W	0	0xE0020008 S0SPDR	0xE0030008 S1SPDR
SPCCR	SPI 时钟计数寄存器。 该寄存器控制主机 SCK 的频率。	R/W	0	0xE002000C S0SPCCR	0xE003000C S1SPCCR
SPINT	SPI 中断标志寄存器。该寄存器包含 SPI 接口的中断标志。	R/W	0	0xE002001C S0SPINT	0xE003001C S1SPINT

* 复位值仅指已使用位中保存的数据，不包括保留位的内容。

SPI 控制寄存器（S0SPCR - 0xE0020000, S1SPCR - 0xE0030000）

SPCR 寄存器根据每个配置位的设定来控制 SPI 的操作，见表 5.115。

表 5.115 SPI 控制寄存器

SPCR	功能	描述	复位值
2:0	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
3	CPHA	时钟相位控制决定 SPI 传输时数据和时钟的关系并控制从机传输的起始和结束。当该位为 1 时，数据在 SCK 的第二个时钟沿采样。当 SSEL 信号激活时，传输从第一个时钟沿开始并在最后一个采样时钟沿结束。 当该位为 0 时，数据在 SCK 的第一个时钟沿采样。传输从 SSEL 信号激活时开始，并在 SSEL 信号无效时结束。	0
4	CPOL	时钟极性控制。当该位为 1 时，SCK 为低有效。为 0 时，SCK 为高有效。	0
5	MSTR	主模式选择。为 1 时，SPI 处于主模式。为 0 时，SPI 处于从模式。	0
6	LSBF	LSBF 用来控制传输的每个字节的移动方向。为 1 时，SPI 数据传输 LSB (位 0)在先。为 0 时，SPI 数据传输 MSB (位 7)在先。	0
7	SPIE	SPI 中断使能。为 1 时，每次 SPIF 或 MODF 置位时都会产生硬件中断。为 0 时，SPI 中断被禁止。	0

SPI 状态寄存器（S0SPSR - 0xE0020004, S1SPSR - 0xE0030004）

SPSR 寄存器根据配置位的设定来控制 SPI 的操作，见表 5.116。

表 5.116 SPI 状态寄存器

SPSR	功能	描述	复位值
2:0	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
3	ABRT	从机中止。该位为 1 时表示发生了从机中止。当读取该寄存器时，该位清零。	0
4	MODF	模式错误。为 1 时表示发生了模式错误。先通过读取该寄存器清零 MODF 位，再写 SPI 控制寄存器。	0
5	ROVR	读溢出。为 1 时表示发生了读溢出。当读取该寄存器时，该位清零。	0
6	WCOL	写冲突。为 1 时表示发生了写冲突。先通过读取该寄存器清零 WCOL 位，再访问 SPI 数据寄存器。	0
7	SPIF	SPI 传输完成标志。为 1 时表示一次 SPI 数据传输完成。在主模式下，该位在传输的最后一个周期置位。在从机模式下，该位在 SCK 的最后一个数据采样边沿置位。当第一次读取该寄存器时，该位清零。然后才能访问 SPI 数据寄存器。 注：SPIF 不是 SPI 中断标志。中断标志位于 SPINT 寄存器中。	0

SPI 数据寄存器（S0SPDR - 0xE0020008,S1SPDR – 0xE0030008）

双向数据寄存器为 SPI 提供数据的发送和接收，见表 5.117。发送数据通过将数据写入该寄存器来实现。SPI 接收的数据可从该寄存器中读出。处于主模式时，写该寄存器将启动 SPI 数据传输。从数据传输开始到 SPIF 状态位置位并且还没有读取状态寄存器的这段时间内不能对该寄存器执行写操作。

表 5.117 SPI 数据寄存器

SPDR	功能	描述	复位值
7:0	数据	SPI 双向数据	0

SPI 时钟计数寄存器（S0SPCCR - 0xE002000C，S1SPCCR – 0xE003000C）

该寄存器控制主机 SCK 的频率，见表 5.118。寄存器指示构成一个 SPI 时钟的 pclk 周期的数据。该寄存器的值必须为偶数。因此 bit0 必须为 0。该寄存器的值还必须大于等于 8。如果寄存器的值不符合上述条件，可能导致产生不可预测的动作。

SPI 速率可以这样进行计算：PCLK 速率/SPCCR 值。pclk 速率为 CCLK/VPB 除数，由 VPBDIV 寄存器的内容决定。

表 5.118 SPI 时钟计数寄存器

SPCCR	功能	描述	复位值
7:0	计数值	SPI 时钟计数值设定	0

SPI 中断寄存器（S0SPINT - 0xE002001C,S1SPINT – 0xE003001C）

该寄存器包含 SPI 接口的中断标志，见表 5.119。

表 5.119 SPI 中断寄存器

SPINT	功能	描述	复位值
0	SPI 中断	SPI 中断标志。由 SPI 接口置位以产生中断。向该位写入 1 清零。 注：当 SPIE=1 并且 SPIF 和 WCOL 位中至少有一位为 1 时该位置位。但是，只有当 SPI 中断位置位并且 SPI 中断在 VIC 中被使能，SPI 中断才能由中断处理软件处理。	0
7:1	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

5.14 定时器 0 和定时器 1

5.14.1 描述

定时器 0 和定时器 1 除了外设基地址不相同以外，其它都相同。

定时器对外设时钟（pclk）周期进行计数，根据 4 个匹配寄存器的设定，可设置为匹配(即到达匹配寄存器指定的定时值)时产生中断或执行其它动作。它还包括 4 个捕获输入，用于在输入信号发生跳变时捕获定时器值，并可选择产生中断。

5.14.2 特性

- 带可编程 32 位预分频器的 32 位定时器/计数器
- 具有多达 4 路捕获通道。当输入信号跳变时可取得定时器的瞬时值。也可选择使捕获事件产生中断。
- 4 个 32 位匹配寄存器，匹配时的动作有如下 3 种：
  ——匹配时定时器继续工作，可选择产生中断
  ——匹配时停止定时器，可选择产生中断
  ——匹配时复位定时器，可选择产生中断
- 4 个对应于匹配寄存器的外部输出，匹配时的输出有如下 4 种：
  ——匹配时设置为低电平
  ——匹配时设置为高电平
  ——匹配时翻转
  ——匹配时无动作

5.14.3 应用

- 用于对内部事件进行计数的间隔定时器
- 通过捕获输入实现脉宽解调器
- 自由运行的定时器

5.14.4 管脚描述

表 5.120 所示为每个定时器相关管脚的简要描述。CAP0.x、MAT0.x 为定时器 0 的相关管脚，CAP1.x、MAT1.x 为定时器 1 的相关管脚。

同一路捕获的输入管脚可能有几个，当选择多个管脚作捕获功能时，它们的输入将进行逻辑或，如图 5.57 所示。

表 5.120 定时器相关管脚概况

管脚名称	管脚方向	管脚描述
CAP0.3～CAP00 CAP1.3～CAP1.0	输入	**捕获信号** 捕获管脚的跳变可配置为将定时器值装入一个捕获寄存器，并可选择产生一个中断。可选择多个管脚用作捕获功能，而且，假设如果有 2 个管脚被选择并行提供 CAP0.2 功能，它们的输入将进行逻辑或，所得结果用作一个捕获输入。 3 个管脚可同时选择用作 CAP0.0 的功能。 2 个管脚可同时选择用作 CAP0.1 的功能。 3 个管脚可同时选择用作 CAP0.2 的功能。 1 个管脚可选择用作 CAP0.3 的功能。 1 个管脚可选择用作 CAP1.0 的功能。 1 个管脚可选择用作 CAP1.1 的功能。 2 个管脚可选择用作 CAP1.2 的功能。 2 个管脚可选择用作 CAP1.3 的功能。
MAT0.3～MAT0.0 MAT1.3～MAT1.0	输出	**外部匹配输出 0/1** 当匹配寄存器 0/1（MR3:0）等于定时器计数器（TC）时，该输出可翻转，变为低电平、变为高电平或不变。外部匹配寄存器（EMR）控制该输出的功能。可选择多个管脚并行用作匹配输出功能。例如，同时选择 2 个管脚并行提供 MAT1.3 功能。 2 个管脚可同时选择用作 MAT0.0 的功能。 2 个管脚可同时选择用作 MAT0.1 的功能。 2 个管脚可同时选择用作 MAT0.2 的功能。 1 个管脚可选择用作 MAT0.3 的功能。 1 个管脚可选择用作 MAT1.0 的功能。 1 个管脚可选择用作 MAT1.1 的功能。 2 个管脚可选择用作 MAT1.2 的功能。 2 个管脚可选择用作 MAT1.3 的功能。

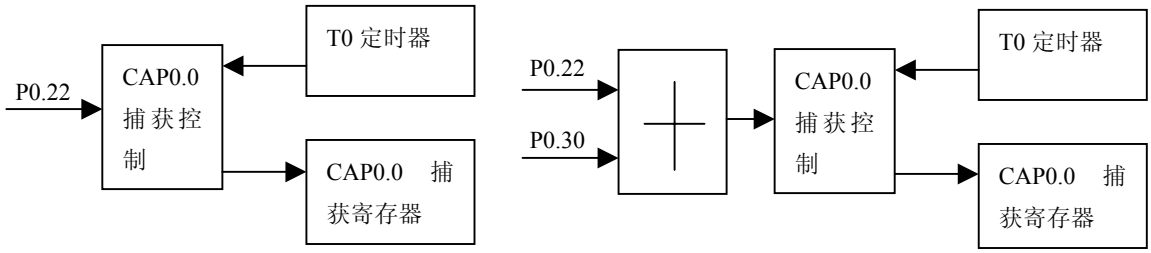
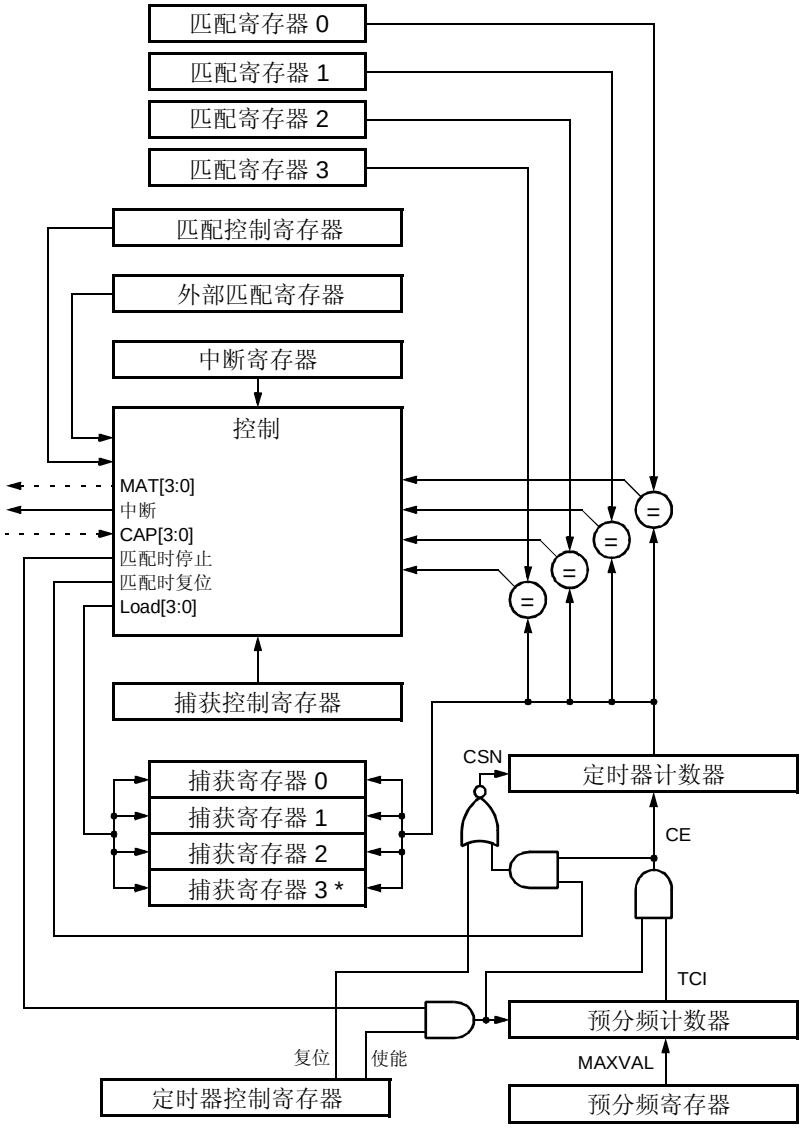


图 5.57 1 个和多个输入管脚捕获示意图

5.14.5 结构

定时器 0 和定时器 1 的方框图，见图 5.58。



* 注：捕获寄存器 3 不能用于定时器 0

图 5.58 定时器方框图

5.14.6 寄存器描述

寄存器汇总

每个定时器所包含的寄存器如表 5.121 所示。

表 5.121 定时器 0 和定时器 1 寄存器映射

名称	描述	访问	复位值	定时器 0 地址&名称	定时器 1 地址&名称
IR	中断寄存器 可以写 IR 来清除中断。可读取 IR 来识别哪个中断源被挂起。	R/W	0	0xE0004000 T0IR	0xE0008000 T1IR
TCR	定时器控制寄存器 TCR 用于控制定时器计数器功能。定时器计数器可通过 TCR 禁止或复位。	R/W	0	0xE0004004 T0TCR	0xE0008004 T1TCR
TC	定时器计数器 32 位 TC 每经过 PR+1 个 pclk 周期加 1。TC 通过 TCR 进行控制。	R/W	0	0xE0004008 T0TC	0xE0008008 T1TC
PR	预分频寄存器 32 位 TC 每经过 PR+1 个 pclk 周期加 1。	R/W	0	0xE000400C T0PR	0xE000800C T1PR
PC	预分频计数器 每当 32 位 PC 的值增加到等于 PR 中保存的值时，TC 加 1。	R/W	0	0xE0004010 T0PC	0xE0008010 T1PC
MCR	匹配控制寄存器 MCR 用于控制在匹配时是否产生中断或复位 TC。	R/W	0	0xE0004014 T0MCR	0xE0008014 T1MCR
MR0	匹配寄存器 0 MR0 可通过 MCR 设定为在匹配时复位 TC，停止 TC 和 PC 和/或产生中断。	R/W	0	0xE0004018 T0MR0	0xE0008018 T1MR0
MR1	匹配寄存器 1 MR1 可通过 MCR 设定为在匹配时复位 TC，停止 TC 和 PC 和/或产生中断。	R/W	0	0xE000401C T0MR1	0xE000801C T1MR1
MR2	匹配寄存器 2 MR2 可通过 MCR 设定为在匹配时复位 TC，停止 TC 和 PC 和/或产生中断。	R/W	0	0xE0004020 T0MR2	0xE0008020 T1MR2
MR3	匹配寄存器 3 MR3 可通过 MCR 设定为在匹配时复位 TC，停止 TC 和 PC 和/或产生中断。	R/W	0	0xE0004024 T0MR3	0xE0008024 T1MR3
CCR	捕获控制寄存器 CCR 控制用于装载捕获寄存器的捕获输入边沿以及在发生捕获时是否产生中断。	R/W	0	0xE0004028 T0CCR	0xE0008028 T1CCR
CR0	捕获寄存器 0 当在 CAP0.0(CAP1.0)上产生捕获事件时，CR0 装载 TC 的值。	RO	0	0xE000402C T0CR0	0xE000802C T1CR0
CR1	捕获寄存器 1 当在 CAP0.1(CAP1.1)上产生捕获事件时，CR1 装载 TC 的值。	RO	0	0xE0004030 T0CR1	0xE0008030 T1CR1
CR2	捕获寄存器 2 当在 CAP0.2(CAP1.2)上产生捕获事件时，CR2 装载 TC 的值。	RO	0	0xE0004034 T0CR2	0xE0008034 T1CR2
CR3	捕获寄存器 3 当在 CAP0.3(CAP1.3)上产生捕获事件时，CR3 装载 TC 的值。	RO	0	0xE0004038 T0CR3	0xE0008038 T1CR3
EMR	外部匹配寄存器 EMR 控制外部匹配管脚 MAT0.0～MAT0.3(MAT1.0～MAT1.3)。	R/W	0	0xE000403C T0EMR	0xE000803C T1EMR

中断寄存器（IR：定时器 0 - T0IR：0xE00040000；定时器 1 - T1IR：0xE0008000）

中断寄存器包含 4 个位用于匹配中断，4 个位用于捕获中断。如果有中断产生，IR 中的对应位会置位，否则为 0。向对应的 IR 位写入 1 会复位中断。写入 0 无效。

IR 寄存器描述见表 5.122。

表 5.122 中断寄存器

IR	功能	描述	复位值
0	MR0 中断	匹配通道 0 的中断标志	0
1	MR1 中断	匹配通道 1 的中断标志	0
2	MR2 中断	匹配通道 2 的中断标志	0
3	MR3 中断	匹配通道 3 的中断标志	0
4	CR0 中断	捕获通道 0 事件的中断标志	0
5	CR1 中断	捕获通道 1 事件的中断标志	0
6	CR2 中断	捕获通道 2 事件的中断标志	0
7	CR3 中断	捕获通道 3 事件的中断标志	0

定时器控制寄存器（TCR：定时器 0 – T0TCR：0xE0004004；定时器 1 – T1TCR：0xE0008004）

定时器控制寄存器 TCR 用于控制定时器计数器的操作。

TCR 寄存器描述见表 5.123。

表 5.123 定时器控制寄存器

TCR	功能	描述	复位值
0	计数器使能	为 1 时，定时器计数器和预分频计数器使能计数。为 0 时，计数器被禁止。	0
1	计数器复位	为 1 时，定时器计数器和预分频计数器在 pclk 的下一个上升沿同步复位。计数器在 TCR 的 bit1 恢复为 0 之前保持复位状态。	0

定时器计数器（TC：定时器 0 – T0TC：0xE0004008；定时器 1 – T1TC：0xE0008008）

当预分频计数器到达计数的上限时，32 位定时器计数器 TC 加 1。如果 TC 在到达计数上限之前没有被复位，它将一直计数到 0xFFFFFFFF 然后翻转到 0x00000000，该事件不会产生中断。如果需要，可用匹配寄存器检测溢出。

预分频寄存器（PR：定时器 0 – T0PR：0xE000400C；定时器 1 – T1PR：0xE000800C）

32 位预分频寄存器指定了预分频计数器的最大值。

预分频计数器寄存器（PC：定时器 0 – T0PC：0xE0004010；定时器 1 – T1PC：0xE0008010）

预分频计数器使用某个常量来控制 pclk 的分频。这样可实现控制定时器分辨率和定时器溢出时间之间的关系。预分频计数器每个 pclk 周期加 1。当其到达预分频寄存器中保存的值时，定时器计数器加 1，预分频计数器在下个 pclk 周期复位。这样，当 PR=0 时，定时器计数器每个 pclk 周期加 1，当 PR=1 时，定时器计数器每 2 个 pclk 周期加 1。

匹配寄存器（MR0 - MR3）

匹配寄存器值连续与定时器计数值相比较。当两个值相等时自动触发相应动作。这些动作包括产生中断，复位定时器计数器或停止定时器。所执行的动作由 MCR 寄存器控制。

匹配控制寄存器（MCR：定时器 0 – T0MCR：0xE0004014；定时器 1 – T1MCR：

0xE00080014）

匹配控制寄存器用于控制在发生匹配时所执行的操作。每个位的功能见表 5.124。

表 5.124 匹配控制寄存器

MCR	功能	描述	复位值
0	中断(MR0)	为 1 时，MR0 与 TC 值的匹配将产生中断。为 0 时，中断被禁止。	0
1	复位(MR0)	为 1 时，MR0 与 TC 值的匹配将使 TC 复位。为 0 时，该特性被禁止。	0
2	停止(MR0)	为 1 时，MR0 与 TC 值的匹配将使 TC 和 PC 停止，TCR 的 bit0 清零。为 0 时，该特性被禁止。	0
3	中断(MR1)	为 1 时，MR1 与 TC 值的匹配将产生中断。为 0 时，中断被禁止。	0
4	复位(MR1)	为 1 时，MR1 与 TC 值的匹配将使 TC 复位。为 0 时，该特性被禁止。	0
5	停止(MR1)	为 1 时，MR1 与 TC 值的匹配将使 TC 和 PC 停止，TCR 的 bit0 清零。为 0 时，该特性被禁止。	0
6	中断(MR2)	为 1 时，MR2 与 TC 值的匹配将产生中断。为 0 时，中断被禁止。	0
7	复位(MR2)	为 1 时，MR2 与 TC 值的匹配将使 TC 复位。为 0 时，该特性被禁止。	0
8	停止(MR2)	为 1 时，MR2 与 TC 值的匹配将使 TC 和 PC 停止，TCR 的 bit0 清零。为 0 时，该特性被禁止。	0
9	中断(MR3)	为 1 时，MR3 与 TC 值的匹配将产生中断。为 0 时，中断被禁止。	0
10	复位(MR3)	为 1 时，MR3 与 TC 值的匹配将使 TC 复位。为 0 时，该特性被禁止。	0
11	停止(MR3)	为 1 时，MR3 与 TC 值的匹配将使 TC 和 PC 停止，TCR 的 bit0 清零。为 0 时，该特性被禁止。	0

捕获寄存器（CR0 - CR3）

每个捕获寄存器都与一个/几个器件管脚相关联。当管脚发生特定的事件时，可将定时器计数值装入该寄存器。捕获控制寄存器的设定决定捕获功能是否使能以及捕获事件在管脚的上升沿、下降沿或是双边沿发生。

捕获控制寄存器（CCR：定时器 0 – T0CCR：0xE0004028；定时器 1 – T1CCR：0xE0008028）

当发生捕获事件时，捕获控制寄存器用于控制将定时器计数值是否装入 4 个捕获寄存器中的一个以及是否产生中断。同时设置上升沿和下降沿位也是有效的配置，这样会在双边沿触发捕获事件。

CCR 寄存器描述见表 5.125。在下面的描述中，“n”代表定时器的编号 0 或 1。

表 5.125 捕获控制寄存器

CCR	功能	描述	复位值
0	CAPn.0 上升沿捕获	为 1 时，CAPn.0 上 0 到 1 的跳变将导致 TC 的内容装入 CR0。为 0 时，该特性被禁止。	0
1	CAPn.0 下降沿捕获	为 1 时，CAPn.0 上 1 到 0 的跳变将导致 TC 的内容装入 CR0。为 0 时，该特性被禁止。	0
2	CAPn.0 事件中断	为 1 时，CAPn.0 的捕获事件所导致的 CR0 装载将产生一个中断。为 0 时，该特性被禁止。	0
3	CAPn.1 上升沿捕获	为 1 时，CAPn.1 上 0 到 1 的跳变将导致 TC 的内容装入 CR1。为 0 时，该特性被禁止。	0

接上表

CCR	功能	描述	复位值
4	CAPn.1 下降沿捕获	为 1 时，CAPn.1 上 1 到 0 的跳变将导致 TC 的内容装入 CR1。为 0 时，该特性被禁止。	0
5	CAPn.1 事件中断	为 1 时，CAPn.1 的捕获事件所导致的 CR1 装载将产生一个中断。为 0 时，该特性被禁止。	0
6	CAPn.2 上升沿捕获	为 1 时，CAPn.2 上 0 到 1 的跳变将导致 TC 的内容装入 CR2。为 0 时，该特性被禁止。	0
7	CAPn.2 下降沿捕获	为 1 时，CAPn.2 上 1 到 0 的跳变将导致 TC 的内容装入 CR2。为 0 时，该特性被禁止。	0
8	CAPn.2 事件中断	为 1 时，CAPn.2 的捕获事件所导致的 CR2 装载将产生一个中断。为 0 时，该特性被禁止。	0
9	CAPn.3 上升沿捕获	为 1 时，CAPn.3 上 0 到 1 的跳变将导致 TC 的内容装入 CR3。为 0 时，该特性被禁止。	0
10	CAPn.3 下降沿捕获	为 1 时，CAPn.3 上 1 到 0 的跳变将导致 TC 的内容装入 CR3。为 0 时，该特性被禁止。	0
11	CAPn.3 事件中断	为 1 时，CAPn.3 的捕获事件所导致的 CR3 装载将产生一个中断。为 0 时，该特性被禁止。	0

外部匹配寄存器（EMR：定时器 0 – T0EMR：0xE000403C；定时器 1 – T1EMR：0xE0008003C）

外部匹配寄存器提供外部匹配管脚 MATn.0～MATn.3(n 为 0 或 1)的控制和状态。

EMR 寄存器描述见表 5.126。

表 5.126 外部匹配寄存器

EMR	功能	描述	复位值
0	外部匹配 0	不管 MAT0.0/MAT1.0 是否连接到管脚，该位都会反映 MAT0.0/MAT1.0 的状态。当 MR0 发生匹配时，该输出可翻转，变为低电平，变为高电平或不执行任何动作。位 EMR[4:5]控制该输出的功能。	0
1	外部匹配 1	不管 MAT0.1/MAT1.1 是否连接到管脚，该位都会反映 MAT0.1/MAT1.1 的状态。当 MR1 发生匹配时，该输出可翻转，变为低电平，变为高电平或不执行任何动作。位 EMR[6:7]控制该输出的功能。	0
2	外部匹配 2	不管 MAT0.2/MAT1.2 是否连接到管脚，该位都会反映 MAT0.2/MAT1.2 的状态。当 MR2 发生匹配时，该输出可翻转，变为低电平，变为高电平或不执行任何动作。位 EMR[8:9]控制该输出的功能。	0
3	外部匹配 3	不管 MAT0.3/MAT1.3 是否连接到管脚，该位都会反映 MAT0.3/MAT1.3 的状态。当 MR3 发生匹配时，该输出可翻转，变为低电平，变为高电平或不执行任何动作。位 EMR[10:11]控制该输出的功能。	0
5:4	外部匹配控制 0	决定外部匹配 0 的功能。**表 5.127 所示为这两个位的编码。**	0
7:6	外部匹配控制 1	决定外部匹配 1 的功能。**表 5.127 所示为这两个位的编码。**	0
9:8	外部匹配控制 2	决定外部匹配 2 的功能。**表 5.127 所示为这两个位的编码。**	0

接上表

EMR	功能	描述	复位值
11:10	外部匹配控制 3	决定外部匹配 3 的功能。**表 5.127 所示为这两个位的编码。**	0

表 5.127 外部匹配控制

EMR[11:10], EMR[9:8] EMR[7:6],或 EMR[5:4]	功能
00	不执行任何动作
01	将对应的外部匹配输出设置为 0（如果连接到管脚，则输出低电平）
10	将对应的外部匹配输出设置为 1（如果连接到管脚，则输出高电平）
11	使对应的外部匹配输出翻转

5.14.7 定时器举例操作

图 5.59 所示为定时器配置为在匹配时复位计数并产生中断。预分频器设置为 2，匹配寄存器设置为 6。在发生匹配的定时器周期结束时，定时器计数值复位。这样就使匹配值具有完整长度的周期。指示匹配发生的中断在定时器到达匹配值的下一个时钟产生。

图 5.60 所示为定时器配置为在匹配时停止并产生中断。预分频器设置为 2，匹配寄存器设置为 6。在定时器到达匹配值的下一个周期中，TCR 中的定时器使能位清零并产生指示匹配发生的中断。

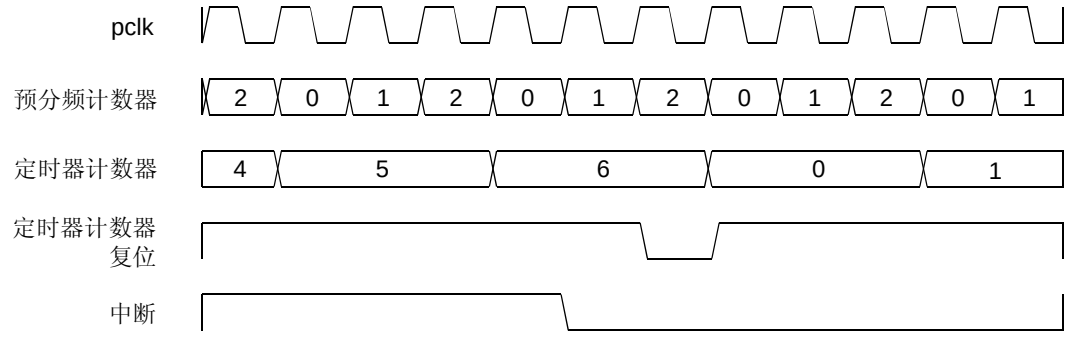


图 5.59 定时器周期设置为 PR=2, MRx=6, 匹配时使能中断和复位

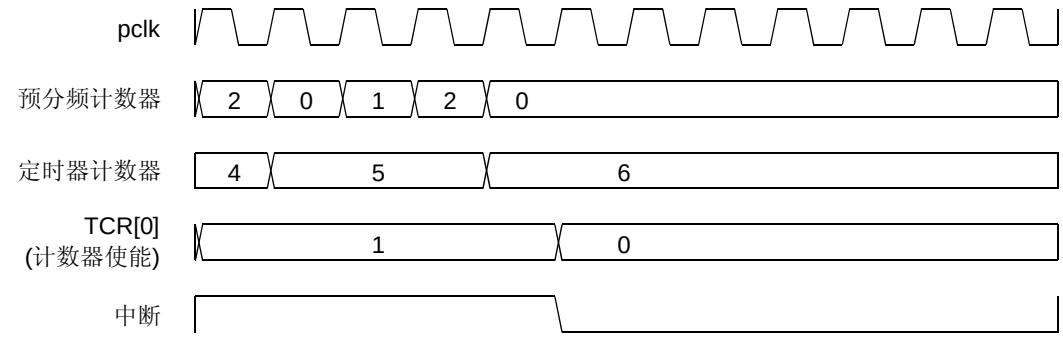


图 5.60 定时器周期设置为 PR=2, MRx=6, 匹配时使能中断和停止定时器

5.14.8 使用示例

LPC2114/2124/2210/2212/2214 的两个 32 位定时器，分别具有 4/3 路捕获、4 路比较匹配并输出电路，定时器是增量计数的，但上溢时不会产生中断标志，而只能通过比较匹配或捕获输入产生中断标志。两个定时器具有同样的寄存器，只是地址不同而已。

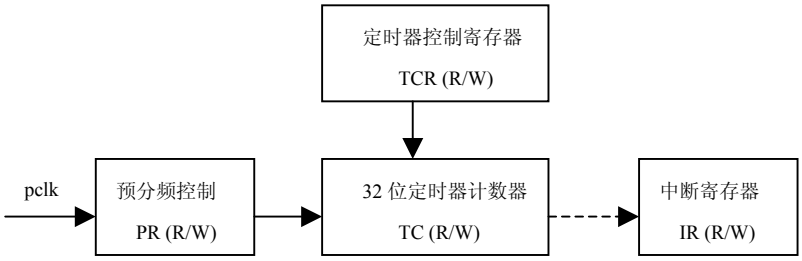


图 5.61 基本定时器的寄存器功能框图

如图 5.61，32 位定时器 TC 的计数频率由 pclk 经过 PR 进行分频控制得到，而定时器的启动/停止、计数复位由 TCR 控制，当有捕获事件或比较匹配事件发生时，IR 会设置相关中断标志(因为不是定时器溢出而产生中断，所以上图采用虚线连接)，若已打开中断允许(VIC)则会产生中断。当然，预分频控制器 PR 只是控制分频数，而其对应的分频计数器是 PC，但用户无须操作 PC 寄存器。

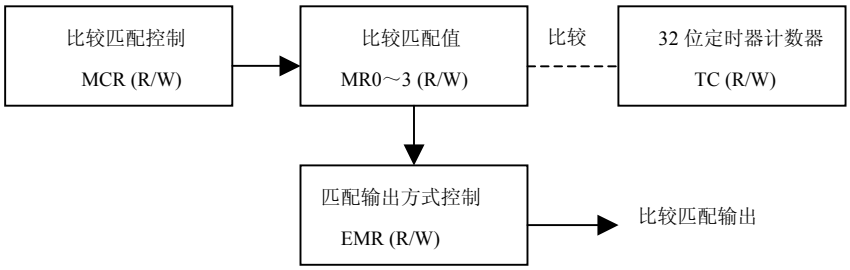


图 5.62 定时器的比较匹配寄存器功能框图

如图 5.62，定时器比较匹配由控制寄存器 MCR 进行匹配操作设置，而 MR0～3 寄存器则为 4 路比较匹配通道的比较值。当比较匹配时，将会按照 MCR 设置的方法产生中断或复位 TC 等，而且 EMR 可以控制比较匹配输出，可以匹配输出高电平、低电平、电平翻转等。

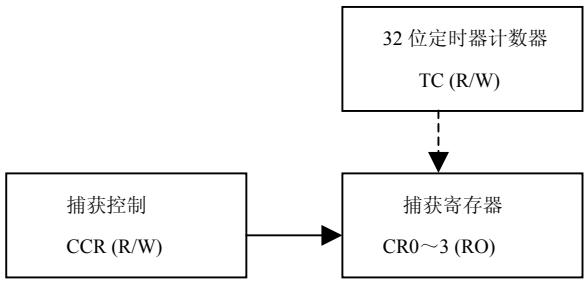


图 5.63 定时器的捕获寄存器功能框图

如图 5.63，定时器 TC 运行过程中，当有捕获触发信号产生时，捕获电路将会立即把当时的定时器值 TC 复制到对应触发通道的捕获寄存器中。捕获可以设置为上升沿触发、下降沿触发、双边沿触发，并可设置为捕获中断，这些设置是通过 CCR 完成的。

定时器基本操作方法：

- 计算定时器的时钟频率，设置 PR 寄存器进行分频操作；
- 设置比较匹配通道的初值及其工作模式，若是使用捕获功能，则设置捕获方式；
- 若使用定时器的相关中断，则设置 VIC，使能中断；
- 设置 TCR，启动定时器定时器。

定时器计数时钟频率计算如下：

$$\text{计数时钟频率} = \frac{\text{Fpclk}}{\text{N}+1}$$

其中，N 为 PR 的值。

1. 定时器 0 初始化

如程序清单 5.32 为定时器 0 初始化示例，程序设置定时器 0 的时钟不分频，T0MR0 匹配后复位定时器并产生中断标志，定时值设置为 Fpclk/10，即 0.1S 定时值。

程序清单 5.32 定时器 0 初始化示例

```
/*****************************************************************************
* 名称：Time0Init()
* 功能：初始化定时器 0，定时时间为 0.1S，然后启动定时器。
* 入口参数：无
* 出口参数：无
*****************************************************************************/
void  Time0Init(void)
{     T0TC = 0;                          // 定时器设置为 0
      T0PR = 0;                          // 时钟不分频
      T0MCR = 0x03;                      // 设置 T0MR0 匹配后复位 T0TC，并产生中断标志
      T0MR0 = Fpclk/10;                  // 设置 0.1S 匹配值
      T0TCR = 0x01;                      // 启动定时器 0
}
```

2. 读取定器值

如程序清单 5.33 所示为使用定时器进行脉宽(脉冲宽度)测量的示例，脉冲从 P0.0 口输入，程序等待 P0.0 口变为低电平后启动定时器开始测量，当 P0.0 口变为高电平时停止定时器，然后从 T0TC 寄存器读取定时计数值。

程序清单 5.33 用定时器进行脉宽测量示例

```
      T0TC = 0;
      T0PR = 0;
      while((IO0PIN&0x00000001) != 0);     // 等待 P0.0 口变为低电平
      T0TCR = 0x01;                        // 启动定时器 0
      while((IO0PIN&0x00000001) == 0);     // 等待 P0.0 口恢复为高电平
      T0TCR = 0x00;
      time = T0TC;
```

3. 匹配输出

如程序清单 5.34 所示为定时器匹配输出的初始化示例程序，程序设置了 MR1 匹配后复位定时器，并且 MAT0.1 输出电平翻转，这样将会产生占空比为 50%的脉冲频率。

程序清单 5.34 定时器匹配输出初始化示例

```
/*****************************************************************************
* 名称：Time0Init1()
* 功能：初始化定时器 0，设置 MR1 匹配时 MAT0.1 输出取反，然后启动定时器。
* 入口参数：无
* 出口参数：无
*****************************************************************************/
void  Time0Init1(void)
{     T0TC = 0;
      T0PR = 0;
      T0MCR = 0x10;                    // 设置 T0MR1 匹配后复位 T0TC
      T0EMR = 0xC0;                    // T0MR1 匹配后 MAT0.1 输出翻转
      T0MR1 = 5000;                    // 输出频率周期控制
      T0TCR = 0x01;
}
```

4. 定时器捕获

程序清单 5.35 为使用定时器进行捕获的初始化示例程序，先将口线 P0.2 设置为 CAP0.0 功能，使能定时器 0 的捕获通道 0，然后启动定时器 0 运行，当有捕获事件产生时即自动把定时器的当前值装载到 T0CR0 寄存器中。

程序清单 5.35 定时器捕获功能初始化示例

```
      PINSEL0 = 0x20;                  // 设置 P0.2 为 CAP0.0 功能
      T0PR = 0;
      T0CCR = 0x02;                    // 设置 CAP0.0 下降沿捕获
      T0TC = 0;
      T0TCR = 0x01;
```

5.15 脉宽调制器（PWM）

LPC2114/2124/2210/2212/2214 的脉宽调制器建立在标准定时器之上（此定时器是 PWM 专用的，不是定时器 0 或定时器 1），通过匹配功能及一些控制电路来实现 PWM 输出。

5.15.1 特性

- 带可编程 32 位预分频器的 32 位定时器/计数器
- 7 个匹配寄存器，可实现 6 个单边沿控制或 3 个双边沿控制 PWM 输出，或这两种类型的混合输出：
  ——连续操作，可选择在匹配时产生中断
  ——匹配时停止定时器，可选择产生中断
  ——匹配时复位定时器，可选择产生中断
- 支持单边沿控制和双边沿控制的 PWM 输出。单边沿控制 PWM 输出在每个周期开

始时总是为高电平，除非输出保持恒定低电平，如图 5.64 所示(其中 T 表示一个 PWM 周期)。双边沿控制 PWM 输出可在一个周期内的任何位置产生边沿，这样就可以产生正或负脉冲，如图 5.65 所示。

图 5.64 不同占空比的单边沿控制 PWM 输出

图 5.65 双边沿控制 PWM 输出的正负脉冲

- 脉冲周期和宽度可以是任何的定时器计数值。这样可实现灵活的分辨率和重复速率的设定。所有 PWM 输出都以相同的重复率发生。
- 匹配寄存器更新与脉冲输出同步，防止产生错误的脉冲。软件必须在新的匹配值生效之前设置好这些寄存器。
- 如果不使能 PWM 模式，可作为一个标准定时器

5.15.2 引脚描述

表 5.128 汇集了所有与 PWM 相关的引脚。

表 5.128 PWM 引脚汇总

引脚名称	引脚方向	引脚描述
PWM1	输出	PWM 通道 1 输出
PWM2	输出	PWM 通道 2 输出
PWM3	输出	PWM 通道 3 输出
PWM4	输出	PWM 通道 4 输出
PWM5	输出	PWM 通道 5 输出
PWM6	输出	PWM 通道 6 输出

5.15.3 描述

PWM 基于标准的定时器模块并具有其所有特性。不过 LPC2114/2124/2210/2212/2214 只将其 PWM 功能输出到引脚。定时器对外设时钟(pclk)进行计数，定时控制是基于 7 个匹配寄存器，在到达指定的定时值时可选择产生中断或执行其它动作。PWM 功能是一个附加特性，建立在匹配寄存器事件基础之上。

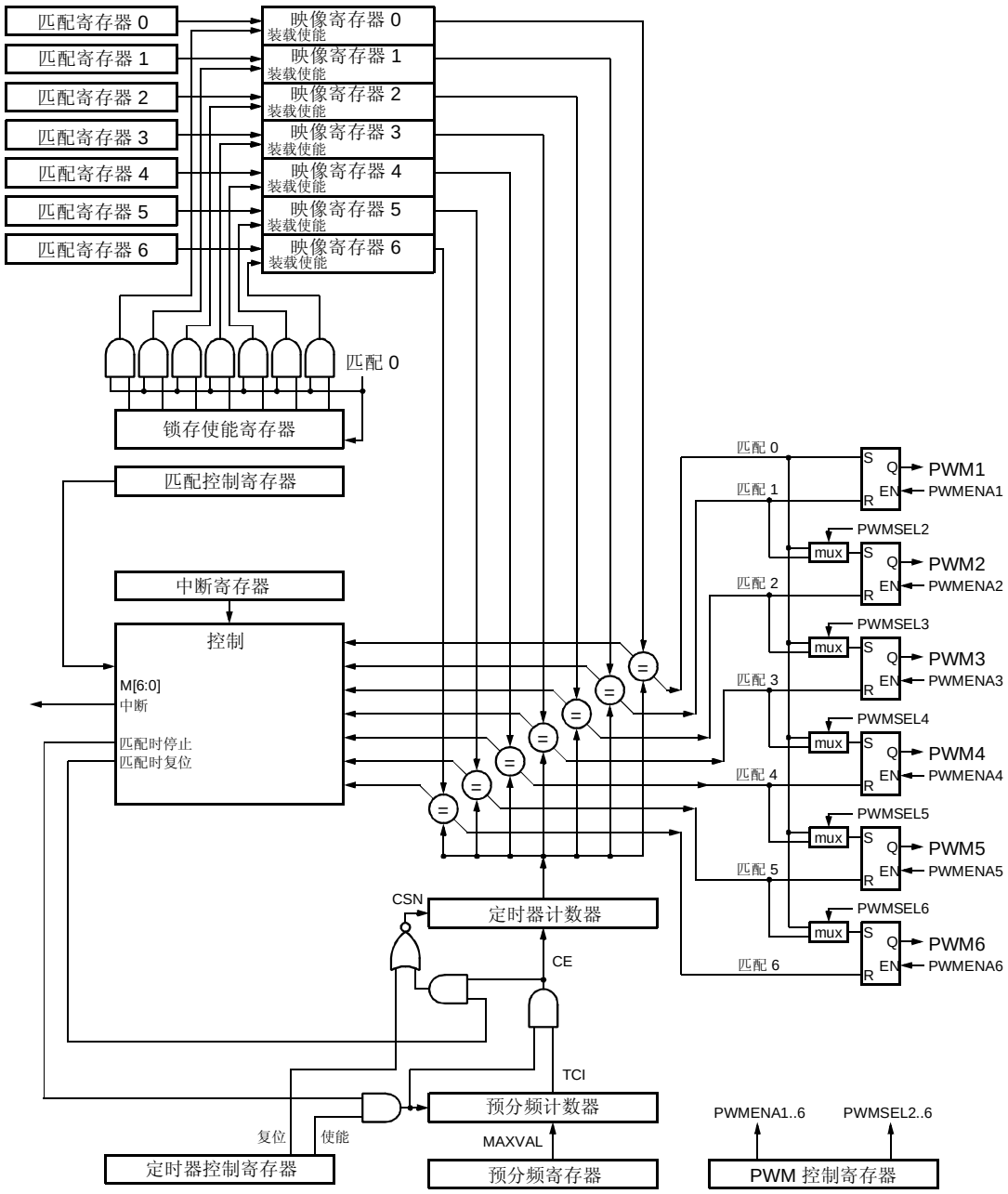
- **单边沿控制 PWM 描述**。两个匹配寄存器可用于提供单边沿控制的 PWM 输出。一个匹配寄存器（PWMMR0）通过匹配时重新设置计数值来控制 PWM 周期。另一个匹配寄存器控制 PWM 边沿的位置。每个额外的单边沿控制 PWM 输出只需要一个匹配寄存器，因为所有 PWM 输出的重复率是相同的，都是使用匹配寄存器 0 来控制的。多个单边沿控制 PWM 输出在每个 PWM 周期的开始，当 PWMMR0(即匹配寄存器 0)发生匹配时，输出都会变为高电平。
- **双边沿控制 PWM 描述。**3 个匹配寄存器可用于提供一个双边沿控制 PWM 输出。也就是说，PWMMR0 匹配寄存器控制 PWM 周期，其它匹配寄存器控制两个 PWM

边沿位置。每个额外的双边沿控制 PWM 输出只需要两个匹配寄存器，因为所有 PWM 输出的重复率是相同的，都是使用匹配寄存器 0 来控制的。

使用双边沿控制 PWM 输出时，指定的匹配寄存器控制输出的上升和下降沿。这样就产生了正脉冲（当上升沿先于下降沿时）和负脉冲（当下降沿先于上升沿时）。独立控制上升和下降沿位置的能力使 PWM 可以应用于更多的领域。例如，多相位电机控制通常需要 3 个非重叠的 PWM 输出，而这 3 个输出的脉宽和位置需要独立进行控制。

5.15.4 结构

图 5.66 所示为 PWM 的方框图。在标准定时器模块上增加的部分位于图的右边和顶端。图 5.66 的 PWM 输出逻辑允许通过 PWMSELn 位选择单边沿或者双边沿控制的 PWM 输出。



注：该图用来解释 PWM 的功能，不是一个具体的设计方案。

图 5.66 PWM 方框图

图 5.67 所示为一个用来说明 PWM 值与波形输出之间关系的例子。表 5.129 所示为不同 PWM 输出的匹配寄存器选项，表中的“置位”表示输出高电平，而“复位”表示输出低电平。LPC2000 系列微控制器支持 N-1 个单边沿 PWM 输出或(N-1)/2 个双边沿 PWM 输出，其中 N 为匹配寄存器的个数，最大值为 7。

图 5.67 所示的波形是单个 PWM 周期，它演示了在下列条件下的 PWM 输出：

- 定时器配置为 PWM 模式
- 匹配寄存器 0 配置为在发生匹配事件时复位定时器/计数器
- 控制位 PWMSEL2 和 PWMSEL4 置位
- 匹配寄存器值如下：

  MR0=100 (PWM 速率)

  MR1=41, MR2=78 (PWM2 输出)

  MR3=53, MR4=27 (PWM4 输出)

  MR5=65 (PWM5 输出)

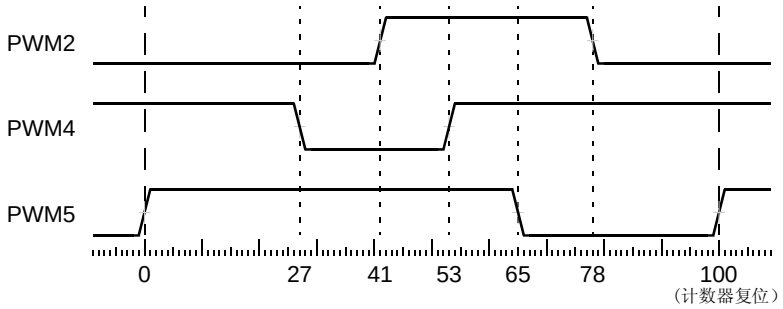


图 5.67 PWM 波形举例

表 5.129 PWM 触发器的置位和复位输入

PWM 通道	单边沿 PWM(PWMSELn=0)		双边沿 PWM(PWMSELn=1)	
	置位	复位	置位	复位
1	匹配 0	匹配 1	匹配 0[1]	匹配 1[1]
2	匹配 0	匹配 2	匹配 1	匹配 2
3	匹配 0	匹配 3	匹配 2[2]	匹配 3[2]
4	匹配 0	匹配 4	匹配 3	匹配 4
5	匹配 0	匹配 5	匹配 4[2]	匹配 5[2]
6	匹配 0	匹配 6	匹配 5	匹配 6

注：

1. 这种情况下与单边沿模式相同，因为匹配 0 是相邻的匹配寄存器。PWM1 不能实现双边沿输出。

2. 通常不建议使用 PWM 通道 3 和通道 5 作为双边沿 PWM 输出，因为这样会减少可用的双边沿 PWM 的个数。使用 PWM2, PWM4 和 PWM6 可得到最多个数的双边沿 PWM 输出。

单边沿控制的 PWM 输出规则

- 所有单边沿控制的 PWM 输出在 PWM 周期开始时都为高电平，除非它们的匹配值等于 0。如图 5.68 所示。
- 每个 PWM 输出在到达其匹配值时都会变为低电平。如果没有发生匹配（即匹配值大于 PWM 周期的值），PWM 输出将一直保持高电平。如图 5.68 所示。

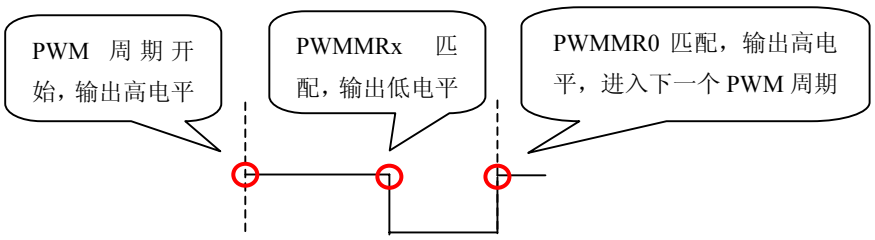


图 5.68 单边沿控制 PWM 规则示意图

双边沿控制的 PWM 输出规则

当一个新的周期将要开始时，使用以下 5 个规则来决定下一个 PMW 输出的值：

- 在一个 PWM 周期结束时（与下一个 PWM 周期的开始重合的时间点），使用下一个 PMW 周期的匹配值，例外见第 3 点规则。如图 5.69 所示。
- 等于 0 或当前 PWM 周期（与匹配通道 0 的值相同）的匹配值等效，例外见第 3 点规则。例如，在 PWM 周期开始时的下降沿请求与 PWM 周期结束时的下降沿请求等效。
- 当匹配值正在改变时，如果有其中一个“旧”匹配值等于 PWM 周期的值且不等于 0，并且新的匹配值不等于 0 或 PWM 速率，那么旧的匹配值将再次被使用。
- 如果同时请求 PWM 输出置位和清零，清零优先。当置位和清零匹配值相同时，或者置位或清零值等于 0 并且其它值等于 PWM 周期的值时(这里所说的“其它值”是指用于 PWM 周期控制的匹配寄存器值)，可能发生这种状况。
- 如果匹配值超出范围（大于 PWM 周期的值），将不会发生匹配事件，匹配通道对输出不起作用。也就是说 PWM 输出将一直保持一种状态，可以为低电平、高电平或是保持输出“无变化”。

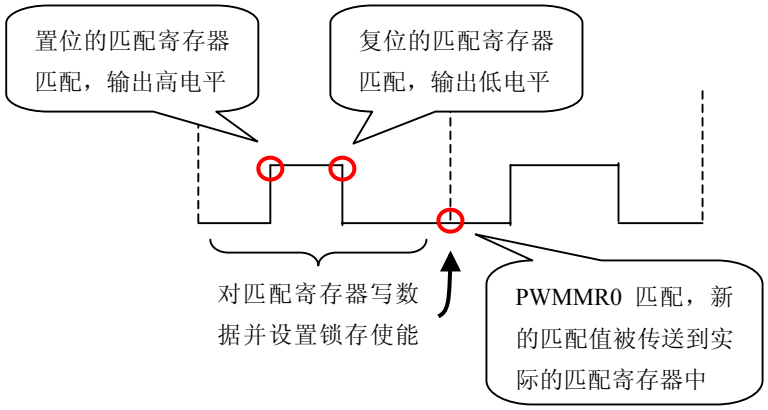


图 5.69 双边沿控制 PWM 规则示意图

5.15.5 寄存器描述

寄存器汇总

PWM 模块包含的寄存器见表 5.130。

表 5.130 PWM 寄存器映射

名称	描述	访问	复位值	地址
PWMIR	PWM 中断寄存器 可以写 IR 来清除中断。可读取 IR 来识别哪个中断源被挂起。	R/W	0	0xE0014000

接上表

名称	描述	访问	复位值	地址
PWMTCR	PWM 定时器控制寄存器 TCR 用于控制定时器计数器功能。定时器计数器可通过 TCR 禁止或复位。	R/W	0	0xE0014004
PWMTC	PWM 定时器计数器 32 位 TC 每经过 PR+1 个 pclk 周期加 1。TC 通过 TCR 进行控制。	R/W	0	0xE0014008
PWMPR	PWM 预分频寄存器 TC 每经过 PR+1 个 pclk 周期加 1。	R/W	0	0xE001400C
PWMPC	PWM 预分频计数器 每当 32 位 PC 的值增加到等于 PR 中保存的值时，TC 加 1。	R/W	0	0xE0014010
PWMMCR	PWM匹配控制寄存器 MCR用于控制在匹配时是否产生中断或复位 TC。	R/W	0	0xE0014014
PWMMR0	PWM 匹配寄存器 0 MR0 可通过 MCR 设定为在匹配时复位 TC，停止 TC 和 PC 和/或产生中断。此外，MR0 和 TC 的匹配将置位所有单边沿模式的 PWM 输出，并置位双边沿模式下的 PWM1 输出。	R/W	0	0xE0014018
PWMMR1	PWM 匹配寄存器 1 MR1 可通过 MCR 设定为在匹配时复位 TC，停止 TC 和 PC 和/或产生中断。此外，MR1 和 TC 的匹配将清零单边沿模式或双边沿模式下的 PWM1，并置位双边沿模式下的 PWM2 输出。	R/W	0	0xE001401C
PWMMR2	PWM 匹配寄存器 2 MR2 可通过 MCR 设定为在匹配时复位 TC，停止 TC 和 PC 和/或产生中断。此外，MR2 和 TC 的匹配将清零单边沿模式或双边沿模式下的 PWM2，并置位双边沿模式下的 PWM3 输出。	R/W	0	0xE0014020
PWMMR3	PWM 匹配寄存器 3 MR3 可通过 MCR 设定为在匹配时复位 TC，停止 TC 和 PC 和/或产生中断。此外，MR3 和 TC 的匹配将清零单边沿模式或双边沿模式下的 PWM3，并置位双边沿模式下的 PWM4 输出。	R/W	0	0xE0014024
PWMMR4	PWM 匹配寄存器 4 MR4 可通过 MCR 设定为在匹配时复位 TC，停止 TC 和 PC 和/或产生中断。此外，MR4 和 TC 的匹配将清零单边沿模式或双边沿模式下的 PWM4，并置位双边沿模式下的 PWM5 输出。	R/W	0	0xE0014040
PWMMR5	PWM 匹配寄存器 5 MR5 可通过 MCR 设定为在匹配时复位 TC，停止 TC 和 PC 和/或产生中断。此外，MR5 和 TC 的匹配将清零单边沿模式或双边沿模式下的 PWM5，并置位双边沿模式下的 PWM6 输出。	R/W	0	0xE0014044
PWMMR6	PWM 匹配寄存器 6 MR6 可通过 MCR 设定为在匹配时复位 TC，停止 TC 和 PC 和/或产生中断。此外，MR6 和 TC 的匹配将清零单边沿模式或双边沿模式下的 PWM6，	R/W	0	0xE0014048
PWMPCR	PWM 控制寄存器 使能 PWM 输出并选择 PWM 通道类型为单边沿或双边沿控制。	R/W	0	0xE001404C
PWMLER	PWM 锁存使能寄存器 使能使用新的 PWM 匹配值。	R/W	0	0xE0014050

PWM 中断寄存器（PWMIR - 0xE0014000）

中断寄存器包含 11 个位（见表 5.131）。其中 7 个位用于匹配中断，4 个位保留将来之

用。如果有中断产生，PWMIR 中的对应位会置位，否则为 0。向对应的 IR 位写入 1 会复位中断。写入 0 无效。

表 5.131 中断寄存器

PWMIR	功能	描述	复位值
0	PWMMR0 中断	PWM 匹配通道 0 的中断标志	0
1	PWMMR1 中断	PWM 匹配通道 1 的中断标志	0
2	PWMMR2 中断	PWM 匹配通道 2 的中断标志	0
3	PWMMR3 中断	PWM 匹配通道 3 的中断标志	0
4	保留	应用程序不能向该位写入 1	0
5	保留	应用程序不能向该位写入 1	0
6	保留	应用程序不能向该位写入 1	0
7	保留	应用程序不能向该位写入 1	0
8	PWMMR4 中断	PWM 匹配通道 4 的中断标志	0
9	PWMMR5 中断	PWM 匹配通道 5 的中断标志	0
10	PWMMR6 中断	PWM 匹配通道 6 的中断标志	0

PWM 定时器控制寄存器（PWMTCR - 0xE0014004）

PWM 定时器控制寄存器（PWMTCR）用于控制 PWM 定时器计数器的操作。每个位的功能见表 5.132。

表 5.132 定时器控制寄存器

PWMTCR	功能	描述	复位值
0	计数器使能	为 1 时，PWM 定时器计数器和 PWM 预分频计数器使能计数。为 0 时，计数器被禁止。	0
1	计数器复位	为 1 时，PWM 定时器计数器和 PWM 预分频计数器在 pclk 的下一个上升沿同步复位。计数器在 TCR 的 bit1 恢复为 0 之前保持复位状态。	0
2	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
3	PWM 使能	为 1 时，PWM 模式使能。PWM 模块将映像寄存器连接到匹配寄存器。只有在 PWMLER 中的相应位置位后，发生的匹配 0 事件才会使程序写入匹配寄存器的值生效。需要注意的是，决定 PWM 周期（PWM 匹配 0）的匹配寄存器必须在使能 PWM 之前设定。否则不会发生使映像寄存器内容生效的匹配事件。	0

PWM 定时器计数器（PWMTC - 0xE0014008）

当预分频计数器到达计数的上限时，32 位定时器计数器 TC 加 1。如果 PWMTC 在到达计数上限之前没有被复位，它将一直计数到 0xFFFFFFFF 然后翻转到 0x00000000。该事件不会产生中断。如果需要，可用匹配寄存器检测溢出。

PWM 预分频寄存器（PWMPR - 0xE001400C）

32 位预分频寄存器指定了预分频寄存器的最大值。

PWM 预分频计数器寄存器（PWMPC - 0xE0014010）

预分频计数器使用某个常量来控制 pclk 的分频，再使之用于 PWM 定时器计数器。这

样可实现控制定时器分辨率和定时器溢出时间之间的关系。预分频计数器每个 pclk 周期加 1。当其到达 PWM 预分频计数器中保存的值时，PWM 定时器计数器加 1，PWM 预分频计数器在下个 pclk 周期复位。这样，当 PWMPR=0 时，PWM TC 每 1 个 pclk 周期加 1，当 PWMPR=1 时，PWMTC 每 2 个 pclk 周期加 1。

PWM 匹配寄存器（PWMMR0 –PWMMR6）

PWM 匹配寄存器值连续与 PWM 定时器计数值相比较。当两个值相等时自动触发相应动作。这些动作包括产生中断，复位 PWM 定时器计数器或停止定时器。所执行的动作由 PWMMCR 寄存器控制。

PWM 匹配控制寄存器（PWMMCR - 0xE0014014）

PWM 匹配控制寄存器用于控制在发生匹配时所执行的操作。每个位的功能见表 5.133。

表 5.133 匹配控制寄存器

PWMMCR	功能	描述	复位值
0	中断 (PWMMR0)	为 1 时，PWMMR0 与 PWMTC 值的匹配将产生中断。为 0 时，该中断被禁止。	0
1	复位 (PWMMR0)	为 1 时，PWMMR0 与 PWMTC 值的匹配将使 PWMTC 复位。为 0 时，该特性被禁止。	0
2	停止 (PWMMR0)	为 1 时，PWMMR0 与 PWMTC 值的匹配将使 PWMTC 和 PWMPC 停止并使 PWMTCR[0]复位为 0。为 0 时，该特性被禁止。	0
3	中断 (PWMMR1)	为 1 时，PWMMR1 与 PWMTC 值的匹配将产生中断。为 0 时，该中断被禁止。	0
4	复位 (PWMMR1)	为 1 时，PWMMR1 与 PWMTC 值的匹配将使 PWMTC 复位。为 0 时，该特性被禁止。	0
5	停止 (PWMMR1)	为 1 时，PWMMR1 与 PWMTC 值的匹配将使 PWMTC 和 PWMPC 停止并使 PWMTCR[0]复位为 0。为 0 时，该特性被禁止。	0
6	中断 (PWMMR2)	为 1 时，PWMMR2 与 PWMTC 值的匹配将产生中断。为 0 时，该中断被禁止。	0
7	复位 (PWMMR2)	为 1 时，PWMMR2 与 PWMTC 值的匹配将使 PWMTC 复位。为 0 时，该特性被禁止。	0
8	停止 (PWMMR2)	为 1 时，PWMMR2 与 PWMTC 值的匹配将使 PWMTC 和 PWMPC 停止并使 PWMTCR[0]复位为 0。为 0 时，该特性被禁止。	0
9	中断 (PWMMR3)	为 1 时，PWMMR3 与 PWMTC 值的匹配将产生中断。为 0 时，该中断被禁止。	0
10	复位 (PWMMR3)	为 1 时，PWMMR3 与 PWMTC 值的匹配将使 PWMTC 复位。为 0 时，该特性被禁止。	0
11	停止 (PWMMR3)	为 1 时，PWMMR3 与 PWMTC 值的匹配将使 PWMTC 和 PWMPC 停止并使 PWMTCR[0]复位为 0。为 0 时，该特性被禁止。	0
12	中断 (PWMMR4)	为 1 时，PWMMR4 与 PWMTC 值的匹配将产生中断。为 0 时，该中断被禁止。	0
13	复位 (PWMMR4)	为 1 时，PWMMR4 与 PWMTC 值的匹配将使 PWMTC 复位。为 0 时，该特性被禁止。	0

接上表

PWMMCR	功能	描述	复位值
14	停止 (PWMMR4)	为 1 时，PWMMR4 与 PWMTC 值的匹配将使 PWMTC 和 PWMPC 停止并使 PWMTCR[0]复位为 0。为 0 时，该特性被禁止。	0
15	中断 (PWMMR5)	为 1 时，PWMMR5 与 PWMTC 值的匹配将产生中断。为 0 时，该中断被禁止。	0
16	复位 (PWMMR5)	为 1 时，PWMMR5 与 PWMTC 值的匹配将使 PWMTC 复位。为 0 时，该特性被禁止。	0
17	停止 (PWMMR5)	为 1 时，PWMMR5 与 PWMTC 值的匹配将使 PWMTC 和 PWMPC 停止并使 PWMTCR[0]复位为 0。为 0 时，该特性被禁止。	0
18	中断 (PWMMR6)	为 1 时，PWMMR6 与 PWMTC 值的匹配将产生中断。为 0 时，该中断被禁止。	0
19	复位 (PWMMR6)	为 1 时，PWMMR6 与 PWMTC 值的匹配将使 PWMTC 复位。为 0 时，该特性被禁止。	0
20	停止 (PWMMR6)	为 1 时，PWMMR6 与 PWMTC 值的匹配将使 PWMTC 和 PWMPC 停止并使 PWMTCR[0]复位为 0。为 0 时，该特性被禁止。	0

PWM 控制寄存器（PWMPCR - 0xE001404C）

PWM 控制寄存器用于使能并选择每个 PMW 通道的类型。每个位的功能详见表 5.134。

表 5.134 PWM 控制寄存器

PWMPCR	功能	描述	复位值
1:0	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
2	PWMSEL2	为 0 时，PWM2 选择单边沿控制模式；为 1 时，选择双边沿控制模式。	0
3	PWMSEL3	为 0 时，PWM3 选择单边沿控制模式；为 1 时，选择双边沿控制模式。	0
4	PWMSEL4	为 0 时，PWM4 选择单边沿控制模式；为 1 时，选择双边沿控制模式。	0
5	PWMSEL5	为 0 时，PWM5 选择单边沿控制模式；为 1 时，选择双边沿控制模式。	0
6	PWMSEL6	为 0 时，PWM6 选择单边沿控制模式；为 1 时，选择双边沿控制模式。	0
8:7	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
9	PWMENA1	为 1 时，使能 PWM1 输出；为 0 时，禁止 PWM1 输出。	0
10	PWMENA2	为 1 时，使能 PWM2 输出；为 0 时，禁止 PWM2 输出。	0
11	PWMENA3	为 1 时，使能 PWM3 输出；为 0 时，禁止 PWM3 输出。	0
12	PWMENA4	为 1 时，使能 PWM4 输出；为 0 时，禁止 PWM4 输出。	0
13	PWMENA5	为 1 时，使能 PWM5 输出；为 0 时，禁止 PWM5 输出。	0
14	PWMENA6	为 1 时，使能 PWM6 输出；为 0 时，禁止 PWM6 输出。	0
15	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

PWM 锁存使能寄存器（PWMLER - 0xE0014050）

当 PWM 匹配寄存器用于产生 PWM 时，PWM 锁存使能寄存器用于控制 PWM 匹配寄存器的更新。当定时器处于 PWM 模式时如果软件对 PWM 匹配寄存器执行写操作，写入的值将保存在一个映像寄存器中。当 PWM 匹配 0 事件发生时（在 PWM 模式下，通常也会复位定时器），如果对应的锁存使能寄存器位已经置位，那么映像寄存器的内容将传送到实际的匹配寄存器中。此时，新的值将生效并决定下一个 PWM 周期。当发生新值传送时，LER

中的所有位都自动清零。在 PWMLER 中相应位置位和 PWM 匹配 0 事件发生之前，任何写入 PWM 匹配寄存器的值都不会影响 PWM 操作。

例如，当 PWM2 配置为双边沿操作并处于运行中时，改变定时的典型事件顺序如下：

- 将新值写入 PWM 匹配 1 寄存器；
- 将新值写入 PWM 匹配 2 寄存器；
- 写 PWMLER，同时置位 bit1 和 bit2；
- 更改的值将在下一次定时器复位时（当 PWM 匹配 0 事件发生时）生效。

写两个 PWM 匹配寄存器的顺序并不重要，因为在写 PWMLER 之前，写入的新匹配值都无效。这样就确保了两个值同时生效。如果使用单个值，也可用同样的方法更改。

PWMLER 中所有位的功能如表 5.135 所示。

表 5.135 PWM 锁存使能寄存器

PWMLER	功能	描述	复位值
0	使能 PWM 匹配 0 锁存	将该位置位允许最后写入 PWM 匹配 0 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效。见 PWM 匹配控制寄存器（PWMMCR）的描述。	0
1	使能 PWM 匹配 1 锁存	将该位置位允许最后写入 PWM 匹配 1 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效。见 PWM 匹配控制寄存器（PWMMCR）的描述。	0
2	使能 PWM 匹配 2 锁存	将该位置位允许最后写入 PWM 匹配 2 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效。见 PWM 匹配控制寄存器（PWMMCR）的描述。	0
3	使能 PWM 匹配 3 锁存	将该位置位允许最后写入 PWM 匹配 3 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效。见 PWM 匹配控制寄存器（PWMMCR）的描述。	0
4	使能 PWM 匹配 4 锁存	将该位置位允许最后写入 PWM 匹配 4 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效。见 PWM 匹配控制寄存器（PWMMCR）的描述。	0
5	使能 PWM 匹配 5 锁存	将该位置位允许最后写入 PWM 匹配 5 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效。见 PWM 匹配控制寄存器（PWMMCR）的描述。	0
6	使能 PWM 匹配 6 锁存	将该位置位允许最后写入 PWM 匹配 6 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效。见 PWM 匹配控制寄存器（PWMMCR）的描述。	0
7	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

5.15.6 使用示例

LPC2114/2124/2210/2212/2214 的 PWM 功能是建立在标准的定时器之上，它同样具有 32 位定时器及预分频控制电路，7 个匹配寄存器，可实现 6 个单边 PWM 或 3 个双边 PWM 输出，也可以采用这两种类型的混合输出。具有匹配中断、匹配 PWMTC 复、匹配 PWMTC 停止功能，如果不使能 PWM 模式，可作为一个标准的定时器。PWM 的基本寄存器功能框图如图 5.70 所示。

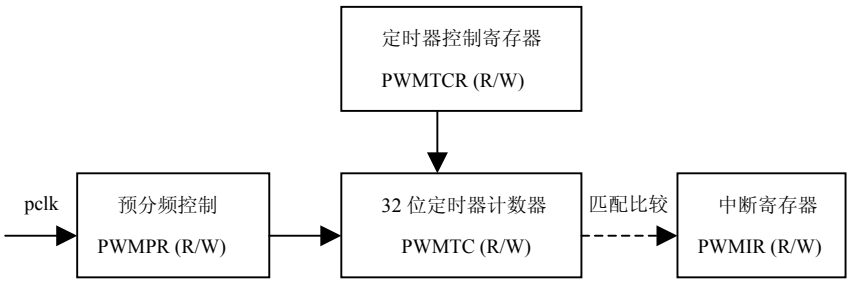


图 5.70 PWM 的基本寄存器功能框图

如图 5.70 所示，32 位定时器 PWMTC 的计数频率由 pclk 经过 PWMPR 进行分频控制得到，而定时器的启动/停止、计数复位由 PWMTCR 控制，当有比较匹配事件发生时，PWMIR 会设置相关中断标志(因为不是定时器溢出而产生中断，所以上图采用虚线连接)，若已打开中断允许(VIC)则会产生中断。当然，预分频控制器 PWMPR 只是控制分频数，而其对应的分频计数器是 PWMPC，但用户不需要操作 PWMPC 寄存器。

如图 5.71 所示，定时器比较匹配由控制寄存器 PWMMCR 进行匹配操作设置，而 PWMMR0～6 则为 7 路比较匹配通道的比较值寄存器。当比较匹配时，将会按照 PWMMCR 设置的方法产生中断或复位 PWMTC 等，而且 PWMPCR 可以控制单边/双边 PWM 输出，允许/不允许 PWM 输出。另外，为了确保对 PWMMR0～6 的比较值进行修改过程中不影响 PWM 输出，使用了一个 PWMLER 锁存使能寄存器，当要修改 MR0～6 的比较值时，只有控制 PWMLER 的对应位置位，在匹配 0 事件发生后此值才会生效。

PWM 基本操作方法：

- 连接 PWM 功能引脚输出，即设置 PINSEL0、PINSEL1；
- 设置 PWM 定时器的时钟分频值 (PWMPR)，得到所要的定时器时钟；
- 设置比较匹配控制(PWMMCR)，并设置相应比较值(PWMMRx)；
- 设置 PWM 输出方式并允许 PWM 输出(PWMPCR)及锁存使能控制(PWMLER)；
- 设置 PWMTCR，启动定时器，使能 PWM；
- 运行过程中要更改比较值时，更改之后要设置锁存使能。

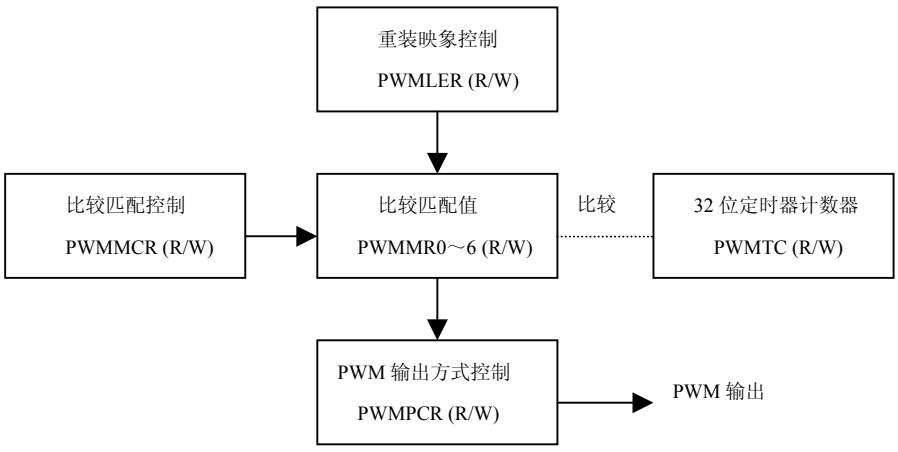


图 5.71 PWM 的比较匹配寄存器功能框图

使用双边沿 PWM 输出时，建议使用 PWM2、PWM4、PWM6；使用单边 PWM 输出时，在 PWM 周期开始时为高电平，匹配后为低电平，使用 PWMMR0 作为 PWM 周期控制，PWMMRx 作为占空比控制。

1. PWM1 单边沿控制

设置 PWM1 输出。设置 PWM1 为单边沿控制的 PWM 输出，PWM 周期由匹配寄存器 0 控制，当匹配寄存器 0 匹配时，PWM1 输出高电平，PWM 占空比由匹配寄存器 1 控制，当匹配寄存器 1 匹配时，PWM1 输出低电平，初始化程序如程序清单 5.36 所示。

程序清单 5.36 PWM1 单边沿控制的 PWM 输出

```
PWMPCR = 0x200;                 // 使能 PWM1，模式为单边沿控制
PWMMCR = 0x02;                  // 当 PWMMR0 匹配时复位 PWM 定时器
PWMMR0 = 0x10000;               // 设置 PWM 周期
PWMMR1 = 0x6000;                // 设置 PWM 占空比
PWMLER = 0x03;                  // 使能 PWM 匹配 0、1 锁存
PWMTCR = 0x09;                  // PWM 使能，启动 PWM 定时器
```

2. PWM2 双边沿控制

设置 PWM2 输出双边沿控制 PWM。进行双边沿控制 PWM 输出时，使用匹配寄存器 1 的匹配控制 PWM2 输出高电平，而匹配寄存器 2 自身匹配控制 PWM2 输出低电平，PWM 周期由匹配寄存器 0 控制，初始化设置如程序清单 5.37 所示。

程序清单 5.37 PWM2 双边沿控制的 PWM 输出

```
PWMPCR = 0x404;                 // 使能 PWM2，模式为双边沿控制
PWMMCR = 0x02;
PWMMR0 = 0x10000;
PWMMR1 = 0x2000;                // PWM 高电平匹配值
PWMMR2 = 0x7000;                // PWM 低电平匹配值
PWMLER = 0x07;
PWMTCR = 0x09;
```

5.16 A/D 转换器

5.16.1 特性

- 10 位逐次逼近式模数转换器
- 4 个（LPC2114/2124）或 8 个（LPC2210/2212/2214）引脚复用为 A/D 输入脚
- 测量范围：0～3.3V
- 10 位转换时间>=2.44us
- 一路或多路输入的 Burst 转换模式
- 可选择由输入引脚的跳变或定时器的匹配信号触发转换
- 具有掉电模式

5.16.2 描述

A/D 转换器的基本时钟由 VPB 时钟提供。可编程分频器可将时钟调整至逐步逼近转换所需的 4.5MHz（最大）。10 位精度要求的转换需要 11 个 A/D 转转换时钟。

5.16.3 引脚描述

A/D 引脚描述见表 5.136。A/D 转换器的参考电压来自 V_{3A} 和 V_{SSA} 引脚。

表 5.136 A/D 引脚描述

引脚名称	类型	引脚描述
AIN7 ～ AIN0	输入	**模拟输入** A/D 转换器单元可测量 8 个输入信号的电压（但 64 脚封装的 LPC2100 系列微控制器中模拟输入引脚只有 AIN3～AIN0）。注意：这些模拟输入总是连接到引脚上，即使通过 PINSELn(n 为 1 或 2)寄存器将它们设定为其它功能引脚。通过将这些引脚设置为 GPIO 口输出来实现 A/D 转换器的简单自检。
V_{3A},V_{SSA}	电源	**模拟电源和地** 它们分别与标称为 V_3 和 V_{SSD} 的电压相同，但为了降低噪声和出错几率，两者应当隔离。转换器单元的 VrefP 和 VrefN 信号在内部与这两个电源信号相连。

5.16.4 寄存器描述

寄存器汇总

A/D 转换器的基址是 0xE0034000。A/D 转换器包含 2 个寄存器，见表 5.137。寄存器功能框图见图 5.72。

表 5.137 A/D 寄存器

名称	描述	访问	复位值	地址
ADCR	A/D 控制寄存器。A/D 转换开始前，必须写入 ADCR 寄存器来选择工作模式。	读/写	0x0000 0001	0xE003 4000
ADDR	A/D 数据寄存器。该寄存器包含 ADC 的 DONE 标志位和 10 位的转换结果（当 DONE 位为 1 时，转换结果才是有效的）。	读/写	NA	0xE003 4004

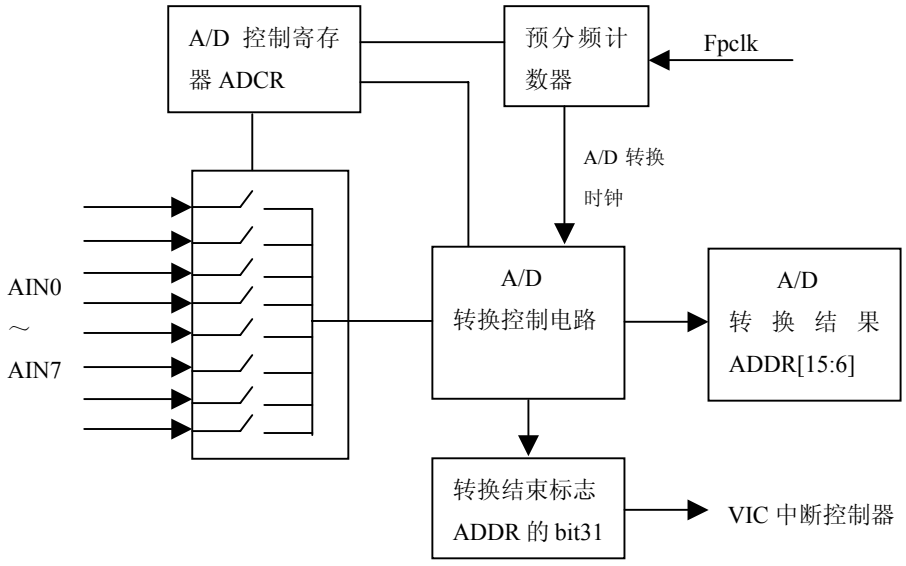


图 5.72 A/D 模块的寄存器功能框图

A/D 控制寄存器（ADCR – 0xE0034000）

ADCR 寄存器描述见表 5.138。

表 5.138 A/D 控制寄存器

ADCR	名称	描述	复位值
7:0	SEL	从 AIN3～AIN0(LPC2114/2124)或 AIN7～AIN0(LPC2212/2214)中选择采样和转换输入脚。在 64 脚封装的 LPC2114/2124 中只有 bit3～bit0 可置位。软件控制模式下，只有一位可被置位。硬件扫描模式下，SEL 可为 1～8 中的任何一个值(在 64 脚封装的 LPC2114/2124 中 SEL 从 1～4 中取值)。SEL 为零时等效于为 0x01。	0x01
15:8	CLKDIV	将 VPB 时钟（PLCK）进行（CLKDIV 的值+1）分频得到 A/D 转换时钟，该时钟必须小于或等于 4.5MHz。典型地，软件将 CLKDIV 编程为最小值来得到 4.5MHz 或稍低于 4.5MHz 的时钟，但某些情况下（例如测量高阻抗的模拟信号）可能需要更低的时钟。	0
16	BURST	如果该位为 0，转换由软件控制，需要 11 个时钟方能完成。如果该位为 1，A/D 转换器以 CLKS 字段选择的速率重复执行转换，并从 SEL 字段中为 1 的位对应的引脚开始扫描。A/D 转换器起动后的第一次转换的是 SEL 字段中为 1 的位中的最低有效位对应的模拟输入，然后是为 1 的更高有效位对应的模拟输入（如果可用）。重复转换通过清零该位终止，但该位被清零时并不会中止正在进行的转换。	0
19:17	CLKS	该字段用来选择 Burst 模式下每次转换使用的时钟数和所得 ADDR 转换结果的 LS 位中可确保精度的位的数目，CLKS 可在 11 个时钟（10 位）～4 个时钟（3 位）之间选择：000＝11 个时钟/10 位，001＝10 个时钟/9 位，…111＝4 个时钟/3 位。	000
21	PDN	1：A/D 转换器处于正常工作模式。 0：A/D 转换器处于掉电模式。	0
23:22	TEST1:0	这些位用于器件测试。00＝正常模式，01＝数字测试模式，10＝DAC 测试模式，11＝一次转换测试模式。	0
26:24	START	当 BURST 为 0 时，这些位控制着 A/D 转换是否启动和何时启动： 000：不启动（PDN 清零时使用该值） 001：立即启动转换 010：当 ADCR 寄存器 bit27 选择的边沿出现在 P0.16/EINT0/ MAT0.2/CAP0.2 脚时启动转换 011：当 ADCR 寄存器 bit27 选择的边沿出现在 P0.22/CAP0.0/ MAT0.0 脚时启动转换 *注意：START 选择 100-111 时 MAT 信号不必输出到引脚上* 100：当 ADCR 寄存器 bit27 选择的边沿在 MAT0.1 出现时启动转换 101：当 ADCR 寄存器 bit27 选择的边沿在 MAT0.3 出现时启动转换 110：当 ADCR 寄存器 bit27 选择的边沿在 MAT1.0 出现时启动转换 111：当 ADCR 寄存器 bit27 选择的边沿在 MAT1.1 出现时启动转换	000
27	EDGE	该位只有在 START 字段为 010～111 时有效。 0：在所选 CAP/MAT 信号的下降沿启动转换 1：在所选 CAP/MAT 信号的上升沿启动转换	0

A/D 数据寄存器（ADDR – 0xE0034004）

ADDR 寄存器描述见表 5.139。其中 ADDR[15:6]为 10 位的 A/D 转换结果，bit15 为最

高位。

表 5.139 A/D 数据寄存器

ADDR	名称	描述	复位值
31	DONE	A/D 转换完成标志位，当 A/D 转换结束时该位置位。该位在 ADDR 被读出和 ADCR 被写入时清零。如果 ADCR 在转换过程中被写入，该位置位，并启动一次新的转换。	0
30	OVERUN	Burst 模式下，如果在转换产生 LS 位的结果前一个或多个转换结果被丢失和覆盖，该位置位。该位通过读 ADDR 寄存器清零。	0
29:27		这些位读出为 0。它们用于未来 CHN 字段的扩展，使之兼容包含更多通道的转换器。	0
26:24	CHN	这些位包含的是 LS 位的转换通道。	X
23:16		这些位读出时为 0。它们允许连续 A/D 值的累加，而不需要使用与门屏蔽处理，使得至少有 256 个值不会溢出到 CHN 字段。	0
15:6	V/VddA	当 DONE 为 1 时，该字段包含一个二进制数，用来代表 SEL 字段选中的 Ain 脚的电压。该字段根据 VddA 脚上的电压对 Ain 脚的电压进行划分。该字段为 0 表明 Ain 脚的电压小于，等于或接近于 VssA；该字段为 0x3FF 表明 Ain 脚的电压接近于，等于或大于 VddA。	X
5:0		这些位读出时为 0。专门用于未来的扩展和功能更强大的 A/D 转换器。	0

5.16.5 操作

硬件触发转换

如果 ADCR 的 BURST 位为 0 且 START 字段的值包含在 010-111 之内，当所选引脚或定时器匹配信号发生跳变时 A/D 转换器启动一次转换。即是可以选择在 4 个匹配信号中任何一个的指定边沿启动转换，或者在 2 个捕获/匹配引脚中任何一个的指定边沿启动转换。

时钟产生

用于产生 A/D 转换时钟的分频器在 A/D 转换器空闲时保持复位状态，以便在 ADCR 的 START 字段被写入 001 或所选引脚或匹配信号触发时可以立刻启动采样时钟。这个特性可以节省功率，尤其适用于 A/D 转换不频繁的场合。

中断

当 DONE 位为 1 时，将对向量中断控制器（VIC）发送中断请求。通过软件设置 VIC 中 A/D 转换器的中断使能位来控制是否产生中断。DONE 在读 ADDR 操作时清零。

精度和引脚设置

当 A/D 转换器用来测量 Ain 脚的电压时，并不理会引脚在引脚选择寄存器中的设置，但是通过选择 Ain 功能(即禁能引脚的数字功能)可以提高转换精度。

5.16.6 使用示例

使用 ADC 模块时，先要将测量通道引脚设置为 AINx 功能，然后通过 ADCR 寄存器设置 ADC 的工作模式，ADC 转换通道，转换时钟(CLKDIV 时钟分频值)，并启动 ADC 转换。可以通过查询或中断的方式等待 ADC 转换完毕，转换数据保存在 ADDR 寄存器中。

ADC 转换时钟分频值计算：

$$CLKDIV = \frac{Fpclk}{Fadclk} - 1$$

其中 Fadclk 为所要设置的 ADC 时钟，其值不能大于 4.5MHz。

初始化 ADC 模块

如程序清单 5.38 所示为使用 AIN0 进行 10 位 ADC 转换的初始化程序，转换时钟设置为 1MHz。

程序清单 5.38 AIN0 初始化示例

```
PINSEL1 = 0x00400000;                       // 设置 P0.27 为 AIN0 功能
ADCR =   (1 << 0)                       |   // SEL = 1 ，选择通道 0
         ((Fpclk / 1000000 - 1) << 8)   |   // CLKDIV = Fpclk / 1000000 - 1 ，即转换时钟为 1MHz
         (0 << 16)                      |   // BURST = 0 ，软件控制转换操作
         (0 << 17)                      |   // CLKS = 0 ，使用 11clock 转换
         (1 << 21)                      |   // PDN = 1 ， 正常工作模式(非掉电转换模式)
         (0 << 22)                      |   // TEST1:0 = 00 ，正常工作模式(非测试模式)
         (1 << 24)                      |   // START = 1 ，直接启动 ADC 转换
         (0 << 27);                         // EDGE = 0 (CAP/MAT 引脚下降沿触发 ADC 转换)
```

5.17 实时时钟

5.17.1 特性

- 带日历和时钟功能
- 超低功耗设计，支持电池供电系统
- 提供秒、分、小时、日、月、年和星期
- 可编程基准时钟分频器允许调节 RTC 以适应不同的晶振频率

5.17.2 描述

实时时钟(RTC)提供一套计数器在系统工作时对时间进行测量。RTC 消耗的功率非常低，这使其适合于由电池供电的，CPU 不连续工作(空闲模式)的系统。

说明，由于 LPC2100/LPC2200 系列微控制器的 RTC 是没有独立的时钟源，使用的时钟频率是通过对 F_{pclk} 分频得到，所以 CPU 不能进入掉电模式。

5.17.3 结构

RTC 结构图见图 5.73。

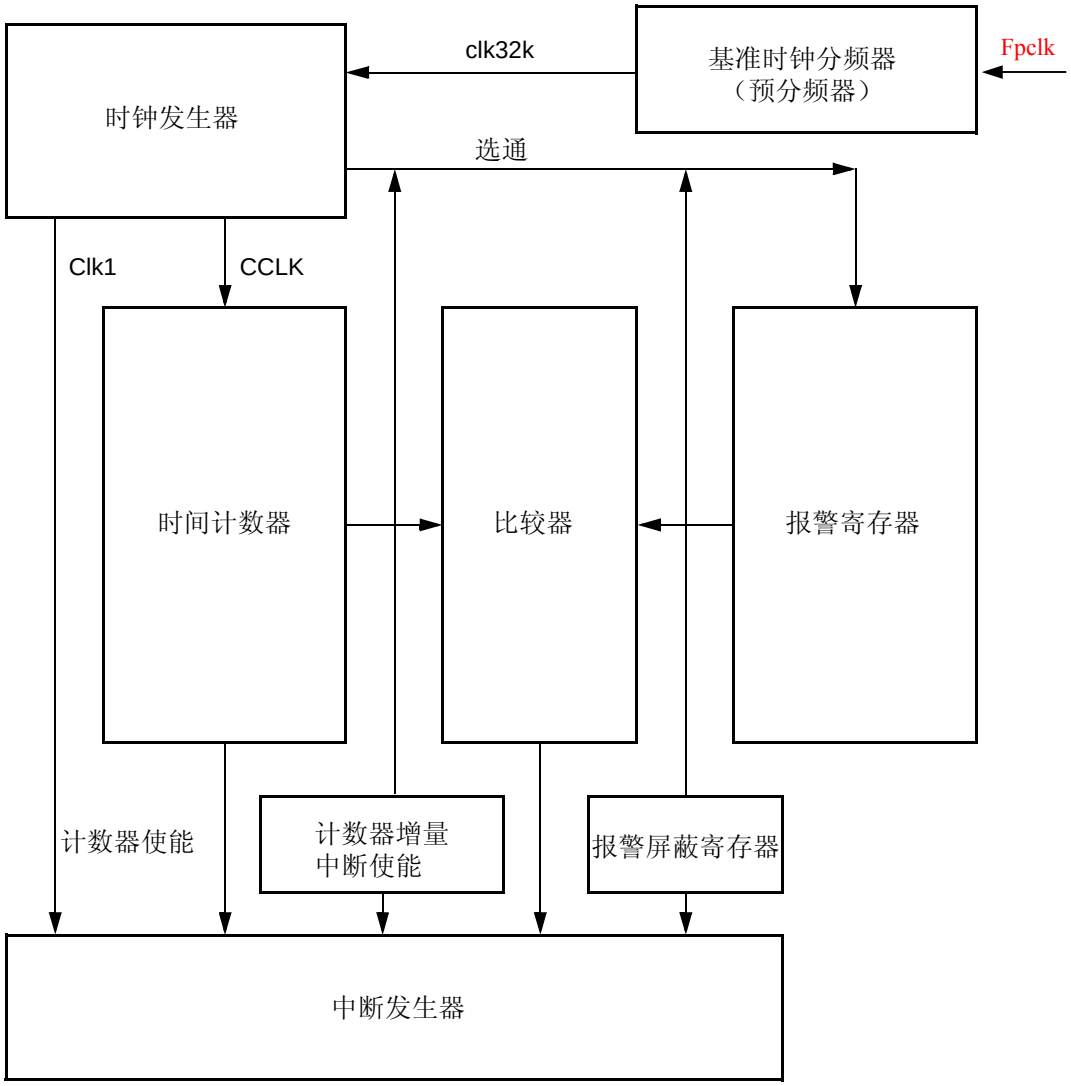


图 5.73 RTC 方框图

5.17.4 RTC 中断

RTC 中断的产生由中断位置寄存器(ILR)、计数器递增中断寄存器(CIIR)、报警寄存器和报警屏蔽寄存器(AMR) 控制。ILR 可单独标志 CIIR 和 AMR 中断(ILR 寄存器实际就是一个中断标志寄存器)。

- CIIR 中的每个位都对应一个时间计数器，比如分、秒计数器。如果 CIIR 使能某一个特定的计数器，那么该计数器的值每增加一次就产生一个中断。增量中断控制原理示意图如图 5.74 所示。
- 报警寄存器允许用户设定产生中断的日期和/或时间。AMR 提供一个屏蔽报警比较的机制，如果所有非屏蔽报警寄存器与它们对应的时间计数器的值相匹配时，则会产生中断。报警中断控制原理示意图如图 5.75 所示。

图 5.74 增量中断控制原理示意图

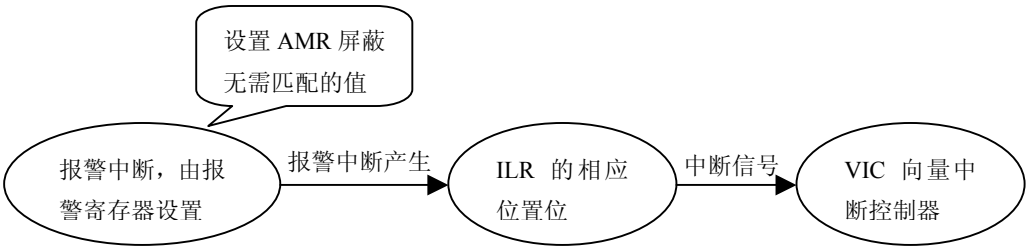


图 5.75 报警中断控制原理示意图

5.17.5 闰年计算

RTC 执行一个简单的位比较，看年计数器的最低两位(即 YEAR[1:0])是否为 0。如果为 0，那么 RTC 认为这一年为闰年。RTC 认为所有能被 4 整除(年计数器的最低两位为 0 时，一定被 4 整除)的年份都为闰年。这个算法从 1901 年到 2099 年都是准确的，但在 2100 年出错，2100 年并不是闰年。闰年对 RTC 的影响只是改变 2 月份的长度、日期（月）和年的计数值。

5.17.6 寄存器描述

寄存器汇总

RTC 包含了许多寄存器。地址空间按照功能分成 4 个部分。前 8 个地址为混合寄存器组。第二部分的 8 个地址为定时器计数器组，第三部分的 8 个地址为报警寄存器组。最后一部分为基准时钟分频器。

实时时钟模块所包含的寄存器见表 5.140，每一部分的寄存器组均用双横线隔开。

表 5.140 实时时钟寄存器映射

名称	规格	描述	访问	复位值	地址
ILR	2	中断位置寄存器	R/W	*	0xE0024000
CTC	15	时钟节拍计数器	RO	*	0xE0024004
CCR	4	时钟控制寄存器	R/W	*	0xE0024008
CIIR	8	计数器递增中断寄存器	R/W	*	0xE002400C
AMR	8	报警屏蔽寄存器	R/W	*	0xE0024010
CTIME0	(32)	完整时间寄存器 0	RO	*	0xE0024014
CTIME1	(32)	完整时间寄存器 1	RO	*	0xE0024018
CTIME2	(32)	完整时间寄存器 2	RO	*	0xE002401C
SEC	6	秒寄存器	R/W	*	0xE0024020
MIN	6	分寄存器	R/W	*	0xE0024024
HOUR	5	小时寄存器	R/W	*	0xE0024028
DOM	5	日期（月）寄存器	R/W	*	0xE002402C
DOW	3	星期寄存器	R/W	*	0xE0024030
DOY	9	日期（年）寄存器	R/W	*	0xE0024034
MONTH	4	月寄存器	R/W	*	0xE0024038
YEAR	12	年寄存器	R/W	*	0xE002403C
ALSEC	6	秒报警值	R/W	*	0xE0024060
ALMIN	6	分报警值	R/W	*	0xE0024064

接上表

名称	规格	描述	访问	复位值	地址
ALHOUR	5	小时报警值	R/W	*	0xE0024068
ALDOM	5	日期（月）报警值	R/W	*	0xE002406C
ALDOW	3	星期报警值	R/W	*	0xE0024070
ALDOY	9	日期（年）报警值	R/W	*	0xE0024074
ALMON	4	月报警值	R/W	*	0xE0024078
ALYEAR	12	年报警值	R/W	*	0xE002407C
PREINT	13	预分频值，整数部分	R/W	0	0xE0024080
PREFRAC	15	预分频值，小数部分	R/W	0	0xE0024084

* RTC 当中除预分频器部分之外的其它寄存器都不受器件复位的影响。如果 RTC 使能，这些寄存器必须通过软件来初始化。

5.17.7 混合寄存器组

表 5.141 所示为混合寄存器组的 8 个寄存器。

表 5.141 混合寄存器

地址	名称	规格	描述	访问
0xE0024000	ILR	2	中断位置寄存器。读出的该位置寄存器的值指示了中断源。向寄存器的一个位写入 1 来清除相应的中断。	R/W
0xE0024004	CTC	15	时钟节拍计数器。该寄存器的值来自时钟分频器。	RO
0xE0024008	CCR	4	时钟控制寄存器。控制时钟分频器的功能。	R/W
0xE002400C	CIIR	8	计数器递增中断寄存器。当计数器递增时，选择一个计数器产生中断。	R/W
0xE0024010	AMR	8	报警屏蔽寄存器。控制报警寄存器的屏蔽。	R/W
0xE0024014	CTIME0	32	完整时间寄存器 0	RO
0xE0024018	CTIME1	32	完整时间寄存器 1	RO
0xE002401C	CTIME2	32	完整时间寄存器 2	RO

中断位置（ILR - 0xE0024000）

中断位置寄存器是一个 2 位的寄存器，它指出哪些模块产生中断(ILR 寄存器实际就是一个中断标志寄存器)。向一个位写入 1 会清除相应的中断，写入 0 无效。这样程序员可以读取该寄存器并将读出的值回写到寄存器中清除检测到的中断。

ILR 寄存器描述见表 5.142。

表 5.142 中断位置寄存器

ILR	功能	描述
0	RTCCIF	为 1 时，计数器增量中断模块产生中断。向该位写入 1 清除计数器增量中断。
1	RTCALF	为 1 时，报警寄存器产生中断。向该位写入 1 清除报警中断。

时钟节拍计数器（CTC - 0xE0024004）

时钟节拍计数器是用于产生秒的时钟节拍计数，这是一个只读寄存器，但它可通过时钟控制寄存器（CCR）复位为 0。

CTC 寄存器描述见表 5.143。

表 5.143 时钟节拍计数器

CTC	功能	描述
0	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。
15:1	时钟节拍计数器	位于秒计数器之前，CTC 每秒计数 32768 个时钟。由于 RTC 预分频器的关系，这 32768 个时间增量的长度可能并不全部相同。详见基准时钟分频器（预分频器）。

时钟控制寄存器（CCR - 0xE0024008）

时钟控制寄存器是一个 4 位的寄存器，它用于控制时钟分频电路的操作，包括启动 RTC 和复位时钟节拍计数器 CTC 等。

CCR 寄存器描述见表 5.144。

表 5.144 时钟控制寄存器

CCR	功能	描述
0	CLKEN	时钟使能 当该位为 1 时，时间计数器使能。为 0 时，时间计数器都被禁止，这时可对其进行初始化。
1	CTCRST	CTC 复位 为 1 时，时钟节拍计数器复位。在 CCR 的 bit1 变为 0 之前，它将一直保持复位状态。
3:2	CTTEST	测试使能 在正常操作中，这些位应当全为 0。

计数器增量中断寄存器（CIIR - 0xE002400C）

计数器增量中断寄存器（CIIR）可使计数器每次增加时产生一次中断，比如设置秒增加中断为 1，则每秒钟均产生一次中断。在中断位置寄存器的 bit0 写入 1 之前(即清除增量中断标志之前)，该中断一直保持有效。

CIIR 寄存器描述见表 5.145。

表 5.145 计数器增量中断寄存器位

CIIR	功能	描述
0	IMSEC	为 1 时，秒值的增加产生一次中断。
1	IMMIN	为 1 时，分值的增加产生一次中断。
2	IMHOUR	为 1 时，小时值的增加产生一次中断。
3	IMDOM	为 1 时，日期（月）值的增加产生一次中断。
4	IMDOW	为 1 时，星期值的增加产生一次中断。
5	IMDOY	为 1 时，日期（年）值的增加产生一次中断。
6	IMMON	为 1 时，月值的增加产生一次中断。
7	IMYEAR	为 1 时，年值的增加产生一次中断。

报警屏蔽寄存器（AMR - 0xE0024010）

报警屏蔽寄存器（AMR）允许用户屏蔽任意报警寄存器，如年报警寄存器。表 5.146 所示为 AMR 位与报警寄存器位之间的关系。对于报警功能来说，若要产生中断，非屏蔽的报警寄存器必须匹配对应的时间计数器值，且只有从不匹配到匹配时才会产生中断。向中断位置寄存器（ILR）的 bit1 写入 1 清除相应的中断。**如果所有屏蔽位都置位，报警将被禁止。**

表 5.146 报警屏蔽寄存器位

AMR	功能	描述
0	AMRSEC	为 1 时，秒值不与报警寄存器比较。
1	AMRMIN	为 1 时，分值不与报警寄存器比较。
2	AMRHOUR	为 1 时，小时值不与报警寄存器比较。
3	AMRDOM	为 1 时，日期（月）值不与报警寄存器比较。
4	AMRDOW	为 1 时，星期值不与报警寄存器比较。
5	AMRDOY	为 1 时，日期（年）值不与报警寄存器比较。
6	AMRMON	为 1 时，月值不与报警寄存器比较。
7	AMRYEAR	为 1 时，年值不与报警寄存器比较。

5.17.8 完整时间寄存器

时间计数器的值可选择以一个完整格式读出，程序员只需执行 3 次读操作即可读出所有的时间计数器值，完整时间寄存器见表 5.147、表 5.148 和表 5.149。每个时间值(比如秒值、分值)的最低位分别位于寄存器的 bit0, bit8, bit16 或 bit24。

完整时间寄存器为只读寄存器。要更新时间计数器的值，必须对时间计数器寻址。

完整时间寄存器 0（CTIME0 - 0xE0024014）

完整时间寄存器 0 包含的时间值为：秒、分、小时和星期。

CTIME0 寄存器描述见表 5.147。

表 5.147 完整时间寄存器 0

CTIME0	功能	描述
31:27	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。
26:24	星期	星期值 该值的范围为 0~6。
23:21	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。
20:16	小时	小时值 该值的范围为 0~23。
15:14	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。
13:8	分	分值 该值的范围为 0~59。
7:6	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。
5:0	秒	秒值 该值的范围为 0~59。

完整时间寄存器 1（CTIME1 - 0xE0024018）

完整时间寄存器 1 包含的时间值为：日期（月）、月和年。

CTIME1 寄存器描述见表 5.148。

表 5.148 完整时间寄存器 1

CTIME1	功能	描述
31:28	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。
27:16	年	年值 该值的范围为 0~4095。
15:12	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。
11:8	月	月值 该值的范围为 1~12。

接上表

CTIME1	功能	描述
7:5	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。
4:0	日期（月）	日期（月）值 该值的范围为 1~28, 29, 30 或 31（取决于月份以及是否为闰年）。

完整时间寄存器 2（CTIME2 - 0xE002401C）

完整时间寄存器 2 仅包含日期（年）。**使用前需要先初始化 DOY 寄存器，**因为 CTIME2 寄存器的值来源于 DOY 寄存器，而 DOY 寄存器需要单独的初始化，即是初始化年、月、日时间计数器不会使 DOY 的内容改变。

CTIME2 寄存器描述见表 5.149。

表 5.149 完整时间寄存器 2

CTIME2	功能	描述
8:0	日期（年）	日期（年）值 该值的范围为 1~365（闰年为 366）。
31:9	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。

5.17.9 时间计数器组

时间值包含 8 个寄存器，见表 5.150，表中所示的寄存器是可读或可写的。时间计数器的关系和值见表 5.151。其中，DOY 寄存器也需要单独的初始化，即是初始化年、月、日时间计数器不会使 DOY 的内容改变。

表 5.150 时间计数器寄存器

地址	名称	规格	描述	访问
0xE0024020	SEC	6	秒值 该值的范围为 0~59。	R/W
0xE0024024	MIN	6	分值 该值的范围为 0~59。	R/W
0xE0024028	HOUR	5	小时值 该值的范围为 0~23。	R/W
0xE002402C	DOM	5	日期（月）值 该值的范围为 1~28,29,30 或 31（取决于月份以及是否为闰年）。[1]	R/W
0xE0024030	DOW	3	星期值 该值的范围为 0~6。[1]	R/W
0xE0024034	DOY	9	日期（年）值 该值的范围为 1~365（闰年为 366）。[1]	R/W
0xE0024038	MONTH	4	月值 该值的范围为 1~12。	R/W
0xE002403C	YEAR	12	年值 该值的范围为 0~4095。	R/W

注：1. 这些值只能在适当的时间间隔处递增且在定义的溢出点复位。为了使这些值有意义，它们不能进行计算且必须被正确初始化。

表 5.151 时间计数器的关系和值

计数器	规格	计数驱动源	最小值	最大值
秒	6	Clk1（见图 5.73）	0	59
分	6	秒	0	59
小时	5	分	0	23
日期（月）	5	小时	1	28, 29, 30 或 31
星期	3	小时	0	6
日期（年）	9	小时	1	365 或 366（闰年）

接上表

计数器	规格	计数驱动源	最小值	最大值
月	4	日期（月）	1	12
年	12	月或日期（年）	0	4095

5.17.10 报警寄存器组

报警寄存器见表 5.152。这些寄存器的值与时间计数器相比较，如果所有未屏蔽（见“报警屏蔽寄存器”）的报警寄存器都与它们对应的时间计数器相匹配，那么将产生一次中断。向中断位置寄存器的 bit1 写入 1 清除中断。

表 5.152 报警寄存器

地址	名称	规格	描述	访问
0xE0024060	ALSEC	6	秒报警值	R/W
0xE0024064	ALMIN	6	分报警值	R/W
0xE0024068	ALHOUR	5	小时报警值	R/W
0xE002406C	ALDOM	5	日期（月）报警值	R/W
0xE0024070	ALDOW	3	星期报警值	R/W
0xE0024074	ALDOY	9	日期（年）报警值	R/W
0xE0024078	ALMON	4	月报警值	R/W
0xE002407C	ALYEAR	12	年报警值	R/W

5.17.11 基准时钟分频器（预分频器）

基准时钟分频器（在下文中称为预分频器）允许从任何频率高于 65.536kHz（2×32.768kHz）的外设时钟源产生一个 32.768kHz 的基准时钟。这样，不管外设时钟的频率为多少，RTC 总是以正确的速率运行。预分频器通过一个包含整数和小数部分的值对外设时钟(pclk)进行分频。这样就产生了一个不是恒定频率的连续输出，有些时钟周期比其它周期多 1 个 pclk 周期，但是每秒钟的计数总数总是 32768。

预分频器结构见图 5.76。

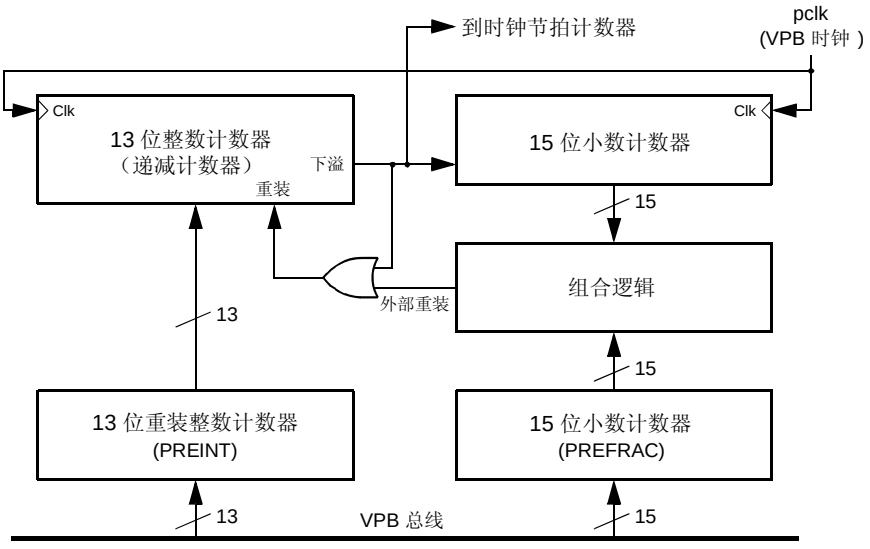


图 5.76 RTC 预分频器方框图

基准时钟分频器包含一个 13 位整数计数器和一个 15 位小数计数器，见表 5.153。使用该规格的原因如下：

- 对于 LPC2114/2124/2210/2212/2214 所支持的频率，13 位整数计数器是必要的。可以这样进行计算：频率 160MHz 除以 32768 再减去 1 等于 4881，余数为 26,624。保存 4881 需要 13 个位。13 位实际所能支持的最高频率为 268.4MHz（32768×8192）。
- 余数的最大值为 32767，需要 15 位来保存。

表 5.153 基准时钟分频寄存器

地址	名称	规格	描述	访问
0xE0024080	PREINT	13	预分频值，整数部分	R/W
0xE0024084	PREFRAC	15	预分频值，小数部分	R/W

预分频整数寄存器（PREINT - 0xE0024080）

PREINT 寄存器描述见表 5.154。预分频值的整数部分计算如下：

PREINT = int (pclk/32768) – 1 (其中 PREINT 的值必须大于等于 1)

表 5.154 预分频整数寄存器

PREINT	功能	描述	复位值
15:13	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
12:0	预分频整数	包含 RTC 预分频值的整数部分	0

预分频小数寄存器（PREFRAC - 0xE0024084）

PREFRAC 寄存器描述见表 5.155。预分频值的小数部分计算如下：

PREFRAC = pclk – ((PREINT+1)×32768)

表 5.155 预分频小数寄存器

PREFRAC	功能	描述	复位值
15	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA
14:0	预分频小数	包含 RTC 预分频值的小数部分	0

预分频器的使用举例

先假设一个最简单的状况，pclk 频率为 65.537kHz。那么：

PREINT = int (pclk/32768) – 1 = 1, PREFRAC = pclk – ((PREINT+1)×32768) = 1

使用此设定，每秒钟有 32767 次 2 个 pclk 周期计数，1 次 3 个 pclk 周期计数，加起来每秒刚好为 RTC 提供 32768 个时钟。

再假设一个比较实际的状况，pclk 频率为 10MHz。那么：

PREINT = int (pclk/32768) – 1 = 304, PREFRAC = pclk – ((PREINT+1)×32768) = 5760

这时，有 5760 个预分频器输出时钟宽度为 306 (305+1)个 pclk 周期。余下的时钟宽度为 305 个 pclk 周期。

采用相似的方法可以将任何高于 65536kHz 的 pclk 频率（每秒钟的周期数必须是偶数）转换成 RTC 的 32kHz 基准时钟。唯一需要注意的是，如果 PREFRAC 不等于 0，那么每秒当中的 32768 个时钟长度是不完全相同的，有些时钟会比其它时钟多 1 个 pclk 周期。虽然较长的脉冲已经尽可能地分配到剩余的脉冲当中，但是在希望直接观察时钟节拍计数器的应用中可能需要注意这种“抖动”。

5.17.12 RTC 使用注意事项

由于 RTC 的时钟源为 VPB 时钟（pclk），时钟出现的任何中断都会导致时间值的偏移。如果 RTC 初始化错误或 RTC 运行时间内出现了一个错误，它们带来的变化都将影响真实的时钟时间。

LPC2114/2124／2210/2212/2214 在断电时不能保持 RTC 的状态。如果时钟源丢失、中断或改变，RTC 也无法维持时间计数。**芯片的断电将使 RTC 寄存器的内容完全丢失。**进入掉电模式时由于 Fpclk 已停止，会使时间的更新出现误差。在系统操作过程中（重新配置 PLL、VPB 定时器或 RTC 预分频器）改变 RTC 的时间基准会使累加时间出现错误。

5.17.13 使用示例

实时时钟(RTC)可用来进行定时报警，日期及时分秒计时等等。RTC 不具备独立时钟源，其计数时钟由 pclk 进行分频得到，它的基准时钟分频器允许调节任何频率高于 65.536KHz 的外设时钟源产生一个 32.768KHz 的基准时钟，实现准确计时操作。在微处理器掉电模式下 RTC 是停止的。

如图 5.77，实时时钟的时钟源是由 PCLK 通过基准时钟分频器(PREINT、PREFRAC)，调整出 32768Hz 的频率，然后供给 CTC 计数器；CTC 是一个 15 位的计数器，它位于秒计数器之前，CTC 每秒计数 32768 个时钟；当有 CTC 秒进行位时，完整时间 CTME0～2、RTC 时间寄存器(如 SEC、MIN 等)将会更新；RTC 中断有两种，一种是增量中断，由 CIIR 进行控制，另一种为报警中断，由 AMR 寄存器和各报警时间寄存器控制，如 ALSEC、ALMIN 等；报警位置寄存器 ILR 用来产生相应的中断标志；RTC 时钟控制寄存器 CCR 用于使能实时时钟，CTC 复位控制等。

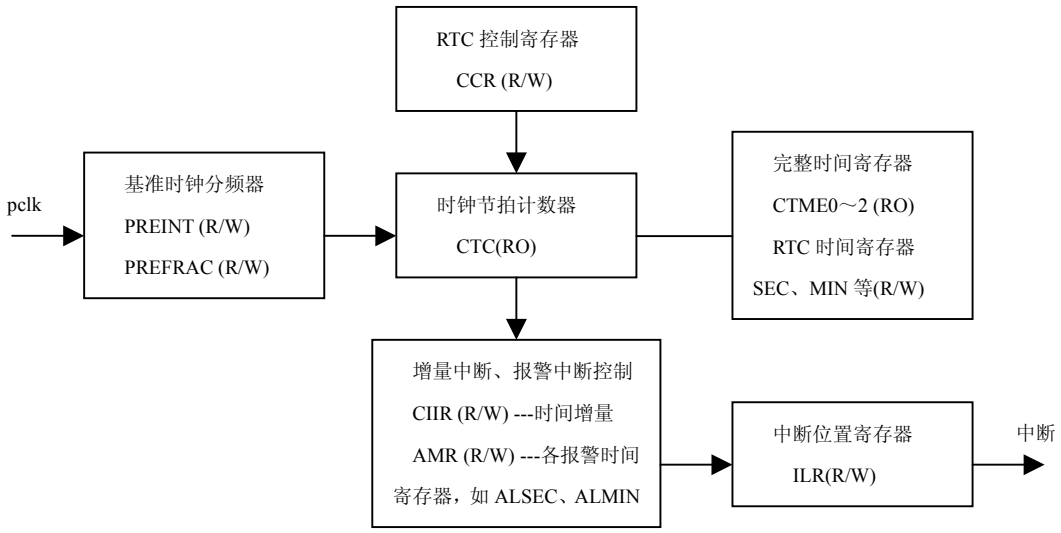


图 5.77 RTC 的寄存器功能框图

其中，日期寄存器(表示“日”)有两个，分别为 DOY 和 DOM，DOY 表示为一年中的第几日，值为 1～365(闰年为 366)；而 DOM 则为一月中的第几日，值为 1～28/29/30/31，一般日期计数使用 DOM 即可。

预分频寄存器值的计数如下：

$$PREINT = \text{int}(\frac{PCLK}{32768}) - 1$$

$$PREFRAC = PCLK - ((PREINT + 1) \times 32768)$$

RTC 基本操作方法：

- 设置 RTC 基准时钟分频器(PREINT、PREFRAC)；
- 初始化 RTC 时钟值，如 YEAR、MONTH、DOM 等；
- 报警中断设置，如 CIIR、AMR 等；
- 启动 RTC，即 CCR 的 CLKEN 位置位；
- 读取完整时间寄存器值，或等待中断。

1. RTC 初始化

程序清单 5.39 为 RTC 初始化示例程序。程序先设置基准时钟分频器，然后初始化时钟值，再启动 RTC。

程序清单 5.39 RTC 初始化示例

```
/*****************************************************************************
* 名称：RTCIni()
* 功能：初始化实时时钟。
* 入口参数：无
* 出口参数：无
*****************************************************************************/
void  RTCIni(void)
{  PREINT = Fpclk / 32768 - 1;              // 设置基准时钟分频器
   PREFRAC = Fpclk - (Fpclk / 32768) * 32768;

   YEAR = 2004;                             // 初化年
   MONTH = 2;                               // 初化月
   DOM = 19;                                // 初化日
   DOW = 4;                                 // 初化星期
   HOUR = 8;                                // 初化时
   MIN = 30;                                // 初化分
   SEC = 0;                                 // 初化秒

   CIIR = 0x01;                             // 设置秒值的增量产生一次中断
   CCR = 0x01;                              // 启动 RTC
}
```

2. 设置定时报警

定时报警设置示例程序如程序清单 5.40 所示。

程序清单 5.40 定时报警设置示例

```
   ILR = 0x03;                              // 清除 RTC 中断标志
   CIIR = 0x02;                             // 设置分值增量中断
   ALHOUR = 12;
   ALMIN = 0;
   ALSEC = 0;
```

```
    AMR = 0xF8;
```

3. 读取 RTC 时钟值

按读时间计数寄存器方式读取 RTC 时钟程序如程序清单 5.41 所示。

程序清单 5.41 读取 RTC 时钟值—时间计数寄存器

```
struct   DATE
{
    uint16        year;
    uint8         mon;
    uint8         day;
    uint8         dow;
};
struct   TIME
{
    uint8         hour;
    uint8         min;
    uint8         sec;
};

/*****************************************************************************
* 名称：GetTime()
* 功能：读取 RTC 的时钟值。
* 入口参数：d        保存日期的 DATE 结构变量的指针
*           t        保存时间的 TIME 结构变量的指针
* 出口参数：无
*****************************************************************************/
void   GetTime(struct DATE *d,   struct TIME *t)
{
    d->year = YEAR;
    d->mon = MONTH;
    d->day = DOM;
    t->hour = HOUR;
    t->min = MIN;
    t->sec = SEC;
}
```

按读完整时间寄存器方式读取 RTC 时钟程序如程序清单 5.42 所示，其中 DATA、TIME 结构与程序清单 5.41 的相同。

程序清单 5.42 读取 RTC 时钟值—完整时间寄存器

```
/*****************************************************************************
* 名称：GetTime()
* 功能：读取 RTC 的时钟值。
```

```
* 入口参数：d        保存日期的 DATE 结构变量的指针
*           t        保存时间的 TIME 结构变量的指针
* 出口参数：无
*********************************************************************************/
void  GetTime(struct DATE *d,  struct TIME *t)
{   uint32  times,  dates;

    times = CTIME0;
    dates = CTIME1;

    d->year = (dates>>16)&0xFFF;          // 取得年的值
    d->mon = (dates>>8)&0x0F;             // 取得月的值
    d->day = dates&0x1F;                  // 取得日的值
    t->hour = (times>>16)&0x1F;           // 取得时的值
    t->min = (times>>8)&0x3F;             // 取得分的值
    t->sec = times&0x3F;                  // 取得秒的值
}
```

5.18 看门狗

5.18.1 特性

- 带内部预分频器的可编程 32 位定时器
- 如果没有周期性重装(即喂狗)，则产生片内复位
- 具有调试模式
- 看门狗由软件使能，但只能由硬件复位或看门狗复位/中断来禁止
- 错误/不完整的喂狗时序会导致复位/中断(如果看门狗已经使能)
- 具有指示看门狗复位的标志
- 可选择 $t_{pclk}\times4$ 倍数的时间周期：从($t_{pclk}\times256\times4$) 到 ($t_{pclk}\times2^{32}\times4$)

5.18.2 应用

看门狗的用途是使微控制器在进入错误状态后的一定时间内复位。当看门狗使能时，如果用户程序没有在周期时间内喂狗（重装），看门狗会产生一个系统复位。

5.18.3 描述

看门狗包括一个 4 分频的预分频器和一个 32 位计数器。时钟 F_{pclk} 通过预分频器输入定时器，定时器进行递减计数。**定时器递减的最小值为 0xFF，如果设置一个小于 0xFF 的值，系统会将 0xFF 装入计数器。**因此最小看门狗间隔为($t_{pclk}\times256\times4$)，最大间隔为($t_{pclk}\times2^{32}\times4$)，两者都是 $t_{pclk}\times4$ 的倍数。看门狗应当根据下面的方法来使用：

——在 WDTC 寄存器中设置看门狗定时器的固定装载值

——在 WDMOD 寄存器中设置模式，并使能看门狗

——通过向 WDFEED 寄存器顺序写入 0xAA 和 0x55 启动看门狗

——在看门狗向下溢出之前应当再次喂狗以防止复位/中断

当看门狗计数器向下溢出时，程序计数器将从 0x00000000 开始，和外部复位一样。可以检查看门狗超时标志（WDTOF）来确定看门狗是否产生复位条件。WDTOF 标志必须由软件清零。

5.18.4 结构

看门狗结构方框图如图 5.78 所示。

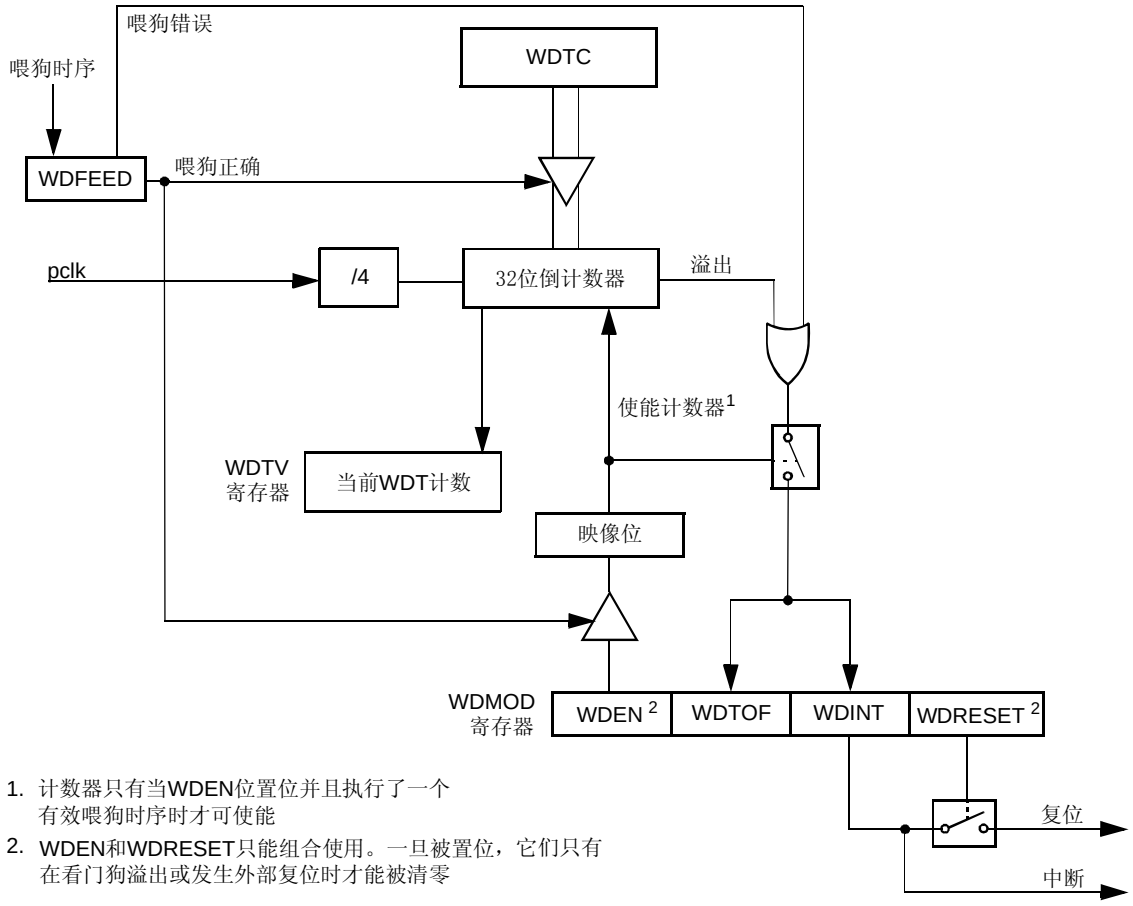


图 5.78 看门狗方框图

5.18.5 寄存器描述

寄存器汇总

看门狗包含 4 个寄存器，见表 5.156。

表 5.156 看门狗寄存器映射

名称	描述	访问	复位值	地址
WDMOD	看门狗模式寄存器 该寄存器包含看门狗定时器的基本模式和状态。	读/设置	0	0xE0000000
WDTC	看门狗定时器常数寄存器 该寄存器决定超时值。	读/写	0xFF	0xE0000004
WDFEED	看门狗喂狗寄存器 向该寄存器顺序写入 AAh 和 55h 使看门狗定时器重新装入预设值(即 WDTC 的值)。	只写	NA	0xE0000008
WDTV	看门狗定时器值寄存器 该寄存器读出看门狗定时器的当前值。	只读	0xFF	0xE000000C

看门狗模式寄存器（WDMOD - 0xE0000000）

WDMOD 寄存器描述见表 5.157。

表 5.157 看门狗模式寄存器

WDMOD	功能	描述	复位值
0	WDEN	看门狗中断使能位（只能置位）	0
1	WDRESET	看门狗复位使能位（只能置位）	0
2	WDTOF	看门狗超时标志	0（外部复位）
3	WDINT	看门狗中断标志（只读）	0
7:4	保留	保留，用户软件不要向其写入 1。从保留位读出的值未被定义。	NA

WDEN 和 RESET 通过 WDEN 和 RESET 的组合来控制看门狗的操作，如下：

WDEN	WDRESET	
0	X	看门狗关闭时的调试/操作
1	0	带看门狗中断的调试，但没有 WDRESET
1	1	带看门狗中断和 WDRESET 的操作

一旦 WDEN 和/或 WDRESET 位设置，就无法使用软件将其清零。这两个标志由外部复位或看门狗定时器溢出清零。**注意，将 WDEN 设置为 1 只是使能 WDT，但没有启动 WDT，当第一次喂狗操作时才启动 WDT。**

WDTOF 当看门狗发生超时，看门狗超时标志置位。该标志由软件清零。

WDINT 当看门狗发生超时，看门狗中断标志置位。产生的任何复位都会使该位清零。

看门狗定时器常数寄存器

WDTC 寄存器决定看门狗超时值，见表 5.158。当喂狗时序产生时，WDTC 的内容重新装入看门狗定时器。它是一个 32 位寄存器，低 8 位在复位时设置为 1。写入一个小于 0xFF 的值会使 0xFF 装入 WDTC，因此超时的最小时间间隔为 $t_{pclk}\times256\times4$。

WDTC 寄存器描述见表 5.158。

表 5.158 看门狗定时器常数寄存器

WDTC	功能	描述	复位值
31:0	计数值	看门狗超时间隔	0xFF

看门狗喂狗寄存器（WDFEED - 0xE0000008）

向该寄存器写入 0xAA，然后写入 0x55 会使 WDTC 的值重新装入看门狗定时器，WDFEED 寄存器描述见表 5.159。如果看门狗通过 WDMOD 寄存器使能，该操作还将启动看门狗运行。在看门狗能够产生中断/复位之前，即看门狗溢出之前，必须完成一次有效的喂狗时序。向 WDFEED 寄存器写入 0xAA 的下一个操作应当是向 WDFEED 寄存器写入 0x55，一次不正确喂狗时序之后的第二个 pclk 周期看门狗复位/中断被触发。

表 5.159 看门狗喂狗寄存器

WDFEED	功能	描述	复位值
7:0	喂狗	喂狗值应当为 0xAA，然后是 0x55。	未定义

看门狗定时器值寄存器（WDTV - 0xE000000C）

WDTV 寄存器用于读取看门狗定时器的当前值，见表 5.160。

表 5.160 看门狗定时器值寄存器

WDTV	功能	描述	复位值
31:0	计数	当前定时器值	0xFF

5.18.6 使用示例

LPC2114/2124/2210/2212/2214 的 WDT 定时器为递减计数，当下溢时将会产生中断或复位。定时器的最小装载值为 0xFF，即最小的 WDT 时间为 $t_{pclk}\times256\times4$。WDT 的时钟源是由系统时钟 pclk 提供，它不具备独立看门狗时钟振荡器，在掉电模式下 WDT 也是停止的，所以不能用来它来唤醒掉电的 CPU。

WDT 溢出时间计算如下：

$$溢出时间 = N \times t_{pclk} \times 4$$

其中，N 为 WDTC 的设置值。

WDT 基本操作方法：

- 设置 WDT 定时器重装值 (WDTC)；
- 设置 WDT 工作模式，启动 WDT (WDMOD)；
- 对 WDFEED 操作，实现喂狗。

1. WDT 初始化示例

WDT 初始化。当要使用 WDT 功能时，置位 WDEN 即可，一旦置位 WDEN，则只有通过复位系统 WDEN 才能复位，WDRESET 位可设置 WDT 用于 WDT 复位还是用于 WDT 中断，若 WDRESET 置位，则用于 WDT 复位。如程序清单 5.43 所示，先设置 WDT 的定时器常数 WDTC，然后设置 WDT 复位工作模式，并启动 WDT。

程序清单 5.43 WDT 初始化

```
WDTC = 0x10000;          // 设置 WDT 定时值
WDMOD = 0x03;            // 设置 WDT 工作模式，启动 WDT
```

2. WDT 喂狗程序

WDT 喂狗操作。对 WDFEED 写入 0xAA，再写入 0x55 即可将 WDTC 的值重新装入看门狗定时器，实现喂狗操作，如程序清单 5.44 所示。

程序清单 5.44 WDT 喂狗操作

```
void  WdtFeed(void)
{  WDFFED = 0xAA;
   WDFFED = 0x55;
}
```

5.19 本章小结

本章是以 LPC2114/2124/2210/2212/2214 为例，详细介绍了 PHILIPS 公司的 LPC2000 系列 ARM7 微控制器的硬件结构，从引脚功能到其它片内外，为电路原理设计、系统程序设计打下基础。

思考与练习

1. 基础知识

a) LPC2114 可使用的外部晶振频率范围是多少？(使用/不使用 PLL 功能时)
b) 描述一下 LPC2210 的 P0.14、P1.20、P1.26、BOOT1 和 BOOT0 引脚在芯片复位时分别有什么作用？并简单说明 LPC2000 系列 ARM7 微控制器的复位处理流程。
c) LPC2000 系列 ARM7 微控制器对向量表有何要求？(向量表中的保留字)
d) 如何启动 LPC2000 系列 ARM7 微控制器的 ISP 功能？相关电路应该如何设计？
e) LPC2000 系列 ARM7 微控制器片内 FLASH 是多位宽度的接口？它是通过哪个功能模块来提高 FLASH 的访问速度？
f) 若 LPC2210 的 BANK0 存储块使用 32 位总线，访问 BANK0 时，地址线 A1、A0 是否有效？EMC 模块中的 BLS0～BLS4 具有什么功能？
g) LPC2000 系列 ARM7 微控制器具有引脚功能复用特性，那么如何设置某个管脚为指定功能？
h) 设置管脚为 GPIO 功能时，如何控制某个管脚单独输入/输出？当需要知道某个管脚当前的输出状态时，是读取 IOPIN 寄存器还是读取 IOSET 寄存器？
i) P0.2 和 P0.3 口是 I^2C 接口，当设置它们为 GPIO 时，是否需要外接上拉电阻才能输出高电平？
j) 使用 SPI 主模式时，SSEL 引脚是否可以作为 GPIO？若不能，SSEL 引脚应如何处理？
k) LPC2114 的两个 UART 是符合什么标准？哪一个 UART 可用作 ISP 通讯？哪一个 UART 具有 MODEM 接口？
l) LPC2114 具有几个 32 位定时器？PWM 定时器是否可以作通用定时器使用？
m) LPC2000 系列 ARM7 微控制器具有哪两种低耗模式？如何降低系统的功耗？

2. 计算 PLL 设置值

假设有一个基于 LPC2114 的系统，所使用的晶振为 11.0592MHz 石英晶振。请计算出最大的系统时钟(cclk)频率为多少 MHz？此时 PLL 的 M 值和 P 值各为多少？请列出计算公式，并编写设置 PLL 的程序段。

3. 存储器重映射

LPC2210 具有(　)种存储映射模式。

(A) 3　　(B) 5　　(C) 1　　(D) 4

当程序已固化到片内 FLASH，向量表保存在 0x00000000 起始处，则 MAP1:0 的值应该为(　)。

(A) 00　　(B) 01　　(C) 10　　(D) 11

LPC2000 系列 ARM7 微控制器存储器重映射的目标起始地址为(　)，共有(　)个字。

(A) 0x00000000，8　　(B) 0x40000000，8

(C) 0x00000000，16　　(D) 0x7FFFE000，8

4. 外部中断唤醒掉电设计

以下代码是初始化外部中断 0，用它来唤醒掉电的 LPC2114，请填空。

```
PINSEL0 = 0x00000000;
PINSEL1 = _____;              // 设置 I/O 口连接，P0.16 设置为 EINT0
EXTMODE = _____;              // 设置 EINT0 为电平触发模式
EXTPOLAR = _____;             // 设置 EINT0 为低电平触发
EXTWAKE = _____;              // 允许外部中断 0 唤醒掉电的 CPU
EXTINT = 0x0F;                // 清除外部中断标志
```

第6章 接口技术与硬件设计

本章以 PHILIPS 的 LPC2000 系列基于 ARM7 处理器核的微控制器为例，介绍 ARM7 的接口技术和硬件设计方法。

6.1 最小系统

一个嵌入式处理器自己是不能独立工作的，必须给它供电、加上时钟信号、提供复位信号，如果芯片没有片内程序存储器，则还要加上存储器系统，然后嵌入式处理器芯片才可能工作。这些提供嵌入式处理器运行所必须的条件的电路与嵌入式处理器共同构成了这个嵌入式处理器的最小系统。而大多数基于 ARM7 处理器核的微控制器都有调试接口，这部分在芯片实际工作时不是必需的，但因为这部分在开发时很重要，所以笔者也把这部分也归入最小系统中。

6.1.1 框图

图 6.1 为嵌入式微控制器的最小系统框图，其中存储器系统是可选的，这是因为很多面向嵌入式领域嵌入式微控制器内部设计了程序存储器和数据存储器，存储器系统不需要自己设计了。调试测试接口也不是必需的，但它在开发工程中发挥的作用极大，所以至少在样品阶段需要设计这部分电路。

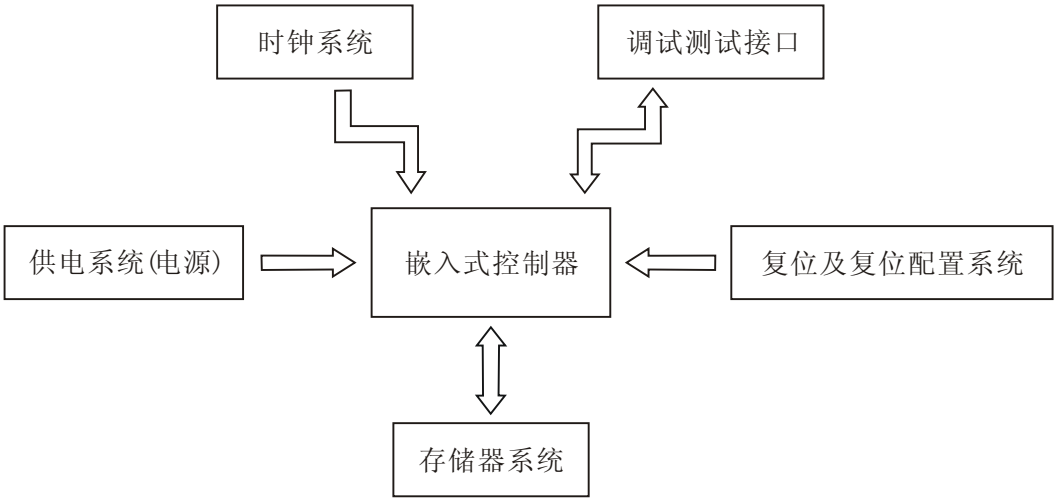


图 6.1 最小系统原理框图

6.1.2 电源

电源系统为整个系统提供能量，是整个系统工作的基础，具有极其重要的地位，但却往往被忽略。依据笔者的经验，如果电源系统处理得好，整个系统的故障往往减少了一大半。电源系统的示意图见图 6.2。

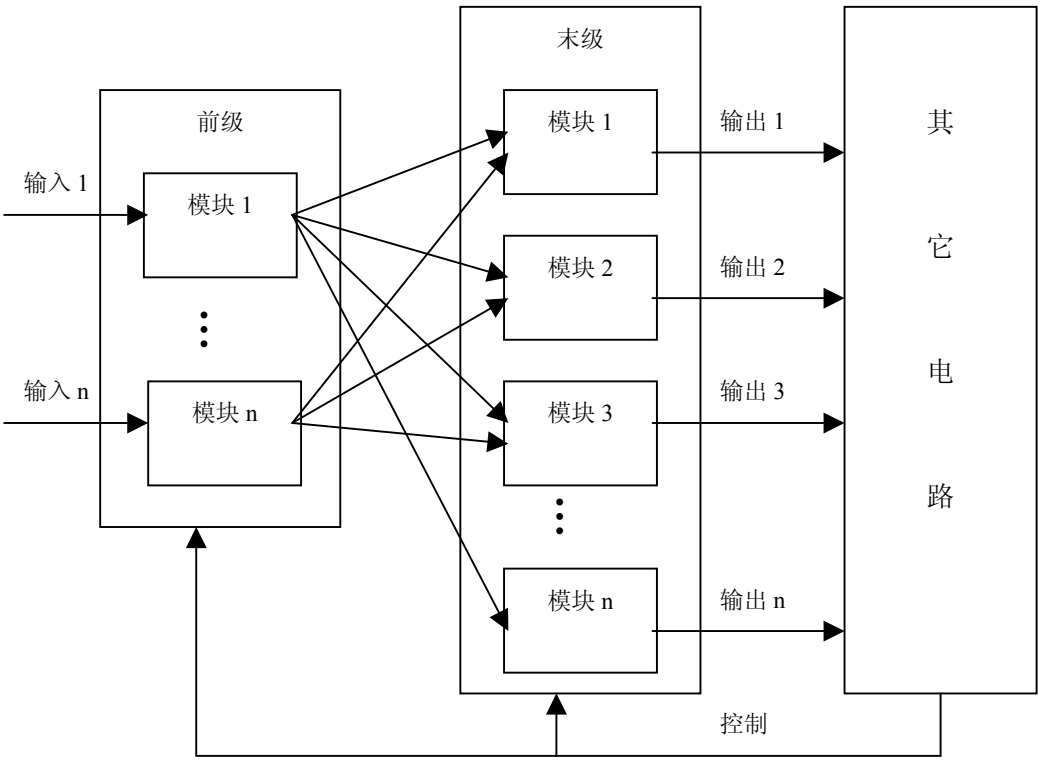


图 6.2 系统示意图

设计电源系统的过程实质是一个权衡的过程，必须考虑如下因素：

1．输出的电压电流功率
2．输入的电压电流
3．安全因素
5．输出纹波
6．电磁兼容和电磁干扰
7．体积限制
8．功耗限制
9．成本限制

电源设计本身是一个很大的课题，其内容不是一本书所能够容纳得下的，本书也不打算详细介绍，读者如果需要了解，请参考相关书籍。下面以 LPC2000 系列的电源系统为例，介绍电源系统的设计步骤：

1．分析需求

目前的 LPC2000 ARM 芯片除 LPC2104/2105/2106 外，均有 4 组电源输入：数字 3.3V、数字 1.8V、模拟 3.3V 和模拟 1.8V。因此，理想情况下电源系统需要提供 4 组独立的电源：两组 3.3V 电源和两组 1.8V 电源，它们需要单点接地或大面积接地。如果系统的其它部分还有其它电源需求，则还需要更多的末级电源。但如果不使用 LPC2000 的 AD 功能，或对 AD 的要求不高，模拟电源和数字电源可以不分开供电。这里假设不使用 LPC2000 的 AD 功能，且其它部分对电源没有特殊要求。这样，末级只需要提供两组电源。

电源的前级设计与末级设计和供给系统的电源输入相关。这里假设输入未经过稳压的 9～12V 直流电源输入。

2．设计末级电源电路

从 LPC2000 的手册可知，其 1.8V 消耗电流的极限是 70mA，其它部分无需 1.8V 电压。为了保证可靠性并为以后升级留下余量，则电源系统 1.8V 能够提供的电流应当大于 300mA。

整个系统在 3.3V 上消耗的电流与外部条件有很大的关系，这里假设电流不超过 200mA，这样，电源系统 3.3V 能够提供 600mA 电流即可。

因为系统对这两组电压的要求比较高，且其功耗不是很大，所以不适合用开关电源，应当用低压差模拟电源（LDO）。合乎技术参数的 LDO 芯片很多，Sipex 半导体 SPX1117 是一个较好的选择，它的性价比较好，且用一些产品可以与它直接替换，减少采购风险。

SPX1117 简介

SPX1117 为一个低功耗正向电压调节器，其可以用在一些高效率、小封装的低功耗设计中。这款器件非常适合便携式电脑及电池供电的应用。SPX1117 有很低的静态电流，在满负载时其低压差仅为 1.1V。当输出电流减少时，静态电流随负载变化，并提高效率。SPX1117 输出电压可调节，以选择 1.5V，1.8V，2.5V，2.85V，3.0V，3.3V 及 5V 的输出电压。

SPX1117 提供了多种 3 引脚封装：SOT-223，TO-252，TO-220 及 TO-263。一个 10μF 的输出电容可有效地保证稳定性，然而在大多数应用中，仅需一个更小的 2.2μF 电容。

SPX1117 的主要特点

- 0.8A 稳定输出电流；
- 1A 稳定峰值电流；
- 3 端固定或可调节电压输出（电压可选：1.5V，1.8V，2.5V，2.85V，3.0V，3.3V 及 5V）；
- 低静态电流；
- 0.8A 时低压差为 1.1V；
- 0.1%线性调整率/0.2%负载调整率；
- 2.2μF 陶瓷电容即可保持稳定；
- 过流及温度保护；
- 多封装：SOT-223，TO-252，TO-220 及 TO-263（现已提供无铅封装）。

SPX1117 的内部功能模块图

SPX1117 的内部功能模块图见图 6.3。

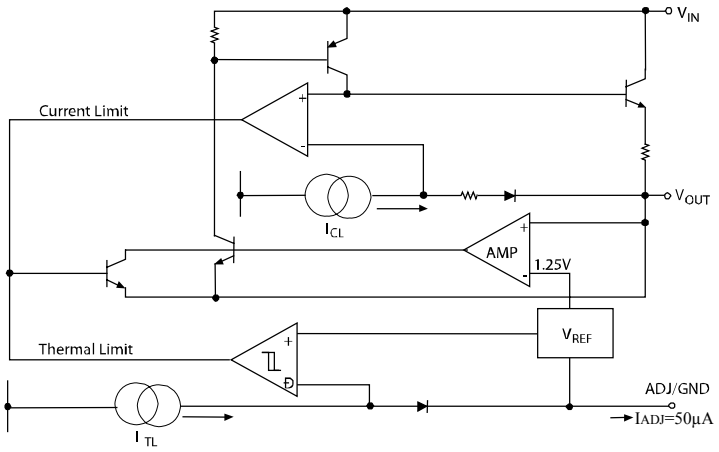


图 6.3 SPX1117 的内部功能模块图

SPX1117 的引脚排列

SPX1117 的引脚排列见图 6.4。

关于 SPX1117 详细的数据手册，请到广州周立功单片机发展有限公司的网站下载，网址为：http://www.zlgmcu.com。根据 SPX1117 的数据手册，电源系统的末级电路设计如图 6.5 所示。

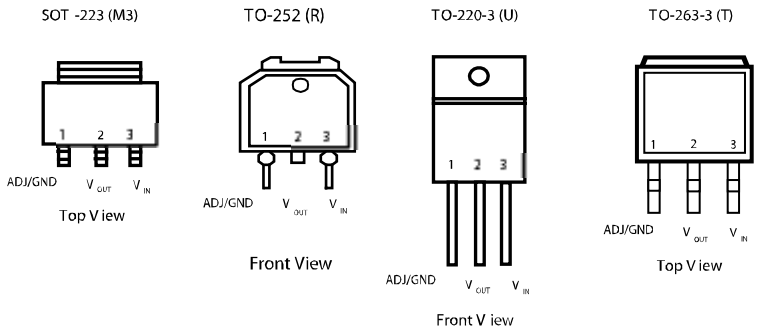


图 6.4 SPX1117 的引脚排列图

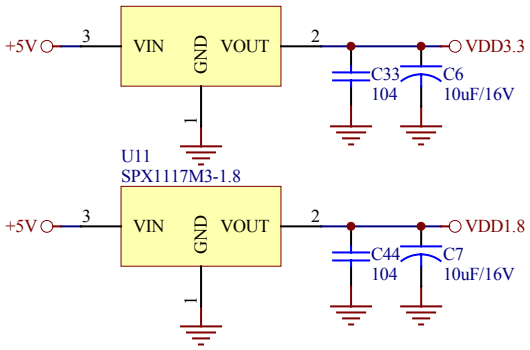


图 6.5 电源系统末级电路实例

3. 设计前级电源电路

尽管 SPX1117 允许的输入电压可达 20V（参考芯片数据手册），但太高的电压使芯片的发热量上升，散热系统不好设计，同时影响芯片的性能。同时，波动的电压对输出电压的波动也有一点影响。太高的压差也失去了选择低压差模拟电源的意义。这样，就需要前级电路调整一下。如果系统可能使用多种电源（如交流电和电池），各种电源的电压输出不一样，就更需要前级调整以适应末级的输入。从图 6.5 可以看出，前级的输出选择为 5V。选择 5V 作为前级的输出有两个原因，其一是这个电压满足 SPX1117 的要求，其二是目前很多器件还是需要 5V 供电的，这个 5V 就可以兼做前级和末级了。

根据系统在 5V 上消耗的电流和体积、成本等方面的考虑，前级电路可以使用开关电源，也可以使用模拟电源。相对模拟电源来说，开关电源效率较高，可以减少发热量，因而在功率较大时可以减小电源模块的体积，但电路复杂、输出电压纹波较大、在功率不是特别大时成本较高，同时开关电源是一个干扰源，对别的电路有一定的影响。模拟电源参考图 6.6，开关电源参考图 6.7。图 6.6、图 6.7 的 D1 均为防止反接电源烧毁电路而设计的。

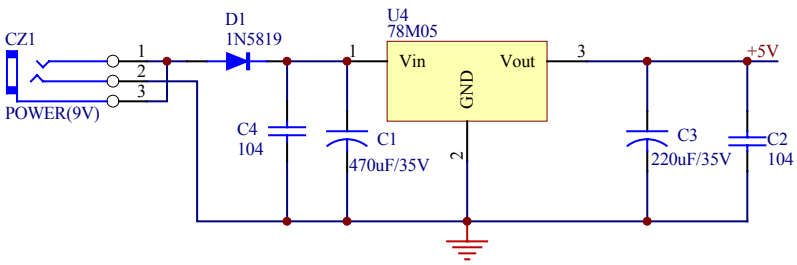


图 6.6 模拟电源

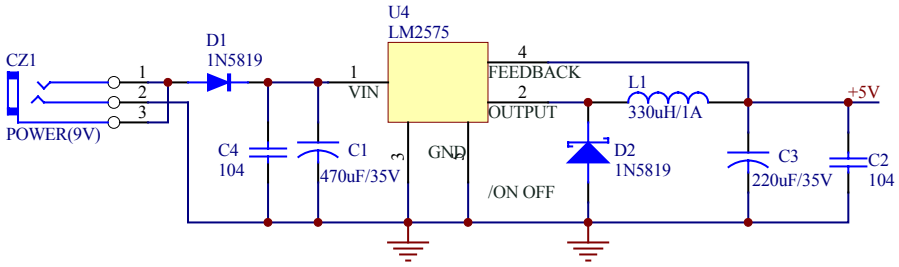


图 6.7 开关电源

关于 7805 的现成的资料很多地方都能找到，这里就不介绍了，读者自己查找一下。关于 LM2575 为美国国家半导体公司生产的开关电源专用模块，其具体信息可以在美国国家半导体公司的网站上下载，网址为：http://www.national.com/。

6.1.3 时钟

目前所有的微控制器均为时序电路，需要一个时钟信号才能工作，大多数微控制器具有晶体振荡器。基于以上事实，我们需要设计时钟电路。简单的方法是利用微控制器内部的晶体振荡器，但有些场合（如减少功耗、需要严格同步等情况）需要使用外部振荡源提供时钟信号。LPC2000 的时钟电路的设计见图 6.8，其中图 6.8(a)为使用微控制器内部的晶体振荡器设计时钟电路，而图 6.8(b)为外部电路产生时钟，关于电路元件参数的选择请参考 5.4.4 小节，而图 6.8(b)中的 Clock 可以是任何稳定的时钟信号源，如有源晶振等。

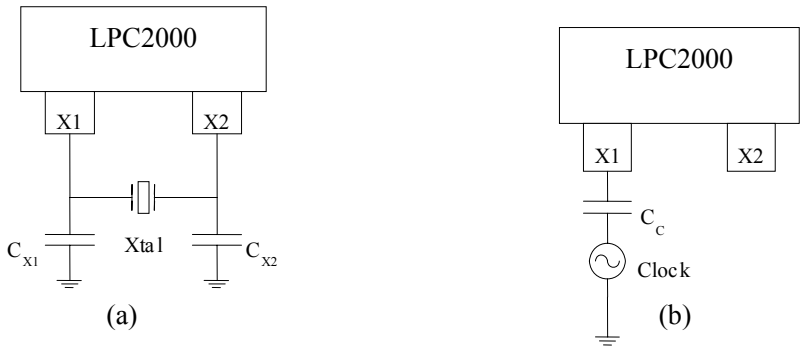


图 6.8 时钟电路设计

6.1.4 复位及复位芯片配置

微控制器在上电时状态并不确定，则造成微控制器不能正确工作。为解决这个问题，所有微控制器均有一个复位逻辑，它负责将微控制器初始化为某个确定的状态。这个复位逻辑需要一个复位信号才能工作。一些微控制器自己在上电时会产生复位信号，但大多数微控制器需要外部输入这个信号。因为这个信号会使微控制器初始化为某个确定的状态，所以这个信号的稳定性和可靠性对微控制器的正常工作有重大影响。图 6.9 为最简单的阻容复位电路，这个电路成本低廉，但不能保证任何情况产生稳定可靠的复位信号，所以一般场合需要使用专门的复位芯片。如果系统不需要手动复位，可以选择 MAX809；如果系统需要手动复位，可以选择 SP708SCN。复位芯片的复位门槛的选择至关重要，一般应当选择微控制器的 IO 口供电电压范围为标准；针对 LPC2000 来说就是这个范围为 3.0V～3.6V，所以其复位门槛应当选择为 2.93V。下面简单介绍一下 SP708 系列复位监控器件的基本原理及其应用设计方法。

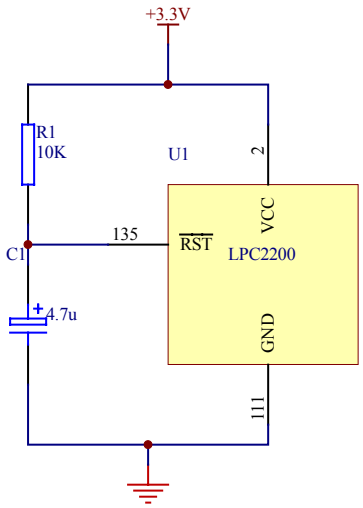


图 6.9 阻容复位电路

SP706/708 简介

SP706P/S/R/T，SP708R/S/T 系列属于微处理器（μP）监控器件。其集成有众多组件，可监测μP 及数字系统中的供电及电池的工作情况。由于以上众多组件的使用，SP706P/S/R/T，SP708R/S/T 系列可有效地增强系统的可靠性及工作效率。SP706P/S/R/T，SP708R/S/T 系列包含一个看门狗定时器，一个μP 复位模块，一个供电失败比较器，及一个手动复位输入模块。SP706P/S/R/T，SP708R/S/T 系列适用于+3.0V 或+3.3V 环境 ，如计算机、汽车系统、控制器及其他一些智能仪器。对于对电源供电要求严格的μP 系统/数字处理系统，SP706P/R/S/T，SP708R/S/T 系列是一款非常理想的选择。

SP708 的主要特点

- 高精度低电压监控器：
  2.63V 下的 SP706P/R 及 SP708R ；
  2.93V 下的 SP706S 及 SP708S；
  3.08V 下的 SP706T 及 SP708T。
- 复位脉冲宽度：200ms；
- 独立的看门狗定时器：溢出周期 1.6s（SP706P/S/R/T）；
- 最大电源电流 40μA；
- 支持开关式 TTL/CMOS 手动复位输入；
- Vcc 下降至 1V 时，产生$\overline{\text{RESET}}$信号；
- RESET 输出：
  SP706P 高电平有效；
  SP706R/S/T 低电平有效；
  SP708R/S/T 支持高/低电平两种方式。
- WDI 可以保持为浮空，以禁止看门狗功能；
- 内嵌 Vcc 干扰抑止电路；
- 提供 8 引脚 PDIP，NSOIC 及μSOIC 封装；
- 内嵌电压监测器，可检测供电失败或电池不足警告；
- 706P/R/S/T 及 708R/S/T 引脚兼容性增强以符合工业标准。

SP706 内部模块图

SP706 内部模块图见图 6.10。

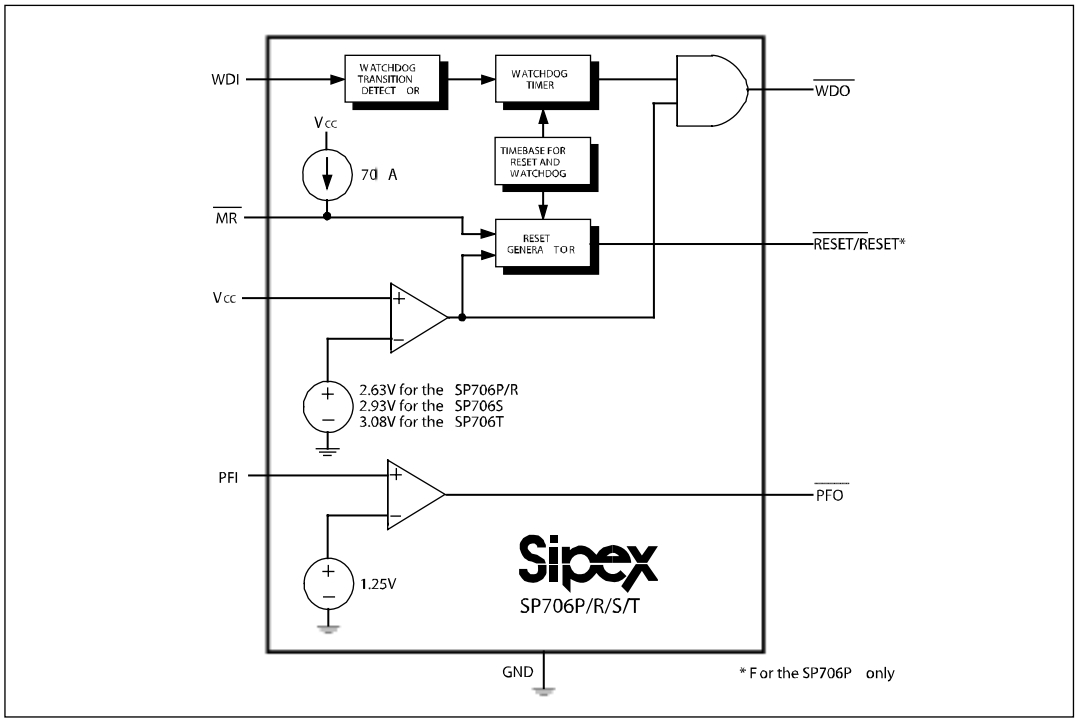


图 6.10 SP706 内部模块图

SP708 引脚分布图

SP708 引脚分布图见图 6.11。

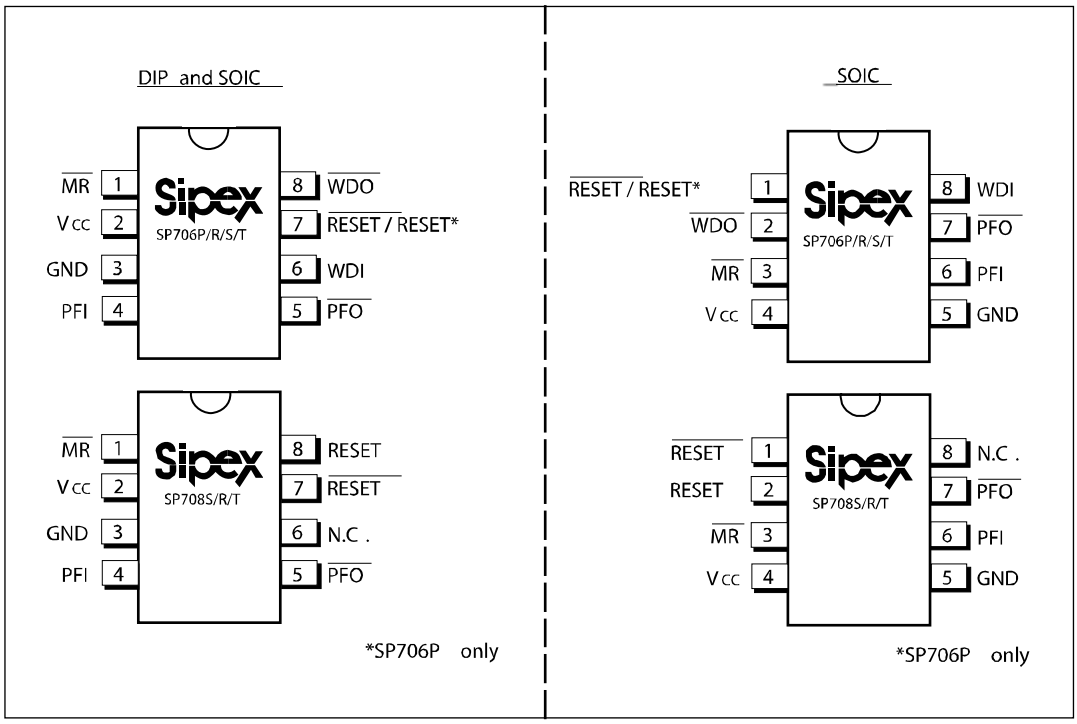


图 6.11 SP708 引脚分布图

关于 SP708 详细的数据手册，请到广州周立功单片机发展有限公司的网站下载，网址为：http://www.zlgmcu.com。根据 SP708 的数据手册，复位电路设计如图 6.12 所示。

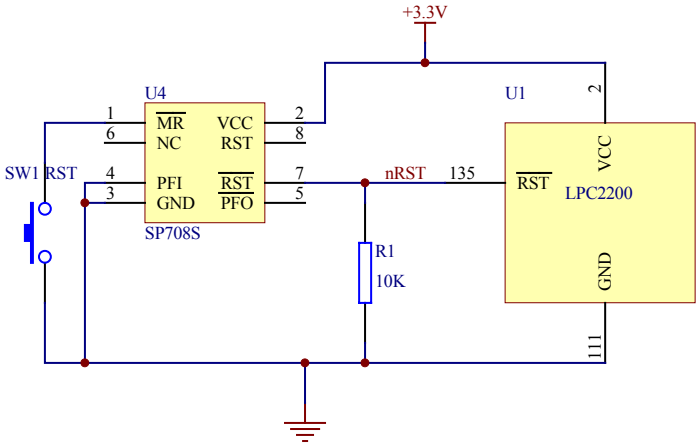


图 6.12 带手动复位的复位电路

微控制器在复位后可能有多种初始状态，具体复位到哪种初始状态是在复位的过程中决定的。复位逻辑可能通过片内只读存储器中的数据决定具体的初始状态，但更多的是通过复位期间的引脚状态决定，也可能通过两者共同决定。用引脚状态配置复位后的初始状态没有统一的方法，需要根据相关芯片的手册决定，图 6.13 就是 LPC2000 复位配置的一个例子。注意，P2.26、P2.27 仅在 LPC2200 系列芯片中存在。

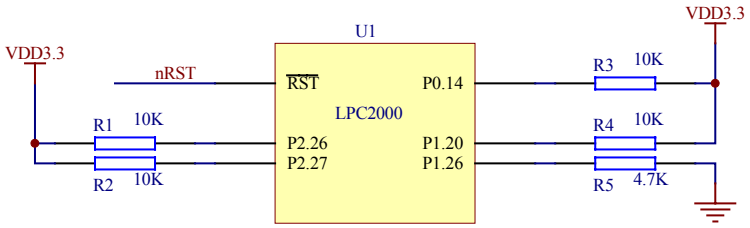


图 6.13 LPC2000 复位配置原理图

6.1.5 存储器系统

对于大部分微控制器来说，存储器系统不是必需的，但如果微控制器没有片内程序存储器或数据存储器时，就必须设计存储器系统，这一般通过微控制器的外部总线接口实现。关于通过微控制器的外部总线的接口方法请参考 6.3 节，这里仅给出 LPC2210 扩展程序存储器的原理图，见图 6.14。

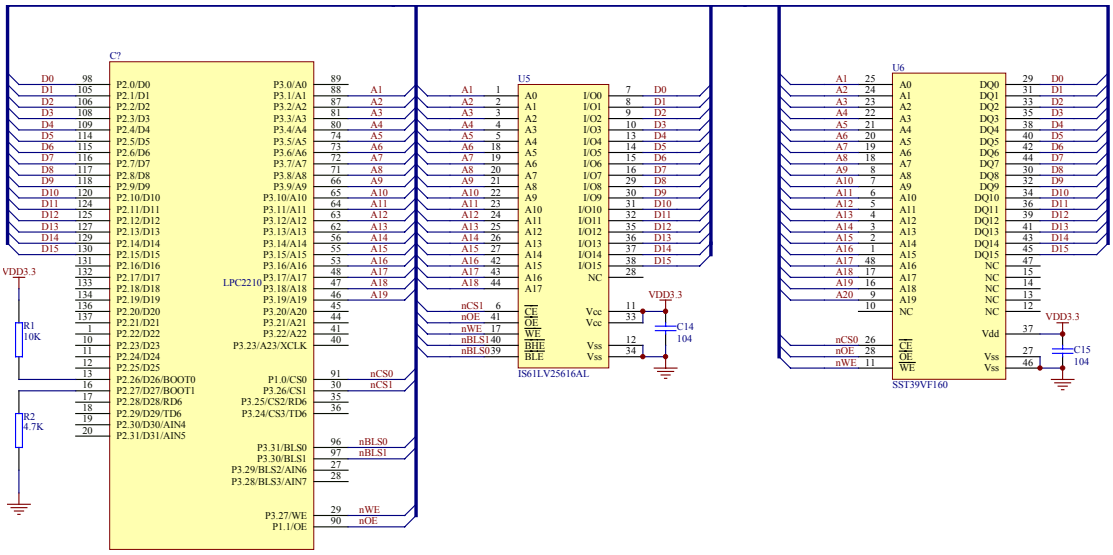


图 6.14 LPC2210 存储器系统实例

6.1.6 调试与测试接口

调试与测试接口不是系统运行必须的，但现代系统越来越强调可测性，调试、测试接口的设计也要重视了。LPC2000 有一个内置 JTAG 调试接口，通过这个接口可以控制芯片的运行并获取内部信息。这部分电路比较简单，见图 6.15 和图 6.16，其它的调试测试接口应根据实际电路而定，例如，简单的就是在适当的地方增加测试点，这里就不举例了。

注意图 6.15 的复位电路与 6.1.4 小结介绍的不一样，它在复位信号和 CPU 之间插入了三态门 74HC125。使用三态门主要是为了复位芯片和 JTAG（ETM）仿真器都可以复位芯片。如果没有 74HC125，当复位芯片输出高电平时，JTAG（ETM）仿真器就不可能把它拉底，这不但不能实现需要的功能，还可能损坏复位芯片或 JTAG（ETM）仿真器。因为这种电路 JTAG（ETM）仿真器对 LPC2000 有完全的控制，其仿真性能最好。不过，由于 74HC125 工作的电压范围低于复位芯片的工作电压范围，这种电路的复位性能低于图 6.16 所示的电路，所以此电路一般用于样机，当产品试生产时应当使用图 6.16 所示的电路。正式产品中可以不需要这部分电路。

还有一点需要注意：图 6.15、图 6.16 所示电路均具有 ETM 接口，但 ETM 功能仅在高级仿真器中具有，如果读者使用的仿真器没有此功能，这个接口可以去掉，同时把接在 TRACESYNC 信号上的电阻也去掉。

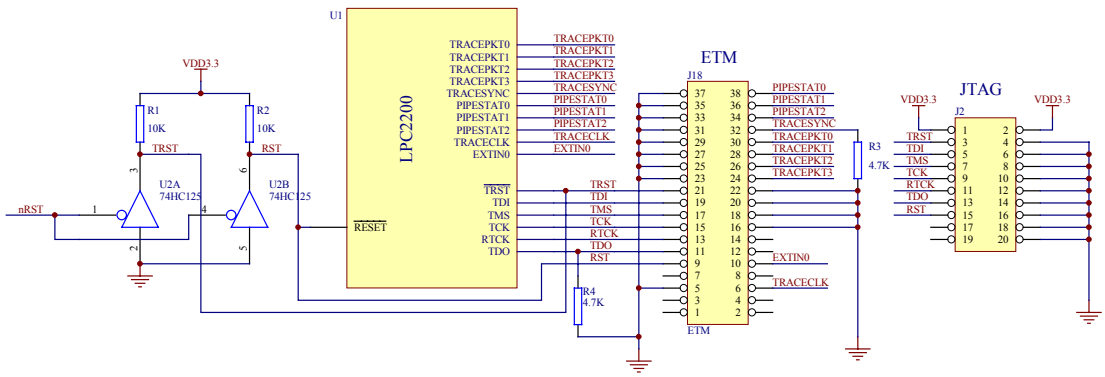


图 6.15 LPC2000 调试接口电路图之一

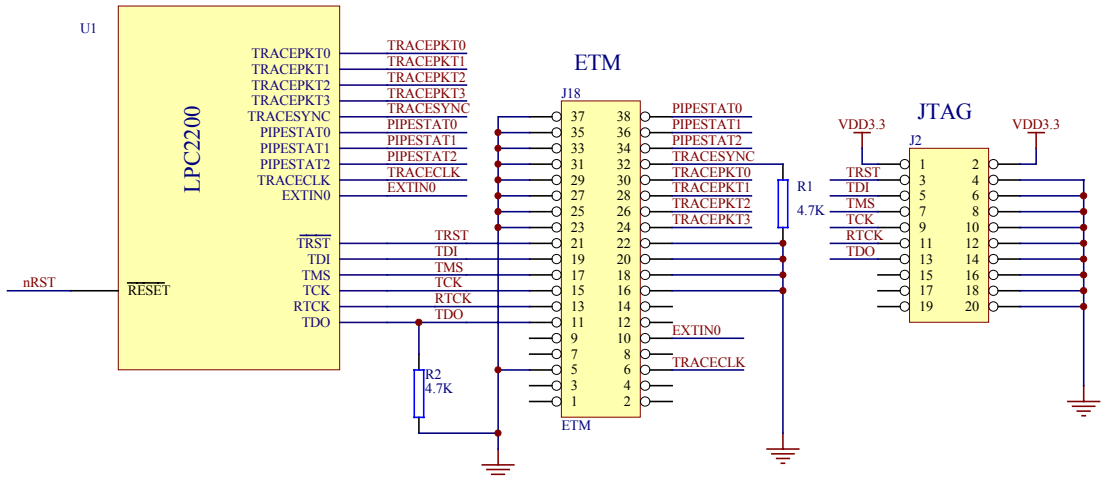


图 6.16 LPC2000 调试接口电路图之二

6.1.7 完整的最小系统

本小节介绍一下 LPC2000 系列 3 个典型芯片的完整的最小系统 LPC2114、LPC2210 和 LPC2214。其中 LPC2114 没有外部总线，但有片内 FLASH（可用于存储程序）和片内 RAM，LPC2210 具有外部总线，但没有片内 FLASH，只有片内 RAM，而 LPC2214 不但有片内 FLASH 和 RAM，还有外部总线。下面分别介绍它们的最小系统应用实例。

1. LPC2114 最小系统

对于 LPC2114 芯片，最小系统需要两组电源、复位电路、晶振电路，P0.14 脚接一个上拉电阻(一个连接到正电源的电阻)禁止 ISP 功能，电路参考图 6.17。

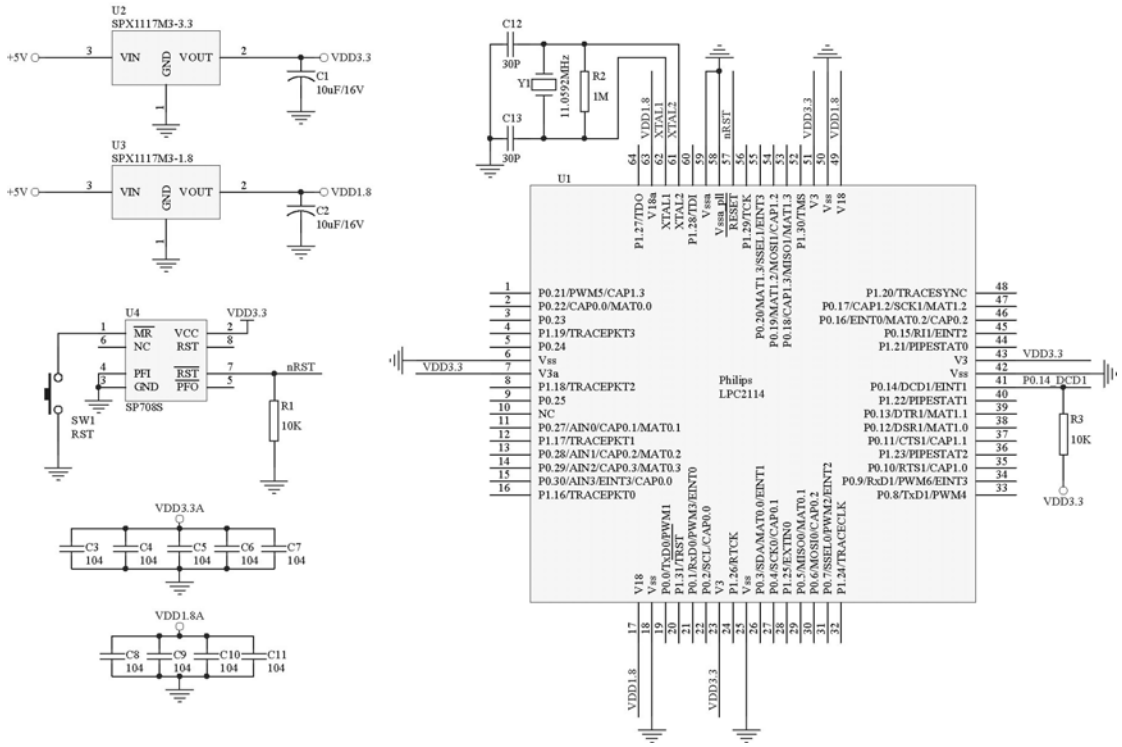


图 6.17 LPC2114 最小系统原理图

2. LPC2210 最小系统

对于 LPC2210 芯片，由于其片内无程序存储器，所以需要外扩 FLASH，还可以扩展静态 RAM。最小系统需要两组电源、复位电路、晶振电路、程序存储器，P0.14 脚接一个上拉电阻(一个连接到正电源的电阻)禁止 ISP 功能，电路参考图 6.18。

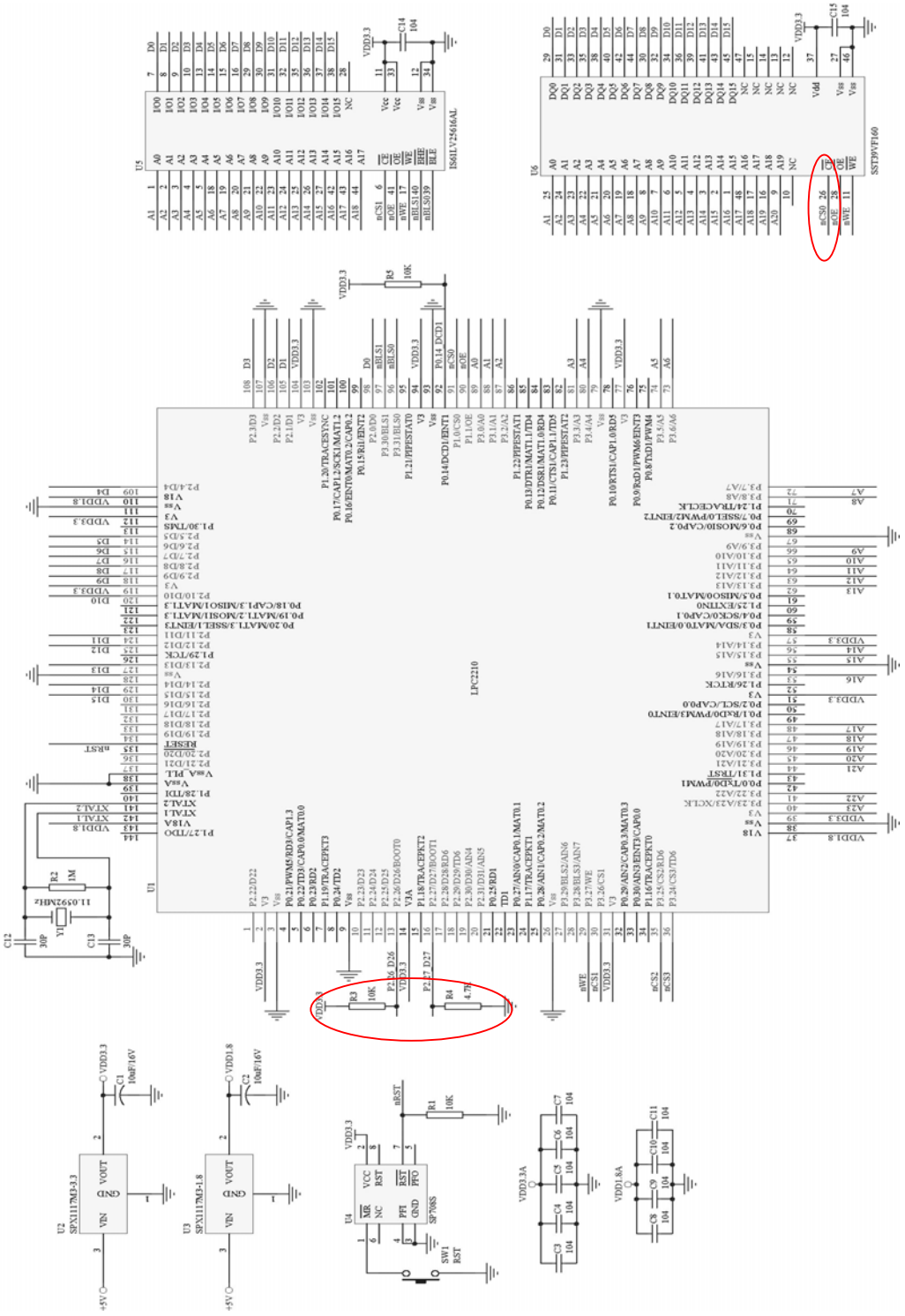


图 6.18 LPC2210 最小系统原理图

LPC2210 需要从外部 Bank0 地址 0x80000000 启动程序，所以 FLASH 地址安排到 Bank0 上，即 FLASH 的片选由 LPC2210 的 CS0 控制。图 6.18 中扩展了总线宽度为 16 位的 FLASH，

所以 D26 引脚要接一个上拉电阻，D27 引脚上要接一个下拉电阻，系统复位后将从 Bank0 启动程序，总线设置为 16 位宽度。

注意，LPC2210 的 I/O 电压为 3.3V，所以外扩存储器的供电电压最好为 3.3V。

3. LPC2214 最小系统

LPC2214 具有片内 FLASH 程序存储器，其最小系统需要两组电源、复位电路、晶振电路，P0.14 脚接一个上拉电阻(一个连接到正电源的电阻)禁止 ISP 功能，电路参考图 6.19。

如图 6.19 所示，D26、D27 引脚均要接一个上拉电阻，系统复位后将从片内 FLASH 程序存储器启动程序，即从 0x0000000 地址处开始运行程序。

虽然用户程序处在于片内 FALSH 中，但是其外部总线还是可以使用的，外部总线可以接片外外设或像图 6.18 那样扩展存储器。

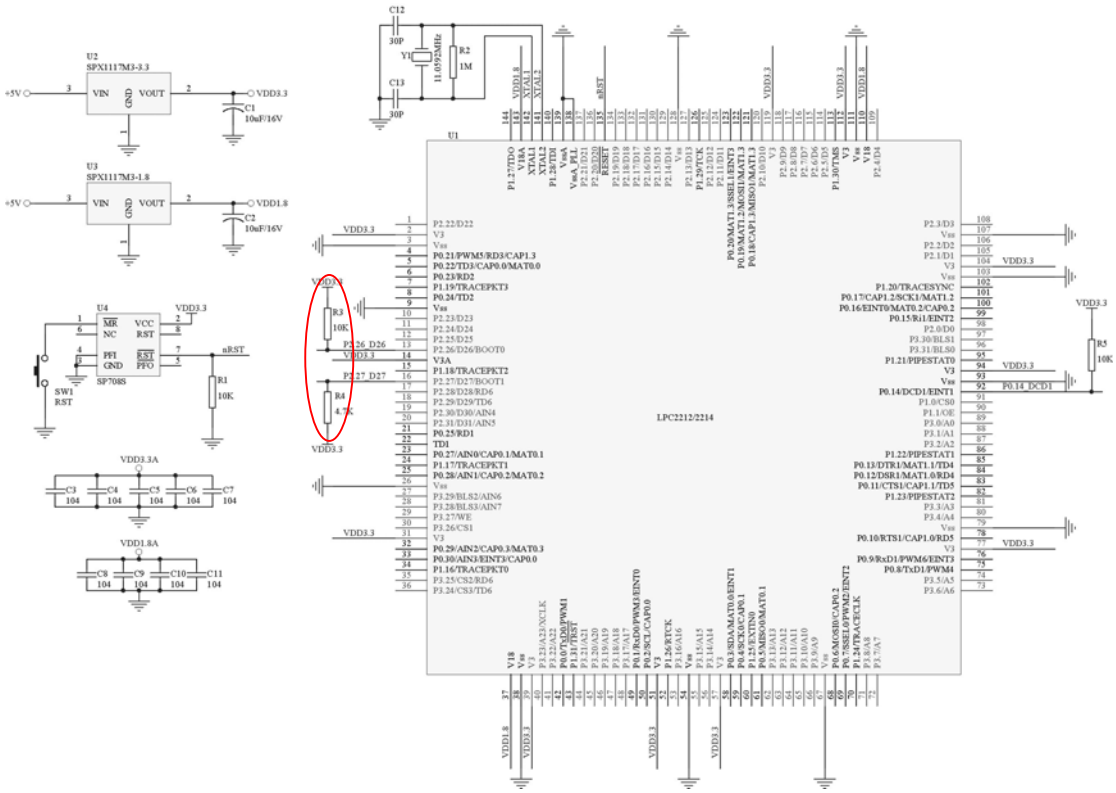


图 6.19 LPC2214 最小系统原理图

6.2 片内外设

6.2.1 GPIO（通用 I/O）

LPC2000 系列的绝大多数 GPIO 为真正的全双向 I/O 口，可以独立控制每一根 I/O 口线的状态是输入还是输出，绝大多数 GPIO 的输出为推挽输出，可以独立控制每一根 I/O 口的输出状态。虽然 LPC2000 系列的 I/O 电压为 3.3V，GPIO 的输出最高为 I/O 口电源电压，但绝大多数 GPIO 能够承受 5V 电压的输入，绝大多数 GPIO 作为输入时是处于高阻状态。

因为 LPC2000 系列的 GPIO 有以上特性，所以可以用它们（通过程序）模拟很多器件的时序达到控制相应器件的目的。下面就介绍 LPC2000 系列 GPIO 的一般用法：

1. 按键

按键为嵌入式系统的输入设备，绝大多数需要人机交互的嵌入式系统都离不开按键。基于 LPC2000 系列微控制器中，使用 GPIO 部件实现按键功能是最简单且低成本的方法。使

用 GPIO 部件实现按键功能通常有两种方法：独立式按键输入及行列式键盘输入。

独立式按键输入编程简单，每个按键都占用一个 GPIO 引脚，如图 6.20 所示。使用时，定义 GPIO 为输入方式，由于每个 GPIO 引脚都接有上拉电阻，所以当没有键按下时，读取 GPIO 状态都为高电平，当有键被按下并读取 GPIO 状态时被按下的 GPIO 引脚为低电平。通过判断 GPIO 引脚电平状态确定按键是否被按下。

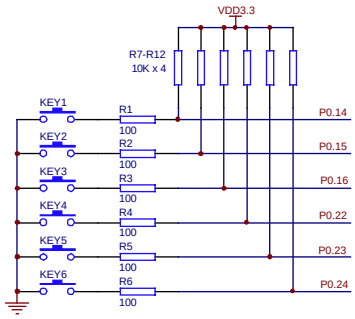


图 6.20 独立式按键输入

如果需要的按键数目较多，而 GPIO 引脚不够时，可以考虑使用行列式键盘输入方式。行列式键盘输入方式使用较少的 GPIO 引脚，可支持较多的按键，其缺点是编程较复杂。如图 6.21 所示，P0.1~P0.4 设置为 GPIO 输出引脚，P0.6~P0.9 设置为 GPIO 输入引脚。P0.1~P0.4 引脚以一定的顺序及频率在同一时间仅使其中一个引脚输出低电平；控制器快速查询 P0.6~P0.9 引脚的电平状态；如果有 1 个键被按下，则在一定时间内 P0.6~P0.9 中将有一个脚为低电平，再查询 P0.1~P0.4 的输出状态，找出这时输出低电平的引脚，就可以很容易地判断出哪个按键被按下。

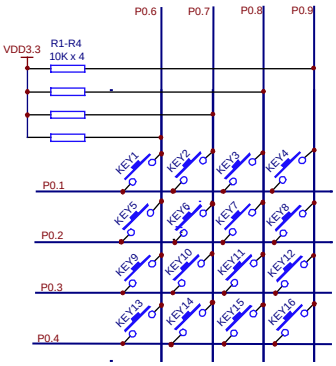


图 6.21 行列式按键输入

2. LED 灯

LED 灯在嵌入式系统常用于作信号灯，指示系统当前的某些状态。LED 的控制很简单，只需在阳极与阴极间提供一个 1.7V 正向电压，并使流经 LED 的电流为 5~10mA，即可以较理想地点亮 LED。如图 6.22 所示，设置 GPIO 引脚为输出方式，使 GPIO 引脚 P0.10 输出低电平时，VDD3.3 与 GPIO 引脚即有 3.3V 的电压差，这时 LDE1 即被点亮；使 GPIO 引脚 P0.10 输出高电平时，VDD3.3 与 GPIO 引脚电压差为 0，这时 LDE1 不能被点亮；电阻 R1~R7 用于限流。如果使用 GPIO 控制较多的 LED 时，需要使用三极管驱动，如图 6.23 所示。

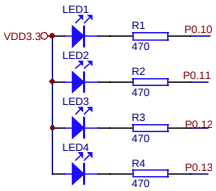


图 6.22 GPIO 直接驱动 LED

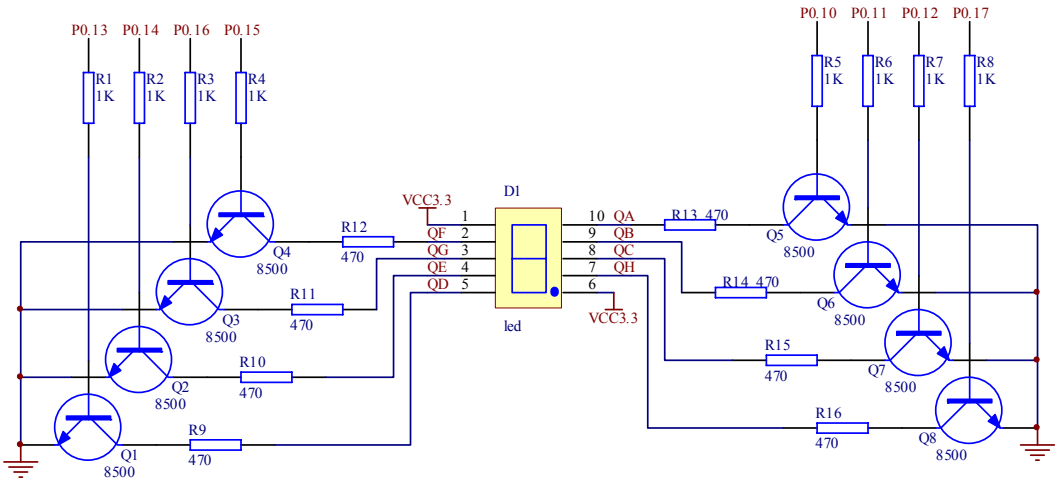


图 6.23 使用三极管驱动 LED 数码管

3. 蜂鸣器

在嵌入式系统中常用的蜂鸣器有直流型和交流型两种。直流型蜂鸣器只需提供额定电压就可以控制蜂鸣器蜂鸣；交流型蜂鸣器则需提供一定频率的交流信号，才可以使蜂鸣器响。直流型蜂鸣器的蜂鸣频率是固定不能更改的，而交流型的则可以通过更改驱动电流的频率来调整蜂鸣频率。两种类型的蜂鸣器都可以使用相同的控制电路，只是控制方式有所不同，如图 6.24 所示。GPIO 提供的输出电流不能够直接驱动蜂鸣器，需经过 PNP 三极管驱动。

设置 GPIO 引脚 P0.7 为输出方式，当设置 P0.7 输出高电平时，三极管 Q1 的发射极与基极的电压差为 0，Q1 截止；当设置 P0.7 输出低电平时， Q1 的发射极与基极的电压差约为 3.3V，Q1 饱和导通，直流型蜂鸣器蜂鸣。对于交流型蜂鸣器，需通过以一定的音频频率改变 P0.7 的输出状态，从而为蜂鸣器提供一定频率的交变信号，使蜂鸣器以一定的频率蜂鸣。

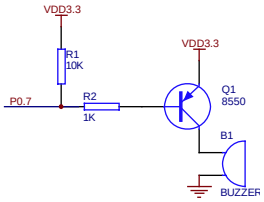


图 6.24 GPIO 控制蜂鸣器

4. 模拟总线

LPC2000 系列部分芯片没有外部总线，当它们需要外接总线设备时就必须用 GPIO 模拟总线了。因为总线需要大量的信号线，而 LPC2000 的 GPIO 资源是宝贵的，所以模拟总线的设计的首要任务是节省 GPIO 的使用量，这就需要地址、数据总线复用了。图 6.25、图 6.26 和图 6.27 分别为 8 位地址、16 位地址和 24 位地址模拟总线的例子，其数据总线均为 8

位。图 6.26 虽然以存储器为例，但对别的总线器件也适用。图中的地址总线（数据总线）使用连续的 GPIO，这是为了编程方便而已，没有特殊的含义。

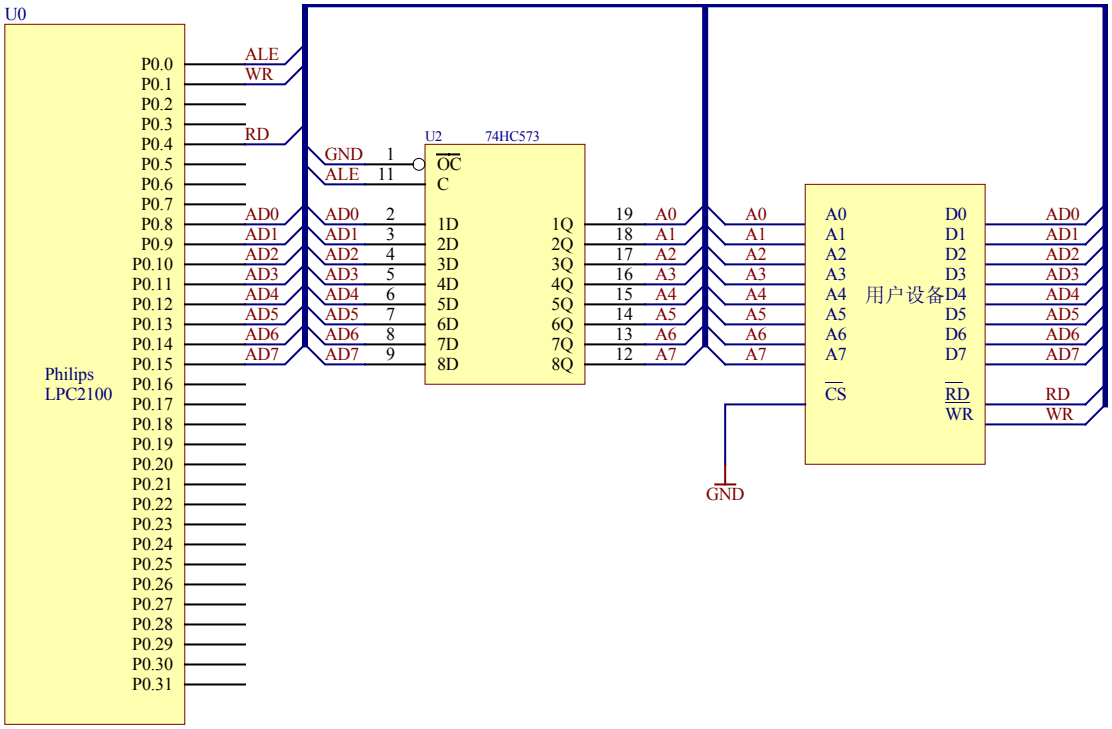


图 6.25　8 位地址的模拟总线的例子

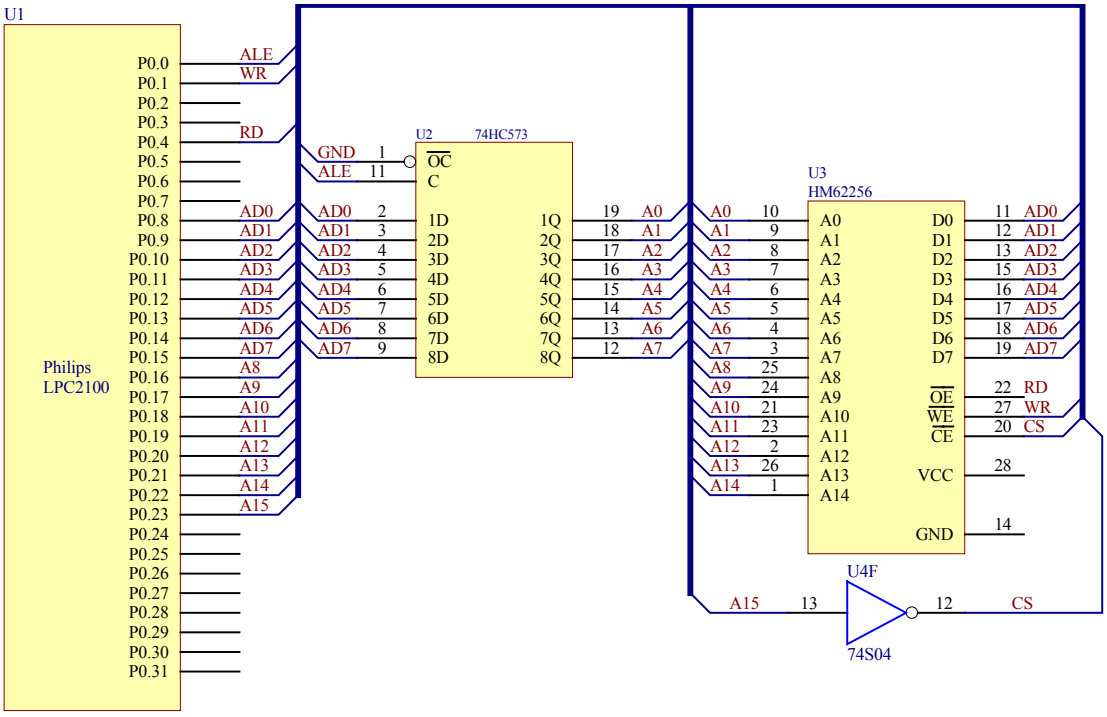


图 6.26　16 位地址的模拟总线的例子

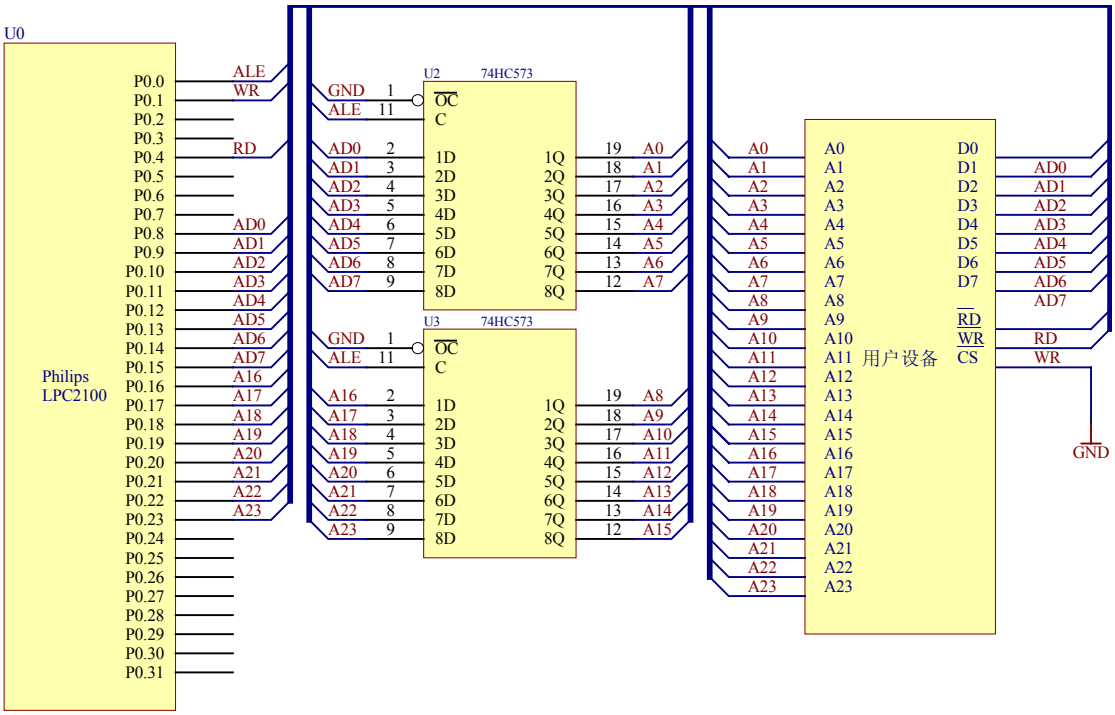


图 6.27 24 位地址的模拟总线的例子

6.2.2 UART、MODEM

1. UART 接口描述

通用异步收发器 UART(即 Universal Asynchronous Receiver and Transmitter)是用硬件实现异步串行通信的通信接口电路。UART 异步串行通信接口是嵌入式系统最常用的接口，可用来与上位机或其它外部设备进行数据通信。

由于 UART 应用的普遍性，所以大部分微控制器内部都集成有 UART 接口，但是不同类型的微控制器的 UART 接口内部电路和操作寄存器并不一定相同。

LPC2000 系列 ARM7 微控制器均具有两个 UART，它们的结构及寄存器符合 16C550 工业标准。

2. UART、16C550 与 RS232 的区别

UART 是通用异步串行通信接口的总称，UART 允许在串行链路上进行全双工的通信，输出/输入的电平为 TTL 电平。一般来说，全双工 UART 定义了一个串行发送引脚(TXD)和一个串行接收引脚(RXD)，可以在同一时刻发送和接收数据。但是不同芯片的 UART 接口内部电路、操作寄存器和工作模式并不一定相同。比如标准 80C51 的 UART 接口，全双工的 UART，但没有波特率发生器(而是使用定时器 1 溢出信号作波特率的时钟)，一个控制寄存器 SCON 和一个串口数据缓冲区 SBUF 寄存器，支持 8 位、9 位数据(第 9 位可当作奇偶校验位)传输模式。

16C550 是一种工业标准的 UART，此类 UART 芯片内部集成了可编程的波特率发生器、发送/接收 FIFO、处理器中断系统和各种线状态错误检测电路等等，并具有完全的 MODEM 控制能力。工作模式为全双工模式，支持 5～8 位数据长度，1/2 位停止位，可选奇偶校验位。

RS232 是美国电子工业协会(EIA)制定的串行通讯标准，又称 RS-232-C(说明，C 代表所公布的版本)。早期它被应用于计算机和调制解调器(MODEM)的连接控制，(MODEM)再通过电话线进行远距离的数据传输。RS232 是一个全双工的通讯标准，它可以同时进行数据接收和发送的工作。RS232 标准包括一个主通道和一个辅助通道，在多数情况下主要使用主通道，即 RXD、TXD、GND 信号。

严格地讲 RS232 接口是 DTE（数据终端设备）和 DCE（数据通信设备）之间的一个接口，DTE 包括计算机、终端、串口打印机等设备。DCE 通常只有 MODEM 和某些交换机等。

RS-232-C 标准采用的接口是 9 针(DB9)或 25 针(DB25)的 D 型插头，常用的 9 针 D 型插头的引脚定义如表 6.1 所示。

表 6.1 RS232 接口定义(9 针)

引脚	符号	功能
1	DCD	数据载波检测
2	RXD	接收数据
3	TXD	发送数据
4	DTR	数据终端准备就绪
5	GND	信号地
6	DSR	数据设备准备就绪
7	RTS	请求发送
8	CTS	清除发送
9	RI	振铃指示

在电气特性上，RS232 标准采用负逻辑方式，标准逻辑“1”对应－5V～－15V 电平，标准逻辑“0”对应+5V～+15V 电平。因此 UART 的 TTL 电平需要进行 RS232 电平转换后，才能与 RS232 接口连接并通讯，可以使用 SP3232E 或 SP3243ECA 芯片进行电平转换。

3. LPC2000 的 UART 接口

LPC2000 系列 ARM7 微控制器包含有两个 UART 接口，分别为 UART0 和 UART1，它们的结构及寄存器符合 16C550 工业标准。说明：UART0 没有完整的 MODEM 接口信号，仅提供 TXD、RXD 信号引脚。在大多数异步串行通讯的应用中，并不需要完整的 MODEM 接口信号(辅助控制信号)，而只使用 RXD、TXD 和 GND 信号即可。

使用 UART 时，数据发送/接收时序参考图 6.28，数据位的宽度是由波特率而定。

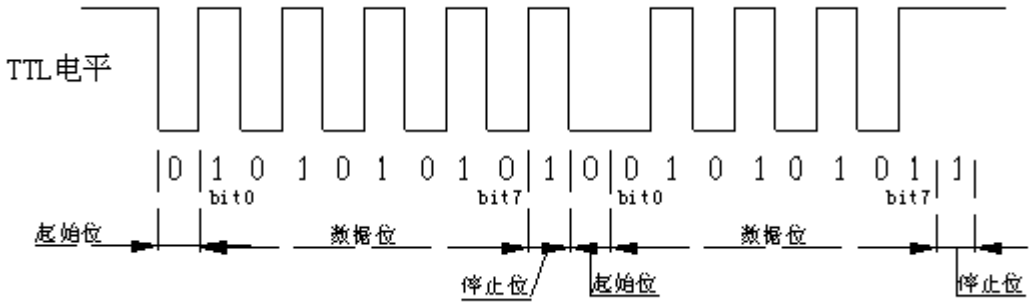


图 6.28 串行数据时序(55H、AAH)—TTL

如果要使用 UART0 与 RS232 接口的设备进行基本的通讯，那么就需要一个 RS232 转换器将 TTL 电平转换成 RS232 电平，电路如图 6.29 所示。RS232 电平的数据发送/接收时序如图 6.30 所示。

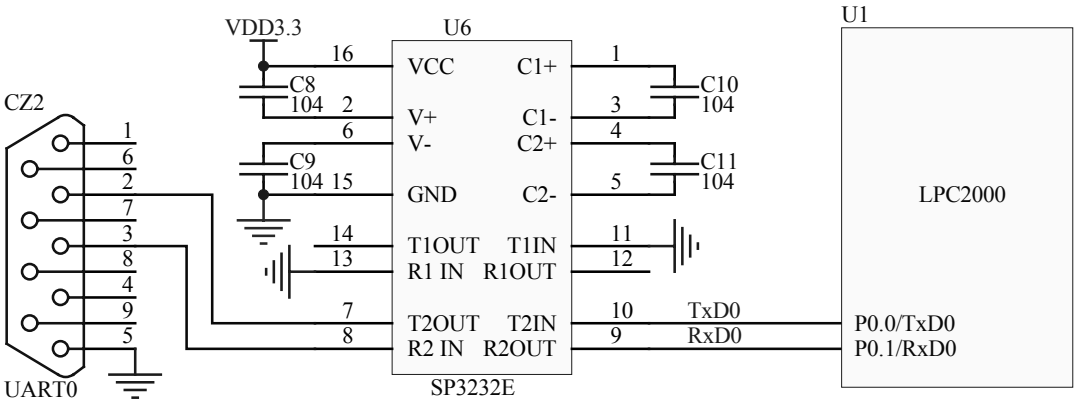


图 6.29 RS232 电平转换电路

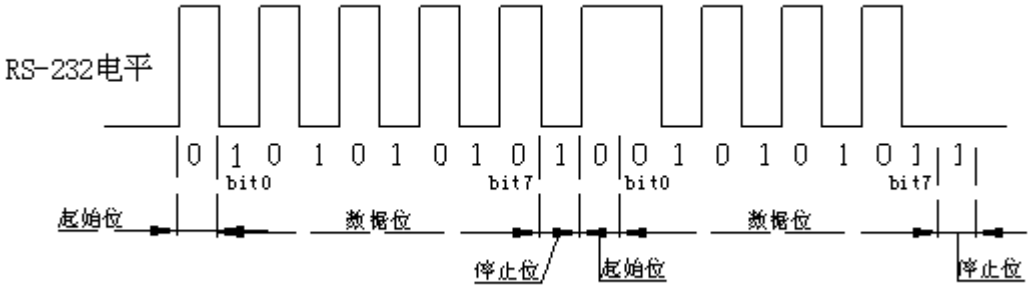


图 6.30 串行数据时序(55H、AAH)—RS232

4. LPC2000 的 MODEM 接口

LPC2000 系列 ARM7 微控制器的 UART1 带有完整的调制解调器(MODEM)接口，只要使用 SP3243ECA 转换芯片将信号转换成 RS232 电平，即可与 MODEM 连接，控制 MODEM 拨号、通讯等等。电路如图 6.31 所示。

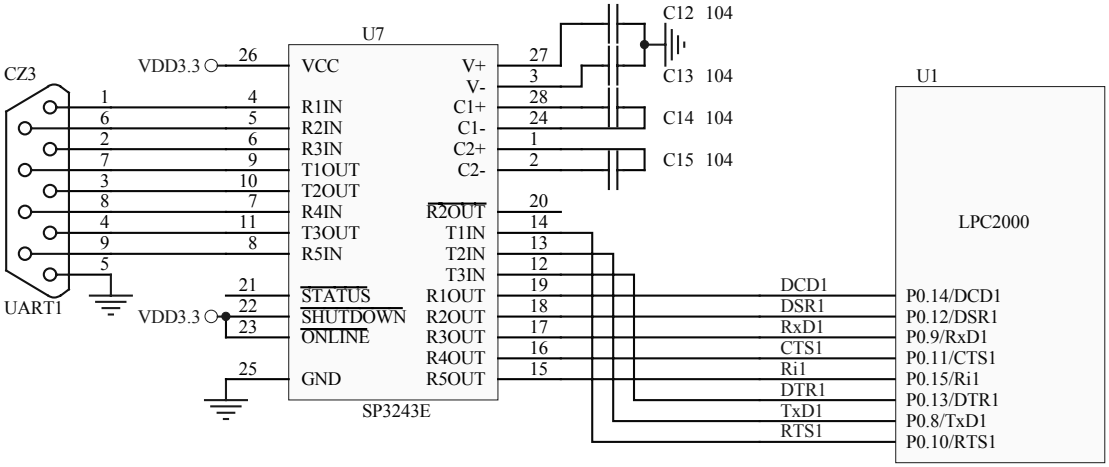


图 6.31 MODEM 接口电路原理

6.2.3 I²C

1. I²C 总线规范简介

I²C BUS（Inter IC BUS）是 Philips 推出的芯片间串行传输总线，它以 2 根连线实现了完善的全双工同步数据传送，可以极方便地构成多机系统和外围器件扩展系统。I²C 总线采用了器件地址的硬件设置方法，通过软件寻址完全避免了器件的片选线寻址方法，从而使硬件系统具有最简单而灵活的扩展方法。

I²C 总线的 2 根线——串行数据（SDA）和串行时钟（SCL）——连接到总线上的任何一个器件，每个器件都应有一个唯一的地址，而且都可以作为一个发送器或接收器。此外，器件在执行数据传输时也可以被看作是主机或从机。

发送器：本次传送中发送数据（不包括地址和命令）到总线的器件。

接收器：本次传送中从总线接收数据（不包括地址和命令）的器件。

主　机：初始化发送、产生时钟信号和终止发送的器件，它可以是发送器或接收器。主机通常是微控制器。

从　机：被主机寻址的器件，它可以是发送器或接收器。

I²C 总线应用系统的典型结构如图 6.32 所示。在该结构中，微控制器 A 可以做为该总线上的唯一主机，其它的器件全部是从机。而另一种方式是微控制器 A 和微控制器 B 都作为总线上的主机。

因此，I²C 总线是一个多主机的总线，也即可以连接多于一个能控制总线的器件到总线。当 2 个以上能控制总线的器件同时发动传输时，只能有一个器件能真正控制总线而成为主机，并使报文不被破坏，这个过程叫仲裁。与此同时，能使多个能控制总线的器件产生时钟信号的同步。

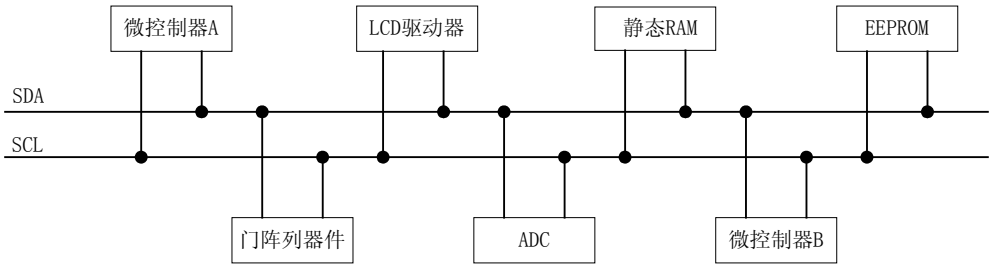


图 6.32 I²C 总线应用系统典型结构

SDA 和 SCL 都是双向线路。连接到总线的器件的输出级必须是漏极开路或集电极开路，都通过一个电流源或上拉电阻连接到正的电源电压，这样才能够实现线与功能。当总线空闲时，这 2 条线路都是高电平。

在标准模式下，总线数据传输的速度为 0 ~ 100kbit/s，在高速模式下，可达 0 ~ 400kbit/s。

2. I²C 总线上的位传输

I²C 总线上每传输一个数据位必须产生一个时钟脉冲。

（1）数据的有效性

SDA 线上的数据必须在时钟线 SCL 的高电平期间保持稳定，数据线的电平状态只有在 SCL 线的时钟信号为低电平时才能改变，如图 6.33 所示。在标准模式下，高低电平宽度必须不小于 4.7μS。

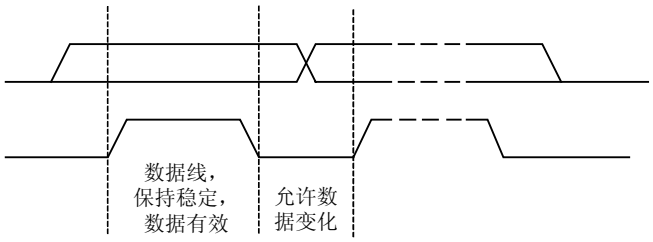


图 6.33 I^2C 总线的位传输

（2）起始和停止信号

在 I^2C 总线中，唯一违反上述数据有效性的是起始 （S）和停止（P）信号，如图 6.34 所示。

起始信号（重复起始信号）：在 SCL 线为高电平时，SDA 线从高电平向低电平切换。

停止信号：在 SCL 线为高电平时，SDA 线由低电平向高电平切换。

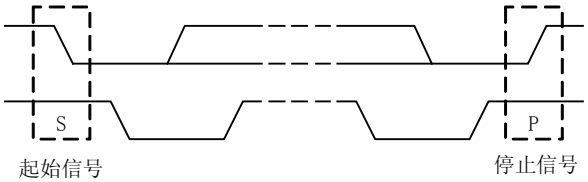


图 6.34 I^2C 总线的起始信号和停止信号

起始和停止信号一般由主机产生。起始信号作为一次传送的开始，在起始信号后总线被认为处于忙的状态。停止信号作为一次传送的结束，在停止信号的某段时间后，总线被认为再次处于空闲状态。重复起始信号既作为上次传送的结束，也作为下次传送的开始。

3．数据传输

（1）字节格式

发送到 SDA 线上的每个字节必须为 8 位。每次传输可以发送的字节数量不受限制。每个字节后必须跟一个应答位。首先传输的是数据的最高位（MSB）（见图 6.35 所示）。

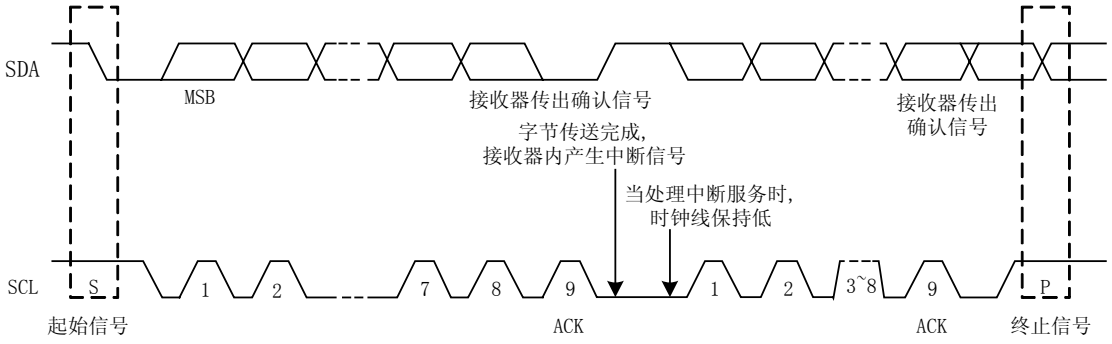


图 6.35 I^2C 总线的数据传输

（2）应答

相应的应答时钟脉冲由主机产生。在应答的时钟脉冲期间，发送器释放 SDA 线（高）。在应答的时钟脉冲期间，接收器必须将 SDA 线拉低，使它在这个时钟脉冲的高电平期间保持稳定的低电平。如图 6.35 中时钟信号 SCL 的第 9 位。

一般说来，被寻址匹配的从机（可继续接收下一个字节的接收器）将产生一个应答位。

如果作为发送器的主机在发送完一个字节后，没有收到应答位（或收到一个非应答位），或者作为接收器的主机没有发送应答位（或发送一个非应答位），那么主机必须产生一个停止信号或重复起始信号来结束本次传输。

若从机（接收器）不能接收更多的数据字节，将不产生这个应答位；主机（接收器）在接收完最后一个字节后不产生应答，通知从机（发送器）数据传输结束。

4．仲裁与时钟发生

（1）同步

时钟同步是通过各个能产生时钟的器件线连接到 SCL 线上来实现的，上述的各个器件可能都有自己独立的时钟，各个时钟信号的频率、周期、相位和占空比可能都不相同，由于“线与”的结果，在 SCL 线上产生的实际时钟的低电平宽度由低电平持续时间最长的器件决定，而高电平宽度由高电平持续时间最短的器件决定。

（2）仲裁

当总线空闲时，多个主机同时启动传输，可能会有不止一个主机检测到满足起始信号，而同时获得主机权，这样就要进行仲裁。当 SCL 线是高电平时，仲裁在 SDA 线发生，当其他主机发送低电平时，发送高电平的主机将丢失仲裁，因为总线上的电平与它自己的电平不同。

仲裁可以持续多位，它的第一个阶段是比较地址位，如果每个主机都尝试寻址相同的器件，仲裁会继续比较数据位，或者比较响应位。因为 I^2C 总线的地址和数据信息由赢得仲裁的主机决定，在仲裁过程中不会丢失信息。

（3）用时钟同步机制作为握手

器件可以快速接收数据字节，但可能需要更多时间保存接收到的字节或准备一个要发送的字节。此时，这个器件可以使 SCL 线保持低电平，迫使与之交换数据的器件进入等待状态，直到准备好下一字节的发送或接收。

5．传输协议

（1）寻址字节

主机产生起始信号后，发送的第一个字节为寻址字节，该字节的头 7 位（高 7 位）为从机地址，最低位（LSB）决定了报文的方向，“0”表示主机写信息到从机，“1”表示主机读从机中的信息，如图 6.36 所示。当发送了一个地址后，总线上的每个器件都将头 7 位与它自己的地址比较。如果一样，器件就会应答主机的寻址，至于是从机（接收器）还是从机（发送器）都由 R/$\overline{W}$ 位决定。

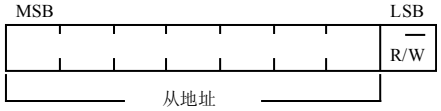


图 6.36 起始信号后的第一个字节

从机地址由一个固定的和一个可编程的部分构成。例如，某些器件有 4 个固定的位（高 4 位）和 3 个可编程的地址位（低 3 位），因此同一总线上共可以连接 8 个相同的器件。I^2C 总线委员会协调 I^2C 地址的分配，保留了 2 组 8 位地址（0000XXX 和 1111XXX），这 2 组地址的用途可查阅相关资料。

（2）传输格式

主机产生起始信号后，发送一个寻址字节，收到应答后紧跟着的就是数据传输，数据传输一般由主机产生的停止位终止。但是，如果主机仍希望在总线上通讯，它可以产生重复起

始信号（Sr）和寻址另一个从机，而不是首先产生一个停止信号。在这种传输中，可能有不同的读/写格式结合。可能的数据传输格式有：

主机（发送器）发送数据到从机（接收器）。如图 6.37 所示，寻址字节的“R/$\overline{W}$”位为 0，数据传输的方向不改变。寻址字节后，主机（接收器）立即读从机（发送器）中的数据。而图 6.38 中，寻址字节的“R/$\overline{W}$”位为 1，在第一次从机产生的响应时，主机（发送器）变成主机（接收器），从机（接收器）变成从机（发送器）。之后，数据由从机发送，主机接收，每个应答由主机产生，时钟信号 CLK 仍由主机产生。若主机要终止本次传输，则发送一个非应答信号（$\overline{A}$），接着主机产生停止信号。

复合格式，见图 6.39。传输改变方向的时侯，起始信号和从机地址都会被重复。但 R/$\overline{W}$ 位取反。如果主机（接收器）发送一个重复起始信号，它之前应该要发送一个非应答信号（$\overline{A}$）。

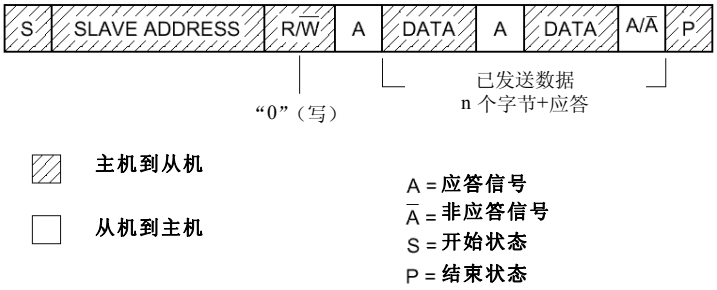


图 6.37 主机（发送器）发送数据到从机（接收器），传输方向不变。

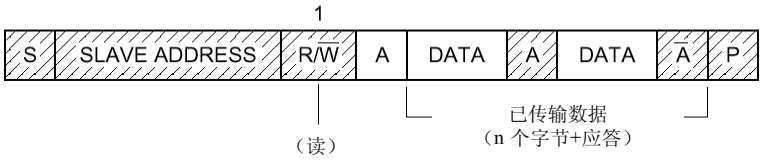


图 6.38 寻址字节后，主机（接收器）立即读从机（发送器）中的数据。

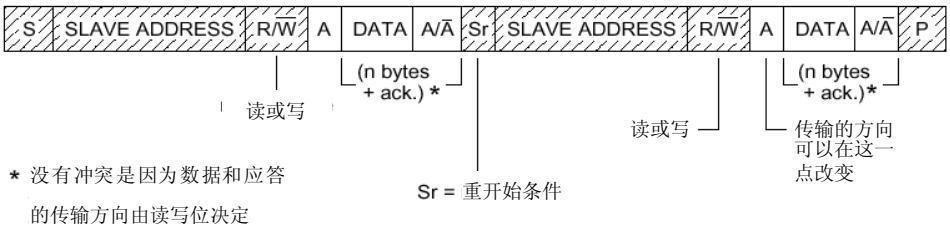


图 6.39 复合格式

6．常用 I²C 器件简介

随着 I²C 总线技术的推出。很多电子厂商都推出了许多带 I²C 总线接口的器件，大量应用于视频、音像及通讯等领域。表 6.2 给出了常用的通用 I²C 接口的种类、型号及寻址字节。

表 6.2 常用 I²C 接口通用器件的种类、型号及寻址字节

种　　类	型　　号	器件地址及寻址字节
128B E²PROM	CAT24WC01	3. 0 1 0 A2 A1 A0 R/$\overline{W}$
256B E²PROM	CAT24WC02	（2） 0 1 0 A2 A1 A0 R/$\overline{W}$
512B E²PROM	CAT24WC04	（3） 0 1 0 A2 A1 a8 R/$\overline{W}$
1024B E²PROM	CAT24WC08	（4） 0 1 0 A2 a9 a8 R/$\overline{W}$
2048B E²PROM	CAT24WC16	（5） 1 0 1 0 a10 a9 a8 R/$\overline{W}$
实时时钟 / 日历芯片	PCF8563	读：0A3H　　写：0A2H
键盘及 LED 驱动器	ZLG7290	从地址：070H
带 32×4 位 RAM 低复用率的通用 LCD 驱动器	PCF8562	只写：0 1 1 1 0 0 SA0 R/$\overline{W}$ （SA0 为该器件的引脚）
通用低复用率 LCD 驱动器	PCF8576D	只写：0 1 1 1 0 0 SA0 R/$\overline{W}$ （SA0 为该器件的引脚）
内嵌 I²C 总线、E²PROM、RESET、WDT 功能的电源监控器件	CAT1161/2	1 0 1 0 a10 a9 a8 R/$\overline{W}$

注：1．A0 A1 和 A2 对应器件的管脚 1、2 和 3

2．A8 A9 和 A10 对应存储阵列地址字地址

下面简单介绍表 6.2 中两个 I²C 器件 CAT24WC02 和 ZLG7290，并给出应用这两个 I²C 器件的一个例子。

（1）ZLG7290 键盘和 LED 驱动器

ZLG7290提供了I²C串行接口和键盘中断信号方便与处理器连接；可驱动8位共阴数码管或64只独立LED和64个按键, 可控扫描位数以及可控任一数码管闪烁，提供数据译码和循环移位段寻址等控制，58个功能键可检测任一键的连击次数，无需外接元件即可直接驱LED即可扩展驱动电流和驱动电压。图6.40为ZLG7290的引脚排列。

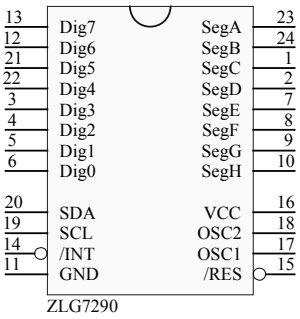


图 6.40　ZLG7290 引脚排列

ZLG7290 各功能引脚对应的功能如表 6.3 所示。

表 6.3　ZLG7290 引脚说明

引脚号	引脚名称	引脚属性	引脚描述
13,12,21,22,3~6	Dig7~ Dig0	输入/输出	LED 显示位驱动及键盘扫描线
10~7,2,1,24,23	SegH~SegA	输入/输出	LED 显示段驱动及键盘扫描线
20	SDA	输入/输出	I²C 总线接口数据/地址线

接上表

引脚号	引脚名称	引脚属性	引脚描述
19	SCL	输入/输出	I^2C 总线接口时钟线
14	/INT	输出	中断输出端，低电平有效
15	/RES	输入	复位输入端，低电平有效
17	OSC1	输入	连接晶体以产生内部时钟
18	OSC2	输出	
16	VCC	电源	电源正（3.3～5.5V）
11	GND	电源	电源地

ZLG7290 更详细的信息及应用例子请到 http://www.zlgmcu.com 网站下载 ZLG7290 的使用手册。

（2）E^2PROM 器件 CAT24WC02

CAT24WC02 是一款 I^2C 总线接口的 E^2PROM 器件，其引脚如图 6.41 所示。各引脚的功能如表 6.4 所示。

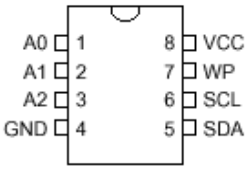


图 6.41 CAT24WC02 芯片引脚

表 6.4 CAT24WC02 引脚描述

引脚	功能描述
A0 A1 A2	器件地址选择
SDA	串行数据/地址
SCL	串行时钟
Vcc	+1.8V 6.0V 工作电压
GND（或 Vss）	电源地

CAT24WC02 的 I^2C 总线地址由引脚 A2 A1 A0 的电平决定，CAT24WC02 的地址的高 4 位固定为 1010，低四位由 A2 A1 A0 决定。当 A2 A1 A0 引脚悬空时，默认值为 0。

CAT24WC02 更详细的信息及应用例子请到 http://www.zlgmcu.com 网站下载 CAT24WC02 的使用手册。

7. I^2C 总线的一个例子

介绍完以上两个 I^2C 器件，下面给出这两个器件与 ARM LPC2000 系列微控制器 I^2C 总线连接的电路原理。

LPC2000 系列微控制器都提供了硬件 I^2C 总线接口和 I^2C 总线控制器，LPC2000 的 I^2C 总线有以下特性：

- 标准的 I^2C 总线接口
- 可配置为主机、从机或主/从机
- 可编程时钟实现通用速率控制
- 主机从机之间双向数据传输

- 多主机总线（无中央主机）
- 同时发送的主机之间进行仲裁，避免了总线数据的冲突
- 在高速模式下，数据传输的速度为 0～400kbit/s

由于 LPC2000 微控制器的 SDA 和 SCL 端口为开漏输出，所以必须在 SDA 和 SCL 线上分别外接一个上拉电阻。

如图 6.42 所示，可以利用 LPC2000 微控制器作为 I^2C 总线的主机，在总线上挂接 I^2C 器件。该总线上挂接着两个 I^2C 器件作为从机，分别为 E^2PROM 器件 CAT24WC02 和键盘和 LED 驱动器 ZLG7290。R46 和 R48 即为 I^2C 总线上的两个上拉电阻。

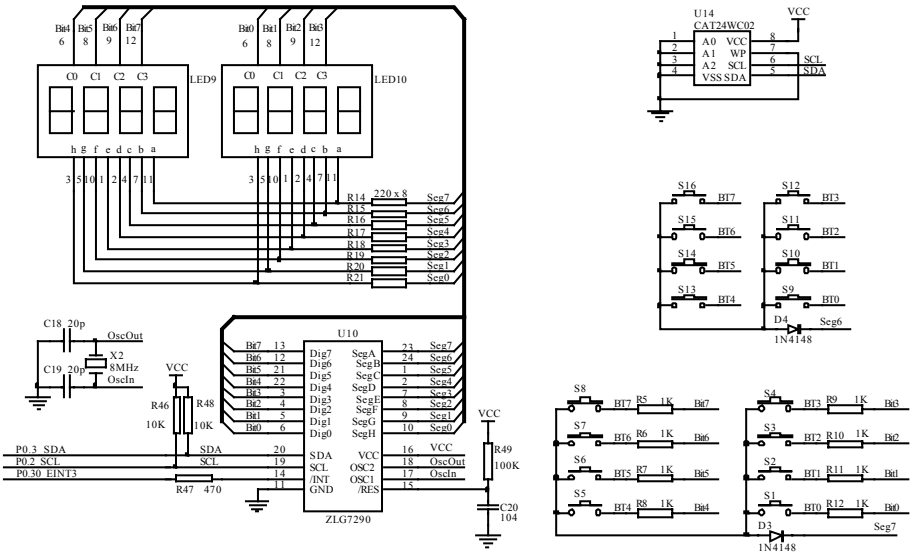


图 6.42 利用 LPC2000 微控制器构成的 I^2C 总线电路

由于一条 I^2C 总线上可以挂接多个器件，因此，LPC2000 可以访问该总线上的这 2 个器件。至于访问哪一个器件是通过器件的地址决定。图 6.42 中的 CAT24WC02 的地址为 0x0A，而 ZLG7290 的地址固定为 0x70。

6.2.4 SPI

SPI（Serial Peripheral Interface——串行外设接口）总线系统是一种同步串行外设接口，允许 MCU 与各种外围设备以串行方式进行通信、数据交换。外围设备包 FLASHRAM、A/D 转换器、网络控制器、MCU 等。SPI 系统可直接与各个厂家生产的多种标准外围器件直接接口，一般使用 4 条线：串行时钟线 SCK、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和低电平有效的从机选择线 SSEL（有的 SPI 接口芯片带有中断信号线 INT，有的 SPI 接口芯片没有主机输出/从机输入数据线 MOSI）。所以，SPI 系统总线一共只需 3~5 位数据线和控制线即可实现与具有 SPI 的各种 I/O 器件接口。各信号含义如下：

SCK 串行时钟，用于同步 SPI 接口间数据传输的时钟信号。该时钟总是由主机驱动并且从机接收。

SSEL 从机选择，SPI 从机选择信号是一个低有效信号，用于指示被选择参与数据传输的从机。每个从机都有各自特定的从机选择输入信号。在数据处理之前，SSEL 必须为低电平并在整个处理过程中保持低电平。如果在数据传输中 SSEL 信号变为高电平，传输中止。

MISO 主入从出，该信号是一个单向的信号，它将数据从从机传输到主机。当器件为从机时，串行数据从该端口输出；当器件为主机时，串行数据从该端口输入；当从机没有被选择时，将该信号驱动为高阻态。

MOSI 主出从入，该信号是一个单向的信号，它将数据从主机传输到从机。当器件为主

机时，串行数据从该端口输出；当器件为从机时，串行数据从该端口输入。

将数据写到 SPI 发送缓冲区后，时钟信号 SCK 的 1 次作用对应一位数据的发送(MISO)和另一位数据的接收（MOSI）；如图 6.43 所示，在主机中数据从移位寄存器中自左向右发出送到从机（MOSI），同时从机中的数据自右向左发到主机（MISO），经过 8 个时钟周期完成 1 个字节的发送。输入字节保留在移位寄存器中，然后从接收缓冲区中读出一个字节的数据。

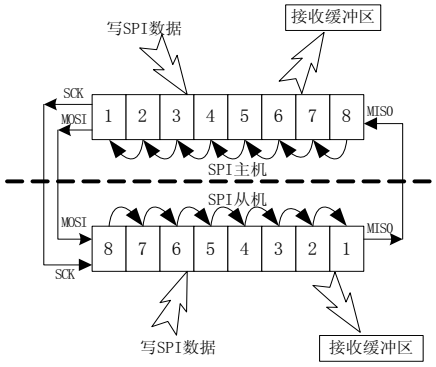


图 6.43 主机和从机的发送和接收

SPI 总线可在软件的控制下构成各种简单的或复杂的系统，如：1 个主 MCU 和几个从 MCU；几个从 MCU 相互连接构成多主机系统（分布式系统）；1 个主 MCU 和 1 个或几个从 I/O 设备。在大多数应用场合中，使用 1 个 MCU 作为主机，它控制数据向 1 个或几个从外围器件的传送。从器件只能在主机发命令时才能接收或向主机传送数据。其数据的传输格式通常是高位（MSB）在前，低位（LSB）在后，在一些增强型的 MCU 高位在前或低位在前都是可通过软件设置的，如 LPC2000 系列微控制器。

SPI 接口总线配置灵活，可用于单主机单从机配置、单主从互换配置、单主多从配置和多主从配置。图 6.44 所示为 SPI 单主单从机配置，由于只有一个主机和从机，所以可以直接使主机的 SSEL 脚接上拉电阻，从机的 SSEL 引脚接地。

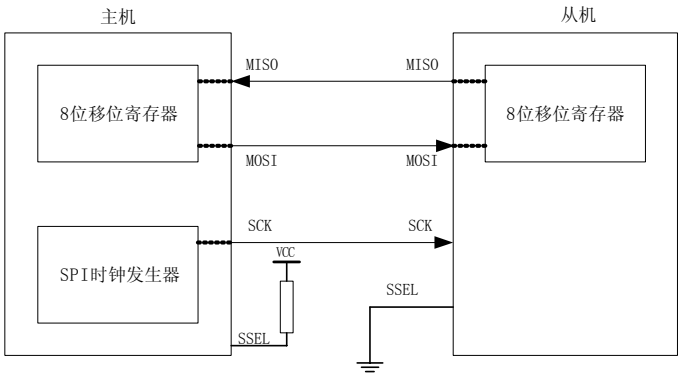


图 6.44 SPI 单主机单从机配置

图 6.45 所示为 SPI 主从互换配置，当没有发生 SPI 操作时，两个器件都可配置为主机；当其中一个器件启动传输时，它可将 SSEL 配置为 GPIO 输出，并使其输出低电平强制另一个器件变为从机。图 6.46 为单主多从机配置，主机使用 GPIO 引脚控制各个从机的 SSEL，从而进行多从机的寻址。图 6.47 为多主多从互换配置，其原理与单主从互换配置相似。空闲时各 SPI 器件都配置为主机，当某一主机需要传输数据时，通过 GPIO 端控制其它 SPI 主机的 SSEL 引脚为低电平，从而强制其它的 SPI 器件变为从机。

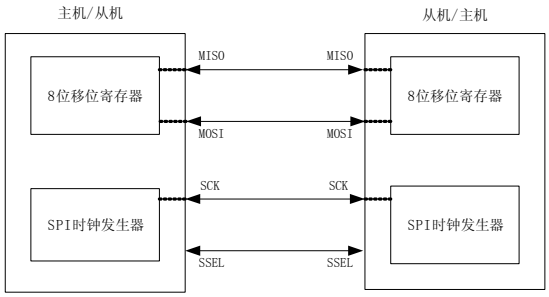


图 6.45 单主从互换配置

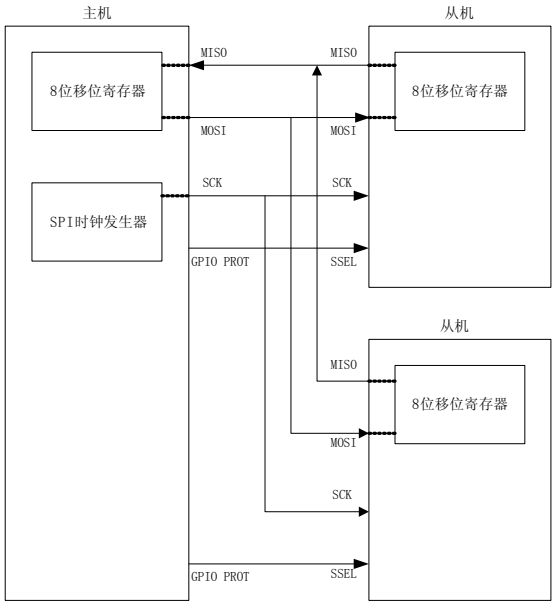


图 6.46 单主机多从机配置

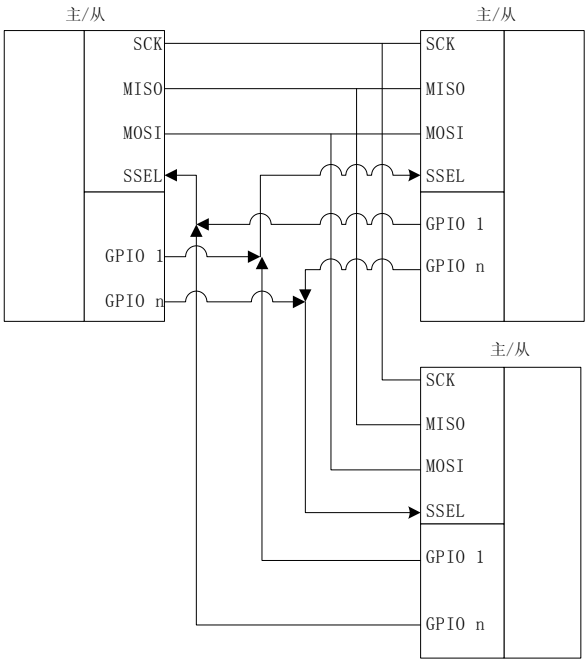


图 6.47 多主多从互换配置

SPI是一个串行输入输出的接口，使用串转并的接口芯片可以实现扩展IO口，常用的串转并芯片有74HC595、74LS164等。图6.48为74HC595逻辑图，SI为串行数据输入引脚，SCK为移位寄存器的时钟输入，/SCLR为清移位寄存器引脚，RCK为锁寄存器锁存时钟引脚，/OE是输出使能引脚，SQ_H为串行数据输出引脚，Q_A~Q_H引脚为并行输出。如表6.5所示，当/SCLR无效（接高电平）、/OE使能（接低电平）时，SCK产生一个上升沿，SI引脚当前电平值将在移位寄存器中左移1位，在下一个上升沿到来时移位寄存器中的所有位都会向左移1位，同时SQ_H引脚也会串行输出移位寄存器中高位的值；当RCK产生上升沿时，移位寄存器的值将会被锁存到锁存器里，并从Q_A~Q_H引脚输出。

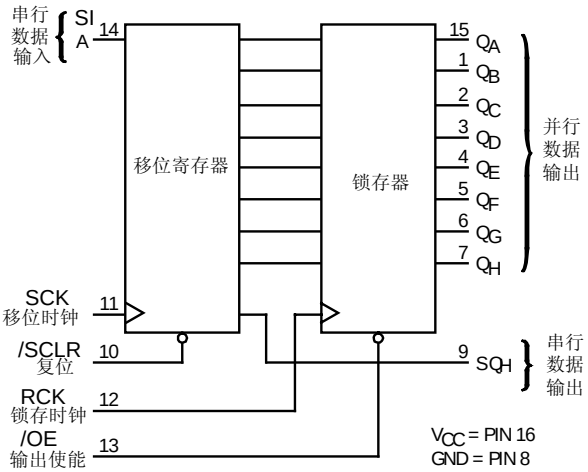


图 6.48 74HC595 逻辑图

表 6.5 74HC595 真值表

输入					输出	
SCK	RCK	/OE	/CSLR	SI	SQ_H	Qn
X	X	L	↓	X	L	NC
X	↑	L	L	X	L	L
X	X	H	L	X	L	Z
↑	X	L	H	H	Q_G'	NC
X	↑	L	H	X	NC	Qn'
↑	↑	L	H	X	Q_G'	Qn'

注：H：高电平；L：低电平；↓：下降沿；↑：上升沿；X：任意态；
Q_X' 表示为Q_X之前的一个状态。

图 6.49 为 SPI 接口与 74HC595 的连接原理图，在 SPI 输出一个字节的数据时 SSEL 产生一个低电平，SPI 主机串行地发该字节的各个位，各个位都依次被锁存在 74HC595 的移位寄存器内，当一个字节的数据输出完成后，SSEL 由低电平变为高电平，从而使 74HC595 的移位寄存器的值被锁存到 74HC595 的锁存器并从其 Q_A~Q_H 引脚输出；在 SPI 输出一个字节数据的同时，74HC595 移位寄存器之前的值也通过 MISO 引脚被 SPI 主机读回。

在把SPI与几种不同的串行I/O芯片相连时，必须使用每片的允许控制端，可用MCU的I/O端口输出线来实现。此时应特别注意这些串行I/O芯片的输入输出特性。

(1) 输入芯片的串行数据输出是否有三态控制端。平时未选中芯片的输出端应处于高阻态。若没有三态控制端，应外加三态门。否则MCU的MISO端只能连接1个输入芯片。

(2) 输出芯片的串行数据输入是否有允许控制端。即应该只有在这片芯片允许时，SCK脉冲才把串行数据移入该芯片；芯片禁止时，SCK对芯片无影响。若没有允许控制端，应在外部用门电路对SCK进行控制后，再加到芯片的时钟输入端，或者SPI只连接1个芯片，不能再连接其它输入或输出芯片。

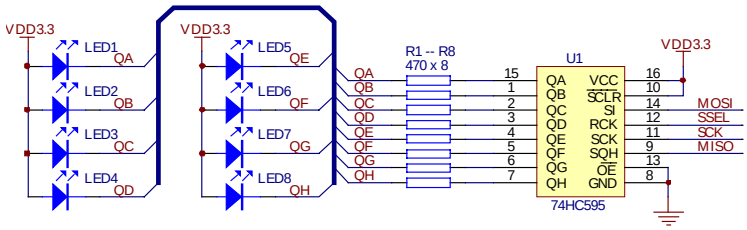


图 6.49 SPI 转并行输出

6.3 总线接口

6.3.1 并行 SRAM

1. 概述

SRAM 为静态 RAM 存储器，具有极高的读写速度，在嵌入式系统中常用来作变量/数据缓冲，或者将程序复制到 SRAM 上运行，以提高系统的性能。注意，SRAM 属于易失性存储器，电源掉电后 SRAM 中的数据将会丢失，所以不可能直接使用 SRAM 引导程序运行。

DRAM 为动态 RAM 存储器，具有存储容量大和价格便宜的特点。DRAM 是用 MOS 电路和电容来作存储元件，由于电容会放电，所以需要定时充电以维持存储内容的正确，例如每隔 2ms 刷新一次数据。

PSRAM（即 Pseudo-SRAM）器件是异步 SRAM 接口技术和利用存储阵列的高密度 DRAM 技术相结合的产物。实际上，这些器件实现了对主机系统透明地自刷新技术。通过扩展包括刷新操作和读出操作两部分时间在内的读出周期的规定周期时间，使得透明的刷新成为可能。这种方法同样也可用于写入周期。

2. SRAM

这里以 IS61LV25616AL 为例，介绍 SRAM 存储器的结构及应用。

IS61LV25616AL 是美国 ISSI 公司的高速 SRAM 器件，采用 CMOS 技术，存储容量为 512K 字节，16 位数据宽度，工作电源 3.3V。

IS61LV25616AL 的功能框图如图 6.50 所示。

IS61LV25616AL 的管脚配置如图 6.51 所示。

IS61LV25616AL 的管脚描述如表 6.6 所示。

芯片使能输入 $\overline{CE}$ 和数据输出使能输入 $\overline{OE}$ 可方便地实现存储器的扩展。低电平有效的写使能（ $\overline{WE}$ ）控制着存储器的写和读操作。高字节（ $\overline{UB}$ ）和低字节（ $\overline{LB}$ ）控制信号控制着对数据字节的访问。IS61LV25616AL 的工作模式如表 6.7 所示。

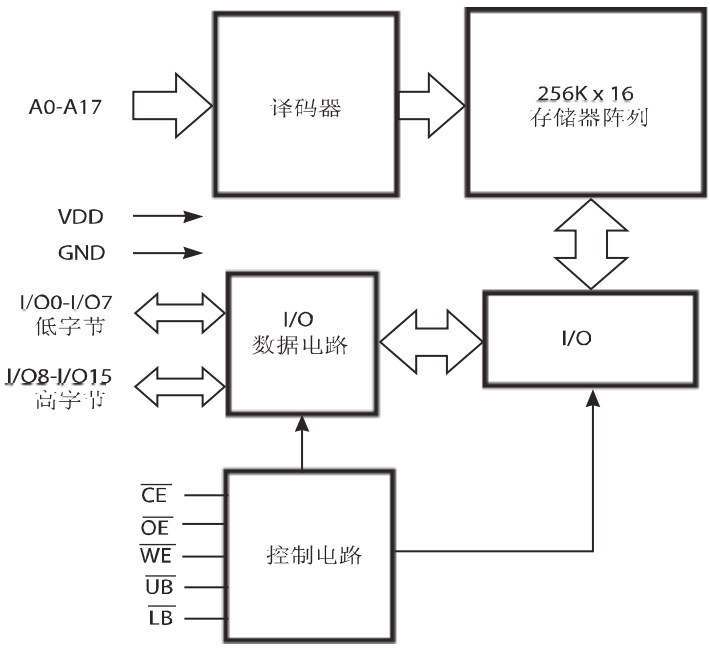


图 6.50 IS61LV25616AL 功能框图

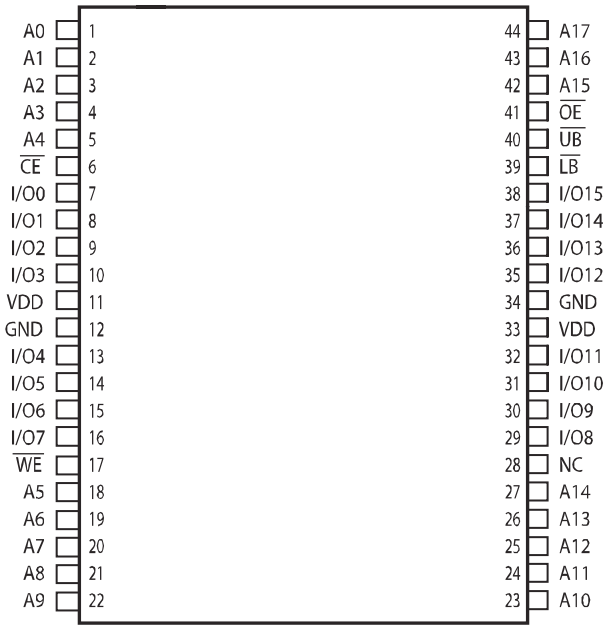


图 6.51 IS61LV25616AL 管脚配置(TSOP)

表 6.6 管脚描述

管脚名称	管脚描述
A0－A17	地址输入
I/O0－I/O15	数据输入/输出
$\overline{CE}$	芯片使能输入
$\overline{OE}$	输出使能输入
$\overline{WE}$	写使能输入
$\overline{LB}$	低字节控制（I/O0－I/O7）

接上表

管脚名称	管脚描述
$\overline{UB}$	高字节控制（I/O8－I/O15）
NC	不连接
V_{DD}	电源
GND	地

表 6.7 IS61LV25616AL 的工作模式

工作模式	$\overline{WE}$	$\overline{CE}$	$\overline{OE}$	$\overline{LB}$	$\overline{UB}$	I/O0 – I/O7	I/O8 – I/O15	V_{DD} 电流
不选择芯片工作	X	H	X	X	X	高阻	高阻	I_{SB1}, I_{SB2}
输出禁能	H	L	H	X	X	高阻	高阻	Icc
	X	L	X	H	H	高阻	高阻	
读	H	L	L	L	H	D_{OUT}	高阻	Icc
	H	L	L	H	L	高阻	D_{OUT}	
	H	L	L	L	L	D_{OUT}	D_{OUT}	
写	L	L	X	L	H	D_{IN}	高阻	Icc
	L	L	X	H	L	高阻	D_{IN}	
	L	L	X	L	L	D_{IN}	D_{IN}	

芯片 IS61LV25616AL 与 LPC2200 的连接如图 6.52 所示。LPC2200 在外部存储器接口 Bank0 上使用 IS61LV25616AL，所以将 LPC2200 的 CS0 与 IS61LV25616AL 的片选引脚连接。存储器连接使用了 16 位总线方式，数据总线使用了 D0～D15，地址总线使用了 A1～A18(16 位总线时，LPC2200 的 A0 引脚没有使用)。为了能够对 IS61LV25616AL 的字单元进行单独的字节操作(如高 8 位，或低 8 位)，所以要把 LPC2200 的 BLS1、BLS0 控制信号分别连接到 IS61LV25616AL 的 $\overline{UB}$、$\overline{LB}$ 引脚。

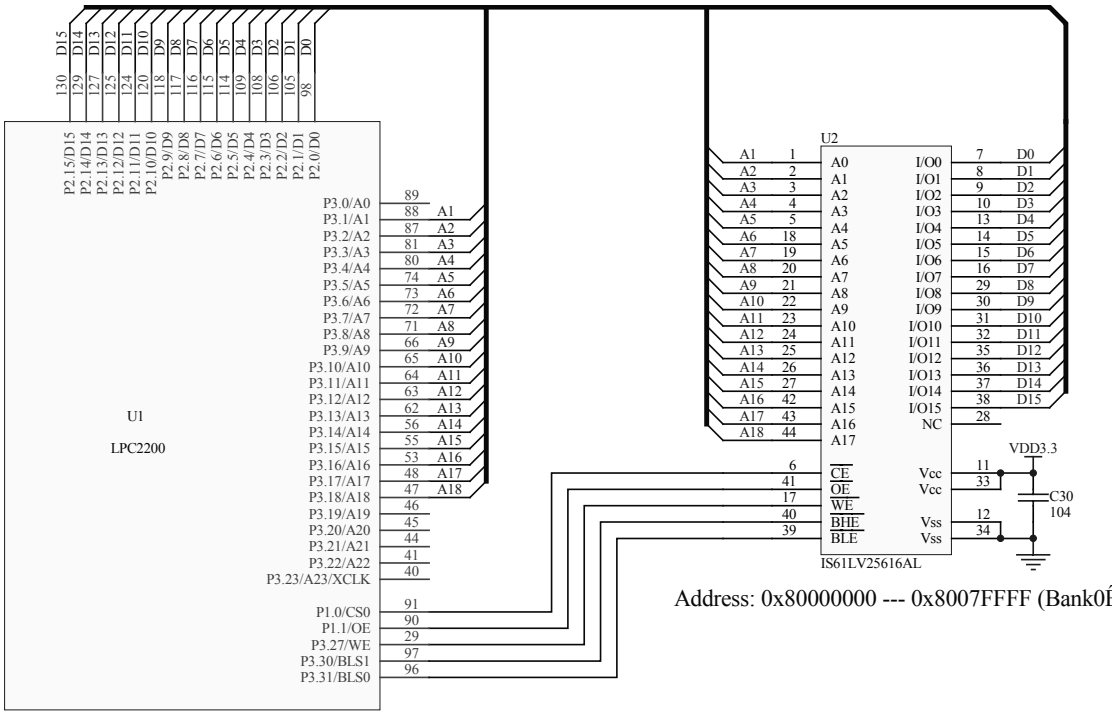


图 6.52 IS61LV25616AL 与 LPC2200 连接电路原理

3. PSRAM

这里以 CellularRAM 的 MT45W4ML16PFA 为例，介绍 PSRAM 存储器的结构及应用。

CellularRAM 是一系列 PSRAM 产品，是一种高速、CMOS 动态随机存取存储器，它们向后可兼容 6T（6 电晶体）结构，适用于低功耗的便携式产品中。CellularRAM 技术有几个特点：它向后可兼容标准异步 SRAM 器件；它是带有 SRAM 接口的 DRAM 技术；它的价格比目前使用的 SRAM 器件更低；器件包含有页面模式读访问，可看作是异步读协议的带宽增加的扩展特性。

为了能在异步存储器总线上实现无缝操作，CellularRAM 产品集成了一种透明的自刷新机制。隐藏刷新不需要系统存储器控制器的额外支持，它对器件的读/写性能没有明显影响。

MT45W4ML16PFA 是 CellularRAM 的一种，是一个 4Meg×16 位的 64Mb 器件。为了减少功耗，内核电压被降低到 1.8V，为了兼容各种不同存储器总线的接口，I/O 电压为 3.0V。

MT45W4ML16PFA 的功能框图如图 6.53 所示。

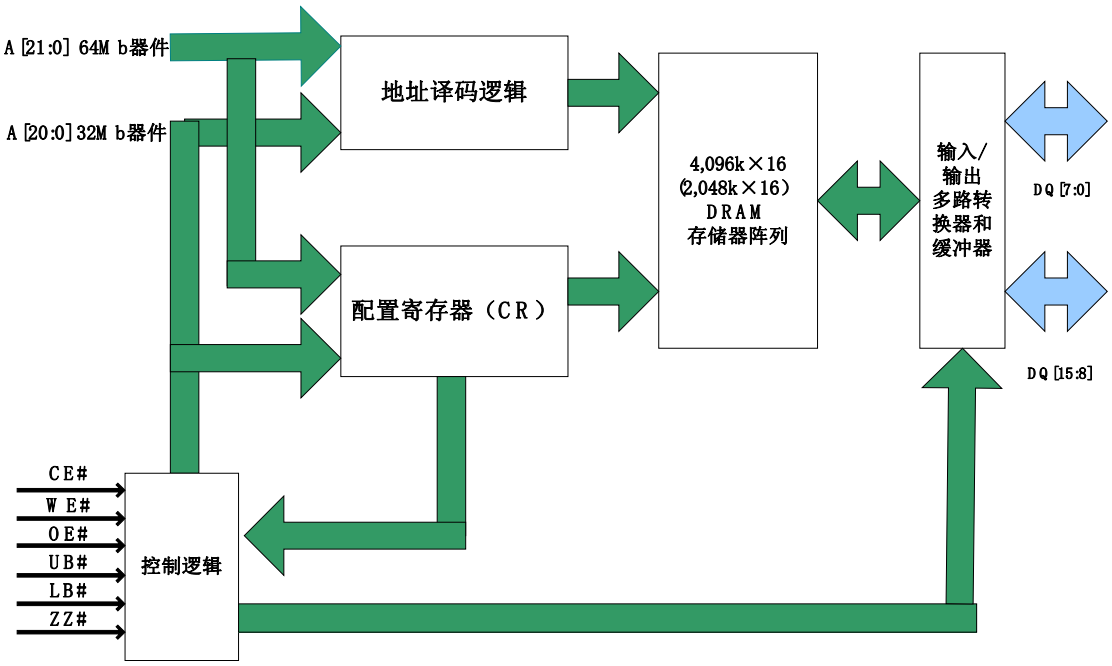


图 6.53 4Meg×16 和 2Meg×16 的功能框图

MT45W4ML16PFA 为 48-Ball VFBGA 封装，管脚配置如图 6.54 所示。

MT45W4ML16PFA 的管脚描述如表 6.8 所示。

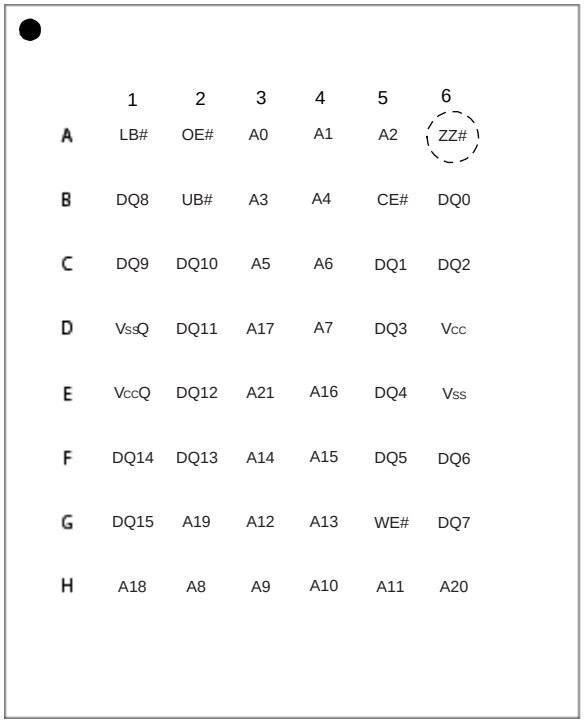


图 6.54 MT45W4ML16PFA 管脚配置(48-Ball VFBGA)

表 6.8 MT45W4ML16PFA 管脚描述

VFBGA 管脚分配	符号	类型	描述
A3,A4,A5,B3,B4,C3 C4,D4,H2,H3,H4,H5 G3,G4,F3,F4,E4,D3, H1,G2,H6,E3	A[21:0]	输入	地址输入：读/写操作的地址输入。这些地址线也用来定义 CR 的装载值。
A6	ZZ#	输入	睡眠使能：当 ZZ#为低电平时，装载 CR 或者器件进入一种低功耗模式（DPD 或 PAR）。
B5	CE#	输入	芯片使能：CE#为低时器件激活；CE#为高时器件禁止并进入等待模式。
A2	OE#	输入	输出使能：OE#为低时使能输出缓冲器；OE#为高时输出缓冲器禁止。
G5	WE#	输入	写使能：WE#为低时使能写操作。
A1	LB#	输入	低字节使能。DQ[7:0]
B2	UB#	输入	高字节使能。DQ[15:8]
B6,C5,C6,D5,E5,F5, F6,G6,B1,C1,C2,D2, E2,F2,F1,G1	DQ[15:0]	输入/输出	数据输入/输出
D6	Vcc	电源	器件电源：（1.7V－1.95V）的器件内核工作电源。
E1	VccQ	电源	I/O 电源：（1,8V, 2.5V, 3.0V）的输入/输出缓冲器电源。
E6	Vss	电源	Vss 必须连接到地。
D1	VssQ	电源	VssQ 必须连接到地。

MT45W4ML16PFA 的操作模式如表 6.9 所示。

表 6.9 总线操作模式

模式	功能	CE#	WE#	OE#	LB#UB#	ZZ#	DQ[15:0][1]	注
等待	等待	H	X	X	X	H	高阻	2,5
读	激活	L	H	L	L	H	数据输出	1,4
写	激活	L	L	X	L	H	数据输入	1,3,4
不工作	空闲	L	X	X	X	H	X	4,5
PAR	部分存储器阵列刷新	H	X	X	X	L	高阻	6
DPD	省电	H	X	X	X	L	高阻	6
装载配置寄存器(CR)	激活	L	L	X	X	L	高阻	

注：

1. 当 LB#和 UB#有效时（低电平），DQ[15:0]的值受影响。当只有 LB#有效时，DQ[7:0]的值受影响。当只有 UB#有效时，DQ[15:8]的值受影响。
2. 当器件处于等待模式时，控制输入（WE#, OE#）、地址输入和数据输入/输出不受外部影响。
3. 当 WE#有效时，OE#输入在内部被禁止，不影响 I/O 口的值。
4. 无论地址是否发生改变，器件在该模式下都将消耗电流。
5. VIN=VccQ 或 0V；为了获得最低等待电流，所有器件管脚必须保持静态。
6. 当配置寄存器位 CR[4]为 0 时，DPD 使能；否则，PAR 使能。

用户可访问配置寄存器（CR）定义了 CellularRAM 器件执行片内刷新的过程以及是否允许页面模式读访问。上电时默认设定值装入 CR 寄存器，CR 的值可在工作过程中随时被更新。装载配置寄存器（CR）操作模式见表 6.9。

自刷新过程中的电流消耗需要特别注意。CellularRAM 包含 3 种降低刷新电流的方法（可供系统访问）。温度补偿刷新（TCR）用来根据环境温度对刷新速度进行调整。温度越低，刷新速度越低，运用该方法可以降低等待模式的电流消耗。通过将睡眠使能管脚 ZZ#设置成低电平可使能 2 种低功耗模式之一：部分存储器阵列刷新（PAR）或省电模式（DPD）。PAR 是仅限定用于对保存有重要数据的 DRAM 阵列部分单元的刷新操作。DPD 禁能所有刷新操作，通常用在无重要数据存储在器件中的场合。这 3 种刷新机制都可通过 CR 进行访问。

页面模式读操作

页面模式是延迟异步读操作的增强特性扩展。含有页面模式的器件在操作过程中，先执行初始异步读访问，再仅通过改变低位地址来将相邻地址单元的数据快速读出。地址 A[3:0]用于决定 16 位地址 CellularRAM 页面的成员。只要 A[4]或更高位地址发生改变，一次新的 t_{AA} 访问将被启动。图 6.55 所示为一次页面模式访问的时序图。

页面模式的工作原理是邻近地址读出的时间较随机读取的时间更短。写操作不包含页面模式操作。

出于刷新的考虑，CE#低电平时间受到限制。CE#为低电平的时间不能大于 t_{CEM}。

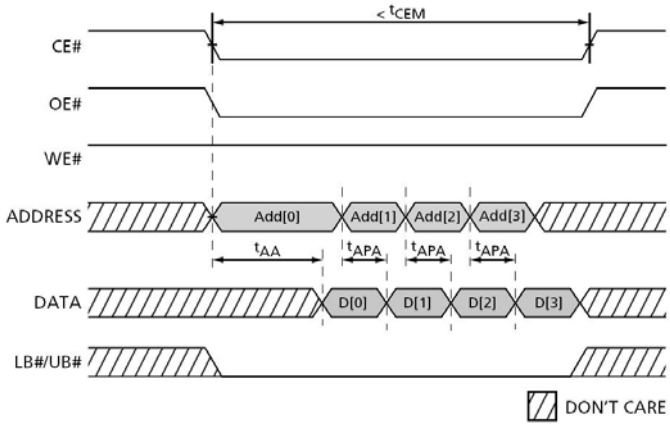


图 6.55 页面读操作

MT45W4ML16PFA 与 LPC2200 应用连接

芯片 MT45W4ML16PFA 与 LPC2200 的连接如图 6.56 所示。LPC2200 在外部存储器接口 Bank0 上使用 MT45W4ML16PFA，所以将 LPC2200 的 CS0 与 MT45W4ML16PFA 的片选引脚连接。存储器连接使用了 16 位总线方式，数据总线使用了 D0～D15，地址总线使用了 A1～A22(16 位总线时，LPC2200 的 A0 引脚没有使用)。为了能够对 MT45W4ML16PFA 的字单元进行单独的字节操作(如高 8 位，或低 8 位)，所以要把 LPC2200 的 BLS1、BLS0 控制信号分别连接到 MT45W4ML16PFA 的 UB#、LB#引脚。

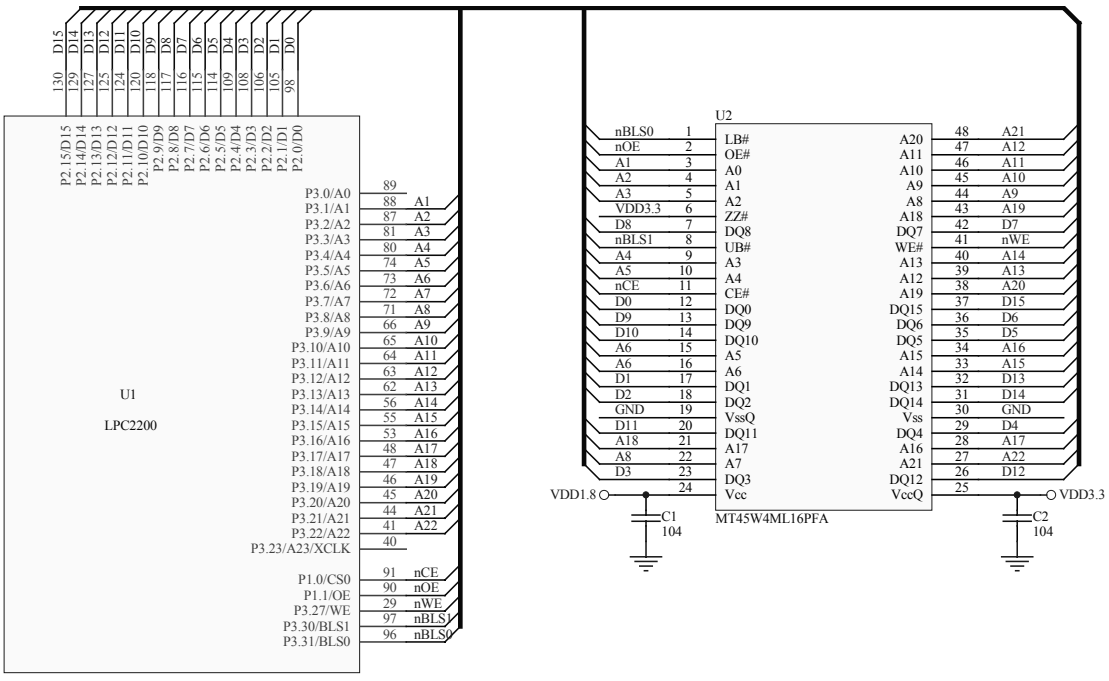


图 6.56 MT45W4ML16PFA 与 LPC2200 连接电路原理

6.3.2 并行 FALSH

1. 概述

FLASH 存储器又称闪存，是一种可在线多次擦除的非易失性存储器，即掉电后数据不会丢失。FLASH 存储器还具有体积小、功耗低、抗振性强等优点，是嵌入式系统的首选存

储设备。

FLASH 存储器主要分为两种，一种为 NOR 型 FLASH，另一种为 NAND 型 FLASH，两种类型 FLASH 特点如表 6.10 所示。

表 6.10 NOR 型和 NAND 型 FLASH 特点

类型	特点	芯片举例
NOR 型	可以直接读取芯片内存储器的数据，速度比较快，但价格较高。 芯片内执行(XIP, eXecute In Place)，应用程序可以直接在 FLASH 上运行，不必再把代码读到系统 RAM 中。	SST39VF160
NAND 型	内部数据以块为单位进行存储，地址线和数据线共用，使用控制信号选择。 极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用 NAND 型的困难在于 FLASH 的管理和需要特殊的系统接口。	K9F2808U0C

2. NOR 型与 NAND 型 FLASH 的区别

NOR 型与 NAND 型 FLASH 存储器的主要区别如下：

(1) 接口差别

NOR 型 FLASH 采用的是 SRAM 接口，提供有足够的地址引脚来寻址，可以很容易地存取其片内的每一个字节；

NAND 型 FLASH 使用复杂的 I/O 口来串行地存取数据，各个产品或厂商的方法可能各不相同。通常是采用 8 个引脚来传送控制、地址和数据信息。

(2) 读写的基本单位

NOR 型 FLASH 操作是以“字”为基本单位；

NAND 型 FLASH 操作是以“页面”为基本单位，页的大小一般为 512 字节。

(3) 性能比较

NOR 型 FLASH 的地址线和数据线是分开的，传输效率很高，程序可以在芯片内执行；

NOR 型的读速度比 NAND 型稍快一些；

NAND 型的写入速度比 NOR 型快很多(由于 NAND 型读写的基本单位为“页面”，所以对于小量数据的写入，总体速度要比 NOR 型慢)；

NAND 型的擦除速度远比 NOR 型快；

NAND 型的擦除单元更小，相应的擦除电路更少。

(4) 容量和成本

NAND 型 FLASH 具有极高的单元密度，容量可以做得比较大，加上其生产过程更为简单，价格也就相应地降低了。

NOR 型 FLASH 占据了容量为 1～16MB 闪存市场的大部分，而 NAND 型 FLASH 只是用在 8～128MB 的产品当中，这也说明 NOR 主要应用在代码存储介质中，NAND 适合于数据存储，NAND 在 CompactFlash、Secure Digital、PC Cards 和 MMC 存储卡市场上所占份额最大。

(5) 软件支持

在 NOR 型 FLASH 上运行代码不需要任何的软件支持，而在 NAND 型 FLASH 上进行同样操作时，通常需要驱动程序，也就是内存技术驱动程序 MTD (Memory Technology

Drivers)。NAND 型和 NOR 型 FLASH 在进行写入和擦除操作时都需要 MTD(说明，MTD 已集成在 FLASH 芯片内部，它是对 FLASH 进行操作的接口)。

3. NOR 型 FLASH 存储器

这里以 SST39VF160 为例，介绍 NOR 型 FLASH 存储器的结构及操作。

SST39VF160 是 SST 公司的 CMOS 多功能 FLASH（MPF）器件，存储容量为 2M 字节，16 位数据宽度(即一个字为 2 字节)，工作电压为 2.7～3.6V。SST39VF160 由 SST 特有的高性能 SuperFlash 技术制造而成，SuperFlash 技术提供了固定的擦除和编程时间，与擦除/编程周期数无关。

SST39VF160 芯片的功能框图如图 6.57 所示。

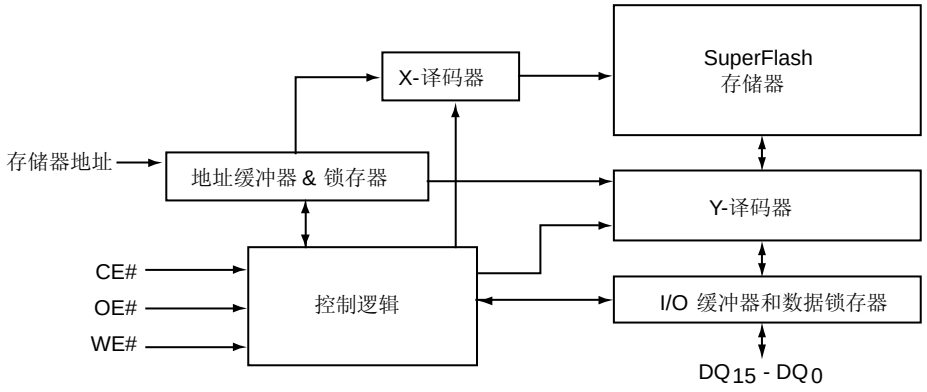


图 6.57 SST39VF160 功能框图

SST39VF160 芯片的管脚配置图如图 6.58 所示。

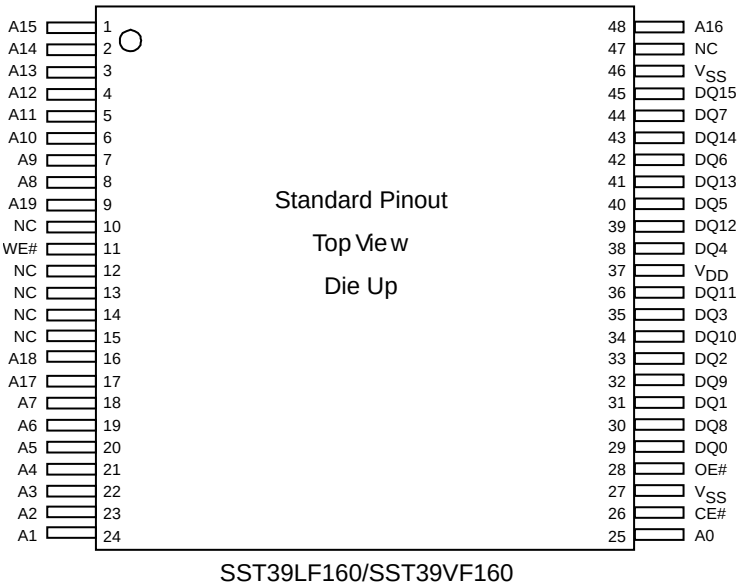


图 6.58 SST39VF160 的管脚配置图

SST39VF160 芯片的管脚描述如表 6.11 所示。

表 6.11 SST39VF160 的管脚描述

符号	管脚名称	功能
A19～A0	地址输入	存储器地址。扇区擦除时，A19～A11 用来选择扇区。块擦除时，A19～A15 用来选择块。
DQ15～DQ0	数据输入/输出	读周期内输出数据，写周期内输入数据。 写周期内数据内部锁存。 OE#或 CE#为高时输出为三态。
CE#	芯片使能	CE#为低时启动器件开始工作。
OE#	输出使能	数据输出缓冲器的门控信号。
WE#	写使能	控制写操作。
V_{DD}	电源	供给电源电压： 2.7～3.6V
Vss	地	
NC	不连接	悬空管脚

从图 6.58 和表 6.11 中可以看出，NOR 型 FLASH 存储器采用的是 SRAM 接口，其地址线和数据线是分开的。

SST39VF160 芯片的工作模式如表 6.12 所示。

表 6.12 工作模式选择

模式	CE#	OE#	WE#	DQ	地址
读	V_{IL}	V_{IL}	V_{IH}	D_{OUT}	A_{IN}
编程	V_{IL}	V_{IH}	V_{IL}	D_{IN}	A_{IN}
擦除	V_{IL}	V_{IH}	V_{IL}	X^1	扇区或块地址， 芯片擦除时为 XXH
等待	V_{IH}	X	X	高阻	X
写禁止	X	V_{IL}	X	高阻/D_{OUT}	X
	X	X	V_{IH}	高阻/D_{OUT}	X
器件标识符软件模式	V_{IL}	V_{IL}	V_{IH}		

注： 1. X 可以是 V_{IL} 或 V_{IH}，但不能为其它值

NOR 型 FLASH 存储器容量越来越大，为了方便数据管理，将 FLASH 划分为块(Block)，每个块又分成扇区(Sector)。SST39VF160 的块大小为 32K 字，扇区大小为 2K 字。

器件操作

SST39VF160 的存储器操作由命令来启动。命令通过标准微处理器写时序写入器件。将 WE#拉低、CE#保持低电平来写入命令。地址总线上的地址在 WE#或 CE#的下降沿（无论哪一个后产生下降沿）被锁存。数据总线上的数据在 WE#或 CE#的上升沿（无论哪一个先产生上升沿）被锁存。存储器操作命令如表 6.13 所示。

(1) 读

SST39VF160 的读操作由 CE#和 OE#控制，只有两者都为低电平时，系统才能从器件的输出管脚获得数据。CE#是器件片选信号。当 CE#为高电平时，器件未被选中工作，只消耗等待电流。OE#是输出控制信号，用来控制输出管脚数据的输出。当 CE#或 OE#为高电平时，数据总线呈现高阻态。存储器操作命令如表 6.13 所示。

(2) 字编程操作

SST39VF160 以字形式进行编程。编程前，包含字的扇区必须完全擦除。编程操作分为

三步。第一步，执行三字节装载时序，用于软件数据保护。第二步，装载字地址和字数据。在字编程操作中，地址在 CE#或 WE#的下降沿（不论哪一个后产生下降沿）锁存。数据在 CE#或 WE#的上升沿（不论哪一个先产生上升沿）锁存。第三步，执行内部编程操作，该操作在第 4 个 WE#或 CE#的上升沿出现（不论哪一个先产生上升沿）之后启动。编程操作一旦启动，将在 20μs 内完成。在编程操作过程中，只有数据#查询位和触发位的读操作有效。在内部编程操作过程中，主机可以自由执行其它任务。该过程中发送的任何命令都被忽略。存储器操作命令如表 6.13 所示。字编程操作流程图如图 6.59 所示。

(3) 扇区/块擦除操作

扇区（或块）擦除操作允许系统对器件执行连续的扇区擦除（或连续的块擦除）。SST39VF160 提供了扇区擦除和块擦除模式。扇区结构统一为 2K 字的规格。块擦除模式是对相同 32K 字的块执行擦除操作。扇区操作通过在最新一个总线周期内执行一个 6 字节的命令时序（扇区擦除命令（30H）和扇区地址（SA））来启动。块擦除操作通过在最新一个总线周期内执行一个 6 字节的命令时序（块擦除命令（50H）和块地址（BA））来启动。扇区或块地址在第 6 个 WE#脉冲的下降沿锁存，命令（30H 或 50H）在第 6 个 WE#脉冲的上升沿锁存。内部擦除操作在第 6 个 WE#脉冲后开始执行。擦除操作是否结束由数据#查询位或触发位决定。扇区或块擦除操作过程中发布的任何命令都被忽略。存储器操作命令如表 6.13 所示。扇区/块擦除操作流程如图 6.61 所示。

(4) 芯片擦除操作

SST39VF160 包含芯片擦除功能，允许用户擦除整个存储器阵列，使其变为“1”状态。这在需要快速擦除整个器件时很有用。

芯片擦除操作通过在最新一个总线周期内执行一个 6 字节的命令（5555H 地址处的芯片擦除命令（10H））时序来启动。在第 6 个 WE#或 CE#的上升沿（无论哪一个先出现上升沿）开始执行擦除操作。擦除过程中，只有触发位或数据#查询位的读操作有效。芯片擦除过程中发布的任何命令都被忽略。存储器操作命令如表 6.13 所示。芯片擦除操作流程如图 6.61 所示。

表 6.13 软件命令时序

命令时序	第 1 个总线写周期		第 2 个总线写周期		第 3 个总线写周期		第 4 个总线写周期		第 5 个总线写周期		第 6 个总线写周期	
	地址[1]	数据[2]	地址[1]	数据[2]	地址[1]	数据[2]	地址[1]	数据[2]	地址[1]	数据[2]	地址[1]	数据[2]
字编程	5555H	AAH	2AAAH	55H	5555H	A0H	WA[3]	数据				
扇区擦除	5555H	AAH	2AAAH	55H	5555H	80H	5555H	AAH	2AAAH	55H	Sax[4]	30H
块擦除	5555H	AAH	2AAAH	55H	5555H	80H	5555H	AAH	2AAAH	55H	Bax[4]	50H
芯片擦除	5555H	AAH	2AAAH	55H	5555H	80H	5555H	AAH	2AAAH	55H	5555H	10H
软件 ID 入口[5,6]	5555H	AAH	2AAAH	55H	5555H	90H						
CFI 查询入口[5]	5555H	AAH	2AAAH	55H	5555H	98H						
软件 ID 退出[7]/CFI 退出	XXH	F0H										
软件 ID 退出[7]/CFI 退出	5555H	AAH	2AAAH	55H	5555H	F0H						

注：1. 对于 SST39LF/VF160 命令时序，地址格式是 A14～A0（Hex），地址 A19～A15 可以是 V_{IL} 或 V_{IH}，不能为其它值。

2. 对于命令时序，DQ15～DQ8 可以是 V_{IL} 或 V_{IH}，但不能为其它值。

3. WA＝编程字地址

4. Sax 用于扇区擦除；使用 A19～A11 地址线

Bax 用于块擦除；使用 A19～A5 地址线

5. 如果器件下电，器件不能保持软件器件 ID 模式。

6. A19～A1＝0；SST 制造商 ID＝00BFH，读出时 A0=0。

SST39LF/VF160 器件 ID＝2782H，读出时 A0=1。

7. 两种软件 ID 退出操作相类似。

写操作状态检测

SST39VF160 提供两种检测写（编程或擦除）周期结束的软件方法，以便优化系统的写周期。软件检测包括 2 个状态位：数据#查询位（DQ7）和触发位（DQ6）。写结束检测模式在 WE#的上升沿后使能，WE#的上升沿用来启动内部的编程或擦除操作。状态检测流程如图 6.60 所示。

非易失性写操作的结束与系统不同步；因此，数据#查询位或触发位的读取可能与写周期结束同时发生。如果这样，系统就可能得到一个错误的结果，即有效数据与 DQ7 或 DQ6 发生冲突。为了防止错误的情况，当一个错误结果出现时，软件程序应当包含一个两次读被访问地址单元的循环。如果两次读取的值均有效，则器件已经完成了写周期，否则拒绝接受数据。

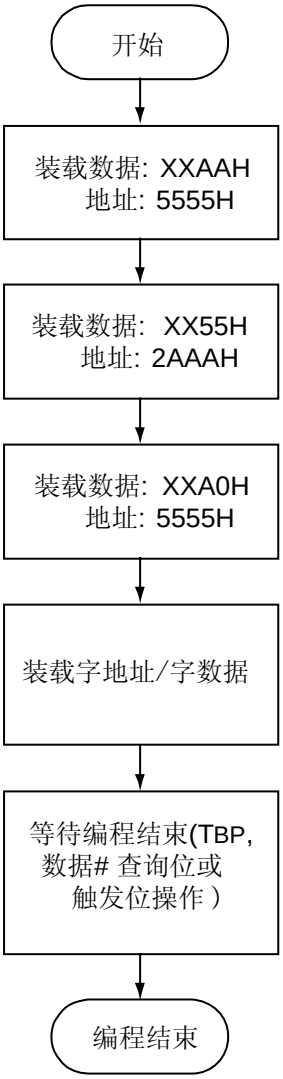
(1) 数据#查询（DQ7）

当 SST39LF/VF160 正在执行内部编程操作时，任何读 DQ7 的动作将得到真实数据的补码。一旦编程操作结束，DQ7 为真实的数据。注意：即使在内部写操作结束后紧接着出现在 DQ7 上的数据可能有效，其余的数据输出管脚上的数据也无效：只有在 1μs 的时间间隔后执行了连续读周期，所得的整个数据总线上的数据才有效。在内部擦除操作过程中，读出的 DQ7 值为‘0’。一旦内部擦除操作完成，DQ7 的值为‘1’。

编程操作的第 4 个 WE#（或 CE#）脉冲的上升沿出现后，数据#查询位有效。对于扇区/块擦除或芯片擦除，数据#查询位在第 6 个 WE#（或 CE#）脉冲的上升沿出现后有效。见图 6.60 的流程图。

(2) 触发位（DQ6）

在内部编程或擦除操作过程中，读取 DQ6 将得到 1 或 0，即所得的 DQ6 在 1 和 0 之间变化。当内部编程或擦除操作结束后，DQ6 位的值不再变化。触发位在编程操作的第 4 个 WE#（或 CE#）脉冲的上升沿后有效。对于扇区/块擦除或芯片擦除，触发位在第 6 个 WE#（或 CE#）脉冲的上升沿出现后有效。



注: X 可以是 V_{IL}或 V_{IH}, 但不能为其它值。

图 6.59 字编程操作流程

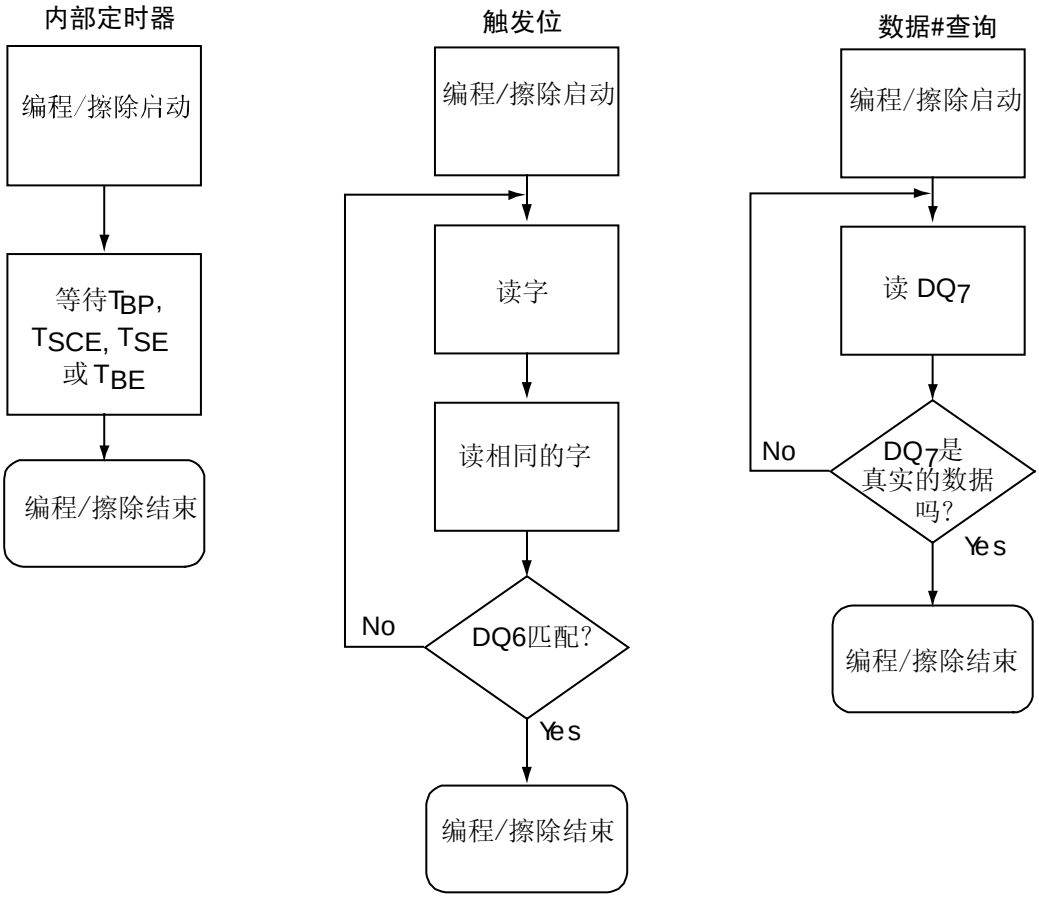
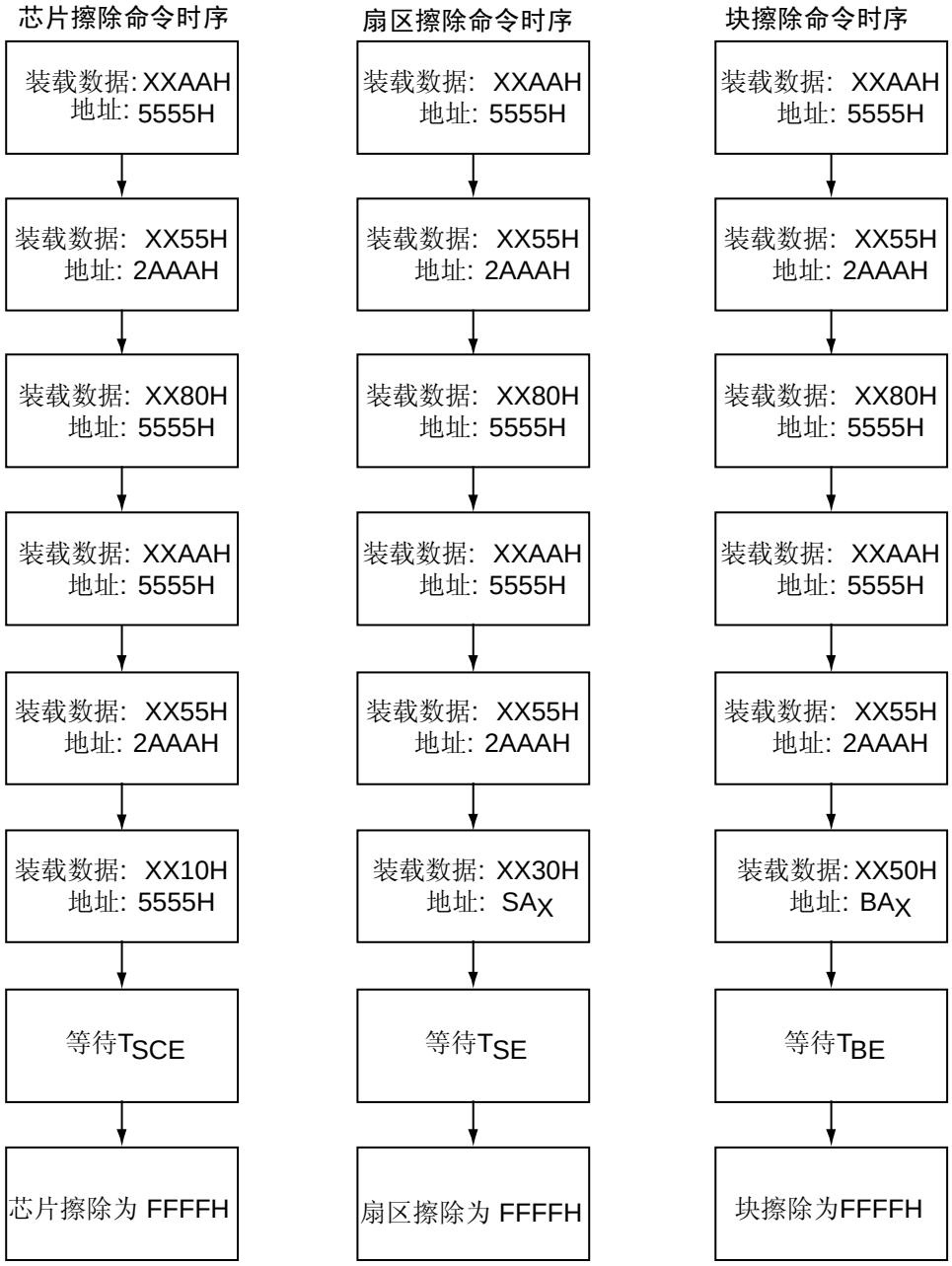

内部定时器
编程/擦除启动
等待TBP, TSCE, TSE 或TBE
编程/擦除结束
触发位
编程/擦除启动
读字
读相同的字
No
DQ6匹配?
Yes
编程/擦除结束
数据#查询
编程/擦除启动
读 DQ7
No
DQ7是真实的数据吗?
Yes
编程/擦除结束

图 6.60 三种等待方法



注：X可以是 V_{IL} 或 V_{IH}，但不能为其它值。

图 6.61 擦除操作流程

SST39VF160 与 LPC2200 应用连接

芯片 SST39VF160 与 LPC2200 的连接如图 6.62 所示。LPC2200 可以使用外部存储器接口 Bank0 上的存储器引导程序运行，所以将 LPC2200 的 CS0 与 SST39VF160 的片选引脚连接(若不需要使用外部存储器引导程序运行，则可以将 SST39VF160 设置为 Bank1、Bank2 或 Bank3)。存储器连接使用了 16 位总线方式，数据总线使用了 D0～D15，地址总线使用了 A1～A20(16 位总线时，LPC2200 的 A0 引脚没有使用)。

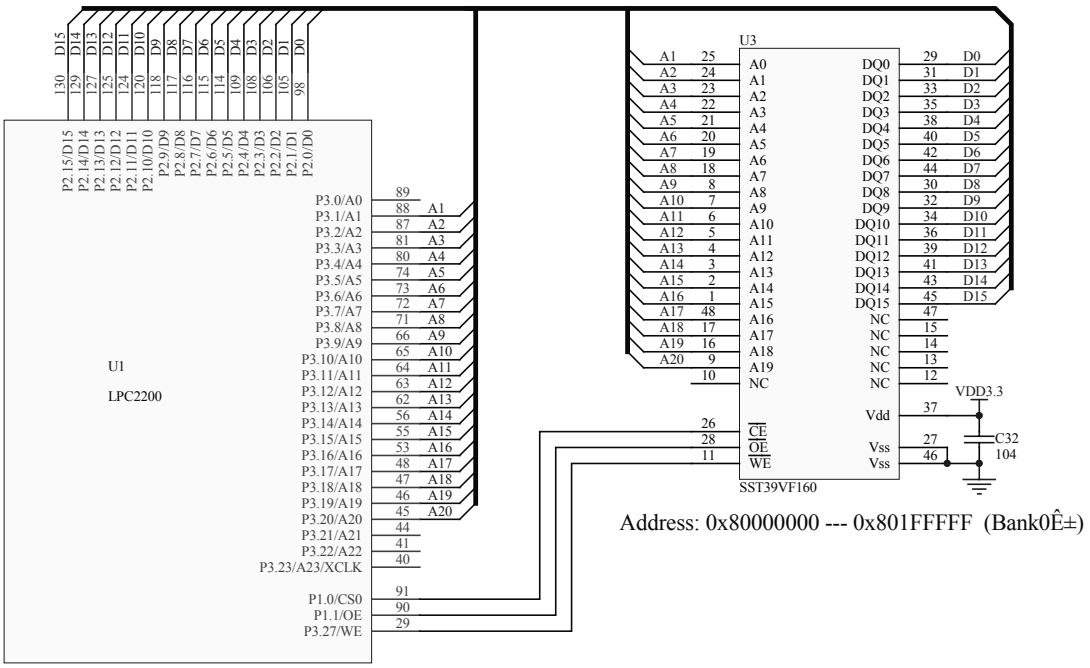


图 6.62 SST39VF160 与 LPC2200 连接电路原理

4. NAND 型 FLASH 存储器

这里以 K9F2808U0C 为例，介绍 NAND 型 FLASH 存储器的结构及操作。

K9F2808U0C 是 SAMSUNG 公司生产的 NAND 型 FLASH 存储器，存储容量为 16M×8Bit，工作电压为 2.7～3.6V。528 字节的页编程操作时间为 200μs，16K 字节的块擦除操作时间为 2ms。页面的数据以每个字 50ns 的速度被读出。片内写控制自动实现所有编程和擦除功能，包括脉冲的周期、内部校验和数据冗余。

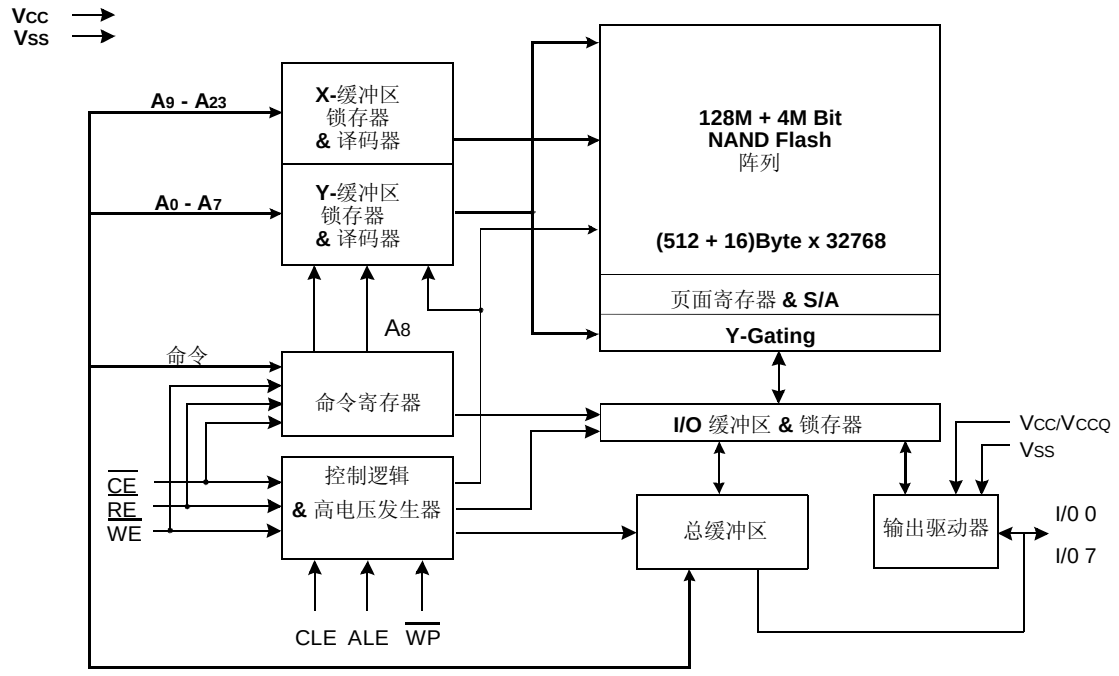


图 6.63 K9F2808U0C 功能框图

K9F2808U0C 芯片的功能框图如图 6.63 所示。数据 I/O、地址输入和操作指令输入均是通过共用 8 位 I/O 总线完成，所以 NAND 型 FALSH 存储器的操作比较复杂。

K9F2808U0C 芯片的管脚配置如图 6.64 所示。

图 6.64 K9F2808U0C 管脚配置(WSOP1)

K9F2808U0C 芯片的管脚描述如表 6.14 所示。

表 6.14 K9F2808U0C 引脚功能

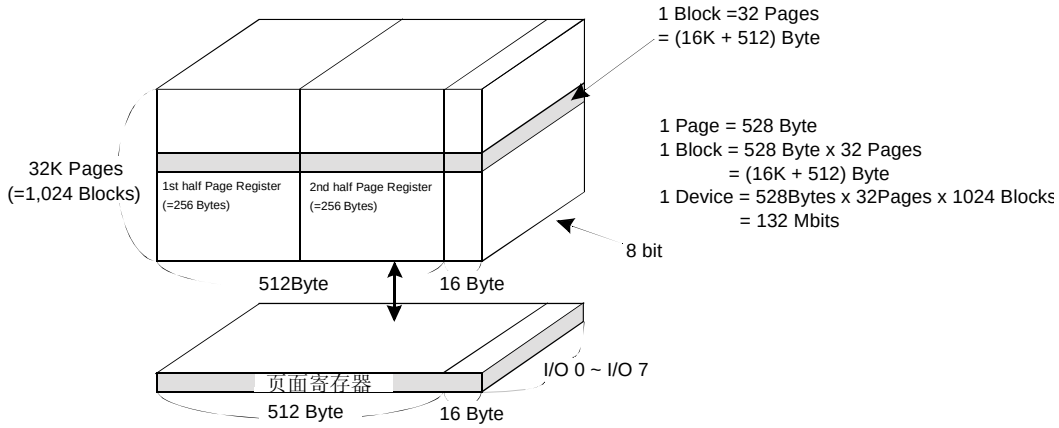
管脚名称	管脚功能
I/O0～I/O7	**数据输入/输出** 这些 I/O 口用来输入命令、地址和数据，以及在读操作时输出数据。当芯片未被选择使用或输出禁止时，I/O 口为高阻态。
CLE	**命令锁存使能** CLE 输入控制着发送到命令寄存器的命令的有效通路。当它为有效的高电平时，命令在 $\overline{WE}$ 信号的上升沿通过 I/O 口锁存到命令寄存器中。
ALE	**地址锁存使能** ALE 输入控制着地址到内部地址寄存器的有效路径。ALE 为高电平时，地址在 $\overline{WE}$ 信号的上升沿被锁存。
$\overline{CE}$	**芯片使能** $\overline{CE}$ 输入是器件选择控制信号。当器件处于忙状态时，$\overline{CE}$ 的高电平被忽略，器件在执行编程或擦除操作时不会返回到等待模式。
$\overline{RE}$	**读使能** $\overline{RE}$ 输入控制着串行数据的输出。当该信号有效时，数据被驱动到 I/O 总线上。$\overline{RE}$ 变成上升沿 tREA 后，数据有效。$\overline{RE}$ 每出现一次上升沿，内部列地址计数器就加 1。
$\overline{WE}$	**写使能** $\overline{WE}$ 输入控制着写 I/O 口操作。命令、地址和数据在 $\overline{WE}$ 脉冲的上升沿被锁存。
$\overline{WP}$	**写保护** $\overline{WP}$ 为电源变化时的无意写提供了写/擦除保护。当 $\overline{WP}$ 管脚为有效低电平时，内部高电压发生器复位。

接上表

管脚名称	管脚功能
R/$\overline{B}$	**读/忙输出** R/$\overline{B}$ 输出用来指示器件的工作状态。为低时，表明器件正在执行编程、擦除或随机读操作，这些操作完成后 R/$\overline{B}$ 返回高电平。该信号是一个开漏输出，当芯片未被选择使用或输出被禁能时它不呈现高阻态。
Vcc	**电源** Vcc 是器件电源。
Vss	**地**
NC	**不连接** 管脚内部不连接。
DNU	未使用，该管脚不连接。

K9F2808U0C 芯片的存储阵列组织如图 6.65 所示。K9F2808U0C 的存储空间分为 32K 页，每一页有（512+16）字节。一个 528 字节的数据寄存器连接到存储器单元阵列，用来实现页面读和页面编程操作中 I/O 缓冲和存储器之间的数据传输。该寄存器被分为两个区：数据区和空闲区。数据区又可分为上、下两个区，每个区为 256 字节；空闲区可以用于存放 ECC 校验和其它今年信息。

K9F2808U0C 芯片的存储器阵列由 16 个单元组成，这 16 个单元串联到一起形成一个 NAND 结构。每个单元位于不同的页面。一个块由两个 NAND 结构的串组成。一个 NAND 结构包含 16 个单元。全部 135168 个 NAND 单元位于一个块中。



	I/O 0	I/O 1	I/O 2	I/O 3	I/O 4	I/O 5	I/O 6	I/O 7	
1st Cycle	A0	A1	A2	A3	A4	A5	A6	A7	列地址
2nd Cycle	A9	A10	A11	A12	A13	A14	A15	A16	行地址
3rd Cycle	A17	A18	A19	A20	A21	A22	A23	L*	(页面地址)

图 6.65 K9F2808U0C 芯片存储阵列

器件操作

对 K9F2808U0C 的操作是通过将特定的指令数据写到芯片指令寄存器中实现，指令与时序的定义如表 6.15 所示。

表 6.15 K9F2808U0C 指令与时序

功能	第一个周期	第二个周期	忙时可接受的命令
读数据寄存器(数据区)	00h/ 01h	-	
读数据寄存器(空闲区)	50h	-	
读器件 ID	90h	-	
复位	FFh	-	O
页编程	80h	10h	
块擦除	60h	D0h	
读状态	70h	-	O

读写操作流程

K9F2808U0C 的页编程操作流程如图 6.66 所示。首先向 I/O 写入页编程指令 80H，然后使用 3 个时钟周期写入目的地址(A0～A23)，接着向 I/O 写入数据。数据发送完成后，写入指令 10H 启动页编程，此时芯片内部的逻辑电路将进行页擦除和数据编程操作。微控制器可以读状态指令 70H 来读取状态寄存器的值，若 D6 位为 1 则表明写操作完成。

写操作完成后，通过读取状态寄存器的 D0 位判断编程是否成功，若 d0 位为 0 表示编程成功，否则表示编程失败。另外，如果没有使用 ECC，则还需要对写操作进行校验，如图 6.66 中的虚线框部分。

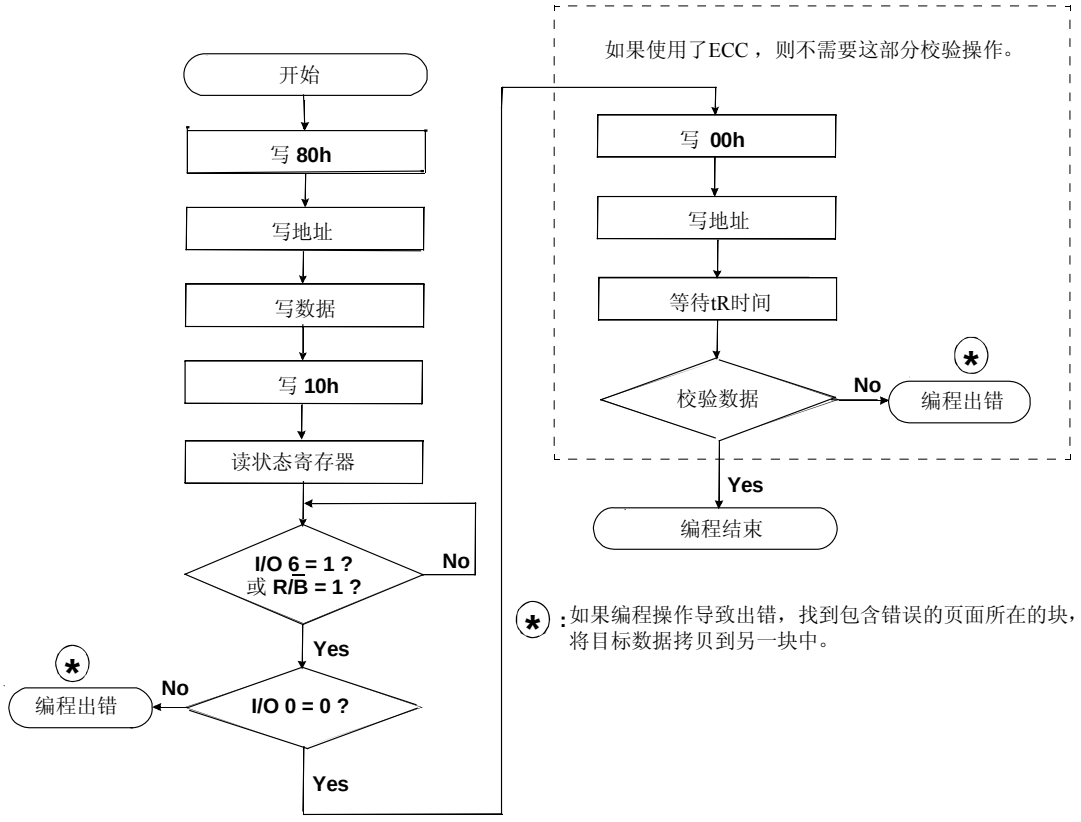


图 6.66 K9F2808U0C 页编程操作流程

K9F2808U0C 的块擦除和读数据操作流程如图 6.67 所示。对于 K9F2808U0C 的擦除是以块为单位，擦除时首先写入块擦除命令 60H，然后输入要擦除块的地址，再写入指令 10H 启动块擦除。微控制可以读状态指令 70H 来读取状态寄存器的值，若 d6 位为 1 则表明擦除完成。写操作完成后，通过读取状态寄存器的 d0 位判断擦除是否成功。

对于 K9F2808U0C 的读数据操作是以页为单位，擦除时首先写入读数据命令 00H，然后输入要读取页的地址，接着从数据寄存器中读取数据，最后进行 ECC 校验。

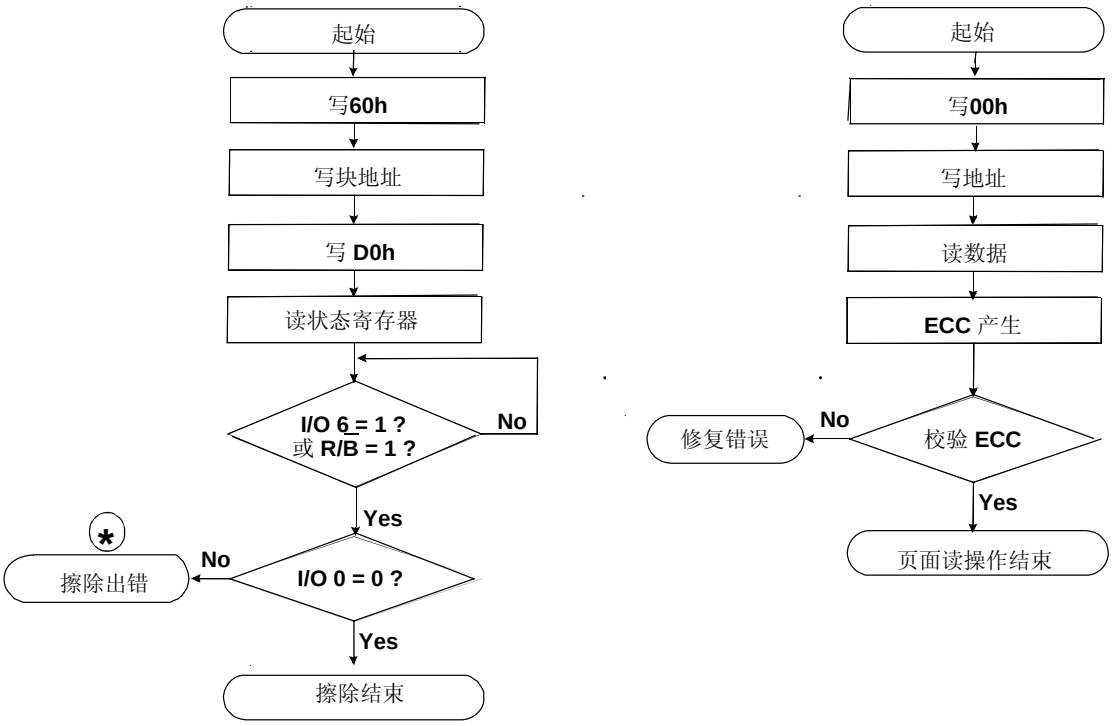


图 6.67 块擦除和读数据操作流程

K9F2808U0C 与 LPC2200 应用连接

K9F2808U0C 与 LPC2200 的应用连接如图 6.68 所示。其中，使用 8 位数据总线（D0～D7）与 K9F2808U0C 的 I/O0～I/O7 引脚相连，通过数据总线发送地址、命令和数据。K9F2808U0C 的片选信号由 CS3 控制，即使用 LPC2200 外部存储器接口的 Bank3 地址空间，而 CLE、ALE 信号分别由 A0、A1 控制，所以 K9F2808U0C 的操作地址如下：

命令输入：0x83000001 (CLE=1，ALE=0)

地址输入：0x83000002 (CLE=0，ALE=1)

数据操作：0x83000000 (CLE=0，ALE=0)

写保护脚 $\overline{WP}$ 接为高电平，禁止 FLASH 的写保护功能。R/B 引脚连接到 LPC2200 的 P1.22，用于查询 K9F2808U0C 的操作状态。

K9F2808U0C 与 LPC2100 应用连接

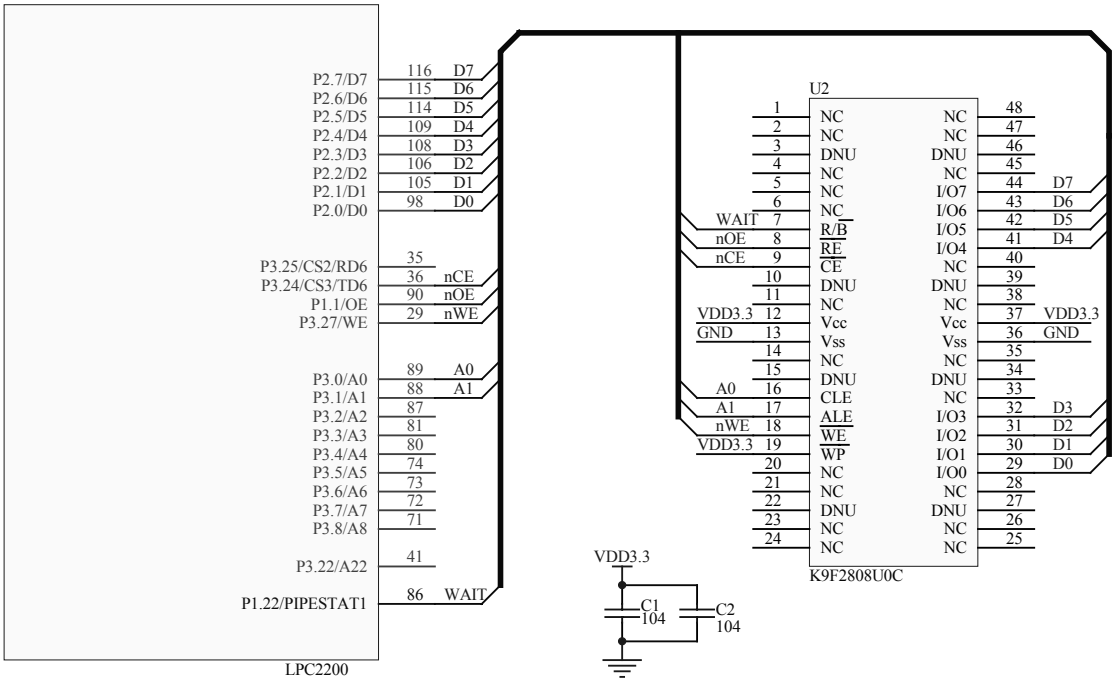


图 6.68 K9F2808U0C 与 LPC2200 连接电路原理(总线方式)

K9F2808U0C 与 LPC2100 的应用连接如图 6.68 所示。K9F2808U0C 的读写操作全部由 I/O 口产生相应时序来完成。

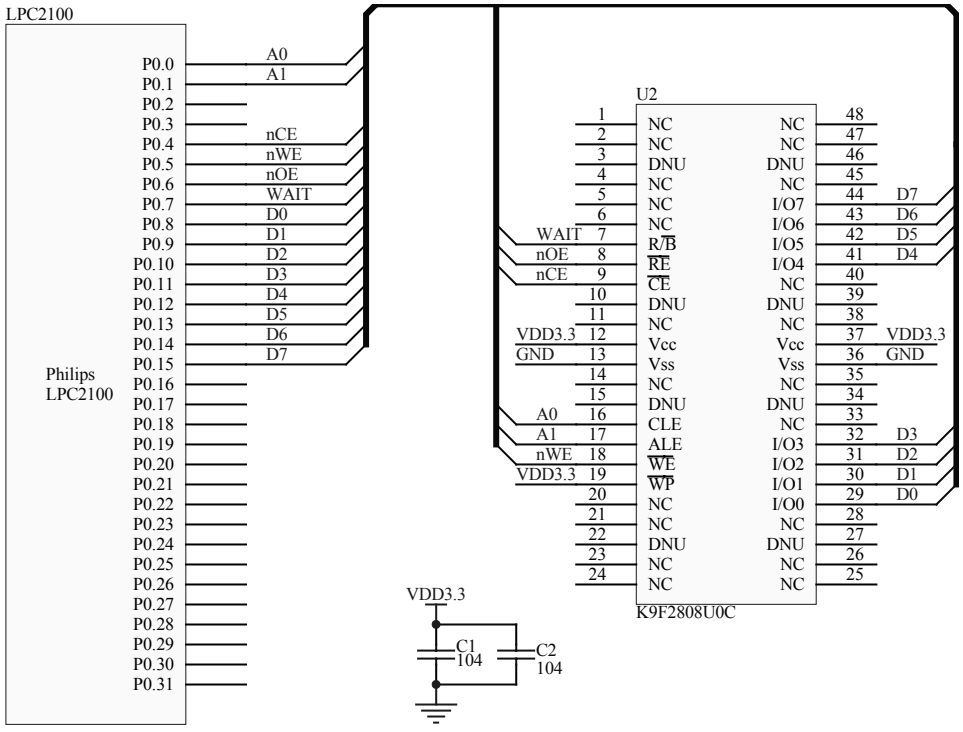


图 6.69 K9F2808U0C 与 LPC2100 连接电路原理(I/O 方式)

6.3.3 USB（D12）接口

USB 总线主要用于 USB 设备与 USB 主机之间的数据通信，特别为 USB 设备与 USB 主机之间大量数据的传输提供了高速、可靠的传输协议。例如：在嵌入式系统中，可以利用

USB 设备与微控制器构成 USB 设备。USB 设备与 PC 机 USB 主控器相连就可以实现嵌入式系统与 PC 机之间的通信了，也就可以实现诸如 U 盘、移动硬盘、USB 接口打印机等功能。

PDIUSBD12 是一款性价比很高的 USB 器件，完全符合 USB1.1 版规范。PDIUSBD12 管脚排列及内部功能框图如图 6.70 所示，表 6.16 为 PDIUSBD12 的管脚说明。

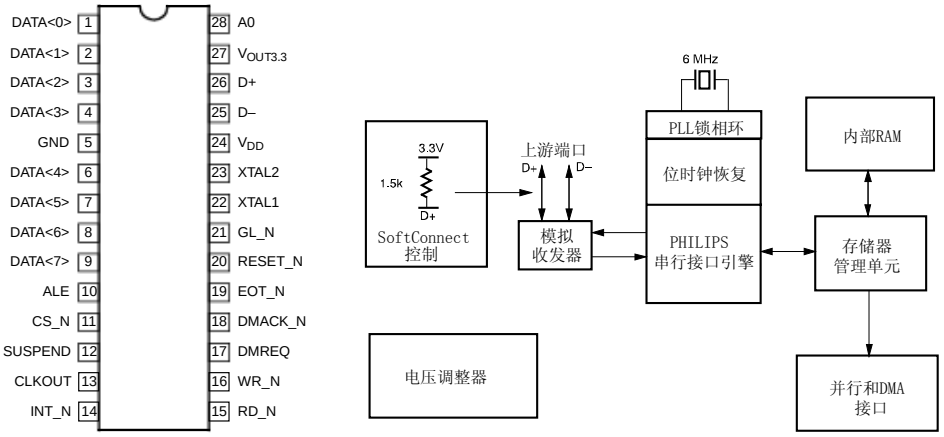


图 6.70 PDIUSBD12 管脚排列及内部功能框图

表 6.16 PDIUSBD12 管脚描述

管脚	符 号	类型	描 述
1	DATA<0>	IO2	双向数据位 0
2	DATA<1>	IO2	双向数据位 1
3	DATA<2>	IO2	双向数据位 2
4	DATA<3>	IO2	双向数据位 3
5	GND	P	地
6	DATA<4>	IO2	双向数据位 4
7	DATA<5>	IO2	双向数据位 5
8	DATA<6>	IO2	双向数据位 6
9	DATA<7>	IO2	双向数据位 7
10	ALE	I	地址锁存使能。在多路地址/数据总线中，下降沿关闭地址信息锁存。将其固定为低电平用于单地址/数据总线配置。
11	CS_N	I	片选（低有效）
12	SUSPEND	I,OD4	器件挂起状态，高电平表示器件处于挂起状态
13	CLKOUT	O2	可编程时钟输出（可控分频）
14	INT_N	OD4	中断（低有效）
15	RD_N	I	读选通（低有效）
16	WR_N	I	写选通（低有效）
17	DMREQ	O4	DMA 请求
18	DMACK_N	I	DMA 应答（低有效）
19	EOT_N	I	DMA 传输结束（低有效）。EOT_N 仅当 DMACK_N 和 RD_N 或 WR_N 一起激活时才有效。
20	RESET_N	I	复位（低有效且不同步）。片内上电复位电路，该管脚可固定接 V_{CC}。
21	GL_N	OD8	GoodLink LED 指示器（低有效）
22	XTAL1	I	晶振连接端 1（6MHz）

接上表

管脚	符 号	类型	描 述
23	XTAL2	O	晶振连接端 2（6MHz）。如果采用外部时钟信号取代晶振，可连接 XTAL1，XTAL2 应当悬空。
24	V_{CC}	P	电源电压（4.0V~5.5V），要使器件工作在 3.3V，对 V_{CC} 和 $V_{OUT3.3}$ 脚都提供 3.3V。
25	D-	A	USB D-数据线
26	D+	A	USB D+数据线
27	$V_{OUT3.3}$	P	3.3V 调整输出。要使器件工作在 3.3V，对 V_{CC} 和 $V_{OUT3.3}$ 脚都提供 3.3V。
28	A0	I	地址位。A0=1 选择命令指令，A0=0 选择数据。该位在多路地址/数据总线配置时可忽略，应将其接固定电平。

注： O2 ： 2mA 驱动输出 OD4 ： 4mA 驱动开漏输出

OD8 ： 8mA 驱动开漏输出 IO2 ： 4mA 输出

在没有使用到 DMA 方式的时候，DMACK_N 和 EOT_N 要求接上拉电阻。

如表 6.16 所示，PIDUSBD12 具有 8bit 的数据总线接口 DATA0 ~ DATA7，片选引脚 CS_N 以及读选通引脚 RD_N 和写选通引脚 WR_N。由此可见，PIDUSBD12 的硬件接口和外部存储器接口很相似，因此，可以当作一片外部 RAM 芯片来进行访问。

ALE（地址锁存）引脚应用于控制 PDIUSBD12 的微处理器的地址/数据为总线复用时的情况，如 80C51 微控制器的外部总线的低 8 位就是数据总线与地址总线复用的情况。

但对于 LPC2200 微控制器，它的外部数据总线与地址总线是分开的。这时，PDIUSBD12 的 DATA0~DATA7 可以直接与 LPC2200 的数据总线的 D0~D7 直接相连就可以构成访问 PDIUSBD12 的数据总线了。

LPC2200 微控制器访问 PDIUSBD12 的地址总线则必须由片选信号和地址信号构成。举个例子，PDIUSBD12 的片选引脚 CS_N 与 LPC2200 的 Bank2 片选引脚相连，而 PDIUSBD12 的 A0 则需要与 LPC2200 的地址线 A0 相连了。如图 6.71 所示为 PDIUSBD12 与 LPC2200 构成的 USB 设备电路原理图。

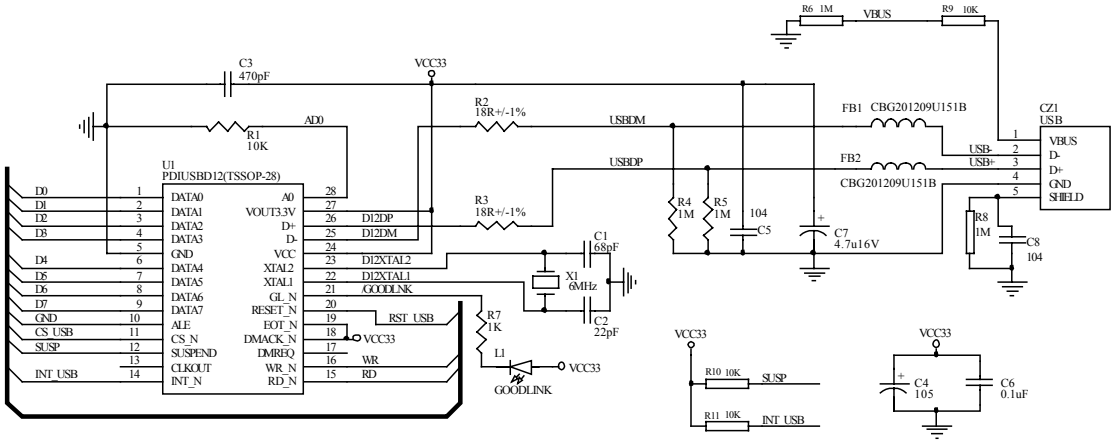


图 6.71 PDIUSBD12 与 LPC2200 电路原理图

图 6.71 中 PDIUSBD12 各引脚（图 6.71 中的网络标号）与 LPC2200 的具体接口如表 6.17 所示。

表 6.17　PDIUSBD12 与 LPC2200 连接关系

PDIUSBD12	功　　能	LPC2200
D0～D7	PDIUSBD12 数据总线	D0～D7（数据总线低 8 位）
AD0	PDIUSBD12 地址总线	A0（地址总线第 0 位）
CS_USB	PDIUSBD12 片选线	nCS2（Bank2 片选）
RD	PDIUSBD12 读使能(低电平有效)	nOE（读使能）
WR	PDIUSBD12 写使能(低电平有效)	nEW（写使能）
INT_USB	PDIUSBD12 中断输出信号	P0.16_EINT0（中断 0）
RST_USB	PDIUSBD12 复位输入信号	P0.10_RTS1
SUSP	PDIUSBD12 挂起输入信号	P0.13_DTR1

这样，PDIUSBD12 就相当于 LPC2200 的一片外部 RAM 了，具有访问它的数据总线、地址总线、读写信号。还有不同于一般外部存储器的中断输出信号以及复位输入和挂起输出信号线。

由于 PDIUSBD12 使用 LPC2200 外部存储控制的 Bank2 部分，Bank2 的存储空间范围为：0x82000000～0x82FFFFFF。由于 PDIUSBD12 只有地址线 A0，因此，只占用了两个字节的地址空间，其地址如下：

数据地址：　0x82000000（偶数地址）

命令地址：　0x82000001（奇数地址）

图 6.71 的电路原理图中，必须注意以下问题：

（1）PDIUSBD12 振荡电路的电容 C1 与 C2 必须分别为 68pF 和 22pF，否则可能会造成 PDIUSBD12 工作时钟不正常。

（2）注意串联在 D+和 D-上的磁珠的型号，如果型号选择不当会影响 USB 总线的稳定性。磁珠元件内部由电阻与电感组成，所以选型时必须综合考虑 USB 总线频率与磁珠的阻抗是否匹配。图 6.71 电路中给出了磁珠元件的一个典型的型号。

（3）D+/ D－ 线上应串接 18 欧的匹配电阻。

至于 LPC2100 系列芯片没有外部总线控制器，如果需要与 PDIUSBD12 连接，可以用 GPIO 模拟总线（参考 6.2.1 小节的相关部分），具体电路可以参考图 6.72。

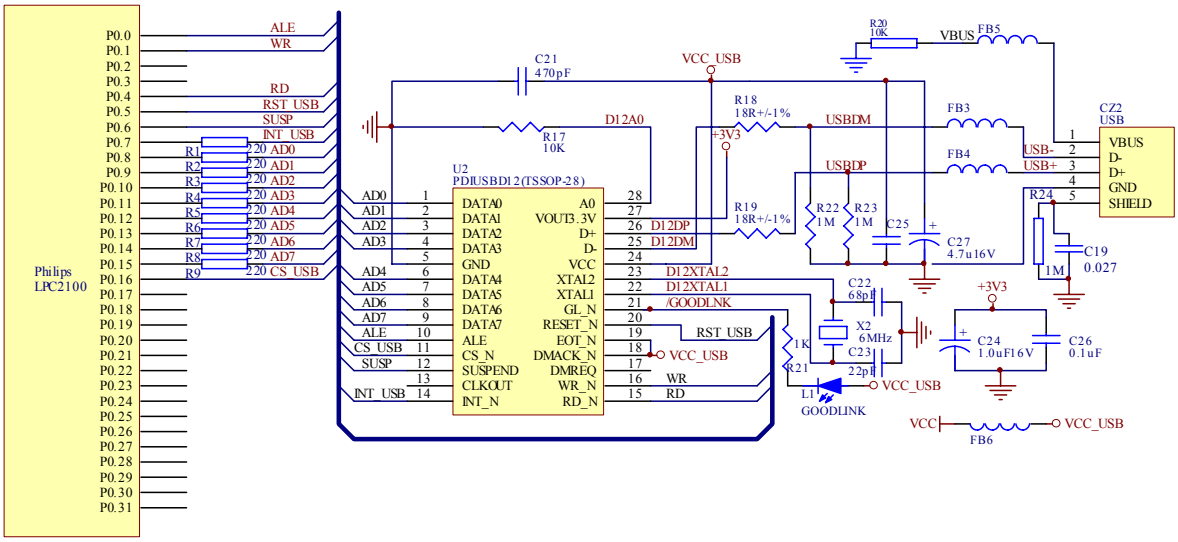


图 6.72　LPC2100 与 PDIUSBD12 连接电路图

6.3.4 液晶接口

1. 液晶显示屏简介

液晶是一种在一定温度范围内呈现既不同于固态、液态，又不同于气态的特殊物质态，它既具有各向异性的晶体所特有的双折射性，又具有液体的流动性。液晶显示器件（英文的简写为 LCD）就是利用液晶态物质的液晶分子排列状态在电场中改变而调制外界光的被动型显示器件。液晶显示屏是平板显示器件中的一种，具有低工作电压、微功耗、无辐射、体积小等特点，被广泛应用于各种各样嵌入式产品中，如手机、PDA、数码相机、电子游戏机和便携式仪表等等。

点阵式图形液晶显示屏是 LCD 的一种，能够动态显示图形、汉字以及各种符号信息，为各种电子产品提供了友好的人机界面。随着 STN 和 TFT 液晶显示屏技术的成熟发展及制造成本的不断降低，点阵式图形液晶显示屏也就成为了嵌入式系统中最主要的图形显示设备。

2. 液晶显示屏分类

按显示原理分：TN（Twist Nematic）扭曲向列型、STN（Super Twist Nematic）超扭曲向列型、TFT（Thin Film Transistor）薄膜晶体管型等。

TN 型液晶屏，将涂有 ITO 透明导电层的两片玻璃基板间夹上一层正介电各向异性液晶，液晶分子沿玻璃表面平行排列，排列方向在上下玻璃之间连续扭转 90°。然后上下各加一偏光片，底面加上反光片，基本就构成了 TN 型液晶显示屏。

由于 TN 型液晶显示器件的电光响应慢，电光曲线陡度不够陡峭，所以在进行动态、多路驱动上有一定限制。使用 TN 型液晶显示技术，最多只能做到 64 路点阵的驱动，这种类型的图形液晶显示屏只能做单色小规模点像素的液晶屏。

STN 型跟 TN 型结构大体相同，只不过液晶分子扭曲 180°，还可以扭曲 210°或 270°等，其特点是电光响应曲线更好，可以适应更多的行列驱动。彩色和单色模式的 STN 技术可以做到 2000 路点阵以上的驱动。

STN 液晶只可以实现伪彩色（一般人眼可以分辨 18bit 色，即 256K 色，所以达到 18bit 色和超过 18bit 色的被称之为真彩色，否则称之为伪彩色）显示，可以实现 VGA（640×480）、SVGA（800×600）等一些较高的分辨率，但由于构成它们的矩阵方式是无源矩阵，每个像素实际上是个无极电容，容易出现串扰现象，从而不能显示真正的活动图像。

TFT 为薄膜晶体管有源矩阵液晶显示器件，在液晶显示屏的每个像素点上设计一个薄膜晶体管来直接驱动，从而可以大大提高液晶显示器的对比度，响应时间，色彩还原能力。由于每个节点都相对独立，并可以连续控制，不仅提高了显示屏的反应速度，同时可以精确控制显示色阶，这样就容易实现真彩色、高分辨率的液晶显示器件。现在的 TFT 型液晶一般都实现了 18bit 以上的彩色（256K 色），在分辨率上，已经实现 VGA（640×480）、SVGA（800×600）、XGA（1024×768）、SXGA（1280×1024）甚至 UXGA（1600×1200）。

按点像素分：单色屏、4 级灰度屏、8 级灰度屏、16 级灰度屏、64 级灰度屏、256 级灰度屏、16 色屏、256 色伪彩色屏、TFT 真彩色屏等。

触摸屏：电阻式触摸屏（四线电阻式触摸屏、五线电阻式触摸屏）、表面声波触摸屏、电容式触摸屏、红外线触摸屏等。

对于点阵式图形液晶显示屏，为了提高人机交互的友好性，常常在显示屏上粘上一层透明的薄膜体层，用于检测屏幕触摸输入信号，形成触摸屏。

四线电阻式触摸屏（简称四线式触摸屏）包含两个透明的阻性层。其中一层在屏幕的左右边缘各有一条垂直总线，另一层在屏幕的底部和顶部各有一条水平总线，如图 6.73 所示。四线式触摸屏是最常用的触摸屏之一，所以这里将对其作重点介绍。

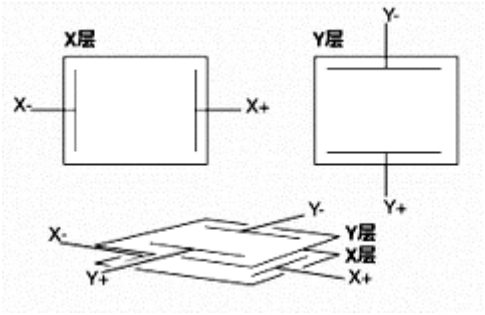


图 6.73 四线式触摸屏结构示意图

触摸屏的两个金属导电层分别用来测量 X 轴和 Y 轴方向的坐标。用于 X 坐标测量的导电层从左右两端引出两个电极，记为 X+和 X-。用于 Y 坐标测量的导电层从上下两端引出两个电极，记为 Y+和 Y-。这就是四线电阻式触摸屏的引线构成。

当在一对电极上施加电压时，在该导电层上就会形成均匀连续的电压分布。若在 X 方向的电极对上施加一确定的电压，而 Y 方向电极对上不加电压时，在 X 平行电压场中，触点处的电压值可以在 Y+（或 Y-）电极上反映出来，通过测量 Y+电极对地的电压大小，便可得知触点的 X 坐标值。同理，当在 Y 电极对上加电压，而 X 电极对上不加电压时，通过测量 X+（或 X-）电极的电压，便可得知触点的 Y 坐标值。测量原理如图 6.74 所示。

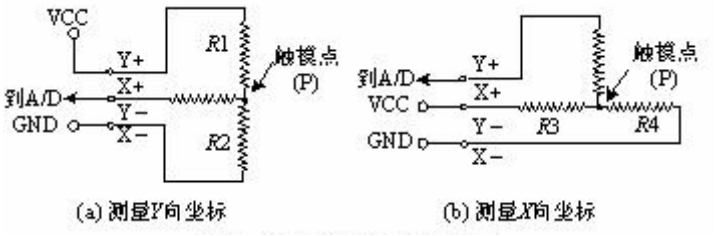


图 6.74 四线式触摸屏测量原理

在使用触摸屏时，需要一个 ADC 转换器将模拟信号转换成数字信号，通常直接使用触摸屏控制器完成这一功能，也可以使用微处理器内部的 ADC 转换器实现。触摸屏控制器的主要功能是在微处理器的控制下向触摸屏的两个方向分时施加电压，并将相应的电压信号传送给自身 A/D 转换器，在微处理器 SPI 口提供的同步时钟作用下将数字信号输出到微处理器。常见的触摸屏控制器如：ADS7843/7846、MK715 等等。

3. 点阵式液晶模块

在嵌入式系统应用中，如果微控制器本身带有液晶驱动控制功能，则可以直接对点阵式液晶显示屏进行连接控制；如果微控制器本身没有液晶驱动控制功能，则需要外扩液晶驱动板来连接液晶显示屏，或者使用点阵式图形液晶显示模块。

由于点阵式液晶显示屏的引脚较多，生产厂家通常会将液晶显示屏和驱动电路装配在一起，形成液晶模块，即 LCD Module，简称 LCM。LCM 是液晶显示产品的另外一种商品形式，它是将液晶显示器件、连接件、驱动和控制集成电路（有些液晶模块需要外接液晶控制器）、PCB 线路板、背光源、结构件装配在一起的组件。液晶模块在很大程度上方便了用户的使用，用户只要将其与微控制器连接，即可进行图形的显示输出控制。

常用点阵液晶控制器如表 6.18 所示。

表 6.18 常用点阵液晶控制器

控制器名称	工作电压	驱动液晶	最大点阵数	生产公司	备注
SED1353F	3. 7～5.5V	单色	1024×1024	EPSON	
		4 级灰度/伪彩色	1024×512		
		16 级灰度/伪彩色	512×512		
		256 伪彩色	512×256		
SED1354F	4. 7～5.5V	16 级灰度/伪彩色	800×600	EPSON	
		256 级灰度/伪彩色	800×600		
		4096 色伪彩色	800×600		
		9/12 位 TFT 液晶	800×600		
		18 位 TFT 液晶	800×600		
T6963C	5V	单色	640×128	TOSHIBA	片内 128 种 5×8、6×8、7×8、8×8 字符 CGROM

4. LPC2000 与液晶模块接口

点阵式液晶模块一般是采用并行接口进行数据传送，对于 LPC2200 系列微控制器可以采用外部存储器接口与液晶模块进行连接，对于 LPC2100 系列微控制器可以使用 I/O 模拟总线的方式与液晶模块进行连接。

这里以 SMG240128A 点阵图形液晶模块为例，介绍如何与 LPC2000 系列微控制器连接使用。

液晶模块介绍

SMG240128A 点阵图形液晶模块的点像素为 240×128 点，黑色字/白色底，STN 液晶屏，视角为 6:00，内嵌控制器为东芝公司的 T6963C，外部显示存储器为 32K 字节。SMG240128A 点阵图形液晶模块如图 6.75 所示，模块的电路原理框图如图 6.76 所示。

液晶模块采用 8 位总线接口与微控制器连接，内部集成了负压 DC-DC 电路（LCD 驱动电压），使用时只需提供单 5V 电源即可。

液晶模块上装有 LED 背光，使用 5V 电源供电，显示字符或图形时 LED 背光可点亮或熄灭。

液晶模块引脚说明如表 6.19 所示。

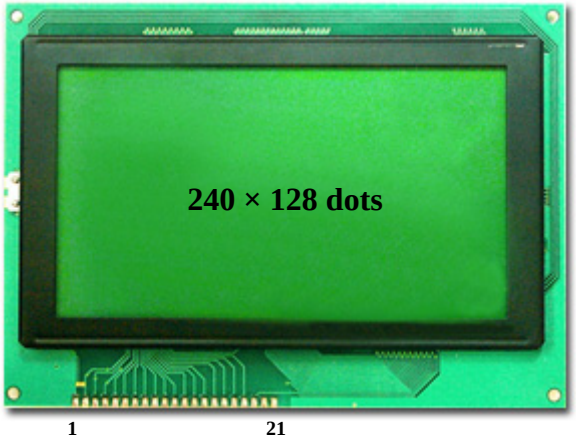


图 6.75 SMG240128A 点阵图形液晶模块

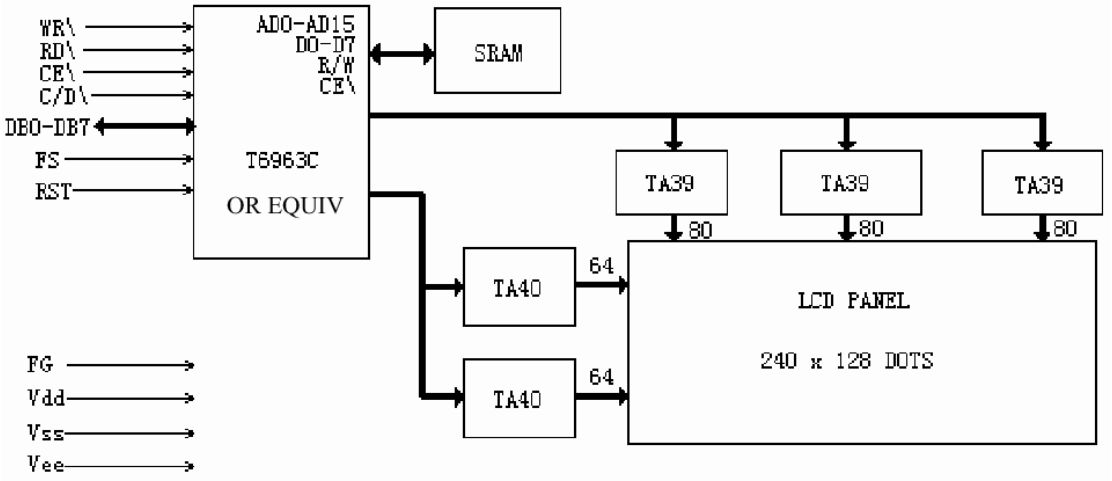


图 6.76 SMG240128A 点阵图形液晶模块原理框图

表 6.19 SMG240128A 点阵图形液晶模块引脚说明

引脚	符号	说明	备注
1	FG	显示屏框架外壳地	接地
2	Vss	电源地	
3	Vdd	电源（+5V）	
4	Vo	LCD 驱动电压（对比度调节负电压输入）	
5	Wr	写操作信号，低电平有效	
6	Rd	读操作信号，低电平有效	
7	CE	片选信号，低电平有效	
8	C/D	C/D=H 时，Wr=L ： 写命令；Rd=L ： 读状态 C/D=L 时，Wr=L ： 写数据；Rd=L ： 读数据	
9	Reset	复位，低电平有效	
10	DB0	数据总线位 0	
11	DB1	数据总线位 1	
12	DB2	数据总线位 2	
13	DB3	数据总线位 3	

接上表

引脚	符号	说明	备注
14	DB4	数据总线位 4	
15	DB5	数据总线位 5	
16	DB6	数据总线位 6	
17	DB7	数据总线位 7	
18	FS	字体选择，为高时 6*8 字体，为低时 8*8 字体	
19	Vout	DC-DC 负电源输出	
20	LED+	背光灯电源正端	
21	LED-	背光灯电源负端	

液晶控制器 T6963C

T6963C 东芝公司点阵式图形液晶控制器，T6963C 常用于中规模的单色点阵图形液晶的显示控制器，其最大特点是具有独特的硬件初始化值设置功能，显示驱动所需的参数如占空比系数、驱动传输的字节数/行、字符的字体选择等均由引脚电平设置。相关参数如下：

工作电压：5.0V

最大驱动液晶点阵：单色 640×128（单屏）

支持存储器大小：64K 字节 SRAM

MCU 接口：8 位并行数据接口

显示方式：图形方式、文本方式、图形文本混合方式

字符发生器：128 种 5×8、6×8、7×8、8×8 字符（CGROM）

可管理外部 2K 字节的 CGRAM（8×8 字符）

驱动 LCD 占空比： 1/16～1/128

低功耗：显示期间电流典型值 3.3mA

T6963C 的控制是通过 8 位数据总线发送命令及数据，读写操作时序图如图 6.77 所示，时序参数如表 6.20 所示。

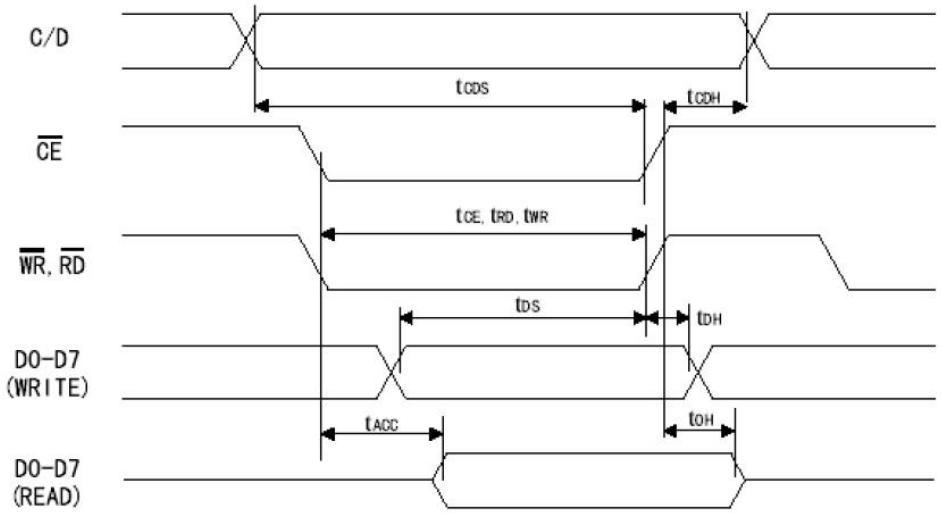


图 6.77 T6963C 读写操作时序图

表 6.20 T6963C 读写操作时序参数表

时序参数	符号	最小值	典型值	最大值	单位	测试条件
C/D 建立时间	t_{CDS}	100	—	—	nS	引脚 C/D
C/D 保持时间	t_{CDH}	10	—	—	nS	
片选、读、写脉冲宽度	t_{CE}，t_{RD}，t_{WR}	80	—	—	nS	—
数据建立时间（写操作）	t_{DS}	80	—	—	nS	引脚 DB0～DB7
数据保持时间（写操作）	t_{DH}	40	—	—	nS	
数据建立时间（读操作）	t_{ACC}	—	—	150	nS	
数据保持时间（读操作）	t_{OH}	10	—	50	nS	

T6963C 指令码表如表 6.21 所示。

表 6.21 T6963C 控制指令码表

指令	编码	数据 1	数据 2	功能
寄存器设置	0010 0001	X 地址	Y 地址	设置光标位置
	0010 0010	数据	00H	设置起始寄存器
	0010 0100	地址低 8 位	地址高 8 位	设置地址指针
设置控制字	0100 0000	地址低 8 位	地址高 8 位	设置文本起始地址
	0100 0001	列	00H	设置文本区宽度
	0100 0010	地址低 8 位	地址高 8 位	设置图形起始地址
	0100 0011	列	00H	设置图形区宽度
模式设定	1000 x000	—	—	逻辑“或”模式
	1000 x001	—	—	逻辑“异或”模式
	1000 x010	—	—	逻辑“与”模式
	1000 x011	—	—	文本特征模式
	1000 0xxx	—	—	内部 CG ROM 模式
	1000 1xxx	—	—	外部 CG ROM 模式
显示模式	1001 0000	—	—	关闭显示
	1001 xx10	—	—	打开光标，闪烁关闭
	1001 xx11	—	—	打开光标，闪烁打开
	1001 01xx	—	—	打开文本方式，关闭图形方式
	1001 10xx	—	—	关闭文本方式，打开图形方式
	1001 11xx	—	—	图形文本混合方式
光标形式	1010 0000	—	—	1 条线
	1010 0001	—	—	2 条线
	1010 0010	—	—	3 条线
	1010 0011	—	—	4 条线
	1010 0100	—	—	5 条线
	1010 0101	—	—	6 条线
	1010 0110	—	—	7 条线
	1010 0111	—	—	8 条线

接上表

指令	编码	数据 1	数据 2	功能
数据自动读写	1011 0000	—	—	数据自动写入设定
	1011 0001	—	—	数据自动读出设定
	1011 0010	—	—	自动复位
数据读写	1100 0000	数据	—	数据写入，地址自动增量
	1100 0001	—	—	数据读出，地址自动增量
	1100 0010	数据	—	数据写入，地址自动减量
	1100 0011	—	—	数据读出，地址自动减量
	1100 0100	数据	—	数据写入，地址保持不变
	1100 0101	—	—	数据读出，地址保持不变
屏幕读取	1110 0000	—	—	
屏幕拷贝	1110 1000	—	—	
位 设置/复位	1111 0xxx	—	—	位 复位
	1111 1xxx	—	—	位 设置
	1111 x000	—	—	0 位 （LSB）
	1111 x001	—	—	1 位
	1111 x010	—	—	2 位
	1111 x011	—	—	3 位
	1111 x100	—	—	4 位
	1111 x101	—	—	5 位
	1111 x110	—	—	6 位
	1111 x111	—	—	7 位 （MSB）

注： 发送指令时，控制 C/D=1；参数和数据操作时，控制 C/D=0。

在向 T6963C 发送命令前，要先判断 T6963C 的当前状态，只有在允许接收命令时才能向器件发送命令。T6963C 状态字及说明如表 6.22 所示。

表 6.22 T6963C 状态字

状态位	功能	说明
S0	命令接收状态	0 为禁止，1 为允许
S1	数据读/写状态	0 为禁止，1 为允许
S2	自动模式读	0 为禁止，1 为允许
S3	自动模式写	0 为禁止，1 为允许
S4	保留	
S5	控制器运行状态	0 为禁止，1 为使能
S6	屏读/屏拷贝出错状态	0 为正确，1 为出错
S7	闪烁状态查询	0 为关闭，1 为显示

SMG240128A 液晶模块中，具有外部显示存储器为 32K 字节，显示存储器与显示屏的关系如图 6.78 所示。液晶显示区域映射图如图 6.79 所示，写为 1 的对应点显示黑色，写为 0 的对应点显示白色。

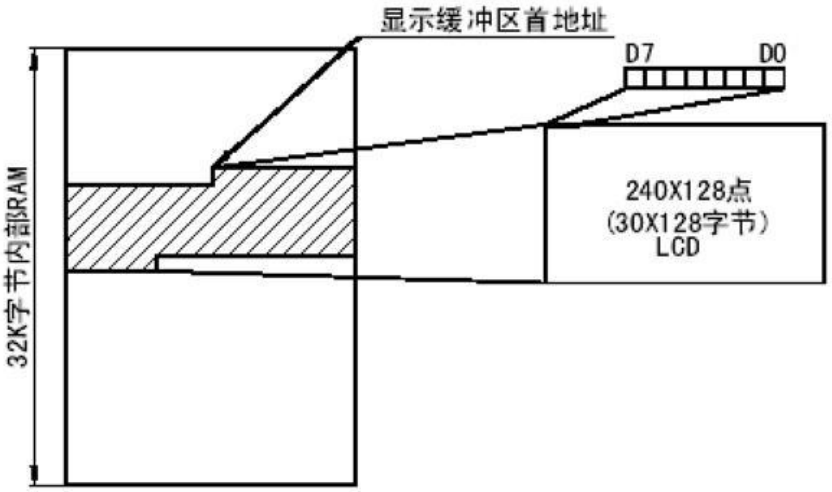


图 6.78 显示存储器与显示屏的关系

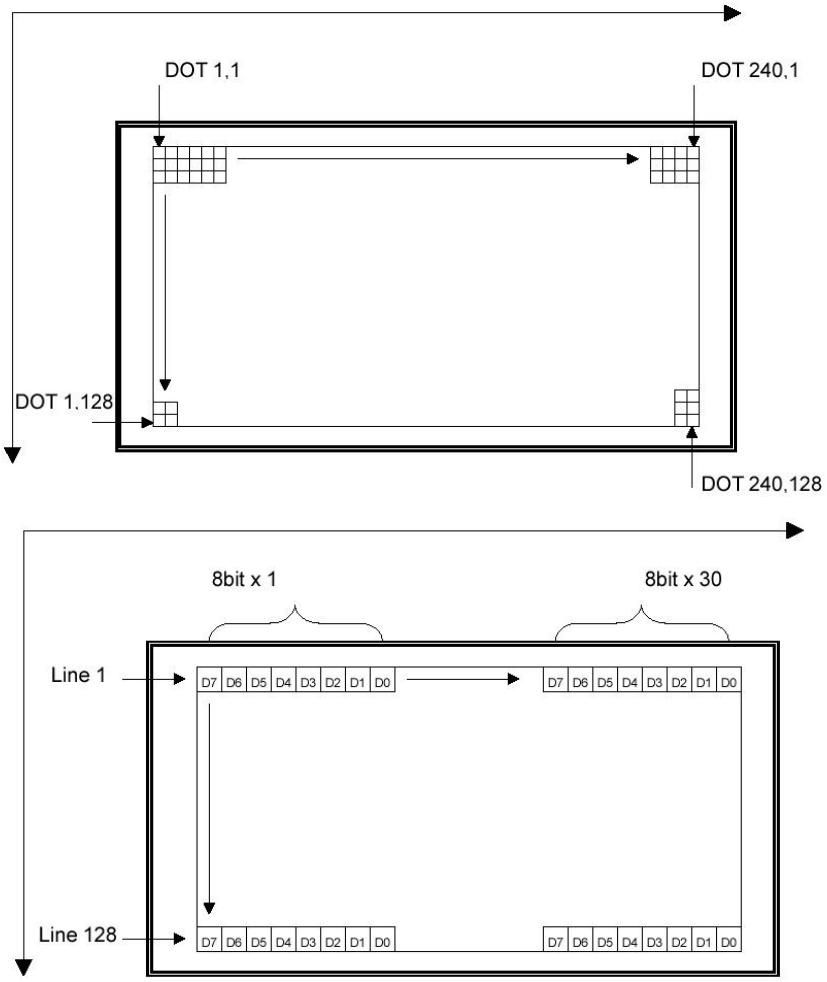


图 6.79 液晶显示区域映射图

液晶模块与 LPC2100 应用连接

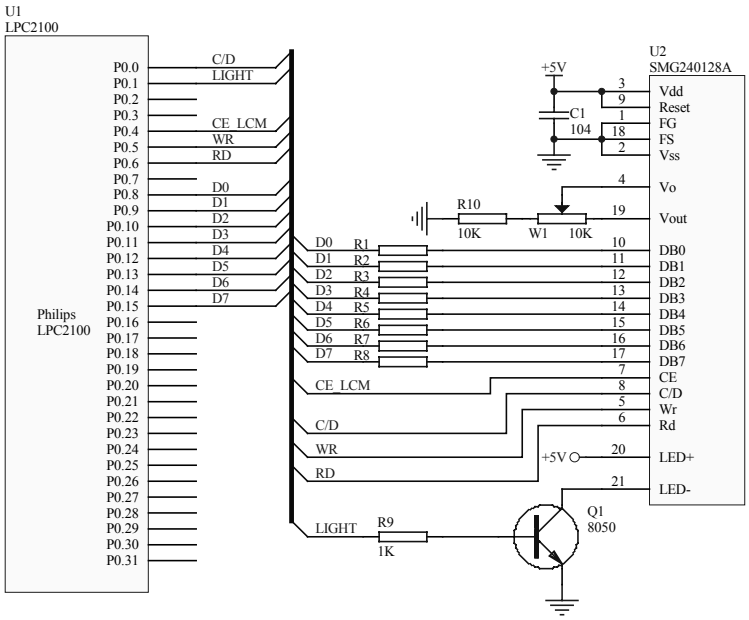


图 6.80 SMG240128A 点阵图形液晶模块应用连接电路—LPC2100

SMG240128A 与 LPC2100 系列微控制器的接口电路参考图 6.80。使用 I/O 口 P0.8～P0.15 作为 8 位总线与 SMG240128A 的 DB0～DB7 相连，片选信号由 P0.4 控制，读写信号由 P0.5、P0.6 输出控制。SMG240128A 的 C/D 引脚与 P0.0 相连，由 P0.0 进行命令或数据的选择。

液晶模块与 LPC2200 应用连接

SMG240128A 与 LPC2200 系列微控制器的接口电路参考图 6.81。采用 8 位总线方式连接 SMG240128A 图形液晶模块，该模块没有地址总线，显示地址和显示数据均通过 DB0～DB7 接口实现。图形液晶模块的 C/D 与 A1 连接，使用 A1 控制模块处理数据/命令。模块的片选信号由 LPC2210 的 CS3 控制，当 nCS3 为 0 电平时，模块被选中，所以其数据操作地址为 0x83000000，命令操作地址为 0x83000002。

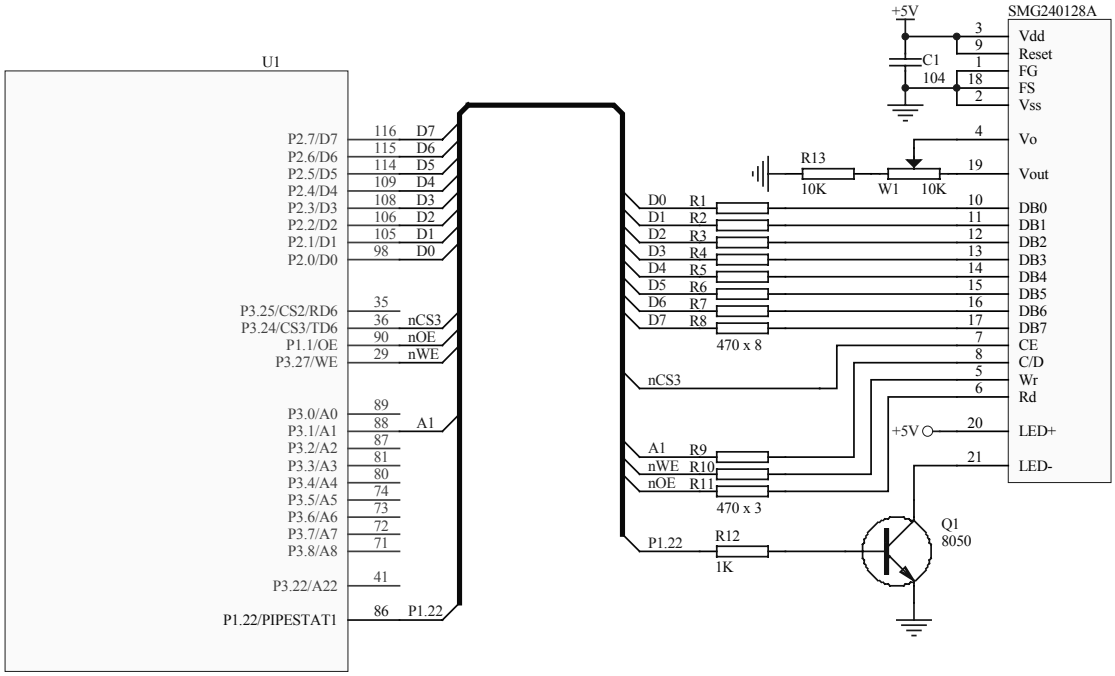


图 6.81 SMG240128A 点阵图形液晶模块应用连接电路—LPC2200

6.3.5 网络接口

TCP/IP 协议和以太网协议是使用最广泛的通讯协议，如果一个嵌入式系统没有以太网接口，其价值将大打折扣。基于底层的以太网协议的实现就由以太网控制器来负责，目前比较常用的 10Mbps 嵌入式以太网控制芯片有 RTL8019AS、CS8900 等，而 100Mbps 的就有 LAN91C111 等，我们先以 RTL8019AS 为例子介绍以太网的控制芯片。

RTL8019AS 简介

RTL8019AS 是一种高度集成的以太网控制芯片，能简单地实现 Plug and Play 并兼容 NE2000、掉电等特性。在全双工模式下，如果是连接到一个同样是全双工的交换机或集线器，就可实现同时接收和发送。这个特性虽然不能把传输速率从 10Mbps 提高到 20Mbps，但是在执行以太网 CSMA/CD 协议时，可以避免更多的冲突的发生。而 Microsoft's Plug and Play 功能就可以为用户减轻对资源配置的烦恼（如 IRQ 、I/O address 等）。又或者是在一些特殊的场合，为了对一些不支持 Microsoft's Plug and Play 的器件的兼容，RTL8091AS 还可以选择跳线模式或非跳线模式。

- 支持 PnP 自动检测模式；
- 支持 Ethernet II 和 IEEE802.3 10Base5, 10Base2, 10BaseT；
- 软件兼容 8 位或 16 位的 NE2000 模式；
- 支持跳线和非跳线模式；
- 支持在非跳线模式下的 Microsoft‘s Plug and Play 配置；
- 支持在全双工模式下的双倍信道带宽；
- 支持 UTP、AUI 、BNC 的自动检测；
- 在 10BaseT 下支持自动极性修正；
- 支持 8 路中断请求（IRQ）；
- 支持 16 位 I/O 地址；
- 内建 16K SRAM；
- 支持四盏可编程诊断 LED。

RTL8019AS 管脚排列及内部功能框图如图 6.82 和图 6.83 所示。

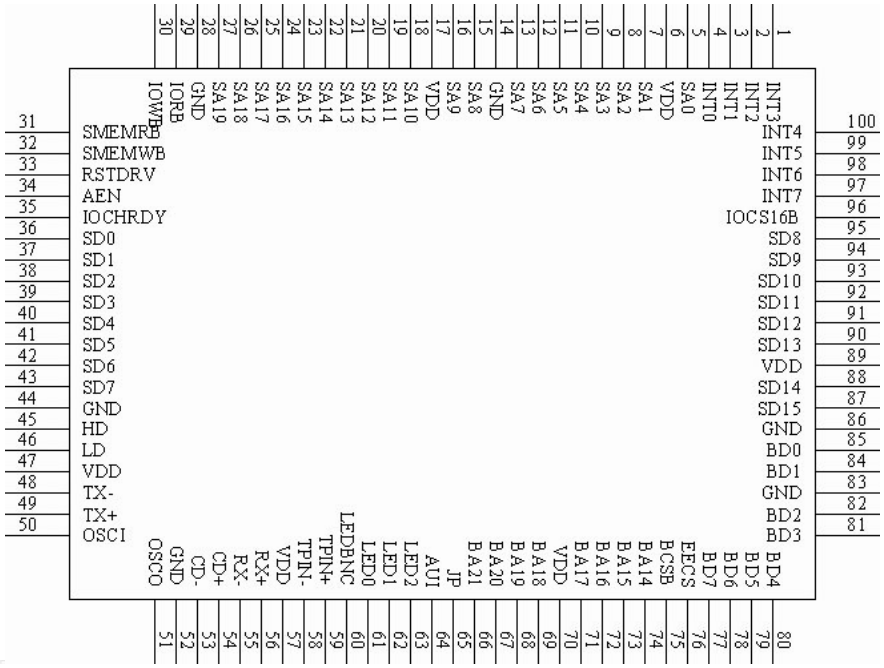


图 6.82 RTL8019AS 管脚排列图

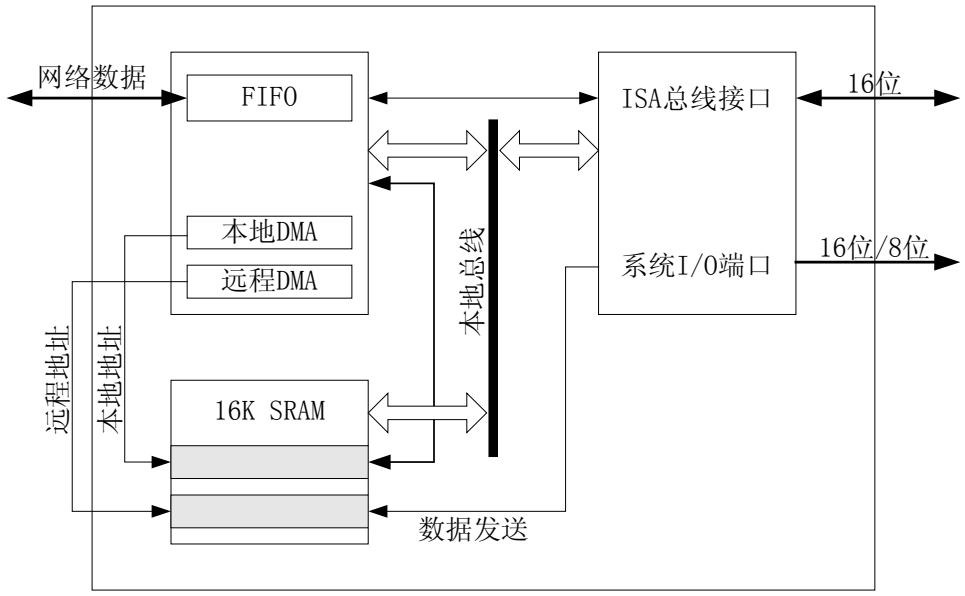


图 6.83 RTL8019AS 内部功能框图

如图 6.83 所示，RTL8019AS 芯片内部集成了 DMA 控制器、ISA 总线控制器和集成 16K SRAM、网络 PHY 收发器。用户可以通过 DMA 方式把需要发送的数据写入片内 SRAM 中，让芯片自动将数据发送出去；而芯片在接收到数据后，用户也可以通过 DMA 方式将其读出。

RTL8019AS 的详细介绍请见 RTL8019AS 数据手册。

表 6.23 RTL8019AS 管脚列表

引脚	信号名称	方向	描述
6，17，47，57，70，89	VDD	P	+5V
14，28，,44，52，83，86	GND	P	地
34	AEN	I	地址使能，为“0”是 I/O 命令有效。
97－100，1－4	INT7-0	O	中断请求。
35	IOCRDY	O	置 0 插入等待周期确认主机的读写命令。
96	IOCS16B [SOLT16]	O	在上电复位后，RTL8019AS 检测该管脚以决定使用 16 位还是 8 位的 ISA 插槽。
29	IORB	I	主机读命令。
30	IOWB	I	主机写命令。
33	RSTDRV	I	复位引脚。
27-18，16-15，13-7，5	SA19-0	I	主机地址总线。
87-88，90-95，43－36	SD15-0	I/O	主机数据总线。
31	SMEMRB	I	主机内存读命令。
32	SMEMWB	I	主机内存写命令。
75	BCSB	O	BROM 芯片的选择。
76	EECS	O	9346 的片选

接上表

引脚	信号名称	方向	描述
66－69，71－74 77－82，84－85	BA21-14 BD7-0	O I/O	BROM address； BROM data bus。
79 78 77	EESK EEDI EEDO	O O I	9346 串行数据时钟； 9346 串行数据输入； 9346 串行数据输出。
66 72－71，69－67 85－84，82－81 70，74 80－78 65	PNP BS4-0 IOS3-0 PL1-0 IRQS2-0 JP	I I I I I I	PnP 模式选择； 选择 BROM 的大小和基地址； 选择 I/O 的基地址； 选择网络传播介质的类型； 选择在 INT7－0 使用那路中断； 跳线模式选择。
64	AUI	I	检测外部是否使用 AUI 接口来传输数据。
54，53	CD+,CD-	I	AUI 的冲突差分信号输入线。
56，55	RX+,RX-	I	AUI 的接收信号线。
49，48	TX+,TX-	O	AUI 的发送信号线。
59，58	TPIN+, TPIN-	I	为双绞线的接收脚。
45，46	TPOUT+, TPOUT-	O	为双绞线的发送脚。
50	X1	I	20MHz 晶振或外部时钟的输入端。
51	X2	O	晶振反馈输出端，接外时钟时不接。
60	LEDBNC	O	连接自动显示。
61	LED0-	O	溢出显示。
62，63	LED1,LED2	O	发送和接受显示。

明白了 RTL8019AS 所提供的资源与硬件接口，便可以设计 RTL8019AS 与 LPC2200 的硬件电路了。

RTL8019AS 与 LPC2200 的硬件电路设计

RTL8019AS 与 LPC2200 一般通过外部总线进行连接。我们假设 RTL8019AS 与 LPC2200 的连接关系如表 6.24 所示。

表 6.24 RTL8019AS 与 LPC2200 连接关系

RTL8019AS	功 能	LPC2200
SD0 ~ SD15	RTL8019AS 数据总线	D0 ~ D15
SA0~SA4	RTL8019AS 地址总线	A1~A5
SA8	RTL8019AS 地址总线	A22
SA5	RTL8019AS 地址总线	nCS3
IORB	RTL8019AS 读使能（低电平有效）	nOE
IOWB	RTL8019AS 写使能（低电平有效）	nEW
INT0	RTL8019AS 中断输出信号	INT_N（P0.7）
RSTDRV	RTL8019AS 复位输入信号	NET_RST（P0.6）

LPC2200 与 RTL8019AS 构成以太网接口的具体电路如图 6.84 所示。

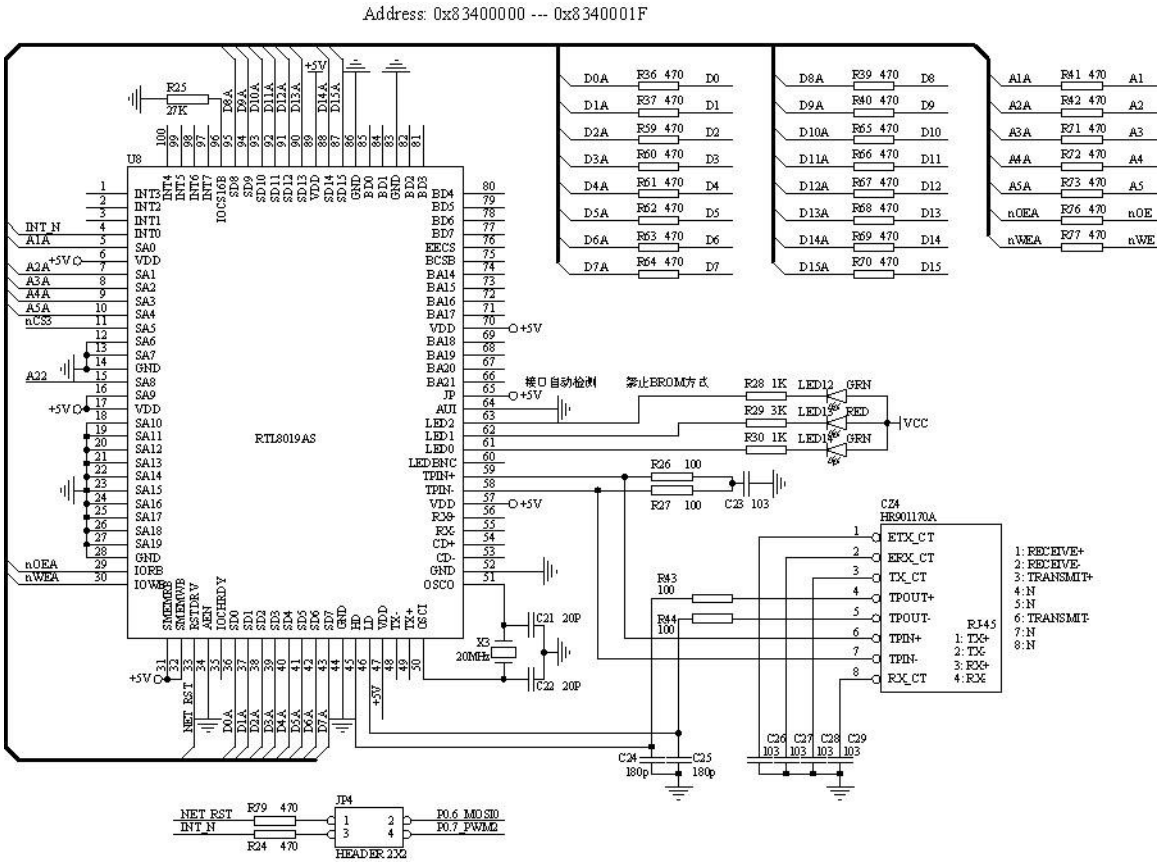


图 6.84 LPC2200 与 RTL8019AS 接口电路图

由图 6.84 和表 6.24，可知 RTL8019AS 使用 LPC2200 外部存储控制的 Bank3 部分，而 RTL8019AS 的 IO 地址为 0x00300~0x0031F，所以 RTL8019AS 在 SA8=1;SA5=0 的时候选通，其地址如下：

数据地址 0x83400000~0x83400001F

NET_RST 为 LPC2200 输出引脚，RTL8019AS 中断信号为中断输入信号，且为外部中断。RTL8019AS 的 SD0~SD15 串了一个 470 欧姆电阻连接到 LPC2200 的 D0~D15。

至于 LPC2100 系列芯片没有外部总线控制器，如果需要与 RTL8019AS 连接，需要用 GPIO 模拟总线（参考 6.2.1 小节的相关部分），具体电路可以参考图 6.85。

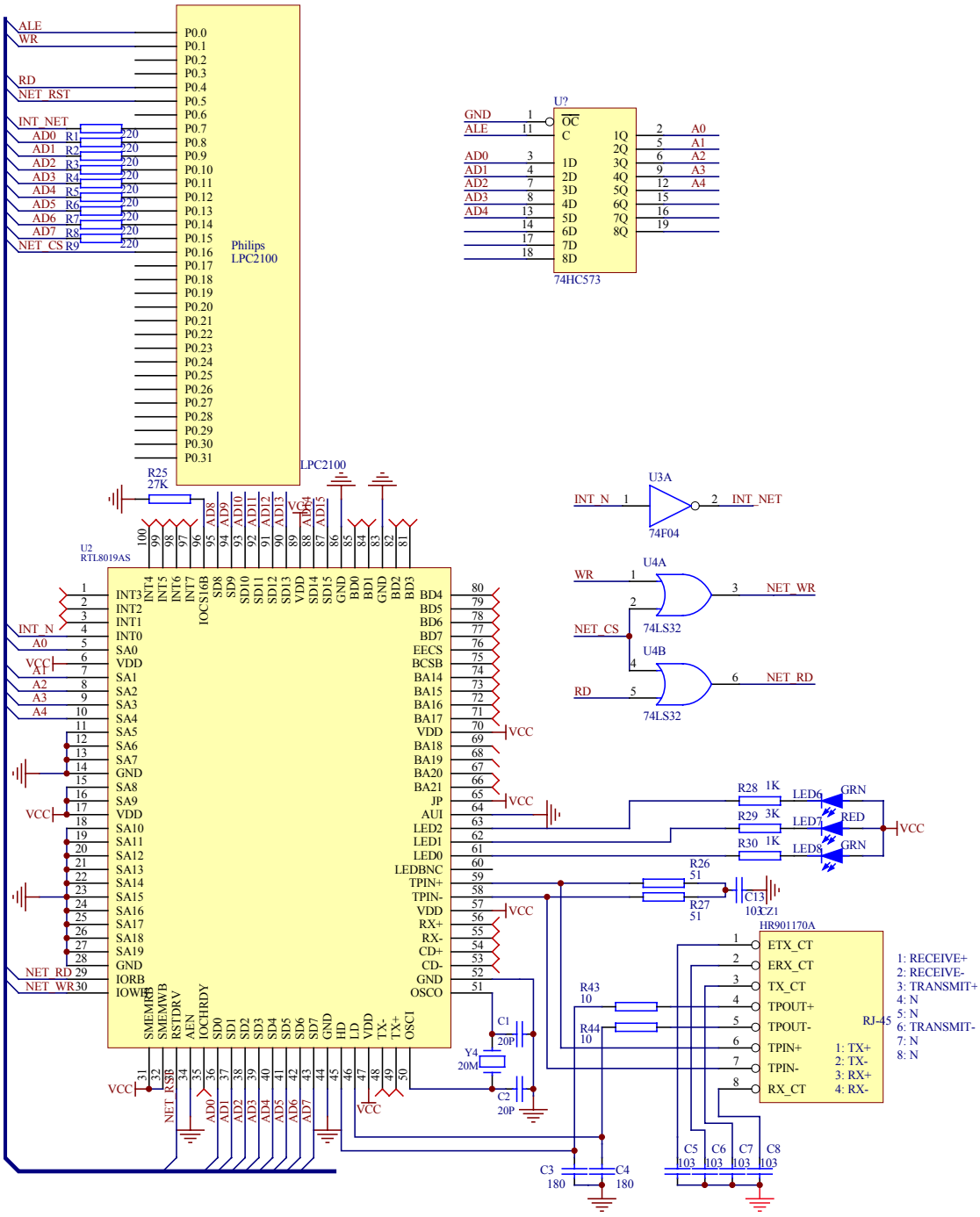


图 6.85 LPC2100 与 RTL8019AS 连接电路图

CS8900

CS8900 芯片是 Cirrus Logic 公司生产的一种以太网处理芯片，它的封装是 100-pin TQFP，内部集成了 4KB SRAM、10BASE-T 收发滤波器，提供 8 位和 16 位两种接口。它的与 RTL8019AS 的不同点有：

- 内建 4KB SRAM（16KB）；
- 工作电压为 3V（5V）；
- 提供工业级芯片（只提供商业级）。

注意：括号内为 RTL8019AS 的功能特点！

LAN91C111 简介

LAN91C111 是标准半导体公司生产的一种面向嵌入式应用的 100/10Mbps 自适应以太网控制器。其功能特点如下：

- 支持 PnP 自动检测模式；
- 支持 10/100Mbps 全双工模式；
- 软件兼容 8 位或 16 位、32 位 CPU 访问模式；
- 芯片内部 32 位数据总线；
- 支持数据突发传输；
- 支持多种嵌入式处理器外部总线；
- 内建 8KB FIFO 缓存。

LAN91C111 管脚排列及内部功能框图如图 6.86、图 6.87 所示。

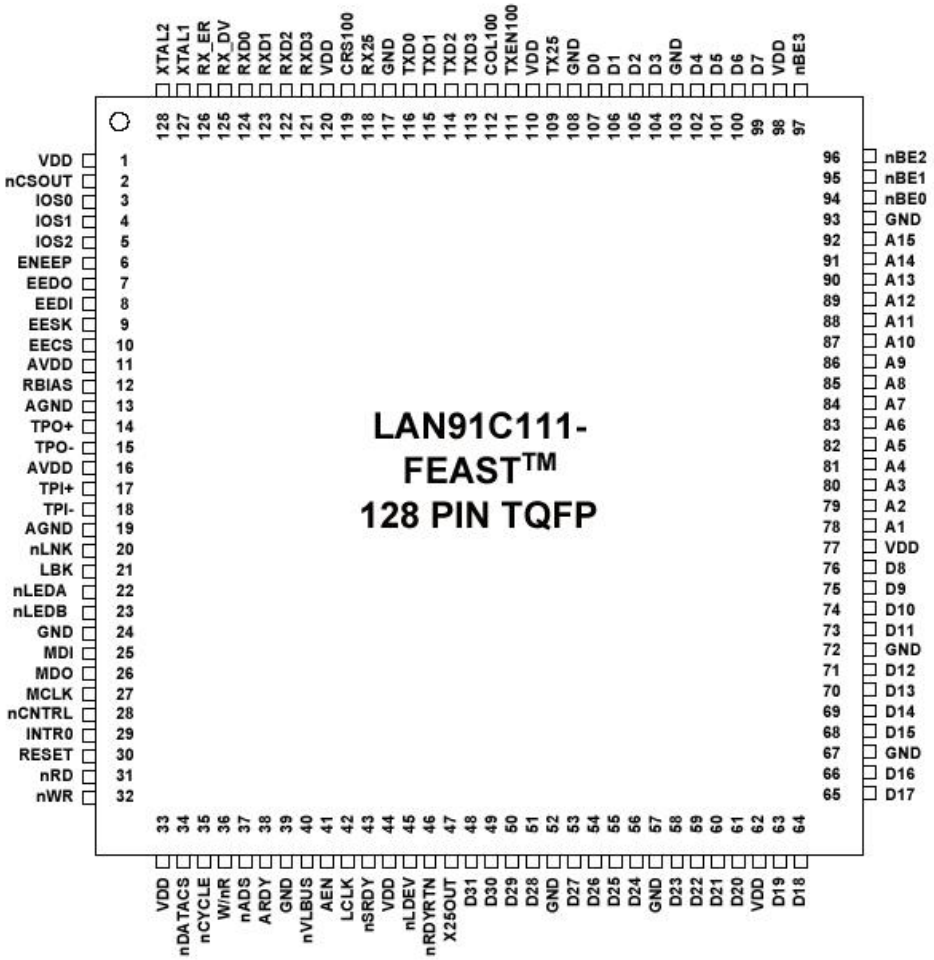


图 6.86 LAN91C111 管脚排列图

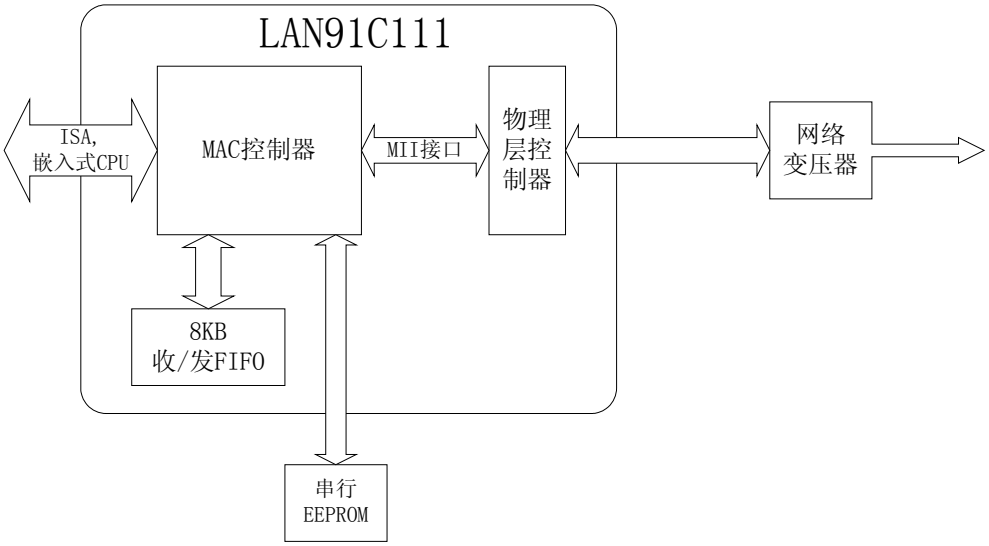


图 6.87 LAN91C111 内部功能框图

如图 6.87 所示，LAN91C111 芯片内部集成了 MAC 控制器、物理层控制器。MAC 控制器可以把数据从 FIFO 中发送到物理层控制器中，再由物理层控制器发送到网络。而芯片于 CPU 的接口比较丰富，如 ISA 总线或者其它嵌入式处理器的外扩总线都可以与其连接。

LAN91C111 的详细介绍请见 LAN91C111 数据手册。

表 6.25 LAN91C111 管脚列表

引脚	信号名称	方向	描述
1，33，44，62，77，98，110，120	VDD	P	+3.3V 电源供电脚
24，39，,52，57，67，72，93，108，117	GND	P	地
11，16	AVDD	P	+3.3V 模拟电源供电脚
13，19	AGND	P	模拟地
78~91	A1~A15	I	地址线。
41	AEN	I	地址使能
94~97	nBE0～nBE3	I	数据发送位数的选择
48~51，53~56，58~61，63~66，68~71，73~76，99~102，104~107	D0~D31	I/O	32 位数据线
30	RESET	I	复位
37	nADS	I	地址锁存
35	nCYCLE	I	同步传输总线时钟
36	W/nR	I	同步传输时的读写控制
40	nVLBUS	I	VL BUS 访问使能
42	LCLK	I	同步突发传输的时钟
38	ARDY	O	异步总线 ready

接上表

引脚	信号名称	方向	描述
43	nSRDY	O	同步总线 ready
46	nRDYRTN	I	同步读信号
29	INT0	O	中断输出
45	nLDEV	O	释放地址和 AEN 信号
31	nRD	I	异步读信号
32	nWR	I	异步写信号
34	nDATACS	I	数据线片选
9 10 7 8	EESK EECS EEDO EEDI	O O O I	EEPROM 连接管脚
3~5	IOS0~IOS3	I	I/O BASE 地址选择
6	ENEEP	I	EEPROM 使能
127 128	XTAL1 XTAL2	CLK	25MHz 晶振连接端
21	LBK	O	回环输出
20	nLNK	I	连接状态输入
28	nCNTRL	O	通用控制端
47	X25out	O	25MHz 频率输出端
111	TXEN100	O	MII 使能 100MHz 传输
119	CRS100	I	MII 接口
125	RX_DV	I	MII 接口
112	COL100	I	MII 接口
113~116	TXD0~TXD3	O	MII 接口
109	TX25	I	MII 接口
118	RX25	I	MII 接口
121~124	RXD0~RXD3	I	MII 接口
25	MDI	I	MII 接口
26	MDO	O	MII 接口
27	MCLK	O	MII 接口
126	RX_ER	I	MII 接口
2	nCSOUT	O	对外部 PHY 的片选
12	RBIAS	NA	传输电流控制
14 15	TPO＋ TPO－	O	双绞线连接输出端
17 18	TPI＋ TPI－	I	双绞线连接输入端
22	nLEDA	O	LED 输出
23	nLEDB	O	LED 输出

明白了 LAN91C111 所提供的资源与硬件接口，便可以设计 LAN91C111 与 LPC2200 的

硬件电路了。

LAN91C111 与 LPC2200 的硬件电路设计

LAN91C111 与 LPC2200 一般通过外部总线进行连接。我们可以假设 LAN91C111 与 LPC2200 的连接关系如表 6.26 所示。

表 6.26 LAN91C111 与 LPC2200 连接关系

LAN91C111	功 能	LPC2200
D0 ～ D15	LAN91C111 数据总线	D0 ～ D15
A1～A3	LAN91C111 地址总线	A1～A3
A8	LAN91C111 地址总线	A22
A5	LAN91C111 地址总线	nCS2
IORB	LAN91C111 读使能（低电平有效）	nOE
IOWB	LAN91C111 写使能（低电平有效）	nEW
nBE2～nBE3	LAN91C111 高 16 位数据选通	VCC
nBE 0～nBE1	LAN91C111 低 16 位数据选通	BLE0～BLE1
AEN	LAN91C111 总线控制	nCS2
nADS	LAN91C111 总线控制	GND
LCLK	LAN91C111 总线控制	GND
nCYCLE	LAN91C111 总线控制	VCC
W/nR	LAN91C111 总线控制	VCC
nRDYRTN	LAN91C111 总线控制	VCC
nLDEV	LAN91C111 总线控制	悬空
nVLBUS	LAN91C111 总线控制	VCC
ARDY	LAN91C111 总线控制管脚	悬空
INT0	LAN91C111 中断输出信号	INT_N（P0.7）
RESET	LAN91C111 复位输入信号	NET_RST（P0.6）

由以上关系，我们可以得到 LAN91C111 与 LPC2200 的连接电路图，如图 6.88 所示。

由图 6.88 和表 6.26 可知 LAN91C111 使用 LPC2200 外部存储控制的 Bank2 部分，而 LAN91C111 的 I/O 地址为 0X00300~0X0030F，所以 LAN91C111 在 A8=1、A5=0 的时候选通，其地址如下：

数据地址 0x82400000~0x8240000F

RESET 为 LPC2200 输出引脚，LAN91C111 中断信号为中断输入信号，且为外部中断。

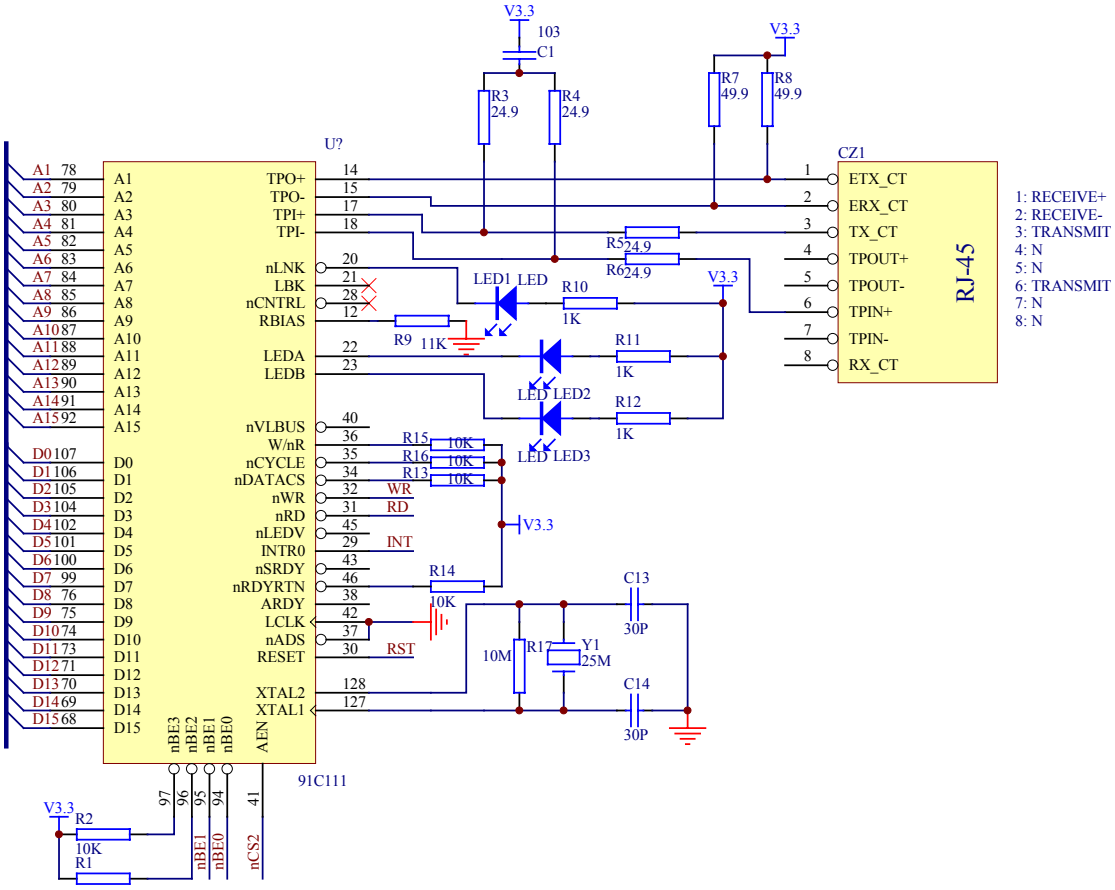


图 6.88 LAN91C111 与 LPC2200 的连接电路图

6.4 其它外设

6.4.1 并行打印机接口

打印机是重要的输出设备，很多测控仪器都需要用它来长久保存数据。在嵌入式系统中使用打印机一般有以下几种方式：

1．直接使用微型打印机的机芯，微控制器直接控制机芯工作。采用这种方式可减少设备的体积，也可能可以降低成本，但开发难度还很大，同时因为各种微型打印机的机芯不统一，带来采购的风险。

2．使用成品微型打印机，这是常用的方式。采用这种方式可减少设备的体积，微型打印机的可选范围也比较大，而且很多微型打印机的接口也比较统一。缺点是成本比较高。

3．使用普通打印机。采用这种方式成本最低，且接口统一，可选的范围也非常广，开发难度也最低。不过，其体积庞大，且对工作环境要求高，不能在工业环境下使用。

第一种方式不在本书的考虑范围内，因为在现实中这种方式很少使用。各种打印机的原理本书也不打算介绍，原因是这方面的资料很多，且打印机内部的控制电路已经屏蔽了不同原理的打印机的差别，打印机的使用者一般不需要过分关心打印机的原理。不过，不同原理的打印机有不同的应用范围，在打印机选型时还是要关心打印机的种类的。

从接口方面来说，打印基本分为标准并行打印机、标准串行打印机、USB 接口打印机、专用接口打印机及它们的组合。因为 USB 接口在微型打印机中还不普及，本书也不作介绍。并行打印机接口已经标准化（IEEE-P1284），其打印机端的接口见图 6.89，各个引脚的信号说明见表 6.27。然而，在电脑端的接口见图 6.90，各个引脚的信号说明见表 6.28，这是最基

本的信号，新的标准与表 6.28 有区别，但兼容表 6.28。

表 6.27 并行打印机接口信号—打印机端

引脚	信号名称	方向	描述
1	/STROBE		数据选通触发脉冲，上升沿时读入数据
2	D0		并行数据的第 0 位，高电平为“1”
3	D1		并行数据的第 1 位，高电平为“1”
4	D2		并行数据的第 2 位，高电平为“1”
5	D3		并行数据的第 3 位，高电平为“1”
6	D4		并行数据的第 4 位，高电平为“1”
7	D5		并行数据的第 5 位，高电平为“1”
8	D6		并行数据的第 6 位，高电平为“1”
9	D7		并行数据的第 7 位，高电平为“1”
10	/ACK		低电平表示数据已被接受而且打印机准备好接收下一数据
11	BUSY		高电平表示打印机正忙，不能接收数据
12	POUT		高电平表示打印机无纸
13	SEL		高电平表示打印机在线
14	/AUTOFEED		低电平使打印机自动换行
15	n/c	-	没有使用
16	0 V		逻辑地
17	CHASSIS GND		屏蔽地
18	+5 V PULLUP		+5 V DC （50 mA max）
19	GND		/STROBE 的信号地
20	GND		D0 的信号地
21	GND		D1 的信号地
22	GND		D2 的信号地
23	GND		D3 的信号地
24	GND		D4 的信号地
25	GND		D5 的信号地
26	GND		D6 的信号地
27	GND		D7 的信号地
28	GND		/ACK 的信号地
29	GND		BUSY 的信号地
30	/GNDRESET		/RESET 的信号地
31	/RESET		低电平复位打印机
32	/FAULT		故障（低电平表示打印机没有联机）
33	0 V		信号地
34	n/c	-	没有使用
35	+5 V		+5 V DC
36	/SLCT IN		选择输入（低电平请求打印机联机，高电平迫使打印机停止联机）

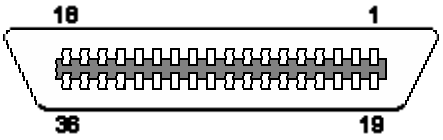


图 6.89 并行打印机接口—打印机端

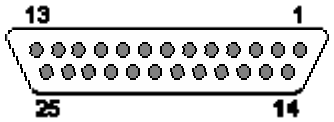


图 6.90 并行打印机接口—电脑端

表 6.28 并行打印机接口信号—电脑端

引脚	信号名称	方向	描述
1	/STROBE		数据选通触发脉冲，上升沿时数据有效
2	D0		并行数据的第 0 位，高电平为“1”
3	D1		并行数据的第 1 位，高电平为“1”
4	D2		并行数据的第 2 位，高电平为“1”
5	D3		并行数据的第 3 位，高电平为“1”
6	D4		并行数据的第 4 位，高电平为“1”
7	D5		并行数据的第 5 位，高电平为“1”
8	D6		并行数据的第 6 位，高电平为“1”
9	D7		并行数据的第 7 位，高电平为“1”
10	/ACK		低电平表示打印机已接受数据而且准备好接收下一数据
11	BUSY		高电平表示打印机正忙，不能接收数据
12	PE		高电平表示打印机无纸
13	SEL		高电平表示打印机在线
14	/AUTOFD		低电平使打印机自动换行
15	/ERROR		高电平表示打印机无故障
16	/INIT		低电平使打印机初始化
17	/SELIN		选择输入（低电平请求打印机联机，高电平迫使打印机停止联机）
18	GND		信号地
19	GND		信号地
20	GND		信号地
21	GND		信号地
22	GND		信号地
23	GND		信号地
24	GND		信号地
25	GND		信号地

并行打印机具有标准的接口和时序，不过其电压按照 5V 设计的，但 LPC2000 系列的 I/O 口为 3.3V，为保证输出的驱动能力，其数据口必须增加长线驱动器（如 74HC245、74HC573 等），而控制口也必须进行电平转换，这可以通过 CMOS 门电路实现。并行打印机接口的参考电路见图 6.91。因为 LPC2000 系列的 I/O 口可以承受 5V 输入，所以图中的电阻不是必须

的，但增加它可以增加安全性。

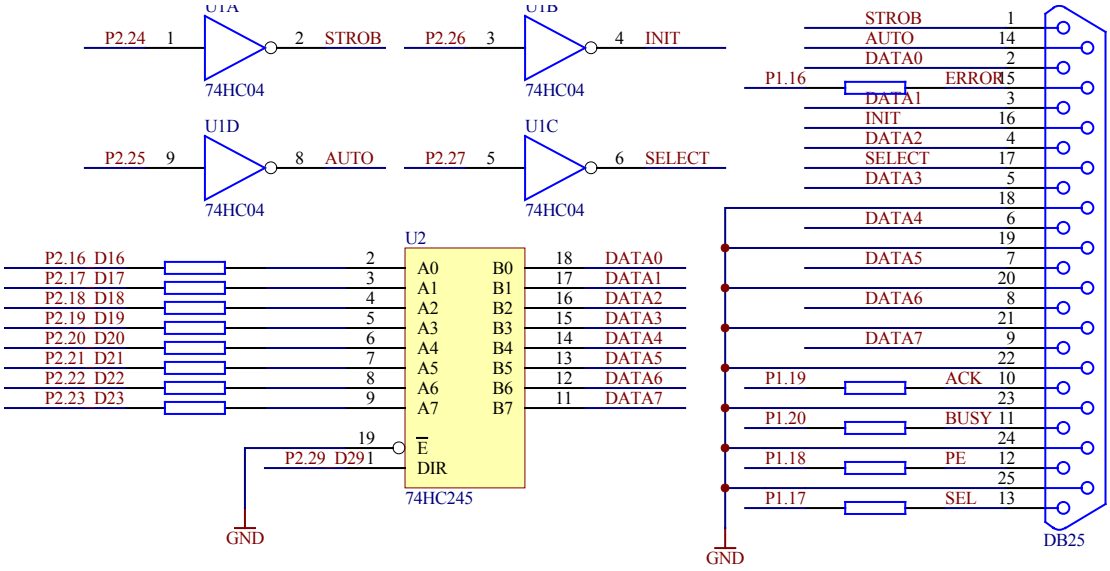


图 6.91 并行打印机接口例子

注意：图中所有电阻的阻值可选范围为 50~500 欧姆。

至于串行打印机，其接口标准不统一，需要根据厂家手册来设计。

下面以炜煌热敏打印机 WH153 为例说明 LPC2000 系列与微型打印机的硬件连接。

炜煌热敏打印机能在安静的打印环境里保持最小的噪声，被广泛应用于医疗仪器或其他需要安静打印的仪器上。高速打印、高分辨率的图象效果还能打印出优美鲜明的轮廓甚至是清晰的品牌标志和条码，具有较高的性价比。

WH153 具有面板式和平台式式种基本形式，接口有并行和串行两种方式，具体见表 6.29。

表 6.29 WH153 选型指南

型 号	打印头	纸宽/mm	每行点数	打印速度	每行字符数（5×7）/ 汉字数（16×16）	接口
WH153PA	M-153	57.5±0.5	384	30mm/秒	24-48 / 9-19	并口
WH153PT	M-153	57.5±0.5	384	30mm/秒	24-48 / 9-19	并口
WH153SA	M-153	57.5±0.5	384	30mm/秒	24-48 / 9-19	串口
WH153ST	M-153	57.5±0.5	384	30mm/秒	24-48 / 9-19	串口
注：“A”表示面板式；“T”表示平台式。						

W H 系列打印机的并行接口与标准并行接口 CENTRONICS 兼容，并行连接方式面板式和平台式插座引脚序号见图 6.92，各个引脚信号见表 6.30。

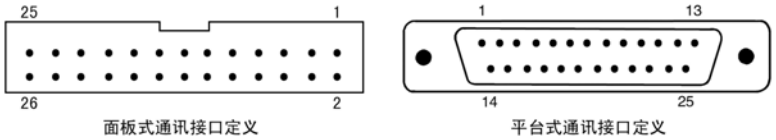


图 6.92 WH153 并行接口引脚图

表 6.30 WH153 并行接口引脚定义

面板式引脚	平台式引脚	信 号	方向	说 明
1	1	-STB	入	数据选通触发脉冲，上升沿时读入数据。
3	2	DATA1	入	并行数据第 1 位当，其逻辑为“1”时为“高”电平。
5	3	DATA2	入	并行数据第 2 位当，其逻辑为“1”时为“高”电平。
7	4	DATA3	入	并行数据第 3 位当，其逻辑为“1”时为“高”电平。
9	5	DATA4	入	并行数据第 4 位当，其逻辑为“1”时为“高”电平。
11	6	DATA5	入	并行数据第 5 位当，其逻辑为“1”时为“高”电平。
13	7	DATA6	入	并行数据第 6 位当，其逻辑为“1”时为“高”电平。
15	8	DATA7	入	并行数据第 7 位当，其逻辑为“1”时为“高”电平。
17	9	DATA8	入	并行数据第 8 位当，其逻辑为“1”时为“高”电平。
19	10	-ACK	出	回答脉冲，“低”电平表示数据已被接受而且打印机准备好接收下一数据。
21	11	BUSY	出	“高”电平表示打印机正“忙”，不能接收数据。
23		PE	—	接地。
25	13	SEL	出	打印机内部经电阻上拉“高”电平，表示打印机在线。
4	15	-ERR	出	打印机内部经电阻上拉“高”电平，表示无故障。
2,6,8,26	14,16,17			空脚。
10－24	25－18	GND	—	接地，逻辑“0”电平。

注: 1.“入”表示输入到打印机。 2.“出”表示从打印机输出。 3.信号的逻辑电平为 TTL 电平。

用户系统与 WH153 微型打印机之间需要一条电缆进行连接，假设 LPC2000 系列按照图 6.91 所示的电路设计打印接口，则电缆的设计应当按照图 6.93 所示的线路来设计。

图 6.91 的 J1 引脚		面板式引脚	平台式引脚
1		1	1
2		3	2
3		5	3
4		7	4
5		9	5
6		11	6
7		13	7
8		15	8
9		17	9
10		19	10
11		21	11
12		23	12
13		25	13
14		2	14
15		4	15
16		6	16
17		8	17
18		10	18
19		12	19
20		14	20
21		16	21
22		18	22
23		20	23
24		22	24
25		24	25
		26	

图 6.93 所示电路与 WH153 的并行接口的连接

WHxxxSx 型打印机的串行接口与 RS-232C 标准兼容，因此可直接将打印机与 IBM PC 相接。串行接口方式的面板式和平台式插座引脚序号见图 6.92，各个引脚信号见表 6.30。

表 6.31 WH153 串行接口引脚定义

10 针面板式	9 孔面板式	平台式	信号	方向	说明
5	3	2	TxD	入	打印机从主计算机接收数据
6	8	5	CTS	出	该信号高电平时，表示打印机正“忙”不能接受数据，而当该信号低电平时，表示打印机“准备好”，可以接收数据
2	6	6	DSR	出	该信号为“SPACE”状态表示打印机“在线”
9	5	7	GND	—	信号地
1	1	8	DCD	出	信号 CTS
10	-	-	+5V	出	直流+5V 3A 电源输入端

注: 1.“入”表示输入到打印机； 2.“出”表示从打印机输出； 3.信号的逻辑电平为 EIA-RS-232C 电平。

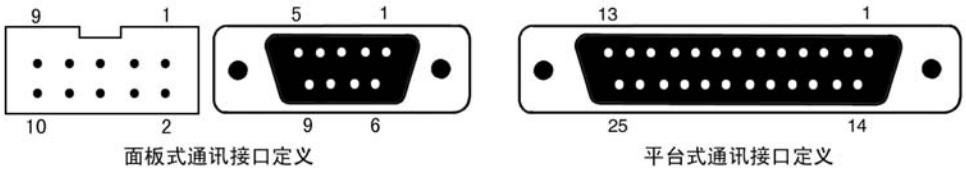


图 6.94 WH153 串行接口引脚图

至于硬件连接，可以利用 LPC2000 的 UART1，W H 系列打印机的串行接口所需要的信号在 UART1 上均存在，只要把这些信号通过 232 电平转换后一一对应地连接到串行打印机上即可，详细的接口电路可参考图 6.31。

6.4.2 CF 卡及 IDE 硬盘接口

CF 卡是一种大容量存储设备，目前已广泛应用在数码相机、PDA、MP3、工控机等嵌入式系统中。CF 卡有 PC 卡 I/O、MEMORY 及 True IDE 等 3 种模式，而 True IDE 模式兼容 IDE 硬盘，该模式比其它的两种模式更实用，是 3 种模式中使用较多的一种。本节中只介绍 CF 卡在 True IDE 模式下的接口。

使用 LPC2000 的通用可编程 I/O 口，模拟产生 ATA 设备的读写时序，实现对 CF 卡及 IDE 硬盘等 ATA 设备读写操作。使用 LPC2000 的 GPIO 功能，可以非常灵活而简单地实现 ATA 读写时序。

如表 6.32 所示，卡上所有的输入和输出引脚都被标志，除了数据总线上的准双向触发态信号。表 6.33 分别描述了 CF 卡在 True IDE 工作模式下的各引脚的功能。

表 6.32 引脚设定及引脚类型

Pin 号	信号名称	类型	Pin 号	信号名称	类型	Pin 号	信号名称	类型
1	GND		18	A02	I	35	-IOWR	I
2	D03	I/O	19	A01	I	36	-WE[3]	I
3	D04	I/O	20	A00	I	37	INTRQ	O
4	D05	I/O	21	D00	I/O	38	VCC	
5	D06	I/O	22	D01	I/O	39	-CSEL	I
6	D07	I/O	23	D02	I/O	40	-VS2	O
7	-CS0	I	24	-IOCS16	O	41	-RESET	I
8	A10[2]	I	25	-CD2	O	42	IORDY	O
9	-ATASEL	I	26	-CD1	O	43	RFU	O
10	A09[2]	I	27	D11[1]	I/O	44	RFU[4]	I
11	A08[2]	I	28	D12[1]	I/O	45	-DASP	O
12	A07[2]	I	29	D13[1]	I/O	46	-PDIAG	O
13	VCC		30	D14[1]	I/O	47	D08[1]	I/O
14	A06[2]	I	31	D15[1]	I/O	48	D09[1]	I/O
15	A05[2]	I	32	-CS1[1]	I	49	D10[1]	I/O
16	A04[2]	I	33	-VS1	O	50	GND	
17	A03[2]	I	34	-IORD	I			

注解：1．这些信号仅对 16 位系统有用，在 8 位系统中无效。设备应允许设置 3 态信号以省电。

2．主控器上的这些信号应该接地。

3．主控器上的这些信号应该与 VCC 连接。

4．该引脚应保持高电平或在主控器上与 VCC 连接。

表 6.33 CF 卡信号描述

信号名	方向	引脚	描述
A2-A0	I	18,19,20	在 True IDE 模式中，A[2:0]可用来选择 Task File（任务文件）中 8 个寄存器中的一个，其它的地址线应该被主控器设置为接地。
-PDIAG	I/O	46	在 IDE 实模式下，诊断信号可通过主/从握手协议输入/输出。
-DASP	I/O	45	在 True IDE 模式下，磁盘启动/从盘就绪信号可通过主/从握手协议输入/输出。
-CD1,-CD2	O	26,25	CF 存储卡及 CF+卡上的这些卡检测引脚接地。他们被主控器用来检测 CF 存储卡及 CF+卡是否完全插进插槽。
-CS0,-CS1	I	7,32	在 True IDE 模式下，当-CS1 用来选择辅助状态寄存器及设备控制寄存器，-CS0 为任务文件寄存器的片选信号。
-CSEL	I	39	卡内部该引脚上拉信号控制设备；当引脚接地，设备被配置为主模式，当引脚为空，设备被配置为从模式。
D15-D00	I/O	31,30,29,28,27,49,48,47,6,5,4,3,2,23,22,21	当所有的数据通过 D[15:0]进行 16 位传输时，任务文件寄存器在总线低位 D[7:0]上以字节方式操作。
GND	-	1,50	Ground。
保留	O	43	在 True IDE 模式，该输出信号无效，无需与主控器连接。
-IORD	I	34	读 CF 卡寄存器信号引脚。
-IOWR	I	35	写 CF 卡寄存器信号引脚。
-ATA SEL	I	9	为了使能 True IDE 模式，该输入信号线应被主控器接地。
INTRQ	O	37	在 True IDE 模式下，该信号线对主控器发出中断请求。
保留	I	44	该输入信号无效，应被置高或通过主控器连接至 VCC。
- RESET	I	41	True IDE 模式下，通过主控器，该输入引脚低电平复位。
VCC	-	13,38	+5V，+3.3V 电源。
-VS1,-VS2	O	33,40	CF 卡工作电压控测信号。-VS1 接地，可使 CF 存储卡/CF+卡在 3.3V 下被读取，-VS2 保留。
-IORDY	O	42	在 True IDE 模式下，该输出信号可当作 IORDY 信号使用
-WE	I	36	在 True IDE 模式下，该输入信号无效，可通过主控器接 VCC。
-IOIS16	O	24	在 True IDE 模式，当设备为一个字数据传输周期时，该输出信号为低。

CF 卡和 IDE 硬盘设备的寄存器地址如表 6.34 所示，其读写时序如表 6.35 和图 6.95 所示。

表 6.34 设备寄存器地址

-CS1	-CS0	A02	A01	A00	-IORD=0	-IOWR=0	Note
1	0	0	0	0	RD 数据	WR 数据	8 位或 16 位
1	0	0	0	1	错误寄存器	特征	8 位
1	0	0	1	0	扇区计数	扇区计数	8 位
1	0	0	1	1	扇区号	扇区号	8 位
1	0	1	0	0	低柱面	低柱面	8 位
1	0	1	0	1	高柱面	高柱面	8 位

接上表

-CS1	-CS0	A02	A01	A00	-IORD=0	-IOWR=0	Note
1	0	1	1	0	选择卡/磁头	选择卡/磁头	8 位
1	0	1	1	1	状态	命令	8 位
0	1	1	1	0	Alt 状态	设备控制	8 位

在图 6.95 中波形的信号不一定表示高电平，而是表示所定义的电平信号有效，从而忽略实际引脚信号的高/低电平状态。如图 6.95 中的-IORD、-IOWR 及-IOCS16 等信号当其波形为高时，则表示其引脚电平信号有效，其实际电平为低电平。

表 6.35 寄存器读/写时序

	项	模式 0（ns）	模式 1（ns）	模式 2（ns）	模式 3（ns）	模式 4（ns）	注
t_0	周期时间（min）	600	383	240	180	120	1
t_1	地址有效时，-IORD/-IOWR 的调整时间（min）	70	50	30	30	25	
t_2	-IORD/-IOWR（min）	165	125	100	80	70	1
t_2	-IORD/-IOWR（min）寄存器（8 位）	290	290	290	80	70	1
t_{2i}	-IORD/-IOWR 唤醒时间（min）	-	-	-	70	25	1
t_3	-IOWR 数据调整时间（min）	60	45	30	30	20	
t_3	-IOWR 数据保持时间（min）	30	20	15	10	10	
t_5	-IORD 数据调整时间（min）	50	35	20	20	20	
t_6	-IORD 数据保持时间（min）	5	5	5	5	5	
t_{6Z}	-IORD 数据触发态	30	30	30	30	30	2
t_7	地址有效时，-IOCS16 的设定时间（最大）	90	50	40	N/a	N/a	4
t_8	地址有效时，-IOCS16 的释放时间（最大）	60	45	30	N/a	N/a	4
t_9	地址有效时，-IORD/-IOWR 的保持时间	20	15	10	10	10	
t_{RD}	读数据有效时,IORDY 的启动时间（min），如果 t_A 后，IORDY 初始化为低。	0	0	0	0	0	
t_A	IORDY 调整时间	35	35	35	35	35	3
t_B	IORDY 脉冲宽度（最大）	1250	1250	1250	1250	1250	
t_C	IORDY 设定到释放的时间（最大）	5	5	5	5	5	

注：-IOIS16 的最大负载为一个 50pF 的 LSTTL。时间级为 ns 级。-IORDY 高电平到-IORD 高电平的最小时间为 0ns，但是必须符合最小-IORD 宽度。

(1) t_0 为最小总周期时间，t_2 为最小指令启动时间，t_{2i} 为最小指令恢复时间及指令失效时间。实际周期时间等于实际指令活动时间加上实际指令停止时间。t_0,t_2,t_{2i} 应遵循时间的要求。最小总周期时间要求大于 t_2 加 t_{2i}。主控器可加长 t_2 或 t_{2i} 的时间长度，以保证 t_0 等于或大于设备驱动识别指令的返回值。CF 存储卡应用时，应可被老式主控器操作。

(2) 参数设定了从-IORD 低电平，到 CF 存储卡（触发态）不能驱动数据总线特定时间。

(3) 从-IORD 或-IOWR 启动到 IORDY 首次采样应有一段延时。如果 IORDY 静止，在 PIO 周期完成前，主控器将等待 IORDY 启动。在-IORD 或-IOWR 活动后的 t_A 时间段中，如果 CF 存储卡没有驱动 IORDY 为低，t_5 应被遵循，t_{RD} 无用。如果在-IORD 或-IOWR 启动后的 t_A 时间段，CF 存储卡驱动 IORDY 为低，t_{RD} 应被遵循，t_5 无用。

(4) t_7 及 t_8 仅可作用于模式 0,1 及 2。在其他模式,该信号无效。

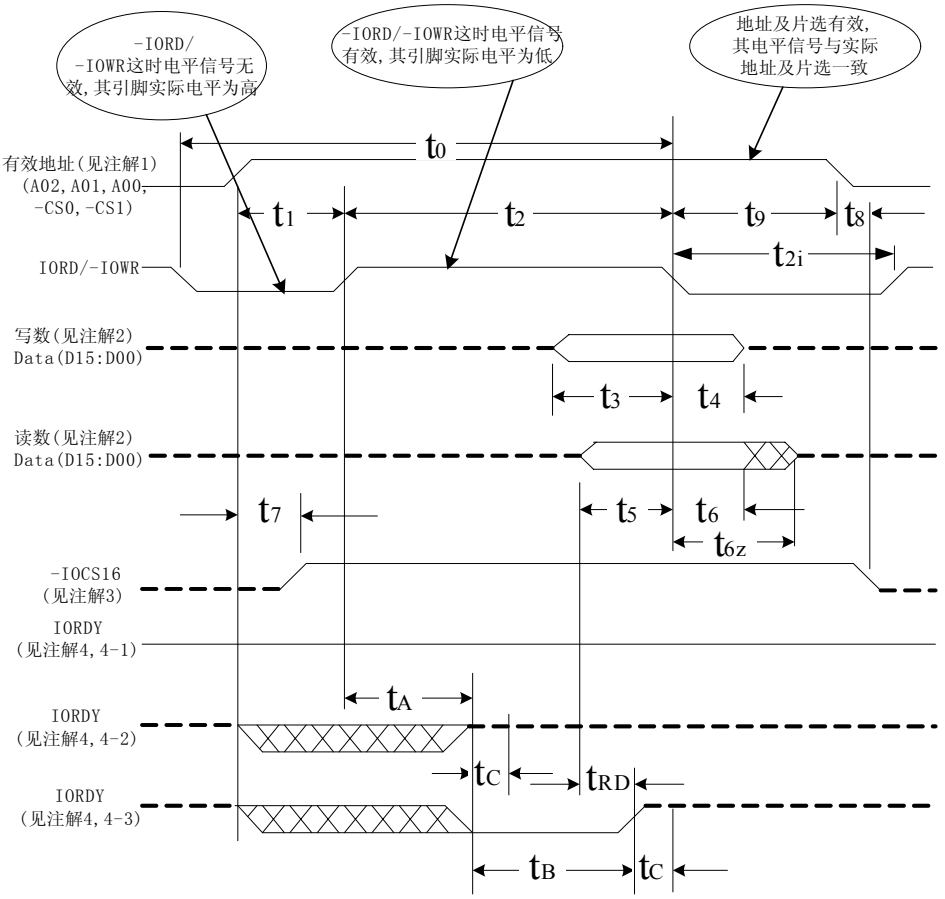


图 6.95 I/O 时序图

注解：

1. 设备地址由-CS0,-CS1 及 A[02:00]决定。
2. 数据由 D[15:00]（16 位）或 D[07:00]（8 位）。
3. -IOCS16 在 PIO 模式 0，1，2 中有效，其他模式忽略该信号。
4. 设备产生 IORDY 低电平以扩展 PIO 周期。-IORD 或-IOWR 被设置 t_A 时间后，主控器可判断周期是否被扩展。IORDY 描述有以下 3 种方式：

   (1) 设备从来不产生 IODRY 低电平：无等待

   (2) 设备在 t_A 前开始驱动 IORDY 为低，将使 IORDY 在 t_A 前被设定：无等待

   (3) 设备在 t_A 前开始驱动 IORDY 为低：有等待。在 IORDY 重新被设定后完成周期。为了在周期

   内产生等待信号及设定-IORD，在 IORDY 被设定前，t_{RD} 信号后设备将把读得的数据置于 D15-D00。

LPC2000 的 GPIO 引脚与 CF 卡及 IDE 硬盘的硬件接线图分别如图 6.96 和图 6.97 所示。

表 6.36 为 LPC2210 的 GPIO 引脚与 CF 卡及 IDE 硬盘引脚连接分配表，表中描述了各 GPIO 引脚与 CF 卡及 IDE 硬盘对应的控制信号线，根据表中的描述配置 LPC2000 相关的寄存器。

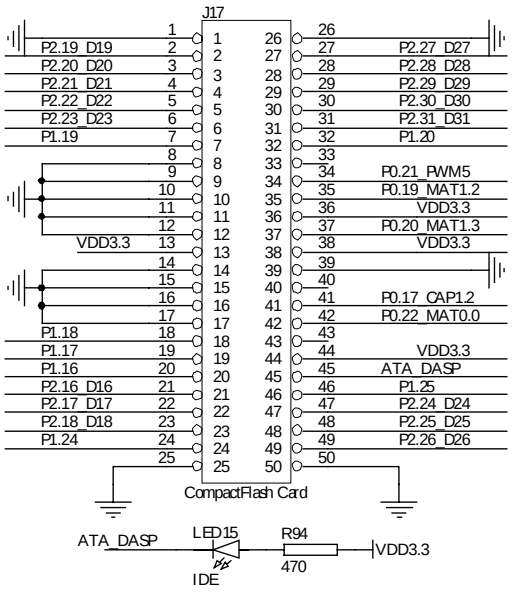


图 6.96 LPC2210 与 CF 卡接线图

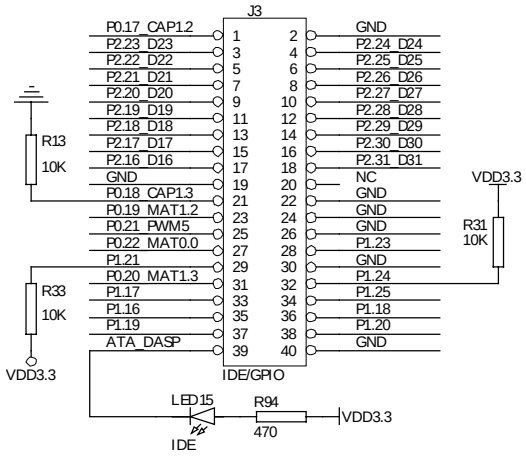


图 6.97 LPC2210 与 IDE 硬盘接线图

表 6.36 LPC2210 的 GPIO 引脚与 CF 卡及 IDE 硬盘连接引脚分配

LPC2210	CF 卡	IDE 硬盘	I/O	LPC2210	CF 卡	IDE 硬盘	I/O
*P0.17	-RESET	-RESET	O	*P1.17	A01	A01	O
*P2.16~P2.31	D00~D15	D00~D15	I 、O	*P1.16	A00	A00	O
P0.18		DMARQ	I	*P1.19	-CS0	-CS0	O
*P0.19	-IOWR	-DIOW	O	P1.23		CSEL	O
*P0.21	-IORD	-DIOR	O	P1.24	-IOCS16	-IOCS16	I
P0.22	IORDY	IORDY	I	P1.25	-PDIAG	-PDIAG	I
P1.21		-DMACK	I	*P1.18	A02	A02	O
P0.20	INTRQ	INTRQ	I	*P1.20	-CS1	-CS1	O

注：1.I/O 输入与输出是相对于 LPC2210 来说的，I 为 LPC2210 的输入，O 为输出。

2.表中“*”号的引脚，为使用到的引脚，其它引脚不需使用，但需要配置为适当的状态。

3.引脚描述见表 6.33。

CF 卡可以在 5V 或 3.3V 下工作，当 CF 工作电源为 5V 时 CF 卡的某些引脚要求输入的逻辑电平最小值为 4.0V，而 GPIO 的输出电平才 3.3V，所示只能使用 3.3V 给 CF 卡供电。

由于寄存器的地址是由 A00、A01、A02、-CS0 和-CS1 决定，将它们都分配在 P1 口是为了简化编程；而数据总线 D00-D15 使用 P2.16~P2.31 使用连续的 GPIO，也是为了也编程方便；其它的 I/O 引脚都没有特别的要求。

思考与练习

1．写出最小系统的定义，并画出最小系统原理框图。
2．写出至少 3 种 GPIO 的应用实例。
3．写出 NAND 和 NOR 型 FLASH 的异同点。
4．介绍 I^2C 和 SPI 总线的特点，并分别介绍几款基于这两种总线的芯片。

第7章 移植μC/OS-II到ARM7

7.1 μC/OS-II 简介

7.1.1 概述

μC/OS-II 读做“micro C O S 2”，意为“微控制器操作系统版本 2”。μC/OS-II 是著名的、源码公开的实时内核，可用于各类 8 位、16 位和 32 位单片机或 DSP。从μC/OS 算起，该内核已有 10 多年应用史，在诸多领域得到广泛应用。

μC/OS-II 是一个完整的、可移植、可固化、可剪裁的占先式实时多任务内核。μC/OS-II 是用 ANSI C 语言编写，包含一小部分汇编代码，使之可以供不同架构的微处理器使用。至今，从 8 位到 64 位，μC/OS-II 已在超过 40 种不同架构的微处理器上运行。

7.1.2 μC/OS-II 的特点

提供源代码：购买参考文献[5]可以获得μC/OS-II V2.52 版本的所有源代码，购买此书的其它版本可以获得相应版本的全部源代码。

可移植性（portable）：μC/OS-II 的源代码绝大部分是使用移植性很强的 ANSI C 写的，与微处理器硬件相关的部分是使用汇编语言写。汇编语言写的部分已经压缩到最低的限度，以使μC/OS-II 便于移植到其它微处理器上。目前，μC/OS-II 已经被移植到多种不同架构的微处理器上。

可固化(ROMmable)：只要具备合适的软硬件工具，就可以将μC/OS-II 嵌入到产品中成为产品的一部分。

可剪裁(scalable)：μC/OS-II 使用条件编译实现可剪裁，用户程序可以只编译自己需要的（μC/OS-II 的）功能，而不编译不要需要的功能，以减少μC/OS-II 对代码空间和数据空间的占用。

可剥夺(preemptive)：μC/OS-II 是完全可剥夺型的实时内核，μC/OS-II 总是运行就绪条件下优先级最高的任务。

多任务：μC/OS-II 可以管理 64 个任务，然而，μC/OS-II 的作者建议用户保留 8 个给μC/OS-II。这样，留给用户的应用程序最多可有 56 个任务。

可确定性：绝大多数μC/OS-II 的函数调用和服务的执行时间具有确定性，也就是说，用户总是能知道μC/OS-II 的函数调用与服务执行了多长时间。

任务栈：μC/OS-II 的每个任务都有自己单独的栈，使用μC/OS-II 的占空间校验函数，可确定每个任务到底需要多少栈空间。

系统服务：μC/OS-II 提供很多系统服务，例如信号量、互斥信号量、时间标志、消息邮箱、消息队列、块大小固定的内存的申请与释放及时间管理函数等。

中断管理：中断可以使正在执行的任务暂时挂起，如果优先级更高的任务被中断唤醒，则高优先级的任务在中断嵌套全部退出后立即执行，中断嵌套层数可达 255 层。

稳定性与可靠性：μC/OS-II 是基于μC/OS 的，μC/OS 自 1992 年以来已经有数百个商业应用。μC/OS-II 与μC/OS 的内核是一样的，只是提供了更多的功能。另外，2000 年 7 月，μC/OS-II 在一个航空项目中得到了美国联邦航空管理局对商用飞机的、符合 RTCA DO – 178B 标准的认证。这一结论表明，该操作系统的质量得到了认证，可以在任何应用中使用。

7.2 移植规划

7.2.1 编译器的选择

目前，针对 ARM 处理器核的 C 语言编译器有很多，如 SDT、ADS、IAR、TASKING 和 GCC 等。据了解，目前在国内最流行的是 SDT、ADS 和 GCC。SDT 和 ADS 均为 ARM 公司自己开发，ADS 为 SDT 的升级版，以后 ARM 公司不再支持 SDT，所以不会选择 SDT。GCC 虽然支持广泛，很多开发套件使用它作为编译器，与 ADS 比较其编译效率较低，这对充分发挥芯片性能很不利，所以最终使用 ADS 编译程序和调试。

7.2.2 任务模式的取舍

ARM7 处理器核具有用户、系统、管理、中止、未定义、中断和快中断七种模式，其中除用户模式外其它均为特权模式。关于 ARM7 处理器核的详细情况参考第 3 章 。由第 3 章 可知，管理、中止、未定义、中断和快中断与相应异常相联系，任务使用这些模式不太不适合。而系统模式除了是特权模式外，其它与用户模式一样，因而可选的给任务使用的模式只有用户模式和系统模式。为了尽量减少任务代码错误对整个程序的影响，缺省的任务模式定为用户模式，可选为系统模式，同时提供接口使任务可以在这两种模式间切换。

7.2.3 支持的指令集

带 T 变量的 ARM7 处理器核具有两个指令集（可以参考第 3 章 ）：标准 32 位 ARM 指令集和 16 位 Thumb 指令集，两种指令集有不同的应用范围。为了最大限度地支持芯片的特性，任务应当可以使用任意一个指令集并可以自由切换，而且不同的任务应当可以使用不同的指令集，这次移植的代码已经实现了这一点。

7.3 移植μC/OS-II

7.3.1 概述

要移植一个操作系统到一个特定的 CPU 体系结构上并不是一件很容易的事情，它对移植者有以下要求：

1. 对目标体系结构要有很深了解；
2. 对 OS 原理要有较深入的了解；
3. 对所使用的编译器要有较深入的了解；
4. 对需要移植的操作系统要有相当的了解；
5. 对具体使用的芯片也要一定的了解。

因此，在移植μC/OS-II 到 ARM7 时，我们必须先把上面几个方面的知识了解得比较透彻。这不单单是阅读资料就可以实现的，还需要不断地实践和实验。关于第 1 点的参考资料最权威的是参考文献[2]，ADS1.2 已经自带这个文档的电子版。关于第 2 点和第 4 点可以参考参考文献[5]。关于第 3 点需要参考 ADS 自带的编译器和连接器的手册，安装 ADS1.2 时都附带了它们的电子版。关于第 5 点，参考资料肯定为具体芯片的数据手册和使用手册了。

这里我们着重说明一下第 4 点，因为这一点的影响是全局性的的，决定移植代码的框架和功能。其它部分也很重要，它们在细节上影响代码，并最终影响代码的正确性、可靠性和健壮性。

μC/OS-II 中要移植的部分见表 7.1。根据μC/OS-II 的要求，移植μC/OS-II 到一个新的体系结构上需要提供 2 个或 3 个文件：OS_CPU.H（C 语言头文件）、OS_CPU_C.C（C 程序源文件）及 OS_CPU_A.ASM（汇编程序源文件），其中 OS_CPU_A.ASM 在某些情况下不需要，

但极其罕见。不需要 OS_CPU_A.ASM 的必须满足以下苛刻条件：

1. 可以直接使用 C 语言开关中断；
2. 可以直接使用 C 语言编写中断服务程序；
3. 可以直接使用 C 语言操作堆栈指针；
4. 可以直接使用 C 语言保存 CPU 的所有寄存器。

同时支持以上 4 点的 C 语言编译器几乎不存在，即使存在，移植代码往往也会使用部分汇编语言来提高移植代码的效率。由表 7.1 可以看出，移植μC/OS-II 需要在 OS_CPU.H 包含几个类型的定义和几个常数的定义；在 OS_CPU_C 和 OS_CPU_A.ASM 中包含几个函数的定义和时钟节拍中断服务程序的代码。实际上，还有一个 includes.h 文件需要关注，因为每一个应用都包含独特的 includes.h 文件。

表 7.1 μC/OS-II 需要移植的代码

移植内容	类型	所属文件	描述
BOOLEAN、INT8U、INT8S、INT16U、INT16S、INT32U、INT32S、FP32、FP64	数据类型	OS_CPU.H	与编译器无关的数据类型
OS_STK	数据类型	OS_CPU.H	堆栈的数据类型
OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL()	宏	OS_CPU.H	开关中断的代码
OS_STK_GROWTH	常量	OS_CPU.H	定义堆栈的增长方向
OS_TASK_SW	函数	OS_CPU.H	任务切换时执行的代码
OSTaskStkInit()	函数	OS_CPU_C.C	任务堆栈初始化函数
OSInitHookBegin()、OSInitHookEnd()、OSTaskCreateHook()、OSTaskDelHook()、OSTaskSwHook()、OSTaskStatHook()、OSTCBInitHook()、OSTimeTickHook()、OSTaskIdleHook()	函数	OS_CPU_C.C	μC/OS-II 在执行某些操作时调用的用户函数，一般为空
OSStartHighRdy()	函数	*OS_CPU_A.ASM	进入多任务环境时运行优先级最高的任务，
OSIntCtxSw()	函数	*OS_CPU_A.ASM	中断退出时的任务切换函数
OSTickISR()	中断服务程序	*OS_CPU_A.ASM	时钟节拍中断服务程序

*某些情况可在 OS_CPU_C.C 中实现。

而笔者的移植也包含 OS_CPU.H、OS_CPU_C.C 及 OS_CPU_A.S 三个文件。将 OS_CPU_A.ASM 更名为 OS_CPU_A.S 是遵照编译器的惯例。实际上，还有一个文件很重要，它就是 IRQ.S，它定义了一个汇编宏，它是μC/OS-II for ARM7 通用的中断服务程序的汇编与 C 函数接口代码。时钟节拍中断服务程序也没有移植，因为其与芯片和应用都强烈相关，需要用户自己编写，不过可以通过 IRQ.S 简化用户代码的编写。

7.3.2 关于头文件 includes.h 和 config.h

μC/OS-II 要求所有.C 文件的都要包含都文件 includes.h，这样使得用户项目中的每个.C 文件不用分别去考虑它实际上需要哪些头文件。使用 INCLUDES.H 的缺点是它可能会包含一些实际不相关的头文件，这意味着每个文件的编译时间可能会增加，但却增强了代码的可移植性。

在本移植中另外增加了一个头文件 config.h，我们要求所有用户程序必须包含 config.h，

在 config.h 中包含 includes.h 和特定的头文件和配置项。而μC/OS-II 的系统文件依然只是包含 includes.h，即μC/OS-II 的系统文件完全不必改动。所有的配置改变包括头文件的增减均在 config.h 中进行，而 includes.h 定下来后不必改动（μC/OS-II 的系统文件需要包含的东西是固定的）。这样，μC/OS-II 的系统文件需要编译的次数大大减少，编译时间随之减少。

7.3.3 编写 OS_CPU.H

1. 不依赖于编译的数据类型

μCOS-II 不使用 C 语言中的 short、int、long 等数据类型的定义，因为它们与处理器类型有关，隐含着不可移植性。代之以移植性强的整数数据类型，这样，既直观又可移植，不过这就成了必须移植的代码。根据 ADS 编译器的特性，这些代码如程序清单 7.1 所示。

程序清单 7.1 不依赖于编译器的数据类型

```
typedef    unsigned char    BOOLEAN;
typedef    unsigned char    INT8U;
typedef    signed char      INT8S;
typedef    unsigned short   INT16U;
typedef    signed short     INT16S;
typedef    unsigned int     INT32U;
typedef    signed int       INT32S;
typedef    float            FP32;
typedef    double           FP64;

typedef    INT32U           OS_STK;
```

2. 使用软中断 SWI 作底层接口

带 T 变量的 ARM7 处理器核具有两个指令集，用户任务还可以使用两种处理器模式：用户模式和系统模式，组合起来具有 4 种方式，各种方式对系统资源有不同的访问控制权限。为了使底层接口函数与处理器状态无关，同时在任务调用相应的函数不需要知道函数位置，本移植使用软中断指令 SWI 作为底层接口，使用不同的功能号区分不同的函数。软中断功能号分配如表 7.2,未列出的为保留功能。不过，用软中断作为操作系统的底层接口就需要在 C 语言中使用 SWI 指令。在 ADS 中，有一个关键字__swi，用它声明一个不存在的函数，则调用这个函数就在调用这个函数的地方插入一条 SWI 指令，并且可以指定功能号。同时，这个函数也可以有参数和返回值，其传递规则与一般函数一样。其代码见

程序清单 7.2。关键字__swi 的具体使用规则请参考 ADS 的编译器手册。

表 7.2 软中断功能

功能号	接口函数	简介
0x00	void OS_TASK_SW(void)	任务级任务切换函数
0x01	_OSStartHighRdy(void)	运行优先级最高的任务,由 OSStartHighRdy 产生
0x02	void OS_ENTER_CRITICAL(void)	关中断
0x03	Void OS_EXIT_CRITICAL(void)	开中断
0x80	Void ChangeToSYSMode(void)	任务切换到系统模式

接上表

功能号	接口函数	简介
0x81	Void ChangeToUSRMode(void)	任务切换到用户模式
0x82	Void TaskIsARM(INT8U prio)	任务代码是 ARM 代码
0x83	Void TaskIsTHUMB(INT8U prio)	任务代码是 THUMB 代码

程序清单 7.2 SWI 服务函数

```
__swi(0x00) void OS_TASK_SW(void);              /*  任务级任务切换函数          */
__swi(0x01) void _OSStartHighRdy(void);         /*  运行优先级最高的任务        */
__swi(0x02) void OS_ENTER_CRITICAL(void);       /*  关中断                      */
__swi(0x03) void OS_EXIT_CRITICAL(void);        /*  开中断                      */
__swi(0x80) void ChangeToSYSMode(void);         /*  任务切换到系统模式          */
__swi(0x81) void ChangeToUSRMode(void);         /*  任务切换到用户模式          */
__swi(0x82) void TaskIsARM(INT8U prio);         /*  任务代码是 ARM 代码         */
__swi(0x83) void TaskIsTHUMB(INT8U prio);       /*  任务代码是 THUMB 代码       */
```

3. OS_STK_GROWTH

μCOS-II 使用结构常量 OS_STK_GROWTH 中指定堆栈的生长方式：

置 OS_STK_GROWTH 为 0 表示堆栈从下往上长。

置 OS_STK_GROWTH 为 1 表示堆栈从上往下长。

虽然 ARM 处理器核对于两种方式均支持，但 ADS 的 C 语言编译器仅支持一种方式，即从上往下长，并且必须是满递减堆栈，所以 OS_STK_GROWTH 的值为 1，代码见程序清单 7.3。

程序清单 7.3 定义堆栈增长方向

```
#define OS_STK_GROWTH       1
```

7.3.4 编写 Os_cpu_c.c 文件

1. OSTaskStkInit()

在编写此函数之前，必须先确定任务的堆栈结构。而任务的堆栈结构是与 CPU 的体系结构、编译器有密切的关联。本移植的堆栈结构见图 7.1。根据图 7.1，很容易写出函数 OSTaskStkInit()的代码，具体代码见程序清单 7.4。

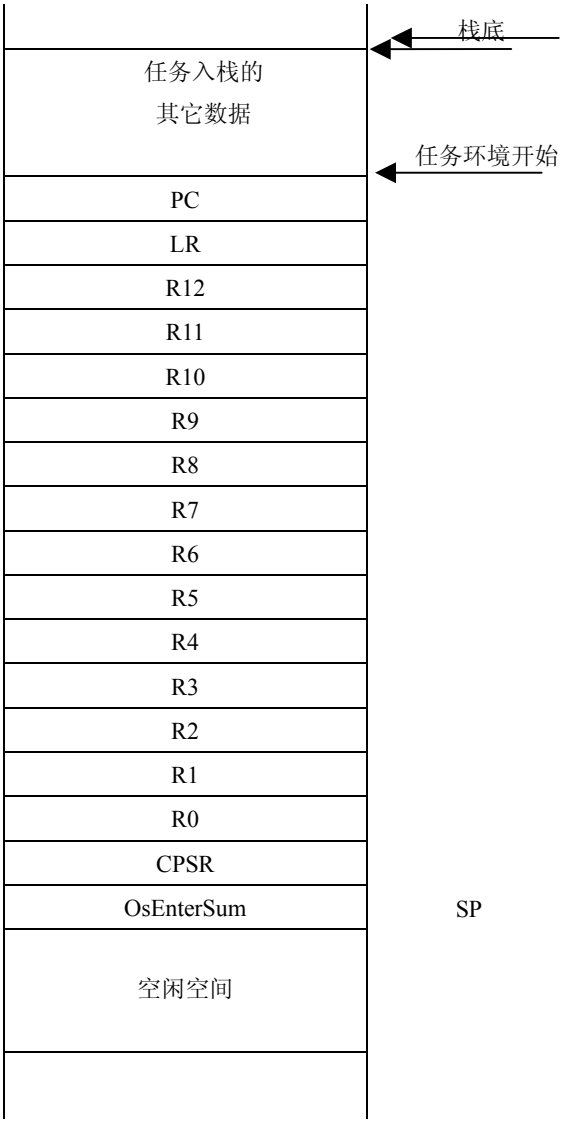


图 7.1 任务堆栈结构

程序清单 7.4 函数 OSTaskStkInt()代码

```
        OS_STK *OSTaskStkInit (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT16U opt)
{
    OS_STK *stk;

    opt     = opt;                          /* 'opt'  没有使用。作用是避免编译器警告      */
    stk     = ptos;                         /*  获取堆栈指针                              */
                                            /*  建立任务环境，ADS1.2 使用满递减堆栈       */
    *stk = (OS_STK) task;                   /*   pc   */
    *--stk = (OS_STK) task;                 /*   lr   */
    *--stk = 0;                             /*   r12  */
    *--stk = 0;                             /*   r11  */
    *--stk = 0;                             /*   r10  */
    *--stk = 0;                             /*   r9   */
```

```
    *--stk = 0;                                 /*  r8    */
    *--stk = 0;                                 /*  r7    */
    *--stk = 0;                                 /*  r6    */
    *--stk = 0;                                 /*  r5    */
    *--stk = 0;                                 /*  r4    */
    *--stk = 0;                                 /*  r3    */
    *--stk = 0;                                 /*  r2    */
    *--stk = 0;                                 /*  r1    */
    *--stk = (unsigned int) pdata;              /*  r0，第一个参数使用 R0 传递    */
    *--stk = (USER_USING_MODE|0x00);            /*  spsr，允许 IRQ, FIQ 中断    */
    *--stk = 0;                                 /*  关中断计数器 OsEnterSum;     */
    return (stk);
}
```

堆栈中有一个程序 OsEnterSum 比较特别，它不是 CPU 的寄存器，而是笔者定义的一个全局变量，主要是用它来保存关中断的次数，这样关中断和开中断就可以嵌套了。在调用 OS_ENTER_CRITICAL()时，它的值增加，同时关中断。在调用 OS_EXIT_CRITICAL()时，它的值减少，并且仅在其值为 0 时开中断。每个任务有独立的 OsEnterSum，在任务切换时保存和恢复各自的 OsEnterSum。这样，各个任务开关中断的状态可以不同，任务不必过分考虑关中断对别的任务的影响。

2. 软件中断异常 SWI 服务程序 C 语言部分

软件中断的 C 语言处理函数代码见程序清单 7.5，其中参数 SWI_Num 为功能号，而 Regs 为指向堆栈中保存寄存器的值的位置。程序的结构比较简单，使用一个 switch 语句把各个功能分隔开，各个功能相对独立。对比表 7.2 可知，软中断的 0、1 号功能并没有在这里实现。它在 0s_cpu_a.s 中实现，具体请参考 7.3.5 小节。

程序清单 7.5 软中断代码的 C 语言部分

```
        void SWI_Exception(int SWI_Num, int *Regs)
{
    OS_TCB     *ptcb;

    switch(SWI_Num)
    {
        case 0x02:
            __asm                                                   (1)
            {
                MRS       R0,SPSR                                   (2)
                ORR       R0,R0,#NoInt                              (3)
                MSR       SPSR_c,R0                                 (4)
            }
            OsEnterSum++;                                           (5)
            break;                                                  (6)
        case 0x03:
            if (--OsEnterSum == 0)                                  (7)
```

```
        {
            __asm                                                   (8)
            {
                MRS     R0,SPSR                                     (9)
                BIC     R0,R0,#NoInt                                (10)
                MSR     SPSR_c,R0                                   (11)
            }
        }
        break;                                                      (12)
    case 0x80:
        __asm                                                       (13)
        {
            MRS     R0,SPSR                                         (14)
            BIC     R0,R0,#0x1f                                     (15)
            ORR     R0,R0,#SYS32Mode                                (16)
            MSR     SPSR_c,R0                                       (17)
        }
        break;                                                      (18)
    case 0x81:
        __asm                                                       (19)
        {
            MRS     R0,SPSR                                         (20)
            BIC     R0,R0,#0x1f                                     (21)
            ORR     R0,R0,#USR32Mode                                (22)
            MSR     SPSR_c,R0                                       (23)
        }
        break;                                                      (24)
    case 0x82:
        if (Regs[0] <= OS_LOWEST_PRIO)                              (25)
        {
            ptcb = OSTCBPrioTbl[Regs[0]];                           (26)
            if (ptcb != NULL)                                       (27)
            {
                ptcb -> OSTCBStkPtr[1] &= ~(1 << 5);                (28)
            }
        }
        break;                                                      (29)
    case 0x83:
        if (Regs[0] <= OS_LOWEST_PRIO)                              (30)
        {
            ptcb = OSTCBPrioTbl[Regs[0]];                           (31)
            if (ptcb != NULL)                                       (32)
            {
                ptcb -> OSTCBStkPtr[1] |= (1 << 5);                 (33)
```

```
                    }
                }
                break;                                                    (34)
            default:
                break;                                                    (35)
        }
}
```

3. OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL()

μCOS-II 使用宏 OS_ENTER_CRITICAL()（程序清单 7.5(1)~(6)）和 OS_EXIT_CRITICAL()（程序清单 7.5(7)~(12)）分别关中断和开中断。关中断和开中断是为了保护临界段代码。这些代码与处理器有关，是需要移植的代码。在 ARM 处理器核中关中断和开中断时通过改变程序状态寄存器 CPSR 中的相应控制位实现。由于使用了软件中断，程序状态寄存器 CPSR 保存到程序状态保存寄存器 SPSR 中，软件中断退出时会将 SPSR 恢复到 CPSR 中，所以程序只要改变程序状态保存寄存器 SPSR 中的相应的控制位就可以了。改变这些位是使用嵌入汇编实现，代码很简单，不再说明。

4. OSStartHighRdy

μC/OS-II 的启动多任务环境的函数叫做 OSStart()，用户在调用 OSStart()之前，必须已经建立了一个或更多任务。OSStart()最终调用函数 OSStartHighRdy()运行多任务启动前优先级最高的任务，OSStartHighRdy()的代码见程序清单 7.6。

程序清单 7.6 OSStartHighRdy()代码

```
    void OSStartHighRdy(void)
{
    _OSStartHighRdy();
}
```

由 7.3.3 小节可知，这是调用软中断的 1 号功能。软中断的 1 号功能并没有在这里实现。它在 0s_cpu_a.s 中实现，具体请参考 7.3.3 小节。

5. 移植增加的特定函数

根据 ARM 核心的特点和移植的目标，为此增加了两个处理器模式转换函数（ChangeToSYSMode()、ChangeToUSRMode()）和两个任务初始指令集设置函数（TaskIsARM()、TaskIsTHUMB()）。它们都是通过软件中断指令 SWI 转换到系统模式，通过软件中断服务程序实现的（参考 7.3.3 小节）。

处理器模式转换函数 ChangeToSYSMode()和 ChangeToUSRMode()使用软件中断功能 0x80 和 0x81 实现，其中函数 ChangeToSYSMode()（程序清单 7.5(13)~(18)）把当前任务转换到系统模式，函数 ChangeToUSRMode()（程序清单 7.5(19)~(24)）把当前任务转换到用户模式，它们可以在任何情况下使用。它们改变程序状态保留寄存器 SPSR 的相应位段，而程序状态保留寄存器会在软件中断退出时复制到程序状态寄存器 CPSR，任务的处理器模式就改变了。

前面已经说明：任务可以使用 ARM 的两种指令集的任意一种运行，但是任务建立时默认的只是一种指令集。如果任务使用的第一条指令与默认的指令集不同，则程序运行错误。为了纠正这个错误，本移植增加两个函数 TaskIsARM()和 TaskIsTHUMB()用于改变任务建立时默认的指令集。函数 TaskIsARM()用于声明指定优先级的任务的第一条指令是 ARM 指令

集中的指令，而函数 TaskIsTHUMB()用于声明指定优先级的任务的第一条指令是 THUMB 指令集中的指令，它们都有唯一的参数：需要改变的任务的优先级，值得注意的是，这两个函数必须在相应的任务建立后但还没有运行时调用。这样，如果在低优先级的任务中创建高优先级的任务就十分危险了。此时，解决的方法有三种：

(1) 高优先级任务使用默认的指令集；

(2) 改变函数 OSTaskCreateHook()使任务默认不是处于就绪状态，建立任务后调用函数 OSTaskResume()来使任务进入就绪状态；

(3) 建立任务时禁止任务切换，调用函数 TaskIsARM()或 TaskIsTHUMB()后再允许任务切换。

函数 TaskIsARM()和 TaskIsTHUMB()使用软件中断功能 0x82 和 0x83 实现。两个功能代码极其相似，代码也比较简单。首先，程序判断传递的参数（任务的优先级）是否在允许的范围内（程序清单 7.5 (25)、(30)）。然后获取任务的任务控制块(tcb)的地址（程序清单 7.5 (26)、(31)），接着判断指针是否有效（程序清单 7.5(27)、(32)），有效则改变指定任务的堆栈中存储的 CPSR 的 T 位(程序清单 7.5(28)、(33))，至于为何这样写参考 7.3.5 小节和 ARM 相关文档，这两个函数仅在任务建立时使用。

6. …Hook()函数

μC/OS-II 有很多由用户编写的…Hook()函数，它在本移植中全为空函数，用户就可以按照μC/OS-II 的要求修改它。

7.3.5 编写 Os_cpu_a.s

1. 软件中断的汇编接口

软中断的汇编与 C 接口程序代码见程序清单 7. 7。

程序清单 7.7 软件中断代码的汇编部分

```
SoftwareInterrupt
        LDR     SP, StackSvc                    (1)
        STMFD   SP!, {R0-R3, R12, LR}           (2)
        MOV     R1, SP                          (3)
        MRS     R3, SPSR                        (4)
        TST     R3, #T_bit                      (5)
        LDRNEH  R0, [LR,#-2]                    (6)
        BICNE   R0, R0, #0xff00                 (7)
        LDREQ   R0, [LR,#-4]                    (8)
        BICEQ   R0, R0, #0xFF000000             (9)
        CMP     R0, #1                          (10)
        LDRLO   PC, =OSIntCtxSw                 (11)
        LDREQ   PC, =__OSStartHighRdy           (12)
        BL      SWI_Exception                   (13)
        LDMFD   SP!, {R0-R3, R12, PC}^          (14)
```

软中断的功能号包含在SWI指令中，程序通过读取该条指令的相应位段获得。由于ARM处理器核具有两个指令集，两个指令集的指令的长度不同，SWI 指令的功能号的位段也不同，所以程序先判断在进入软中断前处理器是在什么指令集状态（程序清单 7. 7(4)、(5)）。如果是 THUMB 指令集状态，则通过程序清单 7. 7(6)句读取指令中，通过程序清单 7. 7(7)

取得指令中的功能号。如果是 ARM 指令集状态，则通过程序清单 7. 7(9)句读取指令，通过程序清单 7. 7(10)取得指令中的功能号。

然后，程序用功能号与 1 比较（程序清单 7. 7(10)），当功能号无符号小于 1 时就是 0 了，就跳转到任务切换函数处（程序清单 7. 7(11)），也就是 OS_TASK_SW()。当功能号等于 1 时，就跳转到第一次任务切换处（程序清单 7. 7(12)）,也就是_OSStartHighRdy。这两个功能不在 C 语言中实现，原因一是因为它们需要明确的堆栈结构，这是 C 语言不能提供的；原因二是两个任务切换程序本身是使用汇编编写，且同在 os_cpu_a.s 这个文件中，使用汇编跳转十分方便。

其它功能就给软件中断的 C 语言处理函数处理，它有两个参数，第一个就是功能号，存于 R0 中，第二个是保存参数和返回值的指针，也就是堆栈中存储用户函数 R0~R3 的位置，实质就是当前堆栈指针的值，它存于 R1 中（程序清单 7. 7(3)）。

2. OS_TASK_SW()和 OSIntCtxSw()

OS_TASK_SW()是在μC/OS-II 从低优先级任务切换到最高优先级任务时被调用的，OS_TASK_SW()总是在任务级代码中被调用的。另一个函数 OSIntExit()被用来在 ISR 使得更高优先级任务处于就绪状态时，执行任务切换功能，它最终调用 OSIntCtxSw()执行任务切换。由 7.3.3 小节可知，OS_TASK_SW()是使用 SWI 软件中断的 0 号功能实现的。根据程序清单 7.7(11)知道，它是调用 OSIntCtxSw 实现的。由程序清单 7.7 可知，此时的堆栈结构如图 7.2 所示。同时，R3 保存着 SPSR。这样，如果中断调用 OSIntCtxSw()时需要相同的堆栈结构，R3 也要保存着 SPSR，这需要中断服务程序保证。OSIntCtxSw()的代码见程序清单 7.8。

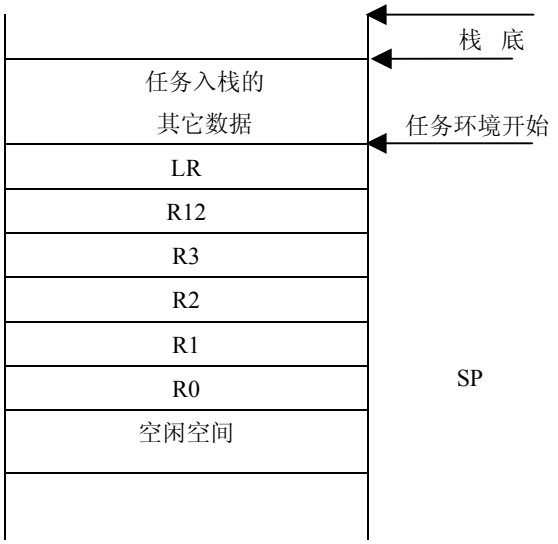


图 7.2 调用 OSIntCtxSw 时的堆栈结构

程序清单 7.8 OSIntCtxSw 代码

```
OSIntCtxSw
                                        ;下面为保存任务环境
    LDR     R2,[SP,#20]                 ;获取 PC                    (1)
    LDR     R12,[SP,#16]                ;获取 R12                   (2)
    MRS     R0,CPSR                                                 (3)

    MSR     CPSR_c,#(NoInt | SYS32Mode)                             (4)
    MOV     R1,LR                                                   (5)
```

```
        STMFD   SP!, {R1-R2}                    ;保存 LR,PC                    (6)
        STMFD   SP!,{R4-R12}                    ;保存 R4-R12                   (7)

        MSR     CPSR_c,R0                                                      (8)
        LDMFD   SP!,{R4-R7}                     ;获取 R0-R3                    (9)
        ADD     SP,SP,#8                        ;出栈 R12,PC                   (10)

        MSR     CPSR_c,#(NoInt | SYS32Mode)                                    (11)
        STMFD   SP!,{R4-R7}                     ;保存 R0-R3                    (12)

        LDR     R1,=OsEnterSum                  ;获取 OsEnterSum               (13)
        LDR     R2,[R1]                                                        (14)
        STMFD   SP!,{R2, R3}                    ;保存 CPSR,OsEnterSum          (15)
                                                ;保存当前任务堆栈指针到当前任务的 TCB
        LDR     R1,=OSTCBCur                                                   (16)
        LDR     R1,[R1]                                                        (17)
        STR     SP,[R1]                                                        (18)

        BL      OSTaskSwHook                    ;调用钩子函数                  (19)
                                                ;OSPrioCur <= OSPrioHighRdy
        LDR     R4,=OSPrioCur                                                  (20)
        LDR     R5,=OSPrioHighRdy                                              (21)
        LDRB    R6,[R5]                                                        (22)
        STRB    R6,[R4]                                                        (23)
                                                ;OSTCBCur <= OSTCBHighRdy
        LDR     R6,=OSTCBHighRdy                                               (24)
        LDR     R6,[R6]                                                        (25)
        LDR     R4,=OSTCBCur                                                   (26)
        STR     R6,[R4]                                                        (27)
OSIntCtxSw_1
                                                ;获取新任务堆栈指针
        LDR     R4,[R6]             ;17 寄存器 CPSR,OsEnterSum,R0-R12,LR,SP    (28)
        ADD     SP,R4,#68                                                      (29)
        LDR     LR,[SP,#-8]                                                    (30)
        MSR     CPSR_c,#(NoInt | SVC32Mode)     ;进入管理模式                  (31)
        MOV     SP,R4                           ;设置堆栈指针                  (32)

        LDMFD   SP!, {R4, R5}                   ;CPSR,OsEnterSum               (33)
        LDR     R3,=OsEnterSum                  ;恢复新任务的 OsEnterSum       (34)
        STR     R4,[R3]                                                        (35)
        MSR     SPSR_cxsf,R5                    ;恢复 CPSR                     (36)
        LDMFD   SP!,{R0-R12,LR,PC }^            ;运行新任务                    (37)
```

因为处于不同的模式，图 7.1 和图 7.2 处于不同的堆栈空间中。程序清单 7.8(1)～(15)

就是按照图 7.1 保存任务状态，其它部分都是按照μC/OS-II 的要求编写的，唯一要注意的是使用系统模式返回任务。这是因为任务可能处于系统模式，也可能处于用户模式；可能使用 ARM 指令集，也可能使用 Thumb 指令集，只有用系统模式的 SPSR 保存任务的 CPSR，然后使用程序清单 7.8(37)返回任务才能正确的切换 CPU 的模式和状态。

3. OSStartHighRdy()

μC/OS-II 的启动多任务环境的函数叫做 OSStart()，用户在调用 OSStart()之前，必须已经建立了一个或更多任务(参考参考文献[5]的第 3.13 节)。OSStart()最终调用函数 OSStartHighRdy()运行多任务启动前优先级最高的任务，由 7.3.3、7.3.4 小节和程序清单 7.7(12)可知，它最终调用__OSStartHighRdy 实现的。__OSStartHighRdy 的代码见程序清单 7.9，__OSStartHighRdy 的编写很简单，按照μC/OS-II 的要求编写即可。

程序清单 7.9 __OSStartHighRdy 代码

```
__OSStartHighRdy
    MSR     CPSR_c, #(NoInt | SYS32Mode)                    (1)
    LDR     R4,=OSRunning                                   (2)
    MOV     R5,#1                                           (3)
    STRB    R5,[R4]                                         (4)
    BL      OSTaskSwHook                                    (5)
    LDR     R6,=OSTCBHighRdy                                (6)
    LDR     R6,[R6]                                         (7)
    B       OSIntCtxSw_1                                    (8)
```

7.3.6 关于中断及时钟节拍

在本移植中，IRQ 是受μC/OS-II 管理的中断，对于 FIQ 不做处理。由于各种 ARM 芯片的中断系统不一样，各个用户的目标板也不一样，中断及时钟节拍时需要进一步移植的代码。不过，笔者编写了一个汇编宏，它是μC/OS-II for ARM7 通用的中断服务程序的汇编与 C 函数接口代码，见程序清单 7.10。

程序清单 7.10 IRQ 异常处理代码的汇编部分

```
    MACRO
$IRQ_Label HANDLER $IRQ_Exception_Function
        EXPORT  $IRQ_Label                          ; 输出的标号                    (1)
        IMPORT  $IRQ_Exception_Function             ; 引用的外部标号                (2)
$IRQ_Label
        SUB     LR, LR, #4                          ; 计算返回地址                  (3)
        STMFD   SP!, {R0-R3, R12, LR}               ; 保存任务环境                  (4)
        MRS     R3, SPSR                            ; 保存状态                      (5)
        STMFD   SP, {R3, SP, LR}^       ; 保存用户状态的 R3,SP,LR,注意不能回写  (6)
                                                    ; OSIntNesting++
        LDR     R2,  =OSIntNesting                                                  (7)
        LDRB    R1, [R2]                                                            (8)
        ADD     R1, R1, #1                                                          (9)
        STRB    R1, [R2]                                                            (10)
```

```
        SUB     SP, SP, #4*3                                              (11)

        MSR     CPSR_c, #(NoInt | SYS32Mode)        ; 切换到系统模式      (12)
        CMP     R1, #1                                                    (13)
        LDREQ   SP, =StackUsr                                             (14)

        BL      $IRQ_Exception_Function             ; 调用 c 语言的中断处理程序 (15)

        LDR     R2, =OsEnterSum          ; OsEnterSum,使 OSIntExit 退出时中断关闭 (16)
        MOV     R1, #1                                                    (17)
        STR     R1, [R2]                                                  (18)

        BL      OSIntExit                                                 (19)

        LDR     R2, =OsEnterSum     ; 因为中断服务程序要退出，所以 OsEnterSum=0 (20)
        MOV     R1, #0                                                    (21)
        STR     R1, [R2]                                                  (22)

        MSR     CPSR_c, #(NoInt | IRQ32Mode)        ; 切换回 irq 模式     (23)
        LDMFD   SP, {R3, SP, LR}^                   ; 恢复用户状态的 R3,SP,LR,注意不能回写 (24)

        LDR     R0, =OSTCBHighRdy                                         (25)
        LDR     R0, [R0]                                                  (26)
        LDR     R1, =OSTCBCur                                             (27)
        LDR     R1, [R1]                                                  (28)
        CMP     R0, R1                                                    (29)

        ADD     SP, SP, #4*3                                              (30)
        MSR     SPSR_cxsf, R3                                             (31)
        LDMEQFD SP!, {R0-R3, R12, PC}^              ; 不进行任务切换      (32)
        LDR     PC, =OSIntCtxSw                     ; 进行任务切换        (33)
    MEND
```

程序清单 7.10 是根据μC/OS-II 对中断服务程序的要求、ARM7 体系结构特点、ADS 编译器特点和 7.3.4 小节的第 2 点（OSIntCtxSw()）的要求编写的，适合所有基于 ARM7 核的芯片。有了这段代码，中断服务程序的 C 语言部分很简单，示意代码见程序清单 7.11。

程序清单 7.11 中断服务程序 C 语言部分示意代码

```
        void ISR(void)
{
    OS_ENTER_CRITICAL()或直接给变量 OsEnterSum 赋一;                        (1)
    清除中断源;                                                           (2)
    通知中断控制器中断结束：                                              (3)
    开中断： OS_EXIT_CRITICAL();                                          (4)
```

```
    用户处理程序;                                                    (5)
}
```

程序清单 7.11(1)、(4)在这里是必须的，因为中断发生时肯定是允许中断的，所以“关中断计数器” OsEnterSum 肯定是 0。这样如果用户在程序清单 7.11(3) 之前调用μC/OS-II 的系统服务函数（它们一般会成对调用 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()）就很可能打开中断，而在系统没有清中断源和/或程序清单 7.11(3)没有执行的情况下就开中断就会造成芯片的中断系统工作异常而使程序工作异常。如果用户程序没有这种情况，则可以不要程序清单 7.11(1)句，而把程序清单 7.11(4)句用普通方法打开中断。如果用户程序在程序清单 7.11(3)后马上调用系统服务函数，甚至可以不要程序清单 7.11(4)句。因为进入中断处理器核自动关中断，所以可以使用直接给变量 OsEnterSum 赋一的方法避免系统函数打开中断。

7.4 移植代码应用到 LPC2000

7.3 节介绍的代码仅为μC/OS-II 在 ARM7 上移植的通用代码，当在具体的项目中应用这些代码时，还需要做一些工作，这是由以下原因造成：

1．因为 ARM 公司的政策，各种基于 ARM7 处理器核的芯片的存储系统不同、片内外设不同，具有的中断源也不同，甚至连中断系统都不一样。这样，对于 ADS1.2 这个集成开发环境来说，需要自己或厂家编写部分启动代码。而中断源不一样，则时钟节拍中断服务程序就不可能统一。至于中断系统都不一样，就更造成了中断服务程序的编写方法不可能完全一样。这样，相关的代码就需要读着自行编写。

2．因为各个项目对资源的要求不一样，所以尽管使用同一系列的芯片，其时钟节拍中断也可能使用不同的中断实现。

尽管如此，把移植代码应用到具体的项目中还是很容易的，本节以 PHILIPS 半导体公司的 LPC2200 系列芯片为例，介绍如何应用这些代码。

7.4.1 编写或获取启动代码

关于启动代码的编写请参考第 5 章 。

7.4.2 挂接 SWI 软件中断

将软中断异常处理程序挂接到内核是通过修改启动代码实现，见程序清单 7.12 的(3)和(11)。

程序清单 7.12 异常向量表

```
Reset
        LDR     PC,ResetAddr                                    (1)
        LDR     PC,UndefinedAddr                                (2)
        LDR     PC,SWI_Addr                                     (3)
        LDR     PC,PrefetchAddr                                 (4)
        LDR     PC,DataAbortAddr                                (5)
        DCD     0xb9205f80                                      (6)
        LDR     PC,[PC, #-0xff0]                                (7)
        LDR     PC,FIQ_Addr                                     (8)
```

```
ResetAddr       DCD     ResetInit               (9)
UndefinedAddr   DCD     Undefined               (10)
SWI_Addr        DCD     SoftwareInterrupt       (11)
PrefetchAddr    DCD     PrefetchAbort           (12)
DataAbortAddr   DCD     DataAbort               (13)
nouse           DCD     0                       (14)
IRQ_Addr        DCD     IRQ_Handler             (15)
FIQ_Addr        DCD     FIQ_Handler             (16)
```

7.4.3 中断及时钟节拍中断

编写中断服务程序代码比较简单，按照 7.3.6 小节编写 C 语言处理函数即可。关键在于把程序与芯片的相关中断源挂接，使芯片在产生相应的中断后会调用相应的处理程序。这需要做两方面事情：

1. 增加汇编接口的支持

方法是在文件中 IRQ.S 适当位置添加如程序清单 7.13 所示代码，其中 Xxx 替换为自己需要的字符串。这样，汇编接口就完成了。

程序清单 7.13 中断的汇编接口代码

```
Xxx_Handler     HANDLER Xxx_Exception
```

2. 初始化向量中断控制器

示意代码见程序清单 7.14。其中 X 为分配给中断的优先级，Y 为中断的通道号。

程序清单 7.14 中断初始化代码

```
中断外设初始化;
VICVectAddrX = (uint32) Xxx_Handler ;
VICVectCntlX = (0x20 | Y);
VICIntEnable = 1 << Y;
```

至于时钟节拍中断服务程序的编写除了在程序清单 7.11 的“用户处理程序”中必须调用函数 OSTimeTick()外，没有其它不同，这里不再介绍。

7.4.4 编写应用程序

移植μC/OS-II 是为了在自己的系统使用μC/OS-II。要在自己的系统中使用μC/OS-II 编写自己的应用程序就必须遵守μC/OS-II 的编程规范。程序清单 7.15～程序清单 7.17 是一个很简单的例子（只给出代码的主要部分）。这个例子在运行时，每按一下按键 KEY1 则蜂鸣器鸣叫两声。

程序清单 7.15 为应用程序的 main()函数以及相关的代码，这应当是 Main()函数的最简单的代码了。程序清单 7.15(1)是本章要求包含的头文件，具体信息参考 7.3.2 小节。程序清单 7.15(2)、(3)为分配任务的堆栈空间，由于每个任务的堆栈是独立的，这部分代码不能省略。在使用μC/OS-II 提供的任何功能之前，必须调用函数 OSInit()（程序清单 7.15(4)），它完成μC/OS-II 的初始化并建立空闲任务。在开始多任务之前，必须建立至少一个用户任务（不包括μC/OS-II 的空闲任务），这是通过调用函数 OSTaskCreate()（程序清单 7.15(5)）或函数 OSTaskCreateExt()实现。关于函数 OSTaskCreate()与函数 OSTaskCreateExt()的参数及区别请

参考参考文献[5]。最后函数 main()调用函数 OSStart()将控制权交给μC/OS-II 内核（程序清单 7.15(6)），main()函数也就结束了。

程序清单 7.15 应用程序主函数

```
#include "config.h"                                                              (1)
OS_STK          TaskStartStk[TASK_STK_SIZE];                                     (2)
OS_STK          TaskStk[TASK_STK_SIZE];                                          (3)

        int main (void)
{
    OSInit();                                                                    (4)
    OSTaskCreate(Task1, (void *)0, &TaskStartStk[TASK_STK_SIZE - 1], 0);         (5)
    OSStart();                                                                   (6)
    return 0;                                                                    (7)
}
```

程序清单 7.16 为程序清单 7.15(5)建立的任务的代码。程序清单 7.16 是使用一般的任务程序的框架，绝大多数任务都需要按照这个框架编写代码。另一种不常用的任务代码框架的例子见程序清单 7.17。μC/OS-II 规定任务是不允许返回的，程序清单 7.16 所示任务是通过死循环实现（程序清单 7.16(1)），而程序清单 7.17 所示任务通过调用函数 OSTaskDel()（程序清单 7.17(4)）实现的。用户任务代码必须调用μC/OS-II 提供的服务，如程序清单 7.16 的(2)、(3)、(4)、（5）和程序清单 7.17 的(1)、(2)、(3)、(4)。任务的代码比较简单，读一下注释就可以明白，这里不再说明。

程序清单 7.16 第一个任务——键盘扫描

```
        void   Task1(void *pdata)
{
    pdata = pdata;                                  /*  避免编译警告  */
    TargetInit();                                   /*  目标板初始化  */
    for (;;)                                                                     (1)
    {
        OSTimeDly(OS_TICKS_PER_SEC / 50);           /*  延时 20 毫秒  */         (2)
        if (GetKey() != KEY1)                       /* GetKey 用于获取键盘当前状态  */
        {
            continue;                               /*  不是 KEY1 不理会  */
        }
        OSTimeDly(OS_TICKS_PER_SEC / 50);           /*  延时 20 毫秒，用于去抖  */  (3)
        if (GetKey() != KEY1)                       /*  还是 KEY1 才正确   */
        {
            continue;
        }
        OSTaskCreate(Task2, (void *)0, &TaskStk[TASK_STK_SIZE - 1], 10);         (4)
        while (GetKey() != 0)                                   /*  等待松开按键  */
        {
```

```
            OSTimeDly(OS_TICKS_PER_SEC / 50);                    /* 延时 20 毫秒 */          (5)
        }
    }
}
```

程序清单 7.17 第二个任务——蜂鸣器鸣叫

```
    void  Task2(void *pdata)
{
    pdata = pdata;                          /* 避免编译警告 */
    BeeMoo();                               /* 使蜂鸣器鸣叫 */
    OSTimeDly(OS_TICKS_PER_SEC / 8);        /* 延时 */                          (1)
    BeeNoMoo();                             /* 使蜂鸣器停止鸣叫 */
    OSTimeDly(OS_TICKS_PER_SEC / 4);        /* 延时 */                          (2)
    BeeMoo();                               /* 使蜂鸣器鸣叫 */
    OSTimeDly(OS_TICKS_PER_SEC / 8);        /* 延时 */                          (3)
    BeeNoMoo();                             /* 使蜂鸣器停止鸣叫 */
    OSTaskDel(OS_PRIO_SELF);                /* 删除自己 */                      (4)
}
```

7.5 本章小结

本章介绍了μC/OS-II 在基于 ARM7 处理器核的芯片上的移植过程，并以 PHILIPS 的 LPC2200 系列 ARM 芯片为例介绍了如何使用它们。

由于移植涉及的内容较多，且内容分散，移植工作有一定的难度，对于初学者不做要求，但应该能够把这些移植好的代码应用到具体的实验或项目中。读者可以通过实验加深对本章的理解。

由于本书篇幅的限制，本章内容书写得比较简单。如果读者需要更深入的了解相关知识，请参考相关的技术文献。

思考与练习

1．μC/OS-II 有哪些特点？
2．移植μC/OS-II 需要具备哪些知识？
3．移植μC/OS-II 需要编写哪些文件？
4．移植μC/OS-II 到 ARM7 为何使用 SWI 软件中断异常接口？
5．写出 ADS 中关键字__swi 的具体用法。
6．移植代码为何要增加 ChangeToSYSMode()、ChangeToUSRMode()、TaskIsARM()和 TaskIsTHUMB()这 4 个函数？它们如何使用？
7．在 LPC2200 上编写一个简单的基于μC/OS-II 的程序。

第8章 嵌入式系统开发平台

8.1 如何建立嵌入式系统开发平台

8.1.1 使用平台开发是大势所趋

1. 平台的概念

平台这个词的使用有点泛滥，如建立交流平台、管理平台、服务平台等等。要准确的定义平台不是件容易的事。下面我们分析一下嵌入式系统开发平台的一般结构。

图 8.1 为使用嵌入式系统开发平台开发的产品的一般结构。由图 8.1 我们很容易看出平台有以下特性：

- **层次性**

层次性首先表现在平台本身具有层次，高层的平台是以低层平台为基础搭建起来的。如电路板这个硬件平台是由（多个）元器件平台为基础的，而应用程序接口这个平台是建立在硬件平台之上的。

层次性还表现在每个层次的平台的内部还有层次性。如硬件平台可能由很过电路板构成，而每个电路板就是一个子平台。而每个电路板又可能有不同的部分（存储器系统、IO 系统等等），每部分又是一个子平台。而应用程序接口（API）更是分层次的：板级支持包、驱动程序、OS 处于较低的层次，软件模块处于较高的层次。甚至软件模块之间也有层次，如 I^2C 的模块就在 I^2C 的 EEPROM 的读写模块之下。IC 的内部其实也是分层次的。

越高层次的平台越容易使用，对使用者的要求也就越低。

- **服务性**

平台是为应用服务的，没有应用的平台是无意义的，没有存在的必要。对于元器件来说，线路板就是它的应用。而 API 是硬件平台的应用，应用程序又是 API 的应用。

如果深入分析，平台还有以下特性：

- **规范性**

当一个平台建立起来后，其对应用的接口就具有一定的规范。例如，一个 IC 一经生产，其外围电路的设计就要受其规范的制约。同样，一个软件模块一旦开发完毕，其接口函数及使用方法就确定了，应用程序就需要按照规范使用。

- **通用性**

平台从来不是给一个应用使用的。如果一个平台只能给一个应用使用，作为平台来说，其设计是失败的（尽管其作为最终应用可能是成功的）。这是平台的服务性决定的。如果其只能服务一个应用，其与应用有何区别？

- **技术密集**

平台特别是商业平台通常由相关领域的专家建立。这是因为往往需要适应不同的应用的要求，所以其需要的技术含量是很高的。

2. 使用平台开发的优势

本章所论述的平台均指嵌入式系统的应用程序接口(API)这个层次的平台，而不是其它层次的平台。这是由当前的形式决定的。

在过去和将来相当长的一段时间内，嵌入式系统的应用开发由相关应用领域专家实现。应用的需求使嵌入式系统现在变得越来越复杂，IC 产业的发展使硬件简单化，软件逐渐占系统的较大部分。而以前孤立工作的系统已经联系起来。这样的系统需要方方面面的知识，

不再局限于个别领域，编写软件不是一两个人可以完成的。有鉴于此，不书所述的平台定义为应用程序接口（API）这个嵌入式软件平台，其它层次不重点介绍。

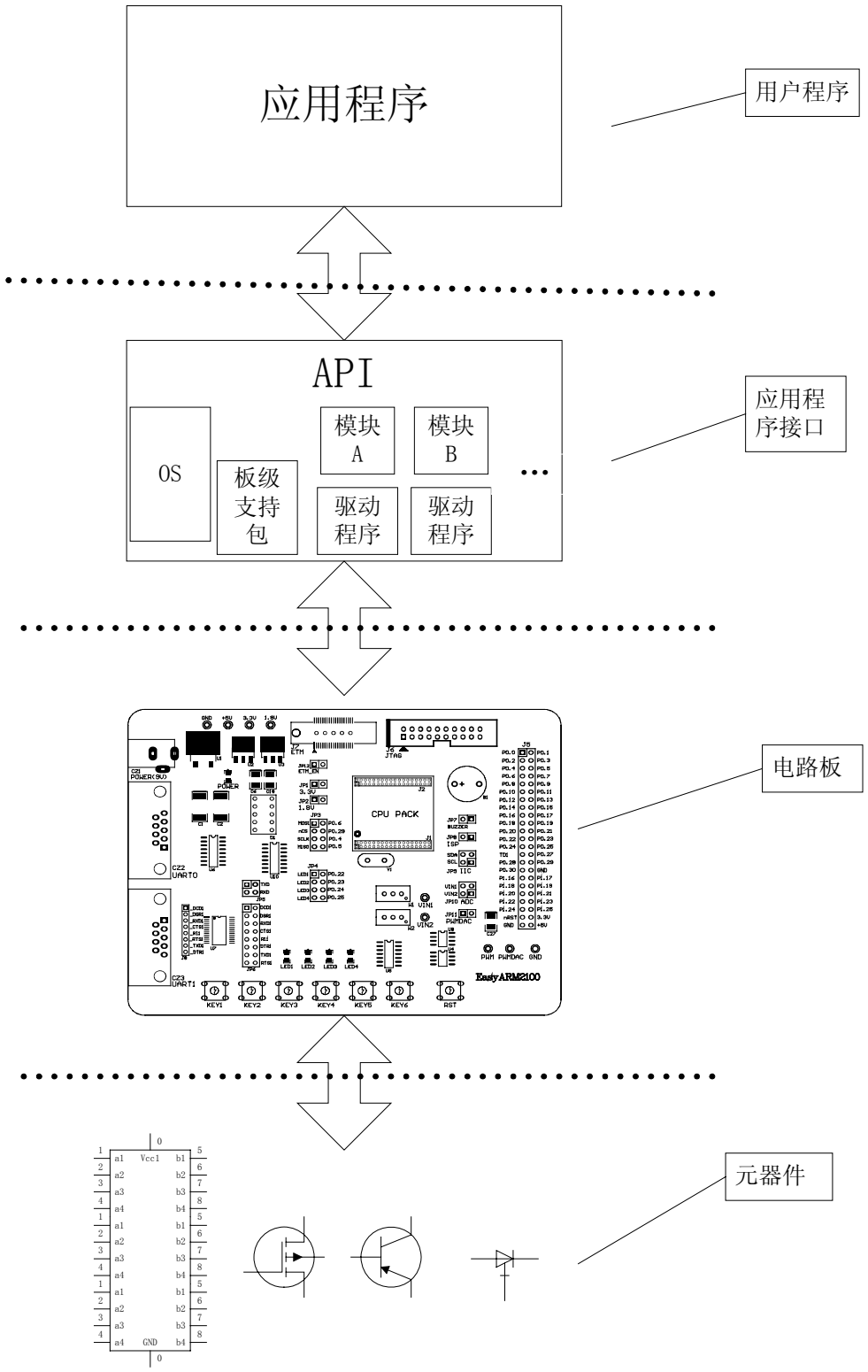


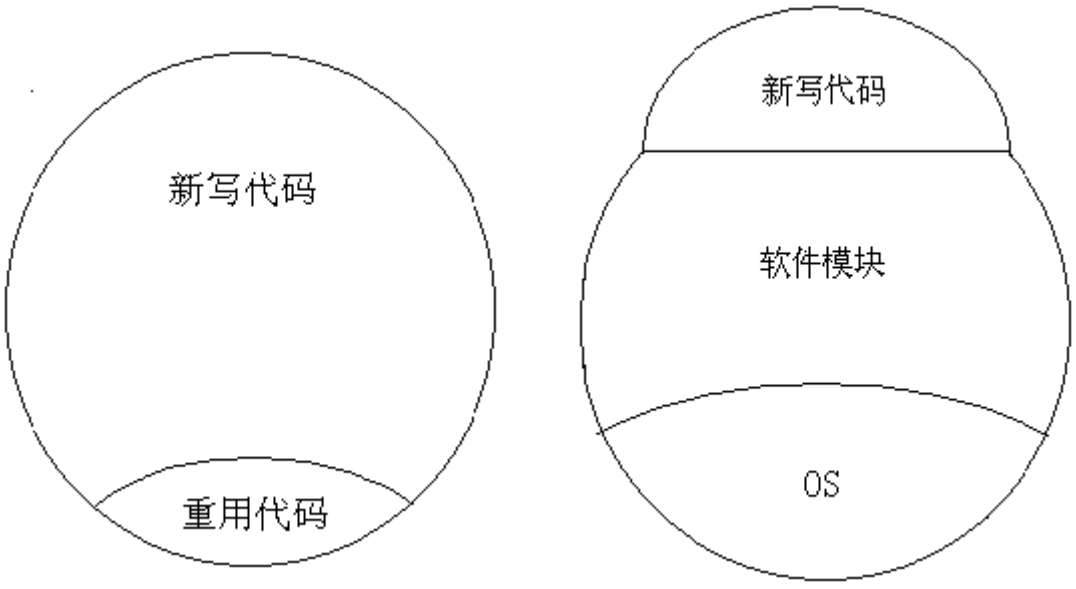
图 8.1 平台的例子

任何一个存在的事物必定有其存在的理由，应用程序接口这个嵌入式软件开发平台的提

出和应用也是因为其能够解决当前及今后的嵌入式开发过程中出现的一些矛盾。以下是使用嵌入式软件开发平台（API）开发的优点：

- **缩短开发时间**

图 8.2 为不使用和使用嵌入式软件开发平台开发产品时代码的构成。图 8.2 假设两者都是优化得比较好的情况，如果大家都优化得不太好，可能。由图 8.2 可以看出，尽管传统方式代码总量少一些，但可重用代码很少，大量代码需要编写。而使用平台开发，虽然代码量大一些，但大部分代码为复用代码（OS、软件模块），需要编写的新代码仅限于这个项目的特殊部分，它通常不多。因为仅需要编写少量新代码，所以代码的编写和调试工作量大大减少，进而缩短开发时间。



传统方式开发　　使用 API 开发

图 8.2　不同方式开发代码的比较

- **降低对开发人员的要求**

由于大部分与本项目无关代码无需重新编写，开发人员无需关心这部分代码涉及的专业知识。而这些专业知识往往不是项目相关的行业知识，而且知识往往覆盖很广，很难被少数几个工程师掌握。开发人员无需关心这部分知识，自然降低了对开发人员的要求。而且，有了编程规范，就降低了软件的逻辑关系，经验不足的工程师经过简单的培训就可以工作了。这又降低了对开发人员的要求。

- **降低开发成本**

缩短了开发时间、降低了开发人员的要求，则开发成本自然得到降低。

- **增加稳定性和可靠性**

OS 和软件模块都是相关领域的专家编写，代码大多经过多次各种场合的检验，其代码的稳定性和可靠性显然比重新编写的代码具有更高的稳定性和可靠性。

- **提高产品性能**

使然使用平台开发其性能比实际能够达到的性能低一些，但 OS 和软件模块都是相关领域的专家编写，其代码往往经过较好的优化，其代码效率肯定比门外汉临时编写的代码高。则造成的结果就是：大多数情况下，使用平台开发产品的性能更好。

- **降低软件对硬件变化的敏感性**

因为开发平台有相应的编程规范，应用程序不直接操作硬件。这样，硬件变化时，只需要修改 OS 和软件模块适应新的硬件环境即可，应用程序无需变换。而 OS 和软件模块都是相关领域的专家编写，在程序编写时就考虑了其硬件的适应性，硬件改变时，只需编写很少且很简单的代码即可，甚至无需编写代码，仅需要简单的配置一下。而且，一些软件模块是可以互相替代的，如模拟 I^2C 模块和硬件 I^2C 模块，它们分别支持不同的硬件，它们使开发平台在更大的硬件范围内兼容。这样，硬件的变化对项目的影响降低了许多。这也间接的降低了开发成本和运营成本，降低了对开发人员的要求。

3. 不适合的场合

虽然使用平台开发有很多优点，但其也不是万能的。有一些场合并不适合使用平台开发，这些场合主要有以下几种：

- **极大批量的产品**

这种产品往往要求硬件成本和生产成本最低化。而使用平台开发不可避免（针对极限情况）带来资源占用的增加，反映到硬件上就是成本的增加。有时这种增加是不可接受的。这种情况不适合用平台开发。不过这样会造成开发成本的上升（由于产量大可以忽略），开发时间加长（有时不能忍受）。其实，这种产品可以首先使用平台开发以缩短开发时间，争取上市时机，打击竞争对手。然后在这基础上优化软硬件，使硬件成本最低化。

- **没有合适的平台但产品需要尽快上市时**

开发平台是需要时间的，甚至比当前的项目的时间更长。但如果后续有类似的产品，则还是建议开发者逐步建立起自己的开发平台，这会对以后所有的项目带来好处。

- **需要极度榨取硬件性能的场合**

除了极大批量的产品需要极度榨取硬件性能外，估计只有科学研究和一些一件一件制造的高价值产品（如巨型机）需要这样做了，在普通的企业中应该不容易见到。

- **其它的一些场合**

8.1.2 建立开发平台的方法

1. 选择操作系统

操作系统（OS）是一个基础的软件平台。因此，建立软件开发平台（自己的 API）首先需要选择一个合适的操作系统。操作系统的选择与应用相关的，但对于嵌入式系统一般需要选择嵌入式操作系统。关于各种嵌入式操作系统的比较请参考相关资料。对于一般的嵌入式操作系统笔者推荐μC/OS-II。关于μC/OS-II 的介绍请参考 7.1 节。

2. 制订 API 规范及应用程序编写规范

有了操作系统，就可以制订 API 规范和程序编写规范了。这些规范一部分已经被操作系统规定好了，另一部分需要自己制订。笔者建议这些规范尽量靠近与已存在的事实标准，以减少学习时间。例如，文件系统的操作的 API 就尽量与 C 语言的标准靠拢，而 TCP/IP 协议的 API 就尽量与 BSD 标准的 Socket 接口靠拢。其它没有事实标准也尽量与类似的事实标准靠拢。例如，所有输入操作的 API 具有 Read 字样等等。

3. 获取成熟的软件模块并修改适合制订的规范

开发人员可以通过各种途径获取成熟的软件模块，例如，芯片供应商和硬件模块供应商都会提供一些软件模块。还可以获取一些商业和自由的软件模块。这些软件模块不一定适合自己的规范，这就需要对这些软件模块进行进一步的修改或包装以适应自己的规范。

4. 编写自己的软件模块

通过外部途径不一定能过获取所有必须的软件模块，此时就需要自己编写软件模块了。关于如何编写自己的软件模块请参考 8.1.3 小节。

8.1.3 编写自己的软件模块

因为开发平台基于操作系统，所以很可能多个任务和/或同时访问同一个 API 函数。这就需要 API 函数必须是可重入的。本节主要介绍如何使这些函数可重入，当然也包括其它一些内容。

1. 使用任务

有一些设备需要 CPU 周期为其服务，典型的是扫描显示和扫描键盘。可以给它们分配相应的任务，与用户任务一起调度。这样，就可以使用任何任务间通信的方法实现。

2. 禁止然后允许中断

一些设备需要的访问时间很短，如全局变量。它们可以通过禁止中断——访问设备——允许中断的方式编写软件模块。

3. 使用信号量

一些设备需要的访问时间比较长，使用方法 2 可能造成关中断时间过长。此时可以使用信号量来编写软件模块。具体过程为申请信号量——访问设备——发送信号量。

4. 使用数据队列

有一些设备具有自己的中断，典型的是串口输出。可以利用消息队列将用户任务需要的服务通过消息队列排队、缓冲起来，利用中断功能依次服务。这样，一般不需要考虑重入这个头疼的问题。

5. 禁止然后允许任务调度

当中断服务程序不可能访问这个资源时，可以使用本方法避免实现函数重入。但笔者不推荐用户使用此方法。

6. 使用一个任务作为模块服务器

当一个软件模块十分复杂，使用上述方法难以保证 API 函数可重入性时，或软件模块需要在单任务环境和多任务环境都能够使用时，可以依照单任务的方式编写，然后编写一个任务作为服务器，有这个服务器直接操作这个软件模块，而其它的任务通过消息队列等方式与之通讯，把需要的操作提交给服务器，而服务器通过消息油箱等方式返回操作结果。

7. 复合方法

有一些驱动程序比较复杂（例如通讯协议），可以结合两种或两种以上的方法实现。

编写软件模块由一个要**注意**的地方是：必须把与硬件相关的部分和与硬件无关的部分分开。这可以通过把与硬件相关的部分放在驱动程序中实现。这样，硬件变化时只需要重新编写驱动程序即可。驱动程序也可以分层实现。也可以通过配置实现，不过这有时不太容易实现。当然可以合并使用。

8.2 数据队列

8.2.1 简介

数据队列一般用于数据缓存，一般用于平衡速率不同的两个部件，使快速部件无需等待慢速部件。数据队列一般是先入先出（FIFO）的，但本数据队列可以配置为后入先出。本数据队列是可配置可裁减的摸块，并且不依赖于操作系统，可以在前后台系统中使用。数据队列使用的空间由用户分配且由这个空间的地址唯一识别一个数据队列。

8.2.2 API 函数集

表 8.1 QueueCreate 函数

函数名称	QueueCreate	所属文件	Queue.c
函数原型	uint8 QueueCreate(void *Buf, uint32 SizeOfBuf, uint8 (* ReadEmpty)(), uint8 (* WriteFull)())		
功能描述	建立数据队列		
编译开关	无		
函数参数	Buf ：为队列分配的存储空间地址（用户分配） SizeOfBuf ：为队列分配的存储空间大小（字节） ReadEmpty：为队列读空时处理程序 WriteFull ：为队列写满时处理程序		
函数返回值	NOT_OK ：参数错误 QUEUE_OK：建立队列成功		
调用模块	无		
特殊说明和注意点	使用数据队列前必须创建数据队列，数据队列使用的空间由用户分配		

范例	if (QueueCreate((void *)QueueBuf, sizeof(QueueBuf), NULL, NULL) == NOT_OK)
	{
	/* 错误处理程序 */
	}		

表 8.2 QueueRead 函数

函数名称	QueueRead	所属文件	Queue.c
函数原型	uint8 QueueRead(QUEUE_DATA_TYPE *Ret, void *Buf)		
功能描述	获取队列中的数据		
编译开关	无		
函数参数	Ret ：存储返回的消息的地址 Buf ：为队列分配的存储空间地址（用户分配）		
函数返回值	NOT_OK ：参数错误 QUEUE_OK ：读取:成功 QUEUE_EMPTY：队列空（无数据）		
调用模块	无		
特殊说明和注意点	使用数据队列前必须创建数据队列，数据队列使用的空间由用户分配		

范例	if (QueueRead(&Ch, (void *)QueueBuf) == QUEUE_OK)
	{
	/* 处理数据 */
	}		

表 8.3 QueueWrite 函数

函数名称	QueueWrite	所属文件	Queue.c
函数原型	uint8 QueueWrite(void *Buf, QUEUE_DATA_TYPE Data)		
功能描述	FIFO 方式发送数据		
编译开关	EN_QUEUE_WRITE		

函数参数	Buf ：为队列分配的存储空间地址（用户分配） Data：发送的数据
函数返回值	NOT_OK ：参数错误 QUEUE_OK：读取成功 OS_Q_FULL：队列满空（存不下数据）
调用模块	无
特殊说明和注意点	使用数据队列前必须创建数据队列，数据队列使用的空间由用户分配

范例	if (QueueWrite((void *)QueueBuf, Data) == OS_Q_FULL)
	{
	/* 满处理程序 */
	}

表 8.4 QueueWriteFront 函数

函数名称	QueueWriteFront	所属文件	Queue.c
函数原型	uint8 QueueWriteFront(void *Buf, QUEUE_DATA_TYPE Data)		
功能描述	LIFO 方式发送数据		
编译开关	EN_QUEUE_WRITE_FRONT		
函数参数	Buf ：为队列分配的存储空间地址（用户分配） Data：发送的数据		
函数返回值	NOT_OK ：参数错误 QUEUE_OK：读取:成功 OS_Q_FULL：队列满空（存不下数据）		
调用模块	无		
特殊说明和注意点	使用数据队列前必须创建数据队列，数据队列使用的空间由用户分配		

范例	if (QueueWriteFront((void *)QueueBuf, Data) == OS_Q_FULL)
	{
	/* 满处理程序 */
	}		

表 8.5 QueueFlush 函数

函数名称	QueueFlush	所属文件	Queue.c
函数原型	void QueueFlush(void *Buf)		
功能描述	清空队列		
编译开关	EN_QUEUE_FLUSH		
函数参数	Buf ：为队列分配的存储空间地址（用户分配）		
函数返回值	无		
调用模块	无		
特殊说明和注意点	使用数据队列前必须创建数据队列，数据队列使用的空间由用户分配		

表 8.6 QueueNData 函数

函数名称	QueueNData	所属文件	Queue.c
函数原型	uint16 QueueNData(void *Buf)		
功能描述	取得队列中数据数		
编译开关	EN_QUEUE_NDATA		
函数参数	Buf ：为队列分配的存储空间地址（用户分配）		
函数返回值	队列中存储的消息数目		
调用模块	无		
特殊说明和注意点	使用数据队列前必须创建数据队列，数据队列使用的空间由用户分配		

范例	If (QueueNData ((void *)QueueBuf) > 0)
	{
	QueueRead(&Ch, (void *)QueueBuf);
	}		

表 8.7 QueueSize 函数

函数名称	QueueSize	所属文件	Queue.c
函数原型	uint16 QueueSize(void *Buf)		
功能描述	取得队列中总共可以存储的数据数目		
编译开关	EN_QUEUE_SIZE		
函数参数	Buf ：为队列分配的存储空间地址（用户分配）		
函数返回值	队列中总共可以存储的数据数目		
调用模块	无		
特殊说明和注意点	使用数据队列前必须创建数据队列，数据队列使用的空间由用户分配		

范例	If (QueueSize ((void *)QueueBuf) - QueueNData ((void *)QueueBuf) > 0)
	{
	QueueWrite (&Ch, (void *)QueueBuf);
	}		

8.3 串口驱动

8.3.1 简介

在实际应用中，嵌入式系统往往不是作为一个独立的控制单元而存在，它还要和其它控制单元进行通信。这些控制单元可以是另一个嵌入式系统，也可以是 PC 机。如果双方均遵循同样的通讯协议，就可以互相通讯了。

8.3.2 API 函数集

表 8.8 UART0Init 函数

函数名称	UART0Init	所属文件	Uart0.c
函数原型	Uint8 UART0Init(uint32 bps)		

功能描述	初始化 UART0
函数参数	Bps：波特率
函数返回值	TRUE :成功 FALSE:失败
调用模块	QueueCreate,OSSemCreate
特殊说明 和注意点	只需要在系统初始化时调用一次。它不包含向量中断控制器部分的配置，这部分需要自己编写，且向量中断控制器必须允许 UART0 中断。

范例	if (UART0Init (115200) == FALSE)
	{
	/* 错误处理程序 */
	}

表 8.9 UART0Putch 函数

函数名称	UART0Putch	所属文件	Uart0.c
函数原型	Void UART0Putch(uint8 Data)		
功能描述	发送一个字节数据		
函数参数	Data：发送的数据		
函数返回值	无		
调用模块	QueueWrite,QueueRead		
特殊说明 和注意点	必须先初始化 UART0 及向量中断控制器		

表 8.10 UART0Write 函数

函数名称	UART0Write	所属文件	Uart0.c
函数原型	Void UART0Write(uint8 *Data, uint16 NByte)		
功能描述	发送多个字节数据		
函数参数	Data:发送数据存储位置 NByte:发送数据个数		
函数返回值	无		
调用模块	UART0Putch		
特殊说明 和注意点	必须先初始化 UART0 及向量中断控制器		

表 8.11 UART0Getch 函数

函数名称	UART0Getch	所属文件	Uart0.c
函数原型	uint8 UART0Getch(void)		
功能描述	接收一个字节数据		
函数参数	无		
函数返回值	接收到的数据		
调用模块	OSSemPend		
特殊说明 和注意点	必须先初始化 UART0 及向量中断控制器		

范例	while (UART0Getch() != 0xff);

8.4 MODEM 接口模块

8.4.1 简介

在实际应用中，嵌入式系统往往不是作为一个独立的控制单元而存在，它还要和其它控制单元进行通信。而这种通讯可能不局限于一个小的地理范围，如跨城市通讯甚至跨州通讯。此时，如果需要布置专门的通讯线路则成本太高，利用现成的通讯网络成为唯一的选择。而电话网络是目前分布最广的廉价通讯网络。不过，要利用电话网络需要一种接口设备，这就是 MODEM。

8.4.2 MODEM 的状态

NOT_INIT_MODEM： 不能初始化 MODEM
NOT_FIND_MODEM： 没有找到 MODEM
MODEM_CLOSE： 连接关闭
MODEM_RING： 发现振铃信号
MODEM_CONNECT： 已经与对方连接上

8.4.3 API 函数集

表 8.12 ModemInit 函数

函数名称	ModemInit	所属文件	modem.c
函数原型	uint8 ModemInit(uint32 bps)		
功能描述	初始化 Modem		
函数参数	与 MODEM 通讯的波特率		
函数返回值	MODEM 状态		
特殊说明和注意点	只需要调用一次		

表 8.13 GetModemState 函数

函数名称	GetModemState	所属文件	modem.c
函数原型	uint8 GetModemState(void)		
功能描述	获取 MODEM 状态		
函数参数	波特率		
函数返回值	MODEM 状态		
特殊说明和注意点	必须先初始化 MODEM		

表 8.14 ModemWrite 函数

函数名称	ModemWrite	所属文件	modem.c
函数原型	uint8 ModemWrite(char *Data, uint16 NByte)		
功能描述	通过 MODEM 发送多个字节数据		
函数参数	Data:发送数据存储位置		

	NByte:发送数据个数
函数返回值	MODEM 状态
特殊说明 和注意点	必须先初始化 MODEM

表 8.15 ModemGetch 函数

函数名称	ModemGetch	所属文件	modem.c
函数原型	uint8 ModemGetch(void)		
功能描述	从 MODEM 获取一个字节数据		
函数参数	无		
函数返回值	获得的数据		
特殊说明 和注意点	必须先初始化 MODEM 没有获取数据此函数不会返回		

表 8.16 ModemDialUp 函数

函数名称	ModemDialUp	所属文件	modem.c
函数原型	uint8 ModemDialUp(char Number[])		
功能描述	通过 modem 拨号		
函数参数	电话号码字符串		
函数返回值	MODEM 状态		
特殊说明 和注意点	必须先初始化 MODEM 拨号时间可能要很长时间		

表 8.17 ModemDialDown 函数

函数名称	ModemDialDown	所属文件	modem.c
函数原型	uint8 ModemDialDown(void)		
功能描述	挂断 MODEM		
函数参数	无		
函数返回值	MODEM 状态		
特殊说明 和注意点	必须先初始化 MODEM		

8.5 I^2C 总线模块

8.5.1 简介

I^2C 总线 10 多年前由 Philips 公司推出，是近年来在微电子通信控制领域广泛采用的一种新型总线标准。它是同步通信的一种特殊形式，具有接口线少，控制方式简化，器件封装形式小，通信速率较高等优点。在主从通信中，可以有多个 I^2C 总线器件同时接到 I^2C 总线上，通过地址来识别通信对象。

I^2C 总线通过 2 根线——串行数据(SDA)和串行时钟(SCL)线——连接到总线上的任何一个器件，每个器件都应有一个唯一的地址，而且都可以作为一个发送器或接收器。此外，器件在执行数据传输时也可以被看作是主机或从机。

8.5.2 API 函数集

表 8.18 I2cInit 函数

函数名称	I2cInit	所属文件	Uart0.c
函数原型	uint8 I2cInit(uint32 FI2c)		
功能描述	初始化 I^2C（主模式）		
函数参数	FI2c: I^2C 总线频率		
函数返回值	TRUE :成功 FALSE:失败		
调用模块	OSSemCreate		
特殊说明和注意点	只需要在系统初始化时调用一次。它不包含向量中断控制器部分的配置，这部分需要自己编写，且向量中断控制器必须允许 I^2C 中断。		

范例	if (I2cInit(100000) == FALSE)
	{
	/* 错误处理程序 */
	}		

表 8.19 I2cWrite 函数

函数名称	I2cWrite	所属文件	Uart0.c
函数原型	uint16 I2cWrite(uint8 Addr, uint8 *Data, int16 NByte)		
功能描述	向 I^2C 从器件写数据		
函数参数	Addr:从机地址 Data:指向将要写的数据的指针 NByte:写的数据数目		
函数返回值	发送的数据字节数		
调用模块	OSSemPend,__I2cWrite,OSSemPost		
特殊说明和注意点	必须先初始化 I^2C 及向量中断控制器		

范例	if (I2cWrite(CAT24WC02, DataBuf, 11) < 11)
	{
	/* 错误处理 */
	}		

表 8.20 I2cRead 函数

函数名称	I2cRead	所属文件	Uart0.c
函数原型	int16 I2cRead(uint8 Addr, uint8 *Ret, uint8 *Eaddr, int16 EaddrNByte, int16 ReadNbyte)		
功能描述	从 I^2C 从器件读数据		
函数参数	Addr:从机地址 Ret:指向返回数据存储位置的指针 Eaddr:扩展地址存储位置 EaddrNByte:扩展地址字节数，0 为无 ReadNbyte:将要读取的字节数目		

函数返回值	无
调用模块	OSSemPend,__I2cWrite,OSMboxPend,OSSemPost
特殊说明 和注意点	必须先初始化 I^2C 及向量中断控制器

范例	If (I2cRead(CAT24WC02, DataBuf, DataBuf, 1, 10) < 11)
	{
	/* 错误处理 */
	}

8.6 SPI 总线模块

8.6.1 简介

串行外围设备接口 SPI（serial peripheral interface）总线技术是 Motorola 公司推出的一种同步串行接口。SPI 总线是一种三线同步总线，因其硬件功能很强，所以，与 SPI 有关的软件就相当简单，使 CPU 有更多的时间处理其他事务。

SPI 是一个全双工的串行接口。它设计成可以处理在一个给定总线上多个互连的主机和从机。在一定数据传输过程中，接口上只能有一个主机和一个从机能够通信。在一次数据传输中，主机总是向从机发送一个字节数据，而从机也总是向主机发送一个字节数据。

8.6.2 API 函数集

表 8.21 SPIInit 函数

函数名称	SPIInit	所属文件	SPI.c
函数原型	uint8 SPIInit(uint8 Fdiv)		
功能描述	初始化 SPI 总线为主模式		
函数参数	Fdiv：用于设定总线频率（总线频率=Fpclk/Fdiv）		
函数返回值	TRUE :成功 FALSE:失败		
调用模块	OSMboxCreate，OSSemCreate		
特殊说明 和注意点	只需要在系统初始化时调用一次。它不包含向量中断控制器部分的配置，这部分需要自己编写，且向量中断控制器必须允许 SPI 中断。		

范例	if (SPIInit (0x30) == FALSE)
	{
	/* 错误处理程序 */
	}		

表 8.22 GetSPIFlag 函数

函数名称	GetSPIFlag	所属文件	SPI.c
函数原型	uint8 GetSPIFlag(void)		
功能描述	获取 SPI 状态		
函数参数	无		
函数返回值	0 :空闲 1 :忙		

调用模块	无
特殊说明 和注意点	无

范例	while (GetSPIFlag() == 0))
	{
	OSTimeDly(2);
	}

表 8.23 SPIStart 函数

函数名称	SPIStart	所属文件	SPI.c
函数原型	void SPIStart(void)		
功能描述	开始访问 SPI		
函数参数	无		
函数返回值	TRUE :成功 FALSE:失败		
调用模块	OSSemPend		
特殊说明 和注意点	在使用 SPI 总线之前必须调用此函数		
范例	SPIStart(); 允许从机; 多次 SPIRW(yy); 禁止从机; SPIEnd();		

表 8.24 SPIRW 函数

函数名称	SPIRW	所属文件	SPI.c
函数原型	uint8 SPIRW(uint8 *Rt, uint8 Data)		
功能描述	将数据通过 SPI 总线发送出去并从 SPI 总线接收一个数据		
函数参数	Rt ：函数通过这个指针返回接收到的数据 Data：发送的数据		
函数返回值	TRUE :成功 FALSE:失败		
调用模块	OSMboxPend,GetOSPrioCur		
特殊说明 和注意点	必须在调用函数 SPIStart()后调用，否则函数没有任何效果		
范例	SPIStart(); 允许从机; 多次 SPIRW(yy); 禁止从机; SPIEnd();		

表 8.25 SPIEnd 函数

函数名称	SPIEnd	所属文件	SPI.c
函数原型	uint8 SPIEnd(void)		
功能描述	访问 SPI 结束		
编译开关	无		
函数返回值	TRUE :成功 FALSE:失败		
调用模块	OSSemPost,GetOSPrioCur		
特殊说明和注意点	在使用 SPI 总线完毕后必须调用此函数，否则别的任务无法使用 SPI 总线。		
范例	SPIStart(); 允许从机; 多次 SPIRW(yy); 禁止从机; SPIEnd();		

8.7 其它软件模块

8.2～**错误！未找到引用源。**节介绍的软件模块都是比较简单和接近底层的软件模块，广州周立功单片机发展有限公司和其它一些公司还有一些比较大型的软件模块，比如 ZLG/FS、ZLG/D12、ZLG/CF、ZLG/IP、ZLG/GUI、MiniType^{TM} 字库及输入法等。限于本书篇幅，在这里不能详细介绍它们，感兴趣的读者可以购买广州周立功单片机发展有限公司推出 EasyARM2200 开发学习套件，其中附带 ZLG/FS、ZLG/D12、ZLG/CF、ZLG/IP、ZLG/GUI 的源代码及它们的一份使用许可证，同时有 MiniType^{TM} 字库的简单实验程序。

思考与练习

1．请描述平台的特称。
2．使用平台开发有什么优缺点？
3．建立嵌入式软件开发平台的有哪些步骤步骤？
4．编写自己软件模块有了些方法？

参 考 文 献

1. ARM 公司．The ARM-THUMB Procedure Call Standard ．ARM 公司，2000 年 10 月 24 日．
2. ARM 公司．ARM Architecture Reference Manual．ARM 公司，2000 年 6 月 27 日．
3. 周立功等．ARM 位控制器基础与实战．北京航空航天大学出版社，2003 年 11 月．
4. ARM 公司．ARM PrimeCell™ Vectored Interrupt Controller(PL190) ．ARM 公司．
5. Jean. Labrosse 著，邵贝贝等译．嵌入式实时操作系统 μC/OS-II(第 2 版)．北京航空航天大学出版社，2003 年 5 月．
6. 马忠梅，马广云等．ARM 嵌入式处理器结构与应用基础．北京航空航天大学出版社，2002 年 2 月．
7. 田泽，于敦山，盛世敏 译．ARM SoC 体系统结构．北京航空航天大学出版社，2002 年 10 月．
8. PHILIPS 公司．LPC2114/2124/2212/2214 User Manual．PHILIPS 公司，2004 年 5 月．
9. PHILIPS 公司．LPC2200 User Manual．PHILIPS 公司，2004 年 5 月．
10. 吴明晖等编著．基于 ARM 的嵌入式系统开发与应用．人民邮电出版社，2004 年 6 月．