Broadview FARSIGHT

高等院校嵌入式人才培养规划教材

ARM处理器开发详解

基于ARM Cortex-A8处理器的开发设计

清华远见嵌入式学院 刘洪涛 邹南 编著

本书特色：

电子工业出版社

高等院校嵌入式人才培养规划教材

ARM处理器开发详解

基于ARM Cortex-A8处理器的开发设计

华清远见嵌入式学院 刘洪涛 邹南 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

作为一种 32 位高性能、低成本的嵌入式 RISC 微处理器，ARM 目前已经成为应用最广泛的嵌入式处理器。目前 Cortex-A 系列处理器已经占据了大部分中高端产品市场。

本书在全面介绍 Cortex-A8 处理器的体系结构、编程模型、指令系统及开发环境的同时，以基于 Cortex-A8 的应用处理器——S5PC100 为核心，详细介绍了系统的设计及相关接口技术。接口技术涵盖了 I/O、中断、串口、存储器、PWM、A/D、DMA、IIC、SPI、Camera、LCD 等，并提供了大量的实验例程。

本书可以作为高等院校电子、通信、自动化、计算机等专业的 ARM 体系结构、接口技术课程的教材，也可作为嵌入式开发人员的参考书。

图书在版编目（CIP）数据

ARM 处理器开发详解：基于 ARM Cortex-A8 处理器的开发设计 / 刘洪涛，邹南编著.—北京：电子工业出版社，2012.9
ISBN 978-7-121-17714-9

Ⅰ. ①A… Ⅱ. ①刘… ②邹… Ⅲ. ①微处理器－系统设计 Ⅳ. ①TP332

中国版本图书馆 CIP 数据核字（2012）第 168402 号

策划编辑：胡辛征
责任编辑：许　艳
特约编辑：赵树刚
印　　刷：
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：18　字数：460 千字
印　　次：2012 年 9 月第 1 次印刷
印　　数：4000 册　　定价：49.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

随着消费群体对产品要求的日益提高，嵌入式技术在机械器具制造业、电子产品制造业、信息通信业、信息服务业等领域得到了大显身手的机会，并被越来越广泛地应用。ARM 作为一种 32 位的高性能、低成本的嵌入式 RISC 微处理器，已得到最广泛的应用。目前，Cortex-A 系列处理器已经占据了嵌入式处理器大部分的中高端产品市场，尤其是在移动设备市场上，几乎占据了绝对垄断的地位。

伴随着 Android 系统的发展，ARM 也越来越被大家所了解和接受，企业对 ARM 技术人才的需求也越来越大。各高校也已经认识到了这一点，并设置了相关课程。但建立一套完整的嵌入式教学课程，是一项非常复杂的工作，尤其是如何和企业需求相结合，更是高校所需要面临的重大问题。目前市场上的嵌入式开发相关书籍大多是针对研发人员编写的，并不太适合高校教学使用。北京华清远见科技信息有限公司长期以来致力于嵌入式培训，为市场输送了大量的嵌入式人才。为了普及嵌入式技术，公司计划着手针对高职院校的特点编写一套嵌入式教材。教材的内容涵盖 ARM 体系结构、接口技术、Linux 操作系统、Linux C 语言及 Linux 应用开发实训。本书重点讲解 ARM 体系结构及接口技术部分。

在学习本书之前，读者需要具有数字电路、C 语言等基础知识。通过本书的学习，读者可以掌握 ARM 体系结构和基于 Cortex-A8 核心的 S5PC100 处理器常见硬件接口的开发方法。

本书以 S5PC100 处理器为平台，介绍了嵌入式系统开发的各个主要环节。本书侧重实践，辅以代码加以讲解，从分析的角度来学习嵌入式开发的各种技术。本书使用的工具是 FS-JTAG 仿真器。FS-JTAG 是华清远见研发中心为了推进 Cortex-A8 ARM 处理器的教学，提高合作企业及合作院校广大技术爱好者和培训学员的学习效率，研发出的低价的可以支持 Cortex-A8 的 ARM 仿真器。

本书将嵌入式软/硬件理论讲解和嵌入式实验实践融合在一起，全书共 16 章。其中，第 1 章为嵌入式系统基础知识，介绍了嵌入式系统的组成及嵌入式开发概述。第 2 章为 ARM 技术概述，讲解了 ARM 体系结构、应用选型及编程模型等。第 3 章为 ARM 微处理器的指令系统，重点介绍了 ARM 指令集。第 4 章为 ARM 汇编语言程序设计，主要介绍了 GUN ARM 汇编伪操作、GNU ARM 汇编支持的伪指令、汇编语言与 C 语言的混合编程。第 5 章为 ARM 开发环境搭建，包括 Eclipse 环境介绍、FS-JTAG 仿真器使用等。第 6 章为 GPIO 编程，介绍了 GPIO 的概念及 S5PC100 的 GPIO 操作方法。第 7 章为 ARM 异常及中断处理，介绍了 ARM 处理器的异常处理及 S5PC100 的中断控制器工作原理。第 8 章为串行通信接口，介绍了串行通信的概念及 S5PC100 串口的操作方法。第 9 章为存储器接口，介绍了 NOR Flash、NAND Flash 存储器的操作方法。第 10 章为定时器与 RTC，介绍了定时器的工作原理及 S5PC100 定时器接口的操作方法。第 11 章为 A/D 转换器，介绍了 A/D 转换器的工作原理及 S5PC100 A/D 控制器的操作方法。

第 12 章为 DMA（PL330）控制器，介绍了 ARM 公司最新的 PL330 DMA 控制器的开发方法和 PL330 指令。第 13 章为 LCD 接口设计，介绍了 S5PC100 的 LCD 控制器的工作原理。第 14 章为 CAMIF 接口技术，结合 OV9650 摄像头，介绍了 S5PC100 CAMIF 控制器的开发方法。第 15 章为 SPI 接口，结合 M24PXX SPI Flash，介绍了 SPI 总线协议和 S5PC100 SPI 控制器开发方法。第 16 章为 I2C 接口，结合 LM75 温度传感器，讲解了 I2C 协议和 S5PC100 的 IIC 控制器开发方法。

本书的出版要感谢华清远见嵌入式培训中心的无私帮助。本书的前期组织和后期审校工作都凝聚了培训中心几位老师的心血，他们认真阅读了书稿，提出了大量中肯的建议，并帮助纠正了书稿中的很多错误。

全书由刘洪涛、邹南承担了书稿的编写及全书的统稿工作。书稿的完成需要感谢赵孝强、温尚书、杨胜利、周志强、谭翠君、曾宏安、曹忠明等老师的帮助。

由于作者水平所限，书中不妥之处在所难免，恳请读者批评指正。对于本书的批评和建议，可以发表到 www.farsight.com.cn 技术论坛。

编　者

2012 年 6 月

目　　录

第 1 章　嵌入式系统基础知识

嵌入式系统已成为当前最为热门的领域之一，它无处不在，受到了社会各方面的广泛关注，更有越来越多的人开始学习嵌入式系统开发。本章将向读者介绍嵌入式系统的基本知识，主要内容如下：

- ❑ 嵌入式系统的概述。
- ❑ 嵌入式系统的组成。
- ❑ 嵌入式系统开发举例。
- ❑ 嵌入式系统开发概述。

1.1　嵌入式系统概述

1.1.1　嵌入式系统简介

嵌入式系统已经广泛地渗透到人们的学习、工作、生活中，我们可以看到，嵌入式系统已经应用在科学研究、工程设计、军事技术、各类产业、商业文化艺术、娱乐业及人们的日常生活等方方面面。表 1-1 列举了嵌入式系统应用的部分领域。

表 1-1　嵌入式系统应用领域举例

领　域	应　用
消费电子	信息家电、智能玩具、通信设备、移动存储、视频监控
工业控制	工控设备、智能仪表、汽车电子、电子农业
网　络	网络设备、电子商务、无线传感器
医务医疗	医疗电子
军事国防	军事电子
航空航天	各类飞行设备、卫星等
物 联 网	追溯系统、仓库存储

随着数字信息技术和网络技术的飞速发展，计算机、通信、消费电子的一体化趋势日益明显，这必将培育出一个庞大的嵌入式应用市场。嵌入式系统技术也成了当前关注、学习研究的热点。大家可能会问究竟什么是嵌入式系统呢？嵌入式系统本身是一个相对模糊的定义，不同的组织对其定义也略有不同，但大意是相同的，我们来看一下嵌入式系统的相关定义。

按照电器工程协会（IEEE）的定义，嵌入式系统是用来控制、监控，或者辅助操作机器、装置、工厂等大规模系统的设备（devices used to control, monitor, or assist the operation of equipment, machinery or plants）。这个定义主要是从嵌入式系统的用途方面来进行定义的。

更具一般性且在多数书籍资料中使用较多的关于嵌入式系统的定义如下：嵌入式系统是指以应用为中心，以计算机技术为基础，软件硬件可剪裁，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。

根据以上嵌入式系统的定义，我们可以看出，嵌入式系统是由硬件和软件相结合组成的具有特定功能、用于特定场合的独立系统。其硬件主要由嵌入式微处理器、外围硬件设备组成；其软件主要包括底层系统软件和用户应用软件。

1.1.2 嵌入式系统的特点

1．专用、软/硬件可剪裁可配置

从嵌入式系统定义可以看出，嵌入式系统是面向应用的，和通用系统最大的区别在于嵌入式系统功能专一。根据这个特性，嵌入式系统的软、硬件可以根据需要进行精心设计、量体裁衣、去除冗余，以实现低成本、高性能。也正因如此，嵌入式系统采用的微处理器和外围设备种类繁多，系统不具通用性。

2．低功耗、高可靠性、高稳定性

嵌入式系统大多用在特定场合，要么是环境条件恶劣，要么要求其长时间连续运转，因此嵌入式系统应具有高可靠性、高稳定性、低功耗等特点。

3．软件代码短小精悍

由于成本和应用场合的特殊性，通常嵌入式系统的硬件资源（如内存等）都比较少，因此对嵌入式系统设计也提出了较高的要求。嵌入式系统的软件设计尤其要求高质量，要在有限的资源上实现高可靠性、高性能的系统。虽然随着硬件技术的发展和成本的降低，在高端嵌入式产品上也开始采用嵌入式操作系统，但其和 PC 资源比起来还是少得可怜，所以嵌入式系统的软件代码依然要在保证性能的情况下，占用尽量少的资源，保证产品的高性价比，使其具有更强的竞争力。

4．代码可固化

为了提高执行速度和系统可靠性，嵌入式系统中的软件一般都固化在存储器芯片或单片机本身中，而不是存储于磁盘中。

5．实时性

很多采用嵌入式系统的应用具有实时性要求，所以大多嵌入式系统采用实时性系统。但需要注意的是嵌入式系统不等于实时系统。

6．弱交互性

嵌入式系统不仅功能强大，而且要求使用灵活方便，一般不需要键盘、鼠标等。人机交互以简单方便为主。

7．嵌入式系统软件开发通常需要专门的开发工具和开发环境

在开发一个嵌入系统时，需要事先搭建开发环境及开发系统，如进行 ARM 编程时，需要安装特定的 IDE，如 MDK、IAR 等，如果需要交叉编译时，除了特定的宿主系统外，还要有目标交叉工具链，之所以这样是因为嵌入式系统不具有通用系统那样的单一性，嵌入式系统具有多样性，因此，不同的目标就要为其准备不同的开发环境。

8．要求开发、设计人员有较高的技能

嵌入式系统是将先进的计算机技术、半导体技术和电子技术与各个行业的具体应用相结合后的产物。这一点就决定了它必然是一个技术密集、资金密集、高度分散、不断创新的知识集成系统，从事嵌入式系统开发的人才也必须是复合型人才。

1.1.3　嵌入式系统的发展

1．嵌入式系统主要经历的 4 个阶段

第 1 阶段是以单芯片为核心的可编程控制器形式的系统。这类系统大部分应用于一些专业性强的工业控制系统中，一般没有操作系统的支持，软件通过汇编语言编写。这一阶段系统的主要特点是：系统结构和功能相对单一，处理效率较低，存储容量较小，几乎没有用户接口。由于这种嵌入式系统使用简单、价格低，因此以前在国内工业领域应用较为普遍，但是现在已经远不能适应高效的、需要大容量存储的现代工业控制和新兴信息家电等领域的需求。

第 2 阶段是以嵌入式 CPU 为基础、以简单操作系统为核心的嵌入式系统。其主要特点是：CPU 种类繁多，通用性比较弱；系统开销小，效率高；操作系统达到一定的兼容性和扩展性；应用软件较专业化，用户界面不够友好。

第 3 阶段是以嵌入式操作系统为标志的嵌入式系统。其主要特点是：嵌入式操作系统能运行于各种不同类型的微处理器上，兼容性好；操作系统内核小、效率高，并且具有高度的模块化和扩展性；具备文件和目录管理，支持多任务，支持网络应用，具备图形窗口和用户界面；具有大量的应用程序接口 API，开发应用程序较简单；嵌入式应用软件丰富。

第 4 阶段是以物联网为标志的嵌入式系统。这是一个正在迅速发展的技术。物联网拥有业界最完整的专业物联产品系列，覆盖从传感器、控制器到云计算的各种应用。物联网一方面可以提高经济效益，大大节约成本；另一方面可以为全球经济的复苏提供技术动力。目前，美国、欧盟等都在投入巨资深入研究探索物联网。我国也正在高度关注、重视物联网的研究，工业和信息化部会同有关部门，在新一代信息技术方面正在开展研究，以形成支持新一代信息技术发展的政策措施。

2．未来嵌入式系统的发展趋势

1）小型化、智能化、网络化、可视化

随着技术水平的提高和人们生活的需要，嵌入式设备正朝着小型化便携式和智能化的方向发展。如果携带笔记本电脑外出办事，你肯定希望它轻薄小巧，甚至可能希望有一种更便携的设备来替代它，目前的 PAD、智能手机，便携投影仪等都是因类似的需求而出现

的。对嵌入式而言，可以说是已经进入了嵌入式互联网时代（有线网、无线网、广域网、局域网的组合），嵌入式设备和互联网的紧密结合，更为我们的日常生活带来了极大的方便和无限的想象空间。除此之外，人工智能、模式识别技术也将在嵌入式系统中得到应用，使得嵌入式系统更具人性化、智能化。

2）多核技术的应用

人们需要处理的信息越来越多，这就要求嵌入式设备运算能力更强，因此需要设计出更强大的嵌入式处理器，多核技术处理器在嵌入式中的应用将更为普遍。

3）低功耗（节能）、绿色环保

嵌入式系统的硬件和软件设计都在追求更低的功耗，以求嵌入式系统能获得更长的可靠工作时间。例如：手机的通话和待机时间，mp3 听音乐的时间等。同时，绿色环保型嵌入式产品将更受人们青睐，在嵌入式系统设计中也会更多地考虑如辐射和静电等问题。

4）云计算、可重构、虚拟化等技术被进一步应用到嵌入式系统中

简单讲，云计算是将计算分布在大量的分布式计算机上，这样我们只需要一个终端，就可以通过网络服务来实现我们需要的计算任务，甚至是超级计算任务。云计算（Cloud Computing）是分布式处理（Distributed Computing）、并行处理（Parallel Computing）和网格计算（Grid Computing）的发展，或者说是这些计算机科学概念的商业实现。在未来几年里，云计算将得到进一步发展与应用。

可重构性是指在一个系统中，其硬件模块或（和）软件模块均能根据变化的数据流或控制流对系统结构和算法进行重新配置（或重新设置）。可重构系统最突出的优点就是能够根据不同的应用需求，改变自身的体系结构，以便与具体的应用需求相匹配。

虚拟化是指计算机软件在一个虚拟的平台上而不是真实的硬件上运行。虚拟化技术可以简化软件的重新配置过程，易于实现软件的标准化。其中 CPU 的虚拟化可以单 CPU 模拟多 CPU 并行运行，允许一个平台同时运行多个操作系统，并且都可以在相互独立的空间内运行而互不影响，从而提高工作效率和安全性，虚拟化技术是降低多内核处理器系统开发成本的关键。虚拟化技术是未来几年最值得期待和关注的关键技术之一。

随着各种技术的成熟与在嵌入式系统中的应用，将不断为嵌入式系统增添新的魅力和发展空间。

5）嵌入式软件开发平台化、标准化、系统可升级，代码可复用将更受重视

嵌入式操作系统将进一步走向开放、开源、标准化、组件化。嵌入式软件开发平台化也将是今后的一个趋势，越来越多的嵌入式软/硬件行业标准将出现，最终的目标是使嵌入式软件开发简单化，这也是一个必然规律。同时随着系统复杂度的提高，系统可升级和代码复用技术在嵌入式系统中得到更多的应用。

6）嵌入式系统软件将逐渐 PC 化

需求和网络技术的发展是嵌入式系统发展的一个源动力，随着移动互联网的发展，将进一步促进嵌入式系统软件 PC 化。如前所述，结合跨平台开发语言的广泛应用，未来嵌入式软件开发的概念将被逐渐淡化，也就是嵌入式软件开发和非嵌入式软件开发的区别将逐渐减小。

7）融合趋势

嵌入式系统软/硬件融合、产品功能融合、嵌入式设备和互联网的融合趋势加剧。嵌入式系统设计中软/硬件结合将更加紧密，软件将是其核心。消费类产品将在运算能力和便携方面进一步融合。传感器网络将迅速发展，其将极大地促进嵌入式技术和互联网技术的融合。

8）安全性

随着嵌入式技术和互联网技术的结合发展，嵌入式系统的信息安全问题日益凸显，保证信息安全也成为了嵌入式系统开发的重点和难点。

1.2　嵌入式系统的组成

从前面的介绍我们可以知道，嵌入式系统总体上是由硬件和软件组成的，硬件是其基础，软件是其核心与灵魂。它们之间的关系如图 1-1 所示。

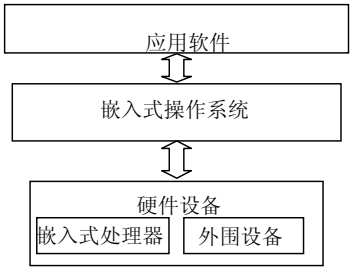

图 1-1　嵌入式系统结构简图

1.2.1　嵌入式系统硬件组成

嵌入式系统硬件设备包括嵌入式处理器和外围设备。其中的嵌入式处理器（CPU）是嵌入式系统的核心部分，它与通用处理器最大的区别在于，嵌入式处理器大多工作在为特定用户群所专门设计的系统中，它将通用处理器中许多由板卡完成的任务集成到芯片内部，从而有利于嵌入式系统在设计时趋于小型化，同时还具有很高的效率和可靠性。如今，全世界嵌入式处理器已经超过 1000 多种，流行的体系结构有 30 多个系列，其中以 ARM、PowerPC、MC 68000、MIPS 等使用得最为广泛。

外围设备是嵌入式系统中用于完成存储、通信、调试、显示等辅助功能的其他部件。目前常用的嵌入式外围设备按功能可以分为存储设备（如 RAM、SRAM、Flash 等）、通信设备（如 RS-232 接口、SPI 接口、以太网接口等）和显示设备（如显示屏等）3 类。

常见存储器概念包括：RAM、ROM、SRAM、DRAM、SDRAM、EPROM、EEPROM、Flash。

存储器可以分为很多种类，其中根据掉电数据是否丢失可以分为 RAM（随机存取存储器）和 ROM（只读存储器），其中 RAM 的访问速度比较快，但掉电后数据会丢失，而 ROM 掉电后数据不会丢失。人们通常所说的内存即指系统中的 RAM。

RAM 又可分为 SRAM（静态存储器）和 DRAM（动态存储器）。SRAM 是利用双稳态触发器来保存信息的，只要不掉电，信息是不会丢失的。DRAM 是利用 MOS（金属氧化物半导体）电容存储电荷来存储信息，因此必须通过不停地给电容充电来维持信息，所以 DRAM 的成本、集成度、功耗等明显优于 SRAM。

而通常人们所说的 SDRAM 是 DRAM 的一种，它是同步动态存储器，利用一个单一的系统时钟同步所有的地址数据和控制信号。使用 SDRAM 不但能提高系统表现，还能简化设计、提供高速的数据传输。在嵌入式系统中经常使用。

EPROM、EEPROM 都是 ROM 的一种，分别为可擦除可编程 ROM 和电可擦除 ROM，但使用不是很方便。

Flash 也是一种非易失性存储器（掉电不会丢失），它擦写方便，访问速度快，已在很大程度上取代了传统的 EPROM 的地位。由于它具有和 ROM 一样掉电不会丢失的特性，因此很多人称其为 Flash ROM。

1.2.2 嵌入式系统软件组成

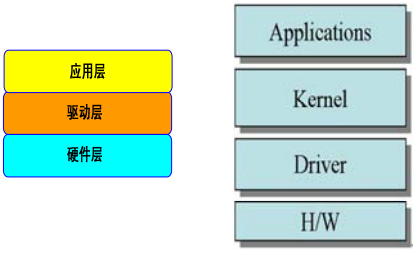

图 1-2 嵌入式系统软件组成图

在嵌入式系统不同的应用领域和不同的发展阶段，嵌入式系统软件组成也不完全相同。其大致如图 1-2 所示。

图 1-2 左侧显示，在某些特殊领域中，嵌入式系统软件没有使用通用计算机系统。嵌入式操作系统从嵌入式发展的第 3 阶段起开始引入。嵌入式操作系统不仅具有通用操作系统的一般功能，如向上提供对用户的接口（如图形界面、库函数 API 等），向下提供与硬件设备交互的接口（硬件驱动程序等），管理复杂的系统资源，同时，它还在系统实时性、硬件依赖性、软件固化性及应用专用性等方面，具有更加鲜明的特点。

应用软件是针对特定应用领域，基于某一固定的硬件平台，用来达到用户预期目标的计算机软件。由于嵌入式系统自身的特点，决定了嵌入式应用软件不仅要求做到准确性、安全性和稳定性等方面需要，而且还要尽可能地进行代码优化，以减少对系统资源的消耗，降低硬件成本。

1.3 嵌入式操作系统举例

嵌入式操作系统主要有商业版和开源版两大阵营，从长远看，嵌入式系统开源、开放将是其发展趋势。

1.3.1　商业版嵌入式操作系统

VxWorks 作为商业版嵌入式操作系统的典型代表，这里有必要简要介绍一下。

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），它是在当前市场占有率最高的嵌入式实时操作系统。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，使得它们造成的延迟很短。另外 VxWorks 提供的多任务机制，对任务的控制采用了优先级抢占（Linux 2.6 内核也采用了优先级抢占的机制）和轮转调度机制，这充分保证了可靠的实时性，并使同样的硬件配置能满足更强的实时性要求。另外 VxWorks 具有高度的可靠性，从而保证了用户工作环境的稳定。同时，VxWorks 还有很完备强大的集成开发环境，这也大大方便了用户的使用。

但是，由于 VxWorks 的开发和使用都需要交高额的专利费，因此大大增加了用户的开发成本。同时，由于 VxWorks 的源码不公开，造成它部分功能的更新（如网络功能模块）滞后。

1.3.2　开源版嵌入式操作系统

嵌入式 Linux（Embedded Linux）作为开源版嵌入式操作系统的典型，这里也简单介绍一下它的特性。

嵌入式 Linux 是指对标准 Linux 经过小型化裁剪处理之后，能够固化在容量只有几 KB 或者几 MB 的存储器芯片或者单片机中，是适合于特定嵌入式应用场合的专用 Linux 操作系统。在目前已经开发成功的嵌入式系统中，大约有一半使用的是 Linux。这与它自身的优良特性是分不开的。

嵌入式 Linux 同 Linux 一样，具有低成本、多种硬件平台支持、优异的性能和良好的网络支持等优点。另外，为了更好地适应嵌入式领域的开发，嵌入式 Linux 还在 Linux 基础上做了部分改进。

1．改善的内核结构

Linux 内核采用的是整体式结构（Monolithic），整个内核是一个单独的、非常大的程序，这样虽然能够使系统的各个部分直接沟通，提高系统响应速度，但与嵌入式系统存储容量小、资源有限的特点不相符。因此，在嵌入式系统经常采用的是另一种称为微内核（Microkernel）的体系结构，即内核本身只提供一些最基本的操作系统功能，如任务调度、内存管理、中断处理等，而类似于文件系统和网络协议等附加功能则运行在用户空间中，并且可以根据实际需要进行取舍。这样就大大减小了内核的体积，便于维护和移植。

2．提高的系统实时性

由于现有的 Linux 是一个通用的操作系统，虽然它也采用了许多技术来加快系统的运行和响应速度，但从本质上来说并不是一个嵌入式实时操作系统。因此，人们利用 Linux 作为底层操作系统，在其上进行实时化改造，从而构建出一个具有实时处理能力的嵌入式

系统，如 RT-Linux 已经成功地应用于航天飞机的空间数据采集、科学仪器测控和电影特技图像处理等各种领域。

1.4 嵌入式系统开发概述

由于受嵌入式系统本身的特性所影响，嵌入式系统开发与通用系统的开发有很大的区别。嵌入式系统的开发主要分为系统总体开发、嵌入式硬件开发和嵌入式软件开发 3 大部分，其总体流程图如图 1-3 所示。

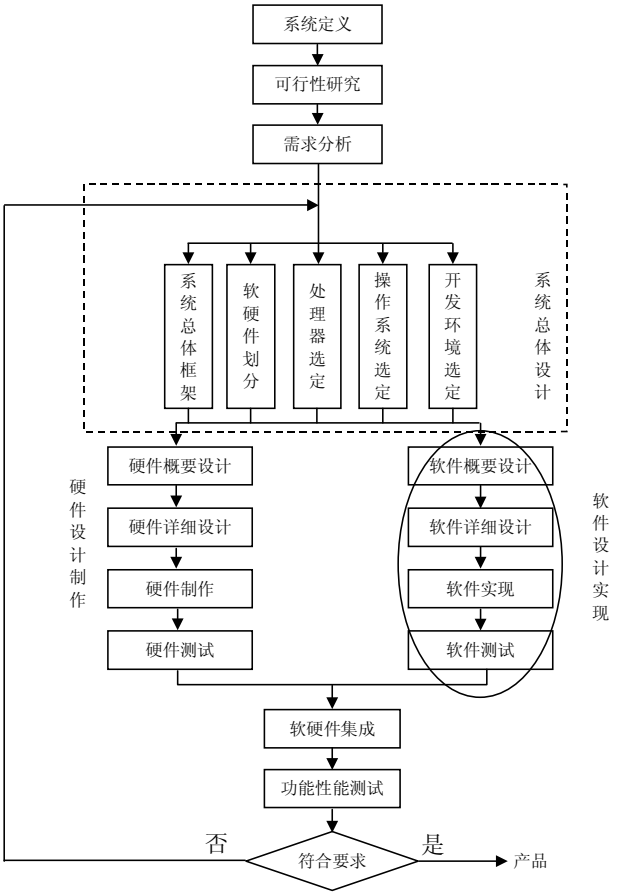

图 1-3　嵌入式系统开发流程图

在系统总体开发中，由于嵌入式系统与硬件依赖程序非常紧密，往往某些需求只能通过特定的硬件才能实现，因此需要进行处理器选型，以更好地满足产品的需求。另外，对于有些硬件和软件都可以实现的功能，就需要在成本和性能上做出选择。通过硬件实现往往会增加产品的成本，但能大大提高产品的性能和可靠性。

再次，开发环境的选择对于嵌入式系统的开发也有很大的影响。这里的开发环境包括嵌入式操作系统的选择及开发工具的选择等。本书在 1.3 节对各种不同的嵌入式操作系统进行了比较，读者可以以此为依据进行相关的选择。比如，对开发成本和进度限制较大的产品可以选择嵌入式 Linux，对实时性要求非常高的产品可以选择 VxWorks 等。

嵌入式软件开发总体流程为图 1-3 中“软件设计实现”部分所示，它同通用计算机软件开发一样，分为需求分析、软件概要设计、软件详细设计、软件实现和软件测试。其中嵌入式软件需求分析与硬件的需求分析合二为一，故没有分开画出。

由于在嵌入式软件开发的工具非常多，为了更好地帮助读者选择开发工具，下面首先对嵌入式软件开发过程中所使用的工具进行简单归纳。

嵌入式软件的开发工具根据不同的开发过程而划分，比如在需求分析阶段，可以选择 IBM 的 Rational Rose 等软件，而在程序开发阶段可以采用 CodeWarrior 等，在调试阶段可以采用 Multi-ICE 等。同时，不同的嵌入式操作系统往往会有配套的开发工具，比如 VxWorks 有集成开发环境 Tornado，WinCE 的集成开发环境 WinCE Platform 等。此外，不同的处理器可能还有针对的开发工具，比如 ARM 的常用集成开发工具 ADS 等。在这里，大多数软件都有比较高的使用费用，但也可以大大加快产品的开发进度，用户可以根据需求自行选择。

嵌入式系统的软件开发与通常软件开发的区别主要在于软件实现部分，其中又可以分为交叉编译和交叉调试两部分，下面分别对这两部分进行讲解。

1．交叉编译

嵌入式软件开发所采用的编译为交叉编译。所谓交叉编译就是在一个平台上生成可以在另一个平台上执行的代码。因此，不同的 CPU 需要有相应的编译器，而交叉编译就如同翻译一样，把相同的程序代码翻译成不同的 CPU 对应语言。要注意的是，编译器本身也是程序，也要在与之对应的某一个 CPU 平台上运行。

这里一般把进行交叉编译的主机称为宿主机，也就是普通的通用计算机，而把程序实际的运行环境称为目标机，也就是嵌入式系统环境。由于一般通用计算机拥有非常丰富的系统资源、使用方便的集成开发环境和调试工具等，而嵌入式系统的系统资源非常紧缺，没有相关的编译工具，因此，嵌入式系统的开发需要借助宿主机来编译出目标机的可执行代码。

由于编译的过程包括编译、链接等几个阶段，因此，嵌入式的交叉编译也包括交叉编译、交叉链接等过程，通常 ARM 的交叉编译器为 arm-elf-gcc，交叉链接器为 arm-elf-ld，交叉编译过程如图 1-4 所示。

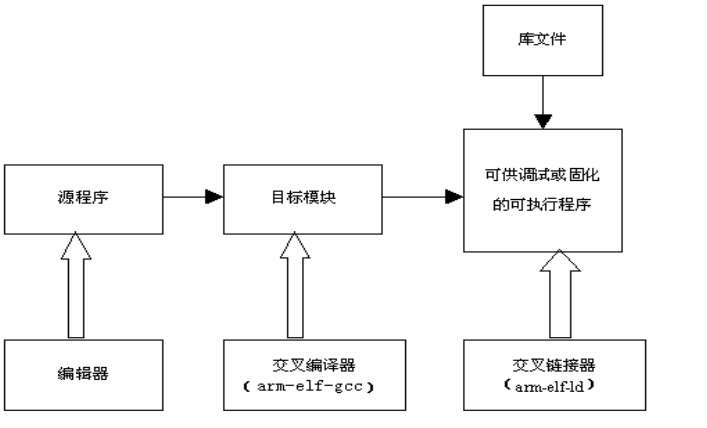

图 1-4　嵌入式交叉编译过程

2．交叉调试

嵌入式软件经过编译和链接后即进入调试阶段，调试是软件开发过程中必不可少的一个环节，嵌入式软件开发过程中的交叉调试与通用软件开发过程中的调试方式有很大的差别。在常见软件开发中，调试器与被调试的程序往往运行在同一台计算机上，调试器是一个单独运行的进程，它通过操作系统提供的调试接口来控制被调试的进程。而在嵌入式软件开发中，调试时采用的是在宿主机和目标机之间进行的交叉调试，调试器仍然运行在宿主机的通用操作系统之上，但被调试的进程却是运行在基于特定硬件平台的嵌入式操作系统中，调试器和被调试进程通过串口或者网络进行通信，调试器可以控制、访问被调试进程，读取被调试进程的当前状态，并能够改变被调试进程的运行状态。

嵌入式系统的交叉调试有多种方法，主要可分为软件方式和硬件方式两种。它们一般都具有如下一些典型特点。

- 调试器和被调试进程运行在不同的机器上，调试器运行在 PC 或者工作站上（宿主机），而被调试的进程则运行在各种专业调试板上（目标机）。
- 调试器通过某种通信方式（串口、并口、网络、JTAG 等）控制被调试进程。
- 在目标机上一般会具备某种形式的调试代理，它负责与调试器共同配合完成对目标机上运行的进程进行调试。这种调试代理可能是某些支持调试功能的硬件设备，也可能是某些专门的调试软件（如 GdbServer）。
- 目标机可能是某种形式的系统仿真器，通过在宿主机上运行目标机的仿真软件，整个调试过程可以在一台计算机上运行。此时物理上虽然只有一台计算机，但逻辑上仍然存在着宿主机和目标机的区别。

下面分别就软件调试和硬件调试两种方式进行详细介绍。

1）软件调试

软件方式调试主要是通过插入调试桩的方式来进行的。用调试桩方式进行调试是通过目标操作系统和调试器内分别加入某些功能模块，二者互通信息来进行调试。该方式的典型调试器有 Gdb 调试器。

Gdb 的交叉调试器分为 GdbServer 和 GdbClient，其中的 GdbServer 作为调试桩安装在目标板上，GdbClient 是驻于本地的 Gdb 调试器。它们的调试原理如图 1-5 所示。

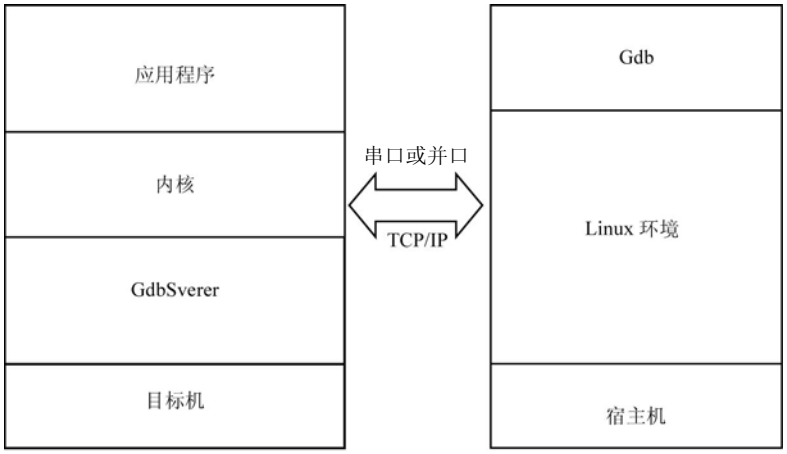

图 1-5　Gdb 远程调试原理图

Gdb 调试桩的工作流程如下。

（1）建立调试器（本地 Gdb）与目标操作系统的通信连接，可通过串口、网卡、并口等多种方式。

（2）在目标机上开启 GdbServer 进程，并监听对应端口。

（3）在宿主机上运行调试器 Gdb，这时，Gdb 就会自动寻找远端的通信进程，也就是 Gdbserver 的所在进程。

（4）在宿主机上的 Gdb 通过 GdbServer 请求对目标机上的程序发出控制命令。这时，Gdbserver 将请求转化为程序的地址空间或目标平台的某些寄存器的访问，这对于没有虚拟存储器的简单的嵌入式操作系统而言，是十分容易的。

（5）GdbServer 把目标操作系统的所有异常处理转向通信模块，并告知宿主机上 Gdb 当前异常。

（6）宿主机上的 Gdb 向用户显示被调试程序产生了哪一类异常。

这样就完成了调试的整个过程。这个方案的实质是用软件接管目标机的全部异常处理及部分中断处理，并在其中插入调试端口通信模块，与主机的调试器进行交互。但是它只能在目标机系统初始化完毕、调试通信端口初始化完成后才能起作用，因此，一般只能用于调试运行于目标操作系统之上的应用程序，而不宜用来调试目标操作系统的内核代码及启动代码。而且，它必须改变目标操作系统，因此，也就多了一个不用于正式发布的调试版。

2）硬件调试

相对于软件调试而言，使用硬件调试器可以获得更强大的调试功能和更优秀的调试性能。硬件调试器的基本原理是通过仿真硬件的执行过程，让开发者在调试时可以随时了解

到系统的当前执行情况。目前嵌入式系统开发中最常用到的硬件调试器是 ROMMonitor、ROMEmulator、In-CircuitEmulator 和 In-CircuitDebugger。

（1）采用 ROMMonitor 方式进行交叉调试需要在宿主机上运行调试器，在目标机上运行 ROM 监视器（ROMMonitor）和被调试程序，宿主机通过调试器与目标机上的 ROM 监视器遵循远程调试协议建立通信连接。ROM 监视器可以是一段运行在目标机 ROM 上的可执行程序，也可以是一个专门的硬件调试设备，它负责监控目标机上被调试程序的运行情况，能够与宿主机端的调试器一同完成对应用程序的调试。

在使用这种调试方式时，被调试程序首先通过 ROM 监视器下载到目标机，然后在 ROM 监视器的监控下完成调试。

优点：ROM 监视器功能强大，能够完成设置断点、单步执行、查看寄存器、修改内存空间等各项调试功能。

缺点：同软件调试一样，使用 ROM 监视器目标机和宿主机必须建立通信连接。

其原理图如图 1-6 所示。

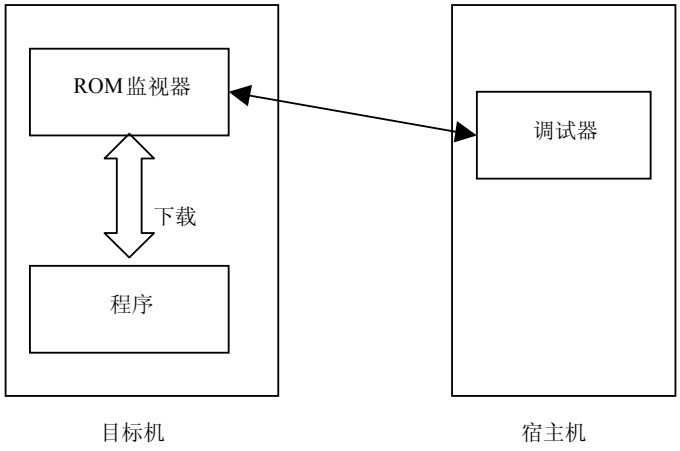

图 1-6　ROMMonitor 调试方式

（2）采用 ROMEmulator 方式进行交叉调试时需要使用 ROM 仿真器，并且它通常被插入到目标机上的 ROM 插槽中，专门用于仿真目标机上的 ROM 芯片。

在使用这种调试方式时，被调试程序首先下载到 ROM 仿真器中，因此等效于下载到目标机的 ROM 芯片上，然后在 ROM 仿真器中完成对目标程序的调试。

优点：避免了每次修改程序后都必须重新烧写到目标机的 ROM 中。

缺点：ROM 仿真器本身比较昂贵，功能相对来讲又比较单一，只适应于某些特定场合。

其原理图如图 1-7 所示。

（3）采用 In-Circuit Emulator（ICE）方式进行交叉调试时需要使用在线仿真器，它是目前最为有效的嵌入式系统的调试手段。它是仿照目标机上的 CPU 而专门设计的硬件，可以完全仿真处理器芯片的行为。仿真器与目标板可以通过仿真头连接，与宿主机可以通过串口、并口、网线或 USB 口等连接方式。由于仿真器自成体系，所以调试时既可以连接目

标板，也可以不连接目标板。在线仿真器提供了非常丰富的调试功能。在使用在线仿真器进行调试的过程中，可以按顺序单步执行，也可以倒退执行，还可以实时查看所有需要的数据，从而给调试过程带来很多的便利。嵌入式系统应用的一个显著特点是与现实世界中的硬件直接相关，并存在各种异变和事先未知的变化，从而给微处理器的指令执行带来各种不确定因素，这种不确定性在目前情况下只有通过在线仿真器才有可能发现。

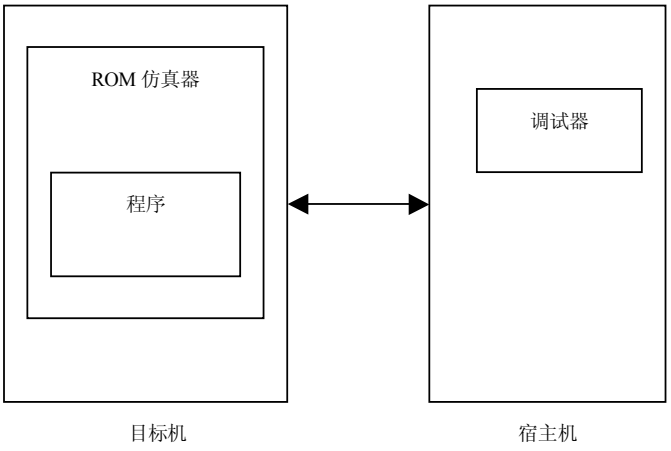

图 1-7　ROMEmulator 调试方式

优点：功能强大，软/硬件都可做到完全实时在线调试。

缺点：价格昂贵。

其原理图如图 1-8 所示。

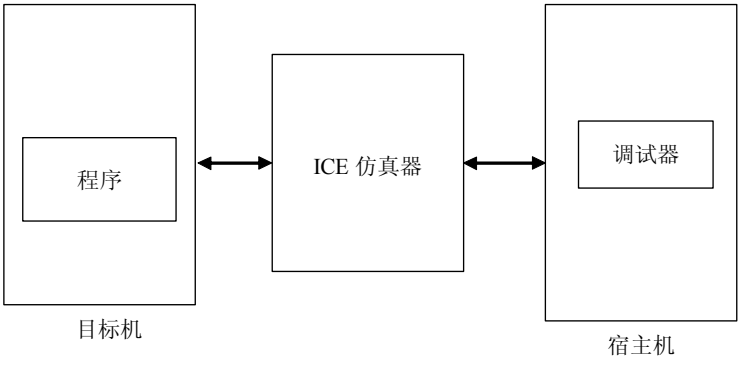

图 1-8　ICE 调试方式

（4）采用 In-Circuit Debugger（ICD）方式进行交叉调试时需要使用在线调试器。由于 ICE 的价格非常昂贵，并且每种 CPU 都需要一种与之对应的 ICE，使得开发成本非常高。一个比较好的解决办法是让 CPU 直接在其内部实现调试功能，并通过在开发板上引出的调试端口发送调试命令和接收调试信息，完成调试过程。应用非常广泛的 ARM 处理器的 JTAG 端口技术就是由此而诞生的。

JTAG 是 1985 年指定的检测 PCB 和 IC 芯片的一个标准。1990 年被修改成为 IEEE 的一个标准，即 IEEE1149.1。JTAG 标准所采用的主要技术为边界扫描技术，它的基本思想就是在靠近芯片的输入/输出引脚上增加一个移位寄存器单元。因为这些移位寄存器单元都分布在芯片的边界上（周围），所以被称为边界扫描寄存器（Boundary-Scan Register Cell）。

当芯片处于调试状态时候，这些边界扫描寄存器可以将芯片和外围的输入/输出隔离开来。通过这些边界扫描寄存器单元，可以实现对芯片输入/输出信号的观察和控制。对于芯片的输入引脚，可通过与之相连的边界扫描寄存器单元把信号（数据）加载到该引脚中去；对于芯片的输出引脚，可以通过与之相连的边界扫描寄存器单元“捕获”（Capture）该引脚的输出信号。这样，边界扫描寄存器提供了一个便捷的方式用于观测和控制所需要调试的芯片。

现在较为高档的微处理器都带有 JTAG 接口，包括 ARM 经典系列、Cortex 系列、DSP 等，通过 JTAG 接口可以方便地对目标系统进行测试，同时，还可以实现 Flash 的编程，是非常受人欢迎的。

优点：连接简单，成本低。

缺点：特性受制于芯片厂商。

其原理图如图 1-9 所示。

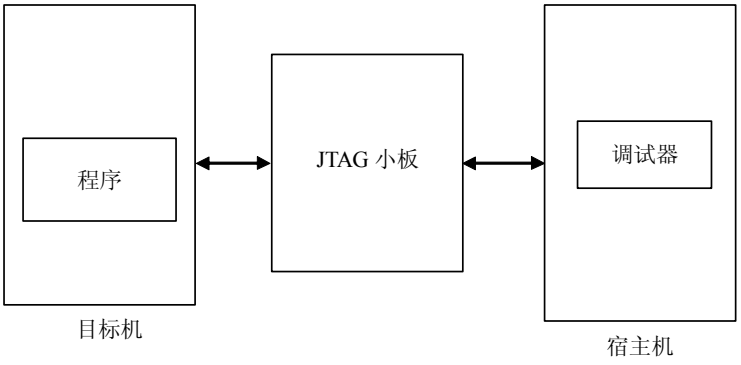

图 1-9　JTAG 调试方式

1.5　学好微处理器在嵌入式学习中的重要性

处理器在嵌入式开发中非常重要，如图 1-10 所示。

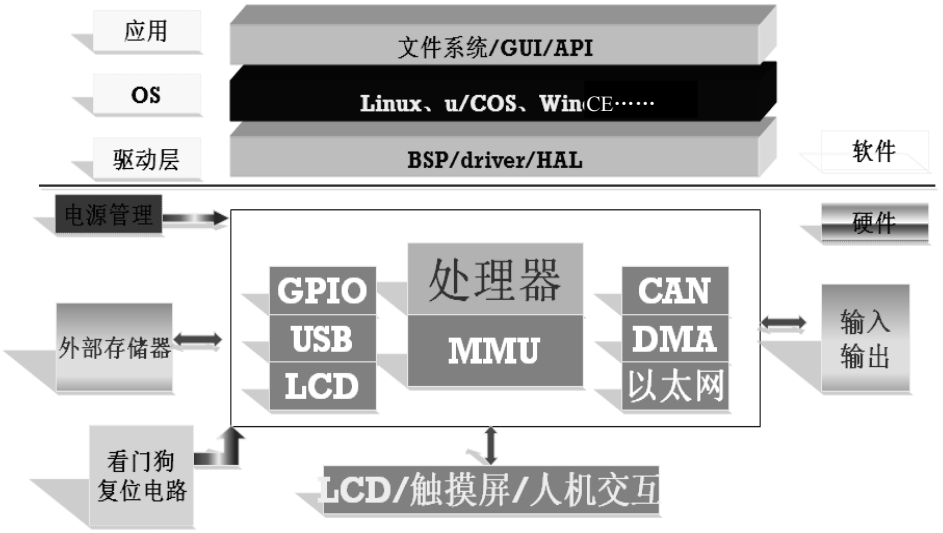

图 1-10　处理器在嵌入式开发中的重要性

1．处理器的角色

在嵌入式开发中，各种外设的控制器操作都要靠处理器来进行，包括初始化、数据存取等，常见的外设控制器如 LCD 控制器、USB 控制器、GPIO、电源管理器、I2C 控制器、SPI 控制器、摄像头控制器等，这些控制器的共有特征都是不仅要对处理器的体系特征有所了解，还要掌握处理器指令以便能够初始化这些控制器，使其能正常操作外设工作。

2．底层的重要性

开发中，作为前期计划之一，就是硬件平台的选择，以及各个外设的选择，其中，嵌入式系统的核心部件是各种类型的嵌入式处理器。目前全世界嵌入式处理器的品种总量已经超过 1000 多种，流行体系结构有三十几个系列。但与全球 PC 市场不同的是，没有一种微处理器和微处理器公司可以主导嵌入式系统，仅以 32 位的 CPU 而言，就有 100 种以上嵌入式微处理器。由于嵌入式系统设计的差异性极大，因此选择是多样化的，这样对于嵌入式处理器的学习就显得很重要了。

3．对于嵌入式理解的深度

在学习中，对于嵌入式的底层来说，除了硬件工程师，那么最底层的则是驱动开发工程师，这种工程师不仅要看得懂芯片手册，更要看懂原理图和硬件紧密相关的资料，在这些基础上才可以做好开发工作，因此，对于处理器的学习来说，不仅是对底层环境的认知加深，更是对一名嵌入式工程师的底层知识的提炼。

在学习时，学习一套嵌入式处理器架构即可，因为大部分的体系架构其实思想是相通的，只要掌握了一种处理器架构，那么其他的只要根据所占部分不大的差异而学习，就能很快掌握新的处理器。

4．行业需求

嵌入式行业是一个新兴而发展迅速的行业，随着网络等云计算技术的推广和应用，智能终端设备遍布于我们的身边，2011 年嵌入式芯片厂商 ARM 曾宣布，基于 ARM 的芯片处理器出货量已接近 80 亿个，这个数量还将以每年至少 30%的速度增长。可见其相关联的产业之巨大，同时巨大的产业变革带来的是新型劳动力的需求和经济利益的扩大，最终，嵌入式行业对芯片型人才的紧缺是相当严重的，因此从长远考虑，学习处理器对于一位即将从事嵌入式开发的工程师来说都是势在必行的事情。

1.6 本章小结

本章向读者简单介绍了嵌入式系统的概念、特点、发展及开发等问题，希望通过阅读本章读者能对嵌入式系统和嵌入式系统开发有一个基本了解，以便为后面章节的学习打下基础。

1.7 思考题

1．什么是嵌入式系统？列举出几个你身边熟悉的嵌入式系统的产品。
2．嵌入式系统由哪几部分组成？
3．列举出 3 种你知道的嵌入式操作系统。
4．简述嵌入式系统的特点。

第 2 章　ARM 技术概述

ARM 体系结构的处理器在嵌入式中的应用是非常广泛的，本章将向读者介绍 ARM 处理器的基本知识。通过阅读本章，读者将了解以下主要内容：

- ARM 体系结构的技术特征及发展。
- ARM 微处理器简介。
- ARM 微处理器结构。
- ARM 微处理器的应用选型。
- Cortex-A8 内部功能及特点。
- 数据类型。
- Cortex-A8 存储系统。
- 流水线。
- 寄存器组织 S。
- 程序状态寄存器。
- SAMSUNG S5PC100 处理器介绍。

2.1　ARM 体系结构的技术特征及发展

ARM（Advanced RISC Machines）有 3 种含义，它是一个公司的名称，是一类微处理器的通称，还是一种技术的名称。

2.1.1　ARM 公司简介

1991 年 ARM 公司（Advanced RISC Machine Limited）成立于英国剑桥，最早由 Arcon、Apple 和 VLSI 合资成立，主要出售芯片设计技术的授权，1985 年 4 月 26 日，第一个 ARM 原型在英国剑桥的 Acorn 计算机有限公司诞生（在美国 VLSI 公司制造）。目前，ARM 架构处理器已在高性能、低功耗、低成本的嵌入式应用领域中占据了领先地位。

ARM 公司最初只有 12 人，经过多年的发展，ARM 公司已拥有近千名员工，在许多国家都设立了分公司，包括在中国上海的分公司。目前，采用 ARM 技术知识产权（IP）核的微处理器，即我们通常所说的 ARM 微处理器，已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场，基于 ARM 技术的微处理器应用约占据了 32 位 RISC 微处理器 80%以上的市场份额，其中，在手机市场，ARM 占有绝对的垄断地位。可以说，ARM 技术正在逐步渗入到人们生活中的各个方面，而且随着 32 位 CPU 价格的不断下降和开发环境的不断成熟，ARM 技术会应用得越来越广泛。

ARM 公司是专门从事基于 RISC 技术芯片设计开发的公司，作为嵌入式 RISC 处理器的知识产权 IP 供应商，公司本身并不直接从事芯片生产，而是靠转让设计许可由合作公司生产各具特色的芯片，世界各大半导体生产商从 ARM 公司购买其设计的 ARM 微处理器核，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的 ARM 微处理器芯片进入市场，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，全世界有几十家大的半导体公司都使用 ARM 公司的授权，其中包括 Intel、IBM、SAMSUNG、LG 半导体、NEC、SONY、PHILIP 等公司，这也使得 ARM 技术获得更多的第三方工具、制造厂商、软件的支持，又使整个系统成本降低，使产品更容易进入市场并被消费者所接受，更具有竞争力。

2.1.2 ARM 技术特征

ARM 的成功，一方面得益于它独特的公司运作模式，另一方面，当然来自于 ARM 处理器自身的优良性能。作为一种先进的 RISC 处理器，ARM 处理器有如下特点。

- 体积小、低功耗、低成本、高性能。
- 支持 Thumb（16 位）/ARM（32 位）双指令集，能很好地兼容 8 位/16 位器件。
- 大量使用寄存器，指令执行速度更快。
- 大多数数据操作都在寄存器中完成。
- 寻址方式灵活简单，执行效率高。
- 指令长度固定。

此处有必要解释一下 RISC 处理器的概念及其与 CISC 微处理器的区别。

1．嵌入式 RISC 微处理器

RISC（Reduced Instruction Set Computer）是精简指令集计算机，RISC 把着眼点放在如何使计算机的结构更加简单和如何使计算机的处理速度更加快速上。RISC 选取了使用频率最高的简单指令，抛弃复杂指令，固定指令长度，减少指令格式和寻址方式，不用或少用微码控制。这些特点使得 RISC 非常适合嵌入式处理器。

2．嵌入式 CISC 微处理器

传统的复杂指令级计算机（CISC）则更侧重于硬件执行指令的功能性，使 CISC 指令及处理器的硬件结构变得更复杂。这些会导致成本、芯片体积的增加，影响其在嵌入式产品的应用。如表 2-1 所示描述了 RISC 和 CISC 之间的主要区别。

表 2-1　RISC 和 CISC 之间主要的区别

指　　标	RISC	CISC
指令集	一个周期执行一条指令，通过简单指令的组合实现复杂操作；指令长度固定	指令长度不固定，执行需要多个周期
流水线	流水线每周期前进一步	指令的执行需要调用微代码的一个微程序
寄存器	更多通用寄存器	用于特定目的的专用寄存器
Load/Store 结构	独立的 Load 和 Store 指令完成数据在寄存器和外部存储器之间的传输	处理器能够直接处理存储器中的数据

2.1.3　ARM 体系架构的发展

在讨论 ARM 体系结构前，先解释一下体系结构的定义。

体系架构定义了指令集（ISA）和基于这一体系结构下处理器的编程模型。基于同种体系结构可以有多种处理器，每个处理器性能不同，所面向的应用不同，每个处理器的实现都要遵循这一体系结构。ARM 体系结构为嵌入系统发展商提供很高的系统性能，同时保持优异的功耗和面积效率。

ARM 体系结构为满足 ARM 合作者及设计领域的一般需求正稳步发展。目前，ARM 体系结构共定义了 7 个版本，从版本 1 到版本 7，ARM 体系的指令集功能不断扩大，不同系列的 ARM 处理器，性能差别很大，应用范围和对象也不尽相同，但是，如果是相同的 ARM 体系结构，那么基于它们的应用软件是兼容的。

1．v1 架构

V1 版本的 ARM 处理器并没有实现商品化，采用的地址空间是 26 位，寻址空间是 64MB，在目前的版本中已不再使用这种结构。

2．v2 架构

与 v1 结构的 ARM 处理器相比，v2 架构的 ARM 处理器的指令结构要有所完善，比如增加了乘法指令并且支持协处理器指令，该版本的处理器仍然采用 26 位的地址空间。

3．v3 架构

从 v3 结构开始，ARM 处理器的体系结构有了很大的改变，实现了 32 位的地址空间，指令结构相对前面的两种结构也所完善。

4．v4 架构

v4 结构的 ARM 处理器增加了半字指令的读取和写入操作，增加了处理器系统模式，并且有了 T 变种——v4T，在 Thumb 状态下支持的是 16 位的 Thumb 指令集。属于 v4T（支持 Thumb 指令）体系结构的处理器（核）有 ARM7TDMI、ARM7TDMI-S（ARM7TDMI 综合版本）、ARM710T（ARM7TDMI 核的处理器）、ARM720T（ARM7TDMI 核的处理器）、ARM740T（ARM7TDMI 核的处理器）、ARM9TDMI、ARM910T（ARM9TDMI 核的处理器）、ARM920T（ARM9TDMI 核的处理器）、ARM940T（ARM9TDMI 核的处理器）和 StrongARM（Intel 公司的产品）。

5．v5 架构

v5 架构的 ARM 处理器提升了 ARM 和 Thumb 两种指令的交互工作能力，同时有了 DSP 指令（-v5E 架构）、Java 指令（-v5J 架构）的支持。属于 v5T（支持 Thumb 指令）体系结构的处理器（核）有 ARM10TDMI 和 ARM1020T（ARM10TDMI 核处理器）。

属于 v5TE（支持 Thumb、DSP 指令）体系结构的处理器（核）有 ARM9E、ARM9E-S（ARM9E 可综合版本）、ARM946（ARM9E 核的处理器）、ARM966（ARM9E 核的处理器）、

ARM10E、ARM1020E（ARM10E 核处理器）、ARM1022E（ARM10E 核的处理器）和 Xscale（Intel 公司产品）。

属于 v5TEJ（支持 Thumb、DSP 指令、Java）体系结构的处理器（核）有 ARM9EJ、ARM9EJ-S（ARM9EJ 可综合版本）、ARM926EJ（ARM9EJ 核的处理器）和 ARM10EJ。

6．v6 架构

v6 架构是在 2001 年发布的，在该版本中增加了媒体指令，属于 v6 体系结构的处理器核有 ARM11（2002 年发布）。v6 体系结构包含 ARM 体系结构中所有的 4 种特殊指令集：Thumb 指令（T）、DSP 指令（E）、Java 指令（J）和 Media 指令。

7．v7 架构

ARMv7 架构是在 ARMv6 架构的基础上诞生的。该架构采用了 Thumb-2 技术，它是在 ARM 的 Thumb 代码压缩技术的基础上发展起来的，并且保持了对现存 ARM 解决方案的完整的代码兼容性。Thumb-2 技术比纯 32 位代码少使用 31%的内存，减小了系统开销，同时能够提供比已有的基于 Thumb 技术的解决方案高出 38%的性能。ARMv7 架构还采用了 NEON 技术，将 DSP 和媒体处理能力提高了近 4 倍。并支持改良的浮点运算，满足下一代 3D 图形、游戏物理应用及传统嵌入式控制应用的需求。

Cortex 系列处理器是基于 ARMv7 架构的，分为 Cortcx-M3、Cortex-R 和 Cortex-A3 类。本章的 2.28 节将会例举一些 Cortex 的特性。

8．v8 架构

ARMv8 是在 32 位 ARM 架构上进行开发的，将被首先用于对扩展虚拟地址和 64 位数据处理技术有更高要求的产品领域，如企业应用、高档消费电子产品。ARMv8 架构包含两个执行状态：AArch64 和 AArch32。AArch64 执行状态针对 64 位处理技术，引入了一个全新指令集 A64，可以存取大虚拟地址空间；而 AArch32 执行状态将支持现有的 ARM 指令集。目前的 ARMv7 架构的主要特性都将在 ARMv8 架构中得以保留或进一步拓展，如 TrustZone 技术、虚拟化技术及 NEON advanced SIMD 技术等。

2.2 ARM 微处理器简介

ARM 处理器的产品系列非常广，包括 ARM7、ARM9、ARM9E、ARM10E、ARM11 和 SecurCore、Cortex 等。每个系列提供一套特定的性能来满足设计者对功耗、性能、体积的要求。SecurCore 是单独一个产品系列，是专门为安全设备而设计的。

表 2-2 总结了 ARM 各系列处理器所包含的不同类型。

表 2-2　ARM 各系列处理器所包含的不同类型

ARM 系列	包 含 类 型
ARM9/9E 系列	ARM920T ARM922T ARM926EJ-S ARM940T ARM946E-S ARM966E-S ARM968E-S
向量浮点运算（Vector Floating Point）系列	VFP9-S VFP10
ARM10E 系列	ARM1020E ARM1022E ARM1026EJ-S
ARM11 系列	ARM1136J-S ARM1136JF-S ARM1156T2(F)-S ARM1176JZ(F)-S ARM11 MPCore
Cortex 系列	Cortex-A Cortex-R Cortex-M
SecurCore 系列	SC100 SC110 SC200 SC210
其他合作伙伴产品	StrongARM XScale MBX

本节简要介绍 ARM 各个系列处理器的特点。

2.2.1　ARM9 处理器系列

ARM9 系列于 1997 年问世。由于采用了 5 级指令流水线，ARM9 处理器能够运行在比 ARM7 更高的时钟频率上，改善了处理器的整体性能；存储器系统根据哈佛体系结构（程序和数据空间独立的体系结构）重新设计，区分了数据总线和指令总线。

ARM9 系列的第一个处理器是 ARM920T，它包含独立的数据指令 Cache 和 MMU（Memory Management Unit，存储器管理单元）。此处理器能够用在要求有虚拟存储器支持的操作系统上。该系列中的 ARM922T 是 ARM920T 的变种，只有一半大小的数据指令 Cache。

ARM940T 包含一个更小的数据指令 Cache 和一个 MPU（Micro Processor Unit，微处理器）。它是针对不要求运行操作系统的应用而设计的。ARM920T、ARM940T 都执行 v4T 架构指令。

ARM9 系列处理器主要应用于下面一些场合：

（1）下一代无线设备，包括视频电话和 PDA 等。

（2）数字消费品，包括机顶盒、家庭网关、MP3 播放器和 MPEG-4 播放器。

（3）成像设备，包括打印机、数码照相机和数码摄像机。

（4）汽车、通信和信息系统。

2.2.2 ARM9E 处理器系列

ARM9 系列的下一代处理器基于 ARM9E-S 内核，这个内核是 ARM9 内核带有 E 扩展的一个可综合版本，包括 ARM946E-S 和 ARM966E-S 两个变种。两者都执行 v5TE 架构指令。它们也支持可选的嵌入式跟踪宏单元，支持开发者实时跟踪处理器指令和数据的执行。当调试对时间敏感的程序段时，这种方法非常重要。

ARM946E-S 包括 TCM（Tightly Coupled Memory，紧耦合存储器）、Cache 和一个 MPU。TCM 和 Cache 的大小可配置。该处理器是针对要求有确定的实时响应的嵌入式控制而设计的。ARM966E-S 有可配置的 TCM，但没有 MPU 和 Cache 扩展。

ARM9 系列的 ARM926EJ-S 内核为可综合的处理器内核，发布于 2000 年。它是针对小型便携式 Java 设备，如 3G 手机和 PDA 应用而设计的。ARM926EJ-S 是第一个包含 Jazelle 技术，可加速 Java 字节码执行的 ARM 处理器内核。它还有一个 MMU、可配置的 TCM 及具有零或非零等待存储器的数据/指令 Cache。

ARM9E 系列处理器主要应用于下面一些场合：

（1）下一代无线设备，包括视频电话和 PDA 等。

（2）数字消费品，包括机顶盒、家庭网关、MP3 播放器和 MPEG-4 播放器。

（3）成像设备，包括打印机、数码照相机和数码摄像机。

（4）存储设备，包括 DVD 或 HDD 等。

（5）工业控制，包括电机控制等。

（6）汽车、通信和信息系统的 ABS 和车体控制。

（7）网络设备，包括 VoIP、WirelessLAN 等。

2.2.3 ARM11 处理器系列

ARM1136J-S 发布于 2003 年，是针对高性能和高能效应而设计的。ARM1136J-S 是第一个执行 ARMv6 架构指令的处理器。它集成了一条具有独立的 Load/Stroe 和算术流水线

的 8 级流水线。ARMv6 指令包含了针对媒体处理的单指令流多数据流扩展，采用特殊的设计改善视频处理能力。

2.2.4　SecurCore 处理器系列

SecurCore 系列处理器提供了基于高性能的 32 位 RISC 技术的安全解决方案。SecurCore 系列处理器除了具有体积小、功耗低、代码密度高等特点外，还具有它自己的特别优势，即提供了安全解决方案支持。下面总结了 SecurCore 系列的主要特点：

（1）支持 ARM 指令集和 Thumb 指令集，以提高代码密度和系统性能。

（2）采用软内核技术以提供最大限度的灵活性，可以防止外部对其进行扫描探测。

（3）提供了安全特性，可以抵制攻击。

（4）提供面向智能卡和低成本的存储保护单元 MPU。

（5）可以集成用户自己的安全特性和其他的协处理器。

SecurCore 系列包含 SC100、SC110、SC200 和 SC210 四种类型。

SecurCore 系列处理器主要应用于一些安全产品及应用系统，包括电子商务、电子银行业务、网络、移动媒体和认证系统等。

2.2.5　StrongARM 和 Xscale 处理器系列

StrongARM 处理器最初是 ARM 公司与 Digital Semiconductor 公司合作开发的，现在由 Intel 公司单独许可，在低功耗、高性能的产品中应用很广泛。它采用哈佛架构，具有独立的数据和指令 Cache，有 MMU。StrongARM 是第一个包含 5 级流水线的高性能 ARM 处理器，但它不支持 Thumb 指令集。

Intel 公司的 Xscale 是 StrongARM 的后续产品，在性能上有显著改善。它执行 V5TE 架构指令，也采用哈佛结构，类似于 StrongARM 也包含一个 MMU。前面说过，Xscale 已经被 Intel 卖给了 Marvell 公司。

2.2.6　MPCore 处理器系列

MPCore 是在 ARM11 核心的基础上构建的，结构上仍属于 V6 指令体系。根据不同的需要，MPCore 可以被配置为 1 到 4 个处理器的组合方式，最高性能达到 2600 Dhrystone MIPS，运算能力几乎与 Pentium Ⅲ 1GHz 处于同一水准（Pentium Ⅲ 1GHz 的指令执行性能约为 2700 Dhrystone MIPS）。多核心设计的优点是在频率不变的情况下让处理器的性能获得明显提升，在多任务应用中表现尤其出色，这一点很适合未来家庭消费电子的需要。例如，机顶盒在录制多个频道电视节目的同时，还可通过互联网收看数字视频点播节目；车内导航系统在提供导航功能的同时，可以向后座乘客提供各类视频娱乐信息等。在这类应用环境下，多核心结构的嵌入式处理器将表现出极强的性能优势。

2.2.7 Cortex 处理器系列

1. ARM Cortex 处理器技术特点

ARMv7 架构是在 ARMv6 架构的基础上诞生的。该架构采用了 Thumb-2 技术，它是在 ARM 的 Thumb 代码压缩技术的基础上发展起来的，并且保持了对现存 ARM 解决方案的完整的代码兼容性。Thumb-2 技术比纯 32 位代码少使用 31%的内存，减小了系统开销，同时能够提供比已有的基于 Thumb 技术的解决方案高出 38%的性能。ARMv7 架构还采用了 NEON 技术，将 DSP 和媒体处理能力提高了近 4 倍。并支持改良的浮点运算，满足下一代 3D 图形、游戏物理应用及传统嵌入式控制应用的需求。此外，ARMv7 还支持改良的运行环境，以迎合不断增加的 JIT（Just In Time）和 DAC（Dynamic Adaptilve Compilation）技术的使用。

在与早期的 ARM 处理器软件兼容性方而，ARMv7 架构在设计时充分考虑到了。ARM Cortex-M 系列支持 Thumb-2 指令集（Thumb 指令集的扩展集），可以执行所有已存的为早期处理器编写的代码。通过一个前向的转换方式，为 ARM Cortex-M 系列处理器所写的用户代码可以与 ARM Cortex-R 系列微处理器完全兼容。ARM Cortex-M 系列系统代码（如实时操作系统）可以很容易地移植到基于 ARM Cortex-R 系列的系统上。ARMCortex-A 和 Cortex-R 系列处理器还支持 ARM 32 位指令集，向后完全兼容早期的 ARM 处理器，包括 1995 年发布的 ARM7TDMI 处理器，2002 年发布的 ARM11 处理器系列。由于应用领域的不同，基于 v7 架构的 Cortex 处理器系列所采用的技术也不相同。在命名方式上，基于 ARMv7 架构的 ARM 处理器已经不再延用过去的数字命名方式，而是冠以 Cortex 的代号。基于 v7A 的称为“Cortex-A 系州”，基于 v7R 的称为“Cortex-R 系列”，基于 v7M 的称为“Cortex-M3”。

2. ARM CorteX-M3 处理器技术特点

ARM Cortex-M3 处理器是为存储器和处理器的尺寸对产品成本影响极大的各种应用专门开发没计的。它整合了多种技术，减少了内存使用，并在极小的 RISC 内核上提供低功耗和高性能，可实现由以往的代码向 32 位微控制器的快速移植。ARM Cortex-M3 处理器是使用最少门数的 ARM CPU，相对于过去的设计大大减小了芯片面积，可减小装置的体积或采用更低成本的工艺进行生产，仅 33000 门的内核性能可达 l.2DMIPS/MHz。此外，基本系统外设还具备高度集成化特点，集成了许多紧耦合系统外设，合理利用了芯片空间，使系统满足下一代产品的控制需求。

ARM Cortex-M3 处理器结合了执行 Thumb-2 指令的 32 位哈佛微体系结构和系统外设，包括 Nested Vectored Interrupt Controller 和 Arbiter 总线。该技术方案在测试和实例应用中表现出较高的性能：在台机电 180 nm 工艺下，芯片性能达 1.2 DMIPS/MHz，时钟频率高达 100 MHz。Cortex-M3 处理器还实现了 Tail-Chaining 中断技术。该技术是一项完全基于硬件的中断处理技术，最多可减少 12 个时钟周期数，在实际应用中可减少 70%的中断；推出了新的单线调试技术，避免使用多引脚进行 JTAG 调试，并全面支持 RealVicw 编译器和 RealView 调试产品。Realview 工具向设计者提供模拟、创建虚拟模型、编译软件、调试、验证和测试基于 ARMv7 架构的系统等功能。

为微控制器应用而开发的 Cortex-M3 拥有以下性能：

- 实现单周期 Flash 应用最优化。
- 准确快速地中断处理。永不超过 12 周期，仅 6 周期 tail-chaining（末尾连锁）。
- 有低功耗时钟门控（Clock Gating）的 3 种睡眠模式。
- 单周期乘法和乘法累加指令。
- ARM Thumb-2 混合的 16/32 位固有指令集，无模式转换。
- 包括数据观察点和 Flash 补丁在内的高级调试功能。
- 原子位操作，在一个单一指令中读取/修改/编写。
- 1．25DMIPS/MHz（与 0．9DMIPS/MHz 的 ARM7 和 1．1DMIPS/MHz 的 ARM9 相比）。

3．ARM Cortex-R4 处理器技术特点

Cortex-R4 处理器支持手机、硬盘、打印机及汽车电子设计，能协助新一代嵌入式产品快速执行各种复杂的控制算法与实时工作的运算；可通过内存保护单元（Memory Protection Unit，MPU）、高速缓存及紧密耦合内存（Tightly Coupled Memory，TCM）让处理器针对各种不同的嵌入式应用进行最佳化调整，且不影响基本的 ARM 指令集兼容性。这种设计能够在沿用原有程序代码的情况下，降低系统的成本与复杂度，同时其紧密耦合内存功能也能提供更小的规格及更高效率的整合，并带来快速的响应时间。

Cortex-R4 处理器采用 ARMv7 体系结构，让它能与现有的程序维持完全的回溯兼容性，能支持现今在全球各地数十亿的系统，并已针对 Thumb-2 指令进行最佳化设计。此项特性带来很多的利益，其中包括：更低的时钟速度所带来的省电效益；更高的性能将各种多功能特色带入移动电话与汽车产品的设计；更复杂的算法支持更高性能的数码影像与内建硬盘的系统。运用 Thumb-2 指令集，加上 RealView 开发套件，使芯片内部存储器的容量最多降低 30%，大幅降低系统成本，其速度比在 ARM9tt6E-S 处理器所使用的 Thumb 指令集高出 40%。由于存储器在芯片中的占用空间愈来愈多，因此这项设计将大幅节省芯片容量，让芯片制造商运用这款处理器开发各种 SoC（System on a Chip）器件。

相比于前几代的处理器，Cortex-R4 处理器高效率的设计方案，使其能以更低的时钟达到更高的性能；经过最佳化设计的 Artisan Mctro 内存，可进一步降低嵌入式系统的体积与成本。处理器搭载一个先进的微架构，具备双指令发送功能，采用 90nm 工艺并搭配 Artisan Advantage 程序库的组件，底面积不到 lmm^2，耗电最低低于 0.27mW/MHz，并能提供超过 600 DMIPS 的性能。

Cortex-R4 处理器在各种安全应用上加入容错功能和内存保护机制，支持最新版 OSEK 实时操作系统；支持 RealView Develop 系列软件开发工具、RealView Create 系列 ESL 工具与模块，以及 Core Sight 除错与追踪技术，协助设计者迅速开发各种嵌入式系统。

4．ARM Cortex-A8 处理器技术特点

ARM Cortex-A8 处理器是一款适用于复杂操作系统及用户应用的应用处理器，支持智能能源管理（Intelligent Energy Manger，IEM）技术的 ARM Artisan 库及先进的泄漏控制技

术，使得 Cortex-A8 处理器实现了非凡的速度和功耗效率。在 65nm 工艺下，ARM Conex-A8 处理器的功耗不到 300mW，能够提供高性能和低功耗。它第一次为低费用、高容量的产品带来了台式机级别的性能。

Conex-A8 处理器是第一款基于下一代 ARMv7 架构的应用处理器，使用了能够带来更高性能、更低功耗和更高代码密度的 Thumb-2 技术。它首次采用了强大的 NEON 信号处理扩展集，为 H.264 和 MP3 等媒体编解码提供加速。Cortex-A8 的解决方案还包括 Jazelle-RCTJava 加速技术，对实时（JTT）和动态调整编译（DAC）提供最优化，同时减少内存占用空间高达 3 倍。该处理器配置了先进的超标量体系结构流水线，能够同时执行多条指令。处理器集成了一个可调尺寸的二级高速缓冲存储器，能够同高速的 16KB 或者 32KB 一级高速缓冲存储器一起工作，从而达到最快的读取速度和最大的吞吐量。新处理器还配置了用于安全交易和数字版权管理的 Trust Zone 技术，以及实现低功耗管理的 IEM 功能。

Cortex-A8 处理器使用了先进的分支预测技术，并且具有专用的 NEON 整型和浮点型流水线进行媒体和信号处理。在使用小于 $4mm^2$ 的硅片及低功耗的 65 nm 工艺的情况下，Cortex-A8 处理器的运行频率将高于 600MHz（不包括 NEON 追踪技术和二级高速缓冲存储器）。在高性能的 90nm 和 65nm 工艺下，Cortex-A8 处理器运行频率最高可达 1GHz，能够满足高性能消费产品设计的需要。

2.2.8 最新 ARM 应用处理器发展现状

（1）从之前的 ARM 单核逐步向双核演变。作为对比，下面依次将近年来最尖端的芯片应用方案列举出来。

- NVIDIA（英伟达）的 Tegra 2 双核处理器及 Tegra 3 四核处理器，已经应用在摩托罗拉双核智能手机 ME860 及 LG Optimus 2X 手机上。
- 三星 Exynos 4210，基于 CORTEX-A9 的双核处理器，目前应用在三星公司推出的 GALAXY SII 智能手机。
- TI 的 OMAP4430 及 OMAP4460 双核 ARM 处理芯片，已应用在 LG Optimus 3D 手机。
- 高通 MSM8260、MSM8660（1.5G）、MSM8960（1.7G）双核处理器及 APQ8060（2.5G）四核心处理器。目前应用的代表有 HTC 的金字塔（Pyramid）双核智能手机，还有国内的小米手机。
- 苹果 A5 双核处理器，典型代表是 iPhone4S 与 iPad2。

（2）内嵌的图形显示芯片越来越强劲。

- Mali 系列由 ARM 出品，Mali-400、Mali-T658 于 2011 年 11 月推出，支持 OpenGL ES 2.0 和 DirectX 接口，可从单核扩展到四核，可提供卓越的二维和三维加速性能。
- PowerVR SGX 系列由 Imagination Technologies 公司出品，包括 PowerVR SGX530/535/540/543MP，支持 DirectX 9、SM3.0 和 OpenGL 2.0。
- SGX535 被苹果公司的 iPhone4 和 iPad 采用，而 SGX540 性能更加强劲，在三星 Galaxy Tab 与魅族 M9 上采用。SGX543MP 作为新一代最强新品，目前已成为苹

果 iPad 2（SGX543MP2/双核）和索尼 NGP（SGX543MP4/四核）的图形内核。

- Adreno 系列由高通公司出品，主要配合 Snapdragon CPU 使用。旗下典型方案有 Adreno200/205/220/300。
- 在图形处理单元上，Tegra 3 从之前 Tegra 2 的 8 核心图形单元升级到 12 核心单元，NV5DIA 官方宣布将有 3 倍的图形性能提升。这 12 个处理核心的 GeForce GPU 专门为下一代移动游戏而打造（完全兼容现有 Tegra 2 游戏），支持更好的动态光影、物理效果和高分辨率环境。典型处理器方案有 NVIDIA Tegra 2 和 NVIDIA Tegra 3。

（3）支持大 RAM，支持大数据量的存储介质。

现在诸多处理器已支持 DDR2、DDR3、LPDDR（mDDR）等类型的内存。这些类型的内存高速度，高精度，并且容量也很高，已属于高速硬件之一。

（4）提升显示控制器性能。最高 2048×1536 分辨率液晶屏显示，如 Tegra 3 处理器。

（5）提升 Camera 性能。最高支持 3200 万像素摄像头

2.3　ARM 微处理器结构

ARM 内核采用 RISC 体系结构。ARM 体系结构的主要特征如下：

（1）采用大量的寄存器，它们都可以用于多种用途。

（2）采用 Load/Store 体系结构。

（3）每条指令都条件执行。

（4）采用多寄存器的 Load/Store 指令。

（5）能够在单时钟周期执行的单条指令内完成一项普通的移位操作和一项普通的 ALU 操作。

（6）通过协处理器指令集来扩展 ARM 指令集，包括在编程模式中增加了新的寄存器和数据类型。

（7）如果把 Thumb 指令集也当做 ARM 体系结构的一部分，那么在 Thumb 体系结构中还可以高密度 16 位压缩形式表示指令集。

2.4　ARM 微处理器的应用选型

随着国内嵌入式应用领域的发展，ARM 芯片必然会获得广泛的重视和应用。但是由于 ARM 芯片有多达十几种的芯核结构、70 多个芯片生产厂家及千变万化的内部功能配置组合，开发人员在选择方案时会有一定的困难。所以对 ARM 芯片做对比研究是十分必要的。

2.4.1 ARM 芯片选择的一般原则

1. 功能

考虑处理器本身能够支持的功能，如 USB、网络、串口、液晶显示功能等。

2. 性能

从处理器的功耗、速度、稳定可靠性等方面考虑。

3. 价格

通常产品总是希望在完成功能要求的基础上，成本越低越好。在选择处理器时需要考虑处理的价格，及由处理器衍生出的开发价格。如开发板价格、处理器自身价格、外围芯片、开发工具、制版价格等。

4. 熟悉程度及开发资源

通常公司对产品的开发周期都有严格的要求，选择一款自己熟悉的处理器可以大大降低开发风险。在自己熟悉的处理器都无法满足功能的情况下，可以尽量选择开发资源丰富的处理器。

5. 操作系统支持

在选择嵌入式处理器时，如果最终的程序需要运行在操作系统上，那么还应该考虑处理器对操作系统的支持。

6. 升级

很多产品在开发完成后都会面临升级的问题，正所谓人无远虑必有近忧。所以在选择处理器时必须要考虑升级的问题。如尽量选择具有相同封装的不同性能等级的处理器；考虑产品未来可能增加的功能。

7. 供货稳定

供货稳定也是选择处理器时的一个重要参考因素，尽量选择大厂家，比较通用的芯片。

2.4.2 选择一款适合 ARM 教学的 CPU

在 ARM 教学中，在选择 CPU 作为学习目标时，主要从芯片功能、开发平台价格、开发资源等方面考虑。

1. ARM 芯核

如果希望学习使用 Windows CE 或 Linux 等操作系统，就需要选择 ARM720T 以上带有 MMU（Memory Management Unit）功能的 ARM 芯片，ARM720T、StrongARM、Cortex-A 系列处理器都带有 MMU 功能。而 ARM7TDMI 没有 MMU，不支持 Windows CE 和大部分的 Linux。目前，uCLinux 及 Linux 2.6 内核等 Linux 系统不需要 MMU 的支持。

2．系统时钟速度

系统时钟决定了 ARM 芯片的处理速度。ARM7 的处理速度为 0.97MIPS/MHz，常见的 ARM7 芯片系统主时钟为 20～133MHz，ARM9 的处理速度为 1.1MIPS/MHz，常见的 ARM9 的系统主时钟为 100～233MHz。Cortex-A 系列的主时钟频率也越来越快，如 Cortex-A8 主频率可以达到 1.2GHz，如果希望学习可以支持较为复杂的操作系统的芯片时，可以选择 ARM9 及 ARM9 以上的芯片。

3．支持内存访问的类型

支持内存访问的类型如表 2-3 所示。

表 2-3　支持内存访问的类型

芯片名	是否有 SDRAM	是否有 DDR2	是否有 mDDR	是否有 DDR3
S3C2410	是	否	否	否
S3C2440	是	否	否	否
S5PC100	否	是	否	否
S5PV310	否	否	否	是

4．USB 接口

USB 接口产品的使用越来越广泛，许多 ARM 芯片内置 USB 控制器，有些芯片甚至同时有 USB Host 和 USB Slave 控制器。表 2-4 显示了内置 USB 控制器的 ARM 芯片。

表 2-4　内置 USB 控制器的 ARM 芯片

芯 片 型 号	ARM 内核	供　应　商	USB （otg）	USB Host
S3C2410	ARM920T	SAMSUNG	1	2
S3C2440	ARM920T	SAMSUNG	1	2
S5PC100	CORTEX-A8	SAMSUNG	1	1
S5PV310	CORTEX-A9	SAMSUNG	1	1

5．GPIO 数量

在某些芯片供应商提供的说明书中，往往申明的是最大可能的 GPIO 数量，但是有许多引脚是和地址线、数据线、串口线等引脚复用的。这样在系统设计时需要计算实际可以使用的 GPIO 数量。

6．中断控制器

ARM 内核只提供快速中断（FIQ）和标准中断（IRQ）两个中断向量。但各个半导体厂家在设计芯片时加入了自己定义的中断控制器，以便支持诸如串行口、外部中断、时钟中断等硬件中断。外部中断控制是选择芯片时必须考虑的重要因素，合理的外部中断设计可以很大程度地减少任务调度工作量。例如 PHILIPS 公司的 SAA7750，所有 GPIO 都可以设置成 FIQ 或 IRQ，并且可以选择上升沿、下降沿、高电平和低电平 4 种中断方式。这使

得红外线遥控接收、指轮盘和键盘等任务都可以作为背景程序运行。而 Cirrus Logic 公司的 EP7312 芯片只有 4 个外部中断源，并且每个中断源都只能是低电平或高电平中断，这样接收红外线信号的场合必须用查询方式，浪费大量 CPU 时间。

7．IIS（Integrate Interface of Sound）接口

IIS 接口即集成音频接口。如果设计音频应用产品，IIS 接口是必需的。

8．nWAIT 信号

这是一个外部总线速度控制信号。不是每个 ARM 芯片都提供这个信号引脚，利用这个信号与廉价的 GAL 芯片就可以实现符合 PCMCIA 标准的 WLAN 卡和 BlueTooth 卡的接口，而不需要外加高成本的 PCMCIA 专用控制芯片。另外，当需要扩展外部 DSP 协处理器时，此信号也是必需的。

9．RTC（Real Time Clock）

很多 ARM 芯片都提供 RTC（实时时钟）功能，但方式不同。如 Cirrus Logic 公司的 EP7312 的 RTC 只是一个 32 位计数器，需要通过软件计算出年月日时分秒；而 SAA7750 和 S3C2410 等芯片的 RTC 直接提供年月日时分秒格式。

10．LCD 控制器

有些 ARM 芯片内置 LCD 控制器，有的甚至内置 64KB 彩色 TFT LCD 控制器。在设计 PDA 和手持式显示记录设备时，选用内置 LCD 控制器的 ARM 芯片（如 S3C2410）较为适宜。

11．PWM 输出

有些 ARM 芯片有 2～8 路 PWM 输出，可以用于电机控制或语音输出等场合。

12．ADC 和 DAC

有些 ARM 芯片内置 2～8 通道 8～12 位通用 ADC，可以用于电池检测、触摸屏和温度监测等。PHILIPS 的 SAA7750 更是内置了一个 16 位立体声音频 ADC 和 DAC，并且带耳机驱动。

13．扩展总线

大部分 ARM 芯片具有外部 SDRAM 和 SRAM 扩展接口，不同的 ARM 芯片可以扩展的芯片数量即片选线数量不同，外部数据总线有 8 位、16 位或 32 位。为某些特殊应用设计的 ARM 芯片（如德国 Micronas 的 PUC3030A）没有外部扩展功能。

14．UART 和 IrDA

几乎所有的 ARM 芯片都具有 1～2 个 UART 接口，可以用于和 PC 通信或用 Angel 进行调试。一般的 ARM 芯片通信波特率为 115200bit/s，少数专为蓝牙技术应用设计的 ARM 芯片的 UART 通信波特率可以达到 920kbit/s，如 Linkup 公司 L7205。

15．时钟计数器和看门狗

一般 ARM 芯片都具有 2～4 个 16 位或 32 位时钟计数器和一个看门狗计数器。

16．电源管理功能

ARM 芯片的耗电量与工作频率成正比，一般 ARM 芯片都有低功耗模式、睡眠模式和关闭模式。

17．DMA 控制器

有些 ARM 芯片内部集成 DMA（Direct Memory Access）接口，可以和硬盘等外部设备高速交换数据，同时减少数据交换时对 CPU 资源的占用。

另外，可以选择的内部功能部件还有 HDLC、SDLC、CD-ROM Decoder、Ethernet MAC、VGA controller 和 DC-DC。可以选择的内置接口有：IIC、SPDIF、CAN、SPI、PCI 和 PCMCIA。

18．封装类型

最后需说明的是封装问题。ARM 芯片现在主要的封装有 QFP、TQFP、PQFP、LQFP、BGA、LBGA 等形式，BGA 封装具有芯片面积小的特点，可以减少 PCB 的面积，但是需要专用的焊接设备，无法手工焊接。另外，一般 BGA 封装的 ARM 芯片无法用双面板完成 PCB 布线，需要多层 PCB 板布线。

最后，根据大专、高职院校的实际情况结合当前及未来一段时间的市场人才需求，经过综合考虑，本书教学选取的是三星公司的 S5PC100 芯片。S5PC100 是一款基于 Cortex-A8 核心的微处理器芯片。本章的后面部分章节将对 Cortex-A8 的一些特性及 S5PC100 进行详细介绍。

2.5　Cortex-A8 内部功能及特点

Cortex-A8 处理器是一款高性能、低功耗的处理器核心，并支持 Cache、虚拟存取，它的特性如下：

- 完全执行 v7-A 体系指令集。
- 可配置 64 位或 128 位 AMBA 高速总线接口 AXI。
- 具有一个集成的整形流水线。
- 具有一个 NEON 技术下执行 SIMD/VFP 的流水线。
- 支持动态分支预取，全局历史缓存，8 入口返回栈。
- 具有独立的数据/指令 MMU。
- 16KB/32KB 可配置 1 级 Cache。
- 具有带奇偶校验及 ECC 校验的 2 级 Cache。
- 支持 ETM 的非侵入式调试。
- 具有静态/动态电源管理功能。

ARMv7 体系指令集方面表现如下特点：

- 支持 ARN Thumb-2 高密度指令集。
- 使用 ThumbEE，执行环境加速。
- 安全扩展体系加强了安全应用的可靠性。
- 先进的 SIMD 体系技术用于加速多媒体应用。
- 支持 VFP 第三代向量浮点运算。

2.6 数据类型

2.6.1 ARM 的基本数据类型

ARM 采用的是 32 位架构，ARM 的基本数据类型有以下 3 种。

- Byte：字节，8bit。
- Halfword：半字，16bit（半字必须与 2 字节边界对齐）。
- Word：字，32bit（字必须与 4 字节边界对齐）。

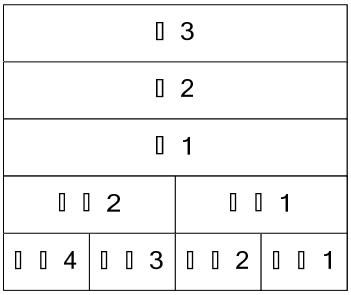

图 2-1 ARM 存储器组织结构

存储器可以看做是序号为 0～2^{32}−1 的线性字节阵列。如图 2-1 所示为 ARM 存储器的组织结构。其中每一个字节都有唯一的地址。字节可以占用任一位置，图中给出了几个例子。长度为 1 个字的数据项占用一组 4 字节的位置，该位置开始于 4 的倍数的字节地址（地址最末两位为 00）。半字占有两个字节的位置，该位置开始于偶数字节地址（地址最末一位为 0）。

注意

（1）ARM 系统结构 v4 以上版本支持以上 3 种数据类型，v4 以前版本仅支持字节和字。

（2）当将这些数据类型中的任意一种声明成 unsigned 类型时，n 位数据值表示范围为 0～2^n−1 的非负数，通常使用二进制格式。

（3）当将这些数据类型的任意一种声明成 signed 类型时，n 位数据值表示范围为−2^{n-1}～2^{n-1}−1 的整数，使用二进制的补码格式。

（4）所有数据类型指令的操作数都是字类型的，如“ADD r1，r0，＃0x1”中的操作数“0x1”就是以字类型数据处理的。

（5）Load/Store 数据传输指令可以从存储器存取传输数据，这些数据可以是字节、半字、字。加载时自动进行字节或半字的零扩展或符号扩展。对应的指令分别为 LDR/BSTRB（字节操作）、LDRH/STRH（半字操作）、LDR/STR（字操作）。详见后面的指令参考。

（6）ARM 指令编译后是 4 个字节（与字边界对齐）。Thumb 指令编译后是 2 个字节（与半字边界对齐）。

2.6.2　浮点数据类型

浮点运算使用在 ARM 硬件指令集中未定义的数据类型。尽管如此，但 ARM 公司在协处理器指令空间定义了一系列浮点指令。通常这些指令全部可以通过未定义指令异常（此异常收集所有硬件协处理器不接受的协处理器指令）在软件中实现，但是其中的一小部分也可以由浮点运算协处理器 FPA10 以硬件方式实现。另外，ARM 公司还提供了用 C 语言编写的浮点库作为 ARM 浮点指令集的替代方法（Thumb 代码只能使用浮点指令集）。该库支持 IEEE 标准的单精度和双精度格式。C 编译器有一个关键字标志来选择这个历程。它产生的代码与软件仿真（通过避免中断、译码和浮点指令仿真）相比既快又紧凑。

2.6.3　存储器大/小端

从软件角度看，内存相对于一个大的字节数组，其中每个数组元素（字节）都是可寻址的。

ARM 支持大端模式（big-endian）和小端模式（little-endian）两种内存模式。

如图 2-2 所示显示了大端模式和小端模式数据存放特点。

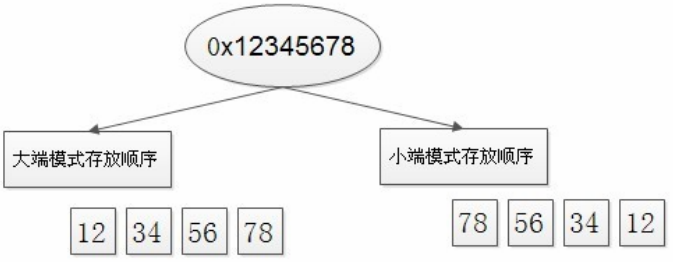

图 2-2　大小端模式存放数据的特点

下面的例子显示了使用内存大/小端（big/little endian）的存取格式。

程序执行前：

```
r0=0x11223344
```

执行指令：

```
r1=0x100
STR r0, [r1]
LDRB r2, [r1]
```

执行后：

```
小端模式下：r2=0x44
大端模式下：r2=0x11
```

上面的例子向我们提示了一个潜在的编程隐患。在大端模式下，一个字的高地址放的是数据的低位，而在小端模式下，数据的低位放在内存中的低地址。要小心对待存储器中一个字内字节的顺序。

2.7 Cortex-A8 内核工作模式

Cortex-A8 基于 ARMv7-A 架构，共有 8 种工作模式，如表 2-5 所示。

表 2-5 S5PC100 处理器的工作模式

处理器工作模式	简 写	描 述
用户模式（User）	usr	正常程序执行模式，大部分任务执行在这种模式下
快速中断模式（FIQ）	fiq	当一个高优先级（fast）中断产生时将会进入这种模式，一般用于高速数据传输和通道处理
外部中断模式（IRQ）	irq	当一个低优先级（normal）中断产生时将会进入这种模式，一般用于通常的中断处理
特权模式（Supervisor）	svc	当复位或软中断指令执行时进入这种模式，是一种供操作系统使用的保护模式
数据访问中止模式（Abort）	abt	当存取异常时将会进入这种模式，用于虚拟存储或存储保护
未定义指令中止模式（Undef）	und	当执行未定义指令时进入这种模式，有时用于通过软件仿真协处理器硬件的工作方式
系统模式（System）	sys	使用和 User 模式相同寄存器集的模式，用于运行特权级操作系统任务
监控模式（Monitor）	mon	可以在安全模式与非安全模式之间进行转换

除用户模式外的其他 7 种处理器模式称为特权模式（Privileged Modes）。在特权模式下，程序可以访问所有的系统资源，也可以任意地进行处理器模式切换。其中以下 6 种又称为异常模式：

（1）快速中断模式（FIQ）。

（2）外部中断模式（IRQ）。

（3）特权模式（Supervior）。

（4）数据访问中止模式（Abort）。

（5）未定义指令中止模式（Undef）。

（6）监控模式（Monitor）。

处理器模式可以通过软件控制进行切换，也可以通过外部中断或异常处理过程进行切换。

大多数的用户程序运行在用户模式下。当处理器工作在用户模式时，应用程序不能够访问受操作系统保护的一些系统资源，应用程序也不能直接进行处理器模式切换。当需要进行处理器模式切换时，应用程序可以产生异常处理，在异常处理过程中进行处理器模式切换。这种体系结构可以使操作系统控制整个系统资源的使用。

当应用程序发生异常中断时，处理器进入相应的异常模式。在每一种异常模式中都有一组专用寄存器以供相应的异常处理程序使用，这样就可以保证在进入异常模式时用户模式下的寄存器（保存程序运行状态）不被破坏。

2.8　Cortex-A8 存储系统

ARM 存储系统有非常灵活的体系结构，可以适应不同的嵌入式应用系统的需要。ARM 存储器系统可以使用简单的平板式地址映射机制（就像一些简单的单片机一样，地址空间的分配方式是固定的，系统中各部分都使用物理地址），也可以使用其他技术提供功能更为强大的存储系统。例如：

（1）系统可能提供多种类型的存储器件，如 Flash、ROM、SRAM 等。

（2）Cache 技术。

（3）写缓存技术（Write Buffers）。

（4）虚拟内存和 I/O 地址映射技术。

大多数的系统通过下面的方法之一可实现对复杂存储系统的管理。

（1）使用 Cache，缩小处理器和存储系统速度差别，从而提高系统的整体性能。

（2）使用内存映射技术实现虚拟空间到物理空间的映射。这种映射机制对嵌入式系统非常重要。通常嵌入式系统程序存放在 ROM/Flash 中，这样系统断电后程序能够得到保存。但是，通常 ROM/Flash 与 SDRAM 相比，速度慢很多，而且基于 ARM 的嵌入式系统中通常把异常中断向量表放在 RAM 中。利用内存映射机制可以满足这种需要。在系统加电时，将 ROM/Flash 映射为地址 0，这样可以进行一些初始化处理；当这些初始化处理完成后将 SDRAM 映射为地址 0，并把系统程序加载到 SDRAM 中运行，这样可很好地满足嵌入式系统的需要。

（3）引入存储保护机制，增强系统的安全性。

（4）引入一些机制保证将 I/O 操作映射成内存操作后，各种 I/O 操作能够得到正确的结果。在简单存储系统中，不存在这样的问题。而当系统引入了 Cache 和 write buffer 后，就需要一些特别的措施。

在 ARM 系统中，要实现对存储系统的管理通常使用协处理器 CP15，它通常也被称为系统控制协处理器（System Control Coprocessor）。

ARM 的存储器系统是由多级构成的，可以分为内核级、芯片级、板卡级、外设级。如图 2-3 所示为存储器的层次结构。

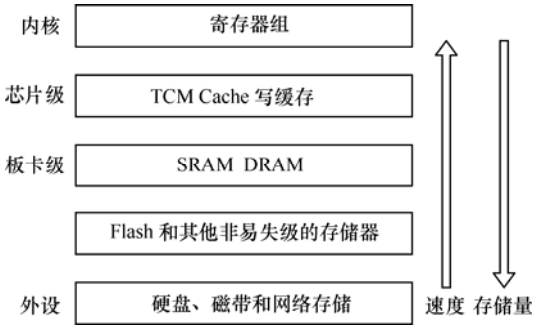

图 2-3　存储器的层次结构

每级都有特定的存储介质，下面对比各级系统中特定存储介质的存储性能。

（1）内核级的寄存器。处理器寄存器组可看做是存储器层次的顶层。这些寄存器被集成在处理器内核中，在系统中提供最快的存储器访问。典型的 ARM 处理器有多个 32 位寄存器，其访问时间为 ns 量级。

（2）芯片级的紧耦合存储器（TCM）是为弥补 Cache 访问的不确定性增加的存储器。TCM 是一种快速 SDRAM，它紧挨内核，并且保证取指和数据操作的时钟周期数，这一点对一些要求确定行为的实时算法是很重要的。TCM 位于存储器地址映射中，可作为快速存储器来访问。

（3）芯片级的片上 Cache 存储器的容量在 8KB～32KB 之间，访问时间大约为 10ns。高性能的 ARM 结构中，可能存在第二级片外 Cache，容量为几百 KB，访问时间为几十 ns。

（4）板卡级的 DRAM。主存储器可能是几 MB 到几十 MB 的动态存储器，访问时间大约为 100ns。

（5）外设级的后援存储器，通常是硬盘，可能从几百 MB 到几个 GB，访问时间为几十 ms。

2.8.1 协处理器（CP15）

ARM 处理器支持 16 个协处理器。在程序执行过程中，每个协处理器忽略属于 ARM 处理器和其他协处理器的指令。当一个协处理器硬件不能执行属于它的协处理器指令时，将产生一个未定义指令异常中断，在该异常中断处理程序中，可以通过软件模拟该硬件操作。例如，如果系统不包含向量浮点运算器，则可以选择浮点运算软件模拟包来支持向量浮点运算。CP15 即通常所说的系统控制协处理器（System Control Coprocessor），它负责完成大部分的存储系统管理。除了 CP15 外，在具体的各种存储管理机制中可能还会用到其他一些技术，如在 MMU 中除了 CP15 外，还使用了页表技术等。

在一些没有标准存储管理的系统中，CP15 是不存在的。在这种情况下，针对 CP15 的操作指令将被视为未定义指令，指令的执行结果不可预知。

CP15 包含 16 个 32 位寄存器，其编号为 0～15。实际上对于某些编号的寄存器可能对应多个物理寄存器，在指令中指定特定的标志位来区分这些物理寄存器。这种机制有些类似于 ARM 中的寄存器，当处于不同的处理器模式时，某些相同编号的寄存器对应于不同的物理寄存器。

CP15 中的寄存器可能是只读的，也可能是只写的，还有一些是可读/可写的。在对协处理器寄存器进行操作时，需要注意以下几个问题：

（1）寄存器的访问类型（只读/只写/可读可写）。

（2）不同的访问引发不同的功能。

（3）相同编号的寄存器是否对应不同的物理寄存器。

（4）寄存器的具体作用。

2.8.2　存储管理单元（MMU）

在创建多任务嵌入式系统时，最好用一个简单的方式来编写、装载及运行各自独立的任务。目前大多数的嵌入式系统不再使用自己定制的控制系统，而使用操作系统来简化这个过程。较高级的操作系统采用基于硬件的存储管理单元（MMU）来实现上述操作。

MMU 提供的一个关键服务是使各个任务作为各自独立的程序在自己的私有存储空间中运行。在带 MMU 的操作系统控制下，运行的任务无须知道其他与之无关的任务的存储需求情况，这就简化了各个任务的设计。

MMU 提供了一些资源以允许使用虚拟存储器（将系统物理存储器重新编址，可将其看成一个独立于系统物理存储器的存储空间）。MMU 作为转换器，将程序和数据的虚拟地址（编译时的连接地址）转换成实际的物理地址，即在物理主存中的地址。这个转换过程允许运行的多个程序使用相同的虚拟地址，而各自存储在物理存储器的不同位置。

这样存储器就有两种类型的地址：虚拟地址和物理地址。虚拟地址由编译器和连接器在定位程序时分配；物理地址用来访问实际的主存硬件模块（物理上程序存在的区域）。

2.8.3　高速缓冲存储器（Cache）

Cache 是一个容量小但存取速度非常快的存储器，它保存最近用到的存储器数据副本。对于程序员来说，Cache 是透明的。它自动决定保存哪些数据、覆盖哪些数据。现在 Cache 通常与处理器在同一芯片上实现。Cache 能够发挥作用是因为程序具有局部性。所谓局部性就是指在任何特定的时间，处理器趋于对相同区域的数据（如堆栈）多次执行相同的指令（如循环）。

Cache 经常与写缓存器（write buffer）一起使用。写缓存器是一个非常小的先进先出（FIFO）存储器，位于处理器核与主存之间。使用写缓存的目的是，将处理器核和 Cache 从较慢的主存写操作中解脱出来。当 CPU 向主存储器做写入操作时，它先将数据写入到写缓存区中，由于写缓存器的速度很高，这种写入操作的速度也将很高。写缓存区在 CPU 空闲时，以较低的速度将数据写入到主存储器中相应的位置。

通过引入 Cache 和写缓存区，存储系统的性能得到了很大的提高，但同时也带来了一些问题。例如，由于数据将存在于系统中不同的物理位置，可能造成数据的不一致性；由于写缓存区的优化作用，可能有些写操作的执行顺序不是用户期望的顺序，从而造成操作错误。

2.9　流水线

2.9.1　流水线的概念与原理

处理器按照一系列步骤来执行每一条指令，典型的步骤如下：

（1）从存储器读取指令（fetch）。

（2）译码以鉴别它属于哪一条指令（decode）。

（3）从指令中提取指令的操作数（这些操作数往往存在于寄存器 reg 中）。

（4）将操作数进行组合以得到结果或存储器地址（ALU）。

（5）如果需要，则访问存储器以存储数据（mem）。

（6）将结果写回到寄存器堆（res）。

并不是所有的指令都需要上述每一个步骤，但是，多数指令需要其中的多个步骤。这些步骤往往使用不同的硬件功能，如 ALU 可能只在第 4 步中用到。因此，如果一条指令不是在前一条指令结束之前就开始，那么在每一步骤内处理器只有少部分的硬件在使用。

有一种方法可以明显改善硬件资源的使用率和处理器的吞吐量，这就是在当前一条指令结束之前就开始执行下一条指令，即通常所说的流水线（Pipeline）技术。流水线是 RISC 处理器执行指令时采用的机制。使用流水线，可在取下一条指令的同时译码和执行其他指令，从而加快执行的速度。可以把流水线看做是汽车生产线，每个阶段只完成专门的处理器任务。

采用上述操作顺序，处理器可以这样来组织：当一条指令刚刚执行完步骤（1）并转向步骤（2）时，下一条指令就开始执行步骤（1）。从原理上说，这样的流水线应该比没有重叠的指令执行快 6 倍，但由于硬件结构本身的一些限制，实际情况会比理想状态差一些。

2.9.2 流水线的分类

1. 3 级流水线 ARM 组织

到 ARM7 为止的 ARM 处理器使用简单的 3 级流水线，它包括下列流水线级。

（1）取指令（fetch）：从寄存器装载一条指令。

（2）译码（decode）：识别被执行的指令，并为下一个周期准备数据通路的控制信号。在这一级，指令占有译码逻辑，不占用数据通路。

（3）执行（excute）：处理指令并将结果写回寄存器。

如图 2-4 所示为 3 级流水线指令的执行过程。

图 2-4　3 级流水线

当处理器执行简单的数据处理指令时，流水线使得平均每个时钟周期能完成 1 条指令。但 1 条指令需要 3 个时钟周期来完成，因此，有 3 个时钟周期的延时（latency），但吞吐率（throughput）是每个周期 1 条指令。

2．5 级流水线 ARM 组织

所有的处理器都要满足对高性能的要求，直到 ARM7 为止，在 ARM 核中使用的 3 级流水线的性价比是很高的。但是，为了得到更高的性能，需要重新考虑处理器的组织结构。有两种方法来提高性能。

（1）提高时钟频率。时钟频率的提高，必然引起指令执行周期的缩短，所以要求简化流水线每一级的逻辑，流水线的级数就要增加。

（2）减少每条指令的平均指令周期数 CPI。这就要求重新考虑 3 级流水线 ARM 中多于 1 个流水线周期的实现方法，以便使其占有较少的周期，或者减少因指令相关造成的流水线停顿，也可以将两者结合起来。

3 级流水线 ARM 核在每一个时钟周期都访问存储器，或者取指令，或者传输数据。只是抓紧存储器不用的几个周期来改善系统性能，效果并不明显。为了改善 CPI，存储器系统必须在每个时钟周期中给出多于一个的数据。方法是在每个时钟周期从单个存储器中给出多于 32 位数据，或者为指令或数据分别设置存储器。

基于以上原因，较高性能的 ARM 核使用了 5 级流水线，而且具有分开的指令和数据存储器。把指令的执行分割为 5 部分而不是 3 部分，进而可以使用更高的时钟频率，分开的指令和数据存储器使核的 CPI 明显减少。

在 ARM9TDMI 中使用了典型的 5 级流水线，5 级流水线包括下面的流水线级。

（1）取指令（fetch）：从存储器中取出指令，并将其放入指令流水线。

（2）译码（decode）：指令被译码，从寄存器堆中读取寄存器操作数。在寄存器堆中有 3 个操作数读端口，因此，大多数 ARM 指令能在 1 个周期内读取其操作数。

（3）执行（execute）：将其中 1 个操作数移位，并在 ALU 中产生结果。如果指令是 Load 或 Store 指令，则在 ALU 中计算存储器的地址。

（4）缓冲/数据（buffer/data）：如果需要则访问数据存储器，否则 ALU 只是简单地缓冲 1 个时钟周期。

（5）回写（write-back）：将指令的结果回写到寄存器堆，包括任何从寄存器读出的数据。

如图 2-5 所示列出了 5 级流水线指令的执行过程。

图 2-5　5 级流水线

在程序执行过程中，PC 值是基于 3 级流水线操作特性的。5 级流水线中提前 1 级来读取指令操作数，得到的值是不同的（PC + 4 而不是 PC + 8）。这里产生代码不兼容是不容许的。但 5 级流水线 ARM 完全仿真 3 级流水线的行为。在取指级增加的 PC 值被直接送到译码级的寄存器，穿过两级之间的流水线寄存器。下一条指令的 PC + 4 等于当前指令的 PC + 8，因此，未使用额外的硬件便得到了正确的 R15。

3．13 级流水线

在 Cortex-A8 中有一条 13 级的流水线，但是由于 ARM 公司没有对其中的技术公开任何相关的细节，这里只能简单介绍一下，从经典 ARM 系列到现在的 Cortex 系列，ARM 处理器的结构在向复杂的阶段发展，但没改变的是 CPU 的取指指令和地址关系，不管是几级流水线，都可以按照最初的 3 级流水线的操作特性来判断其当前的 PC 位置。这样做主要还是为了软件兼容性上的考虑，由此可以判断的是，后面 ARM 所推出的处理核心都想满足这一特点，感兴趣的读者可以自行查阅相关资料。

2.9.3 影响流水线性能的因素

1．互锁

在典型的程序处理过程中，经常会遇到这样的情形，即一条指令的结果被用做下一条指令的操作数。例如，有如下指令序列：

```
LDR R0,[R0,#0]
ADD R0,R0,R1     ;在 5 级流水线上产生互锁
```

从例子可以看出，流水线的操作产生中断，因为第 1 条指令的结果在第 2 条指令取数时还没有产生。第 2 条指令必须停止，直到结果产生为止。

2．跳转指令

跳转指令也会破坏流水线的行为，因为后续指令的取指步骤受到跳转目标计算的影响，因而必须推迟。但是，当跳转指令被译码时，在它被确认是跳转指令之前，后续的取指操作已经发生。这样一来，已经被预取进入流水线的指令不得不被丢弃。如果跳转目标的计算是在 ALU 阶段完成的，那么在得到跳转目标之前已经有两条指令按原有指令流读取。

显然，只有当所有指令都依照相似的步骤执行时，流水线的效率达到最高。如果处理器的指令非常复杂，每一条指令的行为都与下一条指令不同，那么就很难用流水线实现。

2.10 寄存器组织

ARM 处理器有如下 40 个 32 位长的寄存器。

（1）33 个通用寄存器。

（2）6 个状态寄存器：1 个 CPSR（Current Program Status Register，当前程序状态寄存器），6 个 SPSR（Saved Program Status Register，备份程序状态寄存器）。

（3）1 个 PC（Program Counter，程序计数器）。

ARM 处理器共有 7 种不同的处理器模式，在每一种处理器模式中都有一组相应的寄存器组，如图 2-6 所示列出了 ARM 处理器的寄存器组织概要。

图 2-6　寄存器列表

当前处理器的模式决定着哪组寄存器可操作，任何模式都可以存取下列寄存器。

（1）相应的 R0～R12。

（2）相应的 R13（Stack Pointer，SP，栈指向）和 R14（the Link Register，LR，链路寄存器）。

（3）相应的 R15（PC）。

（4）相应的 CPSR。

特权模式（除 System 模式外）还可以存取相应的 SPSR。

通用寄存器根据其分组与否可分为以下两类。

（1）未分组寄存器（Unbanked Register），包括 R0～R7。

（2）分组寄存器（Banked Register），包括 R8～R14。

1．未分组寄存器

未分组寄存器包括 R0～R7。顾名思义，在所有处理器模式下对于每一个未分组寄存器来说，指的都是同一个物理寄存器。未分组寄存器没有被系统用于特殊的用途，任何可采用通用寄存器的应用场合都可以使用未分组寄存器。但由于其通用性，在异常中断所引起的处理器模式切换时，其使用的是相同的物理寄存器，所以也就很容易使寄存器中的数据被破坏。

2．分组寄存器

R8～R14 是分组寄存器，它们每一个访问的物理寄存器取决于当前的处理器模式。

对于分组寄存器 R8～R12 来说，每个寄存器对应两个不同的物理寄存器。一组用于除 FIQ 模式外的所有处理器模式，而另一组则专门用于 FIQ 模式。这样的结构设计有利于加快 FIQ 的处理速度。不同模式下寄存器的使用，要使用寄存器名后缀加以区分。例如，当使用 FIQ 模式下的寄存器时，寄存器 R8 和寄存器 R9 分别记为 R8_fiq、R9_fiq；当使用用户模式下的寄存器时，寄存器 R8 和 R9 分别记为 R8_usr、R9_usr 等。在 ARM 体系结构中，R8～R12 没有任何指定的其他的用途，所以当 FIQ 中断到达时，不用保存这些通用寄存器，也就是说，FIQ 处理程序可以不必执行保存和恢复中断现场的指令，从而可以使中断处理过程非常迅速。所以 FIQ 模式常被用来处理一些时间紧急的任务，如 DMA 处理。

对于分组寄存器 R13 和 R14 来说，每个寄存器对应 6 个不同的物理寄存器。其中的一个是用户模式和系统模式公用的，而另外 5 个分别用于 5 种异常模式。访问时需要指定它们的模式。名字形式如下：

（1）R13_<mode>

（2）R14_<mode>

其中，<mode>可以是以下几种模式之一：usr、svc、abt、und、irp、fiq 及 mon。

R13 寄存器在 ARM 处理器中常用做堆栈指针，称为 SP。当然，这只是一种习惯用法，并没有任何指令强制性的使用 R13 作为堆栈指针，用户完全可以使用其他寄存器作为堆栈指针。而在 Thumb 指令集中，有一些指令强制性地将 R13 作为堆栈指针，如堆栈操作指令。

每一种异常模式拥有自己的 R13。异常处理程序负责初始化自己的 R13，使其指向该异常模式专用的栈地址。在异常处理程序入口处，将用到的其他寄存器的值保存在堆栈中，返回时，重新将这些值加载到寄存器。通过这种保护程序现场的方法，异常不会破坏被其中断的程序现场。

寄存器 R14 又被称为连接寄存器（Link Register，LR），在 ARM 体系结构中具有下面两种特殊的作用。

（1）每一种处理器模式用自己的 R14 存放当前子程序的返回地址。当通过 BL 或 BLX 指令调用子程序时，R14 被设置成该子程序的返回地址。在子程序返回时，把 R14 的值复制到程序计数器（PC）。典型的做法是使用下列两种方法之一。

① 执行下面任何一条指令。

```
MOV  PC, LR
BX  LR
```

② 在子程序入口处使用下面的指令将 PC 保存到堆栈中。

```
STMFD  SP!, {<register>,LR}
```

在子程序返回时，使用如下相应的配套指令返回。

```
LDMFD  SP!, {<register>,PC}
```

（2）当异常中断发生时，该异常模式特定的物理寄存器 R14 被设置成该异常模式的返回地址，对于有些模式 R14 的值可能与返回地址有一个常数的偏移量（如数据异常使用 SUB PC，LR，#8 返回）。具体的返回方式与上面的子程序返回方式基本相同，但使用的指令稍微有些不同，以保证当异常出现时正在执行的程序的状态被完整保存。

R14 也可以被用做通用寄存器使用。

2.11　程序状态寄存器

当前程序状态寄存器（Current Program Status Register，CPSR）可以在任何处理器模式下被访问，它包含下列内容：

（1）ALU（Arithmetic Logic Unit，算术逻辑单元）状态标志的备份。

（2）当前的处理器模式。

（3）中断使能标志。

（4）设置处理器的状态。

每一种处理器模式下都有一个专用的物理寄存器做备份程序状态寄存器（Saved Program Status Register，SPSR）。当特定的异常中断发生时，这个物理寄存器负责存放当前程序状态寄存器的内容。当异常处理程序返回时，再将其内容恢复到当前程序状态寄存器。

CPSR 寄存器（和保存它的 SPSR 寄存器）中的位分配如图 2-7 所示。

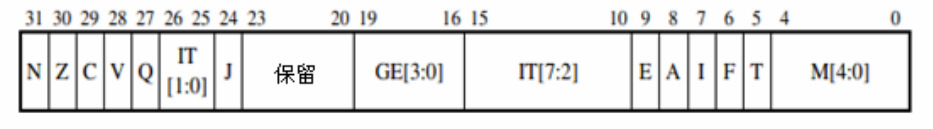

图 2-7　程序状态寄存器格式

下面给出各个状态位的定义。

1．标志位

N（Negative）、Z（Zero）、C（Carry）和 V（oVerflow）通称为条件标志位。这些条件标志位会根据程序中的算术指令或逻辑指令的执行结果进行修改，而且这些条件标志位可由大多数指令检测以决定指令是否执行。

在 ARM 4T 架构中，所有的 ARM 指令都可以条件执行，而 Thumb 指令却不能。

各条件标志位的具体含义如下。

1）N

本位设置成当前指令运行结果的 bit[31]的值。当两个由补码表示的有符号整数运算时，N = 1 表示运算的结果为负数，N = 0 表示结果为正数或零。

2）Z

Z = 1 表示运算的结果为零，Z = 0 表示运算的结果不为零。

3）C

下面分 4 种情况讨论 C 的设置方法。

（1）在加法指令中（包括比较指令 CMN），当结果产生了进位，则 C = 1，表示无符号数运算发生上溢出；其他情况下 C = 0。

（2）在减法指令中（包括比较指令 CMP），当运算中发生错位（即无符号数运算发生下溢出），则 C = 0；其他情况下 C = 1。

（3）对于在操作数中包含移位操作的运算指令（非加/减法指令），C 被设置成被移位寄存器最后移出去的位。

（4）对于其他非加/减法运算指令，C 的值通常不受影响。

4）V

下面分两种情况讨论 V 的设置方法。

（1）对于加/减运算指令，当操作数和运算结果都是以二进制的补码表示的带符号的数时，且运算结果超出了有符号运算的范围是溢出。V = 1 表示符号位溢出。

（2）对于非加/减法指令，通常不改变标志位 V 的值（具体可参照 ARM 指令手册）。

尽管以上 C 和 V 的定义看起来颇为复杂，但使用时在大多数情况下用一个简单的条件测试指令即可，不需要程序员计算出条件码的精确值即可得到需要的结果。

2．Q 标志位

在带 DSP 指令扩展的 ARM v5 及更高版本中，bit[27]被指定用于指示增强的 DAP 指令是否发生了溢出，因此也就被称为 Q 标志位。同样，在 SPSR 中 bit[27]也被称为 Q 标志位，用于在异常中断发生时保存和恢复 CPSR 中的 Q 标志位。

在 ARM v5 以前的版本及 ARM v5 的非 E 系列处理器中，Q 标志位没有被定义，属于待扩展的位。

3．控制位

CPSR 的低 8 位（I、F、T 及 M[4:0]）统称为控制位。当异常发生时，这些位的值将发生相应的变化。另外，如果在特权模式下，也可以通过软件编程来修改这些位的值。

1）中断禁止位

I = 1，IRQ 被禁止。

F = 1，FIQ 被禁止。

2）状态控制位

T 位是处理器的状态控制位。

T = 0，处理器处于 ARM 状态（即正在执行 32 位的 ARM 指令）。

T = 1，处理器处于 Thumb 状态（即正在执行 16 位的 Thumb 指令）。

当然，T 位只有在 T 系列的 ARM 处理器上才有效，在非 T 系列的 ARM 版本中，T 位将始终为 0。

3）模式控制位

M[4:0]作为位模式控制位，这些位的组合确定了处理器处于哪种状态。如表 2-6 所示列出了其具体含义。

只有表中列出的组合是有效的，其他组合无效。

表 2-6　状态控制位 M[4:0]

M[4：0]	处理器模式	可以访问的寄存器
0b10000	User	PC，R14～R0，CPSR
0b10001	FIQ	PC，R14_fiq～R8_fiq，R7～R0，CPSR，SPSR_fiq
0b10010	IRQ	PC，R14_irq～R13_irq，R12～R0，CPSR，SPSR_irq
0b10011	Supervisor	PC，R14_svc～R13_svc，R12～R0，CPSR，SPSR_svc
0b10111	Abort	PC，R14_abt～R13_abt，R12～R0，CPSR，SPSR_abt
0b11011	Undefined	PC，R14_und～R13_und，R12～R0，CPSR，SPSR_und
0b11111	System	PC，R14～R0，CPSR（ARM v4 及更高版本）
0b10110	Secure monitor	PC,R0-R12, CPSR,SPSR_mon,r13_mon,r14_mon

4．IF-THEN 标志位

CPSR 中的 bits[15:10,26:25]称为 if-then 标志位，它用于对 thumb 指令集中 if-then-else 这一类语句块的控制。

其中 IT[7:5]定义为基本条件，如图 2-8 所示。

IT[4:0]被定义为 IF-THEN 语句块的长度。

[7:5]	[4]	[3]	[2]	[1]	[0]	
控制基础	P1	P2	P3	P4	1	4 指令IT块入口点
控制基础	P1	P2	P3	1	0	3 指令IT块入口点
控制基础	P1	P2	1	0	0	2 指令IT块入口点
控制基础	P1	1	0	0	0	1 指令IT块入口点
000	0	0	0	0	0	普通执行状态，无IT块入口点

图 2-8　IF-THEN 标志位[7:5]的定义

5．E 位/A 位/GE[19-16]位的定义

A 表示异步异常禁止位。

E 表示大小端控制位，0 表示小端操作，1 表示大端操作。注意，该位在预取阶段是被忽略的。

GE[19-16]用于表示在 SIMD 指令集中的大于、等于标志。在任何模式下该位可读可写。

2.12 三星 S5PC100 处理器介绍

S5PC100 是著名的半导体公司 SAMSUNG 推出的一款 32 位 RISC 处理器，它为手持设备和一般类型的应用提供了低价格、低功耗、高性能微控制器的解决方案。S5PC100 的内核基于 Cortex-A8，带有 MMU（Memory Management Unit）功能，采用 0.18μm 工艺，其主频可达 203MHz，适合于对成本和功耗敏感的需求。同时它还采用了 AMBA（Advanced Microcontroller Bus Architecture）的新型总线结构，实现了 MMU、AMBA BUS、Harvard 的高速缓冲体系结构，同时支持 Thumb16 位压缩指令集，从而能以较小的存储空间需求，获得 32 位的系统性能。

S5PC100 是一款低功耗、低成本、高性能的移动领域及普通应用的微处理解决方案，集成的 Cortex-A8 使用 v7 架构，以及支持众多的外设。

S5PC100 还采取了 64 位的内部总线架构，并包括许多性能强大的加速硬件，例如针对于图形图像加速、显示、伸缩裁剪、集成的多媒体格式编码（MFC），硬件加速还支持实时会议、模拟电视输出、高清视频、PAL 等。

S5PC100 还支持一个可扩展高端内存的接口，其中 DRAM 接口可配置支持 DDR、DDR2、LPDDR2。

其片上功能如下：

- 先进的电源管理系统。
- 用于安全启动的片内 ROM/片内 RAM。
- 8 位 ITU 601/656 摄像头接口，最大支持 8M 像素裁剪图像，以及 64M 像素的未裁剪图像。
- 多格式编解码单元支持 MPEG-4/H.263/H.264 的编解码，可支持 720p/30 帧每秒。以及对 MPEG2/VC1/DIVx video 的解码，最大支持 720p/30 帧每秒。
- 支持 3D/2D 多媒体加速技术。
- 支持模拟电视信号输出及高清晰度多媒体接口。
- AC97 音频编解码接口，PCM 串行音频接口。
- 3 通道的 24 位 IIS 接口。
- 一个 2 通道的 IIC 总线控制器。
- 一个 3 通道的 SPI 总线控制器。
- 4 通道 UART 接口，包括一个 4M 波特率的蓝牙 2.0 接口。
- 一个红外传感器端口。
- 一个 USB 2.0 OTG 控制器。
- 一个 USB 1.1 HOST。
- SD/MMC 高速数字信号卡接口。
- 24 通道的 DMA 控制器。
- 8×8 矩阵键盘。

- 10 通道的 12 位混合 ADC 接口。
- 可配置的 GPIO 资源。
- 实时时钟/PLL/看门狗等片上外设。

S5PC100 处理器支持大/小端模式存储字数据，其寻址空间可达 4GB，对于外部 I/O 设备的数据宽度，可以是 8/16/32 位，所有的存储器 Bank（共有 15 个）都具有可编程的操作周期，而且支持各种 ROM 引导方式（NOR/Nand Flash、EEPROM/SD/USB/ONENAND 等），其结构框图如图 2-9 所示。

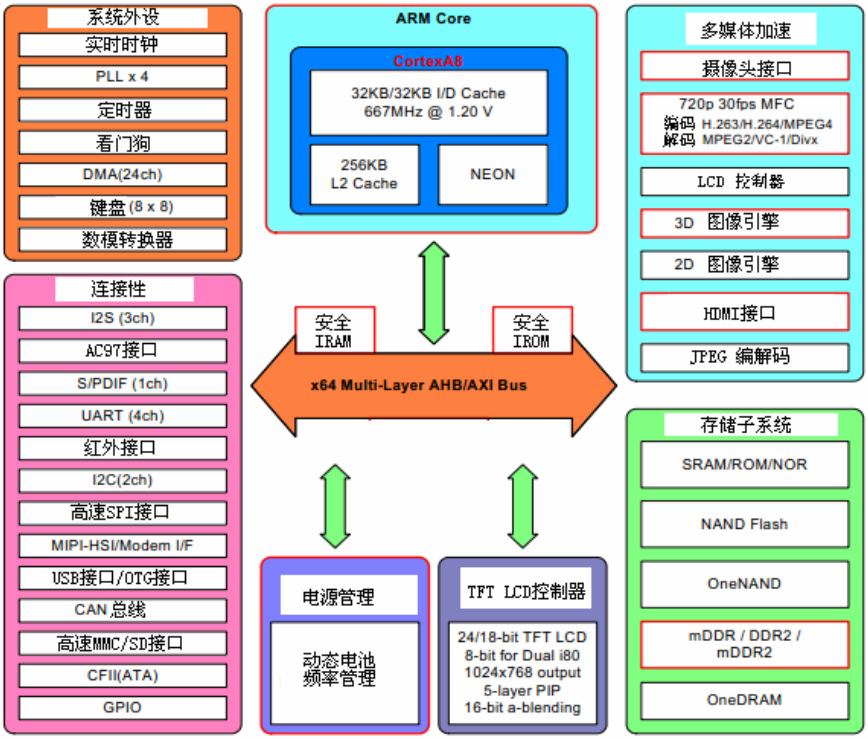

图 2-9　S5PC100 结构框图

2.13　FS_S5PC100 开发平台介绍

FS_S5PC100 是由华清远见研发部自主研发的一款基于 Cortex-A8 核心的嵌入式系统学习开发平台，以适应 Android、Linux、Wince 等智能操作系统的发展及市场需求。华清远见研发中心及教师团队将不断完善推广平台资料，帮助大家快速掌握高端嵌入式技术。

FS_S5PC100 核心实验板如图 2-10 所示。

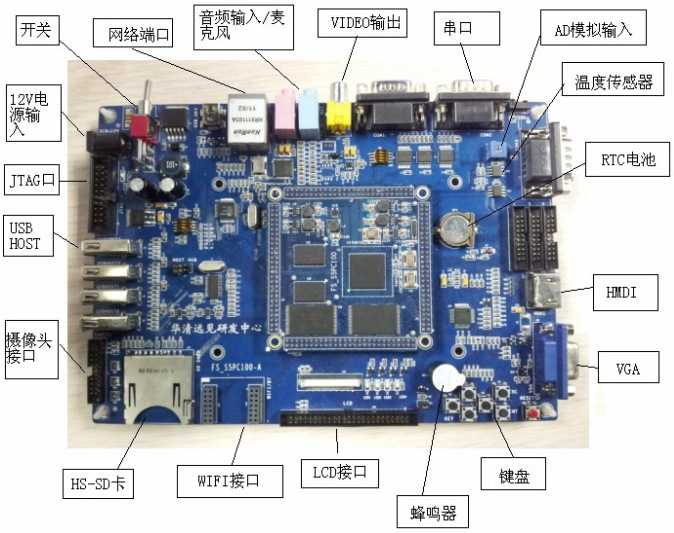

图 2-10　FS_S5PC100 核心实验板

（1）FS_S5PC100 拥有如下丰富的硬件资源。

- 存储器：256MB 的 NAND Flash，256MB 的 DDR2 内存，2MB 的 NOR Flash
- 显示输出接口：LCD 接口、VGA 接口、TVout 接口、HDMI 接口。
- 视频输入接口：Camera 接口。
- 串口：2 路 5 线串口、1 路 3 线串口。
- 红外通信接口：1 路红外收发。
- 存储卡接口：SD 卡接口。
- SPI：SPI 的 E2PROM （用于 SPI 实验）。
- I2C：I2C 的温度传感器（用于 I2C 实验）。
- A/D：变阻器（用于 A/D 实验）。
- USB：USB2.0-OTG 接口、4 路 USB Host 接口。
- PWM：蜂鸣器（用于 PWM 实验）。
- 网络接口：DM9000AE 以太网控制器，实现 10M/100M 自适应以太网通信。
- 外扩接口：蓝牙、WiFi 等。
- 音频接口：WM9714。

（2）FS_S5PC100 软件资源。其中包括下列资源信息。

① 裸机代码资源。

GPIO 测试代码，I2C 总线测试温度传感器，SPI 总线对 EEPROM 的读/写，ADC 模拟数字转换例子，红外传感器收发实例，LCD 裸机驱动，摄像头裸机驱动，NAND/NOR FLASH 的读/写驱动，触摸屏裸机驱动，DMA 控制器例子（内存至内存，串口至内存），RTC 实时时钟驱动，蜂鸣器驱动。

丰富的裸机代码，可以让初学者迅速掌握基于 Cortex-A8 的 FS_S5PC100 开发平台的各项功能，在裸机驱动的帮助下，读者对寄存器操作的认识和各个接口的操作特性有深刻体会。

② 系统移植资源

uboot-1.1.6 源码及 uboot-2011.09 源码，在两种不同版本的 uboot 对比下，可以让读者迅速掌握新旧 uboot 的结构差异，注意，新旧版本 uboot 变化很大。

Android 源码，当前最流行的手机操作系统之一，Google 的 Android 操作系统，读者使用该源代码，可以对 Android 系统架构、开发及移植有很深刻的理解，对有望从事该职业的读者有很大的帮助。

Linux 2.6.29 源代码及 Linux 2.6.35 源代码，同样已经移植到 FS_S5PC100 开发平台的 Linux 内核，其中包含了基于 Linux 架构的驱动模块，让读者从裸机驱动层次过渡到 Linux 驱动架构的层次。其中，Linux 2.6.29 是 Android 的配套 Linux 内核，而 Linux 2.6.35 为原始版的内核。

③ Linux 内核驱动资源。

包括 ADC 驱动、IO 控制蜂鸣器驱动、PWM 控制蜂鸣器驱动、内核 irq 测试、LED 驱动、平台设备驱动、EEPROM 读/写驱动、温度传感器测试驱动等。

④ 综合案例资源。

基于该平台，华清远见研发中心研发了多项典型教学案例，涵盖嵌入式系统开发课程及物联网开发课程，其中有：

《智能感应家居》，该案例的前端使用了 Android 操作系统，并通过多级 ZigBee 节点实现了一个覆盖家居的网络，在该 zigbee 网络中，通过主控端控制各个节点端，从而实现家居控制。

《仓储物联网系统》，该案例实现了从一个 Web 端来对一个由 ZigBee 节点组成的监控网络的操控，其中 FS_S5PC100 作为一级网关，FS11C14 作为数据采集节点，在 zigbee 的网络传输下，将数据汇集到一级网关，最终上报给 Web 服务端。

《3G 机器人》通过 Android 平台和 3G 网络远程控制机器人的行为。

综合案例的学习，不仅巩固了对所用的软件与硬件的知识，并且积累了一定的开发经验，更是让学员自己去尝试项目的开发流程，以及系统架构、编程、调试等步骤。

（3）本书所有源代码均在 FS_S5PC100 开发平台上测试通过。

2.14　本章小结

本章介绍了 ARM 处理器的一些关键技术，如 ARM 核的工作模式、存储系统、流水线、寄存器组织等。并且列举了一款基于 Cortex-A8 核的处理器芯片 A5PC100。通过本章的学习，学员可以对 ARM 核的一些关键技术有所认识。

2.15 练习题

1. 简述 ARM 可以工作的几种模式。
2. ARM 核有多少个寄存器？
3. 什么寄存器用于存储 PC 和 LR 寄存器？
4. R13 通常用来存储什么？
5. 哪种模式使用的寄存器最少？
6. CPSR 的哪一位反映了处理器的状态？

第 3 章　ARM 微处理器的指令系统

ARM 指令集可以分为跳转指令、数据处理指令、程序状态寄存器传输指令、Load/Store 指令、协处理器指令和异常中断产生指令。根据使用的指令类型不同，指令的寻址方式分为数据处理指令寻址方式和内存访问指令寻址方式。

本章主要介绍 ARM 汇编语言。主要内容如下：

- ARM 处理器的寻址方式。
- ARM 处理器的指令集。

3.1　ARM 处理器的寻址方式

ARM 指令的寻址方式分为数据处理指令寻址方式和内存访问指令寻址方式。

3.1.1　数据处理指令寻址方式

数据处理指令的基本语法格式如下：

```
<opcode> {<cond>} {S} <Rd>,<Rn>,<shifter_operand>
```

其中，<shifter_operand>有 11 种形式，如表 3-1 所示。

表 3-1　<shifter_operand>的寻址方式

	语　　法	寻 址 方 式
1	#<immediate>	立即数寻址
2	<Rm>	寄存器寻址
3	<Rm>, LSL　#<shift_imm>	立即数逻辑左移
4	<Rm>, LSL　<Rs>	寄存器逻辑左移
5	<Rm>, LSR　#<shift_imm>	立即数逻辑右移
6	<Rm>, LSR　<Rs>	寄存器逻辑右移
7	<Rm>, ASR　#<shift_imm>	立即数算术右移
8	<Rm>, ASR　<Rs>	寄存器算术右移
9	<Rm>, ROR　#<shift_imm>	立即数循环右移
10	<Rm>, ROR　<Rs>	寄存器循环右移
11	<Rm>, RRX	寄存器扩展循环右移

数据处理指令寻址方式可以分为以下几种。

（1）立即数寻址方式。

（2）寄存器寻址方式。

（3）寄存器移位寻址方式。

1．立即数寻址方式

指令中的立即数是由一个 8bit 的常数移动 4bit 偶数位（0，2，4，…，26，28，30）得到的。所以，每一条指令都包含一个 8bit 的常数 X 和移位值 Y，得到的立即数 ＝X 循环右移（2×Y），如图 3-1 所示。

图 3-1　立即数表示方法

下面列举了一些有效的立即数：

0xFF、0x104、0xFF0、0xFF00、0xFF000、0xFF000000、0xF000000F

下面是一些无效的立即数：

0x101、0x102、0xFF1、0xFF04、0xFF003、0xFFFFFFFF、0xF000001F

下面是一些应用立即数的指令：

```
MOV  R0,#0                ;送 0 到 R0
ADD  R3,R3,#1             ;R3 的值加 1
CMP  R7,#1000             ;将 R7 的值和 1000 比较
BIC  R9,R8,#0xFF00        ;将 R8 中 8～15 位清零，结果保存在 R9 中
```

2．寄存器寻址方式

寄存器的值可以被直接用于数据操作指令，这种寻址方式是各类处理器经常采用的一种方式，也是一种执行效率较高的寻址方式，如：

```
MOV  R2,R0                ;R0 的值送 R2
ADD  R4,R3,R2             ;R2 加 R3，结果送 R4
CMP  R7,R8                ;比较 R7 和 R8 的值
```

3．寄存器移位寻址方式

寄存器的值在被送到 ALU 之前，可以事先经过桶形移位寄存器的处理。预处理和移位发生在同一周期内，所以有效地使用移位寄存器，可以增加代码的执行效率。

下面是一些在指令中使用了移位操作的例子：

```
ADD  R2,R0,R1,LSR  #5
MOV  R1,R0,LSL  #2
RSB  R9,R5,R5,LSL  #1
SUB  R1,R2,R0,LSR #4
MOV  R2,R4,ROR  R0
```

3.1.2　内存访问指令寻址方式

内存访问指令的寻址方式可以分为以下几种。

（1）字及无符号字节的 Load/Store 指令的寻址方式。

（2）杂类 Load/Store 指令的寻址方式。

（3）批量 Load/Store 指令的寻址方式。

（4）协处理器 Load/Store 指令的寻址方式。

1．字及无符号字节的 Load/Store 指令的寻址方式

字及无符号字节的 Load/Store 指令语法格式如下：

```
LDR|STR{<cond>}{B}{T}  <Rd>,<addressing_mode>
```

其中，<addressing_mode>共有 9 种寻址方式，如表 3-2 所示。

表 3-2　字及无符合字节的 Load/Store 指令的寻址方式

	格　　式	模　　式
1	[Rn，#±<offset_12>]	立即数偏移寻址 （Immediate offset）
2	[Rn，±Rm]	寄存器偏移寻址 （Register offset）
3	[Rn，Rm，<shift>#< offset_12>]	带移位的寄存器偏移寻址 （Scaled register offset）
4	[Rn，#±< offset_12>]!	立即数前索引寻址 （Immediate pre-indexed）
5	[Rn，±Rm]!	寄存器前索引寻址 （Register post-indexed）
6	[Rn，Rm，<shift>#< offset_12>]!	带移位的寄存器前索引寻址 （Scaled register pre-indexed）
7	[Rn]，#±< offset_12>	立即数后索引寻址 （Immediate post-indeted）
8	[Rn]，±<Rm>	寄存器后索引寻址 （Register post-indexed）
9	[Rn]，±<Rm>，<shift>#< offset_12>	带移位的寄存器后索引寻址 （Scaled register post-indexed）

上表中，“!”表示完成数据传输后要更新基址寄存器。

2．杂类 Load/Store 指令的寻址方式

使用该类寻址方式的指令的语法格式如下：

```
LDR|STR{<cond>}H|SH|SB|D  <Rd>,<addressing_mode>
```

使用该类寻址方式的指令包括（有符号/无符号）半字 Load/Store 指令、有符号字节 Load/Store 指令和双字 Load/Store 指令。

该类寻址方式分为 6 种类型，如表 3-3 所示。

表 3-3　杂类 Load/Store 指令的寻址方式

	格　式	模　式
1	[Rn，#±<offset_8>]	立即数偏移寻址 （Immediate offset）
2	[Rn，±Rm]	寄存器偏移寻址 （Register offset）
3	[Rn，#±< offset_8>]!	立即数前索引寻址 （Immediate pre-indexed）
4	[Rn，±Rm]!	寄存器前索引寻址 （Register post-indexed）
5	[Rn]，#±< offset_8>	立即数后索引寻址 （Immediate post-indexed）
6	[Rn]，±<Rm>	寄存器后索引寻址 （Register post-indexed）

3．批量 Load/Store 指令寻址方式

批量 Load/Store 指令将一片连续内存单元的数据加载到通用寄存器组中或将一组通用寄存器的数据存储到内存单元中。

批量 Load/Store 指令的寻址模式产生一个内存单元的地址范围，指令寄存器和内存单元的对应关系满足这样的规则，即编号低的寄存器对应于内存中低地址单元，编号高的寄存器对应于内存中的高地址单元。

该类指令的语法格式如下：

```
LDM|STM{<cond>}<addressing_mode>  <Rn>{!},<registers><^>
```

该类指令的寻址方式如表 3-4 所示。

表 3-4　批量 Load/Store 指令的寻址方式

	格　式	模　式
1	IA（Increment After）	后递增方式
2	IB（Increment Before）	先递增方式
3	DA（Decrement After）	后递减方式
4	DB（Decrement Before）	先递减方式

4．堆栈操作寻址方式

堆栈操作寻址方式和批量 Load/Store 指令寻址方式十分类似。但对于堆栈的操作，数据写入内存和从内存中读出要使用不同的寻址模式，因为进栈操作（pop）和出栈操作（push）要在不同的方向上调整堆栈。

下面详细讨论如何使用合适的寻址方式实现数据的堆栈操作。

根据不同的寻址方式，将堆栈分为以下 4 种。

（1）Full 栈：堆栈指针指向栈顶元素（last used location）。

（2）Empty 栈：堆栈指针指向第一个可用元素（the first unused location）。

（3）递减栈：堆栈向内存地址减小的方向生长。

（4）递增栈：堆栈向内存地址增加的方向生长。

根据堆栈的不同种类，将其寻址方式分为以下 4 种。

（1）满递减 FD（Full Descending）。

（2）空递减 ED（Empty Descending）。

（3）满递增 FA（Full Ascending）。

（4）空递增 EA（Empty Ascending）。

如表 3-5 所示列出了堆栈的寻址方式和批量 Load/Store 指令寻址方式的对应关系。

表 3-5　堆栈寻址方式和批量 Load/Store 指令寻址方式的对应关系

批量数据寻址方式	堆栈寻址方式	L 位	P 位	U 位
LDMDA	LDMFA	1	0	0
LDMIA	LDMFD	1	0	1
LDMDB	LDMEA	1	1	0
LDMIB	LDMED	1	1	1
STMDA	STMED	0	0	0
STMIA	STMEA	0	0	1
STMDB	STMFD	0	1	0
STMIB	STMFA	0	1	1

5．协处理器 Load/Store 寻址方式

协处理器 Load/Store 指令的语法格式如下：

```
<opcode>{<cond>}{L}  <coproc>,<CRd>,<addressing_mode>
```

3.2　ARM 处理器的指令集

3.2.1　数据操作指令

数据操作指令是指对存放在寄存器中的数据进行操作的指令。主要包括数据传送指令、算术指令、逻辑指令、比较与测试指令及乘法指令。

如果在数据处理指令前使用 S 前缀，指令的执行结果将会影响 CPSR 中的标志位。数据处理指令如表 3-6 所示。

表 3-6　数据处理指令列表

助 记 符	操　作	行　为
MOV	数据传送	
MVN	数据取反传送	
AND	逻辑与	Rd：=Rn AND op2
EOR	逻辑异或	Rd：=Rn EOR op2
SUB	减	Rd：=Rn − op2
RSB	翻转减	Rd：=op2 − Rn
ADD	加	Rd：=Rn + op2
ADC	带进位的加	Rd：=Rn + op2 + C
SBC	带进位的减	Rd：=Rn− op2 + C − 1
RSC	带进位的翻转减	Rd：=op2 − Rn + C − 1
TST	测试	Rn AND op2 并更新标志位
TEQ	测试相等	Rn EOR op2 并更新标志位
CMP	比较	Rn−op2 并更新标志位
CMN	负数比较	Rn+op2 并更新标志位
ORR	逻辑或	Rd：=Rn OR op2
BIC	位清 0	Rd：=Rn AND NOT（op2）

1．MOV 指令

MOV 指令是最简单的 ARM 指令，执行的结果就是把一个数 N 送到目标寄存器 Rd，其中 N 可以是寄存器，也可以是立即数。

MOV 指令多用于设置初始值或者在寄存器间传送数据。

MOV 指令将移位码（shifter_operand）表示的数据传送到目的寄存器 Rd，并根据操作的结果更新 CPSR 中相应的条件标志位。

（1）指令的语法格式：

```
MOV{<cond>}{S}   <Rd>,<shifter_operand>
```

（2）指令举例：

```
MOV     R0, R0            ; R0 = R0… NOP 指令
MOV     R0, R0, LSL#3     ; R0 = R0 * 8
```

如果 R15 是目的寄存器，将修改程序计数器或标志。这用于被调用的子函数结束后返回到调用代码，方法是把连接寄存器的内容传送到 R15。

```
MOV     PC, R14       ; 退出到调用者，用于普通函数返回，PC 即是 R15
MOVS    PC, R14       ; 退出到调用者并恢复标志位，用于异常函数返回
```

（3）指令的使用如下。

MOV 指令主要完成以下功能。

① 将数据从一个寄存器传送到另一个寄存器。

② 将一个常数值传送到寄存器中。

③ 实现无算术和逻辑运算的单纯移位操作，操作数乘以 2^n 可以用左移 n 位来实现。

④ 当 PC（R15）用做目的寄存器时，可以实现程序跳转。如“MOV　PC，LR”，所以这种跳转可以实现子程序调用及从子程序返回，代替指令“B，BL”。

⑤ 当 PC 作为目标寄存器且指令中 S 位被设置时，指令在执行跳转操作的同时，将当前处理器模式的 SPSR 寄存器的内容复制到 CPSR 中。这种指令“MOVS PC LR”可以实现从某些异常中断中返回。

2．MVN 指令

MVN 是反相传送（Move Negative）指令。它将操作数的反码传送到目的寄存器。

MVN 指令多用于向寄存器传送一个负数或生成位掩码。

MVN 指令将 shifter_operand 表示的数据的反码传送到目的寄存器 Rd，并根据操作结果更新 CPSR 中相应的条件标志位。

（1）指令的语法格式：

```
MNV{<cond>}{S}   <Rd>,<shifter_operand>
```

（2）指令举例如下。

MVN 指令和 MOV 指令相同，也可以把一个数 N 送到目标寄存器 Rd，其中 N 可以是立即数，也可以是寄存器。这是逻辑非操作而不是算术操作，这个取反的值加 1 才是它的取负的值。

```
MVN    R0, #4              ; R0 = -5
MVN    R0, #0              ; R0 = -1
```

（3）指令的使用如下。

MVN 指令主要完成以下功能：

① 向寄存器中传送一个负数。

② 生成位掩码（Bit Mask）。

③ 求一个数的反码。

3．AND 指令

AND 指令将 shifter_operand 表示的数值与寄存器 Rn 的值按位（bitwise）做逻辑与操作，并将结果保存到目标寄存器 Rd 中，同时根据操作的结果更新 CPSR 寄存器。

（1）指令的语法格式：

```
AND{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

（2）指令举例如下。

① 保留 R0 中的 0 位和 1 位，丢弃其余的位。

```
AND    R0, R0, #3
```

② R2 = R1&R3。

```
AND   R2,R1,R3
```

③ R0 = R0&0x01，取出最低位数据。

```
ANDS    R0,R0,#0x01
```

4．EOR 指令

EOR（Exclusive OR）指令将寄存器 Rn 中的值和 shifter_operand 的值执行按位“异或”操作，并将执行结果存储到目的寄存器 Rd 中，同时根据指令的执行结果更新 CPSR 中相应的条件标志位。

（1）指令的语法格式：

```
EOR{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

（2）指令举例：

① 反转 R0 中的位 0 和 1。

```
EOR     R0, R0, #3
```

② 将 R1 的低 4 位取反。

```
EOR     R1,R1,#0x0F
```

③ R2 = R1∧R0。

```
EOR     R2,R1,R0
```

④ 将 R5 和 0x01 进行逻辑异或，结果保存到 R0，并根据执行结果设置标志位。

```
EORS     R0,R5,#0x01
```

5．SUB 指令

SUB（Subtract）指令从寄存器 Rn 中减去 shifter_operand 表示的数值，并将结果保存到目标寄存器 Rd 中，并根据指令的执行结果设置 CPSR 中相应的标志位。

（1）指令的语法格式：

```
SUB{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

（2）SUB 指令举例：

① R0 = R1 − R2。

```
SUB     R0, R1, R2
```

② R0 = R1 − 256。

```
SUB     R0, R1, #256
```

③ R0 = R2− (R3<<1)。

```
SUB     R0, R2, R3,LSL#1
```

6．RSB 指令

RSB（Reverse Subtract）指令从寄存器 shifter_operand 中减去 Rn 表示的数值，并将结果保存到目标寄存器 Rd 中，并根据指令的执行结果设置 CPSR 中相应的标志位。

（1）指令的语法格式：

```
RSB{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

（2）RSB 指令举例：

下面的指令序列可以求一个 64 位数值的负数。64 位数放在寄存器 R0 与 R1 中，其负数放在 R2 和 R3 中。其中 R0 与 R2 中放低 32 位值。

```
RSBS    R2,R0,#0
RSC     R3,R1,#0
```

7．ADD 指令

ADD 指令将寄存器 shifter_operand 的值加上 Rn 表示的数值，并将结果保存到目标寄存器 Rd 中，并根据指令的执行结果设置 CPSR 中相应的标志位。

（1）指令的语法格式：

```
ADD{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

（2）ADD 指令举例：

```
ADD    R0, R1, R2            ; R0 = R1 + R2
ADD    R0, R1, #256      ; R0 = R1 + 256
ADD    R0, R2, R3,LSL#1  ; R0 = R2 + (R3 << 1)
```

8．ADC 指令

ADC 指令将寄存器 shifter_operand 的值加上 Rn 表示的数值，再加上 CPSR 中的 C 条件标志位的值，将结果保存到目标寄存器 Rd 中，并根据指令的执行结果设置 CPSR 中相应的标志位。

（1）指令的语法格式：

```
ADC{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

（2）ADC 指令举例：

ADC 指令将把两个操作数加起来，并把结果放置到目的寄存器中。它使用一个进位标志位，这样就可以做比 32 位大的加法。下面的例子将加两个 128 位的数。

128 位结果：寄存器 R0、R1、R2 和 R3。

第一个 128 位数：寄存器 R4、R5、R6 和 R7。

第二个 128 位数：寄存器 R8、R9、R10 和 R11。

```
ADDS    R0, R4, R8            ;加低端的字
ADCS    R1, R5, R9            ;加下一个字，带进位
ADCS    R2, R6, R10           ;加第三个字，带进位
ADCS    R3, R7, R11           ;加高端的字，带进位
```

9．SBC 指令

SBC（Subtract with Carry）指令用于执行操作数大于 32 位时的减法操作。该指令从寄存器 Rn 中减去 shifter_operand 表示的数值，再减去寄存器 CPSR 中 C 条件标志位的反码［NOT(Carry flag)］，并将结果保存到目标寄存器 Rd 中，并根据指令的执行结果设置 CPSR 中相应的标志位。

（1）指令的语法格式：

```
SBC{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

（2）SBC 指令举例：

下面的程序使用 SBC 实现 64 位减法，（R1，R0）-（R3，R2），结果存放到（R1，R0）。

```
SUBS    R0,R0,R2
SBCS    R1,R1,R3
```

10．RSC 指令

RSC（Reverse Subtract with Carry）指令从寄存器 shifter_operand 中减去 Rn 表示的数值，再减去寄存器 CPSR 中 C 条件标志位的反码［NOT（Carry Flag）］，并将结果保存到目标寄存器 Rd 中，并根据指令的执行结果设置 CPSR 中相应的标志位。

（1）指令的语法格式：

```
RSC{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

（2）RSC 指令举例：

下面的程序使用 RSC 指令实现求 64 位数值的负数。

```
RSBS    R2,R0,#0
RSC     R3,R1,#0
```

11．TST 测试指令

TST（Test）测试指令用于将一个寄存器的值和一个算术值进行比较。条件标志位根据两个操作数做“逻辑与”后的结果设置。

（1）指令的语法格式：

```
TST{<cond>}  <Rn>,<shifter_operand>
```

（2）TST 指令举例：

TST 指令类似于 CMP 指令，不产生放置到目的寄存器中的结果。而是在给出的两个操作数上进行操作并把结果反映到状态标志上。使用 TST 指令来检查是否设置了特定的位。操作数 1 是要测试的数据字而操作数 2 是一个位掩码。经过测试后，如果匹配则设置 Zero 标志，否则清除它。与 CMP 指令一样，该指令不需要指定 S 后缀。

下面的指令测试在 R0 中是否设置了位 0。

```
TST    R0, #1
```

12．TEQ 指令

TEQ（Test Equivalence）指令用于将一个寄存器的值和一个算术值做比较。条件标志位根据两个操作数做“逻辑异或”后的结果设置。以便后面的指令根据相应的条件标志来判断是否执行。

（1）指令的语法格式：

```
TEQ{<cond>}  <Rn>,<shifter_operand>
```

（2）TEQ 指令举例：

下面的指令是比较 R0 和 R1 是否相等，该指令不影响 CPSR 中的 V 位和 C 位。

```
TEQ    R0,R1
```

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

13．CMP 指令

CMP（Compare）指令使用寄存器 Rn 的值减去 operand2 的值，根据操作的结果更新 CPSR 中相应的条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。

（1）指令的语法格式：

```
CMP{<cond>}  <Rn>,<shifter_operand>
```

（2）CMP 指令举例：

CMP 指令允许把一个寄存器的内容与另一个寄存器的内容或立即值进行比较，更改状态标志来允许进行条件执行。它进行一次减法，但不存储结果，而是正确地更改标志位。标志位表示的是操作数 1 与操作数 2 比较的结果（其值可能为大于、小于、相等）。如果操作数 1 大于操作数 2，则此后的有 GT 后缀的指令将可以执行。

显然，CMP 不需要显式地指定 S 后缀来更改状态标志。

① 下面的指令是比较 R1 和立即数 10 并设置相关的标志位。

```
CMP    R1,#10
```

② 下面的指令是比较寄存器 R1 和 R2 中的值并设置相关的标志位。

```
CMP    R1,R2
```

通过上面的例子可以看出，CMP 指令与 SUBS 指令的区别在于 CMP 指令不保存运算结果，在进行两个数据大小判断时，常用 CMP 指令及相应的条件码来进行操作。

14．CMN 指令

CMN（Compare Negative）指令使用寄存器 Rn 的值减去 operand2 的负数值（加上 operand2），根据操作的结果更新 CPSR 中相应的条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。

（1）指令的语法格式：

```
CMN{<cond>}  <Rn>,<shifter_operand>
```

（2）CMN 指令举例：

CMN 指令将寄存器 Rn 中的值加上 shifter_operand 表示的数值，根据加法的结果设置 CPSR 中相应的条件标志位。寄存器 Rn 中的值加上 shifter_operand 的操作结果对 CPSR 中条件标志位的影响，与寄存器 Rn 中的值减去 shifter_operand 的操作结果的相反数对 CPSR 中条件标志位的影响有细微差别。当第 2 个操作数为 0 或者为 0x80000000 时两者结果不同。比如下面两条指令。

```
CMP    Rn,#0
CMN    Rn,#0
```

第 1 条指令使标志位 C 值为 1，第 2 条指令使标志位 C 值为 0。

下面的指令使 R0 值加 1，判断 R0 是否为 1 的补码，若是，则 Z 置位。

```
CMN    R0,#1
```

15．ORR 指令

ORR（Logical OR）为逻辑或操作指令，它将第 2 个源操作数 shifter_operand 的值与寄存器 Rn 的值按位做“逻辑或”操作，结果保存到 Rd 中。

（1）指令的语法格式：

```
ORR{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

（2）ORR 指令举例：

① 设置 R0 中位 0 和 1。

```
ORR    R0, R0, #3
```

② 将 R0 的低 4 位置 1。

```
ORR    R0,R0,#0x0F
```

③ 使用 ORR 指令将 R2 的高 8 位数据移入到 R3 的低 8 位中。

```
MOV    R1,R2,LSR #4
ORR    R3,R1,R3,LSL #8
```

16．BIC 位清零指令

BIC（Bit Clear）位清零指令，将寄存器 Rn 的值与第 2 个源操作数 shifter_operand 的值的反码按位做“逻辑与”操作，结果保存到 Rd 中。

（1）指令的语法格式：

```
BIC{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

（2）BIC 指令举例：

① 清除 R0 中的位 0、1 和 3，保持其余的不变。

```
BIC    R0, R0, #0x1011
```

② 将 R3 的反码和 R2 做“逻辑与”操作，结果保存到 R1 中。

```
BIC   R1,R2,R3
```

3.2.2 乘法指令

ARM 乘法指令完成两个数据的乘法。两个 32 位二进制数相乘的结果是 64 位的积。在有些 ARM 的处理器版本中，将乘积的结果保存到两个独立的寄存器中。另外一些版本只将最低有效 32 位存放到一个寄存器中。无论是哪种版本的处理器，都有乘—累加的变型指令，将乘积连续累加得到总和。而且有符号数和无符号数都能使用。对于有符号数和无符号数，结果的最低有效位是一样的。因此，对于只保留 32 位结果的乘法指令，不需要区分有符号数和无符号数这两种情况。

如表 3-7 所示为各种形式乘法指令的功能。

表 3-7　各种形式乘法指令

操作码[23：21]	助　记　符	意　　义	操　　作
000	MUL	乘（保留 32 位结果）	Rd：=（Rm×Rs）[31：0]
001	MLA	乘—累加（保留 32 位结果）	Rd：=（Rm×Rs＋Rn）[31：0]
100	UMULL	无符号数长乘	RdHi：RdLo：=Rm×Rs
101	UMLAL	无符号长乘—累加	RdHi：RdLo：＋=Rm×Rs
110	SMULL	有符号数长乘	RdHi：RdLo：=Rm×Rs
111	SMLAL	有符号数长乘—累加	RdHi：RdLo：＋=Rm×Rs

其中：

（1）“RdHi：RdLo”是由 RdHi（最高有效 32 位）和 RdLo（最低有效 32 位）连接形成的 64 位数，“[31:0]”只选取结果的最低有效 32 位。

（2）简单的赋值由“:=”表示。

（3）累加（将右边加到左边）是由“+=”表示。

各个乘法指令中的位 S（参考下文具体指令的语法格式）控制条件码的设置会产生以下结果。

① 对于产生 32 位结果的指令形式，将标志位 N 设置为 Rd 的第 31 位的值；对于产生长结果的指令形式，将其设置为 RdHi 的第 31 位的值。

② 对于产生 32 位结果的指令形式，如果 Rd 等于零，则标志位 Z 置位；对于产生长结果的指令形式，RdHi 和 RdLo 同时为零时，标志位 Z 置位。

③ 将标志位 C 设置成无意义的值。

④ 标志位 V 不变。

1．MUL 指令

MUL（Multiply）32 位乘法指令将 Rm 和 Rs 中的值相乘，结果的最低 32 位保存到 Rd 中。

（1）指令的语法格式：

```
MUL{<cond>}{S}   <Rd>,<Rm>,<Rs>
```

（2）指令举例：

① R1 = R2 × R3。

```
MUL    R1, R2, R3
```

② R0 = R3 × R7，同时设置 CPSR 中的 N 位和 Z 位。

```
MULS    R0, R3, R7
```

2．MLA 指令

MLA（Multiply Accumulate）32 位乘—累加指令将 Rm 和 Rs 中的值相乘，再将乘积加上第 3 个操作数，结果的最低 32 位保存到 Rd 中。

（1）指令的语法格式：

```
MLA{<cond>}{S}   <Rd>,<Rm>,<Rs>,<Rn>
```

（2）指令举例：

下面的指令完成 R1 = R2×R3 + 10 的操作。

```
MOV    R0, #0x0A
MLA    R1, R2, R3, R0
```

3．UMULL 指令

UMULL（Unsigned Multiply Long）为 64 位无符号乘法指令。它将 Rm 和 Rs 中的值做无符号数相乘，结果的低 32 位保存到 RsLo 中，高 32 位保存到 RdHi 中。

（1）指令的语法格式：

```
UMULL{<cond>}{S}   <RdLo>,<RdHi>,<Rm>,<Rs>
```

（2）指令举例：

下面的指令完成(R1，R0) = R5 × R8 操作。

```
UMULL   R0, R1, R5, R8;
```

4．UMLAL 指令

UMLAL（Unsigned Multiply Accumulate Long）为 64 位无符号长乘—累加指令。指令将 Rm 和 Rs 中的值做无符号数相乘，64 位乘积与 RdHi、RdLo 相加，结果的低 32 位保存到 RsLo 中，高 32 位保存到 RdHi 中。

（1）指令的语法格式：

```
UMALL{<cond>}{S}   <RdLo>,<RdHi>,<Rm>,<Rs>
```

（2）指令举例：

下面的指令完成(R1，R0) = R5 × R8+(R1，R0)操作。

```
UMLAL   R0, R1, R5,R8;
```

5．SMULL 指令

SMULL（Signed Multiply Long）为 64 位有符号长乘法指令。指令将 Rm 和 Rs 中的值做有符号数相乘，结果的低 32 位保存到 RsLo 中，高 32 位保存到 RdHi 中。

（1）指令的语法格式：

```
SMULL{<cond>}{S}   <RdLo>,<RdHi>,<Rm>,<Rs>
```

（2）指令举例：

下面的指令完成(R3，R2) = R7 × R6 操作。

```
SMULL   R2, R3, R7,R6;
```

6．SMLAL 指令

SMLAL（Signed Multiply Accumulate Long）为 64 位有符号长乘—累加指令。指令将 Rm 和 Rs 中的值做有符号数相乘，64 位乘积与 RdHi、RdLo 相加，结果的低 32 位保存到 RsLo 中，高 32 位保存到 RdHi 中。

（1）指令的语法格式：

```
SMLAL{<cond>}{S}   <RdLo>,<RdHi>,<Rm>,<Rs>
```

（2）指令举例：

下面的指令完成(R3，R2) = R7 × R6 +(R3，R2)操作。

```
SMLAL   R2, R3, R7,R6;
```

3.2.3　Load/Store 指令

Load/Store 内存访问指令在 ARM 寄存器和存储器之间传送数据。ARM 指令中有 3 种基本的数据传送指令。

（1）单寄存器 Load/Store 指令（Single Register），这些指令在 ARM 寄存器和存储器之间提供更灵活的单数据项传送方式。数据项可以是字节、16 位半字或 32 位字。

（2）多寄存器 Load/Store 内存访问指令。这些指令的灵活性比单寄存器传送指令差，但可以使大量的数据更有效地传送。它们用于进程的进入和退出、保存和恢复工作寄存器及复制存储器中的一块数据。

（3）单寄存器交换指令（Single Register Swap）。这些指令允许寄存器和存储器中的数值进行交换，在一条指令中有效地完成 Load/Store 操作。它们在用户级编程中很少用到。它的主要用途是在多处理器系统中实现信号量（Semaphores）的操作，以保证不会同时访问公用的数据结构。

（4）单寄存器的 Load/Store 指令，这种指令用于把单一的数据传入或者传出一个寄存器。支持的数据类型有字节（8 位）、半字（16 位）和字（32 位）。

1．单寄存器的 Load/Store 指令

如表 3-8 所示列出了所有单寄存器的 Load/Store 指令。

表 3-8　单寄存器 Load/Store 指令

指　令	作　用	操　作
LDR	把存储器中的一个字装入一个寄存器	Rd←mem32[address]
STR	将寄存器中的字保存到存储器	Rd→mem32[address]
LDRB	把一个字节装入一个寄存器	Rd←mem8[address]
STRB	将寄存器中的低 8 位字节保存到存储器	Rd→mem8[address]
LDRH	把一个半字装入一个寄存器	Rd←mem16[address]
STRH	将寄存器中的低 16 位半字保存到存储器	Rd→mem16[address]
LDRBT	用户模式下将一个字节装入寄存器	Rd←mem8[address] under user mode
STRBT	用户模式下将寄存器中的低 8 位字节保存到存储器	Rd→mem8[address] under user mode
LDRT	用户模式下把一个字装入一个寄存器	Rd←mem32[address]under user mode
STRT	用户模式下将存储器中的字保存到寄存器	Rd←mem32[address]]under user mode
LDRSB	把一个有符号字节装入一个寄存器	Rd←sign{mem8[address]}
LDRSH	把一个有符号半字装入一个寄存器	Rd←sign{mem16[address]}

1）LDR 指令

LDR 指令用于从内存中将一个 32 位的字读取到目标寄存器。

（1）指令的语法格式：

```
LDR{<cond>}  <Rd>,<addr_mode>
```

（2）指令举例：

```
LDR  R1,[R0,#0x12]        ;将 R0+12 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）
LDR  R1,[R0,R2]           ;将 R0+R2 地址的数据读出，保存到 R1 中（R0 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）
LDR  Rd,label             ;label 为程序标号，label 必须是当前指令的-4～4KB 范围内
LDR  Rd,[Rn],#0x04        ;Rn 的值用做传输数据的存储地址。在数据传送后，将偏移量 0x04 与 Rn 相加，结果
                          写回到 Rn 中。Rn 不允许是 R15
```

2）STR 指令

STR 指令用于将一个 32 位的字数据写入到指令中指定的内存单元。

（1）指令的语法格式：

```
STR{<cond>}  <Rd>,<addr_mode>
```

（2）指令举例：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，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。

① 变量访问。

```
NumCount  .equ  0x40003000          ;定义变量 NumCount
LDR  R0,=NumCount                   ;使用 LDR 伪指令装载 NumCount 的地址到 R0
LDR  R1,[R0]                        ;取出变量值
ADD  R1,R1,#1                       ;NumCount=NumCount+1
STR   R1,[R0]                       ;保存变量
```

② GPIO 设置。

```
GPIO—BASE  .equ   0xe0028000       ;定义 GPIO 寄存器的基地址
…
LDR    R0,=GPIO—BASE
LDR    R1,=0x00ffff00              ;将设置值放入寄存器
STR    R1,[R0,#0x0C]               ;IODIR=0x00ffff00,IOSET 的地址为 0xE0028004
```

③ 程序散转。

```
…
MOV  R2,R2,LSL #2                   ;功能号乘以 4，以便查表
LDR  PC,[PC,R2]                     ;查表取得对应功能子程序地址并跳转
NOP
FUN—TAB  .word   FUN—SUB0
         .word   FUN—SUB1
         .word   FUN—SUB2
          …
```

3）LDRB 指令

LDRB 指令根据 addr_mode 所确定的地址模式将一个 8 位字节读取到指令中的目标寄存器 Rd。

指令的语法格式：

```
LDR{<cond>}B  <Rd>, <addr_mode>
```

4）STRB 指令

STRB 指令从寄存器中取出指定的 8 位字节放入寄存器的低 8 位，并将寄存器的高位补 0。指令的语法格式：

```
STR{<cond>}B  <Rd>,<addr_mode>
```

5）LDRH 指令

LDRH 指令用于从内存中将一个 16 位的半字读取到目标寄存器。

如果指令的内存地址不是半字节对齐的，指令的执行结果不可预知。

指令的语法格式：

```
LDR{<cond>}H  <Rd>,<addr_mode>
```

6）STRH 指令

STRH 指令从寄存器中取出指定的 16 位半字放入寄存器的低 16 位，并将寄存器的高位补 0。

指令的语法格式：

```
STR{<cond>}H  <Rd>,<addr_mode>
```

2．多寄存器的 Load/Store 内存访问指令

多寄存器的 Load/Store 内存访问指令也叫批量加载/存储指令，它可以实现在一组寄存器和一块连续的内存单元之间传送数据。LDM 用于加载多个寄存器，STM 用于存储多个寄存器。多寄存器的 Load/Store 内存访问指令允许一条指令传送 16 个寄存器的任何子集或所有寄存器。多寄存器的 Load/Store 内存访问指令主要用于现场保护、数据复制和参数传递等。如表 3-9 所示列出了多寄存器的 Load/Store 内存访问指令。

表 3-9　多寄存器的 Load/Store 内存访问指令

指　令	作　用	操　作
LDM	装载多个寄存器	{Rd}*^{N}←mem32[start address+4*N]
STM	保存多个寄存器	{Rd}*^{N}→mem32[start address+4*N]

1）LDM 指令

LDM 指令将数据从连续的内存单元中读取到指令中指定的寄存器列表中的各寄存器中。当 PC 包含在 LDM 指令的寄存器列表中时，指令从内存中读取的字数据将被作为目标地址值，指令执行后程序将从目标地址处开始执行，从而实现了指令的跳转。

指令的语法格式：

```
LDM{<cond>}<addressing_mode>  <Rn>{!},  <registers>
```

寄存器 R0～R15 分别对应于指令编码中 bit[0]～bit[15]位。如果 Ri 存在于寄存器列表中，则相应的位等于 1，否则为 0。LDM 指令将数据从连续的内存单元中读取到指令中指定的寄存器列表中的各寄存器中。

指令的语法格式：

```
LDM{<cond>}<addressing_mode><Rn>,<registers_without_pc>
```

2）STM 指令

STM 指令将指令中寄存器列表中的各寄存器数值写入到连续的内存单元中。主要用于块数据的写入、数据栈操作及进入子程序时保存相关寄存器的操作。

指令的语法格式：

```
STM{<cond>}<addressing_mode>  <Rn>{!},  <registers>
```

STM 指令将指令中寄存器列表中的各寄存器数值写入到连续的内存单元中。主要用于块数据的写入、数据栈操作及进入子程序时保存相关寄存器等操作。

指令的语法格式：

```
STM{<cond>}<addressing_mode>  <Rn>,  <registers >^
```

3）数据传送指令应用

LDM/STM 批量加载/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{!},regist{^}
STM{cond}<模式>  Rn{!},regist{^}
```

LDM/STM 的主要用途有现场保护、数据复制和参数传递等。其模式有 8 种，其中前面 4 种用于数据块的传输，后面 4 种是堆栈操作，如下所示。

（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：空递增堆栈。

（7）FA：满递增堆栈。

（8）EA：空递增堆栈。

其中，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寄存器范围，使用“,”分开，如{R1，R2，R6～R9}，寄存器排列由小到大排列；“^”后缀不允许在用户模式下使用，只能在系统模式下使用。若在 LDM 指令用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 复制到 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/存储的是用户模式寄存器，而不是当前模式寄存器。

```
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，然后使用块复制寻址指令 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、STMEA/LDMEA 实现堆栈操作。数据是存储在基址寄存器的地址之上还是之下，地址是存储第一个值之前还是之后、增加还是减少，如表 3-10 所示。

表 3-10　多寄存器的 Load/Store 内存访问指令映射

		向 上 生 长		向 下 生 长	
		满	空	满	空
增加	之前	STMIB			LDMIB
		STMFA			LDMED
	之后		STMIA	LDMIA	
			STMEA	LDMFD	
增加	之前		LDMDB	STMDB	
			LDMEA	STMFD	
	之后	LDMDA			STMDA
		LDMFA			STMED

【举例】　使用 LDM/STM 进行数据复制。

```
LDR  R0,=SrcData              ;设置源数据地址
LDR  R1,=DstData              ;设置目标地址
LDMIA   R0,{R2～R9}           ;加载 8 字数据到寄存器 R2～R9
STMIA   R1,{R2～R9}           ;存储寄存器 R2～R9 到目标地址
```

【举例】　使用 LDM/STM 进行现场寄存器保护，常在子程序或异常处理使用。

```
SENDBYTE:
     STMFD     SP!,{R0～R7,LR}      ;寄存器压栈保护
      …
     BL        DELAY                ;调用 DELAY 子程序
      …
     LDMFD     SP!,{R0～R7,PC}      ;恢复寄存器，并返回
```

3．单数据交换指令

交换指令是 Load/Store 指令的一种特例，它把一个寄存器单元的内容与寄存器内容交换。交换指令是一个原子操作（Atomic Operation），也就是说，在连续的总线操作中读/写一个存储单元，在操作期间阻止其他任何指令对该存储单元的读/写。交换指令如表 3-11 所示。

表 3-11　交换指令 SWP

指　令	作　用	操　作
SWP	字交换	tmp=men32[Rn] mem32[Rn]=Rm Rd=tmp
SWPB	字节交换	tmp=men8[Rn] mem8[Rn]=Rm Rd=tmp

1）SWP 字交换指令

SWP 指令用于将内存中的一个字单元和一个指定寄存器的值相交换。操作过程如下：假设内存单元地址存放在寄存器<Rn>中，指令将<Rn>中的数据读取到目的寄存器 Rd 中，同时将另一个寄存器<Rm>的内容写入到该内存单元中。

当<Rd>和<Rm>为同一个寄存器时，指令交换该寄存器和内存单元的内容。

指令的语法格式：

```
SWP{<cond>}  <Rd>,<Rm>,[<Rn>]
```

2）SWPB 字节交换指令

SWPB 指令用于将内存中的一个字节单元和一个指定寄存器的低 8 位值相交换，操作过程如下：假设内存单元地址存放在寄存器<Rn>中，指令将<Rn>中的数据读取到目的寄存器 Rd 中，寄存器 Rd 的高 24 位设为 0，同时将另一个寄存器<Rm>的低 8 位内容写入到该内存字节单元中。当<Rd>和<Rm>为同一个寄存器时，指令交换该寄存器低 8 位内容和内存字节单元的内容。

指令的语法格式：

```
SWP{<cond>}B  <Rd>,<Rm>,[<Rn>]
```

3）交换指令 SWP 应用

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 的内容写到该内存单元中，使用 SWP 可实现信号量操作。

指令的语法格式：

```
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 为目的寄存器，存储从存储器中加载的数据，同时，Rm 中的数据将会被存储到存储器中。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。

SWP 指令举例：

```
SWP  R1,R1,[R0]          ;将 R1 的内容与 R0 指向的存储单元内容进行交换
SWPB  R1,R2,[R0]         ;将 R0 指向的存储单元内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），使用 SWP 指令可以方便地进行信号量操作
12C_SEM     .equ    0x40003000
            …
12C_SEM_WAIT:
        MOV     R0,#0
        LDR     R0,=12C_SEM
        SWP     R1,R1,[R0]        ;取出信号量，并将其设为 0
        CMP     R1,#0             ;判断是否有信号
        BEQ     12C_SEM_WAIT      ;若没有信号则等待
```

3.2.4　跳转指令

跳转（B）和跳转连接（BL）指令是改变指令执行顺序的标准方式。ARM 一般按照字地址顺序执行指令，需要时使用条件执行跳过某段指令。只要程序必须偏离顺序执行，就要使用控制流指令来修改程序计数器。尽管在特定情况下还有其他几种方式实现这个目的，但转移和转移连接指令是标准的方式。跳转指令改变程序的执行流程或者调用子程序。这种指令使得一个程序可以使用子程序、if-then-else 结构及循环。执行流程的改变迫使程序计数器（PC）指向一个新的地址，ARMv5 架构指令集包含的跳转指令如表 3-12 所示。

表 3-12　ARMv5 架构跳转指令

助 记 符	说　明	操　作
B	跳转指令	pc←label
BL	带返回的连接跳转	pc←label(lr←BL 后面的第一条指令)
BX	跳转并切换状态	pc←Rm&0xfffffffe, T←Rm&1
BLX	带返回的跳转并切换状态	pc←lable, T←1 pc←Rm&0xfffffffe, T←Rm&1 lr←BL 后面的第一条指令

另一种实现指令跳转的方式是通过直接向 PC 寄存器中写入目标地址值，实现在 4GB 地址空间中任意跳转，这种跳转指令又称为长跳转。如果在长跳转指令之前使用“MOV LR”或“MOV　PC”等指令，可以保存将来返回的地址值，也就实现了在 4GB 的地址空间中的子程序调用。

1．跳转指令 B 及带连接的跳转指令 BL

跳转指令 B 使程序跳转到指定的地址执行程序。带连接的跳转指令 BL 将下一条指令的地址复制到 R14（即返回地址连接寄存器 LR）寄存器中，然后跳转到指定地址运行程序。需要注意的是，这两条指令和目标地址处的指令都要属于 ARM 指令集。两条指令都可以根据 CPSR 中的条件标志位的值决定指令是否执行。

（1）指令的语法格式：

```
B{L}{<cond>}  <target_address>
```

BL 指令用于实现子程序调用。子程序的返回可以通过将 LR 寄存器的值复制到 PC 寄存器来实现。下面 3 种指令可以实现子程序返回。

① BX　R14（如果体系结构支持 BX 指令）。

② MOV　PC，R14。

③ 当子程序在入口处使用了压栈指令：

```
STMFD  R13!,{<registers>,R14}
```

可以使用指令：

```
LDMFD  R13!,{<registers>,PC}
```

将子程序返回地址放入 PC 中。

ARM 汇编器通过以下步骤计算指令编码中的 signed_immed_24。

① 将 PC 寄存器的值作为本跳转指令的基地址值。

② 从跳转的目标地址中减去上面所说的跳转的基地址，生成字节偏移量。由于 ARM 指令是字对齐的，该字节偏移量为 4 的倍数。

③ 当上面生成的字节偏移量超过−33 554 432～+33 554 430 时，不同的汇编器使用不同的代码产生策略。否则，将指令编码字中的 signed_immed_24 设置成上述字节偏移量的 bits[25:2]。

（2）程序举例：

① 程序跳转到 LABLE 标号处。

```
    B  LABLE ;
    ADD  R1,R2,#4
    ADD  R3,R2,#8
    SUB  R3,R3,R1
LABLE:
    SUB  R1,R2,#8
```

② 跳转到绝对地址 0x1234 处。

```
B  0x1234
```

③ 跳转到子程序 func 处执行，同时将当前 PC 值保存到 LR 中。

```
BL  func
```

④ 条件跳转：当 CPSR 寄存器中的 C 条件标志位为 1 时，程序跳转到标号 LABLE 处执行。

```
BCC  LABLE
```

⑤ 通过跳转指令建立一个无限循环。

```
LOOP:
    ADD  R1,R2,#4
    ADD  R3,R2,#8
    SUB  R3,R3,R1
    B  LOOP
```

⑥ 通过使用跳转使程序体循环 10 次。

```
    MOV  R0,#10
LOOP:
    SUBS  R0,#1
    BNE   LOOP
```

⑦ 条件子程序调用示例。

```
…
CMP  R0,#5                    ;如果 R0<5
BLLT  SUB1                    ;则调用
BLGE  SUB2                    ;否则调用 SUB2
```

2．带状态切换的跳转指令 BX

带状态切换的跳转指令（BX）使程序跳转到指令中指定的参数 Rm 指定的地址执行程序，Rm 的第 0 位复制到 CPSR 中 T 位，bit[31：1]移入 PC。若 Rm 的 bit[0]为 1，则跳转时自动将 CPSR 中的标志位 T 置位，即把目标地址的代码解释为 Thumb 代码；若 Rm 的位 bit[0]为 0，则跳转时自动将 CPSR 中的标志位 T 复位，即把目标地址代码解释为 ARM 代码。

（1）指令的语法格式：

```
BX{<cond>}  <Rm>
```

① 当 Rm[1：0]=0b10 时，指令的执行结果不可预知。因为在 ARM 状态下，指令是 4 字节对齐的。

② PC 可以作为 Rm 寄存器使用，但这种用法不推荐使用。当 PC 作为<Rm>使用时，指令“BX　PC”将程序跳转到当前指令下面第二条指令处执行。虽然这样跳转可以实现，但最好使用下面的指令完成这种跳转。

```
MOV   PC, PC
```

或

```
ADD   PC, PC, #0
```

（2）指令举例：

① 转移到 R0 中的地址，如果 R0[0]=1，则进入 Thumb 状态。

```
BX   R0;
```

② 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。

```
ADRL   R0,ThumbFun+1 ;
BX   R0;
```

3．带连接和状态切换的连接跳转指令 BLX

带连接和状态切换的跳转指令（Branch with Link Exchange，BLX）使用标号，用于使程序跳转到 Thumb 状态或从 Thumb 状态返回。该指令为无条件执行指令，并用分支寄存器的最低位来更新 CPSR 中的 T 位，将返回地址写入到连接寄存器 LR 中。

（1）语法格式：

```
BLX  <target_add>
```

其中，<target_add>为指令的跳转目标地址。该地址根据以下规则计算。

① 将指令中指定的 24 位偏移量进行符号扩展，形成 32 位立即数。

② 将结果左移两位。

③ 位 H（bit[24]）加到结果地址的第一位（bit[1]）。

④ 将结果累加进程序计数器（PC）中。

计算偏移量的工作一般由 ARM 汇编器来完成。这种形式的跳转指令只能实现−32～32MB 空间的跳转。左移两位形成字偏移量，然后将其累加进程序计数器（PC）中。这时，程序计数器的内容为 BX 指令地址加 8 字节。位 H（bit[24]）也加到结果地址的第一位（bit[1]），使目标地址成为半字地址，以执行接下来的 Thumb 指令。计算偏移量的工作一般由 ARM 汇编器来完成。这种形式的跳转指令只能实现−32～32MB 空间的跳转。

（2）指令的使用

① 从 Thumb 状态返回到 ARM 状态，使用 BX 指令。

```
BX  R14
```

② 可以在子程序的入口和出口增加栈操作指令。

```
PUSH  {<registers>,R14}
POP  {<registers>,PC}
```

3.2.5 状态操作指令

ARM 指令集提供了两条指令，可直接控制程序状态寄存器（Program State Register，PSR）。MRS 指令用于把 CPSR 或 SPSR 的值传送到一个寄存器；MSR 与之相反，把一个寄存器的内容传送到 CPSR 或 SPSR。这两条指令相结合，可用于对 CPSR 和 SPSR 进行读/写操作。程序状态寄存器指令如表 3-13 所示。

表 3-13 程序状态寄存器指令

指　令	作　用	操　作
MRS	把程序状态寄存器的值送到一个通用寄存器	Rd=SPR
MSR	把通用寄存器的值送到程序状态寄存器或把一个立即数送到程序状态字	PSR[field]=Rm 或 PSR[field]=immediate

在指令语法中可看到一个称为 fields 的项，它可以是控制（C）、扩展（X）、状态（S）及标志（F）的组合。

1. MRS

MRS 指令用于将程序状态寄存器的内容传送到通用寄存器中。

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。

（1）指令的语法格式：

```
MRS{cond}  Rd，PSR
```

其中，Rd 为目标寄存器，Rd 不允许为程序计数器（PC）。PSR 为 CPSR 或 SPSR。

（2）指令举例：

```
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 R1 中
```

MRS 指令读取 CPSR，可用来判断 ALU 的状态标志及 IRQ/FIQ 中断是否允许等；在异常处理程序中，读 SPSR 可指定进入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。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。

2．MSR

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。

（1）指令的语法格式：

```
MSR{cond}  PSR_field,#immed_8r
MSR{cond}  PSR_field,Rm
```

其中，PSR 是指 CPSR 或 SPSR。<fields>设置状态寄存器中需要操作的位。状态寄存器的 32 位可以分为 4 个 8 位的域（field）。bits[31:24]为条件标志位域，用 f 表示；bits[23:16]为状态位域，用 s 表示；bits[15:8]为扩展位域，用 x 表示；bits[7:0]为控制位域，用 c 表示；immed_8r 为要传送到状态寄存器指定域的立即数，8 位；Rm 为要传送到状态寄存器指定域的数据源寄存器。

（2）指令举例：

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

注意：

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 位控制位来实现 ARM 状态/Thumb 状态的切换，必须使用 BX 指令来完成处理器状态的切换（因为 BX 指令属转移指令，它会打断流水线状态，实现处理器状态的切换）。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及允许/禁止 IRQ/FIQ 中断等设置。

3．程序状态寄存器指令的应用

【举例】　使能 IRQ 中断。

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

【举例】　禁止 IRQ 中断。

```
DISABLE_IRQ:
```

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

【举例】 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,StackSvc
```

3.2.6 协处理器指令

ARM 体系结构允许通过增加协处理器来扩展指令集。最常用的协处理器是用于控制片上功能的系统协处理器。例如，控制 Cache 和存储管理单元的 cp15 寄存器。此外，还有用于浮点运算的浮点 ARM 协处理器，各生产商还可以根据需要开发自己的专用协处理器。

ARM 协处理器具有自己专用的寄存器组，它们的状态由控制 ARM 状态的指令的镜像指令来控制。程序的控制流指令由 ARM 处理器来处理，所有协处理器指令只能同数据处理和数据传送有关。按照 RISC 的 Load/Store 体系原则，数据的处理和传送指令是被清楚分开的，所以它们有不同的指令格式。ARM 处理器支持 16 个协处理器，在程序执行过程中，每个协处理器忽略 ARM 和其他协处理器指令。当一个协处理器硬件不能执行属于它的协处理器指令时，将产生一个未定义指令异常中断，在该异常中断处理过程中，可以通过软件仿真该硬件操作。如果一个系统中不包含向量浮点运算器，则可以选择浮点运算软件包来支持向量浮点运算。

ARM 协处理器可以部分地执行一条指令，然后产生中断。如除法运算除数为 0 和溢出，这样可以更好地处理运行时产生（run-time-generated）的异常。但是，指令的部分执行是由协处理器完成的，此过程对 ARM 来说是透明的。当 ARM 处理器重新获得执行时，它将从产生异常的指令处开始执行。对某一个协处理器来说，并不一定用到协处理器指令中的所有的域。具体协处理器如何定义和操作完全由协处理器的制造商自己决定，因此，ARM 协处理器指令中的协处理器寄存器的标识符及操作助记符也有各种不同的实现定义。程序员可以通过宏定义这些指令的语法格式。

ARM 协处理器指令可分为以下 3 类。

（1）协处理器数据操作。协处理器数据操作完全是协处理器内部操作，它完成协处理器寄存器的状态改变。如浮点加运算，在浮点协处理器中两个寄存器相加，结果放在第 3 个寄存器中。这类指令包括 CDP 指令。

（2）协处理器数据传送指令。这类指令从寄存器读取数据装入协处理器寄存器，或将协处理器寄存器的数据装入存储器。因为协处理器可以支持自己的数据类型，所以每个寄存器传送的字数与协处理器有关。ARM 处理器产生存储器地址，但传送的字节由协处理器控制。这类指令包括 LDC 指令和 STC 指令。

（3）协处理器寄存器传送指令。在某些情况下，需要 ARM 处理器和协处理器之间传送数据。如一个浮点运算协处理器，FIX 指令从协处理器寄存器取得浮点数据，将它转换为整数，并将整数传送到 ARM 寄存器中。经常需要用浮点比较产生的结果来影响控制流，因此，比较结果必须传送到 ARM 的 CPSR 中。这类协处理器寄存器传送指令包括 MCR 和 MRC。

如表 3-14 所示列出了所有协处理器处理指令。

表 3-14　协处理器指令

助　记　符	操　　作
CDP	协处理器数据操作
LDC	装载协处理器寄存器
MCR	从 ARM 寄存器传数据到协处理器寄存器
MRC	从协处理器寄存器传数据到 ARM 寄存器
STC	存储协处理器寄存器

下面简单介绍一下比较常用的 MCR 及 MRC 命令的用法：

1．ARM 寄存器到协处理器寄存器的数据传送指令 MCR

1）指令编码格式

ARM 寄存器到协处理器寄存器的数据传送指令 MCR（Move to Coprocessor from ARM Register）将 ARM 寄存器<Rd> 的值传送到协处理器寄存器 cp_num 中。如果没有协处理器执行指定操作，将产生未定义指令异常。指令的编码格式如图 3-2 所示。

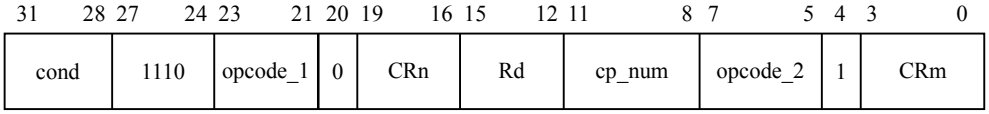

图 3-2　MCR 指令编码格式

2）指令的语法格式

```
MCR{<cond>}  <coproc>，<opcode_1>，<Rd>，<CRn>，<CRm> {<opcode_2>}
```

① <cond>

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway ））。

② <coproc>

指定协处理器的编号，标准的协处理器的名字为 p0、p1、…、p15。

③ <opcode_1>

指定协处理器执行的操作码，确定哪一个协处理器指令将被执行。

④ <Rd>

确定哪一个 ARM 寄存器的数值将被传送。如果程序计数器 PC 的值被传送，指令的执行结果不可预知。

⑤ <CRn>

确定包含第一个操作数的协处理器寄存器。

⑥ <CRm>

确定包含第二个操作数的协处理器寄存器。

⑦ <opcode_2>

指定协处理器执行的操作码，确定哪一个协处理器指令将被执行。通常与<opcode_1>配合使用。

3）指令举例

将 ARM 寄存器 r7 中的值传送到协处理器 p14 的寄存器 c7 中，第一操作数 opcode_1，第二操作数 opcode_2=6。

```
MCR   p14，1，r7，c7，c12，6
```

4）指令的使用

指令的编码格式中，bits[31：24]、bit[20]、bits[15：8]和 bit[4]为 ARM 体系结构定义。其他域由各生产商定义。硬件协处理器支持与否完全由生产商定义，某款 ARM 芯片中，是否支持协处理器或支持哪个协处理器与 ARM 版本无关。生产商可以选择实现部分协处理器指令或者完全不支持协处理器。

2．协处理器寄存器到 ARM 寄存器的数据传送指令 MRC

1）指令编码格式

协处理器寄存器到 ARM 寄存器的数据传送指令 MRC（Move to ARM register from Coprocessor）将协处理器 cp_num 的寄存器的值传送到 ARM 寄存器中。如果没有协处理器执行指定操作，将产生未定义指令异常。指令的编码格式如图 3-3 所示。

31 28	27 24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
cond	1110	opcode_1	1	CRn	Rd	cp_num	opcode_2	1	CRm

图 3-3　MRC 指令编码格式

2）指令的语法格式

```
MRC{<cond>}  <coproc>，<opcode_1>，<Rd>，<CRn>，<CRm>{，<opcode_2>}
```

为指令编码中的条件域。它指示指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway ））。

① <coproc>

指定协处理器的编号，标准的协处理器的名字为 p0、p1、…、p15。

② <opcode_1>

指定协处理器执行的操作码，确定哪一个协处理器指令将被执行。

③ <Rd>

确定哪一个 ARM 寄存器接受协处理器传送的数值。如果程序计数器 PC 被用做目的寄存器，指令的执行结果不可预知 。

④ <CRn>

确定包含第一个操作数的协处理器寄存器。

⑤ <CRm>

确定包含第二个操作数的协处理器寄存器。

⑥ <opcode_2>

指定协处理器执行的操作码，确定哪一个协处理器指令将被执行。通常与<opcode_1>配合使用。

3）指令举例

协处理器源寄存器为 c0 和 c2，目的寄存器为 ARM 寄存器 r4，第一操作数 opcode_1=5，第二操作数 opcode_2=3。

```
MRC  p15，5，r4，c0，c2，3
```

4）指令的使用

如果目的寄存器为程序计数器 r15，则程序状态字条件标准位根据传送数据的前 4bit 确定，后 28bit 被忽略。指令的编码格式中，bits[31:24]、bit[20]、bits[15:8]和 bit[4]为 ARM 体系结构定义。其他域由各生产商定义。

硬件协处理器支持与否完全由生产商定义，某款 ARM 芯片中，是否支持协处理器或支持哪个协处理器与 ARM 版本无关。生产商可以选择实现部分协处理器指令或者完全不支持协处理器。

如果协处理器必须完成一些内部工作来准备一个 32 位数据向 ARM 传送（例如，浮点 FIX 操作必须将浮点值转换为等效的定点值），那么这些工作必须在协处理器提交传送前进行。因此，在准备数据时经常需要协处理器握手信号处于“忙－等待”状态。ARM 可以在忙－等待时间内产生中断。如果它确实得以中断，那么它将暂停握手以服务中断。当它从中断服务程序返回时，将可能重试协处理器指令，但也可能不重试。例如，中断可能导致任务切换，无论哪种情况，协处理器必须给出一致结果，因此，在握手提交阶段之前的准备工作不允许改变处理器的可见状态。

如图 3-4 所示列出了 cp15 的各个寄存器的目的。

寄存器编号	基本作用	在 MMU 中的作用	在 PU 中的作用
0	ID 编码（只读）	ID 编码和 cache 类型	
1	控制位（可读写）	各种控制位	
2	存储保护和控制	地址转换表基地址	Cachability 的控制位
3	存储保护和控制	域访问控制位	Bufferablity 控制位
4	存储保护和控制	保留	保留
5	存储保护和控制	内存失效状态	访问权限控制位
6	存储保护和控制	内存失效地址	保护区域控制
7	高速缓存和写缓存	高速缓存和写缓存控制	
8	存储保护和控制	TLB 控制	保留
9	高速缓存和写缓存	高速缓存锁定	
10	存储保护和控制	TLB 锁定	保留
11	TCM ACCESS	NULL	NULL
12	异常向量表基地址	NULL	NULL
13	进程标识符	进程标识符	
14	保留		
15	因设计而异	因设计而异	因设计而异

图 3-4　cp15 寄存器列表

3.2.7　异常产生指令

ARM 指令集中提供了两条产生异常的指令，通过这两条指令可以用软件的方法实现异常。如表 3-15 所示为 ARM 异常产生指令。

表 3-15　ARM 异常产生指令

助　记　符	含　　义	操　　作
SWI	软中断指令	产生软中断，处理器进入管理模式
BKPT	断点中断指令	处理器产生软件断点

软件中断指令（Software Interrupt，SWI）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，在其他模式下也可以使用 SWI 指令，处理器同样切换到管理模式。

（1）指令的语法格式。

```
SWI{<cond>}   <immed_24>
```

（2）指令举例。

① 下面指令产生软中断，中断立即数为 0。

```
SWI   0;
```

② 产生软中断，中断立即数为 0x123456。

```
SWI   0x123456;
```

③ 使用 SWI 指令时，通常使用以下两种方法进行参数传递。

a.指令 24 位的立即数指定了用户请求的类型，中断服务程序的参数通过寄存器传递。下面的程序产生一个中断号为 12 的软中断。

```
MOV   R0,#34                    ;设置功能号为 34
SWI   12                        ;产生软中断，中断号为 12
```

b.另一种情况，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，参数通过其他寄存器传递。

下面的例子通过 R0 传递中断号，R1 传递中断的子功能号。

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

3.2.8　其他指令介绍

1．特殊指令介绍

Fmxr /Fmrx 指令是 NEON 下的扩展指令，在做浮点运算的时候，要先打开 vfp，因此需要用到 Fmxr 指令。

Fmxr：由 arm 寄存器将数据转移到协处理器中。

Fmrx：由协处理器转移到 arm 寄存器中。

如图 3-5 所示为浮点异常寄存器格式。

图 3-5　浮点异常寄存器格式

如表 3-16 所示为 FPEXC 的位定义。

表 3-16　FPEXC 的位定义

位	域	功能描述
[31]	EX	异常位，该位指定了有多少信息需要存储记录 SIMD/VFP 协处理器的状态
[30]	EN	NEON/VFP 使能位，设置 EN 位 1 则开启 NEON/VFP 协处理器，复位会将 EN 置 0
[29:0]		保留

FPEXC<浮点异常寄存器>，该寄存器是一个可控制 SIMD 及 VFP 的全局使能寄存器，并指定了这些扩展技术是如何记录的。

如果要打开 VFP 协处理器的话，可以用以下指令：

```
mov  r0, #0x40000000
     fmxr fpexc, r0    @ enable NEON and VFP coprocessor
```

2．CLZ 计算前导零数目

（1）语法格式：

```
CLZ {cond} Rd,Rm
```

其中：

- cond 是一个可选的条件代码。
- Rd 是目标寄存器。
- Rm 是操作数寄存器。

（2）用法：CLZ 指令对 Rm 中的值的前导零进行计数，并将结果返回到 Rd 中，如果未在源寄存器中设置任何位，则该结果值为 32，如果设置了位 31，则结果值为 0。

（3）条件标记：该指令不会更改标记。

（4）体系结构：ARMv5 以上。

（5）示例如图 3-6 所示。

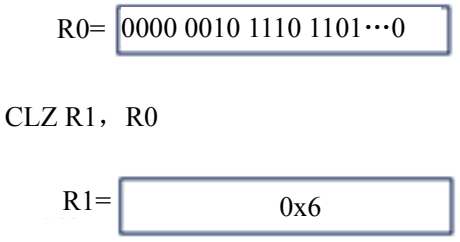

图 3-6　CLZ 例子

3．饱和指令介绍

这是用来设计饱和算法的一组指令，所谓饱和是指出现下列 3 种情况：

（1）对于有符号饱和运算，如果结果小于-2^n，则返回结果将为-2^n。

（2）对于无符号饱和运算，如果整数结果是负值，那么返回的结果将为 0。

（3）对于结果大于 2^n-1 的情况，则返回结果将为 2^n-1。

只要出现这情况，就称为饱和，并且饱和指令会设置 Q 标记，下面简单介绍一下 QADD 带符号加法。

QSUB：带符号减法。

QDADD：带符号加倍加法。

QDSUB：带符号加倍减法。

将结果饱和导入符号范围($-2^{31}\leqslant x \leqslant 2^{31}-1$)内。

（1）语法格式：

```
op{cond} {Rd} ,Rm,Rn
```

其中：

- op 是 QADD，QSUB,QDADD,QDSUB 之一。
- cond 是一个可选的条件代码。
- Rd 是目标寄存器。
- Rm, Rn 是存放操作数的寄存器（注：不要将 r15 用做 Rd,Rm 或 Rn）。

（2）用法如下：

- QADD 指令可将 Rm 和 Rn 中的值相加。
- QSUB 指令可从 Rm 中的值减去 Rn 中的值。
- QDADD/QDSUB 指令涉及并行指令，因此这里不多做讨论。

（3）条件标记：如果发生饱和，则这些指令设置 Q 标记，若要读取 Q 标记的状态，需要使用 MRS 指令。

（4）体系结构：该指令可用于 v5T-E 及 v6 或者更高版本的体系中。

（5）示例如下：

```
QADD r0 ,r1,r9
QSUBLT r9,r0,r1
```

3.3　本章小结

本章在第 2 章的基础上，介绍了 ARM 处理器的寻址方式及 ARM 处理器的指令集。ARM 处理器的寻址方式包括：数据处理指令寻址方式和内存访问指令寻址方式；ARM 处理器的指令集包括：数据操作指令、乘法指令、load/store 指令、跳转指令、状态操作指令、协处理器指令、异常产生指令。

3.4　思考题

1．用 ARM 汇编实现下面列出的操作：

a．r0=15

b．r0=r1/16（有符号数）

c．r1=r2*3

d．r0=-r0

2．BIC 指令的作用是什么？

3．执行 SWI 指令时会发生什么？

4．B、BL、BX 指令的区别是什么？

5．下面哪个数据可以作为数据操作指令的有效立即数：

a．0x101　b．0x1f8　c．0xf000000f　d．0x08000012　e．0x104

6．ARM 在哪些工作工作模式下可以修改 CPSR 寄存器？

7．写一个程序，判断 R0 的值，大于 0x50，则将 R1 的值减去 0x10，并把结果送给 R0。

8．编写一段 ARM 汇编程序，实现数据块复制，将 R0 指向的 8 个字的连续数据保存到 R1 指向的一段连续的内存单元。

第 4 章　ARM 汇编语言程序设计

在第 2、3 章中阐述的体系结构及指令集理论的基础上，本章主要介绍利用 ARM 汇编语言进行编程。ARM 编译器可以支持汇编语言、C/C++、汇编语言与 C/C++的混合编程等，本章将介绍汇编、C 相关的编程方法。

本章主要内容：

- GNU ARM 汇编伪操作。
- GNU ARM 汇编支持的伪指令。
- 汇编语言与 C 的混合编程。

4.1　GNU ARM 汇编器支持的伪操作

4.1.1　伪操作概述

在 ARM 汇编语言程序中，有一些特殊指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常称这些特殊指令助记符为伪操作标识符（directive），它们所完成的操作称为伪操作。伪操作在源程序中的作用是为了完成汇编程序做各种准备工作的，这些伪操作仅在汇编过程中起作用，一旦汇编结束，伪操作的使命就完成。

在 ARM 的汇编程序中，伪操作主要有符号定义伪操作、数据定义伪操作、汇编控制伪操作及其杂项伪操作等。

4.1.2　数据定义（Data Definition）伪操作

数据定义伪操作一般用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。常见的数据定义伪操作有.byte、.short、.long、.quad、.float、.string、.asciz、.ascii 和.rept。数据定义伪操作如下。

（1）伪指令名：

```
.byte
```

用途：单字节定义。

用法：

```
.byte 1,2,0b01,0x34,072,'s' ;
```

（2）伪指令名：

```
.short
```

用途：定义双字节数据，

用法：

```
.short 0x1234,60000 ;
```

（3）伪指令名：

```
.long
```

用途: 定义 4 字节数据。

用法：

```
.long 0x12345678,23876565
```

（4）伪指令名：

```
.quad:
```

用途：定义 8 字节。

用法：

```
.quad 0x1234567890abcd
```

（5）伪指令名：

```
.float
```

用途：定义浮点数。

用法：

```
.float 0f311971.693993751E-40
```

（6）伪指令名：

```
.string/.asciz/.ascii:
```

用途：定义多个字符串。

用法：

```
.string "abcd", "efgh", "hello!"
     .asciz "qwer", "sun", "world!"
     .ascii "welcome\0"（需要注意的是：.ascii 伪操作定义的字符串需要在每行末尾添加结尾字符'\0'）
```

（7）伪指令名：

```
.rept/.endr
```

用途：重复定义伪操作。

用法：

```
        .rept 3
    .byte 0x23
    .endr
```

（8）伪指令名:

```
.equ/.set
```

用途：赋值语句，.equ(.set) 变量名，表达式。

用法：

```
.equ abc 3 @ abc=3
```

4.1.3　汇编控制伪操作

汇编控制伪操作用于控制汇编程序的执行流程，常用的汇编控制伪操作包括以下几条。

1．.if、.else、.endif

1）语法格式

.if、.else、.endif 伪操作能根据条件的成立与否决定是否执行某个指令序列。当.if 后面的逻辑表达式为真，则执行.if 后的指令序列，否则执行.else 后的指令序列。其中，.else 及其后指令序列可以没有，此时，当.if 后面的逻辑表达式为真，则执行指令序列，否则继续执行后面的指令。

提示：

.if、.else、.endif 伪指令可以嵌套使用。

语法格式如下：

```
.if  logical-expressing
…
{.else
…}
.endif  logical-expression：
```

用于决定指令执行流程的逻辑表达式。

2）使用说明

当程序中有一段指令需要在满足一定条件时执行，使用该指令。该操作还有另一种形式。

```
.if  logical-expression
    Instruction
.elseif  logical-expression2
    Instructions
.endif
```

该形式避免了 if-else 形式的嵌套，使程序结构更加清晰、易读。

2．.macro、.endm

1）语法格式

.macro 伪操作可以将一段代码定义为一个整体，称为宏指令，然后就可以在程序中通过宏指令多次调用该段代码。其中，$标号在宏指令被展开时，标号会被替换为用户定义的符号。

宏操作可以使用一个或多个参数，当宏操作被展开时，这些参数被相应的值替换。宏操作的使用方式和功能与子程序有些相似，子程序可以提供模块化的程序设计、节省存储空间并提高运行速度。但在使用子程序结构时需要保护现场，从而增加了系统的开销，

因此，在代码较短且需要传递的参数较多时，可以使用宏操作代替子程序。

包含在.macro 和.endm 之间的指令序列称为宏定义体，在宏定义体的第一行应声明宏的原型（包含宏名、所需的参数），然后就可以在汇编程序中通过宏名来调用该指令序列。在源程序被编译时，汇编器将宏调用展开，用宏定义中的指令序列代替程序中的宏调用，并将实际参数的值传递给宏定义中的形式参数。

提示：

.macro、.endm 伪操作可以嵌套使用。

语法格式如下：

```
  .macro
{$label} macroname {$parameter{,$parameter}…}
  ;code
  .endm
```

（1）{$label}。

（2）$标号在宏指令被展开时，标号会被替换为用户定义的符号。通常，在一个符号前使用“$”表示该符号被汇编器编译时，使用相应的值代替该符号。

（3）Macroname：所定义的宏的名称。

（4）Parameter：宏指令的参数。当宏指令被展开时将被替换成相应的值，类似于函数中的参数。

2）使用说明

在子程序代码比较短，而需要传递的参数比较多的情况下可以使用宏汇编技术。

首先通过.macro 和.endm 伪操作定义宏，包括宏定义体代码。在.macro 伪操作之后的第一行声明宏的原型，其中包含该宏定义的名称及需要的参数。在汇编中可以通过该宏定义的名称来调用它。当源程序被编译时，汇编器将展开每个宏调用，用宏定义体代替源程序中宏定义的名称，并用实际参数值代替宏定义时的形式参数。

3）示例

示例如下：

```
.macro SHIFTLEFT a, b
.if \b < 0
MOV \a, \a, ASR #-\b
.exitm
.endif
MOV \a, \a, LSL #\b
.endm
```

3．.mexit

1）语法格式

.mexit 用于从宏定义中跳转出去。

2）用法

只需要在宏定义的代码中插入该指令即可。

```
.macro SHIFTLEFT a, b
.if \b < 0
 mov \a, \a, ASR #-\b
.exitm
.endif
 mov \a, \a, LSL #\b
.endm
```

4.1.4　杂项伪操作

ARM 汇编中还有一些其他的伪操作，在汇编程序中经常会被使用，包括以下几条。

```
.arm        .arm                    @ 定义以下代码使用 ARM 指令集编译
.code 32    .code 32                @作用同.arm
.code 16    .code 16                @作用同.thumb
.thumb      .thumb                  @定义以下代码使用 Thumb 指令集编译
.section    .section expr           @定义域中包含的段。expr 可以使.text,.data.,.bss
.text       .text {subsection}      @将定义符开始的代码编译到代码段或代码子段（subsection）
.data       .data {subsection}      @将定义符开始的代码编译到数据段或数据子段（subsection）
.bss        .bss {subsection}       @将变量存放到.bss 段或.bss 的子段（subsection）
.align      .align{alignment}{,fill}{,max} @通过用零或指定的数据进行填充来使当前位置与指定边界对齐
.org       .org offset{,expr}     @指定从当前地址加上 offset 开始存放代码，并且从当前地址到当前地址加
                                    上 offset 之间的内存单元，用零或指定的数据进行填充
```

4.2　ARM 汇编器支持的伪指令

ARM 汇编器支持 ARM 伪指令，这些伪指令在汇编阶段被翻译成 ARM 或者 Thumb（或 Thumb-2）指令（或指令序列）。ARM 伪指令包含 ADR、ADRL、LDR 等。

4.2.1　ADR 伪指令

1．语法格式

ADR 伪指令为小范围地址读取伪指令。ADR 伪指令将基于 PC 相对偏移地址或基于寄存器相对偏移地址值读取到寄存器中，当地址值是字节对齐时，取值范围为−255～255，当地址值是字对齐时，取值范围为−1020～1020。当地址值是 16 字节对齐时其取值范围更大。

语法格式如下：

```
ADR{cond}{.W}  register,label
```

1）cond

可选的指令执行条件。

2）.W

可选项。指定指令宽度（Thumb-2 指令集支持）。

3）register

目标寄存器。

4）label

基于 PC 或具有寄存器的表达式。

2．使用说明

ADR 伪指令被汇编器编译成一条指令。汇编器通常使用 ADD 指令或 SUB 指令来实现伪操作的地址装载功能。如果不能用一条指令来实现 ADR 伪指令的功能，汇编器将报告错误。

3．示例

示例如下。

```
        LDR     R4,=data+4*n        ;n 是汇编时产生的变量
        ; code
        MOV     pc,lr
data    .word     value0
        ; n-1 条 DCD 伪操作
        .word     valuen            ;所要装载入 R4 的值
        ;更多 DCD 伪操作
```

4.2.2 ADRL 伪指令

1．语法格式

ADRL 伪指令为中等范围地址读取伪指令。ADRL 伪指令将基于 PC 相对偏移的地址或基于寄存器相对偏移的地址值读取到寄存器中，当地址值是字节对齐时，取值范围为−64～64KB；当地址值是字对齐时，取值范围为−256～256KB。当地址值是 16 字节对齐时，其取值范围更大。在 32 位的 Thumb-2 指令中，地址取值范围到达−1～1MB。

语法格式如下：

```
ADRL{cond}  register,label
```

1）cond

可选的指令执行条件。

2）register

目标寄存器。

3）label

基于 PC 或具体寄存器的表达式。

2．使用说明

ADRL 伪指令与 ADR 伪指令相似，用于将基于 PC 相对偏移的地址或基于寄存器相对偏移的地址值读取到寄存器中。所不同的是，ADRL 伪指令比 ADR 伪指令可以读取更大范围的地址。这是因为在编译阶段，ADRL 伪指令被编译器换成两条指令。即使一条指令可以完成该操作，编译器也将产生两条指令，其中一条为多余指令。如果汇编器不能在两条指令内完成操作，将报告错误，中止编译。

4.2.3　LDR 伪指令

1．语法格式

LDR 伪指令装载一个 32 位的常数和一个地址到寄存器。

语法格式如下：

```
LDR{cond}{.W}  register,=[expr|label-expr]
```

1）Cond

可选的指令执行条件。

2）.W

可选项。指定指令宽度（Thumb-2 指令集支持）。

3）register

目标寄存器。

4）expr

32 位常量表达式。汇编器根据 expr 的取值情况，对 LDR 伪指令做如下处理。

① 当 expr 表示的地址值没有超过 MOV 指令或 MVN 指令的地址取值范围时，汇编器用一对 MOV 和 MVN 指令代替 LDR 指令。

② 当 expr 表示的指令地址值超过了 MOV 指令或 MVN 指令的地址范围时，汇编器将常数放入数据缓存池，同时用一条基于 PC 的 LDR 指令读取该常数。

5）label-expr

一个程序相关或声明为外部的表达式。汇编器将 label-expr 表达式的值放入数据缓存池，使用一条程序相关 LDR 指令将该值取出放入寄存器。

当 label-expr 被声明为外部的表达式时，汇编器将在目标文件中插入链接重定位伪操作，由链接器在链接时生成该地址。

2．使用说明

当要装载的常量超出了 MOV 指令或 MVN 指令的范围时，使用 LDR 指令。

由 LDR 指令装载的地址是绝对地址，即 PC 相关地址。

当要装载的数据不能由 MOV 指令或 MVN 指令直接装载时，该值要先放入数据缓存池，此时 LDR 伪指令处的 PC 值到数据缓存池中目标数据所在地址的偏移量有一定限制。ARM 或 32 位的 Thumb-2 指令中该范围是−4～4KB，Thumb 或 16 位的 Thumb-2 指令中该范围是 0～1KB。

3．示例

（1）将常数 0xff0 读到 R1 中。

```
LDR R3,=0xff0 ;
```

相当于下面的 ARM 指令：

```
MOV R3,#0xff0
```

（2）将常数 0xfff 读到 R1 中。

```
LDR R1,=0xfff ;
```

相当于下面的 ARM 指令：

```
LDR R1,[pc,offset_to_litpool]
…
litpool .word 0xfff
```

（3）将 place 标号地址读入 R1 中。

```
LDR R2,=place ;
```

相当于下面的 ARM 指令：

```
LDR R2,[pc,offset_to_litpool]
…
litpool .word place
```

4.3 GNU ARM 汇编语言的语句格式

在汇编语言程序设计中，经常使用各种符号代替地址（addresses）、变量（variables）和常量（constants）等，以增加程序的灵活性和可读性。尽管符号的命名由编程者决定，但并不是任意的，必须遵循以下的约定。

（1）符号区分大小写，同名的大、小写符号会被编译器认为是两个不同的符号。

（2）符号在其作用范围内必须唯一。

（3）自定义的符号名不能与系统的保留字相同。其中保留字包括系统内部变量（built in variable）和系统预定义（predefined symbol）的符号。

（4）符号名不应与指令或伪指令同名。如果要使用和指令或伪指令同名的符号要用双斜杠“//”将其括起来，如“//SSERT//”。

注意：

虽然符号被双斜杠括起来，但双斜杠并非符号名的一部分。

（5）局部标号以数字开头，其他的符号都不能以数字开头。

1．变量（variable）

程序中的变量是指其值在程序的运行过程中可以改变的量。ARM（Thumb）汇编程序所支持的变量有 3 种。

- 数字变量（numeric）。
- 逻辑变量（logical）。
- 字符串变量（string）。

数字变量用于在程序的运行中保存数字值，但注意数字值的大小不应超出数字变量所能表示的范围。

逻辑变量用于在程序的运行中保存逻辑值，逻辑值只有两种取值情况：真（{TURE}）和假（{FALSE}）。

字符串变量用于在程序的运行中保存一个字符串，注意字符串的长度不应超出字符串变量所能表示的范围。

在 ARM（Thumb）汇编语言程序设计中，可使用 GBLA、GBLL、GBLS 伪指令声明全局变量，使用 LCLA、LCLL、LCLS 伪指令声明局部变量，可使用 SETA、SETL 和 SETS 对其进行初始化。

2．常量（constants）

程序中的常量是指其值在程序的运行过程中不能被改变的量。ARM（Thumb）汇编程序所支持的常量有数字常量、逻辑常量和字符串常量。

数字常量一般为 32 位的整数，当作为无符号数时，其取值范围为 0～232−1，当作为有符号数时，其取值范围为−231～231−1。汇编器认为−n 和 232−n 是相等的。对于关系操作，如比较两个数的大小，汇编器将其操作数看做无符号的数，也就是说“0>−1”，对汇编器来说取值为“假（{FLASE}）”。

逻辑常量只有两种取值情况，真或假。

字符串常量为一个固定的字符串，一般用于程序运行时的信息提示。

3．程序中的变量代换

汇编语言中的变量可以作为一整行出现在汇编程序中，也可以作为行的一部分使用。如果在数字变量前面有一个代换操作符“$”，编译器会将该数字变量的值转换为十六进制的字符串，并将该十六进制的字符串代换“$”后的数字变量。

如果在逻辑变量前面有一个代换操作符“$”，编译器会将该逻辑变量代换为它的取值（真或假）。如果在字符串变量前面有一个代换操作符“$”，编译器会将该字符串变量的值代换“$”后的字符串变量。如果程序中需要字符“$”，则可以用“$$”来表示。汇编器将不进行变量替换，而是将“$$”作为“$”。

4．程序标号（label）

在 ARM 汇编中，标号代表一个地址，段内标号的地址在汇编时确定，而段外标号地址值在链接时确定。根据标号的生成方式，程序标号分为以下 3 种。

- 程序相关标号（Program-relative labels）。
- 寄存器相关标号（Register-relative labels）。
- 绝对地址（Absolute address）。

1）程序相关标号

程序相关标号指位于目标指令前的标号或程序中的数据定义伪操作前的标号。这种标号在汇编时将被处理成 PC 值加上或减去一个数字常量。它常用于表示跳转指令的目标地址或代码段中所嵌入的少量数据。

2）寄存器相关地址

这种标号在汇编时将被处理成寄存器的值加上或减去一个数字常量。它常被用于访问数据段中的数据。这种基于寄存器的标号通常用 MAP 和 FIELD 伪操作定义，也可以用 EQU 伪操作定义。

3）绝对地址

绝对地址是一个 32 位的数字量，使用它可以直接寻址整个内存空间。

5．局部标号

局部标号是一个 0～99 之间的十进制数字，可重复定义。局部标号后面可以紧接一个通常表示该局部变量作用范围的符号。局部变量的作用范围为当前段，也可以用伪操作 ROUT 来定义局部标号的作用范围。

局部标号在子程序或程序循环中常被用到，也可以配合宏定义伪操作（.MACRO 和.MEND）来使程序结构更加合理。在同一个段中，可以使用相同的数字命名不同的局部变量。默认情况下，汇编器会寻址最近的变量。也可以通过汇编器命令选项来改变搜索顺序。

4.4 ARM 汇编语言的程序结构

4.4.1 汇编语言的程序格式

在 ARM（Thumb）汇编语言程序中可以使用.section 来进行分段，其中每一个段用段名或者文件结尾为结束，这些段使用默认的标志，如 a 为允许段，w 为可写段，x 为执行段。

在一个段中，我们可以定义下列的子段：

- .text
- .data
- .bss
- .sdata
- .sbss

由此我们可知道，段可以分为代码段、数据段及其他存储用的段，.text（正文段）包含程序的指令代码；.data(数据段)包含固定的数据，如常量、字符串；.bss（未初始化数据段）包含未初始化的变量、数组等，当程序较长时，可以分割为多个代码段和数据段，多个段在程序编译链接时最终形成一个可执行的映像文件。

```
.section.data
< initialized data here>
.section .bss
< uninitialized data here>
.section .text
.globl _start
```

```
_start:
<instruction code goes here>
```

4.4.2　汇编语言子程序调用

在 ARM 汇编语言程序中，子程序的调用一般是通过 BL 指令来实现的。在程序中，使用指令“BL 子程序”名即可完成子程序的调用。

该指令在执行时完成如下操作：将子程序的返回地址存放在连接寄存器 LR 中，同时将程序计数器 PC 指向子程序的入口点。当子程序执行完毕需要返回调用处时，只需要将存放在 LR 中的返回地址重新复制给程序计数器 PC 即可。在调用子程序的同时，也可以完成参数的传递和从子程序返回运算的结果，通常可以使用寄存器 R0～R3 完成。

注意：
同编译器编译的代码间的相互调用，要遵循 AAPCS（ARM Architecture）。详见 ARM 编译工具手册。

以下是使用 BL 指令调用子程序的汇编语言源程序的基本结构：

```
.text
.global _start
_start:
LDR         R0, =0x3FF5000
LDR         R1, 0xFF
STR         R1, [R0]
LDR         R0, =0x3FF5008
LDR         R1, 0x01
STR         R1, [R0]
BL          PRINT_TEXT
…
PRINT_TEXT:
…
      MOV       PC, BL
…
```

4.4.3　过程调用标准 AAPCS

为了使不同编译器编译的程序之间能够相互调用，必须为子程序间的调用规定一定的规则。AAPCS 就是这样一个标准。所谓 AAPCS，其英文全称为 Procedure Call Standard for the ARM Architecture（AAPCS），即 ARM 体系结构过程调用标准。它是 ABI（Application Binary Interface（ABI）for the ARM Architecture (base standard) [BSABI]）标准的一部分。可以使用“--apcs”选项告诉编译器将源代码编译成符号 AAPCS 调用标准的目标代码。

注意：
使用“ --apcs ”选项并不影响代码的产生，编译器只是在各段中放置相应的属性，标识用户选定的 AAPCS 属性。

1．AAPCS 相关的编译/汇编选项

- none：指定输入文件不使用 AAPCS 规则。
- /interwork：指定输入文件符合 ARM/Thumb 交互标准。
- /nointerwork：指定输入文件不能使用 ARM/Thumb 交互。这是编译器默认选项。
- /ropi：指定输入文件是位置无关只读文件。
- /noropi：指定输入文件是非位置无关只读文件。这是编译器默认选项。
- /pic：同/ropi。
- /nopic：同/noropi。
- /rwpi：指定输入文件是位置无关可读可写文件。
- /norwpi：指定输入文件是非位置无关可读可写文件。
- /pid：同/rwpi。
- /nopid：同/norwpi。
- /fpic：指定输入文件编译成位置无关只读代码。代码中地址是 FPIC 地址。
- /swstackcheck：编译过程中对输入文件使用堆栈检测。
- /noswstackcheck：编译过程中对输入文件不使用堆栈检测。这是编译器默认选项。
- /swstna：如果汇编程序对于是否进行数据栈检查无所谓，而与该汇编程序连接的其他程序指定了选项/swst 或选项/noswst，这时该汇编程序使用选项/swstna。

2．ARM 寄存器使用规则

AAPCS 中定义了 ARM 寄存器使用规则如下：

子程序间通过寄存器 R0、R1、R2、 R3 来传递参数。如果参数多于 4 个，则多出的部分用堆栈传递。被调用的子程序在返回前无须恢复寄存器 R0-R3 的内容。

在子程序中，使用寄存器 R4-R11 来保存局部变量。如果在子程序中使用到了寄存器 R4-R11 中的某些寄存器，子程序进入时必须保存这些寄存器的值，在返回前必须恢复这些寄存器的值；对于子程序中没有用到的寄存器则不必进行这些操作。在 Thumb 程序中，通常只能使用寄存器 R4-R7 来保存局部变量。

寄存器 R12 用做子程序间 scratch 寄存器（用于保存 SP，在函数返回时使用该寄存器出栈），记作 ip。在子程序间的连接代码段中常有这种使用规则。

寄存器 R13 用做数据栈指针，记作 sp。在子程序中寄存器 R13 不能用做其他用途。寄存器 sp 在进入子程序时的值和退出子程序时的值必须相等。

寄存器 R14 称为连接寄存器，记作 lr。它用于保存子程序的返回地址。如果在子程序中保存了返回地址，寄存器 R14 则可以用做其他用途。

寄存器 R15 是程序计数器，记作 pc。它不能用做其他用途。

ARM 寄存器在函数调用过程中的保护规则，如图 4-1 所示。

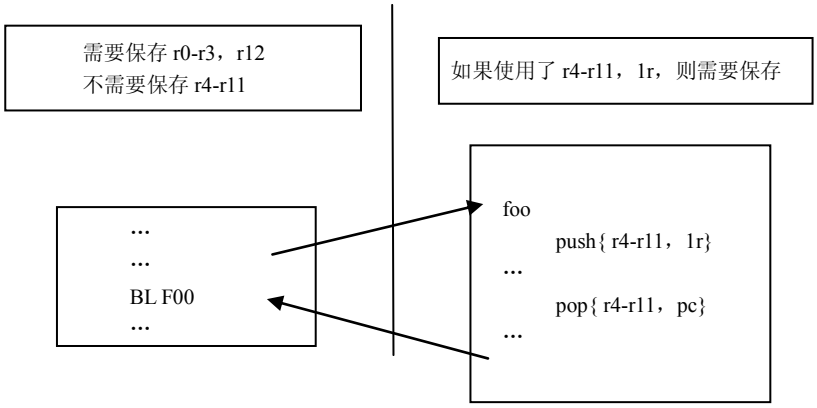

图 4-1　ARM 寄存器在函数调用中的保护规则

4.4.4　汇编语言程序设计举例

通过组合使用条件执行和条件标志设置，可简单地实现分支语句，不需要任何分支指令。这样可以改善性能，因为分支指令会占用较多的周期数；同时这样做也可以减小代码尺寸，提高代码密度。

下面是一段 C 语言程序，该程序实现了著名的 Euclid 最大公约数算法：

```
int gcd(int a, int b)
{
     while (a != b)
       {
          if (a > b)
                 a = a - b;
          else
                 b = b - a;
       }
     return a;
}
```

用 ARM 汇编语言重写来重写这个例子，如下所示。

```
Code1：
Gcd：
    CMP     r0, r1
        BEQ     end
        BLT     less
        SUB     r0, r0, r1
        B       gcd
Less：
        SUB     r1, r1, r0
        B       gcd
```

充分地利用条件执行修改上面的例子，得到 Code2。

```
Code2：
Gcd：
       CMP     r0, r1
```

```
        SUBGT   r0, r0, r1
        SUBLT   r1, r1, r0
        BNE     gcd
```

两段代码的比较如下。

- Code1：仅使用了分支指令。
- Code2：充分利用了 ARM 指令条件执行的特点，仅使用了 4 条指令就完成了全部算法。这对提供程序的代码密度和执行速度十分有帮助。

事实上，分支指令十分影响处理器的速度。每次执行分支指令，处理器都会排空流水线，重新装载指令。

4.5 汇编语言与 C 语言的混合编程

在 C 代码中实现汇编语言的方法有内联汇编和内嵌汇编两种，使用它们可以在 C 程序中实现 C 语言不能完成的一些工作。例如，在下面几种情况中必须使用内联汇编或嵌入型汇编。

4.5.1 GNU ARM 内联汇编

1．内联汇编语法

本小节简单介绍 GNU 风格的 ARM 内联汇编语法要点：

（1）格式

格式如下：

```
asm volatile ("asm code": output: input: changed);
```

必须以“;”结尾，不管有多长对 C 都只是一条语句。

（2）asm 内嵌汇编关键字

volatile：告诉编译器不要优化内嵌汇编，如果想优化可以不加。

（3）ANSI C 规范的关键字

ANSI C 规范的关键字如下。

```
__asm__
__volatile__          //前面和后面都有两个下画线，它们之间没有空格
```

如果后面部分没有内容，“:”可以省略，前面或中间的不能省略“:”，没有 asm code 也不可以省略“""”，没有 changed 必须省略“:”。

2．汇编代码

汇编必须放在一个字符串内，但是字符串中间是不能直接按回车键换行的，可以写成多个字符串，只要字符串之间不加任何符号编译完后就会变成一个字符串：

```
    "mov r0,r0\n\t"          //指令之间必须要换行，\t 可以不加，只是为了在汇编文件中的指令格式对齐
```

```
    "mov r1,r1\n\t"
    "mov r2,r2"
```

字符串内不是只能放指令，可以放一些标签、变量、循环、宏等，还可以把内嵌汇编放在 C 函数外面，用内嵌汇编定义函数、变量、段等，总之就跟直接在写汇编文件一样在 C 函数外面定义内嵌汇编时不能加 volatile：output：input：changed。

注意：

编译器不检查 asm code 的内容是否合法，直接交给汇编器

3．output（ASM --> C）和 input（C --> ASM）

（1）指定输出值：

```
__asm__ __volatile__ (
      "asm code"
      :“constraint”(variable)
);
```

① constraint 定义 variable 的存放位置：

```
r：         使用任何可用的通用寄存器
m：         使用变量的内存地址
```

② output 修饰符：

```
+：         可读可写
=：         只写
&：         该输出操作数不能使用输入部分使用过的寄存器，只能 +& 或 =& 方式使用
```

（2）指定输入值：

```
__asm__ __volatile__ (
      "asm code"
      :
      :“constraint”(variable / immediate)
);
constraint 定义 variable / immediate 的存放位置：
r：         使用任何可用的通用寄存器（变量和立即数都可以）
m：         使用变量的内存地址（不能用立即数）
i     :     使用立即数（不能用变量）
```

（3）使用占位符：

```
int a = 100,b = 200;
int result;
__asm__ __volatile__ (
       “mov    %0,%3\n\t”    //mov    r3,#123   %0 代表 result,%3 代表 123(编译器会自动加 # 号)
        “ldr   r0,%1\n\t”    //ldr    r0,[fp, #-12]      %1 代表 a 的地址
       “ldr    r1,%2\n\t”    //ldr    r1,[fp, #-16]      %2 代表 b 的地址
       “str    r0,%2\n\t” /*str    r0,[fp, #-16]因为%1 和%2 是地址所以只能用 ldr 或 str 指令*/
       “str    r1,%1\n\t” /*str    r1,[fp, #-12]如果用错指令编译时不会报错，要到汇编时才会报错*/
          :“=r” (result), “+m” (a), “+m” (b)     /*out1 是%0，out2 是%1，...，outN 是%N-1*/
          :“i” (123)                             /*in1 是%N，in2 是%N+1，...*/
          );
```

（4）引用占位符：

```
int num = 100;
__asm__ __volatile__ (
       "add    %0,%1,#100\n\t"
       : "=r"(a)
        : "0"(a)              //"0"是零，即%0，引用时不可以加 %，只能 input 引用 output
);                            //引用是为了更能分清输出输入部分
```

（5）& 修饰符：

```
int num;
__asm__ __volatile__ (        //mov     r3, #123          //编译器自动加的指令
      "mov    %0,%1\n\t"      //mov     r3,r3             //输入和输出使用相同的寄存器
        : "=r"(num)
        : "r"(123)
);
int num;
__asm__ __volatile__ (
 //mov     r3, #123
    "mov    %0,%1\n\t"        //mov     r2,r3             //加了&后输入和输出的寄存器不一样了
     : "=&r"(num)             //mov     r3, r2            //编译器自动加的指令
     : "r"(123)
);
```

4．内联汇编示例

下面通过一个例子进一步了解内联汇编的语法。该例子实现了位交换。

```
#include <stdio.h>
unsigned long ByteSwap(unsigned long val)
{
     int ch;
asm volatile (
  "eor r3, %1, %1, ror #16\n\t"
  "bic r3, r3, #0x00FF0000\n\t"
  "mov %0, %1, ror #8\n\t"
  "eor %0, %0, r3, lsr #8"
  : "=r" (val)
 : "0"(val)
  : "r3"
);
}
int main(void)
{
     unsigned long test_a = 0x1234,result;
     result = ByteSwap(test_a);
     printf("Result:%d\r\n", result);
     return 0;
}
```

4.5.2　混合编程调用举例

汇编程序、C 程序相互调用时，要特别注意遵守相应的 AAPCS 规则。下面一些例子具体说明了在这些混合调用中应注意遵守的 AAPCS 规则。

1．从 C 程序调用汇编语言

下面的程序显示了如何在 C 程序中调用汇编语言子程序，该段代码实现了将一个字符串复制到另一个字符串。

```
#include <stdio.h>
extern void strcopy(char *d, const char *s);
int main()
{
 const char *srcstr = "First string - source ";
    char dststr[] = "Second string - destination ";
/* 下面将 dststr 作为数组进行操作 */
    printf("Before copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    strcopy(dststr,srcstr);
    printf("After copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    return(0);
}
```

下面为调用的汇编程序。

```
.global strcopy
strcopy:                        ;R0 指向目的字符串
                                ;R1 指向源字符串
    LDRB R2, [R1],#1            ;加载字节并更新源字符串指针地址
    STRB R2, [R0],#1            ;存储字节并更新目的字符串指针地址
    CMP R2, #0                  ;判断是否为字符串结尾
    BNE strcopy                 ;如果不是，程序跳转到 strcopy 继续复制
    MOV pc,lr                   ;程序返回
```

2．从汇编语言调用 C 程序

下面的例子显示了如何从汇编语言调用 C 程序。

下面的子程序段定义了 C 语言函数。

```
int g(int a, int b, int c, int d, int e)
{
     return a + b + c + d + e;
}
```

下面的程序段显示了汇编语言调用。假设程序进入 f 时，R0 中的值为 i。

```
  ; int f(int i) { return g(i, 2*i, 3*i, 4*i, 5*i); }
.text
.global  _start
_start:
  STR lr, [sp, #-4]!          // 保存返回地址 lr
  ADD R1, R0, R0              // 计算 2*i(第 2 个参数)
  ADD R2, R1, R0              // 计算 3*i(第 3 个参数)
  ADD R3, R1, R2              // 计算 5*i
  STR R3, [sp, #-4]!          // 第 5 个参数通过堆栈传递
  ADD R3, R1, R1              // 计算 4*i(第 4 个参数)
  BL g                        // 调用 C 程序
  ADD sp, sp, #4              // 从堆栈中删除第 5 个参数
  LDR pc, [sp], #4            // 返回
```

4.6 本章小结

本章介绍了 ARM 程序设计的过程与方法，包括汇编语言编程、伪指令的使用、汇编器的使用、汇编语言和 C 语言混合编程等内容。这些内容是嵌入式编程的基础，希望读者掌握。

4.7 思考题

1．在 GNU 风格的 ARM 汇编中如何定义一个全局的数字变量？
2．AAPCS 中规定的 ARM 寄存器的使用规则是什么？
3．什么是内联汇编？什么是嵌入型汇编？两者之间的区别是什么？
4．汇编代码中如何调用 C 代码中定义的函数？
5．请使用 GNU-ARM 与 C 混合编程，实现一个打印语句的例子。

第 5 章　ARM 开发及环境搭建

学习 ARM 汇编的第一件事就是搭建编程环境，如今有非常多的 IDE 及调试软件/仿真硬件，因此这里笔者将提供一些方案给予学习者。大家知道，ARM 公司在前一个开发环境 ADS5.2（不再提供升级）后，推出了 Realview 系列开发环境。其中 Realview MDK 环境以其优越的性价比得到了快速的推广。但本书以 GNU-ARM 汇编风格作为基础，所以会主要介绍在 GNU-ARM 下如何编写 ARM 汇编程序并进行调试。

本章主要介绍它的使用、配置方法，内容主要有：

- 仿真器简介。
- 主流编程环境介绍（Eclipse，MDK）。
- FS-JTAG 的使用方法。

5.1　仿真器简介

5.1.1　FS-JTAG 仿真器介绍

了解行业和相关技术的人都知道，功能完善的 ARM 仿真器和软件调试环境对于学习 ARM 处理器的工作原理和核心知识来说至关重要。由于之前多年的技术发展和行业实践，针对 Cortex-Mx、ARM7、ARM9 及 ARM11 系列处理器，市场上都已经有很多成熟的、价廉物美的仿真器可供选择。而对于目前最新流行的 ARM 应用处理器 Cortex-A8 系列来说，业内的技术工程师们却很难找到价格合适、功能完善的仿真器。国外动辄几千甚至上万美元的价格，无疑阻碍了广大学习者的积极性，为此，华清远见研发中心为了推进 Cortex-A8 ARM 处理器的教学，提高合作企业及合作院校广大技术爱好者和培训学员的学习效率，最新生产研发出 FS-JTAG 仿真器，该款仿真器可以仿真 Cortex-M3、ARM7、ARM9、ARM11、Cortex-A8 等多个 ARM 处理器系列。

如果需要专业一些的调试，则应该选择 ULINK、TRACE 32 这类专业级的仿真器，操作简单，调试功能强大，但价格昂贵。

下面逐一介绍一些常用的仿真器：

（1）FS-JTAG 仿真器（如图 5-1 所示）是一款基于开源的 OpenOcd 接口的仿真器，外观和 JLINK 相同，有着很全的调试功能，再加上 Eclipse 这样强大的集成开发环境，使得它同样能成为工程师的首选，它有着如下的硬件特点。

图 5-1　FS-JTAG 仿真器

① USB 特性：USB2.0 全速接口、USB 电源供电。

② JTAG 特性：IEEE 1149.1 标准。

（2）配套的软件有如下特点：

① Eclipse 集成开发环境：提供实时调试功能，如单步、全速运行、复位、软/硬断点、跳转动态查看寄存器和存储器、变量观察。

② 源码级别调试器 OpenOcd，开源，并且提供良好的交互界面。

③ 支持烧写 nor/nand Flash。

5.1.2　ULINK 介绍

ULINK 是 Keil 公司提供的 USB-JTAG 接口仿真器，目前最新的版本是 2.0。它支持诸多芯片厂商的 8051、ARM7、ARM9、Cortex-M3、Infineon C16x、Infineon XC16x、Infineon XC8xx、STMicroelectronics μPSD 等多个系列的处理器。ULINK2 仿真器如图 5-2 所示，由 PC 的 USB 接口提供电源。ULINK2 不仅包含了 ULINK USB-JTAG 适配器具有的所有特点，还增加了串行线调试（SWD）支持，以及返回时钟支持和实时代理功能。

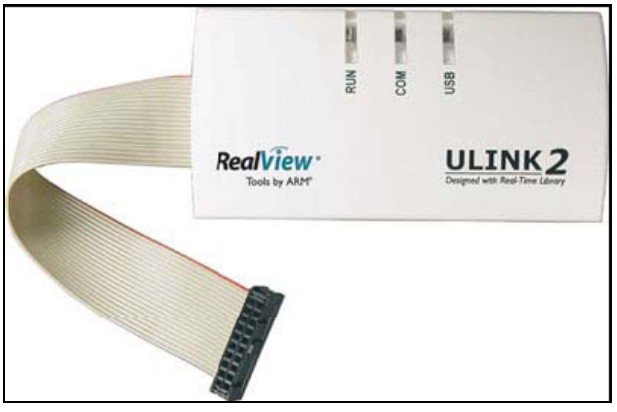

图 5-2　ULINK2 仿真器

ULINK2 的主要功能如下：

- 下载目标程序。
- 检查内存和寄存器。
- 片上调试，整个程序的单步执行。
- 插入多个断点。
- 运行实时程序。
- 对 Flash 存储器进行编程。

ULINK2 的新特点包括：

- 标准 Windows USB 驱动支持，也就是 ULINK2 可即插即用。
- 支持基于 ARM Cortex-M3 的串行线调试。
- 支持程序运行期间的存储器读/写、终端仿真和串行调试输出。
- 支持 10/20 针连接器。

本书将使用 Eclipse 与 FS-JTAG 的搭配方式，所以此处不详细介绍 ULINK2 的使用方法。

5.2　开发环境搭建

Eclipse for ARM 是借用开源软件 Eclipse 的工程管理工具，嵌入 GNU 工具集，使之能够开发 ARM 公司 Cortex-A 系列的 CPU，这里使用 Eclipse for ARM 作为开发软件。在开发箱中的配套光盘中，打开 FS-JTAG 这个目录，可以看如图 5-3 所示的光盘资料。

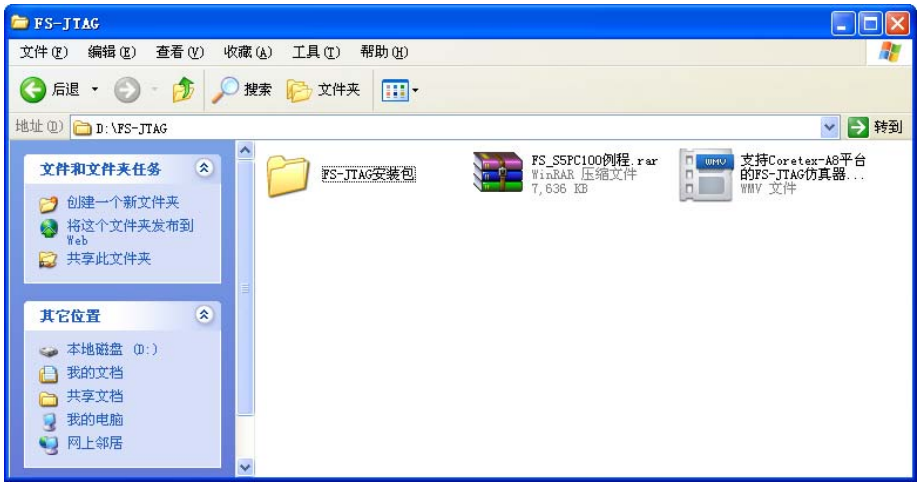

图 5-3　光盘资料

进入 FS-JTAG 安装包，可以看到如图 5-4 所示的安装软件及 USB 驱动，后面的安装步骤中所用到的软件都在这个目录下。

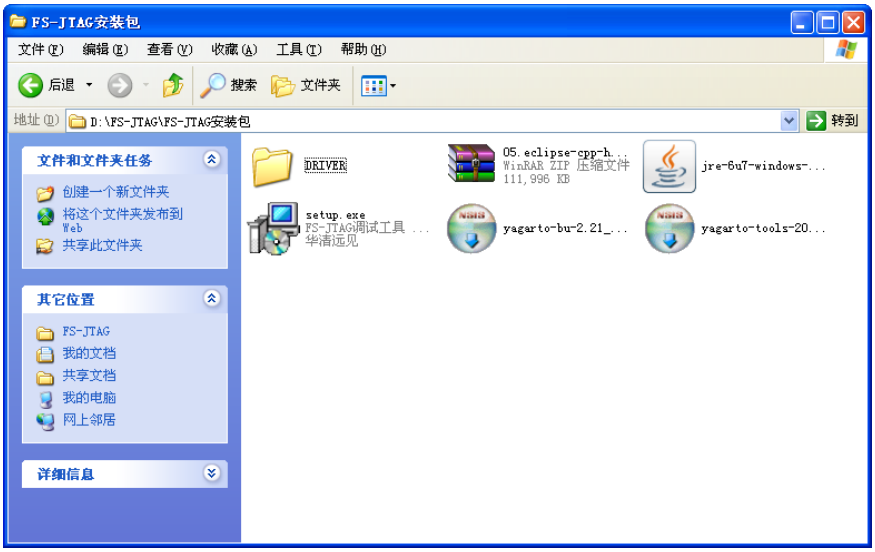

图 5-4　FS-JTAG 安装包

（1）安装 gcc 编译工具：yagarto-bu-2.21_gcc-4.6.2-c-c++_nl-1.19.0_gdb-7.3.1_eabi_20111119.exe。

（2）安装 tools 工具：yagarto-tools-20100703-setup.exe。

（3）安装 FS-JTAG 工具：Setup.exe。

（4）安装 jre-6u7-windows-i586-p-s.exe。

（5）解压 Eclipse 压缩包。

（6）安装 FS-JTAG 驱动：把 FS-JTAG 接入计算机 USB 口，会提示发现新硬件（如图 5-5 所示），选择从列表或指定位置安装，然后单击“下一步”按钮。

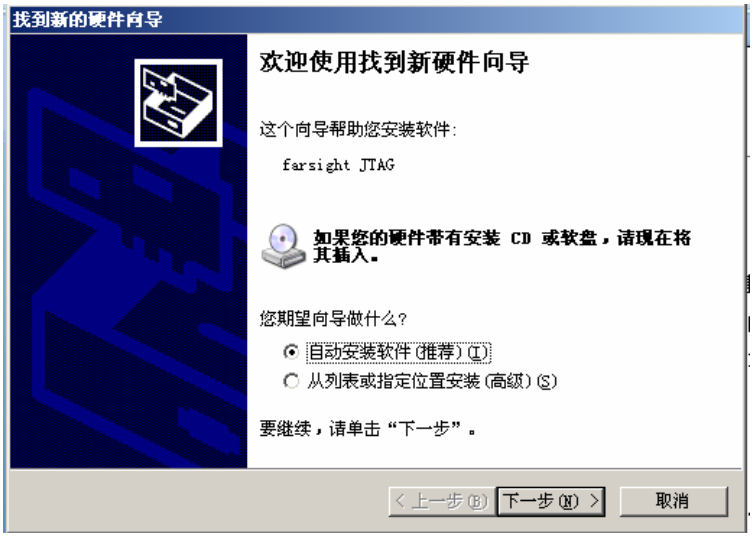

图 5-5　安装驱动界面

单击“下一步”按钮后会出现选择驱动安装目录（如图 5-6 所示），单击“浏览”按钮找到 DRIVER 所在的目录，如图 5-7 所示。

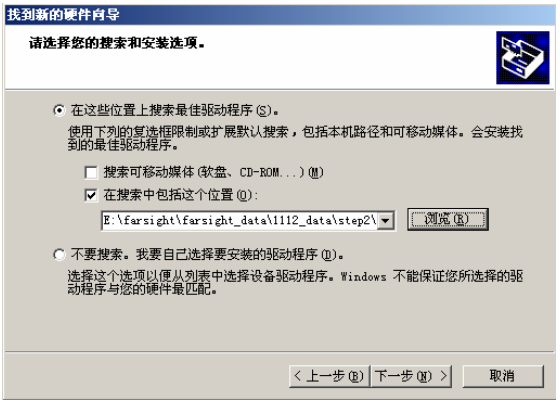

图 5-6　硬件向导

图 5-7　选择驱动文件目录

选择好后，单击“确定”按钮，会提示没有通过微软认证，单击“仍然继续”按钮，如图 5-8 所示。

在安装的过程中，会提示需要 ftdibus.sys 文件，单击“浏览”按钮，在 DRIVER 所在目录找到所需要的文件（如图 5-9 和图 5-10 所示），然后安装即可。

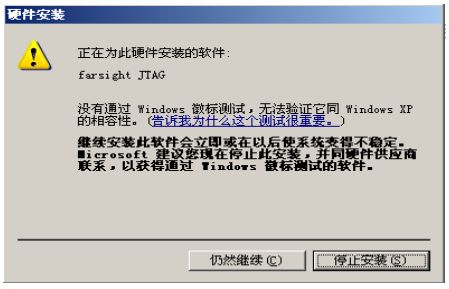

图 5-8　提示信息

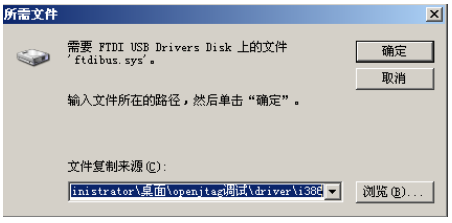

图 5-9　找到 ftdibus.sys 文件

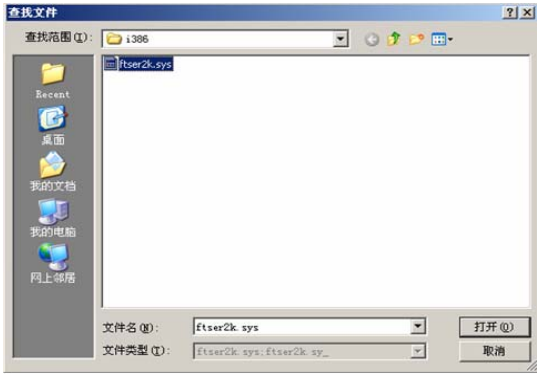

图 5-10　找到 USB 目录

5.3 Eclipse for ARM 使用

1．指定一个工程存放目录

Eclipse for ARM 是一个标准的窗口应用程序，可以单击程序按钮开始运行。打开后必须先指定一个工程存放路径，如图 5-11 所示。

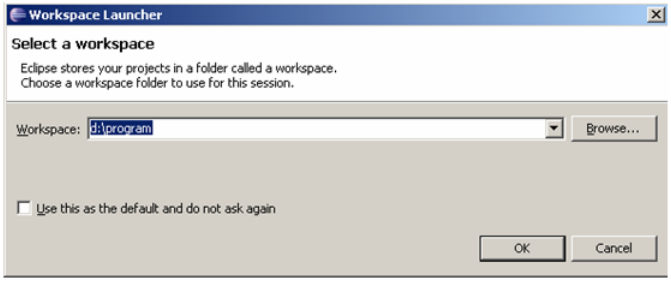

图 5-11 工程路径选择

2．创建一个工程

进入主界面后，选择“File→New→C Project”命令，Eclipse 将打开一个标准对话框，输入希望新建工程的名字并单击“Finish”按钮即可创建一个新的工程，建议对每个新建工程使用独立的文件夹。

3．新建一个 MakeFile 文件

在创建一个新的工程后，选择“File→New→Other”命令，在弹出的对话框中的 General 下单击 file，然后单击 next；然后选择所要指定的工程后，在文件名文本框中输入文件名 MakeFile，单击“Finish”按钮。

4．新建一个脚本文件

选择“File →New→Other”命令，在弹出的对话框中的 General 下单击 file，然后单击 next；然后选择所要指定的工程后，在文件名文本框中输入文件名 s5pc100.init，单击“Finish”按钮。

5．新建一个连接脚本文件

选择“File→New→Other”命令，在弹出的对话框中的 General 下单击 file ，然后单击 next；然后选择所要指定的工程后，在文件名文本框中输入文件名 map.lds，单击“Finish”按钮。

6．新建一个汇编源文件

选择“File →New→Other”命令，在弹出的对话框中的 General 下单击 file ，然后单击 next；然后选择所要指定的工程后，在文件名文本框中输入文件名 start.s，单击“Finish”按钮。

5.4　编译工程

（1）在汇编源文件（start.s）中输入汇编代码：

```
.equ   GPG3CON,   0xE03001C0
.equ   GPG3DAT,   0xE03001C4

.globl _start
_start:
      LDR       R0,=GPG3CON
LDR             R1,=0X10
      STR       R1,[R0]       @//写控制寄存器，IO 引脚使能为输出
LOOP:
      LDR       R0,=GPG3DAT
      MOV       R1,#0X02      @//点亮 led1
      STR       R1,[R0]
      LDR       R2,=0XFFFFF   @//延时
LOOP1:
      SUB       R2,R2,#1
      CMP       R2,#0
      BNE       LOOP1
      MOV       R1,#0X0       @//熄灭 led1
      STR       R1,[R0]
      LDR       R2,=0XFFFFF   @//延时
LOOP2:
      SUB       R2,R2,#1
      CMP       R2,#0
      BNE       LOOP2
      B         LOOP
      .end
```

（2）在 map.lds 中输入如下信息：

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
/*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
 . = 0x34000;
 . = ALIGN(4);
 .text     :
 {
      start.o(.text)
      *(.text)
 }
 . = ALIGN(4);
   .rodata :
 { *(.rodata) }
   . = ALIGN(4);
   .data :
 { *(.data) }
   . = ALIGN(4);
   .bss :
```

```
    { *(.bss) }
}
```

（3）编写 MakeFile 文件编译规则，在 MakeFile 中输入如下信息：

```
all:start.s
 arm-none-eabi-gcc-4.6.2 -O0 -g -c -o start.o start.s
 arm-none-eabi-ld       start.o   -Tmap.lds -o start.elf
 arm-none-eabi-objcopy   -O binary -S start.elf start.bin
arm-none-eabi-objdump   -D start.elf >start.dis
```

（4）在 s5pc100.init 文件中输入如下信息：

```
target remote 127.0.0.1:3333
monitor halt
monitor arm mcr 15 0 1 0 0 0
monitor step 0
```

（5）保存，编译 Project→Bulit All。

5.5 调试工程

5.5.1 配置 FS-JTAG 调试工具

如图 5-12 所示，在 Target 选项中选择 s5pc100，然后在 WorkDir 选项中选择自己的工程目录（D:\program\led），这里是笔者的当前环境，请读者按照自己的实际环境进行填写。

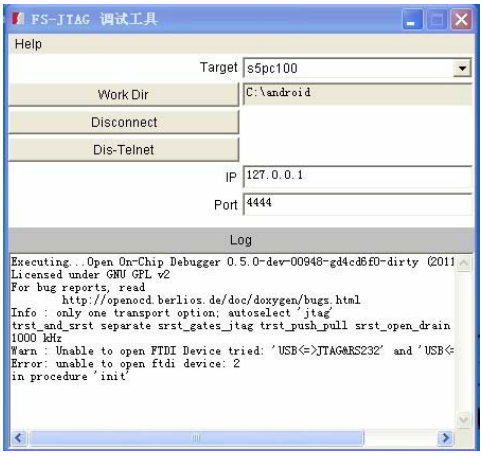

图 5-12　FS-JTAG 工具

上述工作做完之后，单击 Connect 按钮后，该按钮会变为 Disconnect，如图 5-12 所示，即表示已经连接目标板。最后单击 Telnet 按钮（这一步可以跳过），将会弹出如图 5-13 所示界面即表示已经连上目标板。

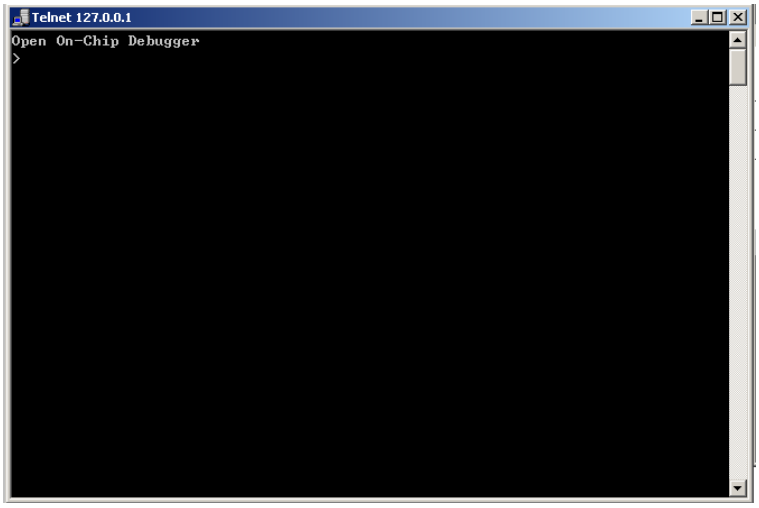

图 5-13　配置 FS-JTAG 调试工具

5.5.2　配置调试工具

在 Eclips 的菜单中选择“Run→Debug Configurations”弹出如图 5-14 对话框。

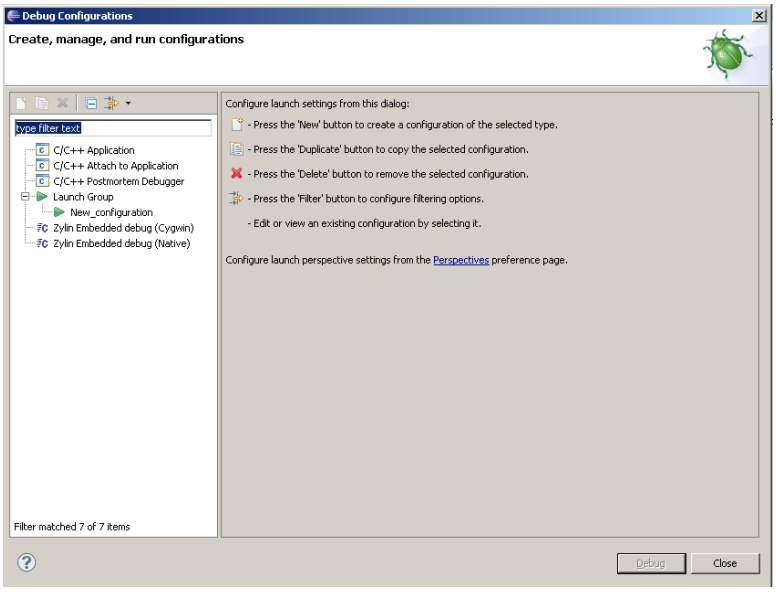

图 5-14　“Debug Configuration”对话框

选择 Zyin Embedded debug(Native)选项，然后单击鼠标右键，在弹出的快捷菜单中选择“New”命令；在 Main 选项卡中的 Project 框中，单击“Browse”按钮选择 led 工程；在 C/C++ Application 中单击“Browse”按钮找到工程目录下的 led.elf 文件，如图 5-15 所示。

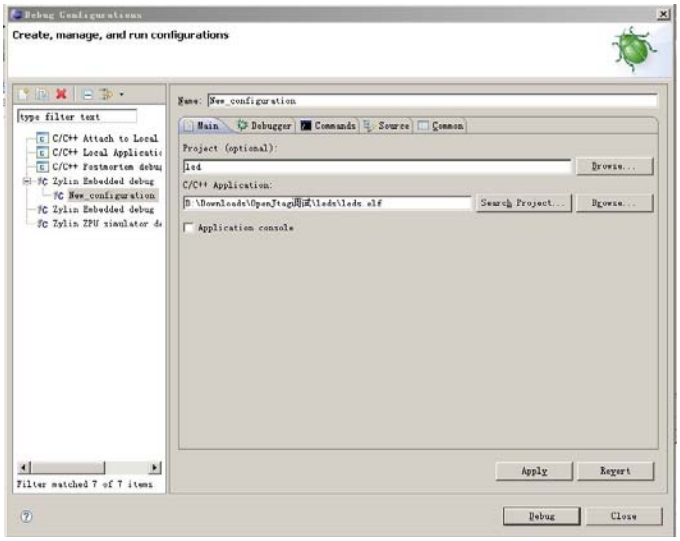

图 5-15　Main 选项卡

在 Debugger 选项卡中的 Main 子选项卡中的 GDB debugger 的框中单击“Browse”按钮选择前面安装的 arm-none-eabi-gdb.exe（这里选择自己的安装目录），在 GDB Command file 中选择自己工程目录下的 s5pc100.init 文件，如图 5-16 所示。

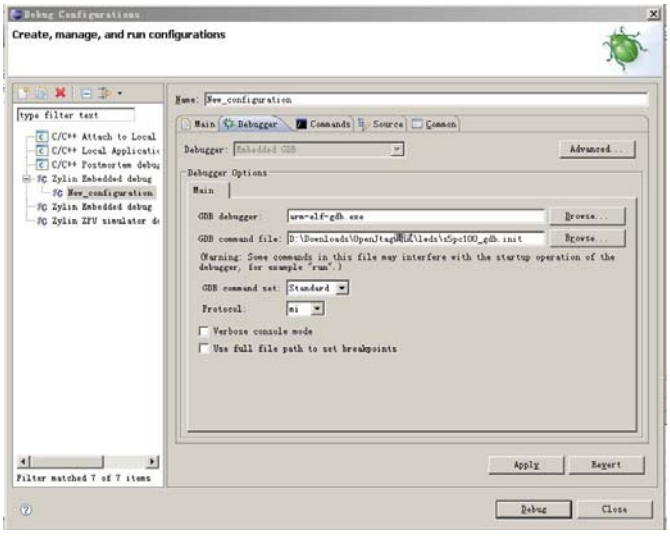

图 5-16　Debugger 选项

在 Command 选项卡中输入如下内容，如图 5-17 所示。

```
load
break _start
c
```

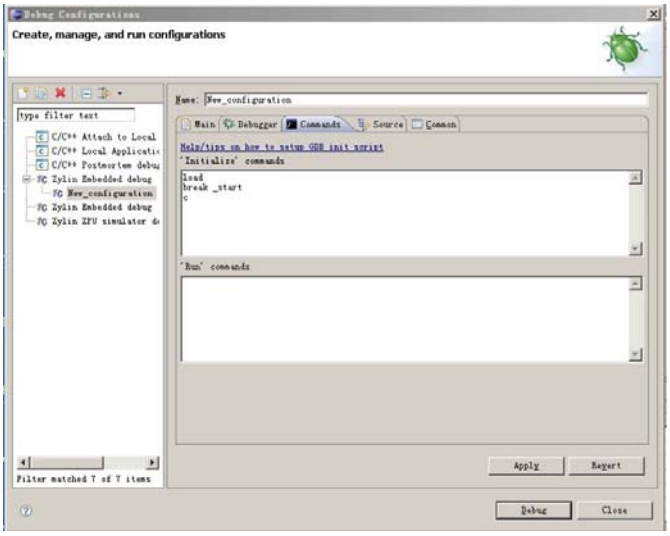

图 5-17　Commands 选项卡

单击“Apply”按钮后，再单击“Debug”按钮开始调试运行，会出现调试主界面，如图 5-18 所示。

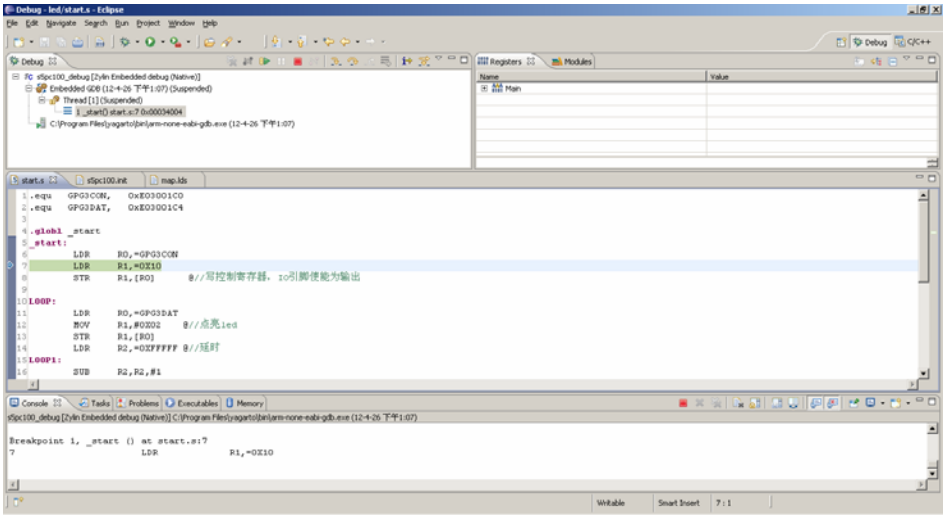

图 5-18　调试主界面

程序会在断点处停下，然后使用单步和全速等工具进行调试运行程序，单击全速运行，会出现 LED1 闪亮。

从图中可以看出一个大概的调试界面，如图 5-19 所示的按钮是和调试有关的，有单步，step over 和 step in 方式。还有 Eclipse 自带的挂起、中断连接功能。下面简单介绍一下各个窗口的用途。

图 5-19　调试按钮

如图 5-20 所示窗口是用来查看函数变量的，可以看到当前 i,j 的值。

如图 5-21 所示窗口是用来查看 ARM 寄存器的，从 r0～r12 通用寄存器的值可以被很清楚的观察到，并且还可观察到 CPSR 当前状态寄存器的值。

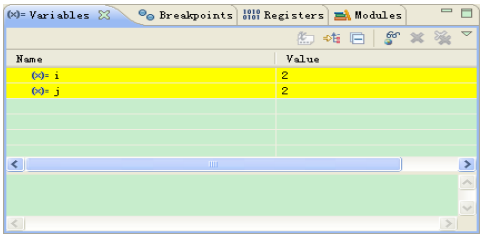

图 5-20　查看变量

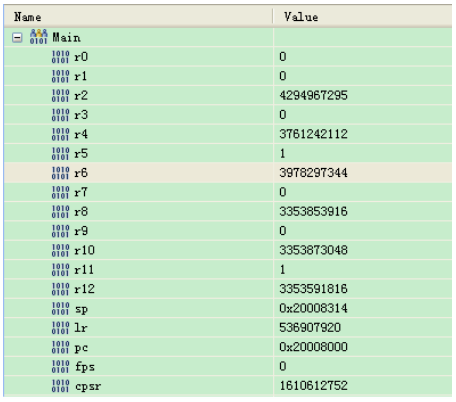

图 5-21　查看寄存器

5.6　本章小结

本章主要介绍了如何编写 GNU-ARM 汇编风格的程序，以及如何基于 S5PC100 在 Eclipse 下进行调试，并且介绍了 FS-JTAG 的详细用法。本书后面章节的大部分实验都是基于这个环境的。工欲善其事，必先利其器，所以必须熟练掌握环境的使用。

5.7　练习题

1．熟悉 Eclipse 开发环境。
2．新建一个工程，编写一个汇编程序实现 3+13=16 的操作。
3．在 5-2 节的基础上，使用 FS-JTAG 进行调试。

第 6 章　GPIO 编程

GPIO 控制技术是接口技术中最简单的一种。本章通过介绍 S5PC100 芯片的 GPIO 控制方法，让读者初步掌握控制硬件接口的方法。本章的主要内容：

- GPIO 功能介绍。
- S5PC100 芯片的 GPIO 控制器详解。
- S5PC100 的 GPIO 应用。

6.1　GPIO 功能介绍

首先应该理解什么是 GPIO。GPIO 的英文全称为 General-Purpose IO ports，也就是通用 IO 接口。在嵌入式系统中常常有数量众多，但是结构却比较简单的外部设备/电路，对这些设备/电路，有的需要 CPU 为之提供控制手段，有的则需要被 CPU 用做输入信号。而且，许多这样的设备/电路只要求一位，即只要有开/关两种状态就够了。比如，控制某个 LED 灯亮与灭，或者通过获取某个引脚的电平属性来达到判断外围设备的状态。对这些设备/电路的控制，使用传统的串行口或并行口都不合适。所以在微控制器芯片上一般都会提供一个“通用可编程 IO 接口”，即 GPIO。接口至少有两个寄存器，即“通用 IO 控制寄存器”与“通用 IO 数据寄存器”。数据寄存器的各位都直接引到芯片外部，而对这种寄存器中每一位的作用，即每一位的信号流通方向，则可以通过控制寄存器中对应位独立地加以设置。比如，可以设置某个引脚的属性为输入、输出或其他特殊功能。

在实际的 MCU 中，GPIO 是有多种形式的。比如，有的数据寄存器可以按照位寻址，有些却不能按照位寻址，这在编程时就要区分了。比如传统的 8051 系列，就区分成可位寻址和不可位寻址两种寄存器。另外，为了使用的方便，很多 MCU 的 GPIO 接口除必须具备两个标准寄存器外，还提供上拉寄存器，可以设置 IO 的输出模式是高阻，还是带上拉的电平输出，或者不带上拉的电平输出。这在电路设计中，外围电路就可以简化不少。

6.2　S5PC100 芯片的 GPIO 控制器详解

6.2.1　特性

S5PC100 的 GPIO 特性包括如下几点：

- 173 个通用控制 IO，141 个睡眠不可中断 IO，32 个睡眠可中断 IO。
- 130 个多功能输入/输出接口。
- 控制引脚状态模式（除了 GPH0、GPH1、GPH2 和 GPH3）。

6.2.2 GPIO 分组预览

- GPA0:8 in/out pin – 2xUART 带控制流
- GPA1:5 in/out pin – 2xUART 不带控制流 or 1xUART 带控制流，1x IrDA。
- GPB:8 in/out pin – 2x SPI 总线接口。
- GPC:5 in/out pin – I2S 总线接口，PCM 接口，AC97 接口。
- GPD:7 in/out pin – 2xI2C 总线接口，PWM 接口，External DMA 接口，SPDIF 接口。
- GPE0,1:14 in/out pin –摄像头接口，SD/MMC 接口。
- GPF0,1,2,3:28 in/out pin – LCD 接口。
- GPG0,1,2,3:25 in/out pin – 3xMMC channel，SPI，I2S，PCM，SPDIF 各种接口。
- GPH0,1,2,3:32 in/out pin –摄像头通道接口，键盘，最大支持 32 位的睡眠可中断接口。
- GPI:8 in/out pin – 启动选项，PWI。
- GPJ0,1,2,3,4:33 in/out pin – Modem IF，HIS，ATA 接口
- GPK0,1,2,3:30 in/out pin –EBI 控制信号。

6.2.3 S5PC100 的 GPIO 常用寄存器分类

1．端口控制寄存器（GPACON-GPHCON）

在 S5PC100 中，大多数的引脚都可复用，所以必须对每个引脚进行配置。端口控制寄存器（GPnCON）定义了每个引脚的功能。

2．端口数据寄存器（GPADAT-GPHDAT）

如果端口被配置成了输出端口，可以向 GPnDAT 的相应位写数据。如果端口被配置成了输入端口，可以从 GPnDAT 的相应位读出数据。

3．端口上拉寄存器（GPBUP-GPHUP）

端口上拉寄存器控制了每个端口组的上拉电阻的允许/禁止。如果某一位为 0，相应的上拉电阻被允许；如果是 1，相应的上拉电阻被禁止。如果端口的上拉电阻被允许，无论在哪种状态（输入、输出、DATAn、EINTn 等）下，上拉电阻都起作用。

6.2.4 GPIO 功能描述

GPIO 功能概括图如图 6-1 所示。

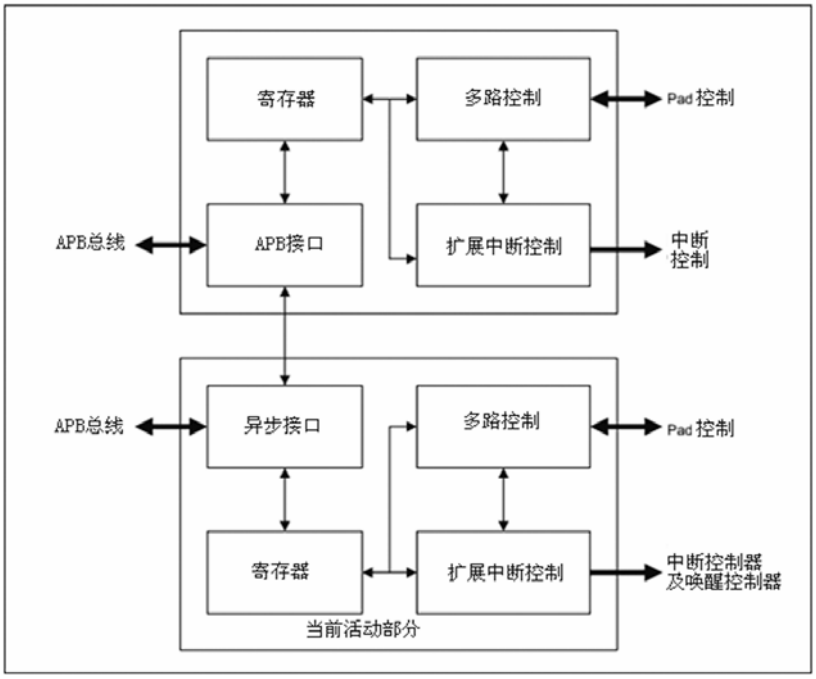

图 6-1　GPIO 功能概括图

在 S5PC100 中，输出端口被分为如表 6-1 所示的种类。

表 6-1

A	GPA0, GPA1, GPB, GPC, GPD, GPE0, GPE1, GPF0, GPF1, GPF2, GPF3, GPG0, GPG2, GPG3, GPH0, GPH1, GPH2, GPH3, GPI, GPJ0, GPJ1, GPJ2, GPJ3, GPJ4, GPK0, GPK1, GPK2, GPK3
B	MP0
OSC_A	ETC4[5](XrtcXTI/XrtcXTO)
OSC_B	ETC4[0](X27mXTI/X27mXTO), ETC4[2](XXTI/XXTO), ETC4[4](XusbXTI/XusbXTO)

6.2.5　S5PC100 I/O 接口常用寄存器详解

提示：

GPxCON、GPxDAT、GPxPULL、GPxDRV 工作在普通模式，GPxPDNCON、GPxPDNPULL 工作在低功耗模式。

对于 GPIO 控制寄存器，现在来看一下每一组 IO 的详细功能描述，考虑到 GPIO 的寄存器很多，这里只列出与后面 GPIO 示例有关的寄存器，如表 6-2 所示。

表 6-2　GPG3 控制寄存器(address = 0xe030001c0)

GPG3CON	位	描述	初始状态
GPG3CON[0]	[3:0]	0000 = 输入, 0001 = 输出, 0010 = SD_2_CLK,0011 = SPI_2_CLK, 0100 = I2S2_SCLK, 0101 = PCM_0_SCLK, 1111 = NWU_INTG14[0]	0000
GPG3CON[1]	[7:4]	0000 = 输入, 0001 = 输出, 0010 = SD_2_CMD,0011 = SPI_2_nSS, 0100 = I2S2_CDCLK, 0101 = PCM_0_EXTCLK, 1111 = NWU_INTG14[1]	0000
GPG3CON[2]	[11:8]	0000 = 输入, 0001 = 输出, 0010 = SD_2_D[0],0011 = SPI_2_MISO, 0100 = I2S2_LRCK, 0101 = PCM_0_FSYNC, 1111 = NWU_INTG14[2]	0000
GPG3CON[3]	[15:12]	0000 = 输入, 0001 = 输出, 0010 = SD_2_D[1], 0011 = SPI_2_MOSI, 0100 = I2S2_SDI, 0101 = PCM_0_SIN, 1111 = NWU_INTG14[3]	0000
GPG3CON[4]	[19:16]	0000 = 输入, 0001 = 输出, 0010 = SD_2_D[2], 0011 = Reserved, 0100 = I2S2_SDO, 0101 = PCM_0_SOUT, 1111 = NWU_INTG14[4]	0000
GPG3CON[5]	[23:20]	0000 = 输入, 0001 = 输出, 0010 = SD_2_D[3], 0011 = Reserved, 0100 = Reserved, 0101 = SPDIF_0_OUT, 1111 = NWU_INTG14[5]	0000
GPG3CON[6]	[27:24]	0000 = 输入, 0001 = 输出, 0010 = SD_2_CDn, 0011 = Reserved, 0100 = Reserved, 0101 = SPDIF_EXTCLK, 1111 = NWU_INTG14[6]	0000

6.2.6　GPIO 数据寄存器

GPIO 数据寄存器如表 6-3 所示。

表 6-3　GPIO 数据寄存器

GPG3DAT[n](n=0~7)	[n]	该寄存器决定了输入或者输出的电平状态	-

6.3　S5PC100 GPIO 的应用

通过 6.1、6.2 节的介绍，读者了解了 GPIO 的功能，以及 S5PC100 芯片 GPIO 控制器的配置方法。本节通过一个简单示例说明 S5PC100 的 GPIO 接口的应用。

示例将利用 S5PC100 的 GPF4、GPF5、GPF6、GPF7 这 4 个 I/O 引脚控制 4 个 LED 发光二极管，使其有规律地闪烁。

6.3.1　电路连接

如图 6-2 所示，LED1～LED4 分别与 GPG3_0～GPG3_3 相连，通过 GPG3_0～GPG3_3 引脚的高低电平来控制三极管的导通性，从而控制 LED 的亮灭。

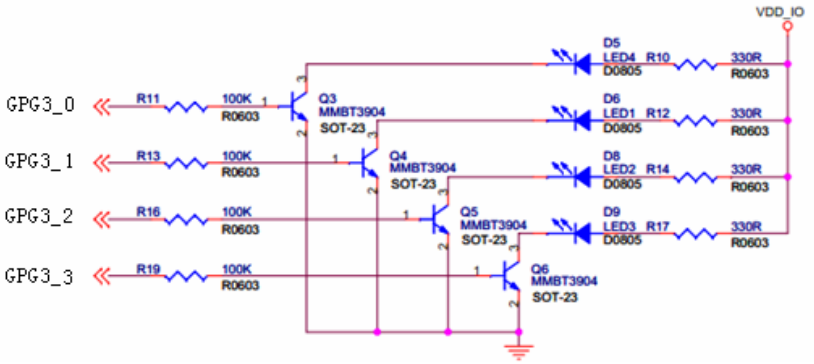

图 6-2　LED 接线原理图

因此，当这几个引脚输出高电平时发光二极管点亮；反之，发光二极管熄灭。

6.3.2　寄存器设置

为了实现控制 LED 的目的，需要通过配置 GPG3CON 寄存器将 GPG3_0、GPG3_1、GPG3_2、GPG3_3 设置为输出属性。通过设置 GFG3DAT 寄存器实现点亮与熄灭 LED。

对于本例来说，GPG3 上拉寄存器可以不用设置。

6.3.3　程序编写

相关代码如下：

```
/*相关 GPIO 口的宏定义，用于下面的寄存器配置*/
volatile unsigned long *gpg3con = (volatile unsigned long *)0xe03001c0;
/*GPG3 控制寄存器*/
volatile unsigned long *gpg3dat = (volatile unsigned long *)0xe03001c4;
/*GPG3 数据寄存器*/

/*延时函数的实现*/
void delay()
{
 volatile int i = 0x100000;
 while (i--);
}

int main()
{
 int i = 0;
 int j;

 /* gpg3con0,1,2,3 设为输出引脚 */
```

```
    *gpg3con &= ~0xffff;
    *gpg3con |= 0x1111;

    while (1)
    {
         /* 说明如下:
          * i's bit0 => gpg3dat[1] => GPG3_1 => LED1
          * i's bit1 => gpg3dat[2] => GPG3_2 => LED2
          * i's bit2 => gpg3dat[3] => GPG3_3 => LED3
          * i's bit3 => gpg3dat[0] => GPG3_0 => LED4
          */
         *gpg3dat &= ~0xf;
         /* 流水灯 : LED1~LED4 与 GPG3DAT 的位置不是顺序对应的, 要调整一下 */
         j = ((i<<1) & ~(1<<4)) | ((i >> 3) & (1<<0));
         gpg3dat |= j;

         i++;
         if (i == 16)
              i = 0;
         delay();
    }
    return 0;
}
```

实验过程与结果：

（1）将程序编译后获得.bin 文件，将该文件通过 uboot 的 dnw 功能传输到内存的 0x20008000 这个地址，然后使用 go 命令去执行。

（2）观察实验结果，可以看到流水灯现象。

注意：

如果使用 FS-JTAG 仿真环境，可以按第 5 章的说明调试程序。如果没有，可以通过 uboot 下载编译后的 bin 文件到内存中运行，通过串口打印语句调试。本书后面的实验方法相同。

6.4 本章小结

通过本章学习，需要理解 GPIO 的概念，掌握 S5PC100 上的 GPIO 编程方法。

6.5 练习题

1. 什么是 GPIO？
2. S5PC100 有几组 GPIO 端口？
3. 如何实现利用 S5PC100 的 GPIO 控制 LED？请画出原理图，并编程实现。

第 7 章　ARM 异常及中断处理

几乎每种处理器都支持特定异常处理。中断是异常中的一种。了解处理器的异常处理相关知识，是学习一种处理器的重要环节。

本章主要内容：

- ARM 异常中断处理概述。
- ARM 体系异常种类。
- ARM 异常的优先级。
- ARM 处理器模式和异常。
- ARM 异常响应和处理程序返回。
- ARM 应用系统中异常中断处理程序的安装。
- ARM 的 SWI 异常中断处理程序设计。
- FIQ 和 IRQ 异常中断程序设计。

7.1　ARM 异常中断处理概述

1．中断的概念

什么是中断，我们从一个生活中的例子引入。你正在家中看书，突然电话铃响了，你放下书本，去接电话，和来电话的人交谈，然后放下电话，回来继续看你的书。这就是生活中的“中断”的现象，就是正常的工作过程被外部的事件打断了。

在处理器中，所谓中断，是一个过程，即 CPU 在正常执行程序的过程中，遇到外部 / 内部的紧急事件需要处理，暂时中断（中止）当前程序的执行，而转去为事件服务，待服务完毕，再返回到暂停处（断点）继续执行原来的程序。为事件服务的程序称为中断服务程序或中断处理程序。严格地说，上面的描述是针对硬件事件引起的中断而言的。用软件方法也可以引起中断，即事先在程序中安排特殊的指令，CPU 执行到该类指令时，转去执行相应的一段预先安排好的程序，然后再返回来执行原来的程序，这可称为软中断。把软中断考虑进去，可给中断再下一个定义：中断是一个过程，是 CPU 在执行当前程序的过程中因硬件或软件的原因插入了另一段程序运行的过程。因硬件原因引起的中断过程的出现是不可预测的，即随机的，而软中断是事先安排的。

2．中断源的概念

仔细研究一下生活中的中断，对于理解中断的概念也很有好处。什么可以引起中断，

生活中很多事件可以引起中断：

有人按门铃了，电话铃响了，你的闹钟响了，你烧的水开了……诸如此类的事件。我们把可以引起中断的信号源称为中断源。

3．中断优先级的概念

设想一下，我们正在看书，电话铃响了，同时又有人按了门铃，你该先做什么呢？如果你正在等一个很重要的电话，一般不会去理会门铃；反之，如果你正在等一位重要的客人，则可能就不会去理会电话了。如果不是这两者（既不等电话，也不等人上门），你可能会按你通常的习惯去处理。总之，这里存在一个优先级的问题，在处理器中也是如此，也有优先级的问题。即同时有多个中断源递交中断申请时的中断控制器对中断源的响应优先级别。需要注意的是，优先级的问题不仅仅发生在两个中断同时产生的情况，也发生在一个中断已产生，又有一个中断产生的情况。比如，你正接电话，有人按门铃的情况，或你正开门与人交谈，又有电话响了的情况。这时也需要根据中断源的优先级来决定下一动作。

ARM 处理器中有 7 种类型的异常，按优先级从高到低的排列如下：复位异常（Reset）、数据异常（Data Abort）、快速中断异常（FIQ）、外部中断异常（IRQ）、预取异常（Prefetch Abort）、软中断异常（SWI）和未定义指令异常（Undefined interrupt）。

注意：

在 ARM 处理器中，异常（Exception）和中断（Interrupt）有些差别，异常主要是从处理器被动接受异常的角度出发，而中断带有向处理器主动申请的色彩。在本书中，对“异常”和“中断”不做严格区分，两者都是指请求处理器打断正常的程序执行流程，进入特定程序循环的一种机制。

7.2 ARM 体系异常种类

在 ARM 体系结构中，存在 7 种异常处理。当异常发生时，处理器会把 PC 设置为一个特定的存储器地址。这一地址放在被称为向量表（vector table）的特定地址范围内。向量表的入口是一些跳转指令，跳转到专门处理某个异常或中断的子程序。

存储器映射地址 0x00000000 是为向量表（一组 32 位字）保留的。在有些处理器中，向量表可以选择定位在存储空间的高地址（从偏移量 0xffff0000 开始）。一些嵌入式操作系统，如 Linux 和 Windows CE 就利用了这一特性。

注意：

Cortex-A8 系统中支持通过设置 CP15 的 C12 寄存器将异常向量表的首地址设置在任意地址。下文标记为 C12（CP15）

如表 7-1 所示列出了 ARM 的 7 种异常类型。

表 7-1　ARM 的 7 种异常类型

异 常 类 型	处理器模式	执行低地址	执行高地址
复位异常（Reset）	特权模式	0x00000000	0xFFFF0000
未定义指令异常（Undefined Interrupt）	未定义指令中止模式	0x00000004	0xFFFF0004
软中断异常（Software Abort）	特权模式	0x00000008	0xFFFF0008
预取异常（Prefetch Abort）	数据访问中止模式	0x0000000C	0xFFFF000C
数据异常（Data Abort）	数据访问中止模式	0x00000010	0xFFFF0010
外部中断异常（IRQ）	外部中断请求模式	0x00000018	0xFFFF0018
快速中断异常（FIQ）	快速中断请求模式	0x0000001C	0xFFFF001C

异常处理向量表如图 7-1 所示。

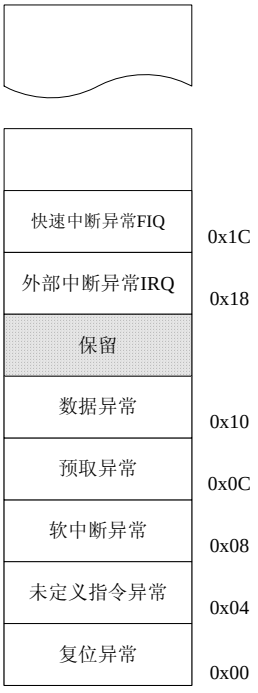

图 7-1　异常处理向量表

当异常发生时，分组寄存器 r14 和 SPSR 用于保存处理器状态，操作伪指令如下：

```
R14_<exception_mode> = return link
SPSR_<exception_mode> = CPSR
CPSR[4：0] = exception mode number
CPSR[5] = 0   /*进入 ARM 状态*/
If <exception_mode> = = reset or FIQ then
   CPSR[6] = 1 /*屏蔽快速中断 FIQ*/
   CPSR[7] = 1 /*屏蔽外部中断 IRQ*/
   PC = exception vector address
```

异常返回时，SPSR 内容恢复到 CPSR，连接寄存器 r14 的内容恢复到程序计数器 PC。

1．复位异常

当处理器的复位引脚有效时，系统产生复位异常中断，程序跳转到复位异常中断处理程序处执行。复位异常中断通常用于系统上电和系统复位两种情况。

当复位异常时，系统（处理器自动执行的，以下几个异常相同）执行下列伪操作。

```
R14_svc = UNPREDICTABLE value
SPSR_svc = UNPREDICTABLE value
CPSR[4：0] = 0b10011 /*进入特权模式*/
CPSR[5] = 0          /*处理器进入 ARM 状态*/
CPSR[6] = 1          /*禁止快速中断*/
CPSR[7] = 1          /*禁止外设中断*/
If high vectors configured then
   PC = 0xffff0000
Else
   PC = 0x00000000
```

复位异常中断处理程序将进行一些初始化工作，内容与具体系统相关。下面是复位异常中断处理程序的主要功能。

- 设置异常中断向量表。
- 初始化数据栈和寄存器。
- 初始化存储系统，如系统中的 MMU 等。
- 初始化关键的 I/O 设备。
- 使能中断。
- 处理器切换到合适的模式。
- 初始化 C 变量，跳转到应用程序执行。

2．未定义指令异常

当 ARM 处理器执行协处理器指令时，它必须等待一个外部协处理器应答后，才能真正执行这条指令。若协处理器没有响应，则发生未定义指令异常。未定义指令异常可用于在没有物理协处理器的系统上，对协处理器进行软件仿真，或通过软件仿真实现指令集扩展。例如，在一个不包含浮点运算的系统中，CPU 遇到浮点运算指令时，将发生未定义指令异常中断，在该未定义指令异常中断的处理程序中可以通过其他指令序列仿真浮点运算指令。

仿真功能可以通过下面步骤实现。

（1）将仿真程序入口地址链接到向量表中未定义指令异常中断入口处（0x00000004 或 0xffff0004），并保存原来的中断处理程序。

（2）读取该未定义指令的 bits[27:24]，判断其是否是一条协处理器指令。如果 bits[27:24]值为 0b1110 或 0b110x，该指令是一条协处理器指令；否则，由软件仿真实现协处理器功能，可以通过 bits[11:8]来判断要仿真的协处理器功能（类似于 SWI 异常实现机制）。

（3）如果不仿真该未定义指令，程序跳转到原来的未定义指令异常中断的中断处理程序行。

当未定义指令异常发生时，系统执行下列伪操作。

```
r14_und = address of next instruction after the undefined instruction
SPSR_und = CPSR
CPSR[4：0] = 0b11011 /*进入未定义指令模式*/
CPSR[5] = 0          /*处理器进入 ARM 状态*/
/*CPSR[6]保持不变*/
CPSR[7] = 1          /*禁止外设中断*/
If  high vectors configured then
     PC = 0xffff0004
Else
     PC = 0x00000004
```

3．软中断异常

软中断异常发生时，处理器进入特权模式，执行一些特权模式下的操作系统功能。软中断异常发生时，处理器执行下列伪操作。

```
r14_svc = address of next instruction after the SWI instruction
SPSR_und = CPSR
CPSR[4：0] = 0b10011 /*进入特权模式*/
CPSR[5] = 0          /*处理器进入 ARM 状态*/
/*CPSR[6]保持不变*/
CPSR[7] = 1          /*禁止外设中断*/
If  high vectors configured then
     PC = 0xffff0008
Else
     PC = 0x00000008
```

4．预取异常

预取异常是由系统存储器报告的。当处理器试图去取一条被标记为预取无效的指令时，发生预取异常。

如果系统中不包含 MMU，指令预取异常中断处理程序只是简单地报告错误并退出；若包含 MMU，引起异常的指令的物理地址被存储到内存中。

预取异常发生时，处理器执行下列伪操作。

```
r14_svc = address of the aborted instruction + 4
SPSR_und = CPSR
CPSR[4：0] = 0b10111 /*进入特权模式*/
CPSR[5] = 0          /*处理器进入 ARM 状态*/
/*CPSR[6]保持不变*/
CPSR[7] = 1          /*禁止外设中断*/
If  high vectors configured then
    PC = 0xffff000C
Else
    PC = 0x0000000C
```

5．数据异常

数据异常时由存储器发出数据中止信号，它由存储器访问指令 Load/Store 产生。当数据访问指令的目标地址不存在或者该地址不允许当前指令访问时，处理器产生数据访问中止异常。当数据异常发生时，处理器执行下列伪操作。

```
r14_abt = address of the aborted instruction + 8
```

```
SPSR_abt = CPSR
CPSR[4：0] = 0b10111
CPSR[5] = 0
/*CPSR[6]保持不变*/
CPSR[7] = 1            /*禁止外设中断*/
If  high vectors configured then
    PC = 0xffff000C10
Else
    PC = 0x00000010
```

当数据访问中止异常发生时，寄存器的值将根据以下规则进行修改。

（1）返回地址寄存器 r14 的值只与发生数据异常的指令地址有关，与 PC 值无关。

（2）如果指令中没有指定基址寄存器回写，则基址寄存器的值不变。

（3）如果指令中指定了基址寄存器回写，则寄存器的值和具体芯片的 Abort Models 有关，由芯片的生产商指定。

（4）如果指令只加载一个通用寄存器的值，则通用寄存器的值不变。

（5）如果是批量加载指令，则寄存器中的值不可预知。

（6）如果指令加载协处理器寄存器的值，则被加载寄存器的值不可预知。

6．外部中断异常

当处理器的外部中断请求引脚有效，而且 CPSR 寄存器的 I 控制位被清除时，处理器产生外部中断异常。系统中各外部设备通常通过该异常中断请求处理器服务。

当外部中断异常发生时，处理器执行下列伪操作。

```
r14_irq = address of next instruction to be executed + 4
SPSR_irq = CPSR
CPSR[4：0] = 0b10010 /*进入特权模式*/
CPSR[5] = 0          /*处理器进入 ARM 状态*/
/*CPSR[6]保持不变*/
CPSR[7] = 1          /*禁止外设中断*/
If  high vectors configured then
    PC = 0xffff0018
Else
    PC = 0x00000018
```

7．快速中断异常

当处理器的快速中断请求引脚有效且 CPSR 寄存器的 F 控制位被清除时，处理器产生快速中断异常。当快速中断异常发生时，处理器执行下列伪操作。

```
r14_fiq = address of next instruction to be executed + 4
SPSR_fiq = CPSR
CPSR[4：0] = 0b10001         /*进入 FIQ 模式*/
CPSR[5] = 0
CPSR[6] = 1
CPSR[7] = 1
If  high vectors configured then
    PC= 0xffff001c
Else
    PC = 0x0000001c
```

7.3　ARM 异常的优先级

每一种异常按如表 7-2 所示中设置的优先级得到处理。

表 7-2　异常优先级

优　先　级	异　　常
最高　　1	复位异常
2	数据异常
3	快速中断异常
4	外部中断异常
5	预取异常
6	软中断异常
最低　　7	未定义指令异常

异常可以同时发生，此时处理器按表 7-2 中设置的优先级顺序处理异常。例如，处理器上电时发生复位异常，复位异常的优先级最高，所以当产生复位时，它将优先于其他异常得到处理。同样，当一个数据异常发生时，它将优先于除复位异常外的其他所有异常而得到处理。

优先级最低的两种异常是软件中断异常和未定义指令异常。因为正在执行的指令不可能既是一条软中断指令，又是一条未定义指令，所以软中断异常和未定义指令异常享有相同的优先级。

7.4　ARM 处理器模式和异常

每一种异常都会导致内核进入一种特定的模式。ARM 处理器异常及其对应的模式如表 7-3 所示。此外，也可以通过编程改变 CPSR，进入任何一种 ARM 处理器模式。

注意：

用户模式和系统模式是仅有的不可通过异常进入的两种模式，也就是说，要进入这两种模式，必须通过编程改变 CPSR。

表 7-3　ARM 处理器异常及其对应模式

异　　常	模　　式	用　　途
快速中断异常	FIQ	进行快速中断请求处理
外部中断请求	IRQ	进行外部中断请求处理
软中断异常	SVC	进行操作系统的高级处理

续表

异　常	模　式	用　途
复位异常	SVC	进行操作系统的高级处理
预取指令中止异常	Abort	虚存和存储器保护
数据中止异常	Abort	虚存和存储器保护
未定义指令异常	Undefined	软件模拟硬件协处理器

7.5　ARM 异常响应和处理程序返回

7.5.1　中断响应的概念

中断的响应过程：当有事件产生，进入中断之前我们必须先记住现在看到书的第几页了，或拿一个书签放在当前页的位置，然后去处理不同的事情（因为处理完了，我们还要回来继续看书），如电话铃响我们要到放电话的地方去，门铃响我们要到门那边去，也就是说不同的中断，我们要在不同的地点处理，而这个地点通常不是固定的。

通常，中断响应大致可以分为以下几个步骤：（1）保护断点，即保存下一个将要执行的指令的地址，就是把这个地址送入堆栈；（2）寻找中断入口，根据不同的中断源所产生的中断，查找不同的入口地址；（3）执行中断处理程序；（4）中断返回，执行完中断指令后，就从中断处返回到主程序，继续执行。

7.5.2　ARM 异常响应流程

1．判断处理器状态

当异常发生时，处理器自动切换到 ARM 状态，所以在异常处理函数中要判断在异常发生前处理器是 ARM 状态还是 Thumb 状态。这可以通过检测 SPSR 的 T 位来判断。

通常情况下，只有在 SWI 处理函数中才需要知道异常发生前处理器的状态。所以在 Thumb 状态下，调用 SWI 软中断异常必须注意以下两点。

（1）发生异常的指令地址为（LR－2）而不是（LR－4）。

（2）Thumb 状态下的指令是 16 位的，在判断中断向量号时使用半字加载指令 LDRH。

下面的例子显示了一个标准的 SWI 处理函数，在函数中通过 SPSR 的 T 位判断异常发生前的处理器状态。

2．向量表

如前面介绍向量表时提到的，每一个异常发生时总是从异常向量表开始跳转。最简单的一种情况是向量表里面的每一条指令直接跳向对应的异常处理函数。其中快速中断处理函数 FIQ_Handler()可以直接从地址 0x1C 处开始，省下一条跳转指令，如图 7-2 所示。

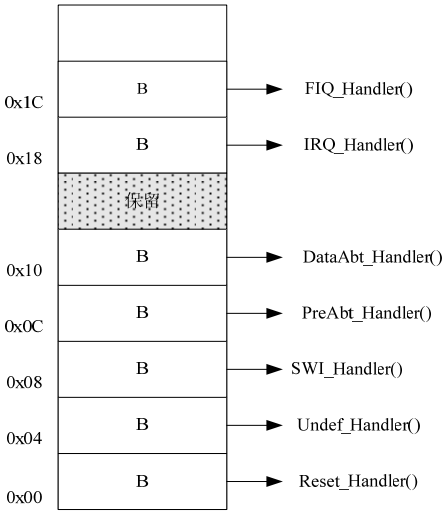

图 7-2　异常处理向量表

跳转指令 B 的跳转范围为±32MB，但很多情况下不能保证所有的异常处理函数都定位在向量的 32MB 范围内，而可能需要更大范围的跳转，而且由于向量表空间的限制，只能由一条指令完成。具体实现方法有下面两种。

（1）MOV　PC,＃imme_value。这种办法将目标地址直接赋值给 PC。但这种方法受格式限制不能处理任意立即数。这个立即数由一个 8 位数值循环右移偶数位得到。

（2）LDR　PC,[PC+offset]。把目标地址先存储在某一个合适的地址空间，然后把这个存储器单元的 32 位数据传送给 PC 来实现跳转。这种方法对目标地址值没有要求，但是存储目标地址的存储器单元必须在当前指令的±4KB 空间范围内。

注意：

在计算指令中引用 offset 数值时，要考虑处理器流水线中指令预取对 PC 值的影响。

7.5.3　从异常处理程序中返回

当一个 ARM 异常处理返回时，一共有 3 件事情需要处理：通用寄存器的恢复、状态寄存器的恢复及 PC 指针的恢复。通用寄存器的恢复采用一般的堆栈操作指令即可，下面重点介绍状态寄存器的恢复及 PC 指针的恢复。

1．恢复被中断程序的处理器状态

PC 和 CPSR 的恢复可以通过一条指令来实现，下面是 3 个例子。

```
MOVS  PC,LR
SUBS  PC,LR,＃4
LDMFD  SP!,{PC}^
```

这几条指令是普通的数据处理指令，特殊之处在于它们把程序计数器寄存器 PC 作为目标寄存器，并且带了特殊的后缀“S”或“^”。其中“S”或“^”的作用就是使指令在执行时，同时完成从 SPSR 到 CPSR 的复制，达到恢复状态寄存器的目的。

2．异常的返回地址

异常返回时，另一个非常重要的问题就是返回地址的确定。前面提到过，处理器进入异常时会有一个保存 LR 的动作，但是该保持值并不一定是正确中断的返回地址。以一个简单的指令执行流水状态图来对此加以说明，如图 7-3 所示。

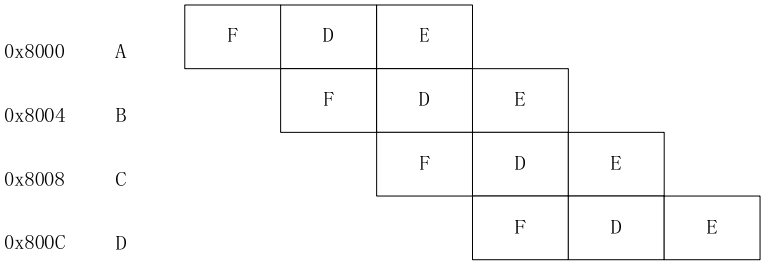

图 7-3　3 级流水线示例

在 ARM 架构中，PC 值指向当前执行指令地址加 8。也就是说，当执行指令 A（地址 0x8000）时，PC 等于 0x8000+8=0x8008，即等于指令 C 的地址。假设指令 A 是 BL 指令，则当执行时，会把 PC 值（0x8008）保存到 LR 寄存器。但是，接下来处理器会对 LR 进行一次自动调整，使 LR=LR－0x4。所以，最终保存在 LR 里的是如图 7-3 所示的 B 指令地址。所以当从 BL 返回时，LR 里面正好是正确的返回地址。

同样的跳转机制在所有的 LR 自动保存操作中都存在。当进入中断响应时，处理器对保存的 LR 也进行一次自动调整，并且跳转动作也是 LR=LR－0x04。由此，就可以对不同异常类型的返回地址依次比较。

假设在指令 B 处（地址 0x8004）发生了异常，进入异常响应后，LR 经过跳转保存的地址值应该是 C 的地址 0x8008。

（1）软中断异常。如果发生软中断异常，即指令 B 为 SWI 指令，从 SWI 中断返回后下一条执行指令就是 C，正好是 LR 寄存器保存的地址，所以直接把 LR 恢复给 PC 即可。

（2）IRQ 或 FIQ 异常。如果发生的是 IRQ 或 FIQ 异常，因为外部中断请求中断了正在执行的指令 B，当中断返回后，需要重新回到 B 指令执行，也就是说，返回地址应该是 B（0x8004），需要把 LR 减 4 送 PC。

（3）Data Abort 数据中止异常。在指令 B 处进入数据异常的响应，但导致数据异常的原因却应该是上一条指令 A。当中断处理程序恢复数据异常后，要回到 A 重新执行导致数据异常的指令，因此返回地址应该是 LR 加 8。为方便起见，表 7-4 中总结了各类异常和返回地址的关系。

表 7-4　异常和返回地址

异　　常	返回地址	用　　途
复位	—	复位没有定义 LR
数据中止	LR-8	指向导致数据中止异常的指令
FIQ	LR-4	指向发生异常时正在执行的指令
IRQ	LR-4	指向发生异常时正在执行的指令
预取指令中止	LR-4	指向导致预取指令异常的那条指令
SWI	LR	执行 SWI 指令的下一条指令
未定义指令	LR	指向未定义指令的下一条指令

7.6　ARM 的 SWI 异常中断处理程序设计

本节主要介绍编写 SWI 处理程序时需要注意的几个问题，包括判断 SWI 中断号，使用汇编语言编写 SWI 异常处理函数，使用 C 语言编写 SWI 异常处理函数，在特权模式下使用 SWI 异常中断处理，从应用程序中调用 SWI。

1．判断 SWI 中断号

当发生 SWI 异常，进入异常处理程序时，异常处理程序必须提取 SWI 中断号，从而得到用户请求的特定 SWI 功能。

在 SWI 指令的编码格式中，后 24 位称为指令的“comment field”。该域保存的 24 位数，即为 SWI 指令的中断号，如图 7-4 所示。

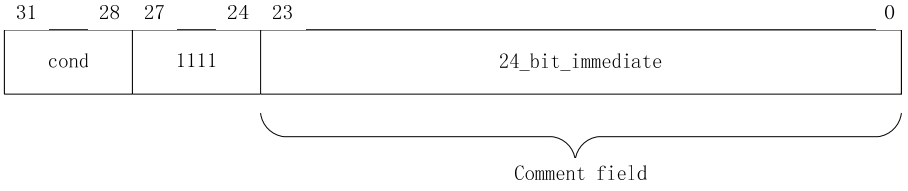

图 7-4　SWI 指令编码格式

第一级的 SWI 处理函数通过 LR 寄存器内容得到 SWI 指令地址，并从存储器中得到 SWI 指令编码。通常这些工作通过汇编语言、内嵌汇编来完成。下面的例子显示了提取中断向量号的标准过程。

```
.SWI_Handler:
STMFD sp!,{r0-r12,lr}    ;保存寄存器
LDR r0,[lr,#-4]          ;计算 SWI 指令地址
BIC r0,r0,#0xff000000    ;提取指令编码的后 24 位
;
; 提取出的中断号放 r0 寄存器，函数返回
```

```
;
LDMFD sp!, {r0-r12,pc}^  ;恢复寄存器
```

在这个例子中，使用 LR-4 得到 SWI 指令的地址，再通过“BIC r0, r0, #0xff000000”指令提取 SWI 指令中断号。

2．使用 C 语言编写 SWI 异常处理函数

虽然第一级 SWI 处理函数（完成中断向量号的提取）必须用汇编语言完成，但第二级中断处理函数（根据提取的中断向量号，跳转到具体处理函数）却可以使用 C 语言来完成。

因为第一级的中断处理函数已经将中断号提取到寄存器 r0 中，所以根据 AAPCS 函数调用规则，可以直接使用 BL 指令跳转到 C 语言函数，而且中断向量号作为第一个参数被传递到 C 函数。例如，汇编中使用了“BL　C_SWI_Handler”跳转到 C 语言的第二级处理函数，而第二级的 C 语言函数示例如下。

```
void C_SWI_handler (unsigned number)
{
   switch (number)
    {
     case 0 : /* SWI number 0 code */
     break;
     case 1 : /* SWI number 1 code */
     break;
     ...
     default : /* Unknown SWI - report error */
    }
}
```

另外，如果需要传递的参数多于 1 个，那么可以使用堆栈，将堆栈指针作为函数的参数传递给 C 类型的二级中断处理程序，就可以实现在两级中断之间传递多个参数。

例如：

```
MOV r1, sp           ;将传递的第二个参数（堆栈指针）放到 r1 中
BL C_SWI_Handler     ;调用 C 函数
```

相应的 C 函数的入口变为：

```
void C_SWI_handler(unsigned number, unsigned *reg)
```

同时，C 函数也可以通过堆栈返回操作的结果。

3．从应用程序中调用 SWI

可从汇编语言或 C/C++ 中调用 SWI。

从汇编语言程序中调用 SWI，只要遵循 AAPCS 标准即可。调用前，设定所有必需的值并发出相关的 SWI。例如：

```
MOV r0, #65      ; 将软中断的子功能号放到 r0 中
SWI 0x0
```

注意：
SWI 指令和其他所有 ARM 指令一样，可以被条件执行。

7.7　FIQ 和 IRQ 中断

7.7.1　中断分支

1．软件控制中断分支

ARM 内核只有两个外部中断输入信号 nFIQ 和 nIRQ。但对于一个系统来说，中断源可能多达几十个。为此，在系统集成时，一般都会有一个异常控制器来处理异常信号，如图 7-5 所示。

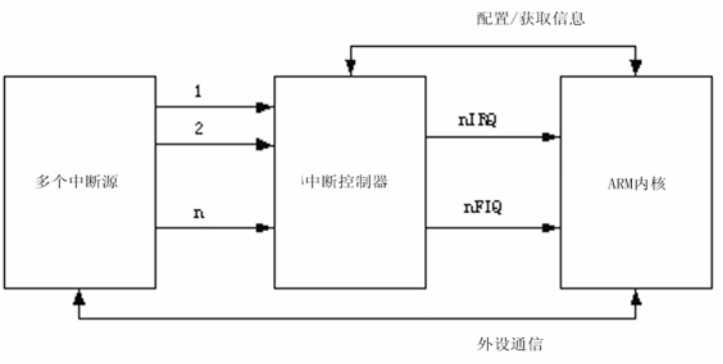

图 7-5　中断系统

这时候用户程序可能存在多个 IRQ/FIQ 的中断处理函数。为了使从向量表开始的跳转始终能找到正确的处理函数入口，需要设置处理机制和方法。在以往的 ARM 芯片中采用的是使用软件来处理异常分支，因为软件可以通过读取中断控制器来获得中断源的信息，从而达到中断分支的目的，如图 7-6 所示。

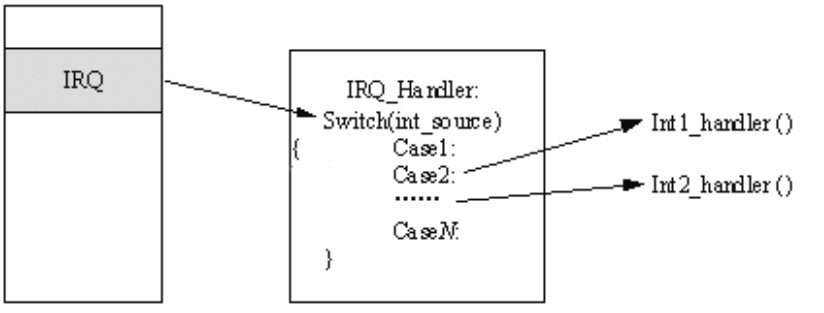

图 7-6　软件控制中断分支

因为软件的灵活性，可以设计出比图 7-6 更好的流程控制方法，如图 7-7 所示。

Int_vector_table 是用户自己开辟的一块存储器空间，里面按次序存放异常处理函数的地址。IRQ_Handler()从中断控制器获取中断源信息，然后再从 Int_vector_table 中的对应地址单元得到异常处理函数的入口地址，完成一次异常响应的跳转。这种方法的好处是用户程序在运行过程中，能够很方便地动态改变异常服务内容。

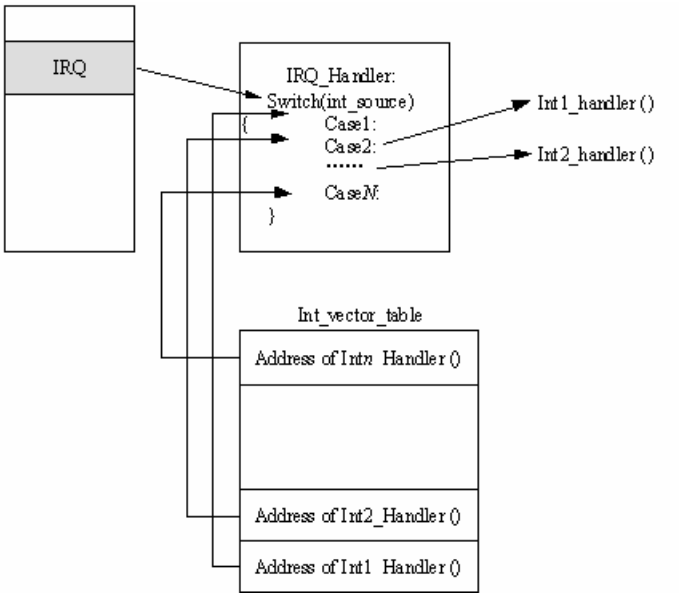

图 7-7　灵活的软件控制中断分支设计

进入异常处理程序后，用户可以完全按照自己的意愿来进行程序设计，包括调用 Thumb 状态的函数等。但对于绝大多数的系统来说，有两个步骤必须处理，一是现场保护，二是要把中断控制器中对应的中断状态标识清除，表明该中断请求已经得到响应。否则，中断函数退出以后，又会被再一次触发，从而进入周而复始的死循环。

2．向量中断控制器

这种类型的中断控制早已出现在了 ARM 芯片中，比如基于 S5PC100 的 Cortex-A8 中，以集成 PL192 向量中断控制器。使用向量中断的优点在于，中断优先级仲裁及中断分支的处理递交给了控制器来处理，这样从获取中断源，再到中断 ISR 的处理，其性能相对于软件方式的实现有很大的提高。下面是使用这种机制的详细介绍。

7.7.2　S5PC100 中断机制分析

1．向量中断概述

S5PC100 集成了 3 个向量中断控制器（后文用 VIC 来表示），采用的是 ARM 基于 PrimeCell 技术下的 PL192 核心，另外还包括了 3 个 TZIC，即针对于 TrustZone 技术所涉及的中断控制器（后文都用 TZIC 表示），其核心为 SP890。

S5PC100 中断控制器支持 94 个中断源，其中 TZIC 为 TrustZone 单独设计了一个安全软件中断接口，它提供了基于安全控制技术的 nFIQ 中断及屏蔽来自非安全系统下的所有中断源。以下是 S5PC100 中断控制器的特点：

- 支持 94 个向量 IRQ 中断。
- 灵活的硬件中断优先级。
- 可编程的中断优先级设置。
- 支持硬件上的优先级屏蔽。
- 支持编程上的优先级屏蔽。
- 内置 IRQ/FIQ/软件中断产生器。
- 内置用于调试方案的寄存器。
- 原始中断状态寄存器/中断源请求状态寄存器。
- 支持特权模式下的限制性存取数据。

当 S5PC100 收到来自片内外设和外部中断请求引脚的多个中断请求时，S5PC100 的中断控制器在中断仲裁过程后向 S5PC100 内核请求 FIQ 或 IRQ 中断。中断仲裁过程依靠处理器的硬件优先级逻辑，在处理器这边会跳转到中断异常处理例程中，执行异常处理程序，这个时候 VICADDRESS 寄存器的值就是仲裁后中断源对应的（ISR）中断处理程序的入口地址，如图 7-8 所示。

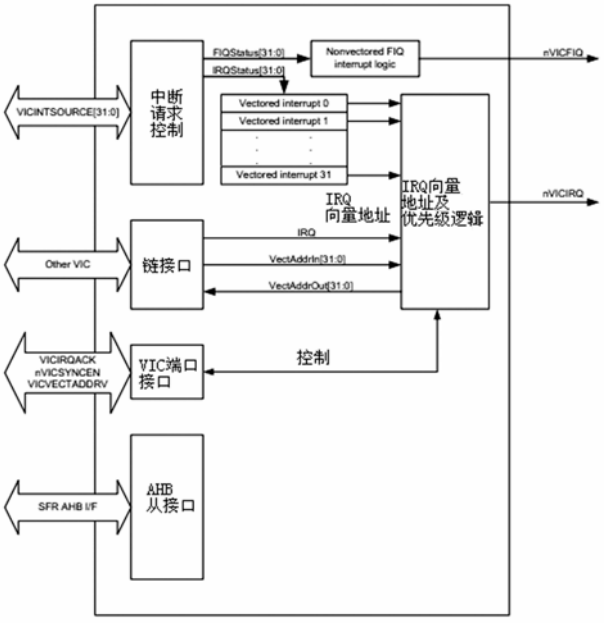

图 7-8　S5PC100 的中断控制器

S5PC100 的中断控制器的任务是在有多个中断发生时，选择其中一个中断通过 IRQ 或 FIQ 向 CPU 内核发出中断请求。实际上，最初 CPU 内核只有 FIQ（快速中断请求）和 IRQ（通用中断请求）两种中断，其他中断都是各个芯片厂家在设计芯片时，通过加入一个中

断控制器来扩展定义的，这些中断根据中断的优先级高低来进行处理，更符合实际应用系统中要求提供多个中断源的要求，除此之外，向量中断控制器比以前的中断方式更加灵活和方便，把判断的任务留给了硬件，使得中断编程更为简洁。

在整个 S5PC100 的中断向量控制器中，可以看到所有中断源会先进入 TZIC 仲裁单元，该单元需要配置为是否可通过该中断源到 VIC 单元，默认下是可以通过的，即默认为非安全模式，这样所有中断直接到 VIC 下仲裁及处理，如图 7-9 所示。

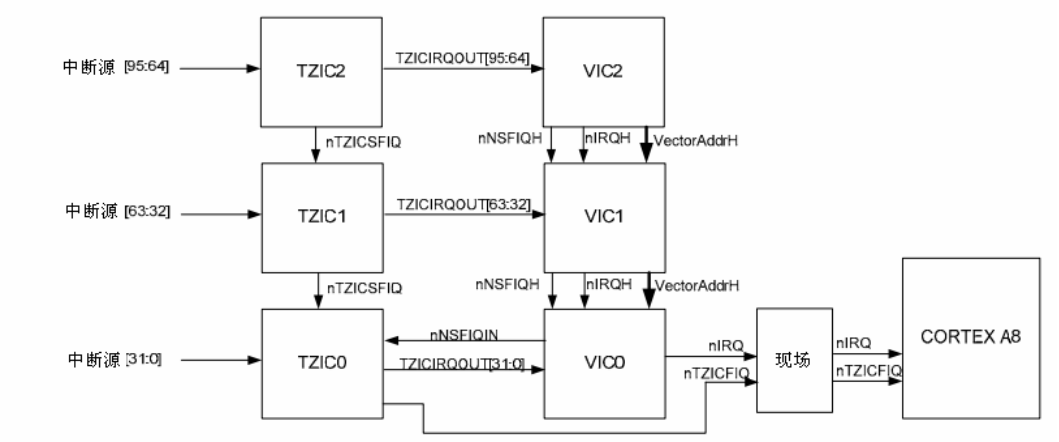

图 7-9　S5PC100 向量中断控制器

2．S5PC100 中断控制

（1）程序状态寄存器的 F 位和 I 位。如果 CPSR 程序状态寄存器的 F 位被设置为 1，那么 CPU 将不接受来自中断控制器的 FIQ（快速中断请求）；如果 CPSR 程序状态寄存器的 I 位被设置为 1，那么 CPU 将不接受来自中断控制器的 IRQ（中断请求）。因此，为了使能 FIQ 和 IRQ，必须先将 CPSR 程序状态寄存器的 F 位和 I 位清零，并且中断屏蔽寄存器 INTMSK 中相应的位也要清零。

（2）中断模式（IntSelect）。Cortex-A8 提供了两种中断模式，即 FIQ 模式和 IRQ 模式。所有的中断源在中断请求时都要确定使用哪一种中断模式。

3．S5PC100 中断源简介

在该芯片中，有 3 个 VIC 单元，其中 VIC0 涵盖了系统、DMA、定时器的中断源。VIC1 包含了 ARM 核心、电源管理、内存管理、存储管理的中断源，VIC2 则包含了多媒体、安全扩展等中断源。限于篇幅，这里只是简要的介绍，详细内容请读者自行查看用户手册。

4．S5PC100 中断控制寄存器

中断选择寄存器（0XE400000C）如表 7-5 所示。

表 7-5　中断选择寄存器（0XE400000C）

域名	位	描述	重置值
IntSelect	[31:0]	选择中断请求的类型 0 = IRQ 中断 1 = FIQ 中断	0x00000000

中断使能寄存器（0XE4000010）如表 7-6 所示。

表 7-6　中断使能寄存器（0XE4000010）

域名	位	描述	重置值
IntEnable	[31:0]	使能中断请求队列，允许执行中断处理 读： 0 = 禁止中断 1 = 使能中断 写： 0 =无影响 1 = 使能中断 复位时，屏蔽所有中断	0x00000000

中断使能清除寄存器（0XE4000014）如表 7-7 所示。

表 7-7　中断使能清除寄存器（0XE4000014）

域名	位	描述	重置值
IntEnable Clear	[31:0]	清除 VICINTENABLE 寄存器相关位： 0 = 无影响 1 = 屏蔽 VICINTENABLE 寄存器	-

ISR 入口地址寄存器（0XE4000F00）如表 7-8 所示。

表 7-8　ISR 入口地址寄存器（0XE4000F00）

域名	位	描述	重置值
VectAddr	[31:0]	当前中断处理程序的地址，重置值为 0x00000000 所读到的地址被作为 ISR 入口地址 向该寄存器写任何值将清除寄存器值	0x00000000

ISR 地址初始化寄存器如表 7-9 所示。

表 7-9　ISR 地址初始化寄存器

域名	位	描述	重置值
VectorAddr 0-31	[31:0]	装载 ISR 入口地址	0x00000000

7.7.3 S5PC100 中断处理程序实例

下面介绍一个中断实例，该例子实现了 S5PC100 按键控制。当按下 KEY1 和 KEY2 时，会从终端上打印出相应的按键信息。其中 KEY1 对应的是 EINT1 中断源，KEY2 对应的是 EINT2 中断源。

1. 电路原理

电路原理图如图 7-10 所示。

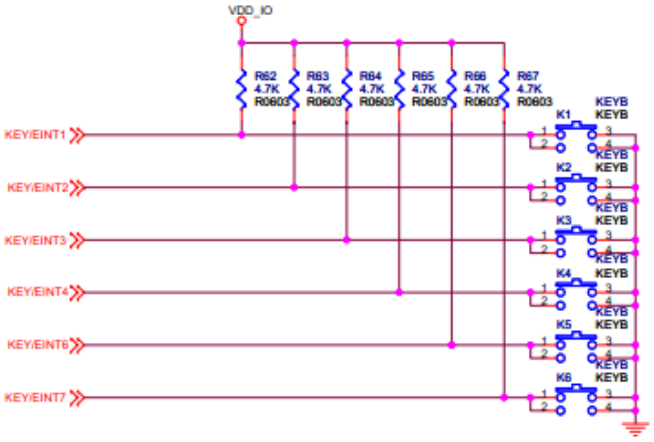

图 7-10　S5PC100 中断实验电路图

2. 编程流程

编程流程如图 7-11 所示。

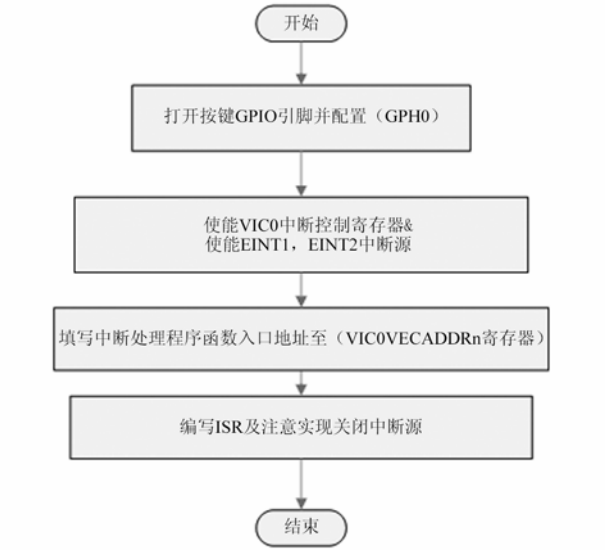

图 7-11　编程流程

3．程序编写

（1）相关寄存器定义如下。

```
#define    VIC0ADDRESS            __REG(0xE4000F00)
#define    VIC1ADDRESS            __REG(0xE4100F00)
#define    VIC2ADDRESS            __REG(0xE4200F00)
#define    VIC0VECADDR1           __REG(0xE4000104)
#define    VIC0VECADDR2           __REG(0xE4000108) //定义寄存器地址
typedef struct {
               unsigned int VIC0IRQSTATUS;
               unsigned int VIC0FIQSTATUS;
               unsigned int VIC0RAWINTR;
               unsigned int VIC0INTSELECT;
               unsigned int VIC0INTENABLE;
}interrupt;
#define  INTERRUPT (* (volatile interrupt *)0xE4000000 )
```

（2）向量中断控制器初始化及配置。

```
 VIC0VECADDR1 = (unsigned int)int_key1; //将中断向量地址寄存器赋值
GPH0.GPH0CON =(GPH0.GPH0CON & (~(0xf<<4)))+(0x2<<4);
 INTERRUPT.VIC0INTENABLE = INTERRUPT.VIC0INTENABLE | (1<<1);//使能 EINT1 源
 VIC0VECADDR2 = (unsigned int)int_key2;//将中断向量地址寄存器赋值
 GPH0.GPH0CON =(GPH0.GPH0CON & (~(0xf<<8)))+(0x2<<8);
 INTERRUPT.VIC0INTENABLE = INTERRUPT.VIC0INTENABLE | (1<<2);//使能 EINT2 源
```

（3）IRQ 跳转函数的实现。

```
void do_irq()
{
 printf("in do_irq\n");
 ((void (*)(void))VIC0ADDRESS)();
}
```

（4）按键 1 处理函数的实现。

```
/*中断处理程序 1*/
void int_key1()
{
 printf("in int_key1\r\n");
 VIC0ADDRESS = 0;//清除中断
}
```

（5）按键 2 处理函数的实现。

```
/*中断处理程序 2*/
void int_key2()
{
 printf("in int_key2\r\n");
 VIC0ADDRESS = 0; //清除中断
}
```

4．实验过程及结果描述

（1)将程序编译后产生.bin 可执行文件，然后使用 uboot 的 dnw 命令下载到 0x20008000 这个内存地址，使用 go 命令去执行，并观察结果。

（2）终端打印结果如图 7-12 所示。

```
Now, Waiting for DNW to transmit data
Download Done!! Download Address: 0x20008000, Download Filesize:0x1028
Checksum is being calculated.
Checksum O.K.
SMDKC100 # go 20008000
## Starting application at 0x20008000 ...
open uart device ok !
please press Key1 or Key2
in do_irq
in int_key2
in do_irq
in int_key2
in do_irq
in int_key1
_
```

图 7-12　终端打印结果

7.8　本章小结

本章讲解了 ARM 处理器的异常原理，以及各种异常的工作模式，另外还有 S5PC100 的向量中断机制及编程方式。读者需要结合实验来加深对异常处理和向量中断的理解。

7.9　练习题

1．简述 ARM 有几种异常，以及每种异常对应的处理器工作模式。

2．使用向量中断编写一个平台上其它按键中断的程序，并实现上升沿、下降沿、高电平、低电平等促发方式。

第 8 章　串行通信接口

串行通信接口广泛地应用于各种控制设备，是计算机、控制主板与其他设备传送信息的一种标准接口。本章主要介绍它的工作原理和编程方法。

主要内容有：

- 串行通信的基本原理。
- S5PC100 异步串行通信。
- 接口电路与程序设计。

8.1　串行通信概述

8.1.1　串行通信与并行通信概念

在微型计算机中，通信（数据交换）有两种方式：串行通信和并行通信。

1．串行通信

串行通信是指计算机与 I/O 设备之间数据传输的各位是按顺序依次一位接一位进行传送。通常数据在一根数据线或一对差分线上传输。

2．并行通信

并行通信是指计算机与 I/O 设备之间通过多条传输线交换数据，数据的各位同时进行传送。

二者比较：串行通信通常传输速度慢，但使用的传输设备成本低，可利用现有的通信手段和通信设备，适合于计算机的远程通信；并行通信的速度快，但使用的传输设备成本高，适合于近距离的数据传送。需要注意的是，对于一些差分串行通信总线，如 RS-485、RS-422、USB 等，它们的传输距离远，且抗干扰能力强，速度也比较快。

8.1.2　异步串行方式的特点

所谓异步通信，是指数据传送以字符为单位，字符与字符间的传送是完全异步的，位与位之间的传送基本上是同步的。异步串行通信的特点可以概括为：

（1）以字符为单位传送信息。

（2）相邻两字符间的间隔是任意长。

（3）因为一个字符中的比特位长度有限，所以需要的接收时钟和发送时钟只要相近就可以。

（4）异步方式特点就是：字符间异步，字符内部各位同步。

8.1.3 异步串行方式的数据格式

异步串行通信的数据格式如图 8-1 所示，每个字符（每帧信息）由 4 部分组成：

（1）1 位起始位，规定为低电 0。

（2）5～8 位数据位，即要传送的有效信息。

（3）1 位奇偶校验位。

（4）1～2 位停止位，规定为高电平 1。

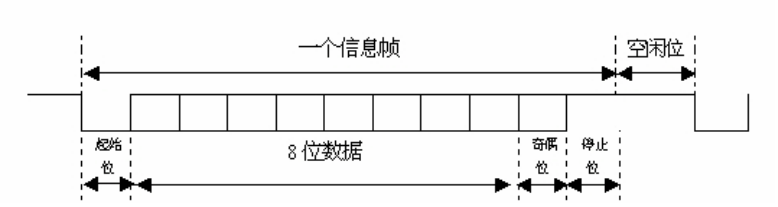

图 8-1 异步串行数据格式

8.1.4 同步串行方式的特点

所谓同步通信，是指数据传送是以数据块（一组字符）为单位，字符与字符之间、字符内部的位与位之间都同步。同步串行通信的特点可以概括为：

（1）以数据块为单位传送信息。

（2）在一个数据块（信息帧）内，字符与字符间无间隔。

（3）因为一次传输的数据块中包含的数据较多，所以接收时钟与发送时钟严格同步，通常要有同步时钟。

8.1.5 同步串行方式的数据格式

同步串行通信的数据格式如图 8-2 所示，每个数据块（信息帧）由 3 部分组成：

（1）2 个同步字符作为一个数据块（信息帧）的起始标志。

（2）n 个连续传送的数据。

（3）2 个字节循环冗余校验码（CRC）。

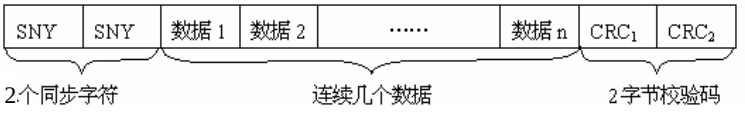

图 8-2 同步串行数据格式

8.1.6　比特率、比特率因子与位周期

比特率是指单位时间传输二进制数据的位数，其单位为位/秒（B/S）或比特。它是一个用以衡量数据传送速率的量。一般串行异步通行的传送速度为 50～19200 bit/s，串行同步通信的传送速度可达 500 kbit/s。

比特率因子是指时钟脉冲频率与比特率的比。

位周期 Td 是指每个数据位传送所需的时间，它与比特率的关系是：Td=1/比特率。它用以反映连续二次采样数据之间的间隔时间。

8.1.7　RS-232C 串口规范

RS-232C 标准（协议）的全称是 EIA-RS-232C 标准，其中 EIA（Electronic Industry Association）代表美国电子工业协会，RS（Recommeded Standard）代表推荐标准，232 是标识号，C 代表 RS232 的最新一次修改（1969），在这之前，有 RS-232B、RS-232A。它规定连接电缆和机械、电气特性、信号功能及传送过程。常用物理标准还有 EIA&#0;RS-232-C、EIA&#0;RS-422-A、EIA&#0;RS-423A 和 EIA&#0;RS-485。 这里只介绍 EIA&#0;RS-232-C（简称 232，RS-232）。例如，目前在 PC 上的 COM1、COM2 接口，就是 RS-232C 接口。

1．9 针串口引脚定义

PC 串口中的典型是 RS-232 及其兼容接口，串口引脚有 9 针和 25 针两类。而一般的 PC 中使用的都是 9 针的接口，25 针串口具有 20mA 电流环接口功能，用 9、11、18、25 针来实现。这里只介绍 9 针的 RS-232C 串口引脚定义，如表 8-1 所示。

表 8-1　9 针的 RS-232C 串口引脚定义

引脚	简写	功能说明
1	CD	载波侦测
2	RXD	接收数据
3	TXD	发送数据
4	DTR	数据终端设备
5	GND	地线
6	DSR	数据准备好
7	RTS	请求发送
8	CTS	清除发送
9	RI	振铃指示

2．RS-232C 电气特性

EIA-RS-232C 对电气特性、逻辑电平和各种信号线功能都做了明确规定。

在 TXD 和 RXD 引脚上电平定义：

逻辑 1=−3～−15V

在 RTS、CTS、DSR、DTR 和 DCD 等控制线上电平定义：

信号有效=+3～+15V

信号无效=−3～−15V

以上规定说明了 RS-232C 标准对应逻辑电平的定义。注意：对于介于−3～+3V 之间的电压处于模糊区电位，此部分电压将使得计算机无法正确判断输出信号的意义，可能得到 0，也可能得到 1，如此得到的结果是不可信的，在通信时体系会出现大量误码，造成通信失败。因此，实际工作时，应保证传输的电平在+3～+15V 或−3～−15V 之间。

3．RS-232C 的通信距离和速度

RS-232C 规定最大的负载电容为 2500pF，这个电容限制了传输距离和传输速率，由于 RS-232C 的发送器和接收器之间具有公共信号地（GND），属于非平衡电压型传输电路，不使用差分信号传输，因此不具备抗共模干扰的能力，共模噪声会耦合到信号中，在不使用调制解调器（MODEM）时，RS-232C 能够可靠进行数据传输的最大通信距离为 15 米，对于 RS-232C 远程，必须通过调制解调器进行远程通信连接，或改为 RS-485 等差分传输方式。

现在个人计算机提供的串行端口终端的传输速度一般都可以达到 115200bit/s，甚至更高，标准串口能够提供的传输速度主要有以下比特率：1200bit/s、2400bit/s、4800bit/s、9600bit/s、19200bit/s、38400bit/s、57600bit/s、115200bit/s 等，在仪器仪表或工业控制场合，9600bit/s 是最常见的传输缩短，在传输距离较近时，使用最高传输速度也是可以的。传输距离和传输速度的关系成反比，适当地降低传输速度，可以延长 RS-232 的传输距离，提高通信的稳定性。

4．RS-232C 电平转换芯片及电路

RS-232C 规定的逻辑电平与一般微处理器、单片机的逻辑电平是不同的，例如，RS-232C 的逻辑“1”是以−3～−15V 来表示的，而单片机的逻辑“1”是以 5V 表示的，S5PC100 的逻辑“1”是以 3.3V 表示的，就必须把单片机的电平（TTL、CMOS 电平）转变为 RS-232C 电平，或者把计算机的 RS-232C 电平转换成单片机的 TTL 或 CMOS 电平，通信时必须对两种电平进行转换。实现电平转换的芯片可以是分立器件，也可以是专用的 RS-232C 电平转换芯片。下面介绍一种在嵌入式系统中应用比较广泛的 MAX3232 芯片。

如图 8-3 所示，主要特点有：

- 符合所有的 RS-232C 规范。
- 单一供电电压+5V 或 3.3V。
- 片内电荷泵，具有升压。电压极行反转能力，能够产生+10V 和−10V 电压 V+、V−。
- 低功耗，典型供电电流 3mA。
- 内部集成 2 个 RS-232C 驱动器。
- 内部集成 2 个 RS-232C 接收器。

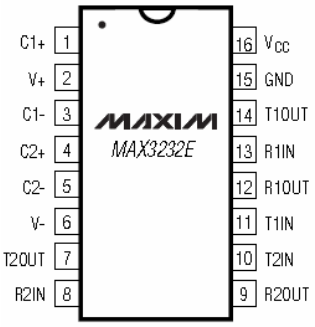

图 8-3　MAX3232 芯片

8.1.8　RS-232C 接线方式

RS-232C 串口的接线方式有全串口连接、3 线连接等方式。本书只介绍最简单、常用的 3 线连接方法。PC 和 PC 或处理器之间的通信，双方都能发送和接收，它们的连接只需使用 3 根线即可，即 RXD、TXD 和 GND，连接方式如图 8-4 所示。

图 8-4　3 线连接法

8.2　S5PC100 异步串行通信

8.2.1　S5PC100 串口控制器概述

1．简述

S5PC100 的通用异步收发（UART）可支持 4 个独立的异步串行输入/输出口，每个口皆可支持中断模式及 DMA 模式，UART 可产生一个中断或者发出一个 DMA 请求，来传送 CPU 与 UART 之间的数据，其中，通道 0 和 2 最大传输速率可达 115.2k 波特率，通道 1 和 3 最大可达到 3M 波特率，并且如果一个外设使用 UCLK 提供时钟，那么 UART 则可以工作在高速模式下，每一个 UART 通道包含两个 64 字节的收发 FIFOs。

2．特点

- 4 组收发通道，同时支持中断模式及 DMA 操作。
- 通道 0、1、2 及带红外 3 通道，都支持 64 字节 FIFO。

- 通道 1 和 3 支持高速操作模式。
- 支持握手模式的发送/接收。

3．概括图

概括图如图 8-5 所示。

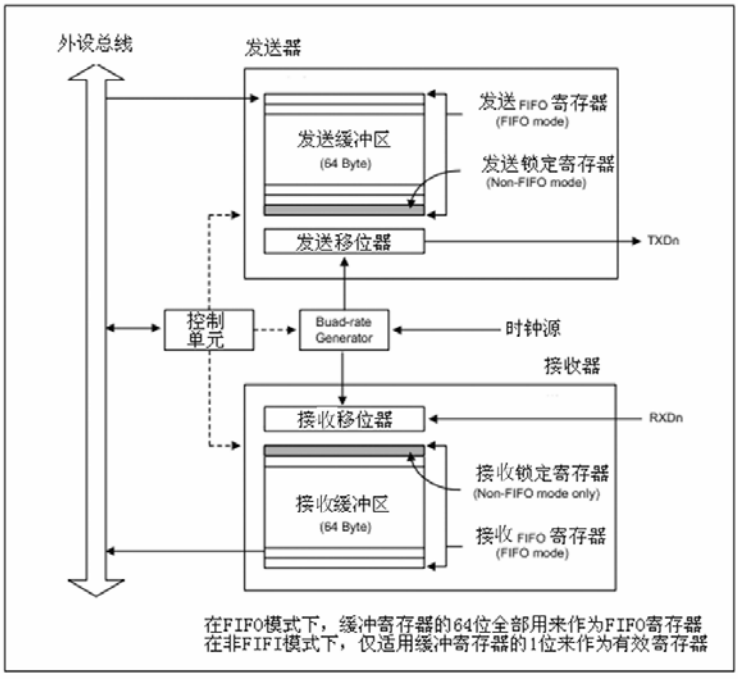

图 8-5　概括图

下面简要介绍 UART 操作，关于数据发送，数据接收，中断产生，比特率产生，轮流检测模式，红外模式和自动流控制的详细介绍，请参照相关教材和数据手册。

发送数据帧是可编程的。一个数据帧包含一个起始位，5～8 个数据位，一个可选的奇偶校验位和 1～2 位停止位，停止位通过行控制寄存器 ULCONn 配置。

与发送类似，接收数据帧也是可编程的。接收数据帧由一个起始位，5～8 个数据位，一个可选的奇偶校验和 1～2 位行控制寄存器 ULCONn 里的停止位组成。接收器还可以检测溢出错、奇偶校验错、帧错误和传输中断，每一个错误均可以设置一个错误标志。

（1）溢出错误（Overrun Error）是指已接收到的数据在读取之前被新接收的数据覆盖。

（2）奇偶校验错是指接收器检测到的校验和与设置的不符。

（3）帧错误指没有接收到有效的停止位。

（4）传输中断表示接收数据 RxDn 保持逻辑 0 超过一帧的传输时间。

在 FIFO 模式下，如果 RxFIFO 非空，而在 3 个字的传输时间内没有接收到数据，则产生超时。

8.2.2　UART 寄存器详解

为了让初学者快速掌握串口通信，下面只针对例程中用到的寄存器给予讲解。对于 S5PC100 中提供的更为复杂的控制寄存器将不再展开，感兴趣的读者可作为扩展内容自行学习。

1．UART 行控制寄存器 ULCONn（ULCON0, R/W, Address = 0xEC00_0000）

ULCONn 的含义如表 8-2 所示。

表 8-2　ULCONn 的含义

ULCONn	位	描述	初始状态
Reserved	[7]		0
Infra-Red Mode	[6]	是否使用红外模式 0=正常模式 1=红外模式	0
Parity Mode	[5:3]	校验方式 0XX=无奇偶校验 100=奇校验 101=偶校验 110=校验位强制为 1 111=校验位强制为 0	000
Number of Stop Bit	[2]	停止位数量 0=1 个停止位 1=2 个停止位	0
Word Length	[1:0]	数据位个数 00=5bit　01=6bit 10=7bit　11=8bit	00

2．UART 行控制寄存器 UCONn（UCON0,R/W Address = 0xEC00_0004）

寄存器详细说明如表 8-3 所示。

表 8-3　UCONn

UCONn	位	描　　述	初　始　值
Clock Selection	[11:10]	x0：PCLK 做比特率发生 01：UART_CLK 11 = SCLK_UART	0
Tx Interrupt Type	[9]	0：Tx 中断脉冲触发 1：Tx 中断电平触发	0
Rx Interrupt Type	[8]	0：Rx 中断脉冲触发 1：　Rx 中断电平触发	0
Rx Time Out Enable	[7]	0：接收超时中断不允许 1: 接收超时中断允许	0

续表

UCONn	位	描　　述	初　始　值
Rx Error Status Interrupt Enable	[6]	0：不产生接收错误中断 1: 产生接收错误中断	0
Loopback Mode	[5]	0：正常模式 1: 发送直接传给接收方式（Loopback）	0
Reserved	[4]	0：正常模式发送 1: 发送间断信号	0
Transmit Mode	[3:2]	发送模式选择 00：不允许发送 01：中断或查询模式 10：DMA0 请求 11:　DMA1 请求	00
Receive Mode	[1:0]	接收模式选择 00：不允许接收 01：中断或查询模式 10：DMA0 请求 11：DMA1 请求	00

3．UART FIFO 控制寄存器 UFCONn（UFCON0,R/W,ADDRESS = 0xEC00_0008）寄存器详细说明如表 8-4 所示。

表 8-4　UFCONn 的含义

UFCONn	位	描　　述	初　始　值
Tx FIFO Trigger Level	[7:6]	决定发送 FIFO 的触发位置 00=0 个字节时触发 01=16 个字节时触发 10=32 个字节时触发 11=48 个字节时触发	00
Rx FIFO Trigger Level	[5:4]	决定接收 FIFO 的触发位置 00=1 个字节时触发 01=8 个字节时触发 10=16 个字节时触发 11=32 个字节时触发	00
Reserved	[3]	保留	0
Tx FIFO Reset	[2]	Tx FIFO 复位后是否清零 0=不清零　1=清零	0
Rx FIFO Reset	[1]	Rx FIFO 复位后是否清零 0=不清零　1=清零	0
FIFO Enable	[0]	使能 FIFO 功能 0=不使能　1=使能	0

4．UART MODEM 控制寄存器 UMCONn（UMCON0,R/W,ADDRESS = 0xEC00_000C）

寄存器详细说明如表 8-5 所示。

表 8-5　UMCONn 的含义

UMCONn	位	描　　述	初 始 值
RTS trigger Level	[7:5]	如果自动流控制位使能，则以下位将决定失效 nRTS 信号： 000 = RX FIFO 填充 63 字节 001 = RX FIFO 填充 56 字节 010 = RX FIFO 填充 48 字节 011 = RX FIFO 填充 40 字节 100 = RX FIFO 填充 32 字节 101 = RX FIFO 填充 24 字节 110 = RX FIFO 填充 16 字节 111 = RX FIFO 填充 8 字节	000
Auto Flow Control (AFC)	[4]	0：不允许使用 AFC 模式 1：允许使用 AFC 模式	0
Reserved	[3:1]	保留，必须全为 0	00
Request to Send	[0]	0：不激活 nRTS 1：激活 nRTS	0

5．发送寄存器 UTXHn 和接收寄存器 URXHn

这两个寄存器存放着发送和接收的数据，在关闭 FIFO 的情况下只有一个字节 8 位数据。需要注意的是，在发生溢出错误时，接收的数据必须被读出来，否则会引发下次溢出错误。

6．比特率分频寄存器 UBRDIVn

用于串口比特率的设置。S5PC100 引入了 UDIVSLOTn，使得波特率的设置比早期处理器更加精确。下面以设置波特率为 115200 为目标，介绍设置方法。

DIV_VAL = (PCLK / (bps*16)) -1

=66.75M/115200*16 - 1　　//PCLK 由系统时钟提供，此为设定 66.75M

=35.214

UBRDIVn = 35（DIV_VAL 的整数部分）。

(UDIVSLOTn 中 1 的数量)/16 = 0.2。

(UDIVSLOTn 中 1 的数量) = 3。

根据手册中的建议 3 0x0888(0000_1000_1000_1000b)11 0xDDD5(1101_1101_1101_0101b)选择“UDIVSLOTn = 0x0888; ”。

7．串口状态寄存器 UTRSTATn（UTRSTAT0,R,ADDRESS = 0xEC00_0010）

寄存器详细说明如表 8-6 所示。

表 8-6　UTRSTATn 的含义

UTRSTATn	位	描　　述	初　始　值
Transmitter empty	[2]	发送缓冲和发送移位寄存器是否都为空 0=否 1=是	1
Transmit buffer empty	[1]	关闭 FIFO 的情况下，发送缓冲是否为空 0=不为空 1=空	1
Receive buffer data ready	[0]	关闭 FIFO 的情况下，接收缓冲是否为空 0=空 1=不为空	0

8.3　接口电路与程序设计

为实现通用串口功能及红外收发功能，这里先实现一个 S5PC100 处理器的串口通信程序，再实现一个基于红外收发的测试例子。

8.3.1　电路连接

从本章前面的知识可以看出 S5PC100 处理器集成了串口控制功能。为了实现 RS-232C 标准的串口通信功能，需要连接一个 MAX3232 电压转换芯片及一个 DB9 接头。S5PC100 串口 0 的电路连接如图 8-6 所示。

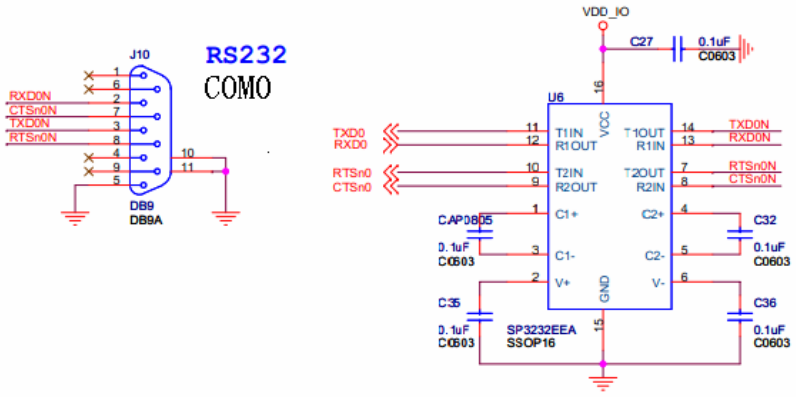

图 8-6　串口连接图

8.3.2　程序编写

程序旨在完成简单的 UART 驱动，并实现打印字符串到终端。

核心代码如下：

```
/*下面定义的是 GPA0 控制器的寄存器*/
typedef struct {
                unsigned int GPA0CON;
                unsigned int GPA0DAT;
                unsigned int GPA0PULL;
                unsigned int GPA0DRV;
                unsigned int GPA0PDNCON;
                unsigned int GPA0PDNPULL;
}gpa;
#define GPA0 (* (volatile gpa * )0xE0300000 )
/*下面定义的是 UART 控制器的寄存器*/
typedef struct {
                unsigned int ULCON0;
                unsigned int UCON0;
                unsigned int UFCON0;
                unsigned int UMCON0;
                unsigned int UTRSTAT0;
                unsigned int UERSTAT0;
                unsigned int UFSTAT0;
                unsigned int UMSTAT0;
                unsigned int UTXH0;
                unsigned int URXH0;
                unsigned int UBRDIV0;
                unsigned int UDIVSLOT0;
                unsigned int UINTP0;
                unsigned int UINTSP0;
                unsigned int UINTM0;
}uart;
#define uart_0 (*(volatile uart*)0XEC000000)
/*寄存器读/写宏定义*/
#define RAW_WRITE(addr,val) (addr = val)
#define RAW_READ(addr) (addr)
void putc(const char data)    //该函数实现了输出一个字符
{
 while(!(uart_0.UTRSTAT0 & 0X2));
 RAW_WRITE(uart_0.UTXH0,data);
 if (data == '\n')
          putc('\r');
}
void puts(const  char  *pstr) //该函数实现输出一个字符串
{
 while(*pstr != '\0')
      putc(*pstr++);
}
/*初始化 GPA0，初始化 UART 功能寄存器，模式寄存器，收发配置寄存器及最重要的控制寄存器*/
void uart0_init(void)
{
 int i;
 RAW_WRITE(GPA0.GPA0CON,0x22);
  RAW_WRITE(uart_0.UFCON0 ,0X00);      //关闭管道
 RAW_WRITE(uart_0.UMCON0, 0X00);       //关闭自动流控制 AFC
 RAW_WRITE(uart_0.ULCON0 , 0X03);      //数据长度为 8 位
  RAW_WRITE(uart_0.UCON0  , 0X305);
 RAW_WRITE(uart_0.UBRDIV0,0X23);//115200
 RAW_WRITE(uart_0.UDIVSLOT0 , 0X3);
}
int main()
{
```

```
    uart0_init();
          char *pst = "S5PC100.h";          //向屏幕打印信息
          puts(pst);

     return 0;
    }
```

8.3.3 调试与运行结果

1. 调试步骤如下。

1）终端设置

在 PC 上运行 Windows 自带的超级终端串口通信程序（比特率为 115200、1 位停止位、无校验位、无硬件流控制）如图 8-7 所示，或者使用其他串口通信程序。

图 8-7　超级终端配置

2）硬件接线

使用目标板附带的串口线连接目标板上 UART0 和 PC 串口 COMx，并将 USB OTG 口插好线。

3）下载程序

（1）先确保开发板上已有 uboot，上电后启动可以看到如图 8-8 所示内容。

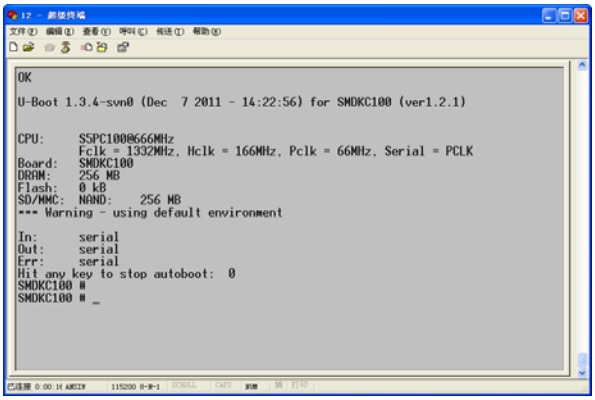

图 8-8　UBOOT 启动

（2）接下来使用 dnw 命令，它可以实现从主机端通过 USB 下载到开发板上，如图 8-9 所示。

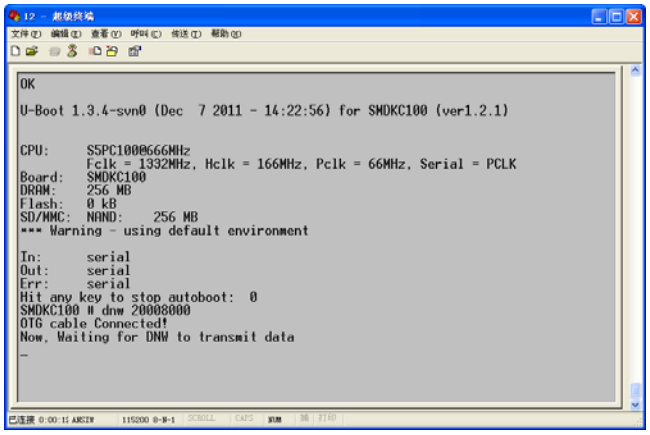

图 8-9　dnw 下载命令

（3）现在，打开 dnw 下载软件，这是由 samsung 开发的配合 dnw 命令的下载工具，如图 8-10 所示。

注意：

dnw 命令后面直接跟目标内存地址，如 dnw 21000000。

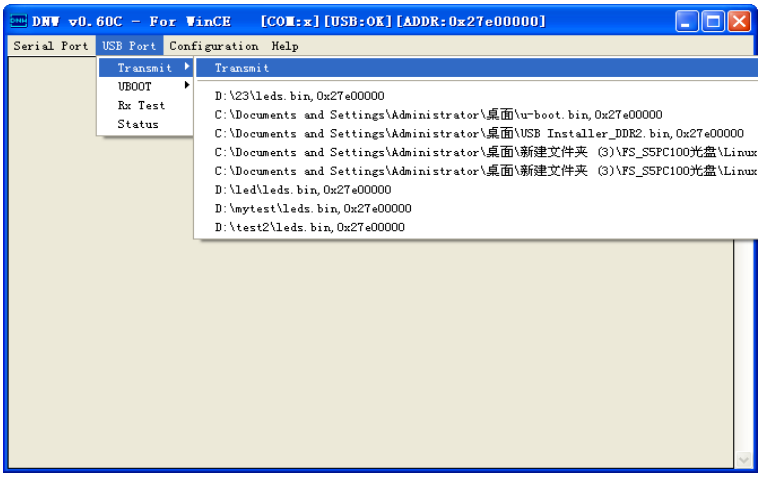

图 8-10　dnw 下载软件

（4）选择准备下载的 bin 文件后，单击发送，如果发送成功，出现如图 8-11 所示内容。

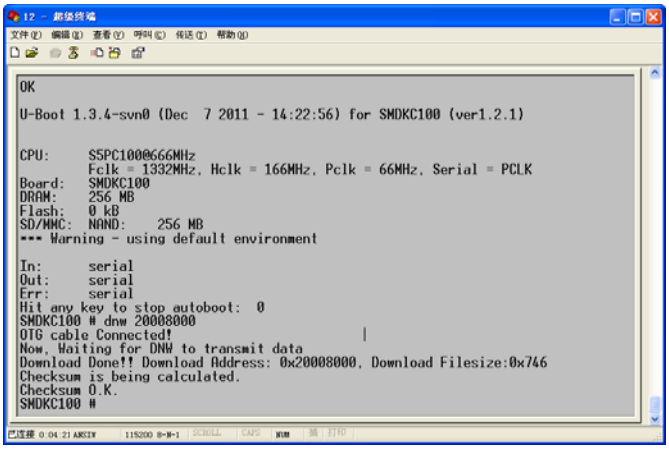

图 8-11　下载程序成功

（5）现在开始运行程序，使用 go 命令即可，如图 8-12 所示。

注意：

go 命令后面直接跟内存地址即可，如 go 20008000。

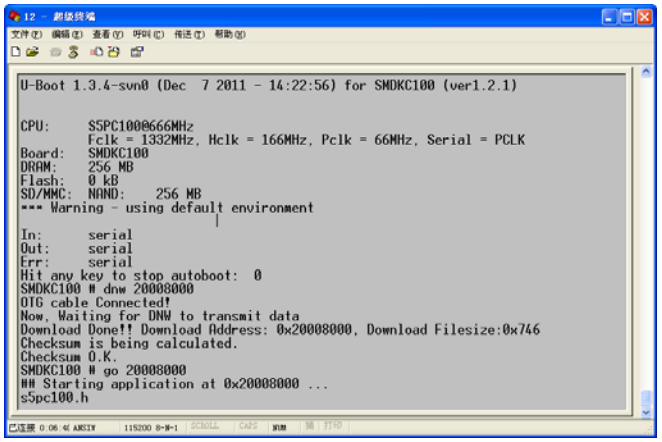

图 8-12　运行程序

可以看到我们的程序成功地打印了语句 S5PC100 和我们程序中定义的字符串。接下来可以回显从终端输入进去的字符。

8.3.4　红外收发程序

其实在使用 UART 的红外模式时，只需将寄存器指定为红外模式，其他工作方式是与通用串口保持一致的，也就是只要选定了工作模式在红外后，其他工作方式按照普通串口来设置就可以了，先看一下红外收发芯片的原理图，如图 8-13 所示。

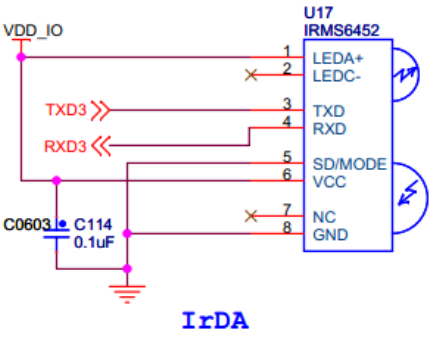

图 8-13　红外收发芯片

由图 8-12 可知，我们现在使用的 IRMS6452 红外收发芯片接在了 uart3 控制器上，这样我们需要先将 uart3 配置起来。下面列出核心代码。

```
typedef struct {
                unsigned int ULCON0;
                unsigned int UCON0;
                unsigned int UFCON0;
                unsigned int UMCON0;
                unsigned int UTRSTAT0;
                unsigned int UERSTAT0;
                unsigned int UFSTAT0;
                unsigned int UMSTAT0;
                unsigned int UTXH0;
                unsigned int URXH0;
                unsigned int UBRDIV0;
                unsigned int UDIVSLOT0;
                unsigned int UINTP0;
                unsigned int UINTSP0;
                unsigned int UINTM0;
}uart0;
#define UART0 ( * (volatile uart0 *)0XEC000000 )
/*串口寄存器结构体保持一致，这里只贴出串口 0 的寄存器结构体*/
/*打印单个字符*/
void putc(const char data)
{
 while(!(UART0.UTRSTAT0 & 0X2));
 UART0.UTXH0 = data;
 if(data == '\n')
      putc('\r');
}
/*输出字符串*/
void puts(const  char  *pstr)
{
 while(*pstr != '\0')
      putc(*pstr++);
}
/*串口控制器 0 的配置函数*/
```

```
void uart0_init(void)
{
  GPA0.GPA0CON = 0X22;    //使能 GPA0
   UART0.UFCON0 = 0X00;   //关闭管道
  UART0.UMCON0 = 0X00;    //关闭自动流控制 AFC
  UART0.ULCON0 = 0X03;    //数据长度为 8 位
  UART0.UCON0  = 0X305;
  UART0.UBRDIV0 = 0X23;   //设置波特率
  UART0.UDIVSLOT0 = 0X3;
}

/*串口控制器 3 的配置函数*/
void uart3_init(void)
{
  GPA1.GPA1CON = 0X2222;
UART3.UFCON3= 0X00;
  UART3.UMCON3 = 0X00;
  UART3.ULCON3 = 0X43 ;
  UART3.UCON3  = 0X305;
  UART3.UBRDIV3 = 0X23;
  UART3.UDIVSLOT3 = 0X3;
}

/*发送部分*/
int main()
{
  uart0_init();
  uart3_init();
  unsigned char temp;
while(1){
puts("you enter char is \n");
  while(!(UART0.UTRSTAT0 & 0x1)); //接收数据准备信号
            {
            temp = UART0.URXH0;
            UART3.UTXH3 = temp;
            UART0.UTXH0 = temp;
            while(!(UART0.UTRSTAT0 & (0x1<<2)));
            while(!(UART3.UTRSTAT3 & (0x1<<2)));
}
  }
}

/*接收部分*/
int main()
{
  uart0_init();
  uart3_init();
  unsigned char temp;
while(1){
  while(!(UART3.UTRSTAT3 & 0x1)); //Receive data ready
            {
            temp = UART3.URXH3;
            UART0.UTXH0 = temp;
```

```
            while(!(UART0.UTRSTAT0 & (0x1<<2)));
            }
    }
  }
```

测试方法为，将收与发程序分别下载到开发板以后，只需一端发送，另一端就可以接收了，该代码只实现了单向传送，希望读者能补充完整。

8.4　本章小结

本章重点介绍了串口通信的概念、数据规范、S5PC100 串口控制器及编程方法。串口控制器对于学习来说是一个比较典型的控制器，有 FIFO 单元，支持中断、DMA 控制。如果读者能够掌握它们的控制方法，对于其它控制器的学习会非常有益。笔者编写了打开 FIFO 后的串口程序、FIFO 集合中断的程序、FIFO 集合中断和 DMA 的程序。读者可以在华清远见研发中心关于 FS_S5PC100 平台的论坛上下载相关代码。

8.5　练习题

1．串行通信与并行通信的概念是什么？

2．同步通信与异步通信的概念及区别是什么？

3．RS-232C 串口通信接口规范是什么？

4．在 S5PC100 串口控制器中，哪个寄存器用来设置串口比特率？

5．编写一个串口程序采用中断的方式，实现向 PC 的串口终端打印一个字符串“hello”的功能。

第 9 章　存储器接口

存储器是计算机系统的一个重要组成部分，通常可以分为非易失存储器和易失存储器。本章将介绍在嵌入式平台上常用的 Flash 存储器（非易失）。

主要内容有：

- Flash ROM 介绍。
- S5PC100 中 NOR Flash 操作。
- S5PC100 中 NAND Flash 操作。

9.1　Flash ROM 介绍

Flash 器件是近年来发展很快的新型半导体存储器。它的主要特点是在不加电的情况下能长期保持存储的信息。就其本质而言，Flash Memory 属于 EEPROM（电擦除可编程只读存储器）类型。它既有 ROM 的特点，又有很高的存取速度，而且易于擦除和重写，功耗很小。

Flash 是在 E2PROM 的基础上发展而来的，它通过向多晶硅浮栅极充电至不同的电平来对应不同的阈电压而代表不同的数据。Flash 存储单元有两种基本类型结构：单级单元 SLC（Single-Level Cell）和多级单元 MLC（Multi-Level Cell）。传统的 SLC 存储单元只有两个阈电压（O/1），只能存储 1 位信息。MLC 的每个存储单元中有 4 个阈电压（00/01/10/11），可以存储 2 位信息；MLC 技术能够得到较大的存储容量。

由于 Flash Memory 的独特优点，如在一些较新的主板上采用 Flash ROM BIOS，会使得 BIOS 升级非常方便。 Flash Memory 可用做固态大容量存储器。目前普遍使用的大容量存储器仍为硬盘。硬盘虽有容量大和价格低的优点，但它是机电设备，有机械磨损，可靠性及耐用性相对较差，抗冲击、抗振动能力弱，功耗大。因此，一直希望找到取代硬盘的手段。由于 Flash Memory 集成度不断提高，价格降低，使其在便携机上取代小容量硬盘已成为可能。在一些 Flash 驱动卡中，除 Flash 芯片外还有由微处理器和其他逻辑电路组成的控制电路。它们与 IDE 标准兼容，可在 DOS 下像硬盘一样直接操作，因此也常把它们称为 Flash 固态盘。Flash Memory 的不足之处仍然是容量还不够大，价格还不够便宜。因此主要用于要求可靠性高，重量轻，但容量不大的便携式系统中。

本书主要讨论 Flash 存储芯片在嵌入式系统中的应用。由于 Flash 器件的成本体积小、抗震性能好等特点，使其非常适合作为非易失存储器应用于嵌入式系统中。

根据存储单元的组合形式差异，Flash 主要有两种类型：“或非 NOR”和“与非 NAND”。

NOR 和 NAND 是现在市场上两种主要的非易失闪存技术。Intel 于 1988 年首先开发出 NOR Flash 技术，彻底改变了原先由 EPROM 和 EEPROM 一统天下的局面。紧接着，1989 年，东芝公司发表了 NAND Flash 结构，强调降低每比特的成本，拥有更高的性能，并且像磁盘一样可以通过接口轻松升级。下面分析二者的特性及对比它们的差别。

1．接口对比

NOR Flash 带有通用的 SRAM 接口，可以轻松地挂接在 CPU 的地址、数据总线上，对 CPU 的接口要求低。NOR Flash 的特点是芯片内执行（eXecute In Place，XIP），这样应用程序就可以直接在 Flash 闪存内运行，不必再把代码读到系统 RAM 中。

NAND Flash 器件使用复杂的 I/O 口来串行地存取数据，8 个引脚用来传送控制、地址和数据信息。由于时序较为复杂，所以越来越多的 ARM 处理器都集成 NAND 控制器。另外，由于 NAND Flash 没有挂接在地址总线上，所以如果想用 NAND Flash 作为系统的启动盘，就需要 CPU 具备特殊的功能，如 s3c2410 被选择为 NAND Flash 启动方式，会在上电时自动读取 NAND Flash 的 4K 数据到地址 0 的 SRAM 中。如果 CPU 不具备这种特殊功能，用户不能直接运行 NAND Flash 上的代码，可以采取其他方式，比如很多使用 NAND Flash 的嵌入式平台除了使用 NAND Flash 以外，还用上了一块小的 NOR Flash 来运行启动代码。

2．容量和成本对比

与 NAND Flash 相比，NOR Flash 的容量要小，一般为 1~32MB。随着技术的发展，容量也会不断增加。

在相同容量的情况下，在价格方面，NOR Flash 相比 NAND Flash 来说较高。另外，由于 NAND Flash 生产过程更为简单，NAND 结构可以在给定的模具尺寸内提供更高的容量，这样也相应地降低了价格。

3．可靠性对比

NAND 器件中的坏块是随机分布的，以前也曾有过消除坏块的努力，但发现成品率太低，代价太高，根本不划算。NAND 器件需要对介质进行初始化扫描以发现坏块，并将坏块标记为不可用。在已制成的器件中，如果通过可靠的方法不能进行这项处理，将导致高故障率。而坏块问题在 NOR Flash 上是不存在的。

在 Flash 的位翻转（一个位发生翻转）现象上，NAND 的出现概率要比 NOR 大得多。这个问题在 Flash 存储关键文件时是致命的，所以在使用 NAND Flash 时建议同时使用 EDC/ECC 等校验算法。

4．寿命对比

在 NAND 闪存中每个块的最大擦写次数是一百万次，而 NOR 的擦写次数是十万次。闪存的使用寿命同时和文件系统的机制也有关，要求文件系统具有损耗平衡功能。

5．升级对比

NOR Flash 的升级较为麻烦，因为不同容量的 NOR Flash 的地址线需求不一样，所以

在更换不同容量的 NOR Flash 芯片时不方便。通常我们会通过在电路板的地址线上做一些跳接电阻来解决这样的问题，针对不同容量的 NOR Flash。

而不同容量的 NAND Flash 的接口是固定的，所以升级简单。

6．读/写性能对比

任何 Flash 器件的写入操作只能在空或已擦除的单元内进行。NAND 器件执行擦除操作是十分简单的，而 NOR 则要求在进行擦除前先将目标块内所有的位都写为 1。擦除 NOR 器件是以 64～128KB 的块进行的，执行一个写入/擦除操作的时间约为 5s。擦除 NAND 器件是以 8～32KB 的块进行的，执行相同的操作最多只需要 4ms。

NOR 的读速度比 NAND 稍快一些。

下面将分别以具体的 NOR Flash 芯片和 NAND Flash 芯片为例，介绍 Flash 的操作方法。

9.2 NOR Flash 操作

9.2.1 AM29LV160D 芯片介绍

AM29LV160D 是一个 1M×16 的 3V 供电的 Flash 器件，该芯片提供 48-ballFBGA 封装，44-PIN SO 封装，48-PIN TSOP 封装。在 023μm 的工艺下，完全兼容 0.32μm 工艺的芯片。AM29LV160D 具有高性能及非常灵活的编程能力，可以选择 Byte 模式及 Word 模式。支持整片擦除，支持片擦除，片保护功能。器件通过触发位或数据查询位来指示编程操作的完成。为了防止意外写的发生，器件还提供了硬件和软件数据保护机制。

AM29LV160D 的工作电压为 2.7～3.6V，单片存储容量为 2MB，如图 9-1 所示为 48-PIN 的 TSOP 封装图。

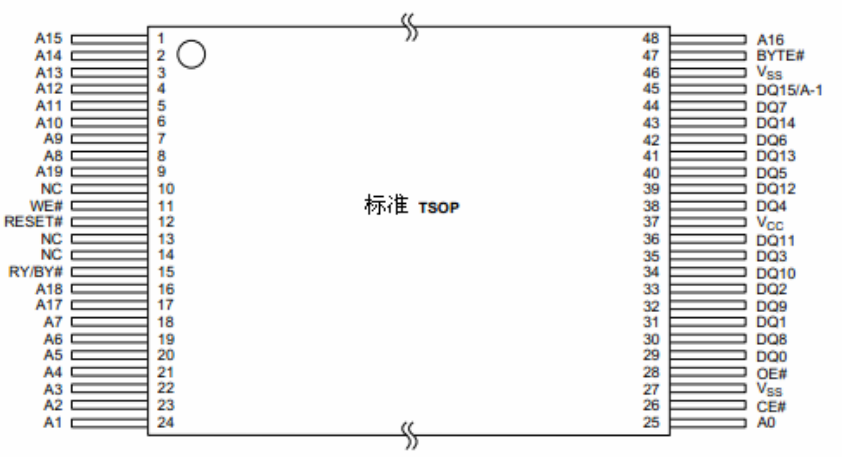

图 9-1　AM29LV160D 引脚图

AM29LV160D 的引脚功能描述如图 9-2 所示。

引脚配置		
A0–A19	=	20（地址线）
DQ0–DQ14	=	15 data 输入/输出
DQ15/A-1	=	DQ15（数据引脚）
BYTE#	=	选择8位模式或者16位模式
CE#	=	芯片擦除
OE#	=	输出使能
WE#	=	写使能
RESET#	=	硬件复位引脚
RY/BY#	=	闲/忙状态输出
V_{CC}	=	3.0v提供电压
V_{SS}	=	设备接地引脚
NC	=	引脚未内部连接

图 9-2　引脚功能描述表

AM29LV160D 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#及 CE#保持低，并且将 OE#拉高来写入命令地址。编程操作时，BYTE#引脚决定设备所接收的数据的长度。

AM29LV160D 的读操作由 CE#和 OE#控制，读的时候，它们必须被拉低，因为只有两者都为低电平时系统才能从器件的输出引脚获得数据。WE#应该维持在高电平，在设备复位后即可进行读操作。在标准微处理器的读周期内，将合法地址输入到设备后，将会读取数据，直到设备的状态被改变，如图 9-3 所示。

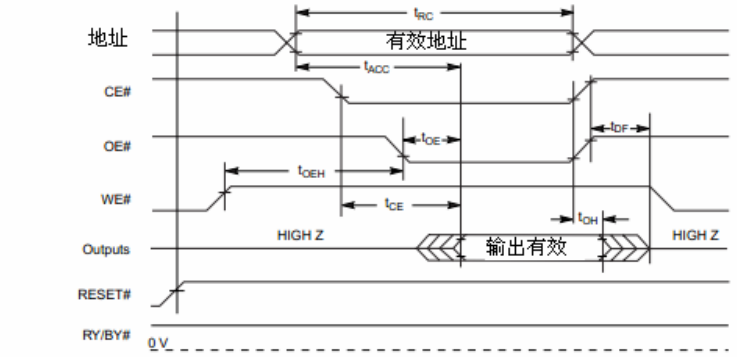

图 9-3　AM29LV160D 读时序

9.2.2　AM29LV160D 字编程操作

AM29LV160D 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 3 步。第一步，执行 3 字节装载时序，用于解除软件数据保护。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#的下升沿（后产生下降沿的那个）锁存，数据

在 CE#或 WE#的上升沿（先产生上升沿的那个）锁存。第三步，执行内部编程操作。在第 4 个 WE#或 CE#的上升沿出现（先产生上升沿的那个）之后启动编程操作。一旦启动，将在 20 s 内完成。4 个总线写周期的软件命令时序如表 9-1 所示。

表 9-1　写周期的软件命令时序

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	
	地址	数据	地址	数据	地址	数据	地址	数据
字编程	555H	AAH	2AAH	55H	555H	A0	编程地址	数据

9.2.3　AM29LV160D 扇区/块擦除操作

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。数据查询位和触发位的定义如下。

数据查询位（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，其余的数据输出引脚上的数据也无效，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。

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。

擦除周期的软件命令时序如表 9-2 所示。

表 9-2　擦除周期的软件命令时序

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	数据	地址	数据	地址	数据	地址	数据	地址	数据	地址	数据
扇区擦除	555H	AAH	2AAH	55H	555H	80H	555H	AAH	2AAH	55H	555H	10H
块擦除	555H	AAH	2AAH	55H	555H	80H	555H	AAH	2AAH	55H	SA	30H

注：SA＝ 扇区地址。

9.2.4　AM29LV160D 芯片擦除操作

AM29LV160D 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 周期的命令序列，擦除命令序列中包括两个解锁命令，一个设备启动的命令，一个片擦除命令，最后两个增加的解锁命令，注意，在擦操作的时候，需要检查 DQ7-DQ0 来获得状态，以此来判断擦除是否完成。如图 9-4 所示为擦除命令时序。

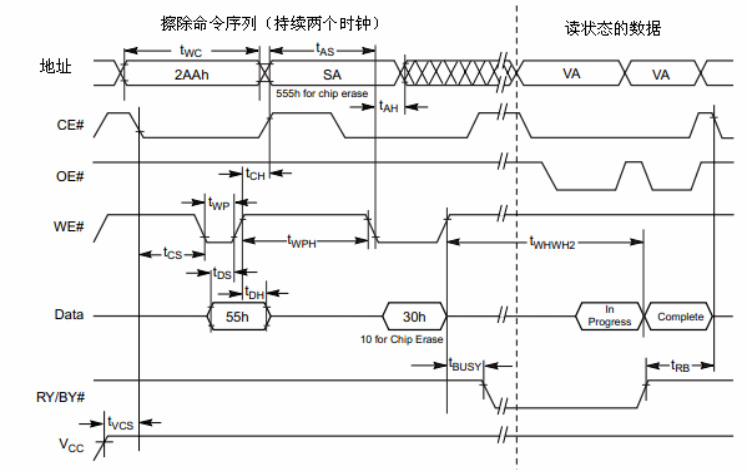

图 9-4　擦除命令时序

9.2.5　AM29LV160D 与 S5PC100 的接口电路

如图 9-5 所示为一片 AM29LV160D 以 16 位的方式和 S5PC100 的接口电路。

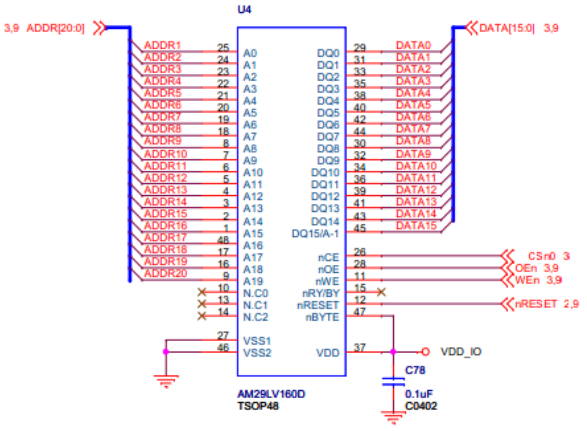

图 9-5　AM92LV160D 与 S5PC100 的接口电路

9.2.6 AM29LV160D 存储器的程序设计

1. 字编程操作

下面举例的 FlashProg 函数，实现的是将从内存“DataPtr”地址的连续“WordCnt”个 16 位的数据写入 AM29LV160D 的“ProgStart”地址。函数中用到的几个宏的定义如下：

```
#define BASEADDR 0x80000000
#define    write_word(addr,value)    ({    (*(volatile    unsigned    short*)(BASEADDR
+(addr<<2)))=(unsigned short)value ;})
#define read_word(addr)  (*(volatile unsigned short*)(BASEADDR + (addr<<2)))
#define reset() ({write_word(0x0,0xf0);})
```

需要注意的是，由于 S5PC100 的 ADDR1 是和 AM29LV160D 的 A0 连接在一起的，所以为了满足 AM29LV160D 的要求，需要将软件命令时序表（参考表 9-2 写周期的软件命令时序）中提到的地址，如“0x555”左移一位，在表达式中表现为“0x555*2”。

另外，函数还利用了数据查询位（DQ7）和查询位（DQ6）来判断编程操作是否完成。实际项目中，学员选择一种方式即可。

```
void WriteBuffWord(unsigned short addr,unsigned short value)
{
 write_word(0x555,0xaa);
 write_word(0x2aa,0x55);
 write_word(0x555,0xa0);
 write_word(addr,value);

 PollToggle 位(0);
 printf("write done \n");
}
```

2. 芯片擦除操作

下面的代码实现 AM29LV160D 的片擦除工作。

```
void ChipErase()
{
 printf("start to erase maybe 1-2 min\n");
 write_byte(0xaaa,0xaa);
 write_byte(0x555,0x55);
 write_byte(0xaaa,0x80);
 write_byte(0xaaa,0xaa);
 write_byte(0x555,0x55);
 write_byte(0xaaa,0x10);


 PollToggle 位(0);
 printf("erase work has done\n");
}
```

3. 读操作

FlashRead 函数实现了从“ReadStart”位置读取“Size”字节的数据到“DataPtr”中。

```
void  FlashRead(unsigned int ReadStart, unsigned short *DataPtr, unsigned int Size)
{
```

```
        int i;
        ReadStart += ROM_BASE;
        for(i=0; i<Size/2; i++)
        *(DataPtr+i)=*((unsigned short *)ReadStart+i);
}
```

4．状态位判断算法

由于每种操作都可以由不同的组合位来判断，因此读者可查看手册具体分析。

```
/*
 *ulAddr the data# polling Algorithm at address
 *
*/
void PollToggle(unsigned long ulAddr)
{
 int  ErrorCode = 1;
 unsigned short usVal1;
 unsigned short usVal2;
 usVal1 = read_word(ulAddr);
 while( ErrorCode == 1)
 {
      usVal1 = read_word( ulAddr);
      usVal2 = read_word( ulAddr );
      usVal1 ^= usVal2;
      if( !(usVal1 & 0x40) )
           break;
if( !(usVal2 & 0x20) )
           continue;
      else
      {
           usVal1 = read_word( ulAddr );
           usVal2 = read_word( ulAddr);
           usVal1 ^= usVal2;
           if( !(usVal1 & 0x40) )
                break;
           else
           {
                ErrorCode = 2;
                reset();
           }
      }
 }
}
```

5．主程序

编写一个程序调用上文的 Flash 操作函数实现读写及擦除功能。

```
int main()
{
 uart0_init();
 printf("start from:\n");
 unsigned short test = read_word(0x150); //从 0x150 地址读出一个 word（16bit）数据
 printf("before write %x \n",test);
 printf("write the data 0x1234 \n");
 WriteBuffWord (0x150,0x1234);    //在 0x150 地址写入一个 word 数据 0x1234
 test = read_word (0x150);        //从 0x150 地址读出一个 word
```

```
    printf("after write ,the value is %x\n",test);      // 打印出结果比对前一次的，看是否一致
    ChipErase();                        //擦除整个芯片
    test = read_word (0x150);           //再次读出数据
    printf("after erase,the value is %x\n",test);
    return 0;
}
```

实验步骤与测试结果：

（1）将程序编译后产生.bin 可执行文件，然后使用 UBOOT 的 DNW 命令下载到 0x20008000 内存地址，使用 go 命令去执行，并观察结果。

（2）终端打印信息如图 9-6 所示。

```
Hit any key to stop autoboot:  0
SMDKC100 #
SMDKC100 # dnw 20008000
Insert a OTG cable into the connector!
OTG cable Connected!
Now, Waiting for DNW to transmit data
Download Done!! Download Address: 0x20008000, Download Filesize:0x1780
Checksum is being calculated.
Checksum O.K.
SMDKC100 # go 20008000
## Starting application at 0x20008000 ...
open uart device ok !start from:
before write ffff
write the data 0x1234
write done
after write ,the value is 1234
start to erase maybe 1-2 min
erase work has done
after erase,the value is ffff
```

图 9-6　测试结果

9.3　NAND Flash 操作

9.3.1　芯片介绍

S5PC100 处理器集成了 8 位 NAND Flash 控制器。目前市场上常见的 8 位 NAND Flash 有三星公司的 K9F2G080U、K9F1G08、K9F2G08 等。K9F2G080U、K9F1G08、K9F2G08 的数据页大小分别为 512B、2KB、2KB。它们在寻址方式上有一定差异，所以程序代码并不通用。

K9F2G080U 是 Samsung 公司生产的采用 NAND 技术的大容量、高可靠 Flash 存储器。该器件存储容量为 256M 字节，除此之外还有 8M 字节的 spare 存储区。该器件采用 TSOP48 封装，工作电压为 2.7～3.6V。K9F2G080U 对 2048 字节一页的写操作所需时间典型值是 25μs，而对 16K 字节一块的擦除操作典型仅需 2ms。8 位 I/O 端口采用地址、数据和命令复用的方法。这样既可减少引脚数，还可使接口电路简洁，如图 9-7 所示。

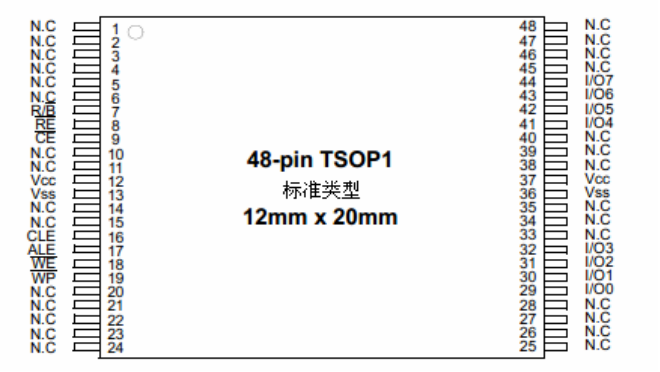

图 9-7　K9F2G080U 引脚图

K9F2G080U 的引脚功能描述如表 9-3 所示。

表 9-3　K9F2G080U 的引脚功能

引 脚 名 称	描　　述	引 脚 名 称	描　　述
I/O0 ~ I/O7	数据输入/输出	WP#	写保护
CLE	命令锁存使能	R/B#	准备好/忙碌
ALE	地址锁存使能	VCC	电源（+2.7V~3.6V）
CE#	片选	VSS	地
RE#	读使能	N.C	空引脚
WE#	写使能		

NAND Flash 的数据是以位的方式保存在 Memory Cell 的。一般来说，一个 Cell 中只能存储一个位。这些 Cell 以 8 个或者 16 个为单位，连成位 Line，形成所谓的 Byte(X8)/Word(X16)，这就是 NAND Device 的位宽。这些 Line 组成 Page，Page 再组织形成一个 Block。K9F2G080U 的相关数据如下：

1block = 64page；1page = 2048bytes + 64byte(Spare Area)

总容量 = 2048 (blocks) × 64(page) × 2048(byte) = 256MB

NAND Flash 以页为单位读/写数据，而以块为单位擦除数据。按照 K9F2G080U 的组织方式可以分 4 类地址：Column Address、halfpage pointer、Page Address 和 Block Address。A[0:28]表示数据在 256MB 空间中的地址。

对 NAND Flash 的操作主要包括读操作、擦除操作、写操作、坏块设别、坏块标识等。本书主要介绍读操作、擦除操作、写操作的实现过程。

9.3.2　读操作过程

K9F2G080U 的寻址分为 5 个 cycle，分别是 A[0:7]、A[8:11]、A[12:19]、A[20:27]、A[28]，如表 9-4 所示。

表 9-4　K9F2G080U 读操作周期

	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
2nd Cycle	A8	A9	A10	A11	*L	*L	*L	*L
3rd Cycle	A12	A13	A14	A15	A16	A17	A18	A19
4th Cycle	A20	A21	A22	A23	A24	A25	A26	A27
5th Cycle	A28	*L	*L	*L	*L	*L	*L	*L

K9F2G080U 提供了两个读指令：“0x00”和“0x30”。读操作的对象为一个页面，建议从页边界开始读写至页结束。K9F2G080U 读操作流程如图 9-8 所示。

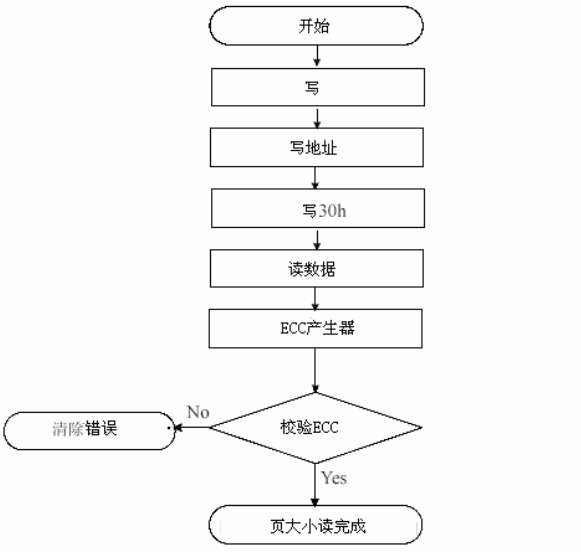

图 9-8　K9F2G080U 读操作流程图

9.3.3　擦除操作过程

如图 9-9 所示为擦除 K9F2G080U 一个块的操作过程。擦除的操作过程为：（1）发送擦除指令“0x60”；（2）发送第 1 个 cycle 地址；（3）发送第 2 个 cycle 地址；（4）发送第 3 个 cycle 地址；（5）发送擦除指令“0xD0”；（6）发送查询状态命令字“0x70”；（7）读取 K9F2G080U 的数据总线，判断 I/O 0 上的值或判断 R/B 线上的值，直到 I/O 6 = 0；（8）判断 I/O 0 是否为 0，从而确定操作是否成功。0 表示成功，1 表示失败。

注意：

擦除的对象是一个数据块，即 128 个页面。

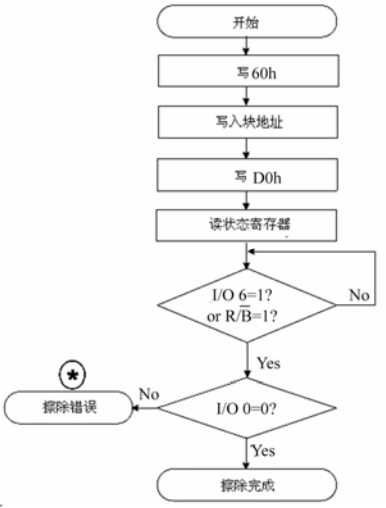

图 9-9　K9F2G080U 擦除操作流程图

9.3.4　写操作过程

如图 9-10 为写入 K9F2G080U 某一扇区或一页的数据流程图。写入的操作过程为：（1）发送编程指令“0x80”；（2）发送第 1~5 个 cycle 地址；（3）向 K9F2G080U 的数据总线发送一个页的数据；（4）发送编程指令“0x10”；（5）发送查询状态命令字“0x70”；（6）读取 K9F2G080U 的数据总线，判断 I/O 0 上的值或判断 R/线上的值，直到 I/O 6=1 或 R/ =1；（7）判断 I/O 0 是否为 0，从而确定操作是否成功。0 表示成功，1 表示失败。

注意：
写入的操作对象是一个页面。

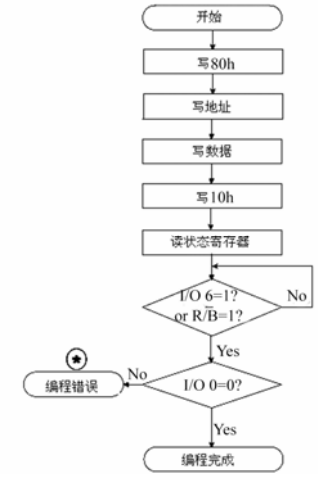

图 9-10　K9F2G080U 写操作流程图

9.4 S5PC100 中 NAND Flash 控制器的操作

9.4.1 S5PC100 NAND Flash 控制器概述

当前，NOR Flash 存储器的价格比较昂贵，而 SDRAM 和 NAND Flash 存储器的价格相对来说比较合适，这样就激发了一些用户希望用 NAND Flash 启动和引导系统，而在 SDRAM 上执行主程序代码的想法。

S5PC100 恰好满足这一要求，它可以实现从 NAND Flash 上执行引导程序。为了支持 NAND Flash 的系统引导，S5PC100 具备了一个内部 iRom。当系统启动时，c 处理器首先执行 iRom 中的代码，这段代码的功能为，将 NAND Flash 存储器的前面 4K 字节自动载入到 iRam 中，然后处理器跳到 iRam 的首地址中并执行引导代码。

9.4.2 S5PC100 NAND Flash 控制器寄存器详解

下面列出 S5PC100 NAND Flash 控制器中的 NFCONF、NFCMD、NFADDR、NFDATA、NFSTAT 几个比较重要寄存器各个位的含义。由于篇幅有限，这里只列出我们关心的寄存器信息。

（1）配置寄存器 NFCONF（地址 0XE7200000）如表 9-5 所示。

表 9-5 NAND Flash 控制寄存器（功能部分）,R/W, ADDRESS=0xE7200000

域	位	描述	复位值
保留	[22:15]	保留	000000000
TACLS	[14:12]	CLE & ALE 持续时间设置(0-7) 持续时间 = HCLK×TACLS	001
保留	[11]	保留	0
TWRPH0	[10:8]	TWRPH0 持续时间设置(0-7) 持续时间= HCLK×(TWRPH0 + 1)	000
保留	[7]	保留	0
TWRPH1	[6:4]	TWRPH1 持续时间设置(0-7) 持续时间= HCLK×(TWRPH1 + 1)	000
MLCFlash	[3]	该位指定了 NAND Flash 的使用方式 0 = SLC NAND Flash 1 = MLC NAND Flash	0
页大小	[2]	NAND Flash 一个页的大小 如果 MLCFlash is 0： 0 = 2048 Bytes/page 1 = 512 Bytes/page 如果 MLCFlash is 1： 0 = 4096 Bytes/page 1 = 2048 Bytes/page	0

续表

域	位	描述	复位值
地址周期	[1]	该位指定了 NAND 存取周期的个数 当页大小是 512 Byte, 0 = 3 address cycle 1 = 4 address cycle 当页大小是 2K or 4K, 0 = 4 address cycle 1 = 5 address cycle	0
保留	[0]	保留	0

（2）NAND Flash 控制寄存器如表 9-6 所示

表 9-6　NAND Flash 控制寄存器（时序控制部分）,R/W,ADDRESS=0xE720000

域	位	描述	复位值
RnB_TransMode	[8]	R/B 位探测方式配置 0 =上升沿探测 1 = 下降沿探测	0
Reg_nCE1	[2]	NAND Flash　nRCS[1] 控制信号	1
Reg_nCE0	[1]	NAND Flash　nRCS[0] 控制信号 0 = 强制 nRCS[0] 拉低 (片选) 1 = 强制 nRCS[0] 拉高 (停用)	1
MODE	[0]	NAND Flash 控制器操作模式 0 = 禁止 NAND Flash 控制器 1 = 使能 NAND Flash 控制器	0

命令寄存器 NFCMD（地址：0XE7200008）如表 9-7 所示。

表 9-7　NFCMD 描述

域	位	描述	复位值
保留	[31:8]	保留	0x000000
REG_CMMD	[7:0]	NAND Flash 存储器命令值	0x00

地址寄存器 NFADDR（地址：0xE720000C）如表 9-8 所示。

表 9-8　NFADDR 描述

域	位	描述	复位值
保留	[31:8]	保留	0x000000
REG_ADDR	[7:0]	NAND Flash 存储器地址值	0x00

数据寄存器 NFDATA（地址：0xE7200010）如表 9-9 所示。

表 9-9　NFDATA 描述

域	位	描述	复位值
NFDATA	[31:0]	NAND Flash 读/写数据值	0x00000000

状态寄存器 NFSTAT（地址：0x4E000010）如表 9-10 所示。

表 9-10　NFSTAT 描述

域	位	描述	复位值
Flash_RnB_GRP	[31:28]	RnB[3:0]引脚状态. 0 = NAND Flash 工作中 1 = NAND Flash 准备工作	0x0
保留	[23:12]	保留	0x800
RnB_TransDetect	[4]	如果 RnB 发生了从低到高, 并引发中断，清除则填 1 0 = RnB 转换未侦测 1 = RnB 转换侦测到	1
Flash_nCE[1] (Read-only)	[3]	nCE[1] 输出引脚的状态	1
Flash_nCE[0] (Read-only)	[2]	nCE[0] 输出引脚的状态	1
保留	[1]	保留	0
保留	[0]	保留	1

9.5　S5PC100 NAND Flash 接口电路与程序设计

9.5.1　K9F2G080U 和 S5PC100 的接口电路

如图 9-11 所示为一片 K9F2G080U 和 S5PC100 的接口电路。

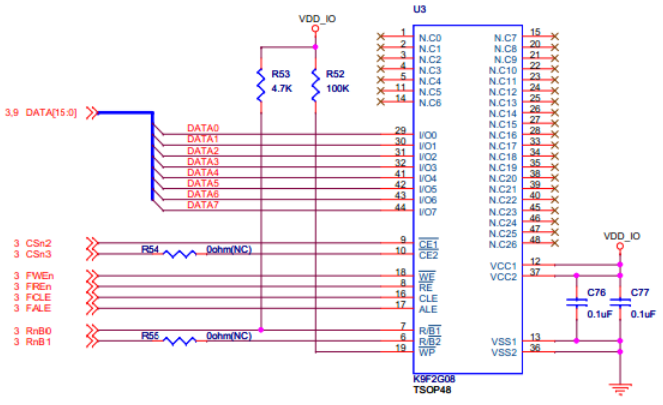

图 9-11　K9F2G080U 和 S5PC100 的接口电路

9.5.2　S5PC100 控制 K9F2G080U 的程序设计

下面举出核心的操作函数。

（1）关键的宏定义。

```
 #define   rNFCONF       (*(volatile unsigned *)0xe7200000)
#define    rNFCONT       (*(volatile unsigned *)0xe7200004)
#define    rNFCMD        (*(volatile unsigned *)0xe7200008)
#define    rNFADDR       (*(volatile unsigned *)0xe720000C)
#define rNFDATA8         (*(volatile unsigned char*)0xe7200010)
#define    rNFSTAT       (*(volatile unsigned *)0xe7200028)
#define CMD_READ1             0x00
#define CMD_READ2             0x30
#define CMD_RESET             0xFF
#define CMD_ERA1              0x60
#define CMD_ERA2              0xd0
#define CMD_WR1               0x80
#define CMD_WR2               0x10
#define NF_CMD(cmd)           {rNFCMD=(cmd);}
#define NF_ADDR(addr)         {rNFADDR=(addr);}
#define NF_RDDATA8()          (rNFDATA8)
#define NF_nFCE_L()           {rNFCONT&=~(1<<1);}
#define NF_nFCE_H()           {rNFCONT|=(1<<1);}
#define NF_WAITRB()           {while(!(rNFSTAT&(1<<28)));}
#define NF_CLEAR_RB()         {rNFSTAT |= (1<<4 );}
#define NF_DETECT_RB()        {while(!(rNFSTAT&(1<<4)));}
#define NF_WAITIO0()          {while(rNFDATA8&(1));}
```

（2）初始化函数。

```
void Nand_Init(void)
{
 rNFCONF |= 0x70 ;
 rNFCONF |= 0x7700;
 rNFCONT |= 0x03 ;
}

static void Nand_Reset(void)
{
 NF_nFCE_L();                          /*片选*/
 NF_CLEAR_RB();                        /*清除 R/B 位*/
 NF_CMD(CMD_RESET);                    /*发送复位命令*/
 NF_DETECT_RB();                       /*探测 R/B 位*/
 NF_nFCE_H();                          /*取消片选*/
}
```

（3）写一个页面的函数。

```
static int nand_write_page(unsigned char *buf, unsigned long addr)
{
 unsigned char *ptr = (unsigned char *)buf;
 unsigned int i, page_num;
 NF_CLEAR_RB();
 NF_CMD(0x80);
```

```
 page_num = addr >> 11; /*地址值除以 2048*/
 /* 写地址 */
 NF_ADDR(0);
 NF_ADDR(0);
 NF_ADDR(page_num& 0xff);
 NF_ADDR(page_num>> 8 & 0xff);
 NF_ADDR(page_num>> 16 & 0xff);

 for (i = 0; i < (2048); i++)
 {
      rNFDATA8 = *ptr;
      ptr++;

 }
 NF_CMD(0x10);

 NF_WAITRB();
 delay(50);
 return 0;
}
```

（4）读一个页面的函数。

```
static int nand_read_page(const int start_addr, unsigned char * const buffer)
{
 int i;
 int page = start_addr>>11;

 Nand_Reset();

 NF_nFCE_L();
 NF_CLEAR_RB();

 NF_CMD(CMD_READ1);
 NF_ADDR(0x0);
 NF_ADDR(0x0);
 NF_ADDR(page&0xff);
 NF_ADDR((page>>8)&0xff);
 NF_ADDR((page>>16)&0xff);
 NF_CMD(CMD_READ2);

 NF_DETECT_RB();

 for (i = 0; i < 2048; i++)
 {
      buffer[i] =  NF_RDDATA8();
 }

 NF_nFCE_H();
return 0;
}
```

（5）擦除一个块操作函数。

```
static int nand_erase_block(unsigned long addr)
{
 Nand_Reset();
 NF_nFCE_L() ;//使能芯片
 NF_CLEAR_RB();
 NF_CMD(CMD_ERA1);
 addr= addr>>17;
 NF_ADDR(addr&0xff);
 NF_ADDR(addr>>8&0xff);
 NF_ADDR(addr>>16&0xff);

 NF_CMD(CMD_ERA2);
 //F_DETECT_RB();
 NF_WAITRB();
 NF_CMD(0x70);
 NF_WAITIO0();
 /* chip Disable */
 NF_nFCE_H();

 return 0;
}
```

以上给出了包括读、写、擦除的函数，读者只需要根据实际情况组合使用，即可实现对 NAND Flash 的操作。

注意：

每次写之前，一定要先擦除块。

6．主程序

编写一个程序调用上文的 Flash 操作函数实现读写及擦除功能。

```
int main()
{
  unsigned long NANDADDR = 0x200000;
  unsigned char*p = (unsigned char*)RAM_BUF;
  int i;
   char *string = "hello world";
  uart0_init();
  Nand_Init();            //初始化 NAND Flash 控制器
  Nand_Reset();           //复位 NAND Flash
  printf("\nbefore the first write\n");
  nand_erase_block(NANDADDR);          //擦除 Nand Falsh 中 NANDADDR 开始的一个 block
  nand_read_page(NANDADDR,p);          //读出 NANDADDR 的一个页面数据
  for(i=0;i<12;i++)
  {
       printf(" %d",p[i]);             //打印出前 12 个字节
  }
  printf("\nwrite the 'hello world'\n");
  nand_write_page(string,NANDADDR);    // NANDADDR 地址处写入'hello world'数据
  nand_read_page(NANDADDR,p);          //再次从 NANDADDR 处读出一个页面
  printf ("\nread from nand : \n");
  for(i=0;i<12;i++)
```

```
    {
        printf("%c",p[i]);          //打印出读出的数据，验证是否和写入的'hello world'一致
    }
    nand_erase_block(NANDADDR);     // 再次擦除
    nand_read_page(NANDADDR,p);     // 再次读出擦除后的数据
    printf("\nafter the erase\n");
    for(i=0;i<12;i++)
    {
        printf(" %d",p[i]);         //打印出读出的数据
    }
    while (1);
    return 0;
}
}
```

实验过程和结果：

（1)将程序编译后产生.bin 可执行文件，然后使用 uboot 的 dnw 命令下载到 0x20008000 内存地址，使用 go 命令去执行，并观察结果。

（2）终端打印信息如图 9-12 所示。

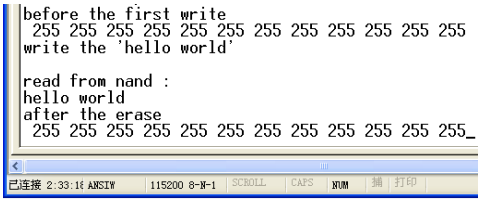

图 9-12　测试结果

9.6　本章小结

本章重点讲解了在嵌入式系统中常用的各种存储器，以及在 S5PC100 芯片中 NAND Flash、NOR Flash 的操作方法。目前 NAND Flash 在嵌入式系统中的应用非常广泛，但因为其接口比较复杂，而且需要处理坏块、坏位，所以开发有一定难度。希望读者借助本章节内容能达到入门目的后，能够结合 Bootload、操作系统继续深入研究 NAND 的坏块管理。

9.7　练习题

1．NOR Flash 和 NAND Flash 的特征及它们之间特性的对比。

2．NOR Flash 的读、写、擦除操作方法。

3．NAND Flash 的读、写、擦除操作方法。

第 10 章　定时器与 RTC

定时器／计数器简称定时器，其作用主要包括产生各种时标间隔、记录外部事件的数量等，是微机中最常用、最基本的部件之一。本章主要内容有：

- S5PC100 PWM 定时器。
- S5PC100 看门狗定时器。
- S5PC100 RTC 实时时钟。

10.1　S5PC100 PWM 定时器

10.1.1　PWM 定时器概述

在 S5PC100 中，一共有 5 个 32 位的定时器，这些定时器可发送中断信号给 ARM 子系统。另外，定时器 0、1、2 包含了脉冲宽度调制（PWM），并可驱动其拓展的 I/O。PWM 对定时器 0 有可选的 dead-zone 功能，以支持大电流设备。要注意的是定时器 3 和 4 是内置不接外部引脚的。

定时器 0 与定时器 1 共用一个 8 位预分频器，定时器 2、定时器 3 与定时器 4 共用另一个 8 位预分频器，每个定时器都有一个时钟分频器，时钟分频器有 5 种分频输出（1/2、1/4、1/8、1/16 和外部时钟 TCLK）。另外，定时器可选择时钟源，定时器 0~4 都可选择外部的时钟源，如 PWM_TCLK。

当时钟被使能后，定时器计数缓冲寄存器（TCNTBn）把计数初值下载到递减计数器中。定时器比较缓冲寄存器（TCMPBn）把其初始值下载到比较寄存器中，并将该值和递减计数器的值进行比较。这种基于 TCNTBn 和 TCMPBn 的双缓冲特性使定时器在频率和占空比变化时能产生稳定的输出。

每个定时器都有一个专用的由定时器时钟驱动的 16 位递减计数器。当递减计数器的计数值达到 0 时，就会产生定时器中断请求来通知 CPU 定时器操作完成。当定时器递减计数器达到 0 的时候，相应的 TCNTBn 的值会自动重载到递减计数器中以继续下次操作。然而，如果定时器停止了，比如在定时器运行时清除 TCON 中的定时器使能位，TCNTBn 的值不会被重载到递减计数器中。

TCMPBn 的值用于脉冲宽度调制（PWM）。当定时器的递减计数器的值和比较寄存器的值相匹配的时候，定时器控制逻辑将改变输出电平。因此，比较寄存器决定了 PWM 输出的开关时间。

10.1.2 PWM 定时器特点

PWM 定时器特点如下。

- 5 个 32 位定时器。
- 2 个 8 位 PCLK 分频器提供一级预分，5 个 2 级分频器用来预分外部时钟。
- 可编程选择 PWM 独立通道。
- 4 个独立的可编程的控制及支持校验的 PWM 通道。
- 静态配置：PWM 停止。
- 动态配置：PWM 启动。
- 支持自动重装模式及触发脉冲模式。
- 一个外部启动引脚。
- 两个 PWM 输出可带 Dead-Zone 发生器。
- 级中断发生器。

如图 10-1 所示的死区功能（Dead Zone）用于电源设备的 PWM 控制。这个功能允许在一个设备关闭和另一个设备开启之间插入一个时间间隔。这个时间间隔可以防止两个设备同时被启动。TOUT0 是定时器 0 的 PWM 输出，nTOUT0 是 TOUT0 的反转信号。如果死区功能被使能，TOUT0 和 nTOUT0 的输出波形就变成了 TOUT0_DZ 和 nTOUT0_DZ，如图 10-2 所示。

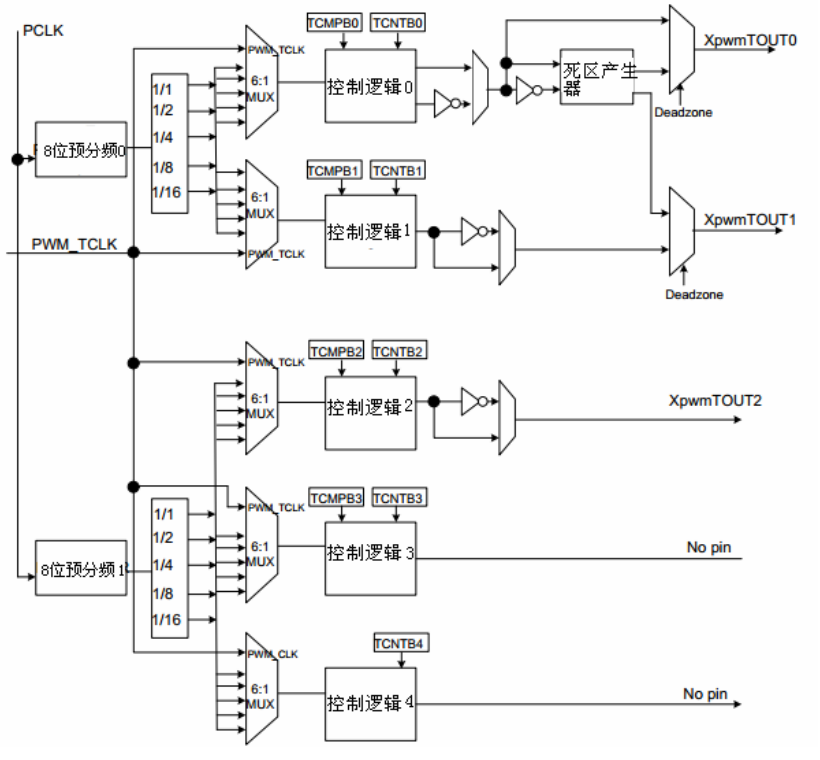

图 10-1 S5PC100 PWM 定时器

nTOUT0_DZ 在 TOUT1 脚上产生。在死区间隔内，TOUT0_DZ 和 nTOUT0_DZ 就不会同时翻转了。注意：在使能 Dead Zone 时 TOUT1 就是图 10-2 中的 nTOUT0。

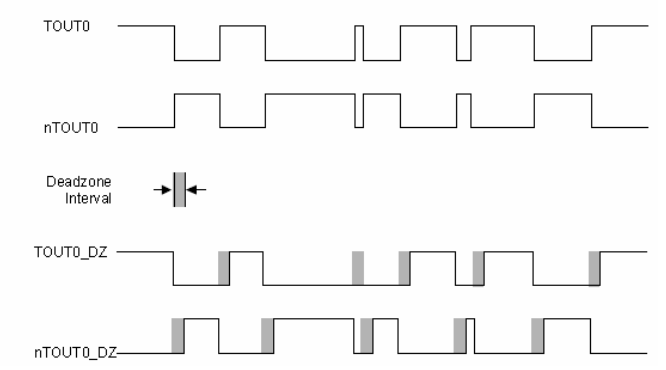

图 10-2　死区功能使能时输出的波形

10.1.3　PWM 定时器的寄存器

S5PC100 控制器共用 18 个 PWM 寄存器。

1．定时器配置寄存器 0 （TFCG0）

定时器输入时钟频率=PCLK/{prescaler value+1}/{divider value}

```
{ prescaler value }=1~255;
{ divider value }=2、4、8、16, TCLK
{Dead zone length} = 0-254
```

定时器配置寄存器 0（TCFG0）如表 10-1 所示。

表 10-1　**TCFG0** 寄存器（0xEA000000）

TCFG0	位	描述	初始状态
保留	[31:24]	保留	0x00
死区长度	[23:16]	这 8 位决定了死区的长度，一个时间单位和定时器 0 设置的相同	0x00
预分频 1	[15:8]	这 8 位定义了定时器 2、3、4 的预分频值	0x01
预分频 0	[7:0]	这 8 位定义了定时器 0 和 1 的预分频值	0x01

2．定时器配置寄存器 1（TCFG1）

定时器配置寄存器 1 主要用于 PWM 定时器的 MUX 输入。

定时器配置寄存器 1 如表 10-2 所示。

表 10-2　寄存器 TCFG1（0xEA000004）

TCFG1	位	描述	初始状态
保留	[31:24]	保留	0x0

续表

TCFG1	位	描述	初始状态
DMA 模式	[23:20]	选择 DMA 通道 0000 = 无选择 0001 = Timer0 0010 = Timer1 0011 = Timer2 0100 = Timer3 0101 = Timer4 0110 = 保留	0000
MUX 4	[19:16]	选择定时器 4 的 MUX 输入 0000 = 1/1 0001 = 1/2 0010 = 1/4 0011 = 1/8 0100=1/16 0101= PWM_TCLK	0000
MUX 3	[15:12]	选择定时器 3 的 MUX 输入 0000 = 1/1 0001 = 1/2 0010 = 1/4 0011 = 1/8 0100=1/16 0101= PWM_TCLK	0000
MUX 2	[11:8]	选择定时器 2 的 MUX 输入 0000 = 1/1 0001 = 1/2 0010 = 1/4 0011 = 1/8 0100=1/16 0101= PWM_TCLK	0000
MUX 1	[7:4]	选择定时器 1 的 MUX 输入 0000 = 1/1 0001 = 1/2 0010 = 1/4 0011 = 1/8 0100=1/16 0101= PWM_TCLK	0000
MUX 0	[3:0]	选择定时器 0 的 MUX 输入 0000 = 1/1 0001 = 1/2 0010 = 1/4 0011 = 1/8 0100=1/16 0101= PWM_TCLK	0000

3．定时器控制寄存器（TCON）

定时器控制寄存器主要用于自动重载、定时器自动更新、定时器启停、输出翻转控制等。定时器控制寄存器如表 10-3 所示。

表 10-3　寄存器 TCON（0xEA000008）

TCON	位	描述	初始状态
Timer 4 自动重装	[22]	决定 Timer 4 自动重载功能 0=单发 1=自动重载	0
Timer 4 手动更新	[21]	决定 Timer 4 的手动更新 0=无操作 1=更新 TCNTB4 寄存器	0
Timer 4 开始/停止	[20]	决定 Timer 4 的启停 0=停止 1=定时器 4 开始	0
Timer 3 自动重载	[19]	决定 Timer 3 自动重载功能 0=单发 1=自动重载	0
保留	[18]	保留	
Timer 3 手动更新	[17]	决定 Timer 3 的手动更新 0=无操作 1=更新 TCNTB3 寄存器	0
Timer 3 开/停	[16]	决定 Timer 4 的启停 0=停止 1=定时器 3 开始	0
Timer 2 自动重载	[15]	决定 Timer 2 自动重载功能 0=单发 1=自动重载	0
Timer 2 输出反相	[14]	决定 Timer 2 输出翻转 0=关闭 1=TOUT2 输出翻转	0
Timer 2 手动更新	[13]	决定 Timer 2 的手动更新 0=无操作 1=更新 TCNTB2、TCMPB2 寄存器	0

续表

TCON	位	描述	初始状态
Timer 2 开/停	[12]	决定 Timer 2 的启停 0=停止 1=定时器 2 开始	0
Timer 1 自动重载	[11]	决定 Timer 1 自动重载功能 0=单发 1=自动重载	0
Timer 1 输出反相	[10]	决定 Timer 1 输出翻转 0=关闭 1=TOUT1 输出翻转	0
Timer 1 手动更新	[9]	决定 Timer 1 的手动更新 0=无操作 1=更新 TCNTB1、TCMPB1 寄存器	0
Timer 1 start/stop	[8]	决定 Timer 1 的启停 0=停止 1=定时器 1 开始	0
保留	[7:5]	保留	
Dead zone enable	[4]	死区使能 0=不使能 1=使能	0
Timer 0 auto reload on/off	[3]	决定 Timer 0 自动重载功能 0=单发 1=自动重载	0
Timer 0 output inverter on/off	[2]	决定 Timer 0 输出翻转 0=关闭 1=TOUT0 输出翻转	0
Timer 0 manual update	[1]	决定 Timer 0 的手动更新 0=无操作 1=更新 TCNTB0、TCMPB0 寄存器	0
Timer 0 start/stop	[0]	决定 Timer0 的启停 0=停止 1=定时器 1 开始	0

4．定时器 n 计数缓冲寄存器（TCNTBn）

该寄存器用于 PWM 定时器的时间计数。定时器 n 计数缓冲寄存器如表 10-4 所示。

表 10-4　TCNTBn 寄存器

TCNTBn	位	描述	初始状态
Timer n 计数器寄存器	[15:0]	定时器 n（0~4）计数缓冲寄存器	0x00000000

5．定时器 n 比较缓冲寄存器（TCMPBn）

该寄存器用于 PWM 波形输出占空比的设置。定时器 n 比较缓冲寄存器如表 10-5 所示。

表 10-5　TCMPBn 寄存器

TCMPBn	位	描述	初始状态
Timer n 比较缓冲寄存器	[15:0]	定时器 n（0~4）比较缓冲寄存器	0x00000000

S5PC100 的 PWM 定时器具有双缓冲功能，如图 10-3 所示，能在不停止当前定时器运行的情况下，重载定时器下次运行的参数。所以尽管新的定时器的值被设置好了，但是当前操作仍能成功完成。

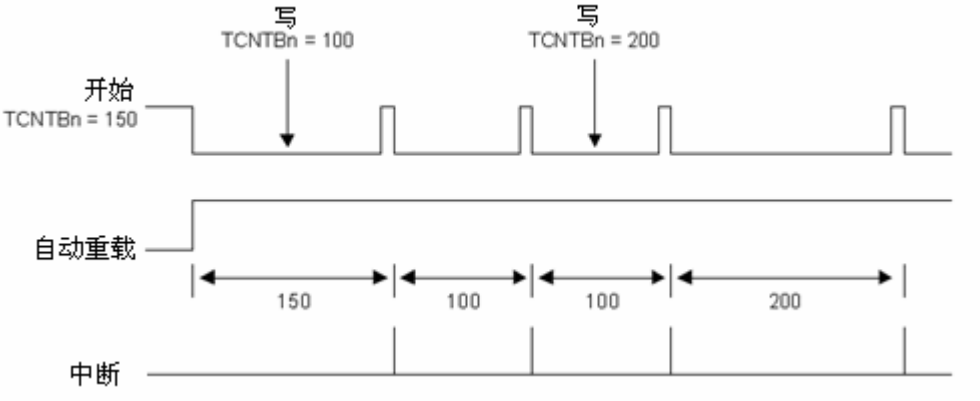

图 10-3　双缓冲功能举例

定时器值可以被写入定时器计数n缓冲寄存器（TCNTBn），当前的计数器的值可以从定时器计数观察寄存器（TCNTOn）读出。读出的TCNTBn 值并不是当前的计数值，而是下次将重载的计数值。

TCNTn 的值等于 0 的时候，自动重载操作把 TCNTBn 的值装入 TCNTn，只有当自动重载功能被使能并且 TCNTn 的值等于 0 的时候才会自动重载。如果 TCNTn 等于 0，自动重载控制位为 0，则定时器停止运行。

使用手动更新位（manual update）和反转位（Inverter）完成定时器的初始化。当递减计数器的值达到 0 时会发生定时器自动重载操作，所以 TCNTn 的初始值必须由用户提前定义好，在这种情况下就需要通过手动更新位重载初始值。以下几个步骤给出如何启动定时器：

（1）向 TCNTBn 和 TCMPBn 写入初始值。

（2）置位相应定时器的手动更新位，不管是否使用反转功能，推荐设置反转位。

（3）置位相应定时器的启动位启动定时器，清除手动更新位。

如果定时器被强制停止，TCNTn 保持原来的值而不从 TCNTBn 重载值。如果要设置一个新的值，必须执行手动更新操作。

注意：

只要 TOUT 的反转位改变，不管定时器是否处于运行状态，TOUT 都会相应改变，因此通常同时配置手动更新位和反转位。

10.1.4 PWM 定时器操作示例

操作 PWM 定时器输出如图 10-4 所示的 PWM 波形。

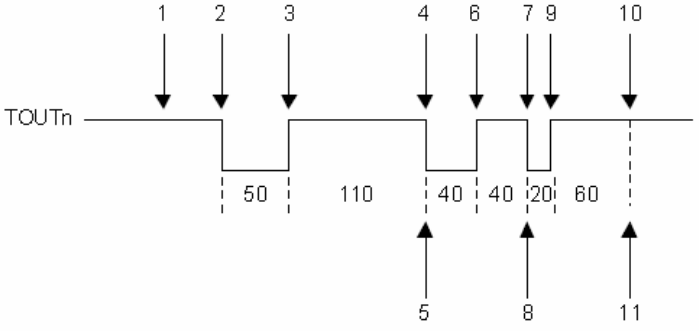

图 10-4 定时器操作实例

操作过程（过程号和图中的标号一致）如下：

（1）使能自动重载功能，设置 TCNTBn 值为 160（50＋110），TCMPBn 值为 110。置位手动更新位，配置反转位。置位手动更新位将使 TCNTBn 和 TCMPBn 的值加载到 TCNTn 和 TCMPn。然后设置 TCNTBn 和 TCMPBn 分别等于 80（40＋40）和 40。

（2）将手动更新位设为 0，将反转位设为 off，使能自动重载功能，置位启动位，则在定时器分辨率内的一段延迟后定时器开始递减计数。

（3）当 TCNTn 和 TCMPn 的值相等的时候，TOUT 输出电平由低变高。

（4）当 TCNTn 的值等于 0 的时候产生中断，并且把 TCNTBn 和 TCMPBn 的值分别自动装入 TCNTn 和 TCMPn。

（5）在中断服务程序中，将 TCNTBn 和 TCMPBn 分别设置为 80（20＋60）和 60。

（6）当 TCNTn 和 TCMPn 的值相等的时候，TOUT 输出电平由低变高。

（7）当 TCNTn 等于 0 的时候，把 TCNTBn 和 TCMPBn 的值分别自动装入 TCNTn 和 TCMPn，并触发中断。

（8）在中断服务子程序中，禁止自动重载和中断请求来停止定时器运行。

（9）当 TCNTn 和 TCMPn 的值相等的时候，TOUT 输出电平由低变高。

（10）尽管 TCNTn 等于 0，但是定时器停止运行，也不再发生自动重载操作，因为定时器自动重载功能被禁止。

（11）不再产生新的中断。

10.2　S5PC100 看门狗定时器

10.2.1　S5PC100 看门狗定时器概述

看门狗（WatchDog）定时器和 PWM 定时功能目的不一样。它的特点是，需要不停地接受信号（一些外置看门狗芯片）或重新设置计数值（如 S5PC100 的看门狗控制器），保持计数值不为 0。一旦一段时间接收不到信号，或计数值到 0，看门狗将发出复位信号复位系统或产生中断。

看门狗的作用是微控制器受到干扰进入错误状态后，使系统在一定时间间隔内复位。因此看门狗是保证系统长期、可靠和稳定运行的有效措施。目前大部分的嵌入式芯片内都集成了看门狗定时器来提高系统运行的可靠性。

S5PC100 处理器的看门狗是当系统被故障（如噪声或者系统错误）干扰时，用于微处理器的复位操作，也可以作为一个通用的 16 位定时器来请求中断操作。看门狗定时器产生 128 个 PCLK 周期的复位信号。主要特性如下：

- 通用的中断方式的 16 位定时器。
- 当计数器减到 0（发生溢出）时，产生 128 个 PLK 周期的复位信号。

看门狗定时器的功能框图如图 10-5 所示。

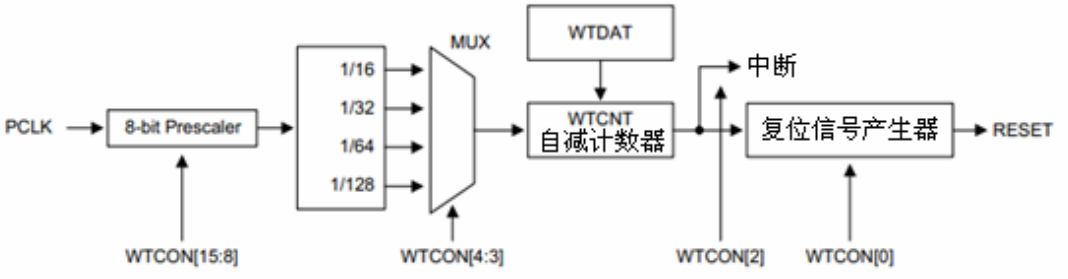

图 10-5　S5PC100 的看门狗的功能框图

看门狗模块包括一个预比例因子放大器，一个四分频的分频器，一个 16 位计数器。看门狗的时钟信号源来自 PCLK，为了得到宽范围的看门狗信号，PCLK 先被预分频，然后再经过分频器分频。预分频比例因子和分频器的分频值，都可以由看门狗控制寄存器（WTCON）决定，预分频比例因子的范围是 0～255，分频器的分频比可以是 16、32、64 或者 128。看门狗定时器时钟周期的计算如下：

t_watchdog = 1/(PCLK/(Prescaler value + 1)/Division_factor)

式中 Prescaler value 为预分频比例放大器的值；Division_factor 是四分频的分频比，可以是 16、32、64 或者 128。

一旦看门狗定时器被允许，看门狗定时器数据寄存器（WTDAT）的值就不能被自动地装载到看门狗计数器（WTCNT）中。因此，看门狗启动前要将一个初始值写入看门狗计数器（WTCNT）中。当 S5PC100 用嵌入式 ICE 调试的时候，看门狗定时器的复位功能不被启动，看门狗定时器能从 CPU 内核信号判断出当前 CPU 是否处于调试状态。如果看门狗定时器确定当前模式是调试模式，尽管看门狗能产生溢出信号，但是仍然不会产生复位信号。

10.2.2 看门狗定时器寄存器

1．看门狗定时器控制寄存器（WTCON）

WTCON 寄存器的内容包括：用户是否启用看门狗定时器、4 个分频比的选择、是否允许中断产生、是否允许复位操作等。

如果用户想把看门狗定时器当做一般的定时器使用，应该中断使能，禁止看门狗定时器复位。WTCON 描述如表 10-6 所示。

表 10-6 WTCON 描述

WTCON	位	描述	复位值
保留	[31:16]	保留	0
预分频值	[15:8]	预分频值： 有效数值范围位<0 to 255>	0x80
保留	[7:6]	保留	00
看门狗定时器	[5]	看门狗时钟使能位： 0 = 禁止 1 = 使能	1
始终选择	[4:3]	时钟分频值： 00 = 16 01 = 32 10 = 64 11 = 128	00
中断产生器	[2]	使能/屏蔽中断功能 0 = 禁止 1 = 使能	0
保留	[1]	保留	0
复位使能/屏蔽	[0]	1 =打开 S5PC100 看门狗产生复位信号 0 =禁止上述功能	1

2．看门狗定时器数据寄存器（WTDAT）

WTDAT 用于指定超时时间，在初始化看门狗操作后看门狗数据寄存器的值不能被自动装载到看门狗计数寄存器（WTCNT）中。然而，如果初始值为 0x8000，则可以自动装载 WTDAT 的值到 WTCNT 中。WTDAT 描述如表 10-7 所示。

表 10-7　WTDAT 描述

WTDAT	位	描述	复位值
保留	[31:16]	保留	0
计数重载值	[15:0]	看门狗重载数值寄存器	0x8000

3．看门狗计数寄存器（WTCNT）

WTCNT 包含看门狗定时器工作的时候计数器的当前计数值。注意在初始化看门狗操作后，看门狗数据寄存器的值不能被自动装载到看门狗计数寄存器（WTCNT）中，所以看门狗被允许之前应该初始化看门狗计数寄存器的值。WTCNT 描述如表 10-8 所示。

表 10-8　WTCNT 描述

WTCNT	位	描述	复位值
保留	[31:16]	保留	0
计数值	[15:0]	看门狗当前计数寄存器	0x8000

10.2.3　看门狗定时器程序编写

1．看门软件程序设计流程

由于看门狗是对系统的复位或者中断的操作，所以不需要外围的硬件电路。要实现看门狗的功能，只需要对看门狗的寄存器组进行操作，即对看门狗的控制寄存器（WTCON）、看门狗数据寄存器（WTDAT）、看门狗计数寄存器（WTCNT）进行操作。

其一般流程如下：

（1）设置看门狗中断操作，包括全局中断和看门狗中断的使能及看门狗中断向量的定义。如果只是进行复位操作，这一步可以不用设置。

（2）对看门狗控制寄存器（WTCON）进行设置，包括设置预分频比例因子、分频器的分频值、中断使能和复位使能等。

（3）对看门狗数据寄存器（WTDAT）和看门狗计数寄存器（WTCNT）进行设置。

（4）启动看门狗定时器。

2．看门狗超时复位测试代码

看门狗超时复位测试代码如下：

```
#include <s5pc100.h>

#define WTCON 0xEA200000
```

```
#define WTDAT (WTCON + 4)
#define WTCNT (WTCON + 8)
#define WTCLRINT (WTCON + 0xC)

#define writel(val,addr) ({*((volatile unsigned long *)addr)= val;})
#define readl(addr) ((*(volatile unsigned long *)addr))

int main()
{

   unsigned  int cfg = 0;
   cfg = (66 << 8) | (2<<3) | (1<<2);
   writel(cfg,WTCON);
   cfg = 0xffff;
   writel(cfg,WTDAT);
   writel(cfg,WTCNT);

   cfg = readl(WTCON) | (1<<5) | 1<<0;
   writel(cfg,WTCON);
   printf("WDT test running\r\n");
   while(1);
return 0;
}
```

下载程序到开发板，使用 uboot 的 go 命令启动测试程序，可以看到如图 10-6 所示的效果。

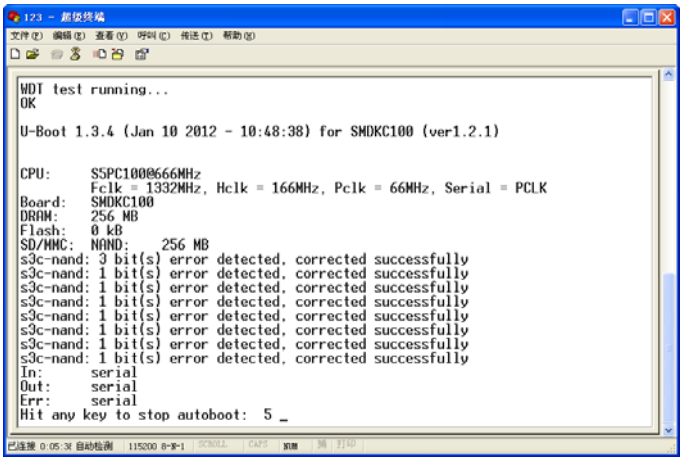

图 10-6　看门狗超时复位

在程序运行时，可以看到终端打印调试语句“WDT test running”后就复位了，复位运行的是 UBOOT 代码，所以可以看到超时后 UBOOT 重启。

3 .看门狗定时喂狗例子

```
#include <s5pc100.h>
#define WTCON 0xEA200000
#define WTDAT (WTCON + 4)
#define WTCNT (WTCON + 8)
```

```
#define WTCLRINT (WTCON + 0xC)
#define writel(val,addr) ({*((volatile unsigned long *)addr)= val;})
#define readl(addr) ((*(volatile unsigned long *)addr))

void delay(unsigned int s)
{
 volatile unsigned int s_  = s;
      while(s_--);
}
int main()
{
  unsigned  int cfg = 0;
  cfg = (66 << 8) | (2<<3) | (1<<2);
  writel(cfg,WTCON);
  cfg = 0xffff;
  writel(cfg,WTDAT);
  writel(cfg,WTCNT);

  cfg = readl(WTCON) | (1<<5) | 1<<0;
  writel(cfg,WTCON);
  printf("WDT test running...\r\n");
  while(1)
  {
     delay(100000);
     writel(readl(WTDAT),WTCNT);
  };
return 0;
}
```

下载程序并执行，可看到终端打印的信息如图 10-7 所示。看门狗因为一直在定时重新填入计数值，即喂狗，看门狗不会超时，故可看到 CPU 不会复位。

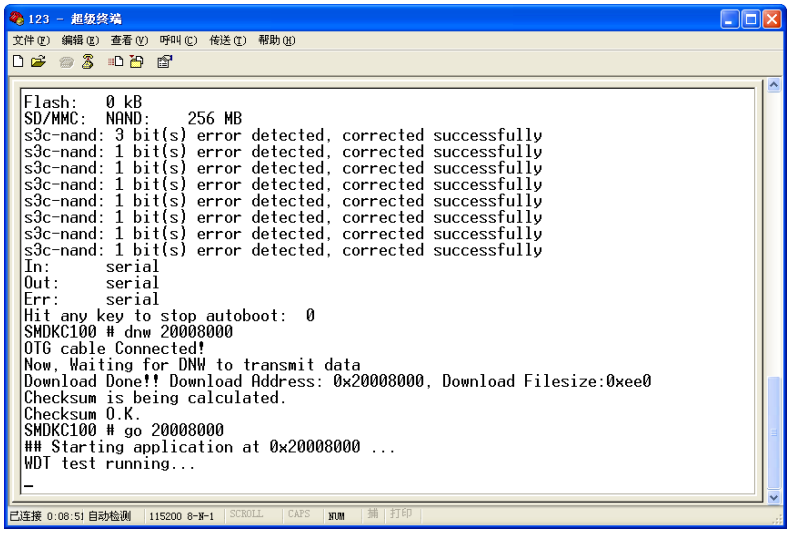

图 10-7　看门狗进行喂狗的例子

10.3 RTC

10.3.1 RTC 介绍

在一个嵌入式系统中，通常采用 RTC 来提供可靠的系统时间，包括时分秒和年月日等，而且要求在系统处于关机状态下它也能够正常工作（通常采用后备电池供电）。它的外围也不需要太多的辅助电路，典型的就是只需要一个高精度的 32.768kHz 晶体和电阻电容等，如图 10-8 所示。

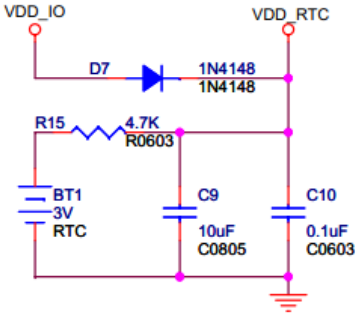

图 10-8 RTC 外接电路

10.3.2 RTC 控制器

实时时钟（RTC）单元可以通过备用电池供电，因此，即使系统电源关闭，它也可以继续工作。RTC 可以通过 STRB/LDRB 指令将 8 位 BCD 码数据送至 CPU。这些 BCD 数据包括秒、分、时、日期、星期、月和年。RTC 单元通过一个外部的 32.768kHz 晶振提供时钟。RTC 具有定时报警的功能，如图 10-9 所示。RTC 控制器功能说明：

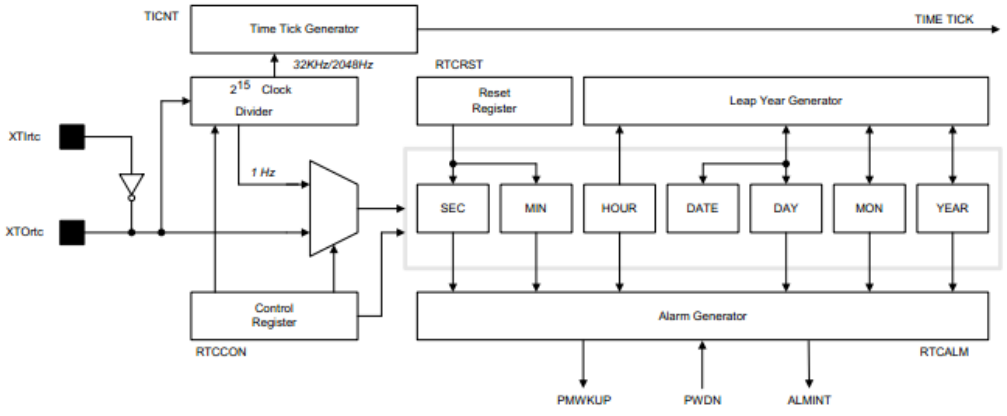

图 10-9 RTC 控制器

- 时钟数据采用 BCD 编码。
- 能够对闰年的年月日进行自动处理。
- 具有告警功能，当系统处于关机状态时，能产生告警中断。
- 具有独立的电源输入。
- 提供毫秒级时钟中断，该中断可用于作为嵌入式操作系统的内核时钟。

10.3.3　RTC 控制器寄存器详解

如表 10-9 所示为相关寄存器描述。

表 10-9　RTC 控制寄存器

RTCCON	位	描述	复位值
保留	[31:9]	保留	0
TICEN	[8]	嘀嗒计时器 0 = 禁止 1 = 使能	0
TICCKSEL	[7:4]	嘀嗒计时器子时钟源选择 4’b0000 = 32768 Hz　4’b0001 = 16384 Hz 4’b0010 = 8192 Hz　4’b0011 = 4096 Hz 4’b0100 = 2048 Hz　4’b0101 =1024 Hz 4’b0110 =512 Hz　4’b0111 =256 Hz 4’b1000 =128 Hz　4’b1001 =64 Hz 4’b1010 =32 Hz　4’b1011 =16 Hz 4’b1100 =8 Hz　4’b1101 =4 Hz 4’b1110 =2 Hz　4’b1111 =1 Hz	4’b0000
CLKRST	[3]	RTC 时钟计数复位 0 = 不复位 1 = 复位	0
CNTSEL	[2]	BCD 计数选择 0 = 分配 BCD 计数 1 = 保留	0
CLKSEL	[1]	BCD 时钟选择 0 = XTAL 1/2 divided clock 1 = 保留（XTAL 供频）	0
RTCEN	[0]	RTC 控制使能 0 = 禁止 1 = 使能	0

如表 10-10 所示为 BCD 值寄存器描述。

表 10-10　BCD 值寄存器

BCDSEC	位	描述	复位值
保留	[31:7]	保留	-
SECDATA	[6:4]	BCD 值 0~5	-
	[3:0]	0~9	-

10.3.4　RTC 测试例子

下面的代码实现了一个将 RTC 的年月日、时分秒读出的功能，可以将注释掉的代码打开来复位值。

```
#define BASE 0xEA300000
#define read(addr) (*(volatile unsigned long*)(addr + BASE))
#define write(addr,val) ((*(volatile unsigned long*)(addr+BASE)) = (val))
int main()
{
 unsigned long cfg = 0;
 cfg |= 0x1;
 write(0x40,cfg);
 int i,j,k;;
 /*清理原先的寄存器内容*/
//    for(i=0;i<4*7;i+=4)
//    write(0x70+i,0);

 printf("ok\n");
      while(1)
 {
      /*打印时分秒、年月日*/
      printf("sec %x ",read(0x70));
      printf("min %x ",read(0x74));
      printf("hor %x \n",read(0x78));
      printf("date %x \n",read(0x7c));
printf("day %x \n",read(0x80));
printf("mon %x \n",read(0x84));
printf("year %x \n",read(0x88));
      /*等待*/
      for(i=0;i<100000;i++)
        for(j=0;j<100000;j++);
 }
}
```

实验过程及结果描述：

（1）程序编译后将产生.bin 可执行文件，然后使用 uboot 的 dnw 命令下载到 0x20008000 内存地址，使用 go 命令去执行，并观察结果。

（2）终端打印信息如图 10-10 所示。

```
hour 12 : min 59 : sec 17
```

图 10-10　打印信息

10.4　本章小结

本章重点讲解了 PWM 和看门狗控制器的工作原理，以及 S5PC100 芯片中 PWM 控制器和看门狗控制器的操作方法，最后还介绍了 RTC 的编程方法。

10.5　练习题

1．PWM 输出波形的特点是什么？
2．在控制系统中为何要加入看门狗功能？
3．编程实现输出占空比为 2∶1、波形周期为 9ms 的 PWM 波形。
4．编程实现 1s 内不对看门狗实现喂狗操作，看门狗会自动复位。
5．编程实现 RTC 的定时功能。

第 11 章　A/D 转换器

A/D 转换又称模数转换，顾名思义，就是把模拟信号数字化。实现该功能的电子器件称为 A/D 转换器，A/D 转换器可将输入的模拟电压转换为与其成比例输出的数字信号。随着数字技术，特别是计算机技术的飞速发展与普及，在现代控制、通信及检测领域中，对信号的处理广泛采用了数字计算机技术。由于系统的实际处理对象往往都是一些模拟量（如温度、压力、位移、图像等），要使计算机或数字仪表能识别和处理这些信号，必须首先将这些模拟信号转换成数字信号，这就必须用到 A/D 转换器。

本章主要内容：

- A/D 转换器原理。
- S5PC100 A/D 转换器。
- A/D 转换器应用举例。

11.1　A/D 转换器原理

11.1.1　A/D 转换基础

在基于 ARM 的嵌入式系统设计中，A/D 转换接口电路是应用系统前向通道的一个重要环节，可完成一个或多个模拟信号到数字信号的转换。模拟信号到数字信号的转换一般来说并不是最终的目的，转换得到的数字量通常要经过微控制器的进一步处理。A/D 转换的一般步骤如图 11-1 所示。

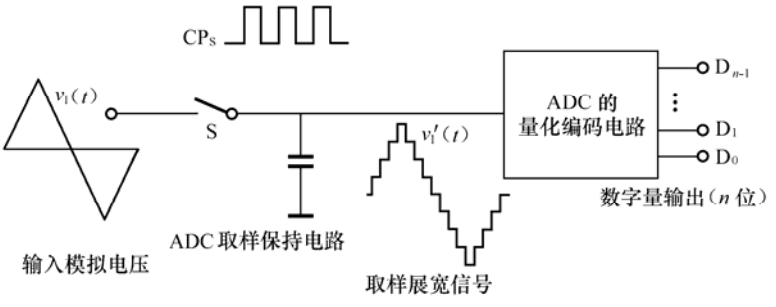

图 11-1　A/D 转换的一般步骤

11.1.2　A/D 转换的技术指标

1．分辨率（Resolution）

数字量变化一个最小量时模拟信号的变化量，定义为满刻度与 2*n* 的比值。分辨率又称精度，通常以数字信号的位数来表示。A/D 转换器的分辨率以输出二进制（或十进制）数的位数表示。从理论上讲，*n* 位输出的 A/D 转换器能区分 2*n* 个不同等级的输入模拟电压，能区分输入电压的最小值为满量程输入的 1/2*n*。在最大输入电压一定时，输出位数愈多，量化单位愈小，分辨率愈高。例如 S5PC100 的 A/D 转换器可以设置输出为 10 位二进制数，输入信号最大值为 3.3V，那么这个转换器应能区分输入信号的最小电压为 3.22mV。

2．转换速率（Conversion Rate）

完成一次从模拟转换到数字的 A/D 转换所需的时间的倒数。积分型 A/D 的转换时间是毫秒级，属低速 A/D；逐次比较型 A/D 是微秒级，属中速 A/D；全并行/串并行型 A/D 可达到纳秒级。采样时间则是另外一个概念，是指 2 次转换的间隔。为了保证转换的正确完成，采样速率（Sample Rate）必须小于或等于转换速率。因此有人习惯上将转换速率在数值上等同于采样速率也是可以接受的。常用单位是 ksps 和 Msps，表示每秒采样千/百万次（kilo / Million Samples per Second）。

3．量化误差（Quantizing Error）

由于 A/D 的有限分辨率而引起的误差，即有限分辨率 A/D 的阶梯状转移特性曲线与无限分辨率 A/D（理想 A/D）的转移特性曲线（直线）之间的最大偏差。通常是 1 个或半个最小数字量的模拟变化量，表示为 1LSB、1/2LSB。量化和量化误差示意图如图 11-2 所示。

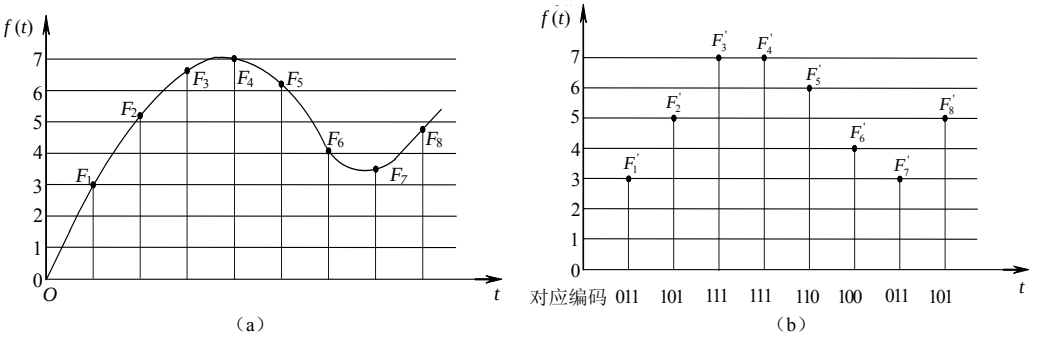

图 11-2　量化与量化误差

4．偏移误差（Offset Error）

输入信号为零时输出信号不为零的值，可外接电位器调至最小。

5．满度误差（Full Scale Error）

满度输出时对应的输入信号与理想输入信号值之差。

6．线性度（Linearity）

实际转换器的转移函数与理想直线的最大偏移，不包括以上 3 种误差。

其他指标还有绝对精度（Absolute Accuracy）、相对精度（Relative Accuracy）、微分非线性、单调性和无错码、总谐波失真（Total Harmonic Distortion，THD）和积分非线性。

11.1.3 A/D 转换器类型

下面简要介绍常用的几种类型的 A/D 转换器的基本原理及特点：积分型、逐次逼近型、并行比较型/串并行型、Σ-Δ 调制型、电容阵列逐次比较型及压频变换型。

1．积分型 A/D 转换器

积分型 A/D 转换器工作原理是将输入电压转换成时间（脉冲宽度信号）或频率（脉冲频率），然后由定时器/计数器获得数字值。积分型 A/D 实际上是 V-T 方式电压对时间的转换，先对输入量化电压以固定时间正向积分，然后再对基准电压反向积分，计数就是对应的 A/D 结果值。

双积分型 A/D 转换是一种间接 A/D 转换技术。首先将模拟电压转换成积分时间，然后用数字脉冲计时方法转换成计数脉冲数，最后将此代表模拟输入电压大小的脉冲数转换成二进制或 BCD 码输出。因此，双积分型 A/D 转换器转换时间较长，一般要大于 40～50ms。其优点是用简单电路就能获得高分辨率，但缺点是由于转换精度依赖于积分时间，因此转换速率极低。初期的单片 A/D 转换器大多采用积分型，现在逐次比较型已逐步成为主流。如图 11-3 所示为双积分型 A/D 的控制逻辑。积分器是转换器的核心部分，它的输入端所接开关 S1 由定时信号控制。当定时信号为不同电平时，极性相反的输入电压 u_i 和参考电压 V_{REF} 将分别加到积分器的输入端，进行两次方向相反的积分，积分时间常数 τ =RC。

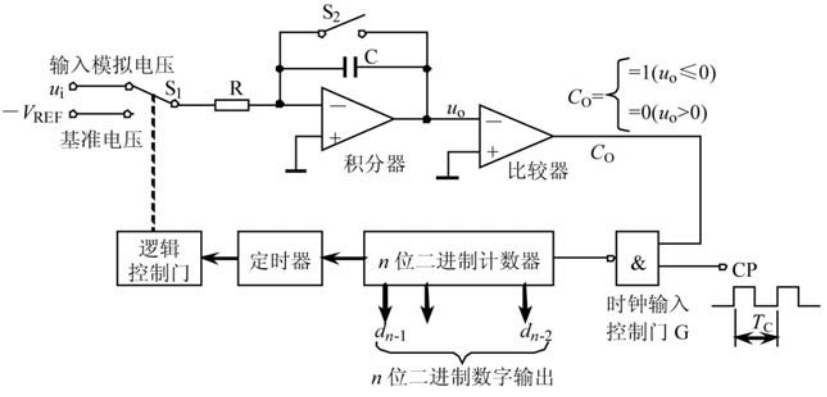

图 11-3　双积分型 A/D 控制逻辑图

过零比较器用来确定积分器的输出电压 u_o 过零的时刻。当 $u_o \geq 0$ 时，比较器输出电压为低电平；当 $u_o < 0$ 时，比较器输出电压为高电平。比较器的输出信号接至时钟控制门（G）作为关门和开门信号。

双积分型 A/D 转换器具有很强的抗干扰能力，故而采用双积分型 A/D 转换器可大大降低对滤波电路的要求。

2．逐次逼近型 A/D

逐次逼近型 A/D 由逐次寄存器、比较器、同精度的 D/A、基准电压组成。从 MSB 开始，顺序地对每一位将输入电压与内置 DA 转换器输出进行比较，经 n 次比较而输出数字值。其电路规模属于中等。其优点是速度较高、功耗低，在低分辨率（<12 位）时价格便宜，但高精度（>12 位）时价格很高。

4 位逐次比较型 A/D 转换器的逻辑电路如图 11-4 所示。

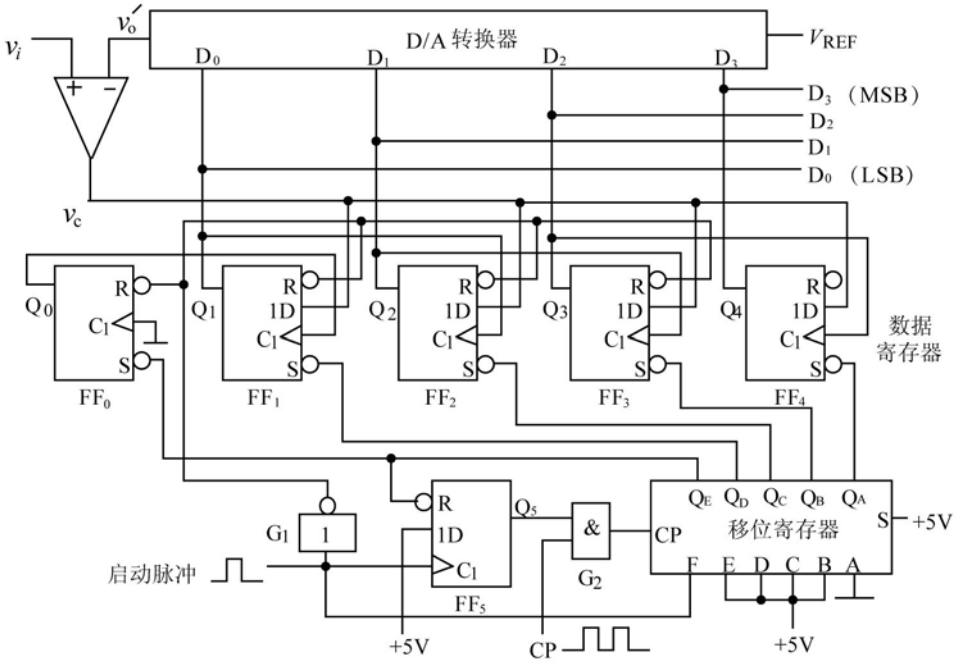

图 11-4　逐次逼近型 A/D 原理图

图中 5 位移位寄存器可进行并入/并出或串入/串出操作，其输入端 F 为并行置数使能端，高电平有效。其输入端 S 为高位串行数据输入。数据寄存器由 D 边沿触发器组成，数字量从 Q_4～Q_1 输出。

电路工作过程如下，当启动脉冲上升沿到达后，FF_0～FF_4 被清零，Q_5 置 1，Q_5 的高电平开启与门 G_2，时钟脉冲 CP 进入移位寄存器。在第 1 个 CP 脉冲作用下，由于移位寄存器的置数使能端 F 以由 0 变 1，并行输入数据 ABCDE 置入，$Q_AQ_BQ_CQ_DQ_E$=01111，Q_A 的低电平使数据寄存器的最高位（Q_4）置 1，即 $Q_4Q_3Q_2Q_1$=1000。D/A 转换器将数字量 1000 转换为模拟电压 ，送入比较器 C 与输入模拟电压 v_i 比较，若 v_i>v'_o，则比较器 C 输出 v_c 为 1，否则为 0。比较结果送 D_4～D_1。

第 2 个 CP 脉冲到来后，移位寄存器的串行输入端 S 为高电平，Q_A 由 0 变 1，同时最高位 Q_A 的 0 移至次高位 Q_B。于是数据寄存器的 Q_3 由 0 变 1，这个正跳变作为有效触发信号加到 FF_4 的 CP 端，使 v_c 的电平得以在 Q_4 保存下来。此时，由于其他触发器无正跳变触发脉冲，v_c 的信号对它们不起作用。Q_3 变 1 后，建立了新的 D/A 转换器的数据，输入电压再与其输出电压 进行比较，比较结果在第 3 个时钟脉冲作用下存于 Q_3……如此进行，直到 Q_E 由 1 变 0 时，使触发器 FF_0 的输出端 Q_0 产生由 0 到 1 的正跳变，做触发器 FF_1 的 CP 脉冲，使上一次 A/D 转换后的 v_c 电平保存于 Q_1。同时使 Q_5 由 1 变 0 后将 G_2 封锁，一次 A/D 转换过程结束。于是电路的输出端 $D_3D_2D_1D_0$ 得到与输入电压 v_i 成正比的数字量。

逐次逼近转换过程和用天平称物重非常相似。天平称重物过程是，从最重的砝码开始试放，与被称物体进行比较，若物体重于砝码，则该砝码保留，否则移去。再加上第二个次重砝码，由物体的重量是否大于砝码的重量决定第二个砝码是留下还是移去。如此一直加到最小一个砝码为止。将所有留下的砝码重量相加，就得此物体的重量。仿照这一思路，逐次比较型 A/D 转换器，就是将输入模拟信号与不同的参考电压做多次比较，使转换所得的数字量在数值上逐次逼近输入模拟量对应值。

3．并行比较/串行比较型 A/D

3 位并行比较型 A/D 转换原理电路如图 11-5 所示，它由电压比较器、寄存器和代码转换器 3 部分组成。

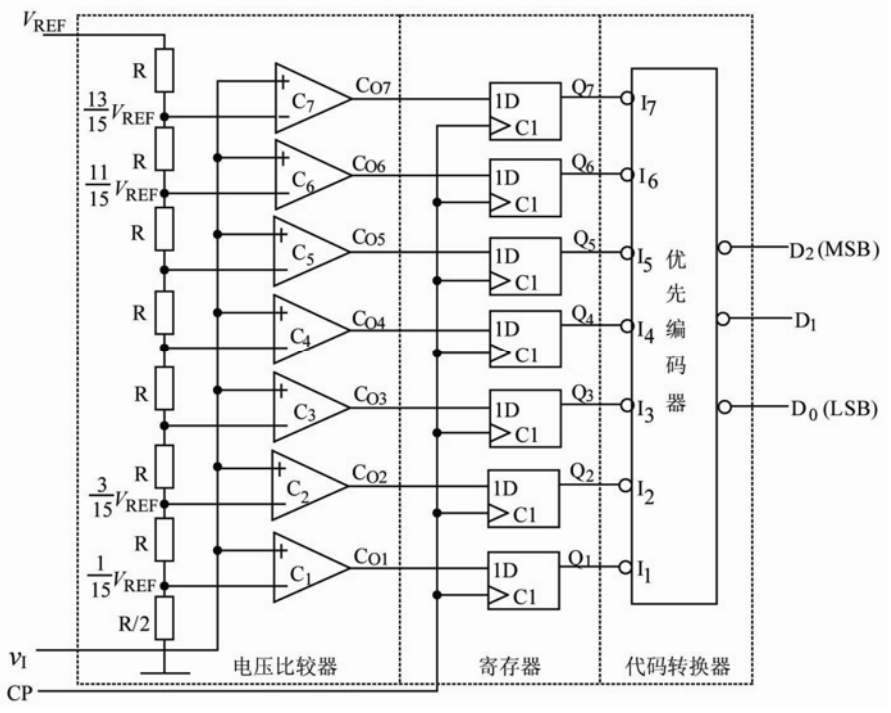

图 11-5　并行比较型 A/D

首先在电压比较器中进行量化电平的划分，用电阻链把参考电压 V_{REF} 分压，得到从 $\frac{1}{15}$ V_{REF}～$\frac{13}{15}V_{REF}$ 之间 7 个比较电平，量化单位 Δ= 。然后，把这 7 个比较电平分别接到 7 个比较器 C_1～C_7 的输入端作为比较基准。同时将输入的模拟电压同时加到每个比较器的另一个输入端上，与这 7 个比较基准进行比较。

并行 A/D 转换器具有如下特点。

（1）由于转换是并行的，其转换时间只受比较器、触发器和编码电路延迟时间限制，因此转换速度最快。

（2）随着分辨率的提高，元件数目要按几何级数增加。一个 n 位转换器，所用的比较器个数为 $2n-1$，如 8 位的并行 A/D 转换器就需要 28−1=255 个比较器。由于位数愈多，电路愈复杂，因此制成分辨率较高的集成并行 A/D 转换器是比较困难的。

（3）使用这种含有寄存器的并行 A/D 转换电路时，可以不用附加取样－保持电路，因为比较器和寄存器这两部分也兼有取样－保持功能。这也是该电路的一个优点。

图 11-5 中的 8 个电阻将参考电压 V_{REF} 分成 8 个等级，其中 7 个等级的电压分别作为 7 个比较器 C_1～C_7 的参考电压，其数值分别为 $V_{REF}/15$、$3V_{REF}/15$…、$13V_{REF}/15$。输入电压为 v_I，它的大小决定各比较器的输出状态，如当 $0 \leqslant v_I < V_{REF}/15$ 时，C_7～C_1 的输出状态都为 0；当 $3V_{REF}/15 \leqslant v_I < 5V_{REF}/15$ 时，比较器 C_6 和 C_7 的输出 $CO_6=CO_7=1$，其余各比较器的状态均为 0。根据各比较器的参考电压值，可以确定输入模拟电压值与各比较器输出状态的关系。比较器的输出状态由 D 触发器存储，经优先编码器编码，得到数字量输出。优先编码器优先级别最高是 I_7，最低的是 I_1。

设 v_I 变化范围是 0～V_{REF}，输出 3 位数字量为 $D_2D_1D_0$，3 位并行比较型 A/D 转换器的输入、输出关系如表 11-1 所示。

表 11-1　3 位并行 A/D 转换器输入与输出关系对照表

模拟输入	比较器输出状态							数字输出	
	C_{O1}	C_{O2}	C_{O3}	C_{O4}	C_{O5}	C_{O6}	C_{O7}	D_2	D_1
$0 \leqslant v_I < V_{REF}/15$	0	0	0	0	0	0	0	0	0
$V_{REF}/15 \leqslant v_I < 3V_{REF}/15$	0	0	0	0	0	0	1	0	0
$3V_{REF} \leqslant v_I < 5V_{REF}/15$	0	0	0	0	0	1	1	0	1
$5V_{REF} \leqslant v_I < 7V_{REF}$	0	0	0	0	1	1	1	0	1
$7V_{REF}/15 \leqslant v_I < 9V_{REF}/15$	0	0	0	1	1	1	1	1	0
$9V_{REF}/15 \leqslant v_I < 11V_{REF}/15$	0	0	1	1	1	1	1	1	0
$11V_{REF}/1 \leqslant v_I < 13V_{REF}/15$	0	1	1	1	1	1	1	1	1
$13V_{REF} \leqslant v_I < V_{REF}$	1	1	1	1	1	1	1	1	1

由于转换是并行的，其转换时间只受比较器、触发器和编码电路延迟时间的限制，因此转换速度最快。随着分辨率的提高，元件数目要按几何级数增加。一个 n 位转换器，所用比较器的个数为 $2n-1$，如 8 位的并行 A/D 转换器就需要 255 个比较器。由于位数愈多，

电路愈复杂，因此制成分辨率较高的集成并行 A/D 转换器是比较困难的。精度取决于分压网络和比较电路。动态范围取决于 V_{REF}。

4．电容阵列逐次比较型

电容阵列逐次比较型 A/D 在内置 D/A 转换器中采用电容矩阵方式，也可称为电荷再分配型。一般的电阻阵列 D/A 转换器中多数电阻的值必须一致。在单芯片上生成高精度的电阻并不容易，如果用电容阵列取代电阻阵列，可以用低廉的成本制成高精度的单片 A/D 转换器。最近的逐次比较型 A/D 转换器大多为电容阵列式的。

5．压频变换型

压频变换型（Voltage-Frequency Converter）是通过间接转换方式实现模数转换的。其原理是首先将输入的模拟信号转换成频率，然后用计数器将频率转换成数字量。从理论上讲这种 A/D 的分辨率几乎可以无限增加，只要采样的时间能够满足输出频率分辨率要求的累积脉冲个数的宽度。其优点是分辨率高、功耗低、价格低，但是需要外部计数电路共同完成 A/D 转换。

11.1.4 A/D 转换的一般步骤

模拟信号进行 A/D 转换的时候，从启动转换到转换结束输出数字量，需要一定的转换时间，在这个转换时间内，模拟信号要基本保持不变。否则转换精度没有保证，特别是当输入信号频率较高时，会造成很大的转换误差。要防止这种误差的产生，必须在 A/D 转换开始时将输入信号的电平保持住，而在 A/D 转换结束后，又能跟踪输入信号的变化。因此，一般的 A/D 转换过程是通过取样、保持、量化和编码这 4 个步骤完成的。一般取样和保持主要由采样保持器来完成，而量化编码就由 A/D 转换器完成。

11.2 S5PC100 A/D 转换器

11.2.1 S5PC100 A/D 转换器概述

1．简述

10 位或 12 位 CMOS 再循环式模拟数字转换器，它具有 10 通道输入，并可将模拟量转换为 10 位或 12 位二进制数。5MHz A/D 转换时钟时，最大 1MSPS。A/D 转换操作具有样本保持的功能，同时也支持降功耗模式。

2．特性

ADC 接口包括如下特性。

- 10bit/12bit 输出位可选。
- 微分误差±1.0LSB。

- 积分误差±2.0LSB。
- 最大转换速率：1 Msps。
- 功耗少，电压输入 3.3v。
- 模拟量输入范围：0～3.3v。
- 支持片上样本保持功能。
- 通用转换模式。

3．模块图

S5PC100 A/D 转换器的控制器接口框图如图 11-6 所示。

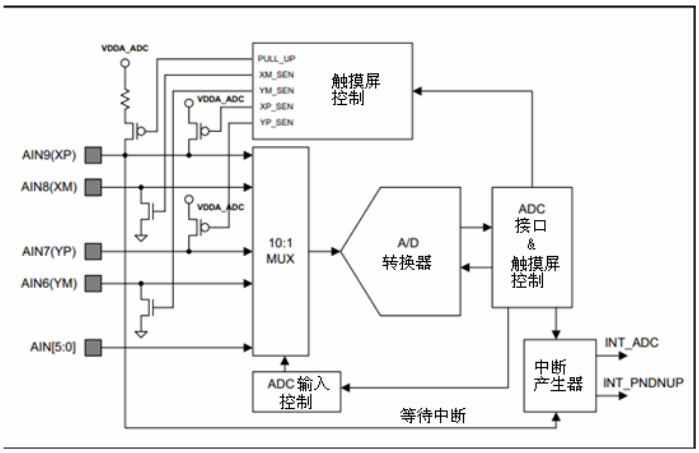

图 11-6　S5PC100 ADC 控制器接口框图

11.2.2　S5PC100 A/D 控制器寄存器

1．寄存器组

S5PC100 中的 A/D 控制器集成了电阻触摸屏控制功能。此处为了简化学习，只考虑的 A/D 转换使用到的两个寄存器，即 A/D 控制寄存器（ADCCON）、A/D 转换数据寄存器（ADCDAT）。

A/D 控制寄存器 ADCCON（address = 0xF3000000）如表 11-2 所示。

表 11-2　ADCCON 描述

ADCCON	位	描　　述	初　始　值
RES	[16]	0=10bit 输出 1=12bit 输出	0
ECFLG	[15]	A/D 转换结束标志 0：A/D 转换正在进行 1：A/D 转换结束	0
PRSCEN	[14]	A/D 转换预分频允许 0：不允许预分频 1：允许预分频	0

续表

ADCCON	位	描　　述	初　始　值
PRSCVL	[13:6]	预分频值 PRSCVL	0xFF
Reserved	[5:3]	保留	0
STDBM	[2]	待机模式选择位 0：正常模式 1：待机模式	1
READ_START	[1]	A/D 转换读－启动选择位 0：禁止 Start-by-read 1：允许 Start-by-read	0
ENABLE_START	[0]	A/D 转换器启动 0：A/D 转换器不工作 1：A/D 转换器开始工作	0

A/D 转换数据寄存器 ADCDAT0（地址 0xF300000C）如表 11-3 所示。

表 11-3　ADCDAT0 描述

ADCDAT0	Bit	描　　述	初　始　值
UPDOWN	[15]	等待中断模式，Stylus 电平选择 0：低电平 1：高电平	—
AUTO_PST	[14]	自动按照先后顺序转换 X，Y 坐标 0：正常 A/D 转换顺序 1：按照先后顺序转换	—
XY_PST	[13:12]	自定义 X，Y 位置 00：无操作模式 01：测量 X 位置 10：测量 Y 位置 11：等待中断模式	—
XPDATA	[11:0]	X 坐标转换数据值（包括正常的 ADC 转换数值）	—

2．A/D 转换的转换时间计算

例如，PCLK 为 66MHz，PRESCALER = 65；所有 10 位转换时间为：

$$66\ \text{MHz} / (65 + 1) = 1\text{MHz}$$

转换时间为 1/(1M/5 cycles) = 5μs。

完成一次 A/D 转换需要 5 个时钟周期。A/D 转换器的最大工作时钟为 2.5MHz，所以最大的采样率可以达到 500kbit/s。

11.3　A/D 转换器应用举例

11.3.1　电路连接

电路连接如图 11-7 所示，利用一个电位计输出电压到 S5PC100 的 ADC_IN0 引脚。输入的电压范围是 0~3.3V。

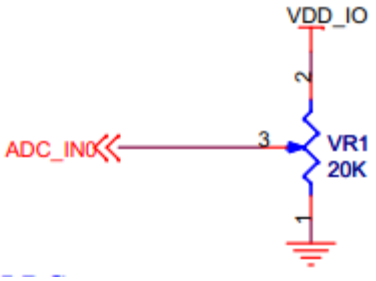

图 11-7　分压电路

11.3.2　程序编写

编写软件程序，实现电压值的获取、显示。 程序主要是对 S5PC100 中的 A/D 模块进行操作，所以软件程序也主要是对 A/D 模块中的寄存器进行操作，其中包括对 ADC 控制寄存器（ADCCON）、ADC 数据寄存器（ADCDAT）的读/写操作。同时为了观察转换结果，可以通过串口在超级终端里面观察。

```
/*该结构体定义了 ADC 控制器相关寄存器*/

typedef struct {
  unsigned int ADCCON;
  unsigned int ADCTSC;
  unsigned int ADCDLY;
  unsigned int ADCDAT0;
  unsigned int ADCDAT1;
  unsigned int ADCUPDN;
  unsigned int ADCCLRINT;
  unsigned int ADCMUX;
  unsigned int ADCPNDCLR;
}adc;
#define ADC     (* (volatile adc * )0xF3000000 ) //to define the ADC register structure

int main()
{
  unsigned  int  temp = 0;
  float ac;
  volatile int count;
```

```
    ADC.ADCMUX = 0;// 初始化 ADC mux 寄存器
    /*12 位模式，AD 使能，预分值 255，普通模式，读开启*/
  ADC.ADCCON = ( 1<<16 | 1<<14 | 0xff<<6 | 0<<2 | 1<<1 ) ;
    temp = ADC.ADCDAT0 & 0XFFF;//get the data at first time

    while(1)//循环读
    {
         while(!(ADC.ADCCON & 0X8000));
         /*读 ECFLG 位的值，如果读结束则该位置 1*/
         temp = ADC.ADCDAT0 & 0XFFF;
         temp = 3.3 * 1000 * temp/0xfff;
         printf("adc=%d mV \n",temp);
         for(count = 1000000; count != 0; count--);
    }
    return 0;
  }
```

11.3.3 调试与运行结果

1. 串口接收设置

在 PC 上运行 Windows 自带的超级终端串口通信程序（波特率为 115 200Bd、1 位停止位、无校验位、无硬件流控制）；或者使用其他串口通信程序。

2. 测试程序

（1）编译程序后可得到 bin 文件。

（2）上电后，执行 uboot。

（3）通过 dnw 下载 bin 文件到内存中。

（4）使用 go 命令执行。

3. 观察实验结果

当执行程序后，终端将打印结果如图 11-8 所示。

```
SD/MMC:  NAND:    256 MB
*** Warning - using default environment

In:      serial
Out:     serial
Err:     serial
Hit any key to stop autoboot:  0
SMDKC100 #
SMDKC100 # dnw 20008000
OTG cable Connected!
Now, Waiting for DNW to transmit data
Download Done!! Download Address: 0x20008000, Download Filesize:0xf40
Checksum is being calculated.
Checksum O.K.
SMDKC100 # go 20008000
## Starting application at 0x20008000 ...
open uart device ok !open uart device ok !adc=1721 mV
adc=1716 mV
adc=1714 mV
adc=1730 mV
adc=1712 mV
adc=1714 mV
adc=1732 mV
```

图 11-8　终端打印结果

11.4　本章小结

本章主要讲解了 A/D 转换器的工作原理，以及 S5CP100 下 A/D 控制器的操作方法。

11.5　练习题

1．A/D 转换器选型时需要考虑哪些指标？

2．根据 A/D 的基本原理，可以将 A/D 控制器分为哪些种类？

3．在 PCLK 为 50MHz 的情况，如何设置 S5PC100 的 A/D 控制器来实现采集速度为 100Ksps？

4．编程实现采集一个范围在 0~3.3V 的电压的测试程序。

第 12 章　DMA（PL330）控制器

S5PC100 所使用的 DMA 控制器不同于以往，因此作为 Cortex-A8 的一个特点，本章重点介绍一下 Cortex-A8 所使用的 DMA 控制器及一些编程方法（还有原理）。旨在介绍全新的 DMA 控制技术，并在文后贴出一些实验代码供给读者参考。

本章内容重点：

- PL330 原理简述。
- DMAC 寄存器介绍。
- DMAC 操作例子。

12.1　PL330 原理概述

12.1.1　DMAC 简述

DMA 作为一种 CPU 与外设传输数据的技术，现在广泛用于各种计算机架构中，它最大的优点就是在无须 CPU 干涉下，完成数据从内存到外设的传递。这一章就给读者讲解一下 S5PC100 中的 DMA 控制器的操作方法。

首先简单介绍一下什么是 DMAC，DMAC 是一个自适应先进的微控制器总线体系的控制器，它由 ARM 公司设计并基于 PrimeCell 技术标准，DMAC 提供了一个 AXI 接口用来执行 DMA 传输，以及两个 APB 接口用来控制这个操作，DMAC 在安全模式技术下用一个 APB 接口执行 TrustZone 技术，其他操作则在非安全模式下执行。DMAC 包括了一个小型的指令集，用来提供一些灵活便捷的操作，为了缩小内存需求，DMAC 则使用了变长指令。

不同于 ARM11 及以前系列的芯片，S5PC100 使用了基于 PrimeCell 技术标准的 PL330（DMA 控制器核心）有了很大的变化，从编程方式上看，它提供了灵活的子指令集，使得用户有更多的组合方式来操作 DMA，从硬件上看，它实现了硬件上的多线程管理，一次编写代码即可让它正常地完成所需的工作，因此这一章的学习是有一定困难的。

如图 12-1 所示为 DMAC 接口框图。

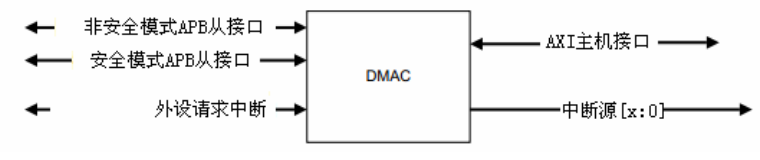

图 12-1　DMAC 接口框图

在 S5PC100 中，三星公司为安全考虑而加入了一套新的技术标准，即多加了一套安全模式，在安全模式下，处理核的寄存器是受到保护并且是与非安全模式隔离开来的，这样在一般的外设接口都会涉及两种模式，DMAC 也不例外，但是这里我们只关注非安全模式。

12.1.2　S5PC100 下的 DMAC 模型

如图 12-2 所示为 DMAC 模型。

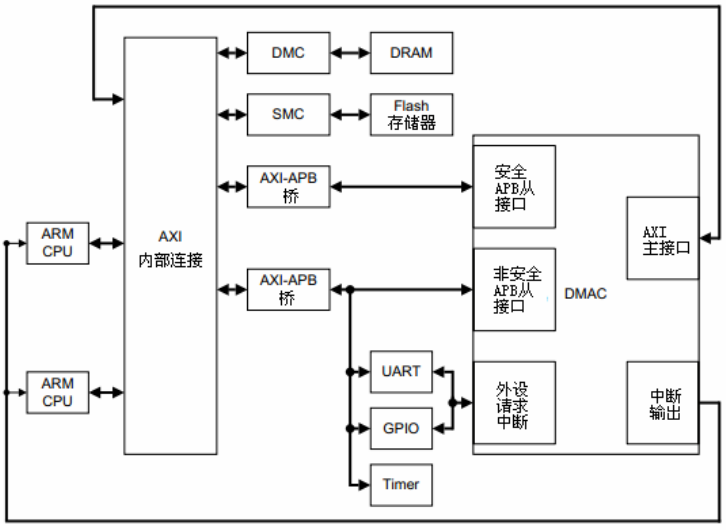

图 12-2　DMAC 模型

AXI 总线主机：DMAC 及一个 ARM 处理器、一个 AXI 互联及两个 AMBA 协议桥。

PrimeCell 的从机：

- 一个动态的内存控制器。
- 一个静态的内存控制器。
- 一个定时器。
- 一个 GPIO。
- 一个 UART。

1．特性

DMAC 提供了如下的特性：

- 一个提供灵活编程接口的 DMA 指令集。
- 单个 AXI 主机接口控制 DMA 传输。
- 双 APB 从机接口下，同时提供基于安全及非安全两套寄存器。
- 支持 TrustZone 技术。
- 支持多种传输类型。
  - 内存至内存。

- 内存至外设。
- 外设至内存。
- 分散/聚集模式。

- 可配置的 RTL，使得 DMAC 对于应用有着更佳的性能。
- 对于每个 DMA 通道都可以配置其安全模式。
- 输出中断信号用来标志 DMA 事件的产生。

2．DMAC 配置特点

下面这些特性为 DMAC 的配置特性。

- AXI 数据总线宽度。
- AXI 读处理活动的个数。
- AXI 写处理活动的个数。
- 并发性的 DMA 通道个数。
- 内部数据缓冲的深度。
- 指令 Cache 的行数，一行的字数。
- 读指令队列的深度。
- 写指令队列的深度。
- 外设请求接口的个数。
- 中断输出信号的个数。

12.1.3 PL330 简述

DMAC 包含了一个执行指令的模块，并且控制了数据的传输，DMAC 通过 AXI 接口来存取这些存储在内存中的指令，DMAC 还可以将一些临时的指令存放在 Cache 中，我们能够配置行宽度及深度。

当然，DMAC 的 8 个通道都是可配置的，且每个都可支持单个并发线程的操作，除此之外，还有一个管理线程专门用来初始化 DMA 通道的线程。它是用来确保每个线程都在正常工作，它使用了 round-robin 来处理当选择执行下一个活动期的 DMA 通道。

DMAC 使用了变长指令集，范围在 1 到 6 字节之间，还为每个通道提供了单独的 PC 寄存器，当一个线程需要执行一条指令时，将先从 Cache 中搜索，如果匹配上则立刻供给数据，另外，线程停止的话，DMAC 将使用 AXI 接口来执行一次 Cache 线填充。

当一个 DMA 通道线程执行一次 load/store 指令，DMAC 将添加指令到有关的读队列和写队列中，DMAC 使用这些队列作为一个指令存储区，它用来优先执行存储在其中的指令，DMAC 还包含了一个 MFIFO 数据缓存区，它用来存储 DMA 传输中读/写的数据。

DMAC 还提供多个中断输出，如微处理器的不干扰，外设的 request 接口还有内存到外设和外设到内存的传输能力，双 APB 接口支持安全及非安全两种模式，编程时，可通过 APB 接口来访问状态寄存器和直接执行 DMAC 指令。

如图 12-3 所示为 S5PC100 中的 DMA 模块图。

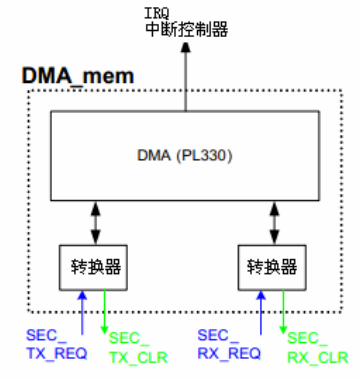

图 12-3　S5PC100 中的 DMA 模块图

如图 12-4 所示为 DMAC 模块图。

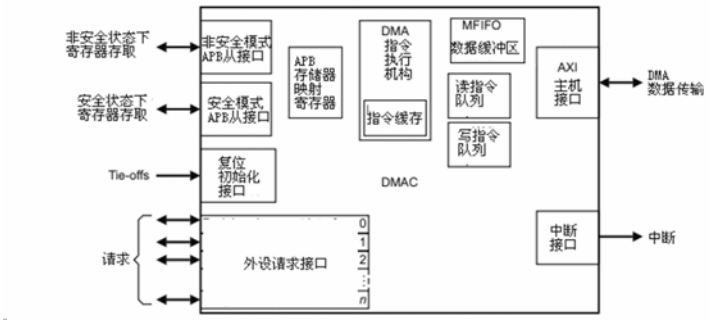

图 12-4　DMAC 模块图

从图 12-4 中可以看出，APB 从机接口下有安全模式及非安全模式两种接口，它们分别能在不同的模式下执行不同需求的功能，寄存器是彼此独立的，也就是各自有自己的一套寄存器，另外，我们还能看到 READ/WRITE 指令队列，当 DMAC 从指令中取到后则先存放在相应的队列中等待执行，MFIFO 则是前文提到的数据缓冲区域，这是一个可配置大小的缓存区，当执行读指令后，DMAC 从源地址中获得数据后，将其先存放在 MFIFO 中，当满足事先设定的触发写条件时，DMAC 则会从 MFIFO 中写数据到目的地址。

12.2　PL330 详解

12.2.1　PL330 指令集

下面列举出一些常用的指令，更多内容，参考 ARM 官网上的 DDI0424A_dmac_pl330_r0p0_trm.pdf 手册。

1．DMAMOV

指令格式：

```
DMAMOV     <dst_reg>,<32bit_immediate>
```

这是一条数据转移指令，它可以移动一个立即数到下面介绍的 3 种类型的寄存器中。

功能描述：

```
< dst_reg >
```

DMAMOV 二进制机器码的 10~8 位描述了目标寄存器的类型。

- B000 代表 SAR（源地址寄存器）。
- B010 代表 DAR（目标地址寄存器）。
- B001 代表 CCR（控制寄存器）。

1）源地址寄存器

该寄存器提供了 DMA 通道的数据源的地址，DMAC 从该地址取得数据，每个通道都有自己的数据源地址寄存器，因此需要单独配置。

如图 12-5 所示为每个通道的源地址寄存器列表。

寄存器地址映射

通道 *n*	0	1	2	3	4	5	6	7
寄存器名	SA_0	SA_1	SA_2	SA_3	SA_4	SA_5	SA_6	SA_7
地址偏移	0x400	0x420	0x440	0x460	0x480	0x4A0	0x4C0	0x4E0

图 12-5　通道源地址寄存器

寄存器详解如图 12-6 所示。

位	名字	功能
[31:0]	src_addr	数据源地址寄存器，注意，每一个通道的寄存器偏移

图 12-6　数据源地址寄存器详解

2）目标地址寄存器

该寄存器提供了 DMA 的目标数据存放地址，和数据源地址寄存器是相互对应的。

如图 12-7 所示为目标地址寄存器偏移。

寄存器地址映射

通道 *n*	0	1	2	3	4	5	6	7
寄存器名	DA_0	DA_1	DA_2	DA_3	DA_4	DA_5	DA_6	DA_7
地址偏移	0x404	0x424	0x444	0x464	0x484	0x4A4	0x4C4	0x4E4

图 12-7　目标地址寄存器偏移

寄存器详解如图 12-8 所示。

位	名字	功能
[31:0]	dst_addr	目标地址寄存器地址

图 12-8　目标地址寄存器详解

3）通道控制寄存器

该寄存器可以控制 DMA 在 AXI 中的传输，并且该寄存器记录了一些关于目标与源寄存器的基本配置。如图 12-9 所示为该寄存器的位分配。

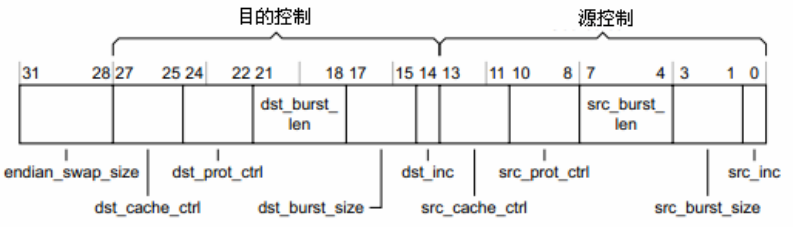

图 12-9　寄存器位分配图

如图 12-10 所示为每个通道的寄存器偏移列表。

寄存器地址映射

通道 *n*	0	1	2	3	4	5	6	7
寄存器名	CC_0	CC_1	CC_2	CC_3	CC_4	CC_5	CC_6	CC_7
地址偏移	0x408	0x428	0x448	0x468	0x488	0x4A8	0x4C8	0x4E8

图 12-10　每个通道的寄存器偏移列表

<32bit_immediate>为 32 位立即数，可被传到指定的寄存器中。

2．DMALD

DMALD 是一条 DMAC 装载指令，它可以从源数据地址中读取数序到 MFIFO 中，如果 src_int 位被设置，则 DMAC 会自动增加源地址的值，如图 12-11 所示。

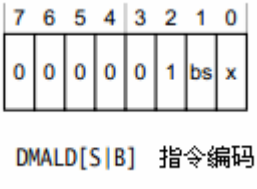

图 12-11　DMALD

指令格式：

```
DMALD[S|B]
```

功能描述如下。

[S]：如果 S 位被指定，则 bs 位被设置为 0，且 x 转换为 0。Request_flag 将被下列情况所影响：

- Request_flag=Single，DMAC 将执行 DMA 装载。
- Request_flag=Burst，DMAC 将执行 DMANOP。

[B]：如果 B 位被指定，则 bs 会置 0，且 x 转换为 1，Request_flag 将被下列情况所影响：

- Request_flag=Single，DMAC 将执行 DMANOP。
- Request_flag=Burst，DMAC 将执行 DMA 装载。

注意：

如果不指定 S、B 位的话，则 DMAC 默认是执行 DMA 装载的。

3．DMAST

该指令与 DMALD 相互对应，它是一条 DMA 存储指令，是将 MFIFO 中的数据转移到目的地址中。目的地址是由目的地址寄存器所指定的，如果 dst_inc 被置位，则 DMAC 会自动增加目的地址的值。

图 12-12　DMAST

指令格式：

```
DMAST[S|B]
```

功能描述如下。

[S]：如果 S 位被指定，则 bs 位被设置为 0，且 x 转换为 1。Request_flag 将被下列情况所影响：

- Request_flag=Single，DMAC 执行单个 DMA 存储。
- Request_flag=Burst，DMAC 执行空指令。

[B]：如果 B 位被指定，则 bs 被设置为 1，且 x 转换为 1，Request_flag 将被下列情况所影响：

- Request_flag=Single，DMAC 执行空指令。
- Request_flag=Burst，DMAC 将执行 DMA 存储。

注意：

如果 S、B 位都不指定，DMAC 默认执行 DMA 存储，且只在 MFIFO 达到可以正常传输的条件下才执行一次传输。

4．DMARMB

读内存栅栏指令，如图 12-13 所示为该指令的译码图。

图 12-13　DMARMB 译码

指令格式：

```
DMARMB
```

功能描述：该指令可以使得当前所有读处理全部被强制取消。

5．DMAWMB

写内存栅栏指令，如图 12-14 所示为该指令的译码图。

图 12-14　DMAWMB 译码

指令格式：

```
DMAWMB
```

功能描述：该指令可以使得写数据管道强制清空。

6．DMAL

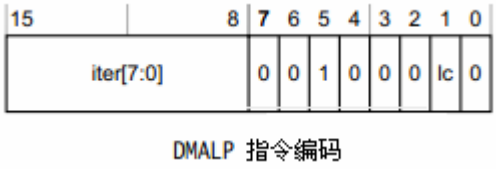

图 12-15　DMALP 译码

指令格式：

```
DMALP  <loop_iterations>
```

<loop_iterations>是一个 8 位表示的循环次数。

- lc 设置为 0 时，DMAC 每写一次值，loop_iterations 则减少 1，直到循环计数为 0 结束。

- lc 设置为 1 时，DMAC 每写一次值，loop_iterations 则减少 1，直到循环计数为 1 结束。

功能描述：循环操作时，将一个指定的 8bit 数字填入循环计数寄存器，该指令用来指定某个指令段的开始位置，需要 DMALPEND 指定该指令段的结束位置，一旦指定后，DMAC 会循环执行介于 DMALP 与 DMALPEND 之间的指令，直到循环次数为 0 结束。

7．DMALPEND

DMALPEND 译码如图 12-16 所示。

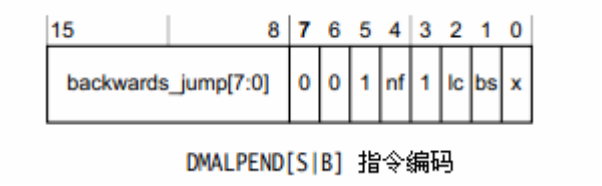

图 12-16　DMALPEND 译码

指令格式：

```
DMALPEND[S|B]
```

[S]：如果 S 位被指定，则 bs 位被设置为 0，且 x 转换为 1。Request_flag 将被下列情况所影响：

- Request_flag=Single，DMAC 将执行循环。
- Request_flag=Burst，DMAC 执行空指令。

[B]：如果 B 位被指定，则 bs 被设置为 1，且 x 转换 1，Request_flag 将被下列情况所影响：

- Request_flag=Single，DMAC 执行空指令。
- Request_flag=Burst，DMAC 将执行循环。

功能描述：该指令每次执行一遍以后查看循环计数寄存器的值。

- 如果是 0，DMAC 则执行 DMANOP 指令。
- 如果不为 0，DMAC 则更新一次循环计数器的值，并跳转到循环指令段的第一条指令执行。

8．DMASEV

DMASEV 译码如图 12-17 所示。

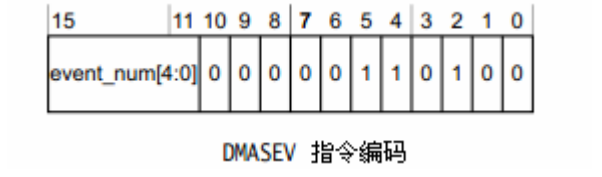

图 12-17　DMASEV 译码

指令格式：

```
DMASEV      <event_num>
<event_num>是 5 位立即数。
```

注意：

如果该数值配置不正确会引发 DMAC 线程中止。

功能描述：使用该命令可以产生一个事件信号。可以有以下两种模式。

- 产生一个事件<event_num>。
- 产生一个中断信号 irq<event_num>。

9．DMAEND

DMAEND 译码如图 12-18 所示。

图 12-18　DMAEND 译码

指令格式：

```
DMAEND
```

功能描述：该指令用来通知 DMAC 结束一次操作集合，换句话说就是，告诉 DMAC 某个线程停止一切的动作，使其为停止态，这时 DMAC 会刷新 MFIFO，并且清空所有相关的 Cache。

12.2.2　相关寄存器详解

1．DBGINST0

此寄存器可控制调试指令、通道、DMAC 线程信息，如图 12-19 所示为寄存器的详细解释。

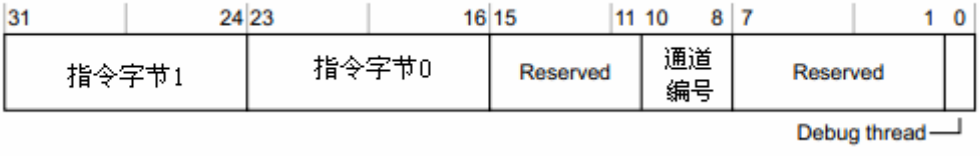

图 12-19　DBGINST0 寄存器

如图 12-20 所示为该寄存器的位分配。

位	名字	功能
[31:24]	指令字节1	指令字节1
[23:16]	指令字节0	指令字节0
[15:11]	-	Reserved.
[10:8]	通道号码	DMA 通道号码： b000 = DMA 通道 0 b001 = DMA 通道 1 b010 = DMA 通道 2 . . . b111 = DMA 通道 7.
[7:1]	-	Reserved.
[0]	Debug thread	0 = DMA 管理线程 1 = DMA 通道

图 12-20　DBGINST0 寄存器位定义

2．DBGINST1

该寄存器控制内存中设置的指令段首地址，也就是 DMAC 第一次取指令的地址。如图 12-21 所示为该寄存器的解释。

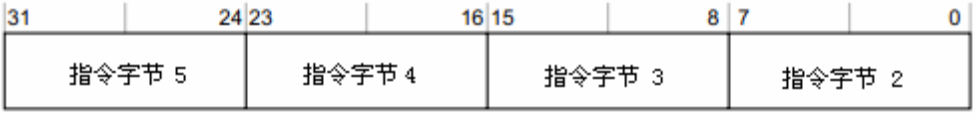

图 12-21　DBGINST1 详解

如图 12-22 所示为寄存器位定义。

位	名字	功能
[31:24]	指令字节 5	指令字节 5
[23:16]	指令字节 4	指令字节 4
[15:8]	指令字节 3	指令字节 3
[7:0]	指令字节 2	指令字节 2

图 12-22　寄存器位定义

3．DBGCMD

该寄存器控制调试命令的执行，通过配置它，可以控制 DMAC 去执行一些指定的工作。该寄存器的详细解释如图 12-23 所示。

位	名字	功能
[31:2]	-	保留
[1:0]	dbgcmd	b00 = 执行DBGINST[1:0]控制的指令 b01 = 保留 b10 = 保留 b11 = 保留

图 12-23　DMGCMD 详解

12.3　S5PC100 PL330 测试例子

由于 PL330 学习起来略显烦琐，因此建议读者在理解代码的基础上做实验，这样才能对 DMA 的学习有一定的深入。篇幅有限这里只给出核心的代码，若读者想参考完整源代码，请去华清远见官方论坛上下载。

如图 12-24 所示为 DMAC 控制流程。

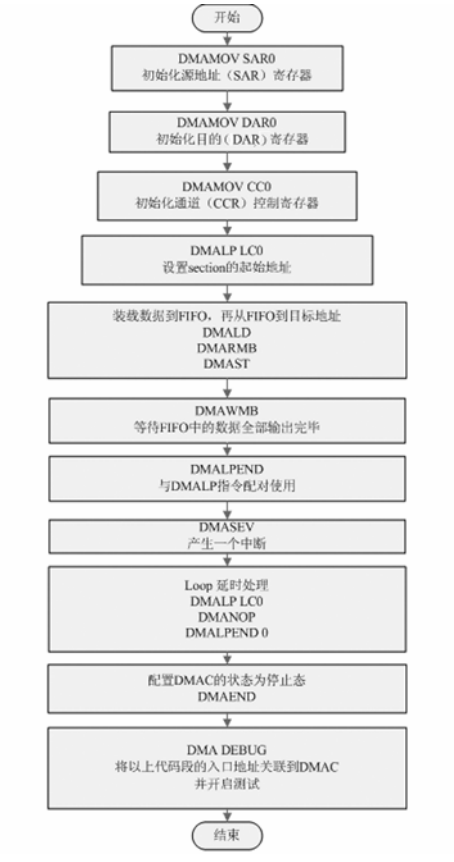

图 12-24　DMAC 控制流程

配合上面的流程图，可以编写代码如下。

（1）相关的宏定义。

```
#define MAX 100
#define Inp(addr)            (*(volatile unsigned int *)(addr))
#define Outp(addr, data)     (*(volatile unsigned int *)(addr) = (data))

extern void printf(const char *fmt, ...);
void int_dma();
volatile char sour[32] = "01234567890123456789012345678 9\n";
volatile char dest[32] = "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb\n";
```

（2）设置 SAR、CCR、DAR 寄存器。

```
//main 函数开始
uart0_init();
 volatile char instr_seq[MAX];
 int size = 0, x;
 int loopstart, loopnum = 2;
 unsigned int source, destination, start, temp;
 source = (unsigned int)sour;
 destination = (unsigned int)dest;
 start = (unsigned int)instr_seq;
 /*setup channel0 for m2m*/

 /*DMAMOV SAR0*/
 instr_seq[size + 0] = (char)(0xbc);
 instr_seq[size + 1] = (char)(0x0);
 instr_seq[size + 2] = (char)((source>>0) & 0xff);
 instr_seq[size + 3] = (char)((source>>8) & 0xff);
 instr_seq[size + 4] = (char)((source>>16) & 0xff);
 instr_seq[size + 5] = (char)((source>>24) & 0xff);
 size = 6;
 /*DMAMOV DAR0*/
 instr_seq[size + 0] = (char)(0xbc);
 instr_seq[size + 1] = (char)(0x2);
 instr_seq[size + 2] = (char)((destination>>0) & 0xff);
 instr_seq[size + 3] = (char)((destination>>8) & 0xff);
 instr_seq[size + 4] = (char)((destination>>16) & 0xff);
 instr_seq[size + 5] = (char)((destination>>24) & 0xff);
 size += 6;
 /*DMAMOV CC0. burst_size 8byte, burst_len 2*/
 instr_seq[size + 0] = (char)(0xbc);
 instr_seq[size + 1] = (char)(0x1);
 instr_seq[size + 2] = (char)(0x17);
 instr_seq[size + 3] = (char)(0xc0);
 instr_seq[size + 4] = (char)(0x5);
 instr_seq[size + 5] = (char)(0x0);
 size += 6;
```

（3）设置指令段的起始地址及执行第一次数据装载并输出 FIFO。

```
 /*DMALP LC0*/
 instr_seq[size + 0] = (char)(0x20);
```

```
 instr_seq[size + 1] = (char)(loopnum - 1);// loopnum should <= 256
 size += 2;
 loopstart = size;
 /*DMALD*/
 instr_seq[size + 0] = (char)(0x04);
 size += 1;
 /*DMARMB*/
 instr_seq[size + 0] = (char)(0x12);
 size += 1;
 /*DMAST*/
 instr_seq[size + 0] = (char)(0x08);
 size += 1;
 /*DMAWMB*/
 instr_seq[size + 0] = (char)(0x13);
 size += 1;
```

（4）产生中断，并延时一段时间。

```
/*DMALPEND 0*/
 instr_seq[size + 0] = (char)(0x38);
 instr_seq[size + 1] = (char)(size - loopstart);
 size += 2;
 /*DMASEV*/
 instr_seq[size + 0] = (char)(0x34);
 instr_seq[size + 1] = (char)(1<<3);
 size += 2;
#endif
/*for loop delay*/

 /*DMALP LC0*/
 instr_seq[size + 0] = (char)(0x20);
 instr_seq[size + 1] = (char)(4);
 size += 2;
 loopstart = size;
 /*DMANOP*/
 instr_seq[size + 0] = (char)(0x18);
 size += 1;
 /*DMALPEND 0*/
 instr_seq[size + 0] = (char)(0x38);
 instr_seq[size + 1] = (char)(1);
 size += 2;
```

（5）结束 DMAC 控制。

```
/*DMAEND*/
 instr_seq[size + 0] = (char)(0x0);
 size += 1;
```

（6）开始 DMAC 控制，设置相应的中断处理，并进行测试结果。

```
/*enable irq*/
 VIC0VECADDR18 = (unsigned int)int_dma;
 INTERRUPT.VIC0INTENABLE |= 1<<18;
```

```
    Outp(0xE8100000+0x20, 0x3);//enable dma INTEN
    /*DMAGO*/
    do{
         x = Inp(0xE8100D00);//check DBGSTATUS
    } while ((x&0x1)==0x1);
    Outp(0xE8100D00+0x8, (0<<24)|(0xa0<<16)|(0<<8)|(0<<0));//DBGINST0
    Outp(0xE8100D00+0xC, start);//DBGINST1
    Outp(0xE8100D00+0x4, 0);//DBGCMD

    while(1);
  //main 函数结束
```

（7）ISR 函数的实现如下。

```
  void do_irq()
  {
    printf("in do_irq\n");
    ((void (*)(void))VIC0ADDRESS)();
  }
  /*ISR*/
  void int_dma()
  {
    VIC0ADDRESS = 0;
    Outp(0xE8100000+0x2C, 0x3);//clear dma INTCLR
    printf("DMA Ending!\n");
    printf("sour = %s", sour);
    printf("dest = %s", dest);
  }
```

实验调试过程与结果：

（1）将程序编译后生成.bin 文件，打开终端使用 uboot 的 dnw 命令通过 USB 线将.bin 文件下载到 0x20008000 这个地址，接着使用 go 命令去执行测试程序。

（2）可以看到下图所示的测试结果，如图 12-25 所示。

```
Hit any key to stop autoboot:  0
SMDKC100 #
SMDKC100 #
SMDKC100 # dnw 20008000
OTG cable Connected!
Now, Waiting for DNW to transmit data
Download Done!! Download Address: 0x20008000, Download Filesize:0x16d4
Checksum is being calculated.
Checksum O.K.
SMDKC100 # go 20008000
## Starting application at 0x20008000 ...
open uart device ok !
in do_irq
DMA Ending!
sour = 01234567890123456789012345678 9
dest = 01234567890123456789012345678 9bb
```

图 12-25　DMA 测试结果

12.4　本章小结

本章内容从 PL330 出发，介绍了基本的 S5PC100 下 DMA 控制器的操作方式和编程模型，旨在将最新的 DMA 控制技术以最简单的方式教授给读者。

12.5　练习题

1．使用 PL330 作为 DMA 控制器的好处。
2．写一个基于外设到外设的 DMA 传输模型。

第 13 章　LCD 接口设计

液晶屏（Liquid Crystal Display，LCD）即人们常说的液晶显示器，具有耗电省，体积小等特点，被广泛地应用于嵌入式系统中。本章主要介绍它的接口设计。

主要内容有：

- LCD 控制器介绍。
- 接口电路与程序设计。

13.1　LCD 控制器

13.1.1　LCD 控制器介绍

1．液晶屏的分类

液晶显示屏按显示原理分为 STN 和 TFT 两种。

STN（Super Twisted Nematic，超扭曲向列）液晶屏：STN 液晶显示器与液晶材料、光线的干涉现象有关，因此显示的色调以淡绿色与橘色为主。STN 液晶显示器中，使用 X、Y 轴交叉的单纯电极驱动方式，即 X、Y 轴由垂直与水平方向的驱动电极构成，水平方向驱动电压控制显示部分为亮或暗，垂直方向的电极则负责驱动液晶分子的显示。STN 液晶显示屏加上彩色滤光片，并将单色显示矩阵中的每一像素分成三个子像素，分别通过彩色滤光片显示红、绿、蓝三原色，也可以显示出色彩。单色液晶屏及灰度液晶屏都是 STN 液晶屏。

TFT（Thin Film Transistor，薄膜晶体管）彩色液晶屏：随着液晶显示技术的不断发展和进步，TFT 液晶显示屏被广泛用于制作成计算机中的液晶显示设备。 TFT 液晶显示屏既可以在笔记本电脑上应用（现在大多数笔记本电脑都使用 TFT 显示屏），也常用于主流台式显示器。

2．液晶屏的显示

液晶屏的显示要求设计专门的驱动与显示控制电路。驱动电路包括提供液晶屏的驱动电源和液晶分子偏置电压，以及液晶显示屏的驱动逻辑；显示控制部分可由专门的硬件电路组成，也可以采用集成电路（IC）模块，比如 EPSON、Silicon Motion 的显示卡驱动器等；还可以使用处理器外围 LCD 控制模块。

13.1.2　S5PC100 的 LCD 控制器介绍

S5PC100 处理器集成了 LCD 控制器，主要功能是 S5PC100 LCD 控制器用于传输显示数据和产生控制信号。它支持屏幕水平和垂直滚动显示。数据的传送采用 DMA（直接内存访问）方式，以达到最小的延迟。它可以支持多种液晶屏。

STN LCD 显示器性能如下。

- 支持 3 种类型的扫描方式：4 位单扫描、4 位双扫描和 8 位单扫描。
- 支持 256 色和 4096 色彩色 STN LCD。
- 典型的实际屏幕大小是：640×480、320×240、160×160 等。
- 最大虚拟屏幕占内存大小为 4M 字节。
- 256 色模式下最大虚拟屏幕大小：4096×1024、2048×2048、1024×4096 等。

TFT LCD 显示器性能如下。

- 支持 1、2、4 或 8bpp 彩色调色显示。
- 支持 16bpp 和 24bpp 非调色真彩显示。
- 在 24bpp 模式下，最多支持 16M 种颜色。
- 支持多种屏幕大小。
- 典型的实际屏幕大小是：640×480、320×240、160×160 等。
- 最大虚拟屏幕占内存大小为 4M 字节。
- 64K 色模式下最大虚拟屏幕大小：2048×1024 等。

1．S5PC100 LCD 控制器功能简述

S5PC100 所集成的 LCD 控制器功能很强大，其中包含了一个本地总线传输图像数据的逻辑模块，以及内置的图像处理单元。这些模块都可通过总线连接至外接的 LCD 接口，LCD 接口包含了 4 种类型，有 RGB 接口、间接的 i80 接口、ITU-R BT.601/656 接口，显示控制器支持最多 5 个叠加图像窗口，每个窗口都支持多种图像格式，以及 256 灰度级绑定，颜色锁定，x-y 坐标控制，软件卷动，可变的窗体尺寸等。

显示控制器支持多种颜色格式，例如 RGB（1bpp-24bpp），YcbCr4:4:4（限于本地总线），显示控制器可编程支持不同需求的图像像素，数据线宽度，时序，刷新率。

2．LCD 外部接口信号

S5PC100 的 LCD 控制器包括了两个时序部分，一个是针对于 RGB 接口，ITU-R601/656 的时序，一个是针对间接 i80 接口的时序。本章将重点介绍关于 RGB 接口的控制器部分。

RGB VIME 产生的控制信号有 VSYNC（垂直同步信号）、HSYNC（水平同步信号）、VDEN（数据有效信号）、VCLK（LCD 时钟），这些信号都可由寄存器配置，还有 VD[23:0] 的数据输出口。如图 13-1 所示为 LCD-RGB 接口时序。

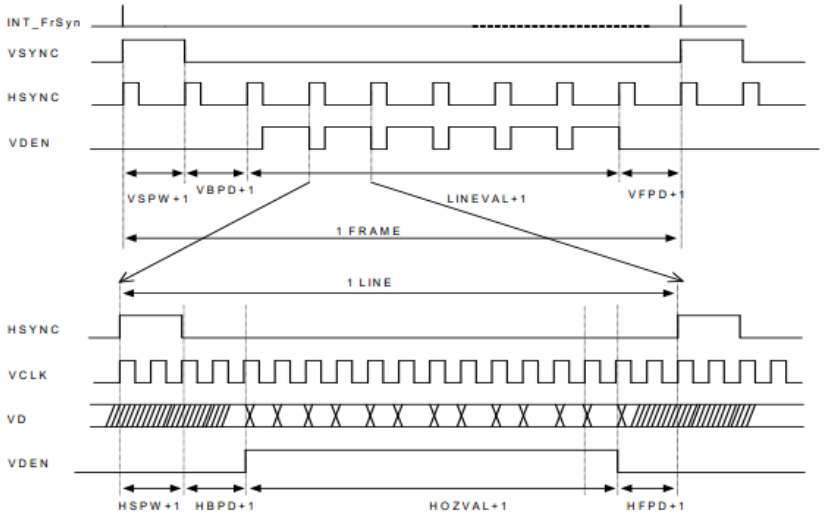

图 13-1　LCD RGB 接口时序图

13.1.3　S5PC100 的 LCD 控制器操作

基于前面介绍的 RGB 接口时序图，我们现在可以简单看看，LCD 的控制流程，下面给出几个简单公式：

HOZVAL = (Horizontal display size -1)

LINEVAL= (Vertical display size -1)

VCLK(Hz)= HCLK/(CLKVAL+1) where CLKVAL >= 1

上述 3 个公式后面在配置寄存器的时候都需要使用到，这里先提出来。下面简单解释一下 RGB 接口时序图的意义，在一帧画面的呈现中，关系到如下步骤，首先 LCD 控制器发出一次 VSYNC 信号，这时会伴随发出 HSYNC 信号，可以想象一下，LCD 屏的显示方式，先选中第一行（VSYNC 信号），然后从第一列开始顺序选中（HSYNC 信号），在每一次的 HSYNC 中，会发生数据传输，而这个时候是由 VCLK 来决定的。

在每一帧时钟信号中，还会有一些与屏显示无关的时钟出现，这就给确定行频和场频带来了一定的复杂性。如在 HSYNC 信号先后会有水平同步信号前肩（HFPD）和水平同步信号后肩（HBPD）出现，在 VSYNC 信号先后会有垂直同步信号前肩（VFPD）和垂直同步信号后肩（VBPD）出现，在这些信号时序内，不会有有效像素信号出现，另外 HSYNC 和 VSYNC 信号有效时，其电平要保持一定的时间，它们分别叫做水平同步信号脉宽（HSPW）和垂直同步信号脉宽（VSPW），这段时间也不能有像素信号。因此计算行频和场频时，一定要包括这些信号。HBPD、HFPD 和 HSPW 的单位是一个 VCLK 的时间，而 VSPW、VFPD 和 VBPD 的单位是扫描一行所用的时间。

在 S5PC100 中，还需要重点考虑 alpha 绑定机制，因为它是 5 个窗口叠加共同成像的原理，因此需要配置一下 alpha 绑定方程，如图 13-2 所示。

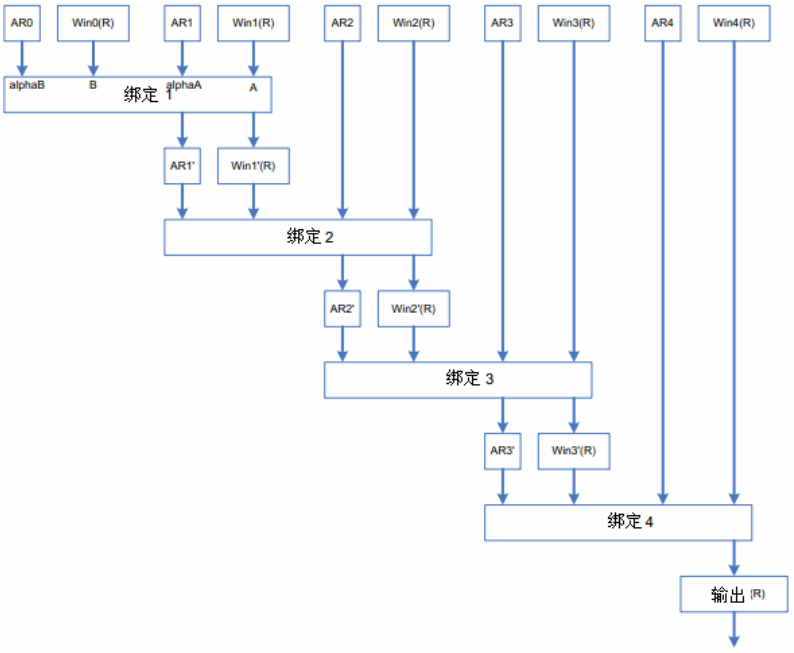

图 13-2　alpha 绑定方程

从图 13-2 中可以看到，每一次的叠加由两个窗口进行，窗口的顺序依次是：

- X0　=　窗口 0 与窗口 1。
- X1　=　X0 与窗口 2。
- X2　=　X1 与窗口 3。
- X4　=　X2 与窗口 4。

最后的 X4 为 LCD 所呈现的图像，如图 13-3 所示。

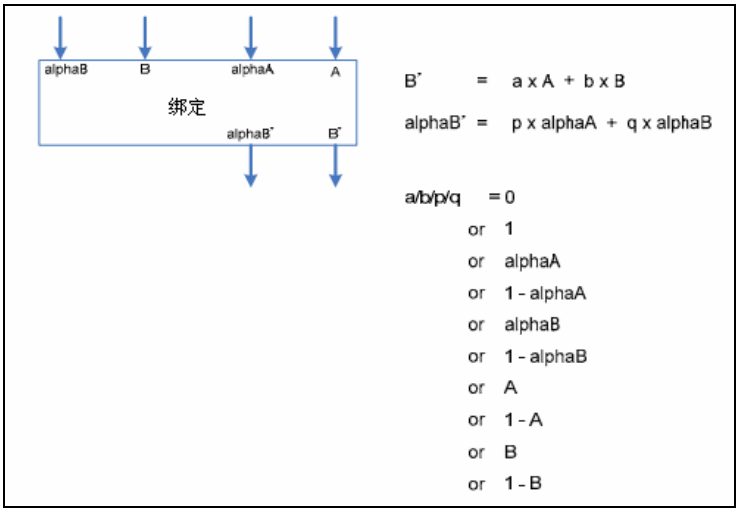

图 13-3　绑定方程

简单解释一下这两个线性方程，B' 是一个色值函数，它由 A 的色值分量与 B 的色值分量线性叠加而成。此处，A 的色值来自 win(n+1)，B 的色值来自 win(n)，a、b 则是线性因子。我们只需要考虑 a、b 的值，就可以得到我们想要的结果，同理 alphaB' 是一个 B' 色值的伴随灰度值函数，专业点说，就是一组 alpha 通道。

它的构成与前一个方程是同构的。p、q 也是线性因子，这样 4 个因子决定了两个窗口的最终叠加色值及伴随 alpha 的值（注意，a、b、p、q 是可选值）。

在 LCD 控制器中，只需要配置 WINn blending equation control register，并将具体的因子配好后，就可以实现了，注意 5 个窗口需要同时配置方程。

再来重点考察一下 alphaB 灰度值，它是一个 8 位 s 的值，0~255 分别代表了不同的灰度级。从方程中可以看出，如果一个 alpha 值与一个色值相乘后，就会得到一个该色值的分量，这样两个窗口的叠加就靠不同的灰度来确定。换句话说，如果要使得 win1-win4 窗体整体透明，那必须要使得 win1-win4 的灰度级最高，则使得 alpha value 为 0，并且色值方程配置 pn=0，qn=0，an=alphaA，bn=(1-alphaA)，其中 n={1,2,3,4}即可实现 window 0 显示，而其他窗口透明。

13.1.4 LCD 控制器寄存器

由于在 S5PC100 中 LCD 控制器的寄存器众多，这里只简单介绍一下我们要用到、并且很重要的寄存器，如表 13-1~13-10 所示，如果想知道更多的信息，请参看 S5PC100 手册。

表 13-1 VIDCON0，R/W,ADDRESS=0xEE000000

域	位	描述	复位值
保留	[31]	保留	0
INTERLACE_F	[29]	逐行扫描方式或者间隔扫描方式 0 = 逐行扫描 1 = 间隔扫描方式(only ITU601/656 Interface)	0
VIDOUT	[27:26]	输出格式： 000: RGB I/F 001 = ITU601/656 010 = Indirect I80 I/F for LDI0 011 = Indirect I80 I/F for LDI1	000
PNRMODE	[18:17]	选择显示模式 (Where, VIDOUT[1:0] == 2’b00). 00 = RGB 格式 (RGB) 01 = RGB 格式 (BGR) 10 = Serial Format (R->G->B) 11 = Serial Format (B->G->R)	00
CLKVALUP	[16]	选择 CLKVAL_F 刷新时序的模式 0 = 总在刷新 1 =当一帧开始时	0

续表

域	位	描述	复位值
CLKVAL_F	[13:6]	决定 VCLK 的速率及 CLKVAL[7:0] VCLK = HCLK / (CLKVAL+1)　且 CLKVAL >= 1 **Note.** 1. VCLK 最大值是 66MHz 2. CLKSEL_F 寄存器选择的时钟源	0
VCLKFREE	[5]	VCLK 控制方式 0 =普通模式（ENVID） 1 = 自由模式	0
CLKDIR	[4]	选择时钟源的通道 0 = 直接获取时钟（VCLK = Clock source） 1 = 除以分频值	0
CLKSEL_F	[2]	选择时钟源 0 = HCLK 1 = SCLK_LCD	0
ENVID	[1]	数据输出使能位 0 = 禁止 1 = 使能	0
ENVID_F	[0]	当前帧结束使能位 0 = 禁止 1 = 使能 如果该位被设置，则该位直到一帧结束时才被禁止	0

表 13-2　VIDCON1,R/W,ADDRESS=0xEE000004

域	位	描述	复位值
LINECNT (read only)	[26:16]	提供 line counter 的计数值（read only） 从 0 到 LINEVAL	0
FSTATUS	[15]	场状态（read only） 0 = 非平坦场 1 = 平坦场	0
VSTATUS	[14:13]	垂直状态（read only） 00 = VSYNC　　01 = 后肩 10 = ACTIVE　　11 = 前肩	0
保留	[12:8]	保留	
IVCLK	[7]	VCLK 极性 0 = VCLK 下降沿取数据 1 = VCLK 上升沿取数据	0

续表

域	位	描述	复位值
IHSYNC	[6]	该位决定了 HSYNC 脉冲极性 0 = 普通　　1 = 反转	0
IVSYNC	[5]	该位决定了 VSYNC 脉冲极性 0 = 普通　　1 = 反转	0
IVDEN	[4]	该位决定了 VDEN 信号极性 0 = 普通　　1 = 反转	0
保留	[3:0]	保留	0x0

表 13-3　VIDTCON0,R/W,ADDRESS=0XEE000010

域	位	描述	Reset
VBPD	[23:16]	垂直后肩所占周期	0x00
VFPD	[15:8]	垂直前肩所占周期	0x00
VSPW	[7:0]	垂直同步脉冲宽度	0x00

表 13-4　VIDTCON1,R/W,ADDRESS=0XEE000014

域	位	描述	复位值
HBPD	[23:16]	水平后肩所占周期	0x00
HFPD	[15:8]	水平前肩所占周期	0x00
HSPW	[7:0]	水平同步脉冲宽度	0x00

表 13-5　VIDTCON2,R/W,ADDRESS=0XEE000018

域	位	描述	复位值
LINEVAL	[21:11]	该位决定了显示屏的垂直尺寸	0
HOZVAL	[10:0]	该位决定了显示屏的水平尺寸	0

表 13-6　VIDTCON2,R/W,ADDRESS=0XEE000020

域	位	描述	复位值
ENLOCAL	[22]	数据存取路径. 0 = DMA 方式 1 = 本地方式（CAMIF 0 ）	0
BUFSTATUS	[21]	缓冲区的编号（这是因为 windows 0,1 可以有两个缓冲区）(Read Only) 0 = 缓冲区 0　　1 = 缓冲区 1	0
BUFSEL	[20]	选择缓冲区(0/1) 0 = 缓冲区 0　　1 = 缓冲区 1	0
BUFAUTOEN	[19]	双缓冲自动控制位 0 = BUFSEL, 1 = 自动改变由 Trigger Input 控制	0
位 SWP	[18]	位交换控制位 0 = 禁止交换　　1 = 使能交换	0

续表

域	位	描述	复位值
BYTSWP	[17]	字节交换控制位 0 = 禁止交换　　1 = 使能交换	0
HAWSWP	[16]	半字交换控制位 0 = 禁止交换　　1 = 使能交换	0
WSWP	[15]	字交换控制位 0 = 禁止交换　　1 = 使能交换	0
保留	[14]	保留	0
InRGB	[13]	输入的源图像的格式 (Only for ‘EnLcal’ enable) 0 = RGB 1 = YCbCr	0
保留	[12:11]	保留 (should be 00)	0
BURSTLEN	[10:9]	DMA’s Burst 最大长度 00 = 16 字 01 = 8 字 10 = 4 字	0
保留	[8:7]	保留	0
BLD_PIX	[6]	选择绑定的类型 0 = 平面绑定 1= 像素绑定	0
BPPMODE_F	[5:2]	BPP (位 s Per Pixel)模式 0000 = 1 bpp 0001 = 2 bpp 0010 = 4 bpp 0011 = 8 bpp (palletized) 0100 = 8 bpp (无调色盘, A: 1-R:2-G:3-B:2) 0101 = 16 bpp (无调色盘, R:5-G:6-B:5) 0110 = 16 bpp (无调色盘, A:1-R:5-G:5-B:5) 0111 = 16 bpp (无调色盘, I :1-R:5-G:5-B:5) 1000 = unpacked 18 bpp (无调色盘, R:6-G:6-B:6) 1001 = unpacked 18 bpp (无调色盘, A:1-R:6-G:6-B:5) 1010 = unpacked 19 bpp (无调色盘, A:1-R:6-G:6-B:6) 1011 = unpacked 24 bpp (无调色盘, R:8-G:8-B:8) 1100 = unpacked 24 bpp (无调色盘,A:1-R:8-G:8-B:7) *1101 = unpacked 25 bpp (无调色盘, A:1-R:8-G:8-B:8) *1110 = unpacked 13 bpp (无调色盘,A:1-R:4-G:4-B:4) 1111 = unpacked 15 bpp (无调色盘, R:5-G:5-B:5)	

续表

域	位	描述	复位值
ALPHA_SEL	[1]	选择 alpha 值的方式 平面绑定时： 0 = using ALPHA0_R/G/B values 1 = using ALPHA1_R/G/B values 像素绑定时： 0 = AEN 使能位置 （细节请参看 S5PC100 手册）	
ENWIN_F	[0]	0 = 窗口禁止 1 = 窗口使能	

表 13-7　FRAME BUFFER ADDRESS 0,R/W,ADDRESS=0XEE0000A0

域	位	描述	复位值
VBANK_F	[31:24]	[31:24] 决定系统内存中的 bank 地址	0
VBASEU_F	[23:0]	[23:0] 决定 frame buffer.的起始地址	0

表 13-8　FRAME BUFFER ADDRESS 1,R/W,ADDRESS=0XEE0000D0

域	位	描述	复位值
VBASEL_F	[23:0]	[23:0] 决定了 frame buffer 的结束地址	0x0

表 13-9　FRAME BUFFER ADDRESS 2 ,R/W,ADDRESS=0XEE000100

域	位	描述	复位值
OFFSIZE_F	[25:13]	虚拟屏幕偏移（ byte）	0
PAGEWIDTH_F	[12:0]	虚拟页宽度（byte)	0

表 13-10　WINDOW 1 BLENDING EQUATION CONTROL REGISTER ,R/W,ADDRESS=0XEE000244

域	位	描述	复位值
保留	[31:22]	保留	0x000
Q_FUNC	[21:18]	alphaB 值为: 0000 = 0 (zero) 0001 = 1 (max) 0010 = **alphaA (alpha value of *foreground) 0011 = 1 － alphaA 0100 = alphaB 0101 = 1 － alphaB 011x = 保留 100x = 保留 1010 = A (foreground color data) 1011 = 1 － A 1100 = B (background color data) 1101 = 1 － B 111x = 保留	0x0

续表

域	位	描述	复位值
保留	[17:16]	保留	00
P_FUNC	[15:12]	alpha 值为: 同上	0x0
保留	[11:10]	保留	00
B_FUNC	[9:6]	B 的值为: 同上	0x3
保留	[5:4]	保留	00
A_FUNC	[3:0]	A 的值为: 同上	0x2

注意：
当中有很多寄存器都是每个窗口都有，但依旧有些差异，如 wincon0 和 wincon(1-4)的[7]位，其含义就不同，因此读者最好自己去查看手册。

13.2　LCD 控制器实例

有了上面对 LCD 控制器的了解，现在编写一个驱动例子，将一张分辨率为 480×272，颜色深度为 16 位图片显示在 LCD 屏，请结合下面的流程图（如图 13-4 所示）及代码模块，深入理解 LCD 驱动流程。

图 13-4　LCD 控制流程图

（1）下面代码，主要是一对 LCD 屏幕物理参数的设置，还有时序的配置，还有基本寄存器的配置。

```
/*下面这个结构体包含了 LCD 屏幕的物理参数，如时序信号的极性
* 屏幕像素大小，颜色深度，刷新频率
*/
static struct LcdPhyInfo innolux430 = {
 .width = 480,
 .height = 272,
 .bpp = 24,
 .freq = 60,

 .timing = {
      .h_fp = 2,
      .h_bp = 2,
      .h_sw = 41,
      .v_fp = 2,
      .v_fpe = 1,
      .v_bp = 2,
      .v_bpe = 1,
      .v_sw = 10,
 },

 .polarity = {
      .rise_vclk = 0,
      .inv_hsync = 1,
      .inv_vsync = 1,
      .inv_vden = 0,
 },
};

struct window win =
{
 .RgbMode = 0x5,          //RGB 模式;
 .DataComeFrom=_DMA,      //使用 DMA 方式获取数据
  .BLD_PIX=0,             //使用数据混合模式
 .ALPHA_SEL=0,            //使用灰度级模式 alpha0_R/G/B
 .OSD_LEFTOPX  =0,
 .OSD_LEFTOPY =0,
 .OSD_RIGHTBOTX = 480,
 .OSD_RIGHTBOTY = 272,
 .OSDSIZE = 480*272,
};
/*下面这个结构体中的配置，请读者对照前面给的那个 alpha 绑定公式并务必理解其含义*/
struct AlphaEquationFactor AlphaEquGlobal[]=
{
 [0] = {
         .winnum = 0,
          .q  = 0x0,
          .p  = 0x0,
         .b=0x3,
         .a=0x2,
      },
```

```
  [1] = {
          .winnum = 1,
           .q  = 0x0,
           .p  = 0x0,
          .b  = 0x3,
          .a  = 0x2,
        },
  [2] = {
          .winnum = 2,
           .q  = 0x0,
           .p  = 0x0,
          .b  = 0x3,
          .a  = 0x2,
        },
  [3] = {
          .winnum = 3,
           .q  = 0x0,
           .p  = 0x0,
          .b  = 0x3,
          .a  = 0x2,
        },

}; //它包含了 4 个窗口的灰度混合方程的因子
/*该结构体时钟贯穿全部代码*/
struct mylcd Lcd =
{
  .Phyinfo = &innolux430,
  .wcount  =1,    //you should open the win0 at last;
  .win[0] = &win,
  .AlphaEqu = AlphaEquGlobal,
  .fb = &fbaddr,
  .hoz = 480,
  .line = 272,
}
```

（2）首先是配置 VIDEO MAIN 和 VIDEO TIME 寄存器

```
/*
*配置:
*                显示主控制器
*                显示时序控制器
*/
int InitGobalReg(struct mylcd *lcd)
{
unsigned int cfg;
  struct LcdPhyInfo *phyinfo = lcd->Phyinfo;
cfg =0;
  cfg |= S3C_VIDCON0_CLKDIR_DIVIDED;  //选择时钟源路径
  cfg |= S3C_VIDCON0_CLKVAL_F(15);  //设置时钟
  writel(cfg, LCDBASEADDR + S3C_VIDCON0);
  /*下列所涉及的物理时序值需要参考所使用的 LCD 屏的出厂值，可参考对应的手册。*/
  cfg = 0;
  if (phyinfo->polarity.rise_vclk)
       cfg |= S3C_VIDCON1_IVCLK_RISING_EDGE;
```

```
 if (phyinfo->polarity.inv_hsync)
      cfg |= S3C_VIDCON1_IHSYNC_INVERT;
 if (phyinfo->polarity.inv_vsync)
      cfg |= S3C_VIDCON1_IVSYNC_INVERT;
 if (phyinfo->polarity.inv_vden)
      cfg |= S3C_VIDCON1_IVDEN_INVERT;
 writel(cfg, LCDBASEADDR + S3C_VIDCON1);
 /*there option reference the lcd control timing and
          you should to check the spec. of TFT-LCD*/
  cfg = 0;
 cfg |= S3C_VIDTCON0_VBPDE(phyinfo->timing.v_bpe-1);
 cfg |= S3C_VIDTCON0_VBPD(phyinfo->timing.v_bp-1);
 cfg |= S3C_VIDTCON0_VFPD(phyinfo->timing.v_fp-1);
 cfg |= S3C_VIDTCON0_VSPW(phyinfo->timing.v_sw-1);
 writel(cfg, LCDBASEADDR + S3C_VIDTCON0);
 cfg = 0;
 cfg |= S3C_VIDTCON1_VFPDE(phyinfo->timing.v_fpe-1);
 cfg |= S3C_VIDTCON1_HBPD(phyinfo->timing.h_bp-1);
 cfg |= S3C_VIDTCON1_HFPD(phyinfo->timing.h_fp-1);
 cfg |= S3C_VIDTCON1_HSPW(phyinfo->timing.h_sw-1);
 writel(cfg, LCDBASEADDR + S3C_VIDTCON1);
 cfg =0;
 cfg |= S3C_VIDTCON2_HOZVAL(lcd->hoz-1);
 cfg |= S3C_VIDTCON2_LINEVAL(lcd->line-1);
 writel(cfg, LCDBASEADDR + S3C_VIDTCON2);
 return 0;
}
```

（3）设置 win 控制寄存器。

```
static int window_control(struct mylcd *lcd,int whichwin)
{
 unsigned int cfg;
 if(whichwin == 0)
 {
      cfg = readl(LCDBASEADDR + S3C_WINCON(0));
      cfg = 0;
      cfg |= S3C_WINCON_HAWSWP_ENABLE;
      cfg |=((lcd->win[0]->RgbMode)<<2);
      writel(cfg, LCDBASEADDR+ S3C_WINCON(0));
}else{
cfg = readl(LCDBASEADDR + S3C_WINCON(whichwin));
      cfg = 0;
      cfg |=S3C_WINCON_HAWSWP_ENABLE;
      cfg |=((lcd->win[whichwin]->RgbMode)<<2);
      writel(cfg, LCDBASEADDR+ S3C_WINCON(whichwin));
 }
return 0;
}
```

（4）设置 win position 寄存器。

```
static void SetWinpos(struct window *win,int inum)
{
 unsigned int cfg;
```

```
     cfg=(win->OSD_LEFTOPX<<11) |(win->OSD_LEFTOPY<<0);
    writel(cfg,LCDBASEADDR+S3C_VIDOSD_A(inum));
    writel((win->OSD_RIGHTBOTX<<11)|(win->OSD_RIGHTBOTY<<0),LCDBASEADDR+S3C_VIDOSD_B(inum)
);
    writel(win->OSDSIZE,LCDBASEADDR+S3C_VIDOSD_C(inum));
   }
```

（5）配置 alpha 绑定函数的因子。

```
   void ConfigAlphaEquFactor(struct mylcd *lcd)
   {

    struct AlphaEquationFactor  *AlphaEqu = lcd->AlphaEqu;

    writel(set_factor(&AlphaEqu[0]), S3C_BLENDEQ1+ LCDBASEADDR);
    writel(set_factor(&AlphaEqu[1]),S3C_BLENDEQ2 + LCDBASEADDR);
    writel(set_factor(&AlphaEqu[2]),S3C_BLENDEQ3 + LCDBASEADDR);
    writel(set_factor(&AlphaEqu[3]),S3C_BLENDEQ4 + LCDBASEADDR);

   }
```

调试与实验结果：

作为测试的图片，可以先使用类似于 Image2Lcd 这样的软件，将一张 16 位的位图转换为 480×272×2 的数组，将其关联到 FrameBuffer 地址。

然后，将代码编译后生成.bin 文件，使用 uboot 的 dnw 功能，将其.bin 文件下载到 0x20008000 这个地址后，使用 go 命令执行测试程序。

这时图片就能显示在 LCD 的显示屏幕上了。

13.3　本章小结

本章主要介绍了基于 S5PC100 的 LCD 控制器，并且详细介绍了 LCD 控制器相关的寄存器及配置方法、编程模型等，最后通过一个实例实现了在 LCD 屏幕上显示一幅图片，希望读者能从中受益。上面的实验只实现了单幅图片 WIN0 的显示，采用的方法是只使能了 WIN0。笔者还实现了 WIN0 和 WIN1 两幅图片的叠层显示，通过修改 alpha 值能够实现半透明等效果，如果读者有兴趣研究，可以到华清远见论坛下载。

13.4　练习题

1．TFT 液晶显示屏外部接口信号有哪些？

2．简述 VFRAME、VLINE、VCLK 这几个信号的作用。

3．编程实现在 LCD 上显示一幅你自己的图片。

第 14 章　CAMIF 接口技术

摄像头技术早已应用在各大领域中，无论是电子消费类产品，还是工控安防，它的作用无疑是巨大的，因此掌握摄像头技术势在必行，所以本章节将介绍在 S5PC100S 下如何使用 CAMIF 接口，即摄像头接口。

本章要点：

- OV9650 介绍。
- SCCB 总线原理。
- S5PC100 的 CAMIF 接口详解。

14.1　OV9650 介绍

14.1.1　芯片功能描述

在介绍如何进行摄像头驱动的开发之前，首先要明白一个概念，那就是控制器和传感器芯片的关系，这里说的 OV9650 仅是一种传感器芯片的类型，它的主要作用仅仅是将光学信息通过物理原理被转换为电子特性，并通过芯片的一系列配置最终获得用户想要的数据。因此有必要将一款芯片示例介绍给读者，所谓会一通百，即便是换了一个芯片类型，只要掌握这样的开发原理，其他的芯片也是不难懂的。

下面介绍的 OV9650 是一款具有诸多功能的摄像头芯片，它被分为下面几个部分。

1．图像传感数组

OV9650 芯片集成了一个 1300 列×1028 行的图像传感数组。

2．时序产生器

一般来看，时序产生器可以控制图像传感数组，有内置的时序信号产生器，外接时序输出（VSYNC，HREF/HSYNC，PCLK）。

3．模拟处理模块

该模块提供了全部的模拟图像处理功能，包括曝光增益控制、自动白平衡，以及其他图像操作控制功能。

4．输出格式转换器

该模块有镜像图形控制、垂直翻转、YUV/Ycbcr 格式、RGB 模式、GRB4:2:2、RGB5:6:5RGB5:5:5。

5．SCCB 接口

OV9650 芯片使用了 SCCB 接口来对其进行操作，SCCB 协议是一种变体的 IIC 协议，将在后面详细介绍。

如图 14-1 所示为 OV9650 模块。

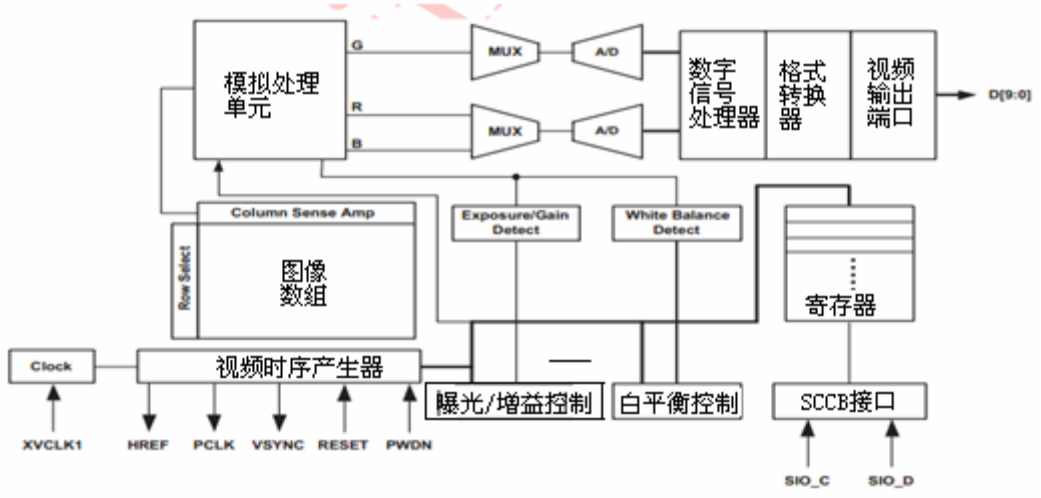

图 14-1　OV9650 模块

14.1.2　OV9650 物理参数

如表 14-1 所示为 OV9650 的物理参数。

表 14-1　OV9650 物理参数

类型		CMOS 摄像头模型（OV 9650 INSIDE）
动态存储尺寸		1300×1028 像素
电压	核心	1.8VDC±10%
电压	模拟输入电压	2.45 to 2.8 VDC
电压	I/O	2.5V to3.3 DC
功率 Requirements	工作时	50mW
功率 Requirements	待机时	30μW
温度 Range	操作期	-20℃ to 70℃

续表

类型		CMOS 摄像头模型（OV 9650 INSIDE）
温度 Range	数据稳定	0℃ to 50℃
输出格式（8bit）		YUV/YCbCr 4:2:2
输出格式（8bit）		GRB 4:2:2
输出格式（8bit）		原始 RGB 数据
最大图像 Transfer Rate	SXGA	15fps
最大图像 Transfer Rate	VGA	30fps
最大图像 Transfer Rate	QVGA,QQVGA,CIF	60fps
最大图像 Transfer Rate	QCIF,QQCIF	120fps
灵敏度		0.9V/Lux-sec
S/N 比例		40 dB
动态范围		62 dB
扫描模式		Progressive
最大曝光间隔		1050 × tROW
伽马校正		Programmable
像素尺寸		3.18μm×3.18μm
暗电流		30mV/s at 60℃
电容量		28Ke
固定图形噪声		<0.03% of VPEAK TO PEAK
图形域		4.13mm×3.28mm
包装尺寸		5095μm×5715μm

14.1.3 OV9650 寄存器详解

OV9650 芯片提供了 170 个寄存器集，其中包含了所有对 OV9650 光学传感器的配置，包括设置采集图像的格式，图像样本的大小等，限于篇幅，这里只介绍一些典型的寄存器，如表 14-2～表 14-5 所示，详细寄存器信息可查看 OV9650 芯片手册。

表 14-2 输出格式寄存器

地址	代号	默认值	读/写	描述
0x12	COM7	0x00	RW	通用控制寄存器 7 Bit[7]：SCCB 寄存器复位 0：无改变 1：复位所有寄存器为默认值 Bit[6]：输出格式 - VGA selection Bit[5]：输出格式 - CIF selection Bit[4]：输出格式 - QVGA selection Bit[3]：输出格式 - QCIF selection Bit[2]：Output format - RGB selection Bit[1]：Reserved Bit[0]：Output format - Raw RGB（COM7[2] must be set high）

表 14-3　系统时钟寄存器

地址	代号	默认值	读/写	描述
0E	COM5	0x01	RW	通用寄存器 5 Bit[7]:　系统时钟源序选择 Bit[6:5]: 保留 Bit[4]:　Slam 模式使能 0:主机模式 1: Slam 模式，用于从机模式 Bit[3:0]: 保留

表 14-4　时序信号配置寄存器

0x15	COM10	00	RW	通用控制 10 Bit[7]:　设置引脚定义 1: RESET to SLHS 及 PWDN to SLVS Bit[6]:　HREF 变为 HSYNC Bit[5]:　PCLK 输出选项 0: PCLK 总是输出 1: HREF 为低时 PCLK 不输出 Bit[4]:　PCLK 反转 Bit[3]:　HREF 反转 Bit[2]:　保留 Bit[1]:　VSYNC 反相 Bit[0]:　HSYNC 反相

表 14-5　通用控制寄存器

0x40	COM15	0xC0	RW	通用控制寄存器 l 15 Bit[7:6]: Data format - output full range enable 0x: Output range: [10] to [F0] 10: Output range: [01] to [FE] 11: Output range: [00] to [FF] Bit[5:4]: RGB 555/565 option (must set COM7[2] high) x0: Normal RGB output 01: RGB 565 11: RGB 555 Bit[3]:　Swap R/B in RGB565/RGB555 format Bit[2:0]: Reserved

14.2 SCCB 总线

14.2.1 SCCB 协议介绍

SCCB 是简化的 I2C 协议，专门为摄像传感器设计的通信总线，其中 SIO-l 是串行时钟输入线，SIO-O 是串行双向数据线，分别相当于 I2C 协议的 SCL 和 SDA，如图 14-2 所示。

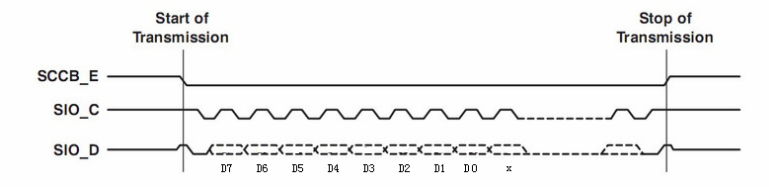

图 14-2 SCCB 协议时序图

SCCB 的总线时序与 I2C 基本相同，它的响应信号 ACK 被称为一个传输单元的第 9 位，分为 Don't care 和 NA。Don't care 位由从机产生，NA 位由主机产生，由于 SCCB 不支持多字节的读/写，NA 位必须为高电平。另外，SCCB 没有重复起始的概念，因此在 SCCB 的读周期中，当主机发送完片内寄存器地址后，必须发送总线停止条件。不然在发送读命令时，从机将不能产生 Don't care 响应信号。

由于 I2C 和 SCCB 的一些细微差别，所以采用 GPIO 模拟 SCCB 总线的方式。SCL 所连接的引脚始终设为输出方式，而 SDA 所连接的引脚在数据传输过程中，通过设置 IODIR 的值，动态改变引脚的输入/输出方式。SCCB 的写周期直接使用 I2C 总线协议的写周期时序；而 SC-CB 的读周期则增加一个总线停止条件。SCCB 是和 I2C 相同的一个协议。

SIO_C 和 SIO_D 分别为 SCCB 总线的时钟线和数据线。目前，SCCB 总线通信协议只支持 100kb/s 或 400kb/s 的传输速度，并且支持两种地址形式：

（1）从设备地址（ID Address，8bit），分为读地址和写地址，高 7 位用于选中芯片， 第 0 位是读/写控制位（R/W），决定是对该芯片进行读或写操作。

（2）内部寄存器单元地址（Sub_ Address，8bit）用于决定对内部的哪个寄存器单元进行操作，通常还支持地址单元连续的多字节顺序读/写操作。SCCB 控制总线功能的实现完全是依靠 SIO_C、SIO_D 两条总线上电平的状态及两者之间的相互配合实现的。SCCB 总线传输的启动和停止条件如图 14-2 所示：采用简单的三相（Phase）写数据的方式，即在写寄存器的过程中先发送 OV7649 的 ID 地址（ID Address），然后发送写数据的目的寄存器地址（Sub_address），最后发送要写入的数据（Write Data）。如果给连续的寄存器写数据，写完一个寄存器后，OV7649 会自动把寄存器地址加 1，程序可继续向下写，而不需要再次输入 ID 地址，从而三相写数据变为了两相写数据，由于本系统只需对有限个不连续寄存器进行配置，如果采用对全部寄存器都加以配置这一方法的话，会浪费很多时间和资源，所以我们只对需要更改数据的寄存器进行写数据。对于每一个需更改的寄存器，都采用三相写数据的方法。

14.2.2　SCCB 的总线编程

为了使读者进一步理解 SCCB 总线，并且掌握其编程方法和了解该协议和 IIC 协议之间的相同与不同点，因此，笔者在这里将给出 SCCB 的开发编程例子，借此能让读者深刻理解本小节的内容。

这里介绍的代码都是使用 GPIO 口模拟的协议，其完全按照 SCCB 所规定的协议来做，因此读者应该理解其代码的意义。

（1）SCCB 协议起始条件。

```
static void  SCCBStart(void)
{
 MAKE_HIGH(SIO_C);
 MAKE_HIGH(SIO_D);
 WAIT_STAB;
 MAKE_LOW(SIO_D);
 WAIT_STAB;
 MAKE_LOW(SIO_C);
 WAIT_STAB;
}
```

（2）SCCB 协议结束条件。

```
static void SCCBEnd(void)
{
 MAKE_LOW(SIO_D);
 WAIT_STAB;
 MAKE_HIGH(SIO_C);
 WAIT_STAB;
 MAKE_HIGH(SIO_D);
 WAIT_STAB;
}
```

（3）写一个位。

```
static void  sccb_write_bit(unsigned char bit)
{
 if (bit)
      MAKE_HIGH(SIO_D);
 else
      MAKE_LOW(SIO_D);
 WAIT_STAB;
 MAKE_HIGH(SIO_C);
 WAIT_CYL;
 MAKE_LOW(SIO_C);
 WAIT_STAB;
}
```

（4）读一个位。

```
staticintsccb_read_bit(void)
{
 inttmp = 0;
```

```
    MAKE_HIGH(SIO_C);
    WAIT_CYL;
    tmp = BIT_READ(SIO_D);
    MAKE_LOW(SIO_C);
    WAIT_STAB;
    returntmp;
}
```

（5）写一个字节。

```
static void  sccb_writechar(unsigned char data)
{
    inti = 0;
    /* data */
    for (i = 0; i< 8; i++ ) {
         sccb_write_bit(data & 0x80);
         data<<= 1;
    }
    /* 9th bit - Don't care */
    sccb_write_bit(1);
}
```

（6）读一个字节。

```
static void  sccb_readchar(unsigned char *val)
{
    inti;
    inttmp = 0;
    CFG_READ(SIO_D);
    for (i = 7; i>= 0; i--)
         tmp |= sccb_read_bit() <<i;

    CFG_WRITE(SIO_D);
    /* 9th bit - N.A. */
    sccb_write_bit(1);
    *val = tmp& 0xff;
}
```

14.3 CAMIF 接口详解

14.3.1 基于 S5PC100 的 CAMIF 接口介绍

1. 简介

现在我们来看一下 OV9650 是如何接入到 S5PC100 中的，当然，它不能是平白无故地就能和 S5PC100 通信，这里要介绍的就是 CAMIF 接口，如图 14-3 所示，该接口还有另一个名字 FIMC（Fully Interactive Mobile Camera Interface），在 Cortex-A8 中，它目前的版本是 4.0。这个接口支持 ITU RBT-601/656 标准，AXI 接口，MIPI 接口。最大输入图像尺寸

为 8192×8192 像素，S5PC100 有 3 个独立的摄像头接口单元，而且每个单元的功能是很强大的。包括时序产生器、DMA 通道、本地通道，以及图像处理单元等。

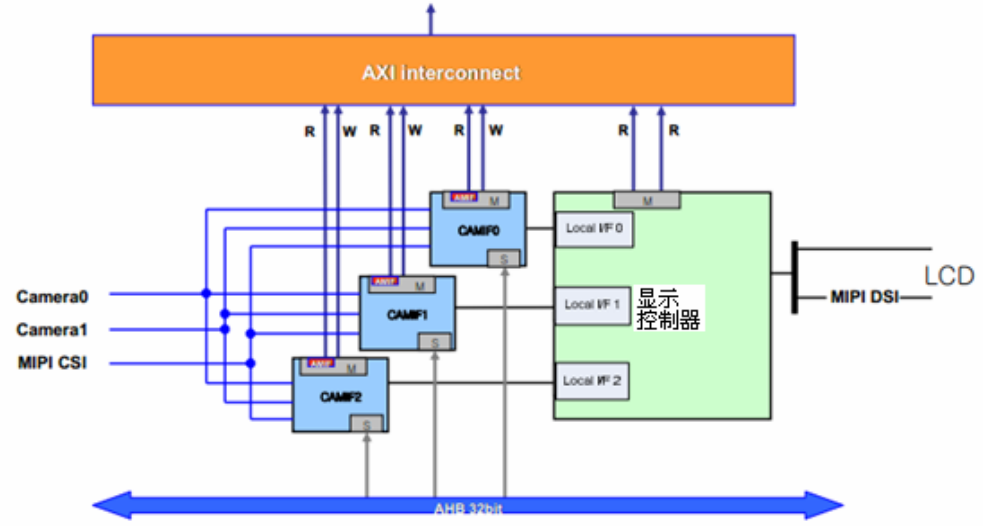

图 14-3　CAMIF

2．特点

CAMIF 的特点包括：

- 支持多种输入模式。
  - DMA（AXI 64 位接口）模式。
  - MIPI（CSI）模式。
- 支持多种输出模式。
  - DMA（AXI 64 位接口）模式。
  - 直接本地 FIFO 模式。
- 支持数字式缩放图像尺寸。
- 可编程的视频同步信号极性。
- 支持最大 8192×8192 的输入图像像素。
- 镜像翻转及旋转。
- 支持帧捕捉功能。

3．时钟

S5PC100 的 CAMIF 接口使用了 3 个时钟源，每个时钟源都是可配置的，就是说我们可以通过对寄存器的设置来选择想要的时钟源，如 ACLK、MCLK、ECLK、PCLK 都可以作为 CAMIF 的时钟源，下面简单介绍一下每个三个时钟源的特点，如图 14-4 所示。

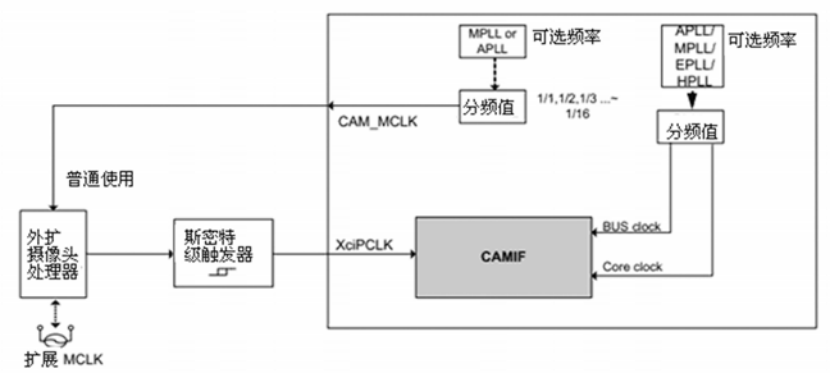

图 14-4　CAMIF 时钟源

CAM_MCLK 是 S5PC100 单独提供的一种时钟源，它可以在 clock 模块中找到相应的配置，因此在编程的时候，一定要记得去设置，CAM_MCLK 时钟源进来后首先分频，紧接着就传给 OV9650 光学传感器芯片，该芯片内部有一个内置的分频器，分频后的时钟即为芯片的工作时钟，此时，OV9650 芯片还会返回一个分频后的时钟给 CAMIF，该时钟类似于一种反馈信号，通知 CAMIF 什么时候去取数据，以及以什么样的时序去取。

而 BUS_CLK 和 CORE_CLK 时钟源是由 APB 总线接过来的时钟，它也需要单独去配置。不过这里要注意，CORE_CLK 时钟源的最大频率为 133MHz，CAM_MCLK 时钟源的最大频率 83MHz。

14.3.2　S5PC100 CAMIF 寄存器详解

S5PC100 CAMIF 寄存器如表 14-6~14-13 所示。

表 14-6　摄像头源格式寄存器（0XEE200000）

域	位	描述	复位值
ITU601_656n	[31]	1 = ITU-R BT.601　8 位模式使能 0 = ITU-R BT.656　8 位模式使能	0
UVOffset	[30]	Cb，Cr 值的偏移控制 1 = Cb=Cb+128 , Cr=Cr+128 0 = +0 (normally used)	0
Reserved	[29]	需要置 0	0
SrcHsize_CAM	[28:16]	图像源水平像素个数 （如果 WIN0 未使能，则该值为 PreHorRatio 的 4 倍，并且该值受外接扩展芯片寄存器的影响） SrcHsize_CAM_ext	0

续表

域	位	描述	复位值
Order422_CAM	[15:14]	摄像头输入 YCbCr 顺序 8 位模式的数据流： 00 = $Y_0Cb_0Y_1Cr_0$… 01 = $Y_0Cr_0Y_1Cb_0$… 10 = $Cb_0Y_0Cr_0Y_1$… 11 = $Cr_0Y_0Cb_0Y_1$…	0
SrcVsize_CAM	[13:0]	图像源垂直像素个数	0

表 14-7　全局控制寄存器（0XEE200008）

域	位	描述	复位值
SwRst	[31]	摄像头接口软复位	0
CamRst_A	[30]	外接摄像头处理器复位或掉电控制	0
SelCam_ITU	[29]	多个 ITU 摄像头选择 1 = ITU 摄像头 A 0 = ITU 摄像头 B	1
TestPattern	[28:27]	00 = 普通模式 01 = 色调测试方案 10 = 水平增加测试方案 11 = 垂直增加测试方案	0
InvPolPCLK	[26]	1 = PCLK 极性反转　0= 普通	0
InvPolVSYNC	[25]	1 = VSYNC 极性反转 0 =普通	0
InvPolHREF	[24]	1 = HREF 极性反转　0 =普通	0
Reserved	[23]	需要置 0	0
InvPolHSYNC	[4]	1 = HSYNC 极性反转 0 = 普通	0
SelCam_CAMIF	[3]	外接摄像头选择 1 = 选择 MIPI 摄像头 0 = 选择 ITU 摄像头	0
InvPolFIELD	[1]	1 = FIELD 极性反转　0 = 普通	0
Cam_Interlace	[0]	外接摄像头扫描方式 1 = 交错扫描　0 = 步进扫描	0

表 14-8　DMA 输出地址寄存器（0XEE200018）

域	位	描述	重置值
CIOYSA1	[31:0]	输出格式 : RGB → RGB 1st frame start address	0

注意：

CAMIF 一共有 4 个 RGB 格式的 DMA 输出地址寄存器。

表 14-9　目标格式寄存器（0XEE200048）

域	位	描述	重置值
InRot90	[31]	1 = 输入旋转 90° 0 = 忽略输入旋转	0
OutFormat	[30:29]	00 = YCbCr 4:2:0 格式 01 = YCbCr 4:2:2 格式 10 = YCbCr 4:2:2 格式 11 = RGB 格式	0
TargetHsize	[28:16]	目标图像的水平像素值	0
OutFlipMd	[15:14]	DMA 输出图像旋转及镜像处理 00 = 普通 01 = X 轴镜像 10 = Y 轴镜像 11 = 180° 旋转	0
OutRot90	[13]	1 = 输出旋转 90° 0 = 忽略输出旋转	0
TargetVsize	[12:0]	目标图像垂直像素值	0

表 14-10　预裁剪控制寄存器 1（0XEE200050）

域	位	描述	重置值
SHfactor	[31:28]	预裁剪偏移因子	0
Reserved	[27:23]	保留	0
PreHorRatio	[22:16]	水平比例	0
Reserved	[15:7]	保留	0
PreVerRatio	[6:0]	垂直比例	0

表 14-11　预裁剪控制寄存器 2 （0XEE200054）

域	位	描述	重置值
Reserved	[31:30]	保留	0
PreDstWidth	[29:16]	目标图像宽度	0
Reserved	[15:14]	保留	0
PreDstHeight	[13:0]	目标图像高度	0

表 14-12　主裁剪控制寄存器（0XEE200058）

域	位	描述	重置值
ScalerBypass	[31]	不采取裁剪模式： 在该模式下，ImgCptEn_SC 应当置 0，但 ImgCptEn 应当置 1	0
ScaleUp_H	[30]	水平比例放大/缩小 1:放大 0:缩小	0

续表

域	位	描述	重置值
ScaleUp_V	[29]	垂直比例放大/缩小 1:放大 0:缩小	0
MainHorRatio	[24:16]	水平裁剪比例	0
ScalerStart	[15]	裁剪使能位 1 = 裁剪开始 0 = 停止	0
InRGB_FMT	[14:13]	输入 RGB 格式： 00 = RGB565 01 = RGB666 10 = RGB888 11 = 保留	0
OutRGB_FMT	[12:11]	输出 RGB 格式： 00 = RGB565 01 = RGB666 10 = RGB888 11 = 保留	0
MainVerRatio	[8:0]	垂直裁剪比例	0

表 14-13　图像捕捉使能寄存器（0XEE2000C0）

域	位	描述	重置值
ImgCptEn	[31]	摄像头全局捕捉使能位	0
ImgCptEn_Sc	[30]	裁剪使能	0
Reserved	[29:26]	保留	0
Cpt_FrEn	[25]	捕捉帧控制 1 = 使能 0 = 禁止	0
Reserved	[24]	保留	0
Cpt_FrPtr	[23:19]	捕捉队列回转位置	0
Cpt_ FrCnt	[17:10]	想要捕捉的帧数量	0
Reserved	[9:0]	保留	0

14.3.3　CAMIF 操作案例

通过前文对包括 OV9650 芯片的介绍，以及如何通过 SCCB 对 OV9650 芯片的配置，再到 S5PC100 下 CAMIF 寄存器的详解，现在将通过一些实际操作代码来向读者展示如何实际操作 CAMIF 使其能驱动摄像头传感器，获得图像数据并显示到 LCD 上。限于篇幅，这里只列出部分代码，完整代码详见华清远见官方论坛。

1．操作流程

操作流程如图 14-5 所示（流程图需要根据 S5PC100 重新修改一下）

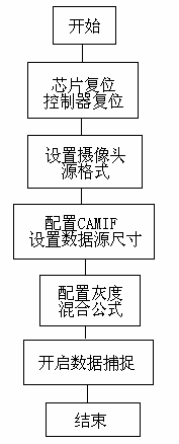

图 14-5　CAMIF 驱动流程

2．关键代码的实现

（1）CAMIF 复位及摄像头传感器芯片复位。

```
/*CAMIF 控制器复位*/
void CramIfreset()
{
 unsigned int cfg=0;
 cfg = readl(CISRCFMT);
 cfg |=(1<<31);
 writel(cfg,CISRCFMT);
 /* 软件复位*/
 cfg = readl(CIGCTRL0);
 cfg |= (1<<31);
 writel(cfg, CIGCTRL0);
 mdelay(1000);
 cfg = readl(CIGCTRL0);
 cfg &= ~(1<<31);
 writel(cfg, CIGCTRL0);
}
/*传感器软件复位*/
void SensorSftReset()
{
 unsigned int cfg=0;
 cfg |= (1<<30);
 writel(cfg,CIGCTRL0);
 mdelay(1000);
 cfg =0;
 cfg &= ~(1<<30);
 writel(cfg,CIGCTRL0);
}
```

（2）设置摄像头源格式。

```
static void SetCrmSrcFmt(struct CRAMIF *cram)
{
 unsigned long cfg =0;
 cfg|=(cram->srcHsize<<16)|(cram->srcVsize<<0)|(1<<31);
 writel(cfg,S5PCRMOFF(0x0));
 /*清除管道的溢出标志*/
 cfg = 0;
 cfg |= (1<<15)|(1<<14)|(1<<30)|(1<<29);
 writel(cfg,S5PCRMOFF(0x4));
 /*设置时序引脚的极性*/
 cfg = 0;
 cfg|=(1<<29)|(1<<7)|(cram->invPolPclk<<26)|(cram->invPolVSync<<25)|(cram->invPolRef<<2
4) |(cram->invPolHSync<<4)|(cram->invPolField<<1);
 writel(cfg,S5PCRMOFF(0x8));
 /*窗口的垂直以及水平偏移寄存器*/
 writel(0,S5PCRMOFF(0x14));
}
```

（3）配置 CAMIF 及设置数据源的尺寸。

```
static void SetCrmSize(struct CRAMIF *cram)
{
 unsigned long cfg =0;
 unsigned int SrcWidth = cram->srcHsize;
 unsigned int SrcHeight = cram->srcVsize;
 unsigned int PrDstWidth  = cram->tgtHsize;
 unsigned int PrDstHeight =   cram->tgtVsize;
 unsigned int H_Shift;
 unsigned int V_Shift;
 unsigned int PrHorRatio ;
 unsigned int PrVerRatio ;
 unsigned int MainHorRatio;
 unsigned int MainVerRatio;
 /*下面所使用的函数来自于 S5PC100 用户参考手册，看后面第 6 个模块*/
 CalculatePrescalerRatioShift(SrcWidth, PrDstWidth, &PrHorRatio,  &H_Shift);
 CalculatePrescalerRatioShift(SrcHeight, PrDstHeight, &PrVerRatio,  &V_Shift);
 MainHorRatio=(SrcWidth<<8)/(PrDstWidth<<H_Shift);
 MainVerRatio=(SrcHeight<<8)/(PrDstHeight<<V_Shift);
 cfg =0 ;
 cfg |=(3<<29)|(PrDstWidth<<16)|(PrDstHeight<<0) ;
 writel(cfg,S5PCRMOFF(0x48));
cfg = 0;
 cfg|=((10-(H_Shift+V_Shift))<<28)|(PrHorRatio<<16)|(PrVerRatio<<0);
 writel(cfg,S5PCRMOFF(0x50));
 cfg = 0;
 cfg |= (PrDstWidth<<16)|(PrDstHeight<<0);
 writel(cfg,S5PCRMOFF(0x54));
 cfg = 0;
 cfg |= (MainHorRatio<<16)|(MainVerRatio<<0);
 writel(cfg,S5PCRMOFF(0x58));
 writel(PrDstWidth*PrDstHeight,S5PCRMOFF(0x5c));
 cfg = 0;
```

```
    cfg |= (PrDstWidth<<16)|(PrDstHeight<<0);
    writel(cfg,S5PCRMOFF(0x184));
  }
```

（4）开启数据捕捉。

```
  void StartToCapture()
  {
    unsigned long cfg ;
    cfg =readl(S5PCRMOFF(0xC0));
    cfg |= (1<<31)|(1<<30);
    writel(cfg,S5PCRMOFF(0xc0));// 开启数据的捕捉

    cfg =readl(S5PCRMOFF(0x58));
    cfg |=(1<<15);
    writel(cfg,S5PCRMOFF(0x58));// 配置完成，开始正常工作
  }
```

（5）配置公式（参考 S5PC100 用户参考手册）。

```
  void CalculatePrescalerRatioShift(unsigned int SrcSize, unsigned int DstSize, unsigned int
    *ratio,unsigned int  *shift)
  {
        if(SrcSize>=64*DstSize) {
            printf("ERROR:  out  of  the  prescaler  range:  SrcSize/DstSize  =  %d(<
64)\n",SrcSize/DstSize);
            while(1);
        }
        else if(SrcSize>=32*DstSize) {
            *ratio=32;
            *shift=5;
        }
        else if(SrcSize>=16*DstSize) {
            *ratio=16;
            *shift=4;
        }
        else if(SrcSize>=8*DstSize) {
            *ratio=8;
            *shift=3;
        }
        else if(SrcSize>=4*DstSize) {
            *ratio=4;
            *shift=2;
        }
        else if(SrcSize>=2*DstSize) {
            *ratio=2;
            *shift=1;
        }
        else {
            *ratio=1;
            *shift=0;
        }
    }
```

实验步骤与测试结果如下。

（1）将摄像头模块插入开发板后，上电。

（2)将程序编译后产生.bin 可执行文件，然后使用 uboot 的 dnw 命令下载到 0x20008000 这个内存地址，使用 go 命令去执行，并观察结果。

（3）观察 LCD 显示屏，这时候会将 CAMERA 采集到的数据显示在 LCD 上。

14.4　本章小结

本章主要介绍了基于 S5PC100 的 CAMIF 控制器，以及 OV9650 芯片，SCCB 的操作方式。还介绍了通常摄像头开发的一般方法，希望借此能让读者掌握至少一种摄像头传感芯片的开发方法。

14.5　练习题

1．CAMIF 外部接口信号有哪些？

2．简述上题中列出的信号作用。

3．编程实现在 LCD 上显示摄像头采集到的数据。

第 15 章　SPI 接口

SPI 作为应用最为广泛的通信总线协议之一，开发人员应当掌握，本章将介绍 SPI 总线协议的基本理论，以及 S5PC100 的 SPI 总线控制器的操作方法。

本章要点：

- SPI 总线协议。
- S5PC100 下 SPI 总线控制器详解。

15.1　SPI 总线协议理论

15.1.1　协议简介

SPI 是英文 Serial Peripheral Interface 的缩写，该协议是由美国摩托罗拉公司推出的一种同步串行传输规范，首先由摩托罗拉公司在其 MC68HCXX 系列处理器上定义，后主要应用在 EEPROM、FLASH、实时时钟、AD 转换器，还有数字信号处理器和数字信号解码器之间。

SPI 是一种高速的全双工、同步的通信总线，并且在芯片的引脚上只占用四根线，节约了芯片的引脚，同时为 PCB 的布局上节省空间，提供方便，正是出于这种简单易用的特性，现在越来越多的芯片集成了这种通信协议。

15.1.2　协议内容

SPI 有 4 个引脚：CS（从器件选择线）、SDO（串行数据输出线）、SDI（串行数据输入线）和 SPICLK（同步串行时钟线）。

SPI 的通信原理很简单，它以主从方式工作，这种模式通常有一个主设备和一个或多个从设备，需要至少 4 根线，事实上 3 根也可以（单向传输时）。也是所有基于 SPI 的设备共有的，这些脚的定义如下：

（1）SDO（MOSI）——主设备数据输出，从设备数据输入。

（2）SDI（MISO）——主设备数据输入，从设备数据输出。

（3）SPICLK——时钟信号，由主设备产生。

（4）CS——从设备使能信号，由主设备控制。

其中 CS 是控制芯片是否被选中的，也就是说只有片选信号为预先规定的使能信号时（高电位或低电位），对此芯片的操作才有效。这就使在同一总线上连接多个 SPI 设备成为可能。其中总线协议时序如图 15-1 所示。

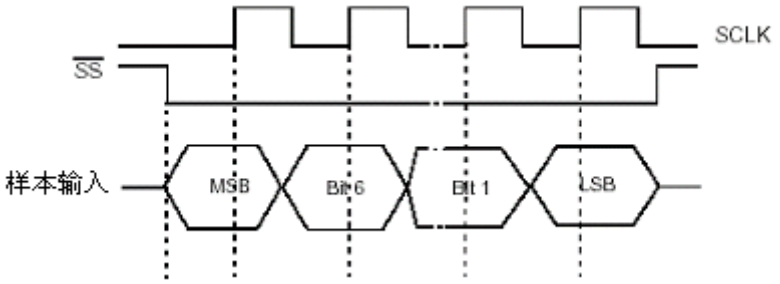

图 15-1　SPI 总线协议时序

接下来就是负责通信的 3 根线了。通信是通过数据交换完成的，这里先要知道 SPI 是串行通信协议，也就是说数据是一位一位地传输的。这就是 SPICLK 时钟线存在的原因，由 SPICLK 提供时钟脉冲，SDO、SDI 则基于此脉冲完成数据传输。数据输出通过 SDO 线，数据在时钟上升沿或下降沿时改变，在紧接着的下降沿或上升沿被读取，完成一位数据传输，输入也使用同样原理。这样，在至少 8 次时钟信号的改变 （上沿和下沿为一次）后，就可以完成 8 位数据的传输。

要注意的是，SPICLK 信号线只由主设备控制，从设备不能控制信号线。同样在一个基于 SPI 的设备中，至少有一个主控设备。这样传输的特点：这样的传输方式有一个优点，即其与普通的串行通信不同，普通的串行通信一次连续传送至少 8 位数据，而 SPI 允许数据一位一位地传送，甚至允许暂停，因为 SPICLK 时钟线由主控设备控制，当没有时钟跳变时，从设备不采集或传送数据。也就是说，主设备通过对 SPICLK 时钟线的控制可以完成对通信的控制。SPI 还是一个数据交换协议：因为 SPI 的数据输入和输出线独立，所以允许同时完成数据的输入和输出。不同的 SPI 设备的实现方式不尽相同，主要是数据改变和采集的时间不同，在时钟信号上沿或下沿采集有不同定义。

在点对点的通信中，SPI 接口不需要进行寻址操作，且为全双工通信，显得简单高效。在多个从设备的系统中，每个从设备需要独立地使能信号，在硬件上要比 I2C 总线控制稍微复杂一些。

注意：

SPI 的一个缺点是没有指定的流控制，没有应答机制确认是否接收到数据。

SPI 控制器为了和外设进行数据交换，根据外设工作要求，其输出串行同步时钟极性和相位可以进行配置，时钟极性（CPOL）对传输协议没有重大影响。如果 CPOL=0，串行同步时钟的空闲状态为低电平；如果 CPOL=1，串行同步时钟的空闲状态为高电平。时钟相位（CPHA）能够配置用于选择两种不同的传输协议之一进行数据传输。如果 CPHA=0，

在串行同步时钟的第一个跳变沿（上升或下降）数据被采样，如图 15-2 所示；如果 CPHA=1，在串行同步时钟的第二个跳变沿（上升或下降）数据被采样，如图 15-3 所示。SPI 主控制器和与之通信的外设时钟相位和极性应该一致，另外，上面提到这些特性将在寄存器中具体实现。

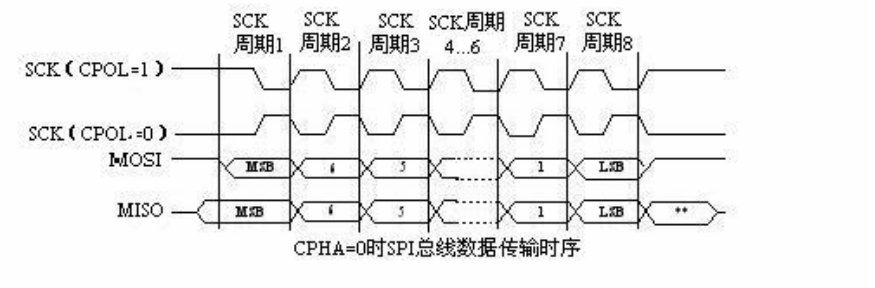

图 15-2　CPHA=0 时的情况

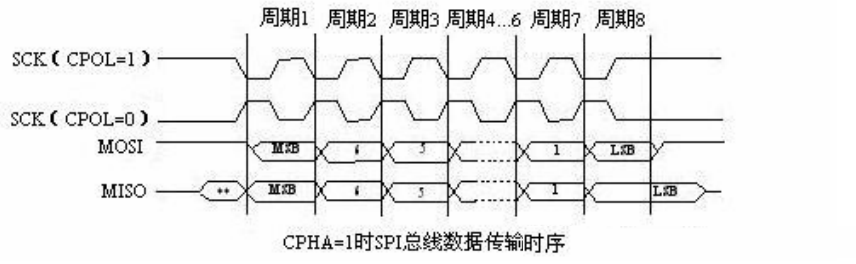

图 15-3　CPHA=1 时的情况

15.2　SPI 控制器详解

15.2.1　S5PC100 的 SPI 控制器简介

S5PC100 包含了两套 8 位、16 位、32 位移位寄存器用于收发。在 SPI 传输数据时，数据的发送及接收数据是同步的，该总线控制器支持摩托罗拉串行外设接口。

下面是该控制器的特性：

- 全双工通信方式。
- 8 位、16 位、32 位移位寄存器。
- 3 时钟源供应。
- 支持 8 位、16 位、32 位总线接口。
- 支持摩托罗拉 SPI 协议。
- 支持两个独立的传输及接收 FIFO。
- 支持主机模式及从机模式。

- 无法传送条件下接收。
- Tx/Rx 频率最大支持 50MHz。

15.2.2　时钟源控制

每个 SPI 都能获得不同的 3 个时钟源，用户可以根据自己的需要来进行配置，需要设置 CLK_CFG 这个寄存器，后面将会介绍。

时钟控制器如图 15-4 所示。

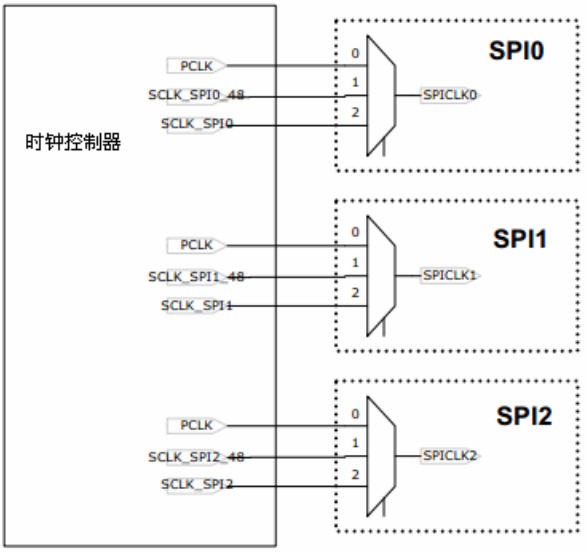

图 15-4　时钟控制器

15.2.3　寄存器详解

如表 15-1 所示为 SPI 配置寄存器。

表 15-1　SPI 配置寄存器

MODE_CFGn	位	描述	复位值
CH_WIDTH	[30:29]	00 = 字节　01 = 半字 10 = 字　11 = 保留	0
TRAILING_CNT	[28:19]	接收 FIFO 中最后写入字节的个数	0
BUS_WIDTH	[18:17]	00 = 字节　01 = 半字 10 = 字　11 = 保留	0

如表 15-2 所示为时钟配置寄存器。

表 15-2　时钟配置寄存器

MODE_CFGn	位	描述	复位值
CH_WIDTH	[30:29]	00 = 字节　　01 = 半字 10 = 字　　11 = 保留	0
TRAILING_CNT	[28:19]	接收 FIFO 中最后写入字节的个数	0
BUS_WIDTH	[18:17]	00 = 字节　　01 = 半字 10 = 字　　11 = 保留	0

如表 15-3 所示为 SPI 模式配置寄存器。

表 15-3　SPI 模式配置寄存器

CLK_CFGn	位	描述	复位值
SPI_CLKSEL	[10:9]	时钟源选择 00 = PCLK　　01 = SCLK_SPI_48 10 = SCLK_SPI　　11 = 保留	0
ENCLK	[8]	时钟使能 0 = 禁止　　1 = 使能	0
SPI_SCALER	[7:0]	SPI 时钟分频值 SPI 时钟输出 =时钟源 /（2 x（预分值 +1））	0

如表 15-4 所示为 SPI 数据发送寄存器。

表 15-4　SPI 数据发送寄存器

SPI_TX_DATAn	位	描述	复位值
TX_DATA	[31:0]	该寄存器包含了所要发送的数据	0

表 15-5 所示为 SPI 数据接收寄存器。

表 15-5　SPI 数据接收寄存器

SPI_RX_DATAn	位	描述	复位值
RX_DATA	[31:0]	该寄存器包含了所要接收的数据	0

如表 15-6 所示为 SPI 状态寄存器。

表 15-6　SPI 状态寄存器

SPI_STATUSn	位	描述	复位值
TX_DONE	[21]	0 = 其他情况 1 = 发送移位寄存器准备	0
RX_FIFO_LVL	[19:13]	RX FIFO 0 ~ 64 字节	0
TX_FIFO_LVL	[12:6]	TX FIFO 0 ~ 64 字节	0
RX_OVERRUN	[5]	Rx Fifo 溢出错误 0 =无误　　1 = 溢出错	0
RX_UNDERRUN	[4]	0 = 无误　　1 = 数据缺失	0

续表

SPI_STATUSn	位	描述	复位值
TX_OVERRUN	[3]	Tx Fifo 溢出错误 0 =无误　　1 = 溢出错	0
TX_UNDERRUN	[2]	0 = 无误　　1 = 数据缺失	0

15.3　SPI 开发例子

这里将介绍一种通过 SPI 通信的 Flash，该芯片是 M24PXX，如图 15-5 所示为该芯片的原理图，每根接线的意义已经清楚地标识出来了。

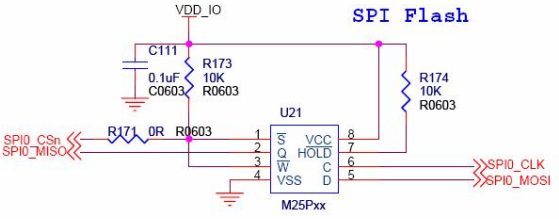

图 15-5　M25PXX 原理图

这一款芯片内部集成了 12 条指令，包括了通用的读、写、配置等命令，还有一个内置的状态寄存器，可以通过该寄存器获取芯片当前状态。

如表 15-7 所示为 M5DPXX 芯片指令集。

表 15-7　M25PXX 芯片指令集

Instrucion	Description	One-byle Instruction Code		Address Bytes	Dummy Bytes	Date Bytes
WREN	写使能	0000 0110	06h	0	0	0
WRDI	写禁止	0000 0100	04h	0	0	0
RDID	读取 ID	1001 1111	9Fh	0	0	1 to 3
RDSR	读状态寄存器	0000 0101	05h	0	0	1 to ∞
WRSR	写状态寄存器	0000 0001	01h	0	0	1
READ	字节数据读取	0000 0011	03h	3	0	1 to ∞
FAST_READ	高速的字节读取	0000 1011	0Bh	3	1	1 to ∞
PP	页编程	0000 0010	02h	3	0	1 to 256
SE	扇区擦除	1101 1000	D8h	3	0	0
BE	块擦除	1100 0111	C7h	0	0	0
DP	深度擦除	1011 1001	B9h	0	0	0
RES	从睡眠模式唤醒以及读出电特性	1010 1011	ABh	0	3	1 to ∞
	唤醒			0	0	0

有了上文的知识做铺垫，现在我们先来看一下 SPI 控制器的基本编程模型：

（1）设置时钟源并配置分频值等参数。

（2）软复位后，并设置 SPI 配置寄存器（SPI CONFIGURATION REGISTER）。

（3）设置模式寄存器。

（4）设置从机选择寄存器。

（5）收发数据。

根据以上信息，这里分成若干模块来逐一实现，全部代码可到华清远见官方论坛上下载。

（1）相关寄存器结构体定义及宏定义。

```
/*SPI 总线控制器寄存器定义*/
typedef struct {
                unsigned int CHCFG;
                unsigned int CLKCFG;
                unsigned int MODECFG;
                unsigned int SLAVESEL;
                unsigned int INTEN;
                unsigned int STATUS;
                unsigned int TXDATA;
                unsigned int RXDATA;
                unsigned int PACKETCNT;
                unsigned int PENDINGCLR;
                unsigned int SWAPCFG;
                unsigned int FBCLK;
}spi;
#define SPI0 ( * (volatile spi *)0XEC300000 )
/* Flash opcodes. */
#define    OPCODE_WREN        0x06 /*写使能*/
#define    OPCODE_WRDA        0x04 /*写禁止*/
#define    OPCODE_RDSR        0x05 /*读状态寄存器*/
#define    OPCODE_WRSR        0x01 /*写状态寄存器*/
#define    OPCODE_NORM_READ   0x03 /*低频率的数据读取*/
#define    OPCODE_FAST_READ   0x0b /*高频率的数据读取*/
#define OPCODE_PP             0x02 /*页编程*/
#define    OPCODE_BE_4K       0x20 /* Erase 4KiB block */
#define    OPCODE_BE_32K      0x52 /* Erase 32KiB block */
#define    OPCODE_CHIP_ERASE  0xc7 /*擦除整块芯片*/
#define    OPCODE_SE          0xd8 /*扇区擦除*/
#define    OPCODE_RDID        0x9f /*读 ID */

/*状态寄存器*/
#define    SR_WIP             1        /*写状态中*/
#define    SR_WEL             2        /*写保护锁*/
```

（2）延时函数，使能芯片及禁用芯片的实现如下。

```
void delay(int times)
{
 volatile int i,j;
 for (j = 0; j < times; j++){
     for (i = 0; i < 100000; i++);
```

```
            i = i + 1;
  }
}
void disable_chip(void)
{
  /* disable chip*/
  SPI0.SLAVESEL |= 0x1;
  delay(1);
}
void enable_chip(void)
{
  /* enable chip*/
  SPI0.SLAVESEL &= ~0x1;
  delay(1);
}
```

（3）软件复位的代码如下。

```
void soft_reset(void)
{
  SPI0.CHCFG |= 0x1 << 5;
  delay(1);
  SPI0.CHCFG &= ~(0x1 << 5);
}
```

（4）接收（字节读）的实现。在实现读字节功能的时候，第一次发送读指令后，紧接着芯片就会回送数据，这时芯片会自动累加地址，因此需要人为控制所读数据的范围，读时序如图 15-6 所示。

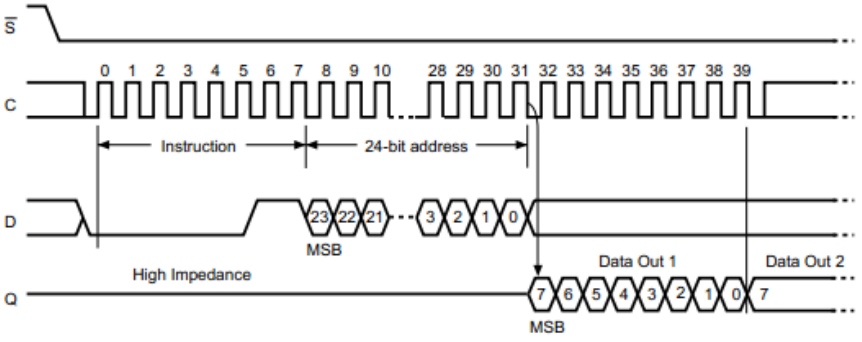

图 15-6　读时序

```
void receive(unsigned char *buf, int len)
{
  int i;
  SPI0.CHCFG &= ~0x1; // disable Tx
  SPI0.CHCFG |= 0x1 << 1; // enable Rx
  delay(1);
  for (i = 0; i < len; i++){
       buf[i] = SPI0.RXDATA;
       delay(1);
  }
```

```
    SPI0.CHCFG &= ~(0x1 << 1);
  }
```

（5）发送（写页面）的实现，这里芯片的页面为 256B。

在发出写指令后，要紧跟着发出第一个数据要存放的地址，这个时候再顺序写入数据，和读的时候一样，芯片会自动累加地址，写时序如图 15-7 所示。

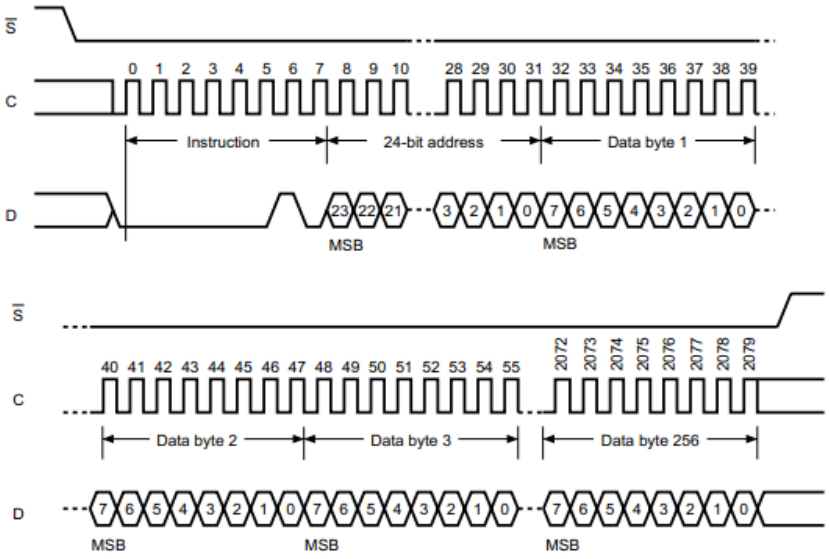

图 15-7　写时序

关键代码如下。

```
  void transfer(unsigned char *data, int len)
  {
    int i;
    SPI0.CHCFG &= ~(0x1 << 1);
    SPI0.CHCFG = SPI0.CHCFG | 0x1; // enable Tx and disable Rx
    delay(1);
    for (i = 0; i < len; i++){
        SPI0.TXDATA = data[i];
        while( !(SPI0.STATUS & (0x1 << 21)) );
        delay(1);
    }
    SPI0.CHCFG &= ~0x1;
  }
```

（6）擦除芯片相关操作如下。

这块芯片提供了两种擦除方式，第一种是分扇区来进行擦除，重点在于选好指定的地址，然后进行擦除，结合如图 15-8 所示的时序图。

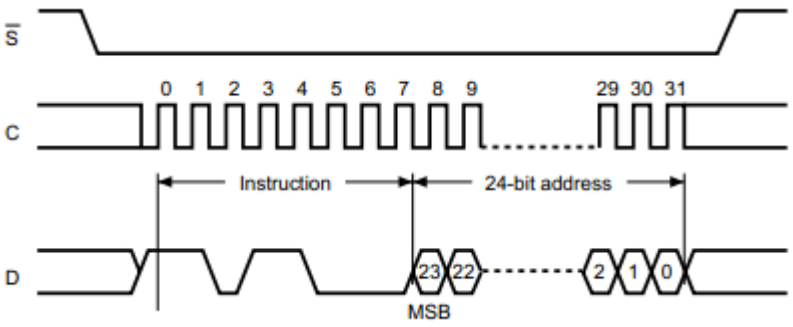

图 15-8　擦除扇区时序图

第二种是整片芯片擦除，如图 15-9 所示为整片芯片的擦除时的时序情况。

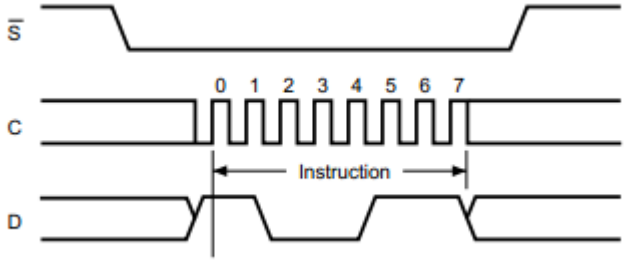

图 15-9　整片芯片擦除时序图

关键代码如下。

```
/*擦除扇区*/
void erase_sector(int addr)
{
 unsigned char buf[4];
 buf[0] = OPCODE_SE;
 buf[1] = addr >> 16;
 buf[2] = addr >> 8;
 buf[3] = addr;
 enable_chip();
 transfer(buf, 4);
 disable_chip();
}
/*擦除芯片*/
void erase_chip()
{
 unsigned char buf[4];
 buf[0] = OPCODE_CHIP_ERASE;
 enable_chip();
 transfer(buf, 1);
 disable_chip();
}
```

（7）解析状态寄存器用来获取芯片是否工作结束，如图 15-10 所示为状态寄存器读时序。

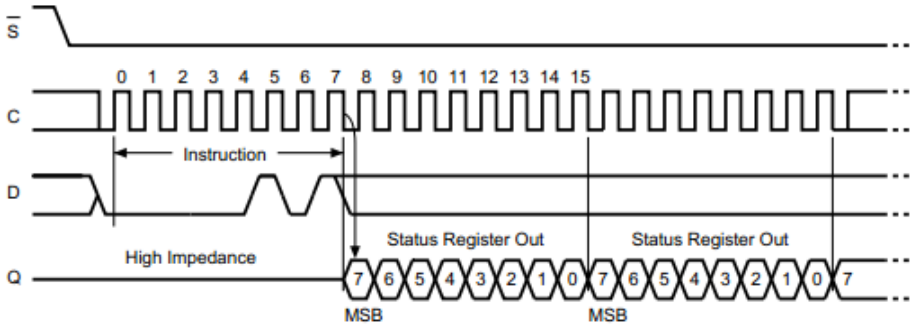

图 15-10　状态寄存器读时序

关键代码如下。

```
void wait_till_write_finished()
{
 unsigned char buf[1];
 enable_chip();
 buf[0] = OPCODE_RDSR;
 transfer(buf, 1);
 while(1) {
      receive(buf, 1);
      if(buf[0] & SR_WIP) {
      //   printf( "Write is still in progress\n" );
      }
      else {
           printf( "Write is finished.\n" );
           break;
      }
 }
 disable_chip();
}
```

（8）读芯片 ID，读 ID 时序如图 15-11 所示。

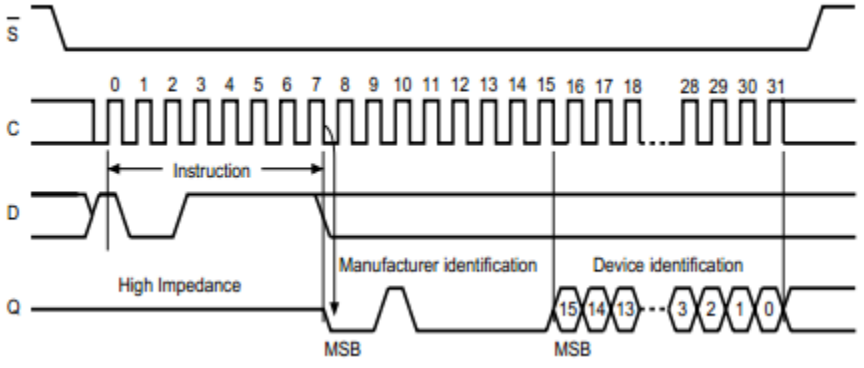

图 15-11　读 ID 时序

关键代码如下。

```
void read_ID(void)
{
 unsigned char buf[3];
 int i;

 buf[0] = OPCODE_RDID;
 soft_reset();
 enable_chip();
 transfer(buf, 1);
 receive(buf, 3);
 disable_chip();
 printf("MI = %x\tMT = %x\tMC = %x\t\n", buf[0], buf[1], buf[2]);
}
```

（9）主程序如下。

```
int main()
{
 unsigned char buf[10] = "home\n";
 unsigned char data[10] = "morning\n";
 uart0_init();
 printf("aaaaa \n");
 cfg_gpio();      //配置 SPI IO 功能
 set_clk();       //使能 SPI 控制器的时钟
 cfg_spi0();      //配置 SPI0 控制器
 while(1)
 {
      read_ID();             //读出 SPI Flash 的 ID 号
      write_spi(buf, 4, 0);  //向目标芯片 0 地址写入 4 个字节数据
      read_spi(data, 4, 0);  //从目标芯片 0 地址读出 4 个字节数据
      printf("read from spi :%s", data);
 }
 return 0;
}
```

实验调试过程与结果：

（1）将程序编译后生成.bin 文件，打开终端使用 uboot 的 dnw 命令通过 USB 线将.bin 文件下载到 0x20008000 这个地址，接着使用 go 命令去执行测试程序。

（2）可以看到如图 5-12 所示的测试结果。

```
Now, Waiting for DNW to transmit data
Download Done!! Download Address: 0x20008000, Download Filesize:0x19fc
Checksum is being calculated.
Checksum O.K.
SMDKC100 # go 20008000
## Starting application at 0x20008000 ...
open uart device ok !
aaaaa
MI = 20 MT = 20 MC = 11
Write is finished.
Write is finished.
read from spi :homeing
```

图 15-12　终端打印结果

15.4 本章小结

本章重点介绍了 SPI 总线协议及 SPI 总线控制器的基本编程方法，希望读者能取得完整代码并实际试验，完全掌握 SPI 总线是很有必要的。

15.5 练习题

1．SPI 总线和 I2C 总线的区别是什么？
2．请编写一个实现读/写 SPI Flash 功能的程序。

第 16 章　I2C 接口

为了使读者掌握常见的 I2C 总线，这一章将从理论到实际应用从头梳理一遍，目的在于给读者一个完整的概念，不仅在理论上掌握了解 I2C 总线，更要在实际运用中灵活使用。

本章要点：

- I2C 总线协议。
- S5PC100 的 I2C 控制器。

16.1　I2C 总线

16.1.1　I2C 总线介绍

I2C（Inter－Integrated Circuit）总线（也称 IIC 或 I^2C）是由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备，是微电子通信控制领域广泛采用的一种总线标准。它是同步通信的一种特殊形式，具有接口线少，控制方式简单，器件封装形式小，通信速率较高等优点。I2C 有着如下的特点。

- 两条总线线路：一条串行数据线 SDA，一条串行时钟线 SCL。
- 每个连接到总线的器件都可以通过唯一的地址联系主机，同时主机可以作为主机发送器或主机接收器。
- 它是一个真正的多主机总线，如果两个或更多主机同时初始化，数据传输可以通过冲突检测和仲裁防止数据被破坏。
- 串行的 8 位双向数据传输位速率在标准模式下可达 100kb/s，快速模式下可达 400kb/s，高速模式下可达 3.4Mb/s。
- 连接到相同总线的 IC 数量只受到总线的最大电容 400pF 限制。

16.1.2　I2C 总线术语

- 发送器：发送数据到总线的器件。
- 接收器：从总线接收数据的器件。
- 主机：初始化发送产生时钟信号和终止发送的器件。
- 从机：被主机寻址的器件。
- 多主机：同时有多于一个主机尝试控制总线但不破坏传输。

- 仲裁：是一个在有多个主机同时尝试控制总线但只允许其中一个控制总线并使传输不被破坏的过程。
- 同步：两个或多个器件同步时钟信号的过程。

16.1.3 I2C 总线位传输

由于连接到 I2C 总线的器件有不同种类的工艺（CMOS、NMOS、双极性），逻辑 0（低）和逻辑 1（高）的电平不是固定的，它由电源 VCC 的相关电平决定，每传输一个数据位就产生一个时钟脉冲，数据有效性如图 16-1 所示。

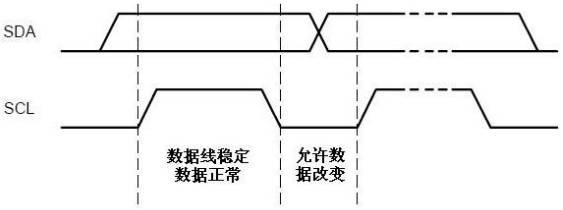

图 16-1 数据有效性

SDA 线上的数据必须在时钟的高电平周期保持稳定。数据线的高或低电平状态 I2C 位传输数据有效性在 SCL 线的时钟信号是低电平时才能改变，起始和停止条件如图 16-2 所示。

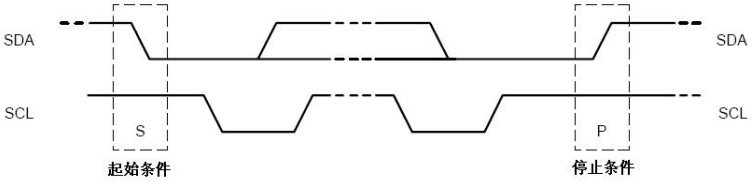

图 16-2 起始和停止条件

SCL 线是高电平时，SDA 线从高电平向低电平切换，这个情况表示起始条件；SCL 线是高电平时，SDA 线由低电平向高电平切换，这个情况表示停止条件。起始和停止条件一般由主机产生，总线在起始条件后被认为处于忙状态，在停止条件的某段时间后总线被认为再次处于空闲状态。如果产生重复起始条件而不产生停止条件，总线会一直处于忙的状态，此时的起始条件（S）和重复起始条件（Sr） 在功能上是一样的。

16.1.4 I2C 总线数据传输

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响应位。首先传输的是数据的最高位（MSB），如果从机要完成一些其他功能后（如一个内部中断服务程序）才能接收或发送下一个完整的数据字节，可以使

时钟线 SCL 保持低电平，迫使主机进入等待状态，当从机准备好接收下一个数据字节并释放时钟线 SCL 后数据传输继续。

2．应答响应

应答响应如图 16-3 所示。

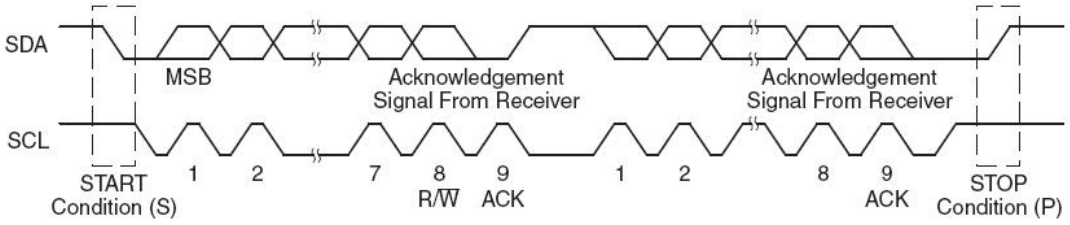

图 16-3 应答响应

数据传输必须带响应，相关的响应时钟脉冲由主机产生。在响应的时钟脉冲期间发送器释放 SDA 线（高）。在响应的时钟脉冲期间，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。

通常被寻址的接收器在接收到每个字节后，会产生一个响应。当从机不能响应从机地址时（如它正在执行一些实时函数不能接收或发送），从机必须使数据线保持高电平，主机然后产生一个停止条件终止传输或者产生重复起始条件开始新的传输。

如果从机接收器响应了从机地址，但是在传输了一段时间后不能接收更多数据字节，主机必须再一次终止传输。这个情况用从机在第一个字节后没有产生响应来表示。从机使数据线保持高电平，主机产生一个停止或重复起始条件。

如果传输中有主机接收器，它必须在从机不产生时钟的最后一个字节不产生响应，向从机发送器通知数据结束。从机发送器必须释放数据线，允许主机产生一个停止或重复起始条件。

16.1.5　I2C 总线寻址方式

1．7 位寻址

第一个字节的头 7 位组成了从机地址，最低位（LSB）是第 8 位，它决定了普通的和带重复开始条件的 7 位地址格式方向。第一个字节的最低位是“0”，表示主机会写信息到被选中的从机；“1”表示主机会向从机读信息，当发送了一个地址后，系统中的每个器件都在起始条件后将头 7 位与它自己的地址比较，如果一样，器件会判定它被主机寻址，至于是从机接收器还是从机发送器，都由 R/W 位决定。

2．10 位寻址

10 位寻址和 7 位寻址兼容，而且可以结合使用。10 位寻址采用了保留的 1111XXX 作为起始条件，或重复起始条件的后第一个字节的头 7 位。10 位寻址不会影响已有的 7 位寻址，有 7 位和 10 位地址的器件可以连接 I2C 总线 10 位地址格式到相同的 I2C 总线。它们都能用于标准模式和高速模式系统。

10 位从机地址由在起始条件或重复起始条件后的头两个字节组成。第一个字节的头 7 位是 11110XX 的组合，其中最后两位 XX 是 10 位地址的两个最高位（MSB）。第一个字节的第 8 位是 R/W 位，决定了传输的方向，第一个字节的最低位是“0”，表示主机将写信息到选中的从机，“1 ”表示主机将向从机读信息。如果 R/W 位是“0 ”，则第二个字节是 10 位从机地址剩下的 8 位；如果 R/W 位是“1”，则下一个字节是从机发送给主机的数据。

16.1.6 快速和高速模式

1．快速模式

快速模式器件可以在 400kb/s 下接收和发送。最小要求是：它们可以和 400kb/s 传输同步，可以延长 SCL 信号的低电平周期来减慢传输。快速模式器件都向下兼容，可以和标准模式器件在 0~100kb/s 的 I2C 总线系统通信。但是，由于标准模式器件不向上兼容，所以不能在快速模式 I2C 总线系统中工作。快速模式 I2C 总线规范与标准模式相比有以下另外的特征：

（1）最大位速率增加到 400kb/s。

（2）调整了串行数据（SDA） 和串行时钟（SCL ）信号的时序。

（3）快速模式器件的输入有抑制毛刺的功能，SDA 和 SCL 输入有施密特触发器。

（4）快速模式器件的输出缓冲器对 SDA 和 SCL 信号的下降沿有斜率控制功能。

（5）如果快速模式器件的电源电压被关断，SDA 和 SCL 的 I/O 引脚必须悬空，不能阻塞总线。

（6）连接到总线的外部上拉器件必须调整以适应快速模式 I2C 总线更短的最大允许上升时间。对于负载最大是 200pF 的总线，每条总线的上拉器件可以是一个电阻，对于负载在 200~400pF 之间的总线，上拉器件可以是一个电流源（最大值 3mA ）或者是一个开关电阻电路。

2．高速模式

高速模式（Hs 模式）器件对 I2C 总线的传输速度有很大的突破。高速模式器件可以在高达 3.4Mb/s 的位速率下传输信息，而且保持完全向下兼容快速模式或标准模式器件，它们可以在一个速度混合的总线系统中双向通信。

高速模式传输除了不执行仲裁和时钟同步外，与快速模式系统有相同的串行总线协议和数据格式。

16.2　I2C 总线控制器

16.2.1　S5PC100 下的 I2C 控制器介绍

S5PC100 处理器支持多主机 I2C 串行总线接口，并且它支持主机发送模式、主机接收模式、从机发送模式和从机接收模式这 4 种模式，如图 16-4 所示为 I2C 总线的概括图。

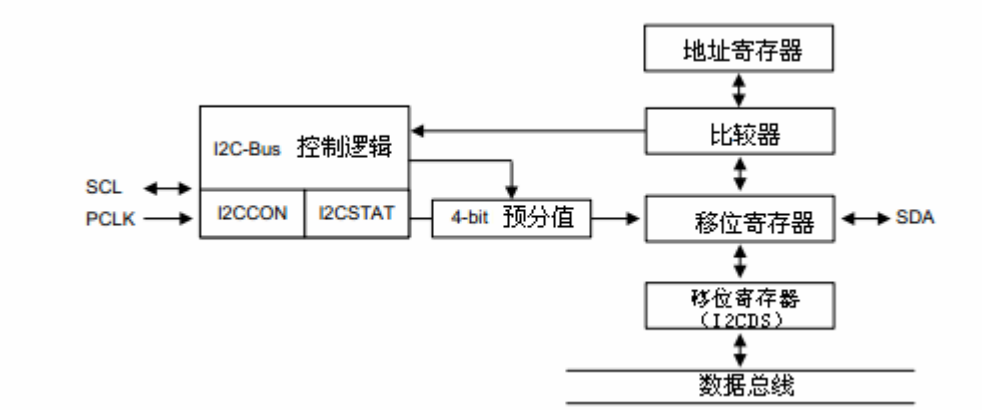

图 16-4　I2C 总线的概括图

16.2.2　I2C 总线控制寄存器详解

如表 16-1 所示为 I2C 总线控制寄存器描述。

表 16-1　I2C 总线控制寄存器

I2CCON	位	描述	复位值
应答产生	[7]	IIC 应答产生使能位 0 = 禁止　1 = 使能	0
Tx 时钟源选择	[6]	IIC 传输时钟预分值选择位 0 = I2CCLK = fPCLK /16 1= I2CCLK = fPCLK /512	0
Tx/Rx 中断	[5]	I C-Bus Tx/Rx 中断控制位 0 = 禁止　1 = 使能	0
传输时钟值	[3:0]	IIC 总线时钟预分值 Tx clock = I2CCLK/（I2CCON[3:0]+1）	未定义

如表 16-2 所示为 I2C 状态寄存器描述。

表 16-2　I2C 状态寄存器

I2CSTAT	位	描述	复位值
模式选择	[7:6]	IIC 总线 主/从 Tx/Rx 模式选择位 00 = 从接收模式 01 = 从发送模式 10 = 主接收模式 11 = 主发送模式	00
忙信号状态位	[5]	IIC 总线忙信号状态位 读：0 = 准备 1 = 忙 写： 产生开启信号	0
串行输出	[4]	IIC 总线数据输出使能/禁止位 0 = 禁止 Rx/Tx 1 = 使能 Rx/Tx	0
Ar 位定量状态标志	[3]	0 = 定量成功 1 = 失效	0
从地址状态标志	[2]	0 = 当开启/停止条件侦测到时清除 1 = 接收到的从地址匹配 I2CADD 的地址值	0
地址 0 状态标志	[1]	0 = 当开启/停止条件侦测到时清除 1= 接收从地址值为 00000000b	0
最后的接收位状态标志	[0]	0 = 为 0 1 = 为 1	0

如表 16-3 所示为 I2C 数据发送/接收移位寄存器描述。

表 16-3　I2C 数据发送/接收移位寄存器

I2CDS	位	描述	复位值
数据移位	[7:0]	8-位数据移位寄存器 如果串行输出使能，则 I2CDS 将变为可写。并且 I2CDS 任何时刻都是可读的，不管当前 I2CSTAT 的设置	未定义

16.3　I2C 总线的实际应用

和 SPI 一样，I2C 的实际应用需要加入 LM75 的介绍、操作流程图、关键代码。

16.3.1　应用分析

现在结合上面已经提到的 I2C 理论基础，我们将以一个例子来进行实际讲解，用 I2C 来操作 LM75 温度传感器。

如图 16-15 所示为 LM75 的原理图。

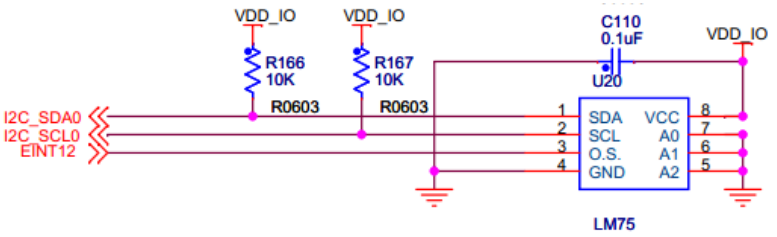

图 16-5　LM75 原理图

可以看到 SDA/SCL 被接到了 S5PC100 的 IIC 控制器上，并且接了一个外部中断，该中断可作为从机应答信号。

下面简单介绍一下 LM75 的操作时序，如图 16-6 所示。

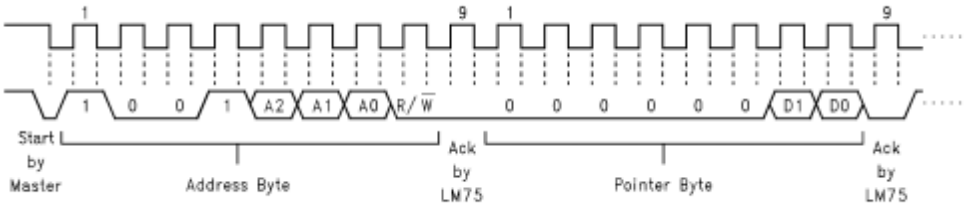

图 16-6　LM75 操作时序第一部分

如图 16-6 所示显示了 LM75 操作时序的第一阶段，可以看到，如果要获取数据，需要先配置一下模式，并且 LM75 的从机地址为 0x90，发送地址后要做的就是配置工作模式，LM75 芯片提供了 4 种模式：

（1）温度（只读模式）。

（2）配置（读/写）。

（3）T（HYST 读/写）。

（4）T（OS 读/写）。

这里选择第一个即可，也就是发送 0x0，接着有如图 16-7 所示的时序。

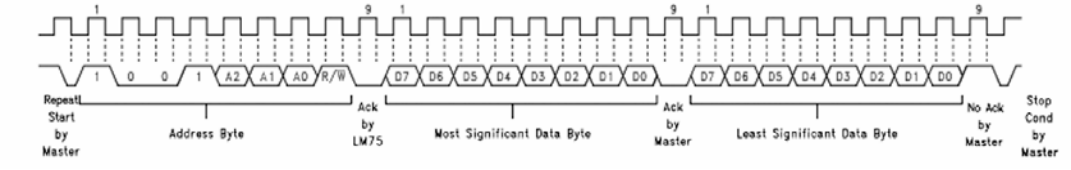

图 16-7　时序第二部分

接下来再次发送从机地址，选择 LM75 芯片后，即可等待芯片回送数据，这时芯片会发送给主机端两次数据，第一次是主要值，第二次是小数部分，最小能精确到 0.5。要注意的是每一次都要进行应答，才能保证数据的有效性。

16.3.2 代码实现

以下是核心代码（完整代码请到华清远见官方论坛上下载）。

```
/*IIC 寄存器结构体定义*/
typedef struct {
                unsigned int I2CCON0;
                unsigned int I2CSTAT0;
                unsigned int I2CADDR0;
                unsigned int I2CDS0;
                unsigned int I2CLC0;
}i2c0_type;
#define  I2C0 (* (volatile i2c0_type *)0xEC100000 )

/*设置 GPIO*/
void cfg_gpio(void)
{
GPD.GPDCON=(GPD.GPDCON&(~((0X0f<<12)|(0x0f<<16))))+((2<<12) | (2<<16)) ;

}

/*延时函数*/
void delay()
{
for(delay=0; delay<0x1fffff; delay++);
}
/*读模式，时序中的第一部分实现代码*/
int read_data_one()
{
 I2C0.I2CDS0 = 0x90; /*LM75 SLAVE ADDRESS */
 I2C0.I2CSTAT0=0xf0; /*Master Trans mode ,START ,ENABLE RX/TX ,*/
 I2C0.I2CCON0=0xef;/*ENABLE ACK BIT, PRESCALER:512 ,RX/TX INTERRUPT ENABLE ,*/
 while(!(I2C0.I2CCON0&(1<<4)));

 I2C0.I2CDS0 = mode;  // READ TEMPERATURE ONLY
 I2C0.I2CCON0 = 0xef;
 while(!(I2C0.I2CCON0&(1<<4)));
 delay();

 return 0;
}
/*读模式，时序中的第二部分实现代码*/
int read_data_two()
{
 int temp;
 int low, high;

 I2C0.I2CDS0 = 0x90;
```

```
    I2C0.I2CSTAT0 =0xb0;
    I2C0.I2CCON0 = 0xef;
    while(!(I2C0.I2CCON0&(1<<4)));

    I2C0.I2CCON0 = 0xef;
         delay();
    high = I2C0.I2CDS0;  //get the data from lm75 chip
  I2C0.I2CCON0 = 0x2f;
    delay();
    low = I2C0.I2CDS0;
    I2C0.I2CSTAT0 = 0x90;
    I2C0.I2CCON0 = 0xef;

    return ((high << 8) | low);
  }

  int main()
  {
    volatile int delay;
    int low, high, temp, config, i;
     uart0_init();
    cfg_gpio();
    /*循环打印采集的数据*/
    while (1){
        read_data_one();              // 配置模式
          temp = read_data_two();     // 开始连续两次读数据
        high = temp >> 8;
          low = temp & 0xff;
  printf("TEMP is : %d.%d\n", high, (((low>>7)==0) ? 0 : 5));
    }

    return 0;
  }
```

实验调试过程与结果：

（1）将程序编译后生成.bin 文件，打开终端使用 uboot 的 dnw 命令通过 USB 线将.bin 文件下载到 0x20008000 这个地址，接着使用 go 命令去执行测试程序。

（2）可以看到如图 16-8 所示的测试结果。

```
TEMP is : 22.5
TEMP is : 22.5
TEMP is : 23.0
TEMP is : 23.0
TEMP is : 23.0
TEMP is : 23.5
TEMP is : 23.5
TEMP is : 23.5
TEMP is : 23.5
TEMP is : 23.5
TEMP is : 23.5
```

图 16-8　测试结果

16.4 本章小结

本章从 I2C 总线协议的基本理论，到 S5PC100 控制器介绍及相关寄存器详解，最后再以 I2C 操作 LM75 温度传感器的应用来结尾，希望读者能深刻掌握 I2C 总线协议。

16.5 练习题

1．请解释一下 I2C 总线的时序图。
2．请举一下实现 I2C 操作 LM75 温度传感器的例子。
3．请讲讲 I2C 总线的特点和缺点。

参考文献

[1] 李佳. ARM 系列处理器应用技术完全手册[M]. 北京：人民邮电出版社，2006.

[2] 刘洪涛. 嵌入式系统技术与设计[M]. 北京：人民邮电出版社，2008

[3] 孙纪坤. 嵌入式 Linux 系统开发技术详解——基于 ARM[M]. 北京:人民邮电出版社，2006.

[4] 田泽. ARM9 嵌入式开发实验与实践[M]. 北京：北京航空航天大学出版社，2006.

[5] 孙天泽. 嵌入式设计及 Linux 驱动开发指南——基于 ARM9 处理器[M]. 北京：电子工业出版社，2007.

[6] 刘洪涛. ARM 体系结构与接口技术[M]. 北京：电子工业出版社，2012.