目录

第 1 章 ARM 微处理器概述

本章简介 ARM 微处理器的一些基本概念、应用领域及特点，引导读者进入 ARM 技术的殿堂。

本章主要内容：

- ARM 及相关技术简介
- ARM 微处理器的应用领域及特点
- ARM 微处理器系列
- ARM 微处理器的体系结构
- ARM 微处理器的应用选型

1.1 ARM－Advanced RISC Machines

ARM（Advanced RISC Machines），既可以认为是一个公司的名字，也可以认为是对一类微处理器的通称，还可以认为是一种技术的名字。

1991 年 ARM 公司成立于英国剑桥，主要出售芯片设计技术的授权。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 75％以上的市场份额，ARM 技术正在逐步渗入到我们生活的各个方面。

ARM 公司是专门从事基于 RISC 技术芯片设计开发的公司，作为知识产权供应商，本身不直接从事芯片生产，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。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，因此既使得 ARM 技术获得更多的第三方工具、制造、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被消费者所接受，更具有竞争力。

1.2 ARM 微处理器的应用领域及特点

1.2.1 ARM 微处理器的应用领域

到目前为止，ARM 微处理器及技术的应用几乎已经深入到各个领域：

1、工业控制领域：作为 32 的 RISC 架构，基于 ARM 核的微控制器芯片不但占据了高端微控制器市场的大部分市场份额，同时也逐渐向低端微控制器应用领域扩展，ARM 微控制器的低功耗、高性价比，向传统的 8 位/16 位微控制器提出了挑战。

2、无线通讯领域：目前已有超过 85%的无线通讯设备采用了 ARM 技术， ARM 以其高性能和低成本，在该领域的地位日益巩固。

3、网络应用：随着宽带技术的推广，采用 ARM 技术的 ADSL 芯片正逐步获得竞争优势。此外，ARM 在语音及视频处理上行了优化，并获得广泛支持，也对 DSP 的应用领域提出了挑战。

4、消费类电子产品：ARM 技术在目前流行的数字音频播放器、数字机顶盒和游戏机中得到广泛采用。

5、成像和安全产品：现在流行的数码相机和打印机中绝大部分采用 ARM 技术。手机中的 32 位 SIM 智能卡也采用了 ARM 技术。

除此以外，ARM 微处理器及技术还应用到许多不同的领域，并会在将来取得更加广泛的应用。

1.2.2 ARM 微处理器的特点

采用 RISC 架构的 ARM 微处理器一般具有如下特点：

1、体积小、低功耗、低成本、高性能；

2、支持 Thumb（16 位）/ARM（32 位）双指令集，能很好的兼容 8 位/16 位器件；

3、大量使用寄存器，指令执行速度更快；

4、大多数数据操作都在寄存器中完成；

5、寻址方式灵活简单，执行效率高；

6、指令长度固定；

1.3 ARM 微处理器系列

ARM 微处理器目前包括下面几个系列，以及其它厂商基于 ARM 体系结构的处理器，除了具有 ARM 体系结构的共同特点以外，每一个系列的 ARM 微处理器都有各自的特点和应用领域。

- ARM7 系列
- ARM9 系列
- ARM9E 系列
- ARM10E 系列
- SecurCore 系列
- Inter 的 Xscale
- Inter 的 StrongARM

其中，ARM7、ARM9、ARM9E 和 ARM10 为 4 个通用处理器系列，每一个系列提供一套相对独特的性能来满足不同应用领域的需求。SecurCore 系列专门为安全要求较高的应用而设计。

以下我们来详细了解一下各种处理器的特点及应用领域。

1.3.1 ARM7 微处理器系列

ARM7 系列微处理器为低功耗的 32 位 RISC 处理器，最适合用于对价位和功耗要求较高的消费类应用。ARM7 微处理器系列具有如下特点：

- 具有嵌入式 ICE－RT 逻辑，调试开发方便。
- 极低的功耗，适合对功耗要求较高的应用，如便携式产品。
- 能够提供 0.9MIPS/MHz 的三级流水线结构。
- 代码密度高并兼容 16 位的 Thumb 指令集。
- 对操作系统的支持广泛，包括 Windows CE、Linux、Palm OS 等。
- 指令系统与 ARM9 系列、ARM9E 系列和 ARM10E 系列兼容，便于用户的产品升级换代。
- 主频最高可达 130MIPS，高速的运算处理能力能胜任绝大多数的复杂应用。

ARM7 系列微处理器的主要应用领域为：工业控制、Internet 设备、网络和调制解调器设备、移动电话等多种多媒体和嵌入式应用。

ARM7 系列微处理器包括如下几种类型的核：ARM7TDMI、ARM7TDMI-S、ARM720T、ARM7EJ。其中，ARM7TMDI 是目前使用最广泛的 32 位嵌入式 RISC 处理器，属低端 ARM 处理器核。TDMI 的基本含义为：

T： 支持 16 为压缩指令集 Thumb；

D： 支持片上 Debug；

M：内嵌硬件乘法器（Multiplier）

I： 嵌入式 ICE，支持片上断点和调试点；

本书所介绍的 Samsung 公司的 S3C4510B 即属于该系列的处理器。

1.3.2 ARM9 微处理器系列

ARM9 系列微处理器在高性能和低功耗特性方面提供最佳的性能。具有以下特点：

- 5 级整数流水线，指令执行效率更高。
- 提供 1.1MIPS/MHz 的哈佛结构。
- 支持 32 位 ARM 指令集和 16 位 Thumb 指令集。
- 支持 32 位的高速 AMBA 总线接口。
- 全性能的 MMU，支持 Windows CE、Linux、Palm OS 等多种主流嵌入式操作系统。
- MPU 支持实时操作系统。
- 支持数据 Cache 和指令 Cache，具有更高的指令和数据处理能力。

ARM9 系列微处理器主要应用于无线设备、仪器仪表、安全系统、机顶盒、高端打印机、数字照相机和数字摄像机等。

ARM9 系列微处理器包含 ARM920T、ARM922T 和 ARM940T 三种类型，以适用于不同的应用场合。

1.3.3 ARM9E 微处理器系列

ARM9E 系列微处理器为可综合处理器，使用单一的处理器内核提供了微控制器、DSP、Java 应用系统的解决方案，极大的减少了芯片的面积和系统的复杂程度。ARM9E 系列微处理器提供了增强的 DSP 处理能力，很适合于那些需要同时使用 DSP 和微控制器的应用场合。

ARM9E 系列微处理器的主要特点如下：

- 支持 DSP 指令集，适合于需要高速数字信号处理的场合。
- 5 级整数流水线，指令执行效率更高。
- 支持 32 位 ARM 指令集和 16 位 Thumb 指令集。
- 支持 32 位的高速 AMBA 总线接口。
- 支持 VFP9 浮点处理协处理器。
- 全性能的 MMU，支持 Windows CE、Linux、Palm OS 等多种主流嵌入式操作系统。
- MPU 支持实时操作系统。
- 支持数据 Cache 和指令 Cache，具有更高的指令和数据处理能力。
- 主频最高可达 300MIPS。

ARM9 系列微处理器主要应用于下一代无线设备、数字消费品、成像设备、工业控制、存储设备和网络设备等领域。

ARM9E 系列微处理器包含 ARM926EJ-S、ARM946E-S 和 ARM966E-S 三种类型，以适用于不同的应用场合。

1.3.4 ARM10E 微处理器系列

ARM10E 系列微处理器具有高性能、低功耗的特点，由于采用了新的体系结构，与同等的 ARM9 器件相比较，在同样的时钟频率下，性能提高了近 50%，同时，ARM10E 系列微处理器采用了两种先进的节能方式，使其功耗极低。

ARM10E 系列微处理器的主要特点如下：

- 支持 DSP 指令集，适合于需要高速数字信号处理的场合。
- 6 级整数流水线，指令执行效率更高。
- 支持 32 位 ARM 指令集和 16 位 Thumb 指令集。

- 支持 32 位的高速 AMBA 总线接口。
- 支持 VFP10 浮点处理协处理器。
- 全性能的 MMU，支持 Windows CE、Linux、Palm OS 等多种主流嵌入式操作系统。
- 支持数据 Cache 和指令 Cache，具有更高的指令和数据处理能力
- 主频最高可达 400MIPS。
- 内嵌并行读/写操作部件。

ARM10E 系列微处理器主要应用于下一代无线设备、数字消费品、成像设备、工业控制、通信和信息系统等领域。

ARM10E 系列微处理器包含 ARM1020E、ARM1022E 和 ARM1026EJ-S 三种类型，以适用于不同的应用场合。

1.3.5 SecurCore 微处理器系列

SecurCore 系列微处理器专为安全需要而设计，提供了完善的 32 位 RISC 技术的安全解决方案，因此，SecurCore 系列微处理器除了具有 ARM 体系结构的低功耗、高性能的特点外，还具有其独特的优势，即提供了对安全解决方案的支持。

SecurCore 系列微处理器除了具有 ARM 体系结构各种主要特点外，还在系统安全方面具有如下的特点：

- 带有灵活的保护单元，以确保操作系统和应用数据的安全。
- 采用软内核技术，防止外部对其进行扫描探测。
- 可集成用户自己的安全特性和其他协处理器。

SecurCore 系列微处理器主要应用于一些对安全性要求较高的应用产品及应用系统，如电子商务、电子政务、电子银行业务、网络和认证系统等领域。

SecurCore 系列微处理器包含 SecurCore SC100、SecurCore SC110、SecurCore SC200 和 SecurCore SC210 四种类型，以适用于不同的应用场合。

1.3.6 StrongARM 微处理器系列

Inter StrongARM SA-1100 处理器是采用 ARM 体系结构高度集成的 32 位 RISC 微处理器。它融合了 Inter 公司的设计和处理技术以及 ARM 体系结构的电源效率，采用在软件上兼容 ARMv4 体系结构、同时采用具有 Intel 技术优点的体系结构。

Intel StrongARM 处理器是便携式通讯产品和消费类电子产品的理想选择，已成功应用于多家公司的掌上电脑系列产品。

1.3.7 Xscale 处理器

Xscale 处理器是基于 ARMv5TE 体系结构的解决方案，是一款全性能、高性价比、低功耗的处理器。它支持 16 位的 Thumb 指令和 DSP 指令集，已使用在数字移动电话、个人数字助理和网络产品等场合。

Xscale 处理器是 Inter 目前主要推广的一款 ARM 微处理器。

1.4 ARM 微处理器结构

1.4.1 RISC 体系结构

传统的 CISC（Complex Instruction Set Computer，复杂指令集计算机）结构有其固有的缺点，即

随着计算机技术的发展而不断引入新的复杂的指令集，为支持这些新增的指令，计算机的体系结构会越来越复杂，然而，在 CISC 指令集的各种指令中，其使用频率却相差悬殊，大约有 20%的指令会被反复使用，占整个程序代码的 80%。而余下的 80%的指令却不经常使用，在程序设计中只占 20%，显然，这种结构是不太合理的。

基于以上的不合理性，1979 年美国加州大学伯克利分校提出了 RISC（Reduced Instruction Set Computer，精简指令集计算机）的概念，RISC 并非只是简单地去减少指令，而是把着眼点放在了如何使计算机的结构更加简单合理地提高运算速度上。RISC 结构优先选取使用频最高的简单指令，避免复杂指令；将指令长度固定，指令格式和寻地方式种类减少；以控制逻辑为主，不用或少用微码控制等措施来达到上述目的。

到目前为止，RISC 体系结构也还没有严格的定义，一般认为，RISC 体系结构应具有如下特点：

- 采用固定长度的指令格式，指令归整、简单、基本寻址方式有 2～3 种。
- 使用单周期指令，便于流水线操作执行。
- 大量使用寄存器，数据处理指令只对寄存器进行操作，只有加载/ 存储指令可以访问存储器，以提高指令的执行效率。

除此以外，ARM 体系结构还采用了一些特别的技术，在保证高性能的前提下尽量缩小芯片的面积，并降低功耗：

- 所有的指令都可根据前面的执行结果决定是否被执行，从而提高指令的执行效率。
- 可用加载/存储指令批量传输数据，以提高数据的传输效率。
- 可在一条数据处理指令中同时完成逻辑处理和移位处理。
- 在循环处理中使用地址的自动增减来提高运行效率。

当然，和 CISC 架构相比较，尽管 RISC 架构有上述的优点，但决不能认为 RISC 架构就可以取代 CISC 架构，事实上，RISC 和 CISC 各有优势，而且界限并不那么明显。现代的 CPU 往往采用 CISC 的外围，内部加入了 RISC 的特性，如超长指令集 CPU 就是融合了 RISC 和 CISC 的优势，成为未来的 CPU 发展方向之一。

1.4.2 ARM 微处理器的寄存器结构

ARM 处理器共有 37 个寄存器，被分为若干个组（BANK），这些寄存器包括：

- 31 个通用寄存器，包括程序计数器（PC 指针），均为 32 位的寄存器。
- 6 个状态寄存器，用以标识 CPU 的工作状态及程序的运行状态，均为 32 位，目前只使用了其中的一部分。

同时，ARM 处理器又有 7 种不同的处理器模式，在每一种处理器模式下均有一组相应的寄存器与之对应。即在任意一种处理器模式下，可访问的寄存器包括 15 个通用寄存器（R0～R14）、一至二个状态寄存器和程序计数器。在所有的寄存器中，有些是在 7 种处理器模式下共用的同一个物理寄存器，而有些寄存器则是在不同的处理器模式下有不同的物理寄存器。

关于 ARM 处理器的寄存器结构，在后面的相关章节将会详细描述。

1.4.3 ARM 微处理器的指令结构

ARM 微处理器的在较新的体系结构中支持两种指令集：ARM 指令集和 Thumb 指令集。其中，ARM 指令为 32 位的长度，Thumb 指令为 16 位长度。Thumb 指令集为 ARM 指令集的功能子集，但与等价的 ARM 代码相比较，可节省 30%～40%以上的存储空间，同时具备 32 位代码的所有优点。

关于 ARM 处理器的指令结构，在后面的相关章节将会详细描述。

1.5 ARM 微处理器的应用选型

鉴于 ARM 微处理器的众多优点，随着国内外嵌入式应用领域的逐步发展，ARM 微处理器必然会获得广泛的重视和应用。但是，由于 ARM 微处理器有多达十几种的内核结构，几十个芯片生产厂家，以及千变万化的内部功能配置组合，给开发人员在选择方案时带来一定的困难，所以，对 ARM 芯片做一些对比研究是十分必要的。

以下从应用的角度出发，对在选择 ARM 微处理器时所应考虑的主要问题做一些简要的探讨。

ARM 微处理器内核的选择

从前面所介绍的内容可知，ARM 微处理器包含一系列的内核结构，以适应不同的应用领域，用户如果希望使用 WinCE 或标准 Linux 等操作系统以减少软件开发时间，就需要选择 ARM720T 以上带有 MMU（Memory Management Unit）功能的 ARM 芯片，ARM720T、ARM920T、ARM922T、ARM946T、Strong-ARM 都带有 MMU 功能。而 ARM7TDMI 则没有 MMU，不支持 Windows CE 和标准 Linux，但目前有 uCLinux 等不需要 MMU 支持的操作系统可运行于 ARM7TDMI 硬件平台之上。事实上，uCLinux 已经成功移植到多种不带 MMU 的微处理器平台上，并在稳定性和其他方面都有上佳表现。

本书所讨论的 S3C4510B 即为一款不带 MMU 的 ARM 微处理器，可在其上运行 uCLinux 操作系统。

系统的工作频率

系统的工作频率在很大程度上决定了 ARM 微处理器的处理能力。ARM7 系列微处理器的典型处理速度为 0.9MIPS/MHz，常见的 ARM7 芯片系统主时钟为 20MHz-133MHz，ARM9 系列微处理器的典型处理速度为 1.1MIPS/MHz，常见的 ARM9 的系统主时钟频率为 100MHz-233MHz，ARM10 最高可以达到 700MHz。不同芯片对时钟的处理不同，有的芯片只需要一个主时钟频率，有的芯片内部时钟控制器可以分别为 ARM 核和 USB、UART、DSP、音频等功能部件提供不同频率的时钟。

芯片内存储器的容量

大多数的 ARM 微处理器片内存储器的容量都不太大，需要用户在设计系统时外扩存储器，但也有部分芯片具有相对较大的片内存储空间，如 ATMEL 的 AT91F40162 就具有高达 2MB 的片内程序存储空间，用户在设计时可考虑选用这种类型，以简化系统的设计。

片内外围电路的选择

除 ARM 微处理器核以外，几乎所有的 ARM 芯片均根据各自不同的应用领域，扩展了相关功能模块，并集成在芯片之中，我们称之为片内外围电路，如 USB 接口、IIS 接口、LCD 控制器、键盘接口、RTC、ADC 和 DAC、DSP 协处理器等，设计者应分析系统的需求，尽可能采用片内外围电路完成所需的功能，这样既可简化系统的设计，同时提高系统的可靠性。

1.6 本章小节

本章对 ARM 微处理器、ARM 技术的基本概念做了一些简单的介绍，希望读者通过对本章的阅读，能对 ARM 微处理器、ARM 技术有一个总体上的认识。

第 2 章 ARM 微处理器的编程模型

本章简介 ARM 微处理器编程模型的一些基本概念，包括工作状态切换、数据的存储格式、处理器异常等，通过对本章的阅读，希望读者能了解 ARM 微处理器的基本工作原理和一些与程序设计相关的基本技术细节，为以后的程序设计打下基础。

本章的主要内容：

- ARM 微处理器的工作状态
- ARM 体系结构的存储器格式
- ARM 微处理器的工作模式
- ARM 体系结构的寄存器组织
- ARM 微处理器的异常状态

在开始本章之前，首先对字（Word）、半字（Half-Word）、字节（Byte）的概念作一个说明：

字（Word）：在ARM体系结构中，字的长度为32位，而在8位/16位处理器体系结构中，字的长度一般为16位，请读者在阅读时注意区分。

半字（Half-Word）：在ARM体系结构中，半字的长度为16位，与8位/16位处理器体系结构中字的长度一致。

字节（Byte）：在ARM体系结构和8位/16位处理器体系结构中，字节的长度均为8位。

2.1 ARM 微处理器的工作状态

从编程的角度看，ARM微处理器的工作状态一般有两种，并可在两种状态之间切换：

- 第一种为ARM状态，此时处理器执行32位的字对齐的ARM指令；
- 第二种为Thumb状态，此时处理器执行16位的、半字对齐的Thumb指令。

当ARM微处理器执行32位的ARM指令集时，工作在ARM状态；当ARM微处理器执行16位的Thumb指令集时，工作在Thumb状态。在程序的执行过程中，微处理器可以随时在两种工作状态之间切换，并且，处理器工作状态的转变并不影响处理器的工作模式和相应寄存器中的内容。

状态切换方法：

ARM指令集和Thumb指令集均有切换处理器状态的指令，并可在两种工作状态之间切换，但ARM微处理器在开始执行代码时，应该处于ARM状态。

进入Thumb状态：当操作数寄存器的状态位（位0）为1时，可以采用执行BX指令的方法，使微处理器从ARM状态切换到Thumb状态。此外，当处理器处于Thumb状态时发生异常（如IRQ、FIQ、Undef、Abort、SWI等），则异常处理返回时，自动切换到Thumb状态。

进入ARM状态：当操作数寄存器的状态位为0时，执行BX指令时可以使微处理器从Thumb状态切换到ARM状态。此外，在处理器进行异常处理时，把PC指针放入异常模式链接寄存器中，并从异常向量地址开始执行程序，也可以使处理器切换到ARM状态。

2.2 ARM 体系结构的存储器格式

ARM体系结构将存储器看作是从零地址开始的字节的线性组合。从零字节到三字节放置第一个存储的字数据，从第四个字节到第七个字节放置第二个存储的字数据，依次排列。作为32位的微处理器，ARM体系结构所支持的最大寻址空间为4GB（2^{32}字节）。

ARM体系结构可以用两种方法存储字数据，称之为大端格式和小端格式，具体说明如下：

大端格式：

在这种格式中，字数据的高字节存储在低地址中，而字数据的低字节则存放在高地址中，如图2.1所示：

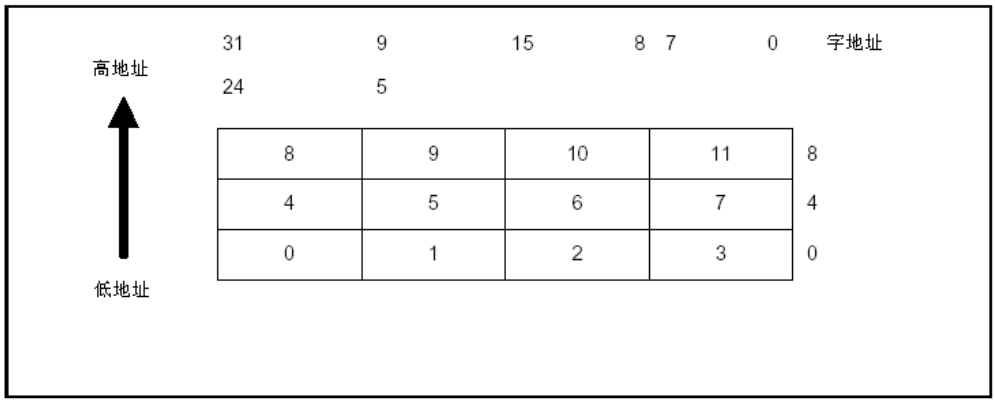


图 2.1　以大端格式存储字数据

小端格式：

与大端存储格式相反，在小端存储格式中，低地址中存放的是字数据的低字节，高地址存放的是字数据的高字节。如图2.2所示：

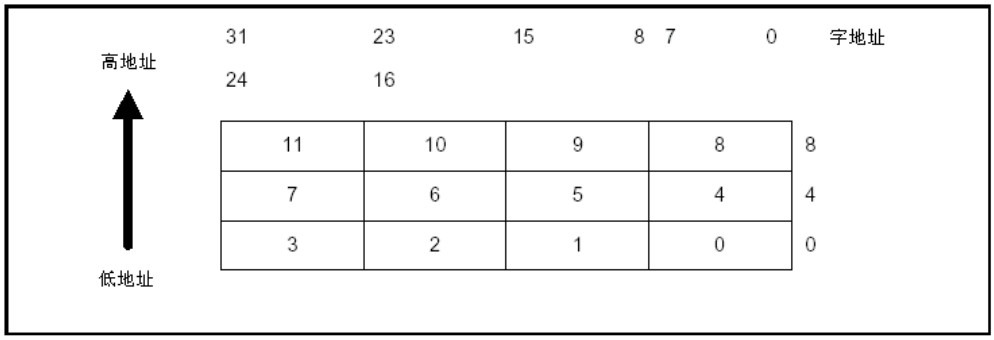


图 2. 2 以小端格式存储字数据

2.3　指令长度及数据类型

ARM微处理器的指令长度可以是32位（在ARM状态下），也可以为16位（在Thumb状态下）。

ARM微处理器中支持字节（8位）、半字（16位）、字（32位）三种数据类型，其中，字需要4字节对齐（地址的低两位为0）、半字需要2字节对齐（地址的最低位为0）。

2.4　处理器模式

ARM微处理器支持7种运行模式，分别为：

— 用户模式（usr）：　　ARM处理器正常的程序执行状态

— 快速中断模式（fiq）：　用于高速数据传输或通道处理

— 外部中断模式（irq）：　用于通用的中断处理

— 管理模式（svc）：　　操作系统使用的保护模式

— 数据访问终止模式(abt)：当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。

— 系统模式（sys）：　　运行具有特权的操作系统任务。

— 未定义指令中止模式（und）：当未定义的指令执行时进入该模式，可用于支持硬件协处

理器的软件仿真。

ARM微处理器的运行模式可以通过软件改变，也可以通过外部中断或异常处理改变。

大多数的应用程序运行在用户模式下，当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。

除用户模式以外，其余的所有6种模式称之为非用户模式，或特权模式（Privileged Modes）；其中除去用户模式和系统模式以外的5种又称为异常模式（Exception Modes），常用于处理中断或异常，以及需要访问受保护的系统资源等情况。

2.5 寄存器组织

ARM微处理器共有37个32位寄存器，其中31个为通用寄存器，6个为状态寄存器。但是这些寄存器不能被同时访问，具体哪些寄存器是可编程访问的，取决微处理器的工作状态及具体的运行模式。但在任何时候，通用寄存器R14～R0、程序计数器PC、一个或两个状态寄存器都是可访问的。

2.5.1 ARM 状态下的寄存器组织

通用寄存器：

通用寄存器包括R0～R15，可以分为三类：

— 未分组寄存器R0～R7；

— 分组寄存器R8～R14

— 程序计数器PC(R15)

未分组寄存器R0～R7：

在所有的运行模式下，未分组寄存器都指向同一个物理寄存器，他们未被系统用作特殊的用途，因此，在中断或异常处理进行运行模式转换时，由于不同的处理器运行模式均使用相同的物理寄存器，可能会造成寄存器中数据的破坏，这一点在进行程序设计时应引起注意。

分组寄存器R8～R14

对于分组寄存器，他们每一次所访问的物理寄存器与处理器当前的运行模式有关。

对于R8～R12来说，每个寄存器对应两个不同的物理寄存器，当使用fiq模式时，访问寄存器R8_fiq～R12_fiq；当使用除fiq模式以外的其他模式时，访问寄存器R8_usr～R12_usr。

对于R13、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与系统模式共用，另外5个物理寄存器对应于其他5种不同的运行模式。

采用以下的记号来区分不同的物理寄存器：

```
R13_<mode>
R14_<mode>
```

其中，mode为以下几种模式之一：usr、fiq、irq、svc、abt、und。

寄存器R13在ARM指令中常用作堆栈指针，但这只是一种习惯用法，用户也可使用其他的寄存器作为堆栈指针。而在Thumb指令集中，某些指令强制性的要求使用R13作为堆栈指针。

由于处理器的每种运行模式均有自己独立的物理寄存器R13，在用户应用程序的初始化部分，一般都要初始化每种模式下的R13，使其指向该运行模式的栈空间，这样，当程序的运行进入异常模式时，可以将需要保护的寄存器放入R13所指向的堆栈，而当程序从异常模式返回时，则从对应的堆栈中恢复，采用这种方式可以保证异常发生后程序的正常执行。

R14也称作子程序连接寄存器（Subroutine Link Register）或连接寄存器LR。当执行BL子程序调用指令时，R14中得到R15（程序计数器PC）的备份。其他情况下，R14用作通用寄存器。与之类似，当发生中断或异常时，对应的分组寄存器R14_svc、R14_irq、R14_fiq、R14_abt和R14_und用来保存R15的返回值。

寄存器R14常用在如下的情况：

在每一种运行模式下，都可用R14保存子程序的返回地址，当用BL或BLX指令调用子程序时，将PC的当前值拷贝给R14，执行完子程序后，又将R14的值拷贝回PC，即可完成子程序的调用返回。以上的描述可用指令完成：

1、执行以下任意一条指令：

```
MOV       PC，LR
BX        LR
```

2、在子程序入口处使用以下指令将R14存入堆栈：

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

对应的，使用以下指令可以完成子程序返回：

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

R14也可作为通用寄存器。

程序计数器PC(R15)

寄存器R15用作程序计数器（PC）。在ARM状态下，位[1:0]为0，位[31:2]用于保存PC；在Thumb状态下，位[0]为0，位[31:1]用于保存PC；虽然可以用作通用寄存器，但是有一些指令在使用R15时有一些特殊限制，若不注意，执行的结果将是不可预料的。在ARM状态下，PC的0和1位是0，在Thumb状态下，PC的0位是0。

R15虽然也可用作通用寄存器，但一般不这么使用，因为对R15的使用有一些特殊的限制，当违反了这些限制时，程序的执行结果是未知的。

由于 ARM 体系结构采用了多级流水线技术，对于 ARM 指令集而言，PC 总是指向当前指令的下两条指令的地址，即 PC 的值为当前指令的地址值加 8 个字节。

图 2.3　ARM 状态下的寄存器组织

在ARM状态下，任一时刻可以访问以上所讨论的16个通用寄存器和一到两个状态寄存器。在非用户模式（特权模式）下，则可访问到特定模式分组寄存器，图2.3说明在每一种运行模式下，哪一些寄存器是可以访问的。

寄存器R16：

寄存器R16用作CPSR(Current Program Status Register，当前程序状态寄存器)，CPSR可在任何运

行模式下被访问，它包括条件标志位、中断禁止位、当前处理器模式标志位，以及其他一些相关的控制和状态位。

每一种运行模式下又都有一个专用的物理状态寄存器，称为SPSR（Saved Program Status Register，备份的程序状态寄存器），当异常发生时，SPSR用于保存CPSR的当前值，从异常退出时则可由SPSR来恢复CPSR。

由于用户模式和系统模式不属于异常模式，他们没有SPSR，当在这两种模式下访问SPSR，结果是未知的。

2.5.2 Thumb 状态下的寄存器组织

Thumb状态下的寄存器集是ARM状态下寄存器集的一个子集，程序可以直接访问8个通用寄存器（R7～R0）、程序计数器（PC）、堆栈指针（SP）、连接寄存器（LR）和CPSR。同时，在每一种特权模式下都有一组SP、LR和SPSR。图2.4表明Thumb状态下的寄存器组织。

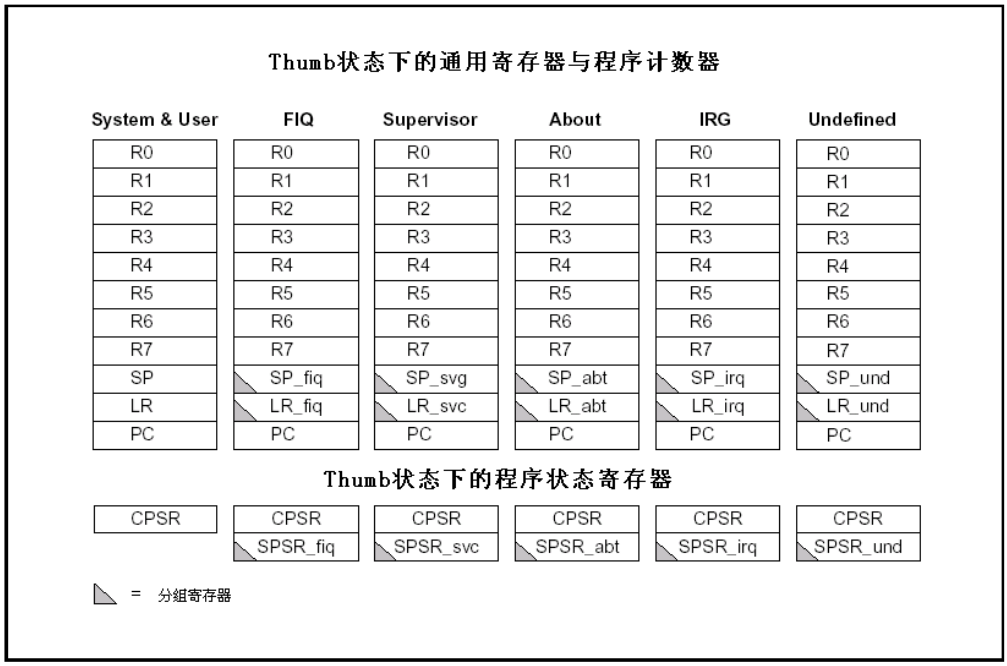


图 2.4 Thumb 状态下的寄存器组织

Thumb状态下的寄存器组织与ARM状态下的寄存器组织的关系：

- Thumb状态下和ARM状态下的R0～R7是相同的。
- Thumb状态下和ARM状态下的CPSR和所有的SPSR是相同的。
- Thumb状态下的SP对应于ARM状态下的R13。
- Thumb状态下的LR对应于ARM状态下的R14。
- Thumb状态下的程序计数器对应于ARM状态下R15

以上的对应关系如图2.5所示：

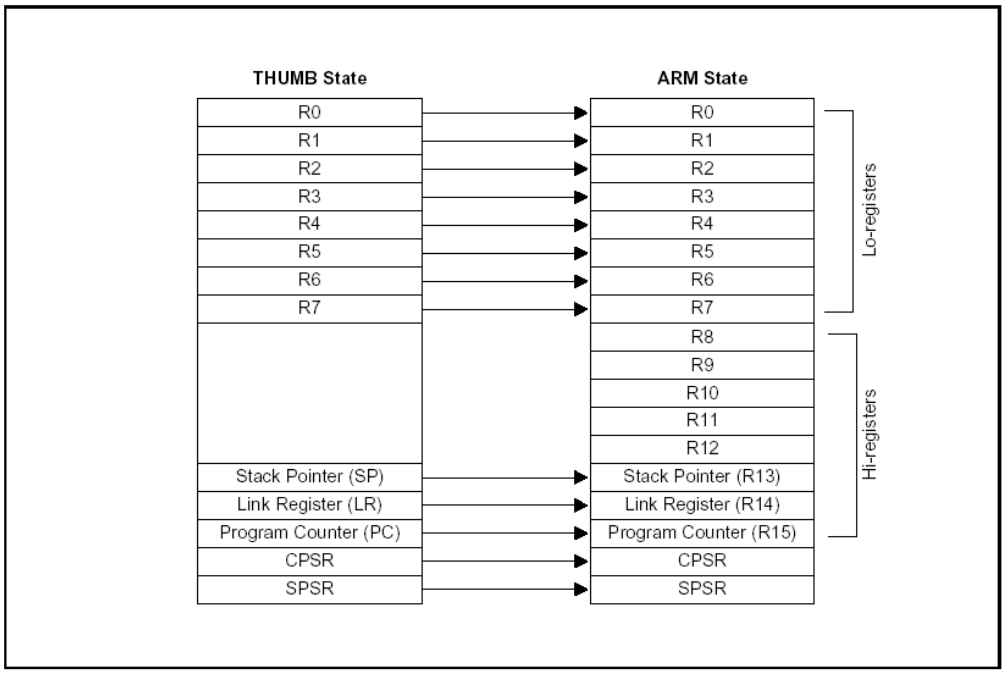


图 2.5　Thumb 状态下的寄存器组织

访问THUMB状态下的高位寄存器（Hi-registers）：

在Thumb状态下，高位寄存器R8～R15并不是标准寄存器集的一部分，但可使用汇编语言程序受限制的访问这些寄存器，将其用作快速的暂存器。使用带特殊变量的MOV指令，数据可以在低位寄存器和高位寄存器之间进行传送；高位寄存器的值可以使用CMP和ADD指令进行比较或加上低位寄存器中的值。

2.5.3　程序状态寄存器

ARM体系结构包含一个当前程序状态寄存器（CPSR）和五个备份的程序状态寄存器（SPSRs）。备份的程序状态寄存器用来进行异常处理，其功能包括：

— 保存ALU中的当前操作信息
— 控制允许和禁止中断
— 设置处理器的运行模式

程序状态寄存器的每一位的安排如图2.6所示：

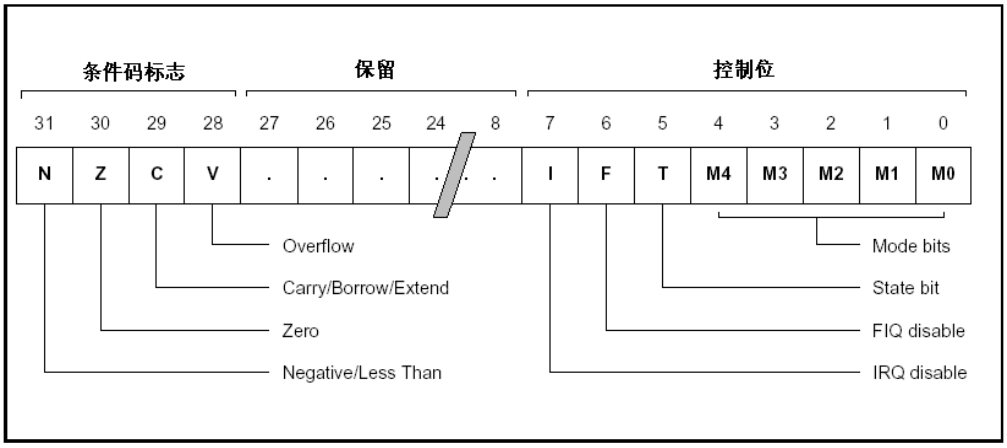


图 2.6　程序状态寄存器格式

条件码标志（Condition Code Flags）

N、Z、C、V均为条件码标志位。它们的内容可被算术或逻辑运算的结果所改变，并且可以决定某条指令是否被执行。

在ARM状态下，绝大多数的指令都是有条件执行的。

在Thumb状态下，仅有分支指令是有条件执行的。

条件码标志各位的具体含义如表2-1所示：

表 2-1 条件码标志的具体含义

标志位	含 义
N	当用两个补码表示的带符号数进行运算时，N=1 表示运算的结果为负数；N=0 表示运算的结果为正数或零；
Z	Z=1 表示运算的结果为零；Z=0 表示运算的结果为非零；
C	可以有 4 种方法设置 C 的值： — 加法运算（包括比较指令 CMN）：当运算结果产生了进位时（无符号数溢出），C=1，否则 C=0。 — 减法运算（包括比较指令 CMP）：当运算时产生了借位（无符号数溢出），C=0，否则 C=1。 — 对于包含移位操作的非加/减运算指令，C 为移出值的最后一位。 — 对于其他的非加/减运算指令，C 的值通常不改变。
V	可以有 2 种方法设置 V 的值： — 对于加/减法运算指令，当操作数和运算结果为二进制的补码表示的带符号数时，V=1 表示符号位溢出。 — 对于其他的非加/减运算指令，V 的值通常不改变。
Q	在 ARM v5 及以上版本的 E 系列处理器中，用 Q 标志位指示增强的 DSP 运算指令是否发生了溢出。在其他版本的处理器中，Q 标志位无定义。

控制位

PSR的低8位（包括I、F、T和M[4：0]）称为控制位，当发生异常时这些位可以被改变。如果处理器运行特权模式，这些位也可以由程序修改。

— 中断禁止位I、F：

I=1 禁止IRQ中断;

F=1 禁止FIQ中断。

— T标志位：该位反映处理器的运行状态。

对于ARM体系结构v5及以上的版本的T系列处理器，当该位为1时，程序运行于Thumb状态，否则运行于ARM状态。

对于ARM体系结构v5及以上的版本的非T系列处理器，当该位为1时，执行下一条指令以引起为定义的指令异常；当该位为0时，表示运行于ARM状态。

— 运行模式位M[4：0]：M0、M1、M2、M3、M4是模式位。这些位决定了处理器的运行模式。具体含义如表2-2所示：

表 2-2 运行模式位 M[4：0]的具体含义

M[4：0]	处理器模式	可访问的寄存器
0b10000	用户模式	PC，CPSR,R0-R14
0b10001	FIQ 模式	PC，CPSR, SPSR_fiq，R14_fiq-R8_fiq, R7～R0
0b10010	IRQ 模式	PC，CPSR, SPSR_irq，R14_irq,R13_irq,R12～R0
0b10011	管理模式	PC，CPSR, SPSR_svc，R14_svc,R13_svc,,R12～R0,
0b10111	中止模式	PC，CPSR, SPSR_abt，R14_abt,R13_abt, R12～R0,
0b11011	未定义模式	PC，CPSR, SPSR_und，R14_und,R13_und, R12～R0,
0b11111	系统模式	PC，CPSR（ARM v4 及以上版本）, R14～R0

由表 2-2 可知，并不是所有的运行模式位的组合都是有效地，其他的组合结果会导致处理器进入一个不可恢复的状态。

保留位

PSR中的其余位为保留位，当改变PSR中的条件码标志位或者控制位时，保留位不要被改变，在程序中也不要使用保留位来存储数据。保留位将用于ARM版本的扩展。

2.6 异常（Exceptions）

当正常的程序执行流程发生暂时的停止时，称之为异常，例如处理一个外部的中断请求。在处理异常之前，当前处理器的状态必须保留，这样当异常处理完成之后，当前程序可以继续执行。处理器允许多个异常同时发生，它们将会按固定的优先级进行处理。

ARM体系结构中的异常，与8位/16位体系结构的中断有很大的相似之处，但异常与中断的概念并不完全等同。

2.6.1 ARM 体系结构所支持的异常类型

ARM体系结构所支持的异常及具体含义如表2-3所示。

表 2-3 ARM 体系结构所支持的异常

异常类型	具体含义
复位	当处理器的复位电平有效时，产生复位异常，程序跳转到复位异常处理程序处执行。
未定义指令	当 ARM 处理器或协处理器遇到不能处理的指令时，产生未定义指令异常。可使用该异常机制进行软件仿真。
软件中断	该异常由执行 SWI 指令产生，可用于用户模式下的程序调用特权操作指令。可使用该异常机制实现系统功能调用。
指令预取中止	若处理器预取指令的地址不存在，或该地址不允许当前指令访问，存储器会向处理器发出中止信号，但当预取的指令被执行时，才会产生指令预取中止异常。
数据中止	若处理器数据访问指令的地址不存在，或该地址不允许当前指令访问时，产生数据中止异常。
IRQ（外部中断请求）	当处理器的外部中断请求引脚有效，且 CPSR 中的 I 位为 0 时，产生 IRQ 异常。系统的外设可通过该异常请求中断服务。
FIQ（快速中断请求）	当处理器的快速中断请求引脚有效，且 CPSR 中的 F 位为 0 时，产生 FIQ 异常。

2.6.2 对异常的响应

当一个异常出现以后，ARM微处理器会执行以下几步操作：

1、将下一条指令的地址存入相应连接寄存器LR，以便程序在处理异常返回时能从正确的位置重新开始执行。若异常是从ARM状态进入，LR寄存器中保存的是下一条指令的地址（当前PC＋4或PC＋8，与异常的类型有关）；若异常是从Thumb状态进入，则在LR寄存器中保存当前PC的偏移量，这样，异常处理程序就不需要确定异常是从何种状态进入的。例如：在软件中断异常SWI，指令 MOV PC，R14_svc总是返回到下一条指令，不管SWI是在ARM状态执行，还是在Thumb状态执行。

2、将CPSR复制到相应的SPSR中。

3、根据异常类型，强制设置CPSR的运行模式位。

4、强制PC从相关的异常向量地址取下一条指令执行，从而跳转到相应的异常处理程序处。

还可以设置中断禁止位，以禁止中断发生。

如果异常发生时，处理器处于Thumb状态，则当异常向量地址加载入PC时，处理器自动切换到ARM状态。

ARM微处理器对异常的响应过程用伪码可以描述为：

```
R14_<Exception_Mode> = Return Link
SPSR_<Exception_Mode> = CPSR
CPSR[4:0] = Exception Mode Number
```

```
CPSR[5] = 0                    ; 当运行于 ARM 工作状态时
If <Exception_Mode> == Reset or FIQ then
                               ; 当响应 FIQ 异常时，禁止新的 FIQ 异常
    CPSR[6] = 1
    CPSR[7] = 1
PC = Exception Vector Address
```

2.6.3 从异常返回

异常处理完毕之后，ARM微处理器会执行以下几步操作从异常返回：

1、将连接寄存器LR的值减去相应的偏移量后送到PC中。

2、将SPSR复制回CPSR中。

3、若在进入异常处理时设置了中断禁止位，要在此清除。

可以认为应用程序总是从复位异常处理程序开始执行的，因此复位异常处理程序不需要返回。

2.6.4 各类异常的具体描述

FIQ（Fast Interrupt Request）

FIQ异常是为了支持数据传输或者通道处理而设计的。在ARM状态下，系统有足够的私有寄存器，从而可以避免对寄存器保存的需求，并减小了系统上下文切换的开销。

若将CPSR的F位置为1，则会禁止FIQ中断，若将CPSR的F位清零，处理器会在指令执行时检查FIQ的输入。注意只有在特权模式下才能改变F位的状态。

可由外部通过对处理器上的nFIQ引脚输入低电平产生FIQ。不管是在ARM状态还是在Thumb状态下进入FIQ模式，FIQ处理程序均会执行以下指令从FIQ模式返回：

```
SUBS   PC,R14_fiq ,#4
```

该指令将寄存器 R14_fiq 的值减去 4 后，复制到程序计数器 PC 中，从而实现从异常处理程序中的返回，同时将 SPSR_mode 寄存器的内容复制到当前程序状态寄存器 CPSR 中。

IRQ（Interrupt Request）

IRQ异常属于正常的中断请求，可通过对处理器的nIRQ引脚输入低电平产生，IRQ的优先级低于FIQ，当程序执行进入FIQ异常时，IRQ可能被屏蔽。

若将CPSR的I位置为1，则会禁止IRQ中断，若将CPSR的I位清零，处理器会在指令执行完之前检查IRQ的输入。注意只有在特权模式下才能改变I位的状态。

不管是在ARM状态还是在Thumb状态下进入IRQ模式，IRQ处理程序均会执行以下指令从IRQ模式返回：

```
SUBS  PC , R14_irq , #4
```

该指令将寄存器 R14_irq 的值减去 4 后，复制到程序计数器 PC 中，从而实现从异常处理程序中的返回，同时将 SPSR_mode 寄存器的内容复制到当前程序状态寄存器 CPSR 中。

ABORT（中止）

产生中止异常意味着对存储器的访问失败。ARM微处理器在存储器访问周期内检查是否发生中止异常。

中止异常包括两种类型：

— 指令预取中止：发生在指令预取时。

— 数据中止：发生在数据访问时。

当指令预取访问存储器失败时，存储器系统向ARM处理器发出存储器中止（Abort）信号，预取的指令被记为无效，但只有当处理器试图执行无效指令时，指令预取中止异常才会发生，如果指令未被执行，例如在指令流水线中发生了跳转，则预取指令中止不会发生。

若数据中止发生，系统的响应与指令的类型有关。

当确定了中止的原因后，Abort处理程序均会执行以下指令从中止模式返回，无论是在ARM状态还是Thumb状态：

```
SUBS PC, R14_abt, #4      ；指令预取中止
SUBS PC, R14_abt, #8      ；数据中止
```

以上指令恢复PC（从R14_abt）和CPSR（从SPSR_abt）的值，并重新执行中止的指令。

Software Interruupt(软件中断)

软件中断指令（SWI）用于进入管理模式，常用于请求执行特定的管理功能。软件中断处理程序执行以下指令从SWI模式返回，无论是在ARM状态还是Thumb状态：

```
MOV  PC , R14_svc
```

以上指令恢复PC（从R14_svc）和CPSR（从SPSR_svc）的值，并返回到SWI的下一条指令。

Undefined Instruction(未定义指令)

当ARM处理器遇到不能处理的指令时，会产生未定义指令异常。采用这种机制，可以通过软件仿真扩展ARM或Thumb指令集。

在仿真未定义指令后，处理器执行以下程序返回，无论是在ARM状态还是Thumb状态：

```
MOVS PC, R14_und
```

以上指令恢复PC（从R14_und）和CPSR（从SPSR_und）的值，并返回到未定义指令后的下一条指令。

2.6.5 异常进入/退出小节

表 2-4 总结了进入异常处理时保存在相应 R14 中的 PC 值，及在退出异常处理时推荐使用的指令。

表 2-4 异常进入/退出

	返回指令	以前的状态		注意
		ARM R14_x	Thumb R14_x	
BL	MOV PC，R14	PC＋4	PC＋2	1
SWI	MOVS PC，R14_svc	PC＋4	PC＋2	1
UDEF	MOVS PC，R14_und	PC＋4	PC＋2	1
FIQ	SUBS PC，R14_fiq，＃4	PC＋4	PC＋4	2
IRQ	SUBS PC，R14_irq，＃4	PC＋4	PC＋4	2
PABT	SUBS PC，R14_abt，＃4	PC＋4	PC＋4	1
DABT	SUBS PC，R14_abt，＃8	PC＋8	PC＋8	3
RESET	NA	－	－	4

注意：

1、在此 PC 应是具有预取中止的 BL/SWI/未定义指令所取的地址。

2、在此 PC 是从 FIQ 或 IRQ 取得不能执行的指令的地址。

3、在此 PC 是产生数据中止的加载或存储指令的地址。

4、系统复位时，保存在 R14_svc 中的值是不可预知的。

2.6.6 异常向量（Exception Vectors）

表2-5显示异常向量地址。

表 2-5 异常向量表

地 址	异 常	进入模式
0x0000,0000	复位	管理模式
0x0000,0004	未定义指令	未定义模式
0x0000,0008	软件中断	管理模式

0x0000,000C	中止（预取指令）	中止模式
0x0000,0010	中止（数据）	中止模式
0x0000,0014	保留	保留
0x0000,0018	IRQ	IRQ
0x0000,001C	FIQ	FIQ

2.6.7 异常优先级（Exception Priorities）

当多个异常同时发生时，系统根据固定的优先级决定异常的处理次序。异常优先级由高到低的排列次序如表2-6所示。

表 2-6 异常优先级

优先级	异 常
1（最高）	复位
2	数据中止
3	FIQ
4	IRQ
5	预取指令中止
6（最低）	未定义指令、SWI

2.6.8 应用程序中的异常处理

当系统运行时，异常可能会随时发生，为保证在 ARM 处理器发生异常时不至于处于未知状态，在应用程序的设计中，首先要进行异常处理，采用的方式是在异常向量表中的特定位置放置一条跳转指令，跳转到异常处理程序，当 ARM 处理器发生异常时，程序计数器 PC 会被强制设置为对应的异常向量，从而跳转到异常处理程序，当异常处理完成以后，返回到主程序继续执行。

2.7 本章小节

本章对 ARM 微处理器的体系结构、寄存器的组织、处理器的工作状态、运行模式以及处理器异常等内容进行了描述，这些内容也是 ARM 体系结构的基本内容，是系统软、硬件设计的基础。

第 3 章 ARM 微处理器的指令系统

本章介绍 ARM 指令集、Thumb 指令集，以及各类指令对应的寻址方式，通过对本章的阅读，希望读者能了解 ARM 微处理器所支持的指令集及具体的使用方法。

本章的主要内容有：

- ARM 指令集、Thumb 指令集概述。
- ARM 指令集的分类与具体应用。
- Thumb 指令集简介及应用场合。

3.1 ARM 微处理器的指令集概述

3.1.1 ARM 微处理器的指令的分类与格式

ARM微处理器的指令集是加载/存储型的，也即指令集仅能处理寄存器中的数据，而且处理结果都要放回寄存器中，而对系统存储器的访问则需要通过专门的加载/存储指令来完成。

ARM微处理器的指令集可以分为跳转指令、数据处理指令、程序状态寄存器（PSR）处理指令、加载/存储指令、协处理器指令和异常产生指令六大类，具体的指令及功能如表3-1所示（表中指令为基本ARM指令，不包括派生的ARM指令）。

表 3-1 ARM 指令及功能描述

助记符	指令功能描述
ADC	带进位加法指令
ADD	加法指令
AND	逻辑与指令
B	跳转指令
BIC	位清零指令
BL	带返回的跳转指令
BLX	带返回和状态切换的跳转指令
BX	带状态切换的跳转指令
CDP	协处理器数据操作指令
CMN	比较反值指令
CMP	比较指令
EOR	异或指令
LDC	存储器到协处理器的数据传输指令
LDM	加载多个寄存器指令
LDR	存储器到寄存器的数据传输指令
MCR	从 ARM 寄存器到协处理器寄存器的数据传输指令
MLA	乘加运算指令
MOV	数据传送指令
MRC	从协处理器寄存器到 ARM 寄存器的数据传输指令
MRS	传送 CPSR 或 SPSR 的内容到通用寄存器指令
MSR	传送通用寄存器到 CPSR 或 SPSR 的指令

MUL	32 位乘法指令
MLA	32 位乘加指令
MVN	数据取反传送指令
ORR	逻辑或指令
RSB	逆向减法指令
RSC	带借位的逆向减法指令
SBC	带借位减法指令
STC	协处理器寄存器写入存储器指令
STM	批量内存字写入指令
STR	寄存器到存储器的数据传输指令
SUB	减法指令
SWI	软件中断指令
SWP	交换指令
TEQ	相等测试指令
TST	位测试指令

3.1.2 指令的条件域

当处理器工作在ARM状态时，几乎所有的指令均根据CPSR中条件码的状态和指令的条件域有条件的执行。当指令的执行条件满足时，指令被执行，否则指令被忽略。

每一条ARM指令包含4位的条件码，位于指令的最高4位[31:28]。条件码共有16种，每种条件码可用两个字符表示，这两个字符可以添加在指令助记符的后面和指令同时使用。例如，跳转指令B可以加上后缀EQ变为BEQ表示“相等则跳转”，即当CPSR中的Z标志置位时发生跳转。

在16种条件标志码中，只有15种可以使用，如表3-2所示，第16种（1111）为系统保留，暂时不能使用。

表 3-2 指令的条件码

条件码	助记符后缀	标 志	含 义
0000	EQ	Z 置位	相等
0001	NE	Z 清零	不相等
0010	CS	C 置位	无符号数大于或等于
0011	CC	C 清零	无符号数小于
0100	MI	N 置位	负数
0101	PL	N 清零	正数或零
0110	VS	V 置位	溢出
0111	VC	V 清零	未溢出
1000	HI	C 置位 Z 清零	无符号数大于
1001	LS	C 清零 Z 置位	无符号数小于或等于
1010	GE	N 等于 V	带符号数大于或等于
1011	LT	N 不等于 V	带符号数小于
1100	GT	Z 清零且（N 等于 V）	带符号数大于
1101	LE	Z 置位或（N 不等于 V）	带符号数小于或等于
1110	AL	忽略	无条件执行

3.2 ARM 指令的寻址方式

所谓寻址方式就是处理器根据指令中给出的地址信息来寻找物理地址的方式。目前ARM指令系统支持如下几种常见的寻址方式。

3.2.1 立即寻址

立即寻址也叫立即数寻址，这是一种特殊的寻址方式，操作数本身就在指令中给出，只要取出指令也就取到了操作数。这个操作数被称为立即数，对应的寻址方式也就叫做立即寻址。例如以下指令：

```
ADD R0，R0，＃1          ；R0←R0＋1
ADD R0，R0，＃0x3f       ；R0←R0＋0x3f
```

在以上两条指令中，第二个源操作数即为立即数，要求以“＃”为前缀，对于以十六进制表示的立即数，还要求在“＃”后加上“0x”或“&”。

3.2.2 寄存器寻址

寄存器寻址就是利用寄存器中的数值作为操作数，这种寻址方式是各类微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。以下指令：

```
ADD R0，R1，R2           ；R0←R1＋R2
```

该指令的执行效果是将寄存器R1和R2的内容相加，其结果存放在寄存器R0中。

3.2.2 寄存器间接寻址

寄存器间接寻址就是以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中。例如以下指令：

```
ADD R0，R1，[R2]         ；R0←R1＋[R2]
LDR  R0，[R1]            ；R0←[R1]
STR  R0，[R1]            ；[R1]←R0
```

在第一条指令中，以寄存器 R2 的值作为操作数的地址，在存储器中取得一个操作数后与 R1 相加，结果存入寄存器 R0 中。

第二条指令将以 R1 的值为地址的存储器中的数据传送到 R0 中。

第三条指令将 R0 的值传送到以 R1 的值为地址的存储器中。

3.2.3 基址变址寻址

基址变址寻址就是将寄存器（该寄存器一般称作基址寄存器）的内容与指令中给出的地址偏移量相加，从而得到一个操作数的有效地址。变址寻址方式常用于访问某基地址附近的地址单元。采用变址寻址方式的指令常见有以下几种形式，如下所示：

```
LDR R0，[R1，＃4]          ；R0←[R1＋4]
LDR R0，[R1，＃4]!         ；R0←[R1＋4]、R1←R1＋4
LDR R0，[R1] ，＃4         ；R0←[R1]、R1←R1＋4
LDR R0，[R1，R2]           ；R0←[R1＋R2]
```

在第一条指令中，将寄存器 R1 的内容加上 4 形成操作数的有效地址，从而取得操作数存入寄存器 R0 中。

在第二条指令中，将寄存器 R1 的内容加上 4 形成操作数的有效地址，从而取得操作数存入寄存器 R0 中，然后，R1 的内容自增 4 个字节。

在第三条指令中，以寄存器 R1 的内容作为操作数的有效地址，从而取得操作数存入寄存器 R0 中，然后，R1 的内容自增 4 个字节。

在第四条指令中，将寄存器 R1 的内容加上寄存器 R2 的内容形成操作数的有效地址，从而取得操作数存入寄存器 R0 中。

3.2.4 多寄存器寻址

采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送。这种寻址方式可以用一条指令完成传送最多 16 个通用寄存器的值。以下指令：

```
LDMIA R0，{R1，R2，R3，R4}        ；R1←[R0]
                                  ；R2←[R0＋4]
                                  ；R3←[R0＋8]
                                  ；R4←[R0＋12]
```

该指令的后缀 IA 表示在每次执行完加载/存储操作后，R0 按字长度增加，因此，指令可将连续存储单元的值传送到 R1～R4。

3.2.5 相对寻址

与基址变址寻址方式相类似，相对寻址以程序计数器 PC 的当前值为基地址，指令中的地址标号作为偏移量，将两者相加之后得到操作数的有效地址。以下程序段完成子程序的调用和返回，跳转指令 BL 采用了相对寻址方式：

```
    BL   NEXT                     ；跳转到子程序 NEXT 处执行
    ……
NEXT
    ……
    MOV  PC，LR                   ；从子程序返回
```

3.2.6 堆栈寻址

堆栈是一种数据结构，按先进后出（First In Last Out，FILO）的方式工作，使用一个称作堆栈指针的专用寄存器指示当前的操作位置，堆栈指针总是指向栈顶。

当堆栈指针指向最后压入堆栈的数据时，称为满堆栈（Full Stack），而当堆栈指针指向下一个将要放入数据的空位置时，称为空堆栈（Empty Stack）。

同时，根据堆栈的生成方式，又可以分为递增堆栈（Ascending Stack）和递减堆栈（Decending Stack），当堆栈由低地址向高地址生成时，称为递增堆栈，当堆栈由高地址向低地址生成时，称为递减堆栈。这样就有四种类型的堆栈工作方式，ARM 微处理器支持这四种类型的堆栈工作方式，即：

- 满递增堆栈：堆栈指针指向最后压入的数据，且由低地址向高地址生成。
- 满递减堆栈：堆栈指针指向最后压入的数据，且由高地址向低地址生成。
- 空递增堆栈：堆栈指针指向下一个将要放入数据的空位置，且由低地址向高地址生成。
- 空递减堆栈：堆栈指针指向下一个将要放入数据的空位置，且由高地址向低地址生成。

3.3 ARM 指令集

本节对 ARM 指令集的六大类指令进行详细的描述。

3.3.1 跳转指令

跳转指令用于实现程序流程的跳转，在 ARM 程序中有两种方法可以实现程序流程的跳转：

- 使用专门的跳转指令。
- 直接向程序计数器 PC 写入跳转地址值。

通过向程序计数器 PC 写入跳转地址值，可以实现在 4GB 的地址空间中的任意跳转，在跳转之

前结合使用

```
MOV   LR，PC
```

等类似指令，可以保存将来的返回地址值，从而实现在 4GB 连续的线性地址空间的子程序调用。

ARM 指令集中的跳转指令可以完成从当前指令向前或向后的 32MB 的地址空间的跳转，包括以下 4 条指令：

— B 跳转指令
— BL 带返回的跳转指令
— BLX 带返回和状态切换的跳转指令
— BX 带状态切换的跳转指令

1、B 指令

B 指令的格式为：

B{条件} 目标地址

B 指令是最简单的跳转指令。一旦遇到一个 B 指令，ARM 处理器将立即跳转到给定的目标地址，从那里继续执行。注意存储在跳转指令中的实际值是相对当前 PC 值的一个偏移量，而不是一个绝对地址，它的值由汇编器来计算（参考寻址方式中的相对寻址）。它是 24 位有符号数，左移两位后有符号扩展为 32 位，表示的有效偏移为 26 位(前后 32MB 的地址空间)。以下指令：

```
B     Label        ；程序无条件跳转到标号 Label 处执行
CMP   R1，＃0      ；当 CPSR 寄存器中的 Z 条件码置位时，程序跳转到标号 Label 处执行
BEQ   Label
```

2、BL 指令

BL 指令的格式为：

BL{条件} 目标地址

BL 是另一个跳转指令，但跳转之前，会在寄存器 R14 中保存 PC 的当前内容，因此，可以通过将 R14 的内容重新加载到 PC 中，来返回到跳转指令之后的那个指令处执行。该指令是实现子程序调用的一个基本但常用的手段。以下指令：

```
BL    Label        ；当程序无条件跳转到标号 Label 处执行时，同时将当前的 PC 值保存到 R14 中
```

3、BLX 指令

BLX 指令的格式为：

BLX 目标地址

BLX 指令从 ARM 指令集跳转到指令中所指定的目标地址，并将处理器的工作状态有 ARM 状态切换到 Thumb 状态，该指令同时将 PC 的当前内容保存到寄存器 R14 中。因此，当子程序使用 Thumb 指令集，而调用者使用 ARM 指令集时，可以通过 BLX 指令实现子程序的调用和处理器工作状态的切换。同时，子程序的返回可以通过将寄存器 R14 值复制到 PC 中来完成。

4、BX 指令

BX 指令的格式为：

BX{条件} 目标地址

BX 指令跳转到指令中所指定的目标地址，目标地址处的指令既可以是 ARM 指令，也可以是 Thumb 指令。

3.3.2 数据处理指令

数据处理指令可分为数据传送指令、算术逻辑运算指令和比较指令等。

数据传送指令用于在寄存器和存储器之间进行数据的双向传输。

算术逻辑运算指令完成常用的算术与逻辑的运算，该类指令不但将运算结果保存在目的寄存器中，同时更新 CPSR 中的相应条件标志位。

比较指令不保存运算结果，只更新 CPSR 中相应的条件标志位。

数据处理指令包括：

— MOV 数据传送指令
— MVN 数据取反传送指令
— CMP 比较指令
— CMN 反值比较指令
— TST 位测试指令
— TEQ 相等测试指令
— ADD 加法指令
— ADC 带进位加法指令
— SUB 减法指令
— SBC 带借位减法指令
— RSB 逆向减法指令
— RSC 带借位的逆向减法指令
— AND 逻辑与指令
— ORR 逻辑或指令
— EOR 逻辑异或指令
— BIC 位清除指令

1、 MOV 指令

MOV 指令的格式为：

MOV{条件}{S} 目的寄存器，源操作数

MOV 指令可完成从另一个寄存器、被移位的寄存器或将一个立即数加载到目的寄存器。其中 S 选项决定指令的操作是否影响 CPSR 中条件标志位的值，当没有 S 时指令不更新 CPSR 中条件标志位的值。

指令示例：

```
MOV   R1，R0          ；将寄存器 R0 的值传送到寄存器 R1
MOV   PC，R14         ；将寄存器 R14 的值传送到 PC，常用于子程序返回
MOV   R1，R0，LSL＃3  ；将寄存器 R0 的值左移 3 位后传送到 R1
```

2、 MVN 指令

MVN 指令的格式为：

MVN{条件}{S} 目的寄存器，源操作数

MVN 指令可完成从另一个寄存器、被移位的寄存器、或将一个立即数加载到目的寄存器。与 MOV 指令不同之处是在传送之前按位被取反了，即把一个被取反的值传送到目的寄存器中。其中 S 决定指令的操作是否影响 CPSR 中条件标志位的值，当没有 S 时指令不更新 CPSR 中条件标志位的值。

指令示例：

```
MVN   R0，＃0         ；将立即数 0 取反传送到寄存器 R0 中，完成后 R0=-1
```

3、 CMP 指令

CMP 指令的格式为：

CMP{条件} 操作数 1，操作数 2

CMP 指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行比较，同时更新 CPSR 中条件标志位的值。该指令进行一次减法运算，但不存储结果，只更改条件标志位。标志位表示的是操作数 1 与操作数 2 的关系(大、小、相等)，例如，当操作数 1 大于操作操作数 2，则此后的有 GT 后缀的指令将可以执行。

指令示例：

```
CMP   R1，R0          ；将寄存器 R1 的值与寄存器 R0 的值相减，并根据结果设置 CPSR 的标志位
CMP   R1，＃100       ；将寄存器 R1 的值与立即数 100 相减，并根据结果设置 CPSR 的标志位
```

4、 CMN 指令

CMN 指令的格式为：

CMN{条件} 操作数 1，操作数 2

CMN 指令用于把一个寄存器的内容和另一个寄存器的内容或立即数取反后进行比较，同时更新 CPSR 中条件标志位的值。该指令实际完成操作数 1 和操作数 2 相加，并根据结果更改条件标志位。

指令示例：

```
CMN   R1，R0          ；将寄存器 R1 的值与寄存器 R0 的值相加，并根据结果设置 CPSR 的标志位
CMN   R1，＃100       ；将寄存器 R1 的值与立即数 100 相加，并根据结果设置 CPSR 的标志位
```

5、 TST 指令

TST 指令的格式为：

TST{条件} 操作数 1，操作数 2

TST 指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行按位的与运算，并根据运算结果更新 CPSR 中条件标志位的值。操作数 1 是要测试的数据，而操作数 2 是一个位掩码，该指令一般用来检测是否设置了特定的位。

指令示例：

```
TST   R1，＃%1        ；用于测试在寄存器 R1 中是否设置了最低位（%表示二进制数）
TST   R1，＃0xffe     ；将寄存器 R1 的值与立即数 0xffe 按位与，并根据结果设置 CPSR 的标志位
```

6、 TEQ 指令

TEQ 指令的格式为：

TEQ{条件} 操作数 1，操作数 2

TEQ 指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行按位的异或运算，并根据运算结果更新 CPSR 中条件标志位的值。该指令通常用于比较操作数 1 和操作数 2 是否相等。

指令示例：

```
TEQ   R1，R2          ；将寄存器 R1 的值与寄存器 R2 的值按位异或，并根据结果设置 CPSR 的标志位
```

7、 ADD 指令

ADD 指令的格式为：

ADD{条件}{S} 目的寄存器，操作数 1，操作数 2

ADD 指令用于把两个操作数相加，并将结果存放到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即数。

指令示例：

```
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 指令的格式为：

ADC{条件}{S} 目的寄存器，操作数 1，操作数 2

ADC 指令用于把两个操作数相加，再加上 CPSR 中的 C 条件标志位的值，并将结果存放到目的寄存器中。它使用一个进位标志位，这样就可以做比 32 位大的数的加法，注意不要忘记设置 S 后缀来更改进位标志。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即数。

以下指令序列完成两个 128 位数的加法，第一个数由高到低存放在寄存器 R7～R4，第二个数由高到低存放在寄存器 R11～R8，运算结果由高到低存放在寄存器 R3～R0：

```
ADDS  R0，R4，R8          ；  加低端的字
ADCS  R1，R5，R9          ；  加第二个字，带进位
ADCS  R2，R6，R10         ；  加第三个字，带进位
ADC   R3，R7，R11         ；  加第四个字，带进位
```

9、 SUB 指令

SUB 指令的格式为：

SUB{条件}{S} 目的寄存器，操作数 1，操作数 2

SUB 指令用于把操作数 1 减去操作数 2，并将结果存放到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令可用于有符号数或无符号数的减法运算。

指令示例：

```
SUB   R0，R1，R2              ；  R0 = R1 - R2
SUB   R0，R1，#256            ；  R0 = R1 - 256
SUB   R0，R2，R3，LSL#1       ；  R0 = R2 - (R3 << 1)
```

10、SBC 指令

SBC 指令的格式为：

SBC{条件}{S} 目的寄存器，操作数 1，操作数 2

SBC 指令用于把操作数 1 减去操作数 2，再减去 CPSR 中的 C 条件标志位的反码，并将结果存放到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令使用进位标志来表示借位，这样就可以做大于 32 位的减法，注意不要忘记设置 S 后缀来更改进位标志。该指令可用于有符号数或无符号数的减法运算。

指令示例：

```
SUBS  R0，R1，R2              ；  R0 = R1 - R2 - ！C，并根据结果设置 CPSR 的进位标志位
```

11、RSB 指令

RSB 指令的格式为：

RSB{条件}{S} 目的寄存器，操作数 1，操作数 2

RSB 指令称为逆向减法指令，用于把操作数 2 减去操作数 1，并将结果存放到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令可用于有符号数或无符号数的减法运算。

指令示例：

```
RSB   R0，R1，R2              ；  R0 = R2 – R1
RSB   R0，R1，#256            ；  R0 = 256 – R1
RSB   R0，R2，R3，LSL#1       ；  R0 = (R3 << 1) - R2
```

12、RSC 指令

RSC 指令的格式为：

RSC{条件}{S} 目的寄存器，操作数 1，操作数 2

RSC 指令用于把操作数 2 减去操作数 1，再减去 CPSR 中的 C 条件标志位的反码，并将结果存放到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令使用进位标志来表示借位，这样就可以做大于 32 位的减法，注意不要忘记设置 S 后缀来更改进位标志。该指令可用于有符号数或无符号数的减法运算。

指令示例：

```
RSC   R0，R1，R2              ；  R0 = R2 – R1 - ！C
```

13、AND 指令

AND 指令的格式为：

AND{条件}{S} 目的寄存器，操作数 1，操作数 2

AND 指令用于在两个操作数上进行逻辑与运算，并把结果放置到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令常用于屏蔽操作数 1 的某些位。

指令示例：

```
AND   R0，R0，＃3             ；  该指令保持 R0 的 0、1 位，其余位清零。
```

14、ORR 指令

ORR 指令的格式为：

ORR{条件}{S} 目的寄存器，操作数 1，操作数 2

ORR 指令用于在两个操作数上进行逻辑或运算，并把结果放置到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令常用于设置操作数 1 的某些位。

指令示例：

```
ORR   R0，R0，＃3              ；  该指令设置 R0 的 0、1 位，其余位保持不变。
```

15、EOR 指令

EOR 指令的格式为：

EOR{条件}{S} 目的寄存器，操作数 1，操作数 2

EOR 指令用于在两个操作数上进行逻辑异或运算，并把结果放置到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即数。该指令常用于反转操作数 1 的某些位。

指令示例：

```
EOR   R0，R0，＃3              ；  该指令反转 R0 的 0、1 位，其余位保持不变。
```

16、BIC 指令

BIC 指令的格式为：

BIC{条件}{S} 目的寄存器，操作数 1，操作数 2

BIC 指令用于清除操作数 1 的某些位，并把结果放置到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即数。操作数 2 为 32 位的掩码，如果在掩码中设置了某一位，则清除这一位。未设置的掩码位保持不变。

指令示例：

```
BIC   R0，R0，＃%1011          ；  该指令清除 R0 中的位 0、1、和 3，其余的位保持不变。
```

3.3.3 乘法指令与乘加指令

ARM 微处理器支持的乘法指令与乘加指令共有 6 条，可分为运算结果为 32 位和运算结果为 64 位两类，与前面的数据处理指令不同，指令中的所有操作数、目的寄存器必须为通用寄存器，不能对操作数使用立即数或被移位的寄存器，同时，目的寄存器和操作数 1 必须是不同的寄存器。

乘法指令与乘加指令共有以下 6 条：

- MUL　　32 位乘法指令
- MLA　　32 位乘加指令
- SMULL　64 位有符号数乘法指令
- SMLAL　64 位有符号数乘加指令
- UMULL　64 位无符号数乘法指令
- UMLAL　64 位无符号数乘加指令

1、MUL 指令

MUL 指令的格式为：

MUL{条件}{S} 目的寄存器，操作数 1，操作数 2

MUL 指令完成将操作数 1 与操作数 2 的乘法运算，并把结果放置到目的寄存器中，同时可以根据运算结果设置 CPSR 中相应的条件标志位。其中，操作数 1 和操作数 2 均为 32 位的有符号数或无符号数。

指令示例：

```
MUL   R0，R1，R2       ；R0 = R1 × R2
MULS  R0，R1，R2       ；R0 = R1 × R2，同时设置 CPSR 中的相关条件标志位
```

2、MLA 指令

MLA 指令的格式为：

MLA{条件}{S} 目的寄存器，操作数 1，操作数 2，操作数 3

MLA 指令完成将操作数 1 与操作数 2 的乘法运算，再将乘积加上操作数 3，并把结果放置到目的寄存器中，同时可以根据运算结果设置 CPSR 中相应的条件标志位。其中，操作数 1 和操作数 2 均为 32 位的有符号数或无符号数。

指令示例：

```
MLA   R0，R1，R2，R3        ；R0 = R1 × R2 + R3
MLAS  R0，R1，R2，R3        ；R0 = R1 × R2 + R3，同时设置 CPSR 中的相关条件标志位
```

3、SMULL 指令

SMULL 指令的格式为：

SMULL{条件}{S}　目的寄存器 Low，目的寄存器低 High，操作数 1，操作数 2

SMULL 指令完成将操作数 1 与操作数 2 的乘法运算，并把结果的低 32 位放置到目的寄存器 Low 中，结果的高 32 位放置到目的寄存器 High 中，同时可以根据运算结果设置 CPSR 中相应的条件标志位。其中，操作数 1 和操作数 2 均为 32 位的有符号数。

指令示例：

```
SMULL R0，R1，R2，R3        ；R0 = （R2 × R3）的低 32 位
                            ；R1 = （R2 × R3）的高 32 位
```

4、SMLAL 指令

SMLAL 指令的格式为：

SMLAL{条件}{S}　目的寄存器 Low，目的寄存器低 High，操作数 1，操作数 2

SMLAL 指令完成将操作数 1 与操作数 2 的乘法运算，并把结果的低 32 位同目的寄存器 Low 中的值相加后又放置到目的寄存器 Low 中，结果的高 32 位同目的寄存器 High 中的值相加后又放置到目的寄存器 High 中，同时可以根据运算结果设置 CPSR 中相应的条件标志位。其中，操作数 1 和操作数 2 均为 32 位的有符号数。

对于目的寄存器 Low，在指令执行前存放 64 位加数的低 32 位，指令执行后存放结果的低 32 位。

对于目的寄存器 High，在指令执行前存放 64 位加数的高 32 位，指令执行后存放结果的高 32 位。

指令示例：

```
SMLAL R0，R1，R2，R3        ；R0 = （R2 × R3）的低 32 位 ＋ R0
                            ；R1 = （R2 × R3）的高 32 位 ＋ R1
```

5、UMULL 指令

UMULL 指令的格式为：

UMULL{条件}{S}　目的寄存器 Low，目的寄存器低 High，操作数 1，操作数 2

UMULL 指令完成将操作数 1 与操作数 2 的乘法运算，并把结果的低 32 位放置到目的寄存器 Low 中，结果的高 32 位放置到目的寄存器 High 中，同时可以根据运算结果设置 CPSR 中相应的条件标志位。其中，操作数 1 和操作数 2 均为 32 位的无符号数。

指令示例：

```
UMULL R0，R1，R2，R3        ；R0 = （R2 × R3）的低 32 位
                            ；R1 = （R2 × R3）的高 32 位
```

6、UMLAL 指令

UMLAL 指令的格式为：

UMLAL{条件}{S}　目的寄存器 Low，目的寄存器低 High，操作数 1，操作数 2

UMLAL 指令完成将操作数 1 与操作数 2 的乘法运算，并把结果的低 32 位同目的寄存器 Low 中的值相加后又放置到目的寄存器 Low 中，结果的高 32 位同目的寄存器 High 中的值相加后又放置到目的寄存器 High 中，同时可以根据运算结果设置 CPSR 中相应的条件标志位。其中，操作数 1 和操作数 2 均为 32 位的无符号数。

对于目的寄存器 Low，在指令执行前存放 64 位加数的低 32 位，指令执行后存放结果的低 32 位。

对于目的寄存器 High，在指令执行前存放 64 位加数的高 32 位，指令执行后存放结果的高 32 位。

指令示例：

```
UMLAL R0，R1，R2，R3        ；R0 =（R2 × R3）的低 32 位 + R0
                            ；R1 =（R2 × R3）的高 32 位 + R1
```

3.3.4 程序状态寄存器访问指令

ARM 微处理器支持程序状态寄存器访问指令，用于在程序状态寄存器和通用寄存器之间传送数据，程序状态寄存器访问指令包括以下两条：

— MRS 程序状态寄存器到通用寄存器的数据传送指令

— MSR 通用寄存器到程序状态寄存器的数据传送指令

1、MRS 指令

MRS 指令的格式为：

MRS{条件}　通用寄存器，程序状态寄存器（CPSR 或 SPSR）

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。该指令一般用在以下几种情况：

－ 当需要改变程序状态寄存器的内容时，可用 MRS 将程序状态寄存器的内容读入通用寄存器，修改后再写回程序状态寄存器。

－ 当在异常处理或进程切换时，需要保存程序状态寄存器的值，可先用该指令读出程序状态寄存器的值，然后保存。

指令示例：

```
MRS   R0，CPSR          ；传送 CPSR 的内容到 R0
MRS   R0，SPSR          ；传送 SPSR 的内容到 R0
```

2、MSR 指令

MSR 指令的格式为：

MSR{条件}　程序状态寄存器（CPSR 或 SPSR）_<域>，操作数

MSR 指令用于将操作数的内容传送到程序状态寄存器的特定域中。其中，操作数可以为通用寄存器或立即数。<域>用于设置程序状态寄存器中需要操作的位，32 位的程序状态寄存器可分为 4 个域：

位[31：24]为条件标志位域，用 f 表示；

位[23：16]为状态位域，用 s 表示；

位[15：8]为扩展位域，用 x 表示；

位[7：0]为控制位域，用 c 表示；

该指令通常用于恢复或改变程序状态寄存器的内容，在使用时，一般要在 MSR 指令中指明将要操作的域。

指令示例：

```
MSR   CPSR，R0          ；传送 R0 的内容到 CPSR
MSR   SPSR，R0          ；传送 R0 的内容到 SPSR
MSR   CPSR_c，R0        ；传送 R0 的内容到 SPSR，但仅仅修改 CPSR 中的控制位域
```

3.3.5 加载/存储指令

ARM 微处理器支持加载/存储指令用于在寄存器和存储器之间传送数据，加载指令用于将存储器中的数据传送到寄存器，存储指令则完成相反的操作。常用的加载存储指令如下：

— LDR　字数据加载指令

— LDRB　字节数据加载指令

— LDRH　半字数据加载指令

— STR　字数据存储指令

— STRB　字节数据存储指令

— STRH　半字数据存储指令

1、LDR 指令

LDR 指令的格式为：

LDR{条件} 目的寄存器，<存储器地址>

LDR 指令用于从存储器中将一个 32 位的字数据传送到目的寄存器中。该指令通常用于从存储器中读取 32 位的字数据到通用寄存器，然后对数据进行处理。当程序计数器 PC 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。该指令在程序设计中比较常用，且寻址方式灵活多样，请读者认真掌握。

指令示例：

```
LDR   R0，[R1]                ；将存储器地址为 R1 的字数据读入寄存器 R0。
LDR   R0，[R1，R2]            ；将存储器地址为 R1+R2 的字数据读入寄存器 R0。
LDR   R0，[R1，＃8]           ；将存储器地址为 R1+8 的字数据读入寄存器 R0。
LDR   R0，[R1，R2] !          ；将存储器地址为 R1+R2 的字数据读入寄存器 R0，并将新地址 R1
＋R2 写入 R1。
LDR   R0，[R1，＃8] !         ；将存储器地址为 R1+8 的字数据读入寄存器 R0，并将新地址 R1
＋8 写入 R1。
LDR   R0，[R1]，R2            ；将存储器地址为 R1 的字数据读入寄存器 R0，并将新地址 R1＋
R2 写入 R1。
LDR   R0，[R1，R2，LSL＃2]!   ；将存储器地址为 R1＋R2×4 的字数据读入寄存器 R0，并将新地
址 R1＋R2×4 写入 R1。
LDR   R0，[R1]，R2，LSL＃2    ；将存储器地址为 R1 的字数据读入寄存器 R0，并将新地址 R1＋
R2×4 写入 R1。
```

2、LDRB 指令

LDRB 指令的格式为：

LDR{条件}B 目的寄存器，<存储器地址>

LDRB 指令用于从存储器中将一个 8 位的字节数据传送到目的寄存器中，同时将寄存器的高 24 位清零。该指令通常用于从存储器中读取 8 位的字节数据到通用寄存器，然后对数据进行处理。当程序计数器 PC 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。

指令示例：

```
LDRB  R0，[R1]          ；将存储器地址为 R1 的字节数据读入寄存器 R0，并将 R0 的高 24 位清零。
LDRB  R0，[R1，＃8]     ;将存储器地址为 R1＋8 的字节数据读入寄存器 R0,并将 R0 的高 24 位清零。
```

3、LDRH 指令

LDRH 指令的格式为：

LDR{条件}H 目的寄存器，<存储器地址>

LDRH 指令用于从存储器中将一个 16 位的半字数据传送到目的寄存器中，同时将寄存器的高 16 位清零。该指令通常用于从存储器中读取 16 位的半字数据到通用寄存器，然后对数据进行处理。当程序计数器 PC 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。

指令示例：

```
LDRH  R0，[R1]          ；将存储器地址为 R1 的半字数据读入寄存器 R0，并将 R0 的高 16 位清零。
LDRH  R0，[R1，＃8]     ;将存储器地址为 R1＋8 的半字数据读入寄存器 R0,并将 R0 的高 16 位清零。
LDRH  R0，[R1，R2]      ；将存储器地址为 R1＋R2 的半字数据读入寄存器 R0，并将 R0 的高 16 位清
零。
```

4、STR 指令

STR 指令的格式为：

STR{条件} 源寄存器，<存储器地址>

STR 指令用于从源寄存器中将一个 32 位的字数据传送到存储器中。该指令在程序设计中比较常用，且寻址方式灵活多样，使用方式可参考指令 LDR。

指令示例：

```
STR   R0，[R1]，＃8     ;将 R0 中的字数据写入以 R1 为地址的存储器中，并将新地址 R1＋8 写入 R1。
STR   R0，[R1，＃8]     ; 将 R0 中的字数据写入以 R1＋8 为地址的存储器中。
```

5、STRB 指令

STRB 指令的格式为：

STR{条件}B 源寄存器，<存储器地址>

STRB 指令用于从源寄存器中将一个 8 位的字节数据传送到存储器中。该字节数据为源寄存器中的低 8 位。

指令示例：

```
STRB  R0，[R1]          ; 将寄存器 R0 中的字节数据写入以 R1 为地址的存储器中。
STRB  R0，[R1，＃8]     ; 将寄存器 R0 中的字节数据写入以 R1＋8 为地址的存储器中。
```

6、STRH 指令

STRH 指令的格式为：

STR{条件}H 源寄存器，<存储器地址>

STRH 指令用于从源寄存器中将一个 16 位的半字数据传送到存储器中。该半字数据为源寄存器中的低 16 位。

指令示例：

```
STRH  R0，[R1]          ; 将寄存器 R0 中的半字数据写入以 R1 为地址的存储器中。
STRH  R0，[R1，＃8]     ; 将寄存器 R0 中的半字数据写入以 R1＋8 为地址的存储器中。
```

3.3.6 批量数据加载/存储指令

ARM 微处理器所支持批量数据加载/存储指令可以一次在一片连续的存储器单元和多个寄存器之间传送数据，批量加载指令用于将一片连续的存储器中的数据传送到多个寄存器，批量数据存储指令则完成相反的操作。常用的加载存储指令如下：

— LDM　　批量数据加载指令

— STM　　批量数据存储指令

LDM（或 STM）指令

LDM（或 STM）指令的格式为：

LDM（或 STM）{条件}{类型} 基址寄存器{！}，寄存器列表{^}

LDM（或 STM）指令用于从由基址寄存器所指示的一片连续存储器到寄存器列表所指示的多个寄存器之间传送数据，该指令的常见用途是将多个寄存器的内容入栈或出栈。其中，{类型}为以下几种情况：

IA　每次传送后地址加 1；

IB　每次传送前地址加 1；

DA　每次传送后地址减 1；

DB　每次传送前地址减 1；

FD　满递减堆栈；

ED　空递减堆栈；

FA　满递增堆栈；

EA　空递增堆栈；

{！}为可选后缀，若选用该后缀，则当数据传送完毕之后，将最后的地址写入基址寄存器，否则基址寄存器的内容不改变。

基址寄存器不允许为 R15，寄存器列表可以为 R0～R15 的任意组合。

{^}为可选后缀，当指令为 LDM 且寄存器列表中包含 R15，选用该后缀时表示：除了正常的数据传送之外，还将 SPSR 复制到 CPSR。同时，该后缀还表示传入或传出的是用户模式下的寄存器，而不是当前模式下的寄存器。

指令示例：

```
STMFD  R13!，{R0，R4-R12，LR}      ；将寄存器列表中的寄存器（R0，R4 到 R12，LR）存入堆栈。
LDMFD  R13!，{R0，R4-R12，PC}      ；将堆栈内容恢复到寄存器（R0，R4 到 R12，LR）。
```

3.3.7 数据交换指令

ARM 微处理器所支持数据交换指令能在存储器和寄存器之间交换数据。数据交换指令有如下两条：

— SWP 字数据交换指令

— SWPB 字节数据交换指令

1、SWP 指令

SWP 指令的格式为：

SWP{条件} 目的寄存器，源寄存器 1，[源寄存器 2]

SWP 指令用于将源寄存器 2 所指向的存储器中的字数据传送到目的寄存器中，同时将源寄存器 1 中的字数据传送到源寄存器 2 所指向的存储器中。显然，当源寄存器 1 和目的寄存器为同一个寄存器时，指令交换该寄存器和存储器的内容。

指令示例：

```
SWP   R0，R1，[R2]          ；将 R2 所指向的存储器中的字数据传送到 R0，同时将 R1 中的字数据传送到 R2 所指向的存储单元。
SWP   R0，R0，[R1]          ；该指令完成将 R1 所指向的存储器中的字数据与 R0 中的字数据交换。
```

2、SWPB 指令

SWPB 指令的格式为：

SWP{条件}B 目的寄存器，源寄存器 1，[源寄存器 2]

SWPB 指令用于将源寄存器 2 所指向的存储器中的字节数据传送到目的寄存器中，目的寄存器的高 24 清零，同时将源寄存器 1 中的字节数据传送到源寄存器 2 所指向的存储器中。显然，当源寄存器 1 和目的寄存器为同一个寄存器时，指令交换该寄存器和存储器的内容。

指令示例：

```
SWPB  R0，R1，[R2]          ；将 R2 所指向的存储器中的字节数据传送到 R0，R0 的高 24 位清零，同时将 R1 中的低 8 位数据传送到 R2 所指向的存储单元。
SWPB  R0，R0，[R1]          ；该指令完成将 R1 所指向的存储器中的字节数据与 R0 中的低 8 位数据交换。
```

3.3.8 移位指令（操作）

ARM 微处理器内嵌的桶型移位器（Barrel Shifter），支持数据的各种移位操作，移位操作在 ARM 指令集中不作为单独的指令使用，它只能作为指令格式中是一个字段，在汇编语言中表示为指令中的选项。例如，数据处理指令的第二个操作数为寄存器时，就可以加入移位操作选项对它进行各种移位操作。移位操作包括如下 6 种类型，ASL 和 LSL 是等价的，可以自由互换：

— LSL 逻辑左移

— ASL 算术左移

— LSR 逻辑右移

— ASR 算术右移

— ROR 循环右移

— RRX 带扩展的循环右移

1、LSL（或 ASL）操作

LSL（或 ASL）操作的格式为：

通用寄存器，LSL（或 ASL） 操作数

LSL（或 ASL）可完成对通用寄存器中的内容进行逻辑（或算术）的左移操作，按操作数所指定的数量向左移位，低位用零来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

操作示例：

```
MOV    R0, R1, LSL#2      ；将 R1 中的内容左移两位后传送到 R0 中。
```

2、LSR 操作

LSR 操作的格式为：

通用寄存器，LSR 操作数

LSR 可完成对通用寄存器中的内容进行右移的操作，按操作数所指定的数量向右移位，左端用零来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

操作示例：

```
MOV    R0, R1, LSR#2      ；将 R1 中的内容右移两位后传送到 R0 中，左端用零来填充。
```

3、ASR 操作

ASR 操作的格式为：

通用寄存器，ASR 操作数

ASR 可完成对通用寄存器中的内容进行右移的操作，按操作数所指定的数量向右移位，左端用第 31 位的值来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

操作示例：

```
MOV    R0, R1, ASR#2      ；将 R1 中的内容右移两位后传送到 R0 中，左端用第 31 位的值来填充。
```

4、ROR 操作

ROR 操作的格式为：

通用寄存器，ROR 操作数

ROR 可完成对通用寄存器中的内容进行循环右移的操作，按操作数所指定的数量向右循环移位，左端用右端移出的位来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。显然，当进行 32 位的循环右移操作时，通用寄存器中的值不改变。

操作示例：

```
MOV    R0, R1, ROR#2      ；将 R1 中的内容循环右移两位后传送到 R0 中。
```

5、RRX 操作

RRX 操作的格式为：

通用寄存器，RRX 操作数

RRX 可完成对通用寄存器中的内容进行带扩展的循环右移的操作，按操作数所指定的数量向右循环移位，左端用进位标志位 C 来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

操作示例：

```
MOV    R0, R1, RRX#2      ；将 R1 中的内容进行带扩展的循环右移两位后传送到 R0 中。
```

3.3.9 协处理器指令

ARM 微处理器可支持多达 16 个协处理器，用于各种协处理操作，在程序执行的过程中，每个协处理器只执行针对自身的协处理指令，忽略 ARM 处理器和其他协处理器的指令。

ARM 的协处理器指令主要用于 ARM 处理器初始化 ARM 协处理器的数据处理操作，以及在 ARM 处理器的寄存器和协处理器的寄存器之间传送数据，和在 ARM 协处理器的寄存器和存储器之间传送数据。ARM 协处理器指令包括以下 5 条：

- CDP　协处理器数操作指令
- LDC　协处理器数据加载指令
- STC　协处理器数据存储指令
- MCR　ARM 处理器寄存器到协处理器寄存器的数据传送指令

— MRC 协处理器寄存器到 ARM 处理器寄存器的数据传送指令

1、CDP 指令

CDP 指令的格式为：

CDP{条件} 协处理器编码，协处理器操作码 1，目的寄存器，源寄存器 1，源寄存器 2，协处理器操作码 2。

CDP 指令用于 ARM 处理器通知 ARM 协处理器执行特定的操作,若协处理器不能成功完成特定的操作，则产生未定义指令异常。其中协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作，目的寄存器和源寄存器均为协处理器的寄存器，指令不涉及 ARM 处理器的寄存器和存储器。

指令示例：

```
CDP   P3，2，C12，C10，C3，4          ；该指令完成协处理器 P3 的初始化
```

2、LDC 指令

LDC 指令的格式为：

LDC{条件}{L} 协处理器编码,目的寄存器，[源寄存器]

LDC 指令用于将源寄存器所指向的存储器中的字数据传送到目的寄存器中,若协处理器不能成功完成传送操作，则产生未定义指令异常。其中，{L}选项表示指令为长读取操作，如用于双精度数据的传输。

指令示例：

```
LDC   P3，C4，[R0]          ；将 ARM 处理器的寄存器 R0 所指向的存储器中的字数据传送到协处理器 P3 的寄存器 C4 中。
```

3、STC 指令

STC 指令的格式为：

STC{条件}{L} 协处理器编码,源寄存器，[目的寄存器]

STC 指令用于将源寄存器中的字数据传送到目的寄存器所指向的存储器中,若协处理器不能成功完成传送操作，则产生未定义指令异常。其中，{L}选项表示指令为长读取操作，如用于双精度数据的传输。

指令示例：

```
STC   P3，C4，[R0]          ；将协处理器 P3 的寄存器 C4 中的字数据传送到 ARM 处理器的寄存器 R0 所指向的存储器中。
```

4、MCR 指令

MCR 指令的格式为：

MCR{条件} 协处理器编码，协处理器操作码 1，源寄存器，目的寄存器 1，目的寄存器 2，协处理器操作码 2。

MCR 指令用于将 ARM 处理器寄存器中的数据传送到协处理器寄存器中,若协处理器不能成功完成操作，则产生未定义指令异常。其中协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作，源寄存器为 ARM 处理器的寄存器，目的寄存器 1 和目的寄存器 2 均为协处理器的寄存器。

指令示例：

```
MCR   P3，3，R0，C4，C5，6    ；该指令将 ARM 处理器寄存器 R0 中的数据传送到协处理器 P3 的寄存器 C4 和 C5 中。
```

5、MRC 指令

MRC 指令的格式为：

MRC{条件} 协处理器编码，协处理器操作码 1，目的寄存器，源寄存器 1，源寄存器 2，协处理器操作码 2。

MRC 指令用于将协处理器寄存器中的数据传送到 ARM 处理器寄存器中,若协处理器不能成功完成操作，则产生未定义指令异常。其中协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作，目的寄存器为 ARM 处理器的寄存器，源寄存器 1 和源寄存器 2 均为协处理器的寄存器。

指令示例：

```
MRC   P3，3，R0，C4，C5，6    ；该指令将协处理器 P3 的寄存器中的数据传送到 ARM 处理器寄存器
```

中。

3.3.10 异常产生指令

ARM 微处理器所支持的异常指令有如下两条：

— SWI 软件中断指令

— BKPT 断点中断指令

1、SWI 指令

SWI 指令的格式为：

SWI{条件} 24 位的立即数

SWI 指令用于产生软件中断，以便用户程序能调用操作系统的系统例程。操作系统在 SWI 的异常处理程序中提供相应的系统服务，指令中 24 位的立即数指定用户程序调用系统例程的类型，相关参数通过通用寄存器传递，当指令中 24 位的立即数被忽略时，用户程序调用系统例程的类型由通用寄存器 R0 的内容决定，同时，参数通过其他通用寄存器传递。

指令示例：

```
SWI   0x02          ；该指令调用操作系统编号位 02 的系统例程。
```

2、BKPT 指令

BKPT 指令的格式为：

BKPT 16 位的立即数

BKPT 指令产生软件断点中断，可用于程序的调试。

3.4 Thumb 指令及应用

为兼容数据总线宽度为 16 位的应用系统，ARM 体系结构除了支持执行效率很高的 32 位 ARM 指令集以外，同时支持 16 位的 Thumb 指令集。Thumb 指令集是 ARM 指令集的一个子集，允许指令编码为 16 位的长度。与等价的 32 位代码相比较，Thumb 指令集在保留 32 代码优势的同时，大大的节省了系统的存储空间。

所有的 Thumb 指令都有对应的 ARM 指令，而且 Thumb 的编程模型也对应于 ARM 的编程模型，在应用程序的编写过程中，只要遵循一定调用的规则，Thumb 子程序和 ARM 子程序就可以互相调用。当处理器在执行 ARM 程序段时，称 ARM 处理器处于 ARM 工作状态，当处理器在执行 Thumb 程序段时，称 ARM 处理器处于 Thumb 工作状态。

与 ARM 指令集相比较，Thumb 指令集中的数据处理指令的操作数仍然是 32 位，指令地址也为 32 位，但 Thumb 指令集为实现 16 位的指令长度，舍弃了 ARM 指令集的一些特性，如大多数的 Thumb 指令是无条件执行的，而几乎所有的 ARM 指令都是有条件执行的；大多数的 Thumb 数据处理指令的目的寄存器与其中一个源寄存器相同。

由于 Thumb 指令的长度为 16 位，即只用 ARM 指令一半的位数来实现同样的功能，所以，要实现特定的程序功能，所需的 Thumb 指令的条数较 ARM 指令多。在一般的情况下，Thumb 指令与 ARM 指令的时间效率和空间效率关系为：

— Thumb 代码所需的存储空间约为 ARM 代码的 60%～70%

— Thumb 代码使用的指令数比 ARM 代码多约 30%～40%

— 若使用 32 位的存储器，ARM 代码比 Thumb 代码快约 40%

— 若使用 16 位的存储器，Thumb 代码比 ARM 代码快约 40%～50%

— 与 ARM 代码相比较，使用 Thumb 代码，存储器的功耗会降低约 30%

显然，ARM 指令集和 Thumb 指令集各有其优点，若对系统的性能有较高要求，应使用 32 位的存储系统和 ARM 指令集，若对系统的成本及功耗有较高要求，则应使用 16 位的存储系统和 Thumb 指令集。当然，若两者结合使用，充分发挥其各自的优点，会取得更好的效果。

3.5 本章小节

本章系统的介绍了 ARM 指令集中的基本指令，以及各指令的应用场合及方法，由基本指令还可以派生出一些新的指令，但使用方法与基本指令类似。与常见的如 X86 体系结构的汇编指令相比较，ARM 指令系统无论是从指令集本身，还是从寻址方式上，都相对复杂一些。

Thumb 指令集作为 ARM 指令集的一个子集，其使用方法与 ARM 指令集类似，在此未作详细的描述，但这并不意味着 Thumb 指令集不如 ARM 指令集重要，事实上，他们各自有其自己的应用场合。

第 4 章 ARM 程序设计基础

ARM 编译器一般都支持汇编语言的程序设计和 C/C++语言的程序设计，以及两者的混合编程。本章介绍 ARM 程序设计的一些基本概念，如 ARM 汇编语言的伪指令、汇编语言的语句格式和汇编语言的程序结构等，同时介绍 C/C＋＋和汇编语言的混合编程等问题。

本章的主要内容：

- ARM 编译器所支持的伪指令
- 汇编语言的语句格式
- 汇编语言的程序结构
- 相关的程序示例

4.1 ARM 汇编器所支持的伪指令

在 ARM 汇编语言程序里，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指令，他们所完成的操作称为伪操作。伪指令在源程序中的作用是为完成汇编程序作各种准备工作的，这些伪指令仅在汇编过程中起作用，一旦汇编结束，伪指令的使命就完成。

在 ARM 的汇编程序中，有如下几种伪指令：符号定义伪指令、数据定义伪指令、汇编控制伪指令、宏指令以及其他伪指令。

4.1.1 符号定义（Symbol Definition）伪指令

符号定义伪指令用于定义 ARM 汇编程序中的变量、对变量赋值以及定义寄存器的别名等操作。常见的符号定义伪指令有如下几种：

- 用于定义全局变量的 GBLA、GBLL 和 GBLS。
- 用于定义局部变量的 LCLA、LCLL 和 LCLS。
- 用于对变量赋值的 SETA、SETL、SETS。
- 为通用寄存器列表定义名称的 RLIST。

1、 GBLA、GBLL 和 GBLS

语法格式：

GBLA（GBLL 或 GBLS） 全局变量名

GBLA、GBLL 和 GBLS 伪指令用于定义一个 ARM 程序中的全局变量，并将其初始化。其中：

GBLA 伪指令用于定义一个全局的数字变量，并初始化为 0；

GBLL 伪指令用于定义一个全局的逻辑变量，并初始化为 F（假）；

GBLS 伪指令用于定义一个全局的字符串变量，并初始化为空；

由于以上三条伪指令用于定义全局变量，因此在整个程序范围内变量名必须唯一。

使用示例：

```
GBLA    Test1                   ；定义一个全局的数字变量，变量名为 Test1
Test1   SETA    0xaa            ；将该变量赋值为 0xaa
GBLL    Test2                   ；定义一个全局的逻辑变量，变量名为 Test2
Test2   SETL    {TRUE}          ；将该变量赋值为真
GBLS    Test3                   ；定义一个全局的字符串变量，变量名为 Test3
```

```
Test3   SETS    “Testing”       ；将该变量赋值为“Testing”
```

2、 LCLA、LCLL 和 LCLS

语法格式：

LCLA（LCLL 或 LCLS） 局部变量名

LCLA、LCLL 和 LCLS 伪指令用于定义一个 ARM 程序中的局部变量，并将其初始化。其中：

LCLA 伪指令用于定义一个局部的数字变量，并初始化为 0；

LCLL 伪指令用于定义一个局部的逻辑变量，并初始化为 F（假）；

LCLS 伪指令用于定义一个局部的字符串变量，并初始化为空；

以上三条伪指令用于声明局部变量，在其作用范围内变量名必须唯一。

使用示例：

```
LCLA    Test4                   ；声明一个局部的数字变量，变量名为 Test4
Test3   SETA    0xaa            ；将该变量赋值为 0xaa
LCLL    Test5                   ；声明一个局部的逻辑变量，变量名为 Test5
Test4   SETL    {TRUE}          ；将该变量赋值为真
LCLS    Test6                   ；定义一个局部的字符串变量，变量名为 Test6
Test6   SETS    “Testing”       ；将该变量赋值为“Testing”
```

3、 SETA、SETL 和 SETS

语法格式：

变量名 SETA（SETL 或 SETS） 表达式

伪指令 SETA、SETL、SETS 用于给一个已经定义的全局变量或局部变量赋值。

SETA 伪指令用于给一个数学变量赋值；

SETL 伪指令用于给一个逻辑变量赋值；

SETS 伪指令用于给一个字符串变量赋值；

其中，变量名为已经定义过的全局变量或局部变量，表达式为将要赋给变量的值。

使用示例：

```
LCLA    Test3                   ；声明一个局部的数字变量，变量名为 Test3
Test3   SETA    0xaa            ；将该变量赋值为 0xaa
LCLL    Test4                   ；声明一个局部的逻辑变量，变量名为 Test4
Test4   SETL    {TRUE}          ；将该变量赋值为真
```

4、 RLIST

语法格式：

名称 RLIST {寄存器列表}

RLIST 伪指令可用于对一个通用寄存器列表定义名称，使用该伪指令定义的名称可在 ARM 指令 LDM/STM 中使用。在 LDM/STM 指令中，列表中的寄存器访问次序为根据寄存器的编号由低到高，而与列表中的寄存器排列次序无关。

使用示例：

```
RegList RLIST   {R0-R5，R8，R10}；将寄存器列表名称定义为 RegList，可在 ARM 指令 LDM/STM 中通过该名称访问寄存器列表。
```

4.1.2 数据定义（Data Definition）伪指令

数据定义伪指令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。常见的数据定义伪指令有如下几种：

— DCB 用于分配一片连续的字节存储单元并用指定的数据初始化。

— DCW（DCWU） 用于分配一片连续的半字存储单元并用指定的数据初始化。

— DCD（DCDU） 用于分配一片连续的字存储单元并用指定的数据初始化。

— DCFD（DCFDU）用于为双精度的浮点数分配一片连续的字存储单元并用指定的数据初始

化。

— DCFS（DCFSU） 用于为单精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。
— DCQ（DCQU） 用于分配一片以 8 字节为单位的连续的存储单元并用指定的数据初始化。
— SPACE 用于分配一片连续的存储单元
— MAP 用于定义一个结构化的内存表首地址
— FIELD 用于定义一个结构化的内存表的数据域

1、 DCB

语法格式：

标号 DCB 表达式

DCB 伪指令用于分配一片连续的字节存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为 0～255 的数字或字符串。DCB 也可用“=”代替。

使用示例：

```
Str DCB “This is a test!”；分配一片连续的字节存储单元并初始化。
```

2、 DCW（或 DCWU）

语法格式：

标号 DCW（或 DCWU） 表达式

DCW(或 DCWU)伪指令用于分配一片连续的半字存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为程序标号或数字表达式。。

用 DCW 分配的字存储单元是半字对齐的，而用 DCWU 分配的字存储单元并不严格半字对齐。

使用示例：

```
DataTest    DCW     1，2，3 ；分配一片连续的半字存储单元并初始化。
```

3、 DCD（或 DCDU）

语法格式：

标号 DCD（或 DCDU） 表达式

DCD（或 DCDU）伪指令用于分配一片连续的字存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为程序标号或数字表达式。DCD 也可用“&”代替。

用 DCD 分配的字存储单元是字对齐的，而用 DCDU 分配的字存储单元并不严格字对齐。

使用示例：

```
DataTest    DCD     4，5，6 ；分配一片连续的字存储单元并初始化。
```

4、 DCFD（或 DCFDU）

语法格式：

标号 DCFD（或 DCFDU） 表达式

DCFD（或 DCFDU）伪指令用于为双精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化。每个双精度的浮点数占据两个字单元。

用 DCFD 分配的字存储单元是字对齐的，而用 DCFDU 分配的字存储单元并不严格字对齐。

使用示例：

```
FDataTest   DCFD    2E115，-5E7 ；分配一片连续的字存储单元并初始化为指定的双精度数。
```

5、 DCFS（或 DCFSU）

语法格式：

标号 DCFS（或 DCFSU） 表达式

DCFS（或 DCFSU）伪指令用于为单精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化。每个单精度的浮点数占据一个字单元。

用 DCFS 分配的字存储单元是字对齐的，而用 DCFSU 分配的字存储单元并不严格字对齐。

使用示例：

```
FDataTest   DCFS    2E5，-5E－7 ；分配一片连续的字存储单元并初始化为指定的单精度数。
```

6、 DCQ(或 DCQU)

语法格式：

标号　　DCQ（或 DCQU）　　表达式

DCQ（或 DCQU）伪指令用于分配一片以 8 个字节为单位的连续存储区域并用伪指令中指定的表达式初始化。

用 DCQ 分配的存储单元是字对齐的，而用 DCQU 分配的存储单元并不严格字对齐。

使用示例：

```
DataTest    DCQ     100 ；分配一片连续的存储单元并初始化为指定的值。
```

7、 SPACE

语法格式：

标号　　SPACE　表达式

SPACE 伪指令用于分配一片连续的存储区域并初始化为 0。其中，表达式为要分配的字节数。SPACE 也可用“%”代替。

使用示例：

```
DataSpace   SPACE   100 ；分配连续 100 字节的存储单元并初始化为 0。
```

8、 MAP

语法格式：

MAP　　表达式{，基址寄存器}

MAP 伪指令用于定义一个结构化的内存表的首地址。MAP 也可用“^”代替。

表达式可以为程序中的标号或数学表达式，基址寄存器为可选项，当基址寄存器选项不存在时，表达式的值即为内存表的首地址，当该选项存在时，内存表的首地址为表达式的值与基址寄存器的和。

MAP 伪指令通常与 FIELD 伪指令配合使用来定义结构化的内存表。

使用示例：

```
MAP     0x100，R0       ；定义结构化内存表首地址的值为 0x100＋R0。
```

9、 FILED

语法格式：

标号　　FIELD　表达式

FIELD 伪指令用于定义一个结构化内存表中的数据域。FILED 也可用“#”代替。

表达式的值为当前数据域在内存表中所占的字节数。

FIELD 伪指令常与 MAP 伪指令配合使用来定义结构化的内存表。MAP 伪指令定义内存表的首地址，FIELD 伪指令定义内存表中的各个数据域，并可以为每个数据域指定一个标号供其他的指令引用。

注意 MAP 和 FIELD 伪指令仅用于定义数据结构，并不实际分配存储单元。

使用示例：

```
MAP     0x100           ；定义结构化内存表首地址的值为 0x100。
A       FIELD   16      ；定义 A 的长度为 16 字节，位置为 0x100
B       FIELD   32      ；定义 B 的长度为 32 字节，位置为 0x110
S       FIELD   256     ；定义 S 的长度为 256 字节，位置为 0x130
```

4.1.3　汇编控制（Assembly Control）伪指令

汇编控制伪指令用于控制汇编程序的执行流程，常用的汇编控制伪指令包括以下几条：

— IF、ELSE、ENDIF

— WHILE、WEND

— MACRO、MEND

— MEXIT

1、 IF、ELSE、ENDIF

语法格式：

IF 逻辑表达式

指令序列 1

ELSE

指令序列 2

ENDIF

IF、ELSE、ENDIF 伪指令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，则执行指令序列 1，否则执行指令序列 2。其中，ELSE 及指令序列 2 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 1，否则继续执行后面的指令。

IF、ELSE、ENDIF 伪指令可以嵌套使用。

使用示例：

```
GBLL  Test                    ；声明一个全局的逻辑变量，变量名为 Test
……
IF    Test = TRUE
  指令序列 1
ELSE
  指令序列 2
ENDIF
```

2、 WHILE、WEND

语法格式：

WHILE 逻辑表达式

指令序列

WEND

WHILE、WEND 伪指令能根据条件的成立与否决定是否循环执行某个指令序列。当 WHILE 后面的逻辑表达式为真，则执行指令序列，该指令序列执行完毕后，再判断逻辑表达式的值，若为真则继续执行，一直到逻辑表达式的值为假。

WHILE、WEND 伪指令可以嵌套使用。

使用示例：

```
GBLA  Counter                 ；声明一个全局的数学变量，变量名为 Counter
Counter     SETA      3       ；由变量 Counter 控制循环次数
……
WHILE Counter < 10
  指令序列
WEND
```

3、 MACRO、MEND

语法格式：

$标号 宏名 $参数 1，$参数 2，……

指令序列

MEND

MACRO、MEND 伪指令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，宏指令可以使用一个或多个参数，当宏指令被展开时，这些参数被相应的值替换。

宏指令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，可以使用宏指令代替子程序。

包含在 MACRO 和 MEND 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。

MACRO、MEND 伪指令可以嵌套使用。

4、MEXIT

语法格式：

MEXIT

MEXIT 用于从宏定义中跳转出去。

4.1.4 其他常用的伪指令

还有一些其他的伪指令，在汇编程序中经常会被使用，包括以下几条：

- AREA
- ALIGN
- CODE16、CODE32
- ENTRY
- END
- EQU
- EXPORT（或 GLOBAL）
- IMPORT
- EXTERN
- GET（或 INCLUDE）
- INCBIN
- RN
- ROUT

1、AREA

语法格式：

AREA　段名　属性 1，属性 2，……

AREA 伪指令用于定义一个代码段或数据段。其中，段名若以数字开头，则该段名需用“|”括起来，如|1_test|。

属性字段表示该代码段（或数据段）的相关属性，多个属性用逗号分隔。常用的属性如下：

- CODE 属性：用于定义代码段，默认为 READONLY。
- DATA 属性：用于定义数据段，默认为 READWRITE。
- READONLY 属性：指定本段为只读，代码段默认为 READONLY。
- READWRITE 属性：指定本段为可读可写，数据段的默认属性为 READWRITE。
- ALIGN 属性：使用方式为 ALIGN 表达式。在默认时，ELF（可执行连接文件）的代码段和数据段是按字对齐的，表达式的取值范围为 0～31，相应的对齐方式为 2 表达式次方。
- COMMON 属性：该属性定义一个通用的段，不包含任何的用户代码和数据。各源文件中同名的 COMMON 段共享同一段存储单元。

一个汇编语言程序至少要包含一个段，当程序太长时，也可以将程序分为多个代码段和数据段。

使用示例：

```
AREA  Init，CODE，READONLY
指令序列
；该伪指令定义了一个代码段，段名为 Init，属性为只读
```

2、 ALIGN

语法格式：

ALIGN {表达式{，偏移量}}

ALIGN 伪指令可通过添加填充字节的方式，使当前位置满足一定的对其方式|。其中，表达式的值用于指定对齐方式，可能的取值为 2 的幂，如 1、2、4、8、16 等。若未指定表达式，则将当前位置对齐到下一个字的位置。偏移量也为一个数字表达式，若使用该字段，则当前位置的对齐方式为：2 的表达式次幂＋偏移量。

使用示例：

```
AREA  Init，CODE，READONLY，ALIEN＝3   ；指定后面的指令为 8 字节对齐。
指令序列
END
```

3、 CODE16、CODE32

语法格式：

CODE16（或 CODE32）

CODE16 伪指令通知编译器，其后的指令序列为 16 位的 Thumb 指令。

CODE32 伪指令通知编译器，其后的指令序列为 32 位的 ARM 指令。

若在汇编源程序中同时包含 ARM 指令和 Thumb 指令时，可用 CODE16 伪指令通知编译器其后的指令序列为 16 位的 Thumb 指令，CODE32 伪指令通知编译器其后的指令序列为 32 位的 ARM 指令。因此，在使用 ARM 指令和 Thumb 指令混合编程的代码里，可用这两条伪指令进行切换，但注意他们只通知编译器其后指令的类型，并不能对处理器进行状态的切换。

使用示例：

```
AREA  Init，CODE，READONLY
……
CODE32                  ；通知编译器其后的指令为 32 位的 ARM 指令
LDR   R0，＝NEXT＋1     ；将跳转地址放入寄存器 R0
BX    R0                ；程序跳转到新的位置执行，并将处理器切换到 Thumb 工作状态
……
CODE16                  ；通知编译器其后的指令为 16 位的 Thumb 指令
NEXT  LDR  R3，＝0x3FF
……
END                     ；程序结束
```

4、 ENTRY

语法格式：

ENTRY

ENTRY 伪指令用于指定汇编程序的入口点。在一个完整的汇编程序中至少要有一个 ENTRY（也可以有多个，当有多个 ENTRY 时，程序的真正入口点由链接器指定），但在一个源文件里最多只能有一个 ENTRY（可以没有）。

使用示例：

```
AREA  Init，CODE，READONLY
ENTRY                   ；指定应用程序的入口点
……
```

5、 END

语法格式：

END

END 伪指令用于通知编译器已经到了源程序的结尾。

使用示例：

```
AREA  Init，CODE，READONLY
……
```

```
    END                          ；指定应用程序的结尾
```

6、 EQU

语法格式：

名称　　EQU　　表达式{，类型}

EQU 伪指令用于为程序中的常量、标号等定义一个等效的字符名称，类似于 C 语言中的＃define。其中 EQU 可用“*”代替。

名称为 EQU 伪指令定义的字符名称，当表达式为 32 位的常量时，可以指定表达式的数据类型，可以有以下三种类型：

CODE16、CODE32 和 DATA

使用示例：

```
    Test    EQU 50               ；定义标号 Test 的值为 50
    Addr    EQU 0x55，CODE32     ；定义 Addr 的值为 0x55，且该处为 32 位的 ARM 指令。
```

7、 EXPORT（或 GLOBAL）

语法格式：

EXPORT　　标号{[WEAK]}

EXPORT 伪指令用于在程序中声明一个全局的标号，该标号可在其他的文件中引用。EXPORT 可用 GLOBAL 代替。标号在程序中区分大小写，[WEAK]选项声明其他的同名标号优先于该标号被引用。

使用示例：

```
  AREA  Init，CODE，READONLY
  EXPORT     Stest                ；声明一个可全局引用的标号 Stest
  ……
  END
```

8、 IMPORT

语法格式：

IMPORT　　标号{[WEAK]}

IMPORT 伪指令用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，而且无论当前源文件是否引用该标号，该标号均会被加入到当前源文件的符号表中。

标号在程序中区分大小写，[WEAK]选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为 0，若该标号为 B 或 BL 指令引用，则将 B 或 BL 指令置为 NOP 操作。

使用示例：

```
  AREA  Init，CODE，READONLY
  IMPORT     Main                 ；通知编译器当前文件要引用标号 Main，但 Main 在其他源文件中
  定义
  ……
  END
```

9、 EXTERN

语法格式：

EXTERN　　标号{[WEAK]}

EXTERN 伪指令用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，如果当前源文件实际并未引用该标号，该标号就不会被加入到当前源文件的符号表中。

标号在程序中区分大小写，[WEAK]选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为 0，若该标号为 B 或 BL 指令引用，则将 B 或 BL 指令置为 NOP 操作。

使用示例：

```
  AREA  Init，CODE，READONLY
```

```
EXTERN      Main                    ；通知编译器当前文件要引用标号 Main，但 Main 在其他源
文件中定义
……
END
```

10、 GET（或 INCLUDE）

语法格式：

GET 文件名

GET 伪指令用于将一个源文件包含到当前的源文件中，并将被包含的源文件在当前位置进行汇编处理。可以使用 INCLUDE 代替 GET。

汇编程序中常用的方法是在某源文件中定义一些宏指令，用 EQU 定义常量的符号名称，用 MAP 和 FIELD 定义结构化的数据类型，然后用 GET 伪指令将这个源文件包含到其他的源文件中。使用方法与 C 语言中的“include”相似。

GET 伪指令只能用于包含源文件，包含目标文件需要使用 INCBIN 伪指令

使用示例：

```
AREA  Init，CODE，READONLY
GET    a1.s                    ；通知编译器当前源文件包含源文件 a1.s
GET    C：\a2.s                ；通知编译器当前源文件包含源文件 C：\ a2.s
……
END
```

11、 INCBIN

语法格式：

INCBIN 文件名

INCBIN 伪指令用于将一个目标文件或数据文件包含到当前的源文件中，被包含的文件不作任何变动的存放在当前文件中，编译器从其后开始继续处理。

使用示例：

```
AREA  Init，CODE，READONLY
INCBIN      a1.dat                  ；通知编译器当前源文件包含文件 a1.dat
INCBIN      C：\a2.txt              ；通知编译器当前源文件包含文件 C：\a2.txt
……
END
```

12、 RN

语法格式：

名称 RN 表达式

RN 伪指令用于给一个寄存器定义一个别名。采用这种方式可以方便程序员记忆该寄存器的功能。其中，名称为给寄存器定义的别名，表达式为寄存器的编码。

使用示例：

```
Temp   RN   R0              ；将 R0 定义一个别名 Temp
```

13、 ROUT

语法格式：

{名称} ROUT

ROUT 伪指令用于给一个局部变量定义作用范围。在程序中未使用该伪指令时，局部变量的作用范围为所在的 AREA，而使用 ROUT 后，局部变量的作为范围为当前 ROUT 和下一个 ROUT 之间。

4.2 汇编语言的语句格式

ARM（Thumb）汇编语言的语句格式为：

{标号} {指令或伪指令} {；注释}

在汇编语言程序设计中，每一条指令的助记符可以全部用大写、或全部用小写，但不用许在一条指令中大、小写混用。

同时，如果一条语句太长，可将该长语句分为若干行来书写，在行的末尾用“\”表示下一行与本行为同一条语句。

4.2.1 在汇编语言程序中常用的符号

在汇编语言程序设计中，经常使用各种符号代替地址、变量和常量等，以增加程序的可读性。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：

— 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。

— 符号在其作用范围内必须唯一。

— 自定义的符号名不能与系统的保留字相同。

— 符号名不应与指令或伪指令同名。

1、 程序中的变量

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数字变量、逻辑变量和字符串变量。

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：真或假。

字符串变量用于在程序的运行中保存一个字符串，但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，并可使用 SETA、SETL 和 SETS 对其进行初始化。

2、 程序中的常量

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 $0\sim2^{32}-1$，当作为有符号数时，其取值范围为 $-2^{31}\sim2^{31}-1$。

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

程序中的变量可通过代换操作取得一个常量。代换操作符为“$”。

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。

使用示例：

```
LCLS    S1                              ；定义局部字符串变量 S1 和 S2
LCLS    S2
S1      SETS        “Test！”
S2      SETS        “This is a $S1”       ；字符串变量 S2 的值为“This is a Test！”
```

4.2.2 汇编语言程序中的表达式和运算符

在汇编语言程序设计中，也经常使用各种表达式，表达式一般由变量、常量、运算符和括号构

成。常用的表达式有数字表达式、逻辑表达式和字符串表达式，其运算次序遵循如下的优先级：

— 优先级相同的双目运算符的运算顺序为从左到右。

— 相邻的单目运算符的运算顺序为从右到左，且单目运算符的优先级高于其他运算符。

— 括号运算符的优先级最高。

1、 数字表达式及运算符

数字表达式一般由数字常量、数字变量、数字运算符和括号构成。与数字表达式相关的运算符如下：

— “＋”、“－”、“×”、“/” 及“MOD”算术运算符

以上的算术运算符分别代表加、减、乘、除和取余数运算。例如，以 X 和 Y 表示两个数字表达式，则：

X＋Y　　表示 X 与 Y 的和。

X－Y　　表示 X 与 Y 的差。

X×Y　　表示 X 与 Y 的乘积。

X/Y　　表示 X 除以 Y 的商。

X：MOD：Y　　表示 X 除以 Y 的余数。

— “ROL”、“ROR”、“SHL”及“SHR”移位运算符

以 X 和 Y 表示两个数字表达式，以上的移位运算符代表的运算如下：

X：ROL：Y　　表示将 X 循环左移 Y 位。

X：ROR：Y　　表示将 X 循环右移 Y 位。

X：SHL：Y　　表示将 X 左移 Y 位。

X：SHR：Y　　表示将 X 右移 Y 位。

— “AND”、“OR”、“NOT”及“EOR”按位逻辑运算符

以 X 和 Y 表示两个数字表达式，以上的按位逻辑运算符代表的运算如下：

X：AND：Y　　表示将 X 和 Y 按位作逻辑与的操作。

X：OR：Y　　表示将 X 和 Y 按位作逻辑或的操作。

：NOT：Y　　表示将 Y 按位作逻辑非的操作。

X：EOR：Y　　表示将 X 和 Y 按位作逻辑异或的操作。

2、 逻辑表达式及运算符

逻辑表达式一般由逻辑量、逻辑运算符和括号构成，其表达式的运算结果为真或假。与逻辑表达式相关的运算符如下：

— “=”、“>”、“<”、“>=”、“<= ”、“/=”、“ <>” 运算符

以 X 和 Y 表示两个逻辑表达式，以上的运算符代表的运算如下：

X = Y　　表示 X 等于 Y。

X > Y　　表示 X 大于 Y。

X < Y　　表示 X 小于 Y。

X >= Y　　表示 X 大于等于 Y。

X <= Y　　表示 X 小于等于 Y。

X /= Y　　表示 X 不等于 Y。

X <> Y　　表示 X 不等于 Y。

— “LAND”、“LOR”、“LNOT”及“LEOR”运算符

以 X 和 Y 表示两个逻辑表达式，以上的逻辑运算符代表的运算如下：

X：LAND：Y　　表示将 X 和 Y 作逻辑与的操作。

X：LOR：Y　　表示将 X 和 Y 作逻辑或的操作。

：LNOT：Y　　表示将 Y 作逻辑非的操作。

X：LEOR：Y　　表示将 X 和 Y 作逻辑异或的操作。

3、 字符串表达式及运算符

字符串表达式一般由字符串常量、字符串变量、运算符和括号构成。编译器所支持的字符串最大长度为 512 字节。常用的与字符串表达式相关的运算符如下：

— LEN 运算符

LEN 运算符返回字符串的长度（字符数），以 X 表示字符串表达式，其语法格式如下：

：LEN：X

— CHR 运算符

CHR 运算符将 0～255 之间的整数转换为一个字符，以 M 表示某一个整数，其语法格式如下：

：CHR：M

— STR 运算符

STR 运算符将将一个数字表达式或逻辑表达式转换为一个字符串。对于数字表达式，STR 运算符将其转换为一个以十六进制组成的字符串；对于逻辑表达式，STR 运算符将其转换为字符串 T 或 F，其语法格式如下：

：STR：X

其中，X 为一个数字表达式或逻辑表达式。

— LEFT 运算符

LEFT 运算符返回某个字符串左端的一个子串，其语法格式如下：

X：LEFT：Y

其中：X 为源字符串，Y 为一个整数，表示要返回的字符个数。

— RIGHT 运算符

与 LEFT 运算符相对应，RIGHT 运算符返回某个字符串右端的一个子串，其语法格式如下：

X：RIGHT：Y

其中：X 为源字符串，Y 为一个整数，表示要返回的字符个数。

— CC 运算符

CC 运算符用于将两个字符串连接成一个字符串，其语法格式如下：

X：CC：Y

其中：X 为源字符串 1，Y 为源字符串 2，CC 运算符将 Y 连接到 X 的后面。

4、 与寄存器和程序计数器（PC）相关的表达式及运算符

常用的与寄存器和程序计数器（PC）相关的表达式及运算符如下：

— BASE 运算符

BASE 运算符返回基于寄存器的表达式中寄存器的编号，其语法格式如下：

：BASE：X

其中，X 为与寄存器相关的表达式。

— INDEX 运算符

INDEX 运算符返回基于寄存器的表达式中相对于其基址寄存器的偏移量，其语法格式如下：

：INDEX：X

其中，X 为与寄存器相关的表达式。

5、 其他常用运算符

— ？运算符

？运算符返回某代码行所生成的可执行代码的长度，例如：

?X

返回定义符号 X 的代码行所生成的可执行代码的字节数。

— DEF 运算符

DEF 运算符判断是否定义某个符号，例如：

：DEF：X

如果符号 X 已经定义，则结果为真，否则为假。

4.3 汇编语言的程序结构

4.3.1 汇编语言的程序结构

在 ARM（Thumb）汇编语言程序中，以程序段为单位组织代码。段是相对独立的指令或数据序列，具有特定的名称。段可以分为代码段和数据段，代码段的内容为执行代码，数据段存放代码运行时需要用到的数据。一个汇编程序至少应该有一个代码段，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象文件。

可执行映象文件通常由以下几部分构成：

— 一个或多个代码段，代码段的属性为只读。

— 零个或多个包含初始化数据的数据段，数据段的属性为可读写。

— 零个或多个不包含初始化数据的数据段，数据段的属性为可读写。

链接器根据系统默认或用户设定的规则，将各个段安排在存储器中的相应位置。因此源程序中段之间的相对位置与可执行的映象文件中段的相对位置一般不会相同。

以下是一个汇编语言源程序的基本结构：

```
AREA  Init，CODE，READONLY
ENTRY
Start
LDR       R0，=0x3FF5000
LDR       R1，0xFF
STR       R1，[R0]
LDR       R0，=0x3FF5008
LDR       R1，0x01
STR       R1，[R0]
-------
END
```

在汇编语言程序中，用 AREA 伪指令定义一个段，并说明所定义段的相关属性，本例定义一个名为 Init 的代码段，属性为只读。ENTRY 伪指令标识程序的入口点，接下来为指令序列，程序的末尾为 END 伪指令，该伪指令告诉编译器源文件的结束，每一个汇编程序段都必须有一条 END 伪指令，指示代码段的结束。

4.3.2 汇编语言的子程序调用

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 中的返回地址重新拷贝给程序计数器 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。

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：

```
AREA  Init，CODE，READONLY
ENTRY
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
      -------
      END
```

4.3.3 汇编语言程序示例

以下是一个基于 S3C4510B 的串行通讯程序，关于 S3C4510B 的串行通讯的工作原理，可以参考第六章的相关内容，在此仅向读者说明一个完整汇编语言程序的基本结构：

```
;*******************************************************************************
; Institute of Automation,Chinese Academy of Sciences
;Description:   This example shows the UART communication!
;Author:        JuGuang,Lee
;Date:
;*******************************************************************************
UARTLCON0   EQU 0x3FFD000
UARTCONT0   EQU 0x3FFD004
UARTSTAT0   EQU 0x3FFD008
UTXBUF0     EQU 0x3FFD00C
UARTBRD0    EQU 0x3FFD014
    AREA Init,CODE,READONLY
    ENTRY
;************************************************
;LED Display
;************************************************
    LDR R1,=0x3FF5000
    LDR R0,=&ff
    STR R0,[R1]
    LDR R1,=0x3FF5008
    LDR R0,=&ff
    STR R0,[R1]
;************************************************
;UART0 line control register
;************************************************
    LDR R1,=UARTLCON0
    LDR R0,=0x03
    STR R0,[R1]
;************************************************
;UART0 control regiser
;************************************************
    LDR R1,=UARTCONT0
    LDR R0,=0x9
    STR R0,[R1]
;************************************************
;UART0 baud rate divisor regiser
```

```
;Baudrate=19200，对应于 50MHz 的系统工作频率
;*************************************************
    LDR R1,=UARTBRD0
    LDR R0,=0x500
    STR R0,[R1]
;*************************************************
;Print the messages!
;*************************************************
LOOP
    LDR R0,=Line1
    BL  PrintLine
    LDR R0,=Line2
    BL  PrintLine
    LDR R0,=Line3
    BL  PrintLine
    LDR R0,=Line4
    BL  PrintLine
    LDR R1,=0x7FFFFF
LOOP1
    SUBS    R1,R1,#1
    BNE LOOP1
    B   LOOP
;*************************************************
;Print line
;*************************************************
PrintLine
    MOV R4,LR
    MOV R5,R0
Line
    LDRB    R1,[R5],#1
    AND R0,R1,#&FF
    TST R0,#&FF
    MOVEQ   PC,R4
    BL  PutByte
    B   Line

PutByte
    LDR R3,=UARTSTAT0
    LDR R2,[R3]
    TST R2,#&40
    BEQ PutByte
    LDR R3,=UTXBUF0
    STR R0,[R3]
    MOV PC,LR
Line1   DCB
    &A,&D,"*****************************************************************",0
Line2   DCB &A,&D,"Chinese Academy of Sciences,Institute of Automation,Complex System
Lab.",0
Line3   DCB &A,&D,"      ARM Development Board Based on Samsung ARM S3C4510B.",0
Line4   DCB
    &A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&
```

```
A,&D,&A,&D,0
    END
```

4.3.4 汇编语言与 C/C++的混合编程

在应用系统的程序设计中，若所有的编程任务均用汇编语言来完成，其工作量是可想而知的，同时，不利于系统升级或应用软件移植，事实上，ARM 体系结构支持 C/C＋以及与汇编语言的混合编程，在一个完整的程序设计的中，除了初始化部分用汇编语言完成以外，其主要的编程任务一般都用 C/C++ 完成。

汇编语言与 C/C++的混合编程通常有以下几种方式：

－ 在 C/C＋＋代码中嵌入汇编指令。

－ 在汇编程序和 C/C＋＋的程序之间进行变量的互访。

－ 汇编程序、C/C＋＋程序间的相互调用。

在以上的几种混合编程技术中，必须遵守一定的调用规则，如物理寄存器的使用、参数的传递等，这对于初学者来说，无疑显得过于烦琐。在实际的编程应用中，使用较多的方式是：程序的初始化部分用汇编语言完成，然后用 C/C＋＋完成主要的编程任务，程序在执行时首先完成初始化过程，然后跳转到 C/C＋＋程序代码中，汇编程序和 C/C＋＋程序之间一般没有参数的传递，也没有频繁的相互调用，因此，整个程序的结构显得相对简单，容易理解。以下是一个这种结构程序的基本示例，该程序基于第五、六章所描述的硬件平台：

```
;*====================================================================
; Institute of Automation, Chinese Academy of Sciences
;File Name:         Init.s
;Description:
;Author:            JuGuang,Lee
;Date:
;====================================================================*
    IMPORT Main                     ;通知编译器该标号为一个外部标号
    AREA    Init,CODE,READONLY      ; 定义一个代码段
    ENTRY                           ; 定义程序的入口点
    LDR R0,=0x3FF0000               ; 初始化系统配置寄存器，具体内容可参考第五、六章
    LDR R1,=0xE7FFFF80
    STR R1,[R0]
    LDR SP,=0x3FE1000               ; 初始化用户堆栈，具体内容可参考第五、六章
    BL  Main                        ; 跳转到 Main（）函数处的 C/C++代码执行
    END                             ; 标识汇编程序的结束
```

以上的程序段完成一些简单的初始化，然后跳转到 Main（）函数所标识的 C/C＋＋代码处执行主要的任务，此处的 Main 仅为一个标号，也可使用其他名称，与 C 语言程序中的 main（）函数没有关系。

```
/*****************************************************************************
* Institute of Automation, Chinese Academy of Sciences
* File Name:        main.c
* Description:      P0,P1 LED flash.
* Author:           JuGuang,Lee
* Date:
*****************************************************************************/
void Main(void)
{
      int i;
      *((volatile unsigned long *) 0x3ff5000) = 0x0000000f;
```

```
      while(1)
        {
        *((volatile unsigned long *) 0x3ff5008) = 0x00000001;
        for(i=0; i<0x7fFFF; i++);
            *((volatile unsigned long *) 0x3ff5008) = 0x00000002;
        for(i=0; i<0x7FFFF; i++);
        }
}
```

4.4 本章小节

本章介绍了 ARM 程序设计的一些基本概念，以及在汇编语言程序设计中常见的伪指令、汇编语言的基本语句格式等，汇编语言程序的基本结构等，同时简单介绍了 C/C＋＋和汇编语言的混合编程等问题，这些问题均为程序设计中的基本问题，希望读者掌握，注意本章最后的两个示例均与后面章节介绍的基于 S3C4510B 的硬件平台有关系，读者可以参考第五、六章的相关内容。

第 5 章 应用系统设计与调试

本章主要介绍基于 S3C4510B 的硬件系统的详细设计步骤、实现细节、硬件系统的调试方法等，通过对本章的阅读，可以使绝大多数的读者具有根据自身的需求、设计特定应用系统的能力。

尽管本章所描述的内容为基于 S3C4510B 的应用系统设计，但由于 ARM 体系结构的一致性、以及外围电路的通用性，本章的所有内容对设计其他基于 ARM 内核芯片的应用系统，也具有很大的参考价值。

本章的主要内容包括：

- 嵌入式系统设计的基本方法。
- S3C4510B 概述。
- S3C4510B 的基本工作原理
- 基于 S3C4510B 的硬件系统设计详述
- 硬件系统的调试方法

5.1 系统设计概述

根据用户需求，设计出特定的嵌入式应用系统，是每一个嵌入式系统设计工程师应该达到的目标。嵌入式应用系统的设计包含硬件系统的设计和软件系统设计两个部分，并且这两部分的设计是互相关联、密不可分的，嵌入式应用系统的设计经常需要在硬件和软件的设计之间进行权衡与折中。因此，这就要求嵌入式系统设计工程师具有较深厚的硬件和软件基础，并具有熟练应用的能力。这也是嵌入式应用系统设计与其他的纯粹的软件设计或硬件设计最大的区别。

本章以北京微芯力科技有限公司（www.winsilicon.com）设计生产的 ARM Linux 评估开发板为原型，详细分析系统的软、硬件设计步骤、实现细节以及调试技巧等。ARM Linux 评估开发板的设计以学习与应用兼顾为出发点，在保证用户完成 ARM 技术的学习开发的同时，考虑了系统的扩展、电路板的面积、散热、电磁兼容性以及安装等问题，因此，该板也可作为嵌入式系统主板，直接应用在一些实际系统中。

图 5.1.1 是 ARM Linux 评估开发板的结构框图，各部分基本功能描述如下：

- 串行接口电路用于 S3C4510B 系统与其他应用系统的短距离双向串行通讯；
- 复位电路可完成系统上电复位和在系统工作时用户按键复位；
- 电源电路为 5V 到 3.3V 的 DC-DC 转换器，给 S3C4510B 及其他需要 3.3V 电源的外围电路供电；
- 10MHz 有源晶振为系统提供工作时钟，通过片内 PLL 电路倍频为 50MHz 作为微处理器的工作时钟；
- FLASH 存储器可存放已调试好的用户应用程序、嵌入式操作系统或其他在系统掉电后需要保存的用户数据等；
- SDRAM 存储器作为系统运行时的主要区域，系统及用户数据、堆栈均位于 SDRAM 存储器中；
- 10M/100M 以太网接口为系统提供以太网接入的物理通道，通过该接口，系统可以 10M 或 100Mbps 的速率接入以太网；
- JTAG 接口可对芯片内部的所有部件进行访问，通过该接口可对系统进行调试、编程等；
- IIC 存储器可存储少量需要长期保存的用户数据；
- 系统总线扩展引出了数据总线、地址总线和必须的控制总线，便于用户根据自身的特定需

求，扩展外围电路。

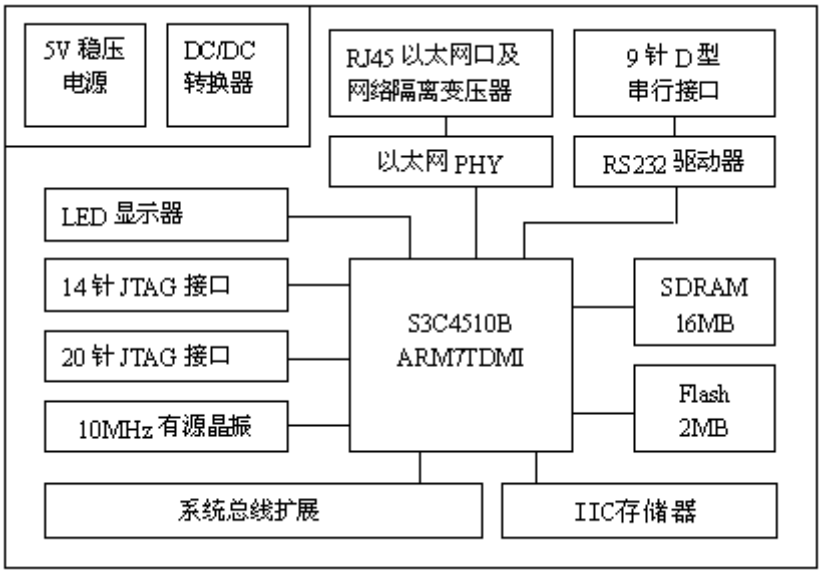


图 5.1.1　ARM Linux 评估开发板的结构框图

5.2　S3C4510B 概述

5.2.1　S3C4510B 及片内外围简介

在进行系统设计之前，有必要对 ARM Linux 评估开发板上的 ARM 芯片 S3C4510B 及其工作原理进行比较详细的介绍，读者只有对该微处理器的工作原理有了较详细的了解，才能进行特定应用系统的设计。

Samsung 公司的 S3C4510B 是基于以太网应用系统的高性价比 16/32 位 RISC 微控制器，内含一个由 ARM 公司设计的 16/32 位 ARM7TDMI RISC 处理器核，ARM7TDMI 为低功耗、高性能的 16/32 核，最适合用于对价格及功耗敏感的应用场合。

除了 ARM7TDMI 核以外，S3C4510B 比较重要的片内外围功能模块包括：

— 2 个带缓冲描述符（Buffer Descriptor）的 HDLC 通道
— 2 个 UART 通道
— 2 个 GDMA 通道
— 2 个 32 位定时器
— 18 个可编程的 I/O 口。

片内的逻辑控制电路包括：

— 中断控制器
— DRAM/SDRAM 控制器
— ROM/SRAM 和 FLASH 控制器
— 系统管理器
— 一个内部 32 位系统总线仲裁器
— 一个外部存储器控制器。

S3C4510B 结构框图如图 5.2.1 所示。

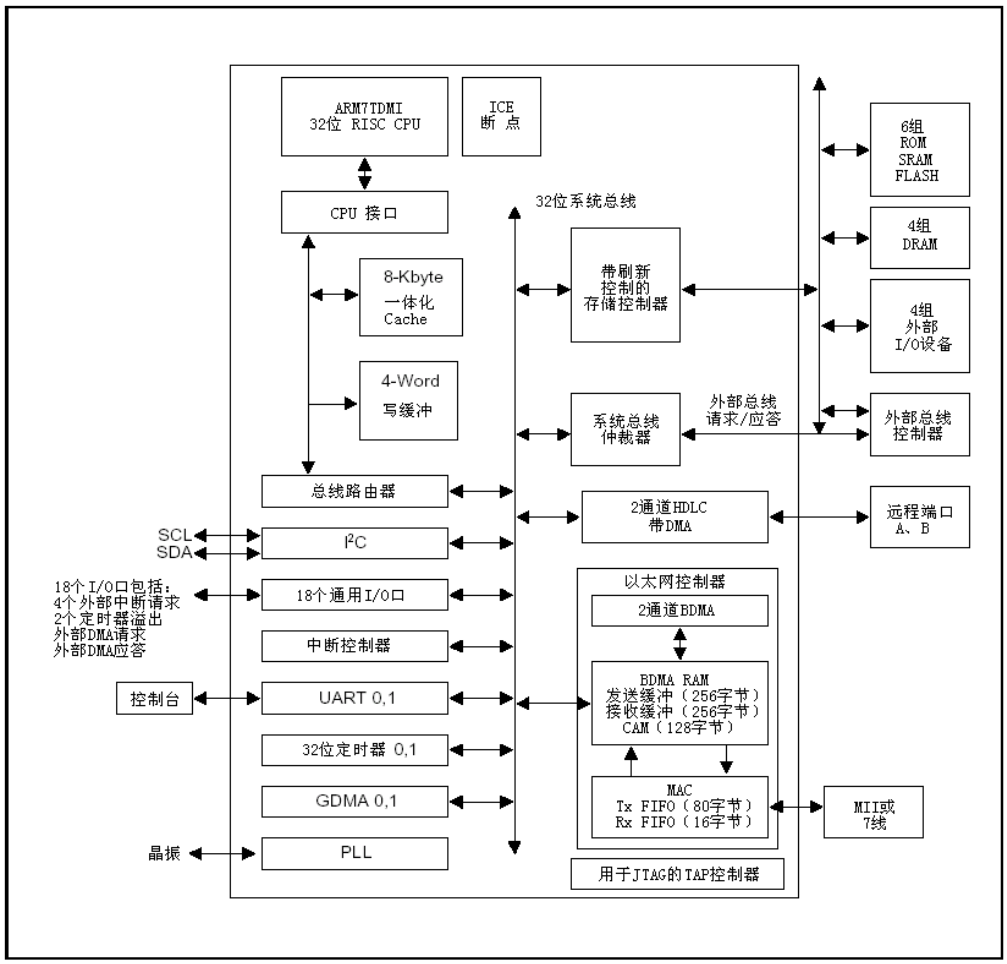


图 5.2.1

S3C4510B 结构框

S3C4510B 的特性描述如下：

体系结构

— 用于嵌入式以太网应用的集成系统
— 全 16/32 的 RISC 架构
— 支持大、小端模式。内部架构为大端模式，外部存储器可为大、小端模式
— 内含效率高、功能强的 ARM7TDMI 处理器核
— 高性价比、基于 JTAG 接口的调试方案
— 边界扫描接口

系统管理器

— 支持 ROM/SRAM、FLASH、DRAM 和外部 I/O 以 8/16/32 位的方式操作
— 带总线请求/应答引脚的外部总线控制器
— 支持 EDO/常规或 SDRAM 存储器
— 可编程的访问周期（可设定 0～7 个等待周期）
— 4 字的写缓冲
— 高性价比的从存储器到外围的 DMA 接口

一体化的指令/数据 Cache

— 一体化的 8K Cache
— 支持 LRC（近期最少使用）替换算法
— Cache 可配置为内部 SRAM

IIC 接口

— 仅支持主控模式

— 串行时钟由波特率发生器生成

Ethernet 控制器

— 带猝发模式的 DMA 引擎
— DMA 发送/接收缓冲区（256 字节发送，256 字节接收）
— MAC 发送/接收 FIFO 缓冲区（80 字节发送，16 字节接收）
— 数据对准逻辑
— 支持端模式变换
— 100M/10Mbps 的工作速率
— 与 IEEE802.3 标准完全兼容
— 提供 MII 和 7 线制 10Mbps 接口
— 站管理信号生成
— 片内 CAM（可达 21 个目的地址）
— 带暂停特性的全双工模式
— 支持长/短包模式
— 包拆装 PDA 生成

HDLC（High-Level Data Link Control）高层数据链路协议

— HDLC 协议特征：标志检测与同步；零插入与删除；空闲检测和发送；FCS 生成和检测（16 位）；终止检测与发送
— 地址搜索模式（可扩展到四字节）
— 可选择 CRC 模式或非 CRC 模式
— 用于时钟恢复的数字 PLL 模块
— 波特率生成器
— 发送和接收支持 NRZ/NRZI/FM/曼切斯特数据格式
— 回环与自动回波模式
— 8 字的发送和接收 FIFO
— 可选的 1 字或 4 字数据传送方式
— 数据对准逻辑
— 可编程中断
— Modem 接口
— 高达 10Mbps 的工作速率
— 基于 8 位位组的 HDLC 帧长度
— 每个 HDLC 有 2 通道 DMA 缓冲描述符用于发送和接收

DMA 控制器

— 用于存储器到存储器、存储器到 UATR、UATR 到存储器数据传送的 2 通道通用 DMA 控制器，不受 CPU 干预
— 可由程序或外部 DMA 请求启动
— 可增减源地址或目的地址，无论 8 位、16 位或 32 位数据传输
— 4 种数据猝发模式

UART

— 2 个可工作于 DMA 方式或中断方式的 UART 模块
— 支持 5、6、7、8 位的串行数据发送和接收
— 波特率可编程
— 1 位或 2 位停止位
— 奇/偶校验
— 间隔信号的生成与检测
— 奇偶校验、覆盖和帧错误检测

— ×16 时钟模式
— 支持红外发送和接收

定时器

— 2 个可编程 32 位定时器
— 间隔模式或触发模式工作

可编程 I/O 口

— 18 个可编程 I/O 口
— 可分别配置为输入模式、输出模式或特殊功能模式

中断控制器

— 21 个中断源，包括 4 个外部中断源
— 正常中断或快速中断模式（IRQ、FIQ）
— 基于优先级的中断处理

PLL

— 外部时钟可由片内 PLL 倍频以提高系统时钟
— 输入频率范围：10～40MHz
— 输出频率可以是输入时钟的 5 倍

工作电压

— 3.3V，偏差不超过 5%

工作温度

— 0℃～70℃

工作频率

— 最高为 50MHz

封装形式

— 208 脚 QFP 封装

5.2.2 S3C4510B 的引脚分布及信号描述

图 5.2.2 是 S3C4510B 的引脚分布图

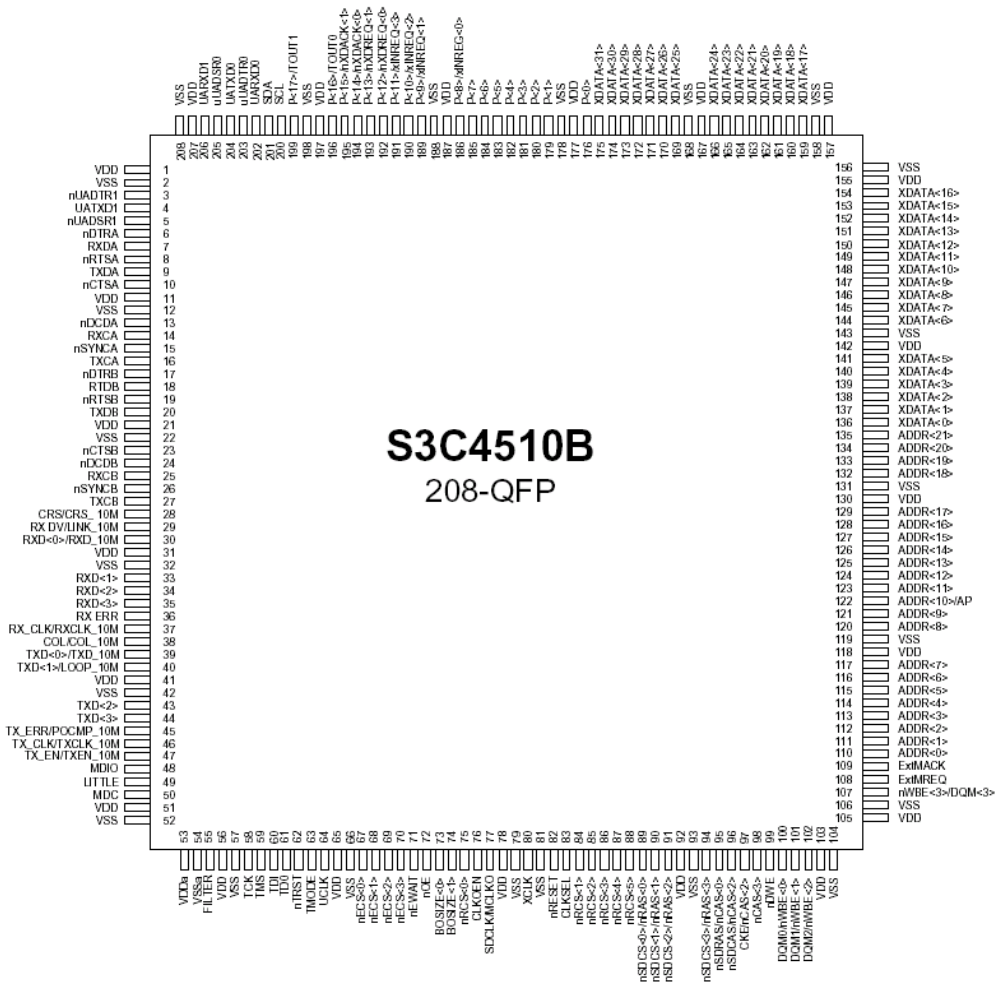


图 5.2.2 S3C4510B 的引脚分布图

各引脚信号描述如下：

表 5-2-1 S3C4510B 的引脚信号描述

信 号	引脚号	类 型	描 述
XCLK	80	I	**S3C4510B 的系统时钟源**。如果 CLKSEN 为低电平，通过 PLL 倍频的输出时钟作为 S3C4510B 的内部系统时钟。如果 CLKSEN 为高电平，XCLK 直接作为 S3C4510B 的内部系统时钟。
MCLKO/SDCLK	77	O	**系统时钟输出**。SDCLK 为 SDRAM 提供时钟信号
CLKSEL	83	I	**时钟选择**。如果 CLKSEL 为低电平，PLL 输出时钟作为 S3C4510B 的内部系统时钟。如果 CLKSEL 为高电平，XCLK 直接作为 S3C4510B 的内部系统时钟。
nRESET	82	I	**复位信号**。nRESET 为 S3C4510B 的复位信号，要使系统可靠复位，nRESET 必须至少保持 64 个主时钟周期的低电平。
CLKOEN	76	I	**时钟输出允许/禁止**。高电平允许系统时钟信号输出，低电平禁止。
TMODE	63	I	**测试模式选择**。低电平为正常工作模式，高电平为芯片测试模式。
FILTER	55	AI	如果使用 PLL，应在该引脚和数字地之间接 820pF 的陶瓷电容。

TCK	58	I	**JTAG 测试时钟**。JTAG 测试时钟信号用于切换状态信息和检测数据的输入输出。该引脚在片内下拉。
TMS	59	I	**JTAG 测试模式选择**。该信号控制 S3C4510B 的 JTAG 测试操作。该引脚在片内上拉。
TDI	60	I	**JTAG 测试数据输入**。在 JTAG 测试操作的过程中，该信号将指令和数据串行送入 S3C4510B。该引脚在片内上拉。
TDO	61	O	**JTAG 测试数据输出**。在 JTAG 测试操作的过程中，该信号将指令和数据串行送出 S3C4510B。
nTRST	62	I	**JTAG 复位信号，低电平复位**。异步复位 JTAG 逻辑。该引脚在片内上拉。
ADDR[21:0]/ ADDR[10]/AP	117-110 129-120 135-132	O	**地址总线**。22 位的地址总线可寻址每一个 ROM/ SRAM 组、FLASH 存储器组、DRAM 组和外部 I/O 组的 4M 字（64M 字节）的地址范围。
XDATA[31:0]	141-136 154-144 166-159 175-169	I/O	**外部数据总线（双向、32 位）**。S3C4510B 支持外部 8 位，16 位，32 位的数据宽度。
nRAS[3:0]/ nSDCS[3:0]	94,91, 90,89	O	**DRAM 行地址锁存信号**。S3C4510B 支持最多 4 个 DRAM 组，每个 nRAS 输出控制一组。nSDCS[3:0]用作 SDRAM 的片选信号。
nCAS[3:0] nCAS[0] /nSDRAS nCAS[1] /nSDCAS nCAS[2]/CKE	98,97, 96,95	O	**DRAM 列地址锁存信号**。无论访问哪一个 DRAM 组，4 个 nCAS 输出信号均表示字节选择。nSDRAS 作为 SDRAM 的行地址锁存信号，nSDCAS 作为 SDRAM 的列地址锁存信号，CKE 作为 SDRAM 的时钟使能信号。
nDWE	99	O	**DRAM 写使能信号**。该引脚为 DRAM 组提供写操作信号。（nWBE[3:0]用于为 ROM/SRAM/FLASH 存储器组提供写操作信号。）
nECS[3:0]	70,69, 68,67	O	**外部 I/O 片选信号**。可以有 4 个外部 I/O 组映射到存储空间，每一个外部 I/O 组的地址范围最大为 16KB。nECS 提供每一个外部 I/O 组的片选信号。
nEWAIT	71	I	**外部等待信号**。该信号用于在访问外部 I/O 设备时，由外设插入等待周期。
nRCS[5:0]	88-84,75	O	**ROM/SRAM/FLASH 片选信号**。S3C4510B 可访问多达 6 个的外部 ROM/SRAM/FLASH 组。
B0SIZE[1:0]	74,73	I	**ROM/SRAM/FLASH 存储器组 0 的数据总线宽度设定**。ROM/SRAM/FLASH 存储器组 0 常用于程序的启动。‘01’＝字节（8 位）；‘10’＝半字（16 位）；‘11’＝字（32 位）；‘00’＝保留
nOE	72	O	**输出使能**。当对存储器进行访问的时候，该信号控制存储器的输出使能。
nWBE[3:0]/ DQM[3:0]	107, 102-100	O	**写字节使能**。当对存储器进行写操作时，该信号控制存储器（DRAM 除外）的写使能。对于 DRAM 存储器组，由 nCAS[3:0]和 nDWE 控制写操作。DQM 用于 SDRAM 数据输入/输出的屏蔽信号。
ExtMREQ	108	I	**外部总线控制器请求信号**。外部总线控制器通过该引脚请求控制外部总线，当该信号有效时，S3C4510B 将外部总线置为高阻状态，以便外部总线控制器取得对外部

			总线的控制。当 ExtMACK 信号为的电平时，S3C4510B 重新取得对外部总线的控制权。
ExtMACK	109	O	**外部总线应答信号**。
MDC	50	O	**管理数据时钟**。该引脚产生 MDIO 数据输入输出时所需的时钟信号。
MDIO	48	I/O	**管理数据输入/输出**。当执行一个读数据的命令时，该引脚输入由物理层产生的数据，当执行一个写数据的命令时，由该引脚输出数据到物理层（PHY）。
LITTLE	49	I	**小端模式选择引脚**。当该引脚为高电平时，S3C4510B 工作在小端模式，当该引脚为低电平时，工作在大端模式。该引脚在片内已下拉，因此，S3C4510B 缺省工作在大端模式。
COL/COL_10M	38	I	**冲突检测/10M 冲突检测**。该引脚显示是否检测到冲突。
TX_CLK/ TXCLK_10M	46	I	**发送时钟/10M 发送时钟**。S3C4510B 在 TX_CLK 的上升沿驱动 TXD[3:0]和 TX_EN，当工作在 MII 模式时，PHY 在 TX_CLK 的上升沿采样 TXD[3:0]和 TX_EN。在发送数据时，TXCLK_10M 由 10M 的 PHY 产生。
TXD[3:0] LOOP_10M TXD_10M	44,43, 40,39	O	**发送数据/10M 发送数据/10M 回环测试**。TXD[3:0]为发送数据引脚，TXD_10M 为 10M 的 PHY 的发送数据引脚，LOOP_10M 由控制寄存器的回环测试位驱动。
TX_EN/ TXEN_10M	47	O	**发送使能/10M 发送使能**。
TX_ERR/ PCOMP_10M	45	O	**发送错误/10M 包压缩使能**。
CRS/CRS_10M	28	I	**载波侦听/10M 载波侦听**。
RX_CLK/ RXCLK_10M	37	I	**接收时钟/10M 接收时钟**。RX_CLK 为连续的时钟信号，当其频率为 25MHz 时，数据传输速率为 100M，当其频率为 2.5MHz 时，数据传输速率为 10M。在接收数据时，RXCLK_10M 由 10M 的 PHY 产生。
RXD[3:0] RXD_10M	35，34， 33，30	I	**接收数据/10M 接收数据**。
RX_DV/ LINK10M	29	I	**接收数据有效/10M 连接状态**。
RX_ERR	36	I	**接收错误**。
TXDA	9	O	**HDLC Ch-A 发送数据**。
RXDA	7	I	**HDLC Ch-A 接收数据**。
nDTRA	6	O	**HDLC Ch-A 终端准备就绪**。nDTRA 引脚指示数据终端设备准备发送或接收。
nRTSA	8	O	**HDLC Ch-A 传送请求**。
nCTSA	10	I	**HDLC Ch-A 传送清除**。
nDCDA	13	I	**HDLC Ch-A 数据载波检测**。
nSYNCA	15	O	**HDLC Ch-A 同步检测**。
RXCA	14	I	**HDLC Ch-A 接收时钟**。
TXCA	16	I/O	**HDLC Ch-A 发送时钟**。
TXDB	20	O	**HDLC Ch-B 发送数据**。
RXDB	18	I	**HDLC Ch-B 接收数据**。
nDTRB	17	O	**HDLC Ch-B 终端准备就绪**。
nRTSB	19	O	**HDLC Ch-B 传送请求**。
nCTSB	23	I	**HDLC Ch-B 传送清除**。

nDCDB	24	I	HDLC Ch-B 数据载波检测。
nSYNCB	26	O	HDLC Ch-B 同步检测。
RXCB	25	I	HDLC Ch-B 接收时钟。
TXCB	27	I/O	HDLC Ch-B 发送时钟。
UCLK	64	I	外部 UART 时钟输入。可由外部输入时钟作为 UART 时钟，通常由系统时钟提供 UART 时钟输入。
UARXD0	202	I	UART0 数据接收。
UATXD0	204	O	UART0 数据发送。
nUADTR0	203	I	UART0 数据终端准备就绪。该输入信号通知 S3C4510B，外设（或其他主机）已准备好发送或接收数据。
nUADSR0	205	O	UART0 数据设备准备就绪。该输出信号通知外设（或其他主机），UART0 已准备好发送或接收数据。
UARXD1	206	I	UART1 数据接收。
UATXD1	4	O	UART1 数据发送。
nUADTR1	3	I	UART1 数据终端准备就绪。参见 nUADTR0。
nUADSR1	5	O	UART1 数据设备准备就绪。参见 nUADSR0。
P[7:0]	185-179, 176	I/O	通用 I/O 口。
XINTREQ [3:0] P[11:8]	191-189, 186	I/O	外部中断请求信号，或作为通用 I/O 口。
NXDREQ[1:0]/ P[13:12]	193,192	I/O	外部 DAM 请求信号，或作为通用 I/O 口。
nXDACK[1:0]/ P[15:14]	195,194	I/O	外部 DAM 应答信号，或作为通用 I/O 口。
TOUT0/P[16]	196	I/O	定时器 0 溢出，或作为通用 I/O 口。
TOUT1/P[17]	199	I/O	定时器 1 溢出，或作为通用 I/O 口。
SCL	200	I/O	I^2C 串行时钟。
SDA	201	I/O	I^2C 串行数据。
VDDP	1，21，41，56，78，92，105，118，130，155，167，177，197	Power	I/O 口电源。
VDDI	11，31，51，65，103，142，157，187，207	Power	芯片内核电源。
VSSP	2，22，42，57，79，81，93，106，119，131，156，168，178，198	GND	I/O 口地。
VSSI	12，32，52，66，104，143，158，188，208	GND	芯片内核地。

VDDA	53	Power	PLL 电源
VSSA	54	GND	PLL 地

5.2.3 CPU 内核概述及特殊功能寄存器（Special Registers）

CPU 内核概述

S3C4510B 的CPU内核是由ARM公司设计的通用32位ARM7TDMI微处理器核，图5.2.3为ARM7TDMI核的结构框图。整个内核架构基于 RISC(Reduced Instruction Set Computer)规则。与 CISC(Complex Instruction Set Computer）系统相比较，RISC 架构的指令集和相关的译码电路更简洁高效。

ARM7TDMI 处理器区别于其他 ARM7 处理器的一个重要特征是其独有的称之为 Thumb 的架构策略。该策略为基本 ARM 架构的扩展，由 36 种基于标准 32 位 ARM 指令集、但重新采用 16 位宽度优化编码的指令格式构成。

由于 Thumb 指令的宽度只为 ARM 指令的一半，因此能获得非常高的代码密度。当 Thumb 指令被执行时，其 16 位的操作码被处理器解码为等效的 32 位标准 ARM 指令，然后 ARM 处理器核就如同执行 32 位的标准 ARM 指令一样执行 16 位的 Thumb 指令。也即是 Thumb 架构为 16 位的系统提供了一条获得 32 位性能的途径。

ARM7TDMI 内核既能执行 32 位的 ARM 指令集，又能执行 16 位的 Thumb 指令集，因此允许用户以子程序段为单位，在同一个地址空间使用 Thumb 指令集和 ARM 指令集混合编程，采用这种方式，用户可以在代码大小和系统性能上进行权衡，从而为特定的应用系统找到一个最佳的编程解决方案。

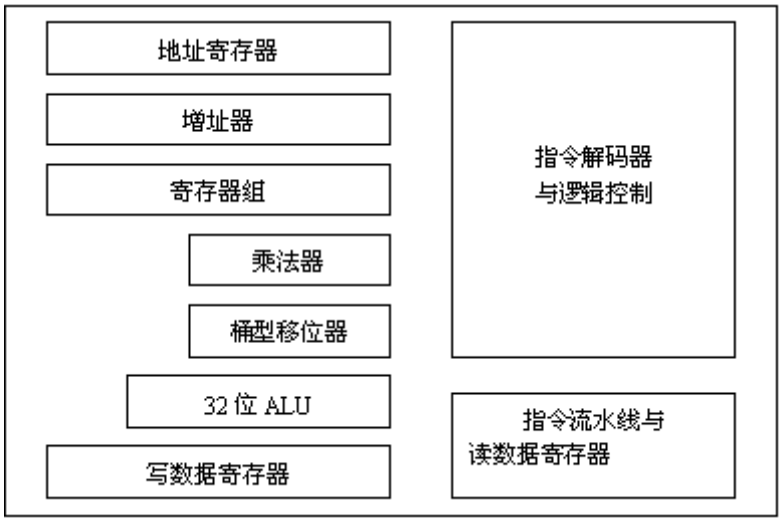


图 5.2.3 ARM7TDMI 核的结构框图

32 位的 ARM 指令集由 13 种基本的指令类型组成，可分为如下四大类：

— 4 类分支指令用于控制程序的执行流程、指令的特权等级和在 ARM 代码与 Thumb 代码之间进行切换。

— 3 类数据处理指令用于操作片上的 ALU、桶型移位器、乘法器以完成在 31 个 32 位的通用寄存器之间的高速数据处理。

— 3 类加载/存储指令用于控制在存储器和寄存器之间的数据传输。一类为方便寻址进行了优化；另一类用于快速的上下文切换；第三类用于数据交换。

— 3 类协处理器指令用于控制外部的协处理器，这些指令以开放统一的方式扩展用于片外功能指令集。

几乎所有的 32 位 ARM 指令都可以条件执行。

16 位的 Thumb 指令集为 32 位 ARM 指令集的扩展，共包含 36 种指令格式，可分为如下四个功能组：

— 4 类分支指令

— 12 类数据处理指令，为标准 ARM 数据处理指令的一个子集

— 8 类加载/存储寄存器指令

— 4 类加载/存储乘法指令

在同一种处理模式下，每一条 16 位的 Thumb 指令都有对应的 32 位 ARM 指令。

工作状态

如前所述，ARM7TDMI 内核支持两种工作状态，并总是处于其中一种工作状态。工作状态可通过软件或异常处理进行切换：

— ARM 状态： 此时执行 32 位字对齐的 ARM 指令。

— Thumb 状态： 此时执行 16 位半字对齐的 Thumb 指令。

操作模式：

ARM7TDMI 内核支持 7 种操作模式：

— 用户模式：正常的程序执行状态。

— FIQ（Fast Interrupt Request）模式:用于支持特殊的数据传送与通道处理。

— IRQ（Interrupt ReQuest）模式：用于通用的中断处理。

— 管理模式：一种用于操作系统的保护模式。

— 中止模式：当数据或指令预取中止时进入该模式

— 系统模式：一种用于操作系统的特权用户模式。

— 未定义模式：当执行了未定义指令时进入该模式。

可用软件控制操作模式的切换，同时外部的中断和异常处理也会导致操作模式的切换。

绝大多数的用户应用程序运行在用户模式。

当系统响应中断或异常、或访问受保护的系统资源时，处理器会进入特权模式（除用户模式以外的所有模式）。

寄存器

S3C4510B 内建 37 个 32 位的寄存器：31 个通用寄存器，6 个状态寄存器，但并不是所有的寄存器都能总是被访问到。在某一时刻寄存器能否访问由处理器的当前工作状态和操作模式决定。

根据微处理器内核的当前工作状态，可分别访问 ARM 状态寄存器集和 Thumb 状态寄存器集：

— ARM 状态寄存器集包含 16 个可以直接访问的寄存器：R0～R15。除 R15 以外，其余的寄存器为通用寄存器，可用于存放地址或数据值。另外一个（第 17 个）寄存器是当前程序状态寄存器 CPSR，用于保存状态信息。

— Thumb 状态寄存器集是 ARM 状态寄存器集的一个子集。可以访问的寄存器有：8 个通用寄存器 R0～R7，程序计数器 PC、堆栈指针寄存器 SP、连接寄存器 LR 和当前程序状态寄存器 CPSR。

在每一种特权模式下，都有对应的分组堆栈指针寄存器 SP、连接寄存器 LR 和备份的程序状态寄存器 SPSR。

Thumb 状态寄存器集与 ARM 状态寄存器集的对应关系如下：

— Thumb 状态下 R0～R7 寄存器与 ARM 状态下 R0～R7 寄存器是相同的。

— Thumb 状态下的 CPSR 和 SPSRs 与 ARM 状态下的 CPSR 和 SPSRs 是相同的。

— Thumb 状态下的 SP、LR 和 PC 直接对应 ARM 状态寄存器 R13、R14 和 R15。

在 Thumb 状态下，寄存器 R8～R15 不属于标准寄存器集的一部分，但在必要的情况下，用户可以通过汇编语言程序访问他们，用作快速的临时存储将单元。

关于寄存器的详细描述，可参阅第二章编程模型的相关内容。

异常

当正常的程序执行流程被中断时，称为产生了异常。例如程序执行转向响应一个外设的中断请求。在优先处理异常时，处理器的当前状态必须保留，以便在异常处理完成之后程序流程能正常返回。并且，多个异常可能会同时发生。

为处理异常，S3C4510B 使用内核的分组寄存器来保存当前状态，原来的 PC 值和 CPSR 的内容被拷贝到对应的 R14（LR）和 SPSR 寄存器中，PC 和 CPSR 中的模式位被调整到相应被处理的异常类型的值。

S3C4510B 的内核支持 7 种类型的异常，每一种异常都有其固定的优先级和对应的特权处理器模

式，如表 5-2-2 所示：

表 5-2-2 S3C4510B 的异常类型

异　　常	进入模式	优先级
复位（Reset）	管理模式	1（最高）
数据中止（Data Abort）	中止模式	2
FIQ	FIQ 模式	3
IRQ	IRQ 模式	4
预取中止（Prefetch Abort）	中止模式	5
未定义指令（Undefined Instruction）	未定义模式	6（最低）
SWI	管理模式	6（最低）

S3C4510B 的特殊功能寄存器

表 5-2-3 为 S3C4510B 片内的特殊功能寄存器描述。

表 5-2-3 S3C4510B 的特殊功能寄存器

分组	寄存器	偏移量	R/W	描　　述	复位值
系统管理器	SYSCFG	0x0000	读/写	系统配置寄存器	0x37FFFF91
	CLKCON	0x3000	读/写	时钟控制寄存器	0x00000000
	EXTACON0	0x3008	读/写	外部 I/O 时序寄存器 1	0x00000000
	EXTACON1	0x300C	读/写	外部 I/O 时序寄存器 2	0x00000000
	EXTDBWTH	0x3010	读/写	分组数据总线的宽度设置寄存器	0x00000000
	ROMCON0	0x3014	读/写	ROM/ARAM/FLASH 组 0 控制寄存器	0x20000060
	ROMCON1	0x3018	读/写	ROM/ARAM/FLASH 组 1 控制寄存器	0x00000060
	ROMCON2	0x301C	读/写	ROM/ARAM/FLASH 组 2 控制寄存器	0x00000060
	ROMCON3	0x3020	读/写	ROM/ARAM/FLASH 组 3 控制寄存器	0x00000060
	ROMCON4	0x3024	读/写	ROM/ARAM/FLASH 组 4 控制寄存器	0x00000060
	ROMCON5	0x3028	读/写	ROM/ARAM/FLASH 组 5 控制寄存器	0x00000060
	DRAMCON0	0x302C	读/写	DRAM 组 0 控制寄存器	0x00000000
	DRAMCON1	0x3030	读/写	DRAM 组 1 控制寄存器	0x00000000
	DRAMCON2	0x3034	读/写	DRAM 组 2 控制寄存器	0x00000000
	DRAMCON3	0x3038	读/写	DRAM 组 3 控制寄存器	0x00000000
	REFEXTCON	0x303C	读/写	刷新与外部 I/O 控制寄存器	0x000083FD
以太网控制器 (BDMA)	BDMATXCON	0x9000	读/写	BDMA 接收控制寄存器	0x00000000
	BDMARXCON	0x9004	读/写	BDMA 发送控制寄存器	0x00000000
	BDMATXPTR	0x9008	读/写	发送帧描述符起始地址寄存器	0x00000000
	BDMARXPTR	0x900C	读/写	接收帧描述符起始地址寄存器	0x00000000
	BDMARXLSZ	0x9010	读/写	接收帧最大长度寄存器	未定义
	BDMASTAT	0x9014	读/写	BDMA 状态寄存器	0x00000000
	CAM	0x9100-0x917C	写	CAM 内容（共 32 字）	未定义

	BDMATXBUF	0x9200-0x92FC	读/写	BDMA Tx 缓冲测试模式地址(64 字)	未定义
	BDMARXBUF	0x9800-0x99FC	读/写	BDMA Rx 缓冲测试模式地址(64 字)	未定义
以太网控制器(MAC)	MACON	0xA000	读/写	以太网 MAC 控制寄存器	0x00000000
	CAMCON	0xA004	读/写	CAM 控制寄存器	0x00000000
	MACTXCON	0xA008	读/写	MAC 发送控制寄存器	0x00000000
	MACTXSTAT	0xA00C	读/写	MAC 发送状态寄存器	0x00000000
	MADRXCON	0xA010	读/写	MAC 接收控制寄存器	0x00000000
	MACRXSTAT	0xA014	读/写	MAC 接收状态寄存器	0x00000000
	STADATA	0xA018	读/写	工作站管理数据寄存器	0x00000000
	STACON	0xA01C	读/写	工作站管理控制与地址寄存器	0x00006000
	CAMEN	0xA028	读/写	CAM 使能寄存器	0x00000000
	EMISSCNT	0xA03C	读/写	错误计数寄存器	0x00000000
	EPZCNT	0xA040	读	中止计数寄存器	0x00000000
	ERMPZCNT	0xA044	读	远程中止计数寄存器	0x00000000
	ETXSTAT	0x9040	读	传送控制帧控制状态寄存器	0x00000000
HDLC A 通道	HMODE	0x7000	读/写	HDLC 模式寄存器	0x00000000
	HCON	0x7004	读/写	HDLC 控制寄存器	0x00000000
	HSTAT	0x7008	读/写	HDLC 状态寄存器	0x00010400
	HINTEN	0x700C	读/写	HDLC 中断使能寄存器	0x00000000
	HTXFIFOC	0x7010	读/写	TxFIFO 帧持续寄存器	_
	HTXFIFOT	0x7014	读	TxFIFO 帧中止寄存器	_
	HRXFIFO	0x7018	读	HDLC RxFIFO 入口寄存器	0x00000000
	HBRGTC	0x701C	写	HDLC 波特率发生时间常数寄存器	0x00000000
	HPRMB	0x7020	读/写	HDLC 前缀常数寄存器	0x00000000
	HSAR0	0x7024	读/写	HDLC 站地址 0 寄存器	0x00000000
	HSAR1	0x7028	读/写	HDLC 站地址 1 寄存器	0x00000000
	HSAR2	0x702C	读/写	HDLC 站地址 2 寄存器	0x00000000
	HSAR3	0x7030	读/写	HDLC 站地址 3 寄存器	0x00000000
	HMASK	0x7034	读/写	HDLC 掩码寄存器	0x00000000
	DMATxPTR	0x7038	读/写	DMA Tx 缓冲描述符指针寄存器	0xFFFFFFFF
	DMARxPTR	0x703C	读/写	DMA Rx 缓冲描述符指针寄存器	0xFFFFFFFF
	HMFLR	0x7040	读/写	最大帧长度寄存器	0xXXXX000
	HRBSR	0x7044	读/写	DMA 接收缓冲长度寄存器	0xXXXX0000
HDLC B 通道	HMODE	0x8000	读/写	HDLC 模式寄存器	0x00000000
	HCON	0x8004	读/写	HDLC 控制寄存器	0x00000000
	HSTAT	0x8008	读/写	HDLC 状态寄存器	0x00010400
	HINTEN	0x800C	读/写	HDLC 中断使能寄存器	0x00000000
	HTXFIFCO	0x8010	写	TxFIFO 帧持续寄存器	_
	HTXFIFOT	0x8014	写	TxFIFO 帧中止寄存器	_
	HRXFIFO	0x8018	读	HDLC RxFIFO 入口寄存器	0x00000000

	HBRGTC	0x801C	读/写	HDLC 波特率发生时间常数寄存器	0x00000000
	HPRMB	0x8020	读/写	HDLC 前缀常数寄存器	0x00000000
	HSAR0	0x8024	读/写	HDLC 站地址 0 寄存器	0x00006000
	HSAR1	0x8028	读/写	HDLC 站地址 1 寄存器	0x00000000
	HSAR2	0x802C	读/写	HDLC 站地址 2 寄存器	0x00000000
	HSAR3	0x8030	读	HDLC 站地址 3 寄存器	0x00000000
	HMASK	0x8034	读	HDLC 掩码寄存器	0x00000000
	DMATxPTR	0x8038	读	DMA Tx 缓冲描述符指针寄存器	0xFFFFFFFF
	DMARxPTR	0x803C	读/写	DMA Rx 缓冲描述符指针寄存器	0xFFFFFFFF
	HMFLR	0x8040	读/写	最大帧长度寄存器	0xXXXX0000
	HRBSR	0x8044	读/写	DMA 接收缓冲长度寄存器	0xXXXX0000
I/O 口	IOPMOD	0x5000	读/写	I/O 口模式寄存器	0x00000000
	IOPCON	0x5004	读/写	I/O 口控制寄存器	0x00000000
	IOPDATA	0x5008	读/写	I/O 口数据寄存器	未定义
中断控制器	INTMOD	0x4000	读/写	中断模式寄存器	0x00000000
	INTPND	0x4004	读/写	中断悬挂寄存器	0x00000000
	INTMSK	0x4008	读/写	中断屏蔽寄存器	0x003FFFFF
	INTPRI0	0x400C	读/写	中断优先级寄存器 0	0x03020100
	INTPRI1	0x4010	读/写	中断优先级寄存器 1	0x07060504
	INTPRI2	0x4014	读/写	中断优先级寄存器 2	0x0B0A0908
	INTPRI3	0x4018	读/写	中断优先级寄存器 3	0x0F0E0D0C
	INTPRI4	0x401C	读/写	中断优先级寄存器 4	0x13121110
	INTPRI5	0x4020	读/写	中断优先级寄存器 5	0x00000014
	INTOFFSET	0x4024	读	中断偏移地址寄存器	0x00000054
	INTOSET_FIQ	0x4030	读	FIQ 中断偏移量寄存器	0x00000054
	INTOSET_IRQ	0x4034	读	IRQ 中断偏移量寄存器	0x00000054
IIC 总线	IICCON	0XF000	读/写	IIC 总线控制状态寄存器	0x00000054
	IICBUF	0xF004	读/写	IIC 总线移位缓冲寄存器	未定义
	IICPS	0xF008	读/写	IIC 总线预分频寄存器	0x00000000
	IICCONOUT	0xF00C	读	IIC 总线预分频计数寄存器	0x00000000
GDMA	GDMACON0	0xB000	读/写	GDMA 通道 0 控制寄存器	0x00000000
	GDMACON1	0xC000	读/写	GDMA 通道 1 控制寄存器	0x00000000
	GDMASRC0	0xB004	读/写	GDMA 源地址寄存器 0	未定义
	GDMADST0	0xB008	读/写	GDMA 目的地址寄存器 0	未定义
	GDMASRC1	0xC004	读/写	GDMA 源地址寄存器 1	未定义
	GDMADST1	0xB008	读/写	GDMA 目的地址寄存器 1	未定义
	GDMACNT0	0xB00C	读/写	GDMA 通道 0 传输计数寄存器	未定义
	GDAMCNT1	0xC00C	读/写	GDMA 通道 1 传输计数寄存器	未定义

UART	ULCON0	0xD000	读/写	UART 通道 0 行控制寄存器	0x00
	ULCON1	0xE000	读/写	UART 通道 1 行控制寄存器	0x00
	UCON0	0xD004	读/写	UART 通道 0 控制寄存器	0x00
	UCON1	0xE004	读/写	UART 通道 1 控制寄存器	0x00
	USTAT0	0xD008	读	UART 通道 0 状态寄存器	0xC0
	USTAT1	0xE008	读	UART 通道 1 状态寄存器	0xC0
	UTXBUF0	0xD00C	写	UART 通道 0 发送保持寄存器	未定义
	UTXBUF1	0xE00C	写	UART 通道 1 发送保持寄存器	未定义
	URXBUF0	0xD010	读	UART 通道 0 接收缓冲寄存器	未定义
	URXBUF1	0xE010	读	UART 通道 1 接收缓冲寄存器	未定义
	UBRDIV0	0xD014	读/写	波特率除数因子寄存器 0	0x00
	UBRDIV1	0xE014	读/写	波特率除数因子寄存器 1	0x00
定时器	TMOD	0x6000	读/写	定时器模式寄存器	0x00000000
	TDATA0	0x6004	读/写	定时器 0 数据寄存器	0x00000000
	TDATA1	0x6008	读/写	定时器 1 数据寄存器	0x00000000
	TCNT0	0x600C	读/写	定时器 0 计数寄存器	0xFFFFFFFF
	TCNT1	0x6010	读/写	定时器 1 计数寄存器	0xFFFFFFFF

5.2.4 S3C4510B 的系统管理器（System Manager）

1. 概述

S3C4510B 微处理器的系统管理器（System Manager）在整个系统工作中起至关重要作用，只有清楚的了解系统管理器在系统中的作用及工作原理，才能进行程序设计和系统开发，但同时，相对于 8 位或 16 位微处理器而言，S3C4510B 系统管理器的工作原理又是比较复杂的，因此需要读者认真细致的阅读该部分内容，并通过编程实践加以掌握。

S3C4510B 微处理器的系统管理器具有以下功能：

— 基于固定的优先级，仲裁来自几个主功能模块的系统总线访问请求。

— 为访问外部存储器提供必需的存储器控制信号。例如 DMA 控制器或 CPU 要访问 DRAM 组的某地址，则系统管理器的 DRAM 控制器就会产生必需的 normal/EDO 或 SDRAM 访问信号。可由 SYSCFG[31]设定访问 normal/EDO 或 SDRAM 的信号。

— 为 S3C4510B 和 ROM/SRAM，以及外部 I/O 组之间的总线通信提供必需的信号。

— 为外部存储器的数据总线和内部数据总线之间的数据流协调总线宽度的差别。

— 对外部存储器和 I/O 设备，S3C4510B 同时支持小端模式和大端模式的访问方式。

通过产生外部总线请求信号，外设可访问 S3C4510B 的外部总线。另外，S3C4510B 可通过插入等待周期（WAIT 信号）访问低速外设。WAIT 信号由外设产生，可延长 CPU 的存储器访问周期。

2. 系统管理器寄存器(System Manager Registers)

系统管理器使用一组专用的特殊功能寄存器来控制外部存储器的读/写操作，通过对该组特殊功能寄存器编程，可以设定：

— 存储器的类型

— 外部数据总线宽度及访问周期

— 定时的控制信号（例如 RAS 和 CAS）

— 存储器组的定位

— 存储器组的大小

在标准系统配置中访问外设必需的控制信号、地址信号和数据信号，系统管理器通过设置特殊功能寄存器的值来控制其产生和处理。特殊功能寄存器也被用于控制对 ROM/SRAM/Flash 组的访问，同时还能控制对多达四个 DRAM 组和四个外部 I/O 组以及一个特殊功能寄存器映射区域的访问。

每个存储器组在组内通过基指针（Base Pointer）寻址，其寻址范围是 64KB（16 位），而基指针本身为 10 位。因此 S3C4510B 的最大可寻址范围是 2^{26}＝64MB（或 16M 字）。

在进行系统存储器映射时，注意两个相连的存储器组的地址空间决不能重叠。图 5.2.4 为 S3C4510B 系统存储器映射。

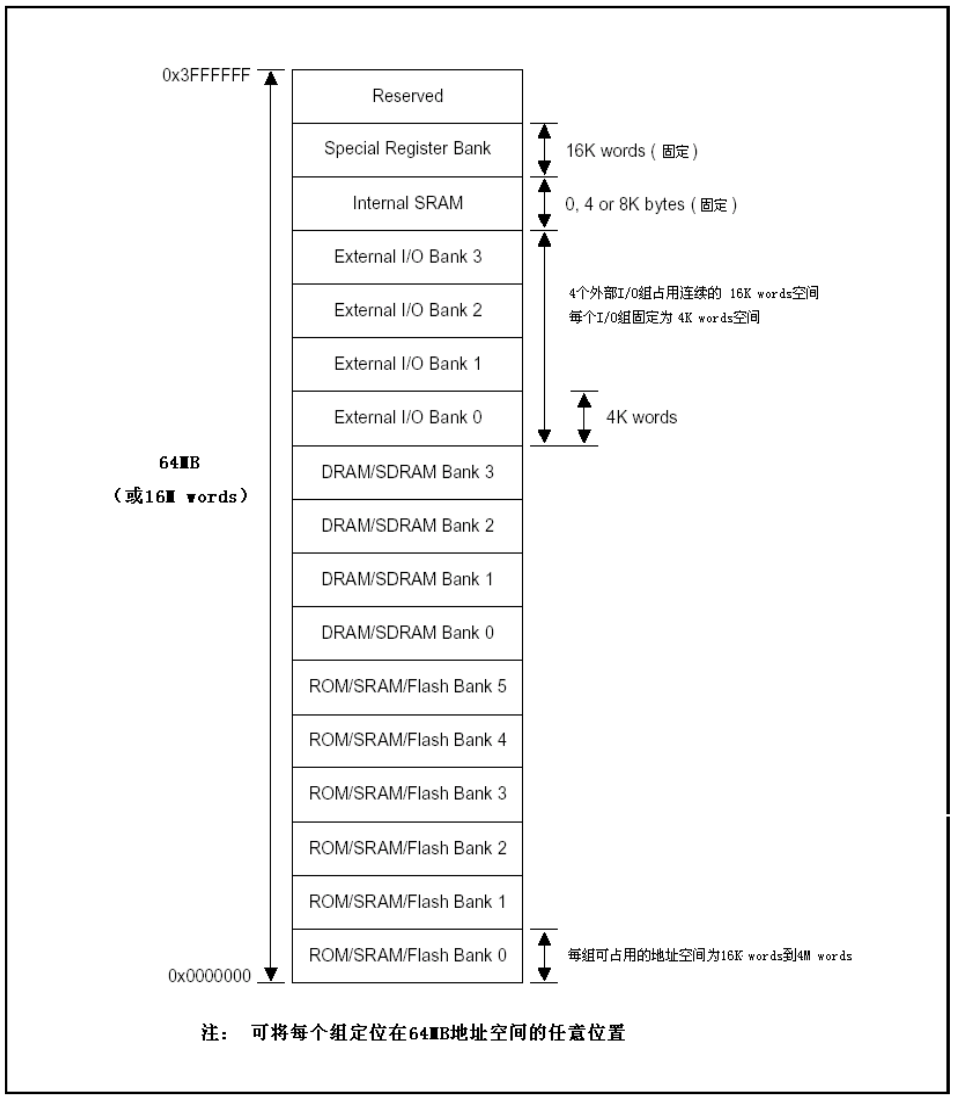


图 5.2.4 S3C4510B 系统存储器映射

系统存储器映射(System Memory Map)

关于 S3C4510B 系统存储器映射，以下是几个应当注意的重点：

— S3C4510B 采用统一编址的方式，将系统的片外存储器、片内存储器、特殊功能寄存器和外部的 I/O 设备，都映射到 64MB 的地址空间，同时，为便于管理，又将地址空间分为如图 5.2.4 所示的若干个存储器组，可以通过配置包含基指针（Base Pointer）和尾指针（End Pointer）的特殊功能寄存器，设定每个存储器组的大小和位置。用户可利用基指针和尾指针设置连续的存储器映射。具体操作如下：即把某个存储器组的基指针的地址设置为前一个存储器组的尾指针的地址。请注意在设定存储器组的控制寄存器时，每两个相连的存储器组的地址空间决不能重叠，即使这些组被禁用。

— 四个外部 I/O 组被定义在一个连续的地址空间中。只需要将基指针分配给外部 I/O 组 0，外部 I/O 组 1 的起始地址就等于外部 I/O 组 0 的起始地址+16KB，同理，外部 I/O 组 2 的起始地址就等于外部 I/O 组 0 的起始地址+32KB，外部 I/O 组 3 的起始地址就等于外部 I/O 组 0 的起始地址+48KB。因此，四个外部组的总的连续的可寻址范围被定义在外部 I/O 组 0 的起始地址+64KB 的地址空间。在整个可寻址的地址空间中，外部 I/O 组的起始地址并没有被固定。通过设定组的基指针，可以设定一个具体的组起始地址，但总的地址空间是连续的 64KB。

— 每个组的起始物理地址为“基指针左移 16 位”，每组末尾的物理地址为“尾指针左移 16 位 - 1”。

在上电或系统复位后，所有组的地址指针寄存器都被初始化到其缺省值。这时，所有的组指针（ROM/SRAM/Flash 组 0 和特殊功能寄存器组除外）都被清零。这意味着：除 ROM/SRAM/Flash 组 0 和特殊功能寄存器组以外，所有其它组在系统启动时都是未被定义的。这一点很重要，用户在进行程序设计时，一般总是要首先通过配置相应寄存器，定义系统的存储空间。

ROM/SRAM/Flash 组 0 的尾指针和基指针的复位值分别为 0x200 和 0x0。这意味着系统复位后将自动定义 ROM/SRAM/Flash 组 0 的地址空间为 32MB，实际地址范围为 0x0000,0000～0x0200,0000-1。ROM/SRAM/Flash 组 0 的这种初始化定义使得系统在上电或复位后，将系统的控制权交给了由用户编写的启动代码，当然这些启动代码应存放在外部 ROM 中的，并映射到 ROM/SRAM/Flash 组 0。当起动代码执行时，它执行各种系统初始化任务，并根据应用系统的外部存储器和设备的实际情况来重新配置系统的存储器映射。

特殊功能寄存器组的基址针在系统复位时被初始化为 0x3FF0000，一般不再改动。

图 5.2.5 是在系统启动或复位时的系统存储器映射。

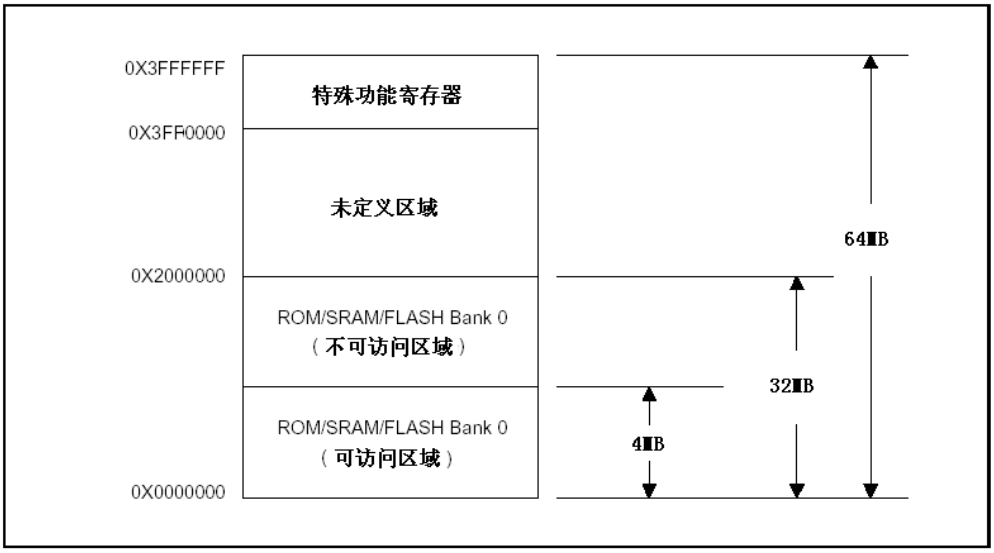


图 5.2.5 系统复位时的存储器映射

表 5-2-4 为系统管理器对应的寄存器，系统的存储器映射可通过对这些寄存器的配置来完成。

表 5-2-4 系统管理器对应的寄存器

寄存器	偏移量	操作	描 述	复位值
SYSCFG	0x0000	读/写	系统配置寄存器	0x37FFFF91
CLKCON	0x3000	读/写	时钟控制寄存器	0x00000000
EXTACON0	0x3008	读/写	外部 I/O 时序寄存器 1	0x00000000
EXTACON1	0x300C	读/写	外部 I/O 时序寄存器 2	0x00000000
EXTDBWTH	0x3010	读/写	分组数据总线的宽度设置寄存器	0x00000000

ROMCON0	0x3014	读/写	ROM/ARAM/FLASH 组 0 控制寄存器	0x20000060
ROMCON1	0x3018	读/写	ROM/ARAM/FLASH 组 1 控制寄存器	0x00000060
ROMCON2	0x301C	读/写	ROM/ARAM/FLASH 组 2 控制寄存器	0x00000060
ROMCON3	0x3020	读/写	ROM/ARAM/FLASH 组 3 控制寄存器	0x00000060
ROMCON4	0x3024	读/写	ROM/ARAM/FLASH 组 4 控制寄存器	0x00000060
ROMCON5	0x3028	读/写	ROM/ARAM/FLASH 组 5 控制寄存器	0x00000060
DRAMCON0	0x302C	读/写	DRAM 组 0 控制寄存器	0x00000000
DRAMCON1	0x3030	读/写	DRAM 组 1 控制寄存器	0x00000000
DRAMCON2	0x3034	读/写	DRAM 组 2 控制寄存器	0x00000000
DRAMCON3	0x3038	读/写	DRAM 组 3 控制寄存器	0x00000000
REFEXTCON	0x303C	读/写	刷新与外部 I/O 控制寄存器	0x000083FD

根据外部存储器的宽度决定外部地址译码方法(External Address Translation Method Depends on the Width of External Memory)

与某些 ARM 芯片不同,S3C4510B 应用系统的地址总线的连接方式相对简单。由于 ARM7TDMI 采用 32 位地址总线，所有的地址都可以看作字节地址，地址总线提供 4GB 的线性寻址空间，当发出字访问信号时，存储系统忽略低 2 位 A[1:0]，当发出半字访问信号时，存储系统忽略低位 A[0]，基于以上原因，某些 ARM 系统在与存储器接口时，地址总线的连接需要错开，而 S3C4510B 则通过一个片内的地址总线生成部件，隐藏该过程，用户在设计系统时，只需将 S3C4510B 的地址总线与存储器的地址总线一一对应连接即可（即 S3C4510B 的 A[0]与外部存储器的 A[0]对齐）。表 5-2-5 和图 5.2.6 说明了该过程。

表 5-2-5　地址总线生成

数据总线宽度	外部地址引脚，ADDR[21:0]	可访问的存储空间
8 位	A21-A0（内部）	4M 字节
16 位	A21-A1（内部）	4M 半字
32 位	A21-A2（内部）	4M 字

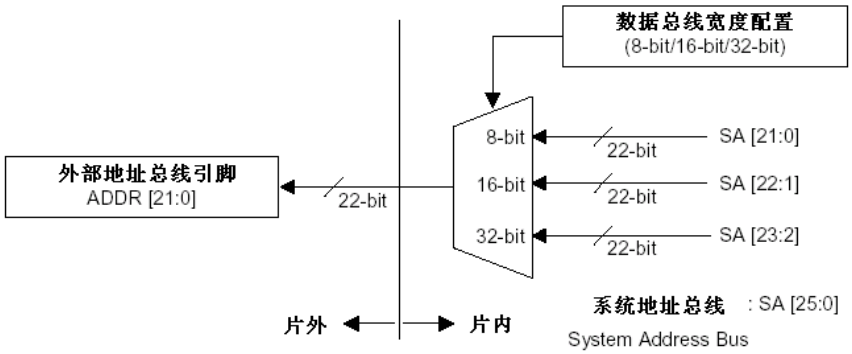


图 5.2.6　地址总线变换

总线仲裁（Bus Arbitration）

对于 S3C4510B 微处理器，“系统总线”是指连接各个功能模块的地址和数据总线。S3C4510B

的片内功能模块和片外设备都可请求总线控制权，然后占用总线以完成数据的传输。但 S3C4510B 在系统设计时，任一时刻只允许一个部件占用总线，因此，当有两个或两个以上的片内功能模块或外设同时请求总线控制权时，就要求总线控制器进行仲裁。

当某一个片内功能模块或外设取得总线控制权时，其他的总线请求被悬挂（等候处理）直到原来的总线控制设备释放总线控制权时才能被响应。

为便于总线仲裁，S3C4510B 内部的每一个功能模块都设置了优先级，总线控制器就根据这个固定的优先级对各个模块的总线请求进行仲裁。通常，总线控制权总是分配给优先级较高的功能模块。表 5-2-6 列出了优先级。

表 5-2-6 总线仲裁优先级

功能模块	总线优先级（分组）
外部总线主控器	A-1（A 组中的最高优先级）
DRAM 存储器刷新控制器	A-2
General DMA1（GDMA1）	A-3
General DMA0（GDMA0）	A-4
高层数据链路控制器 B（HDLC B）	A-5
高层数据链路控制器 A（HDLC A）	A-6
MAC 带缓冲 DMA（BDMA）	A-7（A 组中的最低优先级）
写缓冲器	B-1（B 组中的最高优先级）
总线路由器	B-2（B 组中的最低优先级）

注：S3C4510B 的内部功能模块分为 A、B 两组，在每个组内，其优先级根据设置固定。

外部总线控制（External Bus Mastership）

S3C4510B 微处理器能检测并响应由外部总线主控器产生的总线请求信号（ExtMREQs）。当 CPU 发出外部总线应答信号（ExtMACK）后，总线控制权就交给外部总线主控器，此时外部总线请求信号应继续有效。

当 S3C4510B 的外部总线应答信号有效时，其存储器接口处于高阻状态，以便外部总线主控器能驱动外部存储器接口。

当 S3C4510B 不控制总线时，它也不再进行 DRAM 的刷新操作，因此，当外部的总线主控器取得总线控制权且会持续一段较长的时间，必须负责完成 DRAM 的刷新操作。

3. 控制寄存器（Control Register）

系统配置寄存器(System Configuration Register)

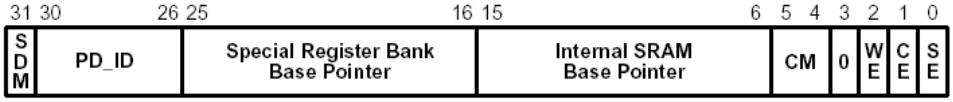
系统管理器中有一个系统配置寄存器 SYSCFG，该寄存器决定系统管理器中特殊功能寄存器组的起始地址，以及片内 SRAM 的使用方式和起始地址。在系统存储器映射中，特殊功能寄存器组的地址空间固定为 64KB，可参考图 5.2.4。

用户可以通过 SYSCFG 的设定控制诸如写缓冲、缓存模式、DRAM 模式、以及片内 SRAM 的起始地址等。

该寄存器的设置对系统的运行及性能有很大的影响，系统设计者应仔细理解每一位所代表的意义并正确设置。

SYSCFG 寄存器

寄存器	偏移地址	操作	描 述	复位值
SYSCFG	0x0000	读/写	系统配置寄存器	0x67FFFF91



[0]Stall 使能（SE）

该位必须置为 0。

[1]Cache 使能（CE）

该位置 1，Cache 操作使能。

[2]写缓冲使能（WE）

该位置 1，写缓冲操作使能。

[5：4]Cache 模式（CM）

该两位决定如何分配片内 8KB 存储器为 Cache 和 SRAM。

00 = 4KB SRAM，4KB Cache

01 = 0KB SRAM，8KB Cache

10 = 8KB SRAM，0KB Cache

11 = 系统保留

注：当设置为 10 时，CE 位自动清零。

[15：6]片内 SRAM 的基指针

该设置值左移 16 位即为片内 SRAM 的起始物理地址。

[25：16]特殊功能寄存器组的基指针

该设置值左移 16 位即为特殊功能寄存器组的起始物理地址。

[30：26]产品号（PD_ID）

00001=S3C4510X（KS32C50100）

11001=S3C4510B

[31]SDRAM 模式

0＝4 个 DRAM 组均设置为普通/EDO DRAM 接口

1＝4 个 DRAM 组均设置为 SDRAM 接口

起始地址设置（Start Address Setting）

系统管理器中特殊功能寄存器组的起始地址在系统加电或复位时被初使化为 0x3FF0000，用户也可通过配置 SYSCFG 寄存器的值，将特殊功能寄存器组的起始地址设定在 64MB 地址空间的任意位置。每一个特殊功能寄存器的实际物理地址为特殊功能寄存器组的起始地址加上该寄存器的偏移地址。

例如，系统复位时特殊功能寄存器组的起始地址初始化为 0x3FF0000，特殊功能寄存器 ROMCON 的偏移地址为 0x3014，因此，ROMCON 的物理地址是：

0x3FF0000＋0x3014＝0x3FF3014

如果用户重新设定特殊功能寄存器组的起始地址为 0x3000000，则 ROMCON 新的物理地址是 0x3003014。

在大多数情况下，并不需要去重新设置特殊功能寄存器组的起始地址，而直接使用其初始化值 0x3FF0000。

Cache 禁止/使能（Cache Disable/Enable）

用户可通过设置 SYSCFG 中的 CE 位来禁止或使能 Cache，但由于 Cache 没有自动更新的特性，用户在使能 Cache 时必须验证数据的一致性。

片内 8KB 的 SRAM 可以通过设置 SYSCFG[5:4]用作 Cache，如果不准备将 8KB 全部用作 Cache，也可将余下的空间作为片内的 SRAM，其起始地址由片内 SRAM 的基指针设定。

写缓冲禁止/允许（Write Buffer Disable/Enable）

S3C4510B 内含 4 个可编程的写缓冲寄存器以提高存储器写操作的速度。当使能写缓冲器，CPU 首先将数据写入写缓冲器，而不直接写入外部的存储器。4 个写缓冲寄存器可以增强 ARM7TDMI 内核的数据存储操作性能。

4. 系统时钟和 MUX 总线控制寄存器(System clock and MUX Bus Control Register)

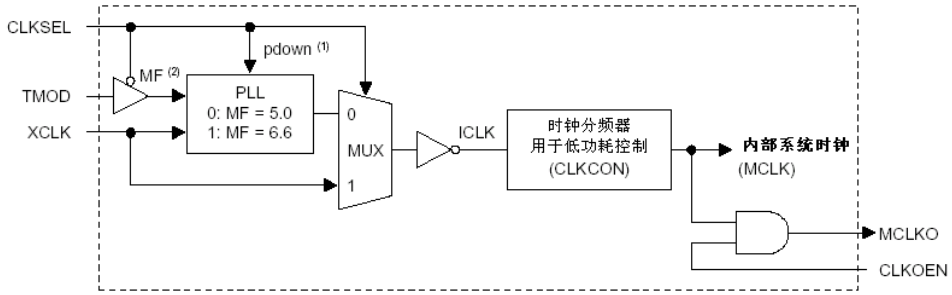
时钟控制寄存器(Clock Control Register，CLKCON)

在系统管理器中有一个时钟控制寄存器，可对系统时钟分频。通过设置时钟的分频值可降低系统的工作频率，该寄存器具体定义与设置，请参考 S3C4510B 用户手册。

系统时钟(System Clock)

如果将 S3C4510B 的 CLKSEL 引脚置为高电平，由 XCLK 引脚输入的外部时钟就直接作为内部系统时钟，若将 CLKSEL 引脚置为低电平，外部时钟则会通过片内的 PLL 电路后才作为内部系统时钟，在这种情况下，内部系统时钟等于 XCLK×MF（倍乘因子）。例如，要得到 50MHz 的内部系统时钟，只需要 10MHz 的外部时钟。

图 5.2.7 为系统时钟电路。



注： 1、若CLKSET为高电平，PLL电路不工作。
2、MF：倍频因子（Multiplication Factor）

图 5.2.7 系统时钟电路

外部 I/O 访问控制寄存器(External I/O Access Control Registers，EXTACON0/1)

系统管理器内的外部 I/O 访问控制寄存器用于控制外部 I/O 的访问周期，该寄存器具体定义与设置，请参考 S3C4510B 用户手册。

数据总线宽度寄存器(Data Bus Width Register，EXTDBWTH)

S3C4510B 可以以 8/16/32 位的数据宽度访问外部 ROM、SRAM、Flash 存储器，DRAM、SDRAM 以及外部 I/O 口。通过设置数据宽度寄存器，用户可设定与特定外部存储器和外部 I/O 组相对应的数据宽度。

EXTDBWTH 寄存器

寄存器	偏移地址	操作	描 述	复位值
EXTDBWTH	0x3010	读/写	配置每组的数据总线宽度	0x00000000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTDBWTH	0	0	0	0	DSX3		DSX2		DSX1		DSX0		DSD3		DSD2		DSD1		DSD0		DSR5		DSR4		DSR3		DSR2		DSR1		DSR0	

[1:0]ROM/SRAM/FLASH 组 0 的数据总线宽度（DSR0）

DSR0 的属性为只读，ROM/SRAM/FLASH 组 0 在系统中通常作为启动 ROM 区，其值由 BOSIZE[1：0]引脚的状态决定。

00 = 不允许

01 = 字节（8 位）

10 = 半字（16 位）

11 = 字（32 位）

[3:2]ROM/SRAM/FLASH 组 1 的数据总线宽度（DSR1）

[5:4] DSR2，[7:6] DSR3，[9:8] DSR4，[11:10] DSR5

00 = 禁止

01 = 字节（8 位）

10 = 半字（16 位）

11 = 字（32 位）

[13:12]DRAM 组 0 的数据总线宽度（DSD0）

[15:14] DSD1，[17:16] DSD2，[19:18] DSD3

00 = 禁止

01 = 字节（8 位）

10 = 半字（16 位）

11 = 字（32 位）

[21:20]外部 I/O 组 0 的数据总线宽度（DSX0）

[23:22] DSX1，[25:24] DSX2，[27:26] DSX3

00 = 禁止

01 = 字节（8 位）

10 = 半字（16 位）

11 = 字（32 位）

当选择“禁止”时，S3C4510B 不产生相应的访问信号。

ROM/SRAM/Flash 控制寄存器(ROM/SRAM/FLASH Control Register，ROMCON)

系统管理器内含 6 个用于控制 ROM、SRAM、Flash 存储器的寄存器，分别对应于 S3C4510B 所支持的 6 个 ROM/SRAM/FLASH 组。

对于 ROM/SRAM/FLASH 组 0，其外部数据总线宽度由 BOSIZE[1：0]引脚的状态决定：

当 BOSIZE[1：0]= “01”，ROM/SRAM/FLASH 组 0 的外部数据总线宽度为 8 位。

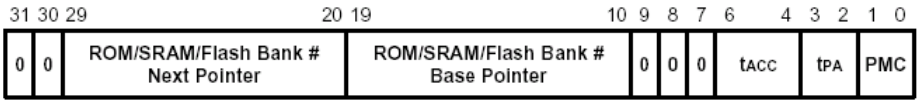
当 BOSIZE[1：0] = “10”，ROM/SRAM/FLASH 组 0 的外部数据总线宽度为 16 位。

当 BOSIZE[1：0] = “11”，ROM/SRAM/FLASH 组 0 的外部数据总线宽度为 32 位。

ROM/SRAM/Flash 控制寄存器

寄存器	偏移地址	操作	描　　述	复位值
ROMCON0	0x3014	读/写	ROM/SRAM/FLASH 组 0 控制寄存器	0x20000060
ROMCON1	0x3018	读/写	ROM/SRAM/FLASH 组 1 控制寄存器	0x00000060
ROMCON2	0x301C	读/写	ROM/SRAM/FLASH 组 2 控制寄存器	0x00000060
ROMCON3	0x3020	读/写	ROM/SRAM/FLASH 组 3 控制寄存器	0x00000060
ROMCON4	0x3024	读/写	ROM/SRAM/FLASH 组 4 控制寄存器	0x00000060
ROMCON5	0x3028	读/写	ROM/SRAM/FLASH 组 5 控制寄存器	0x00000060

以下是 ROM/SRAM/Flash 控制寄存器（ROMCON0－ROMCON5）每位的定义：



[1:0]页模式配置（PMC）

00 = 普通 ROM　　01 = 4 字/页

10 = 8 字/页　　11 = 16 字/页

[3:2]页地址访问时间（tPA）

00 = 5 个周期　　01 = 2 个周期

10 = 3 个周期　　11 = 4 个周期

[6:4]可编程访问周期（tACC）

000 = 该组禁用　　001 = 2 个周期

010 = 3 个周期　　011 = 4 个周期

110 = 7 个周期　　111 = 保留

[19:10] ROM/SRAM/Flash 组基指针

该设置值左移 16 位即为 ROM/SRAM/Flash 组的起始物理地址。

[29:20] ROM/SRAM/Flash 组尾指针

该设置值左移 16 位-1 即为 ROM/SRAM/Flash 组的结束物理地址。

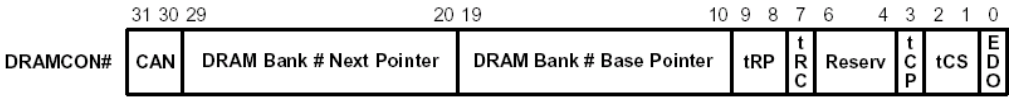
DRAM 控制寄存器(DRAM Control Registers)

系统管理器内含 4 个 DRAM 控制寄存器，DRAMCON0～DRAMCON3，分别对应于 S3C4510B 所支持的 4 个 DRAM 组。REFEXTCON 寄存器用于设置外部 I/O 组 0 的基指针。

S3C4510B 支持 EDO、普通、同步 DRAM（SDRAM）。通过将 SYSCFG[31]置 1 可选择 SDRAM 模式，如果将该位置 1，所有的 DRAM 组均选择 SDRAM，置 0 则所有 DRAM 组均选择 EDO/FP DRAM。

DRAM 及外部 I/O 控制寄存器

寄存器	偏移地址	操作	描　　述	复位值
DRAMCON0	0x302C	读/写	DRAM 组 0 控制寄存器	0x00000000
DRAMCON1	0x3030	读/写	DRAM 组 1 控制寄存器	0x00000000
DRAMCON2	0x3034	读/写	DRAM 组 2 控制寄存器	0x00000000
DRAMCON3	0x3038	读/写	DRAM 组 3 控制寄存器	0x00000000
REFEXTCON	0x303C	读/写	刷新与外部 I/O 控制寄存器	0x00000000



[0]EDO 模式（EDO）

0 = 普通 DRAM（快速页模式 DRAM）

1 = EDO DRAM

[2:1]CAS 锁存时间（tCS）

00 = 1 个周期　　01 = 2 个周期

10 = 3 个周期　　11 = 4 个周期

[3]CAS 预充电时间（tCP）

0 = 1 个周期　　1 = 2 个周期

[6:4]系统保留

系统默认值为 000，但用户在使用时应置为 001

[7]RAS 到 CAS 的延迟（tRC 或 tRCD）

0 = 1 个周期　　1 = 2 个周期

[9:8]RAS 预充电时间（tRP）

00 = 1 个周期　　01 = 2 个周期

10 = 3 个周期　　11 = 4 个周期

[19:10] DRAM 组基指针

该设置值左移 16 位即为 DRAM 组的起始物理地址。

[29:20] DRAM 组尾指针

该设置值左移 16 位－1 即为 DRAM 组的结束物理地址。

[31:30] DRAM 组的列地址位数（CAN）

00 = 8 位　　01 = 9 位

10 = 10 位　　11 = 11 位

DRAM 接口特性(DRAM Interface Features)

S3C4510B 具有完全可编程的外部 DRAM 接口，通过设置相应的 DRAM 控制寄存器，用户可方便的控制各种接口特性。这些可编程的特性包括：

- 外部数据总线宽度。
- 通过 DRAMCON[0]选择快速页模式或 EDO 模式。
- 通过 SYSCFG[31]选择快速页模式/EDO 模式或 SDRAM 模式。
- 每个 DRAM 组的访问周期数、CAS 锁存时间、CAS 预充电时间、RAS 到 CAS 时延、RAS 预充电时间等。

刷新及外部 I/O 控制寄存器（REFEXTCON），用于控制 DRAM 的刷新操作及对外部 I/O 组的访问。通过自动产生刷新控制信号，S3C4510B 系统不再需要其他的外部刷新信号刷新 DRAM。

通过 23 位内部地址总线，S3C4510B 可产生用于 DRAM 访问的行、列地址信号。同时，S3C4510 支持同步或异步 DRAM。

访问 SDRAM (Synchronous DRAM Accesses)

SDRAM 的接口控制信号包括 CKE、SDCLK、nSDCS[3:0]、nSDCAS、nSDRAS 及 DQM[3:0]等。SDRAM 的接口方法将在硬件设计部分详细描述。

DRAM 组的地址空间(DRAM Bank Space)

S3C4510B 的 DRAM 接口支持 4 个 DRAM 组，各 DRAM 组可以是不同的配置。用户可通过 DRAM 控制寄存器（DRAMCON0～DRAMCON3）设置 DRAM 访问周期和存储器组在地址空间中的位置。

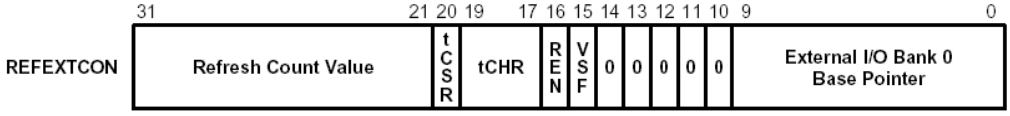
每一个 DRAM 控制寄存器都包含两个 10 位的地址指针，首指针和尾指针，用以指示 DRAM 组的起始地址和结束地址。10 位的指针值对应 A[25:16],因此，在每一个组内，其地址空间为 2^{16}=64KB。

系统初始化值(System Initialization Values)

当系统初始化时，4 个 DRAM 控制寄存器均被置为 0x00000000，同时禁用所有的外部 DRAM。

DRAM 刷新与外部 I/O 控制寄存器(DRAM Refresh and External I/O Control Register，REFEXTCON)

S3C4510B 的 DRAM 接口支持对 EDO/FP DRAM 的 CBR（CAS-before-RAS）刷新模式和 SDRAM 的自动刷新。通过设置 DRAM 刷新与外部 I/O 控制寄存器，REFEXTCON，可控制 DRAM 的刷新模式、刷新时序及刷新间隔。REFEXTCON 同时还包含 10 位外部 I/O 组 0 的基指针值。



[9：0]外部 I/O 组 0 基指针（基地址）

该设置值左移 16 位即为外部 I/O 组 0 的起始物理地址。外部 I/O 组 0 的结束物理地址为该设置值左移 16 位+16KB-1。

4 个外部 I/O 组的地址空间连续，每组均固定大小为 16KB，因此，通过该基指针可以计算出其他 3 个外部 I/O 组的起始和结束地址。

[15]特殊功能寄存器域有效性（VSF）

0 = 存储器组不可访问

1 = 存储器组可访问

[16]刷新使能（REN）

0 = 禁止 DRAM 刷新

1 = 使能 DRAM 刷新

[19:17]CAS 持续时间（tCHR）

000 = 1 个周期　　001 = 2 个周期

010 = 3 个周期　　011 = 4 个周期

100 = 5 个周期　　101 = 未用

110 = 未用　　111 = 未用

[20]CAS 设置时间（tCSR）

0 = 1 个周期

1 = 2 个周期

[31：21]刷新计数值（持续时间）

刷新周期 = (2^{11}-该值+1)/fMCK

以上是对 S3C4510B 的系统管理器及相关特殊功能寄存器的简单介绍，这些内容将会在程序设计，特别是系统初始化部分的程序设计中反复用到。限于篇幅，还有部分系统管理器的特殊功能寄存器未做介绍，请读者自行参考 S3C4510B 用户手册。

5.3 系统的硬件选型与单元电路设计

从这一节开始，将详细描述系统的硬件选型与设计，希望通过对这些章节的阅读，能使读者具有初步设计特定系统的能力。

尽管硬件选型与单元电路设计部分的内容是基于 S3C4510B 的系统，但由于 ARM 体系结构的一致性和常见外围电路的通用性，只要读者能真正理解本部分的设计方法，从而设计出基于其他 ARM 微处理器的系统，应该也是比较容易的。

需要说明，以下的应用电路可能不是最佳的，但经验证是可以正常工作的。

5.3.1 S3C4510B 芯片及引脚分析

S3C4510B 共有 208 只引脚，采用 QFP 封装，这对于那些常使用 8 位/16 位 DIP 封装微控制器的读者来说，可能会觉得有点复杂，然而，尽管 S3C4510B 引脚较多，但根据各自的功能，分布很有规律。

首先，电源和接地引脚有近 50 根，再除去地址总线、数据总线和通用 I/O 口，以及其他的专用模块如 HDLC、UART、IIC、MAC 等的接口，真正需要仔细研究的引脚数就不是很多了，但这些引脚主要是控制信号，需要认真对待，在此先进行简单的分析，其后的单元电路设计里，会有更再详细的说明。

在硬件系统的设计中，应当注意芯片引脚的类型，S3C4510B（也包括其他的微处理器）的引脚主要分为三类，即：输入（I）、输出（O）、输入/输出（I/O）。

输出类型的引脚主要用于 S3C4510B 对外设的控制或通信，由 S3C4510B 主动发出，这些引脚的连接不会对 S3C4510B 自身的运行有太大的影响。

输入/输出类型的引脚主要是 S3C4510B 与外设的双向数据传输通道。

而某些输入类型的引脚，其电平信号的设置是 S3C4510B 本身正常工作的前提，在系统设计时必须小心处理。

S3C4510B 的主要控制信号如下：

LITTLE（Pin49）：**大、小端模式选择引脚**。高电平 = 小端模式；低电平 = 大端模式；该引脚在片内下拉，系统默认为大端模式，但在实际系统中一般使用小端模式，更符合我们的使用习惯，因此该引脚可上拉或接电源。

FILTER（Pin55）：如果使用 PLL 倍频电路，应在该引脚和地之间接 820pF 的陶瓷电容。在实际系统中，一般应使用 PLL 电路，因此，该电容应连接。

TCK、TMS、TDI、TDO、nTRST（Pin58～Pin62）：**JTAG 接口引脚**。根据 IEEE 标准，TCK 应下拉，TMS、TDI 和 nTRST 应上拉。S3C4510B 已按此标准在片内连接，只需要与 JTAG 插座直接相连即可，但某些 ARM 芯片并未做相应的处理，在设计电路时应注意。

TMODE（Pin63）：**测试模式**。高电平 = 芯片测试模式；低电平 = 正常工作模式；用户一般不作芯片测试，该引脚下拉或接地，使芯片处于正常工作模式。

nEWAIT（Pin71）：**外部等待请求信号**。该引脚应上拉。

BOSIZE[1:0]（Pin74,Pin73）：**BANK0 数据宽度选择**。’01’ = 8 位；’10’ = 16 位；’11’ = 32 位；’00’ = 系统保留。

CLKOEN（Pin76）：**时钟输出允许/禁止**。高电平 = 允许；低电平 = 禁止。一些外围器件（如 SDRAM）需要 CPU 的时钟输出作为自身的时钟源，该引脚一般接高电平，使时钟输出为允许状态。

XCLK（Pin80）：**系统时钟源**。接有源晶振的输出。

nRESET（Pin82）：**系统复位引脚**。低电平复位，当系统正常工作时，该引脚应处于高电平状态。

CLKSEL（Pin83）：**时钟选择**。高电平 = XCLK 直接作为系统的工作时钟；低电平 = XCLK 经

过 PLL 电路倍频后作为系统的工作时钟。

ExtMREQ（Pin108）：外部主机总线请求信号。该引脚应下拉。

S3C4510B 的其余引脚为电源线、接地线、数据总线、地址总线以及其他功能模块地输入/输出线，对 CPU 自身地运行地影响相对较小，其连接方式也比较简单，在此不作详述。

5.3.2 电源电路

在该系统中，需要使用 5V 和 3.3V 的直流稳压电源，其中，S3C4510B 及部分外围器件需 3.3V 电源，另外部分器件需 5V 电源，为简化系统电源电路的设计，要求整个系统的输入电压为高质量的 5V 的直流稳压电源。系统电源电路如下图所示：

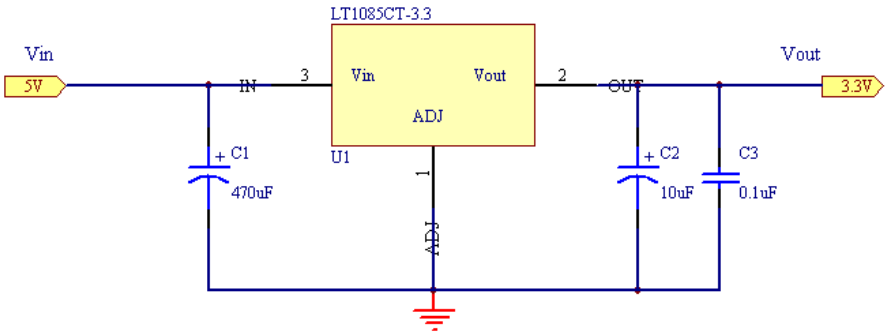


图 5.3.1 系统的电源电路

有很多 DC-DC 转换器可完成 5V 到 3.3V 的转换，在此选用 Linear Technology 的 LT108X 系列。常见的型号和对应的电流输出如下：

LT1083	7.5A
LT1084	5A
LT1085	3A
LT1086	1.5A

设计者可根据系统的实际功耗，选择不同的器件。

5.3.3 晶振电路与复位电路

晶振电路用于向 CPU 及其他电路提供工作时钟。在该系统中，S3C4510B 使用有源晶振。不同于常用的无源晶振，有源晶振的接法略有不同。常用的有源晶振的接法如下图所示：

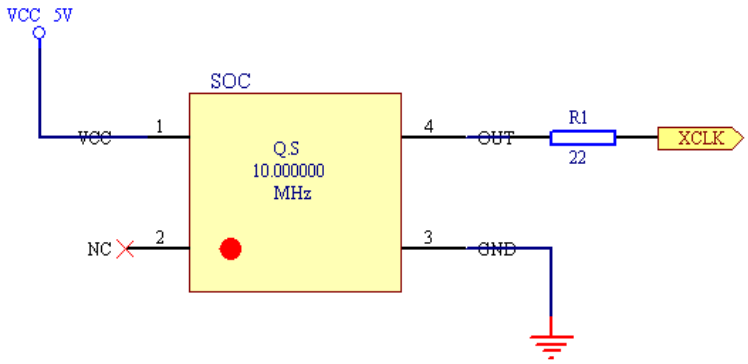


图 5.3.2　系统的晶振电路

根据 S3C4510B 的最高工作频率以及 PLL 电路的工作方式，选择 10MHz 的有源晶振，10MHz 的晶振频率经过 S3C4510B 片内的 PLL 电路倍频后，最高可以达到 50MHz。片内的 PLL 电路兼有频率放大和信号提纯的功能，因此，系统可以以较低的外部时钟信号获得较高的工作频率，以降低因高速开关时钟所造成的高频噪声。

有源晶振的 1 脚接 5V 电源，2 脚悬空，3 脚接地，4 脚为晶振的输出，可通过一个小电阻（此处为 22 欧姆）接 S3C4510B 的 XCLK 引脚。

在系统中，复位电路主要完成系统的上电复位和系统在运行时用户的按键复位功能。复位电路可由简单的 RC 电路构成，也可使用其他的相对较复杂，但功能更完善的电路。

本系统采用较简单的 RC 复位电路，经使用证明，其复位逻辑是可靠的。复位电路如图 5.3.3 所示：

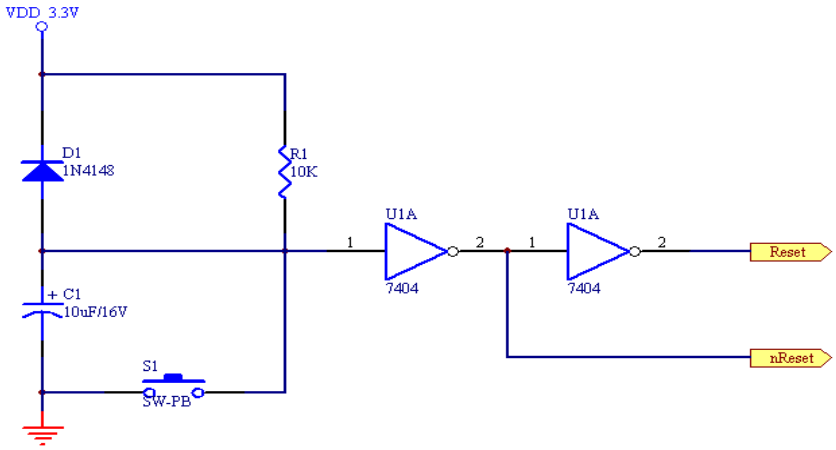


图 5.3.3　系统的复位电路

该复位电路的工作原理如下：在系统上电时，通过电阻 R1 向电容 C1 充电，当 C1 两端的电压未达到高电平的门限电压时，Reset 端输出为低电平，系统处于复位状态；当 C1 两端的电压达到高电平的门限电压时，Reset 端输出为高电平，系统进入正常工作状态。

当用户按下按钮 S1 时，C1 两端的电荷被泻放掉，Reset 端输出为低电平，系统进入复位状态，再重复以上的充电过程，系统进入正常工作状态。

两级非门电路用于按钮去抖动和波形整形；nReset 端的输出状态与 Reset 端相反，以用于高电平复位的器件；通过调整 R1 和 C1 的参数，可调整复位状态的时间。

5.3.4 Flash 存储器接口电路

Flash 存储器是一种可在系统（In-System）进行电擦写，掉电后信息不丢失的存储器。它具有低功耗、大容量、擦写速度快、可整片或分扇区在系统编程（烧写）、擦除等特点，并且可由内部嵌入的算法完成对芯片的操作，因而在各种嵌入式系统中得到了广泛的应用。作为一种非易失性存储器，Flash 在系统中通常用于存放程序代码、常量表以及一些在系统掉电后需要保存的用户数据等。常用的 Flash 为 8 位或 16 位的数据宽度，编程电压为单 3.3V。主要的生产厂商为 ATMEL、AMD、HYUNDAI 等，他们生产的同型器件一般具有相同的电气特性和封装形式，可通用。

以该系统中使用的 Flash 存储器 HY29LV160 为例，简要描述一下 Flash 存储器的基本特性：

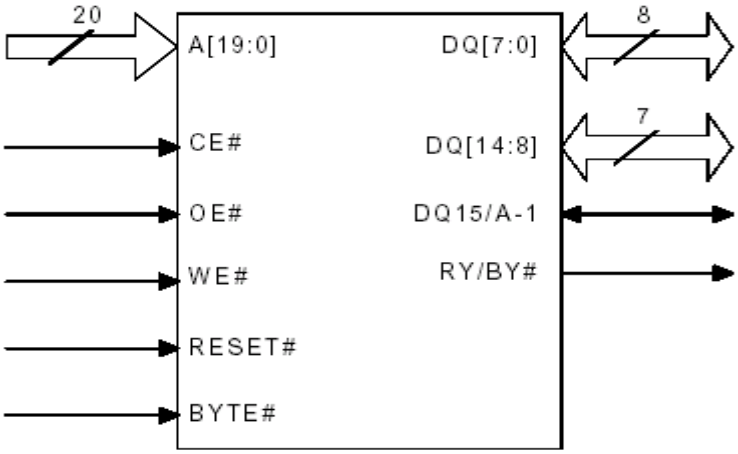


图 5.3.4 HY29LV160 的逻辑框图

HY29LV160 的单片存储容量为 16M 位（2M 字节），工作电压为 2.7V～3.6V，采用 48 脚 TSOP 封装或 48 脚 FBGA 封装，16 位数据宽度，可以以 8 位（字节模式）或 16 位（字模式）数据宽度的方式工作。

HY29LV160 仅需单 3V 电压即可完成在系统的编程与擦除操作，通过对其内部的命令寄存器写入标准的命令序列，可对 Flash 进行编程（烧写）、整片擦除、按扇区擦除以及其他操作。

HY29LV160 的逻辑框图、引脚分布及信号描述分别如图 5.3.4、图 5.3.5 和表 5-3-1 所示：

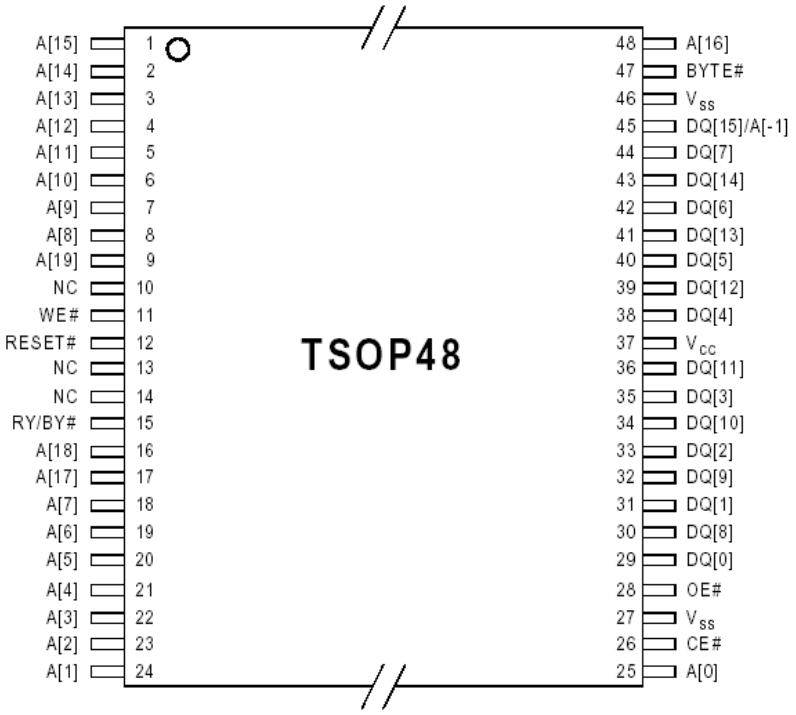


图 5.3.5 HY29LV160 引脚分布（TSOP48 封装）

表 5-3-1 HY29LV160 的引脚信号描述

引 脚	类型	描 述
A[19:0]	I	**地址总线**。在字节模式下，DQ[15]/A[-1]用作 21 位字节地址的最低位。
DQ[15]/A[-1] DQ[14:0]	I/O 三态	**数据总线**。在读写操作时提供 8 位或 16 位的数据宽度。在字节模式下，DQ[15]/A[-1]用作 21 位字节地址的最低位，而 DQ[14:8]处于高阻状态。
BYTE#	I	**模式选择**。低电平选择字节模式，高电平选择字模式
CE#	I	**片选信号，低电平有效**。在对 HY29LV160 进行读写操作时，该引脚必须为低电平，当为高电平时，芯片处于高阻旁路状态
OE#	I	**输出使能，低电平有效**。在读操作时有效，写操作时无效。
WE#	I	**写使能，低电平有效**。在对 HY29LV160 进行编程和擦除操作时，控制相应的写命令。
RESET#	I	**硬件复位，低电平有效**。对 HY29LV160 进行硬件复位。当复位时，HY29LV160 立即终止正在进行的操作。
RY/BY#	O	**就绪/忙 状态指示**。用于指示写或擦除操作是否完成。当 HY29LV160 正在进行编程或擦除操作时，该引脚位低电平，操作完成时为高电平，此时可读取内部的数据。
VCC	--	3.3V 电源
VSS	--	接地

以上为一款常见的 Flash 存储器 HY29LV160 的简介，更具体的内容可参考 HY29LV160 的用户手册。其他类型的 Flash 存储器的特性与使用方法与之类似，用户可根据自己的实际需要选择不同的器件。

下面，我们使用 HY29LV160 来构建 Flash 存储系统。由于 ARM 微处理器的体系结构支持 8 位/16 位/32 位的存储器系统，对应的可以构建 8 位的 Flash 存储器系统、16 位的 Flash 存储器系统或 32 位的 Flash 存储器系统。32 位的存储器系统具有较高的性能，而 16 位的存储器系统则在成本及功耗方面占有优势，而 8 位的存储器系统现在已经很少使用。在此，分别介绍 16 位和 32 位的 Flash 存储器系统的构建。

16 位的 FLASH 存储器系统：

图 5.3.6 为 16 位 Flash 存储器系统的实际应用电路图。

在大多数的系统中，选用一片 16 位的 Flash 存储器芯片（常见单片容量有 1MB、2MB、4MB、8MB 等）构建 16 位的 Flash 存储系统已经足够，在此采用一片 HY29LV160 构建 16 位的 Flash 存储器系统，其存储容量为 2MB。Flash 存储器在系统中通常用于存放程序代码，系统上电或复位后从此获取指令并开始执行，因此，应将存有程序代码的 Flash 存储器配置到 ROM/SRAM/FLASH Bank0，即将 S3C4510B 的 nRCS<0>（Pin75）接至 HY29LV160 的 CE＃端。

HY29LV160 的 RESET＃端接系统复位信号；

OE＃端接 S3C4510B 的 nOE（Pin72）；

WE＃端 S3C4510B 的 nWBE<0>（Pin100）；

BYTE＃上拉，使 HY29LV160 工作在字模式（16 位数据宽度）；

RY/BY#指示 HY29LV160 编程或擦除操作的工作状态，但其工作状态也可通过查询片内的相关寄存器来判断，因此可将该引脚悬空；

地址总线[A19～A0]与 S3C4510B 的地址总线[ADDR19～ADDR0]相连；

16 位数据总线[DQ15~DQ0]与 S3C4510B 的低 16 位数据总线[XDATA15~XDATA0]相连。

注意此时应将 S3C4510B 的 BOSIZE[1:0]置为‘10’，选择 ROM/SRAM/FLASH Bank0 为 16 位工作方式。

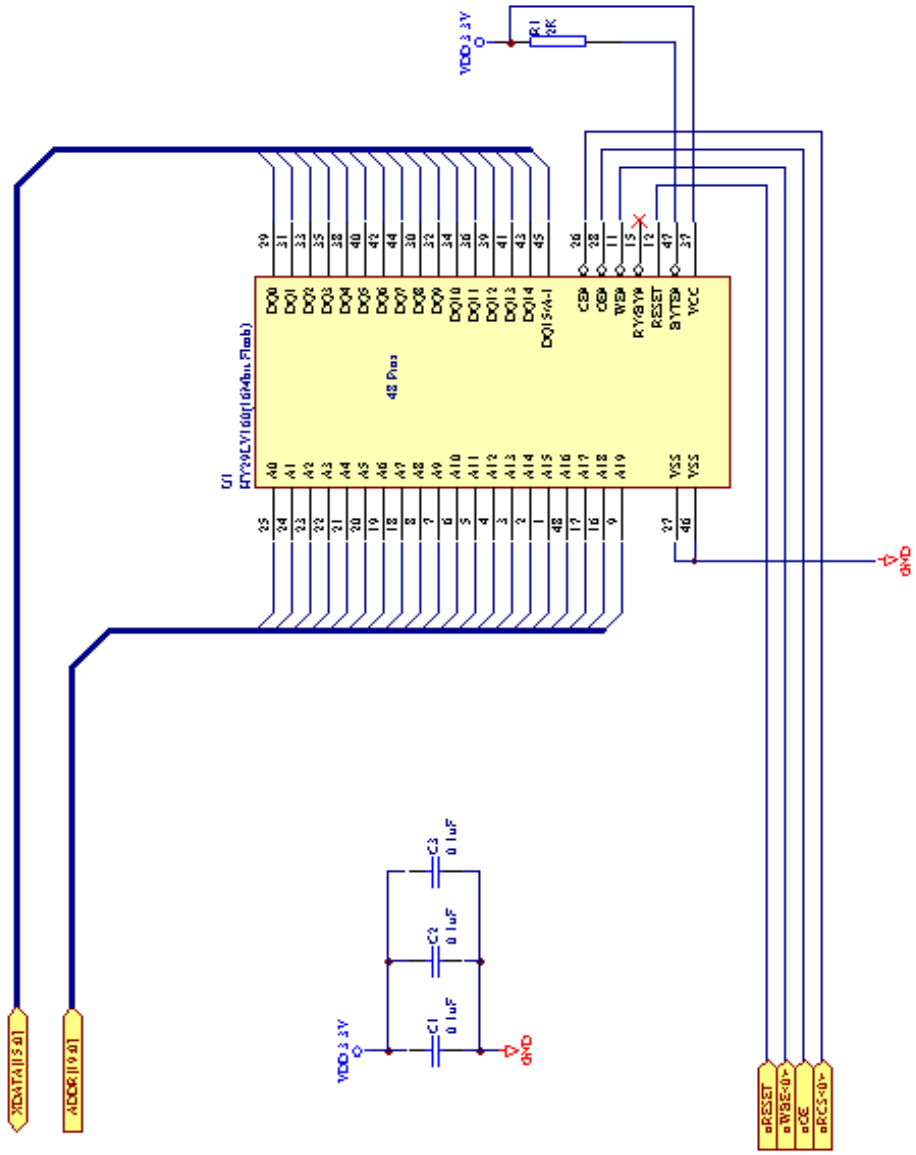


图 5.3.6

16 位 FLASH 存储系统电路图

32 位的 FLASH 存储器系统：

图 5.3.7 为 32 位 Flash 存储器系统的实际应用电路图。

作为一款 32 位的微处理器，为充分发挥 S3C4510B 的 32 性能优势，有的系统也采用两片 16 位数据宽度的 Flash 存储器芯片并联（或一片 32 位数据宽度的 Flash 存储器芯片）构建 32 位的 Flash 存储系统。其构建方式与 16 位的 Flash 存储器系统相似。

采用两片 HY29LV16 并联的方式构建 32 位的 FLASH 存储器系统，其中一片为高 16 位，另一片为低 16 位，将两片 HY29LV16 作为一个整体配置到 ROM/SRAM/FLASH Bank0，即将 S3C4510B 的 nRCS<0>（Pin75）接至两片 HY29LV16 的 CE＃端；

两片 HY29LV160 的 RESET＃端接系统复位信号；

两片 HY29LV160 的 OE＃端接 S3C4510B 的 nOE（Pin72）；

低 16 位片的 WE＃端接 S3C4510B 的 nWBE<0>（Pin100），高 16 位片的 WE＃端接 S3C4510B 的 nWBE<2>（Pin102）；

两片 HY29LV160 的 BYTE＃均上拉，使之均工作在字模式；

两片 HY29LV160 的地址总线[A19～A0]均与 S3C4510B 的地址总线[ADDR19～ADDR0]相连；

低 16 位片的数据总线与 S3C4510B 的低 16 位数据总线[XDATA15~XDATA0]相连，高 16 位片的数据总线与 S3C4510B 的高 16 位数据总线[XDATA31~XDATA16]相连。

注意此时应将 S3C4510B 的 BOSIZE[1:0]置为‘11’，选择 ROM/SRAM/FLASH Bank0 为 32 位工作方式。

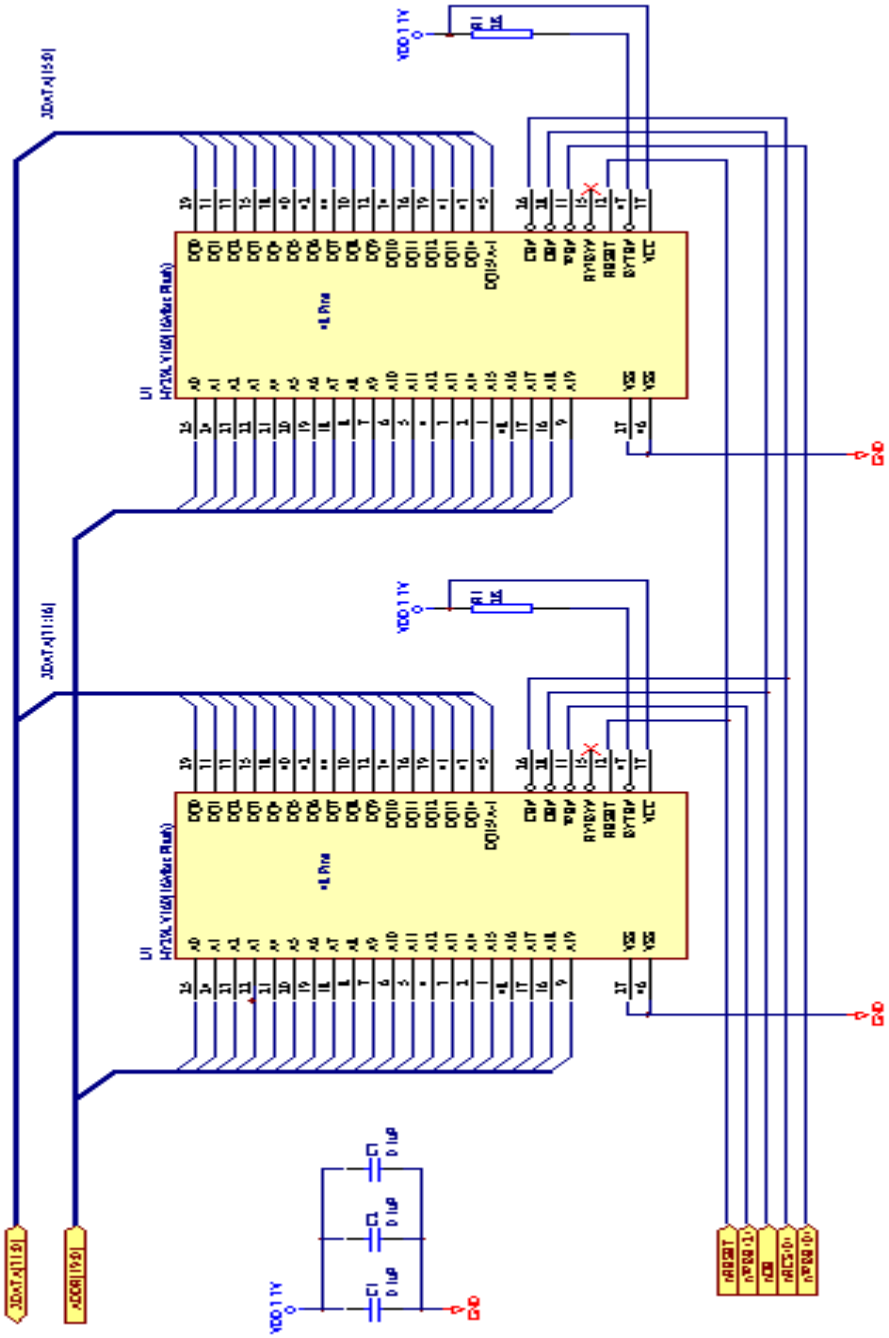


图 5.3.7 32 位 FLASH 存储系统电路图

5.3.5 SDRAM 接口电路

与 Flash 存储器相比较，SDRAM 不具有掉电保持数据的特性，但其存取速度大大高于 Flash 存储器，且具有读/写的属性，因此，SDRAM 在系统中主要用作程序的运行空间，数据及堆栈区。当

系统启动时，CPU 首先从复位地址 0x0 处读取启动代码，在完成系统的初始化后，程序代码一般应调入 SDRAM 中运行，以提高系统的运行速度，同时，系统及用户堆栈、运行数据也都放在 SDRAM 中。

SDRAM 具有单位空间存储容量大和价格便宜的优点，已广泛应用在各种嵌入式系统中。SDRAM 的存储单元可以理解为一个电容，总是倾向于放电，为避免数据丢失，必须定时刷新（充电）。因此，要在系统中使用 SDRAM，就要求微处理器具有刷新控制逻辑，或在系统中另外加入刷新控制逻辑电路。S3C4510B 及其他一些 ARM 芯片在片内具有独立的 SDRAM 刷新控制逻辑，可方便的与 SDRAM 接口。但某些 ARM 芯片则没有 SDRAM 刷新控制逻辑，就不能直接与 SDRAM 接口，在进行系统设计时应注意这一点。

目前常用的 SDRAM 为 8 位/16 位的数据宽度，工作电压一般为 3.3V。主要的生产厂商为 HYUNDAI、Winbond 等。他们生产的同型器件一般具有相同的电气特性和封装形式，可通用。

以该系统中使用的 HY57V641620 为例，简要描述一下 SDRAM 的基本特性及使用方法：

HY57V641620 存储容量为 4 组×16M 位（8M 字节），工作电压为 3.3V，常见封装为 54 脚 TSOP，兼容 LVTTL 接口，支持自动刷新（Auto-Refresh）和自刷新（Self-Refresh），16 位数据宽度。

HY57V641620 引脚分布及信号描述分别如图 5.3.8 和表 5-3-2 所示：

表 5-3-2　HY57V641620 引脚信号描述

引 脚	名称	描　　述
CLK	时钟	芯片时钟输入。
CKE	时钟使能	片内时钟信号控制。
/CS	片选	禁止或使能除 CLK、CKE 和 DQM 外的所有输入信号。
BA0，BA1	组地址选择	用于片内 4 个组的选择。
A11～A0	地址总线	行地址：A11～A0，列地址：A7～A0，自动预充电标志：A10
/RAS， /CAS， /WE	行地址锁存 列地址锁存 写使能	参照功能真值表，/RAS，/CAS 和/WE 定义相应的操作。
LDQM，UDQM	数据 I/O 屏蔽	在读模式下控制输出缓冲；在写模式下屏蔽输入数据
DQ15～DQ0	数据总线	数据输入输出引脚
VDD/VSS	电源/地	内部电路及输入缓冲电源/地
VDDQ/VSSQ	电源/地	输出缓冲电源/地
NC	未连接	未连接

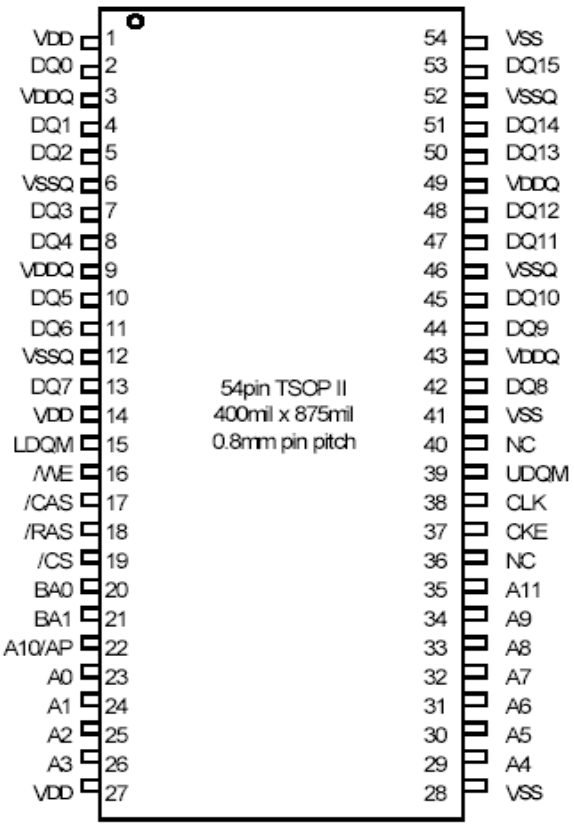


图 5.3.8　HY57V641620 引脚分布

以上为一款常见的 SDRAM HY57V641620 的简介，更具体的内容可参考 HY57V641620 的用户手册。其他类型 SDRAM 的特性与使用方法与之类似，用户可根据自己的实际需要选择不同的器件。

根据系统需求，可构建 16 位或 32 位的 SDRAM 存储器系统，但为充分发挥 32 位 CPU 的数据处理能力，大多数系统采用 32 位的 SDRAM 存储器系统。

HY57V641620 为 16 位数据宽度，单片容量为 8MB，系统选用的两片 HY57V641620 并联构建 32 位的 SDRAM 存储器系统，共 16MB 的 SDRAM 空间，可满足嵌入式操作系统及各种相对较复杂的算法的运行要求。

图 5.3.9 为 32 位 SDRAM 存储器系统的实际应用电路图。

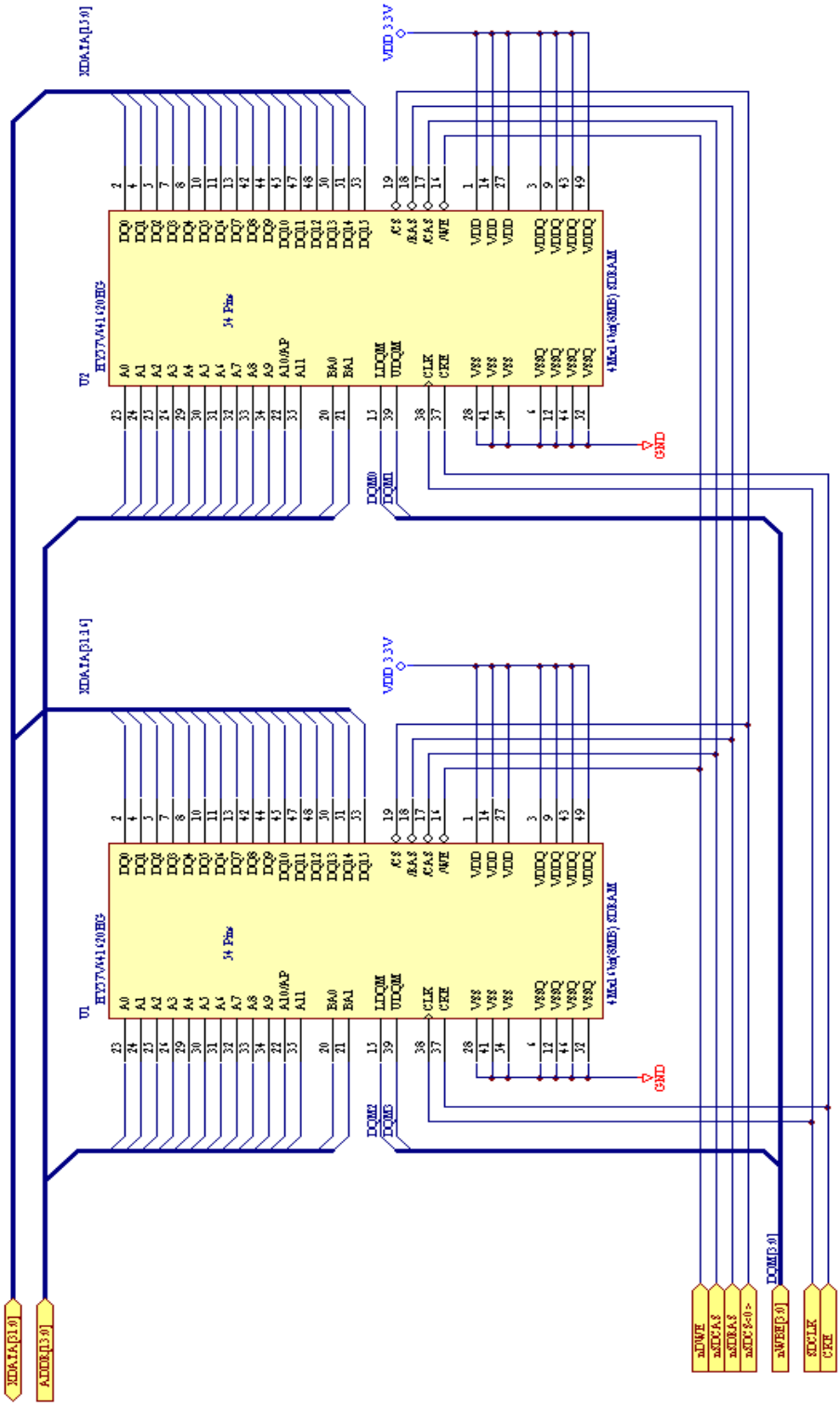


图 5.3.9　32 位 SDRAM 存储器系统的实际应用电路图

与 Flash 存储器相比，SDRAM 的控制信号较多，其连接电路也要相对复杂。

两片 HY57V641620 并联构建 32 位的 SDRAM 存储器系统，其中一片为高 16 位，另一片为低 16 位，可将两片 HY57V641620 作为一个整体配置到 DRAM/SDRAM Bank0～DRAM/SDRAM Bank3 的任一位置，一般配置到 DRAM/SDRAM Bank0，即将 S3C4510B 的 nSDCS<0>（Pin89）接至两片 HY57V641620 的/CS 端。

两片 HY57V641620 的 CLK 端接 S3C4510B 的 SDCLK 端（Pin77）；

两片 HY57V641620 的 CLE 端接 S3C4510B 的 CLE 端（Pin97）；

两片 HY57V641620 的/RAS、/CAS、/WE 端分别接 S3C4510B 的 nSDRAS 端（Pin95）、nSDCAS

端（Pin96）、nDWE 端（Pin99）；

两片 HY57V641620 的 A11～A0 接 S3C4510B 的地址总线 ADDR<11>～ADDR<0>；

两片 HY57V641620 的 BA1、BA0 接 S3C4510B 的地址总线 ADDR<13>、ADDR<12>；

高 16 位片的 DQ15～DQ0 接 S3C4510B 的数据总线的高 16 位 XDATA<31>～XDATA<16>，低 16 位片的 DQ15～DQ0 接 S3C4510B 的数据总线的低 16 位 XDATA<15>～XDATA<0>；

高 16 位片的 UDQM、LDQM 分别接 S3C4510B 的 nWEB<3>、nWEB<2>，低 16 位片的 UDQM、LDQM 分别接 S3C4510B 的 nWEB<1>、nWEB<0>。

5.3.6 串行接口电路

几乎所有的微控制器、PC 都提供串行接口，使用电子工业协会（EIA）推荐的 RS-232-C 标准，这是一种很常用的串行数据传输总线标准。早期它被应用于计算机和终端通过电话线和 MODEM 进行远距离的数据传输，随着微型计算机和微控制器的发展，不仅远距离，近距离也采用该通信方式。在近距离通信系统中，不再使用电话线和 MODEM，而直接进行端到端的连接。

RS-232-C 标准采用的接口是 9 芯或 25 芯的 D 型插头，以常用的 9 芯 D 型插头为例，各引脚定义如表 5-3-3 所示：

表 5-3-3 9 芯 D 型插头引脚信号描述

引 脚	名 称	功 能 描 述
1	DCD	数据载波检测
2	RXD	数据接收
3	TXD	数据发送
4	DTR	数据终端准备好
5	GND	地
6	DSR	数据设备准备好
7	RTS	请求发送
8	CTS	清除发送
9	RI	振铃指示

要完成最基本的串行通信功能，实际上只需要 RXD、TXD 和 GND 即可，但由于 RS-232-C 标准所定义的高、低电平信号与 S3C4510B 系统的 LVTTL 电路所定义的高、低电平信号完全不同，LVTTL 的标准逻辑“1”对应 2V～3.3V 电平，标准逻辑“0”对应 0V～0.4V 电平，而 RS-232-C 标准采用负逻辑方式，标准逻辑“1”对应-5V～-15V 电平，标准逻辑“0”对应+5V～+15V 电平，显然，两者间要进行通信必须经过信号电平的转换，目前常使用的电平转换电路为 MAX232，其引脚分布如图 5.3.10。

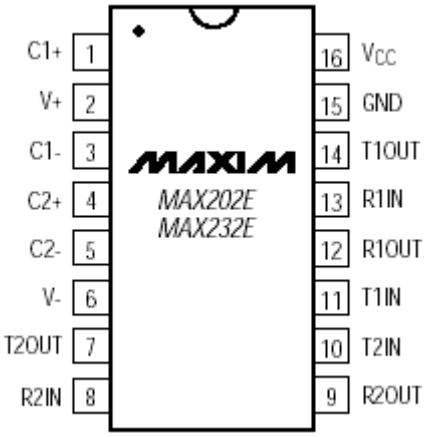


图 5.3.10 MAX232 引脚分布

关于 MAX232 更具体的内容可参考 MAX232 的用户手册。

图 5.3.11 为 MAX232 的常见应用电路图。

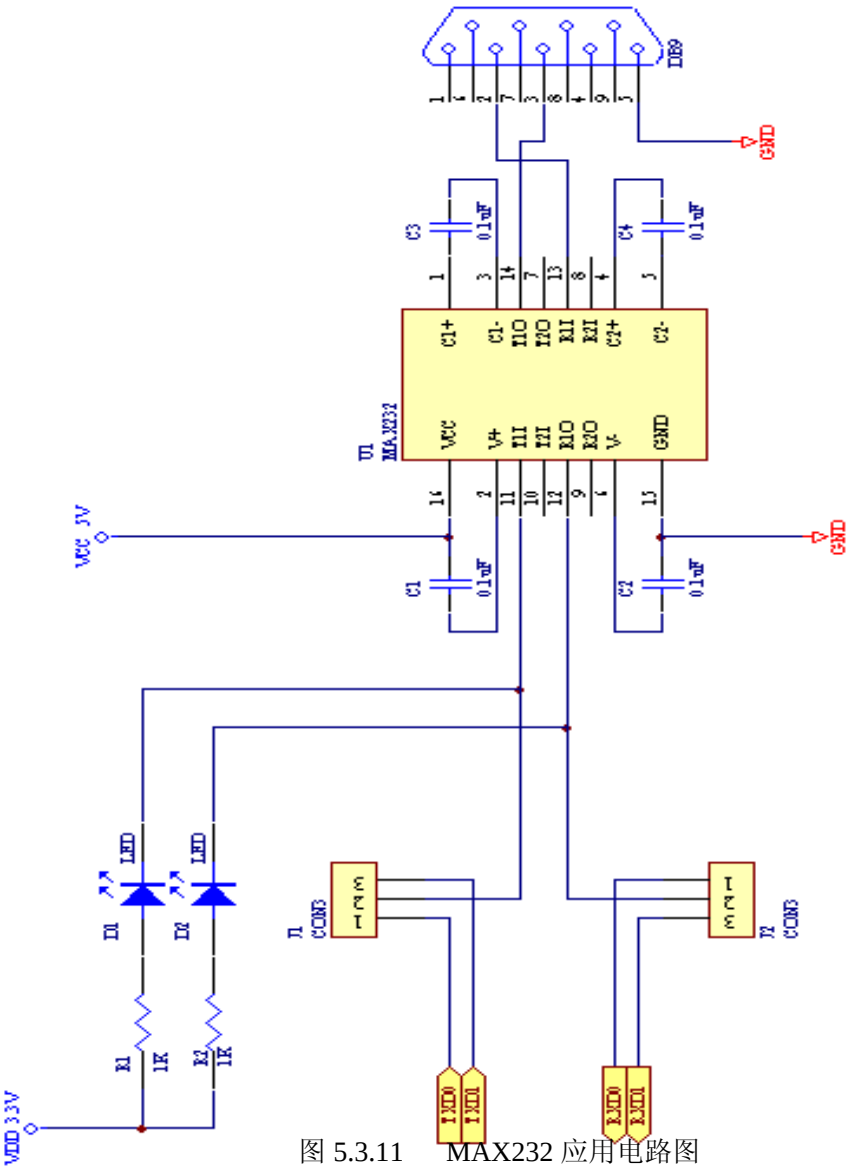


图 5.3.11　MAX232 应用电路图

图 5. 3. 11　MAX232 的常见应用电路图

为缩小电路板面积，系统只设计了一个 9 芯的 D 型插头，通过两个跳线选择 S3C3410B 的 UART0 或 UART1，同时设计数据发送与接收的状态指示 LED，当有数据通过串行口传输时，LED 闪烁，便于用户掌握其工作状态以及进行软、硬件的调试。

5.3.7　IIC 接口电路

IIC 总线是一种用于 IC 器件之间连接的二线制总线。它通过 SDA（串行数据线）及 SCL（串行时钟线）两线在连接到总线上的器件之间传送信息，并根据地址识别每个器件：不管是微控制器、存储器、LCD 驱动器还是键盘接口。带有 IIC 总线接口的器件可十分方便地用来将一个或多个微控制器及外围器件构成系统。尽管这种总线结构没有并行总线那样大的吞吐能力，但由于连接线和连接引脚少，因此其构成的系统价格低，器件间总线简单，结构紧凑，而且在总线上增加器件不影响系统的正常工作，系统修改和可扩展性好。即使有不同时钟速度的器件连接到总线上，也能很方便地确定总线的时钟，因此在嵌入式系统中得到了广泛的应用。

S3C4510B 内含一个 IIC 总线主控器，可方便的与各种带有 IIC 接口的器件相连。在该系统中，外扩一片 AT24C01 作为 IIC 存储器。AT24C01 提供 128 字节的 EEPROM 存储空间，可用于存放少量在系统掉电时需要保存的数据。

AT24C01 引脚分布及信号描述和应用电路如图 5.3.12、图 5.3.13 所示：

引 脚	功 能
NC	未连接
SDA	串行数据
SCL	串行时钟
TEST	测试输入（GND或VCC）

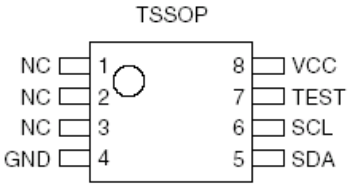


图 5.3.12

AT24C01 引脚分布及信号描述

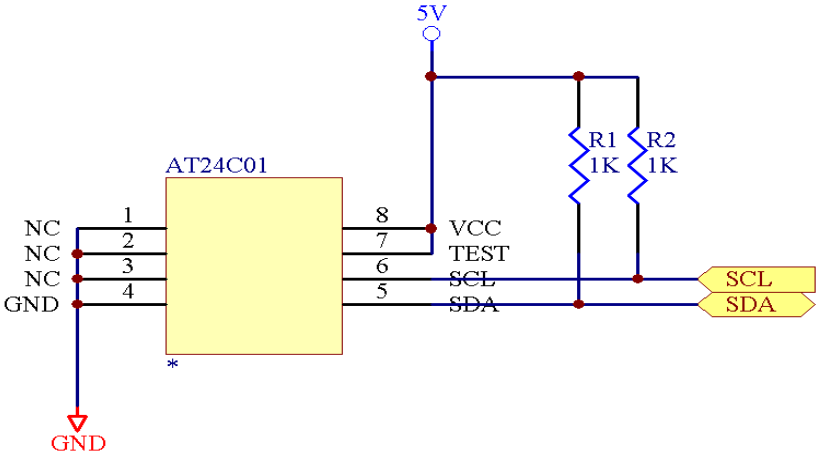


图 5.3.13　AT24C01 应用电路

5.3.8　JTAG 接口电路

JTAG(Joint Test Action Group，联合测试行动小组)是一种国际标准测试协议，主要用于芯片内部测试及对系统进行仿真、调试，JTAG 技术是一种嵌入式调试技术，它在芯片内部封装了专门的测试电路 TAP（Test Access Port，测试访问口），通过专用的 JTAG 测试工具对内部节点进行测试。目前大多数比较复杂的器件都支持 JTAG 协议，如 ARM、DSP、FPGA 器件等。标准的 JTAG 接口是 4 线：TMS、TCK、TDI、TDO，分别为测试模式选择、测试时钟、测试数据输入和测试数据输出。

JTAG 测试允许多个器件通过 JTAG 接口串联在一起，形成一个 JTAG 链，能实现对各个器件分别测试。JTAG 接口还常用于实现 ISP（In-System Programmable 在系统编程）功能，如对 FLASH 器件进行编程等。

通过 JTAG 接口，可对芯片内部的所有部件进行访问，因而是开发调试嵌入式系统的一种简洁高效的手段。目前 JTAG 接口的连接有两种标准，即 14 针接口和 20 针接口，其定义分别如下所示。

14 针 JTAG 接口定义：

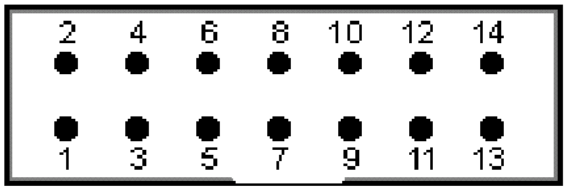


表 5-3-4　14 针 JTAG 接口定义

引　脚	名　称	描　　述
1、13	VCC	接电源
2、4、6、8、10、14	GND	接地
3	nTRST	测试系统复位信号
5	TDI	测试数据串行输入
7	TMS	测试模式选择
9	TCK	测试时钟
11	TDO	测试数据串行输出
12	NC	未连接

20 针 JTAG 接口定义：

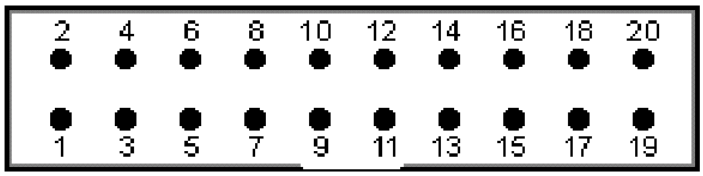


表 5-3-5　20 针 JTAG 接口定义

引　脚	名　称	描　　述
1	VTref	目标板参考电压，接电源
2	VCC	接电源
3	nTRST	测试系统复位信号
4、6、8、10、12、14、16、18、20	GND	接地
5	TDI	测试数据串行输入
7	TMS	测试模式选择
9	TCK	测试时钟
11	RTCK	测试时钟返回信号
13	TDO	测试数据串行输出
15	nRESET	目标系统复位信号
17、19	NC	未连接

5.3.9　10M/100M 以太网接口电路

作为一款优秀的网络控制器，基于 S3C4510B 的系统若没有以太网接口，其应用价值就会大打折扣，因此，就整个系统而言，以太网接口电路应是必不可少的，但同时也是相对较复杂的。从硬件的角度看，以太网接口电路主要由 MAC 控制器和物理层接口（Physical Layer，PHY）两大部分构成，目前常见的以太网接口芯片，如 RTL8019、RTL8029、RTL8039、CS8900、DM9008 等，其内部结构也主要包含这两部分。

S3C4510B 内嵌一个以太网控制器，支持媒体独立接口（Media Independent Interface，MII）和带缓冲 DMA 接口（Buffered DMA Interface，BDI）。可在半双工或全双工模式下提供 10M/100Mbps

的以太网接入。在半双工模式下，控制器支持 CSMA/CD 协议，在全双工模式下支持 IEEE802.3 MAC 控制层协议。

因此，S3C4510B 内部实际上已包含了以太网 MAC 控制，但并未提供物理层接口，因此，需外接一片物理层芯片以提供以太网的接入通道。

常用的单口 10M/100Mbps 高速以太网物理层接口器件主要有 RTL8201、DM9161 等，均提供 MII 接口和传统 7 线制网络接口，可方便的与 S3C4510B 接口。以太网物理层接口器件主要功能一般包括：物理编码子层、物理媒体附件、双绞线物理媒体子层、10BASE-TX 编码/解码器和双绞线媒体访问单元等。

在该系统中，使用 RTL8201 作为以太网的物理层接口。图 5.3.14 为 RTL8201 的引脚分布，表 6-3-6 相关引脚功能描述，表中仅列出芯片在 100Mbps MII 接口方式下的引脚定义，当工作于 7 线制网络接口方式，部分引脚定义不同。更具体的内容和使用方法可参考 RTL8201 的用户手册。

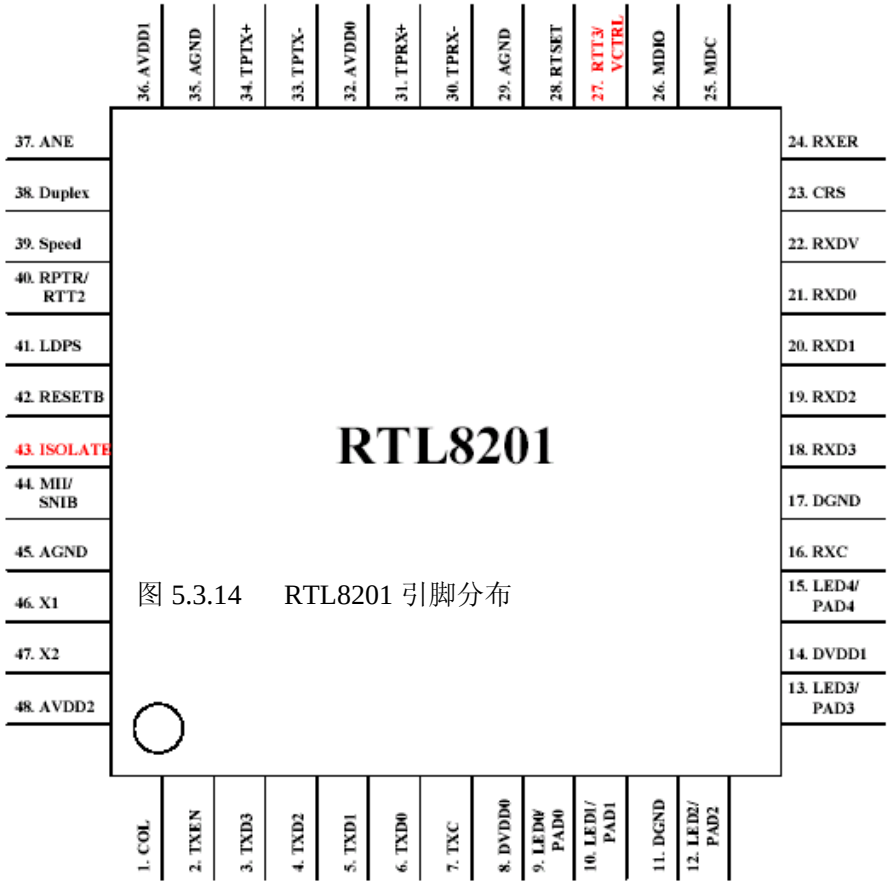


图 5.3.14　RTL8201 引脚分布

图 5. 3. 14　RTL8201 的引脚分布

表 5-3-6　RTL8201 引脚功能描述

信号	类型	引脚	功 能 描 述
TXC	O	7	**发送时钟**：该引脚提供连续时钟信号作为 TXD[3:0]和 TXEN 的时序参考。
TXEN	I	2	**发送使能**：该引脚指示目前 TXD[3:0]上的 4 位信号有效。
TXD[3:0]	I	3,4, 5,6	**发送数据**：当 TXEN 有效时，MAC 随 TXC 同步送出 TXD[3:0]。
RXC	O	16	**接收时钟**：该引脚提供连续时钟信号作为 RXD[3:0]和 RXDV 的时序参考，在 100Mbps 时，RXC 的频率为 25MHz，10Mbps 时为 2.5MHz。
COL	O	1	**冲突检测**：当检测到冲突时，COL 置为高电平。

CRS	I/O	23	**载波侦听**：在非 IDEL 状态时，该引脚置为高电平。
RXDV	O	22	**接收数据有效**：当接收 RXD[3:0]上的数据时，该引脚置高电平，接收结束时置低电平。该信号在 RXC 的上升沿有效。
RXD[3:0]	O	18,19, 20,21	**接收数据**：该引脚随 RXC 同步将数据从 PHY 传送给 MAC。
RXER	O	24	**接收错误**：当接收数据发生错误时，该引脚置为高电平。
MDC	I	25	**站管理时钟信号**：该引脚为 MDIO 提供同步时钟信号，但可能与 TXC 和 RXC 时钟异步。该时钟信号最高可达 2.5MHz。
MDIO	I/O	26	**站数据输入输出**：该引脚提供用于站管理的双向数据信息。
X2	O	47	**25MHz 晶振输出**：该引脚提供 25MHz 晶振输出。当 X1 外接 25MHz 振荡器时，该引脚必须悬空。
X1	I	46	**25MHz 晶振输入**：该引脚提供 25MHz 晶振输入。当外接 25MHz 振荡器时，该引脚作为输入。
TPTX＋ TPTX－	O O	34，33	**发送输出**。
RTSET	I	28	发送差分电阻连接：该引脚应通过一 2.0K 的电阻下拉。
TPRX＋ TPRX－	I I	31，30	**接收输入**。
ISOLATE	I	43	该引脚置高将 RTL8201 与 MAC 和 MDC/MDIO 管理接口隔离。在该模式下，功耗最小。
RPTR/ RTT2	I	40	该引脚置高将 RTL8201 设为转发器工作模式。在测试模式下，该引脚重定义为 RTT2。
SPEED	I	39	该引脚置高 RTL8201 以 100Mbps 的速率工作。
DUPLEX	I	38	该引脚置高使能全双工模式。
ANE	I	37	该引脚置高使能自动协议模式，置低为强制模式。
LDPS	I	41	该引脚置高 RTL8201 进入未连接省电（LDPS）模式。
MII/SNIB	I	44	该引脚置高 RTL8201 进入 MII 模式工作。
LED0/PAD0	O	9	连接 LED 显示。
LED1/PAD1	O	10	全双工 LED 显示。
LED1/PAD2	O	12	10M 连接/应答 LED 显示。
LED1/PAD3	O	13	100M 连接/应答 LED 显示。
LED1/PAD4	O	15	冲突 LED 显示。
RTT3/CTRL	O	27	目前用作测试，可作为将来的功能扩展。
RESETB	I	42	芯片复位引脚，低电平复位。
AVDD0/ AVDD1	P	32，36	模拟电源：为片内模拟电路提供 3.3V 电源，应接去耦合电容。
AVDD2	P	48	为片内 PLL 电路提供 3.3V 电源，应接去耦合电容并用 100ohm@100MHz 磁珠接到模拟地。
AGND	P	29， 35，45	模拟地：接地。
DVDD0/ DVDD1	P	8，14	数字电源：为片内数字电路提供 3.3V 电源。
DGND	P	11，17	数字地：接地

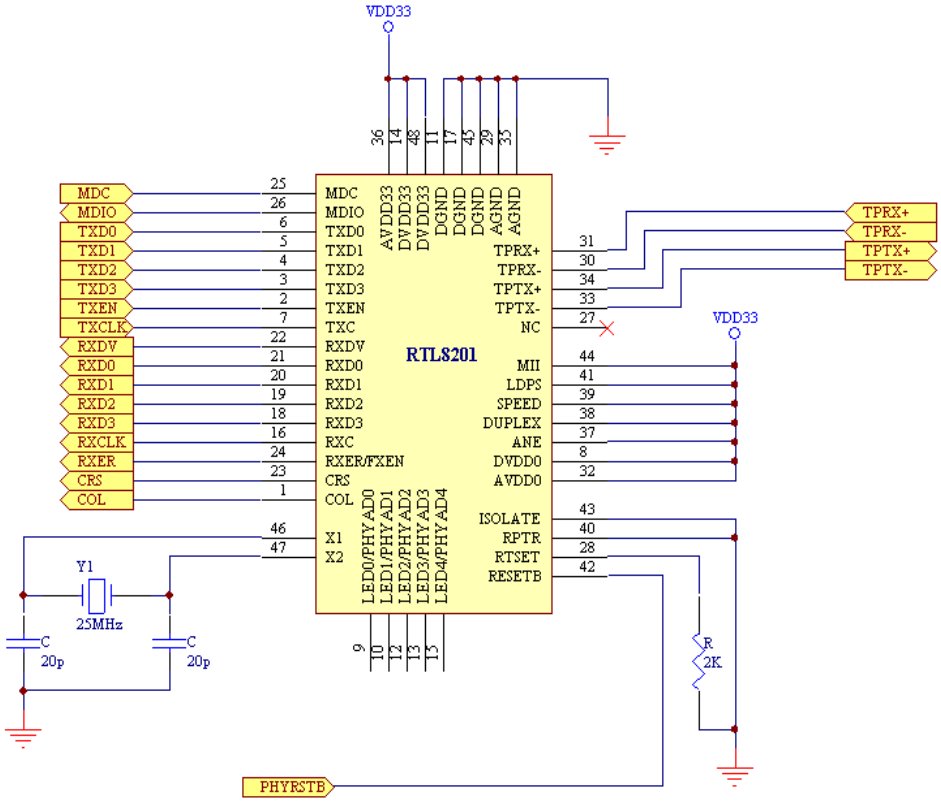


图 5. 3. 15　RTL8201 应用电路图

由于 S3C4510B 片内已有带 MII 接口的 MAC 控制器，而 RTL8201 也提供了 MII 接口，各种信号的定义也很明确，因此 RTL8201 与 S3C4510B 地连接比较简单。图 5.3.15 为 RTL8201 地实际应用电路图。

S3C4510B 的 MAC 控制器可通过 MDC/MDIO 管理接口控制多达 31 个 RTL8201，每个 RTL8201 应有不同的 PHY 地址（可从 00001B 到 11111B）。当系统复位时，RTL8201 锁存引脚 9，10，12，13，15 的初始状态作为与 S3C4510B 管理接口通信的 PHY 地址，但该地址不能设为 00000B，否则 RTL8201 进入掉电模式

为减少芯片的引脚数，RTL8201 的 LED 引脚同时复用为 PHY 的地址引脚，因此引脚 9，10，12，13，15 不能直接接到电源或地。图 5.3.16 为引脚 9，10，12，13，15 地连接方法，此时 RTL8201 的 PHY 地址为 00001B。引脚通过 5.1K 的电阻上拉或下拉，决定 RTL8201 的 PHY 地址，在正常工作时，LED 显示 RTL8201 的工作状态，当不需要 LED 状态显示时，LED＋510 欧姆的电阻可去掉。

在图 5.3.15 中，信号地发送和接收端应通过网络隔离变压器和 RJ45 接口接入传输媒体，其实际应用电路见图 5.3.17。

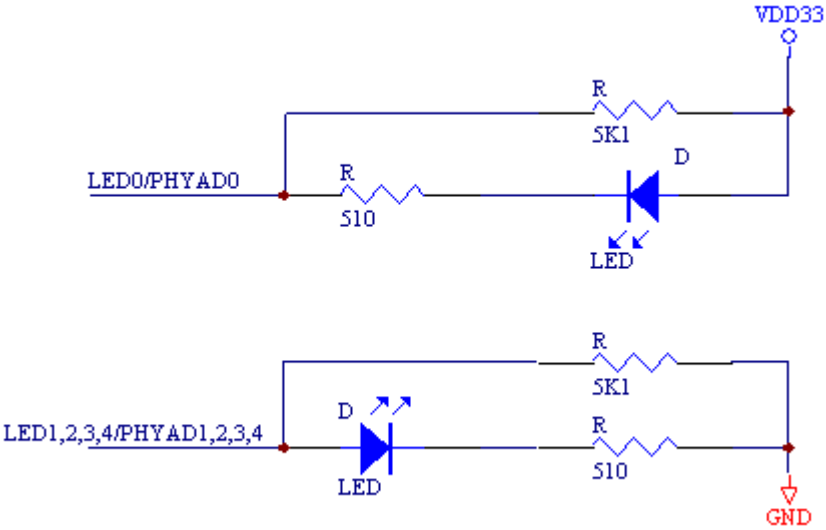


图 5.3.16　RTL8201 的 LED 与 PHY 地址配置

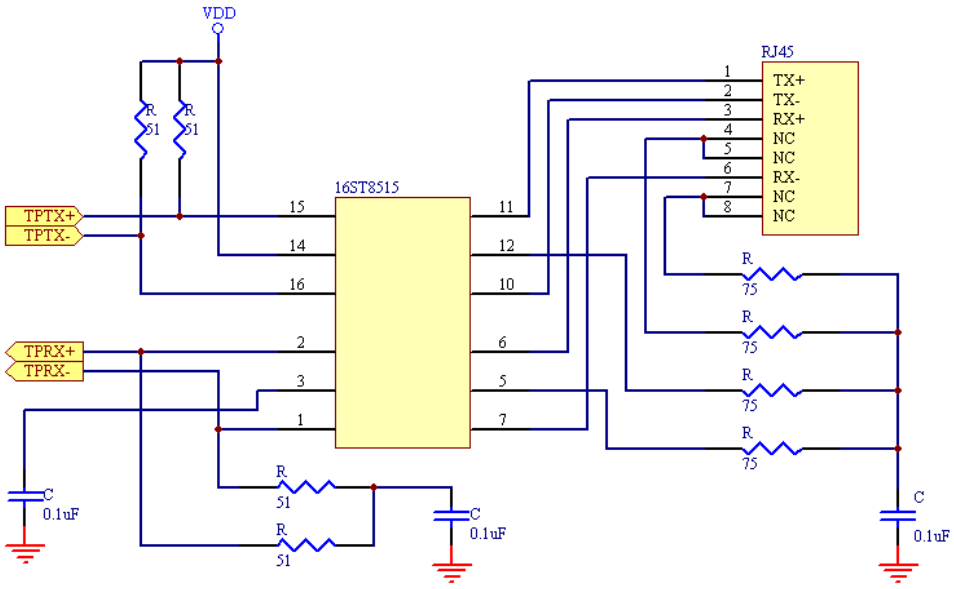


图 5.3.17　RTL8201 与网络隔离变压器及 RJ45 的连接图

5.3.10　通用 I/O 接口电路

S3C4510B 提供了 18 个可编程的 I/O 端口，用户可将每个端口配置为输入模式、输出模式或特殊功能模式，由片内的特殊功能寄存器控制。在该系统的设计中，P3～P0 外接 4 只 LED 显示器，用作程序运行状态的显示或其他输出功能，P7～P4 外接跳线选择高、低电平用作状态输入，以控制程序流程或其他输入功能，其应用电路如图图 5.3.18。

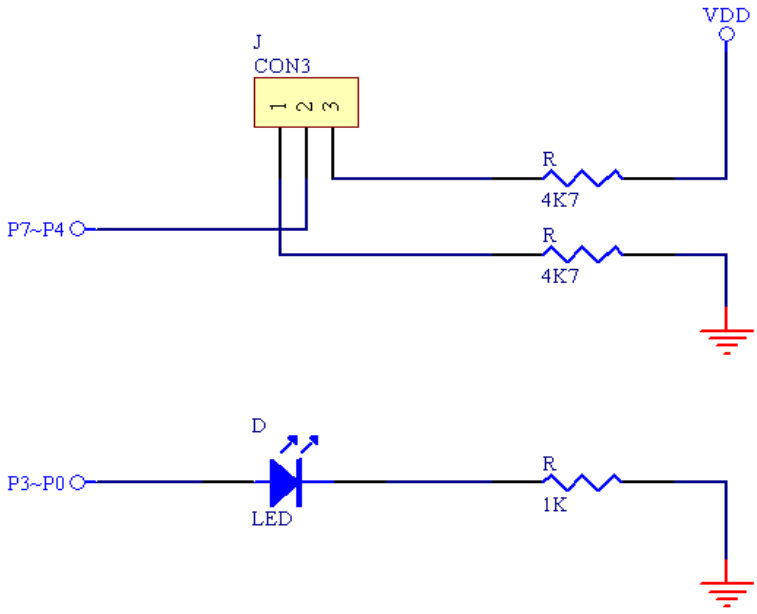


图 5.3.18

通用 I/O 口用作输入输出

5.4 硬件系统的调试

通过 5.3 节对 S3C4510B 应用系统设计方法的阅读，具有一定系统开发经验的读者就可以设计开发自己的特定应用系统，上节所介绍的内容实际上可理解为一个基于 S3C4510B 的最小系统，读者可根据自己的实际需要做适当的增减。

当系统设计制作完成时，必须经过仔细的调试，才能保证系统按照设计意图正常工作。尽管系统的调试与个人对电路工作原理的理解和实际的电路调试经验有很大的关系，但一定的调试方法也是必不可少的。掌握正确的调试方法可使调试工作变得容易，大大缩短系统的开发时间，反之，可能会使整个系统的开发前功尽弃，以失败告终。

本节以单元电路为单位，并结合笔者自身在系统调试时所遇到的一些具有代表性的问题，循序渐进的介绍整个系统的调试过程。在此，笔者建议：当用户的印制电路板制作完毕后，不要急于焊接元器件，请首先对照原理图仔细检查印制电路板的连线，确保无误后方可焊接。同时，尽可能的以各单元电路为单位，一个个焊接调试，以便在调试过程中遇到困难时缩小故障范围，在系统上电后，应先检查电路工作有无异常，芯片在工作时有一定的发热是正常的，但如果有芯片特别发烫，则一定有故障存在，需断电检查确认无误后方可继续通电调试。

调试工具需要示波器、万用表等，同时需要 ARM 调试开发软件 ADS 或 SDT 及相应的仿真器，本系统在调试时使用 ADS1.2 及由北京微芯力科技有限公司开发的 ARM JTAG 仿真器。关于 ADS 的使用方法在以后的章节有详细的叙述。

5.4.1 电源、晶振及复位电路

电源电路、晶振电路和复位电路相对比较简单，按图 5.3.1、图 5.3.2 和图 5.3.3 连接后应该就可以正常工作，此时电源电路的输出因为 DC 3.3V。

用示波器观测，有源晶振的输出应为 10MHz；

复位电路的 RESET 端在未按按钮时输出应为高电平（3.3V），按下按钮后变为低电平，按钮松开后应恢复到高电平。

电源电路、晶振电路和复位电路是整个系统正常工作的基础，应首先保证他们的正常工作。

5.4.2 S3C4510B 及 JTAG 接口电路

在保证电源电路、晶振电路和复位电路正常工作的前提下，可通过 JTAG 接口调试 S3C4510B，在系统上电前，首先应检测 JTAG 接口的 TMS、TCK、TDI、TDO 信号是否已与 S3C4510B 的对应引脚相连，其次应检测 S3C4510B 的 nEWAIT 引脚（Pin71）是否已上拉，ExtMREQ 引脚（Pin108）是否已下拉，对这两只引脚的处理应注意，作者遇到多起 S3C4510B 不能正常工作或无法与 JTAG 接口通信，均与没有正确处理这两只引脚有关。

给系统上电后，可通过示波器查看 S3C4510B 对应引脚的输出波形，判断是否已正常工作，若 S3C4510B 已正常工作，在使能片内 PLL 电路的情况下，SDCLK/MCLKO 引脚（Pin77）应输出频率为 50MHz 的波形，同时，MDC 引脚（Pin50）和其他一些引脚也应有波形输出。

在保证 S3C4510B 已正常工作的情况下，可使用 ADS 或 SDT 通过 JTAG 接口对片内的部件进行访问和控制。

在此，首先通过对片内控制通用 I/O 口的特殊功能寄存器的操作，来点亮连接在 P3～P0 口上的 4 只 LED，用以验证 ADS 或 SDT 调试环境是否已正确设置，以及与 JTAG 接口的连接是否正常。

ADS 和 SDT 均为 ARM 公司为方便用户在 ARM 芯片上进行应用开发而推出的一整套集成开发工具，其中，ADS 为 SDT 的升级版本。该系统的调试以 ADS 为例，同时也适合于 SDT 开发环境。图 5.4.1 为调试系统的硬件连接。

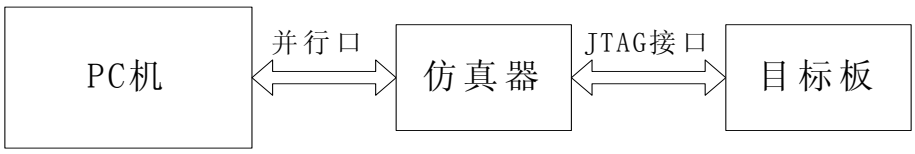


图 5.4.1 调试系统的硬件连接

按图 5.4.1 连接好硬件后，打开 AXD Debugger，建立与目标板（待调试的系统板）的连接，AXD Debugger 有软件仿真方式和带目标系统的调试方式，此时应工作在带目标系统的调试方式。

选择菜单 System Views→Command Line Interface 功能，该选项为 AXD Debugger 的一个命令行窗口，可在该窗口内输入各种调试命令，使用非常方便。在命令行窗口输入：

```
>setmem  0x3FF5000, 0xFFFF, 32
>setmem  0x3FF5008, 0xFFFF, 32
```

setmem 命令用于对特定的地址设置特定的值，待设定的值可以是 8 位、16 位或 32 位，在此，对通用 I/O 口的模式寄存器和数据寄存器设置相应的值，点亮 LED。

S3C4510B 在复位后，特殊功能寄存器的基地址为 0x3FF0000, 由表 5-2-3 可知，I/O 口的模式寄存器偏移地址为 0x5000, 因此，I/O 口的模式寄存器的物理地址为 0x3FF5000，设定该寄存器的值为 0xFFFF，将 I/O 口置为输出方式。I/O 口的数据寄存器的物理地址为 0x3FF5008，设定该寄存器的值为 0xFFFF，将 I/O 口的输出置为高电平。

在执行完以上两条命令后，连接在通用 I/O 口的 4 只 LED 应被点亮，表示调试系统的软、硬件连接完好，可进行下一步的调试工作，否则，应重新检查调试系统。

用户若使用 SDT 作为调试工具，操作方法类似。按图 5.4.1 连接好硬件后，打开 ARM Debugger for Windows，建立与目标板（待调试的系统板）的连接，选择菜单 View→Command 功能，即可显示命令行窗口，在命令行窗口输入：

```
Debug: let 0x3FF5000 = 0xFFFF
Debug: let 0x3FF5008 = 0xFFFF
```

执行完以上两条命令后，连接在通用 I/O 口的 4 只 LED 应被点亮。

关于通用 I/O 口更具体的工作原理和使用方法，可参考 S3C4510B 用户手册。

用户系统若能正常完成上述操作并成功点亮连接在 P3～P0 口上的 LED 显示器，则表明 S3C4510B 已在正常工作，且调试环境也已正确建立，以后的调试工作就相对简单。笔者曾遇到多个用户系统

因为不能完成这步工作，使开发者失去信心而最终放弃。

5.4.3 SDRAM 接口电路的调试

在系统的两类存储器中，SDRAM 相对于 FLASH 存储器控制信号较多，似乎调试应该困难一些，但由于 SDRAM 的所有刷新及控制信号均由 S3C4510B 片内的专门部件控制，无需用户干预，在 S3C4510B 正常工作的前提下，只要连线无误，SDRAM 就应能正常工作，反之，Flash 存储器的编程、擦除操作均需要用户编程控制，且程序还应在 SDRAM 中运行，因此，应先调试好 SDRAM 存储器系统，再进行 Flash 存储器系统的调试。

在进行存储器系统调试之前，用户必须深入了解 S3C4510B 系统管理器关于存储器映射的工作原理。

基于 S3C4510B 系统的最大可寻址空间为 64MB，采用统一编址的方式，将系统的 SDRAM、SRAM、ROM、Flash、外部 I/O 以及片内的特殊功能寄存器和 8K 一体化 SRAM 均映射到该地址空间。为便于使用与管理，S3C4510B 又将 64MB 的地址空间分为若干个组，分别由相应的特殊功能寄存器进行控制：

— ROM/SRAM/Flash 组 0～ROM/SRAM/Flash 组 5，用于配置 ROM、SRAM 或 Flash，分别由特殊功能寄存器 ROMCON0～ROMCON5 控制；

— DRAM/SDRAM 组 0～DRAM/SDRAM 组 3 用于配置 DRAM 或 SDRAM，分别由特殊功能寄存器 DRAMCON0～DRAMCON3 控制；

— 外部 I/O 组 0～外部 I/O 组 3 用于配置系统的其他外扩接口器件，由特殊功能寄存器 REFEXTCON 控制；

— 特殊功能寄存器组用于配置 S3C4510B 片内特殊功能寄存器的基地址以及片内的 8K 一体化 SRAM，由特殊功能寄存器 SYSCFG 控制；

在该系统中，使用了 Flash 存储器和 SDRAM，分别配置在 ROM/SRAM/FLASH 组 0 和 DRAM/SDRAM 组 0，暂未使用外扩接口器件。

参照表 5-2-4 和 5.2.4 节对应特殊功能寄存器的相关描述可知，当系统复位时，只有 ROM/SRAM/FLASH 组 0 被映射到地址空间为 0x0000,0000～0x0200,0000 的位置，特殊功能寄存器的基地址被映射到 0x03FF,0000,片内 8K 一体化 SRAM 的起始地址被映射到 0x03FE,0000,它们是可访问的，而其他的存储器组均未被映射，是不可访问的。

因此，要调试 SDRAM 存储器系统，首先应配置相关的特殊功能寄存器，使系统中的 SDRAM 能被访问。表 5-4-1 为针对该系统的与系统管理器相关的特殊功能寄存器的配置，以下详细说明该系统所使用的相关特殊功能寄存器的配置方法。

表 5-4-1 系统管理器相关特殊功能寄存器的配置

寄存器	偏移量	配置值
SYSCFG	0x0000	0xE7FF,FF82
EXTDBWTH	0x3010	0x0000,3000
ROMCON0	0x3014	0x0200,0060
DRAMCON0	0x302C	0x1401,0380
REFEXTCON	0x303C	0xCE33,83FD

SYSCFG = 0xE7FF,FF82；其含义为：

为 4 个 DRAM 组选择 SDRAM；特殊功能寄存器组的基地址为 0x03FF,0000；片内 SRAM 基地址为 0x03FE,0000；4KB 配置为 SRAM，另外 4KB 配置为 Cache；使能 Cache 操作。

EXTDBWTH = 0x0000,3000；其含义为：

4 个外部 I/O 组禁用；DRAM/SDRAM 组 0 配置为 32 位数据宽度，其余的 DRAM/SDRAM 组禁用；ROM/SRAM/FLASH 组 0 由 BOSIZE[1:0]的状态配置，其余 ROM/SRAM/FLASH 组禁用。

ROMCON0 = 0x0200,0060；其含义为：

该系统共有 2MB 的 FLASH 存储器，映射到地址空间的 0x0000,0000～（0x0020,0000-1）处。

DRAMCON0 = 0x1401,0380；其含义为：

该系统共有 16MB 的 SDRAM，映射到地址空间的 0x0040,0000～（0x0140,0000-1）处。

用户也可将 Flash 存储器和 SDRAM 映射到地址空间的其他位置，但注意组与组之间的地址不要发生重叠。

DRAMCON0 = 0xCE33,83FD；其含义为：

配置 SDRAM 的刷新计数值，刷新时间、刷新使能等。

系统管理器对应的其他特殊功能寄存器使用其复位值，用户也可根据自身系统的特定情况，对相关特殊功能寄存器进行配置。

在 C:\下建立文本文件 memmap.txt，其内容为：

```
setmem 0x3FF0000,0xE7FFFF82,32
setmem 0x3FF3010,0x00003000,32
setmem 0x3FF3014,0x02000060,32
setmem 0x3FF302C,0x14010380,32
setmem 0x3FF303C,0xCE3383FD,32
```

打开 AXD Debugger 的命令行窗口，执行 obey 命令，配置对应的控制寄存器：

```
>obey C:\memmap.txt
```

此时，文本文件 memmap.txt 中的几条控制寄存器配置命令已经执行完毕，Flash 存储器和 SDRAM 已分别映射到地址空间的 0x0000,0000～（0x0020,0000-1）和 0x0040,0000～（0x0140,0000-1）处。

选择菜单 Processor Views→Memory 选项，出现存储器窗口，在存储器起始地址栏输入 SDRAM 的映射起始地址：0x0040,0000,数据区应显示 SDRAM 中的内容，此时所显示的内容为一些随机数。双击其中的任一数据，输入新的值，如输入 0xAA,若对应的存储单元能正确显示刚才输入的数据，则表明 SDRAM 存储器已能正常工作。

在连续的 4 个字节输入 0xAA，然后再输入 0x55，检测 32 位数据是否正确传输，若其中的某一位或几位数据出现错误，则多半是由于对应的数据线不通或连接错误所引起的。

在 SDRAM 能正确访问后，用户就可以将自己编写的各种应用程序，编译并下载到 SDRAM 中运行。

若使用 SDT 调试环境，调试过程与上述步骤相似，简述如下：

在 C:\下建立文本文件 memmap.txt，其内容为：

```
let 0x3FF0000 = 0xE7FFFF82
let 0x3FF3010 = 0x00003000
let 0x3FF3014 = 0x02000060
let 0x3FF302C = 0x14010380
let 0x3FF303C = 0xCE3383FD
```

打开 ARM Debugger for Windows 的命令行窗口（View→Command），执行 obey 命令：

```
>obey C:\memmap.txt
```

此时，Flash 存储器和 SDRAM 已分别映射到地址空间的 0x0000,0000～（0x0020,0000-1）和 0x0040,0000～（0x0140,0000-1）处。

选择菜单 View→Memory 选项，出现存储器的起始地址输入窗口，在此输入 SDRAM 的映射起始地址：0x0040,0000,数据区应显示 SDRAM 中的内容，此时所显示的内容为一些随机数。双击其中的任一数据，输入新的值，如输入 0xAA,若对应的存储单元能正确显示刚才输入的数据，则表明 SDRAM 存储器已能正常工作。

5.4.4 Flash 接口电路的调试

Flash 存储器的调试主要包括 Flash 存储器的编程（烧写）和擦除，与一般的存储器件不同，用户只需对 Flash 存储器发出相应的命令序列，Flash 存储器通过内部嵌入的算法即可完成对芯片的操作，由于不同厂商的 Flash 存储器在操作命令上可能会有一些细微的差别，Flash 存储器的编程与擦除工具一般不具有通用性，这也是为什么 Flash 接口电路相对较难调试的原因之一，因此，应在理解 Flash 存储器编程和擦除的工作原理的情况下，根据不同型号器件对应的命令集，编写相应的程序对其进行操作。

打开 AXD Debugger 的命令行窗口，执行 obey 命令：

```
>obey C:\memmap.txt
```

此时，2MB 的 Flash 存储器映射到地址空间的 0x0000,0000～0x001F,FFFF 处，选择菜单 Processor Views→Memory 选项，出现存储器窗口，在存储器起始地址栏输入 Flash 存储器的映射起始地址：0x0,数据区应显示 Flash 存储器中的内容，若 Flash 存储器为空，所显示的内容应全为 0xFF,否则应为已有的编程数据。双击其中的任一数据，输入新的值，对应存储单元的内容应不能被修改，此时可初步认定 Flash 存储器已能被访问，但是否能对其进行正确的编程与擦除操作，还需要编程验证，通过程序对 Flash 存储器进行编程和擦除操作，放在下一章的内容里说明。

若使用 SDT 调试环境，调试过程与上述步骤相似。

5.4.5 10M/100M 以太网接口电路

以太网接口电路主要由 MAC 控制器和物理层接口（Physical Layer，PHY）两大部分构成，而 MAC 控制器在 S3C4510B 片内,需要用户作硬件调试的只是外接的物理层接口 RTL8201。由于 MAC 控制器的工作原理相对复杂，相应的特殊功能寄存器也比较多，在此不作详述，对此有兴趣的读者可参考已移植到 S3C4510B 的 uClinux 内核代码中对 MAC 控制器的驱动部分。

RTL8201 和 S3C4510B 均有 MII 接口，对应引脚及功能定义明确，只要正确连接，一般都能正常工作。当 RTL8201 正常工作在 100Mbps 状态时，其发送时钟引脚（Pin7）、接收时钟引脚（Pin16）均应有波形输出，同时，对应的 LED 指示灯也能正确指示芯片的工作状态。

5.5 印刷电路板的设计注意事项

在本章结束之前，对该系统的印刷电路板（PCB）设计中应注意的事项作一个简要的说明。

在系统中，S3C4510B 的片内工作频率为 50MHz，其以太网接口电路的工作速率更高达 100MHz 以上，因此，在印刷电路板的设计过程中，应该遵循一些高频电路的设计基本原则，否则会使系统工作不稳定甚至不能正常工作。印刷电路板的设计人员应注意以下几个方面：

— 注意电源的质量与分配。

— 同类型信号线应该成组、平行分布。

5.5.1 电源质量与分配

在设计印刷电路板时，能给各个单元电路提供高质量的电源，就会使系统的稳定性大幅度的提高。但如何能提高电源的质量，常用的手段有以下几个：

1、 电源滤波

为提高系统的电源质量，消除低频噪声对系统的影响，一般应在电源进入印刷电路板的位置和靠近各器件的电源引脚处加上滤波器，以消除电源的噪声，常用的方法是在这些位置加上几十到几百微法的电容。

同时，在系统中除了要注意低频噪声的影响，还要注意元器件工作时产生的高频噪声，一般的方法是在器件的电源和地之间加上 0.1uF 左右地电容，可以很好地滤出高频噪声的影响。

2、 电源分配

实际的工程应用和理论都证实，电源的分配对系统的稳定性有很大的影响，因此，在设计印刷电路板时，要注意电源的分配问题。

在印刷电路板上，电源的供给一般采用电源总线（双面板）或电源层（多层板）的方式。电源总线由两条或多条较宽的线组成，由于受到电路板面积的限制，一般不可能布得过宽，因此存在较大的直流电阻，但在双面板得设计中也只好采用这种方式了，只是在布线的过程中，应尽量注意这个问题。

在多层板的设计中，一般使用电源层的方式给系统供电。该方式专门拿出一层作为电源层而不再在其上布信号线。由于电源层遍及电路板的全面积，因此直流电阻非常的小，采用这种方式可有效的降低噪声，提高系统的稳定性。

5.5.2 同类型信号线的分布

在各种微处理器的输入输出信号中，总有相当一部分是相同类型的，例如数据线、地址线。对这些相同类型的信号线应该成组、平行分布，同时注意它们之间的长短差异不要太大，采用这种布线方式，不但可以减少干扰，增加系统的稳定性，还可以使布线变得简单，印刷电路板的外观更美观。

以本系统的印刷电路板设计为例，成组的信号线主要是数据线和地址线，可在元器件位置确定后，首先完成他们的布线，尽可能做到成组、平行分布，同时应尽可能的短。然后在进行各种控制信号的布线，最后处理电源和接地引脚。

5.6 本章小节

本章主要介绍 S3C4510B 的基本结构和工作原理，同时介绍了设计一个基于 S3C4510B 的最小硬件系统的详细步骤、实现细节以及硬件系统的调试方法等内容。

需要说明，本章中所使用的器件和电路等，可能不是最优的，但可以保证是能正常工作的，在系统开发的过程中，不同的人会碰到不同的问题，本章的内容只是对笔者系统开发过程的一个描述和简要的总结，希望能对读者的系统设计工作有一点帮助。

第 6 章 部件工作原理与编程示例

本章主要以 S3C4510B 的几个常用功能部件为编程对象，介绍基于 S3C4510B 的系统的程序设计与调试，同时简介 BootLoader 的基本原理和编程方法，通过对本章的阅读，可以使读者了解 S3C4510B 各功能部件的工作原理及基本编程方法。

本章的主要内容包括：

- 嵌入式系统应用程序设计的基本方法。
- S3C4510B 通用 I/O 口的工作原理与编程示例。
- S3C4510B 串行通信控制器的工作原理与编程示例。
- S3C4510B 中断控制器的工作原理与编程示例。
- S3C4510B 定时器的工作原理与编程示例。
- S3C4510B DMA 控制器的工作原理与编程示例。
- S3C4510B IIC 总线控制器的工作原理。
- S3C4510B 以太网控制器的工作原理。
- Flash 存储器的工作原理与编程示例。
- BootLoader 简介

6.1 嵌入式系统的程序设计方法

一般说来，对于一个完整的嵌入式应用系统的开发，硬件的设计与调试工作仅占整个工作量的一半，应用系统的程序设计也是嵌入式系统设计一个非常重要的方面，程序的质量直接影响整个系统功能的实现，好的程序设计可以克服系统硬件设计的不足，提高应用系统的性能，反之，会使整个应用系统无法正常工作。

本章从应用的角度出发，以 S3C4510B 的各个功能模块为编程对象，介绍一些实用的程序段，读者既可按自己的需要修改，也可吸收其设计思想和方法，以便设计出适合于自己特定应用系统的实用程序。同时，由于 ARM 体系结构的一致性，尽管以下的应用程序段是针对特定硬件平台开发的，其编程思路同样适合于其他类型的 ARM 微处理器。

不同于基于 PC 平台的程序开发，嵌入式系统的程序设计具有其自身的特点，程序设计的方法也会因系统或因人而异，但其程序设计还是有其共同的特点及规律的。在编写嵌入式系统应用程序时，可采取如下几个步骤：

（1） 明确所要解决的问题：根据问题的要求，将软件分成若干个相对独立的部分，并合理设计软件的总体结构。

（2） 合理配置系统资源：与基于 8 位或 16 位微控制器的系统相比较，基于 32 位微控制器的系统资源要丰富得多，但合理的资源配置可最大限度的发挥系统的硬件潜能，提高系统的性能。对于一个特定的系统来说，其系统资源，如 Flash 、EEPROM、SDRAM、中断控制等，都是有限的，应合理配置系统资源。

（3） 程序的设计、调试与优化：根据软件的总体结构编写程序，同时采用各种调试手段，找出程序的各种语法和逻辑错误，最后应使各功能程序模块化，缩短代码长度以节省存储空间并减少程序执行时间。

此外，由于嵌入式系统一般都应用在环境比较恶劣的场合，易受各种干扰，从而影响到系统的可靠性，因此，应用程序的抗干扰技术也是必须考虑的，这也是嵌入式系统应用程序不同于其他应

用程序的一个重要特点。

6.2 部件工作原理与编程示例

6.2.1 通用 I/O 口工作原理与编程示例

S3C4510B 提供了 18 个可编程的通用 I/O 端口，用户可将每个端口配置为输入模式、输出模式或特殊功能模式，由片内的特殊功能寄存器 IOPMOD 和 IOPCON 控制。

端口 0～端口 7 的工作模式仅由 IOPMOD 寄存器控制，但通过设置 IOPCON 寄存器，端口 8～端口 11 可用作外部中断请求 INTREQ0～INTREQ3 的输入，端口 12、端口 13 可用作外部 DMA 请求 XDREQ0、XDREQ1 的输入，端口 14、端口 15 可作为外部 DMA 请求的应答信号 XDACK0、XDACK1，端口 16 可作为定时器 0 的溢出 TOUT0，端口 17 可作为定时器 1 的溢出 TOUT1。

I/O 端口的功能模块如图 6.2.1 所示：

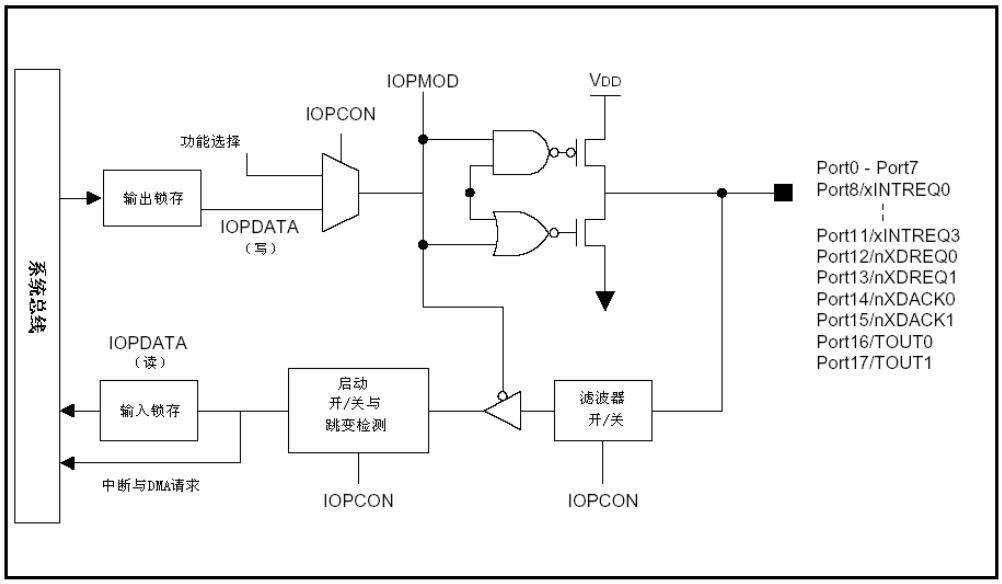


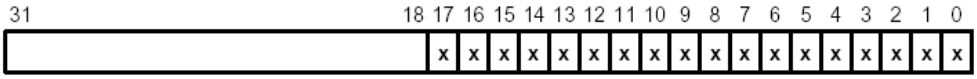
图 6.2.1 通用 I/O 口的功能模块

控制 I/O 口的特殊功能寄存器一共有 3 个：IOPMOD、IOPCON 和 IOPDATA，简要描述如下：

I/O 口模式寄存器（IOPMOD）：

I/O 口模式寄存器 IOPMOD 用于配置 P17～P0。

寄存器	偏移地址	操作	功能描述	复位值
IOPMOD	0x5000	读/写	I/O 口模式寄存器	0x0000,0000



[0]P0 口的 I/O 模式位

0=输入

1=输出

[1]P1 口的 I/O 模式位

0=输入

1=输出

[2]P2 口的 I/O 模式位

0=输入

1=输出

[3～17]P3～P17 口的 I/O 模式位

0=输入

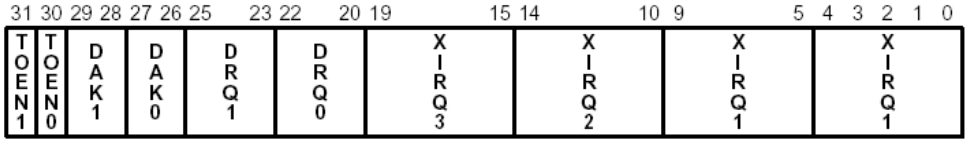
1=输出

I/O 口控制寄存器（IOPCON）：

I/O 口控制寄存器 IOPCON 用于配置端口 P8～P17 的特殊功能，当这些端口用作特殊功能（如外部中断请求、外部中断请求应答、外部 DMA 请求或应答、定时器溢出）时，其工作模式由 IOPCON 寄存器控制，而不再由 IOPMOD 寄存器。

对于特殊功能输入端口，S3C4510B 提供了一个滤波器用于检测特殊功能信号的输入，如果输入信号电平宽度等于三个系统时钟周期，该信号被认为是诸如外部中断请求或外部 DMA 请求等特殊功能信号。

寄存器	偏移地址	操作	功能描述	复位值
IOPCON	0x5004	读/写	I/O 口控制寄存器	0x0000,0000



[4：0]控制端口 8 的外部中断请求信号 0（xIRQ0）输入

[4]端口 8 用作外部中断请求信号 0

0 = 禁止 1 = 使能

[3] 0 = 低电平有效 1 = 高电平有效

[2] 0 = 滤波器关 1 = 滤波器开

[1：0] 00 = 电平检测 01 = 上升沿检测

10 = 下降沿检测 11 = 上升、下降沿均检测

[9：5]控制端口 9 的外部中断请求信号 1（xIRQ1）输入

使用方法同端口 8。

[14：10]控制端口 10 的外部中断请求信号 2（xIRQ2）输入

使用方法同端口 8。

[19：15]控制端口 11 的外部中断请求信号 3（xIRQ3）输入

使用方法同端口 8。

[22：20]控制端口 12 的外部 DMA 请求信号 0（DRQ0）输入

[22]端口 12 用作外部 DMA 请求信号 0（nXDREQ0）

0 = 禁止 1 = 使能

[21] 0 = 滤波器关 1 = 滤波器开

[20] 0 = 低电平有效 1 = 高电平有效

[25：23]控制端口 13 的外部 DMA 请求信号 1（DRQ1）输入

[25]端口 13 用作外部 DMA 请求信号 1（nXDREQ1）

0 = 禁止 1 = 使能

[24] 0 = 滤波器关 1 = 滤波器开

[23] 0 = 低电平有效 1 = 高电平有效

[27：26]控制端口 14 的外部 DMA 应答信号 0（DAK0）输出

[27]端口 14 用作外部 DMA 信号 0（nXDACK0）

0 = 禁止 1 = 使能

[26] 0 = 低电平有效 1 = 高电平有效

[29：28]控制端口 15 的外部 DMA 应答信号 1（DAK1）输出

[29]端口 15 用作外部 DMA 信号 1（nXDACK1）

0 = 禁止　　1 = 使能

[28] 0 = 低电平有效　1 = 高电平有效

[30]控制端口 16 作为定时器 0 溢出信号（TOEN0）

0 = 禁止　　1 = 使能

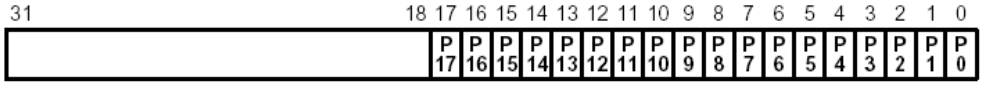
[31]控制端口 17 作为定时器 1 溢出信号（TOEN1）

0 = 禁止　　1 = 使能

I/O 口数据寄存器（IOPDATA）：

当配置为输入模式时，读取 I/O 口数据寄存器 IOPDATA 的每一位对应输入状态，当配置为输出模式时，写每一位对应输出状态。位[17：0]对应于 18 个 I/O 引脚 P17～P0。

寄存器	偏移地址	操作	功能描述	复位值
IOPDATA	0x5008	读/写	I/O 口数据寄存器	未定义



[17：0]对应 I/O 口 P17～P0 的读/写值。I/O 口数据寄存器的值反映对应引脚的信号电平。

以上简述了 S3C4510B 的通用 I/O 口的基本工作原理，更详细的内容可参考 S3C4510B 的用户手册。

作为本章的第一个例子，将比较详细的描述建立项目、编写程序的过程，同时可参考第八章关于 ADS 集成编译调试环境的使用方法。

打开 CodeWarrior for ARM Developer Suite（或 ARM Project Manager），新建一个项目，并新建一个文件，名为 Init.s，具体内容如下：

```
; ************************************************************
; Institute of Automation， Chinese Academy of Sciences
; File Name：    Init.s
; Description：
; Author：       JuGuang.Lee
; Date：
; ************************************************************
        IMPORT      Main
        AREA    Init，CODE，READONLY
        ENTRY
        LDR R0, =0x3FF0000
        LDR R1, =0xE7FFFF80 ；配置 SYSCFG,片内 4K Cache,4K SRAM
        STR     R1, [R0]
        LDR SP, =0x3FE1000  ；SP 指向 4K SRAM 的尾地址，堆栈向下生成
        BL      Main
        B       .
        END
```

该段代码完成的功能为：

配置 SYSCFG 特殊功能寄存器，将 S3C4510B 片内的 8K 一体化的 SRAM 配置为 4K Cache，4K SRAM，并将用户堆栈设置在片内的 SRAM 中。

4K SRAM 的地址为 0x3FE,0000～(0x3FE,1000－1)，由于 S3C4510B 的堆栈由高地址向低地址生成，将 SP 初始化为 0x3FE,1000。

完成上述操作后，程序跳转到 Main 函数执行。

保存 Init.s，并添加到新建的项目。

再新建一个文件，名为 main.c，具体内容如下：

```
/*****************************************************************
* Institute of Automation,Chinese Academy of Sciences
* File Name:    main.c
* Description:
* Author:       JuGuang.Lee
* Date:
*****************************************************************/
#define  IOPMOD   (*(volatile unsigned *)0x03FF5000) //IO port mode register
#define  IOPDATA  (*(volatile unsigned *)0x03FF5008) //IO port data register
void Delay(unsigned int);
int Main()
{
      unsigned long  LED;
      IOPMOD=0xFFFFFFFF;    //将 IO 口置为输出模式
      IOPDATA=0x01;
      for(;;){
            LED=IOPDATA;
            LED=(LED<<1);
            IOPDATA=LED;
            Delay(10);
            if(!(IOPDATA&0x0F))
                IOPDATA=0x01;
            }
      return(0);
}
void Delay(unsigned int x)
{
      unsigned int i,j,k;
      for(i=0;i<=x;i++)
            for(j=0;j<0xff;j++)
                for(k=0;k<0xff;k++);
}
```

保存 main.c，并添加到新建的项目。此时可对该项目进行编译链接，生成可执行的映象文件。

可执行的映象文件主要用于程序的调试，一般在系统的 SDRAM 中运行，并不烧写入 Flash，因此，项目文件在链接时，注意程序的入口点应与系统中 SDRAM 的实际配置地址相对应。链接器默认程序的入口地址为 0x8000，该值应根据实际的 SDRAM 映射地址进行修改。

在编译链接项目文件时，将链接器程序的入口地址为 0x0040,0000。

打开 AXD Debugger（或 ARM Debugger for Windows）的命令行窗口，执行 obey 命令：

```
>obey C:\memmap.txt
```

系统中 SDRAM 被映射到 0x0040,0000～（0x0140,0000-1），从 0x0040,0000 处装入生成的可执行的映象文件，并将 PC 指针寄存器修改为 0x0040,0000，就可单步调试或运行生成的可执行的映象文件。

该程序的运行效果为接在 P0~P3 口的 LED 显示器轮流被点亮。

6.2.2 串行通讯工作原理与编程示例

串行通讯是微计算机之间一种常见的近距离通讯手段，因使用方便、编程简单而广泛使用，几乎所有的微控制器、PC 都提供串行通讯接口。

S3C4510B 的 UART 单元提供两个独立的异步串行 I/O 口（Asynchronous Serial I/O，SIO），每

个通讯口均可工作在中断模式或 DMA 模式，也即 UART 能产生内部中断请求或 DMA 请求在 CPU 和串行 I/O 口之间传送数据。

S3C4510B 的 UART 单元特性包括：

- 波特率可编程
- 支持红外发送与接收
- 1～2 个停止位
- 5、6、7 或 8 个数据位
- 奇偶校验

每一个异步串行通讯口都具有独立的波特率发生器、发送器、接收器和控制单元。波特率发生器可由片内系统时钟 MCLK 驱动，或由外部时钟 UCLK（Pin64）驱动；发送器和接收器都有独立的数据缓冲寄存器和数据移位器。

待发送的数据首先传送到发送缓冲寄存器，然后拷贝到发送移位器并通过发送数据引脚 UATXDn 发送出去。接收数据首先从接收数据引脚 UARXDn 移入移位器，当接收到一个字节时就拷贝到接收缓冲寄存器。

SIO 的控制单元通过软件控制工作模式的选择、状态和中断产生。

当使用 UART 的发送中断功能时，应在初始化 UART 之前先写一个字节数据到 UART 的发送缓冲寄存器，这样，当发送缓冲寄存器空时就可以产生 UART 的发送中断。

图 6.2.2 为串行口的功能模块。

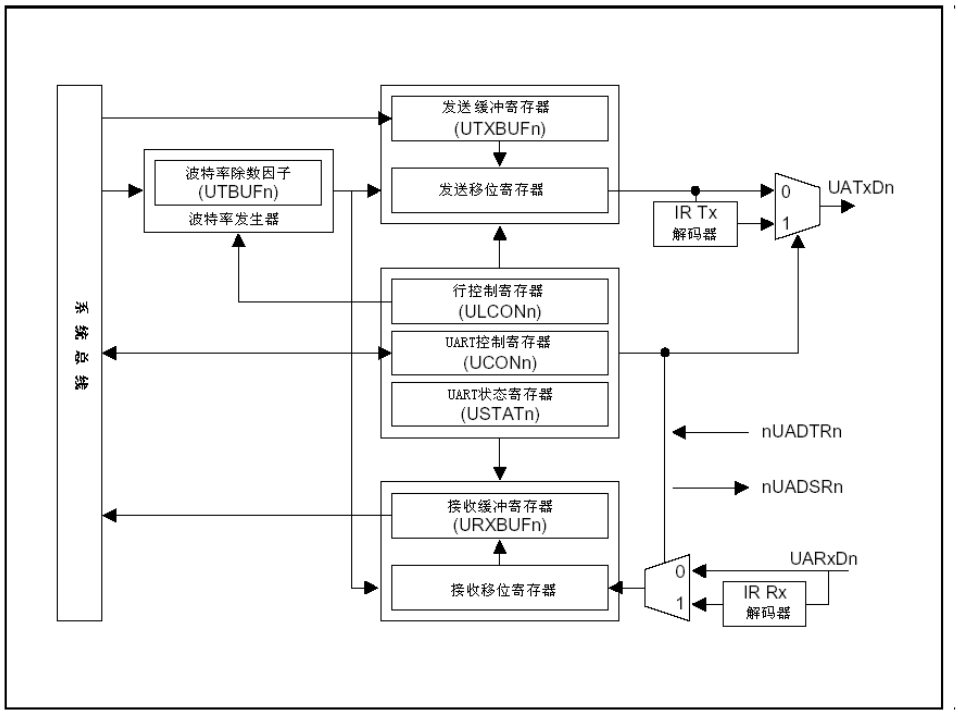


图 6.2.2 串行口功能模块。

表 6-2-1 为 UART 特殊功能寄存器描述

表 6-2-1 UART 特殊功能寄存器

寄存器	偏移地址	操作	功能描述	复位值
ULCON0	0xD000	读/写	UART0 行控制寄存器	0x00
ULCON1	0xE000	读/写	UART1 行控制寄存器	0x00
UCON0	0xD004	读/写	UART0 控制寄存器	0x00
UCON1	0xE004	读/写	UART1 控制寄存器	0x00
USTAT0	0xD008	读	UART 0 状态寄存器	0xC0

USTAT1	0xE008	读	UART1 状态寄存器	0xC0
UTXBUF0	0xD00C	写	UART0 发送保持寄存器	未定义
UTXBUF1	0xE00C	写	UART1 发送保持寄存器	未定义
URXBUF0	0xD010	读	UART0 接收缓冲寄存器	未定义
URXBUF1	0xE010	读	UART1 接收缓冲寄存器	未定义
UBRDIV0	0xD014	读/写	UART0 波特率除数因子寄存器	0x00
UBRDIV1	0xE014	读/写	UART1 波特率除数因子寄存器	0x00
BRDCNT0	0xD018	写	UART0 波特率计数寄存器	0x00
BRDCNT1	0xE018	写	UART1 波特率计数寄存器	0x00
BRDCLK0	0xD01C	写	UART0 波特率时钟监视器	0x00
BRDCLK1	0xE01C	写	UART1 波特率时钟监视器	0x00

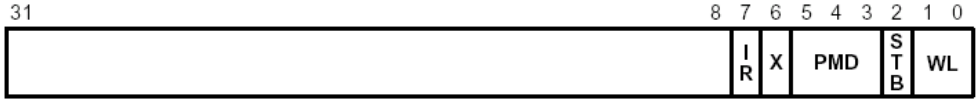
UART 行控制寄存器（UART Line Control Registers，ULCON0、ULCON1）：

寄存器	偏移地址	操作	功能描述	复位值
ULCON0	0xD000	读/写	UART0 行控制寄存器	0x0000，0000
ULCON1	0xE000	读/写	UART1 行控制寄存器	0x0000，0000

表 6-2-2 为 UART 行控制寄存器描述

表 6-2-2　UART 行控制寄存器描述

位	位名	功能描述
[1:0]	每帧字长	该两位指示发送或接收的每帧的数据位：‘00’=5 位，‘01’= 6 位，‘10’= 7 位，‘11’= 8 位。
[2]	停止位数	该位指示每帧数据的停止位数：‘0’= 每帧一个停止位，‘1’= 每帧两个停止位。
[5:3]	校验模式	该三位指示在数据的发送与接收过程中如何生成校验并进行检测：‘0XX’= 无校验，‘100’= 奇校验，‘101’= 偶校验，‘100’= 奇校验，‘110’= 校验强制/检测为 1，‘111’=校验强制/检测为 0。
[6]	串行时钟选择	该位用于选择时钟源： 0 = 内部时钟（MCLK） 1 = 外部时钟（UCLK）
[7]	红外模式选择	S3C4510B 的 UART 模块支持红外（IR）发送与接收。要使能红外模式，需设置 ULCON[7]为 1，否则选择正常 UART 模式。



[1：0]每帧字长（WL）

00=5 位　　01=6 位

10=7 位　　11=8 位

[2]帧末尾停止位（STB）

0 = 每帧一个停止位

1 = 每帧两个停止位

[5：3]校验模式（PMD）

0xx = 无校验

100 = 奇校验

101 = 偶校验

110 = 校验强制/检测为 1

111 = 校验强制/检测为 0

[6]串行时钟选择（SC）

0 = 内部时钟（MCLK）

1 = 外部时钟（UCLK）

[7]红外模式选择（IR）

0 = 正常操作模式

1 = 红外发送模式

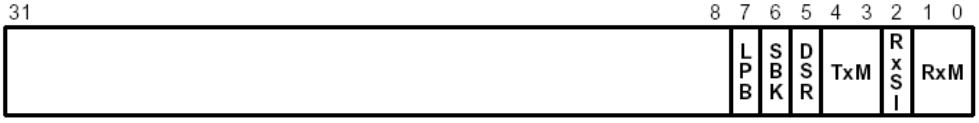
UART 控制寄存器（UART Control Registers，UCON0、UCON1）：

寄存器	偏移地址	操作	功能描述	复位值
UCON0	0xD004	读/写	UART0 控制寄存器	0x0000，0000
UCON1	0xE004	读/写	UART1 控制寄存器	0x0000，0000

表 6-2-3 为 UART 控制寄存器描述

表 6-2-3 UART 控制寄存器描述

位	位名	功能描述
[1:0]	接收模式选择	该两位的值决定当从 UART 接收缓冲寄存器中读取数据时的当前功能： ‘00’=禁止 Rx 模式， ‘01’=产生中断请求， ‘10’=产生 GDMA 通道 0 请求， ‘11’=产生 GDMA 通道 1 请求。
[2]	接收状态中断使能	该位决定当在数据的接收过程中发生异常（如发生间断、帧错误、校验错误或 Overrun 错误）时是否让 UART 产生中断请求： ‘0’= 不产生接收状态中断请求， ‘1’= 产生接收状态中断请求。
[4:3]	发送模式选择	该两位的值决定当写数据到 UART 发送缓冲寄存器中时的当前功能： ‘00’=禁止 Tx 模式， ‘01’=产生中断请求， ‘10’=产生 GDMA 通道 0 请求， ‘11’=产生 GDMA 通道 1 请求。
[5]	数据设备准备好	该位选择是否产生数据设备准备好（DSR）信号输出： 0 = 不产生 DSR 输出， 1 = 产生 DSR 输出（nUADSR 引脚）。
[6]	发送间隔	该位选择是否发送间隔信号： 0 = 不发送间隔信号， 1 = 发送间隔信号。
[7]	回环模式选择	该位选择 UART 是否进入回环模式。在回环模式下，发送的数据的输出置为高电平，发送缓冲寄存器 UTXBUF 在内部直接连接到接收缓冲寄存器 URXBUF。 回环模式仅用于测试目的，在正常操作模式下，该位应为‘0’。



[1：0]接收模式选择（RxM）

00 = 禁止

01 = 产生中断请求

10 = 产生 GDMA 通道 0 请求

11 = 产生 GDMA 通道 1 请求

[2]接收状态中断使能（RxSI）

0 = 不产生接收状态中断

1 = 产生接收状态中断

[4：3]发送模式选择（TxM）

00 = 禁止

01 = 产生中断请求

10 = 产生 GDMA 通道 0 请求

11 = 产生 GDMA 通道 1 请求

[5]数据设备准备好（DSR）

0 = 不产生 DSR 输出（nUADSR 引脚）

1 = 产生 DSR 输出（nUADSR 引脚）

[6]发送间隔（SBK）

0 = 发送间隔信号

1 = 不发送间隔信号

[7]回环使能（LPB）

0 = 正常工作模式

1 = 使能回环模式（仅用于测试）

UART 状态寄存器（UART Status Registers，USTAT0、USTAT1）：

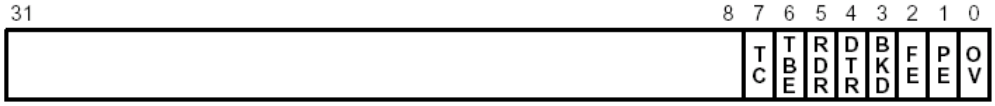
寄存器	偏移地址	操作	功能描述	复位值
USTAT0	0xD008	读	UART0 状态寄存器	0x0000，00C0
USTAT1	0xE008	读	UART1 状态寄存器	0x0000，00C0

表 6-2-4 为 UART 状态寄存器描述

表 6-2-4 UART 状态寄存器描述

位	位名	功能描述
[0]	Overrun 错误	在接收串行数据的操作中，当发生 Overrun 错误时，该位自动置为‘1’。在先前接收到的数据还未读出，而被新收到的数据所覆盖时发生 Overrun 错误。 如果接收状态中断使能位 UCON[2]为‘1’，则当 Overrun 错误发生时产生接收状态中断。 任何时候读取 UART 状态寄存器 USTAT，该位都会自动清零。
[1]	校验错误	在接收串行数据的操作中，当发生校验错误时，该位自动置为‘1’。 如果接收状态中断使能位 UCON[2]为‘1’，则当校验错误发生时产生接收状态中断。 任何时候读取 UART 状态寄存器 USTAT，该位都会自动清零。
[2]	帧错误	在接收串行数据的操作中，当发生数据帧错误时，该位自动置为‘1’。当检测到停止位为‘0’时发生帧错误。 如果接收状态中断使能位 UCON[2]为‘1’，则当帧错误发生时产生接收状态中断。 任何时候读取 UART 状态寄存器 USTAT，该位都会自动清零。
[3]	间隔中断	当接收到间隔信号时，该位自动置为‘1’。 如果接收状态中断使能位 UCON[2]为‘1’，则当接收

		到间隔信号时产生接收状态中断。 任何时候读取 UART 状态寄存器 USTAT，该位都会自动清零。。
[4]	数据终端准备好（DTR）	该位指示数据终端准备好引脚（nUADTR）的当前信号电平。 当该位为‘1’时，数据终端准备好引脚（nUADTR）为低电平，当该位为‘0’时，数据终端准备好引脚（nUADTR）为高电平。
[5]	接收数据准备好	当接收缓冲寄存器 URXBUF 从串行口接收到一个有效数据时，该为自动置为‘1’，然后就可以从 URXBUF 中读取接收到的有效数据。 当该位为‘0’时表示还未接收到一个有效的数据。 当该位为‘1’时，根据当前 UART 接收模式位 UCON[1:0]的设置，可以产生中断或 DMA 请求。
[6]	发送缓冲寄存器空	当发送缓冲寄存器 UTXBUF 为空时，该位自动置为‘1’，此时可以向 UTXBUF 写入下一个要发送的数据。 当该位为‘0’时，表示 UTXBUF 中的数据还未全部拷贝到发送移位寄存器中，此时不能向 UTXBUF 中写入新的发送数据。 当该位为‘1’时，根据当前 SIO 发送模式位 UCON[4:3]的设置，可以产生中断或 DMA 请求。
[7]	发送完成（TC）	当发送缓冲寄存器 UTXBUF 为空且发送移位寄存器也为空时，该位自动置为‘1’。该位用于指示软件是否可以在此时禁用发送功能模块。



[0]Overrun 错误（OV）

0=在接收过程中未产生 Overrun 错误

1=Overrun 错误（如果 UCON[2]置为 1，则会产生接收状态中断）

注：当已接收到的数据还未被读取，而新接收的数据覆盖了原有的数据时，就会产生 Overrun 错误。

[1]校验错误（PE）

0=在接收过程中无校验错误

1=校验错误（如果 UCON[2]置为 1，则会产生接收状态中断）

[2]帧错误（FE）

0=在接收过程中无帧错误

1=帧错误（如果 UCON[2]置为 1，则会产生接收状态中断）

[3]间隔信号检测（BKD）

0=未收到间隔信号

1=收到间隔信号（如果 UCON[2]置为 1，则会产生接收状态中断）

[4]数据终端准备好（DTR）

0=DTR 引脚（nUADTR）置高

1=DTR 引脚（nUADTR）置低

[5]接收数据准备好（RDR）

0=接收缓冲寄存器中无有效数据

1=接收缓冲寄存器中有有效数据（若设置 UCON[1：0]，则产生中断或 DMA 请求）

[6]发送缓冲寄存器空（TBE）

0=发送保持寄存器中有有效数据

1=发送保持寄存器中无数据（若设置 UCON[4：3]，则产生中断或 DMA 请求）

[7]发送结束（TC）

0=正在发送数据

1=发送数据结束

UART 发送缓冲寄存器（UART Transmit Buffer Registers，UTXBUF0、UTXBUF1）：

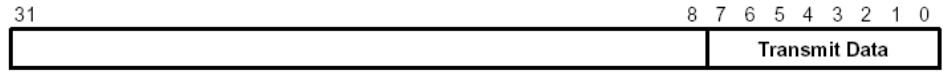
UART 发送缓冲寄存器 UTXBUF0、UTXBUF1，存放待发送的 8 位数据。当把待发送的数据写入该寄存器时，UART 的状态寄存器 USTAT[6]自动清零。

寄存器	偏移地址	操作	功能描述	复位值
UTXBUF0	0xD00C	写	UART0 发送缓冲寄存器	未定义
UTXBUF1	0xE00C	写	UART1 发送缓冲寄存器	未定义

表 6-2-5 为 UART 发送缓冲寄存器描述

表 6-2-5 UART 发送缓冲寄存器描述

位	位名	功能描述
[7:0]	发送数据	该位存放要发送的数据。 当向该寄存器写入数据时，状态寄存器中的发送缓冲寄存器空位 USTAT[6]清‘0’，以防 UTXBUF 中的数据被覆盖。 任何时候向 UTXBUF 中写入新的数据，发送缓冲寄存器空位 USTAT[6]都会被清‘0’。



[7：0]UART 要发送的数据

UART 接收缓冲寄存器（UART Receive Buffer Register，URXBUF0、URXBUF1）：

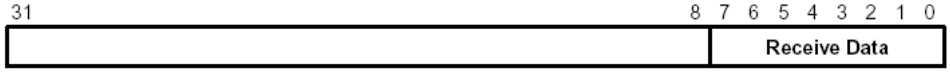
UART 接收缓冲寄存器 URXBUF0、URXBUF1，存放接收到的 8 位串行数据。当 UART 接收完一个数据帧，UART 的状态寄存器 USTAT[5]置为 1。当读取 URXBUF 时，USTAT[5]自动清零。

寄存器	偏移地址	操作	功能描述	复位值
URXBUF0	0xD010	读	UART0 接收缓冲寄存器	未定义
URXBUF1	0xE010	读	UART1 接收缓冲寄存器	未定义

表 6-2-6 为 UART 接收缓冲寄存器描述

表 6-2-6 UART 接收缓冲寄存器描述

位	位名	功能描述
[7:0]	接收数据	该位存放接收到的数据。 当接收完一帧数据时，UART 状态寄存器中的接收数据准备好位 USTAT[5]被置为‘1’，以防从 URXBUF 中的读出的数据无效。 任何时候读取 URXBUF，接收数据准备好位 USTAT[5]都会自动清‘0’。



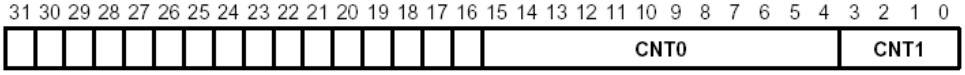
[7：0]UART 接收到的数据

UART 波特率除数因子寄存器（UART Baud Rate Divisor Registers，UBRDIV0、UBRDIV1）：

UART 波特率除数因子寄存器 UBRDIV0、UBRDIV1 的值，决定发送、接收的波特率。

寄存器	偏移地址	操作	功能描述	复位值

UBRDIV0	0xD014	读/写	UART0 波特率除数因子寄存器	0x0000，0000
UBRDIV1	0xE014	读/写	UART1 波特率除数因子寄存器	0x0000，0000



[3：0] 波特率除数因子值（CNT1）

xxx0=除 1

xxx1=除 16

[15：4]时间常数值（CNT0）

CNT0 的计算公式如下：

CNT0=MCLK/（32×BR）－1

MCLK：系统的工作频率。

BR：　通讯的波特率。

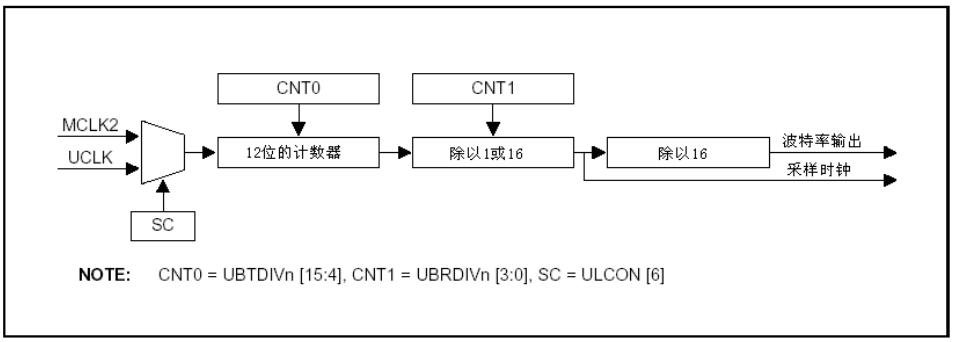
UART 波特率示例：

UART 的波特率发生器的输入时钟可以为系统时钟，也可以从外部引入时钟信号。

若选用系统时钟为波特率发生器的输入时钟，当系统时钟为 50MHz 时，则最大的波特率时钟输出为 MCLK2/16（= 1.5625MHz），其中 MCLK2 为系统时钟 MCLK 除以 2。

UCLK 引脚为 UART0、UART1 的外部时钟输入引脚。UART 波特率发生器的输入时钟 MCLK2 或 UCLK，由寄存器 UCCON[6]选择。

下图为 UART 波特率发生器的结构图和典型的波特率。



Baud Rates (BRGOUT)	MCLK2 = 25 MHz				UCLK = 33 MHz			
	CNT0	CNT1	Freq.	Dev.(%)	CNT0	CNT1	Freq.	Dev.(%)
1200	1301	0	1200.1	0.0	1735	1	1200.08	0.0064
2400	650	0	2400.2	0.0	867	1	2400.15	0.0064
4800	324	0	4807.7	0.2	433	0	4800.31	0.0064
9600	162	0	9585.9	- 0.1	216	0	9600.61	0.0064
19200	80	0	19290.1	0.5	108	0	19113.15	0.45
38400	40	0	38109.8	- 0.8	53	0	38580.15	0.47
57600	26	0	57870.4	0.5	35	0	57870.37	0.47
115200	13	0	111607.1	- 3.1	17	0	115740.74	0.47
230400	6	0	223214.28	3.12	8	0	231481.48	0.47
460860	2	0	520833.34	13.01	4	0	416666.66	9.59

关于 UART 工作原理和使用方法的更详细内容，可参考 S3C4510B 用户手册。

以下的示例完成通过串行口 UART0 发送数据的功能，接收功能的编程与之类似。该示例的通讯协议为：19200 波特、8 位数据、1 位停止、无校验。

打开 CodeWarrior for ARM Developer Suite（或 ARM Project Manager），新建一个项目，并新建一个文件，名为 Init.s，具体内容与第一个例子相同。

保存 Init.s，并添加到新建的项目。

再新建一个文件，名为 main.c，具体内容如下：

```
/************************************************************
* Institute of Automation, Chinese Academy of Sciences
* File Name:    main.c
* Description:
* Author:       JuGuang.Lee
* Date:
************************************************************/
#define ULCON0     (*(volatile unsigned *)0x03FFD000) //UART channel0 line control
register
#define UCON0      (*(volatile unsigned *)0x03FFD004) //UART channel0 control register
#define USTAT0     (*(volatile unsigned *)0x03FFD008) //UART channel0 status register
#define UTXBUF0    (*(volatile unsigned *)0x03FFD00c) //UART channel0 transimit
holding register
#define URXBUF0    (*(volatile unsigned *)0x03FFD010) //UART channel0 recieve buffer
register
#define UBRDIV0    (*(volatile unsigned *)0x03FFD014) //Baud rate divisor register0
#define ULCON1     (*(volatile unsigned *)0x03FFE000) //UART channel1 line control
register
#define UCON1      (*(volatile unsigned *)0x03FFE004) //UART channel1 control register
#define USTAT1     (*(volatile unsigned *)0x03FFE008) //UART channel1 status register
#define UTXBUF1    (*(volatile unsigned *)0x03FFE00c) //UART channel1 transimit
holding register
#define URXBUF1    (*(volatile unsigned *)0x03FFE010) //UART channel1 recieve buffer
register
#define UBRDIV1    (*(volatile unsigned *)0x03FFE014) //Baud rate divisor register1
void InitUART(int Port,int Baudrate);
void PrintUART(int Port,char *s);
int Main()
{
    InitUART(0,0x500);  //19200bps  CPU 工作频率 50MHz  0=COM1;1=COM2
    for(;;){
    PrintUART(0,"Communcation Testting! \r\n");
            }
    return(0);
}
void PrintUART(int Port,char *s)
{
    if(Port==0)
      for(;*s!='\0';s++)
        {
        for(;(!(USTAT0&0x40)););
        UTXBUF0=*s;
        }
    if(Port==1)
      for(;*s!='\0';s++)
        {
        for(;(!(USTAT1&0x40)););
        UTXBUF1=*s;
        }
}
void InitUART(int Port,int Baudrate)
```

```
{
    if(Port==0)
        {
        ULCON0=0x03;
        UCON0=0x09;
        UBRDIV0=Baudrate;
        }
    if(Port==1)
        {
        ULCON1=0x03;
        UCON1=0x09;
        UBRDIV1=Baudrate;
        }
}
```

保存 main.c，并添加到新建的项目。此时可对该项目进行编译链接，生成可执行的映象文件，当可执行的映象文件运行时，会不停的向 UART0 发送字符串“Communcation Testting!”。

6.2.3 中断控制器工作原理与编程示例

中断是计算机的一种基本工作方式，几乎所有的 CPU 都支持中断，S3C4510B 的支持多达 21 个中断源，中断请求可由内部功能模块和外部引脚信号产生。

ARM7TDMI 核可以识别两种类型的中断：正常中断请求（Normal Interrupt Request，IRQ）和快速中断请求（Fast Interrupt Request，FIQ），因此，S3C4510B 的所有中断都可以归类为 IRQ 或 FIQ。S3C4510B 的中断控制器对每一个中断源都有一个中断悬挂位（Interrupt Pending Bit）。

S3C4510B 用如下四个寄存器控制中断的产生和对中断进行处理：

－ 中断优先级寄存器（Interrupt Priority Register）：每一个中断源的索引号写入一个预定义的中断优先级寄存器，以获得特定的优先级。中断优先级预定义为从 0～20。

－ 中断模式寄存器（Interrupt Mode Register）：为每一个中断源定义中断模式，是 IRQ 还是 FIQ。

－ 中断悬挂寄存器（Interrupt Pending Register）：指示中断请求处于悬挂状态（未处理）。如果中断悬挂位被置位，则中断悬挂状态会一直保存，直到 CPU 通过写‘1’到中断悬挂寄存器的相应位清除（注意是写‘1’清除，而不是写‘0’）。当中断悬挂位被置位时，无论中断屏蔽寄存器是否为‘0’，中断服务程序都开始执行。在中断服务程序中，必须通过向中断悬挂寄存器的相应位写‘1’来清除中断悬挂标志，以避免由于同一个中断悬挂位导致中断服务程序的反复执行。

－ 中断屏蔽寄存器（Interrupt Mask Register）：如果中断屏蔽位为‘1’则对应的中断会被禁止，如果中断屏蔽位为‘0’，则对应的中断请求能正常响应。但如果全局中断屏蔽位（位 21）为‘1’，则所有的中断都会被禁止。当有中断请求产生时，对应的中断悬挂位会被置为‘1’，在全局中断屏蔽位和对应的中断屏蔽位为‘0’时，中断请求就会被响应。

S3C4510B 的中断源（Interrupt Sources）

S3C4510B 可支持 21 个中断源，如表 6-2-7 所示：

表 6-2-7 S3C4510B 的中断源

索引号	中断源
[20]	IIC 总线中断
[19]	以太网控制器 MAC 接收中断
[18]	以太网控制器 MAC 发送中断
[17]	以太网控制器 BDMA 接收中断
[16]	以太网控制器 BDMA 发送中断

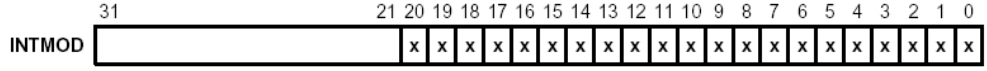
[15]	HDLC 通道 B 接收中断
[14]	HDLC 通道 B 发送中断
[13]	HDLC 通道 A 接收中断
[12]	HDLC 通道 A 发送中断
[11]	定时器 1 中断
[10]	定时器 0 中断
[9]	GDMA 通道 1 中断
[8]	GDMA 通道 0 中断
[7]	UART1 接收与错误中断
[6]	UART1 发送中断
[5]	UART0 接收与错误中断
[4]	UART0 发送中断
[3]	外部中断 3
[2]	外部中断 2
[1]	外部中断 1
[0]	外部中断 0

中断控制器的特殊功能寄存器（Interrupt Controller Special Registers）

中断模式寄存器（Interrupt Mode Register）

中断模式寄存器 INTMOD 通过每一位的设置决定每一种中断是按快速中断（FIQ）还是按正常中断（IRQ）响应。

寄存器	偏移地址	操作	功能描述	复位值
INTMOD	0x4000	读/写	中断模式寄存器	0x0000,0000



[20:0]中断模式位

INTMOD 的 21 个位分别对应于表 6-2-7 中所描述的 21 个中断源，当中断模式位被置为‘1’时，ARM7TDMI 核按 FIQ 方式处理对应的中断，否则按 IRQ 方式处理中断。21 个中断源映射如下：

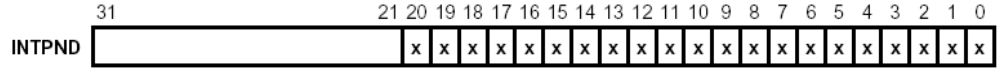
[20] IIC 中断
[19] 以太网控制器 MAC 接收中断
[18] 以太网控制器 MAC 发送中断
[17] 以太网控制器 BDMA 接收中断
[16] 以太网控制器 BDMA 发送中断
[15] HDLC 通道 B 接收中断
[14] HDLC 通道 B 发送中断
[13] HDLC 通道 A 接收中断
[12] HDLC 通道 A 发送中断
[11] 定时器 1 中断
[10] 定时器 0 中断
[9] GDMA 通道 1 中断
[8] GDMA 通道 0 中断
[7] UART1 接收与错误中断
[6] UART1 发送中断
[5] UART0 接收与错误中断
[4] UART0 发送中断
[3] 外部中断 3
[2] 外部中断 2
[1] 外部中断 1

[0] 外部中断 0

中断悬挂寄存器（Interrupt Pending Register）

中断悬挂寄存器 INTPND 保持每一个中断源的中断悬挂位。该寄存器对应的中断悬挂位应在中断服务程序中首先清除，以避免由于同一个中断悬挂位导致中断服务程序的反复执行。

寄存器	偏移地址	操作	功能描述	复位值
INTPND	0x4004	读/写	中断悬挂寄存器	0x0000,0000



[20:0]中断悬挂位

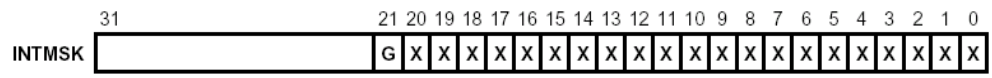
INTPND 的 21 个位分别对应于表 6-2-7 中所描述的 21 个中断源，当中断请求产生时，对应的中断悬挂位被置为‘1’，在中断服务程序中应通过向对应位写入‘1’的方式清除中断悬挂位。21 个中断源映射如下：

[20] IIC 中断
[19] 以太网控制器 MAC 接收中断
[18] 以太网控制器 MAC 发送中断
[17] 以太网控制器 BDMA 接收中断
[16] 以太网控制器 BDMA 发送中断
[15] HDLC 通道 B 接收中断
[14] HDLC 通道 B 发送中断
[13] HDLC 通道 A 接收中断
[12] HDLC 通道 A 发送中断
[11] 定时器 1 中断
[10] 定时器 0 中断
[9] GDMA 通道 1 中断
[8] GDMA 通道 0 中断
[7] UART1 接收与错误中断
[6] UART1 发送中断
[5] UART0 接收与错误中断
[4] UART0 发送中断
[3] 外部中断 3
[2] 外部中断 2
[1] 外部中断 1
[0] 外部中断 0

中断屏蔽寄存器（Interrupt Mask Register）

中断屏蔽寄存器 INTMSK 保持每一个中断源的中断屏蔽位。

寄存器	偏移地址	操作	功能描述	复位值
INTMSK	0x4008	读/写	中断屏蔽寄存器	0x003F,FFFF



[20:0]中断屏蔽位

INTMSK 的 21 个位分别对应于表 6-2-7 中所描述的 21 个中断源，当中断屏蔽位被置为‘1’时，对应的中断请求不能被 CPU 响应，当中断屏蔽位为‘0’时，中断请求会被响应。但如果全局屏蔽位（位 21）为‘1’时，所有的中断请求都不能被响应（但只要中断请求产生，对应的中断悬挂位

都会被设置），当全局屏蔽位被清除时，中断请求会得到响应。21 个中断源映射如下：

[20] IIC 中断
[19] 以太网控制器 MAC 接收中断
[18] 以太网控制器 MAC 发送中断
[17] 以太网控制器 BDMA 接收中断
[16] 以太网控制器 BDMA 发送中断
[15] HDLC 通道 B 接收中断
[14] HDLC 通道 B 发送中断
[13] HDLC 通道 A 接收中断
[12] HDLC 通道 A 发送中断
[11] 定时器 1 中断
[10] 定时器 0 中断
[9] GDMA 通道 1 中断
[8] GDMA 通道 0 中断
[7] UART1 接收与错误中断
[6] UART1 发送中断
[5] UART0 接收与错误中断
[4] UART0 发送中断
[3] 外部中断 3
[2] 外部中断 2
[1] 外部中断 1
[0] 外部中断 0

[21]全局中断屏蔽位

0 = 使能中断请求
1 = 禁止所有的中断请求

关于 S3C4510B 的中断控制器的工作原理和使用方法的更详细内容,可参考 S3C4510B 用户手册。

6.2.4 定时器工作原理与编程示例

S3C4510B 提供两个 32 位的定时器 T0 和 T1，均可工作在间隔模式（Interval Mode）或触发模式（Toggle Mode），对应的信号输出为 TOUT0 和 TOUT1。

通过设置定时器控制寄存器 TCON 中的控制位可以禁止或使能 T0 和 T1。无论何时当定时器计数溢出（减计数）时都会产生中断请求。

间隔模式（Interval Mode）

在这种模式下，当定时器计数溢出时产生一个脉冲输出，该脉冲输出产生定时中断请求，同时从定时器配置输出引脚（TOUTn）Pin196、Pin199 输出。引脚的输出脉冲频率可按下式计算：

f_{OUT} = f_{MCLK}/定时器的数据值

触发模式（Toggle Mode）

在触发模式下，定时器的输出电平会持续到下一次的计数溢出时触发产生翻转。当发生定时器计数溢出时，会产生定时器中断请求，同时由配置引脚输出电平状态。

在该模式下，定时器输出引脚输出占空比为 50%的时钟信号。引脚的输出脉冲频率可按下式计算：

f_{OUT} = f_{MCLK}/（2×定时器的数据值）

图 6.2.3 为定时器输出信号的时序。

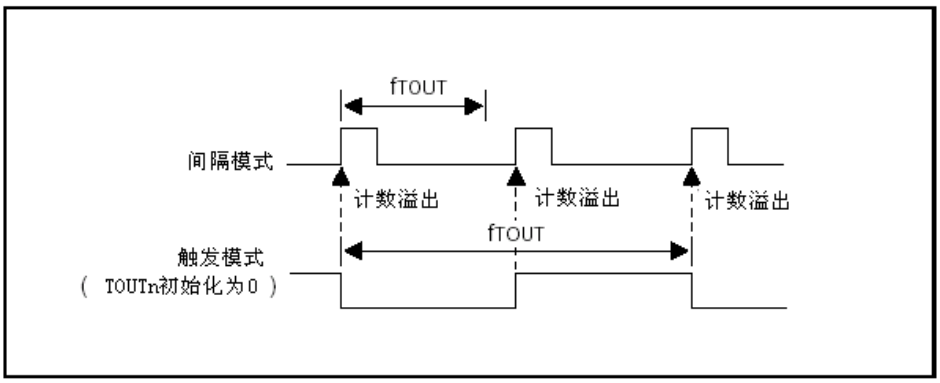


图 6.2.3

定时器输出信号时序

定时器的工作描述

图 6.2.4 为定时器的功能模块图。其工作描述如下：

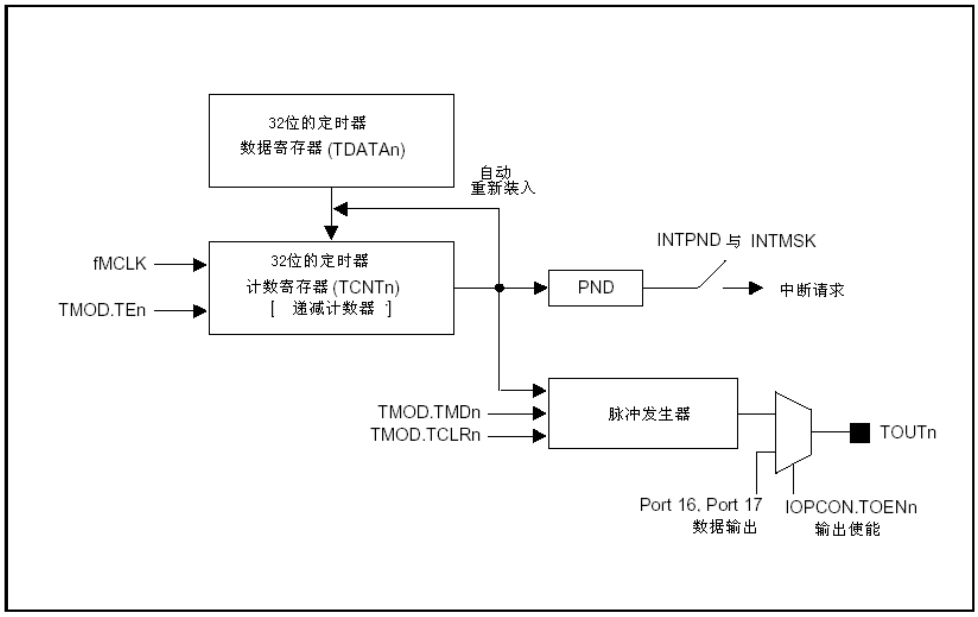


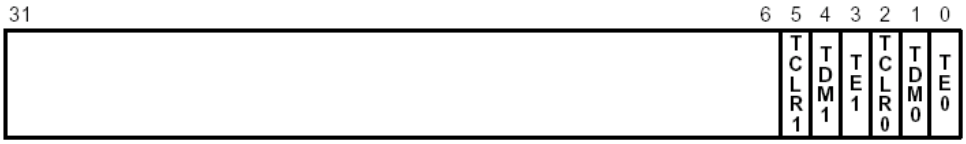
图 6.2.4 定时器的功能模块图

- 当使能计数器时，会向计数器的计数寄存器装入一个数据值，然后计数寄存器开始递减。
- 当定时器计数溢出时，会产生相应的中断请求，同时重新装入原来的数据值并开始递减。
- 在禁用定时器的情况下，可以向定时器的寄存器写入一个新的数据。
- 如果定时器在运行时暂停，原来的数据值不会被自动重新装入。

定时器模式寄存器（Timer Mode Register，TMOD）

定时器模式寄存器 TMOD 用于控制两个 32 位定时器的操作。TMOD 寄存器的设置描述如下：

寄存器	偏移地址	操作	功能描述	复位值
TMOD	0x6000	读/写	定时器模式寄存器	0x0000,0000



[0]定时器 0 使能（TE0）

0 = 禁用定时器 0

1 = 使能定时器 0

[1]定时器 0 模式选择（TMD0）

0 = 间隔模式

1 = 触发模式

[2]定时器 0 初始化 TOUT0 的值（TCLR0）

0 = 在触发模式下初始化 TOUT0 为 0

1 = 在触发模式下初始化 TOUT0 为 1

[3]定时器 1 使能（TE1）

0 = 禁用定时器 1

1 = 使能定时器 1

[4]定时器 1 模式选择（TMD1）

0 = 间隔模式

1 = 触发模式

[5]定时器 1 初始化 TOUT1 的值（TCLR1）

0 = 在触发模式下初始化 TOUT1 为 0

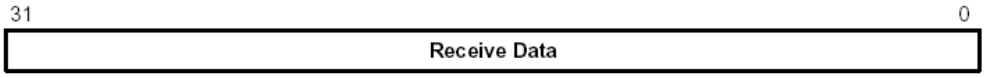
1 = 在触发模式下初始化 TOUT1 为 1

定时器数据寄存器（Timer Data Registers，TDATA0，TDATA1）

定时器数据寄存器 TDATA0 和 TDATA1 的值决定每一个定时器的计数溢出时间的长短。该时间的计算公式为：（定时器数据＋1）个时钟周期。

TDATA 寄存器描述如下：

寄存器	偏移地址	操作	功能描述	复位值
TDATA0	0x6004	读/写	定时器 0 数据寄存器	0x0000,0000
TDATA1	0x6008	读/写	定时器 1 数据寄存器	0x0000,0000



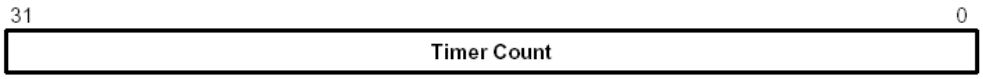
[31：0]定时器 0 或定时器 1 的数据值

定时器计数寄存器（Timer Count Register）

定时器计数寄存器 TCNT0 和 TCNT1 保存定时器 0 或定时器 1 在正常工作情况下的当前计数值。

TCNT 寄存器描述如下：

寄存器	偏移地址	操作	功能描述	复位值
TCNT0	0x600C	读/写	定时器 0 计数寄存器	0xFFFF,FFFF
TCNT1	0x6010	读/写	定时器 1 计数寄存器	0xFFFF,FFFF



[31：0]定时器 0 或定时器 1 的计数值

关于 S3C4510B 定时器的工作原理和使用方法的更详细内容，可参考 S3C4510B 用户手册。

以下的示例显示定时中断服务程序的设计方法，其他中断服务的编程与之类似。

打开 CodeWarrior for ARM Developer Suite（或 ARM Project Manager），新建一个项目，并新建一个文件，名为 Init.s，具体内容如下：

```
; ************************************************************
; Institute of Automation, Chinese Academy of Sciences
;File Name:     Init.s
;Description:   Timer interrupt test.
;Author:        JuGuang, Lee
;Date:
;************************************************************
IOPMOD      EQU     0x3FF5000       ; I/O 口模式寄存器
```

```
IOPDATA     EQU     0x3FF5008       ; I/O 口数据寄存器
TMOD        EQU     0x3FF6000       ; 定时器模式寄存器
TDATA0      EQU     0x3FF6004       ; 定时器数据寄存器
INTMOD      EQU     0x3FF4000       ; 中断模式寄存器
INTPND      EQU     0x3FF4004       ; 中断悬挂寄存器
INTMASK     EQU     0x3FF4008       ; 中断屏蔽寄存器
        AREA    Init,CODE,READONLY
        ENTRY
        B       Reset_Handler    ; 复位异常向量，跳转到程序开始位置。
        B       .                ; 未定义指令异常，跳转到当前位置。
        B       .                ; SWI 异常，跳转到当前位置。
        B       .                ; 指令预取中止异常，跳转到当前位置。
        B       .                ; 数据访问中止异常，跳转到当前位置。
        NOP
        B       IRQ_Handler      ; IRQ 异常，跳转到响应中断服务程序。
        B       .                ; FIQ 异常，跳转到当前位置。
    Reset_Handler
;*********************************
;LED Display
;*********************************
        LDR R1,=IOPMOD
        LDR R0,=&ff
        STR R0,[R1]
        LDR R1,=IOPDATA
        LDR R0,=&03
        STR R0,[R1]
        EOR R0,R0,R0
    LEDDELAY
        ADD R0,R0,#1
        CMP R0,#&180000
        BNE LEDDELAY
        LDR R1,=IOPDATA
        LDR R0,=&0
        STR R0,[R1]
;*************************************
;User Stack
;*************************************
        LDR R0, =0x3FF0000
        LDR R1, =0xE7FFFF80     ; 配置 SYSCFG,片内 4K Cache,4K SRAM
        STR     R1, [R0]
        LDR SP, =0x3FE1000      ; SP 指向 4K SRAM 的尾地址，堆栈向下生成
;*************************************
;Interrupt Special Registers
;*************************************
        LDR R1,=INTMOD          ; 设置中断模式寄存器
        LDR R0,=&0
        STR R0,[R1]
        LDR R1,=INTMSK          ; 设置中断屏蔽寄存器，只允许定时器 0 中断
        LDR R0,=&1FFbFF
        STR R0,[R1]
;*************************************
```

```
;Timer0 Special Registers
;************************************
        LDR R1,=TDATA0          ；定时器 0 的数据寄存器装入初始化值
        LDR R0,=&3FFFFFF
        STR R0,[R1]
        LDR R1,=TMOD            ；使能定时器 0
        LDR R0,=&01
        STR R0,[R1]
        B   .                   ；循环等待中断发生
; ********************************************
; Timer0 Interrupt Service Routine
; ********************************************
    IRQ_Handler
        STMFD   SP!,{R0-R6,LR}  ；保护现场
        LDR R1,=INTPND          ;清 INTPND 中的对应位
        LDR R0,=&400
        STR R0,[R1]
        LDR R0,=IOPDATA         ;读 IOPDATA 的值加一并送回
        LDR R1，[R0]
        ADD R1,R1,#1
        STR R1,[R0]
        LDMFD   SP!,{R0-R6,LR}  ；恢复现场，中断返回
        SUBS    PC,LR,#4
        END
```

保存 Init.s，并添加到新建的项目。此时可对该项目进行编译链接，生成可执行的映象文件。由于异常向量地址是固定不变的，注意在连接该文件时应保证载入地址为 0x0，否则程序不能正常运行。

当可执行的映象文件运行时，会按程序中定义的时间间隔产生定时器中断，通过外部的 LED 显示器显示中断服务程序的执行。

6.2.5 GDMA 工作原理与编程示例

DMA 用于在模块之间进行高速的数据传输，在进行大量数据传输时经常使用，S3C4510B 内建两通道的通用 DMA 控制器（General DMA Controller，GDMA）。在没有 CPU 的干预下，两通道的 GDMA 可完成如下的数据传输：

- 存储器到存储器的双向数据传输。
- UART 到存储器的双向数据传输。

片内的 GDMA 控制器可由软件请求或外部的 DMA 请求（nXDREQ）启动。当一次 DMA 操作完成以后，可由软件再次启动。

S3C4510B 的 DMA 控制器能自动增减源地址与目标地址，同时能以 8 位、16 位或 32 位的方式传送数据。

图 6.2.5 为 GDMA 控制器的功能模块图。

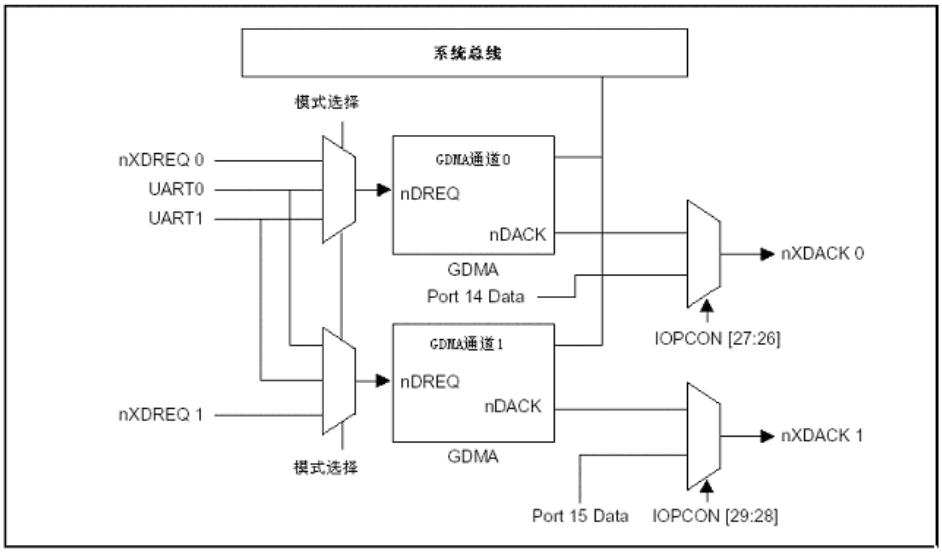


图 6.2.5

GDMA 控制器的功能模块图

GDMA 特殊功能寄存器（GDMA Special Registers）

表 6-2-8 为 GDAM 特殊功能寄存器概述。

表 6-2-8 GDAM 特殊功能寄存器概述

寄存器	偏移地址	操作	功能描述	复位值
GDMACON0	0xB000	读/写	GDMA 通道 0 控制寄存器	0x0000，0000
GDMACON1	0xC000	读/写	GDMA 通道 1 控制寄存器	0x0000，0000
GDMASRC0	0xB004	读/写	GDMA 通道 0 源地址寄存器	未定义
GDMASRC1	0xC004	读/写	GDMA 通道 1 源地址寄存器	未定义
GDMADST0	0xB008	读/写	GDMA 通道 0 目的地址寄存器	未定义
GDMADST1	0xC008	读/写	GDMA 通道 1 目的地址寄存器	未定义
GDMACNT0	0xB00C	读/写	GDMA 通道 0 传输计数寄存器	未定义
GDMACNT1	0xC00C	读/写	GDMA 通道 1 传输计数寄存器	未定义

GDMA 控制寄存器（GDMA Control Registers）

寄存器	偏移地址	操作	功能描述	复位值
GDMACON0	0xB000	读/写	GDMA 通道 0 控制寄存器	0x0000,0000
GDMACON1	0xC000	读/写	GDMA 通道 1 控制寄存器	0x0000,0000

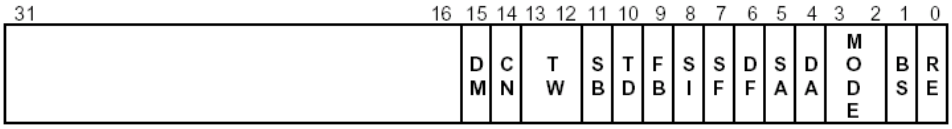
GDMA 控制寄存器描述如表 6-2-9：

表 6-2-9 GDMA 控制寄存器描述

位	位名	功能描述
[0]	运行使能/禁止	该位置‘1’启动 DMA 操作。要停止 DMA 操作，该位必须清‘0’。用户可通过对 GDMA 运行位控制地址（GDMACON 偏移地址＋0x20）的操作控制该位。当使用运行位控制地址时，其他的 GDMA 控制寄存器不受影响。
[1]	忙状态	当 GDMA 操作启动时，该状态位（只读）自动置为‘1’，当 GDMA 空闲时，该位为‘0’。
[3：2]	GDMA 模式选择	GDMA 支持 4 种传输模式： 1） 软件请求模式（存储器到存储器）， 2） 外部 DMA 请求（nXDREQ） 3） UART0 块传输 4） UART1 块传输

[4]	目的地址 增减方式	该位控制在 DMA 操作中目的地址是递增（‘0’）还是递减（‘1’）。
[5]	源地址 增减方式	该位控制在 DMA 操作中源地址是递增（‘0’）还是递减（‘1’）。
[6]	目的地址固定控制	该位决定在 DMA 操作中目的地址是否固定。使用该特性可以完成从多个源地址向一个目的地址的数据传输。
[7]	源地址固定控制	该位决定在 DMA 操作中源地址是否固定。使用该特性可以完成从一个源地址向多个目的地址的数据传输。
[8]	停止中断使能	可以通过设置或清除运行使能位来启动或停止 DMA 操作。如果在 DMA 启动时将该位置‘1’，则当 DMA 操作停止时会产生停止中断，否则不产生。
[9]	4-数据猝发使能	当该位置‘1’时，GDMA 控制器工作于 4-数据猝发模式。在该模式下，一次可完成从 4 个连续的源地址数据的读取，然后传送到 4 个连续的目的地址，此时应注意传输计数寄存器，因为它的值只会递增或递减一次。
[10]	外设传输方向	当模式位[3:2]＝‘10’（UART0 与存储器之间的数据传输）或‘11’（UART1 与存储器之间的数据传输）时，该位规定数据的传输方向。 当该位为‘1’，由存储器到外设。 当该位为‘0’，由外设到存储器。
[11]	单/块模式	该位决定外部 DMA 请求 DMA 操作的次数。 在单模式下（该位为‘0’），每一次 DMA 操作都需要外部的 DMA 请求。 在块模式下（该位为‘1’），只需要一次外部的 DMA 请求即可完成整个 DMA 操作，直到传输计数值为 0。 注意不能同时使用块模式和查询模式，也不能同时使用单模式和连续模式。
[13：12]	传输宽度	该两位决定传输数据的宽度（8 位、16 位或 32 位）。 如果选择字节传输，每次传输完成源地址与目标地址递增或递减 1。 如果选择半字传输，每次传输完成源地址与目标地址递增或递减 2。 如果选择字传输，每次传输完成源地址与目标地址递增或递减 4。
[14]	连续模式	该位可使 DMA 控制器占用系统总线直到 DMA 传输计数器的值为零。必须小心操作该位，使 DMA 传输操作不要超过一个可以接受的时间间隔（例如，不要超过 DRAM 的刷新周期）。 可以同时使用连续模式和软件请求模式。
[15]	查询模式	使用该模式可以提高外部 DMA 操作的速度。当该位为‘1’时，在外部 DMA 请求信号（nXDREQ）有效期间都进行 DMA 数据传输操作，信号的有效时间决定传输的数据量。当 nXDREQ 信号有效且 DMA 控制器取得控制总线权，会一直控制系统总线直到 nXDREQ 信号变为无效。因此，必须小心控制 nXDREQ 信号的有效时间，不要超过一个可以接受的时间间隔（例如，不要超过 DRAM 的刷新周期）。 注意在查询模式下必须清除单/块控制位[11]和连续模式位[14]。

注意：要确保 DMA 操作可靠，必须小心的单独配置 GDMA 控制寄存器的每一位。



[0]运行使能（RE）

0 = 禁用 DMA 操作

1 = 使能 DMA 操作

[1]忙状态（BS）

0 = DMA 空闲

1 = DMA 工作

[3：2]模式选择（MODE）

00 = 软件请求模式（存储器到存储器）。

01 = 外部请求模式（用于外设）。

10 = UART0 块模式。

11 = UART1 块模式。

[4]目的地址增减方向（DA）

0 = 目的地址递增

1 = 目的地址递减

[5]源地址增减方向（SA）

0 = 源地址递增

1 = 源地址递减

[6]目的地址固定控制（DF）

0 = 增/减目的地址

1 = 目的地址不改变（固定）

[7]源地址固定控制（SF）

0 = 增/减源地址

1 = 源地址不改变（固定）

[8]停止中断使能（SI）

0 = 当 DMA 操作停止时不产生停止中断请求

1 = 当 DMA 操作停止时产生停止中断请求

[9]4-数据猝发使能（FB）

0 = 禁止 4-数据猝发模式

1 = 使能 4-数据猝发模式

[10]数据传输方向（仅用于 UART0/1）（TD）

0 = UART0/1 到存储器

1 = 存储器到 UART0/1

[11]单/块模式（SB）

0 = 一次 nXDREQ 初始化一次 DMA 操作

1 = 一次 nXDREQ 初始化整个 DMA 操作

[13：12]传输宽度（TW）

00 = 字节（8 位）　　01 = 半字（16 位）

10 = 字（32 位）　　11 = 未使用

[14]连续模式（CN）

0 = 正常操作模式

1 = 占用总线直到整个 DMA 操作完成

[15]查询模式（DM）

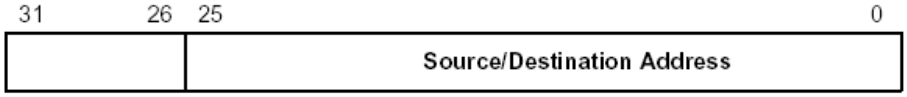
0 = 正常外部 DMA 模式

1 = 查询模式

GDMA 源/目的地址寄存器（GDMA Source/Destination Registers）

GDMA 源/目的地址寄存器为 DMA 通道 0/1 保留 26 位的源/目的地址。根据 GDMA 控制寄存器的设置，在 DMA 操作中源地址和目的地址可以保持不变，也可以递增或递减。

寄存器	偏移地址	操作	功能描述	复位值
GDMASRC0	0xB004	读/写	GDMA 通道 0 源地址寄存器	未定义
GDMASRC1	0xC004	读/写	GDMA 通道 1 源地址寄存器	未定义
GDMADST0	0xB008	读/写	GDMA 通道 0 目的地址寄存器	未定义
GDMADST1	0xC008	读/写	GDMA 通道 1 目的地址寄存器	未定义



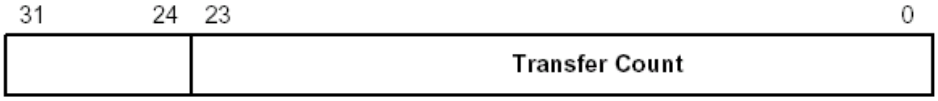
[25：0]源/目的地址

GDMA 传输计数寄存器（GDMA Transfer Count Registers）

GDMA 传输计数寄存器为 DMA 通道 0/1 保留在传输完成后的 24 位当前计数值。

在每一次 DMA 操作完成时计数值总是按 1 增减，无论 GDMA 传输数据的宽度或是处于 4-数据猝发模式。

寄存器	偏移地址	操作	功能描述	复位值
GDMASRC0	0xB004	读/写	GDMA 通道 0 源地址寄存器	未定义
GDMASRC1	0xC004	读/写	GDMA 通道 1 源地址寄存器	未定义
GDMADST0	0xB008	读/写	GDMA 通道 0 目的地址寄存器	未定义
GDMADST1	0xC008	读/写	GDMA 通道 1 目的地址寄存器	未定义



[23：0]传输计数值

GDMA 功能描述

GDMA 可以在请求者和目标之间直接进行数据传输，无需 CPU 的干预。请求者或目标是指存储器、UART 和外设。外设可以通过 nXDREQ 信号请求 GDMA 服务。每一个 DMA 通道都可以通过向相应的寄存器写入数据来进行编程控制，这些寄存器保留有请求者地址、目标地址、传输的数据量以及其他的一些控制内容。UART、外部 I/O 或通过编程（只用于存储器到存储器）都可以请求 DMA 服务。UART 部件与 GDMA 部件在片内已经相连。

当 GDMA 从 nXDREQ 引脚、UART 或软件接收到服务请求时，就开始传输数据，当整个缓冲区的数据传输完毕时，GDMA 就处于空闲状态，如果要进行再一次的数据传输，必须对 GDMA 重新编程，即使是对同一个缓冲区的数据进行重传。

GDMA 的数据传输有三种模式：单模式、块模式和查询模式。

在单模式下，当 4-数据猝发模式处于禁用状态时，每一次 GDMA 请求（可由外部或内部产生）只能传输一个字节数据、或一个半字数据、或一个字数据。当 4-数据猝发模式处于使能状态时，每一次 GDMA 请求可完成 4 次数据传输。在单模式下，每一次的数据传输均需要 DMA 请求。

在块模式下，一次 GDMA 请求（可由外部或内部产生）会完成按寄存器设置的所有数据的传输，直到传输计数器的值为 0。

而在查询模式下，只要检测到 nXDREQ 信号有效，就会持续进行数据的传输。

关于 GDMA 工作原理和使用方法的更详细内容，可参考 S3C4510B 用户手册。

以下的示例完成 GDMA 通道 0 的数据传输，采用软件方式启动 DMA 控制器。

打开 CodeWarrior for ARM Developer Suite（或 ARM Project Manager），新建一个项目，并新

建一个文件，名为 Init.s，具体内容如下：

```
; ************************************************************
; Institute of Automation, Chinese Academy of Sciences
;File Name:     Init.s
;Description:   DMA test.
;Author:        JuGuang，Lee
;Date:
;************************************************************
SRC_ADD     EQU     0x100000        ;源地址
DST_ADD     EQU     0x200000        ; 目的地址
COUNT       EQU     0x100           ;传输数量量
IOPMOD      EQU     0x3FF5000       ;I/O 口模式寄存器
IOPDATA     EQU     0x3FF5008       ;I/O 口数据寄存器
GDMACON0    EQU     0x3FFB000       ; GDMA 通道 0 控制寄存器
GDMASRC0    EQU     0x3FFB004       ; GDMA 通道 0 源地址寄存器
GDMADST0    EQU     0x3FFB008       ; GDMA 通道 0 目的地址寄存器
GDMACNT0    EQU     0x3FFB00C       ; GDMA 通道 0 传输计数寄存器
        AREA    Init,CODE,READONLY
        ENTRY
;*********************************
;LED Display
;*********************************
        LDR R1,=IOPMOD
        LDR R0,=&ff
        STR R0,[R1]
        LDR R1,=IOPDATA
        LDR R0,=&03
        STR R0,[R1]
        EOR R0,R0,R0
    LEDDELAY
        ADD R0,R0,#1
        CMP R0,#&180000
        BNE LEDDELAY
        LDR R1,=IOPDATA
        LDR R0,=&0
        STR R0,[R1]
;*************************************
;User Stack
;*************************************
        LDR R0, =0x3FF0000
        LDR R1, =0xE7FFFF80     ; 配置 SYSCFG,片内 4K Cache,4K SRAM
        STR R1, [R0]
        LDR SP, =0x3FE1000      ; SP 指向 4K SRAM 的尾地址，堆栈向下生成
;*************************************
;Data Initialize
;*************************************
        LDR R1,=SRC_ADD     ; R1 指向源地址，将连续的 0x100 个存储单元初始化
        LDR R0,=0x0
    LOOP
        STR R0,[R1]
        ADD R1,R1,#1
```

```
        ADD R0,R0,#1
        CMP R0,COUNT
        BNE LOOP
;*************************************
; GDMA0 Controll Register
;*************************************
        LDR R1,=GDMASRC0        ; 设置 GDMA 的源地址
        LDR R0,=SRC_ADD
        STR R0,[R1]
        LDR R1,=GDMADST0        ; 设置 GDMA 的目的地址
        LDR R0,=DST_ADD
        STR R0,[R1]
        LDR R1,=GDMACNT0        ; 设置 GDMA 的传输计数器
        LDR R0,=COUNT
        STR R0,[R1]
        LDR R1,=GDMACON0        ; 设置 GDMA 的控制寄存器
        LDR R0,=0x0801
        STR R0,[R1]
        B   .                   ; 循环等待 DMA 传输完毕
        END
```

保存 Init.s，并添加到新建的项目。此时可对该项目进行编译链接，生成可执行的映象文件。

当可执行的映象文件运行时。位于 0x100000 的数据块会以 DMA 方式传输到 0x200000 处。

6.2.6 IIC 总线控制器工作原理

S3C4510B 片内的 IIC 总线控制器具有如下重要特性：

— 仅需要两根传输线。一根为串行数据线（Serial Data Line，SDA），另一根为串行时钟线（Serial Clock Line，SCL）。当 IIC 总线处于空闲状态时，两根传输线均为高电平。

— 连接到总线上的每一个设备都可以通过一个主控器使用唯一的地址进行软件寻址。总线主控器既可以是一个主发送器，也可以是一个主接收器。但 S3C4510B 的 IIC 总线控制器仅支持单主控器模式。

— 支持 8 位、双向，串行数据传输。

— 连接到 IIC 总线的器件数目仅受到最大总线电容（400PF）的限制。

图 6.2.6 为 S3C4510B IIC 总线控制器的功能模块。

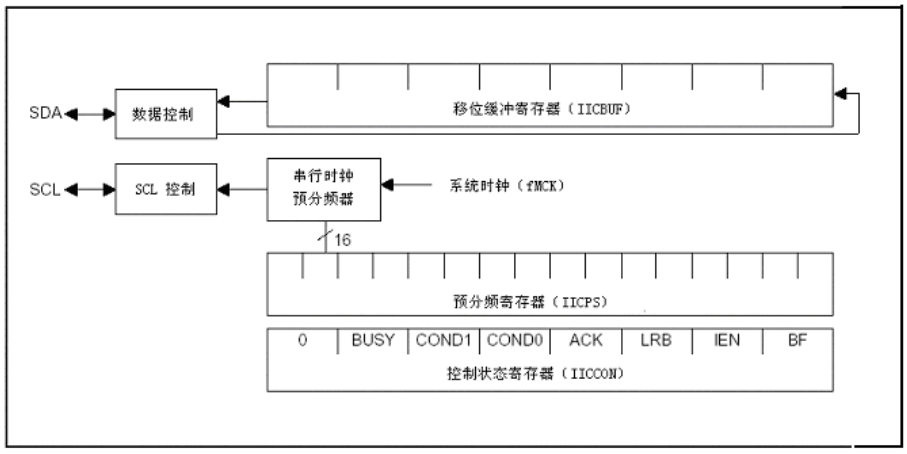


图 6.2.6 S3C4510B IIC 总线控制器的功能模块图

功能描述（Functional Description）

S3C4510B 的 IIC 总线控制器为一个串行 IIC 总线主控器。可通过设置预分频寄存器（Prescaler Register，IICPSR）对串行时钟频率进行编程。串行时钟频率可由下式计算：

MCLK/（16×（预分频寄存器的值＋1）＋3）

可通过对控制状态寄存器（IICCON）的位[5：4]写入“01”发送启动码初始化串行 IIC 总线，然后总线控制器发送 7 位的从设备地址并通过移位缓冲寄存器发送读/写控制位，接收器则在主控器的 SCL 脉冲期间通过将 SDA 线从高电平下拉到低电平作为应答信号。

写数据的操作：先设置控制状态寄存器的 BF 位，然后写入数据到移位缓冲寄存器。移位缓冲寄存器无论是被读还是写，BF 位均会自动清零。若要进行连续的读/写操作，必须设置控制状态寄存器的 ACK 位。

读数据的操作：在设置控制状态寄存器的 BF 位以后，可以进行读数据的操作，当读/写完最后一个字节时，可对 ACK 位进行复位通知发送器/接收器读数据操作的结束。

在读/写操作完成以后，可通过设置 IICCON[5：4]=“10”生成结束码。

IIC 总线概念（IIC-BUS Concepts）

基本操作（Basic Operation）：IIC 总线通过两根传输线，一根串行数据线 SDL，一根串行时钟线 SCL，在连接到总线上的 IC 器件之间传递信息，每一个 IC 器件通过唯一的地址进行识别，根据其特性，可作为发送器或接收器工作。

IIC 总线是一种多主控器总线，有多个 IC 器件具有控制总线的能力。

IIC 总线的数据传输过程描述如下：

第一种情况，一个主 IC 器件要传送数据到其他的从 IC 器件，可分为如下三个步骤：

1、 主 IC 器件寻址从 IC 器件。

2、 主 IC 器件发送数据到从 IC 器件（此时，主器件为发送器，从器件为接收器）。

3、 主 IC 器件终止数据的传输。

第二种情况，一个主 IC 器件要从其他的从 IC 器件获取数据，可分为如下三个步骤：

1、 主 IC 器件寻址从 IC 器件。

2、 主 IC 器件从从 IC 器件接收数据（此时，主器件为接收器，从器件为发送器）。

3、主 IC 器件终止数据的传输。

即使是在第二种情况，也由主 IC 器件产生时序信号并终止数据的传输。

通用特性（General Characteristics）：SDL 和 SCL 均为双向传输线，各通过一个上拉电阻连接到电源正端，当 IIC 总线空闲时，SDL 和 SCL 传输线均为高电平，连接在总线上的 IIC 接口在输出阶段通过漏极开路（Open-drain）或集电极开路(Open-collector)的方式完成线与(Wired-AND)功能。IIC 总线的数据传输速率最高可达到 100Kb/S。可连接到总线上的 IC 器件数目仅受到总线电容的限制（400PF）。

位传输（Bit Transfers）：由于连接到 IIC 总线上的器件各不相同（如有 CMOS 器件，NMOS 器件，TTL 器件等），逻辑 0 或逻辑 1 的电平会根据电源电压的高低发生变化，因此，每传输一个位就产生一个时钟脉冲。

数据有效性（Data Validity）：在时钟信号的高电平期间，SDA 传输线上的电平必须稳定，只有在 SCL 传输线上的时钟信号为低电平时，数据线上的高低电平才允许发生变化。

开始与停止条件（Start and Stop Conditions）：开始和停止条件总是由主器件产生。在开始条件产生后，总线被认为处于忙状态，在完成数据传输产生停止条件后，总线被认为处于空闲状态。

— 开始条件：当 SCL 为高电平时，SDA 产生由高电平到低电平的跳变。

— 停止条件：当 SCL 为高电平时，SDA 产生由低电平到高电平的跳变。

图 6.2.7 为开始与停止条件：

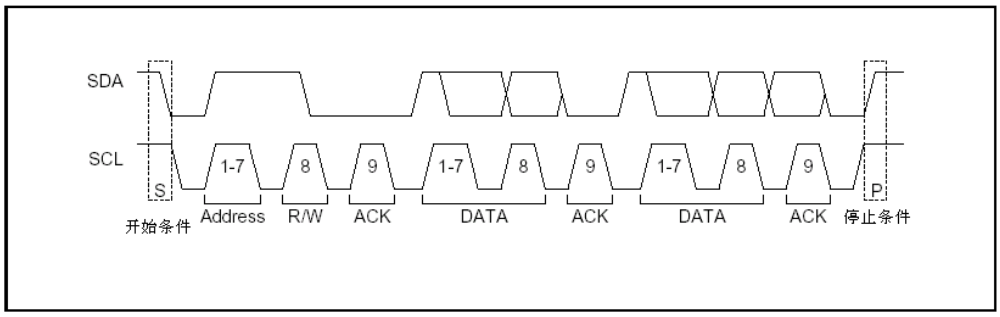


图 6.2.7 开始与停止条件

数据传输操作（Data Transfer Operations）

数据字节格式（Data Byte Format）：每一个写到 SDA 传输线上的数据字节必须为 8 位的长度，每一次传输的字节数没有限制，每传输一个字节必须跟一个应答位 ACK（如图 6.2.7），传输字节时最高位在前（MSB-first）。

如果接收器因为执行其他功能（如中断服务）而不能接收其他剩余的数据字节时，接收器就保持时钟线 SCL 为低电平强制发送器进入等待状态，只有当接收器准备接收其他字节并释放 SCL 传输线时，数据传输才会继续进行。

应答过程（Acknowledge Procedure）：在数据的传输过程中必须带有应答信号，与应答信号相关的时钟脉冲必须由总线主控器产生。在应答时钟脉冲期间，发送器释放 SDA 传输线（为高电平），但此时接收器必须下拉 SDA 传输线，以便在时钟脉冲的高电平期间 SDA 能够保持稳定的低电平。

通常，被寻址的接收器在接收到每一个字节后必须产生一个应答信号，当从接收器不能产生应答信号时，必须释放数据线，然后由主控器产生停止条件中止数据传输。

数据传输格式（Data Transfer Format）：图 6.2.8 显示传输数据的格式。当产生开始条件后，首先发送 7 位的从器件地址，第八位为数据方向位（R/W），“0”表示发送数据（写），“1”表示请求数据（读）。

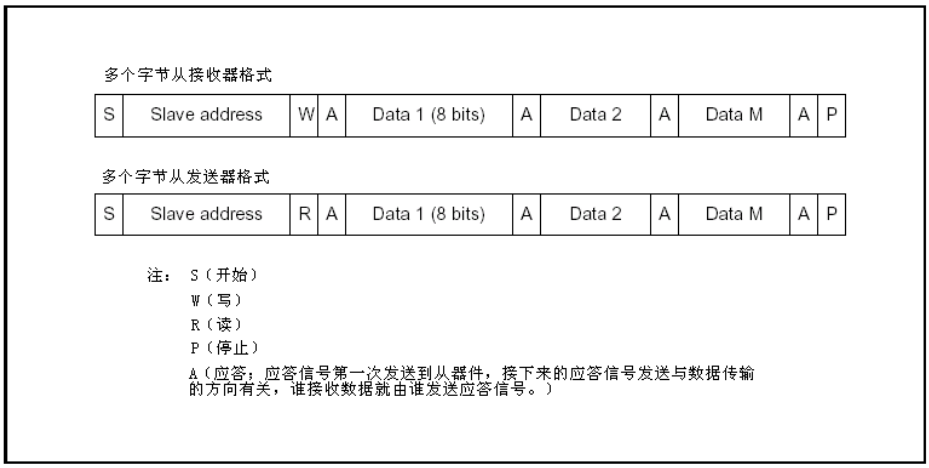


图 6.2.8 数据传输的格式

数据传输一般总是在主控器产生停止条件后中止。但当主控器还要利用总线进行通讯时，主控器可以在不产生前一个停止条件的情况下，再产生开始条件并寻址另一个从器件，该特性可用于支持不同数据传输读/写格式的联合使用。

IIC 总线寻址（IIC-Bus Addressing）：IIC 总线的寻址过程为在开始条件后发送的第一个字节，该字节地址决定主控器选择哪一个从器件。通常，第一个字节总是紧跟在开始过程之后。

还可以通过“广播”寻址方式同时寻址所有的 IC 器件，当使用广播寻址时，理论上所有的 IC 器件都应该返回应答信号，但器件也可以忽略这个这个地址。广播寻址的第二个字节决定其后的操

作。

第一个数据字节的位定义（Definition of Bits in the First Data Byte

第一个数据字节的前 7 位为从器件地址，第 8 位为方向位，决定数据的传输方向（读/写）。

当地址字节发出以后，总线上的每一个 IC 器件将该地址与自己的地址进行比较，若地址匹配，则这个 IC 器件就认为自身被主控器寻址为一个从发送器或从接收器。

广播寻址（General Call Address）：广播寻址方式可用于寻址连接在 IIC 总线上的每一个 IC 器件，但当某个 IC 器件不需要进行数据传输时，将忽略广播寻址而不作任何应答。

如果某个 IC 器件需要获取数据，将发出应答信号并作为一个从接收器。

开始字节（Start Byte）：在每一次的数据传输之前都有一个开始过程，描述如下：

— 一个开始条件，S

— 一个开始字节，“00000001”

— 一个应答时钟脉冲

— 一个重复开始条件，Sr

当需要访问总线的主控器在发送完开始条件 S 后，接着发送开始字节（“00000001”），总线上其余的 IC 器件以较低的采样率对 SDA 传输线进行采样，一直到检测到开始字节中七个零中的某一个。当检测到 SDA 传输线上的低电平时，IC 器件就切换到较高的采样率检测重复开始条件 Sr 用于同步，接收器在检测到重复开始条件 Sr 后复位，忽略该开始字节。

在开始字节发送完毕之后产生一个应答时钟脉冲。

IIC 总线特殊功能寄存器（IIC Bus Special Registers）

IIC 总线控制器由三个特殊功能寄存器：一个控制状态寄存器（IICON），一个预分频寄存器（IICPS）和一个移位缓冲寄存器（IICBUF）。

控制状态寄存器（Control Status Register，IICON）：

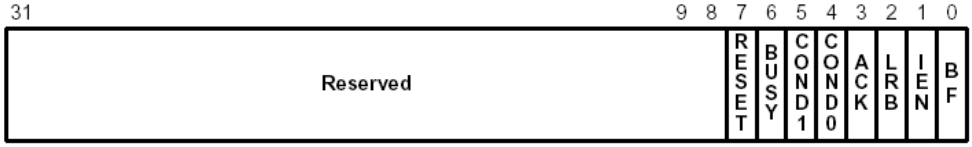
寄存器	偏移地址	操作	功能描述	复位值
IICCON	0xF000	读/写	控制状态寄存器	0x0000,0000

IIC 总线的控制状态寄存器描述如表 6-2-10。

表 6-2-10 IICCON 寄存器描述

位	位名	功能描述
[0]	缓冲标志（BF）	在发送模式下缓冲区空时，或在接收模式下缓冲区满时，BF 置位。 通过对该位写“0”可以清除缓冲区。 IICBUF 寄存器无论是被读还是写，BF 位都自动清零。 如果将该位置为“1”，IIC 总线停止工作，要激活 IIC 总线，应将该位清零。
[1]	中断使能（IEN）	该位置“1”，使能 IIC 总线中断
[2]	最后接收位（LRB）	LRB 位为只读。该位保持 IIC 总线上最后接收到的位。通常该位为从器件的应答信号。要检测从器件的应答信号，可以测试 LRB 位。
[3]	应答使能（ACK）	ACK 位通常置为“1”，以便 IIC 总线控制器在每一个字节后自动发送一个应答信号。 当 IIC 总线控制器工作在接收器模式，且不需要从从发送器接收数据时，该位必须清零。
[5：4]	COND1，COND0	该两位控制开始条件、停止条件和重复开始条件的生成： “00”= 无影响 “01”= 开始 “10”= 停止 “11”= 重复开始

[6]	总线忙（BUSY）	该位为只读，用以指示 IIC 总线是否被使用。为“1”表示总线忙。该位由开始条件和停止条件置位或清零。
[7]	Reset	如果向该位写“1”，IIC 总线控制器复位为初始化状态。
[31：8]	保留	无



[0]缓冲标志（BF）

0 = 当 IICBUF 寄存器被读或写时，该位自动清零。可通过写“0”手动清除 BF。

1 = 在发送模式下缓冲区空时，或在接收模式下缓冲区满时，BF 自动置位。

[1]中断使能（IEN）

0 = 禁止

1 = 使能；BF 位为“1”时产生中断。

[2] 最后接收位（LRB）

使用该只读状态位可以检测接收器（从器件）的应答信号，或当写“11”到 IICCON[5:4]重复开始时监控 SDA 操作。

0 = 最近 SDA 为低电平（接收到 ACK）。

1 = 最近 SDA 为高电平（未接收到 ACK）。

[3]应答使能（ACK）

在接收模式下控制产生 ACK 信号。

0 = 在第 9 个 SCL 脉冲时不产生 ACK 信号。

1 = 在第 9 个 SCL 脉冲时产生 ACK 信号。。

[5：4]COND1 和 CONDO

用于总线控制信号的生成，如开始或停止信号。

00 = 无影响。

01 = 产生开始条件。

10 = 产生停止条件。

11 = SCL 被释放为高电平产生重复开始条件。

[6]总线忙（BUSY）

0 = 总线当前未被使用（空闲）。

1 = 总线正被使用（忙）。

[7]Reset

0 = 正常工作状态。

1 = 复位 IIC 总线控制器。

[31：8]系统保留

移位缓冲寄存器（Shift Buffer Register，IICBUF）：

寄存器	偏移地址	操作	功能描述	复位值
IICBUF	0xF004	读/写	移位缓冲寄存器	未定义

IIC 总线的移位缓冲寄存器描述如表 6-2-11。

表 6-2-11 IICBUF 寄存器描述

位	位名	功能描述
[7：0]	数据	该数据区域用作串行移位寄存器和 IIC 总线的读缓冲接口。所有对总线的读/写操作均要通过该寄存器。IICBUF 寄存器由移位寄存器和数据缓冲两部分构成。 发送数据时，8 位的并行数据首先写入移位寄存器。

		接收数据时，从数据缓冲区读取。
[31：8]	保留	无

预分频寄存器（Prescaler Register，IICPS）：

寄存器	偏移地址	操作	功能描述	复位值
IICPS	0xF008	读/写	预分频寄存器	0x0000,0000

IIC 总线的预分频寄存器描述如表 6-2-12。

表 6-2-12　IICPS 寄存器描述

位	位名	功能描述
[15：0]	预分频值	该预分频值用于产生 IIC 总线的串行时钟。系统时钟除以（16×（预分频值＋1）＋3）作为 IIC 总线的串行时钟。若预分频值为 0，则将系统时钟除以 19 作为 IIC 总线的串行时钟。
[31：16]	保留	无

关于 IIC 总线控制器工作原理和使用方法的更详细内容，可参考 S3C4510B 用户手册。

6.2.7　以太网控制器工作原理

S3C4510B 内嵌一个可以以 10M/100M 的速率工作在半双工或全双工模式下的以太网控制器。在半双工模式下，控制器支持 IEEE802.3 的 CSMA/CD 协议；在全双工模式下，控制器支持包括用于流控的暂停操作的 IEEE802.3 MAC 控制层协议。

以太网控制器的 MAC 层支持媒体独立接口（Media Independent Interface，MII）和带缓冲的 DMA 接口（Buffered DMA Interface，BDI）。MAC 层由发送模块、接收模块、流控模块、用于存储网络地址的匹配地址存储器（Content Address Memory，CAM）以及一些命令寄存器、状态寄存器、错误计数器寄存器构成。

MII 支持在 25MHz 时钟下以 100M 速率的发送与接收操作，和在 2.5MHz 时钟下以 10M 速率的发送与接收操作。同时，MII 遵循 ISO/IEC802-3 中关于从 MAC 层中分离出物理层的媒体独立层标准。

图 6.2.9 为以太网系统的流控框图。

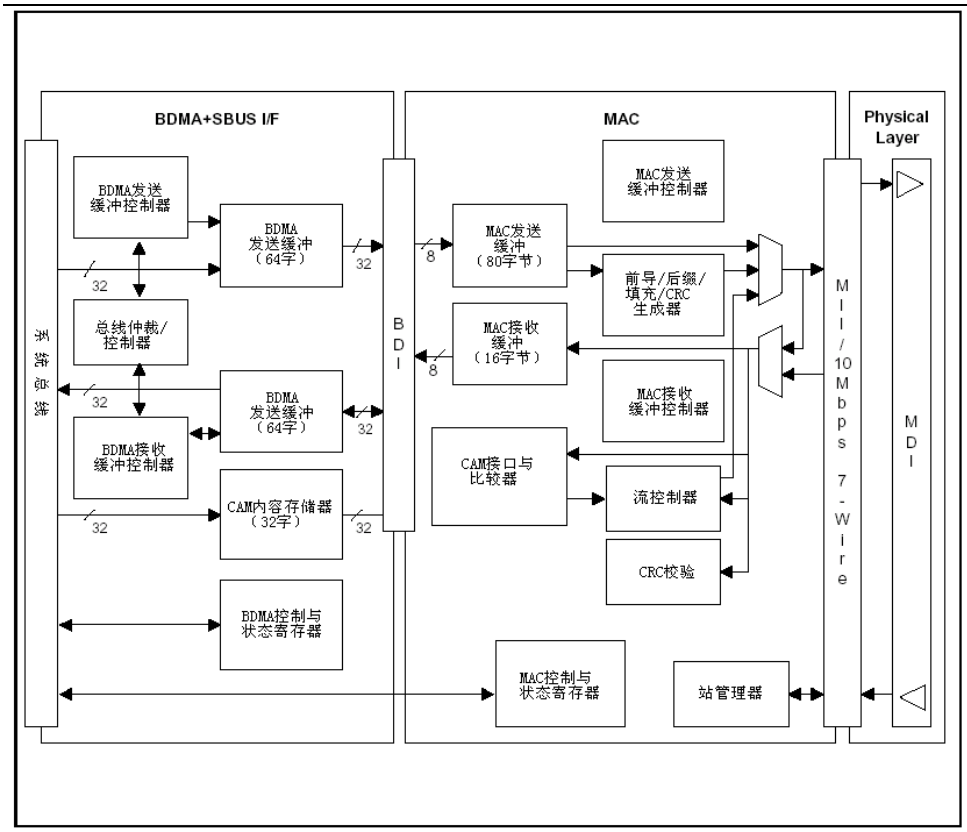


图 6.2.9 以太网系统的流控框图

主要特性

S3C4510B 以太网控制器的主要特性描述如下：

- 为设备联入以太网提供廉价的解决方案。
- 带猝发模式的 BDMA 引擎。
- BDMA 发送/接收缓冲（均为 256 字节）。
- MAC 发送/接收 FIFOs（分别为 80 字节和 16 字节），支持在冲突后重新发送，无需 DMA 请求。
- 数据对准逻辑。
- 端模式变换。
- 支持新/旧传输媒体（与目前的 10M 网络兼容）。
- 10M/100M 的传输速率，提高系统性价比。
- 符合 IEEE802.3 标准，与现有应用系统兼容。
- 支持媒体独立接口（MII）或 7 线制接口。
- 用于物理层配置与连接的站管理（Station Management）信号。
- 片内 CAM（可存储 21 个地址）。
- 支持双倍带宽的全双工模式。
- 硬件支持全双工流控暂停操作。
- 支持特定情况下的长数据包模式。
- 支持用于快速测试的短数据包模式。
- 支持填充生成，数据更易于传输并减少传输时间。

MAC 功能模块

以太网控制器 MAC 层的功能模块描述如表 6-2-13 和图 6.2.10。

表 6-2-13　MAC 功能模块描述

功能模块	描　　述
媒体独立接口（MII）	MII 为物理层和发送/接收模块之间的接口
发送模块	将要发送的数据从发送缓冲区移到 MII。发送模块包括 CRC 生成电路、奇偶校验电路、前导与后缀生成电路。发送模块同时还包含用于处理冲突后的回退和数据帧间间隔的定时器。
接收模块	从 MII 接收数据并存入接收 FIFO。接收模块完成逻辑功能：计算与校验 CRC 值；对从 MII 接收的数据进行奇偶生成和检测最大与最小数据包长度。接收模块同时还包含一个匹配地址存储器（CAM）模块，用于存储网络目的地址，根据该目的地址决定接收或丢弃数据包。
流控模块	辨别 MAC 控制包，并支持用于全双工连接的暂停操作。流控模块同时支持生成暂停控制包，并提供用于暂停控制的定时器和计数器。
MAC 控制（命令）与状态寄存器	控制可编程选项，包括禁止或使能当某条件发生时通知系统的各种信号。状态寄存器保持各种用于错误处理的状态信息，以及用于网络管理的错误计数器累加统计信息等。
回环电路	提供独立于 MII 和物理层的 MAC 层测试。

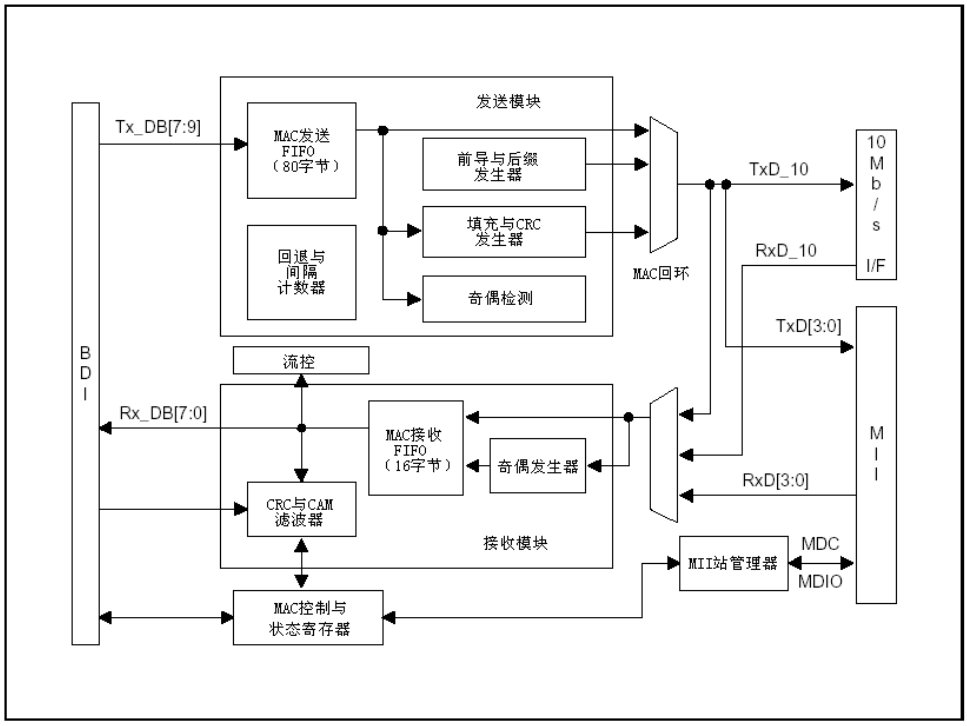


图 6.2.10　MAC 层流控功能模块

媒体独立接口（Media Independent Interface，MII）

发送和接收模块均通过 MII 进行操作，其接口特性描述如下：

- 独立于传输媒体。
- 支持多生产商互操作。
- 支持到 MAC 层和到物理层接口设备的连接。
- 支持 10M 或 100M 的数据传输能力。
- 数据与分隔符的传输和参考时钟同步。
- 提供独立的 4 位数据宽度发送与接收通道。

- 使用 TTL 电平信号，与通用数字 CMOS ASIC 处理器电平兼容。
- 支持到物理层和到站管理设备的连接。
- 提供简单的管理接口。
- 具有驱动有限长度屏蔽电缆的能力。

物理层（Physical Layer Entity，PHY）

物理层完成发送与接收数据的编码/解码。其编码/解码（用于 10BASE-T 的曼切斯特编码、用于 100BASE-X 的 4B/5B 编码以及用于 100BASE-T4 的 8B/6T 编码）方式对 MII 无影响。MII 对原始数据进行接收时，以前导字段开始，到 CRC 字段结束。同时，MII 对原始数据进行发送时，也以前导字段开始，到 CRC 字段结束。MAC 层可生成填充数据并传送到 PHY。

带缓冲的 DMA 接口（Buffered DMA Interface，BDI）

带缓冲的 DMA 接口支持通过系统总线的读/写操作，带有两个 8 位的总线数据收发器，同时可选择奇偶校验功能。数据收发器通过系统接口进行初始化。MAC 层控制器通过响应 BDI 的准备好信号从 BDI 接收数据并发送出去，或将接收到的数据发送给 BDI，由帧结束信号标识各数据包的边界。

MAC 发送模块（MAC Transmit Block）

MAC 发送模块负责数据发送，与 802.3 标准的 CSMA/CD 协议兼容。MAC 发送模块由如下几部分构成：

- 发送 FIFO 和发送控制器。
- 前导与后缀发生器。
- 填充发生器。
- 并行 CRC 发生器。
- 开始逻辑与计数器。
- 回退与重发定时器。
- 发送状态机。

图 6.2.11 为 MAC 发送功能模块图。

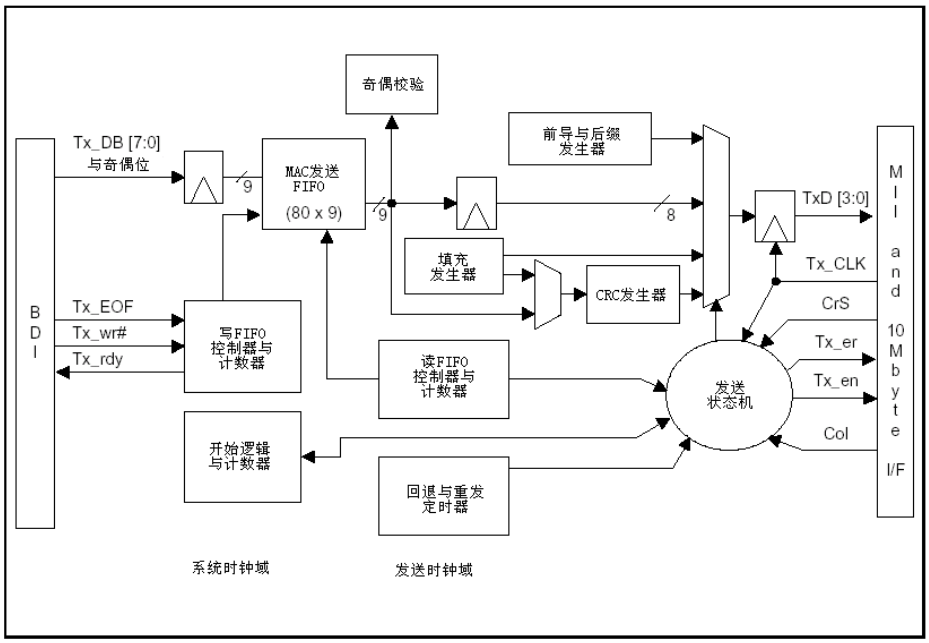


图 6.2.11 MAC 发送功能模块图

发送 FIFO 与读/写控制器（Transmit FIFO and Read/Write Controllers）：发送 FIFO 的大小为 80 字节，每一个字节数据附加一个奇偶校验位，即每个字节为 9 位长度，其中，前 64 个字节数据可以被打包存储和发送，并在发生碰撞时进行重发，无需系统更多的干预。如果没有碰撞发生，则进行数据包发送，剩余的 16 个字节由系统处理，以避免因为 FIFO 不发送而丢失。

当系统接口设置了相应控制寄存器的发送使能位时，发送状态机就从 BDI 请求数据，然后由系统控制器从系统存储器中获取数据。

FIFO 控制器将数据存储在发送 FIFO 中，然后通过一个握手信号通知发送状态机，FIFO 中的数据有效，可以开始发送操作。如果 FIFO 未满，FIFO 控制器向 BDI 请求更多的数据。发送状态机连续的发送数据，直到检测到最后一个字节的帧结束信号，然后追加经过计算的 CRC 值到数据包的末尾（若发送控制寄存器的 CRC 除去位被置位，则不追加），状态寄存器的包发送位被置位，若中断使能同时产生中断。

模块中的写 FIFO 控制器与计数器和发送状态机的读 FIFO 控制器与计数器根据各自的计数值协同工作，尽管他们由不同的时钟源驱动。

FIFO 控制器将数据和校验位存储在 FIFO 中，并对数据进行奇偶校验，如果校验错误，FIFO 控制器就设置一个错误状态位，若对应的中断使能同时产生中断。

前导与后缀发生器（Preamble and Jam Generator）：一旦控制寄存器中的发送使能位被置位，且 FIFO 中有 8 个字节的数据，发送状态机发出 Tx_en 信号启动发送，开始发送前导与起始帧分隔符。

填充发生器（PAD Generator）：如果发送一个短的数据包，MAC 会生成填充字节将短数据包扩展到 64 字节的最小限制，填充字节由“0”组成。有一个控制位可以禁止填充字节的生成。

并行 CRC 发生器（Parallel CRC Generator）：数据包中所有要发送的数据，从目的地址域到其后的所有数据域都可以生成 CRC。用户也可通过设置发送控制寄存器的对应位禁止 CRC 生成。该功能可以用于测试，例如，为测试接收器的错误检测功能，可强制发送 CRC 错误的数据。该功能也可用在某些需要端到端 CRC 校验的网桥与交换机应用场合。

回退与重发定时器（Back-off and Retransmit Timers）：当检测到网络上有碰撞时，发送器模块就停止数据的发送并向网络发送一个特定的阻塞信号，通知所有的节点网络有碰撞。之后，发送器等待一段时间后再重发数据，如果连续 16 次的重发失败，发送状态机就置错误标志位，并在中断使能的情况下产生中断通知系统由于网络的过渡阻塞使数据包发送失败。发送状态机清除 FIFO，MAC 准备下一个数据包的发送。

发送数据奇偶校验器（Transmit Data Parity Checker）：FIFO 中的数据要进行奇校验检测。当数据准备发送时，发送状态机进行奇偶校验，如果检测到奇偶错误，发送数据奇偶校验器作如下操作：

- 停止数据的发送。
- 设置发送状态寄存器中的奇偶错误位。
- 如果中断使能则产生中断。

发送状态机（Transmit State Machine）：发送状态机是发送模块的中央控制逻辑，控制着各种信号的传输、定时器、处理状态寄存器中的错误。

MAC 接收模块（MAC RECEIVE BLOCK）

MAC 接收模块负责数据的接收，与 802.3 标准的 CSMA/CD 协议兼容。

当接收到一个数据包时，接收模块首先检测诸如 CRC 错误、对齐错误、长度错误等错误条件，也可以通过设置控制寄存器的相应位禁止错误检查。接收模块根据 CAM 的状态和目的地址，决定接收或拒绝数据包。MAC 接收模块由以下几个部分构成：

- 接收 FIFO，FIFO 控制器和计数器。
- 接收 BDI 状态机。
- 开始逻辑与计数器。
- 用于地址识别的 CAM 块。
- 并行 CRC 校验器。
- 接收状态机。

接收模块的主要部件如图 6.2.12 所示。

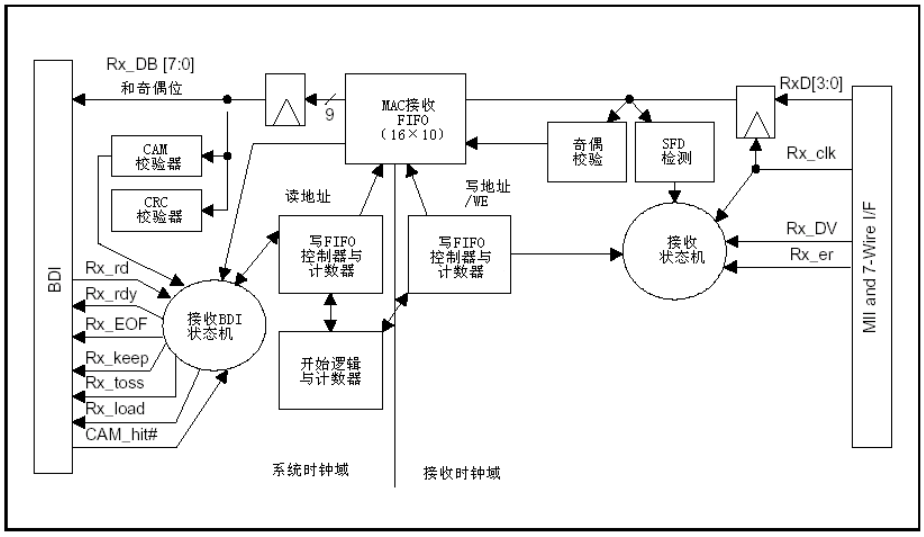


图 6.2.12

MAC 接收功能模块

接收 FIFO 控制器（Receive FIFO Controller）：接收 FIFO 控制器一次接收一个字节的数据，并根据接收到的字节数更新计数器的值。在 FIFO 接收数据时，CAM 模块根据自身存储的地址对数据包的目的地址进行检测，如果 CAM 能识别该地址，FIFO 就继续接收数据包，如果 CAM 不能识别，接收模块就丢弃数据包并清除 FIFO。

地址 CAM 和地址识别（Address CAM and Address Recognition）：CAM 模块完成地址识别。它将接收到的数据包的目的地址与自身存储的地址进行比较，如果地址匹配，接收状态机就继续接收数据。CAM 模块按 6 个字节存储一个地址的方式组织，共有 32 字（128 字节）的大小，最多可存储 21 个地址（126 字节）。

CAM 的地址单元 0、1 和 18 用于发送暂停控制包。因此，用户要发送一个暂停控制包，必须将目的地址写入 CAM0，源地址写入 CAM1，长度/类型、操作码和操作数写入 CAM18。同时，必须对 MAC 发送控制寄存器的发送暂停控制位置位，此外，CAM19 和 CAM20 可用于构造一个由用户定义的控制帧。

接收状态机（Receive State Machine）：在 MII 模式下，接收模块从 MII 的 RxD[3：0]接收数据，数据传输与 25MHz（100M 速率）的接收时钟或与 2.5MHz（10M 速率）的接收时钟同步。在 7-线模式下，传输速率为 10MHz，接收模块从 RxD_10 接收数据。

接收模块检测到数据包的前导与起始帧分隔符（SFD）后，接收状态机按字节对数据进行处理，生成奇偶位并存储到接收 FIFO。如果 CAM 模块接受数据包的目的地址，接收 FIFO 就将数据包的其余部分存储。在接收结束后，接收模块就通过设置接收状态寄存器的对应位标识数据包接收完毕。在接收过程的任何错误都将复位 FIFO，同时接收状态机等待当前数据包的结束，然后进入空闲状态等待下一个前导信号和起始帧分隔符 SFD。

BDMA 接口接收状态机（BDMA Interface Recevie State Machine）：BDMA 接口接收状态机产生 Rx_rdy 信号请求接收 FIFO 中的数据，数据包的最后一个字节数据通过 Rx_EOF 信号标识。如果在接收过程中发生任何错误，或在最后发生 CRC 错误，BDMA 接口接收状态机发出 Rx_toss 信号指示接收到的数据包应该丢弃。

流控模块（Flow Control Block）：流控模块具有如下功能：

- 识别由接收模块接收到的 MAC 控制帧。
- 发送 MAC 控制帧，即使发送器暂停。
- 用于暂停操作的定时器与计数器。
- 命令与状态寄存器（CSR）接口。
- 可选择 MAC 控制帧通过，进行软件处理。

流控模块中的接收逻辑按如下方式识别 MAC 控制帧：

— 长度/类型域必须为 MAC 控制帧规定的特定值；目的地址必须被 CAM 识别；包括 CRC 在内的数据包长度必须为 64 字节；CRC 必须有效；数据帧必须包含一个有效的暂停操作码和操作数。

— 如果长度/类型域不是 MAC 控制帧规定的特定值，MAC 不作响应，数据包被当做普通包处理。如果 CAM 不能识别目的地址，数据包就被 MAC 丢弃；如果包括 CRC 在内的数据包长度不是 64 字节，MAC 也不进行操作，然后数据包被标识为 MAC 控制帧，在允许通过的情况下，传送到软件处理。

用户可通过设置发送状态寄存器中的控制位，进行全双工暂停操作或其他的 MAC 控制功能，即使发送器自身处于暂停状态。两个定时器和两个对应的命令与状态寄存器（CSR）与暂停操作相关。

命令与状态寄存器（CSR）接口在发送与接收控制寄存器和状态寄存器内提供控制位与状态位，这些位可用于初始化 MAC 控制帧的发送、使能或禁用 MAC 的控制功能，以及读取流控计数器的值。

控制位可选择将 MAC 控制帧完全在控制器内处理，或将 MAC 控制帧传送到软件处理。

带缓冲 DMA 接口（Buffered DMA Interface）

带缓冲 DMA（BDMA）控制模块（BDMA Control Blocks）

BDMA 引擎控制一个发送缓冲区和一个接收缓冲区。BDMA 发送缓冲区在数据包被发送时保持数据和状态信息；BDMA 接收缓冲区在数据包被接收时保持数据和状态信息。每一个 FIFO 都有一个控制模块用于控制数据从缓冲区的移入和移出。

图 6.2.13 为 BDMA 控制模块图。

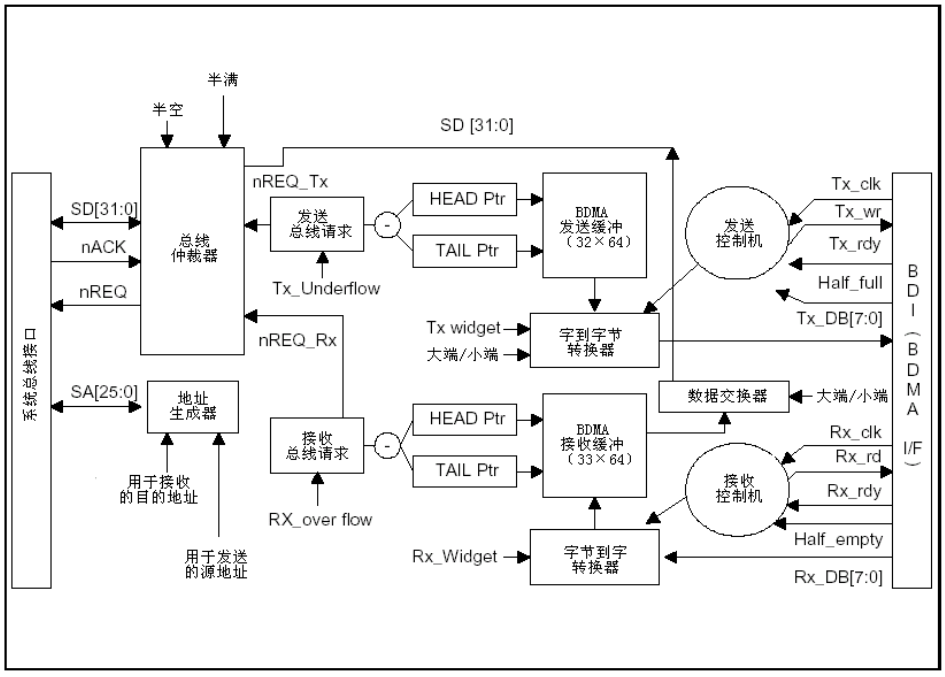


图 6.2.13 BDMA 控制模块图

总线仲裁器（Bus Arbiter）：总线仲裁器决定发送或接收 BDMA 缓冲控制器中的哪一个有较高的优先级访问系统总线，优先级也可以改变。总线仲裁器在以下情况向系统管理器发出总线请求信号（nREQ）：

— 当缓冲区中的数据大于一次接收的数据时。

— 当缓冲区中的剩余空间大于一次发送的数据时。

— 当 EOF（帧结束）标志被存入缓冲区时。

当接收到系统管理器的应答信号后，总线仲裁器根据优先级决定系统总线的访问权。

BDMA 总线控制逻辑（BDMA Bus Control Logic）：BDMA 控制器的功能模块提供对系统总线进行读、写操作的总线控制逻辑，该控制逻辑支持如下操作：

— 猝发尺寸控制，以优化系统总线的使用。

— 发送起始控制（基于发送缓冲区大小的 1/8），使发送能力与系统总线的能力匹配。

— 大、小端字节交换功能，支持大、小端存储格式的数据传输。

— 发送、接收数据对齐部件满足按字对齐的要求。

存储数据结构（Memory Data Structure）：以太网控制器有三种数据结构用于交换控制信息和数据，分别为：

— 发送帧描述符（Transmit Frame Descriptor）。

— 接收帧描述符（Receive Frame Descriptor）。

— 帧数据缓冲（Frame Data Buffer）。

每一个帧描述符由如下元素构成：

— 帧起始地址

— 所有者位

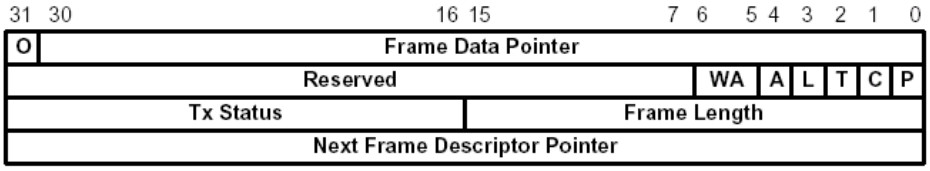
— 发送器控制域

— 状态域

— 帧长度

— 下一个帧描述符指针

以下为发送帧描述符的数据结构：



[0]填充模式选择（P）

0 = 填充模式　　1 = 无填充模式

[1]CRC 模式选择（C）

0 = CRC 模式　　1 = 非 CRC 模式

[2]当前帧发送完毕后 MAC 发送中断使能控制（T）

0 = 禁止　　1 = 使能

[3]小端模式（L）

0 = 大端模式　　1 = 小端模式

[4]帧数据指针递增/递减（A）

0 = 递减　　1 = 递增

[6：5]部件对齐控制（WA）

00 = 无无效字节　　01 = 一个无效字节

10 = 两个无效字节　　11 = 三个无效字节

[31]所有者位（O）

0 = CPU　　1 = BDMA

[30：0]帧数据指针（Frame Data Pointer）

被发送帧数据的地址。

[15：0]帧长度（Frame Length）

发送帧的大小。

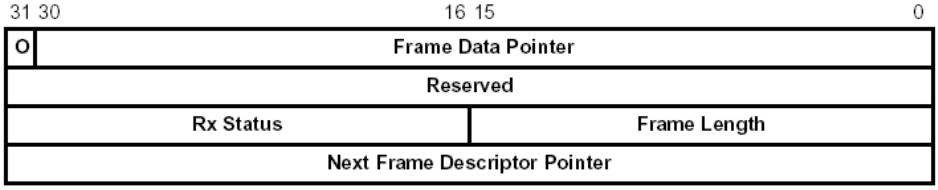
[31：16]发送状态（Tx Status）

该发送帧的发送状态域由 MAC 在发送后更新。

[31：0]下一个帧描述符指针（Next Frame Descriptor Pointer）

下一个帧描述符的地址。

接收帧描述符的数据结构如下：



[31]所有者位（O）

0 = CPU　　1 = BDMA

[30：0]帧数据指针（Frame Data Pointer）

被接收保存的帧数据地址。

[15：0]帧长度（Frame Length）

接收帧的大小。

[31：16]接收状态（Rx Status）

该接收帧的接收状态域由 MAC 在接收完毕后更新。

[31：0]下一个帧描述符指针（Next Frame Descriptor Pointer）

下一个帧描述符的地址。

数据帧（Data Frame）：帧起始地址的最高位，所有者位，控制描述符的所有者。当所有者位为“1”时，描述符属 BDMA 控制器所有，当所有者位为“0”时，描述符属 CPU 所有，描述符的所有者同时拥有相应的数据帧，描述符的帧起始地址指向该数据帧。

当接收到数据帧时，由软件设置 BDMA 模块中的最大帧尺寸寄存器的值为系统帧缓冲尺寸（典型值为：1536 和 2048）。软件同时设置接收帧描述符的起始地址寄存器，使其指向一个帧描述符链。

然后，BDMA 引擎开始设置 BDMA 接收控制寄存器（BDMARXCON）中的 BDMA 接收使能位，当接收到一个数据帧时，数据帧被拷贝到由接收帧起始地址规定的存储器中。接收的数据帧被写入帧数据缓冲区，直到数据帧的结束或帧长度超过设定的最大帧尺寸。

如果成功接收到整个数据帧，由帧描述符中的状态位指示，否则，就设置状态位指示错误发生，同时所有者位被清除并可能产生中断。接着，BDMA 控制器拷贝下一个帧描述符寄存器的值到发送帧描述符起始地址寄存器，如果下一个帧描述符地址为空，BDMA 控制器停止工作，所有后续的帧都被丢弃。否则，描述符被读入，BDMA 控制器按上述步骤开始处理下一个帧。

当 BDMA 控制器读取一个描述符时，若描述符的所有者位没有设置，BDMA 控制器可以有两种处理方式：

- 跳到下一个帧描述符
- 产生中断并停止 BDMA 的工作。

发送帧描述符包含如下内容“

- 一个指向帧数据的 4 字节指针。
- 部件对齐控制位[6：5]
- 帧数据指针递增/递减控制位[4]
- 小端控制位[3]
- 发送后中断使能位[2]
- 无 CRC 位[1]
- 无填充位[0]

在发送过程中，发送帧描述符中两字节的帧长度移入 BDMA 的内部发送寄存器，发送后，发送状态保存到发送帧描述符。然后 BDMA 控制器更新下一个帧描述符地址寄存器。

当发送帧描述符起始地址寄存器指向第一个帧缓冲使，发送器就开始将帧数据发送到发送缓冲存储器。

以太网控制器特殊功能寄存器（Ethernet Controller Special Registers）

S3C4510B 使用的特殊功能寄存器可分为两大类：

- BDMA 控制与状态寄存器。
- MAC 控制与状态寄存器。

BDMA 控制与状态寄存器（BDMA Control and Status Registers）

所有包含存储器地址的寄存器都必须按字对齐格式保存地址。表 6-2-14 为 BDMA 控制与状态寄存器描述。

表 6-2-14　BDMA 控制与状态寄存器

寄存器	偏移地址	操作	功能描述	复位值
BDMATXCON	0x9000	读/写	BDMA 发送控制寄存器	0x00000000
BDMARXCON	0x9004	读/写	BDMA 接收控制寄存器	0x00000000
BDMATXPTR	0x9008	读/写	发送帧描述符起始地址寄存器	0xFFFFFFFF
BDMARXPTR	0x900C	读/写	接收帧描述符起始地址寄存器	0xFFFFFFFF
BDMARXLSZ	0x9010	读/写	接收帧最大尺寸寄存器	未定义
BDMASTAT	0x9014	读/写	BDMA 状态寄存器	0x00000000
CAM	0x9100-0x917C	只读	CAM 内容（32 字）	未定义
BDMATXBUF	0x9200-0x92FC	读/写	BDMA 发送缓冲（64 字），仅用于测试模式寻址	未定义
BDMARXBUF	0x9800-0x98FC 0x9900-0x99FC	读/写	BDMA 接收缓冲（64 字），仅用于测试模式寻址	未定义

BDMA 发送控制寄存器（BDMA Transmit Control Register）

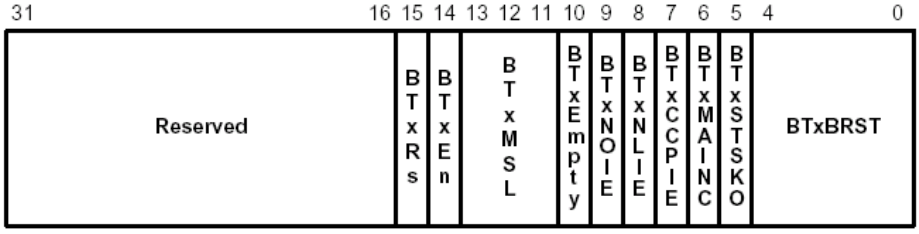
BDMA 发送控制寄存器描述如下：

Registers	Offset	R/W	Description	Reset Value
BDMATXCON	0x9000	R/W	Buffered DMA transmit control register	0x00000000

表 6-2-15　BDMA 发送控制寄存器描述

位	位名	功能描述
[4：0]	BDMA 猝发尺寸（BTxBRST）	在 BDMA 模式下，该值加 1 即为发送尺寸。 若 BTxBRST = 0，则发送尺寸为 1 字； 若 BTxBRST = 31，则发送尺寸为 32 字；
[5]	根据所有者位 BDMA 发送停止或跳过当前帧（BTxSTSKO）	当数据帧的所有者位没有设置时，该位决定 BDMA 发送器跳过当前帧，还是产生中断（如果中断使能）。
[6]	保留	未使用
[7]	BDMA 发送控制包完成中断使能(BTxCCPIE)	当该位被置位时，在 MAC 完成控制包的发送后，BDMA 发送控制包完成中断产生。
[8]	BDMA 发送空表中断使能（BTxNLIE）	当该位被置位时,BDMA 发送空表中断被使能，即当发送帧描述符的起始地址指针为空地址时（0x00000000）产生中断。
[9]	BDMA 非所有者中断使能	该位置位时，BDMA 非所有者中断使能。若当前帧的所有者位不属于 BDMA 控制器，且 BTxSTSKO 位置位时产生

	（BTxNOIE）	中断。
[10]	BDMA 发送缓冲区空中断使能（BTxEmpty）	当该位置“1”时使能发送缓冲区空中断。
[13：11]	BDMA 发送到 MAC 的开始条件（BTxMSL）	这几个位决定当一个新的数据帧到达后，何时将 BDMA 发送缓冲区中新的帧数据移到 MAC 发送 FIFO 中。 “000”表示不等待； “001”表示等到填满 BDMA 发送缓冲器的 1/8； “010”表示等到填满 BDMA 发送缓冲器的 2/8； “011”表示等到填满 BDMA 发送缓冲器的 3/8； “1xx”表示等到填满 BDMA 发送缓冲器的 4/8； 但当帧的最后一个数据到达 BDMA 发送缓冲区时，则立即将其移入 MAC 的发送 FIFO，无论这几位的值如何。
[14]	BDMA 发送使能（BTxEn）	当该位置位时，BDMA 发送模块使能，即使该位被禁止，缓冲区中的数据也要被移入 MAC 发送 FIFO，直到缓冲区下溢。 该位在以下情况自动被禁止： 1、 如果下一个帧指针为空。 2、 如果所有者位为“0”，且 BTxSTSKO 位为“1”。 注意在该位置位之前，帧描述符的起始地址指针必须赋值。
[15]	BDMA 发送模块复位（BTxRS）	该位置“1”，BDMA 发送模块复位。



[4：0]BDMA 的发送尺寸（BTxBRST）

在 BDMA 模式下，该值加 1 即为发送尺寸；若 BTxBRST = 0，则发送尺寸为 1 字；若 BTxBRST = 31，则发送尺寸为 32 字；

[5]根据所有者位 BDMA 发送停止或跳过当前帧（BTxSTSKO）

0 = 如果 BDMA 控制器不是帧的所有者，就跳过当前帧，处理下一个帧描述符。

1 = 如果中断使能，BDMA 发送器就产生中断。

[6]保留

[7] BDMA 发送控制包完成中断使能(BTxCCPIE)

0 = 禁止 BDMA 发送控制包完成中断。

1 = 使能 BDMA 发送控制包完成中断，当 MAC 发送控制包完成时产生中断。

[8]BDMA 发送空表中断使能(BTxNLIE)

0 = 禁止 BDMA 发送空表中断。

1 = 使能 BDMA 发送空表中断，当 BDAMTxPTR 的值为空地址（0x00000000）时产生中断。

[9]BDMA 非所有者中断使能(BTxNOIE)

0 = 禁止 BDMA 非当前发送帧所有者中断。

1 = 使能 BDMA 非当前发送帧所有者中断。

[10]BDMA 发送缓冲区空中断使能(BTxEmpty)

0 = 禁止 BDMA 发送缓冲区空中断。

1 = 使能 BDMA 发送缓冲区空中断。

[13：11]BDMA 发送到 MAC 的开始条件(BTxMSL)

000 = 不等待。

001 = 等待发送缓冲区填满 1/8。

010 = 等待发送缓冲区填满 2/8。

011 = 等待发送缓冲区填满 3/8。

1xx = 等待发送缓冲区填满 4/8。

[14]BDMA 发送使能(BtxEn)

0 = 禁止 BDMA 发送器。

1 = 使能 BDMA 发送器。

[15]BDMA 复位(BTxRS)

0 = 无影响。

1 = 复位 BDMA 发送模块。

BDMA 接收控制寄存器（Buffered DMA Receive Control Register）

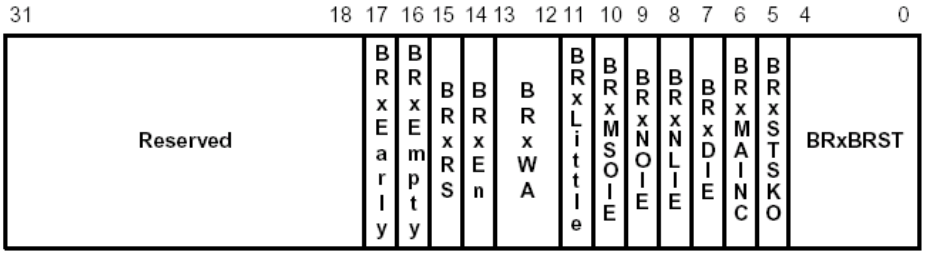
BDMA 的接收控制寄存器 BDMARXCON 描述如下：

Register	Offset Address	R/W	Description	Rest Value
BDMARXCON	0x9004	R/W	Buffered DMA receive control register	0x00000000

表 6-2-16 BDMA 接收控制寄存器描述

位	位名	功能描述
[4：0]	BDMA 接收尺寸（BRxBRST）	在 BDMA 模式下，该值加 1 即为接收尺寸。 若 BTxBRST = 0，则发送尺寸为 1 字； 若 BTxBRST = 31，则发送尺寸为 32 字；
[5]	根据所有者位 BDMA 接收停止或跳过当前帧（BRxSTSKO）	当数据帧的所有者位没有设置时，该位决定 BDMA 接收器跳过当前帧，还是产生中断（如果中断使能）。
[6]	BDMA 接收存储器地址递增/递减（BRxMAINC）	该位决定地址递增或递减。如果该位置为“1”，则地址递增。
[7]	BDMA 每接收帧中断使能(BRxDIE)	当该位被置位时,BDMA 控制器在每次将一个完整的数据帧移入存储器时产生中断。
[8]	BDMA 接收空表中断使能（BRxNLIE）	当该位被置位时,BDMA 接收空表中断被使能，即当接收帧描述符的起始地址指针为空地址时（0x00000000）产生中断。
[9]	BDMA 非所有者中断使能（BRxNOIE）	该位置位时，BDMA 非所有者中断使能。若当前帧的所有者位不属于 BDMA 控制器，且 BRxSTSKO 位置位时产生中断。
[10]	BDMA 接收最大尺寸溢出中断使能（BRxMSOIE）	当该位置“1”时使能 BDMA 接收最大尺寸溢出中断。当接收帧的尺寸大于接收帧最大尺寸寄存器的值时产生中断。
[11]	BDMA 接收大/小端模式（BRxLittle）	该位决定数据以大端或小端格式存储。如果该位置“1”，数据从接收缓冲到系统总线时发生交换。

[13：12]	BDMA 接收字对齐（BRxWA）	接收字对齐控制位决定每一个数据帧的第一个字中有多少个无效字节。这些无效字节由 BDMA 控制器在组装字数据时插入。 “00” = 无无效字节； “01” = 1 个无效字节； “10” = 2 个无效字节； “11” = 3 个无效字节；
[14]	BDMA 接收使能（BRxEn）	当接收使能位置“1”时，BDMA 接收模块使能，即使该位被禁止，MAC 也要接收数据，直到 MAC 接收 FIFO 上溢。 该位在以下情况自动被禁止： 3、 如果下一个帧指针为空。 4、 如果所有者位为“0”，且 BRxSTSKO 位为“1”。 注意在该位置位之前，帧描述符的起始地址指针必须赋值。
[15]	BDMA 接收模块复位（BRxRS）	该位置“1”，BDMA 接收模块复位。
[16]	BDMA 接收缓冲区空中断使能（BRxEmpty）	该位置“1”使能 BDMA 接收缓冲区空中断。
[17]	BDMA 接收数据帧长度通知中断使能（BRxEmpty）	该位置“1”使能 BDMA 接收数据帧长度通知中断。该中断的功能通报从接收数据帧的长度域得到数据帧的长度值。



[4：0]BDMA 的接收尺寸（BRxBRST）

在 BDMA 模式下，该值加 1 即为接收尺寸；若 BRxBRST = 0，则接收尺寸为 1 字；若 BRxBRST = 31，则接收尺寸为 32 字；

[5]根据所有者位 BDMA 接收停止或跳过当前帧（BRxSTSKO）

0 = 如果 BDMA 控制器不是帧的所有者，就跳过当前帧，处理下一个帧描述符。

1 = 如果中断使能，BDMA 接收器就产生中断。

[6]BDMA 接收存储器地址递增/递减（DRxMAINC）

0 = 帧存储器地址递减。

1 = 帧存储器地址递增。

[7] BDMA 每接收帧中断使能(BRxDIE)

0 = 禁止 BDMA 每接收帧中断。

1 = 使能 BDMA 每接收帧中断。

[8]BDMA 接收空表中断使能(BRxNLIE)

0 = 禁止 BDMA 接收空表（0x00000000）中断。

1 = 使能 BDMA 接收空表中断。

[9]BDMA 非所有者中断使能(BRxNOIE)

0 = 禁止 BDMA 非当前接收帧所有者中断。

1 = 使能 BDMA 非当前接收帧所有者中断。

[10]BDMA 接收最大尺寸溢出中断使能(BRxMSOIE)

0 = 禁止 BDMA 接收最大尺寸溢出中断。

1 = 使能 BDMA 接收最大尺寸溢出中断。

[11]BDMA 接收大/小端格式(BRxLittle)

0 = 大端数据帧格式。

1 = 小端数据帧格式（BDMA 接收缓冲区中的帧数据在系统总线进行端格式转换）。

[13：12]BDMA 接收字对齐(BrxWA)

00 = 无无效字节。

01 = 1 个无效字节。

10 = 2 个无效字节。

11 = 3 个无效字节。

[14]BDMA 接收使能(BRxEn)

0 = 禁用 DBMA 接收器。

1 = 使能 BDMA 接收器

[15]BDMA 接收器复位(BRxRS)

0 = 无影响。

1 = 复位 BDMA 接收器

[16]BDMA 接收缓冲空中断(BrxEmpty)

0 = 禁止接收缓冲空中断。

1 = 使能接收缓冲空中断。

[17]BDMA 接收数据帧长度通知中断(BrxEmpty)

0 = 禁止接收数据帧长度通知中断。

1 = 使能接收数据帧长度通知中断。当 BDMA 控制器获得接收帧的长度值时，产生中断。

BDMA 发送帧描述符起始地址寄存器（BDMA Transmit Frame Descriptor Start Address Register）

BDMA 的发送帧描述符起始地址寄存器 BDMATXPTR 描述如下：

寄存器	偏移量	操作	描　　述	复位值
BDMATXPTR	0x9008	读/写	BDMA 发送帧描述符起始地址寄存器	0xFFFFFFFF

表 6-2-17　BDMA 发送帧描述符起始地址寄存器描述

位	位名	功能描述
[25：0]	BDMA 发送帧描述符起始地址	BDMA 发送帧描述符起始地址寄存器的内容为发送帧的帧描述符地址。在 BDMA 的操作过程中，该起始地址会更新为下一个帧地址。

BDMA 接收帧描述符起始地址寄存器（BDMA Receive Frame Descriptor Start Address Register）

BDMA 的接收帧描述符起始地址寄存器 BDMARXPTR 描述如下：

寄存器	偏移量	操作	描　　述	复位值
BDMARXPTR	0x900C	读/写	BDMA 接收帧描述符起始地址寄存器	0xFFFFFFFF

表 6-2-18　BDMA 接收帧描述符起始地址寄存器描述

位	位名	功能描述
[25：0]	BDMA 接收帧描述符起始地址	BDMA 接收帧描述符起始地址寄存器的内容为接收帧的帧描述符地址。在 BDMA 的操作过程中，该起始地址会更新为下一个帧地址。

BDMA 接收帧最大尺寸寄存器（BDMA Receive Frame Maximum Size Register）

BDMA 接收帧最大尺寸寄存器 BDMARXLSZ 描述如下：

Registers	Offset	R/W	Description	Reset Value
BDMARXLSZ	0x9010	R/W	Receive frame maximum size	Undefined

表 6-2-19 BDMA 接收帧最大尺寸寄存器描述

位	位名	功能描述
[15：0]	BDMA 接收帧的最大尺寸（BRxLSZ）	该值控制每帧有多少个字节可以保存到存储器中，如果接收帧的尺寸超过该值，则报告错误。
[31：16]	BDMA 接收帧长度（BRxFSZ），只读	当接收帧长度通知中断产生时，帧长度/类型域保留当前存储帧的长度值。

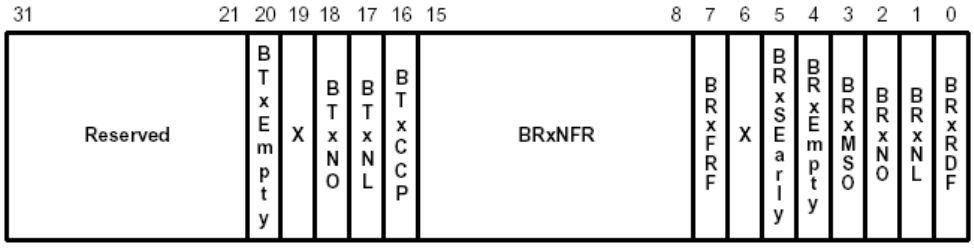
BDMA 状态寄存器（BDMA Status Register）

BDMA 状态寄存器 BDMASTAT 描述如下：

Registers	Offset	R/W	Description	Reset Value
BDMASTAT	0x9014	R/W	Buffered DMA status	0x00000000

表 6-2-19 BDMA 接收帧最大尺寸寄存器描述

位	位名	功能描述
[0]	BDMA 每接收帧操作（BRxRDF）	当该位置“1”时，BDMA 接收器每接收到一个数据帧都移入存储器。若要产生下一帧接收中断，该位必须清零。
[1]	BDMA 接收地址表空（BRxNL）	当该位置“1”时，BDMARXPTR 的地址值为空。
[2]	BDMA 接收器非帧所有者（BRxNO）	当该位置“1”时，BDMA 接收器非当前帧的所有者。
[3]	BDMA 接收最大尺寸溢出（BRxMSO）	当该位置“1”时表示接收帧的尺寸大于接收帧最大尺寸寄存器（BDMARXLSZ）的值。
[4]	BDMA 接收缓冲区空（BRxEmpty）	当该位置“1”时，BDMA 接收缓冲区为空。
[5]	帧长度通知（BRxEarly）	当 BDMA 接收器接收到当前帧的长度/类型域时该位置“1”。
[6]	保留	未使用
[7]	BDMA 接收缓冲区多于一个数据帧（BRxFRF）	当 BDMA 接收缓冲区多于一个数据帧时该位置“1”。。
[15：8]	BDMA 接收缓冲区帧数（BRxNFR）	该值为 BDMA 接收缓冲区当前接收到的总帧数。
[16]	BDMA 发送控制包完成（BTxCCP）	每当 MAC 发送控制包完成时该位置“1”。
[17]	BDMA 接收地址表空（BTxNL）	如果 BDMATXPTR 的值为空，该位置“1”。
[18]	BDMA 发送器非当前帧所有者（BTxNO）	如果 BDMA 发送器非当前帧所有者，该位置“1”。
[19]	保留	未使用
[20]	BDMA 发送缓冲区空（BTxEmpty）	若 BDMA 发送缓冲区空，该位置“1”
[31：21]	保留	未使用



[0]BDMA 每接收帧操作（BRxRDF）

0 = 复位 BDMA 接收器。

1 = 完全接收每一个数据帧并移入存储器。

[1]BDMA 接收地址表空（BRxNL）

0 = 设置新帧描述符的复位状态。

1 = 当前帧描述符的地址为空（0x00000000）。

[2]BDMA 接收器非帧所有者（BRxNO）

0 = BDMA 接收器为当前帧的所有者。

1 = BDMA 接收器非当前帧的所有者（CPU 为所有者），此时 BDMA 接收器停止工作，BRxSTSKO 位置“1”。

[3]BDMA 接收最大尺寸溢出（BRxMSO）

0 = 处于复位状态或下一个帧到达 BDMA 接收缓冲。

1 = 接收帧的大小超过最大帧尺寸。

[4]BDMA 接收缓冲区空（BRxEmpty）

0 = BDMA 接收缓冲区非空。

1 = BDMA 接收缓冲区为空。

[5]BDMA 接收帧长度通知（BRxEarly）

0 = 正常操作。

1 = 通过读取 BDMA 接收最大帧尺寸寄存器 BDMARXLSZ[31：16]的值，可以获取当前帧的长度。

[7]BDMA 接收缓冲区多于一个数据帧（BRxFRF）

0 = BDMA 接收缓冲区仅有一个数据帧。

1 = BDMA 接收缓冲区多于一个数据帧。

[15：8]BDMA 接收缓冲区当前的总数据帧数（BRxNFR）

[16]BDMA 发送控制包完成（BTxCCP）

0 = 写“1”清除该位或复位 BDMA 发送器。

1 = MAC 发送控制包完成。

[17]BDMA 发送接收地址表空（BTxNL）

0 = 设置新帧描述符的复位状态。

1 = 当前帧描述符的地址为空（0x00000000）。

[18]BDMA 发送器非帧所有者（BTxNO）

0 = BDMA 发送器为当前帧的所有者。

1 = BDMA 发送器非当前帧的所有者（CPU 为所有者），此时 BDMA 发送器停止工作，如果 BTxSTSKO 位置“1”。

[20]BDMA 发送缓冲区空（BTxEmpty）

0 = BDMA 发送缓冲区非空。

1 = BDMA 发送缓冲区为空。

注：为使下一帧数据能产生对应的中断，应该通过对其写“1”的方式将位 0、1、2、3、4、16、17、18 和 20 清零。

匹配地址存储器寄存器（Content Address Memory（CAM） Register）

有 21 个 CAM 用于暂停控制包的发送和目的地址的识别，CAM 存放目的每一个目的地址由 6 个字节组成，共占用 32 字的空间（32×4 字节），因此最多可以保留 21 个独立的目的地址。

用户可以使用 CAM 的 0、1 和 18 传送暂停控制包，在传送暂停控制包时，目的地址写入 CAM0，源地址写入 CAM1，包长度/类型，操作码和操作数写入 CAM18，然后设置 MAC 发送控制寄存器的发送暂停控制位。

Registers	Offset	R/W	Description	Reset Value
CAM	0x9100–0x917C	W	CAM content (32 words)	Undefined

表 6-2-20　CAM 寄存器描述

位	位名	功能描述
[31：0]	CAM 内容	CPU 使用 CAM 内容作为目的地址。要使用 CAM 功能，必须设置控制寄存器的相应位。

MAC 寄存器（Media Access Control（MAC）Register）

以太网 MAC 控制器包括一些控制寄存器和状态寄存器，主要有 MAC 控制寄存器、发送与接收控制寄存器，CAM 控制寄存器，一个用于网络管理的计数器，以及一些流控寄存器，如表 6-2-21 所示。

表 6-2-21　MAC 控制与状态寄存器

寄存器	偏移地址	操作	功能描述	复位值
MACON	0xA000	读/写	MAC 控制寄存器	0x00000000
CAMCON	0xA004	读/写	CAM 控制寄存器	0x00000000
MACTXCON	0xA008	读/写	MAC 发送控制寄存器	0x00000000
MACTXSTAT	0xA00C	读/写	MAC 发送状态寄存器	0x00000000
MACRXCON	0xA010	读/写	MAC 接收控制寄存器	0x00000000
MACRXSTAT	0xA014	读/写	MAC 接收状态寄存器	0x00000000
STADATA	0xA018	读/写	站管理数据寄存器	0x00000000
STACON	0xA01C	读/写	站管理控制与地址寄存器	0x00006000
CAMEN	0xA028	读/写	CAM 使能寄存器	0x00000000
EMISSCNT	0xA03C	读清除/写	丢包错误计数器	0x00000000
EPZCNT	0xA040	读	暂停寄存器	0x00000000
ERMPZCNT	0xA044	读	远程暂停寄存器	0x00000000
ETXSTAT	0x9040	读	发送控制帧状态寄存器	0x00000000

MAC 控制寄存器（MAC Control Register）

MAC 控制寄存器为 MAC 提供全局的控制与状态信息，除位[10]（MissRoll）为状态位以外，其余均为 MAC 控制位。

MAC 控制寄存器的设置对发送和接收都有影响，也可以单独对发送或接收操作进行控制。

Registers	Offset	R/W	Description	Reset Value
MACON	0XA000	R/W	MAC control	0x00000000

表 6-2-22　MAC 控制寄存器描述

位	位名	功能描述
[0]	停止请求（HaltReq）	当该位置“1”时，只要当前包的发送或接收完成，就停止数据包的发送或接收。
[1]	立即停止（HaltImm）	当该位置“1”时，立即停止所有的发送或接收操作。
[2]	软件复位（Reset）	当该位置“1”时，复位所有的 MAC 控制与状态寄存器和 MAC 状态机。
[3]	全双工（FullDup）	当该位置“1”时，发送和接收操作同时进行（全双工）。

[4]	MAC 回环（MACLoop）	当该位置“1”时，数据包在 MAC 控制器内进行回环收发。
[5]	保留	未使用
[6]	MII 关闭	该位用于选择连接模式。“1”选择 10M 接口，“0”选择 MII。
[7]	Loop10	当该位置“1”时，Loop_10 外部信号有效
[9：8]	保留	未使用
[10]	Missed Roll	当丢包错误计数器发生卷绕时，该位自动置“1”。
[11]	保留	未使用
[12]	MDC 关闭	当该位为“0”时，使能用于电源管理的 MDC 时钟生成功能，为“1”时，禁用该功能。
[13]	Missed Roll 使能(EnMissRoll)	当该位置“1”时，无论何时发生丢包错误计数器卷绕都将产生中断。
[14]	保留	未使用
[15]	10M 链接状态（Link10）	该的值对应 Link10 引脚的状态。
[31：16]	保留	未使用

CAM 控制寄存器（CAM Control Register）

无论 CAM 是否拒绝数据包，CAM 控制寄存器有三个接收控制位用于接收特殊目的地址的数据包，对应的三类特殊目的地址为：

— 站包地址，第一个字节为偶数。例如：00-00-00-00-00-00。

— 多播组地址，第一个字节为奇数，但地址 FF-FF-FF-FF-FF-FF 除外。例如：01-00-00-00-00-00。

— 广播地址：FF-FF-FF-FF-FF-FF

当使能 CAM 的比较模式，会根据 CAM 存储器的目的地址对到达的信息进行过滤。通过设置三个接收控制位，可以接收所有的数据包，当对 CAM 控制寄存器清零时，拒绝所有的数据包。

Registers	Offset	R/W	Description	Reset Value
CAMCON	0XA004	R/W	CAM control	0x00000000

表 6-2-23 CAM 控制寄存器描述

位	位名	功能描述
[0]	站包接收（StationAcc）	当该位置“1”时，接收目的地址为“单播”（Unicast）站包地址的数据包。
[1]	组接收（GroupAcc）	当该位置“1”时，接收目的地址为“多播”（Multicast）组地址的数据包。
[2]	广播接收（BroadAcc）	当该位置“1”时，接收目的地址为广播地址的数据包。
[3]	Negative CAM（NegCAM）	当该位为“0”时，接收被 CAM 识别的数据包，其余的包丢弃。当该位为“1”时，接收未被 CAM 识别的数据包，其余的包丢弃。
[4]	比较使能（CompEn）	当该位置“1”时使能比较模式。
[31：5]	保留	未使用。

MAC 发送控制寄存器（MAC Transit Control Register）

通过设置 MAC 发送控制寄存器的使能完成控制位和所有的 MAC 错误使能控制位，可在每一个数据包发送完成后产生中断。如果只对 MAC 发送控制寄存器的相应位进行设置，则可以有选择的使能某些特定的中断。

Registers	Offset	R/W	Description	Reset Value
MACTXCON	0XA008	R/W	Transmit control	0x00000000

表 6-2-24 MAC 发送控制寄存器描述

位	位名	功能描述
[0]	发送使能(TxEn)	当该位置“1”时，发送使能，该位清“0”时立即停止发送。
[1]	发送停止请求（TxHalt）	当该位置“1”时，在完成当前帧的发送后停止发送操作。
[2]	无填充（NoPad）	当该位置“1”时，不为少于 64 字节的数据包生成填充。
[3]	无 CRC（NoCRC）	当该位置“1”时，不在数据包末尾增加一个 CRC。
[4]	快速回退（FBack）	当该位置“1”时，使用快速回退用于测试。
[5]	无延迟（NoDef）	当该位置“1”时，禁用延时计数器。延时计数器持续计数直到载波侦听位关闭。
[6]	发送暂停（SdPause）	当该位置“1”时，发送一个暂停命令或其他的 MAC 控制包。当一个完整的 MAC 控制包发送后，发送暂停位自动清除。
[7]	MII 10M SQE 测试模式使能（SQEn）	该位置“1”使能 MII 10M SQE 测试模式。
[8]	Underrun 使能（EnUnder）	当该位置“1”时，若在发送过程中 MAC 发送 FIFO 空，则产生中断。
[9]	延迟使能（EnDefer）	当该位置“1”时，若 MAC 延迟 MAX_DEFERRAL 时间：“0”＝在 100M 时 0.32768 毫秒；“1”＝在 10M 时 3.2768 毫秒，则产生中断。
[10]	无载波使能（EnNCarr）	当该位置“1”时，若在发送一个完整包的过程未检测到载波，则产生中断。
[11]	过冲突使能（EnExColl）	当该位置“1”时，若同一个包在发送过程中产生 16 次冲突，则产生中断。
[12]	延迟冲突使能（EnLateColl）	当该位置“1”时，若在 512 位时间之后发生冲突，则产生中断。
[13]	发送奇偶使能（EnTxPar）	当该位置“1”时，若在 MAC 的发送 FIFO 中检测到奇偶错误，则产生中断。
[14]	完成使能（EnComp）	当该位置“1”时，无论 MAC 发送或丢弃一个包，都产生中断。
[31：15]	保留	未使用

MAC 发送状态寄存器（MAC Transit Status Register）

当有对应的事件发生时，MAC 发送状态寄存器 MACTXSTAT 设置发送状态标志，同时，如果发送控制寄存器的相应中断使能位被设置，还产生中断请求。

Registers	Offset	R/W	Description	Reset Value
MACTXSTAT	0XA00C	R/W	Transmit status	0x00000000

表 6-2-25 MAC 发送状态寄存器描述

位	位名	功能描述
[3：0]	发送冲突计数（TxColl）	该 4 位记录在成功发送数据帧后产生的冲突次数。
[4]	过冲突（ExColl）	当该位置“1”时表示同一个包在发送过程中产生了 16 次冲突，在这种情况下，该数据包被丢弃。
[5]	发送延迟（TxDefered）	当该位置“1”时表示数据包在发送过程中被延迟。

[6]	暂停（Paused）	当该位置“1”时表示数据包在发送时由于收到暂停命令而被延迟。
[7]	发送中断（IntTx）	当该位置“1”时表示在发送数据包时有中断发生。
[8]	Underrun（Under）	当该位置“1”时表示在包的发送过程中 MAC 发送 FIFO 空。
[9]	延迟（Defer）	当该位置“1”时表示 MAC 延迟 MAX_DEFERRAL 时间：在 100M 时 0.32768 毫秒；在 10M 时 3.2768 毫秒。
[10]	无载波（NCarr）	当该位置“1”时表示在发送一个数据包的过程未检测到载波。
[11]	信号质量错误（SQE）	当该位置“1”时表示在规定的时间内未报告 SQE 测试信号。
[12]	延迟冲突（LateColl）	当该位置“1”表示在 512 位时间之后发生冲突。
[13]	发送奇偶错误（TxPar）	当该位置“1”时表示在 MAC 的发送 FIFO 中检测到奇偶错误。
[14]	完成（Comp）	当该位置“1”时表示 MAC 发送或丢弃了一个包。
[15]	发送停止（TxHalted）	如果由于清除了 TxEn 位（MACTXCON 中）或设置了 HaltImm 位（MACON 中），使发送停止，该位置“1”。
[31：16]	保留	未使用

MAC 接收控制寄存器（MAC Receive Control Register）

通过设置 MAC 接收控制寄存器的数据包好位和所有的 MAC 错误使能控制位，可在每一个数据包接收完成后产生中断。如果只对 MAC 接收控制寄存器的相应位进行设置，则可以有选择的使能某些特定的中断。

Registers	Offset	R/W	Description	Reset Value
MACRXCON	0XA010	R/W	Receive control	0x00000000

表 6-2-26　MAC 接收控制寄存器描述

位	位名	功能描述
[0]	接收使能（RxEn）	当该位置“1”时，接收使能，该位清“0”时立即停止接收。
[1]	接收停止请求（RxHalt）	当该位置“1”时，在完成当前帧的接收后停止接收操作。
[2]	长使能（LongEn）	当该位置“1”时，接收数据包的长度大于 1518 字节。
[3]	短使能（ShortEn）	当该位置“1”时，接收数据包的长度小于 64 字节。
[4]	去除 CRC 值（StripCRC）	当该位置“1”时，在完成 CRC 校验后，从数据包中去除 CRC 值。
[5]	控制包通过（PassCtl）	当该位置“1”时，使能控制包到达 MAC。
[6]	忽略 CRC 值（IgnoreCRC）	当该位置“1”时禁用 CRC 校验。
[7]	保留	未使用
[8]	对齐使能（EnAlign）	当该位置“1”时使能对齐中断。当接收到的数据包长度（按位）不是八的倍数或 CRC 无效则产生对齐中断。
[9]	CRC 错误使能（EnCRCErr）	当该位置“1”时使能 CRC 中断。当接收到的数据包 CRC 无效或在接收时物理层产生 Rx_er 信号，则产生 CRC 中断。
[10]	上溢使能（EnOver）	当该位置“1”时使能上溢中断。当接收数据包时 MAC 的接收 FIFO 满，产生上溢中断。
[11]	长错误使能（EnLongErr）	当该位置“1”时使能长错误中断。当接收数据帧长度超过 1518 时，产生长错误中断，除非长使能位被置“1”。
[12]	保留	未使用

[13]	接收奇偶使能（EnRxPar）	当该位置“1”时，若在 MAC 的接收 FIFO 中检测到奇偶错误，则产生中断。
[14]	好使能（EnGood）	当该位置“1”时，若完整接收到一个数据包产生数据包好中断。
[31：15]	保留	未使用

注：以上的数据帧长度不包括前导和起始帧分隔符 SFD。

MAC 接收状态寄存器（MAC Receive Status Register）

当有对应的事件发生时，MAC 接收状态寄存器 MACRXSTAT 设置接收状态标志，当某个状态位被设置时，会保持到另一个数据包到达或通过软件写“1”的方式清除，同时，如果接收控制寄存器的相应中断使能位被设置，还产生中断请求。

Registers	Offset	R/W	Description	Reset Value
MACRXSTAT	0XA014	R/W	Receive status	0x00000000

表 6-2-25 MAC 接收状态寄存器描述

位	位名	功能描述
[4：0]	保留	未使用
[5]	控制帧到达（CtlRecd）	当接收的数据包为 MAC 控制帧（Type = 8808H），或当 CAM 能识别包目的地址，或帧长度为 64 字节，该位置“1”
[6]	接收中断（CtlRecd）	当接收数据包产生中断时，该位置“1”。
[7]	10M 接收状态（Rx10Stat）	当数据包从 7 线接口接收时，该位置“1”，当数据包从 MII 接收时，该位清零。
[8]	对齐错误（AlignErr）	当数据帧的长度（按位）不是 8 的整数倍或 CRC 无效时，该位置“1”。
[9]	CRC 错误（CRCErr）	当数据帧尾部的 CRC 与计算值不匹配时，或在包的接收过程中物理层 PHY 发出 Rx_er 信号时，该位置“1”。
[10]	溢出错误（Overflow）	当需要存储接收的数据而 MAC 接收 FIFO 满时，该位置“1”。
[11]	数据长错误（LongErr）	当 MAC 接收到的数据帧超过 1518 字节时，该位置“1”。但当接收控制寄存器中的长数据使能位为“1”时，该位不会被置为“1”。
[12]	保留	未使用
[13]	接收奇偶错（RxPar）	如果在 MAC 接收 FIFO 中检测到奇偶错时，该位置“1”。
[14]	接收好（Good）	如果数据包被成功接收，未发生任何错误，该位置“1”。当 MAC 接收控制寄存器中的 EnGood 为“1”时，同时会产生中断。
[15]	接收停止（RxHalted）	如果由于清除了 RxEn 位（MACRXCON 中）或设置了 HaltImm 位（MACON 中），使接收停止，该位置“1”
[31：16]	保留	未使用

MAC 站管理数据寄存器（MAC Station Management Data Register）

Registers	Offset	R/W	Description	Reset Value
STADATA	0XA018	R/W	Station management data	0x00000000

表 6-2-26 站管理数据寄存器描述

位	位名	功能描述
[15：0]	站管理数据	该寄存器的内容为用于站管理功能的 16 位数据。

MAC 站管理控制与地址寄存器（MAC Station Management Control and Address Register）

Registers	Offset	R/W	Description	Reset Value
STACON	0XA01C	R/W	Station management control and address	0x00008000

表 6-2-27　站管理控制与地址寄存器描述

位	位名	功能描述
[4：0]	物理层寄存器地址	该 5 位的内容为物理层寄存器地址。
[9：5]	物理层地址	该 5 位的内容为物理层设备的地址。
[10]	写（Wr）	该位置“1”，初始化为写操作，清零为读操作。
[11]	忙（Busy）	当开始读/写操作时，该位置“1”，当读/写操作结束时，MAC 控制器自动清除该位。
[12]	无前导（PreSup）	当该位置“1”时，前导不送到 PHY，为“0”时送到 PHY。
[15：13]	MDC 时钟周期	用于控制 MDC 周期。 STACON[15：13]　　MDC 周期 000　　16×（1/fMCK） 001　　18×（1/fMCK） 010　　20×（1/fMCK） . . 111　　30×（1/fMCK） MDC 周期 = STACON[15：13]×2＋16 STACON[15：13]默认值为“100”。
[31：16]	保留	未使用

CAM 使能寄存器（CAM Enable Register）

CAM 使能寄存器，CAMEN，标识哪一个 CAM 地址有效，可用于直接的比较工作模式。寄存器的 0 到 20 位对应于 21 个 CAM 地址的有效性，当 CAM 地址少于 21 个时，寄存器的较高位被忽略。

Registers	Offset	R/W	Description	Reset Value
CAMEN	0XA028	R/W	CAM enable	0x00000000

表 6-2-28　CAM 使能寄存器描述

位	位名	功能描述
[20：0]	CAM 使能（CAMEn）	若某位置“1”，则对应的 CAM 地址有效，清零则对应的 CAM 地址无效。
[31：21]	保留	未使用

MAC 丢包错误计数寄存器（MAC Missed Error Count Register）

MAC 丢包错误计数寄存器（EMISSCNT）的值指示由于各种错误丢弃的数据包数量。结合数据包发送与接收的状态信息，该寄存器和另外两个暂停计数寄存器提供用于站管理所需要的信息。

对该寄存器进行读操作会清除其内容。

Registers	Offset	R/W	Description	Reset Value
EMISSCNT	0XA03C	R(Clr)/W	Missed error count	0x00000000

表 6-2-29　MAC 丢包错误计数寄存器描述

位	位名	功能描述
[15：0]	对齐错误计数（AlignErrCnt）	接收到的对齐错误包数量。在包接收结束时，如果 Rx_Stat 值指示有一个对齐错误，该软件计数器增加。
	CRC 错误计数（CRCErrCnt）	接收到的 CRC 错误包数量。如果 Rx_Stat 值指示有一个 CRC 错误，该软件计数器增加。
	丢包错误计数（MissErrCnt）	由于 MAC 接收 FIFO 溢出、奇偶错误、或 Rx_En 位被清除等原因，被 MAC 控制器拒绝的有效包数量。该数字不包括被 CAM 拒绝的包数量。

[31：16]	保留	未使用

MAC 接收暂停计数寄存器（MAC Received Pause Count Register）

MAC 接收暂停计数寄存器（EPZCNT）保持 16 位接收暂停计数器的当前值。

Registers	Offset	R/W	Description	Reset Value
EPZCNT	0XA040	R	Pause count	0x00000000

表 6-2-30 MAC 接收暂停计数寄存器描述

位	位名	功能描述
[15：0]	接收暂停计数（EPZCNT）	该计数值指示由于接收来自 MAC 的暂停控制操作包而引起发送器暂停的时间单位数量。

MAC 远程暂停计数寄存器（MAC Remote Pause Count Register）

MAC 远程暂停计数寄存器（ERMPZCNT）保持 16 位远程暂停计数器的当前值。

Registers	Offset	R/W	Description	Reset Value
ERMPZCNT	0XA044	R	Remote pause count	0x00000000

表 6-2-31 MAC 接收暂停计数寄存器描述

位	位名	功能描述
[15：0]	远程暂停计数（ERMPZCNT）	该计数值指示由于发送暂停控制操作包而引起远程 MAC 暂停的时间单位数量。

MAC 发送控制帧状态寄存器（MAC Transmit Control Frame Status Register）

MAC 发送控制帧状态寄存器（ETXSTAT）是一个基于 RAM 的寄存器，提供 MAC 控制包发送到远程站的状态信息。

DMA 引擎读取该寄存器的值，并产生中断通知系统 MAC 控制包的发送已经完成。

Registers	Offset	R/W	Description	Reset Value
ETXSTAT	0X9040	R	Transmit control frame status	0x00000000

表 6-2-32 MAC 发送控制帧状态寄存器描述

位	位名	功能描述
[15：0]	发送状态值（Tx_Stat）	该 16 位的值指示 MAC 控制包发送到远程站的状态信息。该值由 DMA 引擎读取。

以太网控制器的操作（Ethernet Controller Operation）

该部分包括 S3C4510B 以太网控制器的如下内容：

- MAC 帧与包格式。
- 发送一个帧。
- 接收一个帧。
- 全双工暂停操作。
- 错误信号与网络管理。

MAC 帧与包格式（MAC Frame and Packer Formats）

MAC 发送所有域的每一个字节，除帧检测序列 FCS 外，总是最低位在前。在此，对帧和包的概念作一个说明，“包”通常是指发送或接收的所有字节序列。而“帧”仅指由站发送或接收的字节。

表 6-2-33 MAC 帧与包的格式描述

域名	域尺寸	描　　述

前导（Preamble）	7 字节	每一个前导字节的各个位为：10101010，从左到右发送。
起始帧分隔符（SFD）	1 字节	SFD 的每个位为：10101011，从左到右发送。
目的地址	6 字节	目的地址可以是单独的地址，多播（或广播）地址。
源地址	6 字节	MAC 不对源地址字节作检测，但作为一个有效的站地址，发送的第一个位（第一个字节的最低位）必须为“0“
长度或类型	2 字节	当该长度域大于 1500 字节时，MAC 将其作为类型域处理。该域小于或等于 1500 时表示数据域中逻辑链路控制（LLC）数据字节数。MAC 首先发送高字节。
逻辑链路控制（LLC）数据	46 到 1500 字节	用于逻辑链路控制的数据字节。
填充（PAD）	0 到 46 字节	如果 LCC 数据少于 46 字节，MAC 发送时填充全“0”的字节。
帧检测序列(FCS)	4 字节	FCS 域包含 16 位的错误检测码。

图 6.2.14 为一个 IEEE802.3 以太网包（帧）的域。

数据包							
由发送器添加，接收器去除		数据帧				Added by transmitter	
		数据帧				由接收器有选择的去除	
前导	SFD	目的地址	源地址	长度或类型	LLC数据	PAD	FCS
7 bit	1	6 bytes	6 bytes	2 bytes	46-1500 bytes	0-46	4 bytes

图 6.2.14　一个 IEEE802.3 以太网包（帧）的域

对标准 MAC 帧有影响的选项（Options that Affect the Standard MAC Frame）

有些选项会对标准的 MAC 帧产生影响，如下所述：

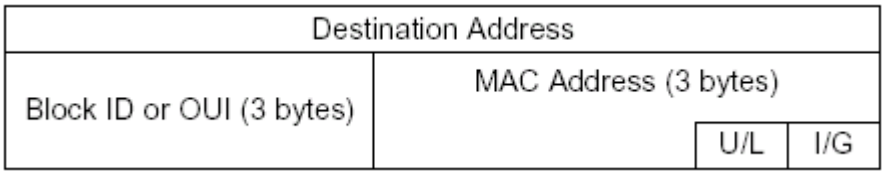
- 某些物理层会发送较长或较短的前导。
- 短包模式允许 LCC 数据域少于 46 字节，通过选项可以不要填充和支持短包的接收。
- 长包模式允许 LCC 数据域多于 1500 字节，通过选项可以支持长包接收。
- “无 CRC”模式不在数据包末尾增加 CRC 域。
- “忽略 CRC”模式允许接收的数据包 CRC 域无效。

目的地址格式（Destination Address Format）

目的地址的第 0 位为地址类型位，该位标识目的地址是独立的地址还是组地址，组地址有时称为“多播”地址，独立地址称为“单播”地址，广播地址为一种特殊的组地址，具有特殊的 16 进制格式：FF-FF-FF-FF-FF-FF。

第 1 位区别目的地址是局部管理地址还是全局管理地址，若为全局管理地址，该位的值为“0”，若为局部管理地址，该位的值为“1”，对于广播地址，该位也必须置“1”。

以下为目的地址格式：



[0]独立或组地址标识（I/G）

0 = 独立（单播）地址。

1 = 组（多播）地址。

[1]全局与局部地址标识（U/L）

0 = 全局地址。

1 = 局部地址。

特定流控目的地址（Special Flow Control Destination Address）

特定流控目的地址用于发送暂停操作包，为使 MAC 能接收到包含该特定目的地址的控制包，目的地址必须放在特定的 CAM 单元里（CAM0），同时该 CAM 单元必须使能、CAM 必须有效。

发送一个帧（Transmitting a Frame）

要发送一个帧，发送控制寄存器中的发送使能位必须置“1”，发送停止请求位必须清“0”，此外，MAC 控制寄存器中的立即停止位和停止请求位也必须清“0”，这些条件通常在 BDMA 控制器初始化完成以后设置，然后由系统使用 Tx_wr#和 Tx_EOF 信号传输字节数据到发送数据缓冲区。

发送状态机开始发送 FIFO 中的数据，并保持前 64 个字节直到本站获得网络控制权，此时，发送模块请求更多的数据并发送，直到系统输入 Tx_EOF 信号通知到达发送的数据包末尾，发送模块添加经过计算的 CRC 到数据包的末尾并发送出去，然后设置发送状态寄存器的第 0 位，表示成功发送。如果中断使能，该操作同时产生一个中断请求。

一个帧的发送操作可以分为一个 MII 接口操作和一个 BDMA/MAC 接口操作。

BDI 发送操作（BDI Transmit Operation）

BDI 发送操作是一个简单的 FIFO 机制。BDMA 引擎保存待发送的数据，当 MAC 成功取得网络控制权时，发送状态机将数据发送出去。

复位后，发送 FIFO 为空，发送模块发出 Tx_rdy 信号，发送被禁止。要使能发送，系统必须设置发送控制寄存器的发送使能位，此外，发送 FIFO 中必须有 8 个字节的数据。BDMA 引擎可以先将数据放入发送 FIFO，后使能发送位，也可以先使能发送位，后将数据放入发送 FIFO。总之，只有两个条件同时满足，才能开始发送操作。

MII 发送操作（MII Transmit Operation）

MII 发送模块由 3 个状态机组成，具体工作原理可以参考 S3C4510B 的用户手册。

接收一个帧（Receiving a Frame）

当接收模块使能时，监控从 MII（在回环模式下）或从发送模块传来的数据流。MII 提供 0 到 7 个前导字节，然后是起始帧分隔符（SFD）。接收模块首先检测前导字节，然后在前 8 个字节中寻找 SFD（10101011），如果不能检测到 SFD，就将包当做碎片丢弃。

其后是目的地址，最低位在前，当接收到一个字节，接收模块生成奇偶位并将其存储到接收 FIFO 中，同时发出 Rx_rdy 信号.接收模块然后接收其后的数据并将其存入接收 FIFO。

S3C4510 的以太网控制器是所有部件中控制最复杂的一个，关于以太网控制器的工作原理更详细的内容，可参考 S3C4510B 的用户手册。

关于以太网控制器的初始化和发送、接收编程，读者可参考已经移植到 S3C4510B 的 uClinux 源代码，该源代码有详细的描述，限于篇幅，不在此列出。源代码位于：/Linux/drivers/net/s3c4510.c。

6.2.8 Flash 存储器工作原理与编程示例

Flash 存储器的操作包括对 Flash 的擦除和烧写，由前面所介绍的 Flash 存储器的工作原理可知，对 Flash 存储器的编程与擦除是与具体的器件型号紧密相关的，由于不同厂商的 Flash 存储器在操作命令上可能会有一些细微的差别，Flash 存储器的烧写、擦除程序一般不具有通用性，针对

不同厂商、不同型号的 Flash 存储器，程序应作相应的修改。

以本系统所使用的 Flash 存储器 HY29LV160 为例，详细说明其操作命令及相关的编程方法，其他厂商或型号的 Flash 存储器的编程方法与之类似。

HY29LV160 是 HYUNDAI 公司生产的 Flash 存储器，其主要特点有：3 V 单电源供电，可使内部产生高电压进行编程和擦除操作；支持 JEDEC 单电源 Flash 存储器标准和 CFI（Common Flash Memory Interface）特性；只需向其命令寄存器写入标准的微处理器指令，具体编程、擦除操作由内部嵌入的算法实现，并且可以通过查询特定的引脚或数据线监控操作是否完成；可以对任一扇区进行读、写或擦除操作，而不影响其它部分的数据。

Flash 存储器的操作命令

向 Flash 存储器的特定寄存器写入地址和数据命令，就可对 Flash 存储器进行烧写、擦除等操作，但操作必须按照一定的顺序，否则就会导致 Flash 存储器复位而使操作命令无法完成。编程指令只能使‘1’变为‘0’，而擦除命令可使‘0’变为‘1’，因此正确的操作顺序是先擦除，后编程，当 Flash 存储器被擦除以后，读出的内容应全为 0xff。以下介绍 HY29LV160 编程命令、整片擦除命令以及对应的程序设计，关于 HY29LV160 更详细的内容可参考本书所附光盘中 HY29LV160 的用户手册。

1、 正常编程命令：

		第一步		第二步		第三步		第四步	
		地址	数据	地址	数据	地址	数据	地址	数据
正常编程	字	0x555	0xAA	0x2AA	0x55	0x555	0xA0	待编程的地址	待编程的数据
	字节	0xAAA		0x555		0xAAA			

如上表所示，向 HY29LV160 的指定地址中写入数据，可按字（16 位）或字节（8 位）操作，共需要四个总线周期，分四步完成，前两个周期是解锁周期，第三个是建立编程命令，最后一个周期完成向待编程地址写入特定的数据。

2、 整片擦除命令：

		第一步		第二步		第三步		第四步	
		地址	数据	地址	数据	地址	数据	地址	数据
整片擦除	字	0x555	0xAA	0x2AA	0x55	0x555	0x80	0x555	0xAA
	字节	0xAAA		0x555		0xAAA		0xAAA	
		第五步		第六步					
		地址	数据	地址	数据				
整片擦除	字	0x2AA	0x55	0x555	0x10				
	字节	0x555		0xAAA					

如上表所示，将 HY29LV160 整片擦除，也可按字（16 位）或字节操作，共需要六个总线周期，分六步完成，前两个是解锁周期，第三个是建立编程命令，第四、第五是解锁周期，最后一个周期是整片擦除周期。

FLASH 存储器的操作检测

当按照规定的命令序列向 Flash 存储器发出命令时，其内部的编程或擦除算法就可自动完成编程或擦除操作，但无论是编程或擦除都需要一定的操作时间，同时用户还应了解其内部的操作检测机制，以便知道操作是否完成或操作是否正确。常用检测的状态位有：跳变位（DQ6）、超时标志位（DQ5）、数据查询位（DQ7）和 Ready/Busy 引脚（RY/BY＃）。

常用的检测方法有三种：第一种是判断引脚 RY/BY＃的状态，在编程或擦除操作过程中，RY/BY＃引脚一直为低电平，操作完成后变为高电平；第二种是检测跳变位 DQ6，在编程或擦除时对任何地址进行连续的读均引起 DQ6 连续跳变，直至操作结束才停止跳变；第三种方法是使用数据线的 DQ7 和 DQ5 位，DQ7 位在编程或擦除过程中输出的数是写入该位数据的反码，当操作完成时输出才变为写入该位的数据；DQ5 的状态为“1”时表示操作超时，此时应再读一次 DQ7 的状态，若 DQ7 输出仍不是写入的数据，则操作失败，复位 Flash 存储器。其流程如图 6.2.1 所示。

对于第一种方法，需要占用系统的 I/O 口以判断引脚 RY/BY＃的状态，故较少使用，而第二种方法对 DQ6 的连续跳变检测相对较困难，因此，常用第三种方法检测 Flash 存储器的工作状态。

本例采用第三种方法编程检测 Flash 存储器的工作状态，程序流程如图 6.2.14。

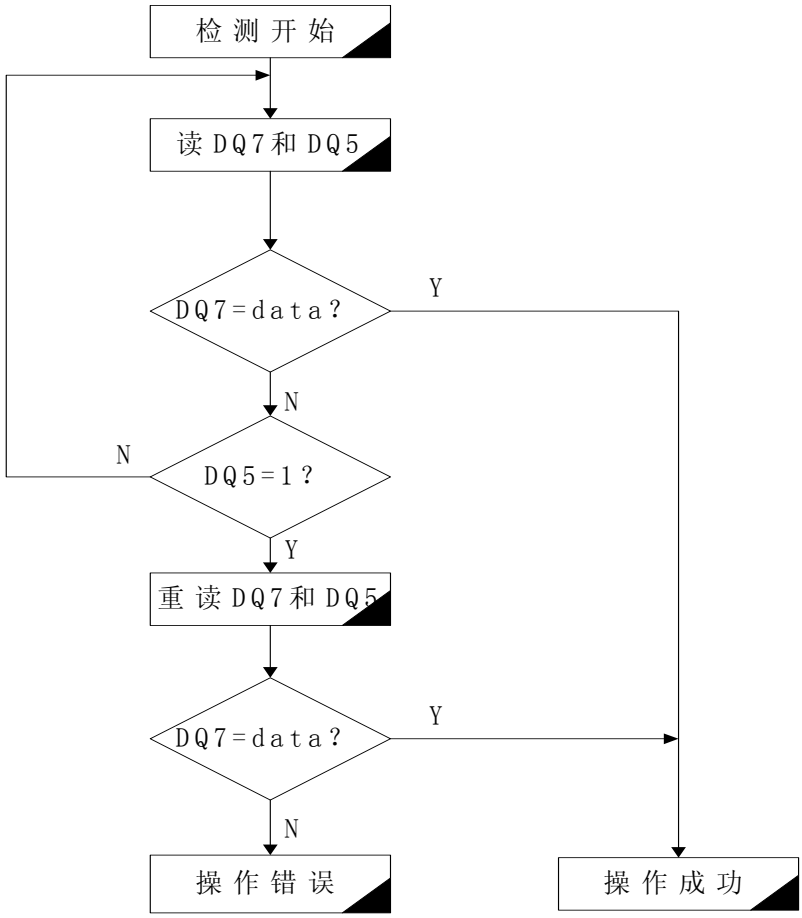


图 6.2.14　检测 Flash 存储器状态的流程图

FLASH 存储器的整片擦除示例

打开 CodeWarrior for ARM Developer Suite（或 ARM Project Manager），新建一个项目，并新建一个文件，名为 Init.s，具体内容与第一个例子相同。

保存 Init.s，并添加到新建的项目。

再新建一个文件，名为 main.c，具体内容如下：

```
/*****************************************************************
* Institute of Automation, Chinese Academy of Sciences
* File Name:    main.c
* Description:
* Author:       JuGuang.Lee
* Date:
*****************************************************************/
#define FLASH_START_ADDR          0x0000
#define FLASH _ADDR_UNLOCK1       0x555
#define FLASH _ADDR_UNLOCK2       0x2aa
#define FLASH _DATA_UNLOCK1       0xaaaa
#define FLASH _DATA_UNLOCK2       0x5555
#define FLASH _SETUP_ERASE        0x8080
#define FLASH _CHIP_ERASE         0x1010
```

```
#define UINT16 unsigned short
int FlashStatusDetect(UINT16 *ptr,  UINT16 Data, int TimeCounter);
int Main()
{
      int i;
      volatile UINT16 *flashPtr=(UINT16 *) FLASH _START_ADDR;
      //按 HY29LV160 整片擦除命令要求写入命令序列，以 16 位方式操作
      *((volatile UINT16 *) FLASH_START_ADDR+ FLASH _ADDR_UNLOCK1) = FLASH
      _DATA_UNLOCK1;
      *((volatile UINT16 *) FLASH _START_ADDR+ FLASH _ADDR_UNLOCK2) = FLASH
      _DATA_UNLOCK2;
      *((volatile UINT16 *) FLASH _START_ADDR+ FLASH _ADDR_UNLOCK1) = FLASH
      _SETUP_ERASE;
      *((volatile UINT16 *) FLASH_START_ADDR+ FLASH_ADDR_UNLOCK1) = FLASH_DATA_UNLOCK1;
      *((volatile UINT16 *) FLASH_START_ADDR+ FLASH_ADDR_UNLOCK2) = FLASH_DATA_UNLOCK2;
      *((volatile UINT16 *) FLASH_START_ADDR+ FLASH_ADDR_UNLOCK1) = FLASH _CHIP_ERASE;
      //判断擦除操作是否正确完成
      if(FlashStatusDetect((UINT16 *)flashPtr, 0xffff, 0x1000000)!=1)
            printf("ERROR!");
      return(0);
}
int FlashStatusDetect(UINT16 *ptr,  UINT16 Data, int TimeCounter)
{
      int tmp = TimeCounter;
      volatile UINT16 *p = ptr;
      UINT16 data1, data2, current_data;
      current_data = Data & 0x8080;
      while((*p& 0x8080) != current_data)
        {
        if(tmp-- <= 0)
        return 0;
        }
      return 1;
}
```

在该程序中，首先以 16 位方式操作方式，按 HY29LV160 整片擦除命令要求写入命令序列，然后判断 DQ7 的状态是否正确。为稳妥起见，在擦除完成之后还应读取 Flash 存储器的内容与 0xFFFF 相比较，看是否相等，若相等则表明擦除操作完全正确，否则表明擦除操作错误或 Flash 存储器已经损坏，读者可自行添加这段代码。

注意符号常量 FLASH_START_ADDR，程序中用其标识系统中 Flash 存储器的起始物理地址，此例假定系统中 Flash 存储器的起始物理地址为 0x0，在系统复位或上电时，位于 ROM/SRAM/FLASH Bank0 的 Flash 存储器的起始物理地址会被映射到 0x0，可用此例程进行整片擦除操作。但当 Flash 中写有应用程序或操作系统时，这些程序在启动并初始化系统后，可能会把 Flash 存储器映射到其他的地址空间，此时应修改该起始地址，否则就不能正确擦除 Flash 存储器。

保存 main.c，并添加到新建的项目，并对该项目进行编译链接，生成可执行的映象文件。

在编译链接项目文件时，将链接器程序的入口地址为 0x0040,0000。

打开 AXD Debugger（或 ARM Debugger for Windows）的命令行窗口，执行 obey 命令：

```
>obey C:\memmap.txt
```

系统中 SDARM 被映射到 0x0040,0000～（0x0140,0000-1），Flash 存储器被映射到起始地址为 0x0 处。

从 0x0040,0000 处装入生成的可执行的映象文件，并将 PC 指针寄存器修改为 0x0040,0000，运行可执行的映象文件，就可整片擦除系统中的 Flash 存储器。

FLASH 存储器的编程示例

该例程完成向 Flash 存储器的一个单元写入数据，以此为基础，读者可以编写更复杂、功能更强的 Flash 烧写工具。

打开 CodeWarrior for ARM Developer Suite（或 ARM Project Manager），新建一个项目，并新建一个文件，名为 Init.s，具体内容与第一个例子相同。

保存 Init.s，并添加到新建的项目。

再新建一个文件，名为 main.c，具体内容如下：

```
/*************************************************************
* Institute of Automation, Chinese Academy of Sciences
* File Name:    main.c
* Description:
* Author:       JuGuang.Lee
* Date:
*************************************************************/
#define FLASH_START_ADDR          0x0000
#define FLASH _ADDR_UNLOCK1       0x555
#define FLASH _ADDR_UNLOCK2       0x2aa
#define FLASH _DATA_UNLOCK1       0xaaaa
#define FLASH _DATA_UNLOCK2       0x5555
#define FLASH _SETUP_WRITE        0xa0a0
#define UINT16 unsigned short
int Main()
{
      volatile UINT16  *to_add;
      to_add= (UINT16 *)0x0;       //写入的地址
      *((volatile UINT16 *) FLASH _START_ADDR + FLASH _ADDR_UNLOCK1) = FLASH
      _DATA_UNLOCK1;
      *((volatile UINT16 *) FLASH _START_ADDR + FLASH _ADDR_UNLOCK2) = FLASH
      _DATA_UNLOCK2;
      *((volatile UINT16 *) FLASH _START_ADDR + FLASH _ADDR_UNLOCK1) = FLASH
      _SETUP_WRITE;
      *to_add = 0x1234;            //写入的数据
      return 0;
}
```

保存 main.c，并添加到新建的项目，并对该项目进行编译链接，生成可执行的映象文件。

在编译链接项目文件时，将链接器程序的入口地址为 0x0040,0000。

打开 AXD Debugger（或 ARM Debugger for Windows）的命令行窗口，执行 obey 命令：

```
>obey C:\memmap.txt
```

系统中 SDARM 被映射到 0x0040,0000～（0x0140,0000-1），Flash 存储器被映射到起始地址为 0x0 处。

从 0x0040,0000 处装入生成的可执行的映象文件，并将 PC 指针寄存器修改为 0x0040,0000，运行可执行的映象文件，程序就会将数据 0x1234 烧写入的 Flash 存储器的 0x0 地址处。

对于编写连续烧写 Flash 存储器多个存储单元的程序，只需循环执行该段代码即可，但应在对每个单元烧写命令发出后进行检测，保证前一个单元烧写结束后再进行下一个存储单元的烧写；当然也可采用延时等待的方法进行连续的烧写。

6.3 BootLoader 简介

在以上的示例中，当完成用户程序的编译并下载到目标板上运行时，总是要首先进行存储器的映射，然后通过 ADS（或 SDT）调试环境下载，显然，这个过程对普通用户来说显得特别烦琐，然而，要在裸板（没有任何程序的系统板）上调试运行程序，也只能采用这种方法。

如果能在用户设计的系统板上烧写一段 BootLoader 程序，就可以将该过程屏蔽起来，让用户通过一些简单的操作，就可完成程序的下载、调试等工作。在嵌入式系统中，BootLoader 的作用与 PC 机上的 BIOS 类似，通过 BootlLoader 可以完成对系统板上的主要部件如 CPU、SDRAM、Flash、串行口等进行初始化，也可以下载文件到系统板、对 Flash 进行擦除与编程。事实上，一个功能完善的 BootLoader 已经相当于一个微型的操作系统了。

BootLoader 作为系统复位或上电后首先运行的代码，一般应写入 Flash 存储器中并从起始物理地址 0x0 开始。BootLoader 根据实现的功能不同，其复杂程度也各不相同。一个简单的 BootLoader 程序可以仅仅完成串行口的初始化，并进行通信，而功能完善的 BootLoader 可以支持比较复杂的命令集，对系统的软硬件资源进行合理的配置与管理。因此，用户可根据自身的需求实现相应的功能。

关于 BootLoader 的具体编写过程，可参考其他的相关资料，在此不作详述。

6.4 本章小节

本章主要介绍了在前一章所设计的最小系统硬件平台上，进行简单程序设计的基本步骤，同时也介绍了 S3C4510B 相关硬件模块的工作原理，通过对本章的阅读，希望读者能掌握基于 S3C4510B 的嵌入式系统的基本编程方法。

本章的难点、但也是读者必须掌握的问题是系统的存储器映射，S3C4510B 任一时刻能寻址的地址空间为 64MB，而系统配置的实际物理存储器一般没有这么多，因此，将系统的实际存储器安排（映射）在 64MB 地址空间的哪个位置，是程序设计与运行时必须清楚的。

第 7 章　嵌入式 uClinux 及其应用开发

本章从构建一个针对 S3C4510B 硬件平台的嵌入式 uClinux 操作系统和在其上进行应用程序的开发入手，逐步讲述如何在 Linux 环境下编写用户应用程序的方法和步骤，并为熟悉 Windows 操作系统的用户介绍在这种平台之上，使用何种工具编写和编译自己的应用。通过本章的学习，读者可以对嵌入式 uClinux 有一定的了解，并且掌握在 Linux 和 Windows 下嵌入式系统应用开发的基本方法。

本章主要内容有：

- 嵌入式 uClinux 系统概况
- 开发工具 GNU 的使用
- 建立 uClinux 开发环境
- 在 uClinux 下开发应用程序

7.1　嵌入式 uClinux 系统概况

在 PC 机上开发应用程序的用户都会有这样的感觉，PC 机有完善的操作系统并提供应用程序接口（API），开发好的应用程序可以直接在操作系统上运行。虽然嵌入式系统的应用程序完全可以在裸板上运行，但为了使系统具有任务管理、定时器管理、存储器管理、资源管理、事件管理、系统管理、消息管理、队列管理和中断处理的能力，提供多任务处理，更好的分配系统资源的功能，用户就需要针对自己的硬件平台和实际应用选择适当的嵌入式操作系统（Embedded Operating System，以下简称 EOS）。本节将结合本书所谈到的硬件平台 S3C4510B，介绍一种针对不带 MMU 的 ARM 微处理器的嵌入式操作系统 uClinux。

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

为了降低硬件成本及运行功耗，有一类 CPU 在设计中取消了内存管理单元（Memory Management Unit，以下简称 MMU）功能模块。最初，运行于这类没有 MMU 的 CPU 之上的都是一些很简单的单任务操作系统，或者更简单的控制程序，甚至根本就没有操作系统而直接运行应用程序。在这种情况下，系统无法运行复杂的应用程序，或者效率很低，而且，所有的应用程序需要重写，并要求程序员十分了解硬件特性。这些都阻碍了应用于这类 CPU 之上的嵌入式产品开发的速度。

然而，随着 uClinux 的诞生，这一切都改变了。

uClinux 从 Linux 2.0/2.4 内核派生而来，沿袭了主流 Linux 的绝大部分特性。它是专门针对没有 MMU 的 CPU，并且为嵌入式系统做了许多小型化的工作。适用于没有虚拟内存或内存管理单元（MMU）的处理器，例如 ARM7TDMI。它通常用于具有很少内存或 Flash 的嵌入式系统。uClinux 是为了支持没有 MMU 的处理器而对标准 Linux 作出的修正。它保留了操作系统的所有特性，为硬件平台更好的运行各种程序提供了保证。在 GNU 通用公共许可证（GNU GPL）的保证下，运行 uClinux 操作系统的用户可以使用几乎所有的 Linux API 函数，不会因为没有 MMU 而受到影响。由于 uClinux 在标准的 Linux 基础上进行了适当的裁剪和优化，形成了一个高度优化的、代码紧凑的嵌入式 Linux，虽然它的体积很小，uClinux 仍然保留了 Linux 的大多数的优点：稳定、良好的移植性、优秀的网络功能、完备的对各种文件系统的支持、以及标准丰富的 API 等。图 7.1 为 uClinux 的基本架构。

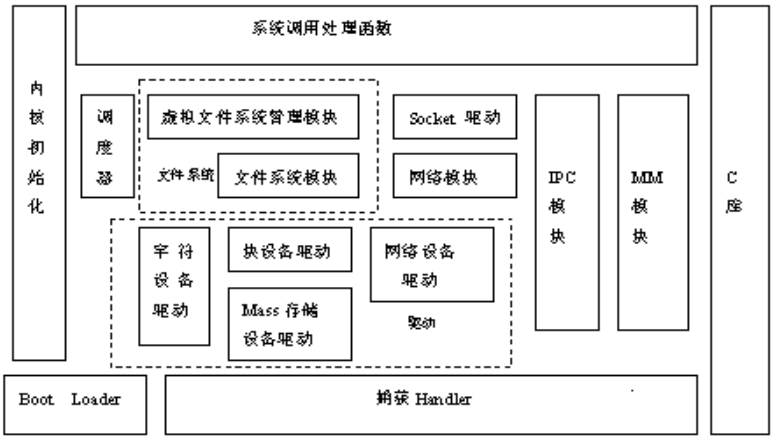


图 7.1 uClinux 的基本架构

Boot Loader：负责 Linux 内核的启动，它用于初始化系统资源，包括 SDRAM。这部分代码用于建立 Linux 内核运行环境和从 Flash 中装载初始化 ramdisk。

内核初始化：Linux 内核的入口点是 start_kernel（）函数。它初始化内核的其他部分，包括捕获，IRQ 通道，调度，设备驱动，标定延迟循环，最重要的是能够 fork “init”进程，以启动整个多任务环境。

系统调用函数/捕获函数：在执行完“init”程序后，内核对程序流不再有直接的控制权，此后，它的作用仅仅是处理异步事件(例如硬件中断)和为系统调用提供进程。

设备驱动：设备驱动占据了 Linux 内核很大部分。同其他操作系统一样，设备驱动为它们所控制的硬件设备和操作系统提供接口。

文件系统：Linux 最重要的特性之一就是对多种文件系统的支持。这种特性使得 Linux 很容易地同其他操作系统共存。文件系统的概念使得用户能够查看存储设备上的文件和路径而无须考虑实际物理设备的文件系统类型。Linux 透明的支持许多不同的文件系统，将各种安装的文件和文件系统以一个完整的虚拟文件系统的形式呈现给用户。

下面介绍一些和 uClinux 相关的知识。

1、 MMU（内存管理单元）和 VM（虚拟内存）

许多嵌入式微处理器都由于没有 MMU 而不支持虚拟内存。没有内存管理单元所带来的好处是简化了芯片设计，降低了产品成本。由于大多数的嵌入式设备没有磁盘或者只有很有限的内存空间，所以无需复杂的内存管理机制。但是由于没有 MMU 的管理，操作系统对内存空间是没有保护的，所有程序访问的地址都是实际物理地址。但从嵌入式系统一般都是实现某种特定功能的角度考虑，对于内存管理的要求完全可以由程序开发人员考虑。

2、实时性的支持

uClinux 本身并不支持实时性，目前存在两种不同的方案提供 uClinux 对实时性的支持，它们分别是 RTLinux(RTL)和 RTAI(Real Time Application Interface)。有了这两种方案，uClinux 可以应用到对实时性要求较高的场合。

3、 平台支持

开发 uClinux 的工具链：

开发 uClinux 通常用标准的 GNU 工具链。经过修改的工具链支持一些高级特性，比如 XIP(Execute-In-Place)技术，共享库支持等。

uClinux 所适用的微控制器：

uClinux 适用于摩托罗拉的 ColdFire/Dragonball，ARM 系列(例如 Atmel, TI, Samsung 等生产的芯片)，Intel i960, Sparc (例如无 MMU 的 LEON), NEC v850，甚至是开放的可综合(到 CLPD 内)的 CPU 核，比如 OPENcore。

4、与标准 Linux 的兼容性

uClinux 除了不能实现 fork()而是使用 vfork()外，其余 uClinux 的 API 函数与标准 Linux 的完全相同。这并不是意味着 uClinux 不能实现多进程，实际上 uClinux 多进程管理是通过 vfork()来实现的，或者是子进程代替父进程执行，直到子进程调用 exit()函数退出，或者是子进程调用 exec()函数执行一个新的进程。大多数标准的 Linux 应用程序在从 Linux 操作系统移植到 uClinux 系统时，几乎不用做什么大的改动，就可以完全达到对一个嵌入式应用程序的要求(例如合理的资源使用)。uClibc 对 libc(可用于标准 Linux 的函数库)做了修改为 uClinux 提供了更为精简的应用程序库。

5、 网络的支持

uClinux 带有一个完整的 TCP/IP 协议，同时它还支持许多其他网络协议。uClinux 对于嵌入式系统来说是一个网络完备的操作系统。

6、应用领域

uClinux 广泛应用于嵌入式系统中，例如 VPN 路由器/防火墙，家用操作终端，协议转换器，IP 电话，工业控制器，Internet 摄像机，PDA 设备等。

在对 uClinux 有了一个初步认识之后，有必要向读者介绍在嵌入式开发中最为普遍使用的编译工具 GNU GCC。

7.2 开发工具 GNU 的使用

GCC(gcc)的不断发展完善使许多商业编译器都相形见绌，GCC 由 GNU 创始人 Richard Stallman 首创，是 GNU 的标志产品，由于 UNIX 平台的高度可移植性，GCC 几乎在各种常见的 UNIX 平台上都有，即使是 Win32/DOS 也有 GCC 的移植。 比如说 SUN 的 Solaris 操作系统配置的编译器就是 GNU 的 GCC。

GNU 软件包括 C 编译器 GCC，C++编译器 G++，汇编器 AS，链接器 LD，二进制转换工具(OBJCOPY，OBJDUMP)，调试工具(GDB，GDBSERVER，KGDB) 和基于不同硬件平台的开发库。在 GNU GCC 支持下用户可以使用流行的 C/C++语言开发应用程序，满足生成高效率运行代码、易掌握的编程语言的用户需求。

这些工具都是按 GPL 版权声明发布，任何人可以从网上获取全部的源代码，无需使用任何费用。关于 GNU 和公共许可证协议的详细资料，读者可以参看 GNU 网站的介绍，http://www.gnu.org/home.html。

GNU 开发工具都是采用命令行的方式，用户掌握起来相对比较困难，不如基于 Windows 系统的开发工具好用，但是 GNU 工具的复杂性是由于它更贴近编译器和操作系统的底层，并提供了更大的灵活性。一旦学习和掌握了相关工具后，就了解了系统设计的基础知识。

运行于 Linux 操作系统下的自由软件 GNU gcc 编译器，不仅可以编译 Linux 操作系统下运行的应用程序，还可以编译 Linux 内核本身，甚至可以作交叉编译，编译运行于其它 CPU 上的程序。所以，在进行嵌入式系统应用程序开发时，这些工具得到了日益广泛的应用。

7.2.1 GCC 编译器

GCC 是 GNU 组织的免费 C 编译器，Linux 的很多发布缺省安装的就是这种。很多流行的自由软件源代码基本都能在 GCC 编译器下编译运行。 所以掌握 GCC 编译器的使用无论是对于编译系统内核还是自己的应用程序都是大有好处的。

下面通过一个具体的例子，学习如何使用 GCC 编译器。

在 Linux 操作系统中，对一个用标准 C 语言写的源程序进行编译，要使用 GNU 的 gcc 编译器。

例如下面一个非常简单的 Hello 源程序(hello.c)：

```
/*********************************************************
 * Institute of Automation, Chinese Academy of Sciences
```

```
    * File Name:     hello.c
    * Description: introduce how to compile a source file with gcc
    * Author:        Xueyuan Nie
    * Date:
    ********************************************************/


void main()
{
  printf("Hello the world\n");
}
```

要编译这个程序，我们只要在 Linux 的 bash 提示符下输入命令：

```
$ gcc -o hello hello.c
```

gcc 编译器就会生成一个 hello 的可执行文件。在 hello.c 的当前目录下执行./hello 就可以看到程序的输出结果，在屏幕上打印出“Hello the world”的字符串来。

命令行中 gcc 表示是用 gcc 来编译源程序；

-o outputfilename 选项表示要求编译器生成文件名为 outputfilename 的可执行文件，如果不指定-o 选项，则缺省文件名是 a.out。在这里生成指定文件名为 hello 的可执行文件，而 hello.c 是我们的源程序文件。

gcc 是一个多目标的工具。gcc 最基本的用法是：

gcc [options] file... ，

其中的 option 是以-开始的各种选项，file 是相关的文件名。在使用 gcc 的时候，必须要给出必要的选项和文件名。gcc 的整个编译过程，实质上是分四步进行的，每一步完成一个特定的工作，这四步分别是：预处理，编译，汇编和链接。它具体完成哪一步，是由 gcc 后面的开关选项和文件类型决定的。

清楚的区别编译和连接是很重要的。编译器使用源文件编译产生某种形式的目标文件(object files)。在这个过程中，外部的符号引用并没有被解释或替换，然后我们使用链接器来链接这些目标文件和一些标准的头文件，最后生成一个可执行文件。 在这个过程中，一个目标文件中对别的文件中的符号的引用被解释，并报告不能被解释的引用，一般是以错误信息的形式报告出来。

gcc 编译器有许多选项，但对于普通用户来说只要知道其中常用的几个就够了。在这里为读者列出几个最常用的选项：

-o 选项表示要求编译器生成指定文件名的可执行文件；

-c 选项表示只要求编译器进行编译，而不要进行链接，生成以源文件的文件名命名但把其后缀由.c 或.cc 变成.o 的目标文件；

-g 选项要求编译器在编译的时候提供以后对程序进行调试的信息；

-E 选项表示编译器对源文件只进行预处理就停止，而不做编译，汇编和链接；

-S 选项表示编译器只进行编译，而不做汇编和链接；

-O 选项是编译器对程序提供的编译优化选项，在编译的时候使用该选项，可以使生成的执行文件的执行效率提高；

-Wall 选项指定产生全部的警告信息。

如果你的源代码中包含有某些函数，则在编译的时候要链接确定的库，比如代码中包含了某些数学函数，在 Linux 下，为了使用数学函数，必须和数学库链接，为此要加入-lm 选项。也许有读者会问，前面那个例子使用 printf 函数的时候为何没有链接库呢？在 gcc 中对于一些常用函数的实现，gcc 编译器会自动去链接一些常用库，这样用户就没有必要自己去指定了。有时候在编译程序的时候还要指定库的路径，这个时候要用到编译器的-L 选项指定路径。比如说我们有一个库在/home/hoyt/mylib 下，这样我们编译的时候还要加上 -L/home/hoyt/mylib。对于一些标准库来说，没有必要指出路径。只要它们在起缺省库的路径下就可以了，gcc 在链接的时候会自动找到那些库的。

GNU 编译器生成的目标文件缺省格式为 elf(executive linked file)格式，这是 Linux 系统所采用的可执行链接文件的通用文件格式。elf 格式由若干段(section)组成，如果没有特别指明，由标准 c 源代码生成的目标文件中包含以下段：.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，字符串等，.bss(未初始化数据段) 包含未初始化的变量和数组等。

读者若想知道更多的选项及其用法，可以查看 gcc 的帮助文档，那里有许多对其它选项的详细说明。

当改变了源文件 hello.c 后，需要重新编译它：

$gcc -c hello.c

然后重新链接生成：

$gcc –o hello.o

对于本例，因为只含有一个源文件，所以当改动了源码后，进行重新的编译链接的过程显得并不是太繁琐，但是，如果在一个工程中包含了若干的源码文件，而这些源码文件中的某个或某几个又被其他源码文件包含，那么，如果一个文件被改动，则包含它的那些源文件都要进行重新编译链接，工作量是可想而知的。 幸运的是，GNU 提供了使这个步骤变得简单的工具，就是下面要介绍给大家的 GNU Make 工具。

7.2.2 GNU Make

make 是负责从项目的源代码中生成最终可执行文件和其他非源代码文件的工具。 make 命令本身可带有四种参数：标志、宏定义、描述文件名和目标文件名。

其标准形式为：

make [flags] [macro definitions] [targets]

Unix 系统下标志位 flags 选项及其含义为：

-f file 指定 file 文件为描述文件，如果 file 参数为 '-' 符，那么描述文件指向标准输入。如果没有 '-f' 参数，则系统将默认当前目录下名为 makefile 或者名为 Makefile 的文件为描述文件。在 Linux 中， GNU make 工具在当前工作目录中按照 GNUmakefile、makefile、Makefile 的顺序搜索 makefile 文件。

-i 忽略命令执行返回的出错信息。

-s 沉默模式，在执行之前不输出相应的命令行信息。

-r 禁止使用隐含规则。

-n 非执行模式，输出所有执行命令，但并不执行。

-t 更新目标文件。

-q make 操作将根据目标文件是否已经更新返回"0"或非"0"的状态信息。

-p 输出所有宏定义和目标文件描述。

-d Debug 模式，输出有关文件和检测时间的详细信息。

Linux 下 make 标志位的常用选项与 Unix 系统中稍有不同，下面只列出了不同部分：

-c dir 在读取 makefile 之前改变到指定的目录 dir。

-I dir 当包含其他 makefile 文件时，利用该选项指定搜索目录。

-h help 文挡，显示所有的 make 选项。

-w 在处理 makefile 之前和之后，都显示工作目录。

通过命令行参数中的 target ，可指定 make 要编译的目标，并且允许同时定义编译多个目标，操作时按照从左向右的顺序依次编译 target 选项中指定的目标文件。如果命令行中没有指定目标，则系统默认 target 指向描述文件中第一个目标文件。

make 如何实现对源代码的操作是通过一个被称之为 makefile 的文件来完成的，在下面的小节里，主要向读者介绍一下 makefile 的相关知识。

7.2.2.1 makefile 基本结构

GNU Make 的主要工作是读一个文本文件 makefile。makefile 是用 bash 语言写的，bash 语言是很像 BASIC 语言的一种命令解释语言。这个文件里主要描述了有关哪些目标文件是从哪些依赖文件中产生的，是用何种命令来进行这个产生过程的。有了这些信息， make 会检查磁盘的文件，如果目标文件的日期(即该文件生成或最后修改的日期)至少比它的一个依赖文件日期早的话，make 就会执行相应的命令，以更新目标文件。

makefile 一般被称为“makefile”或“Makefile”。还可以在 make 的命令行中指定别的文件名。如果没有特别指定的话，make 就会寻找“makefile”或“Makefile”，所以为了简单起见，建议读者使用这两名字。如果要使用其他文件作为 makefile，则可利用类似下面的 make 命令选项指定 makefile 文件：

$ make -f makefilename

一个 makefile 主要含有一系列的规则，如下：

目标文件名 : 依赖文件名

(tab 键) 命令

第一行称之为规则，第二行是执行规则的命令，必须要以 tab 键开始。

下面举一个简单的 makefile 的例子。

```
executable : main.o io.o
    gcc main.o  io.o  -o executable
main.o : main.c
    gcc -Wall -O -g  -c main.c -o main.o
io.o : io.c
    gcc -Wall -O -g  -c io.c -o io.o
```

这是一个最简单的 makefile，make 从第一条规则开始，executable 是 makefile 最终要生成的目标文件。给出的规则说明 executable 依赖于两个目标文件 main.o 和 io.o，只要 executable 比它依赖的文件中的任何一个旧的话，下一行的命令就会被执行。但是，在检查文件 main.o 和 io.o 的日期之前，它会往下查找那些把 main.o 或 io.o 做为目标文件的规则。make 先找到了关于 main.o 的规则，该目标文件的依赖文件是 main.c。makefile 后面的文件中再也找不到生成这个依赖文件的规则了。此时，make 开始检查磁盘上这个依赖文件的日期，如果这个文件的日期比 main.o 日期新的话，那么这个规则下面的命令 gcc -c main.c –o main.o 就会执行，以更新文件 main.o 。同样 make 对文件 io.o 做类似的检查，它的依赖文件是 io.c，对 io.o 的处理和 main.o 类似。

现在， 再回到第一个规则处，如果刚才两个规则中的任何一个被执行，最终的目标文件 executable 都需要重建(因为 executable 所依赖的其中一个 .o 文件就会比它新)，因此链接命令就会被执行。

有了 makefile，对任何一个源文件进行修改后，所有依赖于该文件的目标文件都会被重新编译(因为.o 文件依赖于.c 文件)，进而最终可执行文件会被重新链接(因为它所依赖的.o 文件被改变了)，再也不用手工去一个个修改了。

7.2.2.2 编写 make

1、Makefile 宏定义

makefile 里的宏是大小写敏感的，一般都使用大写字母。它们几乎可以从任何地方被引用，可以代表很多类型，例如可以存储文件名列表，存储可执行文件名和编译器标志等。

要定义一个宏，在 makefile 中，任意一行的开始写下该宏名，后面跟一个等号，等号后面是要设定的这个宏的值。如果以后要引用到该宏时，使用 $ (宏名)，或者是${宏名}，注意宏名一定要写在圆或花括号之内。把上一小节所举的例子，用引入宏名的方法，可以写成下面的形式：

```
OBJS = main.o io.o
CC = gcc
```

```
CFLAGS = -Wall -O -g

executable: $(OBJS)
    $(CC) $(OBJS) -o executable

main.o : main.c
    $(CC) $(CFLAGS) -c main.c -o main.o

io.o : io.c
$(CC) $(CFLAGS) -c io.c -o io.o
```

在这个 makefile 中引入了三个宏定义，所以如果当这些宏中的某些值发生变化时，开发者只需在要修改的宏处，将其宏值修改为要求的值即可，makefile 中用到这些宏的地方会自动变化。在 make 中还有一些已经定义好的内部变量，有几个较常用的变量是$@， $< ，$?，$*, $^ (注意：这些变量不需要括号括住)。

$@ 扩展为当前规则的目标文件名；

$< 扩展为当前规则依赖文件列表中的第一个依赖文件；

$? 扩展为所有的修改日期比当前规则的目标文件的创建日期更晚的依赖文件，该值只有在使用显式规则时才会被使用；

$* 扩展成当前规则中目标文件和依赖文件共享的文件名，不含扩展名；

$^ 扩展为整个依赖文件的列表(除掉了所有重复的文件名)。

利用这些变量，可以把上面的 makefile 写成：

```
OBJS = main.o io.o
CC = gcc
CFLAGS = -Wall -O -g

executable: $(OBJS)
    $(CC) $^ -o $@

main.o : main.c
    $(CC) $(CFLAGS) –c $< -o $@

io.o : io.c
   $(CC) $(CFLAGS) -c  $< -o $@
```

可以将宏变量应用到其他许多地方，尤其是当把它们和函数混合使用的时候，正确使用宏，会给开发者带来极大的便利。

2、隐含规则

请注意，在上面的例子里，几个产生 .o 文件的命令都是以.c 文件作为依赖文件产生 .o 目标(obj)文件，这是一个标准的生成目标文件的步骤。如果把生成 main.o 和 io.o 的规则从 makefile 中删除，make 会查找它的隐含规则，然后会找到一个适当的命令去执行。实际上 make 已经知道该如何生成这些目标文件，它使用变量 CC 做为编译器，并且传递宏 CFLAGS 给 C 编译器(CXXFLAGS 用于 C++ 编译器)，CPPFLAGS(C 预处理选项)， TARGET_ARCH (就目前例子而言，还不用考虑这个宏)，然后它加入开关选项 -c ，后面跟预定义宏 $<(第一个依赖文件名)，最后是开关项 -o，后跟预定义宏$@ (目标文件名)。一个 C 编译的具体命令将 会是：

$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c $< -o $@

在 make 工具中所包含的这些内置的或隐含的规则，定义了如何从不同的依赖文件建立特定类型的目标。Unix 系统通常支持一种基于文件扩展名即文件名后缀的隐含规则。这种后缀规则定义了如何将一个具有特定文件名后缀的文件(例如.c 文件)，转换成为具有另一种文件名后缀的文件(例如.o 文件)：

系统中默认的常用文件扩展名及其含义为：

.o 目标文件

.c C 源文件

.f FORTRAN 源文件

.s 汇编源文件

.y Yacc-C 源语法

.l Lex 源语法

而 GNU make 除了支持后缀规则外还支持另一种类型的隐含规则即模式规则。这种规则更加通用，因为可以利用模式规则定义更加复杂的依赖性规则。同时可用来定义目标和依赖文件之间的关系，例如下面的模式规则定义了如何将任意一个.c 文件转换为文件名相同的.o 文件：

```
%.o : %.c
$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<
```

3、伪目标

如果需要最终产生两个和更多的可执行文件，但这些文件是相互独立的，也就是说任何一个目标文件的重建，不会影响其他目标文件。此时，可以通过使用所谓的伪目标来达到这一目的。一个伪目标和一个真正的目标文件的唯一区别在于，这个目标文件本身并不存在。因此， make 总是会假设它需要被生成，当 make 把该伪目标文件的所有依赖文件都更新后，就会执行它的规则里的命令行。

举一个简单的例子，如果 makefile 开始处输入

all : executable1 executable2

这里 executable1 和 executable2 是最终希望生成的两个可执行文件。 make 把这个 'all' 做为它的主要目标，每次执行时都会尝试把 'all' 更新。但是，由于这行规则里并没有命令来作用在一个叫 'all' 的实际文件上(事实上， all 也不会实际生成)，所以这个规则并不真的改变 'all' 的状态。可既然这个文件并不存在，所以 make 会尝试更新 all 规则，因此就检查它的依赖文件 executable1, exectable2 是否需要更新，如果需要，就把它们更新，从而达到生成两个目标文件的目的。 伪目标在 makefile 中广泛使用。

4、函数

makefile 里的函数跟它的宏很相似，在使用的时候，用一个 $ 符号开始后跟圆括号，在圆括号内包含函数名，空格后跟一系列由逗号分隔的参数。例如，在 GNU Make 里有一个名为 'wildcard' 的函 数，它只有一个参数，功能是展开成一列所有符合由其参数描述的文件名，文件间以空格间隔。可以像下面所示使用这个命令：

SOURCES = $(wildcard *.c)

这样会产生一个所有以 '.c' 结尾的文件的列表，然后存入变量 SOURCES 里。当然你不需要一定要把结果存入一个变量。

另一个有用的函数是 patsubst (patten substitude, 匹配替换的缩写) 函数。它需要 3 个参数：第一个是一个需要匹配的模式，第二个表示用什么来替换它，第三个是一个需要被处理的由空格分隔的字列。例如，处理那个经过上面定义后的变量，

OBJS = $(patsubst %.c,%.o,$(SOURCES))

这个语句将处理所有在 SOURCES 宏中的文件名后缀是 '.c'的文件 ，用 '.o' 把 '.c' 取代。注意这里的 % 符号是通配符，匹配一个或多个字符，它每次所匹配的字符串叫做一个‘柄’(stem) 。在第二个参数里，% 被解释成用第一参数所匹配的那个柄。

感兴趣的读者如果需要更进一步的了解，请参考 GNU Make 手册。

7.2.2.3 makefile 的一个具体例子

在这里给读者举一个简单的 makefile 的例子，通过对这个 makefile 的讲解，来巩固前面介绍的相关知识。

```
INCLUDES =-I/home/nie/mysrc/include  \
```

```
            -I/home/nie/mysrc/extern/include \
            -I/home/nie/mysrc/src \
            -I/home/nie/mysrc/libsrc \
            -I. \
            -I..

EXT_CC_OPTS = -DEXT_MODE
CPP_REQ_DEFINES = -DMODEL=tune1 -DRT -DNUMST=2 \
                  -DTID01EQ=1 -DNCSTATES=0  \
                  -DMT=0 -DHAVESTDIO
RTM_CC_OPTS = -DUSE_RTMODEL
CFLAGS = -O -g
CFLAGS += $(CPP_REQ_DEFINES)
CFLAGS += $(EXT_CC_OPTS)
CFLAGS +=$(RTM_CC_OPTS)
SRCS  = tune1.c   rt_sim.c rt_nonfinite.c  grt_main.c rt_logging.c \
          ext_svr.c updown.c ext_svr_transport.c ext_work.c
OBJS  = $(SRCS:.c=.o)
RM    =  rm –f
CC    = gcc
LD    = gcc
all: tune1
%.o : %.c
$(CC) -c -o $@ $(CFLAGS)  $(INCLUDES)   $<
tune1 : $(OBJS)
$(LD) -o  $@  $(OBJS)  -lm
clean :
  $(RM)  $(OBJS)
```

在这个 makefile 中首先定义了十个宏：

' INCLUDES =-I …'(省略号代表-I 后面的内容)，'-I dirname' 表示将 dirname 所指的目录加入到程序头文件目录列表中去，是在进行预处理过程中使用的参数；

' EXT_CC_OPTS = -DEXT_MODE ' 表示在程序中定义了宏 EXT_MODE，等价于在源代码写入语句' #define EXT_MODE ' ；

接下来的两个宏定义 CPP_REQ_DEFINES 和 RTM_CC_OPTS 起到和 EXT_CC_OPTS 类似的作用；

'CFLAGS =-O -g '是编译器的编译选项，表示在编译的过程中对代码进行基本优化，并产生能被 GNU 调试器(如 gdb)使用的调试信息；

'CFLAGS += ' 表示对这个宏定义在原来的基础上增加新的内容；

' SRCS = …'代表了所有要编译的源代码文件列表；

' OBJS = $(SRCS:.c=.o)'表示把宏 SRC 所代表的所有以.c 结尾的文件名用.o 结尾的文件名替换，即表示各个源文件所对应的目标文件名；

' RM = rm –f ' 表示删除命令，-f 是强制删除选项，使用该符号，在对文件进行删除时，没有提示；

' CC = gcc '表示编译器是用 gcc；

' LD= gcc' 表示链接命令是用 gcc；

all 和 clean 是两个伪目标，在使用 make 命令的时候，如果不指明目标文件名，则是以在 makefile 中出现的第一个目标作为最终目标，所以如果键入命令 make，则伪目标 all 被作为最终的目标而执行，由于这个文件并不存在，所以 make 会尝试更新 all 规则，因此就检查它的依赖文件 tune1 是否需要更新，如果需要，就把它更新，这样伪目标下面的两条规则就会被执行，从而生成可执行文

件 tune1。如果要执行删除命令，只需要键入命令 make clean，就会把所有以.o 结尾的中间文件删除。

另外，请读者注意在本 makefile 的例子中多次用到'\'，该符号用于在 makefile 中，如果一条语句过长时，可以用'\'放在这条语句的右边界，通过回车换行，使下面新一行的语句成为该语句的续行。

在 makefile 文件中，用符号'#'作为注释行语句的开始，以增强 makefile 文件的可读性。

本例假设 makefile 文件名为 makefile，当然也可按照个人的喜好取其他文件名，如果文件名不是 makefile，Makefile 的话，在用 make 命令是，请使用 make –f makefilename。

到此，希望读者能够掌握 make 和 makefile 的基本使用。

7.2.3 使用 GDB 调试程序

无论是多么资深的程序员在编写的程序时，都不大可能一次性就会成功，在程序运行时，会出现许许多多意想不到的错误，一味地只是查看程序用处不大，最有效的方法通过一些手段进入到程序内部进行调试。通常在调试程序的时候如果能够得到以下一些信息，对于开发者找到错误所在是很有帮助的。

1. 程序是运行到哪个语句或者表达式就发生了错误？
2. 如果错误是在执行一个函数的时候出现的，那么是程序的哪一行包含了这个函数的调用语句，在调用该函数的时候传递的实参是什么？
3. 在程序执行到某处时，所关心的某一个变量值为多少？
4. 某个表达式最终运行的结果为何值？

调试器(更准确地说应该称为符号调试器)能够完成上述目标。它是一个能够运行其他程序的应用程序，它和普通意义上的程序的唯一不同之处在于，调试器能够进入到程序源码中，允许开发者进行逐行单步运行，了解程序代码执行顺序，和每条语句执行的结果，可以在程序运行的同时，查看甚至是改变任一变量值。在程序运行出错时，它为程序开发者提供程序运行时的详细细节，从而找到出错的原因。在 Linux 系统中，最常用到的就是 GDB(GNU Degugger)。GDB 是 GNU 自带的调试工具。

7.2.3.1 GDB 常用命令

要想使用 gdb，必须在对源码进行编译的时候，使用-g 编译选项开关，来通知编译器，开发者希望进行程序调试。用了-g 选项后，程序在编译的时候就会包含调试信息，这些调试信息存在目标文件中，它描述了每个函数或变量的数据类型以及源码行号和可执行代码地址间对应关系，gdb 正是通过这些信息使源码和机器码相关联的，它实现了源码级的调试。

为了使用 gdb 调试，只需要在命令行中输入 gdb filename(filename 是用 gcc 编译生成的最终可执行文件名)，该语句启动与调试器的文本接口。就在上一小节中所举 makefile 例子来说，就是键入 gdb tune1，则在屏幕上会出现

```
[nie@uClinux mysrc]$ gdb tune1
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

gdb 虽然运行起来，但是可执行程序 tune1 并没有运行，此时在 gdb 提示符下直接键入 run 命令即可，如果可执行程序在运行的时候需要输入命令行参数，则在 gdb 提示符下可以这样键入命令：run command-line-arguments ，就如同是输入命令：tune1 command-line-arguments 一样，启动了可执

行程序的运行。

有时候，我们希望能够断点调试程序，让程序执行到代码某处时停止继续执行下去，此时可以使用命令 break，该命令的格式为 break place，这里 place 可以是程序代码的行号，某函数名，甚至可以是用 break main ，让程序断点设置在代码一开始执行的地方，比如对于上面举的可执行文件名为 tune1 的例子，它调用了一个函数名为 rtExtModeCheckInit 的子函数，如果想让程序执行到该函数处停止，可以在 gdb 提示符下输入：break rtExtModeCheckInit，此时屏幕上出现下列信息：Breakpoint 1 at 0x8049a28: file grt_main.c, line 604.。当然，也可以使用行号设置中断位置，上面设置中断的语句可以等价为 break 604，可以在屏幕上看到相同的效果。

当设置了断点后，程序会运行到断点处停下来，此时从屏幕上可以得到类似下面的信息：

Breakpoint 1, main (argc=4, argv=0xbffffb84) at grt_main.c:604

604 rtExtModeCheckInit();

(gdb)

当想将某个断点除去，可以在 gdb 提示符下输入命令：delete N，这里 N 表示第几个中断，第一个设置的中断序号为 1，第二个设置的序号为 2，依次类推。如果 delete 后不跟任何序号，在表示把设置的所有断点都删除。如果想查看目前设置断点的情况，可以使用命令 info break，屏幕会显示出每一个设置的断点信息。

在 gdb 提示符下使用 help 命令，会给出有关 gdb 命令的一个简短描述和命令分类。

如果开发者想进入到程序内部进行单步调试，gdb 提供两种命令供选择，step 和 next 命令，两者的区别在于 step 执行每一条语句，如果遇到函数调用，会跳转到到该函数定义的开始行去执行，而 next 则不进入到函数内部，它把函数调用语句当作普通一条语句执行完成。continue 命令是继续运行程序，直到遇到下一个断点或程序结束。

有时候使用者仅仅是在 linux 的 bash 提示符下输入命令 gdb 后，启动了 gdb 而已，此时，如果要加载可执行文件，需要在 gdb 提示符下键入命令：file filename(filename 为可执行文件名)，注意是可执行文件的名字而不是源文件名。

当在调试过程中，想查看一个变量值的时候，可以在 gdb 环境下输入命令：watch variablename ，这里的 variablename 是你想观察的变量名。

还有一个可以显示表达式值的命令 print，其使用规则为 print expressionname，其中 expressionname 为要显示的表达式名。

7.2.3.2 GDB 具体调试实例

下面通过使用一个简单的程序使读者进一步熟悉用 gdb 的调试方法。

源程序名为 example1.c，代码如下：

```
/*****************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name:     example.c
 * Description:  introduce how to use gdb
 * Author:        Xueyuan Nie
 * Date:
 *****************************************************/
#include <stdio.h>
static void display(int i, int *ptr);
int main(void)
{
      int x = 5;
      int *xptr = &x;
      printf("In main():\n");
      printf("   x is %d and is stored at %p.\n", x, &x);
      printf("   xptr holds %p and points to %d.\n", xptr, *xptr);
```

```
        display(x, xptr);
        return 0;
}
void display(int z, int *zptr)
 {
        printf("In display():\n");
        printf("   z is %d and is stored at %p.\n", z, &z);
        printf("   zptr holds %p and points to %d.\n", zptr, *zptr);
}
```

要使用 gdb 调试程序，一定要在编译程序时，使用-g 编译选项，以生成参数符号表(augmented symbol table)，提供调试信息。

首先使用 gcc –g –o example1 example.c 对源代码进行编译，这样就可以使用 gdb 监视 example1 的执行细节。在 bash 提示符下，键入命令:gdb example1，启动了对可执行文件 example1 的调试，在屏幕上会出现下面的信息：

```
[nie@uClinux nie]$ gdb example1
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

最后一行(gdb)就是进入到 gdb 调试中的提示符，此时可以在提示符下输入任何想键入的命令。

现在如果要进行断点调试的话，就需要显示一下要调试的源码，以便知道在哪个地方进行断点设置。在 gdb 下，Linux 最常用文本编辑命令 vi 不能使用，可以使用 list 命令列出可执行文件的源代码的一部分，为了列出源代码的全部，只要多键入几次 list 命令即可。具体操作如下：

```
(gdb) list
1       #include <stdio.h>
2       static void display(int i, int *ptr);
3
4        int main(void) {
5          int x = 5;
6          int *xptr = &x;
7          printf("In main():\n");
8          printf("   x is %d and is stored at %p.\n", x, &x);
9          printf("   xptr holds %p and points to %d.\n", xptr, *xptr);
10         display(x, xptr);
(gdb) list
11         return 0;
12      }
13
14      void display(int z, int *zptr) {
15         printf("In display():\n");
16         printf("   z is %d and is stored at %p.\n", z, &z);
17         printf("   zptr holds %p and points to %d.\n", zptr, *zptr);
18      }
(gdb) list
Line number 19 out of range; example1.c has 18 lines.
(gdb)
```

屏幕上清楚显示出了每一个语句所在的具体行号，比如现在我们想在第五行设置断点，可以在 gdb 提示符下输入命令：break 5，可以看到下面的显示信息：

(gdb) break 5

Breakpoint 1 at 0x8048466: file example1.c, line 5.

断点已经设置好，现在开始让程序运行起来，键入命令 run，也可以键入其缩写形式 r，屏幕上出现的信息如下：

```
(gdb) r
Starting program: /home/nie/example1
Breakpoint 1, main () at example1.c:5
5         int x = 5;
```

上述信息表明，gdb 已经开始执行可执行程序，目前程序运行到 example1.c 程序中 main()函数的第五行处停止，并且显示出即将要执行的第五行语句。

现在我们进行单步调试的工作，输入命令：next，它表明单步执行程序的每一条语句，当用 next 命令执行到函数 display 处时，即当屏幕出现如下所示信息时：

```
(gdb) next
6             int *xptr = &x;
(gdb) next
7             printf("In main():\n");
(gdb) next
In main():
8         printf("   x is %d and is stored at %p.\n", x, &x);
(gdb) next
   x is 5 and is stored at 0xbffffb44.
9         printf("   xptr holds %p and points to %d.\n", xptr, *xptr);
(gdb) next
   xptr holds 0xbffffb44 and points to 5.
10        display(x, xptr);
```

为了进入到函数 display 内部进行调试，输入命令 step，即：

```
(gdb) step
display (z=5, zptr=0xbffffb44) at example1.c:15
15            printf("In display():\n");
```

step 命令使执行进入到函数内部，此时在该函数内部，可以继续使用 step 命令或者是 next 命令进行单步执行，如果不想单步执行，而是直接将程序一次执行完毕，可以输入命令 continue 即可。

要退出 gdb，请键入命令 quit，如果程序此时仍在进行，gdb 会让你确认是否真的要退出，屏幕会出现类似下面的提示信息：

```
   (gdb) quit
The program is running.  Exit anyway? (y or n)
```

按下'y' 即退出调试程序，如果程序本身已经运行完毕，则 quit 命令键入后，会直接退出 gdb,而不出现任何提示信息。

当然除了使用 gdb 进行程序调试外，如果程序比较简短，逻辑又比较简单，此时完全可以不用 gdb，采用 printf 语句在程序当中输出中间变量的值来调试程序，也是一个不错的调试方法。

到此为止，我们已经介绍了 uClinux 操作系统，GNU 工具的使用，有了这些预备知识后，我们将进入到本章的重点内容了。

7.3 建立 uClinux 开发环境

为了实现基于 uClinux 的应用系统的开发，建立或拥有一个完备的 uClinux 开发环境是十分必要的。

基于 uClinux 操作系统的应用开发环境一般是由目标系统硬件开发板和宿主 PC 机所构成。目标硬件开发板(在本书中就是基于 S3C4510B 的开发板)用于运行操作系统和系统应用软件，而目标板所用到的操作系统的内核编译、应用程序的开发和调试则需要通过宿主 PC 机来完成。双方之间一般通过串口，并口或以太网接口建立连接关系。

7.3.1 建立交叉编译器

通常的嵌入式系统的开发都是以装有 Linux 的 PC 机作为宿主机来编译内核和用户应用程序的，但是对于很多长期工作在 Windows 操作系统下的用户来说，突然切换到 Linux 环境下去开发程序会感到诸多不便，因此本书对于不同的读者提供了在宿主机装有不同操作系统时，相应的交叉编译环境建立的方法。

7.3.1.1. 为安装 Linux 的宿主机建立交叉编译器

首先，要在宿主机上安装标准 Linux 操作系统，如 RedHat Linux(本书使用的是 Redhat 7.2)，一定要确保计算机的网卡驱动、网络通讯配置正常，有关如何在 PC 机上安装 Linux 操作系统的问题，请参考有关资料和手册。

由于 uClinux 及它的相关开发工具集大多都是来自自由软件组织的开放源代码，所以在软件开发环境建立的时候，大多数软件都可以从网络上直接下载获得，接下来就可以建立交叉开发环境。

现在介绍一下交叉编译的概念。简单地讲，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。注意这里的平台，实际上包含两个概念：体系结构(Architecture)、操作系统(Operating System)。同一个体系结构可以运行不同的操作系统；同样，同一个操作系统也可以在不同的体系结构上运行。举例来说，我们常说的 x86 Linux 平台实际上是 Intel x86 体系结构和 Linux for x86 操作系统的统称；而 x86 WinNT 平台实际上是 Intel x86 体系结构和 Windows NT for x86 操作系统的简称。就本书所涉及到的目标硬件 S3C4510B 而言，之所以使用交叉编译是因为在该硬件上无法安装我们所需的编译器，只好借助于宿主机，在宿主机上对即将运行在目标机上的应用程序进行编译，生成可在目标机上运行的代码格式。

读者可以从 http://mac.os.nctu.edu.tw/->download 处下载工具链：arm-elf-binutils-2.11-5.i386.rpm，arm-elf-gcc-2.95.3-2.i386.rpm，genromfs-0.5.1-1.i386.rpm 的文件复制到宿主机上的任一目录下。键入下面的命令来安装 rpm 包：

```
$su
# rpm –ivh *.rpm
```

RPM(Red Hat Package Manger)软件包管理程序，是将原本复杂的软件包安装程序，轻松利用单一操作来完成。

RPM 目前支持的平台有 3 种类型：x86(i386)，Sparc 以及 Alpha，可以很容易的从文件名就来判断出使用的平台。像目前下载文件比如 arm-elf-binutils-2.11-5.i386.rpm，arm-elf-binutils 表示文件名，2.11 表示版本编号，5 表示发行序号，也就是目前已经发行的次数，i386 是指此软件包为适用于 Intel x86 的二进制(binary)程序，也就是已经编译并且可以直接安装的软件包，最后的“rpm”表示这是 Red Hat 的 RPM 程序。每一版的 RPM 发布后，若是发现软件有问题，都会重新进行 patch 和 build，这样在发行序号的部分就会增加 1，以表示该版本是上个版本的更新。

这里在所用的命令 rpm –ivh 中，-i 表示 Installation，就是安装指定的 RPM 软件包；

-h 表示 Hash，该参数可在安装期间出现”#”符号，来显示目前的安装过程，这个符号一直持续到安装完成后才停止；

-v 表示 Verbose，显示安装时候的详细信息。

至此我们把交叉编译器已经安装到了宿主机。以后我们就可以用交叉编译器 arm-elf-gcc 编译操作系统内核和用户应用程序了。

读者也可以从网站 http://www.uclinux.org/pub/uClinux/arm-elf-tools/ 上下载最新的 arme-elf-gcc 工具，即脚本文件 arm-elf-tools-20030314.sh，在宿主机上安装该工具链，在该文件所在目录下，键入：

$ su

ls –l arm-elf-tools-20030314.sh

该命令显示文件的各种属性，如果该脚本文件属性的不是可执行的，则还需要输入命令：# chmod 755 arm-elf-tools-20030314.sh

以将其属性改为可执行属性，然后通过键入命令:

#sh ./arm-elf-tools-20030314.sh

就可以执行该文件。执行后/usr/local/bin/路径下有 gcc, g++, binutils, genromfs, flthdr 和 elf2flt 等各种实用工具。

7.3.1.2 为安装 windows 的宿主机建立交叉编译器

这部分内容是专门针对那些对 Linux 环境和 Linux 中的应用程序不熟悉，宁愿用 PC 上基于 Windows 的操作系统来开发嵌入式系统的读者而写的。

1. Cygwin 软件介绍

为了在 Windows 下开发嵌入式操作系统应用程序，可以在 Windows 环境下装上 Cygwin 软件。Cygwin 是一个在 Windows 平台上运行的 Unix 模拟环境，是 Cygnus Solutions 公司开发的自由软件。它对于学习掌握 Unix/Linux 操作环境，或者进行某些特殊的开发工作，尤其是使用 GNU 工具集在 Windows 上进行嵌入式系统开发，非常有用。

Cygnus 当初首先把 gcc，gdb 等开发工具进行了改进，使它们能够生成并解释 win32 的目标文件。然后，把这些工具移植到 windows 平台上去。一种方案是基于 win32 API 对这些工具的源代码进行大幅修改，这样做显然需要大量工作。因此，Cygnus 采取了一种不同的方法——他们写了一个共享库(就是 cygwin1.dll)，把 win32 API 中没有的 Unix 风格的调用(如 fork,spawn,signals,select,sockets 等)封装在里面，也就是说，他们基于 win32 API 写了一个 Unix 系统库的模拟层。这样，只要把这些工具的源代码和这个共享库连接到一起，就可以使用 Unix 主机上的交叉编译器来生成可以在 Windows 平台上运行的工具集。以这些移植到 Windows 平台上的开发工具为基础，Cygnus 又逐步把其他的工具(几乎不需要对源代码进行修改，只需要修改他们的配置脚本)软件移植到 Windows 上来。这样，在 Windows 平台上运行 bash 和开发工具、用户工具，感觉好像在 Unix 上工作。 关于 Cygwin 实现的更详细描述，请参考 http://cygwin.com/cygwin-ug-net/cygwin-ug-net.html。

2. Cygwin 软件的安装

要得到 Cygwin 的最新安装版本，请到 Cygwin 的主页 http://cygwin.com/上下载最新的 Cygwin，在该页面的右上角有" Install Cygwin Now "，点击此处，就会先下载一个叫做 setup.exe 的 GUI 安装程序，用它能下载一个完整的 Cygwin。图 7.2 所示为在点击 setup.exe 后出现"选择安装类型"对话框。建议读者把 Cygwin 整个安装包先下载到本地，再进行本地安装比较方便，即在下图先选择第二个选项，等到将 Cygwin 完全下载后，再选择第三个选项进行本地安装。

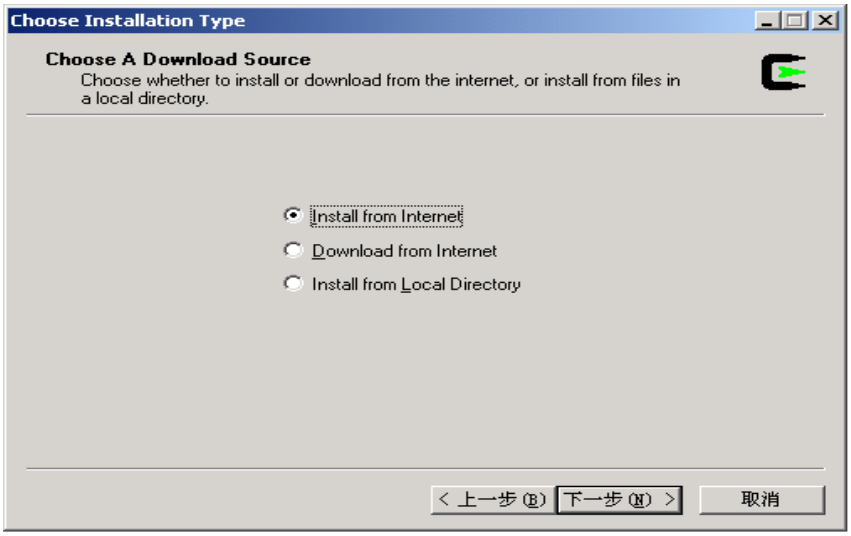


图 7.2　选择安装类型

安装的时候建议最好不要安装到 C:\目录下，比如安装在 D:\下。

在安装的过程中，会让用户选择安装哪些包，这些包主要是确定开发环境，编译工具等，如果不能确定具体需要哪些包的话，而硬盘空间足够的情况下，就选择全部安装。在出现的对话框的"All"的右边点击"Default"，直到变成"Install"，如下图 7.3 所示：

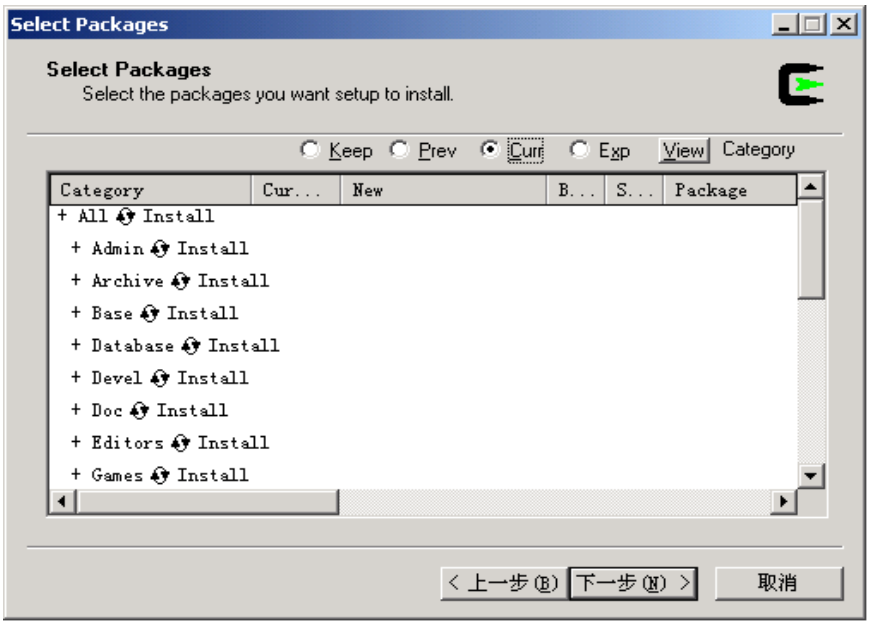


图 7.3　选择安装包

Cygwin 的安装过程时间比较长，请读者耐心等待。当出现创建图标的画面点击“完成”按钮之后，屏幕会有几秒钟的闪动，出现类似下面的画面如图 7.4 所示，这是在执行 Cygwin 安装后的脚本配置。

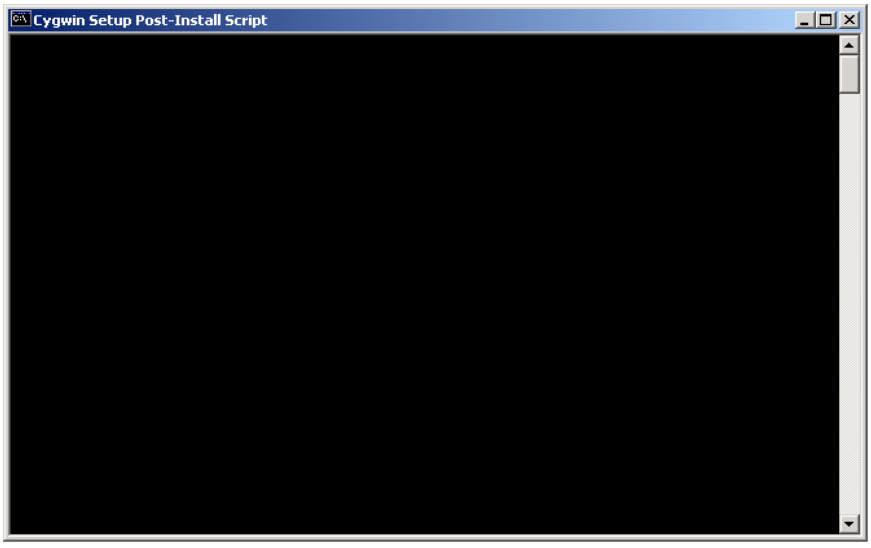


图 7.4　Cygwin 安装后自动配置

自动配置结束后，出现 Cygwin 成功安装结束的提示框。桌面上会出现 Cygwin 的图标。

3.　在 Cygwin 下生成交叉编译器

在自己生成交叉编译器之前，首先对 cygwin 进行一些设置。假设 Cygwin 安装在 d 目录下，在打开 Cygwin 窗口之前，进入到 D:\cygwin 目录，在这个目录下，有一个文件名为 cygwin.bat 的批处理文件，编辑该文件，在第一行后加入 set CYGWIN=title ntea，这是因为 cygwin 的启动批处理文件需要启动 Unix 文件系统模拟。修改完毕后，保存后退出。双击桌面上的 Cygwin 图标，打开后默认用户为在 Windows 中登录的用户名(这里所使用的操作系统是 windows 2000 professsional)，在如图

7.5 所示的界面中，在根目录(即 D:\cygwin)下键入：

```
cd bin
mv sh.exe sh-original.exe
ln –s bash.exe sh.exe
```

做上述几步的原因是因为大多数 linux 系统将 sh 符号链接到 bash，Cygwin 上的 sh.exe 和 bash.exe 是不同的，因此必须用 bash 代替 sh。

从网站 http://www.uclinux.org/pub/uClinux/arm-elf-tools/tools-20030314/上下载生成工具链的各种源码，根据脚本文件 build-uclinux-tools.sh 建立可在 windows 下编译用户应用程序的交叉编译器，生成的交叉编译器最终被打包为 arm-elf-tools-cygwin-yyyymmdd.tar.gz 的文件，其中 yyyy 为生成交叉编译器的年，mm 为生成交叉编译器的月份，dd 为日期。

这里，希望读者注意的是在生成交叉编译器的过程中，可能会遇到多次错误，读者应该根据给出的出错信息，进行相应文件的修改。由于习惯上的原因，linux 下的压缩文件一般都是以.tar.gz 或者.tgz 结尾的，虽然用 windows 下的解压软件比如 winzip 或者 winrar 可以解压这些文件，但是推荐读者不要用这些软件在 windows 下解压，因为这样可能会造成某些信息的丢失。

本书生成的交叉编译器名为 arm-elf-tools-cygwin-20030502.tar.gz。

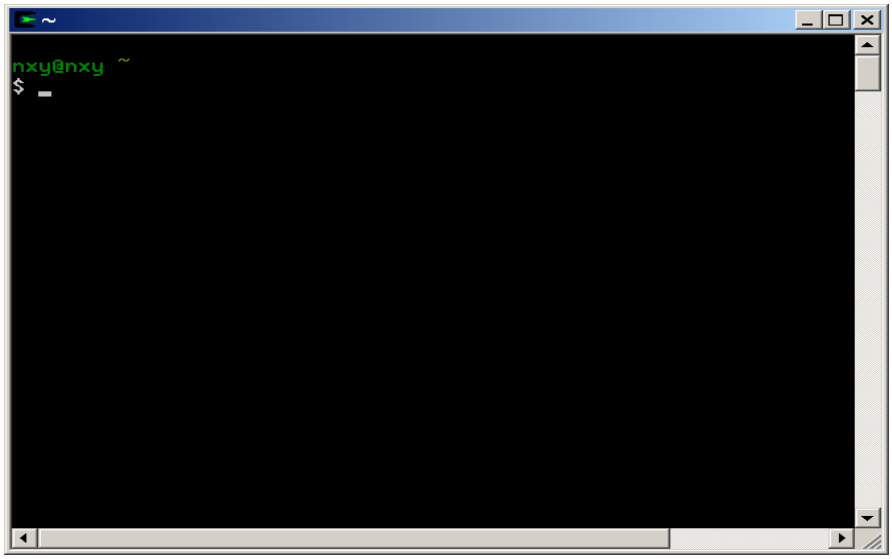


图 7.5 Cygwin 开发环境

4. 在 Cygwin 环境下建立交叉编译器

在根目录下键入：

tar xvzf arm-elf-tools-cygwin-20030502.tar.gz

进行交叉编译器的解压，解压完毕后在/usr/local/bin/目录下可以看到各种 GNU 工具。有了交叉编译器后，熟悉 Windows 的读者就可以在 Windows 下编译在 uClinux 上运行的应用程序了。

7.3.2 uClinux 针对硬件的改动

目前，uClinux 已被成功移植到 S3C4510B 及其他多款 ARM 芯片上，但由于嵌入式操作系统的运行是与嵌入式系统的硬件密切相关的，而硬件的设计则会因为使用场合的不同而千差万别，因此，在 uClinux 内核源代码中和硬件紧密相关的部分就应该针对特定的硬件作出适当的修改，由于 uClinux 内核源代码包含很大一部分的硬件驱动程序，不可能一一列举，在此，就基于 S3C4510B 的最小系统的设计与运行相关的部分作简单的介绍，希望对读者有所启发。

uClinux 内核源代码中对 S3C4510B 片内特殊功能寄存器以及其他相关硬件信息的定义位于 uClinux-Samsung\Linux-2.4.x\include\asm-armnommu\arch-samsung\hardware.h 文件中，其中有几个地方值得注意：

```
/*
 * define S3C4510b CPU master clock
 */
#define MHz      1000000
#define fMCLK_MHz(50 * MHz)
#define fMCLK    (fMCLK_MHz / MHz)
#define MCLK2    (fMCLK_MHz / 2)
```

以上定义了系统工作的主时钟频率为 50MHz，若用户系统的工作频率不同，应在此处修改，若串行口采用内部时钟信号用于波特率生成，该频率同时还与串行通信波特率有关。

```
/*********************************/
/* System Memory Control Register */
/*********************************/
#define DSR0     (2<<0)  /* ROM Bank0 */
#define DSR1     (0<<2)  /* 0: Disable, 1: Byte, 2: Half-Word, 3: Word */
#define DSR2     (0<<4)
#define DSR3     (0<<6)
#define DSR4     (0<<8)
#define DSR5     (0<<10)
#define DSD0     (2<<12) /* RAM Bank0 */
#define DSD1     (0<<14)
#define DSD2     (0<<16)
#define DSD3     (0<<18)
#define DSX0     (0<<20) /* EXTIO0 */
#define DSX1     (0<<22)
#define DSX2     (0<<24)
#define DSX3     (0<<26)
#define rEXTDBWTH(DSR0|DSR1|DSR2|DSR3|DSR4|DSR5 | DSD0|DSD1|DSD2|DSD3 |
DSX0|DSX1|DSX2|DSX3)
```

以上定义了系统存储器控制寄存器，按照以上定义，ROM/SRAM/FLASH Bank0 定义为 16 位数据宽度（事实上，ROM/SRAM/FLASH Bank0 的数据宽度由 B0SIZE[1:0]的状态决定），而 ROM/SRAM/FLASH Bank1～ROM/SRAM/FLASH Bank5 禁用；DRAM/SDRAM Bank0 定义为 16 位数据宽度，DRAM/SDRAM Bank1～DRAM/SDRAM Bank3 禁用；外部 I/O 组全部禁用；若用户系统的存储器系统配置不同，应在此处修改。

之后还做了其他一些改动，包括对 ROM/SRAM/FLASH Bank0 控制寄存器的设置，Flash 容量的设置，DRAM/SDRAM Bank0 控制寄存器的设置，SDRAM 容量的设置等，这些设置均应该与用户系统对应。

7.3.3 编译 uClinux 内核

作为操作系统的核心，uClinux 内核负责管理系统的进程、内存、设备驱动程序、文件系统和网络系统，决定着系统的各种性能。uClinux 内核的源代码是完全公开的，任何人只要遵循 GPL，就可以对内核加以修改并发布给他人使用，因此，在广大编程人员的支持下，uClinux 的内核版本不断更新，新的内核修改了旧的内核的缺陷，并增加了许多新的特性，用户如果想在自己的系统中使用这些新的特性，或想根据自己的系统量身定制更高效、更稳定可靠的内核，就需要重新编译内核。一般说来，更新的内核版本会支持更多的硬件，具有更好的进程管理能力，运行速度会更快、更稳定，并且一般都会修复旧版本中已发现的缺陷等，因此，经常选择升级更新的系统内核是必要的。

uClinux 内核采用模块化的组织结构，通过增减内核模块的方式来增减系统的功能，因此，正确合理的设置内核的功能模块，从而只编译系统所需功能的代码，会对系统的运行进行如下几个方面的优化：

— 用户根据自身硬件系统的实际情况定制编译的内核因为具有更少的代码，一般会获得更高的运行速度。

— 由于内核代码在系统运行时会常驻内存，因此，更短小的内核会获得更多的用户内存空间。

— 减少内核中不必要的功能模块，可以减少系统的漏洞，从而增加系统的稳定性和安全性。

uClinux 的内核源代码可以从许多网站上免费下载，内核的发布一般有两种形式，一种是完整的内核版本，完整的内核版本一般是.tar.gz 文件，使用时需要解压。另一种是通过对旧的版本发布补丁（patch），达到升级的效果。

本例所采用的在Linux下使用的交叉编译器和uClinux-Samsung-20020318.tar.gz源码均来自网站 http://mac.os.nctu.edu.tw。

在准备好 uClinux 的内核源代码后，利用交叉编译器就可以编译生成运行在硬件目标板上的 uClinux 内核。

从 http://mac.os.nctu.edu.tw 上下载 uClinux 内核源代码 uClinux-Samsung-20020318.tar.gz，保存到宿主机的用户目录。运行解压命令：

tar xzvf uClinux-Samsung-250020318.tar.gz

解压完毕后，就会在用户目录下生成 uClinux-Samsung 目录，以下命令进入到该目录中：

$ cd uClinux-Samsung

1. 键入命令：

make menuconfig

内核配置。该命令执行完毕后生成文件.config，它保存这个配置信息。下一次再做 make menuconfig 的时候将产生新的.config 文件，原来的.config 被改名为.config.old。

此时会出现菜单配置对话框，要求进行目标平台的选择，如图 7.6 所示，输入回车后，出现供选择的具体的供应商和产品列表，在这里我们选择：Samsung/4510B，如图 7.7 所示，在库的选择上，我们选择 uC-libc，其他选项暂时不用修改，保存好设置后，存盘退出。

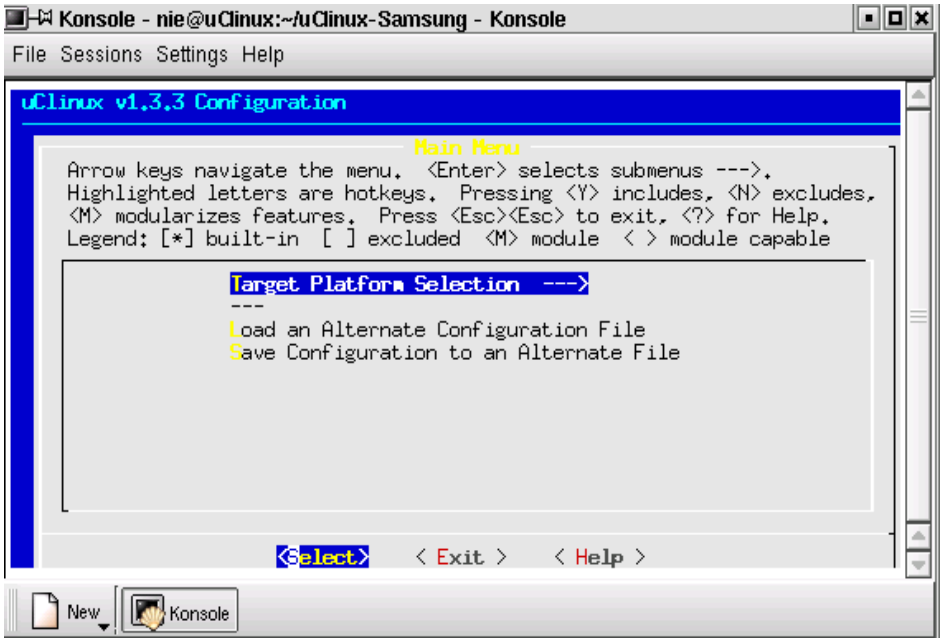


图 7.6 目标平台配置

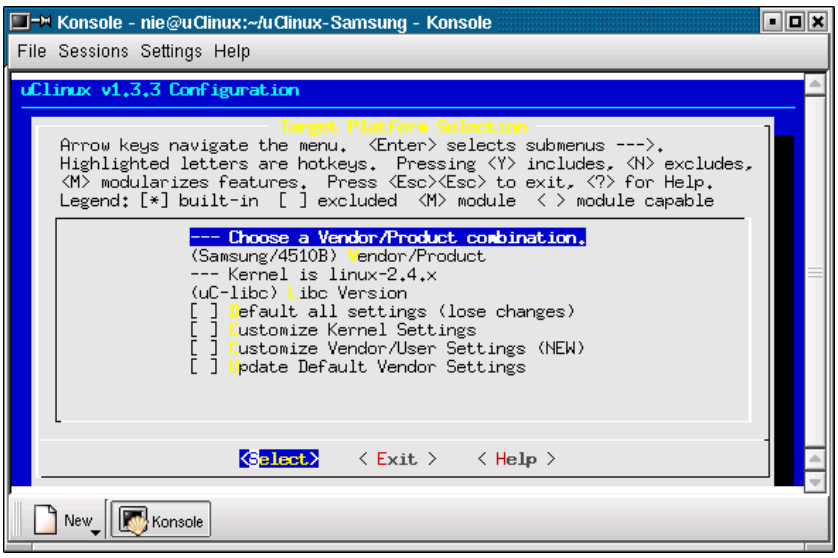


图 7.7　选择合适的产品类型

2．键入命令：make dep

该命令用于寻找依存关系。

3．键入命令：make clean

该命令清除以前构造内核时生成的所有目标文件，模块文件和一些临时文件。

4．键入命令：make lib_only

该命令编译库文件。

5．键入命令：make user_only

该命令编译用户应用程序文件。

6．键入命令：make romfs

该命令生成 romfs 文件系统。

7．键入命令：make image

注意做到这一步的时候可能会出现错误的信息提示，类似于：

arm-elf-objcopy: /home/nie/uClinux-Samsung/linux-2.4.x/linux: No such file or directory

make[1]: *** [image] Error 1

make[1]: Leaving directory `/home/nie/uClinux-Samsung/vendors/Samsung/4510B'

make: *** [image] Error 2

这是因为第一次编译时还没有 romfs.o，所以出错，等 romfs.o 编译好了以后，如果再进行内核的编译，就不会出现这个错误信息了。它完全不影响内核的编译，可以完全不必理会这个错误信息。继续进行编译工作。

8．键入命令：make

通过各个目录的 Makefile 文件进行，会在各目录下生成一大堆目标文件。

上述步骤完成后，就完成了对 uClinux 源码的编译工作。整个编译过程视计算机运行速度而定，大约需要十几分钟左右。

在编译内核的时，建议在 Linux 平台下进行。

7.3.4　内核的加载运行

当内核的编译工作完成之后，会在/ uClinux-Samsung/images 目录下看到两个内核文件：image.ram 和 image.rom，其中，可将 image.rom 烧写入 ROM/SRAM/FLASH Bank0 对应的 Flash 存储器中，当系统复位或上电时，内核自解压到 SDRAM，并开始运行。

image.ram 可直接在系统的 SDRAM 中运行，使用 ADS（或 SDT）集成开发环境将系统的 SDRAM

映射到起始地址为 0x0 处，并将 image.ram 载入从 0x8000 开始的 SDRAM 中，加载完毕后，修改 PC 指针寄存器的值为 0x8000 并执行。

注意该内核默认串行口 COM1 为输入输出控制台，波特率为 19200，8 个数据位，1 个停止位，无校验。

7.4 在 uClinux 下开发应用程序

当完成了上述所有工作后，一个嵌入式应用开发平台就已经搭建好了，在这个平台之上，就可以根据不同需要开发嵌入式应用了。图 7.8 所示为一个基于 uClinux 的嵌入式系统典型框架结构，下面将向读者介绍如何将自己开发的应用程序添加到目标板上运行。

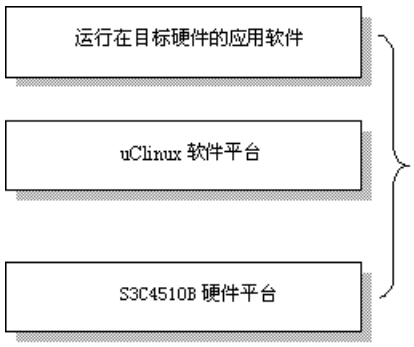


图 7.8 基于 uClinux 嵌入式系统框图

基于 uClinux 系统的应用程序的开发通常是在标准 Linux 平台上(本书已经介绍了适用于 Windows 环境的交叉编译器，所以也可以在 Windows 平台)用交叉编译工具来完成。由于 uClinux 是为没有内存管理单元(MMU)的处理器和控制器而设计的，并做了较大幅度的精简，所以可能出现这样的情况：在标准 Linux 下可以使用的某些函数在 uClinux 下却用不了，这个时候，就需要用户编写相应的库函数了。当然绝大多数的函数它们都还是通用的。除此以外，在 x86 版本的 gcc 编译器下编译通过的软件，通常不需要做太大的改动就可以用刚才我们建立的交叉编译工具编译成可以在 uClinux 上运行的文件格式。因此开发在 uClinux 下运行的程序，基本上就和开发在 Linux 下运行的程序是一样的，关于 Linux 下的编程，读者可以参考其他更详细的资料，以下就一个简单的例子，描述其基本开发过程。

考虑一个定时中断的例子，文件名为 lednxy.c，其源代码如下：

```
/*******************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name:     lednxy.c
 * Description:   timing interrupt
 * Author:        Xueyuan Nie
 * Date:
 *******************************************************/
#include <signal.h>
#include <unistd.h>
#define IOPMOD   (*(volatile unsigned *)0x3ff5000)
#define IOPDATA  (*(volatile unsigned *)0x3ff5008)
int i=0;
static void sig_alarm(int signumber)
{
   if(i==3) i=0;
   IOPDATA=i++;
```

```
    alarm(2);
}

int main(void)
{
  IOPMOD=0xff;
  if(signal(SIGALRM,sig_alarm)==SIG_ERR)
  {
      printf("some error occurs\n");
       return  1;
  }
  alarm(2);
  while(1);
  return 0;
}
```

在代码中，SIGALRM 为系统定义的信号的名字，在头文件里被定义为一个正整数，用户自定义函数 sig_alarm()为信号处理函数， 系统函数 alarm()用来设定一个 2 秒的定时器，当定时器时间片终止的时候，进程将会产生 SIGALRM 信号，在程序中用函数 signal()实现了信号 SIGALRM 和信号处理函数 sig_alarm()的连接，这样，当用 alarm()函数设置时钟的时间段终止时，就会有 SIGALRM 信号产生，程序就会转而执行函数 sig_alarm()，从而实现每隔 2 秒钟，I/O 口数据寄存器的值发生一次变化，达到控制 LED 等的目的。有关 signal()函数和 alarm()函数的使用，读者可以查阅有关在 Linux 上的 C 编程方面的内容，本书在此不作详述。

该程序达到的效果就是，让目标硬件上的 P0 和 P1 口的两个 LED 显示器按照 P0 亮，P1 亮，P0、P1 全亮的顺序，每隔 2 秒实现其中的一种状态。

在装有标准 Linux 的宿主机(或装有 Cygwin 的 windows 的 PC 机)上，用前面已经建立好的交叉编译工具编译源文件，在该程序所在的目录下键入如下命令：

arm-elf-gcc –Wall –O2 –Wl,-elf2flt –o lednxy　lednxy.c

仍然在该目录下，键入命令：

ls

可以查看到在该目录下生成了文件名为 lednxy 的文件。

在键入的编译命令中，选项 ：

-Wall　指定产生全部的警告信息；

-O2　是一个二级优化选项，它表示告诉编译器产生尽可能小和尽可能快的代码；

-Wl　的一般用法是’’-Wl,option’’ 就是把它后面的选项传递给链接器，在本命令中就是把’’-elf2flt ‘’传给链接器；

-elf2flt　指定自动调用 elf 转换 flat 格式的工具；之所以要使用该选项是因为，由于 GNU 工具本身并不支持 flat 格式的二进制文件，然而，uClinux 目前只支持 flat 格式的可执行文件，因此必须使用相应的二进制工具进行格式转换。flat 格式是对 elf 格式的很大的文件头和一些段信息做了简化的文件格式。

编译成功后得到的 lednxy 就可以在 uClinux 环境上运行了。关于如何将生成的可执行代码加入到 uClinux，将在后面的章节讲述。

除了以命令行的形式进行代码编译外，我们还可以利用前面提到的 makefile 的知识，用 makefile 文件实现代码编译的功能。

下面给出本例相应的 makefile 文件(该文件名为 makefile)。

```
CFLAGS  = -Wall –Os –Dlinux –D__linux__ -Dunix –D__uClinux__ -DEMBED
LDFLAGS = -Wl,-elf2flt
CC      = arm-elf-gcc
LD      = arm-elf-gcc
```

```
TARGT   = lednxy
OBJ     = $(TARGT).o
SRC     = $(TARGT).c
all: $(TARGT)
%.o : %.c
      $(CC) $(CFLAGS) –c $< -o $@
$(TARGT): $(OBJ)
      $(CC) $(CFLAGS) $(LDFLAGS) –o $@ $(OBJ)
```

整个编译过程如下：

```
[nie@uClinux usr]$ make
arm-elf-gcc –Wall –Os –Dlinux –D__linux__ -Dunix –D__uClinux__ -DEMBED  -c
lednxy.c –o lednxy.o
arm-elf-gcc –Wall –Os –Dlinux –D__linux__ -Dunix –D__uClinux__ -DEMBED
-Wl,-elf2flt –o lednxy lednxy.o
```

可以用工具 arm-elf-flthdr 查看生成的 lednxy 的格式，它是一个能够操作和显示 flat 格式文件的头信息的可执行程序。在生成 lednxy 的当前路径下键入命令：

arm-elf-flthdr lednxy

后，可以看到以下对该文件头描述的信息，

```
lednxy
Magic:       bFLT
Rev:         4
Build Date:   Thu Jun 19 10:31:14 2003
Entry:       0x50
Data Start:    0x1c80
Data End:     0x2010
BSS End:     0x22a0
Stack Size:    0x1000
Reloc Start:   0x2010
Reloc Count:  0x4f
Flags:       0x1 ( Load-to-Ram )
```

从显示的信息，可以看出文件 lednxy 的确是一个 flat 格式的文件，是可以在 uClinux 环境下运行的。

7.4.1 串行通信

所谓串行通信就是在传输数据的时候每次只传输一位，其传输的速率通常用“位/秒”来表示，即通常所说的“波特率”。

Linux 对所有各类设备文件的输入输出操作，看上去就像对普通文件的输入输出一样，所以 Linux 对串口的操作，也是通过设备文件访问的。为了访问串口，只需要打开相应的设备文件即可。设备文件/dev/ttyS*是用于挂起 Linux 终端的文件。默认地，在 Linux 下，串行口 COM1 和 COM2 对应的设备分别为/dev/ttyS0 和/dev/ttyS1。

在程序中，很容易配置串口的属性，这些属性定义在结构体 struct termios 中。为在程序中使用该结构体，需要包含文件<termbits.h>，该头文件定义了结构体 struct termios。

```
#define NCCS 19
struct termios {
      tcflag_t c_iflag;            /* input mode flags */
      tcflag_t c_oflag;            /* output mode flags */
      tcflag_t c_cflag;            /* control mode flags */
      tcflag_t c_lflag;            /* local mode flags */
```

```
        cc_t c_line;                /* line discipline */
        cc_t c_cc[NCCS];             /* control characters */
};
```

下面对结构体中的各个成员做一个简单介绍。

在 c_iflag 中的输入模式标志符控制所有的输入处理过程，就是说，从设备发送的字符在被 read 函数读取之前要经过处理。类似的，成员 c_oflag 控制输出处理过程，c_cflag 包含对端口的设置，如，波特率，字符位数，停止位等。存储在成员 c_lflag 的本地模式标志符决定是否显示字符，是否发送信号到应用程序等。数组 c_cc 包含了控制字符的定义和超时参数。成员 c_line 在 POSIX(Portable Operating System Interface for UNIX)系统中不使用。

下面结合一个简单的实例，说明如何对串口进行读写操作。

```
/*****************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name:     serialcomm.c
 * Description: communication with serial
 * Author:        Xueyuan Nie
 * Date:
 *****************************************************/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#define BAUDRATE B19200
#define SERIALDEVICE "/dev/ttyS0"
int main()
{
        int fd,ncount;
        struct termios oldtio,newtio;
        char buf[]="This is a simple application for serial communication\r\n";
       fd = open(SERIALDEVICE, O_RDWR | O_NOCTTY );
       if (fd <0)
         {
               perror(SERIALDEVICE);
               exit(-1);
         }
       tcgetattr(fd,&oldtio);
       bzero(&newtio, sizeof(newtio));
       newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
       newtio.c_iflag = IGNPAR | ICRNL;
       newtio.c_oflag = 0;
       newtio.c_lflag = ICANON;
       tcflush(fd, TCIFLUSH);
       fcntl(fd,F_SETFL,0);
       tcsetattr(fd,TCSANOW,&newtio);
       ncount=write(fd,buf,sizeof(buf));
       printf("the bytes written to serial is %d\n",ncount);
       printf("character to send is: %s\n",buf);
       perror("write");
       tcsetattr(fd,TCSANOW,&oldtio);
       close (fd);
```

```
        return 0;
}
```

程序首先为波特率常数定义了宏值，为设备文件定义了设备名常数。有关波特率常数的定义可参见<termbits.h>(该头文件包含在 termios.h 中)。

对于普通用户而言，是不允许访问设备文件的，如果要访问，要么以 root 账号登录，要么需要改变文件的访问属性。假定设备文件是可以访问的，用 open 函数打开设备文件，返回一个文件描述符(file descriptors,fd)，通过文件描述符来访问文件。O_RDWR 标志表示对该文件可读可写，O_NOCTTY 表示该程序不会成为控制终端，这样就避免了当在键盘输入类似 ctrl+c 的命令后，终止程序的运行。

然后用 tcgetattr 保存串口的当前设置，给端口设置新的属性，通过对 c_cflag 的赋值，设置波特率，字符大小(CS8 表示 8 位数据位，1 位停止位，没有奇偶校验位)，使能本地连接，使能串行口驱动读取输入数据。

通过设置 c_iflag ，控制端口对字符的输入处理过程，IGNPAR 符号常量表示忽略奇偶性错误的字节，并不对输入数据进行任何校验，ICRNL 将回车符映射为换行符。

设置原始数据输出，使能规范输入。

在对 struct termios 结构体的各个成员赋值完毕后，调用 tcsetattr 函数选择新的设置，常数 TCSANOW 表示新设置立即生效。

调用 write 函数往串口发送数据，此时如果打开超级终端应该可以看到写入的字符串。对串口操作结束后，恢复原有的端口设置，关闭打开的设备文件。

以上是一个简单的对串口进行写操作的程序，因为通过超级终端来显示，所以没有调用 read 函数，如果接收数据的一端是其他设备的话，有可能需要读者再编写一个接收数据的程序，运行发送和接收程序的两台设备通过串行口进行连接。也可以将接收和发送的程序在同一台设备上运行，通过一根交叉线(就是将 TXD－数据传输信号和另一个端口的 RXD －接收数据信号相连起来)将两个串口接在一起。

下面就针对上述提到的情况，再举一个有关接收和发送数据的程序，通过串口交叉线的连接运行在不同设备(也可以是同一台设备)上的例子。

假设接收程序 readtest.c 运行在装有标准 Linux 的 PC 机上，发送程序 writetest.c 运行在目标板 S3C4510B 上，两台设备的串口通过交叉线连接在一起。

接收程序 readtest.c 的源码如下：

```
/*****************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name:     readtest.c
 * Description:   receive data from the serial
 * Author:        Xueyuan Nie
 * Date:
 *****************************************************/
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include "math.h"

int spfd;
int main()
```

```
{
 char fname[16],hd[16],*rbuf;
 int  retv,i,ncount=0;
 struct termios oldtio;
 int realdata=0;

spfd=open("/dev/ttyS1",O_RDWR|O_NOCTTY);
 perror("open /dev/ttyS1");
 if(spfd<0) return -1;

 tcgetattr(spfd,&oldtio);
 cfmakeraw(&oldtio);
 cfsetispeed(&oldtio,B19200);
 cfsetospeed(&oldtio,B19200);
 tcsetattr(spfd,TCSANOW,&oldtio);
 rbuf=hd;
 printf("ready for receiving data...\n");

 retv=read(spfd,rbuf,1);
 if(retv==-1) perror("read");
  while(*rbuf!='\0')
  {
    ncount+=1;
    rbuf++;
    retv=read(spfd,rbuf,1);
    printf("the number received is %d\n",retv);
    if(retv==-1) perror("read");

  }
 for(i=0;i<ncount;i++)
 {
   realdata+=(hd[i]-48)*pow(10,ncount-i-1);
 }
 printf("complete receiving the data %d\n",realdata);

 close(spfd);
 return 0;
}
```

发送程序 writetest.c 的源码如下：

```
/*********************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name:     writetest.c
 * Description:   send data to serial
 * Author:        Xueyuan Nie
 * Date:
 *********************************************************/
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
int spfd;

int main(int argc, char *argv[])
{
  char fname[16],*sbuf;
  int sfd,retv,i;
  struct termios oldtio;

 spfd=open("/dev/ttyS1",O_RDWR|O_NOCTTY);
if(spfd<0)
{
   perror("open /dev/ttyS1");
   return -1;
 }
 printf("ready for sending data...\n");
 tcgetattr(spfd,&oldtio);
 cfmakeraw(&oldtio);
 cfsetispeed(&oldtio,B19200);
 cfsetospeed(&oldtio,B19200);
 tcsetattr(spfd,TCSANOW,&oldtio);

 fname[0]='1';
 fname[1]='2';
 fname[2]='3';
 fname[3]='\0';

 sbuf=(char *)malloc(4);
 strncpy(sbuf,fname,4);
 retv=write(spfd,sbuf,4);
 if(retv==-1) perror("write");

 printf("the number of char sent is %d\n",retv);

  close(spfd);
 return 0;
}
```

本例程实现：在发送端发送数字 123，在接收端接收并显示接收到的数据。

这里请读者注意的是，发送方按字符发送数据，接收方将接收的字符相应的 ascii 值与字符 0 所对应的 ascii 值相减，最终得到实际的十进制数值。

按照前面介绍的方法编译程序，有关如何将可执行文件添加到目标板的方法将在下一小节介绍。

开始运行程序。先在装有 Linux 的 PC 上运行接收程序，然后在 S3C4510B 上运行发送程序，整个运行的过程如下所示：

在 Linux 的 PC 上：

```
root@uClinux nie]# ./recvtest &
[1] 2171
[root@uClinux nie]# open /dev/ttyS1: Success
```

```
ready for receiving data...
[root@uClinux nie]# the number received is 1
the number received is 1
the number received is 1
complete receiving the data 123

[1]+  Done                    ./recvtest
```

在目标板上:

```
/var/tmp>  ./writetest
ready for sending data...
the number of char sent is 4
```

这里所举的例子比较简单，旨在为读者介绍最基本的串行通信的步骤，读者可以此为基础，开发出满足自己需求的应用程序来。

7.4.2 socket 编程

uClinux 本身就是一个网络的产物，它可以从网上供人们自由免费的下载，正是通过很多爱好者利用网络修改，改善 Linux，才得到我们现在的 uClinux，所以没有网络可以说就看不到今天的 uClinux。因此，在学习 uClinux 的时候，就不能不涉及到网络，而要掌握在 uClinux 下设计用户应用程序，就必须要学习有关 uClinux 下的网络编程。本节主要讲述当前在网络编程中被广泛使用的 socket。

socket 一般被翻译为“套接字”，简而言之就是网络进程中的 ID。

其实网络通信，本质就是进程间的通信，在网络中，每个节点都有唯一一个网络地址，即通常说的 IP 地址，两个进程在通信的时候，必须首先要确定通信双方的网络地址。但是网络地址只能确定进程所在的 PC 机，然而同一台 PC 可能有好几个网络进程，只有网络地址是不能够确定到底是哪个进程，所以套接字还需要提供其他信息，那就是端口号，同一台 PC 机，一个端口号只能分配给一个进程。所以，网络地址和端口号结合在一起，才可以共同确定整个 Internet 中的一个网络进程。

套接字最常用的有两种：流式套接字(Stream Socket)和数据报套接字(Datagram Socket)。在 Linux 中，分别称为”SOCK_STREAM”和”SOCK_DGRAM”。

这两种套接字的区别在于它们使用不同的协议。流式套接字使用 TCP 协议，数据报套接字使用的是 UDP 协议。

TCP(Transmission Control Protocol)传输控制协议，是 TCP/IP 体系中的运输层协议，是面向连接的，因而可提供可靠的，按序传送数据流，它的可靠是因为它使用三段握手协议来传输数据，并且采用“重发机制”确保数据的正确发送，接收端收到数据后要发出一个肯定确认，而发送端必须接收到接收端的确认信息后，否则发送端会重发数据。同时 TCP 是无错误传递的，有自己的检错和纠错机制，使用 TCP 协议的套接字是属于流式套接字。大家熟知的 telnet 就是使用的流式套接字。

UDP(User Datagram Protocol)用户数据报协议提供无连接的不可靠的服务，在传送数据之前不需要建立连接。远地主机在接收接收到 UDP 数据报后，不需要给出任何应答，这样的话，如果发送一个数据报，可能到达也可能丢失。如果发送多个包，到达接收端的次序可能是颠倒的。数据报套接字有时候也称为“无连接套接字”，大家熟悉的 TFTP 和 NFS 使用的就是该协议。

大多数情况下，如果只是将数据包发送给给定地址的机器，是不能够确定到底把数据包发送给机器哪一个进程的，端口号的指定才能够更明确的指明。适用于通信的用户应用程序可以使用从 1 到 65535 的任何一个端口号，并将它分配给端口。这些号通常分成以下几个范围段：

端口 0，不使用。如果传递的端口号是 0，就会为进程分配一个 1024 到 5000 之间的一个没有使用的端口。

端口 1～255，保留给特定的服务，如 FTP，远程网，FINGER 等。

端口 256～1023，保留给别的一般服务如 Routing function(路由函数)。

端口 1024～4999，可以被任意的客户机端口所使用，客户机套接字通常会使用这个范围段的端口。

端口 5000～65535，为用户定义的服务器端口所使用。如果一个客户机需要事先知道服务器的端口，那么服务器套接字就应该使用这个范围的端口值。

下面结合一个具体的服务器端的例子，使读者熟悉 socket 编程的方法。

```
/*******************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name:     comsamp.c
 * Description: communication with socket
 * Author:        Xueyuan Nie
 * Date:
 *******************************************************/

  #include <float.h>
  #include <stdio.h>
  #include <memory.h>
  #include <stdlib.h>
  #include <string.h>
  #include <signal.h>
  #include <unistd.h>
  #include <netinet/in.h>
  #include <netdb.h>
  #include <sys/time.h>
  #include <sys/types.h>
  #include <sys/socket.h>

  /*=========*
   * Defines *
   *=========*/
  #ifndef TRUE
  #define FALSE 0
  #define TRUE  1
  #endif

  #ifndef EXIT_FAILURE
  #define EXIT_FAILURE  1
  #endif
  #ifndef EXIT_SUCCESS
  #define EXIT_SUCCESS  0
  #endif

  #ifndef EXT_NO_ERROR
  #define EXT_NO_ERROR 0
  #endif
  #ifndef EXT_ERROR
  #define EXT_ERROR   1
  #endif

  #ifndef INVALID_SOCKET
  #define INVALID_SOCKET -1
  #endif
```

```
#ifndef SOCK_ERR
#define SOCK_ERR -1
#endif
/*==================================*
 * Global data local to this module *
 *==================================*/
typedef int SOCKET;
typedef struct ConnectData_tag {
   int    port;
  int    waitForStart;
   SOCKET   sFd;   /* socket to listen/accept on       */
   SOCKET   msgFd; /* socket to send/receive messages   */
} ConnectData;

ConnectData *CD;
int        i=0;
int        connectionMade = 0;

/*=================*
 * Local functions *
 *=================*/
void prompt_info(int signumber)
{
  char src[]="this is a test for socket\n";
  int nBytesToSet=strlen(src);
  send(CD->msgFd, src, nBytesToSet, 0);
}

void init_sigaction(void)
{
  struct sigaction act;
  act.sa_handler=prompt_info;
  act.sa_flags=0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGPROF,&act,NULL);
}

void init_time(double t_usec)
{
   struct itimerval value;
    int int_usec;
    int_usec=(int)(t_usec*1000000);
   value.it_value.tv_sec=0;
   value.it_value.tv_usec=int_usec;      value.it_interval=value.it_value;
   setitimer(ITIMER_PROF,&value,NULL);
}

int ModeInit(void)
{
   int error = EXT_NO_ERROR;
```

```
   error = ExtInit(CD);
   if (error != EXT_NO_ERROR) goto EXIT_POINT;
   printf("Succeeded in creating listening Socket by NXY\n");
EXIT_POINT:
   return(error);
} /* end ModeInit */

/* Function: ExtInit
 * Abstract:
 *  Called once at program startup to do any initialization.
 *  A socket is created to listen for
 *  connection requests from the client.  EXT_NO_ERROR is returned
*   on success, EXT_ERROR on failure.
*  NOTES:
 *   This function should not block.
 */
int ExtInit(ConnectData *UD)
{
   int              sockStatus;
   struct sockaddr_in serverAddr;
   int            sFdAddSize = sizeof(struct sockaddr_in);
   int            option    = 1;
   int            port      = 17725;
   int            error     = EXT_NO_ERROR;
   SOCKET          sFd       = INVALID_SOCKET;

#ifdef WIN32
   WSADATA data;

   if (WSAStartup((MAKEWORD(1,1)),&data)) {
      fprintf(stderr,"WSAStartup() call failed.\n");
      error = EXT_ERROR;
      goto EXIT_POINT;
   }
#endif

   /*
    * Create a TCP-based socket.
    */
   memset((char *) &serverAddr,0,sFdAddSize);
   serverAddr.sin_family     = AF_INET;
   serverAddr.sin_port       = htons(port);
   serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

   sFd = socket(AF_INET, SOCK_STREAM, 0);
   if (sFd == INVALID_SOCKET) {
      fprintf(stderr,"socket() call failed.\n");
      error = EXT_ERROR;
      goto EXIT_POINT;
   }
```

```
        /*
         * Listening socket should always use the SO_REUSEADDR option
         * ("Unix Network Programming - Networking APIs:Sockets and XTI",
         *   Volume 1, 2nd edition, by W. Richard Stevens).
         */
        sockStatus =
setsockopt(sFd,SOL_SOCKET,SO_REUSEADDR,(char*)&option,sizeof(option));
        if (sockStatus == SOCK_ERR) {
            fprintf(stderr,"setsocketopt() call failed.\n");
            error = EXT_ERROR;
            goto EXIT_POINT;
        }

        sockStatus =
            bind(sFd, (struct sockaddr *) &serverAddr, sFdAddSize);
        if (sockStatus == SOCK_ERR) {
            fprintf(stderr,"bind() call failed.\n");
            error = EXT_ERROR;
            goto EXIT_POINT;
        }

        sockStatus = listen(sFd, 1);
        if (sockStatus == SOCK_ERR) {
            fprintf(stderr,"listen() call failed.\n");
            error = EXT_ERROR;
            goto EXIT_POINT;
        }

    EXIT_POINT:
        UD->msgFd = INVALID_SOCKET;
        UD->port=17725;
        if (error == EXT_ERROR) {
            if (sFd != INVALID_SOCKET) {
                close(sFd);
            }
            UD->sFd = INVALID_SOCKET;
        } else {
            UD->sFd = sFd;
        }
        return(error);
    } /* end ExtInit */


    int OpenConnection(ConnectData *UD)
    {
        struct sockaddr_in clientAddr;
        int            sFdAddSize    = sizeof(struct sockaddr_in);

        int             error        = EXT_NO_ERROR;
        SOCKET           msgFd        = INVALID_SOCKET;
```

```
    const SOCKET      sFd          = UD->sFd;
    /*
     * Wait to accept a connection on the message socket.
     */
    msgFd = accept(sFd, (struct sockaddr *)&clientAddr,
                &sFdAddSize);
    if (msgFd == INVALID_SOCKET) {
       fprintf(stderr,"accept() for message socket failed.\n");
       error = EXT_ERROR;
       goto EXIT_POINT;
    }
    connectionMade = 1;

EXIT_POINT:
    if (error != EXT_NO_ERROR) {
       if (msgFd != INVALID_SOCKET) {
          close(msgFd);
       }

       UD->msgFd = INVALID_SOCKET;
    } else {
       UD->msgFd = msgFd;
       if(msgFd !=INVALID_SOCKET)
           printf("Succeeded in creating  socket!\n");
    }

    return(error);
}


/* Function: main
 *
 * Abstract:
 *
 */
int main(int argc, const char *argv[])
{
    int error;
   const char *option=argv[1];

   CD=(ConnectData *)malloc(sizeof(ConnectData));
    memset(CD,0,sizeof(ConnectData));
   if (strcmp(option, "-w") == 0)
       CD->waitForStart=1;
   else
        CD->waitForStart=0;
   ModeInit();

    while((CD->waitForStart)&&(connectionMade==0))
   {
       error=OpenConnection(CD);
```

```
        if(error) exit(EXIT_FAILURE);
    }

    init_sigaction();
     init_time(2.0);
     while (1);

     return(EXIT_SUCCESS);

} /* end main */
```

下面就结合本例，介绍如何在 linux(uClinux)下建立通信双方中服务器端的程序。

本例是一个服务器程序，采用流式套接字，因为流式套接字提供了一种可靠的面向连接的数据传输方法。正如它的名字所指的那样，不管是对单个的数据报，还是对于数据包，流式套接字都提供一种流式数据传输。流式套接字由 socket()函数调用来创建，而且调用时必须用 bind()函数为它分配一个地址。

在创建好一个套接字，并赋给它一个地址之后，需要用一种方法来建立和客户机的连接，为了做到这一点，要使用 listen()函数。该函数告诉套接字开始侦听客户机的连接请求。一旦将套接字设置成侦听连接后，实际的连接就可以由 accept()函数来完成。如果连接成功的接受，accept()函数将返回一个新套接字的描述符，正是由 accept()函数所创建的这个新套接字会被用作以后处理新的连接。在该例程中，ConnectData 结构体中的 msgFd 套接字就是用来真正和客户端进行通信的 socket。

原来的侦听套接字将会继续侦听新的连接请求，而新的请求可能会通过 accept()函数的再一次调用而获得接受。

到目前为止，读者已经看到有两类套接字了，一个是由 socket()函数创建的，称之为“侦听套接字”(listening socket)，另一类是由 accept()函数创建的，称之为“连接套接字”(connected socket)，它们的区别如表 7.1 所示。

表 7.1　两种套接字比较

	侦听套接字	连接套接字
创建	socket()	accept()
应用	bind(),listen(),accept()	文件读写调用 read(),write() 网络文件专用函数 send(),recv()
作用	监听来自客户端的连接请求，并建立连接	与某一个客户进程连接，完成具体的数据传输工作
生存周期	一个服务器进程与一个监听套接字相对应，与服务器进程同时存在或消灭	一次连接对应一个连接套接字，建立连接时创建套接字，连接结束时关闭

在主程序中，利用信号处理函数，进行每隔 2 秒定时的往客户端发送字符串(有关信号处理函数的知识，在本节开始已有介绍)。

网络应用程序包括两个部分：一部分是服务器端的应用程序，主要是用于接受客户端的连接请求，接收客户端的信息，处理客户端的计算请求，向客户端发送计算结果和应答信息等。另一部分就是客户端应用程序，主要用于申请连接到服务器，向服务器发送计算请求，处理服务器发回的计算结果和其他信息。

本书所给的例子只是服务器端的应用程序，对于客户端程序，在此只为读者做一个简单的介绍。

在客户端的应用程序为了让服务器接收一个连接请求，必须首先也要建立一个 socket，一般也是使用流式套接字，接着发起一个请求，通过调用函数 connect()来实现。

一旦客户机套接字和服务器套接字建立了连接，双方就可以通过 send()和 recv()函数的调用来发送和接收数据了。

流式套接字基本使用方法如图 7.9 所示。

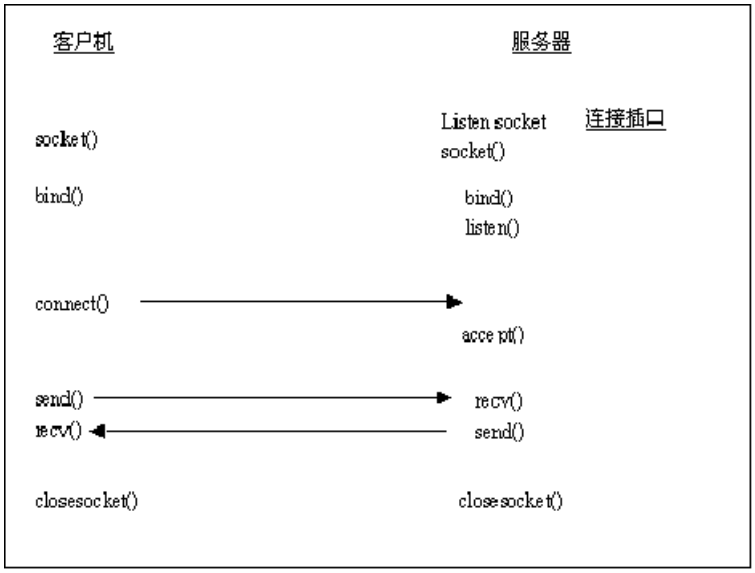


图 7.9　流式套接字用法

如果想断开连接，调用函数 close()真正释放和套接字相关的系统资源。

7.4.3　添加用户应用程序到 uClinux

以下通过一个具体实例向读者介绍将程序添加到 uClinux 的标准方法。

例如要把前面提到的源程序 lednxy.c 添加到运行于目标板上的 uClinux 操作系统中，则该文件应在目录/home/nie/uClinux-Samsung/user 下，进入 uClinux-Samsung/user 目录并建立一个自己的子目录，比如键入：

mkdir myapp，

这样在 user 目录下就建立了一个新的子目录 myapp，把 lednxy.c 拷贝到 myapp 目录下，并将该源文件相应的 makefile 文件也拷贝到该目录下。注意，为了使用标准方法，我们应该修改一下刚才的 makefile 文件，这个文件名应为 Makefile，写成这样的形式：

```
EXEC = lednxy
OBJS = lednxy.o
all: $(EXEC)
$(EXEC): $(OBJS)
	$(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS)
romfs:
	$(ROMFSINST) /bin/$(EXEC)
clean:
	rm -f $(EXEC) *.elf *.gdb *.o
```

进入 user 目录，增加一行语句到该目录下的 Makefile 文件中，

dir_$(CONFIG_USER_MYAPP_LEDNXY)　　+= myapp

该语句的作用是让编译器可以访问到我们所创建的 myapp 目录下的 makefile 文件，保存后退出。

切换到目录/home/nie/uClinux-Samsung/config 下，编辑 Configure.help 文件，即输入一下命令

```
cd  ../config
vi  Configure.help
```

这是一个包含了在配置的时候出现的所有文本信息的文件。在这个文件中加入类似下面的语句块：

CONFIG_USER_ MYAPP_LEDNXY

This program is an example.

注意第二行文本信息必须要空两格开始。每行的字符要小于 70 个。添加完毕后，保存退出。

不过，用户也可不必修改该文件，因为它仅仅是提供一个在线文本信息显示的功能，对于添加用户程序到uCllinux影响不大。

接下来需要修改 uClinux 系统中对编译器来讲比较重要的一个文件 config.in。

仍然是在 config 目录下，打开该文件，在最后增加类似下面的语句：

```
#########################################################################

  mainmenu_option  next_comment
  comment  ' My Application '

 bool 'lednxy'          CONFIG_USER_MYAPP_LEDNXY
comment  ' My Application'

endmenu

#########################################################################
```

现在我们已经把要做的修改的相关工作完成了，接下来需要进行内核的编译工作，按照在 7.3.3 中谈到的编译 uClinux 内核的步骤进行就可以了。

值得注意的一点是在第一步 make menuconfig 进行内核配置的时候，在 Target Platform Selection ,要选中 Customize Vendor/User Settings (NEW) 如图 7.10 所示，选中了该选项后，与最初我们配置内核过程不同的是，它还会在 make menuconfig 的最后出现如图 7.11 所示对话框，让你进行用户应用程序的配置，在对话框里出现的文字是在 config.in 文件中添加的文字，选中要编译的应用程序所在路径，就会出现如图 7.12 所示的对话框，显示所选中目录下的，在 config.in 中所设定的应用程序文件名，选中要编译的文件名，保存好内核配置后退出。用这种方法生成的可执行文件在 romfs/bin 下。

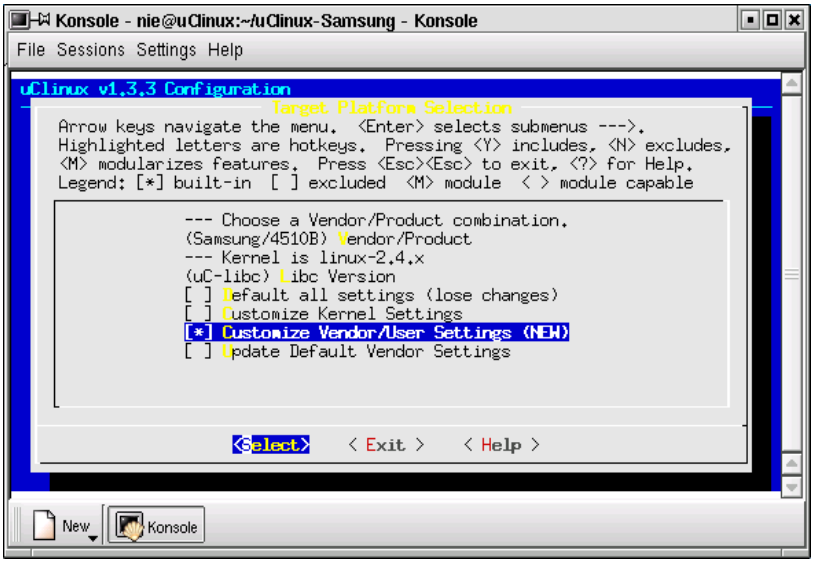


图 7. 10　添加用户应用程序配置

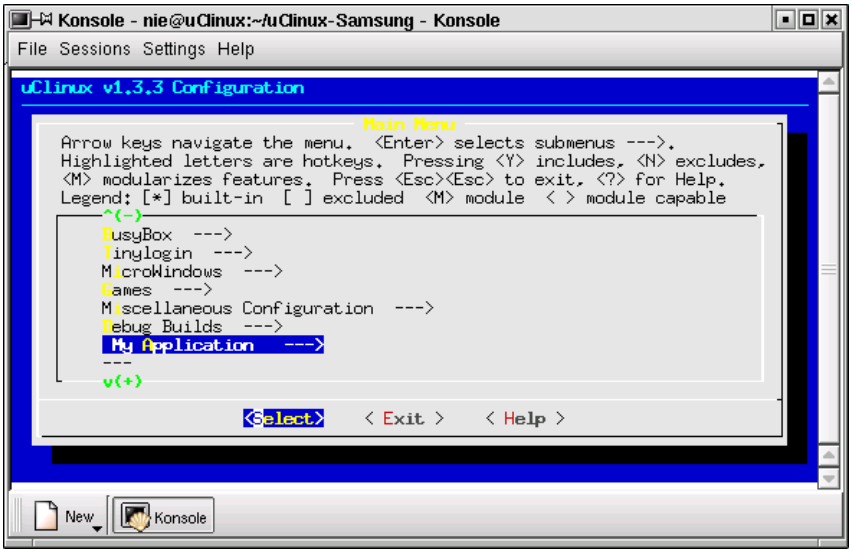


图 7.11 选择要配置的用户应用程序

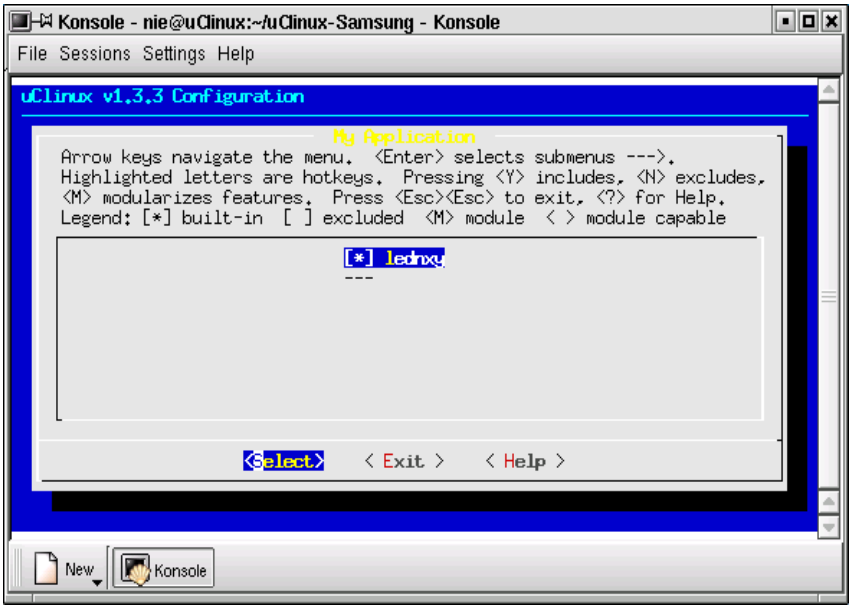


图 7.12 选择要编译的源文件

当用户应用程序做了修改后，需要重新编译内核，但是此时只要进行内核编译的后四步，即从 make user_only 开始即可，不必再从内核配置开始了。

以上介绍的是一种基本的添加用户应用程序的方法，如果读者觉得比较麻烦，还可以使用下面一种较为简单的方法，这种方式是将用户的应用程序作为 uClinux 自身的应用程序对待，在内核编译时一起完成。

在 uClinux-Samsung/romfs/usr 下面编写用户应用程序源代码以及它对应的 makefile 文件，就在该目录下编译这个 makefile 文件，将生成的可执行文件拷贝到 uClinux-Samsung/romfs/bin 下。

进行内核的部分编译工作，用这种方法只需要做编译内核的最后三步工作，即：

```
make romfs
make iamge
make
```

最后都把在 uClinux-Samsung/images 下生成的 image.rom 文件烧写到系统的 FLASH 存储器中，uClinux 启动后，用户的应用程序在/bin 目录下，此时可运行用户程序。

在 Windows 环境中，可以使用超级终端建立串口与目标硬件连接。超级终端的一些端口属性需

要设置，该内核默认的端口设置为：COM1，波特率为 19200，数据位为 8，无校验，停止位为 1，无流控。通过超级终端可以看到整个 uClinux 的启动过程。

对于本例，在 uClinux 启动后，从超级终端中键入 cd bin，进入到 bin 目录下，运行 lednxy 程序，可以看到该程序对两个 LED 显示器的控制效果。

上面介绍的方法中，在将用户应用程序添加到 uClinux 内核运行时，都需要对内核进行部分或全部的编译，每次对内核编译完成后，都要先将 FLASH 存储器中的内容擦除，然后重新烧写新编译好的内核到 FLASH 存储器中去，这对于程序开发来说，是非常不方便的。下面介绍一种通过网络来传输可执行文件，避免每次测试程序运行效果时都要编译一次内核。

7.4.4 通过网络添加应用程序到目标系统

作为一款优秀的网路控制器，基于 S3C4510B 的系统一般都提供以太网接口，通过以太网接口从网络添加用户程序到目标系统运行，显然比前面所介绍的方法方便得多，特别是在用户应用程序的调试过程中，若每做一点修改都要求重新编译内核并烧写入 FLASH 存储器运行，其工作量是可想而知的。

事实上，鉴于 uClinux 操作系统本身强大的网络功能，同时基于 S3C4510B 的系统提供以太网接口，通过局域网可方便的在运行 uClinux 目标系统和运行 Linux 宿主机上进行文件传输。运行目标系统的 uClinux 内核在编译的过程中，已默认选择了 FTP 和其他一些网络服务，同时，宿主机上的 Linux 在默认时，也会安装运行 FTP 服务，因此，当目标系统的 uClinux 启动运行以后，可将目标系统作为 FTP 客户端，而运行 Linux 宿主机作为 FTP 服务器，进行双向的文件传输。

但由于目前所使用的 uClinux 操作系统内核采用 ROMFS 作为其根文件系统，当目标系统的 uClinux 启动运行以后，其目录大多数是建在 FLASH 存储器中，因而是不可写的，只有/var、/tmp 等少数几个目录是建立在 SDRAM，是可读写的，但若目标系统掉电，内容就丢失了，因此只能作为应用程序调试之用，当应用程序调试完成后，还应将其写入 FLASH 存储器。当然，若能在目标系统中使用 JFFS/JFFS2,用以代替 ROMFS 作为其根文件系统，则整个目标系统就像有磁盘一样方便，用户应用程序的加载再也不用像前面介绍的方式进行了。关于 JFFS/JFFS2 文件系统的建立，请读者参考相关技术资料，在此仅描述如何将用户程序通过局域网，从 FTP 服务器(运行 Linux 宿主机)上，传输到运行 uClinux 的目标系统(FTP 客户机)并执行的过程：

将目标系统与 Linux 宿主机连接在同一网段中，在宿主机的任意目录下编写应用程序，并用交叉编译工具生成 flat 格式的文件。

启动目标系统的 uClinux，通过超级终端，输入下面的命令：

ifconfig　eth0 192.168.100.50

ifconfig 命令用于显示及设置目标系统的网卡配置，例如，IP 地址，子网掩码，IRQ 及 IO Port 等。在上述命令中，参数 eth0 代表目标系统的网络设备，IP 地址 192.168.100.50 为目标系统的 IP 地址，注意应与宿主机在同一网段内(此时宿主机的 IP 地址为：192.1681.100.21)。

执行命令：

ifconfig　–all

可以看到目标系统的 IP 地址已被正确配置，显示信息如下：

```
/var/tmp> ifconfig -all
eth0     Link encap:Ethernet  HWaddr 00:40:95:36:35:34
         inet addr:192.168.100.52  Bcast:192.168.100.255  Mask:255.255.255.0
         UP BROADCAST RUNNING  MTU:1500  Metric:1
         RX packets:30533 errors:10 dropped:0 overruns:0 frame:0
         TX packets:21090 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:100
         Interrupt:17
```

```
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:19 errors:0 dropped:0 overruns:0 frame:0
          TX packets:19 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

这里我们介绍几个比较重要的名词。

Link encap 即 Link encapsulate，是用来表示将信息分割为数据包的方法，比如 Ethernet。

Hwaddr 即 Hardware address，网卡的硬件地址，又称为 MAC(Media Access Control)地址，它是直接烧写到网卡芯片上的。由 12 个 16 进制值组成，每两个数字为一组，每组之间用“：”分割开。

inet addr 即 internet address，就是主机的 IP 地址。

Bcast 即 Broadcast，这个是指广播地址，若接收者的地址为广播地址，则表示该信息可在同一时间发送到网络中的所有计算机。通常，广播地址是由主机的 IP 地址所属的地址类来决定，

Mask 是子网掩码，主要是用来将 IP 地址分成网络 ID 和主机 ID 两部分。它是由一连串的“1”和一连串的“0”组成。“1”对应于网络号码和子网号码字段，而“0”对应于主机号码字段。对于不同类的 IP 地址，对应的子网掩码是不同的。表 7.2 是不同类的 IP 地址使用范围，表 7.3 是不同类的 IP 地址所使用的子网掩码。

表 7.2　IP 地址使用范围

网络类别	最大网络数目	第一个可用的网络号码	最后一个可用的网络号码	每个网络中的最大主机数
A	126	1	126	16777214
B	16382	128.1	191.254	65534
C	2097150	192.0.1	223.255.254	254

表 7.3　不同类的 IP 地址使用的子网掩码

Class IP	子网掩码
A	255.0.0.0
B	255.255.0.0
C	255.255.255.0

子网掩码和 IP 地址转换为二进制数后，将两者相“与”，相与之后得到的结果就是网络 ID。

MTU 即 Maximum Transmission Unit，网络传输时，数据包最大的传输单位，Ethernet 的 MTU 默认值是 1500 字节。

Metric 来源主机将信息送至目的地主机，所需经过的转送次数，有些路由通信协议，在计算最短路径时，必须参考此数值。

RX 表示已经接收的数据包总数，数据包流失数量以及碰撞的数量。

TX 表示已经发送的数据包总数，数据包流失数量以及碰撞的数量。

测试一下与宿主机的连接,键入命令：

ping 192.168.100.21

应能得到宿主机的应答信息，类似如下所示：

```
/var/tmp> ping 192.168.100.21
PING 192.168.100.21 (192.168.100.21): 56 data bytes
64 bytes from 192.168.100.21: icmp_seq=0 ttl=255 time=0.0 ms
64 bytes from 192.168.100.21: icmp_seq=1 ttl=255 time=0.0 ms
64 bytes from 192.168.100.21: icmp_seq=2 ttl=255 time=0.0 ms
64 bytes from 192.168.100.21: icmp_seq=3 ttl=255 time=0.0 ms
64 bytes from 192.168.100.21: icmp_seq=4 ttl=255 time=0.0 ms
```

当目标系统与宿主机已正确建立连接后，进入目标系统的可写目录/var 或/tmp，并登录到宿主机：

cd　var

ftp　192.168.100.21

此时输入宿主机上的合法用户名及密码，便与宿主机建立了 FTP 连接。为传输二进制文件，键入命令：

ftp>binary

200 Type set to I

这里请读者注意：在用 FTP 进行文件传输的时候，一定要选好文件传输的模式，FTP 缺省模式为二进制模式，但是为了保险起见，还是手动把传输模式显式改写为二进制模式。通常的 txt，html 和绝大多数 PS 文件都是文本格式的，而其他的可执行文件，压缩文件都是二进制格式。两种格式之间要用 ascii 和 binary 命令切换，两者都可以适当缩写。

传输已编译好的可执行文件（如 lednxy）到目标系统并退出 FTP 服务，键入如下所示命令：

ftp>get　lednxy

ftp>bye

此时，可执行文件 lednxy 已传输至目录/var 或/tmp 下，但文件的可执行属性未被设置，添加文件的可执行属性，键入命令：

chmod　755　lednxy

chmod 是一个文件权限修改的命令，在文件创建的时候会自动设置存取权限，若是这些默认权限无法适合企业环境的需求，就可以利用 chmod 命令来修改存取权限。通常在权限修改的时候可以用两种方式表示权限类，数字表示法和文字表示法。

这里我们采用的是数字表示法，就是说将读取(r)，写入(w)和执行(x)分别以 4，2，1 来代表，没有授予的权限的部分就表示值为 0，然后再把所授予的权限先加而成。表 7.4 为读者列出了几个例子。

表 7.4　存取权限范例

原始权限	转换为数字	数字表示法
rwxrwxr-x	(421) (421)(421)	775
rwxr-xr-x	(421)(401)(401)	755
rw-rw-r--	(420)(420)(400)	664
rw-r--r--	(420)(400)(400)	644

每三位字符为一组，这样权限可以被分为三组，第一组表示此文件拥有者的存取权限，第二组表示该文件拥有者所属组成员的存取权限，最后一组表示该文件拥有者所属组之外的用户存取权限。希望读者能够研究清楚权限的分配。

这里所键入的命令表示授予文件拥有者读取写入和执行的权限，而该文件拥有者所属组成员和该文件拥有者所属组之外的用户只拥有读取和执行的权限。

执行程序 lednxy，键入命令：

./lednxy

显然，这种方法在应用程序的开发中是比较有用的，使用这种方式可以使用户在宿主机的开发环境下，编译代码，但编译出来的 flat 格式的文件并没有放到硬件目标系统的 FLASH 存储器，而是在系统的 SDRAM 中运行，这就大大节省了调试的时间，内核编译只需要进行一次，使开发人员能够将更多的精力投入到应用程序的开发中来。

7.5　本章小结

本章主要介绍了 uClinux 嵌入式操作系统的基本概况，和在 Linux 以及 Windows 下如何开发 uClinux 用户应用程序等内容。首先是 uClinux 的一些基本概念的介绍，接着通过实例介绍了 gnu

gcc,make,gdb 等一系列 gnu 工具的使用。然后，详细介绍了如何在 Linux 和 Windows 下建立交叉编译环境，以及如何编译 uClinux 内核的过程。通过几个简单的程序，使读者对在 uClinux 下编写串行通信程序有一个初步的了解。网络编程也是本章向读者介绍的一个重点，通过给出读者一个运行在服务器端的程序，让读者熟悉套接字的概念，分类和通常的 client/server 模式的程序框架。简略的提到了一些有关网络方面的知识。最后通过具体的例程为读者讲述了如何将用户应用程序加入到 uClinux 操作系统中运行的方法。

感兴趣的读者可以按照本章所介绍的步骤，搭建一个适宜于自己开发环境平台的交叉环境，并以本章所给的例子作为开发自己应用程序的起点，熟练掌握如何在 uClinux 下如何开发自己的嵌入式系统应用程序。

第8章 ARM ADS 集成开发环境的使用

在这一章里，将介绍 ARM 开发软件 ADS(ARM Developer Suite)。通过学习如何在 CodeWarrior IDE 集成开发环境下编写，编译一个工程的例子，使读者能够掌握在 ADS 软件平台下开发用户应用程序。本章还描述了如何使用 AXD 调试工程，使读者对于调试工程有个初步的理解，为进一步的使用和掌握调试工具起到抛砖引玉的作用。

本章主要内容有：

- ADS 软件组成介绍
- 使用 ADS 创建工程
- 用 AXD 进行代码调试

8.1 ADS 集成开发环境组成介绍

ARM ADS 全称为 ARM Developer Suite。是 ARM 公司推出的新一代 ARM 集成开发工具。现在 ADS 的最新版本是 1.2，它取代了早期的 ADS1.1 和 ADS1.0。它除了可以安装在 Windows NT4，Windows 2000，Windows 98 和 Windows 95 操作系统下，还支持 Windows XP 和 Windows Me 操作系统。

ADS 由命令行开发工具，ARM 时实库，GUI 开发环境(Code Warrior 和 AXD)，实用程序和支持软件组成。 有了这些部件，用户就可以为 ARM 系列的 RISC 处理器编写和调试自己的开发应用程序了。

下面就详细介绍一下 ADS 的各个组成部分。

8.1.1 命令行开发工具

这些工具完成将源代码编译，链接成可执行代码的功能。

ADS 提供下面的命令行开发工具：

armcc

armcc 是 ARM C 编译器。这个编译器通过了 Plum Hall C Validation Suite 为 ANSI C 的一致性测试。armcc 用于将用 ANSI C 编写的程序编译成 32 位 ARM 指令代码。

因为 armcc 是我们最常用的编译器，所以对此作一个详细的介绍。

在命令控制台环境下，输入命令：

armcc –help

可以查看 armcc 的语法格式以及最常用的一些操作选项

armcc 最基本的用法为： armcc [options] file1 file2 ... filen

这里的 option 是编译器所需要的选项，fiel1,file2…filen 是相关的文件名。

这里简单介绍一些最常用的操作选项。

-c：表示只进行编译不链接文件；

-C：(注意：这是大写的 C)禁止预编译器将注释行移走；

-D<symbol>：定义预处理宏，相当于在源程序开头使用了宏定义语句#define symbol ，

这里 symbol 默认为 1；

-E：仅仅是对 C 源代码进行预处理就停止；

-g<options>：指定是否在生成的目标文件中包含调试信息表；

-I<directory>：将 directory 所指的路径添加到#include 的搜索路径列表中去； -J<directory>：用 directory 所指的路径代替默认的对#include 的搜索路径；

-o<file>：指定编译器最终生成的输出文件名。

-O0：不优化；

-O1：这是控制代码优化的编译选项，大写字母 O 后面跟的数字不同，表示的优化级别就不同，-O1 关闭了影响调试结果的优化功能；

-O2：该优化级别提供了最大的优化功能；

-S：对源程序进行预处理和编译，自动生成汇编文件而不是目标文件；

-U<symbol>：取消预处理宏名，相当于在源文件开头，使用语句#undef symbol;

-W<options>：关闭所有的或被选择的警告信息；

有关更详细的选项说明，读者可查看 ADS 软件的在线帮助文件。

armcpp

armcpp 是 ARM C++编译器。它将 ISO C++ 或 EC++ 编译成 32 位 ARM 指令代码。

tcc

tcc 是 Thumb C 编译器。该编译器通过了 Plum Hall C Validation Suite 为 ANSI 一致性的测试。tcc 将 ANSI C 源代码编译成 16 位的 Thumb 指令代码。

tcpp

tcpp 是 Thumb C++ 编译器。 它将 ISO C++ 和 EC++ 源码编译成 16 位 Thumb 指令代码。

armasm

armasm 是 ARM 和 Thumb 的汇编器. 它对用 ARM 汇编语言和 Thumb 汇编语言写的源代码进行汇编。

armlink

armlink 是 ARM 连接器。该命令既可以将编译得到的一个或多个目标文件和相关的一个或多个库文件进行链接，生成一个可执行文件，也可以将多个目标文件部分链接成一个目标文件，以供进一步的链接。ARM 链接器生成的是 ELF 格式的可执行映像文件。

armsd

armsd 是 ARM 和 Thumb 的符号调试器。它能够进行源码级的程序调试。用户可以在用 C 或汇编语言写的代码中进行单步调试，设置断点，查看变量值和内存单元的内容。

8.1.1.1 armcc 用法详解

下面为读者介绍上述的 4 种 ARM C 和 C++编译器的命令通用语法。

compiler [PCS-options] [source-language] [search-paths] [preprocessor-options] [output-format] [target-options] [debug-options] [code-generation-options] [warning-options] [additional-checks] [error-options] [source]

用户可以通过命令行操作选项控制编译器的执行。所有的选项都是以符号"-"开始，有些选项后面还跟有参数。在大多数情况下，ARM C 和 C++编译器允许在选项和参数之间存在空格。

命令行中各个选项出现顺序可以任意。

这里的 compiler 是指 armcc，tcc， armcpp 和 tcpp 中的一个；

PCS-options：指定了要使用的过程调用标准；

source-language：指定了编译器可以接受的编写源程序的语言种类。对于 C 编译器默认的语言是 ANSI C，对于 C++编译器默认是 ISO 标准 C++；

search-paths：该选项指定了对包含的文件(包括源文件和头文件)的搜索路径；

preprocessor-options：该选项指定了预处理器的行为，其中包括预处理器的输出和宏定义等特性；

output-format：该选项指定了编译器的输出格式，可以使用该项生成汇编语言输出列表文件和目标文件；

target-options：该选项指定目标处理器或 ARM 体系结构；

debug-options：该选项指定调试信息表是否生成，和该调试信息表生成时的格式；

code-generation-options：该选项指定了例如优化，字节顺序和由编译器产生的数据对齐格式等

选项；

warning-options：该选项决定警告信息是否产生；

additional-checks：该选项指定了几个能用于源码的附加检查，例如检查数据流异常，检查没有使用的声明等；

error-options：该选项可以关闭指定的可恢复的错误，或者将一些指定的错误降级为警告；

source：该选项提供了包含有 C 或 C++源代码的一个或多个文件名，默认的，编译器在当前路径寻找源文件和创建输出文件。如果源文件是用汇编语言编写的(也就是说该文件的文件名是以.s 作为扩展名)，汇编器将被调用来处理这些源文件。

如果操作系统对命令行的长度有限制，可以使用下面的操作，从文件中读取另外的命令行选项：

-via filename

该命令打开文件名为 filename 的文件，并从中读取命令行选项。用户可以对-via 进行嵌套调用，亦即，在文件 filename 中又通过-via finlename2 包含了另外一个文件。

在下面的例子中，从 input.txt 文件中读取指定的选项，作为 armcpp 的操作选项：

armcpp -via input.txt source.c

以上是对编译器选项的一个简单概述。它们(包括后面还要介绍的其他一些命令工具)既可以在命令控制台环境下使用，同时由于它们被嵌入到了 ADS 的图形界面中，所以也可以在图形界面下使用。

8.1.1.2 armlink 用法详解

在介绍 armlink 的使用方法之前，先介绍要涉及到的一些术语。

映像文件(image)：是指一个可执行文件，在执行的时候被加载到处理器中。一个映像文件有多个线程。它是 ELF(Executable and linking format)格式的。

段(Section)：描述映像文件的代码或数据块。

RO：是 Read-only 的简写形式。

RW：是 Read-write.的简写形式。

ZI：是 Zero-initialized 的简写形式。

输入段(input section)：它包含着代码，初始化数据或描述了在应用程序运行之前必须要初始化为 0 的一段内存。

输出段(output section)：它包含了一系列具有相同的 RO，RW 或 ZI 属性的输入段。

域(Regions)：在一个映像文件中，一个域包含了 1 至 3 个输出段。多个域组织在一起，就构成了最终的映像文件。

Read Only Position Independent(ROPI)：它是指一个段，在这个段中代码和只读数据的地址在运行时候可以改变。

Read Write Position Independent(RWPI)：它是指一个段，在该段中的可读/写的数据地址在运行期间可以改变。

加载时地址：是指映像文件位于存储器(在该映像文件没有运行时)中的地址。

运行时地址：是指映像文件在运行时的地址。

下面介绍一下 armlink 命令的语法

完整的连接器命令语法如下：

armlink [-help] [-vsn] [-partial] [-output file] [-elf] [-reloc][-ro-base address] [-ropi]

[-rw-base address] [-rwpi] [-split]

[-scatter file][-debug|-nodebug][-remove?RO/RW/ZI/DBG]|-noremove] [-entry location]

[-keep section-id] [-first section-id] [-last section-id] [-libpath pathlist] [-scanlib|-noscanlib] [-locals|-nolocals] [-callgraph] [-info topics] [-map] [-symbols] [-symdefs file] [-edit file] [-xref] [-xreffrom object(section)] [-xrefto object(section)] [-errors file] [-list file] [-verbose]

[-unmangled |-mangled] [-match crossmangled][-via file] [-strict]

[-unresolved symbol][-MI|-LI|-BI] [input-file-list]

上面各选项的含义分别为：

-help

这个选项会列出在命令行中常用的一些选项操作。

-vsn

这个选项显示出所用的 armlink 的版本信息。

-partial

用这个选项创建的是部分链接的目标文件而不是可执行映像文件。

-output file

这个选项指定了输出文件名，该文件可能是部分链接的目标文件，也可能是可执行映像文件。如果输出文件名没有特别指定的话，armlink 将使用下面的默认：

如果输出是一个可执行映像文件，则生成的输出文件名为__image.axf；

如果输出是一个部分链接的目标文件，在生成的文件名为__object.o；

如果没有指定输出文件的路径信息，则输出文件就在当前目录下生成。如果指定了路径信息，则所指定的路径成为输出文件的当前路径。

-elf

这个选项生成 ELF 格式的映像文件，这也是 armlink 所支持的唯一的一种输出格式，这是默认选项。

-reloc

这个选项生成可重定址的映像。

一个可重定址的映像具有动态的段，这个段中包含可重定址信息，利用这些信息可以在链接后，进行映像文件的重新定址；

-reloc，-rw-base 一起使用，但是如果没有-split 选项，链接时会产生错误。

-ro-base address

这个选项将包含有 RO(Read-Only 属性)输出段的加载地址和运行地址设置为 address，该地址必须是字对齐的，如果没有指定这个选项，则默认的 RO 基地址值为 0x8000。

-ropi

这个选项使得包含有 RO 输出段的加载域和运行域是位置无关的。如果该选项没有使用，则相应的域被标记为绝对的。通常每一个只读属性的输入段必须是只读位置无关的。如果使用了这个选项，armlink 将会进行以下操作：

检查各段之间的重定址是否有效；

确保任何由 armlink 自身生成的代码是只读位置无关的。

这里希望读者注意的是，ARM 工具直到 armlink 完成了对输入段的处理后，才能够决定最终的生成映像是否为只读位置无关的。这就意味着 ，即使为编译器和汇编器指定了 ROPI 选项，armlink 也可能会产生 ROPI 错误信息。

-rw-base address

这个选项设置包含 RW(Read/Write 属性)输出段的域的运行时地址，该地址必须是字对齐的。

如果这个选项和-split 选项一起使用，将设置包含 RW 输出段的域的加载和运行时地址都设置在 address 处。

-rwpi

这个选项使得包含有 RW 和 ZI(Zero Initialization，初始化为 0)属性的输出段的加载和运行时域为位置无关的。如果该选项没有使用，相应域标记为绝对的。这个选项要求-rw-base 选项后有值，如果-rw-base 没有指定的话，默认其值为 0，即相当于-rw-base 0。通常每一个可写的输入段必须是可读/ 可写的位置无关的。

如果使用了该选项，armlink 会进行以下的操作：

检查可读/可写属性的运行域的输入段是否设置了位置无关属性；

检查在各段之间的重定址是否有效；

生成基于静态寄存器 sb 的条目，这些在 RO 和 RW 域被拷贝和初始化的时候会用到。

编译器并不会强制可写的数据一定要为位置无关的，这就是说，即使在为编译器和汇编器指定了 RWPI 选项，armlink 也可能生成数据不是 RWPI 的信息。

-split

这个选项将包含 RO 和 RW 属性的输出段的加载域，分割成 2 个加载域。一个是包含 RO 输出段的加载域，默认的加载地址为 0x8000，但是可以用-ro-base 选项设置其他的地址值，另一个加载域包含 RO 属性的输出段，由-rw-base 选项指定加载地址，如果没有使用-rw-base 选项的话，默认使用的是-rw-base 0。

-scatter file

这个选项使用在 file 中包含的分组和定位信息来创建映像内存映射。

注意，如果使用了该选项的话，必须要重新实现堆栈初始化函数__user_initial_stackheap()。

-debug

这个选项使输出文件包含调试信息，调试信息包括，调试输入段，符号和字符串表。这是默认的选项。

-nodebug

这个选项使得在输出文件中不包含调试信息。生成的映像文件短小，但是不能进行源码级的调试。armlink 对在输入的目标文件和库函数中发现的任何调试输入段都不予处理，当加载映像文件到调试器中的时候，也不包含符号和字符串信息表。这个选项仅仅是对装载到调试器的映像文件的大小有影响，但是对要下载到目标板上的二进制代码的大小没有任何影响。

如果用 armlink 进行部分链接生成目标文件而不是映像文件，则虽然在生成的目标文件中不含有调试输入段，但是会包含符号和字符串信息表。

这里特别请读者注意的是：

如果要在链接完成后使用 fromELF 工具的话，不可使用-nodebug 选项，这是因为如果生成的映像文件中不包含调试信息的话，则有下面的影响：

fromELF 不能将映像文件转换成其他格式的文件；

fromELF 不能生成有意义的反汇编列表。

-remove (RO/RW/ZI/DBG)

使用这个选项会将在输入段未使用的段从映像文件中删除。如果输入段中含有映像文件入口点或者该输入段被一个使用的段所引用，则这样的输入段会当作已使用的段。

在使用这个选项时候要注意，不要删除异常处理函数。使用-keep 选项来标识异常处理函数，或用 ENTRY 伪指令标明是入口点。

为了更精确的控制删除未使用的段，可以使用段属性限制符。可以使用以下的段属性限制符：

RO

删除所有未使用的 RO 属性的段；

RW

删除所有未使用的 RW 属性的段；

ZI

删除所有未使用的 ZI 属性的段；

DBG

删除所有未使用的 DEBUG 属性的段。

这些限制符出现的顺序是任意的，但是它们必须要有”( )”括住，多个限制符之间要用符号”/”进行间隔。ADS 软件中默认选项是-remove (RO/RW/ZI/DBG)。

如果没有指定段属性限制符，则所有未使用的段都会被删除。因为-remove 就等价于-remove(RO/RW/ZI/DBG)选项。

-noremove

这个选项保留映像文件中所有未被使用的段。

-entry location

这个选项指定映像文件中唯一的初始化入口点。一个映像文件可以包含多个入口点，使用这个命令定义的初始化入口点是存放在可执行文件的头部，以供加载程序加载时使用。当一个映像文件被装载时，ARM 调试器使用这个入口点地址来初始化 PC 指针。初始化入口点必须满足下面的条件：

映像文件的入口点必须位于运行域内；

运行域必须是非覆盖的，并且必须是固定域(就是说，加载域和运行域的地址相同)。

在这里可以用以下的参数代替 location 参数：

1. 入口点地址：这是一个数值，例如-entry 0x0；
2. 符号：该选项指定映像文件的入口点为该符号所代表的地址处，比如：

-entry int_handler

表示程序入口点在符号 int_handler 所在处。

如果该符号有多处定义存在，armlink 将产生出错信息。

offset+object(section)：该选项指定在某个目标文件的段的内部的某个偏移量处为映像文件的入口地址，例如：

-entry 8+startup(startupseg)

如果偏移量值为 0，可以简写成 object(section)，如果输入段只有一个，则可以简化为 object。

-keep section-id

使用该选项，可以指定保留一个输入段，这样的话，即使该输入段没有在映像文件中使用，也不会被删除。参数 section-id 取下面一些格式：

1． symbol

该选项指定定义 symbol 的输入段不会在删除未使用的段时被删除。如果映像文件中有多处 symbol 定义存在，则所有包含 symbol 定义的输入段都不会被删除。例如：

-keep int_handler

则所有定义 int_handler 的符号的段都会保留，而不被删除。

为了保留所有含有以_handler 结尾的符号的段，可以使用如下的选项：

-keep *_handler

2． object(section)

这个选项指定了在删除未使用段时，保留目标文件中的 section 段。输入段和目标名是不区分大小写的，例如，为了在目标文件 vectors.o 中保留 vect 段，使用：

-keep vectors.o(vect)

为了保留 vectors.o 中的所有以 vec 开头的段名，可以使用选项：

-keep vectors.o(vec*)

3． object

这个选项指定在删除未使用段时，保留该目标文件唯一的输入段。目标名是不区分大小写的，如果使用这个选项的时候，目标文件中所含的输入段不止一个的话，armlink 会给出出错信息。

比如，为了保留每一个以 dsp 开头的只含有唯一输入段的目标文件，可以使用如下的选项：

-keep dsp*.o

-first section-id

这个选项将被选择的输入段放在运行域的开始。通过该选项，将包含复位和中断向量地址的段放置在映像文件的开始，可以用下面的参数代替 section-id:

1．symbol

选择定义 symbol 的段。禁止指定在多处定义的 symbol，因为多个段不能同时放在映像文件的开始。

2．object(section)

从目标文件中选择段放在映像文件的开始位置。在目标文件和括号之间不允许存在空格，例如

-first init.o(init)

3．object

选择只有一个输入段的目标文件。如果这个目标文件包含多个输入段，armlink 会产生错误信息。用这个选项的例子如下：

-first init.o

这里希望读者注意的是：

使用-first 不能改变在域中按照 RO 段放在开始，接着放置 RW 段，最后放置 ZI 段的基本属性排放顺序。如果一个域含有 RO 段，则 RW 或 ZI 段就不能放在映像文件的开头。类似地，如果一个域有 RO 或 RW 段，则 ZI 段就不能放在文件开头。

两个不同的段不能放在同一个运行时域的开头，所以使用该选项的时候只允许将一个段放在映像文件的开头。

-last section-id

这个选项将所选择的输入段放在运行域的最后。例如，用这个选项能够强制性的将包含校验和的输入段放置在 RW 段的最后。使用下面的参数可以替换 section-id。

1. symbol

选择定义 symbol 的段放置在运行域的最后。不能指定一个有多处定义的 symbol。使用该参数的例子如下：

-last checksum

2. object(section)

从目标文件中选择 section 段。在目标文件和后面的括号间不能有空格，用该参数的例子为：

-last checksum.o(check)

3. object

选择只有一个输入段的目标，如果该目标文件中有多个输入段，armlink 会给出出错信息。

和-first 选项一样，需要读者注意的是;

使用-last 选项不能改变在域中将 RO 段放在开始，接着放置 RW 段，最后放置 ZI 段的输出段基本的排放顺序。如果一个域含有 ZI 段，则 RW 段不能放在最后，如果一个域含有 RW 或 ZI 段，则 RO 段不能放在最后。

在同一个运行域中，两个不同的段不能同时放在域的最后位置。

-libpath pathlist

这个选项为 ARM 标准的 C 和 C++库指定了搜索路径列表。

注意，这个选项不会影响对用户库的搜索路径。

这个选项覆盖了环境变量 ARMLIB 所指定的路径。参数 pathlist 是一个以逗号分开的多个路径列表，即为 path1, path2,... pathn，这个路径列表只是用来搜索要用到的 ARM 库函数。默认的，对于包含 ARM 库函数的默认路径是由环境变量 ARMLIB 所指定的。

-scanlib

这个选项启动对默认库(标准 ARM C 和 C++库)的扫描以解析引用的符号。这个选项是默认的设置。

-noscanlib

该选项禁止在链接时候扫描默认的库。

-locals

这个选项指导链接器在生成一个可执行映像文件的时候，将本地符号添加到输出符号信息表中。该选项是默认设置。

-nolocals

这个选项指导链接器在生成一个可执行映像文件的时候，不要将本地符号添加到输出符号信息表中。如果想减小输出符号表的大小，可以使用该选项。

-callgraph

该选项创建一个 HTML 格式的静态函数调用图。这个调用图给出了映像文件中所有函数的定义和引用信息。对于每一个函数它列出了：

1. 函数编译时候的处理器状态(ARM 状态还是 Thumb 状态)；
2. 调用 func 函数的集合；
3. 被 func 调用的函数的集合；
4. 在映像文件中使用的 func 寻址的次数。

此外，调用图还标识了下面的函数：

1. 被 interworking veneers 所调用的函数；
2. 在映像文件外部定义的函数；
3. 允许未被定义的函数(以 weak 方式的引用)；

静态调用图还提供了堆栈使用信息，它显示出了：

1. 每个函数所使用的堆栈大小；
2. 在全部的函数调用中，所用到的最大堆栈大小。

-info topics

这个选项打印出关于指定种类的信息，这里的参数 topics 是指用逗号间隔的类型标识符列表。类型标识符列表可以是下面所列出的任意一个：

1. sizes

为在映像文件中的每一个输入对象和库成员列出了代码和数据(这里的数据包括，RO 数据，RW 数据，ZI 数据和 Debug 数据)的大小；

2. totals

为输入对象文件和库，列出代码和数据(这里的数据包括，RO 数据，RW 数据，ZI 数据和 Debug 数据) 总的大小；

3. veneers

给出由 armlink 生成的 veneers 的详细信息；

4. unused

列出由于使用-remove 选项而从映像文件中被删除的所有未使用段。

注意：在信息类型标识符列表之间不能存在空格，比如可以输入

-info sizes,totals

但是不能是

-info sizes, totals(即在逗号和 totals 之间有空格是不允许的)

-map

这个选项创建映像文件的信息图。映像文件信息图包括映像文件中的每个加载域，运行域和输入段的大小和地址，这里的输入段还包括调试信息和链接器产生的输入段。

-symbols

这个选项列出了链接的时候使用的每一个局部和全局符号。该符号还包括链接生成的符号。

-symdefs file

这个选项创建一个包含来自输出映像文件的全局符号定义的符号定义文件。

默认的，所有的全局符号都写入到符号定义文件中。如果文件 file 已经存在，链接器将限制生成在已存在的 symdefs 文件中已列出的符号。

如果文件 file 没有指明路径信息，链接器将在输出映像文件的路径搜索文件。如果文件没有找到，就会在该目录下面创建文件。

在链接另一个映像文件的时候，可以将符号定义文件作为链接的输入文件。

-edit file

这个选项指定一个 steering 类型的文件，该文件包含用于修改输出文件中的符号信息表的命令。可以在 steering 文件中指定具有以下功能的命令：

隐藏全局符号。使用该选项可以在目标文件中隐藏指定的全局符号。

重命名全局符号。使用这个选项可以解决符号命名冲突的现象。

-xref

该选项列出了在输入段间的所有交叉引用。

-xreffrom object(section)

这个选项列出了从目标文件中的输入段对其他输入段的交叉引用。如果想知道某个指定的输入段中的引用情况，就可以使用该选项。

-xrefto object(section)

该选项列出了从其他输入段到目标文件输入段的引用。

-errors file

使用该选项会将诊断信息从标准输出流重定向到文件 file 中。

-list file

该选项将-info，-map，-symbols，-xref，-xreffrom 和 –xrefto 这几个选项的输出重新定向到文件 file 中。

如果文件 file 没有指定路径信息，就会在输出路径创建该文件，该路径是输出映像文件所在的路径。

-verbose

这个选项将有关链接操作的细节打印出来，包括所包括的目标文件和要用到的库。

-unmangled

该选项指定链接器在由 xref，-xreffrom，-xrefto，和-symbols 所生成的诊断信息中显示出 unmangled C++符号名。

如果使用了这个选项，链接器将 unmangle C++符号名以源码的形式显示出来。这个选项是默认的。

-mangled

这个选项指定链接器显示由-xref，-xreffrom，-xrefto，和-symbols 所产生的诊断信息中的 mangled C++符号名。如果使用了该选项，链接器就不会 unmangle C++符号名了。符号名是按照它们在目标符号表中显示的格式显示的。

-via file

该选项表示从文件 file 中读取输入文件名列表和链接器选项。

在 armlink 命令行可以输入多个-via 选项，当然，-via 选项也能够不含在一个 via 文件中。

-strict

这个选项告诉链接器报告可能导致错误而不是警告的条件。

-unresolved symbol

这个选项将未被解析的符号指向全局符号 symbol。Symbol 必须是已定义的全局符号，否则，symbol 会当作一个未解析的符号，链接将以失败告终。这个选项在自上而下的开发中尤为有用，在这种情况下，通过将无法指向相应函数的引用指向一个伪函数的方法，可以测试一个部分实现的系统。

该选项不会显示任何警告信息。

input-file-list

这是一个以空格作为间隔符的目标或库的列表。

有一类特殊的目标文件，即 symdef 文件，也可以包含在文件列表中，为生成的映像文件提供全局的 symbol 值。

在输入文件列表中有两种使用库的方法。

1. 指定要从库中提取并作为目标文件添加到映像文件中的特定的成员。
2. 指定某库文件，链接器根据需要从其中提取成员。

armlink 按照以下的顺序处理输入文件列表：

1. 无条件的添加目标文件

2. 使用匹配模式从库中选择成员加载到映像文件中去。例如使用下面的命令：

armlink main.o mylib(stdio.o) mylib(a*.o).

将会无条件的把 mylib 库中所有的以字母 a 开头的目标文件和 stdio.o 在链接的时候链接到生成的映像文件中去。

3. 添加为解析尚未解析的引用的库到库文件列表。

8.1.2 ARM 运行时库

本小节为读者介绍一下 ARM C/C++库方面的相关内容。

8.1.2.1 运行时库类型和建立选项

ADS 提供以下的运行时库来支持被编译的 C 和 C++代码：

ANSI C 库函数：

这个 C 函数库是由以下几部分组成：

1. 在 ISO C 标准中定义的函数；
2. 在 semihosted 环境下(semihosting 是针对 ARM 目标机的一种机制，它能够根据应用程序代码的输入/输出请求，与运行有调试功能的主机通讯。这种技术允许主机为通常没有输入和输出功能的目标硬件提供主机资源)用来实现 C 库函数的与目标相关的函数；
3. 被 C 和 C++编译器所调用的支持函数。

ARM C 库提供了额外的一些部件支持 C++，并为不同的结构体系和处理器编译代码。

C++库函数：

C++库函数包含由 ISO C++库标准定义的函数。C++库依赖于相应的 C 库实现与特定目标相关的部分，在 C++库的内部本身是不包含与目标相关的部分。这个库是由以下几部分组成的：

1. 版本为 2.01.01 的 Rogue Wave Standard C++库；
2. C++编译器使用的支持函数；
3. Rogue Wave 库所不支持的其他的 C++函数。

正如上面所说，ANSI C 库使用标准的 ARM semihosted 环境提供例如，文件输入/输出的功能。Semihosting 是由已定义的软件中断(Software Interrupt)操作来实现的。在大多数的情况下，semihosting SWI 是被库函数内部的代码所触发，用于调试的代理程序处理 SWI 异常。调试代理程序为主机提供所需要的通信。Semihosted 被 ARMulator ，Angel 和 Multi-ICE 所支持。用户可以使用在 ADS 软件中的 ARM 开发工具去开发用户应用程序，然后在 ARMulator 或在一个开发板上运行和调试该程序。

用户可以把 C 库中的与目标相关的函数作为自己应用程序中的一部分，重新进行代码的实现。这就为用户带来了极大的方便，用户可以根据自己的执行环境，适当的裁剪 C 库函数。

除此之外，用户还可以针对自己的应用程序的要求，对与目标无关的库函数进行适当的裁剪。

在 C 库中有很多函数是独立于其他函数的，并且与目标硬件没有任何依赖关系。对于这类函数，用户可以很容易地从汇编代码中使用它们。

在建立自己的用户应用程序的时候，用户必须指定一些最基本的操作选项。例如：

字节顺序，是大端模式(big endian:字数据的高字节存放在低地址，低字节存放在高地址)，还是小端模式(little endian:字数据的高字节存放在高地址，低字节存放在低地址)；

浮点支持：可能是 FPA，VFP，软件浮点处理或不支持浮点运算；

堆栈限制：是否检查堆栈溢出；

位置无关(PID)：数据是从与位置无关的代码还是从与位置相关的代码中读/写，代码是位置无关的只读代码还是位置相关的的只读代码。

当用户对汇编程序，C 程序或 C++程序进行链接的时候，链接器会根据在建立时所指定的选项，选择适当的 C 或 C++运行时库的类型。选项各种不同组合都有一个相应的 ANSI C 库类型。

8.1.2.2 库路径结构

库路径是在 ADS 软件安装路径的 lib 目录下的两个子目录。假设，ADS 软件安装在 e:\arm\adsv1_2 目录，则在 e:\arm\adsv1_2\lib 目录下的两个子目录 armlib 和 cpplib 是 ARM 的库所在的路径。

armlib

这个子目录包含了 ARM C 库，浮点代数运算库，数学库等各类库函数。与这些库相应的头文件在 e:\arm\adsv1_2\include 目录中。

cpplib

这个子目录包含了 Rogue Wave C++库和 C++支持函数库。Rogue Wave C++库和 C++支持函数库合在一起被称为 ARM C++库。与这些库相应的头文件安装在 e:\arm\adsv1_2\include 目录下。

环境变量 ARMLIB 必须被设置成指向库路径。另外一种指定 ARM C 和 ARM C++库路径的方法是，在链接的时候使用操作选项-libpath directory(directory 代表库所在的路径)，来指明要装载的库的路径。

无需对 armlib 和 cpplib 这两个库路径分开指明，链接器会自动从用户所指明的库路径中找出这两个子目录。

这里需要让读者特别注意的以下几点：

1. ARM C 库函数是以二进制格式提供的；
2. ARM 库函数禁止修改。如果读者想对库函数创建新的实现的话，可以把这个新的函数编译成目标文件，然后在链接的时候把它包含进来。这样在链接的时候，使用的是新的函数实现而不是原来的库函数。
3. 通常情况下，为了创建依赖于目标的应用程序，在 ANSI C 库中只有很少的几个函数需要实现重建。
4. Rogue Wave Standard C++函数库的源代码不是免费发布的，可以从 Rogue Wave Software Inc.，或 ARM 公司通过支付许可证费用来获得源文件。

8.1.3 GUI 开发环境(Code Warrior 和 AXD)

8.1.3.1 CodeWarrior 集成开发环境

CodeWarrior for ARM 是一套完整的集成开发工具，充分发挥了 ARM RISC 的优势, 使产品开发人员能够很好的应用尖端的片上系统技术. 该工具是专为基于 ARM RISC 的处理器而设计的, 它可加速并简化嵌入式开发过程中的每一个环节，使得开发人员只需通过一个集成软件开发环境就能研制出 ARM 产品，在整个开发周期中,开发人员无需离开 CodeWarrior 开发环境, 因此节省了在操做工具上花的时间,使得开发人员有更多的精力投入到代码编写上来，

CodeWarrior 集成开发环境(IDE)为管理和开发项目提供了简单多样化的图形用户界面。用户可以使用 ADS 的 CodeWarrior IDE 为 ARM 和 Thumb 处理器开发用 C，C++，或 ARM 汇编语言的程序代码。通过提供下面的功能，CodeWarrior IDE 缩短了用户开发项目代码的周期。

1. 全面的项目管理功能；
2. 子函数的代码导航功能，使得用户迅速找到程序中的子函数。

可以在 CodeWarrior IDE 为 ARM 配置在 8.1.1 中介绍的各种命令工具，实现对工程代码的编译，汇编和链接。

在 CodeWarrior IDE 中所涉及到的 target 有两种不同的语义。

目标系统(Target system)

是特指代码要运行的环境，是基于 ARM 的硬件。比如，要为 ARM 开发板上编写要运行在它上面的程序，这个开发板就是目标系统。

生成目标(Build target)

是指用于生成特定的目标文件的选项设置(包括汇编选项，编译选项，链接选项以及链接后的处

理选项)和所用的文件的集合。

CodeWarrior IDE 能够让用户将源代码文件，库文件还有其他相关的文件以及配置设置等放在一个工程中。每个工程可以创建和管理生成目标设置的多个配置。例如，要编译一个包含调试信息的生成目标和一个基于 ARM7TDMI 的硬件优化生成目标，生成目标可以在同一个工程中共享文件，同时使用各自的设置。

CodeWarrior IDE 为用户提供下面的功能：

源代码编辑器，它集成在 CodeWarrior IDE 的浏览器中，能够根据语法格式，使用不同的颜色显示代码；

源代码浏览器，它保存了在源码中定义的所有符号，能够使用户在源码中快速方便的跳转；

查找和替换功能，用户可以在多个文件中，利用字符串通配符，进行字符串的搜索和替换；

文件比较功能，可以使用户比较路径中的不同文本文件的内容。

ADS 的 CodeWarrior IDE 是基于 Metrowerks CodeWarrior IDE 4.2 版本的。它经过适当的裁剪以支持 ADS 工具链。

针对 ARM 的配置面板为用户提供了在 CodeWarrior IDE 集成环境下配置各种 ARM 开发工具的能力，这样用户可以不用在命令控制台下就能够使用在 8.1.1 和将在 8.1.4 中介绍的各种命令。

以 ARM 为目标平台的工程创建向导，可以使用户以此为基础，快速创建 ARM 和 Thumb 工程。

尽管大多数的 ARM 工具链已经集成在 CodeWarrior IDE，但是仍有许多功能在该集成环境中没有实现，这些功能大多数是和调试相关的，因为 ARM 的调试器没有集成到 CodeWarrior IDE 中。

由于 ARM 调试器(AXD)没有集成在 CodeWarrior IDE 中，这就意味着，用户不能在 CodeWarrior IDE 中进行断点调试和查看变量。

对于熟悉 CodeWarrior IDE 的用户会发现，有许多的功能已经从 CodeWarrior IDE For ARM 中移走，比如快速应用程序开发模板等。

在 CodeWarrior IDE For ARM 中有很多的菜单或子菜单是不能使用的。下面介绍一下这些不能使用的选项。

1. View 菜单下不能使用的菜单选项有：

Processes，Expressions，Global Variable，Breakpoints，Registers。

2. Project 菜单不能使用的菜单选项：

Precompile 子菜单。因为 ARM 编译器不支持预编译的头文件。

3. Debug 菜单

该菜单中没有一个子菜单是可以使用的。

4. Browser 菜单中不能使用的菜单选项：

New Property，New Method 和 New Event Set。

5. Help menu 中不能用于 ADS 的菜单选项有：

CodeWarrior Help，Index，Search 和 Online Manuals。

有关 CodeWarrior IDE 中一些常用菜单的使用，将在后面的举例中具体说明的，在此，不在赘述。

8.1.3.2 ADS 调试器

调试器本身是一个软件，用户通过这个软件使用 debug agent 可以对包含有调试信息的，正在运行的可执行代码进行比如变量的查看，断点的控制等调试操作。

ADS 中包含有 3 个调试器：

AXD(ARM eXtended Debugger)：ARM 扩展调试器；

armsd(ARM Symbolic Debugger)：ARM 符号调试器；

与老版本兼容的 Windows 或 Unix 下的 ARM 调试工具，ADW/ADU(Application Debugger Windows/Unix)。

下面对在调试映像文件中所涉及到的一些术语做一个简单的介绍。

Debug target

在软件开发的最初阶段，可能还没有具体的硬件设备。如果要测试所开发的软件是否达到了预期的效果，这可以由软件仿真来完成。即使调试器和要测试的软件运行在同一台 PC 上，也可以把目标当作一个独立的硬件来看待。

当然，也可以搭建一个 PCB 板，这个板上可以包含一个或多个处理器，在这个板上可以运行和调试应用软件。

只有当通过硬件或者是软件仿真所得到的结果达到了预期的效果，才算是完成了应用程序的编写工作。

调试器能够发送以下指令：

1. 装载映像文件到目标内存；
2. 启动或停止程序的执行；
3. 显示内存，寄存器或变量的值；
4. 允许用户改变存储的变量值。

Debug agent

Debug agent 执行调试器发出的命令动作，比如：设置断点，从存储器中读数据，把数据写到存储器等。

Debug agent 既不是被调试的程序，也不是调试器。在 ARM 体系中，它有这几种方式：Multi-ICE(Multi-processor in-circuit emulator)，ARMulator 和 Angel。其中 Multi-ICE 是一个独立的产品，是 ARM 公司自己的 JTAG 在线仿真器，不是由 ADS 提供的。

AXD 可以在 Windows 和 UNIX 下，进行程序的调试。它为用 C，C++，和汇编语言编写的源代码提供了一个全面的 Windows 和 UNIX 环境。

在后面的章节中，会结合具体实例为读者介绍如何使用 AXD 调试器。

8.1.4 实用程序

ADS 提供以下的实用工具来配合前面介绍的命令行开发工具的使用

fromELF

这是 ARM 映像文件转换工具。该命令将 ELF 格式的文件作为输入文件，将该格式转换为各种输出格式的文件，包括 plain binary(BIN 格式映像文件), Motorola 32-bit S-record format(Motorola 32 位 S 格式映像文件), Intel Hex 32 format(Intel 32 位格式映像文件)，和 Verilog-like hex format(Verilog 16 进制文件)。FromELF 命令也能够为输入映像文件产生文本信息，例如代码和数据长度。

armar

ARM 库函数生成器将一系列 ELF 格式的目标文件以库函数的形式集合在一起，用户可以把一个库传递给一个链接器以代替几个 ELF 文件。

Flash downloader

用于把二进制映像文件下载到 ARM 开发板上的 Flash 存储器的工具

8.1.5 支持的软件

ADS 为用户提供下面的软件，使用户可以在软件仿真的环境下或者在基于 ARM 的硬件环境调试用户应用程序。

ARMulator

这是一个 ARM 指令集仿真器，集成在 ARM 的调试器 AXD 中，它提供对 ARM 处理器的指令集的仿真，为 ARM 和 Thumb 提供精确的模拟。用户可以在硬件尚未做好的情况下，开发程序代码。

8.2 使用 ADS 创建工程

本节通过一个具体实例，为读者介绍如何使用该集成开发环境，利用 CodeWarrior 提供的建立工程的模板建立自己的工程，并学会如何进行编译链接，生成包含调试信息的映像文件和可以直接烧写的 Flash 中的.bin 格式的二进制可执行文件。

8.2.1 建立一个工程

工程将所有的源码文件组织在一起，并能够决定最终生成文件存放的路径，输出的格式等。

在 CodeWarrior 中新建一个工程的方法有两种，可以在工具栏中单击“New”按钮，也可以在“File”菜单中选择“New…”菜单。这样就会打开一个如图 8.1 所示的对话框。

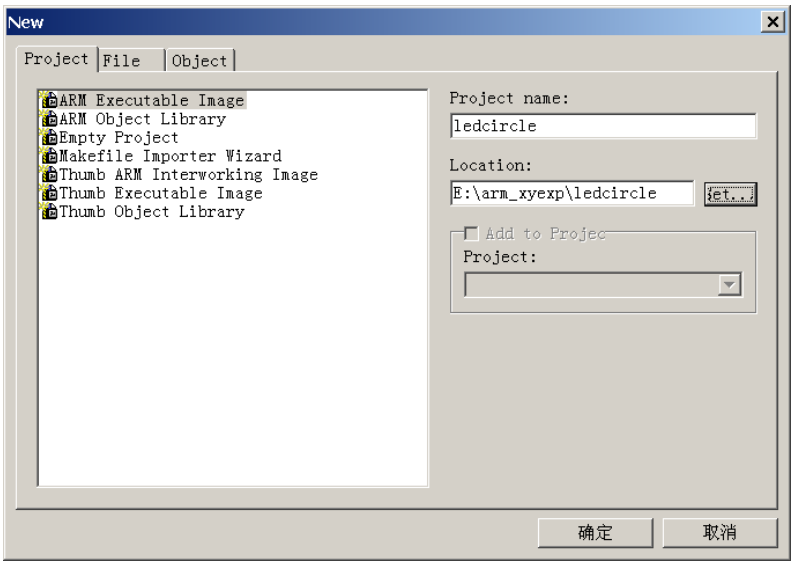


图 8.1 新建工程对话框

在这个对话框中为用户提供了 7 种可选择的工程类型。

ARM Executabl Image：用于由 ARM 指令的代码生成一个 ELF 格式的可执行映像文件；

ARM Object Library：用于由 ARM 指令的代码生成一个 armar 格式的目标文件库；

Empty Project：用于创建一个不包含任何库或源文件的工程；

Makefile Importer Wizard：用于将 Visual C 的 nmake 或 GNU make 文件转入到 CodeWarrior IDE 工程文件；

Thumb ARM Executable Image：用于由 ARM 指令和 Thumb 指令的混和代码生成一个可执行的 ELF 格式的映像文件；

Thumb Executable image：用于由 Thumb 指令创建一个可执行的 ELF 格式的映像文件；

Thumb Object Library：用于由 Thumb 指令的代码生成一个 armar 格式的目标文件库。

在这里选择 ARM Executable Image，在“Project name:”中输入工程文件名，本例为“ledcircle”，点击“Location:”文本框的“Set…”按钮，浏览选择想要将该工程保存的路径，将这些设置好后，点击“确定”，即可建立一个新的名为 ledcircle 的工程。

这个时候会出现 ledcircle.mcp 的窗口，如图 8.2 所示，有三个标签页，分别为 files,link order,target 默认的是显示第一个标签页 files。通过在该标签页点击鼠标右键，选中“Add Files…”可以把要用到的源程序添加到工程中。

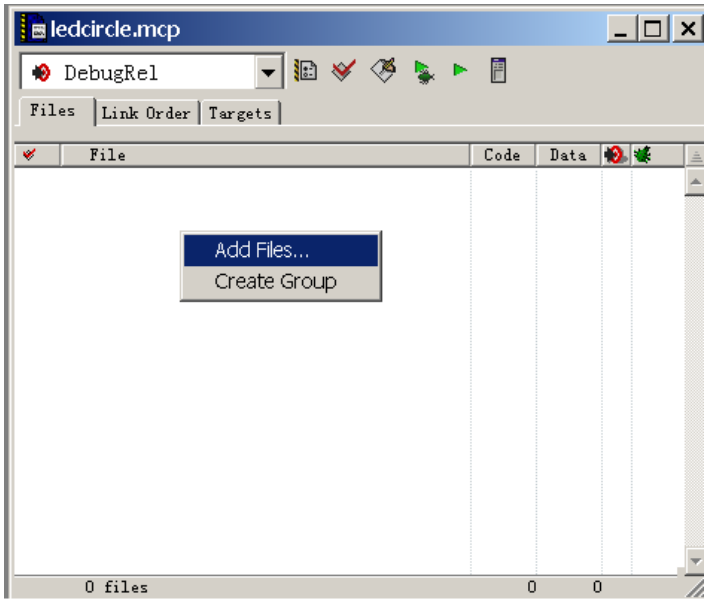


图 8.2　新建工程打开窗口

对于本例，由于所有的源文件都还没有建立，所以首先需要新建源文件。

在“File”菜单中选择“New”，在打开的如图 8.1 所示的对话框中，选择标签页 File，在 File name 中输入要创建的文件名，输入“Init.s”，点击“确定”关闭窗口。

在打开的文件编辑框中输入下面的汇编代码：

```
; ************************************************************
; Chinese Academy of Sciences, Institute of Automation
; File Name:    Init.s
; Description:
; Author:       JuGuang.Li
; Date:
; ************************************************************
        IMPORT      Main
        AREA    Init, CODE, READONLY
        ENTRY
        LDR R0, =0x3FF0000
        LDR R1, =0xE7FFFF80  ;配置 SYSCFG,片内 4K Cache,4K SRAM
        STR     R1, [R0]
        LDR SP, =0x3FE1000   ;SP 指向 4K SRAM 的尾地址，堆栈向下生成
        BL      Main
        B       .
        END
```

在这段代码中，伪操作 IMPORT 告诉编译器符号 Main 不是在该文件中定义的，而是在其他源文件中定义的符号，但是本源文件中可能要用到该符号。接下来用伪指令 AREA 定义段名为 Init 的段为只读的代码段，伪指令 ENTRY 指出了程序的入口点。下面就是用汇编指令实现了配置 SYSCFG 特殊功能寄存器，将 S3C4510B 片内的 8K 一体化的 SRAM 配置为 4K Cache，4K SRAM，并将用户堆栈设置在片内的 SRAM 中。

4K SRAM 的地址为 0x3FE,0000～(0x3FE,1000-1)，由于 S3C4510B 的堆栈由高地址向低地址生成，将 SP 初始化为 0x3FE,1000。

完成上述操作后，程序跳转到 Main 函数执行。

保存 Init.s 汇编程序。

用同样的方法，再建立一个名为 main.c 的 C 源代码文件。具体代码内容如下：

```
//*****************************************************************
//Chinese Academy of Sciences, Institute of Automation
//File Name:    main.c
//Description:
//Author:       JuGuang.Li
//Date:
//*****************************************************************
#define  IOPMOD   (*(volatile unsigned *)0x03FF5000) //IO port mode register
#define  IOPDATA  (*(volatile unsigned *)0x03FF5008) //IO port data register
void Delay(unsigned int);
int Main()
{
      unsigned long  LED;
      IOPMOD=0xFFFFFFFF;    //将 IO 口置为输出模式
      IOPDATA=0x01;
      for(;;){
            LED=IOPDATA;
            LED=(LED<<1);
            IOPDATA=LED;
            Delay(10);
            if(!(IOPDATA&0x0F))
                IOPDATA=0x01;
            }
      return(0);
}
void Delay(unsigned int x)
{
      unsigned int i,j,k;
      for(i=0;i<=x;i++)
            for(j=0;j<0xff;j++)
                for(k=0;k<0xff;k++);
      }
```

该段代码首先将 I/O 模式寄存器设置为输出模式，为 I/O 数据寄存器赋初值为 0x1，通过将 I/O 数据寄存器的数值进行周期性的左移，实现使接在 P0~P3 口的 LED 显示器轮流被点亮的功能。(注意这里的 if 语句，是为了保证当 I/O 数据寄存器中的数在移位过程中，第 4 位为数字“1”时，使数字 1 通过和 0xFF 相与，又重新回到 I/O 数据寄存器的第 0 位，从而保证了数字 1 一直在 I/O 数据寄存器的低四位之间移位。)

在这里还有一个细节，希望读者注意。在建立好一个工程时，默认的 target 是 DebugRel，还有另外两个可用的 target，分别为 Realse 和 Debug，这三个 target 的含义分别为：

DebugRel：使用该目标，在生成目标的时候，会为每一个源文件生成调试信息；

Debug：使用该目标为每一个源文件生成最完全的调试信息；

Release：使用该目标不会生成任何调试信息。

在本例中，使用默认的 DebugRel 目标。

现在已经新建了两个源文件，要把这两个源文件添加到工程中去。

为工程添加源码常用的方法有两种，既可以使用入图 8.2 所示方法，也可以在“Project”菜单项中，选择“Add Files…”，这两种方法都会打开文件浏览框，用户可以把已经存在的文件添加到工程中来。当选中要添加的文件时，会出现一个对话框，如图 8.3 所示，询问用户把文件添加到何类目标中，在这里，我们选择 DebugRel 目标。把刚才创建的两个文件添加到工程中来。

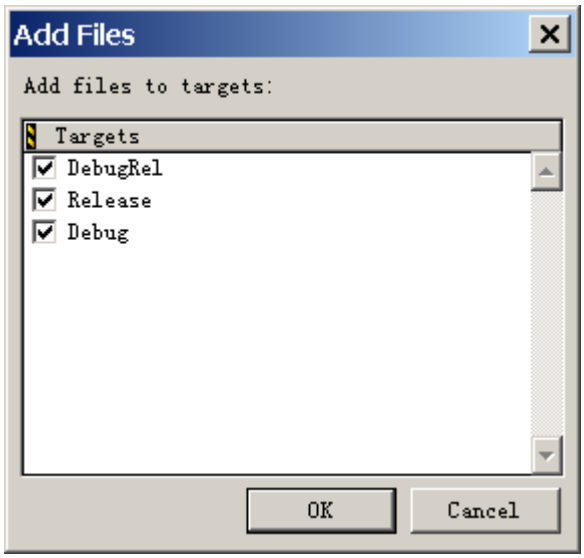


图 8.3　选择添加文件到指定目标

到目前为止，一个完整的工程已经建立。

下面该对工程进行编译和链接工作。

8.2.2　编译和链接工程

在进行编译和链接前，首先讲述一下如何进行生成目标的配置。

点击 Edit 菜单，选择“DebugRel Settings…”(注意，这个选项会因用户选择的不同目标而有所不同)，出现如图 8.2 所示的对话框。

这个对话框中的设置很多，在这里指介绍一些最为常用的设置选项，读者若对其他未涉及到的选项感兴趣，可以查看相应的帮助文件。

1. target 设置选项

Target Name 文本框显示了当前的目标设置。

Linker 选项供用户选择要使用的链接器。在这里默认选择的是 ARM Linker，使用该链接器，将使用 armlink 链接编译器和汇编器生成的工程中的文件相应的目标文件。

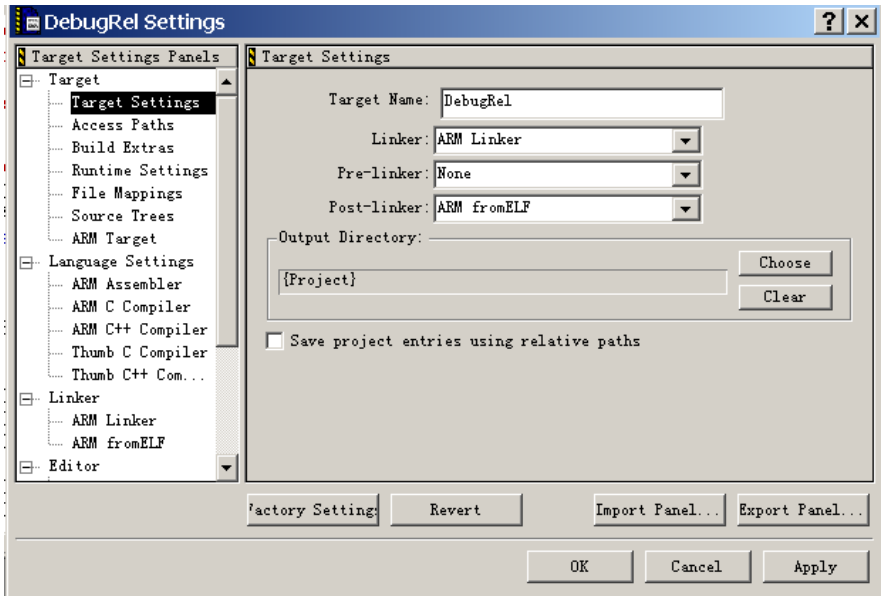


图 8.4　DebugRel 设置对话框

这个设置中还有两个可选项，None 不是不用任何链接器，如果使用它，则工程中的所有文件都不会被编译器或汇编器处理。ARM Librarian 表示将编译或汇编得到的目标文件转换为 ARM 库文件。对于本例，使用默认的链接器 ARM Linker。

Pre-linker：目前 CodeWarrior IDE 不支持该选项。

Post-Linker：选择在链接完成后，还要对输出文件进行的操作。因为在本例中，希望生成一个可以烧写到 Flash 中去的二进制代码，所以在这里选择 ARM fromELF，表示在链接生成映像文件后，再调用 FromELF 命令将含有调试信息的 ELF 格式的映像文件转换成其他格式的文件。

2. Language Settings

因为本例中包含有汇编源代码，所以要用到汇编器。首先看 ARM 汇编器。这个汇编器实际就说在 8.1 节中谈到的 armasm，默认的 ARM 体系结构是 ARM7TDMI，正好符合目标板 S3C4510B，无需改动。字节顺序默认就是小端模式。其他设置，就用默认值即可。

还有一个需要注意的就是 ARM C 编译器，它实际就是调用的命令行工具 armcc。使用默认的设置就可以了。

细心的读者可能会注意到，在设置框的右下脚，当对某项设置进行了修改，该行中的某个选项就会发生相应的改动，如图 8.5 所示。实际上，这行文字就显示的是在 8.1 中介绍的相应的编译或链接选项，由于有了 CodeWarrior，开发人员可以不用再去查看繁多的命令行选项，只要在界面中选中或撤消某个选项，软件就会自动生成相应的代码，为不习惯在 DOS 下键入命令行的用户提供了极大的方便。

3. Linker 设置

鼠标选中 ARM Linker，出现如图 8.6 所示对话框。这里详细介绍该对话框的主要的标签页选项，因为这些选项对最终生成的文件有着直接的影响。

在标签页 Output 中，Linktype 中提供了三种链接方式。Partial 方式表示链接器只进行部分链接，经过部分链接生成的目标文件，可以作为以后进一步链接时的输入文件。Simple 方式是默认的链接方式，也是最为频繁使用的链接方式，它链接生成简单的 ELF 格式的目标文件，使用的是链接器选项中指定的地址映射方式。Scattered 方式使得链接器要根据 scatter 格式文件中指定的地址映射，生成复杂的 ELF 格式的映像文件。这个选项一般情况

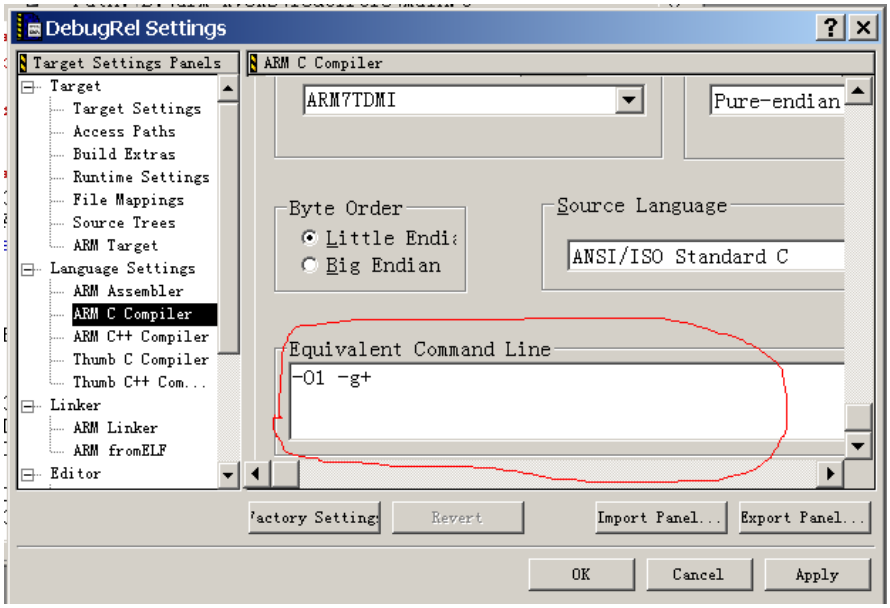


图 8.5 命令行工具选项设置

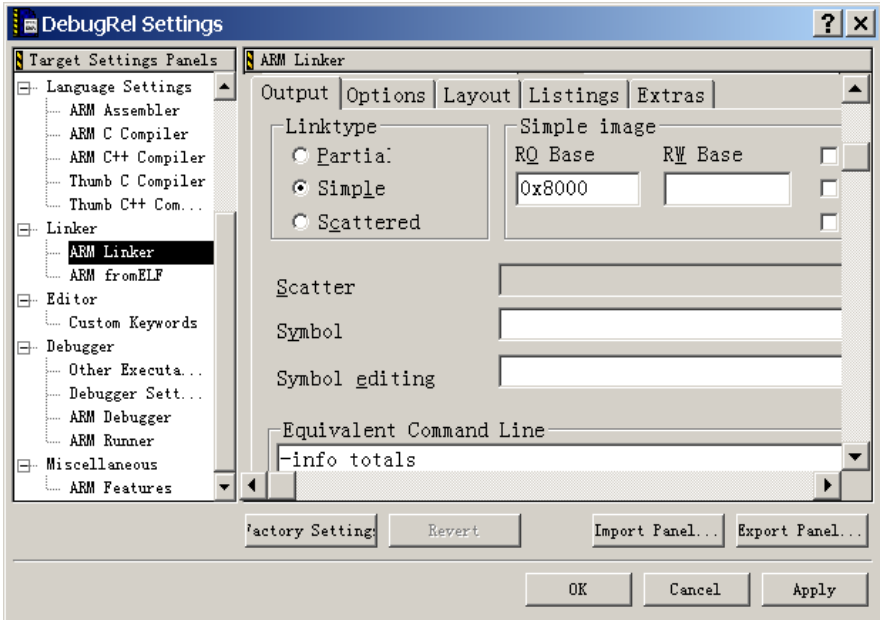


图 8.6　链接器设置

下，使用不太多。

因为我们所举的例子比较简单，选择 Simple 方式就可以了。

在选中 Simple 方式后，就会出现 Simple image。

RO Base：这个文本框设置包含有 RO 段的加载域和运行域为同一个地址。默认是 0x8000。这里用户要根据自己硬件的实际 SDRAM 的地址空间来修改这个地址，保证在这里填写的地址，是程序运行时，SDRAM 地址空间所能覆盖的地址。针对本书所介绍的目标板，就可以使用这个默认地址值。

RW Base：这个文本框设置了包含 RW 和 ZI 输出段的运行域地址。如果选中 split 选项，链接器生成的映像文件将包含两个加载域和两个运行域，此时，在 RW Base 中所输入的地址为包含 RW 和 ZI 输出段的域设置了加载域和运行域地址

Ropi：选中这个设置将告诉链接器使包含有 RO 输出段的运行域位置无关。使用这个选项，链接器将保证下面的操作：

检查各段之间的重定址是否有效；

确保任何由 armlink 自身生成的代码是只读位置无关的。

Rwpi：选中该选项将会告诉链接器使包含 RW 和 ZI 输出段的运行域位置无关。如果这个选项没有被选中，域就标识为绝对。每一个可写的输入段必须是读写位置无关的。如果这个选项被选中，链接器将进行下面的操作，

检查可读/可写属性的运行域的输入段是否设置了位置无关属性；

检查在各段之间的重地址是否有效；

在 Region$$Table 和 ZISection$$Table 中添加基于静态存储器 sb 的选项。

该选项要求 RW Base 有值，如果没有给它指定数值的话，默认为 0 值。

Split Image：选择这个选项把包含 RO 和 RW 的输出段的加载域分成 2 个加载域：一个是包含 RO 输出段的域，一个是包含 RW 输出段的域。

这个选项要求 RW Base 有值，如果没有给 RW Base 选项设置，则默认是-RW Base 0。

Relocatable：选择这个选项保留了映像文件的重定址偏移量。这些偏移量为程序加载器提供了有用信息。

在 Options 选项中，需要读者引起注意的是 Image entry point 文本框。它指定映像文件的初始入口点地址值，当映像文件被加载程序加载时，加载程序会跳转到该地址处执行。如果需要，用户可以在这个文本框中输入下面格式的入口点：

入口点地址：这是一个数值，例如-entry 0x0

符号：该选项指定映像文件的入口点为该符号所代表的地址处，比如：

-entry int_handler

如果该符号有多处定义存在，armlink 将产生出错信息。

offset+object(section)：该选项指定在某个目标文件的段的内部的某个偏移量处为映像文件的入口地址，例如：

-entry 8+startup(startupseg)

在此处指定的入口点用于设置 ELF 映像文件的入口地址。

需要引起注意的是，这里不可以用符号 main 作为入口点地址符号，否则将会出现类似“Image dose not have an entry point(Not specified or not set due to multiple choice)”的错误信息。

关于 ARM Linker 的设置还有很多，对于想进一步深入了解的读者，可以查看帮助文件，都有很详细的介绍。

在 Linker 下还有一个 ARM fromELF，如图 8.7 所示：

fromELF 就是在 8.1 节中介绍的一个实用工具，它实现将链接器，编译器或汇编器的输出代码进行格式转换的功能。例如，将 ELF 格式的可执行映像文件转换成可以烧写到 ROM 的二进制格式文件；对输出文件进行反汇编，从而提取出有关目标文件的大小，符号和字符串表以及重定址等信息。

只有在 Target 设置中选择了 Post-linker，才可以使用该选项。

在 Output format 下拉框中，为用户提供了多种可以转换的目标格式，本例选择 Plain binary，这是一个二进制格式的可执行文件，可以被烧些的目标板的 Flash 中。

在 Output file name 文本域输入期望生成的输出文件存放的路径，或通过点击 Choose...按钮从文件对话框中选择输出文件。如果在这个文本域不输入路径名，则生成的二进制文件存放在工程所在的目录下。

进行好这些相关的设置后，以后在对工程进行 make 的时候，CodeWarrior IDE 就会在链接完成后调用 fromELF 来处理生成的映像文件。

对于本例的工程而言，到此，就完成了 make 之前的设置工作了。

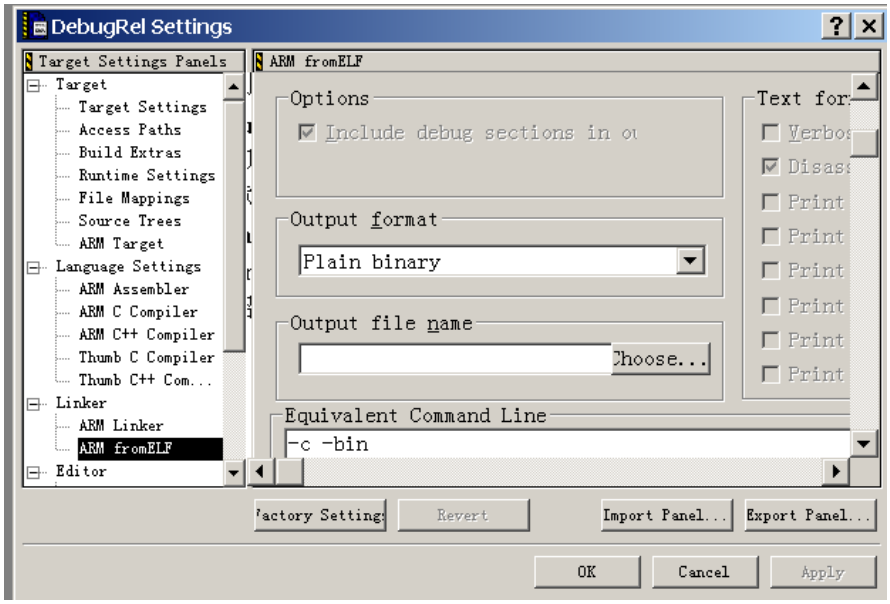


图 8.7 ARM fromELF 可选项

点击 CodeWarrior IDE 的菜单 Project 下的 make 菜单，就可以对工程进行编译和链接了。整个编译链接过程如图 8.8 所示：

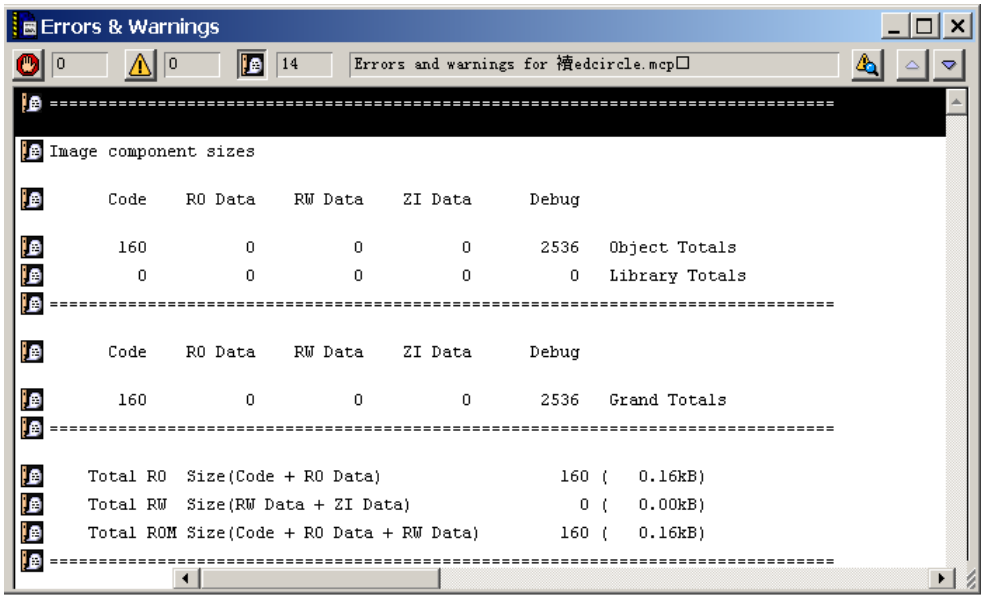


图 8.8　编译和链接过程

在工程 ledcircle 所在的目录下，会生成一个名为：工程名_data 目录，在本例中就是 ledcircle_data 目录，在这个目录下不同类别的目标对应不同的目录。在本例中由于我们使用的是 DebugRe 目标，所以生成的最终文件都应该在该目录下。进入到 DebugRel 目录中去，读者会看到 make 后生成的映像文件和二进制文件，映像文件用于调试，二进制文件可以烧写到 S3C4510B 的 Flash 中运行。

8.2.3　使用命令行工具编译应用程序

如果用户开发的工程比较简单，或者只是想用到 ADS 提供的各种工具，但是并不想在 CodeWarrior IDE 中进行开发。在这种情况下，再为读者介绍一种不在 CodeWarrior IDE 集成开发环境下，开发用户应用程序的方法，当然前提是用户必须安装了 ADS 软件，因为在编译链接的过程中要用到 ADS 提供的各种命令工具。

这种方法对于开发包含较少源代码的工程是比较实用的。

首先用户可以用任何编辑软件(比如 UltraEdit)编写 8.2.1 中所提到的两个源文件 Init.s 和 main.c。接下了，可以利用在第 7 章中介绍的 makefile 的知识，编写自己的 makefile 文件。对于本例，编写的 makefile 文件(假设该 makefile 文件保存为 ads_mk.mk)如下：

```
PAT = e:/arm/adsv1_2/bin
CC  = $(PAT)/armcc
LD      = $(PAT)/armlink
OBJTOOL = $(PAT)/fromelf

RM  = $(PAT)/rm -f
AS  = $(PAT)/armasm -keep -g
ASFILE  = e:/arm_xyexp/Init.s
CFLAGS  = -g -O1  -Wa -DNO_UNDERSCORES=1
MODEL   = main
SRC     = $(MODEL).c
OBJS    = $(MODEL).o
all:    $(MODEL).axf  $(MODEL).bin  clean

%.axf:$(OBJS) Init.o
```

```
        @echo "### Linking ..."
        $(LD)  $(OBJS) Init.o -ro-base 0x8000  -entry Main -first Init.o  -o $@ -libpath
e:/arm/adsv1_2/lib
    %.bin: %.axf
        $(OBJTOOL) -c -bin -output $@ $<
        $(OBJTOOL) -c -s -o $(<:.axf=.lst) $<

    %.o:%.c
        @echo "### Compiling $<"
        $(CC) $(CFLAGS)  -c $< -o $@

    clean:
        $(RM) Init.o $(OBJS)
```

由于 ADS 在安装的时候没有提供 make 命令，可以将在第 7 章中用到的 make 命令直接拷贝到 ADS 安装路径的 bin 目录下，比如 ADS 安装在目录 e:\arm\adsv1_2 下， 可以将 make 命令拷贝到 e:\arm\adsv1_2\bin 目录下，在 command console 下的编译过程如图 8.9 所示：

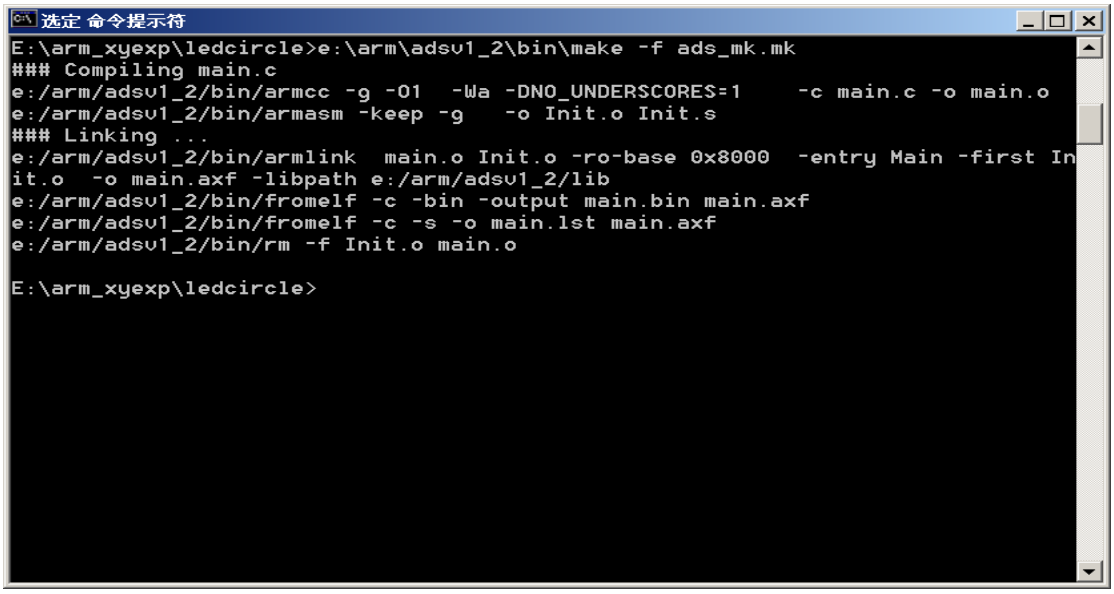


图 8.9　在 command console 下编译过程

经过上述编译链接，以及链接后的操作，在 e:\arm_xyexp\ledcircle 目录下会生成两个新的文件，main.axf 和 main.bin。

用这种方式生成的文件与在 CodeWarrior IDE 界面通过各个选项的设置，生成的文件是一样的。

在所举的例子中，都生成了包含有调试信息的可执行映像文件，即以.axf 结尾的文件。下面一节将介绍用如何用 AXD 对程序进行源码级的调试。

8.3　用 AXD 进行代码调试

AXD(ARM eXtended Debugger)是 ADS 软件中独立于 CodeWarrior IDE 的图形软件，打开 AXD 软件，默认是打开的目标是 ARMulator。这个也是调试的时候最常用的一种调试工具，本节主要是结合 ARMulator 介绍在 AXD 中进行代码调试的方法和过程，使读者对 AXD 的调试有初步的了解。

要使用 AXD 必须首先要生成包含有调试信息的程序，在 8.2 节中，已经生成的 ledcircle.axf 或 main.axf 就是含有调试信息的可执行 ELF 格式的映像文件。

1. 在 AXD 中打开调试文件

在菜单 File 中选择“Load image…”选项，打开 Load Image 对话框，找到要装载的.axf 映像文件，点击“打开”按钮，就把映像文件装载到目标内存中了。

在所打开的映像文件中会有一个蓝色的箭头指示当前执行的位置。对于本例，打开映像文件后，如图 8.10 所示：

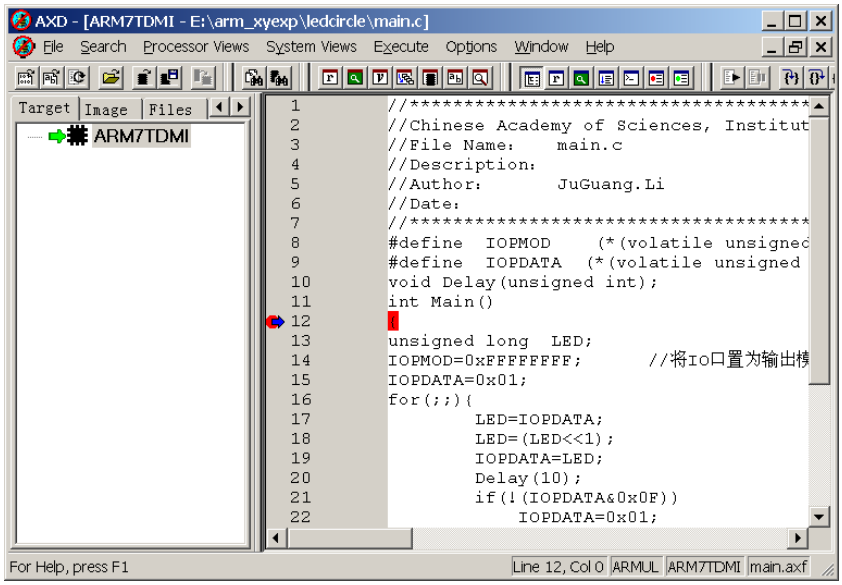


图 8.10 在 axd 下打开映像文件

在菜单 Execute 中选择“Go”，将全速运行代码。要想进行单步的代码调试，在 Execute 菜单中选择“Step”选项，或用 F10 即可以单步执行代码，窗口中蓝色箭头会发生相应的移动。

有时候，用户可能希望程序在执行到某处时，查看一些所关心的变量值，此时可以通过断点设置达到此要求。将光标移动到要进行断点设置的代码处，在 Execute 菜单中，选择“Toggle Breakpoint”或按 F9，就会在光标所在位置出现一个实心圆点，表明该处为断点。

还可以在 AXD 中查看寄存器值，变量值，某个内存单元的数值等等。

下面就结合本章中的例子，介绍在 AXD 中调试过程。

2. 查看存储器内容

在程序运行前，可以先查看两个宏变量 IOPMOD 和 IOPDATA 的当前值。方法是：从 Processor Views 菜单中选择“Memory”选项，如图 8.11 所示。

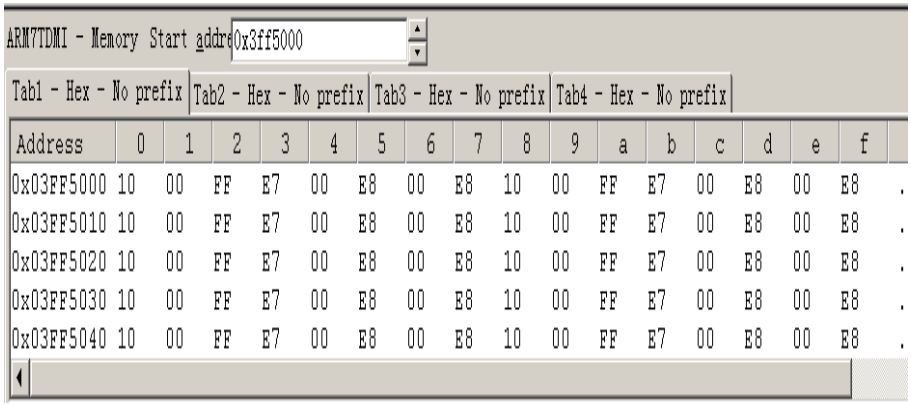


图 8.11 查看存储器内容

在 Memory Start address 选择框中，用户可以根据要查看的存储器的地址输入起始地址，在下面的表格中会列出连续的 64 个地址。因为 I/O 模式控制寄存器和 I/O 数据控制寄存器都是 32 位的控制寄存器，所以从 0x3ff5000 开始的连续四个地址空间存放的是 I/O 模式控制寄存器的值，从图 8.11 可以读出该控制寄存器的值开始为 0xE7FF0010，I/O 数据控制寄存器的内容是从地址 0x3FF5008 开

始的连续四个地址空间存放的内容。从图 8.11 中可以看出 IODATA 中的初始值为 0x E7FF0010，注意因为用的是小端模式，所以读数据的时候注意高地址中存放的是高字节，低地址存放的是低字节。

现在对程序进行单步调试，当程序运行到 for 循环处时，可以再一次查看这两个寄存器中的内容，此时存储器的内容如图 8.12 所示：

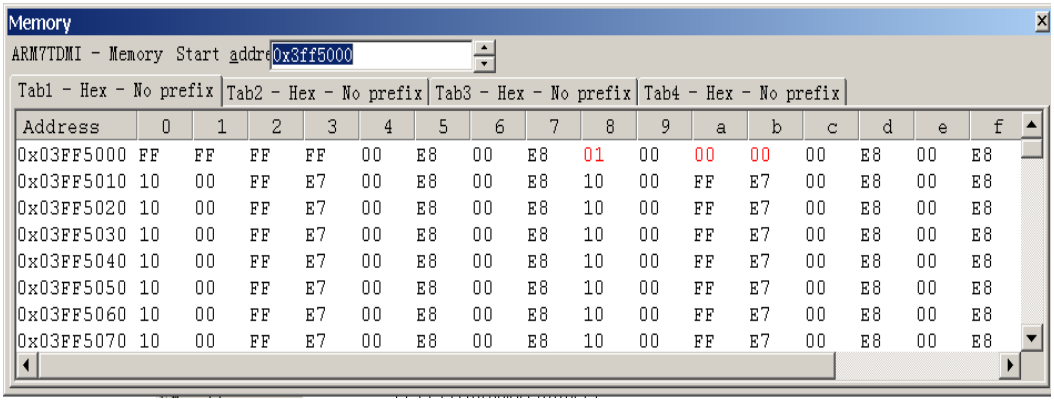


图 8.12　单步运行后的存储器内容

从图中可以看出运行完两个赋值语句后，两个寄存器的内容的确发生了变化，在地址 0x3FF5000 作为起始地址的连续四个存储单元中，可以读出 I/O 模式控制寄存器的内容为 0xFFFFFFFF，在地址 0x3FF5008 开始的连续的四个存储单元中，可以读出 I/O 数据控制寄存器的内容为 0x00000001。

3. 设置断点

可以在 for 循环体的“Delay(10);”语句处设置断点，将光标定位在该语句处，使用快捷键 F9 在此处设置断点，按 F5 键，程序将运行到断点处，如果读者想查看子函数 Delay 是如何运行的，可以在 Execute 菜单中选择“Step In”选项，或按下 F8 键，进入到子函数内部进行单步程序的调试。如图 8.13 所示。

4. 查看变量值

在 Delay 函数的内部，如果用户希望查看某个变量的值，比如查看变量 i 的值，可以在 Processor Views 菜单中选择“Watch”,会出现如图 8.14 所示的 watch 窗口，然后用鼠标选中变量 i，点击鼠标邮件，在快捷菜单中选中“Add to watch”，如图 8.14 所示，这样变量 i 默认是添加到 watch 窗口的 Tab1 中。程序运行过程中，用户可以看到变量 i 的值在不断的

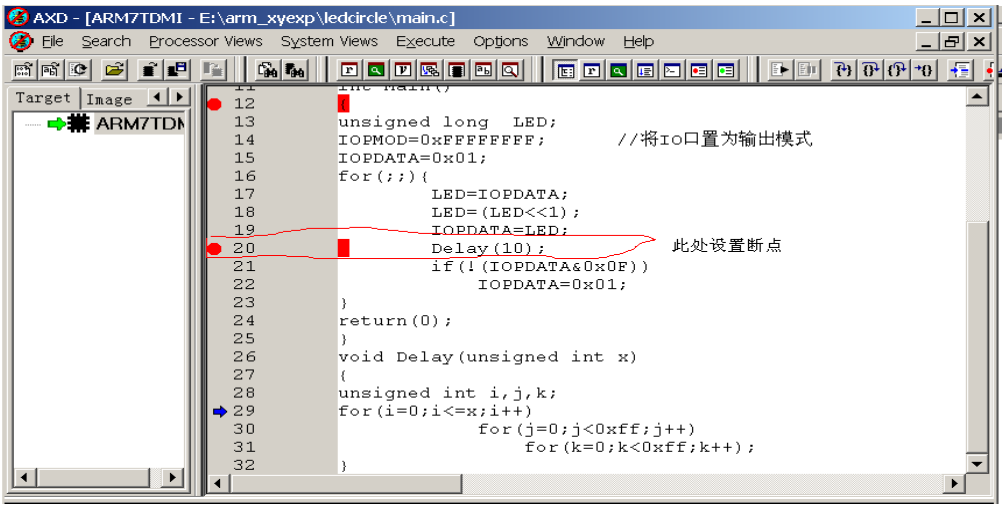


图 8.13　设置断点

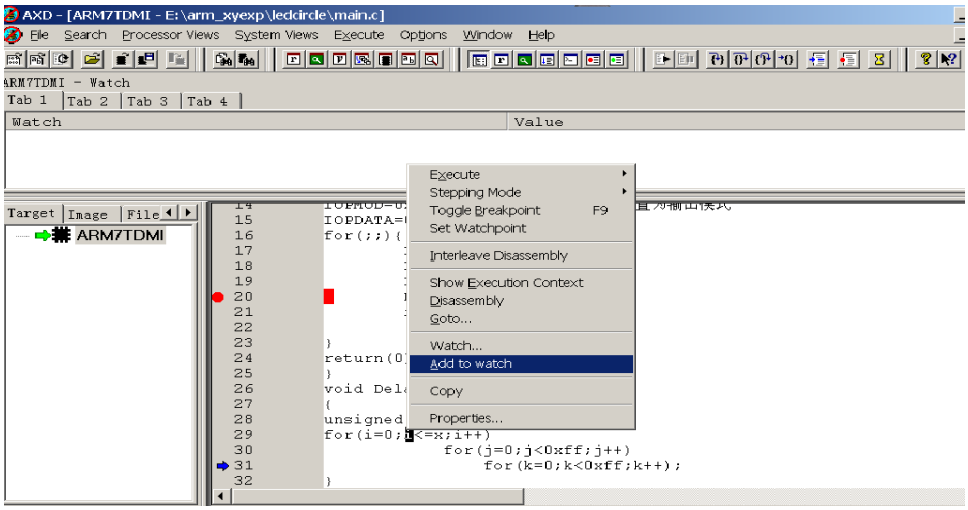


图 8.14　查看变量

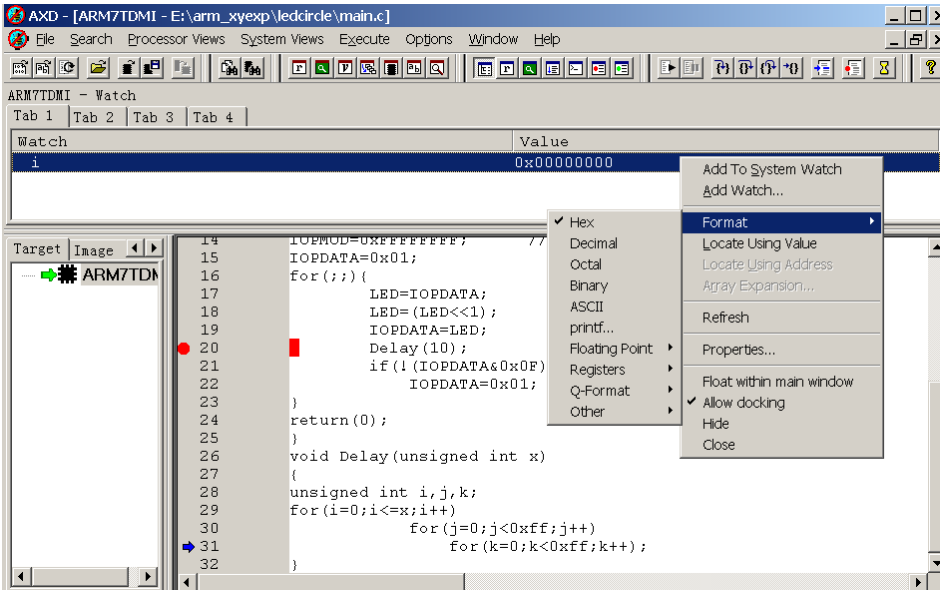


图 8.15　改变变量的格式

变化。默认显示变量数值是以十六进制格式显示的，如果用户对这种显示格式不习惯的话，可以通过在 watch 窗口点击鼠标右键，在弹出的快捷菜单中选择“Format”选项，如图 8.15 所示，用户可以选择所查看的变量显示数据的格式。如果用户想从 Delay 函数中跳出到主函数中去，最简单的方法就是将光标定位到你想跳转到的主函数处，在 Execute 菜单中选择“Run to Cursor”选项，则程序会从 Delay 函数中跳转到光标所在位置。

8.4　本章小结

本章主要介绍了 ADS 软件。首先介绍了 ADS 软件的基本组成部分，接着重点地介绍了最常用的 2 个命令工具 armcc 和 armlink 的使用语法和各个操作选项。然后结合一个具体的应用实例，介绍如何在 CodeWarrior IDE 环境下建立自己的新工程，编译和链接工程生成可以调试的映像文件和二进制文件的过程，同时补充了一种不在 CodeWarrior IDE 集成开发环境下，利用第 7 章介绍的有关 make 的知识，利用 ADS 提供的编译和链接等命令工具编写自己的 makefile 文件，来开发和编译程序的方法。最后介绍了如何使用 ADS 的调试软件 AXD 调试应用程序。

BY 三星光棱 04-1-20

Email: bwbtmh@sohu.com