

EduKitIII S3C2410 MDK 实验教程

基于 Embest EduKitIII
S3C2410 嵌入式开发与应用实验平台



深圳市英蓓特信息技术有限公司

Embest Info & Tech Co., Ltd.

地址: 深圳市罗湖区太宁路 85 号罗湖科技大厦 509 室 (518020)

Telephone: 86-755-25635626 25638952 25638953 25631365

Fax: 86-755-25616057

E-mail: sales@embedinfo.com support@embedinfo.com

Website: <http://www.embedinfo.com> <http://www.embed.com.cn>

目 录

第一章 嵌入式系统开发与应用概述	3
1.1 嵌入式系统开发与应用	3
1.2 基于ARM的嵌入式开发环境概述	5
1.3 各种ARM开发工具简介	7
1.4 如何学习基于ARM嵌入式系统开发	15
第二章 EMBEST ARM实验教学系统	16
2.1 教学系统介绍	16
2.2 教学系统安装	22
2.3 集成开发环境使用说明	27
第三章 嵌入式软件开发基础实验	54
3.1 ARM汇编指令实验一	54
3.2 ARM汇编指令实验二	63
3.3 Thumb 汇编指令实验	69
3.4 ARM处理器工作模式实验	74
3.5 C语言实例一	79
3.6 C语言实验程序二	82
3.7 汇编与C语言相互调用实例	88
3.8 综合实验	92
第四章 基本接口实验	102
4.1 存储器实验	102
4.2 IO口实验	115
4.3 中断实验	125
4.4 串口通信实验	136
4.5 实时时钟实验	145
4.6 数码管显示实验	152
4.7 看门狗实验	158
第五章 人机接口实验	165
5.1 液晶显示实验	165
5.2 5x4 键盘控制实验	183
5.3 触摸屏控制实验	189
第六章 通信与接口实验	202
6.1 IIC串行通信实验	202
6.2 以太网通讯实验	214
6.3 音频接口IIS实验	230
6.4 USB接口实验	237
6.5 SPI接口通讯实验	251
6.6 红外模块控制实验	259
第七章 基础应用实验	269
7.1 A/D转换实验	269
7.2 PWM步进电机控制实验	275
第八章 高级应用实验	288
8.1 GPRS模块控制实验	288

第一章 嵌入式系统开发与应用概述

1.1 嵌入式系统开发与应用

以嵌入式计算机为技术核心的嵌入式系统是继网络技术之后，又一个 IT 领域新的技术发展方向。由于嵌入式系统具有体积小、性能强、功耗低、可靠性高以及面向行业具体应用等突出特征，目前已经广泛地应用于军事国防、消费电子、信息家电、网络通信、工业控制等各个领域。嵌入式的广泛应用可以说是无所不在。就我们周围的日常生活用品而言，各种电子手表、电话、手机、PDA、洗衣机、电视机、电饭锅、微波炉、空调器都有嵌入式系统的存在，如果说我们生活在一个充满嵌入式的世界，是毫不夸张的。据统计，一般家用汽车的嵌入式计算机在 24 个以上，豪华汽车的在 60 个以上。难怪美国汽车大亨福特公司的高级经理也曾宣称，“福特出售的‘计算能力’已超过了 IBM”，由此可见嵌入式计算机工业的应用规模、应用深度和应用广度。

嵌入式系统组成的核心部件是各种类型的嵌入式处理器/DSP。随着嵌入式系统不断深入到人们生活中的各个领域，嵌入式处理器也进而得到前所未有的飞速发展。目前据不完全统计，全世界嵌入式处理器/DSP 的品种总量已经超过 1500 多种，流行体系结构也有近百个系列，现在几乎每个半导体制造商都生产嵌入式处理器/DSP，越来越多的公司有自己的处理器/DSP 设计部门。

嵌入式微处理器技术的基础是通用计算机技术。现在许多嵌入式处理器也是从早期的 PC 机的应用发展演化过来的，如早期 PC 诸如 TRS-80、Apple II 和所用的 Z80 和 6502 处理器，至今仍为低端的嵌入式应用。在应用中，嵌入式微处理器具有体积小、重量轻、成本低、可靠性高的优点。嵌入式处理器目前主要有 Am186/88、386EX、SC-400、Power PC、68000、MIPS、ARM 等系列。

在早期实际的嵌入式应用中，芯片选择时往往以某一种微处理器内核为核心，在芯片内部集成必要的 ROM/EPROM/Flash/EEPROM、SRAM、接口总线及总线控制逻辑、定时/计数器、WatchDog、I/O、串行口、脉宽调制输出、A/D、D/A 等各种必要的功能和外设。为了适应不同的应用需求，一般一个系列具有多种衍生产品，每种衍生产品的处理器内核几乎都是一样的，不同之处在于存储器的种类、容量和外设接口模块的配置及芯片封装。这样可以最大限度地和实际的应用需求相匹配，满足实际产品的开发需求，使得芯片功能不多不少，从而降低功耗、减少成本。随着嵌入式系统应用普及的日益广泛，嵌入式系统的复杂度提高，控制算法也更加冗繁，尤其是嵌入式 Internet 的广泛应用、嵌入式操作系统的引入、触摸屏等复杂人机接口的广泛使用以及芯片设计及加工工艺的提高，以 32 位处理器为核的 SOC 系统芯片的大范围使用，极大的推动了嵌入式 IT 技术的发展速度。计算机应用的普及、互联网技术的使用以及纳米微电子技术的突破，也正有力地推动着未来的工业生产、商务活动、科学实验和家庭生活等领域的自动化和信息化进程。全生产过程自动化产品制造、大范围电子商务活动、高度协同科学实验以及现代化家庭起居，也为嵌入式产品造就了崭新而巨大的商机。

除了信息高速公路的交换机、路由器和 Modem，构建 CIMS 所需的 DCS 和机器人以及规模较大的家用汽车电子系统。最有量产效益和时代特征的嵌入式产品应数因特网上的信息家电，如 Web 可视电话、Web 游戏机、Web PDA(俗称电子商务、商务通)、WAP 电话手机、以及多媒体产品，如 STB(电视机顶盒)、DVD 播放机、电子阅读机。

嵌入式应用领域近几年发展起来的一项概念和技术就是嵌入式 Internet 实际应用，也代表着它是指设备通过嵌入式模块而非 PC 系统直接接入 Internet，以 Internet 为介质实现信息交互的过程，通常又称为非 PC 的 Internet 接入。

嵌入式 Internet 将世界各地不同地方、不同种类及不同应用的设备联系起来，实现信息资源的实时共享。嵌入式 Internet 技术的广泛应用满足了二十一世纪人们要求随时随地获取信息、处理信息的需求。对 IT 产业发展规律进行总结，如果说前 20 年 PC 机的广泛应用是集成电路、IT 相关技术发展的驱动器，并且极大的促进 IT 相关技术发展的话，那么后 20 年除了 PC 技术要继续高速发展之外，主要驱动器应该是与 Internet 结合的可移动的（Mobile）、便携的（Portable）、实时嵌入式 Internet 信息处理设备。目前嵌入式 Internet 还是局限于智能家居（家电上网）、工业控制和智能设备的应用等。随着相关应用技术的发展，嵌入式技术必将会和许多领域的实际应用相结合，以难以想象的应用范围和速度发展，这必然会极大拓展嵌入式应用的广度和深度，体现嵌入式更加广泛的与实际应用密切结合的实用价值。

嵌入式 Internet 应用的核心是高性能、低功耗的各种基于网络信息处理的嵌入式系统芯片 SOC 及相关应用技术，它们是以 Internet 为介质，通过嵌入式网络处理器实现信息交互过程的非 PC Internet 设备接入。

随着 Internet 技术的成熟、带宽的运行速度的提高，ICP 和 ASP 在网上提供的信息内容日趋丰富、应用项目也多种多样。像手机、电话座机及电冰箱、微波炉等嵌入式电子设备的功能已不再单一，电气结构也更为复杂。为了满足应用功能的升级，系统设计技术人员一方面采用更强大的嵌入式处理器如 32 位、64 位 RISC 芯片或信号处理器 DSP 增强处理能力；同时还采用实时多任务编程技术和交叉开发工具技术来控制功能的复杂性，简化应用程序设计过程、保障软件质量和缩短产品开发周期。

目前，市面上已有几千种嵌入式芯片可供选择。由于面向应用的需要，许多产品设计人员还是根据自己产品特点设计自己的嵌入式芯片。通常设计人员首先获得嵌入式微处理器核的授权，然后增加他们应用产品所需的专门特点的接口模块。例如，针对数码像机处理器有可能加一个电荷耦合芯片；对网络应用产品处理则可能加一个以太网接口，而嵌入式微处理器核应用会越来越多，选用不同的核，会使电路的性能差别很大。

ARM 系列处理器核是英国先进 RISC 机器公司（Advanced RISC Machines, ARM）的产品。ARM 公司自成立以来，一直以 IP(Intelligence Property)提供者的身份向各大半导体制造商出售知识产权，而自己从不介入芯片的生产销售，它提供一些高性能、低功耗、低成本和高可靠性的 RISC 处理器核、外围部件和系统级芯片的应用解决设计方案。

ARM 处理器核具有低功耗、低成本等卓越性能和显著优点，越来越多的芯片厂商早已看好 ARM 的前景。ARM 处理器核得到了众多的半导体厂家和整机厂商的大力支持，在 32 位嵌入式应用领域获得了巨大的成功,如 Intel、Motorola、IBM、NS、Atmel、Philips、NEC、OKI、SONY 等世界上几乎所有的半导体公司获得 ARM 授权，开发具有自己特色的基于 ARM 的嵌入式系统芯片。

目前非常流行的 ARM 芯核有 ARM7TDMI, ARM720T, ARM9TDMI, ARM920T, ARM940T, ARM946T, ARM966T, XScale 等。ARM 公司 2003 在美国加利福尼亚州圣荷西市召开的嵌入式处理器论坛上公布了四个新的 ARM11 系列微处理器内核（ARM1156T2-S 内核、ARM1156T2F-S 内核、ARM1176JZ-S 内核和 ARM11JZF-S 内核）、应用 ARM1176JZ-S 和 ARM11JZF-S 内核系列的 PrimeXsys

平台、相关的 CoreSight 技术。ARM 公司日前发布最新的 Cortex 处理器，分为 R、M、A 三个系列，其中 A 系列与应用（Application）有关；如应用于高清电视、手机通用芯片等；R 系列与实时嵌入系统（Realtime）有关；M 系列与微控制器（MCU）有关。它将给消费和低功耗移动产品带来重大变革，使得最终用户可以享受到更高水准的娱乐和创新。此外，ARM 芯片还获得了许多实时操作系统(Real Time Operating System)供应商的支持，比较知名的有：Windows CE、uCLinux、pSOS、VxWorks、Nucleus、EPOC、uC/OS、BeOS、Palm OS、QNX 等。

ARM 公司具有完整的产业链，ARM 的全球合作伙伴主要为半导体和系统伙伴、操作系统伙伴、开发工具伙伴、应用伙伴、ARM 技术共享计划（ATAP），ARM 的紧密合作伙伴已发展为 122 家半导体和系统合作伙伴、50 家操作系统合作伙伴，35 家技术共享合作伙伴，并在 2002 年在上海成立中国全资子公司。早在 1999 年，ARM 就已突破 1.5 亿个，市场份额超过了 50%，而在最新的市场调查表明，在 2001 年度里，ARM 占据了整个 32、64 位嵌入式微处理器市场的 75%，在 2002 年度里，占据了整个 32、64 位嵌入式微处理器市场的 79.5%，全世界已使用了 20 多亿个 ARM 核。ARM 已经成为业界的龙头老大，“每个人口袋中装着 ARM”，是毫不夸张的。因为几乎所有的手机、移动设备、PDA 几乎都是用具有 ARM 核的系统芯片开发的。

1.2 基于 ARM 的嵌入式开发环境概述

ARM 技术是高性能、低功耗嵌入式芯片的代名词，在嵌入式尤其是在基于嵌入式 Internet 方面应用广泛。因此，学习嵌入式系统的开发应用技术，应该是基于某种 ARM 核系统芯片应用平台基础上进行，在讲述嵌入式系统开发应用之前，应该对基于 ARM 的嵌入式开发环境进行了解，本节主要对如何构造 ARM 嵌入式开发环境等基本情况介绍。

1.2.1 交叉开发环境

作为嵌入式系统应用的 ARM 处理器，其应用软件的开发属于跨平台开发，因此需要一个交叉开发环境。交叉开发是指在一台通用计算机上进行软件的编辑编译，然后下载到嵌入式设备中进行运行调试的开发方式。用来开发的通用计算机可以选用比较常见的 PC 机、工作站等，运行通用的 Windows 或 Unix 操作系统。开发计算机一般称为主机，嵌入式设备称为目标机，在主机上编译好的程序，下载到目标机上运行，交叉开发环境提供调试工具对目标机上运行的程序进行调试。

交叉开发环境一般由运行于主机上的交叉开发软件(最少必须包含编译调试模块)、主机到目标机的调试通道组成。

运行于主机上的交叉开发软件最少必须包含编译调试模块，其编译器为交叉编译器。作为主机的一般为基于 x86 体系的桌上型计算机，而编译出的代码必须在 ARM 体系结构的目标机上运行，这就是所谓的交叉编译了。在主机上编译好目标代码后，通过主机到目标机的调试通道将代码下载到目标机，然后由运行于主机的调试软件控制代码在目标机上运行调试。为了方便调试开发，交叉开发软件一般为一个整合编辑、编译汇编链接、调试、工程管理及函数库等功能模块的集成开发环境 IDE（Integrated Development Environment）。

组成 ARM 交叉开发环境的主机到目标机的调试通道一般有以下三种：

1) 基于 JTAG 的 ICD(In-Circuit Debugger)。

JTAG 的 ICD 也称为 JTAG 仿真器，是通过 ARM 芯片的 JTAG 边界扫描口进行调试的设备。JTAG 仿真器通过 ARM 处理器的 JTAG 调试接口与目标机通信，通过并口或串口、网口、USB 口与主机通

讯。JTAG 仿真器比较便宜，连接比较方便。通过现有的 JTAG 边界扫描口与 ARM CPU 核通信，属于完全非插入式(即不使用片上资源)调试，它无需目标存储器，不占用目标系统的任何应用端口。通过 JTAG 方式可以完成：

- 读出/写入 CPU 的寄存器，访问控制 ARM 处理器内核。
- 读出/写入内存，访问系统中的存储器。
- 访问 ASIC 系统。
- 访问 I/O 系统
- 控制程序单步执行和实时执行
- 实时地设置基于指令地址值或者基于数据值的断点。

基于 JTAG 仿真器的调试是目前 ARM 开发中采用最多的一种方式。

2)Angel 调试监控软件。

Angel 调试监控软件也称为驻留监控软件，是一组运行在目标板上的程序，可以接收宿主主机上调试器发送的命令，执行诸如设置断点、单步执行目标程序、读写存储器、查看或修改寄存器等操作。宿主主机上的调试软件一般通过串行端口、以太网口、并行端口等通讯端口与 Angel 调试监控软件进行通信。与基于 JTAG 的调试不同，Angel 调试监控程序需要占用一定的系统资源，如内存、通信端口等。驻留监控软件是一种比较低廉有效的调试方式，不需要任何其他的硬件调试和仿真设备。Angel 调试监控程序的不便之处在于它对硬件设备的要求比较高，一般在硬件稳定之后才能进行应用软件的开发，同时它占用目标板上的一部分资源，如内存、通信端口等，而且不能对程序的全速运行进行完全仿真，所以对一些要求严格的情况不是很适合。

3)在线仿真器 ICE(In-Circuit Emulator)。

在线仿真器 ICE 是一种模拟 CPU 的设备，在线仿真器使用仿真头完全取代目标板上的 CPU，可以完全仿真 ARM 芯片的行为，提供更加深入的调试功能。在和宿主主机连接的接口上，在线仿真器也是通过串行端口或并行端口、网口、USB 口通信。在线仿真器为了能够全速仿真时钟速度很高的 ARM 处理器，通常必须采用极其复杂的设计和工艺，因而其价格比较昂贵。在线仿真器通常用在 ARM 的硬件开发中，在软件的开发中较少使用，其价格昂贵，也是在线仿真器难以普及的因素。

1.2.2 模拟开发环境

在很多时候为保证项目进度，硬件和软件开发往往同时进行，这时作为目标机的硬件环境还没有建立起来，软件的开发就需要一个模拟环境来进行调试。模拟开发环境建立在交叉开发环境基础之上，是对交叉开发环境的补充。这时，除了宿主主机和目标机之外，还需要提供一个在宿主主机上模拟目标机的环境，使得开发好的程序直接在这个环境里运行调试。模拟硬件环境是非常复杂的，由于指令集模拟器与真实的硬件环境相差很大，即使用户使用指令集模拟器调试通过的程序也有可能无法在真实的硬件环境下运行，因此软件模拟不可能完全代替真正的硬件环境，这种模拟调试只能作为一种初步调试，主要是用作用户程序的模拟运行，用来检查语法、程序的结构等简单错误，用户最终还必须在真实的硬件环境中实际运行调试，完成整个应用的开发。

1.2.3 评估电路板

评估电路板，也称作开发板，一般用来作为开发者学习板、实验板，可以作为应用目标板出来之前的软件测试、硬件调试的电路板。尤其是对应用系统的功能没有完全确定、初步进行嵌入式开发且没有相关开发经验的非常重要。开发评估电路板并不是 ARM 应用开发必须的，对于有经验的工程师完

全可以自行独立设计自己的应用电路板和根据开发需要设计实验板。好的评估电路板一般文档齐全，对处理器的常用功能模块和主流应用都有硬件实现，并提供电路原理图和相关开发例程与源代码供用户设计自己的应用目标板和应用程序作参考。选购合适于自己实际应用的开发板可以加快开发进度，可以减少自行设计开发的工作量。

1.2.4 嵌入式操作系统

随着嵌入式应用的迅猛发展，以前不怎么知名的嵌入式操作系统概念开始流行起来，以至很多初学者认为嵌入式开发必须采用嵌入式操作系统。实际上，一个嵌入式应用是否采用嵌入式操作系统，采用哪种嵌入式操作系统完全由项目的复杂程度、实时性要求、应用软件规模、目标板硬件资源以及产品成本等因素决定。早期的嵌入式系统并没有操作系统，只不过有一个简单的控制循环而已，对很简单的嵌入式系统开发来说，这可能满足开发需求。随着嵌入式系统在复杂性上的增长，一个操作系统显得重要起来，有些复杂的嵌入式系统也许是因为设计者坚持不要操作系统才使系统开发过程非常复杂。

嵌入式操作系统一般可以提供内存管理、多任务管理、外围资源管理，给应用程序设计带来很多好处，但嵌入式操作系统同时也会占用一定的系统资源，并且要在用户自己的目标板上运行起来，并基于操作系统来设计自己的应用程序，也会相应地带来很多新的问题。所以对于不太复杂的应用完全可以不用操作系统，而对于应用软件规模较大的场合，采用操作系统则可以省掉很多麻烦。嵌入式操作系统是嵌入式开发中一个非常大的课题，目前已有专门的书籍做详细讲解，这里就不进行讨论了。

关于各种嵌入式操作系统及其对 ARM 处理器的支持情况，用户也可以访问网站 <http://www.embed.com.cn> 了解，该网站对目前流行的大多数嵌入式操作系统都有介绍。

用户选用 ARM 处理器开发嵌入式系统时，建立嵌入式开发环境是非常重要的，以上对嵌入式开发环境的基本情况作了简单介绍，一般来说一套具备最基本功能的交叉开发环境是 ARM 嵌入式开发必不可少的，至于嵌入式实时操作系统、评估板等其他开发工具则可以根据应用软件规模和开发计划选用。

1.3 各种 ARM 开发工具简介

用户选用 ARM 处理器开发嵌入式系统时，选择合适的开发工具可以加快开发进度，节省开发成本，用户在建立自己的基于 ARM 嵌入式开发环境时，可供选择的开发工具是非常多的，目前世界上有几十多家公司提供不同类别的 ARM 开发工具产品，根据功能的不同，分别有编译软件、汇编软件、链接软件、调试软件、嵌入式操作系统、函数库、评估板、JTAG 仿真器、在线仿真器等。有些工具是成套提供的，有些工具则需要组合使用。在本节中，我们将简要介绍几种比较流行的 ARM 开发工具，包括 ARM SDT、ARM ADS、Multi 2000、RealViewMDK 等集成开发环境以及 OPENice32-A900 仿真器、Multi-ICE 仿真器、ULink 2 仿真器等。

1.3.1 ARM 的 SDT

ARM SDT 的英文全称是 ARM Software Development Kit，是 ARM 公司 (www.arm.com) 为方便用户在 ARM 芯片上进行应用软件开发而推出的一整套集成开发工具。ARM SDT 经过 ARM 公司逐年的维护和更新，目前的最新版本是 2.5.2，但从版本 2.5.1 开始，ARM 公司宣布推出一套新的集成开发工具 ARM ADS 1.0，取 ARM SDT 而代之，今后将不会再看到 ARM SDT 的新版本。

ARM SDT 由于价格适中，同时经过长期的推广和普及，目前拥有最广泛的 ARM 软件开发用户群体，也被相当多的 ARM 公司的第三方开发工具合作伙伴集成在自己的产品中，比如美国 EPI 公司的 JEENI 仿真器。

ARM SDT（以下关于 ARM SDT 的描述均是以版本 2.50 为对象）可在 Windows95、98、NT 以及 Solaris 2.5/2.6、HP-UX 10 上运行，支持最高到 ARM9（含 ARM9）的所有 ARM 处理器芯片的开发，包括 Strong ARM。

ARM SDT 包括一套完整的应用软件开发工具：

- armcc ARM 的 C 编译器，具有优化功能，兼容于 ANSI C。
- tcc THUMB 的 C 编译器，同样具有优化功能，兼容于 ANSI C。
- armasm 支持 ARM 和 THUMB 的汇编器。
- armlink ARM 连接器，连接一个和多个目标文件，最终生成 ELF 格式的可执行映像文件。
- armsd ARM 和 THUMB 的符号调试器。

以上工具为命令行开发工具，均被集成在 SDT 的两个 Windows 开发工具 ADW 和 APM 中，用户无需直接使用命令行工具。

- APM Application Project Manager，ARM 工程管理器，完全图形界面，负责管理源文件，完成编辑、编译、链接并最终生成可执行映像文件等功能，见下图。

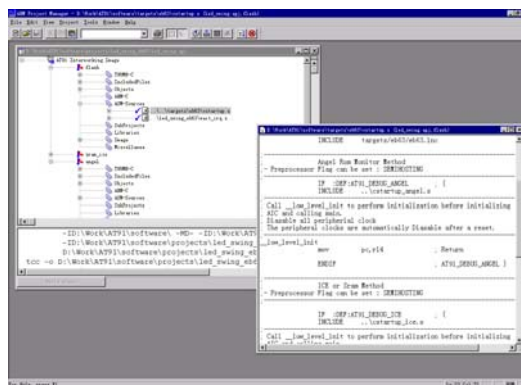


图 1-1 APM 项目管理器窗口

- ADW Application Debugger Windows，ARM 调试工具，ADW 提供一个调试 C、C++ 和汇编源文件的全窗口源代码级调试环境，在此也可以执行汇编指令级调试，同时可以查看寄存器、存储区、栈等调试信息。

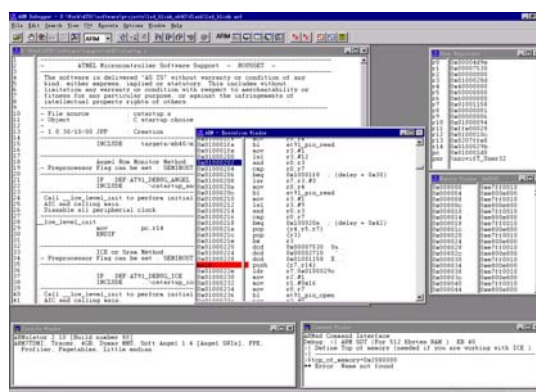


图 1-2 ADW 窗口

ARM SDT 还提供一些实用程序，如 fromELF、armprof、decaxf 等，可以将 ELF 文件转换为不同的格式，执行程序分析以及解析 ARM 可执行文件格式等。

ARM SDT 集成快速指令集模拟器，用户可以在硬件完成以前完成一部分调试工作；ARM SDT 提供 ANSI C、C++、Embedded C 函数库，所有库均以 lib 形式提供，每个库都分为 ARM 指令集和 THUMB 指令集两种，同时在各指令集中也分为高字节结尾（big endian）和低字节结尾（little endian）两种。

用户使用 ARM SDT 开发应用程序可选择配合 Angel 驻留模块或者 JTAG 仿真器进行，目前大部分 JTAG 仿真器均支持 ARM SDT。

1.3.2 ARM 的 ADS

ARM ADS 的英文全称为 ARM Developer Suite，是 ARM 公司推出的新一代 ARM 集成开发工具，用来取代 ARM 公司以前推出的开发工具 ARM SDT。

ARM ADS 起源于 ARM SDT，对一些 SDT 的模块进行了增强并替换了一些 SDT 的组成部分，用户可以感受到的最强烈的变化是 ADS 使用 CodeWarrior IDE 集成开发环境替代了 SDT 的 APM，使用 AXD 替换了 ADW，现代集成开发环境的一些基本特性如源文件编辑器语法高亮，窗口驻留等功能在 ADS 中才得以体现。

ARM ADS 支持所有 ARM 系列处理器包括最新的 ARM9E 和 ARM10，除了 ARM SDT 支持的运行操作系统外还可以在 Windows2000/Me 以及 RedHat Linux 上运行。

ARM ADS 由六部分组成：

- 代码生成工具（Code Generation Tools）

代码生成工具由源程序编译、汇编、链接工具集组成。ARM 公司针对 ARM 系列每一种结构都进行了专门的优化处理，这一点除了作为 ARM 结构的设计者的 ARM 公司，其他公司都无法办到，ARM 公司宣称，其代码生成工具最终生成的可执行文件最多可以比其他公司工具套件生成的文件小 20%。

- 集成开发环境（CodeWarrior IDE from Metrowerks）

CodeWarrior IDE 是 Metrowerks 公司一套比较有名的集成开发环境，有不少厂商将它作为界面工具集成在自己的产品中。CodeWarrior IDE 包含工程管理器、代码生成接口、语法敏感编辑器、源文件和类浏览器、源代码版本控制系统接口、文本搜索引擎等，其功能与 Visual Studio 相似，但界面风格比较独特。ADS 仅在其 PC 机版本中集成了该 IDE。



图 1-3 源程序窗口

- 调试器（Debuggers）

调试器部分包括两个调试器：ARM 扩展调试器 AXD（ARM eXtended Debugger）、ARM 符号调试器 armsd（ARM symbolic debugger）。

AXD 基于 Windows9X/NT 风格，具有一般意义上调试器的所有功能，包括简单和复杂断点设置、栈显示、寄存器和存储区显示、命令行接口等。

Armsd 作为一个命令行工具辅助调试或者用在其他操作系统平台上。

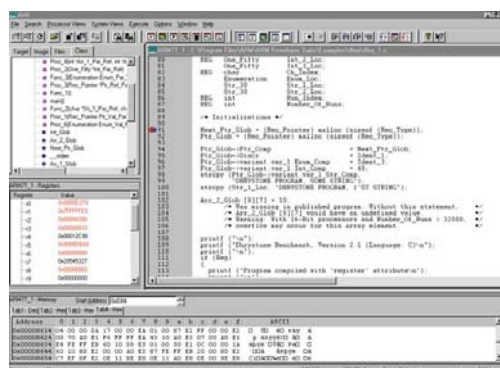


图 1-4 AXD 窗口

- 指令集模拟器（Instruction Set Simulators）

用户使用指令集模拟器无需任何硬件即可在 PC 机上完成一部分调试工作。

- ARM 开发包（ARM Firmware Suite）

ARM 开发包由一些底层的例程和库组成，帮助用户快速开发基于 ARM 的应用和操作系统。具体包括系统启动代码、串行口驱动程序、时钟例程、中断处理程序等，Angel 调试软件也包含在其中。

- ARM 应用库（ARM Applications Library）

ADS 的 ARM 应用库完善和增强了 SDT 中的函数库，同时还包括一些相当有用的提供了源代码的例程。

用户使用 ARM ADS 开发应用程序与使用 ARM SDT 完全相同，同样是选择配合 Angel 驻留模块或者 JTAG 仿真器进行，目前大部分 JTAG 仿真器均支持 ARM ADS。

ARM ADS 的零售价为 5500 美元，如果选用不固定的许可证方式则需要 6500 美元。

1.3.3 Multi 2000

Multi 2000 是美国 Green Hills 软件公司(www.ghs.com)开发的集成开发环境，支持 C/C++/Embedded C++/Ada 95/Fortran 编程语言的开发和调试，可运行于 Windows 平台和 Unix 平台，并支持各类设备的远程调试。

Multi 2000 支持 Green Hills 公司的各类编译器以及其它遵循 EABI 标准的编译器，同时 Multi 2000 支持众多流行的 16 位、32 位和 64 位处理器和 DSP，如 PowerPC、ARM、MIPS、x86、Sparc、TriCore、SH-DSP 等，并支持多处理器调试。

Multi 2000 包含完成一个软件工程所需要的所有工具，这些工具可以单独使用，也可集成第三方系统工具。Multi 2000 各模块相互关系以及和应用系统相互作用如下图所示：

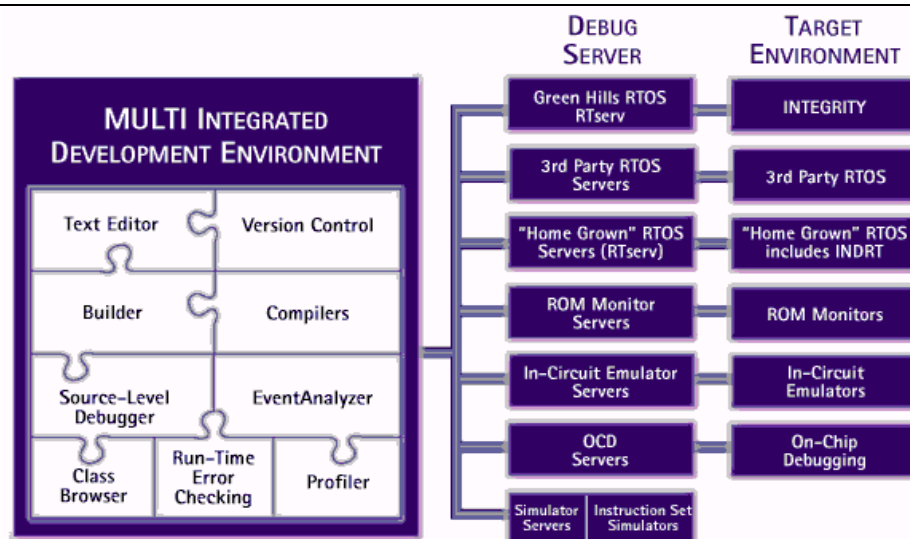


图 1-5 MULTI2000 模块与应用系统

- 工程生成工具（Project Builder）

工程生成工具实现对项目源文件、目标文件、库文件以及子项目的统一管理，显示程序结构，检测文件相互依赖关系，提供编译和链接的图形设置窗口，并可对编程语言的进行特定环境设定。

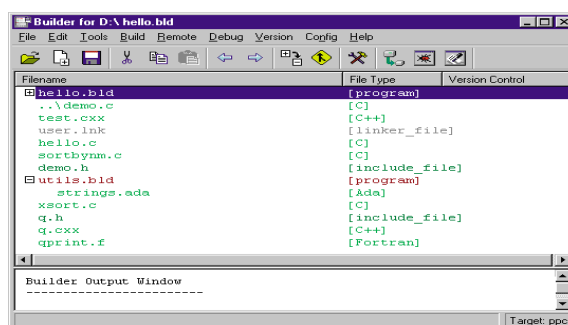


图 1-6 项目生成工具界面

- 源代码调试器（Source-Level Debugger）

源代码调试器提供程序装载、执行、运行控制和监视所需要的强大的窗口调试环境，支持各类语言的显示和调试，同时可以观察各类调试信息。

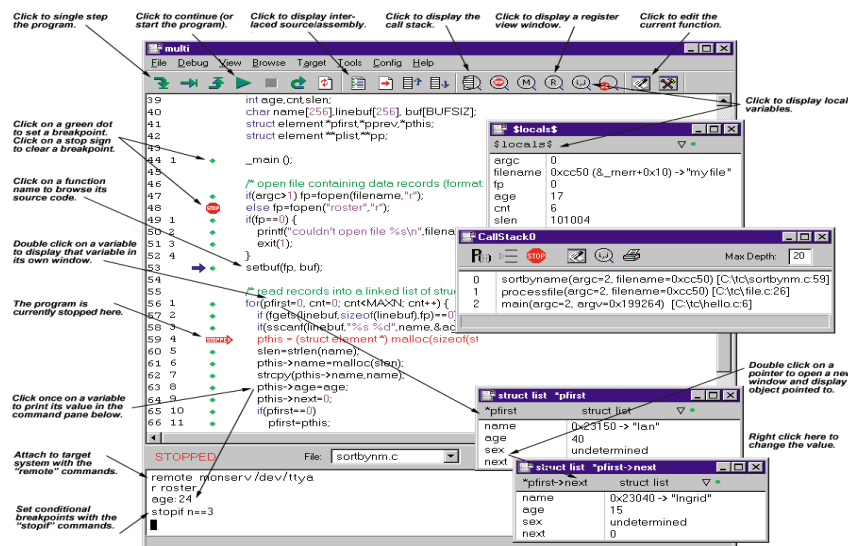


图 1-7 源码级调试器界面信息

● 事件分析器 (Event Analyzer)

事件分析器提供用户观察和跟踪各类应用系统运行和 RTOS 事件的可配置的图形化界面，它可移植到很多第三方工具或集成到实时操作系统中，并对以下事件提供基于时间的测量：任务上下文切换、信号量获取/释放、中断和异常、消息发送/接受、用户定义事件。

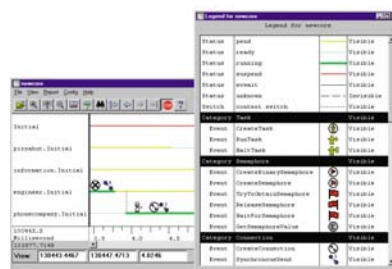


图 1-8 事件分析器界面

● 性能剖析器 (Performance Profiler)

性能剖析器提供对代码运行时间的剖析，可基于表格或图形显示结果，有效的帮助用户优化代码。

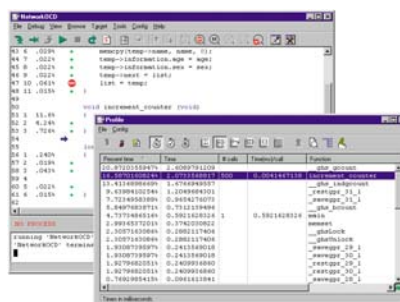


图 1-9 性能剖析器界面

● 实时运行错误检查工具 (Run-Time Error Checking)

实时运行错误检查工具提供对程序运行错误的实时检测，对程序代码大小和运行速度只有极小影响，并具有内存泄漏检测功能。

● 图形化浏览器 (Graphical Brower)

图形化浏览器提供对程序中的类、结构变量、全局变量等系统单元的单独显示，并可显示静态的函数调用关系以及动态的函数调用表。

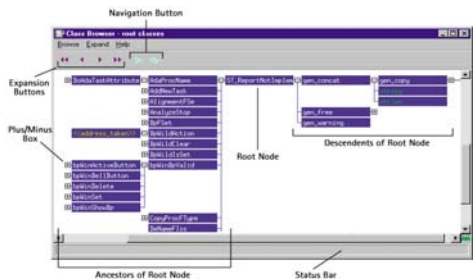


图 1-10 浏览器界面

- 文本编辑器（Text Editor）

Multi 2000 的文本编辑器是一个具有丰富特性的用户可配置的文本图形化编辑工具，提供关键字高亮显示、自动对齐等辅助功能。

- 版本控制工具（Version Control System）

Multi 2000 的版本控制工具和 Multi 2000 环境紧密结合，提供对应用工程的多用户共同开发功能。Multi 2000 的版本控制工具通过配置对支持很多流行的版本控制程序，如 Rational 公司的 ClearCase 等。

1.3.4 RealView MDK

MDK（Microcontroller Development Kit）是 Keil 公司（An ARM Company）开发的 ARM 开发工具，是用来开发基于 ARM 核的系列微控制器的嵌入式应用程序的开发工具。它适合不同层次的开发者使用，包括专业的应用程序开发工程师和嵌入式软件开发的入门者。MDK 包含了工业标准的 C 编译器、宏汇编器、调试器、实时内核等组件，支持所有基于 ARM 的设备，能帮助工程师按照计划完成项目。

Keil ARM 开发工具集集成了很多有用的工具（如图 1-11 所示），正确的使用它们，可以有助于快速完成项目开发。

组件	Part Number	
	MDK-ARM ^{2,3}	DB-ARM
μVision IDE	✓	✓
RealView C/C++ Compiler	✓	
RealView Macro Assembler	✓	
RealView Utilities	✓	
RTL-ARM Real-Time Library	✓	
μVision Debugger	✓	✓
GNU GCC1	✓	✓

图 1-11 MDK 开发工具的组件

以下是 MDK 所包含组件的一些说明：

- μVision IDE 集成开发环境和 μVision Debugger 调试器可以创建和测试应用程序，可以用 RealView、CARM 或者 GNU 的编译器来编译这些应用程序；
- MDK-ARM 是 PK-ARM 的一个超集；
AARM 汇编器、CARM C 编译器、LARM 连接器和 OHARM 目标文件到十六进制的转换器仅包含在 MDK-ARM 开发工具集中。

MDK 的最新版本是 μVision 3，可以开发基于 ARM7、ARM9、Cortex-M3 的微控制器应用程序，它易学易用且功能强大。以下是它的一些主要特性：

- μVision 3 集成了一个能自动配置工具选项的设备数据库；
- 工业标准的 RealView C/C++编译器能产生代码容量最小、运行速度最快的高效应用程序，同

时它包含了一个支持 C++ STL 的 ISO 运行库；

- 集成在 μ Vision 3 中的在线帮助系统提供了大量有价值的信息，可加速应用程序开发速度；
- 包含大量的例程，帮助开发者快速配置 ARM 设备，以及开始应用程序的开发；
- μ Vision 3 集成开发环境能帮助工程人员开发稳健、功能强大的嵌入式应用程序；
- μ Vision 3 调试器能够精确地仿真整个微控制器，包括其片上外设，使得在没有目标硬件的情况下也能测试开发程序；
- 包含标准的微控制器和外部 Flash 设备的 Flash 编程算法；
- ULINK USB-JTAG 仿真器可以实现 Flash 下载和片上调试；
- RealView RL-ARM 具有网络和通信的库文件以及实时软件；
- 还可使用第三方工具扩展 μ Vision 3 的功能；
- μ Vision 3 还支持 GNU 的编译器。

本教程的所有例程均在 MDK 下开发，在第二章中将对 MDK 的使用作详细介绍。

1.3.5 OPENice32-A900 仿真器

OPENice32-A900 仿真器是韩国 AIJI 公司(www.aijisystem.com)生产的。OPENice32-A900 是 JTAG 仿真器，支持基于 ARM7/ARM9/ARM10 核的处理器以及 Intel Xscale 处理器系列。它与 PC 之间通过串口或 USB 口或网口连接，与 ARM 目标板之间通过 JTAG 口连接。OPENice32-A900 仿真器主要特性如下：

- 支持多核处理器和多处理器目标板。
- 支持汇编与 C 语言调试。
- 提供在板(on-board)flash 编程功能。
- 提供存储器控制器设置 GUI。
- 可通过升级软件的方式支持更新的 ARM 核。

OPENice32-A900 仿真器自带宿主主机调试软件 AIJI Spider，但需要使用第三方编译器。AIJI Spider 调试器支持 ELF/DWARF1/DWARF2 等调试符合信息文件格式，可以通过 OPENice32-A900 仿真器下载 BIN 文件到目标板，控制程序在目标板上的运行并进行调试。支持单步、断点设置、查看寄存器/变量/内存以及 Watch List 等调试功能。

OPENice32-A900 仿真器也支持一些第三方调试器，包括 Linux GDB 调试器和 EWARM、ADS/SDT 等调试工具。

1.3.6 Multi-ICE 仿真器

Multi-ICE 是 ARM 公司自己的 JTAG 在线仿真器，目前的最新版本是 2.1 版。

Multi-ICE 的 JTAG 链时钟可以设置为 5 kHz 到 10 MHz，实现 JTAG 操作的一些简单逻辑由 FPGA 实现，使得并行口的通信量最小，以提高系统的性能。Multi-ICE 硬件支持低至 1V 的电压。Multi-ICE 2.1 还可以外部供电，不需要消耗目标系统的电源，这对调试类似于手机等便携式、电池供电设备是很重要的。

Multi-ICE 2.x 支持该公司的实时调试工具 MultiTrace，MultiTrace 包含一个处理器，因此可以跟踪触发点前后的轨迹，并且可以在不终止后台任务的同时对前台任务进行调试，在微处理器运行时改变存储器的内容，所有这些特性使延时降到最低。

Multi-ICE 2.x 支持 ARM7、ARM9、ARM9E、ARM 10 和 Intel Xscale 微结构系列。它通过 TAP 控制器串联，提供多个 ARM 处理器以及混合结构芯片的片上调试。它还支持低频或变频设计以及超低压核的调试，并且支持实时调试。

Multi-ICE 提供支持 Windows NT4.0、Windows95/98/2000/Me、HPUX 10.20 和 Solaris V2.6/7.0 的驱动程序。

Multi-ICE 主要优点：

- 快速的下载和单步速度。
- 用户控制的输入/输出位。
- 可编程的 JTAG 位传送速率。
- 开放的接口，允许调试非 ARM 的核或 DSP。
- 网络连接到多个调试器。
- 目标板供电，或外接电源。

1.3.7 ULINK 2 仿真器

ULINK 是 Keil 公司提供的 USB-JTAG 接口仿真器，目前最新的版本是 2.0。它支持诸多芯片厂商的 8051、ARM7、ARM9、Cortex M3、Infineon C16x、Infineon XC16x、Infineon XC8xx、STMicroelectronics μ PSD 等多个系列的处理器。ULINK 2 内部实物如图 1-12 所示，电源由 PC 机的 USB 接口提供。ULINK2 不仅包含了 ULINK USB-JTAG 适配器具有的所有特点，还增加了串行线调试（SWD）支持，返回时钟支持和实时代理功能。ULINK2 适用与标准的 Windows USB 驱动等功能。

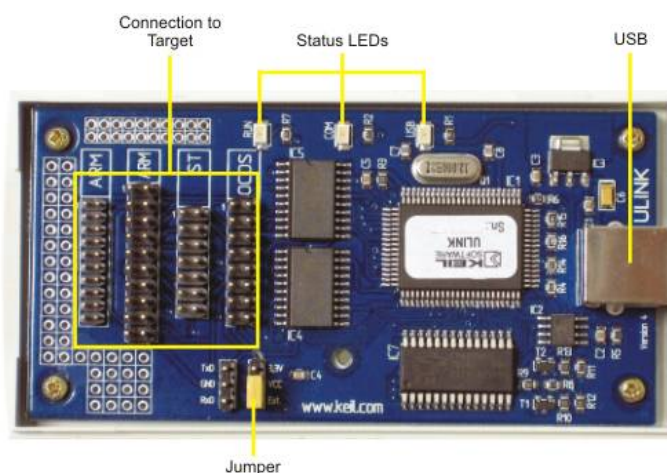


图 1-12 ULINK 2（无盖）

ULINK 2 的主要功能：

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

ULINK2 新特点

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

本教程中所有的例程使用的 ULINK USB-JTAG 仿真器套件即 ULINK 2 仿真器。

1.4 如何学习基于 ARM 嵌入式系统开发

ARM 微处理器因其卓越的低功耗、高性能在 32 位嵌入式应用中已位居世界第一，是高性能、低功耗嵌入式处理器的代名词，为了顺应当今世界技术革新的潮流，了解、学习和掌握嵌入式技术，就

必然要学习和掌握以 ARM 微处理器为核心的嵌入式开发环境和开发平台，这对于研究和开发高性能微处理器、DSP 以及开发基于微处理器的 SOC 芯片设计及应用系统开发是非常必要的。

那么究竟如何学习嵌入式的开发和应用？技术基础是关键。技术基础决定了学习相关知识、掌握相关技能的潜能。嵌入式技术融合具体应用系统技术、嵌入式微处理器/DSP 技术、系统芯片 SOC 设计制造技术、应用电子技术和嵌入式操作系统及应用软件技术，具有极高的系统集成性，可以满足不断增长的信息处理技术对嵌入式系统设计的要求。因此学习嵌入式系统首先是基础知识学习，主要是相关的基本硬件知识，如一般处理器及接口电路（Flash/ SRAM/SDRAM /Cache, UART, Timer, GPIO, Watchdog、USB、IIC 等...）等硬件知识，至少了解一种 CPU 的体系结构；至少了解一种操作系统（中断，优先级，任务间通信，同步...）。对于应用编程，要掌握 C、C++ 及汇编语言程序设计（至少会 C），对处理器的体系结构、组织结构、指令系统、编程模式、一般对应用编程要有一定的了解。在此基础上必须在实际工程实践中掌握一定的实际项目开发技能。

其次对于嵌入式系统开发的学习，必须要有一个较好的嵌入式开发教学平台。功能全面的开发平台一方面为学习提供了良好的开发环境，另一方面开发平台本身也是一般的典型实际应用系统。在教学平台上开发一些基础例程和典型实际应用例程，对于初学者和进行实际工程应用也是非常必要的。

嵌入式系统的学习必须对基本内容有深入的了解。在处理器指令系统、应用编程学习的基础上，重要的是加强外围功能接口应用的学习，主要是人机接口、通讯接口，如 USB 接口、A/D 转换、GPIO、以太网、IIC 串行数据通信、音频接口、触摸屏等知识的掌握。

嵌入式操作系统也是嵌入式系统学习重要的一部分，在此基础上才能进行各种设备驱动应用程序的开发。

第二章 Embest ARM 实验教学系统

2.1 教学系统介绍

Embest ARM 教学系统包括 μ Vision IDE 集成开发环境，ULINK USB-JTAG 仿真器，Embest EduKit-III 开发板、各种连接线、电源适配器以及实验指导书等。基本实验模型示意图如 2-1 所示：



图 2-1 实验模型示意图

2.1.1 μ Vision 3 集成开发环境

1) μ Vision 3 是一个基于窗口的软件开发平台,它集成了功能强大的编辑器、工程管理器以及 make 工具。 μ Vision3 IDE 集成的工具包括 C 编译器、宏汇编器、链接/定位器和十六进制文件生成器。 μ Vision 有编译和调试两种工作模式,两种模式下设计人员都可查看并修改源文件。图 2-2 是编译模式下典型的窗口配置, μ Vision IDE 由多个窗口、对话框、菜单栏、工具栏组成。其中菜单栏和工具栏用来实现快速的命令;工程工作区 (Project Workspace) 用于项目管理、寄存器调试、函数管理、手册管理等;输出窗口 (Output Window) 用于显示编译信息、搜索结果以及调试命令交互灯;内存窗口 (Memory Window) 可以不同格式显示内存中的内容;观测窗口 (Watch & Call Stack Window) 用于观察、修改程序中的变量以及当前的函数调用关系;工作区 (Workspace) 用于文件编辑、反汇编输出和一些调试信息显示;外设对话框 (Peripheral Dialogs) 帮助设计者观察片内外围接口的工作状态。

2) μ Vision IDE 主要功能特点及组件

μ Vision IDE 可在 Windows 98、2000、NT 及 XP 等操作系统上运行,主要支持基于 ARM7、ARM9、Cortex-M3 系列处理器,目前最新的版本为 μ Vision 3,其主要特点如下:

- μ Vision 3 集成了一个能自动配置工具选项的设备数据库;
- 工业标准的 RealView C/C++ 编译器能产生代码容量最小、运行速度最快的高效应用程序,同时它包含了一个支持 C++ STL 的 ISO 运行库;
- 集成在 μ Vision 3 中的在线帮助系统提供了大量有价值的信息,可加速应用程序开发速度;
- 包含大量的例程,帮助开发者快速配置 ARM 设备,以及开始应用程序的开发;
- μ Vision 3 集成开发环境能帮助工程人员开发稳健、功能强大的嵌入式应用程序;
- μ Vision 3 调试器能够精确地仿真整个微控制器,包括其片上外设,使得在没有目标硬件的情况下也能测试开发程序;
- 包含标准的微控制器和外部 Flash 设备的 Flash 编程算法;
- ULINK USB-JTAG 仿真器可以实现 Flash 下载和片上调试;
- RealView RL-ARM 具有网络和通信的库文件以及实时软件;
- 还可使用第三方工具扩展 μ Vision 3 的功能;
- μ Vision 3 还支持 GNU 的编译器。

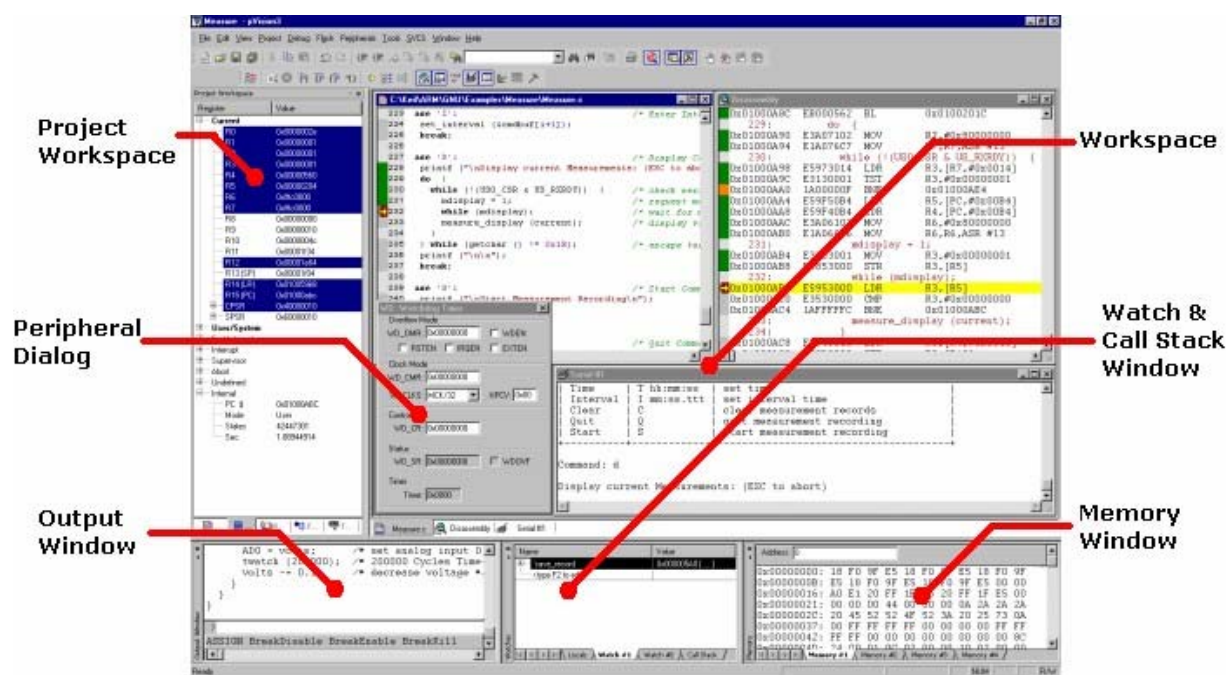


图 2-2 μVision IDE 开发环境软件界面

μVision IDE 包含以下功能组件，能加速嵌入式应用程序开发过程：

- 功能强大的源代码编辑器；
- 可根据开发工具配置的设备数据库；
- 用于创建和维护工程的工程管理器；
- 集汇编、编译和链接过程于一体的编译工具；
- 用于设置开发工具配置的对话框；
- 真正集成高速 CPU 及片上外设模拟器的源码级调试器；
- 高级 GDI 接口，可用于目标硬件的软件调试和 Keil ULINK 仿真器的连接；
- 用于下载应用程序到 Flash ROM 中的 Flash 编程器；
- 完善的开发工具手册、设备数据手册和用户向导。

μVision IDE 使用简单、功能强大，是保证设计者完成设计任务的重要保证。μVision IDE 还提供了大量的例程及相关信息，有助于开发人员快速开发嵌入式应用程序。

μVision IDE 有编译和调试两种工作模式。编译模式用于维护工程文件和生成应用程序；调试模式下，则可以用功能强大的 CUP 和外设仿真器来测试程序，也可以使用调试器经 Keil ULINK USB-JTAG 适配器（或其他 AGDI 驱动器）来连接目标系统测试应用程序。ULINK 仿真器能用于下载应用程序到目标系统的 Flash ROM 中。

在嵌入式软件开发时，完成设计和编码后，即开始调试程序，这是软件开发的第三步。一个几千行的程序，其编译可达到没有一个警告，然而在运行时却可能达不到正常的设计需求、甚至系统无法运行起来而崩溃，更为难以查找的是系统运行只是在偶然的情况下出现问题或崩溃。当程序不能顺利运行，而又不能简单、直观的分析、知道问题的症结所在时，就该使用调试器来监视此程序的运行了。μVision IDE 调试器提供程序装载、执行、运行控制和监视所需要的强大的窗口调试环境，支持源码显

示和调试，同时可以观察各类调试信息。 μ Vision IDE 具有功能强大的调试器， μ Vision 3 调试器用于调试和测试应用程序，它提供了两种操作模式：仿真模式和 GDI 驱动器模式。可以在 Options for Target – Debug 对话框内进行选择，如图 2-3 所示。

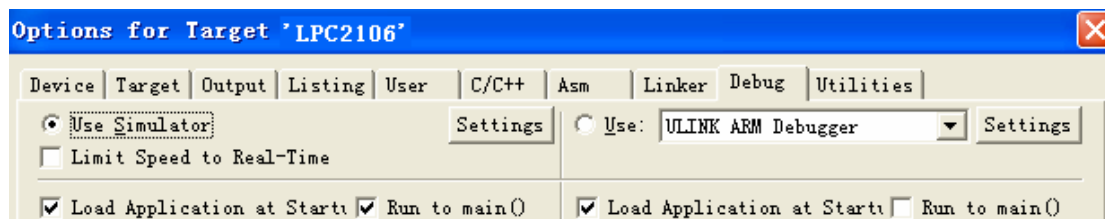


图 2-3 调试器操作模式的选择

◆ 仿真模式

仿真模式可在无目标系统硬件情况下，仿真微控器的许多特性。可在目标硬件准备好之前，把 μ Vision 3 调试器配置为软件仿真，可以测试和调试所开发的嵌入式应用， μ Vision 3 能仿真大量的外围设备包括串口、外部 I/O 及时钟等。在为目标程序选择 CPU 时，相应外围接口就被从设备库中选定。

◆ GDI 驱动器模式

GDI 驱动模式下，可使用高级 GDI 驱动器，例如 ULINK ARM Debugger 来连接目标硬件。对 μ Vision 3 来说，以下几种驱动器均可用于连接目标硬件：

- JTAG/OCDS 适配器：连接到片上的调试系统，如 AMR Embedded ICE；
- 监视器：可以集成在用户硬件上、也可以在许多评估板上；
- 仿真器：连接到目标硬件的 CPU 引脚上
- 测试硬件：例如 Infineon SmartCard ROM 监视器，或 Philips SmartMX DBox。

2.1.2 ULINK USB-JTAG 仿真器

JTAG 仿真器也称为 JTAG 调试器，是通过 ARM 芯片的 JTAG 边界扫描口进行调试的设备。JTAG 仿真器连接比较方便，通过现有的 JTAG 边界扫描口与 ARM CPU 核通信，属于完全非插入式(即不使用片上资源)调试，它无需目标存储器，不占用目标系统的任何端口，而这些是驻留监控软件所必需的。

另外，由于 JTAG 调试的目标程序是在目标板上执行，仿真更接近于目标硬件，因此，许多接口问题，如高频操作限制、AC 和 DC 参数不匹配，电线长度的限制等被最小化了。使用集成开发环境配合 JTAG 仿真器进行开发是目前采用最多的一种调试方式。ULINK USB-JTAG 仿真器如图 2-4 (a) 所示。

ULINK USB-JTAG 仿真器支持众多 Philips、Samsung、Atmel、Analog Devices、Sharp、ST 等众多厂商 ARM7 及 ARM9 内核的 ARM 微控制器，其将 PC 机的 USB 端口与用户的目标硬件相连(通过 JTAG 或 OCD)，使用户可在目标硬件上调试代码。通过使用 Keil μ Vision IDE/调试器和 ULINK USB-JTAG 仿真器，用户可以方便地编辑、下载和在实际的目标硬件上测试嵌入式的程序，并且具有三个 LED 灯分别显示 RUN, COM, 和 USB 状态。使用 ULINK USB-JTAG 仿真器可以实现以下功能：

- USB 通讯接口高速下载用户代码

- 存储区域/寄存器查看
- 快速单步程序运行
- 添加多个程序断点
- 运行实时程序
- 片内 Flash 编程
- 运行实时程序



(a) ULINK USB-JTAG 仿真器图

(b) ULINK2 USB-JTAG 仿真器图

图 2-4 ULINK 系列仿真器图

ULINK2 通过 JTAG, SWD, 或 OCDS 将目标硬件与您电脑的 USB 端口连接起来, 使用 ULINK2 您可以调试在目标硬件上运行的嵌入式程序。ULINK2 不仅包含了 ULINK USB-JTAG 适配器具有的所有功能, 而且具有如下新的特点:

- 标准 Windows USB 驱动支持 ULINK2 即插即用
- 支持基于 ARM Cortex-M3 的串行线调试
- 支持程序运行期间的存储器读写、终端仿真和串行调试输出
- 既支持 20 针引脚, 同时也支持 10 针引脚
- ULINK2 仿真器图如图 2-4 (b) 所示。

2.1.3 Embest EduKit-III 嵌入式教学实验平台

Embest EduKit-III 嵌入式教学实验平台是英蓓特公司针对高校需求, 在 II 型的基础上研发的最新的第三代教学系统。Embest EduKit-II 自从推出市场以来, 受到了老师们的一致认可。在此基础上, 英蓓特公司综合了众多使用老师的意见, 在 II 型实验系统基础上整合了更多的资料, 整理了布局, 使整个实验系统更加合理, 更加精致, 推出了 Embest EduKit-III 教学实验系统。该硬件平台如图 2-5 所示。



图 2-5 实验系统硬件平台

Embest EduKit-III 开发板的基本资源如下：

- 采用多 CPU 子板
 - ARM7 S3C44B0 子板
 - ARM9 S3C2410 子板
 - DSP Blackfin 533 子板（选配）
 - Intel Xscale270 子板（选配）
- 64M NandFlash, 2—32M NorFlash
- 64M SDRAM
- 4Kbit IIC BUS 的串行 EEPROM
- 2 个 232 串口，一个 422 串口，一个 485 串口
- 两个中断按钮，一个复位按钮，4 个 LED
- 5.7 寸 320*240 STN 彩色 LCD 及 TSP 触摸屏
- 4 × 5 键盘
- 20 针 JTAG 接口
- PS/2 接口
- 2 个 USB 主口
- 1 个 USB 从口
- SD 接口模块
- PCI 扩展接口
- 以太网接口
- 8 段数码管
- 双 CAN 总线模块
- A/D 模块

- IDE 硬盘接口模块
- CF 卡接口模块
- CPLD 模块
- 直流电机模块
- 步进电机模块
- MICROPHONE 输入口
- IIS 音频信号输出口
- 高速 USB2.0 Ulink2 仿真器一个
- YS244-JTAG 简易仿真头一个
- 固态硬盘 32M × 8bit (选配)
- GPRS 模块 (选配)
- GPS 模块 (选配)
- 蓝牙 (选配)
- 摄像头模块 (选配)
- WIFI 模块 (选配)
- DAC 模块 (选配)

2.1.4 各种连接线与电源适配器

实验系统除了提供以上的组件以外，还提供了各种连接时候需要的电缆线。包括交叉网线，USB 线，交叉串口线，并口线和两根 JTAG 线（分别是 20 针和 14 针接口）。

2.2 教学系统安装

Embest ARM 教学系统包括 μ Vision 3 集成开发环境、ULINK USB-JTAG 仿真器、Embest EduKit-III、各种连接线等。其中 μ Vision 3 属软件平台部分，其余属于硬件平台部分。

本节主要介绍如何安装实验系统的软件平台、如何搭建和如何进行软件平台与硬件平台的连接。

在安装 μ Vision3 IDE 集成开发环境之前请首先阅读软件使用许可协议）

安装 μ Vision 3 评估软件必须满足的最小的系统要求为：

- 操作系统：Windows 98，Windows NT4，Windows 2000，Windows XP；
- 硬盘空间：30M 以上；
- 内存：128M 以上。

μ Vision IDE 集成开发环境的安装方法如下：

- (1) 购买 MDK 的安装程序，或从 <http://www.realview.com.cn/xz-down.asp> 下载 MDK 的评估版；

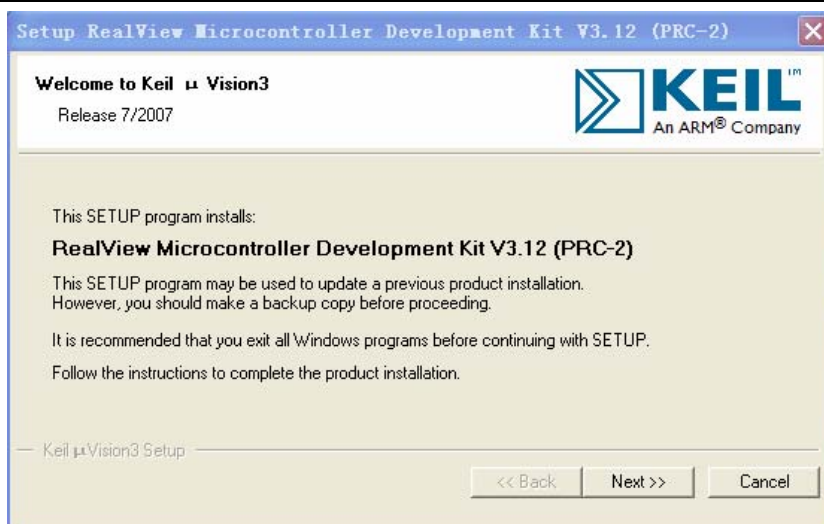


图 2-6 MDK 安装界面 1

(2) 双击安装文件，弹出如图 2-6 对话框。建议在安装之前关闭所有的应用程序，单击 Next，弹出如图 2-7 所示对话框；



图 2-7 MDK 安装界面 2

(3) 仔细阅读许可协议，选中 I agree to all the terms of the preceding License Agreement 选项，单击 Next，弹出如图 2-8 所示对话框；

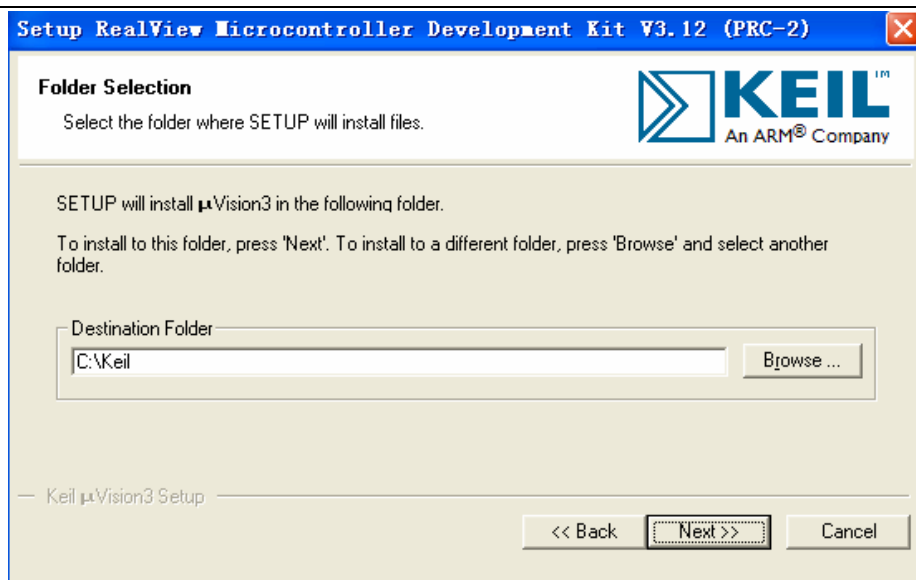


图 2-8 MDK 安装界面 3

(4) 单击 Browse 选择安装路径，然后单击 Next，弹出如图 2-9 所示对话框；

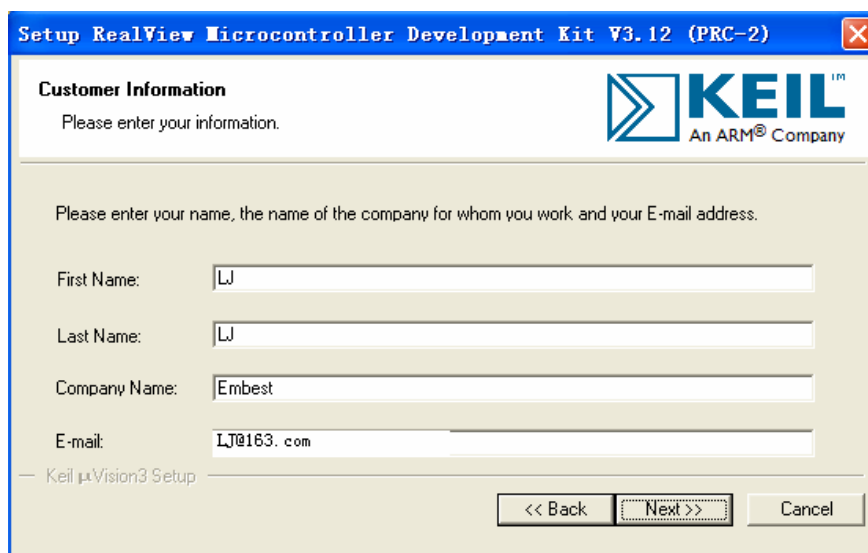


图 2-9 MDK 安装界面 4

(5) 输入 First Name、Last Name、Company Name 以及 E-mail 地址后，单击 Next；安装程序将在计算机上安装 MDK，依据机器性能的不同，安装程序大概耗时半分钟到两分钟不等，之后将会弹出如图 2-10 所示对话框，单击 Finish 结束安装。至此，开发人员就可在计算机上使用 MDK 软件来开发应用程序了。

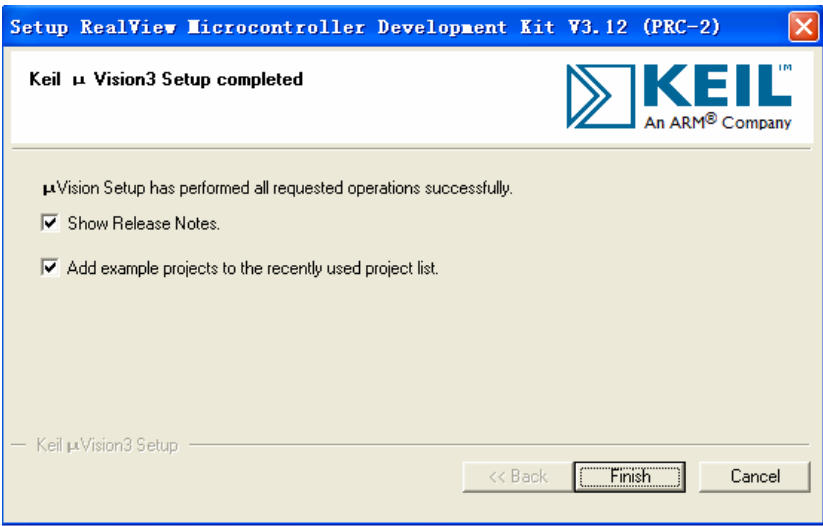



图 2-10 MDK 安装界面 5

μVision IDE 集成开发环境安装完毕后，点击 μVision IDE 的图标  即可运行 μVision IDE。第一次使用 μVision IDE 正式版时，用户必须注册。μVision 3 的有两种许可证：单用户许可证和浮动许可证。单用户许可证只允许单用户最多在二台计算机上使用 MDK，而浮动许可证则允许局域网众多台计算机分时使用 MDK。目前，所有的 Keil 软件均可使用单用户许可证注册，绝大多数 Keil 软件可使用浮动许可证注册。下面分别介绍两种许可证注册：

■ 单用户许可证注册过程

- (1) 安装好 μVision 3；
- (2) 在 μVision IDE 中，单击 File – License Management 菜单项进入许可证管理对话框；
- (3) 选择 Single-User License 页，在该页右边的 CID（Computer ID）文本框中会自动产生 CID；
- (4) 用CID和MDK提供的PSN（产品序列号）在 <https://www.keil.com/license/embest.htm>注册，确保输入邮箱的正确性；
- (5) 通过注册后，在所填写注册信息的邮箱中将会收到许可证 ID 码 LIC(License ID Code)；
- (6) 将得到的许可证 ID 输入 New License ID Code (LIC) 文本框，然后单击右边的 Add LIC 按钮，此时这册成功，如图 2-11 所示：

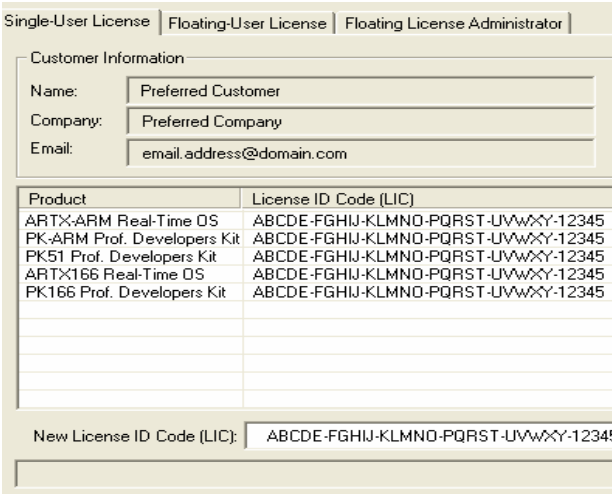


图 2-11 单用户许可证注册成功后界面

■ 浮动许可证注册过程

浮动许可证的注册过程比单用户许可证的注册过程复杂。由于它是局域网上多用户分时复用的，所以必须保证浮动许可证在公用的网络服务器上，从而保证开发团队中的每一个人都能使用该许可证。下面是浮动许可证的注册过程：

- (1) 安装好 μ Vision 3；
- (2) μ Vision IDE 中，单击 File – License Management 菜单项进入许可证管理对话框；
- (3) 选择 Floating License 页，在该页右边的 CID 文本框中会自动产生 Computer ID；
- (4) 单击“增加产品”按钮，选择浮动许可证文件（由浮动许可证管理员创建）的路径然后单击“确认”按钮，它会自动打开 <https://www.keil.com/license/lic30floating.asp> 浮动许可证注册页面；
- (5) 在上述页面中输入 CID 和浮动许可证码 PSN 以及其他相关信息，确保输入的电子邮箱地址的正确性；
- (6) 通过注册后，在所填写注册信息的邮箱中将会收到许可证 ID 码；
- (7) 将得到的许可证 ID 输入 New License ID Code (LIC) 文本框，然后单击右边的 Add LIC 按钮，此时注册成功，如图 2-12 所示。

Product	License ID Code (LIC)	Support
ARTX-ARM Real-Time OS	ABCDE-FGHIJ-KLMNO-PQRST-UVWXY-12345	Expires: .
PK-ARM Prof. Developers Kit	ABCDE-FGHIJ-KLMNO-PQRST-UVWXY-12345	Expires: .
PK51 Prof. Developers Kit	ABCDE-FGHIJ-KLMNO-PQRST-UVWXY-12345	Expires: .
ARTX166 Real-Time OS	ABCDE-FGHIJ-KLMNO-PQRST-UVWXY-12345	Expires: .
PK166 Prof. Developers Kit 3 user	ABCDE-FGHIJ-KLMNO-PQRST-UVWXY-12345	Expires: .

图 2-12 浮动许可证注册成功后界面

使用浮动许可证注册的某个网络用户，如果要使用 MDK，可单击图 2-12 对话框右边的 Check out 按钮，该用户将获得许可证，此时开发小组的其他用户将只能使用 MDK 评估版的功能。该用户使用完之后，需及时单击 Check in 按钮（默认的情况下一个小时后 MDK 将自动 Check in），否则其它用户将无法正常使用 MDK。

■ 浮动许可证的管理

- (1) 安装好 μ Vision 3；
- (2) μ Vision IDE 中，单击 File – License Management 菜单项进入许可证管理对话框；
- (3) 选择 Floating License Administrator 页，如图 2-13 所示在 Path 文本框中设置正确的服务器共享文件夹的路径；
- (4) 在 PSN 文本框中输入正确的产品序列号；
- (5) 单击 Create FLE 按钮，即在服务器共享文件夹中产生注册浮动许可证时需要的 FLE 文件。

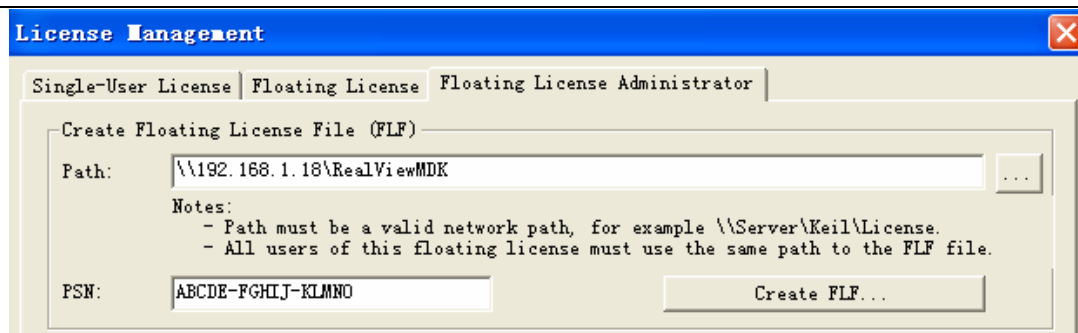


图 2-13 正确创建 FLE 文件后的界面

软件安装完毕后，请详细阅读相关软件说明及软件使用手册。

实验软件平台和硬件平台的连接如图 2-1 所示，PC 端与仿真器通过实验系统提供的并口线连接，仿真器和开发板通过一根 20 针的 JTAG 线连接。

其中需要注意：

1) 仿真器驱动程序在安装 μ Vision IDE 时自动安装，第一次使用 ULINK 仿真器时，PC 机会自动加载其驱动程序，该驱动程序已在 Windows 2000、Windows XP 和 Windows XP SP2 上测试通过，如果驱动不能自动加载，可以访问 <http://www.keil.com/support/>。

2) 硬件平台最好预先参照 Embest EduKit-III 用户手册（在 Embest ARM 教学系统光盘中）进行基本硬件检测。

2.3 集成开发环境使用说明

2.3.1 μ Vision IDE 主框架窗口

μ Vision IDE 由如图 2-2 所示的多个窗口、对话框、菜单栏、工具栏组成。其中菜单栏和工具栏用来实现快速的操作命令；工程工作区（Project Workspace）用于文件管理、寄存器调试、函数管理、手册管理等；输出窗口（Output Window）用于显示编译信息、搜索结果以及调试命令交互灯；内存窗口（Memory Window）可以不同格式显示内存中的内容；观测窗口（Watch & Call Stack Window）用于观察、修改程序中的变量以及当前的函数调用关系；工作区（Workspace）用于文件编辑、反汇编输出和一些调试信息显示；外设对话框（Peripheral Dialogs）帮助设计者观察片内外围接口的工作状态。

本节将主要介绍 μ Vision IDE 的菜单栏、工具栏、常用快捷方式，以及各种窗口的内容和使用方法，以便让读者能快速了解 μ Vision IDE，并能对 μ Vision IDE 进行简单和基本的操作。

μ Vision IDE 集成开发环境的菜单栏可提供如下菜单功能：编辑操作、工程维护、开发工具配置、程序调试、外部工具控制、窗口选择和操作，以及在线帮助等。工具栏按钮可以快速执行 μ Vision 3 的命令。状态栏显示了编辑和调试信息，在 View 菜单中可以控制工具栏和状态栏是否显示。键盘快捷键可以快速执行 μ Vision 3 的命令，它可以通过菜单命令 Edit - Configuration - Shortcut Key 来进行配置。

2.3.2 工程管理

1. 工程管理介绍

在 μ Vision IDE 集成开发环境中，工程是一个非常重要的概念，它是用户组织一个应用的所有源文件、设置编译链接选项、生成调试信息文件和最终的目标 Bin 文件的一个基本结构。一个工程管理一个应用程序的所有源文件、库文件、其它输入文件，并根据实际情况进行相应的编译链接设置，一个工程须生成一个相对应的目录，以进行文件管理。

μVision IDE 工程管理提供以下功能：

μVision IDE 的工作区由五部分组成，分别为 Files（文件）页、工作区如图 2-14 所示，它显示了工程结构。

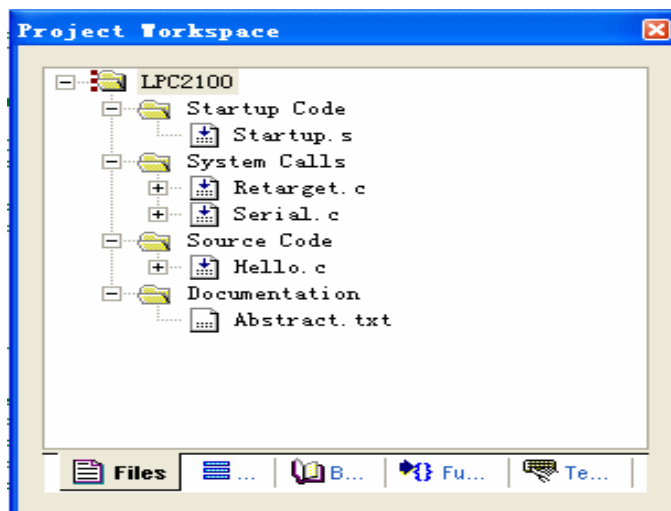


图 2-14 工程管理区结构图

- 以工程为单位定义设置应用程序的各选项，包括目标处理器和调试设备的选择与设置，调试相关信息的配置，以及编译、汇编、链接等选项的设置等。系统提供一个专门的对话框来设置这些选项。
- 提供 build 菜单和工具按钮，让用户轻松进行工程的编译、链接。编译、链接信息输出到输出窗口中的 Build 标签窗中，如图 2-15 所示，编译链接出现的错误，通过鼠标左键双击错误信息提示行来定位相应的源文件行。

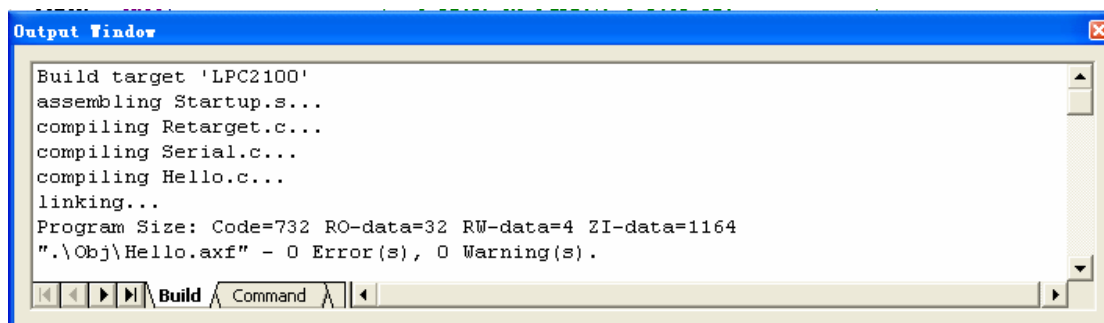


图 2-15 编译链接输出子窗口

- 一个应用工程编译链接后根据编译器的设置生成相应格式的调试信息文件，调试通过的程序转换成二进制格式的可执行文件后最终在目标板上运行。

2. 工程的创建

μVision 3 所提供的工程管理，使得基于 ARM 处理器的应用程序设计开发变得越来越方便。通常使用 μVision 3 创建一个新的工程需要以下几步：选择工具集、创建工程并选择处理器、创建源文件及文件组、配置硬件选项、配置对应启动代码、最后编译链接生成 HEX 文件。

1) 选择工具集

利用 μVision 3 创建一个基于处理器的应用程序，首先要选择开发工具集。单击 **Project - Manage-Components, Environment, and Books** 菜单项，在如图 2-16 所示对话框中，可选择所使用的工具集。在 μVision 3 中既可使用 ARM RealView 编译器、GNU GCC 编译器，也可以使用 K

el CARM 编译器。当使用 GNU GCC 编译器时，需要安装相应的工具集。在本例程中选择 ARM RealView 编译器，MDK 环境默认的编译器，可不用配置。

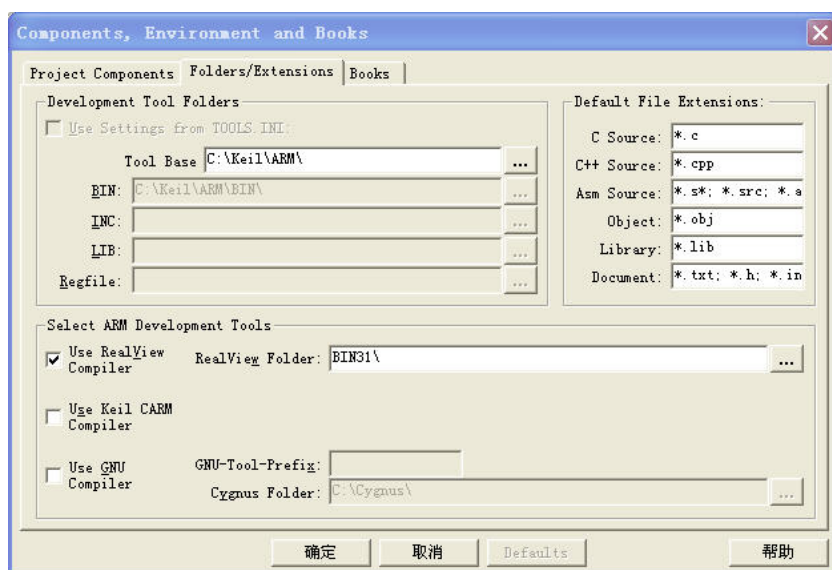


图 2-16 选择工具集

2) 创建工程并选择处理器

单击 **Project - New μ Vision Project...** 菜单项， μ Vision 3 将打开一个标准对话框，输入希望新建工程的名字即可创建一个新的工程，建议对每个新建工程使用独立的文件夹。这里先建立一个新的文件夹 **Hello**，在前述对话框中输入 **Hello**， μ Vision 将会创建一个以 **Hello.UV2** 为名字的新工程文件，它包含了一个缺省的目标（target）和文件组名。这些内容在 **Project Workspace** 窗口中可以看到。

创建一个新工程时， μ Vision 3 要求设计者为工程选择一款对应处理器，如图 2-17 所示，该对话框中列出了 μ Vision 3 所支持的处理器设备数据库，也可单击 **Project - Select Device...** 菜单项进入此对话框。选择了某款处理之后， μ Vision 3 将会自动为工程设置相应的工具选项，这使得工具的配置过程简化。

对于大部分处理器设备， μ Vision 3 会提示是否在目标工程里加入 CPU 的相关启动代码。如图 2-18 所示。启动代码是用来初始化目标设备的配置，完成运行时系统的初始化工作，对于嵌入式系统开发而言是必不可少的，单击 **Ok** 便可将启动代码加入工程，这使得系统的启动代码编写工作量大大减少。

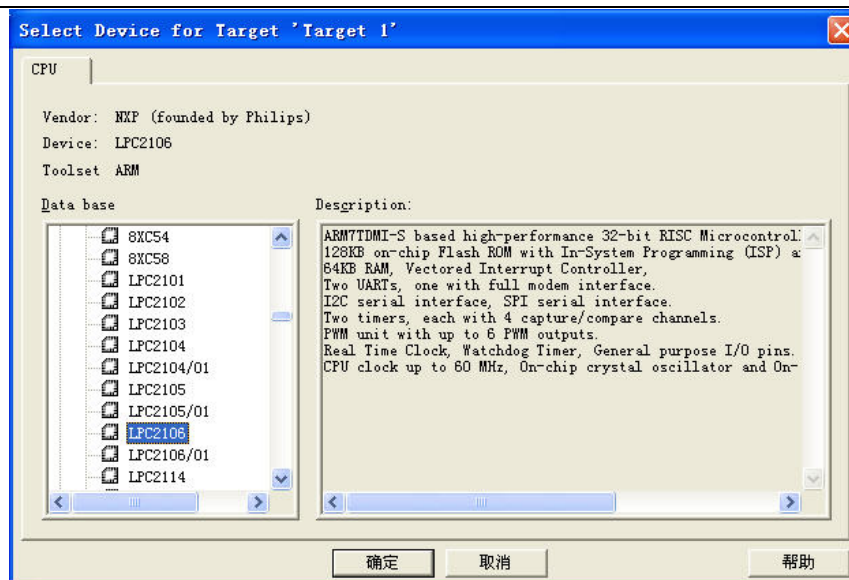


图 2-17 选择处理器



图 2-18 加入启动代码

在设备数据库中为工程选择 CPU 后，Project Workspace - Books 内就可以看到相应设备的用户手册，以供设计者参考，如图 2-19 所示。

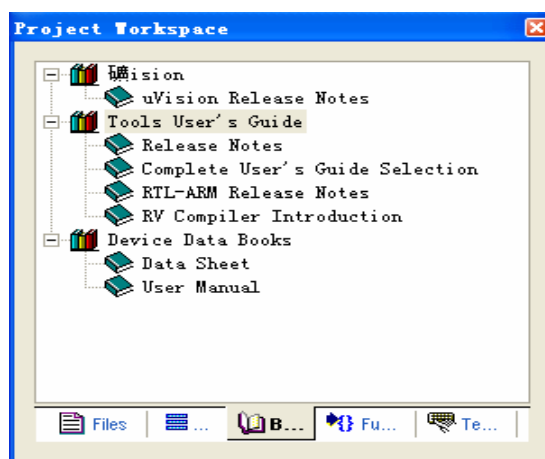


图 2-19 相应设备数据手册

3. 建立一个新的源文件

创建一个工程之后，就应开始编写源程序。选择菜单项 File - New 可创建新的源文件，μVision IDE 将会打开一个空的编辑窗口用以输入源程序。在输入完源程序后，选择 File - Save As... 菜单

项保存源程序，当以*.C 为扩展名保存源文件时，μVision IDE 将会根据语法以彩色高亮字体显示源程序。

4. 工程中文件的加入

创建完源文件后便可以在工程里加入此源文件，μVision 提供了多种方法加入源文件到工程中。例如，在 Project Workspace – Files 菜单项中选择文件组，右击将会弹出如图 2-20 所示快捷菜单，单击选项 Add Files to Group... 打开一个标准文件对话框，将已创建好的源文件加入到工程中。

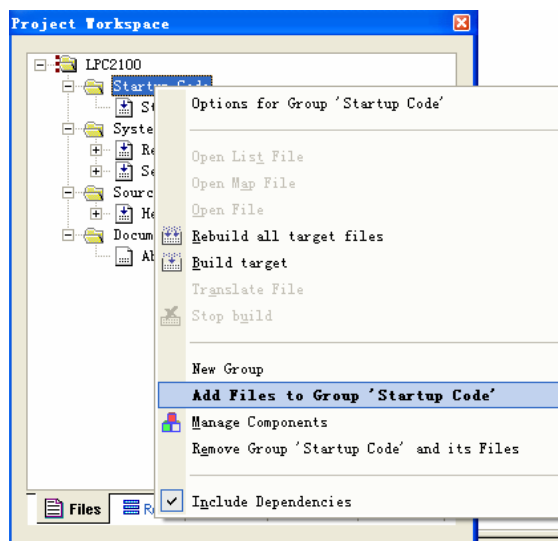


图 2-20 加入源文件到工程中

通常，设计人员应采用文件组来组织大的工程，将工程中同一模块或者同一类型的源文件放在同一文件组中。例如，可在 Project –Manage- Components, Environment, Books...对话框中创建自己的文件组 Syssem Files 来管理 CPU 启动代码和其它系统配置文件等，如图 2-23 所示。可使用 New (Insert)按钮可创建新的文件组，或在 Groups 文件组中选定一个文件组，然后点击 **Add Files** 为其添加文件。

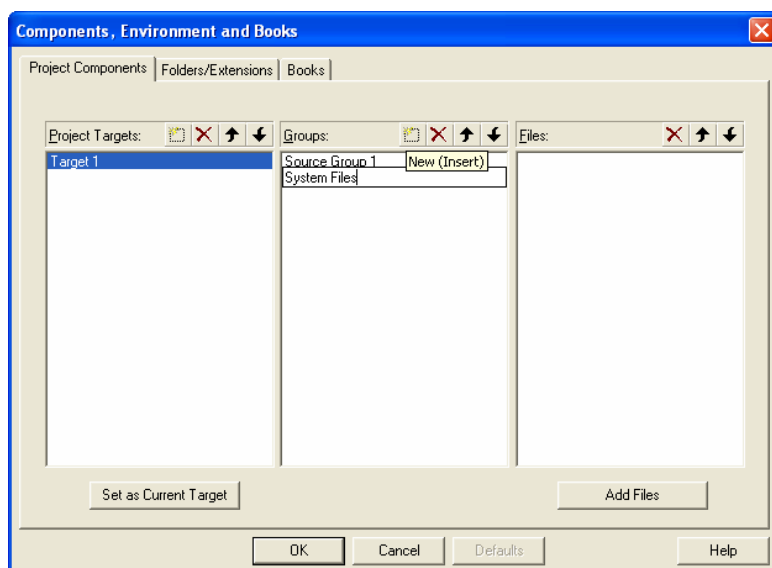



图 2-21 创建新的文件组

5. 设置活动工程

在 μ Vision IDE 中可以存在几个同时打开的工程，但只有一个工程处于活动状态并显示在工程区中，处于活动状态的工程才可以作为调试工程。可在图 2-21 中的工程目标框中选择需要激活的工程，然后单击 `Set as Current Target` 按钮即可。

2.3.3 工程基本配置

1. 硬件选项配置

μ Vision 3 可根据目标硬件的实际情况对工程进行配置。通过单击目标工具栏图标或者单击菜单项 `Project - Options for Target`，在弹出的 `Target` 页面可指定目标硬件和所选择设备片内组件的相关参数，如图 2-22 所示。

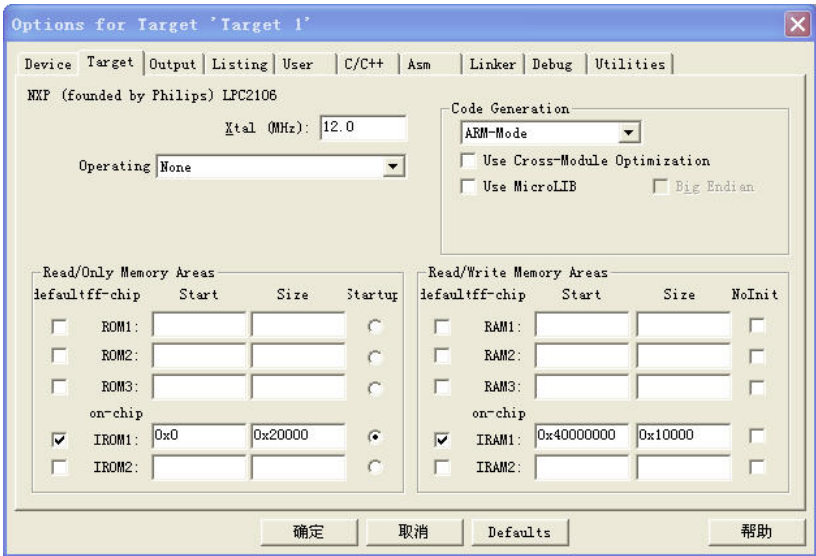


图 2-22 处理器配置对话框

表 2-1 对 `Target` 页面选项作一个简要说明：

表 2-1 目标硬件配置选项说明表

选项	描述
晶振	设备的晶振频率。大部分基于 ARM 的微控制器使用片内 PLL 作为 CPU 时钟源。多数情况下 CPU 时钟和晶振频率是不一致的，依据硬件设备不同设置其相应的值
使用片内 ROM/RAM	定义片内的内存部件的地址空间以供链接器/定位器使用。注意对于一些设备来说需要在启动代码中反映出这些配置
操作系统	允许为目标工程选择一个实时操作系统

2. 处理器启动代码配置

通常情况下，ARM 程序都需要初始化代码用来配置所对应的目标硬件。如 2.3.2 节所述，当创建一个应用程序时 μ Vision 3 会提示使用者自动加入相应设备的启动代码。

μ Vision 3 提供了丰富的启动代码文件，可在相应文件夹中获得。例如，针对 Keil 开发工具的启动代码放在 `..\ARM\Startup` 文件夹下，针对 GNU 开发工具的在 `..\ARM\GNU\Startup` 文件夹下，针对 ADS 开发工具的在文件夹 `..\ARM\ADS\Startup` 下。以 LPC2106 处理器为例，其启动代码文件为 `...\Startup\Philips\Startup.s`，可把这个启动代码文件复制到工程文件夹下。在图 2-16 中双击 `Startup.s`

源文件，根据目标硬件作相应的修改即可使用。 μ Vision 3 里大部分启动代码文件都有一个配置向导（Configuration Wizard），如图 2-23 所示，它提供了一种菜单驱动方式来配置目标板的启动代码。

开发工具提供缺省的启动代码，对于大部分单芯片应用程序来说是一个很好的起点，但是开发者必须根据目标硬件来调整部分启动代码的配置，否则很可能是无法使用的。例如，CPU/PLL 时钟和总线系统往往会根据目标系统的不同而不同，不能够自动地配置。一些设备还提供了片上部件的使能/禁止可选项，这就需要开发者对目标硬件有足够的了解，能够确保启动代码的配置和目标硬件完全匹配。在图 2-25 中的 Text Editor 页面中，提供了标准文本编辑窗口可打开并修改相应的启动代码。

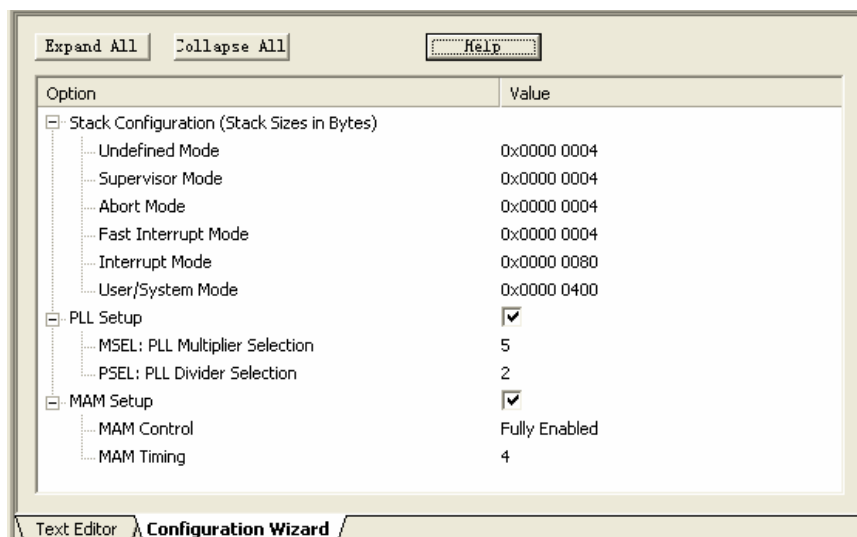


图 2-23 启动代码文件配置向导

3. 仿真器配置

选择菜单项 **Project->Project-Option for Target** 或者直接单击 ，打开 Option for Target 对话框的 Debug 页，弹出如下对话框，进行仿真器的连接配置。



图 2-24 Option for Target 对话框 Debug 页

使用 ULINK 仿真器时，为仿真器选择合适的驱动以及为应用程序和可执行文件下载进行配置，对图 2-24 对话框的设置如图 2-25、图 2-26 所示：

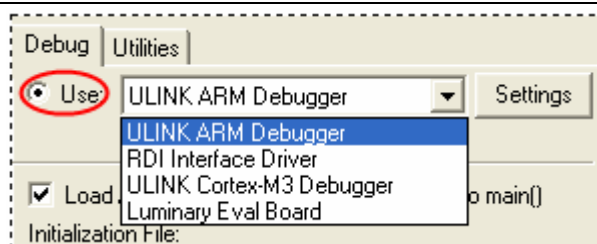


图 2-25 仿真器驱动配置图

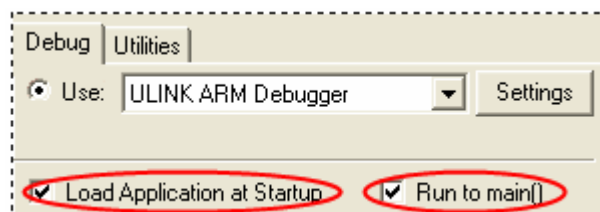


图 2-26 仿真器下载应用程序配置图

PC 机通过 ULINK USB-JTAG 仿真器与目标板连接成功之后，可以打开图 2-26 中的 Settings 选项查看 ULINK 信息，如图 2-28 所示：

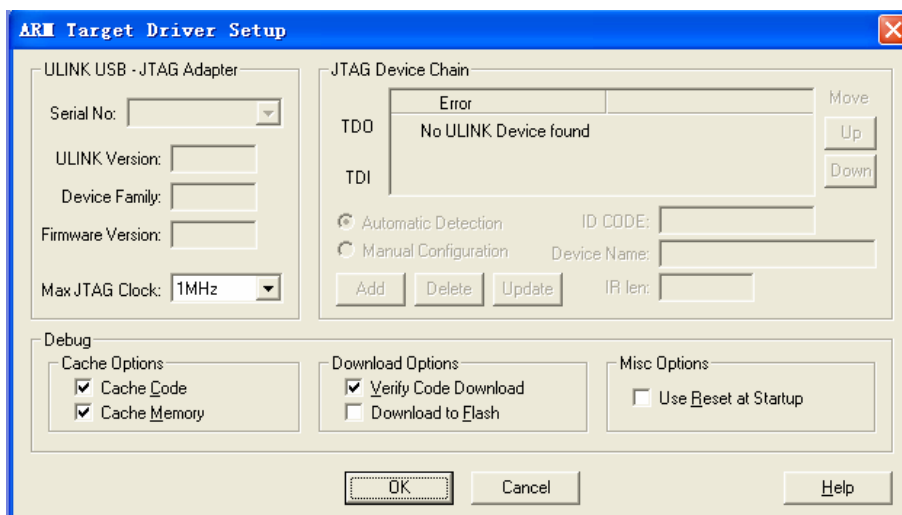


图 2-27 未连 ULINK 仿真器前

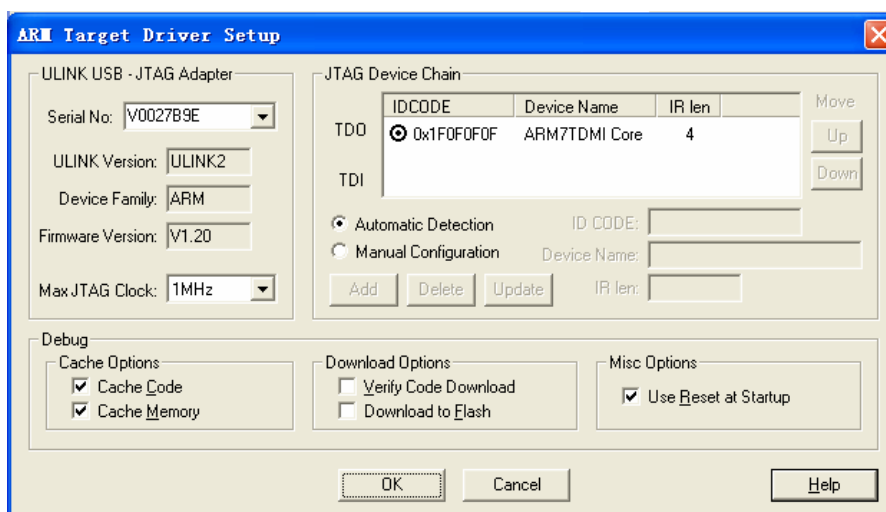


图 2-28 连接 ULINK 仿真器后

4. 工具配置

工具选项主要设置 Flash 下载选项。打开菜单栏的 **Project->Project-Option for Target** 对话框选择其“Utilities”页，或者打开菜单 **Flash->Configure Flash Tools...**，将弹出如图 2-29 所示的对话框。

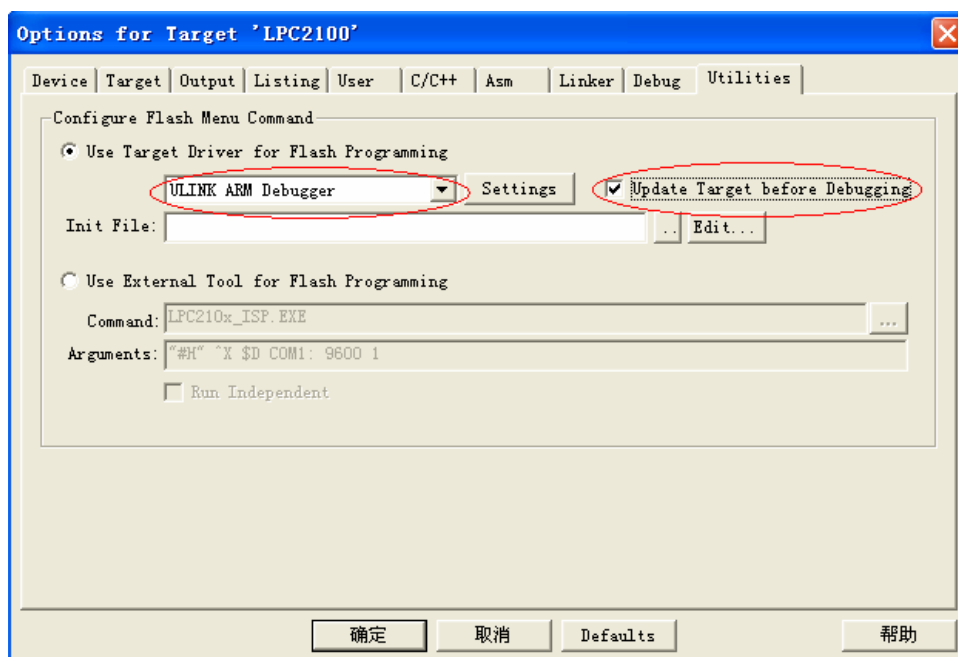


图 2-29 选择 ULINK 下载代码到 FLASH

在图 2-29 对话框中选“Use Target Driver for Flash Programming”，再选择“ULINK ARM Debugger”，同时钩上“Update Target before Debugging”选项。这时还没有完成设置，还需要选择编程算法，点击“Settings”将弹出如图 2-32 所示的对话框。

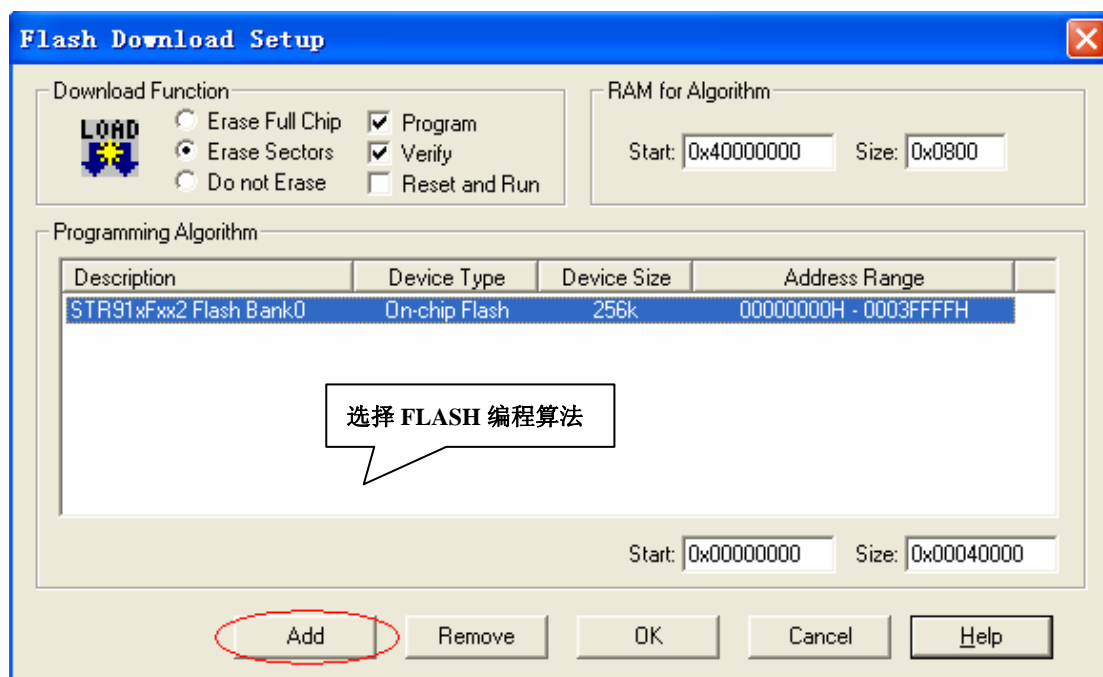


图 2-30 FLASH 下载设置

点击图 2-30 的对话框中的“Add”，将弹出如图 2-31 所示的对话框，在该对话框中选对需要的 FLASH 编程算法。例如对 STR912FW 芯片，由于其 FLASH 为 256kB 则需要选择如下图 2-31 所标注的 FLASH 编程算法。

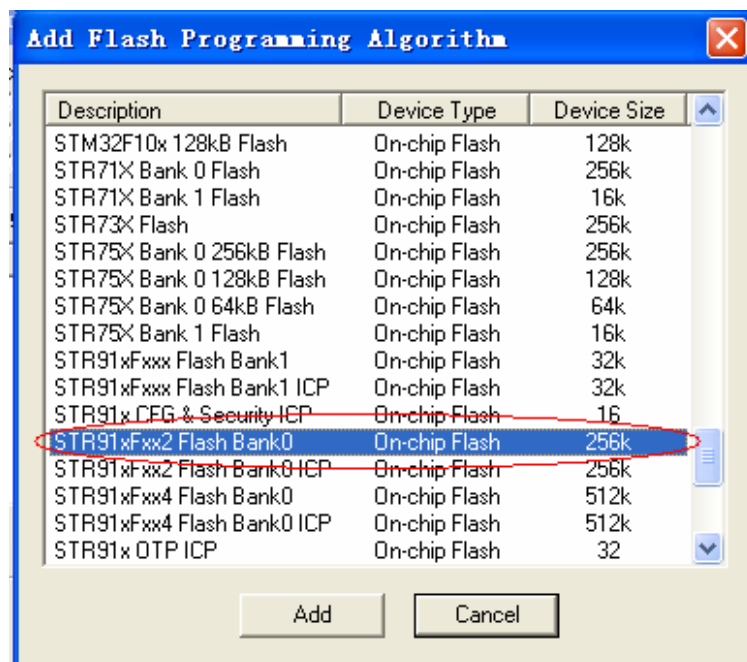


图 2-31 选择 FLASH 编程算法

5. 调试设置

μVision 3 调试器提供了两种调试模式，可以从 Project->Options for Target 对话框的 Debug 页内选择操作模式，如图 2-32 所示。

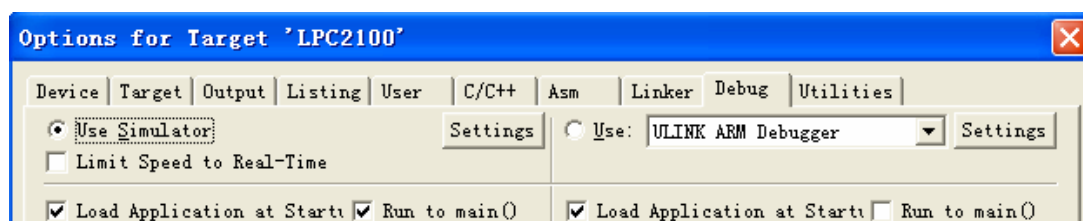



图 2-32 调试器的选择

软件仿真模式：在没有目标硬件情况下，可以使用仿真器（Simulator）将 μVision3 调试器配置为软件仿真器。它可以仿真微控制器的许多特性，还可以仿真许多外围设备包括串口、外部 I/O 口及时钟等。所能仿真的外围设备在为目标程序选择 CPU 时就被选定了。在目标硬件准备好之前，可用这种方式测试和调试嵌入式应用程序。

GDI 驱动模式：使用高级 GDI 驱动设备连接目标硬件来进行调试，例如使用 ULINK Debugger。对 μVision 3 来说，可用于连接的驱动设备有：

- JTAG/OCDS 适配器：它连接到片上调试系统，例如 AMR Embedded ICE。
- Monitor(监视器)：它可以集成在用户硬件上、也可以用在许多评估板上。
- Emulator(仿真器)：它连接到目标硬件的 CPU 引脚上。
- In-System Debugger(系统内调试器)：它是用户应用程序的一部分，可以提供基本的测试功能。
- Test Hardware(测试硬件)：如 Philips SmartMX DBox 、 Infineon SmartCard ROM Monitor RM66P 等。

使用仿真器调试时，选择菜单项 **Project->Project-Option for Target** 或者直接单击 ，打开 **Option for Target** 对话框的 **Debug** 页，弹出如下对话框，可进行调试配置。

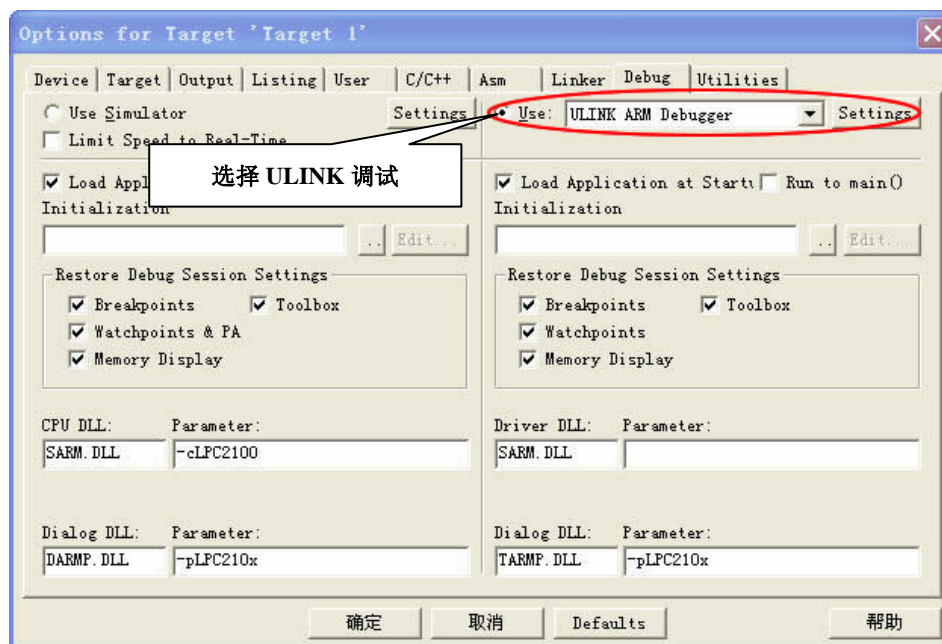


图 2-33 选择 ULINK USB-JTAG 仿真器调试

如果目标板已上电，并且与 ULINK USB-JTAG 仿真器连接上，点击图 2-33 中的“Settings”，将弹出如图 2-34 所示的对话框，正常则可读取目标板芯片 ID 号。如果读不出 ID 号，则需要检查 ULINK USB-JTAG 仿真器与 PC 或目标板的连接是否正确。

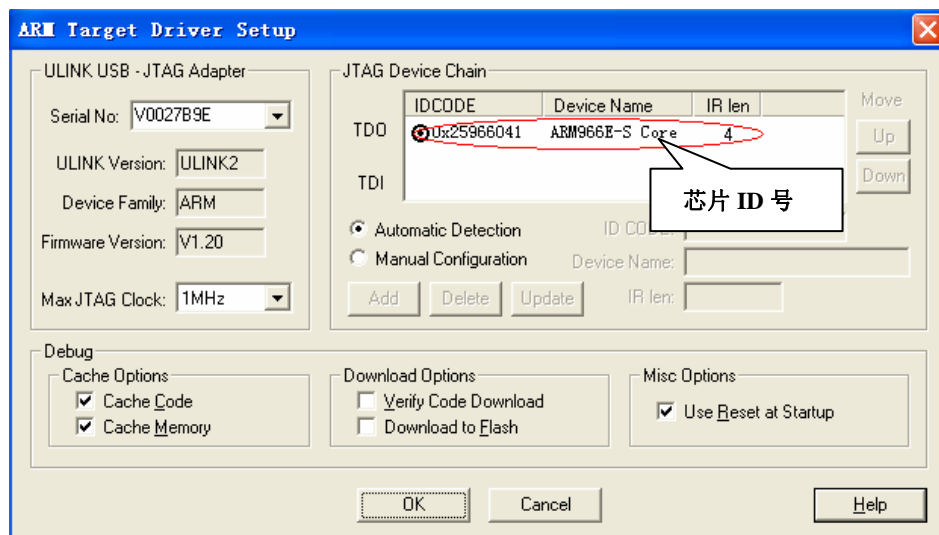


图 2-34 读取设备 ID

6. 编译配置

选择编译器：

μVision IDE 目前支持 RealView、Keil CARM 和 GNU 这三种编译器，从菜单栏的 **Project->Manage->Component,Environment,Books...** 或者直接单击工具栏中的  图标，打开其 **Folder/Extensions** 页进入编译器选择界面。我们使用 RealView 编译器如图 2-35 所示。

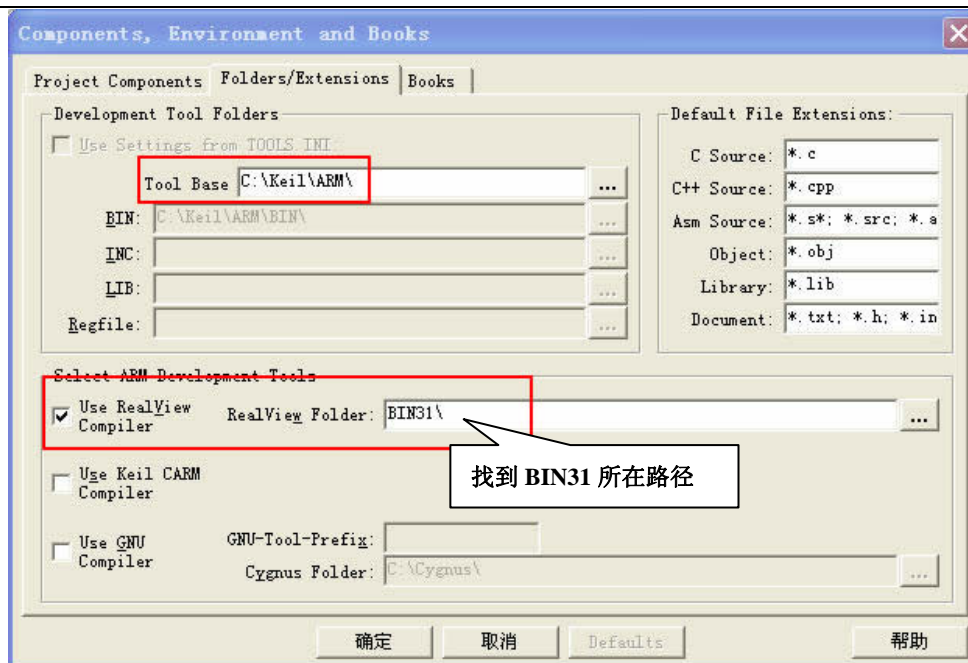



图 2-35 选择编译器

配置编译器：

选择好编译器后，单击图标，打开 Option for Target 对话框的 C/C++ 页，出现如图 2-36 的编译属性配置页面（这里主要说明 RealView 编译器的编译配置）：

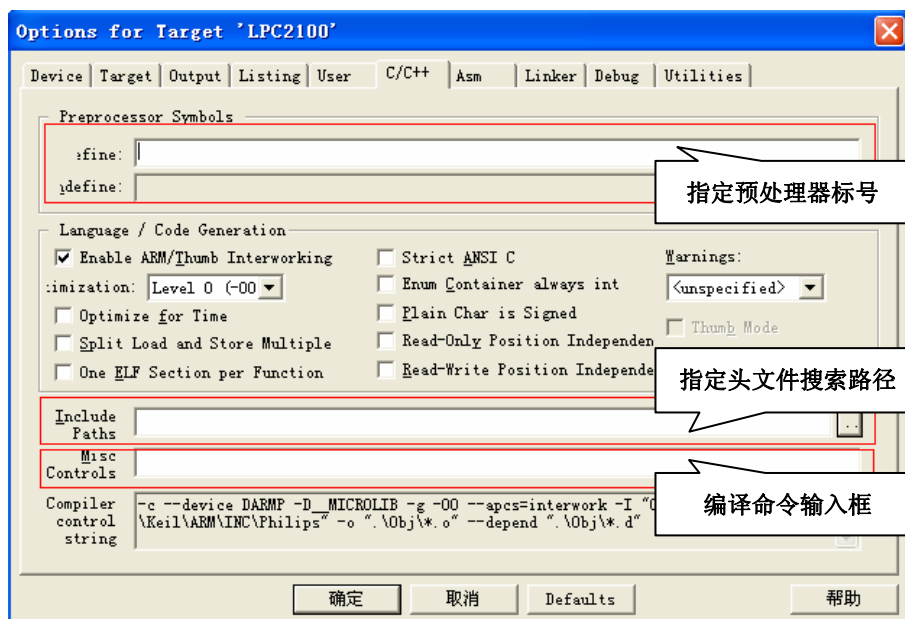


图 2-36 编译器配置页

各个编译选项说明如下：

Enable ARM/Thumb Interworking: 生成 ARM/Thumb 指令集的目标代码，支持两种指令之间的函数调用。

Optimization: 优化等级选项，分四个档次。

Optimize for Time: 时间优化。

Split Load and Store Multiple: 非对齐数据采用多次访问方式。

One ELF Section per Function: 每个函数设置一个 ELF 段。

Strict ANSI C: 编译标准 ANSI C 格式的源文件。

Enum Container always int: 枚举值用整型数表示。

Plain Char is Signed: Plain Char 类型用有符号字符表示。

Read-Only Position Independent: 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: 段中的可读/写的数据地址在运行期间可以改变。

Warning: 编译源文件时, 警告信息输出提示选项。

7. 汇编选项设置

单击图标, 打开 Option for Target 对话框的 Asm 页, 出现如图 2-37 的汇编属性配置页面:

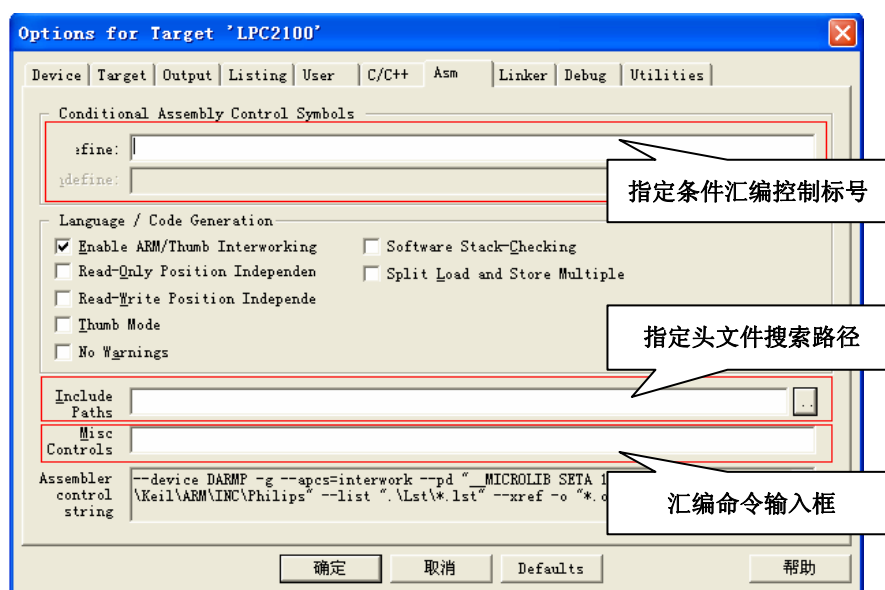


图 2-37 汇编配置页

各个汇编选项说明如下:

Enable ARM/Thumb Interworking: 生成 ARM/Thumb 指令集的目标代码, 支持两种指令之间的函数调用。

Read-Only Position Independent: 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: 段中的可读/写的数据地址在运行期间可以改变。


Thumb Mode: 只编译 THUMB 指令集的汇编源文件。

No Warnings: 不输出警告信息。

Software Stack-Checking: 软件堆栈检查。

Split Load and Store Multiple: 非对齐数据采用多次访问方式。

8. 链接选项设置

链接器/定位器用于将目标模块进行段合并, 并对其定位, 生成程序。既可通过命令行方式使用链接器, 也可在 μ Vision IDE 中使用链接器。单击图标, 打开 Option for Target 对话框的 Linker 页, 出现如图 2-38 的链接属性配置页面:

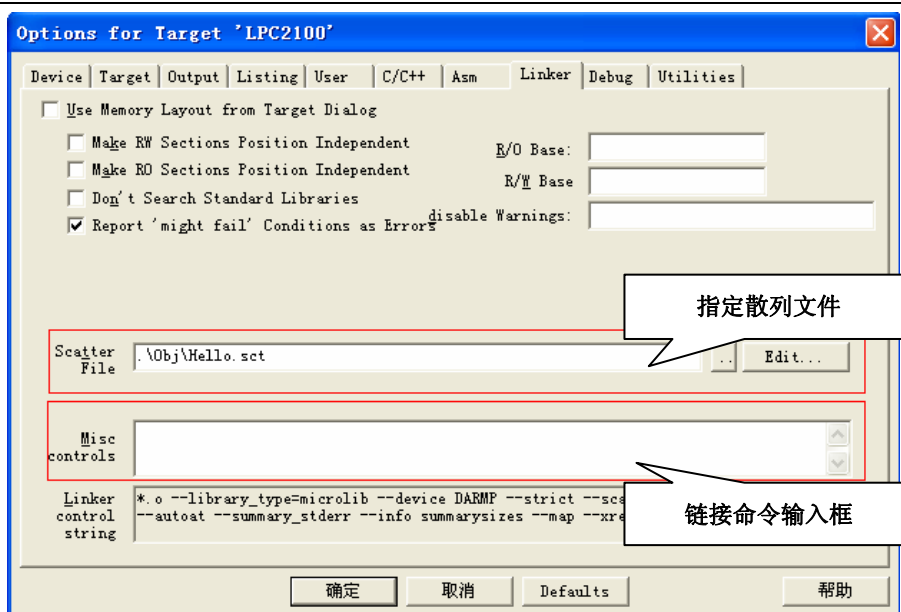


图 2-38 链接配置页

各个链接选项配置说明如下：

Make RW Sections Position Independent: RW 段运行时可改变。

Make RO Sections Position Independent: RO 段运行时可改变。

Don't search Standard Libraries: 链接时不搜索标准库。

Report 'might fail' Conditions as Err: 将'might fail'报告为错误提示输出。

R/O Base: R/O 段起始地址输入框。

R/W Base: R/W 段起始地址输入框。

9. 输出文件设置

在 Project->Option for Target 的 Output 页中配置输出文件，如图 2-39 所示。

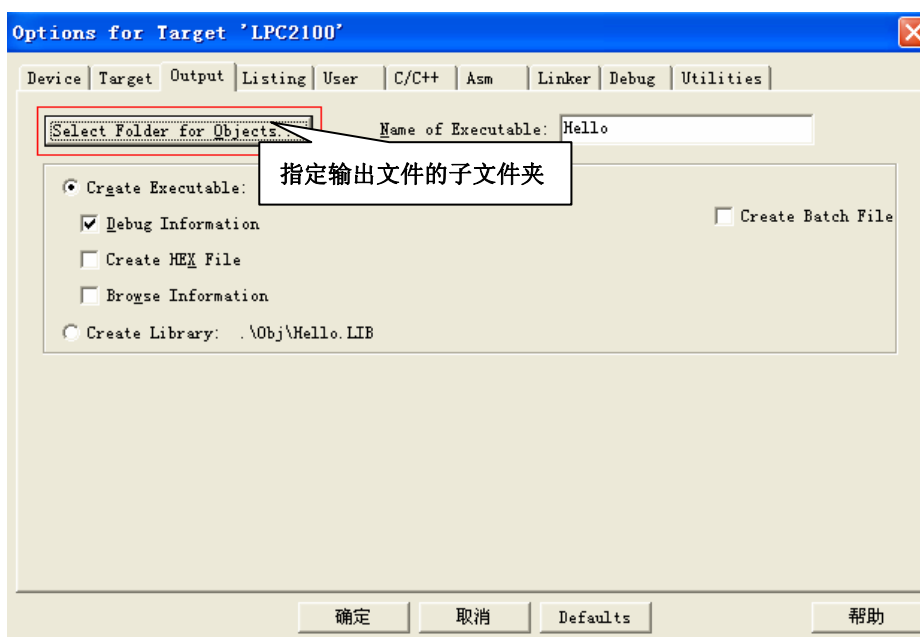


图 2-39 输出文件配置页

输出文件配置选项说明如下

Name of Executable: 指定输出文件名。


Debug Information: 允许时, 在可执行文件内存储符号的调试信息。

Create HEX File: 允许时, 使用外部程序生成一个 HEX 文件进行 Flash 编程。

Big Endian: 输出文件采用大端对齐方式。

Create Batch File: 创建批文件

2.3.4 工程的编译链接

完成工程的设置后, 就可以对工程进行编译链接了。用户可以通过选择主窗口 **Project** 菜单的 **Build target** 项或工具条  按钮, 编译相应的文件或工程, 同时将在输出窗的 **Build** 子窗口中输出有关信息。如果在编译链接过程中, 出现任何错误, 包括源文件语法错误和其它错误时, 编译链接操作立刻终止, 并在输出窗的 **Build** 子窗口中提示错误, 如果是语法错误, 用户可以通过鼠标左键双击错误提示行, 来定位引起错误的源文件行。

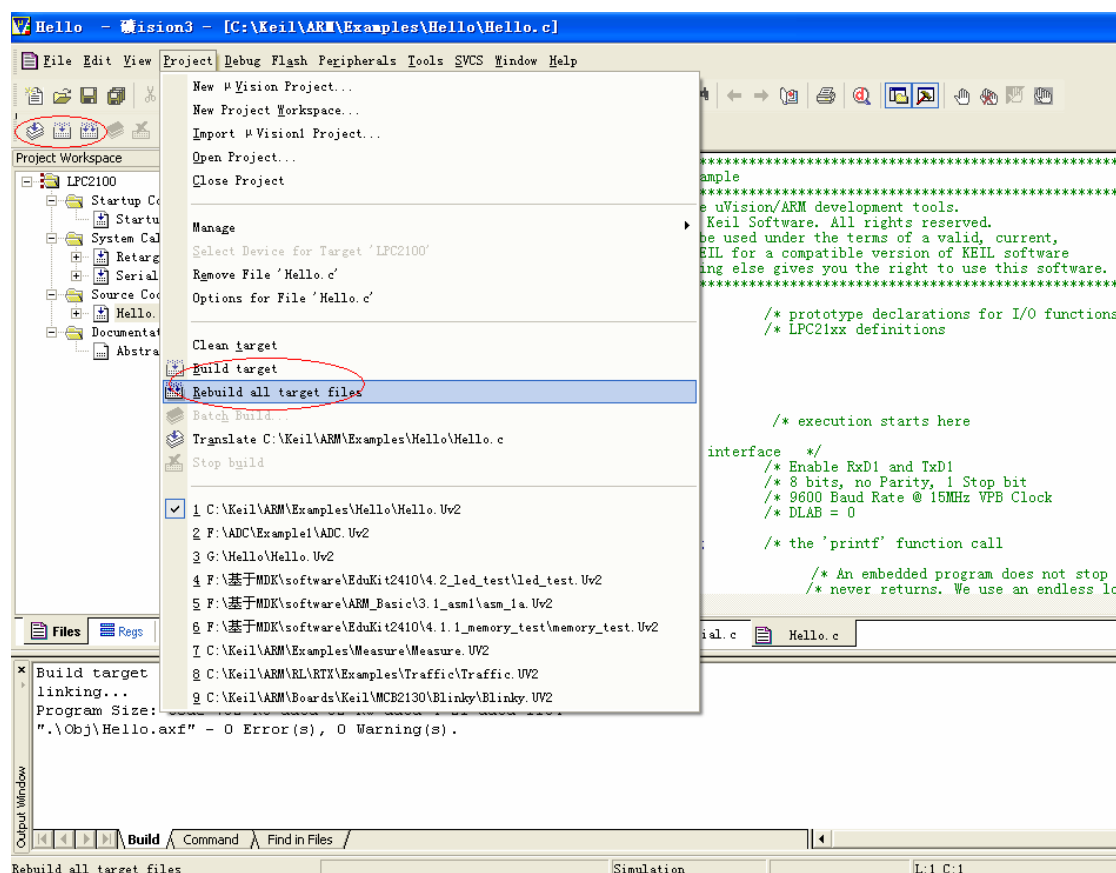


图 2-40 工程 Project 菜单和工具条

2.3.5 加载调试

μVision 3 调试器提供了软件仿真和 GDI 驱动两种调试模式, 采用 ULINK 仿真器调试时, 首先将集成环境与 ULINK 仿真器连接, 按照前面 2.4.3 小节中的工程配置方法对要调试的工程进行配置后, 点击 **Flash->Download** 菜单项可将目标文件下载到目标系统的指定存储区中, 文件下载后即可进行在线仿真调试。

1. 断点和单步

调试器可以控制目标程序的运行和停止, 并反汇编正在调试的二进制代码, 同时可通过设置断点来控制程序的运行, 辅助用户更快的调试目标程序。μVision IDE 的调试器可以在源程序、反汇编程

序、以及源程序汇编程序混合模式窗口中设置和删除断点。在 μ Vision3 中设置断点的方式非常灵活，甚至可以在程序代码被编译前

在源程序中设置断点。定义和修改断点的方式有如下几种：

使用文件工具栏，只要在编辑窗口或反汇编窗口中选中要插入断点的行，然后再单击工具栏上的按钮就可以定义或修改断点；

使用快捷菜单上的断点命令，在编辑窗口或反汇编窗口中单击右键即可打开快捷菜单；

在 **Debug-Breakpoints...**对话框中，可以查看、定义、修改断点。这个对话框可以定义及访问不同属性的断点；

在 **Output Window-Command** 页中，使用 **BreakSet**、**BreakKill**、**BreakList**、**BreakEnable**、**BreakDisable** 命令对断点进行管理。

在断点对话框中可以查看及修改断点，如图 2-41 所示。可以在 **Current Breakpoints** 列表中通过单击复选框来快捷地 **Disable** 或 **Enable** 一个断点。在 **Current Breakpoints** 列表中双击可以修改选定的断点。

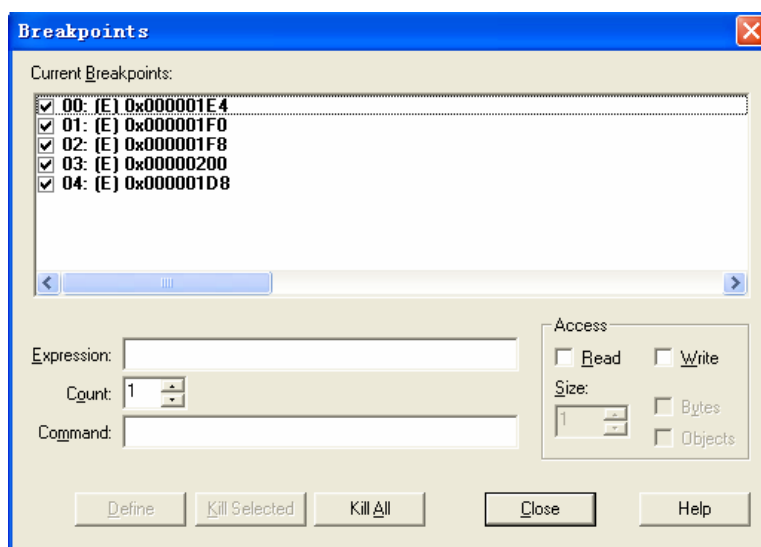


图 2-41 断点对话框

如图 2-41 所示，可以在断点对话框表达式的文本框中输入一个表达式来定义断点。根据表达式的类型可以定义如下类型的断点：

当表达式是代码地址时，一个类型为 **Execution Break(E)**的断点被定义，当执行到指定的代码地址时，此断点有效。输入的代码地址要参考每条 CPU 指令的第一个字节；

当对话框中一个内存访问类型（可读、可写或既可读又可写）被选中时，那么将会定义一个类型为 **Access Break(A)**的断点。当指定的内存访问发生时，此断点有效。可以字节方式指定内存访问的范围，也可以指定表达式的目标范围。**Access Break** 类型的表达式必须能转化为内存地址及内存类型。在 **Access Break** 类型的断点停止程序执行或执行命令之前操作符(&, &&, <, <=, >, >=, ==, and !=)可用于比较变量的值；

当表达式不能转换为内存地址时，一个 **Conditional Break(C)**类型的断点将被定义，当指定的条件表达式为真时，此断点有效。在每个 CPU 指令后，均需要重新计算表达式的值，因此，程序执行速度会明显降低。

在 **command** 文本框中可以为断点指定一条命令，程序执行到断点时将执行该命令， μ Vision 3 执行命令后会继续执行目标程序。在此指定的命令可以是 μ Vision 3 的调试命令或信号函数。 μ Vision

n3 中, 可以使用系统变量 `_break_` 来停止程序的执行。`Count` 的值用于指定断点触发前断点表达式为真的次数。

2. 反汇编窗

反汇编窗用于显示反汇编二进制代码后得到的汇编级代码, 可以混合源代码显示, 也可以混合二进制代码显示。反汇编窗可以设置和清除汇编级别断点, 并可按照 ARM 或 THUMB 格式的反汇编二进制代码。

如图 2-42 所示, 反汇编窗口可用于将源程序和反汇编程序一起显示也可以只显示反汇编程序。通过 **Debug -> View Trace Records** 可以查看前面指令的执行记录。为了实现这一功能, 需要设置 **Debug -> Enable/Disable Trace Recording**。

若选择反汇编窗口作为当前窗口, 那么程序的执行是以 CPU 指令为单位的而不是以源程序中的行为单位的。可以用工具条上的按钮或快捷菜单命令为选中的行设置断点或对断点进行修改。还可以使用对话框 **Debug -> Inline Assembly...** 来修改 CPU 指令, 它允许设计这对调试的目标程序进行临时修改。

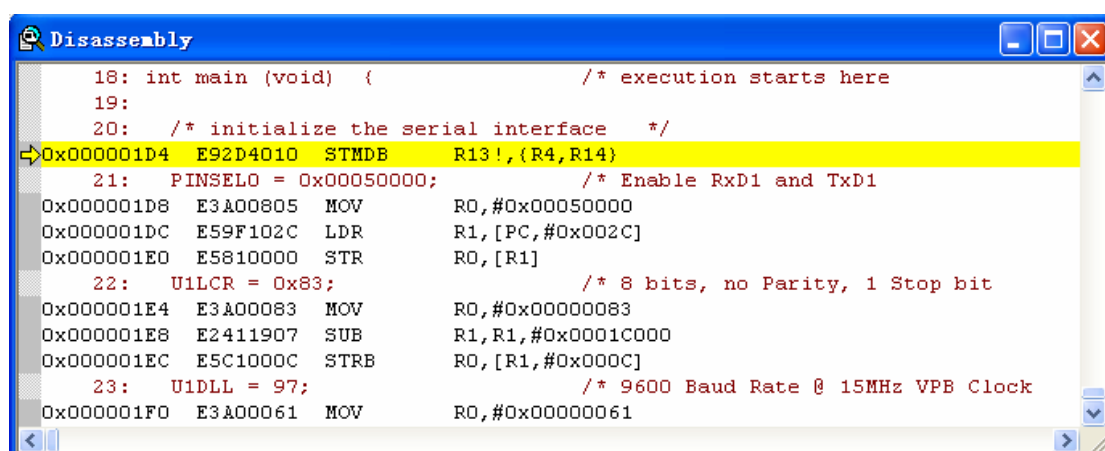


图 2-42 源文件与反汇编指令交叉显示窗口

3. 寄存器窗

在 **Project Workspace - Regs** 页中列出了 CPU 的所有寄存器, 按模式排列共有八组, 分别为 **Current** 模式寄存器组、**User/System** 模式寄存器组、**Fast Interrupt** 模式寄存器组、**Interrupt** 模式寄存器组、**Supervisor** 模式寄存器组、**Abort** 模式寄存器组、**Undefined** 模式寄存器组以及 **Internal** 模式寄存器组, 如图 2-43 所示。在每个寄存器组中又分别有相应的寄存器。在调试过程中, 值发生变化的寄存器将会以蓝色显示。选中指定寄存器单击或按 **F2** 键便可以出现一个编辑框, 从而可以改变此寄存器的值。

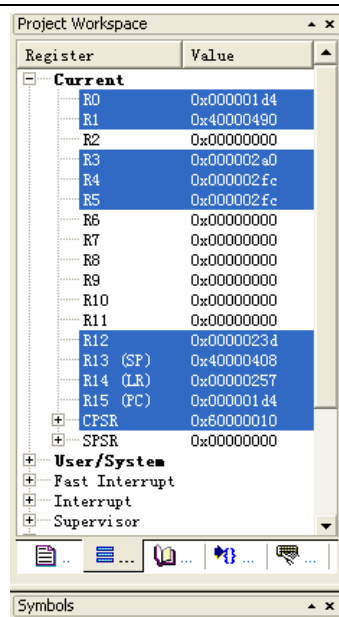


图 2-43 Regs 页

4. 存储区窗

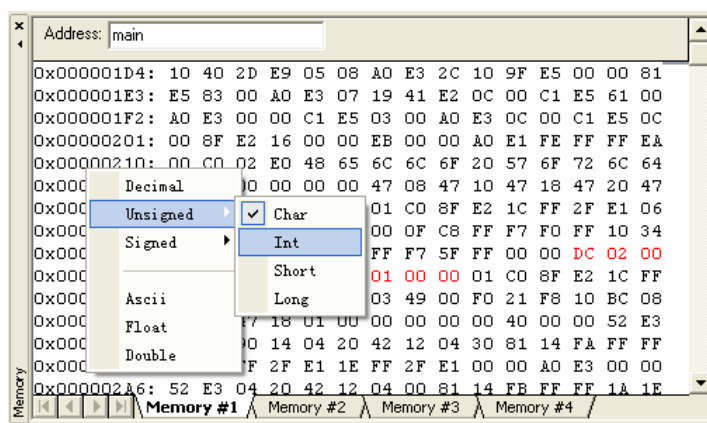


图 2-44 内存窗口

通过内存窗口可以查看与显示存储情况，View-Memory Window 可以打开存储器窗口，如图 2-44 所示。 μ Vision 3 可仿真了高达 4GB 的存储空间，这些空间可以通过 MAP 命令或 Debug - Memory Map 打开内存映射对话框来映射为可读的、可写的、可执行的，如图 2-45。 μ Vision 3 能够检查并报告非法的存储访问。

从图 2-44 中可看出内存窗口有四个 Memory 页，分别为 Memory#1、Memory#2、Memory#3、Memory#4，即可同时显示四个指定存储区域的内容。在 Address 域内，输入地址即可显示相应地址中的内容。需要说明的是，它支持表达式输入，只要这个表达式代表了某个区域的地址即可，例如图 2-44 中所示的 main。双击指定地址处会出现编辑框，可以改变相应地址处的值。在存储区内单击右键可以打开如图 2-44 所示的快捷菜单，在此可以选择输出格式。通过选中 View -> Periodic Window Update，可以在运行时实时更新此内存窗口中的值。在运行过程中，若某些地址处的值发生变化，将会以红色显示。

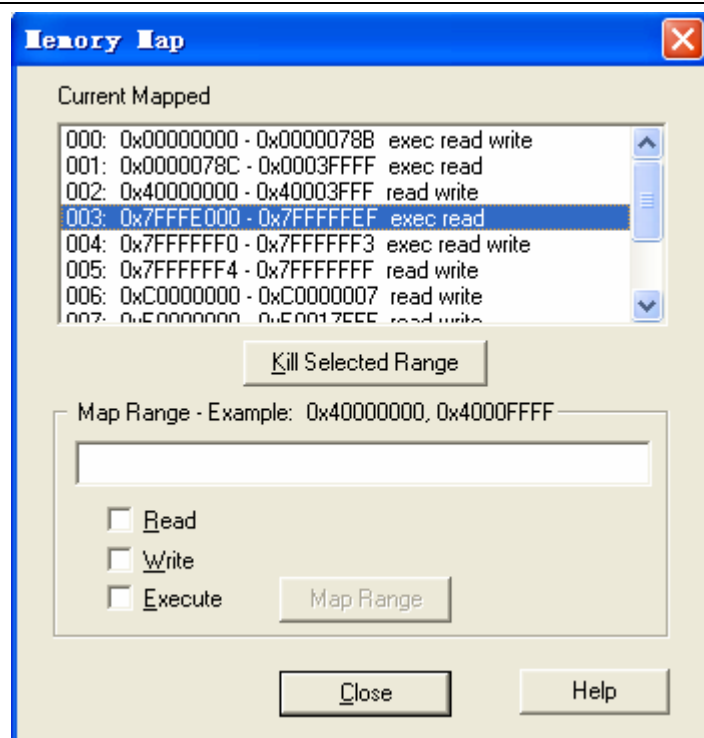


图 2-45 内存映射对话框

内存映射对话框可以用来设定哪些地址空间用于存储数据、哪些地址空间用于存储程序。也可以用 MAP 命令来完成上述工作。在载入目标应用时，μVision 3 自动地对应用进行地址映射，一般不需要映射额外的地址空间，但被访问的地址空间没有被明确声明时必须进行地址映射，如存储映射 I/O 空间。如上图所示，每一个存储空间均可指定为可读、可写、可执行，若在编辑框内输入“MAP 0X0C000000,0X0E000000 READ WRITE EXEC”，此命令就是将从 0X0C000000 到 0X0E000000 这部分区域映射为可读的、可写的、可执行的。在目标程序运行期间，μVision 3 使用存储映射来保证程序没有访问非法的存储区。

5. 观测窗口

观测窗口（Watch Windows）用于查看和修改程序中变量的值，并列出了当前的函数调用关系。在程序运行结束之后，观测窗口中的内容将自动更新。也可通过菜单 View - Periodic Window Update 设置来实现程序运行时实时更新变量的值。观测窗口共包含四个页：Locals 页、Watch #1 页、Watch #2 页、Call Stack 页，分别介绍如下。

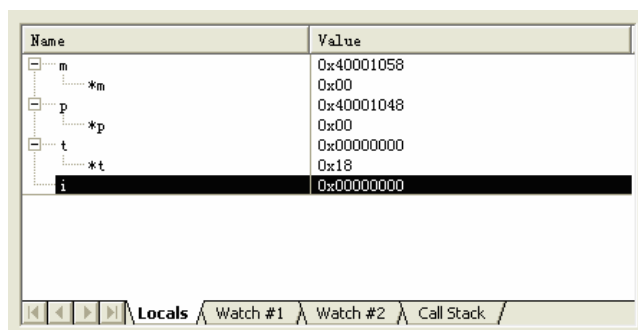


图 2-46 Watch 窗口之 Locals 页

Locals 页： 如图 2-46 所示，此页列出了程序中当前函数中全部的局部变量。要修改某个变量的值，只需选中变量的值，然后单击或按 F2 即可弹出一个文本框来修改该变量的值。

Watch 页：如图 2-47 所示，观测窗口有 2 个 Watch 页，此页列出了用户指定的程序变量。有三种方式可以把程序变量加到 Watch 页中：

- 在 Watch 页中，选中<type F2 to edit>，然后按 F2，会出现一个文本框，在此输入要添加的变量名即可，用同样的方法，可以修改已存在的变量；
- 在工作空间中，选中要添加到 Watch 页中的变量，右击会出现快捷菜单，在快捷菜单中选择 Add to Watch Window，即可把选定的变量添加到 Watch 页中；
- 在 Output Window 窗口的 Command 页中，用 WS(WatchSet) 命令将所要添加的变量添入 Watch 页中。

若要修改某个变量的值，只需选中变量的值，再单击或按 F2 即可出现一个文本框修改该变量的值。若要删除变量，只需选中变量，按 Delete 键或在 Output Window 窗口的 Command 页中用 WK(WatchKill) 命令就可以删除变量。

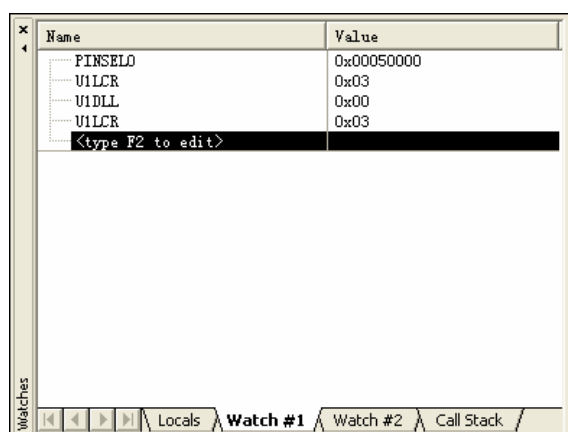


图 2-47 Watch 窗口之 Watch 页

Call Stack：如图 2-48 所示，此页显示了函数的调用关系。双击此页中的某行，将会在工作区中显示该行对应的调用函数以及相应的运行地址。

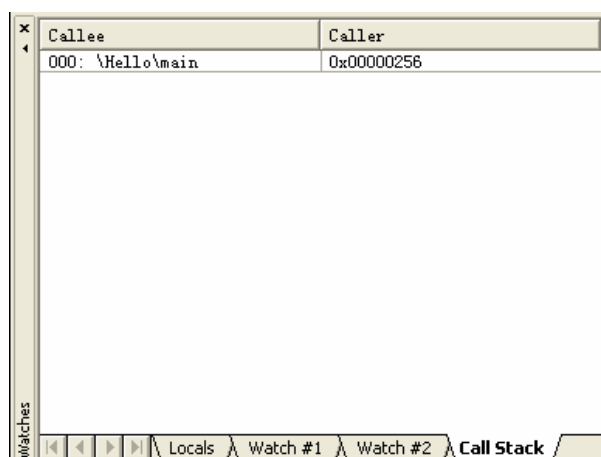


图 2-48 Watch 窗口之 Call Stack 页

6. 代码统计对话框

μVision 3 提供了一个统计代码 (Code Coverage) 执行情况的功能, 这个功能以代码统计对话框的形式表示出来, 如图 2-51 所示。在调试窗口中, 已执行的代码行在左侧以绿色标出。当测试嵌入式应用程序时, 可以用此功能来查看那些程序还没有被执行。

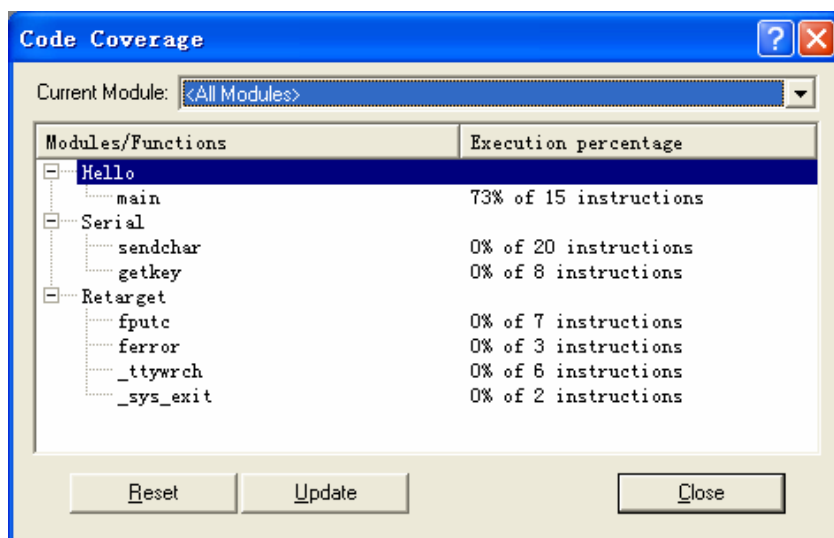


图 2-49 代码统计对话框

如图 2-51 所示代码统计对话框提供了程序中各个模块及函数的执行情况。在 **Current Module** 下拉列表框中列出了程序所有要模块, 而在下面的则显示了相应模块中指令的执行情况, 即每个模块或函数的指令执行百分比, 只要是执行了的部分均以绿色标出。在 **Output Window - Command** 页中可以用 **COVERAGE** 调试命令将此信息输出到输出窗口中。

7. 执行剖析器

μVision ARM 仿真器包含一个执行剖析器, 它可以记录执行全部程序代码所需的时间。可以通过选中 **Debug - Execution Profiling** 来使能此功能。它具有两种显示方式: **Call** (显示执行次数) 和 **Time** (显示执行时间)。将鼠标放在指定的入口处, 即可显示有关执行时间及次数的详细信息。如图 2-50 所示。

对 C 源文件, 可能使用编辑器的源文件的大纲视图特性, 用此特性可以将几行源文件代码收缩为一行, 以此可查看某源文件块的执行时间。

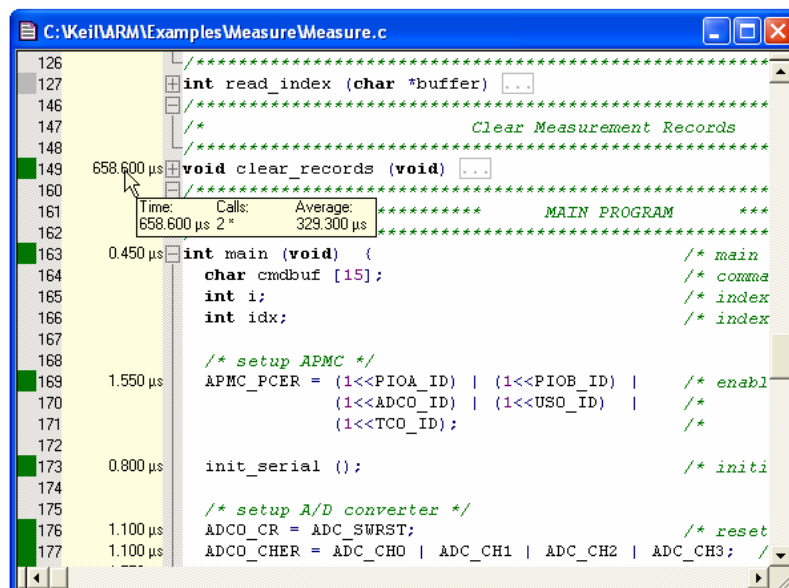


图 2-50 执行剖析器

在反汇编窗口中，可以显示每条汇编指令的执行信息，如图 2-51 所示。

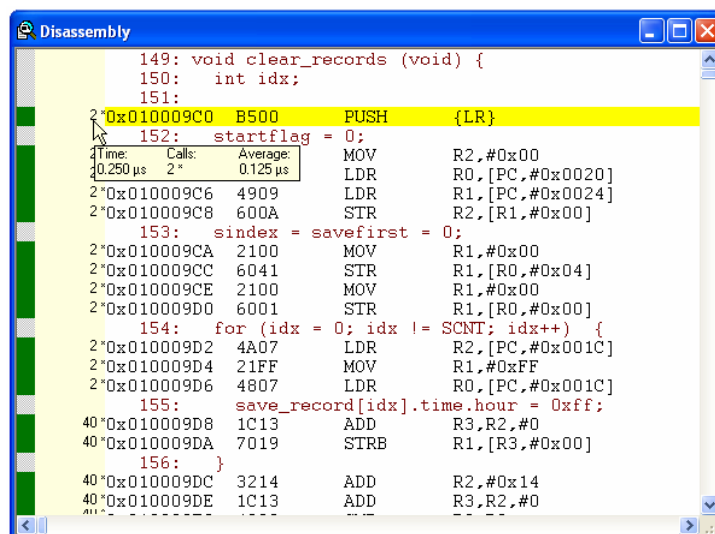


图 2-51 反汇编窗口

需要注意：执行剖析器得到的执行时间是基于当前的时钟设置，当代码以不同的时钟执行多次时，可能会得到一个错误的执行时间。另外，目前执行剖析器仅能用于 ARM 仿真器。

8. 性能分析仪

μVision 3 ARM 仿真器的执行剖析器能够显示已知地址区域的执行统计的信息。对没有调试信息的地址区域，显示列表中是不会显示这块区域的执行情况，例如 ARM ADS/RealView 工具集的浮点库。

μVision 3 性能分析仪则可用于显示整个模块的执行时间及各个模块被调用的次数。μVision 3 的仿真器可以记录整个程序代码的执行时间及函数调用情况，如图 2-52 所示。

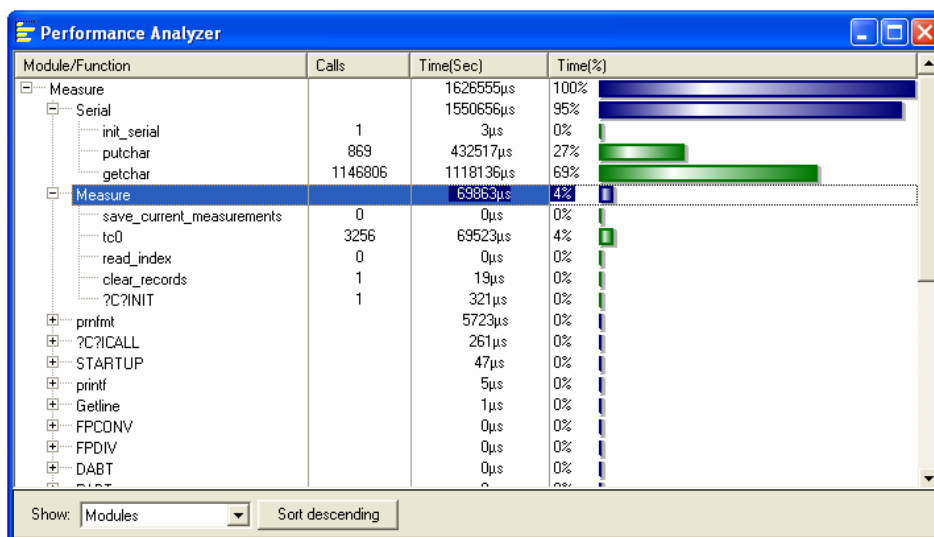


图 2-52 性能分析仪

图 2-52 中的 Show 下拉框用于选择以模块或函数的形式进行显示。Sort descending 按钮则用于以降序来排列各模块或函数的执行时间。表头各项含义分别为：Module/Funcation 是模块或函数名；Calls 是函数的调用次数；Time(Sec)是花费在函数或模块区域内的执行时间；Time (%) 是花费在函数或模块区域内的时间百分比。

9. 串行窗口

μVision 3 提供了两个串行窗口用于串行输入及输出，如图 2-53 所示。从被仿真的处理器所输出的数据会在此窗口中显示，在此窗口中输入的字符也会被输入到被仿真的 CPU 中。

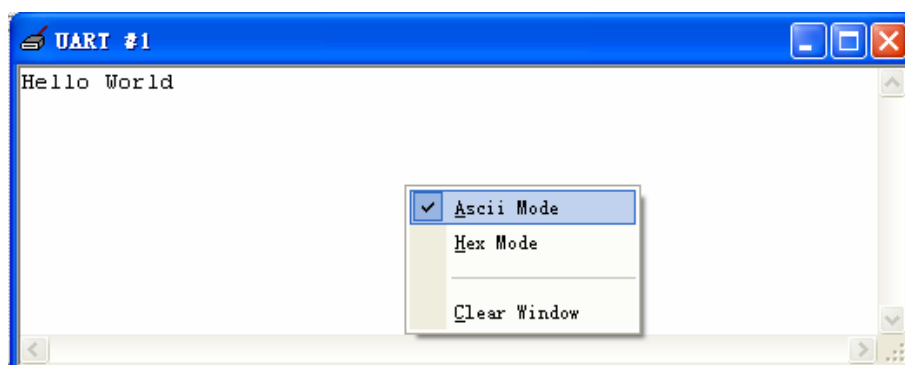


图 2-53 串行窗口

利用串行窗口，在不需要外部硬件的情况下也可以仿真 CPU 的 UART。在 Output Window - Command 页中使用 ASSIGN 命令也可以将串口输出指定为 PC 的 COM 口。

10. 工具箱

如图 2-54 所示，工具箱中包含用户可配置的按钮，单击工具箱上的按钮可以执行相关的调试命令或调试函数。工具箱按钮可以在任何时间执行，甚至是运行测试程序时。

在 Output Window-Command 页中用 DEFINE BUTTON 命令可定义工具箱按钮，语法格式：

```
>DEFINE BUTTON "button_label", "command"
```

其中，button_label 是显示在工具箱按钮上的名字；command 是按下此按钮被时要执行的命令。

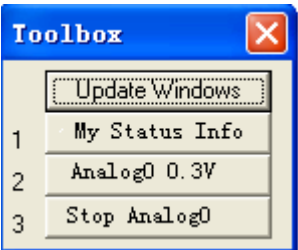


图 2-54 工具箱

下面的例子说明了如何使用命令来定义图 2-54 所示的工具箱按钮：

```
>DEFINE BUTTON "Decimal Output", "radix=0x0A">DEFINE BUTTON "Hex Output", "radix=0x10"

>DEFINE BUTTON "My Status Info", "MyStatus ()" /* call debug function */
>DEFINE BUTTON "Analog0..5V", "analog0 ()" /* call signal function */
>DEFINE BUTTON "Show R15", "printf ("\R15=%04XH\n\n")"
```

11. 输出窗口调试命令对话框

通过在 Output Window – Command 页中键入命令，可以交互的方面使用 μVision 3 调试器。此窗口还能提供一般的调试输出信息，并允许键入用于查看或修改变量和寄存器的表达式（expressions），也可用于调用调试函数（debug functions）。图 2-55 为输出窗口命令对话框。

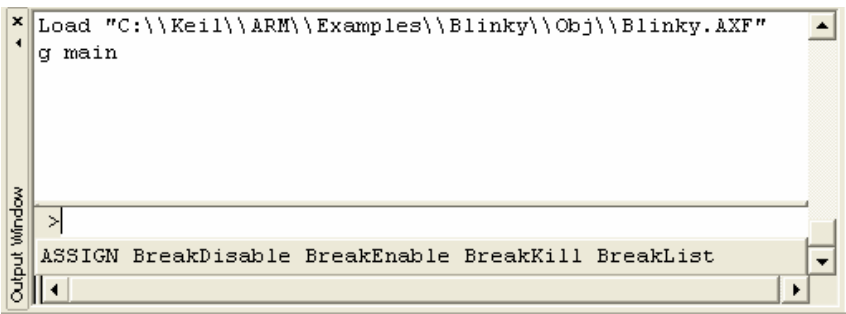


图 2-55 输出窗口命令对话框

调试命令 在调试窗口的“>”提示符后可以输入调试命令，仅用首字母来键入命令，例如 WatchSet 命令仅需要键入 WS。

还可以在命令窗口中显示和改变变量，寄存器和存储位置，例如，可以在命令提示符中输入如表 2-2 所示的文本命令。

表 2-2 利用命令修改变量及寄存器

命令	结果
R7 = 12	为寄存器 R7 分配值 12.
CPSR	显示寄存器 CPSR 的值.
time.hour	显示时间结构体的成员：小时.

<code>time.hour++</code>	时间结构体的成员小时递增.
<code>index = 0</code>	为 index 分配值 0.

调试函数

还可以在命令提示符处输入调试函数来进行程序调试，例如：

ListInfo (2)

在命令键入处，有语法生成器可以帮助显示命令、选项以及参数。随着命令的键入，μVision 3 会自动减少所列出的命令以与所键入的字符相匹配。例如图 2-56 所示。

若键入 B,语法生成器会减少所列出的命令。

>B

BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess

若命令确定的话可用的命令选项会被列出。

>BS|

BreakSet

Build Command Find in Files

语法生成器指导进入命令并避免错误。

>BS WRITE saverecords[0],1,"_break_=1"

"<command>" <cr>

Build Command Find in Files

图 2-56 命令输入的语法提示

12. 符号窗

在符号窗口 View - Symbol Window 中显示了定义在当前被载入的应用程序中的公有符号、局部符号及行号信息。CPU 特殊功能寄存器 SFR 符号也显示在此窗口中，如图 2-57 所示。

图 2-57 符号窗口

可以选择符号类型并用符号窗口中的选项过滤信息，如表 2-3 所示。

表 2-3 符号窗口各选项含义

选项	描述
Mode	选择 PUBLIC、LOCALS 或 LINE。公有 PUBLIC 符号的作用域是整个应用程序；局部 LOCALS 函数的作用被限制在一个模块或函数中；行 LINE 是与源文本中的行号信息相关的。
Current Module	选择其信息应该被显示的源模块。
Mask	指定一个通配字符串以用于匹配符号名。通配字符串由文字数字符及通配字符组成： <div><div>#</div><div>匹配一位数字 (0 – 9)</div></div>

- 51 -

	\$	匹配任意字符
	*	匹配 0 个或多个字符
Apply	应用 mask，并显示更新的符号列表	

表 2-4 通配字符用法

通配字符	匹配符号名 ...
*	匹配任意符号。这是符号浏览器中的默认掩码。
#	匹配在任意位置处包含一个数字位的符号
_a\$#*	以一个下划线开始，后面是个字母 a，再后面是任意个字符，再后面是一个数字位，以 0 个或多个字符结束，例如 _ab1 or _a10value.
_*ABC	以一个下划线开始，后面是 0 个或多个字符，以 ABC 结束。

2.3.6 Flash 编程工具

μVision 3 集成了 Flash 编程工具，所有的相关配置将被保存在当前工程中。在菜单项 Project-Options-Utilities 下可配置当前工程所使用的 Flash 编程工具，开发人员既可使用外部的命令行驱动工具（通常由芯片销售商提供），也可使用 Keil ULINK USB-JTAG 适配器等工具。

用户可通过 Flash 菜单启动 Flash 编程器，若设置了 Project-Options-Utilities-Update Target before Debugging，那么在调试器启动之前 Falsh 编程器也将启动。

μVision 3 为 Flash 编程工具提供了一个命令接口，在 Project-Option for Target 对话框的 Utilities 页中可配置 Flash 编程器，通过菜单项 Flash-Configure Flash Tools 也可进入此对话框，如图 2-58 所示。一旦配置好了命令接口方式，就可以通过 Flash 菜单下载(Download)或擦除(Erase)目标板中 Flash 存储器的内容。

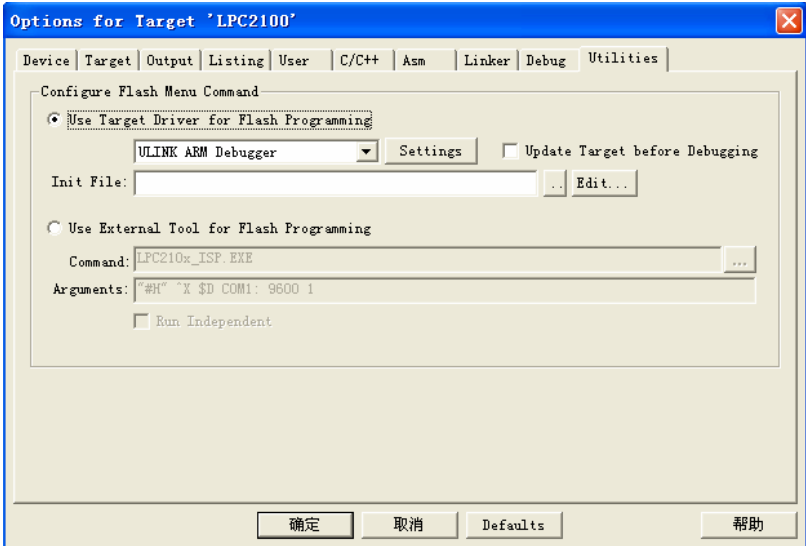


图 2-58 Flash 编程器的配置对话框

μVision 3 提供了两种 Flash 编程的方法：目标板驱动和外部工具。

- 目标板驱动:

μVision 3 提供了 3 种 Flash 编程驱动，ULINK ARM Debugger、ULINK Cortex-M3 Debugger 及 RDI Interface Driver。选择一个 Flash 编程驱动程序，单击右边的 Settings 按钮，弹出图 2-59 所示的对话框（根据选择的驱动程序不同，弹出的对话框略有差异）。最右边复选框决定是否在调试前更新目标板中 Flash 的内容。Init File 文本框中的初始化文件，包括总线的配置、附加程序的下载及发或调试函数。对大多数 Flash 芯片而言，μVision 3 的设备数据库已经提供了片上 Flash Rom 的正确配置，例如图 2-59 所示的 LPC2104 Flash。

- 外部工具：

使用第三方的基于命令行的 Flash 编程工具。通过命令行及参数调用下载工具，如图 2-60 中使用的是 LPC210x_ISP.exe 在线编程器。使用键码序列(Key Sequences)指定输出文件名、设备名及 Flash 编程器的时钟频率等。设置 Run Independent 复选框决定编程工具是否能独立运行。

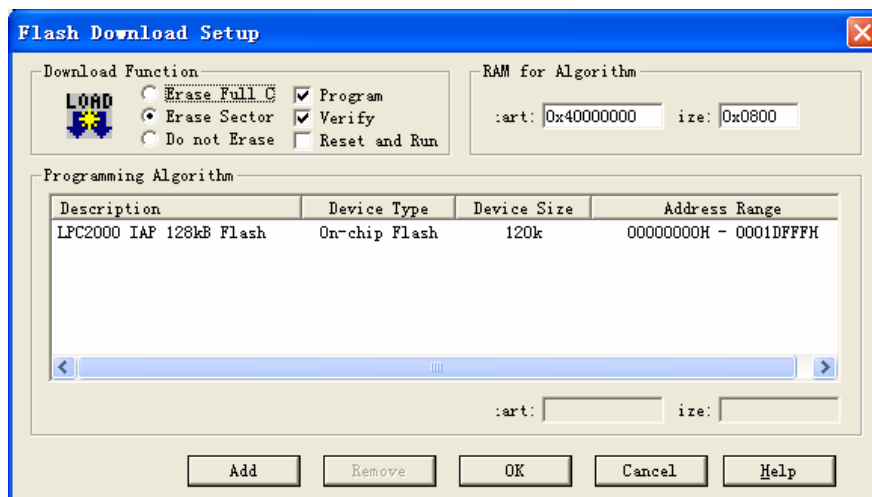


图 2-59 ULINK ARM Debugger 的 Flash 下载设置对话框

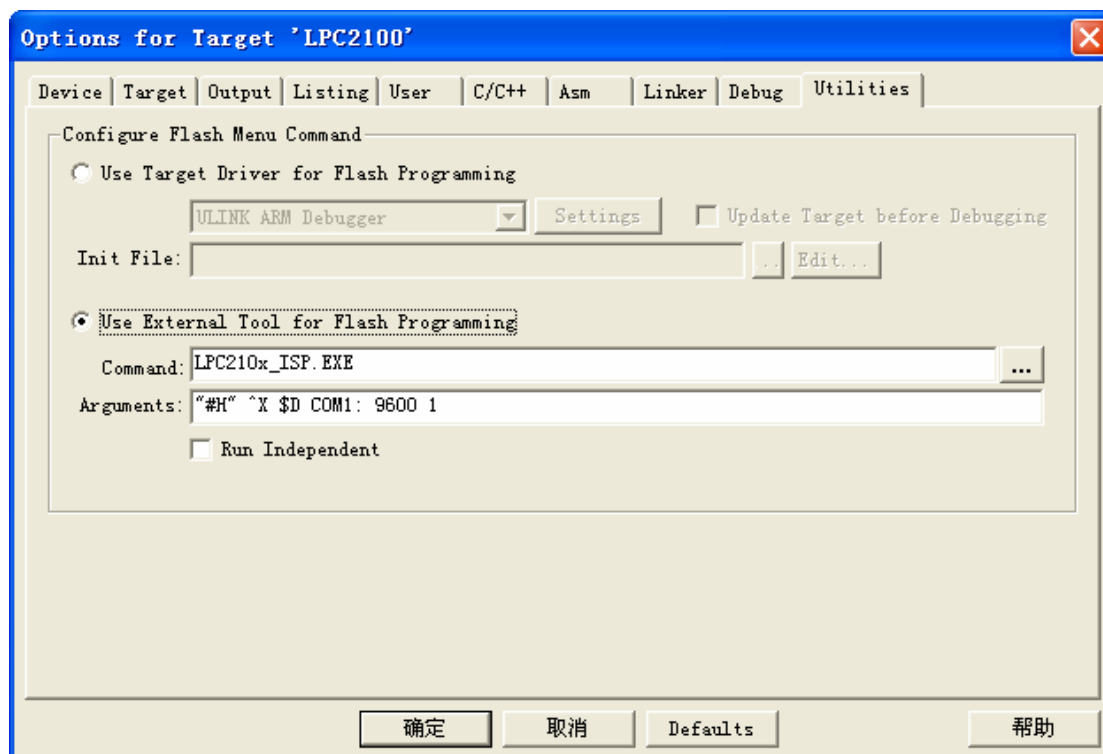


图 2-60 使用外部 Flash 编程工具

第三章 嵌入式软件开发基础实验

本章将介绍几个简单的基础开发实例，既能帮助读者掌握 MDK 嵌入式软件的基本开发过程，同时也能让读者了解 ARM 处理器的基本结构、指令集、存储系统以及基本接口编程。

本书所有的实例均在 Embest EduKit-III（选用 S3C2410 处理器子板）教学实验平台上运行通过。Embest EduKit-III 是一款功能强大的 32 位的嵌入式开发板，可任意选用以下处理器子板：ARM7 的三星的 S3C44B0X、ARM9 的三星的 S3C2410A 芯片、ARM10 的 Intel 的 XScale 芯片和 ADI 的 Blackfin533 芯片。该实验板除了提供键盘、LED、LCD、触摸屏和串口等一些常用的功能模块外，还具有 IDE 硬件接口、PCI 接口、CF 存储卡接口、以太网接口、USB 接口、IrDA 接口、IIS 接口、SD 卡接口以及步进电机等丰富的接口模块，并可根据用户要求扩展 GPRS 等模块。Embest EduKit-III 实验平台能给用户在 32 位 ARM 嵌入式领域进行实验和开发提供丰富的支持和极大的便利。

3.1 ARM 汇编指令实验一

3.1.1 实验目的

- 初步学会使用 μ Vision3 IDE for ARM 开发环境及 ARM 软件模拟器；
- 通过实验掌握简单 ARM 汇编指令的使用方法。

3.1.2 实验设备

- 硬件：PC 机。
- 软件： μ Vision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

3.1.3 实验内容

- 熟悉开发环境的使用并使用 ldr/str, mov 等指令访问寄存器或存储单元；
- 使用 add/sub/lsl/lsr/and/orr 等指令，完成基本算术/逻辑运算。

3.1.4 实验原理

ARM 处理器共有 37 个寄存器：

- 31 个通用寄存器，包括程序计数器(PC)。这些寄存器都是 32 位的；
- 6 个状态寄存器。这些寄存器也是 32 位的，但是只是使用了其中的 12 位。

这里简要介绍通用寄存器，关于状态寄存器的介绍，请参照下一节。

3.1.4.1 ARM 通用寄存器

通用寄存器（R0-R15）可分为三类：

- 不分组寄存器 R0~R7；
- 分组寄存器 R8~R14；
- 程序计数器 PC。

(1) 不分组寄存器 R0~R7

不分组寄存器 R0~R7 在所有处理器模式下，它们每一个都访问一样的 32 位寄存器。它们是真正的通用寄存器，没有体系结构所隐含的特殊用途。

(2) 分组寄存器 R8~R14

分组寄存器 R8~R14 对应的物理寄存器取决于当前的处理器模式。若要访问特定的物理寄存器而不依赖当前的处理器模式，则使用规定的名字。

寄存器 R8~R12 各有两组物理寄存器：一组为 FIQ 模式，另一组为除了 FIQ 以外的所有模式。寄存器 R8~R12 没有任何指定的特殊用途，只是在作快速中断处理时使用。寄存器 R13, R14 各对应 6 个分组的物理寄存器，1 个用于用户模式和系统模式，其它 5 个分别用于 5 种异常模式。寄存器 R13 通常用做堆栈指针，称为 SP；寄存器 R14 用作子程序链接寄存器，也称为 LR。

(3) 程序计数器 PC

寄存器 R15 用做程序计数器 (PC)。

在本实验中，ARM 核工作在用户模式，R0~R15 可用。

3.1.4.2 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。字节零到字节三放置第一个字 (WORD)，字节四到字节七存储第二个字，以此类推。

ARM 体系结构可以用两种方法存储字数据，分别称为大端格式和小端格式。

● 大端格式

在这种格式中，字数据的高位字节存储在低地址中，而字数据的低位字节则存放在高地址中，如图 3-1 所示。

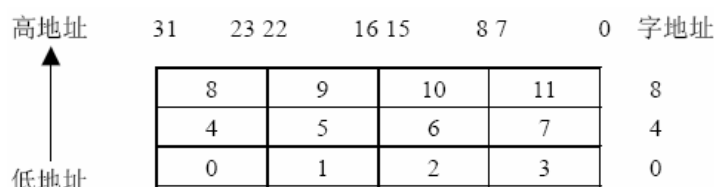


图 3-1 大端格式

● 小端格式

在这种格式中，字数据的高位字节存储在高地址中，而字数据的低位字节则存放在低地址中，如图 3-2 所示。

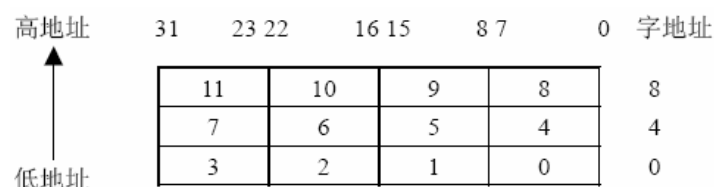


图 3-2 小端格式

3.1.4.4 REALVIEW 基础知识

μVision3 IDE 集成了 REALVIEW 汇编器 AARM、编译器 CARM、链接器 LARM，若采用 GNU 编译器则需要下载安装相应的工具包。本书所有例程代码均按照 REALVIEW 的语法和规则来书写。关于 AARM、CARM 和 LARM 的规范和具体使用，可参照 μVision3 IDE 所带的帮助文档，在此不再赘述。这里简单介绍几个相关基本知识：

● ENTRY

设置程序默认入口点，一个程序可有多条 ENTRY，但一个源文件最多只有一个 ENTRY。

● EQU

EQU 伪操用于将数字常量、基于寄存器的值和程序中的标号定义为一个字符名称。语法格式：

symbol EQU expression

其中, **expression** 可以是一个寄存器的名字, 也可由程序标号、常量或者 32 位的地址常量组成的表达式。**symbol** 是 EQU 伪操作所定义的字符名称。示例: **COUNT EQU 0X1FFF**

- **EXTERN/IMPORT**

IMPORT (**EXTERN** 功能完全相同)用于声明在其他模块中定义但需要在本文件中使用的符号。**EXTERN** 声明的变量必须是在其他模块中用 **EXPORT** 或 **GLOBAL** 声明过的。语法格式:

```
IMPORT class (symbol, symbol ...)
```

其中, **class** 为变量的类型, 可以为 ARM、CODE16、CODE32、DATA、CONST、THUMB; **symbol** 为所声明的变量名。

- **EXPORT/GLOBAL**

EXPORT (**GLOBAL** 功能完全相同)用于声明在本文件中定义但能在其他模块中使用的变量, 相当于定义了一个全局变量。语法格式:

```
EXPORT symbol, symbol...
```

其中, **symbol** 为所声明的变量名。

- **AREA**

AREA 用于定义一个代码段或数据段, ARM 汇编程序设计采用分段式设计, 一个 ARM 源程序至少有一个代码段, 大的程序会有若干个代码段和数据段。语法格式:

```
AREA segment-name, class-name, attributes ,...
```

其中, **segment-name** 为所定义段的名称; **class-name** 为所定义段的类型名称, 可以为系统类型 (CODE, CONST, DATA, ERAM) 或用户定义类型; **attributes** 为段的属性。

- **END**

END 用于标记汇编文件的结束行, 即标号后的代码不作处理。

3.1.5 实验操作步骤

3.1.5.1 新建工程

首先在\Keil\ARM\Examples\EduKit2410_for_MDK\3.1_asm1目录下建立文件夹命名为 **asm1_a**, 运行 **µVision3 IDE** 集成开发环境, 选择菜单项 **Project - New...** - **µVision Project**, 系统弹出一个对话框, 按照图 3-3 所示输入相关内容。点击“保存”按钮, 将创建一个新工程 **asm1_a.Uv2**。

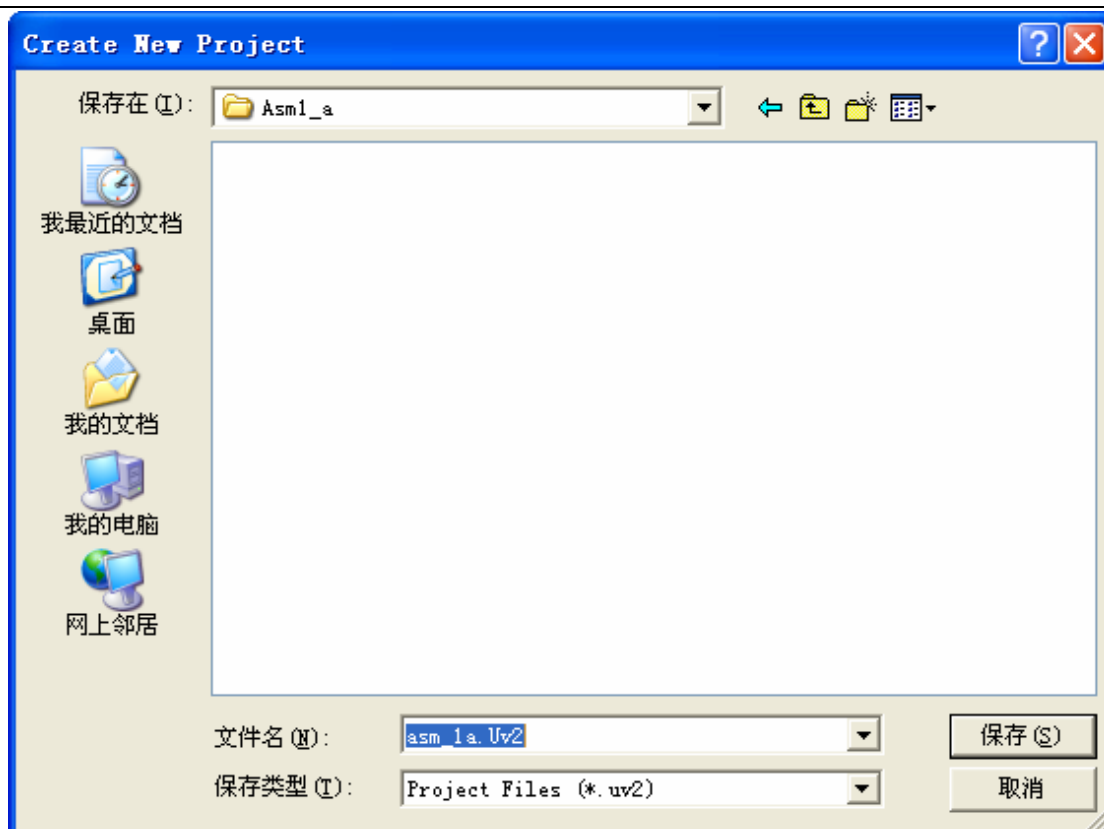


图 3-3 新建工程

3.1.5.2 为工程选择 CPU

新建工程后，要为工程选择 CPU，如图 3-4 所示，在此选择 SAMSUNG 的 S3C2410A

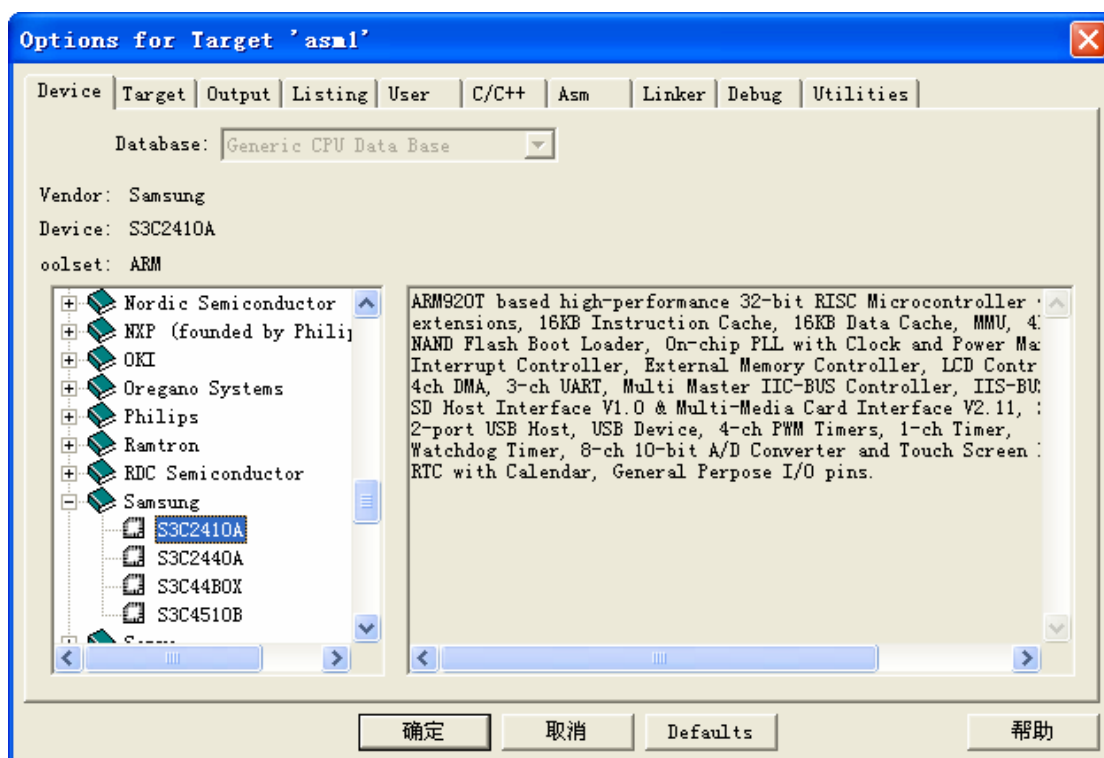


图 3-4 选择 CPU

3.1.5.3 添加启动代码

在图 3-4 中点“确定”后，会弹出一个对话框，问是否要添加启动代码。如图 3-5 所示。

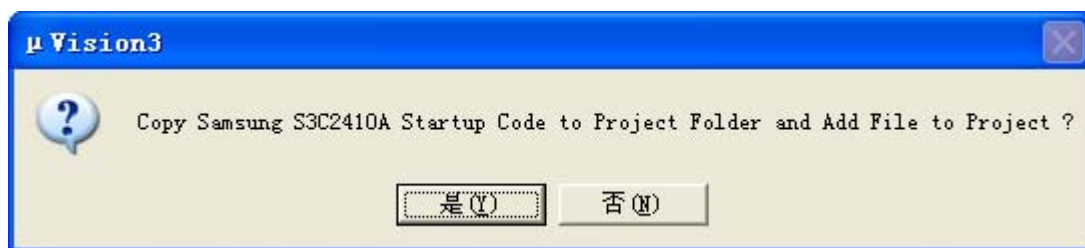


图 3-5 添加启动代码

由于本实验是简单的汇编实验，因此不需要启动代码，选择否。

3.1.5.4 选择开发工具

要为工程选择开发工具，在 Project - Manage - Components, Environment and Books - Folder/Extensions 对话框的 Folder/Extensions 页内选择开发工具，如图 3-6 所示。

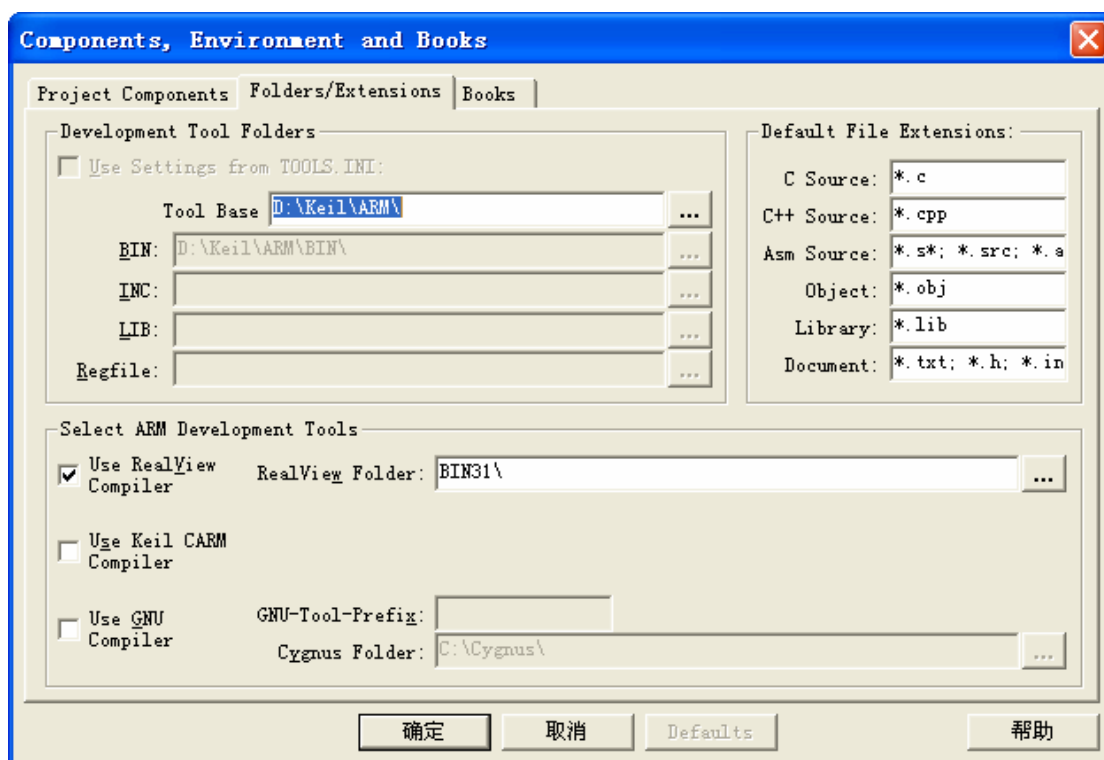


图 3-6 选择开发工具

从图中可以看到，有三个开发工具可选，在此选择 RealView Compiler。

3.1.5.5 建立源文件

点击菜单项 File - New，系统弹出一个新的、没有标题的文本编辑窗，输入光标位于窗口中第一行，按照实验参考程序编辑输入源文件代码。编辑完后，保存文件 asm1_a.s。（源代码可以参考光盘 \software\EduKit2410_for_MDK\3.1_asm1\Asm1_a 中的 asm1_a.s 文件）

3.1.5.6 添加源文件

单击工程管理窗口中的相应右键菜单命令，选择 Add Files to...，会弹出文件选择对话框，在工程目录下选择刚才建立的源文件 asm1_a.s。如图 3-7 所示。

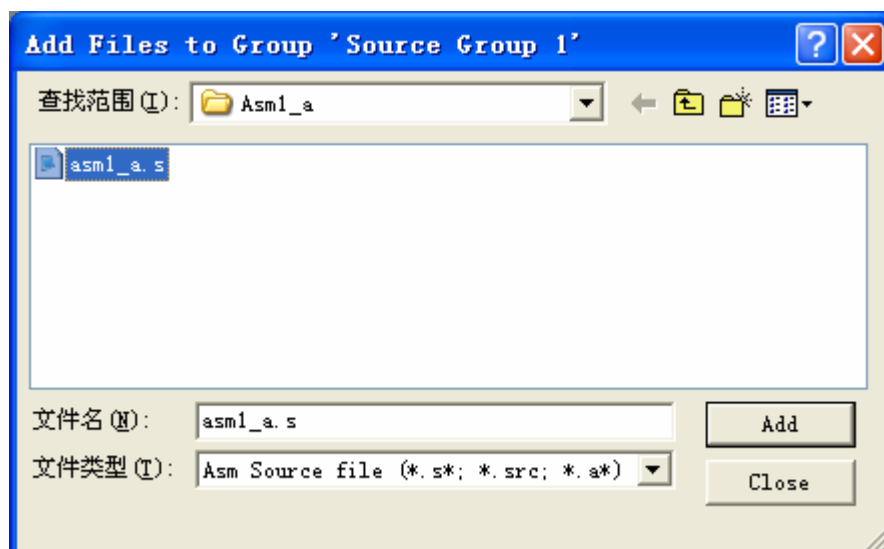


图 3-7 添加源文件

3.1.5.7 工程配置

把光盘 software\ EduKit2410_for_MDK\3.1_asm1\Asm1_a 目录中的 DebugINRam.ini 文件拷贝到\Keil\ARM\Examples\EduKit2410_for_MDK\3.1_asm1\Asm1_a 目录下。选择菜单项 Project->Option for Target..., 将弹出工程设置对话框, 如图 3-8 所示。对话框会因所选开发工具的不同而不同, 在此仅对 Target 选项页、Linker 选项页及 Debug 选项页进行配置。Target 选项页的配置如图 3-8; Linker 选项页的配置如图 3-9; Debug 选项页的配置如图 3-10。需要注意, 在 Debug 选项页内需要一个初始化文件: DebugINRam.ini。此.INI 文件用于设置生成的.AXF 文件下载到目标中的位置, 以及调试前的寄存器、内存的初始化等配置操作。它是由调试函数及调试命令组成调试命令脚本文件。

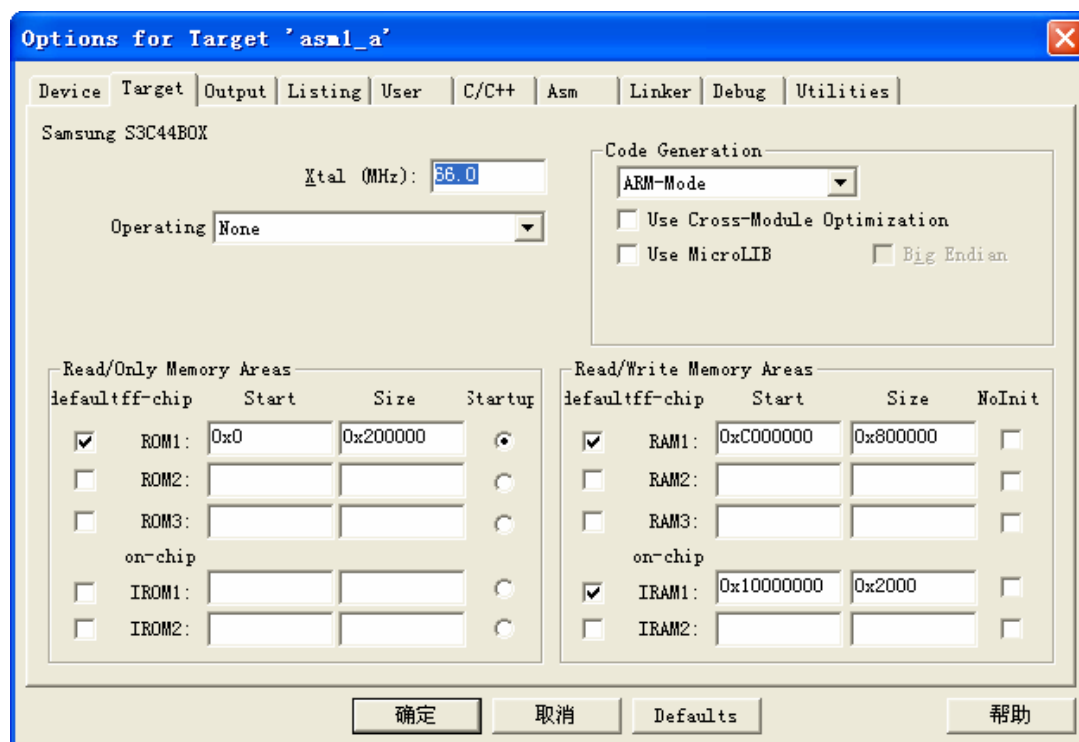


图 3-8 基本配置—Target

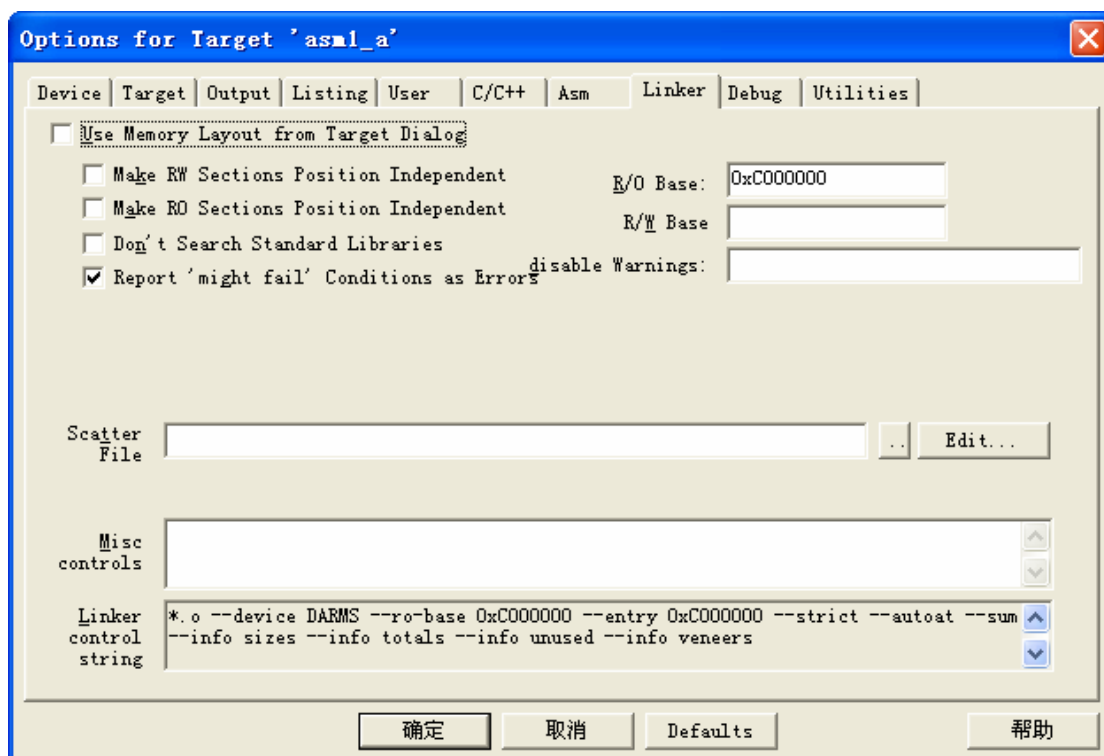


图 3-9 基本配置—Linker

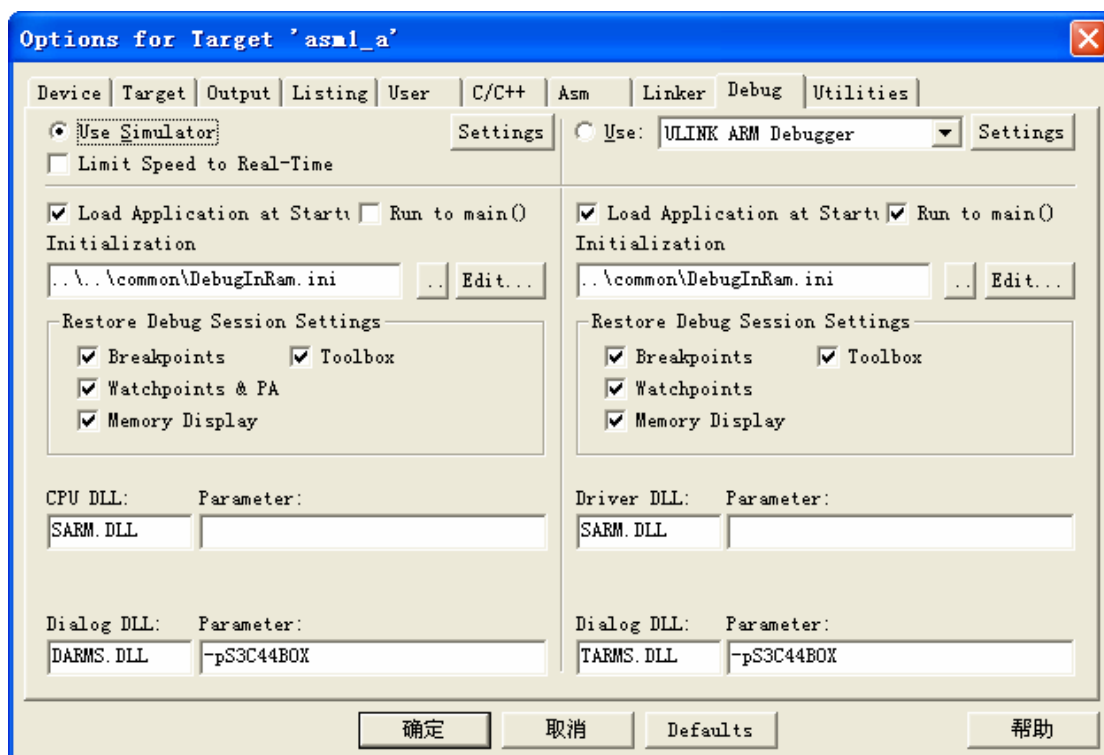


图 3-10 基本配置—Debug

3.1.5.8 生成目标代码

选择菜单项 **Project - Build target** 或快捷键 **F7**，生成目标代码。在此过程中，若有错误，则进行修改，直至无错误。若无错误，则可进行下一步的调试。

3.1.5.9 调试

选择菜单项 **Debug - Start/Stop Debug Session** 或快捷键 **Ctrl+F5**，即可进入调试模式。若没有目标硬件，可以用 μ Vision 3 IDE 中的软件仿真器。如果使用 MDK 试用版，则在进入调试模式前，会有如下对话框弹出，如图 3-11 所示。

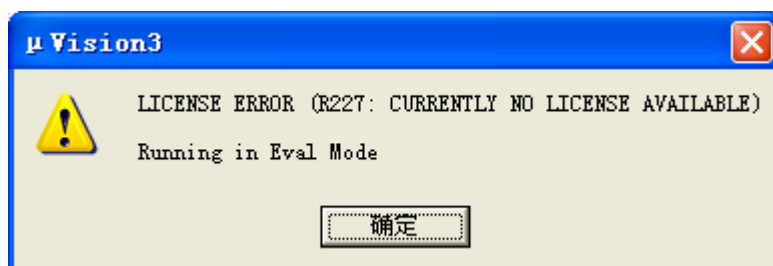


图 3-11 在软件仿真下调试程序

确定后即可调试了，做如下调试工作：

- ◆ 打开 **memory** 窗口，单步执行，观察地址 **0x30200000** 中内容的变化；
- ◆ 单步执行，观察寄存器的变化；
- ◆ 结合实验内容和相关资料，观察程序运行，通过实验加深理解 **ARM** 指令的使用；
- ◆ 理解和掌握实验后，完成实验练习题。

实验B与上述步骤完全相同，只要把对应的 **asm1_a.s** 文件改成 **asm1_b.s** 以及工程名即可。

3.1.6 实验参考程序

(1) 实验 A

汇编程序

```

;*****
; NAME:      asm1_a.s
*
; Author:    TYW/WUHAN R&D Center,Embest
*
; Desc:      ARM instruction examples
*
; History:   2007.5.1
*
;*****

; /*----- */
; /*          constant define
*/
; /*----- */
x      EQU 45          ; x=45
y      EQU 64          ; y=64/
stack_top EQU 0x30200000 ; define the top address for stacks

export Reset_Handler

; /*----- */
; /*          code
*/
; /*----- */

```



```

        AREA text, CODE, READONLY
        export
Reset_Handler          ; code start */
        ldr    sp, =stack_top
        mov    r0, #x                ; put x value into R0
        str    r0, [sp]              ; save the value of R0 into stacks
        mov    r0, #y                ; put y value into R0
        ldr    r1, [sp]              ; read the data from stack, and put it into
R1
        add    r0, r0, r1            ; R0=R0+R1
        str    r0, [sp]
        stop
        b      stop                  ; end the code f-cycling
        end

```

调试命令脚本文件

```

/** <<< Use Configuration !disalbe! Wizard in Context Menu >>> */
/*Name: DebugINRam.ini*/

FUNC void Setup (void)
{
    // <o> Program Entry Point, .AXF File download Address
    PC = 0x03000000;
}
map 0x00000000,0x00200000 read write exec //Map this memory to be read、write and exec
map 0x30000000,0x34000000 read write exec //Map this memeory to be read,write and exec
Setup();                                // Setup for Running
//g, main

```

(2) 实验 B

汇编程序

```

;#*****
;# NAME:      asm1_b.s
*
;# Author:    WUHAN R&D Center, Embest
*
;# Desc:      ARM instruction examples
*
;# History:    TianYunFang 2007.05.12
*
;#*****
;#----- */
;#
;#                      constant define
*/
;#----- */
x      EQU 45           ;/* x=45 */
y      EQU 64           ;/* y=64 */
z      EQU 87           ;/* z=87 */
stack_top EQU 0x30200000 ;/* define the top address for stacks*/
        export  Reset_Handler

;#----- */
;#                      code

```

```

*/
; /*----- */
AREA text, CODE, READONLY

Reset_Handler                                ; /* code start */
    mov     r0, #x                                ; /* put x value into R0 */
    mov     r0, r0, lsl #8                      ; /* R0 = R0 << 8 */
    mov     r1, #y                                ; /* put y value into R1 */
    add     r2, r0, r1, lsr #1                  ; /* R2 = (R1>>1) + R0 */
    ldr     sp, =stack_top
    str     r2, [sp]

    mov     r0, #z                                ; /* put z value into R0 */
    and     r0, r0, #0xFF                      ; /* get low 8 bit from R0 */
    mov     r1, #y                                ; /* put y value into R1 */
    add     r2, r0, r1, lsr #1                  ; /* R2 = (R1>>1) + R0 */

    ldr     r0, [sp]                                ; /* put y value into R1 */
    mov     r1, #0x01
    orr     r0, r0, r1
    mov     r1, R2                                ; /* put y value into R1 */
    add     r2, r0, r1, lsr #1                  ; /* R2 = (R1>>1) + R0 */

stop
    b       stop                                ; /* end the code f-cycling */
END

```

调试命令脚本文件与实验 A 相同。

3.1.7 练习题

1. 编写程序循环对 R4~R11 进行累加 8 次赋值, R4~R11 起始值为 1~8, 每次加操作后把 R4~R11 的内容放入 SP 栈中, SP 初始设置为 0x800。最后把 R4~R11 用 LDMFD 指令清空赋值为 0。
2. 更改实验 A 中 X、Y 的值, 观察执行结果。

3.2 ARM 汇编指令实验二

3.2.1 实验目的

- 通过实验掌握使用 ldm/stm, b, bl 等指令完成较为复杂的存储区访问和程序分支;
- 学习使用条件码, 加强对 CPSR 的认识;

3.2.2 实验设备

- 硬件: PC 机。
- 软件: μVision IDE for ARM 集成开发环境, Windows 98/2000/NT/XP。

3.2.3 实验内容

- 熟悉开发环境的使用并完成一块存储区的拷贝。
- 完成分支程序设计, 要求判断参数, 根据不同参数, 调用不同的子程序。

3.2.4 实验原理

1. ARM 程序状态寄存器

在所有处理器模式下都可以访问当前的程序状态寄存器 **CPSR**。**CPSR** 包含条件码标志，中断禁止位，当前处理器模式以及其它状态和控制信息。每种异常模式都有一个程序状态保存寄存器 **SPSR**。当异常出现时，**SPSR** 用于保存 **CPSR** 的状态。

CPSR 和 **SPSR** 的格式如下：

31	30	29	28	27		7	6	5	4	3	2	1	0
N	Z	C	V	Q	DNM (RAZ)	I	F	T	M	M	M	M	M

1) 条件码标志：

N, **Z**, **C**, **V** 大多数指令可以检测这些条件码标志以决定程序指令如何执行。

2) 控制位：

最低 8 位 **I**, **F**, **T** 和 **M** 位用做控制位。当异常出现时改变控制位。当处理器在特权模式下也可以由软件改变。

- 中断禁止位：**I** 置1 则禁止**IRQ** 中断；**F** 置1 则禁止**FIQ** 中断。
- **T** 位：**T**=0 指示**ARM** 执行；**T**=1 指示**Thumb** 执行。在这些体系结构系统中，可自由地使用能在**ARM** 和**Thumb** 状态之间切换的指令。
- 模式位：**M0**, **M1**, **M2**, **M3** 和**M4** (**M**[4:0]) 是模式位.这些位决定处理器的工作模式.如表 12-1 所示。

表 3-1 ARM 工作模式 **M**[4:0]

M [4:0]	模式	可访问的寄存器
0b10000	用户	PC, R14~R0, CPSR
0b10001	FIQ	PC, R14_fiq~R8_fiq, R7~R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq~R8_fiq, R12~R0, CPSR, SPSR_irq
0b10011	管理	PC, R14_svc~R8_svc, R12~R0, CPSR, SPSR_svc
0b10111	中止	PC, R14_abt~R8_abt, R12~R0, CPSR, SPSR_abt
0b11011	未定义	PC, R14_und~R8_und, R12~R0, CPSR, SPSR_und
0b11111	系统	PC, R14~R0, CPSR

3) 其他位

程序状态寄存器的其他位保留，用作以后的扩展。

2. 本实验涉及到的汇编指令语法及规则

ldr

ldr 伪指令将一个 32 位的常数或者一个地址值读取到寄存器中。当需要读取到寄存器中的数据超过了 **mov** 或者 **mnv** 指令可以操作的范围时，可以使用 **ldr** 伪指令将该数据读取到寄存器中。在汇编编译器处理源程序时，如果该常数没有超过 **mov** 或者 **mnv** 可以操作的范围，则 **ldr** 指令被这两

条指令中的一条所替代，否则，该常数将被放在最近的一个文字池内（literal pool），同时，本指令被一条基于 PC 的 ldr 指令替代。

语法格式：

ldr <register> , = <expression>

其中，expression 为需要读取的32 位常数；register 为目标寄存器。

示例：

```
ldr r1, =0xff
ldr r0, =0xffff0000
```

adr

adr 指令将基于 PC 的地址值或者基于寄存器的地址值读取到寄存器中。在汇编编译器处理源程序时，adr 伪指令被编译器替换成一条合适的指令。通常，编译器用一条 add 指令或者 sub 指令来实现该伪指令的功能。如果标号超出范围或者标号在同一文件（和同一段）内没有定义，则会产生一个错误。该指令不使用文字池（literal pool）。

语法格式：

adr <register> , <label>

其中，register 为目标寄存器；label为基于PC 或者寄存器的地址表达式。

示例：

```
label1
    mov r0, #25
    adr r2, label1
```

ltorg

ltorg 用于声明一个文字池。

语法格式：

ltorg

3.2.5 实验操作步骤

（1）实验 A

1) 在\Keil\ARM\Examples\EduKit2410_for_MDK\3.2_asm2 目录下建立文件夹命名为 Asm2_1，参考 3.1.5 小节实验的操作步骤建立一个新的工程，命名为 AsmTest2_1；

2) 参考 3.1.5 小节实验的步骤和实验参考程序编辑输入源代码，保存文件为 asm_code1.s(源代码可以参考光盘\software\EduKit2410_for_MDK\3.2_asm2\Asm2_1 中的 asm_code1.s 文件)；

3) 在 Project workspace 工作区中右击 target1->Source Group 1，在弹出菜单中选择 “Add file to Group ‘Source Group 1’ ”，在随后弹出的文件选择对话框中，选择刚才建立的源文件 asm_code1.s；

4) 把光盘\software\EduKit2410_for_MDK\3.2_asm2\Asm2_1 目录中的 DebugINRam.ini 文件拷贝到\Keil\ARM\Examples\EduKit2410_for_MDK\3.2_asm2\Asm2_1 目录下。选择菜单项 Project->Option for Target...，将弹出工程设置对话框，如图 3-8 所示。在这个工程里只需把 Linker 选项页的配置对话框中的 R/W Base 改为 0x30000000 即可。其他设置与 3.1.5 小节实验中的工程配置相同。

5) 选择菜单项 Project ->Build target 或快捷键 F7，生成目标代码；

6) 选择菜单项 **Debug ->Start/Stop Debug Session** 或快捷键 **Ctrl+F5**, 即可进入调试模式。这里使用的是 **μVision3 IDE** 中的软件仿真器;

7) 选择菜单项 **Debug ->run** 或 **F5**, 即可运行代码;

8) 打开 **memory** 窗口, 观察地址 **0x30000058~0x30000094** 的内容, 与地址 **0x300000a8~0x300000e4** 的内容;

9) 单步执行程序并观察和记录寄存器与 **memory** 的值变化, 注意观察步骤 8 里面的地址的内容变化, 当执行 **stmfd, ldmfd, ldmia** 和 **stmia** 指令的时候, 注意观察其后面参数所指的地址段或寄存器段的内容变化;

10) 结合实验内容和相关资料, 观察程序运行, 通过实验加深理解 **ARM** 指令的使用;

11) 理解和掌握实验后, 完成实验练习题。

(2) 实验 B

1) 参考 3.1.5 小节实验的操作步骤建立一个新的工程, 命名为 **AsmTest2_2**;

2) 参考 3.1.5 小节实验的步骤和实验参考程序编辑输入源代码, 保存文件为 **asmcode2.s** (如果存在, 可以调过此步骤);

3) 在 **Project workspace** 工作区中右击 **target1->Source Group 1**, 在弹出菜单中选择 “**Add file to Group 'Source Group 1'**”, 在随后弹出的文件选择对话框中, 选择刚才建立的源文件 **ThumbCode2.s**;

4) 参照实验 A 的步骤完成目标代码的生成与调试。

5) 理解和掌握实验后, 完成实验练习题。

3.2.6 实验参考程序

(1) 实验 A 参考源代码:

```

;#*****
;# NAME:   ARMcode.s
;#
;# Author:   EWUHAN  R & D Center, st
;#
;# Desc:   ARMcode examples
;#
;#          copy words from src to dst
;#
;# History:   shw.He 2005.02.22
;#
;#*****
;#----- */
;#/*                                     code
;#/*
;#/*----- */

GLOBAL Reset_Handler
area start,code,readwrite
entry
code32

num      EQU      20                                ;/* Set number of
words to be copied */

```

```

Reset_Handler
    ldr    r0, =src                /* r0 = pointer to source block */
    ldr    r1, =dst                /* r1 = pointer to destination block */
*/
    mov    r2, #num                /* r2 = number of words to copy */

    ldr    sp, =0x30200000         /* set up stack pointer (r13) */
blockcopy
    movs    r3, r2, LSR #3         /* number of eight word multiples */
    beq     copywords             /* less than eight words to move ? */

    stmfd   sp!, {r4-r11}         /* save some working registers */
octcopy
    ldmia   r0!, {r4-r11}         /* load 8 words from the source */
    stmia   r1!, {r4-r11}         /* and put them at the destination */
    subs    r3, r3, #1            /* decrement the counter */
    bne     octcopy               /* ... copy more */

    ldmfd   sp!, {r4-r11}         /* don't need these now - restore
originals */

copywords
    ands    r2, r2, #7            /* number of odd words to copy */
    beq     stop                  /* No words left to copy ? */
wordcopy
    ldr     r3, [r0], #4           /* a word from the source */
    str     r3, [r1], #4           /* store a word to the destination */
    subs    r2, r2, #1            /* decrement the counter */
    bne     wordcopy              /* ... copy more */

stop
    b       stop

/*----- */
/*                               make a word pool
*/
/*----- */

ltorg
src
    dcd     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst
    dcd     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
end

```

(2) 实验 B 参考源代码:

```

;#*****
;# NAME:   ARM_code2.s
;#
;# Author:   WUHAN R&D Center, Embest
;#
;# Desc:   ARM instruction examples

```

```

*
;#      Example for Condition Code
*
;# History:    shw.He 2005.02.22
*
;#*****

; /*----- */
                        code
; /*----- */

        area start,code,readwrite
        entry
        code32
num equ    2                        ;/* Number of entries in jump table */
        export Reset_Handler
Reset_Handler
        mov     r0, #0                ;/* set up the three parameters */
        mov     r1, #3
        mov     r2, #2
        bl      arithfunc             ;/* call the function */

stop
        b       stop

;# *****
;# * According R0 valude to execute the code
*
;# *****
arithfunc                                ;/* label the function */
        cmp     r0, #num               ;/* Treat function code as unsigned
integer */
        bhs     DoAdd                  ;/* If code is >=2 then do operation 0.
*/

        adr     r3, JumpTable           ;/* Load address of jump table */
        ldr     pc, [r3,r0,LSL#2]       ;/* Jump to the appropriate routine */

JumpTable
        dcd     DoAdd
        dcd     DoSub

DoAdd
        add     r0, r1, r2              ;/* Operation 0, >1 */
        bx      lr                     ;/* Return */

DoSub
        sub     r0, r1, r2              ;/* Operation 1 */

```



```

bx      lr                                ;/* Return */

end                                         ;/* mark the end of this file */

```

本节可使用 3.1 节的调试脚本文件。

3.2.7 练习题

1. 打开开发板光盘中的启动文件，观察启动文件中复位异常的编写及 Itorg 的使用及其功能；
2. 新建工程，并自行编写汇编程序，分别使用 ldr、str、ldmia、stmia 操作，实现对某段连续存储单元写入数据，并观察操作结果。

3.3 Thumb 汇编指令实验

3.3.1 实验目的

- 通过实验掌握 ARM 处理器 16 位 Thumb 汇编指令的使用方法。

3.3.2 实验设备

- 硬件：PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

3.3.3 实验内容

- 使用 THUMB 汇编语言，完成基本 reg/mem 访问，以及简单的算术/逻辑运算。
- 使用 THUMB 汇编语言，完成较为复杂的程序分支，push/pop，领会立即数大小的限制，并体会 ARM 工作状态与 THUMB 工作状态的区别。

3.3.4 实验原理

(1) ARM 处理器工作状态

ARM 处理器共有两种工作状态：

ARM：32 位，这种状态下执行字对准的 ARM 指令；

Thumb：16 位，这种状态下执行半字对准的 Thumb 指令。

在 Thumb 状态下，程序计数器 PC 使用位 1 选择另一个半字。

注意：ARM 和 Thumb 之间状态的切换不影响处理器的模式或寄存器的内容。

ARM 处理器在两种工作状态之间可以切换。

进入 Thumb 状态。当操作数寄存器的状态位（位 0）为 1 时，执行 BX 指令进入 Thumb 状态。如果处理器在 Thumb 状态进入异常，则当异常处理（IRQ，FIQ，Undef，Abort 和 SWI）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（位 0）为 0 时，执行 BX 指令进入 ARM 状态。此外，在处理器进行异常处理（IRQ，FIQ，Undef，Abort 和 SWI）时，把 PC 放入异常模式链接寄存器中，从异常向量地址开始执行也可进入 ARM 状态。

(2) Thumb 状态的寄存器集

Thumb 状态下的寄存器集是 ARM 状态下寄存器集的子集。程序员可以直接访问 8 个通用的寄存器(R0~R7)，PC,SP,LR 和 CPSP。每一种特权模式都有一组 SP，LR 和 SPSR。

Thumb 状态的 R0~R7 与 ARM 状态的 R0~R7 是一致的；

Thumb 状态的 CPSR 和 SPSR 与 ARM 状态的 CPSR 和 SPSR 是一致的；

Thumb 状态的 SP 映射到 ARM 状态的 R13；

Thumb 状态的 LR 映射到 ARM 状态的 R14；

Thumb 状态的 PC 映射到 ARM 状态的 PC (R15)。

Thumb 寄存器与 ARM 寄存器的关系如图3-7 所示。

(3) 本实验涉及到的伪操作

Code [16|32]

code 伪操作用于选择当前汇编指令的指令集。参数 16 选择 Thumb 指令集，参数 32 选择 ARM 指令集。

语法格式：

code[16|32]

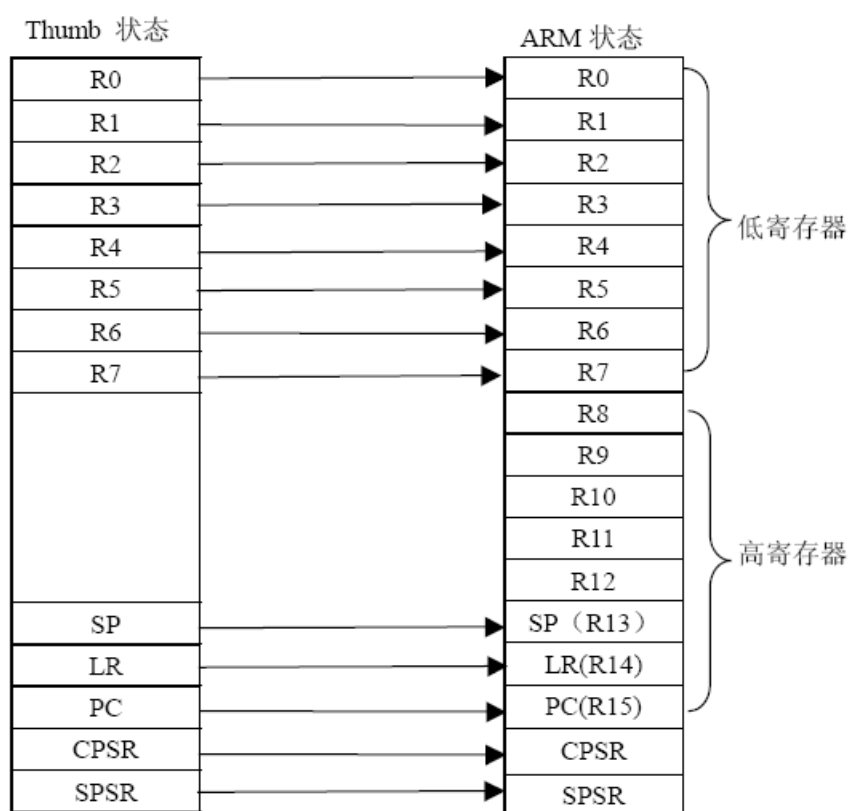


图 3-7 寄存器状态图

thumb

同 code 16。

arm

同 code 32。

align

align 伪指令通过添加补丁字节使当前位置满足一定的对齐方式。

语法格式：

align {expr{, offset}}

其中: **expr** 为数字表达式, 用于指定对齐的方式。取值为 2 的 n 次幂, 如 1、2、4、8 等, 不能为 0。若没有 **expr**, 则默认为字对齐方式。

Offset 为数字表达式。当前位置对齐到下面形式的地址处: $\text{offset} + n * \text{expr}$

示例

`align 4, 3` ; 字对齐

3.3.5 实验操作步骤

(1) 实验 A

1) 在 \Keil\ARM\Examples\EduKit2410_for_MDK\3.3_thumbcode\ThumbTest_1 目录下建立一个新的工程, 命名为 `Thumb_test1`;

2) 参考 3.1.5 小节实验的步骤和实验参考程序编辑输入源代码, 保存文件为 `ThumbCode1.s`;

3) 在 Project workspace 工作区中右击 `target1->Source Group 1`, 在弹出菜单中选择 “Add file to Group ‘Source Group 1’”, 在随后弹出的文件选择对话框中, 选择刚才建立的源文件 `ThumbCode1.s`;

4) 选择菜单项 `Project ->Build target` 或快捷键 `F7`, 生成目标代码;

5) 选择菜单项 `Debug ->Start/Stop Debug Session` 或快捷键 `Ctrl+F5`, 即可进入调试模式。这里使用的是 μ Vision3 IDE 中的软件仿真器;

6) 选择菜单项 `Debug ->run` 或 `F5`, 即可运行代码;

7) 记录代码执行区中每条指令的地址, 注意指令最后尾数的区别;

8) 观察 Project workspace 工作区中寄存器的变化, 特别是 `R0` 和 `R1` 的值的变化;

9) 结合实验内容和相关资料, 观察程序运行, 通过实验加深理解 ARM 指令和 Thumb 指令的不同;

10) 理解和掌握实验后, 完成实验练习题。

(2) 实验 B

1) 在 \Keil\ARM\Examples\EduKit2410_for_MDK\3.3_thumbcode\ThumbTest_2 目录下建立一个新的工程, 命名为 `Thumb_test2`;

2) 参考 3.1.5 小节实验的步骤和实验参考程序编辑输入源代码, 保存文件为 `ThumbCode2.s`;

3) 在 Project workspace 工作区中右击 `target1->Source Group 1`, 在弹出菜单中选择 “Add file to Group ‘Source Group 1’”, 在随后弹出的文件选择对话框中, 选择刚才建立的源文件 `ThumbCode2.s`;

4) 选择菜单项 `Project ->Build target` 或快捷键 `F7`, 生成目标代码;

5) 选择菜单项 `Debug ->Start/Stop Debug Session` 或快捷键 `Ctrl+F5`, 即可进入调试模式。这里使用的是 μ Vision3 IDE 中的软件仿真器;

6) 选择菜单项 `Debug ->run` 或 `F5`, 即可运行代码;

7) 注意并记录 ARM 指令下和 Thumb 指令状态下 `stmfd`, `ldmfd`, `ldmia` 和 `stmia` 指令执行的结果, 指令的空间地址数值, 数据存储的空间大小等等;

8) 理解和掌握实验后, 完成实验练习题。

3.3.6 实验参考程序

(1) 实验 A 参考源代码:

```
;#*****
```

```

;# NAME:   ThumbCode.s                               *
;# Author:   Wuhan R&D Center, Embest                 *
;# Desc:    ThumbCode examples                       *
;# History:   WuHan R&D Center 2007.01.12            *
;#*****
;/*-----*/
;/*          unable to locate source file.          code*/
;/*-----*/

    area start,code,readonly
    entry
    code32                ;/* Subsequent instructions are ARM */
    export Reset_Handler

Reset_Handler
    adr    r0, Tstart + 1    ;/* Processor starts in ARM state, */
    bx     r0                ;/* so small ARM code header used */
                                ;/* to call Thumb main program.   */
    nop
    code16

Tstart
    mov     r0, #10          ;/* Set up parameters */
    mov     r1, #3
    bl      doadd            ;/* Call subroutine */

stop
    b       stop

;/*-----*/
;/* Subroutine code:R0 = R0 + R1 and return          */
;/*-----*/
doadd
    add     r0, r0, r1       ;/* Subroutine code */
    mov     pc, lr          ;/* Return from subroutine. */
    end      ;/* Mark end of file */

```

本节可使用 3.1 节的调试脚本文件。

(2) 实验 B 参考源代码:

```

;*****
; NAME:   Thumbcode2.s                               *
; Author: WuHan R&D Center, Embest                 *
; Desc:   Thunmbcode examples                       *
;         copy words from src to dst                *
; History: WuHan R&D Center 2007.01.12            *
;*****
;                               code                  */
;-----*/

    area start,code,readonly
    entry
    code32                ;/* Subsequent instructions are ARM */
    num equ 20             ;/* Set number of words to be copied */
    export Reset_Handler

Reset_Handler

```

```

/* Subsequent instructions are ARM header
*/
    ldr    sp, =0x30200000          /* set up user_mode stack pointer (r13)
*/
    adr    r0, Tstart + 1           /* Processor starts in ARM state,
*/
    bx     r0                      /* so small ARM code header used */
                                   /* to call Thumb main program. */
    code16                          /* Subsequent instructions are Thumb. */
Tstart
    ldr     r0, =src                 /* r0 = pointer to source block */
    ldr     r1, =dst                 /* r1 = pointer to destination block */
    mov     r2, #num                 /* r2 = number of words to copy */
blockcopy
    lsr     r3, r2, #2               /* number of four word multiples */
    beq     copywords               /* less than four words to move ? */
    push    {r4-r7}                 /* save some working registers */
quadcopy
    ldmia   r0!, {r4-r7}             /* load 4 words from the source */
    stmia   r1!, {r4-r7}             /* and put them at the destination */
    sub     r3, #1                  /* decrement the counter */
    bne     quadcopy                /* ... copy more */
    pop     {r4-r7}                 /* don't need these now - restore originals */
copywords
    mov     r3, #3                  /* bottom two bits represent
number... */
    and     r2, r3                  /* ...of odd words left to copy */
    beq     stop                    /* No words left to copy ? */
wordcopy
    ldmia   r0!, {r3}               /* a word from the source */
    stmia   r1!, {r3}               /* store a word to the destination */
    sub     r2, #1                  /* decrement the counter */
    bne     wordcopy                /* ... copy more */
stop
    b       stop

/*----- */
/* make a word pool */
/*----- */
    align
src
    dcd     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst
    dcd     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
end

```

3.3.7 练习题

编写程序从 ARM 状态切换到 Thumb，在 ARM 状态下把 R2 赋值为 0x12345678，在 Thumb 状态下把 R2 赋值为 0x87654321。同时观察并记录 CPSR，SPSR 的值，分析各个标志位。

3.4 ARM 处理器工作模式实验

3.4.1 实验目的

- 通过实验掌握学会使用 `msr/mrs` 指令实现 ARM 处理器工作模式的切换；
- 观察不同模式下的寄存器，加深对 CPU 结构的理解；
- 通过实验进一步熟悉 ARM 汇编指令。

3.4.2 实验设备

- 硬件：PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

3.4.3 实验内容

通过 ARM 汇编指令，在各种处理器模式下切换并观察各种模式下寄存器的区别；掌握 ARM 不同模式的进入与退出。

3.4.4 实验原理

1. ARM 处理器模式

ARM 体系结构支持下表 3-2 所列的 7 种处理器模式。

表 3-2 处理器模式

处理器模式	说明
用户 usr	正常程序执行模式
FIQ fiq	支持高速数据传送或通道处理
IRQ irq	用于通用中断处理
管理 svc	操作系统保护模式
中止 abt	实现虚拟存储器和/或存储器保护
未定义 und	支持硬件协处理器的软件仿真
系统 sys	运行特权操作系统任务

在软件控制下可以改变模式，外部中断或异常处理也可以引起模式发生改变。

大多数应用程序在用户模式下执行。当处理器工作在用户模式时，正在执行的程序不能访问某些被保护的系统资源，也不能改变模式，除非异常(exception)发生。这允许通过合适地编写操作系统来控制系统资源的使用。

除用户模式外的其他模式称为特权模式。它们可以自由的访问系统资源和改变模式。其中的 5 种称为异常模式，即：

- FIQ (Fast Interrupt request)；
- IRQ (Interrupt ReQuest)；

- 管理 (Supervisor);
- 中止(Abort) ;
- 未定义(Undefined) 。

当特定的异常出现时, 进入相应的模式。每种模式都有某些附加的寄存器, 以避免异常出现时用户模式的状态不可靠。

剩下的模式是系统模式。仅 ARM 体系结构 V4 及其以上的版本有该模式。不能由任何异常而进入该模式。它与用户模式有完全相同的寄存器, 然而它是特权模式, 不受用户模式的限制。它供需要访问系统资源的操作系统任务使用, 但希望避免使用与异常模式有关的附加寄存器。避免使用附加寄存器保证了当任何异常出现时, 都不会使任务的状态不可靠。

2. 程序状态寄存器

前一节提到的程序状态寄存器 CPSR 和 SPSR 包含了条件码标志, 中断禁止位, 当前处理器模式以及其他状态和控制信息。每种异常模式都有一个程序状态保存寄存器 SPSR。当异常出现时, SPSR 用于保留 CPSR 的状态。

CPSR 和 SPSR 的格式如下:

31	30	29	28	27	26		8	7	6	5	4	3	2	1	0
N	Z	C	V	Q	DNM(RAZ)		I	F	T	M	M	M	M	M	M

1) 条件码标志 :

N, Z, C, V 大多数指令可以检测这些条件码标志以决定程序指令如何执行

2) 控制位 :

最低 8 位 I, F, T 和 M 位用作控制位。当异常出现时改变控制位。当处理器在特权模式下也可以由软件改变。

- 中断禁止位:I 置 1 则禁止 IRQ 中断; F 置 1 则禁止 FIQ 中断。
- T 位:T=0 指示 ARM 执行; T=1 指示 Thumb 执行。在这些体系结构的系统中, 可自由的使用能在 ARM 和 Thumb 状态之间切换的指令。
- 模式位:M0, M1, M2, M3 和 M4 (M[4:0]) 是模式位.这些位决定处理器的工作模式.如表 3-3 所示。

表 3-3 ARM 工作模式 M[4:0]

M[4:0]	模式	可访问的寄存器
0b10000	用户	PC, R14~R0,CPSR
0b10001	FIQ	PC, R14_fiq~R8_fiq,R7~R0,CPSR,SPSR_fiq
0b10010	IRQ	PC, R14_irq~R8_irq,R12~R0,CPSR,SPSR_irq
0b10011	管理	PC, R14_svc~R8_svc,R12~R0,CPSR,SPSR_svc
0b10111	中止	PC, R14_abt~R8_abt,R12~R0,CPSR,SPSR_abt
0b11011	未定义	PC, R14_und~R8_und,R12~R0,CPSR,SPSR_und
0b11111	系统	PC, R14~R0,CPSR


3) 其他位

程序状态寄存器的其他位保留, 用作以后的扩展。

3.4.5 实验操作步骤

1) 参考 3.1.5 小节实验 A 的步骤建立一个新的工程, 命名为 ARMMode, 处理器选择 S3C2410A;

2) 参考 3.1.5 小节实验 A 的步骤和实验参考程序编辑输入源代码，编辑完毕后，保存文件为 `armmode.s`;

3) 单击工具栏的  图标，或单击工程管理窗口中的相应右键菜单 **Manage Components** 命令，弹出 **Componets, Environment and Books** 对话框，在该对话框中为相应的文件组添加刚才新建的源文件 **ARMMode.s**;

4) 参考 3.1.5 小节实验 A 的步骤进行相应设置、生成目标代码和下载目标代码调试;

5) 打开寄存器窗，单步执行，观察并记录寄存器 **R0** 和 **CPSR** 的值的变化和每次变化后执行寄存器赋值后的 36 个寄存器的值的变化情况，尤其注意各个模式下 **R13** 和 **R14** 的值;

6) 结合实验内容和相关资料，观察程序运行，通过实验加深理解 ARM 各种状态下寄存器的使用;

7) 理解和掌握实验后，完成实验练习题。

3.4.6 实验参考程序

```

*****
;
; NAME:      ARMmode.s                                *
; Author:    Wuhan R&D Center, Embest                  *
; Desc:      ARM instruction examples                  *
;            Example for ARM mode                     *
; History:    JianYing, Wang 2007.05.15               *
*****

; /*-----*/
; /*                                constant define                                */
; /*-----*/

EXPORT start

; /*-----*/
; /*                                code                                           */
; /*-----*/

AREA    |.text|, CODE, READONLY

start
; /*-----*/
; /* Setup interrupt / exception vectors                                          */
; /*-----*/

    b      Reset_Handler
Undefined_Handler
    b      Undefined_Handler
    b      SWI_Handler
Prefetch_Handler
    b      Prefetch_Handler
Abort_Handler
    b      Abort_Handler
    nop
; /* Reserved vector */
IRQ_Handler

```



```

        b        IRQ_Handler

FIQ_Handler
        b        FIQ_Handler

SWI_Handler
        bx lr

Reset_Handler

visitmen
;-----*/
;/*   into System mode                               */
;-----*/
        mrs     r0,cpsr                ;/* read CPSR value          */
        bic     r0,r0,#0x1f            ;/* clear low 5 bit           */
        orr     r0,r0,#0x1f            ;/* set the mode as System mode */
        msr     cpsr_cxfs,r0

        mov     r0, #1                 ;/* initialization the register in System mode */
        mov     r1, #2
        mov     r2, #3
        mov     r3, #4
        mov     r4, #5
        mov     r5, #6
        mov     r6, #7
        mov     r7, #8
        mov     r8, #9
        mov     r9, #10
        mov     r10, #11
        mov     r11, #12
        mov     r12, #13
        mov     r13, #14
        mov     r14, #15

;-----*/
;/*   into FIQ mode                               */
;-----*/
        mrs     r0,cpsr                ;/* read CPSR value          */
        bic     r0,r0,#0x1f            ;/* clear low 5 bit           */
        orr     r0,r0,#0x11            ;/* set the mode as FIQ mode   */
        msr     cpsr_cxfs,r0

        mov     r8, #16                 ;/* initialization the register in FIQ mode */
        mov     r9, #17
        mov     r10, #18
        mov     r11, #19
        mov     r12, #20
        mov     r13, #21
        mov     r14, #22

```

```

; /*-----*/
; /*  into SVC mode */
; /*-----*/
    mrs  r0,cpsr                ;/* read CPSR value */
    bic  r0,r0,#0x1f            ;/* clear low 5 bit */
    orr  r0,r0,#0x13            ;/* set the mode as SVC mode */
    msr  cpsr_cxfs,r0

    mov  r13, #23                ;/* initialization the register in SVC mode */
    mov  r14, #24

; /*-----*/
; /*  into Abort mode */
; /*-----*/
    mrs  r0,cpsr                ;/* read CPSR value */
    bic  r0,r0,#0x1f            ;/* clear low 5 bit */
    orr  r0,r0,#0x17            ;/* set the mode as Abort mode */
    msr  cpsr_cxfs,r0

    mov  r13, #25                ;/* initialization the register in Abort mode */
    mov  r14, #26

; /*-----*/
; /*  into IRQ mode */
; /*-----*/
    mrs  r0,cpsr                ;/* read CPSR value */
    bic  r0,r0,#0x1f            ;/* clear low 5 bit */
    orr  r0,r0,#0x12            ;/* set the mode as IRQ mode */
    msr  cpsr_cxfs,r0

    mov  r13, #27                ;/* initialization the register in IRQ mode */
    mov  r14, #28

; /*-----*/
; /*  into UNDEF mode */
; /*-----*/
    mrs  r0,cpsr                ;/* read CPSR value */
    bic  r0,r0,#0x1f            ;/* clear low 5 bit */
    orr  r0,r0,#0x1b            ;/* set the mode as UNDEF mode */
    msr  cpsr_cxfs,r0

    mov  r13, #29                ;/* initialization the register in UNDEF mode */
    mov  r14, #30

    b    Reset_Handler          ;/* jump back to Reset_Handler */

```

END

本节可使用 3.1 节的调试脚本文件。

3.4.7 练习题

参考第一个例子，把其中系统模式程序更改为用户模式程序，编译调试，观察运行结果，检查是否正确，如果有错误，分析其原因；（提示：不能从用户模式直接切换到其他模式，可以先使用 SWI 指令切换到管理模式）。

3.5 C 语言实例一

3.5.1 实验目的

学会使用 μ Vision IDE for ARM 开发环境编写简单的 C 语言程序；

学会编写和使用调试函数；

掌握通过 memory/register/watch/variable 窗口分析判断运行结果。

3.5.2 实验设备

- 硬件：PC 机。
- 软件： μ Vision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

3.5.3 实验内容

利用函数初始化栈指针，并使用 c 语言完成延时函数。

3.5.4 实验原理

1. 调试函数

μ Vision3 具有强大的调试功能，其中之一就是它的调试函数。 μ Vision3 有一个内嵌的调试函数编辑器，可以通过 Debug -> Function Editor 打开。在此编辑器中，可以写编写调试函数并可以编译此函数。调试函数的功能有：

扩展的 μ Vision3 Debugger 的能力；

产生外部中断；

生成存储器内容文件；

定期更新模拟输入值；

输入串行数据到片上串口；

其它。

具体来说，用户在集成环境与目标板连接时、软件调试过程中以及复位目标板后，有时需要集成环境自动完成一些特定的功能，比如复位目标板、清除看门狗、屏蔽中断寄存器、存储区映射等，这些特定的功能可以通过执行一组命令序列完成。而这一组命令序列可以写在调试函数中。

2. 调试函数的执行方法

编写好调试函数后，可以使用 INCLUDE 命令读取并处理调试函数。若保存调试函数的脚本文件名为 MYFUNCS.INI，在命令窗口中输入如下命令 μ Vision3 就可读取并解释 MYFUNCS.INI 中的内容。

>INCLUDE MYFUNCS.INI

MYFUNCS.INI 可以包含调试命令和函数定义，当然也可以通过如下的方式来执行：把此文件放入 Options for Target -> Debug -> Initialization File 内，这样每当启动 μ Vision3 Debugger 器时，MYFUNCS.INI 中的内容就会被处理。

3. 常用命令介绍**GO**

GO 用于指定程序从那里执行及在那里结束。

指令格式：Go startaddr, stopaddr

若 startaddr 被指定，则程序从 startaddr 处开始执行，否则从当前地址开始执行。若 stopaddr 被指定，则程序在 stopaddr 处结束，否则，运行到最近的断点处。

命令举例：

G,main //从 main 处开始执行

Display

Display 用于显示存储区域的内容。

指令格式：Display startaddr, endaddr

在命令窗口或存储器窗口（若打开）显示从 startaddr 到 endaddr 区域中的内容。可以以各种格式来显示存储器中的内容。

命令举例：

D main /* Output beginning at main */

其它命令请参考帮助文档 DEBUG COMMAND

3.5.5 实验操作步骤

- 1) 参考 3.1.5 小节实验的操作步骤建立一个新的工程，命名为 C_Test1；
- 2) 参考 3.1.5 小节实验的步骤和实验参考程序编辑输入源代码，保存文件为 C1.c；
- 3) 按照实验参考程序建立调试脚本文件 DebugInRam.ini；
- 4) 在 Project workspace 工作区中右击 target1->Source Group 1，在弹出菜单中选择 "Add file to Group 'Source Group 1' "，在随后弹出的文件选择对话框中，选择刚才建立的源文件 C_Call.c；
- 5) 选择菜单项 Project ->Build target 或快捷键 F7，生成目标代码；
- 6) 选择菜单项 Debug ->Start/Stop Debug Session 或快捷键 Ctrl+F5，即可进入调试模式。这里使用的是 μ Vision3 IDE 中的软件仿真器；在 Option For Target 对话框的 Debug 页中将 Initialization 文本框的内容清空；
- 7) 选择菜单项 Debug ->run 或 F5，即可运行代码；
- 8) 在 Output Windows 中的 Command 输入栏中输入 "Include DebugInRam.ini" 命令；
- 9) 单步执行，通过 memory、register、watch&call stack 等窗口分析判断结果，在 watch 框中输入要观察变量 I 和变量 J 的值，并记录下来。特别注意观察变量 I 的变化并记录下来；
- 10) 结合实验内容和相关资料，学习和尝试一些调试命令，观察程序运行，通过实验；
- 11) 理解和掌握实验后，完成实验练习题。

3.5.6 实验参考程序

1. C 程序

```

/*-----*/
/* NAME      :      Cl.C                      */
/* AUTHOR     :      Wuhan R&D Center, Embest  */
/* DESC       :      C EXAMPLE                  */
/* HISTORY    :      1.8.2006                  */
/* MODIFY     :      TYW,Wuhan R&D Center      */
/*-----*/

/*-----/*
      function declare                      */
/*-----*/
void delay(int nTime);
/*-----*/
/* NAME      :      START                      */
/* FUNC       :      ENTRY POINT                */
/* PARA       :      NONE                      */
/* RET        :      NONE                      */
/* MODIFY     :                                  */
/* COMMENT    :                                  */
/*-----*/

main()
{
    int i = 5;
    for( i ; )
    {
        delay(i);
    }
}
/*-----*/
/* NAME      :      DELAY                      */
/* FUNC       :      DELAY SOME TIME            */
/* PARA       :      nTime -- INPUT             */
/* RET        :      NONE                      */
/* MODIFY     :                                  */
/* COMMENT    :                                  */
/*-----*/
void delay(int nTime)
{
    int i, j = 0;
    for(i = 0; i < nTime; i++)
    {
        for(j = 0; j < 10; j++)
        {
        }
    }
}

```

2. 调试脚本文件

```

/** <<< Use Configuration !disalbe! Wizard in Context Menu >>> **
FUNC void Setup (void)
{
    // <o> Program Entry Point
    PC =main;;
}
//LOAD debug_in_RAM\Project.axf INCREMENTAL      // Download
//map 0x000,0x200000 READ WRITE EXEC
map 0x30000000,0x30200000 READ WRITE exec
Setup();          // Setup for Running
//g, main

```

3.5.7 练习题

参考汇编实验，编写程序，实现从汇编语言中使用 B 或 BL 命令跳转到 C 语言程序的 Main()函数中执行，并从 Main()函数中调用 delay()函数。

3.6 C 语言实验程序二

3.6.1 实验目的

掌握建立基本完整的 ARM 工程，包含启动代码，连接属性的配置等；

了解 ARM9 的启动过程，学会使用 MDK 编写简单的 C 语言程序和汇编启动代码并进行调试；

掌握如何指定代码入口地址与入口点；

掌握通过 memory、register、watch、Local 等窗口分析判断结果。

3.6.2 实验设备

- 硬件：PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

3.6.3 实验内容

用 c 语言编写延时函数，使用嵌入汇编。

3.6.4 实验原理

1. ARM 异常向量表

当正常的程序执行流程暂时挂起时，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，必须保存当前的处理器状态，以便从异常程序返回时可以继续执行当前的程序。ARM 异常向量表如下：

表 3-4 ARM 异常向量表

地址	异常	入口模式
0x00000000	RESET	管理
0x00000004	Undefined Instruction	未定义
0x00000008	Software interrupt	管理
0x0000000C	Prefetch abort	中止
0x00000010	Data abort	中止

0x00000014	Reserved	保留
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

处理器允许多个异常同时发生，这时，处理器会按照固定的顺序进行处理，参照下面的异常优先级。

高优先级：

1 ---- Reset

2 ---- Data abort

3 ---- FIQ

4 ---- IRQ

5 ---- Prefetch abort

低优先级：

6 ---- Undefined Instruction, Software interrupt

由上可见，**Reset** 入口，即为整个程序的实际入口点。因此，我们在编写代码的时候，第一条语句是在**0x00000000** 处开始执行的。一般地，我们使用下面的代码：

```

        area RESET,code,readonly
        entry
        b      Reset_Handler
Undefined_Handler
        b      Undefined_Handler
SWI_Handler
        b      SWI_Handler
Prefetch_Handler
        b      Prefetch_Handler
Abort_Handler
        b      Abort_Handler
        nop
IRQ_Handler
        b      IRQ_Handler
FIQ_Handler
        b      FIQ_Handler
Reset_Handler
        ldr sp, =0x0C002000
        .
        .
        .

```

2. 分散加载文件

Scatter file（分散加载描述文件）用于 **LARM** 链接器的输入参数，它指定映像文件内部各区域的 **download** 与运行时位置。**LARM** 将会根据 **scatter file** 生成一些区域相关的符号，它们是全局的供用户建立运行时环境时使用。通过这个文件可以指定程序的入口地址。在利用 **MDK** 进行实际应用程序开发时，常常需要使用道分散加载文件，例如以下情况：

存在复杂的地址映射：例如代码和数据需要分开放在在多个区域。

存在多种存储器类型：例如包含 **Flash**、**ROM**、**SDRAM**、快速 **SRAM**。需要根据代码与数据的特性把他们放在不同的存储器中，比如中断处理部分放在快速 **SRAM** 内部来提高响应速度，而把不常用到的代码放到速度比较慢的 **Flash** 内。

函数的地址固定定位：可以利用 **Scatter file** 实现把某个函数放在固定地址，而不管其应用程序是否已经改变或重新编译。

利用符号确定堆与堆栈：

内存映射的 IO：采用 **scatter file** 可以实现把某个数据段放在精确的地址处。

因此对于实际的嵌入式系统来说 **scatter file** 是必不可少的，因为嵌入式系统通常采用了 ROM，RAM，和内存映射的 IO。关于 **Scatter file** 的相关知识非常多，详细内容可以参考 MDK 所带的帮助，下面给出一个简单实例。

```

LOAD_ROM 0x0000 0x8000
{
    EXEC_ROM 0x0000 0x8000
    {
        *(+RO)
    }
    RAM 0x10000 0x6000
    {
        *(+RW, +ZI)
    }
}

```

这个分散加载描述文件对应的分散加载映像如图 3-12 所示，文件中各项内容的含义分别是：

```

LOAD_ROM(下载区域名称) 0x0000(下载区域起始地址) 0x8000(下载区域最大字节数)
{
    EXEC_ROM(第一执行区域名称) 0x0000(第一执行区域起始地址) 0x8000(第一执行区域最大字节数)
    {
        *(+RO(代码与只读数据))
    }
    RAM(第二执行区域名称) 0x10000(第二执行区域起始地址) 0x6000(第二执行区域最大字节数)
    {
        *(+RW(读写变量), +ZI(未初始化变量))
    }
}

```

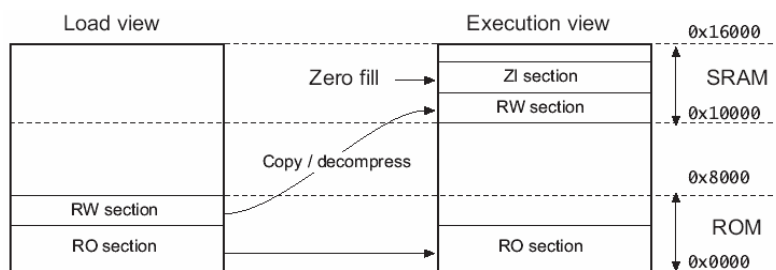


图 3-12 分散加载映像图

3. 内嵌汇编语言

编译 C 时，可以通过 `__asm` 汇编程序说明符调用内嵌汇编程序。说明符后面跟随有一列包含在大括号中的汇编程序指令。例如：

```

__asm
{
    instruction [; instruction]
}

```



```
...
[instruction]
}
```

如果两条指令在同一行中，必须用分号将其分隔。如果一条指令占用多行，必须用反斜线符号（\）指定续行。可在内嵌汇编语言块内的任意位置处使用 C 注释。可在任何可以使用 C 语句的地方使用 `_asm` 语句。

3.6.5 实验操作步骤

1) 参考 3.1.5 小节实验的操作步骤建立一个新的工程，命名为 **CTest2**，注意在建立工程的过程中添加设备数据库中 **S3C2410** 芯片自带的启动代码，也可手动添加启动代码 **startup.s**;

2) 参考 3.1.5 小节实验的步骤和实验参考程序编辑输入源代码，保存文件为 **CCode.c**;

3) 在 **Project workspace** 工作区中右击 **target1->Source Group 1**，在弹出菜单中选择 **"Add file to Group 'Source Group 1' "**，在随后弹出的文件选择对话框中，选择刚才建立的源文件 **CCode.c**;

4) 在 **Option for Target** 对话框 **Linker** 页 **Scatter File** 对话框中添加分散加载描述文件 **CTest2.sct**，文件内容参考 3.3.6 小节。

5) 选择菜单项 **Project ->Build target** 或快捷键 **F7**，生成目标代码;

6) 选择菜单项 **Debug ->Start/Stop Debug Session** 或快捷键 **Ctrl+F5**，即可进入调试模式。这里使用的是 **μVision3 IDE** 中的软件仿真器;

7) 选择菜单项 **Debug ->run** 或 **F5**，即可运行代码;

8) 打开 **memory**、**register**、**watch**、**Local** 窗口，单步执行，并通过 **memory/register/watch/variable** 窗口分析判断结果。注意观察程序如何从跳转进主程序 **__main**，在 **call stack** 窗口观察当前执行函数之间的调用。在 **watch** 框中输入要观察变量 **I** 的值，并记录下来。特别注意在 **local** 窗口观察变量 **I** 的变化并记录下来。

9) 结合实验内容和相关资料，观察程序运行;

10) 理解和掌握实验后，完成实验练习题。

3.6.6 实验参考程序

CCode.c 的源代码

```

/*****
* File:      c2.c
* Author:    WUHAN R&D Center, embest
* Desc:      c language example 2
* History:   XU Liang Ping
*****/
/*****
* name:      _nop_
* func:      The following example embed assemble language
* para:      none
* ret:       none
* modify:
* comment:
*****/
void _nop_()
{

```

```

    int temp=0;
    __asm
    {
        mov temp,temp
    }
}

/*****
* name:      delay
* func:      delay time
* para:      none
* ret:       none
* modify:
* comment:
*****/
void delay(void)
{
    int i=0;

    for(; i <= 10; i++)
    {
        __nop_();
    }
}

/*****
* name:      delay10
* func:      delay time
* para:      none
* ret:       none
* modify:
* comment:
*****/
void delay10(void)
{
    int i;

    for(i = 0; i <= 10; i++)
    {
        delay();
    }
}

/*****
* name:      _main
* func:      c code entry
* para:      none
* ret:       none
* modify:
* comment:
*****/

```

```

__main()
{
    // int i = 5;

    for( ; ; )
    {
        delay10();
    }
}

```

Startup.s 的源代码

```

;#*****
;# File: startup.s
;# Author: Wuhan R&D Center, embest
;# Desc: C start up codes;#
;#*****
;/*-----*/
;/*          global symbol define          */
;/*-----*/
; .global _start

;/*-----*/
;/*          code          */
;/*-----*/
RESET,code,readonly
    entry
;# Set interrupt / exception vectors
    b    Reset_Handler
Undefined_Handler
    b    Undefined_Handler
SWI_Handler
    b    SWI_Handler
Prefetch_Handler
    b    Prefetch_Handler
Abort_Handler
    b    Abort_Handler
    nop                                ;/* Reserved vector */
IRQ_Handler
    b    IRQ_Handler
FIQ_Handler
    b    FIQ_Handler
Reset_Handler
; ldr sp, =0x0C002000

;# *****
;# Branch on C code Main function (with interworking)      *

;# Branch must be performed by an interworking call as      *
;# either an ARM or Thumb.main C function must be          *
;# supported. This makes the code not position-independant.*
;# A Branch with link would generate errors                  *
;# *****

```

```

        IMPORT __main
        LDR     R0, =__main
        BX      R0
;   # jump to __main()

;# *****
;# * Loop for ever                               *
;# * End of application. Normally, never occur.   *
;# * Could jump on Software Reset ( B 0x0 ).      *
;# *****
End
        b      End
    end

```

CTest2.sct 的源代码

```

; *****
; *** Scatter-Loading Description File generated by uVision ***
; *****

LR_ROM1 0x30000000      {      ; load region
    ER_ROM1 0x30000000 0x00200000 { ; load address = execution address
        *.o (RESET, +First)
;    *(InRoot$$Sections)
        .ANY (+RO)
    }
    RW_RAM1 0x30300000 0x03D00000 { ; RW data
        .ANY (+RW +ZI)
    }
}

```

调试命令脚本文件可使用3.1节的。

3.6.7 练习题

1. 改进 C 语言使用实验一中的练习题，在 C 语言文件中定义全局及局部变量，并在编译链接时使用链接脚本文件，使用 Tools 菜单下的 Disassemble all 产生 objdump 文件，从该文件中观察代码及变量在目标输出代码中的存放情况。

2. 在以上实验例程 C 语言文件中，加入嵌入汇编语言，使用汇编指令实现读写某存储单元的值，初步掌握嵌入汇编语言的使用。

3.7 汇编与 C 语言相互调用实例

3.7.1 实验目的

阅读 S3C2410 启动代码，观察处理器启动过程；

学会使用 MDK 集成开发环境辅助窗口来分析判断调试过程和结果；

学会在 MDK 集成开发环境中编写、编译与调试汇编和 C 语言相互调用的程序。

3.7.2 实验设备

- 硬件：PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

3.7.3 实验内容

使用汇编完成一个随机数产生函数，通过 C 语言调用该函数，产生一系列随机数，存放数组里面。

3.3.4 实验原理

ARM过程调用ATPCS (ARM)

ATPCS是一系列用于规定应用程序之间相互调用的基本规则，这此规则包括：

支持数据栈限制检查；

支持只读段位制无关（ROPI）；

支持可读/写段位置无关（RWPI）；

支持 ARM 程序和 Thumb 程序的混合使用；

处理浮点运算。

使用以上规定的 ATPCS 规则时，应用程序必须遵守如下：

程序编写遵守 ATPCS；

变量传递以中间寄存器和数据栈完成；

汇编器使用-apcs 开关选项。

关于其他ATPCS规则，用户可以参考ARM处理器相关书籍或登录ARM公司网站。

程序只要遵守ATPCS相应规则，就可以使用不同的源代码编写程序。程序间的相互调用最主要的是解决参数传递问题。应用程序之间使用中间寄存器及数据栈来传递参数，其中，第一个到第四个参数使用R0-R3，多于四个参数的使用数据栈进行传递。这样，接收参数的应用程序必须知道参数的个数。

但是，在应用程序被调用时，一般无从知道所传递参数的个数。不同语言编写的应用程序在调用时可以自定义参数传递的约定，使用具有一定意义的形式来传递，可以很好地解决参数个数的问题。常用的方法是把第一个或最后一个参数作为参数个数（包括个数本身）传递给应用程序。

ATPCS中寄存器的对应关系如表3-5所列：

表 3-5 ATPCS 规则中寄存器列表

ARM 寄存器		ATPCS 别名	ATPCS 寄存器说明
4	R0-R3 <==>	a1-a4	参数/结果/scratch 寄存器 1-
	R4 <==>	v1	局部变量寄存器 1
	R5 <==>	v2	局部变量寄存器 2
	R6 <==>	v3	局部变量寄存器 3
5	R7 <==>	v4、wr	局部变量寄存器 4 Thumb 状态工作寄存器
	R8 <==>	v5	ARM 状态局部变量寄存器

R9	<==>	v6、sb	ARM 状态局部变量寄存器 6 RWPI 的静态基址寄存器
R10	<==>	v7、sl	ARM 状态局部变量寄存器 7 数据栈限制指针寄存器
R11	<==>	v8	ARM 状态局部变量寄存器 8
R12	<==>	ip	子程序内部调用的临时（scratch）寄存器
R13	<==>	sp	数据栈指针寄存器
R14	<==>	lr	链接寄存器
R15	<==>	PC	程序计数器

3.7.5 实验操作步骤

- 1) 参考 3.1.5 小节实验的操作步骤建立一个新的工程，命名为 **explasm**，注意在建立工程的过程中添加设备数据库中 **S3C2410** 芯片自带的启动代码，也可手动添加启动代码 **startup.s**；
- 2) 参考 3.1.5 小节实验的步骤和实验参考程序编辑输入源代码，保存文件为 **randtest.c** 和 **random.s**；
- 3) 在 **Project workspace** 工作区中右击 **target1->Source Group 1**，在弹出菜单中选择“Add file to Group 'Source Group 1'”，在随后弹出的文件选择对话框中，选择刚才建立的源文件 **randtest.c** 和 **random.s**；
- 4) 在 **Option for Target** 对话框 **Linker** 页 **Scatter File** 对话框中添加分散加载描述文件 **explasm.sct**，文件内容与 3.3.6 小节 **CTest2.sct** 相同。
- 5) 选择菜单项 **Project ->Build target** 或快捷键 **F7**，生成目标代码；
- 6) 选择菜单项 **Debug ->Start/Stop Debug Session** 或快捷键 **Ctrl+F5**，即可进入调试模式。这里使用的是 **µVision3 IDE** 中的软件仿真器；
- 7) 选择菜单项 **Debug ->run** 或 **F5**，即可运行代码；
- 8) 打开 **memory**、**register**、**watch**、**Local** 窗口，单步执行，并通过 **memory**、**register**、**watch**、**variable** 窗口分析判断结果。注意观察程序如何从跳转进主程序 **__main**，在 **call stack** 窗口观察当前执行函数之间的调用。
- 9) 结合实验内容和相关资料，观察程序运行；
- 10) 理解和掌握实验后，完成实验练习题。

3.7.6 实验参考程序

(1) randtest.c 参考源代码：

```

/*****
* File:      randtest.c
* Author:    Wuhan R&D Center, embest
* Desc:      Random number generator demo program
*           Calls assembler function 'randomnumber' defined in random.s
* History:
*****/

```

```

/*-----*/
/*
                                extern function
*/
/*-----*/
extern unsigned int randomnumber( void );

/*****
* name:          main
* func:          c code entry
* para:          none
* ret:           none
* modify:
* comment:
*****/

main()
{
    unsigned int i,nTemp;
    unsigned int unRandom[10];

    for( i = 0; i < 10; i++ )
    {
        nTemp = randomnumber();
        unRandom[i] = nTemp;
    }

    return(0);
}

```

(2) random.s 参考源代码:

```

;#*****
;# File:    random.s
;# Author: embest
;# Desc:    Random number generator
*
;# This uses a 33-bit feedback shift register to generate a pseudo-randomly *
;# ordered sequence of numbers which repeats in a cycle of length 2^33 - 1 *
;# NOTE: randomseed should not be set to 0, otherwise a zero will be generated *
;# continuously (not particularly random!).
;# This is a good application of direct ARM assembler, because the 33-bit *
;# shift register can be implemented using RRX (which uses reg + carry). *
;# An ANSI C version would be less efficient as the compiler would not use RRX.*
;# AREA    |Random$$code|, CODE, READONLY
;# History:
;#*****

/*-----*/
/*
                                global symbol define
*/
/*-----*/
EXPORT randomnumber

```

```

EXPORT  seed

; /*----- */
; /*          code                               */
; /*----- */

AREA  RAND, CODE, READONLY
randomnumber
;# on exit:
;# a1 = low 32-bits of pseudo-random number
;# a2 = high bit (if you want to know it)
    ldr    ip, seedpointer
    ldmia  ip, {a1, a2}
    tst    a2, a2, lsr#1          ;/* to bit into carry */
    movs   a3, a1, rrx            ;/* 33-bit rotate right */
    adc    a2, a2, a2             ;/* carry into LSB of a2 */
    eor    a3, a3, a1, lsl#12     ;/* (involved!) */
    eor    a1, a3, a3, lsr#20     ;/* (similarly involved!)*
    stmia  ip, {a1, a2}
    mov    pc, lr

seedpointer
    DCD    seed
seed      DCD    0x55555555
          DCD    0x55555555

END

```

3.7.7 练习题

参考3.3.6小节中“实验A参考源代码”，改进3.6节“C语言程序实验2”的练习题例程，使用嵌入汇编语言实现 $R1+R2=R0$ 的加法运算，运算结果保存在 $R0$ 。调试时打开Register窗口，观察嵌入汇编语句运行前后 $R0$ 、 $R1$ 、 $R2$ 、 SP 寄存器以及ATPCS寄存器对应的ARM寄存器内容的变化。

3.8 综合实验

3.8.1 实验目的

掌握处理器启动配置过程；

掌握使用 μ Vision IDE 辅助信息窗口来分析判断调试过程和结果，学会查找软件调试时的故障或错误；

掌握使用 μ Vision IDE 开发工具进行软件开发与调试的常用技巧。

3.8.2 实验设备

- 硬件：PC 机。
- 软件： μ Vision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

3.8.3 实验内容

完成一个完整的工程，要求包含启动代码，汇编函数和 C 文件，而且 C 文件包含 ARM 函数和 Thumb 函数，并可以相互调用。

3.8.4 实验原理

1. μ Vision IDE 开发调试辅助窗口

使用 μ Vision IDE 嵌入式开发环境，用户可以使用源代码编辑窗口编写源程序文件、使用反汇编窗口观察程序代码的执行、使用 Register 窗口观察程序操作及 CPU 状态、使用外围寄存器窗口观察当前处理器的配置、使用 Memory 窗口观察内存单元使用情况、使用 Watch 或 Variables 窗口观察程序变量、使用操作控制台执行特殊命令、使用性能分析仪窗口分析代码的性能、使用逻辑分析仪模拟处理器及其外设的时序、使用串行端口窗口模拟串口的工作，加上调试状态下丰富的右键菜单功能，用户可以使用 μ Vision IDE 实现或发现任何一部分应用软件、修改任何一个开发或运行时的错误。

结合 μ Vision IDE 随安装软件自带的丰富的样例程序，配合高性能的 Ulink2 仿真器，可以使用户把主要精力放在项目软件开发上。

2. 生成 HEX 文件

在 μ Vision 3 IDE 中选择菜单页 Project-Options for Target-Output，在弹出的对话框中，选择 Create HEX File 单选项即可；

单击“Select Folder for Objects...”按钮，为生成的十六进制文件选择存储路径；

在“Name of Executable”文本框中输入生成十六进制文件的名字；

选择 Create HEX File，并单击“确认”按钮；

重新编译该工程文件就可以得到十六进制文件。

3.8.5 实验操作步骤

1. 打开 Keil uVision3，在菜单中选择“Project->New->uVision project”创建新的工程，命名为 interwork，CPU 选择为 Samsung 的 S3C2410A，接着会弹出如图 3-6 所示的对话框，选择“否”，则不会将 Samsung S3C2410A 的启动代码拷贝到新工程。

2. 参考 3.8.6 的参考程序创建 C 语言程序 thumb.c、arm.c 和汇编语言程序 entry.s、random.s，并将这些程序添加到新建的工程中。

3. 在菜单中选择“Project->Options for Target 'Target 1'”则会弹出如图 3-8 所示的工程配置界面，依据处理器及目标板的实际配置对工程进行配置。

4. 用鼠标右键点击“Project Workspace”中的文件“thumb.c”，如图 3-12 所示，选择“Options for File 'thumb.c'”，则出现文件“thumb.c”的配置对话框，在该对话框中选择“C/C++”标签，如图 3-13 所示，使“Thumb Mode”有效。“Thumb Mode”有效时在配置对话框下面的“Compiler control string”右边的框中会有“--thumb”选项。工程中的其它文件采用默认配置。

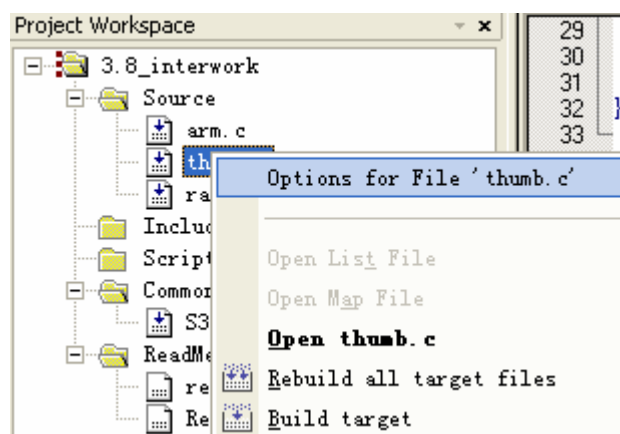


图 3-12 打开文件配置对话框

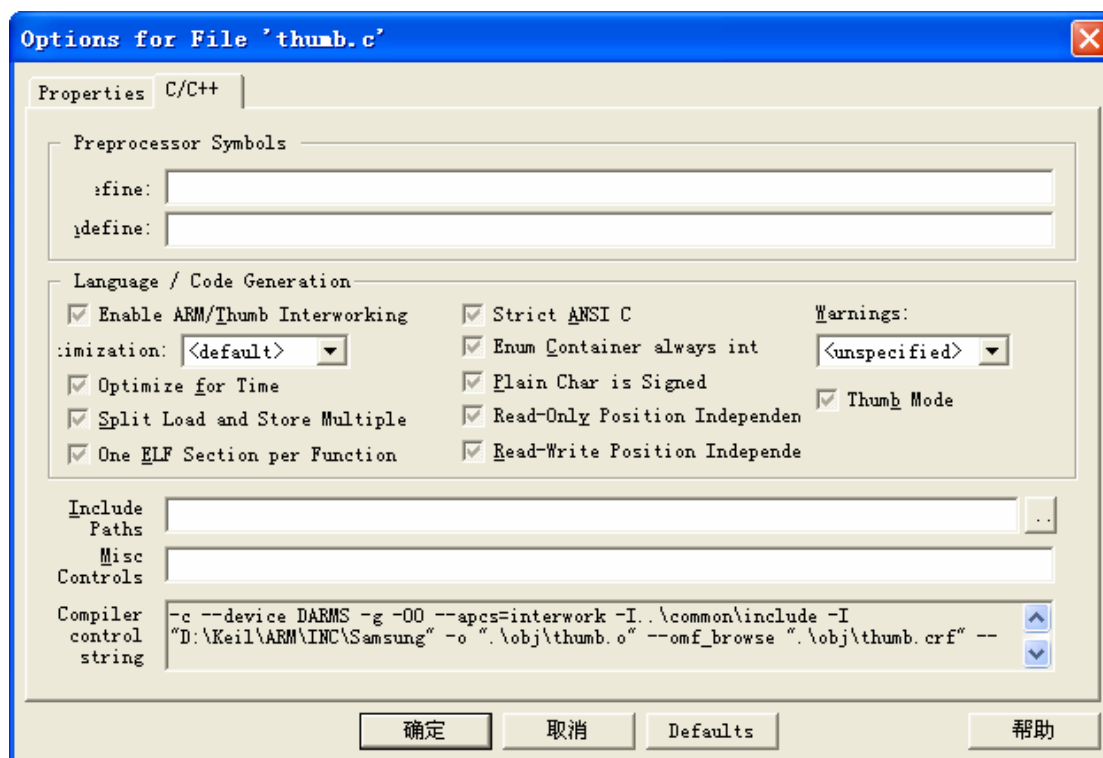


图 3-13 文件配置选项

5. 对工程进行编译、调试，使工程能够正确执行。
6. 在菜单中选择“Debug->Start/Stop Debug Session”或使用快捷键“Ctrl + F5”进入调试状态，单步执行程序，通过寄存器窗口、存储器窗口等分析判断运行结果。
7. 单步跟踪 ARM 函数与 Thumb 函数相互调用的反汇编代码，分析 arm 内核的状态切换过程；
8. 理解和掌握实验后，参考样例程序完成实验练习题。

3.8.6 参考程序

1. arm.c 参考程序：

```

/*****
* File:   arm.c
* Author:   WuHan R&D Center, Embest
* Desc:   arm instruction ,c program
* History:
*****/

/*-----*/
/*   extern variable           */
/*-----*/
extern char szArm[20];
extern int randomnumber(void);

/*****
* name:      delaya
* func:      delay function of arm instructon

```

```

* para:      nTime---input
* ret:       none
* modify:
* comment:
*****/
static void delaya(int nTime)
{
    int i, j, k;
    k = 0;

    for(i = 0; i < nTime; i++)
    {
        for(j = 0; j < 10; j++)
            k++;
    }
}

/*****
* name:      arm_function
* func:      example
* para:      none
* ret:       none
* modify:
* comment:
*****/
void arm_function(void)
{
    int i;
    int nLoop;
    unsigned int unRandom;
    char *p = "Hello from ARM world";

    for(i = 0; i < 20; i++)
        szArm[i] = (*p++);

    delaya(2);

    for( nLoop = 0; nLoop < 10; nLoop++ )
    {
        unRandom = randomnumber();
    }
}

```

2. s3c2410A.s 参考程序:

```

;*****
;NAME:      entry.s

```

```

*

;Author:      Wuhan R&D Center, Embest
              *
;Desc:        debugging with arm and thumb instruction
              *
;History:
              *
;
              *
;*****
;*/----- */
;*/                                constant define
              */
;*/----- */
count          EQU      20

;*/----- */
;*/                                extern function
*/

;*/----- */
;*/                                global symbol define
              */
;*/----- */

;*/----- */
;*/                                code
              */
;*/----- */

    PRESERVE8
    area RESET, code, readonly
    arm
    entry
    code32

Reset_Handler
    export Reset_Handler
    b      Reset_Handler
Undefined_Handler
    b      Undefined_Handler
SWI_Handler
    b      SWI_Handler
Prefetch_handler

```

```

b      Prefetch_handler
Abort_Handler
b      Abort_Handler
nop                                /*      Reserved      vector      */

IRQ_Handler
b      IRQ_Handler
FIQ_Handler
b      FIQ_Handler

Reset_Handle

ldr     sp, =0x34000000            /* set up user_mode stack pointer (r13) */
mov     r0, #count                /* the number of loop*/
mov     r1, #0
mov     r2, #0
mov     r3, #0
mov     r4, #0
mov     r5, #0
mov     r6, #0

loop0
add     r1, r1, #1
add     r2, r2, #1
add     r3, r3, #1
add     r4, r4, #1
add     r5, r5, #1
add     r6, r6, #1

subs    r0, r0, #1
bne     loop0

adr     r0, Thumb_Entry+1         /*jump to thumb program */
bx      r0
nop
nop

;# *****
;# *   thumb program entry           *
;# *****

code16
Thumb_Entry
mov     r0, #count

mov     r1, #0

```

```

mov     r2, #0
mov     r3, #0
mov     r4, #0
mov     r5, #0
mov     r6, #0
mov     r7, #0

loop1
add     r1, #1
add     r2, #1
add     r3, #1
add     r4, #1
add     r5, #1
add     r6, #1
add     r7, #1

sub     r0, #1
bne     loop1

import thumb_function
bl      thumb_function

b       Thumb_Entry

END

```

3. random.s 参考程序:

```

;#*****
;# Filef°    random.s                                *
;# Author:   embest                                  *
;# Descf°    Random                                number                generator
*
;#          This uses a 33-bit feedback shift register to generate a pseudo-randomly
*
;#          ordered sequence of numbers which repeats in a cycle of length 2^33 - 1
*
;#          NOTE: randomseed should not be set to 0, otherwise a zero will be generated
*
;#          continuously (not particularly random!).
*
;#          This is a good application of direct ARM assembler, because the 33-bit
*
;#          shift register can be implemented using RRX (which uses reg + carry).
*
;#          An ANSI C version would be less efficient as the compiler would not use RRX.

```

```

*
;#                               AREA                |Random$$code|,   CODE,   READONLY
*

;# History:
    *
;#*****

;/*----- */
;/*                               global symbol define
    */
;/*----- */
global randomnumber
global seed
code32
;/*----- */
;/*                               code
    */
;/*----- */
area tcode , code , readonly

EXPORT randomnumber
randomnumber
ldr    ip, seedpointer
ldmia  ip, {a1, a2}
tst    a2, a2, lsr #1           ;/* to bit into carry*/
movs   a3, a1, rrx              ;/* 33-bit rotate right */
adc    a2, a2, a2               ;/* carry into LSB of a2 */
eor    a3, a3, a1, lsl #12      ;/* (involved!) */
eor    a1, a3, a3, lsr #20      ;/* (similarly involved!)*
stmia  ip, {a1, a2}
BX lr

seedpointer
DCD    seed

;   area tdata, data, readwrite
seed
DCD    0x55555555
DCD    0x55555555

END

```

4. thumb.c 参考程序:

```

/*****
* File:  thumb.c
* Author:   Wuhan R&D Center, Embest
* Desc:    c program of Thumb instruction

```

```

* History:
***** /

/*-----*/
/*   global variables                               */
/*-----*/

char szArm[22];
char szThumb[22];
unsigned long ulTemp = 0;

/*-----*/
/*   extern function                               */
/*-----*/
extern void arm_function(void);

/*****
* name:      delayt
* func:      delay functin with Thumb instruction
* para:      nTime --- input
* ret:       none
* modify:
* comment:
*****/

static void delayt(int nTime)
{
    int i, j, k;

    k = 0;
    for(i = 0; i < nTime; i++)
    {
        for(j = 0; j < 10; j++)
            k++;
    }
}

/*****
* name:      thumb_function
* func:      The following example use Thumb instruction
* para:      none
* ret:       none
* modify:
* comment:
*****/

int thumb_function(void)
{
    int i;

```



```
char * p = "Hello from Thumb World";

ulTemp++;
arm_function();

    delayt(2);

for(i = 0; i < 22; i++)
    szThumb[i] = (*p++);

return 0;
}
```

3.8.7 练习题

1. 阅读 Keil uVision3 安装目录下的启动文件 S3C2410A.s，尽量理解每一条语句的功能；
2. 编写一个汇编文件和一个 C 语言文件，实现从汇编语言中传递简单数学运算参数给由 C 语言程序编写的简单数学运算函数，并从 C 语言程序中返回运算结果；新建工程名为 smath 的工程，同时加入 S3C2410A.s 文件，参照基础实验工程配置，对新建工程进行配置；编译、调试新工程；并在软件模拟器中下载到 0x00000000 处执行，跟踪程序的执行。

第四章 基本接口实验

4.1 存储器实验

4.1.1 实验目的

- 通过实验熟悉 ARM 的内部存储空间分配；
- 掌握对存储区配置方法；
- 掌握对存储区进行读写访问的方法。

4.1.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

4.1.3 实验内容

- 熟练使用命令脚本文件对 ARM 存储控制寄存器进行正确配置；
- 使用 C 语言编程，实现对 RAM 的读写访问。

4.1.4 实验原理

1. 存储控制器

S3C2410 处理器的存储控制器可以为片外存储器访问提供必要的控制信号，它主要包括以下特点：

- 支持大、小端模式（通过软件选择）
- 地址空间：包含 8 个地址空间，每个地址空间的大小为 128M 字节，总共有 1G 字节的地址空间。
- 除 BANK0 以外的所有地址空间都可以通过编程设置为 8 位、16 位或 32 位对准访问。BANK0 可以设置为 16 位、32 位访问。
- 8 个地址空间中，6 个地址空间可以用于 ROM、SRAM 等存储器，2 个用于 ROM、SRAM、SDRAM 等存储器。
- 7 个地址空间的起始地址及空间大小是固定的。
- 1 个地址空间的起始地址和空间大小是可变的。
- 所有存储器空间的访问周期都可以通过编程配置。
- 提供外部扩展总线的等待周期。
- SDRAM 支持自动刷新和掉电模式。

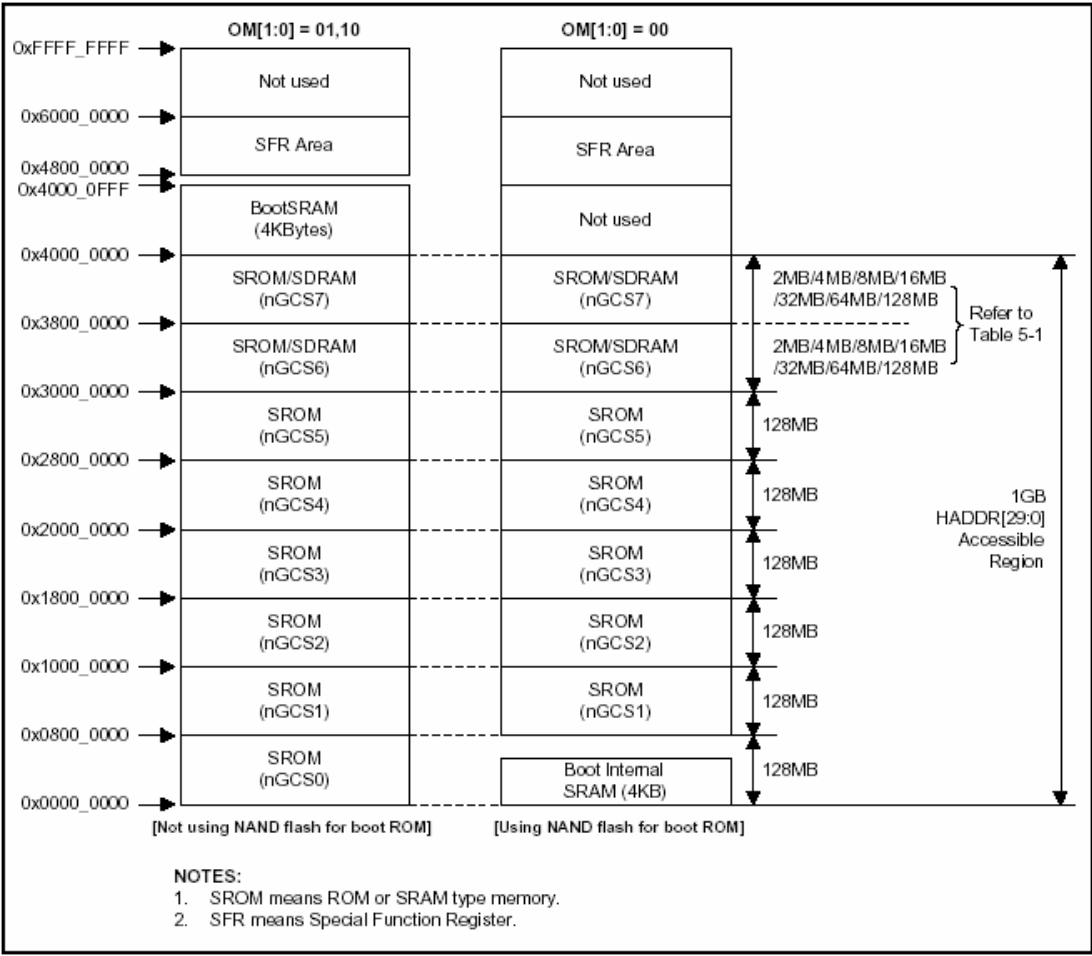


图 4-1 S3C2410 复位后的存储器地址分配

注意: BANK6/BANK7 的空间大小必须相同。

图 4-1 为 S3C2410 复位后的存储器地址分配图。从图中可以看出，特殊功能寄存器位于 0X48000000 到 0X60000000 的空间内。Bank0-Bank5 的起始地址和空间大小都是固定的，Bank6 的起始地址是固定的，但是空间大小和 Bank7 一样是可变的，可以配置为 2/4/8/16/32M/64M/128M。Bank6 和 Bank7 的详细地址和空间大小的关系可以参考表 4-1。

表 4-1 Bank6/Bank7 地址

Address	2MB	4MB	8MB	16MB	32MB
Bank 6					
Start address	0xc00_0000	0xc00_0000	0xc00_0000	0xc00_0000	0xc00_0000
End address	0xc1f_fff	0xc3f_fff	0xc7f_fff	0xcff_fff	0xdff_fff
Bank 7					
Start address	0xc20_0000	0xc40_0000	0xc80_0000	0xd00_0000	0xe00_0000
End address	0xc3f_fff	0xc7f_fff	0xcff_fff	0xdff_fff	0xfff_fff

BANK0 总线宽度

BANK0 (nGCS0) 的数据总线宽度可以配置为 16 位或 32 位。因为 BANK0 为启动 ROM（映射地址为 0X00000000）所在的空间，所以必须在第一次访问 ROM 前设置 BANK0 数据宽度，该数据宽度是由复位后 OM[1:0]的逻辑电平决定的，表 4-2 所示。

表 4-2 数据宽度选择

OM1 (Operating Mode 1)	OM0 (Operating Mode 0)	Booting ROM Data width
0	0	Nand Flash Mode
0	1	16-bit
1	0	32-bit
1	1	Test Mode

表 4-3 存储器（SROM/SDRAM）地址引脚连接

MEMORY ADDR. PIN	S3C2410X ADDR. @ 8-bit DATA BUS	S3C2410X ADDR. @ 16-bit DATA BUS	S3C2410X ADDR. @ 32-bit DATA BUS
A0	A0	A1	A2
A1	A1	A2	A3
...

SDRAM 空间地址引脚连接

表 4-4 SDRAM 空间地址配置

Bank Size	Bus Width	Base Component	Memory Configuration	Bank Address
2MB	x8	16Mb	(1M x 8 x 2banks) x 1 ea	A20
	x16		(512K x 16 x 2banks) x 1 ea	
4MB	x8	16Mb	(2M x 4 x 2banks) x 2 ea	A21
	x16		(1M x 8 x 2banks) x 2 ea	
	x32		(512K x 16 x 2banks) x 2 ea	
8MB	x16	16Mb	(2M x 4 x 2banks) x 4 ea	A22
	x32		(1M x 8 x 2banks) x 4 ea	
	x8	64Mb	(4M x 8 x 2banks) x 1 ea	A[22:21]
	x8		(2M x 8 x 4banks) x 1 ea	
	x16		(2M x 16 x 2banks) x 1 ea	A22
	x16		(1M x 16 x 4banks) x 1 ea	A[22:21]
	x32		(512K x 32 x 4banks) x 1 ea	
	16MB	16Mb	(2M x 4 x 2banks) x 8 ea	A23
		64Mb	(8M x 4 x 2banks) x 2 ea	A[23:22]
			(4M x 4 x 4banks) x 2 ea	
			(4M x 8 x 2banks) x 2 ea	A23
			(2M x 8 x 4banks) x 2 ea	A[23:22]
			(2M x 16 x 2banks) x 2 ea	A23
		128Mb	(1M x 16 x 4banks) x 2 ea	A[23:22]
			(4M x 8 x 4banks) x 1 ea	
			(2M x 16 x 4banks) x 1 ea	
			(2M x 16 x 4banks) x 1 ea	
32MB	x16	64Mb	(8M x 4 x 2banks) x 4 ea	A24
	x16		(4M x 4 x 4banks) x 4 ea	A[24:23]
	x32		(4M x 8 x 2banks) x 4 ea	A24
	x32		(2M x 8 x 4banks) x 4 ea	A[24:23]
	x16	128Mb	(4M x 8 x 4banks) x 2 ea	
	x32		(2M x 16 x 4banks) x 2 ea	
	x8	256Mb	(8M x 8 x 4banks) x 1 ea	
	x16		(4M x 16 x 4banks) x 1 ea	
64MB	x32	128Mb	(4M x 8 x 4banks) x 4 ea	A[25:24]
	x16	256Mb	(8M x 8 x 4banks) x 2 ea	
	x32		(4M x 16 x 4banks) x 2 ea	
	x8	512Mb	(16M x 8 x 4banks) x 1 ea	
128MB	x32	256Mb	(8M x 8 x 4banks) x 4 ea	A[26:25]
	x8	512Mb	(32M x 4 x 4banks) x 2 ea	
	x16		(16M x 8 x 4banks) x 2 ea	
	x32		(8M x 16 x 4banks) x 2 ea	

nWAIT 引脚功能

如果和每个地址空间相关联的 WAIT 被允许，某个地址空间处于激活状态的时候应该通过外部 nWAIT 引脚来延长 nOE 持续时间。从 $t_{acc}-1$ 核对 nWAIT，在采样 nWAIT 为高电平的后一个时钟周期使 nOE 变为高电平，nWE 信号和 nOE 信号相同。

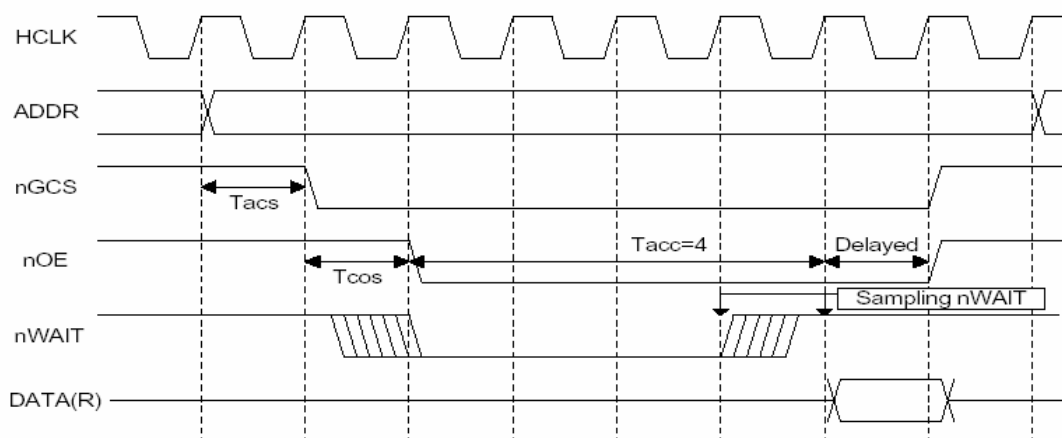


图 4-2 S3C2410X 的外部 nWAIT 时序图 ($t_{acc}=4$)

nXBREQ/nXBACK 引脚操作

如果 nXBREQ 被允许，处理器会在 nXBACK 引脚输出低电平作为应答信号，如果 nXBACK 引脚输出低电平，地址/数据总线和存储器控制信号会处于高阻状态，如图 4-3 所示。如果 nXBREQ 没有被允许，nXBACK 也无效。

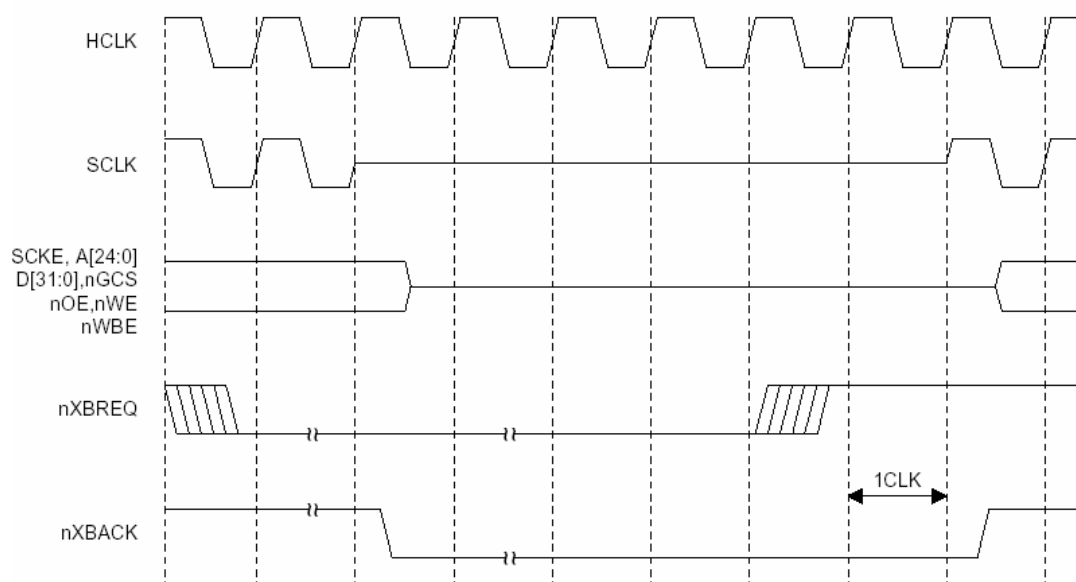


图 4-3 S3C2410X 的 nXBREQ/nXBACK 时序表

ROM 接口举例：

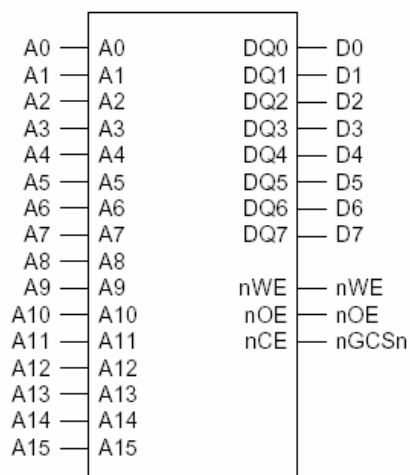


图 4-4 存储器与 8bitROM 接口

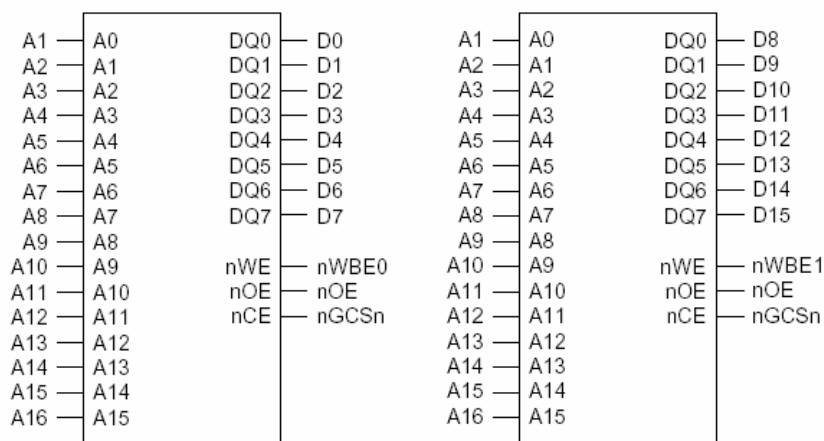


图 4-5 存储器与 8bit ROM×2 接口

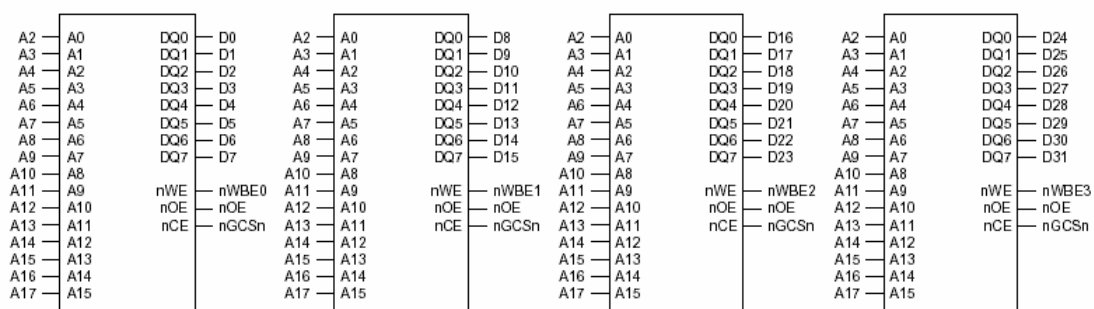


图 4-6 存储器与 8bit ROM*4 接口

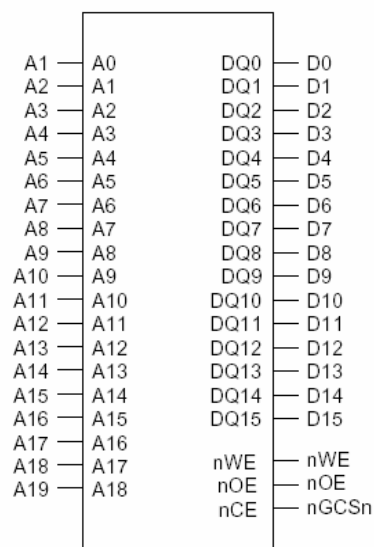


图 4-7 存储器与 16bit ROM 接口

SRAM 存储器接口举例：

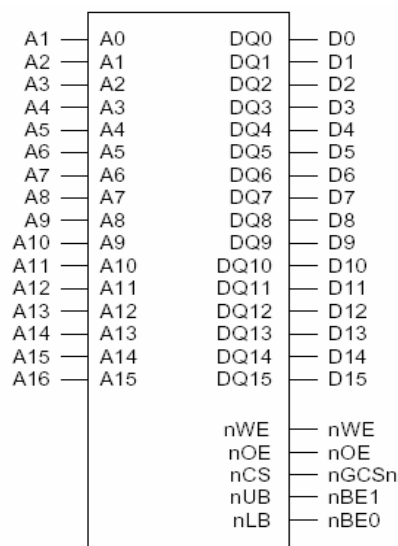


图 4-8 存储器与 16bit SRAM 接口

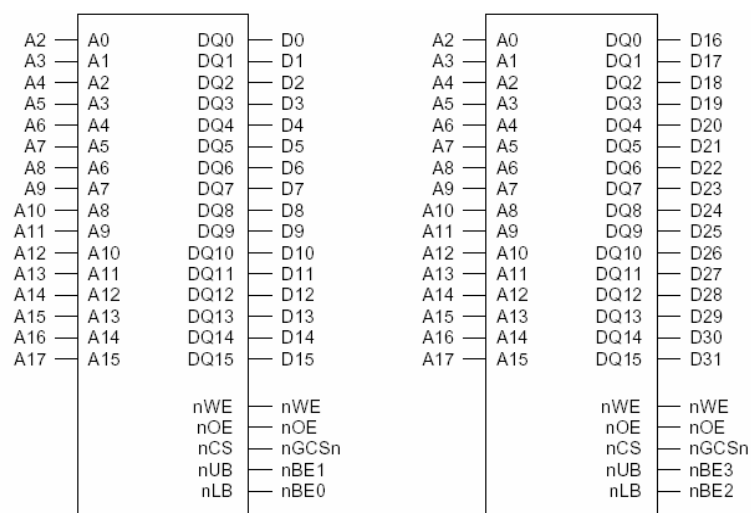


图 4-9 存储器与 16bit SRAM×2 接口

SDRAM 接口举例：

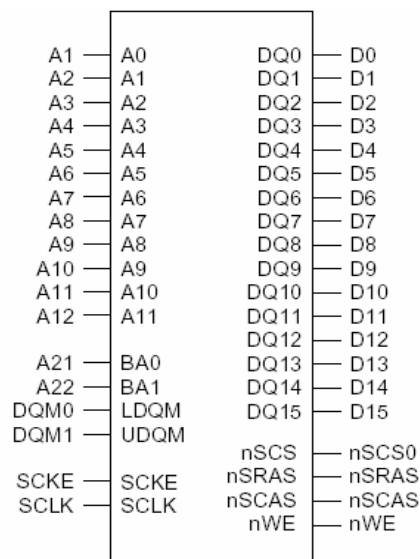


图 4-10 存储器与 16bit SDRAM 接口（8MB：1Mb×16×4BANKS）

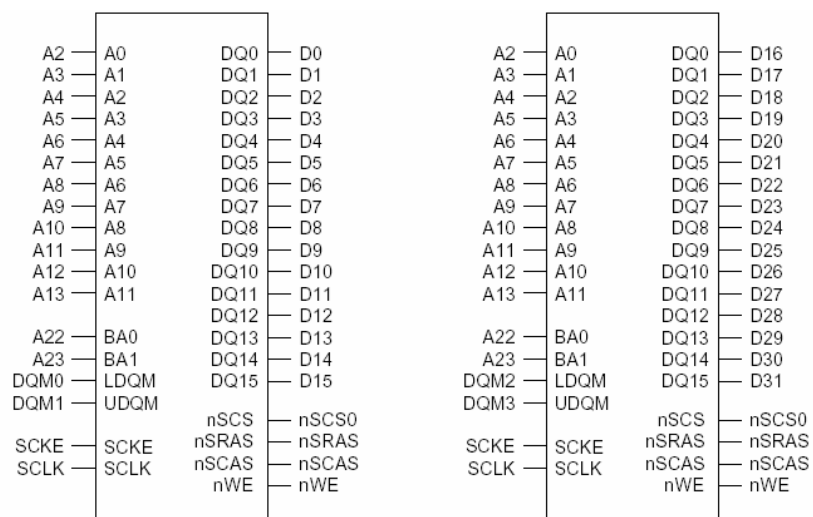


图 4-11 存储器与 16bit SDRAM 接口 (16MB: 1Mb×16×4banks×2)

可编程访问周期

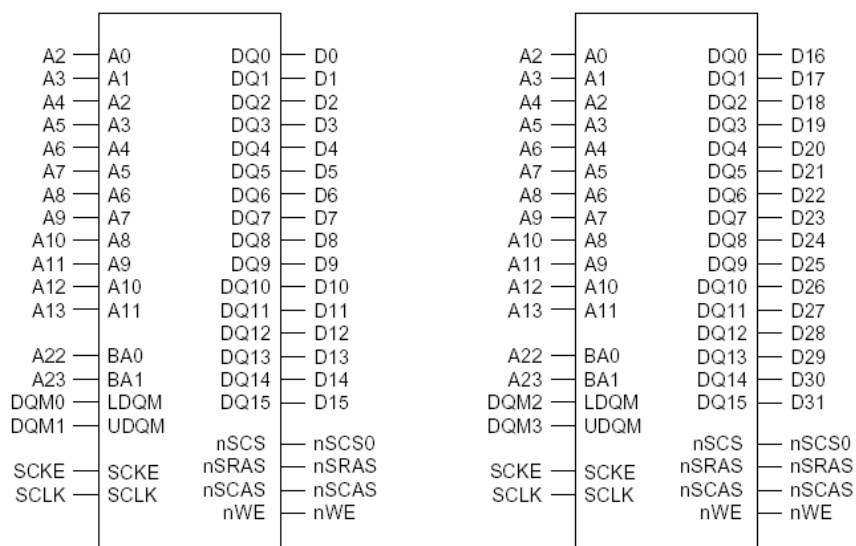


图 4-12 S3C2410X nGCS 时序图

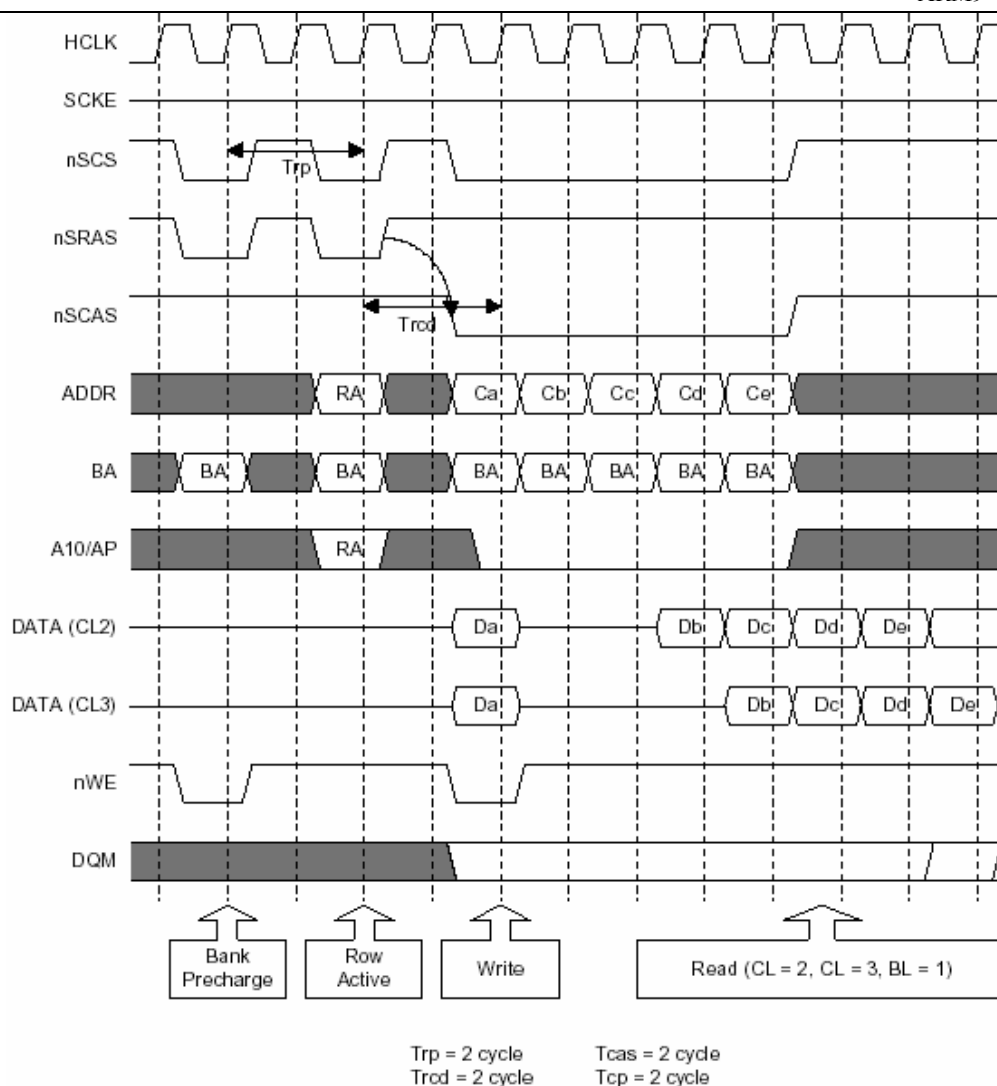


图 4-13 S3C2410X SDRAM 时序图

存储器控制专用寄存器

总线宽度/等待控制寄存器 (BWSCON)

Register	Address	R/W	Description	Reset Value
BWSCON	0x48000000	R/W	Bus width & wait status control register	0x000000

寄存器各位功能:

[DW_i]: i=0~7, 其中 DW0 为只读, 因为 bank0 数据总线宽度在复位后已经由

OM[1: 0]的电平决定。DW1~DW7 可写, 用于配置 bank1~bank7 的数据总线宽度, 00 表示 8 位数据总线宽度, 01 表示 16 位数据总线宽度, 10 表示 32 位数据总线宽度, 11 保留。

[WS_i]: i=1~7, 写入 0 则对应的 bank_i 等待状态不使用, 写入 1 则对应的 bank_i 等待状态使能。

[ST_i]: i=1~7, 决定 SRAM 是否使用 UB/LB。0 表示不使用 UB/LB, 引脚[14: 11]定义为 nWBE[3: 0]; 1 表示使用 UB/LB, 引脚[14: 11]定义为 nBE[3: 0]。

Bank 控制寄存器(BANKCONn: nGCS0-nGCS5)

Register	Address	R/W	Description	Reset Value
BANKCON0	0x48000004	R/W	Bank 0 control register	0x0700
BANKCON1	0x48000008	R/W	Bank 1 control register	0x0700
BANKCON2	0x4800000C	R/W	Bank 2 control register	0x0700
BANKCON3	0x48000010	R/W	Bank 3 control register	0x0700
BANKCON4	0x48000014	R/W	Bank 4 control register	0x0700
BANKCON5	0x48000018	R/W	Bank 5 control register	0x0700

Bank 控制寄存器(BANKCONn: nGCS6-nGCS7)

Register	Address	R/W	Description	Reset Value
BANKCON6	0x4800001C	R/W	Bank 6 control register	0x18008
BANKCON7	0x48000020	R/W	Bank 7 control register	0x18008

刷新控制寄存器 (REFRESH)

Register	Address	R/W	Description	Reset Value
REFRESH	0x48000024	R/W	SDRAM refresh control register	0xac0000

BANK 大小寄存器 (BANKSIZE)

Register	Address	R/W	Description	Reset Value
BANKSIZE	0x48000028	R/W	Flexible bank size register	0x0

SDRAM 模式设置寄存器 (MRSR)

Register	Address	R/W	Description	Reset Value
MRSRB6	0x4800002C	R/W	Mode register set register bank6	xxx
MRSRB7	0x48000030	R/W	Mode register set register bank7	xxx

以上所提到的寄存器的详细解释及设置请参考 S3C2410 数据手册。

下面列举了 13 个存储控制寄存器的配置示例：

```

BANKCON0_Val EQU 0x00000700
BANKCON1_Val EQU 0x00000700
BANKCON2_Val EQU 0x00000700
BANKCON3_Val EQU 0x00000700
BANKCON4_Val EQU 0x00000700
BANKCON5_Val EQU 0x00000700
BANKCON6_Val EQU 0x00018005
BANKCON7_Val EQU 0x00018005
BWSCON_Val EQU 0X22119120;0x22111110
REFRESH_Val EQU 0x008e0459
BANKSIZE_Val EQU 0x00000032;0x000000b2
MRSRB6_Val EQU 0x00000030
MRSRB7_Val EQU 0x00000030

```

观察上面寄存器介绍中的寄存器地址可以发现，13 个寄存器分布在从 0x48000000 开始的连续地址空间，所以上面的程序可以利用指令“stmia r0, {r1-r13}”实现将配置好的寄存器的值依次写入到相应的寄存器中。

2. 片选信号设置和外围地址空间分配

Embest ARM 教学实验系统的外围地址空间分配和片选信号设置如表 4-5 所示：

表 4-5 S3C2410X 的片选信号设置和外围地址空间分配

片选信号				选择的接口或器件	片选控制寄存器	S3C2410 地址范围
NGCS0				FLASH	BANKCON0	0X0000_0000~0X07FF_FFFF
NGCS6				SDRAM	BANKCON6	0X3000_0000~0X3DF_FFFF
n G C S 1	A22	A21	A20		BANKCON1	
	0	0	0	USB_CS		0X0800_0000~0X080F_FFFF
	0	0	1	CAN_CS		0X0810_0000~0X081F_FFFF
	0	1	0	CF_CS0		0X0820_0000~0X082F_FFFF
	0	1	1	CF_CS1		0X0830_0000~0X083F_FFFF
	1	0	0	保留		
	1	0	1	LCD_CD		0X0850_0000~0X085F_FFFF
	1	1	0			
	1	1	1			
n G C S 2	A22	A21	A20		BANKCON2	
	0	0	0	扩展输出寄存器 1		0X1000_0000~0X100F_FFFF
	0	0	1	扩展输入寄存器 1		0X1010_0000~0X101F_FFFF
	0	1	0	扩展输入寄存器 2		0X1020_0000~0X102F_FFFF
	0	1	1	CF_MMRD/WR		0X1030_0000~0X103F_FFFF
	1	0	0	CF_IORD/WR		0X1040_0000~0X104F_FFFF
	1	0	1	NO USE		0X1050_0000~0X105F_FFFF
	1	1	0	NO USE		0X1060_0000~0X106F_FFFF
	1	1	1	NO USE		0X1070_0000~0X107F_FFFF
NGCS3				ETHERNET	BANKCON3	0X1800_0000~0X1FFF_FFFF

4.1.5 实验操作步骤

1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接到目标板上。使用 Embest EduKit-III 实验板附带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板，打开实验例程目录 4.1_memory_test 子目录下的 memory_test. Uv2 例程，编译链接工程；
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境（工程默认已经配置正确），点击工具栏 “”，在 Option for Target 对话框的 Linker 页中选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug Session 项或点击工具栏 “”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；
- 4) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；
- 5) 打开 Memory 窗口，点击 Memory1 在地址输入栏中输入 0x3e000000；
- 6) 在工程管理窗口中双击 memory_test.c 就会打开该文件，在约第 37 行 (*pt=(UINT32T)pt;)设置断点后，点击 Debug 菜单 Go 键运行程序；
- 7) 当程序停留到断后,按 F10，在 Memory1 窗口观察地址的内容并与地址相比较，对比是否一致，继续执行 F10。
- 8) 去掉断点，按 F5 并执行程序，观察超级终端串口程序的输出。
- 9) 结合实验内容和实验原理部分，掌握汇编语言和高级语言程序访问 RAM 指令的使用方法。

4. 观察实验结果

在 PC 机上观察超级终端程序主窗口，可以看到如下界面：

```
boot success...

Memory Test(3e000000h-30ff0000h):WR

Memory Test(3e000000h-30ff0000h):RD

O.K.
```

5. 完成实验练习题

理解和掌握实验后，完成实验练习题。

4.1.6 实验参考程序

```

/*****
* File:    main.c
* Author:  Wuhan R&Dembest
* Desc:    c main entry
* History:
*****/

/*-----*/
/*
                                includes files
*/

/*-----*/
#include "2410lib.h"
/*****
* name:      main
* func:      c code entry
* para:      none
* ret:       none
* modify:
* comment:
*****/

void main(int argc, char **argv)
{
    sys_init();          /* Initial s3c2410's Clock, MMU, Interrupt, Port and UART */
    memory_test();
    while(1)
    {
    };
}

```

```

/*****
* File:    memory.c
* Author:  Wuhan R&D Center, embest
* Desc:    memory test file
* History:
*****/

#include "def.h"
#include "option.h"
void memory_test(void)
{
    int i;
    UINT32T data;
    int memError=0;
    UINT32T *pt;

    // memory test
    uart_printf("\nMemory Test(%xh-%xh):WR\n", _RAM_STARTADDRESS+0xe00000,

```

```

(_ISR_STARTADDRESS&0xf0ff0000));
//memory write
pt=(UINT32T *)(_RAM_STARTADDRESS+0xe00000);
while((UINT32T)pt<(_ISR_STARTADDRESS&0xf0ff0000))
{
    *pt=(UINT32T)pt;
    pt++;
}
//MEMORY READ
uart_printf("Memory Test(%xh-%xh):RD\n", _RAM_STARTADDRESS+0xe00000,
(_ISR_STARTADDRESS&0xf0ff0000));
//uart_printf("\b\bRD");
pt=(UINT32T *)(_RAM_STARTADDRESS+0xe00000);
while((UINT32T)pt<(_ISR_STARTADDRESS&0xf0ff0000))
{
    data=*pt;
    if(data!=(UINT32T)pt)
    {
        memError=1;
        uart_printf("\b\bFAIL:0x%x=0x%x\n",i,data);
        break;
    }
    pt++;
}
if(memError==0)
    uart_printf("\n\b\bO.K.\n");
}

```

4.1.7 练习题

编写程序对 SRAM 进行字节，半字的读写访问。

4.2 IO 口实验

4.2.1 实验目的

- 掌握 S3C2410X 芯片的 I/O 控制寄存器的配置；
- 通过实验掌握 ARM 芯片使用 I/O 口控制 LED 显示；
- 了解 ARM 芯片中复用 I/O 口的使用方法。

4.2.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

4.2.3 实验内容

编写程序，控制实验平台的发光二极管 LED1,LED2,LED3,LED4，使它们有规律的点亮和熄灭，具体顺序如下：LED1 亮->LED2 亮->LED3 亮->LED4 亮->LED1 灭->LED2 灭->LED3 灭->LED4 灭->全亮->全灭，如此反复。

4.2.4 实验原理

S3C2410X 芯片上共有 71 个多功能的输入输出管脚，他们分为 7 组 I/O 端口。

- 一个 23 位的输出端口（端口 A）；
- 两个 11 位的输入/输出端口（端口 B、H）；
- 四个 16 位的输入/输出端口（端口 C、D、E、G）；
- 一个 8 位的输入/输出端口（端口 F）；

可以很容易的每组端口来满足不同系统配置和设计的需要。在运行程序之前必须对每个用到的管脚功能进行设置，如果某些管脚的复用功能没有使用，可以先将该管脚设置为 I/O 口。

1. S3C2410X I/O 口控制寄存器

端口控制寄存器（GPACON-GPHCON）

在 S3C2410X 中，大多数的管脚都复用，所以必须对每个管脚进行配置。端口控制寄存器（PnCON）定义了每个管脚的功能。

如果 GPF0-GPF7 和 GPG0-GPG7 在掉电模式使用了弱上拉信号，这些端口必须在中断模式配置。

端口数据寄存器（GPADAT-GPHDAT）

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

端口上拉寄存器（GPBUP-GPHUP）

端口上拉寄存器控制了每个端口组的上拉电阻的允许/禁止。如果某一位为 0，相应的上拉电阻被允许，如果是 1，相应的上拉电阻被禁止。

如果端口的上拉电阻被允许，无论在何种状态（INPUT, OUTPUT, DATAn, EINTn 等）下，上拉电阻都要起作用。

多状态控制寄存器

这个寄存器控制数据端口的上拉电阻，高阻态，USB pad 和 CLKOUT 选项。

外部中断控制寄存器（EXTINTN）

24 个外部中断有各种各样的中断请求信号，EXTINTN 寄存器可以配置信号的类型为低电平触发，高电平触发，下降沿触发，上升沿触发，两沿触发中断请求。

8 个外部中断引脚有数字滤波器（参考数据手册中的 EINTFLTn）。

只有 16 个外部中断(EINT[15:0])用于唤醒 cpu。

掉电模式和 I/O 端口

在掉电模式下仍然保持所有的 GPIO 的所有状态值，可以参考相应章节的内容。

EINTMASK 在掉电模式下也不能阻止唤醒 cpu。但是如果 EINTMASK 屏蔽了 EINT[15:4]的某一位，仍然可以唤醒 cpu，但是 SRCPND 的 EINT4_7 位和 EINT8_23 位不能在 cpu 唤醒后马上被置位。

I/O 端口控制寄存器

端口 A 控制寄存器（GPACON/GPADAT）

Register	Address	R/W	Description	Reset Value
GPACON	0x56000000	R/W	Configure the pins of port A	0x7FFFFFFF
GPADAT	0x56000004	R/W	The data register for port A	Undefined
Reserved	0x56000008	—	Reserved	Undefined
Reserved	0x5600000C	—	Reserved	Undefined

GPACON[22:0]中的某一位置位，设置与该位相对应的引脚为输出口，清零某位可以设置相应的引脚为功能端口，功能端口的如表 4-6 所示：

表 4-6 端口 A 功能配置

端口 A	功能	端口 A	功能	端口 A	功能	端口 A	功能
GPA0	ADDR0	GPA6	ADDR21	GPA12	nGCS1	GPA18	ALE
GPA1	ADDR16	GPA7	ADDR22	GPA13	nGCS2	GPA19	nFWE
GPA2	ADDR17	GPA8	ADDR23	GPA14	nGCS3	GPA20	nFRE
GPA3	ADDR18	GPA9	ADDR24	GPA15	nGCS4	GPA21	nRSTOUT
GPA4	ADDR19	GPA10	ADDR25	GPA16	nGCS5	GPA22	nFCE
GPA5	ADDR20	GPA11	ADDR26	GPA17	CLE		

端口被配置为输出引脚后，引脚的状态喝相应的 bit 位状态一致。当端口被配置为功能引脚后，读出来的值不确定。

端口 B 控制寄存器（GPBCON,GPBDAT,GPBUP）

Register	Address	R/W	Description	Reset Value
GPBCON	0x56000010	R/W	Configure the pins of port B	0x0
GPBDAT	0x56000014	R/W	The data register for port B	Undefined
GPBUP	0x56000018	R/W	Pull-up disable register for port B	0x0
Reserved	0x5600001C	—	Reserved	Undefined

端口 B 控制寄存器对端口 B 的具体配置如下：

GPBCON	Bit	Description	
GPB10	[21:20]	00 = Input 10 = nXDREQ0	01 = Output 11 = reserved
GPB9	[19:18]	00 = Input 10 = nXDACK0	01 = Output 11 = reserved
GPB8	[17:16]	00 = Input 10 = nXDREQ1	01 = Output 11 = Reserved
GPB7	[15:14]	00 = Input 10 = nXDACK1	01 = Output 11 = Reserved
GPB6	[13:12]	00 = Input 10 = nXBREQ	01 = Output 11 = reserved
GPB5	[11:10]	00 = Input 10 = nXBACK	01 = Output 11 = reserved
GPB4	[9:8]	00 = Input 10 = TCLK0	01 = Output 11 = reserved

端口 B 被配置位输入端口后，可以从引脚上读出相应的外部源输入的数据。如果端口被配置位输出端口，向该位写入的数据可以被发送到相应的引脚上。如果该引脚被配置位功能引脚，读出的数据不确定。

置位 GPBUP[10:0]中的某位，允许端口 B 的相应位的上拉功能，反之禁止上拉功能。

端口 C 控制寄存器 (GPCCON, GPCDAT, GPCUP)

Register	Address	R/W	Description	Reset Value
GPCCON	0x56000020	R/W	Configure the pins of port C	0x0
GPCDAT	0x56000024	R/W	The data register for port C	Undefined
GPCUP	0x56000028	R/W	Pull-up disable register for port C	0x0
Reserved	0x5600002C	—	Reserved	Undefined

端口 C 控制寄存器 (GPCCON) 的具体配置如下图所示：

GPCCON	Bit	Description	
GPC15	[31:30]	00 = Input 10 = VD[7]	01 = Output 11 = Reserved
GPC14	[29:28]	00 = Input 10 = VD[6]	01 = Output 11 = Reserved
GPC13	[27:26]	00 = Input 10 = VD[5]	01 = Output 11 = Reserved
GPC12	[25:24]	00 = Input 10 = VD[4]	01 = Output 11 = Reserved
GPC11	[23:22]	00 = Input 10 = VD[3]	01 = Output 11 = Reserved
GPC10	[21:20]	00 = Input 10 = VD[2]	01 = Output 11 = Reserved
GPC9	[19:18]	00 = Input 10 = VD[1]	01 = Output 11 = Reserved
GPC8	[17:16]	00 = Input 10 = VD[0]	01 = Output 11 = Reserved
GPC7	[15:14]	00 = Input 10 = LCDVF2	01 = Output 11 = Reserved
GPC6	[13:12]	00 = Input 10 = LCDVF1	01 = Output 11 = Reserved
GPC5	[11:10]	00 = Input 10 = LCDVF0	01 = Output 11 = Reserved
GPC4	[9:8]	00 = Input 10 = VM	01 = Output 11 = Reserved
GPC3	[7:6]	00 = Input 10 = VFRAME	01 = Output 11 = Reserved
GPC2	[5:4]	00 = Input 10 = VLINE	01 = Output 11 = Reserved
GPC1	[3:2]	00 = Input 10 = VCLK	01 = Output 11 = Reserved
GPC0	[1:0]	00 = Input 10 = LEND	01 = Output 11 = Reserved

如果端口 C 被配置为输入端口，可以从引脚读出相应外部输入源输入的数据。如果端口被配置为输出端口，向寄存器写的数据可以被送往相应的引脚。如果端口被配置为功能引脚，从该引脚读出的数据不确定。

置位 GPCUP[15:0]的某一位允许相应引脚的上拉功能，否则禁止上拉功能。

端口 D 控制寄存器 (GPDCON, GPDDAT, GPDUP)

Register	Address	R/W	Description	Reset Value
GPDCON	0x56000030	R/W	Configure the pins of port D	0x0
GPDDAT	0x56000034	R/W	The data register for port D	Undefined
GPDUP	0x56000038	R/W	Pull-up disable register for port D	0xF000
Reserved	0x5600003C	—	Reserved	Undefined

GPDCON 配置情况如下：

GPDCON	Bit	Description	
GPD15	[31:30]	00 = Input 10 = VD23	01 = Output 11 = nSS0
GPD14	[29:28]	00 = Input 10 = VD22	01 = Output 11 = nSS1
GPD13	[27:26]	00 = Input 10 = VD21	01 = Output 11 = Reserved
GPD12	[25:24]	00 = Input 10 = VD20	01 = Output 11 = Reserved
GPD11	[23:22]	00 = Input 10 = VD19	01 = Output 11 = Reserved
GPD10	[21:20]	00 = Input 10 = VD18	01 = Output 11 = Reserved
GPD9	[19:18]	00 = Input 10 = VD17	01 = Output 11 = Reserved
GPD8	[17:16]	00 = Input 10 = VD16	01 = Output 11 = Reserved
GPD7	[15:14]	00 = Input 10 = VD15	01 = Output 11 = Reserved
GPD6	[13:12]	00 = Input 10 = VD14	01 = Output 11 = Reserved
GPD5	[11:10]	00 = Input 10 = VD13	01 = Output 11 = Reserved
GPD4	[9:8]	00 = Input 10 = VD12	01 = Output 11 = Reserved
GPD3	[7:6]	00 = Input 10 = VD11	01 = Output 11 = Reserved
GPD2	[5:4]	00 = Input 10 = VD10	01 = Output 11 = Reserved
GPD1	[3:2]	00 = Input 10 = VD9	01 = Output 11 = Reserved
GPD0	[1:0]	00 = Input 10 = VD8	01 = Output 11 = Reserved

如果端口 D 被配置为输入端口，可以从引脚读出相应外部输入源输入的数据。如果端口被配置为输出端口，向寄存器写的的数据可以被送往相应的引脚。如果端口被配置为功能引脚，从该引脚读出的数据不确定。

置位 GPDUP[15:0]的某一位允许相应引脚的上拉功能，否则禁止上拉功能。

端口 E 控制寄存器（GPECON,GPEDAT,GPEUP）

Register	Address	R/W	Description	Reset Value
GPECON	0x56000040	R/W	Configure the pins of port E	0x0
GPEDAT	0x56000044	R/W	The data register for port E	Undefined
GPEUP	0x56000048	R/W	pull-up disable register for port E	0x0
Reserved	0x5600004C	—	Reserved	Undefined

GPECON	Bit	Description	
GPE15	[31:30]	00 = Input 10 = IICSDA	01 = Output (open drain output) 11 = Reserved
GPE14	[29:28]	00 = Input 10 = IIC_SCL	01 = Output (open drain output) 11 = Reserved
GPE13	[27:26]	00 = Input 10 = SPICLK0	01 = Output 11 = Reserved
GPE12	[25:24]	00 = Input 10 = SPIMOSI0	01 = Output 11 = Reserved
GPE11	[23:22]	00 = Input 10 = SPIMISO0	01 = Output 11 = Reserved
GPE10	[21:20]	00 = Input 10 = SDDAT3	01 = Output 11 = Reserved
GPE9	[19:18]	00 = Input 10 = SDDAT2	01 = Output 11 = Reserved
GPE8	[17:16]	00 = Input 10 = SDDAT1	01 = Output 11 = Reserved
GPE7	[15:14]	00 = Input 10 = SDDAT0	01 = Output 11 = Reserved
GPE6	[13:12]	00 = Input 10 = SDCMD	01 = Output 11 = Reserved
GPE5	[11:10]	00 = Input 10 = SDCLK	01 = Output 11 = Reserved
GPE4	[9:8]	00 = Input 10 = I2SSDO	01 = Output 11 = I2SSDI
GPE3	[7:6]	00 = Input 10 = I2SSDI	01 = Output 11 = nSS0
GPE2	[5:4]	00 = Input 10 = CDCLK	01 = Output 11 = Reserved
GPE1	[3:2]	00 = Input 10 = I2SSCLK	01 = Output 11 = Reserved
GPE0	[1:0]	00 = Input 10 = I2SLRCK	01 = Output 11 = Reserved

如果端口 E 被配置为输入端口，可以从引脚读出相应外部输入源输入的数据。如果端口被配置为输出端口，向寄存器写的的数据可以被送往相应的引脚。如果端口被配置为功能引脚，从该引脚读出的数据不确定。

置位 GPEUP[15:0]的某一位允许相应引脚的上拉功能，否则禁止上拉功能。

端口 F 控制寄存器 (GPFCON,GPFDAT,GPFUP)

Register	Address	R/W	Description	Reset Value
GPFCON	0x56000050	R/W	Configure the pins of port F	0x0
GPFDAT	0x56000054	R/W	The data register for port F	Undefined
GPFUP	0x56000058	R/W	Pull-up disable register for port F	0x0
Reserved	0x5600005C	—	Reserved	Undefined

GPFCON	Bit	Description	
GPF7	[15:14]	00 = Input 10 = EINT7	01 = Output 11 = Reserved
GPF6	[13:12]	00 = Input 10 = EINT6	01 = Output 11 = Reserved
GPF5	[11:10]	00 = Input 10 = EINT5	01 = Output 11 = Reserved
GPF4	[9:8]	00 = Input 10 = EINT4	01 = Output 11 = Reserved
GPF3	[7:6]	00 = Input 10 = EINT3	01 = Output 11 = Reserved
GPF2	[5:4]	00 = Input 10 = EINT2	01 = Output 11 = Reserved
GPF1	[3:2]	00 = Input 10 = EINT1	01 = Output 11 = Reserved
GPF0	[1:0]	00 = Input 10 = EINT0	01 = Output 11 = Reserved

如果端口 F 被配置为输入端口，可以从引脚读出相应外部输入源输入的数据。如果端口被配置为输出端口，向寄存器写的的数据可以被送往相应的引脚。如果端口被配置为功能引脚，从该引脚读出的数据不确定。

置位 GPFUP[15:0]的某一位允许相应引脚的上拉功能，否则禁止上拉功能。

端口 G 控制寄存器 (GPGCON,GPGDAT,GPGUP)

Register	Address	R/W	Description	Reset Value
GPGCON	0x56000060	R/W	Configure the pins of port G	0x0
GPGDAT	0x56000064	R/W	The data register for port G	Undefined
GPGUP	0x56000068	R/W	Pull-up disable register for port G	0xF800
Reserved	0x5600006C	—	Reserved	Undefined

GPGCON 对端口 G 的配置详情如下：

GPGCON	Bit	Description	
GPG15	[31:30]	00 = Input 10 = EINT23	01 = Output 11 = nYPON
GPG14	[29:28]	00 = Input 10 = EINT22	01 = Output 11 = YMON
GPG13	[27:26]	00 = Input 10 = EINT21	01 = Output 11 = nXPON
GPG12	[25:24]	00 = Input 10 = EINT20	01 = Output 11 = XMON
GPG11	[23:22]	00 = Input 10 = EINT19	01 = Output 11 = TCLK1
GPG10 (5V Tolerant Input)	[21:20]	00 = Input 10 = EINT18	01 = Output 11 = Reserved
GPG9 (5V Tolerant Input)	[19:18]	00 = Input 10 = EINT17	01 = Output 11 = Reserved
GPG8 (5V Tolerant Input)	[17:16]	00 = Input 10 = EINT16	01 = Output 11 = Reserved
GPG7	[15:14]	00 = Input 10 = EINT15	01 = Output 11 = SPICK1
GPG6	[13:12]	00 = Input 10 = EINT14	01 = Output 11 = SPIMOS1
GPG5	[11:10]	00 = Input 10 = EINT13	01 = Output 11 = SPIMISO1
GPG4	[9:8]	00 = Input 10 = EINT12	01 = Output 11 = LCD_PWREN
GPG3	[7:6]	00 = Input 10 = EINT11	01 = Output 11 = nSS1
GPG2	[5:4]	00 = Input 10 = EINT10	01 = Output 11 = nSS0
GPG1	[3:2]	00 = Input 10 = EINT9	01 = Output 11 = Reserved
GPG0	[1:0]	00 = Input 10 = EINT8	01 = Output 11 = Reserved

如果端口 G 被配置为输入端口，可以从引脚读出相应外部输入源输入的数据。如果端口被配置为输出端口，向寄存器写的的数据可以被送往相应的引脚。如果端口被配置为功能引脚，从该引脚读出的数据不确定。

置位 GPGUP[15:0]的某一位允许相应引脚的上拉功能，否则禁止上拉功能。

端口 H 控制寄存器 (GPHCON,GPHDAT,GPHUP)

Register	Address	R/W	Description	Reset Value
GPHCON	0x56000070	R/W	Configure the pins of port H	0x0
GPHDAT	0x56000074	R/W	The data register for port H	Undefined
GPHUP	0x56000078	R/W	Pull-up disable register for port H	0x0
Reserved	0x5600007C	—	Reserved	Undefined

GPHCON	Bit	Description	
GPH10	[21:20]	00 = Input 10 = CLKOUT1	01 = Output 11 = Reserved
GPH9	[19:18]	00 = Input 10 = CLKOUT0	01 = Output 11 = Reserved
GPH8	[17:16]	00 = Input 10 = UCLK	01 = Output 11 = Reserved
GPH7	[15:14]	00 = Input 10 = RXD2	01 = Output 11 = nCTS1
GPH6	[13:12]	00 = Input 10 = TXD2	01 = Output 11 = nRTS1
GPH5	[11:10]	00 = Input 10 = RXD1	01 = Output 11 = Reserved
GPH4	[9:8]	00 = Input 10 = TXD1	01 = Output 11 = Reserved
GPH3	[7:6]	00 = Input 10 = RXD0	01 = Output 11 = reserved
GPH2	[5:4]	00 = Input 10 = TXD0	01 = Output 11 = Reserved
GPH1	[3:2]	00 = Input 10 = nRTS0	01 = Output 11 = Reserved
GPH0	[1:0]	00 = Input 10 = nCTS0	01 = Output 11 = Reserved

如果端口 H 被配置为输入端口，可以从引脚读出相应外部输入源输入的数据。如果端口被配置为输出端口，向寄存器写的的数据可以被送往相应的引脚。如果端口被配置为功能引脚，从该引脚读出的数据不确定。

置位 GPHUP[15:0]的某一位允许相应引脚的上拉功能，否则禁止上拉功能。

2. 电路设计

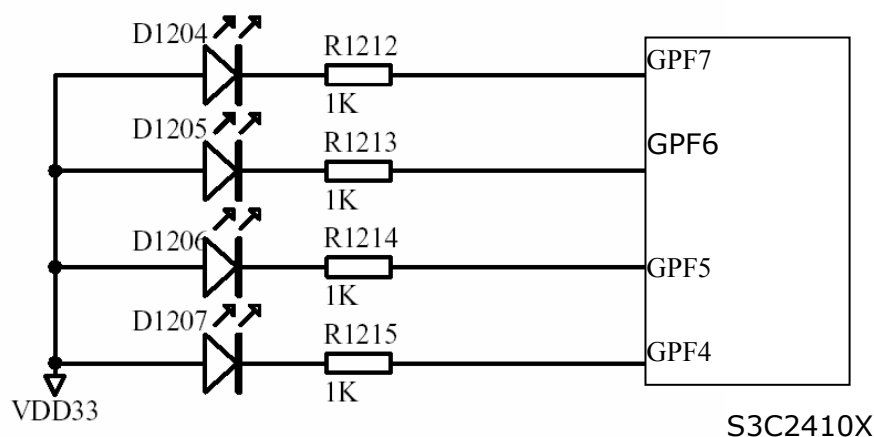


图 4-14 LED1-4 连接图

如图 4-14 所示, LED1-4 分别与 GFP7-4 相连, 通过 GFP7-4 引脚的高低电平来控制发光二极管的亮与灭。当这几个管脚输出高电平的时候发光二极管熄灭, 反之, 发光二极管点亮。

4.2.5 实验操作步骤



1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接到目标板上。使用 Embest EduKit-III 实验板附带的串口线, 连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序 (波特率 115200、1 位停止位、无校验位、无硬件流控制); 或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下 (如果已经拷贝, 可跳过此步骤);
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板, 打开实验例程目录 4.2_led_test 子目录下的 led_test. Uv2 例程, 编译链接工程;
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境 (工程默认已经配置正确), 点击工具栏 “”, 在 Option for Target 对话框的 Linker 页中选择 RuninRAM.sct 分散加载文件, 点击 MDK 的 Debug 菜单, 选择 Start/Stop Debug Session 项或点击工具栏 “”, 下载工程生成的.axf 文件到目标板的 RAM 中调试运行;
- 4) 如果需要将程序烧写固化到 Flash 中, 仅需要更改分散加载文件即可 (**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序, 建议实验中不操作**)。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件, 重新编译工程, 点击 MDK 的 Flash 菜单, 选择 Download 烧写调试代码到目标系统的 Nor Flash 中, 重启实验板, 实验板将会运行烧写到 Nor Flash 中的代码;
- 5) 在工程管理窗口中双击 led_test.c 就会打开该文件, 分别在约第 34 行 (for(i=0;i<100000;i++);) 和 58 行 (for(i=0;i<100000;i++);) 设置断点后, 点击 Debug 菜单 Go 运行程序;
- 6) 程序停到第一个断点处, 观察四个灯是否都被点亮 (注意观察渐变过程), 按 step out, 跳出这个子函数, 继续执行;
- 7) 程序运行到 led_off(), 按 step into, 停到第二个断点处, 观察四个灯是否都熄灭 (注意观察渐变过程)。按 step out, 继续执行;
- 8) 去掉断点, 重新下载, 执行程序;

4. 观察实验结果

观察发光二极管的亮灭情况, 可以观察到的现象与前面实验内容中的相符, 说明实验成功的实现了对 I/O 的操作。

```
boot success...
```

```
I/O (Diode Led) Test Example
```

```
end.
```

5. 完成实验练习题

理解和掌握实验后，完成实验练习题。

4.2.6 实验参考程序

```

/*****/

* File:    led_test.c

* Author:  Wuhan R&D Center, embest

* Desc:    IO port test file

* History:

*****/

#include "2410lib.h"

/*****/

* name:      led_xxx

* func:      the led operations

* para:      none

* ret:      none

*****/

void led_on(void)
{
    int i,nOut;
    nOut=0xF0;
    rGPFDAT=nOut & 0x70;
    for(i=0;i<100000;i++);
    rGPFDAT=nOut & 0x30;
    for(i=0;i<100000;i++);
    rGPFDAT=nOut & 0x10;
    for(i=0;i<100000;i++);
    rGPFDAT=nOut & 0x00;
    for(i=0;i<100000;i++);
}

void led_off(void)
{
    int i,nOut;
    nOut=0;
    rGPFDAT = 0;
    for(i=0;i<100000;i++);
    rGPFDAT = nOut | 0x80;
}

```



```

        for(i=0;i<100000;i++);

        rGPFDAT |= nOut | 0x40;

        for(i=0;i<100000;i++);

        rGPFDAT |= nOut | 0x20;

        for(i=0;i<100000;i++);

        rGPFDAT |= nOut | 0x10;

        for(i=0;i<100000;i++);

    }

    void led_on_off(void)
    {

        int i;

        rGPFDAT=0;

        for(i=0;i<100000;i++);

        rGPFDAT=0xF0;

        for(i=0;i<100000;i++);

    }

```

4.2.7 实验练习题

自己编写程序使数码管以不同的显示方式显示。

4.3 中断实验

4.3.1 实验目的

- 通过实验掌握 S3C2410X 的中断控制寄存器的使用；
- 通过实验掌握 S3C2410X 处理器的中断响应过程；
- 通过实验掌握不同中断触发方式下中断产生的过程；
- 通过实验掌握 ARM 处理器的中断方式和中断处理过程；
- 通过实验掌握 ARM 处理器中断处理的软件编程方法；

4.3.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

4.3.3 实验内容

编写中断服务程序，实现：

通过 UART0 选择中断触发方式，使能外部中断 Eint0、Eint11；

在不同的中断触发方式下，使用 Embest EduKit-III 实验平台的按钮 SB1202 触发 EINT0，同时在超级终端的主窗口中显示外部中断号；

在不同的中断触发方式下，使用 Embest EduKit-III 实验平台的按钮 SB1203 触发 EINT11，同时在超级终端的主窗口中显示外部中断号；

4.3.4 实验原理

1. S3C2410X 的中断

S3C2410X 的中断控制器可以接受多达 56 个中断源的中断请求。S3C2410X 的中断源可以由片内外设提供，比如 DMA、UART、IIC 等，其中 UARTn 中断和 EINTn 中断是逻辑或的关系，它们共用一条中断请求线。

S3C2410X 的中断源也可以由处理器的外部中断输入引脚提供，这部分中断源如下所示（11 个）：

INT_ADC	ADC 转换中断
INT_TC	触摸屏中断
INT_ERR2	UART2 收发错误中断
INT_TXD2	UART2 发送中断
INT_RXD2	UART2 接收中断
INT_ERR1	UART1 收发错误中断
INT_TXD1	UART1 发送中断
INT_RXD1	UART1 接收中断
INT_ERR0	UART0 收发错误中断
INT_TXD0	UART0 发送中断
INT_RXD0	UART0 接收中断

当 S3C2410X 收到来自片内外设和外部中断请求引脚的多个中断请求时，S3C2410X 的中断控制器在中断仲裁过程后向 S3C2410X 内核请求 FIQ 或 IRQ 中断。中断仲裁过程依靠处理器的硬件优先级逻辑，处理器在仲裁过程结束后将仲裁结果记录到 INTPEND 寄存器，以告知用户中断由哪个中断源产生。S3C2410X 的中断控制器的处理过程如图 4-3-1 所示。

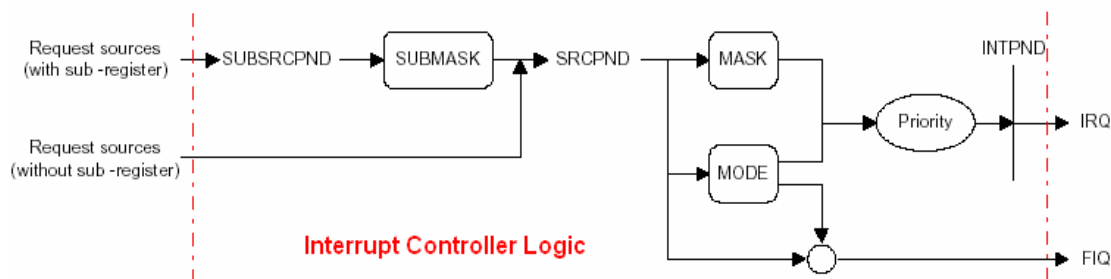


图 4-3-1 S3C2410X 的中断控制器

S3C2410X 的中断控制器的任务是在有多个中断发生时，选择其中一个中断通过 IRQ 或 FIQ 向 CPU 内核发出中断请求。

实际上最初 CPU 内核只有 FIQ（快速中断请求）和 IRQ（通用中断请求）两种中断，其它中断都是各个芯片厂家在设计芯片时，通过加入一个中断控制器来扩展定义的，这些中断根据中断的优先级高低来进行处理，更符合实际应用系统中要求提供多个中断源的要求。例如，如果你定义所有的中断源为 IRQ 中断（通过中断模式寄存器设置），并且同时有 10 个中断发出请求，这时可以通过读中断优先级寄存器来确定哪一个中断将被优先执行。

S3C2410X 的中断处理流程：当有中断源请求中断时，中断控制器处理中断请求，并根据处理结果向 CPU 内核发出 IRQ 请求或 FIQ 请求，同时，CPU 的程序指针 PC 将指向 IRQ 异常入口（0X18）或 FIQ 异常入口（0X1C），程序从 IRQ 异常入口（0X18）或 FIQ 异常入口（0X1C）开始执行。

2. S3C2410X 的中断控制

- 程序状态寄存器的 F 位和 I 位

如果 CPSR 程序状态寄存器的 F 位被设置为 1，那么 CPU 将不接受来自中断控制器的 FIQ（快速中断请求），如果 CPSR 程序状态寄存器的 I 位被设置为 1，那么 CPU 将不接受来自中断控制器的 IRQ（中断请求）。因此，为了使能 FIQ 和 IRQ，必须先将 CPSR 程序状态寄存器的 F 位和 I 位清零，并且中断屏蔽寄存器 INTMSK 中相应的位也要清零。

- 中断模式（INTMOD）

ARM920T 提供了 2 种中断模式，FIQ 模式和 IRQ 模式。所有的中断源在中断请求时都要确定使用哪一种中断模式。

- 中断挂起寄存器（INTPND）

S3C2410X 有两个中断挂起寄存器：源中断挂起寄存器（SRCPND）和中断挂起寄存器（INTPND），用于指示对应的中断是否被激活。当中断源请求中断的时候，SRCPND 寄存器的相应位被置 1，同时 INTPND 寄存器中也有唯一的一位在仲裁程序后被自动置 1，如果屏蔽位被设置为 1，相应的 SRCPND 位会被置 1，但是 INTPND 寄存器不会有变化，如果 INTPND 被置位，只要标志 I 或标志 F 一被清零，就会执行相应的中断服务程序。在中断服务子程序中要先向 SRCPND 中的相应位写 1 来清除源挂起状态，再用同样的方法来清除 INTPND 的相应位的挂起状态。

可以通过 $INTPND = INTPND$ ；来实现清零，以避免写入不正确的数据引起错误

- 中断屏蔽寄存器（INTMSK）

当 INTMSK 寄存器的屏蔽位为 1 时，对应的中断被禁止；当 INTMSK 寄存器的屏蔽位为 0 时，则对应的中断正常执行。如果一个中断的屏蔽位为 1，在该中断发出请求时挂起位还是会被设置为 1，但中断请求都不被受理。

3. S3C2410X 的中断源

在 56 个中断源中，有 30 个中断源提供给中断控制器，其中外部中断 EINT4/5/6/7 通过逻辑“或”的形式提供给中断控制器，EINT8-EINT23 也通过逻辑“或”的形式提供给中断控制器（见图 4-3-1）。

表 4-3-1 S3C2410X 的中断源

Sources	Descriptions	Arbiter Group
INT_ADC	ADC EOC and Touch interrupt (INT_ADC/INT_TC)	ARB5
INT_RTC	RTC alarm interrupt	ARB5
INT_SPI1	SPI1 interrupt	ARB5
INT_UART0	UART0 Interrupt (ERR, RXD, and TXD)	ARB5
INT_IIC	IIC interrupt	ARB4
INT_USBH	USB Host interrupt	ARB4
INT_USBD	USB Device interrupt	ARB4
Reserved	Reserved	ARB4
INT_UART1	UART1 Interrupt (ERR, RXD, and TXD)	ARB4
INT_SPI0	SPI0 interrupt	ARB4
INT_SDI	SDI interrupt	ARB 3
INT_DMA3	DMA channel 3 interrupt	ARB3
INT_DMA2	DMA channel 2 interrupt	ARB3
INT_DMA1	DMA channel 1 interrupt	ARB3
INT_DMA0	DMA channel 0 interrupt	ARB3
INT_LCD	LCD interrupt (INT_FrSyn and INT_FiCnt)	ARB3
INT_UART2	UART2 Interrupt (ERR, RXD, and TXD)	ARB2
INT_TIMER4	Timer4 interrupt	ARB2
INT_TIMER3	Timer3 interrupt	ARB2
INT_TIMER2	Timer2 interrupt	ARB2
INT_TIMER1	Timer1 interrupt	ARB 2
INT_TIMER0	Timer0 interrupt	ARB2
INT_WDT	Watch-Dog timer interrupt	ARB1
INT_TICK	RTC Time tick interrupt	ARB1
nBATT_FLT	Battery Fault interrupt	ARB1
Reserved	Reserved	ARB1
EINT8_23	External interrupt 8 – 23	ARB1
EINT4_7	External interrupt 4 – 7	ARB1
EINT3	External interrupt 3	ARB0
EINT2	External interrupt 2	ARB0
EINT1	External interrupt 1	ARB0
EINT0	External interrupt 0	ARB0

4. S3C2410X 的中断控制寄存器

S3C2410X 的中断控制器有 5 个控制寄存器：源挂起寄存器（SRCPND）、中断模式寄存器（INTMOD）、中断屏蔽寄存器（INTMSK）、中断优先级寄存器（PRIORITY）、中断挂起寄存器（INTPND）。

中断源发出的中断请求首先被寄存器在中断源挂起寄存器（SRCPND）中，INTMOD 把中断请求分为两组：快速中断请求（FIQ）和中断请求（IRQ），PRIORITY 处理中断的优先级。

● 源挂起寄存器（SRCPND）

中断控制寄存器 INTCON 共有 32 位，每一位对应着一个中断源，当中断源发出中断请求的时候，就会置位源挂起寄存器的相应位。反之，中断的挂起寄存器的值为 0。

寄存器	地址	R/W	描述	复位值
SRCPND	0x4A000000	R/W	0 = 中断没有发出请求 1 = 中断源发出中断请求	0x00000000

● 中断模式寄存器（INTMOD）

中断模式寄存器 INTMOD 共有 32 位，每一位对应着一个中断源，当中断源的模式位设置为 1 时，对应的中断会由 ARM920T 内核以 FIQ 模式来处理。相反的，当模式位设置为 0 时，中断会以 IRQ 模式来处理。

寄存器	地址	R/W	描述	复位值
INTMOD	0x4A000004	R/W	0=IRQ 模式 1=FIQ 模式	0x00000000

注意，中断控制寄存器中只有一个中断源可以被设置为 FIQ 模式，因此只能在紧急情况下使用 FIQ。如果 INTMOD 寄存器把某个中断设为 FIQ 模式，FIQ 中断不影响 INTPND 和 INTOFFSET 寄存器，因此，这两个寄存器只对 IRQ 模式中断有效。

● 中断屏蔽寄存器 (INTMSK)

这个寄存器有 32 位，分别对应一个中断源。当中断源的屏蔽位设置为 1 时，CPU 不响应该中断源的中断请求，反之，等于 0 时 CPU 能响应该中断源的中断请求。

寄存器	地址	R/W	描述	复位值
INTMSK	0x4A000008	R/W	0=允许响应中断请求 1=中断请求被屏蔽	0xFFFFFFFF

● 中断挂起寄存器 (INTPND)

中断挂起寄存器 INTPND 共有 32 位，每一位对应着一个中断源，当中断请求被响应的时候，相应的位会被设置为 1。在某一时刻只有一个位能为 1，因此在中断服务子程序中可以通过判断 INTPND 来判断哪个中断正在被响应，在中断服务子程序中必须在清零 SRCPND 中相应位后清零相应的中断挂起位，清零方法和 SRCPND 相同。

寄存器	地址	R/W	描述	复位值
INTPND	0x4A000010	R/W	0=未发生中断请求 1=中断源发出中断请求	0x00000000

注意：

- 1. FIQ 响应的时候不会影响 INTPND 相应的标志位*
- 2. 向 INTPND 等于“1”的位写入“0”时，INTPND 寄存器和 INTOFFSET 寄存器会有无法预知的结果，因此，千万不要向 INTPND 的“1”位写入“0”，推荐的清零方法是把 INTPND 的值重新写入 INTPND，尽管我们也没有这么做。*

● IRQ 偏移寄存器

中断偏移寄存器给出 INTPND 寄存器中哪个是 IRQ 模式的中断请求。

寄存器	地址	R/W	描述	复位值
INTOFFSET	0x4A000014	R	指示中断请求源的 IRQ 模式	0x00000000

S3C2410X 中的优先级产生模块包含 7 个单元，1 个主单元和 6 个从单元。两个从优先级产生单元管理 4 个中断源，四个从优先级产生单元管理 6 个中断源。主优先级产生单元管理 6 个从单元。

每一个从单元有 4 个可编程优先级中断源和 2 个固定优先级中断源。这 4 个中断源的优先级是由 ARB_SEL 和 ARM_MODE 决定的。另外 2 个固定优先级中断源在 6 个中断源中的优先级最低。

- 外部中断控制寄存器 (EXTINTn)

S3C2410X 的 24 个外部中断有几种中断触发方式，EXTINTn 配置外部中断的触发类型是电平触发、边沿触发以及触发的极性。

EXTINT0/1/2/3 具体配置参考数据手册。

- 外部中断屏蔽寄存器 (EXTMASK)

寄存器	地址	R/W	描述	复位值
EXTMASK	0x560000A4	R/W	外部中断屏蔽标志	0x00FFFFFF0

EXTMASK[23:4]分别对应外部中断 23-4，等于 1 对应的中断被屏蔽，等于 0，允许外部中断。EXTMASK[3:0]保留。

5. 电路原理

本实验选择的是外部中断 EXTINT0 和 EXTINT11。中断的产生分别来至按钮 SB1202 和 SB1203，当按钮按下时，EXTINT0 或 EXTINT11 和地连接，输入低电平，从而向 CPU 发出中断请求。当 CPU 受理中断后，进入相应的中断服务程序，通过超级终端的主窗口显示当前进入的中断号。电路原理图如图 4-3-2 所示。

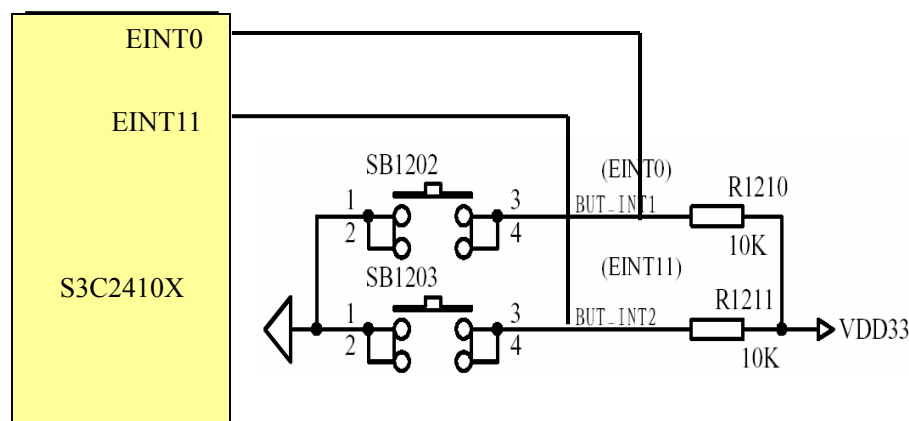


图 4-3-2 S3C2410X 中断实验电路图

4.3.5 实验操作步骤

1. 准备实验环境


使用 ULINK USB-JTAG 仿真器连接到目标板上。使用 Embest EduKit-III 实验板附带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置


在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板，打开实验例程目录 4.3_int_test 子目录下的 int_test.Uv2 例程，编译链接工程；
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境（工

程默认已经配置正确），点击工具栏 “”，在 Option for Target 对话框的 Linker 页中

选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug

Session 项或点击工具栏 “”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；

- 4) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；
- 5) 点击 Peripherals 菜单->Interrupt Controller，打开中断控制器的窗口，在试验中观察中断控制寄存器的值的变化，如下图 4-3-3 所示：

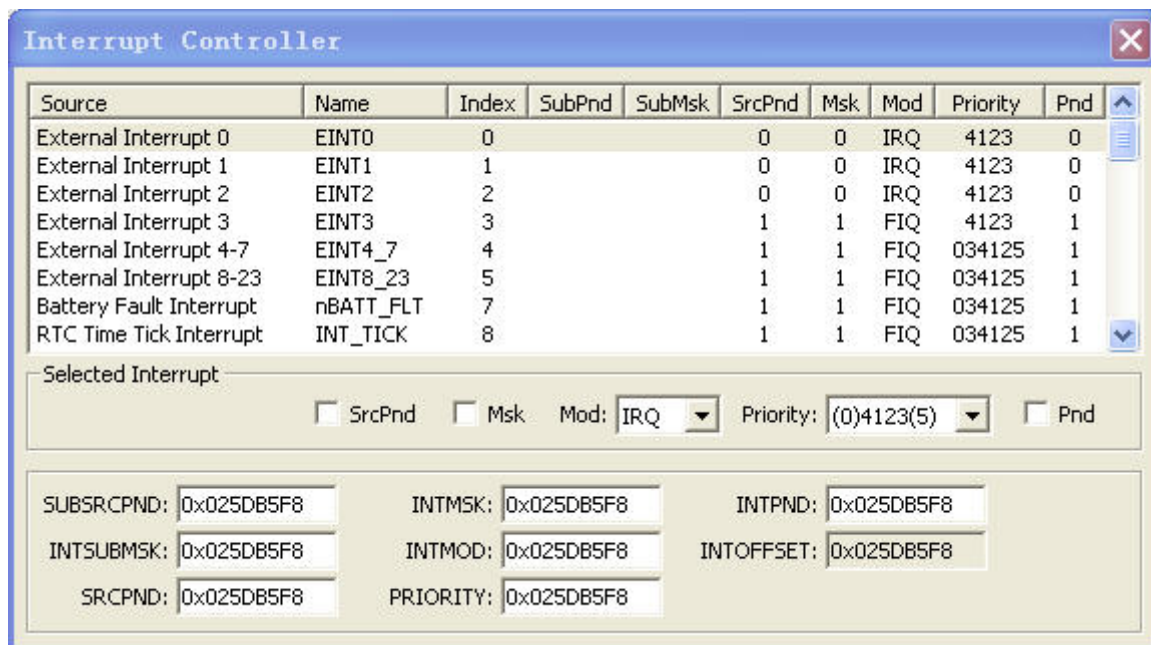


图 4-3-3 μ Vision IDE 中断控制器窗口

- 6) 在工程管理窗口中双击 int_test.c 就会打开该文件，分别在两个中断服务程序的第一行语句处设置断点，在函数 int_test() 的函数调用语句 “int_init();” 处设置断点，点击菜单 Debug-> Run 或 F5 键运行程序，程序正确运行后，会在超级终端上输出如下信息：

```
boot success...
```

```
External Interrupt Test Example
```

```
1.L-LEVEL    2.H-LEVEL    3.F-EDGE(default here)    4.R-EDGE
```

```
5.B-EDGE
```

```
Select number to change the external interrupt type:
```

```
Press the Buttons (SB1202/SB1203) to test...
```

```
Press SPACE(PC) to exit...
```

- 7) 程序停留在断点，在超级终端界面，使用 PC 机键盘，输入所需设置的中断触发方式（默认 3）后，在程序界面按 F10，此时注意观察图 4-3-3 中中断控制寄存器的值，即中断配置情况；如果选择 2，是高电平，一直有中断。
- 8) 连续按 F10，当前光标运行到（while(g_nKeyPress&(g_nKeyPress<6))), 等待按下按钮产生中断；当按下 SB1202/或 SB1203 后，按 F10 两次，程序停留到中断服务程序入口的断点，再次观察图 4-3-2 中中断控制寄存器的值，右击 INTERRUPT,刷新寄存器窗口，注意观察各个值在程序运行前后的变化（提示：中断申请标志位应该被置位）；
- 9) 去掉断点，重新下载执行程序，按下数字键选择相应的中断触发方式，按下按键 SB1202 或 SB1203，在超级终端的主窗口中观察输出结果是否与事实相符；
- 10) 选择不同的外部中断信号触发方式，观察不同中断触发方式按下 SB1202 或 SB1203 时，超级终端输出中断情况；
- 11) 结合实验内容和实验原理部分，掌握 ARM 处理器中断操作过程，如中断使能、设置中断触发方式和中断源识别等，重点理解 ARM 处理器的中断响应及中断处理的过程。

4. 观察实验结果

等待选择输入所需中断方式设置：

```
boot success...

External Interrupt Test Example

1.L-LEVEL    2.H-LEVEL    3.F-EDGE(default here)    4.R-EDGE
5.B-EDGE

Select number to change the external interrupt type:

Press the Buttons (SB1202/SB1203) to test...

Press SPACE(PC) to exit...
```

在 PC 机键盘上输入 3 选择下降沿触发，并按下按钮 SB1202

```
3.F-EDGE

EINT0 interrupt occurred.

end.
```

按下 SB1203 键，超级终端主窗口中继续显示：

```
External Interrupt Test Example

1.L-LEVEL    2.H-LEVEL    3.F-EDGE(default here)    4.R-EDGE
5.B-EDGE

Select number to change the external interrupt type:

Press the Buttons (SB1202/SB1203) to test...

Press SPACE(PC) to exit...
```



```
EINT11 interrupt occurred.
end.
```

其他选择类似。

5. 完成实验练习题

理解和掌握实验后，完成实验练习题。

4.3.6 实验参考程序

1. 中断初始化程序

```

/*****
* name:      int_init
* func:      Interrupt initialize
* para:      none
* ret:       none
* modify:
* comment:
*           EINT0 --- SB202  EINT11 --- SB203
*****/
void int_init(void)
{
    rSRCPND = rSRCPND;           // clear all interrupt
    rINTPND = rINTPND;          // clear all interrupt

    rGPFCON = (rGPFCON & 0xffcc) | (1<<5) | (1<<1);    // PF0/2 = EINT0/2
    rGPGCON = (rGPGCON & 0xff3fff3f) | (1<<23) | (1<<7); // PG3/11 = EINT11/19

    pISR_EINT0   = (UINT32T)int0_int;    // isrEINT0;
    pISR_EINT8_23 = (UINT32T)int11_int;   // isrEINT11_19;

    rEINTPEND = 0xffffffff;
    rSRCPND   = BIT_EINT0 | BIT_EINT8_23; //to clear the previous pending states
    rINTPND   = BIT_EINT0 | BIT_EINT8_23;

    rEXTINT0 = (rEXTINT0 & ~(7<<8) | (0x7<<0))) | 0x2<<8 | 0x2<<0;
    rEXTINT1 = (rEXTINT1 & ~(7<<12)) | 0x2<<12;

    rEINTMASK &= ~(1<<11);
    rINTMSK   &= ~(BIT_EINT0 | BIT_EINT8_23);
}

/*****
* name:      int_test
* func:      Extern interrupt test
* para:      none

```

```

* ret:      none
* modify:
* comment:
*          EINT0 --- SB202  EINT11 --- SB203
*****/
void int_test(void)
{
    int nIntMode;

    uart_printf("\n External Interrupt Test Example\n");
    uart_printf(" 1.L-LEVEL  2.H-LEVEL  3.F-EDGE(default here)  4.R-EDGE  5.B-EDGE\n");
    uart_printf(" Select number to change the external interrupt type:");
    uart_printf(" \nPress the Buttons (SB1202/SB1203) to test...\n");
    uart_printf(" Press SPACE(PC) to exit...\n");

    int_init();
    g_nKeyPress = 3;          // only 3 times test (for board test)
    while(g_nKeyPress & (g_nKeyPress < 6)) // SB1202/SB1203 to exit board test
    {
        nIntMode = uart_getkey();

        switch(nIntMode)
        {
            case '1':
                uart_printf(" 1.L-LEVEL\n");
                // EINT0/2=low level triggered,EINT11=low level triggered
                rEXTINT0 = (rEXTINT0 & ~((7<<8) | (0x7<<0))) | 0x0<<8 | 0x0<<0;
                rEXTINT1 = (rEXTINT1 & ~(7<<12)) | 0x0<<12;
                break;

            case '2':
                uart_printf(" 2.H-LEVEL\n");
                // EINT0/2=high level triggered,EINT11=high level triggered
                rEXTINT0 = (rEXTINT0 & ~((7<<8) | (0x7<<0))) | 0x1<<8 | 0x1<<0;
                rEXTINT1 = (rEXTINT1 & ~(7<<12)) | 0x1<<12;
                break;

            case '3':
                uart_printf(" 3.F-EDGE\n");
                // EINT0/2=falling edge triggered, EINT11=falling edge triggered
                rEXTINT0 = (rEXTINT0 & ~((7<<8) | (0x7<<0))) | 0x2<<8 | 0x2<<0;
                rEXTINT1 = (rEXTINT1 & ~(7<<12)) | 0x2<<12;
                break;

            case '4':
                uart_printf(" 4.R-EDGE\n");
                // EINT0/2=rising edge triggered,EINT11=rising edge triggered
                rEXTINT0 = (rEXTINT0 & ~((7<<8) | (0x7<<0))) | 0x4<<8 | 0x4<<0;

```

```

        rEXTINT1 = (rEXTINT1 & ~(7<<12)) | 0x4<<12;
        break;
    case '5':
        uart_printf(" 5.B-EDGE\n");
        // EINT0/2=both edge triggered,EINT11=both edge triggered
        rEXTINT0 = (rEXTINT0 & ~(7<<8) | (0x7<<0))) | 0x6<<8 | 0x6<<0;
        rEXTINT1 = (rEXTINT1 & ~(7<<12)) | 0x6<<12;
        break;
    case ' ':
        return;
    default:
        break;
    }
    delay(10000);
}
uart_printf(" end.\n");
}

```

2. 中断服务程序

```

/*****
* name:      int0_int
* func:      EXTINT0 interrupt service routine
* para:      none
* ret:       none
* modify:
* comment:
*****/
void __irq int0_int(void)
{
    uart_printf( " EINT0 interrupt occurred.\n" );
    ClearPending( BIT_EINT0 );
    if( g_nKeyPress )
        g_nKeyPress--=1;
}

/*****
* name:      int11_int
* func:      EXTINT11 interrupt service routine
* para:      none
* ret:       none
* modify:
* comment:
*****/
void __irq int11_int(void)
{
    if(rEINTPEND==(1<<11))

```

```

{
    uart_printf(" EINT11 interrupt occurred.\n");
    rEINTPEND=(1<<11);
    if(g_nKeyPress<20)
        g_nKeyPress+=1;
    else
        g_nKeyPress=0;
}
else if(rEINTPEND==(1<<19))
{
    uart_printf(" EINT19 interrupt occurred.\n");
    rEINTPEND=(1<<19);
}
else
{
    uart_printf(" rEINTPEND=0x%x\n",rEINTPEND);
    rEINTPEND=(1<<19)|(1<<11);
}
ClearPending(BIT_EINT8_23);
}

```

4.3.7 练习题

1. 熟悉 S3C2410X 芯片的时钟控制器及其相关的寄存器，掌握中断响应的完整过程。
2. 编写程序实现：按下 SB1202 或 SB1203 后点亮实验系统的 LEDs 一段时间后熄灭。

4.4 串口通信实验

4.4.1 实验目的

- 了解 S3C2410X 处理器 UART 相关控制寄存器的使用。
- 熟悉 ARM 处理器系统硬件电路中 UART 接口的设计方法。
- 掌握 ARM 处理器串行通信的软件编程方法。

4.4.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

4.4.3 实验内容

- 编写 S3C2410X 处理器的串口通信程序，
- 监视串行口 UART0 动作；
- 将从 UART0 接收到的字符串回送显示。

4.4.4 实验原理

1. S3C2410X 串行通讯 (UART) 单元

S3C2410X UART 单元提供三个独立的异步串行通信接口，皆可工作于中断和 DMA 模式。使用系统时钟最高波特率达 230.4Kbps，如果使用外部设备提供的时钟，可以达到更高的速率。每一个 UART 单元包含一个 16 字节的数据接收和发送 FIFO，用于数据的接收和发送。

S3C2410X UART 支持可编程波特率，红外发送/接收，一个或两个停止位，5bit/6bit/ 7bit/或 8bit 数据宽度和奇偶校验。

2. 波特率的产生

波特率由一个专用的 UART 波特率分频寄存器 (UBRDIVn) 控制，计算公式如下：

$$\text{UBRDIVn} = (\text{int})(\text{ULK}/(\text{bps} \times 16)) - 1$$

$$\text{或者 } \text{UBRDIVn} = (\text{int})(\text{PLK}/(\text{bps} \times 16)) - 1$$

其中：时钟选用 ULK 还是 PLK 由 UART 控制寄存器 UCONn[10] 的状态决定。如果 UCONn[10]=0, 用 PLK 作为波特率发生，否则选用 ULK 做波特率发生。UBRDIVn 的值必须在 1 到 (216-1) 之间。

例如：ULK 或者 PLK 等于 40MHz，当波特率为 115200 时，

$$\begin{aligned} \text{UBRDIVn} &= (\text{int})(40000000/(115200 \times 16)) - 1 \\ &= (\text{int})(21.7) - 1 \\ &= 21 - 1 = 20 \end{aligned}$$

3. UART 通信操作

下面简略介绍 UART 操作，关于数据发送，数据接收，中断产生，波特率产生，轮流检测模式，红外模式和自动流控制的详细介绍，请参照相关教材和数据手册。

发送数据帧是可编程的。一个数据帧包含一个起始位，5 到 8 个数据位，一个可选的奇偶校验位和 1 到 2 位停止位，停止位通过行控制寄存器 ULCONn 配置。

与发送类似，接收帧也是可编程的。接收帧由一个起始位，5 到 8 个数据位，一个可选的奇偶校验和 1 到 2 位行控制寄存器 ULCONn 里的停止位组成。接收器还可以检测溢出错误，奇偶校验错，帧错误和传输中断，每一个错误均可以设置一个错误标志。

溢出错误 (Overrun error) 是指已接收到的数据在读取之前被新接收的数据覆盖。

奇偶校验错是指接收器检测到的校验和与设置的不符。

帧错误指没有接收到有效的停止位。

传输中断表示接收数据 RxDn 保持逻辑 0 超过一帧的传输时间。

在 FIFO 模式下，如果 Rx FIFO 非空，而在 3 个字的传输时间内没有接收到数据，则产生超时。

4. UART 控制寄存器

1) UART 行控制寄存器 ULCONn

该寄存器的第 6 位决定是否使用红外模式，位 5~3 决定校验方式，位 2 决定停止位长度，位 1 和 0 决定每帧的数据位数。

2) UART 控制寄存器 UCONn

该寄存器决定 UART 的各种模式。

UCONn[10] = 1: ULK 做波特率发生; 0: PLK 做波特率发生。

UCONn[9] = 1: Tx 中断电平触发; 0: Tx 中断脉冲触发。

UCONn[8] = 1: Rx 中断电平触发; 0: Rx 中断脉冲触发。

UCONn[7] = 1: 接收超时中断允许; 0: 接收超时中断不允许。

UCONn[6] = 1: 产生接收错误中断; 0: 不产生接收错误中断。

UCONn[5] = 1: 发送直接传给接收方式 (Loopback); 0: 正常模式。

UCONn[4] = 1: 发送间断信号; 0: 正常模式发送。

UCONn[3: 2]: 发送模式选择

00: 不允许发送;

01: 中断或查询模式

10: DMA0 请求 (UART0)

DMA3 请求 (UART2)

11: DMA1 请求 (UART1) .

UCONn[1: 0]: 接收模式选择

00: 不允许接收

01: 中断或查询模式

10: DMA0 请求 (UART0)

DMA3 请求 (UART2)

11: DMA1 请求 (UART1)

3) UART FIFO 控制寄存器 UCONn

UFCONn[7:6] = 00: Tx FIFO 寄存器中有 0 个字节就触发中断

01: Tx FIFO 寄存器中有 4 个字节就触发中断

10: Tx FIFO 寄存器中有 8 个字节就触发中断

11: Tx FIFO 寄存器中有 0 个字节就触发中断

UFCONn[5:4] = 00: Rx FIFO 寄存器中有 0 个字节就触发中断

01: Rx FIFO 寄存器中有 4 个字节就触发中断

10: Rx FIFO 寄存器中有 8 个字节就触发中断

11: Rx FIFO 寄存器中有 0 个字节就触发中断

UFCONn[3]: 保留。

UFCONn[2] = 1: FIFO 复位清零 Tx FIFO; 0: FIFO 复位不清零 Tx FIFO

UFCONn[1] = 1: FIFO 复位清零 Rx FIFO; 0: FIFO 复位不清零 Rx FIFO

UFCONn[0] = 1: 允许 FIFO 功能; 0: 不允许 FIFO 功能

4) UART MODEM 控制寄存器 UMCOnn(n=0 或 1)

UMCONn[7:5] 保留, 必须全为 0

UMCONn[4] = 1: 允许使用 AFC 模式; 0: 不允许使用 AFC

UMCONn[3:1] 保留, 必须全为 0

UMCONn[0] = 1: 激活 nRTS; 0: 不激活 nRTS

5) 发送寄存器 UTXH 和接收寄存器 URXH

这两个寄存器存放着发送和接收的数据, 当然只有一个字节 8 位数据。需要注意的是在发生溢出错误的时候, 接收的数据必须被读出来, 否则会引发下次溢出错误。

6) 波特率分频寄存器 UBRDIV。

在例程目录下的 common\include\2410addr.h 文件中有关于 UART 单元各寄存器的定义。

```
// UART
#define rULCON0      (*(volatile unsigned *)0x50000000) //UART 0 Line control
#define rUCON0       (*(volatile unsigned *)0x50000004) //UART 0 Control
#define rUFCON0      (*(volatile unsigned *)0x50000008) //UART 0 FIFO control
#define rUMCON0      (*(volatile unsigned *)0x5000000c) //UART 0 Modem control
#define rUTRSTAT0    (*(volatile unsigned *)0x50000010) //UART 0 Tx/Rx status
#define rUERSTAT0    (*(volatile unsigned *)0x50000014) //UART 0 Rx error status
#define rUFSTAT0     (*(volatile unsigned *)0x50000018) //UART 0 FIFO status
#define rUMSTAT0     (*(volatile unsigned *)0x5000001c) //UART 0 Modem status
#define rUBRDIV0     (*(volatile unsigned *)0x50000028) //UART 0 Baud rate divisor

#define rULCON1      (*(volatile unsigned *)0x50004000) //UART 1 Line control
#define rUCON1       (*(volatile unsigned *)0x50004004) //UART 1 Control
#define rUFCON1      (*(volatile unsigned *)0x50004008) //UART 1 FIFO control
#define rUMCON1      (*(volatile unsigned *)0x5000400c) //UART 1 Modem control
#define rUTRSTAT1    (*(volatile unsigned *)0x50004010) //UART 1 Tx/Rx status
#define rUERSTAT1    (*(volatile unsigned *)0x50004014) //UART 1 Rx error status
#define rUFSTAT1     (*(volatile unsigned *)0x50004018) //UART 1 FIFO status
#define rUMSTAT1     (*(volatile unsigned *)0x5000401c) //UART 1 Modem status
#define rUBRDIV1     (*(volatile unsigned *)0x50004028) //UART 1 Baud rate divisor

#define rULCON2      (*(volatile unsigned *)0x50008000) //UART 2 Line control
#define rUCON2       (*(volatile unsigned *)0x50008004) //UART 2 Control
#define rUFCON2      (*(volatile unsigned *)0x50008008) //UART 2 FIFO control
#define rUMCON2      (*(volatile unsigned *)0x5000800c) //UART 2 Modem control
#define rUTRSTAT2    (*(volatile unsigned *)0x50008010) //UART 2 Tx/Rx status
#define rUERSTAT2    (*(volatile unsigned *)0x50008014) //UART 2 Rx error status
#define rUFSTAT2     (*(volatile unsigned *)0x50008018) //UART 2 FIFO status
#define rUMSTAT2     (*(volatile unsigned *)0x5000801c) //UART 2 Modem status
#define rUBRDIV2     (*(volatile unsigned *)0x50008028) //UART 2 Baud rate divisor

#ifdef __BIG_ENDIAN
#define rUTXH0      (*(volatile unsigned char *)0x50000023) //UART 0 Transmission Hold
#define rURXH0      (*(volatile unsigned char *)0x50000027) //UART 0 Receive buffer
#define rUTXH1      (*(volatile unsigned char *)0x50004023) //UART 1 Transmission Hold
#define rURXH1      (*(volatile unsigned char *)0x50004027) //UART 1 Receive buffer
#define rUTXH2      (*(volatile unsigned char *)0x50008023) //UART 2 Transmission Hold
#define rURXH2      (*(volatile unsigned char *)0x50008027) //UART 2 Receive buffer

#define WrUTXH0(ch) (*(volatile unsigned char *)0x50000023)=(unsigned char)(ch)
#define RdURXH0()  (*(volatile unsigned char *)0x50000027)
#define WrUTXH1(ch) (*(volatile unsigned char *)0x50004023)=(unsigned char)(ch)
```

```

#define RdURXH1()  (*(volatile unsigned char *)0x50004027)
#define WrUTXH2(ch) (*(volatile unsigned char *)0x50008023)=(unsigned char)(ch)
#define RdURXH2()  (*(volatile unsigned char *)0x50008027)

#define UTXH0      (0x50000020+3)  //Byte_access address by DMA
#define URXH0      (0x50000024+3)
#define UTXH1      (0x50004020+3)
#define URXH1      (0x50004024+3)
#define UTXH2      (0x50008020+3)
#define URXH2      (0x50008024+3)

#else //Little Endian
#define rUTXH0 (*(volatile unsigned char *)0x50000020) //UART 0 Transmission Hold
#define rURXH0 (*(volatile unsigned char *)0x50000024) //UART 0 Receive buffer
#define rUTXH1 (*(volatile unsigned char *)0x50004020) //UART 1 Transmission Hold
#define rURXH1 (*(volatile unsigned char *)0x50004024) //UART 1 Receive buffer
#define rUTXH2 (*(volatile unsigned char *)0x50008020) //UART 2 Transmission Hold
#define rURXH2 (*(volatile unsigned char *)0x50008024) //UART 2 Receive buffer

#define WrUTXH0(ch) (*(volatile unsigned char *)0x50000020)=(unsigned char)(ch)
#define RdURXH0()  (*(volatile unsigned char *)0x50000024)
#define WrUTXH1(ch) (*(volatile unsigned char *)0x50004020)=(unsigned char)(ch)
#define RdURXH1()  (*(volatile unsigned char *)0x50004024)
#define WrUTXH2(ch) (*(volatile unsigned char *)0x50008020)=(unsigned char)(ch)
#define RdURXH2()  (*(volatile unsigned char *)0x50008024)

#define UTXH0      (0x50000020)    //Byte_access address by DMA
#define URXH0      (0x50000024)
#define UTXH1      (0x50004020)
#define URXH1      (0x50004024)
#define UTXH2      (0x50008020)
#define URXH2      (0x50008024)

#endif

```

5. UART 初始化代码

下面列出的两个函数，是我们本实验用到的两个主要函数，包括 **UART** 初始化，字符的接收函数，希望大家仔细阅读，理解每一行的含义。这几个函数可以在例程目录下\common\include\2410lib.c 文件内找到。

```

void uart_init(int nMainClk, int nBaud, int nChannel)
{
    int i;

    if(nMainClk == 0)
        nMainClk    = PCLK;

```



```

switch (nChannel)
{
case UART0:
    rUFCON0 = 0x0;    //UART channel 0 FIFO control register, FIFO disable
    rUMCON0 = 0x0;    //UART chaneel 0 MODEM control register, AFC disable
    rULCON0 = 0x3;    //Line control register : Normal,No parity,1 stop,8 bits
    rUCON0  = 0x245;    // Control register
    rUBRDIV0=( (int)(nMainClk/16./nBaud+0.5) -1 );    // Baud rate divisor register 0
    break;

case UART1:
    rUFCON1 = 0x0;    //UART channel 1 FIFO control register, FIFO disable
    rUMCON1 = 0x0;    //UART chaneel 1 MODEM control register, AFC disable
    rULCON1 = 0x3;
    rUCON1  = 0x245;
    rUBRDIV1=( (int)(nMainClk/16./nBaud) -1 );
    break;

case UART2:
    rULCON2 = 0x3;
    rUCON2  = 0x245;
    rUBRDIV2=( (int)(nMainClk/16./nBaud) -1 );
    rUFCON2 = 0x0;    //UART channel 2 FIFO control register, FIFO disable
    break;

default:
    break;
}

for(i=0;i<100;i++);
delay(0);
}

```

下面是接收字符的实现函数：

```

/*****
* name:      uart_getch
* func:      Get a character from the uart
* para:      none
* ret:  get a char from uart channel
* modify:
* comment:
*****/

char uart_getch(void)
{
    if(f_nWhichUart==0)
    {

```

```

while(!(rUTRSTAT0 & 0x1)); //Receive data ready
return RdURXH0();
}
else if(f_nWhichUart==1)
{
while(!(rUTRSTAT1 & 0x1)); //Receive data ready
return RdURXH1();
}
else if(f_nWhichUart==2)
{
while(!(rUTRSTAT2 & 0x1)); //Receive data ready
return RdURXH2();
}
}
}

```

6. RS232 接口电路

本教学实验平台的电路中，UART0 串口电路如图 4-4-1 所示，UART0 只采用二根接线 RXD0 和 TXD0，因此只能进行简单的数据传输及接收功能。UART0 则采用 MAX3221E 作为电平转换器。

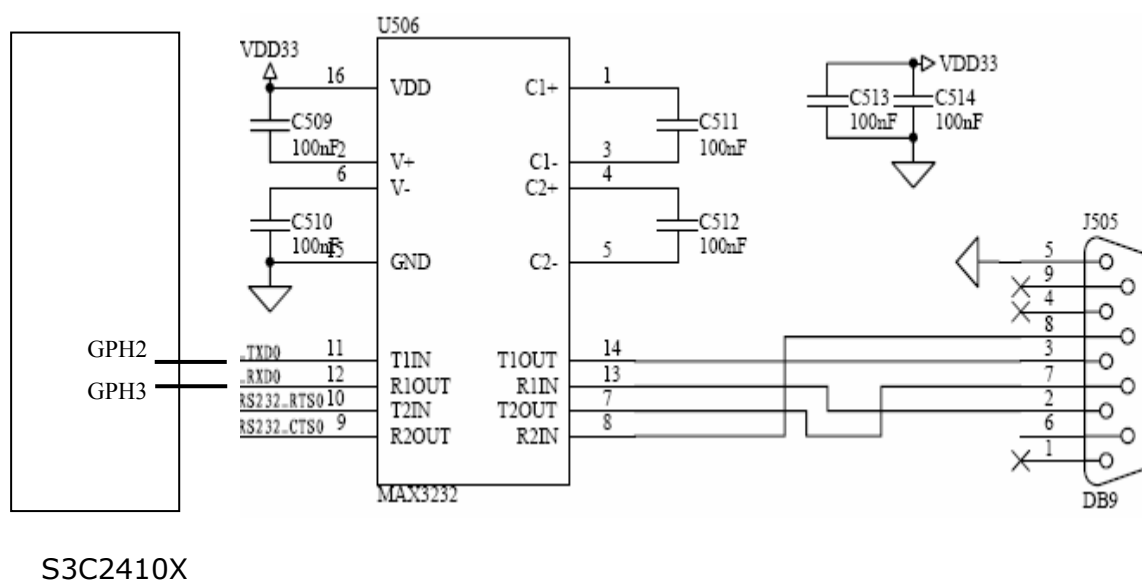


图 4-4-1 UART0 与 S3C2410 的连接图

4.4.5 实验操作步骤



1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接到目标板上。使用 Embest EduKit-III 实验板附带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制），或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板，打开实验例程目录 4.4_uart_test 子目录下的 uart_test.Uv2 例程，编译链接工程；
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境（工程默认已经配置正确），点击工具栏 “”，在 Option for Target 对话框的 Linker 页中选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug Session 项或点击工具栏 “”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；
- 4) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；
- 5) 在超级终端的 “Please input words that you want to transmit:” 提示后输入想要发送的数据，并已回车作为发送字符串的结尾标志。
- 6) 继续运行程序，直至程序的结尾。
- 7) 结合实验内容和实验原理部分，熟练掌握 S3C2410X 处理器 UART 模块的使用。

4. 观察实验结果

在执行到第 3)步中的第 4)步时，可以看到超级终端上输出等待输入字符，

```
boot success...

UART0 Communication Test Example

Please input words, then press Enter:

/>
```

如果输入字符就会马上显示在超级终端上（假设输入为 abcdefg），输入回车符后打印一整串字符：

```
The words that you input are:

abcdefg
```

5. 完成实验练习题

理解和掌握实验后，完成实验练习题。

4.4.6 实验参考程序

本实验的参考程序如下：

```
/*
*****
* File:      uart_communication.c
```

```

* Author:  embest
* Desc:    uart test file
* History:
*          R.X.Huang, Programming modify, March 12, 2005
*****/

/*          include files          */
#include "2410lib.h"

/*          function declare          */
/*****

* name:      uart0_test
* func:      uart test function
* para:      none
* ret: none
* modify:
* comment:
*****/

void uart0_test()
{
    char cInput[256];
    U8 ucInNo=0;
    char c;
    uart_init(0,115200,0);
    uart_printf("\n UART0 communication test!!\n");
    uart_printf(" Please input words that you want to transmit:\n");
    uart_printf(" ");
    g_nKeyPress = 1;
    while(g_nKeyPress==1)          // only for board test to exit
    {
        c=uart_getch();
        //uart_sendbyte(c);
        uart_printf("%c",c);
        if(c!='\r')
            cInput[ucInNo++]=c;
        else
        {
            cInput[ucInNo]='\0';
            break;
        }
    }
    delay(1000);
    uart_printf("The words that you input are:\n");
    uart_printf(cInput);
}

```

串口通信函数库中的其它函数:

```
void uart_getString(char *pString);
```

```
int uart_getintnum(void);
```

```
void uart_sendbyte(int nData);
```

```
void uart_sendstring(char *pString);
```

这些函数的详细定义，请参考\common\include\2410lib.c;

4.4.7 练习题

1. 编写程序实现在 LCD 上显示从串口接收到的字符。
2. 思考题：怎样在本例程的基础上，增加错误检测功能？

4.5 实时时钟实验

4.5.1 实验目的

- 了解实时时钟的硬件控制原理及设计方法。
- 掌握 S3C2410X 处理器的 RTC 模块程序设计方法。

4.5.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

4.5.3 实验内容

学习和掌握 Embest ARM 教学实验平台中 RTC 模块的使用，编写应用程序，修改时钟日期及时间的设置，以及使用 EMBEST ARM 教学系统的串口，在超级终端显示当前系统时间。

4.5.4 实验原理

1. 实时时钟（RTC）

实时时钟（RTC）器件是一种能提供日历/时钟、数据存储等功能的专用集成电路，常用作各种计算机系统的时钟信号源和参数设置存储电路。RTC 具有计时准确、耗电低和体积小等特点，特别是在各种嵌入式系统中用于记录事件发生的时间和相关信息，如通信工程、电力自动化、工业控制等自动化程度高的领域的无人值守环境。随着集成电路技术的不断发展，RTC 器件的新品也不断推出，这些新品不仅具有准确的 RTC，还有大容量的存储器、温度传感器和 A/D 数据采集通道等，已成为集 RTC、数据采集和存储于一体的综合功能器件，特别适用于以微控制器为核心的嵌入式系统。

RTC 器件与微控制器之间的接口大都采用连线简单的串行接口，诸如 I2C、SPI、MICROWIRE 和 CAN 等串行总线接口。这些串口由 2~3 根线连接，分为同步和异步。

2. S3C2410X 实时时钟（RTC）单元

S3C2410X 实时时钟（RTC）单元是处理器集成的片内外设。由开发板上的后备电池供电，可以在系统电源关闭的情况下运行。RTC 发送 8 位 BCD 码数据到 CPU。传送的数据包括秒、分、小时、星期、日期、月份和年份。RTC 单元时钟源由外部 32.768KHz 晶振提供，可以实现闹钟（报警）功能。

S3C2410X 实时时钟（RTC）单元特性：

- BCD 数据：秒、分、小时、星期、日期、月份和年份
- 闹钟（报警）功能：产生定时中断或激活系统
- 自动计算闰年

- 无 2000 年问题
- 独立的电源输入
- 支持毫秒级时间片中断，为 RTOS 提供时间基准

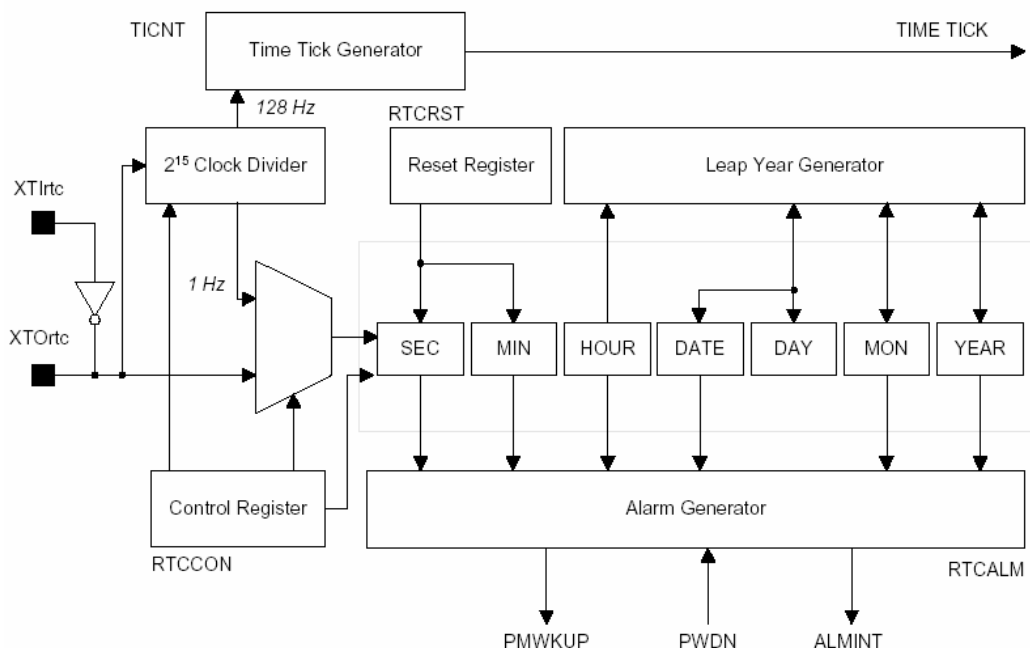


图 4-5-1 S3C2410X 处理器 RTC 功能框图

● 读/写寄存器

访问 RTC 模块的寄存器，首先要设 RTCCON 的 bit0 为 1。CPU 通过读取 RTC 模块中寄存器 BCDSEC、BCDMIN、BCDHOUR、BCDDAY、BCDDATE、BCDMON 和 BCDYEAR 的值，得到当前的相应时间值。然而，由于多个寄存器依次读出，所以有可能产生错误。比如：用户依次读取年（1989）、月（12）、日（31）、时（23）、分（59）、秒（59）。当秒数为 1 到 59 时，没有任何问题，但是，当秒数为 0 时，当前时间和日期就变成了 1990 年 1 月 1 日 0 时 0 分。这种情况下（秒数为 0），用户应该重新读取年份到分钟的值（参考程序设计）。

● 后备电池：

RTC 单元可以使用后备电池通过管脚 RTCVDD 供电。当系统关闭电源以后，CPU 和 RTC 的接口电路被阻断，后备电池只需要驱动晶振和 BCD 计数器，从而达到最小的功耗。

● 闹钟功能

RTC 在指定的时间产生报警信号，包括 CPU 工作在正常模式和休眠（power down）模式

下。在正常工作模式，报警中断信号（ALMINT）被激活。在休眠模式，报警中断信号和唤醒信号（PMWKUP）同时被激活。RTC 报警寄存器（RTCALM）决定报警功能的使能/屏蔽和完成报警时间检测。

● 时间片中断

RTC 时间片中断用于中断请求。寄存器 TICNT 有一个中断使能位和中断计数。该中断计数自动递减，当达到 0 时，则产生中断。中断周期按照下列公式计算：

$$\text{Period} = (n + 1) / 128 \text{ second}$$

其中，n 为 RTC 时钟中断计数，可取值为（1-127）

● 置零计数功能

RTC 的置零计数功能可以实现 30、40 和 50 秒步长重新计数，供某些专用系统使用。当使用 50 秒置零设置时，如果当前时间是 11:59:49，则下一秒后时间将变为 12:00:00。

注意：所有的 RTC 寄存器都是字节型的，必须使用字节访问指令（STRB、LDRB）或字符型指针访问。

4.5.5 实验设计

1. 硬件电路设计

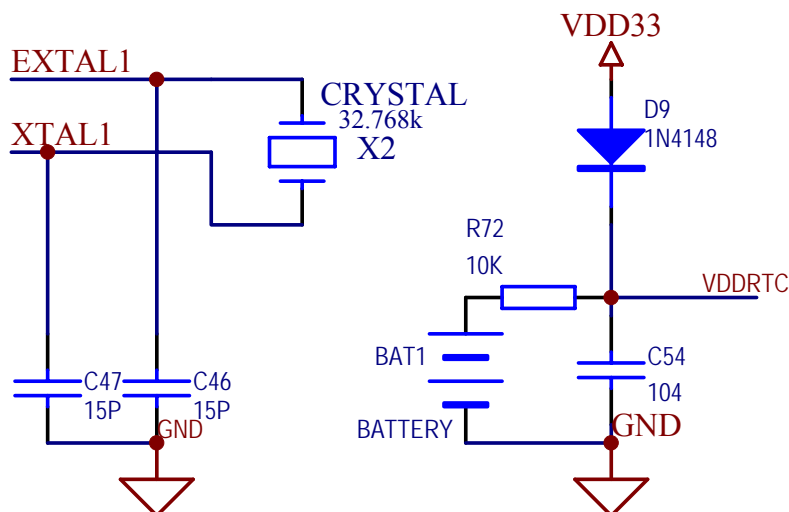


图 4-5-2 实时时钟外围电路

2. 软件程序设计

(1) 时钟设置

时钟设置程序须实现时钟工作情况以及数据设置有效性检测功能。具体实现可参考示例程序设计。

(2) 时钟显示

时钟参数通过实验系统串口 0 输出到超级终端，显示内容包括年月日时分秒。参数以 BCD 码形式传送，用户使用串口通信函数（参见串口通信实验）将参数取出显示。

```

/*****
* name:      rtc_display
* func:      display rtc value
* para:      none
* ret:       none
* modify:
* comment:
*****/

void rtc_display(void)
{
    INT32T nTmp;
    // INT32T nKey;
    uart_printf("\n Display current Date and time: \n");
    rRTCCON = 0x01;      // No reset, Merge BCD counters, 1/32768, RTC Control enable
    uart_printf(" Press any key to exit.\n");
}

```

```

while(!uart_getkey())
{
    while(1)
    {
        if(rBCDYEAR==0x99)
            g_nYear = 0x1999;
        else
            g_nYear = 0x2000 + rBCDYEAR;
        g_nMonth = rBCDMON;
        g_nWeekday = rBCDDAY;
        g_nDate = rBCDDATE;
        g_nHour = rBCDHOUR;
        g_nMin = rBCDMIN;
        g_nSec = rBCDSEC;

        if(g_nSec!=nTmp)    // Same time is not display
        {
            nTmp = g_nSec;
            break;
        }
    }
    uart_printf("%02x:%02x:%02x%10s, %02x/%02x/%04x\r",
                g_nHour,g_nMin,g_nSec,day[g_nWeekday],g_nMonth,g_nDate,g_nYear);
}

rRTCCON = 0x0; //No reset, Merge BCD counters, 1/32768, RTC Control disable(for power consumption)
uart_printf("\n\n Exit display.\n");
}

```

4.5.6 实验操作步骤


1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接到目标板上。使用 Embest EduKit-III 实验板附带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 运行实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板，打开实验例程目录 4.5_rtc_test 子目录下的 rtc_test.Uv2 例程，编译链接工程；
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境（工程默认已经配置正确），点击工具栏 “”，在 Option for Target 对话框的 Linker 页中

选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug Session 项或点击工具栏 “”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；

- 4) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；

- 5) 点击 Debug 菜单的 Go 或 F5 键运行程序。

4. 观察实验结果

- 1) 在 PC 机上观察超级终端程序主窗口，可以看到如下界面：

```
boot success...

RTC Test Example
RTC Check(Y/N)?Y
```

- 2) 用户可以选择是否对 RTC 进行检查，检查正确的话，继续执行程序，检查不正确时也会提示是否重检查：

```
RTC Alarm Test for S3C2410
3f:26:06
RTC Alarm Interrupt O.K.
3f:26:08
0. RTC Time Setting    1. Only RTC Display
Please Selet :
```

- 3) 用户可以选择是否重新进行时钟设置，当输入不正确时也会提示是否重新设置：

```
Please Selet : 0
Please input 0x and Two digit then press Enter, such as 0x99.
Year (0x?): 0x07
Month (0x?): 0x06
Date (0x?): 0x03
1:Sunday  2:Monday  3:Thesday  4:Wednesday  5:Thursday  6:Friday
7:Saturday
Day of week : 1
```

Hour (0x?): 0x21

Minute(0x?): 0x30

Second(0x?): 0x01

4) 最终超级终端输出信息如下:

Display current date and time:

Press any key to exit.

21:30:01 Sunday, 06/03/2007

Exit display.

Press any key to exit.

5. 完成实验练习题

理解和掌握实验后, 完成实验练习题。

4.5.7 实验参考程序

1. RTC 报警控制程序

```

/*****
* name:    rtc_alarm_test
* func:    test rtc alarm
* para:    none
* ret:     f_nlsRtcInt = 0 : rtc not work
*          f_nlsRtcInt = 1 : rtc work fine
* modify:
* comment:
*****/

int rtc_alarm_test(void)
{
    // INT32T g_nHour=0xff0000,g_nMin=0xff00,g_nSec=0xff;
    uart_printf(" RTC Alarm Test for S3C2410 \n");

    // rtc_init();
    rRTCCON  = 0x01;          // No reset, Merge BCD counters, 1/32768, RTC Control enable
    rALMYEAR = rBCDYEAR ;
    rALMMON  = rBCDMON;
    rALMDATE = rBCDDATE ;
    rALMHOUR = rBCDHOUR ;
    rALMMIN  = rBCDMIN  ;
    rALMSEC  = rBCDSEC + 2;
    f_nlsRtcInt = 0;
    pISR_RTC = (unsigned int)rtc_int;
    rRTCALM  = 0x7f;          //Global,g_nYear,g_nMonth,Day,g_nHour,Minute,Second alarm enable
    rRTCCON  = 0x0;          // No reset, Merge BCD counters, 1/32768, RTC Control disable
    rINTMSK &= ~(BIT_RTC);

```

```

    uart_printf(" %02x:%02x:%02x\n",rBCD HOUR,rBCD MIN,rBCD SEC);
    // while(f_nIsRtcInt==0);
    delay(21000);          // delay 2.1s
    uart_printf(" %02x:%02x:%02x\n",rBCD HOUR,rBCD MIN,rBCD SEC);

    rINTMSK |= BIT_RTC;
    rRTCCON  = 0x0;        // No reset, Merge BCD counters, 1/32768, RTC Control disable
    return f_nIsRtcInt;
}

```

2. 时钟设置控制程序

```

/*****
* name:    rtc_set
* func:    set a new g_nDate & time to rtc
* para:    none
* ret:     none
* modify:
* comment:
*****/
void rtc_set(void)
{
    uart_printf("\n Please input 0x and Two digit then press Enter, such as 0x99.\n");
    uart_printf(" Year  (0x?): ");
    g_nYear = uart_getintnum();

    uart_printf(" Month (0x?): ");
    g_nMonth = uart_getintnum();

    uart_printf(" Date  (0x?): ");
    g_nDate = uart_getintnum();

    uart_printf("\n 1:Sunday 2:Monday 3:Thuesday 4:Wednesday 5:Thursday 6:Friday 7:Saturday\n");
    uart_printf(" Day of week : ");
    g_nWeekday = uart_getintnum();

    uart_printf("\n Hour  (0x?): ");
    g_nHour = uart_getintnum();

    uart_printf(" Minute(0x?): ");
    g_nMin = uart_getintnum();

    uart_printf(" Second(0x?): ");
    g_nSec = uart_getintnum();
}

```

```

rRTCCON = rRTCCON & ~(0xf) | 0x1; // No reset, Merge BCD counters, 1/32768, RTC Control enable

rBCDYEAR = rBCDYEAR & ~(0xff) | g_nYear;
rBCDMON = rBCDMON & ~(0x1f) | g_nMonth;
rBCDDAY = rBCDDAY & ~(0x7) | g_nWeekday; // SUN:1 MON:2 TUE:3 WED:4 THU:5 FRI:6 SAT:7
rBCDDATE = rBCDDATE & ~(0x3f) | g_nDate;
rBCDHOUR = rBCDHOUR & ~(0x3f) | g_nHour;
rBCDMIN = rBCDMIN & ~(0x7f) | g_nMin;
rBCDSEC = rBCDSEC & ~(0x7f) | g_nSec;

rRTCCON = 0x0; // No reset, Merge BCD counters, 1/32768, RTC Control disable
}

```

4.5.8 练习题

编写程序检测 RTC 的时间片（RTC_Tick）功能及置零计数功能。

4.6 数码管显示实验

4.6.1 实验目的

- 通过实验掌握 LED 的显示控制方法。
- 通过实验加深对 IIC 总线工作原理的掌握。

4.6.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

4.6.3 实验内容

编写程序使实验板上八段数码管循环显示 0 到 9 字符。

4.6.4 实验原理

1. 八段数码管

嵌入式系统中，经常使用八段数码管来显示数字或符号，由于它具有显示清晰、亮度高、使用电压低、寿命长的特点，因此使用非常广泛。

● 结构

八段数码管由八个发光二极管组成，其中七个长条形的发光管排列成“日”字形，右下角一个点形的发光管作为显示小数点用，八段数码管能显示所有数字及部份英文字母。见图 4-6-1。

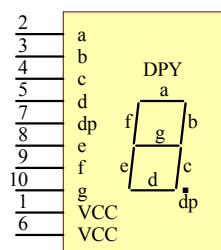


图 4-6-1 八段数码管的结构

● 类型

八段数码管有两种不同的形式：一种是八个发光二极管的阳极都连在一起的，称之为共阳极八段数码管；另一种是八个发光二极管的阴极都连在一起的，称之为共阴极八段数码管。

● 工作原理

以共阳极八段数码管为例，当控制某段发光二极管的信号为低电平时，对应的发光二极管点亮，当需要显示某字符时，就将该字符对应的所有二极管点亮；共阴极二极管则相反，控制信号为高电平时点亮。

电平信号按照 **dp, g, e...a** 的顺序组合形成的数据字称为该字符对应的段码，常用字符的段码表如下：

表 4-6-1 常用字符的段码表

字符	dp	g	f	e	d	c	b	a	共阴极	共阳极
0	0	0	1	1	1	1	1	1	3FH	C0H
1	0	0	0	0	0	1	1	0	06H	F9H
2	0	1	0	1	1	0	1	1	5BH	A4H
3	0	1	0	0	1	1	1	1	4FH	B0H
4	0	1	1	0	0	1	1	0	66H	99H
5	0	1	1	0	1	1	0	1	6DH	92H
6	0	1	1	1	1	1	0	1	7DH	82H
7	0	0	0	0	0	1	1	1	07H	F8H
8	0	1	1	1	1	1	1	1	7FH	80H
9	0	1	1	0	1	1	1	1	6FH	90H
A	0	1	1	1	0	1	1	1	77H	88H
B	0	1	1	1	1	1	0	0	7CH	83H
C	0	0	1	1	1	0	0	1	39H	C6H
D	0	1	0	1	1	1	1	0	5EH	A1H
E	0	1	1	1	1	0	0	1	79H	86H
F	0	1	1	1	0	0	0	1	71H	8EH
-	0	1	0	0	0	0	0	0	40H	BFH
.	1	0	0	0	0	0	0	0	80H	7FH
熄灭	0	0	0	0	0	0	0	0	00H	FFH

● 显示方式

八段数码管的显示方式有两种，分别是静态显示和动态显示。

静态显示是指当八段数码管显示一个字符时，该字符对应段的发光二极管控制信号一直保持有效。

动态显示是指当八段数码管显示一个字符时，该字符对应段的发光二极管是轮流点亮的，即控制信号按一定周期有效，在轮流点亮的过程中，点亮时间是极为短暂的（约 1ms），由于人的视觉暂留现象及发光二极管的余辉效应，数码管的显示依然是非常稳定的。

2. 电路原理

Embest EduKit-III 教学电路中使用的是共阴极八段数码管。数码管的显示由芯片 ZLG7290 进行控制(具体的控制原理和方法可以参考 5.2 节的 5X4 键盘控制实验)，它的 DIG1~DIG8 引脚输出 LED 显示所需的位驱动信号，而 SEGA~SEGG 及 DP 引脚输出 LED 显示所需的段驱动信号。相关电路见图 4-6-2 和图 4-6-3。

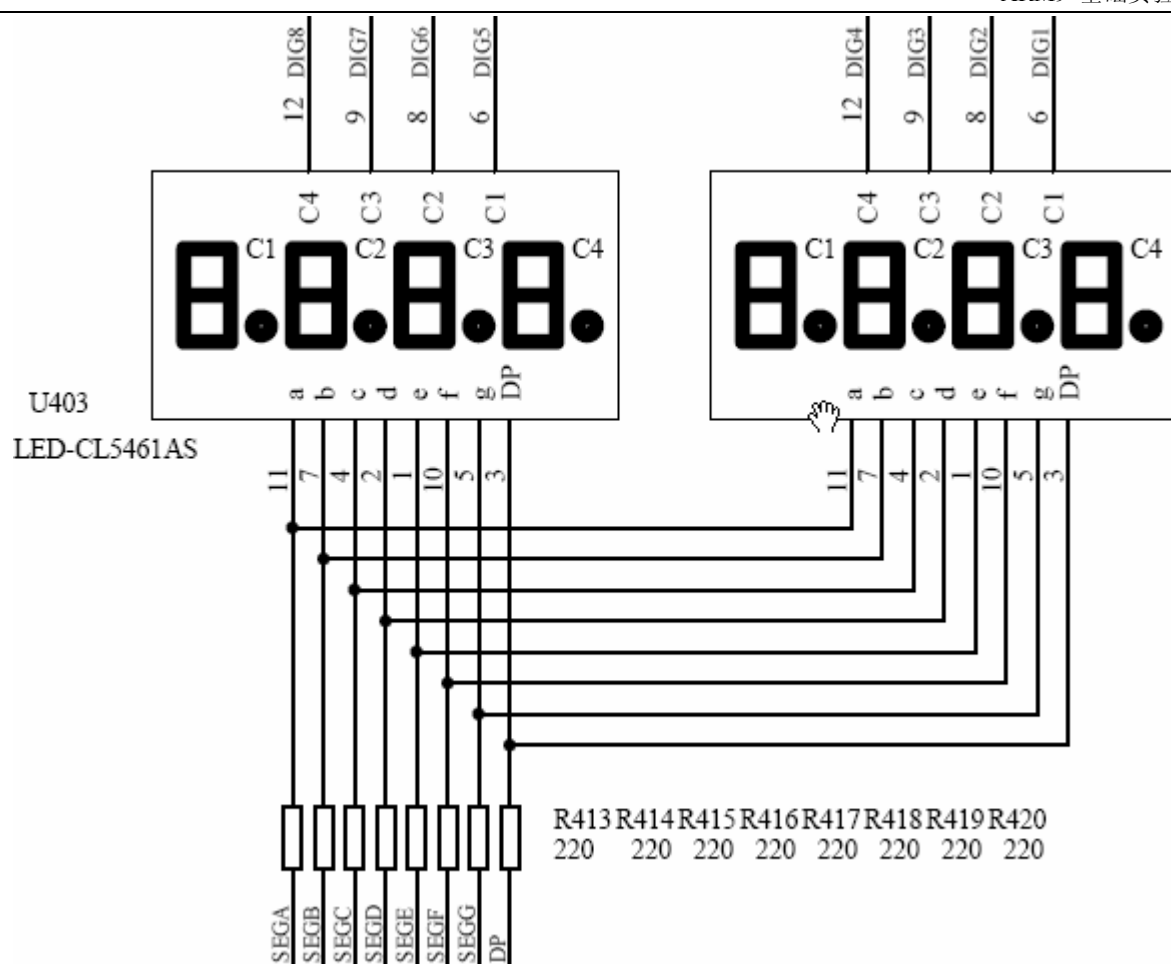


图 4-6-2 八段数码管连接电路

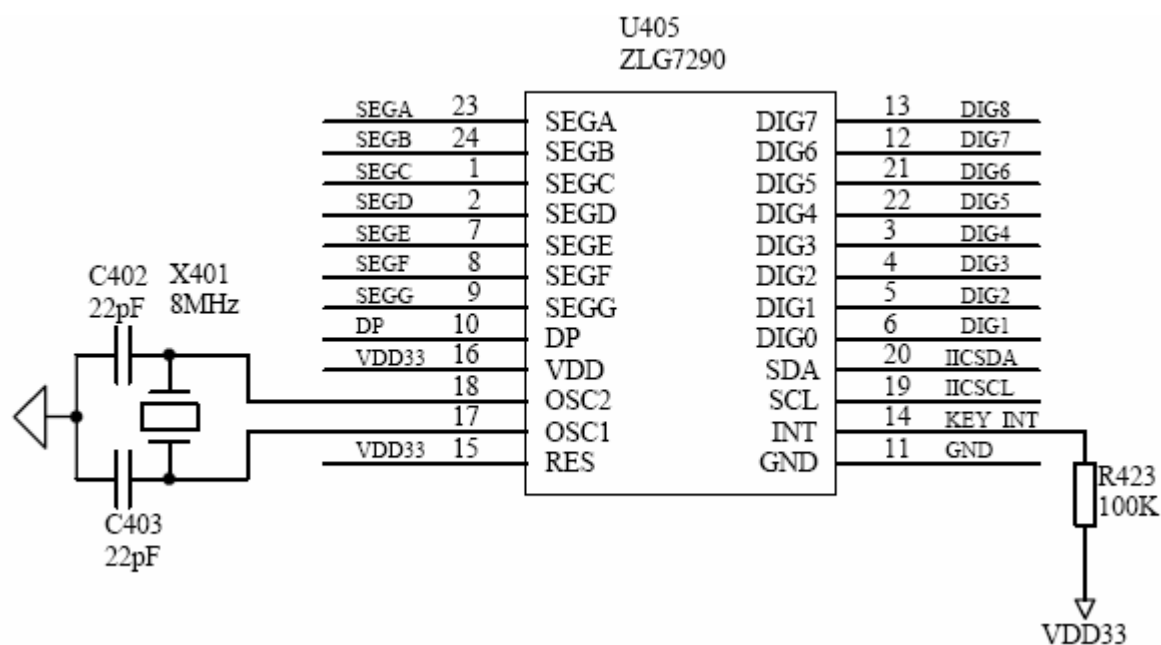


图 4-6-3 八段数码管控制电路

3. 软件程序设计

八段数码管上显示的字符段码储存在数组 `f_szDigital[]` 中，通过 IIC 总线依次将他们写入到 ZL G7290 芯片的显示缓存寄存器（DpRam0~DpRam7）中即可完成字符在 8 段数码管上的显示。

1) IIC 中断使能设置

由于 LED 的段码是通过 IIC 总线传输的，需要采用中断方式来检测每个字节的传输，所以需要定义中断处理程序入口，使能中断。具体的实现方法如下：

```
rINTMSK = rINTMSK & (~(BIT_ALLMSK|BIT_IIC)); //使能中断
pISR_IIC= (unsigned)iic_int; //将 IIC 中断处理程序指针指向 iic_int
```

2) 初始化 IIC 接口

初始化 IIC 接口就是对 IIC 的相关寄存器进行初始设置。如下所示：

```
rIICADD = 0x10; // S3C2410X 从设备地址
rIICCON = 0xef; // 使能 ACK 和 IIC 总线中断，设置 IICCLK 为
                // MCLK/512，清除 pending 位以便响应中断。
rIICSTAT= 0x10; // 使能发送/接收中断
```

3) iic_write () 函数介绍

函数原型：void iic_write(UINT32T unSlaveAddr,UINT32T unAddr,UINT8T ucData)

参数说明：unSlaveAddr --- 输入，IIC 从设备地址（ZLG7290 地址为 0X70H）

unAddr --- 输入，数据地址（即 ZLG7290 显示缓冲区地址
0X10H~0X17H）

ucData --- 输入，数据值

函数返回值：NULL

所以，通过函数 `iic_write(0x70, 0x10+i, f_szDigital[k])` 即可将数组 `f_szDigital[]` 中的第 `k+1` 个元素为段码的字符显示在第 `i` 个 8 段数码管上。

4.6.5 实验方法与操作步骤


1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接到目标板上。使用 Embest EduKit-III 实验板附带的串口线，连接实验板上的 UART0 和 PC 机的串口。


2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板，打开实验例程目录 4.6_8led_test 子目录下的 8led_test.Uv2 例程，编译链接工程成功；
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境（工程默认已经配置正确），点击工具栏 “”，在 Option for Target 对话框的 Linker 页中

选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug

Session 项或点击工具栏 “”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；

- 4) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；

- 5) 点击 Debug 菜单的 Go 或 F5 键运行程序。

4. 观察实验结果

- 1) 在 PC 机上观察超级终端程序主窗口，可以看到如下界面：

boot success...

8-segment Digit LED Test Example (Please look at LED)

- 2) 实验系统八段数码管循环显示 0 ~ 9 字符。

5. 完成实验练习题

理解和掌握实验后，完成实验练习题。

4.6.6 实验参考程序

1. LED 测试程序

```

/*****
* name:      led8_test
* func:      test 8led
* para:      none
* ret:       none
* modify:
* comment:
*****/

void led8_test(void)
{
    int i, j, k;
    iic_init();
    for(;;)
    {
        for(j=0; j<10; j++)
        {
            for(i=0; i<8; i++)
            {
                k = 9-(i+j)%10;
                iic_write(0x70, 0x10+i, f_szDigital[k]);
            }
            delay(10000);
        }
    }
}

```


}

2. IIC 总线读写程序

```

/*****
* name:      iic_write_8led()
* func:      write data to iic
* para:      unSlaveAddr --- input, chip slave address
*            unAddr    --- input, data address
*            ucData    --- input, data value
* ret:      none
* modify:
* comment:
*****/

void iic_write_8led(UINT32T unSlaveAddr,UINT32T unAddr,UINT8T ucData)
{
    f_nGetACK = 0;

    // Send control byte
    rIICDS = unSlaveAddr;                // 0x70
    rIICSTAT = 0xf0;                    // Master Tx,Start
    while(f_nGetACK == 0);              // Wait ACK
    f_nGetACK = 0;

    // Send address
    rIICDS = unAddr;
    rIICCON = 0xef;                    // Resumes IIC operation.
    while(f_nGetACK == 0);              // Wait ACK
    f_nGetACK = 0;

    // Send data
    rIICDS = ucData;
    rIICCON = 0xef;                    // Resumes IIC operation.
    while(f_nGetACK == 0);              // Wait ACK
    f_nGetACK = 0;

    // End send
    rIICSTAT = 0xd0;                    // Stop Master Tx condition
    rIICCON = 0xef;                    // Resumes IIC operation.
    delay(5);                          // Wait until stop condtion is in effect.
}

/*****
* name:      iic_read_8led()
* func:      read data from iic
* para:      unSlaveAddr --- input, chip slave address
*            unAddr      --- input, data address
*            pData       --- output, data pointer
* ret:      none
*****/

```

```

* modify:
* comment:
*****/

void iic_read_8led(UINT32T unSlaveAddr,UINT32T unAddr,UINT8T *pData)
{
    char cRecvByte;
    f_nGetACK = 0;

    // Send control byte
    rIICDS = unSlaveAddr;                                // Write slave address to IICDS
    rIICSTAT = 0xf0;                                       // Master Tx,Start
    while(f_nGetACK == 0);                                  // Wait ACK
    f_nGetACK = 0;

    // Send address
    rIICDS = unAddr;
    rIICCON = 0xef;                                         // Resumes IIC operation.
    while(f_nGetACK == 0);                                  // Wait ACK
    f_nGetACK = 0;

    // Send control byte
    rIICDS = unSlaveAddr;                                  // 0x70
    rIICSTAT = 0xb0;                                       // Master Rx,Start
    rIICCON = 0xef;                                         // Resumes IIC operation.
    while(f_nGetACK == 0);                                  // Wait ACK
    f_nGetACK = 0;

    // Get data
    cRecvByte = rIICDS;
    rIICCON = 0x2f;
    delay(1);

    // Get data
    cRecvByte = rIICDS;
    // End receive
    rIICSTAT = 0x90;                                       // Stop Master Rx condition
    CCON = 0xef;                                           // Resumes IIC operation.
    delay(5);                                              // Wait until stop condtion is in effect.

    *pData = cRecvByte;
}

```

4.6.7 练习题

编写程序循环显示八段数码管的各段。

4.7 看门狗实验

4.7.1 实验目的

- 了解看门狗的作用
- 掌握处理器 S3C2410X 处理器看门狗控制器的使用

4.7.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

4.7.3 实验内容

通过使用 S3C2410 处理器集成的看门狗模块，对其进行如下操作：

- 掌握看门狗的操作方式和用途。
- 对看门狗模块进行软件编程，实现看门狗的定时功能和复位功能。

4.7.4 实验原理

1. 看门狗概述

看门狗的作用是微控制器受到干扰进入错误状态后，使系统在一定时间间隔内复位。因此看门狗是保证系统长期、可靠和稳定运行的有效措施。目前大部分的嵌入式芯片内都集成了看门狗定时器来提高系统运行的可靠性。

S3C2410 处理器的看门狗

S3C2410 处理器的看门狗是当系统被故障如噪声或者系统错误干扰时，用于微处理器的复位操作的。也可以作为一个通用的 16 位定时器来请求中断操作。看门狗定时器产生 128 个 PCLK 周期的复位信号。主要特性如下：

通用的中断方式的 16 位定时器。

当计数器减到 0（发生溢出），产生 128 个 PLK 周期的复位信号。

看门狗定时器的操作：

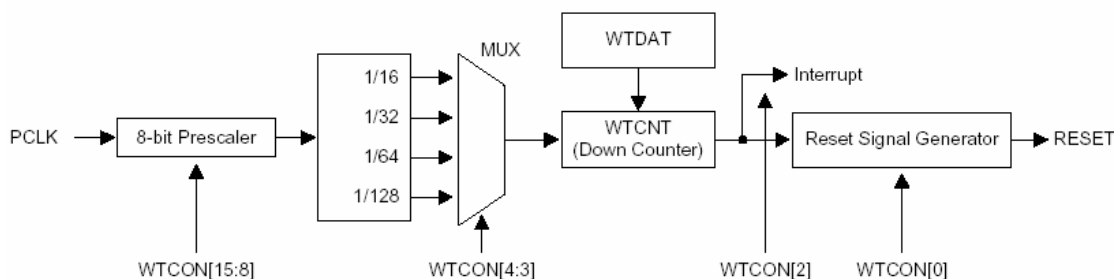


图 4-7-1 S3C2410X 看门狗的功能框图

看门狗模块包括一个预比例因子放大器，一个四分频的分频器，一个 16 位计数器。看门狗的时钟信号来自 PCLK，为了得到宽范围的看门狗信号，PCLK 先被预分频，之后再经过分频器分频。预分频比例因子和分频器的分频值，都可以由看门狗控制寄存器（WTCN）决定，预分频比例因子的范围由 0 到 255，分频器的分频比可以是 16，32，64 或者 128。

看门狗定时器时钟周期的计算

计算公式 $t_{\text{watchdog}} = 1 / (\text{PCLK} / (\text{Prescaler value} + 1) / \text{Division_factor})$

式中 Prescaler value 为预分频比例放大器的值。

Division_factor 是四分频的分频比，可以是 16，32，64 或者 128。

一旦看门狗定时器被允许，看门狗定时器数据寄存器（WTDAT）的值不能被自动的装载到看门狗计数器（WTCNT）中。因此，看门狗启动前要将一个初始值写入看门狗计数器（WTCNT）中。

调试环境下的看门狗

当 S3C2410X 用嵌入式 ICE 调试的时候，看门狗定时器的复位功能不被启动，看门狗定时器能从 cpu 内核信号判断出当前 cpu 是否处于调试状态。如果看门狗定时器确定当前模式是调试模式，尽管看门狗能产生溢出信号，但是仍然不会产生复位信号。

2. 看门狗定时器寄存器组

1) 看门狗定时器控制寄存器 (WTCON)

WTCON 寄存器的内容包括：用户是否启用看门狗定时器、4 个分频比的选择、是否允许中断产生、是否允许复位操作等。

如果用户想把看门狗定时器当作一般的定时器使用，应该中断使能，禁止看门狗定时器复位。

寄存器	地址	读/写	描述	复位值
WTCON	0x53000000	读/写	看门狗定控制寄存器	0x8021

表 4-7-1 WTCON 位描述

WTCON	位	描 述	复位值
预装比例因子	15:8	预装比例值，有效范围值 0~255	0x80
保留	7:6	保留	00
看门狗使能	5	使能和禁止看门狗定时器 0=禁止看门狗定时器 1=使能看门狗定时器	0
时钟选择	4:3	这两位决定时钟分频因素 00:1/16 01:1/32 10:1/64 11:1/128	00
中断使能	2	中断的禁止和使能 0=禁止中断产生 1=使能中断产生	0
保留	1	保留	0
复位使能	0	禁止和使能看门狗复位信号的输出 1=看门狗复位信号使能 0=看门狗复位信号禁止	1

2) 看门狗定时器数据寄存器 (WTDAT)

WTDAT 用于指定超时时间，在初始化看门狗操作后看门狗数据寄存器的值不能被自动装载到看门狗计数寄存器 (WTCNT) 中。然而，如果初始值为 0x8000，可以自动装载 WTDAT 的值到 WTCNT 中。

寄存器	地址	读/写	描述	复位值
WTDAT	0x53000004	读/写	看门狗数据寄存器	0x8000

3) 看门狗计数寄存器 (WTCNT)

WTCNT 包含看门狗定时器工作的时候计数器的当前计数值。注意在初始化看门狗操作后看门狗数据寄存器的值不能被自动装载到看门狗计数寄存器 (WTCNT) 中, 所以看门狗被允许之前应该初始化看门狗计数寄存器的值。

寄存器	地址	读/写	描述	复位值
WTCNT	0x53000008	读/写	看门狗计数器的当前值	0x8000

4.7.5 实验设计

1. 软件程序设计

由于看门狗是对系统的复位或者中断的操作, 所以不需要外围的硬件电路。要实现看门狗的功能, 只需要对看门狗的寄存器组进行操作。即对看门狗的控制寄存器 (WTCN)、看门狗数据寄存器 (WTDAT)、看门狗计数寄存器 (WTCNT) 的操作。

一般流程如下:

1) 设置看门狗中断操作包括全局中断和看门狗中断的使能, 和看门狗中断向量的定义, 如果只是进行复位操作, 不需要复位操作, 这一步可以不用设置。

2) 对看门狗控制寄存器 (WTCN) 的设置, 包括设置预分频比例因子、分频器的分频值、中断使能和复位使能等。

3) 对看门狗数据寄存器 (WTDAT) 和看门狗计数寄存器 (WTCNT) 的设置。

4) 启动看门狗定时器。

2. 看门狗在函数 delay () 中的使用。

```

/*****
* name:      delay
* func:      delay time, 100us resolution.
* para:      nTime -- input,
*             nTime=0: nAdjust the delay function by WatchDog timer.
*             nTime>0: the number of loop time,
* ret:      none
* modify:
* comment:
*****/

void delay(int nTime)
{
    int i,adjust=0;
    if(nTime==0)
    {
        nTime = 200;
        adjust = 1;
        delayLoopCount = 400;
        //PCLK/1M,Watch-dog disable,1/64,interrupt disable,reset disable
        rWTCN = ((PCLK/1000000-1)<<8)|(2<<3);
        rWTDAT = 0xffff;                // for first update
        rWTCNT = 0xffff;                // resolution=64us @any PCLK
    }
}

```

```

        rWTCON = ((PCLK/1000000-1)<<8)|(2<<3)|(1<<5); // Watch-dog timer start
    }
    for(;nTime>0;nTime--)
        for(i=0;i<delayLoopCount;i++);
    if(adjust==1)
    {
        rWTCON = ((PCLK/1000000-1)<<8)|(2<<3); // Watch-dog timer stop
        i = 0xffff - rWTCNT;
        //1count->64us, 200*400 cycle runtime = 64*i us
        delayLoopCount = 8000000/(i*64);    // 200*400:64*i=1*x:100 -> x=80000*100/(64*i)
    }
}

```

nTime=0 的时候调整 delayLoopCount 的值,在调整的过程中把看门狗看作一个普通的计数器。然后根据计数器中的数据来确定延时 100us 时, delayLoopCount 的值。根据程序和看门狗时钟频率输出可知,在 ntime=200,delayLoopCount=400 的时间内,看门狗计数器记了 i 个脉冲,看门

狗的时钟频率为 $\frac{1M}{64}$ Hz, , 所以可以计算出一个 delayLoopCount 需要的时间,即一个 for(i=0;i<1;i++);的时间为:

$$t = \frac{64 \times 10^{-6} \times i}{200 \times 400}$$

因此,要延时 100us,delayLoopCount 的值应该为:

$$\text{delayLoopCount} = \frac{100 \times 10^{-6}}{t} = \frac{100 \times 10^{-6} \times 80000}{64 \times 10^{-6} \times i} = \frac{8000000}{64 \times i}$$

4.7.6 实验操作步骤


1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接到目标板上。使用 Embest EduKit-III 实验板附带的串口线,连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序(波特率 115200、1 位停止位、无校验位、无硬件流控制);或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下(如果已经拷贝,可跳过此步骤);
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板,打开实验例程目录 4.7_watchdog_test 子目录下的 watchdog_test.Uv2 例程,编译链接工程;
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境(工程默认已经配置正确), 点击工具栏 “”, 在 Option for Target 对话框的 Linker 页中

选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug Session 项或点击工具栏 “”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；

- 4) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；

- 5) 点击 Debug 菜单的 Go 或 F5 键运行程序。

4. 观察实验结果

在 PC 机的超级终端的主窗口中观察实验的结果如下：

```
boot success...

WatchDog Timer Test Example
10 seconds:
1s  2s  3s  4s  5s  6s  7s  8s  9s  10s

O.K. end.
```

5. 完成实验练习题

理解和掌握实验后，完成实验练习题。

4.7.7 实验参考程序

```
/******
 * name:      watchdog_test
 * func:      watchdog timer test function
 * para:      none
 * ret:       none
 * modify:
 * comment:
 *****/

void watchdog_test(void)
{
    //Initialize interrupt registers
    rSRCPND |= 0x200;
    rINTPND |= 0x200;
    //Initialize WDT registers
    pISR_WDT = (unsigned)watchdog_int;
    rWTCON = (PCLK/(100000-1)<<8)|(3<<3)|(1<<2);// 1M,1/128, enable interrupt
    rWTDAT = 781;
    rWTCNT = 781;
    rWTCON |= (1<<5);           // start watchdog timer
    rINTMOD &= 0xFFFFFDFF;
    rINTMSK &= 0xFFFFFDFF;
    while((f_ucSecondNo)<11);
```

```
}  
/*****  
* name:      watchdog_int  
* func:      watchdog interrupt service routine  
* para:      none  
* ret:       none  
* modify:  
* comment:  
*****/  
void watchdog_int(void)  
{  
    ClearPending(BIT_WDT);  
    f_ucSecondNo++;  
    if(f_ucSecondNo<11)  
        uart_printf(" %ds ",f_ucSecondNo);  
    else  
        uart_printf("\n O.K.");  
}
```

4.7.8 实验练习题

参考实验例程，重新调整看门狗定时器的预分频值和分频器的分频值，让看门狗定时器每两秒发生一次中断。并在五秒后复位。

第五章 人机接口实验

5.1 液晶显示实验

5.1.1 实验目的

- 初步掌握液晶屏的使用及其电路设计方法。
- 掌握 S3C2410X 处理器的 LCD 控制器的使用。
- 通过实验掌握液晶显示文本及图形的方法与程序设计。

5.1.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

5.1.3 实验内容

通过使用 Embest EduKit-III 实验板的 256 色彩色液晶屏（320x240）进行电路设计，掌握液晶屏作为人机接口界面的设计方法，并编写程序实现：

画出多个矩形框

显示 ASCII 字符

显示汉字字符

显示彩色位图

5.1.4 实验原理

1. 液晶显示屏（LCD）

液晶屏（LCD: Liquid Crystal Display）主要用于显示文本及图形信息。液晶显示屏具有轻薄、体积小、低耗电量、无辐射危险、平面直角显示以及影像稳定不闪烁等特点，因此在许多电子应用系统中，常使用液晶屏作为人机界面。

● 主要类型及性能参数

液晶显示屏按显示原理分为 STN 和 TFT 两种：

STN（Super Twisted Nematic，超扭曲向列）液晶屏

STN 液晶显示器与液晶材料、光线的干涉现象有关，因此显示的色调以淡绿色与橘色为主。STN 液晶显示器中，使用 X、Y 轴交叉的单纯电极驱动方式，即 X、Y 轴由垂直与水平方向的驱动电极构成，水平方向驱动电压控制显示部分为亮或暗，垂直方向的电极则负责驱动液晶分子的显示。STN 液晶显示屏加上彩色滤光片，并将单色显示矩阵中的每一像素分成三个子像素，分别通过彩色滤光片显示红、绿、蓝三原色，也可以显示出色彩。单色液晶屏及灰度液晶屏都是 STN 液晶屏。

TFT（Thin Film Transistor，薄膜晶体管）彩色液晶屏

随着液晶显示技术的不断发展和进步，TFT 液晶显示屏被广泛用于制作成电脑中的液晶显示设备。TFT 液晶显示屏既可在笔记本电脑上应用（现在大多数笔记本电脑都使用 TFT 显示屏），也常用于主流台式显示器。

使用液晶显示屏时，主要考虑的参数有外形尺寸、分辨率、点宽、色彩模式等。以下是 Embest EduKit-III 实验板所选用的液晶屏（LRH9J515XA STN/BW）主要参数：

表 5-1-1 LRH9J515XA STN/BW 液晶屏主要技术参数

型号	LRH9J515XA	外形尺寸	93.8×75.1×5mm	重量	45g
像素	320 × 240	画面尺寸	9.6cm (3.8inch)	色彩	16 级灰度
电压	21.5V (25℃)	点宽	0.24 mm/dot	附加	带驱动逻辑

可视屏幕的尺寸及参数示意如图 5-1-1 所示：

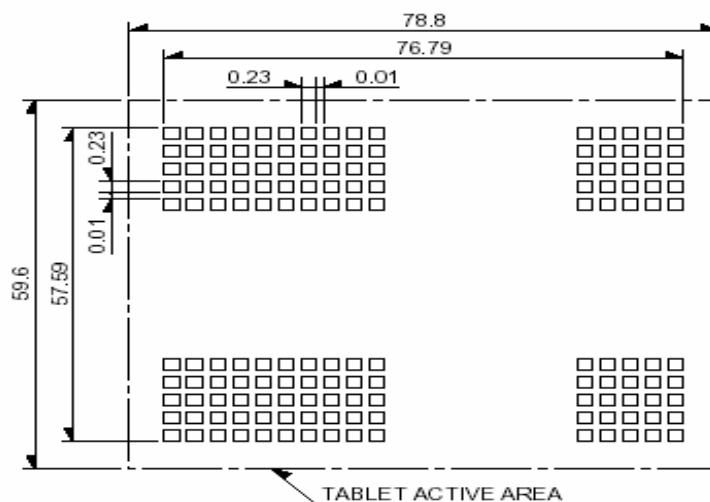


图 5-1-1 液晶屏参数示意图

液晶屏外形如图 5-1-2 所示：

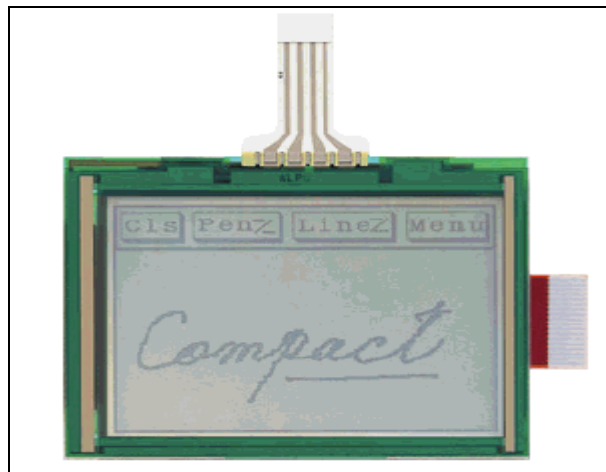


图 5-1-2 LRH9J515XA STN/BW 液晶屏外形

● 驱动与显示

液晶屏的显示要求设计专门的驱动与显示控制电路。驱动电路包括提供液晶屏的驱动电源和液晶分子偏置电压，以及液晶显示屏的驱动逻辑；显示控制部分可由专门的硬件电路组成，也可以采用集成电路（IC）模块，比如 EPSON 的视频驱动器等；还可以使用处理器外围 LCD 控制模块。实验板的驱动与显示系统包括 S3C2410X 片内外设 LCD 控制器、液晶显示屏的驱动逻辑以及外围驱动电路。

2. S3C2410X LCD 控制器

● LCD 控制器特点

S3C2410X 处理器集成了 LCD 控制器，主要功能是 S3C2410X LCD 控制器用于传输显示数据和产生控制信号。它并支持屏幕水平和垂直滚动显示。数据的传送采用 DMA（直接内存访问）方式，以达到最小的延迟。它可以支持多种液晶屏：

STN LCD:

- 支持3种类型的扫描方式：4位单扫描，4位双扫描和8位单扫描
- 支持单色，4级灰度和16级灰度显示
- 支持256色和4096色彩色STN LCD
- 支持多种屏幕大小

典型的实际屏幕大小是：640×480，320×240，160×160和其它

最大虚拟屏幕占内存大小为4M字节

256色模式下最大虚拟屏幕大小：4096×1024，2048×2048，1024×4096和其它

TFT LCD:

- 支持1，2，4或8bpp彩色调色显示
- 支持16bpp和24bpp非调色真彩显示
- 在24bpp模式下，最多支持16M种颜色
- 支持多种屏幕大小

典型的实际屏幕大小是：640×480，320×240，160×160和其它

最大虚拟屏幕占内存大小为4M字节

64K色模式下最大虚拟屏幕大小：2048×1024和其它

● LCD 控制器内部结构

LCD 控制器主要提供液晶屏显示数据的传送、时钟和各种信号的产生与控制功能。S3C2410X 处理器的 LCD 控制器主要部分框图如图 5-3 所示：

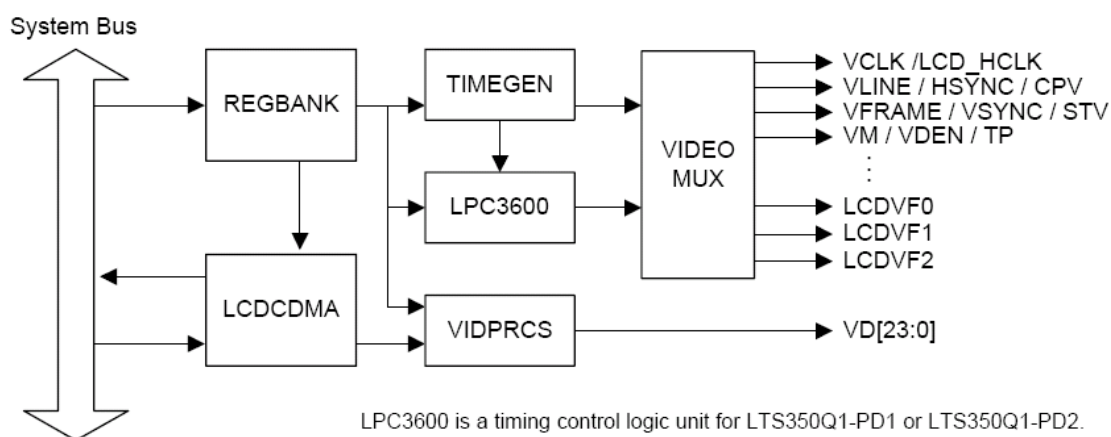


图 5-1-3 LCD 控制器框图

S3C2410X LCD 控制器用于传输显示数据和产生控制信号，例如 VFRAME，VLINE，VCLK，VM 等等。除了控制信号之外，S3C2410X 还提供数据端口供显示数据传输，也就是 VD[23:0]。如图...所示。LCD 控制器包含 REGBANK，LCDCDMA，VIDPRCS，TIMEGEN 和 LPC3600 等控制模块。REGBANK 中有 17 个可编程的寄存器组和 256X16 调色板内存用于配置 LCD 控制器。LCDCDMA 是一个专用的 DMA，它负责自动的将帧缓冲区中的显示数据发往 LCD 驱动器。通过特定的 DMA，显示数据可以不需要 CPU 的干涉，自动的发送到屏幕上。VIDPRCS 将 LCDCDMA 发送过

来的数据变换为合适的格式（例如 4/8 位单扫描或 4 位双扫描显示模式）之后通过 VD[23:0]发送到 LCD 驱动器。TMIEGEN 包含可编程逻辑用于支持不同 LCD 驱动器对时序以及速率的需求。VFRAME, VLINE, VCLK, VM 等控制信号由 TIMEGEN 产生。在 LCD 控制起的 33 个输出接口中有 24 个用户数据输出，9 个用于控制。如下表所示。

表 5-1-2 S3C2410X LCD 控制器输出接口说明

输出接口信号	描述
VFRAME/VSYNV/STV	帧同步信号（STN）/垂直同步信号（TFT）/SEC TFT 信号
VLINE/HSYNV/CPV	行同步信号（STN）/水平同步信号（TFT）/ SEC TFT 信号
VCLK/LCD_HCLK	时钟信号(STN/TFT)/SEC TFT 信号
VD[23:0]	LCD 显示数据输出端口(STN/TFT/SEC TFT)
VM/VDEN/TP	交流控制信号(STN)/数据使能信号(TFT)/SEC TFT 信号
LEND/STH	行结束信号(TFT)/SEC TFT 信号
LCD_PWREN	LCD 电源使能
LCDVF0	SEC TFT 信号 OE
LCDVF1	SEC TFT 信号 REV
LCDVF2	SEC TFT 信号 REVB

表 5-1-3 LCD 控制器寄存器列表^[注]

寄存器名	内存地址	读写	说 明	复位值
LCDCON1	0X4D000000	R/W	LCD 控制寄存器 1	0x00000000
LCDCON2	0X4D000004	R/W	LCD 控制寄存器 2	0x00000000
LCDCON3	0X4D000008	R/W	LCD 控制寄存器 3	0x00000000
LCDCON4	0X4D00000C	R/W	LCD 控制寄存器 4	0x00000000
LCDCON5	0X4D000010	R/W	LCD 控制寄存器 5	0x00000000
LCDSADDR1	0X4D000014	R/W	STN/TFT:高位帧缓存地址寄存器 1	0x00000000
LCDSADDR2	0X4D000018	R/W	STN/TFT:低位帧缓存地址寄存器 2	0x00000000
LCDSADDR3	0X4D00001C	R/W	STN/TFT:虚屏地址寄存器	0x00000000
REDLUT	0X4D000020	R/W	STN:红色定义寄存器	0x00000000
GREENLUT	0X4D000024	R/W	STN:绿色定义寄存器	0x00000000
BLUELUT	0X4D000028	R/W	STN:蓝色定义寄存器	0x0000
DITHMODE	0X4D00004C	R/W	STN:抖动模式寄存器	0x00000
TPAL	0X4D000050	R/W	TFT:临时调色板寄存器	0x00000000
LCDINTPND	0X4D000054	R/W	指示 LCD 中断 pending 寄存器	0x0
LCDSRCPND	0X4D000058	R/W	指示 LCD 中断源 pending 寄存器	0x0
LCDINTMSK	0X4D00005C	R/W	中断屏蔽寄存器（屏蔽哪个中断源）	0x3
LPCSEL	0X4D000060	R/W	LPC3600 模式控制寄存器	0x4

注:

以下实验说明中只是简单地介绍控制寄存器的含义, 详细使用请参考 S3C2410X 处理器数据手册。

地址从 0x14A0002C 到 0x14A00048 禁止使用, 因为这个区域用作测试用保留地址。

S3C2410X 能够支持 STN LCD 和 TFT LCD, 这两种 LCD 屏在显示的时候有很大的差别, 而且所涉及到的寄存器也会不同。Embest EduKit-III 实验平台采用的是 STN LCD, 下面对 STN LCD 的显示过程进行详细的介绍。

● STN LCD 显示

(1) LCD 控制器时间相关参数设定

TIMEGEN 产生 LCD 驱动器所需要的控制信号, 例如 VFRAME, VLINE, VCLK 和 VM。这些控制信号又和 REGBANK 中的寄存器 LCDCON1/2/3/4/5 的设置密切相关。可以对 REGBANK 中的这些寄存器进行设置以产生适合于不同种类 LCD 驱动器的控制信号。

VFRAME 脉冲是 LCD 的帧控制信号, 它在每一帧的整个第一行期间有效。其作用是将 LCD 扫描线每一屏显示的头部。

VM 交流控制电压信号用于控制像素的亮和灭。VM 信号的频率取决于 LCDCON1 寄存器的 MMODE 位和 LCDCON4 寄存器的 MVAL 域。如果 MMODE 位为 0, 则 VM 信号就在每一帧切换一次; 如果 MMODE 位为 1, 则 VM 信号的切换位置由 MVAL[7:0], 即 LCDCON4[15:8] 的值来决定。具体的公式为:

$$\text{VM 速率} = \text{VLINE 速率} / (2 \times \text{MVAL})$$

$$\text{VM Rate} = \text{VLINE Rate} / (2 \times \text{MVAL})$$

VFRAME 和 VLINE 脉冲的产生依赖于对 LCDCON2/3 寄存器中 HOZVAL 域和 LINEVAL 域的配置, HOZVAL 和 LINEVAL 可以由 LCD 的大小和显示模式根据如下公式来确定:

$$\text{HOZVAL} = (\text{水平尺寸} / \text{VD 数据位})$$

在彩色模式下:

$$\text{水平尺寸} = 3 \times \text{水平像素点数}$$

在 4 位单扫描模式和 4 位双扫描模式下, VD 数据位均为 4, 而在 8 位单扫描模式下 VD 数据位为 8。

$$\text{LINEVAL} = \text{垂直尺寸} - 1 \quad : \text{单扫描模式}$$

$$\text{LINEVAL} = (\text{垂直尺寸} / 2) - 1 : \text{双扫描模式}$$

VCLK 取决于 LCDCON1 中 CLKVAL 域 (LCDCON1[17:8]) 的配置, CLKVAL 最小值为 2。

$$\text{VCLK(Hz)} = \text{HCLK} / (\text{CLKVAL} \times 2)$$

帧速率取决于 VFRAME 的信号频率。帧速率和 LCDCON1/2/3/4 寄存器中的域 WLH[1:0] (VLINE 脉冲宽度), WDLY[1:0] (VLINE 脉冲之后延迟宽度), HOZVAL, LINEBLANK 和 LINEVAL 以及 VCLK, HCLK 有密切关系。多数 LCD 驱动器都需要按照下面的方法设置适当的帧速率。

$$\text{帧速率(Hz)} = 1 / [\{ (1/\text{VCLK}) \times (\text{HOZVAL} + 1) + (1/\text{HCLK}) \times (\text{A} + \text{B} + (\text{LINEBLANK} \times 8)) \} \times (\text{LINEVAL} + 1)]$$

其中:

$$\text{A} = 2(4 + \text{WLH}), \text{B} = 2(4 + \text{WDLY})$$

LINEBANK ---- 水平扫描信号 LINE 持续时间设置 (MCLK 个数)

LINEVAL ---- 显示屏的垂直尺寸

VCLK 是 LCD 控制器的时钟信号，VCLK 的计算需要先计算数据传送速率，并由此设定的一个大于数据传送速率的值为 CLKVAL (LCDCON1[17:8])。

数据传送速率 = 水平尺寸 × 垂直尺寸 × 帧速率 × 模式值(MV)

表 5-1-4 每一种显示模式的 MV 值

液晶类型	4 位双扫描	4 位单扫描	8 位单扫描
单色液晶	1/8	1/4	1/8
4 级灰度屏	1/8	1/4	1/8
16 级灰度屏	1/8	1/4	1/8
彩色液晶	3/8	3/4	3/8

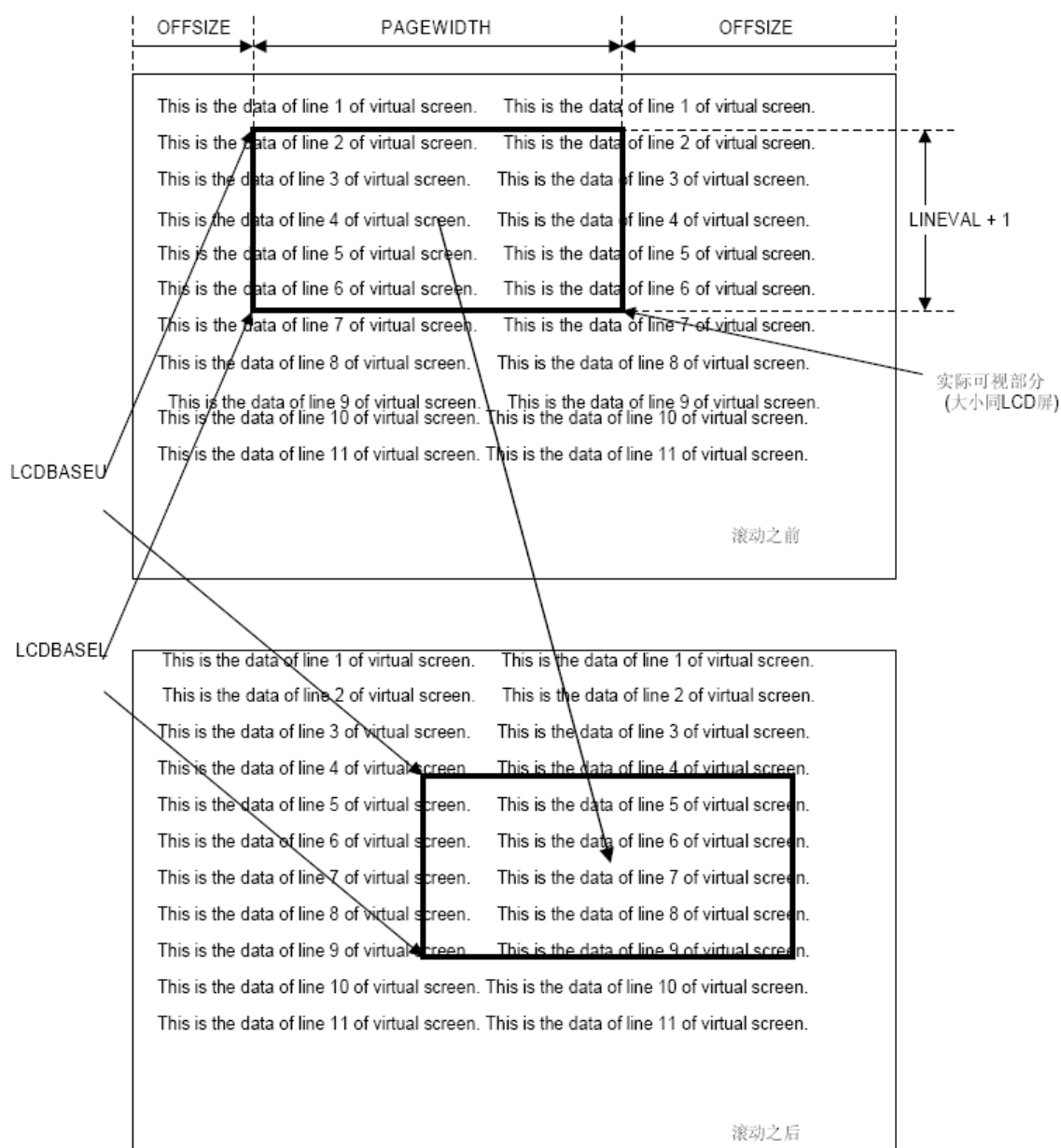


图 5-1-4 屏幕滚动显示的示例（单扫描）

(2) LCD 控制器帧显示控制参数设定

LCD 控制器中与帧显示控制相关的寄存器为 LCDSADDR1~3。在表 5-1-3 中已经分别进行的说明，下面对这组寄存器中与帧显示控制相关的各个域再分别加以详细介绍：

LCDBANK(LCDSADDR1[29:21]): 访问系统内存中显示存储区的地址 (A[30:22]) 值

LCDBASEU(LCDSADDR1[20:0]): 双扫描时，设置为帧缓存地址 (A[21:1]) 的高位缓存起始地址指针；单扫描时，设置为帧缓存起始地址指针

LCDBASEL(LCDSADDR2[20:0]): 双扫描时，设置为帧缓存地址 (A[21:1]) 的低位缓存起始地址指针；单扫描时，设置为帧缓存结束地址指针。LCDBASEL=((帧缓存结束地址)>>)+1=LCDBASEU+(PAGEWIDTH+OFFSIZE)x(LINEVAL+1)

OFFSIZE(LCDSADDR3[21:11]): 显示存储区的前行最后半字和后行第一个半字之间的半字数 (计算方法: 虚拟屏幕上一行的半字数 - LCD 屏上一行的半字数)

PAGEWIDTH(LCDSADDR3[11:0]): 显示存储区的可见帧宽度 (半字数)

具体的参数设置可以参考下面几个例子：

例 1. LCD: 2bpp STN, 320×240, 四位双扫描 虚拟屏幕: 1024×1024

1个半字 = 8像素 (4级灰度, 2位表示1像素)

虚拟屏幕上的一行 = 128个半字 (1024×2/16)

LCD屏上的一行 = 40个半字 (320×2/16)

OFFSIZE = 128 - 40 = 88 = 0X58

PAGEWIDTH = 40 = 0X28

LCDBASEL = LCDBASEU + (PAGEWIDTH + OFFSIZE) × (LINEVAL + 1) = 100 + (40 + 88) × 120 = 0X3C64

例 2. LCD 液晶屏: 320×240, 16 级灰度, 单扫描

数据帧首地址 = 0x0c500000

偏移点数 = 2048 dots (512 半字) (假定值)

LINEVAL = 240-1 = 0xef

PAGEWIDTH = 320×4/16 = 0x50

OFFSIZE = 512 = 0x200 (假定值)

LCDBANK = 0x0c500000 >> 22 = 0x31

LCDBASEU = 0x100000 >> 1 = 0x80000 (假定值)

LCDBASEL = 0x80000 + (0x50 + 0x200) × (0xef + 1) = 0xa2b00

例 3. LCD 液晶屏: 320×240, 16 级灰度, 双扫描

数据帧首地址 = 0x0c500000

偏移点数 = 2048 dots (512 半字) (假定值)

LINEVAL = 120-1 = 0x77

PAGEWIDTH = 320×4/16 = 0x50

OFFSIZE = 512 = 0x200 (假定值)

LCDBANK = 0x0c500000 >> 22 = 0x31

LCDBASEU = 0x100000 >> 1 = 0x80000 (假定值)

LCDBASEL = 0x80000 + (0x50 + 0x200) × (0x77 + 1) = 0x91580

例 4. LCD 液晶屏：320×240，彩色，单扫描

数据帧首地址 = 0x0c500000

偏移点数 = 1024 dots (512 半字) (假定值)

LINEVAL = 240-1 = 0xef

PAGEWIDTH = 320×8/16 = 0xa0

OFFSIZE = 512 = 0x200

LCDBANK = 0x0c500000 >> 22 = 0x31

LCDBASEU = 0x100000 >> 1 = 0x80000

LCDBASEL = 0x80000 + (0xa0 + 0x200) × (0xef + 1) = 0xa7600

(3) STN LCD 控制器信号时序

显示数据从内存中发送到 LCD 驱动器上就可以完成数据显示。首先，在 VCLK 的控制下，显示数据被送到 LCD 驱动器的移位寄存器上，当某一行上的数据全部进入移位寄存器后，这些数据在 VLINE 信号的下降沿经过 WDLY 指定的延迟时间后被显示到 LCD 屏上。

VM 信号提供显示所需要的交流信号，来控制每一个像素的亮和灭。VM 信号的极性变化有两种模式：当 MMODE 为 0 时，它每一帧变化一次；当 MMODE 为 1 时，它变化的周期由 MVAL 来决定。

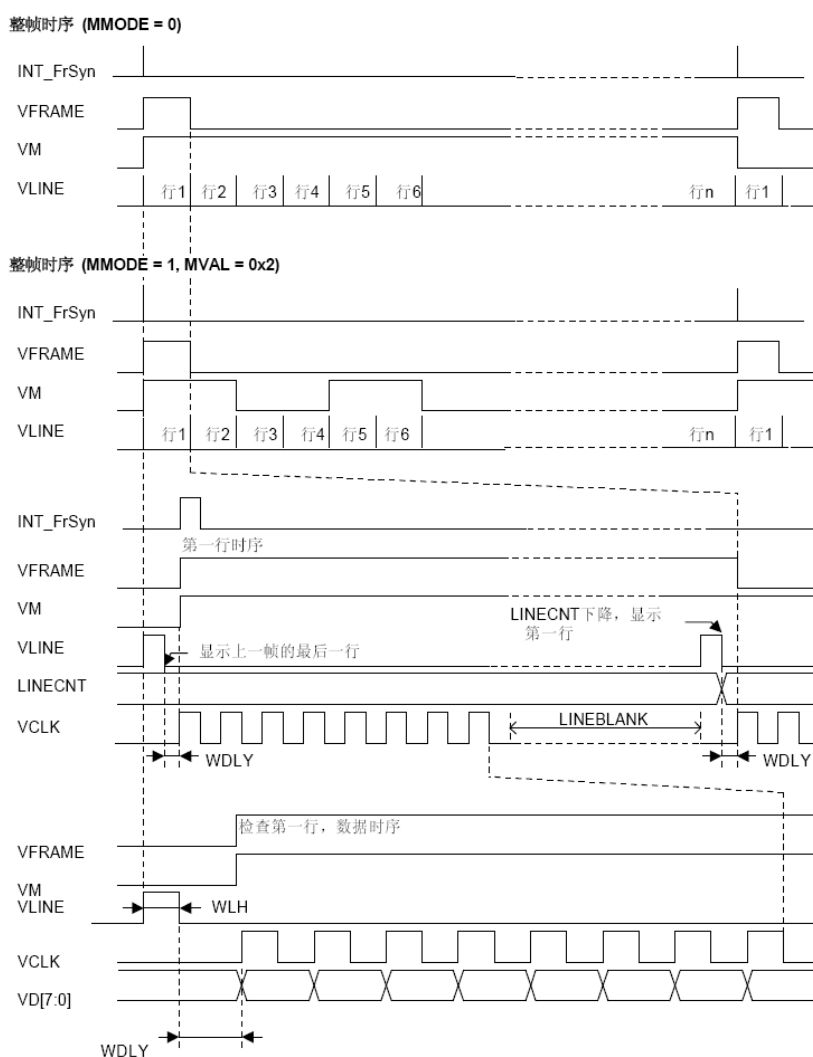


图 5-1-4. 8 位单扫描模式下 STN LCD 控制器信号时序（整帧）

注： 1. *WLH(STN)* --- *VLINE* 高电平的系统时钟个数 (*LCDCON4[7:0]*设置)
2. *WDLY(STN)* --- *VLINE* 后 *VCLK* 延时系统时钟个数 (*LCDCON3[25:19]*设置)

(4) 扫描模式支持

S3C2410X LCD 控制器扫描工作方式通过 *PNRMOD* (*LCDCON1[6:5]*) 设置。

表 5-1-5 扫描模式选择

PNRMOD	00	01	10	11
模式	4 位双扫描 (STN)	4 位单扫描 (STN)	8 位单扫描 (STN)	TFT

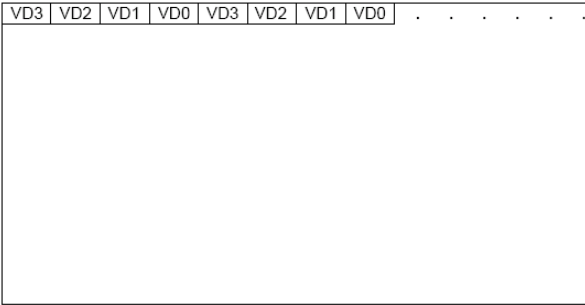
4 位双扫描 (STN) --- 显示控制器分别使用两个扫描线进行数据显示。显示数据从 *VD[3:0]* 获得高扫描数据；*VD[7:4]* 获得低扫描数据；彩色液晶屏数据位代表 RGB 色

4 位单扫描 (STN) --- 显示控制器扫描线从左上角位置进行数据显示。显示数据从 *VD[3:0]* 获得，*VD[7:4]* 没有使用；彩色液晶屏数据位代表 RGB 色

8 位单扫描 (STN) --- 显示控制器扫描线从左上角位置进行数据显示。显示数据从 *VD[7:0]* 获得；彩色液晶屏数据位代表 RGB 色



4位双扫描显示



4位单扫描显示



8位单扫描显示

图 5-1-5. 单色显示扫描类型 (STN)

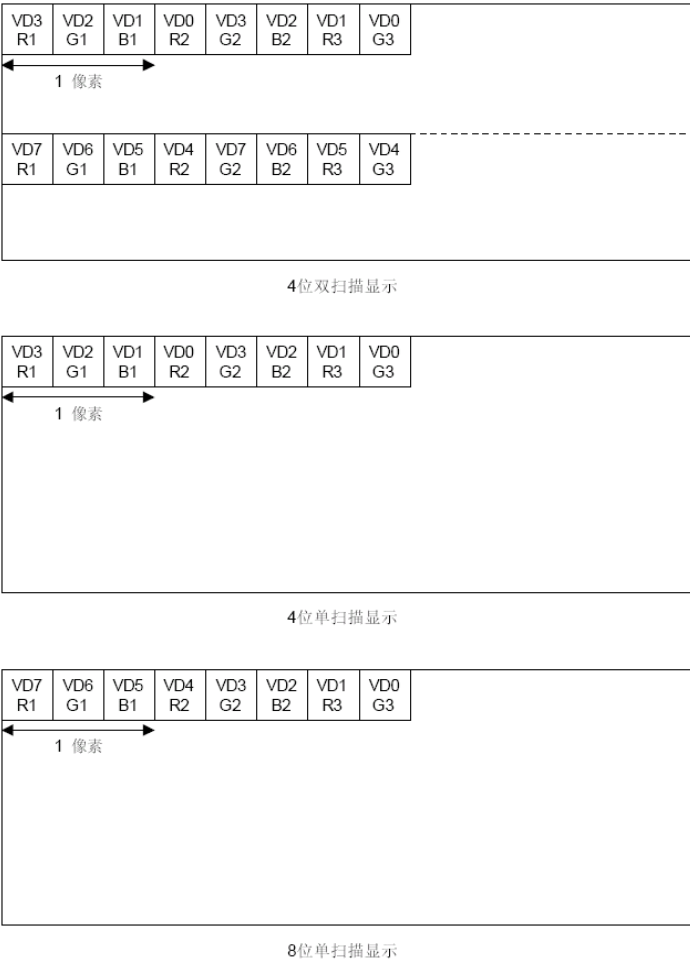


图 5-1-6 彩色显示扫描类型（STN）

(5) 数据的存放与显示

液晶控制器传送的数据表示了一个像素的属性：4 级灰度屏用两个数据位代表一个像素；16 级灰度屏时使用 4 个数据位代表一个像素；256 色 RGB 彩色模式使用 8 个数据位（R[7:5]、G[4:2]、B[1:0]）代表一个像素；4096 色 RGB 彩色模式使用 12 个数据位（4 位为红，4 位为绿，4 位为蓝）代表一个像素（每八个像素，即 12 个字节为显示数据的存放边界）。显示缓存中存放的数据必须符合硬件及软件设置，即要注意字节对齐方式。

在 4 位或 8 位单扫描方式时，数据的存放与显示如图 5-8 所示：

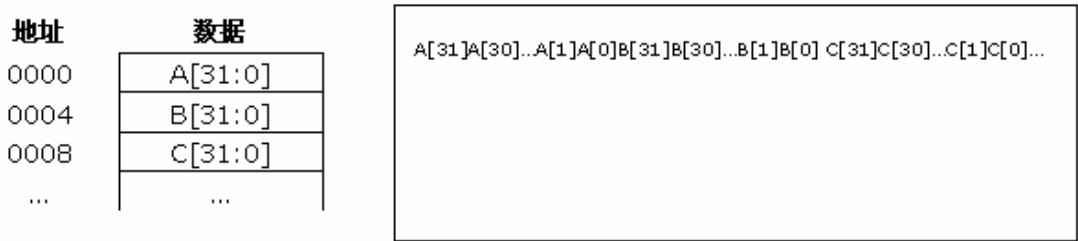


图 5-1-7 4 位或 8 位单扫描数据显示

在 4 位双扫描方式时，数据的存放与显示如图 5-1-8 所示：

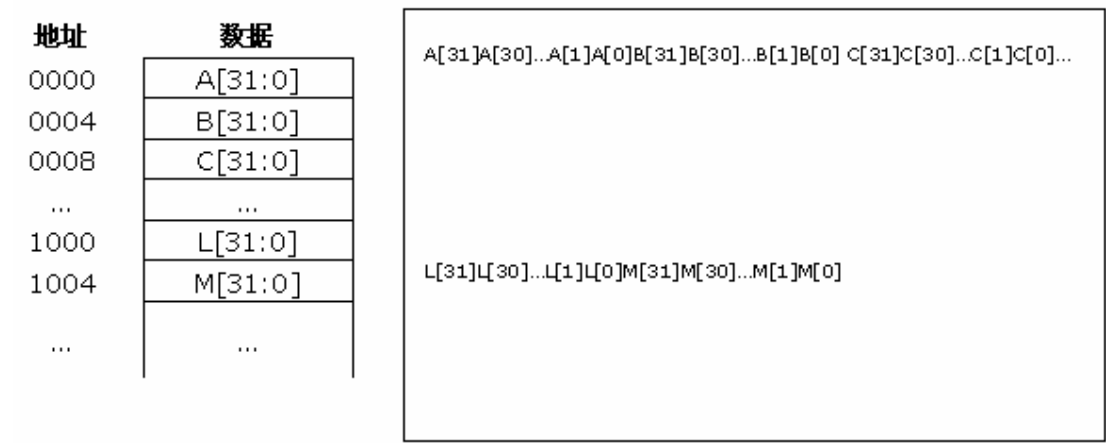


图 5-1-8 4 位双扫描数据显示

5.1.5 实验设计

1. 电路设计

进行液晶屏控制电路设计时必须提供电源驱动、偏压驱动以及 LCD 显示控制器。由于 S3C2410X 处理器本身自带 LCD 控制器，而且可以驱动实验板所选用的液晶屏，所以控制电路的设计可以省去显示控制电路，只需进行电源驱动和偏压驱动的课程设计即可。

● 液晶电路结构框图

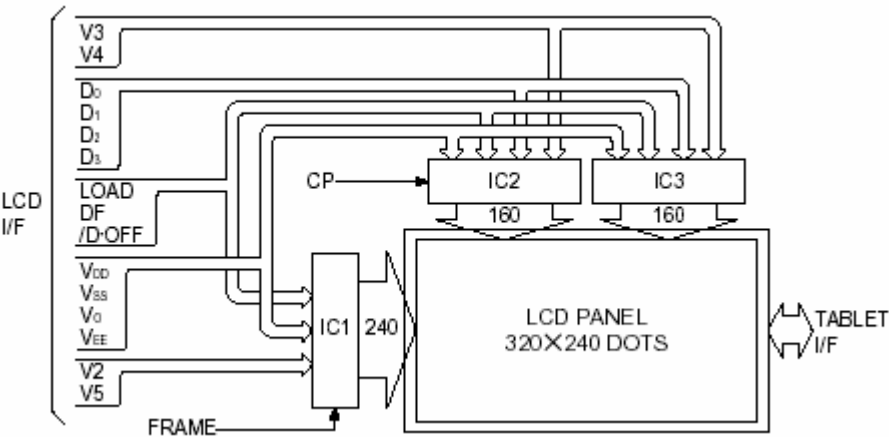


图 5-1-9 LCD 结构框图

● 引脚说明

表 5-1-5 液晶屏管脚

1	V5 偏压 5	6	VO 电源地	11	CP 时钟宽度
2	V2 偏压 2	7	LOAD 逻辑控制（内部）	12	V4 偏压 4
3	VEE 驱动电压	8	VSS 信号地	13	V3 偏压 3
4	VDD 逻辑电压	9	DF 驱动交流信号	14-17	D3-D0 数据
5	FRAME	10	/D-OFF 像素开关	18	NC 未定义

● 控制电路设计

由前述可知实验板所选用的液晶屏的驱动电源是 21.5V, 因此直接使用实验系统的 3V 或 5V 电源时需要电压升压控制, 实验系统采用的是 MAX629 电源管理模块, 以提供液晶屏的驱动电源。偏压电源可由系统升压后的电源分压得到。以下是 S3C2410 实验板的电源驱动和偏压驱动参考电路。

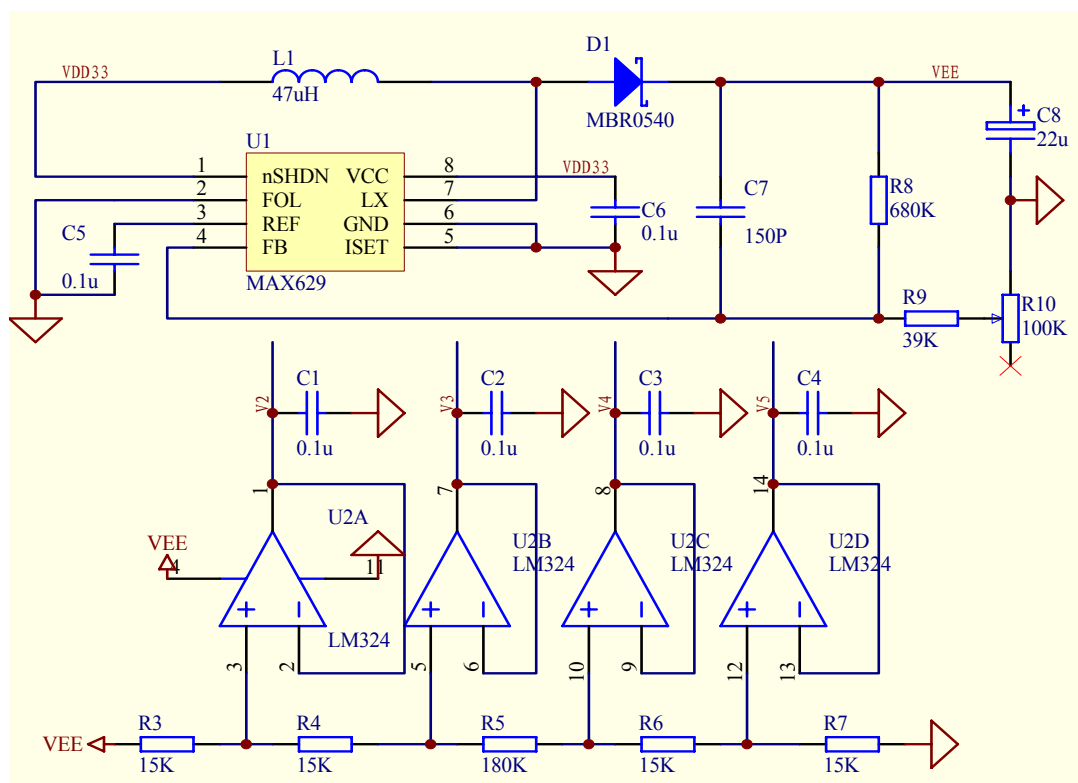


图 5-1-10 电源驱动与偏压驱动电路

2. 软件程序设计

由于实验要求在液晶显示屏上显示包括矩形、字符和位图文件, 所以实验程序设计主要包括三大部分。

● 设计思路

使用液晶屏显示最基本的是像素控制数据的使用, 像素控制数据的存放与传送形式, 决定了显示的效果。这也是所有显示控制的基本程序设计思想。图形显示可以直接使用像素控制函数实现; 把像素控制数据按一定形式存放即可实现字符显示, 比如 ASCII 字符、语言文字字符等。

Embest EduKit-III 实验平台的 LCD 显示部分像素控制函数按如下设计:

```
void (*PutPixel)(UINT32T,UINT32T,UINT32T);           //define pointer of function,display single pixel
void (*BitmapView)(UINT8T *pBuffer);                 // define pointer of function,display bitmap
```

● 矩形显示

矩形显示可以通过两条水平线和两条垂直线组成, 因此在液晶显示屏上显示矩形实际就是画线函数的实现。画线函数则通过反复调用像素控制函数得到水平线或垂直线。

```
void Glib_Line(int x1,int y1,int x2,int y2,int color); //draw line
void Glib_Rectangle(int x1,int y1,int x2,int y2,int color); //draw rectangle
void Glib_FilledRectangle(int x1,int y1,int x2,int y2,int color); //filled rectangle
void Glib_ClearScr(U32 c,int type); //clear screen
```

● 字符显示

字符的显示包括 ASCII 字符和汉字的显示。字符的显示可以采用多种形式字体，其中常用的字体大小有（W x H 或 H x W）：8x8、8x16、12x12、16x16、16x24、24x24 等，用户可以使用不同的字库以显示不同的字体。如实验系统中使用 8x16 字体显示 ASCII 字符，使用 24x24 字体显示汉字。

不管显示 ASCII 字符还是点阵汉字，都是通过查找预先定义好的字符表来实现，这个存储字符的表我们叫做库，相应的有 ASCII 库和汉字库。

```
const INT8U g_Auc_Ascii8x16[]={ //ASCII 字符查找表 }
```

ASCII 字符的存储是把字符显示数据存放在以字符的 ASCII 值为下标的库文件（数组）中，显示时再按照字体的长与宽和库的关系取出作为像素控制数据显示。ASCII 库文件只存放 ANSI ASCII 的共 255 个字符。请参考程序 `ascii8x16.c`。

```
const INT8U g_auc_HZK24[]={ //点阵汉字查找表 }
```

点阵汉字库是按照方阵形式进行数据存放，所以汉字库的字体只能是方形的。汉字库的大小与汉字显示的个数及点阵数成正比。请参考程序 `hzk24.c`。

● 位图文件显示

通过把位图文件转换成一定容量的显示数组，并按照一定的数据结构存放。与字符的显示一样，传送的数据需要设计软件控制程序。

Embest EduKit-III 实验平台位图显示的存放数据结构(请参考程序 `bmp_data.c`)及控制程序：

```
const INT8U g_ucBitmap [] = { // 位图文件数据};
```

位图显示由 `void (*BitmapView) (UINT8T *pBuffer)` 完成，它实际上是指向下面函数的一个函数指针。

```
void BitmapViewCstn8Bit (UINT8T *pBuffer);
```

5.1.6 实验操作步骤

1. 准备实验环境



使用 ULINK USB-JTAG 仿真器连接到目标板上。使用 Embest EduKit-III 实验板附带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 `µVision IDE for ARM` 通过 ULINK USB-JTAG 仿真器连接实验板，打开实验例程目录 5.1_color_lcd_test 子目录下的 `color_lcd_test.Uv2` 例程，编译链接工程成功；
- 3) 根据 ReadMe 目录下的 `ReadMeCommon.txt` 及 `readme.txt` 文件配置集成开发环境（工

程默认已经配置正确），点击工具栏 “”，在 Option for Target 对话框的 Linker 页中选择 `RuninRAM.sct` 分散加载文件，点击 MDK 的 Debug 菜单，选择 `Start/Stop Debug Session` 项或点击工具栏 “”，下载工程生成的 `.axf` 文件到目标板的 RAM 中调试运行；

- 4) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的

Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；

5) 点击 Debug 菜单的 Go 或 F5 键运行程序。

4. 观察实验结果

在 PC 机上观察超级终端程序主窗口，可以看到如下界面：

```
Boot success...

LCD display Test Example (please look at LCD screen)

Initializing GPIO ports.....

[STN 256 COLOR(8bit/1pixel) LCD TEST]
```

此时，观察 LCD 液晶屏，用户可以看到包含多个矩形框、ASCII 字符、汉字字符和彩色位图显示；

5. 完成实验练习题

理解和掌握实验后，完成实验练习题。

5.1.7 实验参考程序

1. 液晶屏初始化

```
/******
* name:      Lcd_Init
* func:      LCD initialization
* para:      int type    --    LCD display type
* ret:       none
* modify:
* comment:
*****/

void Lcd_Init(int type)
{
    switch(type)
    {
        case MODE_STN_1BIT:
            //setup address and sizeof display buffer
            frameBuffer1Bit=(UINT32T (*)(SCR_XSIZE_STN/32))LCDFRAMEBUFFER;
            //4 bit scan, 1bpp STN display model, ENVID=off
            rLCDCON1=(CLKVAL_STN_MONO<<8)|(MVAL_USED<<7)|(1<<5)|(0<<1)|0;
            //vertical size LCD_YSIZE_STN(240)-1, other bits set 0 while STN LCD display
            rLCDCON2=(0<<24)|(LINEVAL_STN<<14)|(0<<6)|(0<<0);
            //horizontal size LCD_XSIZE_STN/4-1

            rLCDCON3=(WDLY_STN<<19)|(HOZVAL_STN<<8)|(LINEBLANK_MONO<<0);
            rLCDCON4=(MVAL<<8)|(WLH_STN<<0);
            rLCDCON5=0;
            //frame buffer beginning address
```

```

        rLCSADDR1=((((UINT32T)frameBuffer1Bit>>22)<<21)
        |M5D((UINT32T)frameBuffer1Bit>>1);
        rLCSADDR2=M5D(((UINT32T)frameBuffer1Bit+
        (SCR_XSIZE_STN*LCD_YSIZE_STN/8))>>1);
        rLCSADDR3=((((SCR_XSIZE_STN-LCD_XSIZE_STN)/16)<<11)
        |(LCD_XSIZE_STN/16);
        break;
//.....skip servral initial model
case MODE_CSTN_8BIT:
    //setup address and size of diplay buffer
    frameBuffer8Bit=(UINT32T (*)(SCR_XSIZE_CSTN/4))LCDFRAMEBUFFER;
    //8 bit scan, 8bpp 256 STN display model, ENVID=off
    rLCDCON1=(CLKVAL_CSTN<<8)|(MVAL_USED<<7)|(2<<5)|(3<<1)|0;
    //vertical size LCD_YSIZE_STN(240)-1, other set 0 while STN LCD display
    rLCDCON2=(0<<24)|(LINEVAL_CSTN<<14)|(0<<6)|(0<<0);
    rLCDCON3=(WDLY_CSTN<<19)|(HOZVAL_CSTN<<8)|(LINEBLANK_CSTN<<0);
    rLCDCON4=(MVAL<<8)|(WLH_CSTN<<0);
    rLCDCON5=0;
    //frame buffer beginning address
    rLCSADDR1=((((UINT32T)frameBuffer8Bit>>22)<<21 )
    |M5D((UINT32T)frameBuffer8Bit>>1);
    rLCSADDR2=M5D( ((UINT32T)frameBuffer8Bit+
    (SCR_XSIZE_CSTN*LCD_YSIZE_CSTN/1))>>1 );
    rLCSADDR3=((((SCR_XSIZE_CSTN-LCD_XSIZE_CSTN)/2)<<11)
    |(LCD_XSIZE_CSTN/2);
    //color register
    rDITHMODE=0x0;
    rREDLUT =0xfdb96420;
    rGREENLUT=0xfdb96420;
    rBLUELUT =0xfb40;
    break;
//.....skip several inialization models
default:
    break;
}
}

```

2. 显示像素和位图

```

/*-----*/
/*      functions declare      */
/*-----*/

void (*PutPixel)(UINT32T,UINT32T,UINT32T);           //define pointer of function
void (*BitmapView)(UINT8T *pBuffer);

```

```

/*****
* name:      Glib_Init()
* func:      Glib initialization
* para:      type    --  lcd display mode
* ret:       none
* modify:
* comment:   PutPixel & BitmapView relocation
*****/

void Glib_Init(int type)
{
    switch(type)
    {
        case MODE_STN_1BIT:
            PutPixel=_PutStn1Bit;           //point pointer of function PutPixel to _PutStn1Bit
            BitmapView=BitmapViewStn1Bit;   // point pointer of function BitmapView to BitmapViewStn1Bit
            break;
        //.....skip several inialization models
        case MODE_CSTN_8BIT:
            PutPixel=_PutCstn8Bit;
            BitmapView=BitmapViewCstn8Bit;
            break;
        //.....skip several inialization models
        default:
            break;
    }
}

/*****
* name:      _PutCstn8Bit()
* func:      put pixel to 8bpp  256 color Cstn
* para:      UINT32T x    --  x coordinate
*            UINT32T y    --  y coordinate
*            UINT32T c    --  color value
* ret:       none
* modify:
* comment:
*****/

void _PutCstn8Bit(UINT32T x,UINT32T y,UINT32T c)
{
    //SCR_XSIZE_CSTN, SCR_YSIZE_CSTN are virtual screen horizital and vertical numbers of pixel
    if(x<SCR_XSIZE_CSTN&& y<SCR_YSIZE_CSTN)
        frameBuffer8Bit[(y)][(x)/4]=( frameBuffer8Bit[(y)][x/4]
        & ~(0xff000000>>((x)%4)*8) ) | ( (c&0x000000ff)<<((4-1-((x)%4))*8) );
}

/*****
* name:      BitmapViewCstn8Bit

```



```

* func:      Display Bmp on 8bpp 256 color Cstn
* para:      UINT8T *pBuffer    --    Bmp data
* ret:       none
* modify:
* comment:
*****/

void BitmapViewCstn8Bit(UINT8T *pBuffer)
{
    UINT32T i, j;
    UINT32T *pView = (UINT32T*)frameBuffer8Bit;

    for (i = 0; i < SCR_YSIZE_STN; i++)
    {
        for (j = 0; j < LCD_XSIZE_STN/4; j++)
        {
            pView[j] = ((*pBuffer) << 24) + ((*pBuffer+1) << 16)
                      + ((*pBuffer+2) << 8) + (*pBuffer+3);
            pBuffer += 4;
        }
        pView+=SCR_XSIZE_STN/4;
    }
}

```

3. 显示 AcsII 码和汉字

```

/*****
* name:      Lcd_DspAscll8X16
* func:      Display 8x16 Ascll
* para:      U16 x0          x coordinate of start point
*            U16 y0          y coordinate of start point
*            INT8U ForeColor  foreground color
*            INT8U *s        string to display
* ret:       none
* modify:
* comment:
*****/

void Lcd_DspAscll8X16(U16 x0, U16 y0, INT8U ForeColor, INT8U * s)
{
    INT16T i,j,k,x,y,xx;
    INT8U qm;
    S32 ulOffset;
    INT8T ywbuf[16],temp[2];

    for(i = 0; i < strlen((const char*)s); i++)
    {
        if((INT8U)*(s+i) >= 161)
        {

```

```

        temp[0] = *(s + i);
        temp[1] = '\0';
        return;
    }
    else
    {
        qm = *(s+i);
        ulOffset = (S32)(qm) * 16;
        for (j = 0; j < 16; j++)
        {
            ywbuf[j] = g_auc_Ascii8x16[ulOffset + j];
        }
        for(y = 0; y < 16; y++)
        {
            for(x = 0; x < 8; x++)
            {
                k = x % 8;
                if (ywbuf[y] & (0x80 >> k))
                {
                    xx = x0 + x + i*8;
                    PutPixel( xx, y + y0, (INT8U)ForeColor);
                }
            }
        }
    }
}

/*****
* name:      Lcd_DspHz24
* func:      Display 24x24 Chinese
* para:      U16 x0          x coordinate of start point
*            U16 y0          y coordinate of start point
*            INT8U ForeColor foreground color
*            INT8U *s        string to display
* ret:       none
* modify:
* comment:
*****/
void Lcd_DspHz24(U16 x0, U16 y0, INT8U ForeColor, INT8U *s)
{
    INT16T i,j,k,x,y,xx;
    INT8U qm,wm;
    S32 ulOffset;
    INT8T hzbuf[72],temp[2];

```

```

for(i = 0; i < strlen((const char*)s); i++)
{
    if((((INT8U)*(s+i))) < 161)
    {
        temp[0] = *(s+i);
        temp[1] = '\0';
        break;
    }
    else
    {
        qm = *(s+i) - 176;
        wm = *(s+i + 1) - 161;
        ulOffset = (S32)(qm * 94 + wm) * 72;
        for (j = 0; j < 72; j++)
        {
            hzbuf[j] = g_auc_HZK24[ulOffset + j];
        }
        for(y = 0; y < 24; y++)
        {
            for(x = 0; x < 24; x++)
            {
                k = x % 8;
                if (hzbuf[y * 3 + x / 8] & (0x80 >> k))
                {
                    xx = x0 + x + i*12;
                    PutPixel( xx, y + y0, (INT8U)ForeColor);
                }
            }
        }
        i++;
    }
}
}

```

5.1.8 练习题

修改程序使用 DMA 方式在彩色液晶屏上显示彩色位图

5.2 5x4 键盘控制实验

5.2.1 实验目的

- 通过实验掌握键盘控制与设计方法。
- 熟练编写 ARM 核处理器 S3C2410X 中断处理程序。

5.2.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。

- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

5.2.3 实验内容

- 使用实验板上 5x4 用户键盘，编写程序接收键盘中断。
- 通过 IIC 总线读入键值，并将读到的键值发送到串口。

5.2.4 实验原理

1. 常规键盘电路设计原理

用户设计行列键盘接口，一般常采用三种方法读取键值。一种是中断式，另两种是扫描法和反转法。

中断式

在键盘按下时产生一个外部中断通知 CPU，并由中断处理程序通过不同的地址读取数据线上的状态，判断哪个按键被按下。本实验采用中断方式实现用户键盘接口。

扫描法

对键盘上的某一行发送低电平，其他为高电平，然后读取列值，若列值中有一位是低，表明该行与低电平对应列的键被按下。否则扫描下一行。

反转法

先将所有行扫描线输出低电平，读列值，若列值有一位是低，表明有键按下；接着所有列扫描线输出低电平，再读行值。根据读到的值组合就可以查表得到键码。

2. 使用 ZLG7290 的键盘电路设计原理

(1) ZLG7290 的特点

- IIC 串行接口，提供键盘中断信号，方便与处理器接口；
- 可驱动 8 位共阴数码管或 64 只独立 LED 和 64 个按键；
- 可控扫描位数，可控任一数码管闪烁；
- 提供数据译码和循环，移位，段寻址等控制；
- 8 个功能键，可检测任一键的连击次数；
- 无需外接元件即直接驱动 LED，可扩展驱动电流和驱动电压；
- 提供工业级器件，多种封装形式 PDIP24，SO24。

(2) ZLG7290 的引脚说明

采用 24 引脚封装，引脚图如下所示。其引脚功能分述如下：

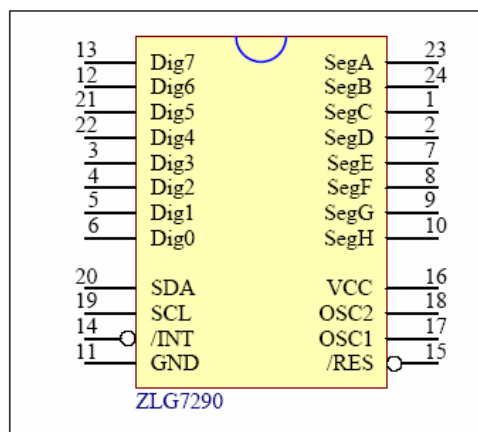


图 5-2-1 ZLG7290 引脚图

(3) ZLG7290 的寄存器说明

系统寄存器 (SystemReg): 地址 00H, 复位值 11110000B。系统寄存器保存 ZLG7290 的系统状态, 并可对系统运行状态进行配置。

KeyAvi (SystemReg.0): 置 1 时表示有效的按键动作 (普通键的单击, 连击, 和功能键状态变化), /INT 引脚信号有效变为低电平清 0 表示无按键动作/INT 引脚信号无效变为高阻态有效的按键动作消失后或读 Key 后 KeyAvi 位自动清 0。

键值寄存器 (Key): 地址 01H, 复位值 00H。Key 表示被压按键的键值。当 Key=0 时, 表示没有键被压按。

连击次数计数器 (RepeatCnt): 地址 02H, 复位值 00H。RepeatCnt=0 时, 表示单击键。RepeatCnt 大于 0 时, 表示键的连击次数。用于区别出单击键或连击键, 判断连击次数可以检测被按时间。

功能键寄存器 (FunctionKey): 地址 03H, 复位值 0FFH。FunctionKey 对应位的值=0 表示对应功能键被压按 (FunctionKey.7 ~FunctionKey.0 对应 S64 S57)。

命令缓冲区 (CmdBuf0~CmdBuf1): 地址 07H~08H, 复位值 00H~00H。用于传输指令。

闪烁控制寄存器 (FlashOnOff): 地址 0CH, 复位值 0111B/0111B。高 4 位表示闪烁时亮的时间, 低 4 位表示闪烁时灭的时间, 改变其值同时也改变了闪烁频率, 也能改变亮和灭的占空比。FlashOnOff 的 1 个单位相当于 150~250ms (亮和灭的时间范围为: 1~16, 0000B 相当 1 个时间单位), 所有像素的闪烁频率和占空比相同。

扫描位数寄存器 (ScanNum): 地址 0DH, 复位值 7。用于控制最大的扫描显示位数 (有效范围为 0~7, 对应的显示位数为: 1~8), 减少扫描位数可提高每位显示扫描时间的占空比, 以提高 LED 亮度。不扫描显示的显示缓存寄存器则保持不变。如 ScanNum=3 时, 只显示 DpRam0~DpRam3 的内容。

显示缓存寄存器 (DpRam0~DpRam7): 地址 10H~17H, 复位值 00H~00H。缓存中位置 1 表示该像素亮, DpRam7~DpRam0 的显示内容对应 Dig7~Dig0 引脚。

(4) ZLG7290 的通信接口

ZLG7290 的 IIC 接口传输速率可达 32kbit/s, 容易与处理器接口通信, 并提供键盘中断信号, 提高主处理器时间效率。ZLG7290 的从地址 slave address 为 70H (01110000B)。我们从它的键值寄存器 (01H) 中读取按键值 (ucChar 用于保存读到的键值):

```
iic_read(0x70, 0x1, &ucChar);
```

有效的按键动作都会令系统寄存器 (SystemReg) 的 KeyAvi 位置 1, /INT 引脚信号有效 (变为低电平)。用户的键盘处理程序可由 /INT 引脚低电平中断触发, 以提高程序效率; 也可以不采样 /INT 引脚信号节省系统的 I/O 数, 而轮询系统寄存器的 KeyAvi 位。要注意读键值寄存器会令 KeyAvi 位清 0, 并会令 /INT 引脚信号无效。为确保某个有效的按键动作所有参数寄存器的同步性, 建议利用 IIC 通信的自动增址功能连续读 RepeatCnt, FunctionKey 和 Key 寄存器, 但用户无需太担心寄存器的同步性问题, 因为键参数寄存器变化速度较缓慢 (典型 250ms, 最快 9ms)。

ZLG7290 内可通过 IIC 总线访问的寄存器地址范围为: 00H~17H, 任一寄存器都可按字节直接读写, 也可以通过命令接口间接读写或按位读写, 请参考 ZLG7290 芯片手册。ZLG7290 支持自动增址功能 (访问一寄存器后寄存器地址自动加一) 和地址翻转功能 (访问最后一寄存器后寄存器地址翻转 00H)。ZLG7290 的控制和状态查询全部是通过读/写寄存器实现的, 用户只需象读写 24C02 内的单元一样即可实现对 ZLG7290 的控制, 关于 IIC 总线访问的细节请参考 IIC 总线规范。

5.2.5 实验设计

1. 键盘硬件电路设计

(1) 键盘连接电路

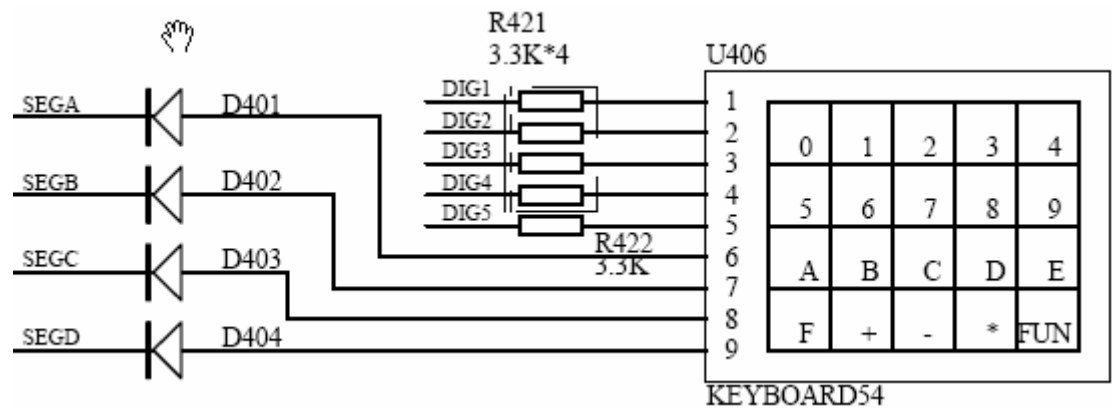


图 5-2-2 5x4 键盘连接电路

(2) 键盘控制电路

键盘控制电路使用芯片 ZLG7290 控制，如图 5-2-3。对应下图中的 14 引脚 KEY_INT 捕捉由键盘按下产生的中断触发信号。

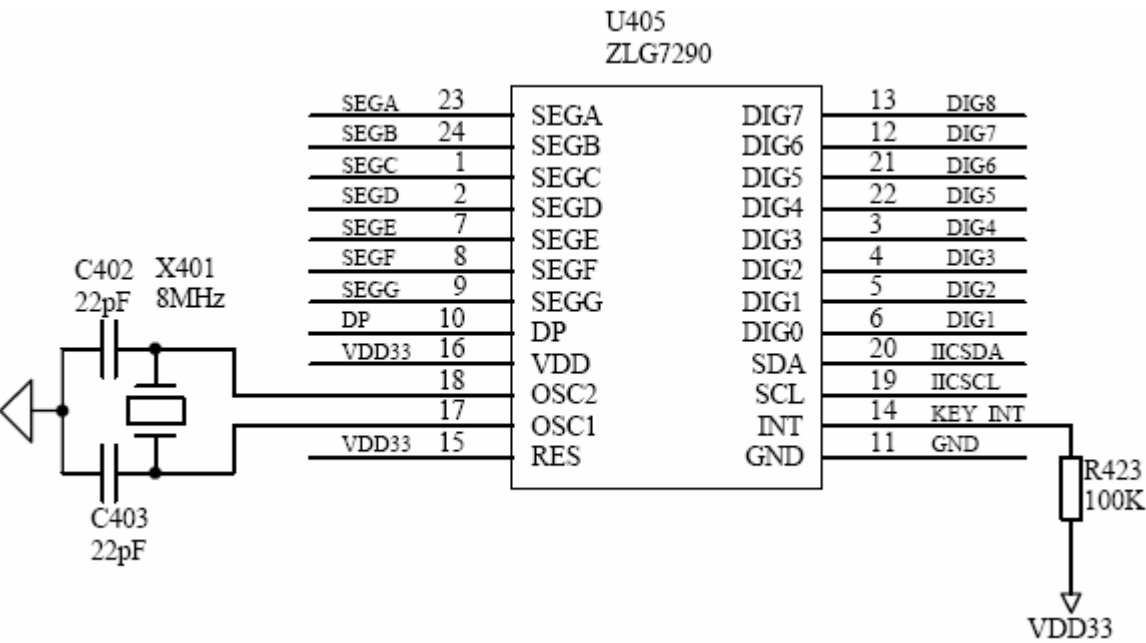


图 5-2-3 5x4 键盘控制电路

(3) 工作过程

键盘动作由芯片 ZLG7290 检测，当键盘按下时，芯片检测到后在 INT 引脚产生中断触发电平通知处理器，处理器通过 IIC 总线读取芯片 ZLG7290 键值寄存器（01H）中保存的键值（具体的读取方法可以参考 6.1 中的 IIC 串行通信实验）。

5.2.6 实验操作步骤



1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接到目标板上。使用 Embest EduKit-III 实验板附带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板，打开实验例程目录 5.2_keyboard_test 子目录下的 keyboard_test.Uv2 例程，编译链接工程成功；
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境（工程默认已经配置正确），点击工具栏 “”，在 Option for Target 对话框的 Linker 页中选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug Session 项或点击工具栏 “”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；
- 4) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；
- 5) 点击 Debug 菜单的 Go 或 F5 键运行程序。

4. 观察实验结果

- 1). 在 PC 机上观察超级终端程序主窗口，可以看到如下界面：

```
Boot success...

Keyboard Test Example
```

- 2). 用户可以按下实验系统的 5x4 键盘，在超级终端上观察结果。

5. 完成实验练习题

理解和掌握实验后，完成实验练习题。

5.2.7 实验参考程序

1. 键盘控制初始化

```

/*****
* name:      keyboard_test
* func:      test keyboard
* para:      none
* ret:       none
* modify:
* comment:
*****/

```

```

*****/

void keyboard_test(void)
{
    UINT8T ucChar;
    UINT8T szBuf[40];
    uart_printf("\n Keyboard Test Example\n");
    uart_printf(" Press any key to exit...\n");
    keyboard_init();
    g_nKeyPress=0xFE;
    while(1)
    {
        f_nKeyPress = 0;
        while(f_nKeyPress==0)
        {
            if(uart_getkey())                // Press any key from UART0 to exit
                return;
            else if(ucChar==7)                // or press 5x4 Key-7 to exit
                return;
            else if(g_nKeyPress!=0xFE)        // or SB1202/SB1203 to exit
                return;
        }
        iic_read_keybd(0x70, 0x1, &ucChar);    // get data from ZLG7290
        if(ucChar != 0)
        {
            ucChar = key_set(ucChar);          // key map for EduKitII
            if(ucChar<16)
                sprintf(&szBuf, " press key %d",ucChar);
            else if(ucChar<255)
                sprintf(&szBuf, " press key %c",ucChar);

            if(ucChar==0xFF)
                sprintf(&szBuf, " press key FUN");

            #ifdef BOARDTEST
            print_lcd(200,170,0x1c,&szBuf);
            #endif
            uart_printf(szBuf);
            uart_printf("\n");
        }
    }
}

```

2. 中断服务程序

```

/*****
* name:      keyboard_int
* func:      keyboard interrupt handler
* para:      none
* ret:       none

```



```

* modify:
* comment:
*****/

void keyboard_init(void)
{
    int i;
    iic_init_8led();
    for(i=0; i<8; i++)
    {
        iic_write_8led(0x70, 0x10+i, 0xFF);    // write data to DpRam0~DpRam7(Register of ZLG7290)
        delay(5);
    }
    iic_init_keybd();                          // enable IIC and EINT1 int
    pISR_EINT1 = (int)isrEINT1;//keyboard_int;
}
/*****

* name:      iic_int_keybd
* func:      IIC interrupt handler
* para:      none
* ret:       none
* modify:
* comment:
*****/

void __irq iic_int_keybd(void){
    ClearPending(BIT_IIC);
    f_nGetACK = 1;
}

```

5.2.8 练习题

编写程序实现双键同时按下时键盘的检测及处理程序。

5.3 触摸屏控制实验

5.3.1 实验目的

- 通过实验掌握触摸屏（TSP）的设计与控制方法。
- 掌握 S3C2410X 处理器的 A/D 转换功能。

5.3.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

5.3.3 实验内容

点击触摸屏任意位置，将触摸屏坐标转换为液晶对应坐标后显示坐标位置。

5.3.4 实验原理

1. 触摸屏（TSP）

触摸屏（TSP: Touch Screen Panel）按其技术原理可分为五类：矢量压力传感式、电阻式、电容式、红外线式和表面声波式，其中电阻式触摸屏在嵌入式系统中用的较多。

表面声波触摸屏

表面声波触摸屏的边角有 X、Y 轴声波发射器和接收器，表面有 X、Y 轴横竖交叉的超声波传输。当触摸屏幕时，从触摸点开始的部分被吸收，控制器根据到达 X、Y 轴的声波变化情况和声波传输速度计算出声波变化的起点，即触摸点。

电容感应触摸屏

人相当于地，给屏幕表面通上一个很低的电压，当用户触摸屏幕时，手指头吸收走一个很小的电流，这个电流分别从触摸屏四个角或四条边上的电极中流出，并且理论上流经这四个电极的电流与手指到四角的距离成比例，控制器通过对这四个电流比例的计算，得出触摸点的位置。

红外线触摸屏

红外线触摸屏，是在显示器屏幕的前面安装一个外框，外框里有电路板，在 X、Y 方向排布红外发射管和红外接收管，一一对应形成横竖交叉的红外线矩阵。当有触摸时，手指或其它物体就会挡住经过该处的横竖红外线，由控制器判断出触摸点在屏幕的位置。

电阻触摸屏

电阻触摸屏是一个多层的复合膜，由一层玻璃或有机玻璃作为基层，表面涂有一层透明的导电层，上面再盖有一层塑料层，它的内表面也涂有一层透明的导电层，在两层导电层之间有许多细小的透明隔离点把它们隔开绝缘。工业中常用 ITO（Indium Tin Oxide 氧化锡）导电层。当手指触摸屏幕时，平常绝缘的两层导电层在触摸点位置就有了一个接触，控制器检测到这个接通后，其中一面导电层接通 Y 轴方向的 5V 均匀电压场，另一导电层将接触点的电压引至控制电路进行 A/D 转换，得到电压值后与 5V 相比即可得触摸点的 Y 轴坐标，同理得出 X 轴的坐标。这是所有电阻技术触摸屏共同的基本原理。电阻式触摸屏根据信号线数又分为四线、五线、六线...电阻触摸屏等类型。信号线数越多，技术越复杂，坐标定位也越精确。

四线电阻触摸屏，采用国际上评价很高的电阻专利技术：包括压模成型的玻璃屏和一层透明的防刮塑料，或经过硬化、清晰或抗眩光处理的尼龙，内层是透明的导体层，表层与底层之间夹着拥有专利技术的分离点（Separator Dots）。这类触摸屏适合于需要相对固定人员触摸的高精度触摸屏的应用场合，精度超过 4096x4096，有良好的清晰度和极微小的视差。主要优点还表现在：不漂移，精度高，响应快，可以用手指或其它物体触摸，防尘防油污等，主要用于专业工程师或工业现场。

Embest EduKit-III 采用四线式电阻式触摸屏，点数为 320x240，实验系统由触摸屏、触摸屏控制电路和数据采集处理三部分组成。

被按下的触摸屏状态如图 5-3-1 所示

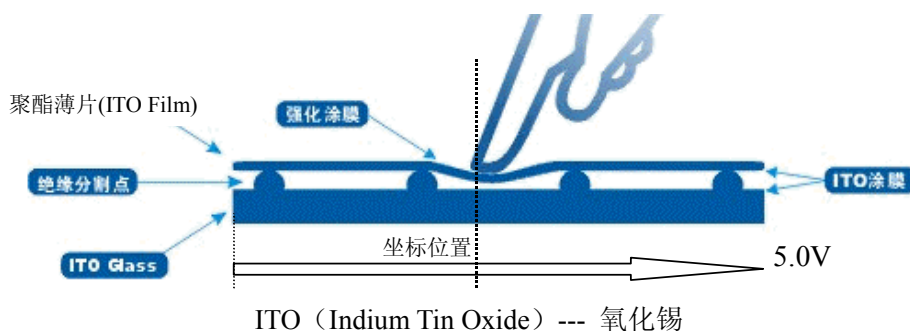


图 5-3-1 触摸屏按下



图 5-3-2 本实验台所使用的触摸屏外观图

● 等效电路结构

电阻触摸屏采用一块带统一电阻外表面的玻璃板。聚酯表层紧贴在玻璃面上，通过小的透明的绝缘颗粒与玻璃面分开。聚酯层外表面坚硬耐用，内表面有一个传导层。当屏幕被触摸时，传导层与玻璃面表层进行电子接触。产生的电压就是所触摸位置的模拟表示。

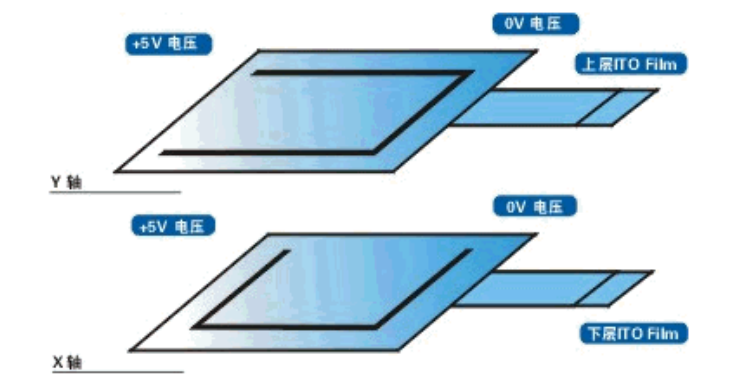


图 5-3-3 等效电路示意图

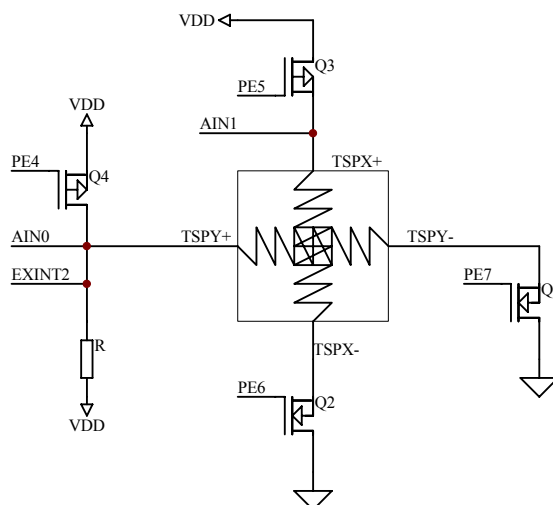


图 5-3-4 触摸屏的等效电路

● 触摸屏原点

电阻式触摸屏是通过电压的变化范围来判定按下触摸屏的位置，所以其原点就是触摸屏 X 电阻面和 Y 电阻面接通产生最小电压处。随着电阻的增大，A/D 转换所产生数值不断增加，形成坐标范围。

触摸原点的确定有很多种方法，比如常用的对角定位法、四点定位法、实验室法等。

对角定位法

系统先对触摸屏的对角坐标进行采样，根据数值确定坐标范围，可采样一条对角线或两条对角线的顶点坐标。这种方法简单易用，但是需要多次采样操作并进行比较，以取得定位的准确性。本实验板采用这种定位方法。

四点定位法

同对角定位法一样，需要进行数据采样，只是需要采样四个顶点坐标以确定有效坐标范围，程序根据四个采样值的大小关系进行坐标定位。这种方法的定位比对角定位法可靠，所以被现在许多带触摸屏的设备终端使用。

实验室法

触摸屏的坐标原点、坐标范围由生产厂家在出厂前根据硬件定义好。定位方法是按照触摸屏和硬件电路的系统参数，对批量硬件进行最优处理定义取得。这种方法适用于触摸屏构成的电路系统有较好的电气特性，且不同产品有较大相似性的场合。

● 触摸屏的坐标

触摸屏坐标值可以采用多种不同的计算方式。常用的有多次采样取平均值法、二次平方处理法等。Embest EduKit-III 教学系统的触摸屏坐标值计算采用取平均值法，首先从触摸屏的四个顶角得到两个最大值和两个最小值，分别标识为 Xmax、Ymax 和 Xmin、Ymin。

参照图 5-3-10 组成的坐标识别控制电路，X、Y 方向的确定见表 5-3-1。

表 5-3-1 确定 X、Y 方向

	A/D 通道	N-FET	P-FET
X	AIN0	Q1(-) = 1; Q2(+) = 0	Q3(-) = 0; Q4(+) = 1
Y	AIN1	Q1(+) = 0; Q2(-) = 1	Q3(+) = 1; Q4(-) = 0

注意：在 Embest EduKit-III 实验台中 AIN0 对应 AIN7，AIN1 对应 AIN5

当触摸屏被按下时，首先导通 FET 管组 Q2 和 Q4，X+ 与 X- 回路加上 +5V 电源，同时将 FET 管组 Q1 和 Q3 关闭，断开 Y+ 和 Y-；再启动处理器的 A/D 转换通道 0，电路电阻与触摸屏按下产生的电阻输出分量电压，并由 A/D 转换器将电压值数字化，计算出 X 轴的坐标。

接着先导通 FET 管组 Q1 和 Q3，Y+ 与 Y- 回路加上 +5V 电源，同时将 MOS 管组 Q2 和 Q4 关闭，断开 X+ 和 X-；再启动处理器的 A/D 转换通道 1，电路电阻与触摸屏按下产生的电阻输出分量电压，并由 A/D 转换器将电压值数字化，计算出 Y 轴的坐标。

确定 X、Y 方向后坐标值的计算可通过以下方式求得（请参照程序设计）：

$$X = (X_{\max} - X_a) \times 240 / (X_{\max} - X_{\min}) \quad X_a = [X1 + X2 + \dots + X_n] / n$$

$$Y = (Y_{\max} - Y_a) \times 320 / (Y_{\max} - Y_{\min}) \quad Y_a = [Y1 + Y2 + \dots + Y_n] / n$$

2. A/D 转换器（ADC）

● A / D 转换器的类型

A / D 转换器种类繁多，分类方法也很多。其中常见的包括以下分类：

按照工作原理可分为：计数式 A / D 转换器、逐次逼近型、双积分型和并行 A / D 转换几类。

按转换方法可分为：直接 A/D 转换器和间接 A/D 转换器。所谓直接转换是指将模拟量转换成数字量；而间接转换则是指将模拟量转换成中间量，再将中间量转换成数字量。

按分辨率可分为：二进制的 4 位、6 位、8 位、10 位、12 位、14 位、16 位和 BCD 码的 3 位半、4 位半、5 位半等。

按转换速度可分为：低速（转换时间 $\geq 1\text{s}$ ）、中速（转换时间 $\leq 1\text{ms}$ ）、高速（转换时间 $\geq 1\mu\text{s}$ ）和超高速（转换时间 $\leq 1\text{ns}$ ）。

按输出方式可分为：并行、串行、串并行等。

● A/D 转换器的工作原理

A/D 转换的方法很多，下面介绍常用的 A/D 转换原理。

(1) 计数式

这种 A/D 转换原理最简单直观，它由 D/A 转换器、计数器和比较器组成，如图 5-21 所示。计数器由零开始计数，将其计数值送往 D/A 转换器进行转换，将生成的模拟信号与输入模拟信号在比较器内进行比较，若前者小于后者，则计数值加 1，重复 D/A 转换及比较过程。因为计数值是递增的，所以 D/A 输出的模拟信号是一个逐步增加的量，当这个信号值与输出模拟量比较相等时（在允许的误差范围内），比较器产生停止计数信号，计数器立即停止计数。此时 D/A 转换器输出的模拟量即为模拟输入值，计数器的值就是转换成的相应的数字量值。这种 A/D 转换器结构简单、原理清楚，但是转换速度与精度之间存在严重矛盾即若要提高转换速度，则转换器输出与输入的误差就越大，反之亦然。所以在实际中很少使用。

(2) 逐次逼近式

逐次逼近 A/D 转换器是由一个比较器、D/A 转换器、寄存器及控制逻辑电路组成，如图 5-3-6 所示。和计数式相同，逐次逼近式也要进行比较，以得到转换数字值。但在逐次逼近式中，是用一个寄存器控制 D/A 转换器。逐次逼近式是从高位到低位依次开始逐位试探比较。S3C2410X 处理器集成了这种 A/D 转换器。

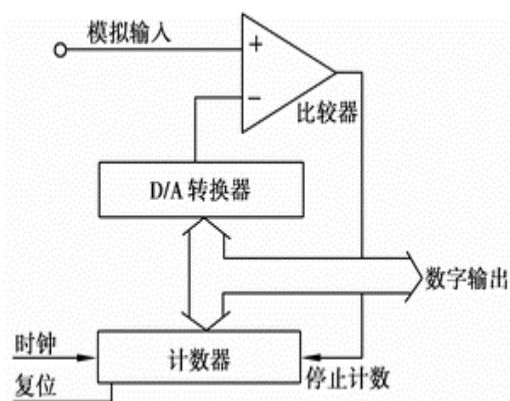


图 5-3-5 计数式 A/D 转换图

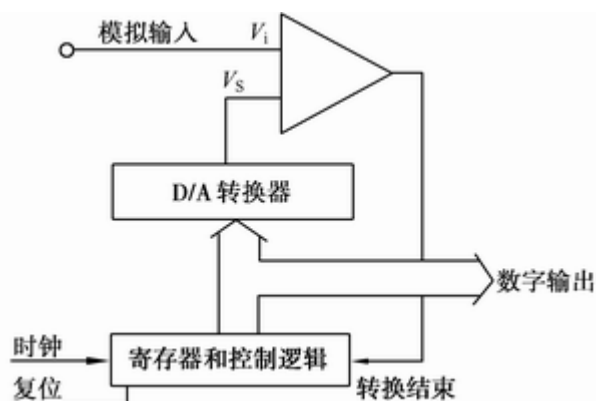


图 5-3-6 逐次逼近 A/D 转换图

逐次逼近式转换过程如下：初始时寄存器各位清为 0，转换时，先将最高位置 1，送入 D/A 转换器，经 D/A 转换后生成的模拟量送入比较器中与输入模拟量进行比较，若 $V_s < V_i$ ，该位的 1 被保留，否则被清除。然后次高位置为 1，将寄存器中新的数字量送入 D/A 转换器，输出的 V_s 再与 V_i 比较，若 $V_s < V_i$ ，保留该位的 1，否则清除。重复上述过程，直至最低位。最后寄存器中的内容即为输入模拟值转换成的数字量。

对于 n 位逐次逼近式 A/D 转换器，要比较 n 次才能完成一次转换。因此，逐次逼近式 A/D 转换器的转换时间取决于位数和时钟周期。转换精度取决于 D/A 转换器和比较器的精度，一般可达 0.01

%, 转换结果也可串行输出。逐次逼近式 A/D 转换器可应用于许多场合, 是应用最为广泛的一种 A/D 转换器。

● A/D 转换器主要性能指标

(1) 分辨率

分辨率是指 A/D 转换器能分辨的最小模拟输入量。通常用能转换成的数字量的位数来表示, 如 8 位、10 位、12 位、16 位等。位数越高, 分辨率越高。如分辨率为 10 位, 表示 A/D 转换器能分辨满量程的 $1/1024$ 的模拟增量, 此增量亦可称为 1LSB 或最低有效位的电压当量。

(2) 转换时间

转换时间是 A/D 转换完成一次转换所需的时间。即从启动信号开始到转换结束并得到稳定数字输出量为止的时间。一般来说, 转换时间越短则转换速度就越快。不同的 A/D 转换器转换时间差别较大, 通常为微秒数量级。

(3) 量程

量程是指所能转换的输入电压范围。

(4) 绝对精度

A/D 转换器的绝对精度是指在输出端产生给定的数字代码的情况下, 实际需要的模拟输入值与理论上要求的模拟输入值之差。

(5) 相对精度

相对精度是指 A/D 转换器的满刻度值校准以后, 任意数字输出所对应的实际模拟输入值 (中间值) 与理论值 (中间值) 之差。线性 A/D 转换器的相对精度就是它的线性度。精度代表电气或工艺精度, 其绝对值应小于分辨率, 因此常用 1 LSB 的分数形式来表示。

3. S3C2410X 处理器的 A/D 转换

处理器内部集成了采用近似比较算法 (计数式) 的 8 路 10 位 ADC, 集成零比较器, 内部产生比较时钟信号; 支持软件使能休眠模式, 以减少电源损耗。其主要特性:

- 精度 (Resolution): 10-bit
- 微分线性误差 (Differential Linearity Error): ± 1.5 LSB
- 积分线性误差 (Integral Linearity Error): ± 2.0 LSB
- 最大转换速率 (Maximum Conversion Rate): 500 KSPS
- 输入电压 (Input voltage range): 0-3.3V
- 片上采样保持电路
- 正常模式
- 单独 X,Y 坐标转换模式
- 自动 X,Y 坐标顺序转换模式
- 等待中断模式

4. S3C2410X 处理器 TSP 控制器

● 寄存器组

处理器集成的 TSP 只使用到 3 个个寄存器, 即 ADC 控制寄存器 (ADCCON)、触摸屏控制寄存器 (ADCTSC)、ADC 数据寄存器 (ADCDAT)。

ADC 控制寄存器 (ADCCON)

寄存器	地址	R/W	功能描述	复位值
ADCCON	0x58000000	R/W	ADC 控制寄存器	0x3FC4

ADCCON[15]: A/D 转换结束标志

0: A/D 转换正在进行;

1: A/D 转换结束

ADCCON[14]: AD 转换预分频允许

0: 不允许预分频

1: 允许预分频

ADCCON[13:6]:预分频值 PRSCVL

PRSCVL 在 0 到 255 之间, 实际的分频值为 PRSCVL+1

ADCCON[5:3]: 模拟信道输入选择

000 = AIN0

001 = AIN1

010 = AIN2

011 = AIN3

100 = AIN4

101 = AIN5

110 = AIN6

111 = AIN7

ADCCON[2]: 待机模式选择位

0:正常模式

1:待机模式

ADCCON[1]: A/D 转换读—启动选择位

0:禁止 Start-by-read

1:允许 Start-by-read

ADCCON[0]: A/D 转换器启动

0:A/D 转换器不工作

1: A/D 转换器开始工作

触摸屏控制寄存器 (ADCTSC)

寄存器	地址	R/W	功能描述	复位值
ADCTSC	0x58000004	R/W	TSP 控制寄存器	0x058

ADCTSC[8]: 保留, 必须为 0

ADCTSC[7]: 选择 YMON 输出值

0: 输出为 0

1: 输出为 1

ADCTSC[6]: 选择 YPON 输出值

- 0: 输出为 0
1: 输出为 1
- ADCTSC[5]: 选择 XMON 输出值
0: 输出为 0
1: 输出为 1
- ADCTSC[4]: 选择 XPON 输出值
0: 输出为 0
1: 输出为 1
- ADCTSC[3]: 上拉开关使能
0: 上拉使能
1: 上拉禁止
- ADCTSC[2]: 自动按顺序转换 X,Y 坐标选择位
0: 正常模式
1: 自动按顺序转换 X,Y 坐标使能
- ADCTSC[1:0]: 手工设置 X,Y 坐标转换
00: 无操作
01: X 坐标转换
10: Y 坐标转换
11: 等待中断模式

ADC 数据寄存器 (ADCDAT0, ADCDAT1)

寄存器	地址	R/W	功能描述	复位值
ADCDAT0	0x5800000C	R	ADC 数据寄存器	—

- ADCDAT0[15]: 等待中断模式, Stylus 电平选择
0: 低电平
1: 高电平
- ADCDAT0[14]: 自动按照先后顺序转换 X,Y 坐标
0: 正常 ADC 顺序
1: 按照先后顺序转换
- ADCDAT0[13:12]: 自定义 X,Y 位置
00: 无操作模式
01: 测量 X 位置
10: 测量 Y 位置
11: 等待中断模式

ADCDAT0[11:10]: 保留

ADCDAT0[9:0]: X 坐标转换数据值

寄存器	地址	R/W	功能描述	复位值
ADCDAT1	0x58000010	R	ADC 数据寄存器	—

ADCDAT1[15:10]与 ADCDAT0[15:10]功能相同

ADCDAT0[9:0]: Y 坐标转换数据值

A/D 转换的转换时间计算

例如 PCLK 为 50MHz, PRESCALER=49; 所有 10 位转换时间为:

$$50 \text{ MHz} / (49+1) = 1\text{MHz}$$

$$\text{转换时间为 } 1/(1\text{M}/5 \text{ cycles}) = 5\mu\text{s},$$

注意, A/D 转换器的最大工作时钟为 2.5MHz, 所以最大的采样率可以达到 500ksps。

5.3.5 实验设计

1. 电路设计

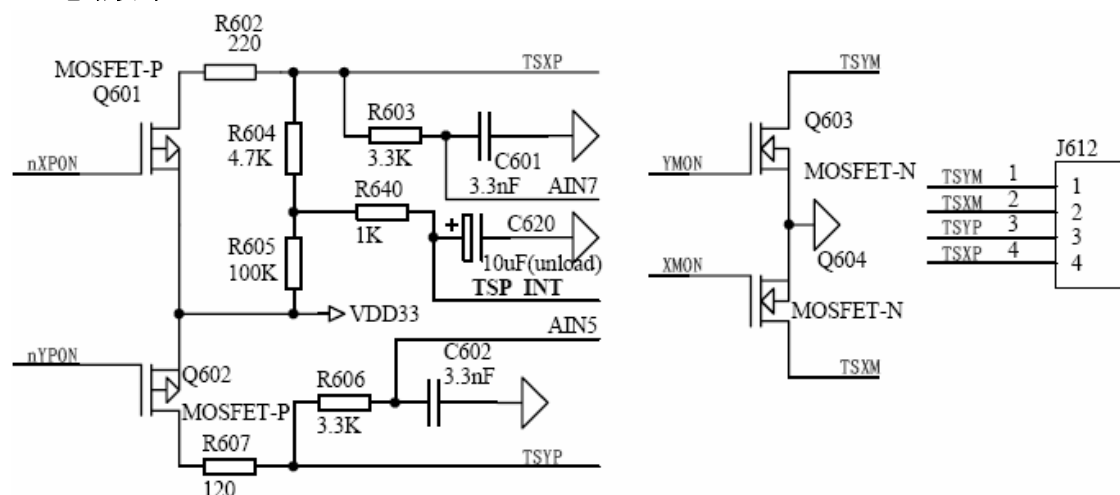


图 5-3-7 触摸屏坐标转换控制电路

当手指触摸屏幕时, 平常绝缘的两层导电层在触摸点位置就有了一个接触, 控制器检测到这个接通后, 产生中断通知 CPU 进行 A/D 转换; 具体原理如下: 当触摸屏被按下时, 首先导通 FET 管组 Q602 和 Q604, X 轴回路加上 +5V 电源, 同时将 FET 管组 Q1 和 Q3 关闭; 再启动处理器的 A/D 转换通 AIN7, 电路电阻与触摸屏按下产生的电阻输出分量电压, 并由 A/D 转换器将电压值数字化, 计算出 X 轴的坐标。中断处理程序通过导通不同 MOS 管组 (见表 5-3-1), 使接触部分与控制器电路构成电阻电路, 并产生一个电压降作为坐标值输出。

2. 软件程序设计

实验主要是对 S3C2410X 中的 TSP 模块进行操作, 所以软件程序也主要是对 A/D 和 TSP 模块中的寄存器进行操作, 包括对 ADC 控制寄存器 (ADCCON)、TSP 控制寄存器 (ADCTSC)、ADC 数据寄存器 (ADCDAT) 的读写操作。同时为了观察点触的触摸屏的位置信息, 可以通过串口在超级终端里面观察。

5.3.6 实验操作步骤



1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接到目标板上。使用 Embest EduKit-III 实验板附带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 运行实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板，打开实验例程目录 5.3_tsp_test 子目录下的 tsp_test.Uv2 例程，编译链接工程；
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境（工程默认已经配置正确），点击工具栏 “”，在 Option for Target 对话框的 Linker 页中选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug Session 项或点击工具栏 “”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；
- 4) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；
- 5) 点击 Debug 菜单的 Go 或 F5 键运行程序。

4. 观察实验结果

在 PC 上观察超级终端程序主窗口，可以看到如下界面：

```

Boot success...

Touch Screen Test Example.

Stylus Down!!

X-Posion[AIN5] is 0012

Y-Posion[AIN7] is 1006

Stylus Down!!

X-Posion[AIN5] is 0010

Y-Posion[AIN7] is 1009

```

5. 完成实验练习题

理解和掌握实验后，完成实验练习题。

5.3.7 实验参考程序

1. 触摸屏初始化

```

/*****
* name:      tsp_test
* func:
* para:      none
* ret:       none
* modify:    R.X.Huang, March 12, 2005
*****/

void tsp_test(void)
{
    uart_printf(" Touch Screen Test Example.\n");
    rADCCLK = 50000;                // ADC Start or Interval Delay
    rGPGCON |= 0xFF000000;
    rGPGUP = 0xFFFF;
    rADCTSC = (0<<8) | (1<<7) | (1<<6) | (0<<5) | (1<<4) | (0<<3) | (1<<2) | (0);
        //auto sequential x/y position conversion,no operation,      XP pull-up
    rADCCON  = (1<<14) | (ADCPRS<<6) | (5<<3) | (0<<2) | (0<<1) | (0);
        // Enable Prescaler,Prescaler,AIN7/5 fix,Normal,Disable read start,No operation
    rADCTSC = (0<<8) | (1<<7) | (1<<6) | (0<<5) | (1<<4) | (0<<3) | (0<<2) | (3);
        //YM:GND,YP:AIN5,XM:Hi-z,XP:external voltage,XP pullup En,AUTO sequential,
        delay(100);
    pISR_ADC = (unsigned)isrADC;        // pISR_ADC
    rINTMSK &= ~(BIT_ADC);
    rINTSUBMSK = ~(BIT_SUB_TC);
    uart_printf(" Press any key to exit...\n");
    g_nKeyPress = 1;
    while(g_nKeyPress==1)                // only for board test to exit
    {
        if(uart_getkey()) return;        // or press any key to exit
    }
    rINTSUBMSK |= BIT_SUB_TC;
    rINTMSK |= BIT_ADC;
    uart_printf(" end.\n");
}

```

2. 中断服务程序

```

void tsp_int(void)
{
    int i;
    UINT32T szPos[40];
    rINTSUBMSK |= (BIT_SUB_ADC | BIT_SUB_TC); // Mask sub interrupt (ADC and TC)
    if( rADCTSC & 0x100)
    {
        uart_printf(" Stylus Up!!\n");
    }
}

```

```

    rADCTSC&=0xff; // Set stylus down interrupt
}
else
{
    uart_printf(" Stylus Down!!\n");
    szPos[30] = g_nPosX;
    szPos[34] = g_nPosY;
    // <X-Position Read>
    //Hi-Z,AIN5,GND,Ext vlt,Pullup Dis,Normal,X-position
    rADCTSC = (0<<8)|(0<<7)|(1<<6)|(1<<5)|(0<<4)|(0<<3)|(0<<2)|(1);
    //adc input ain5
    rADCCON = (1<<14)|(39<<6)|(5<<3)|(0<<2)|(1<<1)|(0);
    rADCDAT0;
    delay(10);
    for(i = 0; i<nSampleNo; i++)
    {
        while(!(0x8000 & rADCCON)); // Check ECFLG
        szPos[i] = (0x3ff & rADCDAT0);
        g_nPosX += szPos[i];
    }
    g_nPosX = g_nPosX/nSampleNo;
    uart_printf(" X-Posion[AIN5] is %04d\n", g_nPosX);
    // <Y-Position Read>
    //GND,Ext vlt,Hi-Z,AIN7,Pullup Dis,Normal,Y-position
    rADCTSC = (0<<8)|(1<<7)|(0<<6)|(0<<5)|(1<<4)|(0<<3)|(0<<2)|(2);
    //adc input ain7
    rADCCON = (1<<14)|(39<<6)|(7<<3)|(0<<2)|(1<<1)|(0);
    rADCDAT1;
    delay(10);
    for(i = 0; i<nSampleNo; i++)
    {
        while(!(0x8000 & rADCCON)); // Check ECFLG
        szPos[i] = (0x3ff & rADCDAT1);
        g_nPosY += szPos[i];
    }
    g_nPosY = g_nPosY/nSampleNo;
    uart_printf(" Y-Posion[AIN7] is %04d\n", g_nPosY);

    //GND,AIN,Hi-z,AIN,Pullup En,Normal,Waiting mode
    rADCTSC=(1<<8)|(1<<7)|(1<<6)|(0<<5)|(1<<4)|(0<<3)|(0<<2)|(3);
}

#ifdef BOARDTEST
sprintf(&szPos, "(X1,Y1):(%d,%d)",szPos[30],szPos[34]);
print_lcd(195,170,0x20,&szPos);
sprintf(&szPos, "(X2,Y2):(%d,%d)",g_nPosX,g_nPosY);

```

```

print_lcd(195,178,0x1c,&szPos);
#endif
rSUBSRCPND |= BIT_SUB_TC;
rINTSUBMSK = ~(BIT_SUB_TC);          // Unmask sub interrupt (TC)
ClearPending(BIT_ADC);

}          uart_printf(" Y-Posion[AIN7] is %04d\n", szPos[5]);
          //GND,AIN,Hi-z,AIN,Pullup En,Normal,Waiting mode
          rADCTSC = (1<<8)|(1<<7)|(1<<6)|(0<<5)|(1<<4)|(0<<3)|(0<<2)|(3);
}
r SUBSRCPND |= BIT_SUB_TC;
rINTSUBMSK = ~(BIT_SUB_TC);          // Unmask sub interrupt (TC)
ClearPending(BIT_ADC);
}

```

5.3.8 练习题

改变触摸屏的工作模式设置，使其工作在自动按顺序做 A/D 转换模式，并结合液晶显示实验编写程序在 LCD 上显示坐标值。

第六章 通信与接口实验

6.1 IIC 串行通信实验

6.1.1 实验目的

- 通过实验掌握 IIC 串行数据通信协议的使用。
- 掌握 EEPROM 器件的读写访问方法。
- 通过实验掌握 S3C2410X 处理器的 IIC 控制器的使用。

6.1.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

6.1.3 实验内容

编写程序对实验板上 EEPROM 器件 AT24C04 进行读写访问。

写入 EEPROM 某一地址，再从该地址读出，输出到超级终端；

把读出内容和写入内容进行比较，检测 S3C2410X 处理器通过 IIC 接口，是否可以正常读写 EEPROM 器件 AT24C04。

6.1.4 实验原理

1. IIC 接口以及 EEPROM

IIC 总线为同步串行数据传输总线，其标准总线传输速率为 100kb/s，增强总线可达 400kb/s。总线驱动能力为 400pF。S3C2410X RISC 微处理器能支持多主 IIC 总线串行接口。下图为 IIC 总线的内部结构框图。

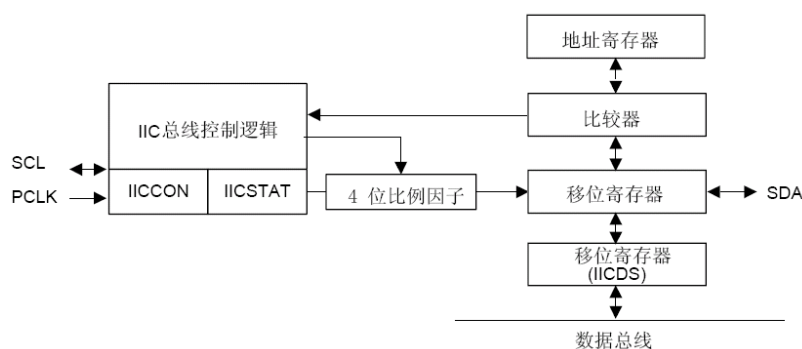


图 6-1-1 IIC 总线内部结构框图

IIC 总线可构成多主和主从系统。在多主系统结构中，系统通过硬件或软件仲裁获得总线控制使用权。应用系统中 IIC 总线多采用主从结构，即总线上只有一个主控节点，总线上的其它设备都作为从设备。IIC 总线上的设备寻址由器件地址接线决定，并且通过访问地址最低位来控制读写方向。

目前，通用存储器芯片多为 EEPROM，其常用的协议主要有两线串行连接协议（IIC）和三线串行连接协议。带 IIC 总线接口的 EEPROM 有许多型号，其中 AT24CXX 系列使用十分普遍。产品包括 AT2401/02/04/08/16 等，其容量（字节数 × 页）分别为 128x8/256x8/512x8/1024x8/2048x8，适用于 2V~5V 的低电压的操作。具有低功耗和高可靠性等优点。

AT24 系列存储器芯片采用 CMOS 工艺制造，内置有高压泵，可在单电压供电条件下工作。其标准封装为 8 脚 DIP 封装形式，如图 6-1-1。

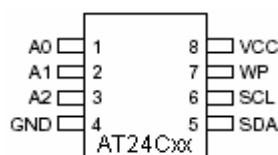


图 6-1-2 AT24 系列 EEPROM 的 DIP8 封装示意图

各引脚的功能说明如下：

SCL ---- 串行时钟。遵循 ISO/IEC7816 同步协议；漏极开路，需接上拉电阻。

在该引脚的上升沿，系统将数据输入到每个 EEPROM 器件，在下降沿输出。

SDA ---- 串行数据线；漏极开路，需接上拉电阻。

双向串行数据线，漏极开路，可与其他开路器件“线或”。

A0、A1、A2 ---- 器件/页面寻址地址输入端。

在 AT24C01/02 中，引脚被硬连接；其他 AT24Cxx 均可接寻址地址线。

WP ---- 读写保护。

接低电平时可对整片空间进行读写；高电平时不能读写受保护区。

Vcc/GND ---- 一般输入+5V 的工作电压。

2. IIC 总线的读写控制逻辑

开始条件 (START_C) 在开始条件下，当 SCL 为高电平时，SDA 由高转为低。

停止条件 (STOP_C) 在停止条件下，当 SCL 为高电平时，SDA 由低转为高。

确认信号 (ACK) 在接收方应答下，每收到一个字节后便将 SDA 电平拉低。

数据传送 (Read/Write)

IIC 总线启动或应答后 SCL 高电平期间数据串行传送；低电平期间为数据准备，并允许 SDA 线上数据电平变换。总线以字节 (8bit) 为单位传送数据，且高有效位(MSB)在前。IIC 数据传送时序如图 6-1-2 所示：

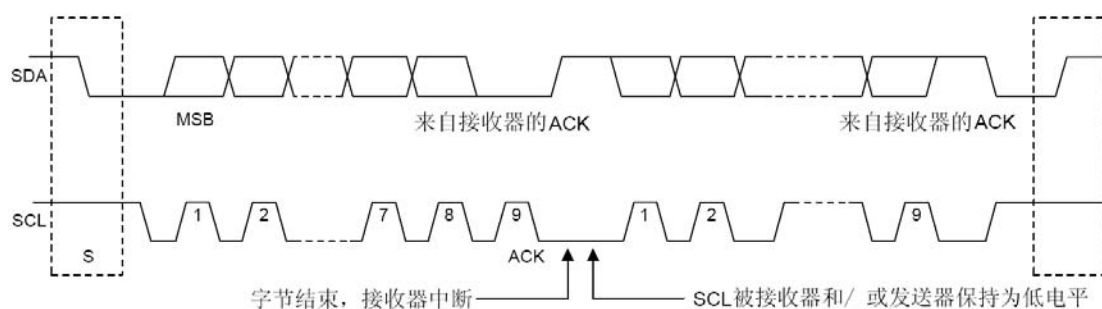


图 6-1-3 IIC 总线信号的时序

3. EEPROM 读写操作

1) AT24C04 结构与应用简述

AT24C04 由输入缓冲器和 EEPROM 阵列组成。由于 EEPROM 的半导体工艺特性写入时间为 5-10ms，如果从外部直接写入 EEPROM，每写一个字节都要等候 5-10ms，成批数据写入时则要等候更长的时间。具有 SRAM 输入缓冲器的 EEPROM 器件，其写入操作变成对 SRAM 缓冲器的装载，装载完后启动一个自动写入逻辑将缓冲器中的全部数据一次写入 EEPROM 阵列中。对缓冲器的输入称为页写，缓冲器的容量称为页写字节数。AT24C04 的页写字节数为 8，占用最低 3 位地址。写入不超过页写字节数时，对 EEPROM 器件的写入操作与对 SRAM 的写入操作相同；若超过页写字节数时，应等候 5-10ms 后再启动一次写操作。

由于 EEPROM 器件缓冲区容量较小（只占据最低 3 位），且不具备溢出进位检测功能，所以，从非零地址写入 8 个字节数或从零地址写入超过 8 个字节数会形成地址翻卷，导致写入出错。

2) 设备地址 (DADDR)

AT24C04XX 的器件地址是 1010。

3) AT24CXX 的数据操作格式

在 IIC 总线中对 AT24C04 内部存储单元读写，除了要给出器件的设备地址 (DADDR) 外还须指定读写的页面地址 (PADDR)，两者组成操作地址 (OPADDR) 如下：

1010 A2 A1- R/W (-为无效)

Embest ARM 教学系统中引脚 A2A1A0 为 000，因此系统可寻址 AT24C04 全部页面共 4KB 字节。按照 AT24C04 器件手册读写地址 (ADDR=1010 A2 A1- R/W) 中的数据操作格式如下：

写入操作格式

写任意地址 ADDR_W

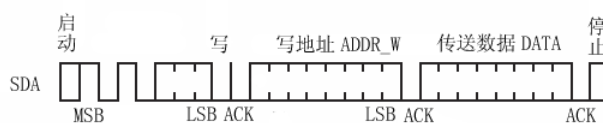


图 6-1-4 任意写一个字节

从地址 ADDR_W 起连续写入 n 个字节（同一页面）

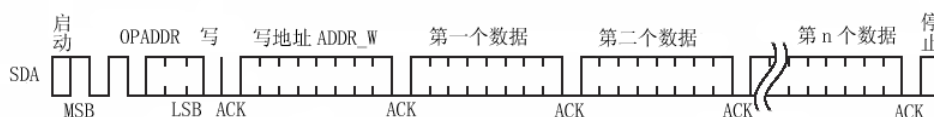


图 6-1-5 写 n 个字节

读出操作格式

读任意地址 ADDR_R

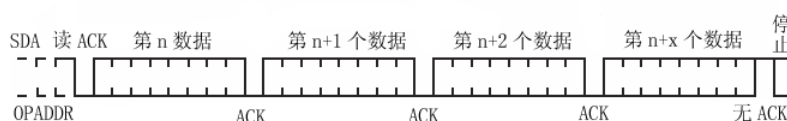


图 6-1-6 任意读一个字节

从地址 ADDR_R 起连续读出 n 个字节（同一页面）



图 6-1-7 读 n 个字节

在读任意地址操作中除了发送读地址外还要发送页面地址（PADDR），因此在连续读出 n 个字节操作前要进行一个字节 PADDR 写入操作，然后重新启动读操作；注意读操作完后没有 ACK。

4. S3C2410X 处理器 IIC 接口

1) S3C2410X IIC 接口

S3C2410X 处理器为用户进行应用设计提供了支持多主总线的 IIC 接口。处理器提供符合 IIC 协议的设备连接的双向数据线 IICSDA 和 IIC SCL，在 IIC SCL 高电平期间，IICSDA 的下降沿启动上升沿停止。S3C2410X 处理器可以支持主发送、主接收、从发送、从接收四种工作模式。在主发送模式下，处理器通过 IIC 接口与外部串行器件进行数据传送，需要使用到如下寄存器：

IIC 总线控制寄存器 IICCON

寄存器	地址	读/写	描述	复位值
IICCON	0x54000000	R/W	IIC 总线控制寄存器	0x0X

IICCON	位	描述	初始值
应答使能 ^[注 1]	[7]	IIC 总线应答使能位 0：禁止，1：使能 在输出模式下，IICSDA 在 ACK 时间被释放 在输入模式下，IICSDA 在 ACK 时间被拉低	0
输出时钟源选择	[6]	IIC 总线发送时钟预分频选择位 0：IICCLK = fPCLK / 16 1：IICCLK = fPCLK / 512	0
发送/接收中断使能 ^[注 3]	[5]	IIC 总线中断使能位 0：禁止，1：使能	0
中断未决位 ^[注 2]	[4]	IIC 总线未处理中断标志。不能对这一位写入 1，置 1 是系统自动产生的。当这位被置 1，IIC SCL 信号将被拉低，IIC 传输也停止了。如果想要恢复操作，将该位清零。 0：1) 当读出 0 时，没有发生中断 2) 当写入 0 时，清除未决条件并恢复中断响应 1：1) 当读出 1 时，发生了未决中断 2) 不可以进行写入操作	0
发送时钟值	[3:0]	发送时钟预分频器的值，这四位预分频器的值决定了 IIC 总线进行发送的时钟频率，对应关系如下： Tx clock = IICCLK / (IICCON[3:0] + 1).	Undefined

注：

1. 在 Rx 模式下访问 EEPROM 时, 为了产生停止条件, 在读取最后一个字节数据之后不允许产生 ACK 信号。

2. IIC 总线上发生中断的条件: 1) 当一个字节的读写操作完成时; 2) 当一个通常的通话发生或者是从地址匹配上时; 3) 总线仲裁失败时。

3. 如果 IICON[5]=0, IICON[4]就不能够正常工作了。因此, 建议务必将 IICCON[5]设置为 1, 即使你暂时并不用 IIC 中断。

IIC 总线状态寄存器 IICSTAT (地址: 0x54000004)

IICSTAT	位	描述	初始值
模式选择	[7:6]	IIC总线主从, 发送/接收模式选择位。 00: 从接收模式; 01: 从发送模式; 10: 主接收模式; 11: 主发送模式	00
忙信号状态/起始/停止条件	[5]	IIC总线忙信号状态位 0: 读出为0, 表示状态不忙; 写入0, 产生停止条件 1: 读出为1, 表示状态忙; 写入1, 产生起始条件 IICDS中的数据在起始条件之后自动被送出	0
串行数据输出使能	[4]	IIC总线串行数据输出使能/禁止位 0: 禁止发送/接收; 1: 使能发送接收	0
仲裁状态位	[3]	IIC总线仲裁程序状态标志位 0: 总线仲裁成功 1: 总线仲裁失败	0
从地址状态标志位	[2]	IIC总线从地址状态标志位 0: 在探测到起始或停止条件时, 被清零; 1: 如果接收到的从器件地址与保存在IICADD中的地址相符, 则置1	0
0地址状态标志位	[1]	IIC总线0地址状态标志位 0: 在探测到起始或停止条件时, 被清零; 1: 如果接收到的从器件地址为0, 则置1	0
应答位状态标志	[0]	应答位(最后接收到的位)状态标志 0: 最后接收到的位为0 (ACK接收到了) 1: 最后接收到的位为1 (ACK没有接收到)	0

IIC 总线地址寄存器 IICADD (地址: 0x54000008)

IICADD	位	描述	初始值
从器件地址	[7:0]	7位从器件地址: 如果IICSTAT中的串行数据输出使能位为0, IICADD就变为写使能。IICADD总为可读	XXXXXXXX

IIC 总线发送接收移位寄存器 IICDS (地址: 0x5400000C)

IICDS	位	描述	初始值
数据移位寄存器	[7:0]	IIC接口发送/接收数据所使用的8位数据移位寄存器: 当IICSTAT中的串行数据输出使能位为1, 则IICDS写使能。IICDS总为可读	XXXXXXXX

2) 使用 S3C2410X IIC 总线读写方法

单字节写操作 (R/W=0)**Addr:** 设备、页面及访问地址

START_C	Addr(7bit) W	ACK	DATA(1Byte)	ACK	STOP_C
---------	--------------	-----	-------------	-----	--------

同一页面的多字节写操作 (R/W=0)**OPADDR:** 设备及页面地址 (高 7 位)

START_C	OPADDR(7bit) W	ACK	Addr	DATA(nByte)	ACK	STOP_C
---------	----------------	-----	------	-------------	-----	--------

单字节读串行存储器件 (R/W=1)**Addr:** 设备、页面及访问地址

START_C	Addr(7bit) R	ACK	DATA(1Byte)	ACK	STOP_C
---------	--------------	-----	-------------	-----	--------

同一页面的多字节读操作 (R/W=1)**Addr:** 设备、页面及访问地址

START_C	P & R	ACK	Addr	ACK	P & R	ACK	DATA(nByte)	ACK	STOP_C
---------	-------	-----	------	-----	-------	-----	-------------	-----	--------

P & R = OPADDR_R = 1010xxx (字节高 7 位) R: 重新启动读操作

6.1.5 实验设计

1. 程序设计

本实验的内容就是将 0~F 这 16 个数按顺序写入到 EEPROM (AT24C04) 的内部存储单元中, 然后在依次将他们读出, 并通过实验板的串口 UART0 输出到在 PC 机上运行的 windows 自带超级终端上。在本实验中, EEPROM 是被作为 IIC 总线上的从设备来进行处理的, 其工作过程涉及到 IIC 总线的主发送和主接收两种工作模式。下图 6-1-8 和图 6-1-9 所示详细说明了这两种工作模式下的程序流程。关于它们的具体实现可以参考实验参考程序中的 write_24c040() 和 read_24c040() 这两个函数。

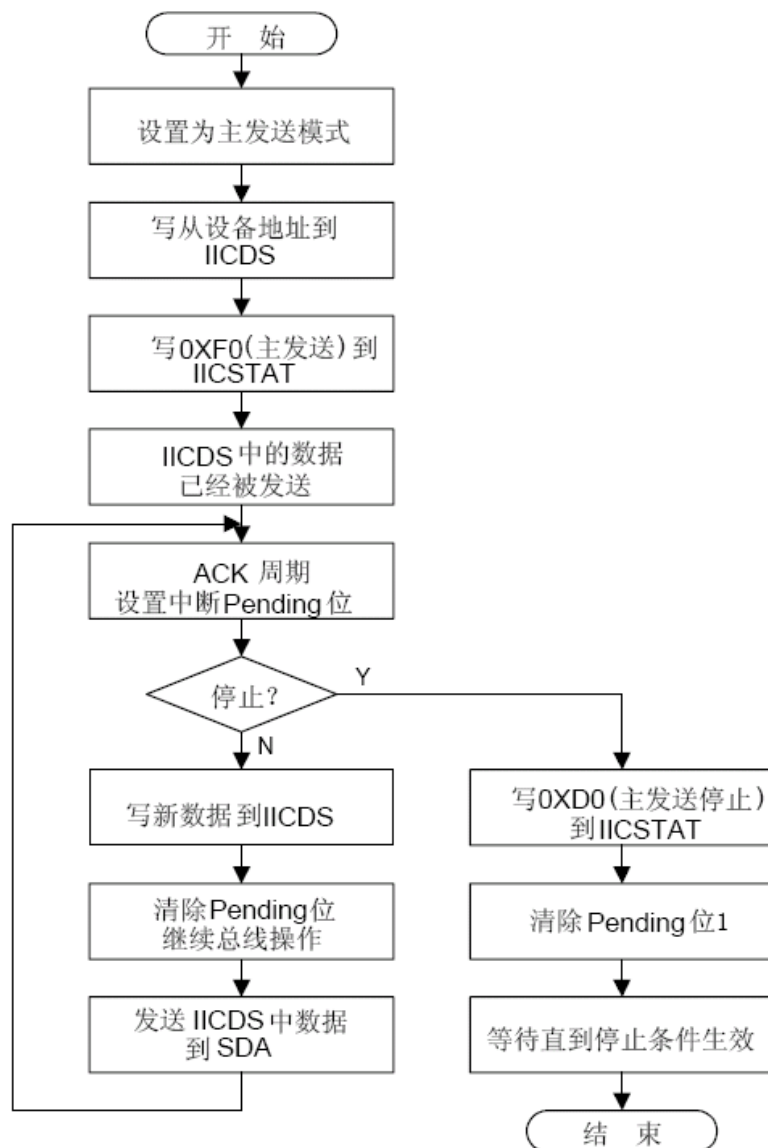


图 6-1-8 IIC 主发送程序设计流程图（S3C2410X）

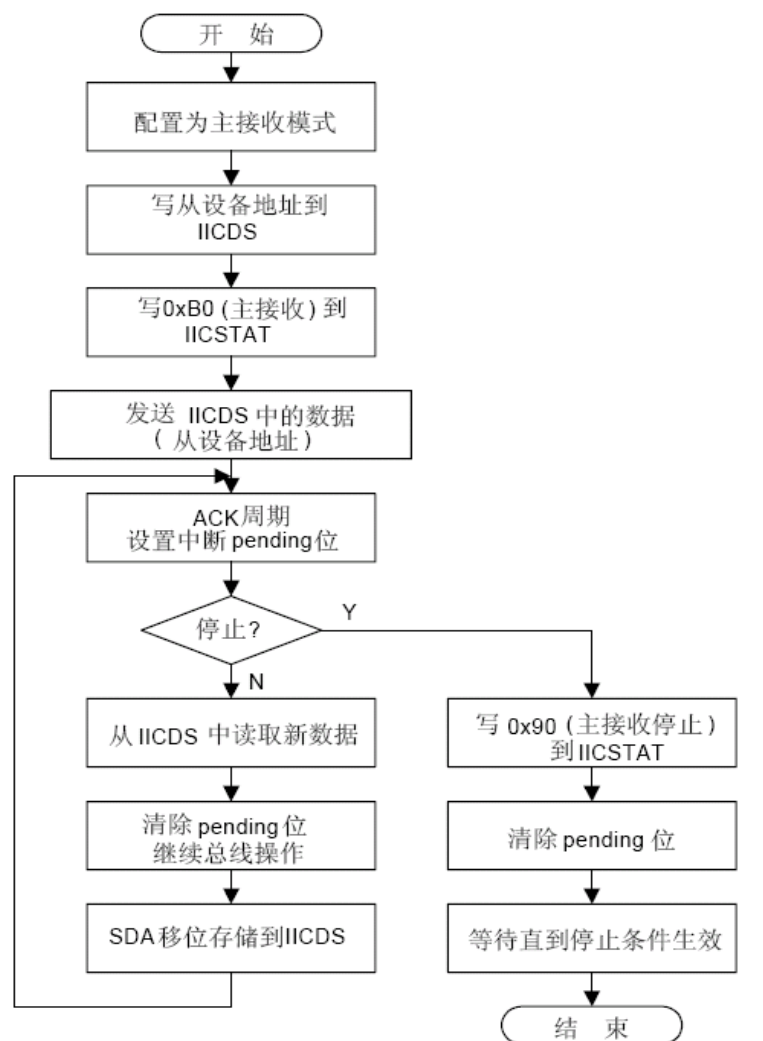


图 6-1-9 IIC 主接收程序设计流程图 (S3C2410X)

2. 电路设计

EduKit-III 实验平台中, 使用 S3C2410X 处理器内置的 IIC 控制器作为 IIC 通信主设备, AT24C04 EEPROM 为从设备。电路设计如图 6-1-9 所示:

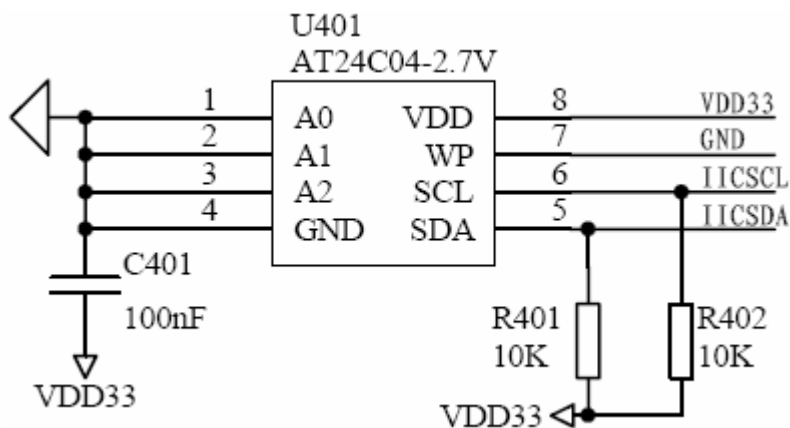


图 6-1-9 AT24C04 控制电路

6.1.6 实验操作步骤



1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接到目标板上。使用 Embest EduKit-III 实验板附带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 运行实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板，打开实验例程目录 6.1_i2c_test 子目录下的 i2c_test.Uv2 例程，编译链接工程；
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境（工程默认已经配置正确），点击工具栏 “”，在 Option for Target 对话框的 Linker 页中选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug Session 项或点击工具栏 “”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；
- 4) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；
- 5) 点击 Debug 菜单的 Go 或 F5 键运行程序。
- 6) 结合实验内容和实验原理部分，掌握在 S3C2410X 处理器中使用 IIC 接口访问 EEPROM 存储空间的编程方法。

6. 观察实验结果

在 PC 机上观察超级终端程序主窗口，可以看到如下界面：

```

Boot success

IIC operate Test Example

IIC Test using AT24C04...

Write char 0-f into AT24C04

Read 16 bytes from AT24C04

 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f

end.

```

7. 完成实验练习题

理解和掌握实验后，完成实验练习题。

6.1.7 实验参考程序

1. 初始化及测试主程序

```

/*-----*/
/*****
* name:      iic_test
* func:      test iic
* para:      none
* ret:       none
* modify:
* comment:
*****/

void iic_test(void)
{
    UINT8T      szData[16];
    UINT8T      szBuf[40];
    unsigned int  i, j;
    uart_printf("\n IIC Protocol Test Example, using AT24C04...\n");
    uart_printf(" Write char 0-f into AT24C04\n");
    f_nGetACK = 0;
    // Enable interrupt
    rINTMOD  = 0x0;
    rINTMSK &= ~BIT_IIC;
    pISR_IIC = (unsigned)iic_int_24c04;
    // Initialize iic
    rIICADD = 0x10;                // S3C2410X slave address
    rIICCON = 0xaf;                // Enable ACK, interrupt, SET IICCLK=MCLK/16
    rIICSTAT = 0x10;              // Enable TX/RX
    // Write 0 - 16 to 24C04
    for(i=0; i<16; i++)
    {
        iic_write_24c04(0xa0, i, i);
        delay(10);
    }
    // Clear array
    for(i=0; i<16; i++)
        szData[i]=0;
    // Read 16 byte from 24C04
    for(i=0; i<16; i++)
        iic_read_24c04(0xa0, i, &(szData[i]));
    // Printf read data
    uart_printf(" Read 16 bytes from AT24C04\n");
    for(i=0; i<16; i++)
    {
#ifdef BOARDTEST
        sprintf(szBuf, " %2x ", szData[i]);
        if(i<8)

```

```

        Lcd_DspAscll6x8(194+10*i,170,0xec,szBuf);
    else
        Lcd_DspAscll6x8(194+10*(i-8),180,0xec,szBuf);
#endif
    uart_printf(" %2x ", szData[i]);
}
rINTMSK |= BIT_IIC;
uart_printf("\n end.\n");
}

```

2. 中断服务程序

```

/*****
* name:      iic_int_24c04()
* func:      IIC interrupt handler
* para:      none
* ret:       none
* modify:
* comment:
*****/

void iic_int_24c04(void)
{
    ClearPending(BIT_IIC);
    f_nGetACK = 1;
}

```

3. IIC 写 AT24C04 程序

```

/*****
* name:      iic_write_24c040
* func:      write data to 24C040
* para:      unSlaveAddr --- input, chip slave address
*            unAddr      --- input, data address
*            ucData      --- input, data value
*****/

void iic_write_24c040(UINT32T unSlaveAddr,UINT32T unAddr,UINT8T ucData)
{
    f_nGetACK = 0;
    // Send control byte
    rIICDS = unSlaveAddr;                // 0xa0
    rIICSTAT = 0xf0;                    // Master Tx,Start
    while(f_nGetACK == 0);              // Wait ACK
    f_nGetACK = 0;
    //Send address
    rIICDS = unAddr;
    rIICCON = 0xaf;                      // Resumes IIC operation.
    while(f_nGetACK == 0);              // Wait ACK
    f_nGetACK = 0;
    // Send data

```



```

    rIICDS = ucData;
    rIICCON = 0xaf; // Resumes IIC operation.
    while(f_nGetACK == 0); // Wait ACK
    f_nGetACK = 0;
    // End send
    rIICSTAT = 0xd0; // Stop Master Tx condition
    rIICCON = 0xaf; // Resumes IIC operation.
    delay(5); // Wait until stop condition is in effect.
}

```

4. IIC 读 AT24C04 程序

```

/*****
* name:      read_24c040
* func:      read data from 24C040
* para:      unSlaveAddr --- input, chip slave address
*            unAddr      --- input, data address
*            pData       --- output, data pointer
* ret:       none
*****/

void read_24c040(UINT32T unSlaveAddr,UINT32T unAddr,UINT8T *pData)
{
    char cRecvByte;
    f_nGetACK = 0;
    rIICDS = unSlaveAddr; //IIC slave address is 0xa0
    rIICSTAT = 0xf0; //master send model, then start
    while(f_nGetACK == 0); //wait ACK
    f_nGetACK = 0;
    rIICDS = unAddr; //send data address
    rIICCON = 0xaf; //restart IIC process
    while(f_nGetACK == 0); //wait ACK
    f_nGetACK = 0;
    rIICDS = unSlaveAddr; //IIC slave address is 0xa0
    rIICSTAT = 0xb0; // master send model, then start
    rIICCON = 0xaf; // restart IIC process
    while(f_nGetACK == 0); //wait ACK
    f_nGetACK = 0;
    cRecvByte = rIICDS; //receive data
    rIICCON = 0x2f;
    delay(1);
    cRecvByte = rIICDS;
    rIICSTAT = 0x90; //stop master receive
    rIICCON = 0xaf; // restart IIC process
    delay(5); //wait till stop validate
    *pData = cRecvByte; //save received data to pData
}

```

6.1.8 练习题

编写程序往 AT24C04 芯片上存储某天日期字符串，再读出，并通过串口或液晶屏输出。

6.2 以太网通讯实验

6.2.1 实验目的

- 通过实验了解以太网通讯原理和驱动程序开发方法。
- 通过实验了解 IP 网络协议和网络应用程序开发方法。

6.2.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机，以太网集线器(Hub，可选)。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

6.2.3 实验内容

熟悉以太网控制器 CS8900A，在内部以太网局域网上基于 TFTP/IP 协议,下载代码到目标板上。

6.2.4 实验原理

1. 以太网通讯原理

以太网是由 Xerox 公司开发的一种基带局域网碰撞检测（CSMA/CD）机制，使用同轴电缆作为传输介质，数据传输速率达到 10M；使用双绞线作为传输介质，数据传输速率达到 100M/1000M。现在普遍遵从 IEEE802.3 规范。

1) 结构

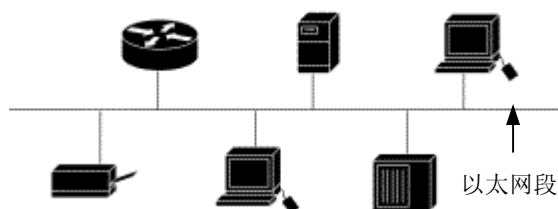


图 6-2-1 以太网结构示意图

2) 类型

以太网/IEEE802.3——采用同轴电缆作为网络媒体，传输速率达到 10Mbps；

100Mbps 以太网——又称为快速以太网，采用双绞线作为网络媒体，传输速率达到 100Mbps；

1000Mbps 以太网——又称为千兆以太网，采用光缆或双绞线作为网络媒体。

3) 工作原理

以太网的传输方法，也就是以太网的介质访问控制（MAC）技术称为载波监听多路存取和冲突检测（CSMA / CD），下面我们分步来说明其原理：

载波监听：当你所在的网站（计算机）要向另一个网站发送信息时，先监听网络信道上有无信息正在传输，信道是否空闲。

信道忙碌：如果发现网络信道正忙，则等待，直到发现网络信道空闲为止。

信道空闲：如果发现网络信道空闲，则向网上发送信息。由于整个网络信道为共享总线结构，网上所有网站都能够收到你所发出的信息，所以网站向网络信道发送信息也称为“广播”。但只有你想要发送数据的网站识别和接收这些信息。

冲突检测：网站发送信息的同时，还要监听网络信道，检测是否有另一台网站同时在发送信息。如果有，两个网站发送的信息会产生碰撞，即产生冲突，从而使数据信息包被破坏。

遇忙停发：如果发送信息的网站检测到冲突，则立即停止发送，并向网上发送一个“冲突”信号，让其它网站也发现该冲突，从而摒弃可能一直在接收受损的信息包。

多路存取：如果发送信息的网站因“碰撞冲突”而停止发送，就需等待一段时间，再回到第一步，重新开始载波监听和发送，直到数据成功发送为止。

所有共享型以太网上的网站，都是经过上述六步步骤，进行数据传输的。由于 CSMA / CD 介质访问控制法规定在同一时间里，只能有一个网站发送信息，其它网站只能收听和等待，否则就会产生“碰撞”。所以当共享型网络用户增加时，每个网站在发送信息时产生“碰撞”的概率增大，当网络用户增加到一定数目后，网站发送信息产生的“碰撞”会越来越多，想发送信息的网站不断地进行：监听—Λ 发送—Λ 碰撞—Λ 停止发送—Λ 等待—Λ 再监听—Λ 再发送……

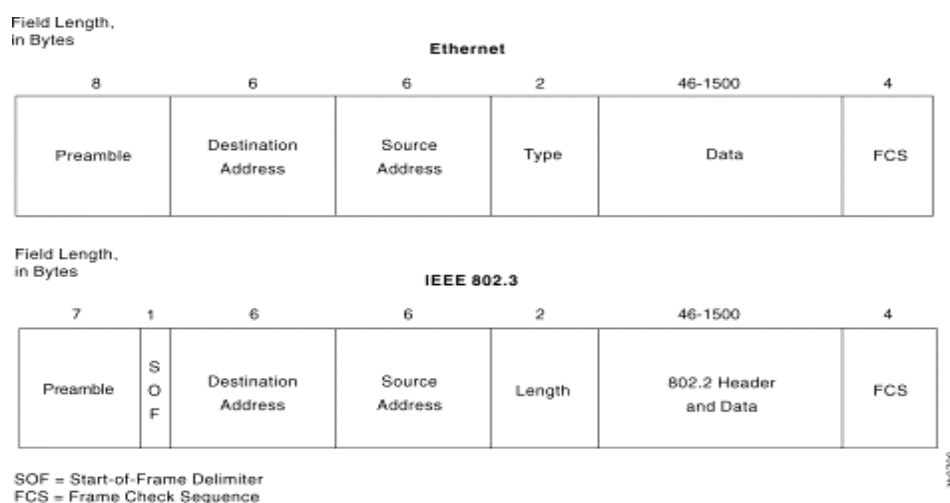


图 6-2-2 以太网/IEEE 802.3 帧的基本组成

4) 以太网/IEEE802.3 帧的结构

如上图所示，以太网和 IEEE802.3 帧的基本结构如下：

前导码：由 0、1 间隔代码组成，可以通知目标站作好接收准备。IEEE 802.3 帧的前导码占用 7 个字节，紧随其后的是长度为 1 个字节的帧首定界符（SOF）。以太网帧把 SOF 包含在了前导码当中，因此，前导码的长度扩大为 8 个字节。

帧首定界符（SOF）：IEEE 802.3 帧中的定界字节，以两个连续的代码 1 结尾，表示一帧的实际开始。

目标和源地址：表示发送和接收帧的工作站的地址，各占据 6 个字节。其中，目标地址可以是单址，也可以是多点传送或广播地址。

类型（以太网）：占用 2 个字节，指定接收数据的高层协议。

长度（IEEE 802.3）：表示紧随其后的以字节为单位的数据段的长度。

数据（以太网）：在经过物理层和逻辑链路层的处理之后，包含在帧中的数据将被传递给在类型段中指定的高层协议。虽然以太网版本 2 中并没有明确作出补齐规定，但是以太网帧中数据段的长度最小应当不低于 46 个字节。

数据（IEEE 802.3）：IEEE 802.3 帧在数据段中对接收数据的上层协议进行规定。如果数据段长度过小，使帧的总长度无法达到 64 个字节的最小值，那么相应软件将会自动填充数据段，以确保整个帧的长度不低于 64 个字节。

帧校验序列（FSC）：该序列包含长度为 4 个字节的循环冗余校验值（CRC），由发送设备计算产生，在接收方被重新计算以确定帧在传送过程中是否被损坏。

2. IP 网络协议原理

TCP/IP 协议是一组包括 TCP(Transmission Control Protocol)协议和 IP(Internet Protocol)协议，UDP (User Datagram Protocol) 协议、ICMP (Internet Control Message Protocol) 协议和其他一些协议的协议组。

1) 结构

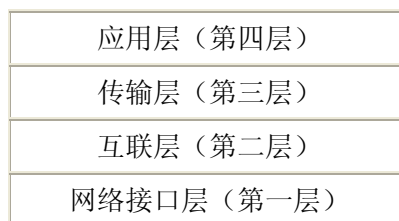


图 6-2-3 TCP/IP 协议层次

TCP/IP 协议采用分层结构，共分为四层，每一层独立完成指定功能，如上图所示：

网络接口层：负责接收和发送物理帧，它定义了将数据组成正确帧的规程和在网络中传输帧的规程，帧是指一串数据，它是数据在网络中传输的单位。网络接口层将帧放在网上，或从网上把帧取下来。

互联层：负责相邻结点之间的通信，本层定义了互联网中传输的“信息包”格式，以及从一个节点通过一个或多个路由器运行必要的路由算法到最终目标的“信息包”转发机制。主要协议有 IP、ARP、ICMP、IGMP 等。

传输层：负责起点到终点的通信，为两个用户进程之间建立、管理和拆除有效的端到端连接。主要协议有 TCP、UDP 等。

应用层：它定义了应用程序使用互联网的规程。应用程序通过这一层访问网络，主要遵从 BSD 网络应用接口规范。主要协议有 SMTP、FTP、TELNET、HTTP 等。

主要协议介绍

IP

网际协议 IP 是 TCP/IP 的心脏，也是网络层中最重要的协议。

IP 层接收由更低层（网络接口层例如以太网设备驱动程序）发来的数据包，并把该数据包发送到更高层——TCP 或 UDP 层；相反，IP 层也把从 TCP 或 UDP 层接收来的数据包传送到更低层。IP 数据包是不可靠的，因为 IP 并没有做任何事情来确认数据包是按顺序发送的或者没有被破坏。

当前 IP 协议有 IPv4 和 IPv6 两个版本，IPv4 正被广泛使用，IPv6 是下一代高速互联网的基础协议。下面这个表是 IPv4 的数据包格式：



	生存时间	协议	首部校验和	

	源 IP 地址			

	目的 IP 地址			

	选项			

	数据			

IP 协议头的结构定义如下：

```
struct    ip_header
{
    UIntip_v:4;           //协议版本
    UInt   ip_hl:4;       //协议头长度
    UInt8  ip_tos;        //服务类型
    UInt16 ip_len;        //数据包长度
    UInt16 ip_id;         //协议标识
    UInt16 ip_off;        //分段偏移域
    UInt8  ip_ttl;        //生存时间
    UInt8  ip_p;          //IP 数据包的高层协议
    UInt16 ip_sum;        //校验和
    Struct in_addr ip_src, ip_dst; //源和目的 IP 地址
};
```

ip_v: IP 协议的版本号, IPv4 为 4, IPv6 为 6。

ip_hl: IP 包首部长度,这个值以 4 字节为单位.IP 协议首部的固定长度为 20 个字节,如果 IP 包没有选项,那么这个值为 5。

ip_tos: 服务类型,说明提供的优先权。

ip_len: 说明 IP 数据的长度.以字节为单位。

ip_id: 标识这个 IP 数据包。

ip_off: 碎片偏移, 这和上面 ID 一起用来重组碎片的。

ip_ttl: 生存时间, 每经过一个路由时减一,直到为 0 时被抛弃。

ip_p: 协议, 表示创建这个 IP 数据包的高层协议.如 TCP,UDP 协议。

ip_sum: 首部校验和, 提供对首部数据的校验。

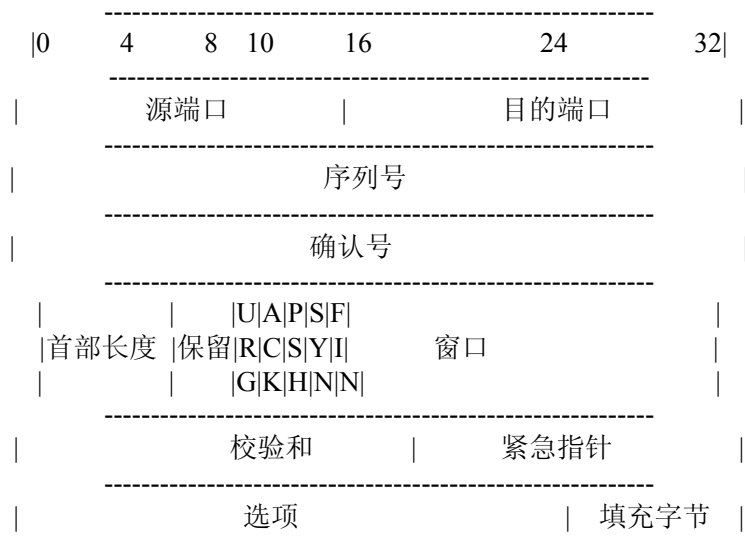
ip_src,ip_dst: 发送者和接收者的 IP 地址。

IP 地址实际上是采用 IP 网间网层通过上层软件完成“统一”网络物理地址的方法,这种方法使用统一的地址格式,在统一管理下分配给主机。Internet 网上不同的主机有不同的 IP 地址,在 IPv4 协议中,每个主机的 IP 地址都是由 32 比特,即 4 个字节组成的。为了便于用户阅读和理解,通常采用“点分十进制表示方法”表示,每个字节为一部分,中间用点号分隔开来。如 211.154.134.93 就是嵌入开发网 WEB 服务器的 IP 地址。每个 IP 地址又可分为两部分。网络号表示网络规模的大小,主机号表示网络中主机的地址编号。按照网络规模的大小,IP 地址可以分为 A、B、C、D、E 五类,其中 A、B、C 类是三种主要的类型地址,D 类专供多目传送用的多目地址,E 类用于扩展备用地址。

➤ TCP

如果 IP 数据包中有已经封好的 TCP 数据包,那么 IP 将把它们向“上”传送到 TCP 层。TCP 将包排序并进行错误检查,同时实现虚电路间的连接。TCP 数据包中包括序号和确认,所以未按照顺序收到的包可以被排序,而损坏的包可以被重传。

TCP 将它的信息送到更高层的应用程序,例如 Telnet 的服务程序和客户程序。应用程序轮流将信息送回 TCP 层,TCP 层便将它们向下传送到 IP 层,设备驱动程序和物理介质,最后到接收方。下面是 TCP 协议的数据包头格式:



关于 TCP 协议的详细情况,请查看 RFC793 文档。

TCP 对话通过三次握手来初始化。三次握手的目的是使数据段的发送和接收同步;告诉其它主机其一次可接收的数据量,并建立虚连接。

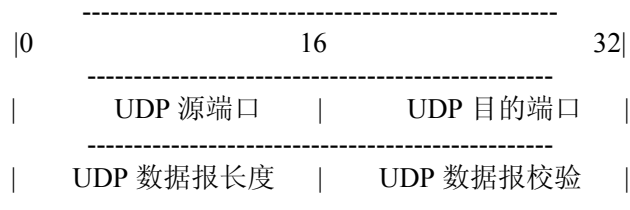
我们来看看这三次握手的简单过程:

- (1) 初始化主机通过一个同步标志置位的数据段发出会话请求。
- (2) 接收主机通过发回具有以下项目的数据段表示回复:同步标志置位、即将发送的数据段的起始字节的序号、应答并带有将收到的下一个数据段的字节序号。
- (3) 请求主机再回送一个数据段,并带有确认序号和确认号。

➤ UDP

UDP 与 TCP 位于同一层,但对于数据包的顺序错误或重发不处理。因此,UDP 不被应用于那些使用虚电路的面向连接的服务,UDP 主要用于那些面向查询---应答的服务,例如 NFS。相对于 FTP 或 Telnet,这些服务需要交换的信息量较小。使用 UDP 的服务包括 NTP(网落时间协议)和 DNS(DNS 也使用 TCP)。

UDP 协议的数据包头格式为:



UDP 协议适用于无须应答并且通常一次只传送少量数据的应用软件。

➤ ICMP

ICMP 与 **IP** 位于同一层，它被用来传送 **IP** 的控制信息。它主要是用来提供有关通向目的地址的路径信息。**ICMP** 的“**Redirect**”信息通知主机通向其他系统的更准确的路径，而“**Unreachable**”信息则指出路径有问题。另外，如果路径不可用了，**ICMP** 可以使 **TCP** 连接“体面地”终止。**PING** 是最常用的基于 **ICMP** 的服务。

➤ **ARP**

要在网络上通信，主机就必须知道对方主机的硬件地址。地址解析就是将主机 **IP** 地址映射为硬件地址的过程。地址解析协议 **ARP** 用于获得在同一物理网络中的主机的硬件地址。

解释本地网络 **IP** 地址过程：

(1) 当一台主机要与别的主机通信时，初始化 **ARP** 请求。当该 **IP** 断定 **IP** 地址是本地时，源主机在 **ARP** 缓存中查找目标主机的硬件地址。

(2) 要是找不到映射的话，**ARP** 建立一个请求，源主机 **IP** 地址和硬件地址会被包括在请求中，该请求通过广播，使所有本地主机均能接收并处理。

(3) 本地网上的每个主机都收到广播并寻找相符的 **IP** 地址。

(4) 当目标主机断定请求中的 **IP** 地址与自己的相符时，直接发送一个 **ARP** 答复，将自己的硬件地址传给源主机。以源主机的 **IP** 地址和硬件地址更新它的 **ARP** 缓存。源主机收到回答后便建立起了通信。

➤ **TFTP** 协议

TFTP 是一个传输文件的简单协议，一种简化的 **TCP/IP** 文件传输协议，它基于 **UDP** 协议而实现，支持用户从远程主机接收或向远程主机发送文件。此协议设计的时候是进行小文件传输的。因此它不具备通常的 **FTP** 的许多功能，它只能从文件服务器上获得或写入文件，不能列出目录，不进行认证，它传输 8 位数据。

因为 **TFTP** 使用 **UDP**，而 **UDP** 使用 **IP**，**IP** 还可以使用其它本地通信方法。因此一个 **TFTP** 包中会有以下几段：本地媒介头，**IP** 头，数据报头，**TFTP** 头，剩下的就是 **TFTP** 数据了。**TFTP** 在 **IP** 头中不指定任何数据，但是它使用 **UDP** 中的源和目标端口以及包长度域。由 **TFTP** 使用的包标记 (**TID**) 在这里被用做端口，因此 **TID** 必须介于 0 到 65,535 之间。

初始连接时候需要发出 **WRQ** (请求写入远程系统) 或 **RRQ** (请求读取远程系统)，收到一个确定应答，一个确定的可以写出的包或应该读取的第一块数据。通常确认包包括要确认的包的包号，每个数据包都与一个块号相对应，块号从 1 开始而且是连续的。因此对于写入请求的确定是一个比较特殊的情况，因此它的包的包号是 0。如果收到的包是一个错误的包，则这个请求被拒绝。创建连接时，通信双方随机选择一个 **TID**，因为是随机选择的，因此两次选择同一个 **ID** 的可能性就很小了。每个包包括两个 **TID**，发送者 **ID** 和接收者 **ID**。在第一次请求的时候它会将请求发到 **TID 69**，也就是服务器的 69 端口上。应答时，服务器使用一个选择好的 **TID** 作为源 **TID**，并用上一个包中的 **TID** 作为目的 **ID** 进行发送。这两个被选择的 **ID** 在随后的通信中会被一直使用。

此时连接建立，第一个数据包以序列号 1 从主机开始发出。以后两台主机要保证以开始时确定的 **TID** 进行通信。如果源 **ID** 与原来确定的 **ID** 不一样，这个包会被认为发送到了错误的地址而被抛弃。

网络应用程序开发方法

进行网络应用程序开发有两种方法：一是采用 **BSD Socket** 标准接口，程序移植能力强；二是采用专用接口直接调用对应的传输层接口，效率较高。

➤ **BSD Socket** 接口编程方法

Socket (套接字) 是通过标准的文件描述符和其它程序通讯的一个方法。每一个套接字都用一个半相关描述：{协议，本地地址、本地端口}来表示；一个完整的套接字则用一个相关描述：{协议，

本地地址、本地端口、远程地址、远程端口}，每一个套接字都有一个本地的由操作系统分配的唯一套接字号。

► 传输层专有接口编程方法

网络协议都可以直接提供专有函数接口给上层或者跨层调用，用户可以调用每个协议代码中特有的接口实现快速数据传递。

实验板的网络协议包提供的就是 TFTP 协议的专用接口，应用程序可以通过它接收用户从主机上使用 TFTP 传递过来的数据。主要接口函数有：

extern char *TftpRecv(int* len)——接收用户数据，网络协议包自动完成连接过程，并从网络获取用户传递过来的数据包，每次接收最大长度由 len 指定，返回前改写成实际接收到的数据长度，函数返回数据首地址指针，若返回值为 NULL，表示接收故障。

MakeAnswer()——返回应答信号，每当用户处理完一个数据包后，需要调用此函数给对方一个确认信号，使得数据传输得以继续。

3. 关于 CS8900A

CS8900A 特点介绍

CS8900A 是由美国 CIRRUS LOGIC 公司生产的以太网控制器，由于其优良的性能、低功耗及低廉的价格，使其在市场上 10Mbps 嵌入式网络应用中占有相当的比例。

CS8900A 主要性能为：

- (1) 符合 Ethernet II 与 IEEE802.3 (10Base5、10Base2、10BaseT) 标准；
- (2) 全双工，收发可同时达到 10Mbps 的速率；
- (3) 内置 SRAM，用于收发缓冲，降低对主处理器的速度要求；
- (4) 支持 16 位数据总线，4 个中断申请线以及三个 DMA 请求线；
- (5) 8 个 I/O 基地址，16 位内部寄存器，IO Base 或 Memory Map 方式访问；
- (6) 支持 UTP、AUI、BNC 自动检测，还支持对 10BaseT 拓扑结构的自动极性修正；
- (7) LED 指示网络激活和连接状态；
- (8) 100 脚的 LQFP 封装，缩小了 PCB 尺寸。

► CS8900A 引脚分布

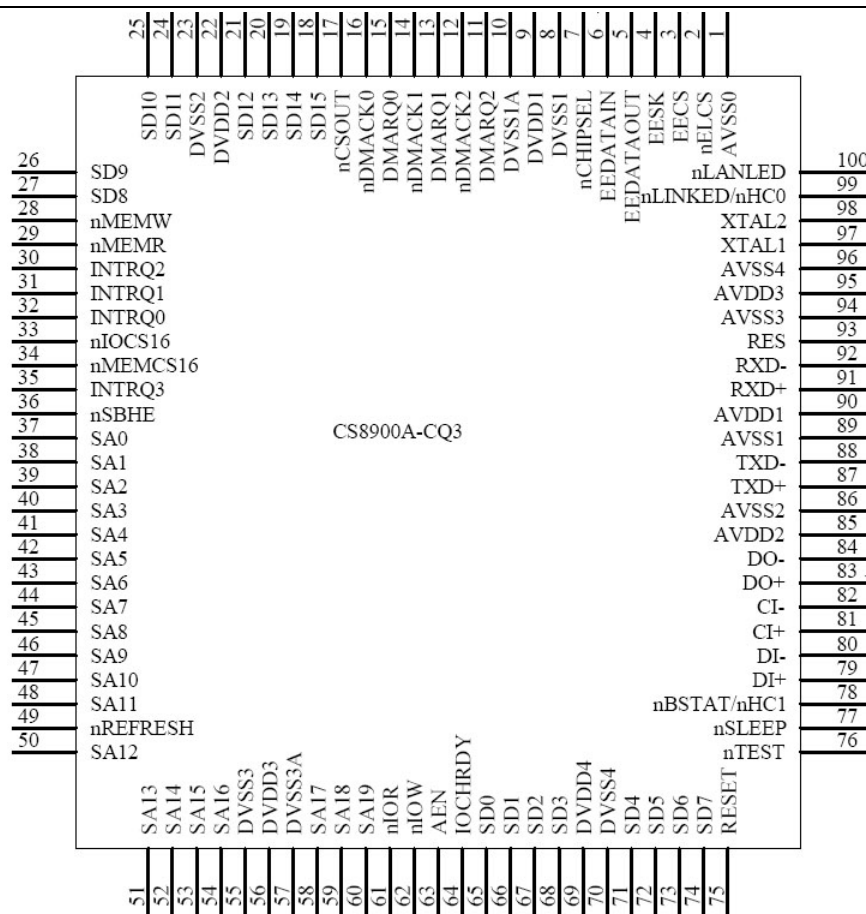


图 6-2-4 CS8900A 的引脚分布图

➤ 复位与初始化过程

引起 CS8900A 复位的因素有很多种，有人为的需要，也有意外产生的复位。如外部复位信号（在 RESET 引脚加至少 400ns 的高电平）引起复位，上电自动复位，下电复位（电压低于 2.5V），EEPROM 校验失败引起复位以及软件复位等。复位之后，CS8900A 需要重新进行配置。

每次复位之后（除 EEPROM 校验失败引起复位以外），CS8900A 都会检查 EEDaIn 引脚，判断是否有外部的 EEPROM 存在。如果 EEDI 是高电平，则说明 EEPROM 存在，CS8900A 会自动将 EEPROM 中的数据加载到内部寄存器中(Embest EduKit-III 实验平台未采用外接 EEPROM 的方法，具体的配置过程可以参考 CS8900A 的硬件手册)；如果 EEDI 为低电平，则 EEPROM 不存在，CS8900A 会按照下表所示进行默认的配置。

表 6-2-1 CS8900A 的默认配置

PacketPage 地址	寄存器内容	寄存器描述
0020h	0300h	I/O 基地址
0022h	XXXX XXXX XXXX X100	中断号
0024h	XXXX XXXX XXXX XX11	DMA 通道号
0026h	0000h	DMA 基址到待发送帧的偏移
0028h	X000h	DMA 帧数
002Ah	0000h	DMA 字节数
002Ch	XXX0 0000h	内存基地址
0030h	XXX0 0000h	引导 PROM 基地址

0034h	XXX0 0000h	引导 PROM 地址掩码
0102h	0003h	寄存器 3—RxCFG
0104h	0005h	寄存器 5—RxCTL
0106h	0007h	寄存器 7—TxCFG
0108h	0009h	寄存器 9—TxCMD
010Ah	000Bh	寄存器 B—BufCFG
010Ch	Undefined	保留
010Eh	Undefined	保留
0110h	Undefined	保留
0112h	00013h	寄存器 13—LineCTL
0114h	0015h	寄存器 15—SelfCTL
0116h	0017h	寄存器 17—BusCTL
0118h	0019h	寄存器 19--TestCTL

➤ PACKETPAGE 结构

CS8900A 的结构的核心是提供高效访问方法的内部寄存器和缓冲内存。

PacketPage 是 CS8900A 中集成的 RAM。它可以用作接收帧和待发送帧的缓冲区，除此之外还有一些其他的用途。PacketPage 为内存或 I/O 空间提供了一种统一的访问控制方法，减轻了 CPU 的负担，降低了软件开发的难度。此外，它还提供一系列灵活的配置选项，允许开发者根据特定的系统需求设计自己的以太网模块。PacketPage 中可以供用户操作的部分可以划分为以下几个部分：

0000h—0045h	总线接口寄存器
0100h—013Fh	状态与控制寄存器
0140h—014Fh	发送初始化寄存器
0150h—015Dh	地址过滤寄存器
0400h	接收帧地址
0A00h	待发送帧地址

（具体地址及定义见实验参考程序 1）

4. CS8900A 工作模式介绍

CS8900A 有两种工作模式，一种是 I/O 访问方式，一种是内存访问方式。网卡芯片复位后默认工作方式为 I/O 连接，I/O 端口基址为 300H，下面对它的几个主要工作寄存器进行介绍（寄存器后括号内的数字为寄存器地址相对基址 300H 的偏移量）。

LINECTL (0112H) LINECTL 决定 CS8900 的基本配置和物理接口。在本系统中，设置初始值为 00d3H，选择物理接口为 10BASE-T，并使能设备的发送和接收控制位。

RXCTL (0104H) RXCTL 控制 CS8900 接收特定数据报。设置 RXTCL 的初始值为 0d05H，接收网络上的广播或者目标地址同本地物理地址相同的正确数据报。

RXCFG (0102H) RXCFG 控制 CS8900 接收到特定数据报后会引发接收中断。RXCFG 可设置为 0103H，这样当收到一个正确数据报后，CS8900 会产生一个接收中断。

BUSCT (0116H) BUSCT 可控制芯片的 I/O 接口的一些操作。设置初始值为 8017H，打开 CS8900 的中断总控制位。

ISQ (0120H) ISQ 是网卡芯片的中断状态寄存器，内部映射接收中断状态寄存器和发送中断状态寄存器的内容。

PORT0 (0000H) 发送和接收数据时，CPU 通过 PORT0 传递数据。

TXCMD (0004H) 发送控制寄存器，如果写入数据 00C0H，那么网卡芯片在全部数据写入后开始发送数据。

TXLENG (0006H) 发送数据长度寄存器，发送数据时，首先写入发送数据长度，然后将数据通过 PORT0 写入芯片。

以上为几个最主要的工作寄存器（为 16 位），CS8900 支持 8 位模式，当读或写 16 位数据时，低位字节对应偶地址，高位字节对应奇地址。例如，向 TXCMD 中写入 00C0H，则可将 00h 写入 305H，将 C0H 写入 304H。

系统工作时，应首先对网卡芯片进行初始化，即写寄存器 LINECTL、RXCTL、RCCFG、BUSCT。发数据时，写控制寄存器 TXCMD，并将发送数据长度写入 TXLENG，然后将数据依次写入 PORT0 口，如将第一个字节写入 300H，第二个字节写入 301H，第三个字节写入 300H，依此类推。网卡芯片将数据组织为链路层类型并添加填充位和 CRC 校验送到网络同样，处理器查询 ISO 的数据，当有数据来到后，读取接收到的数据帧。

5. CS8900A 驱动程序设计

以太网驱动程序是针对实验板上的以太网接口芯片 CS8900A 编程，正确初始化芯片，并提供数据输入输出和控制接口给高层网络协议使用。

源码中，cs8900a.c 文件是 CS8900A 的驱动程序，函数功能如下：

CS_Init()——CS8900A 初始化。初始化步骤为：①检测 CS8900A 芯片是否存在，然后软件复位 CS8900A；②如果使用 Memory Map 方式访问 CS8900A 芯片内部寄存，就设置 CS8900A 内部寄存基地址（默认为 IO 方式访问）；③设置 CS8900A 的 MAC 地址；④关闭事件中断（本例子使用查询方式，如果使用中断方式，则添加中断服务程序再打开 CS8900A 中断允许位）；⑤配置 CS8900A 10BT，然后允许 CS8900A 接收和发送，接收发送 64-1518 字节的网络帧及网络广播帧（见实验参考程序 2）。

CS_Close()——关闭 CS8900A 芯片数据收发功能，及关闭中断请求。

CS_Reset()——复位 CS8900A 芯片。

CS_Identification()——获得 CS8900A 芯片 ID 和修订版本号。

CS_TransmitPacket ()——数据包输出。将要发送的网络数据包从网口发送出去。发送数据包时，先把发送命令写到发送命令寄存器，把发送长度写到发送长度寄存器，然后等待 CS8900A 内部总线状态寄存器发送就绪位置位，便将数据包的数据顺序写到数据端口寄存器（16 位宽，一次两个字节）。

CS_ReceivePacket ()——数据包接收。查询数据接收事件寄存器，若有数据帧接收就绪，读取接收状态寄存器（与接收事件寄存器内容一致，忽略之），及读取接收长度寄存器，得到数据帧的长度，然后从数据端口寄存器顺序读取数据（16 位宽，一次两个字节）。

6.2.5 实验操作步骤

1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接到目标板上，使用 Embest EduKit-III 实验板自带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 配置

- 1) 将 PC 机 IP 地址设置为 192.192.192.x (x 取值在 30-200 之间)。
- 2) 运行 PC 机 DOS 窗口或者“开始”系统按钮上的“运行”菜单，输入命令 command:

```
C:\DOCUME~1\SS>cd \
```

```
C:\>arp -s 192.192.192.200 00-06-98-01-7e-8f
```

```
C:\>arp -a
```



```
Interface: 192.192.192.36 --- 0x2
```

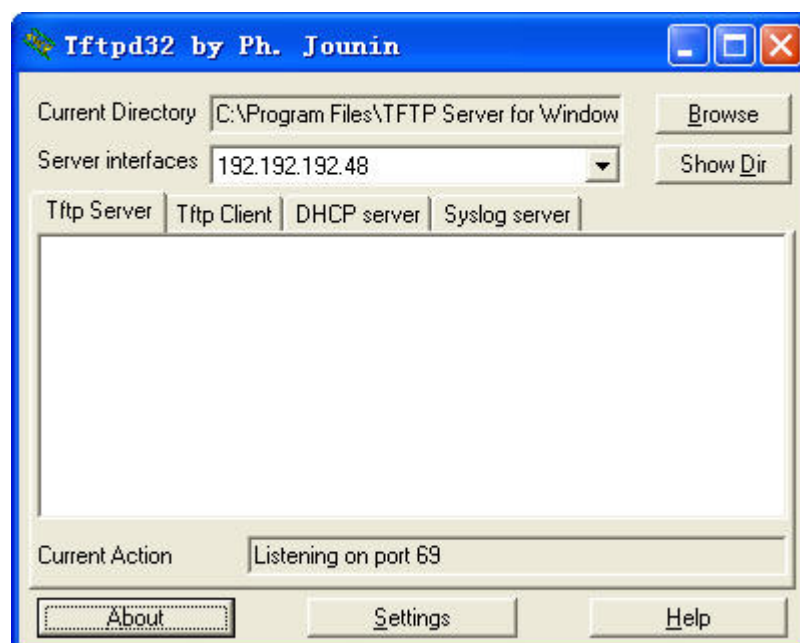
Internet Address	Physical Address	Type
192.192.192.200	00-06-98-01-7e-8f	static

为 PC 机添加一个到目标板的地址解析。

- 3) 在 PC 机上运行 windows 自带的超级终端串口通信程序 (波特率 115200、1 位停止位、无校验位、无硬件流控制); 或者使用其它串口通信程序。使用教学系统提供的交叉网线连接开发板网口 (NET1) 和 PC 机; 也可以使用直通网线, 连接到与 PC 同一局域网的 HUB 上。

4. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下 (如果已经拷贝, 可跳过此步骤);
- 2) 使用 ULINK USB-JTAG 仿真器连接到目标板上, 打开实验例程目录 6.2_tftp_test 子目录下的 tftp_test.Uv2 例程, 编译链接工程, 直到链接工程成功;
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境 (工程默认已经配置正确), 点击工具栏 “”, 在 Option for Target 对话框的 Linker 页中选择 RuninRAM.sct 分散加载文件, 点击 MDK 的 Debug 菜单, 选择 Start/Stop Debug Session 项或点击工具栏 “”, 下载工程生成的.axf 文件到目标板的 RAM 中调试运行;
- 4) 如果需要将程序烧写固化到 Flash 中, 仅需要更改分散加载文件即可 (**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序, 建议实验中不操作**)。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件, 重新编译工程, 点击 MDK 的 Flash 菜单, 选择 Download 烧写调试代码到目标系统的 Nor Flash 中, 重启实验板, 实验板将会运行烧写到 Nor Flash 中的代码;
- 5) 点击 Debug 菜单的 Run 或 F5 键全速运行程序;
- 6) 运行光盘中附带的 DHCP Server 软件 (Tools\tftpd32\tftpd32.exe), 以动态分配 IP 地址 (也可用网络中的 DHCP 服务器):



在 PC 上观察超级终端程序主窗口，可以看到如下界面：

```
boot success...

Reset CS8900A successful,   Rev F.

S/s -- DHCP IP addr

D/d -- Default IP addr(192.192.192.200)

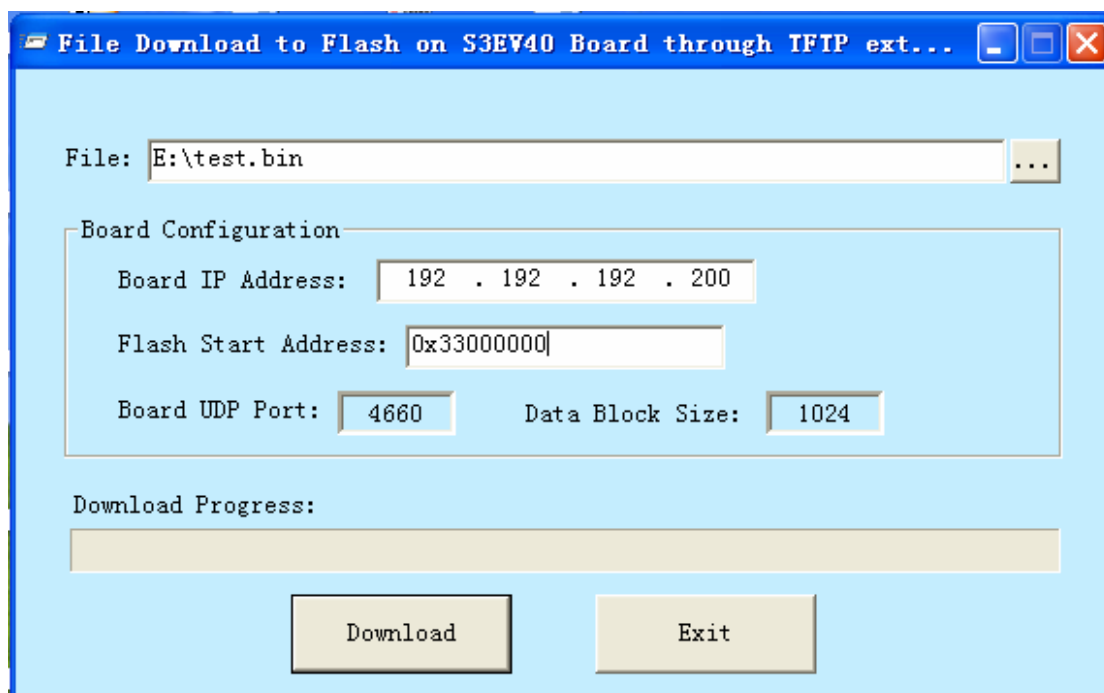
Y/y -- Input New IP addr
```

选择 s，表示 DHCP 测试，选择 d，默认 IP，选择 y，输入新 IP，超级终端显示如下：

```
Press a key to continue ... s
Set local ip 192.192.192.201
Press any key to exit ...
Press a key to continue ... d
Set local ip 192.192.192.200
Press any key to exit ...
Press a key to continue ... y
Please input IP address(xxx.xxx.xxx.xxx)
then press ENTER:192.192.192.218
Set local ip 192.192.192.218
Press any key to exit ...
```

图 6-2-6 超级终端显示的程序运行状态

- 8) 在 PC 上运行 TFTPDown.exe 程序（在 CD1 根目录 Tools 下），目标板地址输入 192.192.192.200, Flash Start Address 输入 0x33000000, 然后选取想要下载的文件（BIN 文件、ELF 文件等各种文件均可），点击 Download, 程序开始通过 TFTP 协议下载文件到目标板 Flash 中，成功或者出错都有提示对话框。



注：Flash Start Address 也可以是 SDRAM 地址：0x30000000~0x37ffc00

图 6-2-7 TFTPDownload 运行后的配置界面

用户按照上图参数设置完毕后，点击 Download 按钮，TFTP 正确下载会出现以下提示：

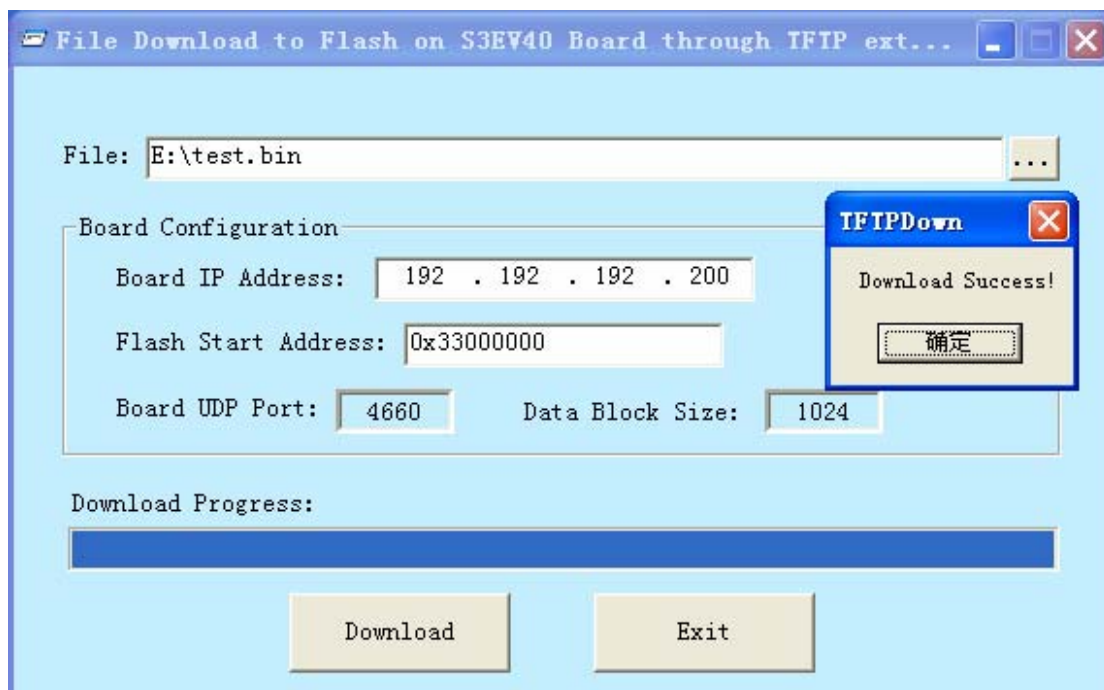


图 6-2-8 TFTP Download 测试成功图

当开发板所在的网络不能正常通信或开发板硬件不正常时，TFTP 文件下载会显示下载失败提示。用户可以先检查网络通信是否正常再进行调试，直到通信成功。

5. 观察实验结果

使用 MDK 停止目标板运行，打开 Memory 窗口，输入 0x33000000，然后检查 Flash 中的数据是否和下载的文件数据一致。

6. 完成实验练习题

理解和掌握实验后，完成实验练习题。

6.2.6 实验参考程序

1. CS8900A 的寄存器定义

```
/*          CS8900A Register defines          */
// Bus interface registers
#define REG_Identification      0x0000
#define REG_IOBaseAddress      0x0020
#define REG_InterruptNumber    0x0022
#define REG_DMACHannelNumber    0x0024
#define REG_DMAStartOfFrame     0x0026
#define REG_DMARframeCount      0x0028
#define REG_RxDMAByteCount      0x002A
#define REG_MemoryBaseAddress   0x002C
#define REG_BootBaseAddress     0x0030
#define REG_BootAddressMask     0x0034
#define REG_EEPROMCommand       0x0040
#define REG_EEPROMData          0x0042
#define REG_ReceivedByteCounter 0x0050

// Status and control registers
#define REG_RxCFG                0x0102
#define REG_RxCTL                0x0104
#define REG_TxCFG                0x0106
#define REG_TxCMD                0x0108
#define REG_BufCFG               0x010A
#define REG_LineCTL              0x0112
#define REG_SelfCTL              0x0114
#define REG_BusCTL               0x0116
#define REG_TestCTL              0x0118

#define REG_ISQ                  0x0120
#define REG_RxEvent              0x0124
#define REG_TxEvent              0x0128
#define REG_BufEvent             0x012C
#define REG_RxMISS               0x0130
#define REG_TxCOL                0x0132
#define REG_LineST               0x0134
#define REG_SelfST               0x0136
```

```

#define REG_BusST          0x0138
#define REG_TDR            0x013C

// Initiate transmit registers
#define REG_TxCommand      0x0144
#define REG_TxLength       0x0146

// Address filter registers
#define REG_LAF            0x0150
#define REG_IA             0x0158

// Frame location register
#define REG_RxStatus       0x0400
#define REG_RxLength       0x0402
#define REG_RxFrame        0x0404
#define REG_TxFrame        0x0A00

```

2. CS8900A 初始化程序

```

/*****
* name:      CS_Init
* func:      initialize CS8900A chip
* para:      none
* ret:       1: initialize success; 0: initialize fail
* modify:
* comment:
*****/

int CS_Init(void)
{
    unsigned short id, rev;
    CS_Probe();                // detailed function reference CS8900A.c
    if(!CS_Reset())
    {
        uart_printf(" Reset CS8900A failed.\n");
        return 0;
    }
    uart_printf("Reset CS8900A successful, ");
    #ifdef MEM_MODE
        CS_SetupMemoryBase(MEM_BASE);
    #endif
    id = CS_Identification(&rev);
    switch(rev)
    {
        case REV_B: uart_printf(" Rev B.\n"); break;
        case REV_C: uart_printf(" Rev C.\n"); break;
        case REV_D: uart_printf(" Rev D.\n"); break;
    }
}

```



```

        case REV_F: uart_printf(" Rev F.\n"); break;
    }
    CS_SetupMacAddr(mac_addr);
    CS_EnableIrq(0);
    CS_Configuration();
    return 1;
}

```

3. 相关协议数据结构

//UDP 协议头结构

struct UDPHDR

```

{
    USHORT uh_sport;           /* Source port      */
    USHORT uh_dport;           /* Destination port */
    USHORT uh_ulen;             /* UDP length       */
    USHORT uh_sum;              /* UDP checksum     */
};

```

//TFTP 协议头结构

struct TFTP HDR

```

{
    short th_opcode;           /* packet type      */
    short tu_block;            /* block            */
};

```

//DHCP 协议头结构

struct DHCP HDR

```

{
    UCHAR  bp_op;               /* Packet opcode type: 1=request, 2=reply */
    UCHAR  bp_htype;            /* Hardware address type: 1=Ethernet */
    UCHAR  bp_hlen;             /* Hardware address length: 6 for Ethernet */
    UCHAR  bp_hops;             /* Gateway hops */
    ULONG  bp_xid;              /* Transaction ID */
    USHORT bp_secs;             /* Seconds since boot began */
    USHORT bp_flags;            /* RFC1532 broadcast, etc. */
    ULONG  bp_ciaddr;           /* Client IP address */
    ULONG  bp_yiaddr;           /* 'Your' IP address */
    ULONG  bp_siaddr;           /* Server IP address */
    ULONG  bp_giaddr;           /* Gateway IP address */
    UCHAR  bp_chaddr[16];       /* Client hardware address */
    char   bp_sname[64];        /* Server host name */
    char   bp_file[128];        /* Boot file name */
    UCHAR  bp_options[312];     /* Vendor-specific area */
};

```

6.2.7 练习题

改写 TFTP_TEST 例程，改变实验板 IP 地址，下载的时候改变 Flash 起始地址，重新进行实验，检查是否正确下载。

6.3 音频接口 IIS 实验

6.3.1 实验目的

- 掌握有关音频处理的基础知识。
- 通过实验了解 IIS 音频接口的工作原理。
- 通过实验掌握对处理器 S3C2410 X 中 IIS 模块电路的控制方法。
- 通过实验掌握对常用 IIS 接口音频芯片的控制方法。

6.3.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

6.3.3 实验内容

编写程序播放一段由 wav 文件保存的录音。

6.3.4 实验原理

1. 数字音频基础

采样频率和采样精度

在数字音频系统中，通过将声波波形转换成一连串的二进制数据再现原始声音，这个过程中使用的设备是模拟/数字转换器（Analog to Digital Converter，即 ADC），ADC 以每秒上万次的速率对声波进行采样，每次采样都记录下了原始声波在某一时刻的状态，称之为样本。

每秒采样的数目称为采样频率，单位为 HZ（赫兹）。采样频率越高所能描述的声波频率就越高。系统对于每个样本均会分配一定存储位（bit 数）来表达声波的声波振幅状态，称之为采样精度。采样频率和精度共同保证了声音还原的质量。

人耳的听觉范围通常是 20Hz~20KHz，根据奈魁斯特（NYQUIST）采样定理，用两倍于一个正弦波的频率进行采样能够真实地还原该波形，因此当采样频率高于 40KHz 时可以保证不产生失真。CD 音频的采样规格为 16bit，44KHz，就是根据以上原理制定。

音频编码

脉冲编码调制 PCM（Pulse Code Modulation）编码的方法是对语音信号进行采样，然后对每个样值进行量化编码，在“采样频率和采样精度”中对语音量化和编码就是一个 PCM 编码过程。ITU-T 的 64kbit / s 语音编码标准 G.711 采用 PCM 编码方式，采样速率为 8KHz，每个样值用 8bit 非线性的 μ 律或 A 律进行编码，总速率为 64kbit / s。

CD 音频即是使用 PCM 编码格式，采样频率 44KHz，采样值使用 16bit 编码。

使用 PCM 编码的文件在 Windows 系统中保存的文件格式一般为大家熟悉的 wav 格式，实验中用到的就是一个采样 44.100KHz，16 位立体声文件 t.wav。

在 PCM 基础上发展起来的还有**自适应差分脉冲编码调制 ADPCM**（Adaptive Differential Pulse Code Modulation）。ADPCM 编码的方法是对输入样值进行自适应预测，然后对预测误差进行量化编码。CCITT 的 32kbit / s 语音编码标准 G.721 采用 ADPCM 编码方式，每个语音采样值相当于使用 4bit 进行编码。

其他编码方式还有线性预测编码 LPC (Linear Predictive Coding)，低时延码激励线性预测编码 LD-CELP (Low Delay-Code Excited Linear Prediction) 等。

目前流行的一些音频编码格式还有 MP3 (MPEG Audio Layer-3)，WMA (Windows Media Audio)，RA (RealAudio)，它们有一个共同特点就是压缩比高，主要针对网络传输，支持边读边放。

2. IIS 音频接口

IIS (Inter-IC Sound) 是一种串行总线设计技术，是 SONY、PHILIPS 等电子巨头共同推出的接口标准，主要针对数字音频处理技术和设备如便携 CD 机、数字音频处理器等。IIS 将音频数据和时钟信号分离，避免由时钟带来的抖动问题，因此系统中不再需要消除抖动的器件。

IIS 总线仅处理音频数据，其它信号如控制信号等单独传送，基于减少引脚数目和布线简单的目的，IIS 总线只由三根串行线组成：时分复用的数据通道线，字选择线和时钟线。

continuous serial clock (SCK);

word select (WS);

serial data (SD);

使用 IIS 技术设计的系统的连接配置参见图 6-3-1。

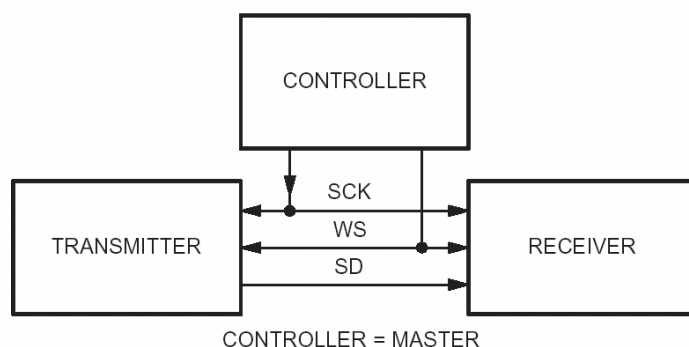


图 6-3-1 IIS 系统连接简单配置图

IIS 总线接口的基本时序参见图 6-3-2。

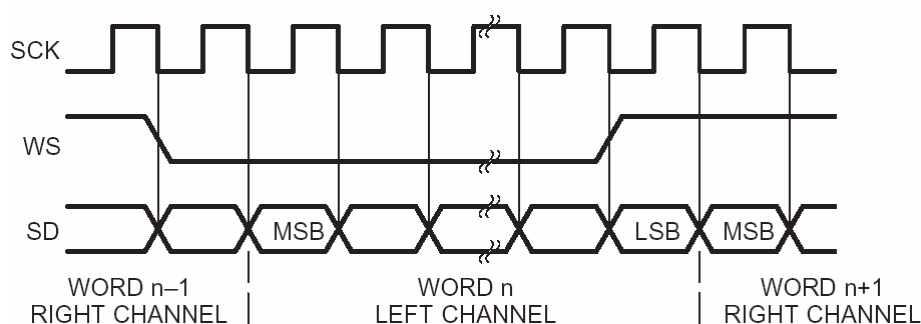


图 6-3-2 IIS 接口基本时序图

WS 信号线指示左通道或右通道的数据将被传输，SD 信号线按高有效位 MSB 到低有效位 LSB 的顺序传送字长的音频数据，MSB 总在 WS 切换后的第一个时钟发送，如果数据长度不匹配，接收器和发送器将自动截取或填充。关于 IIS 总线的其它细节可参见 [《I2S bus specification》](#)。

在实验中，IIS 总线接口由处理器 S3C2410 的 IIS 模块和音频芯片 UDA1341 硬件实现，我们需要关注的是正确的配置 IIS 模块和 UDA1341 芯片，音频数据的传输反而比较简单。

3. 电路设计原理

S3C2410X 外围模块 IIS 说明

(1) 信号线

处理器中与 IIS 相关的信号线有五根：

- 串行数据输入 IISDI，对应 IIS 总线接口中的 SD 信号，方向为输入。
- 串行数据输出 IISDO，对应 IIS 总线接口中的 SD 信号，方向为输出。
- 左右通道选择 IISLRCK，对应 IIS 总线接口中的 WS 信号，即采样时钟。
- 串行位时钟 IISCLK，对应 IIS 总线接口中的 SCK 信号。
- 音频系统主时钟 CODECLK，一般为采样频率的 256 倍或 384 倍，符号为 256fs 或 384fs，其中 fs 为采样频率。CODECLK 通过处理器主时钟分频获得，可以通过在程序中设定分频寄存器获取，分频因子可以设为 1 到 32。CODECLK 与采样频率的对应表格见下表，实验中需要正确的选择 IISLRCK 和 CODECLK。

表 6-3-1 音频主时钟和采样频率的对应表

IISLRCK (fs)	8.000 KHz	11.025 KHz	16.000 KHz	22.050 KHz	32.000 KHz	44.100 KHz	48.000 KHz	64.000 KHz	88.200 KHz	96.000 KHz
CODECLK (MHz)	256fs									
	2.0480	2.8224	4.0960	5.6448	8.1920	11.2896	12.2880	16.3840	22.5792	24.5760
	384fs									
	3.0720	4.2336	6.1440	8.4672	12.2880	16.9344	18.4320	24.5760	33.8688	36.8640

需要注意的是，处理器主时钟可以通过配置锁相环寄存器进行调整，结合 CODECLK 的分频寄存器设置，可以获得所需要的 CODECLK。

(2) 寄存器

处理器中与 IIS 相关的寄存器有三个：

IIS 控制寄存器 IISCON，通过该寄存器可以获取数据高速缓存 FIFO 的准备就绪状态，启动或停止发送和接收时的 DMA 请求，使能 IISLRCK、分频功能和 IIS 接口。

IIS 模式寄存器 IISMOD，该寄存器选择主/从、发送/接收模式，设置有效电平、通道数据位，选择 CODECLK 和 IISLRCK 频率。

IIS 分频寄存器 IISPSR。

(3) 数据传送

数据传送可以选择普通模式或者 DMA 模式，普通模式下，处理器根据 FIFO 的准备就绪状态传送数据到 FIFO，处理器自动完成数据从 FIFO 到 IIS 总线的发送，FIFO 的准备就绪状态通过 IIS 的 FIFO 控制寄存器 IISFCON 获取，数据直接写入 FIFO 寄存器 IISFIF。DMA 模式下，对 FIFO 的访问和控制完全由 DMA 控制器完成，DMA 控制器自动根据 FIFO 的状态发送或接收数据。

DMA 方式下数据的传送细节请参考处理器手册中 DMA 章节。

音频芯片 UDA1341TS 说明

电路中使用的音频芯片是 PHILIPS 的 UDA1341TS 音频数字信号编译码器，UDA1341TS 可将立体声模拟信号转化为数字信号，同样也能把数字信号转换成模拟信号，并可用 PGA（可编程增益控制），AGC（自动增益控制）对模拟信号进行处理；对于数字信号，该芯片提供了 DSP（数字音频处理）功能。实际使用中，UDA1341TS 广泛应用于 MD、CD、notebook、PC 和数码摄像机等。

UDA1341TS 提供两组音频输入信号线、一组音频信号输出线，一组 IIS 总线接口信号，一组 L3 总线。

IIS 总线接口信号线包括位时钟输入 BCK、字选择输入 WS、数据输入 DATAI、数据输出 DATAO 和音频系统时钟 SYSCLK 信号线。

UDA1341TS 的 L3 总线，包括微处理器接口数据 L3DATA、微处理器接口模式 L3MODE、微处理器接口时钟 L3CLOCK 三根信号线，当该芯片工作于微控制器输入模式使用的，微处理器通过 L3 总线对 UDA1341TS 中的数字音频处理参数和系统控制参数进行配置。处理器 S3C2410X 中没有 L3 总线专用接口，电路中使用 I/O 口连接 L3 总线。L3 总线的接口时序和控制方式参见 UDA1341TS 手册。

电路连接

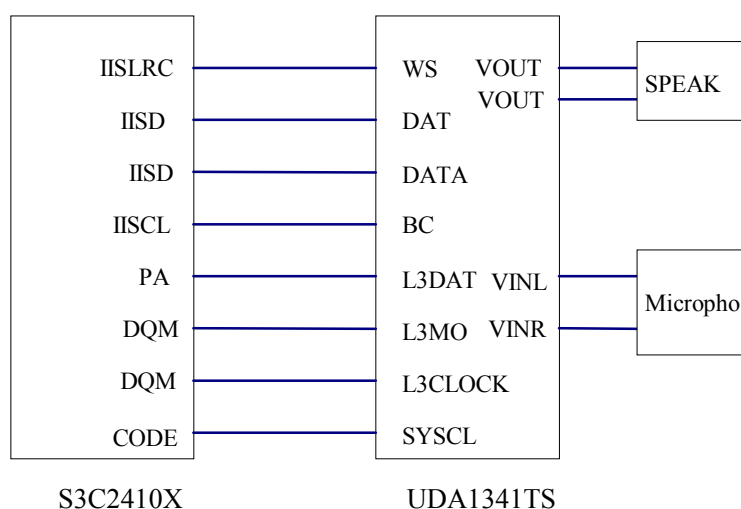


图 6-3-3 IIS 接口电路

6.3.5 实验步骤

1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接到目标板上，使用 Embest EduKit-III 实验板附带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 ULINK USB-JTAG 仿真器连接到目标板上，打开实验例程目录 6.3_iis_test 子目录下的 iis_test.Uv2 例程，编译链接工程，直到链接工程成功；
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境（工程默认已经配置正确），点击工具栏“”，在 Option for Target 对话框的 Linker 页中选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug Session 项或点击工具栏“”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；

4) 如果需要将程序烧写固化到 Flash 中, 仅需要更改分散加载文件即可(慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序, 建议实验中不操作)。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件, 重新编译工程, 点击 MDK 的 Flash 菜单, 选择 Download 烧写调试代码到目标系统的 Nor Flash 中, 重启实验板, 实验板将会运行烧写到 Nor Flash 中的代码;

5) 点击 Debug 菜单的 Run 或 F5 键全速运行程序, 或者单步调试程序。

4. 观察实验结果

1). 在 PC 机上观察超级终端程序主窗口, 可以看到如下界面:

```
Boot success...

IIS test example

Menu(press digital to select):

1: play wave file

2: record and play
```

2) 选择程序操作方式, 选择 1 的话将会听到音乐, 选择 2 的话将会进行录音(本板配置是 microphone 插座)。

```
Start recording....

Press any key to end recording

End of record!!!

Press any key to play record data!!!
```

3) 按任意键进行录音回放。

5. 完成实验练习题

理解和掌握实验后, 完成实验练习题。

6.3.6 实验参考程序

1. 初始化程序

```

/*****
* name:      iis_init
* func:      Initialize IIS circuit
* para:      none
*****/

void iis_init(void)
{
    rGPBUP = rGPBUP & ~(0x7<<2) | (0x7<<2); //The pull up function is disabled GPB[4:2] 1 1100
    rGPBCON = rGPBCON & ~((1<<9)|(1<<7)|(1<<5)) | (1<<8)|(1<<6)|(1<<4);
    //GPB[4:2]=Output(L3CLOCK):Output(L3DATA):Output(L3MODE)
    rGPEUP = rGPEUP | 0x1f; //The pull up function is disabled GPE[4:0] 1 1111
    rGPECON = rGPECON & ~((1<<8)|(1<<6)|(1<<4)|(1<<2)|(1<<0)) | (1<<9)|(1<<7)|(1<<5)|(1<<3)|(1<<1);
    //GPE[4:0]=I2SSDO:I2SSDI:CDCLK:I2SSCLK:I2SLRCK

```

```

f_nDMADone = 0;
init_1341(PLAY);           // initialize philips UDA1341 chip
}

```

2. IIS 控制程序

```

/*****
* name:      iis_test
* func:      Test IIS circuit
* para:      none
* ret:       none
* modify:
* comment:
*****/

void iis_test(void)
{
    UINT8T    ucInput;
    int nSoundLen=155956;
    uart_printf("\n IIS test example\n");
    iis_init();           // initialize IIS
    uart_printf(" Menu(press digital to select):\n");
    uart_printf(" 1: play wave file \n");
    uart_printf(" 2: record and play\n");
    g_nKeyPress = 1;      // only for board test to select and exit
    while((ucInput != '1') & (ucInput != '2'))
    {
        ucInput = uart_getkey();
        if(g_nKeyPress!=1)           // SB1202/SB1203 to exit board test
        {
            ucInput='1';           // any select "Play wav"
            break;
        }
    };
    uart_printf(" %c\n",ucInput);
    if(ucInput == 0x31)
    {
#ifdef BOARDTEST
        memcpy((void *)0x30200000, g_ucWave, nSoundLen);
#endif
        iis_play_wave(1,(UINT8T *)0x30200000,nSoundLen);//      nSoundLen = 155956;
    }
    if(ucInput == 0x32)
        iis_record();
    uart_printf(" end.\n");
    iis_close();           // close IIS
}

/*****

```

```

* name:      iis_play_wave
* func:      play wave data
* para:      nTimes  --   input, play times
* ret:       none
* modify:
* comment:
*****/

void iis_play_wave(int nTimes,UINT8T *pWavFile, int nSoundLen)
{
    int i;
    ClearPending(BIT_DMA2);
    rINTMOD = 0x0;
    // initialize philips UDA1341 chip
    init_1341(PLAY);
    // set BDMA interrupt
    pISR_DMA2 = (unsigned)dma2_done;
    rINTMSK  &= ~(BIT_DMA2);
    for(i=nTimes; i!=0; i--)
    {
        // initialize variables
        f_nDMADone = 0;
        //DMA2 Initialize
        rDISRCC2 = (0<<1) + (0<<0);           //AHB, Increment
        rDISRC2  = ((INT32T)(pWavFile));
        rDIDSTC2 = (1<<1) + (1<<0);           //APB, Fixed
        rDIDST2  = ((INT32T)IISFIFO);         //IISFIFO
        rDCON2 = (1<<31)+(0<<30)+(1<<29)+(0<<28)+(0<<27)+(0<<24)+(1<<23)+(0<<22)+(1<<20)+nSoundLen/2;
        //Handshake, sync PCLK, TC int, single tx, single service, I2SSDO, I2S request,
        //Auto-reload, half-word, size/2
        rDMASKTRIG2 = (0<<2)+(1<<1)+0;         //No-stop, DMA2 channel on, No-sw trigger
        //IIS Initialize
        //Master,Tx,L-ch=low,iis,16bit ch.,CDCLK=384fs,IISCLK=32fs
        rIISCON = (1<<5)+(0<<4)+(0<<3)+(1<<2)+(1<<1);
        rIISMOD = (0<<8) + (2<<6) + (0<<5) + (0<<4) + (1<<3) + (1<<2) + (1<<0);
        rIISPSR = (2<<5) + 2;                 //Prescaler_A/B=3
        //Tx DMA enable,Rx DMA disable,Tx not idle,Rx idle,prescaler enable,stop
        rIISFCON = (1<<15) + (1<<13);         //Tx DMA,Tx FIFO --> start piling....
        rIISCON |= 0x1;                       // enable IIS
        while( f_nDMADone == 0);              // DMA end
        rINTMSK |= BIT_DMA2;
        rIISCON = 0x0;                       // IIS stop
    }
}

```


6.3.7 练习题

编写程序实现通过按钮调整音量的大小。

6.4 USB 接口实验

6.4.1 实验目的

了解 USB 接口基本原理。

掌握通过 USB 接口与 PC 通讯的编程技术。

6.4.2 实验设备

- 硬件: Embest EduKit-III 实验平台, ULINK USB-JTAG 仿真器套件, PC 机。
- 软件: μ Vision IDE for ARM 集成开发环境, Windows 98/2000/NT/XP。

6.4.3 实验内容

为 S3C2410x 处理器集成的 USB Device 单元编写设备控制器驱动程序;

实现 PC 机端 USB 主机与实验板 USB 设备进行数据接收与发送。

6.4.4 实验原理

1. USB 基础知识

(1) 定义

通用串行总线协议 USB (Universal Serial Bus) 是由 Intel、Compaq、Microsoft 等公司联合提出的一种新的串行总线标准, 主要用于 PC 机与外围设备的互联。1994 年 11 月发布第一个草案, 1996 年 2 月发布第一个规范版本 1.0, 2000 年 4 月发布高速模式版本 2.0, 对应的设备传输速度也从 1.5Mb/s 的低速和 12Mb/s 的全速提高到如今的 480Mb/s 的高速。其主要特点是:

支持即插即用: 允许外设主机和其它外设工作时进行连接配置使用及移除。

传输速度快: USB 支持三种设备传输速率: 低速设备 1.5 Mb/s、中速设备 12 Mb/s 和高速设备 480Mb/s。

连接方便: USB 可以通过串行连接或者使用集线器 Hub 连接 127 个 USB 设备, 从而以一个串行通道取代 PC 上其他 I/O 端口如串行口、并行口等, 使 PC 与外设之间的连接更容易。

独立供电: USB 接口提供了内置电源。

低成本: USB 使用一个 4 针插头作为标准插头, 通过这个标准插头, 采用菊花链形式可以把多达 127 个的 USB 外设连接起来, 所有的外设通过协议来共享 USB 的带宽。

(2) 组成

USB 规范中将 USB 分为五个部分: 控制器、控制器驱动程序、USB 芯片驱动程序、USB 设备以及针对不同 USB 设备的客户驱动程序。

控制器 (Host Controller), 主要负责执行由控制器驱动程序发出的命令, 如位于 PC 主板的 USB 控制芯片。

控制器驱动程序 (Host Controller Driver), 在控制器与 USB 设备之间建立通信信道, 一般由操作系统或控制器厂商提供。

USB 芯片驱动程序 (USB Driver), 提供对 USB 芯片的支持, 设备上的固件 (Firmware)。

USB 设备 (USB Device), 包括与 PC 相连的 USB 外围设备。

设备驱动程序 (Client Driver Software), 驱动 USB 设备的程序, 一般由 USB 设备制造商提供。

(3) 传输方式

针对设备对系统资源需求的不同，在 USB 规范中规定了四种不同的数据传输方式：

同步传输 (Isochronous)，该方式用来联接需要连续传输数据，且对数据的正确性要求不高而对时间极为敏感的外部设备，如麦克风、喇叭以及电话等。同步传输方式以固定的传输速率，连续不断地在主机与 USB 设备之间传输数据，在传送数据发生错误时，USB 并不处理这些错误，而是继续传送新的数据。同步传输方式的发送方和接收方都必须保证传输速率的匹配，不然会造成数据的丢失。

中断传输 (Interrupt)，该方式用来传送数据量较小，但需要及时处理，以达到实时效果的设备，此方式主要用在偶然需要少量数据通信，但服务时间受限制的键盘、鼠标以及操纵杆等设备上。

控制传输 (Control)，该方式用来处理主机到 USB 设备的数据传输，包括设备控制指令、设备状态查询及确认命令，当 USB 设备收到这些数据和命令后，将依据先进先出的原则处理到达的数据。主要用于主机把命令传给设备、及设备把状态返回给主机。任何一个 USB 设备都必须支持一个与控制类型相对应的端点 0。

批量传输 (Bulk)，该方式不能保证传输的速率，但可保证数据的可靠性，当出现错误时，会要求发送方重发。通常打印机、扫描仪和数字相机以这种方式与主机联接。

(4) 关键词

USB 主机(Host) USB 主机控制总线上所有的 USB 设备和所有集线器的数据通信过程，一个 USB 系统中只有一个 USB 主机，USB 主机检测 USB 设备的连接和断开、管理主机和设备之间的标准控制管道、管理主机和设备之间的数据流、收集设备的状态和统计总线的活动、控制和管理主机控制器与设备之间的电气接口，每一毫秒产生一帧数据，同时对总线上的错误进行管理和恢复。

USB 设备 (Device) 通过总线与 USB 主机相连的称为 USB 设备。USB 设备接收 USB 总线上的所有数据包，根据数据包的地址域来判断是否接收；接收后通过响应 USB 主机的数据包与 USB 主机进行数据传输。

端点 (Endpoint) 端点是位于 USB 设备中与 USB 主机进行通信的基本单元。每个设备允许有多个端点，主机只能通过端点与设备进行通讯，各个端点由设备地址和端点号确定在 USB 系统中唯一的地址。每个端点都包含一些属性：传输方式、总线访问频率、带宽、端点号、数据包的最大容量等。除控制端点 0 外的其他端点必须在设备配置后才能生效，控制端点 0 通常用于设备初始化参数。USB 芯片中，每个端点实际上就是一个一定大小的数据缓冲区。

管道 (Pipe) 管道是 USB 设备和 USB 主机之间数据通信的逻辑通道，一个 USB 管道对应一个设备端点，各端点通过自己的管道与主机通信。所有设备都支持对应端点 0 的控制管道，通过控制管道主机可以获取 USB 设备的信息，包括：设备类型、电源管理、配置、端点描述等。

2. USB 设备设计原理

USB 设备控制器提供多个标准的端点，每个端点都支持单一的总线传输方式。端点 0 支持控制传输，其他端点支持同步传输、批量传输或中断传输中的任意一种。管理和使用这些端点，实际上就是通过操作相应的控制寄存器、状态寄存器、中断寄存器和数据寄存器来实现。其中，控制寄存器用于设置端点的工作模式、启用端点的功能等；状态寄存器用于查询端点的当前状态；中断寄存器则用于设置端点的中断触发和响应功能；数据寄存器则是设备与主机交换数据用的缓冲区。

(1) 关于 S3C2410X 的 USB 设备控制器

Embest EduKit-III 实验平台的 S3C2410X 上集成了一个 USB 设备控制器。它具体如下特性：

- 完全兼容 USB 1.1 规范
- 全速 (12Mbps) USB 设备
- 集成 USB 收发器

- 支持控制，中断和批量传输
- 包含 5 个带有 FIFO 的端点(1 个带 16 字节 FIFO 的双向控制端点和 4 个带 64 字节 FIFO 的双向支持批量传输的端点)
- 带有支持批量传输的 DMA 接口
- 支持挂起和远程唤醒功能

USB 设备控制器的寄存器说明

所有的寄存器都是通过字节或字方式进行访问，在 little endian 和 big endian 方式下，访问的偏移地址会有不同。下表列出了 USB 设备控制器的寄存器（详细的介绍请参考 S3C2410X 的芯片手册）。

表 6-4-1 S3C2410X USB 设备控制器寄存器

寄存器名	描述	偏移地址
FUNX_ADDR_REG	功能地址寄存器	0x140(L)/0x143(B)
PWR_REG	电源管理寄存器	0x144(L)/0x147(B)
EP_INT_REG	端点中断寄存器	0x148(L)/0x14B(B)
USB_INT_REG	USB 中断寄存器	0x158(L)/0x15B(B)
EP_INT_EN_REG	端点中断使能寄存器	0x15C(L)/0x15F(B)
USB_INT_EN_REG	帧序号低字节寄存器	0x16C(L)/0x16F(B)
FRAME_NUM1_REG	帧序号高字节寄存器	0x170(L)/0x173(B)
FRAME_NUM2_REG	USB 中断使能寄存器	0x174(L)/0x177(B)
INDEX_REG	索引寄存器	0x178(L)/0x17B(B)
EP0_FIFO_REG	端点 0 的 FIFO 寄存器	0x1C0(L)/0x1C3(B)
EP1_FIFO_REG	端点 1FIFO 寄存器	0x1C4(L)/0x1C7(B)
EP2_FIFO_REG	端点 2FIFO 寄存器	0x1C8(L)/0x1CB(B)
EP3FIFO_REG	端点 3FIFO 寄存器	0x1CC(L)/0x1CF(B)
EP4FIFO_REG	端点 4FIFO 寄存器	0x1D0(L)/0x1D3(B)
EP1_DMA_CON	端点 1 DMA 控制寄存器	0x200(L)/0x203(B)
EP1_DMA_UNIT	端点 1 DMA 传输单元计数器	0x204(L)/0x207B)
EP1_DMA_FIFO	端点 1 DMA FIFO 计数器	0x208(L)/0x20B(B)
EP1_DMA_TTC_L	端点 1 DMA 传输单元计数器低字节	0x20C(L)/0x20F(B)
EP1_DMA_TTC_M	端点 1 DMA 传输单元计数器中字节	0x210(L)/0x213(B)
EP1_DMA_TTC_H	端点 1 DMA 传输单元计数器高字节	0x214(L)/0x217(B)
EP2DMA_CON	端点 2 DMA 控制寄存器	0x218(L)/0x21B(B)
EP2DMA_UNIT	端点 2 DMA 传输单元计数器	0x21C(L)/0x21F(B)
EP2DMA_FIFO	端点 2 DMA FIFO 计数器	0x220(L)/0x223(B)
EP2DMA_TTC_L	端点 2 DMA 传输单元计数器低字节	0x224(L)/0x227(B)
EP2DMA_TTC_M	端点 2 DMA 传输单元计数器中字节	0x228(L)/0x22B(B)
EP2DMA_TTC_H	端点 2DMA 传输单元计数器高字节	0x22C(L)/0x22F(B)
EP3DMA_CON	端点 3 DMA 控制寄存器	0x240(L)/0x243(B)
EP3DMA_UNIT	端点 3 DMA 传输单元计数器	0x244 (L)/0x247(B)
EP3DMA_FIFO	端点 3 DMA FIFO 计数器	0x248(L)/0x24B(B)
EP3DMA_TTC_L	端点 3 DMA 传输单元计数器低字节	0x24C(L)/0x24F(B)
EP3DMA_TTC_M	端点 3 DMA 传输单元计数器中字节	0x250(L)/0x253(B)
EP3DMA_TTC_H	端点 3 DMA 传输单元计数器高字节	0x254(L)/0x257(B)

EP4DMA_CON	端点 4 DMA 控制寄存器	0x258(L)/0x25B(B)
EP4DMA_UNIT	端点 4 DMA 传输单元计数器	0x25C(L)/0x25F(B)
EP4DMA_FIFO	端点 4 DMA FIFO 计数器	0x260(L)/0x263(B)
EP4DMA_TTC_L	端点 4 DMA 传输单元计数器低字节	0x264(L)/0x267(B)
EP4DMA_TTC_M	端点 4 DMA 传输单元计数器中字节	0x268(L)/0x26B(B)
EP4DMA_TTC_H	端点 4 DMA 传输单元计数器高字节	0x26C(L)/0x26F(B)
MAXP_REG	端点最大包寄存器	0x18C(L)/0x18F(B)
IN_CSR1_REG/EP0_CSR	输入端点控制状态寄存器 1/端点 0 控制状态寄存器	0x184(L)/0x187(B)
IN_CSR2_REG	输入端点控制状态寄存器 2	0x188(L)/0x18B(B)
OUT_CSR1_REG	输出端点控制状态寄存器 1	0x190(L)/0x193(B)
OUT_CSR2_REG	输出端点控制状态寄存器 2	0x194(L)/0x197(B)
OUT_FIFO_CNT1_REG	端点写输出字节数寄存器 1	0x198(L)/0x19B(B)
OUT_FIFO_CNT2_REG	端点写输出字节数寄存器 2	0x19C(L)/0x19F(B)

(2) 设备驱动程序设计

在所有的操作之前，必须对 S3C2410X 的杂项控制器进行如下设置：

```
rMISCCR=rMISCCR&~(1<<3);    //使用 USB 设备而不是 USB 主机功能
rMISCCR=rMISCCR&~(1<<13);    //使用 USB 端口 1 模式
```

初始化 USB

在使用 USB 之前必须要进行初始化。USB 主机和 USB 设备接口都需要 48Mhz 的时钟频率。在 S3C2410X 中，这个时钟是由 UPLL (USB 专用 PLL) 来提供的。USB 初始化的第一步就是要对 UPLL 控制器进行设置。

```
ChangeUPLLValue(40,1,2);    //UCLK=48Mhz
```

在 USB1.x 规范中，规定了 5 种标准的 USB 描述符：设备描述符 (Device Descriptor)、配置描述符 (Configuration Descriptor)、接口描述符 (Interface Descriptor)、端点描述符 (EndPoint Descriptor) 和字符串描述符 (String Descriptor)。每个 USB 设备只有一个设备描述符，而一个设备中可以包含一个或多个配置描述符，即 USB 设备可以有多种配置。设备的每一个配置中又可以包含一个或多个接口描述符，即 USB 设备可以支持多种功能 (接口)，接口的特性通过接口描述符提供。在 Embest EduKit-III 实验平台的 USB 设备中只有一种配置，支持一种功能。关于设备描述符表的初始化以及配置由下面的两个函数实现：

```
InitDescriptorTable();    //初始化描述符表
ConfigUsbd();    //设备的配置
```

在 ConfigUsbd 函数中，将 USB 设备控制器的端点 0 设置为控制端点，端点 1 设置为批量输入端点，端点 3 设置为批量输出端点，端点 2 和端点 4 暂时没有使用，同时，还使能了端口 0，1，3 的中断和 USB 的复位中断。

```
rEP_INT_EN_REG=EP0_INT|EP1_INT|EP3_INT;
rUSB_INT_EN_REG=RESET_INT;
```

除此之外，初始化过程还对中断服务程序入口等进行了设置。

USB 中断

S3C2410X 能够接收 56 个中断源请求，当它接收到来自 USB 设备的中断请求时就会将 SRCPND 寄存器的 INT_USBD 置位，经过仲裁之后，中断控制器向内核发送 IRQ 中断请求。

USB 中断服务例程

当内核接收 USB 设备的中断请求之后，就会转入相应的中断服务程序运行。这个中断服务程序入口是在 USB 初始化时设置的。

```
pISR_USBD = (unsigned)IsrUsbd;
```

USB 读写

USB 设备的读写是通过管道来完成的。管道是 USB 设备和 USB 主机之间数据通信的逻辑通道，它的物理介质就是 USB 系统中的数据线。在设备端，管道的主体是“端点”，每个端点占据各自的管道和 USB 主机通信。

所有的设备都需要有支持控制传输的端点，协议将端点 0 定义为设备默认的控制端点。在设备正常工作之前，USB 主机必须为设备分配总线上唯一的设备地址，并完成读取设备的各种描述符，根据描述符的需求为设备的端点配置管道，分配带宽等工作，另外，在设备的工作过程中，主机希望及时的获取设备的当前状态。以上的过程是通过端点 0 来完成的。

USB 设备和主机之间的数据的接收和发送采用的是批量传输方式。端点 1 为批量输入端点，端点 3 为批量输出端点（输入和输出以 USB 主机为参考）。端点 3 数据的批量传输由 DMA 接口实现。

本实验假设读者对 USB 通信有一定的认识，并由于篇幅有限，本文不能对 USB 规范协议作更详细的介绍，有兴趣的读者可以参考相关书籍。

6.4.5 实验操作步骤

1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接到目标板上，使用 Embest EduKit-III 实验板自带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 ULINK USB-JTAG 仿真器连接到目标板上，打开实验例程目录 6.4_usb_test 子目录下的 usb_test.Uv2 例程，编译链接工程，直到链接工程成功；
- 3) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；
- 4) 安装 USB 设备驱动程序。

运行 USB 例程后，Windows 弹出发现新硬件的提示对话框，按照安装向导安装驱动程序 Embest EduKit-III USB Driver，驱动程序安装文件位于 6.4_usb_test\Driver 目录。

注意：驱动安装一定要在本例程下载运行正确后进行；

如果运行 DNW 后，状态条为[USB:x]，请重新插入 USB 线。

- 5) 运行 USB 数据传送演示软件

运行位于 6.4_usb_test 目录下 DNW.exe，并进行适当的配置（注意 USB Port 的 Download Address 设置应该是在系统的 RAM 空间，而且不能和当前运行的程序空间重叠）：

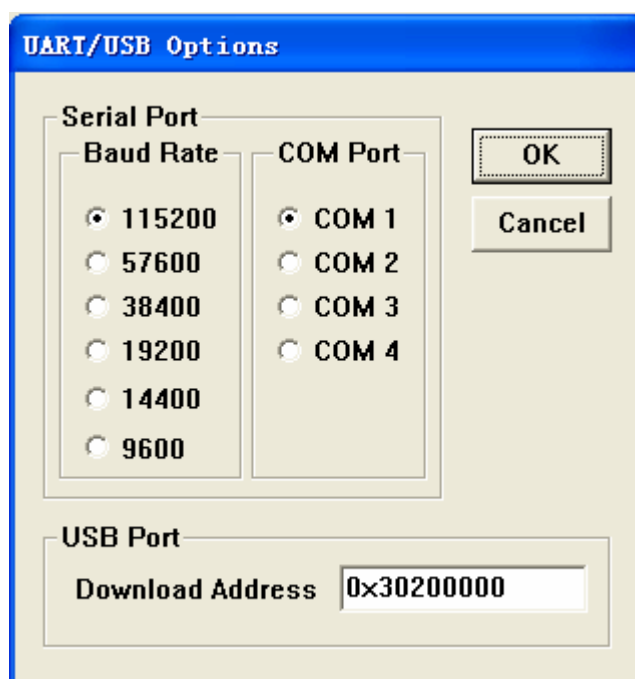




图 6-4-1 DNW.exe 配置

- 6) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境（工程默认已经配置正确），点击工具栏 “”，在 Option for Target 对话框的 Linker 页中选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug Session 项或点击工具栏 “”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；

7) 发送和接收数据

设置好 DNW.exe 之后，将 USB 连接线一端连接到 PC 机 USB 口，一端连接到实验平台的 USB 设备接口，并点击 DNW Serial->Connect 连接串口；

当程序正确运行后，DNW 的标题栏上会显示信息：[COM1, 115200bps] [USB: OK]，这表明串行口和 USB 连接成功。

在 Select Menu 中，选择[1]，给出要下载到的 SDRAM 地址，回车输入后可以看到提示信息：USB host is connected. Waiting a download.

选择 USB Port->Transmit，即可进行程序的下载。下图显示了下载完成后的结果；

选择 USB Port->Rx Test 接收来自 USB 设备端点 1 上发送过来的测试数据；

选择 USB Port->Status 观察 USB 的状态；

6.4.6 实验参考程序

1. USB 中断服务程序

```

/*****
* name:      IsrUsbd
* func:      USBD Interrupt handler
* para:      none

```

```

* ret:          none
* modify:
* comment:
*****/
void __irq IsrUsbd(void)
{
    UINT8T usbdIntpnd,epIntpnd;
    UINT8T saveIndexReg=rINDEX_REG;
    usbdIntpnd=rUSB_INT_REG;
    epIntpnd=rEP_INT_REG;
    if(usbdIntpnd&SUSPEND_INT)
    {
        rUSB_INT_REG=SUSPEND_INT;
        DbgPrintf( "<SUS]");
    }
    if(usbdIntpnd&RESUME_INT)
    {
        rUSB_INT_REG=RESUME_INT;
        DbgPrintf("<RSM]");
    }
    if(usbdIntpnd&RESET_INT)
    {
        DbgPrintf( "<RST]");
        ReconfigUsbd();
        rUSB_INT_REG=RESET_INT;
        PrepareEp1Fifo();
    }
    if(epIntpnd&EP0_INT)
    {
        rEP_INT_REG=EP0_INT;
        Ep0Handler();
    }
    if(epIntpnd&EP1_INT)
    {
        rEP_INT_REG=EP1_INT;
        Ep1Handler();
    }
    if(epIntpnd&EP2_INT)
    {
        rEP_INT_REG=EP2_INT;
        DbgPrintf("<2:TBD]");
    }
    if(epIntpnd&EP3_INT)
    {
        rEP_INT_REG=EP3_INT;
        Ep3Handler();
    }
}

```

```

    }
    if(epIntpnd&EP4_INT)
    {
        rEP_INT_REG=EP4_INT;
        DbgPrintf("<4:TBD]");
    }
    ClearPending(BIT_USBD);
    rINDEX_REG=saveIndexReg;
}

```

2. 端点 0 的数据传输程序

```

/*****
* name:      Ep0Handler
* func:      EP0 Interrupt handler
* para:      none
* ret:       none
* modify:
* comment:
*****/

void Ep0Handler(void)
{
    static int ep0SubState;
    int i;
    UINT8T ep0_csr;
    rINDEX_REG=0;
    ep0_csr=rEP0_CSR;
    DbgPrintf("<0:%x]", ep0_csr);
    if(ep0_csr & EP0_SETUP_END)
    {
        DbgPrintf("[SETUPEND]");
        CLR_EP0_SETUP_END();
        if(ep0_csr & EP0_OUT_PKT_READY)
        {
            FLUSH_EP0_FIFO();
            CLR_EP0_OUT_PKT_RDY();
        }
        ep0State=EP0_STATE_INIT;
        return;
    }
    if(ep0_csr & EP0_SENT_STALL)
    {
        DbgPrintf("[STALL]");
        CLR_EP0_SENT_STALL();
        if(ep0_csr & EP0_OUT_PKT_READY)
        {
            CLR_EP0_OUT_PKT_RDY();
        }
    }
}

```



```

    ep0State=EP0_STATE_INIT;
    return;
}
if((ep0_csr & EP0_OUT_PKT_READY) && (ep0State==EP0_STATE_INIT))
{
    RdPktEp0((UINT8T *)&descSetup,EP0_PKT_SIZE);
    PrintEp0Pkt((UINT8T *)&descSetup); //DEBUG
    switch(descSetup.bRequest)
    {
        case GET_DESCRIPTOR:
            switch(descSetup.bValueH)
            {
                case DEVICE_TYPE:
                    DbgPrintf("[GDD]");
                    CLR_EP0_OUT_PKT_RDY();
                    ep0State=EP0_STATE_GD_DEV_0;
                    break;
                case CONFIGURATION_TYPE:
                    DbgPrintf("[GDC]");
                    CLR_EP0_OUT_PKT_RDY();
                    if((descSetup.bLengthL+(descSetup.bLengthH<<8))>0x9)
                        ep0State=EP0_STATE_GD_CFG_0;
                    else
                        ep0State=EP0_STATE_GD_CFG_ONLY_0;
                    break;
                case STRING_TYPE:
                    DbgPrintf("[GDS]");
                    CLR_EP0_OUT_PKT_RDY();
                    switch(descSetup.bValueL)
                    {
                        case 0:
                            ep0State=EP0_STATE_GD_STR_I0;
                            break;
                        case 1:
                            ep0State=EP0_STATE_GD_STR_I1;
                            break;
                        case 2:
                            ep0State=EP0_STATE_GD_STR_I2;
                            break;
                        default:
                            DbgPrintf("[UE:STRI?]");
                            break;
                    }
                    ep0SubState=0;
                    break;
                case INTERFACE_TYPE:

```

```

        DbgPrintf("[GDI]");
        CLR_EP0_OUT_PKT_RDY();
        ep0State=EP0_STATE_GD_IF_ONLY_0;
        break;
    case ENDPOINT_TYPE:
        DbgPrintf("[GDE]");
        CLR_EP0_OUT_PKT_RDY();
        switch(descSetup.bValueL&0xf)
        {
            case 0:
                ep0State=EP0_STATE_GD_EP0_ONLY_0;
                break;
            case 1:
                ep0State=EP0_STATE_GD_EP1_ONLY_0;
                break;
            default:
                DbgPrintf("[UE:GDE?]");
                break;
        }
        break;
    default:
        DbgPrintf("[UE:GD?]");
        break;
}
break;
case SET_ADDRESS:
    DbgPrintf("[SA:%d]",descSetup.bValueL);
    rFUNC_ADDR_REG=descSetup.bValueL | 0x80;
    CLR_EP0_OUTPKTRDY_DATAEND();
    ep0State=EP0_STATE_INIT;
    break;
case SET_CONFIGURATION:
    DbgPrintf("[SC]");
    CLR_EP0_OUTPKTRDY_DATAEND();
    ep0State=EP0_STATE_INIT;
    isUsbdSetConfiguration=1;
    break;
default:
    DbgPrintf("[UE:SETUP=%x]",descSetup.bRequest);
    CLR_EP0_OUTPKTRDY_DATAEND();
    ep0State=EP0_STATE_INIT;
    break;
}
}
switch(ep0State)
{

```

```

case EP0_STATE_INIT:
    break;
case EP0_STATE_GD_DEV_0:      // GET_DESCRIPTOR:DEVICE
    DbgPrintf("[GDD0]");
    WrPktEp0((UINT8T *)&descDev+0,8); //EP0_PKT_SIZE
    SET_EP0_IN_PKT_RDY();
    ep0State=EP0_STATE_GD_DEV_1;
    break;
case EP0_STATE_GD_DEV_1:
    DbgPrintf("[GDD1]");
    WrPktEp0((UINT8T *)&descDev+0x8,8);
    SET_EP0_IN_PKT_RDY();
    ep0State=EP0_STATE_GD_DEV_2;
    break;
case EP0_STATE_GD_DEV_2:
    DbgPrintf("[GDD2]");
    rPktEp0((UINT8T *)&descDev+0x10,2);    //8+8+2=0x12
    SET_EP0_INPKTRDY_DATAEND();
    ep0State=EP0_STATE_INIT;
    break;
case EP0_STATE_GD_CFG_0:
    DbgPrintf("[GDC0]");
    WrPktEp0((UINT8T *)&descConf+0,8); //EP0_PKT_SIZE
    SET_EP0_IN_PKT_RDY();
    ep0State=EP0_STATE_GD_CFG_1;
    break;
case EP0_STATE_GD_CFG_1:
    DbgPrintf("[GDC1]");
    WrPktEp0((UINT8T *)&descConf+8,1);
    WrPktEp0((UINT8T *)&descClf+0,7);
    SET_EP0_IN_PKT_RDY();
    ep0State=EP0_STATE_GD_CFG_2;
    break;
case EP0_STATE_GD_CFG_2:
    DbgPrintf("[GDC2]");
    WrPktEp0((UINT8T *)&descClf+7,2);
    WrPktEp0((UINT8T *)&descEndpt0+0,6);
    SET_EP0_IN_PKT_RDY();
    ep0State=EP0_STATE_GD_CFG_3;
    break;
case EP0_STATE_GD_CFG_3:
    DbgPrintf("[GDC3]");
    WrPktEp0((UINT8T *)&descEndpt0+6,1);
    WrPktEp0((UINT8T *)&descEndpt1+0,7);
    SET_EP0_IN_PKT_RDY();
    ep0State=EP0_STATE_GD_CFG_4;

```

```

        break;
case EP0_STATE_GD_CFG_4:
    DbgPrintf("[GDC4]");
    //zero length data packit
    SET_EP0_INPKTRDY_DATAEND();
    ep0State=EP0_STATE_INIT;
    break;
case EP0_STATE_GD_CFG_ONLY_0:
    DbgPrintf("[GDC0]");
    WrPktEp0((UINT8T *)&descConf+0,8); //EP0_PKT_SIZE
    SET_EP0_IN_PKT_RDY();
    ep0State=EP0_STATE_GD_CFG_ONLY_1;
    break;
case EP0_STATE_GD_CFG_ONLY_1:
    DbgPrintf("[GDC01]");
    WrPktEp0((UINT8T *)&descConf+8,1);
    SET_EP0_INPKTRDY_DATAEND();
    ep0State=EP0_STATE_INIT;
    break;
case EP0_STATE_GD_IF_ONLY_0:
    DbgPrintf("[GDI0]");
    WrPktEp0((UINT8T *)&descIf+0,8);
    SET_EP0_IN_PKT_RDY();
    ep0State=EP0_STATE_GD_IF_ONLY_1;
    break;
case EP0_STATE_GD_IF_ONLY_1:
    DbgPrintf("[GDI1]");
    WrPktEp0((UINT8T *)&descIf+8,1);
    SET_EP0_INPKTRDY_DATAEND();
    ep0State=EP0_STATE_INIT;
    break;
case EP0_STATE_GD_EP0_ONLY_0:
    DbgPrintf("[GDE00]");
    WrPktEp0((UINT8T *)&descEndpt0+0,7);
    SET_EP0_INPKTRDY_DATAEND();
    ep0State=EP0_STATE_INIT;
    break;
case EP0_STATE_GD_EP1_ONLY_0:
    DbgPrintf("[GDE10]");
    WrPktEp0((UINT8T *)&descEndpt1+0,7);
    SET_EP0_INPKTRDY_DATAEND();
    ep0State=EP0_STATE_INIT;
    break;
case EP0_STATE_GD_STR_I0:
    DbgPrintf("[GDS0_0]");
    WrPktEp0((UINT8T *)descStr0, 4 );

```

```

        SET_EP0_INPKTRDY_DATAEND();
        ep0State=EP0_STATE_INIT;
        ep0SubState=0;
        break;
case EP0_STATE_GD_STR_I1:
    DbgPrintf("[GDS1_%d]",ep0SubState);
    if( (ep0SubState*EP0_PKT_SIZE+EP0_PKT_SIZE)<sizeof(descStr1) )
    {
        WrPktEp0((UINT8T *)descStr1+(ep0SubState*EP0_PKT_SIZE),
            EP0_PKT_SIZE);
        SET_EP0_IN_PKT_RDY();
        ep0State=EP0_STATE_GD_STR_I1;
        ep0SubState++;
    }
    else
    {
        WrPktEp0((UINT8T *)descStr1+(ep0SubState*EP0_PKT_SIZE),
            sizeof(descStr1)-(ep0SubState*EP0_PKT_SIZE));
        SET_EP0_INPKTRDY_DATAEND();
        ep0State=EP0_STATE_INIT;
        ep0SubState=0;
    }
    break;
case EP0_STATE_GD_STR_I2:
    DbgPrintf("[GDS2_%d]",ep0SubState);
    if( (ep0SubState*EP0_PKT_SIZE+EP0_PKT_SIZE)<sizeof(descStr2) )
    {
        WrPktEp0((UINT8T *)descStr2+(ep0SubState*EP0_PKT_SIZE),
            EP0_PKT_SIZE);
        SET_EP0_IN_PKT_RDY();
        ep0State=EP0_STATE_GD_STR_I2;
        ep0SubState++;
    }
    else
    {
        DbgPrintf("[E]");
        WrPktEp0((UINT8T *)descStr2+(ep0SubState*EP0_PKT_SIZE),
            sizeof(descStr2)-(ep0SubState*EP0_PKT_SIZE));
        SET_EP0_INPKTRDY_DATAEND();
        ep0State=EP0_STATE_INIT;
        ep0SubState=0;
    }
    break;
default:
    DbgPrintf("UE:G?D");
    break;

```

```

    }
}

```

3. 端点 1 的数据传输程序

```

/*-----*/
/*                      global variables                      */
/*-----*/

UINT8T ep1Buf[EP1_PKT_SIZE];
int transferIndex=0;

/*****
* name:      PrepareEp1Fifo
* func:      PrepareEp1Fifo
* para:      none
* ret:       none
* modify:
* comment:
*****/

void PrepareEp1Fifo(void)
{
    int i;
    UINT8T in_csr1;
    rINDEX_REG=1;
    in_csr1=rIN_CSR1_REG;
    for(i=0;i<EP1_PKT_SIZE;i++) ep1Buf[i]=(UINT8T)(transferIndex+i);
    WrPktEp1(ep1Buf,EP1_PKT_SIZE);
    SET_EP1_IN_PKT_READY();
}

/*****
* name:      Ep1Handler
* func:      Ep1Handler
* para:      none
* ret:       none
* modify:
* comment:
*****/

void Ep1Handler(void)
{
    UINT8T in_csr1;
    int i;
    rINDEX_REG=1;
    in_csr1=rIN_CSR1_REG;
    uart_printf(" <1:%x]",in_csr1);
    if(in_csr1 & EPI_SENT_STALL)
    {
        uart_printf(" [STALL]");
    }
}

```

```

        CLR_EP1_SENT_STALL();
        return;
    }
    PrintEpiPkt(ep1Buf,EP1_PKT_SIZE);
    transferIndex++;
    PrepareEp1Fifo();
    return;
}

```

6.4.7 实验练习题

编写程序，通过端点 1 发送自定义的信息到 PC 机上 USB 主机，并使用 DNW 观察。

6.5 SPI 接口通讯实验

6.5.1 实验目的

- 了解 S3C2410X 处理器 SPI 相关控制寄存器的使用。
- 熟悉 ARM 处理器系统硬件电路中 SPI 接口的设计方法。
- 掌握 ARM 处理器 SPI 接口通讯的软件编程方法。

6.5.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：µVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

6.5.3 实验内容

掌握 SPI 相关控制寄存器的使用方法后，编写程序实现 EduKit-III 系统监视 SPI 口的状态以及把指定的数据从 SPI0 口发送，SPI1 口把接收到的数据通过串口显示。

6.5.4 实验原理

1. SPI 接口概述

S3C2410X 包含有两个串行外围设备接口（SPI 口），每个 SPI 口都有两个分别用于发送和接收的 8bit 移位寄存器，在一次 SPI 通讯当中数据被同步的发送（串行移出）和接收（串行移入）。8 位串行数据的速率由相关的控制寄存器的内容决定。如果只想发送，接收到的是一些虚拟的数据。另外，如果只想接收，发送的数据也可以是一些虚拟的“1”。

SPI 接口特性

- 与 SPI 接口协议 v2.11 兼容
- 8 位用于发送的移位寄存器
- 8 位用于接收的移位寄存器
- 8 位预分频逻辑
- 具有查询，中断和 DMA 三种传送模式

SPI 接口操作

通过 SPI 接口，S3C2410X 可以与外设同时发送/接收 8 位数据。串行时钟线同步两条数据线的数据传输，用于移位和数据采样。如果 SPI 是主设备，数据传输速率由 SPPREN 寄存器的相关位控制。用户可以修改频率来调整波特率寄存器的值。如果 SPI 是从设备，其他的主设备提供时钟，向 SPDATn 寄存器中写入字节数据，SPI 发送/接收操作就同时启动。在某些情况下，从器件使能信号 nSS 要在向 SPDATn 寄存器中写入字节数据之前激活。

编程步骤

如果 ENSCK 和 SPCONn 中的 MSTR 位都被置位，向 SPDATn 寄存器写一个字节数据，就启动一次发送。也可以使用如下的典型的编程步骤来操作 SPI 口：

设置波特率预分频寄存器（SPPREn）；

设置 SPCONn 配置 SPI 模块；

向 SPDATn 中写十次 0xFF 来初始化 MMC 或 SD 卡。

把一个 GPIO（当作 nSS）清零来激活 MMC 或 SD 卡。

发送数据—>核查发送准备好标志（REDY=1），之后向 SPDATn 中写数据。

接收数据（1）：禁止 SPCONn 的 TAGD 位，正常模式—>向 SPDAT 中写 0xFF，确定 REDY 被置位后，从读缓冲区中读出数据。

接收数据（2）：使能 SPCONn 的 TAGD 位，自动发送虚拟数据模式—>确定 REDY 被置位后，从读缓冲区中读出数据。（之后自动开始数据传输）

置位 GPIO 引脚（当作 nSS 的那个引脚），停止 MMC 或 SD 卡。

SPI 口的传输格式

S3C2410X 支持 4 种不同的数据传输格式，图 6-5-1 为具体的时序图。

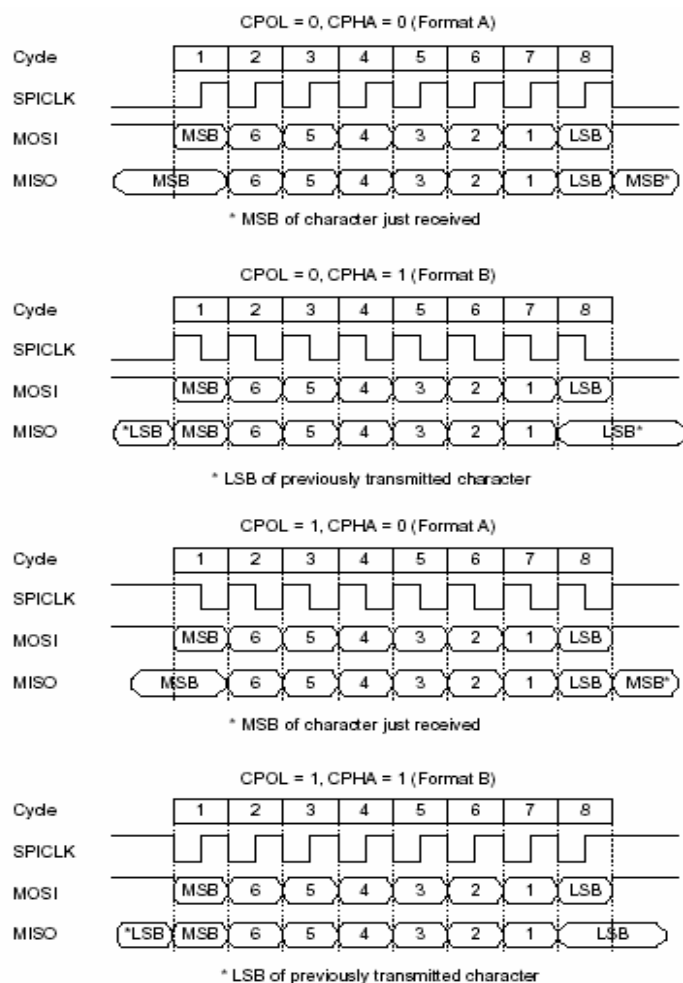


图 6-5-1 SPI 接口时序图

SPI 从设备 format B 接收数据模式

如果 SPI 从设备接收模式被激活，并且 SPI 格式被选为 B，SPI 操作将会失败：

内部的 READY 信号在 SPI_CNT 变为 0 前变成高电平。因此，DMA 模式 DATA_READ 信号在最后一位被锁存之前置位。

三种 SPI 通信模式

DMA 模式：该模式不能用于从设备 format B 形式。

查询模式：如果接收从设备采用 format B 形式，DATA_READ 信号应该比 SPICLK 延迟一个相位。

中断模式：如果接收从设备采用 format B 形式，DATA_READ 信号应该比 SPICLK 延迟一个相位。

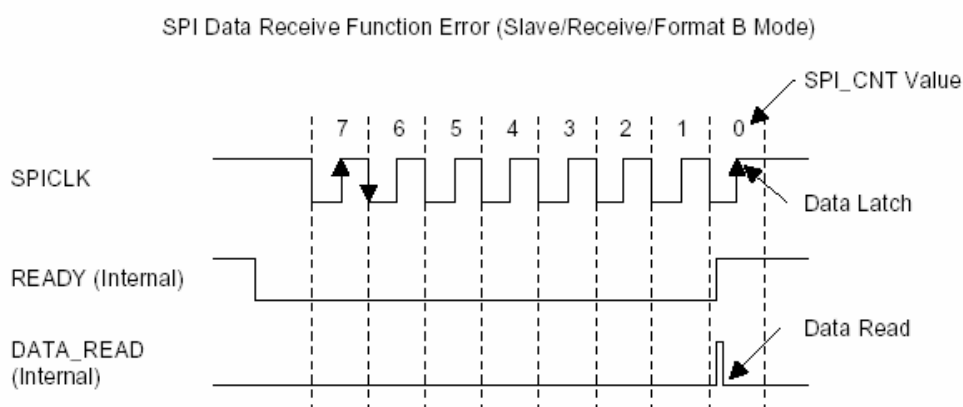


图 6-5-2 SPI 从设备接收数据模式

2. SPI 接口特殊寄存器

SPI 控制寄存器 (SPICONn)

寄存器	地址	R/W	功能描述	复位值
SPCON0	0x59000000	R/W	SPI0 控制寄存器	0x00
SPCON1	0x59000020	R/W	SPI1 控制寄存器	0x00

该寄存器控制 SPI 的工作模式

SPCONn[6: 5]: SPTDAT 的读写模式

00: 查询模式

01: 中断模式

10: DMA 模式

11: 保留

SPCONn[4]: SCK 允许/禁止位

0: 禁止 SCK

1: 允许 SCK

SPCONn[3]: 主/从选择位

0: 从设备

- 1: 主设备
- SPCONn[2]: 时钟极性选择位
- 0: 时钟高电平起作用
- 1: 时钟低电平起作用
- SPCONn[1]: 时钟相位选择位
- 0: format A
- 1: format B
- SPCONn[0]: 自动发送虚拟数据允许选择位
- 0: 正常模式
- 1: 自动发送虚拟数据模式

SPI 状态寄存器(SPSTAn)

寄存器	地址	R/W	功能描述	复位值
SPSTA0	0x59000004	R	SPI0 状态寄存器	0x01
SPSTA1	0x59000024	R	SPI1 状态寄存器	0x01

- SPSTAn[7: 3]: 保留
- SPSTAn[2]: 数据碰撞错误标志
- 0: 未检测到碰撞
- 1: 检测到碰撞错误
- SPSTAn[1]: 多主设备错误标志
- 0: 未检测到该错误
- 1: 发现多主设备错误
- SPSTAn[0]: 数据传输完成标志
- 0: 未完成
- 1: 完成数据传输

SPI 引脚控制寄存器 (SPPINn)

当一个 SPI 系统被允许，nSS 之外的引脚的数据传输方向都由 SPCONn 的 MSTR 位控制，nSS 引脚总是输入。

当 SPI 是一个主设备，nSS 引脚用于检测多主设备错误（如果 SPPIN 的 ENMUL 位被使能），另外还需要一个 GPIO 来选择从设备。如果 SPI 被配置为从设备，nSS 引脚用来被选择为从设备。

寄存器	地址	R/W	功能描述	复位值
SPPIN0	0x59000008	R/W	SPI0 控制寄存器	0x02
SPPIN1	0x59000028	R/W	SPI1 控制寄存器	0x02

- SPPINn[7: 3]: 保留

- SPPINn[2]:** 多主设备错误检测使能 (ENMUL)
 0: 禁止该功能
 1: 允许该功能
- SPPIN[1]:** 保留, 总为 1
- SPPIN[0]:** 主设备发送完一个字节后继续驱动还是释放
 0: 释放
 1: 继续驱动

SPIMISO 和 **SPIMOSI** 数据引脚用于发送或者接收串行数据, 如果 **SPI** 口被配置为主设备, **SPIMISO** 就是主设备的数据输入线, **SPIMOSI** 就是主设备的数据输出线, **SPICLK** 是时钟输出线, 如果 **SPI** 口被配置为从设备, 这些引脚的功能就正好相反。在一个多主设备的系统中, **SPICLK**, **SPIMOSI**, **SPIMISO** 都是一组一组单独配置的。

SPI 波特率预分频寄存器 (SPIPREn)

寄存器	地址	R/W	功能描述	复位值
SPPRE0	0x5900000C	R/W	SPI0 波特率预分频因子寄存器	0x00
SPPRE1	0x5900002C	R/W	SPI1 波特率预分频因子寄存器	0x00

$SPPREn[7:0] = \text{Prescaler value}$ 可以通过 Prescaler value 计算波特率, 公式如下:

$$\text{Baud Rate} = \text{PCLK} / 2 / (\text{Prescaler value} + 1)$$

SPI 发送数据寄存器 (SPTDATn)

发送数据寄存器中存放 **SPI** 口待发送的数据。

寄存器	地址	R/W	功能描述	复位值
SPTDAT0	0x59000010	R/W	SPI0 发送数据寄存器	0x00
SPTDAT1	0x59000030	R/W	SPI1 发送数据寄存器	0x00

SPI 接收数据寄存器 (SPRDATn)

接收数据寄存器中存放 **SPI** 口接收到的数据

寄存器	地址	R/W	功能描述	复位值
SPRDAT0	0x59000014	R	SPI0 接收数据寄存器	0x00
SPRDAT1	0x59000034	R	SPI1 接收数据寄存器	0x00

3. SPI 口接口电路

Embest EduKit-III 实验平台没有单独引出 **SPI** 接口, 而是通过 **PCI** 插槽引出相应管脚。电路如图 6-5-2 所示 (即 **PCI** 接口电路)

本实验用 S3C2410X 处理器的 SPI0 口发送数据，SPI1 口接收数据，实验时需要把相应信号接到一起(注：因为连接比较复杂此实验可以做为选做实验)：

SPICLK0(B44) \leftrightarrow SPICLK1(A46) nSS0(B47) \leftrightarrow nSS1 (B46)

1. 准备实验环境

2. 串口接收设置

3. 打开实验例程

- #### 4. 观察实验结果

- 256 -

The words that SPI0 transmit are:

S3C2410SPI COMMUNICATION TEST!

.....Tx OK

.....Rx OK

The words that SPI1 receive are:

S3C2410SPI COMMUNICATION TEST!

5. 完成实验练习题

理解和掌握实验后，完成实验练习题。

6.5.6 实验参考程序

1. SPI 测试程序

```

/*****
* name:      spi_test
* func:      spi_test function
* para:      none
* ret:       none
* modify:
* comment:
*****/

void spi_test(void)
{
    // int i;
    uart_printf(" The words that SPI0 transmit are:\n");
    uart_printf(" %s\n",cTxData);
    pISR_SPI0=(unsigned)spi0_int;
    pISR_SPI1=(unsigned)spi1_int;
    //clear interrupt pending
    ClearPending(BIT_SPI0);
    ClearPending(BIT_SPI1);
    //IO port configuration
    rGPECON=(2<<26)|(2<<24)|(2<<22);           // SPI1 configued slave
    rGPGCON=(3<<14)|(3<<12)|(3<<10)|(3<<6)|(1<<4); // nSS0 bit is output
    rGPGUP &=0xFF13;
    rGPEUP &=0xC7FF;
    rSPPRE0=PCLK/2/ucSpiBaud-1;
    // interrupt mode,enable ENSCK,master,CPOL=0,CPHA=0,normal mode
    rSPCON0=(1<<5)|(1<<4)|(1<<3)|(0<<2)|(0<<1)|(0<<0);
    rSPPRE1=PCLK/2/ucSpiBaud-1;
    // interrupt mode,enable ENSCK,slave,CPOL=0,CPHA=0,normal mode
    rSPCON1=(1<<5)|(1<<4)|(0<<3)|(0<<2)|(0<<1)|(0<<0);
    rSPPIN0=(0<<2)|(1<<1)|(0<<0);

```

```

//dis-ENMUL,SBO,release
rGPGDAT &=0xFFFB;           // nSS0=0

rINTMOD &= ~(BIT_SPI0 | BIT_SPI1);
rINTMSK &= ~(BIT_SPI0 | BIT_SPI1);

while(!(rSPSTA0 & rSPSTA1 & 0x1));
cTxEnd=0;
rSPTDAT0=*cTxData++;

rINTMSK |= (BIT_SPI0 | BIT_SPI1);
uart_printf(" end.\n");
}

```

2. SPI0 中断程序

```

/*****
* name:      spi0_int
* func:      spi0 interrupt service routine
* para:      none
* ret:       none
* modify:
* comment:
*****/

void __irq spi0_int(void)
{
    ClearPending(BIT_SPI0);
    while(!(rSPSTA0&0x01));
    if((*cTxData)!='\0')
        rSPTDAT0=*cTxData++;
    else if(cTxEnd==0)
    {
        cTxEnd=1;
        rSPTDAT0='\0';
        uart_printf("\n .....Tx OK\n");
    }
    if(cTxEnd==1)
        rINTMSK |= BIT_SPI0;
}

```

3. SPI1 中断程序

```

/*****
* name:      spi1_int
* func:      spi1 interrupt service routine

```

```

* para:      none
* ret:       none
* modify:
* comment:
*****/

void __irq spi1_int(void)
{
    ClearPending(BIT_SPI1);
    while(!(rSPSTA1&0x1));
    cRxData[cRxNo++]=rSPRDAT1;

    if((cRxNo>0)&&(cRxData[cRxNo-1]=='\0'))
    {
        rINTMSK |= BIT_SPI1;
        uart_printf(" .....Rx OK\n\n");
        uart_printf(" The words that SPI1 receive are:\n");
        uart_printf(" %s",cRxData);
    }
}

```

6.5.7 练习题

编写程序实现 SPI1 发送数据，SPI0 接收数据。

6.6 红外模块控制实验

6.6.1 实验目的

通过实验掌握 ARM 处理器的 UART 控制方式和工作原理。

了解 IrDA 模块的工作原理。

6.6.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

6.6.3 实验内容

编写程序，实现：

把红外模块 ZHX1010 收到的数据发送给 S3C2410X 的 UART1，保存并通过超级终端的主窗口显示；

把内存中的数据通过 S3C2410X 的 UART1 发送给红外模块 ZHX1010，并由红外模块发射出去。

6.6.4 实验原理

1. 红外数据传输概述

红外数据传输，使用传播介质——红外线。红外线是波长在 750nm~1mm 之间的电磁波，是人眼看不到的光线。红外数据传输一般采用红外波段内的近红外线，波长在 0.75μm~25μm 之间。红外数据协会成立后，为保证不同厂商的红外产品能获得最佳的通信效果，限定所用红外波长在 850nm~900nm。

IrDA 是国际红外数据协会的英文缩写，制定的协议有：IrDA1.0 协议基于异步收发器 UART，最高通信速率在 115.2kbps，简称 SIR(Serial Infrared，串行红外协议)；IrDA1.1 协议提高通信速率到 4Mbps，简称 FIR(Fast Infrared，快速红外协议)，同时在低速时保留 1.0 协议规定；之后，IrDA 又推出了最高通信速率在 16Mbps 的协议，简称 VFIR(Very Fast Infrared，特速红外协议)。

红外传输距离在几 cm 到几十 m，发射角度通常在 0~15°，发射强度与接收灵敏度因不同器件不同应用设计而强弱不一。使用时只能以半双工方式进行红外通信。

红外收发器件集发射与接收于一体。通常，器件的发射部分含有驱动器，接收部分含有放大器，并且内部集成有关断控制逻辑。关断控制逻辑在发送时关断接收，以避免引入干扰；不使用红外传输时，该控制逻辑通过 SD 引脚接受指令，关断器件电源供应，以降耗节能。使用器件时需要在 LED 引脚接入适当的限流电阻。大多数红外收发器件带有屏蔽层。该层不要直接接地，可以通过串联一磁珠再接地，以引入干扰影响接收灵敏度。

红外检测器件的主要部件是红外敏感接收管，有独立接收管构成器件的，有内含放大器的，有集成放大器与解调器的。接收灵敏度是衡量红检测器件的主要性能指标，接收灵敏度越高，传输距离越远，误码率越低。

2. 红外收发芯片 THX1010 概述

本实验采用的芯片是 Zilog 的 THX1010，THX1010 SIR 收发器适用于便携式低功耗产品，比如手机、数码相机、便携式打印机、笔记本电脑或者 PDA 等。THX1010 内部集成了红外发射二极管、一个红外检测二极管、一个数字 AC 耦合驱动、一个接收器解码电路。外围电路及其简单，只需要两个外部电阻和一个外部电容即可工作。

特性：

- 符合 IrDA SIR 标准 (2-4-115.2kbps，至少 1m 通信距离)；
- 宽电源电压范围：2.4V—5.5V；
- 低功耗，3V 供电时电流 90uA；
- 外观小巧：9.9mm 长×3.7mm 宽×4.0mm 高；
- 外围电路简单：两个电阻，一个电容；
- 扩展温度范围：-30~85 度；
- 符合 IEC-825-1 1 类眼保标准；

引脚说明：

引 脚	名 称	描 述	I/O
1	LEDA	IRED 正极	—
2	TXD	发送输入	I
3	RXD	接收输出	O
4	SD	调电模式选通，低电平选通芯片工作	I
5	Vcc	供电电源	—
6	GND	地	—

3. 串口控制

详细参考 4.4 节 UART 通讯实验。

4. 电路连接图

本实验中总体电路连接图

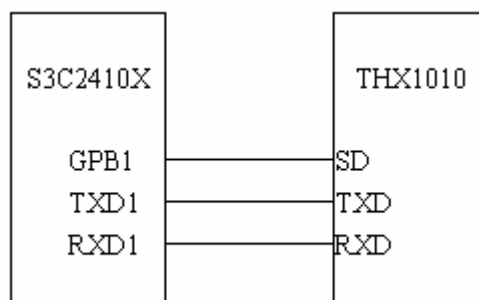


图 6-6-1 电路连接图

THX1010 把接收到的数据通过 S3C2410X 的 UART1 传送给 CPU, S3C2410X 把待发送的数据通过 UART1 发送给 THX1010 的 TXD 端, 再通过红外发射二极管发送。

6.6.5 实验操作步骤



1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接目标板, 使用 Embest EduKit-III 实验板附带的串口线, 连接实验板上的 UART0 和 PC 机的串口, 把跳线 SW503、SW504 断开 (1, 2, 3 都不接)。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序 (波特率 115200、1 位停止位、无校验位、无硬件流控制); 或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下 (如果已经拷贝, 可跳过此步骤);
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板, 打开实验例程目录 6.6_irda_test 子目录下的 irda_test.Uv2 例程, 编译链接工程;
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境 (工程默认已经配置正确), 点击工具栏 “”, 在 Option for Target 对话框的 Linker 页中选择 RuninRAM.sct 分散加载文件, 点击 MDK 的 Debug 菜单, 选择 Start/Stop Debug Session 项或点击工具栏 “”, 下载工程生成的.axf 文件到目标板的 RAM 中调试运行;
- 4) 如果需要将程序烧写固化到 Flash 中, 仅需要更改分散加载文件即可 (**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序, 建议实验中不操作**)。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件, 重新编译工程, 点击 MDK 的 Flash 菜单, 选择 Download 烧写调试代码到目标系统的 Nor Flash 中, 重启实验板, 实验板将会运行烧写到 Nor Flash 中的代码;
- 5) 点击 Debug 菜单的 Go 或 F5 键运行程序;
- 6) 程序正确运行后, 会在超级终端上输出如下信息:

```
boot success...
```

```
IrDA Receive Test Example
```

```
Select the baud rate:
```

```
1)9600    2)19200    3)38400    4)57600    5)115200
```

7) 使用 PC 机键盘,输入数字 1~5 中的任意一个数字来设置红外数据传输波特率。

8) 输入 2 后, 选择波特率为 19.2kbps, 出现等待接收红外信号界面:

```
rUBRDIV1=164
```

```
Now... Rx with IrDA
```

9) 连续按遥控 5 次, 出现接收到信号, 并等待选择波特率传输。

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 240, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
.....
```

```
IrDA Transfer Test Example
```

```
Select the baud rate:
```

```
1)9600    2)19200    3)38400    4)57600    5)115200
```

10) 输入 2, 出现传输结果。

```
rUBRDIV1=164
```

```
Now... Tx with IrDA
```

```
.....0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 240, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
Transfer data count: 256
```

```
end.
```

5. 完成实验练习题

理解和掌握实验后, 完成实验练习题, 让 IRDA 直接发送信号。

6.6.6 实验参考程序

```

/*****
* name:      irda_tx_int
* func:      IrDA transmit interrupt handler
* para:      none
* ret:       none
* modify:
* comment:
*****/

```

```

*****/

void irda_tx_int(void)
{
    rINTSUBMSK |= (BIT_SUB_RXD1|BIT_SUB_TXD1|BIT_SUB_ERR1);
    if(IrDA_cnt < (IrDA_BUFLen))
    {
        uart_printf(" %d,",*IrDAdataPt);
        WrUTXH1(*IrDAdataPt++);
        IrDA_cnt++;
        rSUBSRCPND = (BIT_SUB_TXD1);
        rINTSUBMSK &= ~(BIT_SUB_TXD1);
        ClearPending(BIT_UART1);
    }
    else
    {
        IrDA_end=1;
        while(rUFSTAT1 & 0x2f0);           //Until FIFO is empty
        while(!(rUTRSTAT1 & 0x4))         //Until Tx shifter is empty
        ClearPending(BIT_UART1);
        rINTMSK |= BIT_UART1;
    }
}

}

/*****
* name:      irda_rx_or_err
* func:      IrDA receive or error interrupt entry point
* para:      none
* ret:       none
* modify:
* comment:
*****/

void irda_rx_or_err(void)
{
    rINTSUBMSK |= (BIT_SUB_RXD1|BIT_SUB_TXD1|BIT_SUB_ERR1);    // Just for the safety
    if(rSUBSRCPND & BIT_SUB_RXD1)
        irda_rx_int();
    else
        irda_rx_int_err();
    rSUBSRCPND = (BIT_SUB_RXD1 | BIT_SUB_TXD1 | BIT_SUB_ERR1);
    rINTSUBMSK &= ~(BIT_SUB_RXD1|BIT_SUB_ERR1);
    ClearPending(BIT_UART1);
}

/*****
* name:      irda_rx_int
* func:      IrDA receive interrupt handle function
* para:      none

```

```

* ret:      none
* modify:
*****/

void irda_rx_int(void)
{
    while( (rUFSTAT1 & 0x100) || (rUFSTAT1 & 0xf) )
    {
        *IrDAdataPt = rURXH1;
        uart_printf(" %d,", *IrDAdataPt++);
        IrDA_cnt++;
    }
    if(IrDA_cnt >= IrDA_BUFLen)
    {
        IrDA_end = 1;
        rINTMSK |= BIT_UART1;
    }
}

/*****
* name:      irda_rx_int_err
* func:      IrDA transmit interrupt handle function
* para:      none
* ret:      none
* modify:
* comment:
*****/

void irda_rx_int_err(void)
{
    switch(rUERSTAT1)//to clear and check the status of register bits
    {
        case '1':
            uart_printf(" Overrun error\n");
            break;
        case '2':
            uart_printf(" Parity error\n");
            break;
        case '4':
            uart_printf(" Frame error\n");
            break;
        case '8':
            uart_printf(" Breake detect\n");
            break;
        default :
            break;
    }
}

```

```

/*****
* name:      irda_test_tx
* func:      IrDA transmit test function
* para:      none
* ret:       none
*****/

void irda_test_tx(void)
{
    int i;
    IrDA_cnt=0;
    IrDA_end=0;
    IrDAdataFl=(volatile UINT8T *)IrDABUFFER;
    IrDAdataPt=(volatile UINT8T *)IrDABUFFER;
    irda_port_set();
    uart_select(0);
    pISR_UART1=(UINT32T)irda_tx_int;
    uart_printf("\n IrDA Transfer Test Example\n");
    uart_printf(" Select the baud rate:\n"); // Select IrDA baud rate
    uart_printf(" 1)9600    2)19200    3)38400    4)57600    5)115200\n");
    i=uart_getch ();
    switch(i)
    {
        case '1':
            IrDA_BAUD=9600;
            break;
        case '2':
            IrDA_BAUD=19200;
            break;
        case '3':
            IrDA_BAUD=38400;
            break;
        case '4':
            IrDA_BAUD=57600;
            break;
        case '5':
            IrDA_BAUD=115200;
            break;
        default:
            break;
    }
    rUBRDIV1=( (int)(PCLK/16./IrDA_BAUD) -1 );
    uart_printf(" rUBRDIV1=%d\n", rUBRDIV1);
    uart_txempty(0);
    rGPBDAT &= ~(1<<1); // Enable nIrDATXDEN
    //Tx and Rx FIFO Trigger Level:4byte,Tx and Rx FIFO Reset,FIFO on

```

```

rUFCON1 = (1<<6) | (0<<4) | (1<<2) | (1<<1) | (1);
// PCLK,Tx&Rx:Level,Rx timeout:x,Rx error int:x,Loop-back:x,Send break:x,Tx:x,Rx:x
rUCON1 = (0<<10) | (1<<9) | (1<<8) | (0<<7) | (0<<6) | (0<<5) | (0<<4) | (0<<2) | (0);
// IrDA,No parity,One stop bit, 8bit
rULCON1 = (1<<6) | (0<<3) | (0<<2) | (3);    //uart_printf(" Press any key to start Tx first...\n");
//uart_getch();
rUCON1 = (0<<10) | (1<<9) | (1<<8) | (0<<7) | (0<<6) | (0<<5) | (0<<4) | (1<<2) | (0);
// PCLK,Tx&Rx:Level,Rx timeout:x,Rx error int:x,Loop-back:x,Send break:x,Tx:int,Rx:x
uart_printf(" Now... Tx with IrDA\n");

rINTMSK    &=~ (BIT_UART1);
rINTSUBMSK &=~ (BIT_SUB_RXD1|BIT_SUB_TXD1|BIT_SUB_ERR1);
    while(!IrDA_end);
rINTSUBMSK |= (BIT_SUB_RXD1|BIT_SUB_TXD1|BIT_SUB_ERR1);
rUFCON1 = (3<<6)|(2<<4)|(1<<2)|(1<<1)|(0);
rGPBDAT |= (1<<1);                // Disable nIrDATXDEN
uart_printf("\n Transfer data count: %d\n",IrDA_cnt);
rINTMSK    |= BIT_UART1;
rINTSUBMSK |= (BIT_SUB_RXD1|BIT_SUB_TXD1|BIT_SUB_ERR1);
irda_port_return();
}
/*****
* name:      irda_test_rx
* func:      IrDA receive test function
* para:      none
* ret:       none
* modify:
* comment:
*****/
void irda_test_rx(void)
{
    unsigned int i;
    IrDA_cnt=0;
    IrDA_end=0;
    IrDA_err=0;
    IrDAdataFl = (volatile UINT8T *)IrDABUFFER;
    IrDAdataPt = (volatile UINT8T *)IrDABUFFER;
    irda_port_set();
    uart_select(0);
    pISR_UART1 = (unsigned)irda_rx_or_err;
    uart_printf("\n IrDA Receive Test Example\n");
    uart_printf(" Select the baud rate:\n"); // Select IrDA baud rate
    uart_printf(" 1)9600    2)19200    3)38400    4)57600    5)115200\n");
    i=uart_getch ();
    switch(i)
    {

```

```

    case '1':
        IrDA_BAUD=9600;
        break;
    case '2':
        IrDA_BAUD=19200;
        break;
    case '3':
        IrDA_BAUD=38400;
        break;
    case '4':
        IrDA_BAUD=57600;
        break;
    case '5':
        IrDA_BAUD=115200;
        break;
    default:
        break;
}

rUBRDIV1=( (int)(PCLK/16./IrDA_BAUD) -1 );
uart_printf(" rUBRDIV1=%d\n", rUBRDIV1);
rGPBDAT &= ~(1<<1);          // Enable nIrDATXDEN
//Tx and Rx FIFO Trigger Level:4byte,Tx and Rx Reset,FIFO En
rUFCON1 = (1<<6) | (0<<4) | (1<<2) | (1<<1) | (1);
// PCLK,Tx&Rx:Level,Rx timeout:x,Rx error int:o,Loop-back:x,Send break:x,Tx:x,Rx:x
rUCON1 = (0<<10) | (1<<9) | (1<<8) | (0<<7) | (1<<6) | (0<<5) | (0<<4) | (0<<2) | (0);
// Infra-red mode,No parity,One stop bit, 8bit
rULCON1 = (1<<6) | (0<<3) | (0<<2) | (3);
delay(1);
// PCLK,Tx&Rx:Level,Rx timeout:x,Rx error int:o,Loop-back:x,Send break:x,Tx:x,Rx:int
rUCON1 = (0<<10) | (1<<9) | (1<<8) | (0<<7) | (1<<6) | (0<<5) | (0<<4) | (0<<2) | (1);
uart_printf(" Now... Rx with IrDA\n");    rINTMSK    &= ~(BIT_UART1);
rINTSUBMSK &= ~(BIT_SUB_RXD1|BIT_SUB_ERR1);
while(!IrDA_end);
rINTMSK    |= BIT_UART1;
rINTSUBMSK |= (BIT_SUB_RXD1 | BIT_SUB_TXD1 | BIT_SUB_ERR1);    rUFCON1 = (3<<6) | (2<<4) |
(1<<2) | (1<<1) | (0);
    irda_port_return();
}

/*****
* name:      irda_test
* func:      IrDA receive and transfer test function
* para:      none
* ret:       none
* modify:
* comment:

```

```
*****/  
  
void irda_test(void)  
{  
    UINT32T szBuf[40];  
    irda_test_rx();  
    irda_test_tx();  
    uart_printf(" end.\n");  
}
```

6.6.7 练习题

编写程序实现不定长的红外数据发送和接收程序。

第七章 基础应用实验

7.1 A/D 转换实验

7.1.1 实验目的

- 通过实验掌握模数转换 (A/D) 的原理。
- 掌握 S3C2410X 处理器的 A/D 转换功能。

7.1.2 实验设备

- 硬件: Embest EduKit-III 实验平台, ULINK USB-JTAG 仿真器套件, PC 机。
- 软件: μ Vision IDE for ARM 集成开发环境, Windows 98/2000/NT/XP。

7.1.3 实验内容

设计分压电路, 利用 S3C2410X 集成的 A/D 模块, 把分压值转换为数字信号, 并通过超级终端和数码管观察转换结果。

7.1.4 实验原理

1. A/D 转换器 (ADC)

随着数字技术, 特别是计算机技术的飞速发展与普及, 在现代控制、通信及检测领域中, 对信号的处理广泛采用了数字计算机技术。由于系统的实际处理对象往往都是一些模拟量 (如温度、压力、位移、图像等), 要使计算机或数字仪表能识别和处理这些信号, 必须首先将这些模拟信号转换成数字信号, 这就必须用到 A/D 转换器。

2. A/D 转换的一般步骤

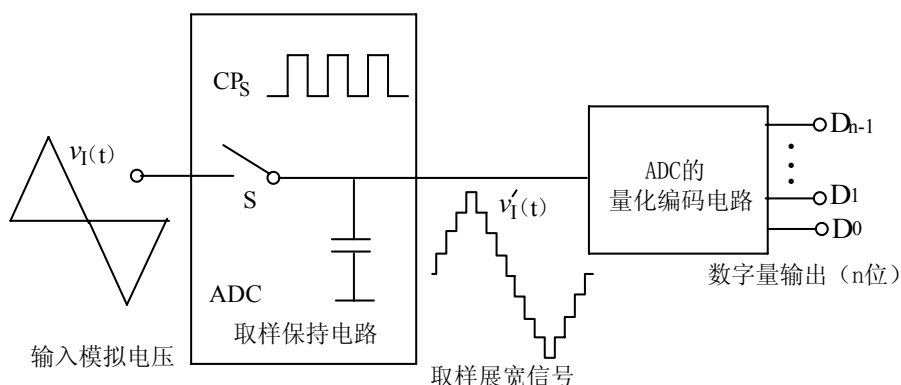


图 7-1-1 模拟量到数字量的转换过程

模拟信号进行 A/D 转换的时候, 从启动转换到转换结束输出数字量, 需要一定的转换时间, 在这个转换时间内, 模拟信号要基本保持不变。否则转换精度没有保证, 特别当输入信号频率较高时, 会造成很大的转换误差。要防止这中误差的产生, 必须在 A/D 转换开始时将输入信号的电平保持住, 而在 A/D 转换结束后, 又能跟踪输入信号的变化。因此, 一般的 A/D 转换过程是通过取样、保持、量化和编码这四个步骤完成的。一般取样和保持主要由采样保持器来完成, 而量化编码就由 A/D 转换器完成。

3. S3C2410X 处理器的 A/D 转换

处理器内部集成了采用近似比较算法（计数式）的 8 路 10 位 ADC，集成零比较器，内部产生比较时钟信号；支持软件使能休眠模式，以减少电源损耗。其主要特性：

- 精度（Resolution）：10-bit
- 微分线性误差（Differential Linearity Error）： ± 1.5 LSB
- 积分线性误差（Integral Linearity Error）： ± 2.0 LSB
- 最大转换速率（Maximum Conversion Rate）：500 KSPS
- 输入电压（Input voltage range）：0-3.3V
- 片上采样保持电路
- 正常模式
- 单独 X,Y 坐标转换模式
- 自动 X,Y 坐标顺序转换模式
- 等待中断模式

4. S3C2410X 处理器 A/D 转换器的使用

寄存器组

处理器集成的 ADC 只使用到两个寄存器，即 ADC 控制寄存器（ADCCON）、ADC 数据寄存器（ADCDAT）。

ADC 控制寄存器（ADCCON）

寄存器	地址	R/W	功能描述	复位值
ADCCON	0x58000000	R/W	ADC 控制寄存器	0x3FC4

ADCCON[15]： A/D 转换结束标志

0： A/D 转换正在进行；

1： A/D 转换结束

ADCCON[14]： AD 转换预分频允许

0： 不允许预分频

1： 允许预分频

ADCCON[13:6]:预分频值 PRSCVL

PRSCVL 在 0 到 255 之间，实际的分频值为 PRSCVL+1

ADCCON[5:3]： 模拟信道输入选择

000 = AIN0

001 = AIN1

010 = AIN2

011 = AIN3

100 = AIN4

101 = AIN5

110 = AIN6

111 = AIN7

- ADCCON[2]: 待机模式选择位
 0:正常模式
 1:待机模式
- ADCCON[1]: A/D 转换读—启动选择位
 0:禁止 Start-by-read
 1:允许 Start-by-read
- ADCCON[0]: A/D 转换器启动
 0:A/D 转换器不工作
 1: A/D 转换器开始工作

ADC 数据寄存器 (ADCDAT0,ADCDAT1)

寄存器	地址	R/W	功能描述	复位值
ADCDAT0	0x5800000C	R	ADC 数据寄存器	—

- ADCDAT0[15]: 等待中断模式, Stylus 电平选择
 0:低电平
 1:高电平

- ADCDAT0[14]: 自动按照先后顺序转换 X,Y 坐标
 0:正常 ADC 顺序
 1:按照先后顺序转换

- ADCDAT0[13:12]: 自定义 X,Y 位置
 00:无操作模式
 01:测量 X 位置
 10:测量 Y 位置
 11:等待中断模式

- ADCDAT0[11:10]: 保留

- ADCDAT0[9:0]: X 坐标转换数据值

寄存器	地址	R/W	功能描述	复位值
ADCDAT1	0x58000010	R	ADC 数据寄存器	—

- ADCDAT1[15:10]与 ADCDAT0[15:10]功能相同

- ADCDAT0[9:0]: Y 坐标转换数据值

A/D 转换的转换时间计算

例如 PCLK 为 50MHz, PRESCALER=49; 所有 10 位转换时间为:

$$50 \text{ MHz} / (49+1) = 1\text{MHz}$$

转换时间为 $1/(1\text{M}/5 \text{ cycles})=5\mu\text{s}$,

注意，A/D 转换器的最大工作时钟为 2.5MHz，所以最大的采样率可以达到 500ksps。

7.1.5 实验设计

1. 分压电路设计

分压电路比较简单，为了保证电压转换时是稳定的，可以直接调节可变电阻得到稳定的电压值。Embest EduKit-III 实验平台的分压电路见图 7-1-2。

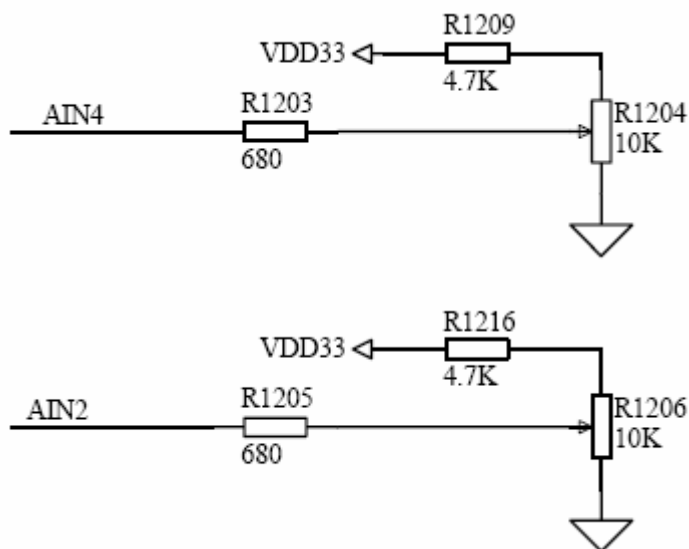


图 7-1-2 分压电路

2. 软件程序设计

实验主要是对 S3C2410X 中的 A/D 模块进行操作，所以软件程序也主要是对 A/D 模块中的寄存器进行操作，其中包括对 ADC 控制寄存器（ADCCON）、ADC 数据寄存器（ADCDAT）的读写操作。同时为了观察转换结果，可以通过串口在超级终端里面观察。

7.1.6 实验操作步骤


1. 准备实验环境


使用 ULINK USB-JTAG 仿真器连接目标板，使用 Embest EduKit-III 实验板附带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板，打开实验例程目录 7.1_adc_test 子目录下的 adc_test.Uv2 例程，编译链接工程；
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境（工程默认已经配置正确），点击工具栏 “”，在 Option for Target 对话框的 Linker 页中

选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug Session 项或点击工具栏 “”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；

- 4) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；

- 5) 点击 Debug 菜单的 Go 或 F5 键运行程序；

4. 观察实验结果

- 1). 在 PC 机上观察超级终端程序主窗口，可以看到如下界面：

```
boot success...
```

```
[ ADC_IN Test,channel 2]
```

```
ADC conv. freq. = 2500000Hz
```

```
Please adjust AIN2 value!
```

```
The results of ADC are:
```

- 2). 本实验对 AIN2 输入信号进行 A/D 转换，调节电位器 R1206，可以在超级终端和数码管上观察到变化的数据，本实验使用 start-by-read 模式，每 1 秒钟对数据进行一次采样，共采样 20 个点。超级终端主窗口和数码管显示出采样点的数据经过转换后的结果。

```
[ ADC_IN Test,channel 2]
```

```
ADC conv. freq. = 2500000Hz
```

```
Please adjust AIN2 value!
```

```
The results of ADC are:
```

```
3.2742  3.2742  2.9097  1.9452  1.6774  1.6161  1.0871  0.8194
1.2194  1.2806  1.6290  1.6290  1.9161  2.1065  2.2839  2.2871
1.7032  1.1806  0.7161  0.0677
```

5. 完成实验练习题

理解和掌握实验后，完成实验练习题。

7.1.7 实验参考程序

```
#include "2410lib.h"
#define REQCNT 100
#define ADC_FREQ 2500000
#define LOOP 10000
/*-----*/
/*                      global variables                      */
/*-----*/
volatile UINT8T unPreScaler;
volatile char nEndTest;
```

```

extern void iic_init_8led(void);
extern void iic_write_8led(UINT32T unSlaveAddr, UINT32T unAddr, UINT8T ucData);
extern void iic_read_8led(UINT32T unSlaveAddr, UINT32T unAddr, UINT8T *pData);
/*****
* name:      adc_test
* func:      ADC convert test
* para:      none
* ret:       none
* modify:
* comment:
*****/

void adc_test(void)
{
    int i,j;
    UINT16T usConData;
    float usEndData;
    UINT8T f_szDigital[10] = {0xFC,0x60,0xDA,0xF2,0x66,0xB6,0xBE,0xE0,0xFE,0xF6}; // 0 ~ 9
    uart_printf("\n Adc Conversion Test Example (Please look at 8-seg LED)\n");
    iic_init_8led(); // initialize iic and leds
    for(i=0;i<8;i++)
        iic_write_8led(0x70,0x10+i,0);
    uart_printf(" ADC_IN Test,channel 2\n");
    uart_printf(" ADC conv. freq. = %dHz\n",ADC_FREQ);
    unPreScaler = PCLK/ADC_FREQ -1;
    rADCCON=(1<<14)|(unPreScaler<<6)|(2<<3)|(0<<2)|(1<<1); //enable prescaler,ain2,normal,start by read
    uart_printf(" Please adjust AIN2 value!\n");
    uart_printf(" The results of ADC are:\n");
    usConData=rADCDA0&0x3FF;
    for(j=0;j<20;j++) // sample and show data both by UART and leds
    {
        while(!(rADCCON & 0x8000));
        usConData=rADCDA0&0x3FF;
        usEndData=usConData*3.3000/0x3FF;
        uart_printf(" %0.4f ",usEndData);
        iic_write_8led(0x70,0x10+4,f_szDigital[(int)usEndData]+1);
        usEndData=usEndData-(int)usEndData;
        for(i=0;i<4;i++)
        {
            usEndData=usEndData*10;
            iic_write_8led(0x70,0x10+3-i,f_szDigital[(int)usEndData]);
            usEndData=usEndData-(int)usEndData;
        }
        delay(10000);
    }
    uart_printf(" end.\n");
}

```

}

7.1.8 练习题

参考实验程序，采用分压电路，对第四通道的输入电压进行循环采样，循环周期为 0.5s，并求出采样数据的平均值。

7.2 PWM 步进电机控制实验

7.2.1 实验目的

- 通过实验掌握 S3C2410X 的 PWM 控制方式和工作原理
- 通过实验掌握 S3C2410X 的定时器寄存器的使用；
- 熟练掌握控制 S3C2410X 的 PWM 定时器的软件编程方法。

7.2.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

7.2.3 实验内容

编写程序，实现：

分别用定时器 T0，T1 的 PWM 控制步进电机；

按下按键 SB1202 启动步进电机，按下 SB1203 改变步进电机的转动方向；

改变定时器 T0，T1 的计数器寄存器的值改变步进电机的转动速率。

7.2.4 实验原理

1. S3C2410X 的 PWM 定时器概述

S3C2410X 有 5 个 16 位定时器，其中定时器 0、定时器 1、定时器 2 与定时器 3 具有脉冲宽度调制（PWM）功能，定时器 4 仅供内部定时而没有输出引脚。定时器 0 具有死区生成器，可以控制大电流设备。

定时器 0 与定时器 1 共用一个 8bit 预分频器，定时器 2、定时器 3 与定时器 4 共用另一个 8bit 预分频器，每个定时器都有一个时钟分频器，时钟分频器有 5 种分频输出（1/2，1/4，1/8，1/16 和外部时钟 TCLK）。每个定时器都从时钟分频器接收的时钟信号，时钟分频器从相应的 8bit 预分频器接收时钟信号。可编程 8bit 预分频器根据存储在 TCFG0 和 TCFG1 中的数据对 PCLK 进行分频。

当时钟被使能后，定时器计数缓冲寄存器（TCNTBn）把计数初值下载到递减计数器中。定时器比较缓冲寄存器（TCMPBn）把其初始值下载到比较寄存器中，并将该值和递减计数器的值进行比较。这种基于 TCNTBn 和 TCMPBn 的双缓冲特性使定时器在频率和占空比变化时产生稳定的输出。

每个定时器都有一个专用的由定时器时钟驱动的 16 位递减计数器。当递减计数器的计数值达到 0 的时，就会产生定时器中断请求来通知 CPU 定时器操作完成。当定时器递减计数器达到 0 的时候相应的 TCNTBn 的值会自动重载到递减计数器中以继续下次操作。然而，如果定时器停止了，比如在定时器运行时清除 TCON 中的定时器使能位，TCNTBn 的值不会被重载到递减计数器中。

TCMPBn 的值用于脉冲宽度调制（PWM）。当定时器的递减计数器的值和比较寄存器的值相匹配的时候，定时器控制逻辑将改变输出电平。因此，比较寄存器决定了 PWM 输出的开关时间。

S3C2410x 的定时器特性：

- 5 个 16bit 定时器
- 两个 8bit 预分频器和两个 4bit 分频器

- 输出波形的占空比可编程(PWM)
- 自动重载模式或者单脉冲模式
- 具有死区生成器

自动重载与双缓冲

S3C2410X 的 PWM 定时器具有双缓冲功能，能在不停止当前定时器运行的情况下，重载定时器下次运行的参数，所以尽管新的定时器的值被设置好了，但是当前操作仍能成功完成。

定时器值可以被写入定时器计数缓冲寄存器 (TCNTBn)，当前的计数器的值可以从定时器计数观察寄存器 (TCNTOn) 读出，读出的 TCNTBn 值并不是当前的计数值，而是下次将重载的计数值。

TCNTn 的值等于 0 的时候，自动重载操作把 TCNTBn 的值装入 TCNTn，只有当自动重载功能被使能并且 TCNTn 的值等于 0 的时候才会自动重载。如果 TCNTn 等于 0，自动重载控制位为 0，则定时器停止运行。

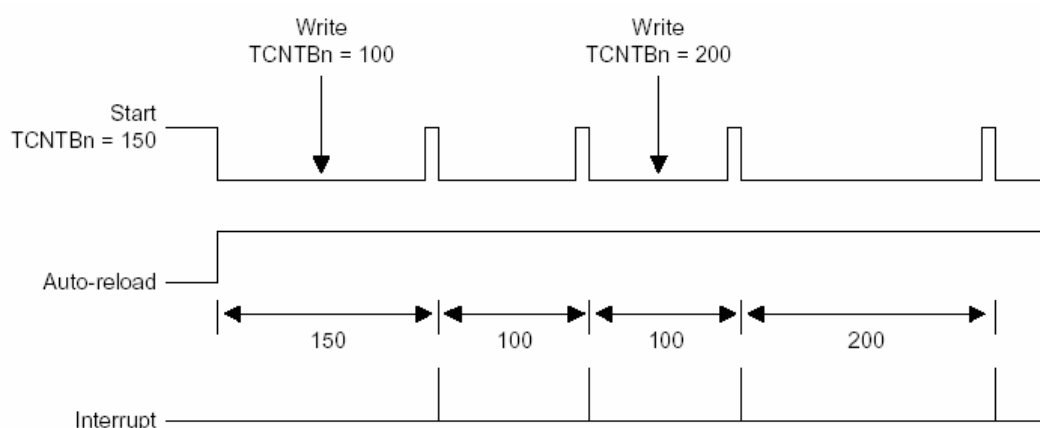


图 7-3-1 双缓冲功能举例

使用手动更新位和反转位完成定时器的初始化

当递减计数器的值达到 0 时会发生定时器自动重载操作，所以 TCNTn 的初始值必须由用户提前定义好，在这种情况下就需要通过手动更新位重载初始值。以下几个步骤给出如何启动定时器：

- (1) 向 TCNTBn 和 TCMPBn 写入初始值。
- (2) 置位相应定时器的手动更新位，不管是否使用反转功能，推荐设置反转位。
- (3) 置位相应定时器的启动位启动定时器，清除手动更新位。

如果定时器被强制停止，TCNTn 保持原来的值而不从 TCNTBn 重载值。如果要设置一个新的值必须执行手动更新操作。

注意：只要 TOUT 的反转位改变，不管定时器是否处于运行状态，TOUT 都会相应的改变，因此通常同时配置手动更新位和反转位。

定时器操作示例

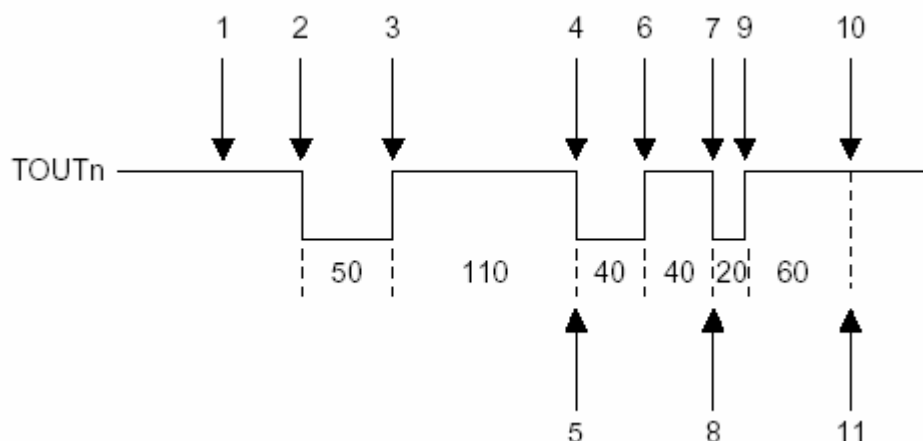


图 7-3-2 定时器操作示例

(1) 使能自动重载功能，设置 TCNTBn 值为 160 (50+110) TCMPBn 值为 110。置位手动更新位，配置反转位。置位手动更新位将使 TCNTBn 和 TCMPBn 的值加载到 TCNTn 和 TCMPn。然后设置 TCNTBn 和 TCMPBn 分别等于 80 (40+40) 和 40。

(2) 将手动更新位设为 0，将反转位设为 off，使能自动重载功能，置位启动位，则在定时器分辨率内的一段延迟后定时器开始递减计数。

(3) 当 TCNTn 和 TCMPn 的值相等的时候，TOUT 输出电平由低变高。

(4) 当 TCNTn 的值等于 0 的时候产生中断并且 TCNTBn 的值装入暂存器中，在下一个时钟到来时将暂存器中的值重载到 TCNTn。

(5) 在中断服务程序中，将 TCNTBn 和 TCMPBn 分别设置为 80 (20+60) 和 60。

(6) 当 TCNTn 和 TCMPn 的值相等的时候，TOUT 输出电平由低变高。

(7) 当 TCNTn 等于 0 的时候，把 TCNTBn 和 TCMPBn 的值分别自动装入 TCNTn 和 TCMPn，并触发中断。

(8) 在中断服务子程序中，禁止自动重载和中断请求来停止定时器运行。

(9) 当 TCNTn 和 TCMPn 的值相等的时候，TOUT 输出电平由低变高。

(10) 尽管 TCNTn 等于 0，但是定时器停止运行，也不在发生自动重载操作，因为定时器自动重载功能被禁止。

(11) 不再产生新的中断。

死区生成器

死区功能用于电源设备的 PWM 控制。这个功能允许在一个设备关闭和另一个设备开启之间插入一个时间间隔。这个时间间隔可以防止两个设备同时被启动。

TOUT0 是定时器 0 的 PWM 输出，nTOUT0 是 TOUT0 的反转信号。如果死区功能被使能，TOUT0 和 nTOUT0 的输出波形就变成了 TOUT0_DZ 和 nTOUT0_DZ (如图 7-3-3 所示)。nTOUT0_DZ 在 TOUT1 脚上产生。

在死区间隔内，TOUT0_DZ 和 nTOUT0_DZ 就不会同时翻转了。

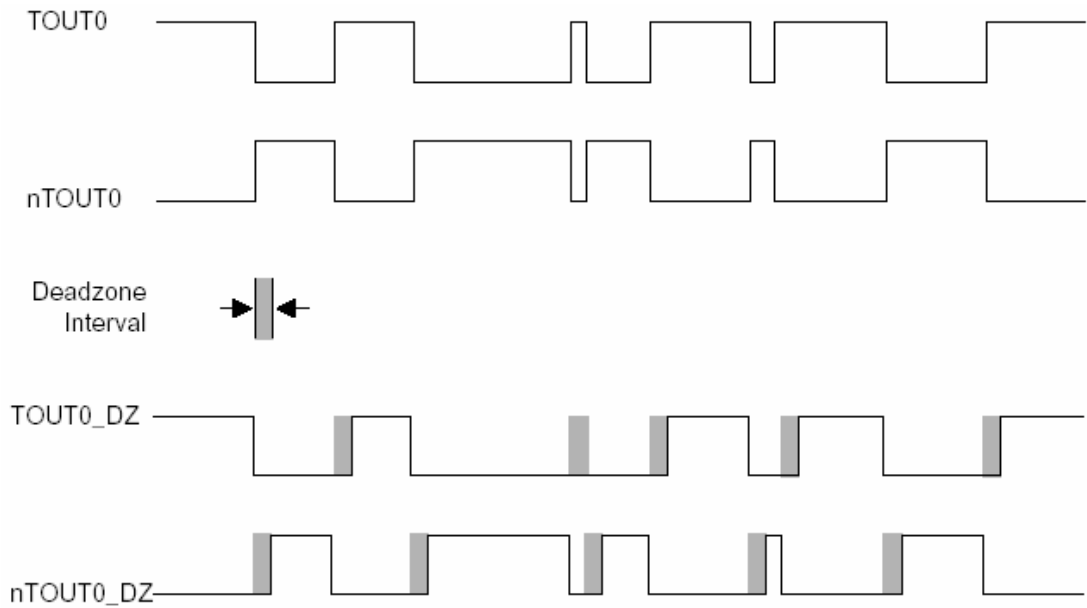


图 7-3-3 死区功能允许波形图

2. S3C2410X 的 PWM 定时器控制寄存器

定时器配置寄存器 0 (TCFG0, 地址: 0x51000000)。
定时器输入时钟频率(TCLK)=PCLK/ (预分频值+1) /分频器分频值。
预分频值=0—255。
分频器的分频值为 2、4、8、16。

$$\text{PWM 输出时钟频率} = \frac{\text{定时器输入时钟频率 (TCLK)}}{\text{定时器计数缓冲寄存器值 (TCNTBn)}}。$$

$$\text{PWM 输出信号的占空比} = \frac{\text{定时器比较缓冲寄存器值 (TCMPBn)}}{\text{定时器计数缓冲寄存器值 (TCNTBn)}}。$$

TCFG0	位	描述	初始化状态
保留	[31:24]		0x00
死区长度	[23:16]	这 8bit 控制死区的长度，一个单元时间的长度等于定时器 0 的一单元时间长度	0x00
预分频器 1	[15:8]	这 8bit 数据等于定时器 2、3、4 的预分频值	0x00
预分频器 0	[7:0]	这 8bit 数据等于定时器 0、1 的预分频值	0x00

定时器控制寄存器 1 (TCFG1, 地址: 0x51000004)

TCFG1	位	描述	初始化状态
保留	[31:24]		00000000
DMA 模式选择	[23:20]	DMA 请求选择 0000=无 DMA 通道选择 0001=定时器 0 DMA 方式	0000

		0010=定时器 1 DMA 方式 0011=定时器 2 DMA 方式 0100=定时器 3 DMA 方式 0101=定时器 4 DMA 方式 0110=保留	
多路开关 4 选择	[19:16]	定时器 4 多路输入选择 0000=1/2 0001=1/4 0010=1/8 0011=1/16 01XX=外部时钟 1	0000
多路开关 3 选择	[15:12]	定时器 3 多路输入选择 0000=1/2 0001=1/4 0010=1/8 0011=1/16 01XX=外部时钟 1	0000
多路开关 2 选择	[11:8]	定时器 2 多路输入选择 0000=1/2 0001=1/4 0010=1/8 0011=1/16 01XX=外部时钟 1	0000
多路开关 1 选择	[7:4]	定时器 1 多路输入选择 0000=1/2 0001=1/4 0010=1/8 0011=1/16 01XX=外部时钟 0	0000
多路开关 0 选择	[3:0]	定时器 0 多路输入选择 0000=1/2 0001=1/4 0010=1/8 0011=1/16 01XX=外部时钟 0	0000

定时器控制寄存器 (TCON, 地址: 0x51000008)

TCON	位	描述	初始化状态
定时器 4 自动重载 on/off	[22]	0=定时器 4 运行 1 次; 1=自动重载模式	0
定时器 4 手动更新位(*)	[21]	0=无操作, 1=更新 TCNTB4	0
定时器 4 启动位	[19]	0=无操作, 1=启动定时器 4	0
定时器 3 自动重载 on/off	[18]	0=定时器 3 运行 1 次; 1=自动重载模式	0
定时器 3 输出倒相位	[17]	0=倒相关闭, 1=TOUT3 倒相	0
定时器 3 手动更新位	[16]	0=无操作, 1=更新 TCNTB3	0
定时器 3 启动位(*)	[15]	0=无操作, 1=启动定时器 3	0
定时器 2 自动重载 on/off	[14]	0=定时器 2 运行 1 次; 1=自动重载模式	0
定时器 2 输出倒相位	[13]	0=倒相关闭, 1=TOUT2 倒相	0
定时器 2 手动更新位(*)	[12]	0=无操作, 1=更新 TCNTB2	0
定时器 2 启动位	[11]	0=无操作, 1=启动定时器 2	0
定时器 1 自动重载 on/off	[10]	0=定时器 2 运行 1 次; 1=自动重载模式	0
定时器 1 输出倒相位	[9]	0=倒相关闭, 1=TOUT1 倒相	0
定时器 1 手动更新位(*)	[8]	0=无操作, 1=更新 TCNTB1	0
定时器 1 启动		0=无操作, 1=启动定时器 1	0
保留	[7:5]	保留	
死区功能允许	[4]	0=禁止, 1=允许	0

定时器 0 自动重载 on/off	[3]	0=定时器 0 运行 1 次; 1=自动重载模式	0
定时器 0 输出倒相位	[2]	0=倒相关闭, 1=TOUT0 倒相	0
定时器 0 手动更新位(*)	[1]	0=无操作, 1=更新 TCNTB0	0
定时器 1 启动	[0]	0=无操作, 1=启动定时器 0	0

(*): 在下次写入的时候一定要清零

定时器减法缓冲寄存器 (TCNTBn) 和比较缓冲寄存器(TCMPBn)

寄存器	R/W	描述	复位值
TCNTBn	R/W	定时器 n 减法缓冲寄存器	0x00000000
TCMPBn	R/W	定时器 n 比较缓冲寄存器	0x00000000

TCNTBn[15:0]设置递减缓冲寄存器的值

TCMPBn[15:0]设置比较缓冲寄存器的值

定时器观察寄存器 (TCNTO_n, 地址: 0x5100000C~0x5100003C)

寄存器	地址	寄存器	地址	寄存器	地址
TCNTB0	0x5100000C	TCMPB0	0x51000010	TCNTO0	0x51000014
TCNTB1	0x51000018	TCMPB1	0x5100001C	TCNTO1	0x51000020
TCNTB2	0x51000024	TCMPB2	0x51000028	TCNTO2	0x5100002C
TCNTB3	0x51000030	TCMPB3	0x51000034	TCNTO3	0x51000038
TCNTB4	0x5100003C			TCNTO4	0x51000040

3. 电路原理

步进电机是一种将电脉冲转化为角位移的执行机构。通俗一点讲：当步进驱动器接收到一个脉冲信号，它就驱动步进电机按设定的方向转动一个固定的角度（及步进角）。可以通过控制脉冲个数来控制角位移量，从而达到准确定位的目的；同时您可以通过控制脉冲频率来控制电机转动的速度和加速度，从而达到调速的目的。

本实验使用的电机驱动器是日本东芝公司的 TA8435 芯片，该芯片的功能如下：

- 双向正弦曲线步进电机驱动器；
- 输出电流平均值为 1.5A，峰值 2.5A；
- PWM 输入源；
- 高电压 Bi-CMOS 处理技术；
- 电机可双向转动；
- HZIP-25P 封装；
- RESET 端输入阻抗 100kohm；
- 输出电机控制电流正负 2mA；

引脚功能如下：

引脚	符 号	功能描述
----	-----	------

1	SG	信号地
2	RESET	低电平有效
3	ENABLE	低电平选通
4	OSC	由外部电容计算晶振频率
5	CW/CCW	顺时针方向/逆时针方向转动选择
6	CK2	时钟输入
7	CK1	时钟输入
8	M1	激励控制输入
9	M2	激励控制输入
10	REF IN	输入参考电压
11	MO	监听输出
12	NC	未连接
13	VCC	逻辑电源
14	NC	未连接
15	VMB	输出电压
16	$\phi\bar{B}$	输出 $\phi\bar{B}$
17	PG-B	电源地
18	NFB	B 通道输出电流检测
19	ϕB	输出 ϕB
20	$\phi\bar{A}$	输出 $\phi\bar{A}$
21	NFA	A 通道输出电流检测
22	PG-A	电源地
23	ϕA	输出 ϕA
24	VMA	输出电源
25	NC	未连接

本实验用到的控制引脚有 EN、CW/CCW、M2、具体连接电路如下图所示

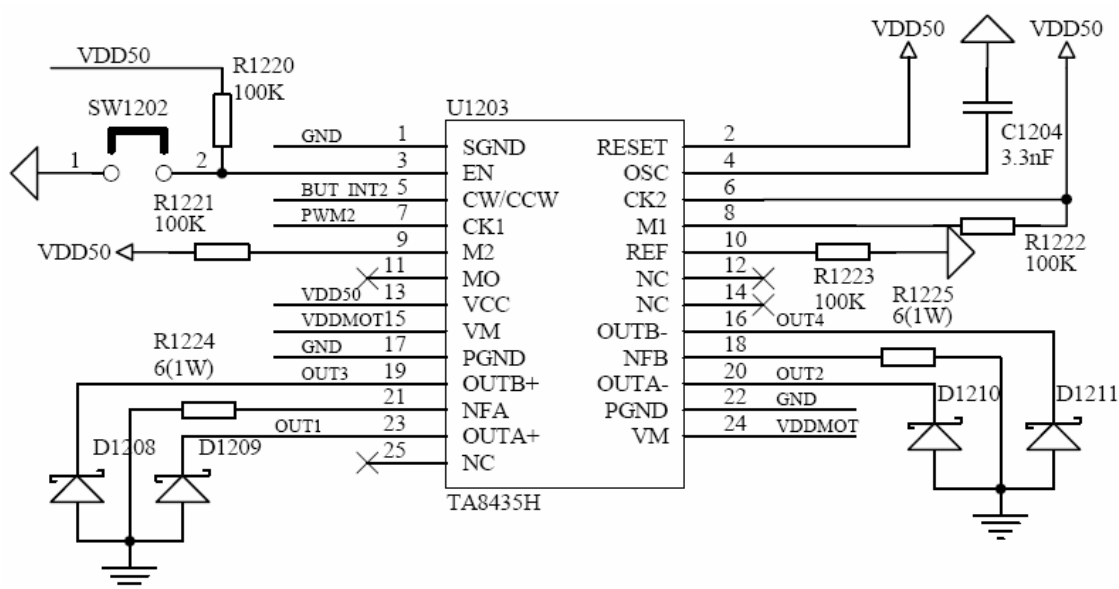


图 7-3-4 电机驱动电路图

注意：电机驱动电路图 7-3-4 中 PWM2 连接到 S3C2410X 的 TOUT1/GPB1 引脚（即本实验中使用定时器 1 的 PWM 驱动步进电机），BUT INT2 连接到 S3C2410X 的 EINT11/GPG3 引脚，用于选择步进电机的转动方向。

OUT1、OUT2、OUT3、OUT4 连到电机的控制端。如图 7-3-5 所示。

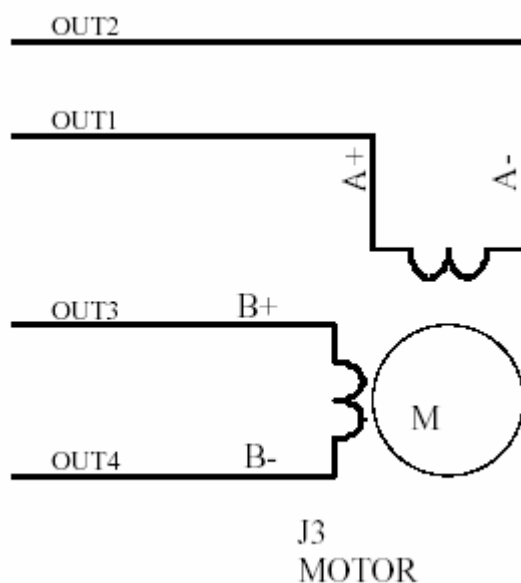


图 7-3-5 电机连接图

芯片 TA8435H 输出的相位信息如下图 7-3-6 所示。其中 I_A 、 I_B 分别对应 OUT1 与 OUT2、OUT3 与 OUT4 之间连线的电流。

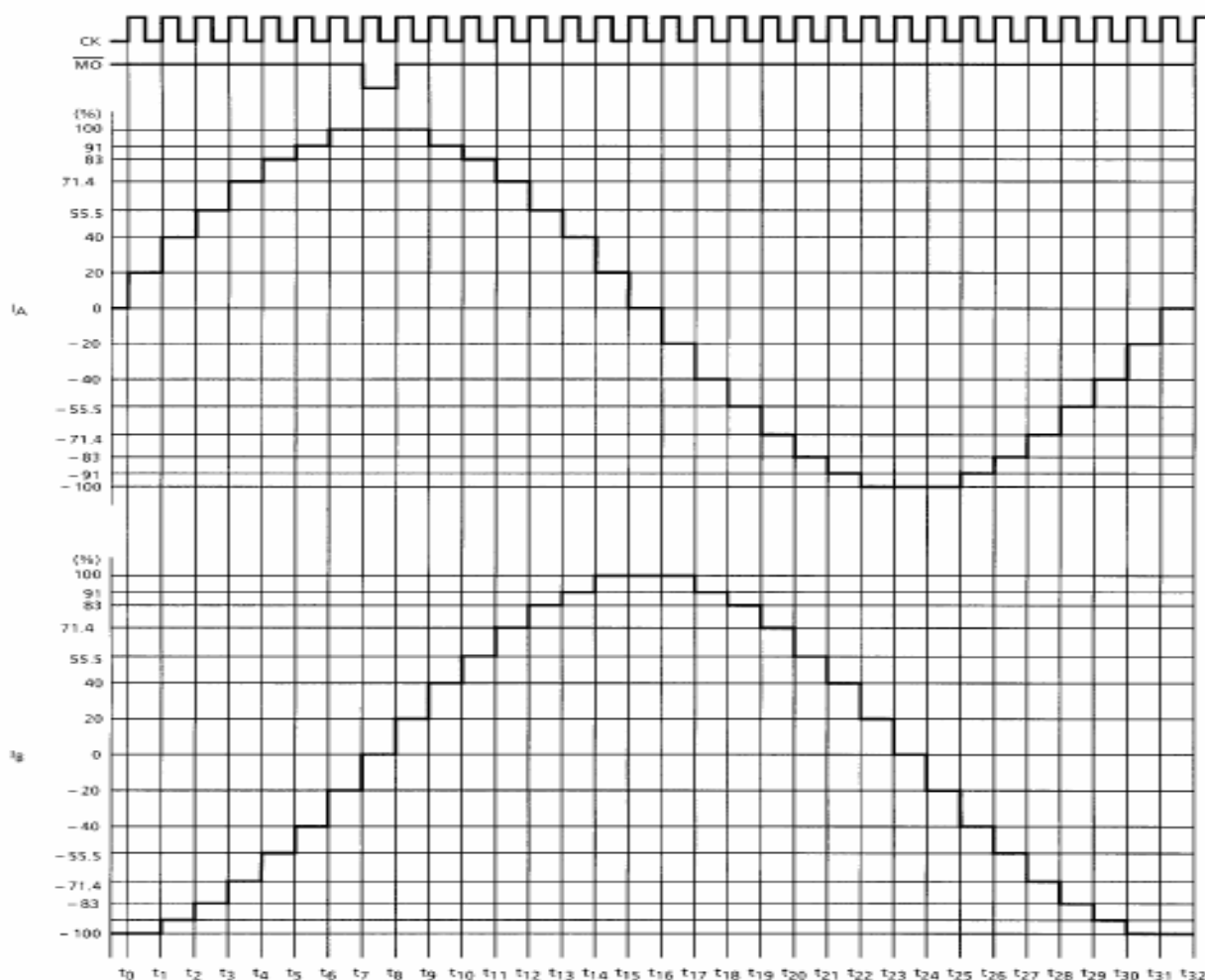


图 7-3-6 TA8435H 输出电机驱动电流示意图

注意：上图中相邻两个时间点的间隔等于 PWM 定时器输出的一个周期。占空比决定了定时器输出功率，一般占空比为 0.3-0.4，由于 PWM 输出信号 TOUT1 的输出信号的高低电平时间长短是不确定的，所以建议正常工作的时候的占空比为 0.5。

7.2.5 实验操作步骤

1. 准备实验环境



使用 ULINK USB-JTAG 仿真器连接目标板，使用 Embest EduKit-III 实验板附带的串口线，连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下（如果已经拷贝，可跳过此步骤）；
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板，打开实验例程目录 7.2_pwm_test 子目录下的 pwm_test.Uv2 例程，编译链接工程；

- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境（工程默认已经配置正确），点击工具栏 “”，在 Option for Target 对话框的 Linker 页中选择 RuninRAM.sct 分散加载文件，点击 MDK 的 Debug 菜单，选择 Start/Stop Debug Session 项或点击工具栏 “”，下载工程生成的.axf 文件到目标板的 RAM 中调试运行；
- 4) 如果需要将程序烧写固化到 Flash 中，仅需要更改分散加载文件即可（**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序，建议实验中不操作**）。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件，重新编译工程，点击 MDK 的 Flash 菜单，选择 Download 烧写调试代码到目标系统的 Nor Flash 中，重启实验板，实验板将会运行烧写到 Nor Flash 中的代码；
- 5) 点击 Debug 菜单的 Go 或 F5 键运行程序；
- 6) 程序正确运行后会在超级终端上输出如下信息：

```
boot success...
```

```
PWM test
```

```
Press SB1203 to change the direction of the motor
```

```
You can press the following key to change the duty ratio:
```

```
1-->PWM1: 1/2,100Hz(default)
```

```
2-->PWM1: 1/3,100Hz
```

```
3-->PWM1: 1/3,200Hz
```

```
4-->PWM1: 1/2,200Hz
```

```
Press SPACE to return!
```

7) 使用 PC 机键盘，输入数字 1—4，控制定时器 1 的 PWM 输出脉冲，利用定时器 1 的 PWM 输出脉冲控制步进电机。

8) 按下 SB1203 后（不松手），电机改变转动方向。也可以随时选择输入不同的数字来改变 PWM 输出脉冲的占空比，对于本实验用的小型步进电机，工作频率在 100Hz 左右能功能在最佳状态，当然也可以改变占空比和频率观察电机转动状况。

9) 结合实验内容和实验原理部分，掌握 S3C2410X 处理器定时器 PWM 工作原理。

4. 观察实验结果

在超级中断主窗口中输入 1-4，观察不同频率及占空比情况下 PWM 电机工作情况。

5. 完成实验练习题

理解和掌握实验后，完成实验练习题。

7.2.6 实验参考程序

```

/*****
* File:      pwm_test.c
* Author:    embest
* Desc:      s3c2410x pwm timer test
* History:

```



```

*****/

/*-----*/
/*          include files                      */
/*-----*/
#include "2410lib.h"

/*-----*/
/*          global variables                    */
/*-----*/
static INT8T Key_pressed;

/*****
* name:      timer1_int
* func:      Timer1 Interrupt handler
* para:      none
* ret:       none
* modify:
* comment:
*****/
void __irq timer1_int(void)
{
    ClearPending(BIT_TIMER1);
    rINTMSK |= BIT_TIMER1;          // Disable timer1 interrupt

    switch(Key_pressed)
    {
        case '1':
            rTCNTB1 = 792;
            rTCMPB1 = 396;
            uart_printf("\n You selected 1\n");
            break;
        case '2':
            rTCNTB1 = 792;
            rTCMPB1 = 264;
            uart_printf("\n You selected 2\n");
            break;
        case '3':
            rTCNTB1 = 396;
            rTCMPB1 = 132;
            uart_printf("\n You selected 3\n");
            break;
        case '4':
            rTCNTB1 = 396;
            rTCMPB1 = 198;
            uart_printf("\n You selected 4\n");
    }
}

```

```

        break;
default:
    break;
}
}

/*****
* name:      timer_init
* func:      initialize PWM Timer0
* para:      none
* ret:       none
* modify:
* comment:
*****/
static void timer_init(void)
{
    rGPBCON = rGPBCON & 0xFFFFF0|(1<<3); // Enable TOUT1

    /* Set interrupt handler */
    pISR_TIMER1=(unsigned)timer1_int;

    /* Timer clock = PCLK/(prescaler value + 1)/16 */
    rTCFG0 = (39<<8)|(39);           // Prescaler value = 39
    rTCFG1 = (3<<4)|3;               // Divider value = 16

    rTCON = 0; //disable dead zone,auto reload off,inverter off,stop

    ClearPending(BIT_TIMER1);
}

/*****
* name:      pwm_test
* func:      initialize PWM Timer0
* para:      none
* ret:       none
* modify:
* comment:
*****/
void pwm_test(void)
{
    uart_printf(" Press SB1203 to change the direction of the motor\n");

    timer_init();

    /* Initialize PWM timer1 register */
    rTCNTB1 = 792;

```

```

rTCMPB1 = 396;
rTCON |= (1<<11)|(1<<10)|(1<<9)|(1<<8);
    // auto reload on,inverter on>manual update on,start

/* Clear manual update bit to start counting */
    rTCON &= ~(1<<10)|(1<<9);    // inverter off>manual update off

while(1)
{
    uart_printf("\nYou can press the following key to change the duty ratio:\n");
    uart_printf(" 1-->PWM1: 1/2,100Hz(default)\n");
    uart_printf(" 2-->PWM1: 1/3,100Hz\n");
    uart_printf(" 3-->PWM1: 1/3,200Hz\n");
    uart_printf(" 4-->PWM1: 1/2,200Hz\n");
    uart_printf(" Press SPACE to return!\n\n");

    while(!(rUTRSTAT0 & 0x1));    // if receive data ready on UART0

    Key_pressed = RdURXH0();        // or get value from UART0

    if( Key_pressed == ' ' )        // press SPACE to return
        break;

    if(( Key_pressed<'1' )||( Key_pressed>'4' ))
        uart_printf(" You have pressed an invalid key! Again please! \n");

    /* Enable timer1 interrupt */
    rINTMSK &= ~BIT_TIMER1;
}

rTCON = 0;    //disable dead zone,auto reload off,inverter off,stop
uart_printf(" end.\n");
}

```

7.2.7 练习题

1. 熟悉 S3C2410X 芯片的定时器 PWM 输出原理。
2. 改变定时器参数，产生其他频率的 PWM 输出控制电机。

第八章 高级应用实验

8.1 GPRS 模块控制实验

8.1.1 实验目的

- 通过实验掌握 ARM 处理器的 UART 控制方式和工作原理。
- 了解 GPRS 模块的使用方法，通过 GSM 收发短消息（SMS）
- 掌握简单 AT 命令集的使用

8.1.2 实验设备

- 硬件：Embest EduKit-III 实验平台，ULINK USB-JTAG 仿真器套件，PC 机，GPRS 子板，SIM 卡一张。
- 软件：μVision IDE for ARM 集成开发环境，Windows 98/2000/NT/XP。

8.1.3 实验内容

编写程序，实现：

改变 S3C2410X 和 GPRS 模块通信速率；

使用 GPRS 模块收发短信，阅读短信，删除短信；

8.1.4 实验原理

1. GPRS 模块概述

GPRS(General Packet Radio Service, 通用分组无线业务)是在现有 GSM 系统上发展起来的一种新的承载业务。基于这种业务的各种应用也蓬勃发展起来。典型的应用有：工业控制、环境保护、道路交通、网上银行、移动办公、零售服务、公安系统等。

GPRS 允许用户在端到端分组转义模式下发送和接收数据，而不需要利用电路交换的模式，比较适合于突发性的，频繁的，数据量小的数据传输，也适用于偶尔数据量大的数据传输。

目前，GPRS 模块的提供商有西门子、摩托罗拉、飞利浦、大唐、中兴、华为等。其中西门子和摩托罗拉公司的 GPRS 模块产品较为常见。

本实验使用的 GPRS 模块是西门子公司生产的 MC35i。该款 GPRS 模块具有很高的性能，可以广泛应用于以下场合：POS 终端、自动售货机、安全系统、远程遥测、交通控制、导航系统、手持设备、GPRS 调制解调器等。

模块特性：

双频段：EGSM900MHz 和 GSM1800MHz；

输出功率：4 类 EGSM 频段：2W，1 类 GSM 频段：1W；

使用 AT 命令控制；

SIM 应用工具包；

电源范围：3.3V—4.8V；

节电模式；

供电消耗：

—空闲模式：25mA

—通话模式：平均 300mA

—GPRS 模式：平均 360mA

—发送突发：最大 2.5A

—调电模式：50uA

—睡眠模式：最大 3.5mA

尺寸：54.5×36×3.6mm；

重量：9g；

工作温度：-20~55 度

模块接口：

40-pin ZIF 连接器

—电源供电：

—SIM 卡 3V 供电；

—RS232 双向数据总线；

—自适应波特率

—两路模拟音频信号接口

50ohm GSC 连接器；

要获得更多信息，请参考该模块的详细资料。

2. 标准 V.25ter AT 命令

AT 命令符合 ITU-T（国际电联）V.25ter 文件标准。

现简单介绍几个和收发短消息有关的几个 AT 命令。

AT 命令	功能描述
AT	与模块连接
AT+CMGF=?	返回当前的工作模式
AT+CMGF=n	设置当前工作模式，n=0：PDU 模式，n=1：text 模式
AT+CMGS	发送短信息
AT+CMGR=<index>	读短信息，其中 index 是消息在当前存储区中的序列号
AT+CMGL	输出短信息列表
AT+CMGD=<index>	删除短消息，删除当前存储区中序列号为 index 的短消息
AT+CNMI=<mode>	设置收到的短消息报告模式，mode=0：缓冲短消息结果码； mode=1：在数据通信状态下，阻止结果码发送到 TE()； Mode=2：无论何种状态下，都向 TE 发送结果码。

短消息发送过程:

1. 在超级终端窗口中输入 **AT+CMGF=1**, 设定模块工作模式为 **text** 模式;
2. 返回 **OK** 后, 输入 **AT+CMGS=13XXXXXXXXXX** 输入手机号, 按下回车;
3. 在 **>** 后输入所要发送的信息内容, 注意, 现在 **GPRS** 模块中, 不支持中文短信息发送, 只能发送 **ASCII** 码字符。

```
AT
OK
AT+CMGF=1
OK
AT+CMGS=13988888888
> HELLO,Welcom to Embest!
+CMGS: 206
OK
```

未读短消息接收过程:

1. 在超级终端窗口中输入 **AT+CMGF=1**, 设定当前工作模式为 **text** 模式;
2. 返回 **OK** 后, 输入 **AT+CMGL**, 按下回车;
3. 在超级终端窗口中可以显示当前存储区中短消息列表中未读的消息。

```
AT+CMGL
+CMGL: 6,"REC UNREAD", "+8613249808153", "05/04/29,
19:48:05+00"
Hello,Welcom to use Embest EduKit-III ! ←-收到的短消息内容
OK
```

读取单条短消息过程:

1. 在超级终端窗口中输入 **AT+CMGF=1**, 设定当前工作模式为 **text** 模式;
2. 返回 **OK** 后, 输入 **AT+CMGR=index**, 其中 **index** 为想要读取的目的消息在存储区中的序号, 按下回车;
3. 在超级终端窗口中可以显示当前存储区中的选定短消息。

```
at+cmgr=6
+CMGR: "REC READ", "+8613249808153", "05/04/29,
19:48:05+00"
Hello,Welcom to use Embest EduKit-III ! ←-收到的短消息内容
OK
```

短消息删除过程:

1. 在超级终端窗口中输入 AT+CMGD=index,其中 index 为要删除的短消息在存储区中的序号,并按下回车键。
2. 在超级终端窗口中显示 OK, 表明删除完成。

```
at+cmgd=6
```

```
OK
```

3. 电路连接图

本实验中总体电路连接图

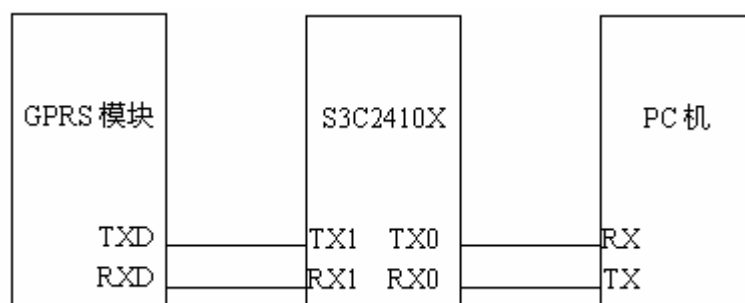


图 8-1-2 电路连接图

S3C2410X 处理器通过 UART0 接收 PC 机在输入超级终端输入的控制信息或着 AT 命令等, 并通过 UART1 控制 GPRS 模块, 使其工作在合适的模式, 并完成短消息的发送或者接收工作。

8.1.5 实验操作步骤



1. 准备实验环境

使用 ULINK USB-JTAG 仿真器连接目标板, 使用 Embest EduKit-III 实验板附带的串口线, 连接实验板上的 UART0 和 PC 机的串口。

2. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序 (波特率 115200、1 位停止位、无校验位、无硬件流控制); 或者使用其它串口通信程序。

3. 打开实验例程

- 1) 拷贝实验平台附带光盘 CD1\CD1_Basic_070615\Software 文件夹到 RealView MDK 软件的安装目录的 Keil\ARM\Boards\Embest\目录下 (如果已经拷贝, 可跳过此步骤);
- 2) 使用 μ Vision IDE for ARM 通过 ULINK USB-JTAG 仿真器连接实验板, 打开实验例程目录 8.1_gprs_test 子目录下的 gprs_test.Uv2 例程, 编译链接工程;
- 3) 根据 ReadMe 目录下的 ReadMeCommon.txt 及 readme.txt 文件配置集成开发环境 (工程默认已经配置正确), 点击工具栏 “”, 在 Option for Target 对话框的 Linker 页中选择 RuninRAM.sct 分散加载文件, 点击 MDK 的 Debug 菜单, 选择 Start/Stop Debug Session 项或点击工具栏 “”, 下载工程生成的.axf 文件到目标板的 RAM 中调试运行;
- 4) 如果需要将程序烧写固化到 Flash 中, 仅需要更改分散加载文件即可 (**慎用!!! 这一步的操作将会破坏 Flash 中原有固化程序, 建议实验中不操作**)。在 Option for Target 对话框的 Linker 页中选择 RuninFlash.sct 分散加载文件, 重新编译工程, 点击 MDK 的 Flash 菜单, 选择 Download 烧写调试代码到目标系统的 Nor Flash 中, 重启实验板, 实验板将会运行烧写到 Nor Flash 中的代码;

- 5) 点击 Debug 菜单的 Go 或 F5 键运行程序;
- 6) 程序正确运行后, 会在超级终端上输出如下信息:

```
boot success...

GPRS Modem Communication Test Example
Aug  4 2007 10:28:55
  0 --- 1200
  1 --- 2400
  2 --- 4800
  3 --- 9600

Baud rate select for GPRS modem: 3

AT command >
```

- 7) 使用 PC 机键盘, 输入数字 0~3 中的任意一个数字来设置和 GPRS 模块通信速率。
- 8) 输入 3 后, 选择波特率为 9.6kbps。
- 9) 再次点击 Debug 菜单 RUN 或 F5 键运行程序。
- 10) 根据前面的关于 AT 命令的介绍可以通过 GPRS 模块发送、接收、删除短消息。

4. 观察实验结果

在超级中断主窗口中输入 3, 可以观察到如下画面:

```
Apr 30 2005 09:35:19
  0 --- 1200
  1 --- 2400
  2 --- 4800
  3 --- 9600

Baud rate select for GPRS modem:
Baud rate select for GPRS modem: 3

AT command >AT
OK
```

5. 完成实验练习题

理解和掌握实验后, 完成实验练习题。

8.1.6 实验参考程序

```
void gps_test(void)
```



```

{
    char cInputChar;
    char nStrBuf[200];
    int  set_baud,i = 0;
    unsigned char * baud_sel[][2]={
        "0","1200",
        "1","2400",
        "2","4800",
        "3","9600",
        0,0};

    uart_change_baud(UART1,4800);
    while(1)
    {
        cInputChar = uart_tran();
        if (f_nUartSelect == UART0)
        {
#ifdef PRINT_UART0
            uart_select(0);                // send to user
            uart_sendbyte(cInputChar);
#endif
            uart_select(1);                // send to another uart
        }
        if (f_nUartSelect == UART1)
        {
            uart_select(0);                // send to another uart

            if (cInputChar == '$')
            {
                if(i != 0)
                {
                    if(!strncmp(nStrBuf,"$GPRMC", 6))    // Format: $GPRMC,DATA,...,DATA
                    {
                        gps_info(MSG_GPRMC,&nStrBuf[6]);
                    }
                }
                i = 0;
            }
            nStrBuf[i] = cInputChar;
            i++;
        }
        //uart_sendbyte(cInputChar);
    }
}

/*****
* name:      gps_info
* func:      report the info read from GPS module
*****/

```

```

* para:      nType:   input, GPS info type (defined)
              pPt:   input, string info pointer
* ret:      none
* modify:
* comment:
*****/

void gps_info(UINT32T nType, UINT8T *pPt)
{
    int i,err,nTime,nDate,nDotNum=0;
    char buf[16];

    switch(nType)
    {
    case MSG_GPRMC:
        while(*(pPt+1) != '*')
        {
            i = 0;
            while(*++pPt != ',')
                buf[i++] = *pPt;
            buf[i] = '\0';

            nDotNum++;
            switch(nDotNum)
            {
            case 1:                // 格林威治,063741.998: 6 时 37 分 41.998 秒
                nTime = ((buf[0]-0x30)*10+(buf[1]-0x30))<<16\
                    ((buf[2]-0x30)*10+(buf[3]-0x30))<<8\
                    (buf[4]-0x30)*10+(buf[5]-0x30);
                break;
            case 2:                // 信息有效(A)/无效(V)
                if(buf[0] != 'A')
                {
                    uart_printf(" InValid Message!");
                    err =1;
                }
                break;
            case 3:                // 2234.2551,N: 北纬 22.342551 度
                if(!err) uart_printf(" North Latitude: %s",buf);
                break;
            case 5:                // 11408.0338,E: 东经 114.080338 度
                if(!err) uart_printf(" East Longitude: %s\n",buf);
                break;
            case 9:                // 150805: 2005 年 8 月 15 日
                uart_printf(" 20%c%c-%c%c-%c%c", \
                    buf[4],buf[5], \
                    buf[2],buf[3], \

```

```

        buf[0],buf[1]);
    uart_printf(" %02d:%02d:%02d\n",(nTime>>16)&0xFF,(nTime>>8)&0xFF,(nTime)&0xFF);

    default:
        break;
    }
}
break;
case MSG_GPGGA:
    // add your code here
    break;
case MSG_GPGSA:
    // add your code here
    break;
case MSG_GPGSV:
    // add your code here
default:
    break;
}
}

/*****
* name:      uart_tran
* func:      Get a character from the uart
* para:      none
* ret:       get a char from uart channel
* modify:
* comment:
*****/

char uart_tran(void)
{
    while(1)
    {
        if(rUTRSTAT0 & 0x1)
        {
            f_nUartSelect = UART0;
            return RdURXH0();                // Receive data read
        }
        else if(rUTRSTAT1 & 0x1)
        {
            f_nUartSelect = UART1;
            return rURXH1();                // Receive data ready
        }
    }
}
}

```

8.1.7 练习题

1. 学习 GPRS 的 PDU 模式收发短信的原理。
2. 编写程序实现 GPRS 在 PDU 模式下收发短信。