



Cortex-A9

ARM 体系结构与接口技术

FS4412 部分

(V1.1)

华清远见教育集团●研发中心



170-9108-5953



2306275952/2668462267



<http://dev.hqyj.com>



support@farsight.com.cn

华清远见
dev.hqyj.com



前言

本书从嵌入式 ARM 体系结构的基础知识、系统环境搭建到综合应用，共分三个层次深入浅出地为读者拨开萦绕于嵌入式 ARM 体系结构这个概念的重重迷雾，引领读者渐渐步入嵌入式的世界，帮助探索者实实在在地把握第三次 IT 科技浪潮的方向。本书的特色如下：

- (1) 重基础，适合教学。
- (2) 重素质，全面讲解。本书在一般性教材的基础上，对嵌入式系统的软硬件开发环境进行了大量的讲解，可以让读者更进一步、更全面地了解嵌入式的开发过程。
- (3) 重实践，与实际项目相结合。本书每个原理讲解都对应着丰富的实验内容，并且原理讲解和实验内容一一对应，使得每个知识都可以被深刻的理解运用，本书附带的光盘中给出了参考设计代码和文档。
- (4) 重应用。书中的实例对时下经常使用的功能、设备、器材进行讲解和说明，力求教材所涉及的内容能紧跟行业实际应用的需要。

本书共分 14 个章节，每个章节下面包含一个对应此章节内容的实验。

第 1 章：嵌入式 ARM 技术概论。

第 2 章：ARM 微处理器指令系统。实验内容为：ADD 实验。

第 3 章：ARM 汇编语言程序设计。实验内容为：伪指令实验。

第 4 章：ARM 开发及环境搭建。实验内容为：eclipse for ARM 工具环境搭建。

第 5 章：GPIO 编程。实验内容为：GPIO 控制实验。

第 6 章：ARM 异常及中断处理。实验内容为：ARM 中断实验。

第 7 章：串行通讯接口。实验内容为：串口通信实验。

第 8 章：PWM 定时器。实验内容为：PWM 蜂鸣器实验。

第 9 章：看门狗定时器。实验内容为：看门狗 WDT 实验。

第 10 章：RTC 实时时钟。实验内容为：RTC 实时时钟实验。

第 11 章：A/D 转换器。实验内容为：A/D 转换实验。

第 12 章：I2C 接口。实验内容为：I2C 实验（E2PROM 读写实验）。

第 13 章：SPI 接口。实验内容为：SPI 实验（CAN 总线通讯实验）。

第 14 章：MMU 虚拟内存管理。实验内容为：MMU 实验。



目录

前言.....	I
目录.....	i
第 1 章 ARM 开发环境搭建.....	- 1 -
1.1 仿真器简介.....	- 1 -
1.1.1 FS-JTAG 仿真器介绍.....	- 1 -
1.2 开发环境搭建.....	- 2 -
1.2.1 FS-JTAG 相关工具安装.....	- 2 -
1.2.2 FS-JTAG 工具安装问题及解决方案.....	- 17 -
1.2.3 连接硬件平台.....	- 25 -
1.2.4 USB 转串口驱动安装.....	- 28 -
1.2.5 Putty 串口终端配置.....	- 29 -
1.3 Eclipse for ARM 使用.....	- 33 -
1.4 创建一个新工程.....	- 35 -
1.4.1 创建一个汇编工程.....	- 35 -
1.4.2 创建一个 C 工程.....	- 43 -
1.5 导入一个已有工程.....	- 59 -
1.6 调试工程.....	- 65 -
1.6.1 配置 FS-JTAG 调试工具.....	- 65 -
1.6.2 配置调试工具.....	- 65 -
1.6.3 查看变量及寄存器的方法.....	- 72 -
1.6.4 断点设置方法.....	- 73 -
1.6.5 查看内存数据信息方法.....	- 73 -
1.6.6 调试结束后的处理.....	- 73 -
1.7 64 位 eclipse 编译出错问题及解决.....	- 75 -
1.8 本章小结.....	- 77 -
1.9 练习题.....	- 77 -
第 2 章 嵌入式 ARM 技术概论.....	- 78 -
2.1 ARM 体系结构的技术特征及发展.....	- 78 -
2.1.1 ARM 公司简介.....	- 78 -
2.1.2 ARM 技术特征.....	- 79 -
2.1.3 ARM 体系架构的发展.....	- 79 -
2.2 ARM 微处理器简介.....	- 81 -
2.2.1 ARM9 处理器系列.....	- 82 -
2.2.2 ARM9E 处理器系列.....	- 82 -



2.2.3	ARM11 处理器系列	- 83 -
2.2.4	SecurCore 处理器系列	- 83 -
2.2.5	StrongARM 和 Xscale 处理器系列.....	- 83 -
2.2.6	MPCore 处理器系列.....	- 84 -
2.2.7	Cortex 处理器系列	- 84 -
2.2.8	最新 ARM 应用处理器发展现状	- 86 -
2.3	ARM 微处理器结构.....	- 87 -
2.4	ARM 微处理器的应用选型	- 87 -
2.4.1	ARM 芯片选择的一般原则	- 88 -
2.4.2	选择一款适合 ARM 教学的 CPU	- 88 -
2.5	Cortex-A9 内部功能及特点	- 91 -
2.6	数据类型	- 92 -
2.6.1	ARM 的基本数据类型	- 92 -
2.6.2	浮点数据类型	- 92 -
2.6.3	存储器大/小端	- 93 -
2.7	Cortex-A9 内核工作模式	- 93 -
2.8	Cortex-A9 存储系统	- 94 -
2.8.2	协处理器 (CP15)	- 95 -
2.8.3	存储管理单元 (MMU)	- 96 -
2.8.4	高速缓冲存储器 (Cache)	- 96 -
2.9	流水线.....	- 97 -
2.9.1	流水线的概念与原理	- 97 -
2.9.2	流水线的分类	- 97 -
2.9.3	影响流水线性能的因素	- 99 -
2.10	寄存器组织	- 99 -
2.11	程序状态寄存器	- 101 -
2.12	三星 Exynos4412 处理器介绍	- 104 -
2.13	FS4412 开发平台介绍.....	- 106 -
2.14	本章小结	- 112 -
2.15	练习题	- 112 -
第 3 章	ARM 微处理器指令系统	- 113 -
3.1	ARM 处理器的寻址方式	- 113 -
3.1.1	数据处理指令寻址方式	- 113 -
3.1.2	内存访问指令寻址方式	- 114 -
3.2	ARM 处理器的指令集	- 117 -



2.2.1	数据操作指令	- 117 -
3.2.2	指令举例如下。	- 118 -
3.2.3	乘法指令	- 123 -
3.2.4	Load/Store 指令	- 126 -
3.2.5	跳转指令	- 132 -
3.2.6	状态操作指令	- 135 -
3.2.7	协处理器指令	- 137 -
3.2.8	异常产生指令	- 140 -
3.2.9	其他指令介绍	- 141 -
3.3	ARM 汇编实验	- 143 -
3.3.1	实验目的	- 143 -
3.3.2	实验原理	- 143 -
3.3.3	实验内容	- 143 -
3.3.4	实验步骤	- 144 -
3.3.5	实验现象	- 145 -
3.4	本章小结	- 145 -
3.5	练习题	- 146 -
第 4 章	ARM 汇编语言程序设计	- 147 -
4.1	GNU ARM 汇编器支持的伪操作	- 147 -
4.1.1	伪操作概述	- 147 -
4.1.2	数据定义 (Data Definition) 伪操作	- 147 -
4.1.3	汇编控制伪操作	- 148 -
4.1.4	杂项伪操作	- 150 -
4.2	ARM 汇编器支持的伪指令	- 151 -
4.2.1	ADR 伪指令	- 151 -
4.2.2	ADRL 伪指令	- 152 -
4.2.3	LDR 伪指令	- 152 -
4.3	GNU ARM 汇编语言的语句格式	- 154 -
4.4	ARM 汇编语言的程序结构	- 155 -
4.4.1	汇编语言的程序格式	- 155 -
4.4.2	汇编语言子程序调用	- 156 -
4.4.3	过程调用标准 AAPCS	- 157 -
4.4.4	汇编语言程序设计举例	- 158 -
4.5	汇编语言与 C 语言的混合编程	- 159 -
4.5.1	GNU ARM 内联汇编	- 159 -



4.5.2	混合编程调用举例	- 162 -
4.6	ARM 伪指令实验	- 163 -
4.6.1	实验目的	- 163 -
4.6.2	实验原理	- 163 -
4.6.3	实验内容	- 163 -
4.6.4	实验步骤	- 164 -
4.6.5	实验现象	- 164 -
4.7	本章小结	- 165 -
4.8	思考题	- 165 -
第 5 章	GPIO 编程	- 166 -
5.1	GPIO 功能介绍	- 166 -
5.2	Exynos4412 芯片的 GPIO 控制器详解	- 166 -
5.2.1	特性	- 166 -
5.2.2	GPIO 分组预览	- 166 -
5.2.3	Exynos4412 的 GPIO 常用寄存器分类	- 167 -
5.2.4	GPIO 功能描述	- 167 -
5.2.5	Exynos4412 I/O 接口常用寄存器详解	- 168 -
5.2.6	GPIO 数据寄存器	- 168 -
5.3	GPIO 控制实验	- 168 -
5.3.1	实验目的	- 168 -
5.3.2	实验原理	- 168 -
5.3.3	实验内容	- 170 -
5.3.4	实验步骤	- 172 -
5.3.5	实验现象	- 172 -
第 6 章	ARM 异常及中断处理	- 173 -
6.1	ARM 异常中断处理概述	- 173 -
6.2	ARM 体系异常种类	- 174 -
6.3	ARM 异常响应和处理程序返回	- 180 -
6.3.1	中断响应的概念	- 180 -
6.3.2	ARM 异常响应流程	- 180 -
6.3.3	从异常处理程序中返回	- 181 -
6.4	ARM 的 SWI 异常中断处理程序设计	- 183 -
6.5	FIQ 和 IRQ 中断	- 185 -
6.5.1	中断分支	- 185 -
6.6	Exynos4412 中断机制分析	- 186 -



6.6.1	Exynos4412 中断概述	186 -
6.6.2	EXYNOS4412 中断控制	187 -
6.7	ARM 中断实验	195 -
6.7.1	实验目的	195 -
6.7.2	实验原理	195 -
6.7.3	实验内容	196 -
6.7.4	实验步骤	200 -
6.7.5	实验现象	200 -
6.8	本章小结	201 -
6.9	练习题	201 -
第 7 章	串行通讯接口	202 -
7.1	串行通信概述	202 -
7.1.1	串行通信与并行通信概念	202 -
7.1.2	异步串行方式的特点	202 -
7.1.3	异步串行方式的数据格式	202 -
7.1.4	同步串行方式的特点	203 -
7.1.5	同步串行方式的数据格式	203 -
7.1.6	比特率、比特率因子与位周期	203 -
7.1.7	RS-232C 串口规范	203 -
7.1.8	RS-232C 接线方式	205 -
7.2	EXYNOS4412 异步串行通信	206 -
7.2.1	EXYNOS4412 串口控制器概述	206 -
7.2.2	UART 寄存器详解	207 -
7.3	串口通信实验	212 -
7.3.1	实验目的	212 -
7.3.2	实验原理	212 -
7.3.3	实验内容	214 -
7.3.4	实验步骤	216 -
7.3.5	实验现象	216 -
7.4	本章小结	217 -
7.5	练习题	217 -
第 8 章	PWM 定时器	218 -
8.1	PWM 定时器概述	218 -
8.2	PWM 定时器特点	218 -
8.3	PWM 定时器的寄存器	219 -



8.4	PWM 定时器操作示例.....	- 223 -
8.5	实验 PWM 蜂鸣器实验.....	- 224 -
8.5.1	实验目的	- 224 -
8.5.2	实验原理	- 225 -
8.5.3	实验内容	- 225 -
8.5.4	实验步骤	- 227 -
8.5.5	实验现象	- 228 -
8.6	本章小结	- 228 -
8.7	练习题	- 228 -
第 9 章	看门狗定时器	- 229 -
9.1	EXYNOS4412 看门狗定时器概述	- 229 -
9.2	看门狗定时器寄存器	- 230 -
9.3	看门狗 WDT 实验	- 231 -
9.3.1	实验目的	- 231 -
9.3.2	实验原理	- 231 -
9.3.3	实验内容	- 231 -
9.3.4	实验步骤	- 236 -
9.3.5	实验现象	- 236 -
9.4	本章小结	- 238 -
9.5	练习题	- 238 -
第 10 章	RTC 实时时钟	- 239 -
10.1	RTC 介绍	- 239 -
10.2	RTC 控制器	- 239 -
10.3	RTC 控制器寄存器详解	- 240 -
10.4	实时时钟 RTC 实验	- 241 -
10.4.1	实验目的	- 241 -
10.4.2	实验原理	- 241 -
10.4.3	实验内容	- 241 -
10.4.4	实验步骤	- 246 -
10.4.5	实验现象	- 247 -
10.5	练习题	- 248 -
第 11 章	A/D 转换器	- 249 -
11.1	A/D 转换器原理	- 249 -
11.1.1	A/D 转换基础	- 249 -
11.1.2	A/D 转换的技术指标	- 249 -



11.1.3	A/D 转换器类型	250 -
11.1.4	A/D 转换的一般步骤	254 -
11.2	EXYNOS4412 A/D 转换器	254 -
11.2.1	EXYNOS4412 A/D 转换器概述	254 -
11.2.2	EXYNOS4412 A/D 控制器寄存器	256 -
11.3	A/D 实验	257 -
11.3.1	实验目的	257 -
11.3.2	实验原理	257 -
11.3.3	实验内容	257 -
11.3.4	实验步骤	259 -
11.3.5	实验现象	259 -
11.4	本章小结	260 -
11.5	练习题	260 -
第 12 章	I2C 接口	261 -
12.1	I2C 总线	261 -
12.1.1	I2C 总线介绍	261 -
12.1.2	I2C 总线术语	261 -
12.1.3	I2C 总线位传输	261 -
12.1.4	I2C 总线数据传输	262 -
12.1.5	I2C 总线寻址方式	263 -
12.1.6	快速和高速模式	263 -
12.2	I2C 总线控制器	264 -
12.2.1	EXYNOS4412 下的 I2C 控制器介绍	264 -
12.2.2	I2C 总线控制寄存器详解	264 -
12.3	I2C 重力感应/陀螺仪实验	266 -
12.3.1	实验目的	266 -
12.3.2	实验原理	266 -
12.3.3	实验内容	268 -
12.3.4	实验步骤	271 -
12.3.5	实验现象	272 -
第 13 章	SPI 接口	273 -
13.1	SPI 总线协议理论	273 -
13.1.1	协议简介	273 -
13.1.2	协议内容	273 -
13.2	SPI 控制器详解	275 -



13.2.1	EXYNOS4412 的 SPI 控制器简介	- 275 -
13.2.2	时钟源控制	- 275 -
13.2.3	寄存器详解	- 276 -
13.3	SPI/CAN 总线实验	- 277 -
13.3.1	实验目的	- 277 -
13.3.2	实验原理	- 277 -
13.3.3	实验内容	- 279 -
13.3.4	实验步骤	- 286 -
13.3.5	实验现象	- 287 -
13.4	本章小结	- 288 -
13.5	练习题	- 288 -
第 14 章	MMU 虚拟内存管理	- 289 -
14.1	什么是 MMU	- 289 -
14.2	MMU 作用	- 289 -
14.3	MMU 属性	- 290 -
14.4	MMU 实验	- 294 -
14.4.1	实验目的	- 294 -
14.4.2	实验原理	- 294 -
14.4.3	实验内容	- 296 -
14.4.4	实验步骤	- 298 -
14.4.5	实验现象	- 299 -



华清远见
dev.hqyj.com



第 1 章 ARM 开发环境搭建

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

本章主要介绍它的使用、配置方法，内容主要有：

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

1.1 仿真器简介

1.1.1 FS-JTAG 仿真器介绍

FS_JTAG ARM 仿真器是继华清远见研发中心自主研发的 Cortex 系列开发平台，及配套案例资料和技术支持获得业内合作企业及参训学员的一致好评之后，经过几个月的潜心研究和专注努力，由华清远见研发中心自主研发 Cortex-A 系列 ARM 仿真器。这无疑为业内合作企业、合作院校及广大培训学员带来了非常好的消息。

了解行业和相关技术的人都知道，功能完善的 ARM 仿真器和软件调试环境对于学习 ARM 处理器的工作原理和核心知识来说至关重要。由于之前多年的技术发展和行业实践，针对 Cortex-M 系列、ARM7、ARM9、以及 ARM11 系列处理器，市场上都已经有很多成熟的、价廉物美的仿真器可供选择。而对于目前最新流行的 ARM 应用处理器 Cortex-A 系列来说，业内的技术工程师们却很难找到价格合适，功能完善的仿真器，国外动辄几千甚至上万的价格，让很多人感叹之余，只能望而却步。

华清远见研发中心为了推进 Cortex-A 系列 ARM 处理器的教学，提高合作企业及合作院校广大技术爱好者和培训学员的学习效率，最新生产研发出 FS-JTAG 仿真器，该款仿真器可以仿真 Cortex-M3、ARM7、ARM9、ARM11、Cortex-A8、Cortex-A9 等多个 ARM 处理器系列。同时，华清远见研发中心也在 FS-S5PC100、FS210、FS4412 教学平台上使用 FS-JTAG 环境开发了全套的裸机接口代码，再配合研发中心出版的图书及实验指导书，相信一定会成为业内最实用的 ARM 处理器配套教学平台和辅助学习资料。



图 FS-JTAG 仿真器

1.2 开发环境搭建

Eclipse for ARM 是借用开源软件 Eclipse 的工程管理工具，嵌入 GNU 工具集，使之能够开发 ARM 公司 Cortex-A 系列的 CPU，这里使用 Eclipse for ARM 作为开发软件。在开发箱中的配套光盘中，打开 FS-JTAG 这个目录，可以看如图所示的光盘资料。Eclipse for ARM 开发工具所在光盘路径：**【华清远见-CORTEXA9 资料\工具软件\Windows\FS-JTAG】**

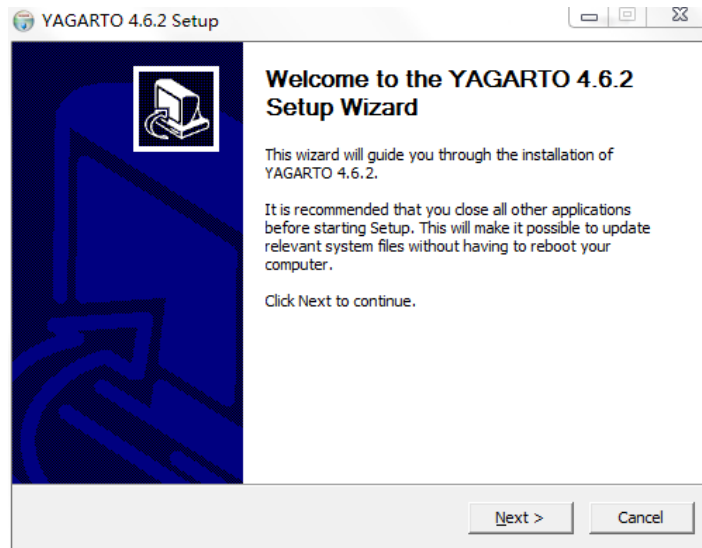


图 光盘资料

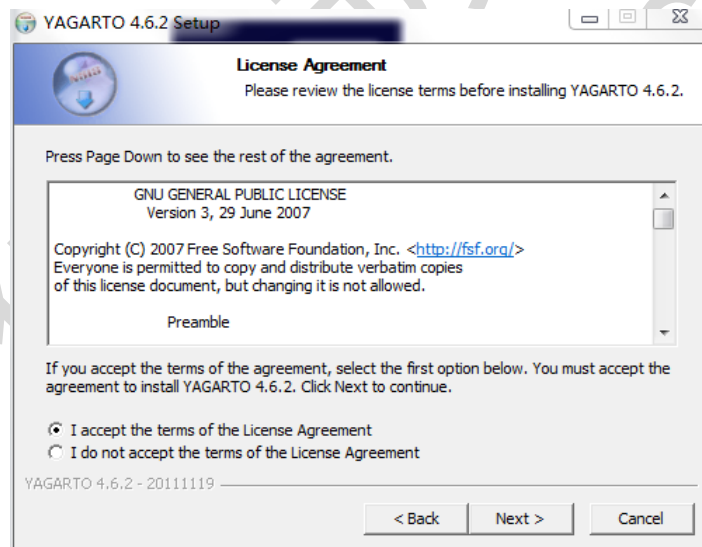
1.2.1 FS-JTAG 相关工具安装

(1) 安装 GCC 编译工具

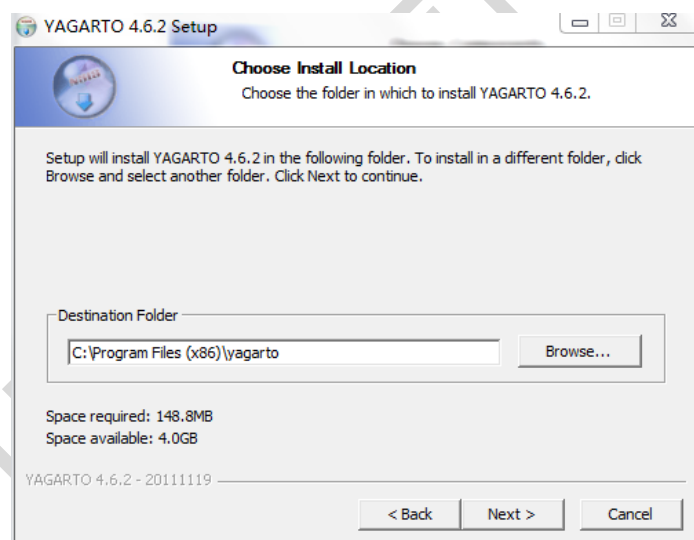
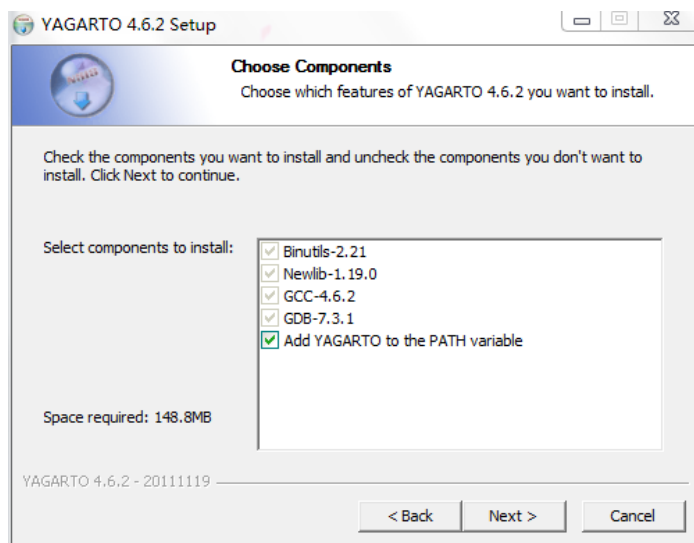
双击安装**【华清远见-CORTEXA9 资料\工具软件\Windows\FS-JTAG\Yagarto 工具包】**目录下的文件：
yagarto-bu-2.21_gcc-4.6.2-c-c++_nl-1.19.0_gdb-7.3.1_eabi_20111119.exe



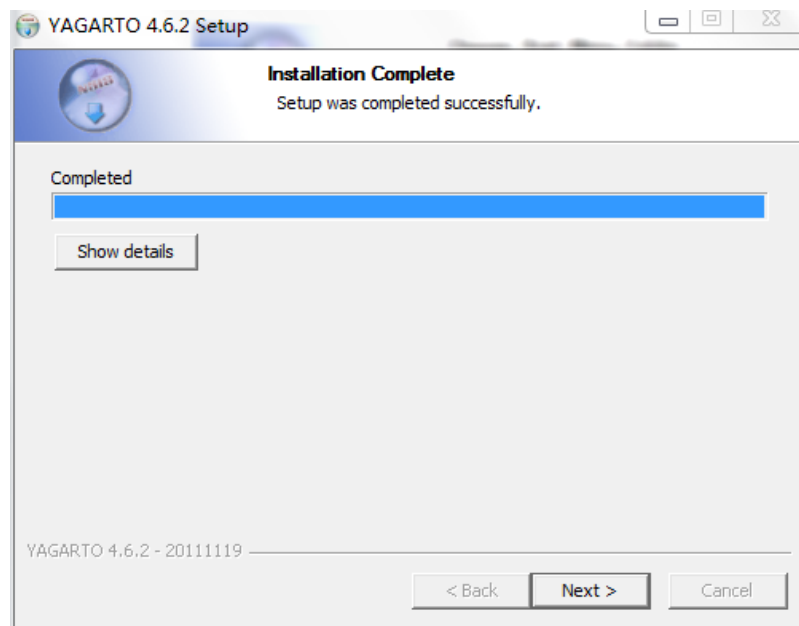
点击 Next



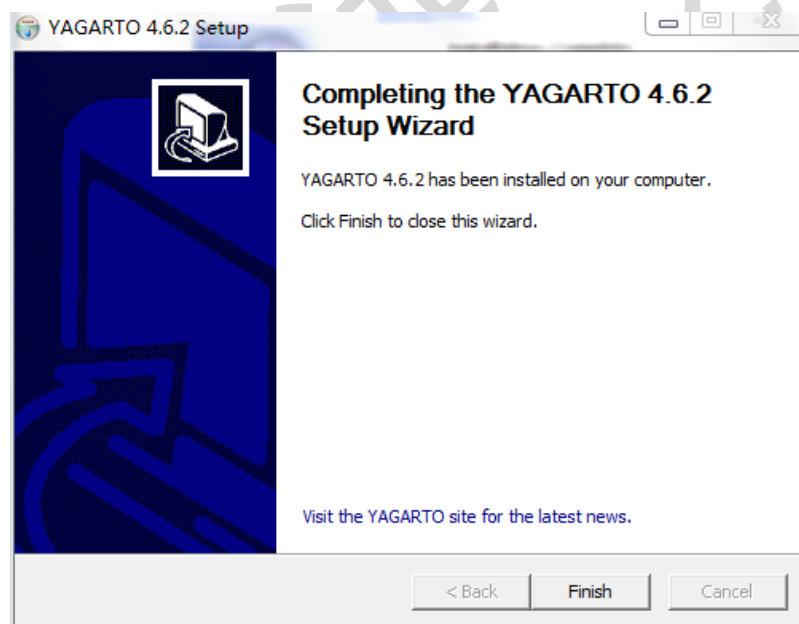
选择 “I accept the terms of the License Agreement” 然后点击 Next



点击 **Install**



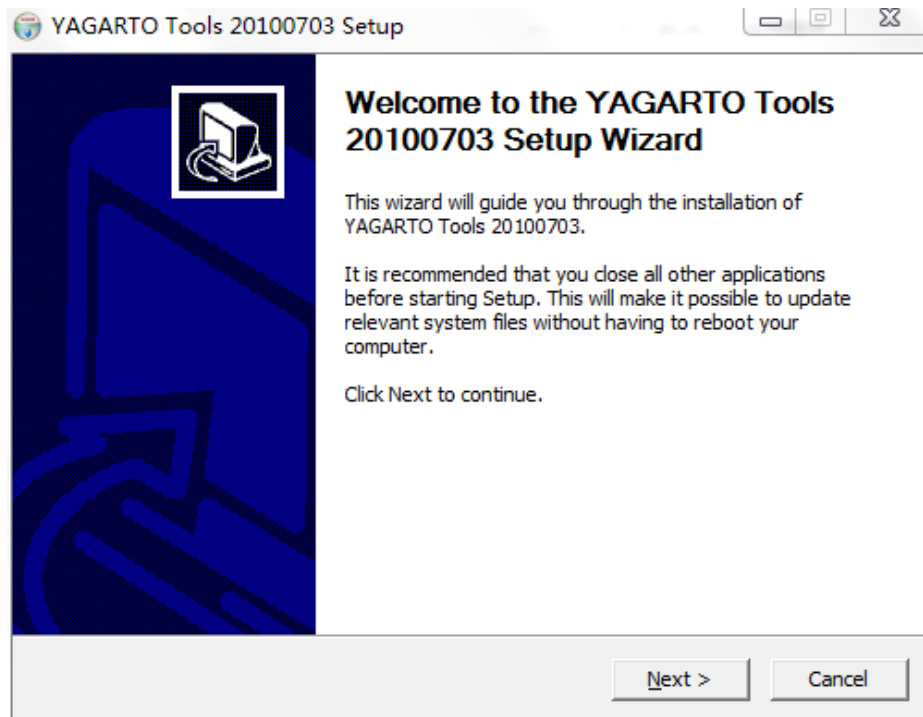
点击 **Next**



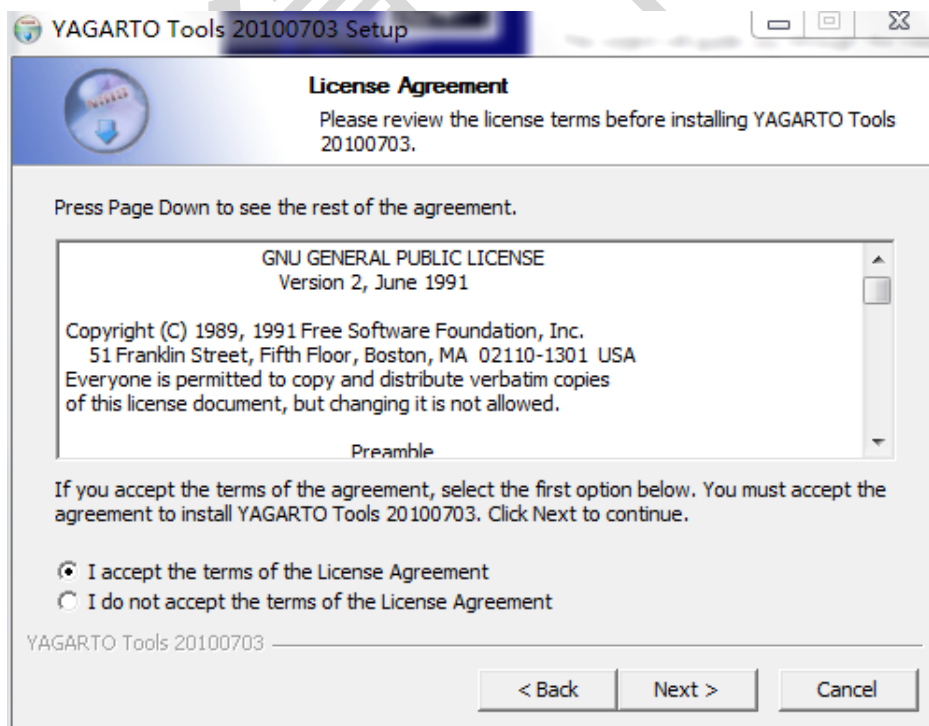
点击 **Finish** 完成安装

(2) 安装 Yagarto 工具包

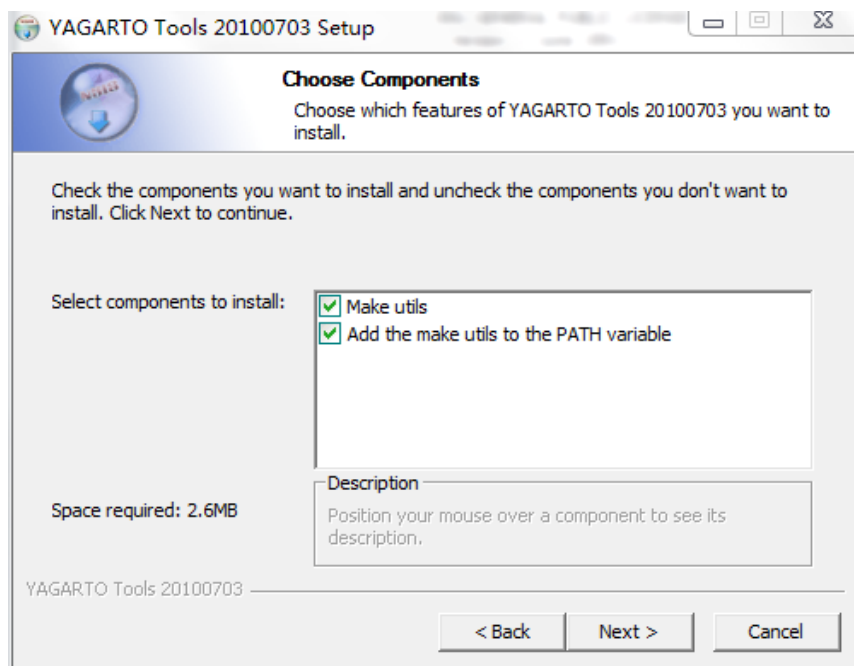
双击安装【华清远见-CORTEXA9 资料\工具软件\Windows\F5-JTAG\Yagarto 工具包】目录下的文件：
yagarto-tools-20100703-setup.exe



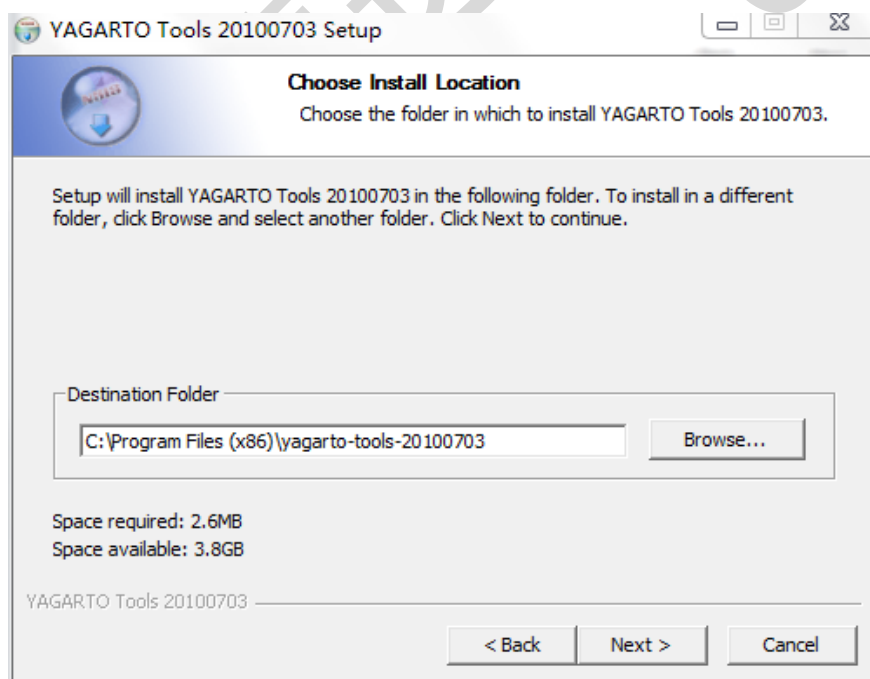
点击 Next



选择 “I accept the terms of the License Agreement” 然后点击 Next



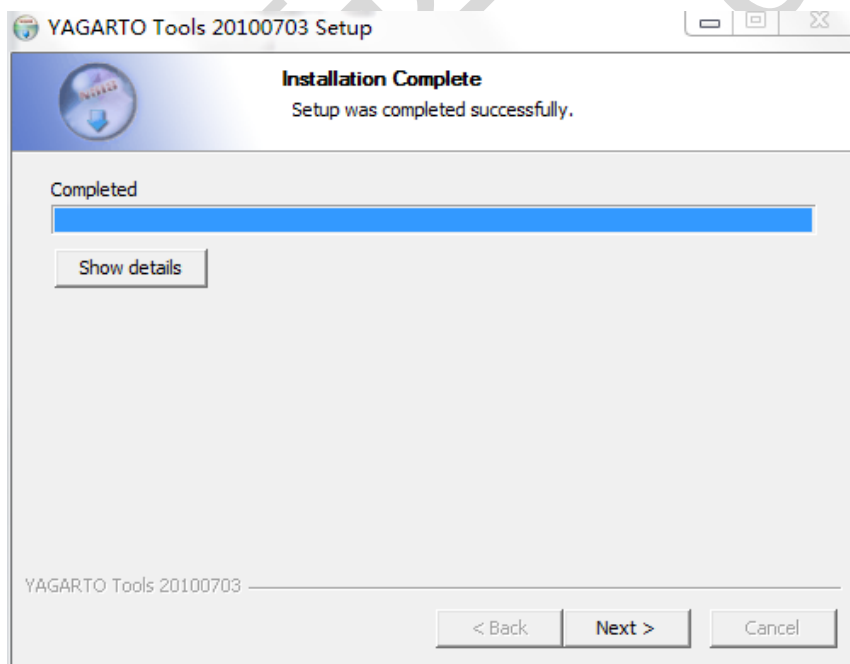
点击 Next



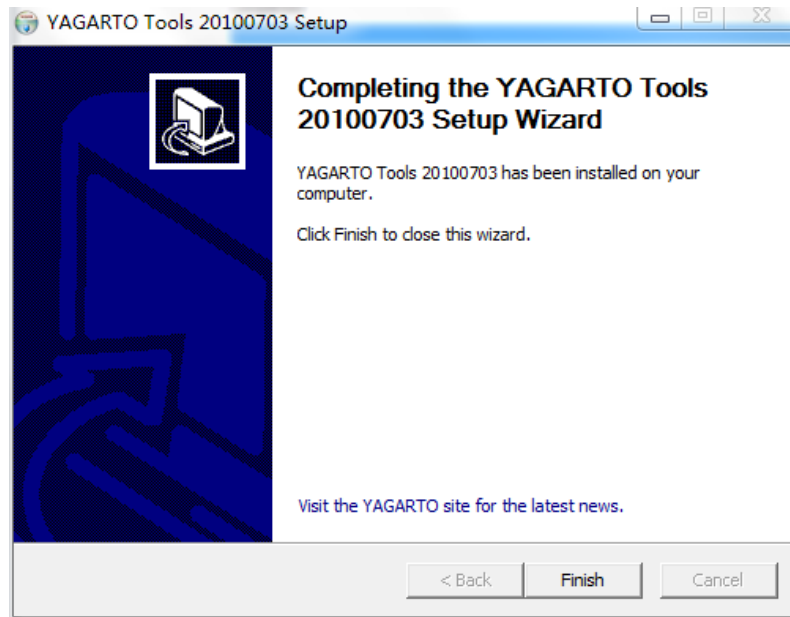
选择安装路径点击 Next



点击 **Install** 安装



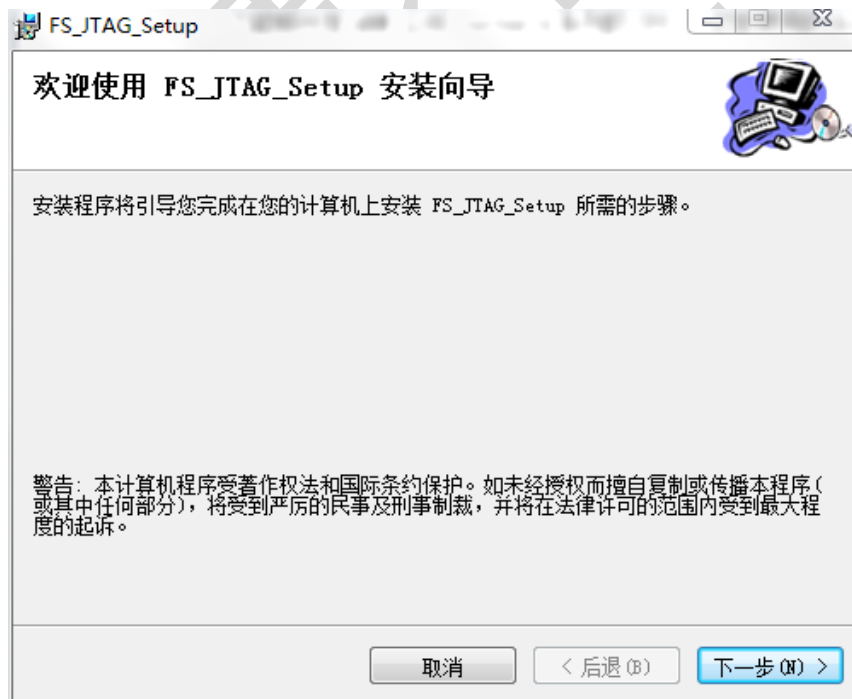
点击 **Next**



点击 **Finish** 完成安装

(3) 安装 FS-JTAG 调试软件

双击“FS-JTAG 安装包”下的 setup.exe 安装 FS-JTAG 工具。



点击 **“下一步”**



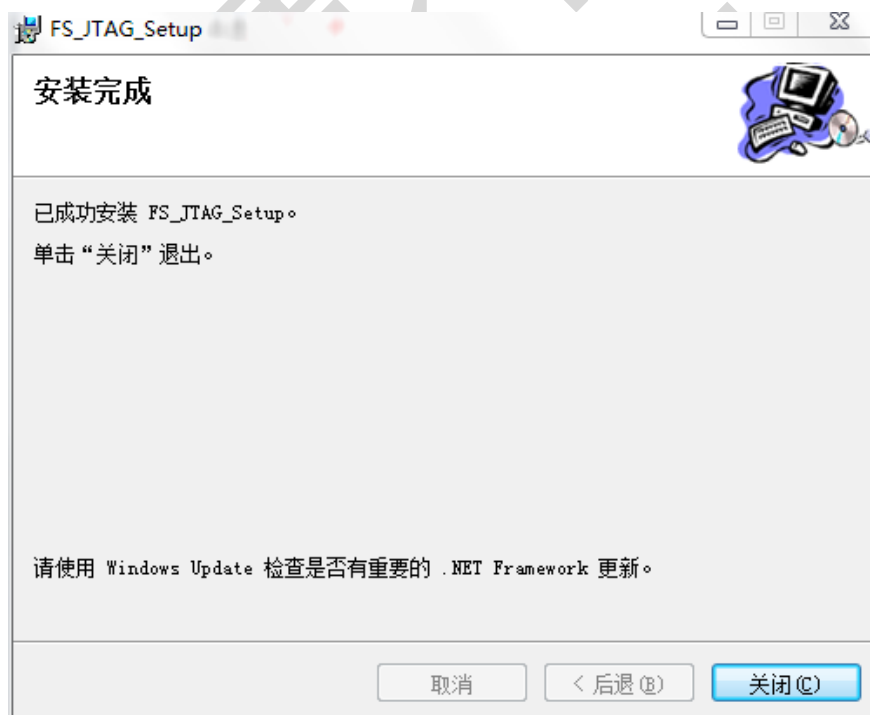
选择安装路径然后点击“**下一步**”



点击“**下一步**”



等待安装完成



点击“关闭”完成安装

注意:如果是 Windows xp 在安装过程中可能出现缺少插件提示,需按提示安装“FS-JTAG\FX-JTAG 安装包”目录下“DotNetFX”和“WindowsInstaller3_1”中相关文件

(4) 安装 JRE



双击安装【华清远见-CORTEXA9 资料\工具软件\Windows\Fs-JTAG\JRE】录下的文件：
jre-6u7-windows-i586-p-s.exe



点击“接受”这个过程可能需要几分钟；



点击“完成”完成安装

(5) 安装 FS-JTAG 驱动

XP/Win7 驱动路径：【华清-CORTEXA9 资料\工具软件\Windows\Fs-JTAG\DRIVER\Windows】

Win8 及以上驱动路径：【华清远见-CORTEXA9 资料\工具软件\Windows\Fs-JTAG\DRIVER\Win8.1】

注意：这里面含有 64 位和 32 位的驱动，amd64 对应 64 位，i386 对用 32 位。



- Windows xp 系统在安装驱动时略有不同，把 FS-JTAG 接入计算机 USB 口，会提示发现新硬件（如图所示），选择从列表或指定位置安装，然后单击“下一步”按钮：

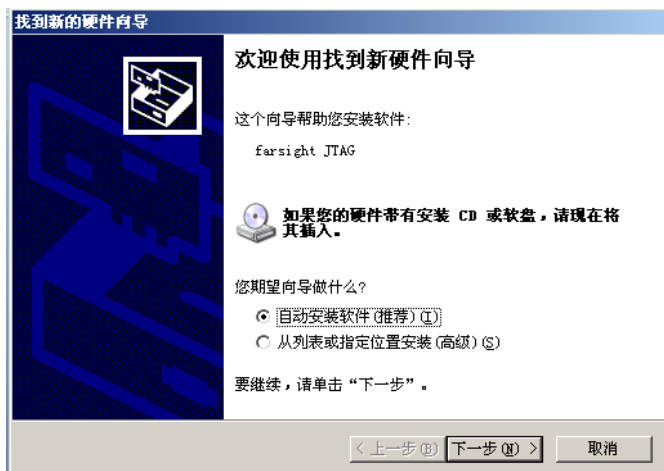


图 安装驱动界面

选择“从列表或指定位置安装”，单击“下一步”按钮后会出现选择驱动安装目录（如图左所示），单击“浏览”按钮找到 DRIVER 所在的目录，如图右所示。

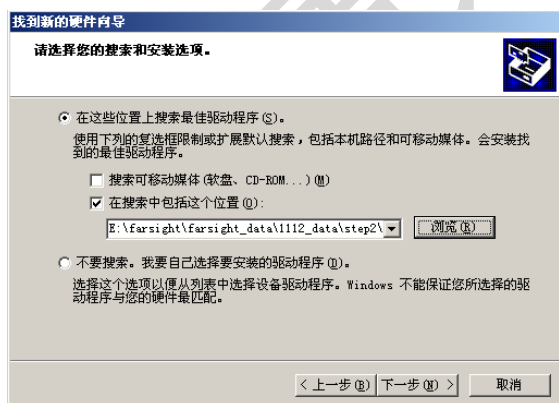


图 硬件向导

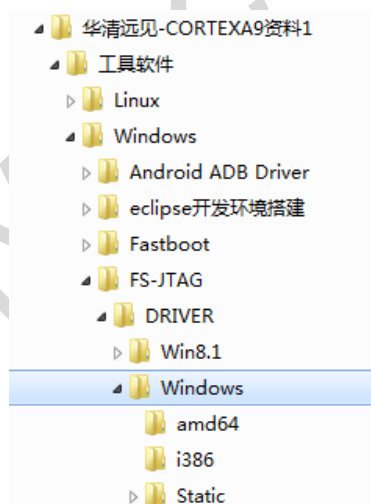


图 选择驱动文件目录

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

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

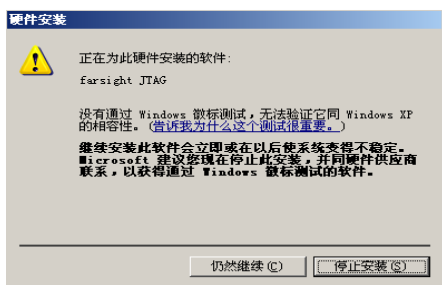


图 提示信息



图 找到 ftdibus.sys 文件



注意：例如你的是 XP 32 位系统，驱动路径为：【华清远见-CORTEXA9 资料\工具软件\Windows\FS-JTAG\DRIVER\Windows\i386】（64 位的在 amd64 目录里）。

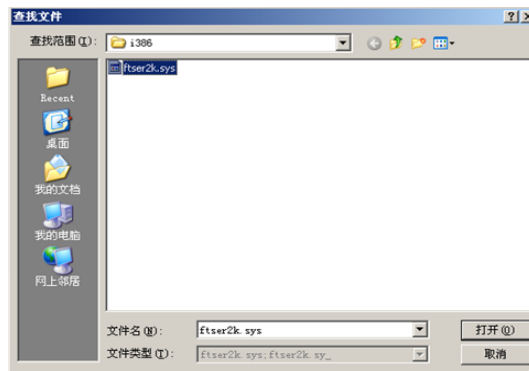


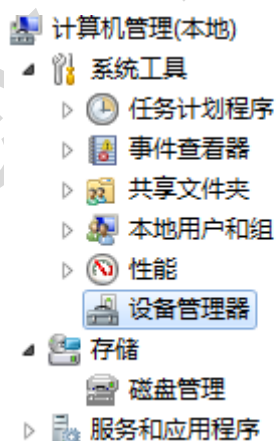
图 找到 USB 目录

- Windows 7 及以上版本驱动安装步骤：

将 FS-JTAG 通过 USB 线与 PC 连接，右键点击“我的电脑”选择“管理”



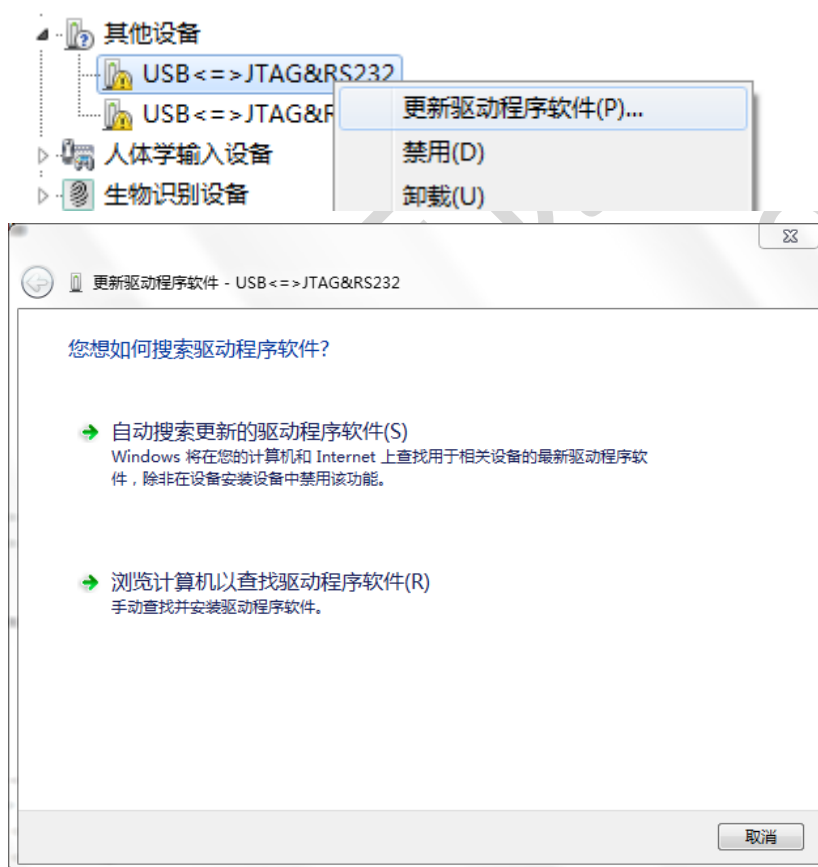
在左侧栏里选择“设备管理”



显示如下：



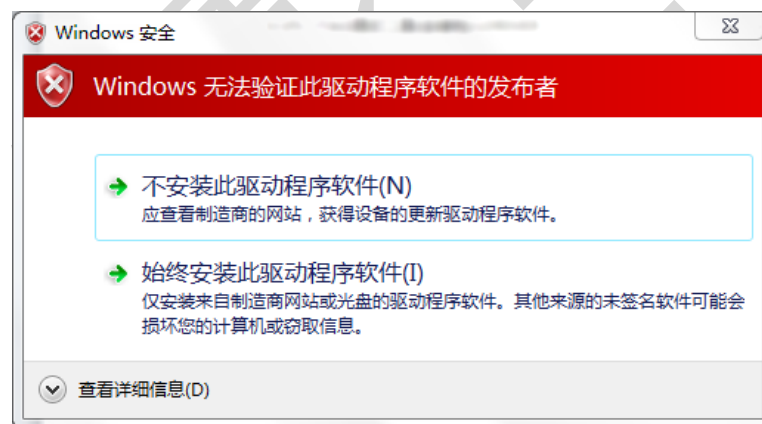
右键点击选择“更新驱动”



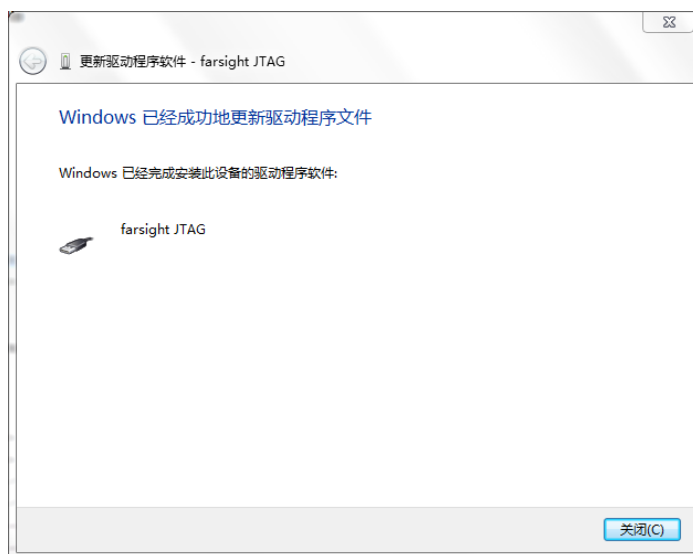
选择“浏览计算机以查询驱动程序软件(R)”；



点击浏览选择【FS-JTAG 调试工具(安装包)\DRIVER】目录主要“包括子文件夹”必须选择，点击“下一步”。



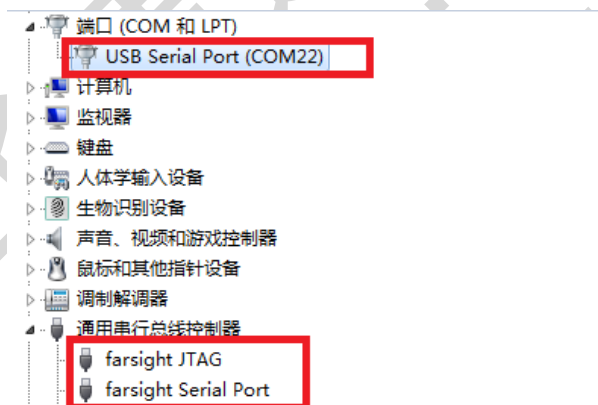
安装过程出现上图提示，点击“始终安装此驱动程序软件(I)”继续安装。



点击“关闭”完成安装

注意：此安装过程需要进行3次，直到设备管理器中没有叹号标记或未知设备。

这是设备管理器中会出现如下选项：如果下面选项没有全部出现，右键点击有黄色叹号的选项更新驱动，过程同上。



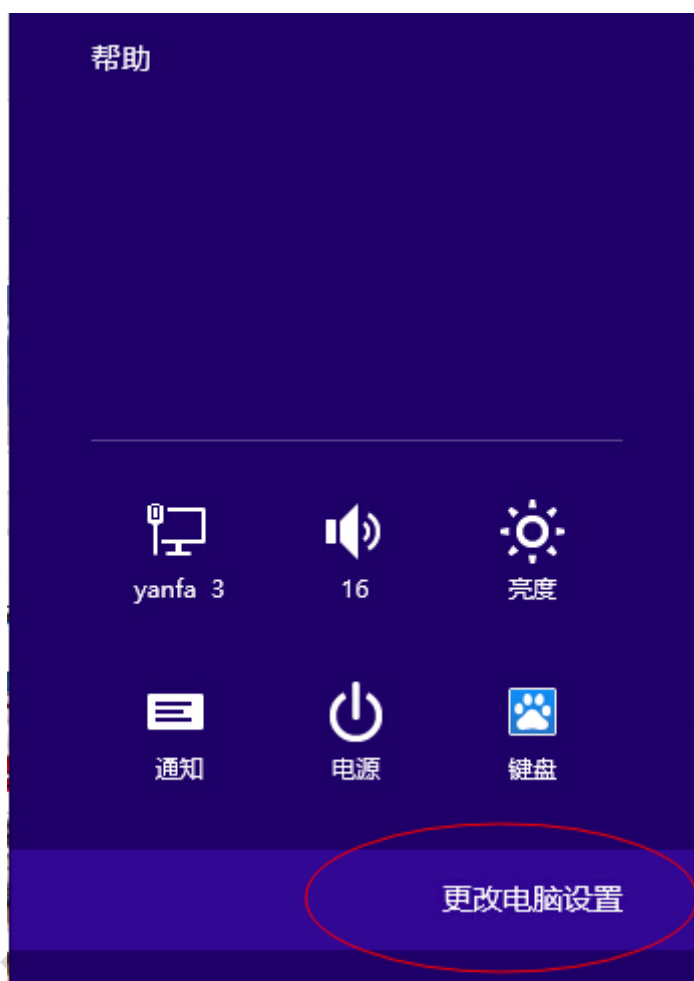
1.2.2 FS-JTAG 工具安装问题及解决方案

在搭建 Win8.1 及以上系统版本过程中会出现如下问题，此处给出解决方案。

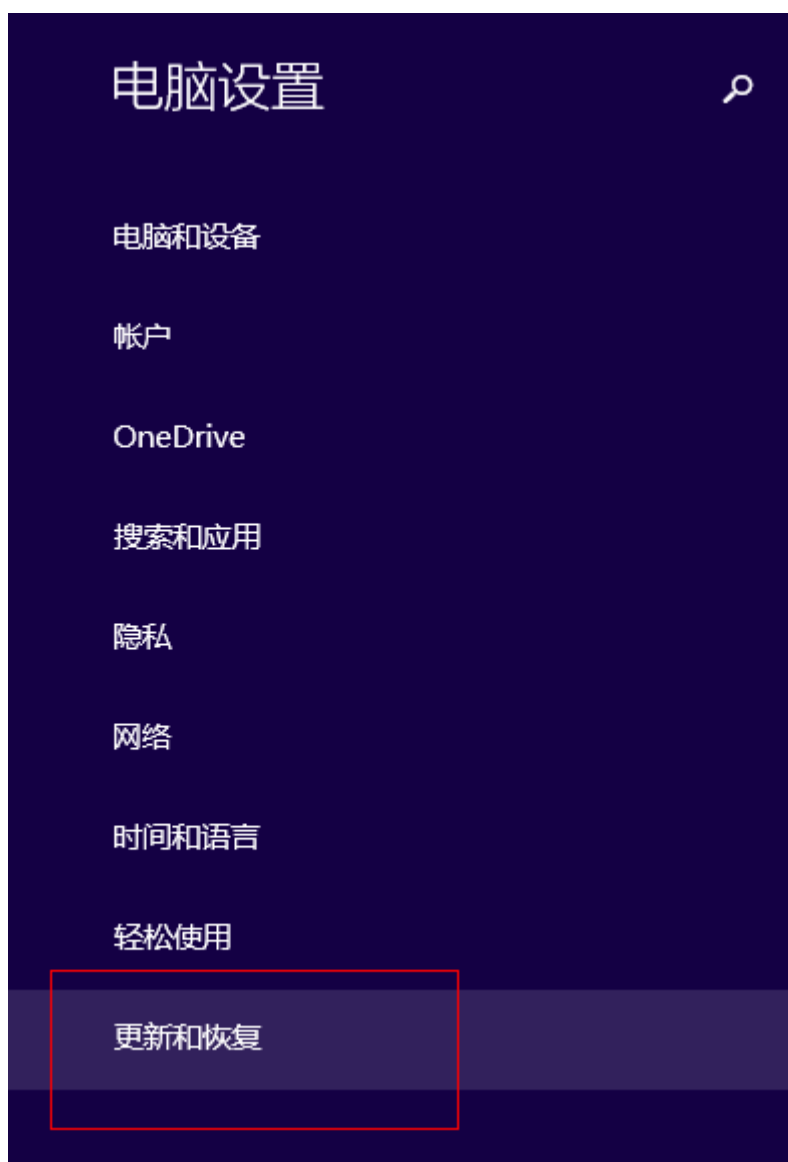
问题一：Win8.1/10 安装驱动出现“文件的哈希值不在指定的目录”，出现该问题的原因是 Win8.1/10 版本系统与设备驱动不兼容，没有得到数字签名通过。需要进入“高级启动”模式，此时 Win8.1 和 Win10 略有不同。

- Win8.1 进入“高级启动”模式：

(1) 鼠标移到右下角，点击“设置”，再点击“更改电脑设置”



(2) 点击最后一个“更新和恢复”，再点击“恢复”



(3) 点击“恢复”之后，在右边点击高级启动下面的“重新启动”



恢复电脑而不影响你的文件

如果你的电脑无法正常运行，你可以在不丢失照片、音乐、视频和其他个人文件的情况下对它进行恢复。

开始

删除所有内容并重新安装 Windows

如果要回收你的电脑或完全重新使用，可以将其初始化为出厂设置。

开始

高级启动

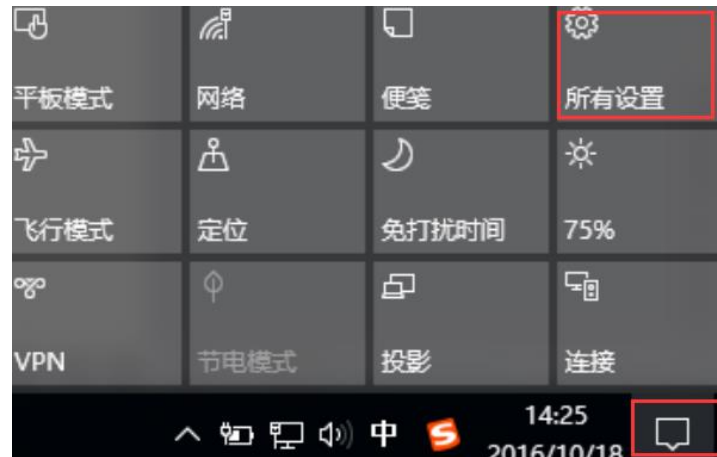
通过设备或磁盘(如 U 盘或 DVD)启动，更改 Windows 启动设置，或者从系统映像还原 Windows。这将重新启动电脑。

立即重启



- Win10 进入“高级启动”模式：

(1) 将鼠标移动到右下角，点击“所有设置”，弹出 Windows 设置窗口。



(2) 在窗口输入框内输入“更改高级启动设置”，回车即可。



(3) 选择红色方框内的“恢复”并点击“立即重启”。



进入“高级启动”模式后 Win8.1 和 Win10 操作相同：

- (1) 点击“立即启动”后会弹出如下窗口，点击“疑难解答”

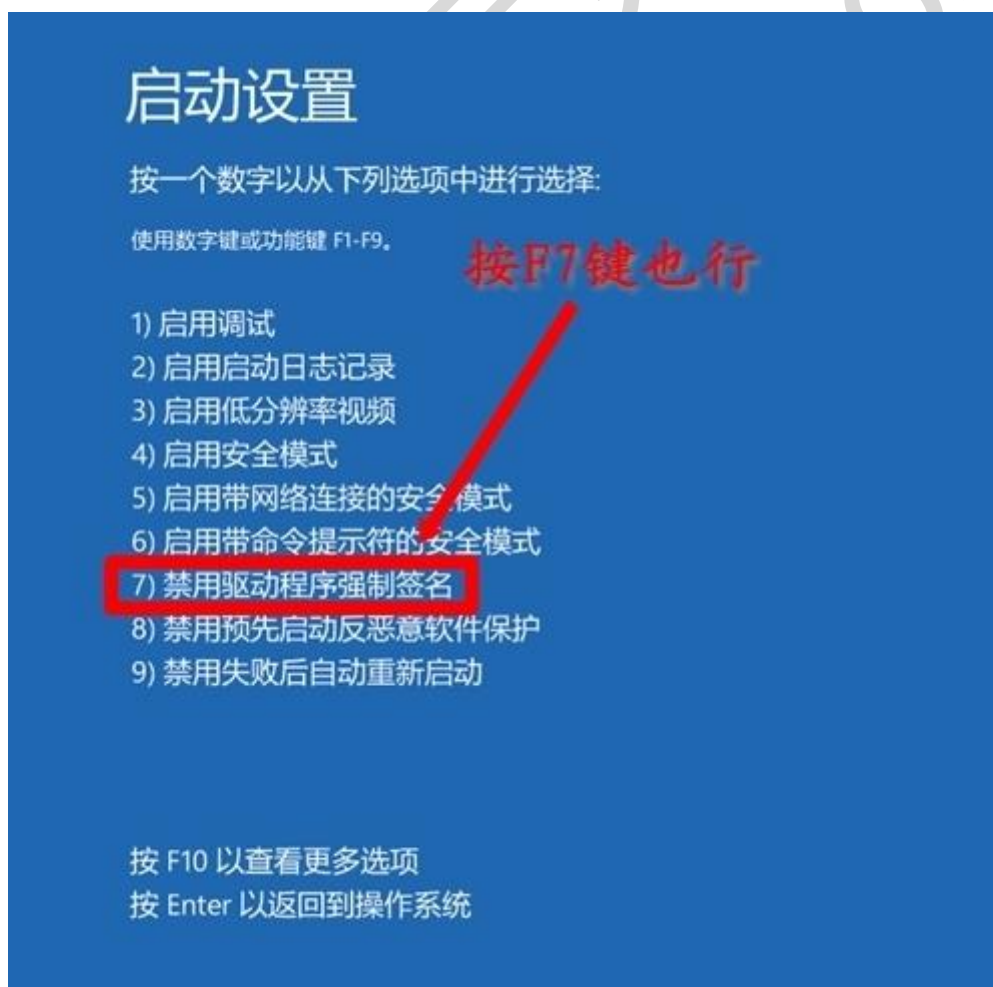


- (2) 点击高级，启动设置，重启

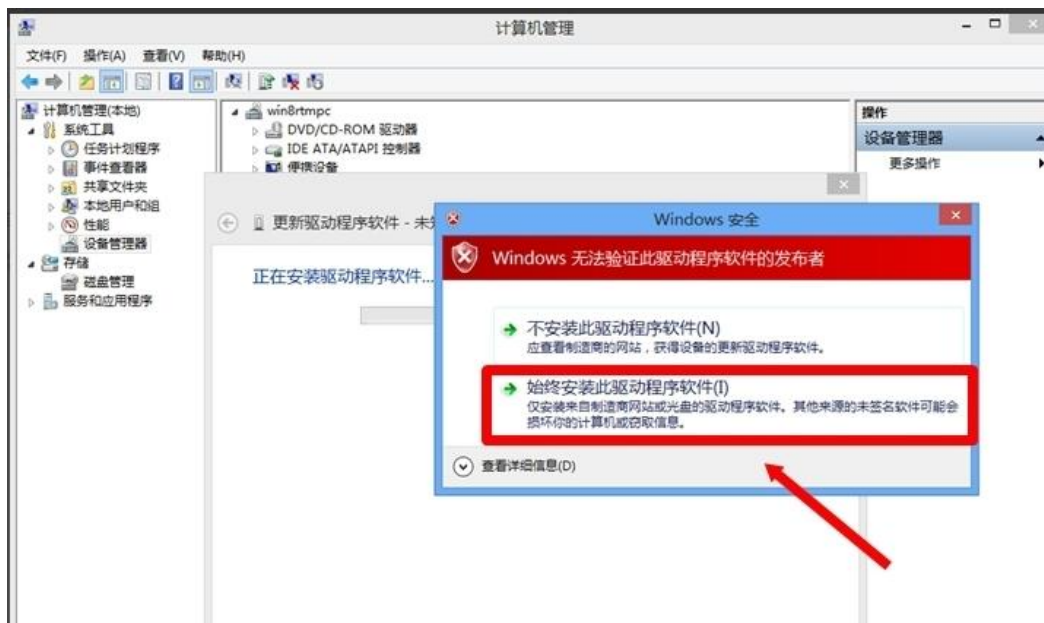




(3) 这会重启之后就跳出来安全模式等列表了，选择倒数第三个，禁用强制驱动程序签名，对应哪个数字就按那个数字。

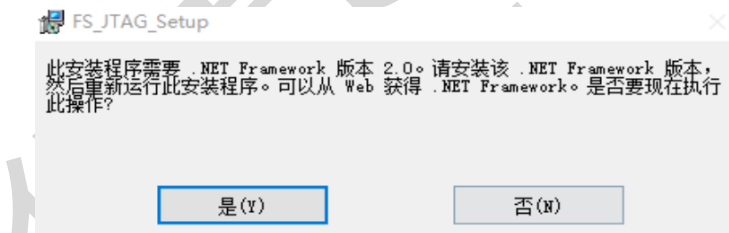


(4) 重启，驱动就可以成功安装了



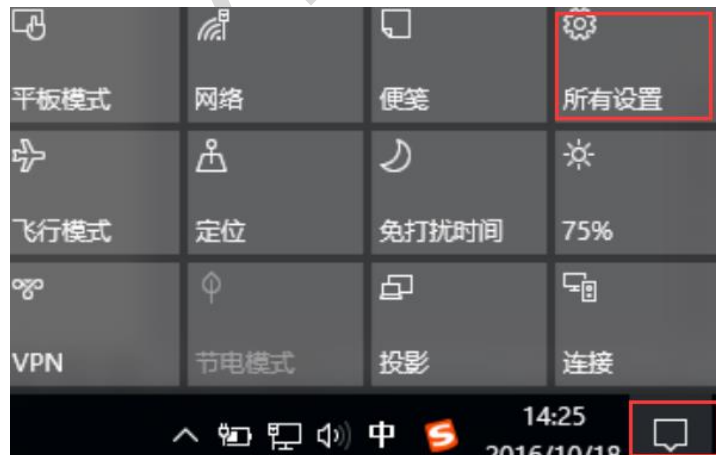
问题二：

在安装 FS-JTAG 调试软件时，电脑出现如下图情况：

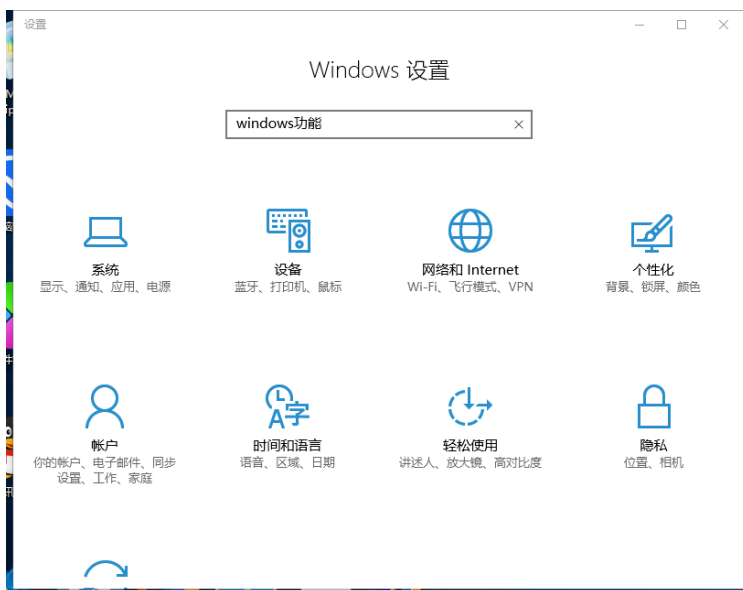


这是由于 Windows 系统.NET 版本问题导致，该问题的解决方法如下：

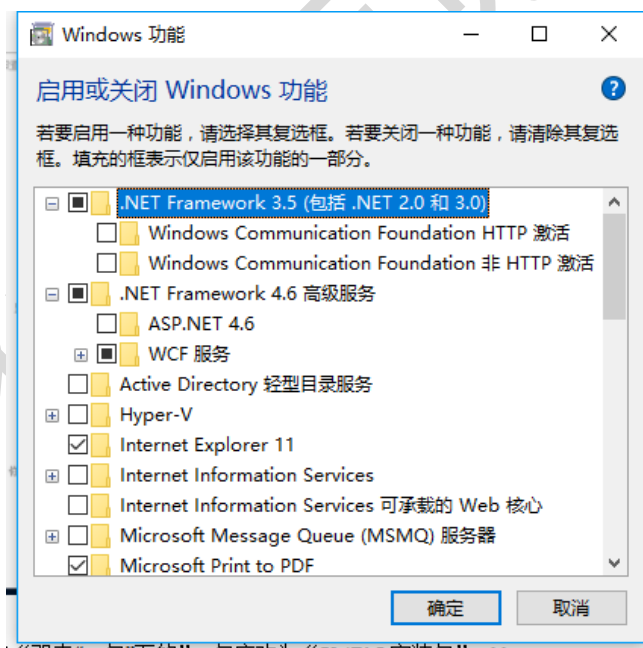
- (1) 将鼠标移动到右下角，点击“所有设置”，弹出 Windows 设置窗口。



- (2) 在窗口输入框内输入“Windows 功能”，回车即可。



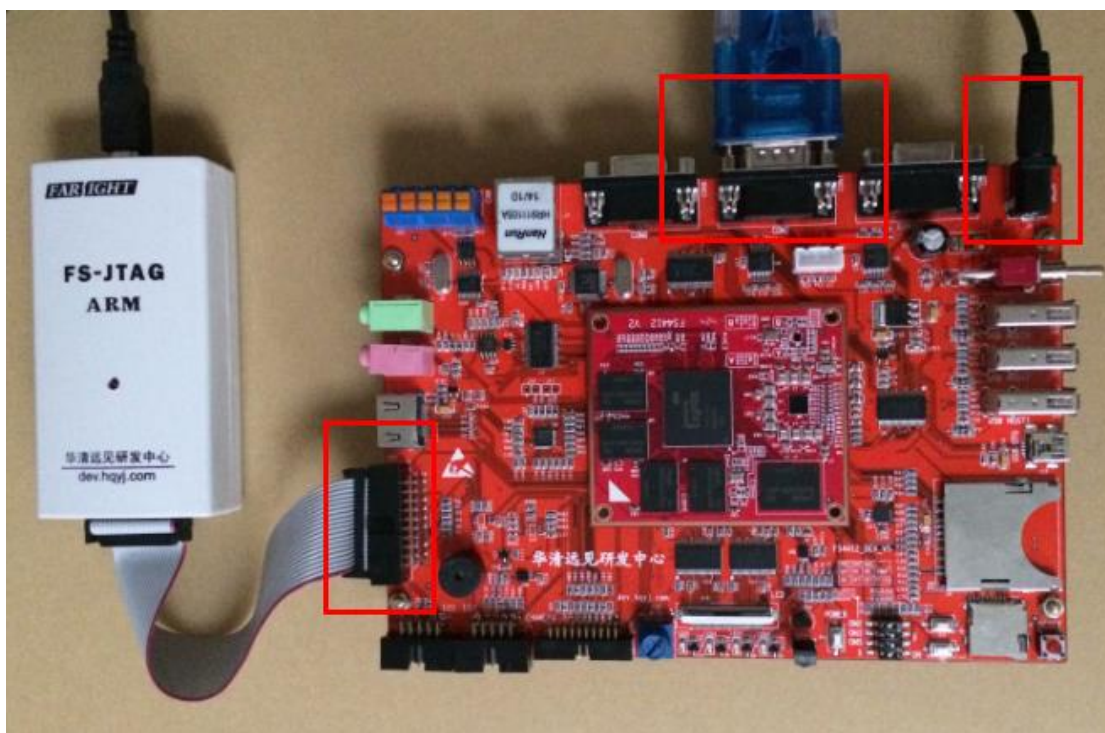
(3) 在弹出的 Windows 功能界面下勾选“.NET Framework 3.5 (包括 .NET 2.0 和 3.0)”即可。



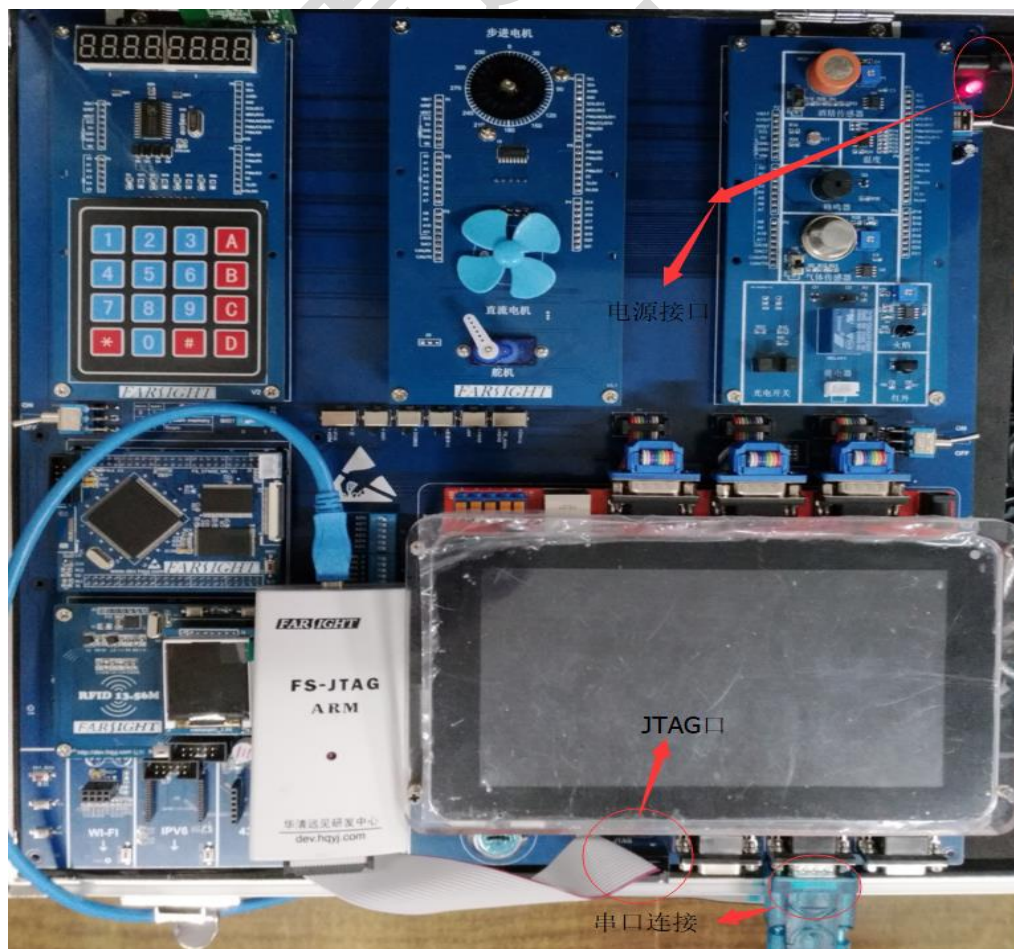
1.2.3 连接硬件平台

由于我们提供多种设备平台，每种设备的连接方式会有差异，所以请根据自己使用的设备平台选择对应的连接方式。

(1) 下图所示是 FS4412 的硬件连接图，需连接仿真器、USB 转串口线、电源。

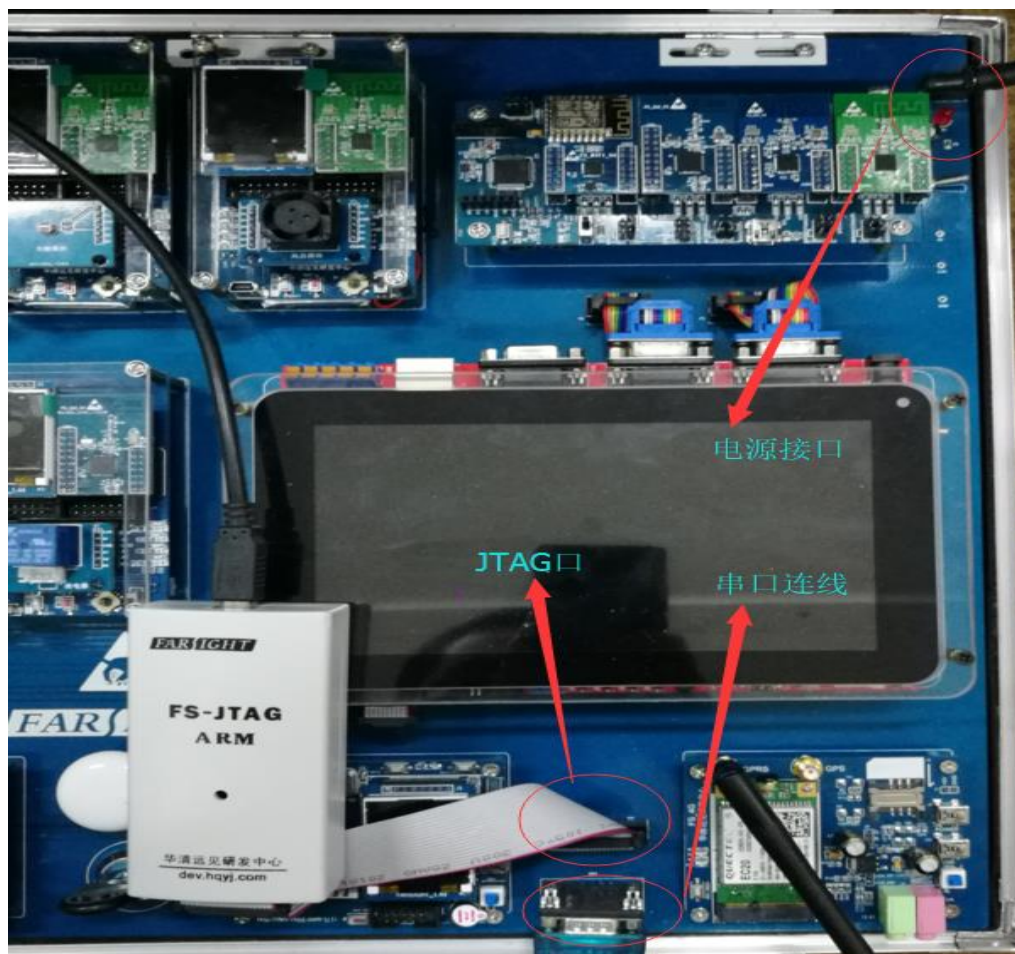


(2) 下图所示是 FS4412M4 的硬件连接图，需连接仿真器、USB 转串口线、电源。



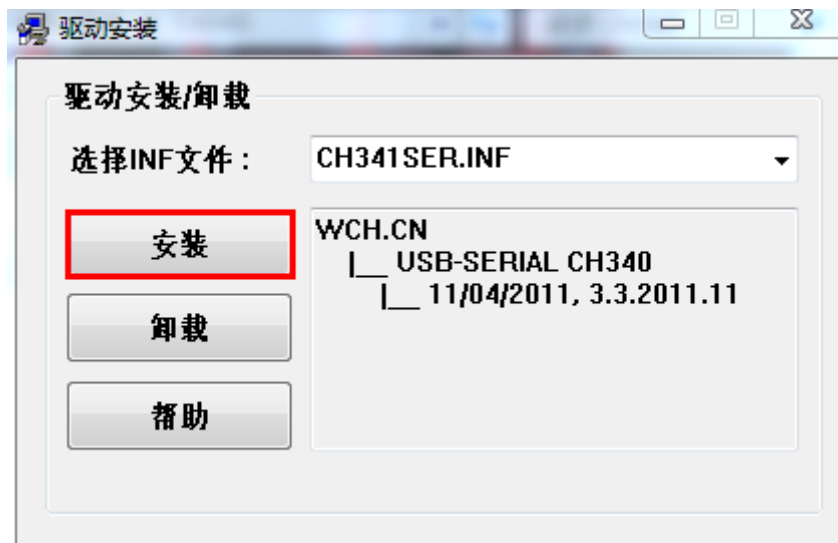


(3) 下图所示是 WSN4412C 的硬件连接图，需连接仿真器、USB 转串口线、电源。

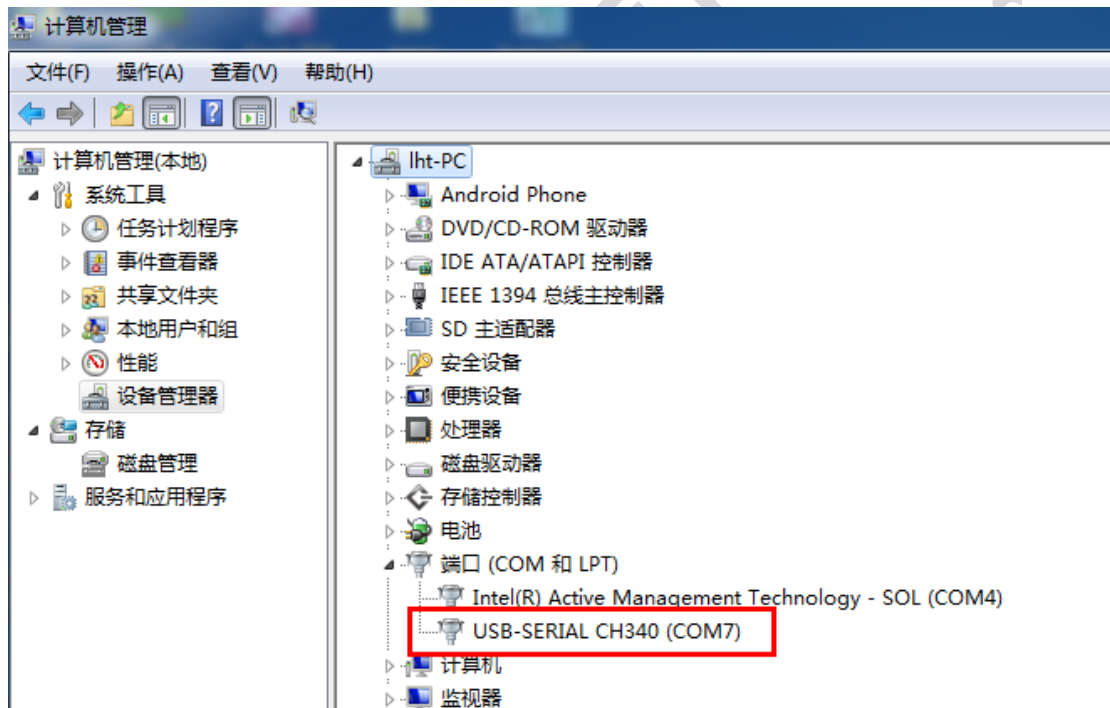


(4) 下图所示是 WSN4412A 和 WSN4412B 的硬件连接图，需连接仿真器、USB 转串口线、电源。



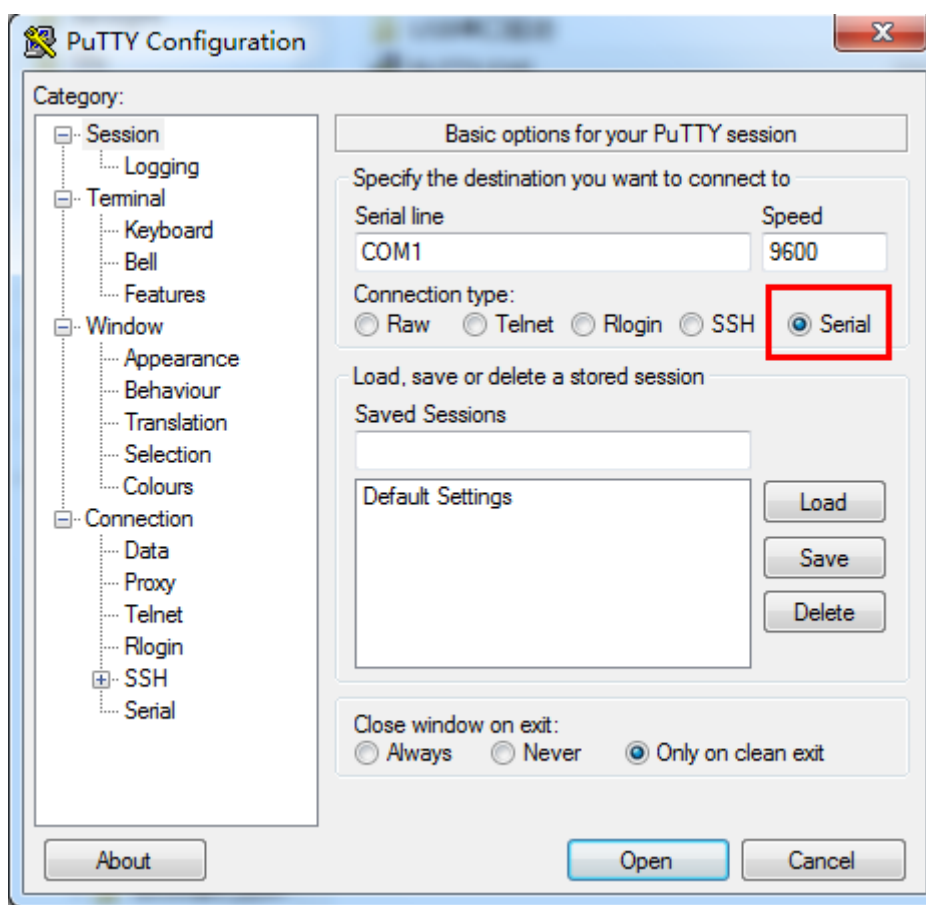


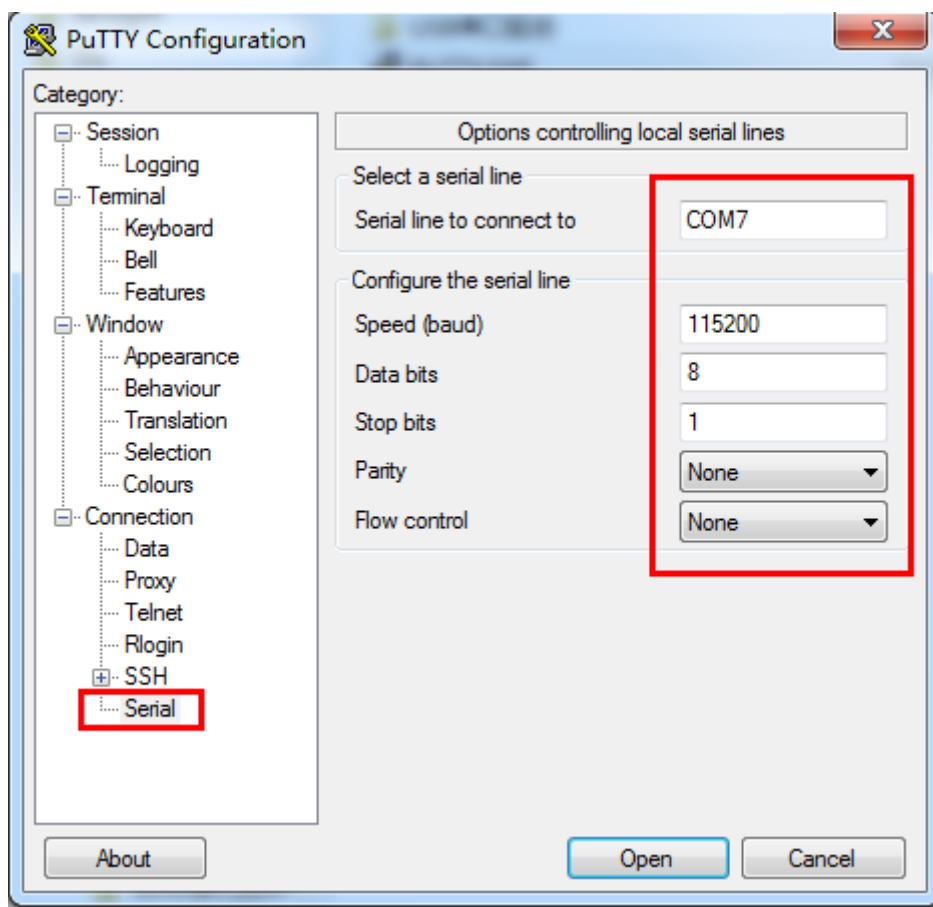
等待 20 秒左右，系统会提示安装完成。可以在设备管理器中查看到串口的信息，从而确定串口号。



1.2.5 Putty 串口终端配置

运行【华清远见-CORTEXA9 资料\工具软件\Windows\串口调试工具\PUTTY.EXE】。



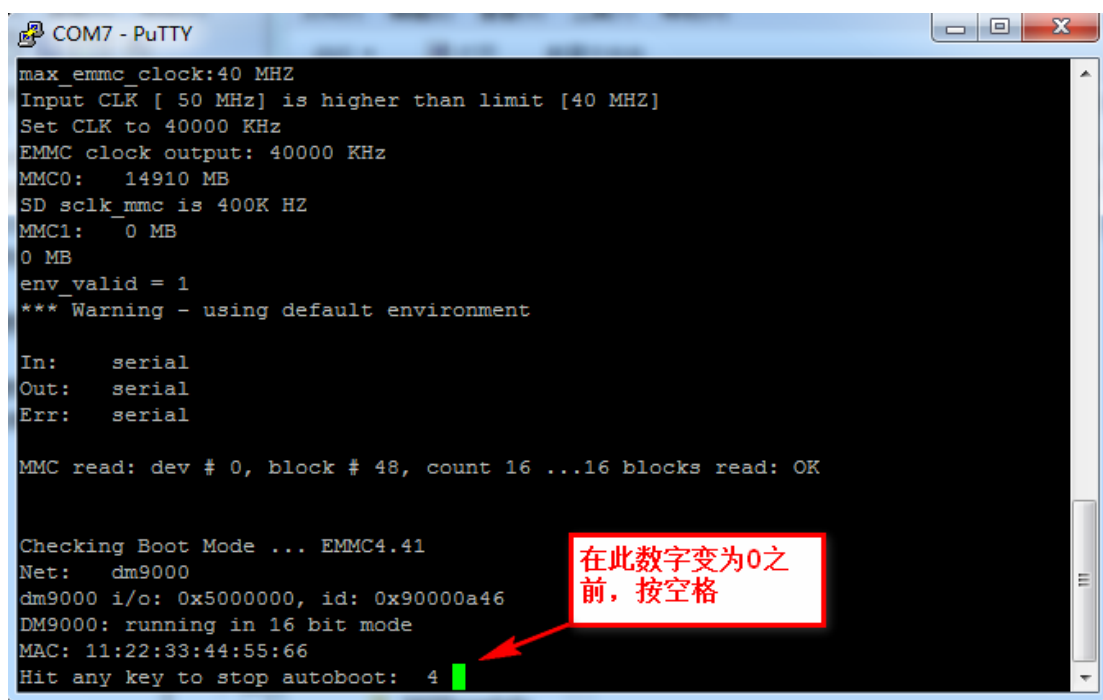


COM7 是串口号，不同机器、不同接口都有差异，请查看设备管理器中的信息。最后点击“Open”打开串口。

关闭开发板电源，将拨码开关 SW1 调至 0110(EMMC 启动模式)，然后打开电源



给开发板上电，此时串口终端会显示



```
COM7 - PuTTY
max_emmc_clock:40 MHZ
Input CLK [ 50 MHz] is higher than limit [40 MHz]
Set CLK to 40000 KHz
EMMC clock output: 40000 KHz
MMC0: 14910 MB
SD sclk_mmc is 400K HZ
MMC1: 0 MB
0 MB
env_valid = 1
*** Warning - using default environment

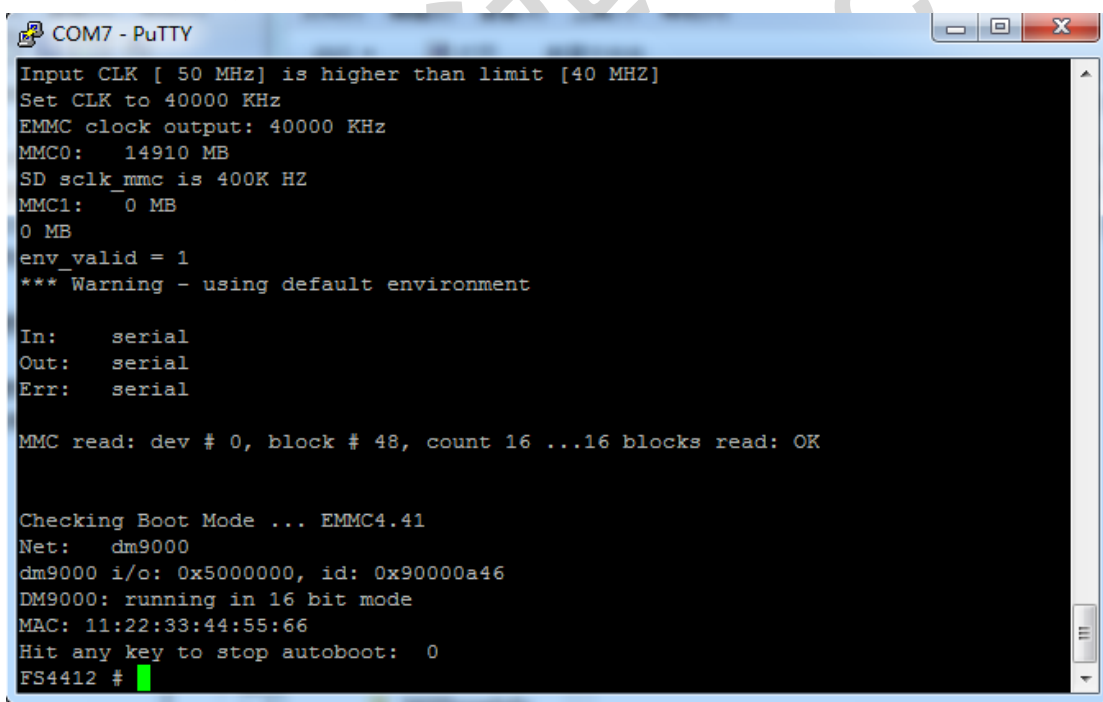
In: serial
Out: serial
Err: serial

MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK

Checking Boot Mode ... EMMC4.41
Net: dm9000
dm9000 i/o: 0x50000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
Hit any key to stop autoboot: 4
```

在此数字变为0之前，按空格

让系统停留在下图状态。



```
COM7 - PuTTY
Input CLK [ 50 MHz] is higher than limit [40 MHz]
Set CLK to 40000 KHz
EMMC clock output: 40000 KHz
MMC0: 14910 MB
SD sclk_mmc is 400K HZ
MMC1: 0 MB
0 MB
env_valid = 1
*** Warning - using default environment

In: serial
Out: serial
Err: serial

MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK

Checking Boot Mode ... EMMC4.41
Net: dm9000
dm9000 i/o: 0x50000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
Hit any key to stop autoboot: 0
FS4412 #
```

注意：以后每次连接仿真前，都需要确定处于此状态，保证不要启动到Linux，因为启动到Linux后，MMU功能会打开，导致仿真器无法正常使用。也可以改变uboot的bootcmd命令，让uboot不引导linux。方法如下图所示：在uboot终端输入：


```

bootcmd=sdfuse flashall;movi read kernel 40008000;movi read rootfs 40d00000 1000
00;bootm 40008000 40d00000

Environment size: 515/16380 bytes
FS4412 # setenv bootcmdbf $bootcmd
FS4412 # setenv bootcmd
FS4412 # saveenv
Saving Environment to SMDK bootable device...
.Environment have been saved eMMC done
FS4412 # print
bootdelay=5
baudrate=115200
ethaddr=11:22:33:44:55:66
ethact=dm9000
filesize=3A9E44
fileaddr=40008000
gatewayip=192.168.100.1
ipaddr=192.168.100.191
serverip=192.168.100.192
writekernel=tftp 40008000 zImage ; movi write kernel 40008000
bootargs=root=/dev/nfs nfsroot=192.168.100.192:/source/rootfs ip=192.168.100.191
:::::eth0:off init=/linuxrc console=ttySAC2,115200
stdin=serial
stdout=serial
stderr=serial
bootcmdbf=$bootcmd

Environment size: 427/16380 bytes
FS4412 #
    
```

重新启动 uboot，自动停在 uboot 终端。

1.3 Eclipse for ARM 使用

Eclipse for ARM 工具路径：【华清远见-CORTEXA9 资料\工具软件\Windows\FS-JTAG\eclipse\eclipse-cpp-helios-SR1-win32.zip】解压文件后，运行 eclipse.exe 文件。（注意：在 Win7 以上的用户，使用管理员模式打开）

(1) 指定一个工程存放目录

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

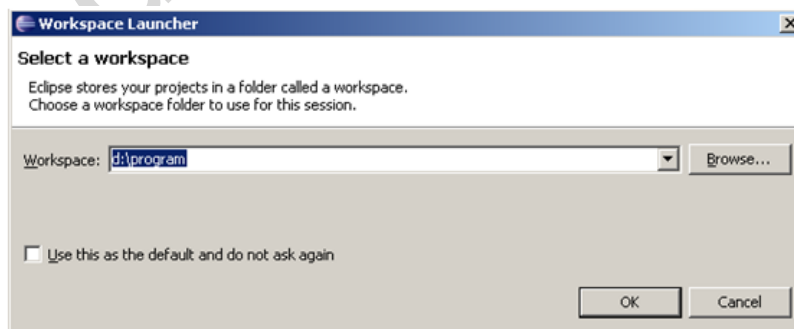


图 工程路径选择

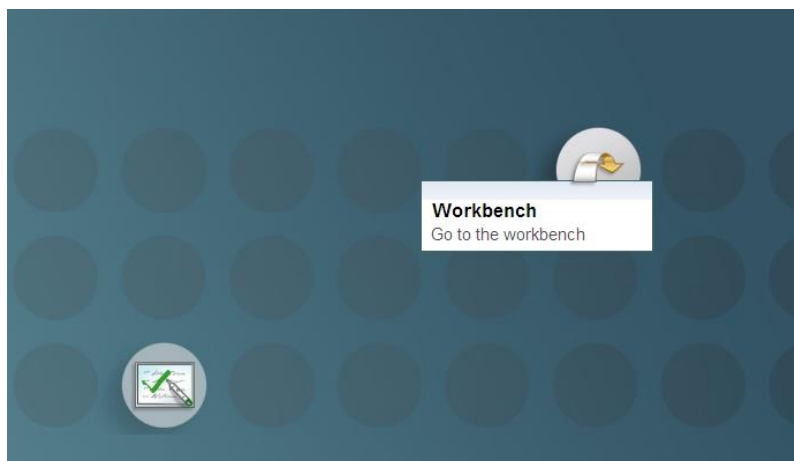


图 进入工程界面

第一次打开会出现这个界面，然后点击“Workbench”进入工程界面。弹出 工作界面。

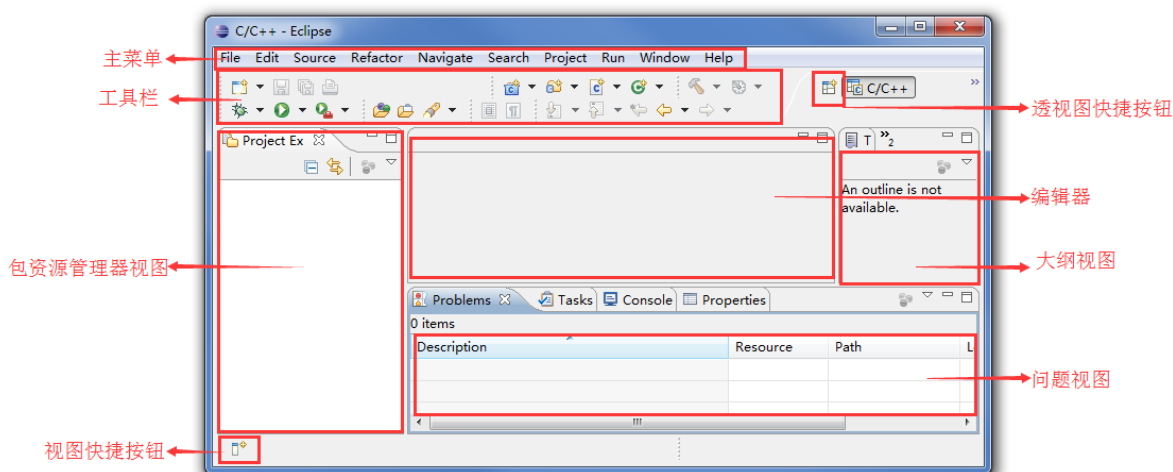


图 工程工作界面

- 主菜单：包括文件、编辑、源代码、重构、运行、窗口等菜单。大部分的向导和各种配置对话框都可以从主菜单中打开。
- 工具栏：包括文件工具栏、调试、运行等。工具栏中的按钮都是相应的菜单的快捷方式。
- 包资源管理器视图：用于显示项目中的源文件、引用的库等。
- 视图快捷按钮：用来切换到提供的其他视图。
- 透视图快捷按钮：用来切换到提供的各个透视图，提供 6 种透视图，常用的有：CVS 资源库研究、C/C++（缺省值）、调试等视图，后文调试项目即为调试视图，可通过点击此按钮返回主窗口。
- 编辑器：用于代码的编辑。
- 大纲视图：用于显示代码的纲要结构，单击结构树的各结点可以在编辑器中快速定位代码。
- 问题视图：用于显示代码或项目配置的错误，双击错误项可以快速定位代码。



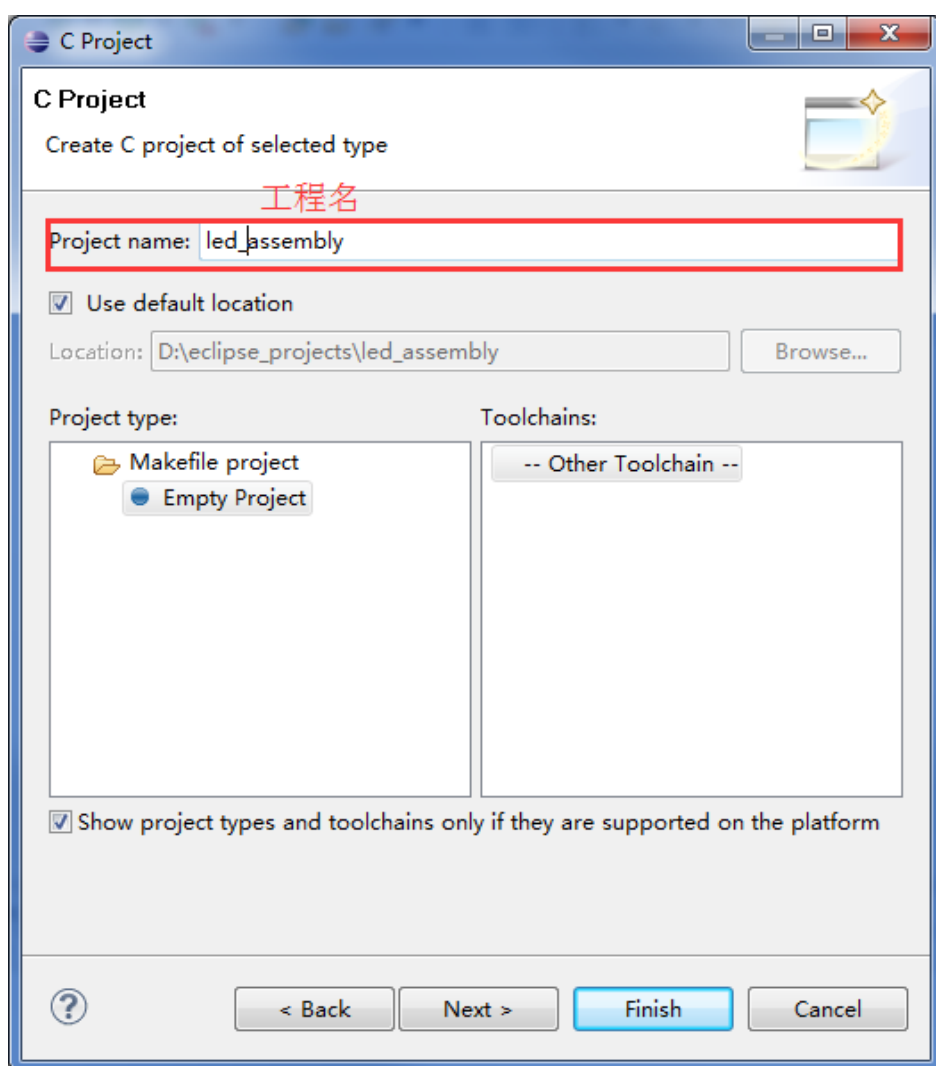
1.4 创建一个新工程


1.4.1 创建一个汇编工程

在一个汇编工程中,必须包含如下必要的文件:

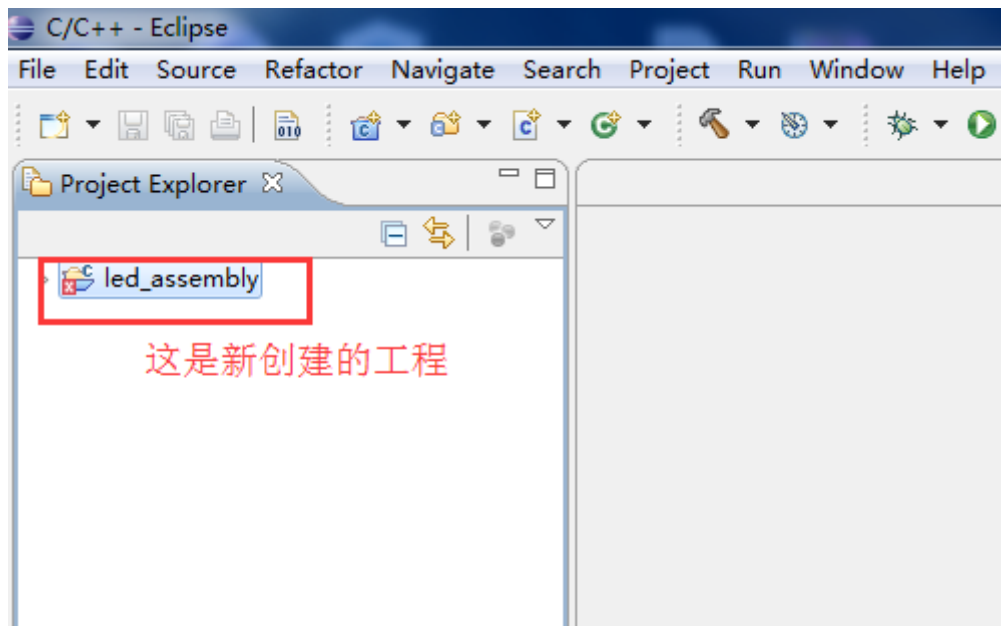
—工程名称	
—led_assembly.S	//存放汇编工程源代码
—Makefile	//用来定义整个工程的编译规则
—map.lds	//链接脚本文件
—Exynos4412.init	//仿真初始化文件

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



注意: 创建的 Empty Project 工程会出现差号,例如 “ led_assembly”,编译工程时出现编译时找不到 Makefile 文件,无法编译。不用担心,接下来我们会在工程中手动添加 Makefile 文件,添加完所有必要文件后编译工程,小叉号消失。

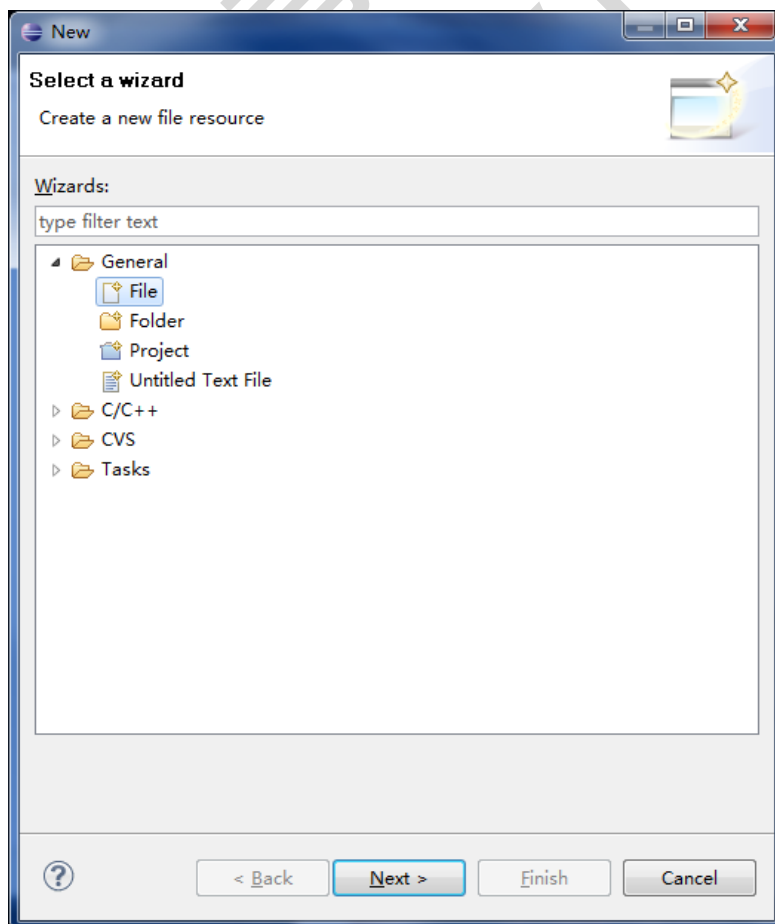
点击“Finish”,工程创建完成。



创建完成后，会在工程工作界面里有你创建的工程名，接下来添加工程必要文件。

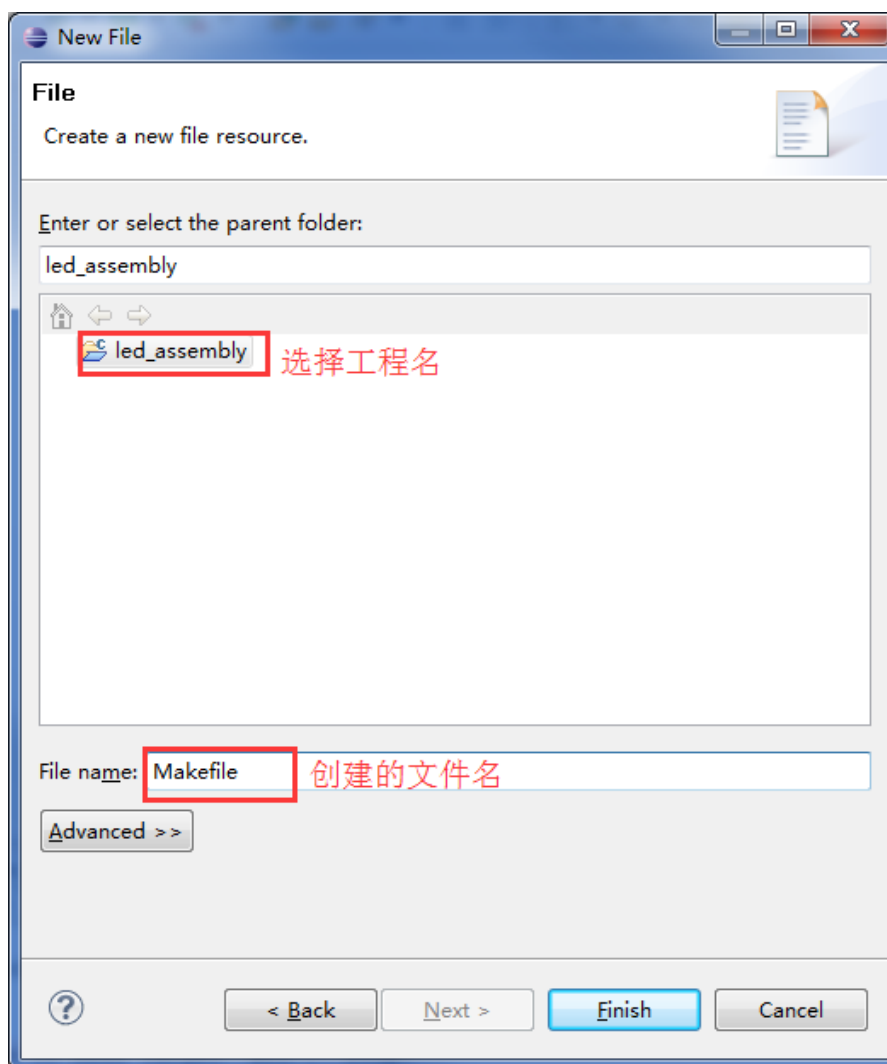
(1) 新建一个 Makefile 文件

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





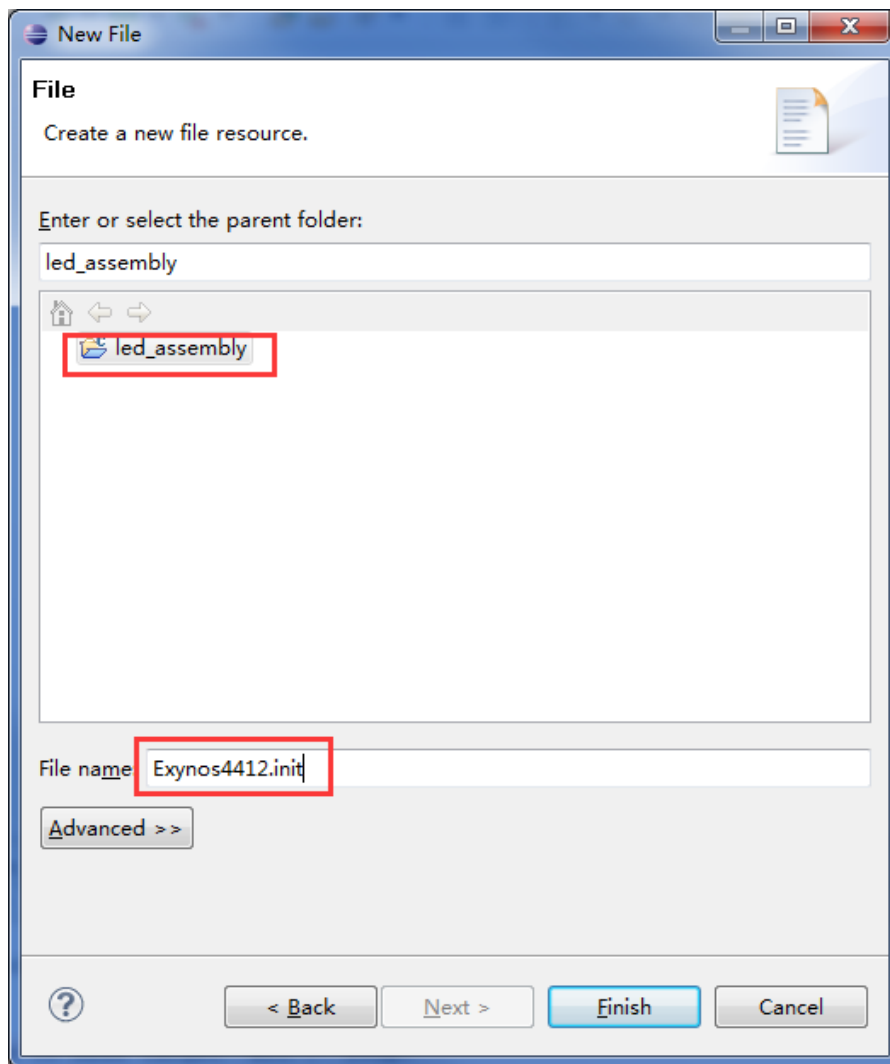
点击“Next”，弹出 Enter or select the parent tolde 对话框。



选择工程名，并在 File name 框内输入创建的文件名字

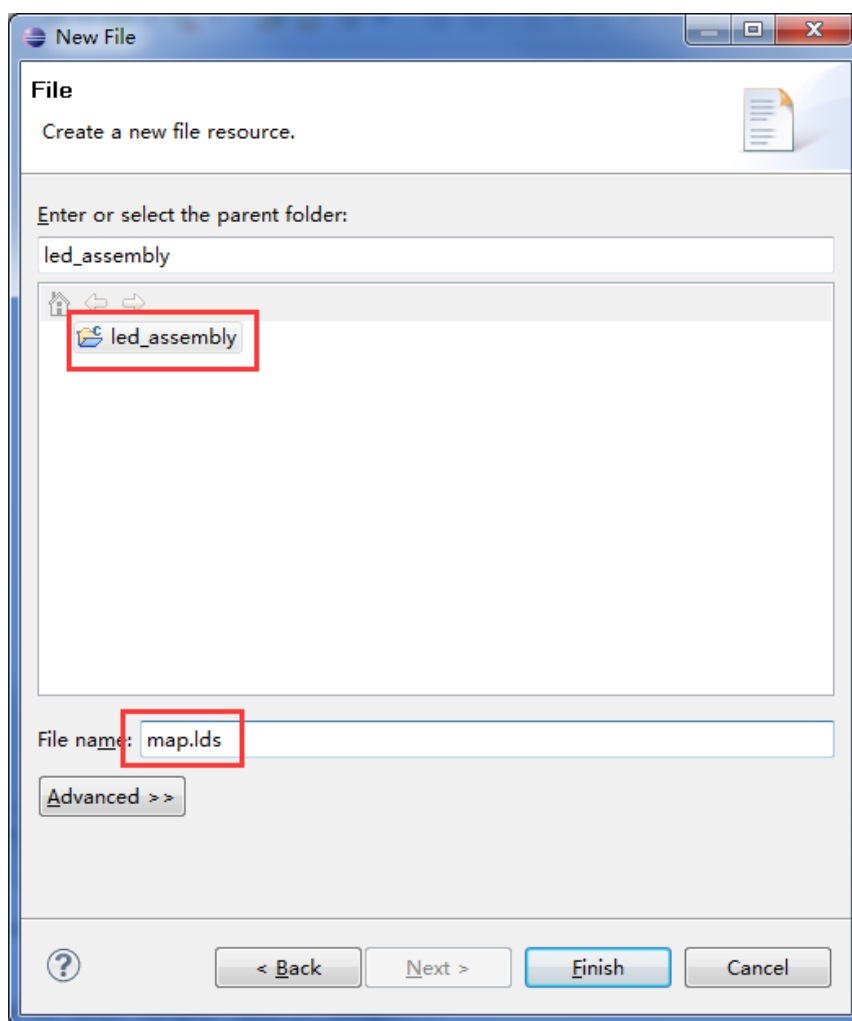
(2) 新建一个脚本文件

选择“File→New→Other”命令，在弹出的对话框中的 General 下单击 file，然后单击 next；然后选择所要指定的工程后，在文件名文本框中输入文件名 Exynos4412.init，单击“Finish”按钮。



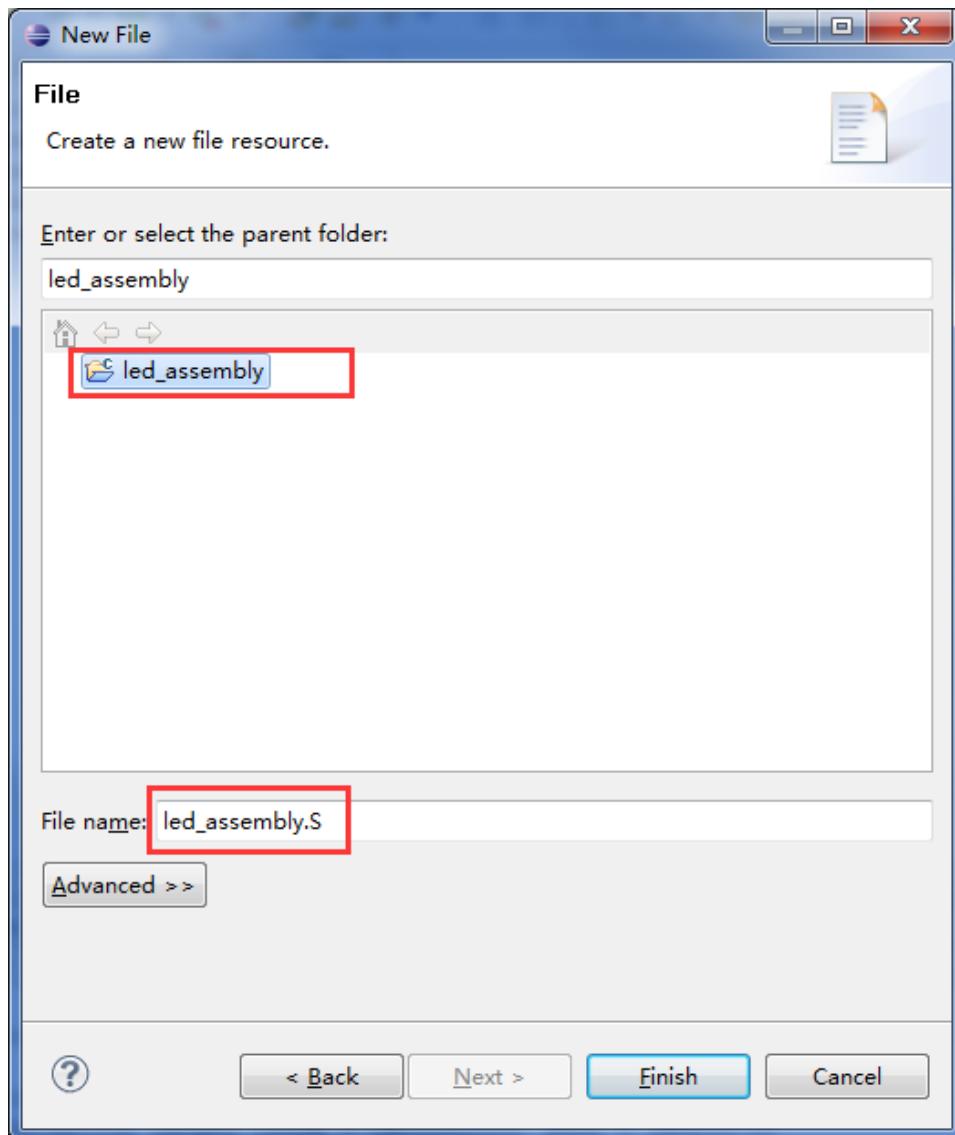
(3) 新建一个链接脚本文件

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



(4) 新建一个汇编源文件

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



至此工程及所需文件已经基本创建完成，为了能够实现工程的相关功能，我们必须在对应的文件中添加相应的代码。

(1) 在汇编源文件（led_assembly.S）中输入汇编代码

```

1  .equ  GPF3CON,  0x114001E0
2  .equ  GPF3DAT,  0x114001E4
3  .text
4  .global _start
5  _start:
6      LDR        R0,=GPF3CON
7      LDR        R1,=0X10000
8      STR        R1,[R0]          @//写控制寄存器，IO 引脚使能为输出
9  LOOP:
10     LDR        R0,=GPF3DAT
11     MOV        R1,#0X10         @//点亮 led4
12     STR        R1,[R0]
    
```




```

13      LDR      R2,=0FFFFFFF  @//延时
14 LOOP1:
15      SUB      R2,R2,#1
16      CMP      R2,#0
17      BNE      LOOP1
18      MOV      R1,#0X0      @//熄灭 led4
19      STR      R1,[R0]
20      LDR      R2,=0FFFFFFF  @//延时
21 LOOP2:
22      SUB      R2,R2,#1
23      CMP      R2,#0
24      BNE      LOOP2
25      B        LOOP
26      .end
27
    
```

(2) 在 map.lds 中输入如下信息:

```

1  OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
2  /*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
3  OUTPUT_ARCH(arm)
4  ENTRY(_start)
5  SECTIONS
6  {
7      . = 0x40008000;
8      . = ALIGN(4);
9      .text      :
10     {
11         led_assembly.o(.text)
12         *(.text)
13     }
14     . = ALIGN(4);
15     .rodata :
16     { *(.rodata) }
17     . = ALIGN(4);
18     .data :
19     { *(.data) }
20     . = ALIGN(4);
21     .bss :
22     { *(.bss) }
23 }
24
    
```



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

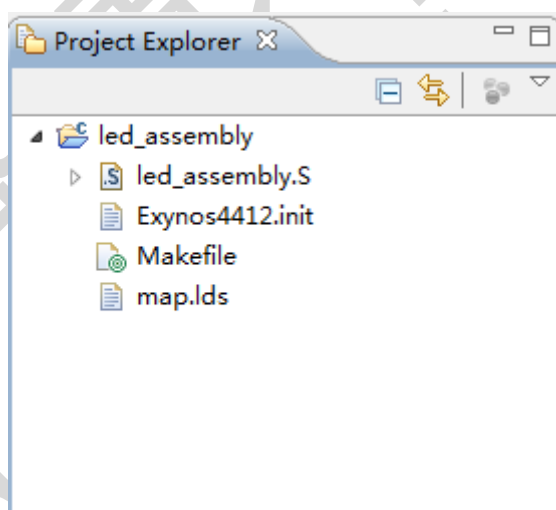
```
1 SHELL=C:/Windows/system32/cmd.exe
2
3 all:clean
4     arm-none-eabi-gcc-4.6.2 -O0 -g -c -o led_assembly.o led_assembly.S
5     arm-none-eabi-ld      led_assembly.o -Tmap.lds -o led_assembly.elf
6     arm-none-eabi-objcopy -O binary -S led_assembly.elf led_assembly.bin
7     arm-none-eabi-objdump -D led_assembly.elf > led_assembly.dis
8
9 clean:
10     rm -rf *.o *.bin *.elf *.dis
```

注意：MakeFile 每条命令的开头必须以[Tab]键开头

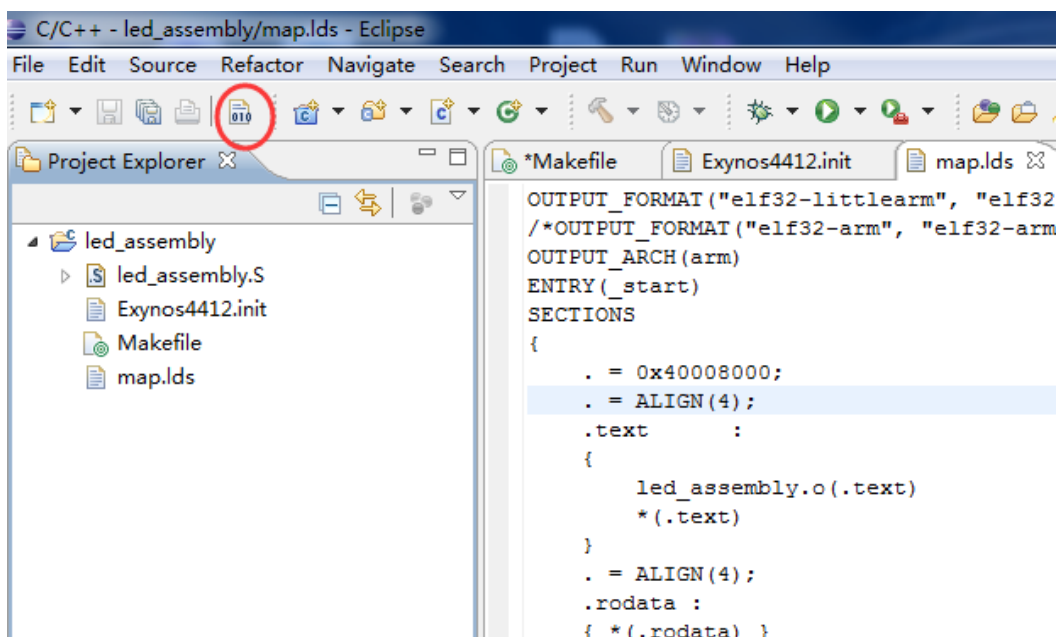
(4) 在 Exynos4412.init 文件中输入如下信息：

```
1 target remote localhost:3333
2 monitor halt reset
```

所有文件编辑完成后，保存文件，创建好的工程如下图所示：



点击如下图所示的编译图标（或者快捷键“**Ctrl + B**”）



编译成功后如下图所示：

```
C-Build [led_assembly]
rm -rf *.o *.bin *.elf *.dis
arm-none-eabi-gcc-4.6.2 -O0 -g -c -o led_assembly.o led_assembly.S
arm-none-eabi-ld led_assembly.o -Tmap.lds -o led_assembly.elf
arm-none-eabi-objcopy -O binary -S led_assembly.elf led_assembly.bin
arm-none-eabi-objdump -D led_assembly.elf > led_assembly.dis
```

至此 ARM 裸板汇编工程已经创建完成，如暂时不使用该工程时我们可以 Close project，下次使用时再 Open project（同时只能有一个工程是打开的），该工程相关配置不变。工程调试过程请参照本文 1.6 章节。

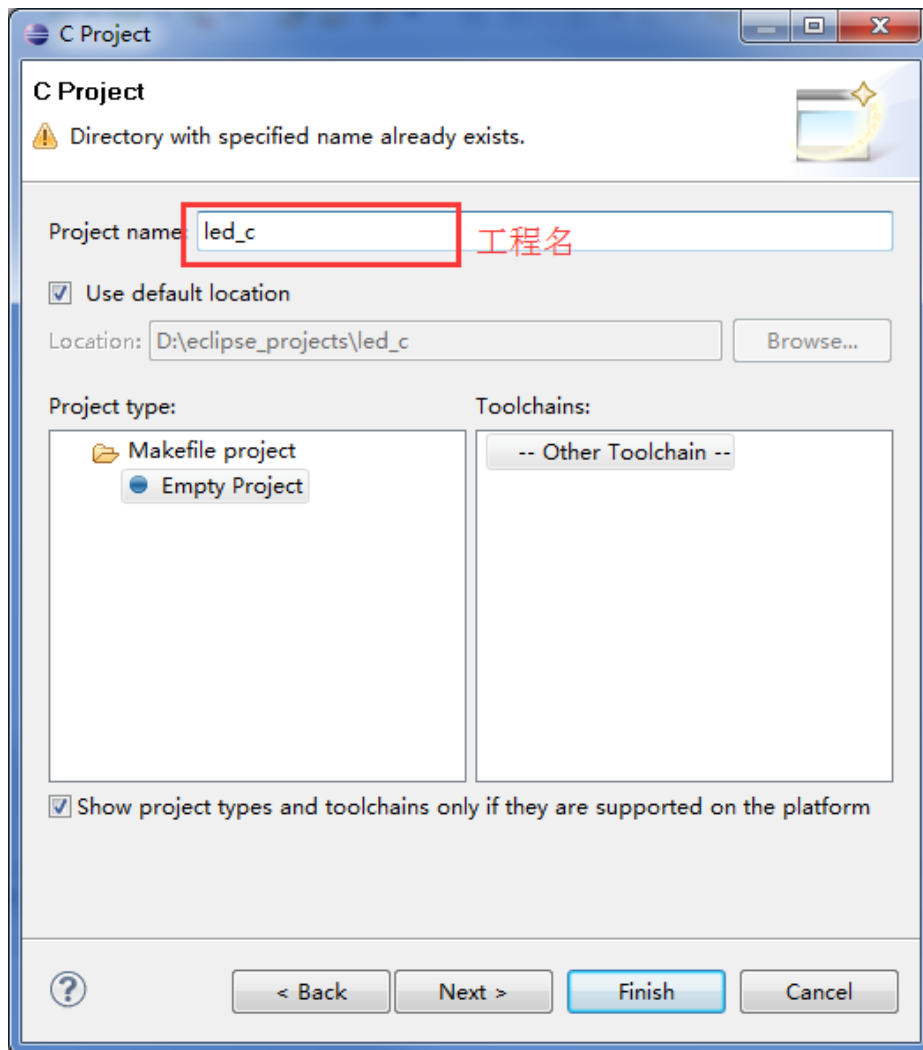
1.4.2 创建一个 C 工程


在一个 C 工程中,必须包含如下必要的文件：

—工程名称	
—common	//存放华清远见 FS_4412 通用库，已囊括本开发板所有硬件资源
—include	
—src	
—start	//存放汇编工程源代码
—start.S	
—main.c	//C 工程源码
—Exynos4412.init	//存放仿真用初始化文件
—map.lds	//链接脚本文件
—Makefile	//用来定义整个工程的编译规则

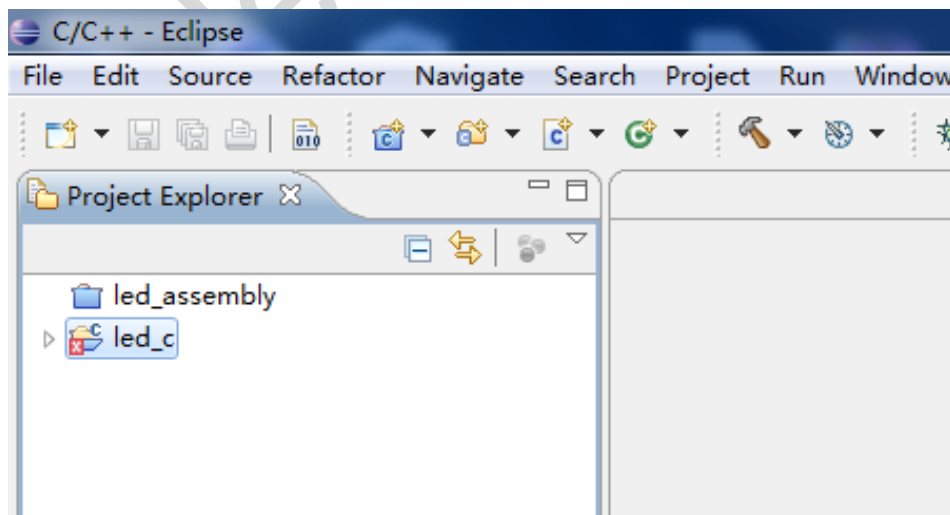
注：在 C 实验过程中，common 文件夹、start 文件夹、Makefile 文件、map.lds 文件、Exynos4412.init 文件是通用的，我们可以直接拷贝已有 C 工程中的这些文件或者自行对这些文件进行修改编写。

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



注意：创建的 Empty Project 工程会出现差号，例如“ led_c”，编译工程时出现编译时找不到 Makefile 文件，无法编译。不用担心，接下来我们会在工程中手动添加 Makefile 文件，添加完所有必要文件后编译工程，小叉号消失。

点击“Finish”，工程创建完成。

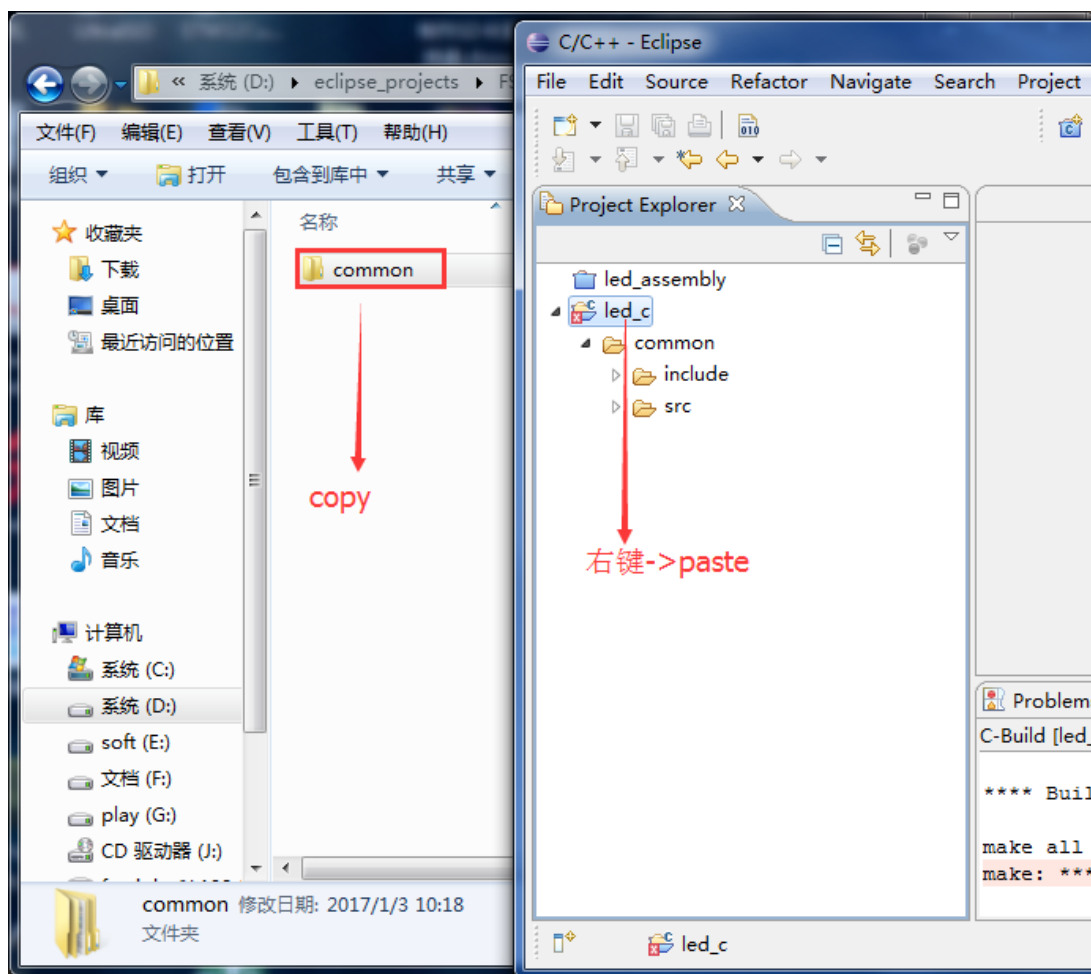




创建完成后，会在工程工作界面里有你创建的工程名，接下来添加工程必要文件。

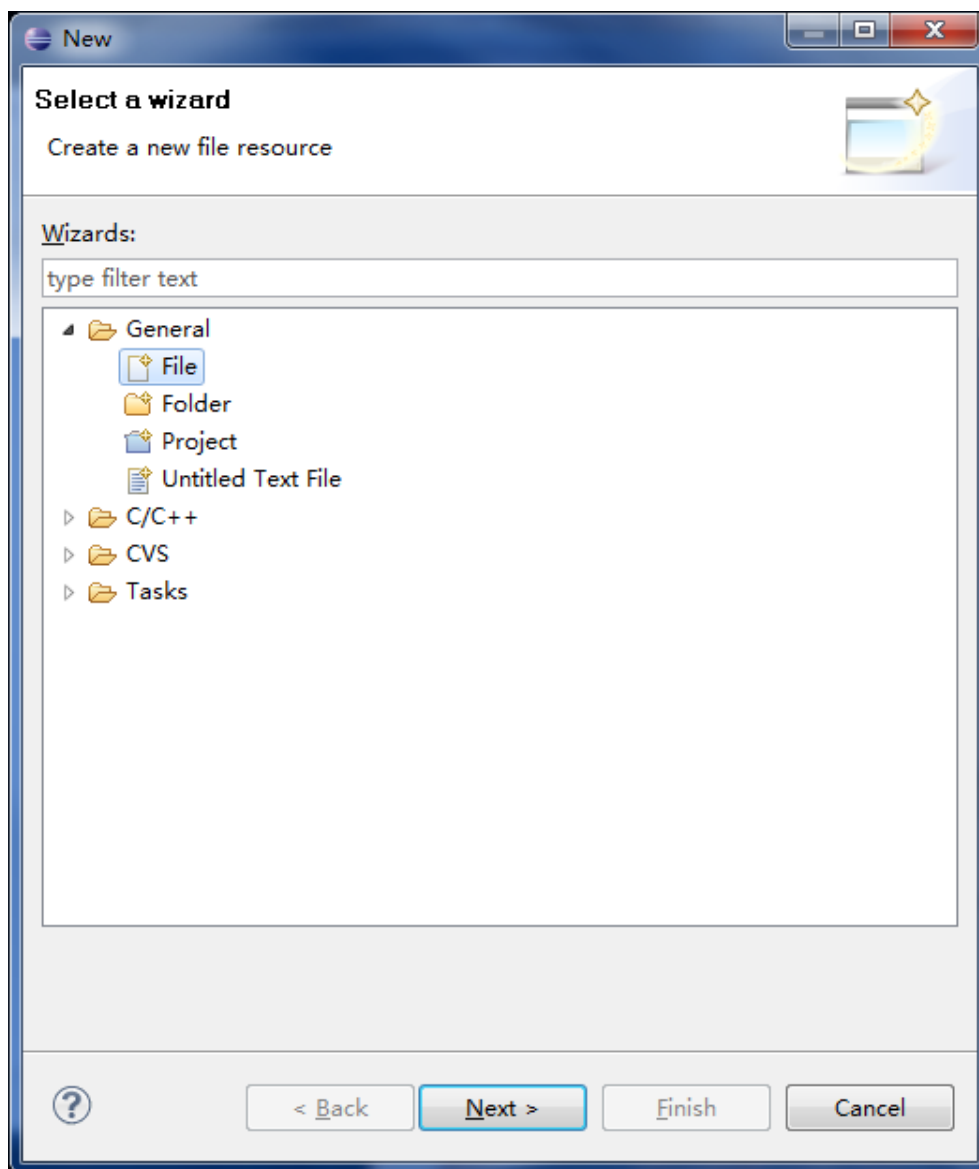
(1) 添加华清远见 FS_4412 通用库

拷贝【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\04-led\common】到 eclipse 工作目录下。如：D:\eclipse_projects\FS4412 目录，复制 common 文件夹，进入 Eclipse 界面，右键点击我们新建的 c 工程，选择 Paste。

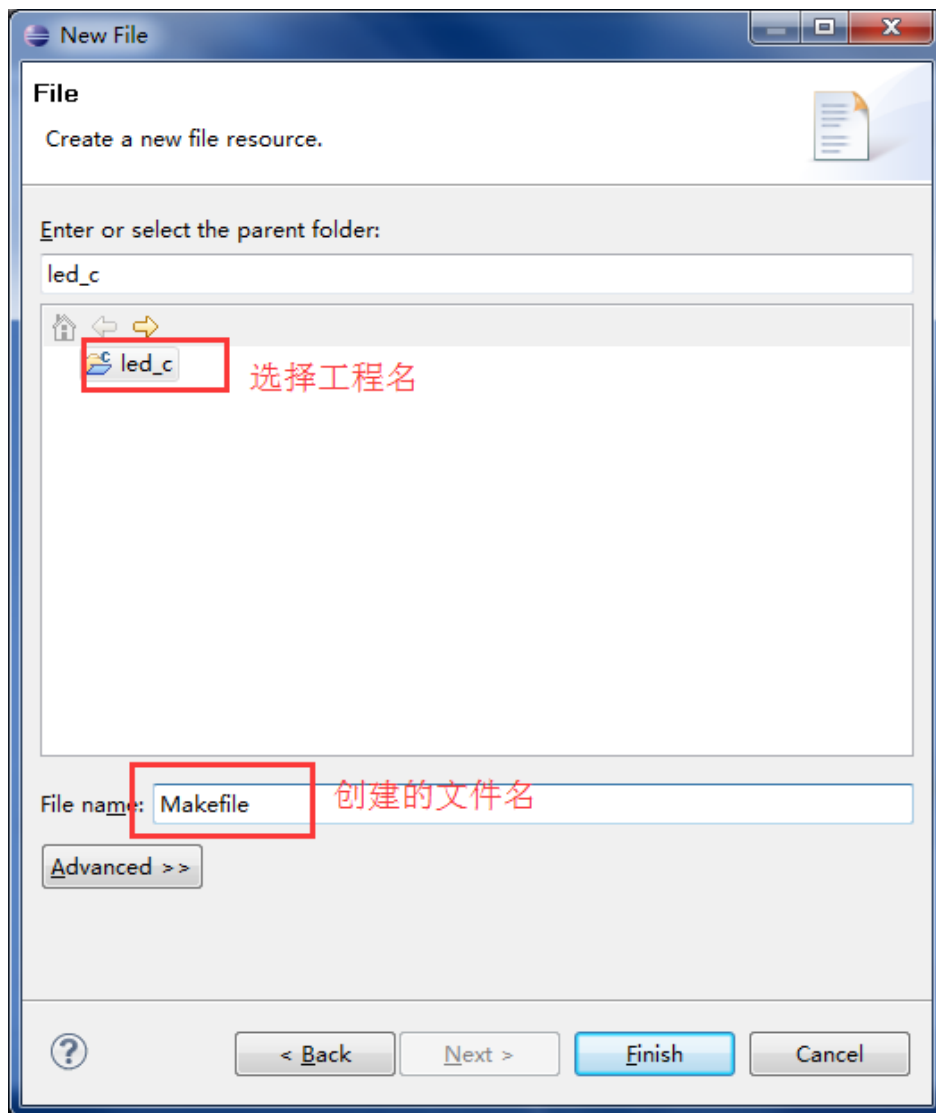


(2) 新建一个 MakeFile 文件

选择“File→New→Other”命令，在弹出的对话框 New 中的“General”下单击 File，然后单击 Next；然后选择所要指定的工程后，在文件名文本框中输入文件名 MakeFile，单击“Finish”按钮。（在 C 工程中，本文提供 Makefile 均为通用 Makefile，所以我们可以直接拷贝其他 C 工程（如 04-led）中的 Makefile 进行修改）



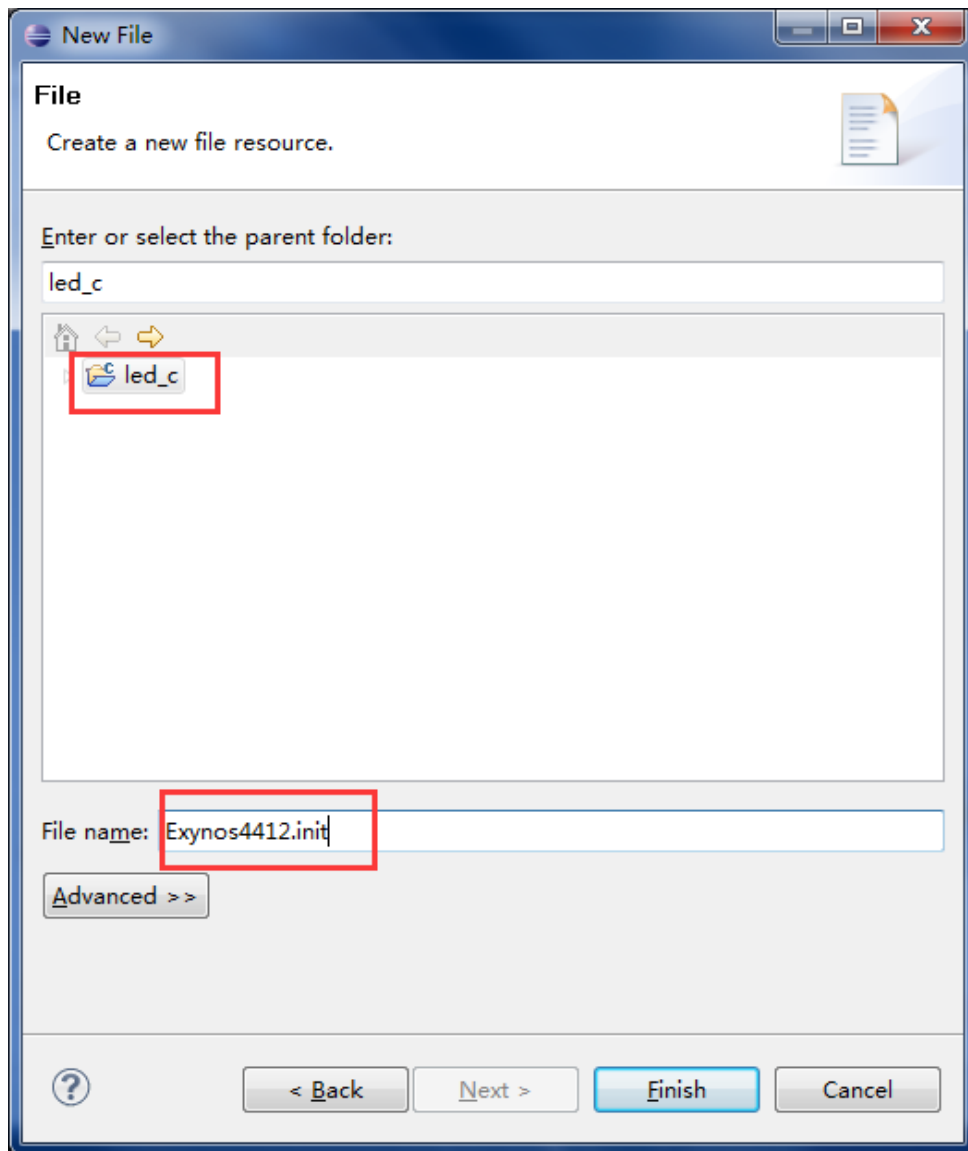
点击“Next”，弹出 Enter or select the parent to lde 对话框。



选择工程名，并在 File name 框内输入创建的文件名字

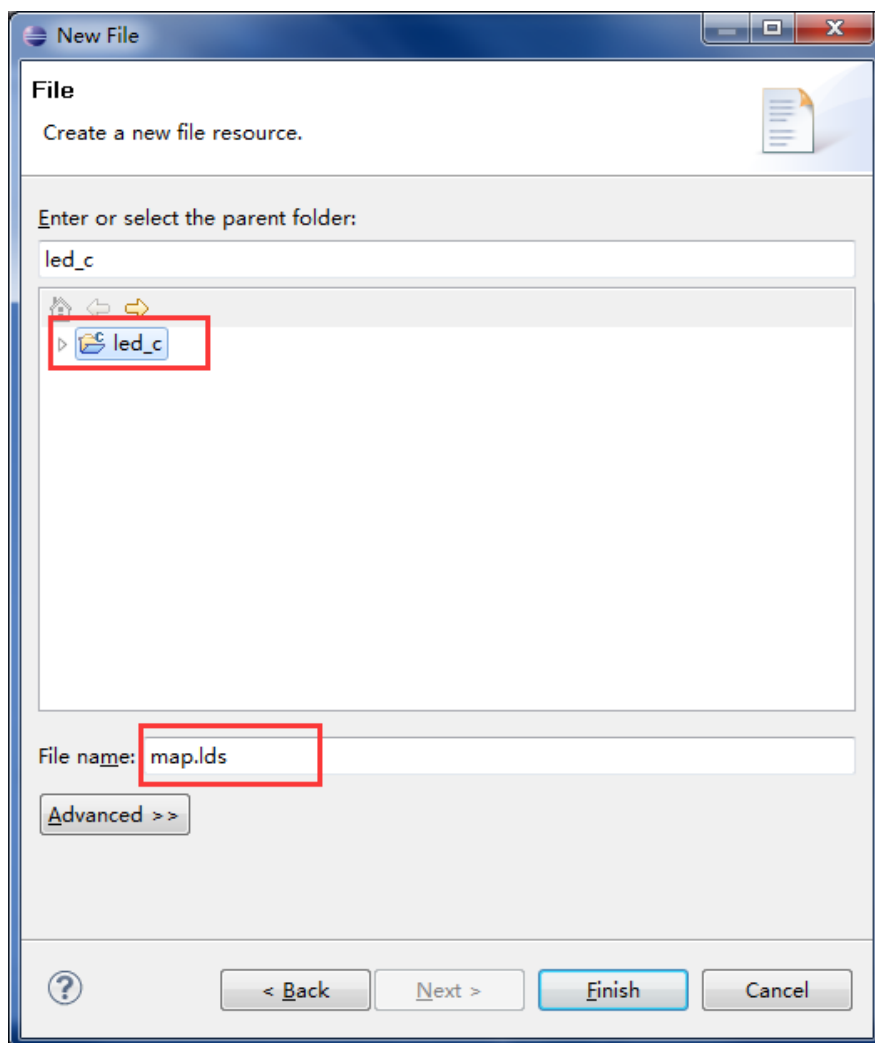
(3) 新建一个脚本文件

选择“File→New→Other”命令，在弹出的对话框中的 General 下单击 file，然后单击 next；然后选择所要指定的工程后，在文件名文本框中输入文件名 Exynos4412.init，单击“Finish”按钮。（在 C 工程中，所有的 Exynos4412.init 的作用是一样的，所以我们可以直接拷贝其他 C 工程（如 04-led）中的 Exynos4412.init 到本工程中）



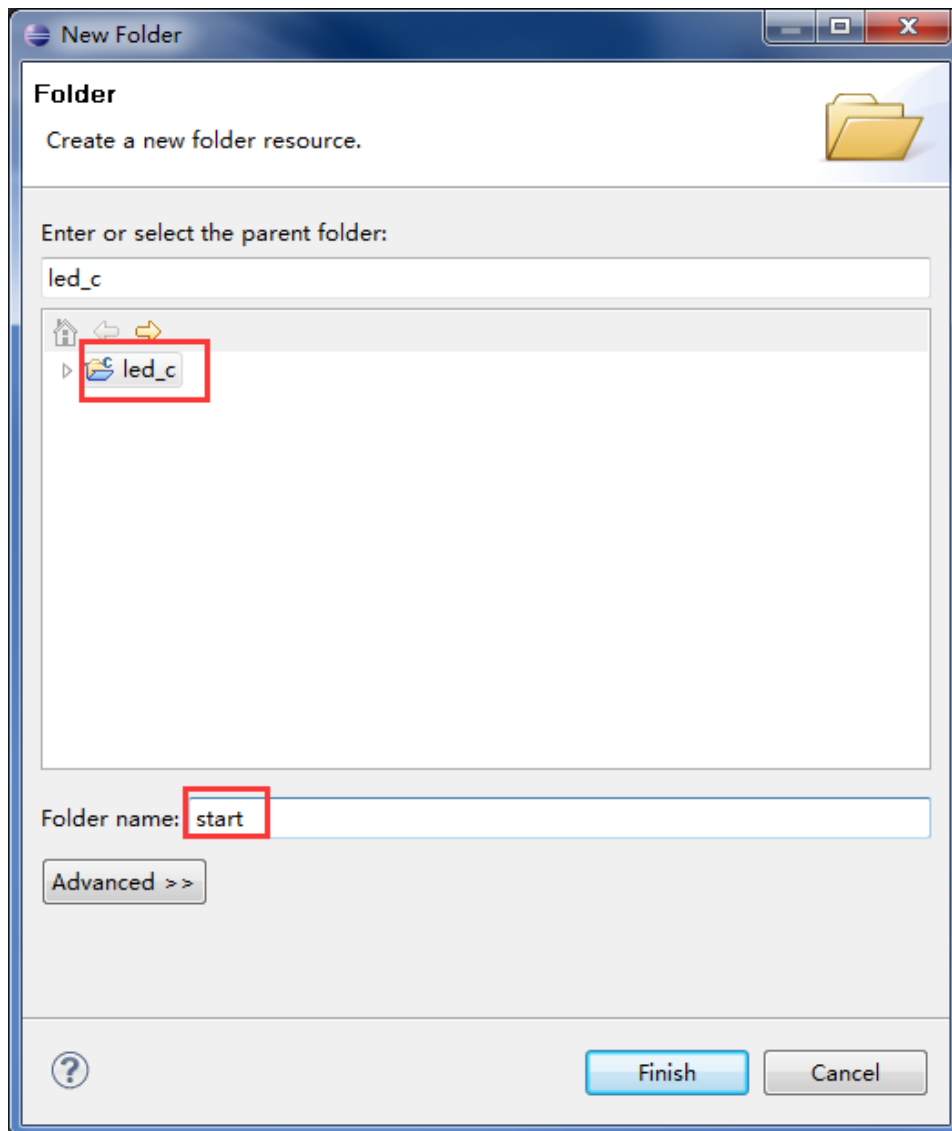
(4) 新建一个连接脚本文件

选择“File→New→Other”命令，在弹出的对话框中的 General 下单击 file ，然后单击 next；然后选择所要指定的工程后，在文件名文本框中输入文件名 map.lds，单击“Finish”按钮。（在 C 工程中，所有的 map.lds 的作用是一样的，所以我们可以直接拷贝其他 C 工程（如 04-led）中的 map.lds 到本工程中）

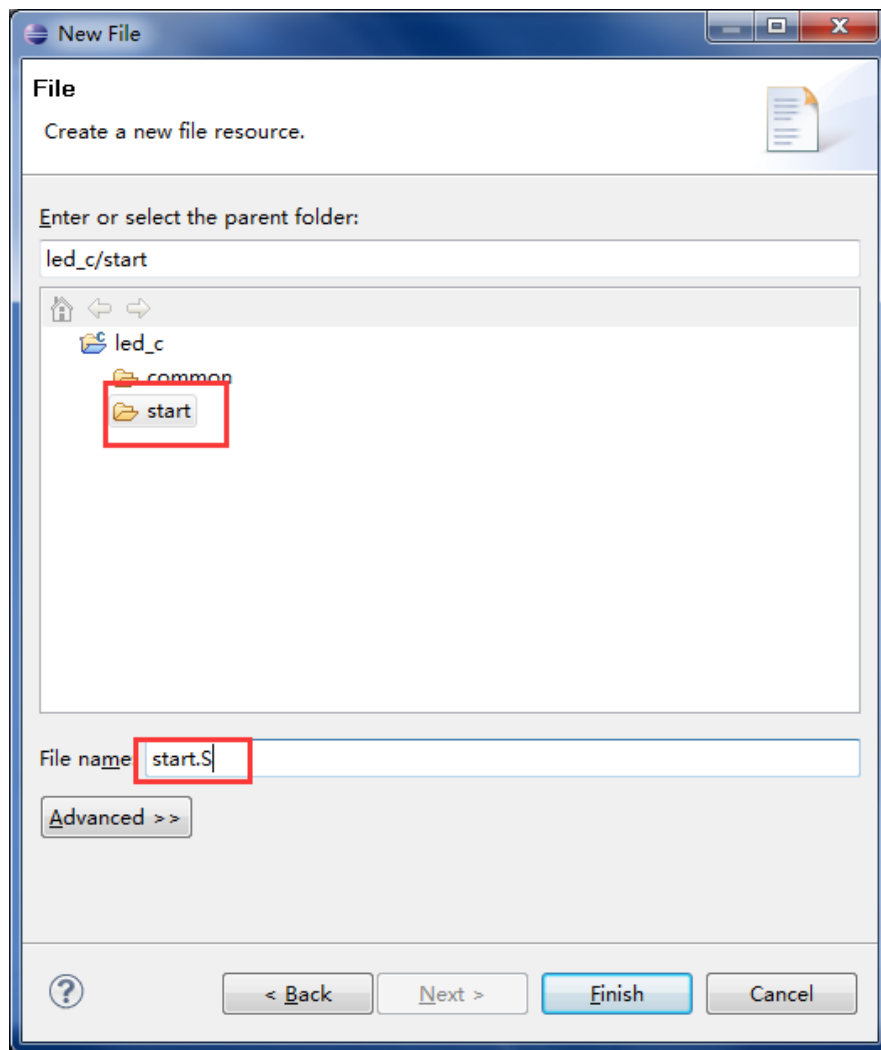


(5) 新建一个汇编源文件

首先在工程目录下创建 start 文件夹，“File→New→Other”命令，在弹出的对话框中的 General 下单击 Folder，然后单击 next；然后选择所要指定的工程后，在文件名文本框中输入文件夹名 start，单击“Finish”按钮。

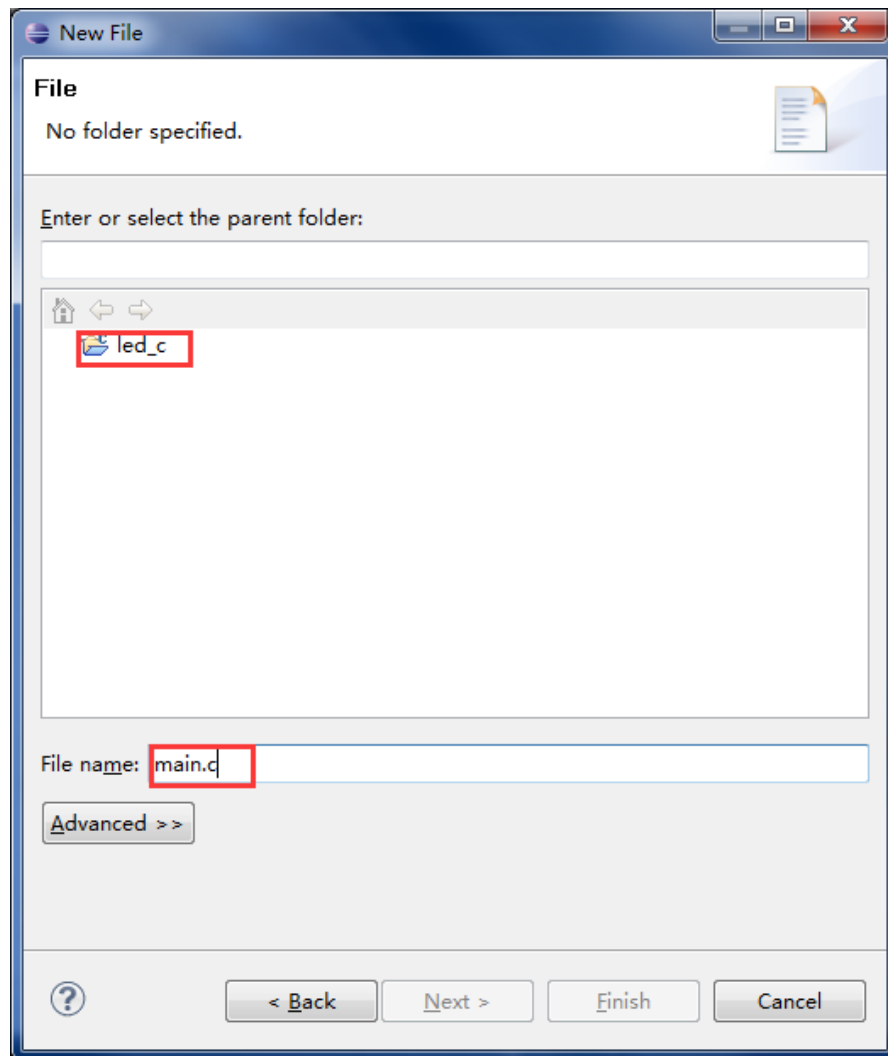


创建好 start 文件夹之后选择“File→New→Other”命令，在弹出的对话框中的 General 下点击 File，单击 next；然后选择所要指定的工程文件（这里选择刚创建的 start 文件），在文件名文本框中输入文件名 start.S（注 start 扩展名.S 为大写），单击“Finish”按钮。（在 C 工程中，所有的 start.S 的作用是一样的，所以我们可以直接拷贝其他 C 工程（如 04-led）中的 start.S 到本工程中 start 文件夹中）



(6) 新建一个 main 文件

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



至此工程及所需文件已经基本创建完成，为了能够实现工程的相关功能，我们必须在对应的文件中添加相应的代码。（在创建工程文件过程时，如某工程文件为拷贝已有工程中的文件，那么该文件可以略过此步操作）

（1）在汇编源文件（start/start.S）中输入汇编代：

```

1  .text
2  .global _start
3  _start:
4      b      reset
5      ldr    pc,_undefined_instruction
6      ldr    pc,_software_interrupt
7      ldr    pc,_prefetch_abort
8      ldr    pc,_data_abort
9      ldr    pc,_not_used
10     ldr    pc,_irq
11     ldr    pc,_fiq
12
13 _undefined_instruction: .word _undefined_instruction
    
```



```

14 _software_interrupt:      .word  _software_interrupt
15 _prefetch_abort:         .word  _prefetch_abort
16 _data_abort:             .word  _data_abort
17 _not_used:               .word  _not_used
18 _irq:                    .word  irq_handler
19 _fiq:                    .word  _fiq
20
21
22 reset:
23
24     ldr r0,=0x40008000
25     mcr p15,0,r0,c12,c0,0      @ Vector Base Address Register
26
27     mrs     r0,cpsr
28     bic     r0,r0,#0x1f
29     orr     r0,r0,#0xd3
30     msr     cpsr,r0           @ Enable svc mode of cpu
31
32     mov r0, #0xffffffff
33     mcr p15, 0, r0, c1, c0, 2
34                                     @ Defines access permissions for each coprocessor
35                                     @ Privileged and User mode access
36
37     /*
38     * Invalidate L1 I/D
39     */
40     mov r0, #0                @ set up for MCR
41     mcr p15, 0, r0, c8, c7, 0 @ invalidate TLBs
42     mcr p15, 0, r0, c7, c5, 0 @ invalidate icache
43
44     @Set the FPEXC EN bit to enable the FPU:
45     MOV r3, #0x40000000
46     fmxr FPEXC, r3
47
48     /*
49     * disable MMU stuff and caches
50     */
51     mrc p15, 0, r0, c1, c0, 0
52     bic r0, r0, #0x00002000 @ clear bits 13 (--V-)
53     bic r0, r0, #0x00000007 @ clear bits 2:0 (-CAM)
54     orr r0, r0, #0x00001000 @ set bit 12 (---I) Icache
55     orr r0, r0, #0x00000002 @ set bit 1 (--A-) Align
56     orr r0, r0, #0x00000800 @ set bit 11 (Z---) BTB
57     mcr p15, 0, r0, c1, c0, 0
58
59     /* LED Test Code */
60     ldr r0, =0x114001E0

```



```

58     ldr r1, [r0]
59     bic r1, r1, #0xf0000
60     orr r1, r1, #0x10000
61     str r1, [r0]
62
63     ldr r0, =0x114001E8
64     ldr r1, [r0]
65     bic r1, r1, #0x300
66     str r1, [r0]
67
68     ldr r0, =0x114001E4
69     ldr r1, [r0]
70     orr r1, r1, #0x10
71     str r1, [r0]
72
73 init_stack:
74     ldr     r0, stacktop /*get stack top pointer*/
75
76     /******svc mode stack*****/
77     mov     sp, r0
78     sub     r0, #128*4 /*512 byte for irq mode of stack*/
79     /****irq mode stack***/
80     msr     cpsr, #0xd2
81     mov     sp, r0
82     sub     r0, #128*4 /*512 byte for irq mode of stack*/
83     /****fiq mode stack****/
84     msr     cpsr, #0xd1
85     mov     sp, r0
86     sub     r0, #0
87     /****abort mode stack****/
88     msr     cpsr, #0xd7
89     mov     sp, r0
90     sub     r0, #0
91     /****undefine mode stack****/
92     msr     cpsr, #0xdb
93     mov     sp, r0
94     sub     r0, #0
95     /*** sys mode and usr mode stack ***/
96     msr     cpsr, #0x10
97     mov     sp, r0 /*1024 byte for user mode of stack*/
98
99     b       main
100
101     .align 4
    
```



```

102
103     /**** swi_interrupt handler ****/
104
105
106     /**** irq_handler ****/
107 irq_handler:
108
109     sub lr,lr,#4
110     stmfd sp!,{r0-r12,lr}
111 // bl do_irq
112     ldmfd sp!,{r0-r12,pc}^
113
114 stacktop:    .word    stack+4*512
115 .data
116
117 stack:    .space    4*512
    
```

(2) 在 map.lds 中输入如下信息

```

1 OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
2 /*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
3 OUTPUT_ARCH(arm)
4 ENTRY(_start)
5 SECTIONS
6 {
7     . = 0x40008000;
8     . = ALIGN(4);
9     .text :
10    {
11        start/start.o(.text)
12        *(.text)
13    }
14    . = ALIGN(4);
15    .rodata :
16    { *(.rodata) }
17    . = ALIGN(4);
18    .data :
19    { *(.data) }
20    . = ALIGN(4);
21    .bss :
22    { *(.bss) }
23 }
    
```



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

```

1 # CORTEX-A9 PERI DRIVER CODE
2 # VERSION 2.0
3 # ATHUOR www.dev.hqyj.com
4 # MODIFY DATE
5 # 2014.05.28 Makefile
6
7 SHELL=C:\Windows\System32\cmd.exe
8 CROSS_COMPILE = arm-none-eabi-
9 NAME = led_c
10
11 CFLAGS += -g -O0 -mabi=apcs-gnu -mcpu=neon -mfloat-abi=softfp \
12         -fno-builtin -nostdinc -I ./common/include
13 LD = $(CROSS_COMPILE)ld
14 CC = $(CROSS_COMPILE)gcc
15 OBJCOPY = $(CROSS_COMPILE)objcopy
16 OBJDUMP = $(CROSS_COMPILE)objdump
17
18 OBJSs := $(wildcard start/*.S) $(wildcard common/src/*.S) \
19         $(wildcard *.S) \
20         $(wildcard start/*.c) $(wildcard common/src/*.c) \
21         $(wildcard usr/*.c) $(wildcard *.c)
22 OBJS := $(patsubst %.S,%.o,$(OBJSs))
23 OBJ := $(patsubst %.c,%.o,$(OBJS))
24
25 %.o: %.S
26     $(CC) $(CFLAGS) -c -o $@ $<
27 %.o: %.c
28     $(CC) $(CFLAGS) -c -o $@ $<
29 all:clean $(OBJ)
30     $(LD) $(OBJ) -T map.lds -o $(NAME).elf
31     $(OBJCOPY) -O binary $(NAME).elf $(NAME).bin
32     $(OBJDUMP) -D $(NAME).elf > $(NAME).dis
33
34 clean:
35     rm -rf $(OBJ) *.elf *.bin *.dis *.o
    
```

注意：MakeFile 每条命令的开头必须以[Tab]键开头，如该文件为拷贝自己有工程，那么直接修改第九行 NAME 即可。

(4) 在 Exynos4412.init 文件中输入如下信息：

```

1 target remote localhost:3333
2 monitor halt reset
    
```




(5) 在 main.c 文件中输入如下信息:

```

1  /*
2  *@brief    This example describes how to use GPIO to drive LEDs
3  *@date:     02. June. 2014
4  *@author    liujh@farsight.com.cn
5  *@Contact   Us: http://dev.hqyj.com
6  *Copyright(C) 2014, Farsight
7  */
8
9  #include "exynos_4412.h"
10 /******
11  * @brief      mydelay_ms program body
12  * @param[in]   int (ms)
13  * @return      None
14  *****/
15 void mydelay_ms(int ms)
16 {
17     int i, j;
18     while(ms--)
19     {
20         for (i = 0; i < 5; i++)
21             for (j = 0; j < 514; j++);
22     }
23 }
24
25 /*-----MAIN FUNCTION-----*/
26 /******
27  * @brief      Main program body
28  * @param[in]   None
29  * @return      int
30  *****/
31 int main(void)
32 {
33     /*
34     *Config
35     */
36
37     GPX2.CON = (GPX2.CON & ~(0xf<<28)) | 1<<28; //GPX2_7:output, LED2
38     GPX1.CON = (GPX1.CON & ~(0xf)) | 1; //GPX1_0:output, LED3
39     GPF3.CON = (GPF3.CON & ~(0xf<<16 | 0xf<<20)) | (1<<16 | 1<<20);
40     //GPF3_4:output, LED4  GPF3_5:output, LED5
41
42     while(1)
43     {

```

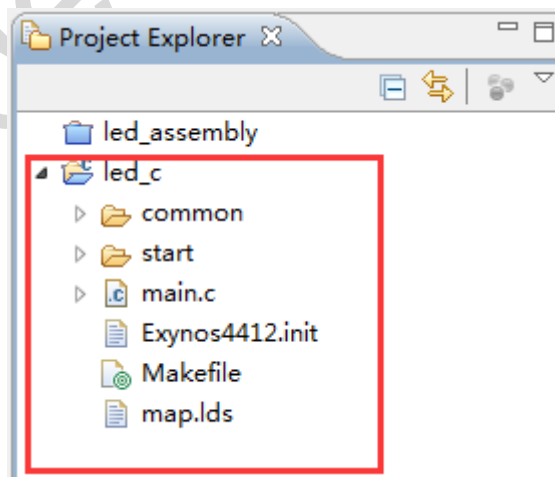


```

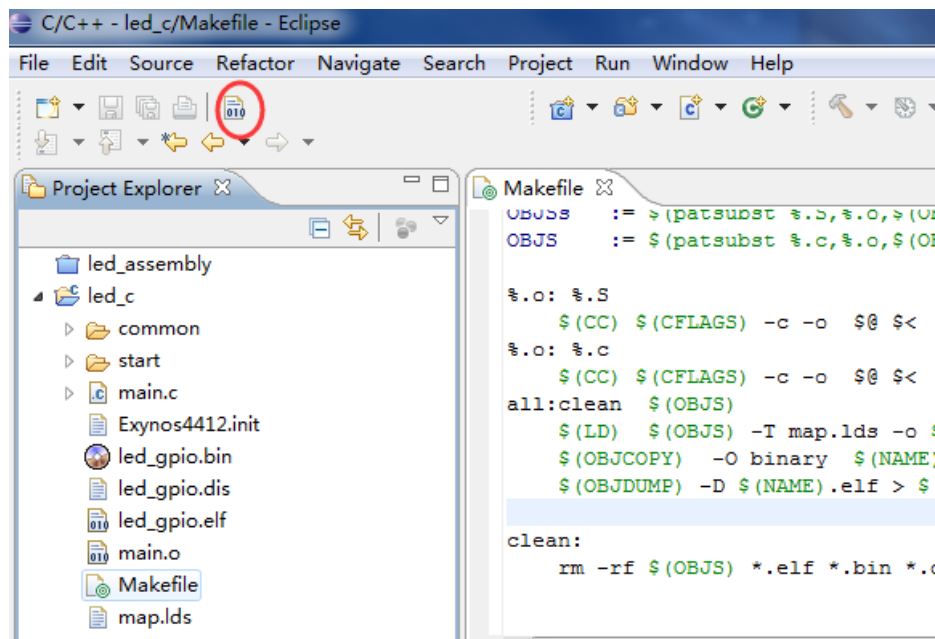
43 //Turn on LED2
44 GPX2.DAT |= 0x1 << 7;
45 mydelay_ms(500);
46
47 //Turn on LED3
48 GPX1.DAT |= 0x1;
49 //Turn off LED2
50 GPX2.DAT &= ~(0x1<<7);
51 mydelay_ms(500);
52
53 //Turn on LED5
54 GPF3.DAT |= (0x1 << 5);
55 //Turn off LED3
56 GPX1.DAT &= ~0x1;
57 mydelay_ms(500);
58
59 //Turn on LED4
60 GPF3.DAT |= (0x1 << 4);
61 //Turn off LED5
62 GPF3.DAT &= ~(0x1 << 5);
63 mydelay_ms(500);
64
65 //Turn off LED4
66 GPF3.DAT &= ~(0x1 << 4);
67 }
68 return 0;
69 }

```

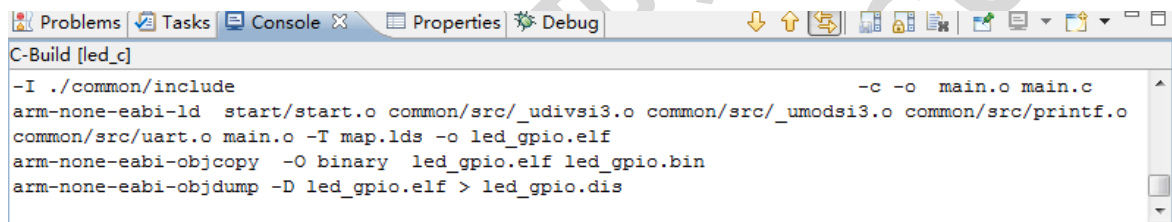
所有文件编辑完成后，保存文件，创建好的工程如下图所示：



点击如下图所示的编译图标（或者快捷键“Ctrl + B”）



编译成功后如下图所示



至此 ARM 裸板 C 工程已经创建完成，如暂时不使用该工程时我们可以 Close project，下次使用时再 Open project（同时只能有一个工程是打开的），该工程相关配置不变。工程调试过程请参照本文 1.6 章节。

1.5 导入一个已有工程

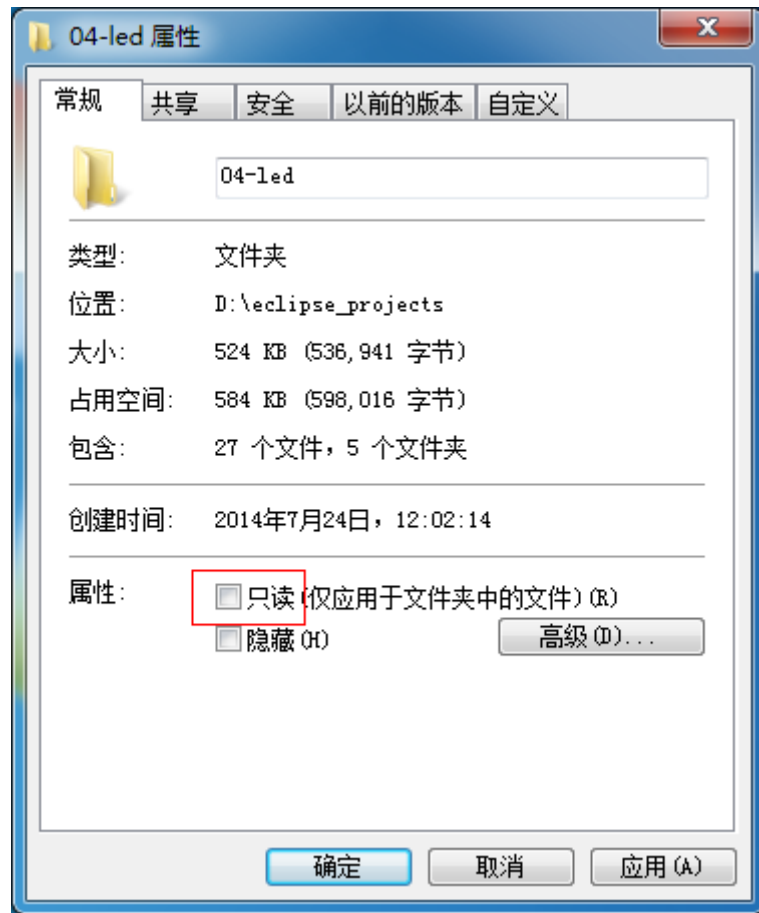
本文提供了相关项目实验源码，在实验中可以直接导入相应工程即可。

1. 打开 Eclipse 开发工具，在 Project Explorer 中添加 LED 工程

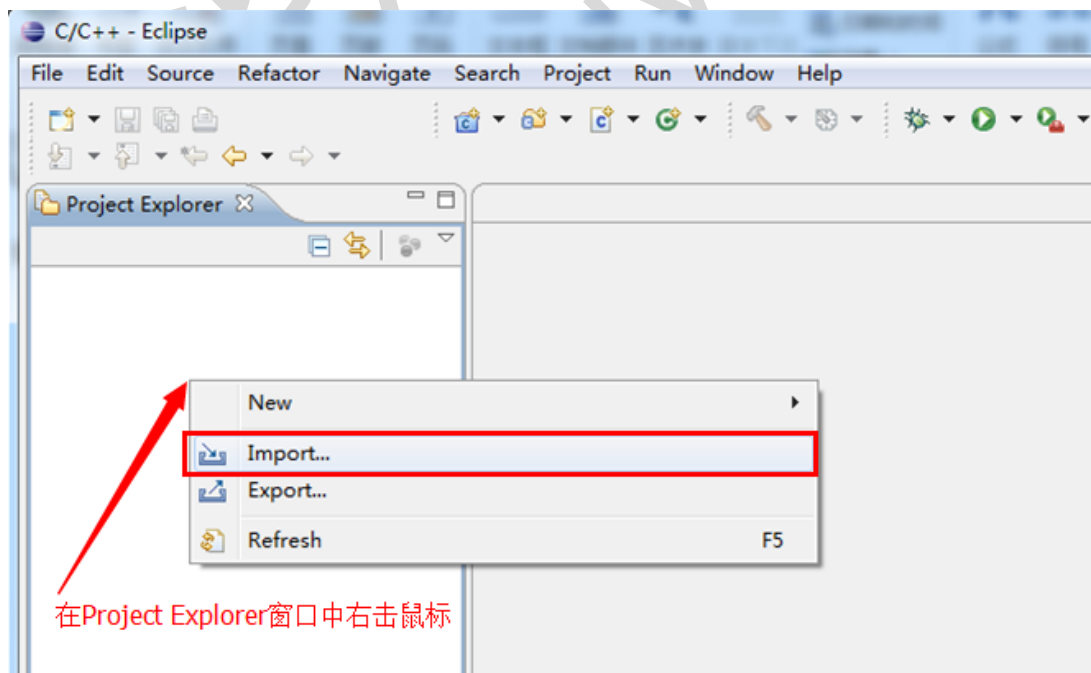
（注意：工程要放在英文路径下，不能有中文路径）

拷贝【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\04-led】到 eclipse 工作目录下。如：D:\eclipse_projects\FS4412 目录。

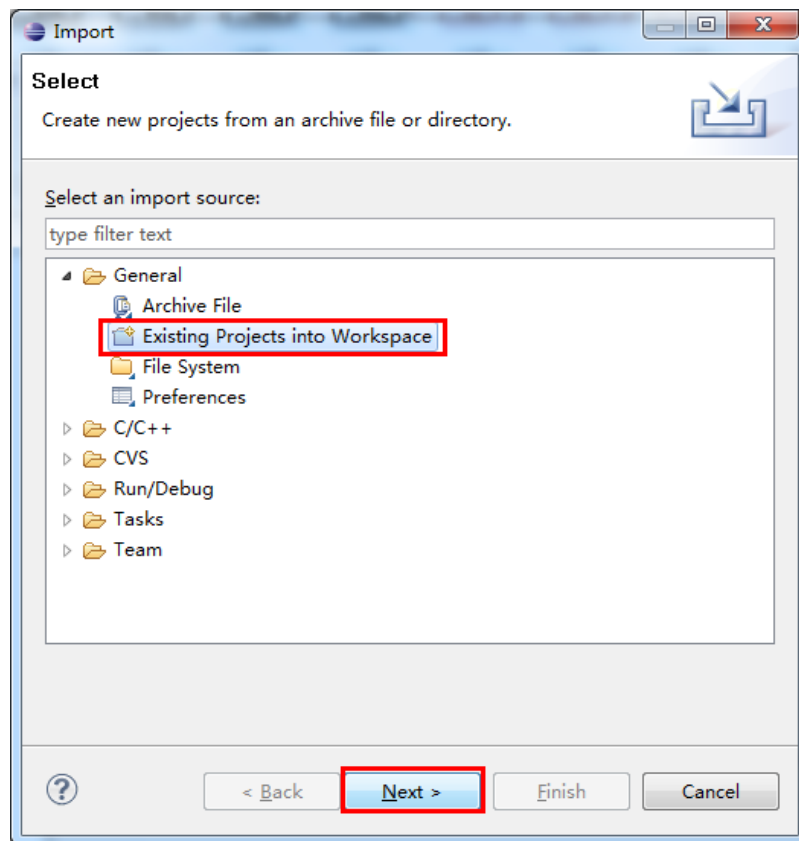
（注意：如果从光盘介质直接拷贝出来的目录可能会出现文件有只读属性的情况，建议查看文件夹的只读属性是否选中，确保在未勾选的状态下）



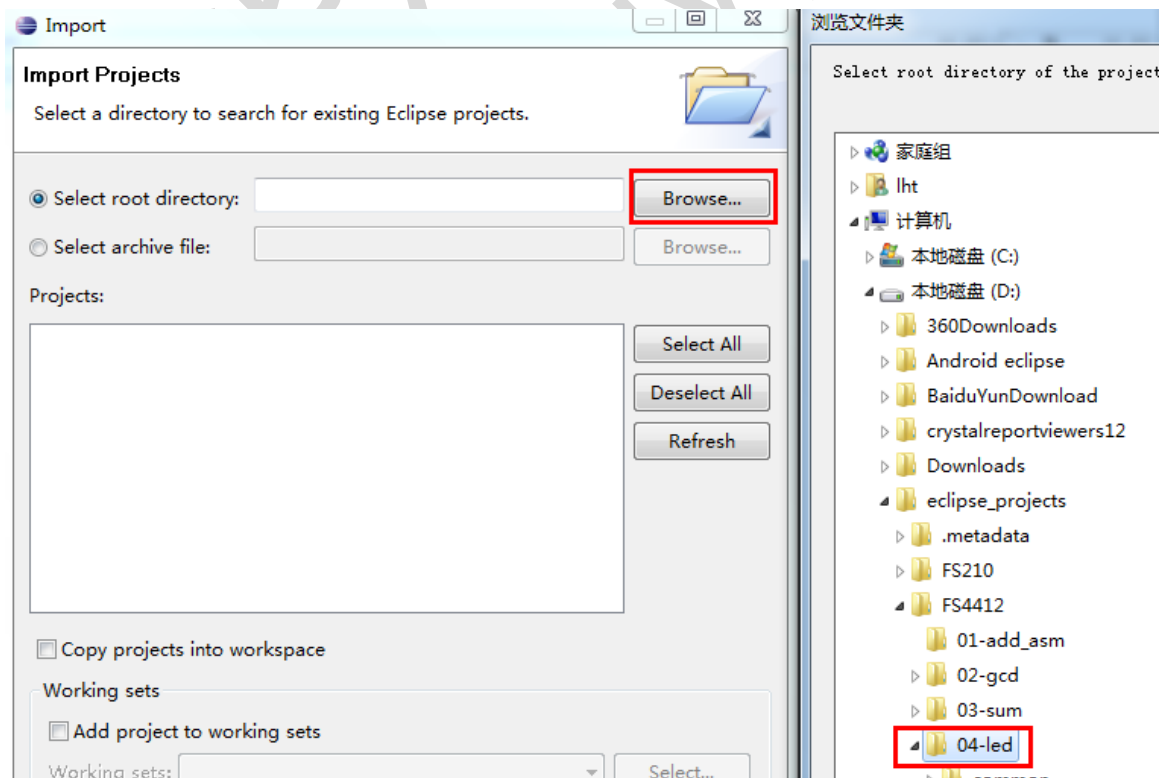
如图所示，在 Project Explorer 窗口中右击鼠标，选择“Import...”，如下图所示



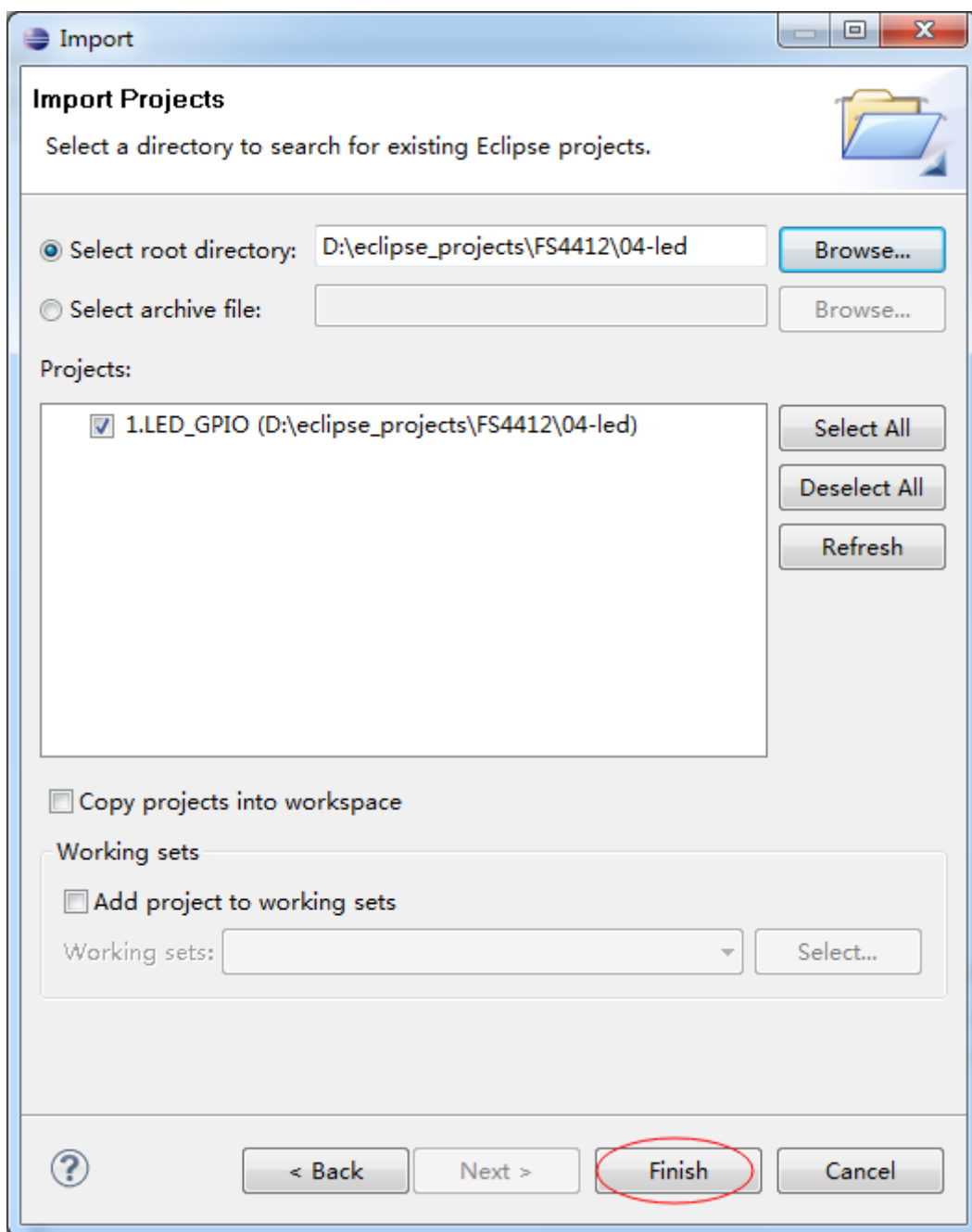
选择“Import...”后，出现如下图所示的窗口，选中“Existing Projects Into Workspace”然后点击“Next”



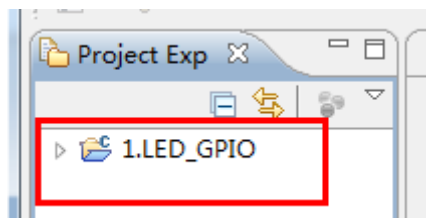
选择“Next”后出现如下窗口，点击“Browse...” → 出现“浏览文件夹”窗口，在“浏览文件夹”窗口中选中实验“04-led”后点击“确定”

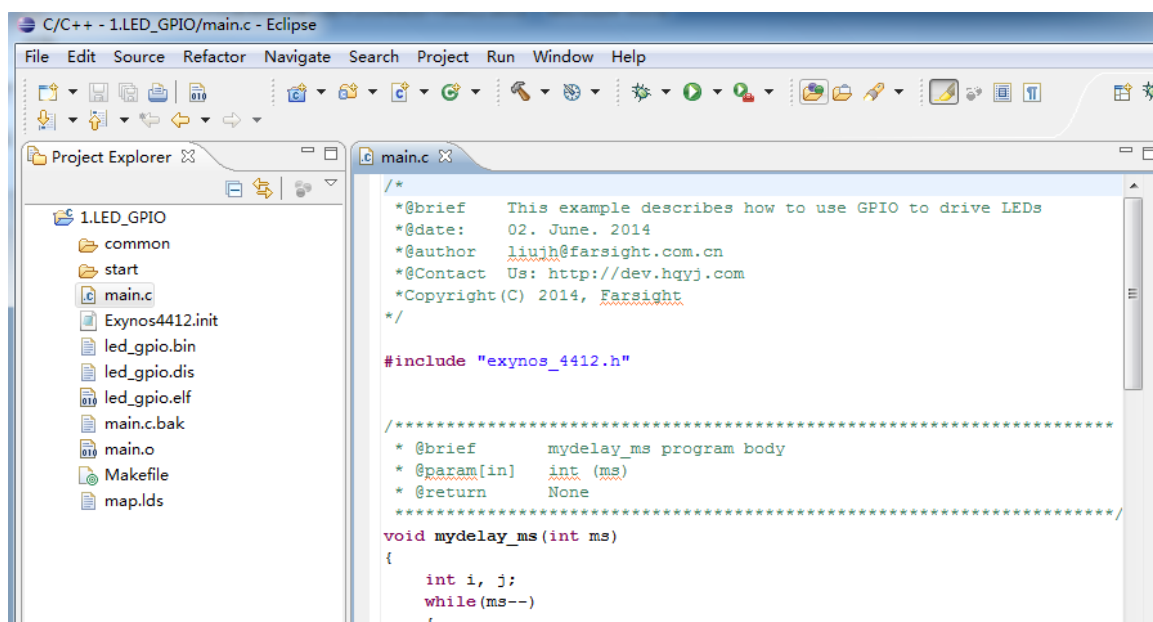


点击“确定”后出现如下所示窗口，直接点击“Finish”即可。



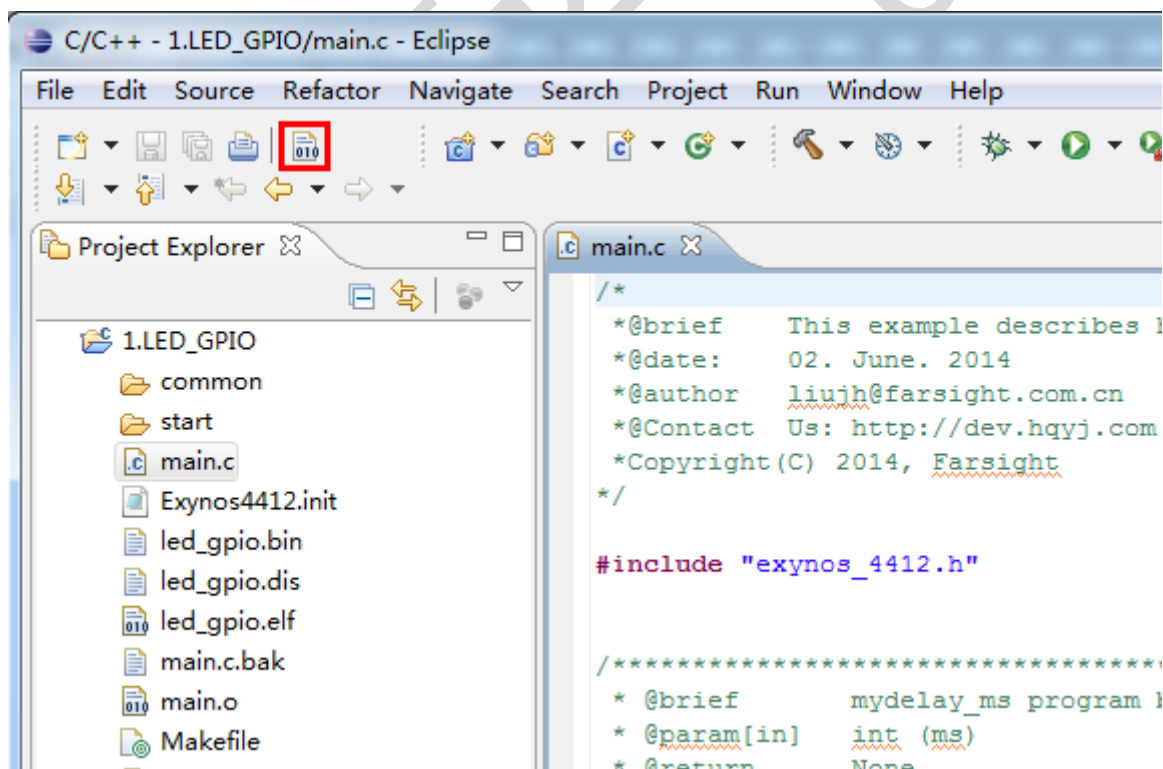
添加成功后可以在“Project Explorer”中看到“1.LED_GPIO”工程成功导入。





2. 编译程序

工程导入成功后，可以点击如下图所示的编译图标（或者快捷键“Ctrl + B”）



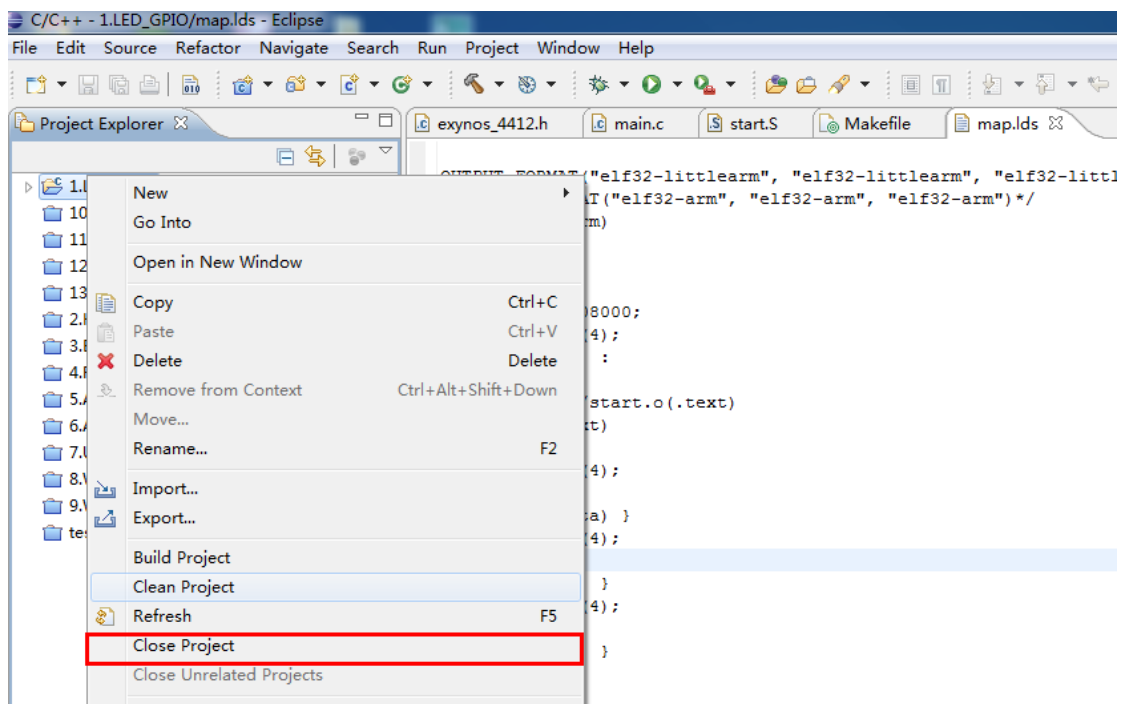
编译成功后如下图所示



```

C-Build [1.LED_GPIO]
common/src/printf.o common/src/printf.o
arm-none-eabi-gcc -g -O0 -mabi=apcs-gnu -mcpu=neon -mfloat-abi=softfp -fno-builtin
-nostdinc -I ./common/include -c -o
common/src/uart.o common/src/uart.o
arm-none-eabi-gcc -g -O0 -mabi=apcs-gnu -mcpu=neon -mfloat-abi=softfp -fno-builtin
-nostdinc -I ./common/include -c -o
main.o main.o
arm-none-eabi-ld start/start.o common/src/_udivsi3.o common/src/_umodsi3.o
common/src/printf.o common/src/uart.o main.o -T map.lds -o led_gpio.elf
arm-none-eabi-objcopy -O binary led_gpio.elf led_gpio.bin
    
```

至此已经成功导入已有工程,如暂时不使用该工程时我们可以 Close project,下次使用时再 Open project (同时只能有一个工程是打开的),该工程相关配置不变。[工程调试过程请参照本文 1.6 章节。](#)





1.6 调试工程

调试前的配置工作对于每个工程来说大同小异，所以本小结以之前导入的 04-led 工程为例。

1.6.1 配置 FS-JTAG 调试工具

打开 FS-JTAG 软件，在 Target 选项中选择 exynos4412，通信速率设置 200kHz。单击 Connect 按钮后，该按钮会变为 Disconnect，如图所示，即表示已经连接目标板。由于当前开发板还没有运行程序，仿真器无法获取相关信息，所以显示错误报告，是正常现象。

通信速率可以选择 200KHz，如果出现连接不稳定的情况，则将速率调制 100KHz。

注意：该软件在 win7 以上的 windows 操作系统中请使用管理员模式打开，否则会出现软件崩溃的现象。

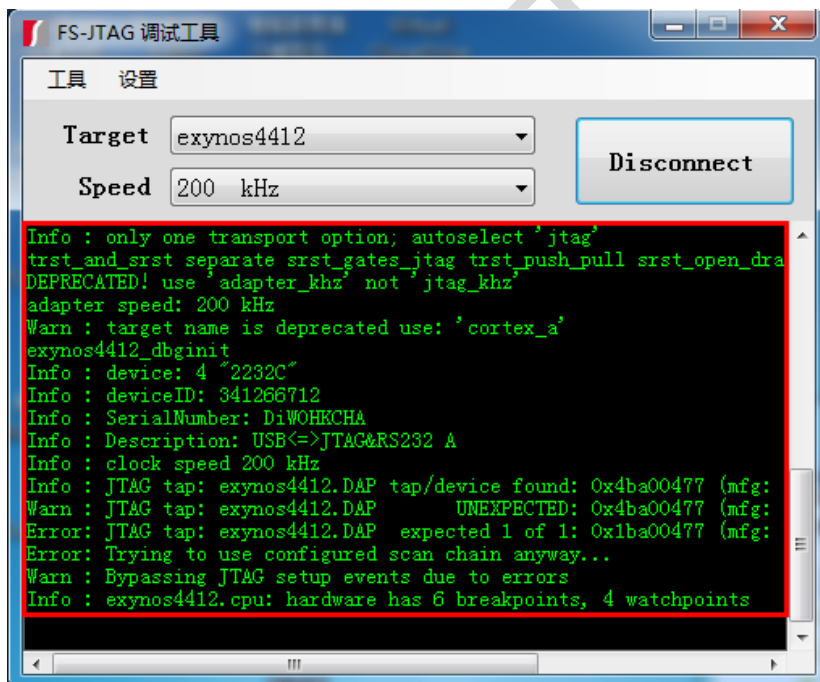
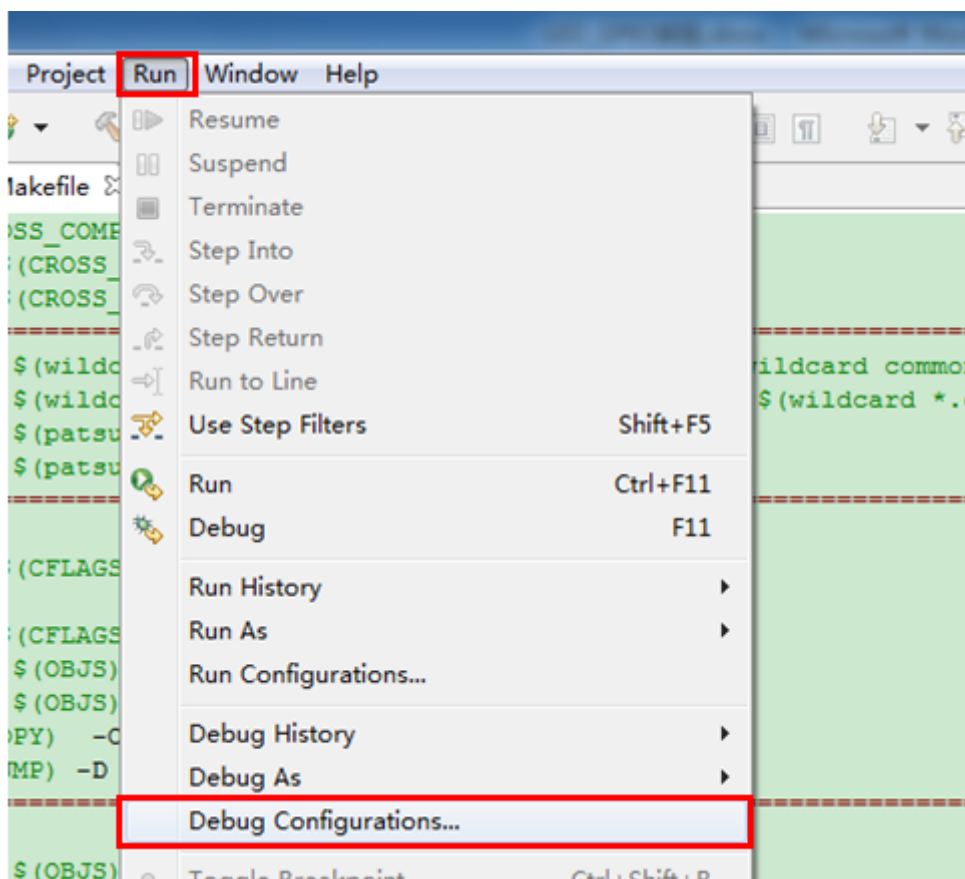


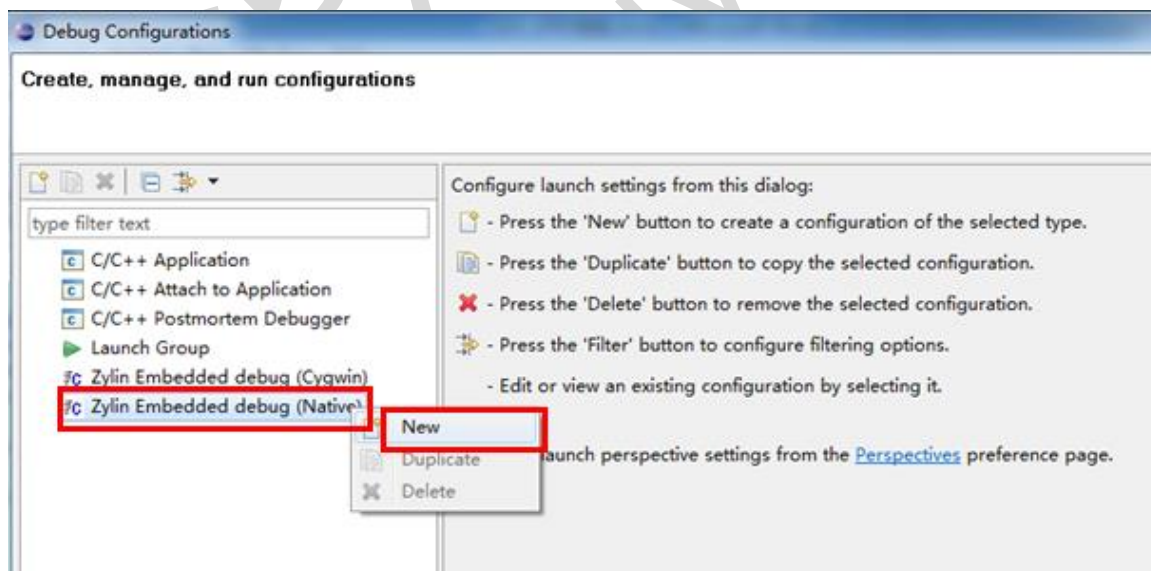
图 5-12 FS-JTAG 工具

1.6.2 配置调试工具

选中 “Run” → “Debug Configur...”

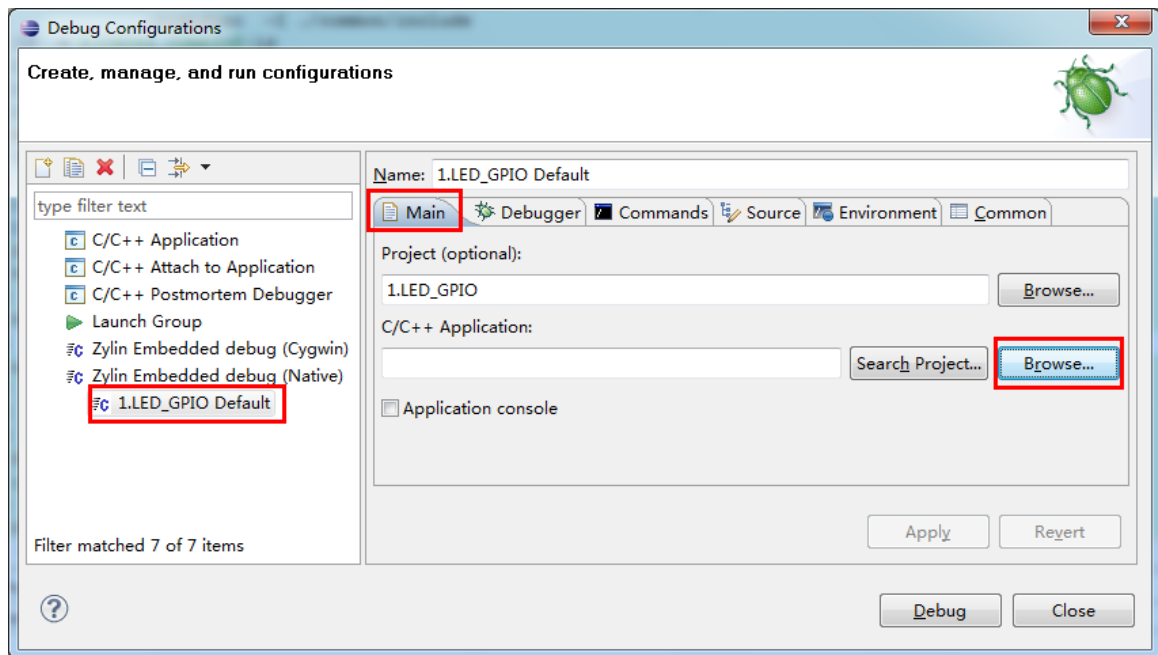


出现如下所示窗口，在“Zylin Embedded debug (Native)”选项上右击鼠标，在点击“New”选项



点击“New”选项后出现如下所示窗口

【main 选项卡】



然后再次点击“Main”选项卡中红色框内的“Browse”选项选择将要执行的程序，调试的程序格式为 **xx.elf** 的文件。找到 LED_GPIO 工程对应的目录，选中目录中的 **led_gpio.elf** 文件，然后选择“打开”，如下图所示。

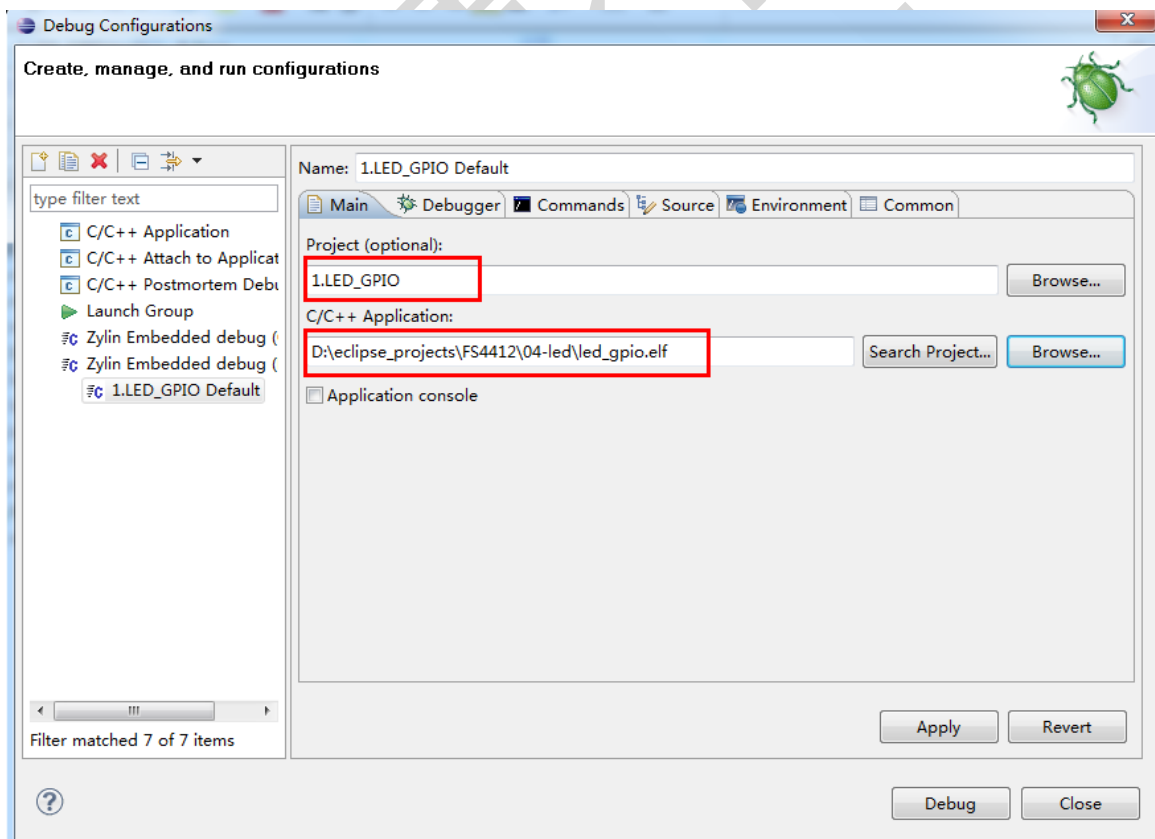


图 Main 选项卡



【Debugger 选项卡】

在 Debugger 选项卡中的 Main 子选项卡中的 GDB debugger 的框中单击“Browse”按钮选择前面安装的 arm-none-eabi-gdb.exe（这里选择自己安装 gcc 编译工具的安装目录【C:\Program Files\yagarto\bin】，在 GDB Command file 中选择自己工程目录下的 Exynos4412.init 文件，如图所示。

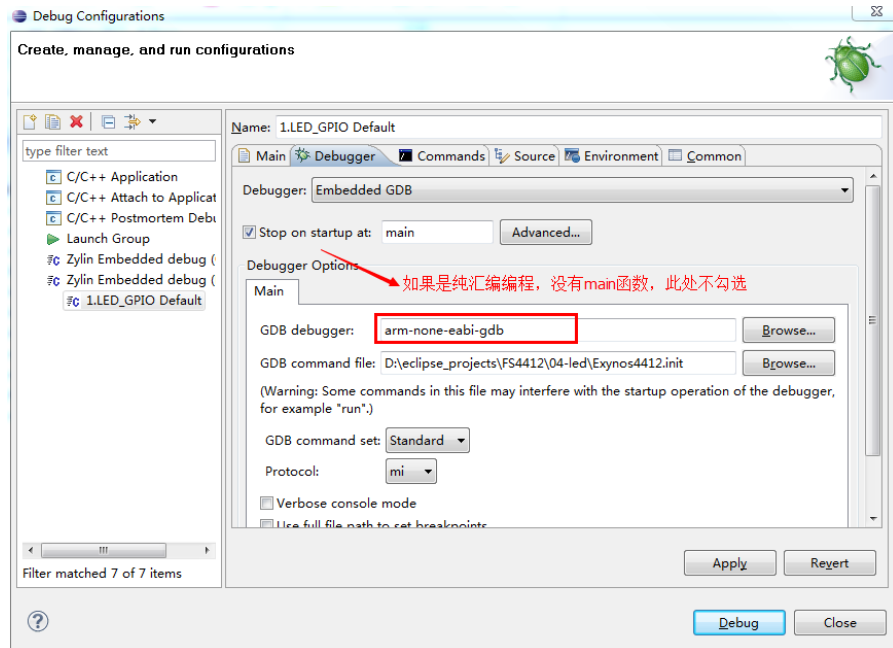
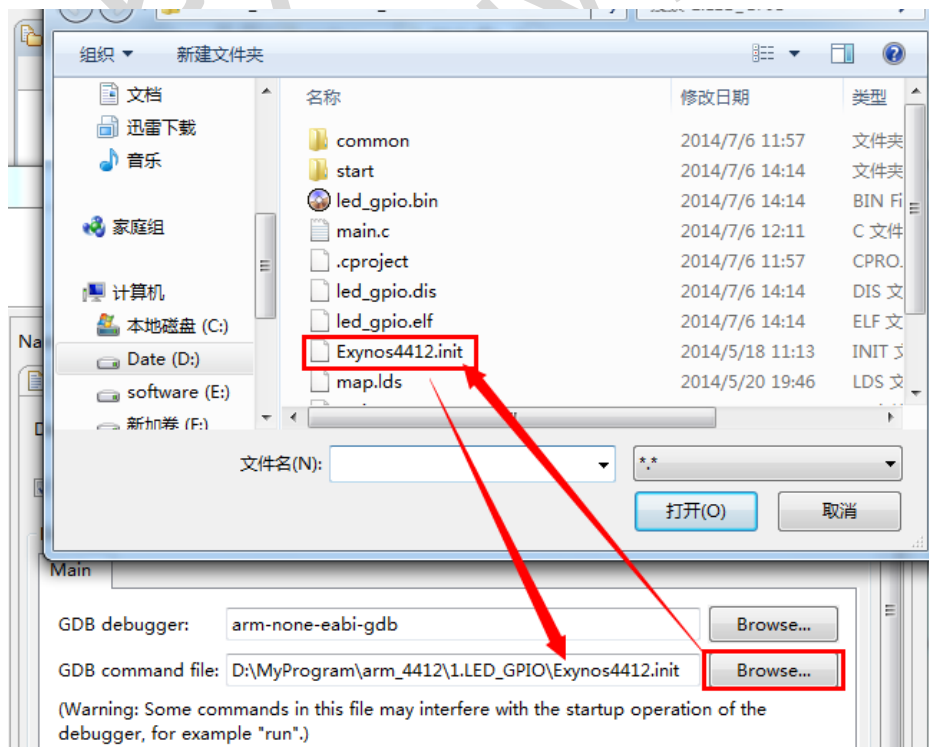


图 Debugger 选项



注：在 Debugger 选项卡中有一个 Stop on startup at: 选项，如调试工程为 c 工程，有 main 函数则勾选该选项、否则不勾选。



【Command 选项卡】

在 Command 选项中，如下图所示，在 “Initialize’ commands” 中添加 3 行命令

```
load
break main
c
```

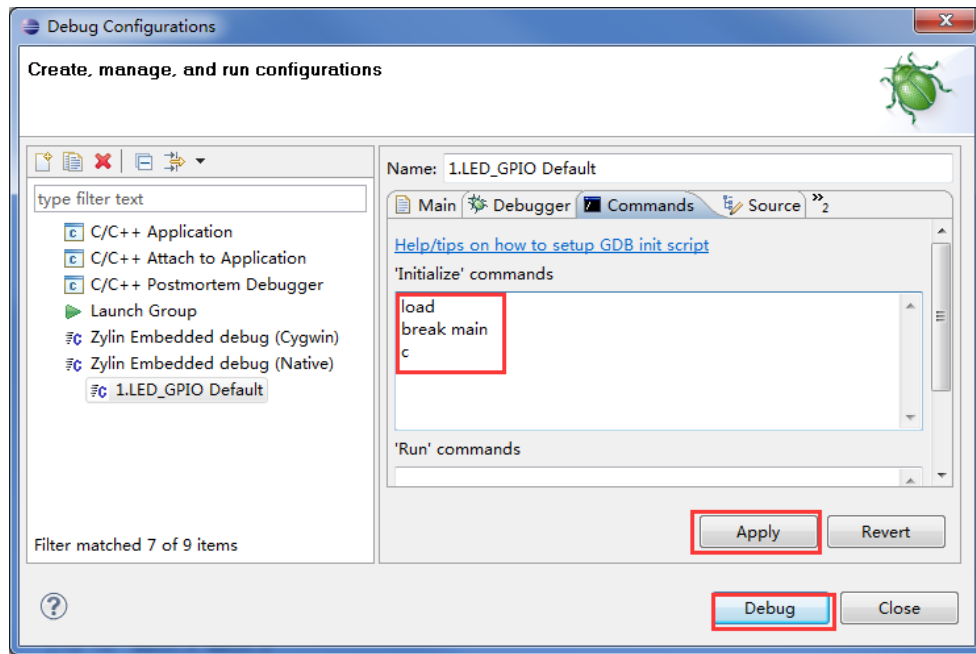
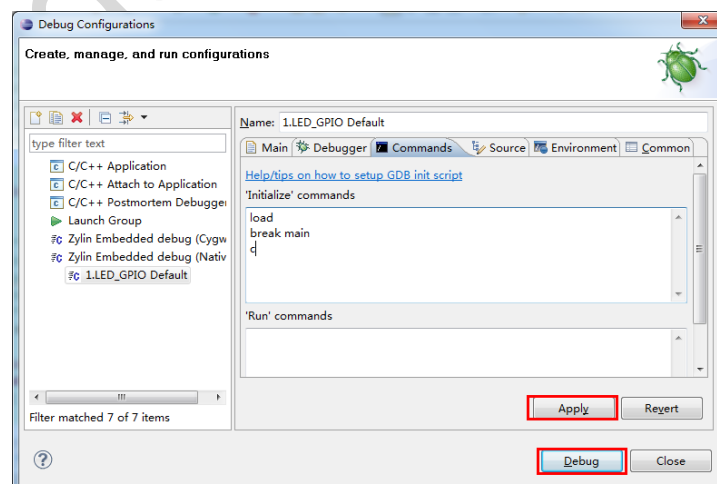


图 Commands 选项卡

注：在 Command 选项卡中，如我们配置的工程为 C 工程，则按如上步骤操作即可，如配置的工程为汇编工程，添加的 3 行命令应作如下修改：

```
load
break _start
c
```

添加完后点击如下图所示的 “Apply”，调试选项配置完成。再点击 “Debug” 选项进入就可以进行调试了。



调试主界面如图所示:

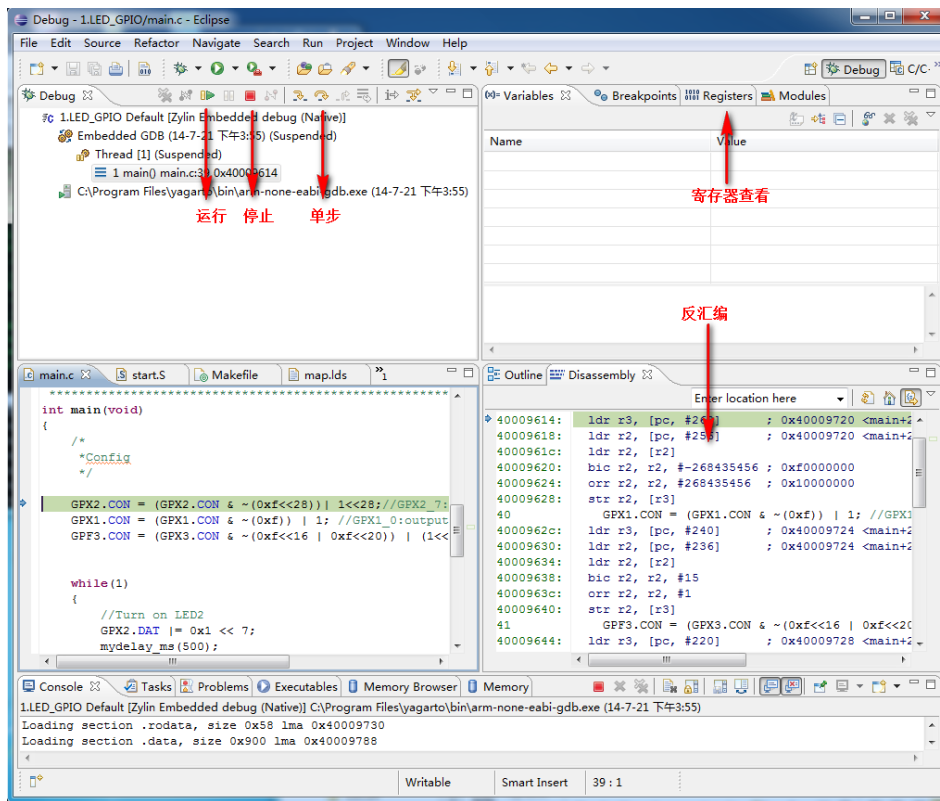
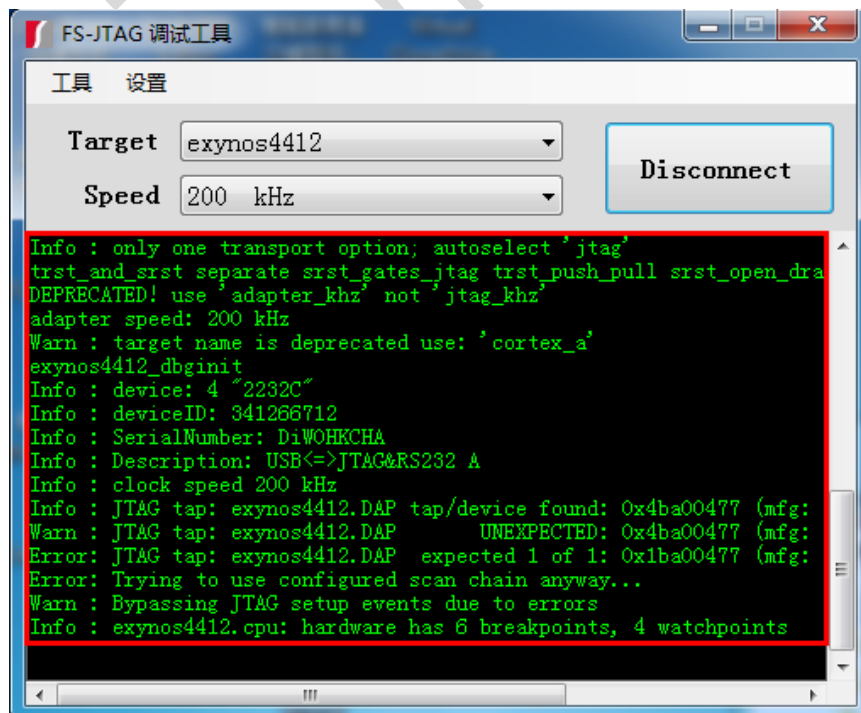

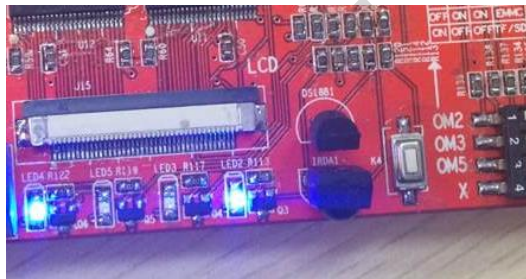
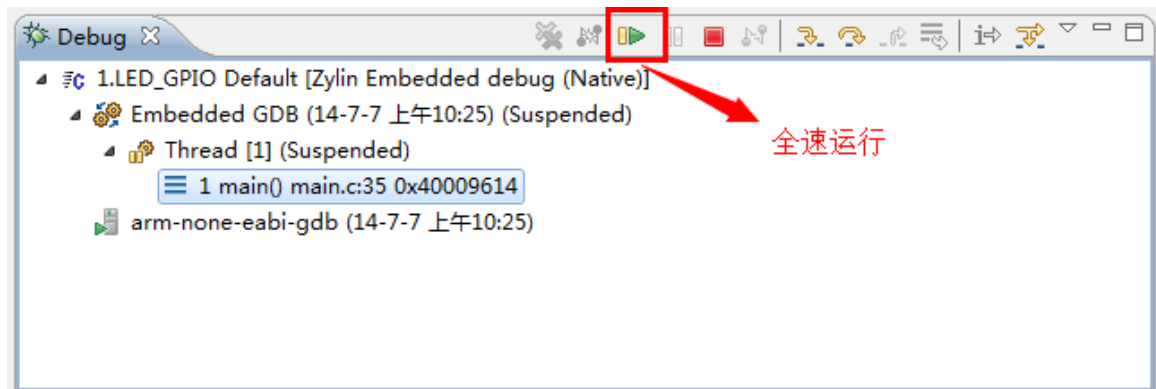


图 调试主界面

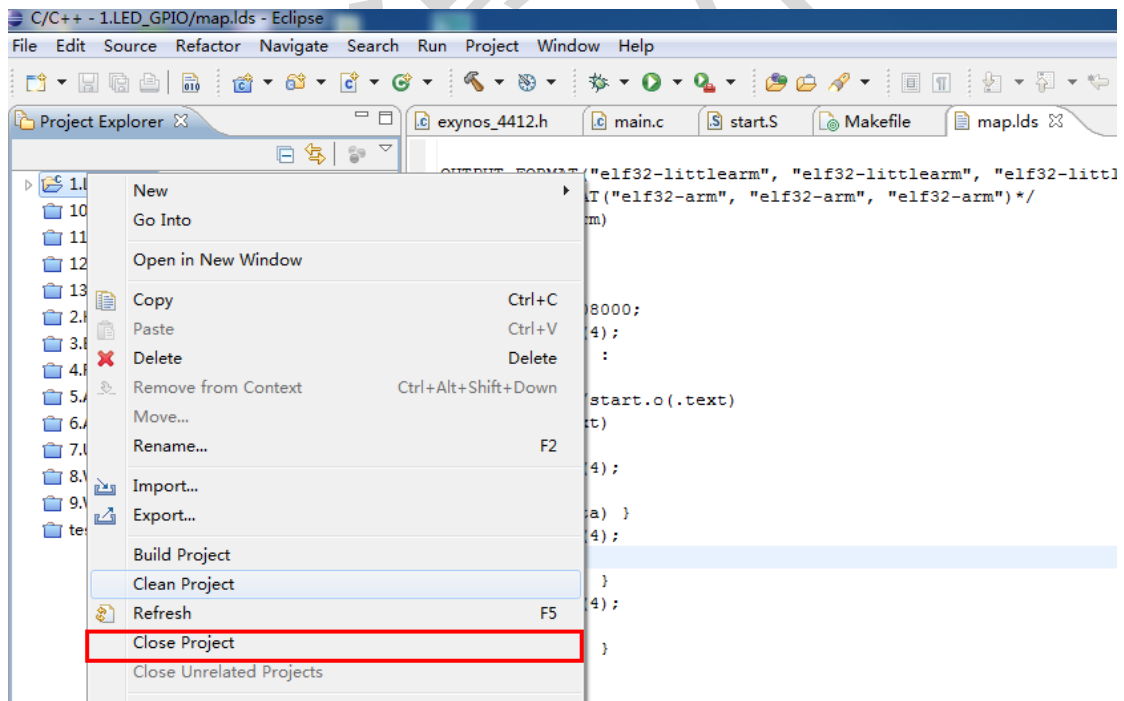
注意: 在进入 Debug 模式前, 必须保证此时 FS-Jtag 为下图所示的状态, 否则不能成功进入 Debug 状态



进入调试界面后，点击全速运行  的调试按钮，如果开发板上 led 灯开始闪烁，则表示整个开发环境搭建成功。



小技巧：导入的工程不用时可以 *Close project*, 下次使用时再 *Open Project*, 同时只能有一个工程是打开的。这样做的好处是：不用每次重新配置。



注意：在 Debug 调试时，要保持 Debug 框里只有一项在调试。



1.6.3 查看变量及寄存器的方法

可以在程序暂停状态状态下查看变量或寄存器的值。

如图所示窗口是用来查看函数变量的，可以看到当前 i 的值。

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

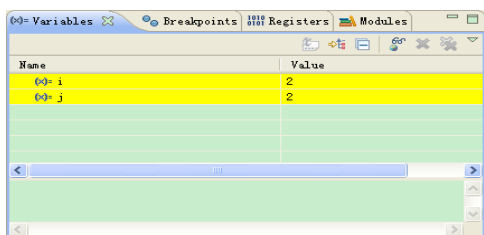


图 查看变量

Name	Value
Main	
r0	0
r1	0
r2	4294967295
r3	0
r4	3761242112
r5	1
r6	3978297344
r7	0
r8	3353853916
r9	0
r10	3353873048
r11	1
r12	3353591816
sp	0x20008314
lr	536907920
pc	0x20008000
fps	0
cpsr	1610612752

图 查看寄存器

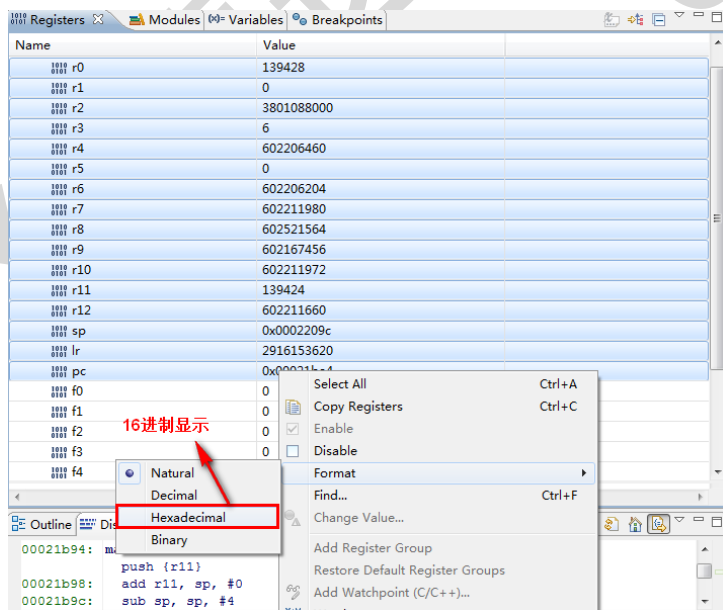
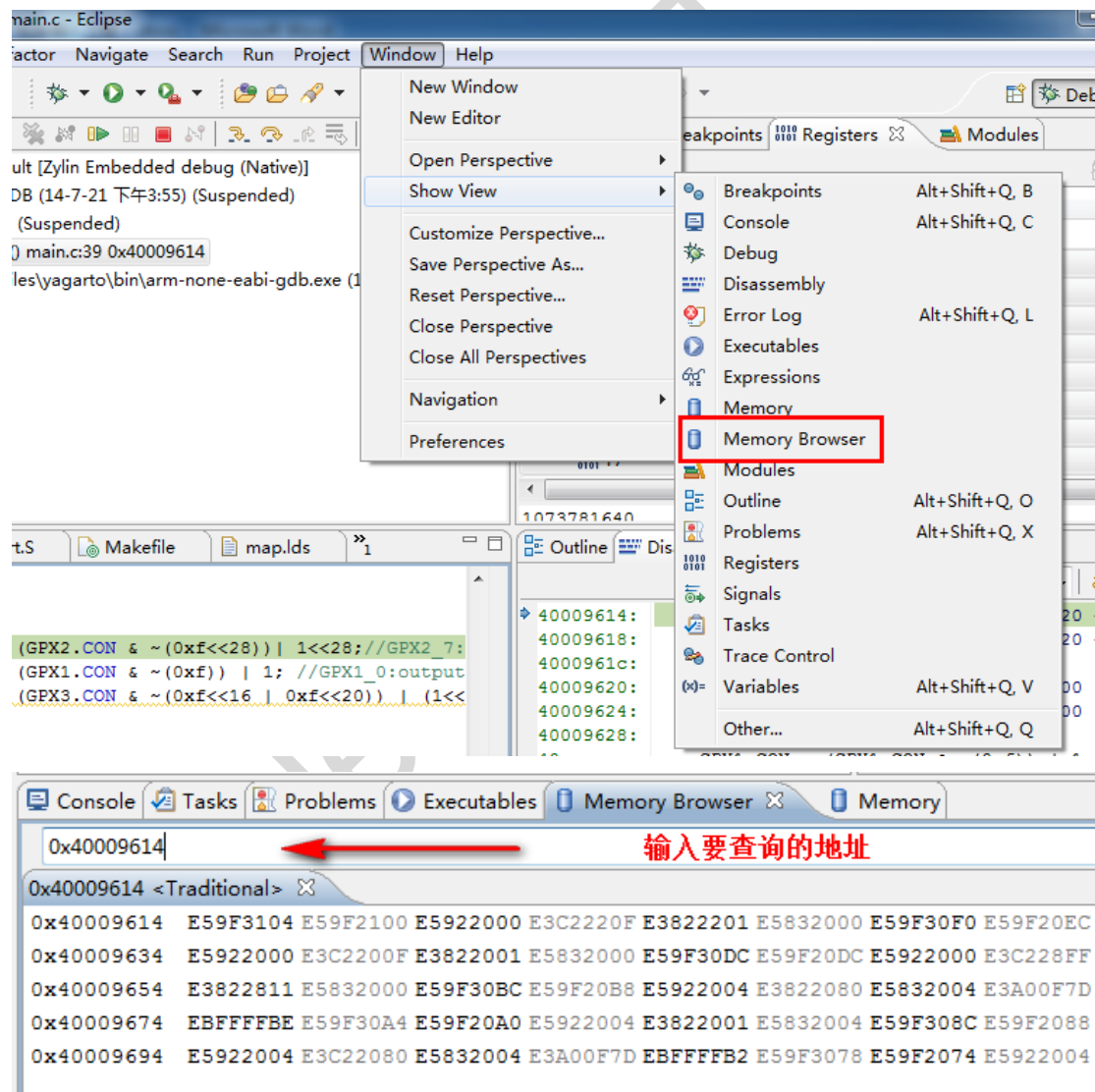


图 16 进制方式查看寄存器

1.6.4 断点设置方法



1.6.5 查看内存数据信息方法



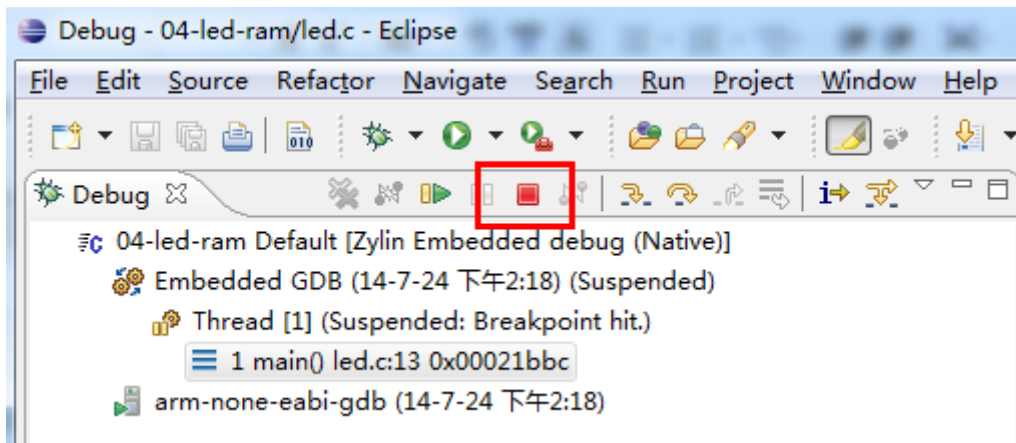
1.6.6 调试结束后的处理

- 1、一次调试结束后，需要停止调试

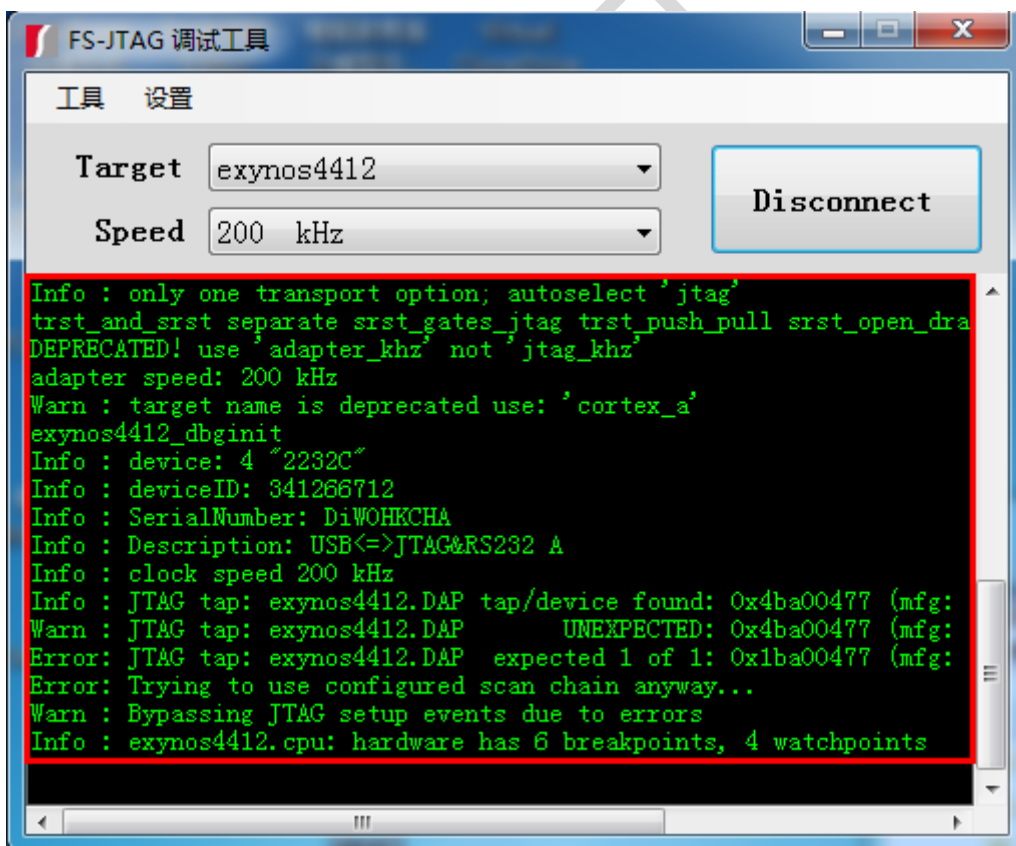
如果没有停止上次的程序，在下次 debug 时会出现调试不了的情况。此时在 Debug 窗口会出现多



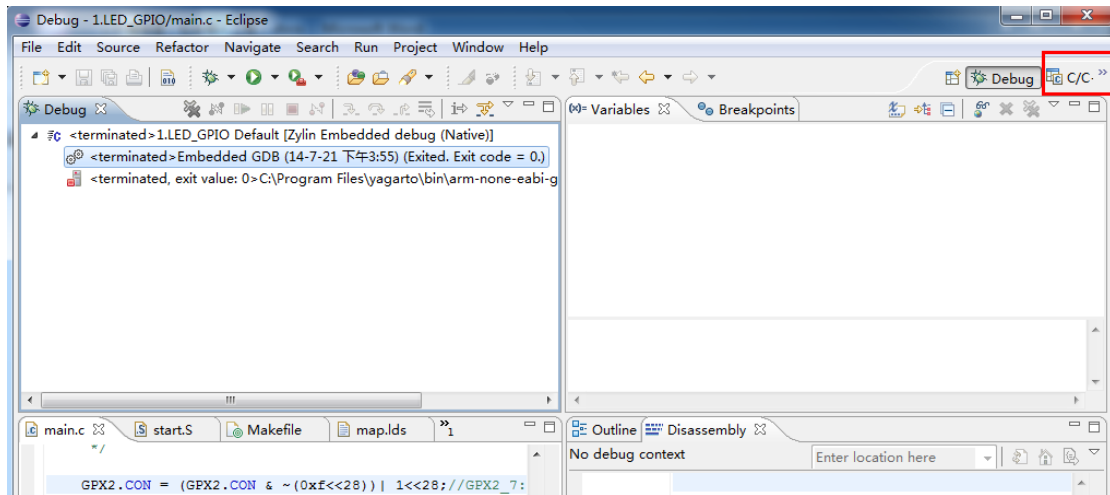
个待调试的任务。需要把所有任务都停止掉。



2、重新启动开发板到 uboot、重新连接 FS-JTAG 仿真器

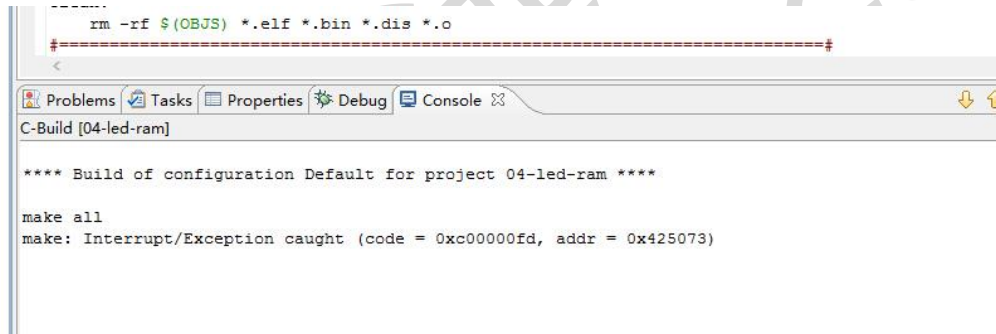


3、如果需要修改程序，需要切回到工程编辑界面

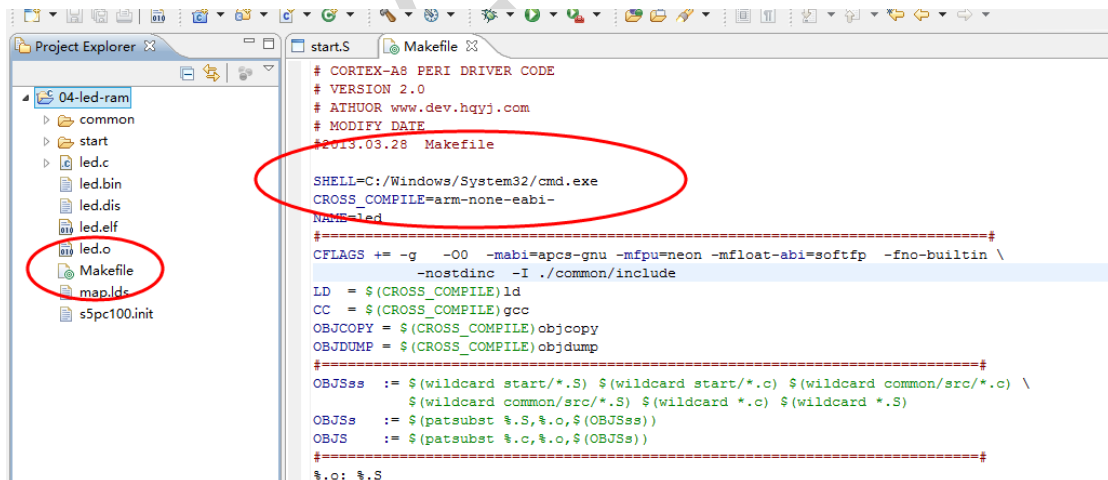


1.7 64 位 eclipse 编译出错问题及解决

问题一、



解决方法:



问题二、

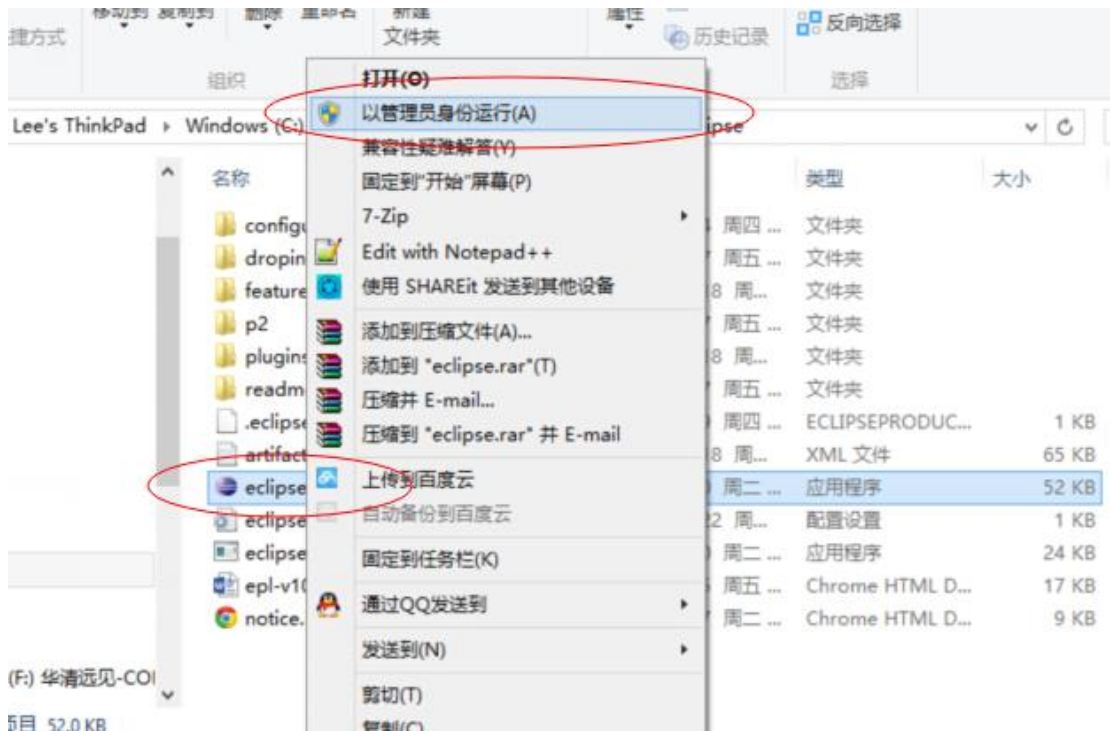
```

C-Build [04-led-ram]
rm -rf start/start.o common/src/pmic.o common/src/printf.o common/src/uart.o common/src/_udivsi3.o common/src/_umodsi3.o led.o
*.elf *.bin *.dis *.o
arm-none-eabi-gcc -g -O0 -mabi=apcs-gnu -mcpu=neon -mfloat-abi=softfp -fno-builtin -nostdinc -I ./common/include
-c -o start/start.o start/start.S
process_begin: CreateProcess(C:\Program Files (x86)\yagarto\bin\arm-none-eabi-gcc.exe, arm-none-eabi-gcc -g -O0 -mabi=apcs-gnu
-mcpu=neon -mfloat-abi=softfp -fno-builtin -nostdinc -I ./common/include -c -o start/start.o start/start.S, ...) failed.
make (e=740): 请求的操作需要提升。

make: *** [start/start.o] Error 740
    
```

解决方法:

管理员模式打开 eclipse



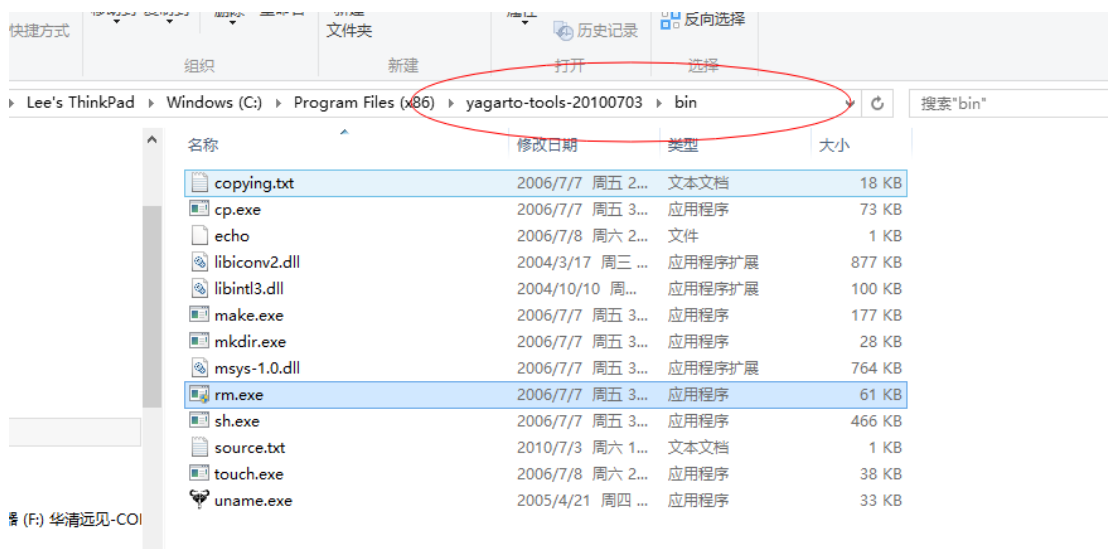
问题三、

```

C-Build [04-led-ram]
**** Build of configuration Default for project 04-led-ram ****

make all
rm -rf start/start.o common/src/pmic.o common/src/printf.o common/src/uart.o common/src/_udivsi3.o common/src/_umodsi3.o led.o
*.elf *.bin *.dis *.o
AllocationBase 0x0, BaseAddress 0x71590000, RegionSize 0x30000, State 0x10000
C:\Program Files (x86)\yagarto-tools-20100703\bin\rm.exe: *** Couldn't reserve space for cygwin's heap, Win32 error 0
make: *** [clean] Error 1
    
```

解决方法:



此路径下的所有可执行文件兼容性设置为 xp

1.8 本章小结

本章主要介绍了如何编写 GNU-ARM 汇编风格的程序, 以及如何基于 EXYNOS4412 在 Eclipse 下进行调试, 并且介绍了 FS-JTAG 的详细用法。本书后面章节的大部分实验都是基于这个环境的。工欲善其事, 必先利其器, 所以必须熟练掌握环境的使用。

1.9 练习题

1. 熟悉 Eclipse 开发环境。
2. 新建一个工程, 编写一个汇编程序实现 $3+13=16$ 的操作。



第 2 章 嵌入式 ARM 技术概论

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

- ARM 体系结构的技术特征及发展。
- ARM 微处理器简介。
- ARM 微处理器结构。
- ARM 微处理器的应用选型。
- Cortex-A9 内部功能及特点。
- 数据类型。
- Cortex-A9 存储系统。
- 流水线。
- 寄存器组织 S。
- 程序状态寄存器。
- SAMSUNG EXYNOS4412 处理器介绍。

2.1 ARM 体系结构的技术特征及发展

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

2.1.1 ARM 公司简介

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

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

ARM 公司是专门从事基于 RISC 技术芯片设计开发的公司,作为嵌入式 RISC 处理器的知识产权 IP 供应商,公司本身并不直接从事芯片生产,而是靠转让设计许可由合作公司生产各具特色的芯片,世界各大半导体生产商从 ARM 公司购买其设计的 ARM 微处理器核,根据各自不同的应用领域,加入适当的外围电路,从而形成自己的 ARM 微处理器芯片进入市场,利用这种合伙关系,ARM 很快成为许多全球性 RISC 标准的缔造者。目前,全世界有几十家大的半导体公司都使用 ARM 公司的授权,其中包括 Intel、IBM、SAMSUNG、LG 半



导体、NEC、SONY、PHILIP 等公司，这也使得 ARM 技术获得更多的第三方工具、制造厂商、软件的支持，又使整个系统成本降低，使产品更容易进入市场并被消费者所接受，更具有竞争力。

2.1.2 ARM 技术特征

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

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

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

1、嵌入式 RISC 微处理器

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

2、嵌入式 CISC 微处理器

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

表 RISC 和 CISC 之间主要的区别

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

2.1.3 ARM 体系架构的发展

在讨论 ARM 体系结构前，先解释一下体系结构的定义。

体系架构定义了指令集（ISA）和基于这一体系结构下处理器的编程模型。基于同种体系结构可以有多种处理器，每个处理器性能不同，所面向的应用不同，每个处理器的实现都要遵循这一体系结构。ARM



体系结构为嵌入系统发展商提供很高的系统性能，同时保持优异的功耗和面积效率。

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

1、v1 架构

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

2、v2 架构

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

3、v3 架构

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

4、v4 架构

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

5、v5 架构

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

属于 v5TE（支持 Thumb、DSP 指令）体系结构的处理器（核）有 ARM9E、ARM9E-S（ARM9E 可综合版本）、ARM946（ARM9E 核的处理器）、ARM966（ARM9E 核的处理器）、ARM10E、ARM1020E（ARM10E 核处理器）、ARM1022E（ARM10E 核的处理器）和 Xscale（Intel 公司产品）。

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

6、v6 架构



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

7、v7 架构

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

Cortex 系列处理器是基于 ARMv7 架构的,分为 Cortex-M3、Cortex-R 和 Cortex-A3 类。

本章的 2.28 节将会例举一些 Cortex 的特性。

8、v8 架构

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

2.2 ARM 微处理器简介

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

表总结了 ARM 各系列处理器所包含的不同类型。

表 ARM 各系列处理器所包含的不同类型

ARM 系列	包含类型
ARM9/9E 系列	ARM920T
	ARM922T
	ARM926EJ-S
	ARM940T
	ARM946E-S
	ARM966E-S
	ARM968E-S
向量浮点运算 (Vector Floating Point) 系列	VFP9-S
	VFP10
ARM10E 系列	ARM1020E
	ARM1022E



	ARM1026EJ-S
ARM11 系列	ARM1136J-S ARM1136JF-S ARM1156T2(F)-S ARM1176JZ(F)-S ARM11 MPCore
Cortex 系列	Cortex-A Cortex-R Cortex-M
SecurCore 系列	SC100 SC110 SC200 SC210
其他合作伙伴产品	StrongARM XScale MBX

本节简要介绍 ARM 各个系列处理器的特点。

2.2.1 ARM9 处理器系列

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

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

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

ARM9 系列处理器主要应用于下面一些场合：

- (1) 下一代无线设备，包括视频电话和 PDA 等。
- (2) 数字消费品，包括机顶盒、家庭网关、MP3 播放器和 MPEG-4 播放器。
- (3) 成像设备，包括打印机、数码照相机和数码摄像机。
- (4) 汽车、通信和信息系统。

2.2.2 ARM9E 处理器系列

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

ARM946E-S 包括 TCM（Tightly Coupled Memory，紧耦合存储器）、Cache 和一个 MPU。TCM 和 Cache



的大小可配置。该处理器是针对要求有确定的实时响应的嵌入式控制而设计的。ARM966E-S 有可配置的 TCM，但没有 MPU 和 Cache 扩展。

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

ARM9E 系列处理器主要应用于下面一些场合：

- (1) 下一代无线设备，包括视频电话和 PDA 等。
- (2) 数字消费品，包括机顶盒、家庭网关、MP3 播放器和 MPEG-4 播放器。
- (3) 成像设备，包括打印机、数码照相机和数码摄像机。
- (4) 存储设备，包括 DVD 或 HDD 等。
- (5) 工业控制，包括电机控制等。
- (6) 汽车、通信和信息系统的 ABS 和车体控制。
- (7) 网络设备，包括 VoIP、WirelessLAN 等。

2.2.3 ARM11 处理器系列

ARM1136J-S 发布于 2003 年，是针对高性能和高能效而设计的。ARM1136J-S 是第一个执行 ARMv6 架构指令的处理器。它集成了一条具有独立的 Load/Store 和算术流水线的 8 级流水线。ARMv6 指令包含了针对媒体处理的单指令流多数据流扩展，采用特殊的设计改善视频处理能力。

2.2.4 SecurCore 处理器系列

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

- (1) 支持 ARM 指令集和 Thumb 指令集，以提高代码密度和系统性能。
- (2) 采用软内核技术以提供最大限度的灵活性，可以防止外部对其进行扫描探测。
- (3) 提供了安全特性，可以抵制攻击。
- (4) 提供面向智能卡和低成本的存储保护单元 MPU。
- (5) 可以集成用户自己的安全特性和其他的协处理器。

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

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

2.2.5 StrongARM 和 Xscale 处理器系列

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



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

2.2.6 MPCore 处理器系列

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

2.2.7 Cortex 处理器系列

1、ARM Cortex 处理器技术特点

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

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

2、ARM Cortex-M3 处理器技术特点

ARM Cortex-M3 处理器是为存储器和处理器的尺寸对产品成本影响极大的各种应用专门开发设计的。它整合了多种技术, 减少了内存使用, 并在极小的 RISC 内核上提供低功耗和高性能, 可实现由以往的代码向 32 位微控制器的快速移植。ARM Cortex-M3 处理器是使用最少门数的 ARM CPU, 相对于过去的设计大大减小了芯片面积, 可减小装置的体积或采用更低成本的工艺进行生产, 仅 33000 门的内核性能可达



1.2DMIPS/MHz。此外，基本系统外设还具备高度集成化特点，集成了许多紧耦合系统外设，合理利用了芯片空间，使系统满足下一代产品的控制需求。

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

为微控制器应用而开发的 Cortex-M3 拥有以下性能：

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

3、ARM Cortex-R4 处理器技术特点

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

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

相比于前几代的处理器，Cortex-R4 处理器高效率的设计方案，使其能以更低的时钟达到更高的性能；经过最佳化设计的 Artisan Mctro 内存，可进一步降低嵌入式系统的体积与成本。处理器搭载一个先进的微架构，具备双指令发送功能，采用 90nm 工艺并搭配 Artisan Advantage 程序库的组件，底面积不到 1mm²，耗电最低低于 0.27mW/MHz，并能提供超过 600 DMIPS 的性能。



Cortex-R4 处理器在各种安全应用上加入容错功能和内存保护机制，支持最新版 OSEK

实时操作系统；支持 RealView Develop 系列软件开发工具、RealView Create 系列 ESL 工具与模块，以及 Core Sight 除错与追踪技术，协助设计者迅速开发各种嵌入式系统。

4、ARM Cortex-A9 处理器技术特点

ARM Cortex-A9 处理器是一款适用于复杂操作系统及用户应用的应用处理器，支持智能能源管理（Intelligent Energy Manger, IEM）技术的 ARM Artisan 库及先进的泄漏控制技术，使得 Cortex-A9 处理器实现了非凡的速度和功耗效率。在 32nm 工艺下，ARM Conex-A9 Exynos 处理器的功耗大大降低，能够提供高性能和低功耗。它第一次为低费用、高容量的产品带来了台式机级别的性能。

Conex-A9 处理器是第一款基于 ARMv7 多核架构的应用处理器，使用了能够带来更高性能、更低功耗和更高代码密度的 Thumb-2 技术。它首次采用了强大的 NEON 信号处理扩展集，为 H.264 和 MP3 等媒体编解码提供加速。Cortex-A9 的解决方案还包括 Jazelle-RCTJava 加速技术，对实时（JTT）和动态调整编译（DAC）提供最优化，同时减少内存占用空间高达 3 倍。该处理器配置了先进的超标量体系结构流水线，能够同时执行多条指令。处理器集成了一个可调尺寸的二级高速缓冲存储器，能够同高速的 16KB 或者 32KB 一级高速缓冲存储器一起工作，从而达到最快的读取速度和最大的吞吐量。新处理器还配置了用于安全交易和数字版权管理的 Trust Zone 技术，以及实现低功耗管理的 IEM 功能。

Cortex-A9 处理器使用了先进的分支预测技术，并且具有专用的 NEON 整型和浮点型流水线进行媒体和信号处理。

Cortex A9 时代三星一共发布了两代产品，第一代是 Galaxy S II 和 MX 采用的 Exynos 4210，第二代有两款，一款是双核的 Exynos 4212，一款是四核的 Exynos 4412。第一代产品采用的是 45nm 工艺制造，由于三星的 45nm 工艺在业内是比较落后的，虽然通过种种手段将 Exynos 4210 的频率提升到了 1.4GHz，但这么做的代价也是非常明显的——功耗激增（这点在 MX 上我们也看到了）。总体而言，Exynos 4212 和 4412 在架构上和 Exynos 4210 并没有区别，大体上的硬件配置也是一样的，最大的区别就在于 Exynos 4212/4412 采用了三星最新的 32nm HKMG 工艺。

2.2.8 最新 ARM 应用处理器发展现状

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

- NVIDIA（英伟达）的 Tegra 2 双核处理器及 Tegra 3 四核处理器，已经应用在摩托罗拉双核智能手机 ME860 及 LG Optimus 2X 手机上。
- 三星 Exynos 4412，基于 CORTEX-A9 的双核处理器，目前应用在三星公司推出的 GALAXY SII 智能手机。
- TI 的 OMAP4430 及 OMAP4460 双核 ARM 处理芯片，已应用在 LG Optimus 3D 手机。
- 高通 MSM8260、MSM8660（1.5G）、MSM8960（1.7G）双核处理器及 APQ8060（2.5G）四核心处理器。目前应用的代表有 HTC 的金字塔（Pyramid）双核智能手机，还有国内的小米手机。
- 苹果 A8 64 位处理器，典型代表是 iPhone6。



- (2) 内嵌的图形显示芯片越来越强劲。
- Mali 系列由 ARM 出品, Mali-400、Mali-T658 于 2011 年 11 月推出, 支持 OpenGL ES 2.0 和 DirectX 接口, 可从单核扩展到四核, 可提供卓越的二维和三维加速性能。
- PowerVR SGX 系列由 Imagination Technologies 公司出品, 包括 PowerVR SGX530/535/540/543MP, 支持 DirectX 9、SM3.0 和 OpenGL 2.0。
- SGX535 被苹果公司的 iPhone4 和 iPad 采用, 而 SGX540 性能更加强劲, 在三星 Galaxy Tab 与魅族 M9 上采用。SGX543MP 作为新一代最强新品, 目前已成为苹果 iPad 2 (SGX543MP2/双核) 和索尼 NGP (SGX543MP4/四核) 的图形内核。
- Adreno 系列由高通公司出品, 主要配合 Snapdragon CPU 使用。旗下典型方案有 Adreno200/205/220/300。
- 在图形处理单元上, Tegra 3 从之前 Tegra 2 的 8 核心图形单元升级到 12 核心单元, NVIDIA 官方宣布将有 3 倍的图形性能提升。这 12 个处理核心的 GeForce GPU 专门为下一代移动游戏而打造(完全兼容现有 Tegra 2 游戏), 支持更好的动态光影、物理效果和高分辨率环境。典型处理器方案有 NVIDIA Tegra 2 和 NVIDIA Tegra 3。

(2) 支持大 RAM, 支持大数据量的存储介质。

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

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

(4) 提升 Camera 性能。最高支持 3200 万像素摄像头

2.3 ARM 微处理器结构

ARM 内核采用 RISC 体系结构。ARM 体系结构的主要特征如下:

- (1) 采用大量的寄存器, 它们都可以用于多种用途。
- (2) 采用 Load/Store 体系结构。
- (3) 每条指令都条件执行。
- (4) 采用多寄存器的 Load/Store 指令。
- (5) 能够在单时钟周期执行的单条指令内完成一项普通的移位操作和一项普通的 ALU 操作。
- (6) 通过协处理器指令集来扩展 ARM 指令集, 包括在编程模式中增加了新的寄存器和数据类型。
- (7) 如果把 Thumb 指令集也当做 ARM 体系结构的一部分, 那么在 Thumb 体系结构中还可以高密度 16 位压缩形式表示指令集。

2.4 ARM 微处理器的应用选型

随着国内嵌入式应用领域的发展, ARM 芯片必然会获得广泛的重视和应用。但是由于 ARM 芯片有多达几十种的芯核结构、70 多个芯片生产厂家及千变万化的内部功能配置组合, 开发人员在选择方案时会有一定的困难。所以对 ARM 芯片做对比研究是十分必要的。



2.4.1 ARM 芯片选择的一般原则

1、功能

考虑处理器本身能够支持的功能，如 USB、网络、串口、液晶显示功能等。

2、性能

从处理器的功耗、速度、稳定可靠性等方面考虑。

3、价格

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

4、熟悉程度及开发资源

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

5、操作系统支持

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

6、升级

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

7、供货稳定

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

2.4.2 选择一款适合 ARM 教学的 CPU

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

1、ARM 芯核

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

2、系统时钟速度



系统时钟决定了 ARM 芯片的处理速度。ARM7 的处理速度为 0.97MIPS/MHz，常见的 ARM7 芯片系统主时钟为 20~133MHz，ARM9 的处理速度为 1.1MIPS/MHz，常见的 ARM9 的系统主时钟为 100~233MHz。Cortex-A 系列的主时钟频率也越来越快，如 Cortex-A9 主频率可以达到 1.6GHz 以上，如果希望学习可以支持较为复杂的操作系统的芯片时，可以选择 ARM9 及 ARM9 以上的芯片。

3、支持内存访问的类型

支持内存访问的类型如表所示。

表 支持内存访问的类型

芯片名	是否有 SDRAM	是否有 DDR2	是否有 mDDR	是否有 DDR3
S3C2410	是	否	否	否
S3C2440	是	否	否	否
S5PV210	否	是	否	否
S5PV310	否	否	否	是
EXYNOS4412	否	否	否	是

4、USB 接口

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

表 内置 USB 控制器的 ARM 芯片

芯片型号	ARM 内核	供应商	USB (otg)	USB Host
S3C2410	ARM920T	SAMSUNG	1	2
S3C2440	ARM920T	SAMSUNG	1	2
EXYNOS4412	CORTEX-A8	SAMSUNG	1	1
S5PV310	CORTEX-A9	SAMSUNG	1	1
EXYNOS4412	CORTEX-A9	SAMSUNG	1	2

5、GPIO 数量

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

6、中断控制器

ARM 内核只提供快速中断（FIQ）和标准中断（IRQ）两个中断向量。但各个半导体厂家在设计芯片时加入了自己定义的中断控制器，以便支持诸如串行口、外部中断、时钟中断等硬件中断。外部中断控制是选择芯片时必须考虑的重要因素，合理的外部中断设计可以很大程度地减少任务调度工作量。例如 PHILIPS 公司的 SAA7750，所有 GPIO 都可以设置成 FIQ 或 IRQ，并且可以选择上升沿、下降沿、高电平和低电平 4 种中断方式。这使得红外线遥控接收、指轮盘和键盘等任务都可以作为背景程序运行。而 Cirrus Logic 公司的 EP7312 芯片只有 4 个外部中断源，并且每个中断源都只能是低电平或高电平中断，这样接收



红外线信号的情况下必须用查询方式，浪费大量 CPU 时间。

7、IIS（Integrate Interface of Sound）接口

IIS 接口即集成音频接口。如果设计音频应用产品，IIS 接口是必需的。

8、nWAIT 信号

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

9、RTC（Real Time Clock）

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

10、LCD 控制器

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

11、PWM 输出

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

12、ADC 和 DAC

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

13、扩展总线

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

14、UART 和 IrDA

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

15、时钟计数器和看门狗



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

16、 电源管理功能

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

17、 DMA 控制器

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

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

18、 封装类型

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

最后，院校的实际情况结合当前及未来一段时间的市场人才需求，经过综合考虑，本书教学选取的是三星公司的 Exynos4412 芯片。Exynos4412 是一款基于 Cortex-A9 核心的微处理器芯片。本章的后面部分章节将对 Cortex-A9 的一些特性及 Exynos4412 进行详细介绍。

2.5 Cortex-A9 内部功能及特点

Cortex-A9 处理器是一款高性能、低功耗的处理器核心，并支持 Cache、虚拟存取，它的特性如下：

- 完全执行 v7-A 体系指令集。
- 可配置 64 位或 128 位 AMBA 高速总线接口 AXI。
- 具有一个集成的整形流水线。
- 具有一个 NEON 技术下执行 SIMD/VFP 的流水线。
- 支持动态分支预取，全局历史缓存，8 入口返回栈。
- 具有独立的数据/指令 MMU。
- 16KB/32KB 可配置 1 级 Cache。
- 具有带奇偶校验及 ECC 校验的 2 级 Cache。
- 支持 ETM 的非侵入式调试。
- 具有静态/动态电源管理功能。
- ARMv7 体系指令集方面表现如下特点：
 - 支持 ARM Thumb-2 高密度指令集。
 - 使用 ThumbEE，执行环境加速。
 - 安全扩展体系加强了安全应用的可靠性。
 - 先进的 SIMD 体系技术用于加速多媒体应用。



- 支持 VFP 第三代向量浮点运算。

2.6 数据类型

2.6.1 ARM 的基本数据类型

ARM 采用的是 32 位架构，ARM 的基本数据类型有以下 3 种。

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

存储器可以看做是序号为 $0 \sim 2^{32}-1$ 的线性字节阵列。如图 2-1

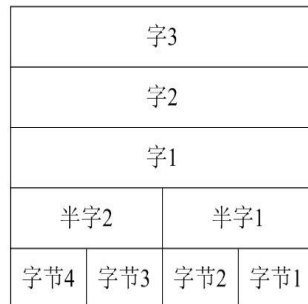


图 2-1 ARM 存储器组织结构

所示为 ARM 存储器的组织结构。其中每一个字节都有唯一的地址。

字节可以占用任一位置，图中给出了几个例子。长度为 1 个字的数据

项占用一组 4 字节的位置，该位置开始于 4 的倍数的字节地址（地址最末两位为 00）。半字占有两个字节的位置，该位置开始于偶数字节地址（地址最末一位为 0）。

注意：

- (1) ARM 系统结构 v4 以上版本支持以上 3 种数据类型，v4 以前版本仅支持字节和字。
- (2) 当将这些数据类型中的任意一种声明成 unsigned 类型时， n 位数据值表示范围为 $0 \sim 2^n-1$ 的非负数，通常使用二进制格式。
- (3) 当将这些数据类型的任何一种声明成 signed 类型时， n 位数据值表示范围为 $-2^{n-1} \sim 2^{n-1}-1$ 的整数，使用二进制的补码格式。
- (4) 所有数据类型指令的操作数都是字类型的，如“ADD r1, r0, #0x1”中的操作数“0x1”就是以字类型数据处理的。
- (5) Load/Store 数据传输指令可以从存储器存取传输数据，这些数据可以是字节、半字、字。加载时自动进行字节或半字的零扩展或符号扩展。对应的指令分别为 LDR/BSTRB（字节操作）、LDRH/STRH（半字操作）、LDR/STR（字操作）。详见后面的指令参考。
- (6) ARM 指令编译后是 4 个字节（与字边界对齐）。Thumb 指令编译后是 2 个字节（与半字边界对齐）。

2.6.2 浮点数据类型

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



2.6.3 存储器大/小端

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

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

如图所示显示了大端模式和小端模式数据存放特点。

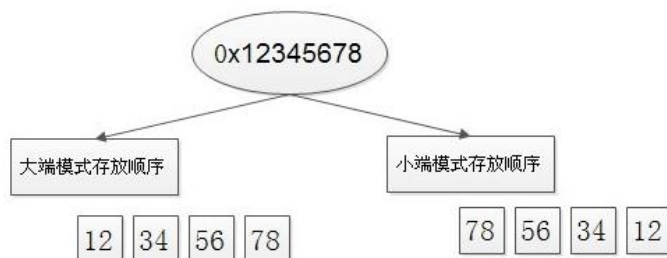


图 大小端模式存放数据的特点

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

程序执行前：

```
r0=0x11223344
```

执行指令：

```

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

执行后：

```

小端模式下：r2=0x44
大端模式下：r2=0x11
  
```

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

2.7 Cortex-A9 内核工作模式

Cortex-A9 基于 ARMv7-A 架构，共有 8 种工作模式，如表所示。

表 Exynos4412 处理器的工作模式

处理器工作模式	简 写	描 述
用户模式（User）	usr	正常程序执行模式，大部分任务执行在这种模式下
快速中断模式（FIQ）	fiq	当一个高优先级（fast）中断产生时将会进入这种模式，一般用于高速数据传输和通道处理
外部中断模式（IRQ）	irq	当一个低优先级（normal）中断产生时将会进入这种模式，一般用于通常的中断处理
特权模式（Supervisor）	svc	当复位或软中断指令执行时进入这种模式，是一种供操作系统使用的保护模式
数据访问中止模式（Abort）	abt	当存取异常时将会进入这种模式，用于虚拟存储或存储保护



未定义指令中止模式 (Undef)	und	当执行未定义指令时进入这种模式，有时用于通过软件仿真协处理器硬件的工作方式
系统模式 (System)	sys	使用和 User 模式相同寄存器集的模式，用于运行特权级操作系统任务
监控模式 (Monitor)	mon	可以在安全模式与非安全模式之间进行转换

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

- (1) 快速中断模式 (FIQ)。
- (2) 外部中断模式 (IRQ)。
- (3) 特权模式 (Supervisor)。
- (4) 数据访问中止模式 (Abort)。
- (5) 未定义指令中止模式 (Undef)。
- (6) 监控模式 (Monitor)。

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

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

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

2.8 Cortex-A9 存储系统

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

- (1) 系统可能提供多种类型的存储器件，如 Flash、ROM、SRAM 等。
- (2) Cache 技术。
- (3) 写缓存技术 (Write Buffers)。
- (4) 虚拟内存和 I/O 地址映射技术。

大多数的系统通过下面的方法之一可实现对复杂存储系统的管理。

- (1) 使用 Cache，缩小处理器和存储系统速度差别，从而提高系统的整体性能。
- (2) 使用内存映射技术实现虚拟空间到物理空间的映射。这种映射机制对嵌入式系统非常重要。

通常嵌入式系统程序存放在 ROM/Flash 中，这样系统断电后程序能够得到保存。但是，通常 ROM/Flash 与 SDRAM 相比，速度慢很多，而且基于 ARM 的嵌入式系统中通常把异常中断向量表放在 RAM 中。利用内存映射机制可以满足这种需要。在系统加电时，将 ROM/Flash 映射为地址 0，这样可以进行一些初始化处理；当这些初始化处理完成后将 SDRAM 映射为地址 0，并把系统程序加载到 SDRAM 中运行，这样



可很好地满足嵌入式系统的需要。

(3) 引入存储保护机制，增强系统的安全性。

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

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

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



图 存储器的层次结构

每级都有特定的存储介质，下面对比各级系统中特定存储介质的存储性能。

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

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

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

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

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

2.8.2 协处理器（CP15）

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



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

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

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

- (1) 寄存器的访问类型（只读/只写/可读可写）。
- (2) 不同的访问引发不同的功能。
- (3) 相同编号的寄存器是否对应不同的物理寄存器。
- (4) 寄存器的具体作用。

2.8.3 存储管理单元（MMU）

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

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

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

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

2.8.4 高速缓冲存储器（Cache）

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

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

通过引入 Cache 和写缓存区，存储系统的性能得到了很大的提高，但同时也带来了一些问题。例如，



由于数据将存在于系统中不同的物理位置，可能造成数据的不一致性；由于写缓存区的优化作用，可能有些写操作的执行顺序不是用户期望的顺序，从而造成操作错误。

2.9 流水线

2.9.1 流水线的概念与原理

处理器按照一系列步骤来执行每一条指令，典型的步骤如下：

- (1) 从存储器读取指令（fetch）。
- (2) 译码以鉴别它属于哪一条指令（decode）。
- (3) 从指令中提取指令的操作数（这些操作数往往存在于寄存器 reg 中）。
- (4) 将操作数进行组合以得到结果或存储器地址（ALU）。
- (5) 如果需要，则访问存储器以存储数据（mem）。
- (6) 将结果写回到寄存器堆（res）。

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

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

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

2.9.2 流水线的分类

1、3 级流水线 ARM 组织

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

- (1) 取指令（fetch）：从寄存器装载一条指令。
- (2) 译码（decode）：识别被执行的指令，并为下一个周期准备数据通路的控制信号。在这一级，指令占有译码逻辑，不占用数据通路。

- (3) 执行（execute）：处理指令并将结果写回寄存器。

如图所示为 3 级流水线指令的执行过程。

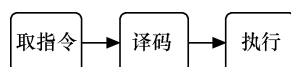


图 3 级流水线

当处理器执行简单的数据处理指令时，流水线使得平均每个时钟周期能完成 1 条指令。但 1 条指令需



要 3 个时钟周期来完成，因此，有 3 个时钟周期的延时（latency），但吞吐率（throughput）是每个周期 1 条指令。

2、5 级流水线 ARM 组织

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

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

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

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

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

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

- （1）取指令（fetch）：从存储器中取出指令，并将其放入指令流水线。
- （2）译码（decode）：指令被译码，从寄存器堆中读取寄存器操作数。在寄存器堆中有 3 个操作数读端口，因此，大多数 ARM 指令能在 1 个周期内读取其操作数。
- （3）执行（execute）：将其中 1 个操作数移位，并在 ALU 中产生结果。如果指令是 Load 或 Store 指令，则在 ALU 中计算存储器的地址。
- （4）缓冲/数据（buffer/data）：如果需要则访问数据存储器，否则 ALU 只是简单地缓冲 1 个时钟周期。
- （5）回写（write-back）：将指令的结果回写到寄存器堆，包括任何从寄存器读出的数据。

如图所示列出了 5 级流水线指令的执行过程。

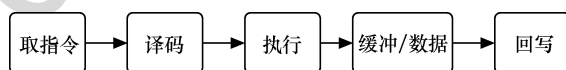


图 5 级流水线

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

3、13 级流水线



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

2.9.3 影响流水线性能的因素

1、互锁

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

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

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

2、跳转指令

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

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

2.10 寄存器组织

ARM 处理器有如下 40 个 32 位长的寄存器。

- (1) 33 个通用寄存器。
- (2) 6 个状态寄存器：1 个 CPSR (Current Program Status Register, 当前程序状态寄存器)，6 个 SPSR (Saved Program Status Register, 备份程序状态寄存器)。
- (3) 1 个 PC (Program Counter, 程序计数器)。

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



图 寄存器列表

当前处理器的模式决定着哪组寄存器可操作，任何模式都可以存取下列寄存器。

- (1) 相应的 R0~R12。
- (2) 相应的 R13 (Stack Pointer, SP, 栈指向) 和 R14 (the Link Register, LR, 链路寄存器)。
- (3) 相应的 R15 (PC)。
- (4) 相应的 CPSR。

特权模式 (除 System 模式外) 还可以存取相应的 SPSR。

通用寄存器根据其分组与否可分为以下两类。

- (1) 未分组寄存器 (Unbanked Register)，包括 R0~R7。
- (2) 分组寄存器 (Banked Register)，包括 R8~R14。

1、未分组寄存器

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

2、分组寄存器

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

对于分组寄存器 R8~R12 来说，每个寄存器对应两个不同的物理寄存器。一组用于除 FIQ 模式外的所有处理器模式，而另一组则专门用于 FIQ 模式。这样的结构设计有利于加快 FIQ 的处理速度。不同模式



下寄存器的使用，要使用寄存器名后缀加以区分。例如，当使用 FIQ 模式下的寄存器时，寄存器 R8 和寄存器 R9 分别记为 R8_fiq、R9_fiq；当使用用户模式下的寄存器时，寄存器 R8 和 R9 分别记为 R8_usr、R9_usr 等。在 ARM 体系结构中，R8~R12 没有任何指定的其他的用途，所以当 FIQ 中断到达时，不用保存这些通用寄存器，也就是说，FIQ 处理程序可以不必执行保存和恢复中断现场的指令，从而可以使中断处理过程非常迅速。所以 FIQ 模式常被用来处理一些时间紧急的任务，如 DMA 处理。

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

(1) R13_<mode>

(2) R14_<mode>

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

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

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

寄存器 R14 又被称为连接寄存器(Link Register, LR)，在 ARM 体系结构中具有下面两种特殊的作用。每一种处理器模式用自己的 R14 存放当前子程序的返回地址。当通过 BL 或 BLX 指令调用子程序时，R14 被设置成该子程序的返回地址。在子程序返回时，把 R14 的值复制到程序计数器(PC)。典型的做法是使用下列两种方法之一。

a) 执行下面任何一条指令。

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

b) 在子程序入口处使用下面的指令将 PC 保存到堆栈中。

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

在子程序返回时，使用如下相应的配套指令返回。

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

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

R14 也可以被用做通用寄存器使用。

2.11 程序状态寄存器

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



- (1) ALU (Arithmetic Logic Unit, 算术逻辑单元) 状态标志的备份。
- (2) 当前的处理器模式。
- (3) 中断使能标志。
- (4) 设置处理器的状态。

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

CPSR 寄存器 (和保存它的 SPSR 寄存器) 中的位分配如图所示。

31	30	29	28	27	26	25	24	23	20	19	16	15	10	9	8	7	6	5	4	0
N	Z	C	V	Q	IT [1:0]	J	保留	GE[3:0]	IT[7:2]	E	A	I	F	T	M[4:0]					

图 程序状态寄存器格式

下面给出各个状态位的定义。

1、标志位

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

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

各条件标志位的具体含义如下。

1) N

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

2) Z

$Z = 1$ 表示运算的结果为零, $Z = 0$ 表示运算的结果不为零。

3) C

下面分 4 种情况讨论 C 的设置方法。

- (1) 在加法指令中 (包括比较指令 CMN), 当结果产生了进位, 则 $C = 1$, 表示无符号数运算发生上溢出; 其他情况下 $C = 0$ 。
- (2) 在减法指令中 (包括比较指令 CMP), 当运算中发生错位 (即无符号数运算发生下溢出), 则 $C = 0$; 其他情况下 $C = 1$ 。
- (3) 对于在操作数中包含移位操作的运算指令 (非加/减法指令), C 被设置成被移位寄存器最后移出去的位。
- (4) 对于其他非加/减法运算指令, C 的值通常不受影响。



4) V

下面分两种情况讨论 V 的设置方法。

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

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

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

2、Q 标志位

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

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

3、控制位

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

1) 中断禁止位

I = 1，IRQ 被禁止。

F = 1，FIQ 被禁止。

2) 状态控制位

T 位是处理器的状态控制位。

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

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

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

3) 模式控制位

M[4:0]作为位模式控制位，这些位的组合确定了处理器处于哪种状态。如表所示列出了其具体含义。只有表中列出的组合是有效的，其他组合无效。

表 状态控制位 M[4:0]

M[4 : 0]	处理器模式	可以访问的寄存器
0b10000	User	PC, R14~R0, CPSR
0b10001	FIQ	PC, R14_fiq~R8_fiq, R7~R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq~R13_irq, R12~R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc~R13_svc, R12~R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt~R13_abt, R12~R0, CPSR, SPSR_abt



0b11011	Undefined	PC, R14_und~R13_und, R12~R0, CPSR, SPSR_und
0b11111	System	PC, R14~R0, CPSR (ARM v4 及更高版本)
0b10110	Secure monitor	PC,R0-R12, CPSR,SPSR_mon,r13_mon,r14_mon

4、IF-THEN 标志位

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

其中 IT[7:5]定义为基本条件,如图所示。

IT[4:0]被定义为 IF-THEN 语句块的长度。

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

图 IF-THEN 标志位[7:5]的定义

5、E 位/A 位/GE[19-16]位的定义

A 表示异步异常禁止位。

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

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

2.12 三星 Exynos4412 处理器介绍

Exynos 4 Quad 四核处理器虽然从去年 9 月我们就知道了会有四核处理器,但一直到前三星才总算是正式发布了四核产品——三星叫它 Exynos 4 Quad,实际上就是我们知道的 Exynos 4412。Galaxy S III 会用这颗处理器,MX 32/64GB 版本也会用,但是很多人肯定会有这样的疑问,那就是四核到底有用吗?双核都已经这么热了,四核岂不是要熔化,电池一天得充 8 次吗?现在有了官方的资料,总算可以比较详细的了解 4412 的参数了。

Cortex A9 时代三星一共发布了两代产品,第一代是 Galaxy S II 和 MX 采用的 Exynos 4210,第二代有两款,一款是双核的 Exynos 4212,一款是四核的 Exynos 4412。第一代产品采用的是 45nm 工艺制造,由于三星的 45nm 工艺在业内是比较落后的,虽然通过种种手段将 Exynos 4210 的频率提升到了 1.4GHz,但这么做的代价也是非常明显的功耗激增(这点在 MX 上我们也看到了)。总体而言,Exynos 4212 和 4412 在架构上和 Exynos 4210 并没有区别,大体上的硬件配置也是一样的,最大的区别就在于 Exynos 4212/4412 采用了三星最新的 32nm HKMG 工艺。

那么这个工艺到底有多少效果?三星放出了一段视频,其中给出了精确的功耗对比。

左边是 GPU 的功耗对比,右边是 CPU 的功耗对比。图中 4210 和 4212 的 GPU 都运行在 266MHz

的频率下,而 CPU 则是 4210 运行在 1.2GHz,4212 运行在 1.5GHz。结果很明显,不论是 CPU 还是 GPU,32nm HKMG 的功耗都比 45nm 低 40%,CPU 甚至是在频率高了 25%的情况下。1.2GHz 的 Exynos 4210,CPU 满载的平均功耗达到了 1.6W,而 4212 则不到 1W,而 MX 采用的 CPU 频率高达 1.4GHz,因此可想而知,满载功很可能接近 2W 大关,足足是 4212 的两倍。

这样的差距在实际上会进一步得到放大。我们知道,四核的 Exynos 4412 并不会跑在 1.5GHz,而是 1.4GHz,因此四核处理器在达到双核两倍性能的同时,功耗却只有双核的八成。换句话说,四核处理器在实现双核同样性能的时候,大约只需要区区 40%的电力,这意味着续航和发热都可能会大大改善。虽然四核的绝对性能对我们而言实际上没有什么太大的意义,但是 32nm HKMG 带来的功耗降低是非常显著的,即便不为了性能,也有足够的理由去选择。

32nm 工艺带来的低功耗,同样也转化到了 GPU 上,我们知道 MX 的 GPU 运行频率大约是 233MHz,因此在 Exynos 4412 上,如果保持同样的 GPU 功耗,频率可以设定到大约 400MHz,从而实现 170%的性能。我们也看到了 Exynos 4412 的一些跑分成绩,的确和 Exynos 4210 相比有着几乎两倍的表现,而这就是源于 GPU 频率设置到超过 400MHz。

一句话总结一下,四核的 Exynos 4412 处理器和现在的双核 Exynos 4210 相比,可以做到同样 CPU 性能下功耗降低 60%,同样功耗的情况下 GPU 性能提升 80%。

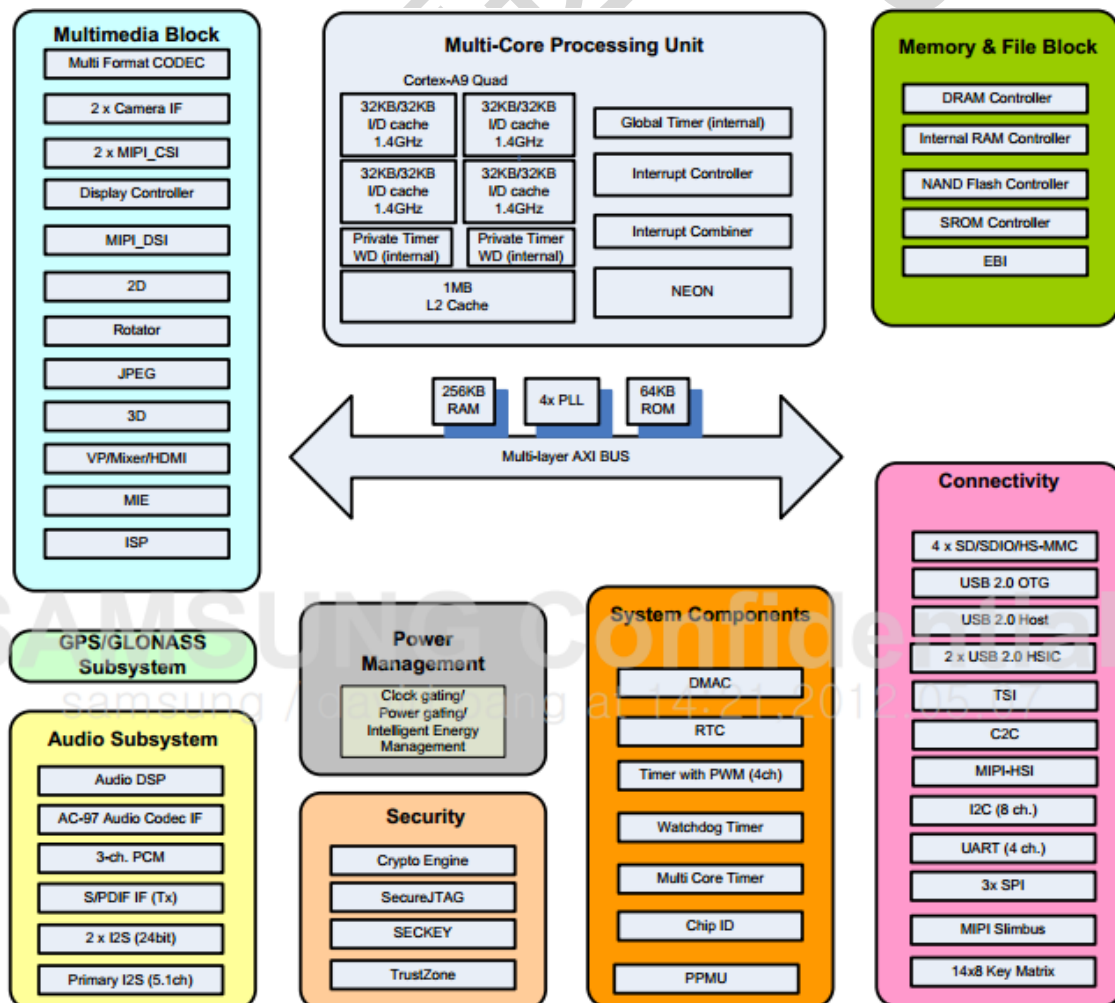




图 Exynos4412 结构框图

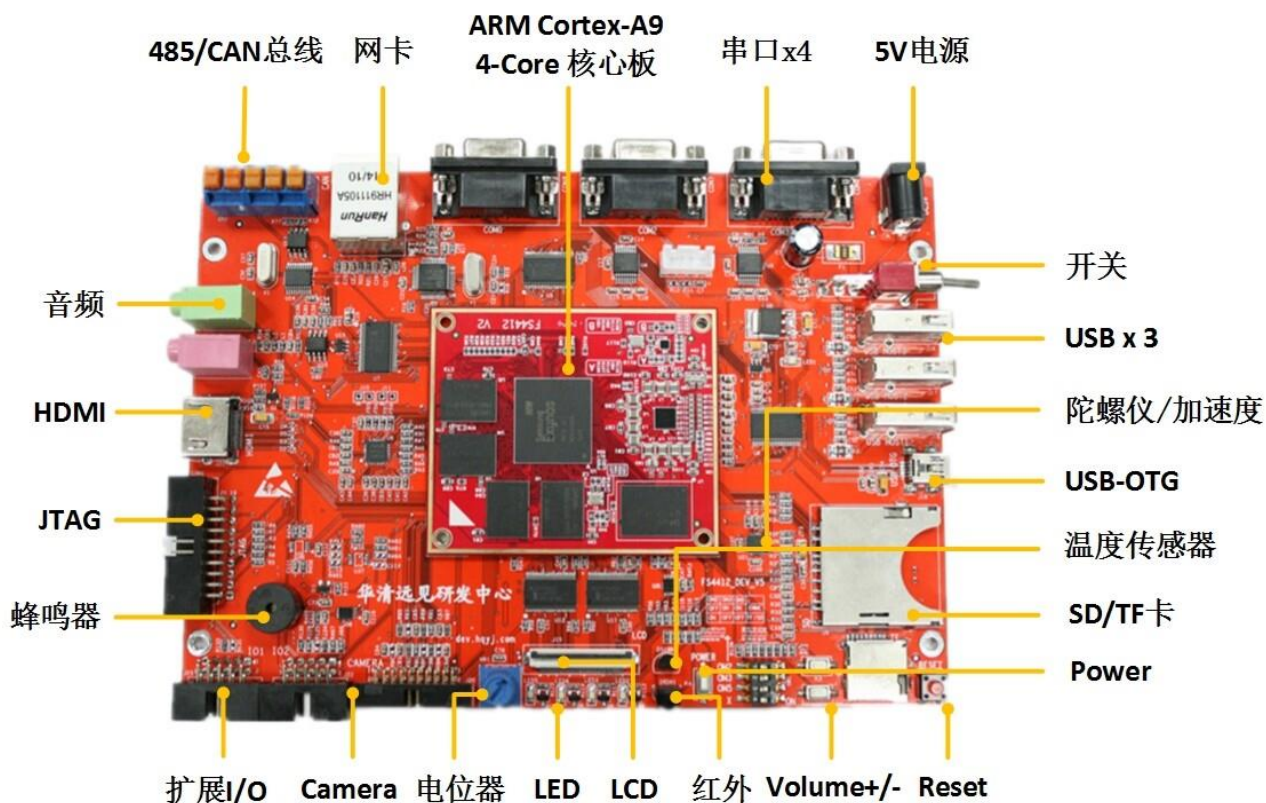
2.13 FS4412 开发平台介绍

FS4412 采用的处理器使用 Samsung 最新的 ARM Cortex-A9 四核 CPU 的 Exynos4412, 主频达到 1.4~1.6GHz。该芯片采用了最新的 32nm 的先进工艺制程, 功耗方面有了明显的降低。

Exynos4412 处理器已经广泛应用于多个领域。在我们熟悉的智能手机中, 如: 三星 Galaxy SIII, 魅族、联想、纽曼等等, 都有基于 Exynos4412 的产品。

随着 ARM 处理器、Linux 操作系统、Android 系统的快速发展, 嵌入式教学对硬件平台的要求越来越高。FS4412 平台是华清远见研发中心根据之前丰富的教学、研发经验, 专为下一代教学开发设计的。平台除了有系统、丰富的软件实验资源外, 硬件设计上也有很多特色。FS4412 核心实验板如图所示。





图：板载硬件资源

一、丰富的硬件接口

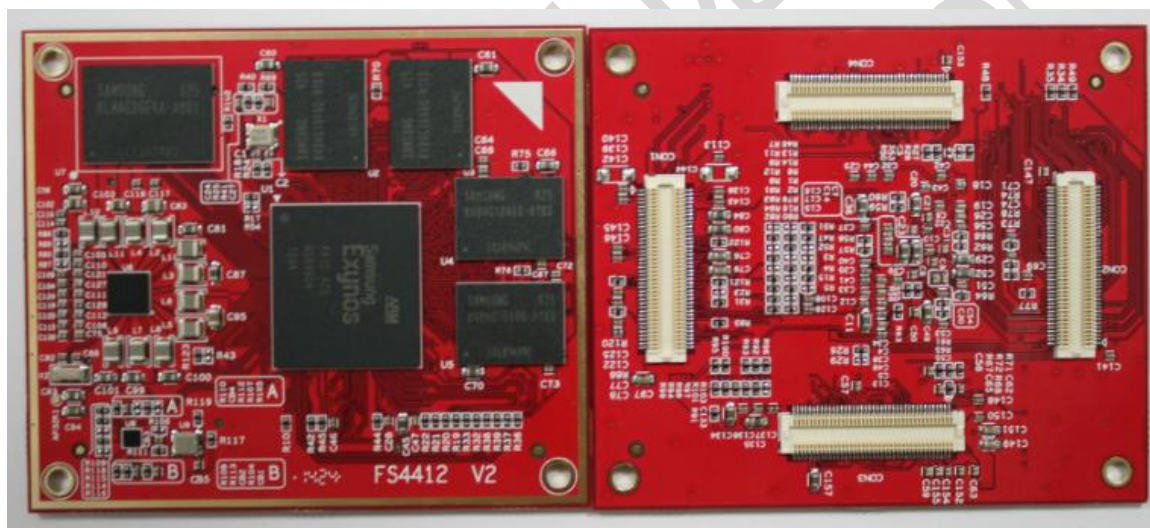
接口技术是嵌入式系统技术中非常核心的环节。FS4412 平台针对嵌入式系统培训中重要的硬件接口，都板载了典型的接口芯片，方便教学。

接口名称	接口芯片	重要程度
A/D	电位计(可调电阻)	★★★★★
PWM	无源蜂鸣器	★★★★★
GPIO	4 个 LED 灯	★★★★★
I2C	加速度/陀螺仪传感器	★★★★★
SPI	SPI 接口的 CAN 线芯片	★★★★★
UART	3 个	★★★★★
单总线	温度传感器/红外接收器	★★★★
I2S	音频接口芯片	★★★★★
USB	3 路 USB HOST、1 路 USB OTG	★★★★★



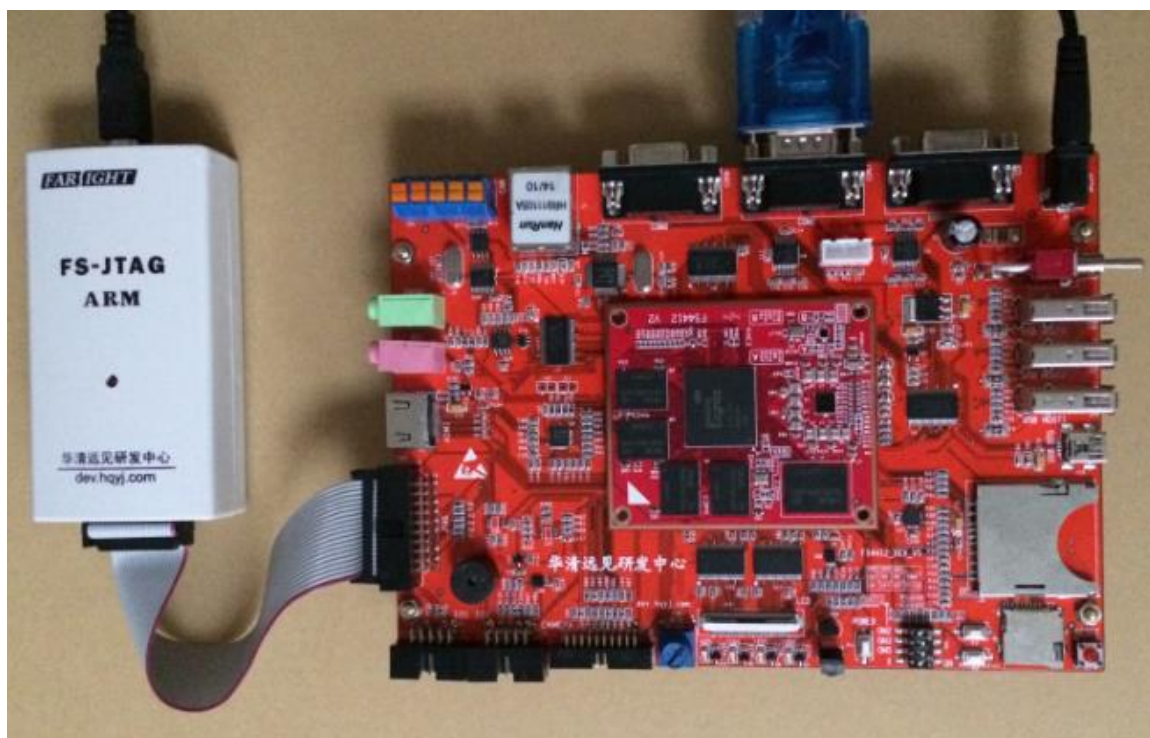
CAN 总线	1 路 CAN 总线扩展	★★★
SDIO	1 路 SD 卡/TF 卡接口	★★★★★
CSI	1 路摄像头接口	★★★★★
LCD RGB/LVDS	一个 RGB/LVDS 接口，配置 1024*600 的液晶屏	★★★★★
异步系统扩展总线	100M 网卡芯片	★★★★★
HDMI	支持 1080P 输出	★★★

二、功能强大的核心板

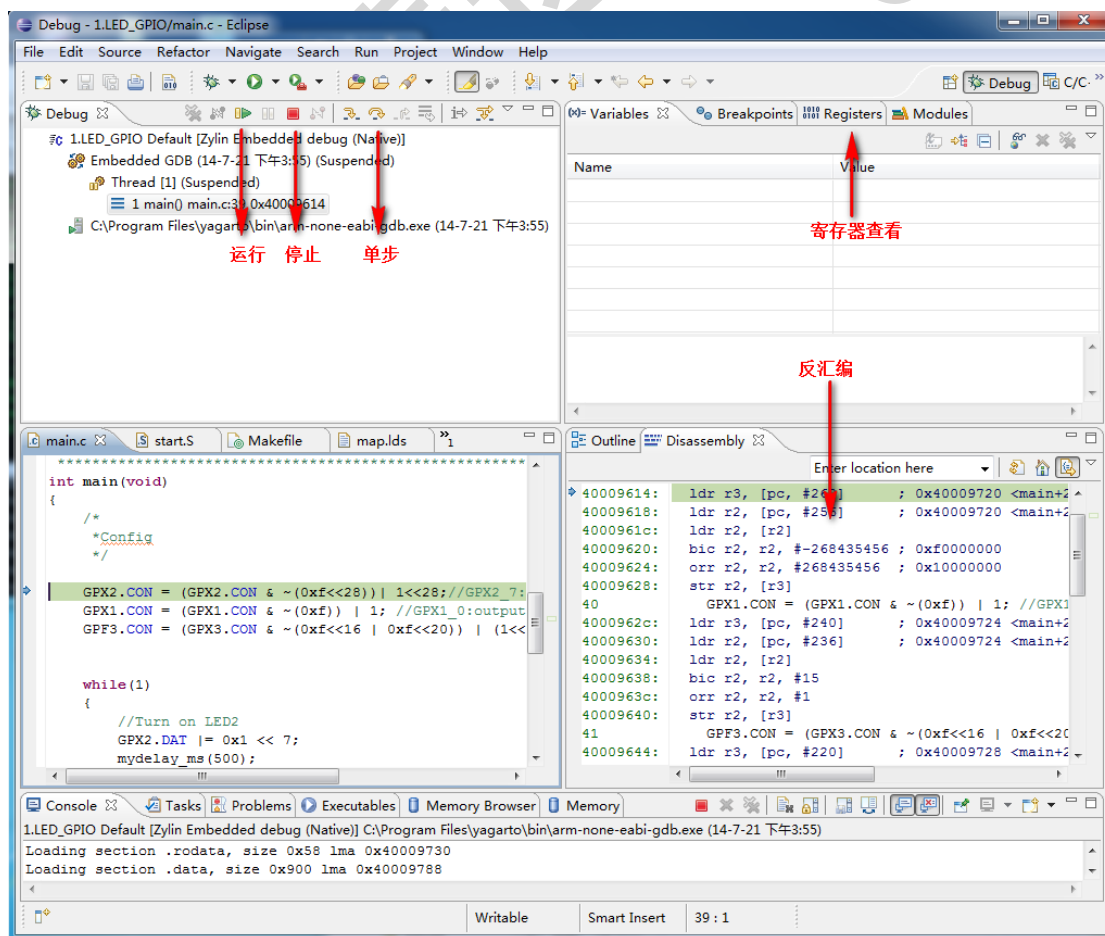


图：核心板资源

三、支持 Cortex-A9 仿真器（选配）



图：FS-JTAG Cortex-A9 仿真器



图：仿真器上位机调试环境



四、支持 Android 红外遥控



图：Android 遥控器

五、设计思路

从软、硬件两个方面，充分考虑教学需求，具体如下：

硬件接口方面设计	板载了典型的按键、I2C、SPI、单总线、A/D、PWM、等重要的基本接口器件。 板载了 USB、SD 卡、HDMI、LCD、Camera 等接口。 支持 CAN 总线、485 总线等常用现场总线、Android 红外遥控
内存设计	采用 1GB、2GB 两种内存，用户可选。对于教学用途来说没有差别 采用 4GB、16GB 两种 eMMC 闪存，用户可选。对于教学用途来说没有差别
PMU 设计	采用流行的 PMU 电源管理芯片，而非简单的分立电源
仿真器支持	自主研发了 FS-JTAG 仿真器，能够仿真 FS4412，实现单步、断点、内存查看等功能。并编写了系统的 ARM 裸机测试程序。
系统软件设计	提供完善的 ARM 处理器、Linux 系统移植、Linux 驱动、Linux 应用层、Android 底层、Android 应用层实验代码和实验文档。
软件项目方面	多个 Linux、Android 综合项目。提供源码及项目设计文档。

六、FS4412 拥有如下丰富的硬件资源。

	功能部件	型号参数
--	------	------



核心配置	CPU	<ul style="list-style-type: none"> - Samsung Exynos 4 Quad（四核处理器） - 32nm HKMG - 1433 MHz（最多可达 1.6GHz）
	GPU	<ul style="list-style-type: none"> - Mali-400MP（主频可达 400MHz）
	屏幕	<ul style="list-style-type: none"> - LVDS 40 Pin 显示接口 - 7 寸 1024 x 600 高分辨率显示屏 - 多点电容触摸屏
	RAM 容量	<ul style="list-style-type: none"> - 1GB DDR3（可选配至 2GB）
	ROM 容量	<ul style="list-style-type: none"> - 4GB eMMC（可选配至 16GB）
	多启动方式	<ul style="list-style-type: none"> - eMMC 启动、MicroSD(TF)/SD 卡启动 - 通过控制拨码开关切换启动方式 - 可以实现双系统启动
板载接口	存储卡接口	<ul style="list-style-type: none"> - 1 个 MicroSD(TF)卡接口 - 1 个 SD 卡接口 - 最高可扩展至 64GB
	摄像头接口	<ul style="list-style-type: none"> - 20 Pin 接口，支持 OV3640 300 万像素摄像头
	HDMI 接口	<ul style="list-style-type: none"> - HDMI A 型接口 - HDMI v1.4a - 最高 1080p@30fps 高清数字输出
	JTAG 接口	<ul style="list-style-type: none"> - 20 Pin 标准 JTAG 接口 - 支持 FS-JTAG Cortex-A9 ARM 仿真器 - 独家支持详尽的 ARM 裸机程序
	USB 接口	<ul style="list-style-type: none"> - 1 路 USB OTG - 3 路 USB HOST 2.0（可扩展 USB-HUB）
	音频接口	<ul style="list-style-type: none"> - 1 路 Mic 接口 - 1 路 Speaker 耳机输出 - 1 路 Speaker 立体声功放输出（外置扬声器）
	网卡接口	<ul style="list-style-type: none"> - DM9000 百兆网卡
	RS485 接口	<ul style="list-style-type: none"> - 1 路 RS485 总线接口
	CAN 总线接口	<ul style="list-style-type: none"> - 1 路 CAN 总线接口
	串口	<ul style="list-style-type: none"> - 1 路 5 线 RS232 串口 - 2 路 3 线 RS232 串口 - 1 路 TTL 串口



	扩展 I/O 接口	- 1 路 I2C (已将 1.8V 转换为 3.3V) - 1 路 SPI (已将 1.8V 转换为 3.3V) - 3 路 ADC (1 路含 10K 电阻) - 多路 GPIO、外部中断 (已将 1.8V 转换为 3.3V)
板级资源	按键	- 1 个 Reset 按键 - 1 个 Power 按键 - 2 个 Volume (+/-) 按键
	LED	- 1 个电源 LED - 4 个可编程 LED
	蜂鸣器	- 1 个无源 PWM 蜂鸣器
	红外接收器	- 1 个 IRM3638 红外接收器 - 可选配红外遥控器在 Android 下使用
	温度传感器	- 1 个 DS18B20 温度传感器
	ADC	- 1 路电位器输入 (Android 下可模拟电池电量)
	RTC	- 1 个内部 RTC 实时时钟
操作系统支持		- Linux3.0、Linux3.14(Device Tree)、Android4.0、QT

2.14 本章小结

本章介绍了 ARM 处理器的一些关键技术，如 ARM 核的工作模式、存储系统、流水线、寄存器组织等。并且列举了一款基于 Cortex-A9 核的处理器芯片 EXYNOS4412。通过本章的学习，学员可以对 ARM 核的一些关键技术有所认识。

2.15 练习题

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



第 3 章 ARM 微处理器指令系统

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

本章主要介绍 ARM 汇编语言。主要内容如下：

- ARM 处理器的寻址方式。
- ARM 处理器的指令集。

3.1 ARM 处理器的寻址方式

ARM 指令的寻址方式分为数据处理指令寻址方式和内存访问指令寻址方式。

3.1.1 数据处理指令寻址方式

数据处理指令的基本语法格式如下：

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

其中，<shifter_operand>有 11 种形式，如表所示。

表 <shifter_operand>的寻址方式

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

数据处理指令寻址方式可以分为以下几种。

- (1) 立即数寻址方式。
- (2) 寄存器寻址方式。
- (3) 寄存器移位寻址方式。

1、立即数寻址方式

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



图 立即数表示方法

下面列举了一些有效的立即数：

0xFF、0x104、0xFF0、0xFF00、0xFF000、0xFF00000、0xF000000F

下面是一些无效的立即数：

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

下面是一些应用立即数的指令：

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

2、寄存器寻址方式

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

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

3、寄存器移位寻址方式

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

下面是一些在指令中使用了移位操作的例子：

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

3.1.2 内存访问指令寻址方式

内存访问指令的寻址方式可以分为以下几种。

- (1) 字及无符号字节的 Load/Store 指令的寻址方式。
- (2) 杂类 Load/Store 指令的寻址方式。
- (3) 批量 Load/Store 指令的寻址方式。
- (4) 协处理器 Load/Store 指令的寻址方式。

1、字及无符号字节的 Load/Store 指令的寻址方式



字及无符号字节的 Load/Store 指令语法格式如下：

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

其中，<addressing_mode>共有 9 种寻址方式，如表所示。

表 字及无符号字节的 Load/Store 指令的寻址方式

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

上表中，“!”表示完成数据传输后要更新基址寄存器。

2、杂类 Load/Store 指令的寻址方式

使用该类寻址方式的指令的语法格式如下：

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

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

该类寻址方式分为 6 种类型，如表所示。

表 杂类 Load/Store 指令的寻址方式

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



5	[Rn], #±<offset_8>	立即数后索引寻址 (Immediate post-indexed)
6	[Rn], ±<Rm>	寄存器后索引寻址 (Register post-indexed)

3、批量 Load/Store 指令寻址方式

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

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

该类指令的语法格式如下：

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

该类指令的寻址方式如表所示。

表 批量 Load/Store 指令的寻址方式

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

4、堆栈操作寻址方式

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

下面详细讨论如何使用合适的寻址方式实现数据的堆栈操作。

根据不同的寻址方式，将堆栈分为以下 4 种。

- (1) Full 栈：堆栈指针指向栈顶元素 (last used location)。
- (2) Empty 栈：堆栈指针指向第一个可用元素 (the first unused location)。
- (3) 递减栈：堆栈向内存地址减小的方向生长。
- (4) 递增栈：堆栈向内存地址增加的方向生长。

根据堆栈的不同种类，将其寻址方式分为以下 4 种。

- (1) 满递减 FD (Full Descending)。
- (2) 空递减 ED (Empty Descending)。
- (3) 满递增 FA (Full Ascending)。
- (4) 空递增 EA (Empty Ascending)。

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

表 堆栈寻址方式和批量 Load/Store 指令寻址方式的对应关系

批量数据寻址方式	堆栈寻址方式	L 位	P 位	U 位
----------	--------	-----	-----	-----



LDMDA	LDMFA	1	0	0
LDMIA	LDMFD	1	0	1
LDMDB	LDMEA	1	1	0
LDMIB	LDMED	1	1	1
STMDA	STMED	0	0	0
STMIA	STMEA	0	0	1
STMDB	STMFD	0	1	0
STMIB	STMFA	0	1	1

5、协处理器 Load/Store 寻址方式

协处理器 Load/Store 指令的语法格式如下：

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

3.2 ARM 处理器的指令集

2.2.1 数据操作指令

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

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

表 数据处理指令列表

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

1、MOV 指令

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



寄存器，也可以是立即数。

MOV 指令多用于设置初始值或者在寄存器间传送数据。

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

(1) 指令的语法格式：

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

(2) 指令举例：

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

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

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

(3) 指令的使用如下。

MOV 指令主要完成以下功能。

- 将数据从一个寄存器传送到另一个寄存器。
- 将一个常数值传送到寄存器中。
- 实现无算术和逻辑运算的单纯移位操作，操作数乘以 2^n 可以用左移 n 位来实现。
- 当 PC (R15) 用做目的寄存器时，可以实现程序跳转。如 “MOV PC, LR”，所以这种跳转可以实现子程序调用及从子程序返回，代替指令 “B, BL”。
- 当 PC 作为目标寄存器且指令中 S 位被设置时，指令在执行跳转操作的同时，将当前处理器模式的 SPSR 寄存器的内容复制到 CPSR 中。这种指令 “MOVS PC LR” 可以实现从某些异常中断中返回。

2、MVN 指令

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

MVN 指令多用于向寄存器传送一个负数或生成位掩码。

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

(1) 指令的语法格式：

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

3.2.2 指令举例如下。

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

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

(1) 指令的使用如下。



MVN 指令主要完成以下功能：

- (a) 向寄存器中传送一个负数。
- (b) 生成位掩码 (Bit Mask)。
- (c) 求一个数的反码。

3、AND 指令

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

(1) 指令的语法格式：

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

(2) 指令举例如下。

(a) 保留 R0 中的 0 位和 1 位，丢弃其余的位。

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

(b) $R2 = R1 \& R3$ 。

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

(c) $R0 = R0 \& 0x01$ ，取出最低位数据。

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

4、EOR 指令

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

(1) 指令的语法格式：

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

(2) 指令举例：

(a) 反转 R0 中的位 0 和 1。

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

(b) 将 R1 的低 4 位取反。

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

(c) $R2 = R1 \wedge R0$ 。

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

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

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

5、SUB 指令

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

(1) 指令的语法格式：

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




(2) SUB 指令举例:

(a) $R0 = R1 - R2$ 。

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

a) $R0 = R1 - 256$ 。

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

b) $R0 = R2 - (R3 \ll 1)$ 。

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

6、RSB 指令

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

(1) 指令的语法格式:

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

(2) RSB 指令举例:

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

```
RSBS   R2, R0, #0
```

```
RSC    R3, R1, #0
```

7、ADD 指令

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

(1) 指令的语法格式:

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

(2) ADD 指令举例:

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

```
ADD    R0, R1, #256        ; R0 = R1 + 256
```

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

8、ADC 指令

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

(1) 指令的语法格式:

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

(2) ADC 指令举例:

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

128 位结果: 寄存器 R0、R1、R2 和 R3。



第一个 128 位数：寄存器 R4、R5、R6 和 R7。

第二个 128 位数：寄存器 R8、R9、R10 和 R11。

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

9、SBC 指令

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

(1) 指令的语法格式:

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

(2) SBC 指令举例:

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

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

10、RSC 指令

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

(1) 指令的语法格式:

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

(2) RSC 指令举例:

下面的程序使用 RSC 指令实现求 64 位数值的负数。

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

11、TST 测试指令

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

(1) 指令的语法格式:

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

(2) TST 指令举例:

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



下面的指令测试在 R0 中是否设置了位 0。

```
TST    R0, #1
```

12、 TEQ 指令

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

(1) 指令的语法格式:

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

(2) TEQ 指令举例:

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

```
TEQ    R0, R1
```

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

13、 CMP 指令

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

(1) 指令的语法格式:

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

(2) CMP 指令举例:

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

显然, CMP 不需要显式地指定 S 后缀来更改状态标志。

a) 下面的指令是比较 R1 和立即数 10 并设置相关的标志位。

```
CMP    R1, #10
```

b) 下面的指令是比较寄存器 R1 和 R2 中的值并设置相关的标志位。

```
CMP    R1, R2
```

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

14、 CMN 指令

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

(1) 指令的语法格式:

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



(2) CMN 指令举例:

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

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

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

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

```
CMN    R0, #1
```

15、 ORR 指令

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

(1) 指令的语法格式:

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

(2) ORR 指令举例:

(a) 设置 R0 中位 0 和 1。

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

(b) 将 R0 的低 4 位置 1。

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

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

```
MOV    R1, R2, LSR #4
```

```
ORR    R3, R1, R3, LSL #8
```

16、 BIC 位清零指令

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

(1) 指令的语法格式:

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

(2) BIC 指令举例:

(a) 清除 R0 中的位 0、1 和 3, 保持其余的不变。

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

(b) 将 R3 的反码和 R2 做“逻辑与”操作, 结果保存到 R1 中。

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

3.2.3 乘法指令

ARM 乘法指令完成两个数据的乘法。两个 32 位二进制数相乘的结果是 64 位的积。在有些 ARM 的处



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

如表 3-7 所示为各种形式乘法指令的功能。

表 3-7 各种形式乘法指令

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

其中：

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

(2) 简单的赋值由“:=”表示。

(3) 累加（将右边加到左边）是由“+=”表示。

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

- 对于产生 32 位结果的指令形式，将标志位 N 设置为 Rd 的第 31 位的值；对于产生长结果的指令形式，将其设置为 RdHi 的第 31 位的值。
- 对于产生 32 位结果的指令形式，如果 Rd 等于零，则标志位 Z 置位；对于产生长结果的指令形式，RdHi 和 RdLo 同时为零时，标志位 Z 置位。
- 将标志位 C 设置成无意义的值。
- 标志位 V 不变。

4、MUL 指令

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

(1) 指令的语法格式：

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

(2) 指令举例：

(a) $R1 = R2 \times R3$ 。

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

(b) $R0 = R3 \times R7$ ，同时设置 CPSR 中的 N 位和 Z 位。

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

5、MLA 指令



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

(1) 指令的语法格式:

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

(2) 指令举例:

下面的指令完成 $R1 = R2 \times R3 + 10$ 的操作。

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

6、UMULL 指令

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

(1) 指令的语法格式:

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

(2) 指令举例:

下面的指令完成 $(R1, R0) = R5 \times R8$ 操作。

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

7、UMLAL 指令

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

(1) 指令的语法格式:

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

(2) 指令举例:

下面的指令完成 $(R1, R0) = R5 \times R8 + (R1, R0)$ 操作。

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

8、SMULL 指令

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

(1) 指令的语法格式:

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

(2) 指令举例:

下面的指令完成 $(R3, R2) = R7 \times R6$ 操作。

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

9、SMLAL 指令



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

(1) 指令的语法格式:

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

(2) 指令举例:

下面的指令完成 $(R3, R2) = R7 \times R6 + (R3, R2)$ 操作。

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

3.2.4 Load/Store 指令

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

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

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

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

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

1、单寄存器的 Load/Store 指令

如表所示列出了所有单寄存器的 Load/Store 指令。

表 单寄存器 Load/Store 指令

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



LDRSB	把一个有符号字节装入一个寄存器	$Rd \leftarrow \text{sign}\{\text{mem8}[\text{address}]\}$
LDRSH	把一个有符号半字装入一个寄存器	$Rd \leftarrow \text{sign}\{\text{mem16}[\text{address}]\}$

1> LDR 指令

LDR 指令用于从内存中将一个 32 位的字读取到目标寄存器。

(1) 指令的语法格式:

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

(2) 指令举例:

```
LDR R1,[R0,#0x12] ;将 R0+12 地址处的数据读出,保存到 R1 中(R0 的值不变)
LDR R1,[R0]        ;将 R0 地址处的数据读出,保存到 R1 中(零偏移)
LDR R1,[R0,R2]      ;将 R0+R2 地址的数据读出,保存到 R1 中(R0 的值不变)
LDR R1,[R0,R2,LSL #2] ;将 R0+R2×4 地址处的数据读出,保存到 R1 中(R0、R2 的值不变)
LDR Rd,label        ;label 为程序标号,label 必须是当前指令的-4~4KB 范围内
LDR Rd,[Rn],#0x04   ;Rn 的值用做传输数据的存储地址。在数据传送后,将偏移量 0x04 与 Rn
                    ;相加,结果写回到 Rn 中。Rn 不允许是 R15
```

2> STR 指令

STR 指令用于将一个 32 位的字数据写入到指令中指定的内存单元。

(1) 指令的语法格式:

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

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

a) 变量访问。

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

b) GPIO 设置。

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

c) 程序散转。

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



...

3> LDRB 指令

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

指令的语法格式：

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

4> STRB 指令

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

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

5> LDRH 指令

LDRH 指令用于从内存中将一个 16 位的半字读取到目标寄存器。

如果指令的内存地址不是半字节对齐的，指令的执行结果不可预知。

指令的语法格式：

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

6> STRH 指令

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

指令的语法格式：

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

2、多寄存器的 Load/Store 内存访问指令

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

表 多寄存器的 Load/Store 内存访问指令

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

1> LDM 指令

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

指令的语法格式：

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

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

指令的语法格式：



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

2> STM 指令

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

指令的语法格式：

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

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

指令的语法格式：

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

3> 数据传送指令应用

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

```
LDM{cond}<模式> Rn{!},regist{^}  
STM{cond}<模式> Rn{!},regist{^}
```

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

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

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

```
LDMIA R0!,{R3~R9} ;加载 R0 指向的地址上的多字数据，保存到 R3~R9 中，R0 值更新  
STMIA R1!,{R3~R9} ;将 R3~R9 的数据存储到 R1 指向的地址上，R1 值更新  
STMFD SP!,{R0~R7,LR} ;现场保存，将 R0~R7、LR 入栈
```



LDMFD SP!, {R0~R7, PC}^ ; 恢复现场, 异常处理返回

在进行数据复制时, 先设置好源数据指针, 然后使用块复制寻址指令 LDMIA/STMIA、LDMIB/STMIB、LDMDB/STMDB 进行读取和存储。而进行堆栈操作时, 则要先设置堆栈指针, 一般使用 SP 然后使用堆栈寻址指令 STMFD/LDMFD、STMED/LDMED、STMEA/LDMEA 实现堆栈操作。数据是在基址寄存器的地址之上还是之下, 地址是存储第一个值之前还是之后、增加还是减少, 如表 3-10 所示。

表 3-10 多寄存器的 Load/Store 内存访问指令映射

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

【举例】 使用 LDM/STM 进行数据复制。

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

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

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

3、单数据交换指令

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

表 交换指令 SWP

指 令	作 用	操 作
-----	-----	-----



SWP	字交换	tmp=mem32[Rn] mem32[Rn]=Rm Rd=tmp
SWPB	字节交换	tmp=mem8[Rn] mem8[Rn]=Rm Rd=tmp

4> SWP 字交换指令

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

当<Rd>和<Rm>为同一个寄存器时，指令交换该寄存器和内存单元的内容。

指令的语法格式：

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

5> SWPB 字节交换指令

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

指令的语法格式：

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

6> 交换指令 SWP 应用

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

指令的语法格式：

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

其中，B 为可选后缀，若有 B，则交换字节；否则交换 32 位字。Rd 为目的寄存器，存储从存储器中加载的数据，同时，Rm 中的数据将会被存储到存储器中。若 Rm 与 Rn 相同，则为寄存器与存储器内容进行交换。Rn 为要进行数据交换的存储器地址，Rn 不能与 Rd 和 Rm 相同。

SWP 指令举例：

```
SWP R1, R1, [R0]           ; 将 R1 的内容与 R0 指向的存储单元内容进行交换
SWPB R1, R2, [R0]          ; 将 R0 指向的存储单元内容读取一字节数据到 R1 中（高 24 位清零），并
将 R2 的内容                写入到该内存单元中（最低字节有效），使用 SWP 指令可以方便
地进行信号量操作
12C_SEM .equ 0x40003000
...
```



```
12C_SEM_WAIT:
    MOV    R0, #0
    LDR    R0, =12C_SEM
    SWP    R1, R1, [R0]      ;取出信号量, 并将其设为 0
    CMP    R1, #0           ;判断是否有信号
    BEQ    12C_SEM_WAIT     ;若没有信号则等待
```

3.2.5 跳转指令

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

表 ARMv5 架构跳转指令

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

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

1、跳转指令 B 及带连接的跳转指令 BL

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

（1） 指令的语法格式：

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

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

- BX R14（如果体系结构支持 BX 指令）。
- MOV PC, R14。
- 当子程序在入口处使用了压栈指令：



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

可以使用指令:

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

将子程序返回地址放入 PC 中。

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

(a) 将 PC 寄存器的值作为本跳转指令的基地址值。

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

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

(2) 程序举例:

a) 程序跳转到 LABEL 标号处。

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

b) 跳转到绝对地址 0x1234 处。

```
B 0x1234
```

c) 跳转到子程序 func 处执行,同时将当前 PC 值保存到 LR 中。

```
BL func
```

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

```
BCC LABEL
```

e) 通过跳转指令建立一个无限循环。

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

f) 通过使用跳转使程序体循环 10 次。

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

g) 条件子程序调用示例。

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



2、带状态切换的跳转指令 BX

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

（1） 指令的语法格式：

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

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

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

```
MOV PC, PC
```

或

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

（2） 指令举例：

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

```
BX R0;
```

b) 跳转到 R0 指定的地址，并根据 R0 的最低位来切换处理器状态。

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

```
BX R0;
```

3、带连接和状态切换的连接跳转指令 BLX

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

（1） 语法格式：

```
BLX <target_add>
```

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

- （a） 将指令中指定的 24 位偏移量进行符号扩展，形成 32 位立即数。
- （b） 将结果左移两位。
- （c） 位 H（bit[24]）加到结果地址的第一位（bit[1]）。
- （d） 将结果累加进程序计数器（PC）中。

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



空间的跳转。

(2) 指令的使用

a) 从 Thumb 状态返回到 ARM 状态，使用 BX 指令。

```
BX R14
```

b) 可以在子程序的入口和出口增加栈操作指令。

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

3.2.6 状态操作指令

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

表 程序状态寄存器指令

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

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

1、MRS

MRS 指令用于将程序状态寄存器的内容传送到通用寄存器中。

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

(1) 指令的语法格式：

```
MRS{cond} Rd, PSR
```

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

(2) 指令举例：

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

MRS 指令读取 CPSR，可用来判断 ALU 的状态标志及 IRQ/FIQ 中断是否允许等；在异常处理程序中，读 SPSR 可指定进入异常前的处理器状态等。MRS 与 MSR 配合使用，实现 CPSR 或 SPSR 寄存器的读—修改—写操作，可用来进行处理器模式切换，允许/禁止 IRQ/FIQ 中断等设置。另外，进程切换或允许异常中断嵌套时，也需要使用 MRS 指令读取 SPSR 状态值并保存起来。

2、MSR

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

(1) 指令的语法格式：



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

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

(2) 指令举例：

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

注意：

只有在特权模式下才能修改状态寄存器。

程序中不能通过 MSR 指令直接修改 CPSR 中的 T 位控制位来实现 ARM 状态/Thumb 状态的切换，必须使用 BX 指令来完成处理器状态的切换（因为 BX 指令属转移指令，它会打断流水线状态，实现处理器状态的切换）。MRS 与 MSR 配合使用，实现 CPSR 或 SPSR 寄存器的读—修改—写操作，可用来进行处理器模式切换及允许/禁止 IRQ/FIQ 中断等设置。

3、程序状态寄存器指令的应用

【举例】 使能 IRQ 中断。

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

【举例】 禁止 IRQ 中断。

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

【举例】 堆栈指令初始化。

```
INITSTACK:
    MOV    R0,LR      ;保存返回地址
```

设置管理模式堆栈：

```
MSR    CPSR_c,#0xD3
LDR    SP,StackSvc
```

设置中断模式堆栈：

```
MSR    CPSR_c,#0xD2
LDR    SP,StackSvc
```




3.2.7 协处理器指令

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

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

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

ARM 协处理器指令可分为以下 3 类。

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

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

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

如表所示列出了所有协处理器处理指令。

表 协处理器指令

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



MRC	从协处理器寄存器传数据到 ARM 寄存器
STC	存储协处理器寄存器

下面简单介绍一下比较常用的 MCR 及 MRC 命令的用法：

1、ARM 寄存器到协处理器寄存器的数据传送指令 MCR

1> 指令编码格式

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

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

图 3-2 MCR 指令编码格式

2> 指令的语法格式

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

① <cond>

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

② <coproc>

指定协处理器的编号，标准的协处理器的名字为 p0、p1、…、p15。

③ <opcode_1>

指定协处理器执行的操作码，确定哪一个协处理器指令将被执行。

④ <Rd>

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

⑤ <CRn>

确定包含第一个操作数的协处理器寄存器。

⑥ <CRm>

确定包含第二个操作数的协处理器寄存器。

⑦ <opcode_2>

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

3> 指令举例

将 ARM 寄存器 r7 中的值传送到协处理器 p14 的寄存器 c7 中，第一操作数 opcode_1=1，第二操作数 opcode_2=6。

```
MCR p14, 1, r7, c7, c12, 6
```



4> 指令的使用

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

2、协处理器寄存器到 ARM 寄存器的数据传送指令 MRC

1> 指令编码格式

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

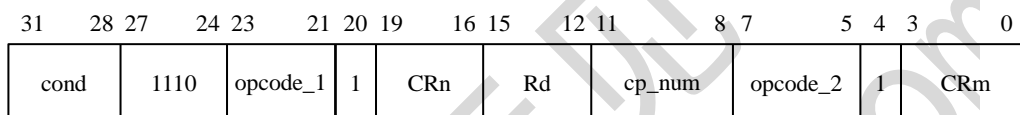


图 MRC 指令编码格式

2> 指令的语法格式

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

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

① <coproc>

指定协处理器的编号，标准的协处理器的名字为 p0、p1、…、p15。

② <opcode_1>

指定协处理器执行的操作码，确定哪一个协处理器指令将被执行。

③ <Rd>

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

④ <CRn>

确定包含第一个操作数的协处理器寄存器。

⑤ <CRm>

确定包含第二个操作数的协处理器寄存器。

⑥ <opcode_2>

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

3> 指令举例

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

```
MRC p15, 5, r4, c0, c2, 3
```



4> 指令的使用

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

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

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

如图所示列出了 cp15 的各个寄存器的目的。

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

图 cp15 寄存器列表

3.2.8 异常产生指令

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

表 ARM 异常产生指令

助记符	含义	操作
SWI	软中断指令	产生软中断，处理器进入管理模式
BKPT	断点中断指令	处理器产生软件断点

软件中断指令（Software Interrupt, SWI）用于产生软中断，从而实现从用户模式变换到管理模式，CPSR



保存到管理模式的 SPSR 中，执行转移到 SWI 向量，在其他模式下也可以使用 SWI 指令，处理器同样切换到管理模式。

(1) 指令的语法格式。

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

(2) 指令举例。

① 下面指令产生软中断，中断立即数为 0。

```
SWI 0;
```

② 产生软中断，中断立即数为 0x123456。

```
SWI 0x123456;
```

③ 使用 SWI 指令时，通常使用以下两种方法进行参数传递。

a. 指令 24 位的立即数指定了用户请求的类型，中断服务程序的参数通过寄存器传递。

下面的程序产生一个中断号为 12 的软中断。

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

b. 另一种情况，指令中的 24 位立即数被忽略，用户请求的服务类型由寄存器 R0 的值决定，参数通过其他寄存器传递。

下面的例子通过 R0 传递中断号，R1 传递中断的子功能号。

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

3.2.9 其他指令介绍

1、特殊指令介绍

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

Fmxr: 由 arm 寄存器将数据转移到协处理器中。

Fmrx: 由协处理器转移到 arm 寄存器中。

如图所示为浮点异常寄存器格式。

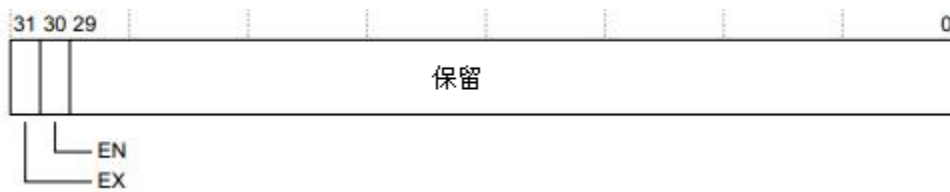


图 浮点异常寄存器格式

如表所示为 FPEXC 的位定义。

表 FPEXC 的位定义



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

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

如果要打开 VFP 协处理器的话，可以用以下指令：

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

2、CLZ 计算前导零数目

(1) 语法格式：

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

其中：

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

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

- (2) 条件标记：该指令不会更改标记。
- (3) 体系结构：ARMv5 以上。
- (4) 示例如图所示。

R0= 0000 0010 1110 1101...0

CLZ R1, R0

R1= 0x6

图 CLZ 例子

3、饱和指令介绍

这是用来设计饱和算法的一组指令，所谓饱和是指出现下列 3 种情况：

- (1) 对于有符号饱和运算，如果结果小于 -2^n ，则返回结果将为 -2^n 。
- (2) 对于无符号饱和运算，如果整数结果是负值，那么返回的结果将为 0。
- (3) 对于结果大于 2^n-1 的情况，则返回结果将为 2^n-1 。

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



QSUB: 带符号减法。

QDADD: 带符号加倍加法。

QDSUB: 带符号加倍减法。

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

(5) 语法格式:

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

其中:

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

(6) 用法如下:

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

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

(8) 体系结构: 该指令可用于 v5T-E 及 v6 或者更高版本的体系中。

(9) 示例如下:

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

3.3 ARM 汇编实验

3.3.1 实验目的

- 掌握 ARM 汇编语言的基本使用;
- 熟悉 eclipse 开发工具建立汇编工程和仿真;

3.3.2 实验原理

根据上面阐述 RAM 汇编语言的使用语法和功能, 编写汇编程序, 实现一个简单的加法操作。

3.3.3 实验内容

1、汇编加法程序设计

2、ASM Code

```
.globl _start
_start:
2   mov r0, #9      //9 存入 r0
3   mov r1, #15     //15 存入 r1
```



```

4      add r1,r1,r0    //r1 与 r0 相加存入 r1
5  stop:
6      b stop
7
8

```

3.3.4 实验步骤

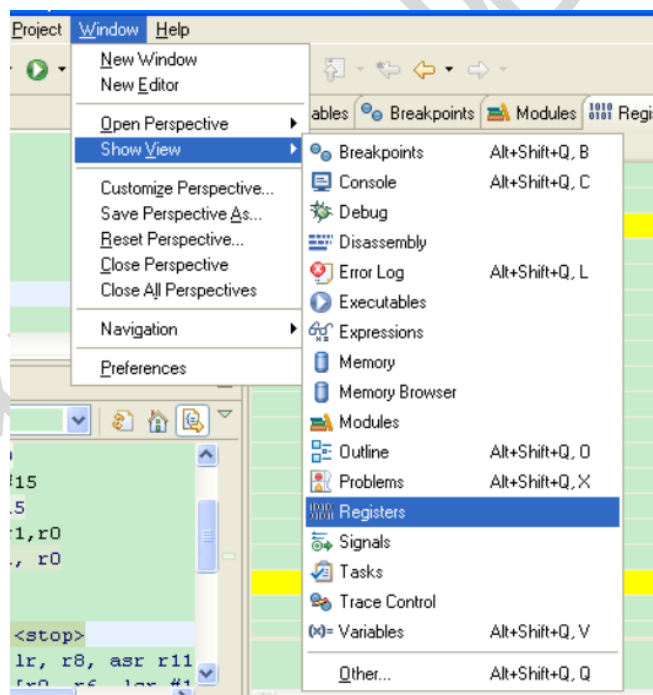
1、 导入光盘实验源码

方法参考第 1 章 ARM 环境搭建。光盘实验源码路径：**【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\01-add_asm】**

注意：ARM 仿真需要安装 ARM 开发环境，请参考**第 1 章 ARM 环境搭建**的部分。

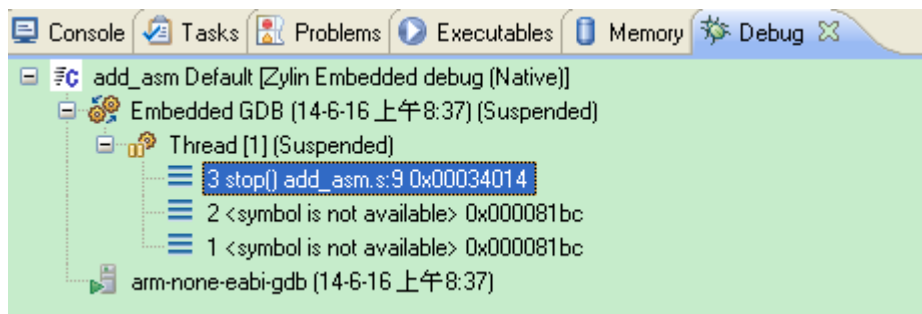
2、 打开“Register”显示框

单击 window -> show view -> Register,




3、 单步仿真


配置完成之后，点击 “ ” 开始仿真，弹出 Debug 框。

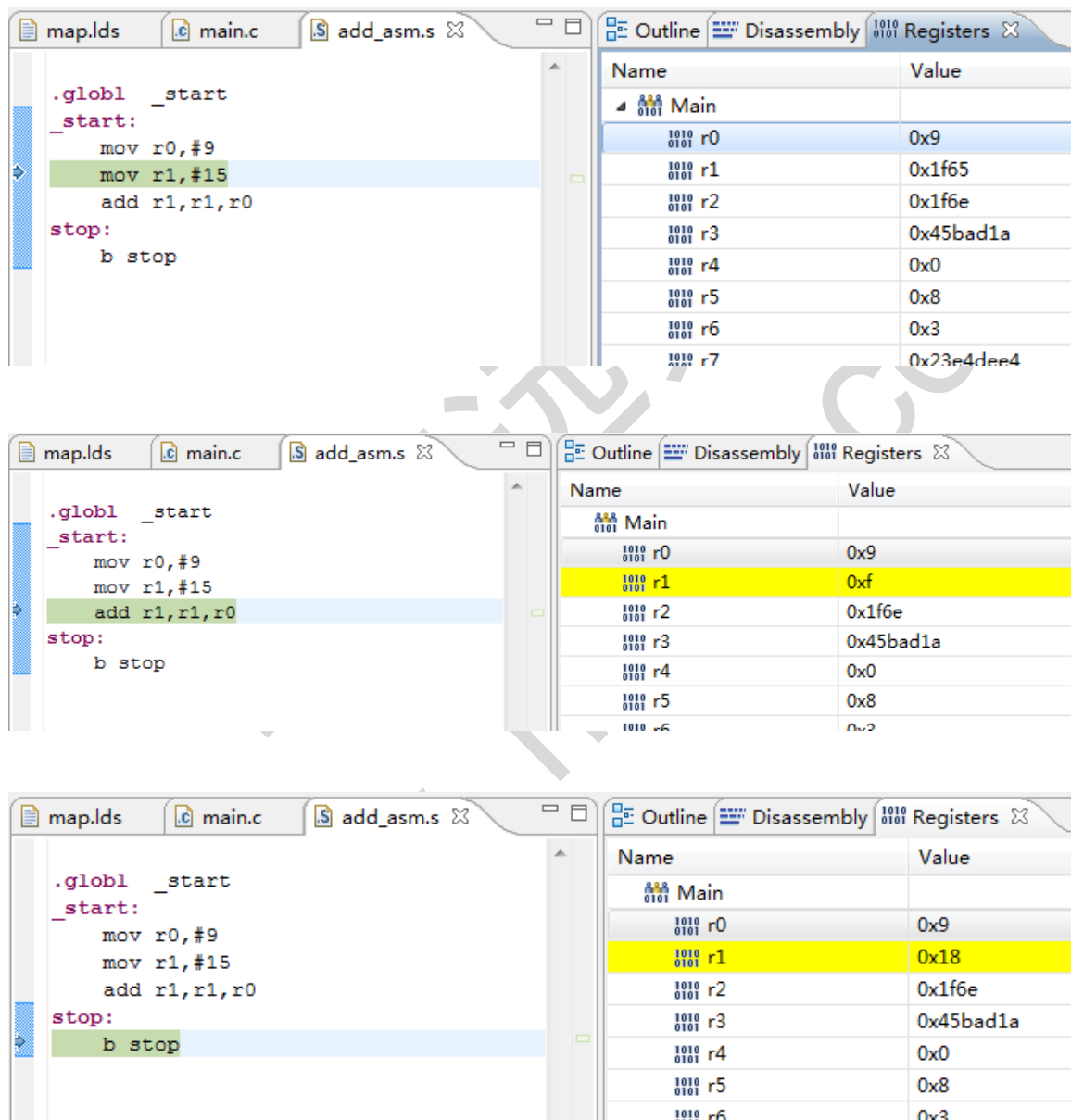




单击 “” 单步仿真。查看仿真现象。

3.3.5 实验现象

1、单击 “” 单步，查看 Rn 寄存器的变化。



Name	Value
Main	
r0	0x9
r1	0x1f65
r2	0x1f6e
r3	0x45bad1a
r4	0x0
r5	0x8
r6	0x3
r7	0x23e4dee4

Name	Value
Main	
r0	0x9
r1	0xf
r2	0x1f6e
r3	0x45bad1a
r4	0x0
r5	0x8
r6	0x3

Name	Value
Main	
r0	0x9
r1	0x18
r2	0x1f6e
r3	0x45bad1a
r4	0x0
r5	0x8
r6	0x3

单步运行可以看到 R0、R1 的值变化。

3.4 本章小结

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



3.5 练习题

- 4、 用 ARM 汇编实现下面列出的操作：
 - 1) $r0=15$
 - 2) $r0=r1/16$ (有符号数)
 - 3) $r1=r2*3$
 - 4) $r0=-r0$
- 5、 BIC 指令的作用是什么？
- 6、 执行 SWI 指令时会发生什么？
- 7、 B、BL、BX 指令的区别是什么？
- 8、 下面哪个数据可以作为数据操作指令的有效立即数：
a. 0x101 b. 0x1f8 c. 0xf000000f d. 0x08000012 e. 0x104
- 9、 ARM 在哪些工作模式下可以修改 CPSR 寄存器？
- 7、 写一个程序，判断 R0 的值，大于 0x50，则将 R1 的值减去 0x10，并把结果送给 R0。
- 8、 编写一段 ARM 汇编程序，实现数据块复制，将 R0 指向的 8 个字的连续数据保存到 R1 指向的一段连续的内存单元。



第 4 章 ARM 汇编语言程序设计

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

本章主要内容：

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

4.1 GNU ARM 汇编器支持的伪操作

4.1.1 伪操作概述

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

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

4.1.2 数据定义（Data Definition）伪操作

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

（1）伪指令名：

```
.byte
```

用途：单字节定义。

用法：

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

（2）伪指令名：

```
.short
```

用途：定义双字节数据，

用法：

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

（3）伪指令名：

```
.long
```

用途：定义 4 字节数据。

用法：

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



(4) 伪指令名:

```
.quad:
```

用途: 定义 8 字节。

用法:

```
.quad 0x1234567890abcd
```

(5) 伪指令名:

```
.float
```

用途: 定义浮点数。

用法:

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

(6) 伪指令名:

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

用途: 定义多个字符串。

用法:

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

(7) 伪指令名:

```
.rept/.endr
```

用途: 重复定义伪操作。

用法:

```
.rept 3
```

```
.byte 0x23
```

```
.endr
```

(8) 伪指令名:

```
.equ/.set
```

用途: 赋值语句, .equ(.set) 变量名, 表达式。

用法:

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

4.1.3 汇编控制伪操作

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

1、.if、.else、.endif

a) 语法格式

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



提示:

.if、.else、.endif 伪指令可以嵌套使用。

语法格式如下:

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

用于决定指令执行流程的逻辑表达式。

b) 使用说明

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

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

该形式避免了 if-else 形式的嵌套, 使程序结构更加清晰、易读。

2、.macro、.endm

a) 语法格式

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

宏操作可以使用一个或多个参数, 当宏操作被展开时, 这些参数被相应的值替换。

宏操作的使用方式和功能与子程序有些相似, 子程序可以提供模块化的程序设计、节省存储空间并提高运行速度。但在使用子程序结构时需要保护现场, 从而增加了系统的开销, 因此, 在代码较短且需要传递的参数较多时, 可以使用宏操作代替子程序。

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

提示:

.macro、.endm 伪操作可以嵌套使用。

语法格式如下:

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

(1) { \$label }。

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



(3) **Macroname:** 所定义的宏的名称。

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

b) 使用说明

在子程序代码比较短，而需要传递的参数比较多的情况下可以使用宏汇编技术。

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

c) 示例

示例如下：

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

3、.mexit

a) 语法格式

`.mexit` 用于从宏定义中跳转出去。

b) 用法

只需要在宏定义的代码中插入该指令即可。

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

4.1.4 杂项伪操作

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

<code>.arm</code>	<code>.arm</code>	@ 定义以下代码使用 ARM 指令集编译
<code>.code 32</code>	<code>.code 32</code>	@ 作用同 <code>.arm</code>
<code>.code 16</code>	<code>.code 16</code>	@ 作用同 <code>.thumb</code>
<code>.thumb</code>	<code>.thumb</code>	@ 定义以下代码使用 Thumb 指令集编译
<code>.section</code>	<code>.section expr</code>	@ 定义域中包含的段。expr 可以使 <code>.text</code> , <code>.data</code> , <code>.bss</code>
<code>.text</code>	<code>.text {subsection}</code>	@ 将定义符开始的代码编译到代码段或代码子段 (subsection)



```
.data      .data {subsection}      @将定义符开始的代码编译到数据段或数据子段
(subsection)

.bss       .bss {subsection}       @将变量存放到.bss段或.bss的子段(subsection)

.align     .align{alignment}{,fill}{,max} @通过用零或指定的数据进行填充来使当前位置
与指定边界对齐

.org       .org offset{,expr}      @指定从当前地址加上 offset 开始存放代码, 并且从当前地
址到当前地址加上 offset 之间的内存单元, 用零或指定的数据进行填充
```

4.2 ARM 汇编器支持的伪指令

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

4.2.1 ADR 伪指令

1、语法规式

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

语法规式如下:

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

- a) cond
可选的指令执行条件。
- b) .W
可选项。指定指令宽度 (Thumb-2 指令集支持)。
- c) register
目标寄存器。
- d) label
基于 PC 或具有寄存器的表达式。

2、使用说明

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

3、示例

示例如下。

```
LDR      R4,=data+4*n      ;n 是汇编时产生的变量
; code
MOV      pc,lr
data     .word     value0
```



```
; n-1 条 DCD 伪操作
.word    valuen          ;所要装载入 R4 的值
;更多 DCD 伪操作
```

4.2.2 ADRL 伪指令

1、语法格式

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

语法格式如下：

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

a) cond

可选的指令执行条件。

b) register

目标寄存器。

c) label

基于 PC 或具体寄存器的表达式。

2、使用说明

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

4.2.3 LDR 伪指令

1、语法格式

LDR 伪指令装载一个 32 位的常数和地址到寄存器。

语法格式如下：

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

(1) Cond

可选的指令执行条件。

(2) .W

可选项。指定指令宽度（Thumb-2 指令集支持）。

(3) register

目标寄存器。



(4) expr

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

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

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

(5) label-expr

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

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

2、使用说明

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

由 `LDR` 指令装载的地址是绝对地址，即 `PC` 相关地址。

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

3、示例

(1) 将常数 `0xff0` 读到 `R1` 中。

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

相当于下面的 `ARM` 指令：

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

(2) 将常数 `0xffff` 读到 `R1` 中。

```
LDR R1,=0xffff ;
```

相当于下面的 `ARM` 指令：

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

(3) 将 `place` 标号地址读入 `R1` 中。

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

相当于下面的 `ARM` 指令：

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



4.3 GNU ARM 汇编语言的语句格式

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

(1) 符号区分大小写,同名的大、小写符号会被编译器认为是两个不同的符号。

(2) 符号在其作用范围内必须唯一。

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

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

注意: 虽然符号被双斜杠括起来,但双斜杠并非符号名的一部分。

(5) 局部标号以数字开头,其他的符号都不能以数字开头。

1、变量(variable)

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

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

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

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

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

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

2、常量(constants)

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

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

逻辑常量只有两种取值情况,真或假。

字符串常量为一个固定的字符串,一般用于程序运行时的信息提示。

3、程序中的变量代换



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

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

4、程序标号（label）

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

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

（1）程序相关标号

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

（2）寄存器相关地址

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

（3）绝对地址

绝对地址是一个 32 位的数字量，使用它可以直接寻址整个内存空间。

5、局部标号

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

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

4.4 ARM 汇编语言的程序结构

4.4.1 汇编语言的程序格式

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

在一个段中，我们可以定义下列的子段：

- .text
- .data



- .bss
- .sdata
- .sbss

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

```
.section.data
< initialized data here>
.section .bss
< uninitialized data here>
.section .text
.globl _start
_start:
<instruction code goes here>
```

4.4.2 汇编语言子程序调用

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

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

注意：

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

以下是使用 BL 指令调用子程序的汇编语言源程序的基本结构：

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



...

4.4.3 过程调用标准 AAPCS

为了使不同编译器编译的程序之间能够相互调用，必须为子程序间的调用规定一定的规则。AAPCS 就是这样一个标准。所谓 AAPCS，其英文全称为 Procedure Call Standard for the ARM Architecture (AAPCS)，即 ARM 体系结构过程调用标准。它是 ABI (Application Binary Interface (ABI) for the ARM Architecture (base standard) [BSABI]) 标准的一部分。

可以使用“--apcs”选项告诉编译器将源代码编译成符号 AAPCS 调用标准的目标代码。

注意：

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

1、AAPCS 相关的编译/汇编选项

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

2、ARM 寄存器使用规则

AAPCS 中定义了 ARM 寄存器使用规则如下：

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

在子程序中，使用寄存器 R4-R11 来保存局部变量。如果在子程序中使用到了寄存器 R4-R11 中的某些寄存器，子程序进入时必须保存这些寄存器的值，在返回前必须恢复这些寄存器的值；对于子程序中



用到的寄存器则不必进行这些操作。在 Thumb 程序中，通常只能使用寄存器 R4-R7 来保存局部变量。

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

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

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

寄存器 R15 是程序计数器，记作 pc。它不能用做其他用途。

ARM 寄存器在函数调用过程中的保护规则，如图 4-1 所示。

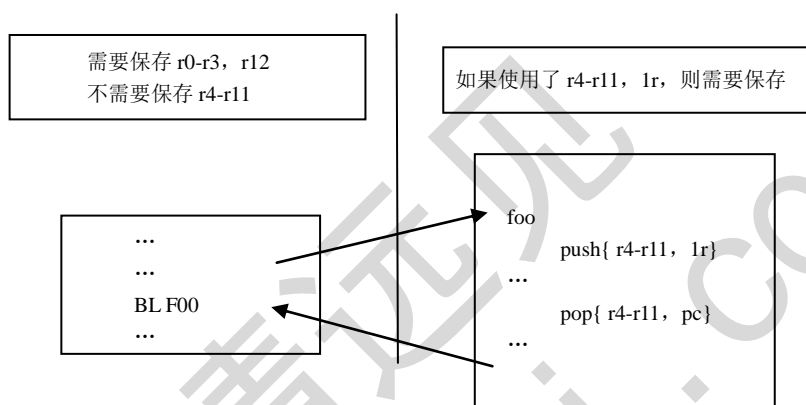


图 4-1 ARM 寄存器在函数调用中的保护规则

4.4.4 汇编语言程序设计举例

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

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

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

用 ARM 汇编语言重写来重写这个例子，如下所示。

```
Code1:
Gcd:
    CMP    r0, r1
```



```

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

充分地利用条件执行修改上面的例子，得到 Code2。

```

Code2:
Gcd:
        CMP      r0, r1
        SUBGT    r0, r0, r1
        SUBLT    r1, r1, r0
        BNE      gcd
    
```

两段代码的比较如下。

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

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

4.5 汇编语言与 C 语言的混合编程

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

4.5.1 GNU ARM 内联汇编

1、内联汇编语法

本小节简单介绍 GNU 风格的 ARM 内联汇编语法要点：

(1) 格式

格式如下：

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

必须以 “;” 结尾，不管有多长对 C 都只是一条语句。

(2) asm 内嵌汇编关键字

volatile: 告诉编译器不要优化内嵌汇编，如果想优化可以不加。

(3) ANSI C 规范的关键字

ANSI C 规范的关键字如下。

```

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

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



没有 changed 必须省略 “:”。

2、汇编代码

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

```
"mov r0,r0\n\t" //指令之间必须要换行，\t 可以不加，只是为了在汇编文件中的指令格式
对齐
"mov r1,r1\n\t"
"mov r2,r2"
```

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

注意：编译器不检查 asm code 的内容是否合法，直接交给汇编器

3、output (ASM --> C) 和 input (C --> ASM)

(1) 指定输出值：

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

① constraint 定义 variable 的存放位置：

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

② output 修饰符：

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

(2) 指定输入值：

```
__asm__ __volatile__ (
    "asm code"
    :
    : "constraint" (variable / immediate)
);
```

constraint 定义 variable / immediate 的存放位置：

```
r :    使用任何可用的通用寄存器（变量和立即数都可以）
m :    使用变量的内存地址（不能用立即数）
i :    使用立即数（不能用变量）
```

(3) 使用占位符：

```
int a = 100,b = 200;
```




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

(4) 引用占位符:

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

(5) & 修饰符:

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

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

4、内联汇编示例

下面通过一个例子进一步了解内联汇编的语法。该例子实现了位交换。

```
#include <stdio.h>
unsigned long ByteSwap(unsigned long val)
{
    int ch;
    asm volatile (
        "eor r3, %1, %1, ror #16\n\t"
        "bic r3, r3, #0x00FF0000\n\t"
        "mov %0, %1, ror #8\n\t"
        "eor %0, %0, r3, lsr #8"
```



```

: "=r" (val)
: "0" (val)
: "r3"
);
}
int main(void)
{
    unsigned long test_a = 0x1234,result;
    result = ByteSwap(test_a);
    printf("Result:%d\r\n", result);
    return 0;
}

```

4.5.2 混合编程调用举例

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

1、从 C 程序调用汇编语言

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

```

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

```

下面为调用的汇编程序。

```

.global strcpy
strcpy:                                ;R0 指向目的字符串
                                        ;R1 指向源字符串

    LDRB R2, [R1],#1                  ;加载字节并更新源字符串指针地址
    STRB R2, [R0],#1                  ;存储字节并更新目的字符串指针地址
    CMP R2, #0                        ;判断是否为字符串结尾
    BNE strcpy                        ;如果不是，程序跳转到 strcpy 继续复制

```



```
MOV pc,lr           ;程序返回
```

2、从汇编语言调用 C 程序

下面的例子显示了如何从汇编语言调用 C 程序。

下面的子程序段定义了 C 语言函数。

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

下面的程序段显示了汇编语言调用。假设程序进入 f 时，R0 中的值为 i。

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

4.6 ARM 伪指令实验

4.6.1 实验目的

- 掌握 ARM 汇编语言的基本使用和一些伪指令的使用；
- 熟悉 eclipse 开发工具建立汇编工程和仿真；

4.6.2 实验原理

根据上面阐述 RAM 汇编语言的使用语法和功能，编写汇编程序，实现将存放在两个内存中的数据相加的操作。

4.6.3 实验内容

1、汇编加法程序设计

```
.text
.global _start
_start:
2   mov     r3,r3
3
```



```

4      ldr    r0,=myarray
5      mov    r4,#0
6 loop:
7      ldr    r1,[r0],#4
8      add    r4,r4,r1
9      cmp    r1,#0
10     bne    loop
11 stop:
12     b      stop
13 myarray:
14     .word   0x11
15     .word   0x22
16     .word   0x00
      .end
    
```

4.6.4 实验步骤

1、导入工程源码和仿真测试

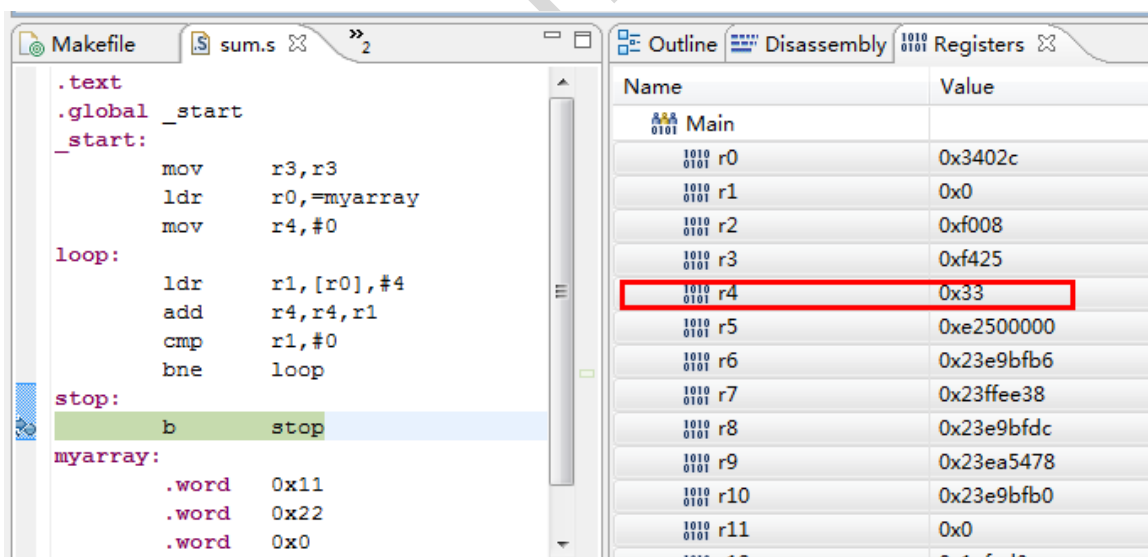
请参考第 1 章节的 ARM 开发环境搭建部分

光盘实验源码路径：【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\03-sum】

注意：ARM 仿真需要安装 ARM 开发环境，请参考第 1 章节的 ARM 开发环境搭建部分。

4.6.5 实验现象

单击 “” 单步，查看 Rn 寄存器的变化。



Name	Value
Main	
r0	0x3402c
r1	0x0
r2	0xf008
r3	0xf425
r4	0x33
r5	0xe2500000
r6	0x23e9bfb6
r7	0x23ffee38
r8	0x23e9bfdc
r9	0x23ea5478
r10	0x23e9bfb0
r11	0x0

两个数据的和保存在 r4，最终 r4 的值为 0x33。



4.7 本章小结

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

4.8 思考题

1. 在 GNU 风格的 ARM 汇编中如何定义一个全局的数字变量？
2. AAPCS 中规定的 ARM 寄存器的使用规则是什么？
3. 什么是内联汇编？什么是嵌入型汇编？两者之间的区别是什么？
4. 汇编代码中如何调用 C 代码中定义的函数？

华清远见
dev.hqyj.com



第 5 章 GPIO 编程

GPIO 控制技术是接口技术中最简单的一种。本章通过介绍 Exynos4412 芯片的 GPIO 控制方法，让读者初步掌握控制硬件接口的方法。本章的主要内容：

- (1) GPIO 功能介绍。
- (2) Exynos4412 芯片的 GPIO 控制器详解。
- (3) Exynos4412 的 GPIO 应用。

5.1 GPIO 功能介绍

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

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

5.2 Exynos4412 芯片的 GPIO 控制器详解

5.2.1 特性

Exynos4412 的 GPIO 特性包括如下几点：

- (1) 304 个多功能输入输出 GPIO
- (2) 37 组通用 GPIO 和 2 组 memory GPIO。

5.2.2 GPIO 分组预览

- (1) GPA0, GPA1: 14 in/out ports-3xUART with flow control, UART without flow control, and/ or 2xI2C
- (2) GPB: 8 in/out ports-2xSPI and/ or 2xI2C and/ or IEM
- (3) GPC0, GPC1: 10 in/out ports-2xI2S, and/ or 2xPCM, and/ or AC97, SPDIF, I2C, and/ or SPI
- (4) GPD0, GPD1: 8 in/out ports-PWM, 2xI2C, and/ or LCD I/F, MIPI
- (5) GPM0, GPM1, GPM2, GPM3, GPM4: 35 in/out ports-CAM I/F, and/ or TS I/F, HSI, and/ or Trace I/F
- (6) GPF0, GPF1, GPF2, GPF3: 30 in/out ports-LCD I/F



- (7) GPJ0, GPJ1: 13 in/out ports-CAM I/F
- (8) GPK0, GPK1, GPK2, GPK3: 28 in/out ports-4xMMC (4-bit MMC), and/ or 2xMMC (8-bit MMC), and/ or GPS debugging I/F
- (9) GPL0, GPL1: 11 in/out ports-GPS I/F
- (10) GPL2: 8 in/out ports-GPS debugging I/F or Key pad I/F
- (11) GPX0, GPX1, GPX2, GPX3: 32 in/out ports-External wake-up, and/ or Key pad I/F

5.2.3 Exynos4412 的 GPIO 常用寄存器分类

(1) 端口控制寄存器 (GPA0CON-GPZCON)

在 Exynos4412 中,大多数的引脚都可复用,所以必须对每个引脚进行配置。端口控制寄存器(GPnCON)定义了每个引脚的功能。

(2) 端口数据寄存器 (GPA0DAT-GPZDAT)

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

(3) 端口上拉寄存器 (GPA0PUD - GPZPUD)

端口上拉寄存器控制了每个端口组的上拉/下拉电阻的使能/禁止。根据对应位的 0/1 组组合,设置对应端口的上拉/下拉电阻功能是否使能。如果端口的上拉电阻被使能,无论在何种状态(输入、输出、DATAn、EINTn 等)下,上拉电阻都起作用。

(4) 驱动能力寄存器 (GPA0DRV - GPZDRV)

设置 GPIO 口的驱动能力。

5.2.4 GPIO 功能描述

GPIO 功能概括图如图所示。

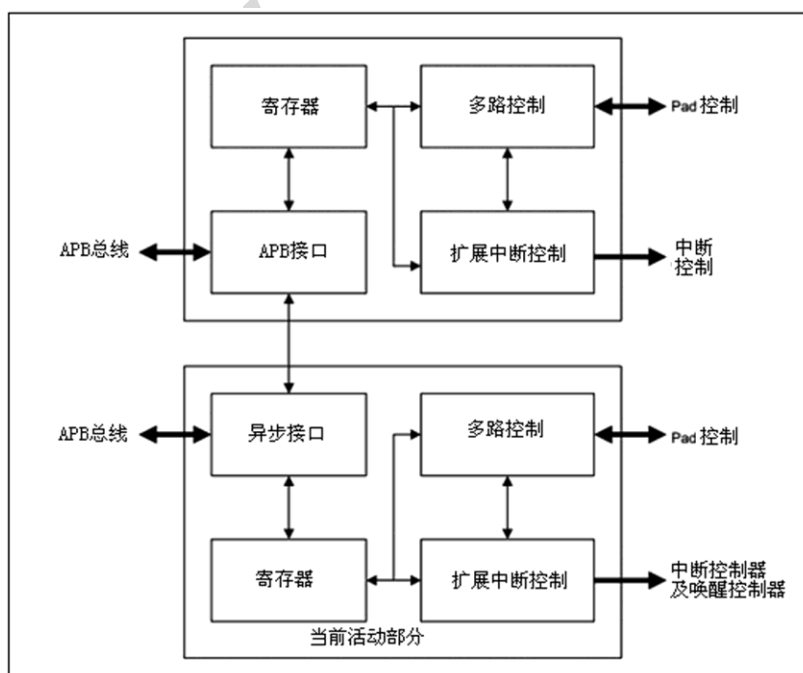




图 GPIO 功能概括图

5.2.5 Exynos4412 I/O 接口常用寄存器详解

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

表 GPF3CON 控制寄存器(可读/可写 Address = 0x1140_0x01E0)

GPC3CON	位	描述	初始状态
GPF3CON[5]	[23:20]	0x0 = Input 0x1 = Output 0x2 = SYS_OE 0x3 to 0xE = Reserve 0xF = EXT_INT16[5]	0000
GPF3CON[4]	[19:16]	0x0 = Input 0x1 = Output 0x2 = VSYNC_LDI 0x3 to 0xE = Reserved 0xF = EXT_INT16[4]	0000
GPF3CON[3]	[15:12]	0x0 = Input 0x1 = Output 0x2 = LCD_VD[23] 0x3 to 0xE = Reserved 0xF = EXT_INT16[3]	0000
GPF3CON[2]	[11:8]	0x0 = Input 0x1 = Output 0x2 = LCD_VD[22] 0x3 to 0xE = Reserved 0xF = EXT_INT16[2]	0000
GPF3CON[1]	[7:4]	0x0 = Input 0x1 = Output 0x2 = LCD_VD[21] 0x3 to 0xE = Reserved 0xF = EXT_INT16[1]	0000
GPF3CON[0]	[3:0]	0x0 = Input 0x1 = Output 0x2 = LCD_VD[20] 0x3 to 0xE = Reserved 0xF = EXT_INT16[0]	0000

5.2.6 GPIO 数据寄存器

GPIO 数据寄存器如表所示。

表 GPF3DAT 数据寄存器(可读/可写 Address = 0x1140_01E4)

GPF3DAT	位	描述	初始状态
GPF3DAT[5:0]	[5:0]	该寄存器决定了输入或者输出的电平状态	0x00

5.3 GPIO 控制实验

通过第 4 章的介绍，读者了解了 GPIO 的功能，以及 GPF3DAT 芯片 GPIO 控制器的配置方法。本章通过一个简单示例说明 Exynos4412 的 GPIO 接口的应用。

5.3.1 实验目的

示例将利用 Exynos4412 的 GPC0_3、GPC0_4 这 2 个 I/O 引脚控制 2 个 LED 发光二极管，使其有规律地闪烁。

5.3.2 实验原理

如图所示，LED2~LED5 分别与 GPX2_7、GPX1_0、GPF3_4、GPF3_5 相连，通过 GPX2_7、GPX1_0、GPF3_4、GPF3_5 引脚的高低电平来控制三极管的导通性，从而控制 LED 的亮灭。

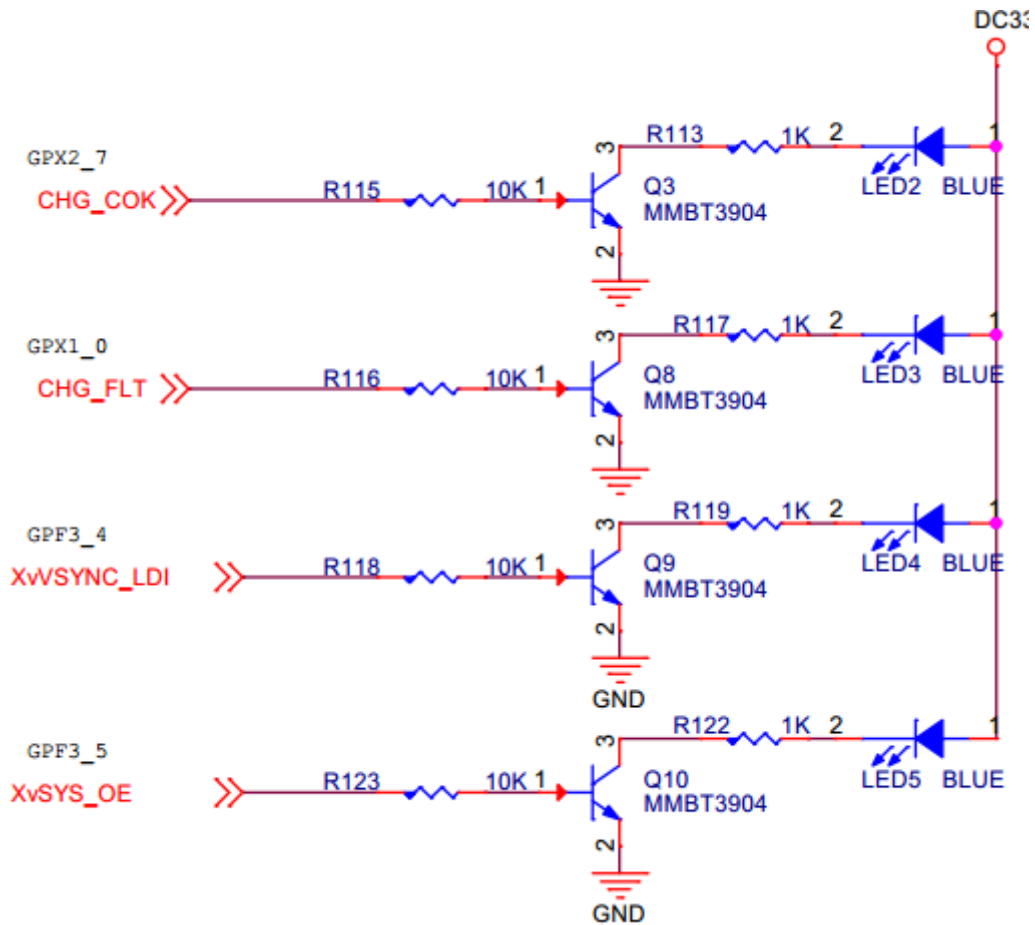


图 LED 接线原理图

根据三极管的特性，当这几个引脚输出高电平时，集电极和发射极导通，发光二极管点亮；反之，发光二极管熄灭。

通过控制 GPX1CON、GPX2CON、GPF3CON 和 GPX1DAT 来控制 GPX2_3 和 GPF3_4 对应的 LED。

6.2.3.198 GPX1CON

- Base Address: 0x1100_0000
- Address = Base Address + 0x0C20, Reset Value = 0x0000_0000

GPX1CON[0]	[3:0]	RW	0x0 = Input 0x1 = Output 0x2 = Reserved 0x3 = KP_COL[0] 0x4 = Reserved 0x5 = ALV_DBG[4] 0x6 to 0xE = Reserved 0xF = WAKEUP_INT1[0]	0x00
------------	-------	----	---	------

6.2.3.199 GPX1DAT

- Base Address: 0x1100_0000
- Address = Base Address + 0x0C24, Reset Value = 0x00

Name	Bit	Type	Description	Reset Value
GPX1DAT[7:0]	[7:0]	RWX	When you configure port as input port then corresponding bit is pin state. When configuring as output port then pin state should be same as corresponding bit. When the port is configured as functional pin, the undefined value will be read.	0x00



6.2.3.202 GPX2CON

- Base Address: 0x1100_0000
- Address = Base Address + 0x0C40, Reset Value = 0x0000_0000

Name	Bit	Type	Description	Reset Value
GPX2CON[7]	[31:28]	RW	0x0 = Input 0x1 = Output 0x2 = Reserved 0x3 = KP_ROW[7] 0x4 = Reserved 0x5 = ALV_DBG[19] 0x6 to 0xE = Reserved 0xF = WAKEUP_INT2[7]	0x00

6.2.3.203 GPX2DAT

- Base Address: 0x1100_0000
- Address = Base Address + 0x0C44, Reset Value = 0x00

Name	Bit	Type	Description	Reset Value
GPX2DAT[7:0]	[7:0]	RWX	When you configure port as input port then corresponding bit is pin state. When configuring as output port then pin state should be same as corresponding bit. When the port is configured as functional pin, the undefined value will be read.	0x00

5.3.3 实验内容

(1) 寄存器设置

为了实现控制 LED 的目的，需要通过配置 GPX1CON、GPX2CON、GPF3CON 寄存器将 GPX2_7、GPX1_0、GPF3_4、GPF3_5 设置为输出属性。通过设置对应的 DAT 寄存器实现点亮与熄灭 LED。

对于本例来说，各个 GPIO 的上拉寄存器可以不用设置。

(2) 程序编写

相关代码如下：

C++ Code

```
#include "exynos_4412.h"

2
3 /*****
4  * @brief      mydelay_ms program body
5  * @param[in]  int (ms)
6  * @return     None
7  *****/
8 void mydelay_ms(int ms)
9 {
10     int i, j;
11     while(ms--)
12     {
13         for (i = 0; i < 5; i++)
14             for (j = 0; j < 514; j++);
15     }
16 }
```



```

15     }
16 }
17
18 /*-----MAIN FUNCTION-----*/
19 /*****
20  * @brief      Main program body
21  * @param[in]  None
22  * @return     int
23  *****/
24 int main(void)
25 {
26     /*
27      *Config
28      */
29
30     GPX2.CON = (GPX2.CON & ~(0xf<<28)) | 1<<28; //GPX2_7:output, LED2
31     GPX1.CON = (GPX1.CON & ~(0xf)) | 1; //GPX1_0:output, LED3
32     GPF3.CON = (GPX3.CON & ~(0xf<<16 | 0xf<<20)) | (1<<16 | 1<<20); //GPF3_4:output, LED4
33                                           //GPF3_5:output, LED5
34
35     while(1)
36     {
37         //Turn on LED2
38         GPX2.DAT |= 0x1 << 7;
39         mydelay_ms(500);
40
41         //Turn on LED3
42         GPX1.DAT |= 0x1;
43         //Turn off LED2
44         GPX2.DAT &= ~(0x1<<7);
45         mydelay_ms(500);
46
47         //Turn on LED5
48         GPF3.DAT |= (0x1 << 5);
49         //Turn off LED3
50         GPX1.DAT &= ~0x1;
51         mydelay_ms(500);
52
53         //Turn on LED4
54         GPF3.DAT |= (0x1 << 4);
55         //Turn off LED5
56         GPF3.DAT &= ~(0x1 << 5);
57         mydelay_ms(500);
58

```



```
59      //Turn off LED4
60      GPF3.DAT &= ~(0x1 << 4);
61  }
62  return 0;
63 }
64
65
```

5.3.4 实验步骤

1、 导入工程源码

请参考第 1 章节的 ARM 开发环境搭建部分。

光盘实验源码路径：【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\04-led】

2、 连接好开发板及 FS-JTAG 仿真器，并且连接好配线。

方法请参考第 1 章节的开发环境搭建的硬件连接部分按对应型号连接。

3、 仿真运行程序

方法请参考第 1 章节的 ARM 开发环境搭建部分。

5.3.5 实验现象

用 FS-JTAG 仿真器仿真程序，可以看到 LED 灯有规律的闪动。





第 6 章 ARM 异常及中断处理

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

本章主要内容：

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

6.1 ARM 异常中断处理概述

(1) 中断的概念

什么是中断，我们从一个生活中的例子引入。你正在家中看书，突然电话铃响了，你放下书本，去接电话，和来电话的人交谈，然后放下电话，回来继续看你的书。这就是生活中的“中断”的现象，就是正常的工作过程被外部的事件打断了。

在处理器中，所谓中断，是一个过程，即 CPU 在正常执行程序的过程中，遇到外部 / 内部的紧急事件需要处理，暂时中断（中止）当前程序的执行，而转去为事件服务，待服务完毕，再返回到暂停处（断点）继续执行原来的程序。为事件服务的程序称为中断服务程序或中断处理程序。严格地说，上面的描述是针对硬件事件引起的中断而言的。用软件方法也可以引起中断，即事先在程序中安排特殊的指令，CPU 执行到该类指令时，转去执行相应的一段预先安排好的程序，然后再返回来执行原来的程序，这可称为软中断。把软中断考虑进去，可给中断再下一个定义：中断是一个过程，是 CPU 在执行当前程序的过程中因硬件或软件的原因插入了另一段程序运行的过程。因硬件原因引起的中断过程的出现是不可预测的，即随机的，而软中断是事先安排的。

(2) 中断源的概念

仔细研究一下生活中的中断，对于理解中断的概念也很有好处。什么可以引起中断，生活中很多事件可以引起中断：

有人按门铃了，电话铃响了，你的闹钟响了，你烧的水开了……诸如此类的事件。我们把可以引起中断的信号源称为中断源。

(3) 中断优先级的概念

设想一下，我们正在看书，电话铃响了，同时又有人按了门铃，你该先做什么呢？如果你正在等一个很重要的电话，一般不会去理会门铃；反之，如果你正在等一位重要的客人，则可能就不会去理会电话了。如果不是这两者（既不等电话，也不等人上门），你可能会按你通常的习惯去处理。总之，这里存在一个优先级的问題，在处理器中也是如此，也有优先级的问題。即同时有多个中断源递交中断申请时的中断控



制器对中断源的响应优先级。需要注意的是，优先级的问題不仅仅发生在两个中断同时产生的情况，也发生在一个中断已产生，又有一个中断产生的情况。比如，你正接电话，有人按门铃的情况，或你正开门与人交谈，又有电话响了的情况。这时也需要根据中断源的优先级来决定下一动作。

ARM 处理器中有 7 种类型的异常，按优先级从高到低的排列如下：复位异常（Reset）、数据异常（Data Abort）、快速中断异常（FIQ）、外部中断异常（IRQ）、预取异常（Prefetch Abort）、软中断异常（SWI）和未定义指令异常（Undefined interrupt）。

注意：在 ARM 处理器中，异常（Exception）和中断（Interrupt）有些差别，异常主要是从处理器被动接受异常的角度出发，而中断带有向处理器主动申请的色彩。在本书中，对“异常”和“中断”不做严格区分，两者都是指请求处理器打断正常的程序执行流程，进入特定程序循环的一种机制。

6.2 ARM 体系异常种类

在 ARM 体系结构中，存在 7 种异常处理。当异常发生时，处理器会把 PC 设置为一个特定的存储器地址。这一地址放在被称为向量表（vector table）的特定地址范围内。向量表的入口是一些跳转指令，跳转到专门处理某个异常或中断的子程序。

存储器映射地址 0x00000000 是为向量表（一组 32 位字）保留的。在有些处理器中，向量表可以选择定位在存储空间的高地址（从偏移量 0xffff0000 开始）。一些嵌入式操作系统，如 Linux 和 Windows CE 就利用了这一特性。

注意：Cortex-A8/A9 系统中支持通过设置 CP15 的 C12 寄存器将异常向量表的首地址设置在 32 字节对齐的任意地址。下文标记为 C12（CP15）。为了保持继承，下文还是会按传统的 0 和 0xFFFF0000 介绍如表所示列出了 ARM 的 7 种异常类型。

表 ARM 的 7 种异常类型

异常类型	处理器模式	执行低地址	执行高地址
复位异常（Reset）	特权模式	0x00000000	0xFFFF0000
未定义指令异常 （Undefined Interrupt）	未定义指令中止模式	0x00000004	0xFFFF0004
软中断异常（SWI）	特权模式	0x00000008	0xFFFF0008
预取异常（Prefetch Abort）	数据访问中止模式	0x0000000C	0xFFFF000C
数据异常（Data Abort）	数据访问中止模式	0x00000010	0xFFFF0010
外部中断异常（IRQ）	外部中断请求模式	0x00000018	0xFFFF0018
快速中断异常（FIQ）	快速中断请求模式	0x0000001C	0xFFFF001C

异常处理向量表如图所示。



图 异常处理向量表

当异常发生时，分组寄存器 r14 和 SPSR 用于保存处理器状态，操作伪指令如下：

R14_<exception_mode> = return link

SPSR_<exception_mode> = CPSR

CPSR[4 : 0] = exception mode number

CPSR[5] = 0 /*进入 ARM 状态*/

If <exception_mode> == reset or FIQ then

CPSR[6] = 1 /*屏蔽快速中断 FIQ*/

CPSR[7] = 1 /*屏蔽外部中断 IRQ*/

PC = exception vector address

异常返回时，SPSR 内容恢复到 CPSR，连接寄存器 r14 的内容恢复到程序计数器 PC。

1. 复位异常

当处理器的复位引脚有效时，系统产生复位异常中断，程序跳转到复位异常中断处理程序处执行。复位异常中断通常用于系统上电和系统复位两种情况。

当复位异常时，系统（处理器自动执行的，以下几个异常相同）执行下列伪操作。

R14_svc = UNPREDICTABLE value

SPSR_svc = UNPREDICTABLE value

CPSR[4 : 0] = 0b10011 /*进入特权模式*/

CPSR[5] = 0 /*处理器进入 ARM 状态*/

CPSR[6] = 1 /*禁止快速中断*/

CPSR[7] = 1 /*禁止外设中断*/



If high vectors configured then

PC = 0xffff0000

Else

PC = 0x00000000

复位异常中断处理程序将进行一些初始化工作，内容与具体系统相关。下面是复位异常中断处理程序的主要功能。

- ① 设置异常中断向量表。
- ② 初始化数据栈和寄存器。
- ③ 初始化存储系统，如系统中的 MMU 等。
- ④ 初始化关键的 I/O 设备。
- ⑤ 使能中断。
- ⑥ 处理器切换到合适的模式。
- ⑦ 初始化 C 变量，跳转到应用程序执行。

2. 未定义指令异常

当 ARM 处理器执行协处理器指令时，它必须等待一个外部协处理器应答后，才能真正执行这条指令。

若协处理器没有响应，则发生未定义指令异常。未定义指令异常可用于在没有物理协处理器的系统上，对协处理器进行软件仿真，或通过软件仿真实现指令集扩展。例如，在一个不包含浮点运算的系统中，CPU 遇到浮点运算指令时，将发生未定义指令异常中断，在该未定义指令异常中断的处理程序中可以通过其他指令序列仿真浮点运算指令。

仿真功能可以通过下面步骤实现。

(1) 将仿真程序入口地址链接到向量表中未定义指令异常中断入口处 (0x00000004 或 0xffff0004)，并保存原来的中断处理程序。

(2) 读取该未定义指令的 bits[27:24]，判断其是否是一条协处理器指令。如果 bits[27:24] 值为 0b1110 或 0b110x，该指令是一条协处理器指令；否则，由软件仿真实现协处理器功能，可以通过 bits[11:8] 来判断要仿真的协处理器功能（类似于 SWI 异常实现机制）。

(3) 如果不仿真该未定义指令，程序跳转到原来的未定义指令异常中断的中断处理程序行。

当未定义指令异常发生时，系统执行下列伪操作。

r14_und = address of next instruction after the undefined instruction

SPSR_und = CPSR

CPSR[4:0] = 0b11011 /*进入未定义指令模式*/

CPSR[5] = 0 /*处理器进入 ARM 状态*/

/*CPSR[6]保持不变*/

CPSR[7] = 1 /*禁止外设中断*/

If high vectors configured then

PC = 0xffff0004



Else

PC = 0x00000004

3. 软中断异常

软中断异常发生时，处理器进入特权模式，执行一些特权模式下的操作系统功能。软中断异常发生时，处理器执行下列伪操作。

r14_svc = address of next instruction after the SWI instruction

SPSR_und = CPSR

CPSR[4:0] = 0b10011 /*进入特权模式*/

CPSR[5] = 0 /*处理器进入 ARM 状态*/

/*CPSR[6]保持不变*/

CPSR[7] = 1 /*禁止外设中断*/

If high vectors configured then

PC = 0xffff0008

Else

PC = 0x00000008

4. 预取异常

预取异常是由系统存储器报告的。当处理器试图去取一条被标记为预取无效的指令时，发生预取异常。

如果系统中不包含 MMU，指令预取异常中断处理程序只是简单地报告错误并退出；若包含 MMU，引起异常的指令的物理地址被存储到内存中。

预取异常发生时，处理器执行下列伪操作。

r14_svc = address of the aborted instruction + 4

SPSR_und = CPSR

CPSR[4:0] = 0b10111 /*进入特权模式*/

CPSR[5] = 0 /*处理器进入 ARM 状态*/

/*CPSR[6]保持不变*/

CPSR[7] = 1 /*禁止外设中断*/

If high vectors configured then

PC = 0xffff000C

Else

PC = 0x0000000C

5. 数据异常

数据异常时由存储器发出数据中止信号，它由存储器访问指令 Load/Store 产生。当数据访问指令的目标地址不存在或者该地址不允许当前指令访问时，处理器产生数据访问中止异常。当数据异常发生时，处理器执行下列伪操作。

r14_abt = address of the aborted instruction + 8



SPSR_abt = CPSR

CPSR[4 : 0] = 0b10111

CPSR[5] = 0

/*CPSR[6]保持不变*/

CPSR[7] = 1 /*禁止外设中断*/

If high vectors configured then

 PC = 0xffff000C10

Else

 PC = 0x00000010

当数据访问中止异常发生时，寄存器的值将根据以下规则进行修改。

(1) 返回地址寄存器 r14 的值只与发生数据异常的指令地址有关，与 PC 值无关。

(2) 如果指令中没有指定基址寄存器回写，则基址寄存器的值不变。

(3) 如果指令中指定了基址寄存器回写，则寄存器的值和具体芯片的 Abort Models 有关，由芯片的生产商指定。

(4) 如果指令只加载一个通用寄存器的值，则通用寄存器的值不变。

(5) 如果是批量加载指令，则寄存器中的值不可预知。

(6) 如果指令加载协处理器寄存器的值，则被加载寄存器的值不可预知。

6. 外部中断异常

当处理器的外部中断请求引脚有效，而且 CPSR 寄存器的 I 控制位被清除时，处理器产生外部中断异常。系统中各外部设备通常通过该异常中断请求处理器服务。

当外部中断异常发生时，处理器执行下列伪操作。

r14_irq = address of next instruction to be executed + 4

SPSR_irq = CPSR

CPSR[4 : 0] = 0b10010 /*进入特权模式*/

CPSR[5] = 0 /*处理器进入 ARM 状态*/

/*CPSR[6]保持不变*/

CPSR[7] = 1 /*禁止外设中断*/

If high vectors configured then

 PC = 0xffff0018

Else

 PC = 0x00000018

7. 快速中断异常

当处理器的快速中断请求引脚有效且 CPSR 寄存器的 F 控制位被清除时，处理器产生快速中断异常。当快速中断异常发生时，处理器执行下列伪操作。

r14_fiq = address of next instruction to be executed + 4



SPSR_fiq = CPSR

CPSR[4 : 0] = 0b10001 /*进入 FIQ 模式*/

CPSR[5] = 0

CPSR[6] = 1

CPSR[7] = 1

If high vectors configured then

PC = 0xffff001c

Else

PC = 0x0000001c

7.3 ARM 异常的优先级

每一种异常按如表所示中设置的优先级得到处理。

表 异常优先级

优 先 级	异 常
最高 1	复位异常
2	数据异常
3	快速中断异常
4	外部中断异常
5	预取异常
6	软中断异常
最低 7	未定义指令异常

异常可以同时发生，此时处理器按表 7-2 中设置的优先级顺序处理异常。例如，处理器上电时发生复位异常，复位异常的优先级最高，所以当产生复位时，它将优先于其他异常得到处理。同样，当一个数据异常发生时，它将优先于除复位异常外的其他所有异常而得到处理。

优先级最低的两种异常是软件中断异常和未定义指令异常。因为正在执行的指令不可能既是一条软中断指令，又是一条未定义指令，所以软中断异常和未定义指令异常享有相同的优先级。

7.4 ARM 处理器模式和异常

每一种异常都会导致内核进入一种特定的模式。ARM 处理器异常及其对应的模式如表所示。此外，也可以通过编程改变 CPSR，进入任何一种 ARM 处理器模式。

注意：

用户模式和系统模式是仅有的不可通过异常进入的两种模式，也就是说，要进入这两种模式，必须通过编程改变 CPSR。

表 ARM 处理器异常及其对应模式

异 常	模 式	用 途
快速中断异常	FIQ	进行快速中断请求处理



外部中断请求	IRQ	进行外部中断请求处理
软中断异常	SVC	进行操作系统的高级处理
复位异常	SVC	进行操作系统的高级处理
预取指令中止异常	Abort	虚存和存储器保护
数据中止异常	Abort	虚存和存储器保护
未定义指令异常	Undefined	软件模拟硬件协处理器

6.3 ARM 异常响应和处理程序返回

6.3.1 中断响应的概念

中断的响应过程：当有事件产生，进入中断之前我们必须先记住现在看到书的第几页了，或拿一个书签放在当前页的位置，然后去处理不同的事情（因为处理完了，我们还要回来继续看书），如电话铃响我们要到放电话的地方去，门铃响我们要到门那边去，也就是说不同的中断，我们要在不同的地点处理，而这个地点通常不是固定的。

通常，中断响应大致可以分为以下几个步骤：

- （1）保护断点，即保存下一个将要执行的指令的地址，就是把这个地址送入堆栈；
- （2）寻找中断入口，根据不同的中断源所产生的中断，查找不同的入口地址；
- （3）执行中断处理程序；
- （4）中断返回，执行完中断指令后，就从中断处返回到主程序，继续执行。

6.3.2 ARM 异常响应流程

1. 判断处理器状态

当异常发生时，处理器自动切换到 ARM 状态，所以在异常处理函数中要判断在异常发生前处理器是 ARM 状态还是 Thumb 状态。这可以通过检测 SPSR 的 T 位来判断。

通常情况下，只有在 SWI 处理函数中才需要知道异常发生前处理器的状态。所以在 Thumb 状态下，调用 SWI 软中断异常必须注意以下两点。

- （1）发生异常的指令地址为 (LR-2) 而不是 (LR-4)。
- （2）Thumb 状态下的指令是 16 位的，在判断中断向量号时使用半字加载指令 LDRH。

2. 向量表

如前面介绍向量表时提到的，每一个异常发生时总是从异常向量表开始跳转。最简单的一种情况是向量表里面的每一条指令直接跳向对应的异常处理函数。其中快速中断处理函数 FIQ_Handler() 可以直接从地址 0x1C 处开始，省下一条跳转指令，如图所示。

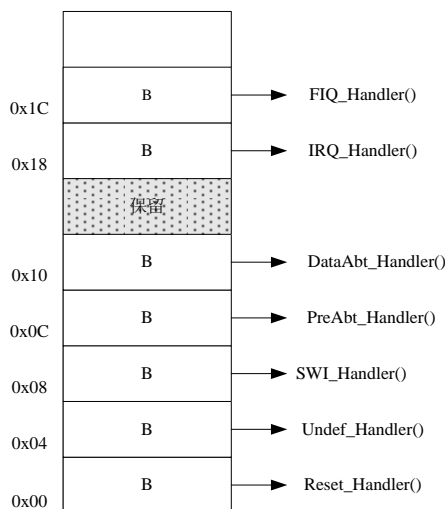


图 异常处理向量表

跳转指令 B 的跳转范围为±32MB,但很多情况下不能保证所有的异常处理函数都定位在向量的 32MB 范围内,而可能需要更大范围的跳转,而且由于向量表空间的限制,只能由一条指令完成。具体实现方法有下面两种。

(1) `MOV PC,#imme_value`。这种办法将目标地址直接赋值给 PC。但这种方法受格式限制不能处理任意立即数。这个立即数由一个 8 位数值循环右移偶数位得到。

(2) `LDR PC,[PC+offset]`。把目标地址先存储在某一个合适的地址空间,然后把这个存储器单元的 32 位数据传送给 PC 来实现跳转。这种方法对目标地址值没有要求,但是存储目标地址的存储器单元必须在当前指令的±4KB 空间范围内。

注意:

在计算指令中引用 offset 数值时,要考虑处理器流水线中指令预取对 PC 值的影响。

6.3.3 从异常处理程序中返回

当一个 ARM 异常处理返回时,一共有 3 件事情需要处理:通用寄存器的恢复、状态寄存器的恢复及 PC 指针的恢复。通用寄存器的恢复采用一般的堆栈操作指令即可,下面重点介绍状态寄存器的恢复及 PC 指针的恢复。

1. 恢复被中断程序的处理器状态

PC 和 CPSR 的恢复可以通过一条指令来实现,下面是 3 个例子。

```
MOVS PC,LR
SUBS PC,LR,#4
LDMFD SP!,{PC}^
```

这几条指令是普通的数据处理指令,特殊之处在于它们把程序计数器寄存器 PC 作为目标寄存器,并且带了特殊的后缀“S”或“^”。其中“S”或“^”的作用就是使指令在执行时,同时完成从 SPSR 到 CPSR 的复制,达到恢复状态寄存器的目的。

2. 异常的返回地址



异常返回时，另一个非常重要的问题就是返回地址的确定。前面提到过，处理器进入异常时会有一个保存 LR 的动作，但是该保持值并不一定是正确中断的返回地址。以一个简单的指令执行流水状态图来对此加以说明，如图所示。

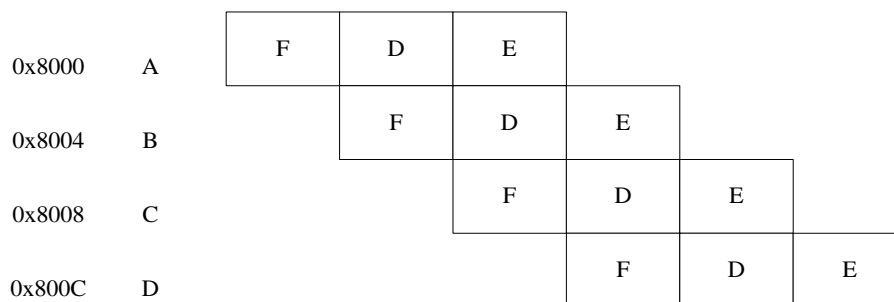


图 3 级流水线示例

在 ARM 指令集中 PC 值为当前执行指令地址加 8，Thumb 指令集中 PC 值为当前执行指令地址加 4，两种情况分析方法一致，后面以 ARM 指令集为例分析。在 ARM 指令集中，当执行指令 A（地址 0x8000）时，PC 等于 $0x8000+8=0x8008$ ，即等于指令 C 的地址。假设指令 A 是 BL 指令，则当执行 BL 指令时，PC 值（0x8008）将保存到 LR 寄存器。但是，接下来处理器会对 LR 进行一次自动调整，即 $LR=LR-0x4$ 。所以，最终保存在 LR 里的是如图 7-3 所示的 B 指令地址。所以当从 BL 返回时，LR 里面正好是正确的返回地址。

同样的调整机制在所有的 LR 自动保存操作中都存在。当进入中断响应时，处理器对保存的 LR 也进行一次自动调整，并且调整动作也是 $LR=LR-0x04$ 。

假设在指令 B 处（0x8004）发生了异常，进入异常响应后，处理器将保存当前 PC 的值到相应模式的 LR 寄存器中，再将 PC 指向对应的异常向量表地址处，从异常模式返回时将 LR 保存的地址返回给 PC 即可，但是实际上返回地址对于不同异常中断是不同的，所以 LR 不一定都是正确的返回地址，也就是说保存的 PC 值会影响返回时的地址，下面详细介绍各种异常中断处理程序的返回方法。

（1）SWI 和未定义指令异常：如果指令 B 为 SWI 指令或者为一条未定义的指令，当执行 B 指令时会产生相应的异常，此时 PC 指向的是 D 指令（0x800C），LR 保存 PC 的值，经过处理器调整后 LR 的值为 C 指令（0x8008）的地址。从 SWI 中断返回后下一条执行指令就是 C，正好是 LR 寄存器保存的地址，所以直接把 LR（0x8008）恢复给 PC 即可。

（2）IRQ 或 FIQ 异常：如果发生的是 IRQ 或 FIQ 异常，处理器会执行完当前的指令 B 然后再处理相应的异常。注意当执行完 B 指令后，流水线的 PC 已经更新指向 0x8010 处，LR 保存 PC 的值，经过处理器调整后 LR 为 0x800C（D 指令地址），但是程序应该返回到 B 指令处（0x8004），所以在返回前要再次对 LR 进行处理，即 $LR=LR-0x04=0x8004$ 。

（3）指令预取中止异常：当预取 B 指令时，若目标地址是非法的，则 B 指令被标记成预取无效的指令，但是处理器依然继续执行 B 之前的指令。当处理器执行被标记为无效的 B 指令时，将产生指令预取中止异常，此时 PC 指向 D 指令（0x800C）。发生指令预取中止异常时，程序应该返回到 B 指令重新读取该



指令，由于 LR 保存 PC (0x800C) 的值，经过处理后 LR 为 0x8008 (C 指令)，所以要想返回到 B 指令则还需要对 LR 进行一次处理即 $LR=LR-0x04=0x8008$ 。

(4) Data Abort 数据中止异常：当 B 指令为数据访问指令，在数据访问时产生异常中断，此时 PC 已经更新指向了 D 指令 (0x8010)，LR 保存的值为 0x8010，经过调整后为 0x800C。产生数据访问中止异常时，程序返回到产生该数据访问中止异常的指令处即 B 指令 (0x8004)，所以在返回时对 LR 进行调整 $LR=LR-0x800C=0x8004$ 。

复位异常中断不需要返回。

表中总结了各类异常和返回地址的关系。

表 异常和返回地址

异 常	返回地址	用 途
复位	—	复位没有定义 LR
数据中止	LR-8	指向导致数据中止异常的指令
FIQ	LR-4	指向发生异常时正在执行的指令
IRQ	LR-4	指向发生异常时正在执行的指令
预取指令中止	LR-4	指向导致预取指令异常的那条指令
SWI	LR	执行 SWI 指令的下一条指令
未定义指令	LR	指向未定义指令的下一条指令

6.4 ARM 的 SWI 异常中断处理程序设计

本节主要介绍编写 SWI 处理程序时需要注意的几个问题，包括判断 SWI 中断号，

使用汇编语言编写 SWI 异常处理函数，使用 C 语言编写 SWI 异常处理函数，在特权模式下使用 SWI 异常中断处理，从应用程序中调用 SWI。

1. 判断 SWI 中断号

当发生 SWI 异常，进入异常处理程序时，异常处理程序必须提取 SWI 中断号，从而得到用户请求的特定 SWI 功能。

在 SWI 指令的编码格式中，后 24 位称为指令的“comment field”。该域保存的 24 位数，即为 SWI 指令的中断号，如图所示。

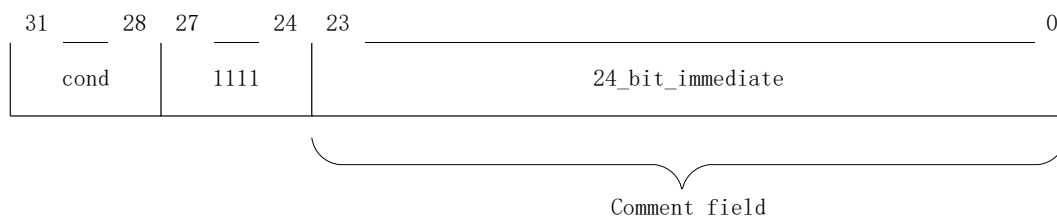


图 SWI 指令编码格式

第一级的 SWI 处理函数通过 LR 寄存器内容得到 SWI 指令地址，并从存储器中得到 SWI 指令编码。



通常这些工作通过汇编语言、内嵌汇编来完成。下面的例子显示了提取中断向量号的标准过程。

.SWI_Handler:

STMFD sp!,{r0-r12,lr} ;保存寄存器

LDR r0,[lr,#-4] ;计算 SWI 指令地址

BIC r0,r0,#0xff000000 ;提取指令编码的后 24 位

;

;提取出的中断号放 r0 寄存器，函数返回

;

LDMFD sp!,{r0-r12,pc}^ ;恢复寄存器

在这个例子中，使用 LR-4 得到 SWI 指令的地址，再通过“BIC r0, r0, #0xff000000”指令提取 SWI 指令中断号。

2. 使用 C 语言编写 SWI 异常处理函数

虽然第一级 SWI 处理函数（完成中断向量号的提取）必须用汇编语言完成，但第二级中断处理函数（根据提取的中断向量号，跳转到具体处理函数）却可以使用 C 语言来完成。

因为第一级的中断处理函数已经将中断号提取到寄存器 r0 中，所以根据 AAPCS 函数调用规则，可以直接使用 BL 指令跳转到 C 语言函数，而且中断向量号作为第一个参数被传递到 C 函数。例如，汇编中使用了“BL C_SWI_Handler”跳转到 C 语言的第二级处理函数，而第二级的 C 语言函数示例如下。

```
void C_SWI_handler (unsigned number)
{
    switch (number)
    {
        case 0 : /* SWI number 0 code */
            break;
        case 1 : /* SWI number 1 code */
            break;
        ...
        default : /* Unknown SWI - report error */
    }
}
```

另外，如果需要传递的参数多于 1 个，那么可以使用堆栈，将堆栈指针作为函数的参数传递给 C 类型的二级中断处理程序，就可以实现在两级中断之间传递多个参数。

例如：

MOV r1, sp ;将传递的第二个参数（堆栈指针）放到 r1 中

BL C_SWI_Handler ;调用 C 函数

相应的 C 函数的入口变为：



```
void C_SWI_handler(unsigned number, unsigned *reg)
```

同时，C 函数也可以通过堆栈返回操作的结果。

3. 从应用程序中调用 SWI

可从汇编语言或 C/C++ 中调用 SWI。

从汇编语言程序中调用 SWI，只要遵循 AAPCS 标准即可。调用前，设定所有必需的值并发出相关的 SWI。例如：

```
MOV r0, #65      ; 将软中断的子功能号放到 r0 中
```

```
SWI 0x0
```

注意：

SWI 指令和其他所有 ARM 指令一样，可以被条件执行。

6.5 FIQ 和 IRQ 中断

6.5.1 中断分支

1. 软件控制中断分支

ARM 内核只有两个外部中断输入信号 nFIQ 和 nIRQ。但对于一个系统来说，中断源可能多达几十个。为此，在系统集成时，一般都会有一个异常控制器来处理异常信号，如图所示。

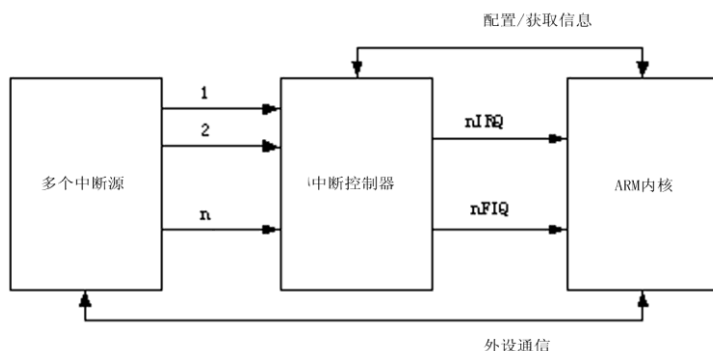


图 中断系统

这时候用户程序可能存在多个 IRQ/FIQ 的中断处理函数。为了使从向量表开始的跳转始终能找到正确的处理函数入口，需要设置处理机制和方法。在以往的 ARM 芯片中采用的是使用软件来处理异常分支，因为软件可以通过读取中断控制器来获得中断源的信息，从而达到中断分支的目的，如图所示。

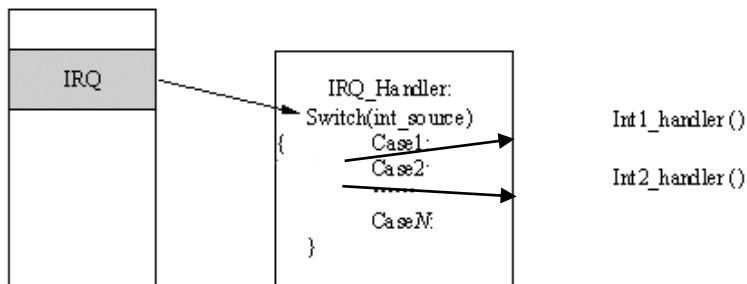


图 软件控制中断分支



因为软件的灵活性，可以设计出比图更好的流程控制方法，如图 7-7 所示。

Int_vector_table 是用户自己开辟的一块存储器空间，里面按次序存放异常处理函数的地址。

IRQ_Handler() 从中断控制器获取中断源信息，然后再从 Int_vector_table 中的对应地址单元得到异常处理函数的入口地址，完成一次异常响应的跳转。这种方法的好处是用户程序在运行过程中，能够很方便地动态改变异常服务内容。

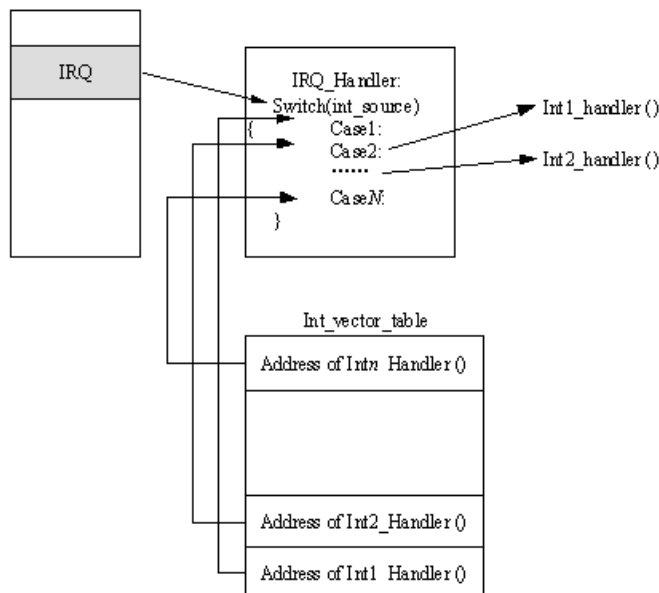


图 灵活的软件控制中断分支设计

进入异常处理程序后,用户可以完全按照自己的意愿来进行程序设计,包括调用 Thumb 状态的函数等。但对于绝大多数的系统来说,有两个步骤必须处理,一是现场保护,二是要把中断控制器中对应的中断状态标识清除,表明该中断请求已经得到响应。否则,中断函数退出以后,又会被再一次触发,从而进入周而复始的死循环。

2. 向量中断控制器

使用向量中断的优点在于,中断优先级仲裁及中断分支的处理递交给了控制器来处理,这样从获取中断源,再到中断 ISR 的处理,其性能相对于软件方式的实现有很大的提高。下面是使用这种机制的详细介绍。

注意: Exynos4412 没有使用向量中断控制器,采用的是软件控制中断分支。

6.6 Exynos4412 中断机制分析

6.6.1 Exynos4412 中断概述

Exynos4412 集成了向量中断控制器(后文用 GIC 来表示),采用的是 ARM 基于 PrimeCell 技术下的 PL390 核心。

Exynos4412 中断控制器支持 160 个中断源,包含: Software Generated Interrupts (SGIs), Private Peripheral Interrupts (PPIs) 和 Shared Peripheral Interrupts (SPIs)。

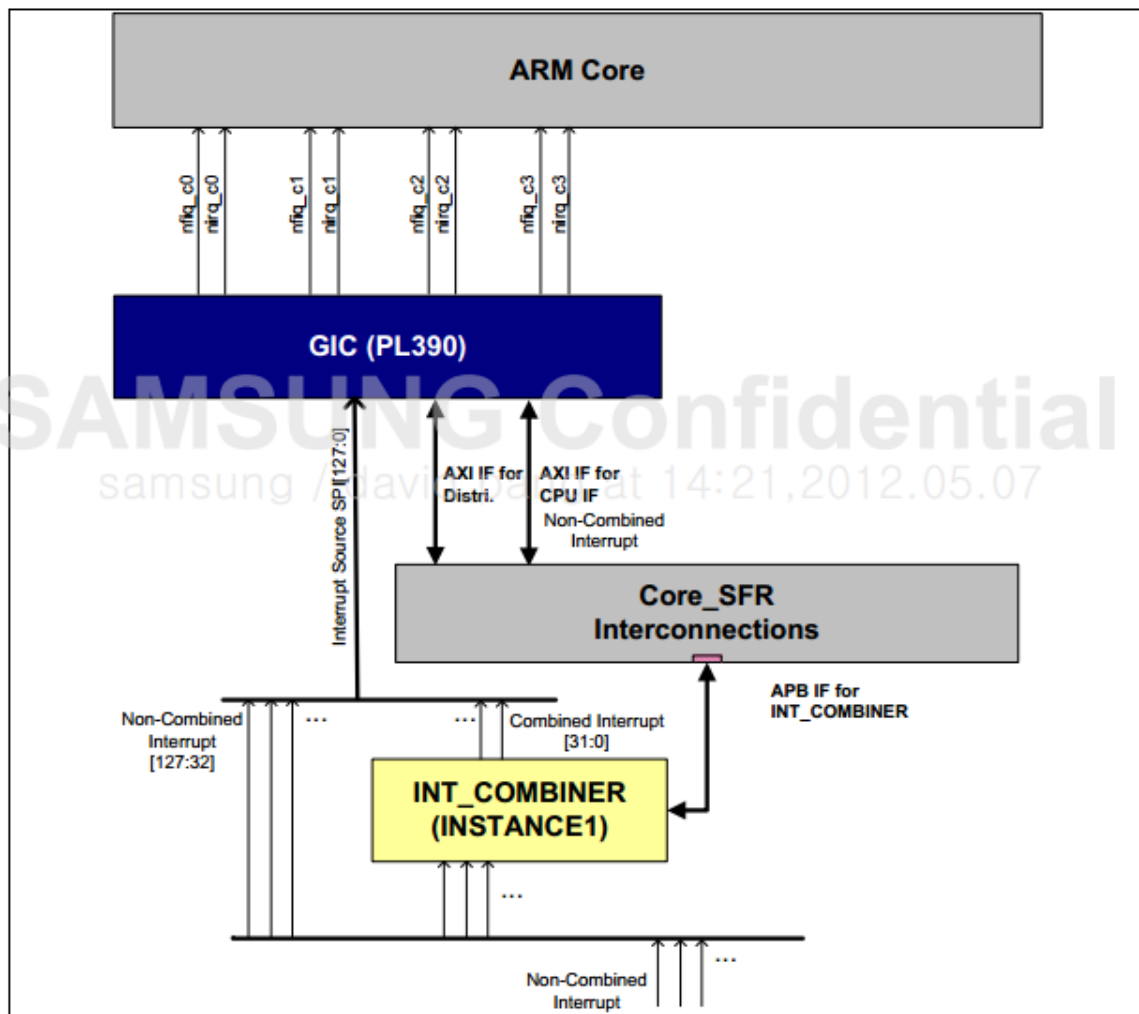


Figure 9-1 Interrupt Sources Connection

6.6.2 EXYNOS4412 中断控制

中断分为中断申请和响应两大部分。下面就以一个外部 IO 中断 **GPX1_1(EINT9)**为例来说明 EXYNOS4412 中断控制。

1. 去除内部上下拉电阻

否则可能会和外部电路的上拉电阻冲突，导致电平不理想

- Base Address: 0x1100_0000
- Address = Base Address + 0x0C28, Reset Value = 0x5555

Name	Bit	Type	Description	Reset Value
GPX1PUD[n]	[2n + 1:2n] N = 0 to 7	RW	0x0 = Disables Pull-up/Pull-down 0x1 = Enables Pull-down 0x2 = Reserved 0x3 = Enables Pull-up	0x5555

2. 配置 GPIO 为中断属性

GPIO 是功能复用的，如果要响应外部中断，需要配置为中断属性。



- Base Address: 0x1100_0000
- Address = Base Address + 0x0C20, Reset Value = 0x0000_0000

GPX1CON[1]	[7:4]	RW	0x0 = Input 0x1 = Output 0x2 = Reserved 0x3 = KP_COL[1] 0x4 = Reserved 0x5 = ALV_DBG[5] 0x6 to 0xE = Reserved 0xF = WAKEUP_INT1[1]	0x00
------------	-------	----	---	------

3. 配置中断触发方式

响应外部中断，需要考虑到信号触发方式。有上升沿、下降沿、高电平、低电平、双沿触发。

EXT_INT41CON 对应 GPX1 的中断配置寄存器。EXYNOS4412 手册描述的不清楚，是通过下面的提示，测试得出的结论。GPX1_1 对应 EXT_INT41[1]。

6.2.3.194	GPX0CON
6.2.3.195	GPX0DAT
6.2.3.196	GPX0PUD
6.2.3.197	GPX0DRV
6.2.3.198	GPX1CON
6.2.3.199	GPX1DAT
6.2.3.200	GPX1PUD
6.2.3.201	GPX1DRV
6.2.3.202	GPX2CON
6.2.3.203	GPX2DAT
6.2.3.204	GPX2PUD
6.2.3.205	GPX2DRV
6.2.3.206	GPX3CON
6.2.3.207	GPX3DAT
6.2.3.208	GPX3PUD
6.2.3.209	GPX3DRV
6.2.3.210	EXT_INT40CON
6.2.3.211	EXT_INT41CON
6.2.3.212	EXT_INT42CON
6.2.3.213	EXT_INT43CON

- Base Address: 0x1100_0000
- Address = Base Address + 0x0E04, Reset Value = 0x0000_0000

EXT_INT41_CON[1]	[6:4]	W	Sets signaling method of EXT_INT41[1] 0x0 = Low level 0x1 = High level 0x2 = Triggers Falling edge 0x3 = Triggers Rising edge 0x4 = Triggers Both edge 0x5 to 0x7 = Reserved	0x0
------------------	-------	---	--	-----

此处设置为下降沿触发。

4. 在 IO 控制器中关闭 GPX1_1 的中断屏蔽，使能中断



6.2.3.223 EXT_INT41_MASK

- Base Address: 0x1100_0000
- Address = Base Address + 0x0F04, Reset Value = 0x0000_00FF

Name	Bit	Type	Description	Reset Value
RSVD	[31:8]	–	Reserved	0x000000
EXT_INT41_MASK[7]	[7]	RW	0x0 = Enables Interrupt 0x1 = Masked	0x1
EXT_INT41_MASK[6]	[6]	RW	0x0 = Enables Interrupt 0x1 = Masked	0x1
EXT_INT41_MASK[5]	[5]	RW	0x0 = Enables Interrupt 0x1 = Masked	0x1
EXT_INT41_MASK[4]	[4]	RW	0x0 = Enables Interrupt 0x1 = Masked	0x1
EXT_INT41_MASK[3]	[3]	RW	0x0 = Enables Interrupt 0x1 = Masked	0x1
EXT_INT41_MASK[2]	[2]	RW	0x0 = Enables Interrupt 0x1 = Masked	0x1
EXT_INT41_MASK[1]	[1]	RW	0x0 = Enables Interrupt 0x1 = Masked	0x1
EXT_INT41_MASK[0]	[0]	RW	0x0 = Enables Interrupt 0x1 = Masked	0x1

设置 EXT_INT41_MASK[1]为 0。

至此，GPIO 相关基础器都已经配置好，中断信号已经可以进入到中断控制器。

5. 配置 GIC 中断控制器 ICDISER 使能中断

从下表中可以看到 GPX1_1(EINT9)属于 SPI 分类，SPI NO25、ID57。

Table 9-2 GIC Interrupt Table (SPI[127:0])

SPI Port No	ID	Int_I_Combiner	Interrupt Source	Source Block
25	57	–	EINT[9]	External Interrupt

寄存器 ICDISER 描述的是和 CPU0、1、2、3 通道相关的中断使能。本例选择的是 CPU0 通道（因为测试程序运行在 CPU0）。中断线是 SPI25。

- Base Address: 0x1049_0000
- Address = Base Address + 0x0100, Reset Value = 0x0000_FFFF (ICDISER0_CPU0)
- Address = Base Address + 0x0104, Reset Value = 0x0000_0000 (ICDISER1_CPU0)
- Address = Base Address + 0x0108, Reset Value = 0x0000_0000 (ICDISER2_CPU0)
- Address = Base Address + 0x010C, Reset Value = 0x0000_0000 (ICDISER3_CPU0)
- Address = Base Address + 0x0110, Reset Value = 0x0000_0000 (ICDISER4_CPU0)
- Address = Base Address + 0x4100, Reset Value = 0x0000_FFFF (ICDISER0_CPU1)
- Address = Base Address + 0x8100, Reset Value = 0x0000_FFFF (ICDISER0_CPU2)
- Address = Base Address + 0xC100, Reset Value = 0x0000_FFFF (ICDISER0_CPU3)

Name	Bit	Type	Description	Reset Value
Set-enable bits	[31:0]	RW	For SPIs and PPIs, for each bit: Reads) 0 = Disables the corresponding interrupt. 1 = Enables the corresponding interrupt. Writes) 0 = No effect. 1 = Enables the corresponding interrupt. A subsequent Read of this bit returns the value 1.	0x0

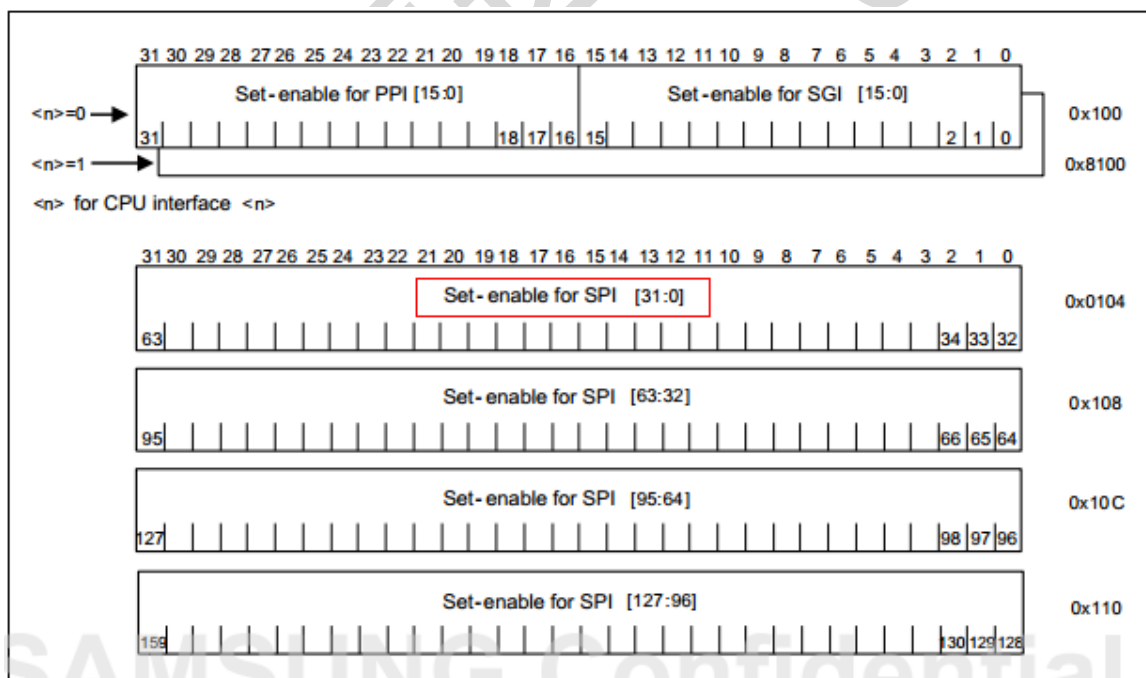


Figure 9-7 ICDISERn Address Map

根据上图所示,编程控制 ICDISER1_CPU0 对应的 25 位。这样中断申请信号 SPI25 可以到达 CPU0 了。接下来需要设置 CPU0 通道,使能 CPU0 通道中断。

6. 配置 GIC 中断控制器使能 CPU0 中断



9.5.1.1 ICCICR_CPU_n

- Base Address: 0x1048_0000
- Address = Base Address + 0x0000, Reset Value = 0x0000_0000 (ICCICR_CPU0)
- Address = Base Address + 0x4000, Reset Value = 0x0000_0000 (ICCICR_CPU1)
- Address = Base Address + 0x8000, Reset Value = 0x0000_0000 (ICCICR_CPU2)
- Address = Base Address + 0xC000, Reset Value = 0x0000_0000 (ICCICR_CPU3)

Name	Bit	Type	Description	Reset Value
RSVD	[31:1]	–	Reserved	0x0
Enable	[0]	RW	Global enable for signaling of interrupts by the CPU Interface to the connected processors. 0 = Disables signaling of interrupts 1 = Enables signaling of interrupts	0x0

7. 配置 CPU0 优先级过滤寄存器

9.5.1.2 ICCPMR_CPU_n

- Base Address: 0x1048_0000
- Address = Base Address + 0x0004, Reset Value = 0x0000_0000 (ICCPMR_CPU0)
- Address = Base Address + 0x4004, Reset Value = 0x0000_0000 (ICCPMR_CPU1)
- Address = Base Address + 0x8004, Reset Value = 0x0000_0000 (ICCPMR_CPU2)
- Address = Base Address + 0xC004, Reset Value = 0x0000_0000 (ICCPMR_CPU3)

Name	Bit	Type	Description	Reset Value
RSVD	[31:8]	–	Reserved	0x0
Priority	[7:0]	RW	The priority mask level for the CPU0 interface When the priority of an interrupt is higher than the value that this field indicates, the interface signals the interrupt to the processor. 256 priority levels support 0x00 – 0xFF (0 to 255), all values	0x0

只有优先级别高于此寄存器的中断，可以发送到 CPU。注意：优先级值越小，级别越高。
本实验设置成 0xff，可以放行所以中断。

中断优先级设置寄存器 ICDIPR_n

ICDIPR14	0x0438	Priority level register (SPI[27:24])	0x0000_0000
----------	--------	--------------------------------------	-------------

Name	Bit	Type	Description	Reset Value
Priority, byte offset 3	[31:24]	RW	Each priority field holds a priority value. Lower the value, greater is the priority of the corresponding interrupt.	0x0
Priority, byte offset 2	[23:16]	RW		0x0
Priority, byte offset 1	[15:8]	RW		0x0
Priority, byte offset 0	[7:0]	RW		0x0

9.5.1.21 ICDIPR_CPU

<n> for CPU interface <n>				
	31	24:23	16:15	8:7
SPI[3:0]	INTID 35	INTID 34	0	INTID 33
SPI[7:4]	INTID 39			INTID 36
SPI[11:8]	INTID 43			INTID 40
.		.		.
SPI[123:120]	INTID 155			INTID 152
SPI[127:124]	INTID 159			INTID 156

本实验没有设置此寄存器。因为所有中断都可通过 CPU0 的优先级限制。

8. GIC 全局中断使能寄存器 ICDDCR 设置

9.5.1.12 ICDDCR

- Base Address: 0x1049_0000
- Address = Base Address + 0x0000, Reset Value = 0x0000_0000

Name	Bit	Type	Description	Reset Value
RSVD	[31:1]	—	Reserved	0x0
Enable	[0]	RW	Global enabled for monitoring peripheral interrupt signals and forwarding pending interrupts to the CPU interfaces. 0 = GIC ignores all peripheral interrupt signals and does not forward pending interrupts to the CPU interfaces. 1 = GIC monitors the peripheral interrupt signals and forwards pending interrupts to the CPU interfaces.	0x0

9. 配置寄存器 ICDIPTR，将中断送到 CPU0 通道

ICDIPTR14	0x0838	Processor targets register (SPI[27:24])	0x0000_0000
-----------	--------	---	-------------

Name	Bit	Type	Description	Reset Value
CPU targets, byte offset 3	[31:24]	RW	Processors in the system number from 0, and each bit in a CPU targets field refers to the corresponding processor. Refer to Table 9-10 for more information.	0x0
CPU targets, byte offset 2	[23:16]	RW		0x0
CPU targets, byte offset 1	[15:8]	RW		0x0
CPU targets, byte offset 0	[7:0]	RW	For example, a value of 0x3 means that the Pending interrupt is sent to processors 0 and 1. For ICDIPTR0 to ICDIPTR7, a Read of any CPU targets field returns the number of the processor that performs the read.	0x0

SPI25 对应的是[15:8]，设置为 1，表示中断送到 CPU0。

10. 配置 ARM 核的 CPSR 寄存器，使能中断

11. 中断响应处理

CPU 在接收到中断信号后，进入异常向量表，然后跳转到中断处理函数处理。中断处理函数中通过对应 CPU 相关的寄存器 ICCIAR，得到中断源 ID 信息，然后处理相关中断。



Name	Bit	Type	Description	Reset Value
RSVD	[31:13]	–	Reserved	0x0
CPUID	[12:10]	R	For SGIs, in a multiprocessor implementation, this field identifies the processor that requests the interrupt. It returns the number of the CPU interface that made the request. For all other interrupts, this field returns as zero.	0x0
ACKINTID	[9:0]	R	The interrupt ID	0x3FF

最后还需要清除中断源的挂起寄存器

- 先清除 GPIO 中断挂起寄存器 EXT_INT41_PEND

6.2.3.227 EXT_INT41_PEND

- Base Address: 0x1100_0000
- Address = Base Address + 0x0F44, Reset Value = 0x0000_0000

Name	Bit	Type	Description	Reset Value
RSVD	[31:8]	–	Reserved	0x000000
EXT_INT41_PEND[7]	[7]	RWX	0x0 = Not occur 0x1 = Interrupt Occurs	0x0
EXT_INT41_PEND[6]	[6]	RWX	0x0 = Not occur 0x1 = Interrupt Occurs	0x0
EXT_INT41_PEND[5]	[5]	RWX	0x0 = Not occur 0x1 = Interrupt Occurs	0x0
EXT_INT41_PEND[4]	[4]	RWX	0x0 = Not occur 0x1 = Interrupt Occurs	0x0
EXT_INT41_PEND[3]	[3]	RWX	0x0 = Not occur 0x1 = Interrupt Occurs	0x0
EXT_INT41_PEND[2]	[2]	RWX	0x0 = Not occur 0x1 = Interrupt Occurs	0x0
EXT_INT41_PEND[1]	[1]	RWX	0x0 = Not occur 0x1 = Interrupt Occurs	0x0
EXT_INT41_PEND[0]	[0]	RWX	0x0 = Not occur 0x1 = Interrupt Occurs	0x0

- 再清除 GIC 中断控制器中挂起位
通过寄存器 ICDICPR 完成



9.5.1.19 ICDICPR_CPU

- Base Address: 0x1049_0000
- Address = Base Address + 0x0280, Reset Value = 0x0000_0000 (ICDICPR0_CPU0)
- Address = Base Address + 0x0284, Reset Value = 0x0000_0000 (ICDICPR1_CPU0)
- Address = Base Address + 0x0288, Reset Value = 0x0000_0000 (ICDICPR2_CPU0)
- Address = Base Address + 0x028C, Reset Value = 0x0000_0000 (ICDICPR3_CPU0)
- Address = Base Address + 0x0290, Reset Value = 0x0000_0000 (ICDICPR4_CPU0)
- Address = Base Address + 0x4280, Reset Value = 0x0000_0000 (ICDICPR0_CPU1)
- Address = Base Address + 0x8280, Reset Value = 0x0000_0000 (ICDICPR0_CPU2)
- Address = Base Address + 0xC280, Reset Value = 0x0000_0000 (ICDICPR0_CPU3)

Name	Bit	Type	Description	Reset Value
Clear-pending bits	[31:0]	RW	<p>For each bit: Reads) 0 = The corresponding interrupt is not pending on any processor. 1 = For SGIs and PPIs, the corresponding interrupt is pending on this processor. For SPIs, the corresponding interrupt is pending on at least one processor.</p> <p>Writes for SPIs and PPIs: 0 = No effect. 1 = The effect depends on whether the interrupt is edge-triggered or level-sensitive:</p> <p>Edge-Triggered Changes the status of the corresponding interrupt to either inactive or active:</p> <ul style="list-style-type: none"> • Inactive: If it was previously pending. • Active: If it was previously active and pending. <p>No effect if the interrupt is not pending.</p> <p>Level Sensitive When the corresponding interrupt is pending, only because of a write to the ICDISPR, the write changes the status of the interrupt to either inactive or active:</p> <ul style="list-style-type: none"> • Inactive: If it was previously pending. • Active: If it was previously active and pending. <p>Otherwise, the interrupt remains pending if the interrupt signal remains asserted.</p> <p>For SGIs, the write is ignored.</p>	0x0

Figure 9-10 illustrates the ICDICPRn address map.

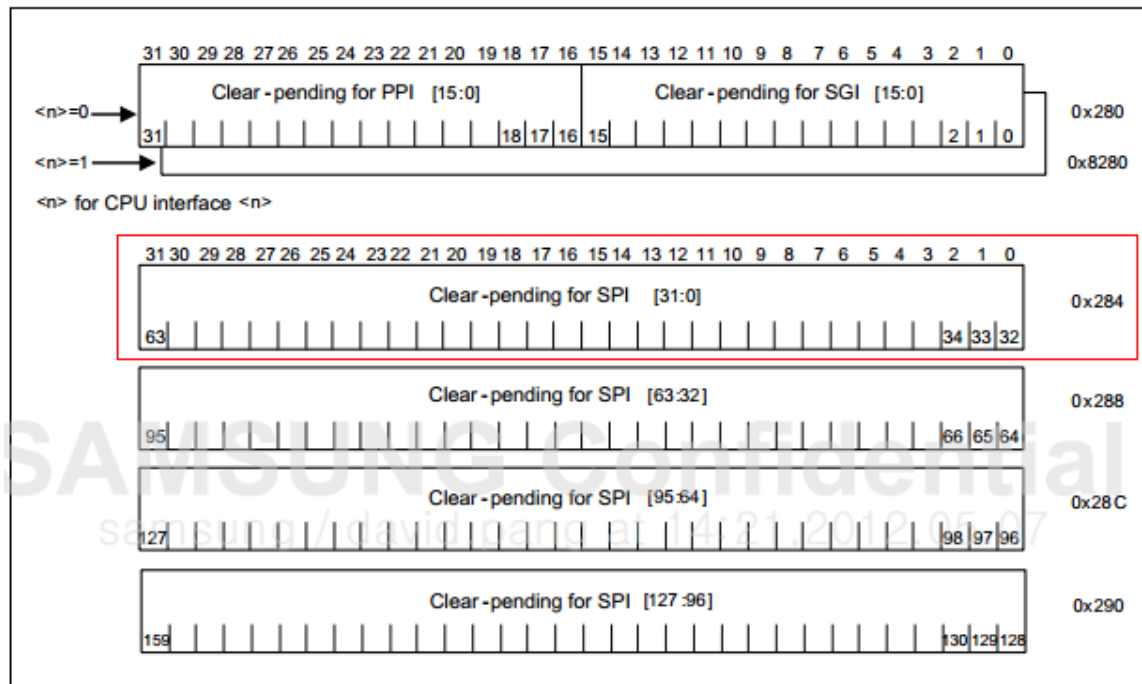


Figure 9-10 ICDICPRn Address Map

- 最后结束对应 cpu 的对应 INTID 的中断。

9.5.1.5 ICCEOIR_CPUn

- Base Address: 0x1048_0000
- Address = Base Address + 0x0010, Reset Value = Undefined (ICCEOIR_CPU0)
- Address = Base Address + 0x4010, Reset Value = Undefined (ICCEOIR_CPU1)
- Address = Base Address + 0x8010, Reset Value = Undefined (ICCEOIR_CPU2)
- Address = Base Address + 0xC010, Reset Value = Undefined (ICCEOIR_CPU3)

Name	Bit	Type	Description	Reset Value
RSVD	[31:13]	—	Reserved	—
CPUID	[12:10]	W	During a multiprocessor implementation, on completion of the processing of an SGI, this field contains the CPUID value from the corresponding ICCIAR access.	—
EOIINTID	[9:0]	W	The ACKINTID value from the corresponding ICCIAR access.	—

6.7 ARM 中断实验

通过第 6 章理论部分的介绍,读者了解了中断的工作原理,以及第 5 章的 Exynos4412 芯片 GPIO 控制器的配置方法。本章通过一个简单示例说明 Exynos4412 的中断处理的应用。

6.7.1 实验目的

示例将利用 Exynos4412 的 K2、K3 这 2 个 I/O 引脚的中断模式,当被按下时进入相应的中断处理函数处理相应的事件。

6.7.2 实验原理

1、电路原理



电路原理如图所示，K2、K3 分别与 GPX1_1、GPX1_2 相连，在没有按下按键时 GPX1_1、GPX1_2 引脚上一直处于高电平，当把这两个引脚设为中断模式并为下降沿中断，则按键被按下俩引脚就会有高电平变为低电平，因此，产生 GPIO 中断进入相应的中断函数，处理中段事件，从终端上打印出相应的按键信息。其中 K2 对应的是 XEINT9 中断源，K3 对应的是 XEINT10 中断源。

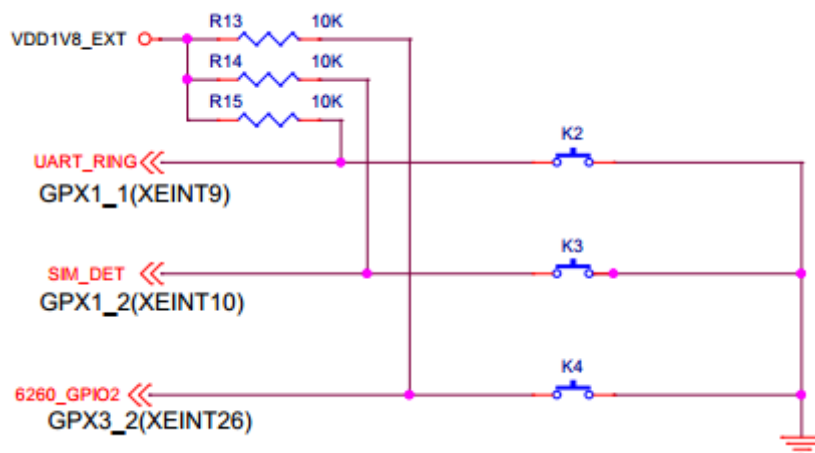


图 EXYNOS4412 中断实验电路图

2、编程流程

- (1) 设置 GPX1_2、GPX1_2 两个管脚没有内部上下拉属性，然后配置为中断模式；
- (2) 设置中断触发方式；
- (3) GPIO 控制器中关闭屏蔽、使能中断；
- (4) 在 GIC 中断控制器中使能中断；
- (5) 设置中断优先级；
- (6) 使能 GIC；
- (7) 选择中断发送给 CPU0；
- (8) 等待中断产生，然后进入中断处理器函数；
- (9) 清楚中断源的挂起状态。

6.7.3 实验内容

1、寄存器设置

为了实现进入中断目的，需要通过配置 GPX1CON 寄存器将 GPX1_1、GPX1_2 设置为中断模式。设置中断方式、中断处理函数、使能中断。

2、程序编写

相关代码如下：

- (1) 设置异常向量表



注意：C12 寄存器用于设定向量表的基地址，这个地址和 map.lds 中的连接地址一致，是 0x40008000。读者理解后，可以更改测试。

```

1      .text
2      .global _start
3      _start:
4          b      reset
5          ldr     pc,_undefined_instruction
6          ldr     pc,_software_interrupt
7          ldr     pc,_prefetch_abort
8          ldr     pc,_data_abort
9          ldr     pc,_not_used
10         ldr     pc,_irq
11         ldr     pc,_fiq
12         _undefined_instruction: .word  _undefined_instruction
13         _software_interrupt:     .word  _software_interrupt
14         _prefetch_abort:        .word  _prefetch_abort
15         _data_abort:            .word  _data_abort
16         _not_used:              .word  _not_used
17         _irq:                   .word  irq_handler
18         _fiq:                   .word  _fiq
19
20     reset:
21
22         ldr r0,=0x40008000
23         mcr p15,0,r0,c12,c0,0    @ Vector Base Address Register
24         .....
25

```

(2) 编写中断处理函数

汇编中的处理

```

1      /**** irq_handler ****/
2      irq_handler:
3
4          sub    lr,lr,#4
5          stmfd  sp!,{r0-r12,lr}
6          bl     do_irq    //跳转到 do_irq 处理器函数
7          ldmdf  sp!,{r0-r12,pc}^

```

主程序中的中断初始化

```

1      /*-----MAIN FUNCTION-----*/

```



```

2  /*****
3  * @brief      Main program body
4  * @param[in]  None
5  * @return     int
6  *****/
7  int main(void)
8  {
9      //LED2 GPX2_7
10     GPX2.GPX2CON |= 0x1 << 28;
11     //LED3 GPX1_0
12     GPX1.GPX1CON |= 0x1;
13     //Led4 GPF3_4
14     GPF3.GPF3CON |= 0x1 << 16;
15
16     //Key_2 Interrupt GPX1_1
17     GPX1.GPX1PUD = GPX1.GPX1PUD & ~(0x3 << 2); // Disables Pull-up/Pull-down
18     GPX1.GPX1CON = (GPX1.GPX1CON & ~(0xF << 4)) | (0xF << 4); //GPX1_1: WAKEUP_INT1[1](EXT_INT41[1])
19     EXT_INT41_CON = (EXT_INT41_CON & ~(0x7 << 4)) | 0x2 << 4;
20     EXT_INT41_MASK = (EXT_INT41_MASK & ~(0x1 << 1)); // Bit: 1 = Enables interrupt
21
22     //Key_3 Interrupt GPX1_2
23     GPX1.GPX1PUD = GPX1.GPX1PUD & ~(0x3 << 4); // Disables Pull-up/Pull-down
24     GPX1.GPX1CON = (GPX1.GPX1CON & ~(0xF << 8)) | (0xF << 8); //GPX1_2:WAKEUP_INT1[2] (EXT_INT41[2])
25     EXT_INT41_CON = (EXT_INT41_CON & ~(0x7 << 8)) | 0x2 << 8;
26     EXT_INT41_MASK = (EXT_INT41_MASK & ~(0x1 << 2)); // Bit: 1 = Enables interrupt
27
28
29     /*
30     * GIC interrupt controller:
31     */
32
33     // Enables the corresponding interrupt SPI25, SPI26 -- Key_2, Key_3
34     ICDISER.ICDISER1 |= (0x1 << 25) | (0x1 << 26);
35
36     CPU0.ICCICR |= 0x1; //Global enable for signaling of interrupts
37
38     CPU0.ICCPMR = 0xFF; //The priority mask level.Priority filter. threshold
39
40     ICDDCR = 1; //Bit1: GIC monitors the peripheral interrupt signals and
41                // forwards pending interrupts to the CPU interfaces2
42
43     ICDIPTR.ICDIPTR14 = 0x01010101; //SPI25 SPI26 interrupts are sent to processor 0
44
45     printf("\n ***** GIC test *****\n");

```



```

46
47     while (1){
48         GPF3.GPF3DAT |= 0x1 << 4;
49         mydelay_ms(500);
50         GPF3.GPF3DAT &= ~(0x1 << 4);
51         mydelay_ms(500);
52     }
53
54     return 0;
55 }
    
```

中断处理函数

```

1      /*****
2      * @brief      IRQ Interrupt Service Routine program body
3      * @param[in]  None
4      * @return     None
5      *****/
6  void do_irq(void )
7  {
8      int irq_num;
9      irq_num = (CPU0.ICCIAR & 0x1FF);
10     switch (irq_num) {
11
12     case 58: //turn on LED2; turn off LED3
13         GPX2.GPX2DAT = 0x1 << 7;
14         GPX1.GPX1DAT &= ~0x1;
15         printf("IRQ interrupt !! turn on LED2; turn off LED3\n");
16
17         //Clear Pend
18         EXT_INT41_PEND |= 0x1 << 2;
19         ICDICPR.ICDICPR1 |= 0x1 << 26;
20         break;
21     case 57: //Turn on Led3; Turn off Led2
22
23         GPX2.GPX2DAT &= ~(0x1 << 7);
24         GPX1.GPX1DAT |= 0x1;
25         printf("IRQ interrupt !! Turn on LED3; Turn off LED2\n");
26
27         //Clear Pend
28         EXT_INT41_PEND |= 0x1 << 1;
29         ICDICPR.ICDICPR1 |= 0x1 << 26;
30         break;
31     }
    
```



```
32 // End of interrupt
33 CPU0.ICCE0IR = (CPU0.ICCE0IR & ~(0x1FF)) | irq_num;
34 }
35
```

6.7.4 实验步骤

1、 导入工程源码

请参考第 1 章节的 ARM 开发环境搭建部分。

光盘实验源码路径：【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\05-Ket_Int】

2、 连接好开发板及 FS-JTAG 仿真器，并且连接好配线。

方法请参考第 1 章节的开发环境搭建的硬件连接部分按对应型号连接。

3、 配置好串口终端

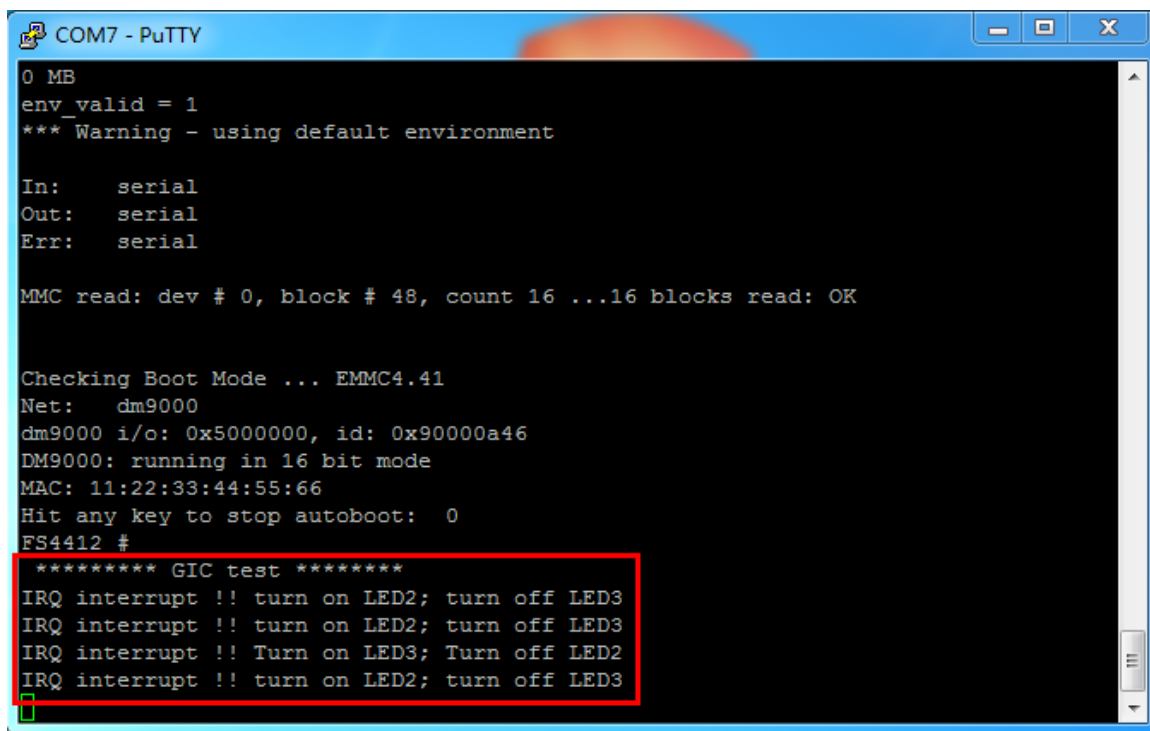
用于显示按键的值。方法请参考第 1 章节的 ARM 开发环境搭建部分。

4、 仿真运行程序

方法请参考第 1 章节的 ARM 开发环境搭建部分。

6.7.5 实验现象

当按下 Key2 或 Key3 的时候观察 LED2 和 LED3 的亮灭情况，通过串口中断也可以看到对应的打印信息。



```
COM7 - PuTTY
0 MB
env_valid = 1
*** Warning - using default environment

In:    serial
Out:    serial
Err:    serial

MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK

Checking Boot Mode ... EMMC4.41
Net:    dm9000
dm9000 i/o: 0x50000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
Hit any key to stop autoboot:  0
FS4412 #
***** GIC test *****
IRQ interrupt !! turn on LED2; turn off LED3
IRQ interrupt !! turn on LED2; turn off LED3
IRQ interrupt !! Turn on LED3; Turn off LED2
IRQ interrupt !! turn on LED2; turn off LED3
```

图 中断处理

6.8 本章小结

本章讲解了 ARM 处理器的异常原理，以及各种异常的工作模式，另外还有 EXYNOS4412 的向量中断机制及编程方式。读者需要结合实验来加深对异常处理和向量中断的理解。

6.9 练习题

1. 简述 ARM 有几种异常，以及每种异常对应的处理器工作模式。
2. 使用向量中断编写一个平台上其它按键中断的程序，并实现上升沿、下降沿、高电平、低电平等促发方式。



第 7 章 串行通讯接口

串行通信接口广泛地应用于各种控制设备，是计算机、控制主板与其他设备传送信息的一种标准接口。本章主要介绍它的工作原理和编程方法。

主要内容有：

- 串行通信的基本原理。
- EXYNOS4412 异步串行通信。
- 接口电路与程序设计。

7.1 串行通信概述

7.1.1 串行通信与并行通信概念

在微型计算机中，通信（数据交换）有两种方式：串行通信和并行通信。

1. 串行通信

串行通信是指计算机与 I/O 设备之间数据传输的各位是按顺序依次一位接一位进行传送。通常数据在一根数据线或一对差分线上传输。

2. 并行通信

并行通信是指计算机与 I/O 设备之间通过多条传输线交换数据，数据的各位同时进行传送。

二者比较：串行通信通常传输速度慢，但使用的传输设备成本低，可利用现有的通信手段和通信设备，适合于计算机的远程通信；并行通信的速度快，但使用的传输设备成本高，适合于近距离的数据传送。需要注意的是，对于一些差分串行通信总线，如 RS-485、RS-422、USB 等，它们的传输距离远，且抗干扰能力强，速度也比较快。

7.1.2 异步串行方式的特点

所谓异步通信，是指数据传送以字符为单位，字符与字符间的传送是完全异步的，位与位之间的传送基本上是同步的。异步串行通信的特点可以概括为：

- (1) 以字符为单位传送信息。
- (2) 相邻两字符间的间隔是任意长。
- (3) 因为一个字符中的比特位长度有限，所以需要的接收时钟和发送时钟只要相近就可以。
- (4) 异步方式特点就是：字符间异步，字符内部各位同步。

7.1.3 异步串行方式的数据格式

异步串行通信的数据格式如图 8-1 所示，每个字符（每帧信息）由 4 部分组成：

- (1) 1 位起始位，规定为低电平 0。
- (2) 5~8 位数据位，即要传送的有效信息。
- (3) 1 位奇偶校验位。
- (4) 1~2 位停止位，规定为高电平 1。

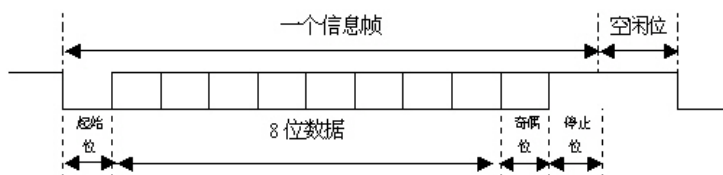


图 异步串行数据格式

7.1.4 同步串行方式的特点

所谓同步通信，是指数据传送是以数据块（一组字符）为单位，字符与字符之间、字符内部的位与位之间都同步。同步串行通信的特点可以概括为：

- （1）以数据块为单位传送信息。
- （2）在一个数据块（信息帧）内，字符与字符间无间隔。
- （3）因为一次传输的数据块中包含的数据较多，所以接收时钟与发送时钟严格同步，通常要有同步时钟。

7.1.5 同步串行方式的数据格式

同步串行通信的数据格式如图所示，每个数据块（信息帧）由 3 部分组成：

- （1）2 个同步字符作为一个数据块（信息帧）的起始标志。
- （2）n 个连续传送的数据。
- （3）2 个字节循环冗余校验码（CRC）。

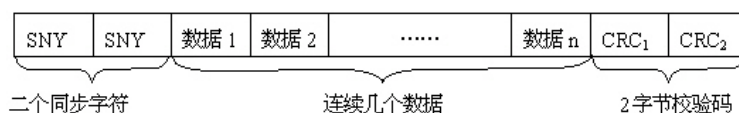


图 同步串行数据格式

7.1.6 比特率、比特率因子与位周期

比特率是指单位时间传输二进制数据的位数，其单位为位/秒（B/S）或比特。它是一个用以衡量数据传送速率的量。一般串行异步通行的传送速度为 50~19200 bit/s，串行同步通信的传送速度可达 500 kbit/s。

比特率因子是指时钟脉冲频率与比特率的比。

位周期 T_d 是指每个数据位传送所需的时间，它与比特率的关系是： $T_d = 1/\text{比特率}$ 。它用以反映连续二次采样数据之间的间隔时间。

7.1.7 RS-232C 串口规范

RS-232C 标准（协议）的全称是 EIA-RS-232C 标准，其中 EIA（Electronic Industry Association）代表美国电子工业协会，RS（Recommended Standard）代表推荐标准，232 是标识号，C 代表 RS232 的最新一次修改（1969），在这之前，有 RS-232B、RS-232A。它规定连接电缆和机械、电气特性、信号功能及传送



过程。常用物理标准还有 EIA-232-C、EIA-422-A、EIA-423A 和 EIA-485。这里只介绍 EIA-232-C(简称 232, RS-232)。例如,目前在 PC 上的 COM1、COM2 接口,就是 RS-232C 接口。

1. 9 针串口引脚定义

PC 串口中的典型是 RS-232 及其兼容接口,串口引脚有 9 针和 25 针两类。而一般的 PC 中使用的都是 9 针的接口,25 针串口具有 20mA 电流环接口功能,用 9、11、18、25 针来实现。这里只介绍 9 针的 RS-232C 串口引脚定义,如表所示。

表 9 针的 RS-232C 串口引脚定义

引脚	简写	功能说明
1	CD	载波侦测
2	RXD	接收数据
3	TXD	发送数据
4	DTR	数据终端设备
5	GND	地线
6	DSR	数据准备好
7	RTS	请求发送
8	CTS	清除发送
9	RI	振铃指示

2. RS-232C 电气特性

EIA-RS-232C 对电气特性、逻辑电平和各种信号线功能都做了明确规定。

在 TXD 和 RXD 引脚上电平定义:

逻辑 1 = -3~-15V

在 RTS、CTS、DSR、DTR 和 DCD 等控制线上电平定义:

信号有效 = +3~+15V

信号无效 = -3~-15V

以上规定说明了 RS-232C 标准对应逻辑电平的定义。注意:对于介于-3~+3V 之间的电压处于模糊区电位,此部分电压将使得计算机无法正确判断输出信号的意义,可能得到 0,也可能得到 1,如此得到的结果是不可信的,在通信时体系会出现大量误码,造成通信失败。因此,实际工作时,应保证传输的电平在+3~+15V 或-3~-15V 之间。

3. RS-232C 的通信距离和速度

RS-232C 规定最大的负载电容为 2500pF,这个电容限制了传输距离和传输速率,由于 RS-232C 的发送器和接收器之间具有公共信号地(GND),属于非平衡电压型传输电路,不使用差分信号传输,因此不具备抗共模干扰的能力,共模噪声会耦合到信号中,在不使用调制解调器(MODEM)时,RS-232C 能够



可靠进行数据传输的最大通信距离为 15 米,对于 RS-232C 远程,必须通过调制解调器进行远程通信连接,或改为 RS-485 等差分传输方式。

现在个人计算机提供的串行端口终端的传输速度一般都可以达到 115200bit/s, 甚至更高, 标准串口能够提供的传输速度主要有以下比特率: 1200bit/s、2400bit/s、4800bit/s、9600bit/s、19200bit/s、38400bit/s、57600bit/s、115200bit/s 等, 在仪器仪表或工业控制场合, 9600bit/s 是最常见的传输速率, 在传输距离较近时, 使用最高传输速度也是可以的。传输距离和传输速度的关系成反比, 适当地降低传输速度, 可以延长 RS-232 的传输距离, 提高通信的稳定性。

4. RS-232C 电平转换芯片及电路

RS-232C 规定的逻辑电平与一般微处理器、单片机的逻辑电平是不同的, 例如, RS-232C 的逻辑“1”是以 -3~-15V 来表示的, 而单片机的逻辑“1”是以 5V 表示的, EXYNOS4412 的逻辑“1”是以 3.3V 表示的, 就必须把单片机的电平 (TTL、CMOS 电平) 转变为 RS-232C 电平, 或者把计算机的 RS-232C 电平转换成单片机的 TTL 或 CMOS 电平, 通信时必须对两种电平进行转换。实现电平转换的芯片可以是分立器件, 也可以是专用的 RS-232C 电平转换芯片。下面介绍一种在嵌入式系统中应用比较广泛的 MAX3232 芯片。

如图所示, 主要特点有:

- ✧ 符合所有的 RS-232C 规范。
- ✧ 单一供电电压+5V 或 3.3V。
- ✧ 片内电荷泵, 具有升压。电压极行反转能力, 能够产生+10V 和-10V 电压 V+、V-。
- ✧ 低功耗, 典型供电电流 3mA。
- ✧ 内部集成 2 个 RS-232C 驱动器。
- ✧ 内部集成 2 个 RS-232C 接收器。

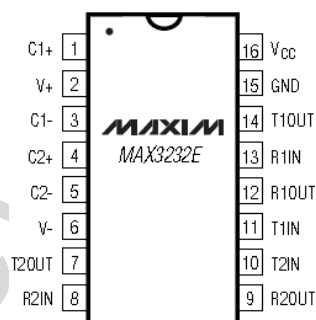


图 MAX3232 芯片

7.1.8 RS-232C 接线方式

RS-232C 串口的接线方式有全串口连接、3 线连接等方式。本书只介绍最简单、常用的 3 线连接方法。PC 和 PC 或处理器之间的通信, 双方都能发送和接收, 它们的连接只需使用 3 根线即可, 即 RXD、TXD 和 GND, 连接方式如图所示。



图 3 线连接法

7.2 EXYNOS4412 异步串行通信

7.2.1 EXYNOS4412 串口控制器概述

1. 简述

EXYNOS4412 的通用异步收发 (UART) 可支持 5 个独立的异步串行输入/输出口，每个口皆可支持中断模式及 DMA 模式，UART 可产生一个中断或者发出一个 DMA 请求，来传送 CPU 与 UART 之间的数据，UART 的比特率最大可达到 4Mbps。每一个 UART 通道包含两 FIFOs 用于数据的收发，其中通道 0 的 FIFO 大小为 256 字节，通道 1、4 的 FIFO 大小为 64 字节，通道 2、3 的 FIFO 大小为 16 字节。

2. 特点

- 5 组收发通道，同时支持中断模式及 DMA 操作。
- 通道 0、1、2 及带红外通道 3。
- 通道 0 带 256byte 的 FIFO，通道 1、4 带 64 byte 的 FIFO，通道 2、3 带 16byteFIFO。
- 通道 0、1、2 支持自动流控功能
- 支持握手模式的发送/接收。

3. 概括图

概括图如图所示：

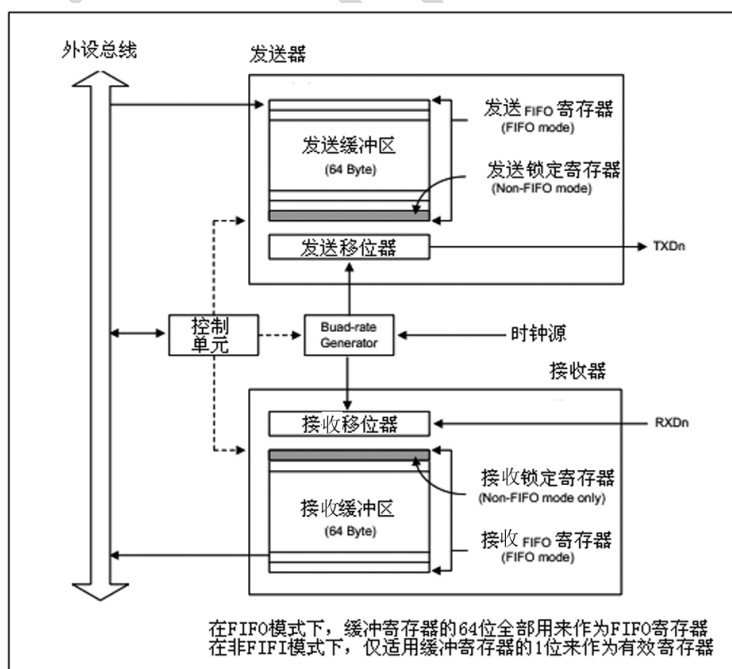




图 概括图

下面简要介绍 UART 操作，关于数据发送，数据接收，中断产生，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，5~8 个数据位，一个可选的奇偶校验位和 1~2 位停止位，停止位通过行控制寄存器 ULCONn 配置。

与发送类似，接收数据帧也是可编程的。接收数据帧由一个起始位，5~8 个数据位，一个可选的奇偶校验和 1~2 位行控制寄存器 ULCONn 里的停止位组成。接收器还可以检测溢出错误、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。

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

(2) 奇偶校验错是指接收器检测到的校验和与设置的不符。

(3) 帧错误指没有接收到有效的停止位。

(4) 传输中断表示接收数据 RxDn 保持逻辑 0 超过一帧的传输时间。

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

7.2.2 UART 寄存器详解

为了让初学者快速掌握串口通信，下面只针对例程中用到的寄存器给予讲解。对于 EXYNOS4412 中提供的更为复杂的控制寄存器将不再展开，感兴趣的读者可作为扩展内容自行学习。

1. UART 行控制寄存器 ULCONn (ULCON0, R/W, Address = 0x1380_0000)

ULCONn 的含义如表所示。

表 ULCONn 的含义

ULCONn	位	描述	初始状态
Reserved	[31:7]	Reserved	0
Infra-Red Mode	[6]	是否使用红外模式 0=正常模式 1=红外模式	0
Parity Mode	[5:3]	校验方式 0XX=无奇偶校验 100=奇校验 101=偶校验 110=校验位强制为 1 111=校验位强制为 0	000
Number of Stop Bit	[2]	停止位数量 0=1 个停止位 1=2 个停止位	0



Word Length	[1:0]	数据位个数 00=5bit 01=6bit 10=7bit 11=8bit	00
-------------	-------	---	----

2. UART 行控制寄存器 UCONn (UCON0,R/W Address = 0x1380_0004)

寄存器详细说明如表所示。

表 UCONn

Name	Bit	Type	Description	Reset Value
RSVD	[31:24]	—	Reserved	0
RSVD	[23]	WO	Reserved	0
Tx DMA Burst Size	[22:20]	RW	<p>Tx DMA Burst Size It is the data transfer size of one DMA transaction. Tx DMA request triggers the DMA transaction. You must program the DMA program to transfer the same data size as this is the value for a single Tx DMA request.</p> <p>000 = 1 byte (Single) 001 = 4 bytes 010 = 8 bytes 011 = 16 bytes 100 = Reserved 101 = Reserved 110 = Reserved 111 = Reserved</p>	0
RSVD	[19]	WO	Reserved	0
Rx DMA Burst Size	[18:16]	RW	<p>Rx DMA Burst Size It is the data transfer size of one DMA transaction. Rx DMA request triggers the DMA transaction. You must program the DMA program to transfer the same data size as this is the value for a single Rx DMA request.</p> <p>000 = 1 byte (Single) 001 = 4 bytes 010 = 8 bytes 011 = 16 bytes 100 = Reserved 101 = Reserved 110 = Reserved 111 = Reserved</p>	0
Rx Timeout Interrupt Interval	[15:12]	RW	<p>Rx Timeout Interrupt Interval Rx interrupt occurs if UART receives no data during $8 \times (N + 1)$ frame time. The default value of this field is 3. It means that the timeout interval is 32 frame time.</p>	0x3
Rx Time-out with empty Rx FIFO ⁽⁴⁾	[11]	R/W	<p>Enables Rx time-out feature when Rx FIFO counter is 0. This bit is valid only when UCONn[7] is 1. 0 = Disables Rx time-out feature when Rx FIFO is empty.</p>	0



Name	Bit	Type	Description	Reset Value
			1 = Enables Rx time-out feature when Rx FIFO is empty.	
Rx Time-out DMA suspend enable	[10]	RW	Enables the suspension of Rx DMA FSM when Rx Time-out occurs. 0 = Disables suspension of Rx DMA FSM 1 = Enables suspension of Rx DMA FSM	0
Tx Interrupt Type	[9]	RW	Interrupt request type. ⁽²⁾ 0 = Pulse (UART requests interrupt when the Tx buffer is empty in the non-FIFO mode or when it reaches the trigger level of Tx FIFO in the FIFO mode.) 1 = Level (Interrupt is requested when Tx buffer is empty in the non-FIFO mode or when it reaches the trigger level of Tx FIFO in the FIFO mode.)	0
Rx Interrupt Type	[8]	RW	Interrupt request type. ⁽²⁾ 0 = Pulse (UART requests interrupt when instant Rx buffer receives data in the non-FIFO mode or when it reaches the trigger level of Rx FIFO in the FIFO mode.) 1 = Level (UART requests interrupt when Rx buffer receives data in the non-FIFO mode or when it reaches the trigger level of Rx FIFO in the FIFO mode.)	0
Rx Time Out Enable	[7]	RW	Enables/disables Rx time-out interrupts when you enable UART FIFO. The interrupt is a receive interrupt. 0 = Disables 1 = Enables	0
Rx Error Status Interrupt Enable	[6]	RW	Enables the UART to generate an interrupt upon an exception, such as a break, frame error, parity error, or overrun error during a receive operation. 0 = Does not generate receive error status interrupt. 1 = Generates receive error status interrupt.	0
Loop-back Mode	[5]	RW	To set this bit to 1 triggers the UART to enter the loop-back mode. This mode is for test purposes only. 0 = Normal operation 1 = Loop-back mode	0
Send Break Signal	[4]	RWX	To set this bit to 1 triggers UART to send a break during 1 frame time. This bit is automatically cleared after sending the break signal. 0 = Normal transmit 1 = Sends the break signal	0
Transmit Mode	[3:2]	RW	Determines which function is able to Write Tx data to the UART transmit buffer. 00 = Disables 01 = Interrupt request or polling mode 10 = DMA mode 11 = Reserved	00
Receive Mode	[1:0]	RW	Determines which function is able to Read data from UART receive buffer. 00 = Disables	00

3. UART FIFO 控制寄存器 UFCONn (UFCON0,R/W,ADDRESS = 0x1380_0008)

寄存器详细说明如表所示。

表 UFCONn 的含义

UFCONn	位	描述	初 始 值
Reserved	[31:11]	Reserved	0
Tx FIFO TriggerLevel	[10:8]	决定发送 FIFO 的触发位置 [Channel 0] 000 = 0 byte 001 = 32 bytes 010 = 64 bytes 011 = 96 bytes 100 = 128 bytes 101 = 160 ytes 110 = 192 bytes 111 = 224 bytes [Channel 1]	000



		000 = 0 byte 001 = 8 bytes 010 = 16 bytes 011 = 24 bytes 100 = 32 bytes 101 = 40 bytes 110 = 48 bytes 111 = 56 bytes [Channel 2, 3] 000 = 0 byte 001 = 2 bytes 010 = 4 bytes 011 = 6 bytes 100 = 8 bytes 101 = 10 bytes 110 = 12 bytes 111 = 14 bytes	
Reserved	[7]	Reserved	0
Rx FIFOTriggerLevel	[6:4]	决定接收 FIFO 的触发位置 [Channel 0] 000 = 32 byte 001 = 64 bytes 010 = 96 bytes 011 = 128 bytes 100 = 160 bytes 101 = 192 bytes 110 = 224 bytes 111 = 256 bytes [Channel 1] 000 = 8 byte 001 = 16 bytes 010 = 24 bytes 011 = 32 bytes 100 = 40 bytes 101 = 48 bytes 110 = 56 bytes 111 = 64 bytes [Channel 2, 3] 000 = 2 byte 001 = 4 bytes 010 = 6 bytes 011 = 8 bytes 100 = 10 bytes 101 = 12 bytes 110 = 14 bytes 111 = 16 bytes	00
Reserved	[3]	保留	0
Tx FIFO Reset	[2]	Tx FIFO 复位后是否清零 0=不清零 1=清零	0
Rx FIFO Reset	[1]	Rx FIFO 复位后是否清零 0=不清零 1=清零	0
FIFO Enable	[0]	使能 FIFO 功能 0=不使能 1=使能	0

4. UART MODEM 控制寄存器 UMCONn (UMCON0,R/W,ADDRESS = :0x1380_004)



寄存器详细说明如表所示。

表 UMCONn 的含义

UMCONn	位	描述	初始值
Reserved	[31:8]	保留	0
RTS trigger Level	[7:5]	<p>如果自动流控制位使能，则以下位将决定失效 nRTS 信号：</p> <p>[Channel 0]</p> <p>000 = 255 bytes 001 = 224 bytes</p> <p>010 = 192 bytes 011 = 160 bytes</p> <p>100 = 128 bytes 101 = 96 bytes</p> <p>110 = 64 bytes 111 = 32 bytes</p> <p>[Channel 1]</p> <p>000 = 63 bytes 001 = 56 bytes</p> <p>010 = 48 bytes 011 = 40 bytes</p> <p>100 = 32 bytes 101 = 24 bytes</p> <p>110 = 16 bytes 111 = 8 bytes</p> <p>[Channel 2]</p> <p>000 = 15 bytes 001 = 14 bytes</p> <p>010 = 12 bytes 011 = 10 bytes</p> <p>100 = 8 bytes 101 = 6 bytes</p> <p>110 = 4 bytes 111 = 2 bytes</p>	000
Auto Flow Control (AFC)	[4]	<p>0: 不允许使用 AFC 模式</p> <p>1: 允许使用 AFC 模式</p>	0
Modem InterruptEnable	[3]	<p>0 = Disables</p> <p>1 = Enables</p>	0
Reserved	[2:1]	保留，必须全为 0	00
Request to Send	[0]	<p>0: 不激活 nRTS</p> <p>1: 激活 nRTS</p>	0

5. 发送寄存器 UTXHn 和接收寄存器 URXHn

这两个寄存器存放着发送和接收的数据，在关闭 FIFO 的情况下只有一个字节 8 位数据。需要注意的是，在发生溢出错误时，接收的数据必须被读出来，否则会引发下次溢出错误。

6. 比特率分频寄存器 UBRDIVn

用于串口比特率的设置。EXYNOS4412 引入了 UDIVSLOTn，使得波特率的设置比早期处理器更加精确。下面以设置波特率为 115200 为目标，介绍设置方法。

$$\text{DIV_VAL} = (\text{PCLK} / (\text{bps} * 16)) - 1$$



$= 40000000 / (115200 * 16) - 1$ //PCLK 由系统时钟提供，此为设定 40MHz

$= 21.7 - 1$

$= 20.7$

UBRDIV_n = 20 (DIV_VAL 的整数部分)。

(UDIVSLOT_n 中 1 的数量)/16 = 0.7。

(UDIVSLOT_n 中 1 的数量) = 11。

所以 UDIVSLOT_n 的值可以为: b1110_1110_1110_1010、b0111_0111_0111_0101 等等其他类似的值, 根据手册中的建议值则可以选择 0xDDD5(1101_1101_1101_0101b)。

7. 串口状态寄存器 UTRSTAT_n (UTRSTAT0,R,ADDRESS = 0x1380_0010)

寄存器详细说明如表所示。

表 UTRSTAT_n 的含义

UTRSTAT _n	位	描述	初 始 值
Reserved	[31:3]	Reserved	0
Transmitter empty	[2]	发送缓冲和发送移位寄存器是否都为空 0=否 1=是	1
Transmit buffer empty	[1]	关闭 FIFO 的情况下, 发送缓冲是否为空 0=不为空 1=空	1
Receive buffer data ready	[0]	关闭 FIFO 的情况下, 接收缓冲是否为空 0=空 1=不为空	0

7.3 串口通信实验

通过第 7 章的介绍, 读者了解了 UART 的功能, 以及 EXYNOS4412 芯片串口控制器的配置方法。本章通过一个简单示例说明 EXYNOS4412 的 UART 接口的应用程序。

7.3.1 实验目的

示例将利用 EXYNOS4412 的复用引脚 XuRXD2、XuTXD2 这 2 个引脚收发串口上的数据, 实现串口调试助手上显示数据。

7.3.2 实验原理

如图所示, COM2 分别与 SP232 的 13、14 引脚相连, 通过 SP3232 的 BUF_XuTXD2/UART_AUDIO_TXD 和 BUF_XuRXD2/UART_AUDIO_RXD 引脚实现 TTI 3.3V 电平转换, 3.3V 电平转换再通过 U8 转变为 1.8V

电平和 CPU 通讯。这样对 EXYNOS4412 的 C8、D8 操作就可以实现在 PC 串口上显示数据。SP3232 起到变压器的作用。

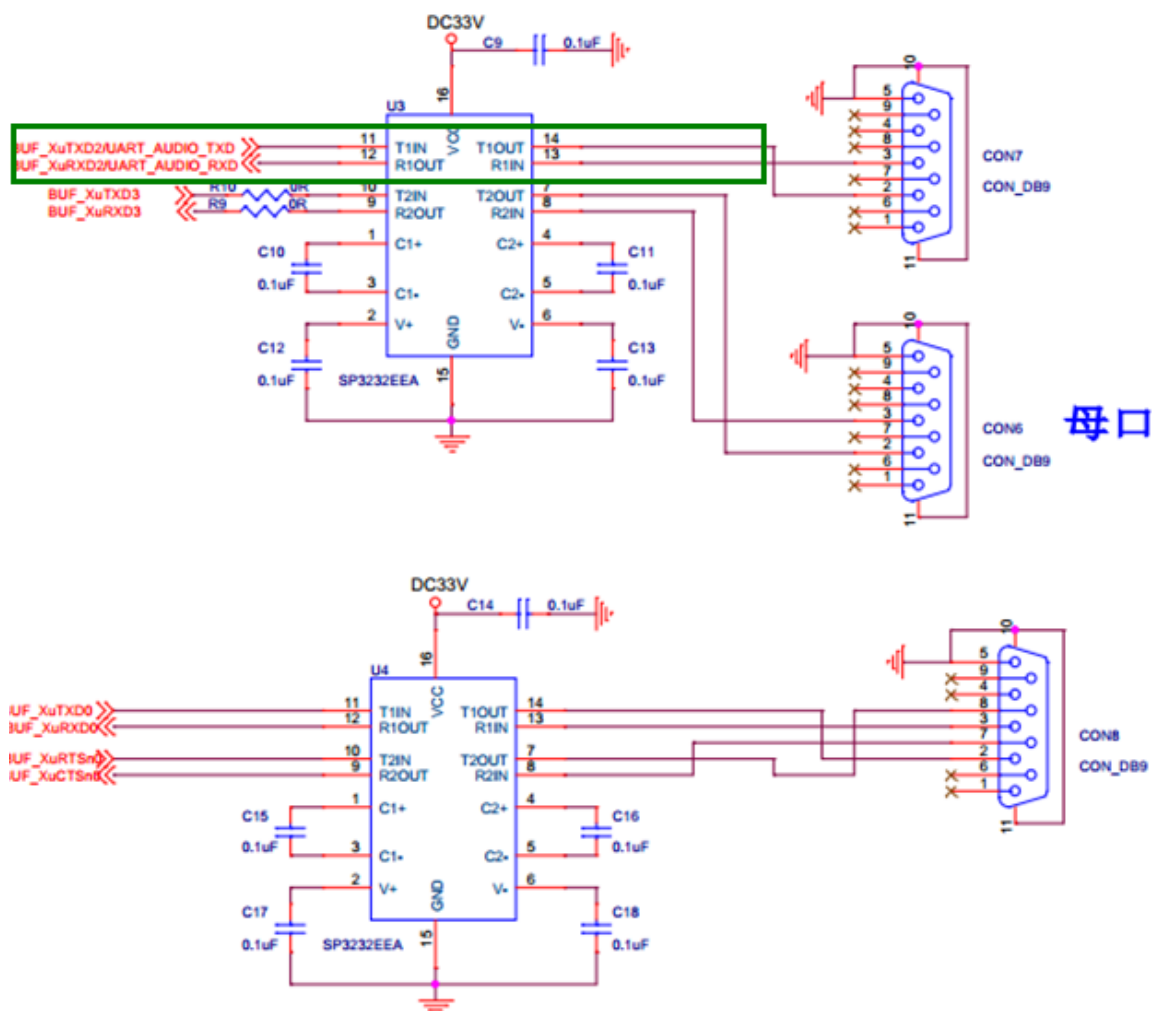


图 COM1 与 SP3232 连接

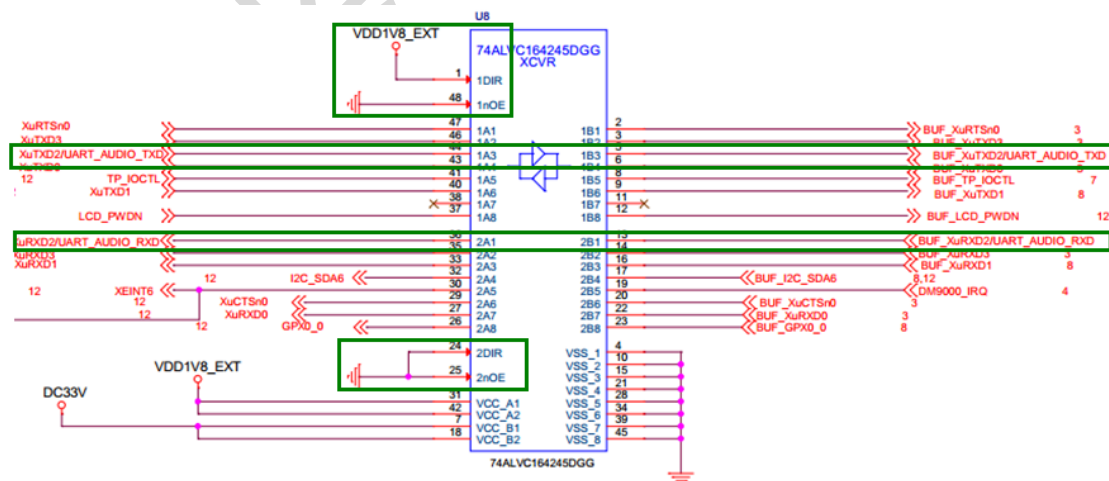




图 SP232 与 EXYNOS4412 相连

PC 端和 EXYNOS4412 要设置相同的串口配置，如：波特率 115200，停止位 1，数据位宽 8 位，无奇偶校验。在 EXYNOS4412 上编程实现串口配置后，向 PC 主机发送一层字符，PC 主机使用串口终端软件显示接收到的字符。

7.3.3 实验内容

1、寄存器设置

为了实现串口调试助手上显示数据，需要通过 GPA1CON 寄存器将 GPA1_0、GPA1_1 配置 UART 属性。设置 UART 串口的属性波特率、停止位、校验位等等。

2、程序编写

串口初始化

```

/*****
 * @brief      uart_init, Normal mode, No parity,One stop bit,8 data bits
 *            Buad-reate : 115200, clock srouce 100Mhz
 * @param[in]  int (ms)
 * @return     None
 *****/
void uart_init(void)
{
    /*UART2 initialize*/
    GPA1.GPA1CON = (GPA1.GPA1CON & ~0xFF) | (0x22); //GPA1_0:RX;GPA1_1:TX

    UART2.ULCON2 = 0x3; //Normal mode, No parity,One stop bit,8 data bits
    UART2.UCON2 = 0x5;  //Interrupt request or polling mode

    /*
     * Baud-rate 115200: src_clock:100Mhz
     * DIV_VAL = (100*10^6 / (115200*16) -1) = (54.3 - 1) = 53.3
     * UBRDIV2 = (Integer part of 53.3) = 53 = 0x35
     * UFRACVAL2 = 0.3*16 = 0x5
     */
    UART2.UBRDIV2 = 0x35;
    UART2.UFRACVAL2 = 0x5;
}

```



串口通讯程序编写

```

1  void putc(const char data)
2  {
3      while(!(UART2.UTRSTAT2 & 0X2));
4      UART2.UTXH2 = data;
5      if (data == '\n')
6          putc('\r');
7  }
8  void puts(const char *pstr)
9  {
10     while(*pstr != '\0')
11         putc(*pstr++);
12 }
13
14 unsigned char getchar()
15 {
16     unsigned char c;
17     while(!(UART2.UTRSTAT2 & 0X1));
18     c = UART2.URXH2;
19     return c;
20 }
21
22 /*-----MAIN FUNCTION-----*/
23 /*****
24  * @brief      Main program body
25  * @param[in]  None
26  * @return     int
27  *****/
28 int main(void) {
29
30     char c, str[] = "uart test!! \n";
31
32     //LED
33     GPX2.GPX2CON = 0x1 << 28;
34     uart_init();
35
36     while(1)
37     {
38         //Turn on LED
39         GPX2.GPX2DAT = GPX2.GPX2DAT | 0x1 << 7;
40         puts(str);
41         mydelay_ms(500);

```



```
42      //Turn off LED
43      GPX2.GPX2DAT = GPX2.GPX2DAT & ~(0x1 << 7);
44      mydelay_ms(500);
45  }
46  return 0;
47 }
48
```

7.3.4 实验步骤

1、 导入工程源码

请参考第 1 章节的 ARM 开发环境搭建部分。

光盘实验源码路径：**【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\10-UART】**

2、 连接好开发板及 FS-JTAG 仿真器，并且连接好配线。

方法请参考第 1 章节的开发环境搭建的硬件连接部分按对应型号连接。

3、 配置好串口终端

方法请参考第 1 章节的 ARM 开发环境搭建部分。

4、 仿真运行程序

方法请参考第 1 章节的 ARM 开发环境搭建部分。可以看到串口调终端上有串口信息。

7.3.5 实验现象

在串口终端上可以看到打印出的信息。



```
COM7 - PuTTY
0 MB
env_valid = 1
*** Warning - using default environment

In:    serial
Out:    serial
Err:    serial

MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK

Checking Boot Mode ... EMMC4.41
Net:    dm9000
dm9000 i/o: 0x50000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
Hit any key to stop autoboot:  0
FS4412 # uart test!!
uart test!!
uart test!!
uart test!!
uart test!!
uart test!!
```

7.4 本章小结

本章重点介绍了串口通信的概念、数据规范、EXYNOS4412 串口控制器及编程方法。串口控制器对于学习来说是一个比较典型的控制器，有 FIFO 单元，支持中断、DMA 控制。如果读者能够掌握它们的控制方法，对于其它控制器的学习会非常有益。

7.5 练习题

1. 串行通信与并行通信的概念是什么？
2. 同步通信与异步通信的概念及区别是什么？
3. RS-232C 串口通信接口规范是什么？
4. 在 EXYNOS4412 串口控制器中，哪个寄存器用来设置串口比特率？
5. 编写一个串口程序采用中断的方式，实现向 PC 的串口终端打印一个字符串“hello”的功能。



第 8 章 PWM 定时器

定时器 / 计数器简称定时器, 其作用主要包括产生各种时标间隔、记录外部事件的数量等, 是微机中最常用、最基本的部件之一。

8.1 PWM 定时器概述

在 EXYNOS4412 中, 一共有 5 个 32 位的定时器, 这些定时器可发送中断信号给 ARM 子系统。另外, 定时器 0、1、2、3 包含了脉冲宽度调制(PWM), 并可驱动其拓展的 I/O。PWM 对定时器 0 有可选的 dead-zone 功能, 以支持大电流设备。要注意的是定时器 4 是内置不接外部引脚的。

定时器 0 与定时器 1 共用一个 8 位预分频器, 定时器 2、定时器 3 与定时器 4 共用另一个 8 位预分频器, 每个定时器都有一个时钟分频器, 时钟分频器有 5 种分频输出 (1/2、1/4、1/8、1/16 和外部时钟 TCLK)。另外, 定时器可选择时钟源, 定时器 0~4 都可选择外部的时钟源, 如 PWM_TCLK。

当时钟被使能后, 定时器计数缓冲寄存器 (TCNTBn) 把计数初值下载到递减计数器中。定时器比较缓冲寄存器 (TCMPBn) 将其初始值下载到比较寄存器中, 并将该值和递减计数器的值进行比较。这种基于 TCNTBn 和 TCMPBn 的双缓冲特性使定时器在频率和占空比变化时能产生稳定的输出。

每个定时器都有一个专用的由定时器时钟驱动的 32 位递减计数器。当递减计数器的计数值达到 0 时, 就会产生定时器中断请求来通知 CPU 定时器操作完成。当定时器递减计数器达到 0 的时候, 相应的 TCNTBn 的值会自动重载到递减计数器中以继续下次操作。然而, 如果定时器停止了, 比如在定时器运行时清除 TCON 中的定时器使能位, TCNTBn 的值不会被重载到递减计数器中。

TCMPBn 的值用于脉冲宽度调制 (PWM)。当定时器的递减计数器的值和比较寄存器的值相匹配的时候, 定时器控制逻辑将改变输出电平。因此, 比较寄存器决定了 PWM 输出的开关时间。

8.2 PWM 定时器特点

PWM 定时器特点如下。

- 5 个 32 位定时器。
- 2 个 8 位 PCLK 分频器提供一级预分, 5 个 2 级分频器用来再预分或预分外部时钟。
- 可编程选择 PWM 独立通道。
- 4 个独立的可编程的控制及支持校验的 PWM 通道。
- 静态配置: PWM 停止。
- 动态配置: PWM 启动。
- 支持自动重装模式及触发脉冲模式。
- 一个外部启动引脚。
- 两个 PWM 输出可带 Dead-Zone 发生器。
- 级中断发生器。

如图所示的死区功能 (Dead Zone) 用于电源设备的 PWM 控制。这个功能允许在一个设备关闭和另一个设备开启之间插入一个时间间隔。这个时间间隔可以防止两个设备同时被启动。TOUT0 是定时器 0 的 PWM 输出, nTOUT0 是 TOUT0 的反转信号。如果死区功能被使能, TOUT0 和 nTOUT0 的输出波形就变成了 TOUT0_DZ 和 nTOUT0_DZ, 如图所示。

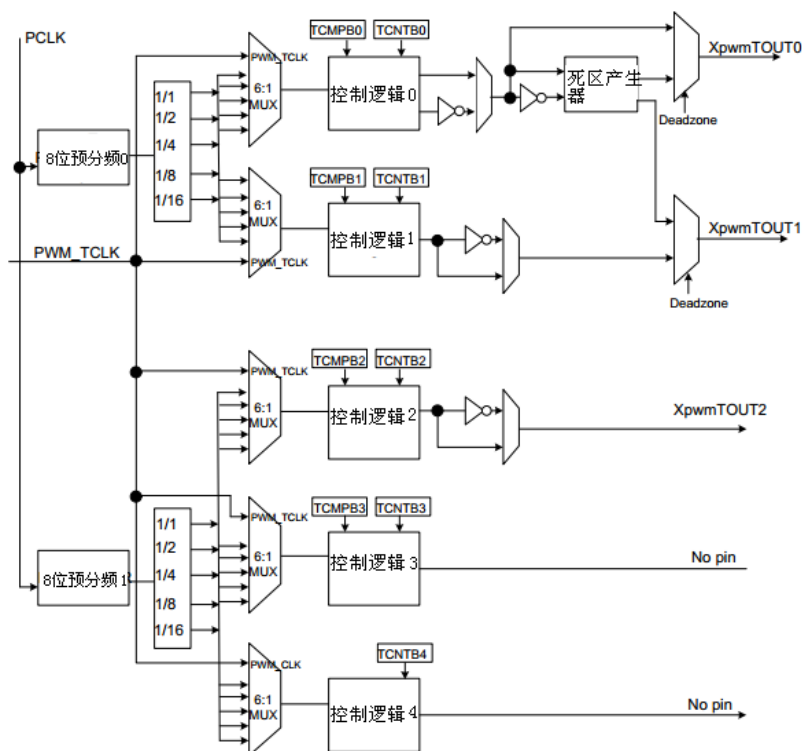


图 EXYNOS4412 PWM 定时器

nTOUT0_DZ 在 TOUT1 脚上产生。在死区间隔内, TOUT0_DZ 和 nTOUT0_DZ 就不会同时翻转了。
注意: 在使能 Dead Zone 时 TOUT1 就是图中的 nTOUT0。

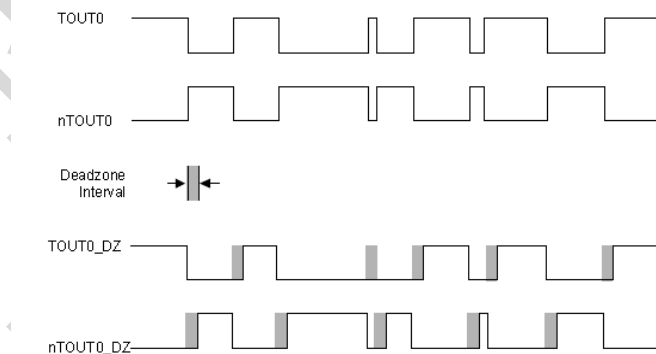


图 死区功能使能时输出的波形

8.3 PWM 定时器的寄存器

EXYNOS4412 控制器共用 18 个 PWM 寄存器。

1. 定时器配置寄存器 0 (TCFG0)

定时器输入时钟频率 = $PCLK / \{prescaler\ value + 1\} / \{divider\ value\}$

```
{ prescaler value } = 1~255;
{ divider value } = 1、2、4、8、16, TCLK
{Dead zone length} = 0~254
```

定时器配置寄存器 0 (TCFG0) 如表所示。

表 TCFG0 寄存器 (0x139D_0000)



TCFG0	位	描述	初始状态
保留	[31:24]	保留	0x00
死区长度	[23:16]	这 8 位决定了死区的长度，一个时间单位和定时器 0 设置的相同	0x00
预分频 1	[15:8]	这 8 位定义了定时器 2、3、4 的预分频值	0x01
预分频 0	[7:0]	这 8 位定义了定时器 0 和 1 的预分频值	0x01

2. 定时器配置寄存器 1 (TCFG1)

定时器配置寄存器 1 主要用于 PWM 定时器的 MUX 输入。

定时器配置寄存器 1 如表所示。

表 寄存器 TCFG1 (0x139D_0004)

TCFG1	位	描述	初始状态
保留	[31:20]	保留	0000
MUX 4	[19:16]	选择定时器 4 的 MUX 输入 0000 = 1/1 0001 = 1/2 0010 = 1/4 0011 = 1/8 0100 = 1/16	0000
MUX 3	[15:12]	选择定时器 3 的 MUX 输入 0000 = 1/1 0001 = 1/2 0010 = 1/4 0011 = 1/8 0100 = 1/16	0000
MUX 2	[11:8]	选择定时器 2 的 MUX 输入 0000 = 1/1 0001 = 1/2 0010 = 1/4 0011 = 1/8 0100 = 1/16	0000
MUX 1	[7:4]	选择定时器 1 的 MUX 输入 0000 = 1/1 0001 = 1/2 0010 = 1/4 0011 = 1/8 0100 = 1/16	0000
MUX 0	[3:0]	选择定时器 0 的 MUX 输入 0000 = 1/1 0001 = 1/2 0010 = 1/4 0011 = 1/8 0100 = 1/16	0000

3. 定时器控制寄存器 (TCON)

定时器控制寄存器主要用于自动重载、定时器自动更新、定时器启停、输出翻转控制等。

定时器控制寄存器如表所示。



表 寄存器 TCON (0x139D_0008)

TCON	位	描述	初始状态
保留	[31:23]	保留	
Timer 4 自动重装	[22]	决定 Timer 4 自动重载功能 0=单发 1=自动重载	0
Timer 4 手动更新	[21]	决定 Timer 4 的手动更新 0=无操作 1=更新 TCNTB4 寄存器	0
Timer 4 开始/停止	[20]	决定 Timer 4 的启停 0=停止 1=定时器 4 开始	0
Timer 3 自动重载	[19]	决定 Timer 3 自动重载功能 0=单发 1=自动重载	0
保留	[18]	保留	
Timer 3 手动更新	[17]	决定 Timer 3 的手动更新 0=无操作 1=更新 TCNTB3 寄存器	0
Timer 3 开/停	[16]	决定 Timer 3 的启停 0=停止 1=定时器 3 开始	0
Timer 2 自动重载	[15]	决定 Timer 2 自动重载功能 0=单发 1=自动重载	0
Timer 2 输出反相	[14]	决定 Timer 2 输出翻转 0=关闭 1=TOUT2 输出翻转	0
Timer 2 手动更新	[13]	决定 Timer 2 的手动更新 0=无操作 1=更新 TCNTB2、TCMPB2 寄存器	0
Timer 2 开/停	[12]	决定 Timer 2 的启停 0=停止 1=定时器 2 开始	0
Timer 1 自动重载	[11]	决定 Timer 1 自动重载功能 0=单发 1=自动重载	0
Timer 1 输出反相	[10]	决定 Timer 1 输出翻转 0=关闭 1=TOUT1 输出翻转	0
Timer 1 手动更新		决定 Timer 1 的手动更新	0



	[9]	0=无操作 1=更新 TCNTB1、TCMPB1 寄存器	
Timer 1 start/stop	[8]	决定 Timer 1 的启停 0=停止 1=定时器 1 开始	0
保留	[7:5]	保留	
Dead zone enable	[4]	死区使能 0=不使能 1=使能	0
Timer 0 auto reload on/off	[3]	决定 Timer 0 自动重载功能 0=单发 1=自动重载	0
Timer 0 output inverter on/off	[2]	决定 Timer 0 输出翻转 0=关闭 1=TOUT0 输出翻转	0
Timer 0 manual update	[1]	决定 Timer 0 的手动更新 0=无操作 1=更新 TCNTB0、TCMPB0 寄存器	0
Timer 0 start/stop	[0]	决定 Timer0 的启停 0=停止 1=定时器 1 开始	0

4. 定时器 n 计数缓冲寄存器 (TCNTBn)

该寄存器用于 PWM 定时器的时间计数。定时器 n 计数缓冲寄存器如表所示。

表 TCNTBn 寄存器

TCNTBn	位	描述	初始状态
Timer n 计数器寄存器	[15:0]	定时器 n (0~4) 计数缓冲寄存器	0x00000000

5. 定时器 n 比较缓冲寄存器 (TCMPBn)

该寄存器用于 PWM 波形输出占空比的设置。定时器 n 比较缓冲寄存器如表所示。

表 TCMPBn 寄存器

TCMPBn	位	描述	初始状态
Timer n 比较缓冲寄存器	[15:0]	定时器 n (0~4) 比较缓冲寄存器	0x00000000

EXYNOS4412 的 PWM 定时器具有双缓冲功能, 如图所示, 能在不停止当前定时器运行的情况下, 重载定时器下次运行的参数。所以尽管新的定时器的值被设置好了, 但是当前操作仍能成功完成。

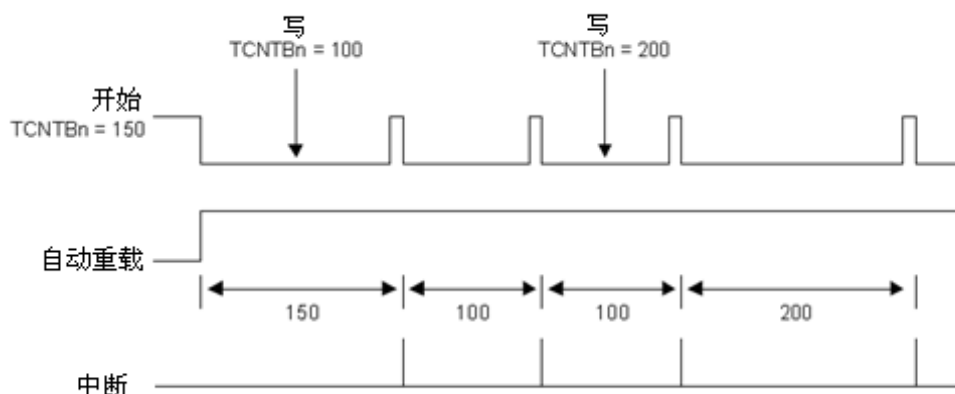


图 双缓冲功能举例

定时器值可以被写入定时器计数 n 缓冲寄存器 (TCNTBn)，当前的计数器的值可以从定时器计数观察寄存器 (TCNTOn) 读出。读出的 TCNTBn 值并不是当前的计数值，而是下次将重载的计数值。

TCNTn 的值等于 0 的时候，自动重载操作把 TCNTBn 的值装入 TCNTn，只有当自动重载功能被使能并且 TCNTn 的值等于 0 的时候才会自动重载。如果 TCNTn 等于 0，自动重载控制位为 0，则定时器停止运行。

使用手动更新位 (manual update) 和反转位 (Inverter) 完成定时器的初始化。当递减计数器的值达到 0 时会发生定时器自动重载操作，所以 TCNTn 的初始值必须由用户提前定义好，在这种情况下就需要通过手动更新位重载初始值。以下几个步骤给出如何启动定时器：

- (1) 向 TCNTBn 和 TCMPBn 写入初始值。
- (2) 置位相应定时器的手动更新位，不管是否使用反转功能，推荐设置反转位。
- (3) 置位相应定时器的启动位启动定时器，清除手动更新位。

如果定时器被强制停止，TCNTn 保持原来的值而不从 TCNTBn 重载值。如果要设置一个新的值，必须执行手动更新操作。

注意：

只要 TOUT 的反转位改变，不管定时器是否处于运行状态，TOUT 都会相应改变，因此通常同时配置手动更新位和反转位。

8.4 PWM 定时器操作示例

操作 PWM 定时器输出如图所示的 PWM 波形。

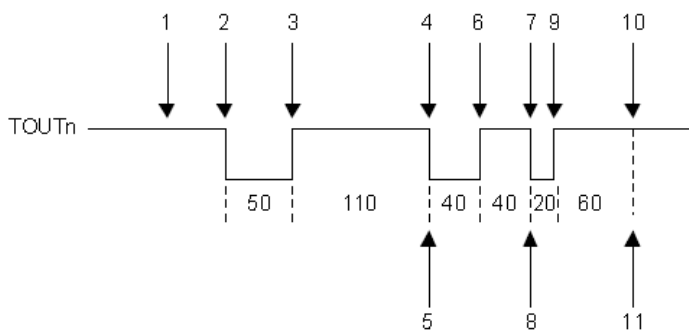


图 定时器操作实例

操作过程（过程号和图中的标号一致）如下：

- （1） 使能自动重载功能，设置 $TCNTB_n$ 值为 160（ $50+110$ ）， $TCMPB_n$ 值为 110。置位手动更新位，配置反转位。置位手动更新位将使 $TCNTB_n$ 和 $TCMPB_n$ 的值加载到 $TCNT_n$ 和 $TCMP_n$ 。然后设置 $TCNTB_n$ 和 $TCMPB_n$ 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 $TCNT_n$ 和 $TCMP_n$ 的值相等的时候， $TOUT$ 输出电平由低变高。
- （4） 当 $TCNT_n$ 的值等于 0 的时候产生中断，并且把 $TCNTB_n$ 和 $TCMPB_n$ 的值分别自动装入 $TCNT_n$ 和 $TCMP_n$ 。
- （5） 在中断服务程序中，将 $TCNTB_n$ 和 $TCMPB_n$ 分别设置为 80（ $20+60$ ）和 60。
- （6） 当 $TCNT_n$ 和 $TCMP_n$ 的值相等的时候， $TOUT$ 输出电平由低变高。
- （7） 当 $TCNT_n$ 等于 0 的时候，把 $TCNTB_n$ 和 $TCMPB_n$ 的值分别自动装入 $TCNT_n$ 和 $TCMP_n$ ，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 $TCNT_n$ 和 $TCMP_n$ 的值相等的时候， $TOUT$ 输出电平由低变高。
- （10） 尽管 $TCNT_n$ 等于 0，但是定时器停止运行，也不再发生自动重载操作，因为定时器自动重载功能被禁止。
- （11） 不再产生新的中断。

通过第 7 章的介绍，读者了解了 UART 的功能，以及 EXYNOS4412 芯片串口控制器的配置方法。本章通过一个简单示例说明 EXYNOS4412 的 UART 接口的应用程序。

8.5 实验 PWM 蜂鸣器实验

8.5.1 实验目的

利用 PWM 定时器实现蜂鸣器控制。



8.5.2 实验原理

蜂鸣器分为“有源”和“无源”两种类型，有源是指其内部自带多谐振荡器等结构，外部只需要提供工作电压，它（内部的振荡器就工作）就能发出固定频率的声音；而无源的是指内部没有带振荡源，需要外部驱动电路提供一定频率的驱动信号，FS4412 开发板所使用蜂鸣器为无源蜂鸣器。

如图所示，定时器 0 的输出引脚 TOUT0 和蜂鸣器的三极管相连，此电路的三极管是 PNP 性，当 TOUT0 是高电平时，此三极管处于饱和状态，电路导通，电流流过蜂鸣器；反之，当 TOUT0 是低电平时，此三极管处于截止状态，电路关断。

应用中常用 PWM 或者直接用 IO 口模拟不同频率方波使无源蜂鸣器发声，所以 MOTOR_PWM 需接入一震荡波形。

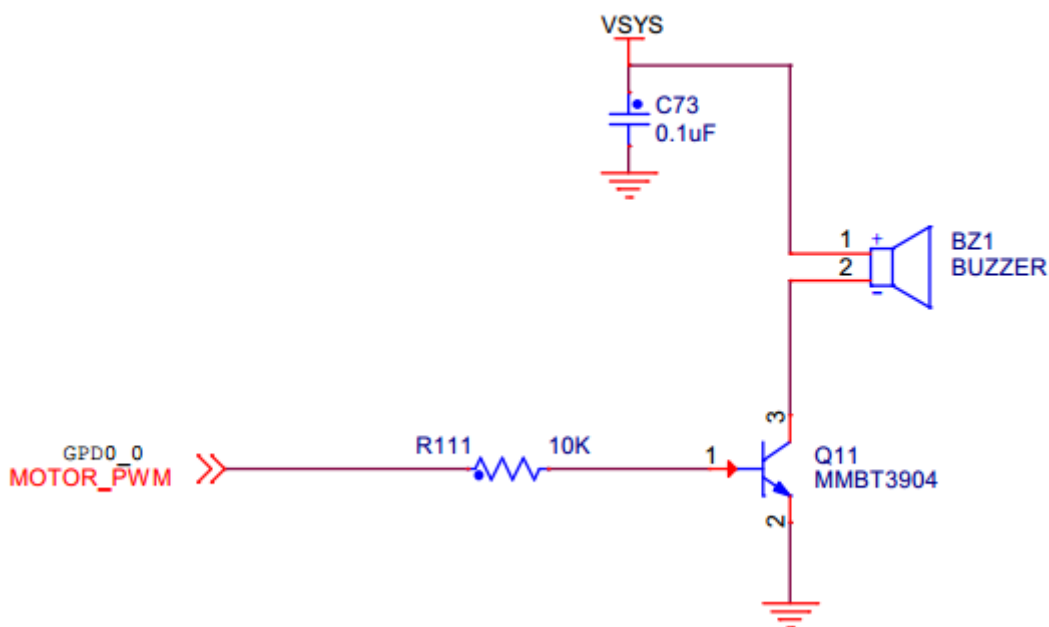


图 蜂鸣器控制电路

8.5.3 实验内容

(1) 寄存器设置

- 1、将 PWMOUT0 对应的引脚配置成 PWM 输出模式
- 2、配置分频值和设置计数缓冲器和比较缓冲器的值
- 3、启动对应的定时器，产生 PWM 波
- 4、不断的改变占空比和 PWM 波的频率可以让蜂鸣器发出不同的声音

(2) 程序编写

相关代码如下：

```
/*
```



```

1  *@brief    This example describes how to use PWM to drive BUZZER
2  *@date:    02. June. 2014
3  *@author   liujh@farsight.com.cn
4  *@Contact  Us: http://dev.hqyj.com
5  *Copyright(C) 2014, Farsight
6  */
7
8  #include "exynos_4412.h"
9
10
11 /*****
12  * @brief      mydelay_ms program body
13  * @param[in]   int (ms)
14  * @return      None
15  *****/
16 void mydelay_ms(int ms)
17 {
18     int i, j;
19     while(ms--)
20     {
21         for (i = 0; i < 5; i++)
22             for (j = 0; j < 514; j++);
23     }
24 }
25
26
27 void PWM_init(void)
28 {
29     GPD0.CON = (GPD0.CON & ~(0xf)) | 0x2; // GPD0_0 : TOUT_0
30     PWM.TCFG0 = (PWM.TCFG0 & ~(0xFF)) | 0x63; //Prescaler 0 value for timer 0; 99 + 1 = 100
31     PWM.TCFG1 = (PWM.TCFG1 & ~(0xF)) | 0x3; // 1/8 input for PWM timer 0
32
33     PWM.TCNTB0 = 200;
34     PWM.TCMPB0 = 100;
35
36     /* auto-reload, Inverter Off, manual update */
37     PWM.TCON = (PWM.TCON & ~(0XF)) | 0XA;
38     /* auto-reload, Inverter Off, manual update off, start Timer0*/
39     PWM.TCON = (PWM.TCON & ~(0xF)) | 0X9;
40
41 }
42 /*-----MAIN FUNCTION-----*/
43 /*****
44  * @brief      Main program body

```



```
45 * @param[in]   None
46 * @return      int
47 *****/
48 int main(void) {
49
50     GPX2.CON = 0x1 << 28;
51     PWM_init();
52
53     while(1)
54     {
55         //Turn on
56         GPX2.DAT = GPX2.DAT | 0x1 << 7;
57         //GPD0.DAT |= 0x1;
58         mydelay_ms(500);
59
60         //Turn off
61         GPX2.DAT = GPX2.DAT & ~(0x1 << 7);
62         // GPD0.DAT &= ~0x1;
63         mydelay_ms(500);
64     }
65     return 0;
66 }
```

8.5.4 实验步骤

1、 导入工程源码

请参考第 1 章节的 ARM 开发环境搭建部分。

光盘实验源码路径：**【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\06-Buzzer_PWM】**

2、 连接好开发板及 FS-JTAG 仿真器，并且连接好配线。

方法请参考第 1 章节的开发环境搭建的硬件连接部分按对应型号连接。

3、 配置好串口终端


方法请参考第 1 章节的 ARM 开发环境搭建部分。

4、 仿真运行程序

方法请参考第 1 章节的 ARM 开发环境搭建部分。



8.5.5 实验现象

Debug 时点击运行按钮，你会听到蜂鸣器鸣叫的声音。

8.6 本章小结

本章重点讲解了 PWM 控制器的工作原理，以及 EXYNOS4412 芯片中 PWM 控制器操作方法。

8.7 练习题

- 1、 PWM 输出波形的特点是什么？
- 2、 编程实现输出占空比为 2：1、波形周期为 9ms 的 PWM 波形。

华清远见
dev.hqyj.com



第 9 章 看门狗定时器

9.1 EXYNOS4412 看门狗定时器概述

看门狗（WatchDog）定时器和 PWM 定时功能目的不一样。它的特点是，需要不停地接受信号（一些外置看门狗芯片）或重新设置计数值（如 EXYNOS4412 的看门狗控制器），保持计数值不为 0。一旦一段时间接收不到信号，或计数值到 0，看门狗将发出复位信号复位系统或产生中断。

看门狗的作用是微控制器受到干扰进入错误状态后，使系统在一定时间间隔内复位。因此看门狗是保证系统长期、可靠和稳定运行的有效措施。目前大部分的嵌入式芯片内都集成了看门狗定时器来提高系统运行的可靠性。

EXYNOS4412 处理器的看门狗是当系统被故障（如噪声或者系统错误）干扰时，用于微处理器的复位操作，也可以作为一个通用的 16 位定时器来请求中断操作。看门狗定时器产生 128 个 PCLK 周期的复位信号。主要特性如下：

- 通用的中断方式的 16 位定时器。
- 当计数器减到 0（发生溢出）时，产生 128 个 PLK 周期的复位信号。

看门狗定时器的功能框图如图所示。

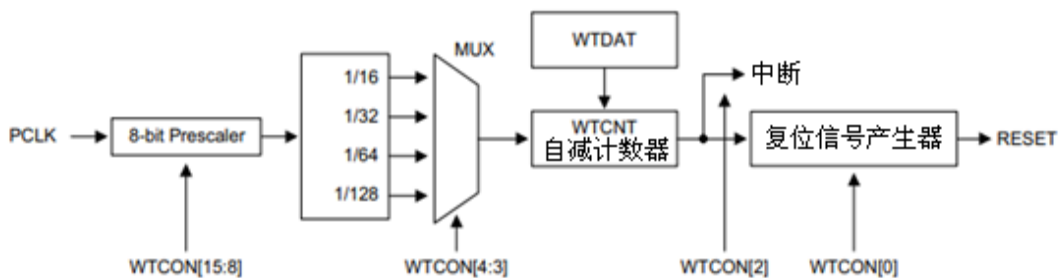


图 EXYNOS4412 的看门狗的功能框图

看门狗模块包括一个预比例因子放大器，一个四分频的分频器，一个 16 位计数器。看门狗的时钟信号来自 PCLK，为了得到宽范围的看门狗信号，PCLK 先被预分频，然后再经过分频器分频。预分频比例因子和分频器的分频值，都可以由看门狗控制寄存器（WTCN）决定，预分频比例因子的范围是 0~255，分频器的分频比可以是 16、32、64 或者 128。看门狗定时器时钟周期的计算如下：

$$t_{\text{watchdog}} = 1 / (\text{PCLK} / (\text{Prescaler value} + 1) / \text{Division_factor})$$

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）的值就不能被自动地装载到看门狗计数器（WTCNT）中。因此，看门狗启动前要将一个初始值写入看门狗计数器（WTCNT）中。当 EXYNOS4412 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。



9.2 看门狗定时器寄存器

1、看门狗定时器控制寄存器（WTCN）

WTCN 寄存器的内容包括：用户是否启用看门狗定时器、4 个分频比的选择、是否允许中断产生、是否允许复位操作等。

如果用户想把看门狗定时器当做一般的定时器使用，应该使能中断，禁止看门狗定时器复位。WTCN 描述如表 10-6 所示。

表 10-6 WTCN 描述

WTCN	位	描述	复位值
保留	[31:16]	保留	0
预分频值	[15:8]	预分频值： 有效数值范围位<0 to 255>	0x80
保留	[7:6]	保留	00
看门狗定时器	[5]	看门狗时钟使能位： 0 = 禁止 1 = 使能	1
始终选择	[4:3]	时钟分频值： 00 = 16 01 = 32 10 = 64 11 = 128	00
中断产生器	[2]	使能/屏蔽中断功能 0 = 禁止 1 = 使能	0
保留	[1]	保留	0
复位使能/屏蔽	[0]	1 = 打开 EXYNOS4412 看门狗产生复位信号 0 = 禁止上述功能	1

2、看门狗定时器数据寄存器（WTDAT）

WTDAT 用于指定超时时间，在初始化看门狗操作后看门狗数据寄存器的值不能被自动装载到看门狗计数寄存器（WTCNT）中。然而，如果初始值为 0x8000，则可以自动装载 WTDAT 的值到 WTCNT 中。WTDAT 描述如表 10-7 所示。

表 10-7 WTDAT 描述

WTDAT	位	描述	复位值
保留	[31:16]	保留	0
计数重载值	[15:0]	看门狗重载数值寄存器	0x8000

3、看门狗计数寄存器（WTCNT）

WTCNT 包含看门狗定时器工作的时候计数器的当前计数值。注意在初始化看门狗操作后，看门狗数据寄存器的值不能被自动装载到看门狗计数寄存器（WTCNT）中，所以看门狗被允许之前应该初始化看



门狗计数寄存器的值。WTCNT 描述如表 10-8 所示。

表 10-8 WTCNT 描述

WTCNT	位	描述	复位值
保留	[31:16]	保留	0
计数值	[15:0]	看门狗当前计数寄存器	0x8000

9.3 看门狗 WDT 实验

9.3.1 实验目的

- 熟悉 EXYNOS4412 中断控制器的使用；
- 理解 EXYNOS4412 处理器的看门狗工作原理；
- 掌握 ARM 处理器的中断方式和看门狗的中断处理方法；

9.3.2 实验原理

根据上面阐述看门狗的工作原理，和对看门狗的寄存器的了解。看门狗是一个内部的系统，不需要外部电路就可以实现对系统状态的监控，系统是否“跑飞”。当系统程序跑分无法“喂狗”时，看门狗控制器可复位系统。

9.3.3 实验内容

3、看门狗软件程序设计流程

由于看门狗是对系统的复位或者中断的操作，所以不需要外围的硬件电路。要实现看门狗的功能，只需要对看门狗的寄存器组进行操作，即对看门狗的控制寄存器（WTCN）、看门狗数据寄存器（WTDAT）、看门狗计数寄存器（WTCNT）进行操作。

其一般流程如下：

- (1) 设置看门狗中断操作，包括全局中断和看门狗中断的使能及看门狗中断向量的定义。如果只是进行复位操作，这一步可以不用设置。
- (2) 对看门狗控制寄存器（WTCN）进行设置，包括设置预分频比例因子、分频器的分频值、中断使能和复位使能等。
- (3) 对看门狗数据寄存器（WTDAT）和看门狗计数寄存器（WTCNT）进行设置。
- (4) 启动看门狗定时器。
- (5) 程序编写

4、看门狗软件程序设计源码

看门狗超时复位测试代码如下：

```
/*
 *@brief This example describes how to use PWM to drive BUZZER
```



```

2  *@date:    02. June. 2014
3  *@author   liujh@farsight.com.cn
4  *@Contact  Us: http://dev.hqyj.com
5  *Copyright(C) 2014, Farsight
6  */
7
8  #include "exynos_4412.h"
9
10
11 /*****
12  * @brief      mydelay_ms program body
13  * @param[in]   int (ms)
14  * @return      None
15  *****/
16 void mydelay_ms(int ms)
17 {
18     int i, j;
19     while(ms--)
20     {
21         for (i = 0; i < 5; i++)
22             for (j = 0; j < 514; j++);
23     }
24 }
25
26
27 void PWM_init(void)
28 {
29     GPD0.CON = (GPD0.CON & ~(0xf)) | 0x2; // GPD0_0 : TOUT_0
30
31     PWM.TCFG0 = (PWM.TCFG0 & ~(0xFF)) | 0x63; //Prescaler 0 value for timer 0; 99 + 1 = 100
32     PWM.TCFG1 = (PWM.TCFG1 & ~(0xF)) | 0x3; // 1/8 input for PWM timer 0
33
34     PWM.TCNTB0 = 200;
35     PWM.TCMPB0 = 100;
36
37     /* auto-reload, Inverter Off, manual update */
38     PWM.TCON = (PWM.TCON & ~(0XF)) | 0XA;
39     /* auto-reload, Inverter Off, manual update off, start Timer0*/
40     PWM.TCON = (PWM.TCON & ~(0xF)) | 0X9;
41
42 }
43
44
45 /*-----MAIN FUNCTION-----*/

```




```

46 /*****
47  * @brief      Main program body
48  * @param[in]   None
49  * @return      int
50  *****/
51 int main(void) {
52
53     GPX2.CON = 0x1 << 28;
54     PWM_init();
55
56     while(1)
57     {
58         //Turn on
59         GPX2.DAT = GPX2.DAT | 0x1 << 7;
60         //GPD0.DAT |= 0x1;
61         mydelay_ms(500);
62
63         //Turn off
64         GPX2.DAT = GPX2.DAT & ~(0x1 << 7);
65         // GPD0.DAT &= ~0x1;
66         mydelay_ms(500);
67     }
68     return 0;
69 }
70
71

```

看门狗中断测试代码

```

1  /*
2  * @brief      This example describes how to use WDT generate interrupt
3  * @date:      12. June. 2014
4  * @author     liujh@farsight.com.cn
5  * @Contact    Us: http://dev.hqyj.com
6  * Copyright(C) 2014, Farsight
7  */
8
9  #include "exynos_4412.h"
10 #include "uart.h"
11

```



```

12 /*****
13  * @brief      IRQ Interrupt Service Routine program body
14  * @param[in]   None
15  * @return      None
16  *****/
17 void do_irq(void)
18 {
19
20     int irq_num;
21     irq_num = (CPU0.ICCIAR & 0x1FF);
22
23     printf("\n ***** RTC_ALARM interrupt !!*****\n");
24     WDT.WTCLRINT = 1;
25     // End of interrupt
26     CPU0.ICCEOIR = (CPU0.ICCEOIR & ~(0x1FF)) | irq_num;
27
28 }
29
30 /*****
31  * @brief      mydelay_ms program body
32  * @param[in]   int (ms)
33  * @return      None
34  *****/
35 void mydelay_ms(int time)
36 {
37     int i, j;
38     while(time--)
39     {
40         for (i = 0; i < 5; i++)
41             for (j = 0; j < 514; j++);
42     }
43 }
44
45 void wdt_init()
46 {
47     WDT.WTCNT = 0x2014; //initial value
48
49     /*
50     *Prescaler value:255, Enables WDT
51     *Prescaler clock division factor 128
52     *Enables WDT interrupt
53     */
54     WDT.WTCON = 0xff<<8 | 1<<5 | 3<<3 | 1<<2 ;
55 }
    
```



```

56
57
58 /*-----MAIN FUNCTION-----*/
59 /*****
60  * @brief      Main program body
61  * @param[in]  None
62  * @return     int
63  *****/
64 int main(void)
65 {
66     GPX2.CON = 0x1 << 28;
67     uart_init();
68
69     /*
70      * GIC interrupt controller:
71      */
72     // Enables the corresponding interrupt SPI43, WDT
73     ICDISER.ICDISER2 |= 1<<11; //ICDISER2:spi 32[bit0] ~ 63[bit31], 43 - 32 = [bit11]
74
75     CPU0.ICCICR |= 0x1; //Global enable for signaling of interrupts
76     CPU0.ICCPMR = 0xFF; //The priority mask level.Priority filter. threshold
77
78     ICDDCR = 1; //Bit1: GIC monitors the peripheral interrupt signals and
79                // forwards pending interrupts to the CPU interfaces2
80
81     //ICDIPTR18:SPI40~SPI43; SPI43 interrupts are sent to processor 0
82     ICDIPTR.ICDIPTR18 = (ICDIPTR.ICDIPTR18 & ~(0xFF<<24)) | 1<<24;
83
84     wdt_init();
85
86     printf("\n*****WDT Interrupt test!!*****\n");
87
88     while(1)
89     {
90         //Turn on LED2
91         GPX2.DAT = GPX2.DAT | 0x1 << 7;
92         mydelay_ms(200);
93
94     #if 0
95         // Feed Dog
96         WDT.WTCNT = 0x2014;
97     #endif
98
99         printf("working...\n");
100        //Turn off LED2

```



```
100     GPX2.DAT = GPX2.DAT & ~(0x1 << 7);  
101     mydelay_ms(200);  
102 }  
103 return 0;  
104 }  
105  
106
```

9.3.4 实验步骤

1、 导入工程源码

请参考第 1 章节的 ARM 开发环境搭建部分。

光盘实验源码路径：【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\11-WDT_RESET (复位)】

光盘实验源码路径：【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\12-WDT_INT (中断)】

2、 连接好开发板及 FS-JTAG 仿真器，并且连接好配线。

方法请参考第 1 章节的开发环境搭建的硬件连接部分按对应型号连接。

3、 配置好串口终端

方法请参考第 1 章节的 ARM 开发环境搭建部分。

4、 仿真运行程序

方法请参考第 1 章节的 ARM 开发环境搭建部分。

9.3.5 实验现象

(1) 复位方式

实验现象如下图所示：产生复位后，不再打印“working...”，LED 灯也不再闪烁



```
COM7 - PuTTY
FS4412 #
***** WDT RESET test!! *****
working...
working...
working...
working...
working...
working...
working...
OK

U-Boot 2010.03 (Jul 14 2014 - 01:41:59) For Farsight FS4412 eMMC

        APLL = 1000MHz, MPLL = 800MHz
        ARM_CLOCK = 1000MHz
PMIC:    S5M8767 (VER5.0)
Board:   FS4412
DRAM:    2 GB
MMC:     max_emmc_clock:40 MHZ
Set CLK to 400 KHz
```

(2) 中断方式

改变第 69 行的代码，将 **#if 0** 改为 **#if 1**

#if 0

```
// Feed Dog
WDT.WTCNT = 0x2014;
```

#endif

改为

#if 1

```
// Feed Dog
WDT.WTCNT = 0x2014;
```

#endif

改完后，编译程序，进入 Debug 模式，运行程序，观察现象。可以发现，由于每次循环都会做一次喂狗的操作，看门狗不再产生复位信号，LED 将一直闪烁下去，串口终端也会一直打印“working...”信息，在打印出第 7 个“working...”后产生了中断。



```
COM7 - PuTTY
Net:  dm9000
dm9000 i/o: 0x5000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
Hit any key to stop autoboot: 0
FS4412 #
*****WDT Interrupt test!!*****
working...
working...
working...
working...
working...
working...
***** RTC_ALARM interrupt !!*****
working...
working...
working...
working...
working...
working...
working...
working...
```

9.4 本章小结

本章重点讲解了看门狗控制器的工作原理，以及 EXYNOS4412 芯片中看门狗控制器的操作方法。

9.5 练习题

- 1、 在控制系统中为何要加入看门狗功能？
- 2、 编程实现 1s 内不对看门狗实现喂狗操作，看门狗会自动复位。

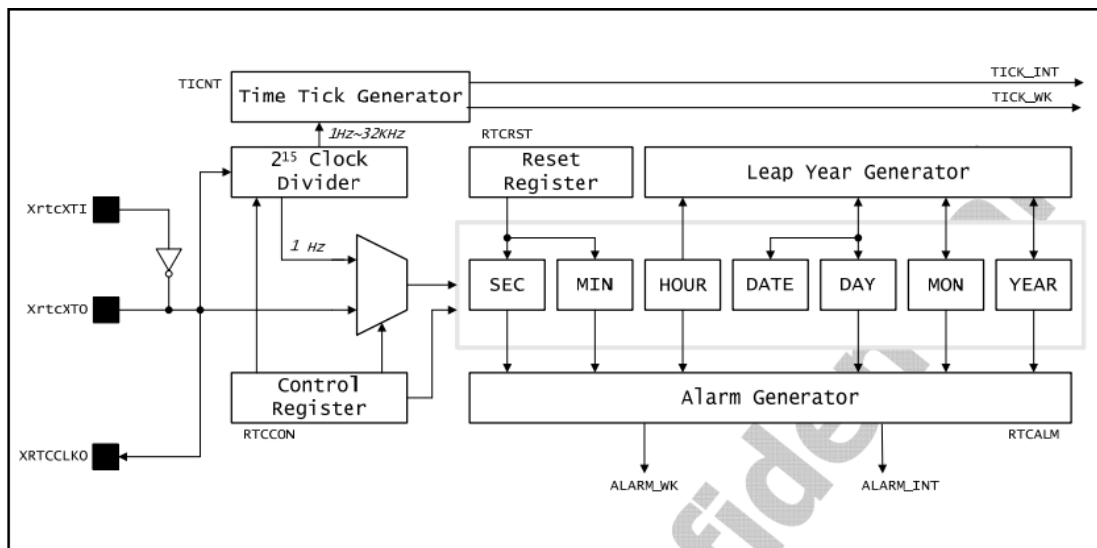


图 RTC 控制器

- 时钟数据采用 BCD 编码。
- 能够对闰年的年月日进行自动处理。
- 具有告警功能，当系统处于关机状态时，能产生告警中断。
- 具有独立的电源输入。
- 提供毫秒级时钟中断，该中断可用于作为嵌入式操作系统的内核时钟。

10.3 RTC 控制器寄存器详解

如表所示为相关寄存器描述。

表 RTC 控制寄存器 (0x1007_0040)

RTCCON	位	描述	复位值
保留	[31:10]	保留	0
CLKOUTEN	[9]	Enables RTC clock output on XRTCCLKO pad 0 = Disables RTC clock output on XRTCCLKO pad 1 = Enables RTC clock output on XRTCCLKO pad	0
TICEN	[8]	嘀嗒计时器 0 = 禁止 1 = 使能	0
TICCKSEL	[7:4]	嘀嗒计时器子时钟源选择 4' b0000 = 32768 Hz 4' b0001 = 16384 Hz 4' b0010 = 8192 Hz 4' b0011 = 4096 Hz 4' b0100 = 2048 Hz 4' b0101 = 1024 Hz 4' b0110 = 512 Hz 4' b0111 = 256 Hz 4' b1000 = 128 Hz 4' b1001 = 64 Hz 4' b1010 = 32 Hz 4' b1011 = 16 Hz 4' b1100 = 8 Hz 4' b1101 = 4 Hz 4' b1110 = 2 Hz 4' b1111 = 1 Hz	4' b0000



CLKRST	[3]	RTC 时钟计数复位 0 = 不复位 1 = 复位	0
CNTSEL	[2]	BCD 计数选择 0 = 分配 BCD 计数 1 = 保留	0
CLKSEL	[1]	BCD 时钟选择 0 = XTAL 1/2 divided clock 1 = 保留 (XTAL 供频)	0
RTCEN	[0]	RTC 控制使能 0 = 禁止 1 = 使能	0

如表所示为 BCD 值寄存器描述，以 SECDATA 为例，其它时间寄存器类似，读者参见 Exynos4412 的 datasheet。

表 BCD 值寄存器

BCDSEC	位	描述	复位值
保留	[31:7]	保留	-
SECDATA	[6:4]	BCD 值 0~5	-
	[3:0]	0~9	-

10.4 实时时钟 RTC 实验

10.4.1 实验目的

- 了解 RTC 的硬件控制原理及设计方法；
- 掌握 EXYNOS4412 处理器的 RTC 模块程序设计方法（计时功能、闹钟功能、时间片功能）；

10.4.2 实验原理

实时时钟（RTC）单元可以在当系统电源关闭后通过备用电池工作。RTC 可以通过使用 STRB/LDRB ARM 操作发送 8 位二进制十进制交换码（BCD）值数据给 CPU。这些数据包括年、月、日、星期、时、分和秒的时间信息。根据上面阐述 RTC 的工作原理和 RTC 的寄存器的介绍。对相应的寄存器读写就可以实现修改时间和现实时间。

10.4.3 实验内容

1、RTC 设计步骤

- 1) 系统复位后在 RTC 控制寄存器中的 CTLEN 必须设置为 1 来使能数据的读/写。
- 2) 设置 RTC 当前时钟时间。
- 3) 同样的在掉电前，RTCEN 位应该清除为 0 来预防误写入 RTC 寄存器中。
- 4) 读取年、月、日等相关寄存器的数据显示到屏幕上。



2、看门软件程序设计

(1) 下面的代码实现了一个 RTC 的年月日、时分秒读出的功能。

```

1  /*****
2  * @brief      RTC_init, second, minute, hour, day, week, month, year
3  * @param[in]  None
4  * @return     None
5  *****/
6  void RTC_init()
7  {
8      RTCCON = 0x1;    // Enables RTC control
9
10     RTC.BCDSEC = 0x11;
11     RTC.BCDMIN = 0x11;
12     RTC.BCDHOUR = 0x11;
13     RTC.BCDDAY = 0x11;
14     RTC.BCDWEEK = 0x11;
15     RTC.BCDMON = 0x11;
16     RTC.BCDYEAR = 0x11;
17
18     RTCCON = 0x0;    // Disables RTC control
19 }
20
21 /*-----MAIN FUNCTION-----*/
22 /*****
23 * @brief      Main program body
24 * @param[in]  None
25 * @return     int
26 *****/
27 int main(void)
28 {
29     GPX2.CON = 0x1 << 28;
30     uart_init();
31
32     RTC_init();
33
34     printf("\n***** RTC ***** \n");
35
36     while(1)
37     {
38         //Turn on
39         GPX2.DAT = GPX2.DAT | 0x1 << 7;
40         mydelay_ms(500);

```



```

41     printf("year 20%x : month %x : date %x :day %x ", RTC.BCDYEAR,\
42           RTC.BCDMON,\
43           RTC.BCDDAY,\
44           RTC.BCDWEEK );
45
46     printf("hour %x : min %x : sec %x\n",RTC.BCDHOUR, RTC.BCDMIN, RTC.BCDSEC);
47     //Turn off
48     GPX2.DAT = GPX2.DAT & ~(0x1 << 7);
49     mydelay_ms(500);
50 }
51 return 0;
52 }
53
54
    
```

(2) 下面的代码实现了一个 20 秒定时功能
 通过 ALARM 寄存器设置定时的规则，20 秒后产生中断。在中断处理函数中打印出中断信息。RTC 的“定时打铃”功能也是很常用的。

```

1  /*
2  *@brief   This example describes how to use RTC's alarm function
3  *@date:    12. June. 2014
4  *@author   liujh@farsight.com.cn
5  *@Contact  Us: http://dev.hqyj.com
6  *Copyright(C) 2014, Farsight
7  */
8  #include "exynos_4412.h"
9  #include "uart.h"
10
11  /*****
12   * @brief      IRQ Interrupt Service Routine program body
13   * @param[in]   None
14   * @return      None
15   *****/
16  void do_irq(void)
17  {
18
19      int irq_num;
20      irq_num = (CPU0.ICCIAR & 0x1FF);
21
    
```



```

22     printf("\n ***** RTC_ALARM interrupt !!*****\n");
23
24     RTCINTP |= 0x2;
25     // End of interrupt
26     CPU0.ICCEOIR = (CPU0.ICCEOIR & ~(0x1FF)) | irq_num;
27
28 }
29
30 /*****
31  * @brief      mydelay_ms program body
32  * @param[in]  int (ms)
33  * @return     None
34  *****/
35 void mydelay_ms(int time)
36 {
37     int i, j;
38     while(time--)
39     {
40         for (i = 0; i < 5; i++)
41             for (j = 0; j < 514; j++);
42     }
43 }
44
45
46 /*****
47  * @brief      RTC_init, second, minute, hour, day, week, month, year
48  * @param[in]  None
49  * @return     None
50  *****/
51 void RTC_init()
52 {
53     RTCCON = 0x1;    // Enables RTC control
54
55     RTC.BCDSEC = 0x11;
56     RTC.BCDMIN = 0x11;
57     RTC.BCDHOUR = 0x11;
58     RTC.BCDDAY = 0x11;
59     RTC.BCDWEEK = 0x11;
60     RTC.BCDMON = 0x11;
61     RTC.BCDYEAR = 0x11;
62
63     RTCCON = 0x0;    // Disables RTC control
64 }
65

```



```

66 /*****
67  * @brief      alarm_init
68  *              Disables year, month, day, hour, minute alarm
69  *              Enable second alarm
70  * @param[in]   None
71  * @return      None
72  *****/
73 void rtc_alarm_init()
74 {
75     RTCALM.ALM = 1; // Disables alarm global, Enables second alarm
76     RTCALM.SEC = 0x20; //BCD value for alarm second, 20''
77     RTCALM.ALM |= 1<<6; // Enables alarm global
78
79 }
80
81
82 /*-----MAIN FUNCTION-----*/
83 /*****
84  * @brief      Main program body
85  * @param[in]   None
86  * @return      int
87  *****/
88 int main(void)
89 {
90     GPX2.CON = 0x1 << 28;
91     uart_init();
92
93     RTC_init();
94
95     /*
96     * GIC interrupt controller:
97     */
98     // Enables the corresponding interrupt SPI44, RTC_ALARM
99     ICDISER.ICDISER2 |= 1<<12; //ICDISER2:spi 32[bit0] ~ 63[bit31], 44 - 32 = [bit12]
100
101     CPU0.ICCICR |= 0x1; //Global enable for signaling of interrupts
102     CPU0.ICCPMR = 0xFF; //The priority mask level.Priority filter. threshold
103
104     ICDDCR = 1; //Bit1: GIC monitors the peripheral interrupt signals and
105                // forwards pending interrupts to the CPU interfaces2
106
107     ICDIPTR.ICDIPTR19 = (ICDIPTR.ICDIPTR19 & ~(0xFF)) | 0x1;
108     //SPI44 interrupts are sent to processor 0
109 
```



```
110     rtc_alarm_init();
111
112     printf("\n***** RTC_ALARM ***** \n");
113
114     while(1)
115     {
116         //Turn on
117         GPX2.DAT = GPX2.DAT | 0x1 << 7;
118         mydelay_ms(500);
119
120         printf("year 20%x : month %x : date %x :day %x ", RTC.BCDYEAR,\
121                                     RTC.BCDMON,\
122                                     RTC.BCDDAY,\
123                                     RTC.BCDWEEK );
124
125         printf("hour %x : min %x : sec %x\n",RTC.BCDHOUR, RTC.BCDMIN, RTC.BCDSEC);
126         //Turn off
127         GPX2.DAT = GPX2.DAT & ~(0x1 << 7);
128         mydelay_ms(500);
129     }
130     return 0;
131 }
132
```

10.4.4 实验步骤

1、 导入工程源码

请参考第 1 章节的 ARM 开发环境搭建部分。

(1) RTC 时间读取实验

光盘实验源码路径：**【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\07-RTC】**

(2) RTC 定时中断实验

光盘实验源码路径：**【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\08-Alarm_RTC】**

2、 连接好开发板及 FS-JTAG 仿真器，并且连接好配线。

方法请参考第 1 章节的开发环境搭建的硬件连接部分按对应型号连接。

3、 配置好串口终端

方法请参考第 1 章节的 ARM 开发环境搭建部分。



4、仿真运行程序

方法请参考第 1 章节的 ARM 开发环境搭建部分。

10.4.5 实验现象

1、 RTC 时间读取实验

通过串口中断观察打印信息，实验结果如下图所示。

```
COM7 - PuTTY
*** Warning - using default environment

In:    serial
Out:    serial
Err:    serial

MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK

Checking Boot Mode ... EMMC4.41
Net:    dm9000
dm9000 i/o: 0x5000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
Hit any key to stop autoboot:  0
FS4412 #
***** RTC *****
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 11
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 12
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 13
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 14
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 15
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 16
```

2、 RTC 定时中断实验

通过串口中断观察打印信息，可以发现当秒达到 20 的时候打印出了中断信息。



```
COM7 - PuTTY
MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK

Checking Boot Mode ... EMMC4.41
Net: dm9000
dm9000 i/o: 0x50000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
Hit any key to stop autoboot: 0
FS4412 #
***** RTC_ALARM *****
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 11
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 13
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 14
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 15
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 16
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 17
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 18
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 19
***** RTC_ALARM interrupt !!*****
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 20
year 2011 : month 11 : date 1 :day 11 hour 11 : min 11 : sec 21
```

10.5 练习题

- 1、编程实现 RTC 的定时打开蜂鸣器功能。



第 11 章 A/D 转换器

A/D 转换又称模数转换,顾名思义,就是把模拟信号数字化。实现该功能的电子器件称为 A/D 转换器, A/D 转换器可将输入的模拟电压转换为与其成比例输出的数字信号。随着数字技术,特别是计算机技术的飞速发展与普及,在现代控制、通信及检测领域中,对信号的处理广泛采用了数字计算机技术。由于系统的实际处理对象往往都是一些模拟量(如温度、压力、位移、图像等),要使计算机或数字仪表能识别和处理这些信号,必须首先将这些模拟信号转换成数字信号,这就必须用到 A/D 转换器。

本章主要内容:

- A/D 转换器原理。
- EXYNOS4412 A/D 转换器。
- A/D 转换器应用举例。

11.1 A/D 转换器原理

11.1.1 A/D 转换基础

在基于 ARM 的嵌入式系统设计中, A/D 转换接口电路是应用系统前向通道的一个重要环节,可完成一个或多个模拟信号到数字信号的转换。模拟信号到数字信号的转换一般来说并不是最终的目的,转换得到的数字量通常要经过微控制器的进一步处理。A/D 转换的一般步骤如图所示。

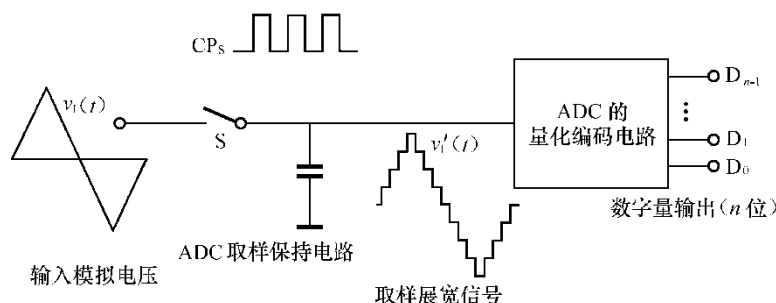


图 A/D 转换的一般步骤

11.1.2 A/D 转换的技术指标

(1) 分辨率 (Resolution)

数字量变化一个最小量时模拟信号的变化量,定义为满刻度与 2^n 的比值。分辨率又称精度,通常以数字信号的位数来表示。A/D 转换器的分辨率以输出二进制(或十进制)数的位数表示。从理论上讲, n 位输出的 A/D 转换器能区分 2^n 个不同等级的输入模拟电压,能区分输入电压的最小值为满量程输入的 $1/2^n$ 。在最大输入电压一定时,输出位数愈多,量化单位愈小,分辨率愈高。例如 EXYNOS4412 的 A/D 转换器可以设置输出为 10 位二进制数,输入信号最大值为 3.3V,那么这个转换器应能区分输入信号的最小电压为 3.22mV。

(2) 转换速率 (Conversion Rate)



完成一次从模拟转换到数字的 A/D 转换所需的时间的倒数。积分型 A/D 的转换时间是毫秒级，属低速 A/D；逐次比较型 A/D 是微秒级，属中速 A/D；全并行/串并行型 A/D 可达到纳秒级。采样时间则是另外一个概念，是指 2 次转换的间隔。为了保证转换的正确完成，采样速率（Sample Rate）必须小于或等于转换速率。因此有人习惯上将转换速率在数值上等同于采样速率也是可以接受的。常用单位是 ksps 和 Msps，表示每秒采样千/百万次（kilo / Million Samples per Second）。

（3） 量化误差（Quantizing Error）

由于 A/D 的有限分辨率而引起的误差，即有限分辨率 A/D 的阶梯状转移特性曲线与无限分辨率 A/D（理想 A/D）的转移特性曲线（直线）之间的最大偏差。通常是 1 个或半个最小数字量的模拟变化量，表示为 1LSB、1/2LSB。量化和量化误差示意图如图所示。

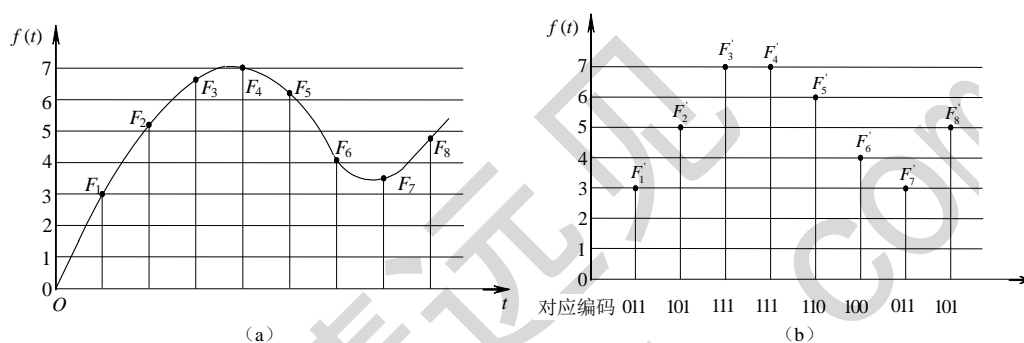


图 量化与量化误差

（4） 偏移误差（Offset Error）

输入信号为零时输出信号不为零的值，可外接电位器调至最小。

（5） 满度误差（Full Scale Error）

满度输出时对应的输入信号与理想输入信号值之差。

（6） 线性度（Linearity）

实际转换器的转移函数与理想直线的最大偏移，不包括以上 3 种误差。

其他指标还有绝对精度（Absolute Accuracy）、相对精度（Relative Accuracy）、微分非线性、单调性和无错码、总谐波失真（Total Harmonic Distortion, THD）和积分非线性。

11.1.3 A/D 转换器类型

下面简要介绍常用的几种类型的 A/D 转换器的基本原理及特点：积分型、逐次逼近型、并行比较型/串并行型、 Σ - Δ 调制型、电容阵列逐次比较型及压频变换型。

1. 积分型 A/D 转换器

积分型 A/D 转换器工作原理是将输入电压转换成时间（脉冲宽度信号）或频率（脉冲频率），然后由定时器/计数器获得数字值。积分型 A/D 实际上是 V-T 方式电压对时间的转换，先对输入量化电压以固定时间正向积分，然后再对基准电压反向积分，计数就是对应的 A/D 结果值。

双积分型 A/D 转换是一种间接 A/D 转换技术。首先将模拟电压转换成积分时间，然后用数字脉冲计时方法转换成计数脉冲数，最后将此代表模拟输入电压大小的脉冲数转换成二进制或 BCD 码输出。因此，



双积分型 A/D 转换器转换时间较长，一般要大于 40~50ms。其优点是用简单电路就能获得高分辨率，但缺点是由于转换精度依赖于积分时间，因此转换速率极低。初期的单片 A/D 转换器大多采用积分型，现在逐次比较型已逐步成为主流。

如图所示为双积分型 A/D 的控制逻辑。积分器是转换器的核心部分，它的输入端所接开关 S_1 由定时信号控制。当定时信号为不同电平时，极性相反的输入电压 u_i 和参考电压 V_{REF} 将分别加到积分器的输入端，进行两次方向相反的积分，积分时间常数 $\tau = RC$ 。

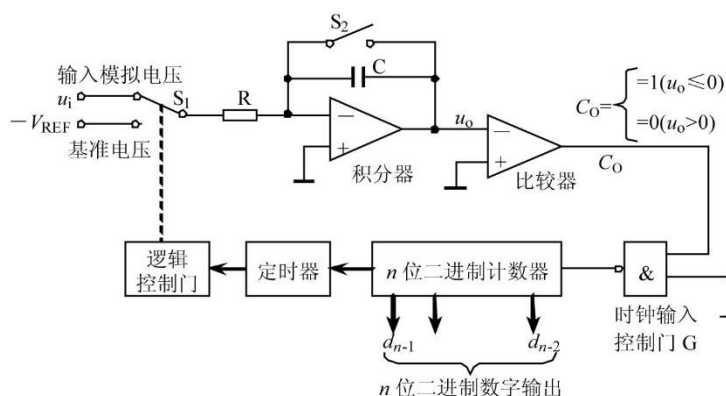


图 双积分型 A/D 控制逻辑图

过零比较器用来确定积分器的输出电压 u_o 过零的时刻。当 $u_o \geq 0$ 时，比较器输出电压为低电平；当 $u_o < 0$ 时，比较器输出电压为高电平。比较器的输出信号接至时钟控制门（G）作为关门和开门信号。

双积分型 A/D 转换器具有很强的抗干扰能力，故而采用双积分型 A/D 转换器可大大降低对滤波电路的要求。

2. 逐次逼近型 A/D

逐次逼近型 A/D 由逐次寄存器、比较器、同精度的 D/A、基准电压组成。从 MSB 开始，顺序地对每一位将输入电压与内置 DA 转换器输出进行比较，经 n 次比较而输出数字值。其电路规模属于中等。其优点是速度较高、功耗低，在低分辨率（<12 位）时价格便宜，但高精度（>12 位）时价格很高。

4 位逐次比较型 A/D 转换器的逻辑电路如图所示。

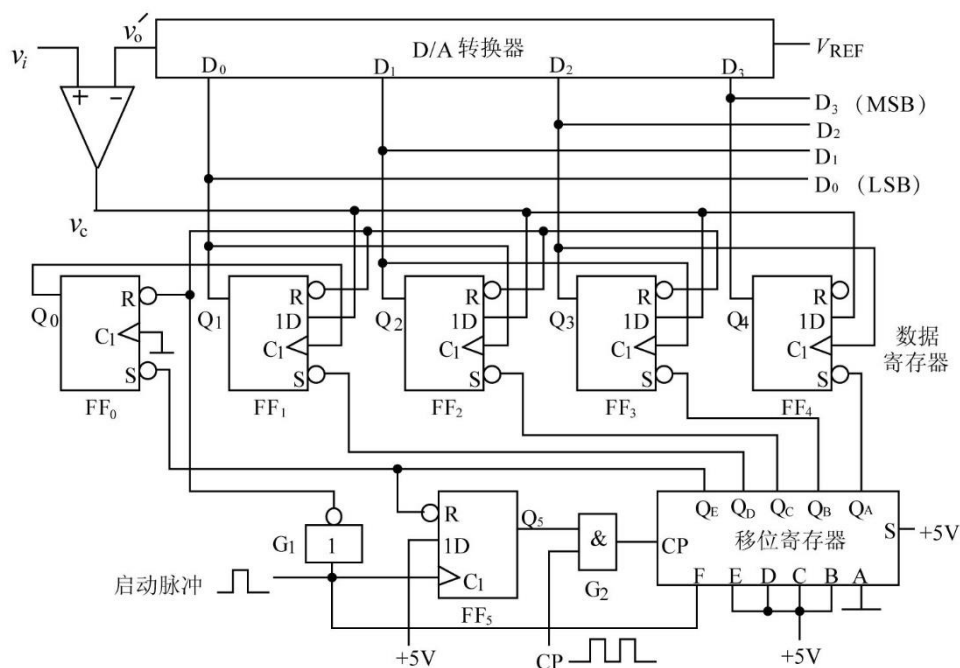


图 逐次逼近型 A/D 原理图

图中 5 位移位寄存器可进行并入/并出或串入/串出操作，其输入端 F 为并行置数使能端，高电平有效。其输入端 S 为高位串行数据输入。数据寄存器由 D 边沿触发器组成，数字量从 $Q_4 \sim Q_1$ 输出。

电路工作过程如下，当启动脉冲上升沿到达后， $FF_0 \sim FF_4$ 被清零， Q_5 置 1， Q_5 的高电平开启与门 G_2 ，时钟脉冲 CP 进入移位寄存器。在第 1 个 CP 脉冲作用下，由于移位寄存器的置数使能端 F 由 0 变 1，并行输入数据 ABCDE 置入， $Q_A Q_B Q_C Q_D Q_E = 01111$ ， Q_A 的低电平使数据寄存器的最高位 (Q_4) 置 1，即 $Q_4 Q_3 Q_2 Q_1 = 1000$ 。D/A 转换器将数字量 1000 转换为模拟电压，送入比较器 C 与输入模拟电压 v_i 比较，若 $v_i > v_o$ ，则比较器 C 输出 v_c 为 1，否则为 0。比较结果送 $D_4 \sim D_1$ 。

第 2 个 CP 脉冲到来后，移位寄存器的串行输入端 S 为高电平， Q_A 由 0 变 1，同时最高位 Q_A 的 0 移至次高位 Q_B 。于是数据寄存器的 Q_3 由 0 变 1，这个正跳变作为有效触发信号加到 FF_4 的 CP 端，使 v_c 的电平得以在 Q_4 保存下来。此时，由于其他触发器无正跳变触发脉冲， v_c 的信号对它们不起作用。 Q_3 变 1 后，建立了新的 D/A 转换器的数据，输入电压再与其输出电压 进行比较，比较结果在第 3 个时钟脉冲作用下存于 Q_3 ……如此进行，直到 Q_E 由 1 变 0 时，使触发器 FF_0 的输出端 Q_0 产生由 0 到 1 的正跳变，做触发器 FF_1 的 CP 脉冲，使上一次 A/D 转换后的 v_c 电平保存于 Q_1 。同时使 Q_5 由 1 变 0 后将 G_2 封锁，一次 A/D 转换过程结束。于是电路的输出端 $D_3 D_2 D_1 D_0$ 得到与输入电压 v_i 成正比的数字量。

逐次逼近转换过程和用天平称物重非常相似。天平称重物过程是，从最重的砝码开始试放，与被称物体进行比较，若物体重于砝码，则该砝码保留，否则移去。再加上第二个次重砝码，由物体的重量是否大于砝码的重量决定第二个砝码是留下还是移去。如此一直加到最小一个砝码为止。将所有留下的砝码重量相加，就得此物体的重量。仿照这一思路，逐次比较型 A/D 转换器，就是将输入模拟信号与不同的参考电压做多次比较，使转换所得的数字量在数值上逐次逼近输入模拟量对应值。

3. 并行比较/串行比较型 A/D

3 位并行比较型 A/D 转换原理电路如图所示，它由电压比较器、寄存器和代码转换器 3 部分组成。

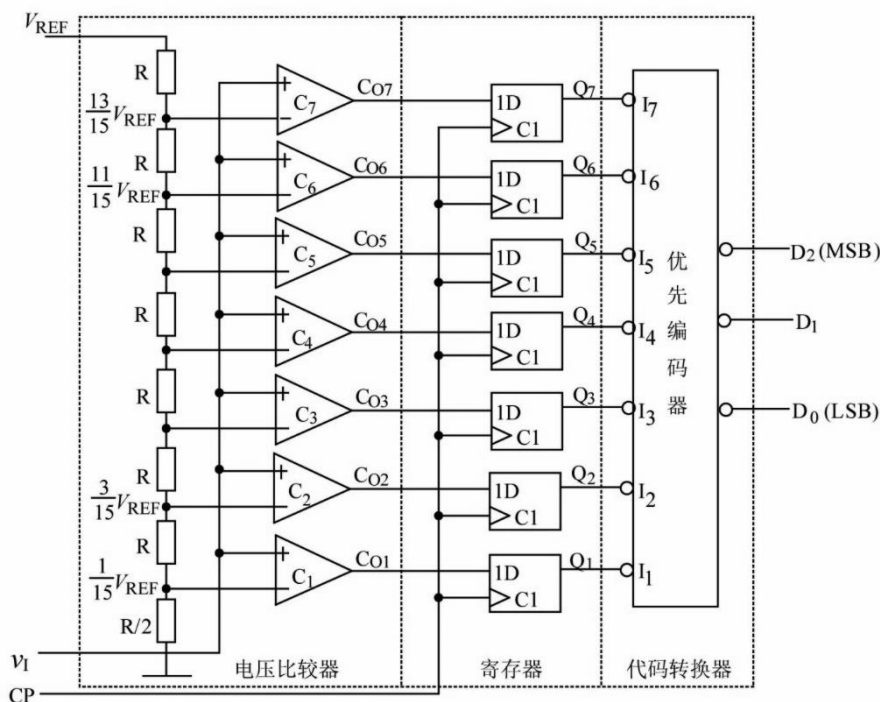


图 11-5 并行比较型 A/D

首先在电压比较器中进行量化电平的划分，用电阻链把参考电压 V_{REF} 分压，得到从 $\frac{1}{15} V_{REF} \sim \frac{13}{15} V_{REF}$ 之间 7 个比较电平。然后，把这 7 个比较电平分别接到 7 个比较器 $C_1 \sim C_7$ 的输入端作为比较基准。同时将输入的模拟电压同时加到每个比较器的另一个输入端上，与这 7 个比较基准进行比较。

并行 A/D 转换器具有如下特点。

- (1) 由于转换是并行的，其转换时间只受比较器、触发器和编码电路延迟时间限制，因此转换速度最快。
- (2) 随着分辨率的提高，元件数目要按几何级数增加。一个 n 位转换器，所用的比较器个数为 $2^n - 1$ ，如 8 位的并行 A/D 转换器就需要 $2^8 - 1 = 255$ 个比较器。由于位数愈多，电路愈复杂，因此制成分辨率较高的集成并行 A/D 转换器是比较困难的。
- (3) 使用这种含有寄存器的并行 A/D 转换电路时，可以不用附加取样—保持电路，因为比较器和寄存器这两部分也兼有取样—保持功能。这也是该电路的一个优点。

图 11-5 中的 8 个电阻将参考电压 V_{REF} 分成 8 个等级，其中 7 个等级的电压分别作为 7 个比较器 $C_1 \sim C_7$ 的参考电压，其数值分别为 $V_{REF}/15$ 、 $3V_{REF}/15 \cdots 13V_{REF}/15$ 。输入电压为 v_1 ，它的大小决定各比较器的输出状态，如当 $0 \leq v_1 < V_{REF}/15$ 时， $C_7 \sim C_1$ 的输出状态都为 0；当 $3V_{REF}/15 \leq v_1 < 5V_{REF}/15$ 时，比较器 C_6 和 C_7 的输出 $CO_6 = CO_7 = 1$ ，其余各比较器的状态均为 0。根据各比较器的参考电压值，可以确定输入模拟电压值与各比较器输出状态的关系。比较器的输出状态由 D 触发器存储，经优先编码器编码，得到数字量输出。优先编码器优先级别最高是 I_7 ，最低的是 I_1 。

设 v_1 变化范围是 $0 \sim V_{REF}$ ，输出 3 位数字量为 $D_2 D_1 D_0$ ，3 位并行比较型 A/D 转换器的输入、输出关系如表所示。



表 3 位并行 A/D 转换器输入与输出关系对照表

模拟输入	比较器输出状态							数字输出	
	C ₀₁	C ₀₂	C ₀₃	C ₀₄	C ₀₅	C ₀₆	C ₀₇	D ₂	D ₁
$0 \leq V_1 < V_{REF}/15$	0	0	0	0	0	0	0	0	0
$V_{REF}/15 \leq V_1 < 3V_{REF}/15$	0	0	0	0	0	0	1	0	0
$3V_{REF}/15 \leq V_1 < 5V_{REF}/15$	0	0	0	0	0	1	1	0	1
$5V_{REF}/15 \leq V_1 < 7V_{REF}/15$	0	0	0	0	1	1	1	0	1
$7V_{REF}/15 \leq V_1 < 9V_{REF}/15$	0	0	0	1	1	1	1	1	0
$9V_{REF}/15 \leq V_1 < 11V_{REF}/15$	0	0	1	1	1	1	1	1	0
$11V_{REF}/15 \leq V_1 < 13V_{REF}/15$	0	1	1	1	1	1	1	1	1
$13V_{REF}/15 \leq V_1 < V_{REF}$	1	1	1	1	1	1	1	1	1

由于转换是并行的，其转换时间只受比较器、触发器和编码电路延迟时间的限制，因此转换速度最快。随着分辨率的提高，元件数目要按几何级数增加。一个 n 位转换器，所用比较器的个数为 $2^n - 1$ ，如 8 位的并行 A/D 转换器就需要 255 个比较器。由于位数愈多，电路愈复杂，因此制成分辨率较高的集成并行 A/D 转换器是比较困难的。精度取决于分压网络和比较电路。动态范围取决于 V_{REF} 。

4. 电容阵列逐次比较型

电容阵列逐次比较型 A/D 在内置 D/A 转换器中采用电容矩阵方式，也可称为电荷再分配型。一般的电阻阵列 D/A 转换器中多数电阻的值必须一致。在单芯片上生成高精度的电阻并不容易，如果用电容阵列取代电阻阵列，可以用低廉的成本制成高精度的单片 A/D 转换器。最近的逐次比较型 A/D 转换器大多为电容阵列式的。

5. 压频变换型

压频变换型（Voltage-Frequency Converter）是通过间接转换方式实现模数转换的。其原理是首先将输入的模拟信号转换成频率，然后用计数器将频率转换成数字量。从理论上讲这种 A/D 的分辨率几乎可以无限增加，只要采样的时间能够满足输出频率分辨率要求的累积脉冲个数的宽度。其优点是分辨率高、功耗低、价格低，但是需要外部计数电路共同完成 A/D 转换。

11.1.4 A/D 转换的一般步骤

模拟信号进行 A/D 转换的时候，从启动转换到转换结束输出数字量，需要一定的转换时间，在这个转换时间内，模拟信号要基本保持不变。否则转换精度没有保证，特别是当输入信号频率较高时，会造成很大的转换误差。要防止这种误差的产生，必须在 A/D 转换开始时将输入信号的电平保持住，而在 A/D 转换结束后，又能跟踪输入信号的变化。因此，一般的 A/D 转换过程是通过取样、保持、量化和编码这 4 个步骤完成的。一般取样和保持主要由采样保持器来完成，而量化编码就由 A/D 转换器完成。

11.2 EXYNOS4412 A/D 转换器

11.2.1 EXYNOS4412 A/D 转换器概述

1. 简述



10 位或 12 位 CMOS 再循环式模拟数字转换器，它具有 4 通道输入，并可将模拟量转换为 10 位或 12 位二进制数。5MHz A/D 转换时钟时，最大 1MSPS。A/D 转换操作具有样本保持的功能，同时也支持低功耗模式。

2. 特性

ADC 接口包括如下特性:

- 10bit/12bit 输出位可选。
- 微分误差 $\pm 2.0\text{LSB}$ 。
- 积分误差 $\pm 4.0\text{LSB}$ 。
- 最大转换速率: 1 Msps。
- 功耗少，电压输入 3.3v。
- 模拟量输入范围: 0~3.3v。
- 支持片上样本保持功能。
- 通用转换模式。

3. 模块图

EXYNOS4412 A/D 转换器的控制器接口框图如图所示。

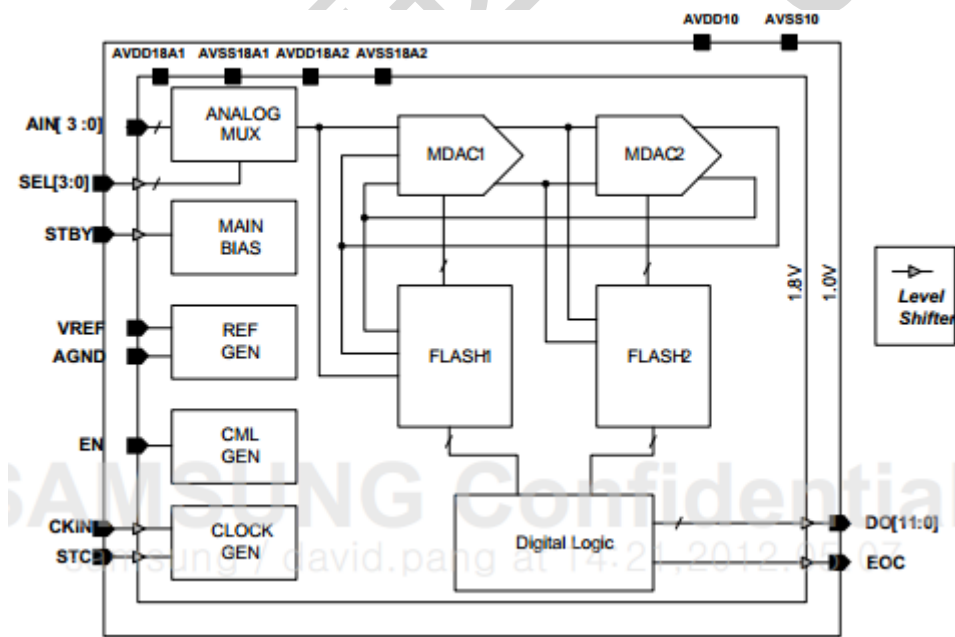


图 EXYNOS4412 ADC 控制器接口框图

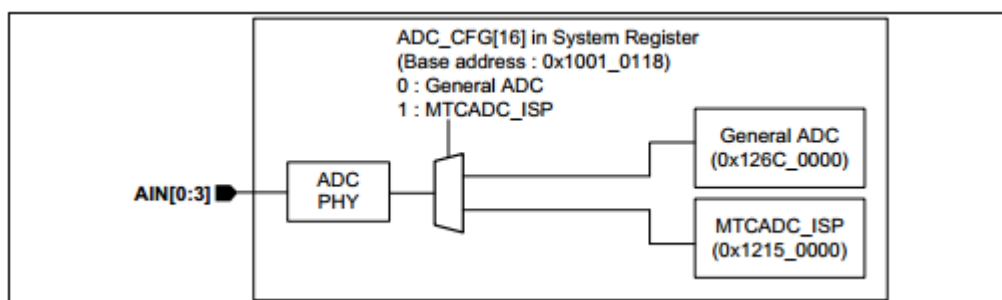


图 ADC 选择



11.2.2 EXYNOS4412 A/D 控制器寄存器

1. 寄存器组

EXYNOS4412 中的 A/D 控制器集成了电阻触摸屏控制功能。此处为了简化学习，只考虑 A/D 转换使用到的两个寄存器，即 A/D 控制寄存器（ADCCON）、A/D 转换数据寄存器（ADCDAT）。

A/D 控制寄存器 ADCCON（address = 0x126C_0000）如表所示。

表 TSADCCON 描述

ADCCON	位	描 述	初 始 值
RES	[16]	0=10bit 输出 1=12bit 输出	0
ECFLG	[15]	A/D 转换结束标志 0: A/D 转换正在进行 1: A/D 转换结束	0
PRSCEN	[14]	A/D 转换预分频允许 0: 不允许预分频 1: 允许预分频	0
PRSCVL	[13:6]	预分频值 PRSCVL	0xFF
Reserved	[5:3]	保留	0
STANDBY	[2]	待机模式选择位 0: 正常模式 1: 待机模式	1
READ_START	[1]	A/D 转换读—启动选择位 0: 禁止 Start-by-read 1: 允许 Start-by-read	0
ENABLE_START	[0]	A/D 转换器启动 0: A/D 转换器不工作 1: A/D 转换器开始工作	0

A/D 转换数据寄存器 ADCDAT（地址 0x126C_000C）如表所示。

表 ADCDAT 描述

ADCDAT	Bit	描 述	初 始 值
DATA	[11:0]	X 坐标转换数据值（包括正常的 ADC 转换数值）	—

2. A/D 转换的转换时间计算

例如，PCLK 为 66MHz，PRESCALER = 65；所有 10 位转换时间为：

$$66 \text{ MHz} / (65 + 1) = 1 \text{ MHz}$$

转换时间为 $1/(1\text{M}/5 \text{ cycles}) = 5\mu\text{s}$ 。

完成一次 A/D 转换需要 5 个时钟周期。A/D 转换器的最大工作时钟为 5MHz，所以最大的采样率可以达到 1Mbit/s。



11.3 A/D 实验

11.3.1 实验目的

- 掌握模数转换 ADC 的原理；
- 掌握 EXYNOS4412 处理器的 A/D 转换功能；

11.3.2 实验原理

如图所示，ADC 电路连接如图所示，利用一个电位计输出电压到 EXYNOS4412 的 ADC_IN1 引脚。输入的电压范围是 0~1.8V。旋转电位器 PR 使 ADCIN1 和 GND 两端的电压发生变化，即 XadcAIN3 引脚采集变化的模拟电压，即 ADC 控制器的数据寄存器可输出对应二进制数值。

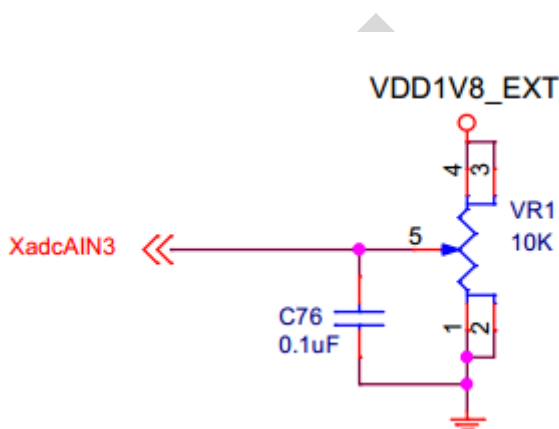


图 分压电路

11.3.3 实验内容

- (1) 寄存器设置：
 - a) 上电使能 ADC 控制器
 - b) 选择转换通道。
 - c) 使能 ADC 读操作和设置分频因子
 - d) 设置工作模式和转换分辨率
- (2) 程序编写如下：

编写软件程序，实现电压值的获取、显示。程序主要是对 EXYNOS4412 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 (ADC DAT) 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。

仿真程序调试，PMIC_InitIp()中将上电使能 ADC 控制器。依据原理图，将 ADC 控制器的模拟信号输入端 3 选通采集模拟信号。通过配置 A/D 控制寄存器 (ADCCON)，选择 12 位精度，时钟预分频为 255，选择读启动转换。采用查询的方式读，然后通过宏 `READ_START` 的设置，设定是否通过读取 A/D 转换数据寄存器 (ADC DAT)，启动下次 ADC 转换，还是手动再次使能转换。

相关代码如下：



```

/*-----MAIN FUNCTION-----*/
/*****
 * @brief      Main program body
 * @param[in]  None
 * @return     int
 *****/
int main(void)
{
    unsigned int  temp_adc = 0, temp_mv;

    GPX2.CON = 0x1 << 28; //GPX2CON[7]: Output drive LED

    uart_init();

    ADC_CFG &= ~(0x1 << 16); //Bit_16:Select ADC Mux 0:General 1:MTCADC

    ADCMUX = 0x3;    //0x3: 0011 = AIN3

#ifdef __READ_START_
    //12bit A/D conversion; enable A/D converter prescaler;
    //prescaler value:255; A/D conversion start by read
    ADCCON = (0x1<<16) | (0x1<<14) | (0xff<<6) | 0x1<<1;
#else
    //12bit A/D conversion; enable A/D converter prescaler; prescaler value:255
    ADCCON = (0x1<<16) | (0x1<<14) | (0xff<<6);
#endif

#ifdef __READ_START_
    temp_adc = ADCDAT & 0xffff;
#endif

    printf("\n***** ADC test *****\n");

    while(1)
    {
        //Turn on LED
        GPX2.DAT |= 0x1 << 7;

#ifdef __READ_START_
        ADCCON |= 0x1; //start ADC conversion
#endif

        mydelay_ms(100);
    }
}

```



```
44         while(!(ADCCON & (0x1<<15))); //等待转换完成
45
46         temp_adc = ADCDAT & 0xfff;
47
48         temp_mv = 1800 * temp_adc / 4095;
49
50         printf("adc value: %d mv\n", temp_mv);
51
52         mydelay_ms(500);
53
54         //Turn off LED
55         GPX2.DAT &= ~(0x1 << 7);
56         mydelay_ms(500);
57     }
58     return 0;
59 }
60
```

11.3.4 实验步骤

1、 导入工程源码

请参考第 1 章节的 ARM 开发环境搭建部分。

光盘实验源码路径：**【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\09-ADC】**

2、 连接好开发板及 FS-JTAG 仿真器，并且连接好配线。

方法请参考第 1 章节的开发环境搭建的硬件连接部分按对应型号连接。


3、 配置好串口终端

方法请参考第 1 章节的 ARM 开发环境搭建部分。

4、 仿真运行程序

方法请参考第 1 章节的 ARM 开发环境搭建部分。

11.3.5 实验现象

Debug 调试点击运行按钮 ，旋转电位器 VR1，查看串口调试助手上的打印信息的变化，逆时针旋转 ADC 采集的数据变小，反之，数值变大。如图所示：

通过串口中断观察打印信息，旋转开发板上的电位器，观察电压的变化。



电位器

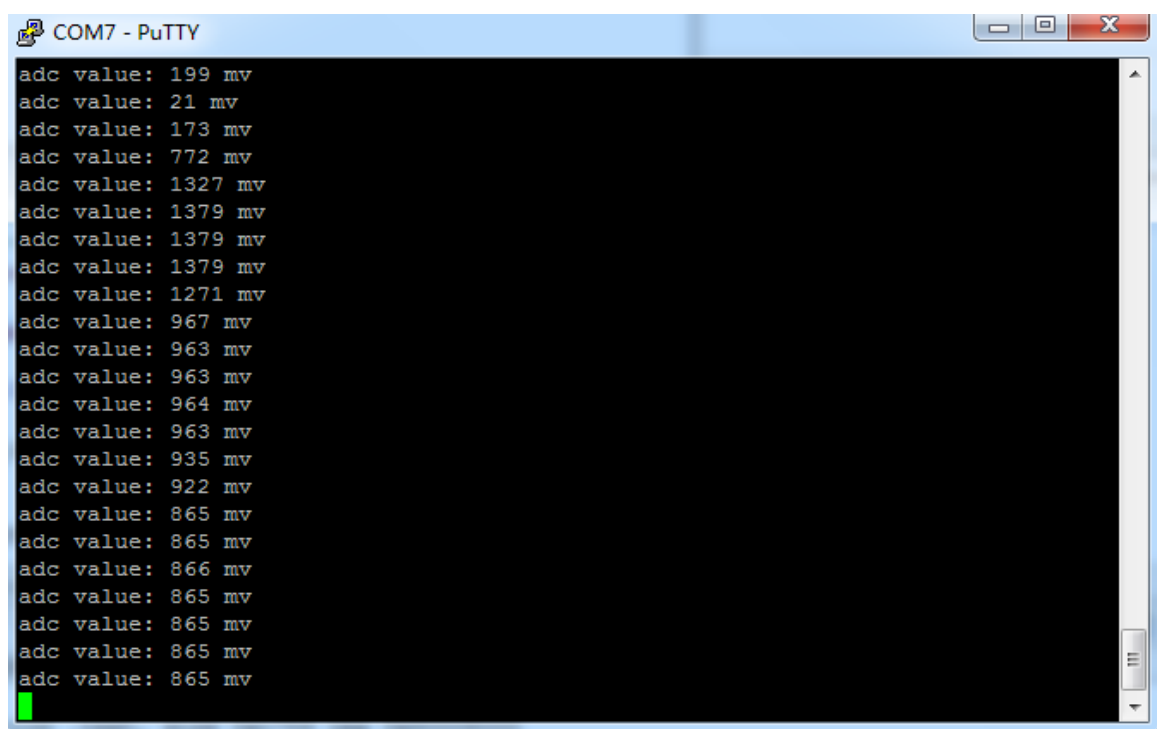


图 ADC 转换的数据

11.4 本章小结

本章主要讲解了 A/D 转换器的工作原理，以及 EXYNOS4412 下 A/D 控制器的操作方法。

11.5 练习题

- 1、 A/D 转换器选型时需要考虑哪些指标？
- 2、 根据 A/D 的基本原理，可以将 A/D 控制器分为哪些种类？
- 3、 在 PCLK 为 50MHz 的情况，如何设置 EXYNOS4412 的 A/D 控制器来实现采集速度为 100Ksps？
- 4、 编程实现采集一个范围在 0~1.8V 的电压的测试程序。



第 12 章 I2C 接口

为了使读者掌握常见的 I2C 总线，这一章将从理论到实际应用从头梳理一遍，目的在于给读者一个完整的概念，不仅在理论上掌握了解 I2C 总线，更要在实际运用中灵活使用。

本章要点：

- I2C 总线协议。
- EXYNOS4412 的 I2C 控制器。

12.1 I2C 总线

12.1.1 I2C 总线介绍

I2C (Inter-Integrated Circuit) 总线 (也称 IIC 或 I²C) 是由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备，是微电子通信控制领域广泛采用的一种总线标准。它是同步通信的一种特殊形式，具有接口线少，控制方式简单，器件封装形式小，通信速率较高等优点。I2C 有着如下的特点。

- 两条总线线路：一条串行数据线 SDA，一条串行时钟线 SCL。
- 每个连接到总线的器件都可以通过唯一的地址联系主机，同时主机可以作为主机发送器或主机接收器。
- 它是一个真正的多主机总线，如果两个或更多主机同时初始化，数据传输可以通过冲突检测和仲裁防止数据被破坏。
- 串行的 8 位双向数据传输位速率在标准模式下可达 100kb/s，快速模式下可达 400kb/s，高速模式下可达 3.4Mb/s。
- 连接到相同总线的 IC 数量只受到总线的最大电容 400pF 限制。

12.1.2 I2C 总线术语

- 发送器：发送数据到总线的器件。
- 接收器：从总线接收数据的器件。
- 主机：初始化发送产生时钟信号和终止发送的器件。
- 从机：被主机寻址的器件。
- 多主机：同时有多于一个主机尝试控制总线但不破坏传输。
- 仲裁：是一个在有多个主机同时尝试控制总线但只允许其中一个控制总线并使传输不被破坏的过程。
- 同步：两个或多个器件同步时钟信号的过程。

12.1.3 I2C 总线位传输

由于连接到 I2C 总线的器件有不同种类的工艺 (CMOS、NMOS、双极性)，逻辑 0 (低) 和逻辑 1 (高) 的电平不是固定的，它由电源 VCC 的相关电平决定，每传输一个数据位就产生一个时钟脉冲，数据有效性如图所示。

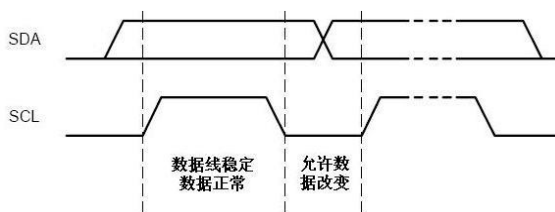


图 数据有效性

SDA 线上的数据必须在时钟的高电平周期保持稳定。数据线的高或低电平状态 I2C 位传输数据有效性在 SCL 线的时钟信号是低电平时才能改变，起始和停止条件如图 16-2 所示。

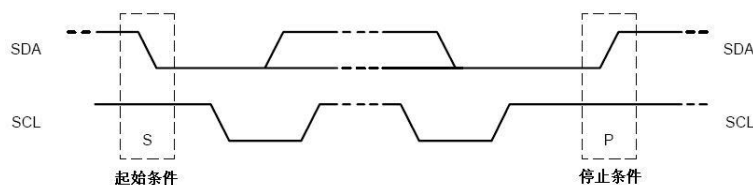


图 起始和停止条件

SCL 线是高电平时，SDA 线从高电平向低电平切换，这个情况表示起始条件；SCL 线是高电平时，SDA 线由低电平向高电平切换，这个情况表示停止条件。起始和停止条件一般由主机产生，总线在起始条件后被认为处于忙状态，在停止条件的某段时间后总线被认为再次处于空闲状态。如果产生重复起始条件而不产生停止条件，总线会一直处于忙的状态，此时的起始条件（S）和重复起始条件（Sr）在功能上是一样的。

12.1.4 I2C 总线数据传输

(1) 字节格式

发送到 SDA 线上的每个字节必须为 8 位，每次传输可以发送的字节数量不受限制。每个字节后必须跟一个响应位。首先传输的是数据的最高位（MSB），如果从机要完成一些其他功能后（如一个内部中断服务程序）才能接收或发送下一个完整的数据字节，可以使时钟线 SCL 保持低电平，迫使主机进入等待状态，当从机准备好接收下一个数据字节并释放时钟线 SCL 后数据传输继续。

(2) 应答响应

应答响应如图所示。

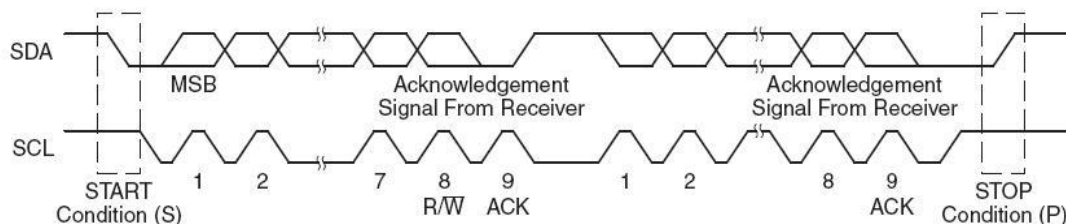


图 应答响应



数据传输必须带响应，相关的响应时钟脉冲由主机产生。在响应的时钟脉冲期间发送器释放 SDA 线（高）。在响应的时钟脉冲期间，接收器必须将 SDA 线拉低，使它在这个时钟脉冲的高电平期间保持稳定的低电平。

通常被寻址的接收器在接收到每个字节后，会产生一个响应。当从机不能响应从机地址时（如它正在执行一些实时函数不能接收或发送），从机必须使数据线保持高电平，主机然后产生一个停止条件终止传输或者产生重复起始条件开始新的传输。

如果从机接收器响应了从机地址，但是在传输了一段时间后不能接收更多数据字节，主机必须再一次终止传输。这个情况用从机在第一个字节后没有产生响应来表示。从机使数据线保持高电平，主机产生一个停止或重复起始条件。

如果传输中有主机接收器，它必须在从机不产生时钟的最后一个字节不产生响应，向从机发送器通知数据结束。从机发送器必须释放数据线，允许主机产生一个停止或重复起始条件。

12.1.5 I2C 总线寻址方式

1、7 位寻址

第一个字节的头 7 位组成了从机地址，最低位（LSB）是第 8 位，它决定了普通的和带重复开始条件的 7 位地址格式方向。第一个字节的最低位是“0”，表示主机会写信息到被选中的从机；“1”表示主机会向从机读信息，当发送了一个地址后，系统中的每个器件都在起始条件后将头 7 位与它自己的地址比较，如果一样，器件会判定它被主机寻址，至于从机接收器还是从机发送器，都由 R/W 位决定。

2、10 位寻址

10 位寻址和 7 位寻址兼容，而且可以结合使用。10 位寻址采用了保留的 1111XXX 作为起始条件，或重复起始条件的后第一个字节的头 7 位。10 位寻址不会影响已有的 7 位寻址，有 7 位和 10 位地址的器件可以连接 I2C 总线 10 位地址格式到相同的 I2C 总线。它们都能用于标准模式和高速模式系统。

10 位从机地址由在起始条件或重复起始条件后的头两个字节组成。第一个字节的头 7 位是 11110XX 的组合，其中最后两位 XX 是 10 位地址的两个最高位（MSB）。第一个字节的第 8 位是 R/W 位，决定了传输的方向，第一个字节的最低位是“0”，表示主机将写信息到选中的从机，“1”表示主机将向从机读信息。如果 R/W 位是“0”，则第二个字节是 10 位从机地址剩下的 8 位；如果 R/W 位是“1”，则下一个字节是从机发送给主机的数据。

12.1.6 快速和高速模式

1、快速模式

快速模式器件可以在 400kb/s 下接收和发送。最小要求是：它们可以和 400kb/s 传输同步，可以延长 SCL 信号的低电平周期来减慢传输。快速模式器件都向下兼容，可以和标准模式器件在 0~100kb/s 的 I2C 总线系统通信。但是，由于标准模式器件不向上兼容，所以不能在快速模式 I2C 总线系统中工作。快速模式 I2C 总线规范与标准模式相比有以下另外的特征：

- （1） 最大位速率增加到 400kb/s。
- （2） 调整了串行数据（SDA）和串行时钟（SCL）信号的时序。



- (3) 快速模式器件的输入有抑制毛刺的功能, SDA 和 SCL 输入有施密特触发器。
- (4) 快速模式器件的输出缓冲器对 SDA 和 SCL 信号的下降沿有斜率控制功能。
- (5) 如果快速模式器件的电源电压被关断, SDA 和 SCL 的 I/O 引脚必须悬空, 不能阻塞总线。
- (6) 连接到总线的外部上拉器件必须调整以适应快速模式 I2C 总线更短的最大允许上升时间。对于负载最大是 200pF 的总线, 每条总线的上拉器件可以是一个电阻, 对于负载在 200~400pF 之间的总线, 上拉器件可以是一个电流源 (最大值 3mA) 或者是一个开关电阻电路。

2、高速模式

高速模式 (Hs 模式) 器件对 I2C 总线的传输速度有很大的突破。高速模式器件可以在高达 3.4Mb/s 的位速率下传输信息, 而且保持完全向下兼容快速模式或标准模式器件, 它们可以在一个速度混合的总线系统中双向通信。

高速模式传输除了不执行仲裁和时钟同步外, 与快速模式系统有相同的串行总线协议和数据格式。

12.2 I2C 总线控制器

12.2.1 EXYNOS4412 下的 I2C 控制器介绍

EXYNOS4412 处理器支持多主机 I2C 串行总线接口, 并且它支持主机发送模式、主机接收模式、从机发送模式和从机接收模式这 4 种模式, 包括四个通道的 I2C 总线接口, 分别用于通用目的、电源管理 PMIC 和高清晰度多媒体接口 HDMI, 如图所示为 I2C 总线的概括图。

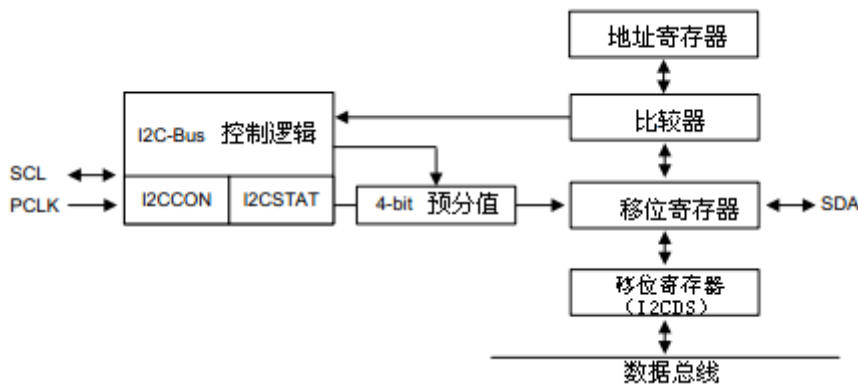


图 I2C 总线的概括图

12.2.2 I2C 总线控制寄存器详解

如表所示为 I2C 总线控制寄存器描述。

表 I2C 总线控制寄存器

I2CCON	位	描述	复位值
保留	[31:8]	保留	0
应答产生	[7]	IIC 应答产生使能位 0 = 禁止 1 = 使能	0
Tx 时钟源选择	[6]	IIC 传输时钟预分频值选择位	0



		0 = I2CCLK = fPCLK /16 1 = I2CCLK = fPCLK /512	
Tx/Rx 中断	[5]	I C-Bus Tx/Rx 中断控制位 0 = 禁止 1 = 使能	0
中断挂起标志位	[4]	0(读) = 未产生中断 1(读) = 产生中断 0(写) = 无效 1(写) = 恢复操作	0
传输时钟值	[3:0]	IIC 总线时钟预分频 Tx clock = I2CCLK / (I2CCON[3:0]+1)	未定义

如表所示为 I2C 状态寄存器描述。

表 I2C 状态寄存器

I2CSTAT	位	描述	复位值
模式选择	[7:6]	IIC 总线 主/从 Tx/Rx 模式选择位 00 = 从接收模式 01 = 从发送模式 10 = 主接收模式 11 = 主发送模式	00
忙信号状态位	[5]	IIC 总线忙信号状态位 读: 0 = 准备 1 = 忙 写: 产生开启信号	0
串行输出	[4]	IIC 总线数据输出使能/禁止位 0 = 禁止 Rx/Tx 1 = 使能 Rx/Tx	0
仲裁状态标志	[3]	0 = 仲裁成功 1 = 失效	0
从地址状态标志	[2]	0 = 当开启/停止条件检测到时清除 1 = 接收到的从地址匹配 I2CADD 的地址值	0
地址 0 状态标志	[1]	0 = 当开启/停止条件检测到时清除 1 = 接收从地址值为 00000000b	0
最后接收位状态标志	[0]	0 = 为 0 1 = 为 1	0

如表所示为 I2C 从机地址寄存器描述。

表 I2C 从机地址寄存器

I2CDS	位	描述	复位值
从机地址	[7:0]	7 位从地址[7:1], 其中[0]位不用 该寄存器随时可读 当 I2CSTAT 的 Rx/Tx 串行输出使能时可写	未定义

如表所示为 I2C 数据发送/接收移位寄存器描述。



表 I2C 数据发送/接收移位寄存器

I2CDS	位	描述	复位值
数据移位	[7:0]	8-位数据移位寄存器 如果串行输出使能，则 I2CDS 将变为可写。并且 I2CDS 任何时刻都是可读的，不管当前 I2CSTAT 的设置	未定义

12.3 I2C 重力感应/陀螺仪实验

12.3.1 实验目的

- 掌握 I2C 串行数据通信协议的使用方法；
- 掌握 EEPROM 器件的读写访问方法；
- 掌握 EXYNOS4412 处理器的 I2C 控制器的使用；

12.3.2 实验原理

现在结合上面已经提到的 I2C 理论基础，我们将以一个例子来进行实际讲解，用 I2C 来操作三轴加速度传感器/陀螺仪芯片 MPU6060。主要有以下特性：

- 1、 Digital-output triple-axis accelerometer with a programmable full scale range of $\pm 2g$, $\pm 4g$, $\pm 8g$ and $\pm 16g$
- 2、 Integrated 16-bit ADCs enable simultaneous sampling of accelerometers while requiring no external multiplexer
- 3、 Accelerometer normal operating current: $500\mu A$
- 4、 Low power accelerometer mode current: $10\mu A$ at 1.25Hz, $20\mu A$ at 5Hz, $60\mu A$ at 20Hz, $110\mu A$ at 40Hz
- 5、 Orientation detection and signaling
- 6、 Tap detection
- 7、 User-programmable interrupts
- 8、 High-G interrupt
- 9、 User self-test

如图所示为 MPU6050 系列的器件地址组成方式。

I ² C ADDRESS	AD0 = 0 AD0 = 1		1101000 1101001			
--------------------------	--------------------	--	--------------------	--	--	--

AD0 是 MPU6050 的第 9 管脚。如果 AD0 接地，地址为：1101000，如果拉高，地址为：1101001。

如图所示为 MPU6050 的电路原理图。

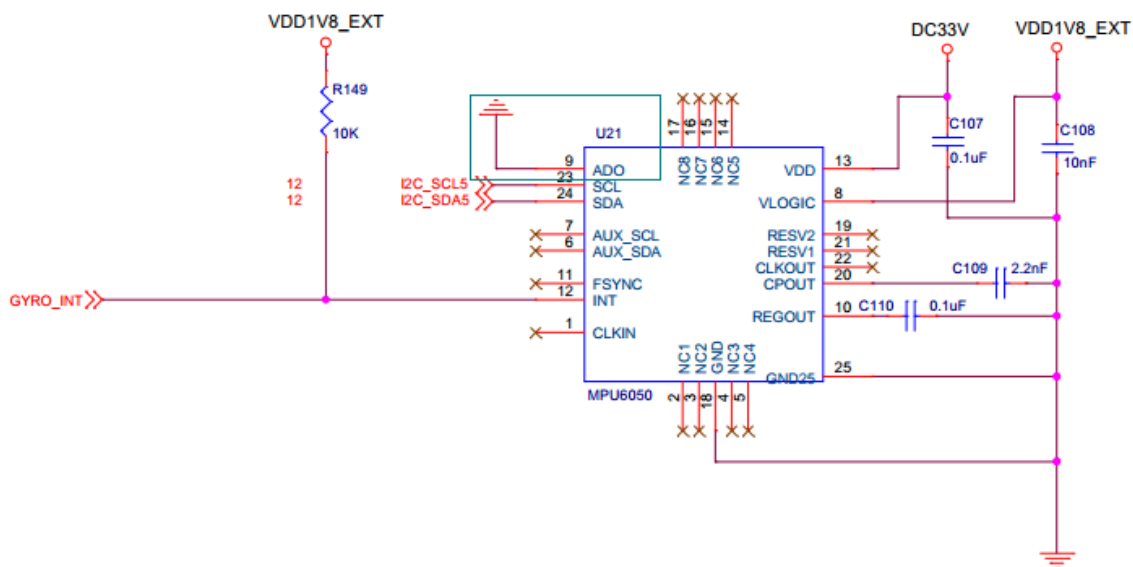


图 MPU6050 原理图

可以看到 SDA/SCL 被接到了 EXYNOS4412 的 IIC 控制器上，下面介绍 MPU6050 的几种操作时序，分别是字节写时序、页写时序、当前地址读、随机地址读和顺序读时序。

(1) 字节写时序 (Byte Write) 如图所示。

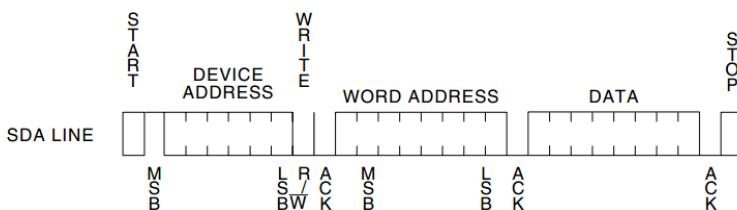


图 MPU6050 字节写时序

如上图所示，字节写时序依次要发送器件地址（包括 LSB 用于读/写，此时为 0 写方向）、器件片内数据写入地址和写入的 8 位数据。

(2) 随机地址读时序 (RANDOM READ) 如图所示。

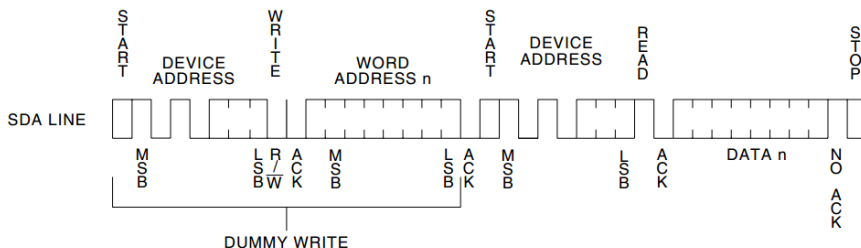


图 MPU6050 字节读时序

如上图所示，字节读时序依次要发送器件地址（包括 LSB 用于读/写，此时为 0 写方向）、器件片内数据要读出数据的地址、停止信号、再次写入开始信号、器件地址（包括 LSB 用于读/写，此时为 1 读方向）、读出 8 位目标数据。



12.3.3 实验内容

编写程序，实现对重力感应芯片 MPU6050 的 Z 轴坐标读操作

- (1) 寄存器设置：
 - a) 配置 GPIO 为 I2C 模式
 - b) 写 I2C 设备地址
 - c) 写 I2C 页地址
 - d) 读/写 I2C 芯片 MPU6050 的 Z 轴寄存器寄存器数据
- (2) 程序编写如下：

I2C 读、写操作函数：实现向 I2C 的一个地址写数据、读数据。

```

1  /*****
2  * @brief      iic write a byte program body
3  * @param[in]  slave_addr, addr, data
4  * @return     None
5  *****/
6  void iic_write (unsigned char slave_addr, unsigned char addr, unsigned char data)
7  {
8      I2C5.I2CDS = slave_addr;
9      I2C5.I2CCON = 1<<7 | 1<<6 | 1<<5; /*ENABLE ACK BIT, PRESCALER:512, ,ENABLE RX/TX */
10     I2C5.I2CSTAT = 0x3 << 6 | 1<<5 | 1<<4; /*Master Trans mode ,START ,ENABLE RX/TX ,*/
11     while(!(I2C5.I2CCON & (1<<4)));
12
13     I2C5.I2CDS = addr;
14     I2C5.I2CCON &= ~(1<<4); //Clear pending bit to resume.
15     while(!(I2C5.I2CCON & (1<<4)));
16
17     I2C5.I2CDS = data; // Data
18     I2C5.I2CCON &= ~(1<<4); //Clear pending bit to resume.
19     while(!(I2C5.I2CCON & (1<<4)));
20
21     I2C5.I2CSTAT = 0xD0; //stop
22
23     I2C5.I2CCON &= ~(1<<4);
24
25     mydelay_ms(10);
26 }
27
28 /*****
29 * @brief      iic read a byte program body
30 * @param[in]  slave_addr, addr, &data

```



```

30  * @return      None
31  *****/
32  void iic_read(unsigned char slave_addr, unsigned char addr, unsigned char *data)
33  {
34      I2C5.I2CDS = slave_addr;
35
36      I2C5.I2CCON = 1<<7 | 1<<6 | 1<<5; /*ENABLE ACK BIT, PRESCALER:512, ,ENABLE RX/TX */
37      I2C5.I2CSTAT = 0x3 << 6 | 1<<5 | 1<<4; /*Master Trans mode ,START ,ENABLE RX/TX ,*/
38      while(!(I2C5.I2CCON & (1<<4)));
39
40      I2C5.I2CDS = addr;
41      I2C5.I2CCON &= ~(1<<4); //Clear pending bit to resume.
42      while(!(I2C5.I2CCON & (1<<4)));
43      I2C5.I2CSTAT = 0xD0; //stop
44
45
46      I2C5.I2CDS = slave_addr | 0x01; // Read
47      I2C5.I2CCON = 1<<7 | 1<<6 | 1<<5; /*ENABLE ACK BIT, PRESCALER:512, ,ENABLE RX/TX */
48
49      I2C5.I2CSTAT = 2<<6 | 1<<5 | 1<<4; /*Master receive mode ,START ,ENABLE RX/TX ,*/
50      while(!(I2C5.I2CCON & (1<<4)));
51
52      I2C5.I2CCON &= ~((1<<7) | (1<<4)); /* Resume the operation & no ack*/
53      while(!(I2C5.I2CCON & (1<<4)));
54
55      I2C5.I2CSTAT = 0x90;
56      I2C5.I2CCON &= ~(1<<4); /*clean interrupt pending bit */
57
58      *data = I2C5.I2CDS;
59      mydelay_ms(10);
60  }
61

```

MPU6050_Init 函数：实现 MPU6050 的初始化。

```

1  /*****
2  * @brief      MPU6050_Init program body
3  * @param[in]  None
4  * @return     None
5  *****/
6  void MPU6050_Init ()
7  {
8      iic_write(SlaveAddress, PWR_MGMT_1, 0x00);

```



```

9     iic_write(SlaveAddress, SMPLRT_DIV, 0x07);
10    iic_write(SlaveAddress, CONFIG, 0x06);
11    iic_write(SlaveAddress, GYRO_CONFIG, 0x18);
12    iic_write(SlaveAddress, ACCEL_CONFIG, 0x01);
13 }
    
```

MPU6050 读函数：实现从 MPU6050 内部地址读数据。

```

1     /*****
2     * @brief      get MPU6050 data program body
3     * @param[in]  addr
4     * @return     int
5     *****/
6     int get_data(unsigned char addr)
7     {
8         char data_h, data_l;
9         iic_read(SlaveAddress, addr, &data_h);
10        iic_read(SlaveAddress, addr+1, &data_l);
11        return (data_h<<8)|data_l;
12    }
13
    
```

主函数：实现从 MPU6050 内部 Z 轴地址读数据。

```

1     /*-----MAIN FUNCTION-----*/
2     /*****
3     * @brief      Main program body
4     * @param[in]  None
5     * @return     int
6     *****/
7     int main(void)
8     {
9
10        unsigned char zvalue;
11
12        GPX2.CON = 0x1 << 28;
13
14        GPB.CON = (GPB.CON & ~(0xF<<12)) | 0x3<<12; // GPBCON[3], I2C_5_SCL
15        GPB.CON = (GPB.CON & ~(0xF<<8)) | 0x3<<8;   // GPBCON[2], I2C_5_SDA
16
    
```



```
17     mydelay_ms(100);
18     uart_init();
19
20     /*-----*/
21     I2C5.I2CSTAT = 0xD0;
22     I2C5.I2CCON &= ~(1<<4);    /*clean interrupt pending bit */
23     /*-----*/
24
25     mydelay_ms(100);
26     MPU6050_Init();
27     mydelay_ms(100);
28
29     printf("\n***** I2C test!! *****\n");
30
31     while(1)
32     {
33         //Turn on
34         GPX2.DAT |= 0x1 << 7;
35
36         data = get_data(GYRO_ZOUT_H);
37         printf(" GYRO --> Z <---:Hex: %0x", data);
38         printf("\n");
39
40         mydelay_ms(20);
41         //Turn off
42         GPX2.DAT &= ~(0x1 << 7);
43         mydelay_ms(500);
44     }
45     return 0;
}
```

12.3.4 实验步骤

1、 导入工程源码

请参考第 1 章节的 ARM 开发环境搭建部分。

光盘实验源码路径：【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\14-i2c】

2、 连接好开发板及 FS-JTAG 仿真器，并且连接好配线。

方法请参考第 1 章节的开发环境搭建的硬件连接部分按对应型号连接。

3、 配置好串口终端

方法请参考第 1 章节的 ARM 开发环境搭建部分。



4、仿真运行程序

方法请参考第 1 章节的 ARM 开发环境搭建部分。

12.3.5 实验现象

实验现象如图所示：通过串口终端打印出 Z 轴的陀螺仪数值。转动电路板，观察变化规律。

```
COM7 - PuTTY
GYRO --> Z <---:Hex: 3
GYRO --> Z <---:Hex: 3
GYRO --> Z <---:Hex: 2
GYRO --> Z <---:Hex: 2
GYRO --> Z <---:Hex: 2
GYRO --> Z <---:Hex: 2
GYRO --> Z <---:Hex: 4
GYRO --> Z <---:Hex: 2
GYRO --> Z <---:Hex: 2
GYRO --> Z <---:Hex: 3
GYRO --> Z <---:Hex: 3
GYRO --> Z <---:Hex: 59
GYRO --> Z <---:Hex: 2e
GYRO --> Z <---:Hex: fa
GYRO --> Z <---:Hex: ffd6
GYRO --> Z <---:Hex: 5e
GYRO --> Z <---:Hex: 6a
GYRO --> Z <---:Hex: fff2
GYRO --> Z <---:Hex: 5
GYRO --> Z <---:Hex: 2
GYRO --> Z <---:Hex: ffff
GYRO --> Z <---:Hex: 0
GYRO --> Z <---:Hex: ffe2
```




第 13 章 SPI 接口

SPI 作为应用最为广泛的通信总线协议之一，开发人员应当掌握，本章将介绍 SPI 总线协议的基本理论，以及 EXYNOS4412 的 SPI 总线控制器的操作方法。

本章要点：

- SPI 总线协议。
- EXYNOS4412 下 SPI 总线控制器详解。

13.1 SPI 总线协议理论

13.1.1 协议简介

SPI 是英文 Serial Peripheral Interface 的缩写，该协议是由美国摩托罗拉公司推出的一种同步串行传输规范，首先由摩托罗拉公司在其 MC68HCXX 系列处理器上定义，后主要应用在 EEPROM、FLASH、实时时钟、AD 转换器，还有数字信号处理器和数字信号解码器之间。

SPI 是一种高速的全双工、同步的通信总线，并且在芯片的引脚上只占用四根线，节约了芯片的引脚，同时为 PCB 的布局上节省空间，提供方便，正是出于这种简单易用的特性，现在越来越多的芯片集成了这种通信协议。

13.1.2 协议内容

SPI 有 4 个引脚：CS（从器件选择线）、SDO（串行数据输出线）、SDI（串行数据输入线）和 SPICLK（同步串行时钟线）。

SPI 的通信原理很简单，它以主从方式工作，这种模式通常有一个主设备和一个或多个从设备，需要至少 4 根线，事实上 3 根也可以（单向传输时）。也是所有基于 SPI 的设备共有的，这些脚的定义如下：

- (1) SDO (MOSI) ——主设备数据输出，从设备数据输入。
- (2) SDI (MISO) ——主设备数据输入，从设备数据输出。
- (3) SPICLK ——时钟信号，由主设备产生。
- (4) CS ——从设备使能信号，由主设备控制。

其中 CS 是控制芯片是否被选中的，也就是说只有片选信号为预先规定的使能信号时（高电位或低电位），对此芯片的操作才有效。这就使在同一总线上连接多个 SPI 设备成为可能。其中总线协议时序如图所示。

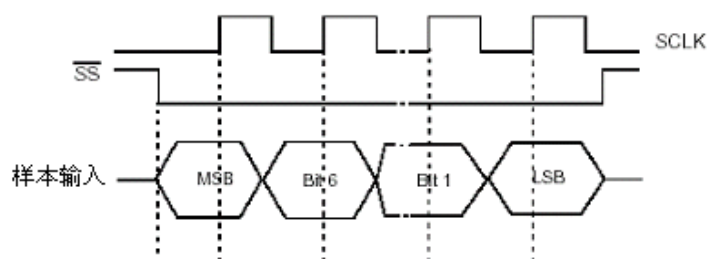


图 SPI 总线协议时序



接下来就是负责通信的 3 根线了。通信是通过数据交换完成的，这里先要知道 SPI 是串行通信协议，也就是说数据是一位一位地传输的。这就是 SPICLK 时钟线存在的原因，由 SPICLK 提供时钟脉冲，SDO、SDI 则基于此脉冲完成数据传输。数据输出通过 SDO 线，数据在时钟上升沿或下降沿时改变，在紧接着的下降沿或上升沿被读取，完成一位数据传输，输入也使用同样原理。这样，在至少 8 次时钟信号的改变（上沿和下沿为一次）后，就可以完成 8 位数据的传输。

要注意的是，SPICLK 信号线只由主设备控制，从设备不能控制信号线。同样在一个基于 SPI 的设备中，至少有一个主控设备。这样传输的特点：这样的传输方式有一个优点，即其与普通的串行通信不同，普通的串行通信一次连续传送至少 8 位数据，而 SPI 允许数据一位一位地传送，甚至允许暂停，因为 SPICLK 时钟线由主控设备控制，当没有时钟跳变时，从设备不采集或传送数据。也就是说，主设备通过对 SPICLK 时钟线的控制可以完成对通信的控制。SPI 还是一个数据交换协议：因为 SPI 的数据输入和输出线独立，所以允许同时完成数据的输入和输出。不同的 SPI 设备的实现方式不尽相同，主要是数据改变和采集的时间不同，在时钟信号上沿或下沿采集有不同定义。

在点对点的通信中，SPI 接口不需要进行寻址操作，且为全双工通信，显得简单高效。在多个从设备的系统中，每个从设备需要独立地使能信号，在硬件上要比 I2C 总线控制稍微复杂一些。

注意：

SPI 的一个缺点是没有指定的流控制，没有应答机制确认是否接收到数据。

SPI 控制器为了和外设进行数据交换，根据外设工作要求，其输出串行同步时钟极性和相位可以进行配置，时钟极性（CPOL）对传输协议没有重大影响。如果 CPOL=0，串行同步时钟的空闲状态为低电平；如果 CPOL=1，串行同步时钟的空闲状态为高电平。时钟相位（CPHA）能够配置用于选择两种不同的传输协议之一进行数据传输。如果 CPHA=0，在串行同步时钟的第一个跳变沿（上升或下降）数据被采样，如图 15-2 所示；如果 CPHA=1，在串行同步时钟的第二个跳变沿（上升或下降）数据被采样，如图 15-3 所示。SPI 主控制器和与之通信的外设时钟相位和极性应该一致，另外，上面提到这些特性将在寄存器中具体实现。

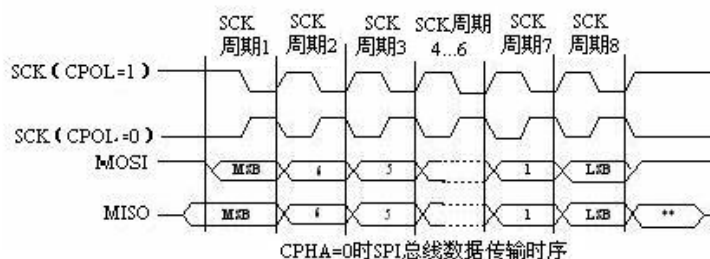


图 CPHA=0 时的情况

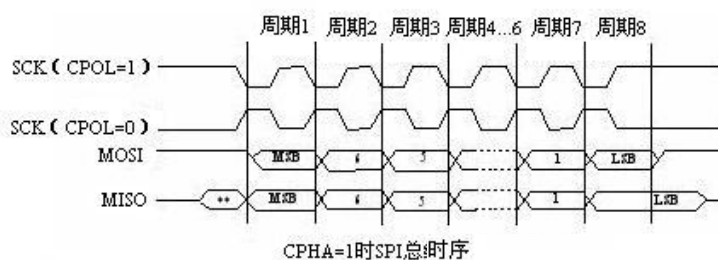


图 CPHA=1 时的情况

13.2 SPI 控制器详解

13.2.1 EXYNOS4412 的 SPI 控制器简介

EXYNOS4412 包含了两套 8 位、16 位、32 位移位寄存器用于收发。在 SPI 传输数据时，数据的发送及接收数据是同步的，该总线控制器支持摩托罗拉串行外设接口。

下面是该控制器的特性：

- 全双工通信方式。
- 8 位、16 位、32 位移位寄存器。
- 2 时钟源供应。
- 支持 8 位、16 位、32 位总线接口。
- 支持摩托罗拉 SPI 协议。
- 支持两个独立的传输及接收 FIFO。
- 支持主机模式及从机模式。
- 无法传送条件下接收。
- Tx/Rx 频率最大支持 50MHz。

13.2.2 时钟源控制

每个 SPI 都能获得不同的 2 个时钟源，用户可以根据自己的需要进行配置，需要设置 CLK_CFG 这个寄存器，后面将会介绍。

时钟控制器如图所示。

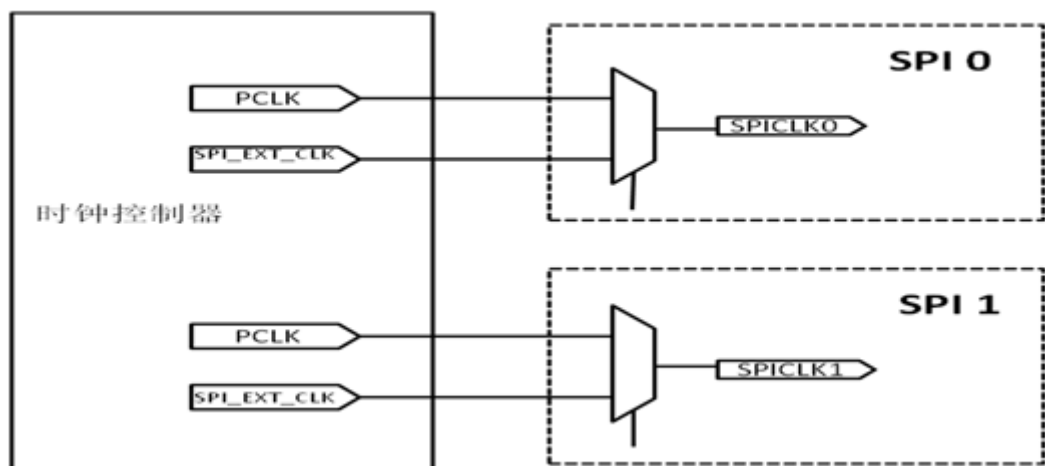




图 时钟控制器

13.2.3 寄存器详解

如表所示为 SPI 配置寄存器。

表 SPI 配置寄存器

CH_CFGn	位	描述	复位值
HIGH_SPEED_EN	[6]	从机模式下 TX 输出时间控制位 0: 禁止 1: 使能 (输出时间为 SPICLK/2)	0
SW_RST	[5]	软件复位	0
SLAVE	[4]	主从模式选择位 0: 主机模式 1: 从机模式	0
CPOL	[3]	CLK 时钟线初始状态位 0: 高 1: 低	0
CPHA	[2]	线上相位传输方式选择位 0: 方式 A 1: 方式 B	0
PX_CH_ON	[1]	SPI 接受通道(RX)使能位 0: 禁止 1: 使能	0
TX_CH_ON	[0]	SPI 接受通道(TX)使能位 0: 禁止 1: 使能	0

如表所示为时钟配置寄存器。

表 时钟配置寄存器

CLK_CFGn	位	描述	复位值
SPI_CLKSEL	[9]	时钟源选择 0 = PCLK 1 = SPI_EXT_CLK	0
ENCLK	[8]	时钟使能 0 = 禁止 1 = 使能	0
SPI_SCALER	[7:0]	SPI 时钟分频值 SPI 时钟输出 = 时钟源 / (2 x (预分频值 + 1))	0

如表所示为 SPI 模式配置寄存器。

表 SPI 模式配置寄存器

MODE_CFGn	位	描述	复位值
CH_WIDTH	[30:29]	通道宽度选择位 00 = 字节 01 = 半字 10 = 字 11 = 保留	0
TRAILING_CNT	[28:19]	接收 FIFO 中最后写入字节的个数	0
BUS_WIDTH	[18:17]	SPI FIFO 宽度选择位 00 = 字节 01 = 半字 10 = 字 11 = 保留	0

如表所示为 SPI 数据发送寄存器。



表 SPI 数据发送寄存器

SPI_TX_DATAn	位	描述	复位值
TX_DATA	[31:0]	该寄存器包含了所要发送的数据	0

表所示为 SPI 数据接收寄存器。

表 SPI 数据接收寄存器

SPI_RX_DATAn	位	描述	复位值
RX_DATA	[31:0]	该寄存器包含了所要接收的数据	0

如表所示为 SPI 状态寄存器。

表 SPI 状态寄存器

SPI_STATUSn	位	描述	复位值
TX_DONE	[25]	0 = 其他情况 1 = 发送移位寄存器准备	0
RX_FIFO_LVL	[23:15]	RX FIFO 0 ~ 256 字节 in port0 0 ~ 64 字节 in port	0
TX_FIFO_LVL	[14:6]	TX FIFO 0 ~ 256 字节 in port0 0 ~ 64 字节 in port	0
RX_OVERRUN	[5]	Rx Fifo 溢出错误 0 = 无误 1 = 溢出错	0
RX_UNDERRUN	[4]	0 = 无误 1 = 数据缺失	0
TX_OVERRUN	[3]	Tx Fifo 溢出错误 0 = 无误 1 = 溢出错	0
TX_UNDERRUN	[2]	0 = 无误 1 = 数据缺失	0

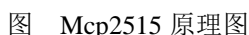
13.3 SPI/CAN 总线实验

13.3.1 实验目的

- 掌握 mcp2515 控制器的使用
- 掌握 can 总线的原理
- 掌握 EXYNOS4412 处理器的 spi 功能

13.3.2 实验原理

现在结合上面已经提到的 I2C 理论基础，我们将以一个例子来进行实际讲解。这里将介绍一种通过 SPI 信号转 CAN 总线信号的例子，使用 CAN 总线控制器 MCP2515，如图所示为该芯片的接线原理图，每根接线的意义已经清楚地标识出来了。



这一款芯片内部集成了 9 条指令，包括了通用的读、写、配置等命令，还有一个内置的状态寄存器，可以通过该寄存器获取芯片当前状态。

如表所示为 MCP2515 芯片指令集。

表 MCP2515 芯片指令集

仿真程序调试，使能 SPI0 控制器，通过配置 SPI0 相关寄存器，选择主机模式，总线宽度和通道宽度均设置为 8bit，片选 MCP2515 芯片。然后初始化 CAN 控制器，将其设置为回环模式。连续读取 8 次终端上的用户输入，将 8 位数据通过 SPI 发送给 CAN 控制器，CAN 的回环模式接收到发送的数据，再通过 SPI 发送到 EXYNOS4412 内的 SPI 控制器中。



13.3.3 实验内容

有了上文的知识做铺垫，现在我们先来看一下 SPI 控制器及 CAN 总线控制器 MCP2515 的基本编程方法：

- a) 设置时钟源并配置分频值等参数。
- b) 软复位后，并设置 SPI 配置寄存器（SPI CONFIGURATION REGISTER）。
- c) 设置模式寄存器。
- d) 设置从机选择寄存器。
- e) 收发数据。

(2) 程序编写如下：

根据以上信息，这里分成若干模块来逐一实现，相关代码如下：

(1) 相关寄存器结构体定义。

```

1  /*
2  *SPI2 REGISTERS
3  */
4  typedef struct {
5      unsigned int CH_CFG      ;
6      unsigned int RESERVED; // 4412's SPI has no CLK_CFG register
7      unsigned int MODE_CFG   ;
8      unsigned int CS_REG     ;
9      unsigned int SPI_INT_EN ;
10     unsigned int SPI_STATUS ;
11     unsigned int SPI_TX_DATA;
12     unsigned int SPI_RX_DATA;
13     unsigned int PACKET_CNT_REG ;
14     unsigned int PENDING_CLR_REG ;
15     unsigned int SWAP_CFG      ;
16     unsigned int FB_CLK_SEL   ;
17 }spi2;
18 #define SPI2 (* (volatile spi2 *)0x13940000 )

```

(2) 延时函数，片选从机芯片及取消片选芯片的实现如下。

```

1 void delay(int times)
2 {
3     volatile int i, j;
4     for (j = 0; j < times; j++)

```



```

4   {
5       for (i = 0; i < 1000; i++);
6   }
7 }
8 /*
9  * 片选从机
10 */
11 void slave_enable(void)
12 {
13     SPI2.CS_REG &= ~0x1; //enable salve
14     delay(3);
15 }
16 /*
17  * 取消片选从机
18 */
19 void slave_disable(void)
20 {
21     SPI2.CS_REG |= 0x1; //disable salve
22     delay(1);
23 }
24

```

(3) 软件复位的代码如下。

```

1 /*
2  * 复位spi控制器
3  */
4 void soft_reset(void)
5 {
6     SPI2.CH_CFG |= 0x1 << 5;
7     delay(1); //延时
8     SPI2.CH_CFG &= ~(0x1 << 5);
9 }

```

(4) 指定地址字节读的实现。

如下图,为了实现指定地址读字节功能,首先传送命令字 0x03 告知从机为读操作,然后传送 CAN 控制器片内地址,上面的两次传送功能是通过函数 send_byte 实现的,而 send_byte 函数通过写 EXYNOS4412 内的发送数据寄存器 SPI_TX_DATA2,自动通过 SPI 总线发送到 CAN 控制器。随后, CAN 总线将返回对应地址数据。通过 recv_byte 函数接收,函数通过读取 EXYNOS4412 内的接收数据寄存器 SPI_RX_DATA2 获得数据。读时序如图所示。

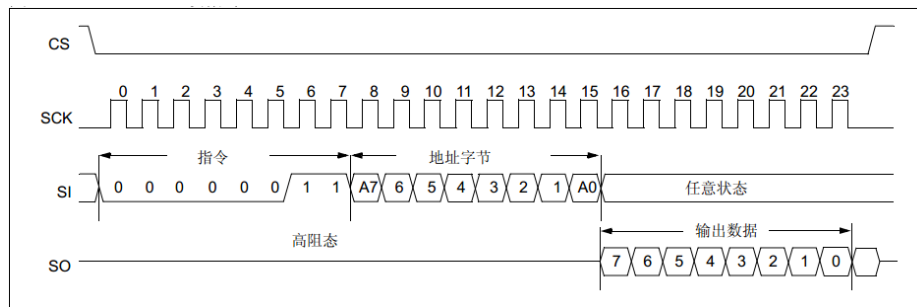


图 读时序

```

1  /*
2   * 功能：向SPI总线发送一个字节
3   */
4  void send_byte(unsigned char data)
5  {
6      SPI2.CH_CFG |= 0x1; // enable Tx Channel
7      delay(1);
8      SPI2.SPI_TX_DATA = data;
9      while( !(SPI2.SPI_STATUS & (0x1 << 25)) );
10     SPI2.CH_CFG &= ~0x1; // disable Tx Channel
11 }
12 /*
13 * 功能：从SPI总线读取一个字节
14 */
15 unsigned char recv_byte()
16 {
17     unsigned char data;
18     SPI2.CH_CFG |= 0x1 << 1; // enable Rx Channel
19     delay(1);
20     data = SPI2.SPI_RX_DATA;
21     delay(1);
22     SPI2.CH_CFG &= ~(0x1 << 1); //disable Rx Channel
23     return data;
24 }
25 //
26 /*
27 * 功能：从指定地址起始的寄存器读取数据。
28 * unsigned char Addr 要读取地址寄存器的地址
29 * 返回值：从地址当中读取的数值
30 */
31 unsigned char read_byte_2515(unsigned char Addr)
32 {
33     unsigned char ret;
34     // CS_SPI = 0;

```



```

35  slave_enable();
36  send_byte(0x03);
37  send_byte(Addr);
38  ret = recv_byte();
39  slave_disable();
40  //    CS_SPI = 1;
41  return(ret);
}

```

(5) 指定地址字节写的实现。

在发出写指令 0x20 后，要紧跟着发出第一个数据要存放的地址，这个时候再顺序写入数据写时序如图所示。

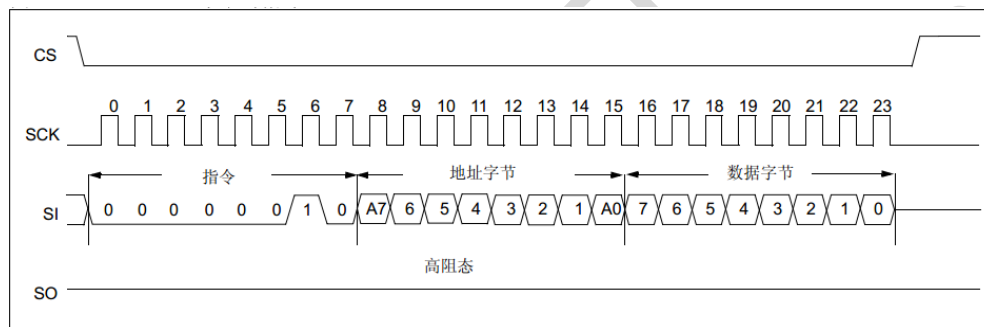


图 写时序

关键代码如下。

```

1  /*
2   * 功能：将数据写入指定地址起始的寄存器。
3   * unsigned char addr 寄存器的地址
4   * unsigned char data 向寄存器写入的数据
5   */
6  void write_byte_2515(unsigned char addr, unsigned char data)
7  {
8      //    CS_SPI = 0;
9      slave_enable();
10     send_byte(0x02);
11     send_byte(addr);
12     send_byte(data);
13     slave_disable();
14     //    CS_SPI = 1;
15 }
16

```



(6) 位修改指令的操作实现。

这块 CAN 控制器芯片提供了位修改功能，避免了传统的读改写操作，使代码更加简洁、高效。它可对特定状态和控制寄存器中单独的位进行置 1 或清零。该命令并非对所有寄存器有效，详见 MCP2515 芯片手册。将 CS 引脚置为低电平，向 MCP2515 发送位修改命令字节。随后依次发送寄存器地址、屏蔽字节以及数据字节。屏蔽字节中的“1”表示允许对寄存器中的相应位进行修改；而“0”则禁止修改。结合如图 15-8 所示的时序图。

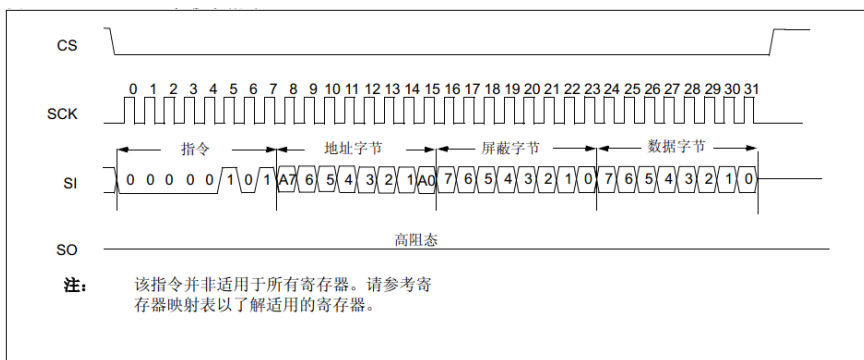


图 15-8 位修改指令时序图

关键代码如下。

```
1 //功能：只修改寄存器中的某些位
2 //入口参数：Addr:寄存器地址 MASK:屏蔽字 为1时可以对当前位修改 dat:数据字节
3 void bit_modify_2515(unsigned char addr, unsigned char mask, unsigned char data)
4 {
5     // CS_SPI = 0 ;
6     slave_enable() ;
7     send_byte(0x05) ;
8     send_byte(addr) ;
9     send_byte(mask) ;
10    send_byte(data) ;
11    slave_disable() ;
12    // CS_SPI = 1 ;
13 }
```

(7) 实现 SPI 的读写功能实验。

通过 SPI 往 MCP2515 控制器的某一寄存器写值 “0x80”，再通过 SPI 读此寄存器的值。

关键代码如下：

C++ Code

```
/*-----MAIN FUNCTION-----*/
/*****
2  * @brief      Main program body
3  * @param[in]  None
  * @return     int
  */
```



```

4  *****/
5  int main(void)
6  {
7      unsigned char data = 0;
8
9      GPX2.CON = 0x1 << 28;
10     uart_init();
11
12     GPC1.CON = (GPC1.CON & ~0xffff0) | 0x55550; //设置 IO 引脚为 SPI 模式
13
14     /*spi clock config*/
15     CLK_SRC_PERIL1 = (CLK_SRC_PERIL1 & ~(0xF<<24)) | 6<<24;
16     // 0x6: 0110 = SCLKMPLL_USER_T 800Mhz
17     CLK_DIV_PERIL2 = 19 << 8 | 3; //SPI_CLK = 800/(19+1)/(3+1)
18
19     soft_reset(); // 软复位 SPI 控制器
20     SPI2.CH_CFG &= ~( (0x1 << 4) | (0x1 << 3) | (0x1 << 2) | 0x3);
21     //master mode, CPOL = 0, CPHA = 0 (Format A)
22     SPI2.MODE_CFG &= ~( (0x3 << 17) | (0x3 << 29));
23     //BUS_WIDTH=8bit,CH_WIDTH=8bit
24     SPI2.CS_REG &= ~(0x1 << 1); //选择手动选择芯片
25     delay(10); //延时
26
27     printf("\n***** SPI test!! *****\n");
28     while(1)
29     {
30         reset_2515(); //复位
31         mydelay_ms(10);
32         printf("spi send '0x80' to 2515.....\n");
33         write_byte_2515(0x0f, 0x80);
34         //CANCTRL 寄存器——进入配置模式 中文 DATASHEET 58 页
35         mydelay_ms(10);
36         data = read_byte_2515(0x0f);
37         printf("spi receive a byte : 0x%0x\n", data);
38         mydelay_ms(1000);
39     }
40
41     return 0;
42 } //main
    
```



(8) 实现 CAN 总线回环模式读写功能实验

本代码通过展示 EXYNOS4412 上的片内 SPI 控制器的操作，在 SPI 总线协议下，实现 SOC 与 CAN 控制器 MCP2515 的通信，将 MCP2515 配置为回环模式。实现数据的自发自收。

C++ Code

```

1  /*-----MAIN FUNCTION-----*/
2  /*****
3   * @brief      Main program body
4   * @param[in]   None
5   * @return     int
6   *****/
7  int main(void)
8  {
9      GPX2.CON = 0x1 << 28;
10     uart_init();
11
12     unsigned char ID[4],buff[8]; //状态字
13     unsigned char key;
14     unsigned char ret;
15     //,j,k,ret0,ret1,ret2,ret3,ret4,ret5,ret6,ret7,ret8,ret9;
16     unsigned int rx_counter;
17     volatile int i=0;
18
19     GPC1.CON = (GPC1.CON & ~0xffff0) | 0x55550; //设置 IO 引脚为 SPI 模式
20
21     /*spi clock config*/
22     CLK_SRC_PERIL1 = (CLK_SRC_PERIL1 & ~(0xF<<24)) | 6<<24;
23     // 0x6: 0110 = SCLKMPLL_USER_T 800Mhz
24     CLK_DIV_PERIL2 = 19 << 8 | 3; //SPI_CLK = 800/(19+1)/(3+1)
25     soft_reset(); // 软复位 SPI 控制器
26     SPI2.CH_CFG &= ~( (0x1 << 4) | (0x1 << 3) | (0x1 << 2) | 0x3);
27     //master mode, CPOL = 0, CPHA = 0 (Format A)
28     SPI2.MODE_CFG &= ~( (0x3 << 17) | (0x3 << 29));
29     //BUS_WIDTH=8bit,CH_WIDTH=8bit
30     SPI2.CS_REG &= ~(0x1 << 1); //选择手动选择芯片
31     mydelay_ms(10); //延时
32     Init_can(); //初始化 MCP2515
33     printf("\n***** SPI CAN test!! *****\n");
34
35     while(1)
36     {
37         //Turn on
38         GPX2.DAT = GPX2.DAT | 0x1 << 7;
39         mydelay_ms(50);

```



```
40     printf("\nplease input 8 bytes\n");
41     for(i=0;i<8;i++)
42     {
43         src[i] = getchar();
44         putc(src[i]);
45     }
46     printf("\n");
47
48     Can_send(src); //发送标准帧
49     mydelay_ms(100);
50     ret = Can_receive(dst); //接收 CAN 总线数据
51     printf("ret=%x\n",ret);
52     printf("src=");
53     for(i=0;i<8;i++) printf(" %x", src[i]);
54     //将 CAN 总线上收到的数据发到串行口
55
56     printf("\n");
57
58     printf("dst=");
59     for(i=0;i<8;i++) printf(" %x",dst[6+i]);
60     //将 CAN 总线上收到的数据发到串行口
61     printf("\n");
62
63     //Turn off
64     GPX2.DAT = GPX2.DAT & ~(0x1 << 7);
65     mydelay_ms(100);
66 } //while(1)
67
68 return 0;
69 } //main
```

13.3.4 实验步骤

1、 导入工程源码

请参考第 1 章节的 ARM 开发环境搭建部分。

(1) 实现 SPI 的读写功能实验。

光盘实验源码路径:【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\15-SPI】

(2) 实现 CAN 总线回环模式读写功能实验。

光盘实验源码路径:【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\16-SPI_CAN】

2、 连接好开发板及 FS-JTAG 仿真器，并且连接好配线。

方法请参考第 1 章节的开发环境搭建的硬件连接部分按对应型号连接。



3、配置好串口终端

方法请参考第 1 章节的 ARM 开发环境搭建部分。

4、仿真运行程序

方法请参考第 1 章节的 ARM 开发环境搭建部分。

13.3.5 实验现象

(1) 实现 SPI 的读写功能实验。

实验现象如图所示：通过 SPI 往 2515 控制器的某一寄存器写值“0x80”，再通过 SPI 读此寄存器的值。

```
COM7 - PuTTY
Checking Boot Mode ... EMMC4.41
Net: dm9000
dm9000 i/o: 0x5000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
Hit any key to stop autoboot: 0
FS4412 # uTTY
***** SPI test!! *****
spi send '0x80' to 2515.....
spi receive a byte : 0x80
spi send '0x80' to 2515.....
spi receive a byte : 0x80
spi send '0x80' to 2515.....
spi receive a byte : 0x80
spi send '0x80' to 2515.....
spi receive a byte : 0x80
spi send '0x80' to 2515.....
spi receive a byte : 0x80
spi send '0x80' to 2515.....
spi receive a byte : 0x80
spi send '0x80' to 2515.....
spi receive a byte : 0x80
spi send '0x80' to 2515.....
spi receive a byte : 0x80
spi send '0x80' to 2515.....
spi receive a byte : 0x80
```

(2) 实现 CAN 总线回环模式读写功能实验

实验现象如图所示：通过 SPI 往 2515 控制器的某一寄存器写值



```
COM7 - PuTTY
Out:  serial
Err:  serial

MMC read: dev # 0, block # 48, count 16 ...16 blocks read: OK

Checking Boot Mode ... EMMC4.41
Net:  dm9000
dm9000 i/o: 0x50000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 11:22:33:44:55:66
Hit any key to stop autoboot: 0
FS4412 #
***** SPI CAN test!! *****

please input 8 bytes
12345678
flag=5
ret=0
src= 31 32 33 34 35 36 37 38
dst= 31 32 33 1a 35 36 37 38

please input 8 bytes
█
```

图 终端打印结果

13.4 本章小结

本章重点介绍了 SPI 总线协议及 SPI 总线控制器的基本编程方法，希望读者能取得完整代码并实际试验，完全掌握 SPI 总线是很有必要的。

13.5 练习题

- 1、 SPI 总线和 I2C 总线的区别是什么？
- 2、 请编写一个实现读/写 SPI 功能的程序。



第 14 章 MMU 虚拟内存管理

14.1 什么是 MMU

MMU 是 Memory Management Unit 的缩写, 针对各种 CPU, MMU 是个可选的配件。MMU 负责的是虚拟地址与物理地址的转换, 提供硬件机制的内存访问授权。现代的多用户多进程操作系统, 需要 MMU, 才能达到每个用户进程都拥有自己的独立的地址空间的目标。使用 MMU, OS 划分出一段地址区域, 在这块地址区域中, 每个进程看到的内容都不一定一样。例如 MICROSOFT WINDOWS 操作系统, 地址 4M-2G 处划分为用户地址空间。进程 A 在地址 0X400000 映射了可执行文件, 进程 B 同样在地址 0X400000 映射了可执行文件。如果 A 进程读地址 0X400000, 读到的是 A 的可执行文件映射到 RAM 的内容。而进程 B 读取地址 0X400000 时则读到的是 B 的可执行文件映射到 RAM 的内容。

14.2 MMU 作用

采用 MMU 还有利于选择性地将页面映射或解映射到逻辑地址空间。物理存储器页面映射至逻辑空间, 以保持当前进程的代码, 其余页面则用于数据映射。类似地, 物理存储器页面通过映射可保持进程的线程堆栈。RTOS 可以在每个线程堆栈解映射之后, 很容易地保留逻辑地址所对应的页面内容。这样, 如果任何线程分配的堆栈发生溢出, 将产生硬件存储器保护故障, 内核将挂起该线程, 而不使其破坏位于该地址空间中的其它重要存储器区, 如另一线程堆栈。这不仅在线程之间, 还在同一地址空间之间增加了存储器保护。

然而这种想法又遇到了另外一个问题, 当 ARM 处理器响应异常事件时, 程序指针将要跳转到一个确定的位置, 假设发生了 IRQ 中断, PC 将指向 0x18(如果为高端启动, 则相应指向 0xffff_0018 处), 而此时 0x18 处仍为非易失性存储器所占据的位置, 则程序的执行还是有一部分要在 FLASH 或者 ROM 中来执行的。那么我们可不可以使程序完全都 SDRAM 中运行那? 答案是肯定的, 这就引入了 MMU, 利用 MMU, 可把 SDRAM 的地址完全映射到 0x0 起始的一片连续地址空间, 而把原来占据这片空间的 FLASH 或者 ROM 映射到其它不相冲突的存储空间位置。例如, FLASH 的地址从 0x0000_0000—0x00ff_ffff, 而 SDRAM 的地址范围是 0x3000_0000—0x31ff_ffff, 则可把 SDRAM 地址映射为 0x0000_0000—0x1fff_ffff 而 FLASH 的地址可以映射到 0x9000_0000—0x90ff_ffff (此处地址空间为空闲, 未被占用)。映射完成后, 如果处理器发生异常, 假设依然为 IRQ 中断, PC 指针指向 0x18 处的地址, 而这个时候 PC 实际上是从位于物理地址的 0x3000_0018 处读取指令。通过 MMU 的映射, 则可实现程序完全运行在 SDRAM 之中。

(1) 使用 DRAM 作为大容量存储器时, 如果 DRAM 的物理地址不连续, 这将给程序的编写调试造成极大不便, 而适当配置 MMU 可将其转换成虚拟地址连续的空间。

(2) ARM 内核的中断向量表要求放在 0 地址, 对于 ROM 在 0 地址的情况, 无法调试中断服务程序, 所以在调试阶段有必要将可读写的存储器空间映射到 0 地址。

(3) 系统的某些地址段是不允许被访问的, 否则会产生不可预料的后果, 为了避免这类错误, 可以通过 MMU 匹配表的设置将这些地址段设为用户不可存取类型。

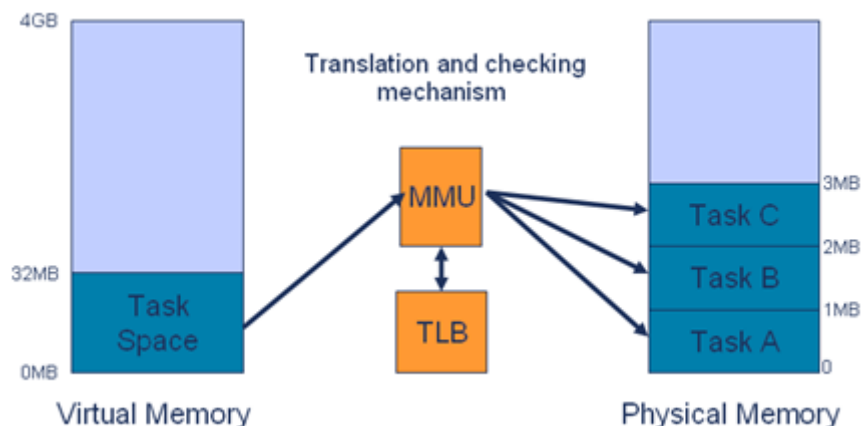
ARM 出品的 CPU, MMU 作为一个协处理器存在。ARM 结构的 MMU 要是有的话, 一定是编号为 15 的协处理器, 可以提供 32BIT 共 4G 的地址空间。



14.3 MMU 属性

1、 虚拟地址到物理地址映射

ARM 处理器产生的地址叫虚拟地址，MMU 允许把这个虚拟地址映射到一个不同的物理地址去。这个物理地址表示了被访问的主存储器的位置。



2、 存储器访问权限(permissions)

这些控制对存储器区域的不可访问权限、只读权限、读写权限。当访问不可访问权限的存储器时，会有一个存储器异常通知 ARM 处理器。

查找整个转换表的过程叫转换表遍历。它由硬件制进行，并需要大量的执行时间（至少一个存储器访问，通常是两个）。为了减少存储器访问的平均消耗，转换表遍历结果被高速缓存在一个或多个叫作 Translation Lookaside Buffers(TLBs)的结构中。通常在 ARM 的实现中每个内存接口有一个 TLB。当存储器中的转换表被改变或选中了不同的转换表(通过写 CP15 的寄存器 2)，先前高速缓存的转换表遍历结果将不再有效。MMU 结构提供了刷新 TLB 的操作。

MMU 结构也允许特定的转换表遍历结果被锁定在一个 TLB 中，这就保证了对相关的存储器区域的访问绝不会导致转换表遍历，这也对那些把指令和数据锁定在高速缓存中的实时代码有相同的好处。

3、 存储器访问的顺序

当 ARM 要访问存储器时，MMU 先查找 TLB 中的虚拟地址表，如果 ARM 的结构支持分开的地址 TLB 和指令 TLB，那么它用：取指令使用指令 TLB，其它的所有访问类别用数据 TLB。如果 TLB 中没有虚拟地址的入口，则转换表遍历硬件从存在主存储器中的转换表中获取转换和访问权限，一旦取到，这些信息将被放在 TLB 中，它会放在一个没有使用的入口处或覆盖一个已有的入口。关于转换表的信息和转换表遍历的实现参见转换过程一节。一旦为存储器访问的 TLB 的入口被拿到，这些信息将被用于：1. C（高速缓存）和 B（缓冲）位被用来控制高速缓存和写缓冲，并决定是否高速缓存。（如果系统中没有高速缓存和写缓冲，则对应的位将被忽略）2. 访问权限和域位用来控制访问是否被允许。如果不允许，则 MMU 将向 ARM 处理器发送一个存储器异常；否则访问将被允许进行。访问权限、域和异常几节有详细描述。3. 对没有高速缓存的系统（包括在没有高速缓存系统中的所有存储器访问），物理地址将被用作主存储器访问的地址。对有高速缓存的系统，在高速缓存没有选中的情况下，物理地址将被用行取(line fetch)的地址。



如果选中了高速缓存，则物理地址将被忽略。5.一级页表转换过程：MMU 支持基于节或页的存储器访问：节（Section）构成 1MB 的存储器块，支持 3 中不同的页尺寸：

微页（Tiny page）构成 1KB 的存储器块

小页（Small page）构成 4KB 的存储器块

大页（Large page）构成 64KB 的存储器块

节和大页是支持允许只用一个 TLB 入口去映射大的存储器区间。小页和大页有附加的访问控制：小页分成 1KB 的子页，和大页分成 16KB 的子页。微页没有子页，对微页的访问控制是对整个页。存在主存储器内的转换表有两个级别：

第一级表 存储节转换表和指向第二级表的指针

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
Fault																															0	0				
Level 2 page table	Level 2 Descriptor Base Address																						Domain Selector		1			0	1							
Section	Section Base Address										AP		Domain Selector		1	C	B														1	0				
	Fine Level 2 Descriptor Base Address																					Domain Selector		1											1	1

第二级表 存储大页和小页的转换表。一种类型的第二级表存储微页转换表。

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Fault																															0	0
Large page	Large Page Base Address																AP3	AP2	AP1	AP0	C	B	0	1								
Small Page	Small Page Base Address																AP3	AP2	AP1	AP0	C	B	1	0								
	Tiny Page - deprecated																			AP	C	B	1	1								

大页=64KB 小页=4KB

MMU 把 CPU 产生的虚拟地址转换成物理地址去访问外部存储器，同时继承并检查访问权限。地址转换有四条路径。路径的选取由这个地址是被标记成节映射访问还是页映射访问确定。页映射访问可以是 大、小和微页的访问。然而，转换过程总是由下面所描述的那样由第一级表的获取开始。节映射的访问只需要读取第一级表，页映射的访问还需要读取第二级表。

1、转换表基址

当片上（on-chip）的 TLB 中不包含被要求的虚拟地址的入口时，转换过程被启动。转换表基址寄存器（CP15 的寄存器 2）保存着第一级转换表基址的物理地址。只有 bits[31:14]有效，bits[13:0]应该是零（SBZ）。所以第一级表必须在 16KB 的边界。

2、取第一级表

转换表基址寄存器（即 cp15 协处理器的 C2 寄存器）的 bits[31:14]与虚拟地址（就是在程序中给定的存储器的虚拟地址，在程序中用的地址）的 bits[31:20]和两个 0 位[bit0:1]连接形成 32 为物理地址，如图 3-2。这个地址选择了一个四字节的转换表入口，它是第一级描述符或是指向第二级页表的指针。

3、第一级描述符



第一级表的每个入口是一个描述它所关联的 1MB 虚拟地址是如何映射的描述符。见一级页表图，根据 bits[1:0]的组合，有四种可能：

如果 bits[1:0]==0b00，所关联的地址没有被映射，试图访问他们将产生一个转换错（fault）。因为他们被硬件忽略，所以软件可以利用这样的描述符的 bits[31:2]做自己的用途。推荐为描述符继续保持正确的访问权限。

如果 bits[1:0]==0b10，这个入口是它所关联地址的节描述符。见节描述符和转换节参考中的细节。

如果 bits[0]==1，这个入口给出粗糙第二级表（bit[1]==0），或精细第二级表（bit[1]==1）。每一种类型的表描述了它所关联的 1MB 存储区域的映射。粗糙第二级表较小，每个表 1KB，每个精细第二级表 4KB。然而粗糙第二级表只能映射大页和小页，精细第二级表可以映射大页、小页和微页。

位说明	说 明
Bit[1:0]:	一级描述符类型标识。00：无效；01：粗页；10：段；11：细页由于没有使用复杂的操作系统，所以此为为 10
C、B:	该一级描述符对应的存储空间的 cache 和 Write Buffer 特性控制位，如表 4-4 所示
U:	由用户定义
Domain:	指明该存储空间所属的域号 0~15；见前面“MMU 中的域”说明
P:	保护标志位；只有 ARMv5 或以上版本处理器有定义
AP:	访问权限控制位，详见前面“ARM 处理器的存储域”说明
TEX:	在 ARMv5 以上版本处理器中有定义，如果 TEX 等于 1，表示系统支持写时分配 cache
粗粒度二级页表基址:	如果 bit[1:0]=01，那么该存储空间是二级分大页管理，1 页=4KB，粗粒度二级页表基址表示的是页描述符表的基址，1KB 对齐，一张页描述符表占 1KB 空间，256 个页描述符条目，描述了 1MB 空间的映射关系以及访问权限和域控制属性
段基址:	如果 bit[1:0]=10，那么该存储空间是分段管理，该段描述符号描述了 1MB 存储空间的映射关系以及访问权限和域控制属性，段基址就是该存储空间的物理基地址，1MB 对齐
细粒度二级页表基址:	如果 bit[1:0]=11，那么该存储空间是二级分小页管理，1 页=1KB，细粒度二级页表基址表示的是页描述符表的基址，4KB 对齐，一张页描述符表占 4KB 空间，1 024 个页描述符条目，描述了 1MB 空间的映射关系以及访问权限和域控制属性

section base address 为段基地址，sbz 意思是 should be zero，ap 是读写权限控制位，domain 为域，c,b 位介绍如下：

(1) 下图为段描述符（section）即一级描述符中的 bit[11:10]位

GCd	GBd	Data Access Mode
0	0	non cacheable, non bufferable
0	1	non cacheable, bufferable
1	0	WT, Write Through data cache
1	1	WB, Write Back data cache

(2) 下图为域描述符（Domains）即一级描述符中的 bit[5:8]位



Domains

- MMU accesses are primarily controlled by domains
- All defined regions of memory have an associated domain
 - Domains are defined as 2-bit access field
 - 16 domains can be defined
- Domains currently allow 3 states
 - **Client** - obey access permissions in the section or page descriptor
 - **Manager** - ignore access permissions in the section or page descriptor (so no fault can be generated)
 - **No access** - any access will generate a domain fault
- Domain access fields can be modified by a single coprocessor write

31

0

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
-----	-----	-----	-----	-----	-----	----	----	----	----	----	----	----	----	----	----

Value	Meaning	Notes
00	No Access	Any access will generate a domain fault.
01	Client	Accesses are checked against the access permission bits in the section or page descriptor.
10	Reserved	Reserved. Currently behaves like the no access mode.
11	Manager	Accesses are <i>not</i> checked against the access permission bits so a permission fault cannot be generated.

00: 当前级别下, 该内存区域不允许被访问, 任何的访问都会引起一个 domain fault

01: 当前级别下, 该内存区域的访问必须配合该内存区域的段描述符中 AP 位进行权限检查

10: 保留状态 (我们最好不要填写该值, 以免引起不能确定的问题)

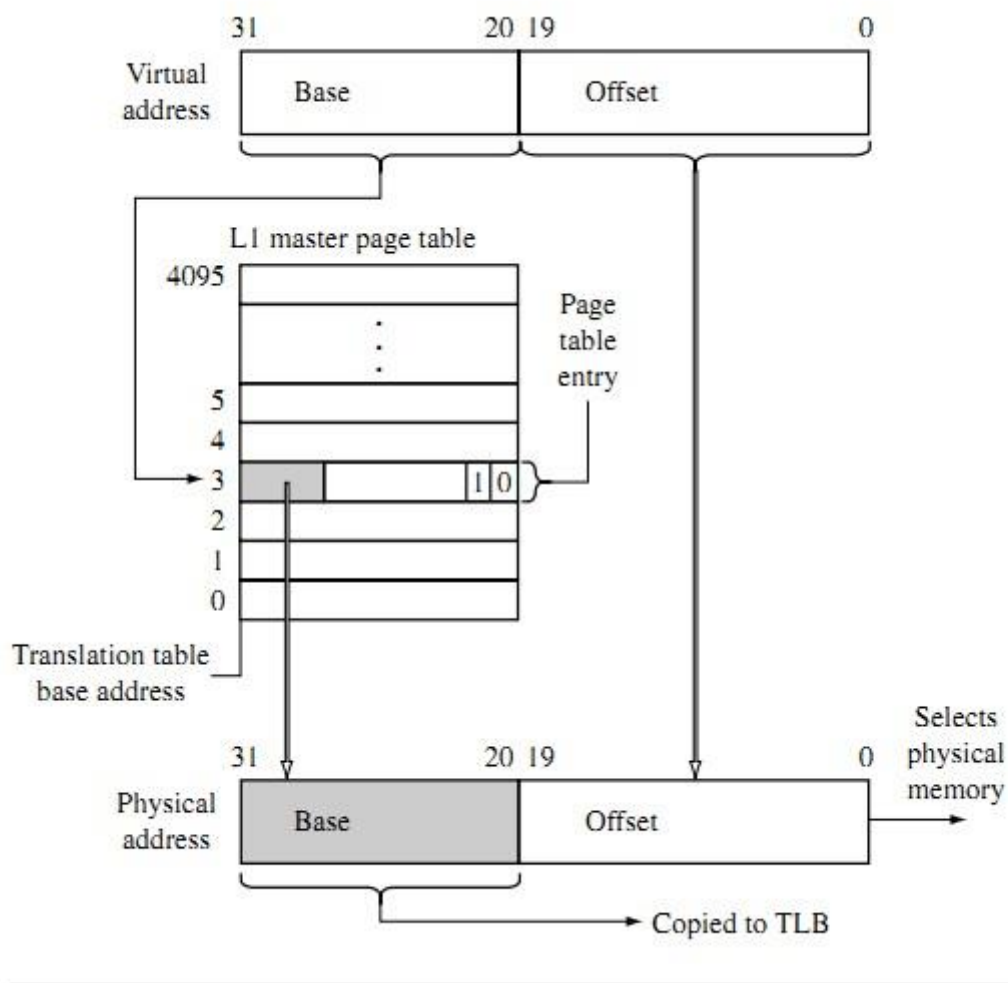
11: 当前级别下, 对该内存区域的访问都不进行权限检查。

(3) 下图为域 AP 述符 (AP) 即一级描述符中的 bit[1:0]位

AP	S	R	Supervisor Permissions	User Permissions	Notes
00	0	0	No access	No access	Any access generates a permission fault
00	1	0	Read only	No access	Supervisor read only permitted
00	0	1	Read only	Read only	Any write generates a permission fault
00	1	1	Reserved		
01	x	x	Read/write	No access	Access allowed only in supervisor mode
10	x	x	Read/write	Read only	Writes in user mode cause permission fault
11	x	x	Read/write	Read/write	All access types permitted in both modes.
xx	1	1	Reserved		



一级页表转换图：



14.4 MMU 实验

14.4.1 实验目的

- 掌握 mmu 的原理，实现物理地址和虚拟地址的映射

14.4.2 实验原理

实现一个一级页表映射。之前的 GPIO 实验实现了点亮 LED 的操作，当时 MMU 没有打开，操作的是寄存器的物理地址。本次实验需要打开 MMU，然后设定一段虚拟地址空间，然后编程将这段虚拟地址空间映射到 GPIO 控制器的物理地址空间。使用一级页表映射。最后通过操作这段虚拟地址控制 LED 亮灭。

(1) 硬件原理图



如图所示，LED2~LED5 分别与 GPX2_7、GPX1_0、GPF3_4、GPF3_5 相连，通过 GPX2_7、GPX1_0、GPF3_4、GPF3_5 引脚的高低电平来控制三极管的导通性，从而控制 LED 的亮灭。

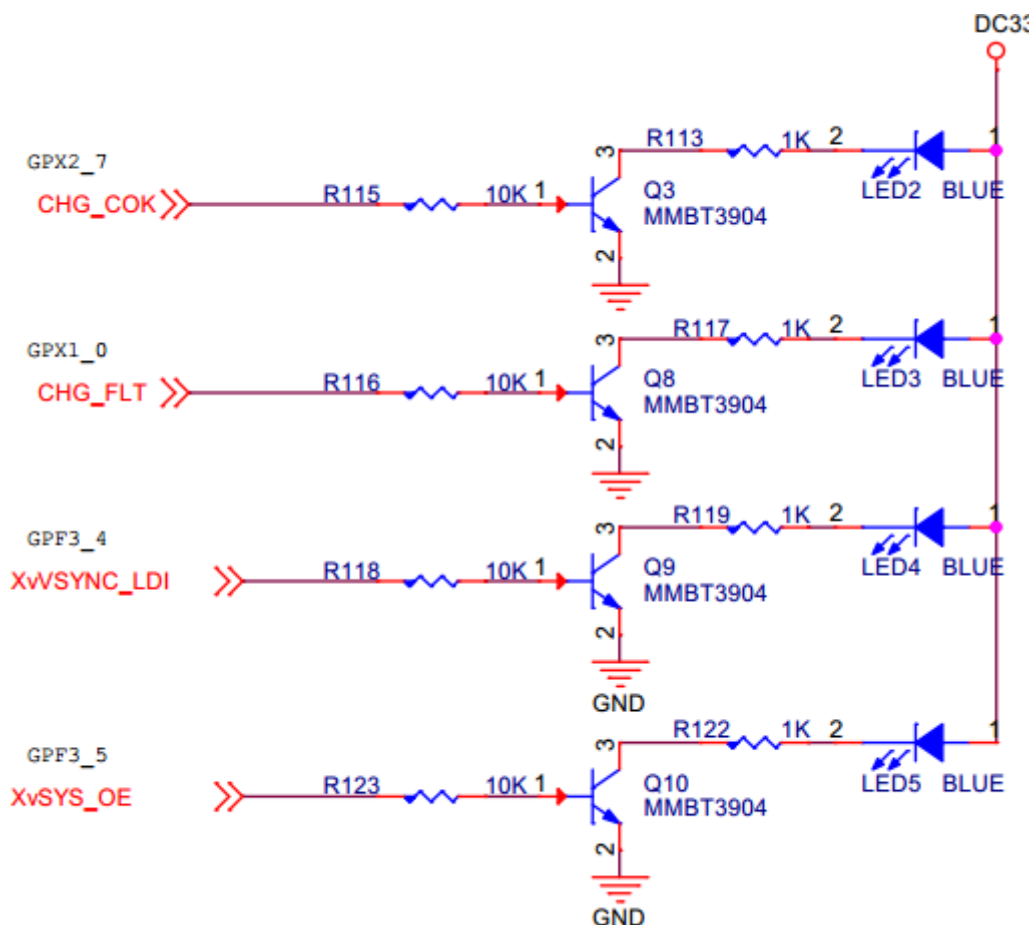


图 LED 接线原理图

根据三极管的特性，当这几个引脚输出高电平时，集电极和发射极导通，发光二极管点亮；反之，发光二极管熄灭。

通过控制 GPX1CON、GPX2CON、GPF3CON 和 GPX1DAT 来控制 GPX2_3 和 GPF3_4 对应的 LED。

6.2.3.198 GPX1CON

- Base Address: 0x1100_0000
- Address = Base Address + 0x0C20, Reset Value = 0x0000_0000

GPX1CON[0]	[3:0]	RW	0x0 = Input 0x1 = Output 0x2 = Reserved 0x3 = KP_COL[0] 0x4 = Reserved 0x5 = ALV_DBG[4] 0x6 to 0xE = Reserved 0xF = WAKEUP_INT1[0]	0x00
------------	-------	----	---	------



6.2.3.199 GPX1DAT

- Base Address: 0x1100_0000
- Address = Base Address + 0x0C24, Reset Value = 0x00

Name	Bit	Type	Description	Reset Value
GPX1DAT[7:0]	[7:0]	RWX	When you configure port as input port then corresponding bit is pin state. When configuring as output port then pin state should be same as corresponding bit. When the port is configured as functional pin, the undefined value will be read.	0x00

14.4.3 实验内容

编写一个程序，控制 LED3 循环闪速。根据原理图说明，需要控制 GPX1_0 管脚输出高低电平。

(1) 编写 MMU 控制函数

在 start.s 初始化汇编文件中实现几个函数：

mmu_disable: 关闭 mmu 功能；

mmu_enable: 使能 mmu 功能；

mmu_set: 设置 MMU 一级页表的地址 0x41000000(通过 CP15 的 C2 寄存器)、设置属性、开启 MMU。

```

1 mmu_disable:
2     mrc      p15,0,r0,c1,c0,0
3     bic      r0,r0,#1
4     mcr      p15,0,r0,c1,c0,0
5     mov      pc,lr
6 mmu_enable:
7     mrc      p15,0,r0,c1,c0,0
8     orr      r0,r0,#1
9     mcr      p15,0,r0,c1,c0,0
10    mov      pc,lr
11 .global mmu_set
12 mmu_set:
13     ldr      r0,=0x41000000
14     mcr      p15,0,r0,c2,c0,0
15     ldr      r5,=0xffffffff
16     @ define the access permissions for each one of the 16 domains
17     mcr      p15,0,r5,c3,c0,0
18     @ Manager. Accesses are not check
19
20     mrc      p15,0,r0,c1,c0,0
21     orr      r0,r0,#1
22     mcr      p15,0,r0,c1,c0,0
23     mov      pc,lr
    
```

(2) 编写一个 MMU 虚拟地址一级映射函数



Vaddrstart: 虚拟地址的开始地址

Vaddrend: 虚拟地址的结束地址

Paddrstart: 物理地址的开始地址

Attr: 一级页表的属性部分, 参见上文一级页表的说明。

```

1 void mmu_setmtt(unsigned int vaddrstart, unsigned int vaddrend,
2                 unsigned int paddrstart, int attr)
3 {
4     unsigned int *ptt;
5     int i,nsection,tt;
6     ptt = (unsigned int *)0x41000000+(vaddrstart >> 20);
7     //计算出虚拟地址一级页表项的地址
8     nsection = (vaddrend >> 20) - (vaddrstart >> 20); //计算映射的长度
9     for(i = 0; i< nsection; i++)
10        *ptt++ = attr | (((paddrstart >> 20) + i) << 20); //填充页表项
11 }

```

(3) 定义虚拟地址

在头文件 exynos_4412.h 中定义物理地址和虚拟地址结构体

```

1 /* GPX1 */
2 typedef struct {
3     unsigned int CON;
4     unsigned int DAT;
5     unsigned int PUD;
6     unsigned int DRV;
7 }gpx1;
8 #define GPX1 (* (volatile gpx1 *)0x11000C20 )
9
10 /* VGXP1 */
11
12 #define VGXP1 (* (volatile gpx1 *)0x91000C20 )

```

(4) 编写主函数

编写主函数实现将虚拟地址: 0x91000000~0x92000000, 映射到物理地址: 0x11000000~0x11000000。

然后操作虚拟地址来控制 LED 闪速。

```

1 int main()
2 {
3     volatile int count;
4     uart_init();
5     mmu_setmtt(0x0,0xff000000,0x0,0xc12); //全局的 1:1 的映射
6     mmu_setmtt(0x91000000,0x92000000,0x11000000,0xc12);

```



```
7   mmu_set(); //注意，调用这个函数时要保证 ARM 处于特权模式
8   VGPX1.CON = (VGPX1.CON & ~(0xf)) | 1; //GPX1_0:output, LED3
9   while(1){
10      //Turn on LED3
11      VGPX1.DAT |= 0x1;
12      mydelay_ms(500);
13      //Turn off LED3
14      VGPX1.DAT &= ~0x1;
15      mydelay_ms(500);
16  }
17  /*
18  //因为之前做过一次全局的 1:1 的映射，所以 mmu 开启情况下，访问之前的实地址还是可以正常访问。
19  GPX1.CON = (GPX1.CON & ~(0xf)) | 1; //GPX1_0:output, LED3
20  while(1){
21      //Turn on LED3
22      GPX1.DAT |= 0x1;
23      mydelay_ms(500);
24      //Turn off LED3
25      GPX1.DAT &= ~0x1;
26      mydelay_ms(500);
27  }
28  */
29  }
30
```

14.4.4 实验步骤

1、 导入工程源码

请参考第 1 章节的 ARM 开发环境搭建部分。

光盘实验源码路径：**【华清远见-CORTEXA9 资料\程序源码\ARM 裸机实验源码\17-mmμ】**

2、 连接好开发板及 FS-JTAG 仿真器，并且连接好配线。

方法请参考第 1 章节的开发环境搭建的硬件连接部分按对应型号连接。

3、 配置好串口终端

方法请参考第 1 章节的 ARM 开发环境搭建部分。

4、 仿真运行程序

方法请参考第 1 章节的 ARM 开发环境搭建部分。



14.4.5 实验现象

Debug 调试点击运行按钮 ，LED3 出现闪速现象。

华清远见
dev.hqyj.com