

UP-Star2410 用户手册





熟悉EWARM集成开发环境	3
H-JTAG的安装与使用	3
EWARM集成开发环境的使用	9
嵌入式系统硬件驱动基础开发案例	49
实验一 ARM的串行口实验	49
实验二 LED点灯实验	60
实验三 按键中断实验	63
实验四 触摸屏驱动实验	66
实验五 LCD的驱动控制实验	75
实验六 uCOS-II在ARM微处理器上的移植及编译	89
附录一 ARM汇编指令集	100
1 ARM指令集	100
2 ARM汇编器所支持的伪指令	116
附录二 嵌入式系统应用编程API函数	127
1. 显示部分 DISPLAY.H	127
2. 操作系统的消息相关函数 OSMESSAGE.H	134
3. 控件的相关函数CONTROL.H	135
4. 文件相关函数(与标准C的文件操作相同)	142
5. 双向链表相关函数LIST.H	144
6. 触摸屏相关函数TCHSCR.H	144
7. 键盘相关函数 KEYBOARD.H	145
8. 液晶显示相关函数 LCD320.H	145
9. 串行口相关函数UHAL.H	146
10. 字符串相关函数USTRING.H	146
11. 系统图形相关函数 FIGURE.H	147
12. 系统启动时相关函数 LOADFILE.H	148
13. 系统附加任务相关函数 OSAddTask.H	149
附录三 IAR EMBEDDED WORKBENCH 的安装	149
附录四 演示实验演示手册	162

熟悉 EWARD 集成开发环境

H-JTAG 的安装与使用

a) H-JTAG 简介

说明：关于部分 H-JTAG 的说明摘 H-Jtag 说明手册，请参考其官方网站<http://www.hjtag.com>

当前 ARM 的学习与开发非常流行，由于 ARM 的软件开发相对以前单片机而言更加复杂，硬件上的考虑也比较多，因此选择一个好的调试方法将可以使得开发的除错过程变得更加直接和简单。现在市面上有很多可用于 ARM 调试的仿真器出售，然而其价格往往都比较贵。这些仿真器一般都有其专用的软件和硬件，在速度和 flash 编程等方面有各自的优势。然而对初学者而言，这些仿真器的成本都太高。而简易仿真器的出现，使得大家可以使用甚至自制 ARM 仿真器硬件。

有了调试器的硬件，还要加上调试代理软件，作为中介，将调试器前端软件（比如 AXD）的调试信息与目标板上的目标芯片交互，才能最终完成仿真的任务。目前，可以免费使用的简易 ARM 仿真器的代理软件很多，差别也比较大，主要表现在易用程度，目标器件支持，调试速度等方面。H-JTAG 作为近来新推出的简易 ARM 仿真器调试代理，其支持器件比较多，支持的调试器前端软件也比较多，特别是支持 keil，其调试速度也很有优势。

H-JTAG 是由 twentyone 推出的一款免费调试代理软件。官方主页为：<http://www.hjtag.com/>

目前我们使用的版本为 V0.6.1，支持下列特性（更新的版本请到 H-JTAG 网站下载试用）：

- 支持 RDI 1.5.0 与 1.5.1;
- 支持 ARM7 与 ARM9（包括 ARM9E-S 与 ARM9EJ-S）;
- 支持 thumb 与 arm 指令集;
- 支持 little-endian 与 big-endian;
- 支持 semihosting;
- 支持 wiggler, sjf-jtag 以及用户自定义的简易调试器硬件接口;
- 支持 WINDOWS 9.X/NT/2000/XP;
- 支持 flash 器件的编程

注：本开发板所用的 JTAG 板即为 SJF-JTAG

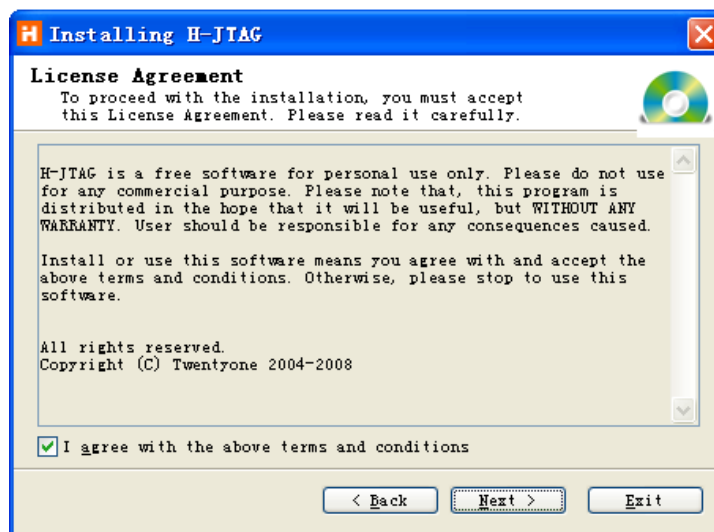
b) 安装 H-JTAG

H-JTAG 安装文件位于光盘的“Windows 平台工具\H-JTAG”目录，双击运行，按照其提示安装即可。

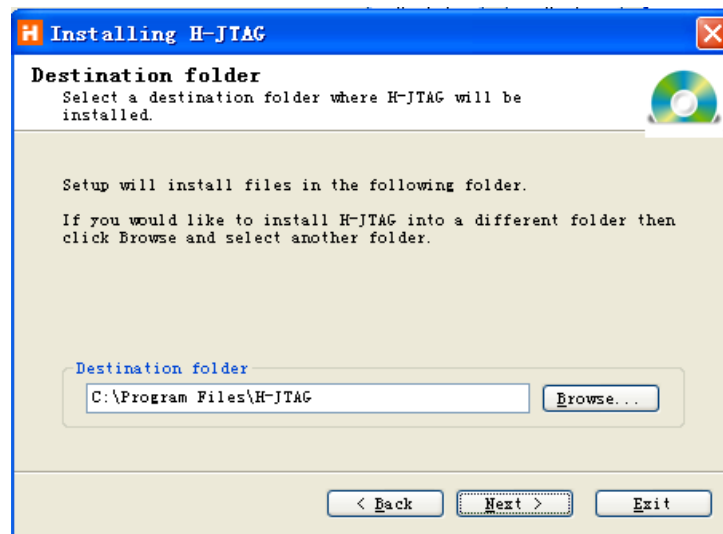
- ① 解压光盘光盘中 /windows 平台工具/H-JTAG V0.6.1.zip 双击 H-JTAG V0.6.1.EXE 进行安装



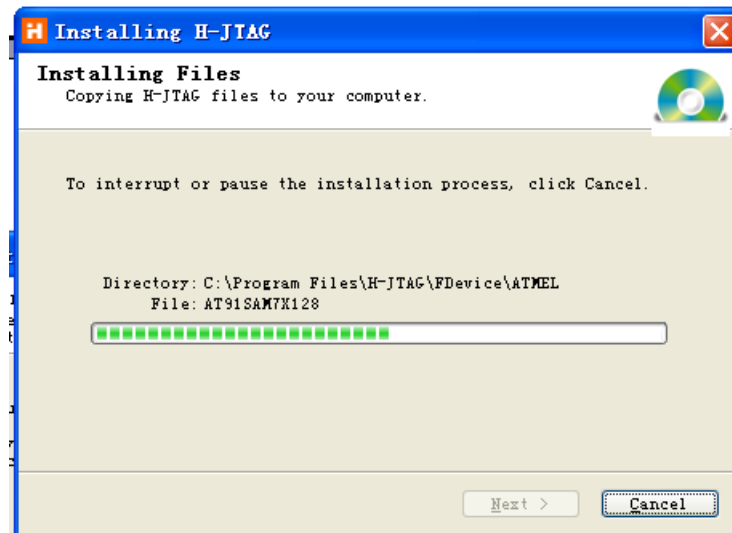
- ② 直接点击 Next >



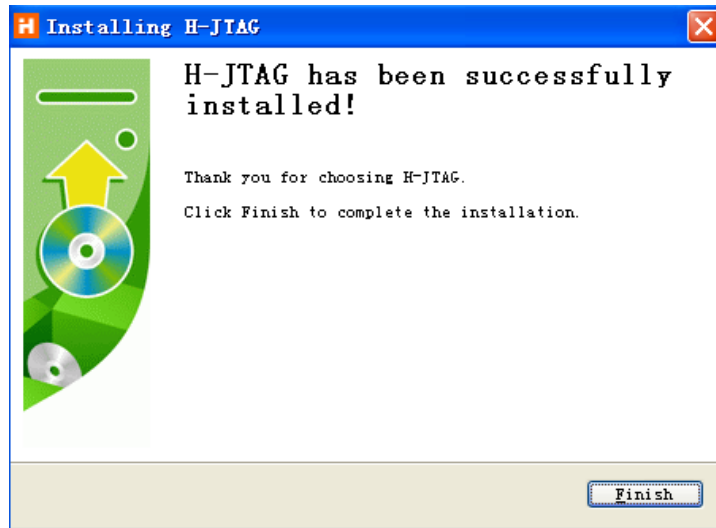
- ③ 在 I agree 前面勾选，后点击 Next >



④ 直接点击 Next >



⑤ 等待一段时间后

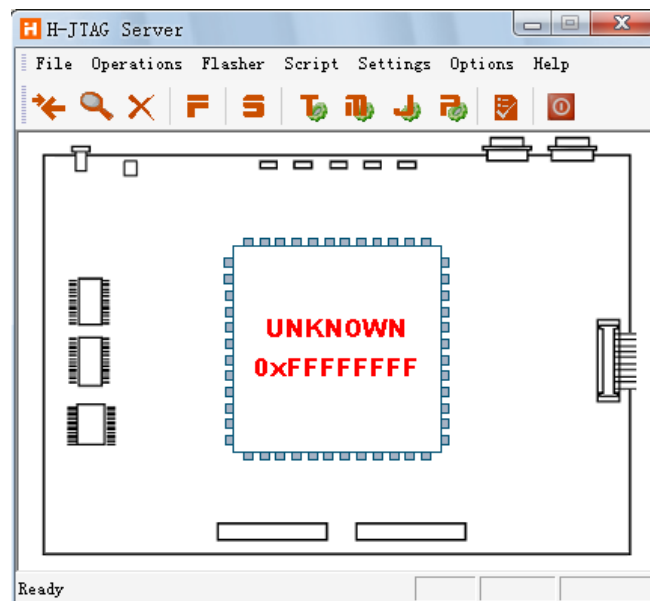


⑥ 点击 Finish 即完成了软件的安装

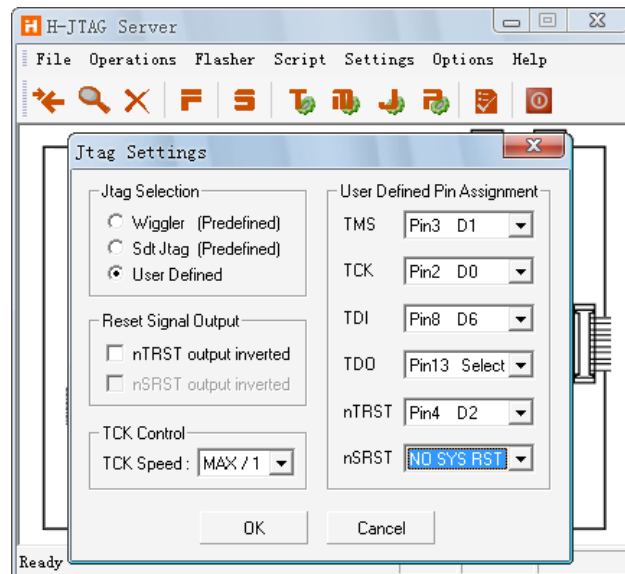
安装完毕，会在桌面生成H-JTAG H-Converter和H-Flasher 快捷方式，双击运行H-JTAG即可运行。

c) 设置 JTAG 端口

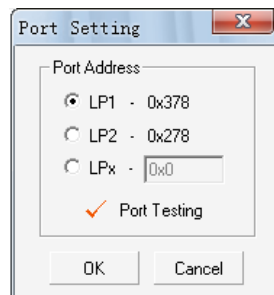
⑦ 双击运行 H-JTAG



⑧ 选择菜单项 Settings>Jtag Settings 修改选项配置如下图：



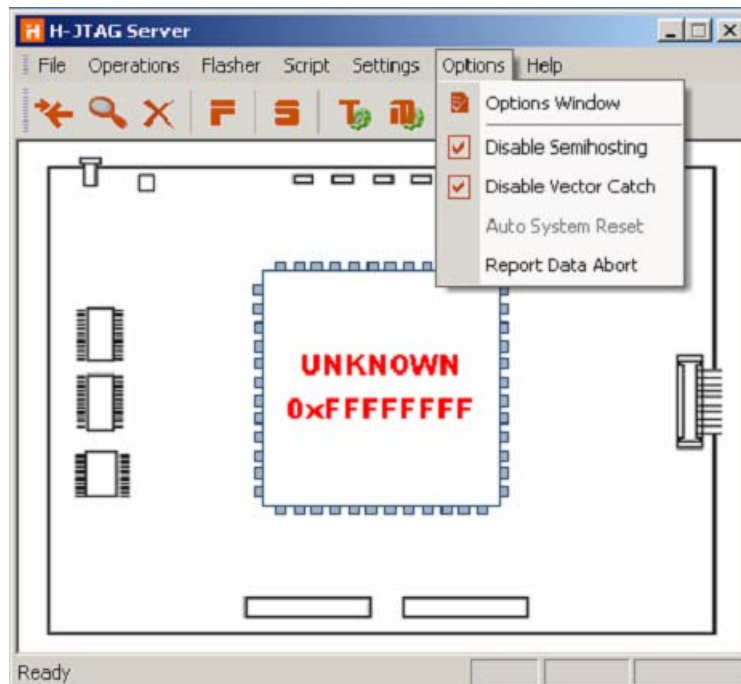
- ⑨ 选择菜单项 Settings>Port Settings 选择 ARM-JTAG 仿真器所用的并口号，点击 Port Testing 按钮，已确认该并口是否可用，否则请检查 BIOS 设置。



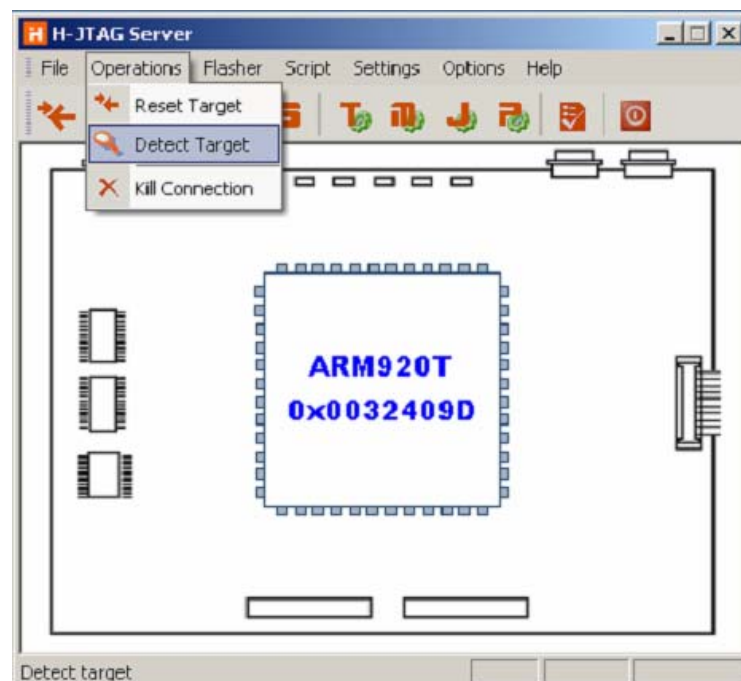
- ⑩ 正常应该弹出



- ⑪ 点击 Options 菜单，检查 Disable Semihosting 和 Disable Vector Catch 是否为选中状态，若没有选中应将其勾选。



- ⑫ 使用 UP-LINK 接通开发板和主机，开启电源。
- ⑬ 在 H-JTAG Server 窗口中选择 Operations> Detect Target 应该能够侦测目标板上的 ARM 7~10 内核如下图，如侦测不到请检测上述步骤。



d) 设置初始化脚本

e) 检测目标器件

使用开发板附带的 JTAG 小板连接开发板的JTAG 接口，并接上打开电源。点主菜单Operations->Detect Target，或者点工具栏相应的图标也可以，这时就可以看到已经检测到目标器件了。

提示：如果没有设置初始化脚本，也可以检测到CPU，但无法进行下面的单步调试。

EWARM 集成开发环境的使用

前 言

IAR Embedded Workbench for ARM 是 IAR Systems 公司为 ARM 微处理器开发的一个集成开发环境(下面简称 IAR EWARM)。比较其他的 ARM 开发环境, IAR EWARM 具有入门容易、使用方便和代码紧凑等特点。故在这里介绍给打算学习使用或正在使用 ARM 芯片的朋友们共同探讨。

IAR Systems公司目前推出的最新版本是IAR Embedded Workbench for ARM version 4.42, 并提供一个 32k代码限制学习版或 30 天时间限制的免费评估版, 可以到IAR公司的网站 www.iar.com/ewarm下载。

IAR EWARM 中包含一个全软件的模拟程序 (simulator)。用户不需要任何硬件支持就可以模拟各种 ARM 内核、外部设备甚至中断的软件运行环境。从中可以了解和评估 IAR EWARM 的功能和使用方法。

我们编译整理的这本快速用户指南采用评估版软件安装目录 C:\Program files\IAR System\Embedded workbench 4.0\ARM\tutor 下的教程为例, 一步一步介绍 IAR EWARM 的使用方法。该教程采用了两个 C 语言程序, tutor.c 和 utilities.c。它们不和任何特定的硬件关联, 所以介绍中的全部操作都是用模拟程序完成的。在以后的章节里, 我们将具体介绍 EWARM 软件及配套硬件工具、如何使用 EWARM 集成开发环境以及在 EWARM 下烧写 Flash 的方法。

如果用户希望在真实的目标板上进行代码运行和调试, 请到万利电子全国各直销点购买 IAR 的 JTAG 仿真器 J-Link。

附：EWARM 的学习步骤

- ① 下载安装 EWARM 32K 学习版软件;
- ② 进入 www.iar.com/ewarm -> Online Demos, 下载Flash格式的软件使用动画演示;

- ③ 以本入门手册结合软件使用的动画演示，进行软件使用的入门学习；
- ④ 在软件安装目录...\arm\src\examples 下，寻找感兴趣的芯片例程学习；
- ⑤ 可选项 1：购买 J-Link 仿真器和开发板，实现在硬件上的代码运行和调试；
- ⑥ 可选项 2：购买由北航出版社的《IAR EWARM 嵌入式系统编程与实践》一书，深入学习；
- ⑦ 学习“可选项 1 或 2”的随附光盘中《Converting ADS Projects to EWARM Projects》白皮书，实践如何移植一个 ADS 工程到 EWARM 格式的工程。

1、EWARM 集成开发环境及配套仿真器简介

IAR Embedded Workbench for ARM version 4.42 是一个针对 ARM 处理器的集成开发环境，包含项目管理器、编辑器、编译连接工具和支持 RTOS 的调试工具，在该环境下可以使用 C/C++ 和汇编语言方便地开发嵌入式应用程序。IAR EWARM 的主要模块如下：

- 项目管理器
- 功能强大的编辑器
- 高度优化的 IAR ARM C/C++ Compiler
- IAR ARM Assembler
- 1 个通用的 IAR XLINK Linker
- IAR XAR 和 XLIB 建库程序和 IAR DLIB C/C++ 运行库
- IAR C-SPY 调试器（先进的高级语言调试器）
- 命令行实用程序

以下介绍一下 EWARM 5.20 版本及其相关配套硬件的一些特点：

1. IAR EWAM 软件的特点

① EWARM 4.42 版基本特点

- 完善的 ARM 内核支持
 - 最新支持到 ARM11 及 Cortex M3 内核
 - 早已支持的其他 ARM 内核
 - ✓ ARM7 (ARM7TDMI, ARM7TDMI-S, ARM720T)
 - ✓ ARM9 (ARM9TDMI, ARM920T, ARM922T, ARM940T, ARM9E, ARM9E-S, ARM926EJ-S, ARM946E-S, ARM966E-S, ARM968E-S)
 - ✓ ARM10 (ARM10E, ARM1020E, ARM1022E, ARM1026EJ-S)

- ✓ XScale (XScale, XScale-IR7)
- 更加客户化地提供芯片级的支持
 - 完备的各厂商 ARM 处理器的 C/C++和汇编语言外设寄存器定义文件支持的芯片厂商有 Analog Devices、ARM、Atmel、Cirrus Logic、Freescale、Intel、NetSilicon、OKI、Philips、Samsung、Sharp、ST 和 TI 等
 - 支持 Analog Devices、Atmel、Freescale、OKI、Philips、ST 和 TI 等厂商的 ARM 处理器的 Flash Loader 程序
 - 软件集成了 400 余个代码例程，对应于各种不同的芯片，位于... \arm\src\examples 目录下
- 进一步改进了编译器速度优化，重写了的浮点运算库
- 对更多嵌入式操作系统的支持
 - 新增支持 OSEK 类操作系统的 OSEK Run-Time Interface (ORTI)
 - 新增支持 OSE Epsilon RTOS 的 Kernel Awareness 调试
 - 新增支持 embOS、SMX、NORTi 等的支持
- 调试器的增强功能
 - 对堆栈运行的监测功能
 - 配合 IAR J-Link 仿真器的新增功能
 - ✓ J-Link TCP/IP 服务器
 - ✓ 调试器和 IAR J-Link 仿真器协同配合，实现对 ARM 处理器的多核调试
 - 对 IAR J-Trace 仿真器提供全面的支持
 - 在 C-SPY 模拟器中可执行 Trace 的模拟
 - 支持同一芯片上多颗 Flash 的 Flash Loader 程序，以及通用的 Flash Loader 开发指南

② EWARM 软件在芯片级支持方面的特色

- 完备的各厂商 ARM 处理器的 C/C++和汇编语言外设寄存器定义文件
- 大量适合于嵌入式代码的编程语言扩展特性，包括存储器关键字，本征函数，中断函数，存储器映射 I/O 等
- 针对评估板的例程，包含 IAR、Analog Devices、Aiji System、ARM、Atmel、Cirrus Logic、Freescale、Keil、OKI、Olimex、Pasat、Philips、

Phytec、ST 和 TI 等厂家的开发板

- 支持 ARM 或 Thumb 模式下大至 4G 字节的应用程序
- 每个函数都能选择在 ARM 或 Thumb 模式下编译
- 可生成 VFP 向量浮点协处理器代码
- 支持 Analog Devices、Atmel、Freescale、OKI、Philips、ST 和 TI 等厂商的 ARM 处理器的 Flash Loader 程序
- 支持 ARM Angel Debug monitor

③ EWARM 编译器的软件特色

- 先进的通用编译器优化和针对特定处理器的速度优化及存储器优化功能
- 轻量运行库，用户可以根据需要自行配置，提供全部源代码
- 灵活的存储器控制，允许详细地为代码和数据分配地址
- 去除不需要的函数和变量
- C/C++变量和函数连接时全局类型检查
- 可选的校验和生成功能，用于运行时映象校验
- 自动将代码和数据放置到非连续的存储器区域
- 强大的可重定位宏汇编器，支持丰富的命令集和操作符

④ EWARM 调试器的软件特色

- 完全集成的源代码和反汇编程序调试器
- 非常细化的执行控制（函数调用级步进）
- 复杂的代码和数据断点
- 丰富的数据监视功能
- Locals, Watch, Auto, Live Watch 和 Quick Watch 等变量查看窗口
- 寄存器和存储器查看窗口
- 支持 STL 容器
- C/C++调用栈窗口，同时还可以显示将要进入的函数
- 双击调用链上的任何函数将更新编辑器、局部变量、寄存器、变量查看和反汇编窗口，以显示在该函数调用时的状态
- 跟踪功能，可以检查执行的历史记录。在跟踪窗口中移动时将更新编

辑器和反汇编窗口以显示合适的位置

- 控制台 I/O 仿真
- 中断和 I/O 模拟仿真
- 类似 C 语言的宏系统，可扩充调试器的功能
- 由主机执行的应用程序系统调用仿真
- 代码覆盖率和执行时间分析工具
- 通用的 Flash Loader 程序及开发指南
- 同时支持多颗 Flash 的 Flash Loader 程序
- 支持 OSEK Run-Time Interface (ORTI)
- 提供为调试器扩充第三方功能的软件开发包，如 RTOS 调试扩充和仿真器驱动扩充
- 命令行调试工具

⑤ IAR C-SPY 支持的调试方法

- IAR J-Link JTAG 接口（支持所有 ARM7 和 ARM9 核，通过 USB 或 TCP/IP 连接）
- IAR J-Trace JTAG 接口（支持所有 ARM7 和 ARM9 核，通过 USB 或 TCP/IP 连接）
- RDI 接口类的第三方仿真器（Abatron BDI1000 & BDI2000, EPI Majic, Ashling Opella, Aiji OpenICE, Signum JTAGjet, ARM Multi-ICE 等）
- Macraigor Wiggler, Raven, mpDemon 和 USBdemon 等调试接口
- EPI Jeeni 仿真器支持
- IAR 的 ROM-Monitor
- ARM 公司的 Angel ROM-Monitor（用于 Atmel 和 Cirrus Logic 的评估板）

⑥ IAR 对嵌入式实时操作系统的 Kernel Awareness 调试支持

操作系统	IAR EWARM 内置的插件	由第三方 RTOS 厂 商提供的插件
CMA-RX	X	
CMX-Tiny ⁺	X	

uC/OS-II		X
ThreadX	X	
RTXC Quadros		X
Fusion RTOS		X
OSEK (ORTI)	X	
ENEA OSE Epsilon	X	
MiSPO NORTi		X
Micro Digital SMX		X
Segger embOS	X	

每种 RTOS 插件都会在 C-SPY 中安装一批新的窗口，其中最重要的是任务或线程列表窗口，在此窗口中可以在指定的任务上设置断点和执行程序。其它不同的监测窗口可以显示 RTOS 内部数据结构的内容，例如定时器、队列、信号量、资源和邮箱等。

⑦ EWARM 图形化的集成开发环境的界面特色

- 分层次的工程组织
- 同一工作空间中允许存放多个工程
- 可停靠的窗口和多视图
- 源代码浏览
- 创建和维护库的工具
- 可以和源代码控制系统相集成
- 文本编辑器
 - 支持多字节字符（汉字）
 - 上下文相关的帮助系统
 - 根据句法着色
 - 无限制的 undo/redo
 - 搜寻、替换和增量搜寻
 - Go to
 - 书签
 - 错误标签：查阅前一个/下一个
 - 自动括号配对
 - 智能缩排
 - 类似网页浏览器的前向/后向源码查阅

- 代码断点的设置/清除/使能/禁止

➤ 命令行编译连接工具

⑧ EWARM 的编程语言和标准

- 遵循 ISO/ANSI C94（带有一些从 C99 标准中挑选的特性）标准的 C 编程语言
- 嵌入式 C++ 扩展，支持模板、多重继承和虚拟继承、名字空间以及其它不增加执行时间或存储器开销的 C++ 特性。完整的嵌入式 C++ 库还包含字符串、流等特性，以及标准模板库（STL）
- IEEE-754 浮点运算规则
- MISRA C 检查器
- 支持大量工业标准的调试和映象文件格式（如 ELF/DWARF），与大多数常见的调试器和仿真器兼容

⑨ 用户帮助

- 完备的例程和工程模板。
- 上下文相关的联机帮助系统，带有库函数查阅功能
- 印刷好的用户指南，带有详细的 step-by-step 教程
- 友好、详尽和精确的错误信息和警告信息

2. IAR J-Link 仿真器简介

IAR J-Link 是 IAR 为支持仿真 ARM 内核芯片推出的 JTAG 方式仿真器。配合 IAR EWARM 集成开发环境支持所有 ARM7/ARM9/Cortex M3 内核芯片的仿真，无需安装任何驱动程序，与 EWARM 集成开发环境无缝连接，操作方便、连接方便、简单易学是学习开发 ARM 最好最实用的开发工具。

同时，最近的有关权威测试显示，J-Link 目前是同类产品中下载调试速度最快的 J-Tag 仿真器：

公司	产品	通讯接口	支持内核	下载速度	对开发板供电功能	备注
Macraigor	Wiggler	LPT	ARM7/9	16 KB/秒	无	即并口仿真头
Keil	U-Link	USB	ARM7	28 KB/秒	无	
IAR	J-Link	USB 2.0	ARM7/9	800 KB/秒	有	

① J-Link ARM 主要特点

- IAR EWARM 集成开发环境无缝连接的 JTAG 仿真器
- 支持所有 ARM7/ARM9/Cortex M3 内核的芯片，包括 Thumb 模式
- 免费升级至支持 ARM11、Xscale
- 支持 SWD 接口调试
- 下载速度高达 800 kB/s
- 最高 JTAG 速度 12 MHz
- 目标板电压范围 1.2V - 3.3V
- 自动速度识别功能
- 监测所有 JTAG 信号和目标板电压
- 完全即插即用
- 使用 USB 电源
- 带 USB 连接线和 20 芯扁平电缆
- 支持多 JTAG 器件串行连接
- 标准 20 芯 JTAG 仿真插头
- 选配 14 芯 JTAG 仿真插头
- 选配用于 5V 目标板的适配器
- 带 J-Link TCP/IP server，允许通过 TCP/ IP 网络使用 J-Link

② IAR J-Link 的物理连接

J-LINK 一端通过 USB 口与 PC 连接，另一端通过标准 20 芯 JTAG 插头与目标板连接。建议首先连接 J-LINK 到 PC, 再连接 J-LINK 到目标系统，最后给目标系统供电(如果目标系统为独立供电、而非由 J-TAG 口供电的情况)。

③ IAR J-Link 主要技术指标

功耗	吸取 USB 供电电力 < 50 mA
通讯方式	USB 2.0 全速
目标板接口	20 芯 JTAG 口 (14 芯 JTAG 口选件)
J-Link 和 ARM 间串行传输速率	最高 12 MHz
支持目标电压	1.2 - 3.3 V (5V 适配头选件)
工作温度	+5 C - +60 C
储存温度	-20 C - +65 C
相对湿度 (无冷凝水)	< 90% RH
体积	100mm x 53mm x 27mm

重量（不含电缆）	70 克
电磁兼容性（EMC）	EN 55022, EN55024

④ 目标板 5V 电源适配器选件

当目标系统为 5V 电源系统时，必须使用 J-LINK 提供的 5V 电源适配器选件。对于 1.2V~3.3V 电源系统，可以直接使用 J-Link。使用时将适配器的 20 芯 IDC 插头插进 J-Link 的 20 芯插座，再将连接目标的 20 芯扁平电缆插进适配器的插座。



5V 适配器选件由目标供电 (3.3V~ 5V)，电流<20mA，有一个 LED 指示电源状态。

⑤ JTAG 插头定义

J-Link 的 JTAG 20 芯的 IDC 插头与 ARM 公司的仿真器插头定义兼容，有关定义如下：

引脚	名称	方向	功 能 描 述
1	VTref	Input	目标系统参考电压。 用于检查目标系统是否供电，并产生一个逻辑电平送给 J-Link 内部比较器。检测结果用来控制输出给目标的逻辑电平幅度。此引脚通常与目标的 Vdd 联，中间不允许串接电阻。
2	Vsupply	NC	J-Link 不用此引脚，在目标系统中连接到 Vdd 或开路。
3	nTRST	Output	JTAG 复位，J-Link 输出给目标的 Reset 信号。 通常连接到目标 CPU 的 nTRST 引脚。目标板上应将此脚上拉到高电位，避免意外复位。
4	GND	-	公共地。
5	TDI	Output	J-Link 输出给目标 CPU 的 JTAG 数据。 通常与目标 CPU 的 TDI 引脚相连。建议在目标板上将此脚上拉到 Vdd。
6	GND	-	公共地。
7	TMS	Output	J-Link 输出给目标 CPU 的 JTAG 模式设置信号。 通常与目标 CPU 的 TMS 引脚相连。建议在目标板上将此脚上拉。
8	GND	-	公共地。
9	TCK	Output	J-Link 输出给目标 CPU 的 JTAG 时钟信号。 通常与目标 CPU 的 TCK 引脚相连。建议在目标板上将此脚上拉。
10	GND	-	公共地。
11	RTCK	Intput	目标 CPU 提供给 J-Link 的测试时钟信号。 有些目标要求 JTAG 的输入与其内部时钟同步。J-Link 利用此引脚的输入可动态地控制自己的 TCK 速率。 若不使用此功能，在目标板上将此脚接地。
12	GND	-	公共地。

13	TDO	Input	目标 CPU 返回给 J-Link 的数据信号。 通常与目标 CPU 的 TDO 引脚相连。
14	GND	-	公共地。
15	RESET	I/O	目标 CPU Reset 信号
16	GND	-	公共地。
17	DBGRO	NC	J-Link 不用此引脚，在目标系统中将此引脚开路。
18	GND		公共地。
19	Vdd	Output	+3.3V 电源输出。
20	GND		公共地。

⑥ JTAG 速度

J-Link 有两种速度设定，即固定 JTAG 速度、自动 JTAG 速度，该功能选项位于 Project Options -> Debugger -> J-Link 设置页面中。

❖ 固定 JTAG 速度

目标被锁定在固定时钟速度。目标能执行的最大 JTAG 速度取决于目标自身。一般来讲，不带 JTAG 同步逻辑的 ARM 内核（如 ARM7-TDMI）能执行与 CPU 速度相当的 JTAG 速度。而带 JTAG 同步逻辑的 ARM 内核（例如 ARM7-TDMI-S，ARM946E-S，ARM966EJ-S）能执行相当 CPU 速度 1/6 的 JTAG 速度。JTAG 速度不应超过 10 MHz。

❖ 自动 JTAG 速度

由 TAP 控制器选择最大的 JTAG 速度。要注意，不带同步逻辑的 ARM 内核可能会工作不稳定。因为 CPU 内核时钟可能慢于最大 JTAG 速度。

2、在 EWARM 中生成一个新项目

EWARM 是按项目进行管理的，它提供了应用程序和库程序的项目模板。项目下面可以分级或分类管理源文件。允许为每个项目定义一个或多个编译连接（build）配置。在生成新项目之前，必须建立一个新的工作区（Workspace）。一个工作区中允许存放一个或多个项目。

另外用户最好建立一个专用的目录存放自己的项目文件。例如在本指南中我们生成一个 C:\Program files\IAR System\My project 目录。现在双击桌面上的 IAR Embedded Workbench 图标，出现 IAR EWARM 开发环境窗口。

1. 生成新的工作区（Workspace）

选择主菜单 File > New > Workspace 生成新工作区。

2. 生成新项目

① 选择主菜单 Project > Create New Project，弹出生成新项目窗口，见

图 1。

本例选择项目模板（Project template）中的 Empty project。

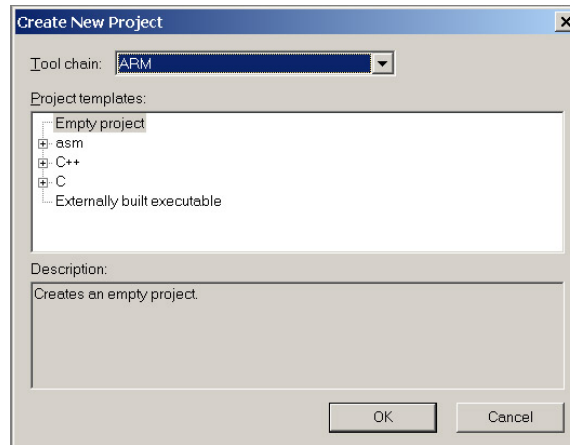


图 1. 生成新项目窗口

- ② 在 Tool chain 栏中选择 ARM，然后点击 OK 按钮。
- ③ 在弹出的另存为窗口中浏览和选择新建的 My projects 目录，输入文件名 project1，然后保存。这时在屏幕左边的 Workspace 窗口中将显示新建的项目名。见图 2 所示：

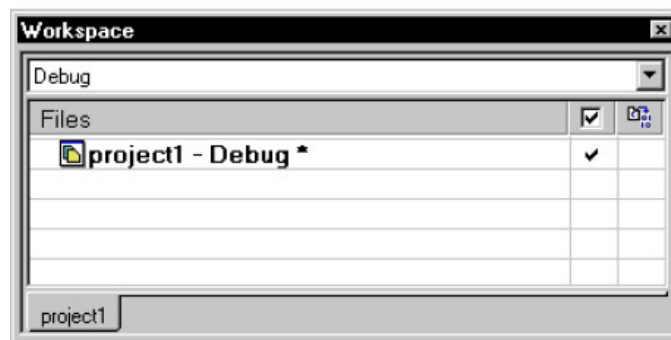


图 2. Workspace 窗口

IAR EWARM 提供两种缺省的项目生成配置，即 Debug 和 Release。本例在 Workspace 窗口顶部的下拉菜单中选取 Debug。现在 My projects 目录下已生成一个 project1.ewp 文件。该文件中包含与 project1 项目设置有关的信息，如 build 选件等。项目名后缀上的 * 号表示该工作区有改变但还没有被保存。

本例调用 `printf` 库函数，这是在 C-SPY 模拟器中的一个低级 `write` 函数。如果用户希望在真实硬件上以 `release` 配置运行例子，就必须提供与硬件相适配的 `write` 函数。

④ 保存工作区

先选择主菜单 `File > Save Workspace`，浏览并选择 `My projects` 目录。然后将工作区取名为 `tutorials` 输入 `File name` 输入框，按保存按钮退出。这时在 `My projects` 目录下将生成一个 `tutorials.eww` 文件，该文件中保存了用户添加到 `tutorials` 工作区中的所有项目。窗口和断点放置等与当前操作有关的其他信息则被存储在 `My projects\ settings` 目录下的文件中。

3. 给项目添加文件

本例我们将采用 `arm\tutor` 目录下的两个源文件，*Tutor.c* 和 *Utilities.c*。

Tutor.c 是一个只用到标准 C 语言的简单程序。它用 Fibonacci 数列的前十个数初始化一个数组，并把结果打印到 `stdout`；*Utilities.c* 包含计算 Fibonacci 数列的实用程序。

IAR EWARM 允许生成若干个源文件组。用户可以根据项目需要来组织自己的源文件。但在本例中没有必要。

- ① 在 `Workspace` 中选择希望添加文件的目的地，可以是项目或源文件组。本例直接选 `project1`。
- ② 选择主菜单 `Project > Add Files` 打开标准浏览窗口，见图 3。选择安装目录 `ARM\tutor` 下的上述 2 个文件，点击打开按钮，把它们添加到 `Project1` 目录下。

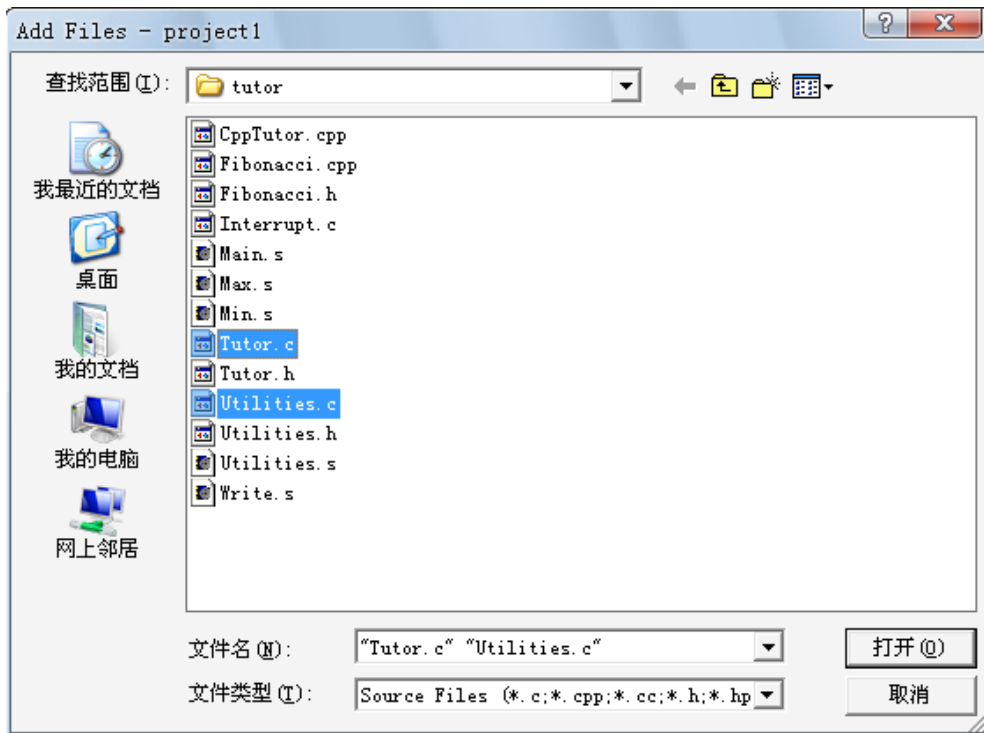


图 3. 添加文件窗口

4. 设置项目选件

生成新项目 and 添加文件后就应该为项目设置选件。IAR EWARM 允许为任何一级目录和文件单独设置选件，但是用户必须为整个项目设置通用的编译连接（build）选件。

① 选择通用选件

选中 Workspace 中的 project1 - Debug，然后选择主菜单 Project > Options。也可以先选择 project1 - Debug，然后选择鼠标右键命令中的 Options。

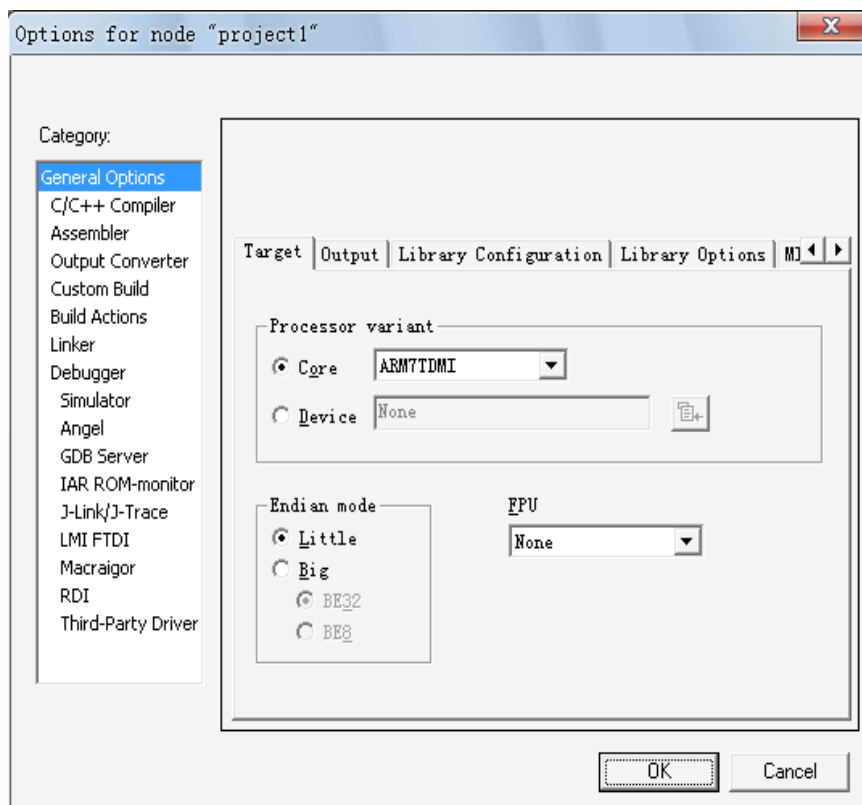
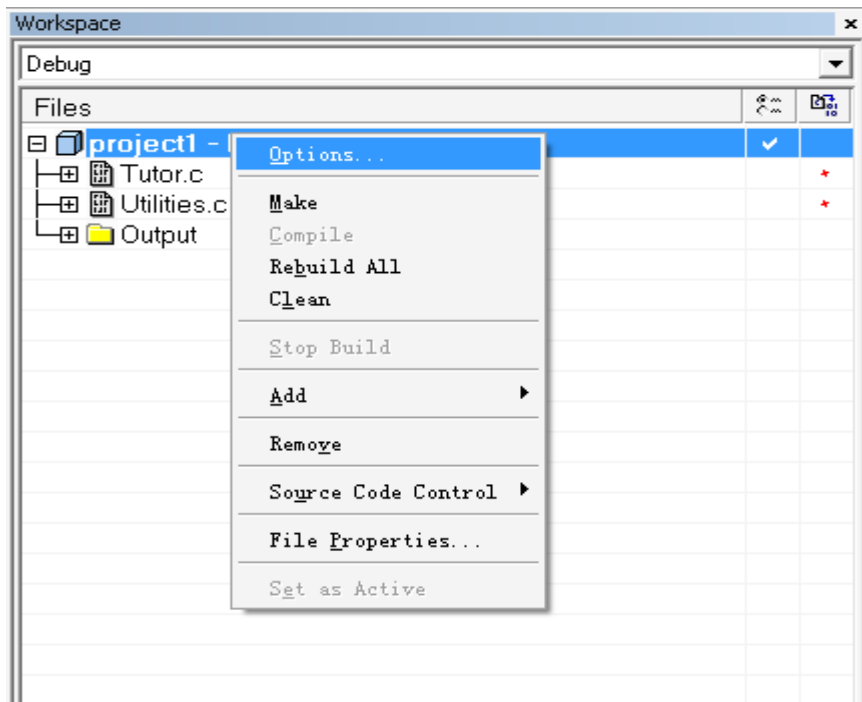


图 4. 项目通用选件窗口

在打开的 Options 窗口左边的 Category 中选择 General Options。
然后分别在：

- Target 页面/Core 条目下选择 ARM7TDMI-S
- Output 页面中，Output file 条目下选择 Executable
- Library Configuration 页面中，Library 条目下选择 Normal

② 选择编译器选件

在 Options 窗口的 Category 中选择 C/C++ Compiler，见图 5。

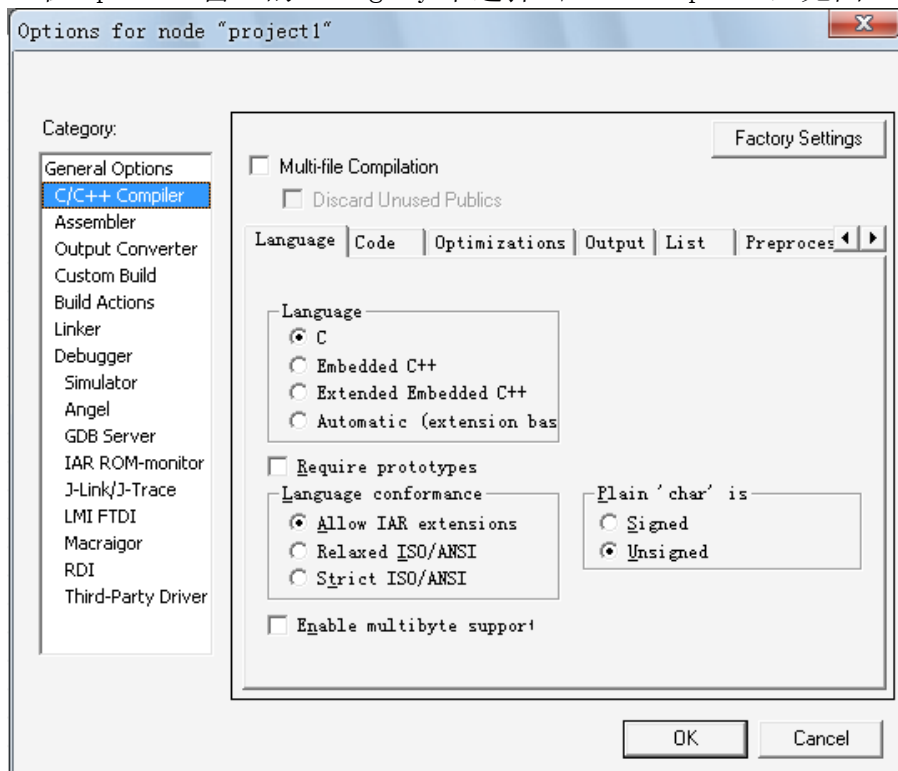


图 5. C/C++ Compiler 选件窗口

然后在：

- Language 页面中，选择 C，Allow IAR extensions 等
- Output 页面中，选择 Generate debug information
- List 页面中，选择 Output list file 和 Assemble mnemonics
- List 页面中，选择 Output list file。并选择 Assembler mnemonics 和 Diagnostics
- 点击 OK 按钮，确认选择的选件

在设置项目选件窗口中有许多其他信息。由于本例比较简单，所以不涉及这

些内容。

3、编译和连接应用程序

这一步编译和连接 (build) 项目程序，同时生成一个编译器列表文件 (compiler list file) 和一个连接器存储器分配文件 (linker map file)。

1. 编译源文件

- ① 选中 workspace 中 *utilities.c* 文件。
- ② 选择主菜单 Project > Compile，或工具条中的 Compile 按钮，或按右键后选择 Compile 命令。编译结束后在消息窗口中出现如图 6 中的信息。

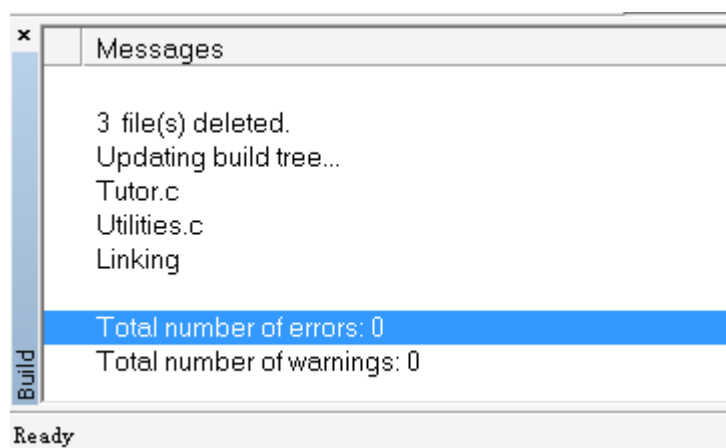


图 6. Build 窗口中的编译处理消息

- ③ 用同样的方法编译 *tutor.c*。

编译完成后在 My projects 目录下将生成一批新子目录。因为我们在建立新项目时选择 Debug 配置，所以在 My projects 目录下自动生成一个 Debug 子目录。Debug 子目录下又包含另 3 个子目录，名字分别为 List、Obj、Exe。它们的用途如下：

- List 目录下存放列表文件，列表文件的后缀是 lst；
- Obj 目录下存放 Compiler 和 Assembler 生成的目标文件，这些文件的后缀为 r79，可以用作 IAR XLINK 连接器的输入文件；
- Exe 目录下存放可执行文件，这些文件的后缀为 d79，可以用作 IAR C-SPY 调试器的输入文件，注意在执行连接处理之前这个目录是空的。

点击 project1 - Debug 前面的+号将目录展开。你可以从自动生成的 Output 目录中看到所有生成的输出文件名以及反映相互依赖关系的头文件名。

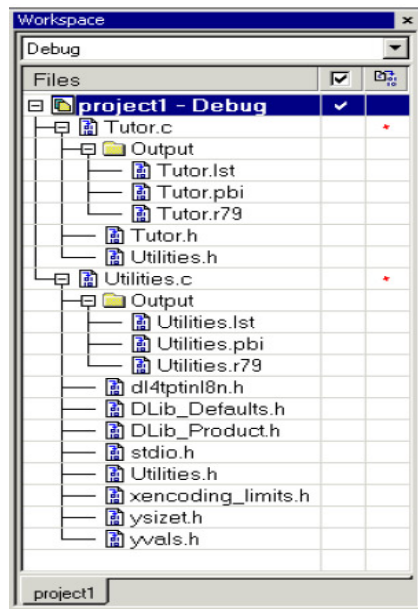


图 7. 编译处理后的文件结构

2. 查看编译器列表文件

现在通过改变编译器选项中的优化级别 (Optimization) 来观察 list 文件是如何自动更新生成的代码量的。

① list 文件的结构

双击 Workspace 窗口中的 *Utilities.lst*, 打开 list 文件, 它包含以下信息:

- 文件头 — 显示编译器的版本信息, 列表文件生成时间, source 文件、list 文件和 object 文件的名称和路径, 编译命令行及选项等信息。
- 文件体 — 显示为每条源语句生成的汇编代码和二进制代码, 以及变量如何被分配到不同的段。
- 文件尾 — 显示所需的堆栈、程序代码以及数据存储器的总量, 同时报告错误和警告信息。

② 选择主菜单 Tools > Options 弹出 IDE Options 对话框, 选择 Editor 页面。选择 Scan for Change Files 选项。此选项将自动打开编辑窗口中

的文件，目前是 *Utilities.lst* 文件。按 OK 按钮。

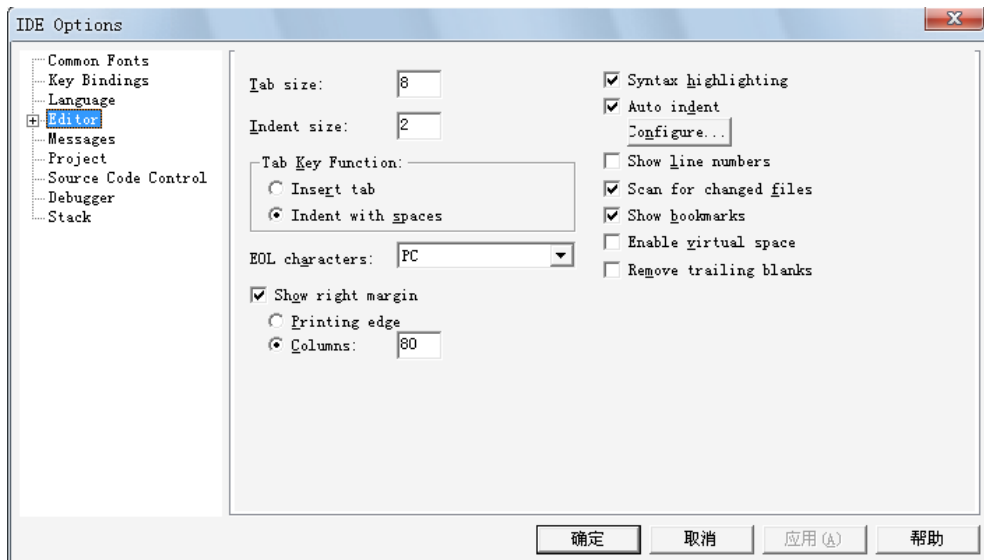


图 8. IDE Option 窗口

- ③ 选中 Workspace 窗口中的 *Utilities.c*，按鼠标右键选择弹出框中的 Options…。从弹出的对话框左边的 Category 中选择 C/C++ Compiler 并确定 Override inherited settings。打开 Optimization 页面，把优化级别从 None 改定为 High。然后按 OK 按钮。
 - ④ 重新编译 *Utilities.c*，请注意这时编辑窗口中的 *Utilities.lst* 文件已经自动被刷新。文件尾显示的代码大小已经因优化级别的升高而减小。
 - ⑤ 对本例而言，Optimization 应选择 None。所以在连接处理前应该将优化级别恢复到原来的设置。这时应选中 *Utilities.c*，按鼠标右键选择弹出框中的 Options…。选择 C/C++ Compiler 并取消 Override inherited settings。然后重新编译 *Utilities.c*。
3. 连接应用程序
- ① 先选中 Workspace 窗口中的 Project1 - Debug，然后选择主菜单 Project > Options，弹出 Options 对话框，见图 9。在左边的 Category 中选择 Linker，显示 IAR XLINK 的各选项页面。

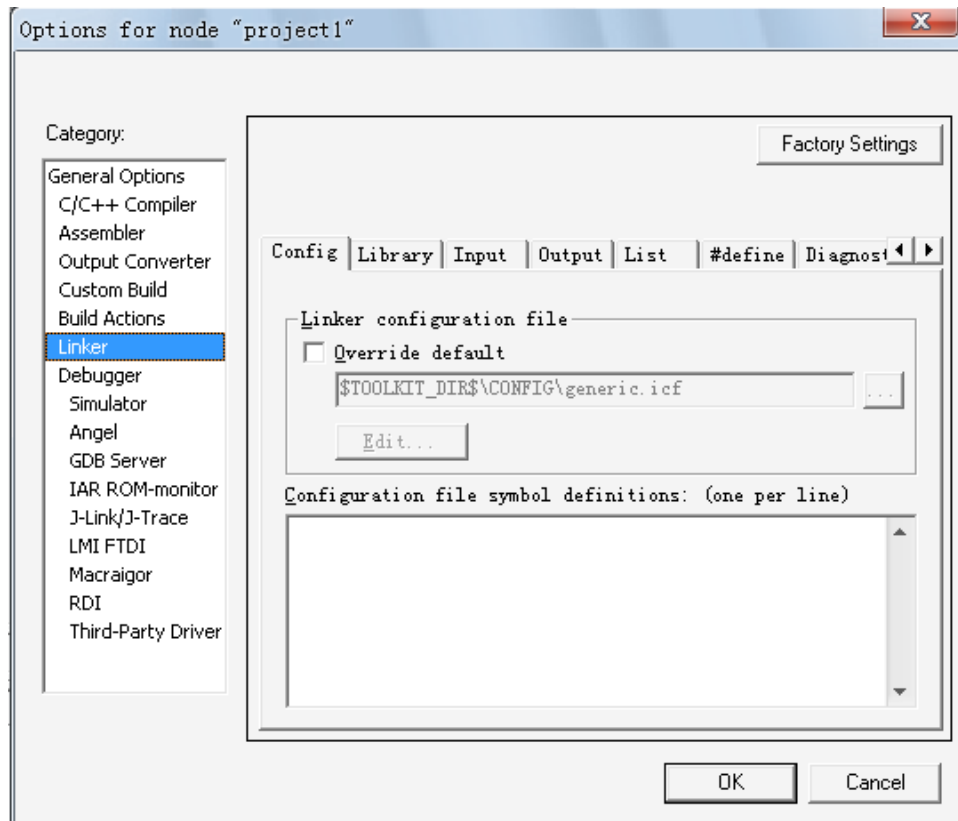


图 9. XLINK 参数选件窗口

本例全部采用缺省的连接处理选件。但是仍需要强调一下输出文件格式和 Linker 命令行文件的选择方法：

❖ 输出格式

选择合适的输出格式十分重要。你可能需要将输出文件送给一个调试器进行调试，这时就要求输出格式带有调试信息。本例采用适合 C-SPY 调试器的缺省输出选件，它们是 Debug information for C-SPY、With runtime control modules 和 With I/O emulation modules，指示需要连接将 stdin 和 stdout 指向 C-SPY 的 I/O 窗口的低级例程。

如果用户希望把应用下载到一个 PROM 或 Flash 编程器，则其输出格式不需要带调试信息，如 Intel-hex 或 Motorola S-records。

EWARM 在生成适合 C-SPY 调试器的缺省输出文件的同时，还可以生成第二个输出文件，供编程器或其它调试器使用。第二个输出文件的格式可以在 Extra Output 页面中选择。

在 List 页面中选择 Generate linker listing 和 Segment map(见图 10)，

允许生成存储器分配 MAP 文件。

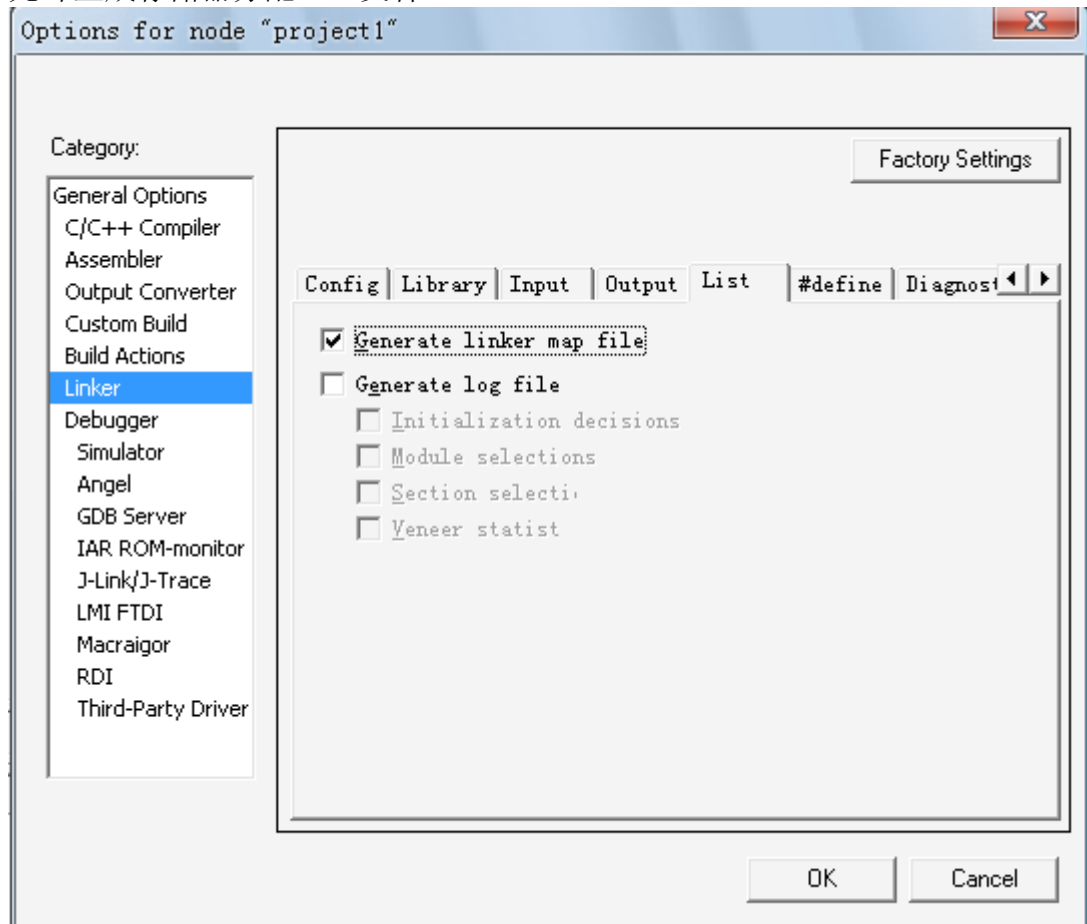


图 10. XLINK 选项中的 List 页面

❖ 连接器命令文件

连接器命令文件（Linker Command File）包含了连接器的各项命令行参数，主要用于控制代码段和数据段在存储器中如何分布。本例使用缺省的连接器的命令文件，请见图 11 中的 Config 页面。

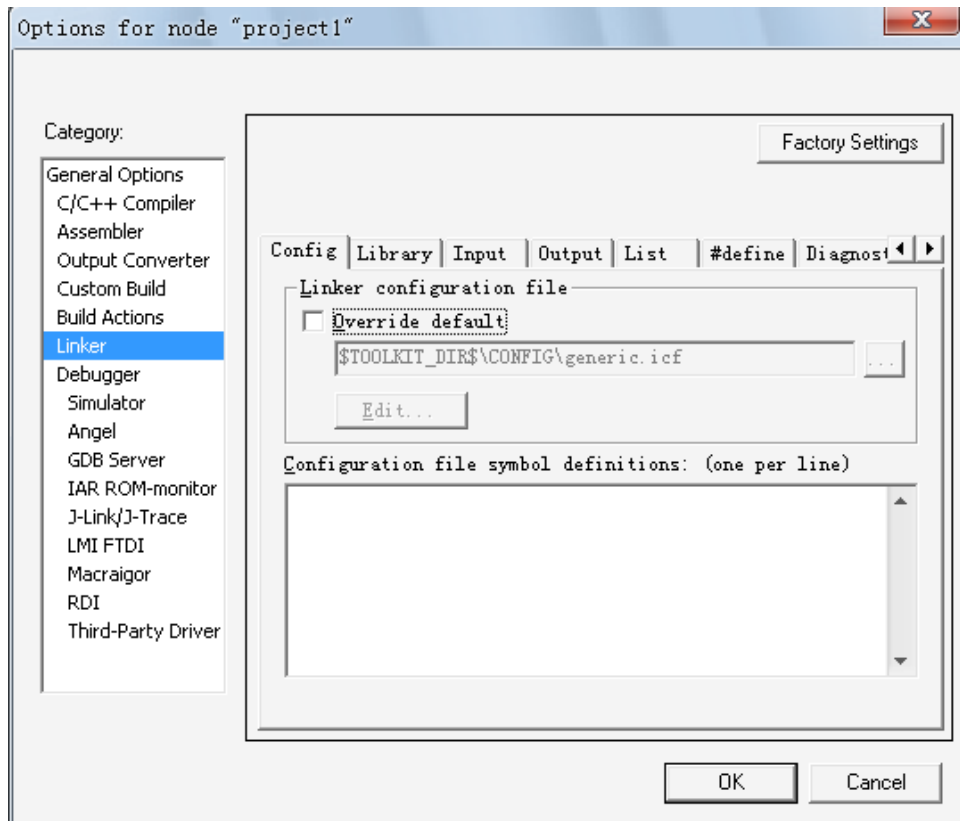


图 11. XLINK 选项中的 Config 页面

注）本例连接器命令文件中的定义不与任何特定的硬件相关联。EWARM 提供的连接器命令文件模板都可以在模拟器（simulator）中使用，但是如果要把它们用于目标系统则必须与实际的硬件存储器分布相适配，因此熟悉连接器命令文件的格式和各段的定义十分重要。用户可以从 arm\src\examples 目录中找到与评估板相关的连接器命令文件。

用户如果要检查连接器命令文件，需用合适的文本编辑器，例如 IAR EWARM 的编辑器。也可以打印出来，检查各项定义是否符合要求。

关于连接器命令文件的定制方法和连接器命令行参数的详细语法，请参阅手册 IAR C/C++ Compiler Reference Guide 和 IAR Linker and Library Tools Reference Guide 的相关章节。最常用的几个连接器命令行参数包括：

- c: 指定目标处理器种类
例：-carm
- D: 定义常量

例：-DROMSTART=00000000

-DROMEND=00100000

-Z：将段放置在连续的存储器空间中

例：-Z (CONST) MYSEGA, MYSEGB=008000-0FFFFFF

-P：将段放置在非连续的存储器空间中

例：-P (DATA) MYDATA=100000-101FFF, 120000-121FFF

② 点击 OK 按钮保存 IAR XLINK 选项

③ 选择主菜单 Project > Make 或鼠标右键 Make 命令，连接目标文件，生成可执行代码。

Build 消息窗口中将显示连接处理的消息。连接的结果将生成一个带调试信息的代码文件 *project1.d79* 和一个存储器分配（MAP）文件 *project1.map*。

4. 查看 MAP 文件

双击 Workspace 中的 *project1.map* 文件名，编辑器窗口中将显示该 MAP 文件。从 MAP 文件中我们可以了解以下内容：

- 文件头中显示连接器版本，输出文件名以及连接命令使用的选项。
- CROSS REFERENCE 段显示程序入口地址。
- RUNTIME MODEL 段显示使用的运行时模块的属性。
- MODULE MAP 段显示所有被连接的文件。每个文件中，作为应用程序一部分加载的有关模块的信息，包括各段和每个段中声明的全局符号都列出来。
- SEGMENTS IN ADDRESS ORDER 段列出了组成应用程序的所有段的起始地址和结束地址，字节数，类型和对齐标准等。
- END OF CROSS REFERENCE 段落显示总的代码和数据字节数。

到此为止，已经生成 *project1.d79* 应用程序并可以用于在 IAR C-SPY 中调试。

4、用 C-SPY 调试应用程序

本例使用 C-SPY 的模拟器 (Simulator) 来展现 IAR C-SPY 调试器的基本特点。前面各节生成的 *project1.d79* 应用程序已经可以用 C-SPY 调试器进行调试。用户利用调试器可以查看变量、设置断点、观察反汇编代码、监视寄存器和存储器、在 Terminal I/O 窗口打印输出。

1. 开始调试

在开始调试之前必须设置几个 C-SPY 选件。具体操作如下：

- ① 选择主菜单 Project > Option, 选择 Category 中的 Debugger。在 Setup 页面, 在 Driver 的下拉菜单中选择 Simulator, 同时选择 Run to main, 点击 OK。

如果用户已经购买了 IAR 的 JTAG 仿真器, 请选择 J-Link。

- ② 选择主菜单 Project > Debug 或工具条上的 Debugger 按钮。IAR C-SPY 将开始装载 *project1.d79*。除了已经打开的窗口外, 将显示一组 C-SPY 专用窗口。

2. 组织窗口

在 EWARM 中可以固定窗口 (所谓 dock), 也可以组织成书签形式, 也可以让它们浮动。改变浮动窗口的大小时其他窗口不受影响。

注意 EWARM IDE 窗口最底部的状态条中包含如何安排窗口的有用信息。详细信息请参见 77 页 *Organizing the windows on screen*。

在开始调试前请确认如图 12 所示的各窗口和内容已经显示在屏幕上。在编辑器窗口应能看到源文件 *Tutor.c* 和 *Utilities.c* 以及 Debug Log 消息窗口。

3. 检查源语句

- ① 检查源语句, 双击 Workspace 中的 *Tutor.c*;
- ② 在编辑器显示文件 *Tutor.c* 后, 用 Debug > Step Over 命令 (或 F10) 步进到 *init_fib* 函数调用语句;
- ③ 用 Debug > Step Into 命令 (或 F11) 进入函数 *init_fib*;

注) Step Over 命令用来执行源程序中的一条语句或一条指令,即使这条语句是一函数调用语句。而 Step Into 命令则进入到函数或子程序调用的内部。

当执行 Step Into 后,活跃窗口已经切换到 Utilities.c,因为 init_fib 在这个文件里。

④ 继续用 Step Into 命令直到 for 循环语句;

⑤ 再用 Step Over 命令回到 for 循环的头。请注意,现在是在函数调用级上而不是语句级步进。

注) 还有一种语句级步进的命令, Debug > Next statement 或工具条上的 Next statement 按钮。这条命令与 Step Into 和 Step over 不同。

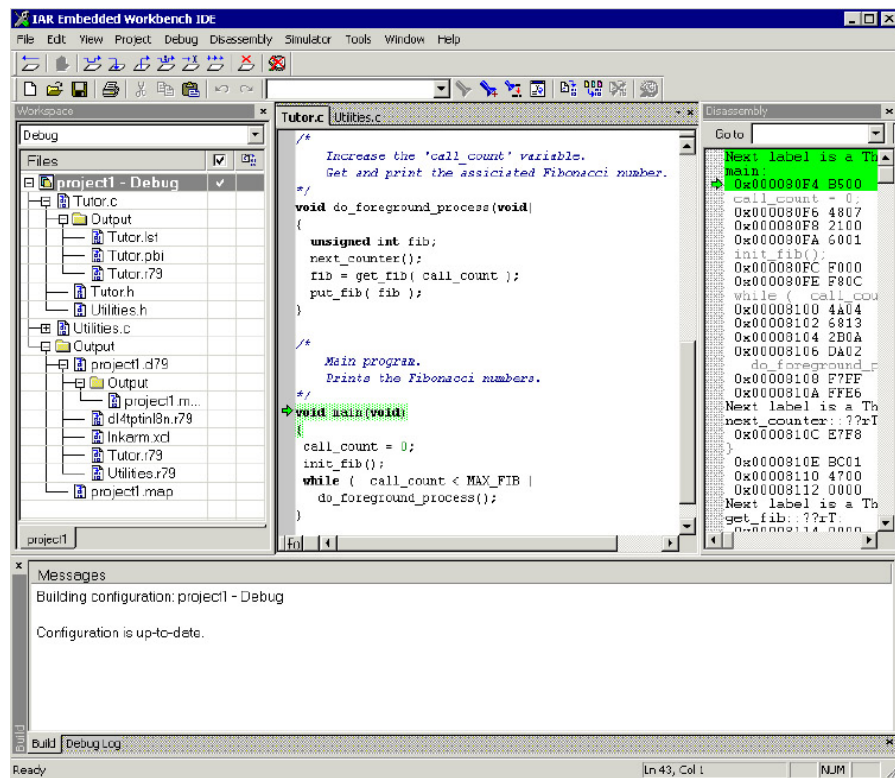


图 12. C-SPY 调试窗口

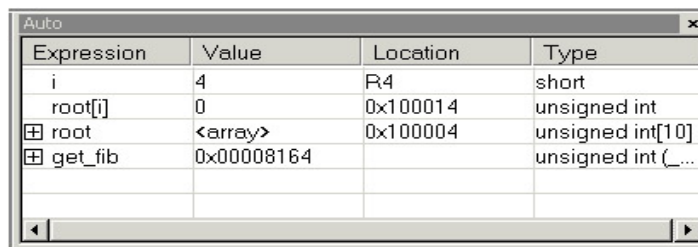
4. 检查变量

C-SPY 允许在源程序上查看变量或表达式,所以可以在执行程序过程中跟踪它们的值的变化。查看变量的方法有几种,在源码窗口用鼠标双击变量名、然后打开 Locals、Live Watch 或 Auto 窗口。如何检查变量的更详细信息请看章节 *Working with variables and expressions*。

注) 当采用 None 优化级时,所有的非静态变量在它们的活动范围内都是活跃的,所以这些变量是完全能够调试的。但如果使用更高级别的优化,变量可能不能完全调试。

① 利用 Auto 窗口查看变量

选择 View > Auto 打开 Auto 窗口。Auto 窗口显示最近修改过的表达式的当前值,单步执行程序观察变量如何变化。



Expression	Value	Location	Type
i	4	R4	short
root[i]	0	0x100014	unsigned int
root	<array>	0x100004	unsigned int[10]
get_fib	0x00008164		unsigned int(_...

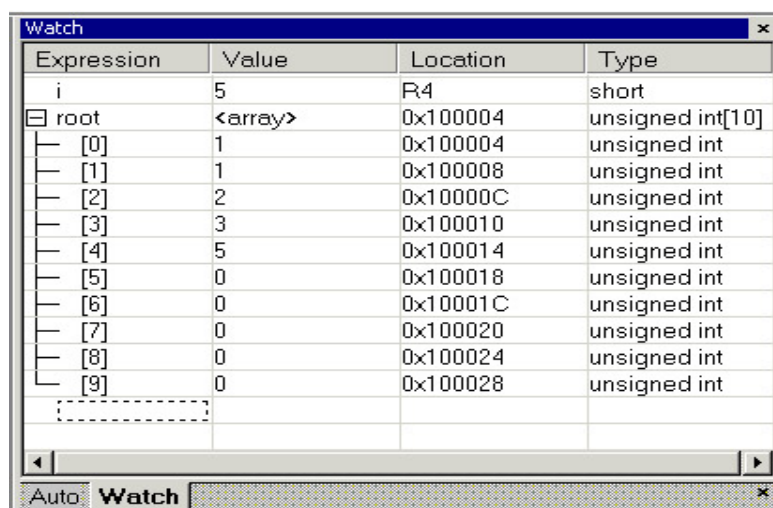
图 13. Auto 窗口中检查变量

② 设置一个 Watchpoint, 利用 Watch 窗口查看变量

选择 View > Watch 打开 Watch 窗口。请注意 Watch 窗口和 Auto 窗口按书签形式显示。按以下步骤在变量 i 上设置一个 Watchpoint。

- 点击 Watch 窗口中的虚线框,当输入区出现时输入 i,然后按 Enter 键。也可以从编辑器窗口拖一个变量到 Watch 窗口。
- 双击 init_fib 函数中的 root 数组名,将其拖到 Watch 窗口。

Watch 窗口将显示 i 和 root 的值。将 root 展开观察每个元素的值。



Expression	Value	Location	Type
i	5	R4	short
root	<array>	0x100004	unsigned int[10]
[0]	1	0x100004	unsigned int
[1]	1	0x100008	unsigned int
[2]	2	0x10000C	unsigned int
[3]	3	0x100010	unsigned int
[4]	5	0x100014	unsigned int
[5]	0	0x100018	unsigned int
[6]	0	0x10001C	unsigned int
[7]	0	0x100020	unsigned int
[8]	0	0x100024	unsigned int
[9]	0	0x100028	unsigned int

图 14. Watch 窗口

- 继续执行单步，观察 i 和 root 值的变化。
- 从 Watch 窗口中除去一个变量时，只需选择它然后删除。

5. 设置和监视断点

IAR C-SPY 具有强大的断点功能。详细请见手册 131 页 *The breakpoint system*。设置断点最简单的方法是将光标定位到某条语句，然后按鼠标右键选择 Toggle Breakpoint 命令。实验方法如下：

① 设置断点

用下面方法在 get_fib(i) 语句上设置断点。在编辑器窗口显示 utilities.c。点击要设置断点的语句，选择主菜单 Edit > Toggle Breakpoint。也可以按工具条上的 Toggle Breakpoint 按钮。这时该语句上将出现断点标记。如果要查看刚定义的断点，选择主菜单 View > Breakpoint 打开 Breakpoint 窗口。在 Debug Log 窗口也显示有关断点执行的信息。

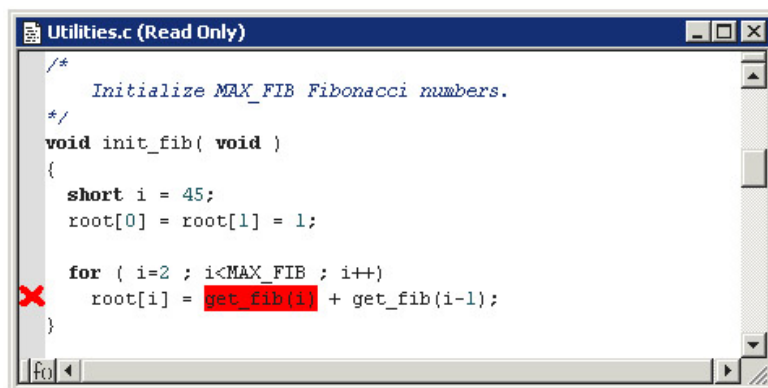


图 15. 设置断点

② 执行到断点

选择主菜单 Debug > Go 或者工具条上的 Go 按钮都可以让程序执行到断点。Watch 窗口将显示 root 表达式的值。Debug Log 窗口将显示关于断点的信息。

③ 消除断点

可用主菜单 Edit > Toggle Breakpoint 或按鼠标右键选择 Toggle Breakpoint。

6. 在反汇编窗口上调试

通常，在 C/C++ 程序上调试应该更快速和更直接。但是如果用户希望在反汇编程序上调试，C-SPY 也提供了这种功能，而且 C-SPY 允许方便地在两种方式上

切换。反汇编程序的调试方法如下：

- ① 按 Reset 按钮复位应用程序。
- ② 调试时反汇编窗口通常是打开的。如果没打开可选择主菜单 View > Disassembly 打开反汇编窗口。

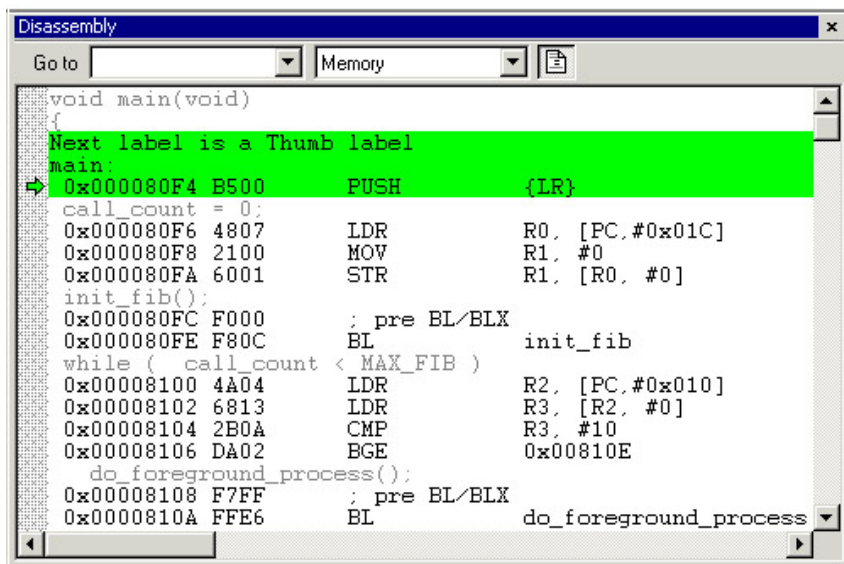


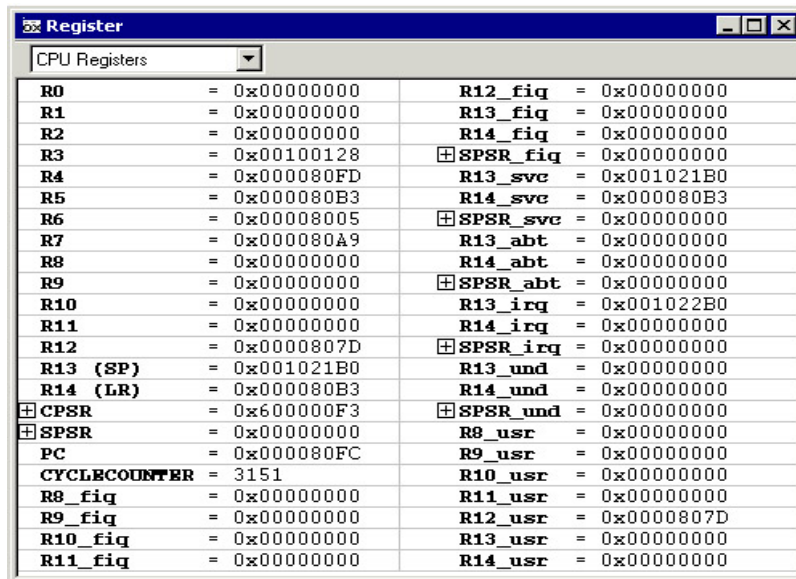
图 16. 反汇编窗口

反汇编窗口如图 16 所示。可以看到汇编代码与 C 语句一一对应。用上面介绍的几种单步命令执行程序观察结果。

7. 监视寄存器

寄存器窗口允许用户监视和修改 CPU 寄存器的内容。具体方法如下：

- ① 选择主菜单 View > Register 打开寄存器窗口，见图 17。



CPU Registers	
R0	= 0x00000000
R1	= 0x00000000
R2	= 0x00000000
R3	= 0x00100128
R4	= 0x000080FD
R5	= 0x000080B3
R6	= 0x00008005
R7	= 0x000080A9
R8	= 0x00000000
R9	= 0x00000000
R10	= 0x00000000
R11	= 0x00000000
R12	= 0x0000807D
R13 (SP)	= 0x001021B0
R14 (LR)	= 0x000080B3
CPSR	= 0x600000F3
SPSR	= 0x00000000
PC	= 0x000080FC
CYCLECOUNTER	= 3151
R8_fiq	= 0x00000000
R9_fiq	= 0x00000000
R10_fiq	= 0x00000000
R11_fiq	= 0x00000000
R12_fiq	= 0x00000000
R13_fiq	= 0x00000000
R14_fiq	= 0x00000000
SPSR_fiq	= 0x00000000
R13_svc	= 0x001021B0
R14_svc	= 0x000080B3
SPSR_svc	= 0x00000000
R13_abt	= 0x00000000
R14_abt	= 0x00000000
SPSR_abt	= 0x00000000
R13_irq	= 0x001022B0
R14_irq	= 0x00000000
SPSR_irq	= 0x00000000
R13_und	= 0x00000000
R14_und	= 0x00000000
SPSR_und	= 0x00000000
R9_usr	= 0x00000000
R10_usr	= 0x00000000
R11_usr	= 0x00000000
R12_usr	= 0x0000807D
R13_usr	= 0x00000000
R14_usr	= 0x00000000

图 17. 寄存器窗口

- ② 用 Step Over 命令执行下一条指令，观察寄存器窗口中的数据如何变化。
- ③ 关闭寄存器窗口。

8. 查看存储器

用户可以在存储器窗口监视所选择的存储器区域。下面是检查与变量 `root` 有关的存储器内容。

- ① 选择主菜单 View > Memory 打开存储器窗口，见图 18(用 8-bit 显示数据)。
- ② 激活 *Utilities.c* 窗口并双击变量 `root`。用鼠标将其拖到存储器窗口。
- ③ 如果希望以 16-bit 显示数据，在存储器窗口定部的下拉菜单中选择 2x Units 命令。

如果 C 应用程序的 `init_fib` 函数没有初始化所有的存储器单元，继续执行单步，同时观察存储器的内容是如何修改的。用户可以在存储器窗口修改存储单元的内容。只需把插入点放在希望修改的地方，然后输入新值就可以了。

- ④ 关闭存储器窗口。

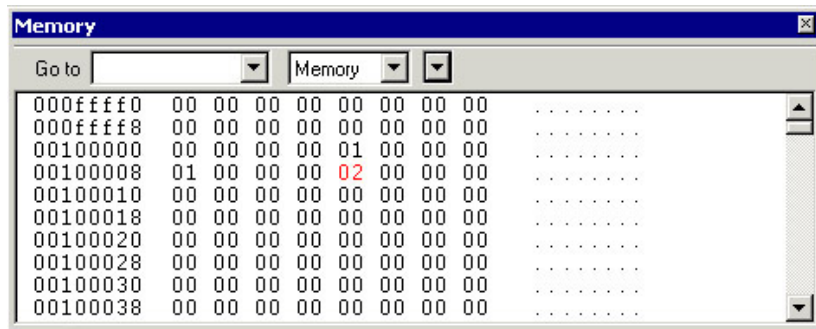


图 18. 8-bit 模式显示存储器窗口

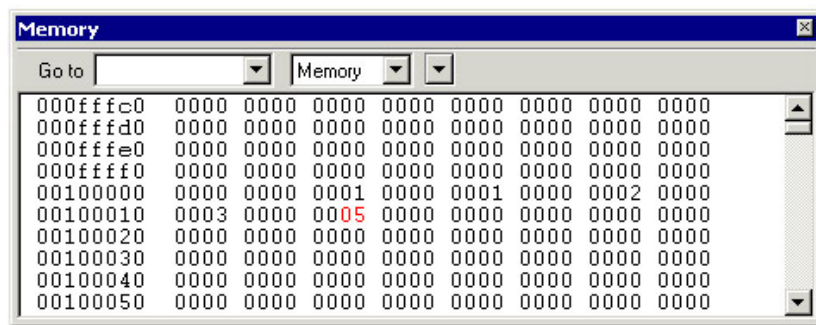


图 19. 16-bit 模式显示存储器窗口

9. 观察 Terminal I/O

用户有时可能希望调试应用程序中的 stdin 和 stdout，但是又没有实际的硬件支持。C-SPY 允许用户使用 Terminal I/O 模拟 stdin 和 stdout。

注)Terminal I/O 只有在使用了连接输出文件选项 With I/O emulation module 时才可用。也就是说，某些把 stdin 和 stdout 指向 Terminal I/O 的低级例程将被连接进应用程序。

- ① 选择主菜单 View > Terminal I/O 显示 I/O 操作的输出，见图 20。

Terminal I/O 窗口显示的内容取决于应用程序执行了多远。

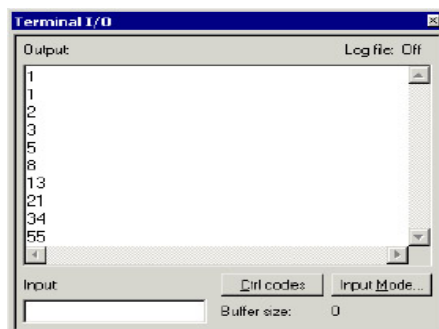


图 20. Terminal I/O 窗口

10. 执行程序到结束

- ① 选择主菜单 Debug > Go 或工具条上的 Go 按钮。因为只有一个断点，所以程序一直执行到结束。同时在 Debug Log 窗口显示已经到达程序 exit 的消息，见图 21。

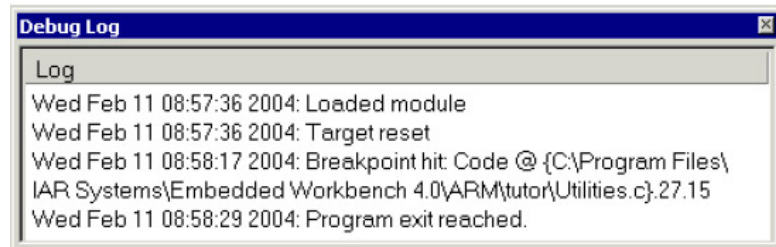


图 21. Debug Log 窗口

- ② 如果要求复位应用程序，选择主菜单 Debug > Reset 或工具条上的 Reset 按钮。
- ③ 如果要退出 C-SPY，选择 Debug > Stop Debugging，或工具条上的 Stop Debugging 按钮。

C-SPY 还提供许多其他的调试功能，如宏和中断模拟等，将在指南的其他章节讨论。有关如何使用 Debug 功能的详细介绍请见手册 Part 4。C-SPY 的特点介绍请见手册 Part 7 以及联机帮助信息。

5、EWARM Flash Loader 开发指南

本章包含以下内容：

- 如何将应用程序下载到 RAM 中
- 如何将应用程序下载到 Flash 中
- 从框架程序和驱动程序两个部份分别介绍 Flash Loader

本文还介绍了如何编写和调试自己的 Flash Loader，最后，详细描述了 Flash Loader 框架 API 函数。

注）本文中的 xx 表示 2 个数字，用于识别所用的处理器。

1. 将应用程序下载到 RAM 中

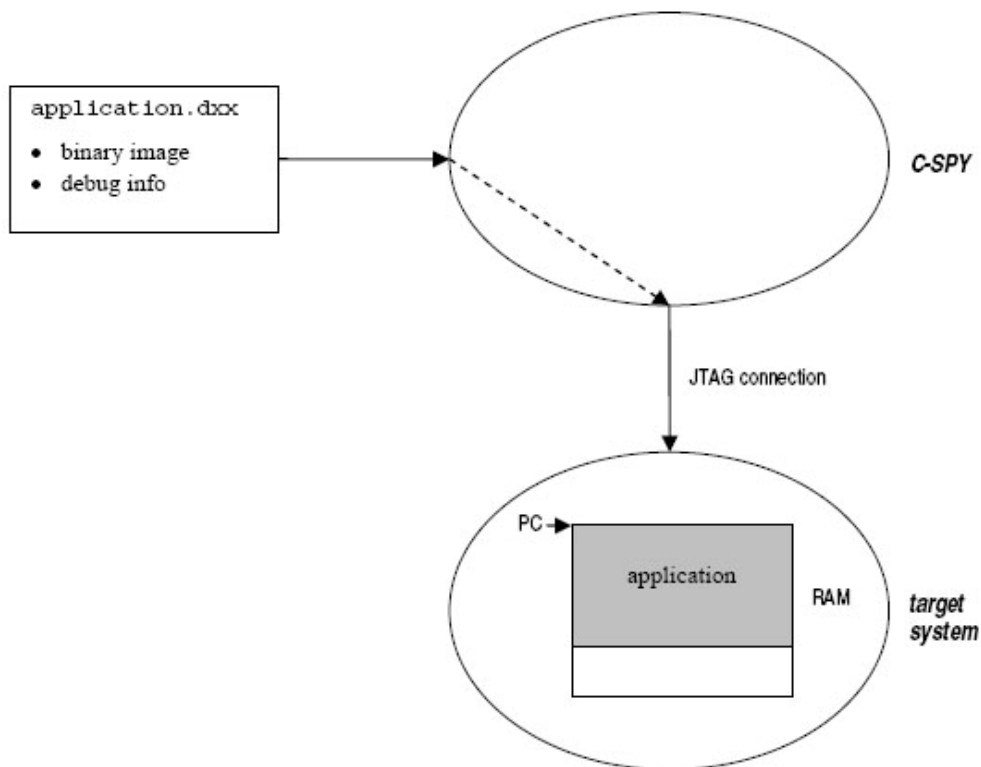
将应用程序下载到 RAM 发生在 C-SPY 启动期间，由 C-SPY 自己控制执行。所

谓下载就是通过 JTAG 接口把数据写进目标系统。

当 C-SPY 启动时，它执行以下步骤：

- 从 application.dxx 文件中读取应用程序的二进制映像和调试信息；
- 通过 JTAG 接口将二进制映像传输到目标系统的 RAM 中；
- 将程序计数器（PC）指向 RAM 中的应用程序入口点。

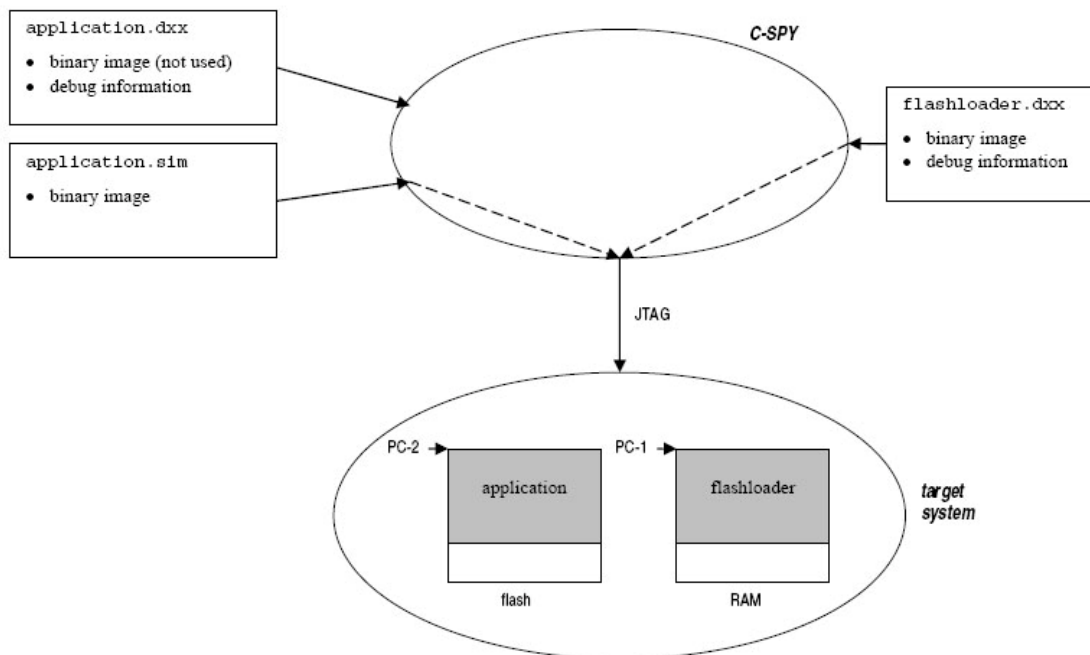
此时 RAM 中的应用程序已经准备好可以运行。



2. 将应用程序下载到 Flash 中

将应用程序下载到 Flash 也发生在 C-SPY 启动期间，但不是由 C-SPY 执行，而是由一个叫做 Flash Loader 的专用程序执行。Flash Loader 先被装入到 RAM 并运行，再将应用程序写进 Flash。链接器（linker）生成两个输出文件，第一个是常规的 UBROF 格式目标文件（扩展名为 dxx），另一个是简单二进制格式目标文件（simple-code，扩展名为 sim）。Simple-code 格式十分简洁而且容易拆包，这是 Flash Loader 得以在目标硬件中执行的重要条件。

Flash Loader 是一个常规的 IAR Embedded Workbench 应用程序，可以在 IAR Embedded Workbench 环境中开发和调试。



当 C-SPY 启动时，它执行以下步骤：

- 从 flashloader.dxx 文件中读取 Flash Loader 的二进制映像；
- 通过 JTAG 接口将该二进制映像写进目标系统的 RAM；
- 将程序计数器（PC-1）指向 Flash Loader 在 RAM 中的入口点，并开始运行；
- 通过文件 I/O，Flash Loader 经由 JTAG 接口把 application.sim 文件中的应用程序二进制映像读入目标系统并写进 Flash 存储器；
- C-SPY 从 application.dxx 文件中读取调试信息，并将程序计数器（PC-2）指向 Flash 中的应用程序入口点；
- 此时 Flash 中的应用程序已经准备好可以运行。

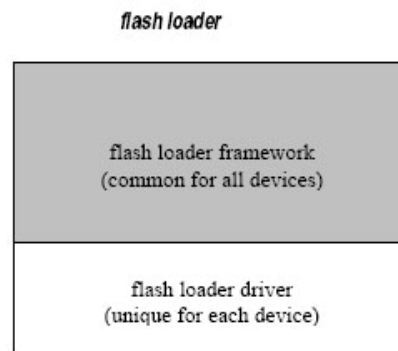
3. Flash Loader 介绍

Flash Loader 是用 IAR Embedded Workbench 开发的本地应用程序。其任务是通过文件 I/O 从主机读取应用程序的二进制映像，将映像拆包，并写进 Flash 存储器。

Flash Loader 可以分成两个部分。一是所有 Flash Loader 所共用的框架部分，其源代码由 IAR Systems 提供并包含在 IAR Embedded Workbench 中；二是驱动部份，它是一小段用于实际烧写 Flash 存储器的小程序。在 IAR Embedded Workbench 中已经包含了一组用于各种芯片的 Flash Loader 驱动程序。由于 Flash Loader 驱动程序很简单，所以用户可以自行编写 IAR Systems 尚未支持的芯片驱动程序。

Flash Loader 框架程序实现了所有 Flash Loader 都具备的公共功能，包含从调试器读取二进制映像，把用户变量（选件）传递给 Flash Loader 的机制，以及为了与用户交互而创建 GUI 元素。GUI 元素包括消息窗口、消息记录和进度条等。缺省情况下，进度条由 Flash Loader 框架程序所控制。

Flash Loader 必须遵循 Flash<device>.dxx 的命名约定。例如假设某器件名叫 IAR X99，则它的 Flash Loader 应命名为 FlashIarX99.dxx。



IAR 公司提供的 Flash Loader 源代码位于下列目录：

arm\src\flashloader\framework	Flash Loader 框架程序源代码，含 API 头文件
arm\src\flashloader\<vendor>\lash<device>	各 Flash Loader 驱动程序源代码，含工程文件

IAR 公司提供的可执行的 Flash Loader 位于下列目录：

arm\config\flashloader\<vendor>\lash<device>.dxx	对应于各驱动程序的 Flash Loader 可执行文件
arm\config\flashloader\<vendor>\lash<device>.mac	可选的 C-SPY 宏文件。如果宏文件的名字和 Flash Loader 可执行文件相同，该宏文件将先于同名 Flash Loader 被装进 RAM 并运行。有些芯片需要对一些 I/O 寄存器进行初始化之后 RAM 才能正常工作，这时此项功能就很有用。

4. 可选的 Flash Loader C-SPY 宏文件

在将 Flash Loader 装入 RAM 之前可能需要执行一个 C-SPY 宏来设置目标系统。例如，某些芯片在复位后 RAM 还不能正常工作，就需要用一个宏来初始化必要的寄存器，以便让 RAM 正常工作。

在将 Flash Loader 装入 RAM 之前所执行的宏应满足以下规定：

- 宏文件应存放在同名 Flash Loader 的目录下；
- 宏文件的扩展名应为 mac；

- 宏文件名应与其关联的 Flash Loader 名相同;
- 宏文件中必须定义 **execUserFlashInit()** 宏函数。C-SPY 将在把 Flash Loader 装进 RAM 之前调用该宏函数。请注意在调试阶段,当 Flash Loader 作为一个应用程序运行时,必须用 **execUserPreload()** 替代 **execUserFlashInit()**。

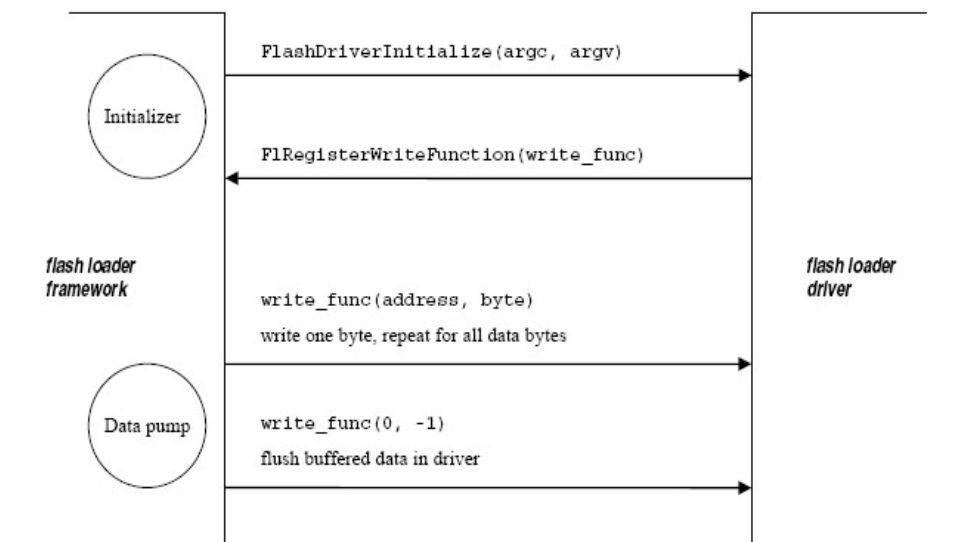
必要的话,在 Flash Loader 运行结束之后,可以用宏函数 **execUserFlashExit()** 来恢复目标系统的初始设置。

5. 与 Flash Loader 框架程序的接口

Flash Loader 框架程序将首先初始化 Flash Loader 驱动程序。此时,驱动程序可以执行各种初始化工作,但至少要向框架程序注册它的写函数。

在初始化之后,框架程序将通过驱动程序的写函数,一次传输一个字节给 Flash Loader。根据所用的 Flash 算法,有可能需要在驱动程序中缓冲多个字节,以便在将一个扇区写入 Flash 存储器之前填满整个扇区空间。从框架程序向驱动程序的最后一次写操作将被视为清空请求,允许驱动程序清空扇区缓冲中的任何剩余数据。如果 Flash Loader 驱动程序没有缓冲任何数据,清空请求可以被忽略。

驱动程序不返回任何错误状态给框架程序。一旦驱动程序中发生错误,驱动程序应通过调用 **FlMessageBox()** API 函数向用户报告错误,然后调用 **FlErrorExit()** 函数退出 Flash Loader。



6. Flash Loader 驱动程序实例

本例展示如何为一种芯片编写 Flash Loader 驱动程序。为简单起见，假设该芯片中有一块很容易编程的 Flash；可以用一个简单的 Flash 算法，在一次操作中将一个字节写入 Flash 存储器。本例还展示了如何读取用户指定的选项，该选项说明了芯片运行的时钟频率。

关于怎样实现一个带扇区缓冲的 Flash Loader，请参考 IAR Embedded Workbench 所安装的 Flash Loader 驱动程序源代码。

```
// Flash loader driver example.
#include <stdio>
#include <stdlib.h>
#include "Interface.h" // The flash loader framework API declaration.

// The CPU clock speed, the default value 4000 kHz is used if no clock // is found.
static int clock = 4000;

// Write one byte to flash at addr.
// If byte == -1 the flash loader framework signals a flush operation
// at the end of the input file.
static void FlashWriteByte(unsigned long addr, int byte)
{
    unsigned char* ptr = (unsigned char*)addr;

    if (byte == -1)
        return; // Simple return when the flush operation is requested.

    // Insert device specific instructions here to enable write
    // access to the flash device.
    *ptr = byte; // Write data byte to flash.

    // If some error occurs when writing to flash, this can be
    // communicated to the user by using code like
    // if (ret != STATUS_CMD_SUCCESS)
    // {
    //     FIMessageBox("CMD_ERASE_SECTORS failed.");
    //     FIERrorExit();
    // }
    // A message box will be displayed by C-SPY and downloading
    // will terminate after the user has clicked the OK button.
}

void FlashDriverInitialize(int argc, char const* argv[])
{
    const char* str;

    // Register the flash write function.
    FIRegisterWriteFunction(FlashWriteByte);
```

```
// See if user has passed a clock speed option.  
// If not, the default CCLK value is used.  
str = FIFindOption("- —clock", 1, argc, argv);  
if (str)  
{  
    clock = strtoul(str, 0, 0);  
}  
}
```

7. 编译链接 Flash Loader

- ① 拷贝一个现有的 Flash Loader，例如
arm\src\flashloader\philips\FlashPhilipsLPC210x;
- ② 确认编译器所用的文件包含路径包括了 Flash Loader 框架程序目录
arm\src\flashloader\framework 和 Flash Loader 驱动程序目录;
- ③ 修改 FlashPhilipsLPC210x.c 和 FlashPhilipsLPC210x.h 这两个文件的名称，使之与所用的芯片相符合;
- ④ 链接器控制文件也应该设置为与所用的芯片相符合。拷贝
FlashPhilipsLPC210x.xcl，并修改其中的地址定义行：
- DMEMSTART=40000000
- DMEMEND=40003FDF

实际使用的地址必须能够映射到目标硬件上。请注意 Flash Loader 的代码和数据都是下载到 RAM 中的，这就是为什么 ROM 段和 RAM 段都映射到同一段存储空间。

栈和堆都应该保持在最小。框架程序大约需要 300 字节的栈空间。请注意下面 xcl 文件中的数字都是十六进制的：

```
- D_CSTAK_SIZE=180  
- D_IRQ_STACK_SIZE=40  
- D_HEAP_SIZE=0
```

Flash Loader 框架程序将使用堆（heap）和 RAMEND（在链接器控制文件中声明）之间的内存作为读缓冲区。这就保证了读缓冲区能够利用所有剩余的内存。读缓冲区应当尽可能的大，以提高下载性能。每次 JTAG 传输的数据字节越多，性能就越高。如果剩余的读缓冲区少于 256 字节，框架程序将会报错，因为少于 256 字节将严重影响性能。

在编译链接 Flash Loader 程序之前，链接选项 **With I/O emulation modules** 必须被打开。得到的输出文件将以 dxx 为文件扩展名。

Flash Loader 已经可以用于把应用程序下载到 Flash。在 Embedded Workbench 中, 打开应用程序工程, 再打开 **Debugger download option** 对话框。使能 Flash download 选项, 并选择 **Override default flash loader** 选项, 指定你生成的 Flash Loader 输出文件。任何需要传递给 Flash Loader 的参数都可以写进 **Flash Loader arguments** 文本域。

现在启动调试器, 就可以用你自己的 Flash Loader 将应用程序下载到 Flash。

8. 调试 Flash Loader

调试 Flash Loader 的方法和调试普通的应用程序一样。需要指出的是, Flash Loader 程序在作为 Flash Loader 被装入调试器时是不能调试的。只有当 Flash Loader 本身就是 IAR Embedded Workbench 中当前打开的工程时, 它才能被调试。

在 Flash Loader 框架程序中有一个调试环境。该环境受头文件 **DriverConfig.h** 中定义的 C 预处理器宏变量控制, 而 **DriverConfig.h** 被包含在框架程序的头文件 **Congig.h** 中。在 **Congig.h** 文件中, 你可以看到哪些变量允许在 **DriverConfig.h** 中被覆盖。

在调试器中以一个独立的应用程序运行 Flash Loader 时有几点不同。要启动框架程序的调试环境, 必须设置调试宏变量 **DEBUG**。要写入 Flash 的文件也必须用宏变量 **DEBUG_FILE** 显式说明。在独立调试时, **argc/argv** 参数传递机制是不工作的, 参数必须用 C 预处理器宏变量 **DEBUG_ARGS** 硬性编码。

9. Flash Loader 框架程序的 API

本节介绍 Flash Loader 框架程序所提供的 API。对于大多数 Flash Loader 来说, 很多 API 函数都是用不到的, 把它们罗列在此只是为了内容的完整性而已。

所有函数都标出了它们的用途。**Mandatory** 表示 Flash Loader 驱动程序必须实现此函数。

Optional 表示 Flash Loader 驱动程序可以根据需要选择性地实现此函数。**Framework** 表示该函数只会被 Flash Loader 框架程序调用, 而通常不应该被 Flash Loader 驱动程序调用。

所有 API 函数的原型都定义在头文件 `arm\src\flashloader\framework\Interface.h` 之中。

初始化函数

```
void FlashDriverInitialize (int argc, char const* argv);
```

用途: *Mandatory*

Flash Loader 驱动程序必须定义此函数。Flash Loader 框架程序用它来初始化 Flash Loader 驱动程序。

argc 传递 Flash 变量的数目;

argv 传递 Flash 变量。

Flash 变量允许通过 C-SPY 的 Flash 选项对话框将参数传递给 Flash Loader。此函数的一个典型用例是将 CPU 的时钟速率传递给 Flash Loader 驱动程序。

```
void FlRegisteWriteFunction (WriteFunctionType write_func);
```

用途: *Mandatory*

Flash Loader 驱动程序调用此函数向 Flash Loader 框架程序注册写函数。变量 **write_func** 是指向写函数的指针。Flash Loader 框架程序将为每个要写进 Flash 的字节调用这个写函数，并将字节和要写的地址作为参数传递。地址的顺序应确保是递增的，但不一定连续（即允许有间隙）。Flash Loader 驱动程序必须从 **FlashDriverInitialize ()** 中调用此函数。

```
typedef void (*WriteFunctionType) (unsigned long address, int byte);
```

此类型声明定义了写函数的函数指针类型。该写函数必须在 Flash Loader 驱动程序中定义。

```
unsigned long FlGetBaseAddress ();
```

这是一个可选函数，用于获得用户在 IDE 中设置的 Flash 基地址。如果用户没有设置 Flash 基地址，该函数返回 0xffffffff。例如，AMD 兼容的 Flash 器件在进行编程和擦除时不依赖于任何控制寄存器，而是对以 Flash 基地址为基准的多个偏移地址进行一系列总线写操作。此时就有必要知道 Flash 的基地址，以便执行可能的 Flash 地址重映射。

变量传递函数

```
const char* FlFindOption (char* option, int with_value, int argc, char const* argv[]);
```

用途: *optional*

此函数用于在变量数组 **argv** 中寻找指定的选项。

with_value 参数说明了该函数是用于检查 **argv** 中是否存在某个选项，或者是否需要返回该选项的值。该选项的值是在 **argv** 中找到匹配变量后的下一个变量。如

果要检查一个 flag 选项，例如--smallram，赋 **with_value** 为 0。如果要检查一个带值的选项，例如--speed 14600，则赋 **with_value** 为 1。

argc 参数是 **argv** 数组中的变量个数，**argv** 参数是字符串指针数组。当调用此函数时，可以直接使用 **FlashDriverInitialize()** 函数中的 **argc/argv** 参数。

如果在 **argv** 中没有找到指定的选项，函数返回一个空指针。

如果 **with_value** 设为 1，函数返回指向匹配选项后面的那个变量的指针。

如果 **with_value** 设为 0，函数返回指向匹配选项在 **argv** 中入口的指针。

```
int FlMakeArgs (char* args, char const* argv[]);
```

用途: *framework*

取一个用 space/tab 分隔的字符串，并生成一个 **argv** 字符串数组，每个数组元素对应于一个选项。**argv** 字符串指针数组必须足够大，以便容纳 **args** 字符串中的所有选项。

函数返回 **argv** 数组中的字符串数目（变量数）。

C-SPY 用户接口函数

```
void FlMessageBox (char* msg);
```

用途: *optional*

C-SPY 将显示一个消息窗口，其中显示 **msg** 参数给出的文本。消息字符串中的文本可以用换行符（\n）分割成多行。Flash Loader 将停止执行，直到按下消息窗口中的 OK 按钮为止。

```
void FlMesagaLog (char* msg);
```

用途: *optional*

C-SPY 将在调试器的 log 窗口显示一个由 **msg** 给出的 log 消息。消息字符串中的文本可以用换行符（\n）分割成多行。

```
void FlProgressBarCreate (char* title);
```

用途: *optional*

C-SPY 将生成一个进度条窗口。**Title** 参数字符串显示在进度条窗口的上方。

```
void FlProgressBarDestroy ();
```

用途: *optional*

C-SPY 将关闭进度条窗口。

```
void FlProgressBarUpdate (int progress);
```

用途: *optional*

C-SPY 将更新进度条，以反映 **progress** 参数的值。**progress** 参数的有效范围是 0 至 100。此函数只有在进度条已经生成的前提下才有效。调用

FlProgressBarValue() 的次数应尽可能少 (<10), 目的是为了减少 JTAG 总线上的 (慢) 传输次数。

void FlOverrideProgressBar ();

用途: *optional*

重载由 Flash Loader 框架程序实现的缺省进度条。大多数情况下, Flash Loader 驱动程序不需要操纵进度条, 因为这是由框架程序中的输入文件读取例程来控制的。如果 Flash Loader 驱动程序实现了它自己的进度条, 就必须调用此函数来禁止缺省的进度条。

void FlErrorExit ();

用途: *optional*

终止 Flash Loader, 并通知 C-SPY 调试器: Flash 下载已经失败。

文件函数

int FlFileOpen (char* name);

用途: *framework*

此函数打开一个文件用于二进制读。如果打开成功返回一个文件句柄; 如果打开失败返回-1。

int FlFileReadByte (int fd);

用途: *framework*

从与文件句柄 **fd** 相关联的已打开文件中读取一个字节。
该函数返回读到的字节; 当已读到文件末尾时返回-1。

void FlFileClose (int fd);

用途: *framework*

关闭与文件句柄 **fd** 相关联的已打开文件。

《IAR EWARM 嵌入式系统编程与实践》

作者: 徐爱钧

首本以 IAR 开发工具为主题的参考书《IAR EWARM 嵌入式系统编程与实践》, 06 年 2 月 27 日由北航出版社正式出版发行。本书以 IAR 公司 4.30A 版本的 EWARM 为核心。

嵌入式系统硬件驱动基础开发案例

实验一 ARM 的串行口实验

一、实验目的

1. 掌握 ARM 的串行口工作原理。
2. 学习编程实现 ARM 的 UART 通讯。
3. 掌握 CPU 利用串口通讯的方法。

二、实验内容

学习串行通讯原理，了解串行通讯控制器，阅读 ARM 芯片文档，掌握 ARM 的 UART 相关寄存器的功能，熟悉 ARM 系统硬件的 UART 相关接口。编程实现 ARM 和计算机实现串行通讯：

ARM 监视串行口，将接收到的字符再发送给串口（计算机与开发板是通过超级终端通讯的），即按 PC 键盘通过超级终端发送数据，开发板将接收到的数据再返送给 PC，在超级终端上显示。

三、预备知识

- 1、用 EWARM 集成开发环境，编写和调试程序的基本过程。
- 2、ARM 应用程序的框架结构。
- 3、了解串行总线

四、实验设备及工具

硬件：ARM 嵌入式开发平台、PC 机 Pentium100 以上、用于 ARM920T 的 JTAG 仿真器、串口线。

软件：PC 机操作系统 Win2000 或 WinXP、EWARM 集成开发环境、仿真器驱动程序、超级终端通讯程序。

五、实验原理及说明

1. 异步串行 I / O

异步串行方式是将传输数据的每个字符一位接一位(例如先低位、后高位)地传送。数据的各不同位可以分时使用同一传输通道,因此串行 I / O 可以减少信号连线,最少用一对线即可进行。接收方对于同一根线上一连串的数字信号,首先要分割成位,再按位组成字符。为了恢复发送的信息,双方必须协调工作。在微型计算机中大量使用异步串行 I / O 方式,双方使用各自的时钟信号,而且允许时钟频率有一定误差,因此实现较容易。但是由于每个字符都要独立确定起始和结束(即每个字符都要重新同步),字符和字符间还可能长度不定的空闲时间,因此效率较低。

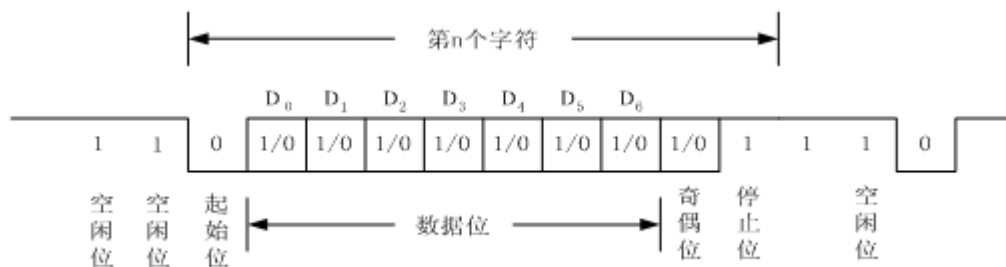


图 3-1 串行通信字符格式

图 3-1 给出异步串行通信中一个字符的传送格式。开始前,线路处于空闲状态,送出连续“1”。传送开始时首先发一个“0”作为起始位,然后出现在通信线上的是字符的二进制编码数据。每个字符的数据位长可以约定为 5 位、6 位、7 位或 8 位,一般采用 ASCII 编码。后面是奇偶校验位,根据约定,用奇偶校验位将所传字符中为“1”的位数凑成奇数个或偶数个。也可以约定不要奇偶校验,这样就取消奇偶校验位。最后是表示停止位的“1”信号,这个停止位可以约定持续 1 位、1.5 位或 2 位的时间宽度。至此一个字符传送完毕,线路又进入空闲,持续为“1”。经过一段随机的时间后,下一个字符开始传送才又发出起始位。

每一个数据位的宽度等于传送波特率的倒数。微机异步串行通信中,常用的波特率为 50, 95, 110, 150, 300, 600, 1200, 2400, 4800, 9600 等。

接收方按约定的格式接收数据,并进行检查,可以查出以下三种错误:

- 1) 奇偶错: 在约定奇偶检查的情况下,接收到的字符奇偶状态和约定不符。
- 2) 帧格式错: 一个字符从起始位到停止位的总位数不对。
- 3) 溢出错: 若先接收的字符尚未被微机读取,后面的字符又传送过来,则产生溢出错误。

每一种错误都会给出相应的出错信息,提示用户处理。

2. 串行接口的物理层标准

通用的串行 I / O 接口有许多种,现仅就最常见的两种标准作简单介绍。

1) EIA RS—232C

这是美国电子工业协会推荐的一种标准(Electronic industries Association Recoil-mended Standard)。它在一种 25 针接插件(DB—25)上定义了串行通信的有关信号。这个标准后来被世界各国所接受并使用到计算机的 I / O 接口中。

(1) 信号连线

在实际异步串行通信中,并不要求用全部的 RS—232C 信号,许多 PC / XT 兼容机仅用 15 针接插件(DB—15)来引出其异步串行 I / O 信号,而 PC 中更是大量采用 9 针接插件(DB—9)来担当此任,因此这里也不打算就 RS—232C 的全部信号作详细解释。图 3-2 给出两台微机利用 RS—232C 接口通信的连线(无 MODEM),我们按 DB—25 的引脚号标注各个信号。下面对图 3-2 中几个主要信号作简要说明。

保护地 通信线两端所接设备的金属外壳通过此线相联。当通信电缆使用屏蔽线时,常利用其外皮金属屏蔽网来实现。由于各设备往往已通过电源线接通保护地,因此,通信线中不必重复接此地线(图中用虚线表示)。例如使用 9 针插头(DB—9)的异步串行 I / O 接口就没有引出保护地信号。

TXD / RXD 是一对数据线, TXD 称发送数据输出, RXD 称接收数据输入。当两台微机以全双工方式直接通信(无 MODEM 方式)时,双方的这两根线应交叉联接(扭接)。

信号地 所有的信号都要通过信号地线构成耦合回路。通信线有以上三条(TXD、RXD 和信号地)就能工作了。其余信号主要用于双方设备通信过程中的联络(握手信号),而且有些信号仅用于和 MODEM 的联络。若采取微型机对微型机直接通信,且双方可直接对异步串行通信电路芯片编程,若设置成不要任何联络信号,则其它线都可可不接。有时在通信线的同一端将相关信号短接以“自握手”方式满足联络要求。这就是如图 3-2(a)所示的情况。

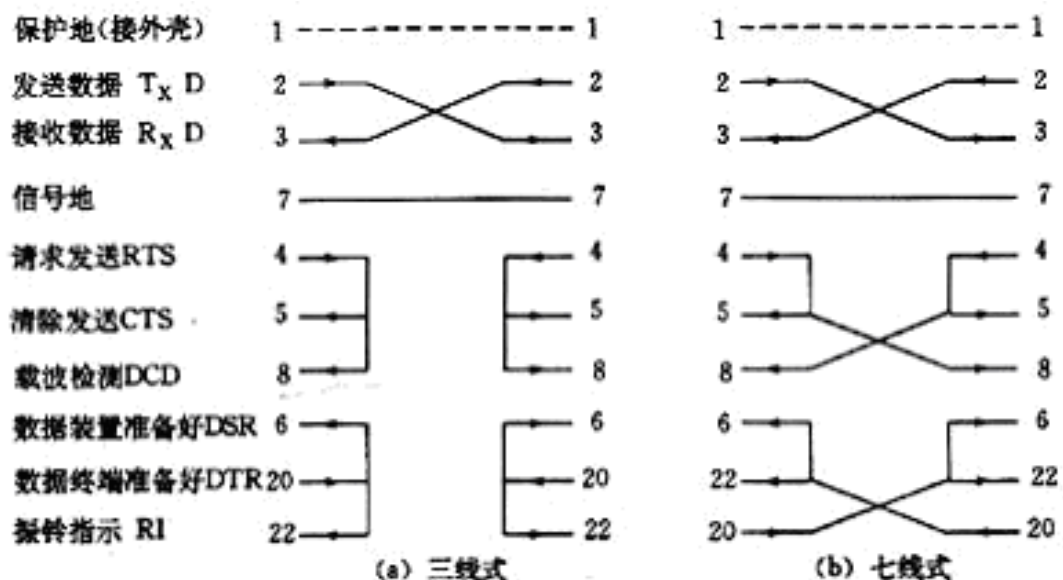


图 3-2 实用 RS-232C 连线

RTS / CTS 请求发送值号 RTS 是发送器输出的准备好信号。接收方准备好后送回清除

发送信号 CTS 后, 发送数据开始进行, 在同一端将这两个信号短接就意味着只要发送器准备好即可发送。

DCD 载波检测(又称接收线路信号检测)。本意是 MODEM 检测到线路中的载波信号后, 通知终端准备接收数据的信号, 在没有接 MODEM 的情况下, 也可以和 RTS、CTS 短接。

相对于 MODEM 而言, 微型机和终端机一样被称为数据终端 DTE(Data Terminal Equipment)而 MODEM 被称为数据通信装置 DCE(Data Communications Equipment), DTE 和 DCE 之间的连接不能像图 3-2 中有“扭接”现象, 而应该是按接插件芯号, 同名端对应相接。此处介绍的 RS-232C 的信号名称及信号流向都是对 DTE 而言的。

DTR / DSR 数据终端准备好时发 DTR 信号, 在收到数据通信装置准备好 DSR 信号后, 方可通信。图 3-2(a)中将这一对信号以“自握手”方式短接。

RI 原意是在 MODEM 接收到电话交换机有效的拨号时, 使 RI 有效, 通知数据终端准备传送。在无 MODEM 时也可和 DTR 相接。

图 3-2(b)给出了无 MODEM 情况下, DTE 对 DTE 异步串行通信线路的完整连接, 它不仅适用于微型机和微型机之间的通信, 还适用于微型机和异步串行外部设备(如终端机、绘图仪、数字化仪等)的连接。

(2) 信号电平规定

RS-232C 规定了双极性的信号逻辑电平:

-3V 到-25V 之间的电平表示逻辑“1”。

+3V 到+25V 之间的电平表示逻辑“0”。

因此这是一套负逻辑定义。

以上标准称为 EIA 电平。PC / XT 系列使用的信号电平是-12V 和+12V, 符合 EIA 标准, 但在计算机内部流动的信号都是 TTL 电平, 因此这中间需要用电平转换电路。常用芯片 MC1488 或 SN75150 将 TTL 电平转换为 EIA 电平, MC1489 或 SN75154 将 EIA 电平转换为 TTL 电平。PC / XT 系列以这种方式进行串行通信时, 在波特率不高于 9600 的情况下, 理论上通信线的长度限制组为 15 米。

2) 20mA 电流环

20mA 电流环并没有形成一套完整的标准, 主要是将数字信号的表示方法不使用电子的高低, 而改用 20mA 电流的有无: “1”信号在环路中产生 20mA 电流; “0”信号无电流产生。当然也需要有电路来实现 TTL 电平和 20mA 电流之间的转换。图 3-3 是 PC / XT 微机中使用的一种 20mA 电流环接口。当发送方 $S_{OUT}=1$ 时, 便有 20mA 电流灌入接收方的光耦合器, 于是光耦合器导通, 使 $S_{IN}=1$ 。反之当发送方 $S_{OUT}=0$ 时环路电流为零, 接收方光耦合器截止, $S_{IN}=0$ 。显然, 当要求双工方式通信时, 双方都应各有收发电路, 通信连线至少要 4 根。由于通信双方利用光耦合器实现电气上隔离, 而且信号又是双端回路方式, 故有很强的抗干扰性, 可以传送远至 1 千米的距离。

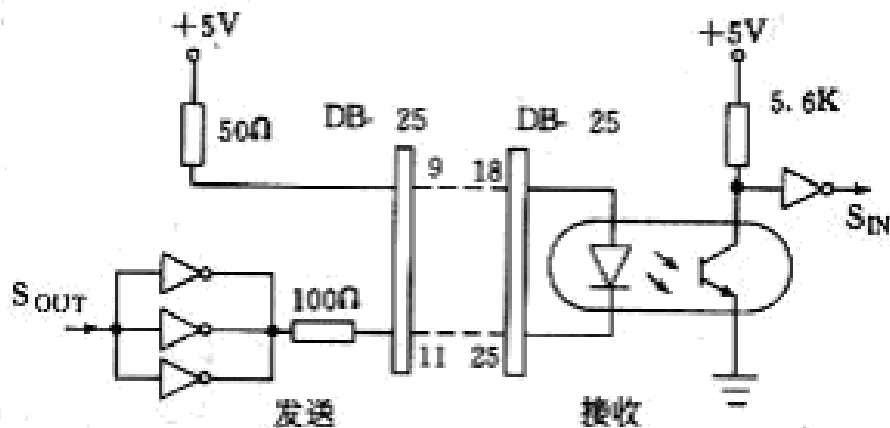


图 3-3 20mA 电流环接口

“0”、“1”信号的表示方法不同外，其他方面(如字符的传输格式)常借用 RS—232C 标准。因此 PC / XT 微机中的异步串行信道接口往往将这两种标准做在一起，实际通过跨接线从二者中择一使用。

ARM 自带三个 UART 端口，每个 UART 通道都有 16 字节的 FIFO（先入先出寄存器）用于接受和发送。用系统时钟最大波特率可达 230.4K，如果用外部时钟(UCLK)UART 可以以更高的波特率运行。

S3C2410X UART 包括可编程波特率，红外发送/接收，插入一个或两个停止位，5 字节，6 字节，7 字节，或 8 字节数据宽度和奇偶校验。

其特点是：

- 基于 DMA 或者中断操作的 RxD0, TxD0, RxD1, TxD1, RxD2, TxD2。
- 包括 IrDA 1.0 和 16 字节 FIFO 的 UART 通道 0, 1, 2。
- 包括 nRTS0, nCTS0, nRTS1 和 nCTS1 的 UART 通道。
- 支持握手方式的接收/发送

与 UART 有关的寄存器主要有以下几个：

- (1) UART 线控制寄存器包括 ULCON0, ULCON1 和 ULCON2，主要用来选择每帧数据位数、停止位数，奇偶校验模式及是否使用红外模式，如表 3-1，3-2 所示。

表 3-1 UART 寄存器设置

Register	Address	R/W	Description	Reset Value
ULCON0	0x50000000	R/W	UART channel 0 line control register	0x00
ULCON1	0x50004000	R/W	UART channel 1 line control register	0x00
ULCON2	0x50008000	R/W	UART channel 2 line control register	0x00

表 3-2 UART 寄存器位描述

ULCONn	Bit	Description	Initial State
--------	-----	-------------	---------------

Reserved	[7]		0
Infra-Red Mode	[6]	Determine whether or not to use the Infra-Red mode. 0 = Normal mode operation 1 = Infra-Red Tx/Rx mode	0
Parity Mode	[5:3]	Specify the type of parity generation and checking during UART transmit and receive operation. 0xx = No parity 100 = Odd parity 101 = Even parity 110 = Parity forced/checked as 1 111 = Parity forced/checked as 0	000
Number of Stop Bit	[2]	Specify how many stop bits are to be used for end-of-frame signal. 0 = One stop bit per frame 1 = Two stop bit per frame	0
Word Length	[1:0]	Indicate the number of data bits to be transmitted or received per frame. 00 = 5-bits 01 = 6-bits 10 = 7-bits 11 = 8-bits	00

- (2) UART控制寄存器包括UCON0, UCON1 and UCON2, 主要用来选择时钟, 接收和发送中断类型 (即电平还是脉冲触发类型), 接收超时使能, 接收错误状态中断使能, 回环模式, 发送接收模式等。如表3-3, 3-4所示

表3-3 UART控制寄存器设置

Register	Address	R/W	Description	Reset Value
UCON0	0x50000004	R/W	UART channel 0 control register	0x00
UCON1	0x50004004	R/W	UART channel 1 control register	0x00
UCON2	0x50008004	R/W	UART channel 2 control register	0x00

表3-4 UART控制寄存器位描述

UCONn	Bit	Description	Initial State
Clock Selection	[1:0]	Select PCLK or UCLK for the UART baud rate. 0=PCLK : $UBRDIVn = (int)(PCLK / (bps \times 16)) - 1$ 1=UCLK (@GPH8) : $UBRDIVn = (int)(UCLK / (bps \times 16)) - 1$	0
Tx Interrupt Type	[9]	Interrupt request type. 0 = Pulse (Interrupt is requested as soon as the Tx buffer becomes empty in Non-FIFO mode or reaches Tx FIFO Trigger Level in FIFO mode.) 1 = Level (Interrupt is requested while Tx buffer is empty in Non-FIFO mode or reaches Tx FIFO Trigger Level in FIFO mode.)	0

Rx Interrupt Type	[8]	Interrupt request type. 0 = Pulse (Interrupt is requested the instant Rx buffer receives the data in Non-FIFO mode or reaches Rx FIFO Trigger Level in FIFO mode.) 1 = Level (Interrupt is requested while Rx buffer is receiving data in Non-FIFO mode or reaches Rx FIFO Trigger Level in FIFO mode.)	0
Rx Time Out Enable	[7]	Enable/Disable Rx time out interrupt when UART FIFO is enabled. The interrupt is a receive interrupt. 0 = Disable 1 = Enable	0
Rx Error Status Interrupt Enable	[6]	Enable 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 = Do not generate receive error status interrupt. 1 = Generate receive error status interrupt.	0
Loopback Mode	[5]	Setting loopback bit to 1 causes the UART to enter the loopback mode. This mode is provided for test purposes only. 0 = Normal operation 1 = Loopback mode	0
Send Break Signal	[4]	Setting this bit causes the UART to send a break during 1 frame time. This bit is automatically cleared after sending the break signal. 0 = Normal transmit 1 = Send break signal	0
Transmit Mode	[3:2]	Determine which function is currently able to write Tx data to the UART transmit buffer register. 00 = Disable 01 = Interrupt request or polling mode 10 = DMA0 request (Only for UART0), DMA3 request (Only for UART2) 11 = DMA1 request (Only for UART1)	00
Receive Mode	[1:0]	Determine which function is currently able to read data from UART receive buffer register. 00 = Disable 01 = Interrupt request or polling mode 10 = DMA0 request (Only for UART0), DMA3 request (Only for UART2) 11 = DMA1 request (Only for UART1)	00

(3) UART错误状态寄存器包括UERSTAT0, UERSTAT1 and UERSTAT2, 此状态寄存器的相关位表明是否有帧错误或溢出错误发生。如表3-5, 3-6所示

表3-5 UART错误状态寄存器设置

Register	Address	R/W	Description	Reset Value
UERSTAT0	0x50000014	R	UART channel 0 Rx error status register	0x0
UERSTAT1	0x50004014	R	UART channel 1 Rx error status register	0x0
UERSTAT2	0x50008014	R	UART channel 2 Rx error status register	0x0

表3-6 UART错误状态寄存器位描述

UERSTATn	Bit	Description	Initial State
Reserved	[3]	0 = No frame error during receive 1 = Frame error (Interrupt is requested.)	0
Frame Error	[2]	Set to 1 automatically whenever a frame error occurs during receive operation. 0 = No frame error during receive 1 = Frame error (Interrupt is requested.)	0
Reserved	[1]	0 = No frame error during receive 1 = Frame error (Interrupt is requested.)	0
Overrun Error	[0]	Set to 1 automatically whenever an overrun error occurs during receive operation. 0 = No overrun error during receive 1 = Overrun error (Interrupt is requested.)	0

注意：在读取UART错误状态寄存器时，这些位(UERSTATn[3:0])会自动清零。

() 在 UART 模块中有三个接收/发送状态寄存器，包括 UTRSTAT0，UTRSTAT1 和 UTRSTAT2。如表 3-7，3-8 所示：

表 3-7 UART 接收/发送寄存器设置

Register	Address	R/W	Description	Reset Value
UTRSTAT0	0x50000010	R	UART channel 0 Tx/Rx status register	0x6
UTRSTAT1	0x50004010	R	UART channel 1 Tx/Rx status register	0x6
UTRSTAT2	0x50008010	R	UART channel 2 Tx/Rx status register	0x6

表 3-8 UART 接收/发送寄存器位描述

UTRSTATn	Bit	Description	Initial State
Transmitter empty	[2]	Set to 1 automatically when the transmit buffer register has no valid data to transmit and the transmit shift register is empty. 0 = Not empty 1 = Transmitter (transmit buffer & shifter register) empty	1
Transmit buffer empty	[1]	Set to 1 automatically when transmit buffer register is empty. 0 =The buffer register is not empty 1 = Empty (In Non-FIFO mode, Interrupt or DMA is requested. In FIFO mode, Interrupt or DMA is requested, when Tx FIFO Trigger Level is set to 00 (Empty)) If the UART uses the FIFO, users should check Tx FIFO Count bits and Tx FIFO Full bit in the UFSTAT register instead of this bit.	1

Receive buffer data ready	[0]	Set to 1 automatically whenever receive buffer register contains valid data, received over the RXDn port. 0 = Empty 1 = The buffer register has a received data (In Non-FIFO mode, Interrupt or DMA is requested) If the UART uses the FIFO, users should check Rx FIFO Count bits and Rx FIFO Full bit in the UFSTAT register instead of this bit.	0
---------------------------	-------------	---	---

()UERSTAT0, UERSTAT1 and UERSTAT2

(2) 在 UART 模块中有 3 个 UART 发送缓冲寄存器, 包括 UTXH0, UTXH1 和 UTXH2, UTXHn 有 8 位发送数据。如下表 3-9:

表3-9 UART发送缓冲寄存器

Register	Address	R/W	Description	Reset Value
UTXH0	0x50000020 (L) 0x50000023 (B)	W (by byte)	UART channel 0 transmit buffer register	-
UTXH1	0x50004020 (L) 0x50004023 (B)	W (by byte)	UART channel 1 transmit buffer register	-
UTXH2	0x50008020 (L) 0x50008023 (B)	W (by byte)	UART channel 2 transmit buffer register	-

其功能如下表3-10所示:

表3-10 UART发送缓冲寄存器功能

UTXHn	Bit	Description	Initial State
TXDATAn	[7:0]	Transmit data for UARTn	-

注意: (L):小端模式, (B):大端模式

(3) 在 UART 模块中有 3 个 UART 接收缓冲寄存器, 包括 URXH0, URXH1 和 URXH2, URXHn 有 8 位接收数据。如表 3-11:

表3-11 UART接收缓冲寄存器

Register	Address	R/W	Description	Reset Value
----------	---------	-----	-------------	-------------

URXH0	0x50000024 (L) 0x50000027 (B)	R (by byte)	UART channel 0 receive buffer register	-
URXH1	0x50004024 (L) 0x50004027 (B)	R (by byte)	UART channel 1 receive buffer register	-
URXH2	0x50008024 (L) 0x50008027 (B)	R (by byte)	UART channel 2 receive buffer register	-

其功能如下表3-12所示：

表3-12 UART接收缓冲寄存器功能

URXHn	Bit	Description	Initial State
RXDATAn	[7:0]	Receive data for UARTn	-

注意：当发生溢出错误时，必须读URXHn。否则，即使UERSTATn的溢出位已经清零，下个已接收数据也会产生溢出错误。

(C) UART 波特率因子寄存器

表3-13 UART波特率因子寄存器设置

Register	Address	R/W	Description	Reset Value
UBRDIV0	0x50000028	R/W	Baud rate divisor register 0	-
UBRDIV1	0x50004028	R/W	Baud rate divisor register 1	-
UBRDIV2	0x50008028	R/W	Baud rate divisor register 2	-

表3-14 UART波特率因子寄存器功能

UBRDIVn	Bit	Description	Initial State
UBRDIV	[15:0]	Baud rate division value	-
		UBRDIVn >0	

UART包括三个波特率因子寄存器UBRDIV0, UBRDIV1 and UBRDIV2，存储在波特率因子寄存器(UBRDIVn)中的值决定串口发送和接收的时钟数率（波特率），计算公式如下：

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

或

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

例如：如果波特率是115200，PCLK or或UCLK is是40 MHz，那么UBRDIVn：

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

六、实验步骤

1. 编写串口驱动函数

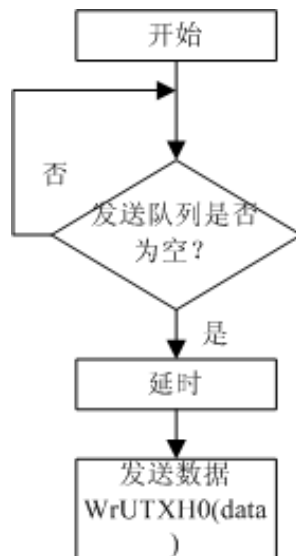


图 3-4 发送数据

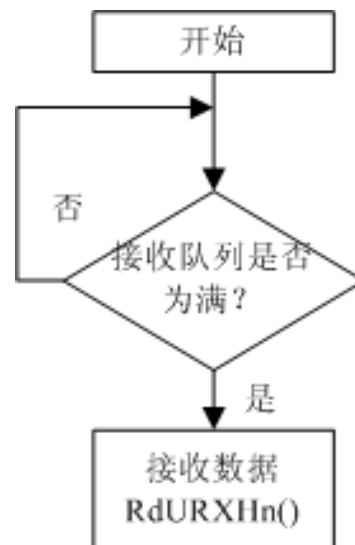


图 3-5 接收数据

2. 在主函数中实现将从串口 0 接收到的数据发送到串口 0 (Main.c):

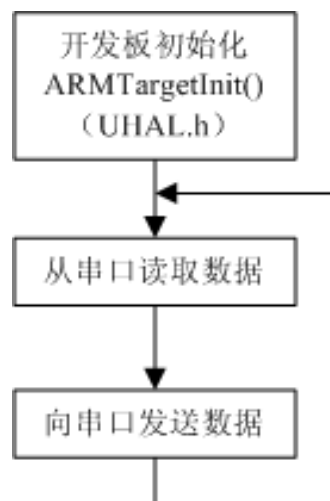


图 3-6 主函数

3. 启动 ARM JTAG 仿真器并进行初始化配置。
4. 启动 EWARM 新建工程，将“Exp1 ARM 串口实验”中的文件添加到工程中并调试运行。

七、思考题

1. 232 串行通讯的数据格式是什么？

2. 串行通讯最少需要几根线，分别如何连接？
3. ARM 的串行口有几个，相应的寄存器是什么？
4. 用中断方式实现串口驱动。

实验二 LED 点灯实验

一、实验目的

1. 了解 ARM 设备外围电路结构与接口原理。
2. 编程实现对嵌入式设备上 LED 灯的控制。

二、实验内容

学习 ARM 外围电路接口基本原理，熟悉编写嵌入式 C 程序实现对外围设备的控制。

三、预备知识

- 1、用 EWARM 集成开发环境，编写和调试程序的基本过程。
- 2、ARM 应用程序的框架结构。

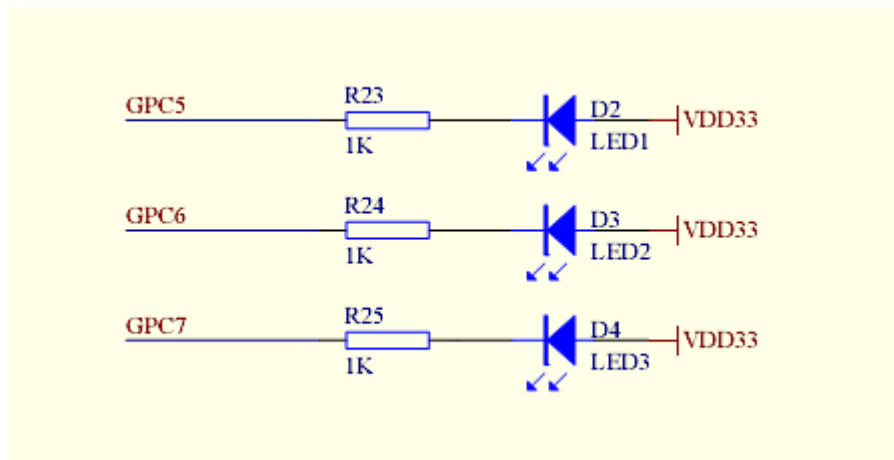
四、实验设备及工具

硬件：ARM 嵌入式开发平台、用于 ARM920T 的 JTAG 仿真器、PC 机 Pentium100 以上。

软件：PC 机操作系统 Win2000 或 WinXP、EWARM 集成开发环境、仿真器驱动程序、超级终端通讯程序

五、实验原理及说明

1. 电路接口



从电路上可以看出 LED1~LED3 分别接到 ARM 的 GPC5~GPC7 I/O 上，LED 共阳极，因此低电平发光，高电平点亮。

此外处理器的 GPIO 作为控制 I/O 要进行必要的设置才能对外设进行正确控制，此实验将相应 I/O 设置为输出模式，并向相应 I/O 数据寄存器进行写入便可控制 LED 的开关。如图给出的 ARM2410GPIO 寄存器配置：

Port C	Selectable Pin Functions		
GPC15	Input/output	<u>VD7</u>	—
GPC14	Input/output	<u>VD6</u>	—
GPC13	Input/output	<u>VD5</u>	—
GPC12	Input/output	<u>VD4</u>	—
GPC11	Input/output	<u>VD3</u>	—
GPC10	Input/output	<u>VD2</u>	—
GPC9	Input/output	<u>VD1</u>	—
GPC8	Input/output	<u>VD0</u>	—
GPC7	Input/output	<u>LCDVF2</u>	—
GPC6	Input/output	<u>LCDVF1</u>	—
GPC5	Input/output	<u>LCDVF0</u>	—
GPC4	Input/output	<u>VM</u>	—
GPC3	Input/output	<u>VFRAME</u>	—
GPC2	Input/output	<u>VLINE</u>	—
GPC1	Input/output	<u>VCLK</u>	—
GPC0	Input/output	<u>LEND</u>	—

PORT C CONTROL REGISTERS (GPCCON, GPCDAT, and GPCUP)

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

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

在主函数中编写 LED 点亮及熄灭函数

```
void Led_on()
{
    rGPCDAT&=~0xe0;// Led on and pouese low
    Uart_Printf(0,"Led light on...");
    Uart_Printf(0,"\n");
}
```

```
void Led_off()
{
    rGPCDAT|=0xe0;// Led off and pouse high
}
```

在主函数中初始化相关硬件：

```
    ARMTargetInit();          // do target (uHAL based ARM system) initialisation //
    rGPCCON=0x5400;           // GPC5~7 OUT
设置 GPIO 为输出模式
```

六、实验步骤

1. 启动 H-JTAG 仿真器并进行初始化配置。
2. 打开 Exp2 LED 点灯实验工程。
3. 编译并下载程序到开发板观察实验现象

实验三 按键中断实验

一、实验目的

1. 掌握 ARM 的中断原理。
2. 会使用 ARM 中断实现按键中断响应。

二、实验现象

按动 5 向中断键实现从串口 0 输出相关中断响应提示信息

三、预备知识

- 1、用 EWARM 集成开发环境，编写和调试程序的基本过程。
- 2、阅读 S3C2410 处理器手册的中断部分文档

四、实验设备及工具

硬件：ARM 嵌入式开发平台、PC 机 Pentium100 以上、用于 ARM920T 的 JTAG 仿真器、串口线。

软件：PC 机操作系统 Win2000 或 WinXP、ARM ADS1.2 集成开发环境、仿真器驱动程序、超级终端通讯程序。

五、实验原理及说明

1. 中断

S3C2410 支持 56 个中断源，两种中断模式，一种是 IRQ，一种是 FIQ。我们通常只用 IRQ 模式。由中断模式寄存器来决定选用哪种中断模式。这些中断可以同时启动也可以只启动单独的几个，当启动多个中断的时候有中断优先级寄存器来安排中断发生的顺序。通常没有特殊的需求我们不改变中断的优先级，而采用默认的中断优先顺序向 CPU 申请中断。不屏蔽的中断当中断发生时都会向 CPU 申请中断。因为有多多个中断源，当中断发生时通过查询中断偏移寄存器来知道哪个中断源发生的中断，也是通过此偏移寄存器的值来调用不同的中断服务程序。上面描述了中断中使用的寄存器的关系，下面给出这些寄存器的关系图如图 1

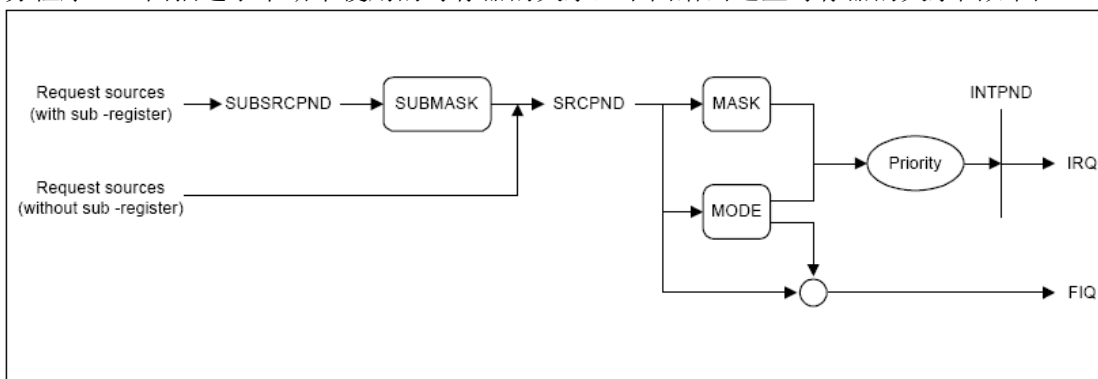


图 1

下面我们以定时器中断为例，说明我们使用的平台发生中断时的程序处理流程，可以以此程序为模板，通过修改中断服务程序定制自己的中断。中断的流程有两条主线，一是当整个程序初始化时即没有发生中断以前的中断初始化流程为一条；另外一条是当程序发生中断时的程序执行流程。

以程序初始化为主线的流程如下：

第一步：初始化中断。设置中断为 IRQ 模式，屏蔽所有的中断，包括子中断。

第二步：设置中断服务程序，并开启相应的中断。设置中断服务程序就是将中断偏移寄存器中的值和具体的中断服务程序连接起来，使得当获得中断偏移寄存器中的值以后可以知道调用哪个中断服务程序。开启相应的中断就是关闭相应的中断屏蔽。因为这里的中断是定时器中断，所以在设置中断服务程序的同时要开启定时器。并在此之前要初始化定时器以等待启动定时器。至此中断全部设置完成。当中断发生时就可以自动调用相应的中断服务程序，完成中断服务程序以后自动返回被中断的程序继续执行。

以发生中断时为主线的流程如下：

第一步：保护现场。

第二步：查找中断源并调用相应的中断服务程序。通过查找中断偏移寄存器获得中断源，在以程序初始化为主线的流程中的第二步中已设置了中断源的中断服务程序。

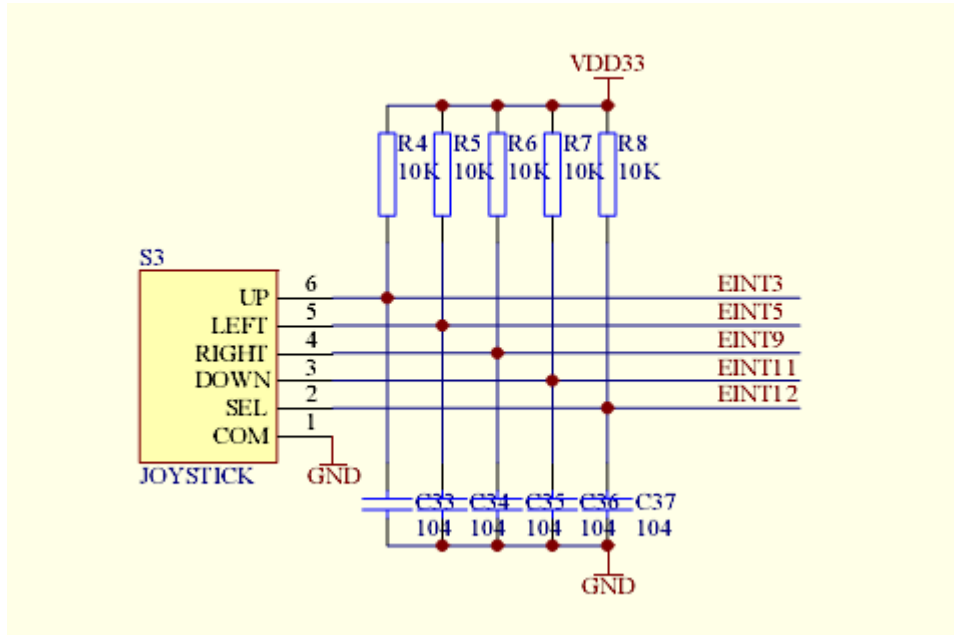
第三步：恢复现场并返回现场。

值得注意的是:在执行中断服务程序前要关闭中断。执行完后清除刚处理过的中断的寄存器的相应位并开启中断。

3. 使用方法

用仿真器跟踪程序的执行，具体了解两条主线的执行过程。在程序执行的过程中查找 S3C2410 处理器手册，了解代码对寄存器作了哪些设置及为什么作此设置。

5. 按键电路连接原理图：



从图中可以很得知，中断按键一共用到了 ARM 处理器的 5 个外部中断分别是 EINT3、EINT5、EINT9、EINT11、EINT12。

分别配置相关 I/O 为中断功能并关联响应的中断处理函数：

```
void start_External_interrupt(void)
{
```

```
    set_external_irq(IRQ_EINT3, 2, 0);
    set_external_irq(IRQ_EINT12, 2, 0);
    set_external_irq(IRQ_EINT9, 2, 0);
    set_external_irq(IRQ_EINT11, 2, 0);
    set_external_irq(IRQ_EINT5, 2, 0);
```

```
    SetISR_Interrupt(IRQ_EINT3, (Interrupt_func_t)external_interrupt_up, NULL);
    SetISR_Interrupt(IRQ_EINT5, (Interrupt_func_t)external_interrupt_left, NULL);
    SetISR_Interrupt(IRQ_EINT9, (Interrupt_func_t)external_interrupt_right, NULL);
    SetISR_Interrupt(IRQ_EINT11, (Interrupt_func_t)external_interrupt_down, NULL);
    SetISR_Interrupt(IRQ_EINT12, (Interrupt_func_t)external_interrupt_sel, NULL);
```

```
}
```

set_external_irq()函数是设置处理器 GPIO 功能函数，SetISR_Interrupt()函数为关联相关中断源与中断处理函数。

六、实验步骤

1. 启动 H-JTAG 仿真器并进行初始化配置。
2. 打开 Exp3 按键中断实验的工程。
3. 编译并下载程序到开发板观察实验现象。

实验四 触摸屏驱动实验

一、实验目的

1. 了解触摸屏基本概念与原理。
2. 理解触摸屏与 LCD 的密切配合。
3. 编程实现对触摸屏的控制。

二、实验内容

学习触摸屏基本原理，理解对触摸屏进行输出标定、与 LCD 显示器配合的过程。

三、预备知识

- 1、用 EWARM 集成开发环境，编写和调试程序的基本过程。
- 2、ARM 应用程序的框架结构。

四、实验设备及工具

硬件：ARM 嵌入式开发平台、用于 ARM920T 的 JTAG 仿真器、PC 机 Pentium100 以上。

软件：PC 机操作系统 Win2000 或 WinXP、EWARM 集成开发环境、仿真器驱动程序、超级终端通讯程序

五、实验原理及说明

1. 触摸屏原理

触摸屏按其工作原理的不同分为表面声波屏、电容屏、电阻屏和红外屏几种。常见的有电阻触摸屏。

如图 3-20 所示，电阻触摸屏的屏体部分是一块与显示器表面非常配合的多层复合薄膜，由一层玻璃或有机玻璃作为基层，表面涂有一层透明的导电层，上面再盖有一层外表面硬化处理、光滑防刮的塑料层，它的内表面也涂有一层透明导电层，在两层导电层之间有许多细小(小于千分之一英寸)的透明隔离点把它们隔开绝缘。

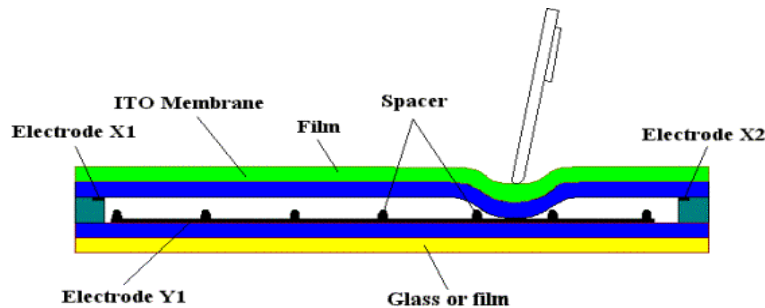


图 3-20 （北泰）触摸屏的结构

如图 3-21 所示，当手指或笔触显示屏时(图 c)，平常相互绝缘的两层导电层就在触摸点位置有了一个接触，因其中一面导电层（顶层）接通 X 轴方向的 5V 均匀电压场(图 a)，使得检测层（底层）的电压由零变为非零，控制器侦测到这个接通后，进行 A/D 转换，并将得到的电压值与 5V 相比即可得触摸点的 X 轴坐标为（原点在靠近接地点的那端）：

$$X_i = L_x * V_i / V \quad (\text{即分压原理})$$

同理得出 Y 轴的坐标，这就是所有电阻触摸屏共同的最基本原理。

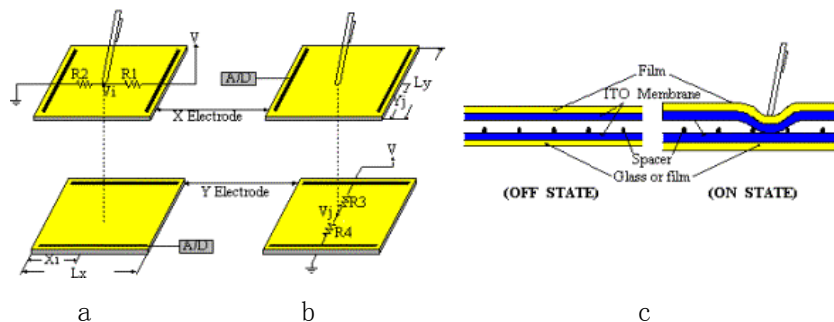


图 3-21 触摸屏坐标识别原理

2. 电阻触摸屏的有关技术

电阻触摸屏的主要部分是一块与显示器表面非常配合的电阻薄膜屏，这是一种多层的复合薄膜，由一层玻璃或有机玻璃作为基层，表面涂有一层叫 ITO 的透明导电层，上面再盖有一层外表面硬化处理、光滑防刮的塑料层，它的内表面也涂有一层导电层（ITO 或镍金）。电阻触摸屏的两层 ITO 工作面必须是完整的，在每个工作面的两条边线上各涂一条银胶，一端加 5V 电压，一端加 0V，就能在工作面的一个方向上形成均匀连续的平行电压分布。在侦测到有触摸后，立刻 A/D 转换测量接触点的模拟量电压值，根据 5V 电压下的等比例公式就能计算出触摸点在这个方向上的位置。

透明的导电涂层材料有两种：

1) ITO, 氧化铟, 弱导体, 特性是当厚度降到 1800 埃以下时会突然变得透明, 透光度为 80%, 再薄下去透光率反而下降, 到 300 埃厚度时又上升到 80%。但遗憾的是 ITO 在这个厚度下非常脆, 容易折断产生裂纹。ITO 是所有电阻触摸屏及电容触摸屏都用到的主要材料, 实际上电阻和电容触摸屏的工作面就是 ITO 涂层。

2) 镍金涂层, 五线电阻触摸屏的外层导电层使用的是延展性极好的镍金涂层材料, 外层导电层由于频繁触摸, 使用延展性好的镍金材料目的是为了延长使用寿命, 但是成本较高, 镍金导电层虽然延展性好, 但是只能作透明导体, 不适合作为电阻触摸屏的工作面, 因为它导电性太好, 不宜作精密电阻测量, 而且金属不易做到厚度非常均匀。

第一代四线触摸屏两层 ITO 工作面工作时都加上 5V 到 0V 的均匀电压分布场: 一个工作面加竖直方向的, 一个工作面加水平方向的。引线至控制器总共需要四根电缆。因为四线电阻触摸屏靠外的外层塑胶及 ITO 涂层被经常触动, 一段时间后外层薄薄的 ITO 涂层就会产生细小的裂纹, 导电工作面一旦有了裂纹, 电流就会绕之而过, 工作面上的电压场分布也就不可能再均匀, 这样, 在裂纹附近触摸屏漂移严重, 裂纹增多后, 触摸屏有些区域可能就再也触摸不到了。

四线电阻触摸屏的基层大多数是有机玻璃, 不仅存在透光率低、风化、老化的问题, 并且存在安装风险, 这是因为有机玻璃刚性差, 安装时不能捏边上的银胶, 以免薄薄的 ITO 和相对厚实的银胶脱裂, 不能用力压或拉触摸屏, 以免拉断 ITO 层。有些四线电阻触摸屏安装后显得不太平整就是因为这个原因。

ITO 是无机物, 有机玻璃是有机物, 有机物和无机物是不能良好结合的, 时间一长就容易剥落。如果能够生产出曲面的玻璃板, 玻璃是无机物, 能和 ITO 非常好的结合为导电玻璃, 这样电阻触摸屏的寿命能够大大延长。

第二代五线电阻触摸屏的基层使用的就是这种导电玻璃, 不仅如此, 五线电阻技术把两个方向的电压场通过精密电阻网络都加在玻璃的导电工作面上, 我们可以简单的理解为两个方向的电压场分时加在同一工作面上, 而外层镍金导电层仅仅用来当作纯导体, 有触摸后靠既检测内层 ITO 接触点电压又检测导通电流的方法测得触摸点的位置。五线电阻触摸屏内层 ITO 需四条引线, 外层只作导体仅仅一条, 至控制器总共需要 5 根电缆。因为五线电阻屏的外层镍金导电层不仅延展性好, 而且只作导体, 只要它不断成两半, 就仍能继续完成作为导体的使命, 而身负重任的内层 ITO 直接与基层玻璃结合为一体成为导电玻璃, 导电玻璃自然没有了有机玻璃作基层的种种弊端, 因此, 五线电阻屏的使用寿命和透光率与四线电阻屏相比有了一个飞跃: 五线电阻屏的触摸寿命是 3 千 5 百万次, 四线电阻屏则小于 1 百万次, 且五线电阻触摸屏没有安装风险, 同时五线电阻屏的 ITO 层能做得更薄, 因此透光率和清晰度更高, 几乎没有色彩失真。

不管是四线电阻触摸屏还是五线电阻触摸屏, 它们都是一种对外界完全隔离的工作环境, 不怕灰尘、水汽和油污, 它可以用任何物体来触摸, 可以用来写字画画, 比较适合工业控制领域及办公室使用。电阻触摸屏共同的缺点是因为复合薄膜的外层采用塑胶材料, 不知

道的人太用力或使用锐器触摸可能划伤整个触摸屏而导致报废。不过，在限度之内，划伤只会伤及外导电层，外导电层的划伤对于五线电阻触摸屏来说没有关系，而对四线电阻触摸屏来说是致命的。

3. 触摸屏的控制

本系统触摸屏的控制是使用的 S3c2410 处理器自带的触摸屏控制器，这部分开发主要参考 S3c2410 处理器的芯片手册的第 416 页到第 419 页，具体详见流程图。这部分的控制主要是设置触摸屏的采样模式，处理器提供的模式：

1. 正常的转换模式
2. 手动的 x/y 位置转换模式
3. 自动的 x/y 位置转换模式

我们这里使用的是第 3 种转换模式。需要注意的是在完成一次 x/y 坐标采样的过程中需要一次模式转换即在点击触摸屏之前是等待中断模式，当有触摸动作产生触摸屏中断以后，在 x/y 的坐标采集驱动中设置成自动的 x/y 位置转换模式，在完成采集以后再转换回等待中断模式，准备下一次的触摸采样。要用到的控制器如下：

ADC控制寄存器

Register	Address	R/W	Description	Reset Value
ADCCON	0x58000000	R/W	ADC control register	0x3FC4

ADCCON	Bit	Description	Initial State
ECFLG	[15]	End of conversion flag (read only). 0 = A/D conversion in process 1 = End of A/D conversion	0
PRSCEN	[14]	A/D converter prescaler enable. 0 = Disable 1 = Enable	0
PRSCVL	[13:6]	A/D converter prescaler value. Data value: 1 ~ 255 Note that division factor is (N+1) when the prescaler value is N.	0xFF
SEL_MUX	[5:3]	Analog input channel select. 000 = AIN 0 001 = AIN 1 010 = AIN 2 011 = AIN 3 100 = AIN 4 101 = AIN 5 110 = AIN 6 111 = AIN 7 (XP)	0
STDBM	[2]	Standby mode select. 0 = Normal operation mode 1 = Standby mode	1

READ_START	[1]	A/D conversion start by read. 0 = Disable start by read operation 1 = Enable start by read operation	0
ENABLE_START	[0]	A/D conversion starts by setting this bit. If READ_START is enabled, this value is not valid. 0 = No operation 1 = A/D conversion starts and this bit is cleared after the start-up.	0

ADC触摸屏控制寄存器

Register	Address	R/W	Description	Reset Value
ADCTSC	0x58000004	R/W	ADC touch screen control register	0x058

ADCTSC	Bit	Description	Initial State
Reserved	[8]	This bit should be zero.	0
YM_SEN	[7]	Select output value of YMON. 0 = YMON output is 0 (YM = Hi-Z). 1 = YMON output is 1 (YM = GND).	0
YP_SEN	[6]	Select output value of nYPON. 0 = nYPON output is 0 (YP = External voltage). 1 = nYPON output is 1 (YP is connected with AIN[5]).	1
XM_SEN	[5]	Select output value of XMON. 0 = XMON output is 0 (XM = Hi-Z). 1 = XMON output is 1 (XM = GND).	0
XP_SEN	[4]	Select output value of nXPON. 0 = nXPON output is 0 (XP = External voltage). 1 = nXPON output is 1 (XP is connected with AIN[7]).	1
PULL_UP	[3]	Pull-up switch enable. 0 = XP pull-up enable 1 = XP pull-up disable	1
AUTO_PST	[2]	Automatically sequencing conversion of X-position and Y-position 0 = Normal ADC conversion 1 = Auto (Sequential) X/Y Position Conversion Mode	0
XY_PST	[1:0]	Manual measurement of X-position or Y-position. 00 = No operation mode 01 = X-position measurement 10 = Y-position measurement 11 = Waiting for Interrupt Mode	0

注意：在自动模式，ADC触摸屏控制寄存器要在开始读之前重新配置

ADC开始延迟寄存器

Register	Address	R/W	Description	Reset Value
ADCDLY	0x58000008	R/W	ADC start or interval delay register	0x00ff

ADCDLY	Bit	Description	Initial State
DELAY	[15:0]	1) Normal Conversion Mode, Separate X/Y Position Conversion Mode, and Auto (Sequential) X/Y Position Conversion Mode. →X/Y Position Conversion Delay Value. 2) Waiting for Interrupt Mode.	00ff
		When Stylus down occurs in Waiting for Interrupt Mode, this register generates Interrupt signal (INT_TC) at intervals of several ms for Auto X/Y Position conversion. NOTE: Do not use Zero value (0x0000)	

ADC转换数据寄存器（ADCDAT0）

Register	Address	R/W	Description	Reset Value
ADCDAT0	0x5800000C	R	ADC conversion data register	-

ADCDAT0	Bit	Description	Initial State
UPDOWN	[15]	Up or down state of Stylus at Waiting for Interrupt Mode. 0 = Stylus down state 1 = Stylus up state	-
AUTO_PST	[14]	Automatic sequencing conversion of X-position and Y-position. 0 = Normal ADC conversion 1 = Sequencing measurement of X-position, Y-position	-
XY_PST	[13:12]	Manual measurement of X-position or Y-position. 00 = No operation mode 01 = X-position measurement 10 = Y-position measurement 11 = Waiting for Interrupt Mode	-
Reserved	[11:10]	Reserved	
XPDATA (Normal ADC)	[9:0]	X-position conversion data value. (include Normal ADC conversion data value) Data value: 0 ~ 3FF	-

ADC转换数据寄存器（ADCDAT1）

Register	Address	R/W	Description	Reset Value
ADCDAT1	0x58000010	R	ADC conversion data register	-

ADCDAT1	Bit	Description	Initial State
---------	-----	-------------	---------------

UPDOWN	[15]	Up or down state of Stylus at Waiting for Interrupt Mode. 0 = Stylus down state 1 = Stylus up state	-
AUTO_PST	[14]	Automatically sequencing conversion of X-position and Y-position. 0 = Normal ADC conversion	-
		1 = Sequencing measurement of X-position, Y-position	
XY_PST	[13:12]	Manual measurement of X-position or Y-position. 00 = No operation mode 01 = X-position measurement 10 = Y-position measurement 11 = Waiting for Interrupt Mode	-
Reserved	[11:10]	Reserved	
YPDATA	[9:0]	Y-position conversion data value Data value: 0 ~ 3FF	-

4.触摸屏相关电路图

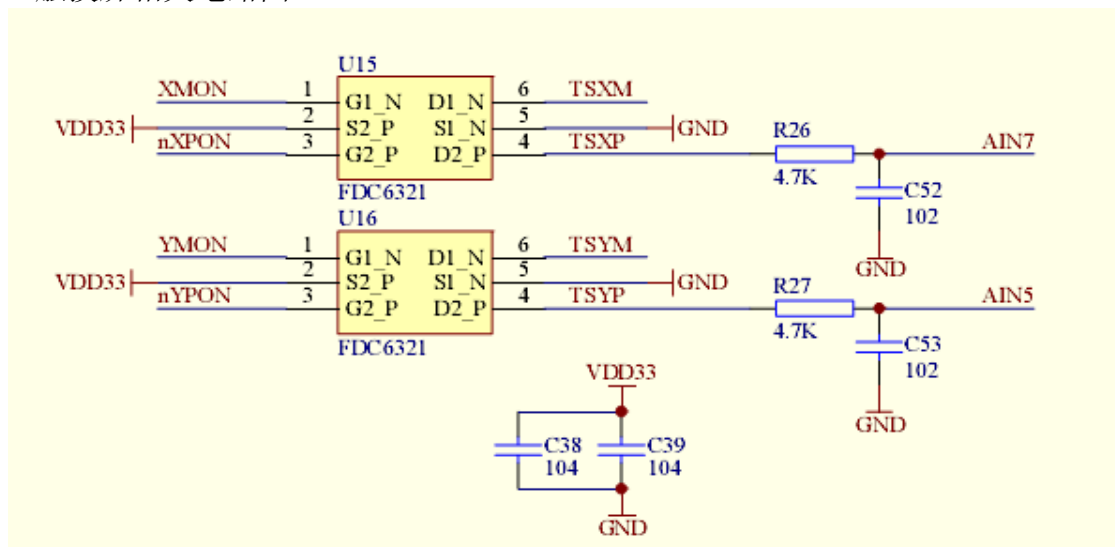


图 3-22 触摸屏电路

5. 触摸屏与显示器的配合

从触摸屏控制器获得的 X 与 Y 值仅是对当前触摸点的电压值的 A/D 转换值，它不具有实用价值。这个值的大小不但与触摸屏的分辨率有关，而且也与触摸屏与 LCD 贴合的情况有关。而且，LCD 分辨率与触摸屏的分辨率一般来说是不一样，坐标也不一样，因此，如果想得到体现 LCD 坐标的触摸屏位置，还需要在程序中进行转换。转换公式如下：

$$x=(x-TchScr_Xmin)*LCDWIDTH/(TchScr_Xmax-TchScr_Xmin)$$

$$y=(y-TchScr_Ymin)*LCDHEIGHT/(TchScr_Ymax-TchScr_Ymin)$$

其中，TchScr_Xmax、TchScr_Xmin、TchScr_Ymax 和 TchScr_Ymin 是触摸屏返回电压值 x、y 轴的范围，LCDWIDTH、LCDHEIGHT 是液晶屏的宽度与高度。

定义驱动函数 (tchscr.c)

```
void TchScr_GetScrXY(int *x, int *y)
```

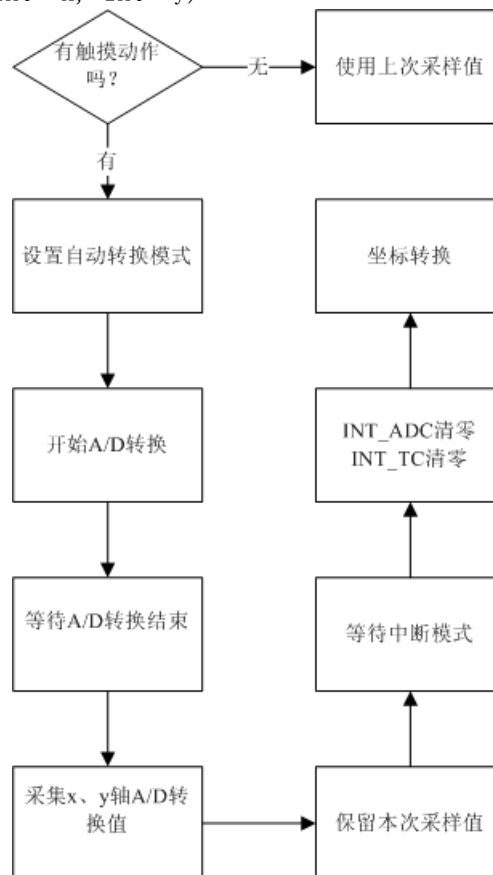


图 3-23 读取触摸点 x 轴电压值

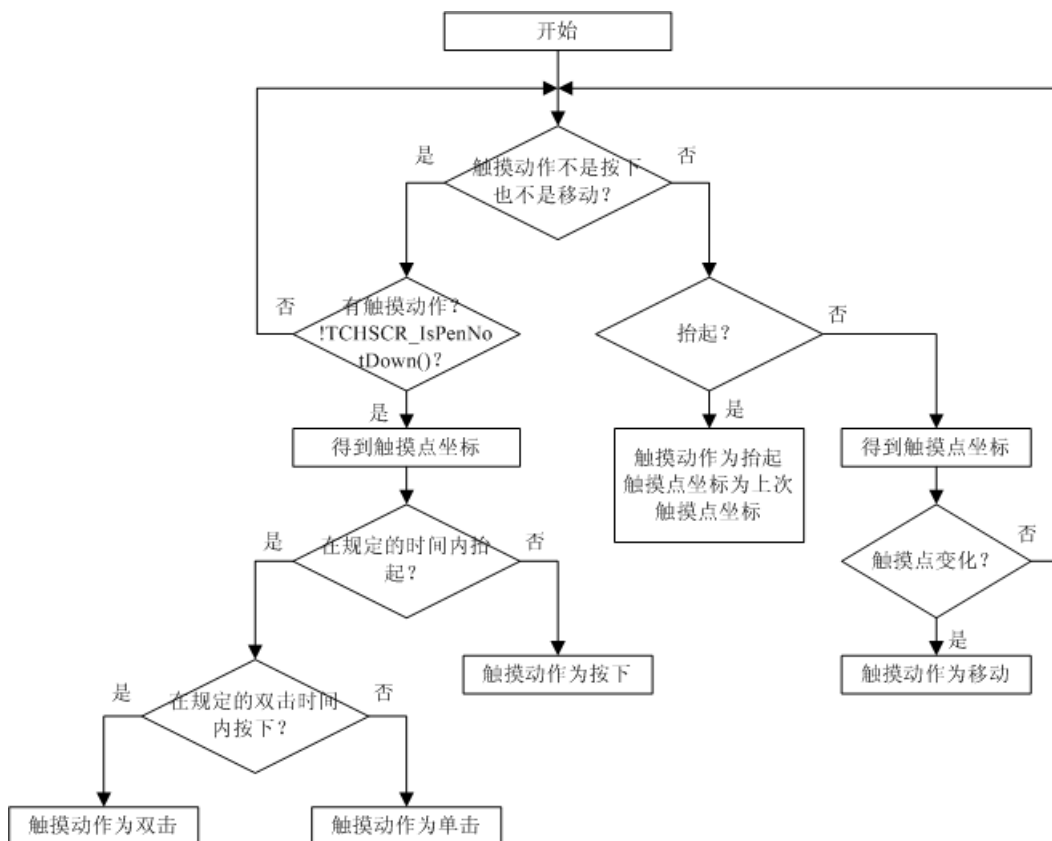


图 3-24 判断触摸动作

4. 编写测试函数(tchscr.c)

将触摸动作及触摸点坐标在超级终端上显示出来。

5. 校准触摸屏坐标输出，转换坐标，与 LCD 紧密配合

可以使用 TchScr_GetScrXY() 函数来获得液晶屏的 x、y 方向的电压范围，分别点触摸屏有效面积的左上角和右下角，得到下列参数：

iTchScr_Xmin=55,TchScr_Xmax=695,

TchScr_Ymin=300,TchScr_Ymax=890;//此数值仅供参考，请以实际校对为准

六、实验步骤

1. 启动 H-JTAG 仿真器并进行初始化配置。
2. 打开 EWARM Exp4 触摸屏驱动实验中的工程。
3. 编译并下载程序到开发板观察实验现象。

七、思考题

- 1) 电阻型触摸屏检测坐标值的原理
- 2) 如果 LCD 坐标原点在右下角, 分辨率为 240X180, 触摸屏坐标原点在右上角, 请给出触摸屏输出坐标的转换公式, 对触摸屏的分辨率有什么影响。

实验五 LCD 的驱动控制实验

一、实验目的

1. 了解 LCD 基本概念与原理。
2. 理解 LCD 的驱动控制。
3. 熟悉用总线方式驱动 LCD 模块。
4. 熟悉用 ARM 内置的 LCD 控制器驱动 LCD。

二、实验内容

学习 LCD 显示器的基本原理, 理解其驱动控制方法。掌握两种 LCD 驱动方式的基本原理和方法。并用编程实现:

1. 用总线方式直接驱动带有驱动模块的 LCD。
2. 用 ARM 内置的 LCD 控制器来驱动 LCD。

三、预备知识

1. 用 EWARM 集成开发环境, 编写和调试程序的基本过程。
2. ARM 应用程序的框架结构。

四、实验设备及工具

硬件: ARM 嵌入式开发平台、用于 ARM920T 的 JTAG 仿真器、PC 机 Pentium100 以上。

软件: PC 机操作系统 Win2000 或 WinXP、EWARM 集成开发环境、仿真器驱动程序、超级终端通讯程序

五、实验原理及说明

1. LCD (Liquid Crystal Display) 原理

液晶得名于其物理特性：它的分子晶体，以液态存在而非固态。这些晶体分子的液体特性使得它具有两种非常有用的特点：1、如果让电流通过液晶层，这些分子将会以电流的流向方向进行排列，如果没有电流，它们将会彼此平行排列。2、如果提供了带有细小沟槽的外层，将液晶倒入后，液晶分子会顺着槽排列，并且内层与外层以同样的方式进行排列。

液晶的第三个特性是很神奇的：液晶层能使光线发生扭转。液晶层表现的有些类似偏光器，这就意味着它能够过滤除了那些从特殊方向射入之外的所有光线。此外，如果液晶层发生了扭转，光线将会随之扭转，以不同的方向从另外一个面中射出。

液晶的这些特点使得它可以被用来当作一种开关——即可以阻碍光线，也可以允许光线通过。液晶单元的底层是由细小的脊构成的，这些脊的作用是让分子呈平行排列。上表面也是如此，在这两侧之间的分子平行排列，不过当上下两个表面之间呈一定的角度时，液晶随着两个不同方向的表面进行排列，就会发生扭曲。结果便是这个扭曲的螺旋层使通过的光线也发生扭曲。如果电流通过液晶，所有的分子将会按照电流的方向进行排列，这样就会消除光线的扭转。如果将一个偏振滤光器放置在液晶层的上表面，扭转的光线通过(如图 A)，而没有发生扭转的光线(如图 B)将被阻碍。因此可以通过电流的通断改变 LCD 中的液晶排列，使光线在加电时射出，而不加电时被阻断。也有某些设计为了省电的需要，有电流时，光线不能通过，没有电流时，光线通过。

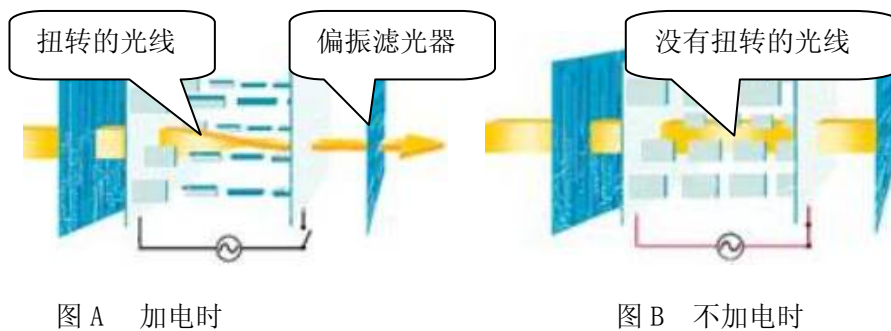


图 3-15 光线穿过与阴断示意图

LCD 显示器的基本原理就是通过给不同的液晶单元供电，控制其光线的通过与否，从而达到显示的目的。因此，LCD 的驱动控制归于对每个液晶单元的通断电的控制，每个液晶单元都对应着一个电极，对其通电，便可使光线通过（也有刚好相反的，即不通电时光线通过，通电时光线不通过）。

2. 电致发光

LCD 的发光原理是通过控制加电与否来使光线通过或挡住，从而显示图形。光源的提供方式有两种：透射式和反射式。笔记本电脑的 LCD 显示屏即为透射式，屏后面有一个光源，因此外界环境可以不需要光源。而一般微控制器上使用的 LCD 为反射式，需要外界提供光源，靠反射光来工作。电致发光 (EL) 是液晶屏提供光源的一种方式。电致发光的特点是低功耗，与二极管发光比较而言体积小。

电致发光 (EL) 是将电能直接转换为光能的一种发光现象。电致发光片是利用此原理经过加工制作而成的一种发光薄片，如图 7-2 所示。其特点是：超薄、高亮度、高效率、低功耗、低热量、可弯曲、抗冲击、长寿命、多种颜色选择等。因此，电致发光片被广泛应用于各种领域。

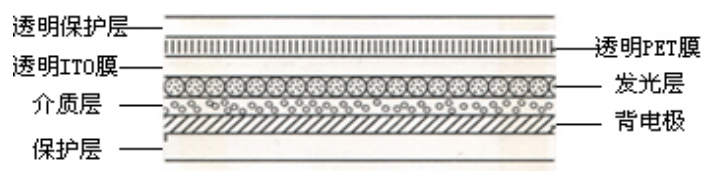


图 3-16 电致发光片的基本结构

3. LCD 的驱动控制

市面上出售的 LCD 有两种类型：

一种是带有驱动电路的 LCD 显示模块，这种 LCD 可以方便地与各种低档单片机进行接口，如 8051 系列单片机，但是由于硬件驱动电路的存在，体积比较大。这种模式常常使用总线方式来驱动。

另一种是 LCD 显示屏，没有驱动电路，需要与驱动电路配合使用。特点是体积小，但却需要另外的驱动芯片。也可以使用带有 LCD 驱动能力的高档 MCU 驱动，如 ARM 系列的 S3C2410X。

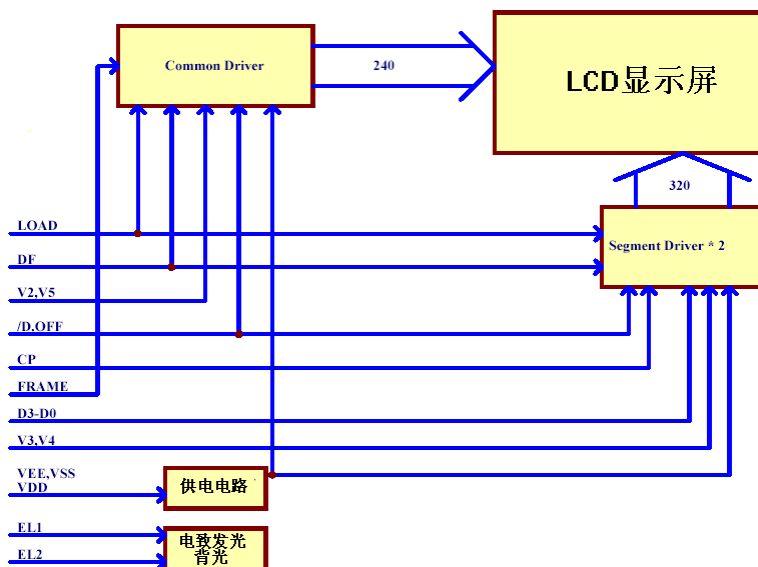


图 3-17 不带驱动电路的 LCD 结构

(1) 总线驱动方式

一般带有驱动模块的 LCD 显示屏使用这种驱动方式，由于 LCD 已经带有驱动硬件电路，因此模块给出的是总线接口，便于与单片机的总线进行接口。驱动模块具有八位数据总线，外加一些电源接口和控制信号。而且自带显示缓存，只需要将要显示的内容送到显示缓存中就可以实现内容的显示。由于只有八条数据线，因此常常通过引脚信号来实现地址与数据线复用，以达到把相应数据送到相应显示缓存的目的。下图为一个典型的显示模块（HY-12864B）提供的总线接口。

Pin No	Description
1	GND
2	Power supply for Logic
3	Power supply for LCD
4	Register selection (H:Data registor, L:Instruction registor)
5	Read/write selection (H:Read,L:Write)
6	Enable signal for chip
7-14	Data Bus line
15	Chip Select Signal for Left Half of the Screen
16	Chip Select Signal for RIGHT Half of the Screen
17	Reset signal
18	Negative voltage output
19	Power supply for Backlight
20	Power supply for Backlight

图 3-18 曲型带驱动液晶模块的总线接口

(2) 控制器扫描方式

S3C2410X 中具有内置的 LCD 控制器，它具有将显示缓存（在系统存储器中）中的 LCD 图象数据传输到外部 LCD 驱动电路的逻辑功能。

S3C2410X 中内置的 LCD 控制器可支持灰度 LCD 和彩色 LCD。在灰度 LCD 上，使用基于时间的抖动算法（time-based dithering algorithm）和 FRC (Frame Rate Control)方法，可以支持单色、4 级灰度和 16 级灰度模式的灰度 LCD。在彩色 LCD 上，可以支持 256 级彩色，使用 STN LCD 可以支持 4096 级彩色。对于不同尺寸的 LCD，具有不同数量的垂直和水平像素、数据接口的数据宽度、接口时间及刷新率，而 LCD 控制器可以进行编程控制相应的寄存器值，以适应不同的 LCD 显示板。

内置的 LCD 控制器提供了下列外部接口信号：

VFRAME/VSYNC/STV：帧同步信号（STN）/垂直同步信号(TFT)/SEC TFT信号

VLIN/HSYNC/CPV：行同步脉冲信号（STN）/水平同步信号（TFT）/SEC TFT信号

VCLK/LCD_HCLK：像素时钟信号（STN/TFT）/SEC TFT信号

VD[23:0]：LCD 像素数据输出端口（STN/TFT/SEC TFT）

VM/VDEN/TP：LCD驱动交流偏置信号（STN）/数据使能信号（TFT）/SEC TFT 信号

LEND/STH：行结束信号（TFT）/SEC TFT信号

LCD_PWREN：LCD面板电源使能控制信号

LCDVF0：SEC TFT OE信号

LCDVF1：SEC TFT REV信号

LCDVF2：SEC TFT REVB信号

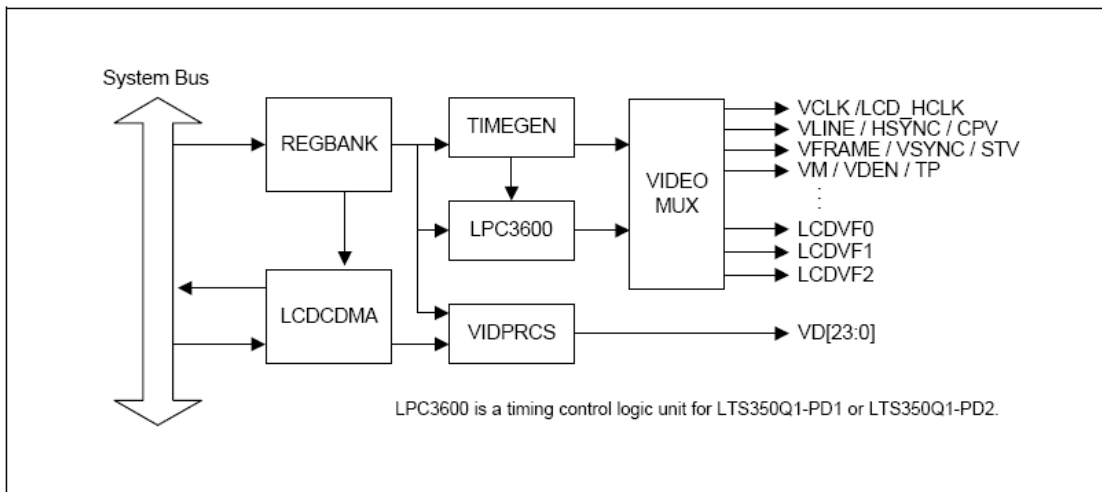


图 7-5 LCD 控制器逻辑框图

图 7-5 为 S3C2410X 中内置的 LCD 控制器的逻辑框图，它用于传输显示数据并产生必要的控制信号，如 VFRAME, VLINE, VCLK, 和 VM 等信号。除了控制信号，还有显示数据的数据端口 VD[23:0]如图 7-5。LCD 控制器包含 REGBANK, LCDCDMA, VIDPRCS, TIMEGEN 和 LPC3600。REGBANK 具有 17 个可编程寄存器和 256×16 颜料存储器，用于配置 LCD 控制器。LCDCDMA 为专用 DMA，它可以自动地将显示数据从帧内存中传送到 LCD 驱动器中。通过使用这一专用的 DMA，可以实现在不需要 CPU 介入的情况下显示数据。VIDPRCS 从 LCDCDMA 接收数据，变换为合适的数据格式（比如 4 / 8 位单一扫描和 4 位双扫描显示模式）后通过 VD[23:0]发送到 LCD 驱动器。TIMEGEN 包含可编程的逻辑，以支持常见 LCD 驱动器所需要的不同接口时序、速率要求。TIMEGEN 部分产生 VFRAME, VLINE, VCLK, VM 等信号。

(3) 与 ARM 自带 LCD 驱动器有关的寄存器

表 7-1 端口 D 寄存器

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

GPDCON	Bit	Description
GPD15	[31:30]	00 = Input 10 = VD23 01 = Output 11 = nSS0
GPD14	[29:28]	00 = Input 10 = VD22 01 = Output 11 = nSS1
GPD13	[27:26]	00 = Input 10 = VD21 01 = Output 11 = Reserved

GPD12	[25:24]	00 = Input 10 = VD20 01 = Output 11 = Reserved
GPD11	[23:22]	00 = Input 10 = VD19 01 = Output 11 = Reserved
GPD10	[21:20]	00 = Input 10 = VD18 01 = Output 11 = Reserved
GPD9	[19:18]	00 = Input 10 = VD17 01 = Output 11 = Reserved
GPD8	[17:16]	00 = Input 10 = VD16 01 = Output 11 = Reserved
GPD7	[15:14]	00 = Input 10 = VD15 01 = Output 11 = Reserved
GPD6	[13:12]	00 = Input 10 = VD14 01 = Output 11 = Reserved
GPD5	[11:10]	00 = Input 10 = VD13 01 = Output 11 = Reserved
GPD4	[9:8]	00 = Input 10 = VD12 01 = Output 11 = Reserved
GPD3	[7:6]	00 = Input 10 = VD11 01 = Output 11 = Reserved
GPD2	[5:4]	00 = Input 10 = VD10 01 = Output 11 = Reserved
GPD1	[3:2]	00 = Input 10 = VD9 01 = Output 11 = Reserved
GPD0	[1:0]	00 = Input 10 = VD8 01 = Output 11 = Reserved

GPDDAT	BIT	Description
GPD[15:0]	[15:0]	When the port is configured as input port, data from external sources can be read to the corresponding pin. When the port is configured as output port, data written in this register can be sent to the corresponding pin. When the port is configured as functional pin, undefined value will be read.

GPDUP	Bit	Description
GPD[15:0]	[15:0]	0: The pull-up function attached to to the corresponding port pin is enabled. 1: The pull-up function is disabled. (GPD[15:12] are 'pull-up disabled' state at the initial condition.)

LCD 驱动控制端口与 ARM 的端口是共用的，因此，要设置相应的寄存器，将其定义为

功能端口，即 LCD 驱动控制端口。

表7-2 LCDCON1

Register	Address	R/W	Description	Reset Value
LCDCON1	0X4D000000	R/W	LCD control 1 register	0x00000000

LCDCON1	Bit	Description	Initial State
LINECNT (read only)	[27:18]	Provide the status of the line counter. Down count from LINEVAL to 0	0000000000
CLKVAL	[17:8]	Determine the rates of VCLK and CLKVAL[9:0]. STN: $VCLK = HCLK / (CLKVAL \times 2)$ ($CLKVAL \geq 2$) TFT: $VCLK = HCLK / [(CLKVAL + 1) \times 2]$ ($CLKVAL \geq 0$)	0000000000
MMODE	[7]	Determine the toggle rate of the VM. 0 = Each Frame, 1 = The rate defined by the MVAL	0
PNRMODE	[6:5]	Select the display mode. 00 = 4-bit dual scan display mode (STN) 01 = 4-bit single scan display mode (STN) 10 = 8-bit single scan display mode (STN) 11 = TFT LCD panel	00
BPPMODE	[4:1]	Select the BPP (Bits Per Pixel) mode. 0000 = 1 bpp for STN, Monochrome mode 0001 = 2 bpp for STN, 4-level gray mode 0010 = 4 bpp for STN, 16-level gray mode 0011 = 8 bpp for STN, color mode 0100 = 12 bpp for STN, color mode 1000 = 1 bpp for TFT 1001 = 2 bpp for TFT 1010 = 4 bpp for TFT 1011 = 8 bpp for TFT 1100 = 16 bpp for TFT 1101 = 24 bpp for TFT	0000
ENVID	[0]	LCD video output and the logic enable/disable. 0 = Disable the video output and the LCD control signal. 1 = Enable the video output and the LCD control signal.	0

表 7-3 LCDCON2

Register	Address	R/W	Description	Reset Value
LCDCON2	0X4D000004	R/W	LCD control 2 register	0x00000000

LCDCON2	Bit	Description	Initial State
---------	-----	-------------	---------------

VBP	[31:24]	TFT: Vertical back porch is the number of inactive lines at the start of a frame, after vertical synchronization period. STN: These bits should be set to zero on STN LCD.	0x00
LINEVAL	[23:14]	TFT/STN: These bits determine the vertical size of LCD panel.	0000000000
VFP	[13:6]	TFT: Vertical front porch is the number of inactive lines at the end of a frame, before vertical synchronization period. STN: These bits should be set to zero on STN LCD.	00000000
VSPW	[5:0]	TFT: Vertical sync pulse width determines the VSYNC pulse's high level width by counting the number of inactive lines. STN: These bits should be set to zero on STN LCD.	000000

LCDCON3

Register	Address	R/W	Description	Reset Value
LCDCON3	0X4D000008	R/W	LCD control 3 register	0x00000000

LCDCON3	Bit	Description	Initial state
HBP (TFT)	[25:19]	TFT: Horizontal back porch is the number of VCLK periods between the falling edge of HSYNC and the start of active data.	0000000
WDLY (STN)		STN: WDLY[1:0] bits determine the delay between VLINE and VCLK by counting the number of the HCLK. WDLY[7:2] are reserved. 00 = 16 HCLK, 01 = 32 HCLK, 10 = 48 HCLK, 11 = 64 HCLK	
HOZVAL	[18:8]	TFT/STN: These bits determine the horizontal size of LCD panel. HOZVAL has to be determined to meet the condition that total bytes of 1 line are 4n bytes. If the x size of LCD is 120 dot in mono mode, x=120 cannot be supported because 1 line consists of 15 bytes. Instead, x=128 in mono mode can be supported because 1 line is composed of 16 bytes (2n). LCD panel driver will discard the additional 8 dot.	0000000000
HFP (TFT)	[7:0]	TFT: Horizontal front porch is the number of VCLK periods between the end of active data and the rising edge of HSYNC.	0X00

LINEBLANK (STN)		STN: These bits indicate the blank time in one horizontal line duration time. These bits adjust the rate of the VLINE finely. The unit of LINEBLANK is HCLK X 8. Ex) If the value of LINEBLANK is 10, the blank time is inserted to VCLK during 80 HCLK.	
-----------------	--	---	--

LCDCON4

Register	Address	R/W	Description	Reset Value
LCDCON4	0X4D00000C	R/W	LCD control 4 register	0x00000000

LCDCON4	Bit	Description	Initial state
MVAL	[15:8]	STN: These bit define the rate at which the VM signal will toggle if the MMODE bit is set to logic '1'.	0X00
HSPW(TFT) WLH(STN)	[7:0]	TFT: Horizontal sync pulse width determines the HSYNC pulse's high level width by counting the number of the VCLK. STN: WLH[1:0] bits determine the VLINE pulse's high level width by counting the number of the HCLK. WLH[7:2] are reserved. 00 = 16 HCLK, 01 = 32 HCLK, 10 = 48 HCLK, 11 = 64 HCLK	0X00

Register	Address	R/W	Description	Reset Value
LCDCON5	0X4D000010	R/W	LCD control 5 register	0x00000000

LCDCON5	Bit	Description	Initial state
Reserved	[31:17]	This bit is reserved and the value should be '0'.	0
VSTATUS	[16:15]	TFT: Vertical Status (read only). 00 = VSYNC 10 = ACTIVE 01 = BACK Porch 11 = FRONT Porch	00
HSTATUS	[14:13]	TFT: Horizontal Status (read only). 00 = HSYNC 01 = BACK Porch 10 = ACTIVE 11 = FRONT Porch	00
BPP24BL	[12]	TFT: This bit determines the order of 24 bpp video memory. 0 = LSB valid 1 = MSB Valid	0
FRM565	[11]	TFT: This bit selects the format of 16 bpp output video data. 0 = 5:5:5:1 Format 1 = 5:6:5 Format	0

INNVCLK	[10]	STN/TFT: This bit controls the polarity of the VCLK active edge. 0 = The video data is fetched at VCLK falling edge 1 = The video data is fetched at VCLK rising edge	0
INVVLIN	[9]	STN/TFT: This bit indicates the VLINE/HSYNC pulse polarity. 0 = Normal 1 = Inverted	0
INVVFRAME	[8]	STN/TFT: This bit indicates the VFRAME/VSYSN pulse polarity. 0 = Normal 1 = Inverted	0
INVVD	[7]	STN/TFT: This bit indicates the VD (video data) pulse polarity. 0 = Normal 1 = VD is inverted.	0
INVVDEN	[6]	TFT: This bit indicates the VDEN signal polarity. 0 = Normal 1 = Inverted	0
INVPWREN	[5]	STN/TFT: This bit indicates the PWREN signal polarity. 0 = Normal 1 = Inverted	0
INVLEND	[4]	TFT: This bit indicates the LEND signal polarity. 0 = Normal 1 = Inverted	0
PWREN	[3]	STN/TFT: LCD_PWREN output signal enable/disable. 0 = Disable PWREN signal 1 = Enable PWREN signal	0
ENLEND	[2]	TFT: LEND output signal enable/disable. 0 = Disable LEND signal 1 = Enable LEND signal	0
BSWP	[1]	STN/TFT: Byte swap control bit. 0 = Swap Disable 1 = Swap Enable	0
HWSWP	[0]	STN/TFT: Half-Word swap control bit. 0 = Swap Disable 1 = Swap Enable	0

表 7-4 LCDSADDR1

Register	Address	R/W	Description	Reset Value
LCDSADDR1	0X4D000014	R/W	STN/TFT: Frame buffer start address 1 register	0x00000000

LCDSADDR1	Bit	Description	Initial State
-----------	-----	-------------	---------------

LCDBANK	[29:21]	These bits indicate A[30:22] of the bank location for the video buffer in the system memory. LCDBANK value cannot be changed even when moving the view port. LCD frame buffer should be within aligned 4MB region, which ensures that LCDBANK value will not be changed when moving the view port. So, care should be taken to use the malloc() function.	0x00
LCDBASEU	[20:0]	For dual-scan LCD: These bits indicate A[21:1] of the start address of the upper address counter, which is for the upper frame memory of dual scan LCD or the frame memory of single scan LCD. For single-scan LCD: These bits indicate A[21:1] of the start address of the LCD frame buffer.	0x000000

表 7-5 LCDSADDR2

Register	Address	R/W	Description	Reset Value
LCDSADDR2	0X4D000018	R/W	STN/TFT: Frame buffer start address 2 register	0x00000000

LCDSADDR2	Bit	Description	Initial State
LCDBASEL	[20:0]	For dual-scan LCD: These bits indicate A[21:1] of the start address of the lower address counter, which is used for the lower frame memory of dual scan LCD. For single scan LCD: These bits indicate A[21:1] of the end address of the LCD frame buffer. $LCDBASEL = ((\text{the frame end address}) \gg 1) + 1$ $= LCDBASEU + (\text{PAGEWIDTH} + \text{OFFSIZE}) \times (\text{LINEVAL} + 1)$	0x0000

注意：当LCD控制器开启时，如果想滚屏需要改变LCDBASEU和LCDBASEL值。但是用户不能在帧尾通过查LCDCON1中LINECNT的值来改变寄存器LCDBASEU和LCDBASEL的值。因为LCD的FIFO取下一帧的数据优先级高于改变LCDBASEU和LCDBASEL值的优先级。

所以，如果改变了帧，那么预取的FIFO数据就不是最新的了，导致LCD的显示不正确。查询LINECNT前，中断要屏蔽掉。如果在读取了LINECNT之后发生任何中断，因为中断服务程序执行消耗的时间导致读取到的LINECNT的值不是最新的。

表 7-6 LCDSADDR3

Register	Address	R/W	Description	Reset Value
LCDSADDR3	0X4D00001C	R/W	STN/TFT: Virtual screen address	0x00000000

			set	
--	--	--	-----	--

LCDSADDR3	Bit	Description	Initial State
OFFSIZE	[21:11]	Virtual screen offset size (the number of half words). This value defines the difference between the address of the last half word displayed on the previous LCD line and the address of the first half word to be displayed in the new LCD line.	0000000000
PAGEWIDTH	[10:0]	Virtual screen page width (the number of half words). This value defines the width of the view port in the frame.	000000000

注意：当ENVID位的为0时，PAGEWIDTH和OFFSIZE的值必须改变。

Example 1. LCD panel = 320*240, 16gray, single scan

Frame start address = 0x0c500000

Offset dot number = 2048 dots (512 half words)

LINEVAL = 240-1 = 0xef

PAGEWIDTH = 320*4/16 = 0x50

OFFSIZE = 512 = 0x200

LCDBANK = 0x0c500000 >> 22 = 0x31

LCDBASEU = 0x100000 >> 1 = 0x80000

LCDBASEL = 0x80000 + (0x50 + 0x200) * (0xef + 1) = 0xa2b00

Example 2. LCD panel = 320*240, 16gray, dual scan

Frame start address = 0x0c500000

Offset dot number = 2048 dots (512 half words)

LINEVAL = 120-1 = 0x77

PAGEWIDTH = 320*4/16 = 0x50

OFFSIZE = 512 = 0x200

LCDBANK = 0x0c500000 >> 22 = 0x31

LCDBASEU = 0x100000 >> 1 = 0x80000

LCDBASEL = 0x80000 + (0x50 + 0x200) * (0x77 + 1) = 0x91580

Example 3. LCD panel = 320*240, color, single scan

Frame start address = 0x0c500000

Offset dot number = 1024 dots (512 half words)

LINEVAL = 240-1 = 0xef

PAGEWIDTH = 320*8/16 = 0xa0

OFFSIZE = 512 = 0x200

LCDBANK = 0x0c500000 >> 22 = 0x31

LCDBASEU = 0x100000 >> 1 = 0x80000

LCDBASEL = 0x80000 + (0xa0 + 0x200) * (0xef + 1) = 0xa7600

Dithering Mode Register

Register	Address	R/W	Description	Reset Value
----------	---------	-----	-------------	-------------

DITHMODE	0X4D00004C	R/W	STN: Dithering mode register. This register reset value is 0x000000 But, user can change this value to 0x12210. (Refer to a sample program source for the latest value of this register.)	0x000000
----------	------------	-----	--	----------

DITHMODE	Bit	Description	Initial state
DITHMODE	[18:0]	Use one of following value for your LCD: 0x000000 or 0x12210	0x000000

RED Lookup Table Register

Register	Address	R/W	Description	Reset Value
REDLUT	0X4D000020	R/W	STN: Red lookup table register	0x00000000

REDLUT	Bit	Description	Initial State
REDVAL	[31:0]	These bits define which of the 16 shades will be chosen by each of the 8 possible red combinations. 000 = REDVAL[3:0], 001 = REDVAL[7:4] 010 = REDVAL[11:8], 011 = REDVAL[15:12] 100 = REDVAL[19:16], 101 = REDVAL[23:20] 110 = REDVAL[27:24], 111 = REDVAL[31:28]	0x00000000

GREEN Lookup Table Register

Register	Address	R/W	Description	Reset Value
GREENLUT	0X4D000024	R/W	STN: Green lookup table register	0x00000000

GREENLUT	Bit	Description	Initial State
GREENVAL	[31:0]	These bits define which of the 16 shades will be chosen by each of the 8 possible green combinations. 000 = GREENVAL[3:0], 001 = GREENVAL[7:4] 010 = GREENVAL[11:8], 011 = GREENVAL[15:12] 100 = GREENVAL[19:16], 101 = GREENVAL[23:20] 110 = GREENVAL[27:24], 111 = GREENVAL[31:28]	0x00000000

BLUE Lookup Table Register

Register	Address	R/W	Description	Reset Value
BLUELUT	0X4D000028	R/W	STN: Blue lookup table register	0x0000

BULELUT	Bit	Description	Initial State
BLUEVAL	[15:0]	These bits define which of the 16 shades will be chosen by each of the 4 possible blue combinations. 00 = BLUEVAL[3:0], 01 = BLUEVAL[7:4] 10 = BLUEVAL[11:8], 11 = BLUEVAL[15:12]	0x0000

这段空间（0x14A0002C to 0x14A00048）不要使用，是留给测试模式使用的

LPC3600 Control Register

Register	Address	R/W	Description	Reset Value
LPCSEL	0X4D000060	R/W	This register controls the LPC3600 modes.	0x4

LPCSEL	Bit	Description	Initial state
RES_SEL	[1]	1 = 240×320	0
LPC_EN	[0]	Determine LPC3600 Enable/Disable. 0 = LPC3600 Disable 1 = LPC3600 Enable	0

Temp Palette Register

Register	Address	R/W	Description	Reset Value
TPAL	0X4D000050	R/W	TFT: Temporary palette register. This register value will be video data at next frame.	0x00000000

TPAL	Bit	Description	Initial state
TPALEN	[24]	Temporary palette register enable bit. 0 = Disable 1 = Enable	0
TPALVAL	[23:0]	Temporary palette value register. TPALVAL[23:16] : RED TPALVAL[15:8] : GREEN TPALVAL[7:0] : BLUE	0x000000

3.4.6 实验步骤

1. 启动 H-JTAG 仿真器并进行初始化配置。
2. 打开 Exp5 LCD 驱动控制实验”的工程。
3. 定义有关常量与宏
4. 编写 LCD 初始化函数(lcd320.c)，设置各功能寄存器。
5. 编写 LCD 刷新函数(lcd320.c)。
此函数主要是将二级缓存 LCDBuffer 的数据由 32 位彩色图形信息转换成 16 位的图形信息，然后放到 pLCDFB 指向的一级缓存。
转换公式：
$$\text{pixcolor} = ((\text{pbuf}[0] \& 0xf8) \ll 11) | ((\text{pbuf}[1] \& 0xfc) \ll 6) | (\text{pbuf}[2] \& 0xf8)。$$

其中，pbuf[0]、pbuf[1]、pbuf[2]是一个像素的 32 位彩色数据的前 24 位，分别代表 R、G、B。
6. 编写主函数(main.c)
在 LCD 上显示 16 位色图形的关键是填充二级显示缓冲，将显示像素的 24 位颜色信息写入 LCDBuffer。将 RGB 三种基本颜色按一定比例混合即可构成更复杂的颜色，每个像素的三种基本颜色分别占一个字节，可以方便的在程序里改写各基本颜色的数值，从而改变该像素的混合颜色。
7. 在 EWARM 集成开发环境中编译、调试和运行工程程序。

3.4.7 思考题

- 1) 液晶显示的基本原理是什么？
- 2) 总线方式驱动液晶模块和使用控制器进行驱动控制有什么异同？
- 3) LCD 显示图形的基本思想是什么？

实验六 uCOS-II 在 ARM 微处理器上的移植及编译

一、实验目的

1. 了解 uCOS-II 内核的主要结构。
2. 掌握将 uCOS-II 内核移植到 ARM920T 处理器上的基本方法。

二、实验内容

1. 将 uCOS-II 内核移植到 ARM920T 微处理器上。
2. 编写两个简单任务，在超级终端上观察两个任务的切换。

三、预备知识

1. 掌握在 EWARM 集成开发环境中编写和调试程序的基本过程。
2. 了解 ARM920T 处理器的结构。
3. 了解 uCOS-II 系统结构。

四、实验设备及工具

硬件：ARM 嵌入式开发平台、用于 ARM920T 的 JTAG 仿真器、PC 机 Pentium100 以上。

软件：PC 机操作系统 Win2000 或 WinXP、EWARM 集成开发环境、仿真器驱动程序、超级终端通讯程序。

五、实验原理

所谓移植，指的是一个操作系统可以在某个微处理器或者微控制器上运行。虽然 uCOS-II 的大部分源代码是用 C 语言写成的，仍需要用 C 语言和汇编语言完成一些与处理器相关的代码。比如：uCOS-II 在读写处理器、寄存器时只能通过汇编语言来实现。因为 uCOS-II 在设计的时候就已经充分考虑了可移植性，所以，uCOS-II 的移植还是比较容易的。

要使 uCOS-II 可以正常工作，处理器必须满足以下要求：

1. 处理器的 C 编译器能产生可重入代码。

可重入的代码指的是一段代码（如一个函数）可以被多个任务同时调用，而不必担心会破坏数据。也就是说，可重入型函数在任何时候都可以被中断执行，过一段时间以后又可以继续运行，而不会因为在函数中断的时候被其他的任务重新调用，影响函数中的数据。下面的两个例子可以比较可重入型函数和非可重入型函数：

程序 1：可重入型函数

```
● void swap(int *x, int *y)
● {
●   int temp;
●   temp=*x;
●   *x=*y;
●   *y=temp;
● }
```

●

程序 2: 非可重入型函数

```
●   int temp;
●   void swap(int *x, int *y)
●   {
●       temp=*x;
●       *x=*y;
●       *y=temp;
●   }
●
```

程序 1 中使用的是局部变量 temp 作为变量。通常的 C 编译器,把局部变量分配在栈中。所以,多次调用同一个函数,可以保证每次的 temp 互不影响。而程序 2 中 temp 定义的是全局变量,多次调用函数的时候,必然受到影响。

代码的可重入性是保证完成多任务的基础,除了在 C 程序中使用局部变量以外,还需要 C 编译器的支持。笔者使用的是 ARM ADS 的集成开发环境,均可以生成可重入的代码。

2. 在程序中可以打开或者关闭中断。

在 uCOS-II 中,可以通过 OS_ENTER_CRITICAL() 或者 OS_EXIT_CRITICAL() 宏来控制系统关闭或者打开中断。这需要处理器的支持,在 ARM920T 的处理器上,可以设置相应的寄存器来关闭或者打开系统的所有中断。

3. 处理器支持中断,并且能产生定时中断(通常在 10Hz~1000Hz 之间)。

uCOS-II 是通过处理器产生的定时器的中断来实现多任务之间的调度的。在 ARM920T 的处理器上可以产生定时器中断。

4. 处理器支持能够容纳一定量数据的硬件堆栈。

5. 处理器有将堆栈指针和其它 CPU 寄存器存储和读出到堆栈(或者内存)的指令。

uCOS-II 进行任务调度的时候,会把当前任务的 CPU 寄存器存放到此任务的堆栈中,然后,再从另一个任务的堆栈中恢复原来的工作寄存器,继续运行另一个任务。所以,寄存器的入栈和出栈是 uCOS-II 多任务调度的基础。

图 4-1 说明了 uC/OS 的结构以及它与硬件的关系。

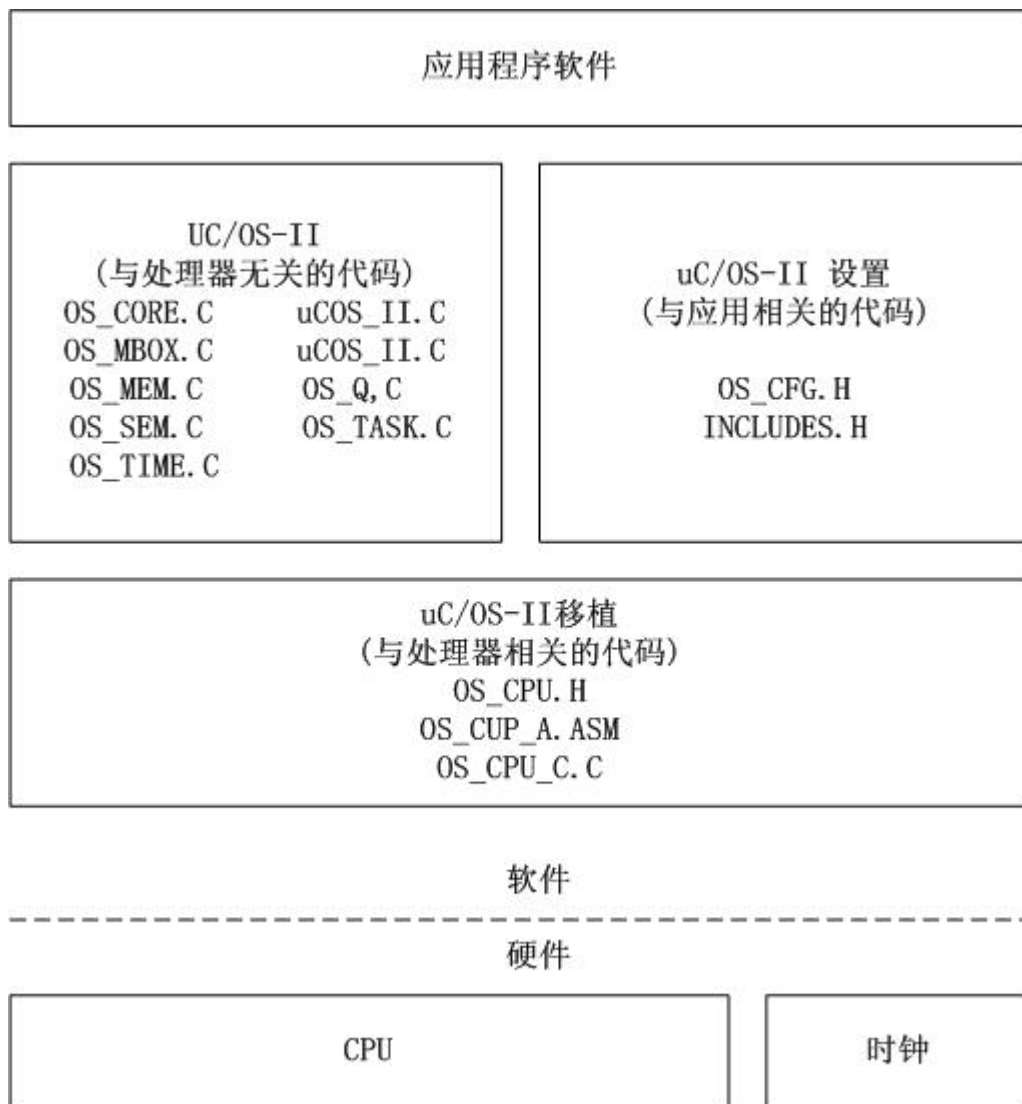


图 4-1 uCOS-II 硬件和软件体系结构

ARM920T 处理器完全满足上述要求。接下来将介绍如何把 uCOS-II 移植到 Samsung 公司的一款 ARM920T 的嵌入式处理器——S3C2410X 上。

六、实验步骤

1. 该实验的文件分为两类，其一是 STARTUP 目录下的系统初始化、配置等文件，其二是 uCOS-II 的全部源码，arch 目录下的 3 个文件是和处理器架构相关的。

2. 设置 os_cpu.h 中与处理器和编译器相关的代码

- typedef unsigned char BOOLEAN;
- typedef unsigned char INT8U;
- typedef signed char INT8S;

- typedef unsigned int INT16U;
- typedef signed int INT16S;
- typedef unsigned long INT32U;
- typedef signed long INT32S;
- typedef float FP32;
- typedef double FP64;
- typedef unsigned int OS_STK;
- typedef unsigned int OS_CPU_SR;
- extern int INTS_OFF(void);
- extern void INTS_ON(void);
- #define OS_ENTER_CRITICAL() { cpu_sr = INTS_OFF(); }
- #define OS_EXIT_CRITICAL() { if(cpu_sr == 0) INTS_ON(); }
- #define OS_STK_GROWTH 1

1) 与编译器相关的数据类型

因为不同的微处理器有不同的字长,所以 uCOS-II 的移植包括了一系列的类型定义以确保其可移植性。尤其是 uCOS-II 代码从不使用 C 的 short, int 和 long 等数据类型,因为它们是与编译器相关的,不可移植。相反的,我们定义的整形数据结构既是可移植的又是直观的。为了方便,虽然 uCOS-II 不使用浮点数据,但我们还是定义了浮点数据类型。

例如,INT16U 数据类型总是代表 16 位的无符号整数。现在,uCOS-II 和用户的应用程序就可以估计出声明为该数据类型的变量的取值范围是 0~65535。将 uCOS-II 移植到 32 位的处理器上也就意味着 INT16U 实际被声明为无符号短整形数据结构而不是无符号整数数据结构。但是,uCOS-II 所处理的仍然是 INT16U。

用户必须将任务堆栈的数据类型告诉给 uCOS-II。这个过程是通过为 OS_STK 声明正确的 C 数据类型来完成的。我们的处理器上的堆栈成员是 16 位的,所以将 OS_STK 声明为无符号整形数据类型。所有的任务堆栈都必须用 OS_STK 声明数据类型。

2) OS_ENTER_CRITICAL() 和 OS_EXIT_CRITICAL()

与所有的实时内核一样,uCOS-II 需要先禁止中断再访问代码的临界区,并且在访问完毕后重新允许中断。这就使得 uCOS-II 能够保护临界区代码免受多任务或中断服务例程(ISR)的破坏。在 S3C2410X 上是通过两个函数(OS_CPU_A.S)实现开关中断的。

● INTS_OFF

- mrs r0, cpsr ; 当前 CSR
- mov r1, r0 ; 复制屏蔽
- orr r1, r1, #0xC0 ; 屏蔽中断位
- msr CPSR, r1 ; 关中断(IRQ and FIQ)
- and r0, r0, #0x80 ; 从初始 CSR 返回 FIQ 位
- mov pc, lr ; 返回

● INTS_ON

- mrs r0, cpsr ; 当前 CSR
- bic r0, r0, #0xC0 ; 屏蔽中断

- `msr CPSR, r0` ; 开中断 (IRQ and FIQ)
- `mov pc, lr` ; 返回

3) OS_STK_GROWTH

绝大多数的微处理器和微控制器的堆栈是从上往下长的。但是某些处理器是用另外一种方式工作的。uCOS-II 被设计成两种情况都可以处理，只要在结构常量 OS_STK_GROWTH 中指定堆栈的生长方式就可以了。

置 OS_STK_GROWTH 为 0 表示堆栈从下往上长。

置 OS_STK_GROWTH 为 1 表示堆栈从上往下长。

3. 用 C 语言编写 6 个操作系统相关的函数 (OS_CPU_C.C)

1) OSTaskStkInit

OSTaskCreate() 和 OSTaskCreateExt() 通过调用 OSTaskStkInit() 来初始化任务的堆栈结构。因此，堆栈看起来就像刚发生过中断并将所有的寄存器保存到堆栈中的情形一样。图 4-2 显示了 OSTaskStkInit() 放到正被建立的任务堆栈中的东西。这里我们定义了堆栈是从上往下长的。

在用户建立任务的时候，用户传递任务的地址，pdata 指针，任务的堆栈栈顶和任务的优先级给 OSTaskCreate() 和 OSTaskCreateExt()。一旦用户初始化了堆栈，OSTaskStkInit() 就需要返回堆栈指针所指的地址。OSTaskCreate() 和 OSTaskCreateExt() 会获得该地址并将它保存到任务控制块 (OS_TCB) 中。

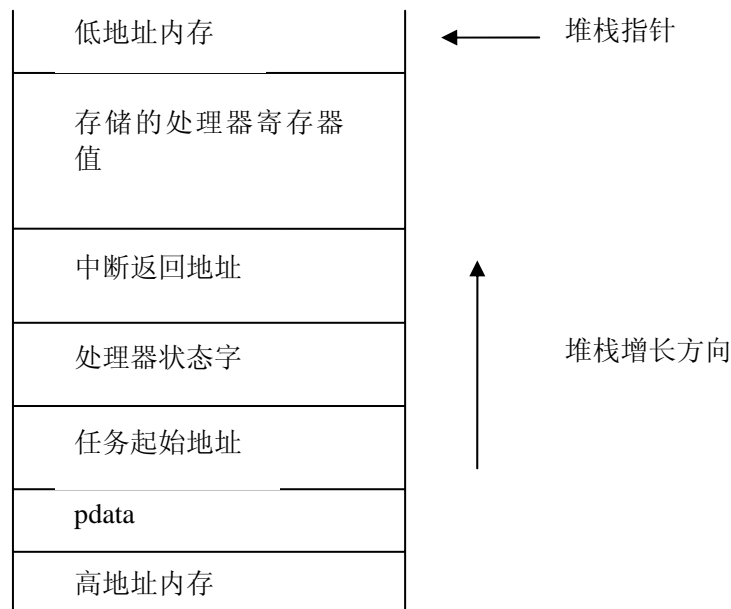


图 4-2 堆栈初始化 (pdata 通过堆栈传递)

- `OS_STK * OSTaskStkInit (void (*task)(void *pd), void *pdata, OS_STK *ptos,`
- `INT16U opt)`
- `{ unsigned int * stk;`
- `stk = (unsigned int *)ptos; /* 装载堆栈指针 */`

```
●    opt++;
    ●    /* 为新任务建立堆栈 */
    ●    *--stk = (unsigned int) task;    /* pc */
    ●    *--stk = (unsigned int) task;    /* lr */
    ●    *--stk = 12;                    /* r12 */
    ●    *--stk = 11;                    /* r11 */
    ●    *--stk = 10;                    /* r10 */
    ●    *--stk = 9;                     /* r9 */
    ●    *--stk = 8;                     /* r8 */
    ●    *--stk = 7;                     /* r7 */
    ●    *--stk = 6;                     /* r6 */
    ●    *--stk = 5;                     /* r5 */
    ●    *--stk = 4;                     /* r4 */
    ●    *--stk = 3;                     /* r3 */
    ●    *--stk = 2;                     /* r2 */
    ●    *--stk = 1;                     /* r1 */
    ●    *--stk = (unsigned int) pdata;   /* r0 */
    ●    *--stk = (SUPMODE);             /* cpsr */
    ●    *--stk = (SUPMODE);             /* spsr */
    ●    return ((OS_STK *)stk);
    ●    }
    ●
```

2) OSTaskCreateHook

当用 OSTaskCreate() 和 OSTaskCreateExt() 建立任务的时候就会调用 OSTaskCreateHook()。该函数允许用户或使用移植实例的用户扩展 uCOS-II 功能。当 uCOS-II 设置完了自己的内部结构后,会在调用任务调度程序之前调用 OSTaskCreateHook()。该函数被调用的时候中断是禁止的。因此用户应尽量减少该函数中的代码以缩短中断的响应时间。

当 OSTaskCreateHook() 被调用的时候,它会收到指向已建立任务的 OS_TCB 的指针,这样它就可以访问所有的结构成员了。

函数原型: void OSTaskCreateHook (OS_TCB *ptcb)

3) OSTaskDelHook

当任务被删除的时候就会调用 OSTaskDelHook()。该函数在把任务从 uCOS-II 的内部任务链表中解开之前被调用。当 OSTaskDelHook() 被调用的时候,它会收到指向正被删除任务的 OS_TCB 的指针,这样它就可以访问所有的结构成员了。OSTaskDelHook() 可以用来检验 TCB 扩展是否被建立(一个非空指针)并进行一些清除操作。

函数原型: void OSTaskDelHook (OS_TCB *ptcb)

4) OSTaskSwHook

当发生任务切换的时候就会调用 OSTaskSwHook()。OSTaskSwHook() 可以直接访问 OSTCBCur 和 OSTCBHighRdy,因为它们是全球变量。OSTCBCur 指向被切换出去的任务 OS_TCB,而 OSTCBHighRdy 指向新任务 OS_TCB。注意在调用 OSTaskSwHook() 期间中断一直是被禁止的。因此用户应尽量减少该函数中的代码以缩短中断的响应时间。

函数原型: void OSTaskSwHook (void)

5) OSTaskStatHook

OSTaskStatHook() 每秒钟都会被 OSTaskStat() 调用一次。用户可以用 OSTaskStatHook() 来扩展统计功能。例如, 用户可以保持并显示每个任务的执行时间, 每个任务所用的 CPU 份额, 以及每个任务执行的频率等。

函数原型: void OSTaskStatHook (void)

6) OSTimeTickHook

OSTimeTickHook() 在每个时钟节拍都会被 OSTaskTick() 调用。实际上, OSTimeTickHook() 是在节拍被 uCOS-II 真正处理, 并通知用户的移植实例或应用程序之前被调用的。

函数原型: void OSTimeTickHook (void)

后 5 个函数为钩子函数, 可以不加代码。只有当 OS_CFG.H 中的 OS_CPU_HOOKS_EN 被置为 1 时才会产生这些函数的代码。

4. 用汇编语言编写 4 个与处理器相关的函数 (OS_CPU.ASM)

1) OSStartHighRdy (); 运行优先级最高的就绪任务

- OSStartHighRdy
- LDR r4, addr_OSTCBCur ; 得到当前任务 TCB 地址
- LDR r5, addr_OSTCBHighRdy ; 得到最高优先级任务 TCB 地址
-
- LDR r5, [r5] ; 获得堆栈指针
- LDR sp, [r5] ; 转移到新的堆栈中
-
- STR r5, [r4] ; 设置新的当前任务 TCB 地址
- LDMFD sp!, {r4} ;
- MSR SPSR, r4
- LDMFD sp!, {r4} ; 从栈顶获得新的状态
- MSR CPSR, r4 ; CPSR 处于 SVC32Mode 模式
- LDMFD sp!, {r0-r12, lr, pc} ; 运行新的任务

2) OS_TASK_SW (); 任务级的任务切换函数

- OS_TASK_SW
- STMFD sp!, {lr} ; 保存 pc
- STMFD sp!, {lr} ; 保存 lr
- STMFD sp!, {r0-r12} ; 保存寄存器和返回地址
- MRS r4, CPSR
- STMFD sp!, {r4} ; 保存当前的 PSR
- MRS r4, SPSR
- STMFD sp!, {r4} ; 保存 SPSR
-
- ; OSPrioCur = OSPrioHighRdy
- LDR r4, addr_OSPrioCur
- LDR r5, addr_OSPrioHighRdy
- LDRB r6, [r5]

- STRB r6, [r4]
-
- ; 得到当前任务 TCB 地址
- LDR r4, addr_OSTCBCur
- LDR r5, [r4]
- STR sp, [r5] ; 保存 sp 在被占先的任务的 TCB
-
- ; 得到最高优先级任务 TCB 地址
- LDR r6, addr_OSTCBHighRdy
- LDR r6, [r6]
- LDR sp, [r6] ; 得到新任务堆栈指针
-
- ; OSTCBCur = OSTCBHighRdy
- STR r6, [r4] ; 设置新的当前任务的 TCB 地址
-
- ; 保存任务方式寄存器
- LDMFD sp!, {r4}
- MSR SPSR, r4
- LDMFD sp!, {r4}
- MSR CPSR, r4
-
- ; 返回到新任务的上下文
- LDMFD sp!, {r0-r12, lr, pc}

3) OSIntCtxSw(); 中断级的任务切换函数

- OSIntCtxSw
 - add r7, sp, #16 ; 保存寄存器指针
 - LDR sp, =IRQStack ; FIQ_STACK
 - mrs r1, SPSR ; 得到暂停的 PSR
 - orr r1, r1, #0xC0 ; 关闭 IRQ, FIQ.
 - msr CPSR_cxsf, r1 ; 转换模式 (应该是 SVC_MODE)
 - ldr r0, [r7, #52] ; 从 IRQ 堆栈中得到 IRQ's LR (任务 PC)
 - sub r0, r0, #4 ; 当前 PC 地址是 (saved_LR - 4)
 - STMFD sp!, {r0} ; 保存任务 PC
 - STMFD sp!, {lr} ; 保存 LR
 - mov lr, r7 ; 保存 FIQ 堆栈 ptr in LR (转到 nuke r7)
 - ldmfd lr!, {r0-r12} ; 从 FIQ 堆栈中得到保存的寄存器
 - STMFD sp!, {r0-r12} ; 在任务堆栈中保存寄存器
 - ; 在任务堆栈上保存 PSR 和任务 PSR
 - MRS r4, CPSR
 - bic r4, r4, #0xC0 ; 使中断位处于使能态
 - STMFD sp!, {r4} ; 保存任务当前 PSR
 - MRS r4, SPSR

- STMFD sp!, {r4} ; SPSR
- ; OSPrioCur = OSPrioHighRdy // 改变当前程序
- LDR r4, addr_OSPrioCur
- LDR r5, addr_OSPrioHighRdy
- LDRB r6, [r5]
- STRB r6, [r4]
- ; 得到被占先的任务 TCB
- LDR r4, addr_OSTCBCur
- LDR r5, [r4]
- STR sp, [r5] ; 保存 sp 在被占先的任务的 TCB
- ; 得到新任务 TCB 地址
- LDR r6, addr_OSTCBHighRdy
- LDR r6, [r6]
- LDR sp, [r6] ; 得到新任务堆栈指针
- ; OSTCBCur = OSTCBHighRdy
- STR r6, [r4] ; 设置新的当前任务的 TCB 地址
- LDMFD sp!, {r4}
- MSR SPSR, r4
- LDMFD sp!, {r4}
- BIC r4, r4, #0xC0 ; 必须退出新任务通过允许中断
- MSR CPSR, r4
- LDMFD sp!, {r0-r12, lr, pc}
-
- 4) OSTickISR(); 时钟节拍中断

多任务操作系统的任务调度是基于时钟节拍中断的，uCOS-II 也需要处理器提供一个定时器中断来产生节拍，借以实现时间的延时和期满功能。但在本系统移植 uCOS-II 时，时钟节拍中断的服务函数并非 uCOS-II 文献中提到的 OSTickISR()，而直接是 C 语言编写的 OSTimeTick()。本系统 uCOS-II 移植时占用的时钟资源是 TIMER1。

在平台初始化函数 ARMTargetInit() 中，调用 uHALr_InitTimers() 函数初始化 TIMER4 相关寄存器；调用 uHALr_InstallSystemTimer(void) 开始系统时钟，其中通过语句 SetISR_Interrupt(IRQ_TIMER4, TimerTickHandle, NULL) 将 TimerTickHandle 函数设置为 TIMER4 的中断服务函数。这些函数在文件 UHAL.C 以及 ISR.C 中。

程序中必须在开始多任务调度之后再允许时钟节拍中断，即在 OSStart() 调用过后，uCOS-II 运行的第一个任务中启动节拍中断。如果在调用 OSStart() 启动多任务调度之前就启动时钟节拍中断，uCOS-II 运行状态可能不确定而导致崩溃，请参考 uCOS-II 文献移植一节。

本系统是在系统任务 SYS_Task 中调用 uHALr_InstallSystemTimer() 函数设置 TIMER4 的 IRQ 中断的，从而启动时钟节拍。SYS_Task() 在文件 OSAddTask.C 中定义，用户不必创建，请参考本实验“完善的 uCOS-II 开发框架”。

完成了上述工作以后，uCOS-II 就可以运行在 ARM 处理器上了。

1. 编写一个简单的多任务程序来测试一下移植是否成功。

为了使 uCOS-II 可以正常运行,除了上述必须的移植工作外,硬件初始化和配置文件也是必须的。STARTUP 目录下的文件还包括中断处理,时钟,串口通信等基本功能函数。

在文件 main.c 中给出了应用程序的基本框架,包括初始化和多任务的创建,启动等。任务创建方法如下:

- 1) 在程序开头定义任务堆栈,任务函数声明和任务优先级:

```
OS_STK TaskName_Stack[STACKSIZE]={0, };    //任务堆栈
void TaskName(void *Id);                    //任务函数
#define TaskName_Prio      N                //任务优先级
```

- 2) 在 main() 函数中调用 OSTaskCreate() 函数之前用下列语句创建任务:

```
OSTaskCreate(TaskName, (void*)0, (OS_STK*)&TaskName_Stack[STACKSIZE-1],
TaskName_Prio);
```

OSTaskCreate() 函数的原型是:

```
INT8U OSTaskCreate (void (*task)(void *pd), void *p_arg, OS_STK *ptos, INT8U prio);
```

需要将任务函数 TaskName, 任务堆栈 TaskName_Stack, 任务优先级 TaskName_Prio 三个参数传给 OSTaskCreate() 函数。根据任务函数的内容决定堆栈大小,宏 STACKSIZE 定义为 4KB,可以在此基数上乘倍。任务优先级越高,TaskName_Prio 值越小;uCOS-II 可以管理 64 个任务,由 OSInit() 创建的空闲任务的优先级最低为 63;uCOS-II 保留 4 个最高和 4 个最低优先级,用户任务可以使用其余 56 个优先级值。

- 3) 编写任务函数内容:

```
void TaskName(void *Id)
{
    //添入任务初始化语句
    for(;;)
    { //添入任务循环内容
        OSTimeDly(SusPendTime); //挂起一定时间,以使其他任务可以占用 CPU
    }
}
```

uCOS-II 至少要有有一个任务,这里已经创建一个系统任务 SYS_Task, 启动系统时钟和多任务切换。

为了验证 uCOS-II 多任务切换的进行,再编写两个简单的任务,分别在超级终端上输出 run task1 和 run task2。可以参考 main.c 的结构创建多个不同功能的任务,观察个任务的切换。

2. 启动 H-JTAG 仿真器并进行初始化配置,打开 EWARM Exp4 触摸屏驱动实验中的工程。编译并下载程序到开发板观察实验现象。

七、思考题

1. UCOS-II 是如何利用定时器中断来实现多任务之间的调度的？

附录一 ARM 汇编指令集

1 ARM 指令集

一、 跳转指令

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

I. 使用专门的跳转指令。

II. 直接向程序计数器 PC 写入跳转地址值。

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

MOV LR, PC

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

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

1、 B 指令

B 指令的格式为:

B{条件} 目标地址

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

B Label ; 程序无条件跳转到标号 Label 处执行

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

BEQ Label

2、BL 指令

BL 指令的格式为：

BL {条件} 目标地址

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

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

到 R14 中

3、BLX 指令

BLX 指令的格式为：

BLX 目标地址

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

4、BX 指令

BX 指令的格式为：

BX {条件} 目标地址

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

二、数据处理指令

数据处理指令可分为数据传送指令、算术逻辑运算指令和比较指令等。

数据传送指令用于在寄存器和存储器之间进行数据的双向传输。

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

比较指令不保存运算结果，只更新 CPSR 中相应的条件标志位。

数据处理指令共以下 16 条。

2、MOV 指令

MOV 指令的格式为：

MOV {条件} {S} 目的寄存器，源操作数

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

指令示例：

MOV R1, R0 ; 将寄存器 R0 的值传送到寄存器 R1

MOV PC, R14 ; 将寄存器 R14 的值传送到 PC，常用于子程序返回

MOV R1, R0, LSL #3 ; 将寄存器 R0 的值左移 3 位后传送到 R1

3、MVN 指令

MVN 指令的格式为：

MVN{条件} {S} 目的寄存器，源操作数

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

指令示例：

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

4、CMP 指令

CMP 指令的格式为：

CMP{条件} 操作数 1，操作数 2

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

指令示例：

CMP R1, R0 ; 将寄存器 R1 的值与寄存器 R0 的值相减，并根据结果设置 CPSR 的标志位

CMPR1, #100 ; 将寄存器 R1 的值与立即数 100 相减，并根据结果设置 CPSR 的标志位

5、CMN 指令

CMN 指令的格式为：

CMN{条件} 操作数 1，操作数 2

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

指令示例：

CMN R1, R0 ; 将寄存器 R1 的值与寄存器 R0 的值相加，并根据结果设置 CPSR 的标志位

CMNR1, #100 ; 将寄存器 R1 的值与立即数 100 相加, 并根据结果设置

CPSR 的标志位

6、TST 指令

TST 指令的格式为:

TST{条件} 操作数 1, 操作数 2

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

指令示例:

TST R1, # %1 ; 用于测试在寄存器 R1 中是否设置了最低位 (%表示二进制数)

TSTR1, #0xffe ; 将寄存器 R1 的值与立即数 0xffe 按位与, 并根据结果设置 CPSR 的标志位

7、TEQ 指令

TEQ 指令的格式为:

TEQ{条件} 操作数 1, 操作数 2

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

指令示例:

TEQ R1, R2 ; 将寄存器 R1 的值与寄存器 R2 的值按位异或, 并根据

结果设置 CPSR 的标志位

8、ADD 指令

ADD 指令的格式为:

ADD{条件}{S} 目的寄存器, 操作数 1, 操作数 2

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

指令示例:

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

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

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

9、ADC 指令

ADC 指令的格式为:

ADC{条件}{S} 目的寄存器, 操作数 1, 操作数 2

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

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

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

10、SUB 指令

SUB 指令的格式为：

SUB{条件}{S} 目的寄存器，操作数 1，操作数 2

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

指令示例：

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

10、SBC 指令

SBC 指令的格式为：

SBC{条件}{S} 目的寄存器，操作数 1，操作数 2

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

指令示例：

```
SUBS    R0, R1, R2           ; R0 = R1 - R2 - !C，并根据结果设置
```

CPSR 的进位标志位

11、RSB 指令

RSB 指令的格式为：

RSB{条件}{S} 目的寄存器，操作数 1，操作数 2

RSB 指令称为逆向减法指令，用于把操作数 2 减去操作数 1，并将结果存放到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器，被移位的寄存器，或一个立即数。

数。该指令可用于有符号数或无符号数的减法运算。

指令示例：

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

12、RSC 指令

RSC 指令的格式为：

RSC{条件}{S} 目的寄存器，操作数 1，操作数 2

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

指令示例：

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

13、AND 指令

AND 指令的格式为：

AND{条件}{S} 目的寄存器，操作数 1，操作数 2

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

指令示例：

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

14、ORR 指令

ORR 指令的格式为：

ORR{条件}{S} 目的寄存器，操作数 1，操作数 2

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

指令示例：

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

15、EOR 指令

EOR 指令的格式为：

EOR{条件}{S} 目的寄存器，操作数 1，操作数 2

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

常用于反转操作数 1 的某些位。

指令示例：

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

16、BIC 指令

BIC 指令的格式为：

BIC{条件}{S} 目的寄存器，操作数 1，操作数 2

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

指令示例：

BIC R0, R0, #011 ; 该指令清除 R0 中的位 0、1、和 3，其余的位保持不变。

三、法指令与乘加指令

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

乘法指令与乘加指令共有以下 6 条：

1、MUL 指令

MUL 指令的格式为：

MUL{条件}{S} 目的寄存器，操作数 1，操作数 2

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

指令示例：

MUL R0, R1, R2 ; $R0 = R1 \times R2$

MULS R0, R1, R2 ; $R0 = R1 \times R2$ ，同时设置 CPSR 中的相关条件标志位

2、MLA 指令

MLA 指令的格式为：

MLA{条件}{S} 目的寄存器，操作数 1，操作数 2，操作数 3

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

指令示例：

MLA R0, R1, R2, R3 ; $R0 = R1 \times R2 + R3$

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

3、SMULL 指令

SMULL 指令的格式为:

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

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

指令示例:

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

4、SMLAL 指令

SMLAL 指令的格式为:

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

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

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

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

指令示例:

SMLALR0, R1, R2, R3 ; R0 = (R2 × R3) 的低 32 位 + R0
; R1 = (R2 × R3) 的高 32 位 + R1

5、UMULL 指令

UMULL 指令的格式为:

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

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

指令示例:

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

6、UMLAL 指令

UMLAL 指令的格式为:

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

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

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

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

指令示例:

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

四、程序状态寄存器访问指令

⑧ MRS 指令

MRS 指令的格式为:

MRS{条件} 通用寄存器, 程序状态寄存器 (CPSR 或 SPSR)

MRS 指令用于将程序状态寄存器的内容传送到通用寄存器中。该指令一般用在以下两种情况:

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

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

指令示例:

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

⑨ MSR 指令

MSR 指令的格式为:

MSR{条件} 程序状态寄存器 (CPSR 或 SPSR) _<域>, 操作数

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

位[31: 24]为条件标志位域, 用 f 表示;

位[23: 16]为状态位域, 用 s 表示;

位[15: 8]为扩展位域, 用 x 表示;

位[7: 0]为控制位域, 用 c 表示;

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

指令示例:

MSR CPSR, R0 ; 传送 R0 的内容到 CPSR

MSR SPSR, R0 ; 传送 R0 的内容到 SPSR

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

五、加载/存储指令

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

1、LDR 指令

LDR 指令的格式为:

LDR{条件} 目的寄存器, <存储器地址>

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

指令示例:

LDR R0, [R1] ; 将存储器地址为 R1 的字数据读入寄存器 R0。

LDR R0, [R1, R2] ; 将存储器地址为 R1+R2 的字数据读入寄存器 R0。
LDR R0, [R1, #8] ; 将存储器地址为 R1+8 的字数据读入寄存器 R0。

LDR R0, [R1, R2] ! ; 将存储器地址为 R1+R2 的字数据读入寄存器 R0,

并将新地址 R1+R2 写入 R1。

LDR R0, [R1, #8] ! ; 将存储器地址为 R1+8 的字数据读入寄存器 R0,

并将新地址 R1+8 写入 R1。

LDR R0, [R1], R2 ; 将存储器地址为 R1 的字数据读入寄存器 R0, 并

将新地址 R1+R2 写入 R1。

LDR R0, [R1, R2, LSL #2] ! ; 将存储器地址为 R1+R2×4 的字数据读入寄存器

R0, 并将新地址 R1+R2×4 写入 R1。

LDRR0, [R1], R2, LSL #2 ; 将存储器地址为 R1 的字数据读入寄存器 R0,

并将新地址 R1+R2×4 写入 R1。

2、LDRB 指令

LDRB 指令的格式为：

LDR{条件}B 目的寄存器，<存储器地址>

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

指令示例：

LDRB R0, [R1] ; 将存储器地址为 R1 的字节数据读入寄存器 R0，并

将 R0 的高 24 位清零。

LDRB R0, [R1, #8] ; 将存储器地址为 R1+8 的字节数据读入寄存器

R0，并将 R0 的高 24 位清零。

3、LDRH 指令

LDRH 指令的格式为：

LDR{条件}H 目的寄存器，<存储器地址>

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

指令示例：

LDRH R0, [R1] ; 将存储器地址为 R1 的半字数据读入寄存器 R0，并

将 R0 的高 16 位清零。

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

LDRHR0, [R1, R2] ; 将存储器地址为 R1+R2 的半字数据读入寄存器
R0，并将 R0 的高 16 位清零。

4、STR 指令

STR 指令的格式为：

STR{条件} 源寄存器，<存储器地址>

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

指令示例：

STR R0, [R1], #8 ; 将 R0 中的字数据写入以 R1 为地址的存储器中，并

将新地址 $R1+8$ 写入 $R1$ 。

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

5、STRB 指令

STRB 指令的格式为:

STR{条件}B 源寄存器, <存储器地址>

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

指令示例:

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

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

6、STRH 指令

STRH 指令的格式为:

STR{条件}H 源寄存器, <存储器地址>

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

指令示例:

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

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

六、批量数据加载/存储指令

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

LDM (或 STM) 指令

LDM (或 STM) 指令的格式为:

LDM (或 STM) {条件} {类型} 基址寄存器{!}, 寄存器列表{^}

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

- IA 每次传送后地址加 1;
- IB 每次传送前地址加 1;
- DA 每次传送后地址减 1;

DB 每次传送前地址减 1;
FD 满递减堆栈;
ED 空递减堆栈;
FA 满递增堆栈;
EA 空递增堆栈;

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

基址寄存器不允许为 R15, 寄存器列表可以为 R0~R15 的任意组合。

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

指令示例:

STMFD R13!, {R0, R4-R12, LR} ; 将寄存器列表中的寄存器 (R0, R4 到
R12, LR) 存入堆栈。

LDMFD R13!, {R0, R4-R12, PC} ; 将堆栈内容恢复到寄存器 (R0, R4 到
R12, LR)。

七、数据交换指令

1、SWP 指令

SWP 指令的格式为:

SWP {条件} 目的寄存器, 源寄存器 1, [源寄存器 2]

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

指令示例:

SWP R0, R1, [R2] ; 将 R2 所指向的存储器中的字数据传送
到 R0, 同时 将 R1 中的字数据传送到 R2 所指
向的存储单元。

SWP R0, R0, [R1] ; 该指令完成将 R1 所指向的存储器中的字数据
与 R0 中的数据
交换。

2、SWPB 指令

SWPB 指令的格式为:

SWPB {条件}B 目的寄存器, 源寄存器 1, [源寄存器 2]

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

指令示例:

SWPB R0, R1, [R2] ; 将 R2 所指向的存储器中的字节数据传送到 R0, R0 的高 24

位清零, 同时将 R1 中的低 8 位数据传送到 R2 所指向的存储单元。

SWPB R0, R0, [R1] ; 该指令完成将 R1 所指向的存储器中的字节数据与

R0 中的低 8 位数据交换。

八、移位指令（操作）

1、LSL（或 ASL）操作

LSL（或 ASL）操作的格式为：

通用寄存器, LSL（或 ASL） 操作数

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

操作示例

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

2、LSR 操作

LSR 操作的格式为：

通用寄存器, LSR 操作数

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

操作示例：

MOV R0, R1, LSR#2 ; 将 R1 中的内容右移两位后传送到 R0 中，左端用

零来填充。

3、ASR 操作

ASR 操作的格式为：

通用寄存器, ASR 操作数

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

操作示例：

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

4、ROR 操作

ROR 操作的格式为：

通用寄存器, ROR 操作数

ROR 可完成对通用寄存器中的内容进行循环右移的操作，按操作数所指定的数量向右循

环移位,左端用右端移出的位来填充。其中,操作数可以是通用寄存器,也可以是立即数(0~31)。显然,当进行32位的循环右移操作时,通用寄存器中的值不改变。

操作示例:

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

5、RRX 操作

RRX 操作的格式为:

通用寄存器,RRX 操作数

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

操作示例:

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

九、协处理器指令

1、CDP 指令

CDP 指令的格式为:

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

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

指令示例:

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

2、LDC 指令

LDC 指令的格式为:

LDC{条件}{L} 协处理器编码,目的寄存器,[源寄存器]

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

指令示例:

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

3、STC 指令

STC 指令的格式为：

STC{条件}{L} 协处理器编码, 源寄存器, [目的寄存器]

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

指令示例：

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

4、MCR 指令

MCR 指令的格式为：

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

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

指令示例：

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

5、MRC 指令

MRC 指令的格式为：

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

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

指令示例：

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

十、异常产生指令

1、SWI 指令

SWI 指令的格式为：

SWI {条件} 24 位的立即数

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

指令示例：

SWI 0x02 ; 该指令调用操作系统编号位 02 的系统例程。

2、BKPT 指令

BKPT 指令的格式为：

BKPT 16 位的立即数

BKPT 指令产生软件断点中断，可用于程序的调试。

2 ARM 汇编器所支持的伪指令

在 ARM 汇编语言程序里，有一些特殊指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常称这些特殊指令助记符为伪指令，他们所完成的操作称为伪操作。伪指令在源程序中的作用是为完成汇编程序作各种准备工作的，这些伪指令仅在汇编过程中起作用，一旦汇编结束，伪指令的使命就完成。

在 ARM 的汇编程序中，有如下 4 种伪指令：符号定义伪指令、数据定义伪指令、汇编控制伪指令、宏指令以及其他伪指令。

一、符号定义（Symbol Definition）伪指令

符号定义伪指令用于定义 ARM 汇编程序中的变量、对变量赋值以及定义寄存器的别名等操作。常见的符号定义伪指令有如下 4 种：

1、GBLA、GBLL 和 GBLS

语法格式：

GBLA (GBLL 或 GBLS) 全局变量名

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

GBLA 伪指令用于定义一个全局的数字变量，并初始化为 0；

GBLL 伪指令用于定义一个全局的逻辑变量，并初始化为 F（假）；

GBLS 伪指令用于定义一个全局的字符串变量，并初始化为空；

由于以上三条伪指令用于定义全局变量，因此在整个程序范围内变量名必须唯一。

使用示例：

GBLA Test1 ; 定义一个全局的数字变量，变量名为 Test1

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

2、LCLA、LCLL 和 LCLS

语法格式:

LCLA (LCLL 或 LCLS) 局部变量名

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

LCLA 伪指令用于定义一个局部的数字变量, 并初始化为 0;

LCLL 伪指令用于定义一个局部的逻辑变量, 并初始化为 F (假);

LCLS 伪指令用于定义一个局部的字符串变量, 并初始化为空;

以上三条伪指令用于声明局部变量, 在其作用范围内变量名必须唯一。

使用示例:

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

3、SETA、SETL 和 SETS

语法格式:

变量名 SETA (SETL 或 SETS) 表达式

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

SETA 伪指令用于给一个数字变量赋值;

SETL 伪指令用于给一个逻辑变量赋值;

SETS 伪指令用于给一个字符串变量赋值;

其中, 变量名为已经定义过的全局变量或局部变量, 表达式为将要赋给变量的值。

使用示例:

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

4、RLIST

语法格式:

名称 RLIST {寄存器列表}

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

使用示例:

RegList RLIST {R0-R5, R8, R10} ; 将寄存器列表名称定义为 RegList, 可

在 ARM 指令 LDM/STM 中通过该名称访

问寄存器列表。

二、数据定义 (Data Definition) 伪指令

数据定义伪指令一般用于为特定的数据分配存储单元,同时可完成已分配存储单元的初始化。常见的数据定义伪指令有如下 9 种:

1、DCB

语法格式:

标号 DCB 表达式

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

使用示例:

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

2、DCW (或 DCWU)

语法格式:

标号 DCW (或 DCWU) 表达式

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

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

使用示例:

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

3、DCD (或 DCDU)

语法格式:

标号 DCD (或 DCDU) 表达式

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

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

使用示例:

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

4、DCFD（或 DCFDU）

语法格式：

标号 DCFD（或 DCFDU） 表达式

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

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

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

5、DCFS（或 DCFSU）

语法格式：

标号 DCFS（或 DCFSU） 表达式

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

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

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

6、DCQ(或 DCQU)

语法格式：

标号 DCQ（或 DCQU） 表达式

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

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

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

7、SPACE

语法格式：

标号 SPACE 表达式

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

使用示例：

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

8、MAP

语法格式：

MAP 表达式{, 基址寄存器}

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

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

MAP 伪指令通常与 FIELD 伪指令配合使用来定义结构化的内存表。

使用示例：

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

9、FIELD

语法格式：

标号 FIELD 表达式

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

表达式的值为当前数据域在内存表中所占的字节数。

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

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

使用示例：

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

三、汇编控制（Assembly Control）伪指令

汇编控制伪指令用于控制汇编程序的执行流程，常用的汇编控制伪指令包括以下 4 条：

1、IF、ELSE、ENDIF

语法格式：

IF 逻辑表达式
指令序列 1
ELSE
指令序列 2
ENDIF

IF、ELSE、ENDIF 伪指令能根据条件的成立与否决定是否执行某个指令序列。当 IF 后面的逻辑表达式为真，则执行指令序列 1，否则执行指令序列 2。其中，ELSE 及指令序列 2 可以没有，此时，当 IF 后面的逻辑表达式为真，则执行指令序列 1，否则继续执行后面的指令。

IF、ELSE、ENDIF 伪指令可以嵌套使用。

使用示例：

GBLL Test ; 声明一个全局的逻辑变量, 变量名为 Test

.....

IF Test = TRUE

指令序列 1

ELSE

指令序列 2

ENDIF

2、WHILE、WEND

语法格式:

WHILE 逻辑表达式

指令序列

WEND

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

WHILE、WEND 伪指令可以嵌套使用。

使用示例:

GBLA Counter ; 声明一个全局的数学变量, 变量名为 Counter

Counter SETA 3 ; 由变量 Counter 控制循环次数

.....

WHILE Counter < 10

指令序列

WEND

3、MACRO、MEND

语法格式:

\$标号 宏名 \$参数 1, \$参数 2,

指令序列

MEND

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

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

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

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

MACRO、MEND 伪指令可以嵌套使用。

4、MEXIT

语法格式：

MEXIT

MEXIT 用于从宏定义中跳转出去。

四、其他常用的伪指令

还有一些其他的伪指令，在汇编程序中经常会被使用，包括以下 13 条：

1、AREA

语法格式：

AREA 段名 属性 1, 属性 2,

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

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

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

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

使用示例：

```
AREA Init, CODE, READONLY
```

指令序列

；该伪指令定义了一个代码段，段名为 Init，属性为只读

2、ALIGN

语法格式：

ALIGN {表达式{, 偏移量}}

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

使用示例：

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

指令序列

```
END
```

3、CODE16、CODE32

语法格式：

CODE16（或 CODE32）

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

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

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

使用示例：

```
AREA Init, CODE, READONLY
```

```
.....
```

```
CODE32 ; 通知编译器其后的指令为 32 位的 ARM 指令
```

```
LDR R0, =NEXT+1 ; 将跳转地址放入寄存器 R0
```

```
BX R0 ; 程序跳转到新的位置执行，并将处理器切换到 Thumb 工作状态
```

```
.....
```

```
CODE16 ; 通知编译器其后的指令为 16 位的 Thumb 指令
```

```
NEXT LDR R3, =0x3FF
```

```
.....
```

```
END ; 程序结束
```

4、ENTRY

语法格式：

ENTRY

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

里最多只能有一个 ENTRY（可以没有）。

使用示例：

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

5、END

语法格式：

END

END 伪指令用于通知编译器已经到了源程序的结尾。

使用示例：

```
AREA Init, CODE, READONLY
.....
END                               ; 指定应用程序的结尾
```

6、EQU

语法格式：

名称 EQU 表达式 {, 类型}

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

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

CODE16、CODE32 和 DATA

使用示例：

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

7、EXPORT（或 GLOBAL）

语法格式：

EXPORT 标号 {[WEAK]}

EXPORT 伪指令用于在程序中声明一个全局的标号，该标号可在其他的文件中引用。

EXPORT 可用 GLOBAL 代替。标号在程序中区分大小写，[WEAK]选项声明其他的同名标号优先于该标号被引用。

使用示例：

```
AREA Init, CODE, READONLY
```

EXPORT Stest ; 声明一个可全局引用的标号 Stest

.....

END

8、IMPORT

语法格式:

IMPORT 标号 {[WEAK]}

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

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

使用示例:

AREA Init, CODE, READONLY

IMPORT Main ; 通知编译器当前文件要引用标号 Main,

但 Main 在其他源文件中定义

.....

END

9、EXTERN

语法格式:

EXTERN 标号 {[WEAK]}

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

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

使用示例:

AREA Init, CODE, READONLY

EXTERN Main ; 通知编译器当前文件要引用标号 Main,

但 Main 在其他源文件中定义

.....

END

10、GET (或 INCLUDE)

语法格式:

GET 文件名

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

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

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

```
AREA    Init, CODE, READONLY
```

```
GETa1.s                        ; 通知编译器当前源文件包含源文件 a1.s
```

```
GE T    C: \a2.s                ; 通知编译器当前源文件包含源文件 C: \a2.s
```

```
.....
```

```
END
```

11、INCBIN

语法格式：

INCBIN 文件名

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

使用示例：

```
AREA    Init, CODE, READONLY
```

```
INCBIN        a1.dat            ; 通知编译器当前源文件包含文件 a1.dat
```

```
INCBIN C: \a2.txt               ; 通知编译器当前源文件包含文件 C: \a2.txt
```

```
.....
```

```
END
```

12、RN

语法格式：

名称 RN 表达式

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

使用示例：

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

13、ROUT

语法格式：

{名称} ROUT

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

附录二 嵌入式系统应用编程 API 函数

1. 显示部分 Display.h

相关结构:

```
typedef struct{
    int DrawPointx;
    int DrawPointy;    //绘图所使用的坐标点
    int PenWidth;      //画笔宽度
    U32 PenMode;       //画笔模式
    COLORREF PenColor; //画笔的颜色
    int DrawOrgx;      //绘图的坐标原点位置
    int DrawOrgy;
    int WndOrgx;       //绘图的窗口坐标位置
    int WndOrgy;
    int DrawRangex;    //绘图的区域范围
    int DrawRangey;
    structRECT DrawRect; //绘图的有效范围
    U8 bUpdataBuffer;  //是否更新后台缓冲区及显示
    U32 Fontcolor;     //字符颜色
}DC,*PDC
typedef struct {
    int left;
    int top;
    int right;
    int bottom;
}structRECT
```

相关函数:

initOSDC

定义: void initOSDC()

功能: 初始化系统的绘图设备上下文 (DC), 为 DC 的动态分配开辟内存空间

CreateDC

定义: PDC CreateDC()

功能: 创建一个绘图设备上下文 (DC), 返回指向 DC 的指针

DestoryDC

定义: void DestoryDC(PDC pdc)

功能: 删除绘图设备上下文 (DC), 释放相应的资源

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

SetPixel

定义: void SetPixel(PDC pdc, int x, int y, COLORREF color)

功能: 设置指定点的像素颜色到 LCD 的后台缓冲区, LCD 范围以外的点将被忽略

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

x,y: 指定的像素坐标

color: 指定的像素的颜色, 高 8 位为空, 接下来的 24 位分别对应 RGB 颜色的 8 位码。

SetPixelOR

定义: void SetPixelOR(PDC pdc, int x, int y, COLORREF color)

功能: 设置指定点的像素颜色和 LCD 的后台缓冲区的对应点或运算, LCD 范围以外的点将被忽略

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

x,y: 指定的像素坐标

color: 指定的像素的颜色, 高 8 位为空, 接下来的 24 位分别对应 RGB 颜色的 8 位码。

SetPixelAND

定义: void SetPixelAND(PDC pdc, int x, int y, COLORREF color)

功能: 设置指定点的像素颜色和 LCD 的后台缓冲区的对应点与运算, LCD 范围以外的点将被忽略

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

x,y: 指定的像素坐标

color: 指定的像素的颜色, 高 8 位为空, 接下来的 24 位分别对应 RGB 颜色的 8 位码。

SetPixelXOR

定义: void SetPixelXOR(PDC pdc, int x, int y, COLORREF color)

功能: 设置指定点的像素颜色和 LCD 的后台缓冲区的对应点异或运算, LCD 范围以外的点将被忽略

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

x,y: 指定的像素坐标

color: 指定的像素的颜色, 高 8 位为空, 接下来的 24 位分别对应 RGB 颜色的 8 位码。

GetFontHeight

定义: int GetFontHeight(U8 fnt)

功能: 返回指定字体的高度

参数说明: fnt: 输出字体的大小型号, 可以是表 1-1 数值中的一种:

表 1-1 字体大小

字体的型号	数值	说明
FONTSIZE_SMALL	1	小字体模式, 12x12 字符
FONTSIZE_MIDDLE	2	中字体模式, 16x16 字符
FONTSIZE_BIG	3	大字体模式, 24x24 字符

TextOut

定义: void TextOut(PDC pdc, int x, int y, U16 *ch, U8 bunicode, U8 fnt)

功能: 在 LCD 屏幕上显示文字

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

x,y: 所输出文字左上角的屏幕坐标

ch: 指向输出文字字符串的指针

bunicode: 是否为 Unicode 编码, 如果是 TRUE, 表示 ch 指向的字符串为 Unicode 字符集; 如果为 FALSE, 表示表示 ch 指向的字符串为 GB 字符集。

fnt: 指定字体的大小型号, 可以是表 1-1 数值中的一种或上表 1-2 的数值:

表 1-2 字的显示方式

显示模式	数值	说明
FONT_NORMAL	0	正常显示
FONT_TRANSPARENT	4	透明背景
FONT_BLACKBK	8	黑底白字

TextOutRect

定义: void TextOutRect(PDC pdc, structRECT* prect, U16* ch, U8 bunicode, U8 fnt, U32 outmode)

功能: 在指定矩形的范围内显示文字, 超出的部分将被裁减

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

prect: 所输出文字的矩形范围

ch: 指向输出文字字符串的指针

bunicode: 是否为 Unicode 编码, 如果是 TRUE, 表示 ch 指向的字符串为 Unicode 字符集; 如果为 FALSE, 表示表示 ch 指向的字符串为 GB 字符集。

fnt: 指定字体的大小型号, 可以是表 1-1 数值中的一种或表 1-2 的数值

outmode: 指定矩形中文字的对齐方式, 可以是表 1-3 中的数值

表 1-3 矩形中文字的对齐方式

对齐方式	数值	说明
TEXTOUT_LEFT_UP	0	文字从左上角开始
TEXTOUT_MID_X	1	水平居中
TEXTOUT_MID_Y	2	垂直居中

MoveTo

定义: void MoveTo(PDC pdc, int x, int y)

功能：把绘图点移动到指定的坐标

参数说明：pdc: 指向绘图设备上下文（DC）的指针

x,y: 移动画笔到绘图点的屏幕坐标

LineTo

定义：void LineTo(PDC pdc, int x, int y)

功能：在屏幕上画线。从当前画笔的位置画直线到指定的坐标位置，并使画笔停留在当前指定的位置

参数说明：pdc: 指向绘图设备上下文（DC）的指针

x,y: 直线绘图目的点的屏幕坐标

DrawRectFrame

定义：void DrawRectFrame(PDC pdc, int left,int top ,int right, int bottom)

功能：在屏幕上绘制指定大小的矩形方框。

参数说明：pdc: 指向绘图设备上下文（DC）的指针

left: 绘制矩形的左边框位置

right: 绘制矩形的右边框位置

top: 绘制矩形的上边框位置

bottom: 绘制矩形的下边框位置

DrawRectFrame2

定义：void DrawRectFrame2(PDC pdc, structRECT *rect)

功能：在屏幕上绘制指定大小的矩形方框。

参数说明：pdc: 指向绘图设备上下文（DC）的指针

rect: 绘制矩形的位置及大小

FillRect

定义：void FillRect(PDC pdc, int left,int top ,int right, int bottom,U32 DrawMode , COLORREF color)

功能：在屏幕上填充指定大小的矩形。

参数说明：pdc: 指向绘图设备上下文（DC）的指针

left: 绘制矩形的左边框位置

right: 绘制矩形的右边框位置

top: 绘制矩形的上边框位置

bottom: 绘制矩形的下边框位置

DrawMode: 矩形的填充模式和颜色，它的数值可以是表 1-4 中的一种和表 1-5 中的或运算的结果

color: 填充的颜色值，高 8 位为空，接下来的 24 位分别对应 RGB 颜色的 8 位码。

表 1-4 绘图模式

绘图模式	数值	说明
GRAPH_MODE_NORMAL	0x00	普通绘图模式
GRAPH_MODE_OR	0x10	或 绘图模式
GRAPH_MODE_AND	0x20	与 绘图模式
GRAPH_MODE_XOR	0x30	异或 绘图模式

表 1-5 前景颜色

图形显示模式	数值	说明
COLOR_BLACK	1	黑色前景色

COLOR_WHITE

0

白色前景色

FillRect2

定义: void FillRect2(PDC pdc, structRECT *rect,U32 DrawMode , COLORREF color)

功能: 在屏幕上填充指定大小的矩形。

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

rect: 绘制矩形的位置及大小

DrawMode: 矩形的填充模式和颜色, 它的数值可以是表 1-4 中的一种和表 1-5 中的或运算的结果。

color: 填充的颜色值, 高 8 位为空, 接下来的 24 位分别对应 RGB 颜色的 8 位码。

ClearScreen

定义: void ClearScreen()

功能: 清除整个屏幕的绘图缓冲区, 即: 清空 LCDBuffer2

SetPenWidth

定义: U8 SetPenWidth(PDC pdc, U8 width)

功能: 设置画笔的宽度, 并返回以前的画笔宽度

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

width: 画笔的宽度, 默认值是 1, 即一个像素点宽

SetPenMode

定义: void SetPenMode(PDC pdc, U32 mode)

功能: 设置画笔画图的模式

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

mode: 绘图的更新模式, 可以是表 1-4 数值中的一种

Circle

定义: void Circle(PDC pdc, int x0, int y0, int r)

功能: 绘制指定圆心和半径的圆

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

x0,y0: 圆心坐标

r: 圆的半径

ArcTo

定义: void ArcTo(PDC pdc, int x1,int y1, U8 arctype, int R)

功能: 绘制圆弧, 从画笔的当前位置绘制指定圆心的圆弧到给定的位置

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

x1,y1: 绘制圆弧的目的位置

arctype: 圆弧的方向可以是表 1-6 参数中的一种:

R: 圆弧的半径

表 1-6 圆弧的方向

圆弧绘制模式	数值	说明
GRAPH_ARC_BACKWARD	0	逆时针画圆
GRAPH_ARC_FORWARD	1	顺时针画圆

SetLCDUpdata

定义: U8 SetLCDUpdata(PDC pdc, U8 isUpdata)

功能: 设定绘图的时候是否及时的更新 LCD 的显示, 返回以前的更新模式

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

isUpdate: 是否更新 LCD 的显示, 可以为 TRUE 或者 FALSE。如果选择及时更新则每调用一次绘图的函数都要更新 LCD 的后台缓冲区并把后台缓冲区复制到前台, 虽然可以保证绘图的实时性, 但是, 总体来讲, 是降低了绘图效率。

Draw3DRect

定义: void Draw3DRect(PDC pdc, int left,int top, int right, int bottom, COLORREF color1,COLORREF color2)

功能: 绘制指定大小和风格的 3D 边框的矩形

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

left: 绘制矩形的左边框位置

right: 绘制矩形的右边框位置

top: 绘制矩形的上边框位置

bottom: 绘制矩形的下边框位置

color1: 左和上的边框颜色, 高 8 位为空, 接下来的 24 位分别对应 RGB 颜色的 8 位码。

color2: 右和下的边框颜色, 高 8 位为空, 接下来的 24 位分别对应 RGB 颜色的 8 位码。

Draw3DRect2

定义: void Draw3DRect2(PDC pdc, structRECT rect, COLORREF color1,COLORREF color2)

功能: 绘制指定大小和风格的 3D 边框的矩形

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

rect: 绘制矩形的位置及大小

color1: 左和上的边框颜色, 高 8 位为空, 接下来的 24 位分别对应 RGB 颜色的 8 位码。

color2: 右和下的边框颜色, 高 8 位为空, 接下来的 24 位分别对应 RGB 颜色的 8 位码。

GetPenWidth

定义: U8 GetPenWidth(PDC pdc)

功能: 返回当前绘图设备上下文 (DC) 画笔的宽度

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

GetPenMode

定义: U32 GetPenMode(PDC pdc)

功能: 返回当前绘图设备上下文 (DC) 画笔的模式

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

SetPenColor

定义: U32 SetPenColor(PDC pdc, U32 color)

功能: 设定画笔的颜色, 返回当前绘图设备上下文 (DC) 画笔的颜色

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

color: 画笔的颜色, 高 8 位为空, 接下来的 24 位分别对应 RGB 颜色的 8 位码。

GetPenColor

定义: U32 GetPenColor(PDC pdc)

功能: 返回当前绘图设备上下文 (DC) 画笔的颜色

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

GetBmpSize

定义: void GetBmpSize(char filename[], int* Width, int* Height)

功能: 取得指定位图文件位图的大小

参数说明: filename[]: 位图文件的文件名

Width: 位图的宽

Height: 位图的高

ShowBmp

定义: void ShowBmp(PDC pdc, char filename[], int x, int y)

功能: 显示指定的位图 (Bitmap) 文件, 到指定的坐标

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

filename[]: 显示的位图 (Bitmap) 文件名

x,y: 显示位图的左上角坐标

SetDrawOrg

定义: void SetDrawOrg(PDC pdc, int x,int y, int* oldx, int *oldy)

功能: 设置绘图设备上下文 (DC) 的原点

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

x,y: 设定的新原点

oldx,oldy: 返回的以前原点的位置

SetDrawRange

定义: void SetDrawRange(PDC pdc, int x,int y, int* oldx, int *oldy)

功能: 设置绘图设备上下文 (DC) 的绘图范围

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

x,y: 设定的横向、纵向绘图的范围, 如果 x (或者 y) 为 1, 则表示 x (或者 y) 方向的比例随着 y (或者 x) 方向的范围按比例缩放。如果参数为-1, 表示方向相反

oldx,oldy: 返回的以前横向、纵向绘图的范围

LineToDelay

定义: void LineToDelay(PDC pdc, int x, int y, int ticks)

功能: 在屏幕上画线。从当前画笔的位置画直线到指定的坐标位置, 并使画笔停留在当前指定的位置

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

x,y: 直线绘图目的点的屏幕坐标

ticks: 指定的延时时间, 系统的时间单位

ArcToDelay

定义: void ArcToDelay(PDC pdc, int x1, int y1, U8 arctype, int R, int ticks)

功能: 按照指定的延时时间绘制圆弧, 从画笔的当前位置绘制指定圆心的圆弧到给定的位置

参数说明: pdc: 指向绘图设备上下文 (DC) 的指针

x1,y1: 绘制圆弧的目的位置

arctype: 圆弧的方向可以是表 1-6 参数中的一种:

R: 圆弧的半径

ticks: 指定的延时时间, 系统的时间单位

2. 操作系统的消息相关函数 OSMessage.h

相关结构:

```
typedef struct {  
    POS_Ctrl pOSCtrl;    //消息所发到的窗口(控件)  
    U32 Message;  
    U32 WParam;  
    U32 LParam;  
}OSMSG, *POSMSG
```

相关函数:

initOSMessage

定义: void initOSMessage()

功能: 操作系统初始化消息, 为消息队列分配内存空间

OSCreateMessage

定义: POSMSG OSCreateMessage(POS_Ctrl pOSCtrl, U32 Message, U32 wparam, U32 lparam)

功能: 向指定的控件创建消息返回指向消息的指针

参数说明: pOSCtrl: 指向控件的指针, 为 NULL 时指桌面

Message: 发送的消息类型, 可以是表 2-1 中的一种

wparam: 随消息发送的附加参数, 参见表 2-2

lparam: 随消息发送的附加参数

表 2-1 系统消息类型

消息类型	数值	说明
OSM_KEY	1	键盘消息
OSM_TOUCH_SCREEN	2	触摸屏消息
OSM_LISTCTRL_SELCHANGE	1001	列表框的选择被改变的消息
OSM_LISTCTRL_SELDCLICK	1002	列表框的选择双击消息
OSM_BUTTON_CLICK	003	单击按钮消息

表 2-2 系统消息参数

消息参数	wparam	lparam
OSM_KEY	键盘扫描码	
OSM_TOUCH_SCREEN	低 16 位存放了触摸点的 x 坐标值, 高 16 位存放了触摸点的 y 坐标值	触摸动作
OSM_LISTCTRL_SELCHANGE	CtrlID	CurrentSel

OSM_LISTCTRL_SELDBCLICK	CtrlID	CurrentSel
OSM_BUTTON_CLICK	CtrlID	

SendMessage

定义: U8 SendMessage(POSMSG pmsg)

功能: 发送消息, 添加消息到消息队列中, 如果队列以慢则返回 FALSE, 否则返回 TRUE

参数说明: pmsg: 指向发送消息的指针

WaitMessage

定义: POSMSG WaitMessage(INT16U timeout)

功能: 在超时的时间内等待消息, 收到消息时返回指向消息结构的指针

参数说明: timeout: 消息等待的超时设定, 如果为 0, 表示没有超时时间

DeleteMessage

定义: void DeleteMessage(POSMSG pMsg)

功能: 删除指定的消息结构, 释放相应的内存

参数说明: pMsg: 指向所要删除消息的指针

3. 控件的相关函数 Control.h

相关结构:

```
typedef struct typeWnd{
    U32 CtrlType;    //控件的类型
    U32 CtrlID;
    structRECT WndRect; //窗口的位置和大小
    structRECT ClientRect; //看翱谕户区域
    U32 FontSize;    //窗口的字符大小
    U32 style;        //窗口的的边框风格
    U8 bVisible;      //是否可见
    struct typeWnd* parentWnd; //控件的父窗口指针
    U8 (*CtrlMsgCallBk)(void*);
    PDC pdc;          //窗口的绘图设备上下文
    U16 Caption[20];  //窗口标题
    List ChildWndList;
    U32 FocusCtrlID;  //子窗口焦点 ID
    U32 preParentFocusCtrlID; //显示窗口之前的父窗口焦点 ID
    OS_EVENT* WndDC_Ctrl_mem; //窗口 DC 控制权
}Wnd, *PWnd

typedef struct {
    U32 CtrlType;    //控件的类型
    U32 CtrlID;
    structRECT ListCtrlRect; //控件的位置和大小
    structRECT ClientRect; //客户区域
    U32 FontSize;    //控件的字符大小
    U32 style;        //控件的的边框风格
```



```
    U8 bVisible;    //是否可见
    PWnd parentWnd; //控件的父窗口指针
    U8 (*CtrlMsgCallBk)(void*);
}OS_Ctrl, *POS_Ctrl

typedef struct{
    U32 CtrlType;    //控件的类型
    U32 CtrlID;
    structRECT ListCtrlRect;    //列表框的位置和大小
    structRECT ClientRect; //列表框列表区域
    U32 FontSize;
    U32 style;    //列表框的风格
    U8 bVisible;    //是否可见
    PWnd parentWnd; //控件的父窗口指针
    U8 (*CtrlMsgCallBk)(void*);
    U16 **pListText;    //列表框所容纳的文本指针
    int ListMaxNum; //列表框所容纳的最大文本的行数
    int ListNum;    //列表框所容纳的文本的行数
    int ListShowNum;    //列表框所能显示的文本行数
    int CurrentHead;    //列表的表头号
    int CurrentSel; //当前选中的列表项号
    structRECT ListCtrlRollRect;    //列表框滚动条方框
    structRECT RollBlockRect;    //列表框滚动条滑块方框
}ListCtrl, *PListCtrl

typedef struct{
    U32 CtrlType;    //控件的类型
    U32 CtrlID; //控件的 ID
    structRECT TextCtrlRect;    //文本框的位置和大小
    structRECT ClientRect; //客户区域
    U32 FontSize;    //文本框的字符大小
    U32 style;    //文本框的风格
    U8 bVisible;    //是否可见
    PWnd parentWnd; //控件的父窗口指针
    U8 (*CtrlMsgCallBk)(void*);
    U8 bIsEdit; //文本框是否处于编辑状态
    char* KeyTable; //文本框的字符映射表
    U16 text[40];    //文本框中的字符块
}TextCtrl, *PTextCtrl

typedef struct{
    U32 CtrlType;    //控件的类型
    U32 CtrlID;
    structRECT PictureCtrlRect; //图片框的位置和大小
    structRECT ClientRect; //客户区域
    U32 FontSize;    //图片框的字符大小
    U32 style;    //图片框的风格
    U8 bVisible;    //是否可见
    PWnd parentWnd; //控件的父窗口指针
    U8 (*CtrlMsgCallBk)(void*);

    char picfilename[12]; //图片文件名
}PictureCtrl, *PPictureCtrl

typedef struct {
```



```
U32 CtrlType;    //控件的类型
U32 CtrlID;
structRECT ButtonCtrlRect; //控件的位置和大小
structRECT ClientRect; //客户区域
U32 FontSize;    //控件的字符大小
U32 style;        //控件的的边框风格
U8 bVisible;     //是否可见
PWnd parentWnd;  //控件的父窗口指针
U8 (*CtrlMsgCallBk)(void*);
U16 Caption[10]; //按钮标题
}ButtonCtrl, *PButtonCtrl
```

相关函数:

initOSCtrl

定义: void initOSCtrl()

功能: 初始化系统的控件, 为动态创建控件分配空间

SetWndCtrlFocus

定义: U32 SetWndCtrlFocus(PWnd pWnd, U32 CtrlID)

功能: 设置窗口中控件的焦点, 返回原来窗口控件焦点的 ID

参数说明: pWnd: 指向窗口的指针, 如果是 NJLL, 表示没有父窗口, 属于桌面

CtrlID: 焦点控件的 ID

GetWndCtrlFocus

定义: U32 GetWndCtrlFocus(PWnd pWnd)

功能: 得到窗口中焦点控件的 ID

参数说明: pWnd: 指向窗口的指针, 如果是 NJLL, 表示没有父窗口, 属于桌面

ReDrawOSCtrl

定义: void ReDrawOSCtrl()

功能: 绘制所有的操作系统可见的控件

GetCtrlfromID

定义: OS_Ctrl* GetCtrlfromID(U32 ctrlID)

功能: 由指定控件的 ID 返回控件的指针, 如果没有这个控件则返回 NULL。控件的 ID 是系统运行过程中唯一的, 它可以用来标识控件

参数说明: ctrlID: 控件的 ID

CreateOSCtrl

定义: OS_Ctrl* CreateOSCtrl(U32 CtrlID, U32 CtrlType, structRECT* prect, U32 FontSize, U32 style, PWnd parentWnd)

功能: 创建控件, 为控件动态分配内存空间, 返回指向控件的指针

参数说明: CtrlID: 创建控件的 ID, 此控件 ID 必须是唯一的

CtrlType: 控件的类型, 可以是表 3-1 数值的一种

prect: 指向控件大小和位置的指针

FontSize: 控件显示文字的字体大小, 可以是表 1-1 数值中的一种

style: 控件的风格, 可以是表 3-1 中的一种

parentWnd: 指向控件父窗口的指针, 如果是 NJLL, 表示没有父窗口, 控件属于桌面

表 3-1 控件风格

控件的显示模式	数值	说明
CTRL_STYLE_DBFRAME	1	双重边框
CTRL_STYLE_FRAME	2	单边框
CTRL_STYLE_3DUPFRAME	3	突起 3D 边框
CTRL_STYLE_3DDOWNFRAME	4	凹陷 3D 无边框
CTRL_STYLE_NOFRAME	5	无边框

SetCtrlMessageCallBk

定义: void SetCtrlMessageCallBk(POS_Ctrl pOSCtrl, U8(*CtrlMsgCallBk)(void*))

功能: 设置控件的消息回调函数。系统收到发给此控件的消息的时候, 调用此回调函数, 如果控件的消息回调函数, 返回 TRUE 的时候, 则控件本身不继续处理消息, 返回 FALSE 的时候, 消息继续发给控件本身处理。

参数说明: pOSCtrl: 指向控件的指针

U8(*CtrlMsgCallBk)(void*): 控件的消息回调函数

OSOnSysMessage

定义: void OSOnSysMessage(void* pMsg)

功能: 系统的消息处理函数, 当收到系统消息的时候, 调用此函数, 把消息传递给各个控件

参数说明: pMsg: 指向消息结构的指针

ShowCtrl

定义: void ShowCtrl(OS_Ctrl *pCtrl, U8 bVisible)

功能: 设定指定的控件是否可见

参数说明: pCtrl: 指向控件的指针

bVisible: 控件是否可见。如果为 TRUE, 则可见; 如果为 FALSE, 则不可见

CreateListCtrl

定义: PListCtrl CreateListCtrl(U32 CtrlID, structRECT* rect, int MaxNum, U32 FontSize, U32 style, PWnd parentWnd)

功能: 创建列表框控件, 返回指向列表框的指针

参数说明: CtrlID: 创建的列表框控件的 ID, 此控件 ID 必须是唯一的

rect: 指向控件大小和位置的指针

MaxNum: 列表框所能列出的最大列表项目数

FontSize: 列表框的字体大小, 可以是表 1-1 数值中的一种

style: 列表框的风格, 可以是表 3-1 中的一种

parentWnd: 指向控件父窗口的指针, 如果是 NJLL, 表示没有父窗口, 空间属于桌面

AddStringListCtrl

定义: U8 AddStringListCtrl(PListCtrl pListCtrl, U16 string[])

功能: 向指定的列表框中添加字符串, 字符串的最大长度为 64 字符

参数说明: pListCtrl: 指向列表框的指针

string: 向列表框中添加的字符串的指针

ListCtrlReMoveAll

定义: void ListCtrlReMoveAll(PListCtrl pListCtrl)

功能: 删除列表框中所有的文本

参数说明: pListCtrl: 指向列表框的指针

DrawListCtrl

定义: void DrawListCtrl(PListCtrl pListCtrl)

功能: 绘制指定的列表框

参数说明: pListCtrl: 指向列表框的指针

ListCtrlSelMove

定义: void ListCtrlSelMove(PListCtrl pListCtrl, int moveNum, U8 Redraw)

功能: 移动列表框高亮度条, 正数下移, 负数上移

参数说明: pListCtrl: 指向列表框的指针

moveNum: 高亮度条移动的相对位置, 正数下移, 负数上移

Redraw: 是否重新绘制空间, 如果为 TRUE, 则重绘; 如果为 FALSE, 则不重绘

ListCtrlOnTchScr

定义: void ListCtrlOnTchScr(PListCtrl pListCtrl, int x, int y, U32 tchaction)

功能: 列表框的触摸屏响应函数, 当有触摸屏消息的时候, 系统自动调用

参数说明: pListCtrl: 指向列表框的指针

x,y: 触摸屏的屏幕坐标

tchaction: 触摸屏的消息可以是下表中的一项

表 3-2 触摸屏动作

触摸屏消息	数值	说明
TCHSCR_ACTION_NULL	0	触摸屏空消息
TCHSCR_ACTION_CLICK	1	触摸屏单击
TCHSCR_ACTION_DBLCLICK	2	触摸屏双击
TCHSCR_ACTION_DOWN	3	触摸屏按下
TCHSCR_ACTION_UP	4	触摸屏抬起
TCHSCR_ACTION_MOVE	5	触摸屏移动

ReLoadListCtrl

定义: void ReLoadListCtrl(PListCtrl pListCtrl,U16* string[],int nstr)

功能: 重新装载类表框中的字符串

参数说明: pListCtrl: 指向列表框的指针

string: 装载的字符串指针

nstr: 装载的字符串的个数

CreateTextCtrl

定义: PTextCtrl CreateTextCtrl(U32 CtrlID, structRECT* prect, U32 FontSize, U32 style, char * KeyTable, PWnd parentWnd)

功能: 创建文本框控件, 返回指向文本控件的指针

参数说明: CtrlID: 创建的文本框控件的 ID, 此控件 ID 必须是唯一的

rect: 指向文本框控件大小和位置的指针

FontSize: 文本框的字体大小, 可以是表 1-1 数值中的一种

style: 文本框的风格, 可以是表 3-1 中的一种

KeyTable: 文本框的字符映射表, 即按键对应的在文本框中显示的字符。如果是 NJLL, 表示使用默认的字符映射表。

parentWnd: 指向控件父窗口的指针, 如果是 NJLL, 表示没有父窗口, 空间属于桌面

DestoryTextCtrl

定义: void DestoryTextCtrl(PTextCtrl pTextCtrl)

功能: 删除文本框控件

参数说明: pTextCtrl: 指向文本框的指针

SetTextCtrlText

定义: void SetTextCtrlText(PTextCtrl pTextCtrl, U16 *pch)

功能: 设置文本框的文本

参数说明: pTextCtrl: 指向文本框的指针

pch: 指向文本框显示文字的字符串指针

GetTextCtrlText

定义: U16* GetTextCtrlText(PTextCtrl pTextCtrl)

功能: 返回指向文本框文字的指针

参数说明: pTextCtrl: 指向文本框的指针

DrawTextCtrl

定义: void DrawTextCtrl(PTextCtrl pTextCtrl);

功能: 绘制指定的文本框

参数说明: pTextCtrl: 指向文本框的指针

AppendChar2TextCtrl

定义: void AppendChar2TextCtrl(PTextCtrl pTextCtrl, U16 ch, U8 IsReDraw)

功能: 在指定文本框中追加一个字符

参数说明: pTextCtrl: 指向文本框的指针

ch: 增加的字符

IsReDraw: 是否要重画。如果为 TRUE, 则重绘; 如果为 FALSE, 则不重绘

TextCtrlDeleteChar

定义: void TextCtrlDeleteChar(PTextCtrl pTextCtrl, U8 IsReDraw)

功能: 在指定文本框中删除最后一个字符

参数说明: pTextCtrl: 指向文本框的指针

IsReDraw: 是否要重画。如果为 TRUE, 则重绘; 如果为 FALSE, 则不重绘

SetTextCtrlEdit

定义: void SetTextCtrlEdit(PTextCtrl pTextCtrl, U8 bIsEdit);

功能: 设置文本框是否为编辑状态

参数说明: pTextCtrl: 指向文本框的指针

IsEdit: 指定文本框是否为编辑状态

TextCtrlOnTchScr

定义: void TextCtrlOnTchScr(PTextCtrl pListCtrl, int x, int y, U32 tchaction)

功能: 文本框的触摸屏响应函数, 当有触摸屏消息的时候, 系统自动调用

参数说明: pTextCtrl: 指向列表框的指针

x,y: 触摸屏的屏幕坐标

tchaction: 触摸屏的消息可以是表 3-2 中的一项

CreatePictureCtrl

定义: PPictureCtrl CreatePictureCtrl(U32 CtrlID, structRECT*prect, char filename[], U32 style, PWnd parentWnd)

功能: 创建图片框, 返回指向图片框的指针

参数说明: CtrlID: 创建的图片框控件的 ID, 此控件 ID 必须是唯一的

prect: 指向图片框控件大小和位置的指针

filename: 图片框中的图片文件名

style: 图片框的风格, 可以是表 3-1 中的一种

parentWnd: 指向控件父窗口的指针, 如果是 NJLL, 表示没有父窗口, 空间属于桌面

DestoryPictureCtrl

定义: void DestoryPictureCtrl(PPictureCtrl pPictureCtrl)

功能: 删除图片框控件

参数说明: pPictureCtrl: 指向图片框的指针

DrawPictureCtrl

定义: void DrawPictureCtrl(PPictureCtrl pPictureCtrl)

功能: 绘制指定的图片框

参数说明: pPictureCtrl: 指向图片框的指针

CreateButton

定义: PButtonCtrl CreateButton(U32 CtrlID, structRECT* prect, U32 FontSize, U32 style, U16 Caption[], PWnd parentWnd)

功能: 创建按钮控件, 返回指向按钮控件的指针

参数说明: CtrlID: 创建的按钮控件的 ID, 此控件 ID 必须是唯一的

prect: 指向按钮控件大小和位置的指针

FontSize: 按钮控件的字体大小

style: 按钮的风格, 可以是表 3-1 中的一种

Caption: 按钮文本

parentWnd: 指向控件父窗口的指针, 如果是 NJLL, 表示没有父窗口, 空间属于桌面

DestoryButton

定义: void DestoryButton(PButtonCtrl pButton)

功能: 删除按钮控件

参数说明: pButton: 指向按钮控件的指针

DrawButton

定义: void DrawButton(PButtonCtrl pButton);

功能: 绘制按钮控件

参数说明: pButton: 指向按钮控件的指针

ButtonOnTchScr

定义: void ButtonOnTchScr(PButtonCtrl pButtonCtrl, int x, int y, U32 tchaction)

功能: 按钮的触摸屏响应函数, 当有触摸屏消息的时候, 系统自动调用

参数说明: pButtonCtrl: 指向按钮控件的指针

x,y: 触摸屏的屏幕坐标

tchaction: 触摸屏的消息可以是表 3-2 中的一项

CreateWindow

定义: PWnd CreateWindow(U32 CtrlID, structRECT* prect, U32 FontSize, U32 style, U16 Caption[], PWnd parentWnd)

功能: 创建窗口, 返回指向窗口的指针

参数说明: CtrlID: 创建的窗口的 ID, 此窗口 ID 必须是唯一的

prect: 指向窗口大小和位置的指针

FontSize: 窗口的字体大小

style: 窗口的风格, 可以是表 3-3 中的一种

Caption: 窗口标题

parentWnd: 指向控件父窗口的指针, 如果是 NJLL, 表示没有父窗口, 空间属于桌面

表 3-3 窗口风格

	数值	说明
WND_STYLE_MODE	0x10000	有模式窗口
WND_STYLE_MODELESS	0x00000	无模式窗口
WND_STYLE_TITLE	0x20000	有窗口标题

ShowWindow

定义: void ShowWindow(PWnd pwnd, BOOLEAN isShow)

功能: 显示窗口

参数说明: pwnd: 指向窗口的指针

isShow: 是否显示窗口

DrawWindow

定义: void DrawWindow(PWnd pwnd)

功能: 绘制窗口

参数说明: pwnd: 指向窗口的指针

WndOnTchScr

定义: void WndOnTchScr(PWnd pCtrl, int x,int y, U32 tchaction)

功能: 窗口的触摸屏响应函数, 当有触摸屏消息的时候, 系统自动调用

参数说明: pCtrl: 指向窗口的指针

x,y: 触摸屏的屏幕坐标

tchaction: 触摸屏的消息可以是表 3-2 中的一项

4. 文件相关函数(与标准 C 的文件操作相同)

文件的打开(fopen 函数)

fopen 函数用来打开一个文件, 其调用的一般形式为:

文件指针名=fopen(文件名,使用文件方式);

其中,

“文件指针名”必须是被说明为 FILE 类型的指针变量;

“文件名”是被打开文件的文件名;

“使用文件方式”是指文件的类型和操作要求。

“文件名”是字符串常量或字符串数组。

使用文件的方式共有 12 种, 下面给出了它们的符号和意义。

“rt” 只读打开一个文本文件, 只允许读数据

“wt” 只写打开或建立一个文本文件, 只允许写数据

“at” 追加打开一个文本文件，并在文件末尾写数据
“rb” 只读打开一个二进制文件，只允许读数据
“wb” 只写打开或建立一个二进制文件，只允许写数据
“ab” 追加打开一个二进制文件，并在文件末尾写数据
“rt+” 读写打开一个文本文件，允许读和写
“wt+” 读写打开或建立一个文本文件，允许读写
“at+” 读写打开一个文本文件，允许读，或在文件末追加数据
“rb+” 读写打开一个二进制文件，允许读和写
“wb+” 读写打开或建立一个二进制文件，允许读和写
“ab+” 读写打开一个二进制文件，允许读，或在文件末追加数据

文件关闭函数（fclose 函数）

文件一旦使用完毕，应用关闭文件函数把文件关闭，以避免文件的数据丢失等错误。

fclose 函数调用的一般形式是：

fclose(文件指针);

读字符函数 fgetc

fgetc 函数的功能是从指定的文件中读一个字符，字符读写函数是以字符(字节)为单位的，函数调用的形式为：

字符变量=fgetc(文件指针);

写字符函数 fputc

fputc 函数的功能是把一个字符写入指定的文件中，字符读写函数是以字符(字节)为单位的，函数调用的形式为：

fputc(字符量，文件指针);

数据块读写函数 fread 和 fwrite

C 语言还提供了用于整块数据的读写函数。可用来读写一组数据，如一个数组元素，一个结构变量的值等。

读数据块函数调用的一般形式为：

fread(buffer,size,count,fp);

写数据块函数调用的一般形式为：

fwrite(buffer,size,count,fp);

其中：

buffer 是一个指针，在 fread 函数中，它表示存放输入数据的首地址。在 fwrite 函数中，它表示存放输出数据的首地址。

size 表示数据块的字节数。

count 表示要读写的数据块块数。

fp 表示文件指针。

5. 双向链表相关函数 List.h

相关结构:

```
typedef struct typeList{    //系统控件的链表
    struct typeList* pNextList;
    struct typeList* pPreList;
    void *pData;
}List,*PList
```

相关函数:

initOSList

定义: void initOSList();

功能: 初始化链表, 为链表分配动态空间

AddListNode

定义: void AddListNode(PList plist, void* pNode);

功能: 在指定的位置为链表增加一个节点

参数说明: plist: 指向链表的当前节点

pNode: 增加的节点

DeleteListNode

定义: void DeleteListNode(PList pList);

功能: 删除链表的指定节点

参数说明: plist: 指向链表的当前节点

GetLastList

定义: PList GetLastList(PList pList);

功能: 返回链表的最后一个节点

参数说明: plist: 指向链表的一个节点

6. 触摸屏相关函数 tchScr.h

TchScr_init

定义: void TchScr_init()

功能: 初始化设置触摸屏, 系统启动初始化硬件的时候调用。包括, 触摸屏的读写芯片和接口

TchScr_GetScrXY

定义: void TchScr_GetScrXY(int *x, int *y)

功能: 获得触摸屏的坐标

参数说明: x,y: 触摸屏的坐标的指针

7. 键盘相关函数 Keyboard.h

Key_init

定义: void Key_init(void)

功能: 在系统任务 SYS_Task 里被调用。

Keyboard_Read

定义: int Keyboard_Read(int ndev, BOOL isBlock)

功能: 读取键盘按键扫描码,如果没有按键, 则返回-1

参数说明: ndev,设备号, isBlock 是否阻塞

8. 液晶显示相关函数 Lcd320.h

LCD_Cls

定义: void LCD_Cls()

功能: 文本模式下清除屏幕。

LCD_Init

定义: void LCD_Init(void)

功能: 初始化 LCD, 在系统启动的时候此函数被调用。

LCD_printf

定义: void LCD_printf(char *format,...)

功能: 在 LCD 的文本模式下输出字符串, 屏幕自动滚动

参数说明: fmt: 所输出的字符串

LCD_ChangeMode

定义: void LCD_ChangeMode(U8 mode)

功能: 改变 LCD 的显示模式

参数说明: mode: 设定的 LCD 的显示模式, 可以是如表 8-1 中的一种

表 8-1 LCD 的显示模式

显示模式	数值	说明
DspTxtMode	0	文本模式
DspGraMode	1	图形模式

LCD_Refresh

定义: void LCD_Refresh()

功能: 更新 LCD 的显示, 把后台缓冲区 LCDBuffer[][]的内容更新到 LCD 的显示屏上, LCDBuffer 中每一个点用一个 32 位的整数表示。

LCDDisplayOpen

定义: void LCDDisplayOpen(U8 isOpen)

功能：打开或者关闭 LCD 显示。

参数说明：isOpen：设定打开或者关闭 LCD 的显示。可以是 TRUE 或者 FALSE。

9. 串行口相关函数 Uhal.h

Uart_Init

定义：int Uart_Init(int whichUart, int baud)

功能：初始化串行口，设置串行口通信的波特率。

参数说明：whichUart：所设定的串行口号

band：所设定的串行口通信的波特率

Uart_Printf

定义：void Uart_Printf(int whichUart, char *fmt,...)功能：输出字符串到串口 0

参数说明：whichUart：所设定的串行口号

fmt：输出到串行口的字符串

Uart_Getch

定义：char Uart_Getch(int whichUart)

功能：接收指定串口的数据。

参数说明：whichUart：所设定的串行口号

Uart_SendByte

定义：int Uart_SendByte(int whichUart,int data)

功能：向指定的串口发送一个字节的的数据

参数说明：whichUart：所设定的串行口号

data：发送的字节数据

10. 字符串相关函数 Ustring.h

Int2Unicode

定义：void Int2Unicode(int number, U16 str[]);

功能：从 int 型变量到 Unicode 字符串的转换

参数说明：number：被转换的整型数字

str：转换成的 Unicode 字符串

Unicode2Int

定义：int Unicode2Int(U16 str[]);

功能：Unicode 字符串到 int 型的转换，遇到字符串结束符'\0'或者非数字字符的时候返回，返回值是转换的结果——int 型整数

参数说明：str：被转换的 Unicode 字符串

strChar2Unicode

定义：void strChar2Unicode(U16 ch2[], const char ch1[]);

功能: char 类型 (包括 GB 编码), 到 Unicode 的编码转换。如果有 GB 编码, 则自动进行 GB 到 Unicode 的转换。

参数说明: ch1: 转换成的 Unicode 字符串

ch2: 被转换的 char 字符串

UstrCpy

定义: void UstrCpy(U16 ch1[],U16 ch2[])

功能: 字符串拷贝。

参数说明: ch1: 目标字符串

ch2: 源字符串

11. 系统图形相关函数 Figure.h

相关结构:

```
typedef struct {  
    int cx;  
    int cy;  
}structSIZE
```

```
typedef struct {  
    int x;  
    int y;  
}structPOINT
```

```
typedef struct {  
    int left;  
    int top;  
    int right;  
    int bottom;  
}structRECT
```

相关函数:

CopyRect

定义: void CopyRect(structRECT* prect1, structRECT* prect2)

功能: 复制一个矩形。

参数说明: prect1: 被复制的目标矩形的指针。

prect2: 复制的源矩形的指针。

SetRect

定义: void SetRect(structRECT* prect, int left, int top, int right, int bottom);

功能: 设置一个矩形的大小。

参数说明: prect: 指向设置矩形的指针。

left: 矩形的左边框。

top: 矩形的上边框。

right: 矩形的右边框。

bottom: 矩形的下边框。

InflateRect

定义: void InflateRect(structRECT* prect, int cx,int cy);

功能: 以矩形的中心为基准, 缩放矩形。

参数说明: prect: 指向设置矩形的指针。

cx: 扩展矩形的水平方向。正数为扩大; 负数为缩小。

cy: 扩展矩形的垂直方向。正数为扩大; 负数为缩小。

RectOffset

定义: void RectOffset(structRECT* prect, int x,int y)

功能: 移动矩形

参数说明: prect: 指向设置矩形的指针。

x: 移动矩形的水平方向相对距离。

y: 移动矩形的处置方向的相对距离。

GetRectWidth

定义: int GetRectWidth(structRECT* prect)

功能: 返回矩形的宽度

参数说明: prect: 指向设置矩形的指针。

GetRectHeight

定义: int GetRectHeight(structRECT* prect)

功能: 返回矩形的高度

参数说明: prect: 指向设置矩形的指针。

IsInRect

定义: U8 IsInRect(structRECT *prect, int x, int y);

功能: 判断指定的点是否在矩形区域之内, 如果是则返回 TRUE, 否则, 返回 FALSE。

参数说明: prect: 指向设置矩形的指针。

x,y: 指定点的 x,y 座标。

IsInRect2

定义: U8 IsInRect2(structRECT *prect, tructPOINT*ppt)

功能: 判断指定的点是否在矩形区域之内, 如果是则返回 TRUE, 否则, 返回 FALSE。

参数说明: prect: 指向设置矩形的指针。

ppt: 指向指定点的结构指针。

12. 系统启动时相关函数 LoadFile.h

LoadFont

定义: U8 LoadFont()

功能: 装载 12×12、16×16、24×24 字库。

13. 系统附加任务相关函数 OSAddTask.h

OSAddTask

定义: void OSAddTask_Init()

功能: 定义了 4 给系统任务: 触摸屏任务, 优先级为 9; 键盘扫描任务, 优先级为 58; 系统任务, 优先级为 1; LCD 刷新任务, 优先级为 59。

键盘扫描任务: 如果键按下, 则发出消息号为 OSM_KEY 的消息, Wparam 参数中存放了键盘扫描码。

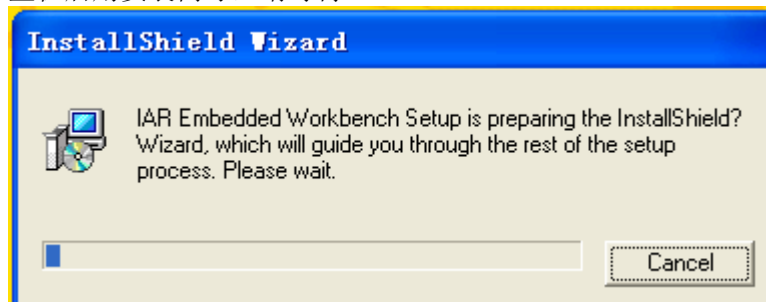
触摸屏任务: 如果有触摸动作, 则发出消息号为 OSM_TOUCH_SCREEN 的消息, Wparam 参数中低 16 位存放了触摸点的 x 坐标值, 高 16 位存放了触摸点的 y 坐标值, LParam 参数中存放了相应的触摸动作。触摸屏可以响应表 3-2 种的触摸动作。

附录三 IAR Embedded Workbench 的安装

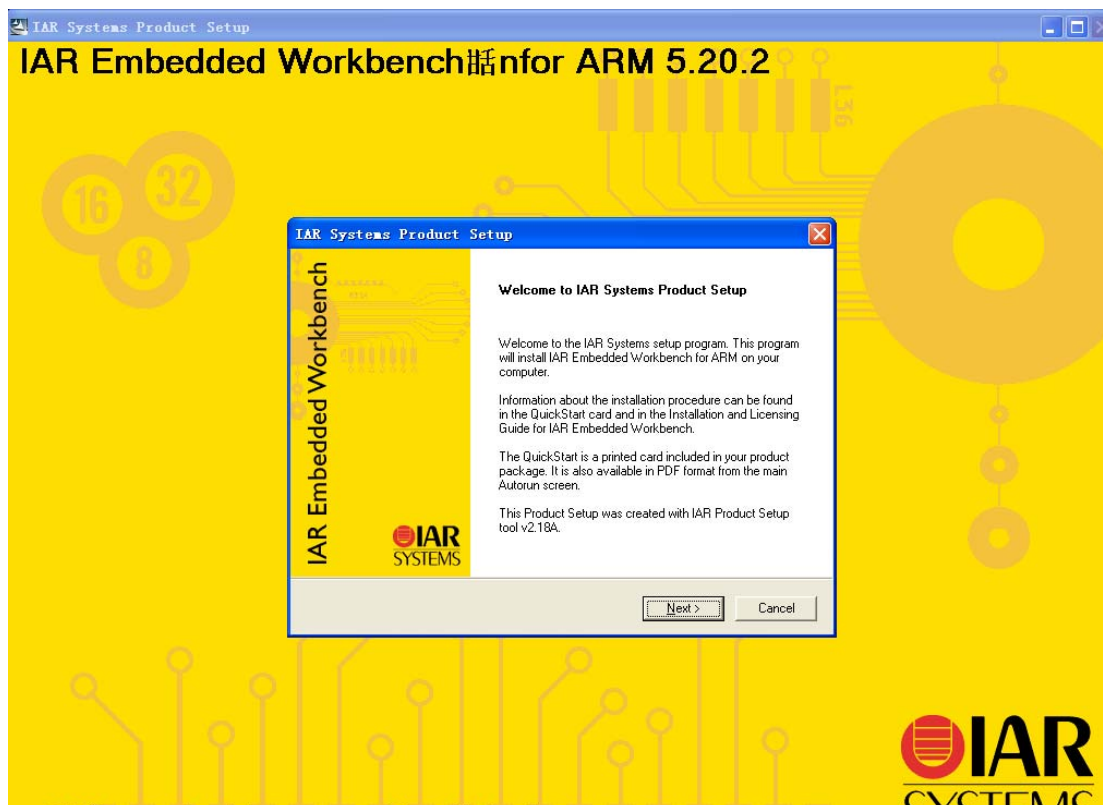
1. 双击 **autorun.exe** 出现如下界面



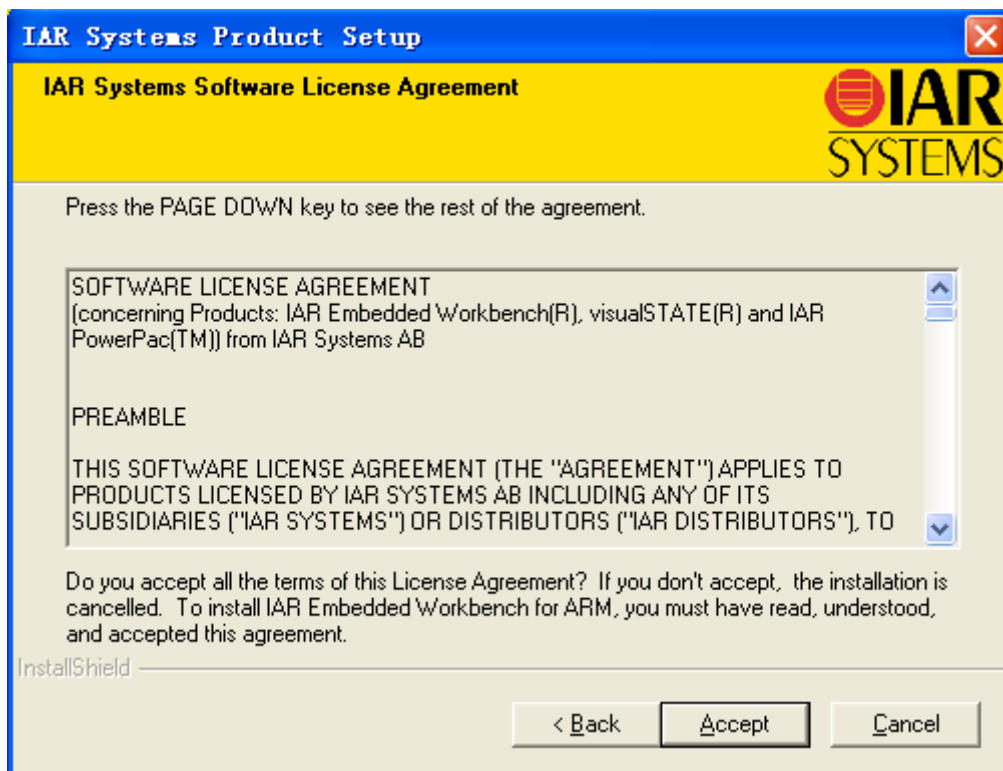
正在启用安装向导，请等待



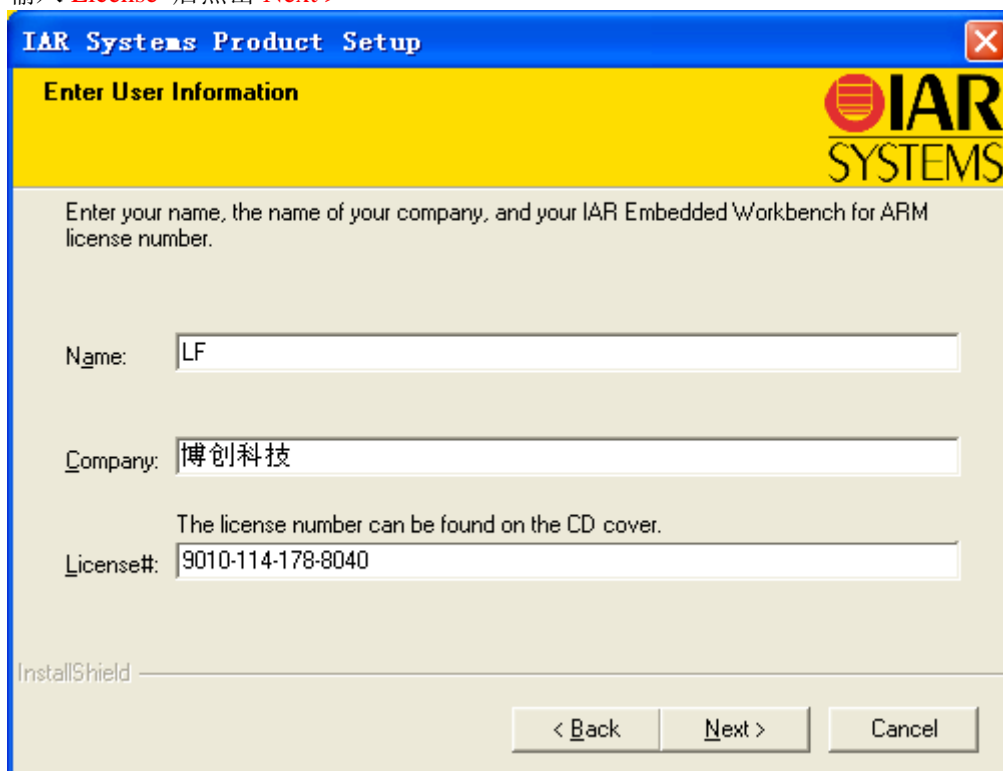
直接点击 **Next >**



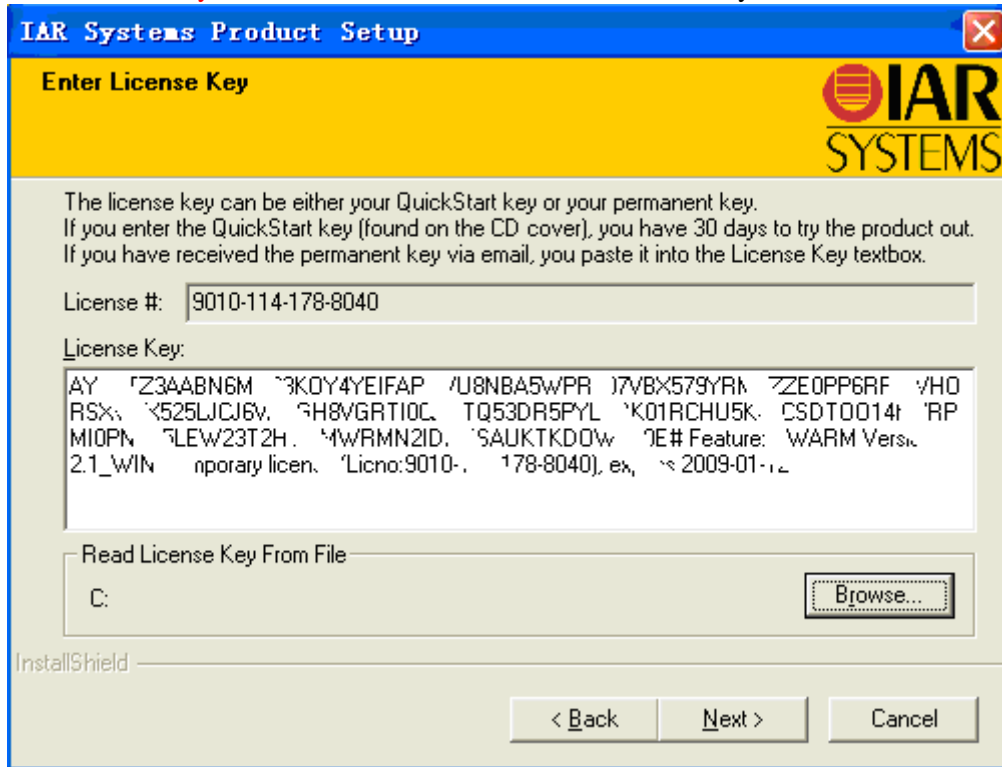
直接点击 **Accept**



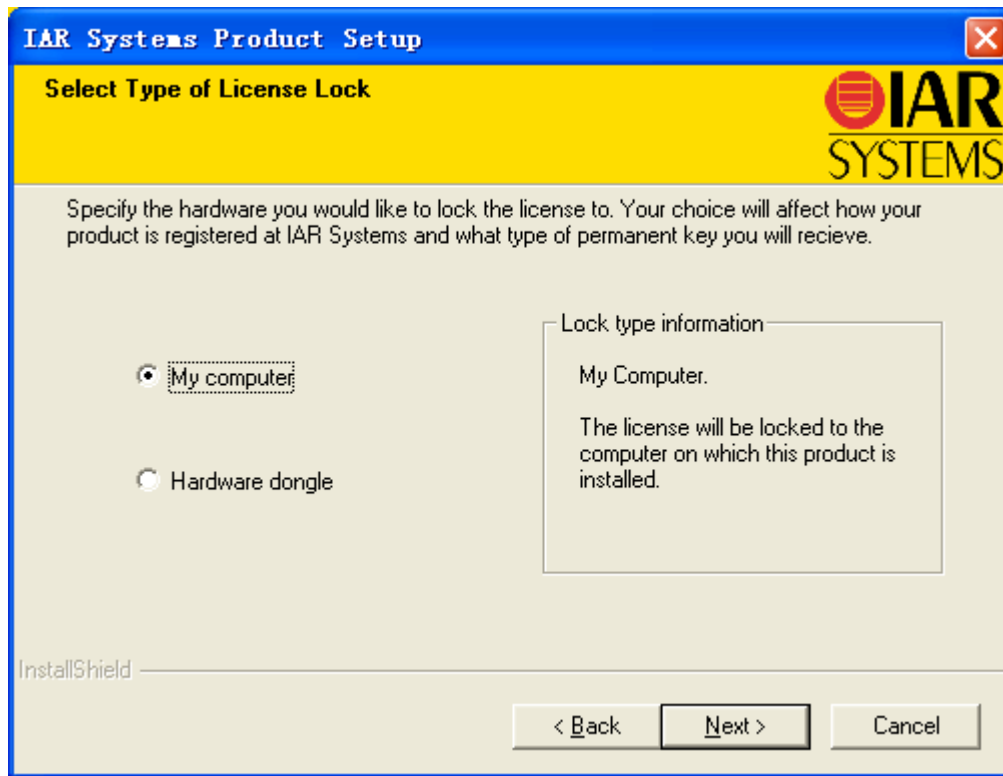
输入 License 后点击 Next >



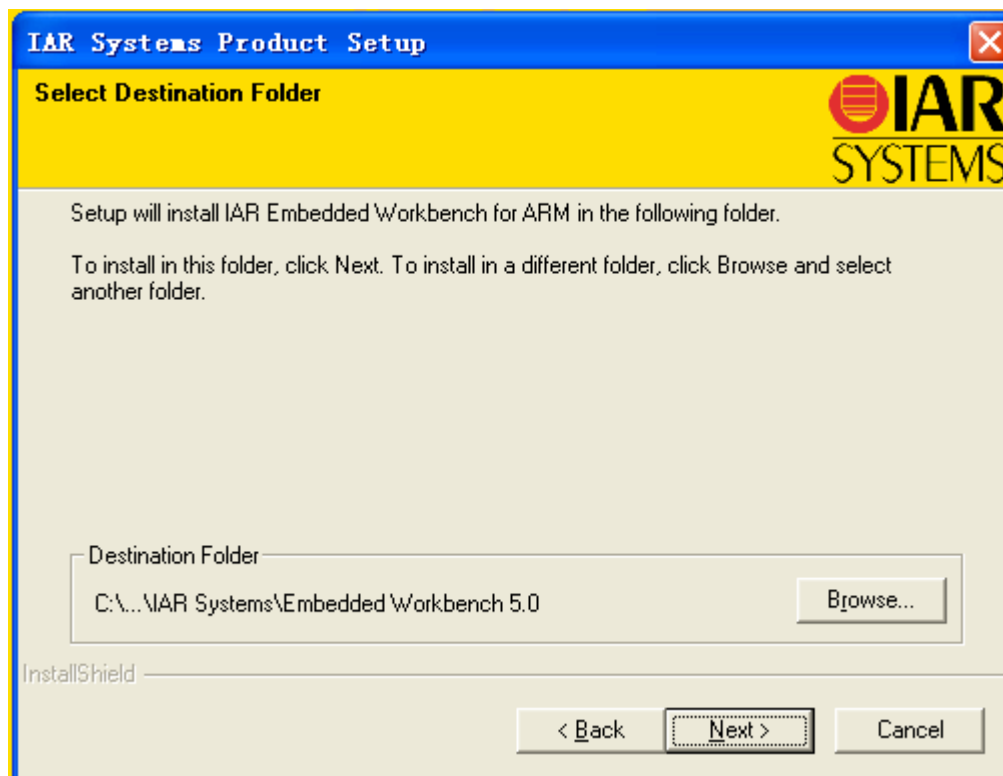
输入 **License Key:** 后点击 **Next >** (注: License 和相应的 Key 需要向 IAR 公司申请授权)



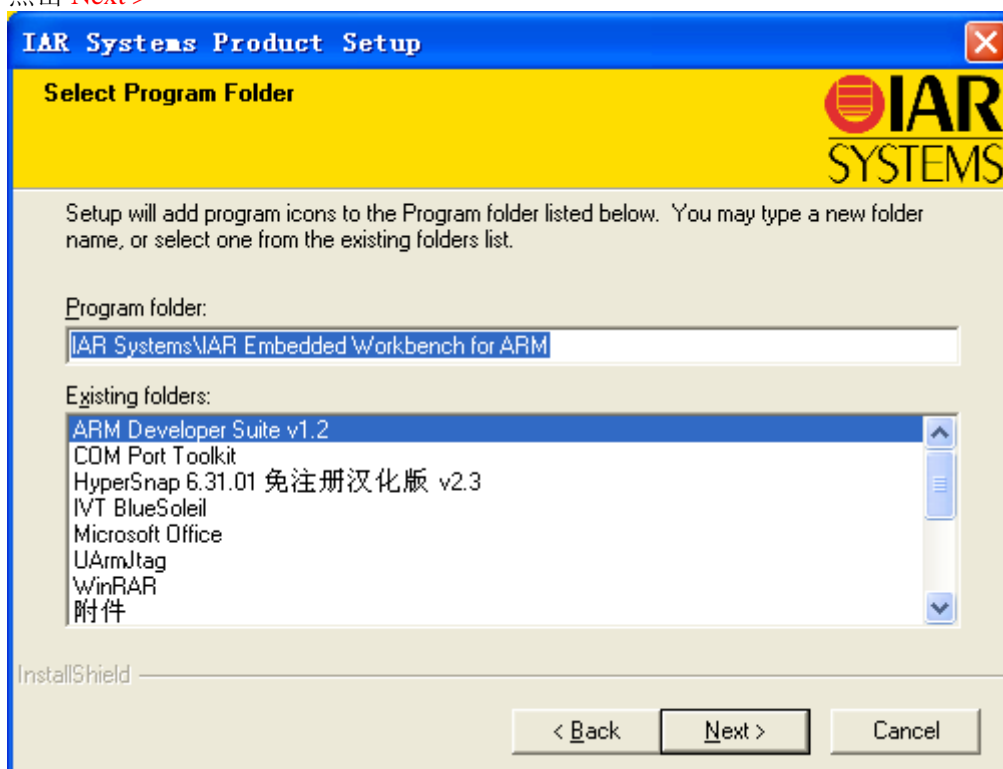
点击 **Next >**



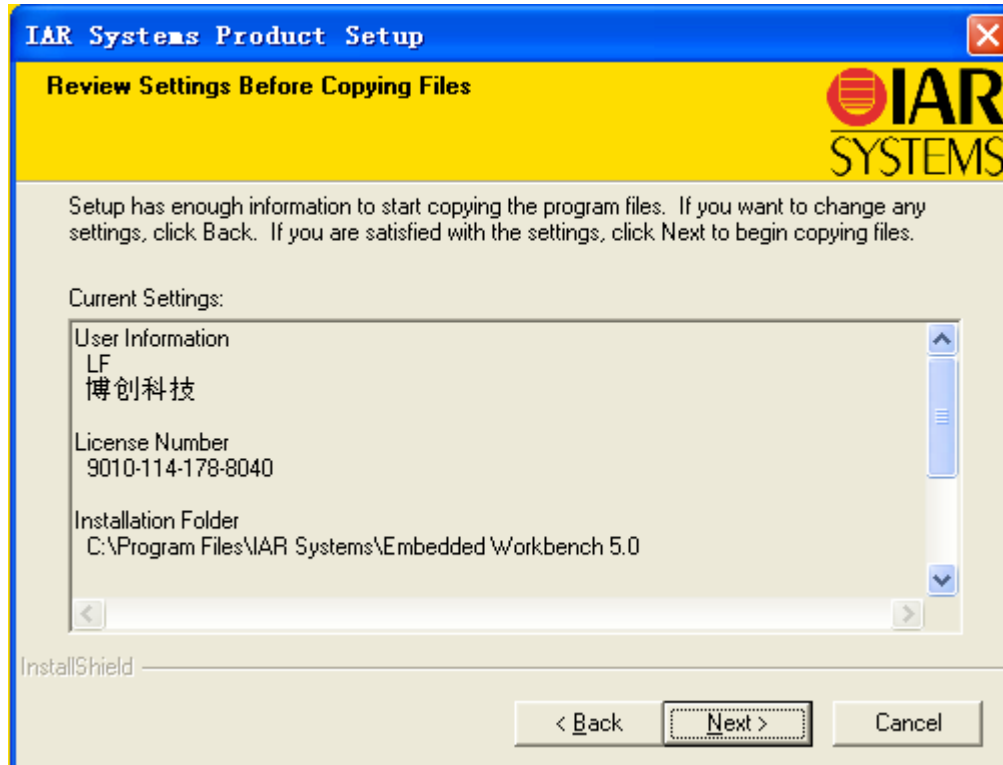
点击 **Next >**



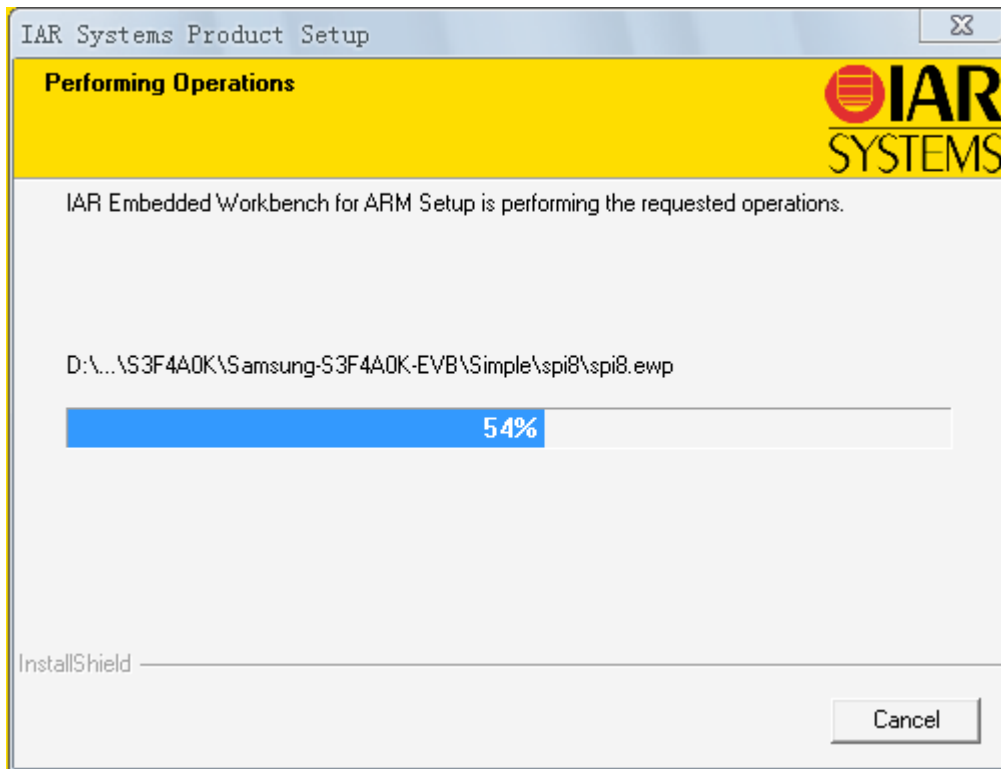
点击 **Next >**



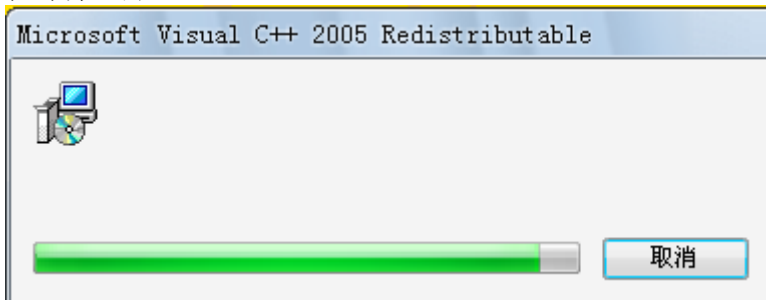
点击 **Next >**



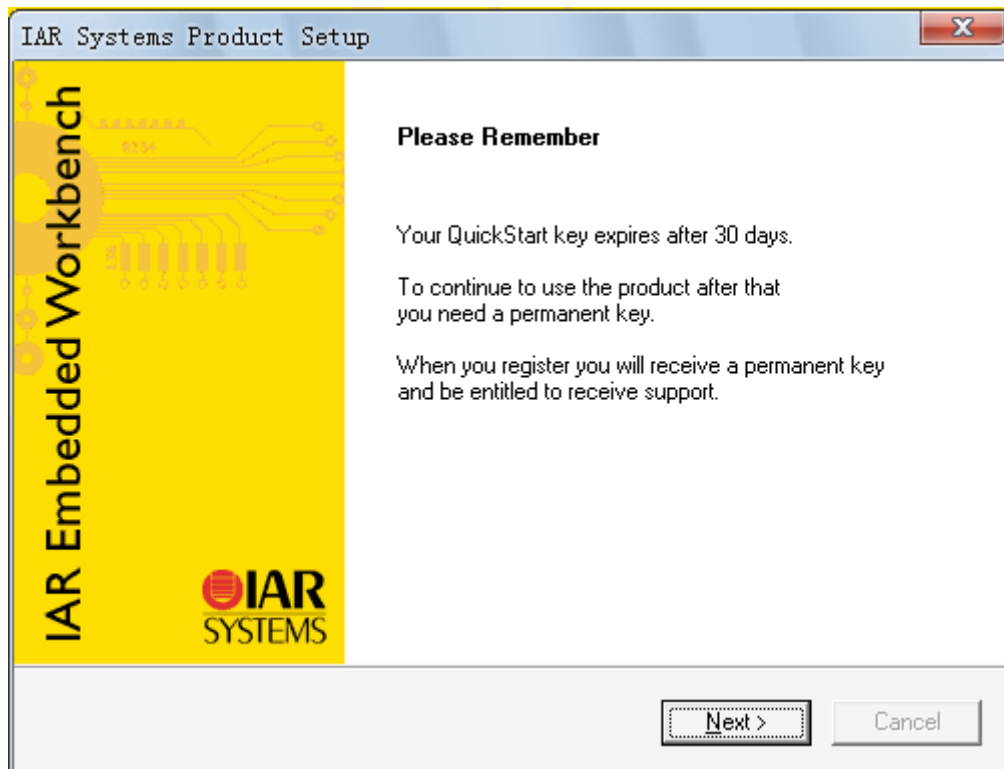
耐心等待！



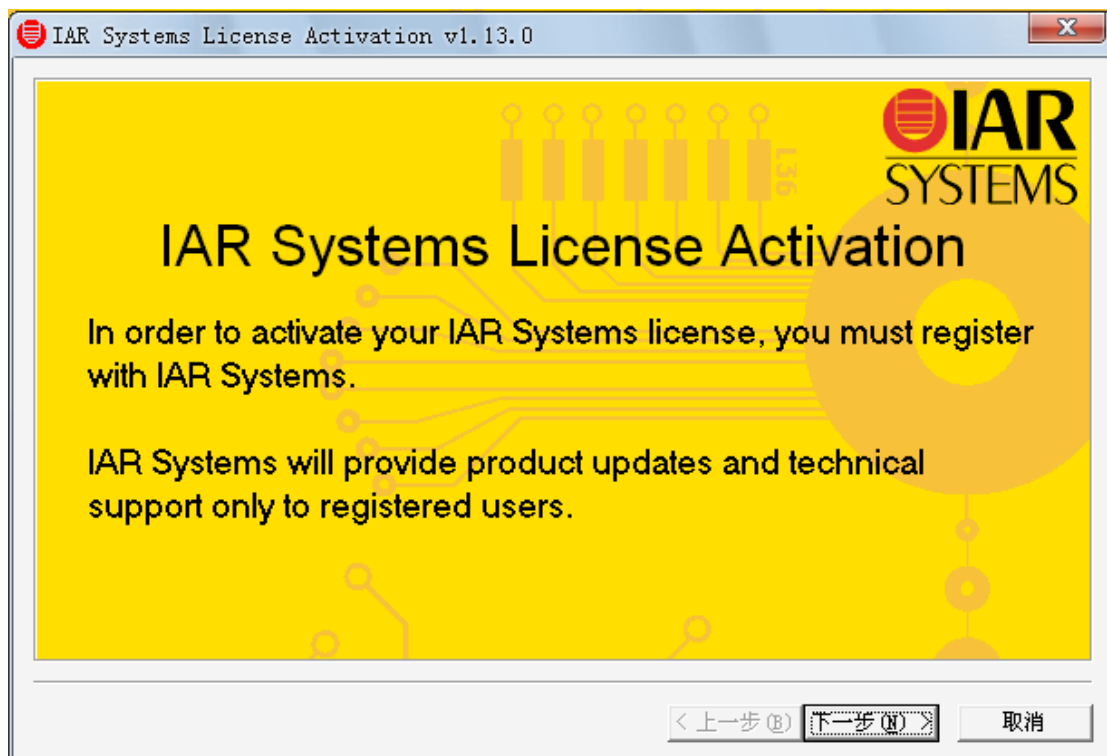
在等待一会!



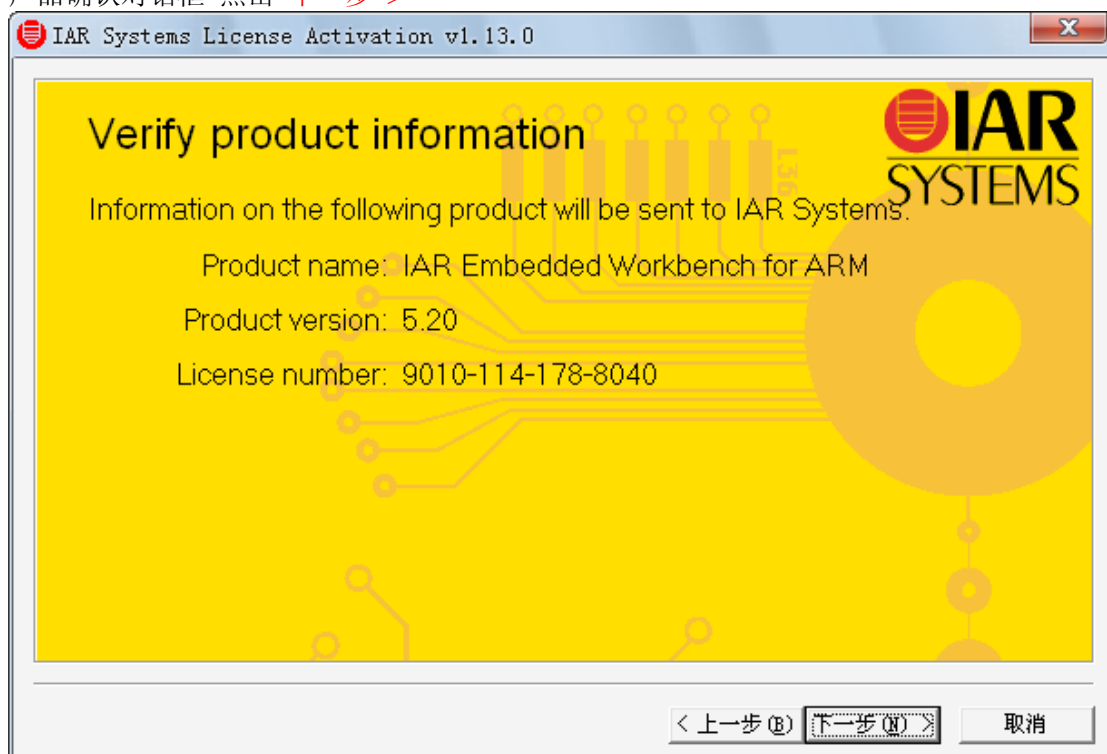
提示我的 Lcense 是 30 天试用版，如果使用的是正版就不会有时间限制，在这里直接点击
[Next >](#)



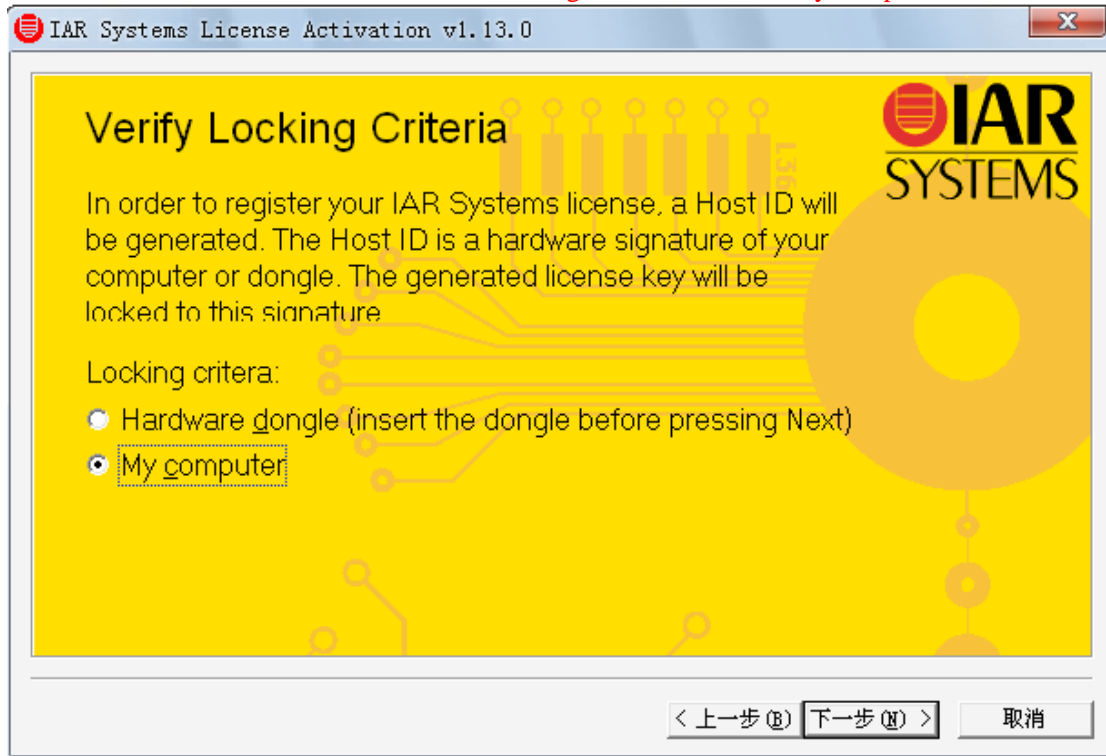
激活 Lciense 点击 下一步 >



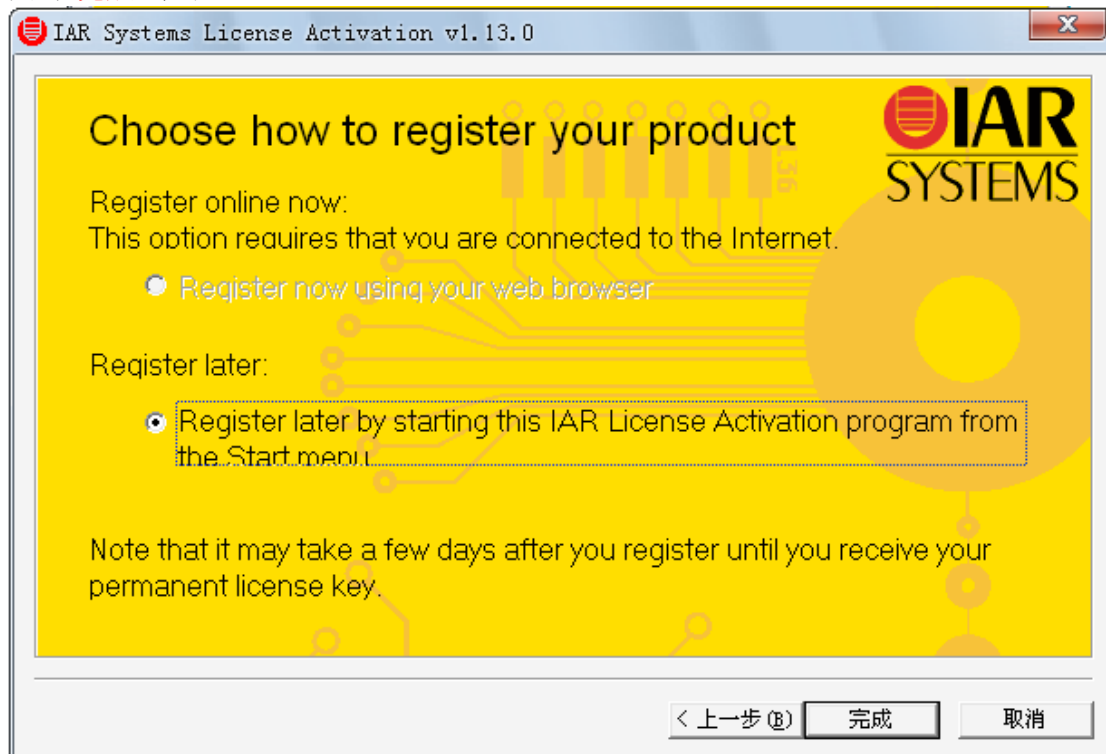
产品确认对话框 点击 下一步 >



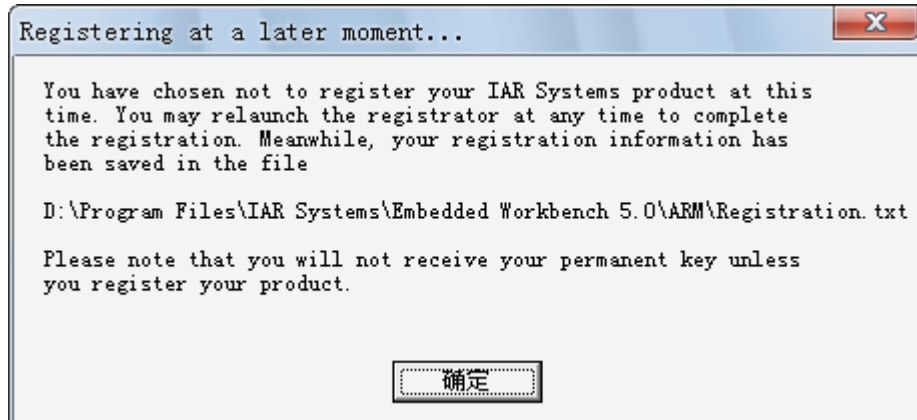
如果有硬件加密狗的话可以选择 **Hardware dongle** 一般我们选择 **My computer**



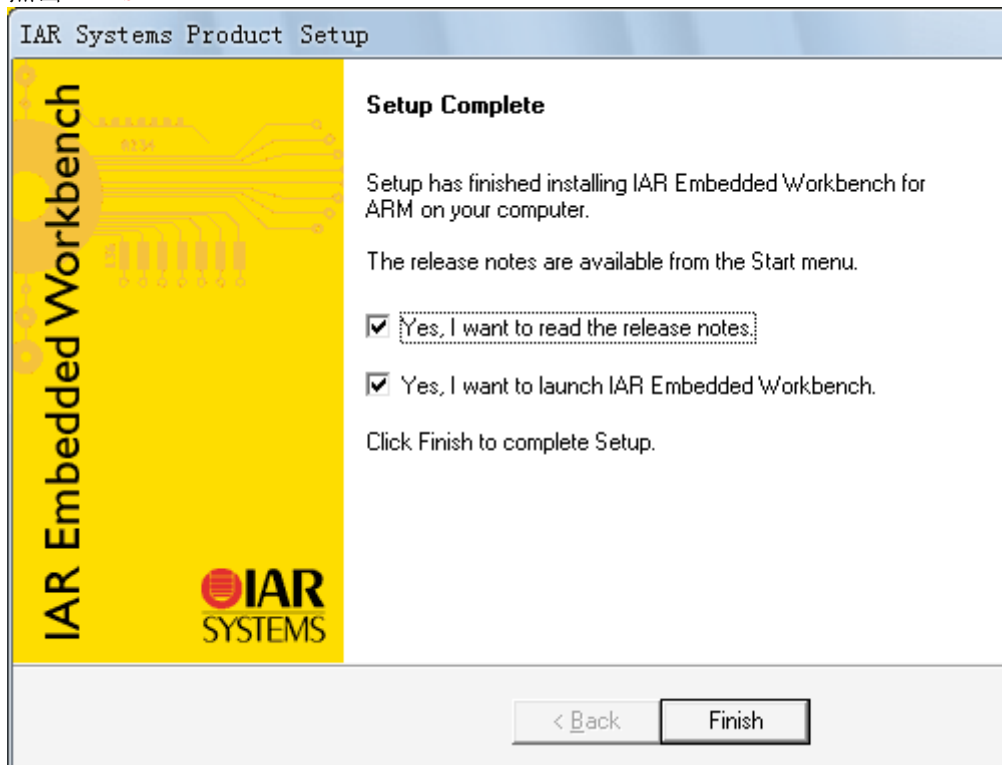
点击 **完成** 即可



点击 **确定**

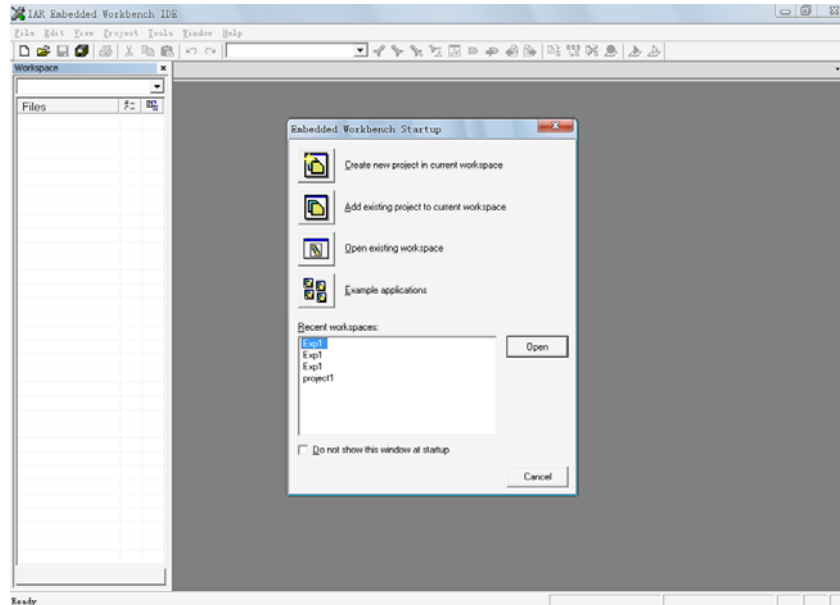


点击 **Finish**

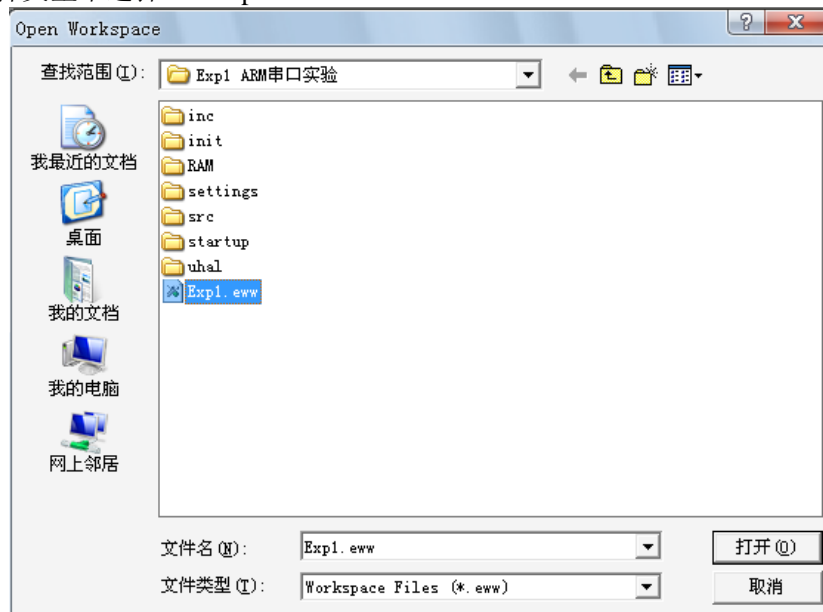


附录四 演示实验演示手册

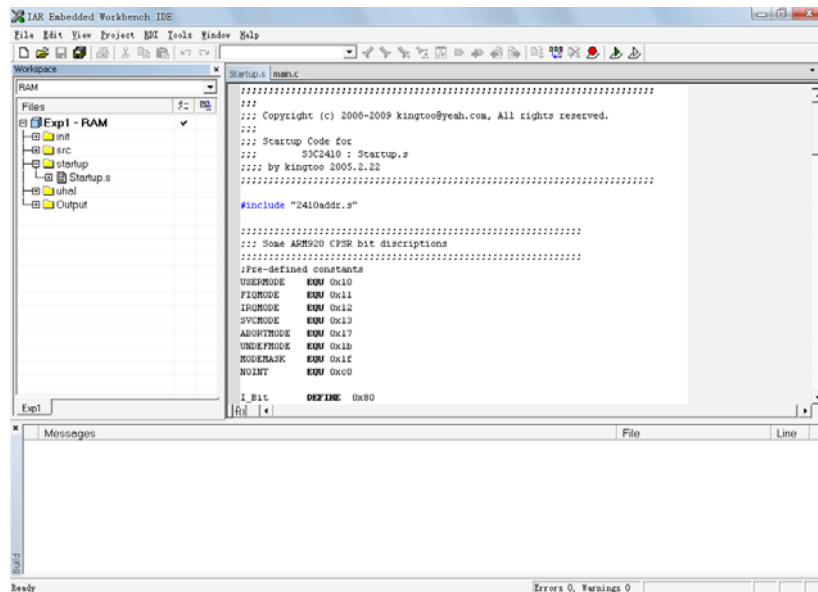
① 运行 IAR Embedded Workbench 应用程序



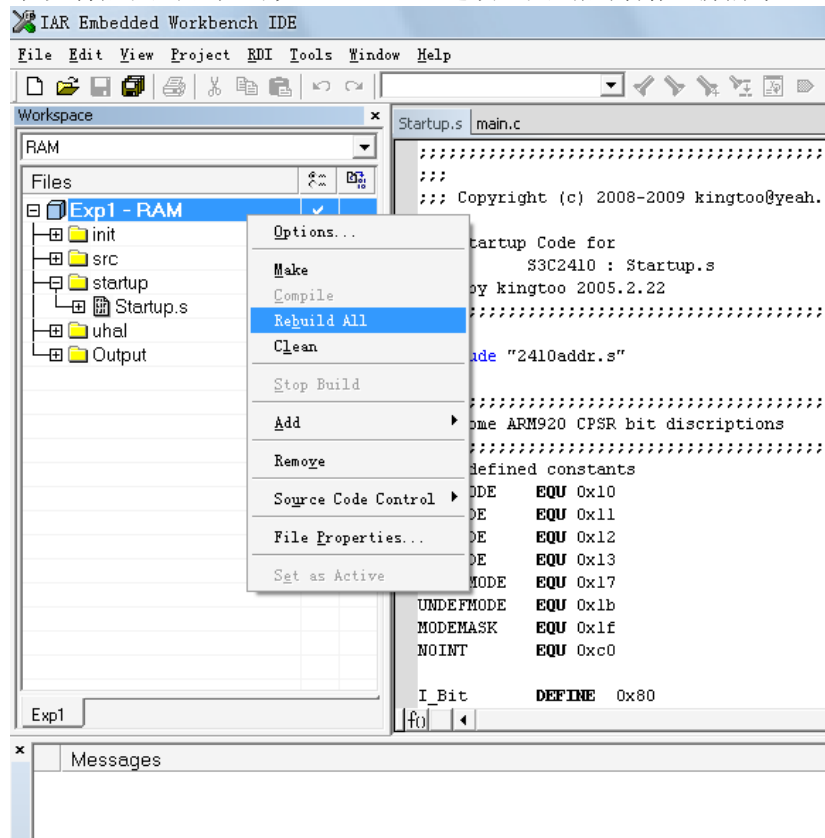
② 点击 Open existing workspace 后找到相应实验所在的文件夹，如果看不到工程文件，请在文件类型中选择 Workspace File (*.Eww)



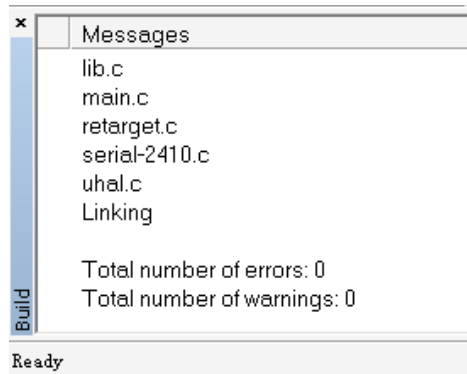
③ 点击后打开相应的工程，如下图



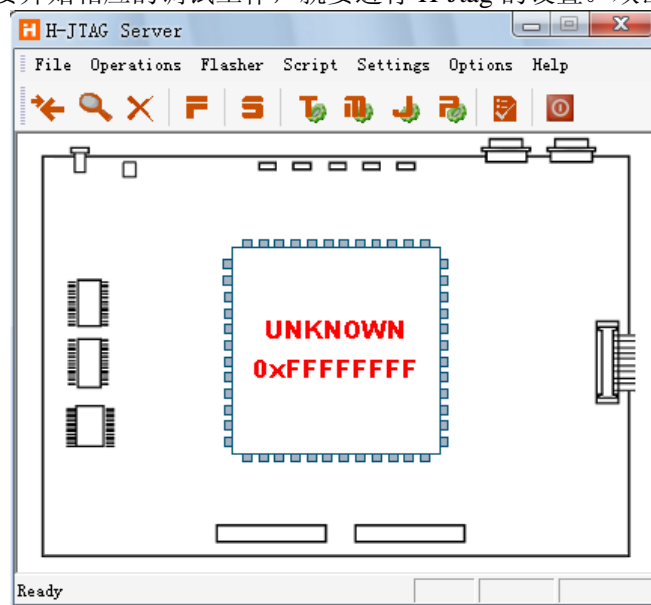
- ④ 选择工程文件后点击右键出现 Rebuild All 选项，点击后开始重新编译。



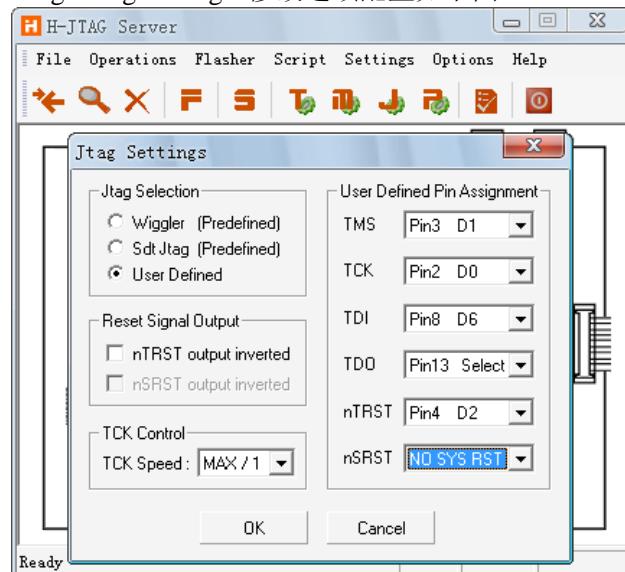
- ⑤ 重新编译完成后会有如下提示，没有警告和错误。



- ⑥ 在这之后就要开始相应的调试工作，就要进行 H-Jtag 的设置。双击运行 H-JTAG

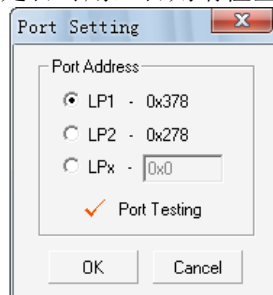


- ⑦ 选择菜单项 Settings>Jtag Settings 修改选项配置如下图：



- ⑧ 选择菜单项 Settings>Port Settings 选择 ARM-JTAG 仿真器所用的并口号，点击 Port

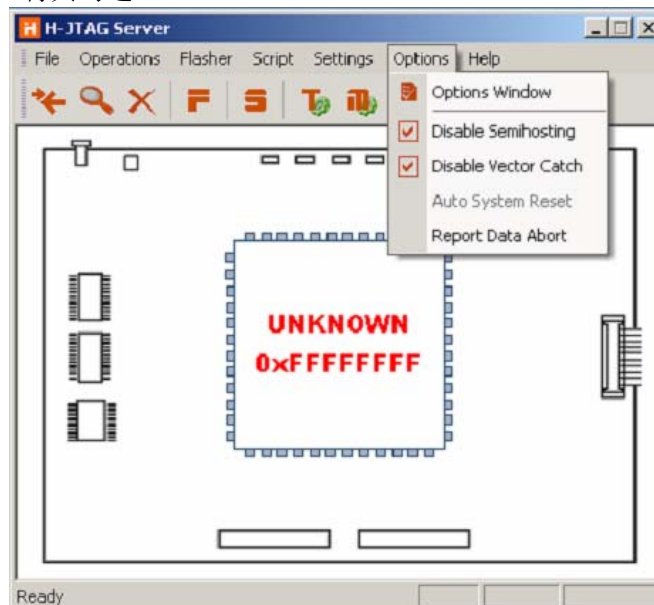
Testing 按钮，已确认该并口是否可用，否则请检查 BIOS 设置。



⑨ 正常应该弹出

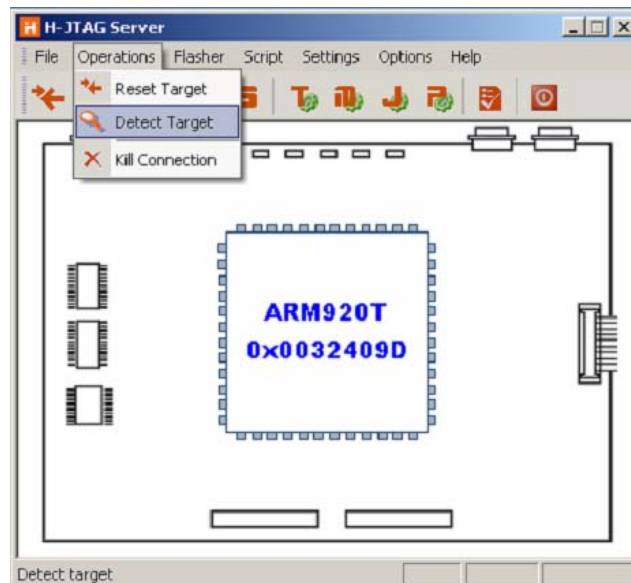


⑩ 点击 Options 菜单，检查 Disable Semihosting 和 Disable Vector Catch 是否为选中状态，若没有选中应将其勾选。

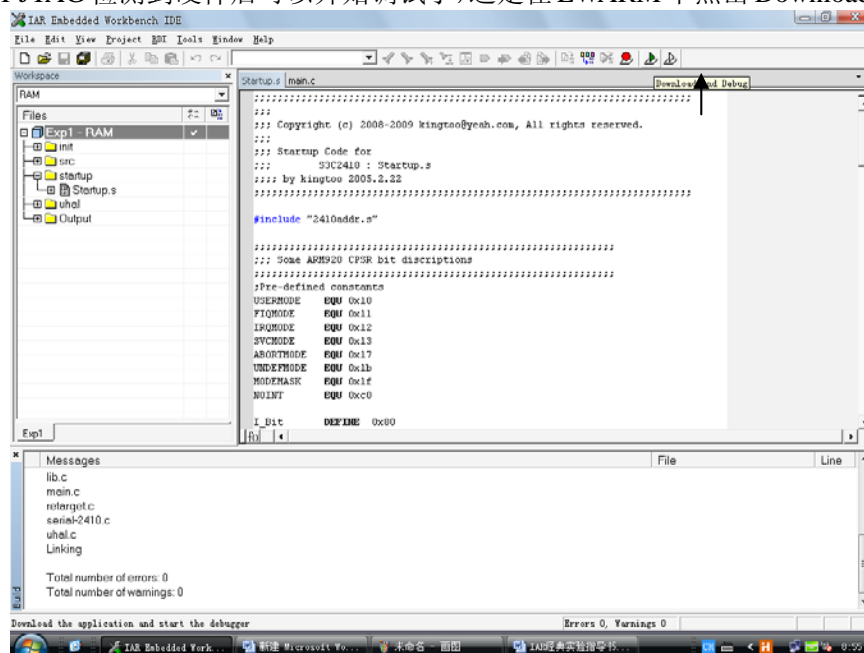


⑪ 使用 UP-LINK 接通开发板和主机，开启电源。

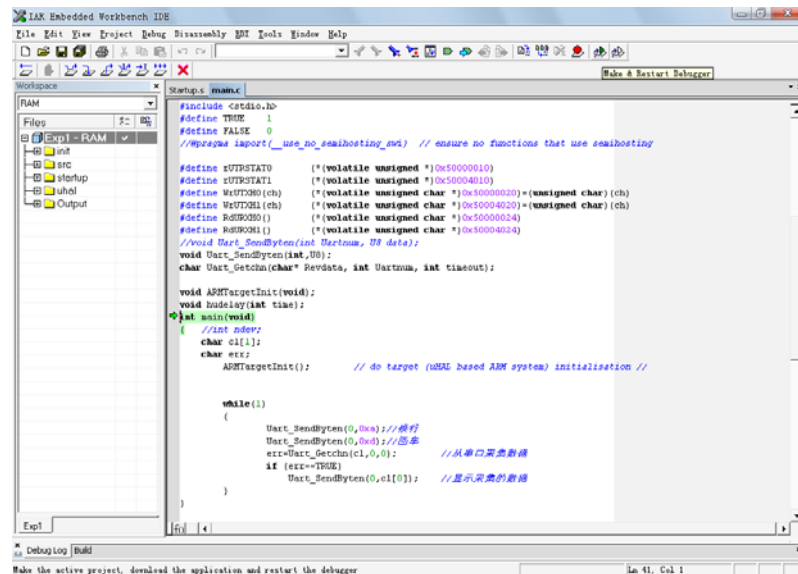
⑫ 在 H-JTAG Server 窗口中选择 Operations> Detect Target 应该能够侦测目标板上的 ARM 7~10 内核如下图，如侦测不到请检测上述步骤。



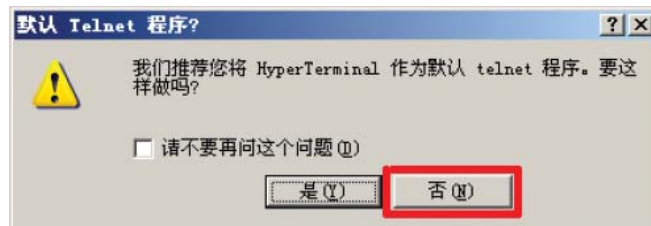
- ⑬ 在H-JTAG 检测到硬件后可以开始调试了,这是在EWARM 下点击 Download and Debug



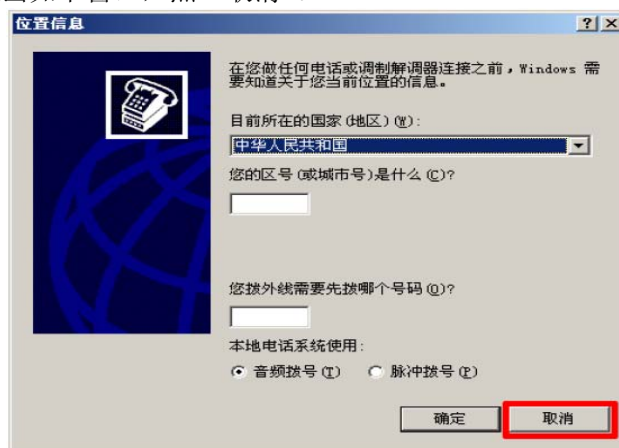
- ⑭ 点击之后出现调试界面,并且已经有运行到了 main 函数所在的位置。



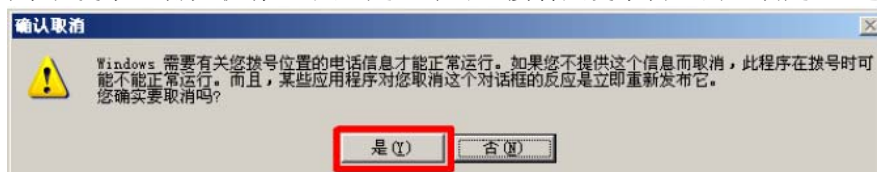
- ⑮ 因为我打开的是串口实验，所以要用到超级终端下面要介绍超级终端的设置，超级终端程序通常位于"开始->程序->附件->通讯"中，选择运行该程序，一般会跳出如图所示窗口，询问你是否要将 Hypertrm 作为默认的 telnet 程序，此时你不需要，因此点“否”按钮。

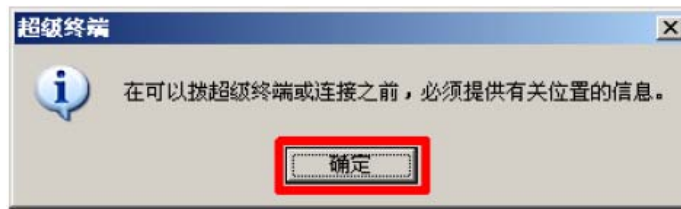


- ⑩ 接下来，会跳出如下窗口，点“取消”。



- ⑩ 此时系统提示“确认取消”，点“是”即可，接着点提示窗口的“确定”，进入下一步。

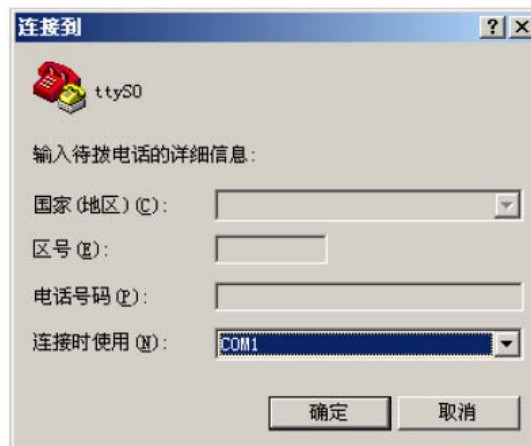




- ⑱ 超级终端会要求你为新的连接取一个名字，如图所示，这里我取了“ttyS0”，Windows 系统会禁止你取类似“COM1”这样的名字，因为这个名字被系统占用了。



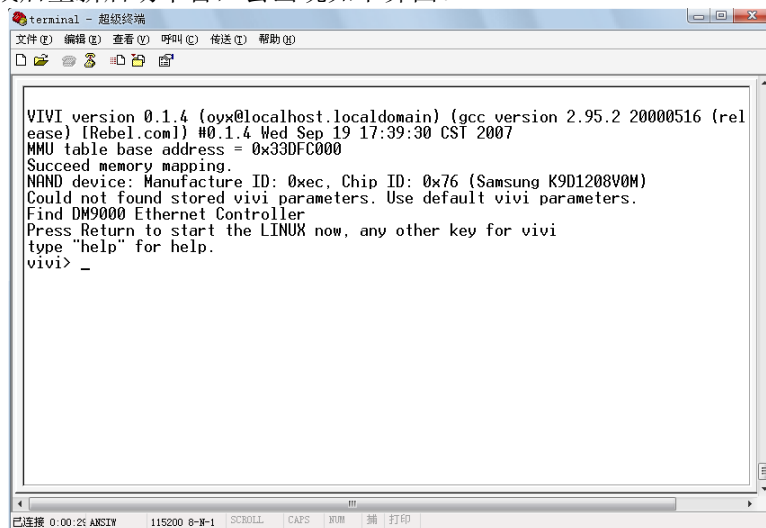
- ⑲ 当你命名完以后，又会跳出一个对话框，你需要选择连接 MINI2440 的串口，我这里选择了串口 1，如图所示：



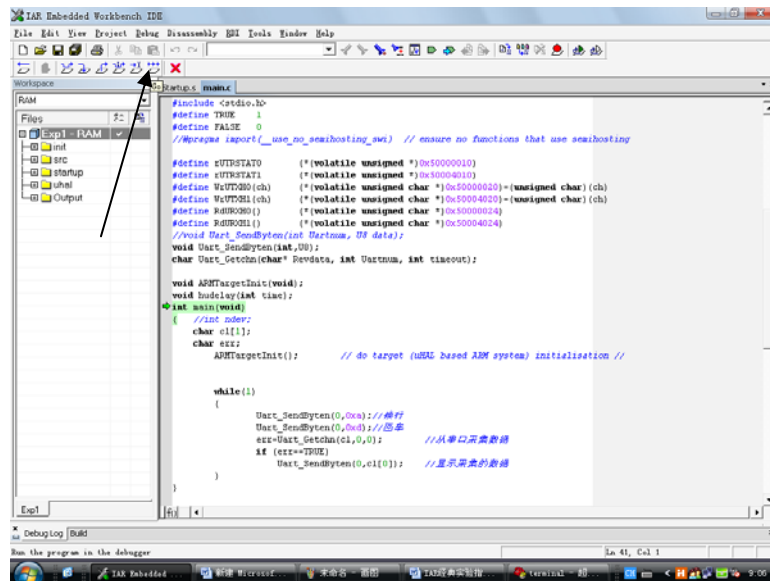
- ⑳ 最后，最重要的一步是设置串口，注意必须选择无流控制，否则，或许你只能看到输出而不能输入，另外板子工作时的串口波特率是 115200，如图所示。



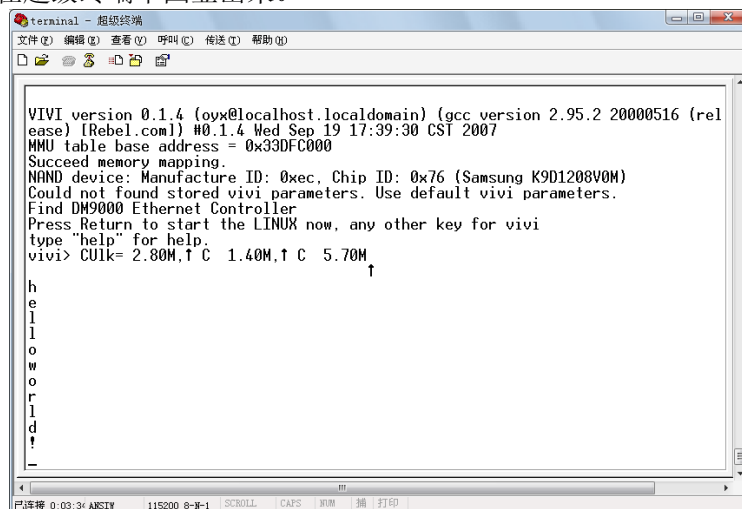
- 21 当所有的连接参数都设置好以后，打开电源开关，系统会出现 vivi 启动界面。选择超级终端“文件”菜单下的“另存为...”，保存该连接设置，以便于以后再连接时就不必重新执行以上设置了。
- 22 设置完成后重新启动平台，会出现如下界面。



- 23 现在我们在 EWARM 的调试界面下点击 Go 按钮就开始了我们的实验



- 24 接下来切换到超级终端下即可出现相应的结果，在串口实验中你输入一个字符后，ARM 会在超级终端中回显出来。



- 25 实验做完后点击 STOP DEBUGGING，退回到编译模式下关闭后退出完成本次实验。其他实验按照一样的流程进行操作即可。

特别说明：如果你做到操作系统实验时需要用到串口向 FLASH 中传输文件时，你也可以参考恢复出厂设置的章节中应用程序的烧写方法将光盘：\EWARM-2410DVP\实验程序\ucos-appendix.tar.bz2 这个压缩包按照同样的方式解压到/mnt/yaffs/目录下即可，这样操作使用网络传输方便快捷，建议使用这种方式。

