第三章．编译和连接应用程序

1．编译连接处理

选择主菜单 Project > Make，或选中工作区中的项目名 STR710-LED – Debug，按鼠标右键在弹出菜单中选择 Make。EWARM 将执行编译连接处理，生成可调试代码文件。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 *STR710-LED.d79* 和一个存储器分配（MAP）文件 *STR710-LED.map*。同时生成项目中汇编源程序和 c 源程序的 *lst* 文件。编译连接后的工作区窗口如图 2-11 所示。

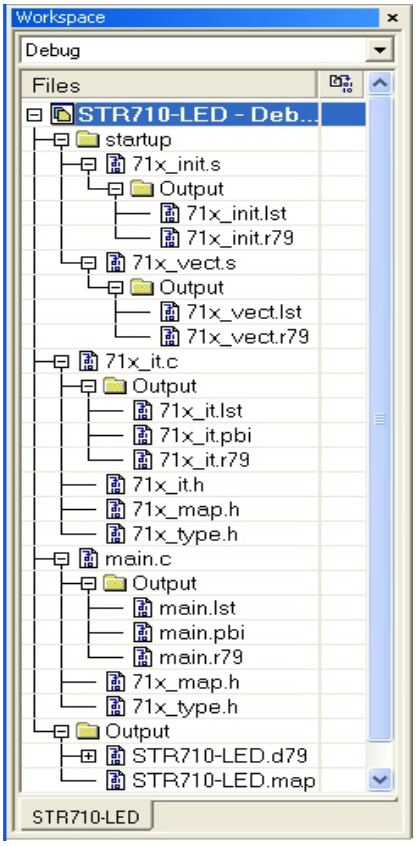

图 2-11. 编译连接处理后的项目文件结构

从编译连接后的工作区窗口中树结构中，我们可以看到每个源文件访问关联了哪些头文件，同时生成了哪些输出文件。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 目录存下放列表文件，列表文件的后缀是 *lst*；
- Obj 目录下存放 Compiler 和 Assembler 生成的目标模块文件，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。

2．查看汇编器/编译器List文件

编译连接处理中难免发现各种各样错误。出错后用户可以从 Build 消息窗口中获得错误信息，也可以从 List 文件中获得错误信息。

双击 Workspace 窗口中的*.*lst* 文件，就可以打开该 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 文件的名字和路径，编译命令行及选件等信息。
- 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。

3．查看MAP文件

双击 Workspace 中的 *STR710-LED.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件。
- CROSS REFERENCE 部分显示程序入口地址。
- RUNTIME MODEL 部分显示使用的运行时模块的属性。
- MODULE MAP 部分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 部分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 部分显示总的代码和数据字节数。

如果编译连接没有任何错误，则生成 *STR710-LED.d79* 应用程序代码，并可以用于在 IAR C-SPY 中调试。

第四章 用 C-SPY 调试应用程序

编译连接生成的 *STR710-LED.d79* 现在可以用 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．设置Debugger选件

在开始调试之前必须为 C-SPY 调试器设置选件，具体操作如下。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 J-Link/J-Trace，同时选择 Run to main（见图 3-1）。如果用户没有购买 IAR 的 JTAG 仿真器，可选择 Simulator 模拟执行。

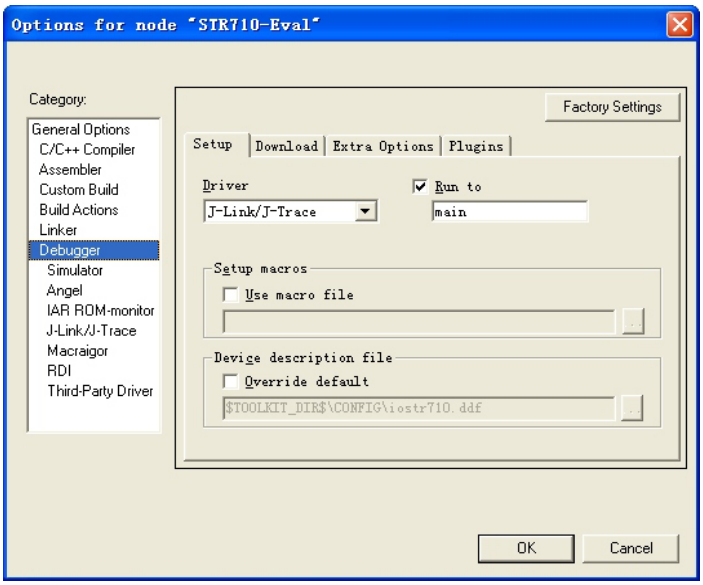

图 3-1. Debugger 选件设置

- 在 Download 页面，选择 veryfy download 和 Use flash loader，见图 3-2。

  要进行应用程序的调试，必须将生成的*STR710-LED.d79*文件下载到目标系统MCU的Flash或RAM中。调试器C-SPY是通过一个叫做Flash Loader的程序完成下载任务的。Flash Loader的详细工作原理以及它和C-SPY的互动机理我们不在这里介绍，用户可以参阅IAR的*Flash Loader Guide*。前面我们在设置General Options选件时，已经指定目标MCU是ST STR710。所以EWARM已经提供了该芯片默认的Flash Loader。如果用户选用的MCU不在EWARM的Device清单中，那就必须自己去编写该芯片的Flash Loader了。

  按 Download 页面（图 3-2）中的 Edit 按钮，在弹出的 Flash Loader Overview 对话框（图 3-3）中按 new…按钮。弹出 Flash Loader Configuration 对话框（图 3-4）。因为我们要求把应用程序下载到 STR710 的内部 Flash 中，所以选择 Relocate 并输入起始地址 0x40000000，按 OK 键确认。

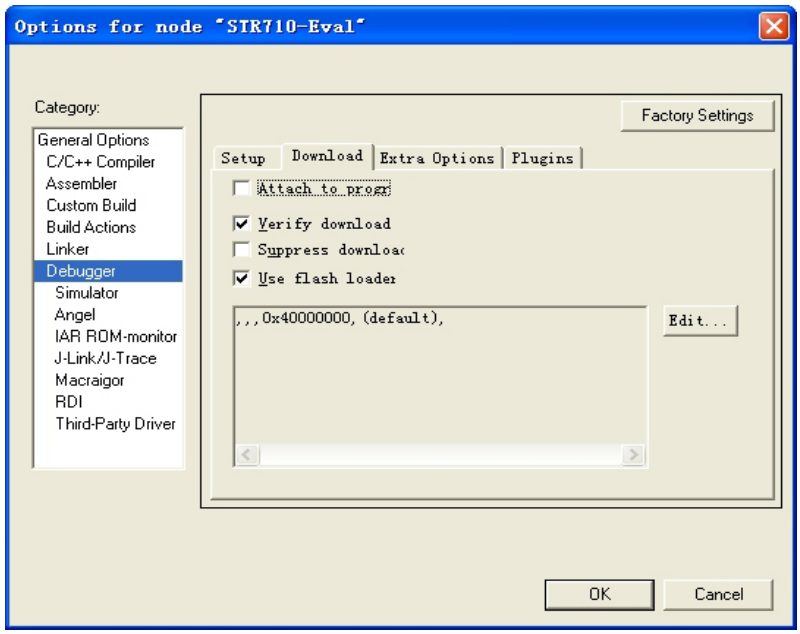

图 3-2. 设置 Downloader 参数

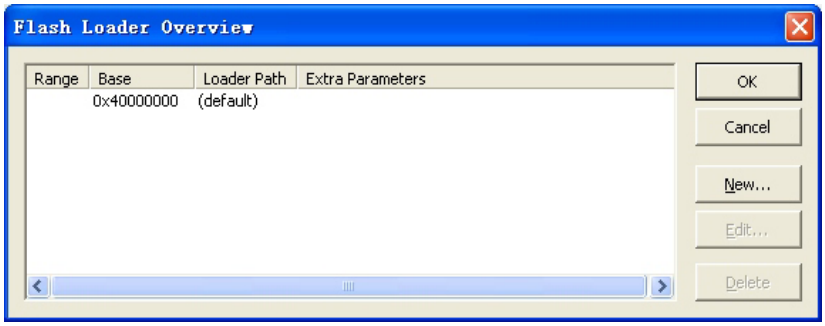

图 3-3. 设置 Flash Loader 参数

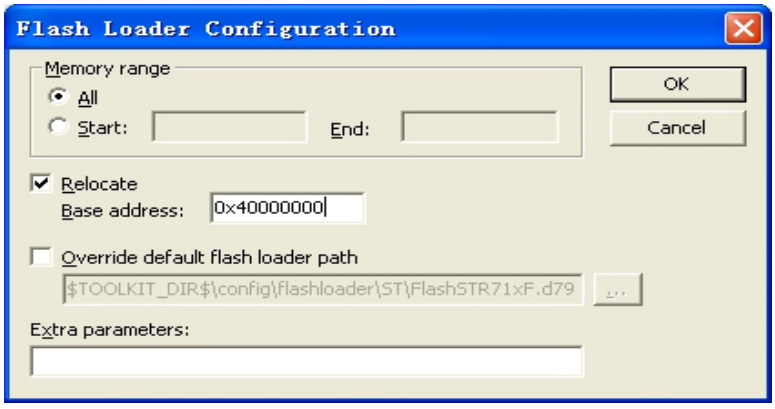

图 3-4. 设置 Flash Loader 下载起始地址

- Debugger 的其它两个设置页面 Extra Options 和 Plugins 可以采用默认值。现在按 OK 键完成设置。

2．加载应用程序

选择主菜单 Project > Debug 或工具条上的 Debugger 按钮或者按键 CTL+D，C-SPY 将开始装载 *STR710-LED.d79*。屏幕上将显示 PC 机通过 J-Link 加载的过程。屏幕上除了已经原先已经打开的窗口外，将显示一组 C-SPY 专用窗口。如 Debug Log 和 Disassembly 窗口。见图 3-5。

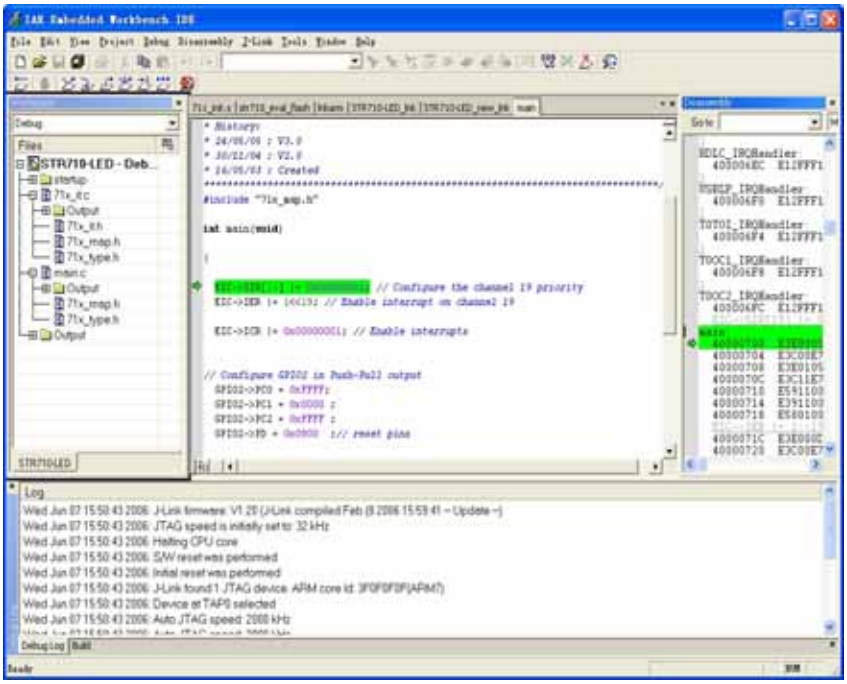

图 3-5. EWARM 的 Debug 窗口

注意：EWARM 开发环境有两种工作状态，编辑状态和调试状态。进入和退出调试状态可以分别用工具条上的 Debug 和 Stop Debugging 按钮。

3．组织窗口

S-SPY 可以固定窗口（所谓 dock），也可以将多个窗口组织成书签形式，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。双击或用鼠标拖动窗口标题栏就可以改变窗口的形式。

4．在源语句上调试

EWARM 有以下调试功能：

① 检查源语句，双击 Workspace 中的 main.c，编辑器窗口显示该文件；

② 用 Debug > Step Over 命令（或 F10）步进函数调用语句；

③ 用 Debug > Step Into 命令（或 F11）进入函数内部；当执行 Step Into 后，编辑窗口上的激活窗口将切换到被进入的函数。

注意：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。

④ 还有一些其他调试命令，如 Step Out（Shift+F11），Go（F5），Next statement，Break，Reset，Autostep 等。用户可以用这些命令灵活地执行自己的调试任务。

5．查看变量

C-SPY 允许在源程序上查看变量或表达式，并在执行程序过程中跟踪它们的值的变化。查看变量的方法有以下几种，它们的功能有一定的区别，用户可以根据自己的需要和喜好使用这些工具。至于更详细的信息请查看 *EWARM USER GUIDE* 中的 *Working with variables and expressions* 章节。

① Tooltip watch

在 Debug 状态下，把光标对准编辑窗口中变量名，在该变量的旁边将显示其数据类型和当前值。

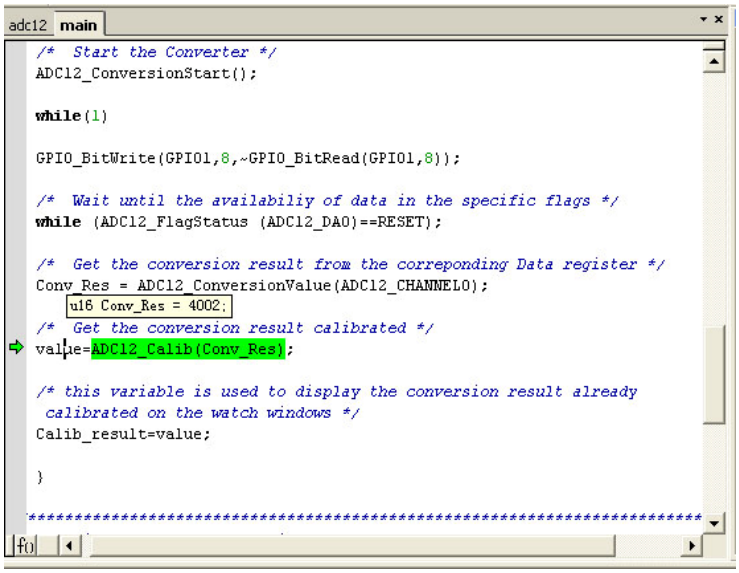

图 3-6. Tooltip 工具显示变量值

② Auto 窗口

Auto 窗口自动显示当前语句的和其周围相关的变量和表达式的值，单步执行程序时可以观察这些变量如何变化。打开 Auto 窗口的方法是双击变量名、使该变量名变成高亮度。然后选择主菜单 View → Auto 命令。图 3-7 是一个 Auto 窗口的例子。

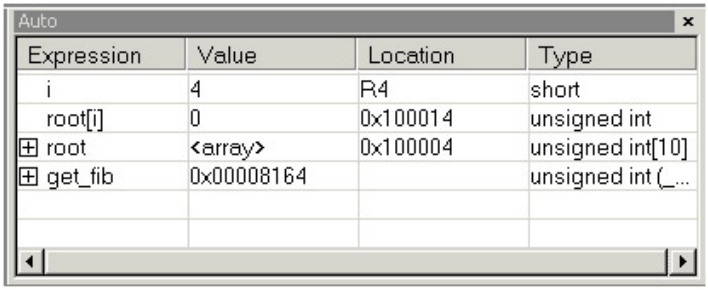

图 3-7. Auto 窗口中检查变量

③ Watch 窗口

打开 Watch 窗口的方法是选择主菜单 View → Watch 命令。把变量添加到 Watch 窗口方法是选中变量（使其变成高亮度），然后用鼠标将其拖进 Watch 窗口。或者用鼠标右键在弹出菜单中选择 Add to Watcn。请注意：Watch 窗口和 Auto 窗口可以平铺显示也可以按书签形式显示。用户可以使用鼠标变换窗口显示方式。

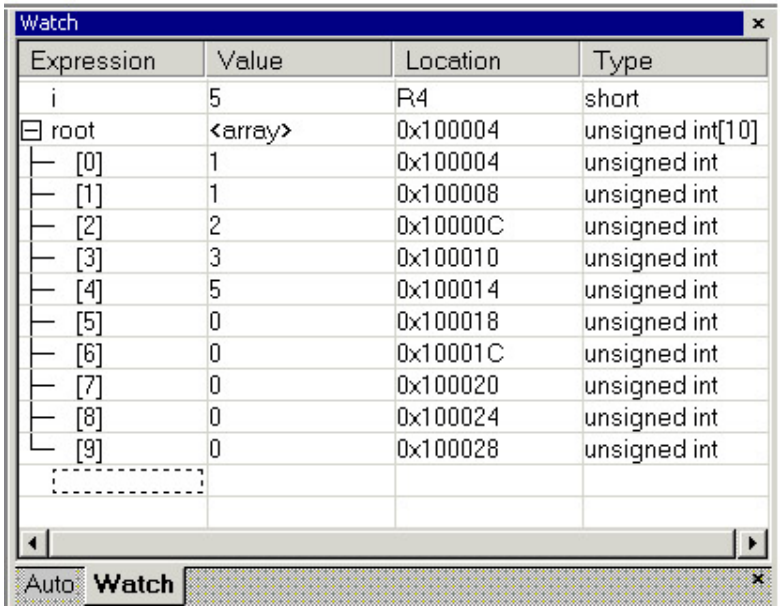

图 3-8. Watch 窗口

图 3-8 是 Watch 窗口的例子，它与 Auto 窗口以书签方式显示。Watch 窗口和 Auto 窗口是最常用的变量观察窗口，实际上用户运用这两个窗口已经可以方便地进行变量查看了。

④ Locals 窗口

打开 Locals 窗口的方法是选择主菜单 View → Locals 命令。此窗口自动显示局部变量，即当前活跃函数的自动变量和参数。

⑤ Live Watch 窗口

打开 Live Watch 窗口的方法是选择主菜单 View → Live Watch 命令。Live Watch 窗口用于观察静止位置上的变量，如全局变量。变量的值在执行时连续变化。

6．设置和监视断点

IAR C-SPY 具有强大的断点功能。详细请见 *EWARM USER GUIDE* 中的 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。

① 设置断点

点击要设置断点的语句，按鼠标右键，选择 Toggle Breakpoint（Code）。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。

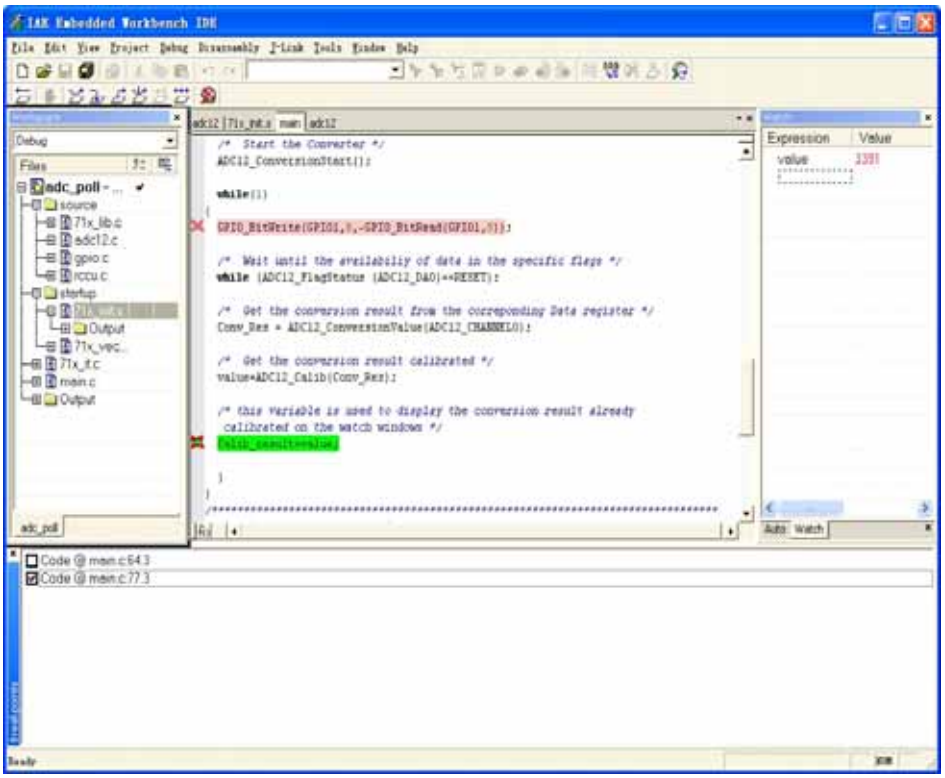

图 3-8. 设置断点和 Breakpoint 窗口

选择主菜单 View > Breakpoint 可以打开 Breakpoint 窗口观察断点设置情况。图 3-8 中的下部窗口是 Breakpoint 窗口。C-SPY 不限制断点设置的数量，但是任何时候最多只允许激活一个断点。在 Breakpoint 窗口中定义断点语句的前面小方框中打勾的是被激活的断点。用户可以在 Breakpoint 窗口中切换激活的断点，达到调试的目的。

② 执行到断点

按 F5 键或工具条上的 Go 按钮都可以让程序执行到断点。Debug Log 窗口将显示关于断点的信息。

③ 清除断点

双击 Breakpoint 窗口中带勾的方框，可以允许或禁止断点。用主菜单 Edit > Toggle Breakpoint 或按鼠标右键选择 Toggle Breakpoint，可以直接从 Breakpoint 窗口删除该断点。

7．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。

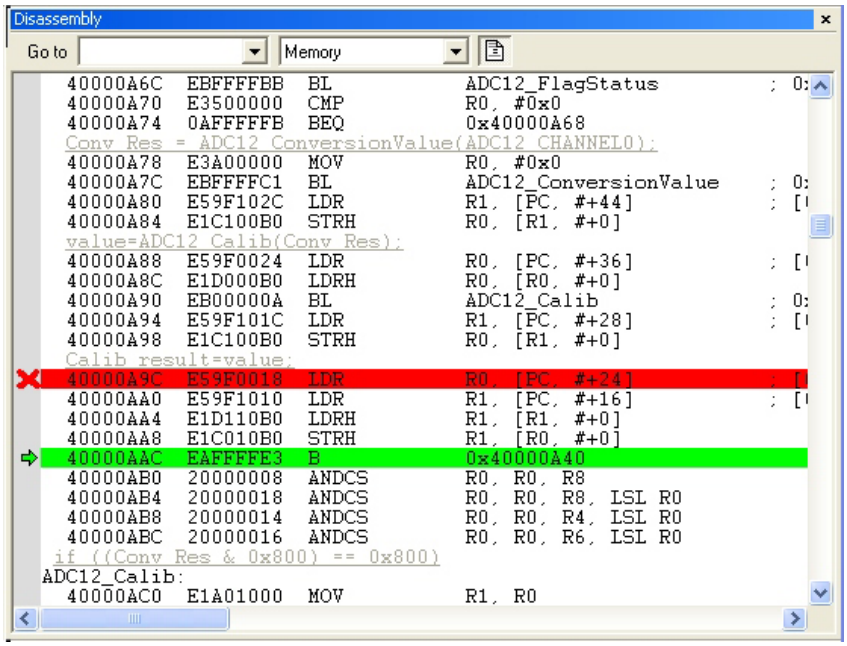

图 3-8. 反汇编窗口

③ 反汇编窗口如图图 3-8 所示，汇编代码与 C 语句一一对应。执行单步命令时将执行单条汇编语句。

④ 如果关闭反汇编窗口，单步命令重新执行单条 C 语句。

8．监视寄存器

寄存器窗口允许用户监视和修改 STR710 寄存器的内容。选择主菜单 View > Register 打开寄存器窗口，见图 3-9。

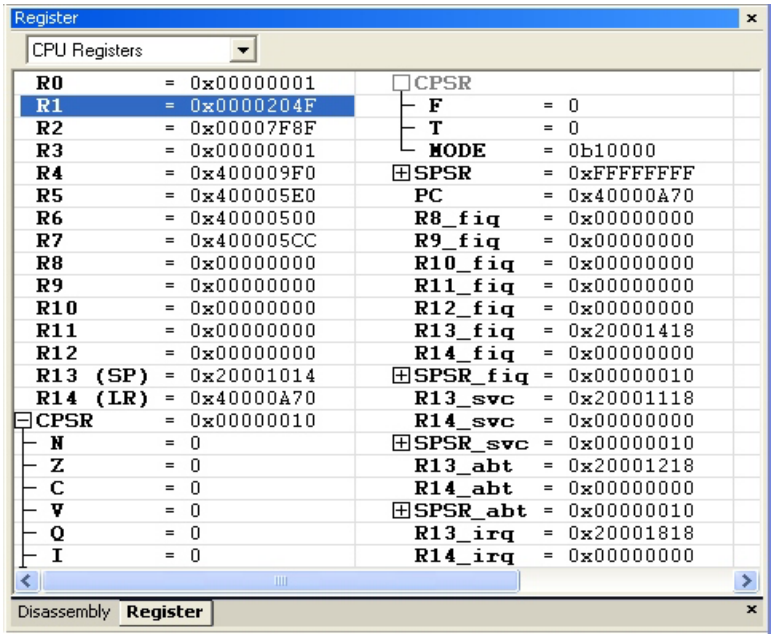

图 3-9. 寄存器窗口

图 3-9 显示的是 CPU 寄存器。C-SPY 允许检查 STR710 的所有寄存器。用户可以从寄存器窗口左上方的下拉菜单中选择需要查看的任何寄存器组。

9．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。

① 选择主菜单 View >Memory 打开存储器窗口，见图 3-10（用 8-bit 显示数据）。

② 激活 *main.c* 窗口并双击某个变量名并用鼠标将其拖到存储器窗口。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。

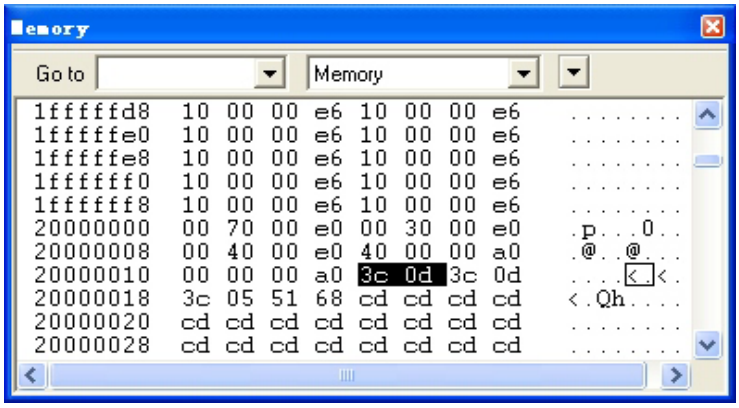

图 3-10. 存储器窗口

10．观察Terminal I/O

当用户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时，C-SPY允许用户使用Terminal I/O模拟stdin和stdout。在本例中没有用到此功能。

注意：Terminal I/O 只有在选用了 Linker 选件 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。

选择主菜单 View > Terminal I/O 显示 I/O 操作的输出，见图 3-12。Terminal I/O 窗口显示的内容取决于应用程序执行了多远。

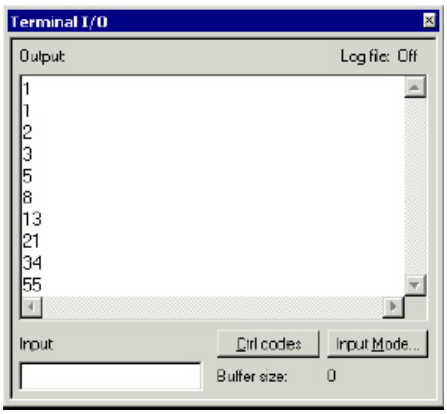

图 3-12. Terminal I/O 窗口

10．执行程序和暂停程序

① 按 F5 键，选择主菜单 Debug > Go 或工具条上的 Go 按钮都可以直接运行程序。如果没有设置断点，程序一直执行到结束。因为在本例中程序设计成无限循环，用户需要停止程序运行可以选择主菜单 Debug → Break。Debug Log 窗口中显示程序运行状态的消息。

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，在再线帮助 Help 下的 *ARM Embeded Workbench USER GUIDE* 中有详细介绍。

第五章 如何编写 ST ARM 应用程序

ARM 芯片生产商为了推销自己的产品，通常都为每一款新芯片设计和生产可以展示该芯片所有功能或主要功能的评估板，同时提供 Demo 程序的源码和所有片上外设的驱动程序。这些源程序通常都是高手写成，所以打算使用该芯片的用户用不着自己再花力气开发底层程序。ST 公司也是如此。我们在下面以 ST 的 STR710 为例，帮助初学者理解 c 语言 ARM 应用程序的组织结构，学习如何利用 ST 公司提供的底层程序开发自己的应用程序。

我们先介绍 ST 公司为 STR710 提供了哪些底层程序及每个程序的功能。然后再详细介绍其中几个重要程序。最后介绍如何利用这些资源编写自己的程序。

1．ST 公司为 STR710 提供的源程序

源 文 件	头 文 件	功 能
71_init.s		启动程序。在进入 c 语言的 main( )之前完成对芯片的初始化，存储器分配及地址重映射。
71x_vect.s		定义 Reset、异常和外设中断向量表。
71x_it.c	71x_it.h	所有异常和片上外设的中断服务程序。全是空函数，由用户根据需要添加内容。
	71x_map.h	以结构的形式描述所有片上外设的控制、状态和数据寄存器。同时定义每种片上外设基地址。
	71x_type.h	定义统一的项目数据类型。
	71x_conf.h	芯片配置文件。只需把目标系统所用外设名前面的 // 符号去掉就可定制目标系统配置。
	71x_lib.h	根据配置文件包含相应的外设头文件
71x_lib.c		给外设结构指针赋初值（基地址）
adc12.c	Adc12.h	12 位 A/D 转换器的设备驱动程序和头文件
apb.c	apb.h	APB 总线设备驱动程序和头文件
bspi.c	bspi.h	BSPI 接口设备驱动程序和头文件
can.c	can.h	CAN 总线设备驱动程序和头文件
eic.c	eic.h	EIC 中断控制器驱动程序和头文件
flash.c	flash.h	片内 FLASH 驱动程序和头文件
gpio.c	gpio.h	GPIO 驱动程序和头文件
i2c.c	i2c.h	I2C 总线驱动程序和头文件
pcu.c	pcu.h	电源管理控制程序和头文件
rccu.c	rccu.h	RCCU 控制程序和头文件
rtc.c	rtc.h	RTC 控制程序和头文件
uart.c	uart.h	UART 驱动程序和头文件
wdg.c	wdg.h	WatchDog 驱动程序和头文件
xti.c	xti.h	外部中断控制程序和头文件

表 5-1. STR710 源程序清单

2．几个主要程序的功能介绍

建立一个最小结构的 ARM 项目，至少需要以下几个程序：

- 71x_init.s
- 71x_vect.s
- 71x_it.c
- 71x_it.h
- 71x_map.h
- main.c

其中 71x_init.s 是启动程序。71x_vect.s 生成中断向量表即段 INTVEC。71x_it.c 存放中断服务程序，该文件只列出函数框架，用户根据需要填写服务程序内容。main 是主函数。我们在前面介绍 EWARM 时举的例子 STR710-LED 就是一个最简单结构的 ARM 应用例子。请注意，在 71x_it.c 文件中的定时器中断服务程序中添加了以下内容：

```
U32 counter=1;                                  // 在文件开始时定义全局变量 counter


void T1TIMI_IRQHandler(void)
{
 TIM1->SR &= ~0x2000;                           // 清除 TIM1 中断标志
 GPIO2->PD = GPIO2->PD <<1 ;                    // 轮流开通和关闭 LED 显示
 counter++;
 if (counter>4)
 {counter=1;
 GPIO2->PD = 0x0800; }
 }
```

其主程序如下：

```
#include "71x_map.h"
int main(void)
{
   EIC->SIR[19] |= 0x00000001;                  // 配置通道 19 的优先级
   EIC->IER |= 1<<19;                           // 开放通道 19 的中断
   EIC->ICR |= 0x00000001;                      // 允许中断
// 把 GPIO2 配置成 Push-Pull output 模式
   GPIO2->PC0 = 0xFFFF ;
   GPIO2->PC1 = 0x0000 ;
   GPIO2->PC2 = 0xFFFF ;
```

```
    GPIO2->PD = 0x0800   ;                          // 复位引脚
    // 配置 TIM1
    TIM1->CR2 &= ~0x2000;
    TIM1->CR2   = 0x2009;
    TIM1->CR1   = 0x8000;
    TIM1->CNTR = 0xFFFF;
    while(1);                                       // 无限循环
}
```

启动运行上面的简单程序后，STR71-EVAL 评估板上的 4 个发光二极管 LD11，LD12，LD13 和 LD14 无限制地循环点亮。LED 的轮流点亮是中断服务程序 T1TIMI_IRQHandler 中执行的。而定时器 1 的配置和启动是在主程序 main 中完成的。在大家注意，上述程序中用到结构指针 TIM1、EIC 和 GPIO，这些都是在 71x_map.h 文件中定义的。我们在后面将对其做进一步介绍。

① 启动程序的功能

什么是启动程序？它干什么用？其实在编写 8051 的 C 语言程序时就有启动程序，它完全是 C 编译器生成的，很多人只是没注意到罢了。例如，用户如果在 main( )之前定义了一个数据数组，而且还赋予每个元素以初值。那么这个赋初值的任务就是由启动程序完成的。

下面让我们深入查看本手册例子中 71x_init.s 的内容。

```
; 程序状态处理器模式域设置值，对应于 CPSR 中的 0-5 位
Mode_USR            DEFINE    0x10          ; 用户模式
Mode_FIQ            DEFINE    0x11          ; 快中断模式
Mode_IRQ            DEFINE    0x12          ; 中断模式
Mode_SVC            DEFINE    0x13          ; 管理模式
Mode_ABT            DEFINE    0x17          ; 中止模式
Mode_UNDEF          DEFINE    0x1B          ; 未定义指令模式
Mode_SYS            DEFINE    0x1F          ; 系统模式
I_Bit               DEFINE    0x80          ; I Bit 置 1 时，,禁止 IRQ
F_Bit               DEFINE    0x40          ; F Bit 置 1 时，禁止 FIQ

EIC_Base_addr       DEFINE    0xFFFFF800    ; EIC 基地址
ICR_off_addr        DEFINE    0x00          ; Interrupt Control register 偏移量
CIPR_off_addr       DEFINE    0x08          ; Current Interrupt Priority Register 偏移量
IVR_off_addr        DEFINE    0x18          ; Interrupt Vector Register 偏移量
```

```
FIR_off_addr            DEFINE     0x1C              ; Fast Interrupt Register 偏移量
IER_off_addr            DEFINE     0x20              ; Interrupt Enable Register 偏移量
IPR_off_addr            DEFINE     0x40              ; Interrupt Pending Bit Register 偏移量
SIR0_off_addr           DEFINE     0x60              ; Source Interrupt Register 0 偏移量

EMI_Base_addr           DEFINE     0x6C000000    ; EMI 基地址
BCON0_off_addr          DEFINE     0x00              ; Bank 0 configuration register 偏移量
BCON1_off_addr          DEFINE     0x04              ; Bank 1 configuration register 偏移量
BCON2_off_addr          DEFINE     0x08              ; Bank 2 configuration register 偏移量
BCON3_off_addr          DEFINE     0x0C              ; Bank 3 configuration register 偏移量

EMI_ENABLE              DEFINE     0x8000          ; 允许 EMI Bank ( Bit 15=1)
EMI_SIZE_16             DEFINE     0x0001          ; 16 位宽度设置值 (Bit 0=1)

GPIO2_Base_addr         DEFINE     0xE0005000    ; GPIO2 基地址
PC0_off_addr            DEFINE     0x00              ; Port Configuration Register 0 偏移量
PC1_off_addr            DEFINE     0x04              ; Port Configuration Register 1 偏移量
PC2_off_addr            DEFINE     0x08              ; Port Configuration Register 2 偏移量
PD_off_addr             DEFINE     0x0C              ; Port Data Register 偏移量

CPM_Base_addr           DEFINE     0xA0000040      ; CPM (MCLK Divide Control Register ) 基地址
BOOTCR_off_addr         DEFINE     0x10              ; CPM - Boot Configuration Register 偏移量
FLASH_mask              DEFINE     0x0000            ; to remap FLASH at 0x0
RAM_mask                DEFINE     0x0002            ; to remap RAM at 0x0

;|-----------------------------------------------------------------------------|
;| - APB Bridge    (System Peripheral)                                         |
;|-----------------------------------------------------------------------------|
APB1_base_addr          DEFINE     0xC0000000        ; APB Bridge1 基地址
APB2_base_addr          DEFINE     0xE0000000        ; APB Bridge2 基地址
CKDIS_off_addr          DEFINE     0x10              ; APB Bridge1 - Clock Disable Register 偏移量
SWRES_off_addr          DEFINE     0x14              ; APB Bridge1 - Software Reset Register 偏移量
CKDIS1_config_all       DEFINE     0x27FB            ; To enable/disable clock of all APB1's peripherals
SWRES1_config_all       DEFINE     0x27FB            ; To reset all APB1's peripherals
CKDIS2_config_all       DEFINE     0x7FDD            ; To enable/disable clock of all APB2's peripherals
```

```
SWRES2_config_all     DEFINE     0x7FDD              ; To reset all APB2's peripherals
;-------------------------------------------------------------------
; ?program_start
;-------------------------------------------------------------------
            MODULE ?program_start                    ; 开始一个库模块
            RSEG     IRQ_STACK:DATA(2)
            RSEG     FIQ_STACK:DATA(2)
            RSEG     UND_STACK:DATA(2)
            RSEG     ABT_STACK:DATA(2)
            RSEG     SVC_STACK:DATA(2)
            RSEG     CSTACK:DATA(2)
            RSEG     ICODE:CODE:NOROOT(2)       ; NOROOT 指定此段中如果没有代码，在连接时可抛弃
            PUBLIC   __program_start                 ; Export symbles to other module
            EXTERN  ?main                            ; Import an external symble
                    CODE32                           ; Generate 32-bit ARM instruction
;*******************************************************************************
;    定义两个宏
;*******************************************************************************
;* Macro 名          : EIC_INIT
;*功能               : 定义初始化 EIC 的宏，执行以下操作
;                     - 禁止 IRQ
;                     - 禁止 FIQ
;                     - 中断向量寄存器 IVR 包含 load PC opcode (0xF59FF00)
;                     - 当前优先级=0
;                     - 禁止所有中断通道
;                     - 所有中断优先级=0
;                     - 所有 SIR 寄存器包含到相关 IRQ 表入口的偏移量
;* Input             : 无
;* Output            : 无
;*******************************************************************************
EIC_INIT    MACRO
          LDR       r3, =EIC_Base_addr
          LDR       r4, =0xE59F0000
          STR       r4, [r3, #IVR_off_addr]          ; Write the LDR pc,[pc,#offset]
                                                     ; instruction code in IVR[31:16]
```

```
        LDR     r2, =32                         ; 32 Channel to initialize
        LDR     r0, =T0TIMI_Addr                ; Read the address of the IRQs
                                                ; address table
        LDR     r1, =0x00000FFF
        AND     r0,r0,r1
        LDR     r5, =SIR0_off_addr              ; Read SIR0 address
        SUB     r4,r0,#8                        ; subtract 8 for prefetch
        LDR     r1, =0xF7E8                     ; add the offset to the 0x00000000
                                                ; address(IVR address + 7E8 = 0x00000000)
                                                ; 0xF7E8 used to complete the
                                                ; LDR pc,[pc,#offset] opcode
        ADD     r1,r4,r1                        ; compute the jump offset
EIC_INI MOV     r4, r1, LSL #16                 ; Left shift the result
        STR     r4, [r3, r5]                    ; Store the result in SIRx register
        ADD     r1, r1, #4                      ; Next IRQ address
        ADD     r5, r5, #4                      ; Next SIR
        SUBS    r2, r2, #1                      ; Decrement the number of SIR registers
                                                ; to initialize
        BNE     EIC_INI                         ; If more then continue
        ENDM
;*******************************************************************************
;* Macro 名        : PERIPHERAL_INIT
;* 功能            : 复位所有外设.
;* Input           : 无.
;* Output          : 无
;*******************************************************************************
PERIPHERAL_INIT MACRO
        LDR     r1, =APB1_base_addr             ; r0= APB1 base address
        LDR     r2, =APB2_base_addr             ; r0= APB2 base address
        LDR     r0, =CKDIS1_config_all
        STRH    r0, [r1, #CKDIS_off_addr]       ; Clock Disabling for all APB1 peripherals
        LDR     r0, =CKDIS2_config_all
        STRH    r0, [r2, #CKDIS_off_addr        ; Clock Disabling for all APB2 peripherals
        LDR     r0, =SWRES1_config_all
        STRH    r0, [r1, #SWRES_off_addr]       ; Keep under reset all APB1 peropherals
```

```
        LDR     r0, =SWRES2_config_all
        STRH    r0, [r2, #SWRES_off_addr]       ; Keep under reset all APB2 peropherals
        MOV     r7, #10                         ; Wait that the selected macrocells exit from reset
loop1   SUBS    r7, r7, #1
        BNE     loop1
        MOV     r0, #0
        STRH    r0, [r1, #SWRES_off_addr        ]; Enable all all APB1 peropherals
        STRH    r0, [r2, #SWRES_off_addr        ]; Enable all all APB2 peropherals
        STRH    r0, [r1, #CKDIS_off_addr]       ; Clock Enabling for all APB1 peripherals
        STRH    r0, [r2, #CKDIS_off_addr]       ; Clock Enabling for all APB2 peripherals
        MOV     r7, #10                          ; Wait that the selected macrocells exit from reset
loop2   SUBS    r7, r7, #1
        BNE     loop2
        ENDM
;*************************************************************************************************
; 重映射请求
; Reset 后，STR710 的两个引脚 BOOT0 和 BOOT1 的状态（外部设定）被锁存进 BOOT 配置
; 寄存器 PCU_BOOTCR 中（0xA0000050）。引导方式通常由此两引脚决定，但是在启动程序
; 中允许在 Reset 后由软件修改硬件设置。如果用户希望在启动程序中规定引导方式，可以删除
; 下面的注释符号。这样 BOOT 引脚上的设置将不影响引导方式。
;               #define     remapping
; 然后确定将哪块存储器映射到地址 0x00000000
; 如果希望重映射到 RAM，删除下一行的注释
;             #define    remap_ram
;   如果希望重映射到 FLASH，则删除下一行的注释
;             #define remap_flash

        IMPORT   T0TIMI_Addr                    ; 定义外部符号 T0TIMI_Addr
__program_start
        LDR     pc, =NextInst
NextInst
        NOP                                     ; Wait for OSC stabilization
        NOP
        NOP
        NOP
```

```
            NOP
            NOP
            NOP
            NOP
            NOP
;*********************************************************************************************************
; ARM 处理器支持 7 种运行模式，其中 6 种是特权模式，一种是用户模式。除用户模式和系
; 统模式外其他都是异常模式。每种处理器模式使用不同的寄存器组，并有自己独立的物理
; 寄存器 R13(SP)。所以在初始化时要将 SP 指向其运行模式的栈空间。堆栈的大小和地址
; 是在连接命令文件中定义的。
;*********************************************************************************************************
            MSR     CPSR_c, #Mode_ABT|F_Bit|I_Bit
            ldr     sp,=SFE(ABT_STACK) & 0xFFFFFFF8     ; 指向 ABT_STACK 栈底

            MSR     CPSR_c, #Mode_UNDEF|F_Bit|I_Bit
            ldr     sp,=SFE(UND_STACK) & 0xFFFFFFF8     ; 指向 UND_STACK 栈底

            MSR     CPSR_c, #Mode_SVC|F_Bit|I_Bit
            ldr     sp,=SFE(SVC_STACK) & 0xFFFFFFF8     ; 指向 SVC_STACK 栈底

; 如果要求复位所有外设，删除下一行注释
;           PERIPHERAL_INIT                             ; 调用宏 PERIPHERAL_INIT

; 如果要求初始化 EIC，删除下一行注释
            EIC_INIT                                    ; 调用宏 EIC_INIT

;*********************************************************************************
;REMAPPING   根据前面的重映射请求决定是否做重映射处理
;Description   :   Remapping   memory whether RAM,FLASH
;                  at Address 0x0 after the application has started executing.
;                  Remapping is generally done to allow RAM   to replace FLASH
;                  at 0x0.
;                  the remapping of RAM allow copying of vector table into RAM
; 下面的程序修改 BOOTCR[1:0]的内容。
;*********************************************************************************
```

```
#ifdef remapping
    #ifdef remap_flash
        MOV     r0, #FLASH_mask
    #endif
    #ifdef remap_ram
        MOV     r0, #RAM_mask
    #endif
        LDR     r1, =CPM_Base_addr
        LDRH    r2, [r1, #BOOTCR_off_addr]          ; 读 BOOTCR 的低 2 位
        BIC     r2, r2, #0x03                       ; 清除 BOOTCR 的低 2 位
        ORR     r2, r2, r0                          ; 根据 R0 修改 BOOTCR 的低 2 位
        STRH    r2, [r1, #BOOTCR_off_addr]          ; 写回 BOOTCR 寄存器
#endif
        MSR     CPSR_c, #Mode_FIQ|I_Bit             ; Change to FIQ mode
        ldr     sp,=SFE(FIQ_STACK) & 0xFFFFFFF8     ; 指向 FIQ_STACK 栈底

        MSR     CPSR_c, #Mode_IRQ|I_Bit; Change to IRQ mode
        ldr     sp,=SFE(IRQ_STACK) & 0xFFFFFFF8     ; 指向 IRQ_STACK 栈底

        MSR     CPSR_c, #Mode_USR                   ; Change to User mode, Enable IRQ and FIQ
        ldr     sp,=SFE(CSTACK) & 0xFFFFFFF8        ; 指向 CSTACK(user) 栈底

; 现在跳转到 C 库函数?main
        b ?main                                     ; 注意只能用指令 B 不能用 BL，因为不再返回。
        LTORG
        END
```

从上面的 STR710-LED 的启动程序清单中我们可以看到，启动程序执行以下任务：

- 定义了 CSPR 寄存器中处理器模式控制字段的值和中断控制字段值；
- 定义了 EIC、EMI、GPIO、APB1 和 APB2 的基地址和相关控制寄存器的偏移量；
- 定义了各种处理器模式的堆栈；
- 定义了初始化 EIC 和外部设备的 2 个宏；
- 决定是否重映射；
- 加载各处理器模式的堆栈指针

- 跳转到 C 的库程序?main

上面 STR710-LED 的启动程序在很多情况下是可以通用的，几乎不做任何修改就可以用于其他项目。当然有时候也需要做一定的修改才能使用。建议初学者先直接引用这个启动程序，等到熟悉以后再根据项目需要做修改。

② 71x_vect.s

71x_vect.s 建立 Reset、异常中断处理程序以及外设中断的向量表，初学者可以直接引用这个启动程序。

③ 71x_it.c 和 71x_it.h

71x_it.c 中包含所有中断服务程序的空函数，用户可以选择加进自己中断服务程序。就象前面介绍的。

④ 71x_cofg.h

71x_cofg.h 是目标系统配置文件，它是用户开发自己的项目时唯一要修改的文件。用户应根据目标板上使用到的芯片资源定制自己的项目配置文件，方法是删除使用设备前面的注释。注意，在 STR710-LED 例子中没有利用 71x_cofg.h。请看下面例子：

```
#ifndef __71x_CONF_H
#define __71x_CONF_H
/* Comment the line below to put the library in release mode */
#define DEBUG
#define inline inline

/*  系统不用的设备名前加注释  */
#define _ADC12                          /*  起用 ADC12 */

/* #define _APB */
/* #define _APB1 */
/* #define _APB2 */
/* #define _BSPI */
/* #define _BSPI0 */
/* #define _BSPI1 */
/* #define _CAN */
/* #define _EIC */
/* #define _EMI */
/* #define _FLASH */
```

```
#define _GPIO                          /* 起用 GPIO 0 和 1 */
#define _GPIO0
#define _GPIO1
/* #define _GPIO2 */
/* #define _I2C */
/* #define _I2C0 */
/* #define _I2C1 */

#define _PCU                           /* 起用 PCU */
#define _RCCU                          /* 起用 RCCU */

/* #define _RTC */
/* #define _TIM */
/* #define _TIM0 */
/* #define _TIM1 */
/* #define _TIM2 */
/* #define _TIM3 */

/* #define _UART */
/* #define _UART0 */
/* #define _UART1 */
/* #define _UART2 */
/* #define _UART3 */
/* #define _USB */
/* #define _WDG */
/* #define _XTI */
/* #define _IRQVectors */

#endif /* __71x_CONF_H */
```

在这个例子中我们为目标系统配置了 ADC12、GPIO0、GPIO1、 PCU 和 RCCU 这几个设备。

⑤ 71x_map.h

在 71x_map.h 文件中以结构的形式定义片上外部设备的控制、状态和数据寄存器。我们以 12 位 A/D 为例：

- 定义 AD 转换器寄存器组的结构如下：

```
typedef volatile struct
  {
      vu16 DATA0;
      vu16 EMPTY1[3];          // EMPTY 是寄存器之间的空地址单元（6 个字节）
      vu16 DATA1;
      vu16 EMPTY2[3];
      vu16 DATA2;
      vu16 EMPTY3[3];
      vu16 DATA3;
      vu16 EMPTY4[3];
      vu16 CSR;
      vu16 EMPTY5[7];
      vu16 CPR;
  } ADC12_TypeDef;
```

- 定义 A/D 转换器寄存器组的基地址

因为 A/D 转换器连接在 APB2 总线上，先定义 APB2 的基地址

```
#define APB2_BASE          0xE0000000
```

然后根据偏移量定义 A/D 转换器寄存器组的基地址

```
#define ADC12_BASE   (APB2_BASE + 0x7000)
```

⑥ 71x_lib.c

71x_lib.c 根据用户的配置文件（71x_conf.h）自动初始化目标系统起用的外设指针。仍以 A/D 转换器为例，见下面：

```
#ifdef _ADC12
    ADC12 = (ADC12_TypeDef *) ADC12_BASE
#endif
```

此后用户在应用程序中需要引用 A/D 转换器的某个寄存器时只需用以下方法即可：

```
ADC12->CSR  = 0x00 ;
```

⑦ 71x_lib.h

71x_lib.h 根据用户的配置文件（71x_conf.h）包含需要访问的头文件，还是以 ADC12 举例来说明。如果我们在 71x_conf.h 中配置了 AD 转换，即删除了#define _ADC12 前的注释。那么在 71x_lib.h 中有：

```
#ifdef _ADC12
    #include “adc12.h”
#endif
```

注意： ST 的所有源文件最初是为 RealView 开发环境编写的。在 RealView 中其输出代码是否带调试信息是根据在 71x_cofg.h 中是否保留语句 **#define DEBUG** 决定的。在 71x_map.h 中有 **#ifdef DEBUG** 的定义。而在 IAR EWARM 开发环境中，输出 DEBUG 还是 RELEASE 版本，是在 EWARM 开发环境中选择的。用户应注意在利用 ST 的源程序时应该怎样处理两者之间关系。

3．如何利用 ST 源程序着手编写应用程序

写到这里相信大家已经看出着手编写 ST STR710 的应用程序是多么容易了。为了清晰起见，我们把流程列在下面：

① 建议备份 ST 的 01STR71x 软件包，以防无意修改或破坏。

② 建立新工作区和新项目，并根据前面介绍设置项目选件。

③ 给项目添加文件。必须添加的文件有：

71x_init.s （注：在项目下建立一个 startup 组，把 71x_init.s 和 71x_vect.s 放在此组）

71x_vect.s

71x_it.c （注：如不需要定时器 1 的中断处理，就把 void T1TIMI_IRQHandler(void)函数内部写空）

71x_lib.c

目标系统所用到设备驱动 c 程序；

④ 在连接器指向的 include 目录中，必须包含以下文件

71x_it.h

71x_map.h

71x_type.h

71x_conf.h

71x_lib.h

至少包含目标系统所用到设备的头文件(*.h)，一般 include 目录应包含所有头文件；

⑤ 根据用户目标系统的外设配置要求修改 71x_cong.h 文件；

⑥ 添加或编写用户自己的 main 主程序，在程序头上添加语句 #include 71x_lib.h，在程序中调用 ST 提供的外设驱动程序完成自己的操作需求。

01STR71x 软件包中有很多单个的片上外设的例子。初学者可以去再试验和了解如何操作这些设备。其中有一个关于 STR710-EVAL 的 12 位 A/D 转换的例子在 “ *GettingStarted* ” 目录下。用户可以直接点击此目录下的 *Led_Dimmer* 工作区名打开。你可以打开主程序窗口和 Watch 窗口，调节电位器（改变 AD 的输入电压），同时单步运行程序并观察变量 Value 的变化。

还有一点提示：由于 ARM 结构不提供位操作指令，所以对寄存器的位置位和位清除可以用下面语句形式实现。

```
GPIO0 -> PC0    |= 1<<3;                          // 将 GPIO0 的 PC0 控制寄存器的 bit 03 置 1
GPIO0 -> PC0    |= 0x0008;                       // 与上面一句的结果一样
EMI -> BCON0 = 0 | (1<<15) | (0x01<<2) |1;     // 将 BCON0 寄存器的 bit 15，bit 2 和 bit 0 置 1，其
                                                 // 他位置 0。
```