第 4 章　ADS 环境下汇编/C/C++程序开发

学习导读

本章对 ARM 汇编的 ADS 编译环境、AXD 调试环境、ARM 可执行二进制代码结构、ARM 汇编与 C 语言、C++语言混合编程进行了详细介绍。本章是全书的重点之一，无论是从事底层开发的读者还是从事上层应用软件开发的读者都需要掌握本章内容。本章所列的源代码程序均可直接在 ADS 环境下编译，读者在学习时可以结合代码一起学习。

在嵌入式程序开发中，BootLoader 程序、操作系统以及驱动程序都大量使用汇编和 C 语言作为开发语言，在上层应用程序开发中，也多采用 C/C++为开发语言。一般来说，BootLoader 程序中多采用汇编程序和 C 语言混合编程。操作系统中有少部分是汇编程序，绝大部分采用 C 语言编写，如 Linux 内核。驱动程序基本上都采用 C 语言为开发语言。为此，本章主要介绍 Windows 平台下 ARM 程序开发平台 ADS 的使用，以及 ADS 平台下编译汇编/C/C++程序原理及方法。

本章 4.1 节主要介绍 Windows 平台下 ARM 集成开发环境 ADS 以及程序开发流程。在 ADS 环境下，可以很方便编译汇编程序、C/C++程序，同时也可以编译 3 种语言的混合程序。

本章 4.2 节主要介绍如何在 C 语言程序中嵌套使用汇编程序，本节以 4 个实例为主线，重点介绍嵌套汇编程序的使用方法和使用技巧。

本章 4.3 节以 4 个实例为主线，主要介绍 C 语言程序与汇编程序混合编程内容。

本章 4.4 节概要介绍了在开发嵌入式 ARM 应用程序时所要遵守的 APCS 标准。

4.1　嵌入式软件开发平台 ADS1.2 应用

4.1.1　ADS 开发平台简介

ADS1.2 发布于 2001 年 12 月，是为嵌入式 ARM 设计的一整套软件开发工具，支持的主机系统有：IBM Compatible PCs with Windows 95 / 98 / 2000 / ME/NT4、Sun Workstations With Solaris 2.6 / 2.7 / 2.8、HP Workstations with HPUX 10.20 / 11、Red Hat Linux 6.2 / 7.1。同时，ADS 开发平台采用 FLEXLM 管理器，管理其 License。目前，ARM 公司已经公开这一部分源程序，用户可以方便地下载和使用。相对来说，ADS1.2 具有以下功能：

- 支持最新的 ARM 内核。
- 支持 ARM926EJ-S、ARM9EJ-S、VFPv2。
- 支持 V5TEJ 体系结构。
- ARMulator 仿真执行 Java Bytecode。

- Bytecode 显示 Jazelle 状态。
- 主机系统可支持 Red Hat Linux。
- 库管理器可合并各种库。
- 使用 pragmas，可实现出色代码和数据的布局。
- 为可重定位的代码设置新的链接选项。
- 编译器同时输出汇编码和目标码。

如图 4.1 所示为采用 ADS 1.2 开发嵌入式程序的步骤。

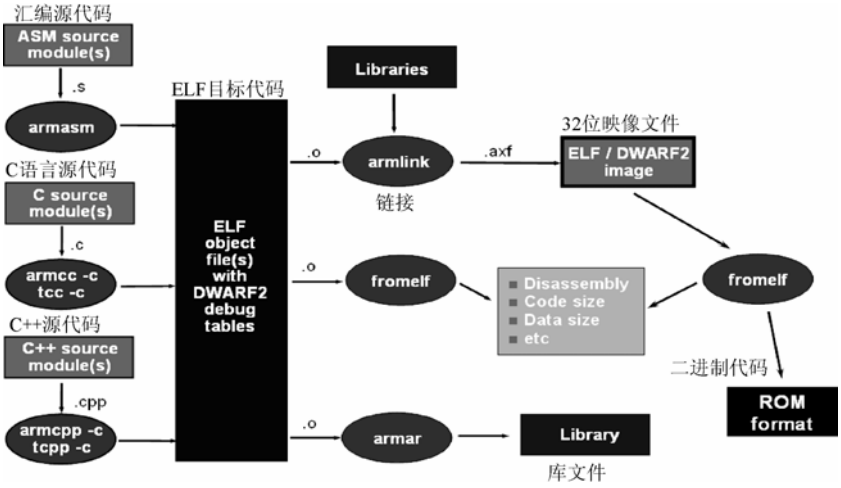


图 4.1　采用 ADS 1.2 开发嵌入式程序的步骤

如图 4.1 所示，在 ADS 环境中程序开发的主要步骤如下。

(1) 编译程序：使用编译器 armasm 编译以.s 后缀名的汇编程序，使用编译器 armcc 或 tcc 编译以.c 为后缀名的 C 语言程序，使用编译器 armcpp 或 tcpp 编译以.cpp 为后缀的 C++程序，编译完成后，生成以.o 为后缀的 ELF 目标文件。

(2) 链接程序：使用 armlink 将.obj 文件链接成.axf 为后缀的可执行文件，这种格式的文件可以在 ADS 1.2 平台用做软件仿真。

(3) 生成二进制文件：使用 fromelf 工具将.axf 文件转换成二进制文件，这种二进制文件可以直接在 ARM 处理器中运行。

(4) 下载程序到嵌入式平台的 Flash 中运行。

如果读者已经安装了 ADS 1.2 平台，则可以在命令行下直接使用这些命令(需要把 ADS 路径加到环境变量中)，当然，读者也可以直接打开 ADS 集成开发环境，使用按钮激活这些命令。在编译程序时，如果是 Thumb 指令，则需要在相应的编译命令前面加上 t，如汇编编译 Thumb 指令，则相应的编译 C 程序的编译器为 tcc。

4.1.2　ARM 命令参数说明

1. armasm 汇编编译指令

armasm 指令用来编译汇编程序，其指令格式：

```
armasm [选项] -o 目标文件 源文件
```

参数选项说明如下：

```
-Errors  错误文件名          ;指定一个错误输出文件
-I  目录 [,目录]             ;指定源文件搜索目录
-PreDefine  预定义宏         ;指定预定义的宏
-NOCache                     ;编译源代码时禁止使用Cache进行优化
-MaxCache <n>                ;编译源代码时使用Cache进行优化
-NOWarn                      ;关闭所有的警告信息
-G                           ;输出调试表
-keep                        ;在目标文件中保存本地符号表
-LIttleend                   ;生成小端(Little-endian)ARM代码
-BIgend                      ;生成大端(Big-endian)ARM代码
-CPU <target-cpu>            ;设立目标板ARM核类型，如arm920t
-16                          ;建立16位的thumb指令
-32                          ;建立32位的ARM指令
```

编译一个汇编文件指令示例如下：

```
c:\>armasm -LIttleend -cpu ARM920T -32 test.s
```

此指令将汇编语言程序 test.s 编译成小端模式、32 位支持 ARM920T CPU 的目标文件。

2. armcc/armcpp　C/C++编译器

armcc 用来编译 C 程序，armcpp 用来编译 C++程序，其指令格式如下：

```
命令：armcc [选项] 源文件1 源文件2 … 源文件n
```

这两个指令的选项说明如下：

```
-c                           ;编译但是不连接
-D                           ;指定一个编译时使用的预定义宏常量
-E                           ;仅对C程序源文件做预处理
-g                           ;产生调试信息表
-I                           ;指定头文件的搜索路径
-o<file>                     ;指定一个输出的目标文件
-O[0/1/2]                    ;指定源代码的优化级别
-S                           ;输出汇编代码来代替目标文件
-CPU <target-cpu>            ;设立目标板ARM核类型，如arm920t
```

编译一个 C 程序指令示例如下：

```
c:\>armcc -c -O1 -cpu ARM920T test.c
```

将 C 程序 test.c 进行编译但不连接，并进行二级优化，支持 ARM920T 处理器。

3. armlink 链接器

armlink 指令用来链接目标文件，其指令格式如下：

```
armlink [选项] 输入文件
```

此指令选项参数说明如下：

```
-partial                ;合并目标文件
-Output 文件            ;指定输出文件名
-scatter 文件           ;按照指定的文件为可执行文件建立内存映射
-ro-base 地址值         ;只读代码段的起始地址
-rw-base 地址值         ;RW/ZI段的起始地址
```

多个目标文件合并成一个目标文件指令如下：

```
c:\>armlink -partial bdmain.o bdport.o bdserial.o bdmmu.o bdisr.o -o testd.o
```

此指令将 bdmain.o、bdport.o、bdserial.o、bdmmu.o、bdisr.o 合并成一个目标文件 testd.o。

将几个目标文件编译成一个可执地文件指令如下：

```
c:\>armlink bd.o bdinit.o -scatter bdscf.scf -o test.axf
```

此指令使用 scatter 文件 bdscf.scf 将 bd.o、bdinit.o 编译成一个.axf 可执行文件 test.axf。

4. fromelf

fromelf 指令从.axf 格式文件生成一个二进制可执行文件。其指令格式如下：

```
fromelf [选项] 输入文件
```

此指令选项说明如下：

```
-bin 二进制文件名           ;产生二进制文件
-elf elf文件名              ;产生一个.elf文件
-text text文件名            ;产生.text文件
```

产生一个可执行的二进制代码示例如下：

```
c:\>fromelf test.axf –bin –o test.bin
```

此指令从 test.axf 生成二进制文件 test.bin。

4.1.3　可执行文件结构及内存映射

1. 可执行文件的结构

在 ADS 下，可执行文件有两种：

- 一种是.axf 文件，带有调试信息，可供 ADS 平台下的 AXD 调试工具使用，进行调试工作。
- 一种是.bin 文件，可执行的二进制代码文件，下载到嵌入式硬件设备 Flash 中运行。

以下重点介绍 .bin 文件的组成，ARM 可执行文件分为两种情况：存放态和运行态。

(1) 存放态。存放态是指可执行文件通过 fromelf 产生后，在存储介质(Flash 或磁盘)上的分布。此时可执行文件由两部分组成：只读部分和读/写部分。如图 4.2 所示为可执行文件的存放态结构。

图 4.2　可执行文件的存放态结构

- 只读部分包括代码段和只读数据段。代码段(.text)用来存放可执行 ARM 指令，只读数据段(.rodata)用来存储常量数据。
- 读/写部分包括读/写数据段(.data)和未初始化数据段(.bss)。

(2) 运行态。可执行文件通过装载过程，迁移到 RAM 中运行，这时候可执行文件就变成运行态，各个段数据也进行了迁移。文件中代码和数据分配情况如图 4.3 所示。主要包括以下几部分。

- RO 段：对应原来的只读部分，包括 .rodata 和 .text 两个部分。
- RW 段：为原来的 .data 部分，主要存放可读/写数据。
- ZI 段：为 .bss 段，所有内容初始化为 0。

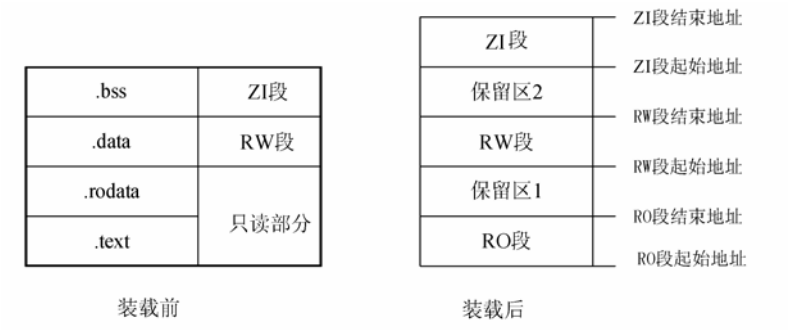


图 4.3　代码和数据分配情况

程序装载过程必须完成执行文件的各个段从存储介质上迁移到 RAM 指定的位置，而这个装载过程由启动程序(如 BootLoader)来完成。

2. 程序装载过程

在 ADS 中，可以通过两种方式来指定可执行代码各段在 RAM 中的位置：

- 一个是用 armlink 来指定。
- 一种是用 Scatter 文件来指定。

例如，指定 RAM 区的起始地址“0x30000000”，可以使用以下方法：

(1) 使用 armlink 指定代码段地址。对于一般的代码只需要指定两个段开始地址，即 RO 段的起始地址和 RW 段的起始地址，而 ZI 段紧接在 RW 段之后即可。关于此指令的使

用请参阅本章 armlink 命令相关内容。

(2) 使用 Scatter 文件指定代码段地址。可以通过 Scatter 文件来规范化指定可执行文件各段的详细地址。以下是一个 Scatter 文件示例：

```
MYLOADER 0x30000000
                        ;MYLOADER: 为可执行文件的名称， 可自定义
                        ;0x3000000: 起始地址
{
    RO 0x30000000
                        ;RO只读段名称
                        ;0x30000000: 只读代码段的起始地址
    {
        init.o (Init, +First)
                        ;Init代码段为可执行文件的第一部分
        * (+RO)         ;所有其他的代码段和只读数据段放在该部分
    }
    RW +0
                        ;RW: RW段的名称
                        ;+0: 表示RW段紧接着RO段
    {
        * (+RW)         ;所有RW段放在该部分
    }
    ZI +0
                        ;ZI: ZI段的名称
                        ;+0: 表示ZI段紧接着RW段
    {
        *(+ZI)          ;所有ZI段放在该部分
    }
}
```

在 ADS 中，各代码段宏定义如下：

```
|Image$$RO$$Base|          ;RO代码段起始地址
|Image$$RO$$Limit|         ;RO代码段结束地址
|Image$$RW$$Base|          ;RW代码段起始地址
|Image$$RW$$Limit|         ;RW代码段结束地址
|Image$$ZI$$Base|          ;ZI代码段起始地址
|Image$$ZI$$Limit|         ;ZI代码段结束地址
```

即在两个$$符号之间的段名称与 Scatter 中指定的段名称相同。

4.1.4　使用 ADS 集成 ARM 程序开发流程

1. 开发平台介绍

ADS 1.2 软件开发环境主要有 CodeWarrior 代码编辑环境以及 AXD 程序调度环境，如图 4.4 所示为 CodeWarrior 代码编辑环境，如图 4.5 所示为 AXD 程序调试环境。

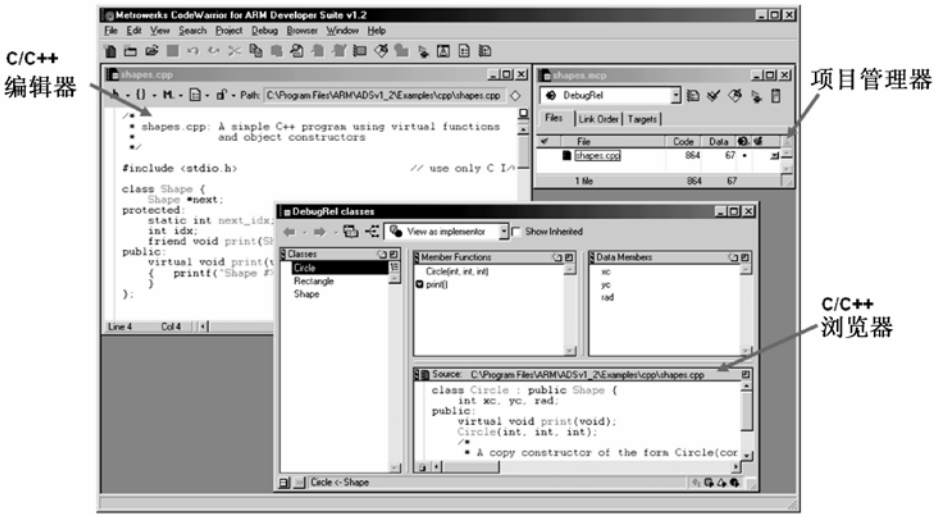


图 4.4　CodeWarrior 代码编辑环境

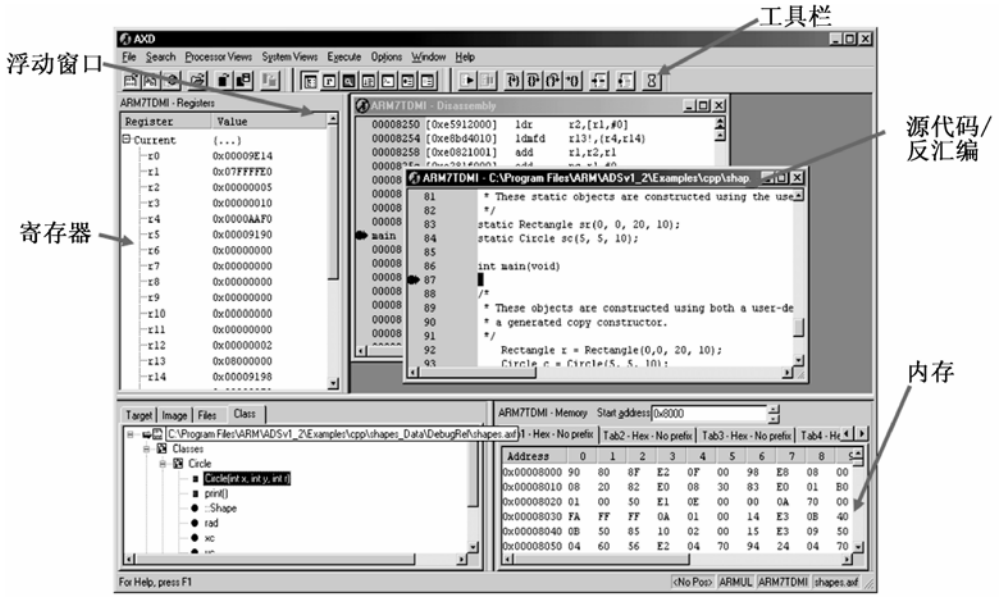


图 4.5　AXD 程序调试环境

2. 创建项目文件

以下以一个具体实例简要介绍使用 ADS 1.2 开发嵌入式程序的流程。

(1) 安装 ADS 1.2 软件包，打开如图 4.6 所示的 ADS 1.2 的 CodeWarrior for ARM Developer Suite 运行环境。

(2) 单击菜单命令“File→New”，打开如图 4.7 所示的新建工程项目对话框，在“Project name”文本框内输入此项目文件的文件名，同时在“Location”文本框中输入此工程文件所在的本地磁盘路径。

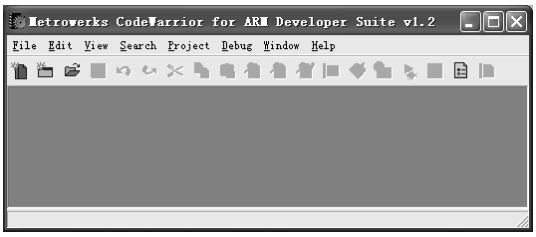


图 4.6　CodeWarrior for ARM Developer Suite 运行环境

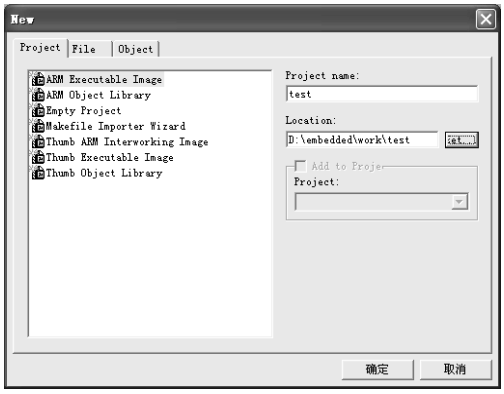


图 4.7　新建项目对话框

(3) 如图 4.8 所示，选择“Debug”版本，单击菜单命令“Edit→Debug Settings”，打开如图 4.9 所示的对话框，设置目标编译选项，包括 CPU 类型，编译器类型等。

(4) 选择“Target Setting”，打开如图 4.9 所示的参数设置对话框，在“Post-linker”文本框中选择“ARM fromELF”。

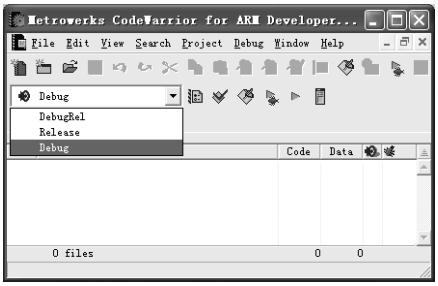


图 4.8　选择 Debug 版本

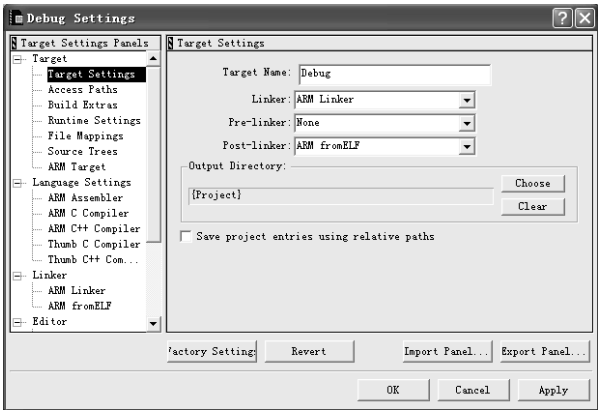


图 4.9　设置目标编译选项

(5) 在如图 4.9 所示的对话框中选择“ARM Linker”项，如图 4.10 所示，在“Output”选项卡的“Simple image”文本框中设置连接的“RO Base”(只读)和“RW Base”(读/写)地址，该内容即是前面介绍的运行态地址。当然，具体地址根据不同的开发板而各不一样，这是由系统的硬件决定的，如果读者仅仅是仿真使用，可以直接采用图中设置。

本处设置的地址 0xc80000 是开发板上 SDRAM 的真实地址，0xc200000 指的是系统可读/写的内存地址。也就是说，在 0xc080000～0xc1fffff 之间是只读区域，存放程序的代码段，在 0xc200000 开始是程序的数据段，可读可写。

如图 4.10 所示的设置只是一种简单设置，读者还可以如图 4.11 所示使用“Scatter”文件来设置链接运行态地址信息。

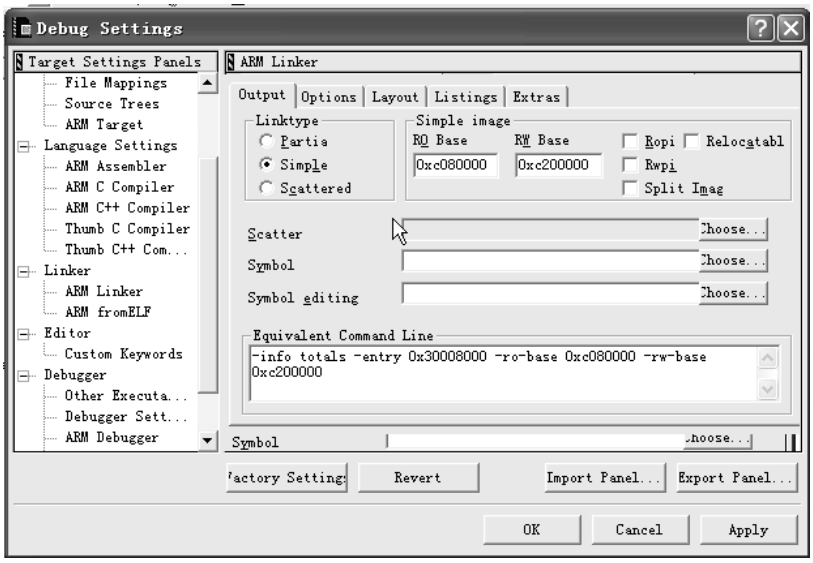


图 4.10　设置链接地址范围

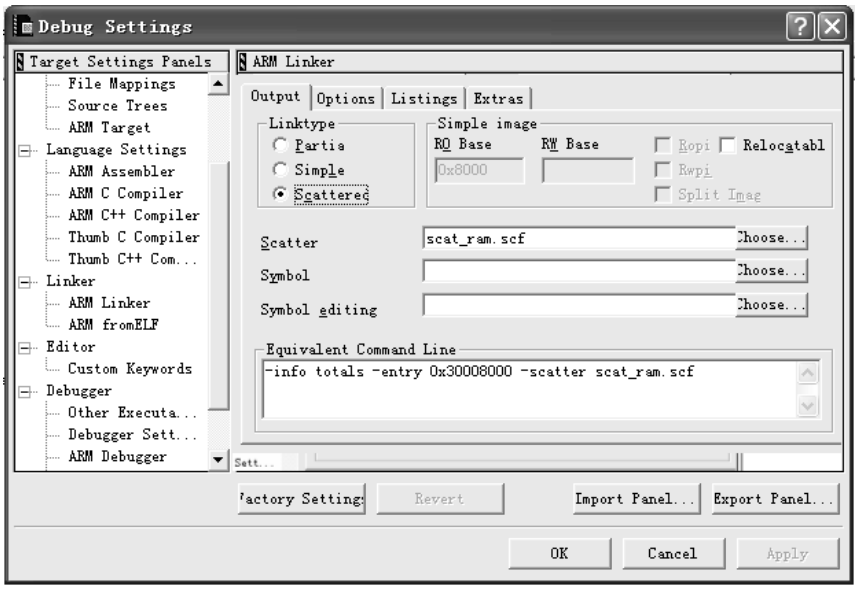


图 4.11　使用 Scatter 文件设置链接运行态地址信息

(6) 如果程序中使用了 C 语言或者 C++语言，则需按如图 4.12 所示设置 C 编译器，并在“Debug Settings”对话框中选择“ARM C Compiler”项，在命令行中添加“-apcs /interwork”，或者在“ATPCS”选项卡中选择“ARM/Thumb interworking”复选框。

(7) 如图 4.13 所示，读者还需要在“Layout”选项卡的“Place at beginning of image”中设置程序的入口模块，指定在生成的代码中，程序从哪一段代码开始执行，本处设置为 init.o 开始运行。其中，目标文件为 init.o，其所在段为 init 段。

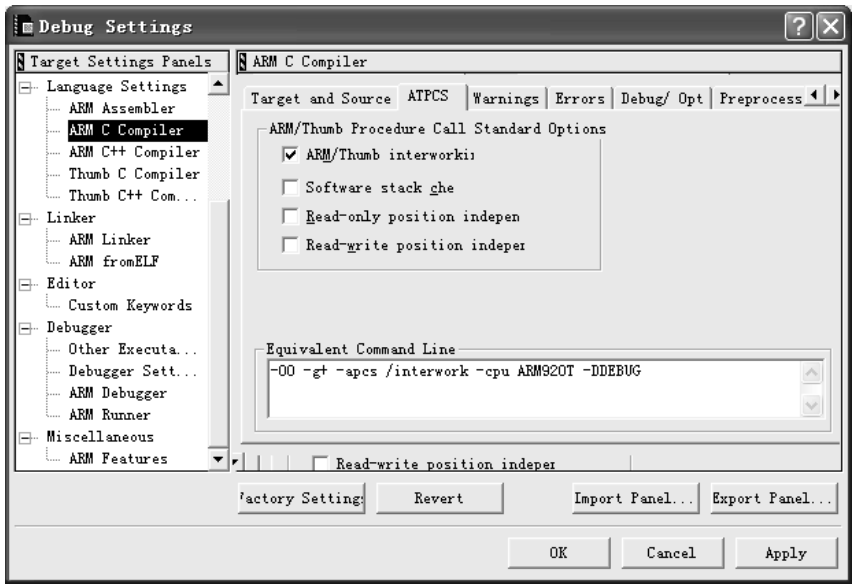


图 4.12　设置“ARM C Compiler”

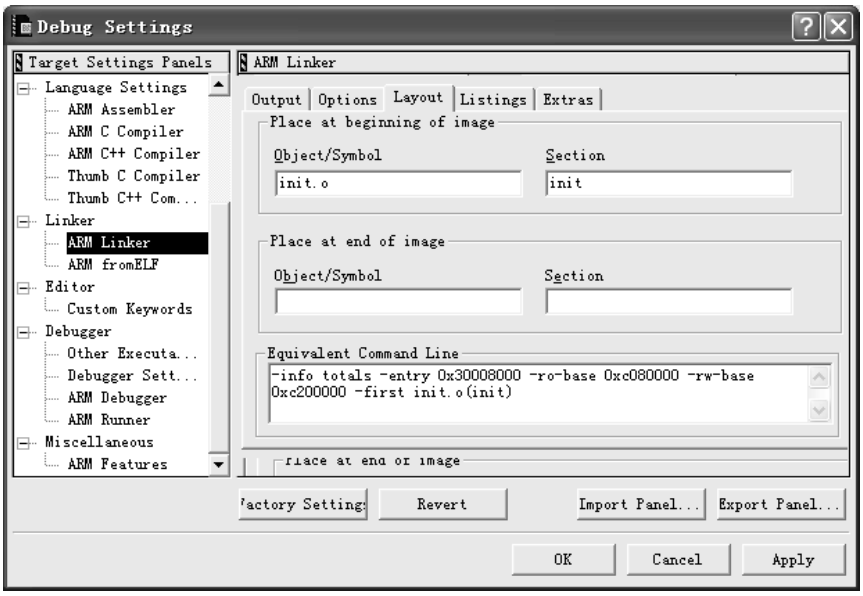


图 4.13　设置入口模块

(8) 如果要输出二进制文件，则需要在如图 4.14 所示的“Debug Settings”对话框中选择“ARM fromELF”项，在“Output file name”文本框中设置输出文件名为 system.bin，这就是要下载到开发板的嵌入式应用程序文件。

(9) 回到工程窗口，选择“Release”版本，单击菜单命令“Edit→Release Settings”对“Release”版本进行参数设置，设置步骤和方法同上。

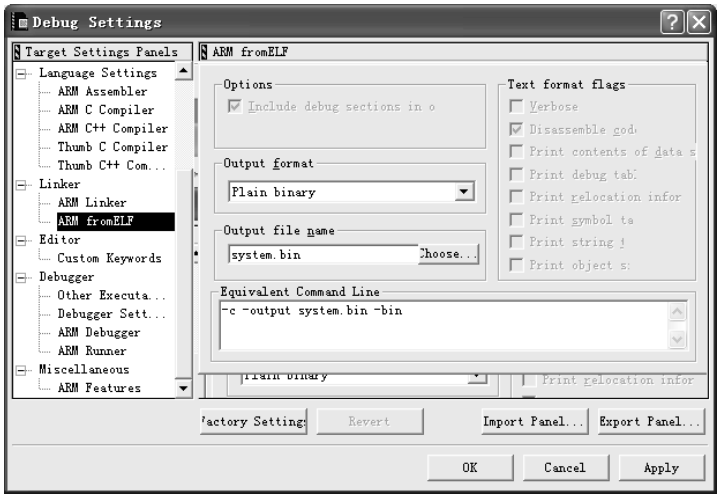


图 4.14　设置输出文件名

3. 添加文件

添加文件的方法如下。

(1) 单击如图 4.6 所示的菜单命令“File→New”，打开如图 4.15 所示的对话框，选择“File”选项卡，设置此文件的文件名和存储位置，然后在打开的编辑器中输入相应的程序。

(2) 如图 4.16 所示，在当前 Files 工程环境空白处单击鼠标右键，选择“Add Files…”选择新建文件，即可把此文件添加到当前项目中。

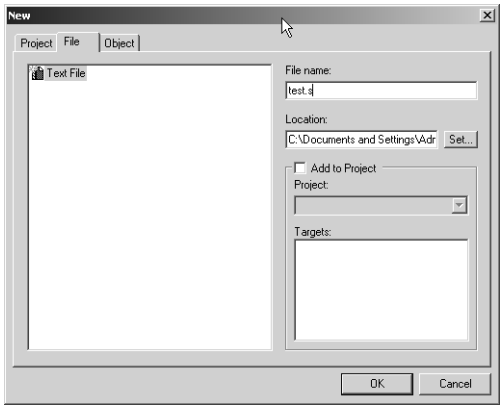


图 4.15　新建文件

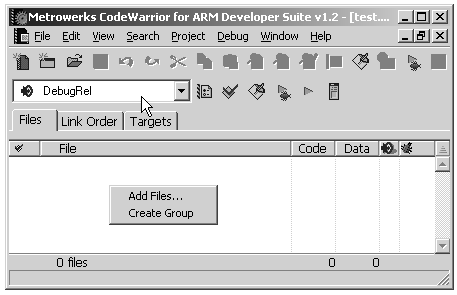


图 4.16　项目添加文件

(3) 单击菜单命令“Project→Add Files(项目→添加文件)”，将与工程相关的所有文件添加到当前工程中。至此，新建一个工程文件操作结束。

4. 编译程序

单击菜单命令“Project→Make”或者直接按 F7 键编译此文件，如果代码有错，将提示相关错误情况；如果文件无错，将显示如下所示的编译信息，此信息标识出目标文件大小、RO 段、RW 段和 ZI 段信息。

```
    ================================================================================
    Image component sizes
           Code    RO Data    RW Data    ZI Data     Debug
         236        60         0          0      2976   Object Totals    //目标
       10000       586         0        300      5192   Library Totals  //库
    //      代码    只读数据    读写数据  初始化为0部分
    ================================================================================
    //总大小
         Code    RO Data    RW Data    ZI Data     Debug
       10236       646         0        300      8168   Grand Totals
    ================================================================================
       Total RO  Size(Code + RO Data)10882 (  10.63kB)         //只读部分大小
       Total RW  Size(RW Data + ZI Data)300 (   0.29kB)        //读/写部分大小
       Total ROM Size(Code + RO Data + RW Data) 10882 (  10.63kB) //整体大小
    ================================================================================
```

5. 调试程序

调试程序的步骤如下。

(1) 如果读者需要调试程序，单击如图 4.6 所示的菜单命令“Project→Debug”或者直接按 F5 键打开如图 4.17 所示的 AXD 调试环境界面。

(2) 单击菜单命令“Options→Targets”，打开如图 4.18 所示的设置目标环境对话框，此对话框用来设置调试程序时程序运行的硬件平台。如果读者有仿真器 ICE，则可以选择 ADP 进行配置。关于 ADP 的设置请读者参阅选用 ICE 说明，如果仅是仿真调试程序，则选择 ARMUL，然后单击“OK”按钮返回。

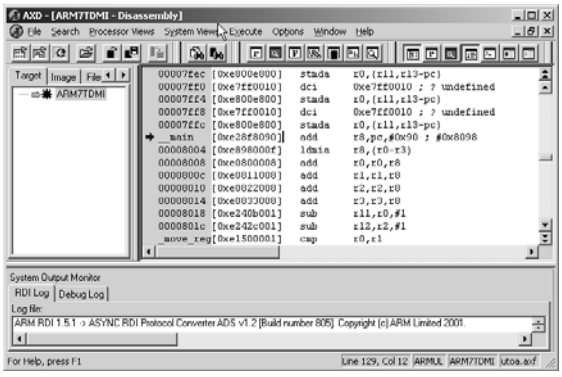


图 4.17　AXD 调试环境界面

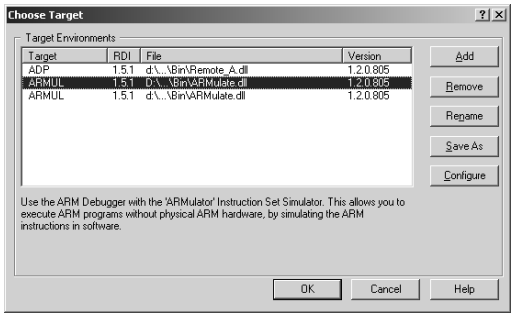


图 4.18　设置目标环境

(3) 在 AXD 中，读者可以单步执行程序，设置程序断点，另外，读者还可以通过“Processor Views”打开对应处理器的寄存器、存储器空间，查看程序运行时 CPU 内部寄存器及存储空间的数据变化。

4.1.5　使用 ADS 调试程序实例

在 ADS 环境下的 AXD 调试器，读者可以逐步跟踪汇编程序的执行，也可以很方便地查看到寄存器和内存单元的数据变化过程。

1. 程序源代码

本节以一个汇编程序为例，介绍如何在 ADS 下的调试器 AXD 环境下如何调试汇编程序。在此程序中，主要演示的是乘法指令的实现。以下是此汇编程序的内容。

```
AREA    EXAMPLE3,CODE,READONLY      ;声明一段名为 EXAMPLE3 的代码
    ENTRY

START
    MOV       R0,#0XFF000000        ;对 R0 赋值 #0XFF000000
    MOV       R1,#-0X10             ;对 R1 赋值 #-0X10
    MOV       R2,#0X1000000         ;对 R2 赋值 #0X1000000
    MOV       R3,#0                 ;对 R3 赋值 #0

    MULS      R4,R0,R1              ;R4＝R0*R1，结果影响寄存器CPSR 的值
    SMULLS    R6,R5,R1,R0           ;R6＝R1*R0 的低32位、R5＝R1*R0 的高32位，
                                    ;结果影响寄存器CPAR 的值
    UMULLS    R8,R7,R1,R0
    ;R8＝R1*R0 的低32位、R7＝R1*R0 的高32位，结果影响寄存器CPSR 的值

    MLA           R3,R0,R1,R2       ;R3＝R0*R1+R2
    SMLAL     R6,R0,R1,R2
    ;R6＝R1*R2 的低32位+R6，R0＝R1*R2 的高32位+R0
    UMLAL   R8,R0,R1,R2
    ;R8＝R1*R2 的低32位+R8，R0＝R1*R2 的高32位+R0，其中R1、R2 的值为32位无符号数
    ;R8、R0 的值为64位无符号数
    STOP
    BL STOP                         ;死循环
    END
```

2. 关键语句在调试器中的运行结果

在 AXD 环境下单击菜单命令“Process Views→Register”可以打开 ARM 处理器在不同模式下的寄存器，即可以以十六进制方式查看各寄存器数据。如图 4.19 所示为进行程序入口时各寄存器数据(所有寄存器数据初始化为 0，PC 指针指向的地址为程序员在编程时设置的 RO 地址(Read Only 地址，请参阅图 4.10)，CPSR 和 SPSR 寄存器中各位置位时(即为 1)为大写，清零时为小写。如图 4.20 所示为以下几行指令执行后的结果，从图中可以看出，影响的寄存器为 R0、R1、R2 和 R3。注意负数的存储形式。

```
    MOV             R0,#0XFF000000
    MOV             R1,#-0X10
    MOV             R2,#0X1000000
    MOV             R3,#0
```

如图 4.21 所示为执行以下乘法指令前各寄存器的值，如果是两个 32 位数乘法，其结

果为 64 位，需要两个寄存器来存储结果。如图 4.22 所示为乘法执行后的结果。注意带 S 后对 CPSR 的影响。

```
MULS     R4,R0,R1                ; R4＝(R0*R1)低32位
SMULLS   R6,R5,R1,R0             ; R6＝(R0*R1)低32位，R5＝(R0*R1)高32位
UMULLS   R8,R7,R1,R0             ; 无符号乘法
```

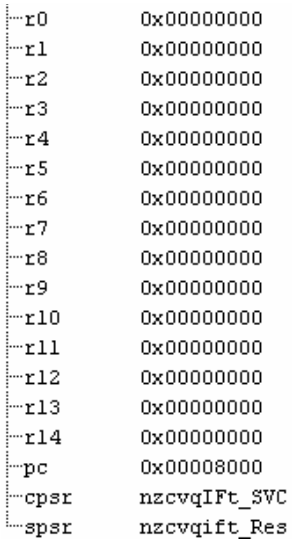


图 4.19　执行各寄存器数据前

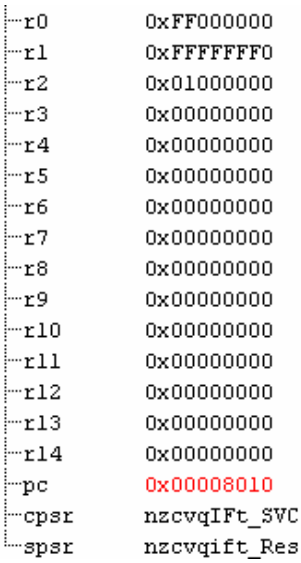


图 4.20　执行各寄存器数据后

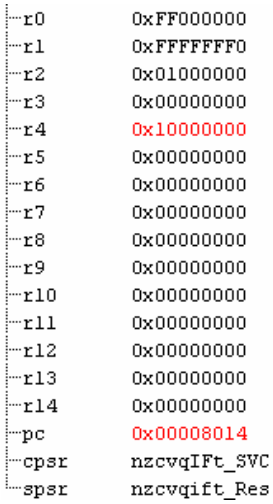


图 4.21　执行乘法前状态

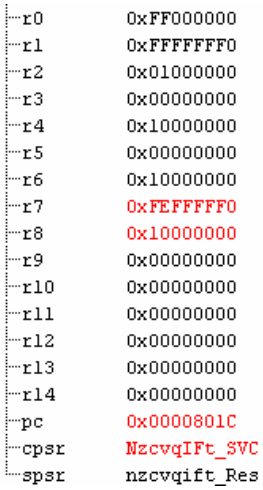
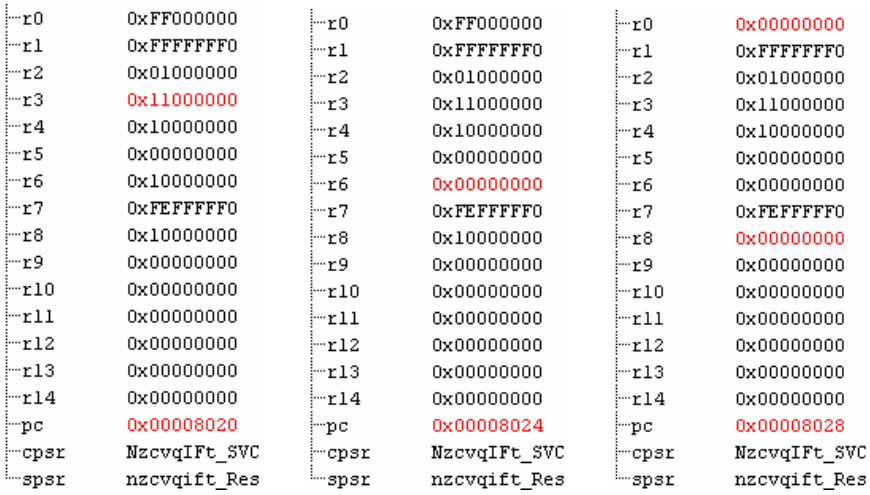


图 4.22　执行乘法后的状态

以下是执行带加法的乘法指令运算，如图 4.23 所示为运行后的状态。

```
MLA      R3,R0,R1,R2             ;(1) (带加法的乘法运算)
SMLAL    R6,R0,R1,R2             ;(2) (带加法的长乘法运算)
UMLAL    R8,R0,R1,R2             ;(3)
```



(a) 执行命令行(1)　(b) 执行命令行(2)　(c) 执行命令行(3)

图 4.23　执行带加法的乘法后的状态

以上演示了使用 AXD 调试时如何查看寄存数据变化的情况，如果要查看内存单元数据，可以在 AXD 环境下单击菜单命令“Process Views→Memory”来打开内存单元。另外，读者还可以查看变量、全局符号等相关参数的变化。关于这些内容，请读者参阅 ADS 手册相关章节。

4.2　C 语言内嵌汇编应用程序开发

在 C/C++程序中嵌套汇编程序可以访问处理器中某些用 C 语言不能访问到的内容，如使用 QADD、QSUB、QDADD 和 QDSUB 等溢出命令和协处理器指令。在 ARM 程序中内嵌汇编程序可以很方便地与 C/C++程序混合在一起。使用 armcc 和 armcpp 内嵌汇编程序支持绝大多数的 ARM 汇编指令、协处理器指令、半字指令和长乘法操作。如果要支持 Thumb 指令集，可以使用 tcc 和 tcpp。

4.2.1　内嵌汇编程序结构

ARM　C 编译器使用__asm 来指定内嵌汇编程序。

ARM　C++编译器在使用 ANSI C++ 标准时支持 asm，但为了限制汇编程序，要求每个语句尽可能简单。例如：

```
asm("instruction[;instruction]");
```

使用 asm 语法结构能够被 C++编辑器支持同时编译 C/C++程序。内嵌汇编程序结构如下：

```
__asm                          //使用的是双下画线
{
instruction [; instruction]
...
```

```
    [instruction]
    }
```

如果同一行中有两条语句程序，则需要在他们之间加上分号；如果同一行需要占用多行，则需要使用“\”符号。另外，不要在语句中间加入注释内容。

内嵌汇编允许受限访问物理寄存器，但不允许写 PC 寄存器。另外，跳转指令仅能使用 B 和 BL，不应该混合使用物理寄存器和 C/C++程序中的结构体。

tcc 和 tcpp 编译器使用 r12(ip)和 r3 寄存器来存储中间结果。在子程序调用时可能需要使用 r0～r3、r12(ip)、r14(lr)，这些寄存器最好不要同时作为物理寄存器使用。

和变量一样，物理寄存器需要初始化才能被访问，当物理寄存器被编译器用来保存和恢复 C/C++变量时，必须被分配到同一个寄存器中。另外，不要使用物理寄存器名作为变量名。

4.2.2　内链汇编实现字符串复制

1. 功能说明

本程序在子程序中嵌套使用汇编程序，在汇编程序中实现两个字符串的复制操作。然后，在主函数中调用子程序，实现一个字符串到一个数组的复制操作。

2. 源代码分析

```
//0401 strcopy.c
#include <stdio.h>
void my_strcpy(const char *src,  char *dst)
//子函数，src为源数据指针，dst为目的指针
{
    int ch;                                    //临时变量
    __asm                                      //内嵌汇编程序指示，双下画线
    {
    loop:
#ifndef __thumb            //如果没有定义__thumb，则以下为ARM指令实现字符串复制
        // ARM version
        LDRB    ch, [src], #1                  //读者数据，地址自加1
        STRB    ch, [dst], #1                  //加载数据，地址自加1
#else
        // Thumb version
        //thumb指令实现字符串复制
        LDRB    ch, [src]                      //加载数据到ch变量
        ADD     src, #1                        //地址自加1
        STRB    ch, [dst]                      //将ch数据加载到dst
        ADD     dst, #1                        //地址自加1
#endif
        CMP     ch, #0                         //比较是否复制完成
        BNE     loop
    }
}

int main(void)                                 //主函数
{
```

```
    const char *a = "Hello world!";          //定义源字符串指针
    char b[20];                              //定义目的数组
    my_strcpy (a,b);                         //执行复制命令
    printf("Original string: '%s'\n", a);    //打印源信息
    printf("Copied  string: '%s'\n", b);     //打印复制信息
    return 0;
}
```

4.2.3　内链汇编禁止和使能中断请求

1. 功能说明

本程序在C语言程序中嵌套汇编程序来实现使能和禁止IRQ请求，在C语言子程序中，使用__asm来实现CPSR寄存器I位清零和置位操作。因为此函数只能在私有模式中运行，所以修改CPSR和SPSR寄存器的数据必须在非用户模式下进行。

2. 源代码分析

```
//0402 irqs.c
__inline void enable_IRQ(void)           //使能IRQ
{
    int tmp;                             //临时变量
    __asm                                //内嵌汇编程序指示
    {
        MRS tmp, CPSR                    //CPSR→temp
        BIC tmp, tmp, #0x80              //清I位
        MSR CPSR_c, tmp                  //temp→CPSR
    }
}
__inline void disable_IRQ(void)          //禁止IRQ
{
    int tmp;
    __asm
    {
        MRS tmp, CPSR                    //CPSR→temp
        ORR tmp, tmp, #0x80              //置I位
        MSR CPSR_c, tmp                  //temp→CPSR
    }
}
int main(void)                           //主函数调用
{
    disable_IRQ();                       //调用子函数
    enable_IRQ();                        //调用子函数
}
```

4.2.4　内链汇编实现64位乘法

1. 功能说明

本程序中实现两个数组元素的64位乘法运算，然后求和，即将两个数组a[10]与b[10]对应位相乘，然后将结果相加。在进行乘法运算时，使用内嵌汇编程序实现64位的乘法运

算。在此程序中，读者需要注意的是相应变量的类型以及如何有效存储 64 位结果。在 AXD 中调试运行结果如下：

```
now the number is 0
,the sum is 10
now the number is 1
,the sum is 28
now the number is 2
,the sum is 52
now the number is 3
,the sum is 80
now the number is 4
,the sum is 110
now the number is 5
,the sum is 140
now the number is 6
,the sum is 168
now the number is 7
,the sum is 192
now the number is 8
,the sum is 210
now the number is 9
,the sum is 220
```

2. 源代码分析

```
//0403 dotprod.c
/* Example 4-3 */
#include <stdio.h>
#define      lo64(a)   (((unsigned*) &a)[0])
// low 32 bits of a long long，存储结果低32位
#define      hi64(a)   (((int*) &a)[1])
// high 32 bits of a long long，存储结果的高32位

__inline__int64 mlal(__int64 sum,int a,int b)   //内联函数，一起编译到main函数中
{
    #if !defined(__thumb) && defined(__TARGET_FEATURE_MULTIPLY)
    __asm                                       //内嵌汇编程序
        {
             SMLAL lo64(sum), hi64(sum), a, b  //长乘法运算
        }
    #else
        sum += (__int64) a * (__int64) b;
    #endif
       return sum;                              //返回结果
}

__int64 dotprod(int *a, int *b, unsigned n)    //求和运算
```

```
{
    __int64 sum = 0;
    do
    {
        sum = mlal(sum, *a++, *b++);                    //实现乘法并进行求和
        printf("now the number is %d\n,the sum is %lld\n",(10-n),sum);
                                                        //打印每次运行结果，共10次
    }
    while (--n != 0);
    return sum;
}

int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };//进行运算的数组元素一
int b[10] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };//进行运算的数组元素二
int main(void)
{
    dotprod(a, b, 10);                              //调用子函数实现
    return 0;
}
```

4.3　C 程序与汇编程序汇合编程

本节主要介绍部分 ARM 汇编及 C 程序汇合编程内容，这些程序都有一定的代表性。

(1) utoa1.s 汇编程序中实现整型数到字符串的转换，在此项中使用了栈和递归函数，在此项目中还使用了 udiv10.s 和 utoatest.c 文件。

(2) divc.c 程序中用来实现常数的除法运算。

(3) random.s 程序用来实现用汇编程序产生伪随机数字，此程序中使用文件名为 randtest.c 的 C 程序来进行测试。

(4) bytedemo.c 用来实现字节反转操作。

4.3.1　无符号整型数到字符串的转换程序

1. 功能说明

转换程序实现了一个无符号整型数到字符串的转换，整个项目中使用了 utoa1.s、udiv10.s 和 utoatest.c3 个文件，其中：

- utoa1.s 汇编程序中实现无符号整型数到字符串的转换，在此项中使用了栈和递归函数。
- udiv10.s 汇编程序被 utoa1.s 调用，使用汇编程序实现无符号整型数除以 10 的运算，然后将存储商和余数。
- utoatest.c 是一个 C 语言程序，实现代码功能测试。

2. 程序运行结果

如果读者安装了 ADS，即可以在 Windows 环境下测试此程序。测试之前，读者需要编

译此工程项目中的汇编源程序和 C 语言源程序，以下是在命令行下编译测试此程序的过程。

```
    …\code\ch04\explasm>armasm utoa1.s -o utoa1.o      //编译汇编程序utoal.s
    …\code\ch04\explasm>armasm udiv10.s -o udiv10.o    //编译汇编程序udiv10.s
    …\code\ch04\explasm>armcc -c utoatest.c            //编译C程序utoatest.c
    …\code\ch04\explasm>armlink utoa1.o udiv10.o utoatest.o -o utoatest
                                                       //链接程序为utoatest
```

编译完成后，使用软件仿真程序 armsd 测试此程序的运行情况。

```
    …\code\ch04\explasm>armsd utoatest
    //在Windows平台仿真运行此程序
    ARM Source-level Debugger, ADS1.2 [Build 805]      //显示当前运行平台
    Software supplied by: Team-EFA
    ARMulator ADS1.2 [Build 805]
    Software supplied by: Team-EFA
    ARM7TDMI, BIU, Little endian, Semihosting, Debug Comms Channel, 4GB, Mapfile,
                                                       //CPU信息
    Timer, Profiler, Tube, Millisecond [20000 cycles_per_millisecond],
Pagetables,
    IntCtrl, Tracer, RDI Codesequences
    Object program file utoatest                //运行的文件
    armsd:                                      //armsd命令提示符
    armsd: r                                    //输入r，按回车键，显示当前寄存器信息
      r0  = 0x00000000  r1  = 0x00000000  r2  = 0x00000000  r3  = 0x00000000
      r4  = 0x00000000  r5  = 0x00000000  r6  = 0x00000000  r7  = 0x00000000
      r8  = 0x00000000  r9  = 0x00000000  r10 = 0x00000000  r11 = 0x00000000
      r12 = 0x00000000  r13 = 0x00000000  r14 = 0x00000000
      pc  = 0x00008000  cpsr = %nzcvqIFt_SVC  spsr = %nzcvqift_Reserved_00
    armsd: g                                    //输入r，运行此程序
    Enter number:                               //提示输入要转换的无符号整型数
    12345                                       //输入字符后，按回车键
    utoa yields:                                //转换完成，显示转换后字符串信息
    12345
    Program terminated normally at PC = 0x0000a0e0 (_sys_exit + 0x8)
                                                //提示程序被swi中断，结束
    +0008 0x0000a0e0: 0xef123456  V4.. :    swi     0x123456
```

如果读者需要查看 armsd 调试环境的可用命令，可以输入“？”符号查看。

```
    armsd: ?                                  //输入?符号查看armsd提示符下可用命令
    help [<keyword>]

    Display help information on one of the following commands:

    Registers       Fpregisters  Coproc          CRegisters     CREGDef
    CWrite          Step         Istep           Examine        List
    Quit            Obey         Go              Break          Unbreak
    Watch           UNWatch      Print           CONtext        OUt
    IN              WHere        BAcktrace       Variable       SYmbols
    LSym            LEt          Arguments       LAnguage       Help
    Type            CAll         WHIle           ALias          LOad
```

```
LOG             RELoad       READsyms        FInd           PUtfile
GEtfile         LOCalvar     COMment         PAuse          LOADConfig
SElectconfig    LISTConfig   LOADAgent       PROfon         PROFOFf
PROFClear       PROFWrite    CCin            CCOut          PROCessor
SYS             SETregister  TRacetrigger    TRACEExtent    TRACEWrite
TRACEStart      TRACESTOp    TRACEFlush
......
```

3. 算法描述

算法程序描述很简单，将一个无符号整型数转换为一个字符串可以采用以下方式实现：将该数除以 10 后得到商和余数，此余数作为转换后字符串的最后一位，然后再将商进行除以 10 的运算，余数作为字符串的倒数第二位，依次进行，直到商为 0 为止。例如，要将无符号整型数 123 转换为字符串，过程如下：

(1) 将 123 除以 10，得到余数为 3，商为 12，将余数 3 作为字符串的最后一位，此时商大于 0，继续进行运算；

(2) 将商 12 除以 10，得到余数为 2，商为 1，将余数 2 作为字符串的倒数第二位，此时商大于 0，继续进行运算；

(3) 将商 1 除以 10，得到余数为 1，商为 0，将余数 1 作为字符串的倒数第三位，此时商等于 0，运算结束。

4. 源码分析

源码分析如下。

(1) utoatest.c 测试程序。utoatest.c 是一个 C 语言编写的测试程序，主要实现输入/输出、打印等功能。

```
//utoatest.c
#include <stdio.h>                        //需要用到基本输入/输出函数
extern char *utoa( char *string, signed int num ); //引入外部函数
int main()
{
  signed int num;                         //此变量用来存储欲转换的无符号整型数据
  char buffer[ 12 ];                      //存储转换后的字符串
  puts( "Enter number:" );                //要求用户输入数据
  if( scanf( "%d", &num ) == 1 )          //将输入的数据存储在num变量中
  {
    *utoa( buffer, num ) = 0;
    //调用utoa函数实现转换功能，buffer和num是相应参数
    puts( "utoa yields:" );               //输出提示
    puts( buffer);                        //输出转换到buffer中的字符串
  }
  return( 0 );                            //返回
}
```

(2) utoal.s 汇编程序。utoal.s 汇编程序是实现无符号整型数到字符串转换的汇编实现程序，其源代码如下：

```
//utoal.s
```

```
; <string> = "<number/10><number%10>"  //此为算法
  AREA |utoa$$code|, CODE, READONLY    //定义段
  EXPORT utoa                          //输出utoa供其他文件使用
  IMPORT udiv10                        //引入其他文件的udiv10函数
utoa                                   //标号utoa
    //递归实现无符号整型数转换到字符串功能
    //在入口位置，a1为buffer起始地址，a2为无符号整型数
    //在结束时，a1存储字符串结束地址
  STMFD  sp!, {v1, v2, lr}     ;将v1和v2变量以及lr入栈，lr为返回地址
  MOV    v1, a1                ;保存a1到v1中，a1为传递的参数buffer地址
  MOV    v2, a2                ;保存a2到v2中，a2为传递的参数无符号整型数num
  MOV    a1, a2                ;a1和a2中都保持欲转换的数num
  BL     udiv10                ;调用udiv10，得到该数除以10后的商，商存储在a1中
  //以下这两句实现被除数减去商的10倍，从而得到余数
    SUB    v2, v2, a1, LSL #3  ;number1=number - 8*quotient
  SUB    v2, v2, a1, LSL #1    ;余数=number1 - 2*quotient
  CMP    a1, #0                ;比较商是否为0
  MOVNE  a2, a1                ;如果不为0，将商传递给a2，否则不执行
  MOV    a1, v1                ;将buffer地址传递给a1
  BLNE   utoa                  ;如果比较结果不为0，继续执行utoa
  ADD    v2, v2, #'0'
  STRB   v2, [a1], #1          ;将余数传递给buffer

  LDMFD  sp!, {v1, v2, pc}      ;出栈操作

  END
```

(3) udiv10.s 汇编程序。udiv10.s 汇编程序实现一个整数除以 10 的运算，返回商。

```
//udiv10.s
      CODE32
       AREA |div10$code|, CODE, READONLY
       EXPORT udiv10                    ; 输出udiv10，以供其他函数调用
udiv10
; 标号udiv10，a1为输入参数，返回的商存储在a1中，返回的余数在a2中
       SUB       a2, a1, #10            ; a2=a1-10
       SUB       a1, a1, a1, lsr #2     ; 逻辑右移2位后进行减法，a1=a1-a1/2/2
       ADD       a1, a1, a1, lsr #4
; 逻辑右移4位后进行加法，a1=a1-a1/2/2/2/2
       ADD       a1, a1, a1, lsr #8     ; 逻辑右移8位后进行加法
       ADD       a1, a1, a1, lsr #16    ; 逻辑右移16位后进行加法
       MOV       a1, a1, lsr #3         ; 逻辑右移3位
       ADD       a3, a1, a1, lsl #2
       SUBS      a2, a2, a3, lsl #1
       ADDPL     a1, a1, #1             ; 商
       ADDMI     a2, a2, #10            ; 余数
       MOV       pc, lr
       END
```

4.3.2　简单除法运算程序

1. 程序功能说明

ARM 指令中没有除法运算，如要实现除法，则需用子程序，本示例程序中实现整数除法运算的 ARM 指令集实现(此例程只实现了除数为(2^n－2^m)和(2^n＋2^m)整数除法)，即程序将打印出被除数为用户输入的数用 ARM 汇编程序实现。

2. 程序运行结果

以下介绍程序运行结果。

(1) 编译程序。

```
    E:\ptpress\embedded ARM\code\ch04\explasm>armcc divc.c -o divc_arm
                                  //编译divc.c
    E:\ptpress\embedded ARM\code\ch04\explasm>dir divc_arm
    ......
    2007-03-17  09:13          36,580 divc_arm
    ......
```

(2) 测试除数能够被 2 整除的情况。

```
    E:\ptpress\embedded ARM\code\ch04\explasm>armsd divc 8     //仿真程序2^n情况
    ARM Source-level Debugger, ADS1.2 [Build 805]
    Software supplied by: Team-EFA
    ARMulator ADS1.2 [Build 805]
    Software supplied by: Team-EFA
    ARM7TDMI, BIU, Little endian, Semihosting, Debug Comms Channel, 4GB, Mapfile,
    Timer, Profiler, Tube, Millisecond [20000 cycles_per_millisecond],
Pagetables,
    IntCtrl, Tracer, RDI Codesequences
    Object program file divc
    armsd: g                    //输入g运行
    8 is an easy case           //提示除以8很简单，只需要进行移位操作，故没有列出代码
    Program terminated normally at PC = 0x0000ae34 (_sys_exit + 0x8)
    +0008 0x0000ae34: 0xef123456  V4.. :    swi     0x123456
    armsd:q                     //然后输入q退出
```

(3) 测试除数为 2^n−2^m 的情况。

```
    E:\ptpress\embedded ARM\code\ch04\explasm>armsd divc 7 //测试2^n－2^m的情况
    ......
    armsd: g                                    //输入g运行
    ; generated by ARM divc                    //7＝2^3－2^0，故实现代码如下所示
    //实现代码起始位置
           CODE32
           AREA |div7$code|, CODE, READONLY
           EXPORT udiv7
    udiv7
    ; takes argument in a1
    ; returns quotient in a1, remainder in a2
    ; cycles could be saved if only divide or remainder is required
```

```
        SUB    a2, a1, #7
        MOV    a1, a1, lsr #1
        ADD    a1, a1, a1, lsr #3
        ADD    a1, a1, a1, lsr #6
        ADD    a1, a1, a1, lsr #12
        ADD    a1, a1, a1, lsr #24
        MOV    a1, a1, lsr #2
        RSB    a3, a1, a1, lsl #3
        SUBS   a2, a2, a3, lsl #0
        ADDPL  a1, a1, #1
        ADDMI  a2, a2, #7
        MOV    pc, lr
        END
//实现代码结束位置
Program terminated normally at PC = 0x0000ae34 (_sys_exit + 0x8)
+0008 0x0000ae34: 0xef123456  V4.. :    swi     0x123456
armsd:q                                     //然后输入q退出
```

(4) 测试除数为 2^n+2^m 的情况。

```
E:\ptpress\embedded ARM\code\ch04\explasm>armsd divc 10
//测试除数为2^n＋2^m的情况
......
armsd: g                                    //输入g运行
; generated by ARM divc
//10＝2^3+2^1，故实现代码如下所示
        CODE32
        AREA |div10$code|, CODE, READONLY
        EXPORT udiv10
udiv10
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
        SUB    a2, a1, #10
        SUB    a1, a1, a1, lsr #2
        ADD    a1, a1, a1, lsr #4
        ADD    a1, a1, a1, lsr #8
        ADD    a1, a1, a1, lsr #16
        MOV    a1, a1, lsr #3
        ADD    a3, a1, a1, lsl #2
        SUBS   a2, a2, a3, lsl #1
        ADDPL  a1, a1, #1
        ADDMI  a2, a2, #10
        MOV    pc, lr
        END
//实现代码结束位置
Program terminated normally at PC = 0x0000ae34 (_sys_exit + 0x8)
+0008 0x0000ae34: 0xef123456  V4.. :    swi     0x123456
armsd:q                                     //然后输入q退出
```

(5) 测试除数不是 2^n+2^m 和 2^n−2^m 的情况。

```
E:\ptpress\embedded ARM\code\ch04\explasm>armsd divc 13
......
```

```
armsd: g                          //输入g运行，13不能用2^n+2^m和的2^n－2^m表示
13 is not one of (2^n-2^m) or (2^n+2^m)    //提示此数没有实现
Program terminated normally at PC ＝ 0x0000ae34 (_sys_exit + 0x8)
+0008 0x0000ae34: 0xef123456  V4.. :    swi     0x123456
armsd:q                                    //然后输入q退出
```

(6) 测试除数为非法字符情况。

```
E:\ptpress\embedded ARM\code\ch04\explasm>armsd divc e
......
armsd: g                                   //输入g运行，e是一个非法字符
0 is not sensible                          //提示错误
Program terminated normally at PC ＝ 0x0000ae34 (_sys_exit + 0x8)
+0008 0x0000ae34: 0xef123456  V4.. :    swi     0x123456
armsd:q                                    //然后输入q退出
```

3. 算法说明

整数除法是以乘法为基础的。例如，x、y 为整数，x 除以 y 相当于 x 乘以 y 的倒数。

```
x/y ＝＝ x * (1/y)
    ＝＝ (x * (2^32/y)) / 2^32    //同时乘以2^32
    ＝＝ (x * (2^32/y)) >> 32     //相当于(x * (2^32/y))右移32位
```

从而可以得到 64 位结果(x*(2^32/y))的高 32 位。

如果 y 是一个常数，而且(2^32/y)也是一个常数，如下列数中：

```
 y              (2^32/y)
 2     10000000000000000000000000000000      #
 3     01010101010101010101010101010101      *
 4     01000000000000000000000000000000      #
 5     00110011001100110011001100110011      *
 6     00101010101010101010101010101010      *
 7     00100100100100100100100100100100      *
 8     00100000000000000000000000000000      #
 9     00011100011100011100011100011100      *
10     00011001100110011001100110011001      *
11     00010111010001011101000101110100
12     00010101010101010101010101010101      *
13     00010011101100010011101100010011
14     00010010010010010010010010010010      *
15     00010001000100010001000100010001      *
16     00010000000000000000000000000000      #
17     00001111000011110000111100001111      *
18     00001110001110001110001110001110      *
19     00001101011110010100001101011110
20     00001100110011001100110011001100      *
21     00001100001100001100001100001100      *
22     00001011101000101110100010111010
23     00001011001000010110010000101100
24     00001010101010101010101010101010      *
25     00001010001111010111000010100011
```

所有标识了#的数是 2^n，这种数很好解决，标记有*符号的是一个循环规律。这些数都是 2^n+2^m 或者是 2^n−2^m (for n>m)的组合。

```
  n     m        (2^n+2^m)  n       m       (2^n-2^m)

  1     0        3          1       0       1
  2     0        5          2       1       2
  2     1        6          2       0       3
  3     0        9          3       2       4
  3     1        10         3       1       6
  3     2        12         3       0       7
  4     0        17         4       3       8
  4     1        18         4       2       12
  4     2        20         4       1       14
  4     3        24         4       0       15
  5     0        33         5       4       16
  5           1         34      5       3       24
  5     2        36         5       2       28
  5     3        40         5       1       30
  5     4        48         5       0       31
```

除以 10 汇编代码程序如下：

```
SUB  a1,  x,  x, lsr #2     ; a1 = x* %0.11000000000000000000000000000000
ADD  a1, a1, a1, lsr #4     ; a1 = x* %0.11001100000000000000000000000000
ADD  a1, a1, a1, lsr #8     ; a1 = x* %0.11001100110011000000000000000000
ADD  a1, a1, a1, lsr #16    ; a1 = x* %0.11001100110011001100110011001100
MOV  a1, a1, lsr #3         ; a1 = x* %0.00011001100110011001100110011001
```

其中，SUB　a1,　x,　x, lsr #2 结果如下所示：

```
a1 = x - x/4
   = x - x* %0.01
   = x* %0.11
```

但这没有考虑到低 32 向高 32 位借位的情况，要采用以下方法，除以 10 改进方案如下：

```
div10
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
    SUB          a2, a1, #10           ; 保持(x-10) 供后面使用
    SUB          a1, a1, a1, lsr #2
    ADD       a1, a1, a1, lsr #4
    ADD       a1, a1, a1, lsr #8
    ADD       a1, a1, a1, lsr #16
    MOV       a1, a1, lsr #3
    ADD       a3, a1, a1, lsl #2
    SUBS      a2, a2, a3, lsl #1       ; 计算 (x-10)-(x/10)*10
    ADDPL     a1, a1, #1               ; fix-up quotient
    ADDMI     a2, a2, #10              ; fix-up remainder
    MOV       pc, lr
    END
```

4. 代码分析

```
#include <stdio.h>
#include <stdlib.h>
#define BANNER                      "generated by ARM divc"
#define DIVIDE_BY_2_MINUS_1         0
#define DIVIDE_BY_2_PLUS_1          1

typedef unsigned int uint;
uint log2( uint n )
{
  uint bit, pow, logn;
  for( bit = 0, pow = 1; bit < 31; bit++, pow <<= 1 )
  {
    if( n == pow ) logn = bit;
  }
  return( logn );
}

int powerof2( int n )                    //被主函数main调用，检测能否被2整除
{
  return( n == ( n & (-n) ) );           //将该数与该数的相反数按位与
}
//powerof2汇编实现代码如下：
//      rsb       r1,r0,#0               //取反操作后加1，即负数
//      bics      r0,r0,r1               //位清零
//      bne       nozero                 //如果不为0，说明不能整除
//      mov       r0,#1                  //如果为0，返回1
//retback
//      mov       pc,r14
//nozero
//      mov       r0,#0                  //返回0
//      b         retback                //子程序返回

void dodiv2m1( uint logd, uint logmsb )
{
  /* 实现除法 2^n-1 */
  printf( "\tMOV    a1, a1, lsr #1\n" );
  while( logd < 32 )
  {
    printf( "\tADD    a1, a1, a1, lsr #%d\n", logd );
    logd <<= 1;
  }
  printf( "\tMOV    a1, a1, lsr #%d\n", logmsb - 1 );
}

void dodiv2p1( uint logd, uint logmsb )
{
  /* 实现除法 2^n + 1 */
```

```
    printf( "\tSUB    a1, a1, a1, lsr #%d\n", logd );
    while( logd < 16 )
    {
      logd <<= 1;
      printf( "\tADD    a1, a1, a1, lsr #%d\n", logd );
    }
    printf( "\tMOV    a1, a1, lsr #%d\n", logmsb );
  }

  void loada4( uint type, uint lsb, uint msb )
  {
    /* 将该数作为一个立即数处理时太大，因此使用寄存器a4来加载 */
    printf( "\tMOV    a4, #0x%x\n", msb );
    switch( type )
    {
    case DIVIDE_BY_2_MINUS_1:
      printf( "\tSUB    a4, a4, #0x%x\n", lsb );
      break;
    case DIVIDE_BY_2_PLUS_1:
      printf( "\tADD    a4, a4, #0x%x\n", lsb );
      break;
    default:
      fputs( "Internal error", stderr );
    }
  }

  void divideby2( uint type, uint n, uint lsb, uint msb )  //输出汇编实现方法
  {
    uint loglsb;
    uint logmsb;
    uint usinga4;
    loglsb = log2( lsb );
    logmsb = log2( msb );
    printf( "; %s\n\n", BANNER );
    printf( "\tCODE32\n\n" );
    printf( "\tAREA |div%d$code|, CODE, READONLY\n\n", n );
    printf( "\tEXPORT udiv%d\n\n", n );
    printf( "udiv%d\n", n );
    printf( "; takes argument in a1\n" );
    printf( "; returns quotient in a1, remainder in a2\n" );
    printf( "; cycles could be saved if only divide or remainder is required\n" );

    usinga4 = ( n >> loglsb ) > 255;
    if( usinga4 )
    {
      loada4( type, lsb, msb );
      printf( "\tSUB    a2, a1, a4\n" );
    }
    else
    {
```

```
      printf( "\tSUB    a2, a1, #%d\n", n );
    }

    /* 1/n as a binary number consists of a simple repeating pattern    */
    /* The multiply by 1/n is expanded as a sequence of ARM instructions */
    /* (there is a rounding error which must be corrected later)       */
    switch( type )
    {
    case DIVIDE_BY_2_MINUS_1:
      dodiv2m1( logmsb - loglsb, logmsb );
      /* Now do multiply-by-n */
      printf( "\tRSB    a3, a1, a1, lsl #%d\n", logmsb - loglsb );
      break;

    case DIVIDE_BY_2_PLUS_1:
      dodiv2p1( logmsb - loglsb, logmsb );
      /* Now do multiply-by-n */
      printf( "\tADD    a3, a1, a1, lsl #%d\n", logmsb - loglsb );
      break;

    default:
      fputs( "Internal error", stderr );
    }

    /* Subtract from adjusted original to obtain remainder */
    printf( "\tSUBS   a2, a2, a3, lsl #%d\n", loglsb );

    /* Apply corrections */
    printf( "\tADDPL  a1, a1, #1\n" );
    if( usinga4 )
    {
      printf( "\tADDMI  a2, a2, a4\n" );
    }
    else
    {
      printf( "\tADDMI  a2, a2, #%d\n", n );
    }

    /* Additional test required for divide-by-3, as result could be  */
    /* off by 2 lsb due to accumulated rounding errors.           */
    if( n == 3 )
    {
      printf( "\tCMP    a2, #3\n" );
      printf( "\tADDGE  a1, a1, #1\n" );
      printf( "\tSUBGE  a2, a2, #3\n" );
    }

    printf( "\tMOV    pc, lr\n\n" );
    printf( "\tEND\n" );
  }
```

```
int main( int argc, char *argv[] )
{
  if( argc != 2 )                  //包含divc程序在内需要两个参数，即divc n
  {
    printf( "Usage: divc <n>\n" );  //提示
    printf( "Generates optimal ARM code for divide-by-constant\n" );
    printf( "where <n> is one of (2^n-2^m) or (2^n+2^m) eg. 10\n" );
    //n必须为(2^n－2^m)和(2^n+2^m)
  }
  else
  {
    int num;
    num = atoi( argv[ 1 ] );       //调用atoi，将字符串转换成整型数
    if( num <= 1 )                 //如果输入的除数不是整数，将返回错误
    {
      fprintf( stderr, "%d is not sensible\n", num );
    }
    else
    {
      uint lsb = 1;
      /* find least-significant bit */
      while( ( num & lsb ) == 0 )      //按位与操作，找到最低非0位
      {
        lsb <<= 1;                     //左移一位
      }

      if( powerof2( num ) )
      //如果此数能够被2整除，则只需要进行相应的向右移位操作即可
      {
        fprintf( stderr, "%d is an easy case\n", num );
                                       //打印提示能够被2整除，退出
      }
      else if( powerof2( num + lsb ) )  //此数为2^n+2^m
      {                                 //也就是说，此数的二进制表示中只能有2个1
        divideby2( DIVIDE_BY_2_MINUS_1, num, lsb, num + lsb );
      }
      else if( powerof2( num - lsb ) )  //此数为2^n－2^m
      {                                 //也就是说，此数的二进制表示中1是连续的
        divideby2( DIVIDE_BY_2_PLUS_1, num, lsb, num - lsb );
                                       //输出除法汇编实现
      }
      else
      {
//如果不是2^n+2^m和2^n－2^m，则显示不能实现
        fprintf( stderr, "%d is not one of (2^n－2^m) or (2^n+2^m)\n", num );
                                       //输出除法汇编实现
      }
    }
  }
```

```
    return( 0 );
}
```

另外，在程序中调用了 C 函数库中的字符串转换成整型数函数 atoi，关于此函数说明如下。表头文件为：

```
#include<stdlib.h>
```

定义函数为：

```
int atoi(const char *nptr);
```

函数功能：atoi( )会扫描参数 nptr 字符串，跳过前面的空格字符，直到遇上数字或正、负符号才开始进行转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回。返回值为转换后的整型数。

4.3.3　产生及测试伪随机数字序列程序

1. 功能说明

使用汇编程序产生伪随机数字序列，在 random.s 文件中用汇编程序实现伪随机数字序列，random.c 为测试程序。

在很多应用中，产生随机序列很有必要，一个最有效的办法即是基于移位产生器来删除和反馈添加某些位。32 位的随机序列需要一个额外的反馈位来实现反馈。在此例子中，使用了一个 33 位的寄存器。算法如下：

(1) 新增加位数据＝ Bit 33 EOR Bit 20。

(2) 左移 33 位数字。

(3) 将新增加位数据加入到最后。

2. 测试程序

测试程序介绍如下。

(1) 编译程序：

```
E:\ptpress\embedded ARM\code\ch04\explasm>armasm random.s -o random.o
                                                                    //编译
E:\ptpress\embedded ARM\code\ch04\explasm>armcc -c randtest.c //编译
E:\ptpress\embedded ARM\code\ch04\explasm>armlink randtest.o random.o -o
randtest                                                            //链接
```

(2) 运行程序：

```
E:\ptpress\embedded ARM\code\ch04\explasm>armsd randtest      //测试

......
armsd: g                                    //运行程序，产生10个伪随机数字序列
randomnumber() returned fffff555
randomnumber() returned 00aaaaa0
randomnumber() returned 2aff57ff
randomnumber() returned e00055ff
randomnumber() returned f55fd5aa
```

```
    randomnumber() returned 07f54aaa
    randomnumber() returned 57500020
    randomnumber() returned 2baa02aa
    randomnumber() returned b5ffaa0a
    randomnumber() returned a05f7f00
    Program terminated normally at PC = 0x00009f80 (_sys_exit + 0x8)
    +0008 0x00009f80: 0xef123456  V4.. :    swi     0x123456
```

3. 源代码分析

源代码分析如下。

(1) 测试程序 randtest.c 源代码：

```
    //randtest.c
    #include <stdio.h>
    extern unsigned int randomnumber( void );      //引入汇编编写的函数
randomnumber( void )
    int main()
    {
      int loop;
      for( loop = 0; loop < 10; loop++ )           //产生10个伪随机序列
      { printf( "randomnumber() returned %08x\n", randomnumber() );
      }
      return( 0 );
    }
```

(2) 产生伪随机序列程序 random.s 源代码：

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

          AREA    |Random$$code|, CODE, READONLY
          EXPORT  randomnumber             //以便其他程序引用此函数
    randomnumber                           //标号
    ; on exit:
    ;      a1 = low 32-bits of pseudo-random number
    ;      a2 = high bit (if you want to know it)
          LDR     ip, |seedpointer|
          LDMIA   ip, {a1, a2}             ;加入数据到a1、a2
          TST     a2, a2, LSR#1            ;得到进位位数据
          MOVS    a3, a1, RRX              ;33位数据循环右移
          ADC     a2, a2, a2               ;将进位位加入到最低位
          EOR     a3, a3, a1, LSL#12       ;逻辑左移12位后逻辑异或操作
```

```
        EOR    a1, a3, a3, LSR#20        ;逻辑左移12位后逻辑异或操作
        STMIA   ip, {a1, a2}             ;恢复a1、a2
        MOV    pc, lr

|seedpointer|
        DCD    seed
        AREA   |Random$$data|, DATA     ;数据段
        EXPORT  seed
seed
        DCD    &55555555
        DCD    &55555555
        END
```

4.3.4　大端和小端存储转换程序

1. 功能说明

大端和小端存储转换程序主要实现一个 32 位字中的 4 个字符反转。ARM 程序有大端和小端的区别，两者不同，即是指在一个 32 位字中的 4 个字符存储顺序不一，此程序实现了大端到小端模式字符顺序转换的功能。读者可以根据此例程，自己编写程序实现。其功能实现如下：将原来序列为“ABCD”字符转换为“DCBA”字符。

2. 测试程序

测试程序介绍如下。

(1) 编译并运行程序。以下是此程序的运行情况：

```
E:\ptpress\embedded ARM\code\ch04\explasm>armcc bytedemo.c -o bytedemo
                                                                  //编译
E:\ptpress\embedded ARM\code\ch04\explasm>armsd bytedemo      //测试
......
armsd: g                                                          //输入g执行
←[2J      AMAZING FACTS ABOUT THE ARM - Reversing the bytes in a word

This function reverses the bytes in a word. The method was discovered in
1986 following a competition between ARM programmers; it requires just 4
instructions and 1 work register. A method using only 3 instructions per
word reversed was also found, but it has some set-up overhead and uses a
2nd register. Can you re-discover this method?

Later, the C compiler was 'taught' to generate exactly the instructions
needed, from C source. Check this claim using armcc -S -DREV byterev.c
and examining the assembly code file produced.

unsigned long reverse(unsigned long v)                        //C程序实现
   {
        unsigned long t;
        t = v ^ ((v << 16) | (v >> 16));   /*  EOR r1,r0,r0,ROR #16    */
        t &= ~0xff0000;                    /*  BIC r1,r1,#&ff0000    */
        v = (v << 24) | (v >> 8);          /*  MOV r0,r0,ROR #8    */
```

```
        return v ^ (t >> 8);                /*  EOR r0,r0,r1,LSR #8     */
    }

To see the method in action, press <Return>. Each time you press <Return>
one step of the reversal process will be executed. The values displayed
are symbolic, starting with the input word D C B A.
```

(2) 以下是实现 DCBA 序列转换为 ABCD 序列的过程：

```
Please press <Return> to proceed           //开始执行
←[2J       AMAZING FACTS ABOUT THE ARM - Reversing the bytes in a word

unsigned long reverse(unsigned long v)
{   unsigned long t;
    t = v ^ ((v << 16) | (v >> 16));       /*  EOR r1,r0,r0,ROR #16  */
    t &= ~0xff0000;                        /*  BIC r1,r1,#&ff0000    */
    v = (v << 24) | (v >> 8);              /*  MOV r0,r0,ROR #8     */
    return v ^ (t >> 8);                   /*  EOR r0,r0,r1,LSR #8   */
}
    v / r0              t / r1        original input in v/r0
   | D | C | B | A |     | x | x | x | x |

Please press <Return> to proceed           //执行第1步结果
//执行t = v ^ ((v<<16) | (v>>16)) 后
  | D | C | B | A |     |D^B|C^A|B^D|A^C|  //即执行EOR r1,r0,r0,ROR #16后

Please press <Return> to proceed           //第2步结果
//执行t &= ~0xff0000后
 | D | C | B | A |     |D^B| 0 |B^D|A^C|   //即执行BIC r1,r1,#&ff0000后

Please press <Return> to proceed           //第3步结果
//执行v = (v << 24) | (v >> 8)后
  | A | D | C | B |     |D^B| 0 |B^D|A^C|  //即执行MOV r0,r0,ROR #8后

Please press <Return> to proceed           //第4步结果
//执行  v = v ^ (t >> 8)
  | A | B | C | D |                          //即执行EOR r0,r0,r1,LSR #8
(original input was: D C B A)
......
```

3. 源代码分析

```
......
#include <stdio.h>
static void cls(void)          //打印字符，以方便显示
{
  printf("\033[2J");
}
static void up(int n)
{
  while (n > 0)
  { printf("\033[1A");  --n;    //打印字符，以方便显示
```

```
  }
}
static void pause(void)       //等待按回车键后继续执行
{
  fprintf(stderr, "Please press <Return> to proceed ");
  while( fgetc(stdin) != '\n' );
}

static void prologue(void)    //打印提示信息，这是对这个程序的解释
{
   printf("\
      AMAZING FACTS ABOUT THE ARM - Reversing the bytes in a word\n\
\n\
This function reverses the bytes in a word. The method was discovered in\n\
1986 following a competition between ARM programmers; it requires just 4\n\
instructions and 1 work register. A method using only 3 instructions per\n\
word reversed was also found, but it has some set-up overhead and uses a\n\
2nd register. Can you re-discover this method?\n\
\n\
Later, the C compiler was 'taught' to generate exactly the instructions\n\
needed, from C source. Check this claim using armcc -S -DREV byterev.c\n\
and examining the assembly code file produced.\n\
\n\
unsigned long reverse(unsigned long v)\n\
{  unsigned long t;\n\
   t = v ^ ((v << 16) | (v >> 16));   /*  EOR r1,r0,r0,ROR #16    */\n\
   t &= ~0xff0000;                    /*  BIC r1,r1,#&ff0000      */\n\
   v = (v << 24) | (v >> 8);          /*  MOV r0,r0,ROR #8        */\n\
   return v ^ (t >> 8);               /*  EOR r0,r0,r1,LSR #8     */\n\
}\n\
\n\
To see the method in action, press <Return>. Each time you press <Return>\n\
one step of the reversal process will be executed. The values displayed\n\
are symbolic, starting with the input word D C B A.\n\
\n");
}

int main()                            //测试主函数
{
   cls();
   prologue();
   pause();
   cls();
   prelude();
   show_state_1();                    //显示初始状态
   pause();
   up(13);
   show_state_2();                    //第1步后状态
   pause();
   up(11);
```

```
        show_state_3();                    //第2步后状态
        pause();
        up(11);
        show_state_4();                    //第3步后状态
        pause();
        up(11);
        show_state_5();                    //第4步后状态
        pause();
        fputc('\n', stdout);
        return( 0 );
    }
```

4.4　ARM 过程调用标准 APCS

4.4.1　APCS 基本概念

ARM 过程调用标准 APCS(ARM Procedure Call Standard)提供了紧凑编写程序的一种机制。如果开发一个基于 ARM 的系统，不要求必须实现 APCS。但如果要编写用来与编译后的 C 程序连接的汇编代码，则必须使用 APCS。APCS 定义以下内容。

- 对寄存器使用的限制。
- 使用栈的惯例。
- 在函数调用之间传递/返回参数。
- 可以被回溯的基于栈结构的格式，它用来提供从失败点到程序入口的函数(和给予的参数)的列表。

总的来说，有多个版本的 APCS (实际上是 16 个)。以下介绍在 RISC OS 上可能遇到的版本。

(1) APCS-A：即 APCS-Arthur。它已经被废弃，因为它有不同的寄存器定义。在此不再介绍。

(2) APCS-R：即 APCS-RISC OS。用于 RISC OS 应用程序在 USR 模式下进行操作以及在 SVC 模式下的模块/处理程序，包括以下显著特点。

- 寄存器定义：sl＝R10, fp＝R11, ip＝R12, sp＝R13, lr＝R14, pc＝R15。
- 它是唯一的最通用的 APCS 版本，所有编译的 C 程序都使用 APCS-R。
- 显式的栈限制检查。
- 26 位程序计数器。
- 不在 FP 寄存器中传递浮点实参。
- 不可重入。标志位必须被恢复。

(3) APCS-U：即 APCS-UNIX。它用于 RISCiX 应用程序(USR 模式)或内核(SVC 模式)。包括以下显著特点。

- 寄存器定义：sl＝R10、fp＝R11、ip＝R12、sp＝R13、lr＝R14、pc＝R15。
- 隐式的栈限制检查(使用 sl)。
- 26 位程序计数器。
- 不在 FP 寄存器中传递浮点实参。
- 不可重入。标志位必须被恢复。

(4) APCS-32：它是 APCS-2(-R 和-U)的一个扩展，允许 32 位程序计数器，并且从执行在 USR 模式下的一个函数中退出时，允许标志位不被恢复。其他概念等同于 APCS-R。

4.4.2　寄存器命名规则

APCS 对 ARM 处理器的 R0～R14 的寄存器进行了重命名，定义规则如表 4.1 所示。

表 4.1　寄存器定义规则

寄存器名字	APCS 命名	寄存器主要功能
R0	a1	工作寄存器，存储参数、结果和临时变量
R1	a2	工作寄存器，存储参数、结果和临时变量
R2	a3	工作寄存器，存储参数、结果和临时变量
R3	a4	工作寄存器，存储参数、结果和临时变量
R4	v1	工作寄存器，存储变量
R5	v2	工作寄存器，存储变量
R6	v3	工作寄存器，存储变量
R7	v4	工作寄存器，存储变量
R8	v5	在 ARM 状态，工作寄存器，存储变量
R9	v6(SB)	在 ARM 状态，存储重入，共享库变量
R10	SL(SL)	栈限制指针
R11	FP	桢指针
R12	IP	指令指针
R13	SP	栈指针
R14	LR	连接寄存器
R15	PC	程序计数器

R0～R3 用来在进行子程序调用时传递参数值，以及保存中间变量值。在 ARM 状态下，在进行子程序调用时，R12 也用来保存中间变量值，通过 R4～R11 用来存储本地变量值，在 Thumb 指令中只使用 v1～v4，寄存器 R12～R15 用做特殊用途。

4.4.3　函数调用参数传递及返回

APCS 在函数调用参数传递及返回时多使用 C 语言的惯例。在传递参数时：

- 前 4 个整数实参(或者更少)装载到 a1～a4；
- 前 4 个浮点实参(或者更少)装载到 f0～f3；
- 其他任何实参存储在内存中，即其余的参数被压入栈顶；

在函数退出时通过把返回连接值传送到程序计数器中来退出函数，并且：

- 如果函数返回一个小于等于一个字大小的值，则把这个值放置到 a1 中；
- 如果函数返回一个浮点值，则把它放入 f0 中；
- SP、FP、SL、v1～v6 和 f4～f7 应当被恢复为在进入函数时它所持有的值。

在函数退出的时候，把返回的连接值、返回的 SP 值和返回的 FP 值装载到 PC、SP 和

FP 中。

本节简要介绍了 APCS 的标准，关于 APCS 的详细说明，请读者参阅 ARM 公司的相关标准文献。

本章总结

在进行嵌入式软件开发中，大量采用了汇编程序与 C 语言的混合编程。本章详细介绍了如何使用 ADS 1.2 开发平台混合编写 ARM 汇编程序、C/C++语言程序。ADS 1.2 开发平台是 Windows 环境下开发 ARM 汇编程序及 C 语言程序通用的开发平台，本章详细介绍了该开发平台的基本功能，以及如何创建工程项目、添加文件、调试 ARM 程序。此外，对 ADS 平台下的编译工具、链接工具以及仿真调试工具进行了详细介绍。

在 ARM 应用程序中，如果仅需要少量汇编代码来实现相应的功能，在很多时候使用内嵌汇编的方式来添加汇编程序。本章 4.2 节以 4 个内嵌汇编实例为主线，重点介绍嵌套汇编程序的使用方法和使用技巧。混合编译以.c 为后缀的 C 语言程序和.s 为后缀的汇编程序应用也很广泛，本章 4.3 节以 4 个实例为主线，主要介绍 C 语言程序与汇编程序混合编程内容。

在进行嵌入式程序开发中，特别是开发混合编程时，一个需要特别关注的问题是 APCS 标准，在程序中尽可能地遵循该标准能够在很大程度上提高程序的可靠性。

读者完成本章的学习后，应该能够在 ADS 平台下熟练地编写 ARM 底层应用程序。

课后习题

1．试列举在进行嵌入式软件开发时，ARM 汇编语言多用来开发哪些类型的程序？

2．在使用 ADS 平台进行嵌入式应用程序开发时，armasm、armcc、armcpp、armlink 的主要功能是什么？如果要编译 Thumb 指令集程序，需要使用哪些命令？

3．试编写一段汇编程序，实现 64 位加法运算，并使用习题 2 中所列工具进行编译链接，生成可执行二进制代码。

4．在 ARM 应用程序中，在存储时可执行文件主要分为哪几个部分？在运行时又分成哪几个主要部分？各部分主要存储形式和存储内容是什么？

5. 使用 ADS 1.2 平台创建工程项目文件过程是什么？试安装 ADS 1.2 平台创建一个工程项目，然后添加相应源代码，使用 AXD 调试，写出详细操作步骤。

6．试编写一段 C 语言与汇编程序的混合程序，在此汇编程序中，实现输入的任意两个数的整数除法运算，在 C 文件中实现测试。

7．试编写一段 C 语言与汇编程序的混合程序，在此汇编程序中，实现两个 64 位数的乘法运算，在 C 文件中调用该函数实现测试，要求明确函数参数传递及返回。

8．为什么要尽可能地遵循 APCS 标准？有什么好处？在 APCS 中，函数调用及返回遵循哪些规则？