AT91SAM7S64 调试笔记

前言

如果您是一个单片机爱好者，当见到一款功能强大、性价比高的处理器时，一定会有一股很想掌握它、运用它的冲动，起码我是这样。5 年前第一次接触单片机（标准的 51 系列），就被它强大的功能所吸引，而痴迷于它，一直到今天。在这期间的不同时期，各种增强型 51，PIC、AVR，DSP 和 ARM 等不同程度的吸引和诱惑着我。有的已经玩过了，有的则没有，但很想玩的这股冲动一直存在心里，特别是对 ARM。记得 2002 年我就知道了 ARM 这个东东，眼睁睁地看着它一天天的火热，但由于对其开发工具及开发过程的不了解等原因一直只处于认识的阶段，尽管也曾玩过 ZLG 的 2104 开发板！可能是 ZLG 团队做的太出色的原因吧（详细的教材与源码，还有非常方便的工程模板），没过多久、没费多少劲就把里面的实验做完了，然后由于自已性格上的缺陷把它给扔在了一边，导致不到一个月就把大部分的东东还给了周老师。直到前段时间，让我有机会真真正正的玩起了 ARM！像最初玩 8051 那样的尽兴（出现问题时，吃不下饭、睡不着觉的那种痛苦和解决问题后的那种畅快）。

这篇文档就是记录了我在前段时间学习、调试目前最低价的 ARM 核处理器——AT91SAM7Sxx 时出现的问题与解决方法。它尽量完整地记录了我从开始不懂 ARM，到最终完成 AT91SAM7S64 各种外围实验的各个环节及整个过程，包括我在开始一个实验前的一些想法，实验目的，以及各个实验中，我以单片机的思维去思考时遇到的各种问题，和这些问题的解决方法。现将自己的一点经验以及体会拿出来与大家共同分享，一来是希望能够为那些在 ARM 门口徘徊迷茫的人提供一些借鉴，使他们顺利越过这道门槛；二来是希望能够抛砖引玉，以结识更多有共同爱好的朋友。由于我也是新手上路，文章中难免疏漏与错误，希望大家不吝指正，如果在调试 AT91SAM7Sxx 时有什么问题，欢迎大家共同讨论。

第一章　开发工具与调试环境

一．目标板

所用实验板是参考 ATMEL 公司官方网站上发布的《AT91SAM7S-EK Evaluation Board User Guide》设计的，相当于 AT91SAM7Sxx 评估测试板，主要用于各种外围实验，结构框图如图 1-1 所示。大家可以自己搭板子，或者直接购买现成的 AT91SAM7Sxx 评估板。

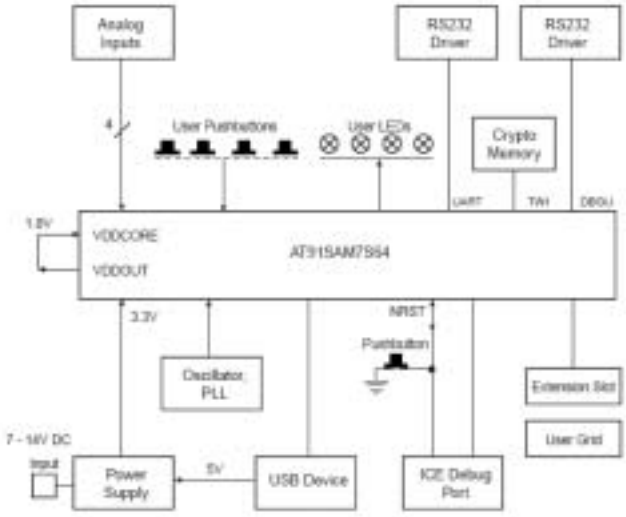


图 1-1. 结构框图

实验板主要以 AT91SAM7S64 微控制器为核心，外部扩展 4 路模拟量输入，4 个按钮和 4 个 LED，I2C 接口存储器，两个 UART 接口（其中一个可作程序下载的 DBUG 接口），1 个 USB 设备接口。

AT91SAM7S 系列微控制器是 ATMEL 公司最近推出的全球首个起步价低于 3 美元的 ARM7 内核闪存控制器，共有 AT91SAM7S32/64/128/256 四个型号，内部分别具有 32KB/64KB/128KB/256KB 的 Flash ROM 和 8KB/16KB/16KB/32KB 的 SRAM，无需扩展存储器 ;除 AT91SAM7S32 外，其它都集成了 USB 2.0 Device，另外还有 10 位的 ADC、12 路的 DMA、I2C、SPI、PWM、实时时钟等众多外围部件，功能强大，特别适合具有 8 位单片机基础转学 32 ARM 的用户。

二．开发工具

在 8 位单片机的开发过程中，都会用到诸集成调试环境和仿真器。同 8 位单片机一样，ARM 也有自己的集成开发环境和硬件访真器。本次实验就是使用 ADS1.2 集成开发环境和技创的 TecorICE 并口 JTAG 仿真器，使用时类似于 51 的仿真器直接挂接在 keil 下使用那样。

我个人认为无论是 ARM、DSP 还是 51 或 PIC 等，其开发工具和开发过程都是类似的。简单的讲都是先在集成开发环境中编辑用户程序，然后经过编译、连接产生目标文件，再通过硬件仿真器进行仿真调试。而对于普通 51 仿真器与 ARM 仿真器，使用起来并没明显的不同，主要的差别大概在于仿真接口（或者说是仿真头）。普通 51 仿真器是使用与目标单片机管脚兼容的仿真头替代目标单片机，用户程序是在仿真器内部的仿真芯片上运行。ARM 核处理器内置 ICE（仿真调试模块），该模块通过标准的 JTAG 接口引脚与 ARM 仿真器相连，此时 ARM 仿真器作为上位调试软件与 ARM 核芯片之间的协议转换器。用户的目标调试文件被下载到目标板上的存储器（可以是外部的或 ARM 处理器内部的存储器）中，通过控制目标芯片的仿真模块实现仿真调试。

第二章 我的第一个实验

——将程序执行到 C 文件的 main 函数

一．背景

当有了 techorICE 仿真器与 AT91SAM7S64 实验板后，像拿到一个不熟悉的普通单片机一样，希望能够写个小程序在里面跑跑，来开始第一步！在做这个实验前，我快速的阅读了 AT91SAM7S64 数据手册和《ARM 体系结构与编程》(我认为这本书非常不错)，还有 techorICE 仿真器的使用手册，并着重阅读了 ADS 软件、仿真器的使用以及起动代码方面的知识，因为这些都是马上要用到的。

二．实验目的

运用 ADS 编写一个小程序，使程序能够从起始的汇编代码运行到 C 程序的 main()函数（这也可称作非常简单的起动代码），并通过仿真器连接目标板，最终能够在 AT91SAM7S64 里正确运行。

三．实验程序和参数设置

1>连接器的选项设置

选项设置如图 2-1 所示。因为在 AT91SAM7S64 中 FLASH 存储器的地址是以 0x0 开始，而 SRAM 的地址是以 0x00200000 开始，所以我将下图中的 RO Base 和 RW Base 分别设置成了 0x0 和 0x00200000。

其它设置请参考有关书籍。

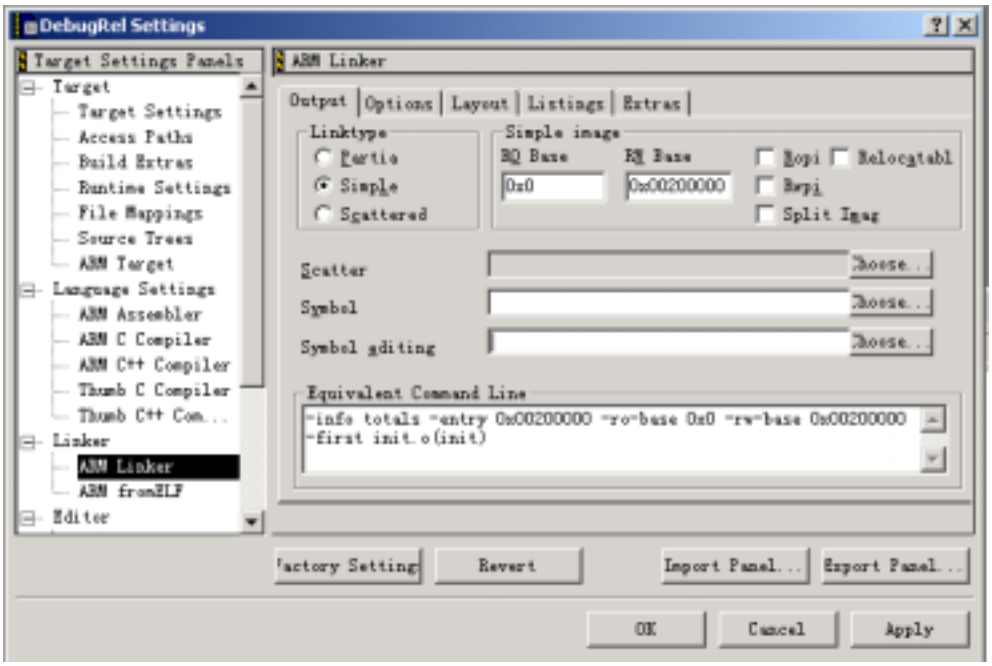


图 2-1. 选项设置图

2>启动代码

在 ARM 应用系统中，芯片复位后，在进入 C 语言的 main()函数前，都要执行一段启动代码。该代码一般都是用汇编语言编写，用来完成系统运行环境和应用程序的初始化，详情请参考有关书籍。由于本实验的目的很简单，就是想让程序复位后，进入 main()函数，所以有些初始化代码尽量精简，留下了下述代码。另外，__main 是 C 语言的内部库函数，可以在进入用户 main()之前完成内部 RAM 的初始化工作，类似 KeilC51 中的 startup.a51。当执行完__main 这段代码后，再跳转到 main()函数。

```
AREA init,CODE,READONLY
 CODE32
Mode_USR        EQU     0x10 ;CPSR 中各种处理器模式对应的控制位
I_Bit           EQU     0x80 ;CPSR 中的中断禁止位
F_Bit           EQU     0x40
USR_Stack       EQU     0x00203000      ;定义 RAM 的最高地址,无重映射
ENTRY
                B               InitReset           ; 0x00 Reset handler
undefvec        B               undefvec            ; 0x04 Undefined Instruction
swivec          B               swivec              ; 0x08 Software Interrupt
pabtvec         B               pabtvec             ; 0x0C Prefetch Abort
dabtvec         B               dabtvec             ; 0x10 Data Abort
rsvdvec         B               rsvdvec             ; 0x14 reserved
irqvec          B               irqvec                  ; 0x18 IRQ
fiqvec          B               fiqvec                  ; 0x1c FIQ
InitReset
 MSR CPSR_c,#Mode_USR | I_Bit | F_Bit ;改成用户模式且禁止 IRQ 和 FIQ 中断
 LDR SP,=USR_Stack
 IMPORT  __main
 b               __main  ;跳转到__main 执行，它位于 C 运行时库中
```

```
 END
```

3>C 语言主函数

在 C 语言主函数中做了一个死循环，如下述所示。

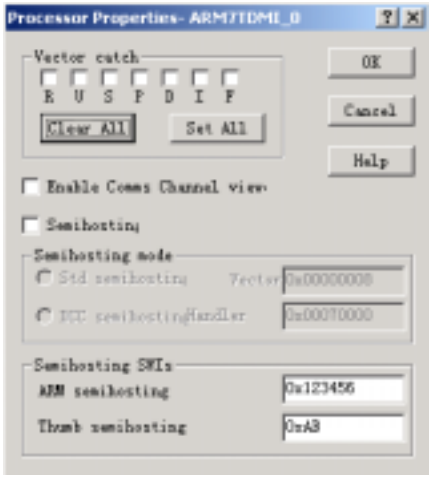
```
int main(void)
{
        while (1);
}
```

四．出现的问题与解决方法

当完成上述操作后，先用软件仿真，很快达到了目的，但将程序通过仿真器在目标板运行时出现了下述问题。

1> 当执行单步运行时，PC 一直停留在 0x0 处，而且 Debug Log 窗口中显示“ RDI Warning 00148: Can't set point ”。

原因是仿真器在 ROM 中设置的断点数是有限的，且单步运行时内部还要占用断点。可以使用“ Option->Config Processor ”打开“ Processor Properties-ARM7TDMI ”窗口，且按照下图设置以关断相应的断点。



2> 装载的代码与实践程序不一样

原因是由于程序没有装载到 AT91SAM7S64 的 FLASH ROM 里，在调试器中显示的是 FLASH ROM 中原先就有的程序。因为在连接器的选项设置中，将 RO Base 和 Image entry point 指向了 0 地址，而在 AT91SAM7S64 的这段空间为 FLASH ROM 区，而仿真器不能直接将代码下载到 FLASH ROM 里。用仿真器只能将代码下载到 AT91SAM7S64 的内部 SRAM 里进行调试，必须将 ARM Linker->Output->Simple image->RO Base 和 Image entry point 的 0，改成 SRAM 的地址 0x002000000。

3>在软件仿真的情况下，执行“ B __main ”指令，能使程序跳到 C 文件的 main 函数，但用硬件仿真时，还没执行到 main 函数时就进入了异常中断。

原因是执行“ B __main ”指令后，程序先跳到__main 库函数的入口，再进行一些初始化操作，最后再跳入用户的 main 函数。但在初始化过程中，由于堆栈或其它原因造成程序出错。有两种方法可以解决这个问题。第一：将“ B __main ”指令直接改成“ B main ”，使程序不进行初始化而直接跳入用户的 main()函数。第二：合理初始化堆栈。由于考虑到刚接触 ARM 和将问题简单化，我选择了第一种方法。

五．总结

1> 在用仿真器时，必须将程序下载到 AT91SAM7S64 的内部 SRAM 中，而不是 Flash ROM。

2> 从汇编代码进入 C 文件函数时，可以直接使用 C 语言中的标号（可参考书中混合编程部分），如执行“ B main ”则直接跳到 C 语言的 main()函数入口。

3> 在起动代码中，可以调用__main()库函数进行存储器的初始化，也可以自己编写更有效的代码进行初始化，在初始化后就可以使用“B __main ”指令直接跳转到C 的main()函数。

第三章 点亮我的 LED

——I/O 输出实验

一．背景

当完成上述实验后，我就可以像使用 51 单片机那样，在 C 文件的 main()函数中通过设置相应的寄存器来达到对相应外设（如 I/O 的输入、输出等）的控制目的。

二．实验目的

通过控制 PIO 的相关寄存器，使特定的 I/O 口输出高电平和低电平，来点亮 LED。

三．实验程序和参数设置

1> 连接器选项设置

RO Base = 0x00200000;

RW Base = 0x00202000;

Image entry point=0x00200000;

2> 启动代码

启动代码与第一个实验中修改后的相同，即将“ __main() ”改成 main()。

3> C 语言的代码

```
#include "AT91SAM7S64.h"        //特殊功能寄存器头文件。类似与 51 单片机中 reg51.h
#include "Board.h"              //定义目标板的头文件
int main(void)
{       *AT91C_PIOA_PER = LED_MASK;//使能 4 个 LED 对应管脚的 I/O 口功能
        *AT91C_PIOA_OER = LED_MASK;//使能 4 个 LED 管脚的输出功能
        while (1)
        {       //可用单步运行来查看输出结果
                *AT91C_PIOA_SODR = LED1;//将 LED1 对应的管脚输出高电平
                *AT91C_PIOA_CODR = LED1;//将 LED1 对应的管脚输出低电平
        }
}
```

四．总结

本实验非常的顺利，没有出现问题。证明 ARM 芯片的内部外围与 8 位单片机内部外围的控制或使用方法在原理上基本是相同的，都是通过设置相关的特殊功能寄存器来实现控制。也就是说只要会单片机也就会 ARM！

第四章 键盘输入

——I/O 输入实验

一．实验目的

能够正确读取 PIO 的管脚状态，实现当有按键按下时，LED 点亮，否则灭。

二．实验程序和参数设置

1>连接器选项设置和启动代码都与上个实验相同

2> C 语言的代码

```
#include "AT91SAM7S64.h" //特殊功能寄存器头文件。类似与 51 单片机中 reg51.h
#include "Board.h"                    //定义目标板的头文件
int main(void)
{ unsigned int Key_Val;    //定义变量，unsigned int 为 32 位，C51 是 16 位
  *AT91C_PIOA_PER = LED_MASK | SW_MASK;//使能 LED、KEY 脚 I/O 口功能
*AT91C_PIOA_ODR = SW_MASK;//使能 4 个 KEY 对应管脚的输入功能
  *AT91C_PIOA_OER = LED_MASK;//使能 4 个 LED 管脚的输出功能
      while (1)
      {
          Key_Val = *AT91C_PIOA_PDSR;//读管脚的电平状态
          if (Key_Val & SW1)
          {
            *AT91C_PIOA_SODR = LED1;//将状态反映给 LED
          }
          else
          {
           *AT91C_PIOA_COER = LED1;//按钮按下时，LED1 亮
          }
      }
 }
```

三．出现的问题与解决方法

1> 无论作输入用的 I/O 口电平如何变化，管脚状态寄存器（AT91C_PIOA_PDSR）的内容始终为 0，

即 I/O 口的输入功能没起作用。

原因是 AT91SAM7Sxx 内部集成了功率管理控制器，用它来控制所有外设的时钟以达到优化功耗的目的。所以只有使能了 PIO 的外围时钟，PIO 外设才会工作，才能读入输入管脚的状态。那么为什么 PIO 作为输出时不需要使能外围时钟呢？我个人认为这与内部外设在数字电路上的实现有关，输出功能只需要组合逻辑电路（不用时钟）就能实现，而输入功能则需要用到时序逻辑电路（需要时钟）才能实现。

因此，必须在 main()函数的开头增加如下两条时钟使能的语句：

```
*AT91C_PMC_SCER = AT91C_CKGR_MOSCEN;//使能系统时钟寄存器的处理器时钟
*AT91C_PMC_PCER = 1 << AT91C_ID_PIOA;     //使能 PIOA 外围时钟
```

四．总结

当要使 AT91SAM7S64 特定的外设工作时，必须使能相应外设的时钟控制寄存器。相反，如果不用相应的外设，禁能相应的时钟可以降低功耗。

第五章 模拟量输入

——A/D 转换实验

一．目的

通过 A/D 转换，能够正确读取外部模拟输入通道的电压值。

二．实验程序和参数设置

1>连接器选项设置和启动代码都与上个实验相同

2>C 语言的代码

```
#include "AT91SAM7S64.h"
#include "Board.h"
volatile unsigned int EXT_AD_Val[4];                //定义 A/D 转换结果值
int main(void)
{unsigned char i;
*AT91C_ADC_CR = AT91C_CKGR_MOSCEN;    //ADC 的软件复位,即清除 ADC 的所有寄存器
*AT91C_PMC_SCER = AT91C_CKGR_MOSCEN;//使能系统时钟寄存器的处理器时钟
*AT91C_PMC_PCER = 1 << AT91C_ID_ADC;      //使能 ADC 时钟
*AT91C_PIOA_PDR = EXT_AD0 | EXT_AD1;      //禁止管脚的 I/O 口功能,使作为模拟输入功能
*AT91C_ADC_MR    = 0x0f1f3f00;                //软件起动,10 位分辨率,128 分频
*AT91C_ADC_CHER = 0x33;                       //使能通道 0,1,4,5
*AT91C_ADC_CHDR = 0xcc;                       //禁能通道 2,3,6,7
*AT91C_ADC_IDR    = 0xfffff;                  //禁止所有 ADC 中断
while (1)
{      *AT91C_ADC_CR = 0x2;                   //起动转换
       while(1)
       {       if ((*AT91C_ADC_SR) & 0x33)          //等待转换结束
```

```
            {       EXT_AD_Val[0] = (*AT91C_ADC_CDR0) & 0x3ff;//读取 10Bit 的结果值
                    EXT_AD_Val[1] = (*AT91C_ADC_CDR1) & 0x3ff;
                    EXT_AD_Val[2] = (*AT91C_ADC_CDR4) & 0x3ff;
                    EXT_AD_Val[3] = (*AT91C_ADC_CDR5) & 0x3ff;
                    for (i = 0; i < 4; i++) EXT_AD_Val[i] = (EXT_AD_Val[i] * 3300) / 1023;
                    break;
            }
        }
    }
    }
```

三．总结

执行 ADC 的软件复位,将清除 ADC 的所有相关寄存器，因此必须在设置 ADC 相关寄存器之前执行。

第六章 RS232 串口通信

——串口 USART0 通讯实验

一．实验目的

利用串口调试软件能够正确接收到 AT91SAM7S64 发出的数据，AT91SAM7S64 也能正确接收到调试软件发出的数据。

二．实验程序和参数设置

1>连接器选项设置和启动代码都与上个实验相同

2>C 语言的代码

```
#include "AT91SAM7S64.h"
#include "Board.h"
unsigned char RBuff[256];                   //定义接收缓冲区
unsigned char index;
int main(void)
{
    unsigned int i, delay;
    *AT91C_CKGR_MOR = 0x701;   //使能主振荡器和设置起振时间
    *AT91C_PMC_MCKR = 0x01;    //选择 Mster Clock is main clock, divided by 0
    *AT91C_PMC_SCER = AT91C_CKGR_MOSCEN;//使能系统时钟寄存器的处理器时钟
    *AT91C_PMC_PCER = AT91C_ID_US0;      //使能 USART0 时钟
    *AT91C_PIOA_PDR = US_RXD_PIN | US_TXD_PIN;//禁止该两个管脚的 I/O 口功能
    *AT91C_PIOA_ASR= US_RXD_PIN | US_TXD_PIN;//将该两个 I/O 口分配给外围 A
    *AT91C_US1_MR =0x8c0;          //正常模式,时钟为 MCK,8 位长度,无校验,1 位停止位,
    *AT91C_US0_IDR    = 0xf3fff;    //禁止所有 UART 相关的中断
```

```
        *AT91C_US0_BRGR = 30;          //设置波特率为 38400Hz，AT91C_US0_BRGR 为 CD 值
        //Baudrate=SelectedClock/(8(2-Over)CD) = MCK/16CD = 18432000/(16*30) = 38400
        *AT91C_US0_CR     = 0x15c;     //复位接收器、发送器和状态位;使能接收与发送
        index = 0;
        while (1)
        {
            for (i = 0; i < 256; i++)
            {     //发送程序
                if ((*AT91C_US0_CSR) & AT91C_US_TXEMPTY)       //判断发送器是否为空
                {
                    *AT91C_US0_THR = i;       //空,则发送数据
                }
                for (delay = 0; delay < 0xfff; delay++);
            }
            if ((*AT91C_US0_CSR) & AT91C_US_RXRDY)
            {//接收程序，在调试该部分时，要将发送部分程序注示掉
                RBuff[index++] = *AT91C_US0_RHR;
            }
        }
    }
```

三．出现的问题与解决方法

1> 状态寄存器中的发送准备位（TXRDY）和发送空标志位（TXEMPTY）一直为 0，表示发送器未准备好和缓冲区不空。

原因是发送器复位后还未使能。不能同时进行发送器（或接收器）复位与使能操作（*AT91C_US0_CR= 0x15c），这样使能操作会无效，必须将它们分开，即先进行复位（*AT91C_US0_CR= 0x10c），再进行使能（*AT91C_US0_CR = 0x50）。

2> 串口接收、发送的数据不对

原因是系统主时钟和分频后的时钟计算错误，引起波特率也计算错误。很有必要深入研究关于时钟的产生、分频及波特率计算等内容。

3> 每次从串口调试软件收到的数据中，低四位正确，高四位错误。

原因是将“ *AT91C_US0_MR =0x8c0; ”写成了“ *AT91C_US1_MR =0x8c0; ”，而引起通讯模式根本不对。可以说这是一个非常低级的错误，但它却花费了我很久的时间才找到症结所在。在找原因的过程中，使我对串口相关的（如各种时钟的产生、波特率的计算等）内容有了更深刻的理解。

四．总结

在本实验中串口为异步模式，波特率的计算如下式所示：

$$Baudrate = SelectedClock/(8(2-Over)CD)$$

其中在 USART 模式寄存器（AT91C_US0_MR）中设置 SelectedClock 为 MCK；Over 为 1 则上式变成如下所示：

$$Baudrate = SelectedClock/(8(2-Over)CD) = MCK/16CD$$

在 Master Clock Register（AT91C_PMC_MCKR）中将 MCK 设置为 Main Clock 且不分频，

即为外部振荡时钟（接在 XIN 和 XOUT 管脚间的晶振）的频率，因为外部晶振是 18.432MHz, 所以 MCK 就为 18432000，则上式变成如下所示：

Baudrate=SelectedClock/(8(2-Over)CD) = MCK/16CD = 18432000/(16*30) = 38400

第七章 串口 DMA 控制实验

一．背景

ＤＭＡ是 Direct Memory Access 的缩写，即“存储器直接访问”。它是指一种高速的数据传输操作，允许在外部设备和存储器之间直接读写数据，即不通过ＣＰＵ，也不需要ＣＰＵ干预。整个数据传输操作在一个称为“ＤＭＡ控制器”的控制下进行的。ＣＰＵ除了在数据传输开始和结束时作一点处理外，在传输过程中ＣＰＵ可以进行其它的工作。这样，在大部分时间里，ＣＰＵ和输入输出都处在并行操作。因此，使整个计算机系统的效率大大提高。

AT91SAM7S64 串口外围 DAM 控制器的工作过程：将要发送的数据缓冲区的起始地址赋给串口 DMA 控制器的发送指针寄存器，再将要发送的字节个数赋给 PDC 的发送计数寄存器，然后无须 CPU 的干预，DMA 自动起动串口发送操作，发送完这些数据后又自动停止；同理，只要将接收数据缓冲区的起始地址赋给串口 DMA 控制器的接收指针寄存器，再将要接收的字节个数赋给 PCD 的接收计数值，DMA 将自动启动串口接收数据，接收完这些数据后，再通知 CPU。

二．实验目的

验证上述所描述的串口 DMA 控制器的工作过程，可用串口调试软件进行验证。

三．实验程序和参数设置

1>连接器选项设置和启动代码都与上个实验相同

2>C 语言的代码

```
#include "AT91SAM7S64.h"
#include "Board.h"
unsigned char RxBuff[256],TxBuff[256];
int main(void)
{
unsigned int i;
*AT91C_CKGR_MOR = 0x701;   //使能主振荡器和设置起振时间
*AT91C_PMC_MCKR = 0x01;//选择 Mster Clock is main clock, divided by 0
*AT91C_PMC_SCER = AT91C_CKGR_MOSCEN;//使能系统时钟寄存器的处理器时钟
*AT91C_PMC_PCER = AT91C_ID_US0;      //使能 USART0 时钟
*AT91C_PIOA_PDR = US_RXD_PIN | US_TXD_PIN;//禁止该两个管脚的 I/O 口功能
*AT91C_PIOA_ASR= US_RXD_PIN | US_TXD_PIN;//将该两个 I/O 口分配给外围 A
*AT91C_US0_CR       = 0x1ac;            //复位接收器和发送器,使能接收与发送,复位状态位
*AT91C_US1_MR =0x8c0;          //正常模式,时钟为 MCK,8 位长度,无校验,1 位停止位,
*AT91C_US0_IDR    = 0xf3fff;    //禁止所有 UART 相关的中断
*AT91C_US0_BRGR = 30;          //设置波特率为 38400Hz，AT91C_US0_BRGR 为 CD 值
```

```
*AT91C_US0_CR     = 0x50;       //使能发送与接收
*AT91C_US0_PTCR = AT91C_PDC_TXTEN | AT91C_PDC_RXTEN;//使能 US0 的 PDC  发送与接收
for (i = 0; i < 256; i++){              //给发送缓冲区覆值
        TxBuff[i] = i;
}       //下面可用单步执行，来观察现象
        *AT91C_US0_TPR = (unsigned int)TxBuff;//覆发送缓冲区起始地址
        *AT91C_US0_TCR = 256;                       //起动 PDC 发送 256 个字节
        *AT91C_US0_RPR = (unsigned int)RxBuff;//覆接收缓冲区起始地址
        *AT91C_US0_RCR = 256;                       //开始 PDC 接收
        while (1);
}
```

四．总结

我们在用 51 等单片机的串口进行收发数据时，因为发送与接收共用一个 Buffer，所以在发送一字节数据后，通常都要加“ while(!TI); ”语句，来等待数据发送完毕；在接收数据时都要使用中断来处理，每当接收到一个字节数据后都要中断一次 CPU。有了 DMA 这个功能，就不用这样浪费 CPU 的时间，可大大的提高 CPU 的实时性能。

第八章 中断控制实验

一．背景

实际上 ARM 的中断与 51 单片机的中断类似，都有类似的中断入口地址（ARM 称异常向量表）。只不过 51 给两个相互的中断入口之间留有足够的空间（如外部中断 0 的中断入口在 03H 处，而定时器 0 的中断入口在 0BH 处），在这段空间中可以放多条指令，这样在编写中断处理程序时非常灵活。ARM 总共有 7 种中断（或异常）类型，它们的入口分别为 00H、04H ------ 1CH，入口与入口之间只够放一条指令，这条通常为“ `B  XX` ”或者“ `LDR  PC, ResetAddr` ”的跳转指令。

二．实验目的

在 IRQ 中断向量地址（0x18）处设置一个断点后全速运行，用按钮产生 PIO 中断输入信号，使产生中断，而跳转到设置的断点处。

三．实验程序和参数设置

1> 连接器选项设置与上个实验相同

2> 启动代码

与前几个实验相比，使能了 IRQ 中断后再跳到 C 语言的主函数。

```
 AREA init,CODE,READONLY
 CODE32
Mode_USR EQU         0x10 ;CPSR 中各种处理器模式对应的控制位
```

```
USR_Stack EQU        0x00204000      ;定义 RAM 的最高地址,无重映射
ENTRY
                    B               InitReset               ; 0x00 Reset handler
Undefvec            B               undefvec                ; 0x04 Undefined Instruction
swivec              B               swivec                  ; 0x08 Software Interrupt
pabtvec             B               pabtvec                 ; 0x0C Prefetch Abort
dabtvec             B               dabtvec                 ; 0x10 Data Abort
rsvdvec             B               rsvdvec                 ; 0x14 reserved
irqvec              B               irqvec                      ; 0x18 IRQ
fiqvec                  B               fiqvec                  ; 0x1c FIQ
InitReset
 MSR CPSR_c,#Mode_USR               ;使能 FIQ,IRQ 中断
 LDR SP,=USR_Stack
 IMPORT         main
 b              main
 END
```

3> C 语言代码

```
#include "AT91SAM7S64.h"
#include "Board.h"
unsigned int Key_Val;
unsigned int key;
int main(void)
{
*AT91C_PMC_SCER = 0x1;                          //使能系统时钟寄存器的处理器时钟
 *AT91C_PMC_PCER = 1 << AT91C_ID_PIOA;    //使能 PIOA 外围时钟
 *AT91C_PIOA_PER = SW_MASK;                 //使能 KEY 引脚的 I/O 口功能
 *AT91C_AIC_IDCR = 1 << AT91C_ID_PIOA;    //禁止 PIO 外围中断功能
      *AT91C_PIOA_ODR = SW_MASK;            //使能 4 个 KEY 管脚的输入功能
 AT91C_BASE_AIC    ->    AIC_SMR[AT91C_ID_PIOA]    =    AT91C_AIC_PRIOR_HIGHEST    |
AT91C_AIC_SRCTYPE_INT_EDGE_TRIGGERED;//中断模式(优先级和触发模式)
 *AT91C_AIC_ICCR = 1 << AT91C_ID_PIOA;    //中断清除
 *AT91C_PIOA_IDR = 0xffffffff;                //禁止所有 PIO 口的中断功能
 *AT91C_PIOA_IER = SW3_MASK;                //使能 PIO 的 SW3 脚中断功能
 *AT91C_AIC_IECR = 1 << AT91C_ID_PIOA;    //使能 PIO 外围中断功能
 while (1);
}
```

四．出现的问题与解决方法

1> CPU 进不了中断，即跳不到 IRQ 中断向量入口地址。

原因是打开了 Memory 窗口,观察中断相关的寄存器。AXD 软件为了在 Memory 窗口中刷新这些寄存器值，在程序运行过程中会访问 CPU 中相应寄存器值。当中断源触发后，在跳到 IRQ 的中断入口之前，IRQ 的中断向量寄存器 AIC_IVR 就因为上述原因被读过 ，这时 CPU 就认为已经完成对 IRQ 中断的处理，因此

就不再跳转到 IRQ 中断入口。

2> 刚一执行“ MSR CPSR_c,#Mode_USR ”语句使能 IRQ 中断，CPU 就立即产生 IRQ 中断。

原因当上一次产生 IRQ 中断后，没有读 PIO 的中断状态寄存器，将其清零。因为中断状态寄存器置 1 时表示自从上一次读取此寄存器，至少检测到了一次电平变化。所以当没有读该寄存器时，该状态位会一直保持着。又因为在重新装载程序进行调试时，没有复位目标 CPU，所以当使能 IRQ 中断后，由于 PIO 中断状态寄存器为 1 的原因而产生中断。

五．总结

个人认为 ARM 的中断与 51 的中断，在本质上并没有多大的区别，出现上述的问题是由于它们在仿真、调试时的差异造成。在用普通的 51 仿真器进行仿真、调试时，如果我们不进行如单步、全速等执行程序运行，内部的各种寄存器、状态寄存器等是不会改变的，此时目标的 CPU 处于停止一样。而用 ARM 仿真器进行仿真、调试时，当你不进行如单步、全速等执行程序运行，内部的各种寄存器、状态寄存器还可能会改变，目标的 CPU 还会处处响应外部，这种情况在调试内部定时器时会更加明显。

第九章 地址重映射控制实验与重映射后的中断实验

一．背景

无论如何，ARM 的中断向量地址都是固定的，它总是位于 0x00 开始往下的 32 个字节，而这些代码位于 ROM 区。当用仿真器进行调试时，由于修改不了 ROM 区的内容，所以在调试中断时，就不能在中断向量表处执行相应的指令进入用户的中断处理程序入口。解决问题的方法就是进行地址重映射，将原来地址为 0x00200000 开始的 SRAM 重映射到以 0x0 开始处。

重映射功能的另外一个好处就是将原来在 ROM 区的用户程序映射到 RAM 区，因为 RAM 的读写速度要比 ROM 的读写速度快很多，所以在执行程序时就不会出现由于 ROM 的读写反应速度跟不上而使 CPU 处于等待状态，提高了 CPU 性能。这个功能我也没有试过，有兴趣的可以亲自试验一下。

二．实验目的

1>执行重映射命令后，使原来地址为 0x00200000 开始的 SRAM 重映射到以 0x0 开始处。将原来地址为 0x0 开始的 Flash ROM 重映射到以 0x00100000 开始处。可用 Memory 窗口观察 0x0 和 0x00100000 开始的内容查看是否实现上述功能。

2>当产生 IRQ 中断后，利用重映射的功能，使程序能够跳转到设定好了的用户中断处理程序入口（AT91F_Default_IRQ_handler）。

三．实验程序和参数设置

1>连接器选项设置与上个实验相同

2>启动代码

下面这段启动代码是在 ATMEL 官方网站上提供的启动代码基础上修改的，主要修改了地址重映射后的中断向量表的一些处理，将跳转指令“ B InitReset ”修改成“ LDR PC, ResetAddr ”，原因是 B 跳转指令被限制在±32Mb 的范围内。另外 AT91F_LowLevelInit 为 AT91SAM7S64 的一些底层的初始化，

再下部分说明。程序如下：

```
        AREA    startup,CODE,READONLY
        CODE32 ; Always ARM mode after reset
        ENTRY
reset
            LDR     PC, ResetAddr
            LDR     PC, UndefinedAddr
            LDR     PC, SWI_Addr
            LDR     PC, PrefetchAddr
            LDR     PC, DataAbortAddr
            NOP
            LDR     PC, IRQ_Addr
            LDR     PC, FIQ_Addr
ResetAddr           DCD     InitReset
UndefinedAddr       DCD     undefvec
SWI_Addr            DCD     swivec
PrefetchAddr        DCD     pabtvec
DataAbortAddr       DCD     dabtvec
Nouse               DCD     0
IRQ_Addr            DCD     IRQ_Handler_Entry
FIQ_Addr            DCD     FIQ_Handler_Entry
undefvec
                B           undefvec            ; 0x04 Undefined Instruction
swivec
                B           swivec              ; 0x08 Software Interrupt
pabtvec
                B           pabtvec             ; 0x0C Prefetch Abort
dabtvec
                B           dabtvec             ; 0x10 Data Abort
;--------------------------------------------------------------------------
;- Function          : FIQ_Handler_Entry
;- Treatments        : FIQ Controller Interrupt Handler.
;- Called Functions  : AIC_FVR[interrupt]
;--------------------------------------------------------------------------
AIC_FVR         EQU     260
FIQ_Handler_Entry
;- Switch in SVC/User Mode to allow User Stack access for C code
; because the FIQ is not yet acknowledged
;- Save and r0 in FIQ_Register
            mov         r9,r0
            ldr         r0 , [r8, #AIC_FVR]
            msr         CPSR_c,#I_BIT | F_BIT | ARM_MODE_SVC
;- Save scratch/used registers and LR in User Stack
            stmfd       sp!, { r1-r3, r12, lr}
```

```
;- Branch to the routine pointed by the AIC_FVR
                mov             r14, pc
                bx              r0
;- Restore scratch/used registers and LR from User Stack
                ldmia           sp!, { r1-r3, r12, lr}
;- Leave Interrupts disabled and switch back in FIQ mode
                msr             CPSR_c, #I_BIT | F_BIT | ARM_MODE_FIQ
;- Restore the R0 ARM_MODE_SVC register
                mov             r0,r9
;- Restore the Program Counter using the LR_fiq directly in the PC
                subs            pc,lr,#4
InitReset
;------------------------------------------------------------------------------
;- Low level Init (PMC, AIC, ? ....) by C function AT91F_LowLevelInit
;------------------------------------------------------------------------------
                EXTERN      AT91F_LowLevelInit
__iramend       EQU             0x00203FF0
;- minumum C initialization
;- call    AT91F_LowLevelInit( void)
                ldr         r13,=__iramend                  ; temporary stack in internal RAM
;--Call Low level init function in ABSOLUTE through the Interworking
            ldr         r0,=AT91F_LowLevelInit
                mov         lr, pc
            bx          r0
;------------------------------------------------------------------------------
;- Stack Sizes Definition
;------------------------
;- Interrupt Stack requires 2 words x 8 priority level x 4 bytes when using
;- the vectoring. This assume that the IRQ management.
;- The Interrupt Stack must be adjusted depending on the interrupt handlers.
;- Fast Interrupt not requires stack If in your application it required you must
;- be definehere.
;- The System stack size is not defined and is limited by the free internal
;- SRAM.
;------------------------------------------------------------------------------
;------------------------------------------------------------------------------
;- Top of Stack Definition
;-------------------------
;- Interrupt and Supervisor Stack are located at the top of internal memory in
;- order to speed the exception handling context saving and restoring.
;- ARM_MODE_SVC (Application, C) Stack is located at the top of the external memory.
;------------------------------------------------------------------------------
IRQ_STACK_SIZE              EQU         (2*8*4)         ; 2 words per interrupt priority level
ARM_MODE_FIQ                EQU         0x11
```

```
ARM_MODE_IRQ                    EQU         0x12
ARM_MODE_SVC                    EQU         0x13
I_BIT                           EQU         0x80
F_BIT                           EQU         0x40
AT91C_BASE_AIC                  EQU         0xFFFFF000
;---------------------------------------------------------------------------
;- Setup the stack for each mode
;-------------------------------
                        ldr         r0, =__iramend
;- Set up Fast Interrupt Mode and set FIQ Mode Stack
                        msr         CPSR_c, #ARM_MODE_FIQ | I_BIT | F_BIT
;- Init the FIQ register
                        ldr         r8, =AT91C_BASE_AIC
;- Set up Interrupt Mode and set IRQ Mode Stack
                        msr         CPSR_c, #ARM_MODE_IRQ | I_BIT | F_BIT
                        mov         r13, r0                         ; Init stack IRQ
                        sub         r0, r0, #IRQ_STACK_SIZE
;- Enable interrupt & Set up Supervisor Mode and set Supervisor Mode Stack
                        msr         CPSR_c, #ARM_MODE_SVC
                        mov         r13, r0
;---------------------------------------------------------------
; ?CSTARTUP
;---------------------------------------------------------------
;               EXTERN__segment_init
                EXTERNmain
; Initialize segments.
; __segment_init is assumed to use
; instruction set and to be reachable by BL from the ICODE segment
; (it is safest to link them in segment ICODE).
;               ldr   r0,=__segment_init
;                       mov         lr, pc
;               b     r0
                EXPORT __main
jump_to_main
                ldr   lr,=call_exit
                ldr   r0,=main
__main
                bx    r0
;---------------------------------------------------------------------------
;- Loop for ever
;---------------
;- End of application. Normally, never occur.
;- Could jump on Software Reset ( B 0x0 ).
;---------------------------------------------------------------------------
```

```
call_exit
End
                b           End
;--------------------------------------------------------------------------------
;- Manage exception
;---------------
;- This module The exception must be ensure in ARM mode
;--------------------------------------------------------------------------------
;--------------------------------------------------------------------------------
;- Function                  : IRQ_Handler_Entry
;- Treatments                : IRQ Controller Interrupt Handler.
;- Called Functions          : AIC_IVR[interrupt]
;--------------------------------------------------------------------------------
AIC_IVR     EQU         256
AIC_EOICR   EQU         304
IRQ_Handler_Entry
;- Manage Exception Entry
;- Adjust and save LR_irq in IRQ stack
                sub             lr, lr, #4
                stmfd           sp!, {lr}
;- Save and r0 in IRQ stack
                stmfd           sp!, {r0}
;- Write in the IVR to support Protect Mode
;- No effect in Normal Mode
;- De-assert the NIRQ and clear the source in Protect Mode
                ldr             r14, =AT91C_BASE_AIC
            ldr             r0 , [r14, #AIC_IVR]
            str             r14, [r14, #AIC_IVR]
;- Enable Interrupt and Switch in Supervisor Mode
                msr             CPSR_c, #ARM_MODE_SVC
;- Save scratch/used registers and LR in User Stack
                stmfd           sp!, { r1-r3, r12, r14}
;- Branch to the routine pointed by the AIC_IVR
                mov             r14, pc
                bx              r0
;- Restore scratch/used registers and LR from User Stack
                ldmia           sp!, { r1-r3, r12, r14}
;- Disable Interrupt and switch back in IRQ mode
                msr             CPSR_c, #I_BIT | ARM_MODE_IRQ
;- Mark the End of Interrupt on the AIC
                ldr             r14, =AT91C_BASE_AIC
                str             r14, [r14, #AIC_EOICR]
;- Restore SPSR_irq and r0 from IRQ stack
                ldmia           sp!, {r0}
```

```
;- Restore adjusted    LR_irq from IRQ stack directly in the PC
                ldmia           sp!, {pc}^
;---------------------------------------------------------------
; ?EXEPTION_VECTOR
; This module is only linked if needed for closing files.
;---------------------------------------------------------------
        EXPORT AT91F_Default_FIQ_handler
        EXPORT AT91F_Default_IRQ_handler
        EXPORT AT91F_Spurious_handler
        CODE32 ; Always ARM mode after exeption
AT91F_Default_FIQ_handler
                b       AT91F_Default_FIQ_handler
AT91F_Default_IRQ_handler
                b       AT91F_Default_IRQ_handler
AT91F_Spurious_handler
                b       AT91F_Spurious_handler
;       ENDMOD
        END
```

3> 底层初始化函数——AT91F_LowLevelInit

在启动代码里，还调用了底层初始化函数，主要完成系统、外围时钟的设置，看门狗的禁能，中断入口地址的初始设置等。

```
#include "Board.h"
extern void AT91F_Spurious_handler(void);          //在启动代码中的中断服务程序
extern void AT91F_Default_IRQ_handler(void);
extern void AT91F_Default_FIQ_handler(void);
void AT91F_LowLevelInit( void)
{
  int                   i;
  AT91PS_PMC        pPMC = AT91C_BASE_PMC;
  AT91C_BASE_MC->MC_FMR = ((AT91C_MC_FMCN)&(50 <<16)) | AT91C_MC_FWS_1FWS ;
  AT91C_BASE_WDTC->WDTC_WDMR= AT91C_SYSC_WDDIS;       //* Watchdog Disable
 //* Set MCK at 47 923 200
  pPMC->PMC_MOR = (( AT91C_CKGR_OSCOUNT & (0x06 <<8) | AT91C_CKGR_MOSCEN ));
  while(!(pPMC->PMC_SR & AT91C_PMC_MOSCS));
          pPMC->PMC_PLLR = ((AT91C_CKGR_DIV & 0x05) |
                                  (AT91C_CKGR_PLLCOUNT & (16<<8)) |
                                  (AT91C_CKGR_MUL & (25<<16)));
            while(!(pPMC->PMC_SR & AT91C_PMC_LOCK));
             pPMC->PMC_MCKR = AT91C_PMC_CSS_PLL_CLK | AT91C_PMC_PRES_CLK_2 ;
        // Set up the default interrupts handler vectors
        AT91C_BASE_AIC->AIC_SVR[0] = (int) AT91F_Default_FIQ_handler ;
        for (i=1;i < 31; i++)
        {
```

```
            AT91C_BASE_AIC->AIC_SVR[i] = (int) AT91F_Default_IRQ_handler ;
        }
        AT91C_BASE_AIC->AIC_SPU    = (int) AT91F_Spurious_handler ;
        i = *AT91C_PIOA_ISR;                        //读中断状态寄存器,用于清楚该中断标志位
    }
```

4> C 语言主函数

```
#include "Board.h"
int main()
{
        AT91_REG Key_Val;
        *AT91C_PMC_SCER = 0x1;                      //使能系统时钟寄存器的处理器时钟
        *AT91C_PMC_PCER = 1 << AT91C_ID_PIOA;       //使能所有外围时钟
        *AT91C_PIOA_PER = SW_MASK;                  //使能 KEY 脚 I/O 口功能
        *AT91C_PIOA_ODR = SW_MASK;                  //使能 4 个 KEY 管脚的输入功能
        *AT91C_AIC_IDCR = 1 << AT91C_ID_PIOA;       //PIO 外围中断禁止
        AT91C_BASE_AIC  ->  AIC_SMR[AT91C_ID_PIOA]  =  AT91C_AIC_PRIOR_HIGHEST  |
AT91C_AIC_SRCTYPE_INT_EDGE_TRIGGERED;       //中断模式(优先级和触发模式)
        *AT91C_AIC_ICCR = 1 << AT91C_ID_PIOA;       //中断清除
        *AT91C_PIOA_IDR = 0xffffffff;               //禁止所有 PIO 口中断
        Key_Val = *AT91C_PIOA_ISR;                  //读中断状态寄存器,用于清楚该中断标志位
        *AT91C_PIOA_IER = SW3_MASK;                 //使能 PIO 的 SW3 脚中断功能
        *AT91C_AIC_IECR = 1 << AT91C_ID_PIOA;       //使能 PIO 外围中断功能
        *AT91C_MC_RCR = AT91C_MC_RCB;               //执行地址重映射
        while (1);
}
```

四．出现的问题与解决方法

1> 在调试过程中，（特别是全速运行）有时会进入 Undefined Instruction 及 Data Abort 等异常。

原因是由地址重映射命令引起。有时在重新装载程序而没有复位目标 CPU 时（此时目标 CPU 仍处于重映射状态），再次执行了重映射命令。

五．总结

1> 当执行地址重映射命令，使进入重映射后，如果再次执行该命令会使它恢复重映射前的状态。

第十章 I^2C 接口实验

注：在 AT91SAM7Sxx 系列中，I^2C 称作 TWI。

一．实验目的

能够正确读写 I^2C 接口芯片存储器（24C02），即写入 24C02 的数据与读出来的数据相同。

二．实验程序和参数设置

1> 连接器选项设置和启动代码与上一个实验相同

2> I^2C 驱动程序

ATMEL 官方网站上有这方便的参考程序。主要由 I^2C 接口的初始化、I^2C 的读和写三部分组成。

```
#include "board.h"
#include "twi.h"
void InitTwi(void)
{       AT91F_TWI_CfgPIO();                                     // 配 置 TWI 的 TWD 和 TWCK 管 脚
        AT91F_PIO_CfgOpendrain(AT91C_BASE_PIOA,(unsigned int)AT91C_PA3_TWD);
        AT91F_TWI_CfgPMC ();                                    //使能 TWI 外围时钟
        AT91F_TWI_Configure (AT91C_BASE_TWI);                   //将 TWI 设置成主模式
        AT91F_SetTwiClock(AT91C_BASE_TWI);                      //计算、设置时钟发生寄存器
}
//*----------------------------------------------------------------------------
//* \fn       AT91F_SetTwiClock
//*计算、设置 TWI 时钟发生寄存器
//*----------------------------------------------------------------------------
void AT91F_SetTwiClock(const AT91PS_TWI pTwi)
{       int sclock;
        sclock = (10*MCK /AT91C_TWI_CLOCK);
        sclock = (MCK /AT91C_TWI_CLOCK);
        if (sclock % 10 >= 5)
                sclock = (sclock /10) - 5;
        else
        sclock = (sclock /10)- 6;
        sclock = (sclock + (4 - sclock %4)) >> 2;        // div 4
       pTwi->TWI_CWGR      = 0x00010000 | sclock | (sclock << 8);
}
//*----------------------------------------------------------------------------
//* \fn       AT91F_TWI_Write
//* \brief Send n bytes to a slave device
//*----------------------------------------------------------------------------
int AT91F_TWI_Write(const AT91PS_TWI pTwi ,int address, char *data2send, int size)
{       unsigned int status;
        pTwi->TWI_MMR=(AT91C_EEPROM_I2C_ADDRESS|       AT91C_TWI_IADRSZ_1_BYTE       )       &
~AT91C_TWI_MREAD;
        pTwi->TWI_IADR = address; // Set TWI Internal Address Register
        status = pTwi->TWI_SR;
```

```
        pTwi->TWI_THR = *(data2send++);
        pTwi->TWI_CR = AT91C_TWI_START;
        while (size-- >1){        // Wait THR Holding register to be empty
                while (!(pTwi->TWI_SR & AT91C_TWI_TXRDY));
                pTwi->TWI_THR = *(data2send++);// Send first byte
        }
        pTwi->TWI_CR = AT91C_TWI_STOP;
        status = pTwi->TWI_SR;
        while (!(pTwi->TWI_SR & AT91C_TWI_TXCOMP)); // Wait transfer is finished
        return AT91C_EEPROM_WRITE_OK;
}
//*-----------------------------------------------------------------------------
//* \fn       AT91F_TWI_Read
//* \brief Read n bytes from a slave device
//*-----------------------------------------------------------------------------
int AT91F_TWI_Read(const AT91PS_TWI pTwi , int address, char *data2rec, int size)
{       unsigned int status;
        pTwi->TWI_MMR=(AT91C_EEPROM_I2C_ADDRESS|AT91C_TWI_IADRSZ_1_BYTE)                |
AT91C_TWI_MREAD;        // Set the TWI Master Mode Register
        pTwi->TWI_IADR = address; // Set TWI Internal Address Register
        pTwi->TWI_CR = AT91C_TWI_START; // Start transfer
        status = pTwi->TWI_SR;
        while (size-- >1){ // Wait RHR Holding register is full
                while (!(pTwi->TWI_SR & AT91C_TWI_RXRDY));
                *(data2rec++) = pTwi->TWI_RHR; // Read byte
        }
        pTwi->TWI_CR = AT91C_TWI_STOP;
        status = pTwi->TWI_SR;
       while (!(pTwi->TWI_SR & AT91C_TWI_TXCOMP)); // Wait transfer is finished
        *data2rec = pTwi->TWI_RHR; // Read last byte
        return AT91C_EEPROM_READ_OK;
}
```

3> 主函数

在主函数中，首先调用了 TWI 的初始化函数，然后进入了超级大循环。在每次写和读时，都对它们的数据缓冲区进行初始化，以查看读、写的正确性。

```
#include "board.h"
#include "twi.h"
int main ( void )
{       int loop,index=0;
        char Wri_data[16], Red_data[16];          //定义写和读缓冲区
        InitTwi();                                //TWI 初始化
        while (1)
        {       for (loop = 0; loop < 16; loop++){ //初始化写和读缓冲区的内容
                        data1[loop] = loop + index;
```

```
            data2[loop] = 0;
        }
        index += 1;
        AT91F_TWI_Write(AT91C_BASE_TWI, 0x0, Wri_data, EEP_RW_CHK_CNT);//写
        AT91F_TWI_Read(AT91C_BASE_TWI, 0x0, Red_data, EEP_RW_CHK_CNT);//读
    }//用单步调试，可比较 Wri_data[16]和 Red_data[16]的内容来判断读写是否正确。
}
```

三．出现的问题与解决方法

1> 无论写入任何数据，读出来都是同样的数，表明数据没有写入（在调试时对其写操作，器件没有产生应答）。

原因是 24C02 的写保护管脚没有接地，内部的数据被写保护了。注意：有些厂家的 EEROM 的该管脚处于悬空时为不保护状态，而有些厂家的 EEROM 会处于保护状态，因此在用之前一定要仔细阅读厂家的数据手册，或不要将该脚悬空。

2> 对 24WC02 写的数据和读的数据不一样。

原因是 I^2C 的时钟太快。在本实验程序中，可以减少 twi.h 中的 AT91C_TWI_CLOCK 常量的数值。或者直接在程序中修改 TWI 时钟波形发生寄存器 TWI_CWGR。

3> 当写入 16 字节数据，再读出 16 字节数据时，最后一个字节总为 0。

原因是 TWI Master Mode Register 的 IADRSZ（器件内部地址长度）设成了两个字节（Two-byte），要将改成一个字节（One-byte）。

第十一章 USB 设备实验

一．背景

在 ATMEL 官方网站上提供了 USB 的应用例子（详情请参考“ BasicUSB Application ”说明），里面有源代码（是用 IAR 编译的，需要稍作修改才能用在 ADS 上），两个不同的 USB 驱动程序。两个不同的 USB 驱动程序，在 PC 机上是两个不同的应用例子。当安装完两个不同的驱动后，一个出现的是调制解调器的设备，可以用超级终端来完成 USB 数据的收发。另一个是 USB 设备，用 ATMEL 提供的‘ BasicUSB_6124.exe ”来完成 USB 数据的收发。我起初一直在用后面的例子来做实验，但试了很久都没有成功，后来改用前面的成功了。

二．USB 驱动安装说明

当第一次与 host PC 机连接时，系统会弹出一个“ 找到新的硬件向导 ”窗口，选择“ 从列表或指定位置安装 ”后点击“ 下一步 ”。在接下来的窗口中选择“ 不要搜索。我要自己选择要安装的驱动程序 ”，然后点击“ 下一步 ”。再在接下来的窗口中点击“ 从磁盘安装 ”，找到“ atm6124ser.inf ”所在的目录后打开。再点击“ 下一步 ”开始安装，最后点“ 完成 ”就可以。

安装完成后，会在设备管理器的“ 调制解调器 ”栏中查看到“ ATMEL AT91 USB serial emulation #2 ”设备。此时就可以使用“ 超级终端 ”通过 USB 与 AT91SAM7S64 通讯了。详请请查看该目录的“ BasicUSB Application.pdf ”文件。

三．实验目的

用 PC 机上的超级终端发送数据，AT91SAM7S64 通过 USB 接收超级终端上发过来的数据（USB 的读操作），再通过 USB 将接收到的数据返回给 PC 机上的超级终端（USB 的写操作），这样就完成了 PC 机与 AT91SAM7S64 的 USB 通讯。如果 AT91SAM7S64 将接收到数据通过 DBUG 串口再发送出去，这样 AT91SAM7S64 就可以当作一个 USB 转串口的设备使用。

四．实验源程序

由于这个实验的源程序比较多，且 ATMEL 提供了详细的文档和源代码，故不再列出。有什么问题可以通过 ccn422@hotmail.com 联系我。

另外因为 ATMEL 的源码中没有使能重映射功能，如果想将程序下载到地址为 0x00200000 的 SRAM 中调试，又要使用在 0x0 地址空间的中断向量表，使程序跳转到中断处理程序，必须使用重映射的功能。所以我在起动代码（AT91F_LowLevelInit）中加了重映射这条命令，如下：

```
if (Remap_Flag == 0)
{ //由于不能重复执行重映射命令,所以加了 Remap_Flag 标志来避免多次执行重映射命令
    Remap_Flag = 0xff;
    *AT91C_MC_RCR = AT91C_MC_RCB;                    //Remap Command Bit
}
```

五．出现的问题与解决方法

1> 每次重新装载程序进行调试时，Remap_Flag 没有等于 0。

原因是进入 C 代码前，没有初始化存储器的内容。用户可以将起动代码中的“ b main ”改成“ b __main ”，即在跳入到 main()函数前执行__main 初始化库函数，也可自己加入初始化存储器的代码。程序如下：

```
InitReset
    IMPORT |Image$$RO$$Limit|   ; End of ROM code (=start of ROM data)
    IMPORT |Image$$RW$$Base|    ; Base of RAM to initialise
    IMPORT |Image$$ZI$$Base|    ; Base and limit of area
    IMPORT |Image$$ZI$$Limit|   ; to zero initialise
    ;*************************************************************
    ;*    Copy and paste RW data/zero initialized data           *
    ;*************************************************************
    LDR         r0, =|Image$$RO$$Limit|    ; Get pointer to ROM data
    LDR         r1, =|Image$$RW$$Base|     ; and RAM copy
    LDR         r3, =|Image$$ZI$$Base|
    ;Zero init base => top of initialised data

    CMP         r0, r1          ; Check that they are different
    BEQ         %F1
0
    CMP         r1, r3          ; Copy init data
```

```
    LDRCC     r2, [r0], #4       ;--> LDRCC r2, [r0] + ADD r0, r0, #4
    STRCC     r2, [r1], #4       ;--> STRCC r2, [r1] + ADD r1, r1, #4
    BCC       %B0
1
    LDR       r1, =|Image$$ZI$$Limit| ; Top of zero init segment
    MOV       r2, #0
2
    CMP       r3, r1             ; Zero init
    STRCC     r2, [r3], #4
    BCC       %B2
```

如果想详细了解该部分的原理，可以参考三星公司为 44B0 写的起动代码部分资料。

2> 这个实验浪费了很多的时间，原因就是因为我一直用的 atm6124.sys 和 atm6124.inf 驱动程序以及 BasicUSB_6124.exe 来做此实验，当执行 BasicUSB_6124.exe 后，总是出现“ 设备不能连接 ”。为此我专门去书店买了《USB2.0 应用与设计》参考，再研究和调试源代码，再加上仔细的阅读‘ BasicUSB Application.pdf ”文件，后来改用 atm6124ser.inf 驱动程序与超级终端来完成了本次实验。

六．总结

如果一开始就很成功的完成了本次实验，我可能对 USB 的原理等不会了解到多少，正是因为出现了问题，我才会为了解决问题而去找相关的资料学习，最终到问题的解决。因此出现问题并不是一件什么坏事，相反，问题的出现会引导我们静下心来向更深层次去探究，最终更深刻更全面地掌握知识。

第十二章 ISP 实验

一．背景

由于前面的实验都是用仿真器将代码下载到 AT91SAM7S64 的 SRAM 里调试的，还不能在实际的 Flash ROM 里跑。所以在这个实验中，我们将通过 ATMEL 提供的 SAM-BA 软件和 AT91SAM7S64 自带的 ROMBoot 功能，完成 AT91SAM7S64 的 Flash ROOM 的在线烧写。

二．实验目的

用前面“ I/O 口输入实验 ”的源程序生成二进制文件，下载到 AT91SAM7S64 的 Flash ROM 中，且能脱机正确运行。

三．操作方法

1> 安装。双击 Install SAM-BA.exe 文件运行，按提示一步步安装即可。
2> 连接好硬件，且使 AT91SAM7S64 处于 RomBoot 状态。
3> 运行。双击 SAM-BA 图标出现如下图所示的启动窗口：

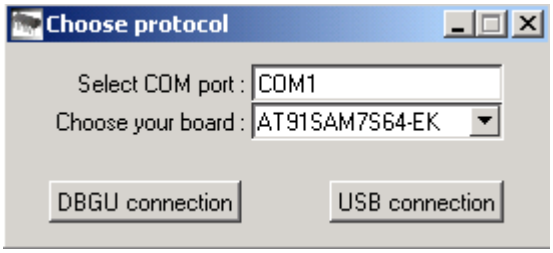


图 4.21 SAM 启动窗口

当正确设置后，按“ DBGU connection ”按钮将打开 SAM-BA 编程环境，如下图所示：

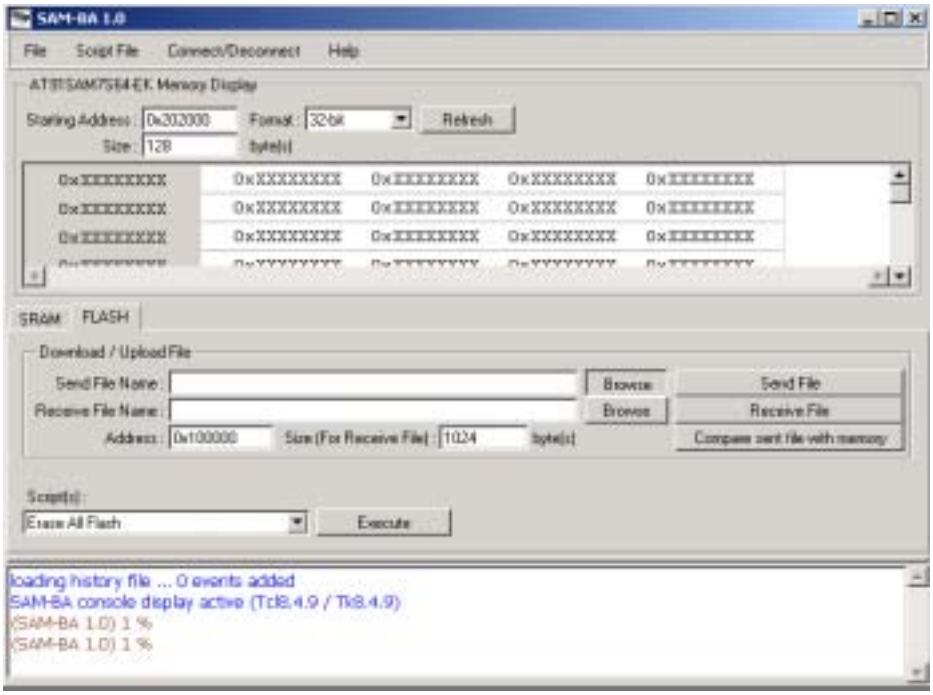


图 4.22　SAM-BA1.0 编程环境

4>　用 ADS1.2 生成二进制文件。

5>　编程。点击“ Flash->Download/Upload File->Send File Name ”项的“ Browse ”按钮打开二进制文件后，点击“ Send File ”按钮开始下载程序。

6>　复位目标板，开始运行用户程序。

四．出现的问题与解决方法

1>　ADS 软件编译后不能产生二进制等目标文件

第一、　可以用 DOS 命令手工生成二进制文件。

第二、　在“ DebugRel Settings ”中，将“ Target->Target Settingsr->Post-linker”项的“ None ”修改成“ ARM fromELF ”，再在“ DebugRel Settings ”中的“ Linker->ARM fromELF->Output format ”中设置成 Plain binary。

2>　不能进入 SAM-BA 软件，总出现下述提示错误信息,但硬件连接都正确。

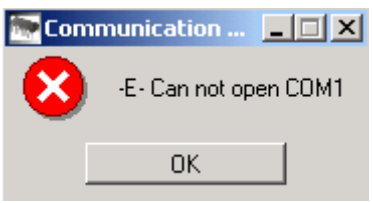


图. 错误信息 1　　图. 错误信息 2

原因是将用户程序下载到 AT91SAM7Sxx 芯片后，同时会擦除掉内部的 BootRom 程序，此时就不能再实现在线下载的功能！因此起动 SAM-BA 软件会弹出上述的出错信息，必须重新恢复内部的 BootRom 引导程序。可将 AT91SAM7S64 的 TST 管脚接高电平 10S 多时间，再重新复位。此时每次复位后都将在 DBGU 口发送“ RoomBoot ”的 ASCII 字符，可用超级终端查看。

3> 不能进行 RomBoot 恢复，且不能与仿真器进行连接。

原因是器件的安全保密位被编程，可以将 AT91SAM7S64 的 ERASE 脚接高，将 flash 内容初始化（或者说是擦除），再进行 RomBoot 恢复。

五．总结

到目前为此，基本上完成了 AT91SAM7S64 的大部分外围实验，并且也能够将程序烧到 AT91SAM7S64 的内部 Flash ROM 里，而脱离仿真器进行运行。我个人认为其它没有完成的实验在原理与使用上基本相同，应该没有什么难点。

可以说现在已经基本上能够将 AT91SAM7S64 像普通单片机一样使用了，但作为 ARM 这种高性能的处理器来讲，这是远远不够的，我希望自己有时间能够将μCOS-Ⅱ移植上去，届时再与大家讨论。

阿南

2005 年 5 月 6 日