

# AVR 单片机 GCC 程序设计

芯 艺

2005-01-01

请到 <http://www.chip-art.net> 或 <http://chipart.17kl.net>

下载最新版本和示例程序

问题反馈：

E-mail : [chipart@chip-art.net](mailto:chipart@chip-art.net)

[changfutong@sina.com](mailto:changfutong@sina.com)

OICQ : 27796915

感谢您的支持!

## 目 录

### 前言

### 第一章 概述

- 1.1 AVR 单片机 GCC 开发概述
- 1.2 一个简单的例子
- 1.3 用 MAKEFILE 管理项目
- 1.4 开发环境的配置
- 1.5 实验板 CA-M8

### 第二章 存储器操作编程

- 2.1 AVR 单片机存储器组织结构
- 2.2 I/O 寄存器操作
- 2.3 SRAM 内变量的使用
- 2.4 在程序中访问 FLASH 程序存储器
- 2.5 EEPROM 数据存储器操作
- 2.6 avr-gcc 段结构与再定位
- 2.7 外部 RAM 存储器操作
- 2.8 堆应用

### 第三章 GCC C 编译器的使用

- 3.1 编译基础
- 3.2 生成静态连接库

### 第四章 AVR 功能模块应用实验

- 4.1 中断服务程序
- 4.2 定时器/计数器应用
- 4.3 看门狗应用
- 4.4 UART 应用
- 4.5 PWM 功能编程
- 4.6 模拟比较器
- 4.7 A/D 转换模块编程
- 4.8 数码管显示程序设计
- 4.9 键盘程序设计
- 4.10 蜂鸣器控制

### 第五章 使用 C 语言标准 I/O 流调试程序

- 5.1 avr-libc 标准 I/O 流描述
- 5.2 利用标准 I/O 流调试程序
- 5.3 最小化的格式化的打印函数

### 第六章 CA-M8 上实现 AT89S52 编程器的实现

- 6.1 编程原理
- 6.2 LuckyProg2004 概述
- 6.3 AT989S52 isp 功能简介
- 6.4 下位机程序设计

### 第七章 硬件 TWI 端口编程

- 7.1 TWI 模块概述
- 7.2 主控模式操作实时时钟 DS1307
- 7.3 两个 Mega8 间的 TWI 通信

### 第八章 BootLoader 功能应用

- 8.1 BootLoader 功能介绍
- 8.2 avr-libc 对 BootLoader 的支持
- 8.3 BootLoader 应用实例
- 8.4 基于 LuckyProg2004 的 BootLoader 程序

### 第九章 汇编语言支持

- 9.1 C 代码中内联汇编程序
- 9.2 独立的汇编语言支持
- 9.3 C 与汇编混合编程

### 第十章 C++语言支持

### 附录 1 avr-gcc 选项

### 附录 2 Intel HEX 文件格式描述

## 前 言

2000 年在学校我第一次接触 Linux , 那时在班里学习做网页的气氛较浓 , 我也是为了学习 PHP 才认识 Linux 的。PHP 解析器的安装过程让我多多少少对 GCC 有了些认识 , 不过当时我并没有意识到它还可以用于嵌入式系统编程上 , 至于 Linux 、认为用处只在于架构一个免费的 WEB 站点。

2002 年在公司我第一次接触了 AVR , 当时公司只有一个 ICC 序列号 , 导致我和一个同事经常抢用一台机器 , 不过这种状况没有持续多久 , 我在浏览双龙的数据光盘时发现一个叫 “ 如何使用 AVR-GCC ” 的 PDF 文档 , 上面没有署名 , 我想我应该感谢该文的作者。我发现这个免费的编译软件远比 ICC 好用。

今天 , 我把使用 AVR-GCC 两年来的体会写在这里 , 愿更多的人了解和使用 GCC 及自由软件。

水平有限 , 错误和缺陷在所难免 , 请广大读者批评指正。

作者 : 芯 艺

2004-12-13

于内蒙古包头

## 声明：

- 1.内容中所有的示例均在硬件上调试通过
- 2.请不要编辑内容和示例程序，发现错误和作者联系
- 3.所有的引用请注明出处
- 4.对使用本软件后的一切后果，本人不负任何责任

## 第一章 概述

### 1.1 AVR 单片机 GCC 开发概述

#### 一 AVR 单片机介绍

##### 诞生：

1997 年 ATMEL 公司挪威设计中心的 A 先生和 V 先生出于市场需求的考虑推出了全新配置的 8 位精简指令集微处理器 ( RISC-Reduced Instruction Set CPU )。起名为 AVR。

##### 系列和主流：

AVR 是一种指令内核的统称，它内部又分 ATtiny、AT90S 和 ATmega 三大系列，分别对应 AVR 的低、中、高档产品。Attiny 系列中常用的有 Attiny15、Attiny2313 等产品，AT90S 系列中常用的有 AT90S2313、AT90S8535、AT90S8515 等。AT90S2313 引脚兼容 AT89C2051，而 AT90S8515 的引脚则兼容 51 单片机，在设计中很容易的替代 51 单片机。但是到目前，AT90S 系列的绝大部分已停产，当某一个 AT90S 系列芯片停产时 ATMEL 通常会在 ATmega 或 ATtiny 系列中推出一个新的替代产品，替代产品往往在引脚兼容的基础上内部资源和性能得到加强。例如 Attiny2313 作为 AT90s2313 的替代产品在 AT90S2313 的基础上增加了片内标定振荡器、增强型上电复位、可编程的掉电检测等多种功能。类似的 ATmega8515 和 ATmega8535 分别作为 AT90S8515 和 AT90S8535 的替代产品继承了很多 Mega 系列的特性。

ATmega48/88/168、ATmega8、ATmega16、ATmega32、ATmega64 和 ATmega128 是 ATmega 系列的主流产品。值得关注的是 ATmega8 这一款单片机，它以丰富的片内资源、低廉的价格深受广大设计人员的喜爱，并在国内得到了较好的推广。然而 ATmega48/88/168 作为它的兼容产品，为用户提供更多功能的选择。

ATmega16 也是个用量较多的器件，它的引脚兼容 AT90S8535，可以取代产品中的 8535 芯片。相对 ATmega8 它除了 I/O 引脚多之外内部集成了 ATmega8 两倍 ( 16K ) 的 FLASH 程序存储器。

随着国内 AVR 用户的增多，ATMEL 也开始了主流器件数据手册的中文翻译工作。从网络上可以找到 Attiny2313、ATmega48/88/168、ATmega8、ATmega16、ATmega32、ATmega64 等器件的官方翻译中文数据手册。这对学习和使用 AVR 器件提供了很大的帮助。

##### 硬件结构：

有关 AVR 硬件结构，各功能模块的描述请参考数据手册或相关的书籍。

##### 开发工具：

更适合采用高级语言开发是 AVR 单片机最初设计时的目的之一。目前 AVR 单片机高级语言开发工具主要有 IAR C、WINAVR、ICCAVR、CodeVision、BASCOS-AVR ( BASIC 语言 )。IAR 是与 AVR 内核协同开发的，很多对 AVR 更适合 C 语言开发方面的改进是根据 IAR 开发过程的，它也是 ATMEL 公司推荐的 C 语言编译器，但它的价格几乎让人难以接受，达到上万元人民币。ICCAVR 是国内 AVR 主要推广单位双龙公司代理的 C 编译器，它价格低廉，性能也 XX，友好的界面把很多烦琐的项目管理和编译设置隐藏了起来，为此它受部分开发人员的欢迎。CodeVision 也是个很不错的 C 编译器，目前在国内也有一定数量的用户。Winavr 是个免费的 AVR 开发程序集，它以著名的自由软件 GCC 为 C/C++ 编译器。下面所

有章节我都会介绍如何使用 WINAVR 开发 AVR 单片机。学习 GCC 的意义绝不仅仅是为了开发 AVR 程序，正如 21ICBBS 上一位网友所说：“如果其它编译器是一棵树，那么 GCC 就是个树林”，GCC 支持多种处理器，包括 ARM、DSP、X86 等 32 位 CPU。它的历史足以说明它是个成熟的编译器。

如果不是个业余的程序开发人员，我不会建议你使用 BASIC 编写 AVR 程序，因为我们在编写与硬件密切相关的单片机程序时通常需要想象编译器是如何将这些代码翻译成汇编程序的，而 C 更适合我们去这样做。

## 二 GCC 编译器

UNIX 上最原始的 C 编译器叫 CC (C Compiler C 编译器)，源于此 GNU 的 C 编译器叫作 GCC (GNU C Compiler)，然而，随着 GCC 支持语言的增加 GCC 这个缩写的意义已演变成了 GNU 编译器集合 (GNU Compiler Collection)，它是 GNU 项目的一个产品，是个开放源代码软件。

GCC 可编译多种语言，目前支持的语言有 C、C++、Objective-C、Fortran、java 和 Ada。

这些高级语言程序通过编译程序前端 (front-end) 后产生解析树，之后与器件相关的后端 (back-end) 程序将它们解释成实际的可执行指令集。前端与后端是完全分开的，解析树是它们中间的产物。GCC 这样的设计使得任何一种语言只要通过合适的语法解析器产生合格格式的解析树，就可以产生 GCC 后端程序所支持的所有器件上的可执行指令集。同样的，任何一种器件只要将树结构翻译成汇编，就可以使用 GCC 前端所支持的所有语言。

要承认的是，以上描述是理论化的，便于理解，实际操作并没有想象的那么简单。事实上前端和后端都不是孤立的。幸运的是 AVR 的确得到了 GCC 的支持，它也是 GCC 支持的唯一一种 8 位处理器。不仅如此，我们还可以在 WINDOWS 平台上安装程序包 WINAVR 来使用 GCC 的 AVR C/C++ 编译程序。

WINAVR 是一组开放源代码的程序集，用于 ATMEL 公司 AVR 系列单片机的开发，它主要包含：

1. GNU 程序包 Binutils

GNU Binutils 非常庞大，WINAVR 仅包含与 AVR 相关的部分，有 AVR 汇编器、连接器以及与机器指令相关的一些工具

2. GNU GCC 的 C 和 C++ 编译器。

3. AVR-LIBC

AVR-LIBC 是 AVR 单片机 C 运行时库，它是为应用程序提供标准 C 函数的连接时定位库。

另外 WINAVR 还包含软件调试器、器件编程软件、文件格式转换工具等多个有用工具，这里不一一列出，请参考 WINAVR 说明文档。WINAVR 项目的 WEB 地址是 <http://sourceforge.net/projects/winavr>，这里可以下载最新的版本，并可订阅邮件组。邮件组是获取 WINAVR 相关帮助信息的重要途径。



## 1.2 一个简单的例子

为了先有一个感性的认识，我们首先看一下如下一段程序和它的编译、连接过程。  
文件 demo1.c :

```
#include <avr/io.h>

int main( void )
{
    unsigned char  i, j, k, led=0;

    DDRB=0xFF;

    while (1)
    {
        if(led)
            PORTB|=0X01;
        else
            PORTB&=0XFE;

        led=!led;

        //延时
        for (i=0; i<255; i++)
            for(j=0; j<255;j++)
                k++;
    }
}
```

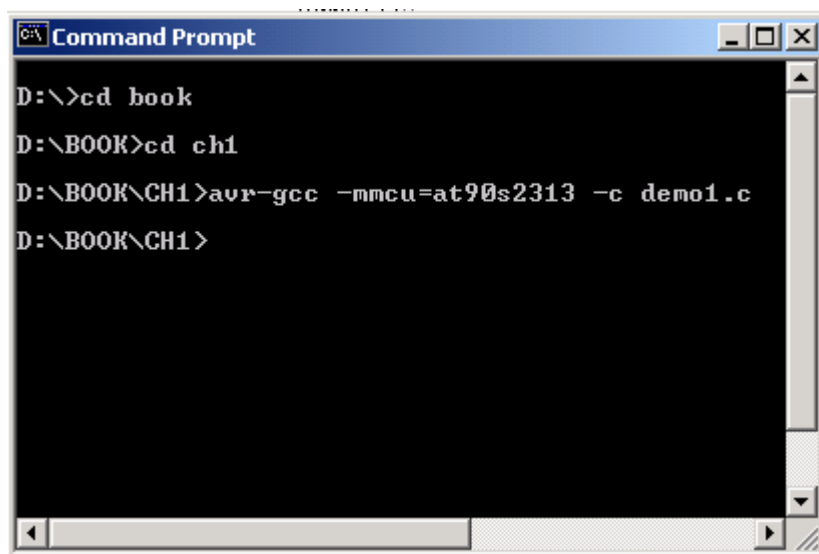
这是一个使接在 PB0 口的 LED 发光管闪烁的程序。有了源程序文件 demo1.c，我们就可以编译它了。通过点击菜单开始->运行 在弹出的对话框中输入“command”，来打开控制台窗口，并在命令行输入：

```
avr-gcc -mmcu=at90s2313 -c demo1.c
```

如图 1-1 所示。

必需告诉编译器 程序的 MCU 类型，这是我们通过命令行选项 -mmcu 来指定的，我们指定的器件为 at90s2313。-c 选项告诉编译器编译完成后不连接。

图 1-1 控制台窗口



```
Command Prompt
D:\>cd book
D:\BOOK>cd ch1
D:\BOOK\CH1>avr-gcc -mmcu=at90s2313 -c demo1.c
D:\BOOK\CH1>
```

编译完成后在工作目录新生成了一个文件：demo1.o，它是我们的目标文件，我们再使用连接器将它连接成可在器件上执行的二进制代码。

在命令行输入：

```
avr-gcc -mmcu=at90s2313 -o demo1.elf demo1.o
```

之后我们会在工作目录看见连接器生成的 demo1.elf。gcc 的连接后生成的文件为 ELF 格式，在命令行我们通常用 .elf 指定其扩展名。ELF 格式文件除了包含不同存储器的二进制格式内容外还包含一些调试信息，所以我们还要借助一个有用工具 avr-objcopy 来提取单片机程序存储器内容。命令行输入：

```
avr-objcopy -j .text -j .data -O ihex demo1.elf demo1.hex
```

gcc 把不同类型的数据分到不同的段落，相关程序存储器的段有 .text 和 .data，我们用选项 -j 指定了要提取的段。选项 -O 用来指定输出格式，这里我们指定为 ihex (intel HEX file)。

到此我们得到了最终可以写入单片机 90S2313 FLASH 存储器的 demo1.hex 文件。用编程器将 demo1.hex 内容写入到单片机，便可看到接在 PB0 口的 LED 不断的闪烁。

以上对一次编译过程的描述只是为了说明 gcc 编译一个 C 源程序的步骤，在实际的应用中我们很少用这种方式编译每一个源程序和每一个更新后的程序。而是借助一个叫 make 的项目管理工具来进行编译操作。Make 由下一节介绍。

## 1.3 用 MAKEFILE 管理项目

在我看来，通常一个编译器（泛指高级语言编译器、汇编器、连接器等等）项目管理器和本编辑器构成一个完整的编程环境。

其中项目管理器负责为每一个源程序文件调用编译器生成目标文件后用连接器将它们组合在一起生成可执行文件。例如 Keil uVision 进行编译时将项目中的每一个源程序文件进行编译后生成对应的.obj 文件，之后将这些目标文件连接到一起生成可执行（例如 iHex 格式）文件。这是对编译过程的粗略的描述，Keil uVision 掩盖了很多的细节，使用过早期 C51 编译器的读者对此一定有所了解。

WINAVR 没有像 Keil uVision 那样集成项目管理器的 IDE，所以我们需要写一个叫做 makefile 的文件来管理程序的编译连接。makefile 是个脚本文件，一个标准的(应该说经典的)可执行文件 make.exe 负责解析它并根据脚本内容来调用编译器、连接器或其它的工具，最终生成可执行代码文件。每次调用 make 时它会比较目标文件与源文件的更新时间，如果源文件比目标文件还要新它会执行 makefile 内的相关指令更新目标文件，如果目标文件与源文件一样新它就跳开这个源文件的编译，避免重复工作，这对于一个较大的工程是个节省时间的有效方法。

### 1.3.1 make 的使用

Make 指令的用法是：

```
Make [-f filename] [names]
```

方括号表示括号里边的内容可以省略。其中 filename 代表 make 所使用的项目描述文件，如果此项省略，则从当前目录下按下列顺序寻找默认的项目描述文件

GNUmakefile.

makefile

Makefile（当然、在 WINDOWS 下不份大小写文件名，也就无所谓了）

names 指定目标名或宏名。若不指定目标名，则 make 命令总是把在 makefile 文件中遇到的第一个目标当作默认目标。

了解 makefile 脚本内容是非常有用的，但这不是必需的，因为 WINAVR 中一个小工具 Mfile 可以为我们生成能够用的 makefile 样本。所以如果对 makefile 不太感兴趣，可以跳过下面的小节。

### 1.3.2 Makefile 项目描述文件

#### 一.目标

make 命令引入了目标（targets）的概念。Makefile 描述文件便是它的第一个目标，make 命令必须处理至少一个目标，否则不会得出任何结果。正如我们在一个没有默认描述文件的当前目录下敲入 make 一样，make 会输出以下的结果：

MAKE : \*\*\*No targets specified and no makefile found. Stop.

1.在项目描述文件中定义目标

一个目标通常从一行的开头开始，并后跟一个冒号。

最简单的 MAKEFILE

```
#当前目录 D:\AVRGCC\TEST
all:
    @echo hello!
#End makefile
```

all: 便是第一个目标

调用此描述文件结果：

```
D:\AVRGCC\TEST>make
```

```
hello!
```

2.默认目标(goal)

在上面提到过，如果调用 make 时不指定目标名则 make 总是假设在描述文件中遇到的第一个目标是默认目标。以下示例可以非常好的说明这一问题。

具有三个目标的 makefile

```
#当前目录 D:\AVRGCC\TEST
one:
    @echo one.
Two:
    @echo two.
Three:
    @echo three.
#End makefile
```

调用 1：

```
D:\AVRGCC\TEST>make
```

```
one.
```

由于在命令行没有指定目标，make 从 makefile 中找到第一个目标 (one) 并执行后既退出。

调用 2：

```
D:\AVRGCC\TEST>make two
```

```
two.
```

由于在命令行明确指定了要执行的目标 (two),make 在 makefile 中找到指定的目标，并执行后退出。

调用 3：

```
D:\AVRGCC\TEST make three one two
```

```
three.
```

```
one.
```

```
two.
```

命令行指定了三个目标，make 一一寻找并执行。

在 makefile 中非默认的目标称为可替换的目标，只有默认的目标与它们存在直接或间接的依赖关系时它们才有可能被调用。

### 二. 依赖关系

makefile 文件按如下格式指定依赖关系：

```
目标 1[目标 2 ... ]:[ : ][依赖 1][依赖 2] ...  
    [命令]
```

如下例

```
#当前目录 D:\AVRGCC\TEST  
one: Two  
@echo one.  
Two:  
@echo two.  
#End makefile
```

执行结果是：

```
d:\avrgcc\test>make  
two.  
one.
```

Make 首先找到第一个目标 one，之后发现目标 one 依赖目标 Two 就先执行 Two 后才执行 one 中的命令。

### 三. Makefile 内容

makefile 内容可分为如下五种类型

规则定义

语法：

目标：依赖

命令

...

其中目标为一个文件名或以空格分开的多个文件名，可含通配符。

例如：

```
%.o : %.c  
    avr-gcc -c $< -o $@
```

以上规则定义了任意一个以 .o 结尾的文件依赖于相同前缀且以 .c 结尾的文件。并执行下边的命令获得。

规则中目标和依赖分别为 %.o 和%.c，在目标通配符“%”代表任意的字符串，而在依赖中代表与目标中代表的对应字符串。

### 隐含规则

隐含规则是 make 预先定义的规则，用选项 `-r` 可取消所有的隐含规则。

例如对于 C 程序 `%.o` 可以自动的从 `%.c` 通过命令

```
$(CC) -c $(CPPFLAGS) $(CFLAGS) ' 生成。
```

### 变量

变量是在 makefile 中描述一字符串的名称。变量可用在目标、依赖、命令和 makefile 其它部分中。变量名由除 `:`、`'`、`#`、`=` 之外的字符组成，对大小写敏感。

变量的定义并赋值格式：

```
变量名 = 变量代表字符串
```

变量的引用格式：

```
$(变量名)
```

例如：

```
CC = avr-gcc
%.o : %.c
$(CC) -c $< -o $@
```

### 命令

命令部分是由 make 传递到系统的字符格式的命令行的组合，在目标被创建时它们按顺序一行一行传递到系统并执行。

字符 `'@'` 开始的命令 在系统的输出中不显示本次的指令行。

### 注释

字符 `'#'` 开头的行为注释行，如果注释需要换行需在行尾加 `'\'`，除包含其它 MAKEFILE 外在行的任意处可插入注释。

## 四.自动变量

在 makefile 中有一组预定义的变量，当每一规则被执行时根据目标和依赖重新计算其值，叫作自动变量。

下面列出了常用的几个自动变量

`$@` ：在规则中代表目标名，如果规则含有多个目标名它将列举所有目标。

`$%` ：仅在目标是存档文件的成员时起作用，代表目标。

如目标 `foo.a(bar.o)` 中 `$@` 代表 `foo.a` `$%` 代表 `bar.o`

`$<` ：在规则中代表第一个依赖文件名

`$?` ：代表在规则中所有以空格隔开的依赖文件名，如果依赖是存档文件的成员则只有成员名被列出。

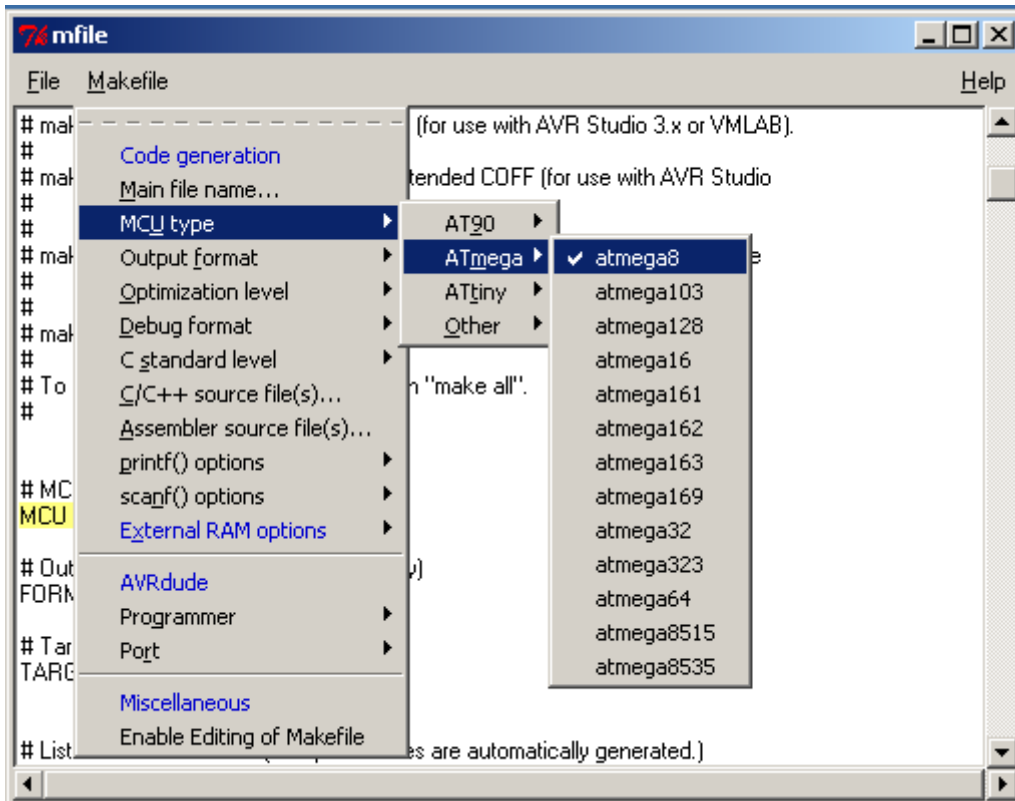
\$^ : 代表在规则中所有以空格隔开的依赖文件名，如果依赖是存档文件的成员则只有成员名被列出。

### 1.3.3 使用 Mfile 生成合适的 makefile

WINAVR 包中有一种简单的 makefile 文件生成工具叫 mfile，它的运行界面如图 1-2，利用它我们可方便的生成合适的 makefile。

- 《Main file name...》菜单指定主程序文件，它将决定主源程序文件名及输出文件名。
- 《Output format》菜单项用于选择最终生成的可指行代码格式，根据编程器支持格式选择即可。
- 《Optimization level》指定 C 代码的优化级，s 代表按最小代码量编译。
- 《C/C++ source file(s)》和《Assembler source files(s)》用于在项目中添加其它 C、C++、和汇编程序源文件。

图 1-2 mfile 生成 makefile



通常我们选择了以上几项就可以了。在编写程序的过程中往往需要在项目中添加源文件，这时只需在生成 makefile 中 SRC 变量后列出源文件名即可。

Mfile 生成的 makefile 有四个目标分别是：

all : 编译程序生成可执行文件。这也是默认的目标，即在命令行当前目录下一个没有任何选项的 make 命令将执行这一目标。

clean : 在当前目录清除编译生成的所有文件。

coff : 将 ELF 格式的调试文件转换成 AVR Studio 3.x 中可以调试的 AVR COFF 格式。

extcoff : 将 ELF 格式的调试文件转换成 AVR Studio 4.07 扩展 COFF 格式。

另外, C 源文件名的后缀改成“.s”后的目标指示将生成该源文件编译后的汇编代码文件(以“.s”结尾)。

## 1.4 开发环境的配置

### 一. 软件环境

UltraEdit + WinAVR 打造超级开发 IDE

UltraEdit 是个文本编辑器, 它支持 C 代码的高亮显示、项目管理及外部工具配置等功能。

首先要安装 UltraEdit 和 WinAVR。

#### (1) UltraEdit 的个性化设置:

下面是我个人习惯的设置

视图->颜色设置 光标所在行文本 设置成黑,光标所在行背景设置成白

高级->配置->编辑 制表符宽度值和缩进空格均设为 4。

高级->配置->备份 保存时备份文件里选择 不备份。

视图->查看列表 选中函数列表

#### (2) 创建编译用的文件

先在硬盘上创建一个目录作为设计大本营, 这里假设为 d:\devdir

UltraEdit 写主程序文件保存到此文件夹中 这里设为 demo.c

用 mfile 生成一个合适的 makefile 保存到 d:\devdir

UltraEdit 创建一项目, 负责管理文件

项目->新建项目 目录选 d:\devdir 输入项目名称(这里假设为 prj)

在接下来的文件设置对话框中的项目文件目录区输入或选择 d:\devdir

选中 相对路径 复选按钮

通过 添加文件 按钮将刚才的 makefile 和 demo.c 添加到项目中, 之后按关闭。

#### (3) 在 UltraEdit 中 make 我的项目

高级 -> 工具配置

在命令行区输入 make

在工作目录区输入 d:\devdir

在菜单项目名称区输入一个任意的菜单名称

选中 输出到列表框 和 捕获输出两个选择按钮后单击 插入按钮 确定。

至此你就可以在 UltraEdit 内就可以 make 你的程序了☺

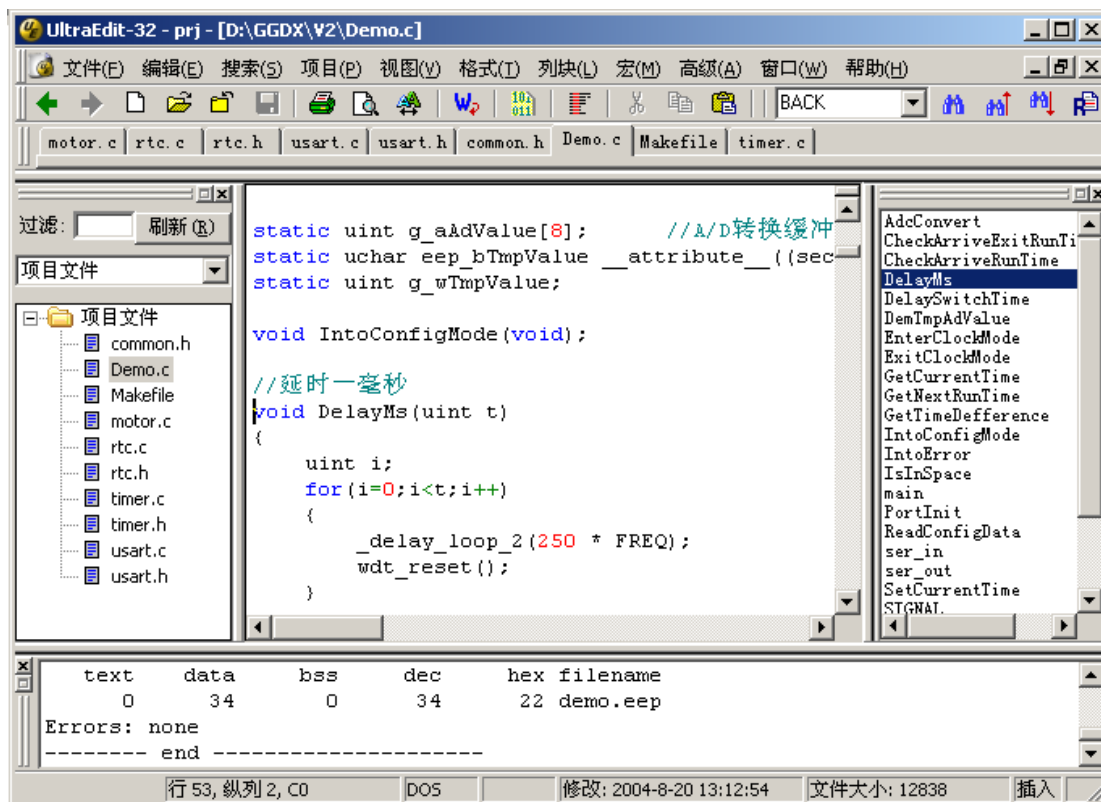
如果不愿意每次编译时找菜单可用快捷键 Ctrl+shift+0。

记得要在你的项目里添加源程序时,除了 在 UltraEdit 项目->文件设置里添加外还要在 makefile 的 SRC 变量后列出来才可编译哦☺



到此 我们的超级无敌 AVR 编程环境打造完成 , 如图 1-3。

图 1-3 配置后的 UltraEdit 外观



## 二 . 硬件环境

### SI-Prog + PonyProg 实现最廉价的下载实验器

AVR 系列单片机提供对程序存储器 (FLASH) 和数据存储器 (EEPROM) 的串行编程功能 (ISP), 使它的程序烧写变得方便。AVR 系列器件内部 FLASH 存储器的编程次数通常可达到 10000 次以上, 所以使用多次烧写的方式调试程序时不必担心器件的损坏。

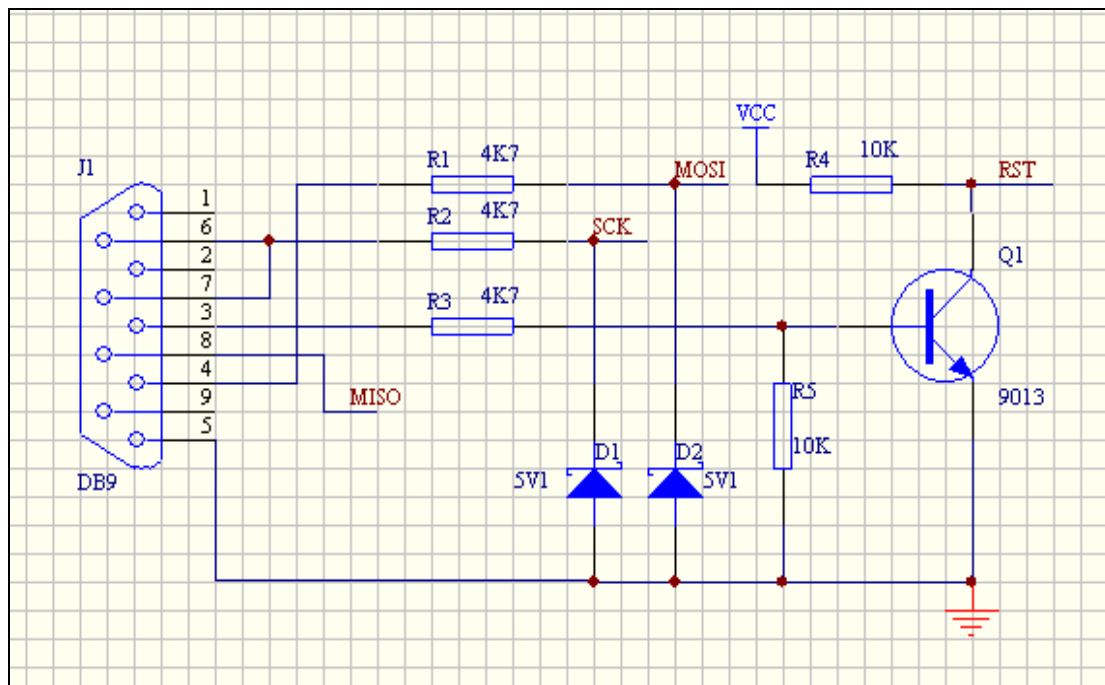
ISP 功能占用三个 I/O 端口 (MOSI、MISO、SCK) 与外部编程逻辑通信, 编程逻辑按指定时序将程序数据串行方式发送到器件, 器件内的 ISP 功能模块负责将数据写入到 FLASH 或 EEPROM。

在实际应用中通常利用 PC 机的并行口或串行口加一个下载适配器 (下载线) 实现一个编程硬件, AVR 的下载线有很多种, 这里向大家推荐 SI-Prog, SI-Prog 具有制作方便 (只需几个分立元件)、接线少 (通过 PC 9 针串行口编程)、支持软件成熟 (PonyProg) 等特点。si-prog 的完整电路可到 <http://www.LancOS.com> 下载。图 1-4

为简化后的电路原理图。

PonyProg 是个串行器件编程软件，支持 AVR 在内的多种器件的串行编程。该软件可到 <http://www.LanOS.com> 下载。

图 1-4 SI-Prog 电路原理图



有了一台安装有 PonyProg 的 PC 机和 SI-Prog，就可以将程序写入到实际器件来验证了，想一想此方案的成本和一个 AVR 芯片能烧写的次数，是不是觉得很值©

## 1.5 实验板 CA-M8

特性:

- 实验器件为有代表性的 AVR 单片机 ATmega8
- 三个 8 位拨码开关实现板上电路与 MCU I/O 脚间的连接与断开
- 74HC595 驱动的两位数数码管显示电路
- 两个可由 I/O 口控制的 LED 发光管
- 两个可连接 Mega8 中断口的按钮开关
- 可由 I/O 口控制的蜂鸣电路
- 可选的外部 4MHz 或 32768Hz 晶振
- UART 与 RS-232 间的电平转换电路，实现与计算机间的串行通信
- 用于对板上器件编程或可引出的 SI\_Prog 下载编程电路
- 用于测试片内 ADC 的电位器调压电路
- 配合 LuckyProg2004 对其它 ISP 器件编程用的接口引出
- 可选的外部按键复位源电路
- 简洁的 7805 稳压电路及指示电路

## 设计目的：

1. 为本书多数例题提供可用的测试硬件
2. 为在校的不满足于 MCS-51 知识、且注重实践的电子类专业学者提供廉价而完整的 AVR 学习硬件
3. 为那些非电子、自动化相关专业的单片机爱好者,并在单位里没有较好条件的人员提供廉价而实用的学习开发板
4. 为 LuckyProg2004 提供标准的硬件电路,因此拥有 CA-M8 的最初目的可能是用于学习开发,而最终可能经常当作编程器使用

## 电路原理图

见第 17 页 图 1-5.

## 配置：

在 CA-M8 上三个 8 路拨码开关将板上功能电路与 ATMEGA8 的 I/O 引脚隔开,为此使用板上电路做实验时应当配置好这些拨码开关。表 1-1 列出了 ATMEGA8 与板上功能电路间的这一对应关系。

表 1-1 CA-M8 配置开关一览表

拨动开关	ATMEGA8 I/O 引脚	板上功能电路	备注
S6-1	PB6(XTAL1)	外部 4MHz 晶振	使用外部晶振时使用
S6-2	PB7(XTAL2)		
S6-3	PB6(TOSC1)	32768Hz 晶振	使用实时时钟功能时使用
S6-4	PB7(TOSC1)		
S6-5	PD4	74HC595 三个控制端口	SER
S6-6	PD5		RCK
S6-7	PD6		SCK
S6-8	PB2	蜂鸣器驱动电路	高电平有响声
S5-1	PC6(RESET)	复位电路	ISP 下载程序时使用
S5-2	SCK	SI_Prog 下载线	
S5-3	MISO		
S5-4	MOSI		
S5-5	RXD	MAX202 电平转换电路	与计算机串行通信时使用
S5-6	TXD		
S5-7	INT0	按键 S3	
S5-8	INT1	按键 S4	
S1-1	PC2	对其它器件串行编程操作端口	RST
S1-2	PC3		MISO
S1-3	PC4		SCK
S1-4	PC5		MOSI
S1-5	AREF	VCC	
S1-6	PC0 ( ADC0 )	电位器调压电路	
S1-7	PB1	红色发光管	低电平亮
S1-8	PB0	黄色发光管	低电平亮

## 时钟源选择:

内部时钟：当通过熔丝位选择内部时钟时应当将 S6-1 ~ S6-4 断开

外头部 4MHz：开通 S6-1 和 S6-2 并将熔丝位设置成外部晶振，此时 S6-3, S6-4 应断开。

外部低频晶振 (32768Hz)：开通 S6-3 和 S6-4 并将熔丝位设置成低频晶振，此时 S6-1, S6-2 应断开。

## 复位源选择:

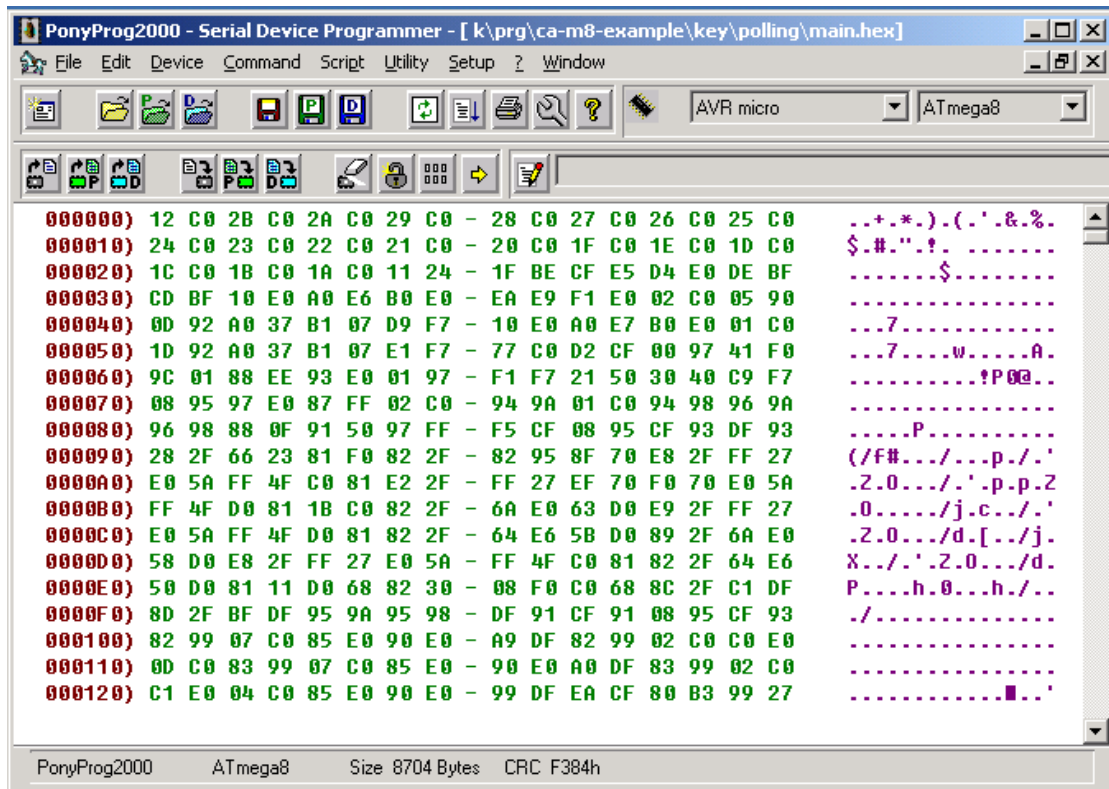
当使用板上复位按钮或 SI\_Prog 下载程序时不仅需打开 S5-1 还要使用 J7 插座的 ERS 和 RST 两个脚用短路块短接。

## 使用板上下载线对器件编程:

使用板上 SI\_Prog 下载程序时应当将 S5-1 ~ S5-4 开通。上位机软件建议使用 PonyProg2000。

PonyProg2000 执行界面如图 1-6 所示：

图 1-6 PonyProg2004 执行界面



J5 是 SI\_Prog 引出的单排插座,可以使用此接口对其它 AVR 器件进行 ISP 编程。

## 配置时的几点注意事项：

1. 当 S6-1 和 S6-2 打开时不能打开 S6-3 和 S6-4.
2. 当 S5-7 和 S5-8 打开时不能在程序中将 PD2 和 PD3 设置为输出高电平.

3. 当 S1-5 打开时不能选择内部 ADC 电压基准.

实物照片：

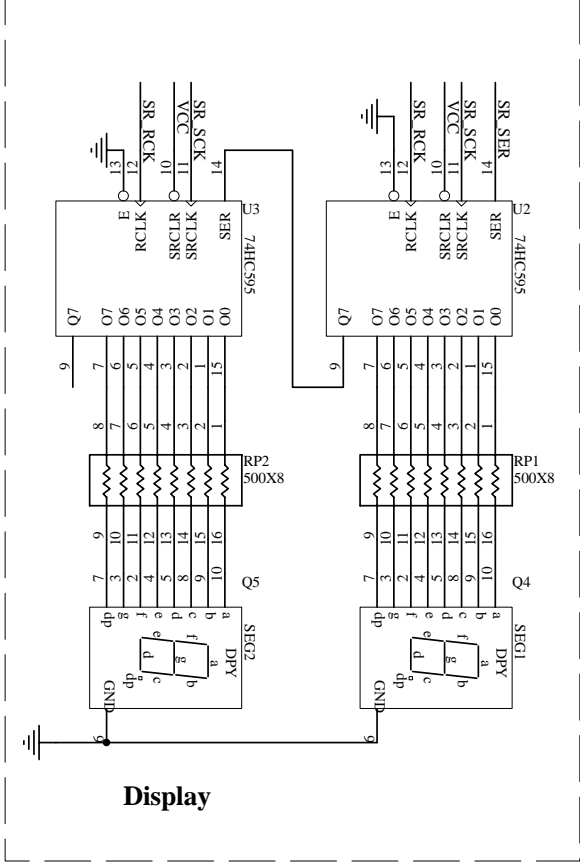
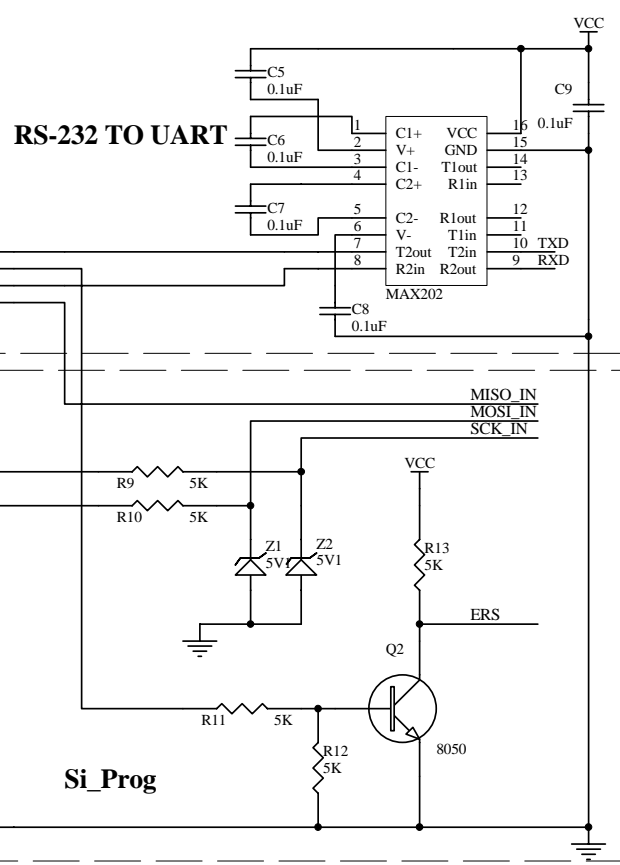
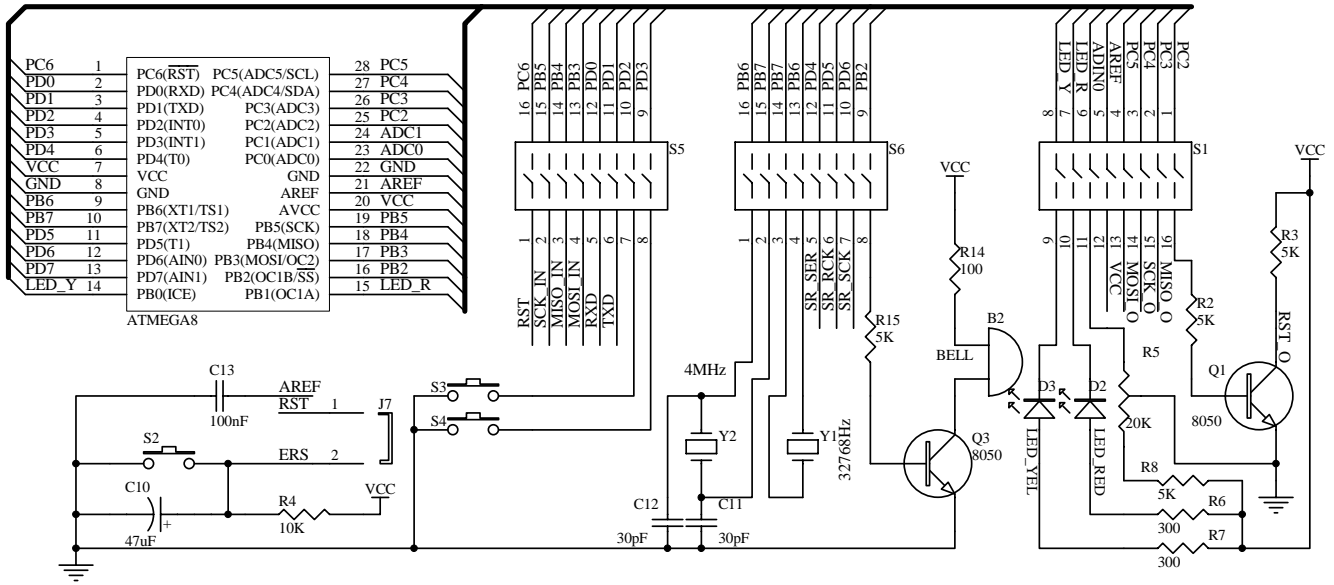
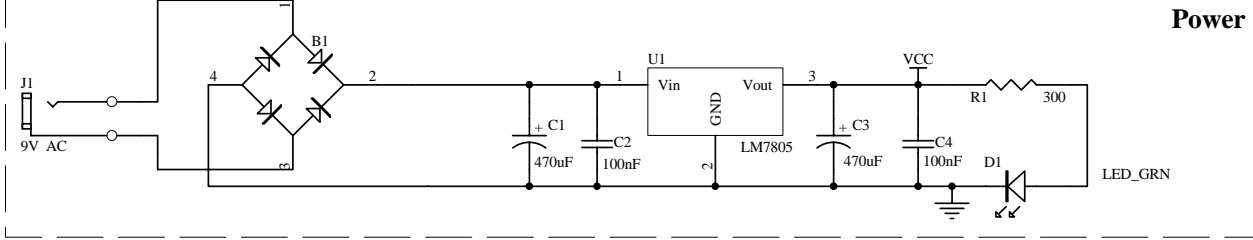
如图 1-7 所示

图 1-7 CA-M8 实物照片



下载更多的示例程序或购买 CA-M8 请到 <http://www.chip-art.net> 或 <http://chipart.17kl.net> . 这里有关于 CA-M8 更多的信息。

读到这里你也许还没有感觉到我其实一直在倡导一种开发模式,那就是 :使用 C 等高级语言开发,利用现代单片机多次烧写特性和 ISP 编程功能等特点,通过将程序写入到实际单片机来验证。我们不需要仿真器,甚至软件仿真都不需要。CA-M8 作为一个开发板,就是个典型的例子。你赞同这种开发模式吗?请继续阅读。



## 第二章 存储器操作

### 2.1 AVR 单片机存储器组织结构

AVR 系列单片机内部有三种类型的被独立编址的存储器，它们分别为：Flash 程序存储器、内部 SRAM 数据存储器和 EEPROM 数据存储器。

Flash 存储器为 1K ~ 128K 字节，支持并行编程和串行下载，下载寿命通常可达 10,000 次。由于 AVR 指令都为 16 位或 32 位，程序计数器对它按字进行寻址，因此 FLASH 存储器按字组织的，但在程序中访问 FLASH 存储区时专用指令 LPM 可分别读取指定地址的高低字节。

寄存器堆（R0 ~ R31）、I/O 寄存器和 SRAM 被统一编址。所以对寄存器和 I/O 口的操作使用与访问内部 SRAM 同样的指令。其组织结构如图 2-1 所示。

图 2-1 AVR SRAM 组织



32 个通用寄存器被编址到最前，I/O 寄存器占用接下来的 64 个地址。从 0X0060 开始为内部 SRAM。外部 SRAM 被编址到内部 SRAM 后。



AVR 单片机的内部有 64 ~ 4K 字节的 EEPROM 数据存储单元，它们被独立编址，按字节组织。擦写寿命可达 100,000 次。

## 2.2 I/O 寄存器操作

I/O 专用寄存器 (SFR) 被编址到与内部 SRAM 同一个地址空间，为此对它的操作和 SRAM 变量操作类似。

avr-libc 对 SFR 定义了符号，使用这些符号需包含头文件 `io.h`，包含方式如下：

```
#include <avr/io.h>
```

`io.h` 文件在编译器包含路径下的 `avr` 目录下，由于 AVR 各器件间存在同名寄存器地址有不同的问题，`io.h` 文件不直接定义 SFR 寄存器宏，它根据在命令行给出的 `-mmcu` 选项再包含合适的 `ioxxxx.h` 文件。

在器件对应的 `ioxxxx.h` 文件中定义了器件 SFR 的预处理宏，在程序中直接对它赋值或引用的方式读写 SFR，如：

```
PORTB=0XFF;  
Val=PINB;
```

从 `io.h` 和其总包含的头文件 `sfr_defs.h` 可以追溯宏 `PORTB` 的本来的面目，例如，我们可以在 `io2313.h` 中找到 `PORTB` 的定义如下：

```
#define PORTB _SFR_I08(0x18)
```

而在 `sfr_defs.h` 中可以找到如下两个定义：

```
#define _SFR_I08(io_addr) _MMIO_BYTE((io_addr) + 0x20)  
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *) (mem_addr))
```

这样，`PORTB = 0XFF;` 就等同于 `*(volatile unsigned char *) (0x38) = 0xff;` 而 `0x38` 正是寄存器 `PORTB` 在器件 AT90S2313 中的地址。这进一步说明了 SFR 与 SRAM 操作的相同之处。在 C 程序中我们可以将符号 `PORTB` 看作是绑定到寄存器 `PORTB` 的变量。

关键字 `volatile` 确保本条指令不会因 C 编译器的优化而被省略。

另外，宏 `_BV(bit)` 是我们操作 I/O 寄存器时频繁用到的，`avr-libc` 建议使用这一宏进行寄存器的位操作，它在文件 `sfr_defs.h` 中定义如下：

```
#define _BV(bit) (1 << (bit))
```



以下是它的使用示例：

```
DDRB=_BV(PB0)|_BV(PB1); //io.h 中定义 PB0: 0 PB1: 1
```

它等同于 `DDRB=0X03;`，这样写的目的是为了提高程序的可读性。不要担心它会生成比 `DDRB=0X03;` 更大的代码，编译器会处理这种事情，最终会输出与 `DDRB=0X03;` 同样的结果。

## 2.3 SRAM 内变量的使用

在 C 程序中一个没有其它属性(attribute)修饰的全局或静态变量将被指定到内部 SRAM，而局部(自动)变量则用寄存器或栈来处理，当程序被解析到汇编时这些全局或静态变量名转换成全局的符号，这些符号标识了内部 SRAM 地址，而局部变量没有这种符号。

avr-libc 建议使用一组数据类型符号，这些数据类型的定义在头文件 `inttype.h` 中，头文件的包含形式如下：

```
#include <inttype.h>
```

其中定义了常用的整数类型如下表所示：

定义值	长度(字节)	值范围
int8_t	1	-128 ~ 127
uint8_t	1	0 ~ 255
int16_t	2	-32768 ~ 32767
uint16_t	2	0 ~ 65535
int32_t	4	-2147483648 ~ 2147483647
uint32_t	4	0 ~ 4294967295
int64_t	8	$-9.22 \times 10^{18} \sim -9.22 \times 10^{18}$
uint64_t	8	0 ~ $1.844 \times 10^{19}$

### 定义、初始化和引用

如下示例：

```
uint8_t val=8;      定义了一个 SRAM 变量并初始化成 8  
val=10;           改变变量值  
const uint8_t val=8; 定义 SRAM 区常量  
register uint8_t val=10; 定义寄存器变量
```

## 2.4 在程序中访问 FLASH 程序存储器

avr-libc 对 FLASH 存储器的读写支持 API 和宏在头文件 `pgmspace.h` 中定义，在源文件

中的包含形式如下：

```
#include <avr/pgmspace.h >
```

在程序存储器内的数据定义使用关键字 `__attribute__((__progmem__))`。在 `pgmspace.h` 中它被定义成符号 `PROGMEM`。

## 1. FLASH 区整数变量应用

定义格式：

数据类型 变量名 PROGMEM = 值；

如：

```
char val8  PROGMEM = 1 ;
int  val16 PROGMEM = 1 ;
long val32 PROGMEM = 1 ;
```

对于不同长度的整数类型 `avr-libc` 提供对应的读取函数：

```
pgm_read_byte(prog_void * addr)
pgm_read_word(prog_void *addr)
pgm_read_dword(prog_void* addr)
```

另外在 `pgmspace.h` 中定义的 8 位整数类型 `prog_char` `prog_uchar` 分别指定在 FLASH 内的 8 位有符号整数和 8 位无符号整数。应用方式如下：

```
char ram_val;           //ram 内的变量
const prog_char flash_val = 1; //flash 内变量

void fn(void)
{
    ram_val=pgm_read_byte(&flash_val); //读 flash 变量值到 RAM 变量
    ... ..
}
```

对于应用程序 FLASH 变量是不可改变的，因此定义时加关键字 `const` 是个好的习惯。

## 2. FLASH 区数组应用：

示例：

```
const prog_uchar flash_array[] = {0,1,2,3,4,5,6,7,8,9}; //定义
```

另外一种形式

```
const unsigned char flash_array[] PROGMEM = {0,1,2,3,4,5,6,7,8,9};
```

读取示例：

```
unsigned char I, ram_val;
for(I=0 ; I<10 ;I ++)// 循环读取每一字节
{
    ram_val = pgm_read_byte(flash_array + I);
    ... ..           //处理
}
```

## 2 . FLASH 区字符串变量的应用

全局定义形式：

```
const char flash_str[] PROGMEM = "Hello, world!";
```

函数内定义形式：

```
const char *flash_str = PSTR("Hello, world!");
```

以下为一个 FLASH 字符串应用示例

```
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <stdio.h>

const char flash_str1[] PROGMEM= "全局定义字符串";

int main(void)
{
    int I;
    char *flash_str2=PSTR("函数内定义字符串");
    while(1)
    {
        scanf("%d",&I);
        printf_P(flash_str1);
        printf("\n");
        printf_P(flash_str2);
        printf("\n");
    }
}
```

## 2.5 EEPROM 数据存储器操作

avr-libc 对 EEPROM 存储器读写 API 定义在 eeprom.h 中，包含形式如下：

```
#include <avr/eeprom.h>
```

这些函数有：

```
eeprom_is_ready()           //EEPROM 忙检测（返回 EEWEIF 位）
eeprom_busy_wait()         //查询等待 EEPROM 准备就绪
uint8_t  eeprom_read_byte (const uint8_t *addr)           //从指定地址读一字节
uint16_t eeprom_read_word (const uint16_t *addr)         //从指定地址一字
void     eeprom_read_block (void *buf, const void *addr, size_t n) //读块
void     eeprom_write_byte (uint8_t *addr, uint8_t val)  //写一字节至指定地址
void     eeprom_write_word (uint16_t *addr, uint16_t val) //写一字到指定地址
void     eeprom_write_block (const void *buf, void *addr, size_t n) //写块
```

在程序中对 EEPROM 操作有两种方式

### 方式一：直接指定 EEPROM 地址

示例：

```
/*
 此程序将 0xaa 写入到 EEPROM 存储器 0 地址处，
 再从 0 地址处读一字节赋给 RAM 变量 val
*/

#include <avr/io.h>
#include <avr/eeprom.h>

int main(void)
{
    unsigned char val;
    eeprom_busy_wait(); //等待 EEPROM 读写就绪
    eeprom_write_byte(0,0xaa); //将 0xaa 写入到 EEPROM 0 地址处
    eeprom_busy_wait();
    val=eeprom_read_byte(0); //从 EEPROM 0 地址处读取一字节赋给 RAM 变量 val

    while(1);
}
```

### 方式二：先定义 EEPROM 区变量法

示例：

```
#include <avr/io.h>
#include <avr/eeprom.h>
```

```
//EEPROM 变量定义
unsigned char val1 __attribute__((section(".eeprom")));

int main(void)
{
    unsigned char val2;

    eeprom_busy_wait();
    eeprom_write_byte (&val1, 0xAA);    // 写 val1
    eeprom_busy_wait();
    val2 = eeprom_read_byte(&val1);    // 读 val1

    while(1);
}
```

在这种方式下变量在 EEPROM 存储器内的具体地址由编译器自动分配。相对方式一，数据在 EEPROM 中的具体位置是不透明的。

为 EEPROM 变量赋的初始值，编译时被分配到.eeprom 段中，可用 avr-objcopy 工具从.elf 文件中提取并产生 ihex 或 binary 等格式的文件，从而可以使用编程器或下载线将其写入到器件的 EEPROM 中。实际上 WINAVR 中 MFILE 生成的 MAKEFILE 已经为我们做了这一切。它会自动生成以“.eep”为后缀的文件，通常它是 iHex 格式。

## 2.6 avr-gcc 段(section)与再定位(relocation)

粗略的讲,一个段代表一无缝隙的数据块(地址范围),一个段里存储的数据都为同一性质,如“只读”数据。as(汇编器)在编译局部程序时总假设从0地址开始,并生成目标文件。最后ld(链接器)在连接多个目标文件时为每一个段分配运行时(run-time)统一地址。这虽然是个简单的解释,却足以说明我们为为什么用段。

ld 将这些数据块正确移动到它们运行时的地址。此过程非常严格,数据的内部顺序与长度均不能发生变化。这样的数据单元叫做段,为段分配运行时地址叫再定位,此任务根据目标文件内的参考地址将段数据调整到运行时地址。

Avr-gcc 中汇编器生成的目标文件(object-file)至少包含四个段,分别为: .text 段、.data 段、.bss 段和.eeprom 段,它们包括了程序存储器(FLASH)代码,内部 RAM 数据,和 EEPROM 存储器内的数据。这些段的大小决定了程序存储器(FLASH)、数据存储器(RAM)、EEPROM 存储器的使用量,关系如下:

程序存储器(FLASH)使用量	= .text + .data
数据存储器(RAM)使用量	= .data + .bss [+ .noinit] + stack [+ heap]
EEPROM 存储器使用量	= .eeprom

### 一 . .text 段

.text 段包含程序实际执行代码。另外,此段还包含.initN 和.finiN 两种段,下面详细讨论这两种段。

段.initN 和段.finiN 是个程序块,它不会象函数那样返回,所以汇编或 C 程序不能调用。.initN、.finiN 和绝对段(absolute section 提供中断向量)构成 avr-libc 应用程序运行框架,用户编写的应用程序在此框架中运行。

### .initN 段

此类段包含从复位到 main()函数开始执行之间的启动(startup)代码。

此类段共定义 10 个分别是.init0 到.init9。执行顺序是从.init0 到.init9。

.init0 :

此段绑定到函数\_\_init()。用户可重载\_\_init(),复位后立即跳到该函数。

.init1 :

未用,用户可定义

.init2 :

将栈指针初始化成器件对应 RAMEND 处,清零\_\_zero\_reg\_\_寄存器

.init3 :

未用,用户可定义

.init4 :

初始化.data 段(从 FLASH 复制全局或静态变量初始值到.data),清零.bss 段。

像 UNIX 一样.data 段直接从可执行文件中装入。Avr-gcc 将.data 段的初始值存储到 flash

rom 里.text 段后，.init4 代码则负责将这些数据复制 SRAM 内.data 段。

.init5：

未用，用户可定义

.init6：

C 代码未用，C++程序的构造代码

.init7：

未用，用户可定义

.init8：

未用，用户可定义

.init9：

跳到 main()

avr-libc 应用程序运行框架是以静态库的形式存储在 <winavr 安装目录>\avr\lib 目录下，在用户程序的连接时它们会自动连接进来。

### **.finiN 段**

此类段包含 main()函数退出后执行的代码。

此类段可有 0 到 9 个，执行次序是从 fini9 到 fini1。

.fini9

此段绑定到函数 exit()。用户可重载 exit()，main 函数一旦退出 exit 就会被执行。

.fini8：

未用，用户可定义

.fini7：

未用，用户可定义

.fini6：

C 代码未用，C++程序的析构代码

.fini5：

未用，用户可定义

.fini4：

未用，用户可定义

.fini3：

未用，用户可定义

.fini2：

未用，用户可定义

.fini1：

未用，用户可定义

.fini0：

进入一个无限循环。

### **用户代码插入到.initN 或.finiN**

可以将用户代码插入到.initN 或.finiN 段当中，这是通过给 C 函数指定段属性来实现的，如下是一段插入到.init1 段的示例：

```
void my_init_portb (void) __attribute__ ((naked)) \
__attribute__ ((section (".init1")));
void my_init_portb (void)
{
    PORTB= 0xff;
    DDRB=0xff;
}
```

使用\_\_attribute\_\_关键字为函数指定属性要在函数的原型声明上，而不是函数的实现例程里。由于属性 section(“.init1”)的指定，编译后函数 my\_init\_portb 生成的代码自动插入到.init1 段中，在 main 函数前就得到执行。naked 属性确保编译后该函数不生成返回指令，使下一个初始化段得以顺序的执行。

## 二 . .data 段

.data 段包含程序中被初始化的 RAM 区全局或静态变量。而对于 FLASH 存储器此段包含在程序中定义变量的初始化数据。类似如下的代码将生成.data 段数据。

```
char err_str[]="Your program has died a horrible death!";
struct point pt={1,1};
```

可以将.data 在 SRAM 内的开始地址指定给连接器，这是通过给 avr-gcc 命令行添加 -Wl,-Tdata,addr 选项来实现的，其中 addr 必须是 0X800000 加 SRAM 实际地址。例如 要将.data 段从 0x1100 开始，则 addr 要给出 0X801100。

## 三 . .bss 段

没有被初始化的 RAM 区全局或静态变量被分配到此段，在应用程序被执行前的 startup 过程中这些变量被清零。

另外 .bss 段有一个子段 .noinit，若变量被指定到.noinit 段中则在 startup 过程中不会被清零。将变量指定到.noinit 段的方法如下：

```
int foo __attribute__ ((section (".noinit")));
```

由于指定到了.noinit 段中，所以不能赋初值，如同以下代码在编译时产生错误：

```
int fol __attribute__ ((section (".noinit"))) = 0x00ff;
```

利用.noinit 段变量可以在复位时判断是否是上电复位，以下是一段示例程序：

```
#include <avr/io.h>
unsigned char rstflag[5] __attribute__ ((section (".noinit"));
```



```
int main(void)
{
    unsigned char j;
    for(j=0;j<5;j++)
    {
        if(rstflag[j]!=j)
            break;
    }
    if(j<5)
    {
        //上电复位
        for(j=0;j<5;j++)
            rstflag[j]=j;
    }
    else
    {
        //其它复位源,说明 RAM 数据没有丢失
    }

    ... ..
}
```

由于 `rstflag` 是 `.noinit` 段变量，复位后 `avr-libc` 应用框架不会初始化该变量，如果没有掉电其内容应该保持，这便是我们判断是否是掉电复位的依据。

#### 四 . `.eeprom` 段

此段存储 EEPROM 变量。

```
static unsigned char eep_buffer[3] __attribute__((section(".eeprom")))= {1,2,3};
```

在连接选项中可指定段的开始地址，如下的选项将 `.noinit` 段指定位到 RAM 存储器 `0X2000` 地址处。

```
avr-gcc ... -Wl,--section-start=.noinit=0x802000
```

要注意的是，在编译时 `Avr-gcc` 将 FLASH、RAM 和 EEPROM 内的段在一个统一的地址空间内处理，flash 存储器被定位到 0 地址开始处，RAM 存储器被定位到 `0x800000` 开始处，eeprom 存储器被定位到 `0X810000` 处。所以在指定段开始地址时若是 RAM 内的段或 eeprom 内的段时要在实际存储器地址前分别加上 `0x800000` 和 `0X810000`。

除上述四个段外，自定义段因需要而可被定义。由于编译器不知道这类段的开始地址，

又称它们为未定义段。必需在连接选项中指定自定义段的开始地址。如下例：

```
void MySection(void) __attribute__((section(".mysection")));
void MySection(void)
{
    printf("hello avr!");
}
```

连接选项：

```
avr-gcc ... -Wl,--section-start=.mysection=0x001c00
```

这样函数 MySection 被定位到了 FLASH 存储器 0X1C00 处。

## 2.7 外部 RAM 的使用

AT90S8515 等 AVR 器件支持外部 RAM，使用外部 RAM 需置位 MCUCR 寄存器的 SRE (SRAM enable)位。从软件上与操作内部 RAM 没有大的区别，从硬件上将使用 RD/WR 读写使能信号，用典型的地址、数据复用端口的时序操作。可参考 51 系列总线相关资料。

## 2.8 堆应用

动态内存分配和堆几乎是同等概念，在局部使用大量的内存时我们通常会使用它，avr-libc 提供两个标准函数 malloc 和 free 支持动态内存分配。有三个全局变量将约束 malloc 的操作，它们是 \_\_malloc\_heap\_start、\_\_malloc\_heap\_end 和 \_\_malloc\_margin。\_\_malloc\_heap\_start 和 \_\_malloc\_heap\_end 分别指向 malloc 可分配内存区域的开始地址和结束地址。

在没用户干预的情况下 GCC 的 RAM 应用规划为：从内部 RAM 的开始处分配 .data 段，之后是 .bss 段，栈指针指向内部 RAM 最顶端，而堆从 .bss 的结束处开始，也就是说 \_\_malloc\_heap\_start 被初始化成 .bss 段结尾地址。\_\_malloc\_heap\_end 被初始化成 0，这时 malloc 函数分配内存时总和当前栈指针保持 \_\_malloc\_margin 个字节距离，\_\_malloc\_margin 被初始化成 32。显然，尽管有这种保护机智，过多的局部变量或过多的函数嵌套会导致堆和栈的碰撞。

用堆的方式使用外部 RAM 是个很好的主意，这时在连接时或运行时设置 \_\_malloc\_heap\_start 和 \_\_malloc\_heap\_end 连接时的配置方式如下：

```
avr-gcc ... -Wl,--defsym=__heap_start=0x802000,--defsym=__heap_end=0x803fff ...
```

可以在运行时直接赋值给 \_\_malloc\_heap\_start 和 \_\_malloc\_heap\_end 的方式将堆指定到外部 RAM 处。

以下是堆应用的简单实例：

```
/*
动态内存分配测试程序
文件名：main.c
硬件:CA-M8
```

```
    时钟：外部 4MHz

    2004-11-28
*/

#include <avr/io.h>
#include <stdio.h>
#include <stdlib.h>

char g_aMemory[256];

//
int usart_putchar(char c)
{
    if(c=='\n')
        usart_putchar('\r');
    loop_until_bit_is_set(UCSRA,UDRE);
    UDR=c;
    return 0;
}
//
int usart_getchar(void)
{
    loop_until_bit_is_set(UCSRA,RXC);
    return UDR;
}

void IoInit(void)
{
    //UART 初始化
    UCSRB=_BV(RXEN)|_BV(TXEN);/*(1<<RXCIE)|(1<<TXCIE)*/
    UBRRL=25; //9600 baud 6MHz:38 4MHz:25

    //i/o 流 UART 连接
    fdevopen(usart_putchar,usart_getchar,0);
}

int main(void)
{
    char *p;
    int i;
    char c;

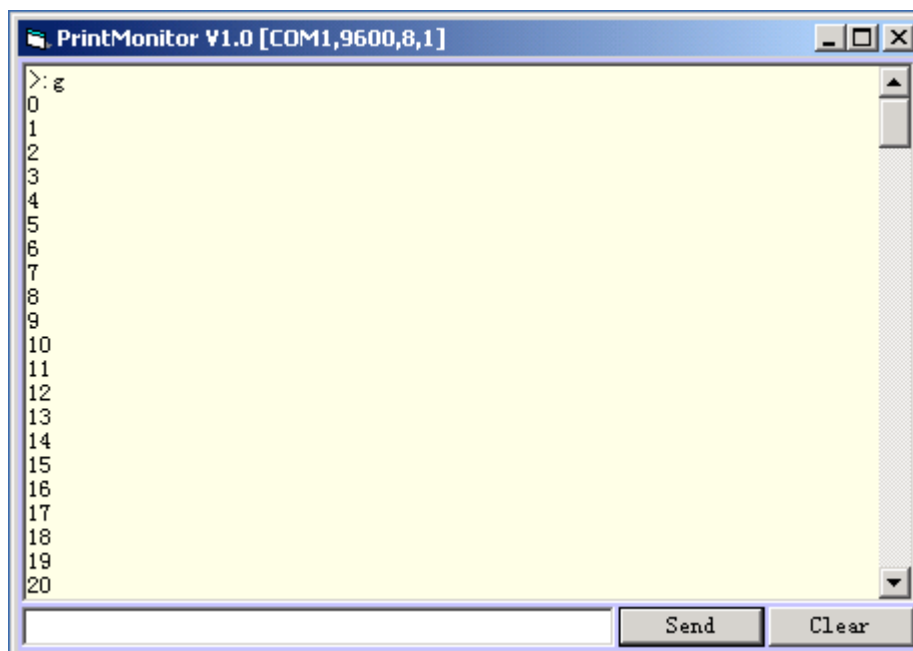
    IoInit();
```

```
// __malloc_heap_start=g_aMemory;
// __malloc_heap_end=g_aMemory+256;

while(1)
{
    p=malloc(100);
    if(p!=0)
    {
        for(i=0;i<100;i++)
            p[i]=i;
        for(i=0;i<100;i++)
            printf("%d\n",p[i]);
        free(p);
    }
    else
        printf("分配内存失败!");
    scanf("%c",&c);
}
}
```

你会得到想象的结果，如下图：

图 2-2 动态内存分配测试结果



关于标准 I/O 输入/输出和异步串行通信在第五章详细描述。

## 第三章 GCC C 编译器的使用

### 3.1 编译基础

这是关于编译 C 程序最基础的理解，也许这些本应该由专门介绍 C 语言书所描述。但出于很多 C 语言书讲的不够直接的现状，还是要简单的介绍一下。我想，对于初学者一定有所帮助。

#### 一. 两种文件

##### C 源文件

以“.c”为扩展名的文件被 GCC 视为 C 源文件。它是包含 C 源程序的文本文件。它们通过预处理器和 C 编译器后生成以.o 为扩展名的目标文件，用于生成最终的可执行文件。

##### C 头文件：

头文件扩展名为“.h”，编译器不会专门编译此文件而生成相应的目标文件。头文件的内容通常是函数原型声明或预处理宏定义。头文件通常使用预处理宏 #include 包含到源文件中。如：

```
#include <avr/io.h>
#include "uart.h"
```

当源程序通过预处理器后这样的语句会被尖括号或引号内指定的头文件内容所替换。语句中使用尖括号时预处理器会从编译器指定的包含目录中查找该文件。而使用引号时预处理器先从当前目录（包括源文件所在目录）中查找指定的文件，找不到才到编译器指定的包含目录查找。尖括号或引号内的斜线符号被解释为路径分隔符。

#### 二. 两种 C 声明

##### 函数原型声明：

在 C 程序中调用一个函数时编译器会根据函数的原型将参数指派到规定的寄存器或压入栈。所以调用函数之前编译器需要知道该函数参数的数据类型、排列以及返回值的情况，函数原型声明就是用于告诉编译器这些信息的目的。函数原型声明在编译器调用该函数之前告知它关于参数和返回值的信息。以下是个函数原型声明的示例：

```
//文件名：main.c

//函数 max 的原型声明
int max(int par1, int par2);

int main(void)
{
    int Ret;
```

```
... ..  
Ret=max(258,330);  
... ..  
}  
  
// 函数 max 的实现代码，返回两个参数中大的一个  
int max(int par1,int par2)  
{  
    return  par1>par2 ? par1:par2;  
}
```

在编译过程中 main 函数不需要知道函数 max 是如何实现的,而要知道 max 的原型用于生成调用代码。为此必须在调用之前声明函数 max 的原型。当然,如果该函数的实现代码在本文件内并且位置在调用函数之前,就不用再声明函数原型了,编译器会自动从实现代码中分析出它的原型。

### 外部变量声明

与函数原型声明类似的道理,外部变量声明告诉编译器:“有这样一个标识存储器的符号,具体位置在哪里就不用管了,连接的时候再说。”

外部变量的声明使用关键字 extern。例如:

```
extern int g_variable;
```

指示了一个数据类型为 int 的全局变量 g\_variable 已在其它源文件中定义。

### 三. 单源文件到可执行程序

从单个源文件生成可执行文件是非常简单的,在第一章中的“一个简单例子”便是个例子,这里就不再多述。

### 四. 从多个源文件到可执行程序

当我们的程序量变得越来越大而在单个源文件中显得杂乱时,你可能希望将代码按用处分别存储到不同的源文件中,之后把这些源文件一起编译成一个可执行文件。例如你想把显示相关的函数放在 display.c 中,把 UART 通信相关的函数放在 uart.c 中,之后在包含入口函数 main 的文件 demo.c 中调用上述两个源文件所实现的函数。这是完全可能的,也是必须这样做的。然而有一个非常坏的主意就是将这些源文件用#include 命令包含到一起再和编译单源文件程序那样编译,通常这样做是没有必要的。

函数原型声明和外部变量声明使得 C 程序的模块化成为了可能,它们是 C 程序从多个源文件生成一个可执行文件的重要手段。要知道,在编译阶段中各个源文件的编译是相互独立的,也就是说生成目标文件(.o)前的对每个源文件的编译过程是互不相干的。下面举一个例子:

假设我们把一个项目中的代码分到了三个源文件 `display.c`、`uart.c` 和 `demo.c` 中，它们的内容分别如下：

`display.c`

```
#include "display.h"
DisplayNumber(uchar num)
{
    //将 num 显示到数码管上
}
```

`uart.c`

```
#include "uart.h"
uchar UartRecvByte(void)
{
    //等待 UART 并接收一字节后返回
}
```

`demo.c`

```
#include "uart.h"
#include "display.h"

int main(void)
{
    uchar c;
    while(1)
    {
        c=UartRecvByte();
        DisplayNumber( c );
    }
}
```

主程序 `main` 调用了 `display.c` 和 `uart.c` 中的函数 `DisplayNumber` 和 `UartRecvByte`。它的功能是从 `UART` 接收一字节后显示到数码管上。你必须现在就知道我是在举例，实际上如果问题真的这么简单就没有必要将代码分到三个文件中。举一个简单的例子是为了便于理解，你可以想象 `display.c` 和 `uart.c` 中有大量的像 `DisplayNumber` 和 `UartRecv` 这样的函数，并

在 demo.c 中被调用。

那么在 demo.c 中 DisplayNumber 和 UartRecvByte 的声明在哪儿呢？通常我们会把声明放到与源文件同名的 .h 文件中。这样我们就用在每一个使用函数的源文件中都要声明，而只需使用#include 命令将头文件包含即可。

Uart.h 和 Display.h 的内容分别如下：

Uart.h

```
#ifndef UART_H
#define UART_H

#include "common.h"
uchar UartRecvByte(void);
#endif
```

Display.h

```
#ifndef DISPLAY_H
#define DISPLAY_H

#include "common.h"

void DisplayNumber(uchar c);

#endif
```

这两个头文件都包含了一个叫 common.h 的文件，这是我用来说明一个问题的。它的内容如下：

common.h

```
#ifndef COMMON_H
#define COMMON_H

#include <avr/io.h>

#define uchar unsigned char
```



```
#endif
```

common.h 包含了 AVR 程序中几乎每个源文件都需要的头文件 io.h ,还定义了在项目其它源文件和头文件都使用的数据类型 uchar。问题是 display.h 和 uart.h 都包含了此文件而 demo.c 将它们两包含进来后会不会导致在 demo.c 中两次包含 common.c ,定义两次的 uchar 而产生错误呢? C 程序头文件通常使用一种预处理宏的技术来保证它在某个源文件中只出现一次。正如上术三个头文件代码所示,将头文件内容被括到一个预处理宏 #ifndef 和 #endif 中间。例如在 demo.c 的预处理过程中,当 demo.c 第一次包含 common.h 时由于符号 COMMON\_H 未定义, #ifndef 条件满足,中间内容便自动插入到 demo.c 中,同时符号 COMMON\_H 被定义,这就保证了下次出现 #include “common.h” 时 #ifndef 这个条件将不再满足,中间内容也就不会再次插入到该源文件中了。

有了上术三个源文件和三个头文件,就可以编译并生成可执行程序了。首先要对每个源文件进行编译生成目标文件(.o), 命令行输入:

```
avr-gcc -mmcu=atmega8 -c display.c
avr-gcc -mmcu=atmega8 -c uart.c
avr-gcc -mmcu=atmega8 -c demo.c
```

以上三行可简化成:

```
avr-gcc -mmcu=atmega8 -c display.c uart.c demo.c
```

这样就生成了所需目标文件 display.o、 uart.o 和 demo.o ,之后将这些目标文件连接到一起便可生成可执行程序。连接命令如下:

```
avr-gcc -mmcu=atmega8 -o demo.elf demo.o uart.o display.o
```

到目前生成的可执行程序是 elf 格式的,如何将它转换成可给器件写入的 HEX 格式或 BIN 格式,我想你应该知道。

罗嗦了这么多,只是为了使 C 基础不太好的读者便于理解,实际的项目中我们会使用 Makefile 脚本,在 Mfile 生成的 makefile 中只要把要编译的源文件都列到 SRC 变量后便可将它们编译并连接到一起。

## 3.2 生成静态连接库

静态连接库是代码重复利用的方式之一。它是编译后生成的目标文件(.o)的集合,在GCC中管理静态库的工具叫ar,相应的在WINAVR中有一可执行文件avr-ar.exe用于生成静态库。

要构造一个静态连接库,先从源文件按正常的编译方式生成目标文件(.o),之后用avr-ar将这些目标文件组合到一起,静态连接库的命名习惯是以“lib”开头,以“.a”为扩展名,avr-libc(GCC AVR 内核C运行时库)都是按这种规则命名的。下面是一个简单的例子它将说明如何生成静态连接库并在应用程序中使用。

文件lib1.c用于生成静态库的源文件一,它仅包一个函数,就是将Mega8 PB1口连接的发光管闪烁一次的FlashLedRed。

```
/*
  文件名：lib1.c
  静态库源程序
  硬件：CA-M8
  2004-11-25
*/

#include <avr/io.h>
#include <avr/delay.h>

#define SET_RED_LED PORTB&=~_BV(PB1)
#define CLR_RED_LED PORTB|=_BV(PB1)

//主程序里的延时函数声明
void DelayMs(unsigned int t);

void FlashLedRed(void)
{
    SET_RED_LED;
    DelayMs(500);
    CLR_RED_LED;
    DelayMs(500);
}
```

文件lib2.c用于生成静态库的源文件二,它同样包一个简单函数FlashLedYel,用于在Mega8 PB0口连接的发光管闪烁一次。

```
/*
```

```
文件名：lib2.c
静态库源程序
硬件：CA-M8
2004-11-25
*/

#include <avr/io.h>
#include <avr/delay.h>

#define SET_YEL_LED PORTB&=~_BV(PB0)
#define CLR_YEL_LED PORTB|=_BV(PB0)

//主程序里的延时函数声明
void DelayMs(unsigned int t);

void FlashLedYel(void)
{
    SET_YEL_LED;
    DelayMs(500);
    CLR_YEL_LED;
    DelayMs(500);
}
```

接下来的工作就是将这两个文件编译后生成一个库文件,之前先看一下调用库里函数的主程序文件,它将调用 FlashLedRed、FlashLedYel 两个函数实现两个发光管不断的闪烁,文件 main.c 的内容如下:

```
/*
文件名：main.c
功 能：静态库测试程序
硬件：CA-M8(ChipArt-Mega8)
编 译: WinAVR-20040720
时 钟：外部 4MHz

更改记录：
2004-11-25
*/

#include <avr/io.h>
#include <avr/delay.h>

#define uchar unsigned char
```

```
#define uint unsigned int

//静态库函数声明
void FlashLedRed(void);
void FlashLedYel(void);

void DelayMs(uint t)
{
    uint i;
    for(i=0;i<t;i++)
        _delay_loop_2(4 * 250);
}

int main(void)
{
    //I/O 初始化
    PORTB=_BV(PB0)|_BV(PB1);
    DDRB=_BV(PB0)|_BV(PB1);

    while(1)
    {
        FlashLedRed();
        FlashLedYel();
    }
}
```

在命令行先选择当前目录为源文件所在目录，输入命令：

```
avr-gcc -c -mmcu=atmega8 lib1.c lib2.c
```

这样就生成了两个文件相应的目标文件 lib1.o 和 lib2.o，再输入：

```
avr-ar -r mylib.a lib1.o lib2.o
```

参数-r 用于指定库文件名，上面的命令将两个目标文件组合到一个库 mylib.a 中。有了库文件就可以在应用中调用库中的函数了。

```
avr-gcc -mmcu=atmega8 main.c mylib.a -o main.elf
```

上面的一行命令将编译 main.c 并与 mylib.a 一起连接后生成可执行文件 main.elf，再用工具 avr-objcopy 从 elf 文件中提取 flash 内的代码生成可写入单片机的 hex 格式文件。命令如下：

```
avr-objcopy -O ihex -R .eeprom main.elf main.hex
```

将 main.hex 写入到 CA-M8，就可以看到红、黄两个发光管不断的闪烁。

如果使用 makefile 编译 main.c 在 Mfile 生成的 makefile 脚本中找到 OBJ 变量并将 mylib.a 列到后边即可。通常我们将有通用功能的函数写到静态库里，便于将来任意一个应用中使用，上面的例子中 lib1.c 和 lib2.c 中均调用了主程序中的函数 DelayMs，这只是为了说明原理，说明这样做是可以的，实际的使用中这种库函数中调用应用程序函数的情况是少见的。

另外，如果你要将你的库交付他人使用，通常还需要写一个说明库中函数原型的头文件，例如象 mylib.a 这样的库，我会写一个如下的头文件与 mylib.a 一起发布。

```
/*文件名： mylib.h*/  
  
#ifndef MYLIB_H  
#define MYLIB_H  
  
void FlashLedRed(void);  
void FlashLedYel(void);  
  
#endif
```

用户会在程序中包含这个头文件并，调用库函数 FlashLedRed、FlashLedYel。

## 第四章 功能模块编程示例

### 4.1 中断服务程序

avr-gcc 为中断提供缺省的入口例程，这些例程的名字已固定，用户可通过重写这些例程来处理中断。如果中断没有被用户重写，说明正常情况下不会产生该中断，缺省的中断例程将程序引导到 0 地址处（既复位）。

Avr-gcc 为重写中断例程提供两个宏来解决细节的问题，它们是 SIGNAL(signame)和 INTERRUPT ( signame )。参数 signame 为中断名称，它的定义在 io.h 中包含。表 4-1 列出了 ATmega8 的 signame 定义，其它器件的 signame 定义可查阅相应的 ioxxx.h 文件。

表 4-1 ATmega8 中断名称定义

signame	中 断 类 型
SIG_INTERRUPT0	外部中断 INTO
SIG_INTERRUPT1	外部中断 INT1
SIG_OUTPUT_COMPARE2	定时器/计数器比较匹配中断
SIG_OVERFLOW2	定时器/计数器 2 溢出中断
SIG_INPUT_CAPTURE1	定时器/计数器 2 输入捕获中断
SIG_OUTPUT_COMPARE1A	定时器/计数器 1 比较匹配 A
SIG_OUTPUT_COMPARE1B	定时器/计数器 1 比较匹配 B
SIG_OVERFLOW1	定时器/计数器 1 溢出中断
SIG_OVERFLOW0	定时器/计数器 0 溢出中断
SIG_SPI	SPI 操作完成中断
SIG_UART_RECV	USART 接收完成
SIG_UART_DATA	USART 寄存器空
SIG_UART_TRANS	USART 发送完成
SIG_ADC	ADC 转换完成
SIG_EEPROM_READY	E <sup>2</sup> PROM 准备就绪
SIG_COMPARATOR	模拟比较器中断
SIG_2WIRE_SERIAL	TWI 中断
SIG_SPM_READY	写程序存储器准备好

以下是个外部中断 0 的重写示例：

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

SIGNAL(SIG_INTERRUPT0)
```

```
{  
    //中断处理程序  
}
```

宏 INTERRUPT 的用法与 SIGNAL 类似，区别在于 SIGNAL 执行时全局中断触发位被清除、其他中断被禁止，INTERRUPT 执行时全局中断触发位被置位、其他中断可嵌套执行。

另外 avr-libc 提供两个 API 函数用于置位和清零全局中断触发位，它们分别是：  
void sei(void) 和 void cli(void)。 它们是经常用到的。

## 4.2 定时器/计数器应用

下面以定时器/计数器 0 为例，说明定时器计数器的操作方法

定时器/计数器 0 相关寄存器：

TCCR0 : 定时器/计数器 0 控制寄存器  
    计数使能，时钟源选择和 CPU 时钟预分频设置  
TCNT0 : 定时器/计数器 0 计数值寄存器  
    包含计数值 (0 ~ 255)  
TIFR : 定时器中断标志寄存器(Timer Interrupt Flag Register)  
    TOV0 位 为定时器/寄存器 0 溢出标志  
TIMSK : 定时器中断屏蔽寄存器(Timer Interrupt Mask Register)  
    TOIE0 位为定时器/寄存器 0 中断使能/禁止控制位

下面两个示例分别使用查询和中断的方式，利用定时器/计数器 0 定时，实现 CA-M8 上的黄色发光管每两秒闪烁一次。应打开 S1-8 ( LEDY )，使黄色发光管连接 PB0 引脚。

**查询模式举例：**

```
/*  
  ATmega8 T/C0 查询方式测试程序  
  文件名：main.c  
  硬件：CA-M8  
  时钟：外部 4MHz  
  
  芯艺 2004-12-01  
*/  
  
#include <avr/io.h>
```

```
#define uchar unsigned char

#define SET_LED PORTB&=0XFE //PB0 接黄色发光管
#define CLR_LED PORTB|=0X01

int main(void)
{
    uchar i,j=0;

    //设置 PB0 口为输出
    DDRB=_BV(PB0);
    PORTD=_BV(PB0);

    //配置 T/C0
    TCNT0=0; // T/C0 开始值
    TCCR0=_BV(CS02)|_BV(CS00); // 预分频 ck/1024 ,计数允许

    while(1)
    {
        //查询定时器方式等待一秒
        //4000000 /1024 /256 /15    1Hz
        for(i=0;i<15;i++)
        {
            loop_until_bit_is_set(TIFR,TOV0);
            //写入逻辑 1 清零 TOV0 位
            TIFR|=_BV(TOV0);
        }

        if(j) //反向 LED 控制脚
            SET_LED, j=0;
        else
            CLR_LED, j=1;
    }
}
```

### 中断模式举例：

```
/*
ATMega8 T/C0 中断方式测试程序
文件名：main.c
硬件:CA-M8
时钟：外部 4MHz
```



```
芯艺 2004-12-01
*/

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

#define uchar unsigned char

#define SET_LED PORTB&=0XFE //PB0 接黄色发光管
#define CLR_LED PORTB|=0X01

static uchar g_bCount=0; //中断计数器
static uchar g_bDirection=0;

//T/C0 中断例程
SIGNAL(SIG_OVERFLOW0)
{
    // 产生中断周期 T = 256 * 1024 / 4MHz
    // T * 15 约等于 1 秒
    if(++g_bCount > 14) //中断 15 次约一秒
    {
        if(g_bDirection) //反向 LED 控制脚
            SET_LED,g_bDirection=0;
        else
            CLR_LED,g_bDirection=1;

        g_bCount=0;
    }
}

int main(void)
{
    //PB0 初始化成输出
    DDRB=_BV(PB0);
    PORTB=_BV(PB0);

    TCNT0=0; // T/C0 开始值
    TCCR0=_BV(CS02)|_BV(CS00); // 预分频 ck/1024 ,计数允许

    TIMSK=_BV(TOIE0); //T/C0 中断允许
    sei(); //总中断标志置位

    while(1);
}
```

}

### 4.3 看门狗应用

avr-libc 提供三个 API 支持对器件内部 Watchdog 的操作，它们分别是：

```
wdt_reset()           // Watchdog 复位
wdt_enable(timeout)  // Watchdog 使能
wdt_disable()        // Watchdog 禁止
```

调用上述函数前要包含头文件 wdt.h，wdt.h 中还定义 Watchdog 定时器超时符号常量，它们用于为 wdt\_enable 函数提供 timeout 值。符号常量分别如下：

符号常量	值含意
WDTO_15MS	Watchdog 定时器 15 毫秒超时
WDTO_30MS	Watchdog 定时器 30 毫秒超时
WDTO_60MS	Watchdog 定时器 60 毫秒超时
WDTO_120MS	Watchdog 定时器 120 毫秒超时
WDTO_250MS	Watchdog 定时器 250 毫秒超时
WDTO_500MS	Watchdog 定时器 500 毫秒超时
WDTO_1S	Watchdog 定时器 1 秒超时
WDTO_2S	Watchdog 定时器 2 秒超时

#### Watchdog 测试程序：

在 CA-M8 上执行此程序时应打开 S1-8 (LEDY)，使黄色发光管连接 PBO 引脚。

```
/*
  Watchdog 测试程序
  main.c
  硬件: CA-M8
  时钟 : 4MHz

  芯艺 2004-12-01
*/

#include <avr/io.h>
#include <avr/wdt.h>
#include <avr/delay.h>

#define uchar unsigned char
#define uint unsigned int

#define SET_LED PORTB&=0XFE //PBO 接黄色发光管
```

```
#define CLR_LED PORTB|=0X01

//误差不会太大的延时 1ms 函数
void DelayMs(uint ms)
{
    uint i;
    for(i=0;i<ms;i++)
        _delay_loop_2(4 *250);
}

int main(void)
{
    DDRB=_BV(PB0);
    PORTB=_BV(PB0); //CLR_LED
    //WDT 计数器同期为一秒
    wdt_enable(WDTO_1S);
    wdt_reset();//喂狗

    DelayMs(500);

    SET_LED;

    //等待饿死狗
    DelayMs(5000);

    SET_LED;
    while(1)
        wdt_reset();
}
```

执行结果：

CA-M8 上的黄色发光管不断的闪烁，证明了 Watchdog 使 MCU 不断的复位。

## 4.4 UART 应用

下面是 UART 功能操作的两个不同方式示例：

在 CA-M8 上做此实验时可打开 S5-5(RXD)和 S5-6(TXD)使 MEGA8 的 RXD、TXD 引脚接到 MAX202 电平转换电路上。

**查询方式:**

```
/*
  查询方式 UART 测试程序
  文件名：main.c
  硬 件：CA-M8
  时 钟：外部 4MHz

  芯艺 2004-12-01
*/

#include <avr/io.h>

#define uchar unsigned char
#define uint unsigned int

void putc(uchar c)
{
  while( !(UCSRA & (1<<UDRE)) );
  UDR=c;
}

uchar getc(void)
{
  while( !(UCSRA & (1<<RXC)) );
  return UDR;
}

int main(void)
{
  //uart 初始化
  UBRRH=0;
  UBRRL=25;//9600 baud 6MHz:38 4MHz:25
  UCSRB=(1<<RXEN)|(1<<TXEN);

  while(1)
  {
```

```
    putc(getc());
}
}
```

程序从 UART 等待接收一字节，接收到数据后立即将数据又从 UART 发送回去。

### 中断方式：

```
/*
  中断方式 UART 测试程序
  main.c
  硬件:CA-M8
  时钟:外部 4MHz

  芯艺 2004-12-01
*/

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

#define uchar unsigned char
#define uint unsigned int

uchar g_bTxdPos=0; //发送定位计数器
uchar g_bTxdLen=0; //等待发送字节数
uchar g_bRxdPos=0; //接收定位计数器
uchar g_bRxdLen=0; //等待接收字节数

uchar g_aSendBuf[16]; //发送数据缓冲区
uchar g_aRecvBuf[16]; //接收数据缓冲区

//接收中断
SIGNAL(SIG_UART_RECV)
{
    uchar c=UDR;
    if(g_bRxdLen>0)
    {
        g_aRecvBuf[g_bRxdPos++]=c;
        g_bRxdLen--;
    }
}

//发送中断
```

```
SIGNAL (SIG_UART_TRANS)
{
    if(--g_bTxdLen>0)
        UDR=g_aSendBuf[++g_bTxdPos];
}

//是否接收完成
uchar IsRecvComplete(void)
{
    return g_bRxdLen==0;
}

//从发送缓冲区发送指定长度数据
void SendToUart(uchar size)
{
    g_bTxdPos=0;
    g_bTxdLen=size;
    UDR=g_aSendBuf[0];
    while(g_bTxdLen>0);
}

//接收指定长度数据到接收缓冲区
void RecvFromUart(uchar size,uchar bwait)
{
    g_bRxdPos=0;
    g_bRxdLen=size;
    if(bwait)
        while(g_bRxdLen>0);
}

int main( void )
{
    uchar i;

    //uart 初始化
    //接收使能、发送使能、接收中断允许、发送中断允许
    UCSRB=(1<<RXCIEN)|(1<<TXCIEN)|(1<<RXEN)|(1<<TXEN);
    UBRR0L=25; // baud=9600    UBRR0=CK/(baud*16) -1

    sei();//总中断允许

    while(1)
    {
        //异步接收 16 字节数据
```

```
    RecvFromUart(16,0);

    //等待接收完成
    while(!IsRecvComplete());

    //将接收到的数据复制到发送缓冲区
    for(i=0;i<16;i++)
        g_aSendBuf[i]=g_aRecvBuf[i];

    //发送回接收到的数据
    SendToUart(16);
}
}
```

利用中断可实现数据的异步发送和接收，正如上面程序所示，调用 RecvFromUart 后主程序可处理其它任务，在执行其它任务时可调用 IsRecvComplete 检测是否接收完成。

### 4.5 PWM 功能编程

Atmega8 共有三个 PWM 信号输出引脚（OC1A、OC1B、OC2），其中 OC1A 和 OC1B 的应用将占用定时器/计数器 1。下面的示例演示了循环改变接在 OC1A（PB1）上的红色发光管亮度。

在 CA-M8 上执行此程序时应打开 S1-7（LEDR），使红色发光管连接 OC1A 引脚。

```
/*
   ATMega8 PWM 功能(T/C 1)测试程序
   main.c
   硬件:CA-M8
   时钟:4MHz

   芯艺 2004-12-01
*/

#include <avr/io.h>
#include <avr/delay.h>

#define uchar unsigned char
#define uint unsigned int

#define FREQ 4

//延时
void DelayMs(uint ms)
{
    uint i;
```

```
    for(i=0;i<ms;i++)
        _delay_loop_2(FREQ * 250);
}

int main (void)
{
    uchar direction=1;
    uchar pwm=0;

    // 8位相位修正 PWM 模式
    // OC1A 向上计数过程中比较匹配时清零
    // 向下计数过程中比较匹配时置位
    // 时钟源：4MHz/64  PWM 频率：4MHz/(64*256)=122Hz
    TCCR1A=_BV(COM1A1)*_BV(COM1B1)*_BV(WGM10);
    TCCR1B=_BV(CS11)|_BV(CS10);

    TCNT1=0;
    DDRB|=_BV(PB1)*_BV(PB2); //OC1A pwm 输出引脚允许

    //循环改变 PWM 输出脉宽，使接在 OC1A 引脚上的发光管亮度发生变化
    while(1)
    {
        if(direction)
        {
            if(++pwm==254)
                direction=0;
        }
        else
        {
            if(--pwm==0)
                direction=1;
        }

        OCR1A=pwm;
        //OCR1B=pwm;
        DelayMs(10);
    }

    return 0;
}
```



## 4.6 模拟比较器

在 CA-M8 上没有专门的模拟比较器测试电路,从外部电位器调节模拟比较器输入时应关闭 S6-7。

```
/*
    模拟比较器测试程序
    MCU：ATMega8
    时钟：内部 4MHz RC 振荡器
*/

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

#define uchar unsigned char

#define SET_RED_LED PORTB&=0XFD //PB1 接红色发光管
#define CLR_RED_LED PORTB|=0X02

#define SET_YEL_LED PORTB&=0XFE //PB0 接黄色发光管
#define CLR_YEL_LED PORTB|=0X01

//模拟比较器中断函数
SIGNAL(SIG_COMPARATOR)
{
    if(ACSR & _BV(ACO))
    {
        SET_YEL_LED;
        CLR_RED_LED;
    }
    else
    {
        CLR_YEL_LED;
        SET_RED_LED;
    }
}

int main(void)
{
    DDRB=0X03;
    PORTB=0X03;

    //模拟比较器上下均触发中断 ACIS1=ACIS0=0
```

```
//中断允许 ACIE=1
ACSR=_BV(ACIE);

sei();

//AIN0:正极 AIN1:负极 AIN0 脚上的电压高于 AIN1 上电压时 AC0=1
if(ACSR & _BV(ACO))
{
    SET_YEL_LED;
    CLR_RED_LED;
}
else
{
    CLR_YEL_LED;
    SET_RED_LED;
}

while(1);
}
```

以上程序实现了用 LED 指示 ATmega8 比较输入引脚 AIN0 和 AIN1 上的电压的高低状态。

### 4.7 A/D 转换模块编程

以下是两种不同的操作方式示例：

在 CA-M8 上使用电位器 R5 调节输入电压时应打开 S1-6，S1-5 是引脚 AREF 是否接 VCC 的开关，下边的程序使用了内部 2.56V 基准电压，所以此开关需关闭。

**查询方式：**

```
/*
Mega8 查询方式 A/D 转换测试程序
文件名:main.c
硬件:CA-M8
编译：WinAVR-20040720

芯艺 2004-09-19
*/

#include <avr/io.h>
#include <avr/delay.h>
#include <stdio.h>

#define uchar unsigned char
```

```
#define uint    unsigned int

static uint g_aAdValue[8];    //A/D 转换缓冲区

void IoInit(void);

uint AdcConvert(void)
{
    uchar i;
    uint ret;
    uchar max_id,min_id,max_value,min_value;

    ADMUX=0Xc0;//内部 2.56V 参考电压，0 通道
    ADCSRA=_BV(ADEN);//使能 ADC，单次转换模式

    //连续转换 8 次
    for(i=0;i<8;i++)
    {
        ADCSRA|=_BV(ADSC);
        _delay_loop_1(60);
        while(ADCSRA&_BV(ADSC))
            _delay_loop_1(60);
        ret=ADCL;
        ret|=(uint)(ADCH<<8);
        g_aAdValue[i]=ret;
    }
    ret=0;
    for(i=1;i<8;i++)
        ret+=g_aAdValue[i];

    //找到最大和最小值索引
    ret/=7;
    max_id=min_id=1;
    max_value=min_value=0;
    for(i=1;i<8;i++)
    {
        if(g_aAdValue[i]>ret)
        {
            if(g_aAdValue[i]-ret>max_value)
            {
                max_value=g_aAdValue[i]-ret;
                max_id=i;
            }
        }
    }
}
```

```
    }
    else
    {
        if(ret-g_aAdValue[i]>min_value)
        {
            min_value=ret-g_aAdValue[i];
            min_id=i;
        }
    }
}

//去掉第一个和最大最小值后的平均值
ret=0;
for(i=1;i<8;i++)
{
    if((i!=min_id)&&(i!=max_id))
        ret+=g_aAdValue[i];
}
if(min_id!=max_id)
    ret/=5;
else
    ret/=6;

ADCSRA=0;//关闭 ADC

return ret;
}

int main(void)
{
    uchar i;
    IoInit();

    while(1)
    {
        scanf("%c",&i);
        if(i=='c')
            printf("%d\n",AdcConvert());
    }
}
```

中断方式：

```
/*
Mega8 中断方式 A/D 转换测试程序
main.c
硬件：CA-M8
编译：WinAVR-20040720

芯艺 2004-09-19
*/

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>
#include <avr/delay.h>
#include <stdio.h>

#define uchar unsigned char
#define uint unsigned int

static uint g_nAdValue=0;

void IoInit(void);

// A/D 转换完成中断
SIGNAL(SIG_ADC)
{
    g_nAdValue=ADCL;
    g_nAdValue|=(uint)(ADCH<<8);
}

int main(void)
{
    uchar i;

    //内部 2.56V 参考电压，0 通道
    ADMUX=0Xc0;

    //使能 ADC,中断允许,自由模式, 时钟：ck/8
    ADCSRA=_BV(ADEN)|_BV(ADIE)|_BV(ADFR)|_BV(ADPS1)|_BV(ADPS0);

    IoInit();//标准输入/输出初始化

    ADCSRA|=_BV(ADSC);//自由模式开始转换
```

```
while(1)
{
    //延时
    for(i=0;i<100;i++)
        _delay_loop_2(4 * 250 * 10);//10ms

    cli();
    printf("%d\n",g_nAdValue);
    sei();
}
}
```

以上是 ATmega8 A/D 转换程序的两种方式,在第一种查询方式中 ADC 按单次转换模式工作,每次转换均由置位 ADSC 触发。在中断方式示例中 ADC 按自由模式工作,自第一次置位 ADSC 起 ADC 就连续不断的进行采样转换、进行数据更新。

这两段程序的执行结果可由串口监测工具 PrintMonitor (第四章 详细介绍) 观测。

### 4.8 数码管显示程序设计

数码管显示分动态和静态两种方式,动态显示接线较少但占用 CPU 资源较多,相反、静态显示具有程序简单,却占用 I/O 口多的特点。动态显示是单片机教学经典的命题,我不再描述。CA-M8 上用于显示数的两个数码管用 74hc595 驱动,属于静态显示,这样的电路在工程中是非常常见的。

74hc595 是个三态输出 8 位移位寄存器/锁存器,多个 595 可以串连使用,非常适合数码管的静态驱动。SER、SCK 和 RCK 三个引脚用于控制 595 进行输入和锁存操作。SCK 是时钟输入引脚,在 SCK 的上升沿 SER 上的电平信号被读入内部寄存器最低位。RCK 是锁存控制引脚,在它的上升沿内部寄存器的内容被锁存到与输出引脚相对应的锁存器中。

在 CA-M8 上 Mega8 的 PD4 ~ PD6 控制这三个引脚,为此在做显示实验时需将拨动开关 S6 的 5、6、7 闭合。

表 4-2 是根据 CA-M8 显示电路列出的显示数 0 ~ 0xf 时所要写入 595 的显示码表。

表 4-2 数码管显示码

595	QH	QG	QF	QE	QD	QC	QB	QA	十六进制
SEG	dp	g	f	e	d	c	b	a	
0	0	0	1	1	1	1	1	1	3F
1	0	0	0	0	0	1	1	0	06
2	0	1	0	1	1	0	1	1	5B
3	0	1	0	0	1	1	1	1	4F
4	0	1	1	0	0	1	1	0	66
5	0	1	1	0	1	1	0	1	6D
6	0	1	1	1	1	1	0	1	7D
7	0	0	0	0	0	1	1	1	07
8	0	1	1	1	1	1	1	1	7F
9	0	1	1	0	1	1	1	1	6F
A	0	1	1	1	0	1	1	1	77
B	0	1	1	1	1	1	0	0	7C
C	0	0	1	1	1	0	0	1	39
D	0	1	0	1	1	1	1	0	5E
E	0	1	1	1	1	0	0	1	79
F	0	1	1	1	0	0	0	1	71

测试程序：

```

/*
控制 595 驱动的数码管显示程序
main.c
硬件:CA-M8
时钟:4MHz

芯艺 2004-12-16
*/

#include <avr/io.h>
#include <avr/delay.h>

#define uchar unsigned char
#define uint unsigned int

#define FREQ 4

#define SER_PORT PORTD
#define SER_DAT PD4
    
```

```
#define SER_RCK PD5
#define SER_SCK PD6

//显示码
uchar g_aDisplayBuf[16]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07, \
    0x7f,0x6f,0x77,0x7c,0x39,0x5e,0x79,0x71};

void DelayMs(uint t)
{
    uint i;
    for(i=0;i<t;i++)
        _delay_loop_2(250 * FREQ);
}

//向 595 写一字节
void ser_out(uchar dat)
{
    uchar i;

    for(i=0;i<8;i++)
    {
        if(dat&0x80)
            SER_PORT|=_BV(SER_DAT);
        else
            SER_PORT&=~_BV(SER_DAT);

        SER_PORT|=_BV(SER_SCK);
        SER_PORT&=~_BV(SER_SCK);
        dat<<=1;
    }
}

//数码管显示数(0~255)
//num :显示的数    hex : 是否用十六进制显示
void display_number(uchar num,uchar hex)
{
    uchar buf[2];//发送显示码缓冲区
    uchar temp;

    if(hex) //十六进制
    {
        buf[0]=g_aDisplayBuf[num>>4];//高位
        buf[1]=g_aDisplayBuf[num&0x0f];//低位
    }
    else//十进制
```



```
{
    buf[1]=g_aDisplayBuf[num%10];
    temp=num%100;
    buf[0]=g_aDisplayBuf[temp/10];
    temp=num/100;

    if(temp>0)
        buf[1]=0x80; //第一个数码管小数点表示百位 1
    if(temp>1)
        buf[0]=0x80; //两个数码管小数点表示百位 2
}

//串行发送数据
ser_out(buf[0]);
ser_out(buf[1]);

//显示
SER_PORT|=_BV(SER_RCK);
SER_PORT&=~_BV(SER_RCK);
}

int main(void)
{
    uchar i=0;
    DDRD=_BV(SER_DAT)|_BV(SER_SCK)|_BV(SER_RCK);
//    DDRB=_BV(PB0)|_BV(PB1);

    while(1)
    {
        display_number(i++,1);
        DelayMs(1000);
    }
}
```

### 4.9 键盘程序设计

CA-M8 上两个按键开关可以直接通过开关 S5-7 和 S5-8 连接到 MEGA8 的 PD2 和 PD3 能上引脚上，要注意的是当开关 S5-7 和 S5-8 被打开时不能将 PD2 和 PD3 设置成输出并置位。

#### 查询方式：

以下程序功能为：当按 S3 时显示的数据加一，当按 S4 时数据的显示在十进制和十六进制间切换。

```
/*
    查询方式测试键盘程序
    main.c
    硬件:CA-M8
    时钟:4MHz

    芯艺 2004-12-17
*/

#include <avr/io.h>
#include <avr/delay.h>

#define uchar unsigned char
#define uint unsigned int

#define SER_PORT PORTD
#define SER_DAT PD4
#define SER_RCK PD5
#define SER_SCK PD6

#define KEY0 PD2
#define KEY1 PD3

//显示码
uchar
g_aDisplayBuf[16]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f,0x77,0x7c,0x39,、
    0x5e,0x79,0x71};

void DelayMs(uint t);
void display_number(uchar num,uchar hex);

//扫描键盘
uchar GetKey(void)
{
    uchar ret=0;
    while(1)
    {
        if(!(PIND&_BV(KEY0)))
        {
            //延时消抖
            DelayMs(5);
            if(!(PIND&_BV(KEY0)))
            {
```

```
        ret=0;
        break;
    }
}
if(!(PIND&_BV(KEY1)))
{
    //延时消抖
    DelayMs(5);
    if(!(PIND&_BV(KEY1)))
    {
        ret=1;
        break;
    }
}
DelayMs(5);
}
//等待弹起
while((PIND & (_BV(KEY0)|_BV(KEY1)))!=(_BV(KEY0)|_BV(KEY1)))
    DelayMs(1);
return ret;
}

int main(void)
{
    uchar i=0,flag=0,key;
    DDRD=_BV(SER_DAT)|_BV(SER_SCK)|_BV(SER_RCK);
    PORTD=_BV(KEY0)|_BV(KEY1);//键盘引脚上拉电阻开
//    DDRB=_BV(PB0)|_BV(PB1);

    display_number(i,flag);
    while(1)
    {
        key=GetKey();
        switch(key)
        {
            case 0:        //显示数据加一
                display_number(++i,flag);
                break;
            case 1:        //十进制和十六进制间切换
                flag=!flag;
                display_number(i,flag);
                break;
            default:
                break;
        }
    }
}
```

```
    }  
  }  
}
```

### 中断方式：

此程序简单实现了按下 S3 时显示的数据加一的功能。

```
/*  
  中断方式测试键盘程序  
  文件名：main.c  
  硬件:CA-M8  
  时钟:4MHz  
  
  芯艺 2004-12-17  
*/  
  
#include <avr/io.h>  
#include <avr/delay.h>  
#include <avr/signal.h>  
#include <avr/interrupt.h>  
  
#define uchar unsigned char  
#define uint  unsigned int  
  
#define SER_PORT    PORTD  
#define SER_DAT PD4  
#define SER_RCK PD5  
#define SER_SCK PD6  
  
#define KEY0 PD2  
#define KEY1 PD3  
  
//显示码  
uchar g_aDisplayBuf[16]={0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f,0x77,\  
    0x7c,0x39,0x5e,0x79,0x71};  
uchar g_bCounter=0;  
  
void DelayMs(uint t);  
void display_number(uchar num,uchar hex);  
  
//外部中断 0  
SIGNAL(SIG_INTERRUPT0)  
{
```

```
    DelayMs(100);
    display_number(++g_bCounter,0);
    GIFR|=_BV(INTF0);//写 1 清除中断标志,消抖
}

int main(void)
{
    DDRD=_BV(SER_DAT)|_BV(SER_SCK)|_BV(SER_RCK);
    PORTD=_BV(KEY0)|_BV(KEY1);//键盘引脚上拉电阻开

    MCUCR=_BV(ISC00)|_BV(ISC01);//上升沿触发
    GICR=_BV(INT0);//外部中断 0 允许
    sei();           //总中断允许

    display_number(g_bCounter,0);
    while(1);
}
```

### 4.10 蜂鸣器控制

当 S6-8 开关打开时 PB2 引脚用于控制板上蜂鸣器。让蜂鸣器发出一个简短的响声十分简单，程序如下：

```
void Beep(void)
{
    PORTB|=_BV(PB2);
    DelayMs(100);
    PORTB&=!_BV(PB2);
}
```

执行 Beep 函数前应当将 PB2 设置为输出。

关于 CA-M8 上更多的测试程序请关注 <http://www.chip-art.com> 或 <http://chipart.17kl.net>。

## 第五章 使用 C 语言标准 I/O 流调试程序

### 5.1 avr-libc 标准 I/O 流描述

avr-libc 提供标准 I/O 流 `stdin`, `stdout` 和 `stderr`。但受硬件资源的限制仅支持标准 C 语言 I/O 流的部分功能。由于没有操作系统支持, avr-libc 又不知道标准流使用的设备, 在应用程序的 startup 过程中 I/O 流无法初始化。同样在 avr-libc 中没有文件的概念, 它也不支持 `fopen()`。做为替代 `fdevopen()` 提供流与设备间的连接。 `fdevopen` 需要提供字符发送、字符接收两个函数, 在 avr-libc 中这两个函数对于字符流与二进制流是没有区别的。

#### 三个核心函数

#### `fdevopen()`

应用程序通过 `fdevopen` 函数为流指定实际的输入输出设备。

```
FILE* fdevopen( int(* put) (char), int(* get)(void), int opts __attribute__((unused))
```

前两个参数均为指向函数的指针, 它们指向的函数分别负责向设备输出一字节和从设备输入一字节的功能。第三个参数保留, 通常指定 0。

如果只指定 `put` 指针, 流按写方式打开, `stdout` 或 `stderr` 成为流的引用名。

如果只指定 `get` 指针, 流按只读方式打开, `stdin` 成为流的引用名。

如果在调用时两者都提供则按读写方式打开, 此时 `stdout`、`stderr` 和 `stdin` 相同, 均可做为当前流的引用名。

(1) 向设备写字符函数:

原型:

```
int put(char c)
{
    ... ..
    return 0;
}
```

返回 0 表示字符传送成功, 返回非零表示失败。

另外, 字符 '`\n`' 被 I/O 流函数传送时直接传送一个换行字符, 因此如果设备在换行前需要回车, 应当在 `put` 函数里发送 '`\n`' 前发字符 '`\r`'。

以下是一个基于UART的 put 示例：

```
int uart_putchar(char c)
{
    if(c=='\n')
        uart_putchar('\r');
    loop_until_bit_is_set(UCSRA,UDRE);
    UDR=c;
    return 0;
}
```

### (2) 从设备输入字符函数

原型：

```
int get(void)
{
    ... ..
}
```

get 函数从设备读取一字节并按 int 类型返回，如果读取时发生了错误需返回 -1。

## fprintf()

```
int fprintf ( FILE * __stream, const char * __fmt, va_list __ap )
```

fprintf 将\_\_ap列出的值按\_\_fmt 字符的形式输出到流\_\_stream。返回输出字节数，若产生错误返回EOF。

fprintf是libc提供的I/O流格式化输出函数的基础,在应用程序中很少调用此函数,为了避免应用中用不到的功能占用宝贵的硬件资源,vfprintf函数支持三种不同链接模式。

(1) 在默认情况下它包含除浮点数格式转换外的所有功能

(2) 最小模式仅包含基本整数类型和字符串转换功能  
要用最小模式链接此函数，使用的链接选项如下：  
-Wl,-u,vfprintf -lprintf\_min

(3) 完全模式支持浮点数格式转换在内的所有功能。

完全模式链接选项如下：

```
-Wl,-u,vfprintf -lprintf_flt -lm
```

### vfscanf()

```
int vfscanf ( FILE * __stream, const char * __fmt, va_list __ap )
```

vfscanf 是 libc 提供的 I/O 流格式化输入函数的基础,在应用程序中很少直接调用。它从\_\_stream 流按字符格式读取\_\_fmt 内容后按转换规则将数据保存到参数列表\_\_ap 内。与vfprintf 类似 vfscanf 也支持三种不同链接模式。

(1)在默认情况下它支持除浮点数格式和格式“%[ ”外的所有转换。

(2)最小模式链接选项：

```
-Wl,-u,vfscanf -lscanf_min -lm
```

(3)完全模式链接选项：

```
-Wl,-u,vfscanf -lscanf_flt -lm
```

## 5.2 利用标准I/O流调试程序

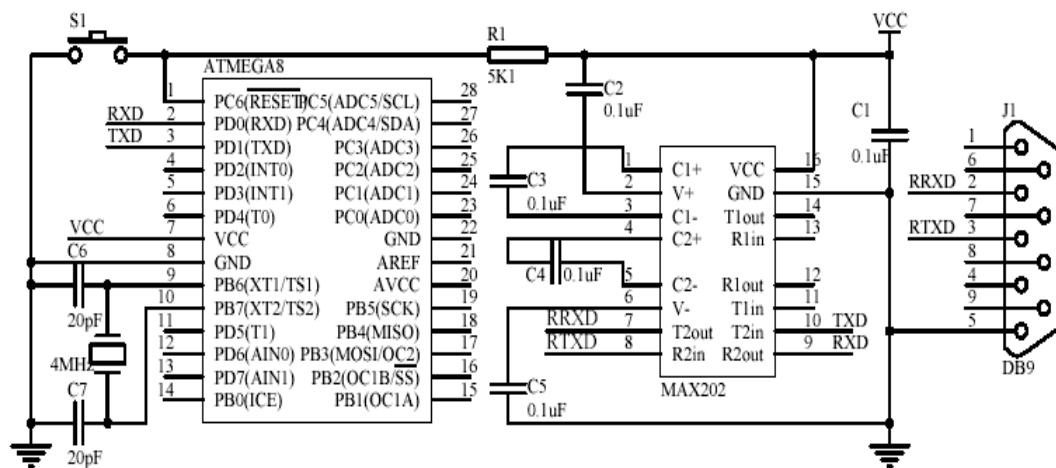
在程序的调试阶段,提供数据格式化输入/输出功能的标准I/O函数是个非常有用的工具,而单片机UART(通用异步收发器)接口是标准I/O的比较合适的设备,通过它可以很容易的将数据传送到PC机,用PC机友好而强大的功能观测程序的执行情况。

### 一. 基础电路

由于 PC 机串行口采用的是 RS232 标准,单片机与 PC 机串行口间的数据通信要有一个电平转换的接口电路,图 5-1 为采用 MAXIM 公司的 MAX202 实现的接口电路,MAX202 支持两路全双工 UART 的电平转换,T1in 和 T2in 为 TTL/CMOS 输入端,应接单片机的 TXD 口,T1in 和 T2in 的输入电平转换后分别从 T1out 和 T2out 输出,T1out 和 T2out 应接 PC 串行口 RXD 引脚(DB9 连接器第 2 脚)。R1in 和 R2in 是 RS232 输入端,应接 PC 串行口 TXD 引脚(DB9 连接器第 3 脚),信号通过电平转换后分别从 R1out 和 R2out 输出,R1out 和 R2out 应接单片机 RXD 引脚。

图 5-1 UART 实现 I/O 流电路原理图





## 二 . 测试程序

使用 avr-libc 的 I/O 函数，需在程序中包含声明文件 stdio.h。要注意的是 stdio 并不在 avr 目录下。以下是一段基于上述电路的两种类型输入/输出函数的测试程序：

```

/*
    avr-libc 标准 i/o 流测试程序
    文件名：main.c
    硬件：CA-M8 实验板
    时钟：外部 4MHz

    2004-11-22
*/

#include <avr/io.h>
#include <avr/pgmspace.h>
#include <stdio.h>

char g_aString[81]; //读取字符串缓冲区

//向 UART 写一字节
int usart_putchar(char c)
{
    if(c=='\n')
        usart_putchar('\r');
    loop_until_bit_is_set(UCSRA,UDRE);
    UDR=c;
    return 0;
}
    
```

```
//从 UART 读一字节
int usart_getchar(void)
{
    loop_until_bit_is_set(UCSRA,RXC);
    return UDR;
}
//初始化 i/o
void lolnit(void)
{
    //UART 初始化
    UCSRB=_BV(RXEN)|_BV(TXEN); /* (1<<RXCIEN)|(1<<TXCIEN) | */
    UBRR1L=25; //9600 baud 6MHz:38 4MHz:25

    //i/o 流 UART 连接
    fdevopen(usart_putchar,usart_getchar,0);
}
//测试主程序
int main(void)
{
    int tmp;

    lolnit();

    while(1)
    {
        //测试 1
        printf("测试 1[输入一字符串]:\n");
        scanf("%s",g_aString);
        printf("您的输入为:%s\n",g_aString);

        //测试 2:使用 FLASH 区的字符串
        printf_P(PSTR("测试 2[输入数字]:\n"));
        scanf_P(PSTR("%d",&tmp));
        printf_P(PSTR("您的输入为:%d\n"),tmp);

    }
}
```

程序在 lolnit 例程中对 UART 接口初始化并使用 fdevopen 函数将 I/O 流连接到 UART 设备。printf/scanf 的格式化字符串将占用 RAM，大量的使用会耗尽有限的片内 RAM。而 printf\_P/scanf\_P 是我们更多的选择，它的格式化字符串要求是在 Flash 内的数据，为此应当借助宏 PSTR 来指定，使用 PSTR 必需包含其定义文件 pgmspace.h。

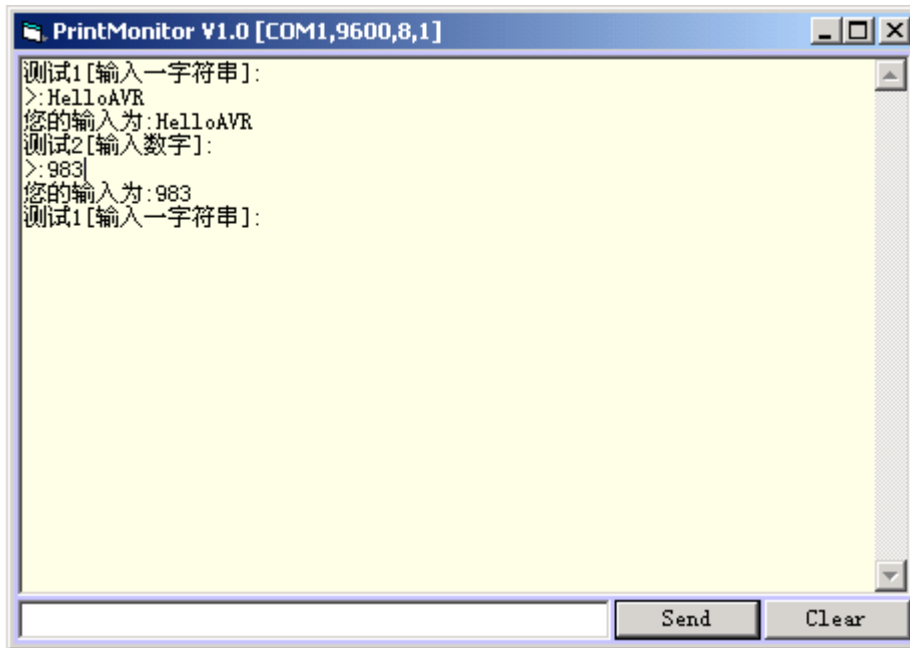
### 三 . 监测工具

目前在网络上 Windows 系统上的串行口调试程序很多,它们均属于 Windows 应用程序,Windows 为串行口的读写提供了缺省的驱动程序,在 C(或 C++)程序中可以使用 Windows API CreateFile 打开符号链接“COMX”(如 COM1、COM2)的方式来进行读写操作。而在 VisualBasic 这样的应用程序开发软件中控件 MsComm 是首选。

监测工具 PrintMonitor 就是一个使用 MsComm 的应用程序,运行界面如图 5-2 所示,它的 VisualBasic 源代码可到 <http://www.chip-art.net> 下载。

PrintMonitor 默认监测串行口 1,将串行口接收到的数据按字符格式显示到上边的编辑框中。下边的编辑框用于输入往串行口发送的字符串,发送按钮按下后将编辑框内字符串发送到串行口并在发送的字符串前添加标记“>:”后按时间顺序插入到上面显示框中。

图 5-2 PrintMonitor 运行界面



我并没有打包 PrintMonitor,所以即使有它的可执行文件,如果系统中没有安装 VisualBasic6.0 它就不会正常工作。由于 PrintMonitor 过于简单,所以建议读者下载它的源代码并读懂它,这样就可以更灵活的调试程序了。

## 5.3 最小化的格式化打印函数

avr-libc 提供的流函数占用的 FLASH 空间较大,尽管最小连接都要占用 1k 多字节的程序存储器,所以 AT90S2313 等内部只有 2K 存储器的器件无法使用,为此我们可能需要写自己的格式化打印函数,通过简化功能的方法缩小它的代码长度。以下是个实例:

main.c

```
/*
    格式化打印输出最小化实现测试程序
    文件名 : main.c
    硬 件 : CA-M8 实验板
    时 钟 : 外部 4MHz

    2004-11-22
*/

#include <avr/io.h>
#include <stdio.h>      // 使用 va_list 类型定义和相关宏定义
#include <stdlib.h>     // itoa 函数声明头文件
#include <avr/delay.h>

//延时一毫秒
void DelayMs(unsigned int t)
{
    unsigned int i;
    for(i=0;i<t;i++)
        _delay_loop_2(250 * 4);
}

//串口写一字节
int usart_putchar(char c)
{
    if(c=='\n')
        usart_putchar('\r');
    loop_until_bit_is_set(UCSRA,UDRE);
    UDR=c;
    return 0;
}

//简化的格式化输出函数
void miniprintf(char *fmt, ...)
{
    va_list ap; //参数变量定义

    char strval[6];
    char *p;
    int nval;
    unsigned char i;
```

```
va_start(ap, fmt); //设置第一个参数为 fmt
for(p = fmt; *p; p++)
{
    if (*p != '%')
    {
        usart_putchar(*p);
        continue;
    }

    p++;
    switch (*p)
    {
        case 'd':
            nval = va_arg(ap, int); //按类型获取下一个参数
            itoa(nval, strval, 10);
            break;
        case 'x':
            nval = va_arg(ap, int); //按类型获取下一个参数
            itoa(nval, strval, 16);
            break;
    }
    i=0;
    for(i=0;strval[i];i++)
        usart_putchar(strval[i]);
}
va_end(ap); //结束参数变量的使用
}

//测试主例程
int main(void)
{
    unsigned int i=-5;

    UCSRB=_BV(RXEN)|_BV(TXEN); /*(1<<RXCIEN)|(1<<TXCIEN)*/
    UBRRL=25; //9600 baud 6MHz:38 4MHz:25

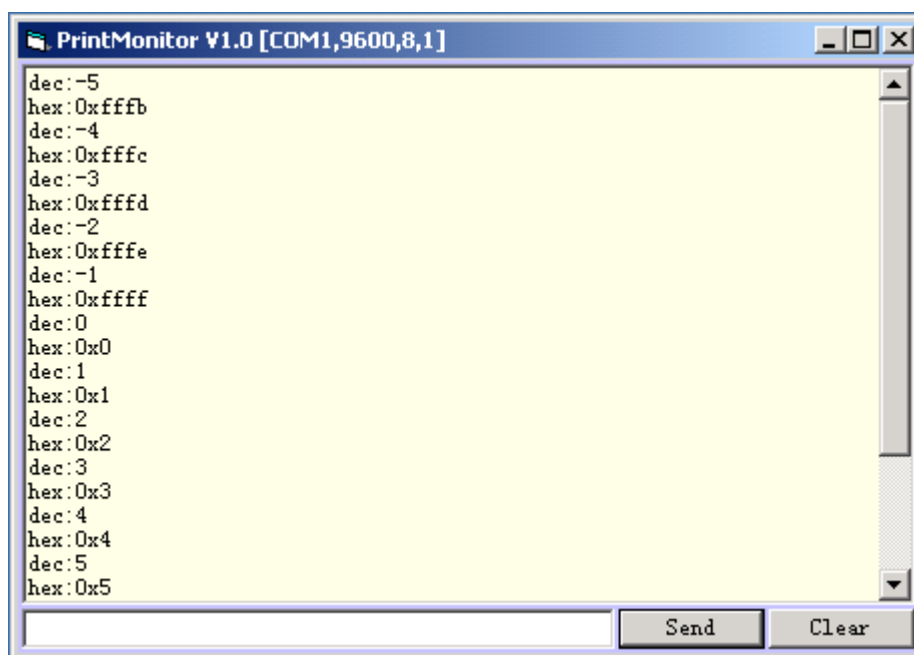
    while(1)
    {
        miniprintf("dec:%d\n", i);
        miniprintf("hex:0x%x\n", i);
        i++;
        DelayMs(1000);
    }
}
```

```
}
```

miniprintf 函数仅实现 16 位整数的十进制和十六进制转换功能，从而缩小了代码长度。va\_list 是 C 语言参数列表类型，宏 va\_start 用于指定第一个参数，va\_arg 负责从参数列表变量提取下一个参数（按指定类型返回），使用完参数列表变量后要在函数返回前要调用 va\_end 表示释放列表。

以上程序在 CA-M8 上运行后用 PrintMonitor 监测结果如下图所示：

图 5-3 PrintMonitor 测试结果



我用 WinAVR-20040720 编译后 FLASH 程序量是 598 字节。如果还嫌大，也可以进一步简化，那就是自己写 itoa 函数了，可以将转换十六进制格式功能去掉。

## 第六章 CA-M8 上实现 AT89S52 下载编程器

ATMEL 推出的 89S 系列单片机具有类似 AVR 的 ISP 编程功能，单片机 ISP 接口为用户提供了一种对 FLASH 存储器串行编程方法。ISP 功能就象操作串行 EEPROM 存储器那样使单片机的编程变得简单方便。

### 6.1 编程原理

CA-M8 上的单片机 ATmega8 与计算机 RS232 通信，将要编程的数据读入后，使用 I/O 口将数据按规定顺序写入到 AT89S52 的 ISP 模块。实现对 FLASH 的编程操作。如图 6-1 所示。

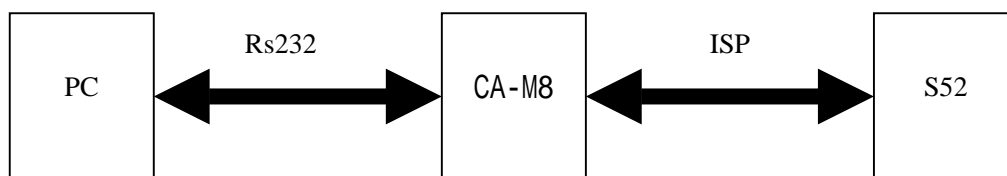
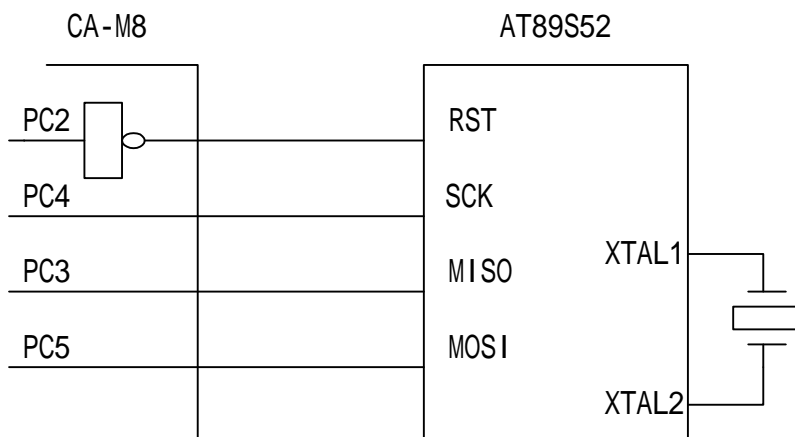


图 6-1 下载编程实现框图：

PC 上的程序利用一个通用的程序：LuckyProg2004，它有一个与很多编程软件那样的操作界面，它的编程操作就是按规定的协议将数据传送到指定的串行口，CA-M8 上的单片机要做的就是接收到数据后写入到 AT89S52，下一节将详细介绍这个软件。

图 6-2 为 CA-M8 与 AT89S52 下载引脚间的连线图，用 ISP 口下载程序时 AT89S52 必需有时钟源，为此编程时在 XTAL0 与 XTAL1 间接一个 3~33MHz 的晶振，ISP 数据通信口 MOSI、MISO 和 SCK 分别接在 ATmega8 的 PC5、PC3 和 PC4 上，而 I/O 口 PC2 通过一个三极管反向后连接到 RST，主要是为了防止目标板上的复位电路使 I/O 口烧坏。

图 6-2 CA-M8 与 AT89S52 下载引脚连线图



## 6.2 LuckyProg 2004 概述

当我们使用 C 语言之后低端单片机的开发几乎没有必要再使用仿真器了,尤其是现在的单片机集成可多次烧写的 Flash 程序存储器和 ISP 板上下载程序功能之后我彻底适应了这种没有仿真器的开发模式。2003 年 ATMEL 推出了支持 ISP 下载功能的 51 单片机 AT89S51 和 AT89S52,当时 ATmega8 价格相对 S51/S52 还比较贵,所以我决定适当的场合使用 S51/S52,但当时没有一个好的下载线能够购买或自制,双龙的下载软件 SL-ISP 号称配合它的并口下载线可支持 S51/S52,我试用了一下发现慢得有些异常。显然,双龙对这套软件的投入是非常让人失望的,我猜测双龙的 VC 工程师使用了简单的 Sleep 调用实现了延时操作导致编程过程的过分延时,这与 PonyProg 的 CPU 频率校准技术是不能比较的。最后我决定自制一个 S52 的下载工具,制作是出于我个人爱好,是完全利用业余时间完成的。我用 VB6 写了一个上位机程序(起名为 LuckyProg S52)与单片机 AT90S2313 间串行通信,再用 2313 的 I/O 口操作 S52 的 ISP 引脚,实现对 S52 的编程。做出来后用了一段时间,使用结果非常好,我的 USB 学习板(PDIUSB12+AT89S52)就是使用它调试的,从 <http://www.chip-art.net> 可以下载关于它的所有资料。

但是在 2004 年下半年双龙又推荐一种新款芯片 ATmega48/88/168 时我才发现,这样跟着 ATMEL 不休止的开发上位机程序是没有意义的。最终我有了一个构思,就是为这种模式的编程提供标准的上位机软件,于是之后的一个月 LuckyProg2004 出现了。它不考虑给什么器件编程,只将程序文件(通常是 intel hex 格式)按固定的协议发送到串行口,这样、要实现一个新器件的编程器时只需调整下位机程序,避免编制烦琐的 WINDOWS 应用程序。

有了 LuckyProg2004 和类似 CA-M8 这样的硬件后,对新推出器件的编程不用再依赖少数数的上位机程序开发商或下载线制作商了。花上半天的时间,根据器件的数据手册写一个单片机程序便可拥有支持该器件的编程器,如果你愿意共享你的程序更多的人会非常容易的拥有这些。

下面详细描述如何使用 LuckyProg2004。

### 一. 界面

LuckyProg2004 界面如图 6-3 所示,不能否认模仿 PonyProg2000 的事实☺

### 二. 器件配置

当你使用 LuckyProg2004 时必须告诉它器件的 FLASH 容量、EEPROM 容量等一些信息,这些信息存诸到一个扩展名为“.lpc”的文件中,每一个器件必须有对应的 LPC 文件,

利用“工具”菜单的“器件配置器...”命令可以很容易的生成该配置文件。如图 6-4 所示,“器件属性”区输入器件的名称、FLASH 大小(按 K 字节)和 EEPROM 大小(按字节)。“通信协议配置”区中波特率为编程该器件时 LuckyProg2004 所使用的波特率。器件 ID 是一个字节数据,必须按十六进制给出,每次对器件的操作 LuckyProg2004 会先发该字节表示对该器件编程。另外,上下位机互相传送大批量的数据(如程序数据、或 EEPROM 数据)时上位机程序规定可连续发送/接收的字节数(数据包长度),每次发送/接收完成一个数据包后接收方需回应一个应答字节。数据包长度根据下位机数据缓冲区大小而被选定。

LuckyProg2004 使用四个字节发送器件的锁定位和熔丝位,在器件配置文件编辑器中可以为四个字节中的每一个位设置名称,在写配置位时每个位对应一个复选按钮,选中的相应位被设置成 1。下位机程序应根据接收的四字节写配置位。

配置好你的器件后将它存储到硬盘,再使用“文件”菜单的“添加器件”命令将它添加到 LuckyProg2004 的器件列表中,就可以使用了。当然,这时你必须要有下位机相应的程序支持真正的 ISP 编程操作,LuckyProg2004 提供的仅仅是从计算机向下位机传送编程所需数据的功能。



图 6-3 LuckyProg2004 运行界面

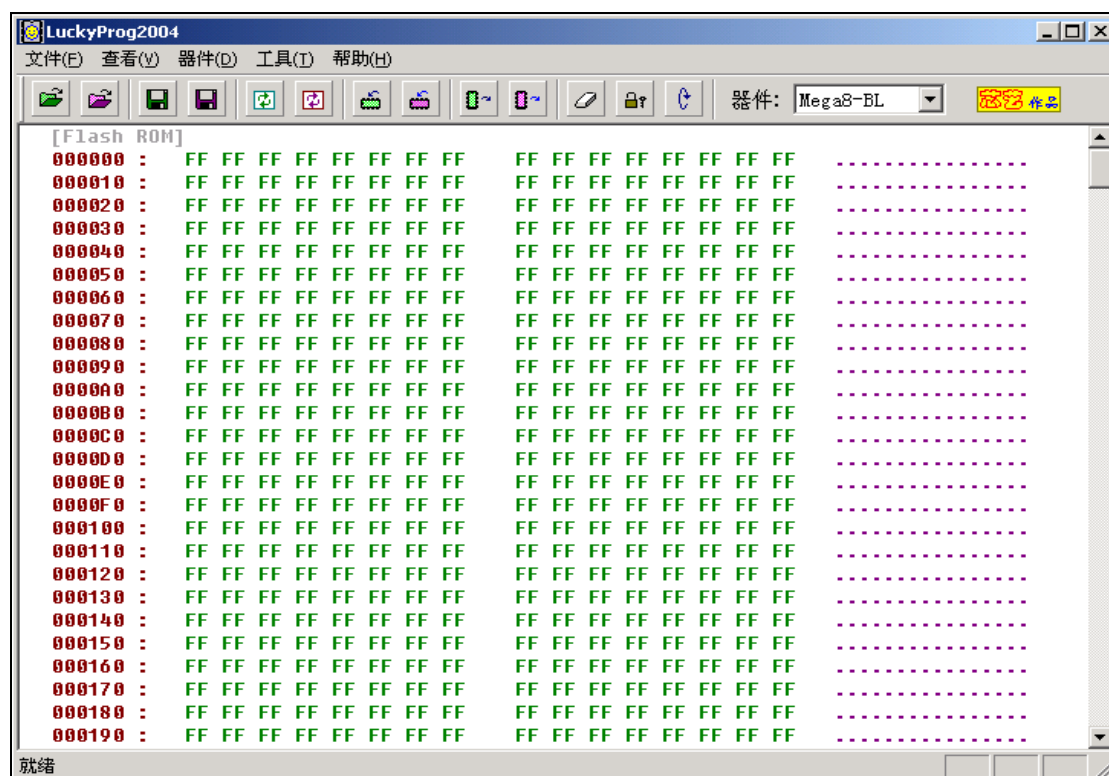


图 6-4 器件配置文件编辑器



## 三 . 编程协议

LuckyProg2004 共支持 10 个与读写串口相关的功能，如表 6-1 所示：

表 6-1 LukcyProg2004 支持命令

命 令	功 能 说 明
LP_WRITE_FLASH	写 FLASH 存储器
LP_READ_FLASH	读 FLASH 存储器
LP_WRITE_EEPROM	写 EEPROM 存储器
LP_READ_EEPROM	读 EEPROM 存储器
LP_ERASE_DEVICE	擦除器件
LP_WRITE_BITS	写配置位
LP_READ_BITS	读配置位
LP_SCAN_HARDWAR	检测硬件
LP_RESET_DEVICE	复位器件
LP_READ_FIRMWARE_MSG	读取固件信息

另外还有两个命令 LP\_VERIFY\_FLASH 和 LP\_VERIFY\_EEPROM 是分别基于 LP\_READ\_FLASH 和 LP\_READ\_EEPROM 的。

1. LP\_WRITE\_FLASH：写 FLASH 存储器命令的流程如图 6-5。LuckyProg2004 将所需写入的数据分成数据包后发送。
2. LP\_READ\_FLASH：读 FLASH 存储器命令的流程如图 6-6。LuckyProg2004 将读取整个 FLASH 空间到缓冲区。
3. LP\_WRITE\_EEPROM:流程如图 6-7,写 EEPROM 存储器命令与写 FLASH 存储器命令类似，将有效数据分成数据包后发送给下位机。
4. LP\_READ\_EEPROM:此命令流程如图 6-8，与读 FLASH 类似，读取整个 EEPROM 到缓冲区。
5. LP\_ERASE\_DEVICE:此命令流程如图 6-9，下位机擦除完器件后应答一字节便可。
6. LP\_WRITE\_BITS:此命令流程如图 6-10 所示，LuckyProg2004 将编辑好的四字节配置位数据发送到下位机。
7. LP\_READ\_BITS:此命令流程如图 6-11 所示，下位机要返回给 LuckyProg2004 四个字节的配置位数据。
8. LP\_SCAN\_HARDWAR: 此命令流程如图 6-12，下位机只回应一字节即可。
9. LP\_RESET\_DEVICE: 此命令流程如图 6-13，下位机复位完器件后回应一字节即可。
10. LP\_READ\_FIRMWARE\_MSG: 此命令流程如图 6-14 所示。此命令从下位机读取 2 4 个字节数据，前 8 个字节认为是作者名（它是不需以 ‘\0’ 结束的字符串），后 1 6 个字节是关于固件的其它信息（也是个需以 ‘\0’ 结尾的字符串）。LuckyProg2004 的菜单“帮助”->“关于当前固件 ...”将执行此命令。

图 6-5 LP\_WRITE\_FLASH 命令流程

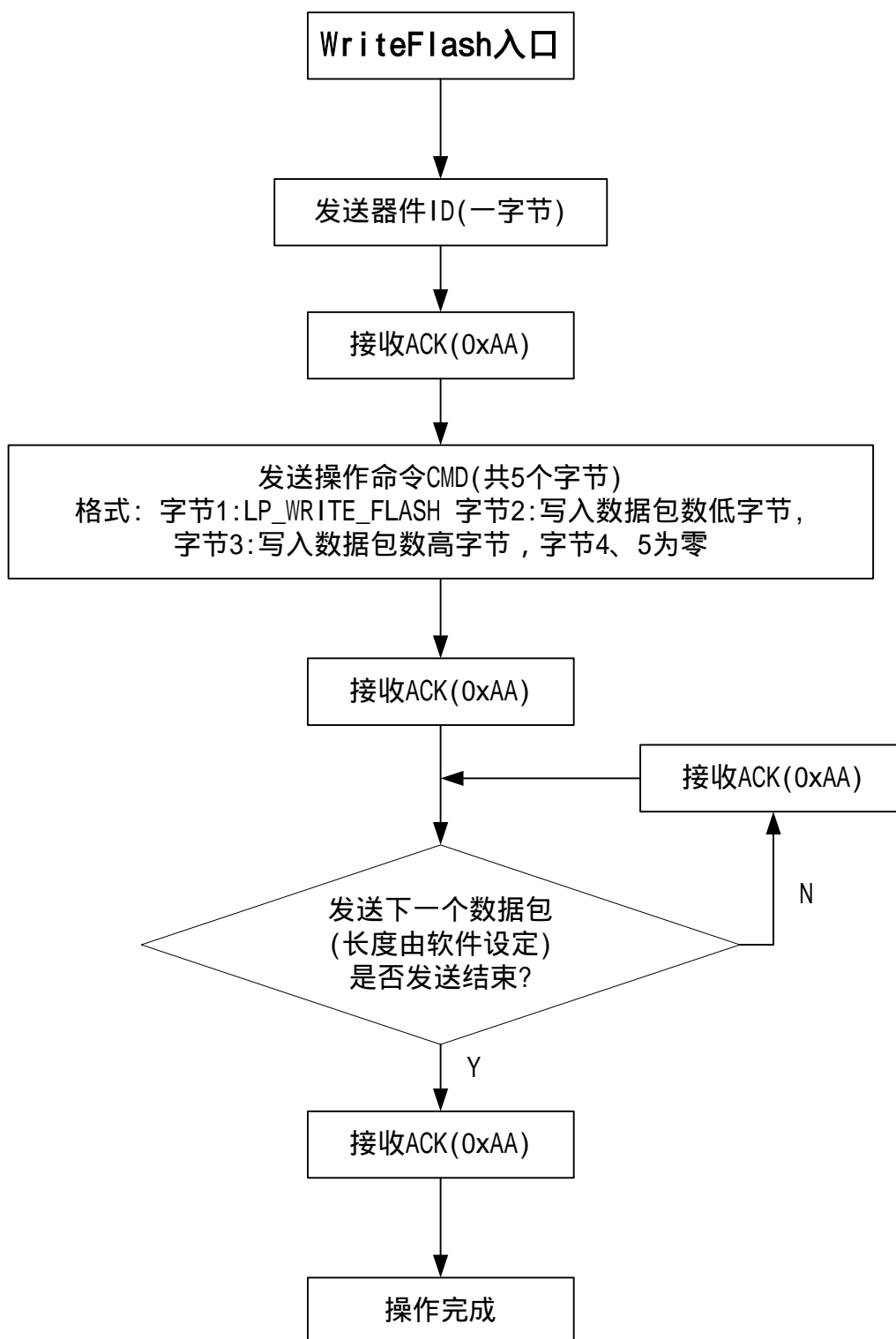


图 6-6 LP\_READ\_FLASH 命令流程

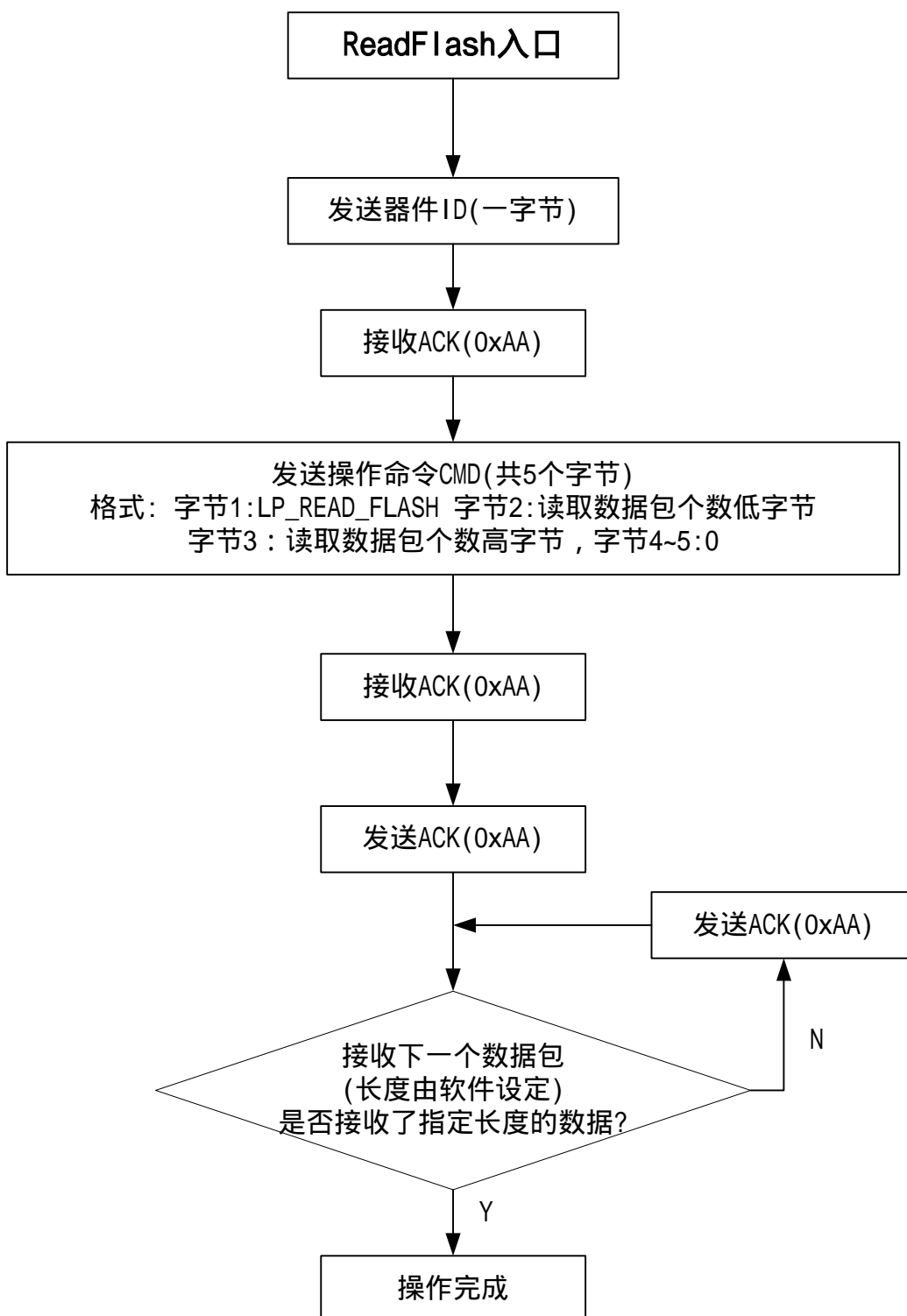


图 6-7 LP\_WRITE\_EEPROM 命令流程

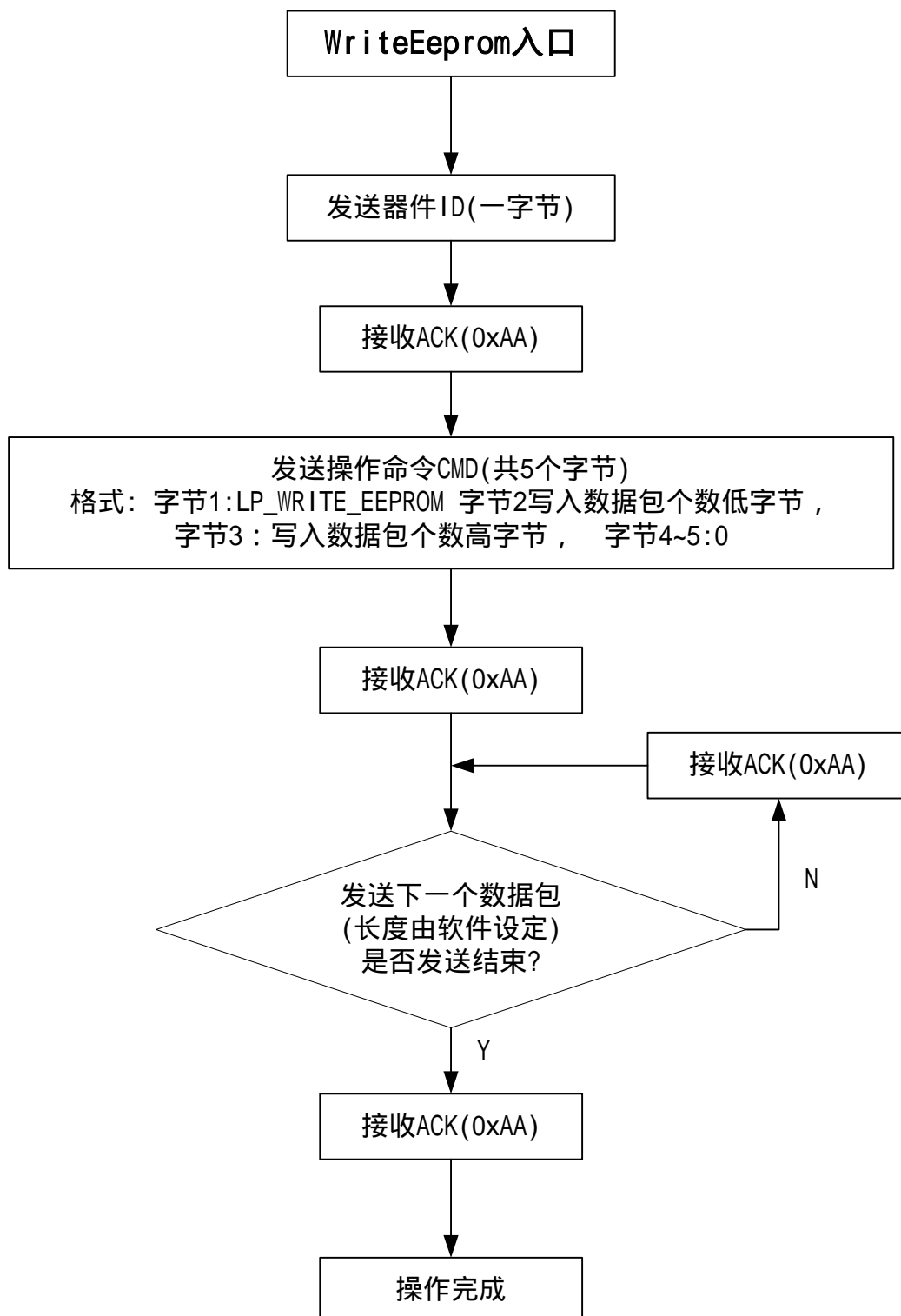


图 6-8 LP\_READ\_EEPROM 命令流程

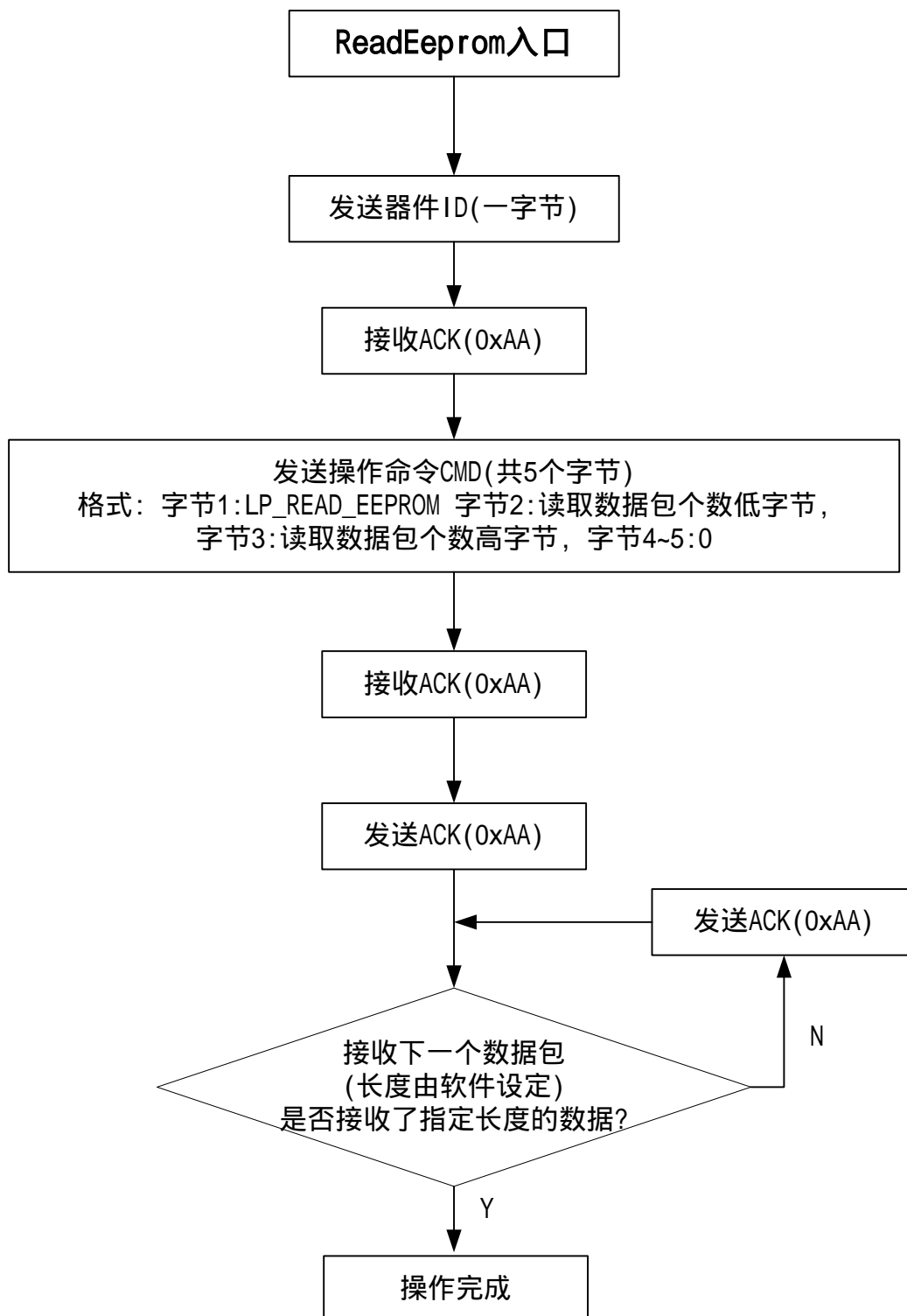


图 6-9 LP\_ERASE\_DEVICE 命令流程

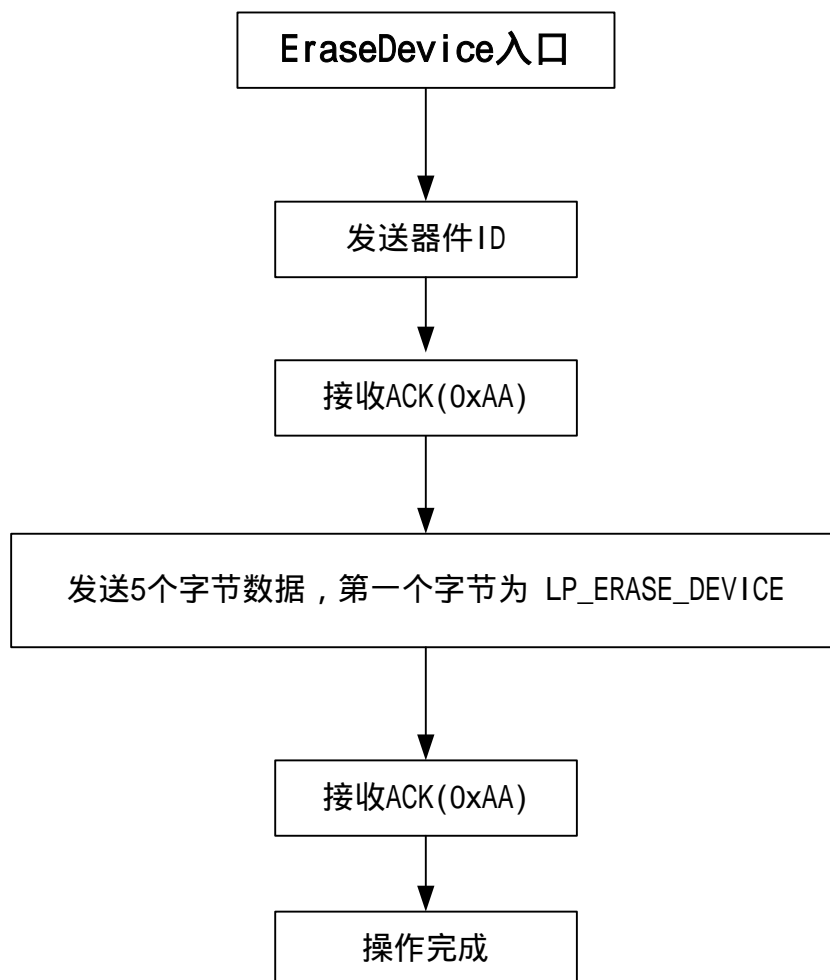


图 6-10 LP\_WRITE\_BITS 命令流程

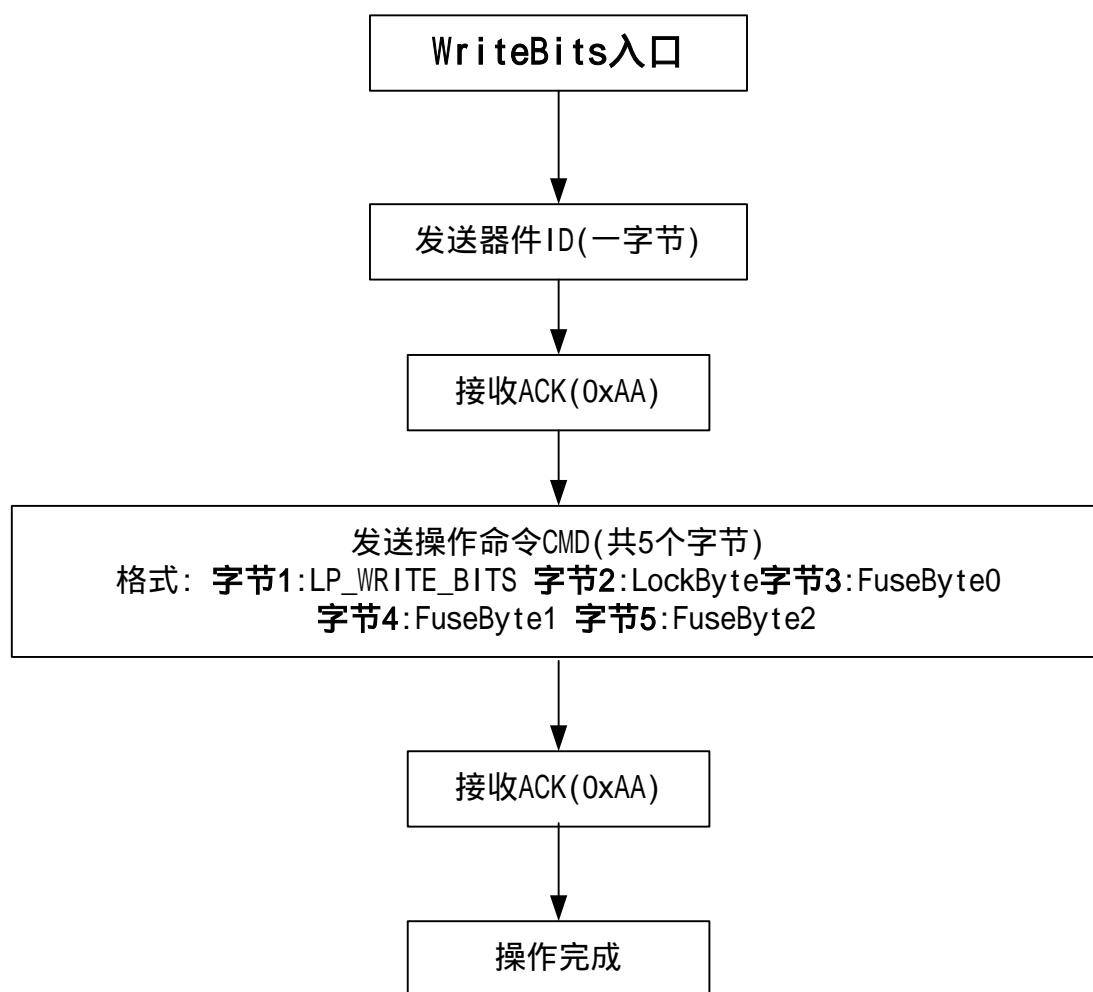




图 6-11 LP\_READ\_BITS 命令流程

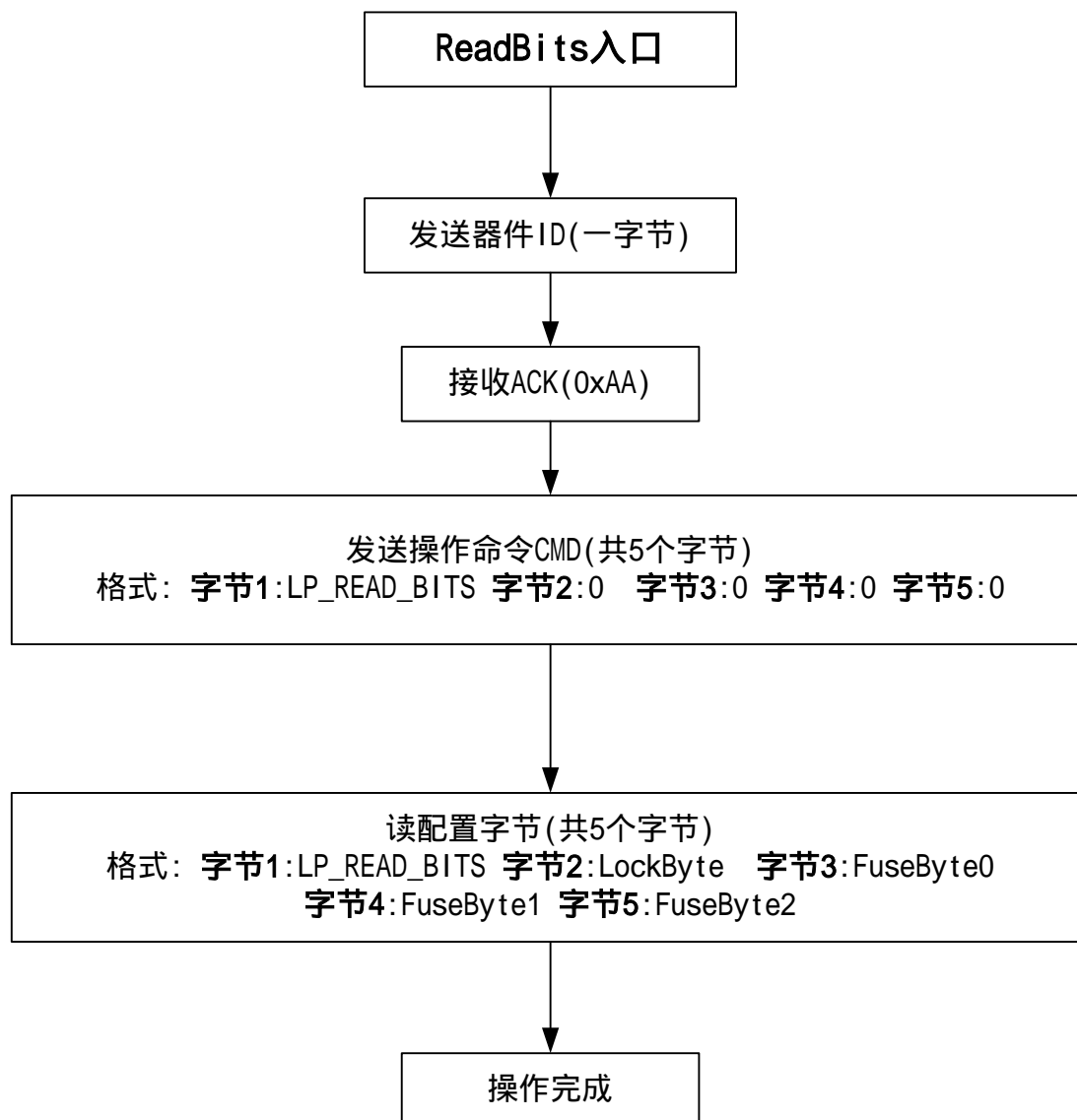


图 6-12 LP\_SCAN\_HARDWARE 命令流程

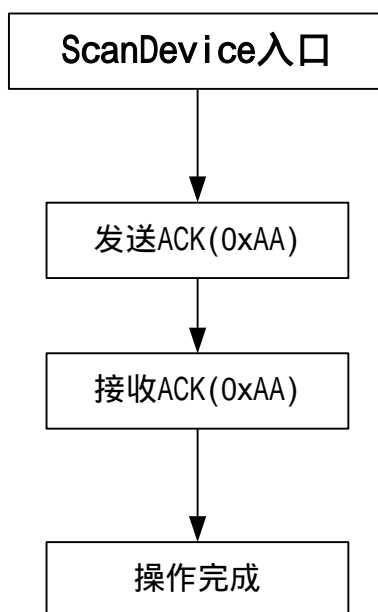


图 6-13 LP\_RESET\_DEVICE 命令流程

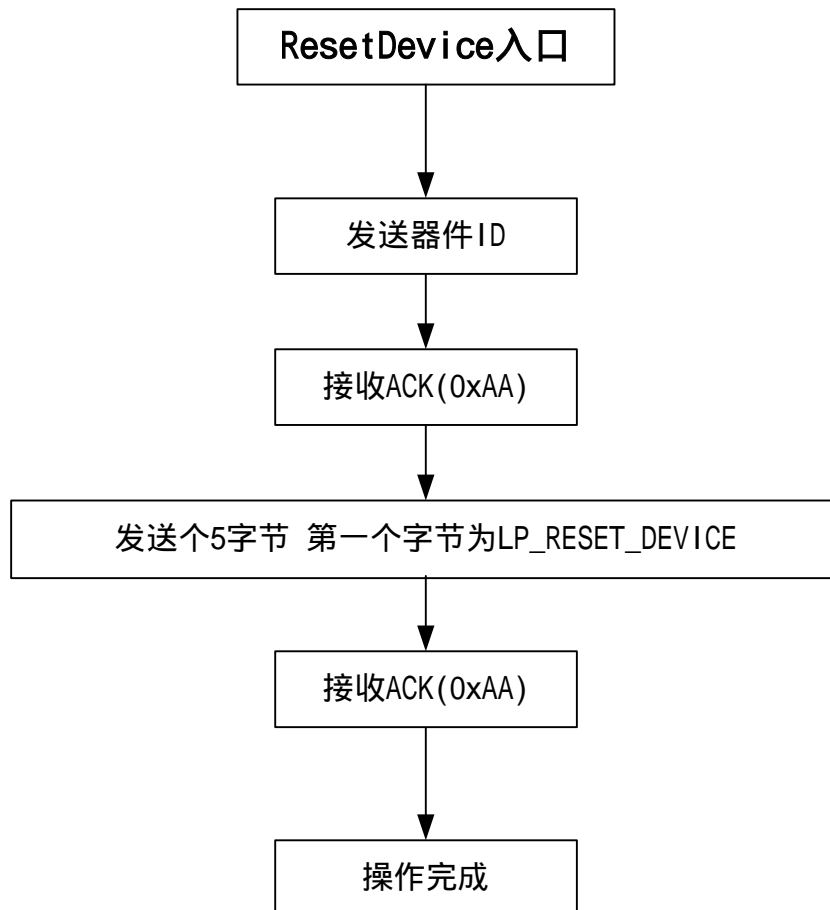
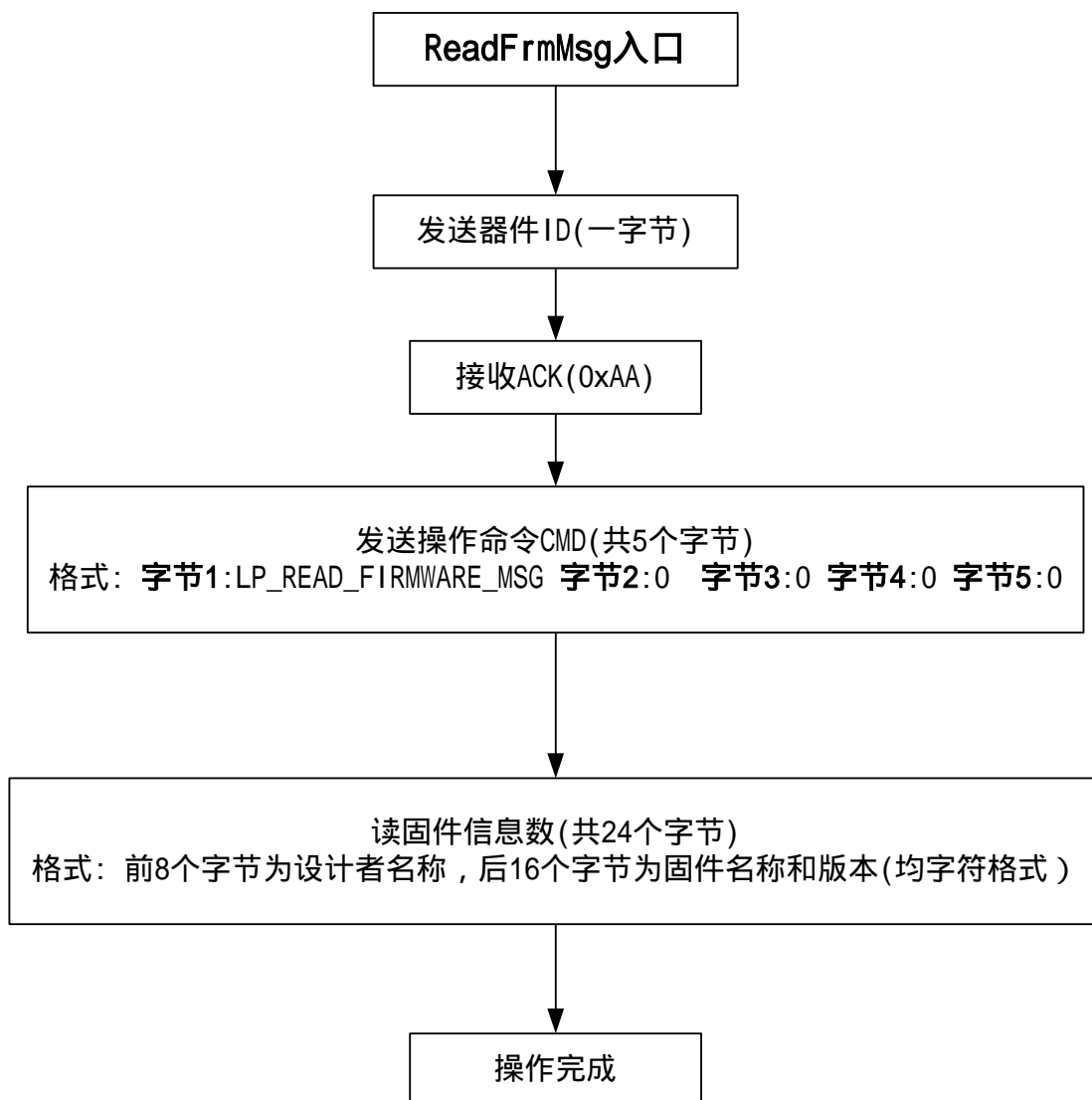


图 6-14 LP\_READ\_FIRMWARE\_MSG 命令流程



### 四 . 编 程 框 架

以下是与 LuckyProg2004 对应下位机程序的框架，它假设硬件为 CA-M8 实验板，WINAVR 编译。

文件：

lp2004.h	:	LuckyProg2004 常量定义头文件
usart.h	:	uart 通信功能函数定义头文件
usart.c	:	uart 通信功能函数实现源文件
main.c	:	主程序模块实现源文件

清单：

LP2004.H

```
#ifndef LP2004_H
#define LP2004_H

#define LP_ACK                0xAA        //应答字节
#define LP_PACKET_SIZE       64          //数据包长度

#define LP_WRITE_FLASH        0x11
#define LP_READ_FLASH         0x12
#define LP_WRITE_EEPROM       0x13
#define LP_READ_EEPROM        0x14
#define LP_ERASE_DEVICE       0x15
#define LP_WRITE_BITS         0x16
#define LP_READ_BITS          0x17
#define LP_SCAN_HARDWAR       0x18
#define LP_RESET_DEVICE       0x19
#define LP_READ_FIRMWARE_MSG  0x1C

#endif
```

usart.h

```
#ifndef MY_USART_H
#define MY_USART_H

#define uchar unsigned char
#define uint unsigned int

#define UART_WAIT            1
```

```
#define UART_NO_WAIT    0

void UartSend(uchar len);

void UartRecv(uchar len,uchar waitflag);

void UartInit(void);

uchar UartIsRecvComplate(void);

uchar *UartGetBuffer(void);

#endif
```

usart.c

```
/*
    文件名：usart.c
    功 能：uart 通信接口
    C P U :ATMega8
    编 译: WinAVR-20040404
    速 度：外部 4MHz
    更改记录：2004-06-22
*/

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

#include "usart.h"
#include "lp2004.h"

static uchar g_bTxdPos,g_bRxdPos,g_bTxdLen,g_bRxdLen;
//UART 缓冲区,是数据包发送/接收缓冲区
static uchar g_aUartBuf[LP_PACKET_SIZE];

void DelayMs(uint t);

//接收完成一字节中断
SIGNAL(SIG_UART_RECV)
{
    uchar c=UDR;
    if(g_bRxdLen>0)
```

```
    {
        g_aUartBuf[g_bRxdPos++]=c;
        g_bRxdLen--;
    }
}

//发送完成一字节中断
SIGNAL (SIG_UART_TRANS)
{
    if(--g_bTxdLen>0)
        UDR=g_aUartBuf[++g_bTxdPos];
    else
        g_bTxdPos=0;
}

//从 UART 缓冲区发送指定长度的数据
void UartSend(uchar len)
{
    g_bTxdPos=0;
    g_bTxdLen=len;
    UDR=g_aUartBuf[0];
    while(g_bTxdLen>0)
        DelayMs(1);
}

//接收指定长度数据到 UART 缓冲区
void UartRecv(uchar len,uchar waitflag)
{
    g_bRxdPos=0;
    g_bRxdLen=len;

    if(waitflag)
    {
        while(g_bRxdLen>0)
            DelayMs(1);
    }
}

//检测异步接收是否完成
uchar UartIsRecvComplate(void)
{
    return !g_bRxdLen;
}
```

```
//获取 UART 数据缓冲区
uchar *UartGetBuffer(void)
{
    return g_aUartBuf;
}

//初始化通用异步串行通信模块
void UartInit(void)
{
    uchar i;

    for(i=0;i<LP_PACKET_SIZE;i++)
        g_aUartBuf[i]=0;
    g_bTxdPos=0;
    g_bRxdPos=0;
    g_bTxdLen=0;
    g_bRxdLen=0;

    UCSRB=_BV(RXCIE)|_BV(TXCIE)|_BV(RXEN)|_BV(TXEN);
    UBRRH=0;
    UBRRL=25;//9600 baud 6MHz:38 4MHz:25
}

```

main.c

```
/*
    文件名：main.c
    功 能：LuckyProg2004 下位机程序编程框架
    硬件  ：CA-M8(ChipArt-Mega8)
    编 译：WinAVR-20040404
    时 钟：外部 4MHz

    更改记录：
    2004-11-20
*/
#include <avr/io.h>
#include <avr/delay.h>
#include <avr/signal.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>

#include "usart.h"
#include "lp2004.h"

```

```
#define FREQ 4          //系统时钟(MHz)
#define DEVICE_ID 0X33 //器件 ID

#define SET_RED_LED PORTB&=~_BV(PB1)
#define CLR_RED_LED PORTB|=_BV(PB1)

#define SET_YEL_LED PORTB&=~_BV(PB0)
#define CLR_YEL_LED PORTB|=_BV(PB0)

uchar g_aCommand[5]; //接收命令缓冲区
uchar g_aConfigBits[4]; //配置位缓冲，可以不用

//固件信息数据（至少 24 字节）
const char g_firmmsg[] PROGMEM = "芯艺    QQ:27796915    ";

//延时 t ms
void DelayMs(uint t)
{
    uint i;
    for(i=0;i<t;i++)
        _delay_loop_2(FREQ * 250);
}

//发送应答命令
void ack()
{
    uchar *p=UartGetBuffer();
    p[0]=LP_ACK;
    UartSend(1);
}

//超时检测功能的 UART 接收例程
uchar UartRecvEx(uint len,uint timeout)
{
    uint count=0;
    UartRecv(len,UART_NO_WAIT);
    while(1)
    {
        count++;
        DelayMs(1);
        if(UartIsRecvComplate())
            return 1;
        if(count>=timeout)
            return 0;
    }
}
```



```
    }
}
//处理 LP_WRITE_FLASH 命令例程
void WriteFlash(void)
{
    uint i;
    uint PacketCount=g_aCommand[1]+(g_aCommand[2]<<8);

    for(i=0;i<PacketCount;i++)
    {
        ack();
        if(!UartRecvEx(LP_PACKET_SIZE,2000))
            break;
        //请将以下语句替换为写 FLASH 的代码
        DelayMs(100);
    }
    ack();
}
//处理 LP_READ_FLASH 命令例程
void ReadFlash(void)
{
    uint i;
    uchar j,*p;
    uint PacketCount=g_aCommand[1]+(g_aCommand[2]<<8);
    p=UartGetBuffer();

    ack();

    for(i=0;i<PacketCount;i++)
    {
        if(!UartRecvEx(1,2000))
            break;
        //请将以下语句替换为读 FLASH 的代码
        for(j=0;j<LP_PACKET_SIZE;j++)
            p[j]=0x88;

        UartSend(LP_PACKET_SIZE);
    }
}
//处理 LP_WRITE_EEPROM 命令例程
void WriteEeprom(void)
{
    uint i;
    uint PacketCount=g_aCommand[1]+(g_aCommand[2]<<8);
```

```
for(i=0;i<PacketCount;i++)
{
    ack();
    if(!UartRecvEx(LP_PACKET_SIZE,2000))
        break;
    //请将以下语句替换为写 EEPROM 的代码
    DelayMs(100);
}
ack();
}
//处理 LP_READ_EEPROM 命令例程
void ReadEeprom(void)
{
    uint i;
    uchar j,*p;
    uint PacketCount=g_aCommand[1]+(g_aCommand[2]<<8);
    p=UartGetBuffer();

    ack();

    for(i=0;i<PacketCount;i++)
    {
        if(!UartRecvEx(1,2000))
            break;
        //请将以下语句替换为读 EEPROM 的代码
        for(j=0;j<LP_PACKET_SIZE;j++)
            p[j]=0x99;

        UartSend(LP_PACKET_SIZE);
    }
}
//处理 LP_WRITE_BITS 命令例程
void WriteBits(void)
{
    g_aConfigBits[0]=g_aCommand[1];
    g_aConfigBits[1]=g_aCommand[2];
    g_aConfigBits[2]=g_aCommand[3];
    g_aConfigBits[3]=g_aCommand[4];
    //请在此输入写配置位代码
    ack();
}
//处理 LP_READ_BITS 命令例程
void ReadBits(void)
```

```
{
    uchar *p;
    p=UartGetBuffer();

    //请在此输入读配置位代码
    p[0]=LP_READ_BITS;
    p[1]=g_aConfigBits[0];
    p[2]=g_aConfigBits[1];
    p[3]=g_aConfigBits[2];
    p[4]=g_aConfigBits[3];
    UartSend(5);
}
//处理 LP_ERASE_DEVICE 命令例程
void EraseDevice(void)
{
    //请在此输入擦除器件的代码
    ack();
}
//处理 LP_RESET_DEVICE 命令例程
void ResetDevice(void)
{
    //请在此输入复位器件的代码
    ack();
}
//处理 LP_READ_FIRMWARE_MSG 命令例程
void ReadFirmwareMsg(void)
{
    uchar i;
    uchar *p=UartGetBuffer();
    for(i=0;i<24;i++)
        p[i]=pgm_read_byte(g_firmmsg+i);
    UartSend(24);
}
//入口函数
int main(void)
{
    uchar i;
    uchar *p=UartGetBuffer();

    DDRB=_BV(PB0)|_BV(PB1);
    PORTB=0X03;

    UartInit();
    sei();
}
```

```
while(1)
{
    CLR_RED_LED;
    SET_YEL_LED;
    //接收器件 ID 或回复器件检测命令
    UartRecv(1,UART_WAIT);
    CLR_YEL_LED;
    SET_RED_LED;
    if(p[0]==DEVICE_ID)
    {
        ack();
    }
    else if(p[0]==LP_ACK)    //LP_SCAN_HARDWAR
    {
        ack();
        continue;
    }
    else
    {
        continue;
    }

    //接收命令
    if(!UartRecvEx(5,2000))
        continue;
    for(i=0;i<5;i++)
        g_aCommand[i]=p[i];

    switch(g_aCommand[0])
    {
    case LP_WRITE_FLASH:
        WriteFlash();
        break;
    case LP_READ_FLASH:
        ReadFlash();
        break;
    case LP_WRITE_EEPROM:
        WriteEeprom();
        break;
    case LP_READ_EEPROM :
        ReadEeprom();
        break;
    case LP_ERASE_DEVICE:
```

```
        EraseDevice();
        break;
    case LP_WRITE_BITS :
        WriteBits();
        break;
    case LP_READ_BITS:
        ReadBits();
        break;
    case LP_RESET_DEVICE:
        ResetDevice();
        break;
    case LP_READ_FIRMWARE_MSG:
        ReadFirmwareMsg();
        break;
    default:
        break;
} //switch
} //while(1)
} //main
```

以上是个简单的框架程序，实际上通过判断器件 ID 可以在一个程序里支持多种器件的编程操作。

Usart.c 文件里包含了 UART 通信函数，利用中断实现了数据接收的异步操作。全局变量 `g_aUartBuf` 是串行发送/接收数据在 RAM 内的缓冲区，它的大小等于数据包的长度 `LP_PACKET_SIZE`。`LP_PACKET_SIZE` 在 `LP2004.H` 中定义，应当根据需要修改这个值。如果要改成 256 及以上的数据，同时需要改变四个计数变量 `g_bTxdPos`，`g_bRxdPos`，`g_bTxdLen`，`g_bRxdLen` 的数据类型，它们在框架中定义为 `unsigned char` 类型，所以最大仅支持 255 字节的计数。缓冲区大小大于或等于 256 时应当将它们的数据类型改为 `unsigned int`。

## 6.3 AT89S52 ISP 功能简介

### 一．串行数据的输入与输出时序

数据在 SCK 的上升沿输入到 S52，SCK 的下降沿输出。另外必须保证串行时钟 SCK 的周期至少大于 6 个 CPU 时钟（XTAL1 上的）周期。

### 二．串行编程算法

#### 1. 上电过程

在 VCC 和 GND 间加上电源的同时 RST 脚加高电平。之后至少要等待 10ms。

#### 2. 发送串行编程使能命令

如果通信失步则串行编程失败。如果同步则在编程时钟的第四个字节器件响应 0X69，表示编程使能命令成功。不论响应正确与否，必需保证四字节的时钟周期。

#### 3. 写程序

通过写指令可对程序存储器的每一字节进行编程。一个写指令使单片机进入自定时的编程模式，在 5V 编程电压下典型编程时间少于 1ms。

#### 4. 读程序

任意位置的程序数据可通过读指令从引脚 MISO/P1.6 读出，可用来校验写入的数据。

5. 编程操作结束后将 RST 引脚拉低，使器件进入正常工作模式。

### 三．编程指令

表 6-2 AT89S52 ISP 下载命令

指 令	指 令 格 式			
	字节 1	字节 2	字节 3	字节 4
编程使能	1010 1100	0101 0011	xxxx xxxx	0110 1001(输出)
器件擦除	1010 1100	100x xxxx	xxxx xxxx	xxxx xxxx
读(字节模式)	0010 0000	xxxA12-A8	A7-A0	D7-D0
写(字节模式)	0100 0000	xxxA12-A8	A7-A0	D7-D0
写锁定位	1010 1100	1110 00B1B2	xxxx xxxx	xxxx xxxx
读锁定位	0010 0100	XXXX XXXX	xxxx xxxx	xxLB3 LB2 LB1 xx
读厂标	0010 1000	XXXA5-A0	xxxx xxxx	厂标字节
读(页模式)	0011 0000	XXXA12-A8	Byte 0	Byte1-255
写(页模式)	0101 0000	XXXA12-A8	Byte 0	Byte1-255

注：

1. 锁定位与模式对应

模式 1 (B1=0、B2=0)：无锁定保护

模式 2 (B1=0、B2=1)：内部锁定位 1 (LB1) 有效

模式 3 (B1=1、B2=0)：内部锁定位 2 (LB2) 有效

模式 4 (B1=1、B2=1)：内部锁定定位 3 (LB3) 有效

2. 将 Reset 拉高后 SCK 至少保持 64 个时钟周期才可执行编程允许命令，在页读写中命令和地址后数据由 0 到 255 的顺序传送，只有接收完这 256 字节的数据后下一个指令才能就绪。

关于 AT89S52 的 ISP 编程更详细的描述请查阅官方的数据手册。

## 6.4 下位机程序设计

### 一、延时功能函数

通常在单片机 C 程序里的延时模块为一个计数循环，延时时间的长短往往是先估计，后通过实验或仿真等方法来验证。avr-libc 提供了两个延时 API 函数，利用这两个函数我们可以较精确的产生所需的延时函数。

第一个函数声明如下：

```
void _delay_loop_1(unsigned char count);
```

它的延时时间为  $\text{count} \times 3$  个系统时钟周期，计数器 count 为 8 位无符号整数，第二个函数声明格式为：

```
void _delay_loop_2(unsigned int count);
```

它将延时  $\text{count} \times 4$  个系统时钟周期，计数器 count 为 16 位无符号整数。

若要调用这两个函数，需先包头文件 delay.h，实际上这两个函数的实现就在此文件里，我们可以从 WINAVR 安装目录里的 \AVR\INCLUDE\AVR 子目录里找到并查看它的内容，以下为 \_delay\_loop\_2 的源程序：

```
static inline void
_delay_loop_2(unsigned int __count)
{
    asm volatile (
        "1: sbiw %0,1" "\n\t"
        "brne 1b"
        : "=w" (__count)
        : "0" (__count)
    );
}
```

首先要说明的是，由于函数的实现写在了头文件里，所以被多个源文件包含是可能的，为此有必要将它声明成局部函数（static）。函数内容为内嵌汇编方式，有关内嵌汇编看第 9 章，这里我们只需知道它的执行需要  $\text{count} * 4$  个时钟周期。要注意的是，inline 关键字说明了函数是内连函数，内连函数如同汇编程序里的宏，编译结果是在每一个调用的地方插入一

次函数的内容。为此有程序空间要求且调用频率高的应用中再写一个延时函数是有必要的。以下是为编程器主控单片机 ATmega8 写的延时程序，它以毫秒为单位执行延时任务。

```
void DelayMs(unsigned int t)
{
    unsigned int i;
    for(i=0;i<t;i++)
        _delay_loop_2(FEQ * 250 - 1);
}
```

其中 FEQ 为系统振荡频率(以 M 为单位)。

## 二 . 程序清单

对 AT89S52 的编程下位机程序是在 LuckyProg2004 下位机程序编程框架的基础上修改的，在 CA-M8 实验板上测试通过。

没有改动 usart.h、usart.c 这两个文件，lp2004.h 中 LP\_PACKET\_SIZE 值设置成了 128。

以下是添加支持 AT89S52 ISP 功能后的 main.c 代码：

```
/*
    文件名：main.c
    功 能：AT89S52 isp 程序
    硬件  ：CA-M8(ChipArt-Mega8)
    编 译：WinAVR-20040404
    时 钟：外部 4MHz

    更改记录：
    2004-11-20
*/
#include <avr/io.h>
#include <avr/delay.h>
#include <avr/signal.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>

#include "usart.h"
#include "lp2004.h"

#define  FREQ 4           //系统时钟(MHz)
#define  DEVICE_ID 0X69 //器件 ID

#define  RST  PC2
```



```
#define SCK PC4
#define MISO PC3
#define MOSI PC5
#define PROGPORT PORTC
#define PROGDDR DDRC
#define PROGPIN PINC

#define SET_RED_LED PORTB&=~_BV(PB1)
#define CLR_RED_LED PORTB|=_BV(PB1)

#define SET_YEL_LED PORTB&=~_BV(PB0)
#define CLR_YEL_LED PORTB|=_BV(PB0)

#define SET_BEEPPORTB|=_BV(PB2)
#define CLR_BEEP PORTB&=~_BV(PB2)

uchar g_aCommand[5]; //接收命令缓冲区

//固件信息数据（至少 24 字节）
const char g_firmmsg[] PROGMEM = "芯 艺 QQ:27796915 ";

//延时 t ms
void DelayMs(uint t)
{
    uint i;
    for(i=0;i<t;i++)
        _delay_loop_2(FREQ * 250);
}

void DelayBus(void)
{
    _delay_loop_1(4);
}

void Beep(uchar n)
{
    uchar i;
    for(i=0;i<n;i++)
    {
        DelayMs(200);
        SET_BEEP;
        DelayMs(200);
        CLR_BEEP;
    }
}
```

```
}
void ISP_WriteByte(uchar dat)
{
    uchar i;
    for(i=0;i<8;i++)
    {
        if(dat&0x80)
            PROGPORT|=_BV(MOSI);
        else
            PROGPORT&=~_BV(MOSI);
        PROGPORT|=_BV(SCK);
        DelayBus();
        PROGPORT&=~_BV(SCK);
        DelayBus();
        dat<<=1;
    }
}
uchar ISP_ReadByte(void)
{
    uchar ret=0;
    uchar i;
    for(i=0;i<8;i++)
    {
        ret<<=1;
        PROGPORT|=_BV(SCK);
        DelayBus();
        if(PROGPIN&_BV(MISO))
            ret|=1;
        PROGPORT&=~_BV(SCK);
        DelayBus();
    }
    return ret;
}
} //S52 编程允许

uchar ProgEnable(void)
{
    //MOSI、SCK 设为输出
    PROGPORT&=~_BV(SCK);
    PROGPORT&=~_BV(MOSI);
    PROGDDR|=_BV(MOSI);
    PROGDDR|=_BV(SCK);

    PROGPORT&=~_BV(RST); //RST 拉高
    DelayMs(100);
}
```

```
ISP_WriteByte(0xac);
ISP_WriteByte(0x53);
ISP_WriteByte(0);

if(ISP_ReadByte()==0x69)
    return 1;
else
    return 0;
}
//S52 复位
void ProgDisable(void)
{
    //MOSI、SCK 设为输入高阻
    PROGPORTE&=~_BV(SCK);
    PROGPORTE&=~_BV(MOSI);
    PROGDDE&=~_BV(MOSI);
    PROGDDE&=~_BV(SCK);

    PROGPORTE&=~_BV(RST);//RST 拉高
    DelayMs(500);
    PROGPORTE|=_BV(RST);//RST 拉低
}
//发送应答命令
void ack(void)
{
    uchar *p=UartGetBuffer();
    p[0]=LP_ACK;
    UartSend(1);
}

//超时检测功能的 UART 接收例程
uchar UartRecvEx(uint len,uint timeout)
{
    uint count=0;
    UartRecv(len,UART_NO_WAIT);
    while(1)
    {
        count++;
        DelayMs(1);
        if(UartIsRecvComplate())
            return 1;
        if(count>=timeout)
            return 0;
    }
}
```

```
}
//处理 LP_WRITE_FLASH 命令例程
void WriteFlash(void)
{
    uint i;
    uchar j;
    uint PacketCount=g_aCommand[1]+(g_aCommand[2]<<8);
    uchar *p=UartGetBuffer();

    if(!ProgEnable())
    {
        Beep(2);
        return ;
    }

    for(i=0;i<PacketCount;i+=2)
    {
        ISP_WriteByte(0x50);
        ISP_WriteByte((uchar)(i/2)); //Write address

        ack();
        if(!UartRecvEx(LP_PACKET_SIZE,2000))
            break;
        for(j=0;j<LP_PACKET_SIZE;j++)
        {
            ISP_WriteByte(p[j]);
            DelayMs(1);
        }

        ack();
        if(!UartRecvEx(LP_PACKET_SIZE,2000))
            break;
        for(j=0;j<LP_PACKET_SIZE;j++)
        {
            ISP_WriteByte(p[j]);
            DelayMs(1);
        }
    }
    ack();

    DelayMs(300);
    ProgDisable();
    Beep(1);
}
```

```
//处理 LP_READ_FLASH 命令例程
void ReadFlash(void)
{
    uint i;
    uchar j,*p;
    uint PacketCount=g_aCommand[1]+(g_aCommand[2]<<8);
    p=UartGetBuffer();

    if(!ProgEnable())
    {
        Beep(2);
        return ;
    }

    ack();

    for(i=0;i<PacketCount;i+=2)
    {
        ISP_WriteByte(0x30);
        ISP_WriteByte(i/2);//Write address

        if(!UartRecvEx(1,2000))
            break;
        for(j=0;j<LP_PACKET_SIZE;j++)
            p[j]=ISP_ReadByte();
        UartSend(LP_PACKET_SIZE);

        if(!UartRecvEx(1,2000))
            break;
        for(j=0;j<LP_PACKET_SIZE;j++)
            p[j]=ISP_ReadByte();
        UartSend(LP_PACKET_SIZE);
    }

    ProgDisable();
    Beep(1);
}

//处理 LP_WRITE_EEPROM 命令例程
void WriteEeprom(void)
{
    uint i;
    uint PacketCount=g_aCommand[1]+(g_aCommand[2]<<8);

    for(i=0;i<PacketCount;i++)
```

```
{
    ack();
    if(!UartRecvEx(LP_PACKET_SIZE,2000))
        break;
    //请将以下语句替换为写 EEPROM 的代码
    DelayMs(100);
}
ack();
}
//处理 LP_READ_EEPROM 命令例程
void ReadEeprom(void)
{
    uint i;
    uchar j,*p;
    uint PacketCount=g_aCommand[1]+(g_aCommand[2]<<8);
    p=UartGetBuffer();

    ack();

    for(i=0;i<PacketCount;i++)
    {
        if(!UartRecvEx(1,2000))
            break;
        //请将以下语句替换为读 EEPROM 的代码
        for(j=0;j<LP_PACKET_SIZE;j++)
            p[j]=0xFF;

        UartSend(LP_PACKET_SIZE);
    }
}
//处理 LP_WRITE_BITS 命令例程
void WriteBits(void)
{
    uchar tmp;
    uchar *p=UartGetBuffer();
    if(!ProgEnable())
    {
        Beep(2);
        return ;
    }
    tmp=0xe0;
    if(p[1]&0x01)
        tmp|=0x02;
    if(p[1]&0x02)
```

```
        tmp|=0x01;

    ISP_WriteByte(0xac);
    ISP_WriteByte(tmp);
    ISP_WriteByte(0);
    ISP_WriteByte(0);
    ProgDisable();
    ack();
    Beep(1);
}
//处理 LP_READ_BITS 命令例程
void ReadBits(void)
{
    uchar c,*p;
    p=UartGetBuffer();
    if(!ProgEnable())
    {
        Beep(2);
        return ;
    }
    ISP_WriteByte(0x24);
    ISP_WriteByte(0);
    ISP_WriteByte(0);
    c=ISP_ReadByte();
    ProgDisable();
    c&=0x1c;
    if(c&0x10)
        p[1]=0x03;
    else if(c&0x08)
        p[1]=0x01;
    else if(c&0x04)
        p[1]=0x02;
    else
        p[1]=0;
    p[0]=LP_READ_BITS;
    p[2]=0;
    p[3]=0;
    p[4]=0;
    UartSend(5);
    Beep(1);
}
//处理 LP_ERASE_DEVICE 命令例程
void EraseDevice(void)
{
```

```
if(!ProgEnable())
    return ;

ISP_WriteByte(0xac);
ISP_WriteByte(0x80);
ISP_WriteByte(0x0);
ISP_WriteByte(0x0);
DelayMs(1000);
ProgDisable();

ack();
Beep(1);
}
//处理 LP_RESET_DEVICE 命令例程
void ResetDevice(void)
{
    ProgDisable();
    ack();
    Beep(1);
}
//处理 LP_READ_FIRMWARE_MSG 命令例程
void ReadFirmwareMsg(void)
{
    uchar i;
    uchar *p=UartGetBuffer();
    for(i=0;i<24;i++)
        p[i]=pgm_read_byte(g_firmmsg+i);
    UartSend(24);
    Beep(1);
}
//入口函数
int main(void)
{
    uchar i;
    uchar *p=UartGetBuffer();

    DDRB=_BV(PB0)|_BV(PB1)|_BV(PB2);
    PORTB=0X03;

    PROGPORTRST=_BV(RST);
    PROGDDBRST=_BV(RST);

    UartInit();
    sei();
```



```
while(1)
{
    CLR_RED_LED;
    SET_YEL_LED;
    //接收器件 ID 或回复器件检测命令
    UartRecv(1,UART_WAIT);
    CLR_YEL_LED;
    SET_RED_LED;
    if(p[0]==DEVICE_ID)
    {
        ack();
    }
    else if(p[0]==LP_ACK)//LP_SCAN_HARDWAR
    {
        ack();
        continue;
    }
    else
    {
        continue;
    }

    //接收命令
    if(!UartRecvEx(5,2000))
        continue;

    for(i=0;i<5;i++)
        g_aCommand[i]=p[i];

    switch(g_aCommand[0])
    {
    case LP_WRITE_FLASH:
        WriteFlash();
        break;
    case LP_READ_FLASH:
        ReadFlash();
        break;
    case LP_WRITE_EEPROM:
        WriteEeprom();
        break;
    case LP_READ_EEPROM :
        ReadEeprom();
        break;
```

```
case LP_ERASE_DEVICE:
    EraseDevice();
    break;
case LP_WRITE_BITS :
    WriteBits();
    break;
case LP_READ_BITS:
    ReadBits();
    break;
case LP_RESET_DEVICE:
    ResetDevice();
    break;
case LP_READ_FIRMWARE_MSG:
    ReadFirmwareMsg();
    break;
default:
    break;
} //switch
} //while(1)
} //main
```

DEVICE\_ID 的值是可任意设置的，只要与 LuckyProg2004 中设置的器件 ID 相匹配便可，这里我设置成了 0x69。

接下来的工作就是配置一下 LuckyProg2004 了，非常的简单，打开“器件配置文件编辑器”输入如图 6-15 所示的内容，保存到一个扩展名为“.lpc”的文件之后，将此文件添加到 LuckyProg2004 的器件列表当中。

图 6-15 AT89S52 配置文件生成



到此我们已经实现了一个非常实用的 AT89S52 下载编程器，一个人的精力的确有限，真心希望 LuckyProg2004 和 CA-M8 能够得到广大单片机爱好者的大力支持，将来能实现更多器件的串行编程。

请关注 <http://www.chip-art.net> 或 <http://chipart.17kl.net>，我将在这里列出 LuckyProg2004 和 CA-M8 支持更多器件后的程序。

## 第七章 硬件 TWI 端口编程

### 7.1 TWI 模块概述

ATMega 系列单片机片内集成两线制串行接口模块,Atmel 文档称它为 TWI 接口。事实上 TWI 与 PHILIPS 的 I2C 总线是同一回事,之所以叫它 TWI 是因为这样的命名可使 Atmel 避免交术语版税。所以, TWI 兼容 I2C 更一种说法。

关于 I2C 协议参考 PHILIPS 相关文档。

AVR 硬件实现的 TWI 接口是面向字节和基于中断的,相对软件模拟 I2C 总线有更好的实时性和代码效率,引脚输入部分还配有毛刺抑制单元,可去除高频干扰。另外,结合 AVR I/O 端口功能,在 TWI 使能时可设置 SCL 和 SDA 引脚对应的 I/O 口内部上拉电阻有效,这样可省去 I2C 要求的外部两个上拉电阻。

下面以 MEGA8 为例,简要介绍 TWI 接口的工作方式。

在 I2C 总线上 MEGA8 可扮演主控制器(主控模式)和从器件(被控模式)的角色。

不论主控模式还是被控模式都应当将 TWI 控制寄存器 TWCR 的 TWEN 位置 1 从而使能 TWI 模块。TWEN 位被置位后 I/O 引脚 PC5 和 PC4 被转换成 SCL 和 SDA,该管脚上的斜率限制和毛刺滤波器生效。如果外部没有接上拉电阻可用类似如下的操作使能该管脚上的内部上拉电阻:

```
DDRC&=0Xcf;  
PORTC|=0X30;
```

对 TWI 控制寄存器 TWCR 的操作可在总线上产生 START 和 STOP 信号,从一个 START 到 STOP 被认为是主控模式的行为。

将 TWI 地址寄存器 TWAR 的第一位 TWGCE 置有效,同时将 TWI 控制寄存器 TWCR 的 TWEA(应答允许)位置 1, TWI 模块就可以对总线上对它的寻址做出应答,并置状态字。

对总线的操作或总线上产生事件后应用程序应当根据 TWI 状态寄存器值来确定下一步的操作。关于不同模式下的状态值的详细描述参考 MEGA8 的数据手册。

对 TWI 模块的操作均为寄存器的读写操作 Avr-libc 没有提供专门的 API。文件 twi.h 定义了状态字的常量和一个返回状态字的宏。

## 7.2 主控模式操作实时时钟 DS1307

### 一 实时时钟 DS1307 介绍

DS1307是低功耗、两线制串行读写接口、日历和时钟数据按BCD码存取与时钟/日历芯片。它提供秒、分、小时、星期、日期、月和年等时钟日历数据。另外它还集成了如下几点功能：

- 56 字节掉电时电池保持的 NV SRAM 数据存储
- 可编程的方波信号输出
- 掉电检测和自动切换电池供电模式

DS1307 把 8 个寄存器和 56 字节的 RAM 进行了统一编址，具体地址和寄存器数据组织格式如下表：

表 7-1 DS1307 内存组织结构

地址	数据	格 式							
		7	6	5	4	3	2	1	0
00	SECONDS	CH	秒 10 位			秒个位			
01	MINUTES	0	分 10 位			分个位			
02	HOURS	0	12 24	10HR A/P	小时 10 位	小时个位			
03	DAY	0	0	0	0	0	星期		
04	DATE	0	0	日期 10 位		日期个位			
05	MONTH	0	0	0	月 10 位	月个位			
06	YEAR	年 10 位			年个位				
07	CONTROL	OUT	0	0	SQWE	0	0	RS1	RS0
08 --- 3fh	RAM 56 byte	用户数据存储区							

在读写过程中 DS1307 内部维护一个地址指针，通过写操作可对它赋值，读和写每一字节时自动加一，当指针越过 DS1307 内部 RAM 尾部时指针将返回到 0 地址处。

DS1307 的时钟和日历数据按 BCD 码存储。

方波信号输出功能：

方波信号输出功能从 SQW/OUT 引脚输出设置频率的方波，CONTROL 寄存器用于控制 SQW/OUT 脚的输出。

BIT7 (OUT)：此位表示在方波输出被禁止时 SQW/OUT 脚的逻辑电平，在 SQWE=0 (输出禁止) 时若 OUT 为 1 则 SQW/OUT 脚为高电平，反之亦然。

BIT4 (SQWE) 方波输出允许/禁止控制位，1 有效。

BIT0 (RS0)、BIT1 (RS1) 用于设定输出波形的频率，如下表：

表 7-2 方波信号输出频率设置

RS1	RS0	输出频率 (Hz)
0	0	1
0	1	4096
1	0	8192
1	1	32768

要注意的是,00h 地址的第 7 位为器件时钟允许位(CH),由于在在开始上电时内部 RAM 内容随机,所以在初始化时将 CH 位设零(时钟允许)是非常重要的。

DS1307 在 TWI 总线上是个从器件,地址(SLA)固定为 1101000。

#### DS1307 写操作 – TWI 被控接收模式

主控制器按如下顺序将数据写入到 DS1307 寄存器或内部 RAM 中:

1. START 信号
2. 写 SLA+W(0xd0)字节,DS1307 应答(ACK)
3. 写 1 字节内存地址(在以下第四步写入的第一字节将存入到 DS1307 内该地址处,DS1307 应答)
4. 写数据(可写多个字节,每一字节写入后 DS1307 内部地址计数器加一,DS1307 应答)
5. STOP 信号

#### DS1307 读操作 – TWI 被控发送模式

主控制器按如下顺序将 DS1307 寄存器或内部 RAM 数据读取:

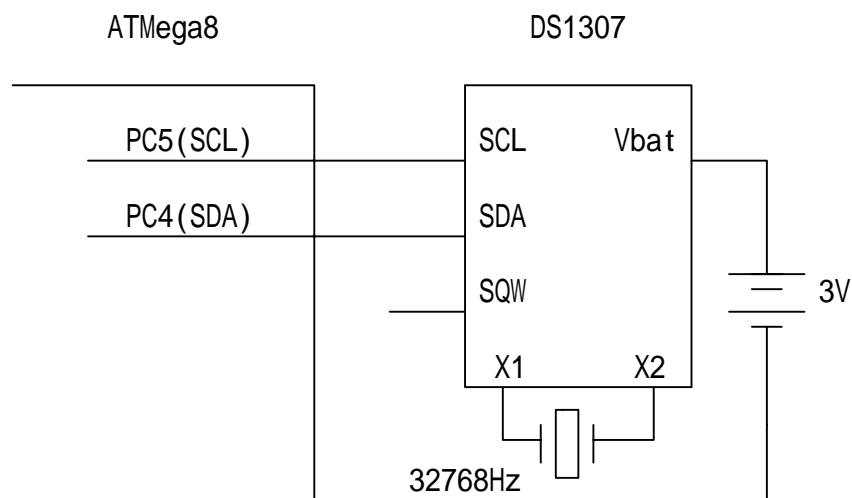
1. START 信号
2. 写 SLA+R(0xd1)字节,DS1307 应答(ACK)
3. 读数据(可读多个字节,读取数据的 DS1307 内部地址由上次写操作或读操作决定,读取每一字节 DS1307 内部地址计数器加一,主器件应答,读取最后一字节时主器件回应一 NACK 信号)
4. STOP 信号

## 二 DS1307 实验电路

DS1307 与 Mega8 间的连接如 6-1 所示,DS1307 的 X1 和 X2 管脚需接 32768Hz 晶振。

Vbat 引脚接的电池电压必需在 2.0V ~ 3.5V 范围内。当 VCC 引脚上的电压降到 1.25 倍电池电压时 DS1307 内部写保护电路生效,RTC 数据和内部 RAM 的读写补禁止。

图 6-1 Mega8 与 DS1307 的连接电路



### 三 程序设计

以下是操作 DS1307 的示例程序，利用基于 USART 的标准 I/O 实现读写日历和时钟。

```
/*  
Mega8 硬件 TWI 接口读写实时时钟 DS1307 程序  
文件名：main.c  
时 钟：外部 4MHz  
  
2004-09-07  
*/  
  
#include <avr/io.h>  
#include <avr/delay.h>  
#include <avr/twi.h>  

```

```
#define RTC_READ 1
#define RTC_WRITE 0

FILE *g_hFile;

uchar g_aTimeBuf[7]; // 日历/时钟 BCD 格式缓冲区
uchar g_aTimeBin[7]; // 时钟/日历二进制格式缓冲区

void DelayMs(uint ms)
{
    uint i;
    for(i=0; i<ms; i++)
        _delay_loop_2(FREQ *250);
}

/*****标准 I/O 功能*****开始*****/
// 标准 I/O 输出函数
int usart_putchar(char c)
{
    if(c=='\n')
        usart_putchar('\r');
    loop_until_bit_is_set(UCSRA, UDRE);
    UDR=c;
    return 0;
}
// 标准 I/O 输入函数
int usart_getchar(void)
{
    loop_until_bit_is_set(UCSRA, RXC);
    return UDR;
}

// 初始化
void IoInit(void)
{
    DDRC=0;
    PORTC|=_BV(PC5)|_BV(PC4); // PC4、PC5 内部上拉电阻开

    // 串行口初始化
    UCSRB=_BV(RXEN)|_BV(TXEN); /* (1<<RXCIE)|(1<<TXCIE) */
    UBRRL=25; // 9600 baud 6MHz:38 4MHz:25
}
```



```
//UART 用于标准 I/O 输入输出
g_hFile=fdevopen(usart_putchar,usart_getchar,0);
}
/*****标准 I/O 功能*****结束*****/

/*****主模式 TWI 操作部分*****开始*****/

//总线上起动停止条件
void twi_stop(void)
{
    TWCR = _BV(TWINT) | _BV(TWSTO) | _BV(TWEN);
}
//总线上起动开始条件
uchar twi_start(void)
{
    TWCR = _BV(TWINT) | _BV(TWSTA) | _BV(TWEN);
    while ((TWCR & _BV(TWINT)) == 0) ;

    return TW_STATUS;
}

//写一字节
uchar twi_writebyte(uchar c)
{
    TWDR = c;
    TWCR = _BV(TWINT) | _BV(TWEN);
    while ((TWCR & _BV(TWINT)) == 0);
    return TW_STATUS;
}

//读一字节 ack: true 时发 ACK , false 时发 NACK
uchar twi_readbyte(uchar *c ,uchar ack)
{
    uchar tmp=_BV(TWINT)|_BV(TWEN);

    if(ack)
        tmp|=_BV(TWEA);
    TWCR=tmp;
    while ((TWCR & _BV(TWINT)) == 0) ;

    *c=TWDR;
```

```
    return TW_STATUS;
}
/*****主模式 TWI 操作部分*****结束*****/

/*****DS1307 操作*****开始*****/
//对 DS1307 内存连续的写操作
uchar rtc_write(uchar addr,uchar *buf,uchar len)
{
    uchar i;

    twi_start();
    twi_writebyte(DS1307_ADDR|TW_WRITE);
    twi_writebyte(addr);//write address
    for(i=0;i<len;i++)
        twi_writebyte(buf[i]);
    twi_stop();
    return 0;
}

//对 DS1307 内存连续的读操作
uchar rtc_read(uchar addr,uchar *buf,uchar len)
{
    uchar i;

    rtc_write(addr,0,0);//set address

    DelayMs(10);

    twi_start();
    twi_writebyte(DS1307_ADDR|TW_READ);
    for(i=0;i<len-1;i++)
        twi_readbyte(buf+i,TW_ACK);
    twi_readbyte(buf+i,TW_NACK);

    twi_stop();
    return 0;
}
/*****DS1307 操作*****结束*****/

/*****接口部分*****开始*****/

//初始化 TWI 功能
```

```
void RtcInit(void)
{
    TWBR=73;
}

//更新或读取 DS1307 日历/时间数据
uchar RtcUpdateData(uchar direction)
{
    uchar ret;
    if(direction) //读
        ret=rtc_read(0,g_aTimeBuf,7);
    else //写
        ret=rtc_write(0,g_aTimeBuf,7);

    return ret;
}

//读 DS1307 用户 RAM
uchar RtcReadRAM(uchar addr,uchar *buf,uchar len)
{
    addr+=8;
    return rtc_read(addr,buf,len);
}

//写 DS1307 用户 RAM
uchar RtcWriteRAM(uchar addr,uchar *buf,uchar len)
{
    addr+=8;
    return rtc_write(addr,buf,len);
}

uchar byte_bintobcd(uchar bin)
{
    uchar ret;
    bin&=0x7f;
    bin%=100;
    ret=bin/10;
    ret <<=4;
    ret|=bin%10;
    return ret;
}

uchar byte_bcdtobin(uchar bcd)
{

```

```
    uchar ret;
    ret=bcd & 0x0f;
    ret+=(bcd>>4)*10;
    return ret;
}

//将二进制格式缓冲区(g_aTimeBin)内容转换成 BCD 格式后保存到
//BCD 格式缓冲区(g_aTimeBuf)
void RtcBinToBCD()
{
    uchar i;
    g_aTimeBin[0]&=0x7f;
    g_aTimeBin[1]&=0x7f;
    g_aTimeBin[2]&=0x3f;
    g_aTimeBin[3]&=0x07;
    g_aTimeBin[4]&=0x3f;
    g_aTimeBin[5]&=0x1f;
    g_aTimeBin[6]&=0xff;

    for(i=0;i<7;i++)
        g_aTimeBuf[i]=byte_bintobcd(g_aTimeBin[i]);
}

//将 BCD 格式缓冲区(g_aTimeBuf)内容转换成二进制格式后保存到
//二进制格式缓冲区(g_aTimeBin)
void RtcBCDToBin()
{
    uchar i;
    for(i=0;i<7;i++)
        g_aTimeBin[i]=byte_bcdtobin(g_aTimeBuf[i]);
}

//写 DS1307 配置字节
void RtcSetSQWOutput(uchar en,uchar level)
{
    //en:方波输出允许 TRUE 有效 level:如果输出禁止 OUT 口的逻辑电平
    uchar c=0;
    if(en) //enable
        c=0x10;
    else //disable
    {
        if(level)
            c=0x80;
    }
}
```

```
    rtc_write(7,&c,1);
}
/*****接口部分*****结束*****/

int main(void)
{
    uchar i;
    char c;
    int tmp[7]; //从标准 I/O 读取缓冲区

    lolnit();
    RtcInnit();

    printf_P(PSTR("输入命令 : g - 打印日历/时钟 , s - 设置日历/时钟 , \
        h - 帮助信息\n"));
    while(1) //main loop
    {
        scanf ("%c",&c);
        if(c=='g')
        {
            RtcUpdateData(RTC_READ);
            RtcBCDToBin();

            printf_P(PSTR("当前日历/时钟 : %d 年%d 月%d 日 星期\
                %d %d:%d:%d\n"),\
                g_aTimeBin[6],g_aTimeBin[5],g_aTimeBin[4],g_aTimeBin[3],\
                g_aTimeBin[2],g_aTimeBin[1],g_aTimeBin[0]);
        }
        else if(c=='s')
        {
            printf_P(PSTR("请按 < 年 月 日 星期 小时 分 秒 > 格式输入 : \n"));
            scanf ("%d,%d,%d,%d,%d,%d,%d",\
                tmp+6,tmp+5,tmp+4,tmp+3,tmp+2,tmp+1,tmp);

            for(i=0;i<7;i++)
                g_aTimeBin[i]=(uchar)tmp[i];

            RtcBinToBCD();
            RtcUpdateData(RTC_WRITE);

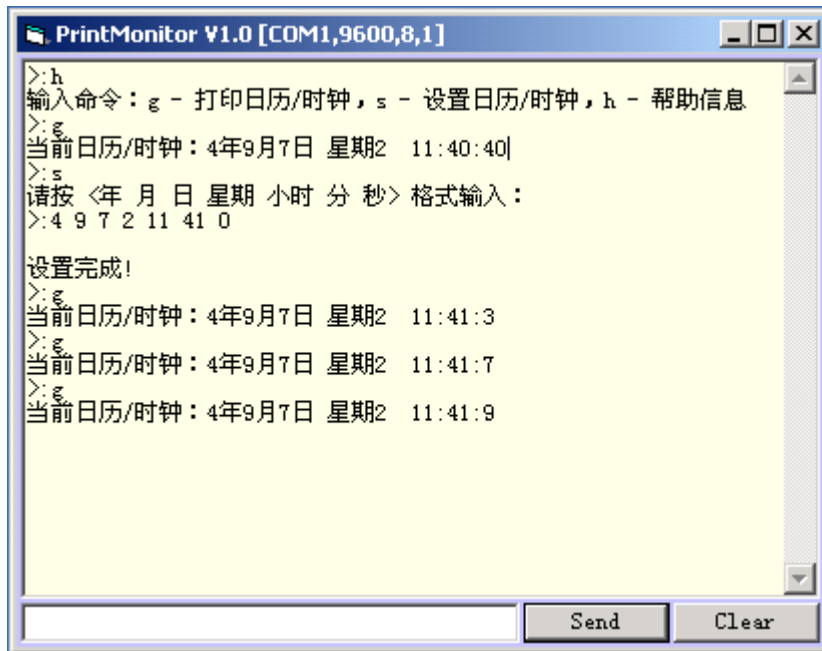
            printf_P(PSTR("\n 设置完成!\n"));
        }
    }
}
```

```
else if(c=='h')
    printf_P(PSTR("输入命令：g - 打印日历/时钟，s - 设置日历/时钟，\
    h - 帮助信息\n"));
} //main loop
return 0;
}
```

从程序中可以看出，TWI 的读和写都是通过置位 TWINT 来实现的，原因是当总线上启动开始条件后写入的第一个字节中已经有了本次操作的传输方向，TWI 模块记住了这一标记，决定了下一步操作是读还是写。

将编译后的代码写入 Mega8，按第五章中介绍电路连接到计算机就可以用 PrintMonitor 测试程序了，测试结果如图 7-2 所示。

图 7-2 PrintMonitor 测试结果



twi\_start, twi\_writebyte, twi\_readbyte 三个函数操作后均返回 TWI 状态寄存器值，在实际应用中应当检测这样的值，处理出现错误的情况。

在编写上述例子时我精心选择了 TWI 接口芯片 DS1307，原因有两个，首先 DS1307 有 DIP8 封装，易于学习使用。其次，DALLAS 公司（MAXIM 全资子公司）对设计开发的支持较好，可以到 <http://www.maxim-ic.com> 申请免费的 DS1307 样片。这样，就可以在 CA-M8 的基础上很容易的测试上述程序了。

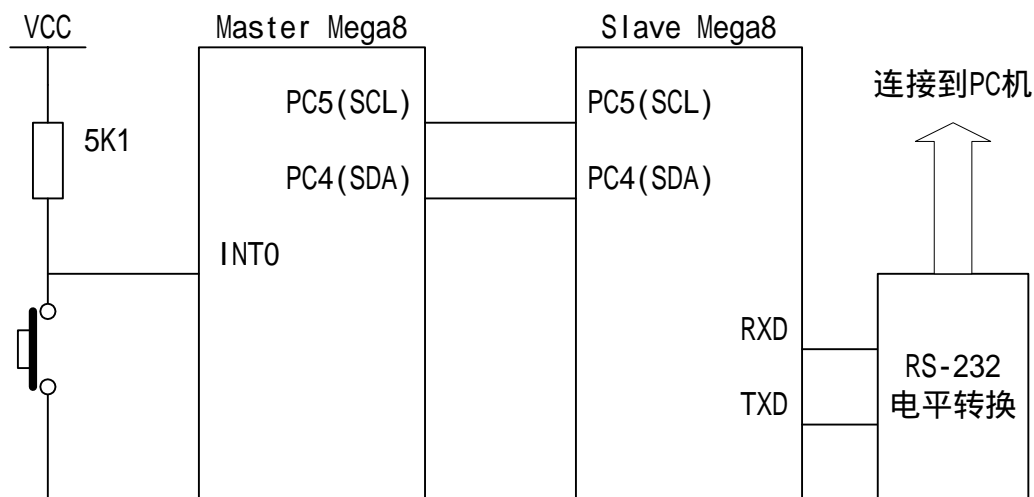
## 7.3 两个 Mega8 间的 TWI 通信

两个 Mega8 用 TWI 总线通信，主要为了说明 Mega8 在 TWI 从模式下工作的编程方法。

### 一. 测试电路

两片 Mega8 间的连接如图 6-3 所示。

图 7-3 两个 MEGA8 间的连接图



Master Mega8 的 PD2 (INT0) 引脚接有一个按键，当键被按下时起动一次的 TWI 写操作，Slave Mega8 的 RXD、TXD 引脚通过电平转换后接到 PC 机串行口，用 PrintMonitor 监测其打印输出的信息。

### 二. 程序设计

#### 1. 主模式单片机程序

主模式程序用查询方式检测并等待按键，当键被按下时向 TWI 口写 0~9。

```

/*
  文件名：master.c
  功 能：两个 Mega8 间的 TWI 通信实验主模式单片机程序

  时 钟：内部 4MHz 振荡器

  2004-09-03
*/

#include <avr/io.h>
#include <avr/delay.h>

```

```
#include <avr/twi.h>

#define uint unsigned int
#define uchar unsigned char
#define WAITPRINTDEBUG DelayMs(100) //为从模式单片机打印调试信息而延时

#define KEY 0X04

#define FREQ 4
#define TWI_ADDRESS 0X32

void DelayMs(uint ms)
{
    uint i;
    for(i=0;i<ms;i++)
        _delay_loop_2(FREQ *250);
}

/*****主模式 TWI 操作部分*****/

//总线上起停条件
void twi_stop(void)
{
    TWCR = _BV(TWINT) | _BV(TWSTO) | _BV(TWEN);
}

//总线上起停开始条件
void twi_start(void)
{
    uchar trycount=0;

    TWCR = _BV(TWINT) | _BV(TWSTA) | _BV(TWEN);
    while ((TWCR & _BV(TWINT)) == 0) ;

    return TW_STATUS;
}

//写一字节
void twi_writebyte(uchar c)
{
    TWDR = c;
    TWCR = _BV(TWINT) | _BV(TWEN);
    while ((TWCR & _BV(TWINT)) == 0);
    return TW_STATUS;
}
```



```
//读一字节 ack: true 时发 ACK , false 时发 NACK
uchar twi_readbyte(uchar *c ,uchar ack)
{
    uchar tmp=_BV(TWINT)|_BV(TWEN);

    if(ack)
        tmp|=_BV(TWEA);
    TWCR=tmp;
    while ((TWCR & _BV(TWINT)) == 0) ;

    *c=TWDR;

    return TW_STATUS;
}
/*****主模式 IIC 操作部分*****结束*****/

//检测按键
uchar WaitKeyDown(void)
{
    uchar key;

    while(1)
    {
        key=PIND & KEY;
        if( key!=KEY)
        {
            DelayMs(30);
            key=PIND & KEY;
            if(key!=KEY)
                break;
        }
        DelayMs(1);
    }

    while((PIND & KEY)!=KEY)
        DelayMs(10);

    return key;
}

int main(void)
{
    uchar i;
```

```
//便能 SCL、SDA 引脚内部上拉电阻
DDRC=0;
PORTC=0X30;

//
DDRD=0;
PORTD=0;

TWBR=73;//波特率

while(1)
{
    WaitKeyDown();

    twi_start();
    WAITPRINTDEBUG;
    twi_writebyte(TWI_ADDRESS|TW_WRITE);
    WAITPRINTDEBUG;
    for(i=0;i<10;i++)
    {
        twi_writebyte(i);
        WAITPRINTDEBUG;
    }
    twi_stop();
}
}
```

## 2. 从模式单片机程序

从模式程序用查询方式等待 TWI 端口中断标志，在 TWI 通信的不同阶段从 UART 打印出对应测试信息。

```
/*
    文件名：slave.c
    功 能：两个 Mega8 间的 TWI 通信实验从模式单片机程序

    时 钟：外部 4MHz 晶振

    2004-09-03
*/

#include <avr/io.h>
#include <avr/twi.h>
```

```
#include <stdio.h>

#define uint unsigned int
#define uchar unsigned char

#define TWI_ADDRESS 0X32

//标准 I/O 输出函数
int usart_putchar(char c)
{
    if(c=='\n')
        usart_putchar('\r');
    loop_until_bit_is_set(UCSRA,UDRE);
    UDR=c;
    return 0;
}

//初始化
void IoInit(void)
{
    //使能 SCL、SDA 引脚内部上拉电阻
    DDRC=0;
    PORTC=0X30;

    //串口初始化
    UCSRB=_BV(RXEN)|_BV(TXEN); /* (1<<RXCIIE)|(1<<TXCIIE) */
    UBRRL=25; //9600 baud 6MHz:38 4MHz:25

    //UART 用于标准 I/O 输入输出
    fdevopen(usart_putchar,0,0);

    //TWI 接口初始化,从器件模式
    TWAR=TWI_ADDRESS | _BV(TWGCE);
    TWCR=_BV(TWEA) | _BV(TWEN);
}

int main(void)
{
    uchar i,j=0;

    IoInit();

    while(1)
    {
```

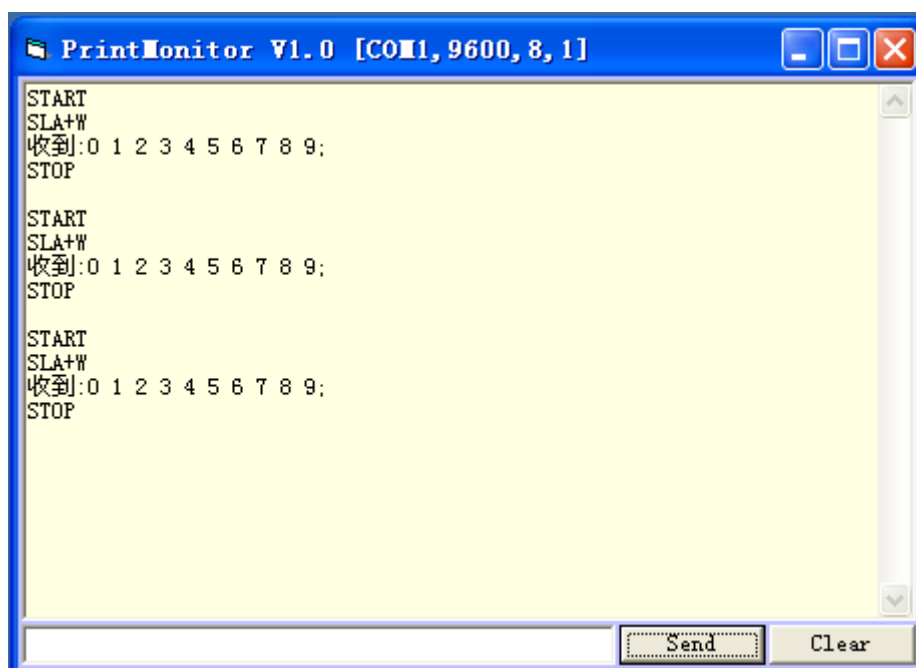
```
while ((TWC & _BV(TWINT)) == 0);
i=TW_STATUS;

switch(i)
{
    case TW_SR_SLA_ACK:
        printf("START\nSLA+W\n");
        break;
    case TW_SR_DATA_ACK:
        if(j==0)
            printf("收到:%d",TWDR);
        else
            printf(" %d",TWDR);
        j++;
        break;
    case TW_SR_STOP:
        printf(";\nSTOP\n\n");
        j=0;
        break;
    default:
        printf("error:%x",(int)i);
        break;
}
TWC=_BV(TWEA) | _BV(TWEN)|_BV(TWINT); //清除 TWINT 位
}
```

### 三. 测试结果

从模式程序输出结果如图 7-4 所示，当主模式单片的键被按下后从单片机先检测到 START 条件，然后收到本机 SLA+W，再接收 1~9 的数据，最后检测到 STOP 条件。

图 7-4 从模式程序打印输出结果



## 第八章 BootLoader 功能应用

### 8.1 BootLoader 功能介绍

BootLoader 提供我们通常所说的 IAP ( In Application Program ) 功能。

多数 Mega 系列单片机具有片内引导程序自编程功能 ( BootLoader )。MCU 通过运行一个常驻 FLASH 的 BootLoader 程序，利用任何可用的数据接口读取代码后写入自身 FLASH 存储器中，实现自编程目的。

下面用 ATMEGA8 来说明此功能

ATMEGA8 片内具有 8K 的 FLASH 程序存储器，BootLoader 功能将其分为 应用程序区 ( Application section ) 和引导加载区 ( Boot loader section )，通过设置熔丝位 BOOTSZ0 和 BOOTSZ1 可将应用区和引导区的大小分别设置成 6K/2K、7K/1K、7K512B/512B 和 7K768B/256B。详细请参考 ATMEGA8 数据手册。

另外，ATMEGA8 还有一个熔丝位 BOOTRST 用于设置复位向量，当 BOOTRST 未被编程时器件的复位从应用程序区首地址 ( 既 0 ) 开始执行，当 BOOTRST 被编程时器件的复位从引导区首地址开始执行。

使用 BootLoader，首先应该根据 BootLoader 程序的大小设置好 BOOTSZ0 和 BOOTSZ1 熔丝位，然后编程 BOOTRST 熔丝位使单片机的复位从引导区开始执行，之后要把 BootLoader 程序定位并写入到引导区 ( 首地址取决于熔丝位 BOOTSZ0 和 BOOTSZ1 的编程状态 )。以上过程需要使用 ISP 或并行编程方式来实现。

当单片机上电复位后 BootLoader 程序开始执行，它通过 USART、TWI 或其它方式从计算机或其它数据源读应用区代码并写入到应用区。事实上 BootLoader 程序有能力读写整个 FLASH 存储器，包括 BootLoader 程序所在的引导区本身，只是写自身所在的引导区的应用较少见。

### 8.2 avr-libc 对 BootLoader 的支持

avr-libc 提供一组 C 程序接口 API 来支持 BootLoader 功能。

包含：

```
#include <avr/boot.h>
```

在 boot.h 中这些 API 均为预定义宏，以下为其主要几个宏。

**boot\_page\_erase ( address )**

擦除 FLASH 指定页

其中 address 是以字节为单位的 FLASH 地址

**boot\_page\_fill ( address, data )**

填充 BootLoader 缓冲页 ,address 为以字节为单位的缓冲页地址(对 mega8 :0~64), 而 data 是长度为两个字节的字数据,因此调用前 address 的增量应为 2。此时 data 的高字节写入到高地址,低字节写入到低地址。

**boot\_page\_write ( address )**

boot\_page\_write 执行一次的 SPM 指令,将缓冲页数据写入到 FLASH 指定页。

**boot\_rww\_enable ( )**

RWW 区读使能

根据自编程的同时是否允许读 FLASH 存储器,FLASH 存储器可分为两种类型:可同时读写区 (RWW Read-While-Write) 和 非同时读写区 (NRWW Not Read-While-Write)。对于 MEGA8 RWW 为前 6K 字节 NRWW 为后 2K 字节。引导加载程序对 RWW 区编程时 MCU 仍可以从 NRWW 区读取指令并执行,而对 NRWW 区编程时 MCU 处于挂起暂停状态。

在对 RWW 区自编程(页写入或页擦除)时,由硬件锁定 RWW 区,RWW 区的读操作被禁止,在对 RWW 区的编程结束后应当调用 boot\_rww\_enable() 使 RWW 区开放。

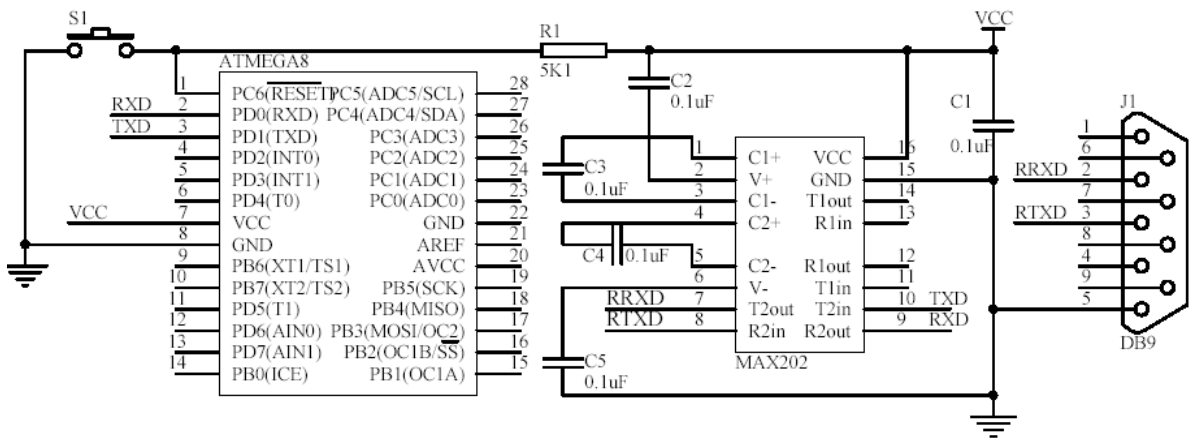
### 8.3 BootLoader 应用实例

本节介绍一种 BootLoader 的一个简单应用实例 - LuckyProg M8BL。BootLoader 程序通过 UART 与计算机通信,执行读、写和执行 FLASH 应用区的操作。

一. 硬件:

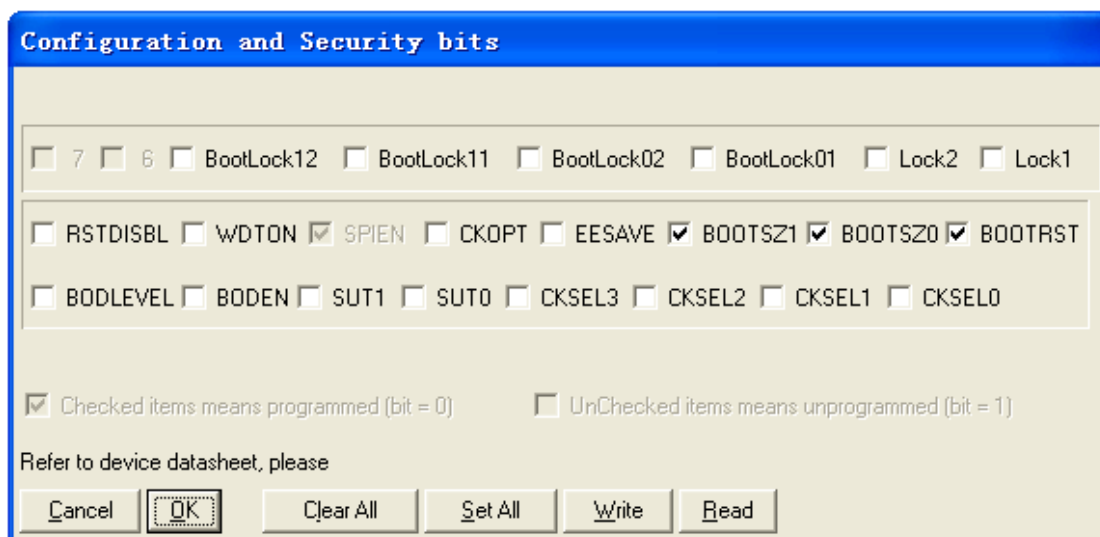
如图 8-1,MEGA8 的 UART 端口通过 MAX202 的电平转换后与计算机 RS-232 串行通信口连接。

图 8-1 LuckyProg M8BL 测试电路



MEGA8 熔丝位设置图如图 8-2 所示，：引导加载区分配 2048 个字节，BOOTRST 被编程（复位后从引导区开始执行），时钟源选择了外部晶振。

图 8-2 Mega8 熔丝位设置



## 二．引导加载程序

引导加载程序不使用中断，查询方式读写 UART，响应 UART 传来的读当前页、写当前页、设置当前页地址、运行应用区代码和 BootLoader 程序检测等命令。每读或写一页当前页地址增一。

“运行应用区代码”命令将 MCU 引导到应用程序区的首地址，在没有特殊情况下只能用硬件复位方式将 MCU 重新引导到引导加载区后才能执行下一次的程序加载操作。

引导加载程序清单：

```
/*  
  
LuckyProg M8BL 引导加载程序 V1.0  
文件 名    :   mboot.c  
器 件    :   ATMEGA8  
时 钟    :   4MHz  
作 者    :   芯 艺  
编 译    :   WinAVR-20040720  
  
*/  
  
#include <avr/io.h>  
#include <avr/pgmspace.h>
```



```
#include <avr/boot.h>

#define uchar unsigned char
#define uint unsigned int

#define  FREQ      4
#define  UART_ACK  0XAA

#define  PAGE_SIZE 64    //按字节

uint g_wPageIndex=0;
uchar g_aPageTemp[PAGE_SIZE];

void (*reset)(void)=0x0000;

void uart_putc(uchar c)
{
    while( !(UCSRA & (1<<UDRE)) );
    UDR=c;
}

uchar uart_getc(void)
{
    while( !(UCSRA & (1<<RXC)) );
    return UDR;
}

void WritePage(void)
{
    uchar i;

    // 接收当前页面数据
    for(i=0;i<PAGE_SIZE;i++)
        g_aPageTemp[i]=uart_getc();

    // 页擦除操作
    boot_page_erase(g_wPageIndex<<6);
    while(boot_rww_busy())
        boot_rww_enable();

    // 填充缓冲页
    for(i = 0; i < PAGE_SIZE; i += 2)
        boot_page_fill((unsigned long)i,*((uint*)(g_aPageTemp +i)));
}
```

```
// 页写入操作
boot_page_write(g_wPageIndex<<6);
while(boot_rww_busy())
    boot_rww_enable();

g_wPageIndex++;
}

void ReadPage(void)
{
    uchar i;

    for(i=0;i<PAGE_SIZE;i++)
        uart_putc(pgm_read_byte(i+(g_wPageIndex*PAGE_SIZE)));

    g_wPageIndex++;
}

int main(void)
{
    uchar tmp;

    //uart 初始化
    UBRRH=0;
    UBRRL=25;//9600 baud 6MHz:38 4MHz:25
    UCSRB=(1<<RXEN)|(1<<TXEN);

    while(1) //main loop
    {
        tmp=uart_getc();//recv command

        switch(tmp)
        {
            case 0xB0://设置页地址
                g_wPageIndex=uart_getc();
                uart_putc(g_wPageIndex);
                break;

            case 0xBF://运行用户程序
                reset();
                break;

            case 0xAF://写一页
                WritePage();
```

```
        uart_putc(UART_ACK);//应答
        break;

    case 0xA0://读一页
        ReadPage();
        break;

    case UART_ACK://回应检测命令
        uart_putc(UART_ACK);
        break;

    default:
        break;
} //switch

} //main loop
} //main
```

编译和链接：

用 Mfile 生成一个 makefile ,设置好以下变量

```
MCU = atmega8
TARGET = mboot
```

之后要做一个额外的修改，那就是将程序入口定位到引导区首地址，这是通过指定 .text 段链接地址来实现的。

在 makefile 中找到如下代码

```
# Linker flags.
# -Wl,...:    tell GCC to pass this to linker.
#   -Map:      create map file
#   --cref:    add cross reference to  map file
LD_FLAGS = -Wl,-Map=$(TARGET).map,--cref
LD_FLAGS += $(EXTMEMOPTS)
LD_FLAGS += $(PRINTF_LIB) $(SCANF_LIB) $(MATH_LIB)
```

并在下边插入以下一行：

```
LD_FLAGS += -Wl,--section-start=.text=0x1800
```

在命令行，mboot.c 和 makefile 所在目录键入 Make 命令，生成引导加载程序 mboot.hex，将它通过编程器或下载线写入到 mega8。

我使用 WinAVR-20040720 编译后生成代码大小为 512 字节，这样，引导区可缩小到从 FLASH 末尾 512 字节处。

### 三．上位机程序

上位机程序是由 Visual C++ 6.0 编写，在 Windows XP 上运行。运行界面如图 8-3：

图 8-3 LuckyProg Mega8 BootLoader V1.0 操作界面



“打开”和“重载”用于打开程序文件，目前只支持二进制（.bin）格式。

“写入”命令将打开的有效代码分页后顺序的向 BootLoader 程序发送“写一页”命令。

“校验”命令通过发送“读一页”命令的方式将打开的代码与 mega8 Flash 内容比较。

“运行”对应 BootLoader 程序的“运行应用区代码”命令。

上位机程序的编写属于 WINDOWS 应用程序编程，这里不再详细描述。

要注意的是，LuckyProg Mega8 BootLoader V1.0 默认使用 COM1 端口，如果要使用其它口需改动源代码。

请到 <http://bitfu.zj.com> 下载 LuckyProg Mega8 BootLoader V1.0 的可执行文件或 VC 源代码。

## 8.4 基于 LuckyProg2004 的 BootLoader 程序

从 LuckyProg2004 最初设计思路中可以发现，引导加载程序利用它是完全可能的。LuckyProg2004 不针对某一特定器件，只管按一定的通信协议将程序文件（通常是.hex 格式）内容发送到串行口，这正是 BootLoader 程序所需要的。以下是在 CA-M8 实验板上实现的 ATmega8 引导加载程序，它实现了 Flash 和 Eeprom 的自编程功能。

### 一. 程序清单：

```
/*
   LuckyProg2004 Mega8 引导加载程序

   文件名   :   demo.c
   更新时间:   2004-11-16

   硬 件:   CA-M8
   时 钟:   外部 4MHz
   编 译:   WinAVR-20040720

*/

#include <avr/io.h>
#include <avr/pgmspace.h>
#include <avr/eeprom.h>
#include <avr/boot.h>

#include "lp2004.h"

#define uchar unsigned char
#define uint unsigned int

#define FREQ 4
//数据包长度设置成 M8 页长度（按字节）
//PAGE_SIZE == LP_PACKET_SIZE
#define PAGE_SIZE 64
#define DEVICE_ID 8
#define MAX_PAGE_COUNT 96

#define SET_RED_LED PORTB&=~_BV(PB1)
#define CLR_RED_LED PORTB|=_BV(PB1)
```

```
#define SET_YEL_LED PORTB&=~_BV(PB0)
#define CLR_YEL_LED PORTB|=_BV(PB0)

uint g_wPageIndex=0;           //读或写例程中用于索引页面
uchar g_aPageTemp[PAGE_SIZE]; //数据包缓冲区
uchar g_aCommand[5];          //命令缓冲区

void (*reset)(void)=0x0000;

void uart_putc(uchar c)
{
    while( !(UCSRA & (1<<UDRE)) );
    UDR=c;
}

uchar uart_getc(void)
{
    while( !(UCSRA & (1<<RXC)) );
    return UDR;
}

//写 flash
void WriteFlash(void)
{
    uchar i;
    uint PageCount=g_aCommand[2];
    PageCount<<=8;
    PageCount+=g_aCommand[1];
    for(g_wPageIndex=0;g_wPageIndex<PageCount;g_wPageIndex++)
    {
        uart_putc(LP_ACK);
        // 接收当前页面数据
        for(i=0;i<PAGE_SIZE;i++)
            g_aPageTemp[i]=uart_getc();
        //6K 以上的不写
        if(g_wPageIndex<MAX_PAGE_COUNT)
        {
            // 页擦除操作
            boot_page_erase(g_wPageIndex<<6);
            while(boot_rww_busy())
                boot_rww_enable();

            // 填充缓冲页
            for(i = 0; i < PAGE_SIZE; i += 2)
                boot_page_fill((unsigned long)i,*((uint*)(g_aPageTemp+i)));
        }
    }
}
```

```
        // 页写入操作
        boot_page_write(g_wPageIndex<<6);
        while(boot_rww_busy())
            boot_rww_enable();
    }

}

uart_putc(LP_ACK);
}
//读 flash
void ReadFlash(void)
{
    uchar i;
    uint PageCount=g_aCommand[2];
    PageCount<<=8;
    PageCount+=g_aCommand[1];
    uart_putc(LP_ACK);
    for( g_wPageIndex=0;g_wPageIndex<PageCount;g_wPageIndex++)
    {
        uart_getc();
        if(g_wPageIndex<MAX_PAGE_COUNT)
        {
            for(i=0;i<PAGE_SIZE;i++)
                uart_putc(pgm_read_byte(i+(g_wPageIndex*PAGE_SIZE)));
        }
        else//6K 以上的页面
        {
            for(i=0;i<PAGE_SIZE;i++)
                uart_putc(0xff);
        }
    }
}
//写 eeprom
void WriteEeprom(void)
{
    uchar i;
    uint PageCount=g_aCommand[2];
    PageCount<<=8;
    PageCount+=g_aCommand[1];
    for(g_wPageIndex=0;g_wPageIndex<PageCount;g_wPageIndex++)
    {
        uart_putc(LP_ACK);
```

```
// 接收页面数据
for(i=0;i<PAGE_SIZE;i++)
    g_aPageTemp[i]=uart_getc();

    eeprom_write_block (g_aPageTemp,PAGE_SIZE *g_wPageIndex, PAGE_SIZE);
}
uart_putc(LP_ACK);
}
//读 eeprom
void ReadEeprom(void)
{
    uchar i;
    uint PageCount=g_aCommand[2];
    PageCount<<=8;
    PageCount+=g_aCommand[1];
    uart_putc(LP_ACK);
    for( g_wPageIndex=0;g_wPageIndex<PageCount;g_wPageIndex++)
    {
        uart_getc();

        for(i=0;i<PAGE_SIZE;i++)
            uart_putc(eeprom_read_byte(i+(g_wPageIndex*PAGE_SIZE)));
    }
}
int main(void)
{
    uchar tmp;

    DDRB=_BV(PB0)|_BV(PB1);
    PORTB=0X03;

    //uart 初始化
    UBRRH=0;
    UBRRL=25;//9600 baud 4MHz:25
    UCSRB=_BV(RXEN)|_BV(TXEN);

    while(1)
    {
        SET_YEL_LED;
        CLR_RED_LED;
        tmp=uart_getc();
        if(tmp==LP_ACK)
        {
            uart_putc(LP_ACK);
```



```
        continue;
    }
    else if(tmp!=DEVICE_ID)
        continue;
    uart_putc(LP_ACK);
    SET_RED_LED;
    CLR_YEL_LED;
    for(tmp=0;tmp<5;tmp++)
        g_aCommand[tmp]=uart_getc();

    switch(g_aCommand[0])
    {
    case LP_WRITE_FLASH    :
        WriteFlash();
        break;
    case LP_READ_FLASH     :
        ReadFlash();
        break;
    case LP_WRITE_EEPROM:
        WriteEeprom();
        break;
    case LP_READ_EEPROM:
        ReadEeprom();
        break;
    case LP_ERASE_DEVICE:
        uart_putc(LP_ACK);
        break;
    case LP_WRITE_BITS:
        break;
    case LP_READ_BITS:
        break;
    case LP_SCAN_HARDWAR:
        break;
    case LP_RESET_DEVICE:
        uart_putc(LP_ACK);
        reset();
        break;
    default:
        break;
    }
}
}
```

## 二 . LuckyProg2004 配置文件的生成

利用 “ 器件配置文编辑器 ” 生成的 lpc 文件如图 8-4 所示。为了不改变 NRWW 区代码把 Flash 大小设置成 6K 字节。

图 8-4 操作 ATmega8 引导加载程序的配置文件



按另存按钮将配置保存到一个 .lpc 文件中，再将它添加到 LuckyProg2004 的器件列表中，便可使用了。

## 第九章 汇编语言支持

### 9.1 C 代码中内联汇编程序

一些对 AVR 硬件的操作须遵守特定的时序，例如要禁止 WatchDog，先同时置位 WDTOE 和 WDE 后在四个时钟周期内清零 WDE 才可实现。但在 C 程序的置位 WDE 操作是否在四个时钟内完成的，不够明确，所以类似的操作通常用直接的汇编程序完成。

avr-gcc 提供一种内联汇编机制，支持 C 与汇编程序的关联编译。使上述问题的解决变得比 C 和汇编程序单独编译链接更方便。

#### 内联汇编声明

示例：`asm("in %0,%1 " : "=r " (value) : "I" (_SFR_IO_ADDR(PORTD)));`

如上例所示，嵌入的汇编由四部分组成：

- (1) 汇编指令本身，它是个表达汇编指令格式的字符串，示例中用为“in %0,%1”。
- (2) 由逗号隔开的输出操作数列表，它是个 C 表达式列表，示例中为“=r”(value)。
- (3) 由逗号隔开的输入操作数列表，它是个 C 表达式列表，示例中为：“I”(\_SFR\_IO\_ADDR(PORTD))。
- (4) Clobber(被破坏的)寄存器指示列表，示例中为空

本例实现的是读取 PORTD 寄存器到 C 变量 value 中，由于 I/O 寄存器常量是按 I/O 寄存器在内部存储器中的统一地址定义的，所以要用宏\_SFR\_IO\_ADDR 来将它转换成 I/O 寄存器相对地址供汇编指令使用。

汇编指令中用符号“%”后加索引值来引用输入输出操作数列表中的操作数，%0 引用第一个操作数，%1 引用第二个操作数，依次类推。在本例中：

%0 引用“=r”(value)

%1 引用“I”(\_SFR\_IO\_ADDR(PORTD))。

内联汇编声明格式：

```
asm(code:output operand list : input operand list [: clobber list])
```

如上声明格式，四部分之间用符号“:”隔开，Clobber 寄存器指示部分空时可以忽略，其它部分可空但不可忽略。例如：

```
asm volatile("cli" : : );
```

如果在后续的 C 代码中没有使用到汇编代码中使用的变量，则在编译器的优化操作将这指令省略，为了防止这种情况的发生，需要在内联声明中指定 volatile 属性，如：

```
asm volatile("in %0,%1 " : "=r " (value) : "I" (_SFR_IO_ADDR(PORTD)));
```

volatile 关键字强制编译器不论在后续代码是否出现变量 value 此操作执行代码作都

要生成。  
汇编指令

(1) 在引用操作数时用百分号后的字母(A, B ...)代表操作数第几字节(A 为低字节), 随后的数字代表第几个操作数, 下面为一个 32 位数据交换高低 16 位的例程:

```
asm volatile( "mov __tmp_reg__,%A0;  
             mov %A0,%D0;  
             mov %D0,__tmp_reg__;  
             mov __tmp_reg__,%B0;  
             mov %B0,%C0;  
             mov %C0,__tmp_reg__ "  
             : "=r" (value)  
             : "0" (value)  
             )
```

如下表 %A0, %B0, %C0, %D0 分别代表 0 号变量四个字节

31.....24	23.....16	15.....8	7.....0	value
%D0	%C0	%B0	%A0	引用

(2) 在一次内联声明中的汇编指令部分可包含多个汇编指令, 如:

```
asm volatile("nop\n\t"  
            "nop\n\t"  
            "nop\n\t"  
            "nop\n\t"  
            ::);
```

要注意的是用换行和 TAB 字符( " \n\t " )将它们隔开。

(3) 当操作数为常数时可直接写入到汇编指部分, 例如:

```
asm volatile("in %0,%1 " : "=r" (value) : "I" (_SFR_IO_ADDR(PORTB)));
```

等同于

```
asm volatile("in %0,0X18 " : "=r" (value) :);
```

但如果写成

```
asm volatile("in %0, _SFR_IO_ADDR(PORTB)" : "=r" (value) :);
```

编译器就会报告错误, 由于汇编指令部分是按字符串形式给出, C 预处理器不处理字符串内容, 在传递到汇编器时预处理宏\_SFR\_IOADDR(PORTB)没有被正确的值替换, 所以导致了错误。因此在汇编指令部分是不可使用 C 预处理宏定义的。

尽管如此, avr-gcc 还是提供了一些特殊寄存器的表达符号, 在汇编指令部分里它们总

是可以直接书写，表 9-1 列出了这些符号和对应寄存器。

表 9-1 专用寄存器定义

符号	寄存器
__SREG__	状态寄存器 SREG ( 0X3F )
__SP_H__	堆栈指针高字节 ( 0X3E )
__SP_L__	堆栈指针低字节 ( 0X3E )
__tmp_reg__	R0
__zero_reg__	R1, 对 C 代码来说其值永远为 0

R0 被 C 编译器看作是临时寄存器，在汇编中引用 R0 后无需恢复其原来的内容。

### 输入/输出操作数

输入/输出操作数列表是个 C 表达式列表，用约束符的方式通知编译器操作数属性，例如在以下代码中：

```
asm("in %0,%1 " : "=r " (value) : "I" (_SFR_IO_ADDR(PORTD)));
```

“=r”说明了表达式 value 结果要分配到 r0 ~ r31 中任意一个寄存器中处理。这样编译器会根据这一描述并关联其它程序分配一合适的寄存器给 value；

表 9-2 列出了约束符及其意义

表 9-2 操作作数约束符

约束符号	适用于	范围
a	后半部分寄存器	R16 ~ R23
b	指针	Y, Z
d	后半部分寄存器	R16 ~ R31
e	指针	X, Y, Z
G	浮点常数	0 . 0
I	6 位正常数	0 ~ 63
J	6 位正常数	-63 ~ 0
K	常整数	2
L	常整数	0
l	前半部分寄存器	R0 ~ R15
M	8 位正常数	0 ~ 255
N	常整数	1
O	常整数	8, 16, 24
P	常整数	1
q	堆栈指针寄存器	SPH:SPL

r	全部寄存器	R0 ~ R31
t	暂存寄存器	R0
w	寄存器对	R24 , R26 , R28 , R30
x	指针 X	
y	指针 Y	
z	指针 Z	
约束符前可加的 修饰符	<b>修饰符</b>	<b>意义</b>
	=	只 写
	+	读 写 ( 内联汇编不支持 )
	&	寄存器只用做输出

注:输出操作数必须为只写操作数, 输入操作数为只读,c 表达式结果是左置的。

表 9-3 详细列出了 AVR 汇编助记符和对应操作数所应使用的约束符

表 9-3 AVR 指令约束符对应表

助 记 符	约 束 符	助 记 符	约 束 符
adc	r,r	add	r,r
adiw	w,I	and	r,r
andi	d,M	asr	r
bclr	I	bld	r,I
brbc	I,label	brbs	I,label
bset	I	bst	r,I
cbi	I,I	cbr	d,I
com	r	cp	r,r
cpc	r,r	cpi	d,M
cpse	r,r	dec	r
elpm	t,z	eor	r,r
in	r,I	inc	r
ld	r,e	ldd	r,b
ldi	d,M	lds	r,label
lpm	t,z	lsl	r

lsr	r	mov	r,r
movw	r,r	mul	r,r
neg	r	or	r,r
ori	d,M	out	I,r
pop	r	push	r
rol	r	ror	r
sbc	r,r	sbc	d,M
sbi	I,I	sbic	I,I
sbiw	w,I	sbr	d,M
sbr	r,I	sbrs	r,I
ser	d	st	e,r
std	b,r	sts	label,r
sub	r,r	subi	d,M
swap	r		

在输入输出操作数使用同一个寄存器的情况下可用操作数索引作为约束符,表示操作数使用与指定索引操作数同一寄存器,如:

```
asm volatile( "SWAP %0" : "=r" (value) : "0" (value));
```

约束符“0”告诉编译器使用与第一个操作数相同的寄存器作为输入寄存器,

有时即使用户没有指定,编译器也有可能使用相同的寄存器作为输入/输出,为避免此类现象可以在输出操作增加修饰符“&”。如下例:

```
asm volatile( " in %0,%1;
out %1,%2 "
: "&r" (input)
: "l" (port), "r" (output)
);
```

此例的目的是读入端口数据后给端口写入另一个数据. 修饰符“&”防止了input和output两个操作数被分配到同一个寄存器

## Clobber

Clobber 指示使编译器在调用汇编程序的前后分别保存和重新装入指定的寄存器内容。

如果指定的寄存器不仅一个要用逗号隔开。

如果用户在汇编代码里直接使用了没有作为操作数声明的寄存器,就需要在 Clobber 里声明以通知编译器. 表 8-4 列出了一个加一原子操作示例及编译后生成的汇编代码。

表 8-4 Clobber 示例

嵌入语句	编译结果
<pre>asm volatile( "cli; ld r24,%a0; inc r24; st %a0,r24; sei " : : "z" (ptr) : "r24" );</pre>	<pre>CLI LD R24 , Z INC R24 ST Z , R24 SEI</pre>

以下为考虑编译器优化的例程

```
{  
    uint8_t s;  
    asm volatile(  
        "in %0, __SREG__"           "\n\t"  
        "cli"                       "\n\t"  
        "ld __tmp_reg__, %a1"       "\n\t"  
        "inc __tmp_reg__"           "\n\t"  
        "st %a1, __tmp_reg__"       "\n\t"  
        "out __SREG__, %0"          "\n\t"  
        : "=&r" (s)  
        : "e" (ptr)  
    );  
}
```

程序是将 ptr 指针指向 RAM 字节加一的原子操作。由于编译器不知道汇编程序修改 RAM 数据,在执行汇编程序前此数据可能暂时保存到其它寄存器中,而汇编程序执行结束后编译器没有更新寄存器,导致可能的错误产生。为了避免此类错误的发生可在内联汇编程序中指定 Clobber :“memory”,这将通知编译器汇编程序可能修改 RAM,执行完代码后要重新装载与被修改 RAM 相关的寄存器。

正确的代码应如下：

```
{  
    uint8_t s;  
    asm volatile(  
        "in %0, __SREG__"           "\n\t"  
        "cli"                       "\n\t"  
        "ld __tmp_reg__, %a1"       "\n\t"  
        "inc __tmp_reg__"           "\n\t"  
        "st %a1, __tmp_reg__"       "\n\t"  
        "out __SREG__, %0"          "\n\t"  
        : "=&r" (s)  
        : "e" (ptr)  
    );  
}
```



```
    "in %0, __SREG__"           "\n\t"  
    "cli"                       "\n\t"  
    "ld __tmp_reg__, %a1"       "\n\t"  
    "inc __tmp_reg__"           "\n\t"  
    "st %a1, __tmp_reg__"       "\n\t"  
    "out __SREG__, %0"          "\n\t"  
    : "=&r" (s)  
    : "e" (ptr)  
    : "memory"  
);  
}
```

解决此类问题的另外一个更好的方法是将指针声明时指定 `volatile` 属性,如下所示:

```
volatile uint8_t *ptr
```

这样,一旦指针指向的变量发生变化,编译器就会重新加载最新的数值。

Clobber 的更多应用在直接使用寄存器上。Clobber 的指定示必会影响编译器优化过程,应当尽可能减少 Clobber 指示。

### 汇编宏应用

将汇编程序定义成宏后写进头文件,这样可以方便的多次使用该段程序, `avr-libc` 的包含头文件里有很多这样的例子,可从 `WINAVR` 安装目录 `avr/include` 中找到很多示例。

为了避免编译器的警告定义汇编宏时要把 `asm` 和 `volatile` 分别替换成 `__asm__` 和 `__volatile__`,实际上它们作用是相同的。

以下是全例子:

```
#define loop_until_bit_is_clear(port,bit) \  
    __asm__ __volatile__ (\  
    "L_%=: " "sbic %0, %1" "\n\t" \  
        "rjmp L_%" \  
        : \  
        : "I" (_SFR_IO_ADDR(port)), \  
        "I" (bit) \  
    )
```

正如上段程序中那样,在汇编中用到标号时可以通过在自义一个标号后加专用宏 `%=` 来告诉编译器预理程序产生一个唯一的标号。

## 9.2 独立的汇编语言支持

像 at90s1200 这样的内部没有 RAM 的 AVR 器件 avr-libc 只提供汇编语言支持,但与其它汇编器不同的是,avr-gcc 使用 C 预处理器,从而使符号常量的定义与 C 程序一样。另外,一个完整的汇编应用程序还要基于与 C 程序一样的应用程序运行框架,既汇编程序不必考虑中断向量表、初始化栈指针等初始化配置过程。

由此可以看出,avr-libc 对汇编语言的支持不是完整的,却能很好的适应与 C 程序的混合编程。

### 一.avr-libc 汇编程序示例

以下是在 avr-libc 用户手册上的一个示例程序,它实现了从 at90s1200 的 PD6 口输出 100KHz 方波。

```

; 系统时钟 : 10.7 MHz
#include <avr/io.h>           ; 包含器件 I/O 端口定义文件,要注意的是,并不是所
                               ; 有 avr-lib 提供的头文件都可以这样包含,如果我们
                               ; 打开一个 ioxxxx.h 就会发现内部都是符号定义,正
                               ; 因为如此它才可以在 C 和汇编中均可包含

work    =    16                ; 为寄存器定义符号,可用#define work 16 替代
tmp     =    17

inttmp  =    19

intsav  =    0

SQUARE  =    PD6

tmconst= 10700000 / 200000    ; 100 kHz => 200000 edges/s
fuzz=   8                      ; # clocks in ISR until TCNT0 is set

        .section .text        ; 指定段

        .global main          ; 外部函数声明,与 C 程序一样 main 标号是应用程序
main:    ; 入口函数,必需将它声明成外部函数使它对于
        ; avr-libc 应用框架可见,应用框架初始化程序的最
        ; 后会跳转到这里

        rcall    ioinit

1:       rjmp    1b            ; 主循环

        .global SIG_OVERFLOW0 ; 定时器 0 中断处理,与 C 程序一样中断函数使用固定

```

```

;名称, 这些名称在<avr/io.h>中已被定义。中断函数
; 必需声明成 .global
SIG_OVERFLOW0:
    ldi    inttmp, 256 - tmconst + fuzz
    out    _SFR_IO_ADDR(TCNT0), inttmp

    in     intsav, _SFR_IO_ADDR(SREG)

    sbic   _SFR_IO_ADDR(PORTD), SQUARE
    rjmp   1f
    sbi    _SFR_IO_ADDR(PORTD), SQUARE
    rjmp   2f
1:    cbi    _SFR_IO_ADDR(PORTD), SQUARE
2:

    out    _SFR_IO_ADDR(SREG), intsav
    reti

ioinit:
    sbi    _SFR_IO_ADDR(DDRD), SQUARE

    ldi    work, _BV(TOIE0)
    out    _SFR_IO_ADDR(TIMSK), work

    ldi    work, _BV(CS00)          ; tmr0: CK/1
    out    _SFR_IO_ADDR(TCCR0), work

    ldi    work, 256 - tmconst
    out    _SFR_IO_ADDR(TCNT0), work

    sei

    ret

.global __vector_default    ; 程序中未指定中断的处理例程, 与中断名称一样
                            ; 名称__vector_default 是固定的, 注意的是必
                            ; 需将它声明成 .global, 如果不重写此例程,
                            ; 应用框架就会在此处插入一个跳转到 0 地址的
                            ; 一条指令(即复位)
__vector_default:
    reti

.end

```

编译:

编译时通常不会直接调用汇编器 (avr-as), 而是通过给 avr-gcc 通用命令接口 (命令 avr-gcc) 设置选项的方式将程序传递到汇编器中。如:

```
avr-gcc -mmcu=at90s1200 -x assembler-with-cpp -o target.elf target.s
```

选项 -x assembler-with-cpp 用于指定语言类型是汇编, target 为文件名。

这样做是因为: 让 avr-gcc 自动调用预处理程序和链接器最终生成可执行目标文件, 预处理器用于解释在汇编程序中定义的常量符号, avr-gcc 调用链接器时会将器件对应的应用程序框架链接进来。

## 9.3 C 与汇编混合编程

### C 编译器使用寄存器约定

(1) r0 和 r1 : 在 C 程序中这两个寄存器用于固定目的, 从不分配局部变量。

r0 是暂存寄存器, 在汇编程序中可以自由的使用, 若在汇编程序中使用了 r0 并且要调用 C 函数则必需在调用 C 函数前要保存它。若在中断函数中使用此寄存器要在使用前保存, 和退出时恢复, C 中断函数会自动保存的恢复 r0。

r1 是零寄存器, C 编译器总认为其内容为零。如果汇编程序要使用这个寄存器, 则在汇编代码返回之前要清零它。中断函数要使用 r1 则必需在退出时将其清零, C 中断函数会自动保存的恢复 r1。

(2) r2-r17、r28-r29 :

局部变量分配寄存器, 汇编程序若改变了这些寄存器内容, 则必需恢复其内容。

(3) r18-r27、r30-r31

局部变量分配寄存器, 汇编程序可自由的使用这此寄存器, 无需恢复, 但若在汇编程序中使用了这些寄存并要调用 C 函数, 则需在调用前保护所用的寄存器。

### C 编译器函数调用规则

参数传递: 函数参数按从左至右的规则分别被分配到 r25 到 r18。每个参数从偶数编号寄存器开始分配低字节, 若参数长度为奇数则高编号寄存器闲置。如: 一个 char 类型的参数传递时分配给 r24, 这时 r25 被闲置, 尽管后边还有参数要传递也不会使用 r25。如下表为函数 void test(unsigned char parbyte, unsigned int parword) 的参数分配情况:

R25	R24	R23	R22
闲置	parbyte	parword 高字节	parword 低字节

另外, 如果参数太多以至于 r25 到 r8 无法容纳, 则剩余的部分从堆栈传递。

返回值传递：与参数分配类似，8 位返回值存放到 r24、16 位值分保存到 r25:r24、32 位值保存到 r25:r24:r23:r22、64 位值保存到 r25:r24:r23:r22:r21:r20:r19:r18。

在默认情况下 C 和汇编程序使用同样的函数名，但可以在 C 程序中给函数指定在汇编里调用名，如：

```
extern long Calc(void) asm ("CALCULATE");
```

声明了函数 Calc，在汇编中用符号 CALCULATE 调用。

### 例一． 在 C 程序中调用汇编函数

C 源文件 main.c

```
/*
   cpu:atmega8
   时钟:4MHz
*/

#include <avr/io.h>
#include <avr/delay.h>

#define uchar unsigned char

#define SET_GRN_LED PORTC&=0XFD //PC1 接绿色发光管
#define CLR_GRN_LED PORTC|=0X02

//声明两个汇编函数
void set_grn_led(void);
void clr_grn_led(void);

void DelayMs(unsigned int t)
{
    unsigned int i;
    for(i=0;i<t;i++)
        _delay_loop_2(4*250);
}

int main(void)
{
    //LED 口初始化
    DDRC=0X0F;
    PORTC=0X0F;
}
```

```
while(1)
{
    DelayMs(1000);
    set_grn_led();
    DelayMs(1000);
    clr_grn_led();
}
}
```

汇编源文件 asm.s

```
#include <avr/io.h>

.section .text
.global set_grn_led           ;外部函数声明
set_grn_led:                  ;点亮绿发光管
    cbi _SFR_IO_ADDR(PORTC),1
    ret

.global clr_grn_led
clr_grn_led:                  ;熄灭绿发光管
    sbi _SFR_IO_ADDR(PORTC),1
    ret
```

程序实现接在 PC1 口的发光管闪烁

### 例二 . 在汇编程序中调用 C 函数

C 源文件 main.c

```
/*
    cpu:atmega8
    时钟:4MHz
*/

#include <avr/io.h>
#include <avr/delay.h>

#define uchar unsigned char

#define SET_GRN_LED PORTC&=0XFD //PC1 接绿色发光管
```

```
#define CLR_GRN_LED PORTC|=0X02
```

```
void DelayMs(unsigned int t) //延时— ms  
{  
    unsigned int i;  
    for(i=0;i<t;i++)  
        _delay_loop_2(4*250);  
}
```

汇编源文件 asm.s

```
#include <avr/io.h>  
  
    .extern DelayMs    ;外部 C 函数声明  
  
    .section .text  
    .global main  
main:  
    ;i/o 初始化  
    ldi r25 , 0x0f  
    out _SFR_IO_ADDR(DDRC),r25  
  
LOOP:  
  
    ldi r25,1        ;参数高字节  
    ldi r24,0xf4    ;参数低字节  
    rcall DelayMs   ;DealyMs(500);  
  
    cbi _SFR_IO_ADDR(PORTC),1  
  
    ldi r25 ,1  
    ldi r24,0xf4  
    rcall DelayMs   ;DelayMs(500);  
  
    sbi _SFR_IO_ADDR(PORTC),1  
  
    rjmp LOOP
```

程序实现接在 PC1 口的发光管每一秒闪烁一次

## 第十章 C++语言支持

可以使用 C++语言组织程序，使程序更具有可读性和可维护性。但使用 C++时要注意的是：Avr-gcc 没有提供 C++语言标准类库和标准模板库。并且不支持 new 和 delete 两个操作符。

编译 C++程序可有两种方式，一种是命名源文件时使用 C++后缀（.C、.cpp、cc）然后送到通用命令接口 avr-gcc。作为另一种选择，第二种方式是直接调用 avr-g++ 命令编译 C++程序。要利用 Mfile 生成的 makefile 编译 C++程序时若采用第一种方式需改动 makefile 内的一些规则，并且在当前版本（WinAVR-20040720）生成的 makefile 中 clean 选项会误删除我们的 C++源程序。为此建议使用第二种方式，这时仅仅将 makefile 中的 CC = avr-gcc 改为 CC=avr-g++ 即可。

### 环境配置：

UltraEdit 没有对类的可视化支持，下面介绍如何配置 VC6 IDE 编译我们的 C++程序。

假设我们要在 d 盘下创建一个 work 目录并在 work 目录下编译程序

- 1.打开 VC 点击菜单 File->new，new 对话框的 project 选项卡中选择 Makefile Location 区输入：D:，Project name 区输入: work 点 OK 进入下一步
2. 点 Finish->OK
- 3.点击菜单 Project->Setting 在弹出对话框的 Build Command Line 区输入：make
- 4.新建或现有的程序源文件保存到此目录中,并添加到项目中。用 Mfile 生成 makefile，配置好后保存到此目录中。
- 5.选择 Tools->Option->Directories 在 Show directories for:区选 Executable files 在 Directories:区分别加入目录：winavr 安装目录\bin 和 winavr 安装目录\UTILS\BIN。
- 6.这样就可用工具栏上的 Build (F7) 按钮在 d:\work 下执行 make 命令了。

### 调用 C 函数

由于没有标准的 C++类库，更多的时候需要调用 C 运行库中的函数。C++作为 C 语言的扩展可以与 C 自然混合使用，但 C 与 C++使用的函数名有些差异，C++调用 C 函数时必须用关键字 extern “C” 声明该函数。

幸运的是 avr-libc 的头文件考虑了这一切。它们的头尾分别有如下的代码：  
开始处：

```
#ifdef __cplusplus
extern "C" {
#endif
```

结尾处：





```
#ifdef __cplusplus
}
#endif
```

这将保证当使用 C++ 编译器时这些头文件中的函数自动的被声明为 extern “C”。

## 一个简单示例

### 一．硬件电路

功能：实现交流同步电机正反向控制

原理：对交流电进行过零检测按控制要求同步触发双向晶闸管，使交流同步电机正向或反向运转。

电路如图 10-1 所示，由于 VCC 接了交流输入的一端，Q3 或 Q5 的导通会把相应晶闸管在第一或三象限内触发。Q2 及其周围电阻构成过零检测电路。

### 二．程序设计

```
/*
  交流电机测试程序
  文件名：main.c
  MCU:at90s2313
  时钟：4MHz
*/

#include <avr/io.h>
#include <avr/delay.h>

#define uchar unsigned char
#define uint unsigned int

#define SET_RED_LED PORTD|=_BV(5) //PD5 接红色发光管
#define CLR_RED_LED PORTD&=~_BV(5)

#define SET_GRN_LED PORTD|=_BV(4) //PD4 接绿色发光管
#define CLR_GRN_LED PORTD&=~_BV(4)

class CControl
{
public:
    CControl();
```

```
public:
    uchar m_bCounter; //RunMotor 成员一次触发次数

    void DelayMs(uint ms);
    void RunMotor(uchar direction); //
};

CControl::CControl()
{
    m_bCounter=0;
}

//根据 direction 将相应晶闸管同步触发 m_bCounter 指定次数，并 LED 指示.
void CControl::RunMotor(uchar direction)
{
    if(direction==1)
    {
        SET_GRN_LED;
        CLR_RED_LED;
    }
    else if(direction==2)
    {
        CLR_GRN_LED;
        SET_RED_LED;
    }
    else
    {
        CLR_GRN_LED;
        CLR_RED_LED;
    }

    for(uchar i=0;i<m_bCounter;i++)
    {
        while((PINB&_BV(0))==1);
        while((PINB&_BV(0))==0);

        if(direction==1)
        {
            PORTB|=_BV(PB3);
            DelayMs(2);
            PORTB&=~_BV(PB3);
        }
        else if(direction==2)
        {
```

```
        PORTB|=_BV(PB2);
        DelayMs(2);
        PORTB&=~_BV(PB2);
    }
    else
        PORTB=0;
}
}

void CControl::DelayMs(uint ms)
{
    uint k=0;
    for(k=0;k<ms;k++)
        _delay_loop_2(1000);
}

CControl g_oMotorCtl; //控制对象全局定义

int main(void)
{
    DDRD=_BV(4)|_BV(5); //发光管 I/O 初始化
    PORTD=0X00;

    PORTB=0;           //控制口 I/O 初始化
    DDRB=_BV(PB3)|_BV(PB2);

    g_oMotorCtl.m_bCounter=200; // 200/(50Hz × 2) =2 秒

    g_oMotorCtl.DelayMs(2000);

    while(1)
    {
        g_oMotorCtl.RunMotor(1); //正向 2 秒
        g_oMotorCtl.RunMotor(0); //停止 2 秒
        g_oMotorCtl.RunMotor(2); //反向 2 秒
        g_oMotorCtl.RunMotor(0); //停止 2 秒
    }
}
}
```

使用 C++写 AVR 程序时要充分的测试，毕竟这种 C++语言的支持是有待完善的。

## 附录 1 命令 avr-gcc 选项

### 一 指定目标 CPU 类

- -mmcu=architecture  
指定 AVR 指令集,目前有:

avr1	最简单内核,只提供汇编支持
Avr2	“ Classic ” 内核,最大可达 8K 字节
Avr3	“ Classic ” 内核,可多于 8K 字节
Avr4	“ Enhanced ” 内核,最大可达 8K 字节
Avr5	“ Enhanced ” 内核,可多于 8K 字节

在默认情况下目标代码在 avr2 指令集内产生

注:如果仅指定指令集而没有指定器件类型(-mmcu=MCU Type), <avr/io.h>无法正确包含器件定义文件,所以在代码中需要包含相应的 ioxxxx.h 文件。

- -mmcu=MCU Type  
指定器件类型,下表列出了可指定的 AVR 器件和相应指令集

指令集	MCU名	对应宏
avr1	at90s1200	__AVR_AT90S1200__
avr1	attiny11	__AVR_ATTiny11__
avr1	attiny12	__AVR_ATTiny12__
avr1	attiny15	__AVR_ATTiny15__
avr1	attiny28	__AVR_ATTiny28__
avr2	at90s2313	__AVR_AT90S2313__
avr2	at90s2323	__AVR_AT90S2323__
avr2	at90s2333	__AVR_AT90S2333__
avr2	at90s2343	__AVR_AT90S2343__
avr2	attiny22	__AVR_ATTiny22__
avr2	attiny26	__AVR_ATTiny26__
avr2	at90s4414	__AVR_AT90S4414__
avr2	at90s4433	__AVR_AT90S4433__
avr2	at90s4434	__AVR_AT90S4434__
avr2	at90s8515	__AVR_AT90S8515__
avr2	at90c8534	__AVR_AT90C8534__
avr2	at90s8535	__AVR_AT90S8535__
avr2	at86rf401	__AVR_AT86RF401__
avr3	atmega103	__AVR_ATmega103__
avr3	atmega603	__AVR_ATmega603__
avr3	at43usb320	__AVR_AT43USB320__

avr3	at43usb355	__AVR__AT43USB355__
avr3	at76c711	__AVR__AT76C711__
avr4	atmega8	__AVR__ATmega8__
avr4	atmega8515	__AVR__ATmega8515__
avr4	atmega8535	__AVR__ATmega8535__
avr5	atmega16	__AVR__ATmega16__
avr5	atmega161	__AVR__ATmega161__
avr5	atmega162	__AVR__ATmega162__
avr5	atmega163	__AVR__ATmega163__
avr5	atmega169	__AVR__ATmega169__
avr5	atmega32	__AVR__ATmega32__
avr5	atmega323	__AVR__ATmega323__
avr5	atmega64	__AVR__ATmega64__
avr5	atmega128	__AVR__ATmega128__
avr5	at94k	__AVR__AT94K__

- -morder1
- -morder2

改变寄存器分配顺序，默认为

r24, r25, r18, r19, r20, r21, r22, r23, r30, r31, r26, r27, r28, r29, r17, r16, r15, r14, r13, r12, r11, r10, r9, r8, r7, r6, r5, r4, r3, r2, r0, r1

顺序 1 :

r18, r19, r20, r21, r22, r23, r24, r25, r30, r31, r26, r27, r28, r29, r17, r16, r15, r14, r13, r12, r11, r10, r9, r8, r7, r6, r5, r4, r3, r2, r0, r1

顺序 2 :

r25, r24, r23, r22, r21, r20, r19, r18, r30, r31, r26, r27, r28, r29, r17, r16, r15, r14, r13, r12, r11, r10, r9, r8, r7, r6, r5, r4, r3, r2, r1, r0

- -mint8

指定 int 为 8 位整数，avr-libc 并没有真正提供此选项，默认使用 16 位的整数

- -mno-interrupts

产生的代码在改变堆栈指针时不关中断，默认时状态寄存器 SREG 的内容被保存到暂存寄存器，改变堆栈指针时全局中断被禁止，再把 SREG 恢复。

- -mcall -prologues

使用函数入/出子程序，在一个复杂的函数将使用大量的寄存器，函数的入口和出口需要保存和恢复寄存器，但这将以增加执行速度为代价。

- -init-stack=nnnn

设置堆栈指针初始值为 nnnn。默认时初始化成符号 \_\_stack，运行库的初始化代码里它被置成 RAMEND。

- -mtiny-stack

只改变堆栈指针的低 8 位

- -mon-tablejump

不产生 jump table 指令，jump table 用于优化 switch 语句。当这种优化被关闭时 switch 被解释成顺序比较，这种代码的速度通常比 jump table 慢，但是在一些特殊的 switch 模块中大多数跳转为 default 标签，若优化成 jump table 将浪费大量的 Flash 空间。

- `-mshort -calls`  
在 8K 以上 FLASH 空间的器件中使用 `rjmp/rcall` 为跳转和调用指令,在 `avr2` 和 `avr4` 指令集( `flash` 存储器小于 8K )中默认此选项。在 `avr3` 和 `avr5` 指令集中跳转和调用默认使用 `jmp/call` ,这两个指令虽然跳转和调用范围覆盖整个 FLASH ROM 地址,但是占用更多的程序存储器和执行时间。
- `-msize`  
把地址、大小和每一语句相应的大小注释到汇编代码中,用于调试。
- `-mdeb`  
产生调试信息到 `stderr`

## 二. 选择通用编译器选项

- `-On`  
`gcc` 有多个优化标志, 这些标志不能直接指定, 而是通过优化等级来说明这些优化开关。`-On` 为优化等级指定选项, `n` 代表等级, 可设置成: `0, 1, 2, 3, s`。  
`-O0` 没优化, 不打开优化标志。  
`-O` 和 `-O1` 等同 编译器将尝试缩短代码量和执行时间, 但这将耗费更多的编译时间。  
`-O2` 在 `-O1` 优化级的基础上, 以缩小代码长度为优先为原则进行优化  
`-O3` 优化的最高等级, 在 `-O2` 级优化的基础上将“简单”函数以内连式编译, 获取执行时间却增大了代码量。  
`-Os` 按大小编译, 类似 `-O2`。若没有指定优化等级, 优化项 `-Os` 成为默认。
- `-Wa, assembler-options`  
传递选项 `assembler-options` 到汇编器。
- `-Wl, linker-option`  
传递选项 `linker-option` 到连接器。
- `-g`  
产生可在 `avr-gdb` 里调试的信息。
- `-ffreestanding`  
假定环境变量“`freestanding`”为 C 标准。这将关闭自动内部函数( 尽管如此仍可以在函数名前指定关键字 `__builtin_` 使其按内部方式编译)。这将在编译时对 `void` 声明的 `main` 函数不警告, 这正说明了单片机应用程序并不需要返回数据到外部环境。在程序中用 `strlen()` 函数指定一字符串常量的长度, 通常在编译时编译器直接用实际长度来替代这一调用, 而在指定 `-ffreestanding` 的情况下编译结果总是在运行时调用 `strlen()` 函数。
- `-funsigned-char`  
定义的 `char` 类型数据按 `unsigned char` 编译。若不指定此项 `char` 被认为 `signed char`, 即无符号字节。
- `-funsigned-bitfields`  
将所有 `bitfield` 声明的变量按无符号编译, 默认按有符号编译。
- `-fshort-enums`  
对 `enum` 类型数据, 根据定义范围分配最少的字节。
- `-fpack-struct`  
所有结构数据分配到一起

### 三 Avr-as 汇编器选项

- -mmcu=architecture
- -mmcu=MCU name  
与 avr-gcc 相同
- -mall-opcodes  
关闭针对实际 MCU 的指令检查，汇编时允许任何可能的 AVR 指令。
- -mno-skip-bug  
产生二字跳转指令(CPSE/SBIC/SBIS/SBRC/SBRS)时不警告。早期的 AVR 器件由于硬件的缺陷不能正常执行这些指令。
- -mno-wrap  
对具有 8K 以上存储器空间的器件禁止 RJMP/RCALL 指令。
- --gstabs  
产生汇编行.stabs 调试信息。这可使 avr-gdb 跟踪汇编源程序。此选项不能用在 C 产生的汇编文件的汇编中,因为这种文件已经包含了 C 源文件行号。
- -a[cdhlms=file]  
允许汇编列表,其子选项如下:
  - c 忽略失败条件
  - d 忽略调试信息
  - h 包含高级语言内容
  - l 包含汇编内容
  - m 包含宏展
  - n 不格式整理
  - s 包含符号
  - =file 设置列表文件名不同的子项可同时列出,但=file 项必须最后指定。

### 四 链接器 avr-ld 选项

- -lname  
查找名为 libname.a 的库文件,并从中查找当前未解释的符号.连接器先从安装目录的上下的/avr/lib 下找,后从-L 选项指定的目录中找.
- -Lpath  
指定用户库文件所在路径
- --defsym symbol=expr  
定义全局符号,symbol 以 expr 为其值.
- -M  
打印连接映象到标准输出.
- -Map mapfile  
打印连接映象到文件 mapfile.
- --cref  
将交索引表到 map 文件或标准输出.
- --section-start sectionname=org



---

指定段 `sectionname` 的开始绝对地址为 `org`.

- `-Tbbs org`
- `-Tdata org`
- `-Ttext org`

分别指定 `bbs data text` 段开始地址.

- `-T scriptfile`

用 `scriptfile` 指定的文件替代默认链接脚本文件, 默认链接脚本存储在 `/avr/lib/ldscripts` 下, 以 AVR 指令集命名, 扩展名为 `.x` 的文件中。它描述多少个不同的存储器段需要链接到一起。

## 附录 2 Intel HEX 文件格式描述

Intel Hex 格式文件由多个记录构成，而每条记录又由 6 种字段构成。

### ● 记录基本格式：

RecordMark	RecordLength	LoadOffset	RecordType	Data	Checksum
记录标志	记录长度	装载偏移	记录类型	数据	校验合

#### RecordMark

每一个记录都以包含数据 0x3A 的字节为开头，在 ASCII 字符中表现为“:”。在 HEX 文件中占一个字节。

#### RecordLength

描述该记录包含数据的长度（以字节为单位），最大描述 255 个字节。表现为两个十六进制字符，在 HEX 文件中占两个字节。

#### LoadOffset

描述本记录中的数据在整个存储器空间中的偏移，用四个十六进制字符描述一个十六位数据，在 HEX 文件中占四个字节。

此字段仅用于数据类型的记录，在其它类型的记录中此字段为“0000”四个字符。

#### RecordType

描述记录类型，表现为两个十六进制字符。描述一字节数据，在 HEX 文件中占两个字节。

- “00”：表示数据记录
- “01”：表示文件中最后一个记录
- “02”：描述扩展段地址的记录
- “03”：描述开始段地址的记录
- “04”：描述扩展线性地址的记录
- “05”：描述开始线性地址的记录

#### Data

每一记录包含无规定长度的 Data 字段，描述方法仍是两个十六进制字符表示一字节数据。此字段的长度由本记录的 RecordLength 决定。

数据的解释取决于记录类型（RecordType）。

#### Checksum

校验合字段

校验方法：

在一条记录内 RecordMark (“:”) 段后所有数据（包括 Checksum）按字节相加后得到的 8 位数据为 0。

### ● 记录分类描述

扩展线性地址记录：

（适用于 32 位格式）



记录名	RecordMark	RecordLength	LoadOffset	RecordType	Data	Checksum
	记录标志	记录长度	装载偏移	记录类型	数据	校验合
内容	“:”	“02”	“0000”	“04”	ULBA	
字节数	1	1	2	1	2	1

此记录用于指定 32 位地址的第 16~31 位。32 位绝对地址的计算方法为：

$$ADR=LBA + DRLO + DRI$$

式中 LAB=ULBA<<16

DRLO为数据记录指定的偏移

DRI为当前数据在数据段中的索引

### 扩展段地址记录

(适用于 16 和 32 位格式)

记录名	RecordMark	RecordLength	LoadOffset	RecordType	Data	Checksum
	记录标志	记录长度	装载偏移	记录类型	数据	校验合
内容	“:”	“02”	“0000”	“02”	USBA	
字节数	1	1	2	1	2	1

此记录用于指定段基地址的第 4~19 位。段基地址 SBA=USBA<<3;

绝对线性地址的算法如下：

$$ADDR=SBA+DRLO + DRI$$

### 数据记录

(适用于 8、16 和 32 位格式)

记录名	RecordMark	RecordLength	LoadOffset	RecordType	Data	Checksum
	记录标志	记录长度	装载偏移	记录类型	数据	校验合
内容	“:”	X	-	“00”	-	
字节数	1	1	2	1	X	1

### 开始线性地址记录

(适用于 32 位格式)

记录名	RecordMark	RecordLength	LoadOffset	RecordType	Data	Checksum
	记录标志	记录长度	装载偏移	记录类型	数据	校验合
内容	“:”	“04”	“0000”	“05”	EIP	
字节数	1	1	2	1	4	1

本记录用于指定目标文件中的开始执行地址，EIP指定32位线性地址

### 开始段地址记录

(适用于 16 位和 32 位格式)

记录名	RecordMark	RecordLength	LoadOffset	RecordType	Data 数据		Checksum
	记录标志	记录长度	装载偏移	记录类型	CS	IP	校验合
内容	“:”	“04”	“0000”	“03”			
字节数	1	1	2	1	2	2	1

本记录用于指定目标文件中的开始执行地址，CS和IP分别为8086处理器的两个寄存器内容

### 文件结束记录

(适用于 8、16 和 32 位格式)

记录名	RecordMark	RecordLength	LoadOffset	RecordType	Checksum
	记录标志	记录长度	装载偏移	记录类型	校验合
内容	“:”	“00”	“0000”	“01”	“FF”
字节数	1	1	2	1	1

由于 avr 是个 8 位处理器，只使用适用于 8 位格式的记录类型。