如何使用 AVR-GCC

安装 GNU C for AVR

一．执行安装程序

二．生成链接用的库文件

$(AVR)表示安装的根目录。（在本人系统里为 f:\avrgcc）

生成库文件关键是要运行位于$(AVR)下的 RUN.BAT。原程序如下：

```
    @echo off
    if NOT %AVR%!==! goto install
    rem set environment variables
    set AVR=f:\AVRGCC
    set CC=avr-gcc
    set PATH=.;f:\AVRGCC\bin;%path%
    doskey
:install
    if %1!==! GOTO end
    rem install libc
    cd f:\AVRGCC\lib\avr-libc-20010701\src
    rem first win32_make_dirs will make some errors(I don't know why?)
    make -f makefile-win32 win32_make_dirs
    make -f makefile-win32
```

```
    make -f makefile-win32 install
    make -f makefile-win32 clean
:end
    f:
    cd f:\AVRGCC
    mode con: lines=43
```

要修改为：

```
    @echo off
    if NOT %AVR%!==! goto install
    rem set environment variables
    set AVR=f:\AVRGCC
    set CC=avr-gcc
rem    set PATH=.;f:\AVRGCC\bin;%path%
    doskey
:install
rem    if %1!==! GOTO end
    rem install libc
    cd f:\AVRGCC\lib\avr-libc-20010701\src
    rem first win32_make_dirs will make some errors(I don't know why?)
    f:\AVRGCC\bin\make -f makefile-win32 win32_make_dirs
    f:\AVRGCC\bin\make -f makefile-win32
    f:\AVRGCC\bin\make -f makefile-win32 install
    f:\AVRGCC\bin\make -f makefile-win32 clean
```

:end

f:

cd f:\AVRGCC

mode con: lines=43

在以后的应用中，运行的是修改之前的 RUN.BAT，但要去掉 rem if %1!==! GOTO end 的“ rem ”。去掉“ rem ”之后，后续的语句将被跳过。因此 MAKE 部分的“ f:\AVRGCC\bin\ ”可加可不加。

编译和链接应用程序

首先在www.avrfreaks.net上下载测试程序集 gcctest.zip，然后安装。

1. 将 GCCTEST\INCLUDE 下的 MAKE1、MAKE2 拷贝到$(AVR)\ INCLUDE
2. 将工作目录的 MAKEFILE (每个工程都要有一个此文件，且可由自己进行修改以适合自己的应用。如果要利用原有文件，则注意只能有一个 C 文件)中的 MCU、TRG、SRC、ASRC、INC、LIB 等项填入合适的内容
3. 在工作目录运行位于$(AVR)\BIN 下的 MAKE.EXE（注意：由于系统可能存在其他应用程序的 MAKE，因此可能还需要加路径。也可以将其改名。）
4. 从 MAKE1、MAKE2 和 MAKEFILE 可以看出，用户可以修改诸如输出文件名等多种选项。

在 C 代码中嵌入汇编指令

一．GCC 的 ASM 声明

首先看一个从 PORTD 读入数据的例子：

asm(“in %0, %1” : “=r”(value) : “I”(PORTD) :);

由上可以看出嵌入汇编的 4 个部分：

1．汇编指令本身，以字符串“in %0, %1”表示；

2．由逗号分隔的输出操作数，本例为“=r”(value)

3．由逗号分隔的输入操作数，本例为“I”(PORTD)

4．Clobber 寄存器

嵌入汇编的通用格式为：

asm(code : output operand list : input operand list : clobber list);

例子中%0 表示第一个操作数，%1 表示第二个操作数。即：

%0 → “=r”(value)

%1 → “I”(PORTD)

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，则优化编译时会将这些语句删除。为了防止这种情况的发生，需要加入 volatile 属性：

asm volatile (“in %0, %1” : “=r”(value) : “I”(PORTD) :);

嵌入汇编的的 Clobber 寄存器部分可以忽略，而其他部分不能忽略，但可以为空。如下例：

asm volatile(“cli” : :);

二．汇编代码

用户可以在 C 代码里嵌入任意的汇编指令，就如同在汇编器里写程序一样。

AVR-GCC 提供了一些特殊的寄存器名称：

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（0x3D）
__tmp_reg__	r0
__zero_reg__	r1。对于 C 代码而言其值永远为 0

三．输入/输出操作数

约束符号	适用于	范围
a		r16~r23
b	指针	Y，Z
d		r16~r31
e	指针	X，Y，Z
G	浮点常数	0.0
I	6 比特正常数	0~63
J	6 比特负常数	-63~0
l		r0~r15
M	8 比特正常数	0~255
N	整数常数	-1
O	整数常数	8，16，24
P	整数常数	1
r		r0~r31
t		R0
W	寄存器对	r24，r26，r28，r30
X	指针 X	r27:r26
Y	指针 Y	r29:r28
Z	指针 Z	r31:r30

要注意的是，在使用这些约束符号时要防止选择错误。例如，用户选择了”r”约束符号，而汇编语句则使用了”ori”。编译器可以在 r0~r31 之间任意选择寄存器。若选择了 r2~r15，则会由于不适用 ori 而出现编译错误。此时正确的约束符应该是”d”。

约束符号还可以有前置修饰符，如下表所示。

修饰符	指定
=	只写操作数

+	读-写操作数（嵌入汇编不支持）
&	寄存器只能用做输出

输出操作数必须为只写操作数，C 表达式结果必须为 l（r0~r15）。编译器不检查汇编指令中的变量类型是否合适。

输入操作数为只读。如果输入/输出使用同一个寄存器怎么办呢？此时可以在输入操作数的约束字符里使用一个一位数字来达到这个目的。这个数字告诉编译器使用与第 n 个（从 0 开始计数）操作数相同的寄存器。例如：

```
asm volatile(“SWAP %0” : “=r”(value) : “0”(value));
```

这条语句的目的是交换变量 value 的高低 4 位。约束符号“ 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。要注意的是，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。在某些情况下会引发严重的问题。如果用户需要区分输入/输出寄存器，则必须为输出操作数增加修饰符”&”。如下例所示。

```
asm volatile(“in %0, %1；
               out %1, %2”
              : “=&r”(input)
              : “I”(port), “r”(output)
             );
```

此例的目的是读入端口数据，然后给端口写入另一个数据。若编译器不幸使用了同一个寄存器作为参数 input 和 output 存储位置，则第一条指令执行后 output 的内容就被破坏了。而用了修饰符”&”之后，这个问题得以解决。

下面为一个高 16 位与低 16 位交换的 32 位数据操作的例子：

```
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)
)
```

“0”代表第一个操作数，“ A ”，“ B ”，“ C ”，“ D ” 表示：

31……24	23……16	15……8	7……0
D	C	B	A

四．Clobber

如前所示，asm 语句的最后一部分为 clobber。如果用户在汇编代码里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。下面为一个中断无关的加一操作例子。

```
asm volatile(“cli;
        ld r24, %a0;
        inc r24;
        st %a0, r24;
        sei”
        :
        :”z”(ptr)
        :”r24”
        )
```

编译结果为：

```
CLI
LD R24， Z
INC R24
ST Z， R24
SEI
```

当然，用户也可以用__tmp_reg__来取代 r24。此时就没有 clobber 寄存器了。

下面为考虑更详细的例子：

```
c_func
{
    uint_t s;
    asm volatile(“in %0, __SREG__;
                    cli;
                    ld, __tmp_reg__, %a1;
                    inc __tmp_reg__;
                    st %a1, __tmp_reg__;
                    out __SREG__, %0”
                    : “=r”(t)
                    :”z”(ptr)
                );
}
```

现在看起来好象没问题了。其实不尽然。由于优化的原因，编译器不会更新 C 代码里其他使用这个数值的寄存器。出于同样的优化原因，上述代码的输入寄存器可能保持的不是当前最新的数值。用户可以加入特殊的”memory” clobber 来强

迫编译器及时更新所有的变量。

更好的方法是将一个指针声明为 volatile，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。

API

嵌入式编程的代码可以简单地分为两部分，一是与硬件无关的算法部分，对其编程与普通C编程没有区别；二是与硬件相关的寄存器/端口操作部分。不同的MCU实现方法各有不同。在 AVR-GCC 里则通过一系列的 API 来解决。当然，用户也可以定义自己的 API。在此简单地介绍目前 AVR-GCC 里定义的 API，以及 AVR-GCC 的工作过程。

一．应用程序启动过程(Start Up)

标准库文件包含一个启动模块(Start Up Module)，用于为真正执行用户程序做环境设置。

启动模块完成的任务如下：

1. 提供缺省向量表
2. 提供缺省中断程序入口
3. 初始化全局变量
4. 初始化看门狗
5. 初始化寄存器 MCUCR
6. 初始化数据段
7. 将数据段.bss 的内容清零
8. 跳转到 main()。（不用调用方式，因为 main()不用返回）

启动模块包含缺省中断向量表，其内容为预先定义好的函数名称。这些函数名称可以由程序员重载。中断向量表的第一个内容为复位向量，执行结果是将程序跳转到_init_。在启动模块里，_init_表示的地址与_real_init_指向的地址相同。如果要加入客户代码，则需要在程序里定义一个_init_函数。在此函数的末尾跳转到

_real_init_。具体实现如下：

```
void _real_init_(void);
void _init_(void) __attribute__((naked));
void _init_(void)
{
    // 用户代码

    // 最后的代码必须为：
    asm ("rjmp _real_init_");
}
```

在_real_init_部分，系统将设置看门狗和 MCUCR 寄存器。启动模块并没有真正取用相应寄存器的设置数值（以符号_init_wdctr_，_init_mcucr_，_init_emcucr_表示），而是通过地址来取得其值。因而用户可以通过链接器的--defsym 选项来设置这些符号的地址。如果用户没有定义，则启动模块将使用缺省值。

接下来系统将从程序存储器里把具有初值的全局变量加载到数据存储器 SRAM。然后是将数据段.bbs 清零。此数据段包含所有没有的初值的非 AUTO 变量。

最后，系统跳转到 main()函数，用户代码开始执行。系统对此特殊函数加入一些特殊的处理。进入此函数后，堆栈指向 SRAM 的末尾。

二．存储器 API

AVR 具有三种存储器：FLASH，SRAM 和 EEPROM。AVR-GCC 将程序代码放在 FLASH，数据放在 SRAM。

I．程序存储器

如果要将数据（如常量，字符串，等等）放在 FLASH 里，用户需要指明数据类型__attribute__((progmem))。为了方便使用，AVR-GCC 定义了一些更直观的符号，如下表所示。

类型	定义
prog_void	void __attribute__((progmem))
prog_char	char __attribute__((progmem))
prog_int	int __attribute__((progmem))
prog_long	long __attribute__((progmem))
prog_long_long	long long __attribute__((progmem))
PGM_P	prog_char const*
PGM_VOID_P	prog_void const*

提供的库函数有：

1．__elpm_inline

用法：uint8_t __elpm_inline(uint32_t addr);

说明：执行 ELPM 指令从 FLASH 里取数。参数为 32 位地址，返回一个 8 位数据。

2．__lpm_inline

用法：uint8_t __elpm_inline(uint16_t addr);

说明：执行 LPM 指令从 FLASH 里取数。参数为 16 位地址，返回一个 8 位数据。

3．memcpy_P

用法：void* memcpy_P(void* dst, PGM_VOID_P src, size_t n);

说明：memcpy 的特殊版本。完成从 FLASH 取 n 个字节的任务。

4．PRG_RDB

用法：uint8_t PGR_RDB(uint16_t addr);

说明：此函数简单地调用__lpm_inline

5．PSTR

用法：PSTR（s）；

说明：参数为字符串。功能是将其放在 FLASH 里并返回地址。

6．strcmp_P

用法：int strcmp(char const*, PGM_P);

说明：功能与 strcmp()类似。第二个参数指向程序存储器内的字符串。

7．strcpy_P

用法：char* strcpy_P(char*, PGM_P);

说明：功能与 strcpy()类似。第二个参数指向程序存储器内的字符串。

8．strlen_P

用法：size_t strlen_P(PGM_P);

说明：功能与 strlen()类似。第二个参数指向程序存储器内的字符串。

9．strncmp_P

用法：size_t strncmp_P(char const*, PGM_P, size_t);

说明：功能与 strncmp()类似。第二个参数指向程序存储器内的字符串。

10．strncpy_P

用法：size_t strncpy_P(char*, PGM_P, size_t);

说明：功能与 strncpy()类似。第二个参数指向程序存储器内的字符串。

II．EEPROM

AVR 内部有 EEPROM，但地址空间与 SRAM 的不相同。在访问时必须通过 I/O 寄存器来进行。EEPROM API 封装了这些功能，为用户提供了高级接口。使用时要包含 eeprom.h。在程序里定义 EEPROM 数据的例子如下：

static uint8_t variable_x __attribute__((section(".eeprom"))) = 0;

不同的 AVR 器件具有不同数目的 EEPROM。链接器将针对不同的器件分配存储器空间。

1．eeprom_is_ready

用法：int eeprom_is_ready(void);

说明：此函数用于指示是否可以访问 EEPROM。如果 EEPROM 正在执行写操作，则在 4ms 内无法访问。此函数查询相应的状态位来指示现在是否可以访问 EEPROM。

2．eeprom_rb

用法：uint8_t eeprom_rb(uint16_t addr);

说明：从 EEPROM 里读出一个字节的内容。参数 addr 用于指示要读出的地址。_EEGET(addr)调用此函数。

3．Eeprom_read_block

用法：void eeprom_read_block(void* buf, uint16_t addr, size_t n);

说明：读出一块 EEROM 的内容。参数 addr 为起始地址，n 表明要读取的字节数。数据被读到 SRAM 的 buf 里。

4．eeprom_rw

用法：unint16_t eeprom_rw(uint16_t addr);

说明：从 EEPROM 里读出一个 16 位的数据。低字节为低 8 位，高字节为高 8 位。参数 addr 为地址。

5．eeprom_wb

用法：void eeprom_wb(uint16_t addr, uint8_t val);

说明：将 8 位数据 val 写入地址为 addr 的 EEPROM 存储器里。_EEPUT（addr，

val）调用此函数。

三．中断 API

由于 C 语言设计目标为硬件无关，因此各种编译器在处理中断时使用的方法都是编译器设计者自己的方法。

在 AVR-GCC 环境里，向量表已经预先定义，并指向具有预定义名称的中断例程。通过使用合适的名称，用户例程就可以由相应的中断所调用。如果用户没有定义自己的中断例程，则器件库的缺省例程被加入。

除了中断向量表的问题，编译器还必须处理相关寄存器保护的问题。中断 API 解决了细节问题。用户只要将中断例程定义为 INTERRUPT（）或 SIGAL（）即可。而对于用户没有定义的中断，缺省例程的处理是 reti 指令。

函数定义可参见 interrupt.h，中断信号符号表参见 sig-avr.h。

1．cli

用法：void cli(void);

说明：通过置位全局中断屏蔽位来禁止中断。其编译结果仅为一条汇编指令。

2．enable_external_int

用法：void enable_external_int(uint8_t ints);

说明：此函数访问 GIMSK 寄存器（对于 MEGA 器件则是 EIMSK 寄存器）。功能与宏 outp()一样。

3．INTERRUPT

用法：INTERRUPT（signame）

说明：定义中断源 signame 对应的中断例程。在执行时，全局屏蔽位将清零，其他中断被使能。ADC 结束中断例程的例子如下所示：

INTERRUPT（SIG_ADC）

{

}

4．sei

用法：void sei(void)；

说明：通过清零全局中断屏蔽位来使能中断。其编译结果仅为一条汇编指令。

5．SIGNAL

用法：SIGNAL（signame）

说明：定义中断源 signame 对应的中断例程。在执行时，全局屏蔽位保持置位，其他中断被禁止。ADC 结束中断例程的例子如下所示：

SIGNAL（SIG_ADC）

{

}

6．timer_enable_int

用法：void timer_enable_int(uint8_t ints);

说明：此函数操作 TIMSK 寄存器。也可以通过 outp()来设置。

四．I/O API

I．I/O 端口 API

1．BV

用法：BV(pos);

说明：将位定义转换成屏蔽码(MASK)。与头文件 io.h 里的位定义一起使用。例如，置位 WDTOE 和 WDE 可表示为“ **BV(WDTOE) | BV(WDE)** ”

2．bit_is_clear

用法：uint8_t bit_is_clear(uint8_t port, uint8_t bit);

描述：如果 port 的 bit 位清零则返回 1。此函数调用 sbic 指令，故 port 应为有效地址。

3．bit_is_set

用法：uint8_t bit_is_set(uint8_t port, uint8_t bit);

描述：如果 port 的 bit 位置位则返回 1。此函数调用 sbis 指令，故 port 应为有效地址。

4．cbi

用法：void cbi(uint8_t port, uint8_t bit);

说明：清零 port 的 bit 位。bit 的值为 0~7。如果 port 为实际 I/O 寄存器，则此函数生成一条 cbi 指令；否则，函数生成相应的优化代码。

5．inp

用法：uint8_t inp(uint8_t port);

说明：从端口 port 读入 8 比特的数值。如果 port 为常数，则函数生成一条 in 指令；若为变量，则函数用直接寻址指令。

6．__inw

用法：uint16_t __inw(uint8_t port);

说明：从 I/O 寄存器读入 16 位的数值。此函数用于读取 16 位寄存器（ADC，ICR1，OCR1，TCNT1）的值，因为读取这些寄存器需要合适的步骤。由于此函数只产生两条汇编指令，因此要在中断禁止时使用，否则有可能由于中断插入到指令之间造成读取错误。

7．__inw_atomic

用法：uint16_t __inw_atomic(uint8_t port);

说明：以原子语句方式读取 16 位 I/O 寄存器的数值。此函数首先禁止中断，读取数据之后再恢复中断状态，因此可以安全地应用在各种系统状态。

8．loop_until_bit_is_clear

用法：oidoid loop_until_bit_is_clear (uint8_t port, uint8_t bit);

说明：此函数简单地调用 sbic 指令来测试端口 port 的 bit 位是否清零。port 必须为有效端口。

9．loop_until_bit_is_set

用法：oidoid loop_until_bit_is_set (uint8_t port, uint8_t bit);

说明：此函数简单地调用 sbis 指令来测试端口 port 的 bit 位是否置位。port 必须为有效端口。

10．outp

用法：void outp(uint8_t val, uint8_t port);

说明：将 val 写入端口 port。如果 port 为常数，则函数生成一条 out 指令；若为变量，则函数用直接寻址指令。

11．__outw

用法：void __outw(uint16_t val, uint8_t port);

说明：将 16 位的 val 写入端口 port。此函数适合于操作 16 位寄存器，如 ADC，ICR1，OCR1，TCNT1。由于此函数只产生两条汇编指令，因此要在中断禁止时使用，否则有可能由于中断插入到指令之间造成读取错误。

12．__outw_atomic

用法：void __outw_atomic(uint16_t val, uint8_t port);

说明：将 16 位的 val 写入端口 port。此函数适合于操作 16 位寄存器，如 ADC，ICR1，OCR1，TCNT1。此函数首先禁止中断，读取数据之后再恢复中断状态，

因此可以安全地应用在各种系统状态。

13．sbi

用法：void sbi(uint8_t port, uint8_t bit);

说明：置位 port 的 bit 位。bit 的值为 0~7。如果 port 为实际 I/O 寄存器，则此函数生成一条 sbi 指令；否则，函数生成相应的优化代码。

五．看门狗 API

以下函数操作看门狗。宏定义参见 wdt.h。

用户可以通过起动代码初始化看门狗。WDTCR 的缺省值为 0。如果你希望将其设置为其他值，则需要在链接命令里加入相应的命令。使用的符号为__init_wdtcr__。如下为将 WDTCR 设置为 0x1f 的例子：

avr-ld –defsym __init_wdtcr__=0x1f

1．wdt_disable

用法：void wdt_disable(void);

说明：关闭看门狗。

2．wdt_enable

用法：void wdt_disable(unit8_t timeout);

说明：使能看门狗。看门狗溢出时间为 timeout。

timeout	**周期**
0	16K CLK
1	32K CLK
2	64K CLK
3	128K CLK
4	256K CLK
5	512K CLK
6	1024K CLK
7	2048K CLK

3．wdt_reset

用法：void wdt_reset(void);

说明：产生喂狗指令 wdr。

附录：AVR-GCC 配置

汇编选项

选项	描述
-mmcu=*name*	指定目标器件 *name* 可以为：at90s1200，at90s2313，at90s2323，at90s2333，at90s2343，at904433，at90s8515，at90s8535，atmega103，atmega161

寄存器使用

如果用户需要进行汇编与 C 的混合编程，必须了解寄存器的使用。

1．寄存器使用

r0 可用做暂时寄存器。如果用户汇编代码使用了 r0，且要调用 C 代码，则在调用之前必须保存 r0。C 中断例程会自动保存和恢复 r0。

r1 C 编译器假定此寄存器内容为“ 0 ”。如果用户使用了此寄存器，则在汇编代码返回之前须将其清零。C 中断例程会自动保存和恢复 r1。

r2-r17，r28，r29 C 编译器使用这些寄存器。如果用户汇编代码需要使用这些寄存器，则必须保存并恢复这些寄存器。

R18-r27，r30，r31 如果用户汇编代码不调用 C 代码则无需保存和恢复这些寄存器。如果用户要调用 C 代码，则在调用之前须保存。

2．函数调用规则

参数表：函数的参数由左至右分别分配给 r25 到 r8。每个参数占据偶数个寄存器。若参数太多以至 r25 到 r8 无法容纳，则多出来的参数将放入堆栈。

返回值：8 位返回值存放在 r24。16 位返回值存放在 r25:r24。32 位返回值存放在 r25:r24:r23:r22。64 位返回值存放在 r25:r24:r23:r22:r21:r20:r19:r18。