ImageCraft ICCAVR 的中文使用说明

翻译　詹卫前

一、ImageCraft 的 ICC AVR 编译器安装

1、运行光盘上的 SETUP.EXE 程序进行安装。

方法一：a、打开“我的电脑”

b、打开光盘驱动器所对应的盘符

c、双击光盘中文件“SETUP.EXE”的图标

d、按照屏幕提示，选定一个安装路径后进行安装

方法二：a、在“开始”菜单中选择运行项目

b、在“运行”对话框中填入“drive:\setup.exe”

注意 drive 对应你的机器中的光盘驱动器盘符

c、按“确定”键开始安装

d、其余同方法一

注意：

①按上述方法进行安装后，得到的是一个只可以使用 30 天的未注册版。对正式版用户还要进行第二步的注册，才可得到一个无时间限制的正式版。

②ICC AVR 正式版分标准版和专业版，在标准版中有一些功能限制。如：代码的压缩、工程和文件的配置检查在标准版中不可以使用。

2、对安装完成的软件进行注册

对首次安装并且使用期未超过 30 天的用户，可以这样注册

a、启动 ICC AVR 编译器的集成环境（IDE）。

b、将正式版中附带的一张名称为“Unlock Disk”的软盘插入你机器的软驱器中。

c、在 IDE 的“Help”菜单中寻找标题为“Importing a License from a Floppy Disk”的一项，并且进行单击。

d、ICCAVR 软件自动进行注册，当注册完成后会提示你注册文件已从软盘移走。当你确定并再次重新启动 ICCAVR 后，会发现软件已经完成注册。

对不是首次安装或使用时间已超过 30 天的用户，可这样注册

a、对这类用户在程序启动时已不能进入 IDE 环境，而是出现一个提示你注册的对话框，你应该选择“YES”按钮。

b、这时会出现一个注册对话框，对话框上有一个标题为“Importing a License from a Floppy Disk”的按钮。

c、将正式版中附带的一张名称为“Unlock Disk”的软盘插入你机器的软盘驱动器中，单击上一步中提到的按钮。

d、ICCAVR 软件自动进行注册，当注册完成后会提示你注册文件已从软盘移走。当你确定并再次重新启动 ICCAVR 后，会发现软件已经完成注册。

注意：

①“Unlock Disk”软盘在注册时应打开写保护，否则无法完成注册。

②完成注册后，“Unlock Disk”软盘成为一张空盘，不可以在另一台机器上进行安装和注册。

③当你需要在不同的电脑中使用 ICCAVR 或在同一台电脑中将 ICCAVR 重新安装在与原来不同的目录位置时，应该首先在“Help”菜单中选择“Transferring

Your License to a Floppy Disk”一项，将你的注册文件传送到一张软盘上，然后再按上述方法进行安装注册。

二、ICCAVR 介绍

1、ImageCraft 的 ICCAVR 介绍

ImageCraft 的 ICCAVR 是一种使用符合 ANSI 标准的 C 语言来开发微控制器（MCU）程序的一个工具，它有以下几个主要特点：

ICCAVR 是一个综合了编辑器和工程管理器的集成工作环境（IDE），其可在 WINDOWS9X/NT 下工作。

源文件全部被组织到工程之中，文件的编辑和工程的构筑也在这个环境中完成。编译错误显示在状态窗口中，并且当你用鼠标单击编译错误时，光标会自动跳转到编辑窗口中引起错误的那一行。这个工程管理器还能直接产生您希望得到的可以直接使用的 INTEL HEX 格式文件，INTEL HEX 格式文件可被大多数的编程器所支持，用于下载程序到芯片中去。

ICCAVR 是一个 32 位的程序，支持长文件名。

出于篇幅考虑，本说明书并不介绍通用的 C 语言语法知识，仅介绍使用 ICC AVR 所必须具备的知识，因此要求读者在阅读本说明书之前，应对 C 语言有了一定程度的理解。

2、ICCAVR 中的文件类型及其扩展名

文件类型是由它们的扩展名决定的，IDE 和编译器可以使用以下几种类型的文件。

输入文件：

.c 扩展名----表示是 C 语言源文件

.s 扩展名----表示是汇编语言源文件

.h 扩展名----表示是 C 语言的头文件

.prj 扩展名----表示是工程文件，这个文件保存由 IDE 所创建和修改的一个工程的有关信息。

.a 扩展名----库文件，它可以由几个库封装在一起。libcavr.a 是一个包含了标准 C 的库和 AVR 特殊程序调用的基本库。如果库被引用，链接器会将其链接到您的模块或文件中。您也可以创建或修改一个符合你需要的库。

输出文件：

.s 对应每个 C 语言源文件，由编译器在编译时产生的汇编输出文件。

.o 由汇编文件汇编产生的目标文件，多个目标文件可以链接成一个可执行文件。

.hex INTEL HEX 格式文件，其中包含了程序的机器代码。

.eep INTEL HEX 格式文件，包含了 EEPROM 的初始化数据。

.cof COFF 格式输出文件，用于在 ATMEL 的 AvrStudio 环境下进行程序调试。

.lst 列表文件，在这个文件中列举出了目标代码对应的最终地址。

.mp 内存映象文件，它包含了您程序中有关符号及其所占内存大小的信息。

.cmd NoICE 2.xx 调试命令文件。

.noi NoICE 3.xx 调试命令文件。

.dbg ImageCraft 调试命令文件。

3、附注和扩充

#pragma （编译附注）

这个编译器接受以下附注：

#pragma interrupt_handler <func1>:<vector number> <func2>:<vector> ...

这个附注必须在函数之前定义，它说明函数 func1、func2 是中断操作函数，所以编译器在中断操作函数中生成中断返回指令 reti 来代替普通返回指令 ret，并且保存和恢复函数所使用的全部寄存器；同样编译器根据中断向量号 vector number 生成中断向量地址。

#pragma ctask <func1> <func2>...

这个附注指定了函数不生成挥发寄存器来保存和恢复代码，它的典型应用是在 RTOS 实时操作系统中让 RTOS 核直接管理寄存器。

#pragma text:<name>

改变代码段名称，使其与命令行选项相适应。

#pragma data:<data>

改变数据段名称，使其与命令行选项相适应。这个附注在分配全局变量至 EEPROM 中时必须被使用，读者可参考访问 EEPROM 的例子。

#pragma abs_address:<address>

函数与全局数据不使用浮动定位（重定位），而是从<address>开始分配绝对地址。这在访问中断向量和其它硬件项目时特别有用。

#pragma end_abs_address

结束绝对定位，使目标程序使用正常浮动定位。

C++ 注释

如果你选择了编译扩充(Project->Options->Compiler)，你可以在你的源代码中使用 C ++ 的 // 类型的注释。

二进制常数

如果你选择了编译扩充(Project->Options->Compiler)，你可以使用 0b<1|0>* 来指定二进制常数，例如，0b10101 等于十进制数 21。

在线汇编

你可以使用 asm("string")函数来指定在线汇编代码，读者可参考在线汇编。

4、代码转换

IAR 或其它 ANSI C 编译系统的代码转换

IAR C 编译器作为应用于 AVR 的第一个 C 编译器，它有十分丰富的源代码。当你从 IAR 编译系统转换到 ImageCraft 编译系统时，绝大多数符合 ANSI C标准的程序代码不需要转换，IAR C 中 IO 寄存器的定义与 ICCAVR 也是相同的。

中断操作描述，ICCAVR 使用 pragma 附注描述中断操作函数，而 IAR 引入了语法扩充（interrupt 关键字），下面是一个对照：

在 ICCAVR 中：

#pragma interrupt_handler func:4 // 4 是这个中断的向量号，func 为中断处理函数名称，ICCAVR 可以使多个中断向量共用一个中断处理函数。

在 IAR 中：

interrupt [vector_name] func() // vector_name 是某一个中断向量的名称，IAR C 的中断向量地址使用中断名称来代替，以增加程序的可读性。

扩充关键字

IAR 引入 flash 关键字将项目分配进入程序存贮空间（FLASH 存贮器），ICCAVR 使用 const 关键字来达到相同的目的。

过程调用转换

在两个编译系统之间函数参数传递使用的寄存器是不同的，这仅影响手工写的汇编函数。

在线汇编、宏等，IAR 不支持在线汇编符号，而 ICCAVR 支持在线汇编。

三、ICCAVR 向导

1、起步

自你启动 IDE 后，首先从 Project 菜单系统选择 Open 命令，进入\icc\examples.avr 目录并且选择并打开“led”工程，工程管理器显示在这个工程中只有一个文件 led.c。然后从 Project 菜单中选择 Options 命令打开工程编译选项，在"Target"标号下选择目标处理器。然后从 Project 菜单中选择 Make Project 命令，IDE 将调用编译器编译这个工程文件，并且在状态窗口中显示所有的信息。

如果没有错误，在与源文件同一个目录（在这个例子中是\icc\examples.avr）中输出一个文件 led.hex。这个文件是 INTEL HEX 格式，大多数能支持 AVR MCU 的编程器和模拟器都支持这种格式，并且能下载这个程序进入你的目标系统，这样就完成了一个程序的构筑。

如果你希望用支持 COFF 调试信息的工具来测试你的程序，比如 AVR Studio，那么你需要从 Project 菜单中选择 Options 命令，在编译标签下选择 COFF 输出文件格式。对一些常用的功能，你也可使用工具条或鼠标右键弹出菜单。例如，你可以在工程窗口单击鼠标右键选择编译选项。

在工程窗口中双击文件名，IDE 将使用编辑器打开这个文件。按这个方法打开 led.c，作为试验可设置一些错误，例如从一行中删除分号“；”。现在从 Project 菜单中选择 Make Project 命令，IDE 首先自动保存已经改变的文件，并且开始编译这个文件。这时在状态窗口中会显示错误信息，单击状态窗口中错误信息行，或单击其左边的错误符号，光标将移到编辑器中错误行的下面一行上（基本上所有 C 编译器都是这样）。

开始一个新的工程

从 Project 菜单中选择 New 命令，并且浏览至你希望输出工程文件的目录，输出文件的名称取决于你的工程文件名称。例如，如果你创建一个名称为 foo.prj 的工程，那么输出文件名称为 foo.hex 或 foo.cof 等。

自从创建你自己的工程后，你可以开始写你的源代码(C 或汇编格式)，并且将这个文件加入到工程文件排列中。单击工具栏中“Build”图标，可以很容易地构筑这个工程，IDE 输出与 ATMEL 的 AVR Studio 完全兼容的 COFF 文件，你可以使用 ATMEL 的 AVR Studio 来调试你的代码。

为更容易地使用这个开发工具，你可以使用应用程序向导来生成一些使用有关硬件的初始化代码。

2、C 程序的剖析

一个 C 程序必须定义一个 main 调用函数，编译器会将你的程序与启动代码和库函数链接成一个“可执行”文件，因此你也可以在你的目标系统中执行它。启动代码的用途在启动文件中很详细地被描述了，一个 C 程序需要设定目标环境，启动代码初始化这个目标使其满足所有的要求。

通常，你的 main 例程完成一些初始化后，然后是无限循环地运行。作为例子，让我们看 \icc\examples 目录中的文件 led.c。

```
#include <io8515.h>
  /* 为使能够看清 LED 的变化图案，延时程序需要有足够的延时时间*/
void Delay()
{
  unsigned char a, b;
```

```
        for (a = 1; a; a++)
            for (b = 1; b; b++)
            ;
}

void LED_On(int i)
    {
    PORTB = ~BIT(i);  /*  低电平输出使 LED 点亮  */
    Delay();
    }

void main()4
    {
    int i;
    DDRB = 0xFF;       /*定义  B 口输出*/
    PORTB = 0xFF;      /* B 口全部为高电平，对应 LED 熄灭*/
    while (1)
            {
            /*LED 向前步进  */
            for (i = 0; i < 8; i++)
                    LED_On(i);
            /* LED 向后步进  */
            for (i = 8; i > 0; i--)
                    LED_On(i);
            /* LED 跳跃*/
            for (i = 0; i < 8; i += 2)
                    LED_On(i);
            for (i = 7; i > 0; i -= 2)
                    LED_On(i);
             }
        }
```

这个 main 例程是很简单的，在初始化一些 IO 寄存器后之后，它运行在一个无限循环中，并且在这个循环中改变 LED 的步进图案。LED 是在 LED_On 例程中被改变的，在 LED_On 例程中直接写正确的数值到 IO 端口。因为 CPU 运行很快，为能够看见图案变化 LED_On 例程调用了延时例程。因为延时的实际延时值不能被确定，这一对嵌套循环只能给出延时的近似延时时间；如果这个实际定时时间是重要的，那么这个例程应该使用硬件定时器来完成延时。

其它的例子，8515intr.c 程序很简单，但同样清楚地显示了如何用 C 写一个中断处理过程，这两个例子可以作为你的程序的起点。

四、ICCAVR 的 IDE 环境

1、编译一个单独的文件

正常建立一个输出文件的次序是，你首先应该建立一个工程文件并且定义属于这个

工程的所有文件。然而，我们有时也需要将一个文件单独地编译为目标文件或最终的输出文件。这时可以这样操作：从 IDE 菜单“File”中选择“Compile File...”命令，来执行“to Object”和“to Output”中的任意一个。当你调用这个命令时，文件应该是打开的，并且在编辑窗口中可以编辑的。

编译一个文件为目标文件（to Object），对检查语法错误和编译一个新的启动文件是很有用的。编译一个文件为输出文件（to Output），对较小的并且是一个文件的程序较为有用。注意：这里使用默认的编译选项。

2、创建一个新的工程

为创建一个新的工程，从菜单“Project”中选择“New”命令，IDE 会弹出一个对话框，在对话框中你可以指定工程的名称，这也是你的输出文件的名称。如果你使用一些已经建立的源文件，你可在菜单“Project”中选择“AddFile(s)”命令。

另外，你可以在菜单“File”中选择“New”命令来建立一个新的源文件来输入你的代码，你可以在菜单“File”中选择“Save ”或“Save As”命令来保存文件。然后你可以象上面所述调用“AddFile(s)”命令将文件加入到工程中，也可在当前编辑窗口中单击鼠标右键选择“Add to Project”将文件加入已打开的工程列表中。通常你输出源文件在工程同一个目录中，但也可不作这样要求。

工程的编译选项使用菜单中“Project”中的“Options”命令。

3、工程管理

工程管理允许你将多个文件组织进同一个工程，而且定义它们的编译选项，这个特性允许你将工程分解成许多小的模块。当你处理工程构筑时，只有一个文件被修改和重新编译。如果一个头文件作了修改，当你编译包含这个头文件的源文件时，IDE 会自动重新编译已经改变的头文件。

一个源文件可以写成 C 或汇编格式的任意一种。C 文件必须使用“.c”扩展名，汇编文件必须使用“.s”扩展名。你可以将任意文件放在工程列表中，例如你可以将一个工程文档文件放在工程管理窗口中，工程管理器在构筑工程时对源文件以外的文件不予理睬。

对目标器件不同的工程，可以在编译选项中设置有关参数。当你新建一个工程时，使用默认的编译选项，你可以将现有编译选项设置成默认选项，也可将默认编译选项装入现有工程中。默认编译选项保存在 default.prj 文件中。

为避免你的工程目录混乱，你可以指定输出文件和中间文件到一个指定的目录，通常这个目录是你的工程目录的一个子目录。

4、编辑窗口

编辑窗口是你与 IDE 交流信息的主要区域，在这个窗口中你可以修改相应的文件。当编译存在错误时，用鼠标单击有关错误信息时，编辑器会自动将光标定位在错误行的位置。注意：对 C 源文件中缺少分号“;”的错误，编辑器定位于其下面一行。

5、应用构筑向导

应用构筑向导是用于创建外围设备初始化代码的一个图形界面。你可以单击工具条中的“Wizard”按钮或菜单“Tools”中的“ApplicationBuilder”命令来调用它。

应用构筑向导使用编译选项中指定的目标 MCU 来产生相应的选项和代码。

应用构筑向导显示目标 MCU 的每一个外围设备子系统，它的使用是很显而易见的。在这里你可以设置 MCU 的所具有的中断、内存、定时器、IO 端口、UART、SPI 和模拟量比较器等外围设备，并产生相应的代码。如果你需要的话，还可产生 main( )函数。

6、状态窗口

状态窗口显示 IDE 的状态信息。

7、终端仿真

IDE 有一个内置的终端仿真器，注意它不包含任意一个 ISP（在系统编程）功能，但它可以作为一个简单的终端，或许可以显示你的目标装置的调试信息，也可下载一个 ASCⅡ码文件。

从 6.20 版本开始，IDE 加入了对 ISP 的支持。

五、菜单解释

1、弹出菜单

在 ICCAVR 环境中单击右键，那么 ICCAVR 会根据实际情况弹出相应的工具菜单。

2、File Menu 文件菜单

New ：新建一个文件，你可在编辑窗口是输入文字或代码。

Reopen：重新打开历史文件，有关历史文件显示的右边的子菜单中。

Open：打开一个已以经存在的文件用于编辑，文件用浏览窗口选择。

Reload ..form Disk：放弃全部的修改，从磁盘中重新装载当前文件。

Reload ....from Back UP：从最后一次的备份文件中装载当前文件。

Save：保存当前文件，如果环境设置中设置了保存备份文件，则将原文件以 <file>.~<ext>形式保存。

Save as：将当前文件用另外一个名称来保存。

Close：关闭当前文件，如果文件有过修改，系统会进行提示。

Compile File ..to Object ：编译当前文件成目标文件。注意目标文件不可以直接用于对芯片编程或用于调试，其主要用于：语法检查、为创建新的启动文件或库产生目标文件。

Compile File ... to Output：编译当前文件成输出文件，其产生的输出文件可用于编程器和调试器。

Save All：保存所有打开的文件。

Closs All：关闭当前打开的所有文件，同样它会提示你保存已经修改的文件。

Print：打印当前文件。

Exit：退出 ICCAVR 的 IDE 环境。

3、Edit Menu 编辑菜单

Undo：撤消最后一次的修改。

Redo：撤消最后一次的 Undo。

Cut：剪切选择的内容到剪帖板。

Copy：拷贝选择的内容到剪帖板。

Paste：将剪帖板内容粘帖在当前光标的位置。

Delete：删除选择的内容。

Select All：选择全部内容。

Block Indent：对选择的整块内容右移

Block Outdent ：对选择的整块内容左移

4、Search menu 寻找菜单

Find ..在编辑窗口中寻找一个文本。

它有以下选项：

Match Case – 区分大小写

Whole Word – 全字匹配

Up/Down –往上或往下

Find in Files... –在当前打开的文件中或在当前工程的所有文件中或当前目录中的文件中寻找一段文本。它有以下选项:

Case Sensitive –大小写敏感

Whole Word - 全字匹配

Regular Expression – 寻找规则的表达式

Replace... – 在编辑器中替换文本。

Find Again – 寻找下一个。

Goto Line Number – 转到指定行号。

Add Bookmark – 添加书签。

Delete Bookmark – 删除书签。

Next Bookmark – 跳转到下一个签。

Goto Bookmark – 跳转到指定的书签。

5、View Menu 视图菜单

Status Window – 如果选中，显示状态窗口。

Project Makefile – 以只读方式打开 makefile 文件

Output Listing File – 以只读方式打开列表文件

6、Project Menu 工程菜单

New... – 创建一个新的工程文件。

Open –打开一个已经存在的工程文件。

Open All Files... – 打开工程的全部源文件。

Close All Files – 关闭全部打开的文件。

Reopen... – 重新打开一个最近打开过的工程文件。

Make Project –解释和编译已经修改的文件为输出文件。

Rebuild All – 重新构筑全部文件，注意在版本升级后对原有工程最好全部重新构筑。

Add File(s) – 添加一个文件到工程中，这个文件可以是非源文件。

Remove Selected Files – 从工程中删除选择的文件。

Option... – 打开工程编译选项对话框。

Close – 关闭工程

Save As... – 将工程换一个名称存盘。

7、Tools Menu 工具菜单

Environment Options – 打开环境和终端仿真器选项对话框

Editor and Print Options –打开编辑和打印选项对话框

AVR Calc – 打开 AVR 计算器，可以计算 UART 的波特率、定时器的定时常数

Application Builder –打开应用向导程序，生成硬件的初始化代码。

Configure Tools – 允许你添加自己的内容到工具菜单

Run –以命令行方式运行一个程序

8、Compiler Options 编译选项

编译选项总共有三个页面：Paths、Compiler 和 Target

在 Paths 页面中有：

Include Path(s) –你可以指定包含文件的路径

Assembler Include Path(s) – 指定汇编包含文件的路径

Library Path – 链接器所使用的库文件的路径。

Output Directory –输出文件的目录

Compiler 页面有：

Strict ANSI C Checking – 严格的 ANSI C 语法检查

Accept Extensions – 接受 C++类型语法扩充

Macro Define(s) – 定义宏，宏之间用空格或分号分开，宏定义形式如下：

name[:value] 或 name[=value]

例如：

DEBUG:1;PRINT=printf

等价于

```
#define DEBUG 1
#define PRINT printf
```

Macro Undefine(s) –同上，但意义相反。

Output File Format – 输出文件格式 COFF/HEX、Intel HEX 或 COFF。

Optimizations – 代码优化

Default – 基本优化，象寄存器分配、共用相同的子例程等

Maximize Code Size Reduction – 只有专业版才可使用，它调用了代码压缩优化，去除了无用的碎片代码。

Target 页面有：

Device Configuration – 选择目标 MCU

Memory Sizes – 要选择"Custom" 时指定内存大小，包括 ROM、SRAM 和 EEPROM

Text Address – 通常代码地址开始于中断向量区域后面。

Data Address – 指定数据起始地址，通常为 0x60。

Use Long JMP/CALL – 指定 MCU 是否支持长跳转和长调用。

Enhanced Core – 指定硬件支持增强核指令。

IO Registers Offset Internal SRAM – 指定内部 SRAM 的偏移量。 例如, 8515 的 SRAM 起始于 0x60, 在 IO 寄存器空间后面延伸了 512 字节。而 Mega603 , IO 寄存器覆盖在 SRAM 空间中，因此 SRAM 也是从 0 开始的。

Internal 对 External SRAM – 指定你的目标系统的数据 SRAM 类型

PRINTF Version – 选择 PRINTF 的版本

Small 或 Basic: 只有 %c, %d, %x, %X, %u, and %s 格式支持

Long: 支持 %ld, %lu, %lx, %lX

Floating point: %f 支持，注意这个选项需要很大的内存。

AVR Studio Simulator IO – 如果选中， AVR Studio 的终端模拟仿真被支持。

Additional Libraries –使用标准库以外的附加库

Strings in FLASH – 字符串只保存在 FLASH 存贮器中。

Return Stack Size – 指定编译器使用的硬件堆栈的大小，编译器使用的软件堆栈的大小不需地指定。

Non Default Startup –允许你指定一个启动文件的位置，系统默认的启动文件在 Paths 页中指定，这样 IDE 可以使用多个启动文件。

Unused ROM Fill Pattern – 用一串十六进制数填充空余的 ROM 空间。

六、C 库函数与启动文件

1、启动文件

这个链接器会自动将启动文件连接到您的程序之前，并将标准库 libcavr.a 与你的程序

相连接。启动文件根据目标 MCU 的不同在 crtavr.o 和 crtatmega.o 中间任意选择一个。启动文件定义了一个全局符号__start，它也是您的程序的起点。启动文件的功能有：

1. 初始化硬件和软件堆栈指针。
2. 从 idata 区拷贝初始化数据到直接寻址数据区 data 区。
3. 将 bss 区全部初始化为零。
4. 调用用户主例程 main 函数。
5. 定义一个退出点，如果你的主函数 main( )一旦退出，它将进入这个退出点进行无限循环。

启动文件也定义了复位向量，你不需要修改启动文件来使用别的中断，具体可参考中断操作部分。

为修改和使用新的启动文件：

cd \icc\libsrc.avr ；进入你安装的编译器路径

<edit crtavr.s> ；编辑修改 crtavr.s 文件

<open crtavr.s using the IDE> ；用 IDE 打开 crtavr.s 文件

<Choose "Compile File To->Object"> ;选择编译到目标文件，创建一个新的 crtavr.o

copy crtavr.o ..\lib ；拷贝到库目录

如果您使用的目标 MCU 是 Mega，你应该用"crtatmega" 代替"crtavr"，注意 Mega 的每个中断入口地址使用两个字（word），而非 Mega 芯片每一个中断入口地址使用一个字（word）.

你也可以有多个启动文件，你可以在工程选项对话框中很方便地直接指定一个启动文件加入您的工程中。注意，您必须指定启动文件的绝对路径或启动文件必须位于工程选项库路径所指定的目录中。

2、常用库介绍

1）、库源代码

这个库源代码（缺省路径为 c:\icc\libsrc.avr\libsrc.zip）是一个密码保护的 ZIP 压缩文件，你可以从互连网上任意下载一个 UNZIP 程序进行解压缩。当本软件被开锁后，密码显示在"About"对话框中。例如：

unzip -s libsrc.zip
; unzip 提示输入密码

2）、AVR 特殊函数----- ICCAVR 有许多访问 UART、EEPROM 和 SPI 的函数，堆栈检查函数对检测堆栈是否溢出很有用。另外我们的互连网上有一个页专门存放用户写的源代码。

3）、io*.h (io2313.h, io8515.h, iom603.h, ... 等.)

这些文件中是从 ATMEL 官方公开的定义 IO 寄存器的源文件经过修改得到的，应该用这些文件来代替老的 avr.h 文件。

```
PORTB = 1;
uc = PORTA;
```

4）、macros.h

这个文件包含了许多有用的宏和定义。

5）、其它头文件

下列标准的 C 头文件是被支持的。如果你的程序使用了头文件所列出的函数，那么包含头文件是一个好习惯，在使用浮点数和长整型数的程序中必须用 #include 预编译指令包含这些包含了这些函数原形的头文件。读者可参考返回非整型值的函数。

assert.h - assert(), 声明宏
ctype.h – 字符类型函数
float.h – 浮点数原形
limits.h – 数据类型的大小和范围
math.h – 浮点运算函数
stdarg.h – 变量参数表.
stddef.h – 标准定义
stdio.h – 标准输入输出（IO）函数
stdlib.h – 包含内存分配函数的标准库
string.h – 字符串处理函数

3、字符类型库

下列函数按照输入的 ACS II 字符集字符分类，使用这些函数之前应当用"#include <ctype.h>"包含。

int isalnum(int c)
如果 c 是数字或字母返回非零数值，否则返回零

int isalpha(int c)
如果 c 是字母返回非零数值，否则返回零

int iscntrl(int c)
如果 c 是控制字符（如 FF, BELL, LF ..等）返回非零数值，否则返回零

int isdigit(int c)
如果 c 是数字返回非零数值，否则返回零

int isgraph(int c)
如果 c 是一个可打印字符而非空格返回非零数值，否则返回零

int islower(int c)
如果 c 是小写字母返回非零数值，否则返回零

int isprint(int c)
如果 c 是一个可打印字符返回非零数值，否则返回零

int ispunct(int c)
如果 c 是一个可打印字符而不是空格、数字或字母返回非零数值，否则返回零

int isspace(int c)
如果 c 是一个空格字符返回非零数值，包括空格 CR, FF, HT, NL, 和 VT，否则返回零

int isupper(int c)
如果 c 是大写字母返回非零数值，否则返回零

int isxdigit(int c)
如果 c 是十六进制数字返回非零数值，否则返回零

int tolower(int c)
如果 c 是大写字母则返回 c 对应的小写字母，其它类型仍然返回 c

int toupper(int c)
如果 c 是小写字母则返回 c 对应的大写字母，其它类型仍然返回 c

4、浮点运算库

下列函数支持浮点数运算，使用这些函数之前必须用"#include <math.h>"包含

float asin(float x)

以弧度形式返回 x 的反正弦值

float acos(float x)

以弧度形式返回 x 的反余弦值

float atan(float x)

以弧度形式返回 x 的反正切值

float atan2(float x, float y)

返回 y/x 的反正切，其范围在-π~+π 之间

float ceil(float x)

返回对应 x 的一个整型数，小数部分四舍五入。

float cos(float x)

返回以弧度形式表示的 x 的余弦值

float cosh(float x)

返回 x 的双曲余弦函数值

float exp(float x)

返回以 e 为底的 x 的幂，即 e^x

float exp10(float x)

返回以 10 为底的幂，即 10^x

float fabs(float x)

返回 x 的绝对值

float floor(float x)

返回不大于 x 的最大整数

float fmod(float x, float y)

返回 x/y 的余数

float frexp(float x, int *pexp)

把浮点数 x 分解成数字部分 y（尾数）和以 2 为底的指数 n 两个部分，即 $x=y\times 2^n$**，y 的范围为** $0.5\leq y<1$**，y 值被函数返回，而 n 值存放到 pexp 指向的变量中。**

float fround(float x)

返回最接近 x 的整型数

float ldexp(float x, int exp)

返回 $x\times 2^{exp}$

float log(float x)

返回 x 的自然对数

float log10(float x)

返回以 10 为底的 x 的对数

float modf(float x, float *pint)

把浮点数分解成整数部分和小数部分，整数部分存放到 pint 指向的变量，小数部分应当大于或等于 0 而小于 1，并且作为函数返回值返回。

float pow(float x, float y)

返回 x^y **值**

float sqrt(float x)

返回 x 的平方根

float sin(float x)

返回以弧度形式表示的 x 的正弦值

float sinh(float x)

返回 x 的双曲正弦函数值

float tan(float x)

返回以弧度形式表示的 x 的正切值

float tanh(float x)

返回 x 的双曲正切函数值

5、标准输入输出库

标准的文件输入输出是不能真正植入微控制器（MCU）的，标准 stdio.h 的许多内容不可以使用，不过有一些 IO 函数是被支持的，同样使用之前应用"#include <stdio.h>"预处理，并且需要初始化输出端口。最低层的 IO 程序是单字符的输入(getchar)和输出(putchar)程序，如果你针对不同的装置使用高层的 IO 函数，例如用 printf 输出 LCD，你需要全部重新定义最底层的函数。

为在 ATMEL 的 AVR Studio 模拟器（终端 IO 窗口）使用标准 IO 函数，应当在编译选项中选中相应的单选钮。

注意：作为缺省，单字符输出函数 putchar 是输出到 UART 装置没有修改，无论如何为使输出能如期望的那样出现在程序终端窗口中，'\n' 字符必须被映射为成对的回车和换行（CR/LF）。

int getchar()

使用查寻方式从 UART 返回一个字符

int printf(char *fmt, ..)

按照格式说明符输出格式化文本 frm 字符串，格式说明符是标准格式的一个子集。

%d--输出有符号十进制整数

%o --输出无符号八进制整数

%x - 输出无符号十六进制整数

%X –除了大写字母使用'A'-'F'外，同 %x

%u - 输出无符号十进制整数

%s – 输出一个以 C 中空字符 NULL 结束的字符串

%c – 以 ASCII 字符形式输出，只输出一个字符

%f – 以小数形式输出浮点数

%S – 输出在 FLASH 存贮器中的字符串常量

printf 支持三个版本，取决于你的特别需要和代码的大小（越高的要求，代码越大）：

基本形：只有 %c, %d, %x, %u, 和 %s 格式说明符是承认的

长整形：针对长整形数的修改 %ld, %lu, %lx 被支持，以适用于精度要求较高的领域

浮点形：全部格式包括%f 被支持

你使用编译选项对话框来选择版本，代码大小的增加是值得关注的。

int putchar(int c)

输出单个字符。这个库程序使用了 UART 以查寻方式输出单个字符，注意输出'\n'字符至程序终端窗口。

int puts(char *s)

输出以 NL 结尾的字符串。

int sprintf(char *buf, char *fmt)

按照格式说明符输出格式化文本 frm 字符串到一个缓冲区，格式说明符同 printf( ) 。

"const char *" 支持功能

cprintf 和 csprintf 是将 FLASH 中的格式字符串分别以 prinf 和 sprinf 形式输出。

6、标准库和内存分配函数

标准库头文件<stdlib.h>**定义了宏** NULL **和** RAND_MAX **和新定义的类型** size_t，**并且描述了下列函数。注意在你调用任意内存分配程序（比如.**. calloc、malloc **和** realloc)**之前，必须调用**_NewHeap **来初始化堆** heap。

int abs(int i)

返回 i **的绝对值**

int atoi(char *s)

转换字符串 s **为整型数并返回它，字符串** s **起始必须是整型数形式字符，否则返回** 0

double atof(const char *s)

转换转换字符串 s **为双精度浮点数并返回它，字符串** s **起始必须是浮点数形式字符串。**

long atol(char *s)

转换字符串 s **为长整型数并返回它,字符串** s **起始必须是长整型数形式字符，否则返回** 0

void *calloc(size_t nelem, size_t size)

分配"nelem"**个数据项的内存连续空间，每个数据项的大小为** size **字节并且初始化为** 0。**如果分配成功返回分配内存单元的首地址，否则返回** 0。

void exit(status)

终止程序运行，典型的是无限循环，它是担任用户 main **函数的返回点。**

void free(void *ptr)

释放 ptr **所指向的内存区**

void *malloc(size_t size)

分配 size **字节的存贮区，如果分配成功则返回内存区地址，如内存不够分配则返回** 0。

void _NewHeap(void *start, void *end)

初始化内存分配程序的堆。一个典型的调用是将符号_bss_end+1 **的地址用作**"start"**值，符号**_bss_end **定义为编译器用来存放全局变量和字符串的数据内存的结束，加** 1 **的目的是堆栈检查函数使用**_bss_end **字节存贮为标志字节，这个结束值不能被放入堆栈中。**

```
extern char _bss_end;
_NewHeap(&_bss_end+1, &_bss_end + 201); // 初始化 200 字节大小的堆
```

int rand(void)

返回一个在 0 **和** RAND_MAX **之间的随机数。**

void *realloc(void *ptr, size_t size)

重新分配 ptr **所指向内存区的大小为** size **字节，**size **可比原来大或小，返回指向该内存区的地址指针。**

void srand(unsigned seed)

初始化随后调用的随机数发生器的种子数。

long strtol(char *s, char **endptr, int base)

按照"base."**的格式转换**"s"**中起始字符为长整型数。如果**"endptr"**不为空，*** endptr **将设定**"s"**中转换结束的位置。**

unsigned long strtoul(char *s, char **endptr, int base)

除了返回类型为无符号长整型数外，其余同"strtol"

7、字符串函数

用"#include <string.h>"**预处理后，编译器支持下列函数。**<string.h>**定义了** NULL、**类型** size_t **和下列字符串及字符阵列函数。**

void *memchr(void *s, int c, size_t n)

在字符串 s 中搜索 n 个字节长度寻找与 c 相同的字符，如果成功返回匹配字符的地址指针，否则返回 NULL。

int memcmp(void *s1, void *s2, size_t n)

对字符串 s1 和 s2 的前 n 个字符进行比较，如果相同则返回 0，如果 s1 中字符大于 s2 中字符，则返回 1，如果 s1 中字符小于 s2 中字符，则返回-1。

void *memcpy(void *s1, void *s2, size_t n)

拷贝 s2 中 n 个字符至 s1，但拷贝区不可以重迭。

void *memmove(void *s1, void *s2, size_t n)

拷贝 s2 中 n 个字符至 s1，返回 s1，其与 memcpy 基本相同，但拷贝区可以重迭。

void *memset(void *s, int c, size_t n)

在 s 中填充 n 个字节的 c，它返回 s。

char *strcat(char *s1, char *s2)

拷贝 s2 到 s1 的结尾，返回 s1。

char *strchr(char *s, int c)

在 s1 中搜索第一个出现的 c，包括结束 NULL 字符。如果成功返回指向匹配字符的指针，如果没有匹配字符找到，返回空指针。

int strcmp(char *s1, char *s2)

比较两个字符串，如果相同返回 0，如果 s1>s2 则返回 1，如果 s1<s2 则返回-1。

char *strcpy(char *s1, char *s2)

拷贝字符串 s2 至字符串 s1，返回 s1。

size_t strcspn(char *s1, char *s2)

在字符串 s1 搜索与字符串 s2 匹配的第一个字符，包括结束 NULL 字符，其返回 s1 中找到的匹配字符的索引。

size_t strlen(char *s)

返回字符串 s 的长度，不包括结束 NULL 字符。

char *strncat(char *s1, char *s2, size_t n)

拷贝字符串 s2（不含结束 NULL 字符）中 n 个字符到 s1，如果 s2 长度比 n 小，则只拷贝 s2，返回 s1。

int strncmp(char *s1, char *s2, size_t n)

基本和 strcmp 函数相同，但其只比较前 n 个字符。

char *strncpy(char *s1, char *s2, size_t n)

基本和 strcpy 函数相同，但其只拷贝前 n 个字符。

char *strpbrk(char *s1, char *s2)

基本和 strcspn 函数相同，但它返回的是在 s1 匹配字符的地址指针，否则返回 NULL 指针。

char *strrchr(char *s, int c)

在字符串 s 中搜索最后出现的 c，并返回它的指针。否则返回 NULL。.

size_t strspn(char *s1, char *s2)

在字符串 s1 搜索与字符串 s2 不匹配的第一个字符，包括结束 NULL 字符，其返回 s1 中找到的第一个不匹配字符的索引。

char *strstr(char *s1, char *s2)

在字符串 s1 中找到与 s2 匹配的子字符串，如果成功它返回 s1 中匹配子字符串的地址指针，否则返回 NULL。

"const char *" 支持函数

这些函数除了它的操作对象是在 FLASH **中常数字符串外，其余同** c **中的函数。**

```
size_t cstrlen(const char *s)
char *cstrcpy(char *dst, const char *src);
int cstrcmp(const char *s1, char *s2);
```

8、**变量参数函数**

<stdarg.h>**提供再入式函数的变量参数处理，它定义了不确定的类型** va_list **和三个宏。**

va_start(va_list foo, <last-arg>)

初始化变量 foo

va_arg(va_list foo, <promoted type>)

访问下一个参数，分派指定的类型。注意那个类型必须是高级类型，如 int、 long **或** double，**小的整型类型如**"char"**不能被支持。**

va_end(va_list foo)

结束变量参数处理

例如，printf()**可以使用** vfprintf()**来实现**

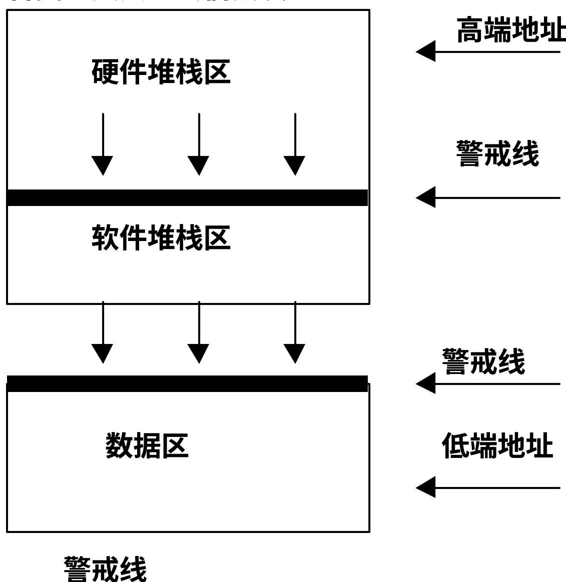
```
#include <stdarg.h>
int printf(char *fmt, ...)
    {
    va_list ap;

va_start(ap, fmt);
    vfprintf(fmt, ap);
    va_end(ap);
    }
```

9、**堆栈检查函数**

有几个库函数是用于检查堆栈是否溢出，内存图如下。如果硬件堆栈增长到软件堆栈中，那么软件堆栈的内容将会被改变，也就是说局部变量和别的堆栈项目被改变。硬件堆栈是用作函数的返回地址，如果你的函数调用层次太深偶然会发生这种情况。

同样地，软件堆栈溢出进数据区域将会改变全局变量或其它静态分配的项目（如果你使用动态分配内存，还会改变堆项目）。这种情况在你定义了太多的局部变量或一个局部集合变量太大也会偶然发生。



警戒线

启动代码写了一个正确的关于数据区的地址字节和一个类似的正确的关于软件堆栈的地址字节作为警戒线。[注意：如果你使用了你自己的启动文件，而其又是以 6.20 版本之前的启动文件为基础的，你将需要额外改造为新的启动文件。]

注意：如果你使用动态分配内存，你必须跳过警戒线字节_bss_end 来分配你的堆。参考内存分配函数。

堆栈检查

你调用_StackCheck(void)函数来检查堆栈溢出，如果警戒线字节仍然保持正确的值那么函数检查通过，如果堆栈溢出，那么警戒线字节将可能被破坏。

注意当你的程序堆栈溢出的时候，你的程序将可能运行不正常或偶然崩溃。当_StackCheck 检查错误条件时，它调用了带一个参数的函数_StackOverflowed(char c)。如果参数是 1，那么硬件堆栈有过溢出；如果参数是 0，那么软件堆栈曾经溢出。在那个例子中制造了两个功能调用，它是两个堆栈都可能溢出的。无论如何，在_StackOverflowed 执行起作用时，第二个调用不可以出现。作为例子，如果函数复位了 CPU，那么将不能返回_StackCheck 函数。

缺省的 _StackOverflowed 函数

当它被调用时，库会用一个缺省的_StackOverflowed 函数来跳转到 0 的位置，因此复位 CPU 和程序。 你可能希望用一个函数来代替它以指示更多的错误条件，一个作为例子它可能切断所有的中断并且点亮 LED。注意自堆栈溢出指示故障程序以来，_StackOverflowed 函数或许不能执行任何太复杂的事或实现程序的正常工作。

这两个函数的原型在头文件 macros.h.中

七、AVR 硬件访问的编程

1、访问 AVR 的低层硬件

AVR 系列使用高级语言编程时有很高的 C 语言密度，它允许你对访问目标 MCU 的底层硬件进行访问。由于 AVR 性能，除了要最大程序地优化代码外很少使用汇编。偶然情况下目标 MCU 的硬件特点在 C 语言中不能很好地使用，很显然使用在线汇编和预处理宏能访问这些特点。

头文件 io*.h（如 io8515.h、iom603.h 等）定义了指定 AVR MCU 的 IO 寄存器细节。这些文件是从 ATMEL 官方发布的文件经过修改，以匹配这个编译器的语法要求。文件 macros.h 定义了许多有用的宏，例如宏 UART_TRANSMIT_ON( )能使 UART 开始工作。

这个编译器的效率很高，当访问由 IO 寄存器映射的内存时能产生单周期指令象 in、out、sbis、sbi 等。参考 IO 寄存器。

注意：老的头文件 avr.h 定义 IO 寄存器的 bit 有一些模糊，尽管 io*.h 定义了它们的 bit 的位置。因此使用 io*.h 和 IO 寄存器的 bit，很多时候你将需要使用定义在 macros.h 文件中的 BIT()宏。例如：

```
avr.h:
#define SRE          0x80// 外部 RAM 使能

... (你的 C 程序)

MCUCR |= SRE;

io8515.h
```

```
#define SRE          7

 ... (你的 C 程序)

 #include <macros.h>

 MCUCR  |= BIT(SRE);
```

2、位操作

一个共同的任务是编程微控制器 MCU 打开或关闭 IO 寄存器的一些位（bit）。很幸运，标准 C 有较好的和适用的位操作功能，而没有借助于汇编指令或其它非标准 C 结构，C 定义了一些按位进行的运算是很有用的。

a | b – 按位或，这个表达式指示中 a 被表达式中的 b 按位进行或运算。这惯用于打开某些位，尤其常用|=的形式，例如：

```
PORTA  |=  0x80;    //  打开位 7 (最高位)
```

a & b – 按位与，这个运算在检查某些位是否置 1 时有用，例如：

```
If ((PORTA & 0x81) == 0)    //  检查位 7 和位 0
```

注意圆括号需要括在&运算符的周围，因为它和= = 相比运算优先级较低，这是 C 程序中很多错误的原因之一。

a ^ b – 按位异或，这个运算对一个位取反有用。例如，在下面的例子中，位 7 是被翻转的：

```
PORTA  ^=  0x80;   //  翻转位 7
```

~a – 按位取反. 在表达式中这个运算执行一个取反。当用按位与运算关闭某些位时，与这个运算组合使用尤其有用，如：

```
PORTA  &=  ~0x80; //  关闭位 7
```

这个编译器对这些运算能产生最理想的机器指令，例如：sbic 指令可以用在根据位的状态进行条件分枝的按位与运算中。

3、程序存贮器和常量数据

AVR 是哈佛结构的 MCU，它的程序存贮器和数据存贮器是分开的，这样的设计是有一些优点的。例如，分开的地址空间允许 AVR 装置比传统结构访问更多的存贮器；例如，Atmega 系列允许有超过 64K 字（WORD）的程序存贮器和 64K 字节的数据存贮器。将来的 MCU 装置可能用到更多的程序存贮器，而程序计数器仍保留在 16 位上。

不幸的是，C 不是在这种机器上发明的。特别地，C 指针是任意一个数据指针或函数指针，C 规则已经指定你不可以假设数据和函数指针能被向前和向后修改。可是同是哈佛结构的 AVR，要求数据指针能指向任一个数据内存和程序内存。

非标准 C 解决了这个问题，ImageCraft AVR 编译器使用"const"限定词表示项目是在程序存贮器中。注意对指针描述，这个 const 限定词可以应用于不同的场合，不管是限定指针变量自己还是指向项目的指针。例如：

```
const int table[] = { 1, 2, 3 };
const char *ptr1;
char * const ptr2;
const char * const ptr3;
```

"table"是表格式样分配进程序存贮器，"ptr1"是一个项目在数据存贮器而指向数据的指

针在程序存贮器，"ptr2"是一个项目在程序存贮器而指向数据的指针在数据存贮器，最后，"ptr3"是项目在程序存贮器而指向数据的指针也在程序存贮器。在大多数的例子中"table" 和和"ptr1"是很典型的，C 编译器生成 LPM 指令来访问程序存贮器。

注意 C 标准不要求"const"数据是放入只读存贮器中，而且在传统结构中，除了正确访问就没有要紧的了。因而，在承认参数的 C 标准中使用 const 限定是非传统的。无论如何，这样做与标准 C 函数定义是有一定冲突的。

例如，标准"strcpy"的原型是 strcpy(char *dst, const char *src)，带有 const 限定的第二个参数表示函数不能修改参数。然而在 ICCAVR 下，const 限定词表示第二个参数指向程序存贮器是不合适的。因此这些函数定义设有 const 限制。

最后，注意只有常数变量以文件存贮类型放入 FLASH 中。例如定义在函数体外的变量或有静态存贮类型限制的变量。如果你使用有 const 限制的局部变量，将不被放入 FLASH 中而可能导致不明确有结果。

4、字符串

在哈佛结构的 AVR 中程序内存和数据内存分开给程序内存和数据内存的说明带来了一定的复杂性，这页说明字符串。

字符串

这个编译器将带有 const 说明的表和项目放入程序存贮器中，最困难的是字符串的分配和处理，问题在于 C 中将字符串转换为 char 指针。如果字符串是分配进程序存贮器中，那么所有字符串库函数中的任意一个必须被复制成不同于指针的操作，或者字符串也必须被分配在数据存贮器中。

ImageCraft 编译器提出了解决这个问题的两个方法：

缺省的字符串分配

这个缺省的方法是同时分配字符串在数据和程序存贮器中，所有涉及的字符串是拷贝进数据存贮器的。为了确保它们的值是正确的，在程序启动时字符串是由程序存贮器拷贝进数据存贮器中的，因此只有单一的字符串拷贝函数是必须的（编译器执行全局变量初始化也是这样处理的）。

如果你希望节省空间，你能使用常量字符型数组来将字符串只分配进程序存贮器中。例如：

```
const char hello[] = "Hello World";
```

在这个例子中，hello 可以在上下文中作为字符串使用，但不能用作标准 C 库中字符串函数的参数。

Printf 已被扩展成带%S 格式字符来输出只存贮于 FLASH 中字符串，另外，新的字符串函数已加入了对只存贮于 FLASH 中字符串的支持。

只分配全部字符串到 FLASH 存贮器中

当对应"Project->Options->Target->Strings In FLASH Only"检查框被选中时，你可以指挥编译器将字符串只放在 FLASH 中，这时称必须很小心地调用库函数。当这个选项是选中的，字符串类型"const char *"是有效的，并且你必须保证函数获得了合适的参数类型。除了新的"const char *"与字符串有关系外，创建了 cprintf 和 csprintf 函数承认字符串格式的类型。参考标准输入输出函数。

注意：当选项 2（只分配全部字符串到 FLASH 存贮器中）时，应使用 cprintf()。对 const char*及 const char ptr[ ]类型字符串，并且加%S 参数。

当选项 1 时，应使用 printf()。对 const char*及 const char ptr[ ]类型字符串，并且加%S 参数。

5、堆栈

生成代码使用两个堆栈：一个是用于子程序调用和中断操作的硬件堆栈，一个是用于以堆栈结构传递的参数、临时变量和局部变量的软件堆栈。

硬件堆栈起初是用于存贮函数返回的地址，它代表了许多小的软件堆栈。通常，如果你的程序没有子程序调用，也不调用象带有%f 格式的 printf()等库函数，那么默认的 16 字节应该在大多数的例子中能良好工作。在绝大多数程序中，除了很繁重的递归调用程序（再入式函数），最多 40 个字节的硬件堆栈应该是足够的。

硬件堆栈是从数据内存的顶部开始分配的，而软件堆栈是在它下面一定数量字节处分配。硬件堆栈和数据内存的大小是受在编译器选项中的目标装置项设定限制的。数据区从 0x60 开始分配，在 IO 空间后面是正确的。允许数据区和软件堆栈彼此相向生长。

如果你选择的目标装置带有 32K 或 64K 的外部 SRAM，那么堆栈是放在内部 SRAM 的顶部，而且向低内存地址方向生长。参考程序和数据内存的使用。

堆栈检查

任意一个程序失败的重要原因是堆栈溢出到其它数据内存的范围。两个堆栈中的任意一个都可能溢出，并且当一个堆栈溢出时会偶然产生坏的事情，你可以使用堆栈检查函数检测溢出情况。

6、在线汇编

除了在汇编文件中写汇编函数外，在线汇编允许你写汇编代码进你的 C 文件中。（当然，在你的工程使用汇编源文件作为一个部件是良好的。）在线汇编的语法是：

```
asm("<string>");
```

多个汇编声明可以被符号\n 分隔成新的一行，"String"可以被用来指定多个声明，除了额外增加的 ASM 关键词。为了在汇编声明中访问一个 C 的变量，可使用%<变量名>格式，如

```
register unsigned char uc;
asm("mov %uc,R0\n"
    "sleep\n");
```

任意一个 C 变量都可以被引用。如果你在汇编指令中需使用一个 CPU 寄存器，你必须使用寄存器存贮类（register）来强制分配一个局部变量到 CPU 寄存器中。

通常，使用在线汇编引用局部寄存器的能力是有限的。如果你在函数中描述了太多的寄存器变量，就很可能没有寄存器可用。在这种情况，你将从汇编程序得到一个错误。那时也不能控制寄存器变量的分配，所以你的在线汇编指令很可能失败。作为例子，使用 LDI 指令需要使用 R16~R31 中的一个寄存器，但这里没有请求使用在线汇编，同样也没有引用上半部分的整数寄存器。

在线汇编可以被用在 C 函数的内部或外部，编译器将在线汇编的每行都分解成可读的。不象 AVR 汇编器，ImageCraft 汇编器允许标签放置在任意地方，所以你可以在你的在线汇编代码中创建标签。当汇编声明在函数外部时，你可能得到一个警告，你不要理睬这个警告。

7、IO 寄存器

IO 寄存器，包括状态寄存器 SREG，可以被两条路线访问。IO 地址在 0x00 和 0x3f 之间，可以使用 IN 和 OUT 指令读写 IO 寄存器；或者使用在 0x20 和 0x5F 之间的数据内存地址，可以使用普通数据访问指令和地址模式。两种方法在 C 中都可使用：

数据内存地址，一个直接地址可以通过加指针类型符号直接访问。例如，SREG 的数据内存在地址是 0x5F:

```
unsigned char c = *(volatile unsigned char *)0x5F; // 读 SREG
*(volatile unsigned char *)0x5F |= 0x80;  // 打开全局断位
```

注意：数据内存地址 0 到 31 涉及到 CPU 寄存器，注意不要不注意地改变 CPU 寄存器。

当访问在 IO 寄存器范围中的数据内存时，编译器自动生成低级指令象 in、out、sbrs、sbrc 等是首选的方法，

IO 地址，你可以使用在线汇编和预处理宏来访问 IO 地址：

```
register unsigned char uc;
asm("in  %uc,$3F");// 读 SREG
asm("out  $3F,%uc");      // 打开全局中断位
```

注意：老的头文件 avr.h 定义 IO 寄存器的 bit 有一些模糊，尽管 io*.h 定义了它们的 bit 的位置。因此使用 io*.h 和 IO 寄存器的 bit，很多时候你将需要使用定义在 macros.h 文件中的 BIT()宏。例如：

```
avr.h:

#define  SRE         0x80// 外部 RAM 使能

... (填充你的 C 程序)

MCUCR  |=  SRE;

io8515.h

#define SRE           7

... (填充你的 C 程序)

#include <macros.h>

MCUCR  |=  BIT(SRE);
```

8、**绝对内存地址**

你的程序可能需要使用绝对内存地址，例如外部 IO 设备通常被映射成特殊的内存。这些可能包括 LCD 界面和双口 SRAM，通常你可以使用在线汇编或单独的汇编文件来描述那些定位在特殊内存地址的数据。在稍后版本的编译器中，已在 C 语言中提供这些能力。

在下面有例子中，假设有一个两字节的 LCD 控制寄存器定位在 0x1000 地址，一个两字节的 LCD 数据寄存器定位在 0x1002 地址，并且有一个 100 字节的双口 SRAM 定位在 0x2000 的地址。

使用汇编模式，在一个汇编文件中输入以下内容。

```
.area memory(abs)
.org 0x1000
_LCD_control_register:: .blkw 1
_LCD_data_register:: .blkw 1
.    org 0x2000
_dual_port_SRAM::       .blkb 100
```

在你的 C 文件中必须这样描述：

```
extern unsigned int LCD_control_register, LCD_data_register;
extern char dual_port_SRAM[100];
```

注意：
界面规定在汇编文件中外部变量名称是带'_'前坠的，并且使用两个冒号定义为全局变量。

使用在线汇编：

在线汇编遵守同样的汇编语法规则，除了它被附加了一个 asm()**伪函数。在** C **文件中，关于上面的汇编代码被变为如下代码：**

```
asm(".area memory(abs)"
    ".org 0x1000"
    "_LCD_control_register:: .blkw 1"
    "_LCD_data_register:: .blkw 1");

asm(".org 0x2000"
    "_dual_port_SRAM:: .blkb 100");
```

在 C **中你仍然要使用**"extern"**描述变量，正象上面使用单独的汇编文件那样，否则** C **编译器不会真正知道在** asm **中的声明。**

9、C **任务**（Tasks）

作为汇编界面的描述和调用规则，编译器通常在生成代码来保存和恢复保护的寄存器。在一些情况下，这些行为可能是不合适的。例如，如果你使用 RTOS（**实时操作系统**），RTOS **管理着寄存器的保存和恢复并作为任务切换处理的一部分，编译器如果再插入这些代码就变得多余了。**

为了禁止这种行为，你可以使用"#pragma ctask"**，例如：**

```
#pragma ctask drive_motor emit_siren
  ....
  void drive_motor() { ... }


  void emit_siren() {...}
```

这个附注(pragma)**必须被用在函数定义之前。注意作为默认的情况，从不返回的程序**"main"**是有这个属性的，它也没有必要为返回保存和恢复任意一个寄存器。**

10、 **中断操作**

C **中断操作**

中断操作中 C **中可以使用，无论函数定义在文件的什么地方，你必须用一个附注**（pragma）**在函数定义之前通知编译器这个函数是一个中断操作：**

```
#pragma interrupt_handler <name>:<vector number> *
```

"vector number" **中断的向量号，注意向量号是从** 1 **开始的，那是复位向量。这个附注有两个作用**:

对中断操作函数，编译器生成 RETI **指令代替** RET **指令，而且保存和恢复在函数中用过的全部寄存器。**

编译器生成以向量号和目标 MCU **为基础的中断向量。**

例如：

```
#pragma interrupt_handler      timer_handler:4
```

```
 ...

void timer_handler()
 {
   ...
 }
```

编译器生成的指令为

```
rjmp  _timer_handler        ; 对普通 AVR  MCU
```

或者

```
jmp  _timer_handler         ;对 Mega  MCU
```

上述指令定位在 0x06（字节地址，针对普通装置）和 0x0c（字节地址，针对 Mega 装置）。（Mega 使用 2 个字作为中断向量，非 Mega 使用 1 字作为中断向量）

如果你希望对多个中断入口使用同一个中断操作， 你可以在一个 interrupt_handler 附注中放置多个用空格分开的名称，分别带有多个不同的向量号。例如：

```
#pragma interrupt_handler timer_ovf:7 timer_ovf:8
```

汇编中断操作

你可以用汇编语言写中断操作。如果在你的汇编操作内部调用 C 函数，无论如何要小心。汇编程序要保存和恢复挥发寄存器（参考汇编界面），C 函数不做这些工作。

如果你使用汇编中断操作，那么你必须自己定义向量。你使用"abs"属性描述绝对区域，用".org"来声明 rjmp 或 jmp 指令的正确地址。注意这个".org" 声明使用的是字节地址。

```
 ;  对全部除 ATMega 以外的 MCU

  .area  vectors(abs)       ;中断向量

 .org 0x6

 rjmp _timer

;  对 ATMega MCU

.area  vectors(abs)         ;  中断向量

.org 0xC

jmp _timer
```

11、 访问 UART

默认的库函数 getchar 和 putchar 使用查寻模式从 UART 中进行读写。在\icc\examples.avr 目录，有一个以中断方式工作的 IO 程序可以代替默认的程序。

12、 访问 EEPROM

EEPROM 在运行时可以使用库函数访问，在调用这些函数之前加入 #include <eeprom.h>。

EEPROM_READ(int location, object)

这个宏调用了 EEPROMReadBytes 函数从 EEPROM 指定位置读取数据送给数据对象，"object"可以是任意程序变量包括结构和数组。例如：

```
int i;

EEPROM_Read(0x1, i); // 读 2 个字节给 i
```

EEPROM_WRITE(int location, object)

这个宏调用了 EEPROMWriteBytes 函数将数据对象写入到 EEPROM 的指定位置，"object"可以是任意程序变量包括结构和数组。例如：

```
int i;

EEPROM_WRITE(0x1, i);    //写两个字节至 0x1
```

这些宏和函数可以用于任意 AVR 装置。可是对 EEPROM 单元少于 256 字节的 MCU，即使不需要高地址字节它们也是欠佳的，因为它仍然是要写的。如果它关系重大，你可以为 EEPROM 较少的目标装置重新编译库源代码。

初始化 EEPROM

EEPROM 可以在你的程序源文件中初始化，在 C 源文件中它作为一个全局变量被分配到特殊调用区域"eeprom."中的，这是可以用附注实现的，结果是产生扩展名为.eep 的输出文件。例如：

```
#pragma data:eeprom

int foo = 0x1234;

char table[] = { 0, 1, 2, 3, 4, 5 };

#pragma data:data

…

int i;

EEPROM_READ((int)&foo, i);    // i 等于  0x1234
```

第二个附注是必须的，为返回默认的"data."区域需要重设数据区名称。

注意因为 AVR 的硬件原因，初始化 EEPROM 数据至 0 地址是不可以使用的。

注意当使用外部描述（比如访问在另一个文件中的 foo），你不需要加入这个附注，例如：

```
extern int foo;

int i;
```

EEPROM_READ((int)&foo, i);

内部函数

如果需要下列函数可以直接使用，但是上面关于宏的描述对大多数装置应该是有能力的。

unsigned char EEPROMread(int location)

从 EEPROM 指定位置读取一个字节

int EEPROMwrite(int location, unsigned char byte)

写一个字节到 EEPROM 指定位置，如果成功返回 0。

void EEPROMReadBytes(int location, void *ptr, int size)

从 EEPROM 指定位置处开始读取"size"个字节至由"ptr."指向的缓冲区。

void EEPROMWriteBytes(int location, void *ptr, int size)

从 EEPROM 指定位置处开始写"size"个字节，写的内容由"ptr."指向的缓冲区提供。

13、**访问 SPI**

一个以查寻模式访问 SPI 的函数是提供的，更多的信息参考 spi.h。

14、**相对转移/调用的地址范围**

一个带 8K 程序存贮器的装置，全部范围内的跳转可以使用相对转移和调用指令(rjmp 和 rcall)。为实现这个目的，相对转移和调用的范围是以 8K 为分界的。例如，一个较远的跳转跳转到 0x2100 字节处（0x2000 为 8K）实际上会跳转到地址 0x100 处。

这个选项是由工程管理器自动检测的，只要目标装置的程序存贮器是 8K 的。

八、C 的运行结构

1、数据类型

类型	长度（字节）	范围
unsigned char	1	0..256
signed char	1	-128..127
char (*)	1	0..256
unsigned short	2	0..65535
(signed) short	2	-32768..32767
unsigned int	2	0..65535
(signed) int	2	-32768..32767
unsigned long	4	0..4294967295
(signed) long	4	-2147483648..2147483647
float	4	+/-1.175e-38..3.40e+38
double	4	+/-1.175e-38..3.40e+38

(*) "char" 等同于 "unsigned char"

floats 和 doubles 是 IEEE 标准 32 位格式，7 位表示指数，23 位表示尾数，1 位表示符号。

位域类型必须被赋予 unsigned 或 signed 关键字，而且将被包含在一个较小的空间中。如可定义成结构：

```
struct {
        unsigned a : 1, b : 1;
```

};

这个结构体的长度只有一个 1 byte。位域是从右往左放置的。

2、 汇编界面和调用规则

a)、名称

C 语言中的名称在汇编文件中是以下划线为前坠的，如函数 main( )在汇编模块中是以 _main( )引用的。名称的有效长度为 32 个字符，在名称后面加两个冒号（::），可以定义成一个全局变量。例如：

```
_foo::
    .word 1
```

(在 C 文件中)

```
extern int foo;
```

b)、传递参数和返回值所使用的寄存器

第一个参数若是整型则通过 R16/R17 传递，第二个参数则通过 R18/R19 传递。如果参数是长整型或浮点数则通过 R16/R17/R18/R19 传递；其余参数通过软件堆栈传递。比整型参数小的（如 char）参数扩展成整型（int）长度传递，即使函数原型是可用的。如果 R16/R17 已传递了第一个参数，而第二个参数是长整型或浮点数，则第二个参数的低半部分通过 R18/R19 传递，而高半部分通过软件堆栈传递。

整型返回值是通过 R16/R17 返回，而长整型或浮点数返回则通过 R16/R17/R18/R19 返回。

c)、保护的寄存器

在汇编函数中必须保护和恢复下列寄存器：

R28/R29 或 Y，这是结构指针。

R10/R11/R12/R13/R14/R15/R20/R21/R22/R23 ，这些寄存器是调用保护寄存器，这些寄存器的内容在被汇编语言函数调用后必须保持不变。

d)、挥发寄存器

别的寄存器如

R0/R1/R2/R3/R4/R5/R6/R7/R8/R9/R24/R25/R26/R27/R30/R31

SREG

可以在汇编语言函数中使用，而不被保护和恢复。这些寄存器是调用挥发寄存器，这些寄存器的内容在被函数调用后可以改变。

e)、中断处理

这不同于普通的函数调用，在中断操作中必须保护和恢复它所使用的全部寄存器。如果你是使用 C 函数来描述中断处理，那么编译器有能力自动完成的。如果使用汇编写中断处理，而它又调用了普通的 C 函数，那么汇编操作必须保护和恢复挥发性寄存器，普通 C 函数调用不保护它们。中断处理操作同普通程序操作是异步的，中断处理或它的函数调用不能改变任意一个 MCU 寄存器。

3、 函数返回非整型值

在调用函数前，必须描述其返回的一个长整型、浮点数或结构值。作为例子，在调用任意浮点函数之前，应当用#include 语句包含头文件<math.h>。否则，在这些程序返回它们的值后你的程序将不工作，这和那些返回整型值的函数是有不同之处的。

返回长整型数或浮点数

长整型数或浮点数返回值是设定在一些寄存器 R16-R19 中。

传递结构值

如果传递结构值，结构允许通过堆栈传递，而不是通过寄存器。传递结构索引(也就是

传递结构的地址）和传递任意数据项目的地址是相同的，都是通过一个 2 字节的指针。

返回结构值

当一个返回结构的函数被调用时，这个调用函数分配一个临时贮藏库，而且传递一个隐藏指针给调用函数。当这个函数返回时，它拷贝返回值进这个临时贮藏库。

4、程序和数据区的使用

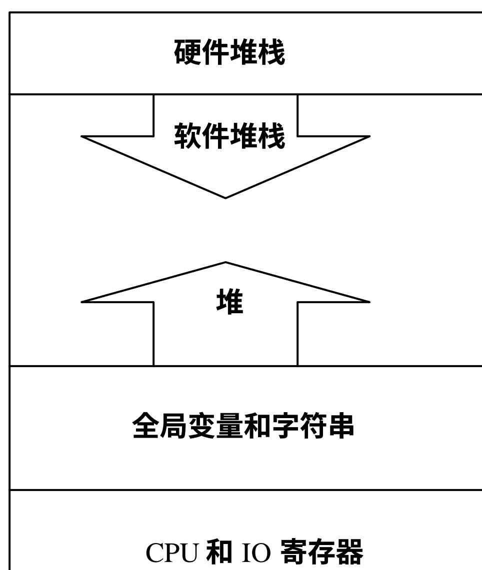
程序存贮器

程序存贮器是被用于保存你的程序代码、常数表和确定数据的初始值比如字符串、全局变量。编译器可以生成一个对应程序存贮器映像的输出文件（HEX 文件），这个文件可以被编程器用来对芯片编程。

通常，编译器不能使用任意 64K 字节以上的程序存贮器。为了访问 64K 字节边界以上的存贮器（如在 Mega103 装置中），你需要在设定 RAMPZ 寄存器后直接调用 ELPM 指令。

数据存贮器（仅指内部 SRAM）

这个数据存贮器是被用于保存变量、堆栈结构和动态内存分配的堆。通常，它们不产生输出文件但在程序运行时使用，一个程序使用数据内存如下图。



内存图的底部是地址 0，开始的 96（0x60）字节是 CPU 寄存器和 IO 寄存器。编译器从 0x60 往上放置全局变量和字符串，在变量区域的顶部你可以分配动态内存。在高端地址，硬件堆栈开始于 SRAM 的最后位置，在它的下面是向下生长的软件堆栈。它要求你作为程序师，要确保硬件堆栈不生长进软件堆栈，而软件堆栈不生长进堆，否则将导致意外的结果。

数据存贮器（外部 SRAM）

如果你选择带有 32K 或 64K 外部 SRAM 的目标装置，那么堆栈是放置在内部 SRAM 的顶部并且是朝低端内存地址向下生长。数据内存是开始于硬件堆栈的顶部并且向上生长。这样分配的原因是在多数场合访问内部 SRAM 比访问外部 SRAM 的速度要快，分配堆栈到较快的内存是有很多好处的。

5、 编程区域

编译器生成代码和数据到不同的区域"areas."，区域按照内存地址增高的顺序被编译器使用。

只读存贮器

interrupt vectors -----这个区域包括中断向量

func_lit – 函数表区。这个区的每个字包括了函数入口的地址，为了与代码压缩完全兼容，所有间接的函数索引必须通过间接的额外对准。如果你在 C 中使用函数指针调用函数，这是自动完成的。在汇编中，举例如下：

```
; 假设 _foo 是函数的名称
.area func_lit
PL_foo:: .word _foo ; 创建函数表入口
. area text
ldi R30,<PL_foo
ldi R31,>PL_foo
rcall xicall
```

你可以间接地在函数表入口地址送入 R30/R31 寄存器对后，使用库函数 xicall 调用这个函数。

lit – 这个区域包括了整型数和浮点数常量

idata -全局变量和字符串的初始值保存在这个区域。

text – 这个区域包括程序代码。

数据内存

data -这个区域包括全局变量、静态变量和字符串，全局变量和字符串的初始值是保存在"idata" 区域，并且是在启动时被拷贝进数据区的。

bss -这个区域包括未初始化的全局变量，按 ANSI C 定义这些变量在启动时将初始化为 0。

EEPROM 存贮器

eeprom - 这个区域包括 EEPROM 数据，EEPROM 数据是写进扩展名为.eep 的输出文件，其格式 INTEL HEX 文件格式。

九、调试

ICCAVR 可以输出 COFF 格式调试文件，使用户可在 ATMEL 的 AVRStudio 中进行源程序级的调试。如果用户想使用 AVRStudio 中的模拟 IO 及终端仿真器，那么在 ICCAVR 的编译选项中必须将"AVR Studio Simulator IO"一项打钩。

十、ICCAVR 汇编参考

1、汇编语法

汇编有以下语法

1)、名称

所有汇编名称必须由下列字符组成：

('_' | [a-z]) [[a-z] | [0-9] | '_'] *

在 ICC 中汇编名称必须由下划线或字母开始，随后跟字母、数字或下划线组成。在这个文档中，名称和符号是同名词。名称可以是表示一个常数值的符号名称或代表某一个 PC

地址的标号名称中的任意一个。一个名称的长度最多为 30 个字符长，而且区分大小写（汇编指令和汇编伪指令除外）。

符号可以只用在程序模块中，也可显式地被其它模块使用。在以前的例子中，符号表示局部符号，而在以后的例子中表示全局符号。

如果在一个文件一个符号没有被定义而直接使用了，那么它是被假设为在其它文件已经定义，而且它的值由链接器决定。

2）、数

如果数带有一个 0x 或$前坠，那么这个数是一个十六进制数

例如: 10

0x10

$10

0xBAD

0xBEEF

0xC0DE

-20

3）、汇编文件格式

汇编文件必须是一个 ASCII 文件（纯文本文件）而且要遵守一定的规则，文件的每一行应该是如下的格式：

[label: [:]] [command] [operands] [;comments]

label 表示标号，一个冒号表示局部符号，两个符号表示全局符号

command 表示操作码（指令码）

operands 表示操作数

[ ] 表示为可选项

comments 表示注释，注意在 C 文件中注释用 / /或 /* …*/ 引导，而在汇编文件中用分号 ；或 // 引导

上式的每项之间必须用一个或多个空格分开，系统汇编时对注释部分不进行处理。

标号

一个后面跟着一个或两个冒号的名称表示标号，标号的值是程序中某一点的 PC 计数器的值，一个标号带两个符号表示全局符号，它在其它模块中也是显式的。

操作码应该是 AVR 指令、伪指令或宏，操作数是指指令所需要的参数。在这个文档中不描述 AVR 指令，因为它们与标准的 Atmel 文档中的指令描述相同。例外的是：

xcall

应用于 Mega 芯片，而且只支持长调用或跳转指令，它可解释为 rcall 或 call 中的任意一个。

xjmp

应用于 Mega 芯片，而且只支持长调用或跳转指令，它可解释为 rjmp 或 jmp 中的任意一个。

4）、表达式

在指令后的操作数可以是表达式，例如直接地址是一个最简单的表达式。

lds R10,asymbol

asymbol 是一个简单表达式有例子，它是一个符号或标号名称。通常一个表达式描述为：

```
expr: term |
          { expr }
           unop expr
           expr binop expr
           term: .        ；当前程序计数器 PC 值
                  name |
               #name
```

圆括号用于分组，其运算优先级高于运算符。表达式不能随意使用，有一个限制是使用链接器的重定位信息，表达式的基本规则是，它只能出现一个重定位符号，例如

lds R10,foo+bar

如果 foo 和 bar 两个全都是外部符号，那么表达式无效。

5）、运算符

下面列出了各种运算符和它们的优先级，运算符的优先级是很有用的。只有加号可以用于重定位符号计算（比如外部符号），其它所有运算符必须用于常数或本文件中定义的符号的运算。

运算符	功能	类型	优先级
*	乘法	二进制	10
/	除法	二进制	10
%	取模	二进制	10
<<	左移	二进制	5
>>	右移	二进制	5
^	按位异或 XOR	二进制	4
&	按位与 AND	二进制	4
|	按位或	二进制	4
-	负号	一元运算符	11
~	取补运算	一元运算符	11
<	取低字节	一元运算符	11
>	取高字节	一元运算符	11

6）、圆点（.）或程序计数器

如果圆点出现在表达式中，那么当前程序计数器的值被放置在圆点的位置。

2、汇编伪指令

1）、.area <name> [(attributes)]

定义代码或数据装入的内存区域，链接器将所有使用同一名称的区域集合至一起，并且根据它们的属性进行连接或覆盖。

属性有两类：一类为

abs，或；绝对定位区域

rel　　；重定位区域

另一类为

con，或　；连接定位

ovr　　　；覆盖定位

绝对定位区域的起始点地址是在汇编文件中由它自己指定的，而重定位区域的起始地址是由送命令选项给链接器来指定的。对带连接属性的区域，链接器连接这个区域到另一

个同名区域后面；对带有覆盖属性的区域，对每一个文件链接器都是从同一地址开始安排区域。下面举例说明它们的区别：

file1.o:

.area text ；10 bytes, **调用** text_1

.area data ；10 bytes

.area text ；20 bytes, **调用** text_2

file2.o:

.area data ；20 bytes

.area text ；40 bytes, **调用** text_3

text_1 **和** text_2 **在这个例子中是正确的名称。实际上，它们是不会获得一个单独的名称的，让我们假设** text **区域的起始地址设置为** 0**。如果这个** text **区域有** "con"**属性，那么** text_1 **将从** 0 **开始，** text_2 **从地址** 10 **开始，而** text_3 **从** 30 **地址开始。如果这个** text **区域有** "ovr" **属性，那么** text_1 **和** text_2 **将分别地从** 0 **和** 10 **开始，但对** text_3 **由于它在另外一个文件中定义，所以它将同样从** 0 **地址开始。所有同名的区域必须有相同有属性，即使它们用在不同的模块中。下面是具有合适属性的全部例子：**

.area foo(abs)

.area foo(abs,con)

.area foo(abs,ovr)

.area foo(rel)

.area foo(rel,con)

.area foo(rel,ovr)

.ascii strings?

2）、.ASCⅡ **字符串**

.ASCIZ **字符串**

这个伪指令用于定义字符串，无论哪一个都要附上一对分界符，在两个分界符之内，任意可打印字符都是有效的。下面是 C **语言中的类型转义字符，转义字符都是从反斜线** \ **开始的：**

\e ESC

\b **退格**

\f **换页**

\n **换行**

\r **回车**

\t TAB

\<**最多三个八进制数**> **字符是等于这个八进制数的值**

ASCIZ **定义在字符串的结尾增加了** NUL **字符** (\0)，**它使**\0 **正确地嵌入字符串的内部。**

例如: asciz "Hello World\n"

asciz "23\0456'

3）、.byte <expr> [,<expr>]*
.word <expr> [,<expr>]*
.long<expr> [,<expr>]*

这些伪指令是定义常数，它们分别表示字节常数(byte)、**字常数**(2byte)**和双字长数**(4byte)。**字和双字常数是以高低字节倒置的格式输出，这个格式用于** AVR **微控制器**(MCU)。**注意双字常数只能用作操作数，而另外两个可以用于重定位表达式。**

例如: .byte1, 2, 3
.word label,foo

4）、.blkb <value>
.blkw <value>
.blkl <value>

这些伪指令是保留空间而没有给它们赋值。指令后面的数分别是指保留的字节、字或双字的数目。

5）、.define <symbol> <value>

定义一个文本替代符，无论何时"symbol"**可用在表达式中（如寄存器符号或标号），它是用**"value."**定义的，例如：**

```
.define quot R15
mov     quot,R16
```

6）、.if <symbol name>
.else
.endif

上述三个伪指令定义了一个条件汇编语句。如果条件<symbol name>**为真（非** 0**）则执行**.if **和**.else **之间的指令，如果条件**<symbol name>**为假（等于** 0**）则执行**.else **和**.endif **之间的指令。**.else **可以省略，条件语句最多可以嵌套** 10 **层。如：**

```
.if cond
lds R10,a
.else
lds R10,b
.endif
```

如果 cond **不等于** 0，**则将** a **装入** R10，**如果** cond **等于** 0，**则将** b **装入** R10。

7）、.macro <macroname>

定义一个宏，由一直到 .endmacro **之间的所有声明组成宏。除了另外一个宏声明外，任意汇编声明可以是宏的组成部分。在宏内部表达式**@digit(digit **由** 0~9 **的数字代替**)**是宏被调用时相应的宏参数。定义的宏名称不能与汇编指令及汇编伪指令名称相冲突。例如：**

定义宏名 "foo"

```
.macro foo
lds @0,a
mov @1,@0
```

.endmacro

调用宏 foo 需要两参数，如

foo R10,R11

等同于

lds R10,a

mov R11,R10

8)、.endmacro　结束一个宏定义

9)、<macro> [<arg0> [,<args>]*]

调用宏是在操作码的位置放置宏名，后面跟上相应的参数。汇编器使用组成宏的声明来替换宏，同时用相应的宏参数来扩展@digit。你可以指定多个参数。

例如:foo bar,x

调用宏 foo，而且带两个参数 bar 和 x.。

<symbol> = <value>

定义一个符号等于常数值，如:

foo = 5

10)、.include "<filename>"

处理指定的由文件名称指定的文件，如果当前目录该文件不存在，汇编器将试图按文件名称指定的路径打开指定的文件。

如: .include "registers.h"

11)、.org <value>

设定程序计数器 PC 的值为"value."，这个伪指令中在带有"abs"属性的区域内有效。注意"value"是字节地址。

例如:.area interrupt_vectors(abs)

org 0xFFD0

dc.wreset

12)、.globl <symbol> [, <symbol>]*

将一个符号定义为全局符号，使其在当前或其它模块中都是显式的，这也和跟着两个冒号的标号相同。否则，符号只能在当前模块中使用。