HCS08 微控制器上有关内存分配的几个问题

师英，shylion@gmail.com

这里讨论 HCS08 **微控制器上的几个有关内存分配的问题，提及其存储器编址模型；数据对齐；**Tiny **和** Small **两种内存模型的差异；以及堆和栈的分配。指正错误与讨论其中细节，请电邮到** shylion@gmail.com**。谢谢。**

目录：

HCS08 **微控制器上有关内存分配的几个问题** ..... - 1 -
1.1. HCS08 **的存储器映射** ..... - 2 -
1.1 **外设寄存器** ..... - 2 -
1.2 RAM ..... - 5 -
1.3 FLASH ..... - 5 -
1.4 **向量（**Vectors**）** ..... - 6 -
1.2. **数据对齐** ..... - 7 -
1.3. HCS08 **的存储器模型：**Tiny **和** Small ..... - 9 -
1.4. **堆（**heap segment**）** 12
1.5. **栈（**stack segment**）** 13

1.1. HCS08 的存储器映射

每个 HCS08 微控制器的存储器映射（Memory Map）都不一样，但是它们都有相同的分配结构——一个线性的统一编址的 16bit （总共 64K）寻址空间。下面我们以 MC9S08AW60 为例认识它的存储器映射。

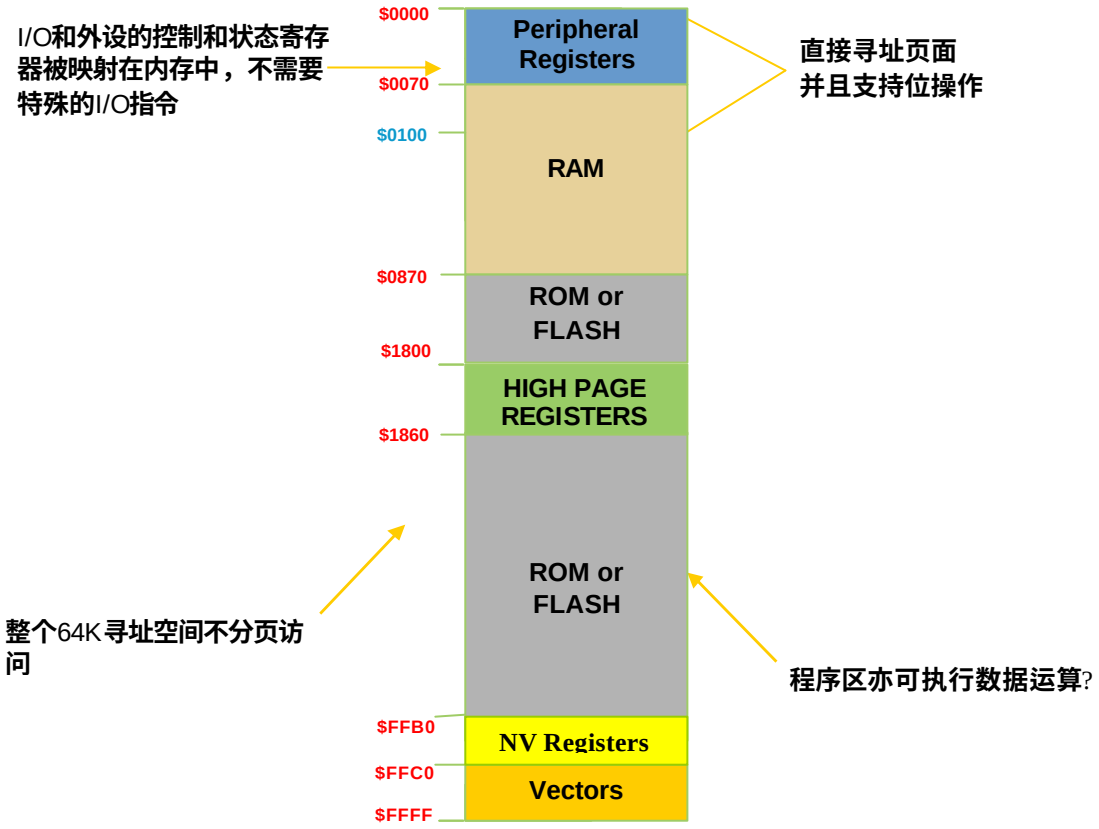

图 1- AW60 的存储器映射

1.1 外设寄存器

外设寄存器的名称区别于 CPU 寄存器。这些寄存器用来控制外设的行为和属性，或者获取外设的状态。微控制器的所有和真实世界的信息交互和对真实世界的“ 控制 ”在应用程序级都是通过操作这些寄存器完成的。

CPU08 的所有外设寄存器都被映射在存储器中，因此不需要特别的 I/O 指令完成对外设的操作。有些微控制器的外设寄存器没有被映射到存储空间来，所以对外设的操作完全依靠 I/O 类指令完成，所以需要额外的代码长度和执行周期作为代价。例如，读取一个端口的状态，然后改变其中的某些位再写回去这样的操作，在 CPU08 上只需要一条指令 BSET。而在某个 8bit 微控制器上，则需要如下的操作：

```
IOREAD      PTA
OR          0b00000001
IOWRITE     PTA
```

第一条指令读取 PTA，第二条指令把 A 的第 0 位置位，第三条指令写回端口 A。这实际上是一个读取-修改-写回（Read-Modify-Write）的过程，但是额外的

存储空间和指令周期被浪费在一个本应该简单的操作上。同样的功能用 CPU08 的汇编语言这样写就可以了：

```
BSET    0, PTAD          ; 机器码为 0xb0（OPCODE）, 0x00（operand）
```

这条指令使用了 2bytes 的指令长度和 5 个指令周期。

CPU08 对并不是把所有的外设寄存器都逐个安排在一段连续的地址空间中，而是根据对它们访问的频率分别放在 3 个不同的位置。这是一种战略优化。

- **直接页面（Direct Page）**
- **高地址页面（Hige Page）**
- **FLASH**

CPU08 把大部分需要经常访问的外设寄存器都映射在寻址空间的直接页面（地址 0x0100 以前），使得对它们的访问更加高效，并且可以进行直接的位操作。对于 AW60，0x0000~0x006F 的地址空间映射了这些外设寄存器。

另外一部分寄存器被映射在 0x1800 以后的扩展寻址范围中，这些寄存器叫做“ High Page Registers ”。这部分空间被安排了一些不经常访问的寄存器，如 SRS，SBDFR，SOPT，SMCLK，SDIDH:SDIDL，SRTISC，SPMSC1，SPMSC2，背景调试用寄存器，FLASH控制寄存器，I/O 端口内部上拉器件、摆率控制、输出能力选择寄存器等。一般来说，这些硬件级的选项在应用程序初始化的时候做一次设置，后面基本不用再去改动它，或者很少改动。关于这些寄存器的说明需要查阅各个微控制器的数据手册。这里我们来看其中的几个例子：

- **System Reset Status Register (SRS)**

这个只读寄存器包含了一些指示标志，用来显示上一次系统复位的原因。对于 AW60，分别为上电复位（POR），复位引脚（PIN），看门狗（COP），非法操作码（ILOP），内部时钟发生器（ICG）和低电压检测（LVD）。这些标志位对于应用程序判断系统异常复位并给出不同的应对措施，或者对应硬件故障诊断很有用处。

向 SRS 写入任何数据会导致看门狗定时器（COP）复位（这个动作经常被叫做“ 喂狗 ”），而不会影响 SRS 的内容。在 CodeWarrior C 头文件中，喂狗被定义成这样一个宏（我们终于看到 C 语言的代码了）：

```
#define __RESET_WATCHDOG() {asm sta SRS;}
```

在这些寄存器中，有一些设置位是“ 只写一次（Write-once） ”的。比如：

- **System Option Register (SOPT)**

这个寄存器包含了 3 个只写一次的控制位，分别控制看门狗使能或者关闭（COPE），看门狗溢出周期选择（COPT），休眠模式使能或者禁止（STOPE）等。应用程序只有一次机会可以改变这些设置，这有点像有些微控制器中的所谓“ 设置位（Configuration Bits） ”，只不过那些设置位并没有映射在内存地址中。

> 注意：如果打开了看门狗（COPE = 1），就要记得时常去喂它。否则它的溢出会导致系统异常复位。
> 注意：如果休眠模式没有被使能（STOPE = 0），程序中的“ STOP ”指令会导致一个“ 非法指令复位 ”异常。

有一些 High Page Registers 对于所有 HCS08 微控制器，其地址都是固定的。比如调试器强制复位（SBDFR），器件 ID 寄存器等。这是为了方便调试器和编程器厂商。

- **System Background Debug Force Reset (SBDFR)**

对于所有 HCS08 微控制器，其地址都是 0x1801。这个寄存器只有一个有效位，第 0 位 BDFR。向这一位写 1 会强制使系统复位。但是，用户程序并不能操作这个控制位，只有通过背景调试模块（BDM）的命令才能操作它。读这个寄存器永远得到 0x00。

- **System Device Identification Register (SDIDH:SDIDL)**

对于所有的 HCS08 微控制器，其地址都是 0x1806:0x1807。16bit 中一共有 12 位是有效的（SDIDH 的低 4 位和 SDIDL 的所有 8 位），它们指示出这个这个硬件的类型。AW60 的 ID 是 0x008。

还有一些叫做“非易失寄存器（Nonvolatile Register）”的单元被映射在0xFFB0到0xFFBF的空间中。这些单元其实就是一些FLASH单元，被指定为安置一些系统配置的参数。如NVBACKKEY，NVPROT，NVOPT等。对这些寄存器的赋值可以通过定义一些指定地址的常数（const）来完成。这些寄存器其实比“只写一次”的寄存器在意义上更接近“配置位”。在HCS08中，这些寄存器主要用来设置系统存储器安全机制（Security）和保护机制（Protection）。

```
; 在汇编语言中给 NVPROT 赋值。Bit0 为 0，bit7~bit1 作为保护地址高 7 位，低地址为
; 0x00FF，因此最后一个不受保护的地址单元为 0xDFFF，也就是说，保护从 0xE000d 到
; 0xFFFF 的空间。
        ORG     #$NVPROT
        DB      #$DE
// 在 C 语言中给 NVPROT 赋值。
        const unsigned char _nvprot @ FFBF = 0xDE
```

- **Flash Options Register，NVOPT，地址0xFFBF**

在这个寄存器中，包含了FLASH加密的几个设置位。很多程序员在设计一个项目的时候，他们会考虑硬件的安全性，免得自己抓断很多头发才写出来的代码被人家几分钟之内给抠出来，复制自己的产品；另外一些例如认证和加密的应用，使得硬件的安全性尤为重要。HCS08提供了系统存储器的安全机制。这个安全机制的设置是被编程在FLASH地址0xFFBF，由其bit1和bit0，SEC01:SEC00决定。在SEC01和SEC00的4种可能取值中，只有1:0这种组合禁止安全机制，其它的取值使能此机制——该地址被擦除后（所有bit全1），安全机制也是被打开的。

系统复位后，NVFOPT的内容被copy到高地址寄存器FOPT中，安全机制开始发挥作用。HCS08的安全机制使得微控制器的RAM和FLASH被作为安全资源，任何没有通过认证的访问都被禁止。直接页面寄存器，高地址寄存器和背景调试控制器不受安全机制的保护。在安全资源中运行的程序代码可以访问存储器空间的任何资源，而安全资源以外的应用程序或者背景调试接口试图访问安全资源都会被禁止。除了擦除整个FLASH。

- **NVBACKKEY，地址0xFFB0~0xFFB7**

如果FOPT寄存器中的bit7，KEYEN被置位，则安全机制的“后门密钥”被使能。这种情况下，应用程序可以通过校验8bytes的后门密钥来暂停安全机制。NVBACKKEY[8]这个常数数组的值为编程者预设的后门密码。

> 注意：没有任何一种加密方法是绝对安全的。一种安全机制被攻击者搞定的可能性正比与它能给攻击者带来的价值。为了提高安全性能，在设置后门密钥的时候，可以考虑使用一些摘要算法（例如 MD5 或者 SHA-1）的结果或者随机数作为密钥。简单的词组和生日等密钥比较容易被字典攻击搞定。

- **NVPROT，地址0xFFBD**

NVPROT和FLASH的保护机制相关。保护机制不同于安全机制，它用来防止FLASH的内容被误改写和擦除。HCS08的FLASH的一个页面包含512bytes。应用程序可以选择需要保护的FLASH的起始页面地址，从这个地址后面的所有页直到最高地址之内的所有页都被保护。为了启用FLASH保护，NVPROT的bit0必须为0，bit7~bit1作为最后一个不受保护的FLASH地址的高7位，A15~A9；低9位默认为1。在复位后，NVPROT的内容被copy到FPROT寄存器。

在程序中包含一个bootloader的时候，保护机制尤其有用。Bootloader将是程序代码中至关重要的部分，它首先需要被保护起来，在任何情况下不应被破坏，通过它可以恢复和更新别的程序模块。

只要FLASH保护被启用，则复位向量和中断向量区也一定被保护起来。为了使bootloader能够更新向量，中断向量可以被重定位（redirection）——把中断向量表搬移到没有保护的地址空间去。通过设置NVOPT的bit6，FNORED，可以使中断向量被重定位。

1.2 RAM

CPU08 的 RAM 大小和器件类型相关。AW60 有 2048bytes 的 RAM 提供应用程序使用，地址为 0x0070~0x0870。在 RAM 里除了定义应用变量，栈也安排在 RAM 中。

地址小于 0x0100 的 RAM 是其中宝贵的资源，除了可以用高效的直接寻址方式访问，这部分单元还支持位操作。

1.3 FLASH

HCS08 的程序存储器为 0.25um 工艺的 FLASH，以 512bytes 为一个页组织。例如 AW60 有 124 页，共 124×512 = 63,280bytes。HCS08 的 FLASH 比 HC08 在性能上有很大提升。和 HC08 相比，我认为最值得圈点的是下面几点：

- **更快的访问（读取，编程，擦除）速度**
- **更多擦写周期，100,000 次**
- **内建的擦除和编程算法，全工作电压范围内可以工作**

注意：如果应用程序要实现 IAP 功能（特别是用于模拟 EEPROM 来存放数据），应该弄明白内嵌 FLASH 存储器的擦写次数（Program/Erase Cycles）以及其他一些基本概念的定义。FLASH 基本数据单元（bit）从 0 改写成 1 的过程叫做擦除（Erase）；从 1 改写成 0 的过程叫做编程（Program）。FLASH 可以按字节被编程，为了增加编程的速度，现代 FLASH 能支持所谓的突发（Burst）和页（Page）编程模式。这两种快速编程模式本质上还是按照字节操作的，只不过使用一些逻辑尽量减少地址线的更新来节省编程时间（HCS08 的 FLASH 支持突发编程模式）。FLASH 只能按页擦除。两次页擦除之间的操作为一个“擦写次数”，包括多次按字节编程操作和一次也擦除操作。

明白了擦写次数的定义，我们就有理论依据用一些数学的方法扩展 FLASH 的擦写次数。这将在后面的内容中详细讨论。

注意：所谓“内建的”编程和擦除算法，是相比 HC08 而言。HC08 的 FLASH 在进行编程/擦除操作的时候，需要应用程序或者调试工具提供完整的时序，包括编程电压产生和关闭，精确的延时等。而 HCS08 只需要发给命令寄存器相应的操作命令就可以了，编程电压和延时由硬件逻辑控制。这使得 HCS08 的 FLASH 更加安全可靠。设想应用程序在 HC08 的 IAP 中，打开编程电压后程序跑飞了，或者发生了逻辑错误，或者在单步调试状态，那么编程电压会一直加在存储单元阵列上，容易造成器件的永久损坏。

FLASH在微控制器中用来存储应用程序目标代码，常数表等内容。由于 CPU08 的 FLASH 可以实现自编程功能，所以也可以用作模拟 EEPROM 来存放数据。也可以实现 bootloader 这样的在线程序更新或者在线编程的功能。

如果 HCS08 器件 FLASH 大小超过 57kbytes，那么它的 编址可能和位于 0x1800 的高地址寄存器发生冲突。为了避免这种冲突，在这些器件上，FLASH 空间被划分为两块，来绕开高地址寄存器。例如 AW60，在 0x870~0x17FF 地址空间有 3984bytes 存储单元，在 0x1860~0xFFFF 地址空间有 59,296bytes 存储单元。除了地址不同，这些两部分 FLASH 在其它方面都是平等的。

注意：虽然 CPU08 的程序代码存放在 FLASH，但是你完全可以把一段代码存放在或者拷贝到 RAM 中去运行。这样做尤其给 FLASH 擦除/编程功能的实现提供了非常必要的方便。当 FLASH 阵列被加上高电压后，从 FLASH 取指是不安全的。

1.4 向量（Vectors）

CPU08 的向量包括复位向量（Reset Vector）和中断向量表（Interrupt Vector Table，IVT）。在系统复位后，复位向量（位于 FLASH 地址 0xFFFE:0xFFFF，其内容实际上是一个地址）被装载到 PC 中，应用程序从该地址开始执行。

在 Bill Blunden 的著作《虚拟机的设计与实现》中，Bill 形象地把向量表比喻为电话号码簿。每一个中断源都有一个唯一的“电话号码”——中断向量将中断源和其中断服务程序关联，当中断条件发生时，CPU 可以根据这个“电话号码”快速找到对应的中断服务程序（ISR）并转去执行。HCS08 的向量映射在 FLASH 的地址 0xFFC0~0xFFFF 之间，由于每个向量都是一个指向特定程序段的地址，每个向量的长度为 16bit。如果这个空间没有用完，剩余的存储空间可以被应用程序自由使用。

当中断发生时，CPU08 的行为为：

- 在栈中保存 CPU 寄存器

- 将 CCR 寄存器中的 I 置位，屏蔽其它中断
- 取得当前等待的中断请求中最高优先级的那个向量
- 从该向量指向的程序段中取得头 3 个字节的操作码和操作数来填充被破坏的指令队列

在第一步中，CPU 寄存器是被自动入栈的，顺序为：PCL，PCH，X，A，CCR——H 没有入栈。

基本上每个外设都拥有至少一个中断源；IRQ 用来接受外部事件的中断请求；SWI 指令引起 CPU 的软件中断（一般用于调试）。

——前面我们提到，中断向量表可以被重定位。这是为了 bootloader 实现方便而增加的一个特别有用的功能。

> 注意：如果 IVT Redirection 功能被打开，应用程序一定要重新 MAP 中断向量表，否则，中断的产生将使应用程序崩溃。

1.2. 数据对齐

在一个硬件系统中，除非全部使用单字节的 unsigned char 类型的数据，否则会遇到数据存储单元对齐的问题，特别是当不同结构的 CPU 之间通信的时候，针对数据对齐的考虑变得不可避免。在 Intel 的平台上，一般采用所谓的低字节在后（Little Endian）的存储方式，即数据的最低字节数据存储在低地址中；而非 Intel 平台则大多采用高字节在后（Big Endian）的存储方式，即高字节数据存放在低地址中。在 CodeWarrior 提供的 C08 编译器中，默认的数据对齐顺序也是 Big Endian 的。考虑下面的测试程序，这段程序定义了一个 long 型的变量（4 bytes），然后顺序单独取出这 4 bytes 赋值给一个 unsigned char 型的变量。

```
void main(void) {
            unsigned char temp;
            long x = 0x12345678;
            long *px = &x;

            temp = (unsigned char) *((unsigned char *)px + 0);
            temp = (unsigned char) *((unsigned char *)px + 1);
            temp = (unsigned char) *((unsigned char *)px + 2);
            temp = (unsigned char) *((unsigned char *)px + 3);
#ifdef __HCS08__
  EnableInterrupts; /* enable interrupts */
  /* include your code here */

  for(;;) {
    __RESET_WATCHDOG(); /* feeds the dog */
  } /* loop forever */
  /* please make sure that you never leave main */
#endif
}
```

这些代码可以在 HCS08 上编译和运行，也可以在 PC 上编译和运行。在 HCS08 上，4 次给 temp 赋值的结果分别为：0x12，0x34，0x56，0x78；而在 PC 上，4 次赋值的结果分别为：0x78，0x56，0x34，0x12。

假如不考虑数据分配的定义，PC 通过串行口分 4 次（每次只能发送 1byte 数据）把 x 发送给一个 HCS08 微控制器，那么微控制器接收到的数字并不是

0x12345678，**而是** 0x78563412，**程序员必须在软件中把这些接收到的数据的顺序调整过来，重新组合成一个正确的值，或者在发送的时候就按照接收者的顺序发送。**

1.3. HCS08 的存储器模型：Tiny 和 Small

08 C 编译器提供两种存储器模式给程序员选择：Tiny 和 Small。在 Tiny 模式下，可供直接使用的 RAM 空间为地址 0x0100 以前的单元（零页或者直接寻址页），对于 AW60，可直接使用的 RAM 地址范围为 0x0070 到 0x00FF，栈和全局变量、静态变量都映射在这部分空间中。虽然 Tiny 模式提供了较快的存取速度和较短的指令长度，但是可供直接使用的 RAM 空间太少，如果要在零页以外的地址空间安排变量，则必须使用#pragma 通知链接器。我们创建一个使用 Tiny 模式的工程，首先来看它的 PRM 文件。

```
SEGMENTS
 /* Here all RAM/ROM areas of the device are listed. Used in PLACEMENT below. */
   ROM               = READ_ONLY    0x1860 TO 0xFFAF;
   Z_RAM             = READ_WRITE   0x0070 TO 0x00FF;
   RAM               = READ_WRITE   0x0100 TO 0x086F;
   ROM1              = READ_ONLY    0x0870 TO 0x17FF;
   ROM2              = READ_ONLY    0xFFC0 TO 0xFFCB;
END

PLACEMENT
/* Here all predefined and user segments are placed into the SEGMENTS defined above. */
   FAR_RAM                                    INTO  RAM;
   DEFAULT_ROM, ROM_VAR, STRINGS      INTO  ROM;
   /* ROM1,ROM2 In case you want to use ROM1,ROM2 as well, be sure the option -OnB=b is
passed to the compiler. */
   _DATA_ZEROPAGE, MY_ZEROPAGE, DEFAULT_RAM        INTO  Z_RAM;
END

STACKSIZE 0x50

VECTOR 0 _Startup /* Reset vector: this is the default entry point for an application. */
```

在 PLACEMENT 段里边，我们可以看到，默认的数据存储空间是 Z_RAM，即地址范围 0x0070 到 0x00FF 之间，有 144bytes。而地址范围 0x0100 到 0x086F 之间的 1904bytes 被定义为 FAR_RAM。在声明全局变量的时候，如果不做特别指定，链接器只在默认的 Z_RAM 中分配它们。另外，栈也被分配在 Z_RAM 中，有 80bytes 被占用了，可供分配全局变量和静态变量的 RAM 空间实际上只剩下了 64bytes。Startup 代码通过宏 INIT_SP_FROM_STARTUP_DESC()来初始化栈，这个宏在文件 hidef.h 中被定义为下面汇编语句的集合：

```
  __asm LDHX @__SEG_END_SSTACK; __asm TXS;
```

这个宏取出__SEG_END_SSTACK，并把它赋值给 SP。__SEG_END_SSTACK 是由链接器定义的对象。从 map 文件中可以找到它的值，被定义为 0x100：

```
OBJECT-ALLOCATION SECTION
   Name            Module            Addr   hSize   dSize    Ref    Section   RLIB
-------------------------------------------------------------------------------------
- LABELS:
   __SEG_END_SSTACK          100      0       0       1
```

我们在 main.c 中写下下面的测试代码：

```
unsigned char auto_data[0x40];        /* auto_data[] will be allocated in the default Z_RAM*/
#pragma DATA_SEG FAR_RAM   /* specify the allocation segment FAR_RAM */
unsigned char data1;
unsigned char data2;
#pragma DATA_SEG DEFAULT    /*specify the default data segment Z_RAM*/

void main(void) {
//  static unsigned char data3;    /* static variable will also be allocated in Z_RAM   /test 1 */
  data1 = auto_data[0];
  data2 = auto_data[1];
//  data3 = 0xff;                                                                    // test 2

  EnableInterrupts; /* enable interrupts */
  /* include your code here */

  for(;;) {
    __RESET_WATCHDOG(); /* feeds the dog */
  } /* loop forever */
  /* please make sure that you never leave main */
}
```

全局变量 auto_data[]**（一个数组）会被分配在默认的** RAM **中，即** Z_RAM **；** #pragma **语句则告诉链接器** data1 **和** data2 **应该分配在** FAR_RAM **中，因为** auto_data[]**已经占用了仅剩的** 64bytes **（另外** 80bytes **被栈占用）。如果去掉程序中的**#pragma **语句，或者去掉语句** test 1 **和** test 2 **的注释（定义了一个静态变量** data3**），或者使数据** auto_data **包含的元素数目大于** 0x40**，都会导致一个链接期错误：**

L1102: Out of allocation space in segment Z_RAM at address 0xb1

我们再来创建一个使用 Small **模式的工程，并观察它的** prm **文件。**

```
SEGMENTS
 /* Here all RAM/ROM areas of the device are listed. Used in PLACEMENT below. */
    ROM               = READ_ONLY    0x1860 TO 0xFFAF;
    Z_RAM             = READ_WRITE   0x0070 TO 0x00FF;
    RAM               = READ_WRITE   0x0100 TO 0x086F;
    ROM1              = READ_ONLY    0x0870 TO 0x17FF;
    ROM2              = READ_ONLY    0xFFC0 TO 0xFFCB;
END

PLACEMENT
/* Here all predefined and user segments are placed into the SEGMENTS defined above. */
    DEFAULT_RAM                           INTO  RAM;
    DEFAULT_ROM, ROM_VAR, STRINGS         INTO  ROM;
    /* ROM1,ROM2 In case you want to use ROM1,ROM2 as well, be sure the option -OnB=b is
passed to the compiler. */
    _DATA_ZEROPAGE, MY_ZEROPAGE,          INTO  Z_RAM;
END

STACKSIZE 0x50

VECTOR 0 _Startup /* Reset vector: this is the default entry point for an application. */
```

和 Tiny **模式相比，区别是** DEAFAULT_RAM **段被放置在** RAM **中，而没有定义** FAR_RAM **段。在这种模式下，链接器会在整个可用的** RAM **空间中自由安排**

全局变量和静态变量，并且栈也不会被放在 Z_RAM 中。在 map 文件中，我们发现__SEG_END_SSTACK 被定义为 0x150，使得栈被分配在 0x0100 到 0x0150 之间的空间中。

```
OBJECT-ALLOCATION SECTION
   Name            Module              Addr   hSize   dSize    Ref    Section   RLIB
-------------------------------------------------------------------------------------
- LABELS:
   __SEG_END_SSTACK           150      0       0       1
```

在 small 模式下，默认的变量不会被分配在零页。如果为了提高程序的效率要把一个变量指定到零页，应该使用下面的#pragma 语句：

```
#pragma DATA_SEG MY_ZEROPAGE
unsigned char dir_data1;     /* dir_data1 will be allocated in Z_RAM */
#pragma DATA_SEG DEFAULT
unsigned char auto_data2;   /* auto_data2 will be allocated in DEFAULT_RAM */
```

从上面的分析可以看出，虽然 tiny 模式使链接器在分配变量时把所有未经特别指定的变量全都映射到零页而提高代码效率，但是零页的空间毕竟有限。而 small 模式会把变量分配到 0x0100 以后的地址，使得默认的 RAM 空间更大。两种模式下，都可以通过使用适当的#pragma 来指定变量分配的段。在编程实践中，可以根据变量的多少和优化的需要酌情使用这两种存储器模式。

还有一点区别特别值得注意：在 Tiny 模式下，指针的长度默认为 8bit。那么假如程序员声明了一个指针变量，并且试图通过它来访问地址 0x0100 以后的存储器单元，一定会得到错误的结果。为了处理这种情况，必须用关键字__far 修饰该指针变量：

```
unsigned char * __far ptr;
```

在 Small 模式下，则因为指针长度默认为 16bit，所以不用特别声明。不过为了提高访问效率，可以用关键字__near 修饰一个指针变量，使其长度为 8bit，来访问地址 0x0100 以前的存储器单元。

```
unsigned char * __near ptr;
```

1.4. 堆（heap segment）

回忆我们在大学里开始学习 C 语言的时候，老师和学长们总在说的难点，仿佛世界上最可怕的事情一样，就是 C 语言中的指针的使用。错误地使用指针确实会给导致系统崩溃，但是什么情况下这种危险才会出现呢？

动态内存分配技术给合理利用有限的资源提供了良好的解决方案，但这也导致上述的潜在危险。我们先来看一段代码：

```
#include <hidef.h>          /* for EnableInterrupts macro */
#include "derivative.h"     /* include peripheral declarations */
#include <stdlib.h>         /* malloc() and free()*/

void main(void) {
 unsigned char *ptrTest = NULL;   // 定义一个指针。定义之初它并不指向任何有意义的内存
 unsigned char i;

 i = *ptrTest;              // (1) 在指针被初始化之前通过该指针访问内存
 pTest = malloc(20);        // 为指针 pTest 分配 20bytes 的内存

i = *ptrTest;               // (2) 指针初始化，但是它指向的内存单元还没有被初始化
 for (i = 0; i < 20; i++) {
  *(ptrTest++) = i;         // (3) 初始化这段内存
 }

 i = *ptrTest;              // (4) 因为++操作符的影响，此时 pTest 已经指向有效内存范围之外
 ptrTest--;                 // 使 pTest 回到有效内存的最后一个单元
 i = *ptrTest;              // (5) 访问有效内存的最后一个单元

 free(ptrTest);             // 释放内存
 i = *ptrTest;              // (6) 此时 pTest 指向的单元已经没有意义，成了所谓“野指针”

 for(;;) {
  __RESET_WATCHDOG(); /* feeds the dog */
 } /* loop forever */
 /* please make sure that you never leave main */
}
```

在这段代码中，首先声明了一个指向 unsigned char 类型的指针变量 ptrTest。在定义之初，pTest 并不指向任何有意义的内存单元。在 CW5.1 中编译上述代码不会产生任何的错误或者警告信息。在声明这个指针变量的时候，我把它初始化为一个空指针（赋值为 NULL），这样做可以避免编译器产生的一个警告：“C5651: Local variable ‘ptrTest’ may be not initialized”。在这些代码中，我在 6 个不同的位置试图通过 ptrTest 去访问内存，下面我们分别来分析这 7 处内存访问的情况：

(1) ptrTest 还没有被初始化，通过它指向的内存单元没有意义；

(2) ptrTest 已经被初始化，运行期库（run-time library）函数 malloc()在数据堆中分配出 20bytes 的空间，ptrTest 指向这段空间。但此时这 20bytes 的内存单元都还没有被初始化，通过 ptrTest 读取到的仍然是没有意义的数据；

(3) 将 ptrTest 指向的 20bytes 内存单元初始化；

(4) 由于循环体中++操作符的影响，退出循环时，pTest 指向了分配内存之外的一个存储单元，这个单元的数据也是没有意义的；

(5) 将 ptrTest 减去 1 后，它指向分配内存的最后一个单元；

(6) 库函数 free()将 ptrTest 指向的内存释放，这段内存从此变为一段自由的空间，虽然先前给这些存储单元赋的值并没有被摧毁，下次调用 malloc()可能会在这段空间中重新分配。空间被释放后，pTest 的值也还没有改变，成了一个所谓的“野指针”，通过它访问内存得到没有意义的数据。

在上述 6 种情况下，只有(3)和(5)的操作是正确的。其它的操作都访问了没有意义的内存单元，然而无论是在编译期还是运行期都不会报错。在实际的程序中，程序员不光会通过指针 ptrTest 去读取内存，还会有改写的操作，甚至有多个这样的指针变量。程序员必须对自己要做什么，做过些什么，还有什么没有做等各种情况明察秋毫，一丝错误都不能犯，否则他立刻会尝到指针带来的苦果。

> 注意：上面的代码在 CW5.1 的默认设置下无法通过编译，这是因为 CW 指定的 Heap 的默认值为 2000bytes，而很少有器件拥有这么大的 RAM（还要考虑栈和全局变量、静态变量的占用）。为了正确编译，需要在 CW5.1 的安装目录下找出文件 C 标准库设置的头文件 libdefs.h，将其中的宏 LIBDEF_HEAPSIZE 改为一个较小的值，然后打开工程 hc08_lib.mcp，重新编译 C 标准库。——然后再编译上面的代码。
>
> 从这么复杂的处理或许可以看出，08C 编译器的开发者可能根本就不情愿让我们使用 Heap 和动态内存分配技术。

在内存管理的概念中，这样提供给程序在运行期动态分配的内存段叫做堆（Heap）。Heap 实际上类似于一个大数组，它预留了一段空间，malloc()函数在 Heap 中开辟出一段空间给一个指针变量，而 free()则释放该指针变量指向的空间。对于 08C，Heap 实际上就是一段 RAM，它一般会被安排在全局变量和栈的中间。08C 的 Heap 有一些简陋的出错处理机制，但是这些机制实在是太简陋了，我们刚刚看到了它的功能。

厂商们（包括硬件厂商和编译器厂商）都不建议用户在 CPU08 平台上使用 Heap，即动态内存分配技术。幸运的是，用 8bit 微控制器处理的事务更多是和真实世界相关的控制功能，而没有什么庞大的算法，所以我们好像也不需要这种有点令人头疼的技术。

1.5. 栈（stack segment）

在 C 语言中，变量都被映射在数据段。

每一个变量都有其生存周期。在应用程序的整个生命周期都生存的变量，就是那些全局变量，它们是被映射在 DEFAULT DATA_SEG 里边，从最低地址顺序排列（在 Small 模式下，这个地址是 0x0100）。在这一段存储器单元中映射的还有静态变量。不考虑堆的存在，在所有的全局变量和静态变量都安排出各自的空间后，紧接着就是栈的空间。

静态局部变量被 static 关键字修饰，它们一般是一些不希望在调用结束后被消灭的局部变量（静态全局变量呢？）。在为变量安排存储空间单元的时候，静态变量和全局变量享有同样的待遇，它也存在于 DEFAULT DATA_SEG 开头的一些空间里面，而不是在栈中。这样，在一个过程调用中改变了静态变量的

值，调用结束后，这个值仍然保留，在下次调用同一过程时，可以在原来继续获得上次的值。

静态局部变量和全局变量的区别在于其作用域。只有声明它的过程才可以使用它，别的过程则没有这种特权。可以说，静态局部变量是“私有的”。

在一个过程中定义的变量，叫做局部变量，它的生存周期和过程相同。也就是说，当过程被调用的时候，才为这些变量分配内存单元；而当调用结束时，为局部变量分配的内存单元被回收，局部变量也随之消亡。这个过程是依靠栈完成的——所有的局部变量都在栈中映射。

在 C 语言的过程调用中，参数的传递和返回值依靠寄存器和栈完成。在内存映射的过程中，参数和返回值的地位和局部变量相同。

TSX 和 TXS 指令使得 H:X 可以和 SP 交换 16bit 的数据。当应用程序试图访问栈空间而又不想动 SP 的时候，用 H:X 作为指针指向栈空间是一个不错的办法。另外，在程序初始化的时候，对栈指针的初始化必须用 TXS 指令来完成。

```
_Startup:
        LDHX  #__SEG_END_SSTACK  ; 初始化栈指针，__SEG_END_SSTACK 由链接器
        TXS                      ; 定义。把 H:X 的值传送到 SP
        CLI                      ; enable interrupts
```

在上面的代码中，我们看到了如何初始化栈指针。__SEG_END_SSTACK 是由链接器定义的常数，在本例中，这个常数等于 0x150（这是个常数是这么来的：在 small 模式下，所有的全局变量都从地址 0x100 起。全局变量顺序安排完成之后——在本例中，一共有 0 个全局变量——下一个可用的地址加上 STACK SIZE 得到的地址被指定为栈底）。执行完第一条指令后，H:X 的值为 0x150，第二条指令把 H:X 的值传送到 SP 中去，执行完成后，SP 的值为 0x14F。我们可以发现，当一个指针值从 H:X 传送到 SP 的时候，这个值被减了 1。反之，把 SP 传送到 X 的时候，传送的值会被自动加上 1，而 SP 本身的值不被改变。这是因为，SP 永远指向下一个可用的内存单元，而索引寄存器应该直接指向目标单元。

> 注意：__SEG_END_SSTACK 对于不同的程序实例并不是一个确定的常数，它是动态的。连接器会首先分配全局变量和静态变量，然后在数据段之后紧接着分配栈。例如在 Small 模式下，DEFAULT_RAM 自地址 0x0100 开始，如果一共有 5bytes 的全局变量和静态变量，STACKSIZE 为 0x50，则__SEG_START_SSTACK 为 0x105，而__SEG_END_SSTACK 为 0x156。

16bit 字长的 SP 用来维护栈。理论上，SP 可以指向 HCS08 寻址空间的任何单元，但实际上栈只能被安置在 RAM 空间中，栈空间的大小可以是整个 RAM 空间的大小。在使用 BSR 或者 JSR 调用子程序（routine）的时候，下一条顺序指令的地址（返回地址）会自动保存在栈中，子程序返回时（RTS），返回地址被取出，自动装载在 PC 中。

```
                    ORG $F000
                _my_init:
F000 AD 02          BSR sub_dummy       ; 调用子程序 sub_dummy
                return_address:         ; 从调用返回的地址，0xF002
F002 A6 55          LDA #$55
                    ; ......
                sub_dummy:              ; 子程序入口地址，0xF004
F004 9D             NOP                 ; 进入后，0xF002 被压入栈中
F005 81             RTS                 ; 从子程序返回，0xF002 装载到 PC 中
```

正如前面所言，SP 永远指向下一个可用的内存单元。栈在物理上是 RAM 的一部分，在系统初始化的时候，应该定空间，即指定 SP 的初始值和栈的大小。在前面我们看到如何初始化栈指针的例子，栈空间大小则应该在链接器参数文件（PRM）中定义：

```
STACKIZE    0x50
```

那么栈被指定为从地址 0x14F 起，大小为 0x50 个 byte 的一段专有存储空间。当执行入栈指令时，入栈的值被保存在 SP 当前指向的单元中，然后 SP 自动减 1（从而指向下一个可用单元）。出栈指令则先把 SP 加 1，使其指向最后一个有效数据的保存地址，然后把这个数据弹出给目标地址。实际上，数据出栈后，原来的地址单元中的数据仍然存在，直到下次入栈的时候会被新数据覆盖。所以，栈空间是从高地址到低地址“向上”增长的。这有点像一个单独的汉诺塔，最后放上去的盘子只能最先被取下来。

为了和 HC05 兼容，CPU08 在复位后被设置为 0x00FF——HC05 的 RAM 只有 256bytes，栈从它的 RAM 的最高地址 0x00FF 开始“向上”生长。栈指针复位指令 RSP 也使 SP 被赋值为 0x00FF。对于 CPU08，这实际上是一个不合理的值，因为从 0x0000 到 0x00FF 的空间，是宝贵的“零页”区域，在这个空间进行寻址只需要一个 byte 就可以表示操作数的地址，即所谓的直接寻址（DIR），DIR 寻址模式可以节省一个字节的指令长度和一个指令周期的执行时间。并且，在直接页面，位操作也可以直接进行。因此，用户程序应该重新初始化 SP，一般来说，应该使 SP 指向 RAM 的最后一个（最高）地址。

> 注意：尽量不要使用 RSP 指令。这个指令会干扰 CPU 对栈的自动维护。并且，它只把 SP 的低 8bit 设置为 0xFF，而不管其高 8bit。

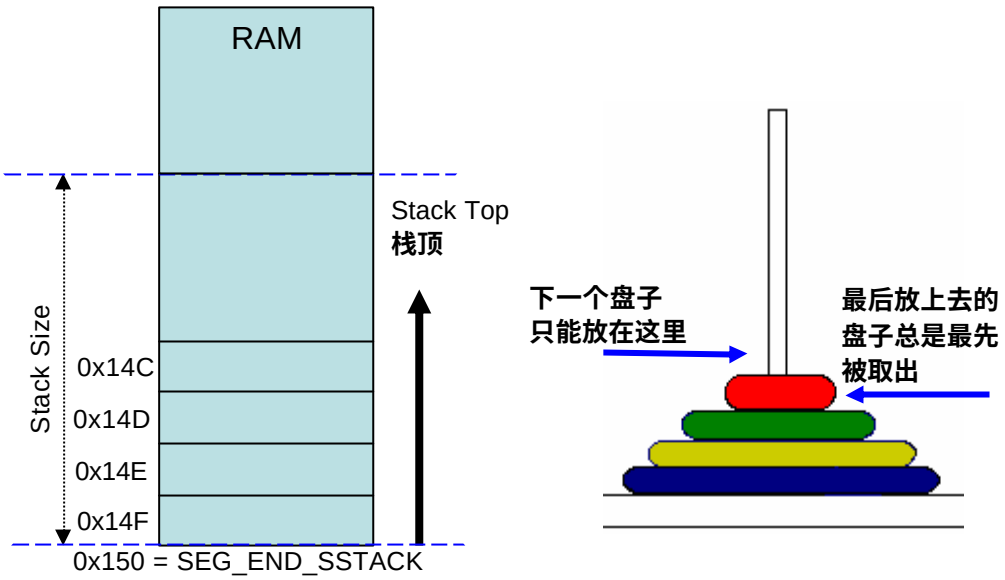

图 2. 栈（Stack）和汉诺塔