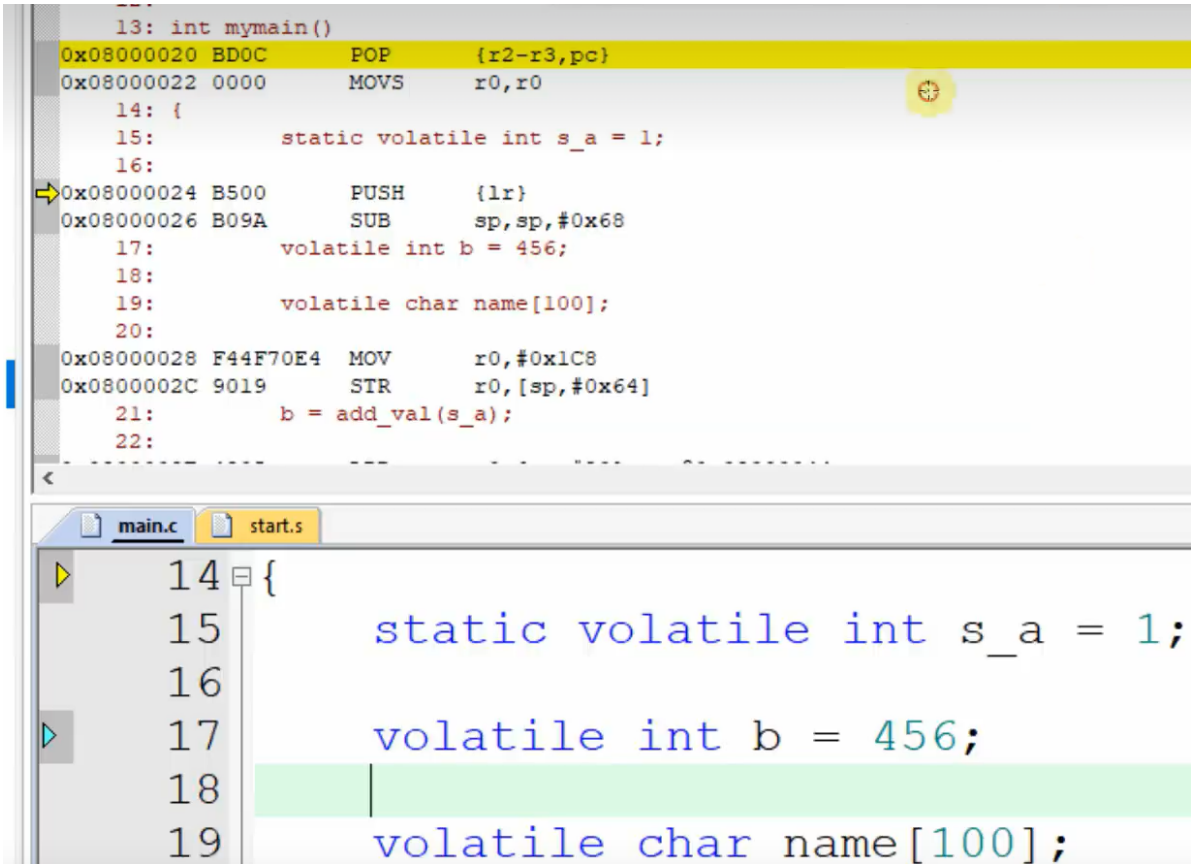
C语言的本质（基于arm深入分析）

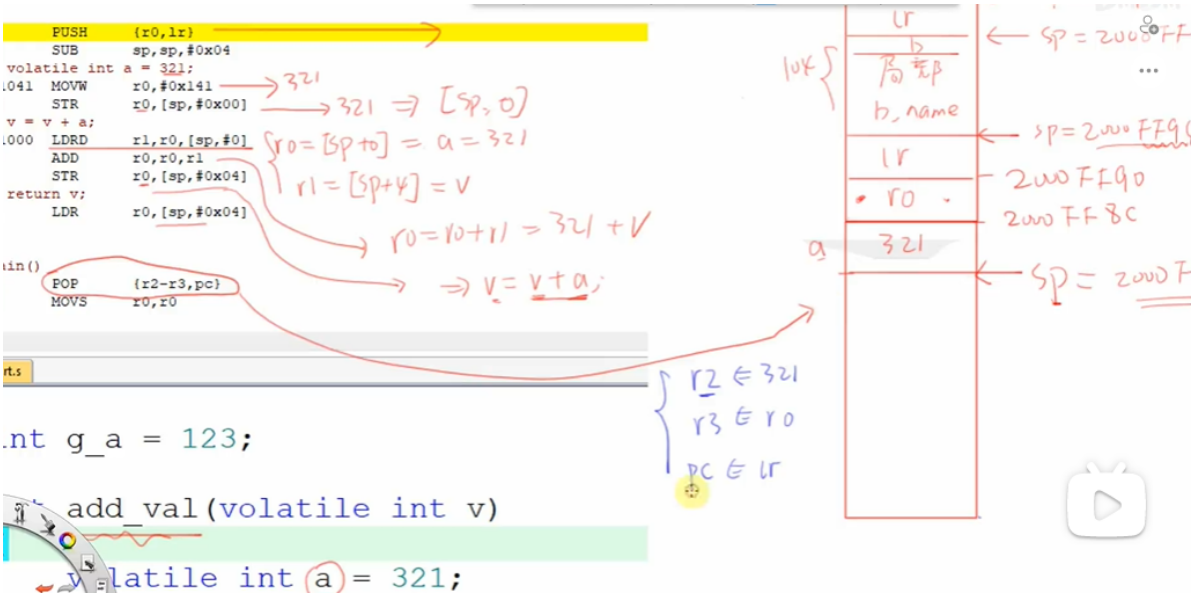
C语言的本质（基于arm深入分析）

1、架构与汇编简明

`push {r3 lr}` 保存lr(返回地址)，r3占坑，放变量



cpu先存入返回地址（因为声明后就调用函数，为避免lr寄存器的值被覆盖，先存入栈中），后面根据设置的变量空间，提前将sp移到足够变量空间之后的位置。



出栈时，`pop`低位的先出，a的值`pop`给r2，ro的值`pop`给r3 lr的值给pc寄存器，同时每`pop`一位，sp上移一位。

常见栈：`push`先移动sp指针再存 pop 先压出，再移动sp指针

编译后的程序下载到flash上，程序开始之后，把有初始值全局变量统一复制到ram上变量所处于的地址上，静态变量和全局变量一样被复制过去，只是别的函数或文件无法访问无初始值的被统一初始化为0，

初始值为0/无初始值的全局变量或静态变量 被类似`memset`(将指针变量 s 所指向的前 n 字节的内存单元用一个“整数” c 替换)统一置0

2、变量

变量，可以变，可读可写，都存在内存中，指针（地址）在32位处理器中都是4字节

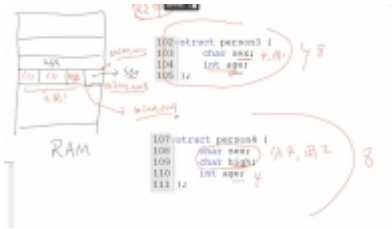
只读的常量放在flash中，加const放于ram中，节省内存

3、结构体

struct类型声明不占空间，声明变量占空间

Struct 里面定义的变量类型和非4的倍数，在内存中会优化到占4个字节

对齐效率高



4、变量赋值

变量的类型是多大空间，一次赋值就会赋值多大空间

5、sizeof和关键字

- sizeof

  `Char *p` `Int *p2` `Sizeof(*p)  = sizeof(int)` `Sizeof(p) = 32`

- Volatile

  防止编译器优化（只在cpu中改变），必须要把值写入内存中用于必须要读的寄存器

- Extern

  可以在文件中直接写，也可以在包含的头文件里面写，Extern建议放在.h文件中声明

  不建议使用全局变量，要的话用static，可以做一个函数接口 get_x()让别人获得变量的值

- vTypedef （类比 #define,define是宏定义，直接替换），typedef是类型定义 可以创建类型别名，可以让使用更方便。

```
typedef int A;
typedef int * B;
typedef struct lcd_operation C;
typedef struct lcd_operation * D;
typedef int (*add_type)(int, int);
```

A、B、C、D都是类型别名，add_type同理

```
add_type f1;
add_type f2;
```

6、函数

- 就是一系列的指令，是一些的机器码
- 调用函数：让cpu的pc寄存器等于一系列机器码的首地址，就是函数地址
- 传入实参只是把（变量的）数值赋给R0，让子函数修改调用者的变量得传递地址

7、指针

只记录首地址，都是4byte（32位），char *p ；p是四字节指针变量，*p是2字节的char型，赋值给*p赋值2字节

定义变量初始化，都是先确认地址，再往地址赋值，直接初始化和指针初始化在汇编无差别

7.1通过指针赋值

```
Struct person {
    char a;
    char b
    int c;
}
struct person wei = {“1”, ”2”, 1};
struct person *pt;
*pt = wei // 把wei的值全部赋给\*pt
            //*pt 也是一个person结构体类型；(故并非传地址)
            //pt是指针类型
```

7.2结构体指针

结构体声明中，不可以放自己的结构体，但可以放自己的结构体指针，结构体用.取成员，指针用 ->取成员

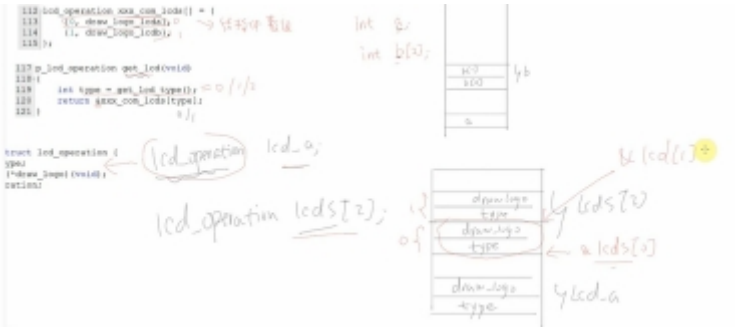
Int (*function)(void)

{

}

定义函数指针，function是变量，占4字节，function和&function都一样 。

get_lcb 取出结构体指针进行传递 4字节，避免了传递结构体，省空间



8、联合体成员

首地址相通，大小取决于最大的成员

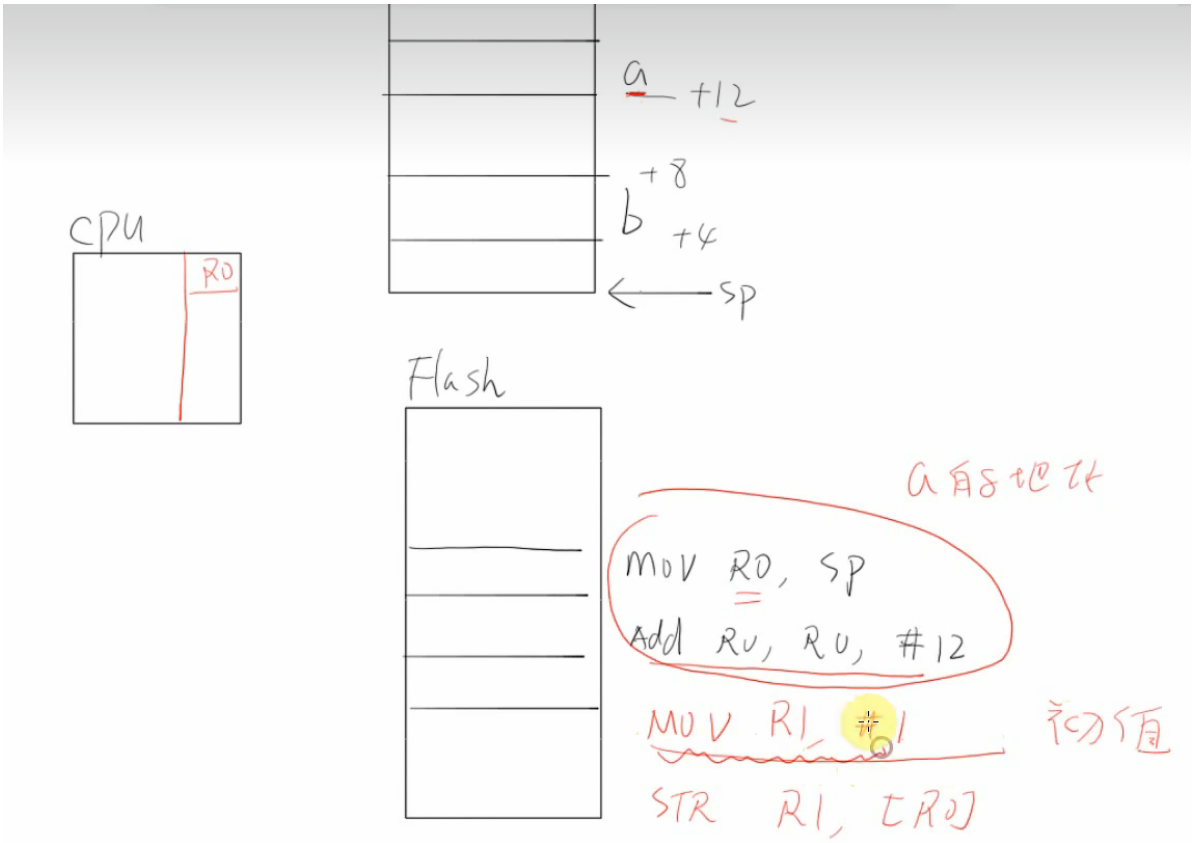
9、头文件的作用（函数声明）

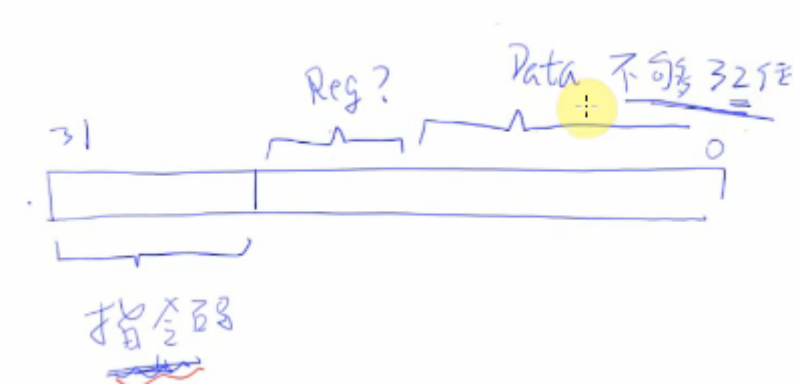
只是告诉编译器，函数怎么使用，用的对不对

10、指针和链表

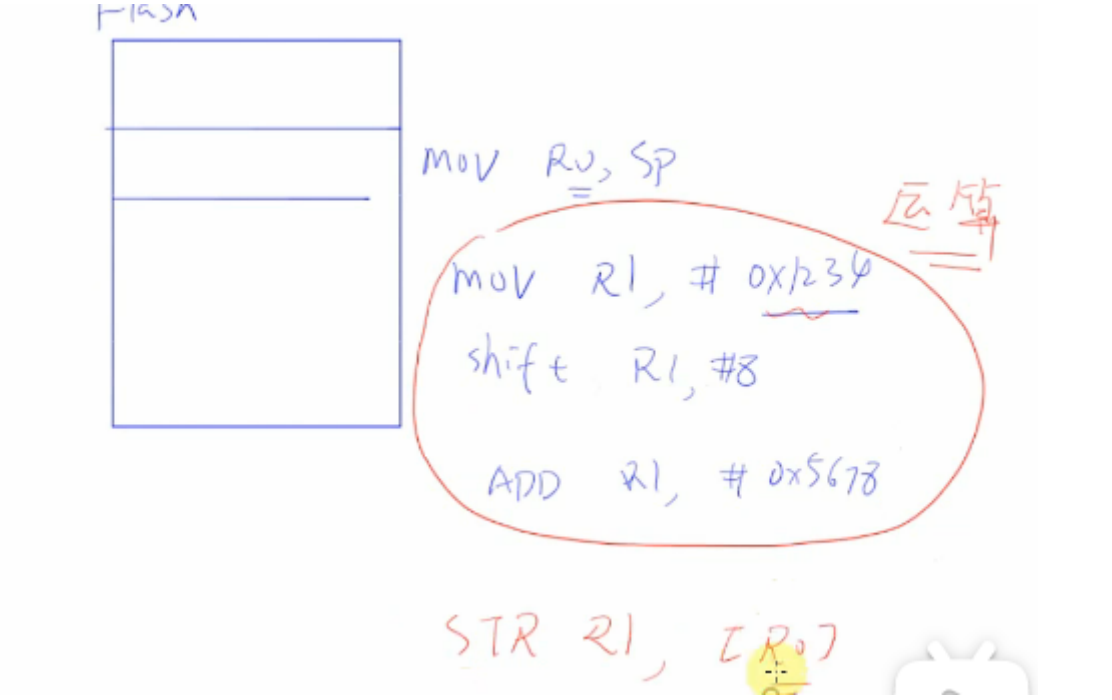
10.1 int变量的初始化（局部）

```
int a = 1;          //简单的数把值内嵌在指令中

MOV R0, SP        //取出SP位置到R0
ADD R0,R0,#12     //令R0等于PC+12（变量地址）
MOV R1, #1        //将a的值存入R1（cpu）
STR R1,[R0]       //将R1（cpu）的值存到地址为R0值的内存上
```

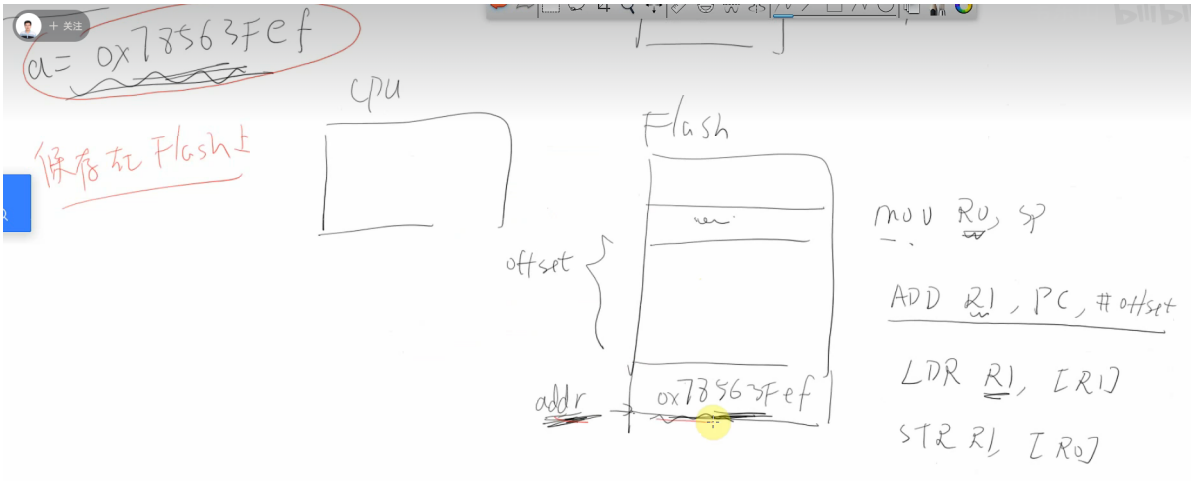





```
int b = 0x123456;    //（机器码一条32位不够放全部数值）复杂的值可以把初始值拆开成几部分，通过运算把他们组合在一起，再写到内存中去

MOV R0,SP      //存地址
ADD R1,#0X1234
SHIFT R1,#8
ADD R1,#0X5678
STR R1,[R0]
```

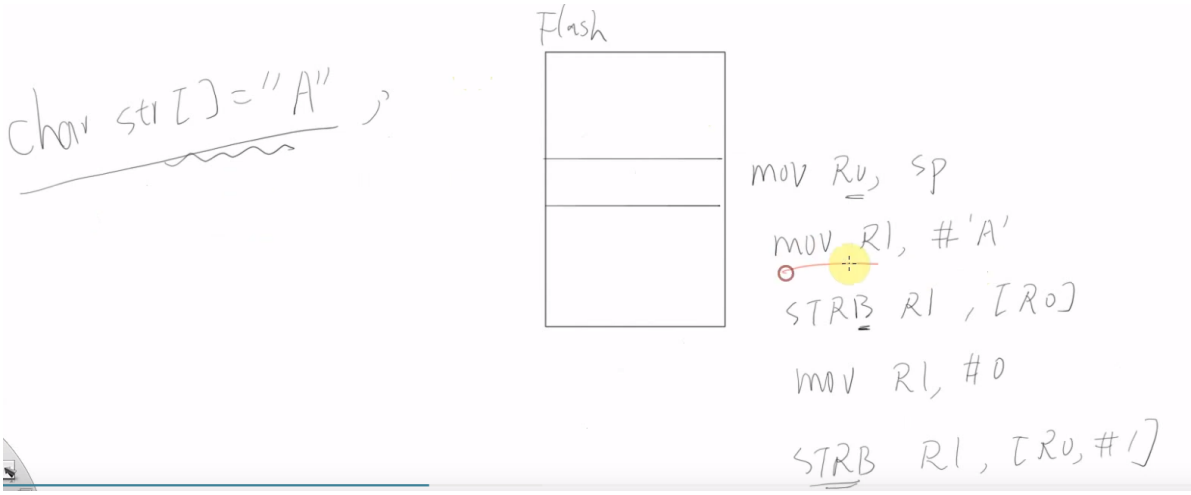



```
int a =78563fef   //更加复杂，无法拆分，初始值存储在flash上

MOV R0,SP
ADD R1,PC #offset
LDR R1,[R1]
STR R1,[R0]
```

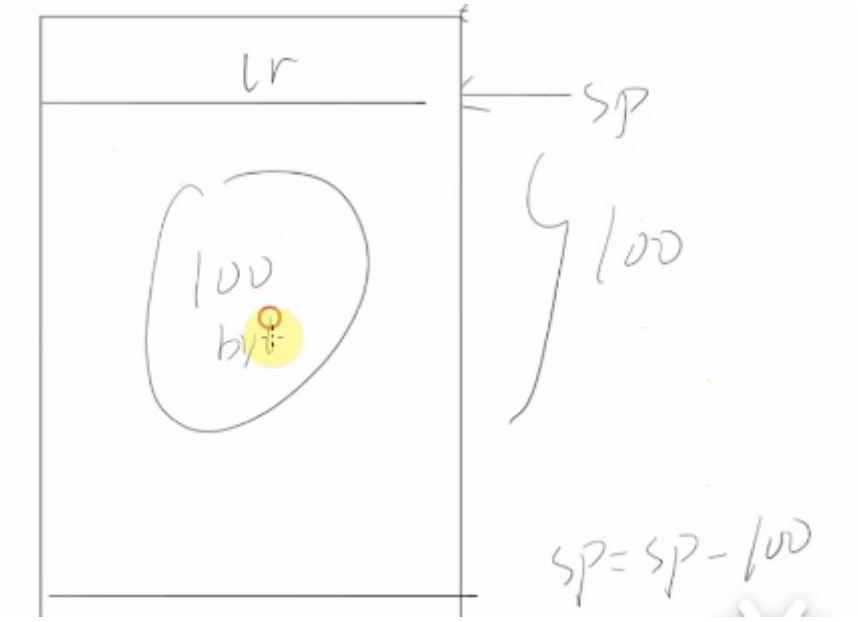


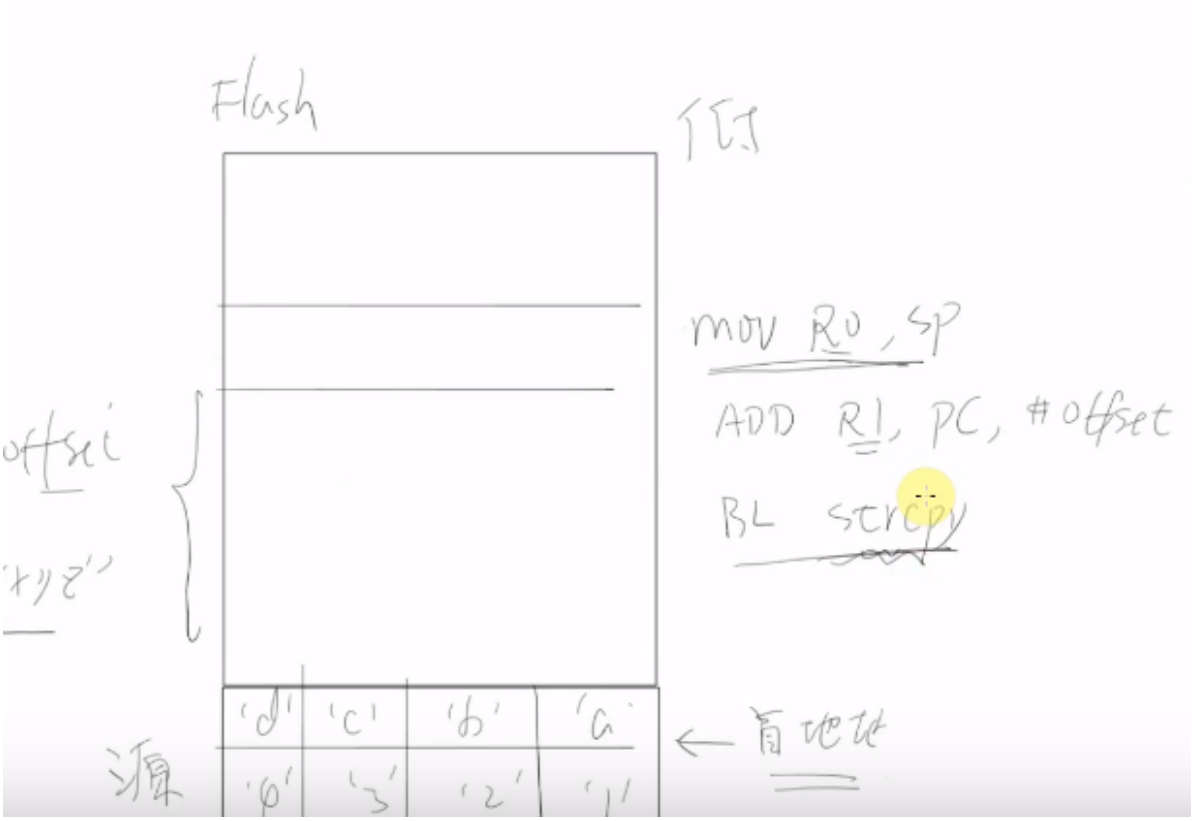
10.2字符串和结构体初始化

```
char str[]= "A";  //简单数值内嵌在指令中。分成两部分先定义数组，栈里面留出位置；再初始化
str[0]= 'A', str[1]= '0';

MOV  R0, SP·
MOV  R1, #'A'
STRB R1, [R0]   //STRB 存字节
MOV  R1, #0
STRB R1, [R0,#1]


```

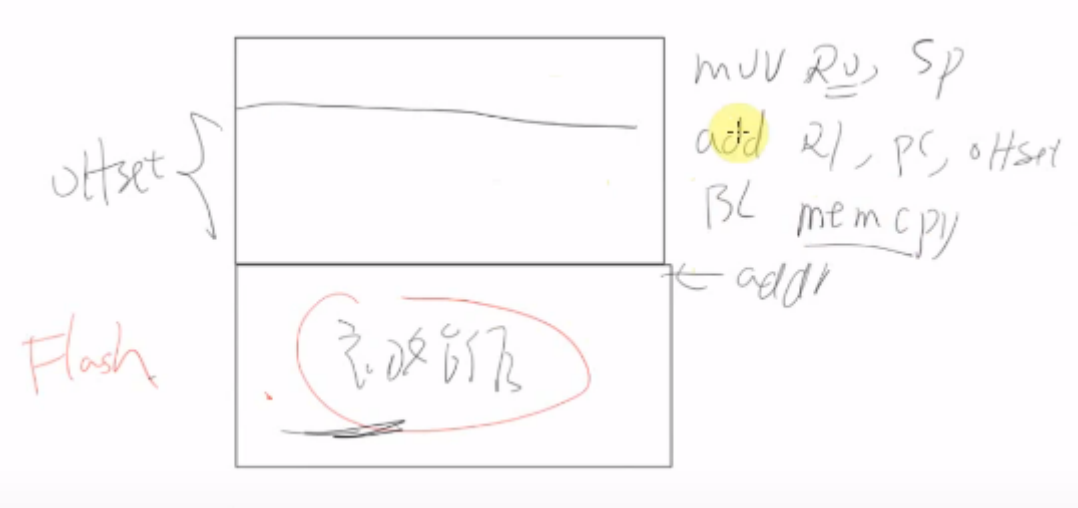



```
char str[100] = "abcde216s641d61sd";    //拆分成两条：先定义数组；再初始化数值

MOV  R0,SP
ADD  R1,PC,#offset
BL strcpy    //字符串操作用strcpy
```

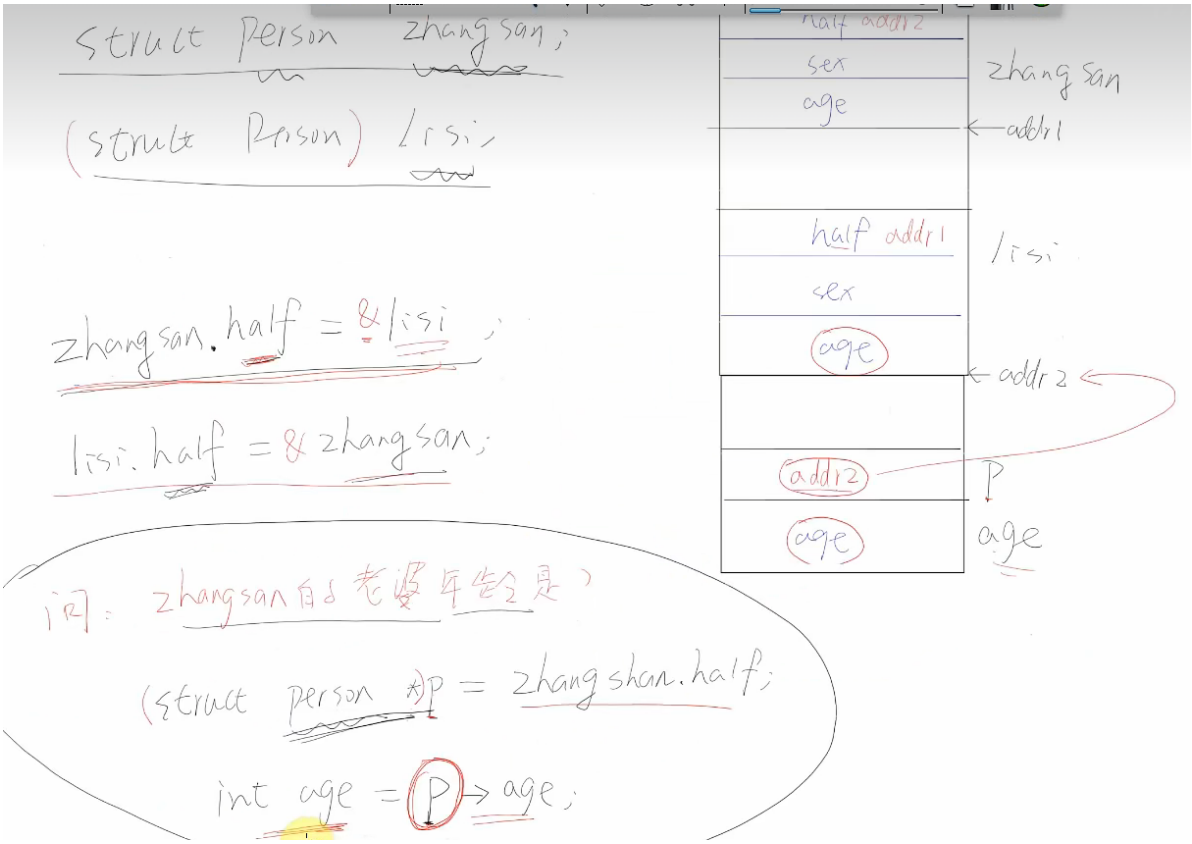




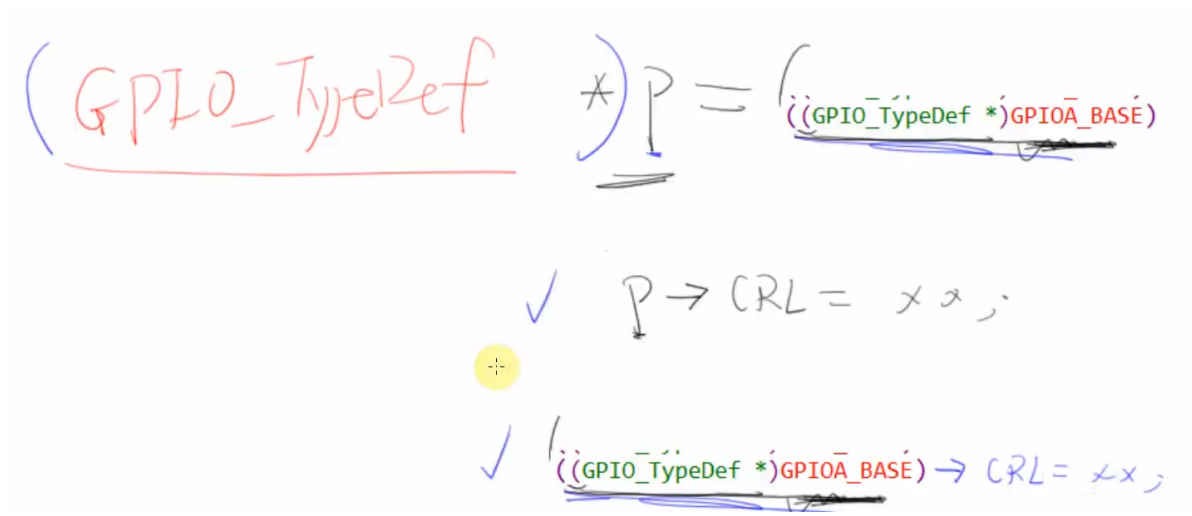

```
struct dog wc = {1,1,1,1,1,1.....}; //通过函数初始化，一样先定义在栈中分配空间，再初始化

MOV R0,SP
ADD R1,PC,offset
BL memcpy       //内存操作用memcpy
```



10.3指针访问结构体



10.4指针访问硬件



10.5链表

链表的实质就是指针，可以存放下一个元素的地址,最原始的链表就是只存下一个的地址，有实际意义的链表成员里面会有别的信息

```
struct list{
    char *name;
    struct person *next;
}

struct person{
    char name;
    char age;
    struct person *next;
}p1
```

```c
void InitList(struct list *pList, char *name)
{
    pList->name = name;
    pList->next = NULL;
}
void AddItemTolist(struct list *pList,struct person *new_persion)
{
    struct person *last
    if(pList->next = NULL)
    {
        last->next= new_persion;
        new_persion->next= NULL;
        return;
    }
    last = pList->next;
    while(last!= NULL)
    {
        last =a_list->next;
    }
    last->next= new_persion;
    new_persion->next= NULL;
}
void DelItemFromList(struct list *pList, struct person *person)  //链表的删除
{
    struct person *pre = NULL;
    struct person *p = pList->next;
    /* 找到person */
    while (p != NULL && p!=person)
    {
        pre = p;
        p=p->next;
    }
    /*退出条件 p== NULL,p==person*/
    if(p == NULL)
    {
        printf("cannot find.\r\n");
        return;
    }
    if(pre == NULL)
    {
        pList->next = p->next;
    }
    else
    {
        pre->next= p->next;
    }


}
int main()
{
    struct list a_list;
    int i;
    InitList(&a_list, "A_class"); //链表的创建
    i = 0;
```

```
    while(p[i].name != NULL)
    {
        AddItemTolist(&a_list,&p1); //添加链表
        i++;
    }

    PrintList(a_list);
}

/****************修改*****************/
struct list{
    char *name;
    struct person head;
}

struct person{
    char name;
    char age;
    struct person *next;
}p1
void InitList(struct list *pList, char *name)
{
    pList->name = name;
    pList->head->next = NULL;
}
void AddItemTolist(struct list *pList,struct person *new_persion)
{
    struct person *last= &pList->head;
    while(last->next!= NULL)
    {
        last =a_list->next;
    }
    last->next= new_persion;
    new_persion->next= NULL;
}
void DelItemFromList(struct list *pList, struct person *person)  //链表的删除
{
    struct person *pre = pList->head;
    /* 找到person */
    while (pre!= NULL && p->next!=person)
    {
        pre=p->next;
    }
    /*退出条件 p== NULL,p==person*/
    if(pre == NULL)
    {
        printf("connot find .\r\n");
        return
    }
    else
        pre->next= person-> next;
}
int main()
{
    struct list a_list;
```

```
121     int i;
122     InitList(&a_list, "A_class"); //链表的创建
123     i = 0;
124     while(p[i].name != NULL)
125     {
126         AddItemTolist(&a_list,&p1); //添加链表
127         i++;
128     }
129 
130     PrintList(a_list);
131 }
132 
```

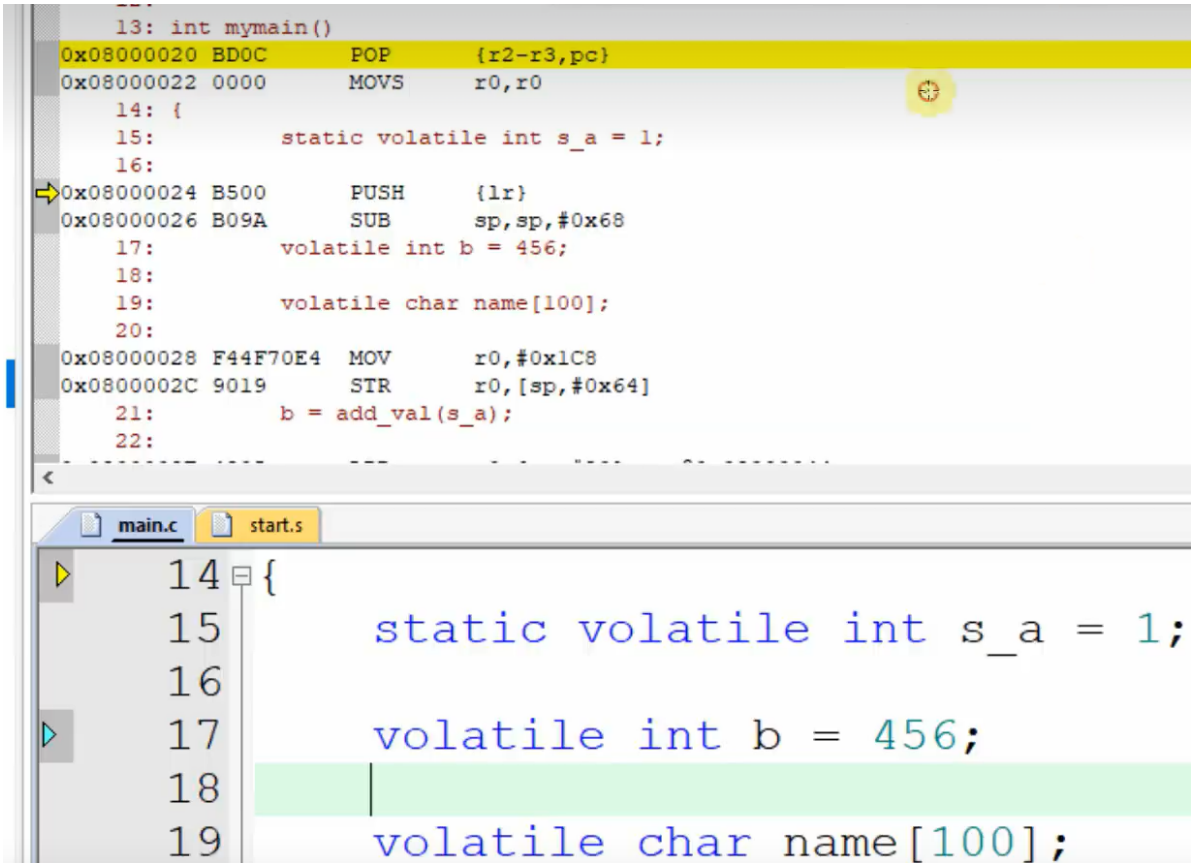
普通链表的缺点：不同的链表，结构体指针类型不同，得重新写一份

改进： 定义指针成员为struct node *next，指向链表下一个成员结构体中的node结构体指针

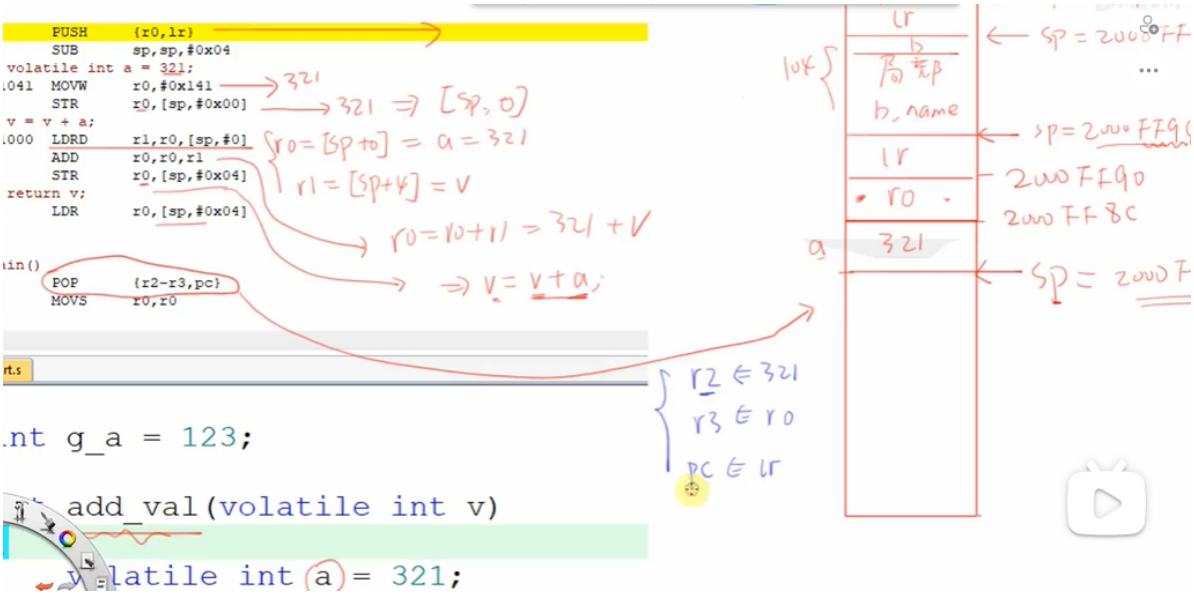
（统一性高）

11、局部变量的分配与初始化

在栈里面`push {r3 lr}`保存lr(返回地址)，r3占坑，然后把局部变量放里面



cpu先存入返回地址（因为声明后就调用函数，为避免lr寄存器的值被覆盖，先存入栈中），后面根据设置的变量空间，提前将sp移到足够变量空间之后的位置。



栈释放局部变量：出栈时，`pop`低位的先出，a的值`pop`给r2，ro的值`pop`给r3 lr的值给pc寄存器，同时每`pop`一位，sp上移一位。

> 常见栈：`push`先移动sp指针再存，`pop`先压出，再移动sp指针

12、全局变量的初始化和空间分配

编译后的程序下载到flash上，程序开始之后，把有初始值全局变量统一复制到ram上变量所处于的地址上，静态变量和全局变量一样被复制过去，只是别的函数或文件无法访问无初始值的被统一初始化为0（这种一整块复制过去的效率比一个一个变量写过去要高，编译之后会在flash中有一个数据段）

初始值为0/无初始值的全局变量或静态变量 被类似`memset`(将指针变量 s 所指向的前 n 字节的内存单元用一个“整数” c 替换)统一置0

13、栈和堆

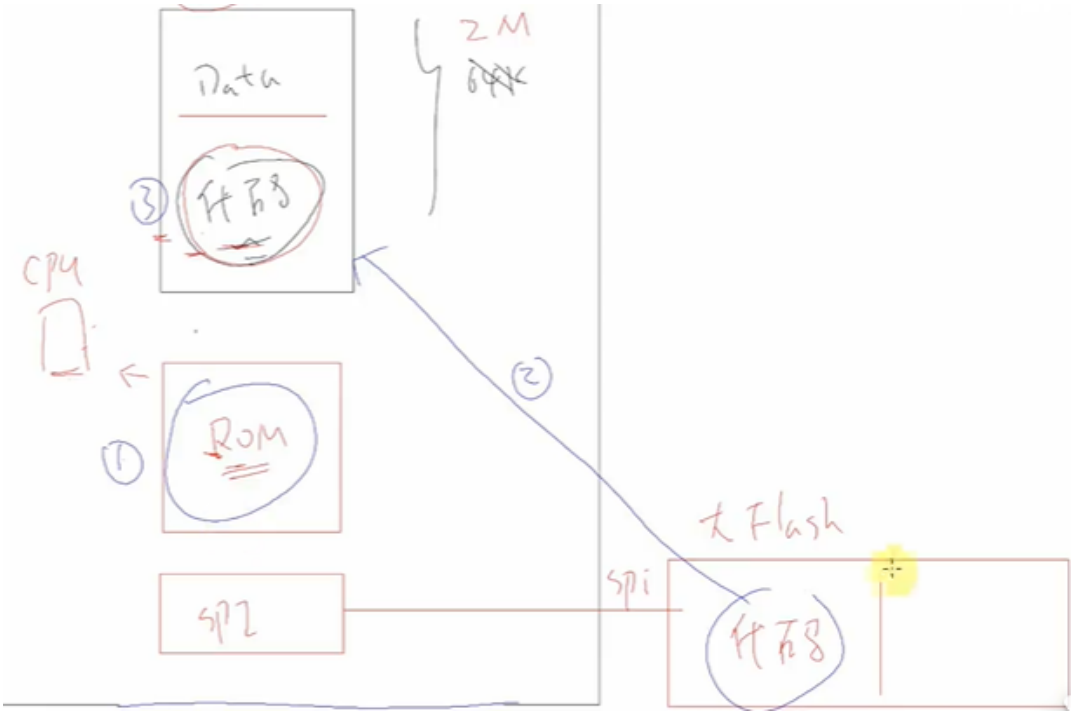
栈是c分配的空闲内存

- 向下增长
- 估计栈大小：寻找使用局部变量最多的调用链
- 选出空闲空间

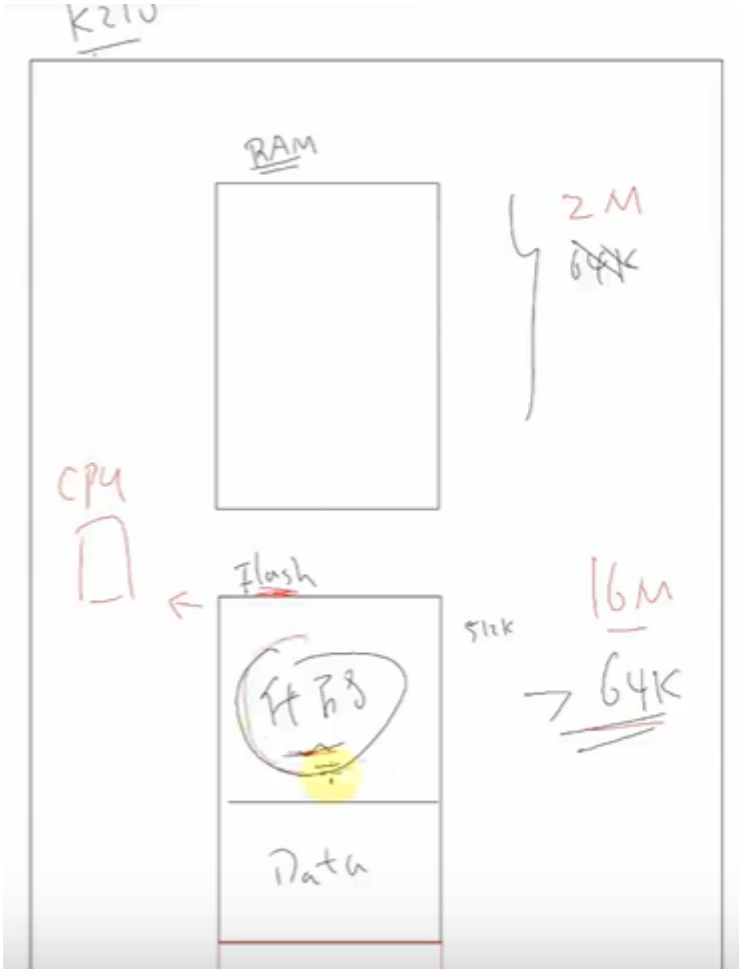
堆是栈之外程序员自己或者他人分配的空闲空间，可以自己控制（栈无法控制）。

14、其余知识点

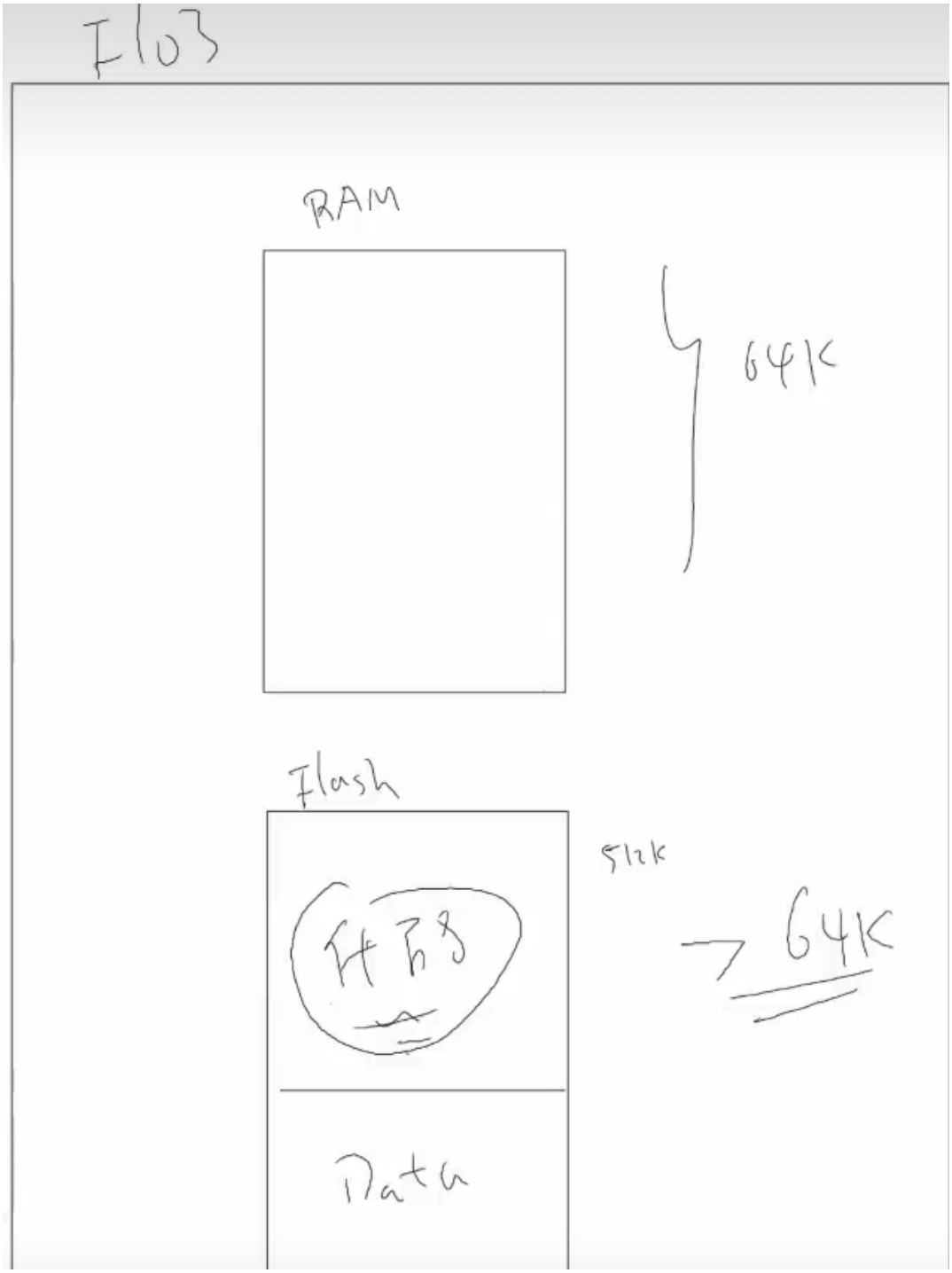
- 全局变量会影响线程安全，保护措施：可以在操作变量时关调度或中断，裸机程序可以放心用，没有太多被打断的可能
- static两个作用1. 只能在本文件使用 2. 不再初始化
- 程序把代码从flash复制到RAM的两种情况：
    - 上电后，运行rom（厂家写的）里面的程序，将代码从外接的FLASH 复制到ram中



- RAM资源比较充足，上电后，程序自己把自己从flash拷贝到RAM中



- RAM资源比较缺乏，不拷贝到RAM中




```
Program Size: Code=5256 RO-data=424 RW-data=48 ZI-data=1832
```

- Code：代码的大小
- RO：常量所占空间
- RW：程序中已经初始化的变量所占空间
- ZI：未初始化的static和全局变量以及堆栈所占的空间

在ARM的集成开发环境中，只读的代码段和常量被称作**RO**段(ReadOnly)；可读写的全局变量和静态变量被称作RW段(ReadWrite)；RW段中要被初始化为零的变量被称为ZI段(ZeroInit)。