

查 IAR 编译信息和程序大小和 XDATA 溢出

一、

1、单片机的存储器分为数据存储器(RAM)和程序存储器(ROM/FLASH):

RAM: 用来存取各种动态的输入输出数据, 中间计算结果以及与外部存储器交换的数据和暂存数据。设备掉电后, 数据就会丢失。

ROM: 通常用来固化存储一些用户写入程序或数据, 用于启动设备和控制设备工作方式。设备掉电后可保存数据。

FLASH: 通常也是用来固化存储一些用户写入程序或数据。设备掉电后不会丢失数据, 同时可以快速读取数据。U 盘、MP3 多用这种存储器。

RAM 包括:

(1)静态 RAM(SRAM): 速度非常快, 是目前读写最快的存储设备。

(2)动态 RAM(DRAM): 保留数据时间短, 但速度比 ROM 快, 计算机内存多为 DRAM。DRAM 包括: EDORAM; DDR RAM; RDRAM; SGRAM; WRAM...

ROM 包括:

(1)PROM: 可编程, 一次性

(2)EPROM: 可擦除可编程, 通过紫外线擦除

(3)EEPROM: 擦除可编程, 通过电子擦除

FLASH 包括: (具体参见

<http://wjf88223.blog.163.com/blog/static/35168001201092685333808/>)

(1)NOR FLASH

(2)NAND FLASH

2、存储类型与存储区关系 (单片机)

data	--->	可寻址片内 ram
bdata	--->	可位寻址的片内 ram
idata	--->	可寻址片内 ram, 允许访问全部内部 ram
pdata	--->	分页寻址片外 ram (MOVX @R0) (256 BYTE/页)
xdata	--->	可寻址片外 ram (64k 地址范围 FFFFH)
code	--->	程序存储区 (64k 地址范围),对应 MOVC @DPTR

3、一个 C 语言程序占用的内存分为以下几个部分:

(1)栈区(stack): 由编译器自动分配释放,存放函数的参数值,局部变量的值等,其操作方式类似于数据结构中的栈。

(2)堆区(heap): 一般由程序员分配释放,若程序员不释放,程序结束时可能由操作系统回收。

(3)全局区(静态区): 全局变量和静态变量的存储位置是在一起的。初始化的全局变量和静态变量在同一块区域,而未初始化的全局变量和未初始化的静态变量在相邻的另一块区域,程序结束后由系统自动释放。

(4)文字常量区: 这一区域用于存放常量字符串,程序结束后由系统释放。

(5)程序代码区: 这一区域用于存放函数体的二进制代码(ROM/FLASH)。

4、堆(heap)和栈(stack)的申请方式

stack: 由系统自动分配。比如,声明在函数中的一个局部变量 `int b`;系统自动在栈中为 `b` 开辟空间。

heap: 需要程序员自己申请,并指明大小。

在 C 中 `malloc` 函数; 如: `p1=(char *)malloc(10);`

在 C++中用 `new` 运算符; 如: `p2=new char[20];`

注: `p1,p2` 本身是在栈中的。

5、例子:

`//main.cpp`

`int a = 0; //全局初始化区`

`char *p1; //全局未初始化区`

`main()`

`{`

`int b; //栈`

`char s[ ] = "abc"; //栈`

`char *p2; //栈`

`char *p3 = "123456"; //123456\0 在常量区, p3 在栈上。`

`static int c =0; //全局（静态）初始化区`

`//分配得来得 10 和 20 字节的区域就在堆区。`

`p1 = (char *)malloc(10);`

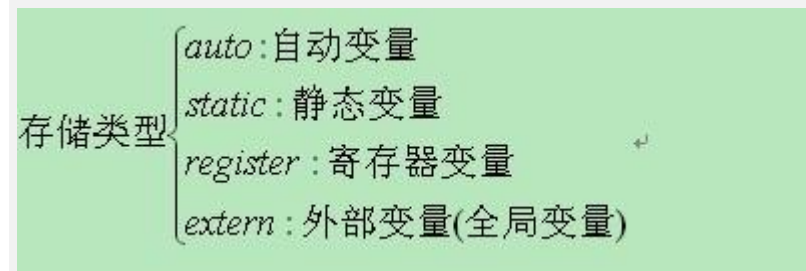
`p2 = (char *)malloc(20);`

`}`

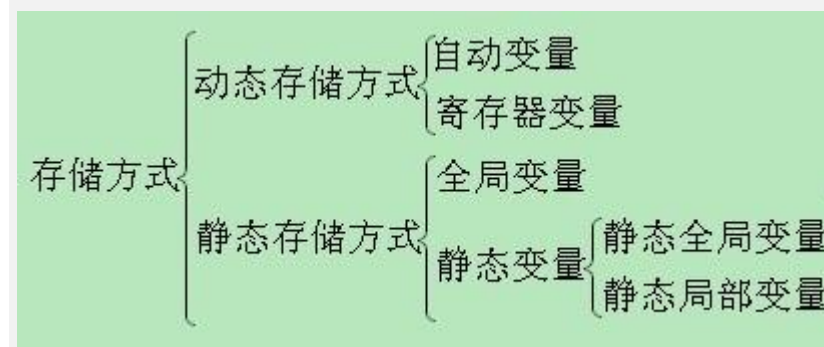
`strcpy(p1, "123456");` //123456\0 放在常量区，编译器可能会将它与 p3 所指向的"123456"优化成一个地方。

6、变量的存储类型和存储方式

存储类型：



存储方式：



静态存储变量通常是在变量定义时就分配一定的存储空间并一直保持不变，直至整个程序结束。

动态存储变量是在程序执行过程中，使用它时才分配存储单元，使用完毕立即释放。

静态存储变量是一直存在的，而动态存储变量则时而存在时而消失。

全局变量：作用域为整个源程序，即所有源文件；

静态全局变量：作用域为定义该变量的源文件；

把全局变量改变为静态全局变量后改变了它的作用域，而生存期不变；

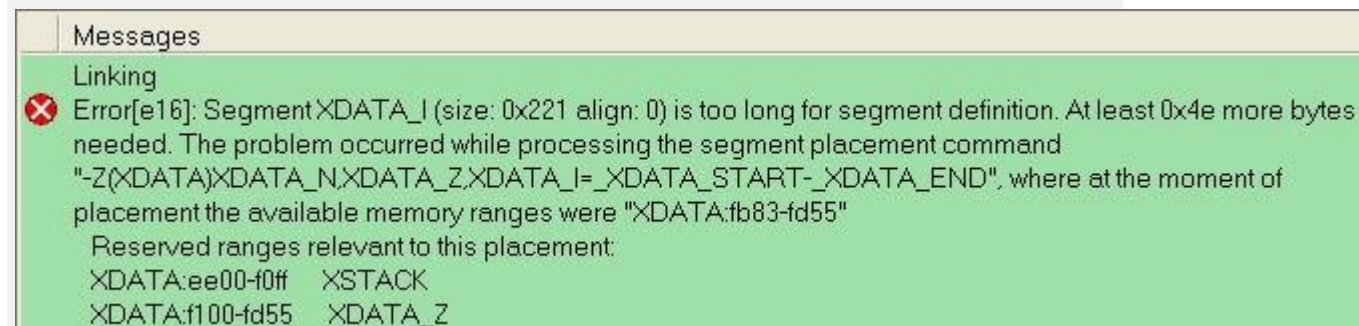
局部变量：生存期为定义它的函数或复合语句；

静态局部变量：生存期为整个源程序；

把局部变量改变为静态局部变量后改变了它的生存期，而作用域不变；

二、

对《SimpleApp 中 Controller/Switch 绑定小实验》中出现的以下 XDATA 溢出错误,记录下个人的一点理解.



《SimpleApp 中 Controller/Switch 绑定小实验》中整个工程编译后的内存占用如下:

```
122 082 bytes of CODE memory
    18 bytes of DATA memory (+ 76 absolute )
  7 109 bytes of XDATA memory (+ 8 absolute )
    192 bytes of IDATA memory
     8 bits of BIT memory

Errors: none
Warnings: none
```

【图 1】

那在函数 `zb_ReceiveDataIndication()` 中添加下面语句:

```
//*****
uint8 buf[]="HELLO";
HalUARTWrite ( 0, buf, sizeof(buf) );

//*****
```

编译后的内存占用如下:

```
122 131 bytes of CODE memory
    18 bytes of DATA memory (+ 76 absolute )
  7 115 bytes of XDATA memory (+ 8 absolute )
    192 bytes of IDATA memory
     8 bits of BIT memory

Errors: none
Warnings: none
```

怎么查看内存信息呢？

查 IAR 编译信息和程序大小

一、设置：

1、怎么设置可以查看单片的内存（消耗）使用状况？

IAR 的菜单栏 -->Tools -->IDE Options -->Messages -->Show build messages 选择 ALL

设置完后 IAR 点 MAKE，没有 error 的话最后会在 Message 框显示代码的大小
(菜单栏-->View-->Messages-->Build)

1. Tools->option->messages show build messages 设置成 all

这样可以在编译后看到了用了多少个 RAM 多少 Code，中间每个文件使用情况也可以看得到。

2. Project->Options...(Alt-F7))的 Linker->List 中勾选 Generate linker map file

这样可以生成 map 文件，这个就更详细了（需要什么自己勾选）。

3. Project->Options...(Alt-F7))的 C/C++ compile ->List 中勾选 output list file

这样可以看到每个文件编译的细节的 list 文件，还可以看到反汇编（需要什么自己勾选）。

二、编译信息

1、下面什么意思

4 960 bytes of CODE memory

4 150 bytes of DATA memory

44 bytes of CONST memory

4 960 个字节的代码(占 FLASH)

4150 个字节的变量数据(占 RAM)

44 个字节的常量数据(占 FLASH)

生成 bin 文件的大小= 4960+44

1、

9 486 bytes of CODE memory

404 bytes of DATA memory (+ 36 absolute)

520 bytes of CONST memory

1、的括号内表示内存的绝对使用量，对应你设的全局变量大小

2、表示共享的存储空间，和单片机的结构有关

与图 1 相比 XDATA 多了 6 字节，个人认为这 6 个字节是：HELLO\0
buf 这个数组应该属于自动变量，属于动态存储方式，只有在使用它，即当 buf
被调用时才给它分配内存单元，开始它的生存期，调用结束，释放存储单元，
结束生存期。

那在 `zb_ReceiveDataIndication()` 中这样添加(`typedef __code const uint8 CCUINT8;`):

```
//*****
static CCUINT8 buf[]="HELLO";
HalUARTWrite ( 0, (uint8 *)buf, sizeof(buf) );

//*****
```

编译后的内存占用如下:

```
122 107 bytes of CODE memory
      18 bytes of DATA memory (+ 76 absolute )
  7 109 bytes of XDATA memory (+ 8 absolute )
      192 bytes of IDATA memory
      8 bits of BIT memory

Errors: none
Warnings: none
```

与图 1 相比可以发现这里字符串并没有占用 `XDATA`, 而是占用 `CODE`。

若没有 `static` 声明为静态变量, 则出现以下错误:

Error[Be009]: memory attributes not allowed on auto variables or parameters

当然, 把它声明为全局变量(?常量)也可行, 在 `SimpleController.c` 开头定义全局变量(?常量):

`CCUINT8 buf[]="HELLO";`

编译后的内存占用如下:

```
122 107 bytes of CODE memory
      18 bytes of DATA memory (+ 76 absolute )
  7 109 bytes of XDATA memory (+ 8 absolute )
      192 bytes of IDATA memory
      8 bits of BIT memory

Errors: none
Warnings: none
```

两种方式所占内存一样, 全局变量与静态变量的存储方式都是静态存储, 存储空间是在编译完成后就分配的, 并且在程序运行的全部过程中都不会撤销。

至于出现错误 **Error[Be009]** 的原因, 和实验室的人讨论也没有得出一个很明确的答案。

`typedef __code const uint8 CCUINT8;`

`CCUINT8 buf[]="HELLO";`

按理说应该是把 `HELLO\0` 放在常量区(???)，编译后就分配内存，程序运行中只能读取不能改写。

(1)把 `CCUINT8 buf[ ]="HELLO"`;放在 `SimpleController.c` 开头声明为全局变量(?常量)，这种存储方式应该是符合的；

(2)把 `static CCUINT8 buf[ ]="HELLO"`;放在 `zb_ReceiveDataIndication()`中声明为静态局部变量(?常量)，这种存储方式也应该是符合的；

(3)把 `CCUINT8 buf[ ]="HELLO"`;放在 `zb_ReceiveDataIndication()`中，按我个人理解，编译器会认为它是一个自动变量，因为关键字“**auto**”可以省略，**auto**不写则隐含确定为“自动存储类别”，属于动态存储方式，即只有在使用它时才给它分配存储单元，开始它的生存期，使用结束后，释放单元，结束生存期。这种存储方式不符合。

对于 `typedef __code const uint8 CCUINT8;`，我是看到协议栈中 `hal_adc.c` 定义过这么个数组：

```
static __code const uint16 HalAdcVddLimit[] =
{
    .....
};
```

以及在 `Font.c` 定义过这么个数组：

```
__code const INT8U Font8X8[] =
{
    .....
}
```

然后拿过来看看能否解决把字符串定义到 **CODE** 中而不占用 **XDATA**，具体很详细的语法意思不知道，总之最终可行。

如果把这个数组定义过大，比如在 `SimpleController.c` 开头定义：

```
CCUINT8 buff[4000];
```

在 `zb_ReceiveDataIndication()`中：

```
HalUARTWrite ( 0, (uint8 *)buff, sizeof(buff) );
```

则编译会出现以下错误：

✖ Error[e16]: Segment CODE_C (size: 0x1ba2 align: 0) is too long for segment definition. At least 0x219 more bytes needed. The problem occurred while processing the segment placement command "-Z(CODE)CODE_C=_CODE_START-_CODE_END", where at the moment of placement the available memory ranges were "CODE:2678-4000"

还缺少 0x219 字节，即 537 字节，那我这样定义：CCUINT8 buff[3463];编译没有问题：

```
125 564 bytes of CODE memory
      18 bytes of DATA memory (+ 76 absolute )
  7 109 bytes of XDATA memory (+ 8 absolute )
      192 bytes of IDATA memory
        8 bits of BIT memory

Errors: none
Warnings: none
```

那我再加 1 字节：CCUINT8 buff[3464];出现这错误：

✖ Error[e16]: Segment CODE_C (size: 0x198a align: 0) is too long for segment definition. At least 0x1 more bytes needed. The problem occurred while processing the segment placement command "-Z(CODE)CODE_C=_CODE_START-_CODE_END", where at the moment of placement the available memory ranges were "CODE:2678-4000"

即多了 0x01 字节。O 了~

另一种方法，自己申请一个堆

最初工程内存占用为：

```
122 082 bytes of CODE memory
      18 bytes of DATA memory (+ 76 absolute )
  7 109 bytes of XDATA memory (+ 8 absolute )
      192 bytes of IDATA memory
        8 bits of BIT memory

Errors: none
Warnings: none
```

那我在 zb_ReceiveDataIndication() 中这样添加：

```
//*****
uint8 *buf=(uint8 *)osal_mem_alloc(5*sizeof(uint8));
buf[0]='H';buf[1]='E';buf[2]='L';buf[3]='L';buf[4]='O';
HalUARTWrite ( 0, buf, sizeof(buf) );
//*****
```

编译后的内存占用如下：

```
122 140 bytes of CODE memory
      18 bytes of DATA memory (+ 76 absolute )
7 109 bytes of XDATA memory (+ 8 absolute )
      192 bytes of IDATA memory
      8 bits of BIT memory

Errors: none
Warnings: none
```

与图 1 相比发现这样定义不占用 XDATA。但个人总感觉如果废话多点，这样把字符一个一个赋进去十分麻烦，可是不知道如何快捷地把一串字符赋给动态开辟的堆.....

我这样添加：

```
//*****
uint8 *buf=(uint8 *)osal_mem_alloc(5*sizeof(uint8));
// buf[0]='H';buf[1]='E';buf[2]='L';buf[3]='L';buf[4]='O';
buf="HELLO";
HalUARTWrite ( 0, buf, sizeof(buf) );
>
//*****
```

与图 1 相比可以看到编译后的内存占用 XDATA 还是多了 6 字节：

```
122 114 bytes of CODE memory
      18 bytes of DATA memory (+ 76 absolute )
7 115 bytes of XDATA memory (+ 8 absolute )
      192 bytes of IDATA memory
      8 bits of BIT memory

Errors: none
Warnings: none
```

我这样添加：

```
//*****
// uint8 *buf=(uint8 *)osal_mem_alloc(5*sizeof(uint8));
char *buf=(char *)osal_mem_alloc(5*sizeof(char));
// buf[0]='H';buf[1]='E';buf[2]='L';buf[3]='L';buf[4]='O';
strcpy(buf, "HELLO");
HalUARTWrite ( 0, (uint8*)buf, sizeof(buf) );
//*****
```

与图 1 相比可以看到编译后的内存占用 XDATA 一样还是多了 6 字节:

```
122 193 bytes of CODE memory
      18 bytes of DATA memory (+ 76 absolute )
  7 115 bytes of XDATA memory (+ 8 absolute )
      192 bytes of IDATA memory
      8 bits of BIT memory

Errors: none
Warnings: none
```

.....还是不知道怎么赋...=.="🤔

小结[我不敢百分百确定,大家自己可以实验下]:

1、不占用 XDATA, 对于字符串:

(1) typedef __code const uint8 CCUINT8; 通过 CCUINT8 定义到 CODE 中; //char 也 OK

(2)通过 osal_mem_alloc()函数动态开辟堆空间, 把字符一个一个赋进去; 用完后调用 osal_mem_free()进行内存释放!

2、不占用 XDATA, 对于一堆需要用数组存储的数据:

通过 osal_mem_alloc()函数动态开辟堆空间, 把数据一个一个赋进去; 用完后调用 osal_mem_free()进行内存释放!

3、对 XDATA 溢出, 我还没有试验过是否可以通过修改配置文件中的 -D_XDATA_END 来解决, 但看了下貌似不可以, 已经达到最大值 8K 了;

// These settings are used for devices that don't use PM2/PM3

-D_IXDATA_START=E000 // The internal IXDATA block is 8K,

-D_IXDATA_END=FEFF // End of IXDATA if PM2/PM3 are not used

//

// These settings must be used for devices that use PM2/PM3.

// Note that the IXDATA_START allows the XSTACK to grow down into the non-persistent RAM, but

// checks in HAL Sleep insure that the stack is back into persistent RAM before entering PM2/PM3.

//-D_IXDATA_START=EE00 // The internal IXDATA block is 4K+,

//-D_IXDATA_END=FD55 // FD56-FE00 is used for saving the

CC2430 registers before sleep.低功耗时这部分用于存储寄存器值

```
//  
// FF00-FFFF is mapped to IDATA.  
//  
//  
// XDATA  
//  
-D_XDATA_START=_IXDATA_START // The IXDATA is used as XDATA.  
-D_XDATA_END=_IXDATA_END  
*****
```

4、若 CODE 溢出： (1)减小程序； (2)把配置文件 f8w2430.xcl/f8w2430pm.xcl 中的-D_CODE_END 改大点：

```
*****  
f8w2430.xcl:  
// CODE  
//  
// These settings determine the size/location of the ROOT segment.  
// Increase _CODE_END to increase ROOT memory, i.e. for constants.  
-D_CODE_START=0x0000 // Code size = 128k for CC2430-F128  
-D_CODE_END=0x4000 // Last address for ROOT bank  
*****
```

```
f8w2430pm.xcl:  
// CODE  
//  
// These settings determine the size/location of the ROOT segment.  
// Increase _CODE_END to increase ROOT memory, i.e. for constants.  
-D_CODE_START=0x0000 // Code size = 128k for CC2430-F128  
-D_CODE_END=0x29FF //(原 0x28FF) Last address for ROOT  
bank 这里我修改增大过的。
```