data,bdata,idata,pdata,xdata,code 存储类型与存储区
2008 年 06 月 12 日 星期四 上午 00:00
转帖:http://blog.csdn.net/dpear/archive/2007/09/12/1781523.aspx

bit

是在内部数据存储空间中 20H .. 2FH 区域中一个位的地址，或者 8051 位可寻址 SFR 的一个位地址。

code
是在 0000H .. 0FFFFH 之间的一个代码地址。

data

是在 0 到 127 之间的一个数据存储器地址，或者在 128 .. 255 范围内的一个特殊功能寄存器（SFR）地址。

idata
是 0 to 255 范围内的一个 idata 存储器地址。

xdata 是 0 to 65535 范围内的一个 xdata 存储器地址。

指针类型和存储区的关系详解

一、存储类型与存储区关系

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)
code	--->	程序存储区 (64k 地址范围),对应 MOVC @DPTR

二、指针类型和存储区的关系

对变量进行声明时可以指定变量的存储类型如:
uchar data x 和 data uchar x 相等价都是在内 ram 区分配一个字节的变量。

同样对于指针变量的声明，因涉及到指针变量本身的存储位置和指针所指向的存储区位置不同而进行相应的存储区类型关键字的
使用如：

```
uchar xdata * data pstr
```

是指在内 ram 区分配一个指针变量("*"号后的 data 关键字的作用)，而且这个指针本身指向 xdata 区("*"前 xdata 关键字的作用)，
可能初学 C51 时有点不好懂也不好记。没关系，我们马上就可以看到对应“*”前后不同的关键字的使用在编译时出现什么情况。

```
......
uchar xdata tmp[10];      //在外 ram 区开辟 10 个字节的内存空间，地址是外 ram 的 0x0000－0x0009
......
```

第 1 种情况:

```
uchar data * data pstr;
pstr=tmp;
```

首先要提醒大家这样的代码是有 bug 的, 他不能通过这种方式正确的访问到 tmp 空间。 为什么？我们把编译后看到下面的汇编
代码：

```
MOV 0x08,#tmp(0x00)          ;0x08 是指针 pstr 的存储地址
```

看到了吗！本来访问外 ram 需要 2 byte 来寻址 64k 空间，但因为使用 data 关键字(在"*"号前的那个)，所以按 KeilC 编译环境来说
就把他编译成指向内 ram 的指针变量了，这也是初学 C51 的朋友们不理解各个存储类型的关键字定义而造成的 bug。特别是当工程中的
默认的存储区类为 large 时，又把 tmp[10] 声明为 uchar tmp[10] 时，这样的 bug 是很隐秘的不容易被发现。

第 2 种情况:

```
uchar xdata * data pstr;
pstr = tmp;
```

这种情况是没问题的，这样的使用方法是指在内 ram 分配一个指针变量("*"号后的 data 关键字的作用)，而且这个指针本身指向
xdata 区("*"前 xdata 关键字的作用)。编译后的汇编代码如下。

```
MOV 0x08,#tmp(0x00)          ;0x08 和 0x09 是在内 ram 区分配的 pstr 指针变量地址空间
MOV 0x09,#tmp(0x00)
```

这种情况应该是在这里所有介绍各种情况中效率最高的访问外 ram 的方法了，请大家记住他。

第 3 种情况:

```
uchar xdata * xdata pstr;
pstr=tmp;
```

这中情况也是对的，但效率不如第 2 种情况。编译后的汇编代码如下。

```
MOV DPTR, #0x000A          ;0x000A,0x000B 是在外 ram 区分配的 pstr 指针变量地址空间
MOV A, #tmp(0x00)
MOV @DPTR, A
INC DPTR
MOV A, #tmp(0x00)
MOVX @DPTR, A
```

这种方式一般用在内 ram 资源相对紧张而且对效率要求不高的项目中。

第 4 种情况:

```
uchar data * xdata pstr;
pstr=tmp;
```

如果详细看了第 1 种情况的读者发现这种写法和第 1 种很相似，是的，同第 1 种情况一样这样也是有 bug 的，但是这次是把 pstr 分
配到了外 ram 区了。编译后的汇编代码如下。

```
MOV DPTR, #0x000A          ;0x000A 是在外 ram 区分配的 pstr 指针变量的地址空间
MOV A, #tmp(0x00)
MOVX @DPTR, A
```

第 5 种情况:

```
uchar * data pstr;
pstr=tmp;
```

大家注意到"*"前的关键字声明没有了，是的这样会发生什么事呢？下面这么写呢！对了用齐豫的一首老歌名来说就是 “请跟我
来”，请跟我来看看编译后的汇编代码，有人问这不是在讲 C51 吗？ 为什么还

要给我们看汇编代码。C51 要想用好就要尽可能提升 C51
编译后的效率，看看编译后的汇编会帮助大家尽快成为生产高效 C51 代码的高手的。还是看代码吧！

```
MOV 0x08, #0X01                ;0x08－0x0A 是在内 ram 区分配的 pstr 指针变量的地址空间
MOV 0x09, #tmp(0x00)
MOV 0x0A, #tmp(0x00)
```

注意：这是新介绍给大家的，大家会疑问为什么在前面的几种情况的 pstr 指针变量都用 2 byte 空间而到这里就用 3 byte 空间了
呢？这是 KeilC 的一个系统内部处理，在 KeilC 中一个指针变量最多占用 3 byte 空间，对于没有声明指针指向存储空间类型的指针，
系统编译代码时都强制加载一个字节的指针类型分辩值。具体的对应关系可以参考 KeilC 的 help 中 C51 User's Guide。

第 6 种情况:

```
uchar * pstr;
pstr=tmp;
```

这是最直接最简单的指针变量声明，但他的效率也最低。还是那句话，大家一起说好吗！编译后的汇编代码如下。

```
MOV DPTR, #0x000A              ;0x000A－0x000C 是在外 ram 区分配的 pstr 指针变量地址空间
MOV A, #0x01
MOV @DPTR, A
INC DPTR
MOV DPTR, #0x000A
MOV A, #tmp(0x00)
MOV @DPTR, A
INC DPTR
MOV A, #tmp(0x00)
MOVX @DPTR, A
```

这种情况很类似第 5 种和第 3 种情况的组合，既把 pstr 分配在外 ram 空间了又增加了指针类型的分辨值。

小结一下：大家看到了以上的 6 种情况，其中效率最高的是第 2 种情况，既可以正确访问 ram 区又节约了代码，效率最差的是第 6
种，但不是说大家只使用第 2 种方式就可以了，还要因情况而定，一般说来应用 51 系列的系统架构的内部 ram 资源都很紧张，最好大家

在定义函数内部或程序段内部的局部变量使用内 ram，而尽量不要把全局变量声明为内 ram 区中。所以对于全局指针变量我建议使用第
3 种情况，而对于局部的指针变量使用第 2 种方式。