GPIO库

GPIO库是基础中的基础，所以默认就是一定要开启并参与编译的。但并非所有函数都有用，可以适当得进行一些优化。

特别注意

在gpio.h中定义的宏定义D00~D77这种，**仅用于gpio库的函数参数使用，不能用于获取IO的电平值。**在ecbm_reg.h定义的宏定义是P00~P77，在官方的头文件也是这样的格式。

```
gpio_mode(D10,GPIO_IN);//正确的，D10作为参数使用，代表的是P1.0。
P10=1;//也是正确的，P10就是对应IO，P1.0置高电平。

gpio_mode(P22,GPIO_IN);//错误的，P22是IO口，不能作为参数使用。
D22=1;//也是错误的，D22是一个数值，对数值赋值会直接报错。
```

那么为什么会搞出两种不同的东西呢？原因就是一开始的理念，就是为了想让IO口作为一个变量，这样在编译好了之后，通过修改变量的值就能修改IO口。给后期维护提供一个十分便利的功能。但是很遗憾，在51单片机里是行不通的。不管你C语言如何使用各种技巧，最终还是要转成汇编语句。而51的汇编是没有对特殊寄存器（SFR）区间接寻址的指令。这就导致了从原理上就不能把IO做为一个参数输入到函数中。

已知官方头文件使用P00~P77作为IO口的别名（其实就是用sbit定义的意思），那么ECBM就得用别的称呼。最终选择了gpio库采用数字的英文digital的首字母D来作为IO的替代宏定义。而adc库用的是模拟的英文analog的首字母A来作为IO的替代宏定义。

正因为要和P--的定义分开，所以D00~D77仅仅使用了最简单的替换法，比如P0口是：

```
#define D00 (0x00)//对应的是P00。
#define D01 (0x01)//对应的是P01。
#define D02 (0x02)//对应的是P02。
#define D03 (0x03)//对应的是P03。
#define D04 (0x04)//对应的是P04。
#define D05 (0x05)//对应的是P05。
#define D06 (0x06)//对应的是P06。
#define D07 (0x07)//对应的是P07。
```

当你不小心把D--当P--使用的时候会发生什么？

```
//读取的时候：
if(D12==1){//你的本意是想判断P1.2是不是高电平，但是编译器翻译宏定义之后就变成了
if(0x12==1)，于是if永远不成立。
    ...
}
//写入的时候：
D12=1;//你的本意是想让P1.2输出高电平，但实际上编译的是0x12=1，由于不能给一个数赋值，keil会立马报错。
```

API

gpio_uppull

函数原型：void gpio_uppull(u8 pin,u8 en);

描述

IO口上拉电阻配置函数，可设定打开或关闭某个IO口的内置上拉电阻。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23
- en：使能开关，1代表打开上拉电阻；0代表关闭上拉电阻。

输出

无

返回值

无

调用例程

直接参数调用：

```
gpio_uppull(D11,1);//打开P1.1脚的上拉电阻。
```

变量做参数调用：

```
u8 pin_set[5]={D10,D21,D22,D41,D40};//定义一个数组，存放5个IO口的编号。
u8 i;//临时变量。
for(i=0;i<5;i++){//打开这5个IO口的上拉电阻。
    gpio_uppull(pin_set[i],1);
}
```

注意事项

1. 本函数调用的寄存器是在扩展寄存器区，所以要保证扩展寄存器的访问使能是打开的。目前system_init()函数内部会自动打开这个使能，但是如果你为了优化而没有调用system_init()函数的话，需要加上“EX_SFR_ENABLE；”。
2. 本函数不支持多个IO同时开启上拉电阻，需要用for循环一个一个设置。

gpio_uppull_ex

函数原型：void gpio_uppull_ex(u8 port,u8 pin,u8 en);

描述

IO口上拉电阻配置函数扩展版，可批量设定打开或关闭几个IO口的内置上拉电阻。

输入

- port：P口的编号，比如P0口就是GPIO_P0。
- pin：IO的编号，GPIO_PIN_0~GPIO_PIN_7对应了Px.0~Px.7。
- en：使能开关，1代表打开上拉电阻；0代表关闭上拉电阻。

输出

无

返回值

无

调用例程

```
gpio_uppull_ex(GPIO_P0,GPIO_PIN_0|GPIO_PIN_1,1);//P0.0和P0.1口打开内部的上拉电阻。
```

注意事项

1. 本函数调用的寄存器是在扩展寄存器区，所以要保证扩展寄存器的访问使能是打开的。目前system_init()函数内部会自动打开这个使能，但是如果你为了优化而没有调用system_init()函数的话，需要加上“EX_SFR_ENABLE；”。
2. 本函数和gpio_uppull函数相比，支持同一个P口的多个IO同时开启上拉电阻。

gpio_mode

函数原型：void gpio_mode(u8 pin,u8 mode);

描述

IO口工作模式设置函数，可设定某个IO口的工作模式。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23
- mode：工作的模式，有以下4个可供选择。
    1. GPIO_PU / GPIO_IN：弱上拉模式，此模式下IO的上拉能力弱，下拉能力强。常用在输入输出双向口。
    2. GPIO_HZ：高阻模式，此模式下IO不能对外输出高低电平，但能读取IO电平。一般用在输入口。
    3. GPIO_OD：开漏模式，此模式下IO不能输出高电平，需要有额外的上拉电阻才能输出高电平。一般用在总线上。
    4. GPIO_PP / GPIO_OUT：推挽模式，此模式下的上拉下拉能力都强，常用在输出口。

输出

无

返回值

无

调用例程

直接参数调用：

```
gpio_mode(D11,GPIO_OUT);//把P1.1脚设置为推挽（输出）模式。
```

变量做参数调用：

```
u8 pin_set[5]={D10,D21,D22,D41,D40};//定义一个数组，存放5个IO口的编号。
u8 i;//临时变量。
for(i=0;i<5;i++){//把这5个IO口设置为输入（弱上拉）模式。
    gpio_mode(pin_set[i],GPIO_IN);
}
```

联动调用：

```
u8 pin_set[5]={D10,D21};//定义一个数组，存放2个IO口的编号。假设把它们当做IIC口。
u8 i;//临时变量。
for(i=0;i<2;i++){
    gpio_mode(pin_set[i],GPIO_IN);//把这2个IO口设置为开漏模式。
    gpio_uppull(pin_set[i],1);//IIC总线的设定就是开漏+上拉。
}
```

注意事项

1. 本函数不支持多个IO同时设置工作模式，需要用for循环一个一个设置。

gpio_mode_ex

函数原型：void gpio_mode_ex(u8 port,u8 pin,u8 mode);

描述

设置IO口工作模式函数扩展版，可同时设置同一P口的多个IO口。

输入

- port：P口的编号，比如P0口就是GPIO_P0。
- pin：IO的编号，GPIO_PIN_0~GPIO_PIN_7对应了Px.0~Px.7。
- mode：工作的模式，有以下4个可供选择。
    1. GPIO_PU / GPIO_IN：弱上拉模式，此模式下IO的上拉能力弱，下拉能力强。常用在输入输出双向口。
    2. GPIO_HZ：高阻模式，此模式下IO不能对外输出高低电平，但能读取IO电平。一般用在输入口。
    3. GPIO_OD：开漏模式，此模式下IO不能输出高电平，需要有额外的上拉电阻才能输出高电平。一般用在总线上。
    4. GPIO_PP / GPIO_OUT：推挽模式，此模式下的上拉下拉能力都强，常用在输出口。

输出

无

返回值

无

调用例程

```
gpio_mode_ex(GPIO_P0,GPIO_PIN_0|GPIO_PIN_1,GPIO_HZ);//P0.0和P0.1口设置为高阻态模式。
```

注意事项

1. 本函数和gpio_mode函数相比，支持多个IO同时设置工作模式。

gpio_speed

函数原型：void gpio_speed(u8 pin,u8 speed);

描述

IO口速度设置函数，可设定某个IO口的工作速度。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23。
- speed：IO速度，有以下2个可供选择。
    1. GPIO_FAST：快速模式。
    2. GPIO_SLOW：慢速模式。

输出

无

返回值

无

调用例程

直接参数调用：

```
gpio_speed(D11,GPIO_FAST);//把P1.1脚设置为快速模式。
```

变量做参数调用：

```
u8 pin_set[5]={D10,D21,D22,D41,D40};//定义一个数组，存放5个IO口的编号。
u8 i;//临时变量。
for(i=0;i<5;i++){//把这5个IO口设置为快速模式。
    gpio_speed(pin_set[i],GPIO_FAST);
}
```

注意事项

1. 本函数不支持多个IO同时设置工作模式，需要用for循环一个一个设置。
2. 其实我实测两个模式似乎没啥区别。
3. 本函数调用的寄存器是在扩展寄存器区，所以要保证扩展寄存器的访问使能是打开的。目前system_init()函数内部会自动打开这个使能，但是如果你为了优化而没有调用system_init()函数的话，需要加上“EX_SFR_ENABLE；”。

gpio_current

函数原型：void gpio_current(u8 pin,u8 current);

描述

IO口驱动电流设置函数，可设定某个IO口的驱动电流。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23。
- current：IO驱动电流，有以下2个可供选择。
    1. GPIO_STR：增强驱动模式。
    2. GPIO_GEN：正常驱动模式。

输出

无

返回值

无

调用例程

直接参数调用：

```
gpio_current(D11,GPIO_STR);//把P1.1脚设置为大电流模式。
```

变量做参数调用：

```
u8 pin_set[5]={D10,D21,D22,D41,D40};//定义一个数组，存放5个IO口的编号。
u8 i;//临时变量。
for(i=0;i<5;i++){//把这5个IO口设置为大电流模式。
    gpio_current(pin_set[i],GPIO_STR);
}
```

注意事项

1. 本函数不支持多个IO同时设置工作模式，需要用for循环一个一个设置。
2. 即使是大电流模式也不应该超过20mA，以防IO烧毁。
3. 本函数调用的寄存器是在扩展寄存器区，所以要保证扩展寄存器的访问使能是打开的。目前system_init()函数内部会自动打开这个使能，但是如果你为了优化而没有调用system_init()函数的话，需要加上“EX_SFR_ENABLE；”。

gpio_write

函数原型：void gpio_write(u8 port,u8 dat);

描述

P口写入函数，直接写入8位数据到某个P口上。

输入

- port：P口编号，宏定义GPIO_P0~GPIO_P7，比如P1口就是GPIO_P1。
- dat：要输出的数据。

输出

无

返回值

无

调用例程

直接参数调用：

```
gpio_write(GPIO_P1,0x05);//P1口输出0x05。
```

变量做参数调用：

```
u8 led_type[8]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};//定义一个数组，存放8个数据。
u8 i;//临时变量。
for(i=0;i<8;i++){
    gpio_write(GPIO_P1,led_type[i]);//研究数组数据即可知这段的效果就是从P1.0到P1.7依次点亮的流水灯效果。
    delay_ms(500);//流水灯不用太快，延时500mS。
}
```

注意事项

1. 本函数在“gpio_write(GPIO_P1,0);”的时候和“P1=0;”是等价的，甚至“P1=0;”的运行速度会快非常多。但是本函数的优点不是运行速度，而是可以用变量作为参数，比如“gpio_write(SBUF,0);”这样写就能用串口实时控制某个P口全置0。

gpio_read

函数原型：u8 gpio_read(u8 port);

描述

P口读出函数，读取某个P口的数据（8位同时读，也就是并口通信）。

输入

- port：P口编号，宏定义GPIO_P0~GPIO_P7，比如P1口就是GPIO_P1。

输出

无

返回值

- 读到的P口数据。

调用例程

直接参数调用：

```
u8 val;//定义一个变量。
val=gpio_read(GPIO_P1);//P1口的值存入变量val中。
```

变量做参数调用：

```
u8 val[3];//定义一个数组，存放3个数据。
u8 i;//临时变量。
for(i=0;i<3;i++){
    val[i]=gpio_read(GPIO_P1+i);//GPIO_P1的宏定义值就是1，所以GPIO_P1+1等价于
GPIO_P2。这个循环的效果就是读取P1~P3的值存到数组中。
}
```

注意事项

1. 本函数在“val=gpio_read(GPIO_P1);”的时候和“val=P1;”是等价的，同样“val=P1;”的运行速度会快非常多。但是本函数的优点不是运行速度，而是可以用变量作为参数，比如“SBUF=gpio_read(SBUF);”这样写就能用串口实时读取某个P口的值。

gpio_out

函数原型：void gpio_out(u8 pin,u8 value);

描述

IO口输出函数，用于输出高低电平。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23。
- value：0代表低电平；非0代表高电平。

输出

无

返回值

无

调用例程

直接参数调用：

```
gpio_out(D15,1);//P1.5脚输出高电平。
```

变量做参数调用：

```
u8 pin_list[3]={D15,D24,D32};//定义一个数组，存放3个引脚编号。
u8 i;//临时变量。
for(i=0;i<3;i++){
    gpio_out(pin_list[i],1);//依次把这3个IO置高。
}
```

注意事项

1. 和上面说的一样，本函数的优点在于控制的IO口是可变的，这一点为后面的库的引脚变动提供了很大的方便。如果不是为了这个便利性的话，还不如直接用“P15=1;”这样的语句来的快。

gpio_in

函数原型：u8 gpio_in(u8 pin);

描述

IO口输入函数，用于读取某个IO的电平状态。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23。

输出

无

返回值

- 该IO的电平状态。0代表低电平；1代表高电平。

调用例程

直接参数调用：

```
u8 val;//定义一个变量。
val=gpio_in(D15);//把P1.5脚的电平状态赋给val。
```

变量做参数调用：

```
u8 pin_list[3]={D15,D24,D32};//定义一个数组，存放3个引脚编号。
u8 value[3];//定义一个数组存放电平状态。
u8 i;//临时变量。
for(i=0;i<3;i++){
    value[i]=gpio_in(pin_list[i]);//依次把这3个IO的电平状态装进数组。
}
```

注意事项

1. 和上面说的一样，本函数的优点在于控制的IO口是可变的，这一点为后面的库的引脚变动提供了很大的方便。如果不是为了这个便利性的话，还不如直接用“val=P15;”这样的语句来的快。

gpio_toggle

函数原型：void gpio_toggle(u8 pin);

描述

IO口翻转函数，用于翻转某个IO的电平状态。

输入

- pin：IO的编号，比如P1.0脚就是D10，P2.3脚就是D23。

输出

无

返回值

无

调用例程

直接参数调用：

```
gpio_toggle(D15);//把P1.5脚的电平状态翻转。
```

变量做参数调用：

```
u8 pin_list[3]={D15,D24,D32};//定义一个数组，存放3个引脚编号。
u8 i;//临时变量。
for(i=0;i<3;i++){
    gpio_toggle(pin_list[i]);//依次把这3个IO的电平状态翻转。
}
```

注意事项

1. 和上面说的一样，本函数的优点在于控制的IO口是可变的，这一点为后面的库的引脚变动提供了很大的方便。如果不是为了这个便利性的话，还不如直接用“P15=!P15;”这样的语句来的快。

gpio_toggle_fast

函数原型：void gpio_toggle_fast(u8 port,u8 pin);

描述

IO口快速翻转函数，gpio_toggle函数的加速版。

输入

- port：P口的编号，比如要控制P1.0，这里填写GPIO_P1
- pin：P口IO的编号，比如要控制P1.0，这里填写GPIO_PIN_0。

输出

无

返回值

无

调用例程

单个IO翻转：

```
gpio_toggle_fast(GPIO_P1,GPIO_PIN_0);//把P1.0脚的电平状态翻转。
```

同P口多个IO翻转：

```
gpio_toggle_fast(GPIO_P1,GPIO_PIN_0|GPIO_PIN_5);//把P1.0脚和P1.5脚的电平状态翻转。
```

注意事项

1. 快速版函数删除了引脚解析，所以不仅加快了运行速度还支持多个IO（当然必须在同一个P口）同时翻转。缺点就是需要两个参数。

gpio_out_fast

函数原型：void gpio_out_fast(u8 port,u8 pin,u8 val);

描述

IO口快速输出函数，gpio_out函数的加速版。

输入

- port：P口的编号，比如要控制P1.0，这里填写GPIO_P1
- pin：P口IO的编号，比如要控制P1.0，这里填写GPIO_PIN_0。
- val：要输出的电平值，0代表低电平；1代表高电平。

输出

无

返回值

无

调用例程

单个IO输出高电平：

```
gpio_out_fast(GPIO_P1,GPIO_PIN_0,1);//P1.0脚输出高电平。
```

同P口多个IO输出低电平：

```
gpio_out_fast(GPIO_P1,GPIO_PIN_0|GPIO_PIN_5,0);//P1.0脚和P1.5脚输出低电平。
```

注意事项

1. 快速版函数删除了引脚解析，所以不仅加快了运行速度还支持多个IO（当然必须在同一个P口）同时输出一个电平值。缺点就是需要两个参数。

gpio_in_fast

函数原型：u8 gpio_in_fast(u8 port,u8 pin);

描述

IO口快速输入函数，gpio_in函数的加速版。

输入

- port：P口的编号，比如要控制P1.0，这里填写GPIO_P1
- pin：P口IO的编号，比如要控制P1.0，这里填写GPIO_PIN_0。

输出

无

返回值

- 该IO的电平状态。0代表低电平；1代表高电平。

调用例程

读IO的电平状态：

```
val=gpio_in_fast(GPIO_P1,GPIO_PIN_0);//把P1.0脚的电平状态赋给val。
```

注意事项

1. 和上面的快速函数差不多，这个也是删除了引脚解析达到加速的目的。但是本函数只支持一个IO的电平状态读取。

优化建议

打开gpio.h的图形化配置界面，在【普通优化设置】中把没有出现的IO口的使能取消掉。举例：目前使用的是STC8G1K08A-8PIN单片机，只有P3.0~P3.3和P5.4、P5.5。那么可以这样选择：

这样一来，不仅能优化空间，还能加快执行速度。而下面的【深度优化】是优化各个函数用的，没有使能的函数将不会参与编译，所以当且仅当整个工程都没有用到这些函数的时候才能优化掉，否则会出大问题。