**汇编语言设计ARM处理器Cortex\_M0+的GPIO驱动LED灯**

【字32位、半字16位、字节8位】32位的ARM处理器的偏移地址（以GPIO为例）是0x00 0x04 0x08 0x0c两两相差4字节即32位。

**汇编语言立即数表示：**立即数#0x11代表16进制下的11展开成二进制为00010001即十进制下的17，而立即数#11代表10进制下的11。【举一反三】在用Verilog语言给FPGA编程时，用到8’b00010001代表8位二进制数00010001也就是十六进制下的11。【注意区分16进制数和BCD码的区别，17换为十六进制数是00010001，换为BCD码是00010111】【二进制简写B、八进制简写O、十进制简写D、十六进制简写H】

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**GPIO**(英语:General-purpose input/output)，通用型之输入输出的简称，功能类似[8051](https://baike.so.com/doc/2108442-2230824.html" \t "https://baike.so.com/doc/_blank)的P0-P3，其接脚可以供使用者由程控自由使用，PIN脚依现实考量可作为通用输入(*GPI*)或通用输出(*GPO*)或通用输入与输出(*GPIO*)，如当clk generator, chip select等。

既然一个引脚可以用于输入、输出或其他特殊功能，那么一定有寄存器用来选择这些功能。对于输入，一定可以通过读取某个寄存器来确定引脚电位的高低;对于输出，一定可以通过写入某个寄存器来让这个引脚输出高电位或者低电位;对于其他特殊功能，则有另外的寄存器来控制它们。

中断服务程序驱动

WAKE-PA0【按下高电平】

SW3-PC12【按下低电平】

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*工作记录\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. **初步学习老何教材了解Cortex-M0+处理器**
2. **搜集资料调用库函数实现GPIO驱动LED灯和外部中断控制LED**
3. **精读参考资料，学习汇编语言对ARM的编程**
4. **汇编实现**

**问题①：**创建工程后主函数是C语言编写的main？那汇编程序放在哪里？

思路一在同一个文件下实现C语言和汇编语言的混合编程main里面嵌套汇编程序

思路二修改ARM的启动引导代码，直接在其中插入需要实现的功能

解决：第一步读懂启动引导代码文件每个模块的功能；第二步找到引导程序跳转到主函数的命令语句（如下图所示）；第三步注释掉该跳转语句，在其下方插入自己的汇编程序。

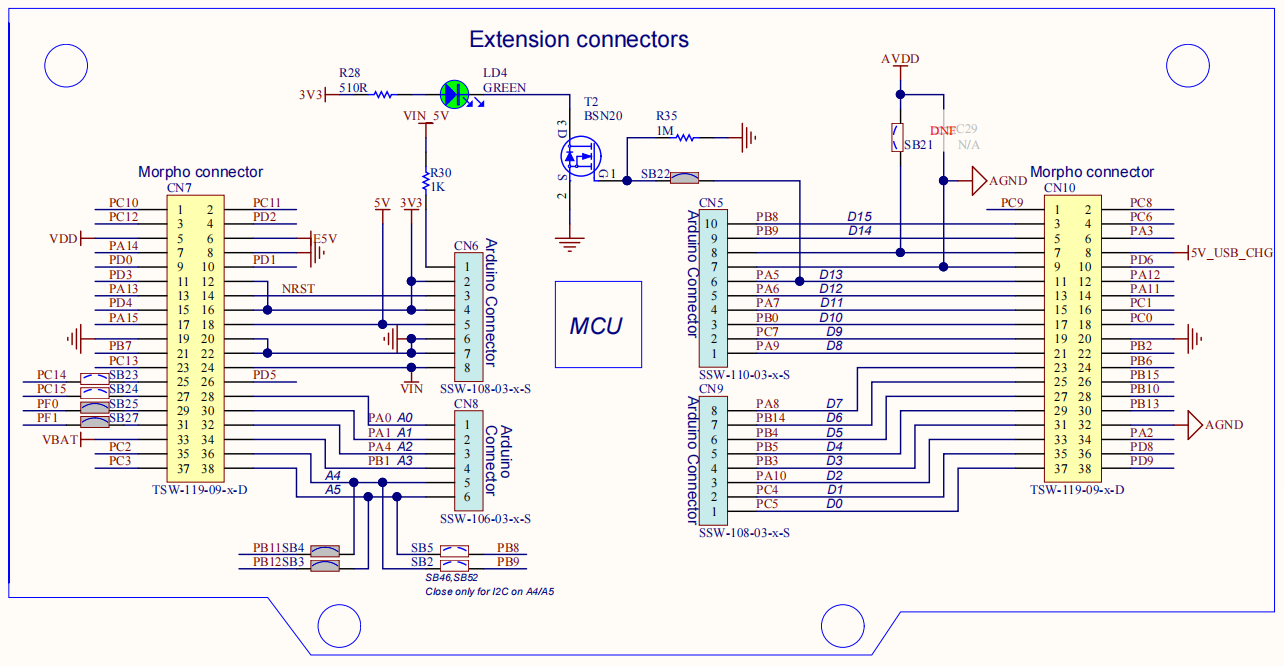
a791a7859aa67bdf76328f74ad54c3c

**问题②：**利用Debug反汇编C语言GPIO出来的汇编语言代码不能直接粘贴在该汇编语言位置处。因为他有调用其他函数。

解决：直接修改底层寄存器、存储器达到输出。

**问题③：**电路是连接LED4的？控制底层哪个端？

解决：在官网上查找STM32G071RB的原理图如下，从其中可以看出，LD4与芯片的PA5相连接。

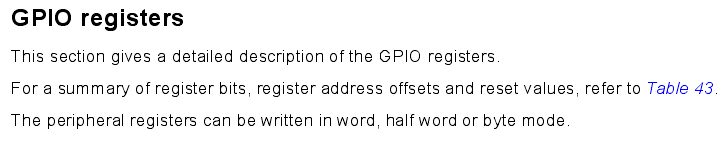


**问题④：**底层连接知道了，那么如何控制它呢？

解决：初步思路：将立即数加载到寄存器（一个寄存器中存储输出数据，另两个寄存器存储端口地址），然后一起送给存储器。

**问题⑤：**外设区域地址映射？

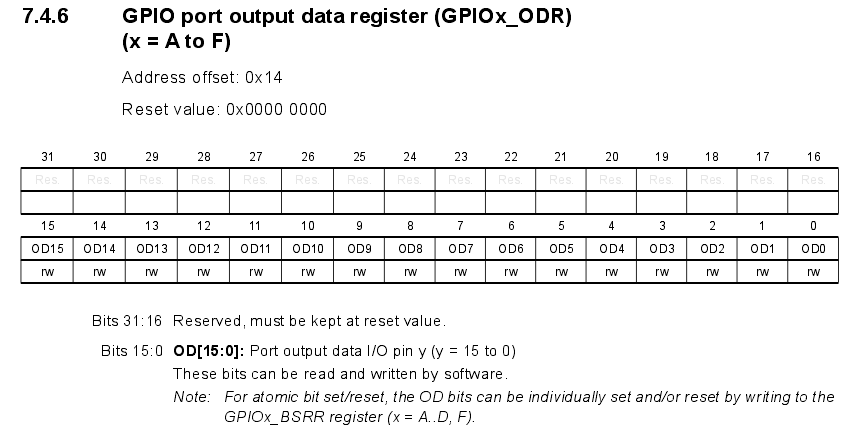
解决：器件手册数据如下：



由其介绍可知：外部寄存器只能按照字（32位）半字（16位）和字节（8位）来进行操作。所以上述指令MOVS R0,#0 STR R0,[R1,R2]是错误的，不能按位改变PA5的值。



如上图所示，我们可以知道，和GPOA相关的寄存器都在0x50000000~0x500003FF地址之间，故其**基址为0x50000000**，要想输出对应值，就需要找到它的输出寄存器，如下：

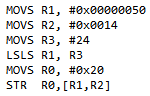


如上图所示，我们可以知道GPIO端口输出数据寄存器的**偏移地址为0x14**。并且我们想要寻找的GPIO\_PIN\_5就在从右往左数第6位的位置。

综上所述：要想对应PA5输出，首先确定其地址：基址0x50000000输出寄存器偏移地址0x14,然后向其输入一个字节数据（8位）0x20（即二进制数00100000）即可让对应的PA5管脚设置为高电平，LD4灯亮。同理，向其输入0x00即可让对应的PA5管脚设置为低电平，LD4灯灭。

**问题⑥：**在对ARM处理器用汇编语言编程时，寄存器传输指令MOVS <Rd>, #imm8对立即数#imm8的范围有限制要求（0到255），所以GPIOA\_PIN\_5的起始地址不能直接传输给寄存器。

解决：寄存器传输指令MOVS和移位指令LSLS搭配实现功能，如下图中：

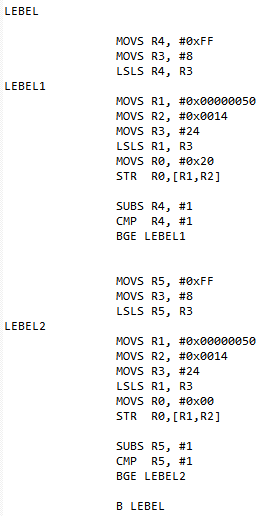


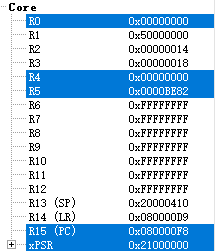
先将立即数#0x00000050传输给寄存器R1，然后将偏移地址#0x0014传输给寄存器R2，之后计算由十六进制数#0x00000050得到GPIOA输出数据寄存器的起始地址0x5000 0000需要向左移24位（4\*6）故在寄存器R3中给出左移位数#24（如果要写成16进制为0x18），接下来LSLS命令让寄存器R1实现左移操作。至此，寄存器R1存储GPIOA输出数据寄存器的起始地址，寄存器R2存储GPIOA输出数据寄存器的偏移地址。

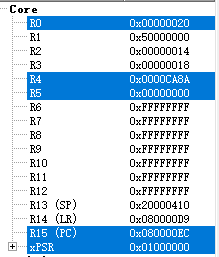
接下来是让寄存器R0中存储对应端口需要输出的值#0x20即GPOA\_PIN\_5高电平。然后利用存储器保存指令STR将寄存器R0中的数据发送到地址为[R1+R2]的GPIOA输出数据寄存器当中。

**问题⑦：**Cortex\_M0的CPU马上就执行完此代码了，即便有亮灯也看不到怎么办？

解决：为程序添加两个状态（亮和灭仅输出高低电平不同）并用循环增加延时（增加每个状态的持续时间），最后在外围套用无条件大循环（相当于C语言中的While（））让LED灯不停闪烁不能跳出大循环。程序代码如下图所示：







上述延时可以用寄存器传输指令MOVS移位指令LSLS减法指令SUBS比较指令CMP和条件转移指令BGE以及标号LEBELi组成。无条件的大循环则用无条件转移指令B和标号LEBEL组成，一旦进入大循环便不在退出。

由以上结果可知，Debug之后的每个寄存器值都是理想值。R1基地址，R2偏移地址，R0存放输出内容，R3存放移位位数（0x18就是16进制下的24）。 R4为0，R5不为0时输出低电平0x00，此时LD4灯灭。R5为0，R4不为0时输高电平0x20，此时LD4灯亮。

**问题⑧：**寄存器的运行是理想了，但是板子上的LED灯还是不亮。

问题原因猜想：可能是没有配置时钟使能（外设时钟使能寄存器）,也可能是GPIO的输出模式设置不对（端口配置高寄存器以及端口输出数据寄存器）。

解决：问题原因寻找确定，在C语言main函数部分查看他进行了那些操作。【启动引导代码让其跳转到main.c文件中的main函数执行，所以要想知道ARM处理器还执行了哪些操作，应该看main函数中开头到其执行输出while（1）之间的代码,此部分必定是初始化设置】

int main(void)

{

HAL\_Init(); //硬件抽象层初始化

SystemClock\_Config(); //系统时钟配置

MX\_GPIO\_Init(); //（GPIO）通用输入输出初始化

while (1)

{

HAL\_GPIO\_TogglePin(GPIOA,GPIO\_PIN\_5);

HAL\_Delay(100);

}

}

void SystemClock\_Config(void) //具体的系统时钟初始化

{

RCC\_OscInitTypeDef RCC\_OscInitStruct = {0}; //定义结构体并初始化

RCC\_ClkInitTypeDef RCC\_ClkInitStruct = {0};

HAL\_PWREx\_ControlVoltageScaling(PWR\_REGULATOR\_VOLTAGE\_SCALE1);

//设置调压器输出电压级别1

RCC\_OscInitStruct.OscillatorType = RCC\_OSCILLATORTYPE\_HSI;

//时钟源为内部HSL高速时钟 【HSE为外部时钟】

RCC\_OscInitStruct.HSIState = RCC\_HSI\_ON; //打开HSI

RCC\_OscInitStruct.HSIDiv = RCC\_HSI\_DIV1; //HSI VCO输入时钟分频系数

RCC\_OscInitStruct.HSICalibrationValue = RCC\_HSICALIBRATION\_DEFAULT;

//HSI校准调整值

RCC\_OscInitStruct.PLL.PLLState = RCC\_PLL\_NONE; //不打开PLL

if (HAL\_RCC\_OscConfig(&RCC\_OscInitStruct) != HAL\_OK)

//初始化

{

Error\_Handler();

}

RCC\_ClkInitStruct.ClockType=RCC\_CLOCKTYPE\_HCLK|RCC\_CLOCKTYPE\_SYSCLK|RCC\_CLOCKTYPE\_PCLK1; //选定时钟源并且配置时钟

RCC\_ClkInitStruct.SYSCLKSource = RCC\_SYSCLKSOURCE\_HSI;

//设置系统时钟源

RCC\_ClkInitStruct.AHBCLKDivider = RCC\_SYSCLK\_DIV1;

//AHB分频系数为1

RCC\_ClkInitStruct.APB1CLKDivider = RCC\_HCLK\_DIV1;

//APB1分频系数为1

if (HAL\_RCC\_ClockConfig(&RCC\_ClkInitStruct,FLASH\_LATENCY\_0)!=HAL\_OK)

//初始化并同时设FLASH的延迟周期为0

{

Error\_Handler();

}

}

//当同时编译多个文件时，所有未加static前缀的全局变量和函数都具有全局可见性，其它的源文件也能访问。

未加static的全局变量，在符号表中是global符号，其他目标文件可见，这样的符号是要参与符号解析的。加了static之后，是local符号，其他目标文件不可见，只在当前文件中可见，不参与符号解析过程。所以多个源文件可以定义同名的static全局变量，不会产生重定义错误。

static void MX\_GPIO\_Init(void) //具体的通用输入输出的初始化

{

GPIO\_InitTypeDef GPIO\_InitStruct = {0}; //定义一个GPIO\_InitTypeDef结构体

\_\_HAL\_RCC\_GPIOA\_CLK\_ENABLE(); //开启LED相关的GPIO外设时钟

HAL\_GPIO\_WritePin(GPIOA, GPIO\_PIN\_5, GPIO\_PIN\_RESET);

//先关闭LED灯

GPIO\_InitStruct.Pin = GPIO\_PIN\_5; //配置GPIOA管脚

GPIO\_InitStruct.Mode = GPIO\_MODE\_OUTPUT\_PP;

GPIO\_InitStruct.Pull = GPIO\_NOPULL;

GPIO\_InitStruct.Speed = GPIO\_SPEED\_FREQ\_LOW;

HAL\_GPIO\_Init(GPIOA, &GPIO\_InitStruct);

//根据****GPIO\_InitTypeDef****结构体中的****Pin****来找到对应的要操作的针脚位置而****Pin****中储存的值为【一堆管脚定义·上方图片】

}

void Error\_Handler(void) //死循环函数

{

while (1)

{ }

}

#ifdef USE\_FULL\_ASSERT

void assert\_failed(uint8\_t \*file, uint32\_t line)

{

}

#endif

【附录】已经经过测试的GPIO函数

HAL\_GPIO\_WritePin(GPIOA,GPIO\_PIN\_5,GPIO\_PIN\_SET); //输出高电平亮

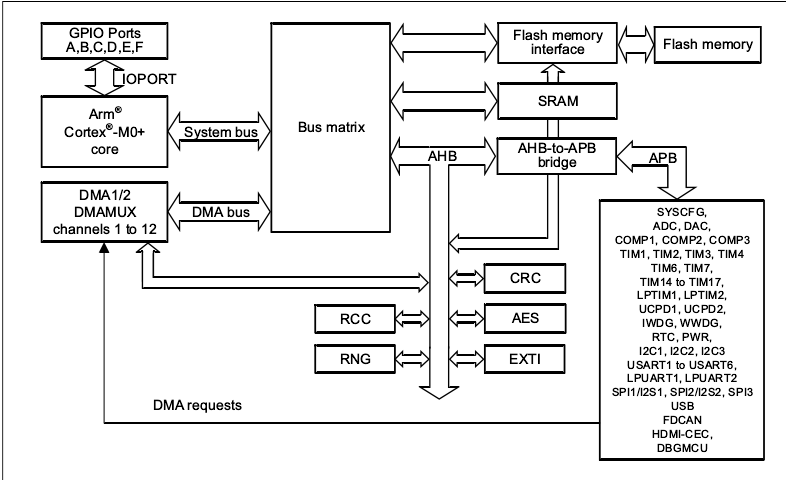
HAL\_GPIO\_WritePin(GPIOA,GPIO\_PIN\_5,GPIO\_PIN\_RESET); //输出低电平灭

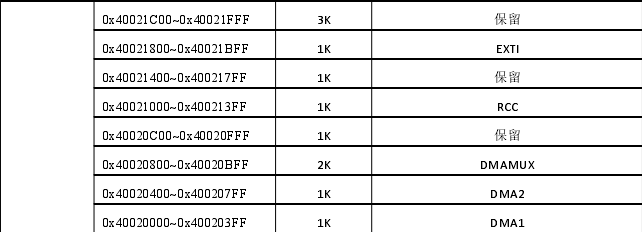
HAL\_GPIO\_TogglePin(GPIOA, GPIO\_PIN\_5); // 反转

由以上部分可知，初始化配置总共三大部分：硬件抽象层初始化函数、系统时钟配置函数和通用输入输出（GPIO）初始化函数。因为我去掉了main函数，所以，要想LED灯亮起来，就得用汇编语言在执行输出部分之前写出这些初始化配置。

三大函数作用分析：①硬件抽象层初始化配置了FLASH预存取功能以及SysTick中断时间1ms。当HAL\_Inint()函数被注释之后LED灯保持常亮而不会闪烁，分析原因，它卡在了HAL\_Delay()函数上，在文件stm32g0xx\_hal.c文件中找到的HAL\_Delay()函数中引用了HAL\_GetTick()函数获取SysTick中断时间间隔，所以会卡在HAL延迟函数。但是HAL\_Inint()函数被注释之后，不会影响代码的执行，HAL\_Delay()延时函数可以用for循环代替也可以实现相同效果。②系统时钟配置函数被注释掉时没有影响。③通用输入输出初始化函数MX\_GPIO\_Init()中执行以下部分：HAL\_RCC\_GPIOA\_CLK\_ENABLE开启LED相关的外设时钟，配置PA5管脚位置、设置为输出模式、PP类型、NOPULL、速度为低速。

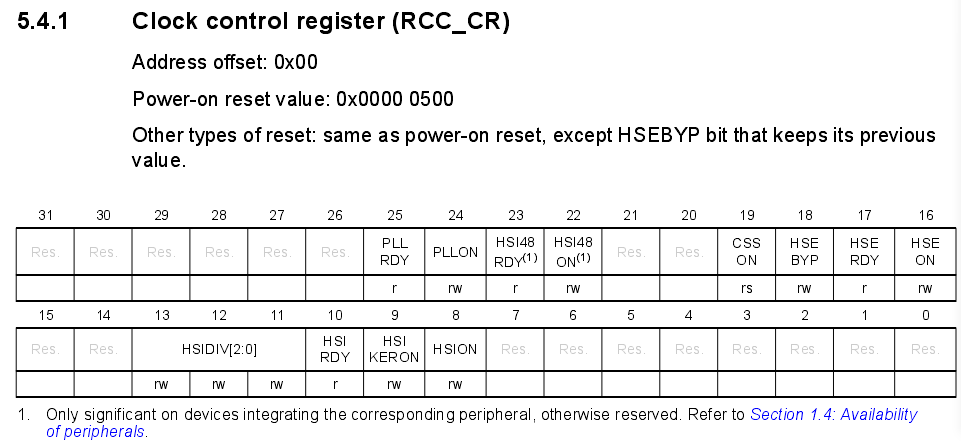
所以，总的来说要分为两大块，GPIO时钟配置和GPIO模式设置。那么这两个设置需要操作哪些寄存器呢？





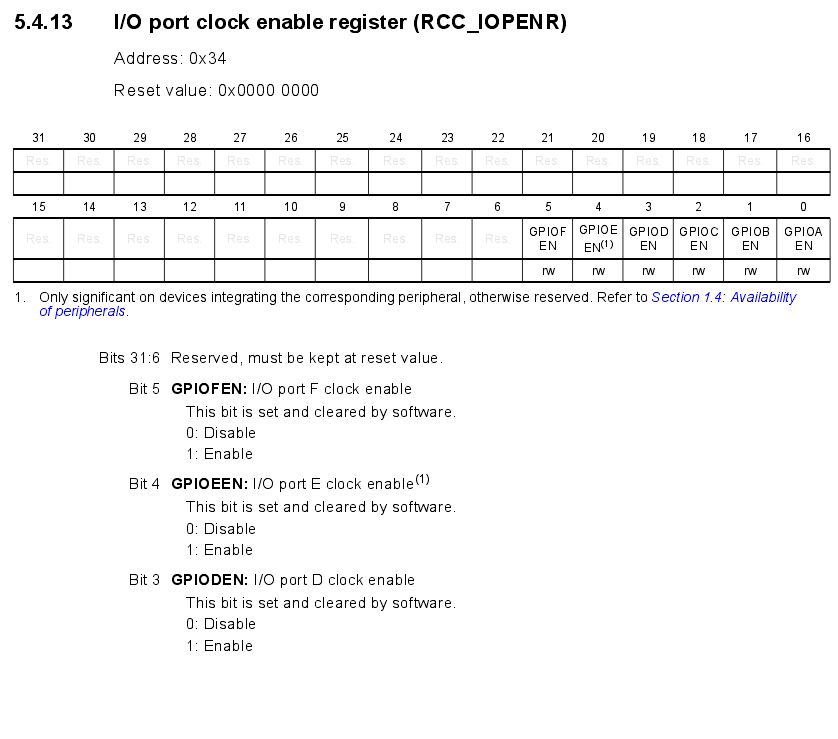
由上图可知，复位和时钟控制RCC的地址范围是：0x40021000~0x400213FF，其基址是0x40021000。

**时钟控制寄存器设置如下：**



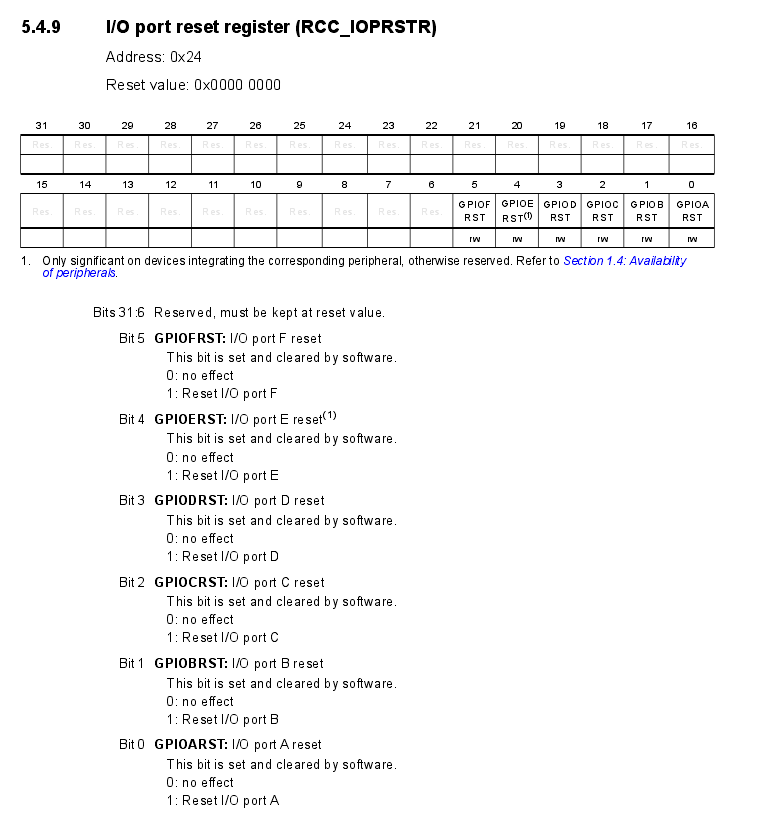
RCC基址是0x40021000该寄存器偏移地址为0x00要想使能时钟应该向该寄存器输入值#0x0500【需要移位】

I**/O端口时钟使能寄存器设置如下：**



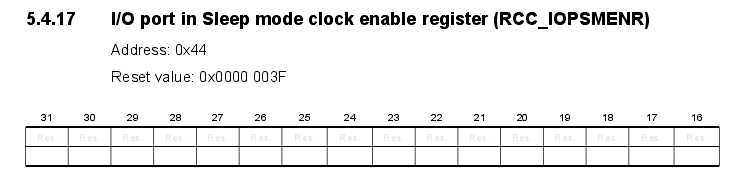
RCC基址是0x40021000该寄存器偏移地址为0x34要想使能GPIOA应该向该寄存器输入值#0x01

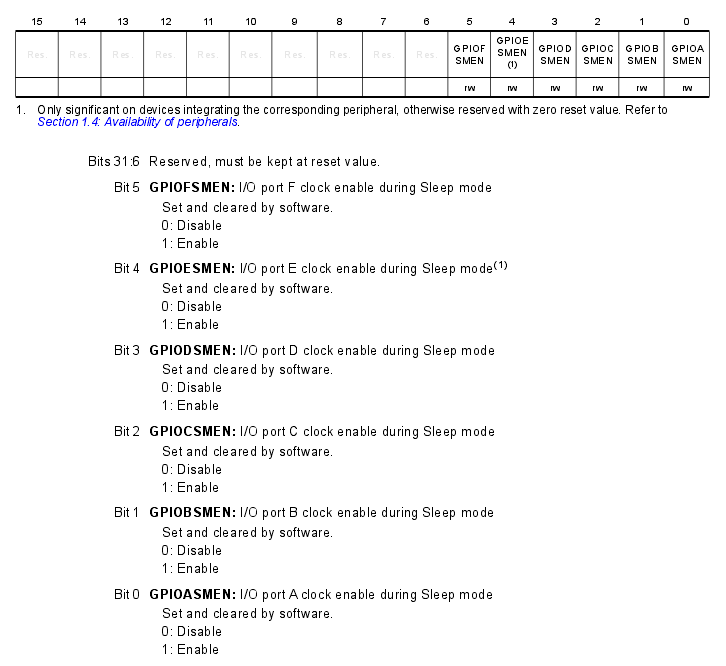
**I/O端口复位寄存器设置如下：**



暂时不对其进行操作。

**睡眠模式下的I/O端口时钟使能寄存器设置如下：**

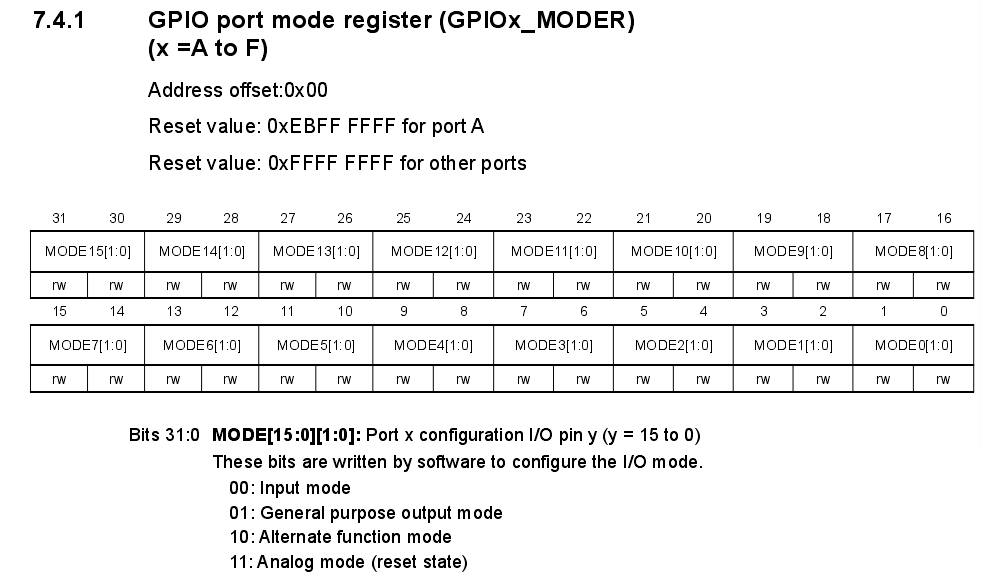




不进入睡眠模式暂不对其进行操作。

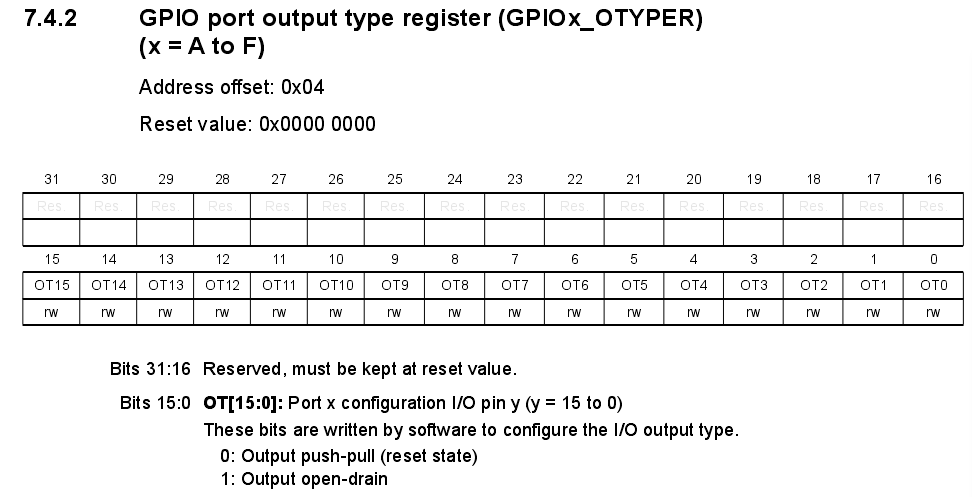
**以下讨论GPIO设置相关的寄存器。**

**GPIO端口模式寄存器设置如下：**



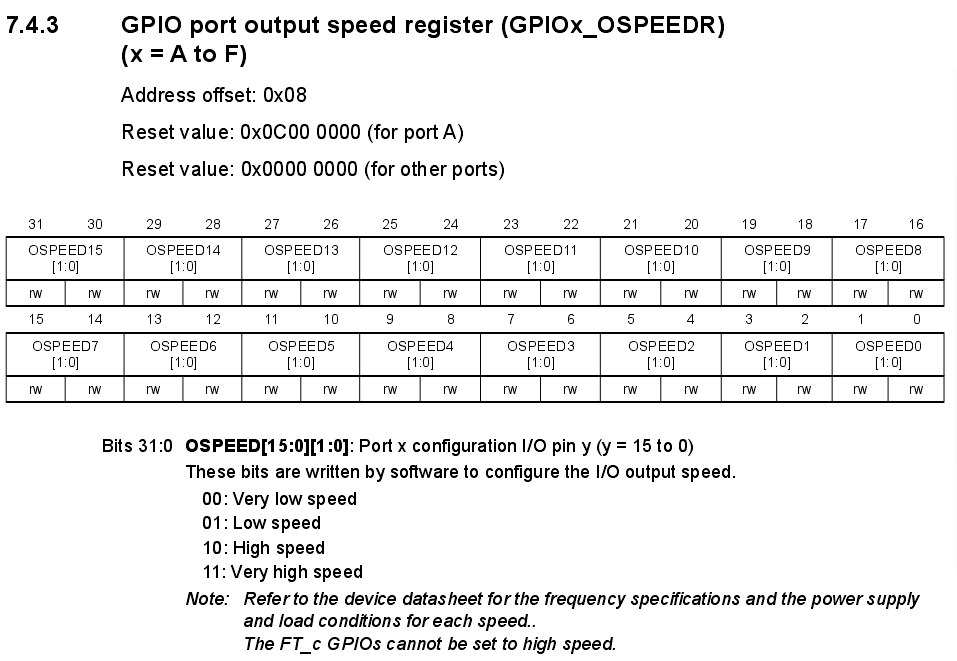
由之前结论知GPIOA的基址为0x50000000，该寄存器偏移地址为0x00。将PA5设置为通用输出模式即第11位和第10位分别为0、1即输入数据#0x0400【需要移位】

**GPIO端口输出类型寄存器设如下:**



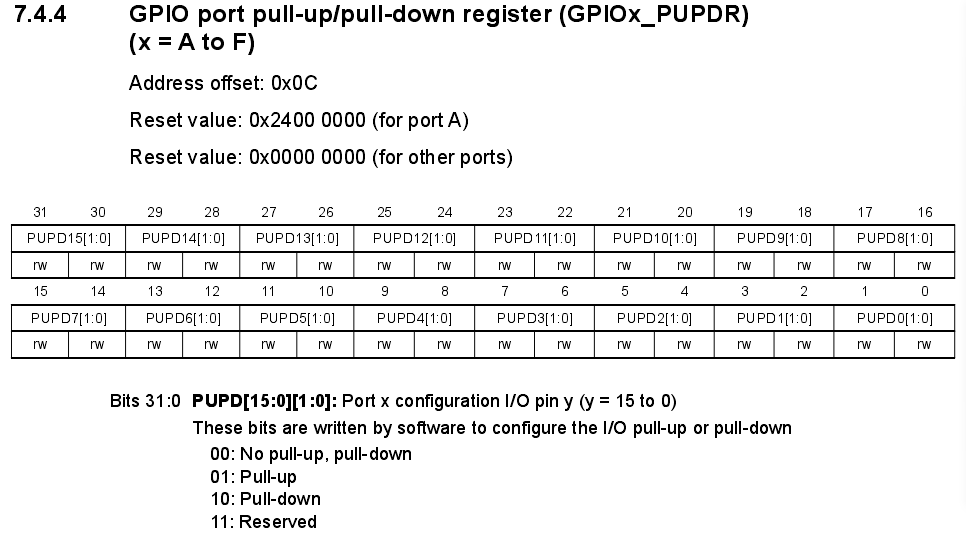
GPIOA的基址为0x50000000，该寄存器偏移地址为0x04。将PA5设置为PP类型即第5位为0即输入数据#0x00。

**GPIO端口输出速度寄存器设置如下：**



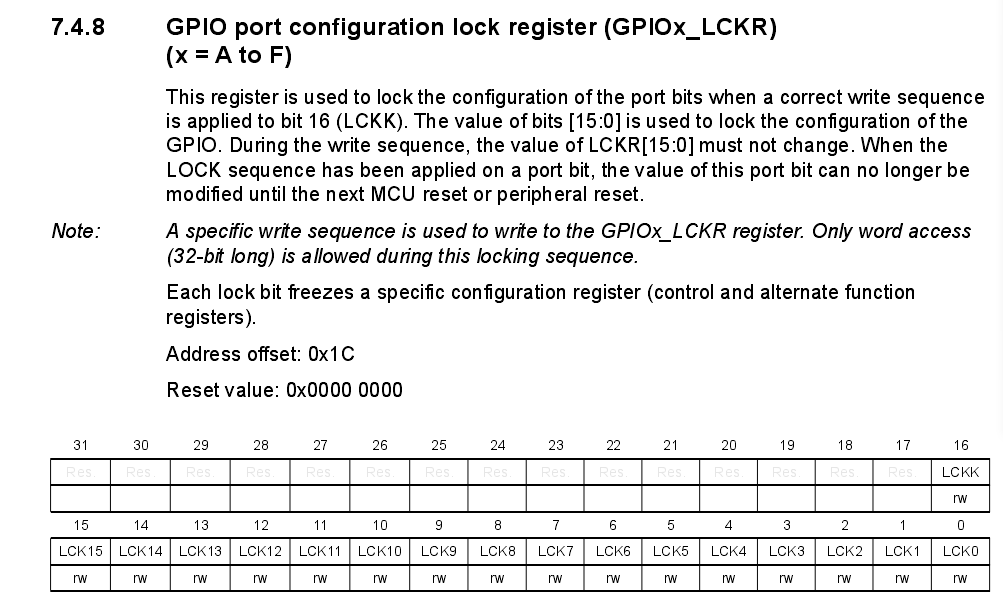
GPIOA的基址为0x50000000，该寄存器偏移地址为0x08。将PA5设置为低速输出即第11位和第10位分别0、1即输入数据#0x0400。

**GPIO端口PULL寄存器设置如下：**

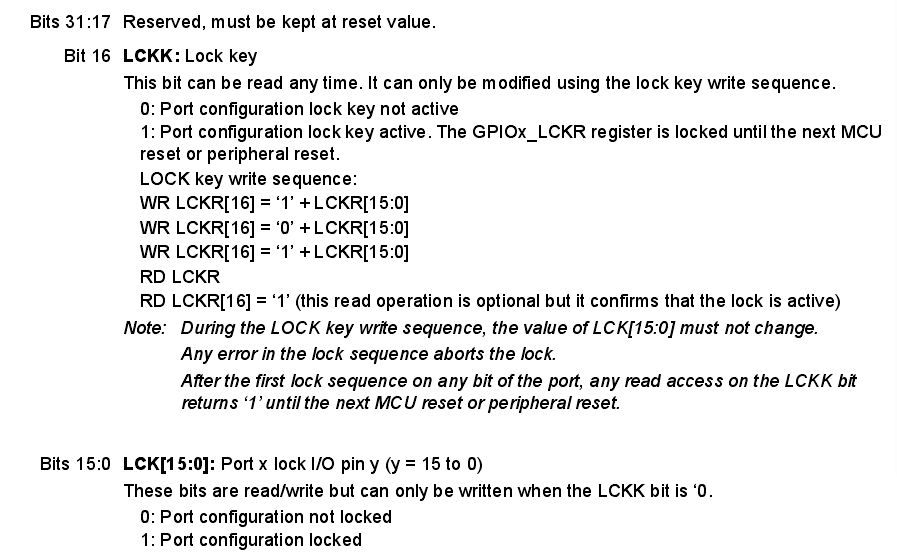


GPIOA的基址为0x50000000，该寄存器偏移地址为0x0C。将PA5设置为NOPULL即第11位和第10位分别0、0即输入数据#0x0000。

**GPIO端口配置锁定寄存器设置如下：**



GPIOA的基址为0x50000000，该寄存器偏移地址为0x1C。将PA5设置解锁即第5位为0即输入数据#0x00。



问题⑨：成功实现之后的总结。

解决：在何老师的指导下，我换了一种分析方法：先用C语言主函数做出成果，然后Debug一下，在调试界面看各个相关寄存器的值以及它更新的先后顺序，然后根据此框架撰写汇编语言初始化代码。总结我之前走的弯路：不能光靠自己想想，写的代码必须有理有据。以后分析寄存器，要把多种资源相结合（教材开发指南、硬件数据手册、CSDN查资料、软件参数分析）【重点：有些寄存器保持初始值就不需要再单独设置了Clock control register、GPIO port configuration lock regester、GPIO port output type register、GPIO port pull\_up/pull\_down regester】

**成果分享：**

**第一步**，在启动引导代码中注释掉跳转到C语言主函数的语句

；BX R0。

**第二步**，初始化复位和时钟控制RCC的相关寄存器（一部分保持默认值不做改变），

寄存器I/O port clock enable register(RCC\_IOPENR)

MOVS R1, #0x00000040

MOVS R3, #12

LSLS R1, R3

MOVS R3, #0x00000021

ADDS R1, R3

MOVS R3, #12

LSLS R1, R3 ;Base address #40021000

MOVS R2, #0x34 ;Address offset #0x34

MOVS R0, #0x01 ;Set value#0x01

STR R0, [R1,R2]

**第三步**，初始化GPIO输出模式，寄存器GPIO port mode regester

MOVS R1, #0x00000050

MOVS R3, #24

LSLS R1, R3 ;Base address #50000000

MOVS R2, #0x00 ;Address offset #0x00

MOVS R0, #0xEB

MOVS R3, #8

LSLS R0, R3

MOVS R3, #0xFF

ADDS R0, R3

MOVS R3, #8

LSLS R0, R3

MOVS R3, #0xF7

ADDS R0, R3

MOVS R3, #8

LSLS R0, R3

MOVS R3, #0xFF

ADDS R0, R3 ;Set value #0xEBFFF7FF

STR R0, [R1,R2]

**第四步**，添加设计代码

LEBEL ;The big cycle,LD4 twinlle

MOVS R4, #0xFF ;The number one cycle,LD4 light

MOVS R3, #8

LSLS R4, R3

LEBEL1

MOVS R1, #0x00000050

MOVS R3, #24

LSLS R1, R3

MOVS R2, #0x14

MOVS R0, #0x20

STR R0, [R1,R2] ;failing

SUBS R4, #1

CMP R4, #1

BGE LEBEL1

MOVS R5, #0xFF ;The number two cycle,LD4 don't light

MOVS R3, #8

LSLS R5, R3

LEBEL2

MOVS R1, #0x00000050

MOVS R3, #24

LSLS R1, R3

MOVS R2, #0x14

MOVS R0, #0x00

STR R0, [R1,R2]

SUBS R5, #1

CMP R5, #1

BGE LEBEL2

B LEBEL