第3单元 S800的嵌入式实验

本章节参考资料：

A. 自编讲义《嵌入式系统实验教程》

B. Tiva™ TM4C1294NCPDT Microcontroller Data Sheet

C. TivaWare™ Peripheral Driver Library User's Guide

D. S800板介绍V0.65

第3章 S800的嵌入式实验

- **3.1 实验一 时钟选择与 GPIO 实验**

  (ref. B-chapter5, 10, C-chapter26,14)

- **3.2 实验二 I2C GPIO扩展及SYSTICK中断实验**

  (ref. B-chapter3,18, C-chapter16,17,28)

- **3.3 实验三 UART串行通讯口实验**

  (ref. B-chapter16, C-chapter30)

实验一 时钟选择与 GPIO 实验

- 实验目的
  - 了解 TM4C1294NCPDT MCU系统控制功能
  - 理解 MCU 中的时钟信号，了解不同时钟对电源消耗的不同
  - 掌握 GPIO 的工作原理，能够结合 GPIO 的输入与输出功能进行实验

实验一 时钟选择与 GPIO 实验

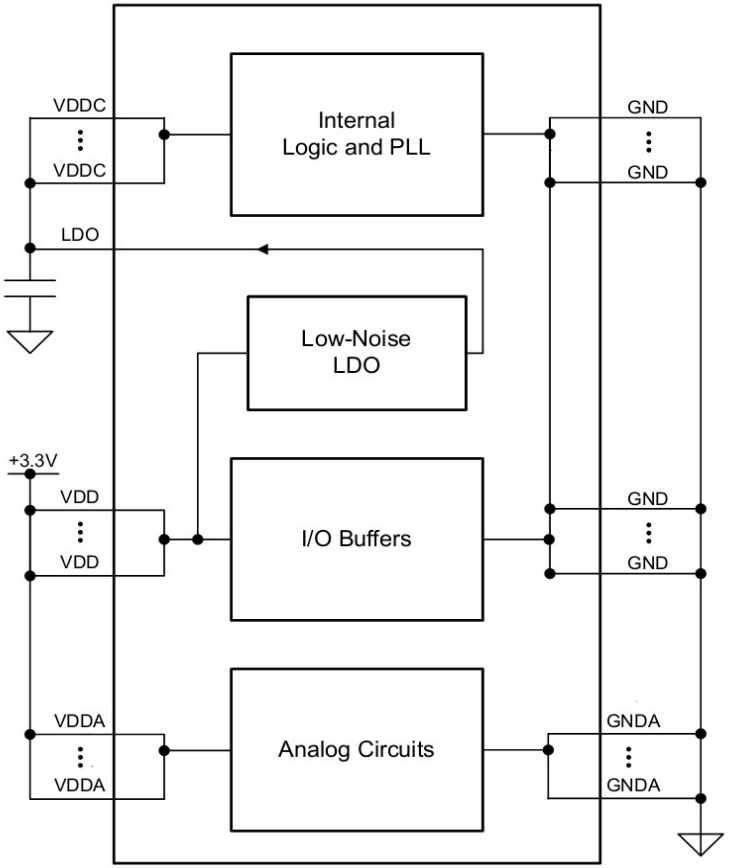
- 预备知识
  - 3.1.1 系统控制（ref. B-chapter5, C-chapter26）
  - 3.1.2 通用输入/输入端口GPIO（ref. B-chapter10, C-chapter14）
  - 3.1.3 按键消抖

3.1.1 系统控制

- TM4C1294NCPDT MCU的系统控制决定设备的整体操作，包括设备的时钟、使能的外设集、设备的配置以及复位等以下几个部分：
  - 电源控制
  - 复位控制
  - 时钟控制
  - 外设控制
  - 运行模式（功耗）控制
  - 杂项功能（延时）控制
  - …等

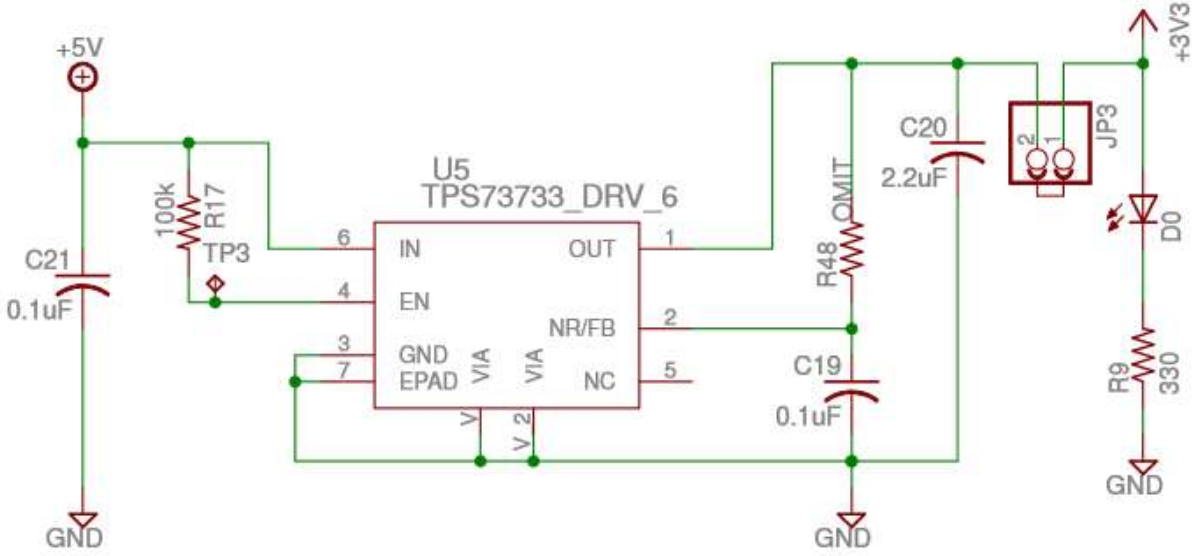
TM4C1294NCPDT MCU电源结构

- 3.3V VDD 是微控制器的工作电压
  - 对外的GPIO等接口采用VDD作为驱动电源
- VDDA 是模拟电源，供ADC使用
  - 独立的VDDA电源可以提高转换精度，避免主电源干扰
  - 将 VDDA 和 VDD 并联易于它们的同步上电或掉电
- VDDC 和 LDO 是内核电源，额定工作电压0.9V~1.2V，由内置的LDO稳压器提供，供内部数字部分工作



S800实验板的电源方案

- MCU的工作电压VDD由外部 5V 电压经过一个低噪音LDO芯片TPS73733降压至 3.3V 供给
  - LDO（Low Drop-Out）：一种线性直流电源稳压器
  - LDO 特点：输入与输出之间的压差低，能达到数百毫伏。具有降低功耗、缩小体积等优点



S800实验板的输入电源

- 外部 5V电压的输入来源共有三个，由电源跳帽JP1设定：

Power Control Jumper:

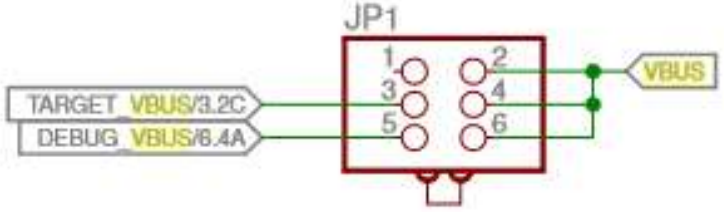
1) To power from Debug install jumper on pins 5 - 6

2) To power from Target USB install jumper on pins 3 - 4

3) To power from BoosterPack 5V install jumper on pins 1 - 2
This is also the off position if BoosterPack does not supply power

When powered from BoosterPack TPS2052B does not provide current limit protection.

When powered by BoosterPack, USB host mode does not supply power to connected devices

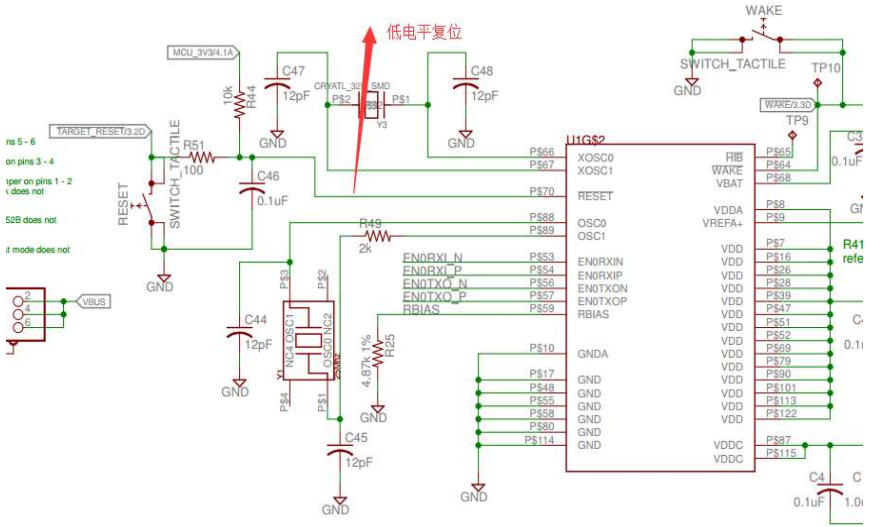


1. 板载ICDI仿真器 5V 输入
2. MICRO-USB OTG 5V 输入
3. 外部 5V 供电插座

复位系统

- 复位源：

1. 上电复位 Power-on Reset（POR）
2. 外部复位输入管脚（RST#），低电平有效
3. 内部掉电检测 Brown-out（BOR）
4. 软件复位（执行函数SysCtlReset()引发的复位）
5. 看门狗（Watchdog）复位
6. 休眠模块事件
7. 主振荡器校验失败(即MOSC失败)



S800复位电路

运行模式（功耗）控制

- TivaWare系统四种运行模式
  - Run mode，正常运行模式，处理器主动执行代码
  - Sleep mode，外设时钟频率不变，处理器和存储子系统不再执行代码
  - Deep-Sleep mode，外设时钟频率降低，处理器停止运行
  - Hibernation mode，只有休眠模块工作，其他都停止

时钟系统

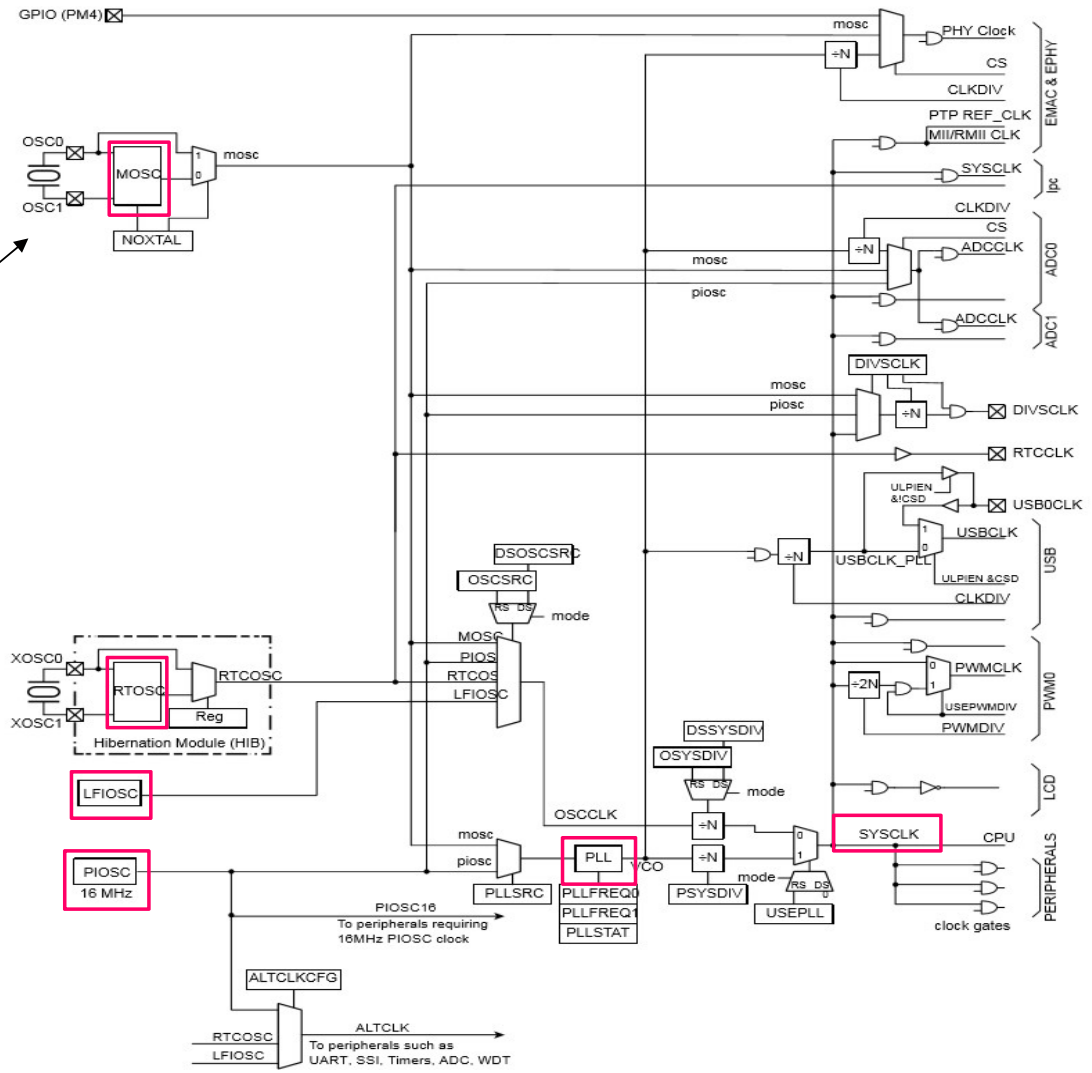
- TM4C1294NCPDT MCU中有多个时钟源可以使用，包括：
  - 高精度内部振荡器（PIOSC），频率16MHz。上电时默认使用
  - 主振荡器（MOSC），频率5~25MHz（S800为25MHz）。上电时Disabled
  - 低精度内部振荡器（LFIOSC），33KHz，用于深度休眠省电模式
  - 休眠模块RTC振荡器时钟源（RTCOSC），32.768KHz或更低

Clock Source	Drive PLL Capability?	PLL Enabled, RSCLKCFG Bit Encodings	SysClk generation capability?	SysClk generation enabled, RSCLKCFG Bit Encodings
Precision Internal Oscillator (PIOSC)	Yes	USEPLL = 1, PLLSRC = 0x0	Yes	USEPLL = 0, OSCSRC = 0x0
Main Oscillator (MOSC)	Yes	USEPLL = 1, PLLSRC = 0x3	Yes	USEPLL = 0, OSCSRC = 0x3
Low Frequency Internal Oscillator (LFIOSC)[a]	No	-	Yes	USEPLL = 0, OSCSRC = 0x2
Hibernation Module RTC Oscillator (RTCOSC). 32.768-kHz Oscillator or HIB LFIOSC	No	-	Yes	USEPLL = 0, OSCSRC = 0x4

系统时钟逻辑

S800外部主振荡器（MOSC）为 25M

PLL(Phase Locked Loop, 锁相环）是一种利用反馈控制原理实现的频率及相位的同步技术，其作用是将电路输出的时钟与其参考时钟保持同步



外设控制

- 外设控制是指对外设的使能、除能、复位等操作。系统所有片上外设只有在使能后才可以工作。
- TivaWare外设控制函数包括：
  - SysCtlPeripheralEnable()：使能片上外设；
  - SysCtlPeripheralDisable()：禁用片上外设，以节省功耗
  - SysCtlPeripheralReady()：确认外设是否准备好
  - SysCtlPeripheralReset()：对外设复位

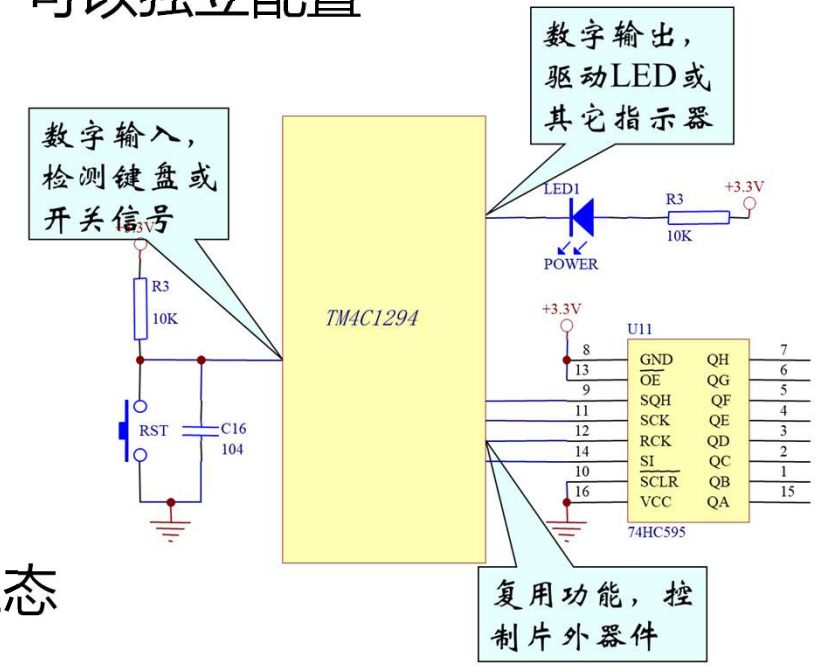
```
void SysCtlPeripheralEnable (uint32_t ui32Peripheral)
bool SysCtlPeripheralReady (uint32_t ui32Peripheral)
其中， ui32Peripheral为指定设备
```

- TivaWare外设控制的ui32Peripheral参数定义在driverlib/sysctl.h

```
#define SYSCTL_PERIPH_ADC0      0xf0003800  // ADC 0
#define SYSCTL_PERIPH_ADC1      0xf0003801  // ADC 1
#define SYSCTL_PERIPH_CAN0      0xf0003400  // CAN 0
#define SYSCTL_PERIPH_CAN1      0xf0003401  // CAN 1
#define SYSCTL_PERIPH_COMP0     0xf0003c00  // Analog Comparator Module 0
#define SYSCTL_PERIPH_EMAC0     0xf0009c00  // Ethernet MAC0
#define SYSCTL_PERIPH_EPHY0     0xf0003000  // Ethernet PHY0
#define SYSCTL_PERIPH_EPI0      0xf0001000  // EPI0
#define SYSCTL_PERIPH_GPIOA     0xf0000800  // GPIO A
#define SYSCTL_PERIPH_GPIOB     0xf0000801  // GPIO B
#define SYSCTL_PERIPH_GPIOC     0xf0000802  // GPIO C
#define SYSCTL_PERIPH_GPIOD     0xf0000803  // GPIO D
#define SYSCTL_PERIPH_GPIOE     0xf0000804  // GPIO E
#define SYSCTL_PERIPH_GPIOF     0xf0000805  // GPIO F
#define SYSCTL_PERIPH_GPIOG     0xf0000806  // GPIO G
                    ……
#define SYSCTL_PERIPH_WTIMER4   0xf0005c04  // Wide Timer 4
#define SYSCTL_PERIPH_WTIMER5   0xf0005c05  // Wide Timer 5
```

3.1.2 GPIO 通用输入/输入端口

- TM4C1294微控制器有15个物理 **GPIO**（General Purpose Input/Output）模块，即端口A、B、C、D、E、F、G、H、J、K、L、M、N、P、Q
- 每个端口最多8个引脚，共90个GPIO引脚，可以独立配置
  - 方向：输入或输出，缺省为输入
  - 模式：
    - 数字输入：弱上拉/下拉输入
    - 数字输出：推挽(缺省) 、开漏、斜率控制等
    - 模拟输入
    - 复用功能
- 除特殊指定之外，所有GPIO口复位后为三态

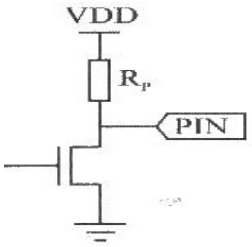


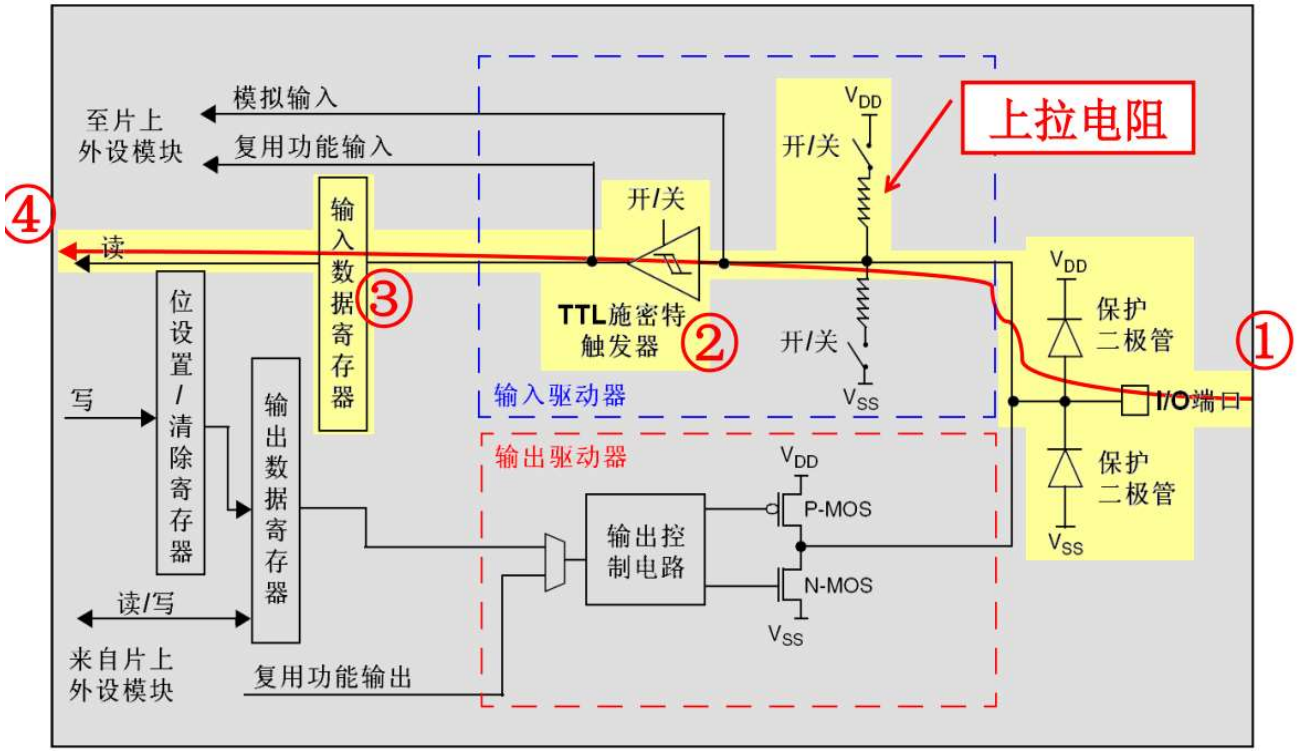
GPIO模式

GPIO	TM4C129x Series
General Purpose Input （输入）	Floating（浮空） Pull-Up (GPIO_PIN_TYPE_STD_WPU) Pull-Down (GPIO_PIN_TYPE_STD_WPD) WAKE (GPIO_PIN_TYPE_WAKE_HIGH) (GPIO_PIN_TYPE_WAKE_LOW)
General Purpose Output （输出）	Push-Pull (GPIO_PIN_TYPE_STD) Open-Drain (GPIO_PIN_TYPE_OD) Push-Pull+Pull-Up (GPIO_PIN_TYPE_STD_WPU) Push-Pull+Pull-Down (GPIO_PIN_TYPE_STD_WPD)
Analog（模拟）	Analog (GPIO_PIN_TYPE_ANALOG)
Alternate Function Output （复用功能）	Push-Pull (GPIO_PIN_TYPE_STD) Open-Drain (GPIO_PIN_TYPE_OD) Push-Pull+Pull-Up (GPIO_PIN_TYPE_STD_WPU) Push-Pull+Pull-Down (GPIO_PIN_TYPE_STD_WPD)

GPIO不同模式下的等效电路

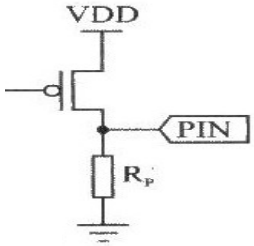
- 输入上拉模式

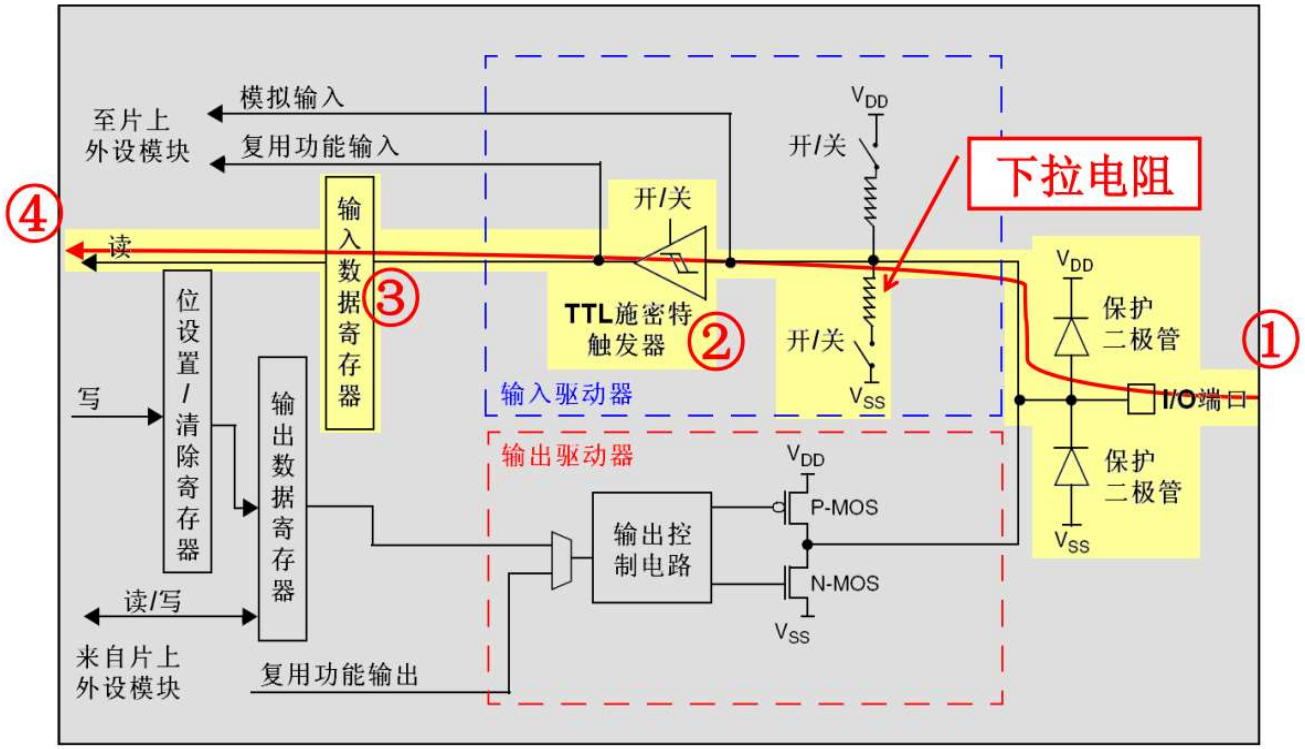




GPIO不同模式下的等效电路

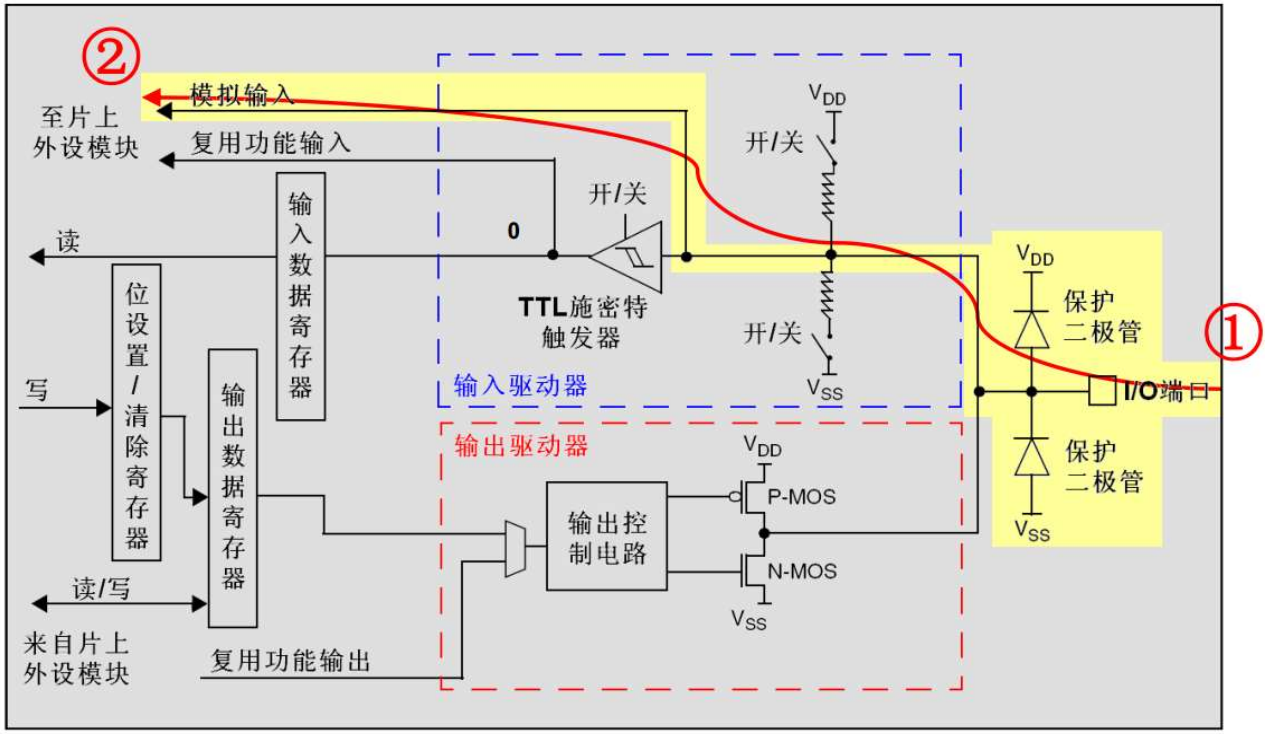
- 输入下拉模式





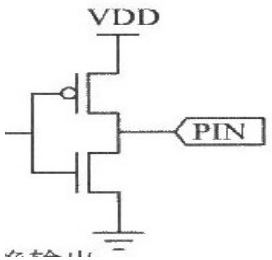
GPIO不同模式下的等效电路

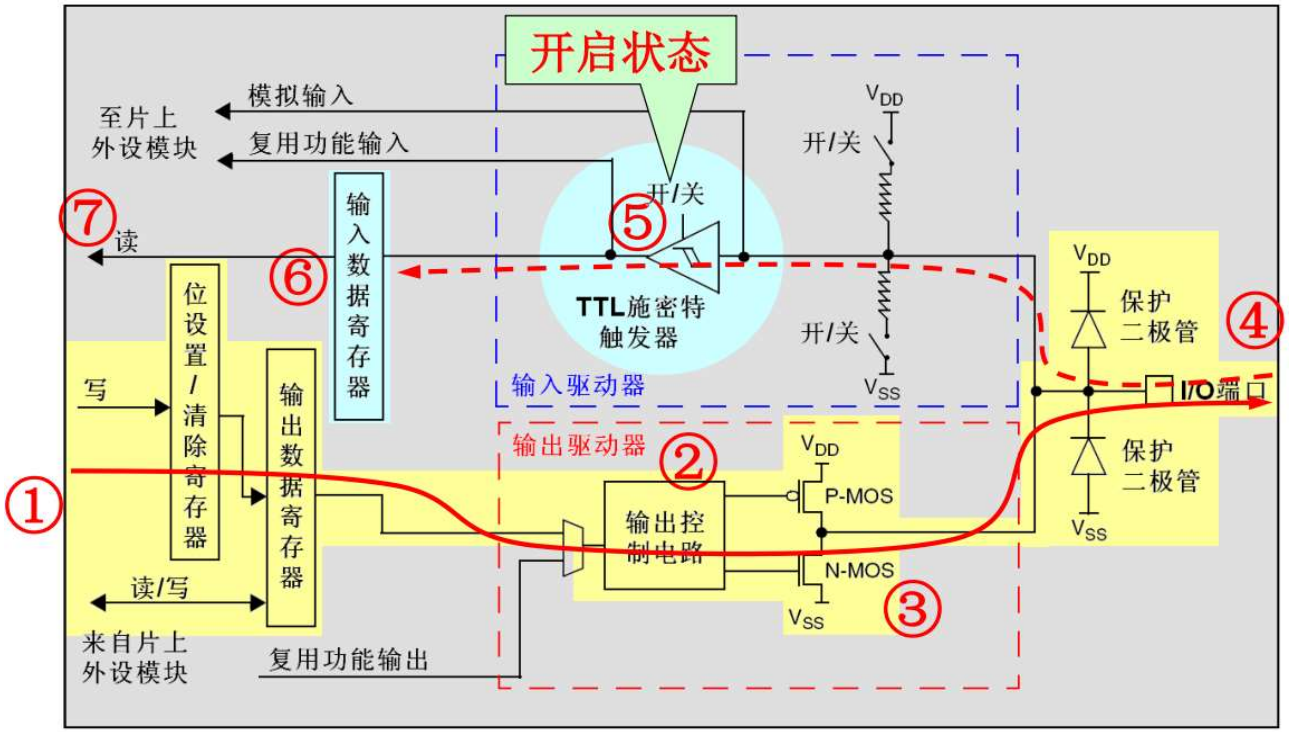
- 模拟输入



GPIO不同模式下的等效电路

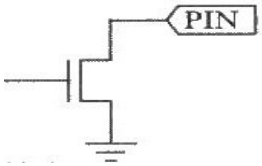
- 推挽输出

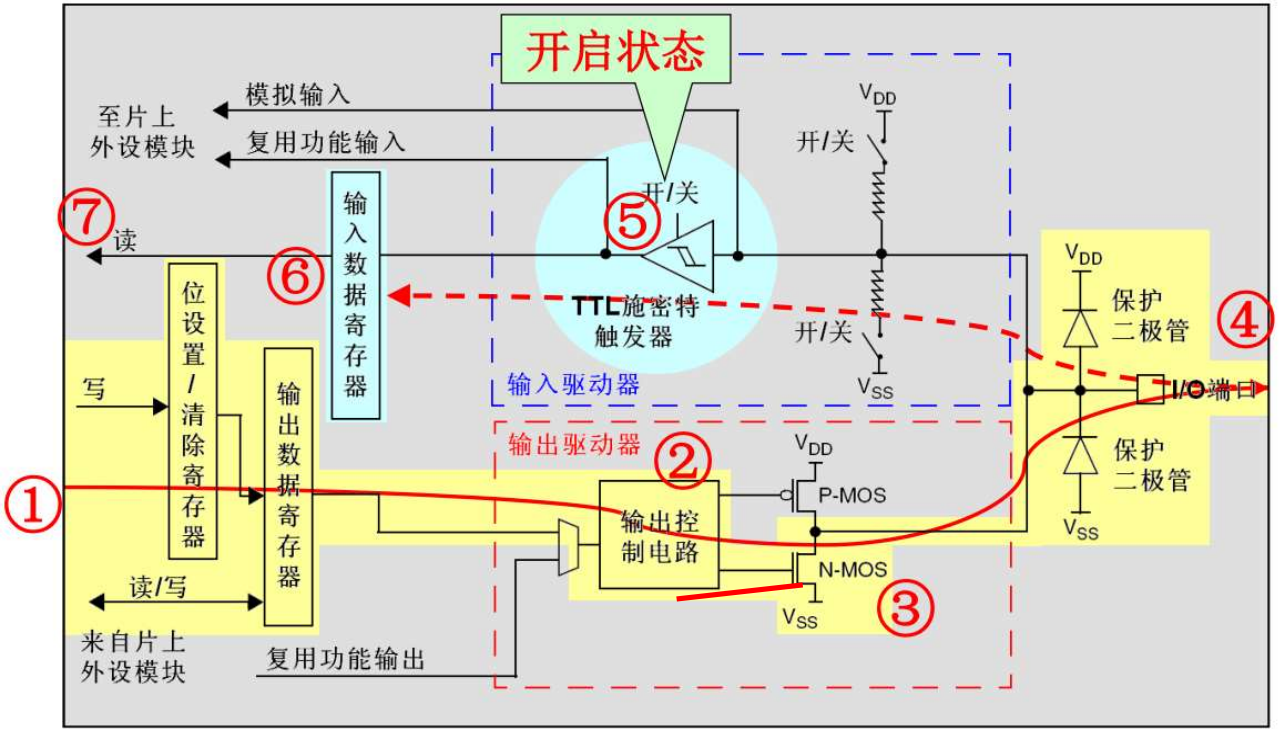




GPIO不同模式下的等效电路

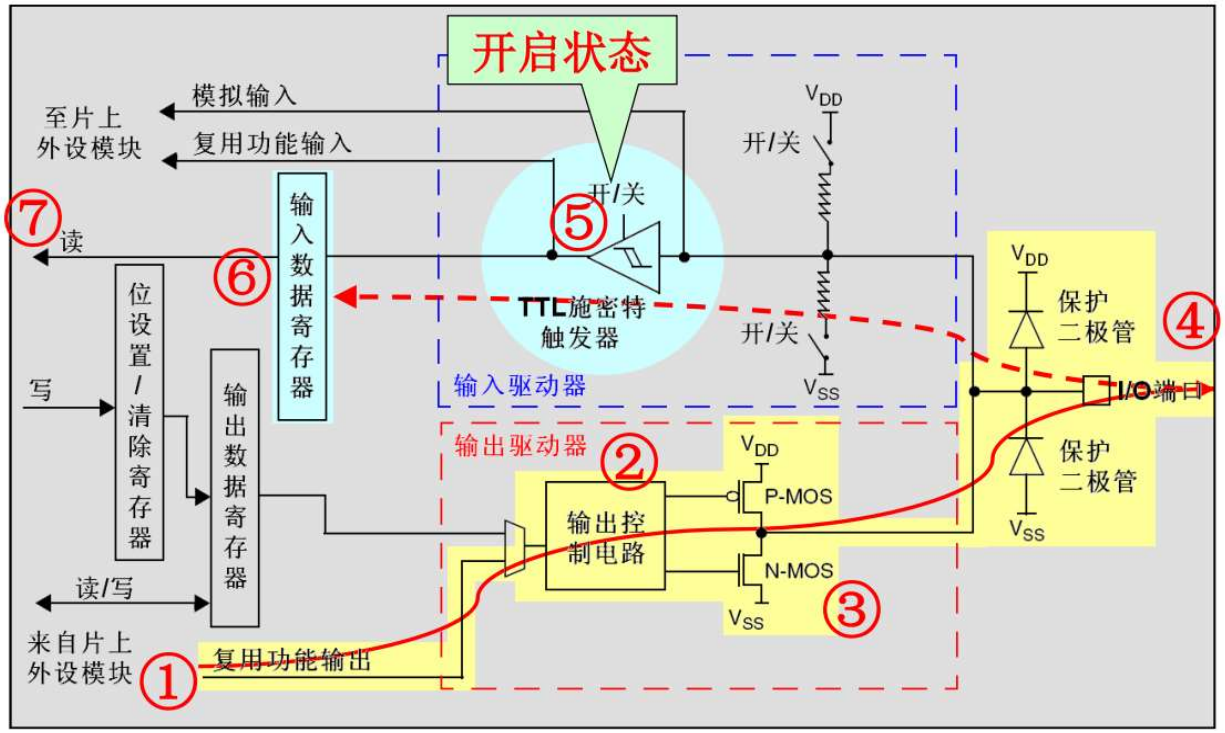
- 开漏输出





GPIO不同模式下的等效电路

- 推挽输出复用功能



GPIO端口地址映射

- GPIO A-H,J端口可经由AHB和APB两种总线访问，K-P,Q端口由AHB：
  - AHB (Advanced High-Performance Bus) 提供更好的访问性能

GPIO 端口	基本地址(AHB)	GPIO 端口	基本地址(AHB)
A	0x4005.8000	J	0x4006.0000
B	0x4005.9000	K	0x4006.1000
C	0x4005.A000	L	0x4006.2000
D	0x4005.B000	M	0x4006.3000
E	0x4005.C000	N	0x4006.4000
F	0x4005.D000	P	0x4006.5000
G	0x4005.E000	Q	0x4006.6000
H	0x4005.F000		

GPIO寄存器

偏移量	名称	读写类型	复位值	功能
0x000	GPIODATA	R/W	0x0000.0000	GPIO数据
0x400	GPIODIR	R/W	0x0000.0000	GPIO方向
0x404	GPIOIS	R/W	0x0000.0000	GPIO中断检测
0x408	GPIOIBE	R/W	0x0000.0000	GPIO中断双边沿检测
0x40C	GPIOIEV	R/W	0x0000.0000	GPIO中断事件
0x410	GPIOIM	R/W	0x0000.0000	GPIO中断屏蔽
0x414	GPIORIS	RO	0x0000.0000	GPIO原始中断状态
0x418	GPIOMIS	RO	0x0000.0000	GPIO屏蔽后的中断状态
0x41C	GPIOICR	W1C	0x0000.0000	GPIO中断清除
0x420	GPIOAFSEL	R/W	-	GPIO备用功能选择
0x500	GPIODR2R	R/W	0x0000.00FF	GPIO 2mA驱动选择
0x504	GPIODR4R	R/W	0x0000.0000	GPIO 4mA驱动选择
0x508	GPIODR8R	R/W	0x0000.0000	GPIO 8mA驱动选择
0x50C	GPIOODR	R/W	0x0000.0000	GPIO开漏选择
0x510	GPIOPUR	R/W	-	GPIO上拉选择
0x514	GPIOPDR	R/W	0x0000.0000	GPIO下拉选择
0x518	GPIOSLR	R/W	0x0000.0000	GPIO斜率控制选择

……

S800实验板上的部分GPIO管脚功能

红板

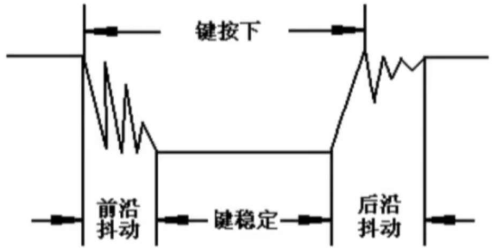
名称	对应管脚	说明
RESET	RESET	TM4C1294NCPDT芯片复位按键，低有效
WAKE	WAKE	从睡眠模式唤醒按键，低有效
USR_SW1	PJ0	用户输入按键，低有效
USR_SW2	PJ1	用户输入按键，低有效
D0		3.3V电源指示，绿LED，高有效
D1	PN1	用户控制绿LED，高有效
D2	PN0	用户控制绿LED，高有效
D3	PF4	用户控制绿LED，高有效
D4	PF0	用户控制绿LED，高有效

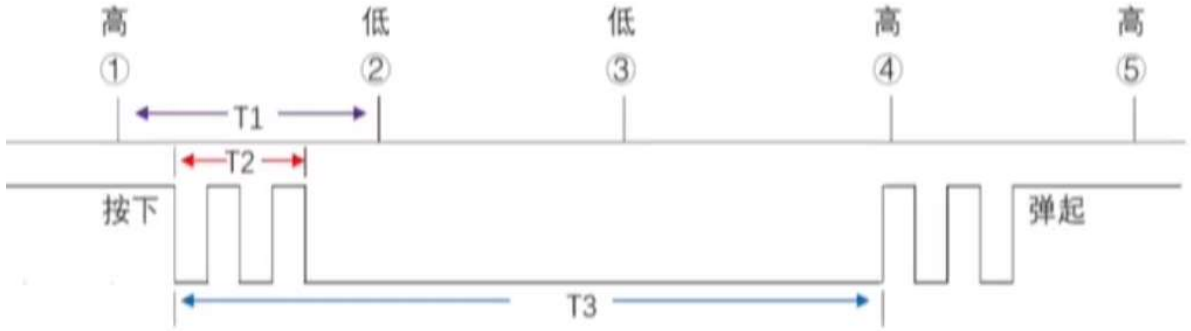
蓝板

名称	对应管脚	说明
SW1-SW8	TCA6424-P01~P08	TCA6424 I2C展GPIO芯片P0口，低有效
LED1-LED8	PCA9557-P0-P7	PCA9557 I2C展GPIO芯片P0口，低有效
LED_M0	PF0	用户控制LED，高有效
LED_M1	PF1	用户控制LED，高有效
LED_M2	PF2	用户控制LED，高有效
LED_M3	PF3	用户控制LED，高有效
D10		3.3V电源指示，红LED，高有效

3.1.3 按键消抖

- 机械特性决定按键存在抖动，正常情况按键持续时间50ms~200ms，抖动时间10ms以内
- 定时检测消抖方法
  - 通过相邻两次读取按键状态，前高后低——按下，前低后高——弹起
  - 要求 T2<T1<T3（T1-读取间隔时间，T2-抖动时间，T3-按键持续时间）





实验一 时钟选择与 GPIO 实验

- 实验内容
  - 例程1-1.c：分别使用内部16M的PIOSC时钟，外部25M的MOSC时钟，以及PLL时钟进行GPIO-PF0的闪烁
  - 编程要点
    1. **程序结构**
    2. 初始化：系统时钟设置、 GPIO配置
    3. 任务1：GPIO的读写
    4. 任务2：GPIO控制LED灯的亮灭

```
int main(void)
{
    uint32_t delay_time, key_value;

    S800_Clock_Init();
    S800_GPIO_Init();

    while(1)
    {
        key_value = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_0);

        if (key_value == 0)                    //USR_SW1-PJ0 pressed
            delay_time = FASTFLASHTIME;
        else
            delay_time = SLOWFLASHTIME;

        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);// Turn on the LED.
        Delay(delay_time);
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);        // Turn off the LED.
        Delay(delay_time);
    }
}
```

C程序组织结构

- main函数
  - 初始化设置
  - 主循环轮询：无限循环，任务检测和处理
    - 明确需求，分解任务
    - 根据需求设置外设状态（**变与不变**）

    不变：确保外设状态只受特定指令影响

    变化：通过延时/顺序/中断操作让硬件产生变化

```
int main()
{
   xxxInit(); //初始化
   while (1) {
        task1();
        task2();
        …
   }
}
```

C程序结构分析

```
int main(void)
{
    uint32_t delay_time, key_value;

    S800_Clock_Init();
    S800_GPIO_Init();                                          //初始化设置

    while(1)                                                   //无限循环
    {
        key_value = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_0);   //task1按键检测

        if (key_value == 0)                    //USR_SW1-PJ0 pressed
            delay_time = FASTFLASHTIME;
        else
            delay_time = SLOWFLASHTIME;

        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);// Turn on the LED.
        Delay(delay_time);
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);        //task2 LED灯控制
        Delay(delay_time);
    }
}
```

实验一 时钟选择与 GPIO 实验

- 实验内容
  - 例程1-1.c：分别使用内部16M的PIOSC时钟，外部25M的MOSC时钟，以及PLL时钟进行GPIO-PF0的闪烁
  - 编程要点
    1. 程序结构
    2. **初始化：系统时钟设置、** GPIO配置
    3. 任务1：GPIO的读写
    4. 任务2：GPIO控制LED灯的亮灭

```
int main(void)
{
    uint32_t delay_time, key_value;

    S800_Clock_Init();
    S800_GPIO_Init();

    while(1)
    {
        key_value = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_0); //read the PJ0 key value

        if (key_value  == 0)                    //USR_SW1-PJ0 pressed
            delay_time = FASTFLASHTIME;
        else
            delay_time = SLOWFLASHTIME;

        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);     // Turn on the LED.
        Delay(delay_time);
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);            // Turn off the LED.
        Delay(delay_time);
    }
}
```

- 系统时钟配置
  - 系统时钟在不同工作模式下使用不同的时钟源
    - Run（PIOSC/MOSC），Sleep，Deep-Sleep（LFIOSC），Hibernation（RTCOSC）
  - 正常运行模式下可选择 PIOSC 或 MOSC 作为系统时钟源
    - 默认配置：使用高精确内部振荡器（PIOSC）16M
    - 使用MOSC：S800实验板外部主振荡器（MOSC）固定为 25M
  - 正常运行模式下可以通过PLL倍频模式将系统时钟调整到不同的频率
    - 以PIOSC或MOSC为参考时钟，将PLL频率配置为320M或480M
    - 然后再经过（1-1024）分频得到想要的系统频率
    - ※ 注意：系统最大频率120M。频率越高，速度越快，功耗越大

- 系统分频器
  - 使用OSC和PLL都可以分频，分频系数都是10位（0~1023整数）
  - PLL的分频： $f_{syclk} = f_{VCO} / (PSYSDIV + 1)$ ， f_{vco} = 320M 或 480M
  - OSC的分频： $f_{syclk} = f_{oscclk} /(OSYSDIV+ 1)$ ， f_{oscclk} = 16M 或 25M
  - 如， f_{VCO}=480MHz时系统分频器与系统时钟频率的关系

System Clock (SYSCLK) (MHz)	f_{VCO} (MHz)= 480 MHz
	System Divisors (PSYSDIV +1)[a]
120	4
60	8
48	10
30	16
24	20
12	40
6	80

a. The use of non-integer divisors introduce additional jitter which may affect interface performance.

...

- TivaWare 时钟控制库函数（参见driverlib/sysctl.h）
  - **SysCtlClockFreqSet( )**：设置系统时钟频率
    - 设置时钟源：OSC/PLL
    - 设置振荡器：PIOSC/MOSC/其它（时钟源用PLL时只能选前PIOSC或MOSC ）
    - 设置PLL频率：480M/320M（时钟源用PLL时设置）

uint32_t **SysCtlClockFreqSet**(uint32_t ui32Config, uint32_t ui32SysClock) ;

ui32SysClock 是希望设置的时钟频率

ui32Config 由以下几组预定义的参数用“或”连接而成：

① 时钟源：SYSCTL_USE_OSC、SYSCTL_USE_PLL（二选一）

② 振荡器：SYSCTL_OSC_MAIN（主振荡器）
SYSCTL_OSC_INT （16M的内部高精度振荡器）
SYSCTL_OSC_INT30（内部低精度振荡器）
SYSCTL_OSC_EXT32（休眠模块振荡器）
（四选一）

③ 用PLL时：SYSCTL_CFG_VCO_480、SYSCTL_CFG_VCO_320 （二选一）

④ 用主振荡器时：SYSCTL_XTAL_25MHZ（外部固定的25MHz）

- SysCtlClockFreSet() 的几组参数定义在 driverlib/sysctl.h 头文件中

```
……
#define SYSCTL_CFG_VCO_480        0xF1000000  // VCO is 480 MHz
#define SYSCTL_CFG_VCO_320        0xF0000000  // VCO is 320 MHz
#define SYSCTL_USE_PLL            0x00000000  // System clock is the PLL clock
#define SYSCTL_USE_OSC             0x00003800  // System clock is the osc clock
 ……
#define SYSCTL_XTAL_25MHZ         0x00000680  // External crystal is 25.0 MHz
#define SYSCTL_OSC_MAIN           0x00000000  // Osc source is main osc
#define SYSCTL_OSC_INT            0x00000010  // Osc source is int. osc
#define SYSCTL_OSC_INT4           0x00000020  // Osc source is int. osc /4
#define SYSCTL_OSC_INT30          0x00000030  // Osc source is int. 30 KHz
#define SYSCTL_OSC_EXT32          0x80000038  // Osc source is ext. 32 KHz
……
```

- 系统时钟设置示例（void S800_Clock_Init(void)）

```
uint32_t ui32SysClock;    //一般定义为全局变量
```

- 例1：使用25M的MOSC将系统时钟频率设置为25M

```
ui32SysClock = SysCtlClockFreqSet ( (SYSCTL_USE_OSC |
    SYSCTL_OSC_MAIN | SYSCTL_XTAL_25MHZ), 25000000);
```

  - 此时系统分频器取2

- 例2：以PIOSC为参考时钟，使用480M的PLL将系统时钟频率设置为80M

```
ui32SysClock = SysCtlClockFreqSet ( (SYSCTL_USE_PLL |
    SYSCTL_OSC_INT | SYSCTL_CFG_VCO_480), 80000000);
```

  - 此时系统分频器取6

实验一 时钟选择与 GPIO 实验

- 实验内容
  - 例程1-1.c：分别使用内部16M的PIOSC时钟，外部25M的MOSC时钟，以及PLL时钟进行GPIO-PF0的闪烁
  - 编程要点
    1. 程序结构
    2. **初始化：**系统时钟设置、 **GPIO配置**
    3. 任务1：GPIO的读写
    4. 任务2：GPIO控制LED灯的亮灭

```
int main(void)
{
    uint32_t delay_time, key_value;

    S800_Clock_Init();
    S800_GPIO_Init();

    while(1)
    {
        key_value = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_0); //read the PJ0 key value

        if (key_value  == 0)                    //USR_SW1-PJ0 pressed
            delay_time = FASTFLASHTIME;
        else
            delay_time = SLOWFLASHTIME;

        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);     // Turn on the LED.
        Delay(delay_time);
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);            // Turn off the LED.
        Delay(delay_time);
    }
}
```

- TivaWare GPIO驱动库函数

(驱动程序参见driverlib/gpio.c，API定义参见 driverlib/gpio.h)

- 函数GPIOPadConfigSet()：配置引脚的驱动强度和类型

```
void GPIOPadConfigSet( uint32_t  ui32Port,
                       uint8_t   ui8Pins,
                       uint32_t  ui32Strength,
                       uint32_t  ui32PinType );
```

ui32Port：GPIO端口基地址，

ui8Pins：引脚位，可以是多个位的”或”

ui32Strength：驱动强度，仅对输出引脚有效。缺省为2-mA

ui32PinType： 引脚类型。缺省为推挽Push-pull

- TivaWare GPIO驱动库函数（续）
  - 函数GPIOPinTypeGPIOInput()：配置引脚为GPIO输入
  - 函数GPIOPinTypeGPIOOutput()：配置引脚为GPIO输出
  - 函数GPIOPinRead()：读指定引脚的值
  - 函数GPIOPinWrite()：写值到指定引脚

```
void GPIOPinTypeGPIOInput(uint32_t ui32Port, uint8_t ui8Pins);
void GPIOPinTypeGPIOOutput(uint32_t ui32Port, uint8_t ui8Pins);
int32_t GPIOPinRead(uint32_t ui32Port, uint8_t ui8Pins);
void GPIOPinWrite(uint32_t ui32Port, uint8_t ui8Pins, uint8_t ui8Val);
```

其中： ui32Port为GPIO端口基地址，
ui8Pins为引脚位，可以是多个位的”或”
ui8Val为要写入的数据

- GPIO端口基地址（参见inc/hw_memmap.h）

```
#define GPIO_PORTA_BASE         0x40004000  // GPIO Port A
#define GPIO_PORTB_BASE         0x40005000  // GPIO Port B
#define GPIO_PORTC_BASE         0x40006000  // GPIO Port C
#define GPIO_PORTD_BASE         0x40007000  // GPIO Port D
#define GPIO_PORTE_BASE         0x40024000  // GPIO Port E
#define GPIO_PORTF_BASE         0x40025000  // GPIO Port F
#define GPIO_PORTG_BASE         0x40026000  // GPIO Port G
#define GPIO_PORTH_BASE         0x40027000  // GPIO Port H
#define GPIO_PORTJ_BASE         0x4003D000  // GPIO Port J
#define GPIO_PORTA_AHB_BASE     0x40058000  // GPIO Port A (high speed)
#define GPIO_PORTB_AHB_BASE     0x40059000  // GPIO Port B (high speed)
#define GPIO_PORTC_AHB_BASE     0x4005A000  // GPIO Port C (high speed)
#define GPIO_PORTD_AHB_BASE     0x4005B000  // GPIO Port D (high speed)
#define GPIO_PORTE_AHB_BASE     0x4005C000  // GPIO Port E (high speed)
#define GPIO_PORTF_AHB_BASE     0x4005D000  // GPIO Port F (high speed)
#define GPIO_PORTG_AHB_BASE     0x4005E000  // GPIO Port G (high speed)
#define GPIO_PORTH_AHB_BASE     0x4005F000  // GPIO Port H (high speed)
#define GPIO_PORTJ_AHB_BASE     0x40060000  // GPIO Port J (high speed)
#define GPIO_PORTK_BASE         0x40061000  // GPIO Port K
#define GPIO_PORTL_BASE         0x40062000  // GPIO Port L
#define GPIO_PORTM_BASE         0x40063000  // GPIO Port M
#define GPIO_PORTN_BASE         0x40064000  // GPIO Port N
#define GPIO_PORTP_BASE         0x40065000  // GPIO Port P
#define GPIO_PORTQ_BASE         0x40066000  // GPIO Port Q
```

- 引脚位定义（参见 driverlib/gpio.h）

```
#define GPIO_PIN_0              0x00000001  // GPIO pin 0
#define GPIO_PIN_1              0x00000002  // GPIO pin 1
#define GPIO_PIN_2              0x00000004  // GPIO pin 2
#define GPIO_PIN_3              0x00000008  // GPIO pin 3
#define GPIO_PIN_4              0x00000010  // GPIO pin 4
#define GPIO_PIN_5              0x00000020  // GPIO pin 5
#define GPIO_PIN_6              0x00000040  // GPIO pin 6
#define GPIO_PIN_7              0x00000080  // GPIO pin 7
```

- 引脚驱动强度（driverlib/gpio.h）用作GPIOPadConfigSet参数

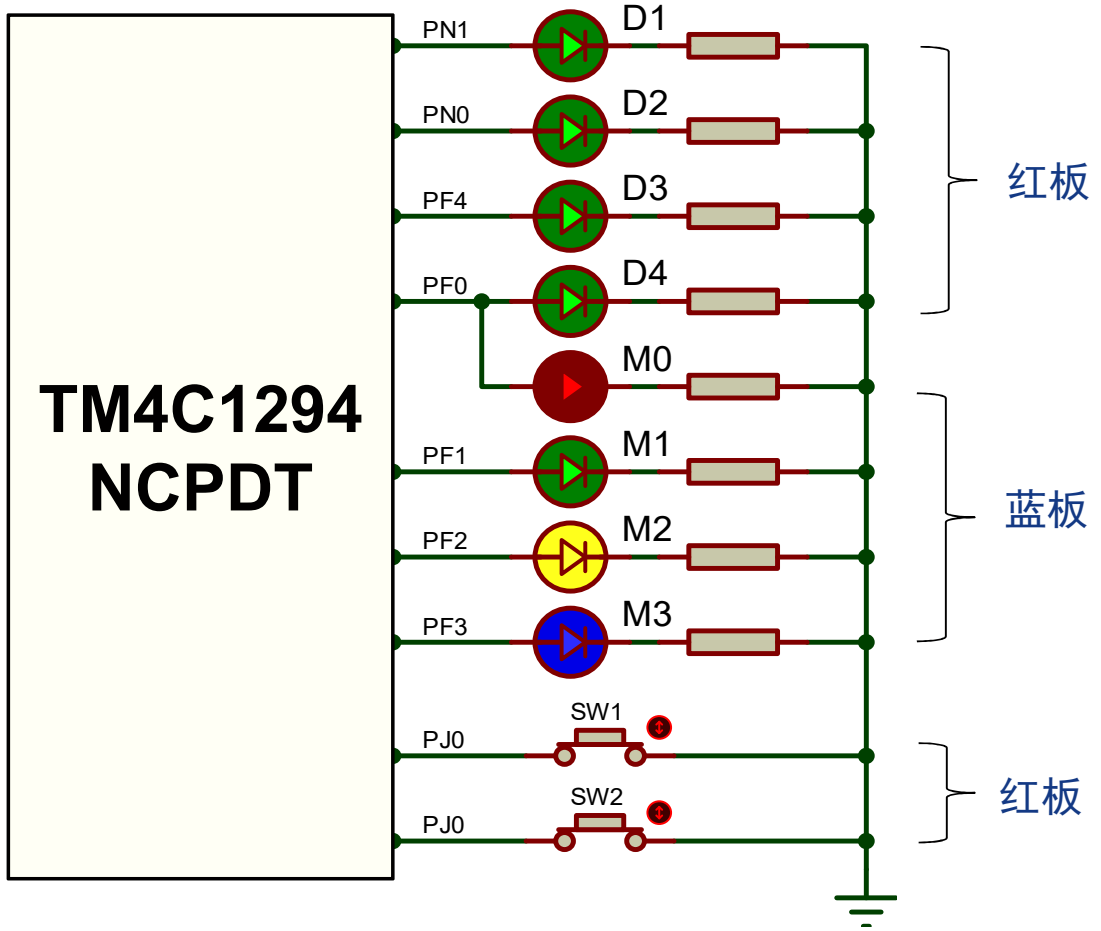
```
#define GPIO_STRENGTH_2MA       0x00000001  // 2mA drive strength
#define GPIO_STRENGTH_4MA       0x00000002  // 4mA drive strength
#define GPIO_STRENGTH_8MA       0x00000066  // 8mA drive strength
#define GPIO_STRENGTH_8MA_SC    0x0000006E  // 8mA drive with slew rate control
#define GPIO_STRENGTH_10MA      0x00000075  // 10mA drive strength
#define GPIO_STRENGTH_12MA      0x00000077  // 12mA drive strength
```

- 引脚类型（driverlib/gpio.h）用作GPIOPadConfigSet参数

```
#define GPIO_PIN_TYPE_STD           0x00000008  // Push-pull
#define GPIO_PIN_TYPE_STD_WPU       0x0000000A  // Push-pull with weak pull-up
#define GPIO_PIN_TYPE_STD_WPD       0x0000000C  // Push-pull with weak pull-down
#define GPIO_PIN_TYPE_OD            0x00000009  // Open-drain
#define GPIO_PIN_TYPE_ANALOG        0x00000000  // Analog comparator
#define GPIO_PIN_TYPE_WAKE_HIGH             0x00000208  // Hibernate wake, high
#define GPIO_PIN_TYPE_WAKE_LOW              0x00000108  // Hibernate wake, low
```

- GPIO操作基本步骤
  - GPIO引脚配置（在主循环之前一次性设置）
    1. 使能引脚对应的端口：SysCtlPeripheralEnable
    2. 测试端口状态：SysCtlPeripheralReady
    3. 设置引脚输入/输出模式：GPIOPinTypeGPIOIntput / GPIOPinTypeGPIOOutput
    4. 设置引脚类型：GPIOPadConfigSet
  - GPIO引脚访问（主循环中可多次读写）
    - 输出（写）：GPIOPinWrite
    - 输入（读）：GPIOPinRead
    - 注：配置为输出的引脚也可以读取数据寄存器的值

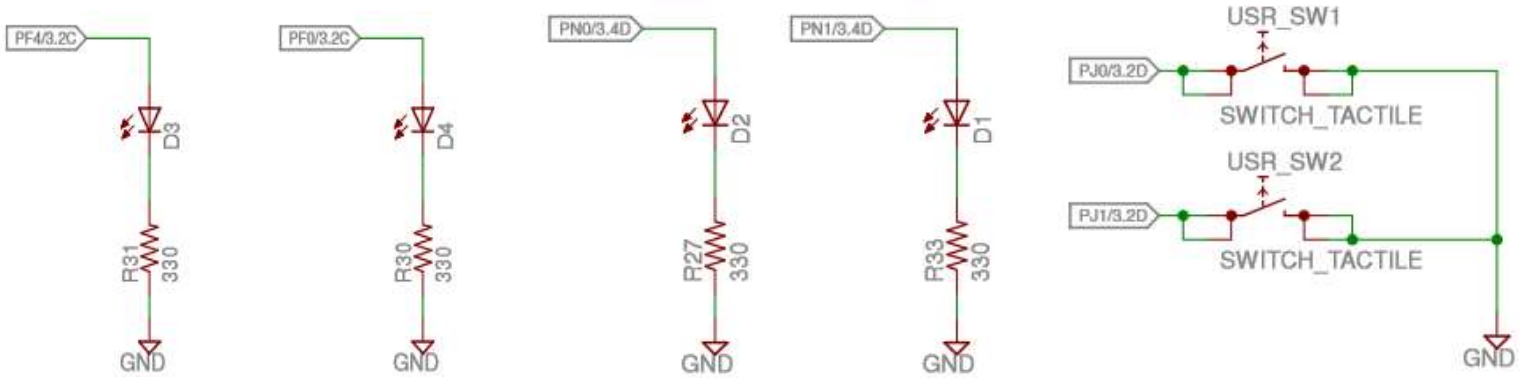
- S800实验板上的部分GPIO管脚功能



LED：高点亮

按钮：按下为低

- 管脚分析
  - 红板上的按键SW1、SW2，对应PJ0、PJ1。低电平有效
    - 设置：输入
    - 分析：这两个引脚没有外接上拉电阻，必须配置为内部弱上拉，才能清楚地区分未按下与按下状态
  - 红板上LED3和4 对应PF4和PF0，LED1和2对应PN0和1
    - 设置：输出
    - 操作：高电平点亮，低电平熄灭



- GIPO引脚配置步骤（**void S800_GPIO_Init(void)**）
- 将端口J的PJ0和PJ1配置为输入

1. 使能端口

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);
```

2. 测试端口状态

```
while ( !SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOJ) );
```

3. 设置端口相关引脚为输入

```
GPIOPinTypeGPIOInput(GPIO_PORTJ_BASE, GPIO_PIN_0 | GPIO_PIN_1);
```

4. 设置引脚类型（弱上拉，视电路设计而定）

```
GPIOPadConfigSet(GPIO_PORTJ_BASE, GPIO_PIN_0|GPIO_PIN_1,
    GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
```

- 将端口F的PF0和PF1配置为输出

1. 使能端口

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
```

2. 测试端口状态

```
while ( !SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOF) );
```

3. 设置端口相关引脚为输出

```
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_0 | GPIO_PIN_1);
```

4. 设置输出状况（可选）

```
GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_0|GPIO_PIN_1,
GPIO_STRENGTH_8MA_SC, GPIO_PIN_TYPE_STD);
```

- GIPO配置示例

```
void S800_GPIO_Init(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);              //使能 PortF
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOF)); //等待ready
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);              //使能 PortJ
    while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOJ));  //等待ready

    //Set PF0 as Output pin
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_0);
    //Set PJ0 as input pin
    GPIOPinTypeGPIOInput(GPIO_PORTJ_BASE, GPIO_PIN_0);
    GPIOPadConfigSet(GPIO_PORTJ_BASE, GPIO_PIN_0,
    GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);  //设弱上拉
}
```

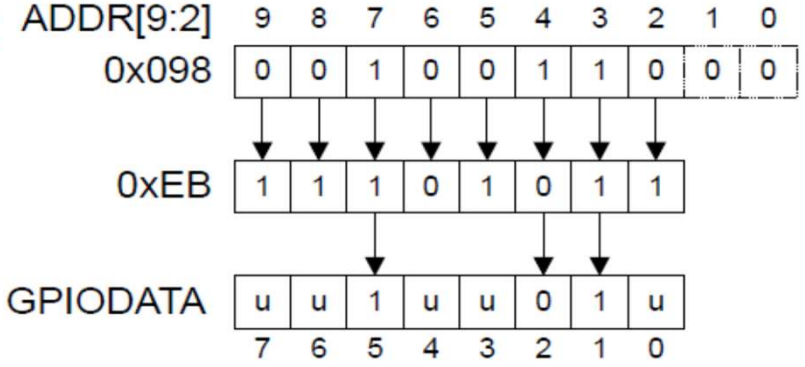
实验一 时钟选择与 GPIO 实验

- 实验内容
  - 例程1-1.c：分别使用内部16M的PIOSC时钟，外部25M的MOSC时钟，以及PLL时钟进行GPIO-PF0的闪烁
  - 编程要点
    1. 程序结构
    2. 初始化：系统时钟设置、 GPIO配置
    3. **任务1：GPIO的读写**
    4. 任务2：GPIO控制LED灯的亮灭

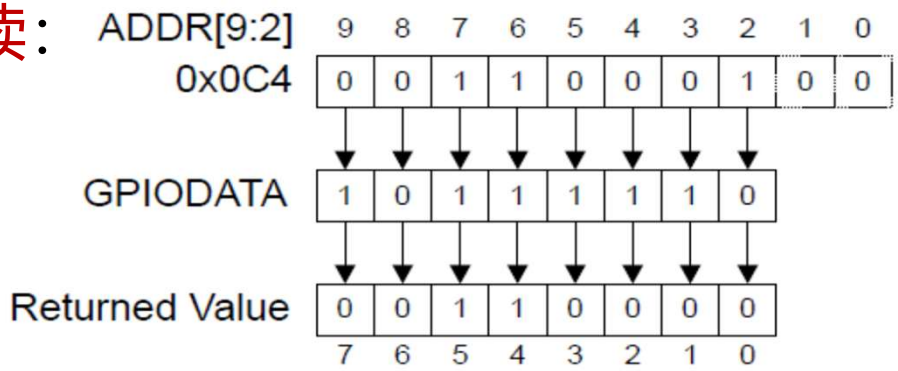
```
int main(void)
{
    uint32_t delay_time, key_value;

    S800_Clock_Init();
    S800_GPIO_Init();

    while(1)
    {
        key_value = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_0); //read the PJ0 key value

        if (key_value  == 0)                    //USR_SW1-PJ0 pressed
            delay_time = FASTFLASHTIME;
        else
            delay_time = SLOWFLASHTIME;

        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);// Turn on the LED.
        Delay(delay_time);
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);          // Turn off the LED.
        Delay(delay_time);
    }
}
```

- GPIO数据寄存器（GPIODATA，低8位有效）
  - 当引脚配置为输出时，往数据寄存器写入数据，将直接传送到引脚，读数据时返回最后一次写入的值
  - 当引脚配置为输入时，读数据寄存器则返回相应引脚的输入值
- 读写数据寄存器时，地址总线的位[9:2]用做屏蔽位，屏蔽位为1则对应位数据被读出或写入，否则返回0或不作修改

写：



读：



- 为了对GPIODATA寄存器所有位同时进行读写可以加偏移0x3FC
  - 如：GPIO F口的基地址为0x4005D000，访问 PF口数据寄存器的地址可以设置为：0x4005D000+0x3FC = 0x4005D3FC
- 在tm4c1294ncpdt.h中

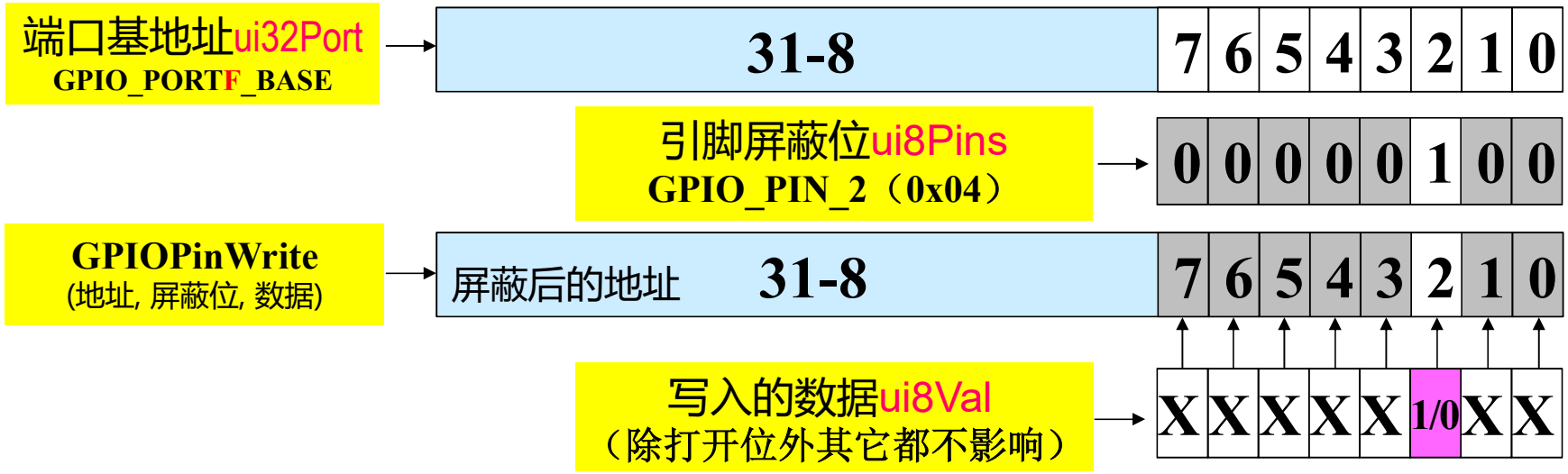
```
#define GPIO_PORTF_AHB_DATA_R   (*((volatile uint32_t *)0x4005D3FC))
```

- 则往PF口读写数据可以写为：

```
GPIO_PORTF_AHB_DATA_R = 0x04；//置高PF2，其他引脚置低
int32_t  key = GPIO_PORTF_AHB_DATA_R；//读PF口
```

- 函数GPIOPinWrite()：写值到指定引脚

  void GPIOPinWrite(uint32_t ui32Port, uint8_t ui8Pins, uint8_t ui8Val);



例：设置PF2：打开引脚2，写入数据到指向PF端口的第2位

```
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0x04); //置高PF2，其他引脚不变
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0x00); //置低PF2，其他引脚不变
```

- 函数GPIOPinRead()：读指定引脚的值

uint32_t GPIOPinRead(uint32_t ui32Port, uint8_t ui8Pins);

例：读取PJ2：打开引脚2，从PJ端口读入数据

```
int32_t key = GPIOPinRead(GPIO_PORTJ_BASE, GPIO_PIN_2);
//key = 0 或者 key = 4
```

实验一 时钟选择与 GPIO 实验

- 实验内容
  - 例程1-1.c：分别使用内部16M的PIOSC时钟，外部25M的MOSC时钟，以及PLL时钟进行GPIO-PF0的闪烁
  - 编程要点
    1. 程序结构
    2. 初始化：系统时钟设置、 GPIO配置
    3. 任务1：GPIO的读写
    4. **任务2：GPIO控制LED灯的亮灭**

```
int main(void)
{
    uint32_t delay_time, key_value;

    S800_Clock_Init();
    S800_GPIO_Init();

    while(1)
    {
        key_value = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_0); //read the PJ0 key value

        if (key_value  == 0)                    //USR_SW1-PJ0 pressed
            delay_time = FASTFLASHTIME;
        else
            delay_time = SLOWFLASHTIME;

        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0);    // Turn on the LED.
        Delay(delay_time);
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);           // Turn off the LED.
        Delay(delay_time);
    }
}
```

GIPO编程示例

- 主循环中检测按键控制LED灯的快慢闪烁

```
while(1)  {
    key_value = GPIOPinRead(GPIO_PORTJ_BASE,GPIO_PIN_0); //读PJ0
    if (key_value == 0)              //USR_SW1 pressed
        delay_time = FASTFLASHTIME;   //快闪
    else
        delay_time = SLOWFLASHTIME;  //慢闪
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, GPIO_PIN_0); //点亮PF0
    Delay(delay_time);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0, 0x0);    //熄灭PF0
    Delay(delay_time);
}
```

- 延时控制
  - TivaWare 时钟控制函数SysCtlDelay( )，可产生一个固定时长的延时

```
void SysCtlDelay(uint32_t ui32Count);
其中，延时时长 = 3× ui32Count ×系统时钟周期
```

如：SysCtlDelay(ui32SysClock / 60)); //延时50ms

其中：ui32SysClock为之前时钟设置得到的系统时钟频率

延时 = 3 * (ui32SysClock/60) * (1/ui32SysClock) = 50ms

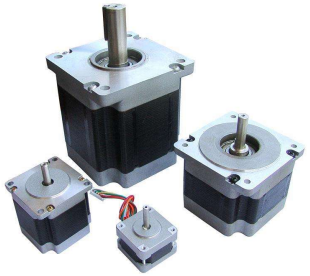
  - 用SysCtlDelay函数替换Delay函数实现指定频率的闪烁

快闪：300ms，慢闪：3s

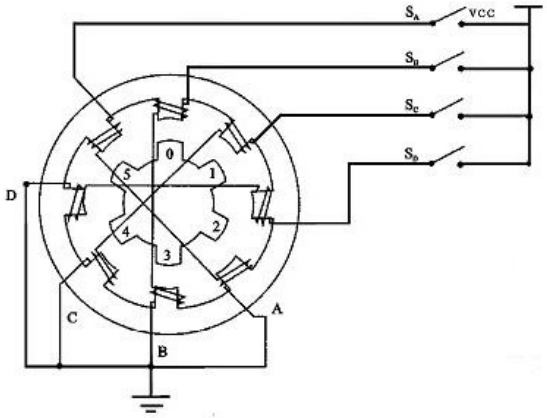
※ 注意：软件延时时间不很精确

实验一 时钟选择与 GPIO 实验

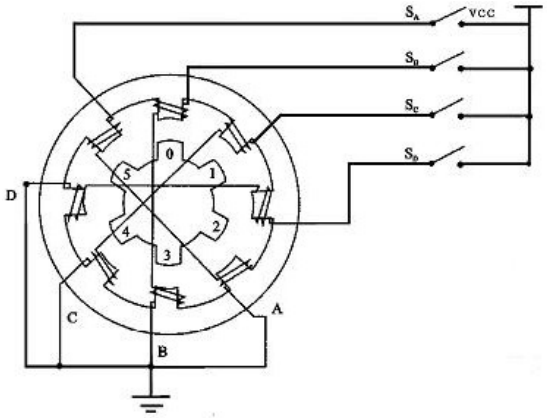
- 例程1-4.c：步进电机的驱动。循环执行如下序列，观察红色指针的转动
  - PF3-ON，Delay（10ms），PF3-OFF
  - PF2-ON，Delay（10ms），PF2-OFF
  - PF1-ON，Delay（10ms），PF1-OFF
  - PF0-ON，Delay（10ms），PF0-OFF
- 编程要点
  1. **步进电机的工作原理**
  2. **GPIO控制步进电机的顺时针或逆时针转动**



- 步进电机的工作原理
  - 一种将电脉冲信号转化为角位移的执行设备。
  - 当步进驱动器接收到一个脉冲信号（指定子绕组改变一次通电状态），它就驱动步进电机按设定的方向转动一个固定的角度（即步距角）。
  - 当对步进电机按一定顺序施加一系列连续不断的控制脉冲时，它可以连续不断地转动。
  - 通过控制脉冲个数来控制角位移量，从而达到准确定位的目的
  - 通过控制脉冲频率来控制电机转动的速度和加速度，从而达到调速的目的



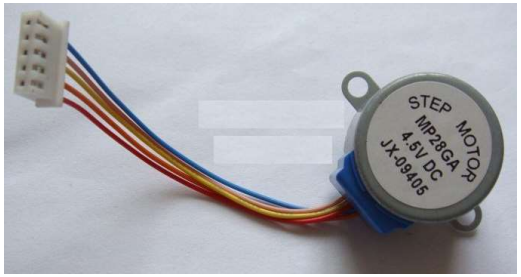
- 步进电机的工作原理（续）
  - 每一个脉冲信号使得步进电机的定子绕组的通电状态改变一次，当通电状态的改变完成一个循环时，转子转过一个齿距。
  - 四相步进电机常见的通电方式
    - 单四拍：A-B-C-D-A……（A-B-C-D为一个循环）
    - 双四拍：AB-BC-CD-DA-AB……
    - 八拍：　A-AB-B-BC-C-CD-D-DA-A……



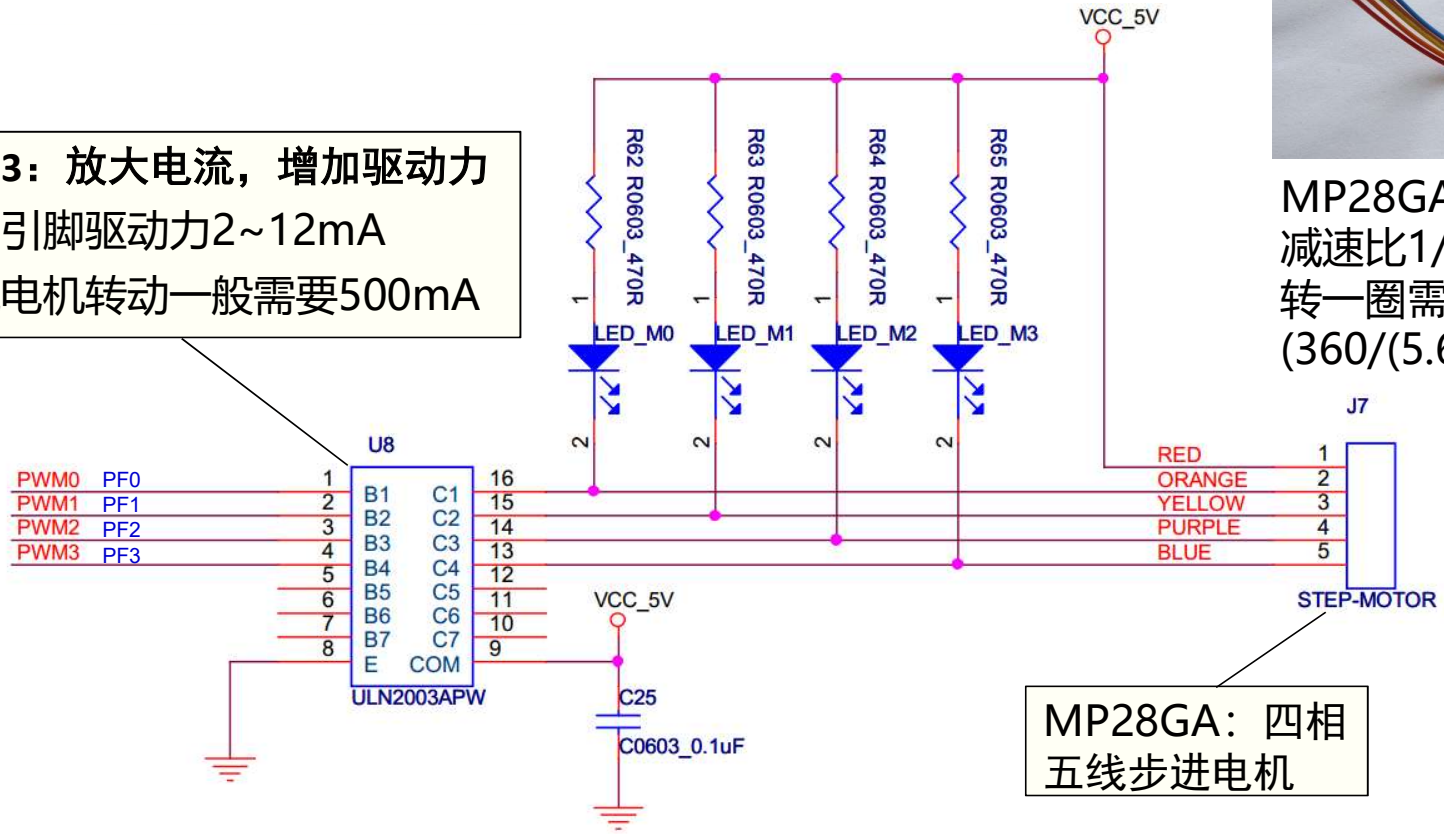
步进电机线路图

ULN2003：放大电流，增加驱动力

- 输出引脚驱动力2~12mA
- 直流电机转动一般需要500mA



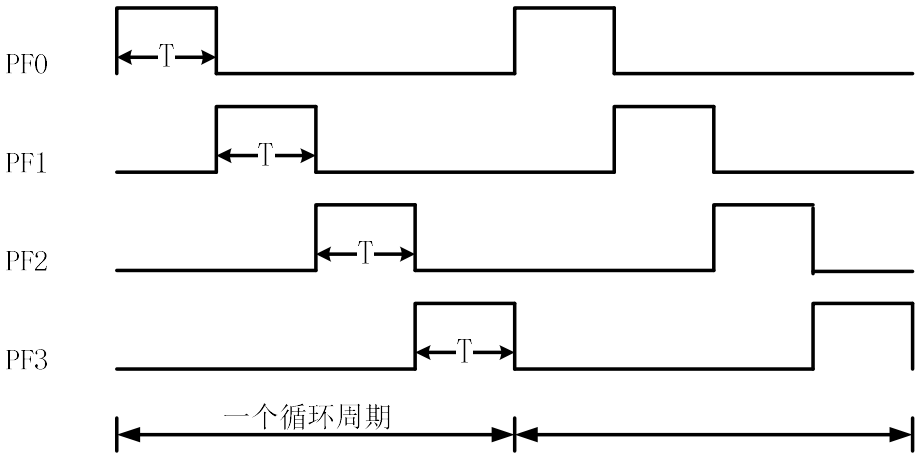
MP28GA步距角5.625/64，减速比1/64，齿数8。
转一圈需要的循环=(360/(5.625*8))*64=512



MP28GA：四相五线步进电机

- GPIO控制步进电机的顺时针或逆时针转动
  - 采用单四拍、双四拍或八拍方式，依次给PF0~PF3（或者PF3~PF0）对应的各个相输入高电平信号，使步进电机的红色指针顺时针一圈，逆时针一圈，循环往复。
  - 改变延时时间可以得到不同的转动速度

单四拍示意图



- 开始你的实验一 -