



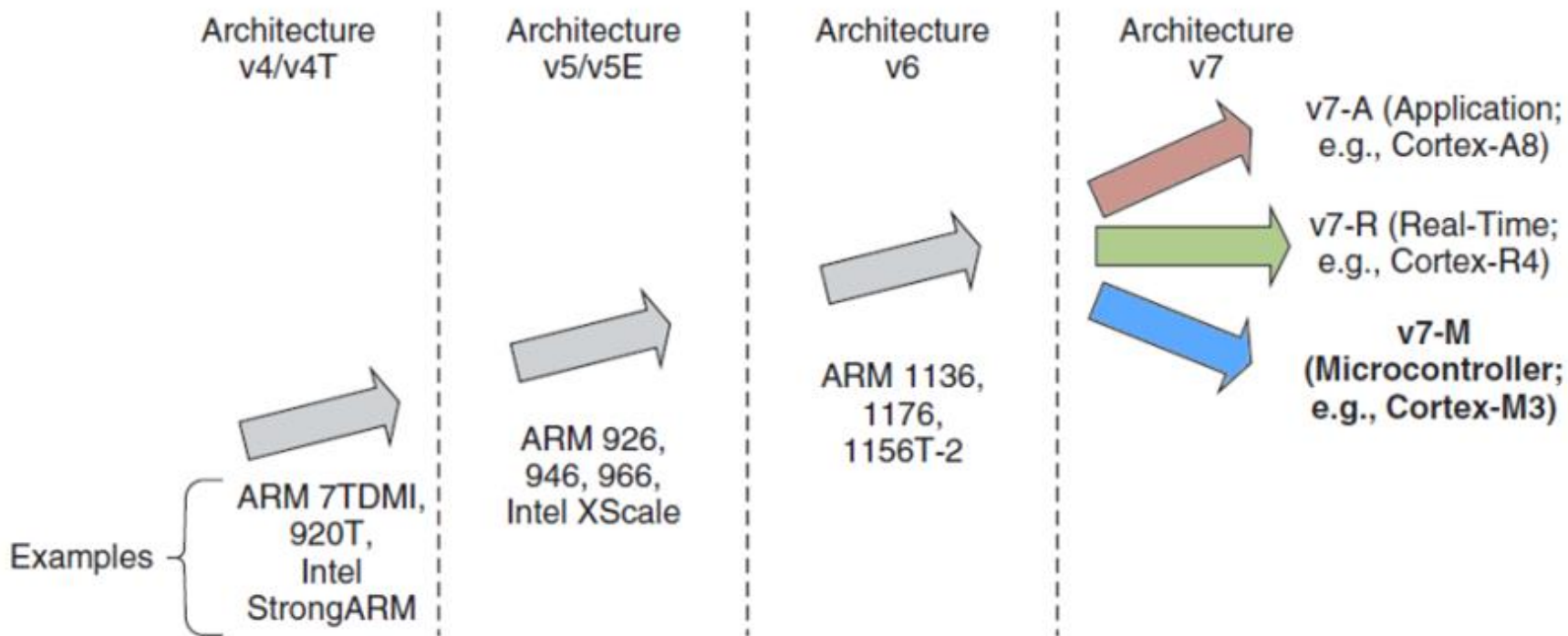
第十章 基于ARM微处理器的开发和应用

- 1、基于ARM内核的STM32微处理器
- 2、嵌入式系统软件体系结构
- 3、通用输入输出GPIO-人机交互接口

1、基于ARM内核的STM32微处理器



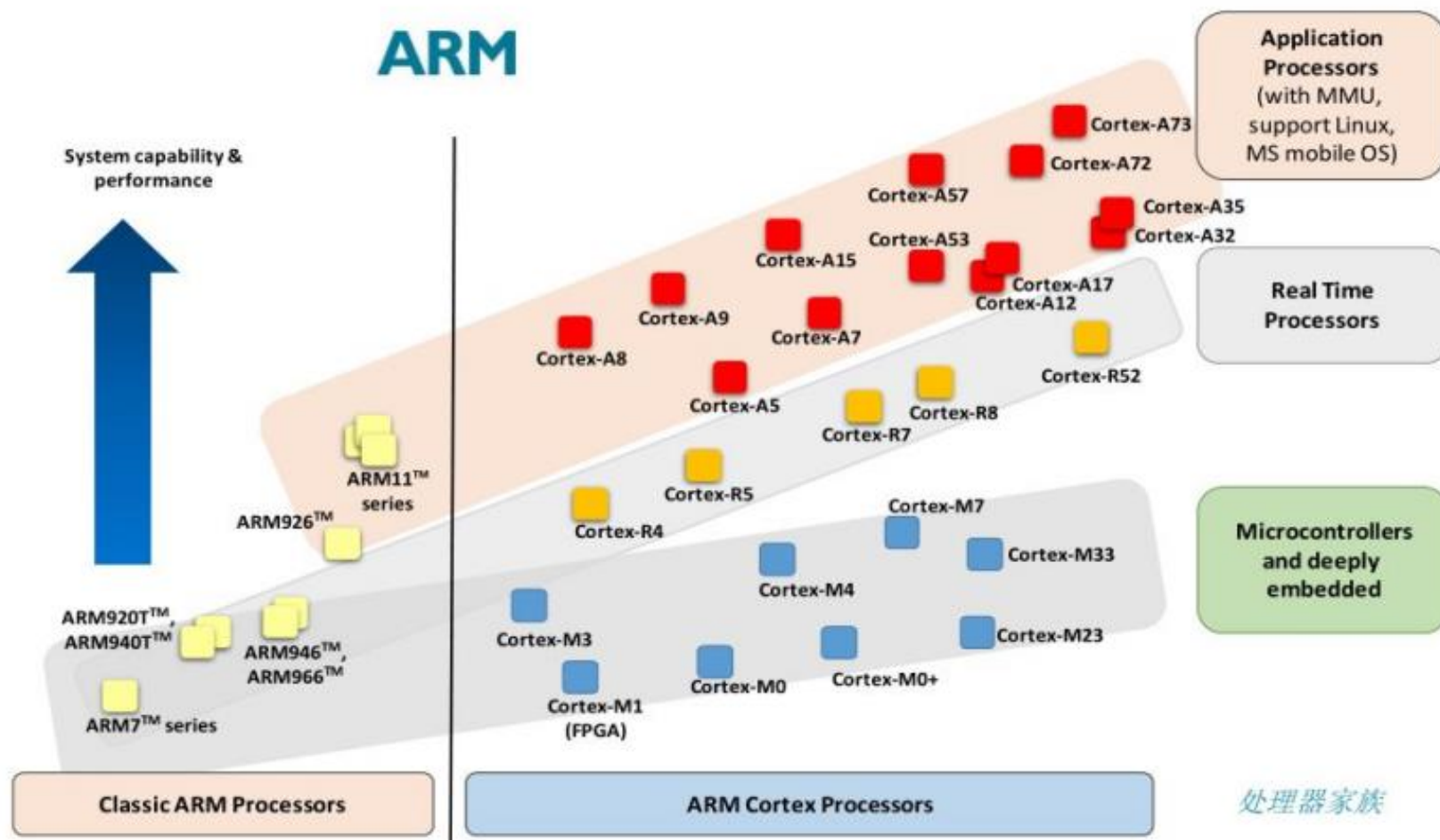
目前市面上常见的arm处理器架构可分为arm7、arm9、arm10、arm11以及cortex系列，每个系列又包括很多种ip内核的产品。生产arm芯片的厂家很多，samsung(三星)、npx、cirrus logic、freescale、ti、st、winbond等主流半导体厂商的arm芯片产品。



1、基于ARM内核的STM32微处理器

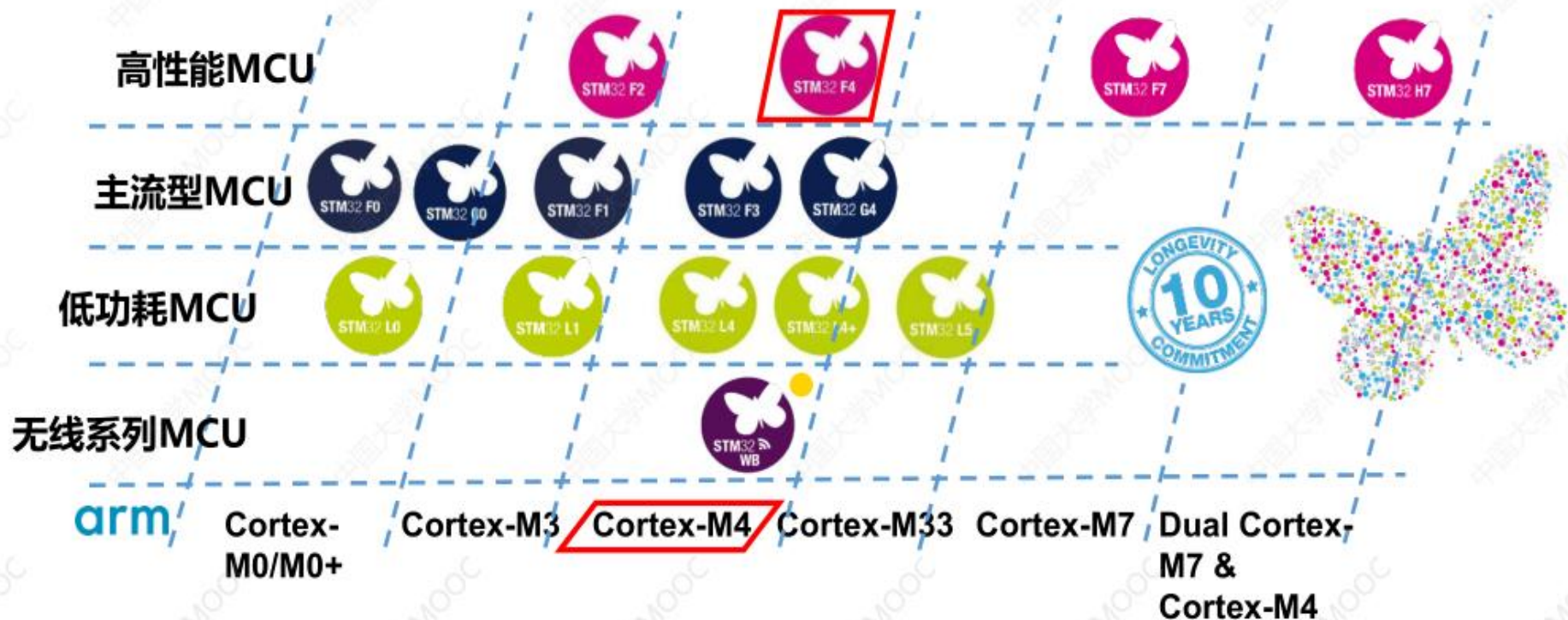


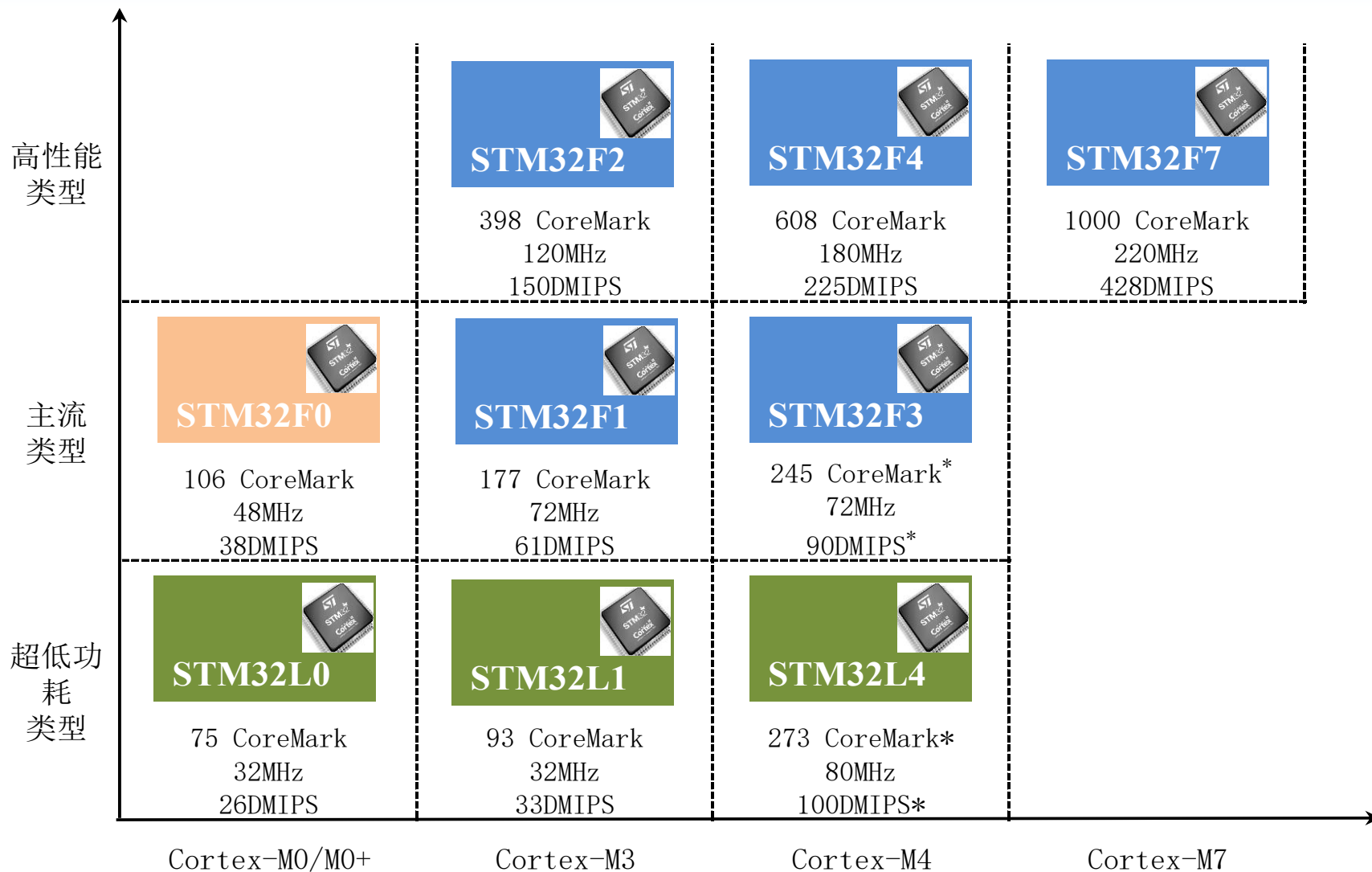
目前市面上常见的arm处理器架构可分为arm7、arm9、arm10、arm11以及cortex系列，每个系列又包括很多种ip内核的产品。生产arm芯片的厂家很多，samsung(三星)、nxp、cirrus logic、freescale、ti、st、winbond等主流半导体厂商的arm芯片产品。





丰富的产品线，面向不同应用







分类	型号	内核	特点
高性能	STM32F2	Cortex-M3	通用型微控制器，最高主频120MHz
	STM32F4	Cortex-M4	通用型微控制器，最高主频180MHz
	STM32F7	Cortex-M7	通用型微控制器，最高主频216MHz
	STM32H7	Cortex-M7	通用型微控制器，最高主频400MHz

分类	型号	内核	特点
主流型	STM32F0	Cortex-M0	低成本、入门级微控制器，高性价比，最高主频48MHz
	STM32F1	Cortex-M3	通用型微控制器，最高主频72MHz
	STM32F3	Cortex-M4	侧重混合信号应用，最高主频72MHz
	STM32G0	Cortex-M0+	F0系列升级产品，最高主频64MHz
	STM32G4	Cortex-M4	F3系列升级产品，最高主频170MHz

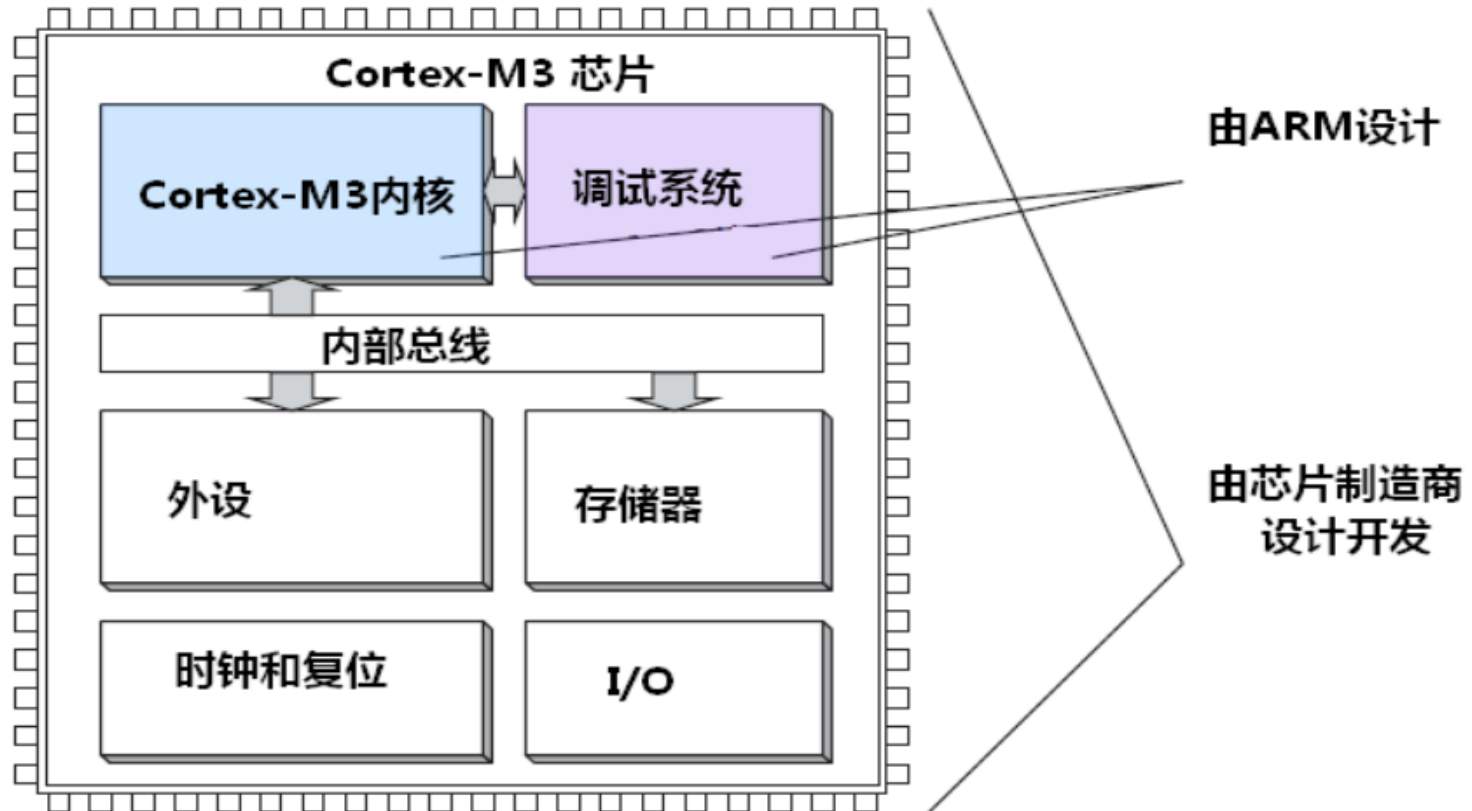


分类	型号	内核	特点
低功耗	STM32L0	Cortex-M0+	低功耗应用，最高主频32MHz
	STM32L1	Cortex-M3	低功耗应用，最高主频32MHz
	STM32L4	Cortex-M4	低功耗应用，最高主频80MHz
	STM32L4+	Cortex-M4	低功耗应用，最高主频120MHz
	STM32L5	Cortex-M33	低功耗应用，提高了安全性，最高主频110MHz

分类	型号	内核	特点
无线	STM32WB	Cortex-M4 Cortex-M0+	双核微控制器，侧重无线应用，支持2.4G无线通信、蓝牙5.0和IEEE 802.15.4无线标准，最高主频64MHz

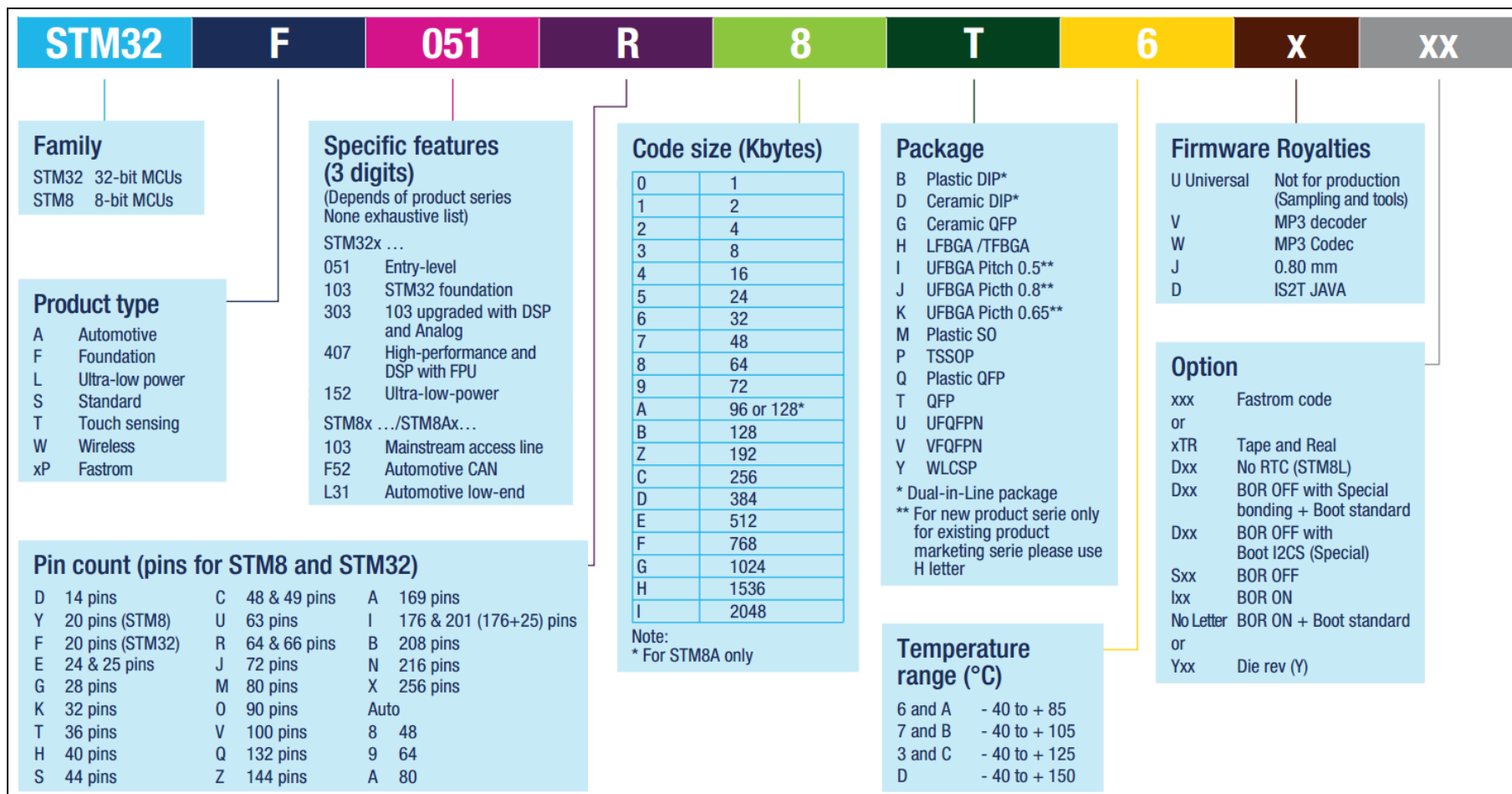


从Cortex-M3内核到基于Cortex-M3的MCU





STM32微控器命名规则





片内 外设

内置丰富外设资源

模拟 外设



模数/数模转换器



电机控制



运算放大器



电压比较器

数字 外设



USB



LCD液晶控制器



SPI / UART



SD卡接口

F4系列 分类

STM32F4系列微控制器分类

入门型

STM32
F401

STM32
F410

STM32
F411

STM32
F412

STM32
F413

STM32
F423

高性能，低成本，小封装
动态效能技术

基础型

STM32
F405

STM32
F415

STM32
F407

STM32
F417

STM32
F446

更多的连接性
和安全性

高级型

STM32
F427

STM32
F437

STM32
F429

STM32
F439

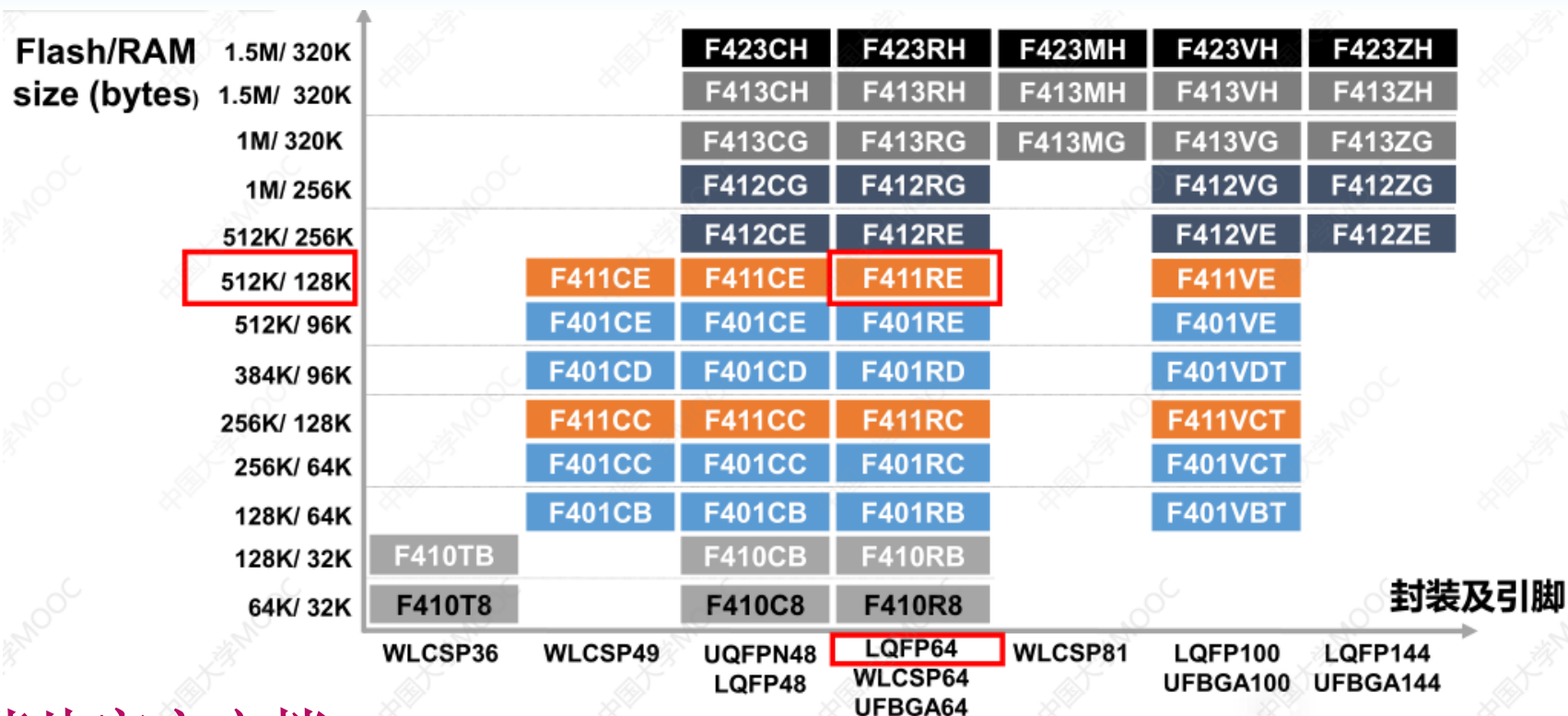
STM32
F469

STM32
F479

丰富的连接外设
加强的图形处理功能
大容量的存储



容量和引脚



芯片官方文档

芯片数据手册

介绍芯片引脚定义、电气特性和元件封装等，一般用于硬件电路设计。

用户参考手册

芯片片内外设的具体描述，比如GPIO、Timer、UART等模块的功能介绍及相关寄存器定义，一般用于软件开发。



Nucleo Board



灵活搭建
产品原型

Discovery Kit



重要功能
参考评估

Evaluation Board



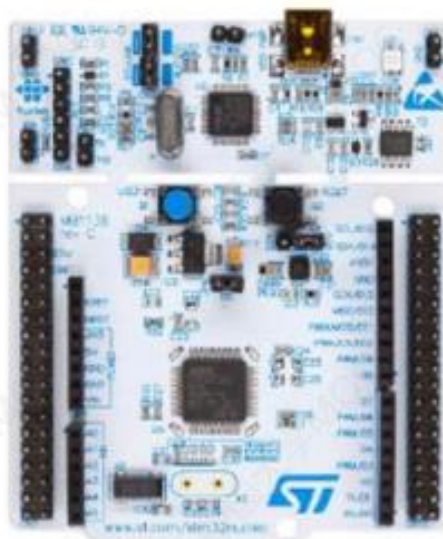
全部功能
参考评估



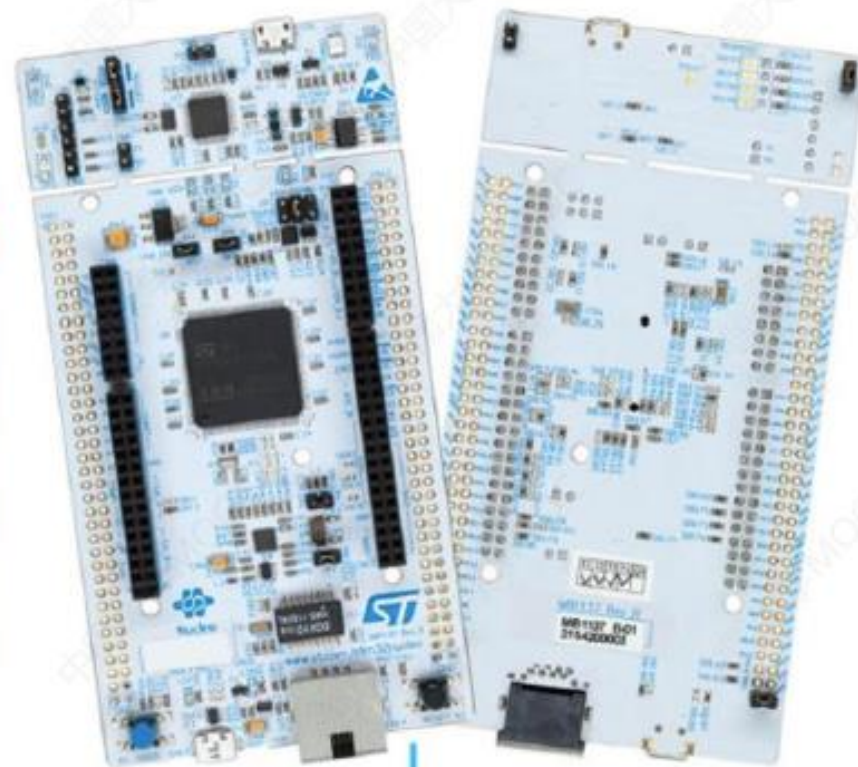
STM32 Nucleo开发板



Nucleo-32

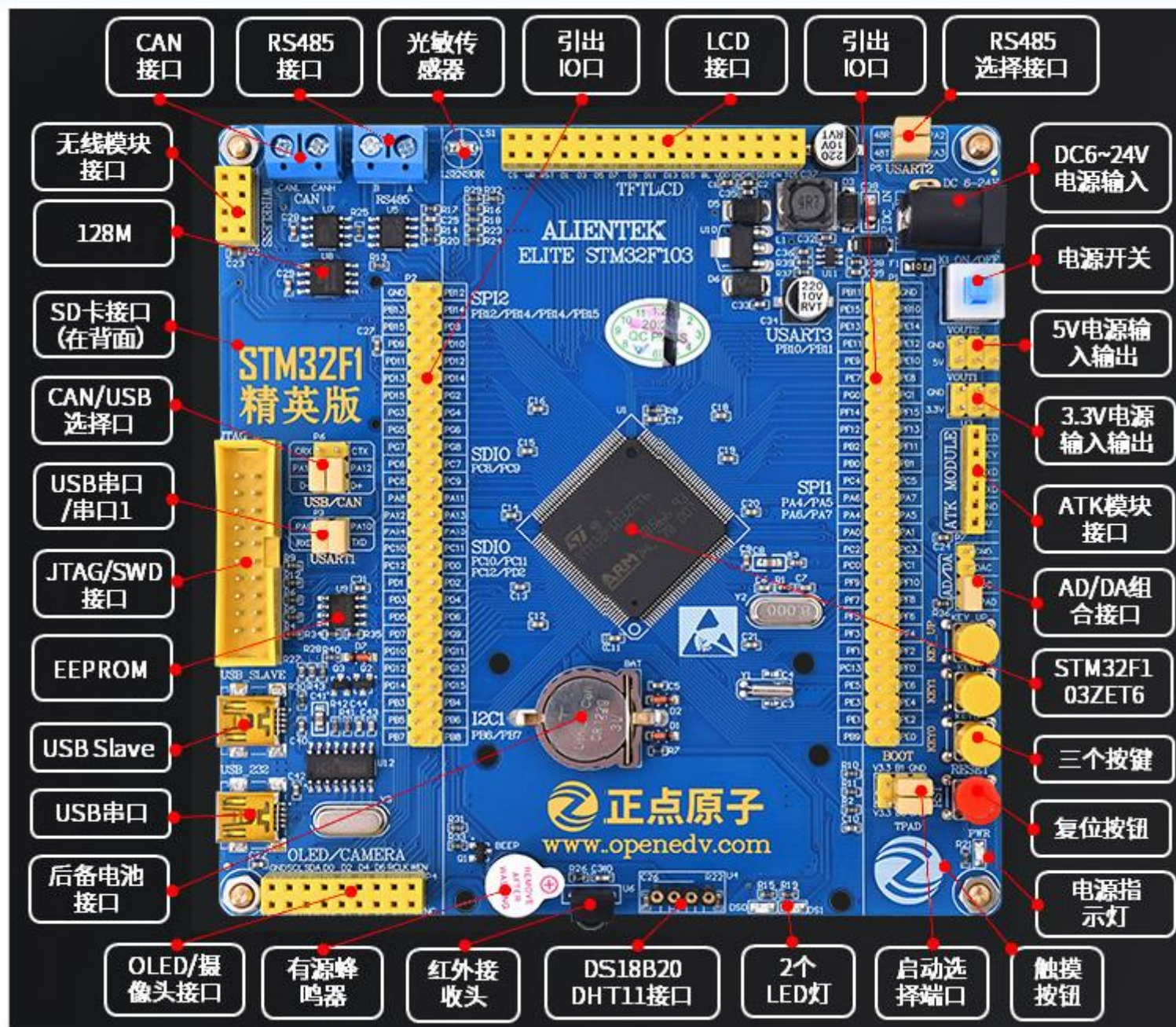


Nucleo-64



Nucleo-144

STM32F103ZET6 ARM开发板





Arduino

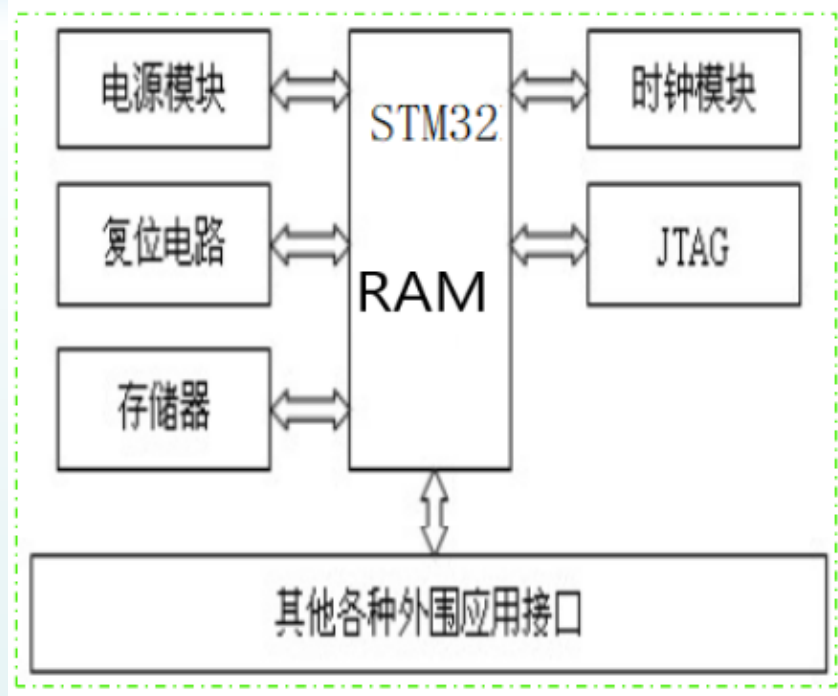
借助Arduino生态圈，接入众多的Arduino扩展板





ARM微处理器最小硬件系统

- **ARM微处理器**;
- **电源模块**，包括CPU内核和I/O接口电源电源;
- **时钟模块**，包括系统主时钟和实时时钟;
- **复位模块**，包括系统上电复位、手动复位和内部复位;
- **存储器模块（可选）**，包括程序保存存储器（FLASH）和程序运行存储器（SDRAM）;
- **JTAG调试接口模块**。





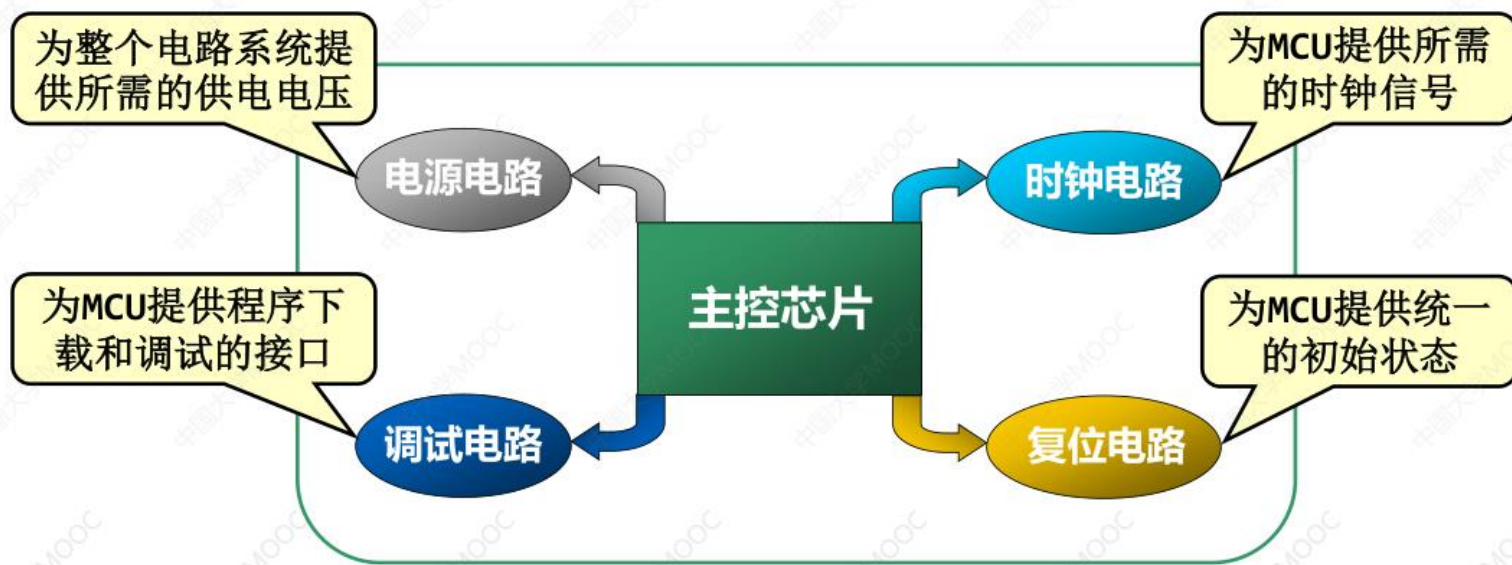
微控制器最小系统的定义

狭义的最小系统

仅包括电源电路、时钟电路、复位电路、调试电路及主控芯片电路

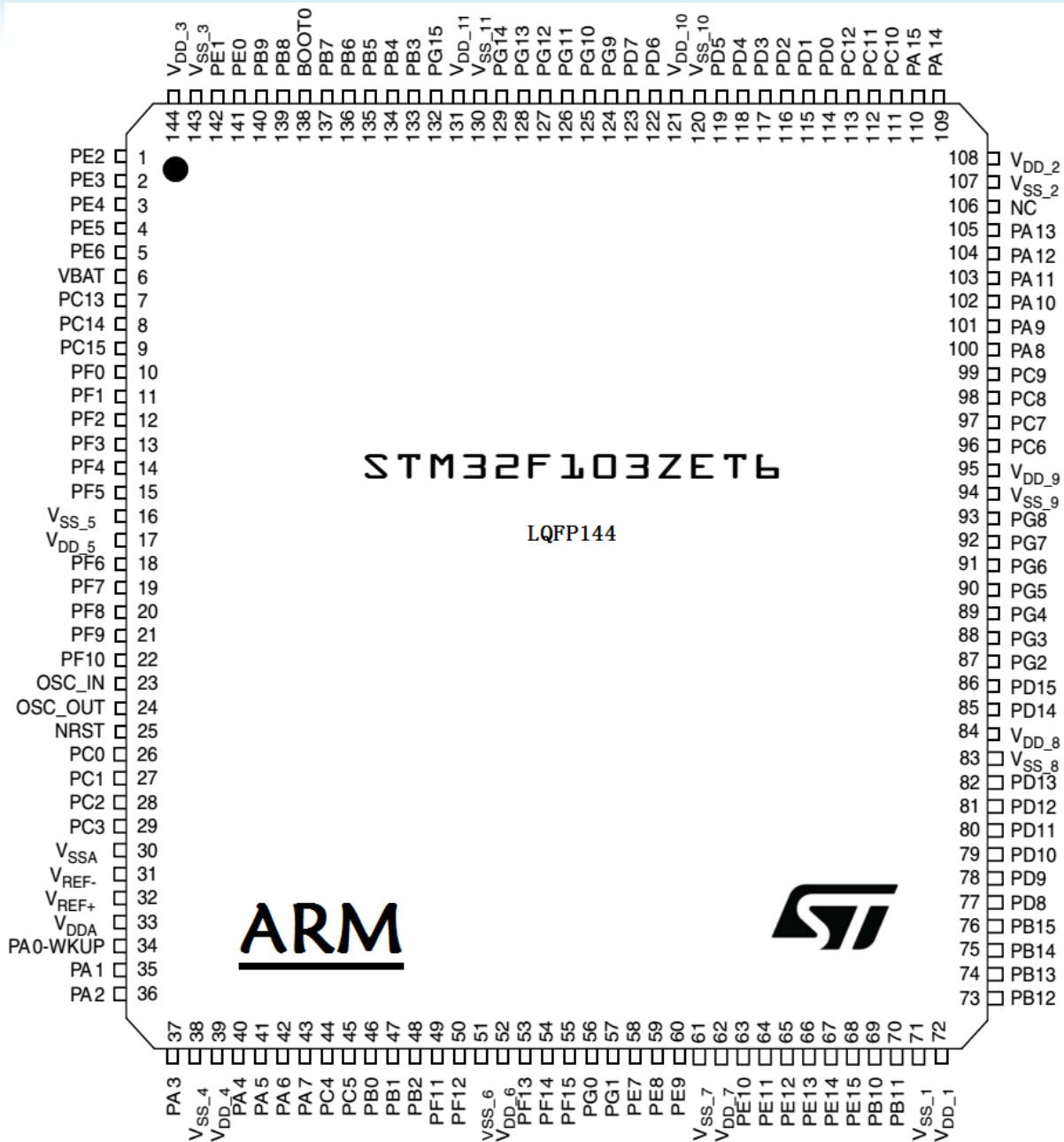
广义的最小系统

除包括上述电路外，还包括基本的人机接口电路，如指示灯、按键、蜂鸣器、数码管及串口通信等电路



CPU模块

ST公司的STM32F103ZET6





STM32F103ZET6芯片的主要特性如下：

- 集成32位的ARM Cortex-M3内核，最高工作频率可达72MHz，计算能力为1.25DMIPS/MHz（Dhrystone 2.1），具有单周期乘法指令和硬件除法器
- 具有512kB片内FLASH存储器和64kB片内SRAM存储器；
- 内部集成了8MHz晶体振荡器，可外接4~16MHz时钟源；
- 2.0V~3.6V单一供电电源，具有上电复位功能（POR）；
- 具有睡眠、停止、待机等三种低功耗工作模式；
- 144管脚LQFP封装（薄型四边引线扁平封装）；
- 内部集成了11个定时器：4个16位的通用定时器，2个16位的可产生PWM波控制电机的定时器，2个16位的可驱动DAC的定时器，2个加窗口的看门狗定时器和1个24位的系统节拍定时器（24位减计数）；



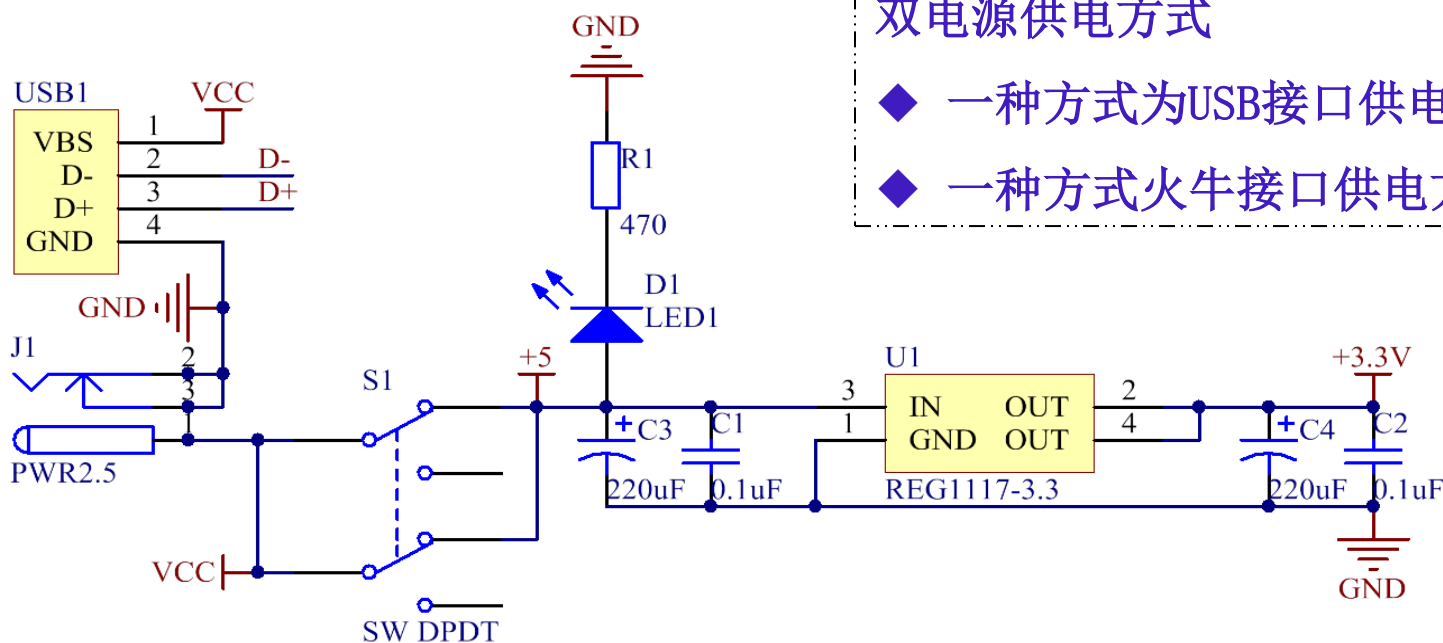
- 2个12位的DAC和3个12位的ADC（21通道）；
- 集成了内部温度传感和实时时钟RTC；
- 具有112根高速通用输入输出口（GPIO），可从其中任选16根作为外部中断输入口，几乎全部GPIO可承受5V输入（PA0~PA7、PB0~PB1、PC0~PC5、PC13~PC15和PF6~PF10除外）；
- 集成了13个外部通信接口：2个I2C、3个SPI（18Mbps，其中复用2个I2S）、1个CAN（2.0B）、5个UART、1个USB 2.0设备和1个并行SDIO；
- 具有12通道的DMA控制器，支持定时器、ADC、DAC、SDIO、I2S、SPI、I2C和UART外设；
- 具有96位的全球唯一编号；
- 工作温度为-40~85℃



最小硬件系统：电源模块

电源模块是系统工作的能量来源，其电压、纹波、内阻和驱动能力等性能直接影响到系统工作的稳定性，因此电源模块在系统设计中至关重要。

- 电源**电压**一定要在系统需求的范围之内
- 电源的**驱动能力**一定要能满足整个系统的功率需求
- 电源纹波和电路**干扰**的处理
- 在设计PCB时需要对模拟电源和数字电源进行物理上的隔离



双电源供电方式

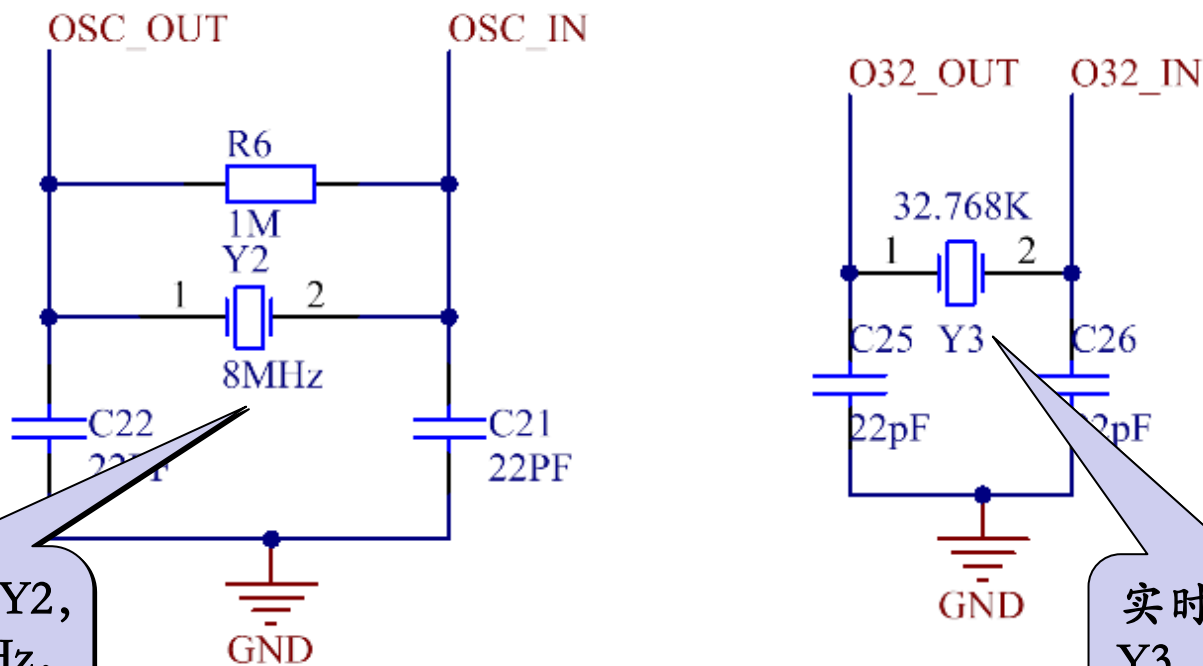
- ◆ 一种方式为USB接口供电方式；
- ◆ 一种方式为火牛接口供电方式，



最小硬件系统：时钟模块

时钟模块为系统提供同步工作信号，其稳定性直接关系到系统的工作稳定性。

- **HSI**时钟：**HSI**时钟信号，由内部**8MHz**的**RC**振荡器产生。
- **HSE**时钟：高速外部时钟信号(**4~16MHz**)由以下两种时钟源产生。
- **LSI**低速内部时钟信号，由内部**40KHz**的**RC**振荡器产生。
- **LSE**低速外部时钟信号，外部**32.768KHz**晶体产生。



系统主晶振Y2，
频率为 8MHz，
为STM32内核
提供振荡源；

每一个晶振的两端分别接上
两个22PF的对地微调电容

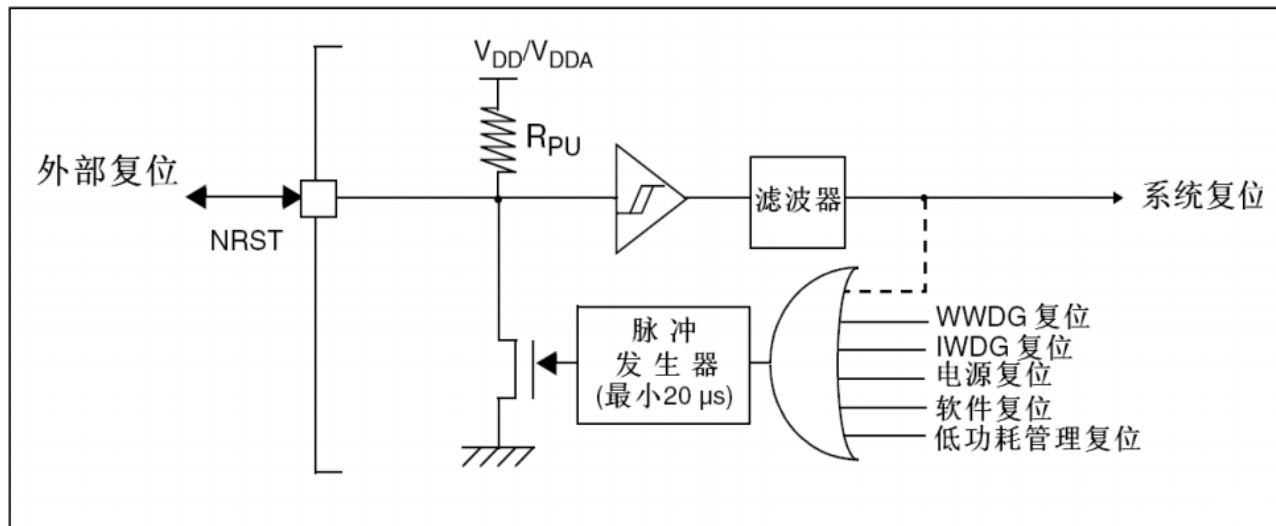
实时时钟晶振
Y3，频率为
32.768 KHz。



最小硬件系统：复位模块

系统上电复位、手动复位和内部复位

内部复位电路



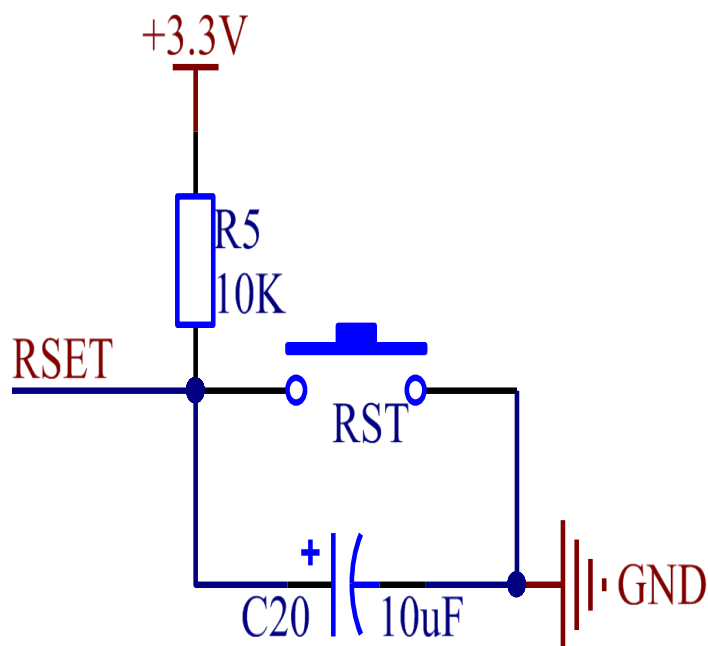
- 1) NRST引脚上的低电平(外部复位)
- 2) 窗口看门狗计数终止(WWDG复位)
- 3) 独立看门狗计数终止(IWDG复位)
- 4) 软件复位(SW复位)
- 5) 低功耗管理复位
- 6) 外部复位：一般工程中采用电源管理及复位芯片，实验设备中往往采用简单的RC复位电路。



系统上电复位、手动复位

一般来说系统对外部复位信号波形有一定的要求，若不能满足要求(例如持续时间过短)，则系统将不能正常工作。

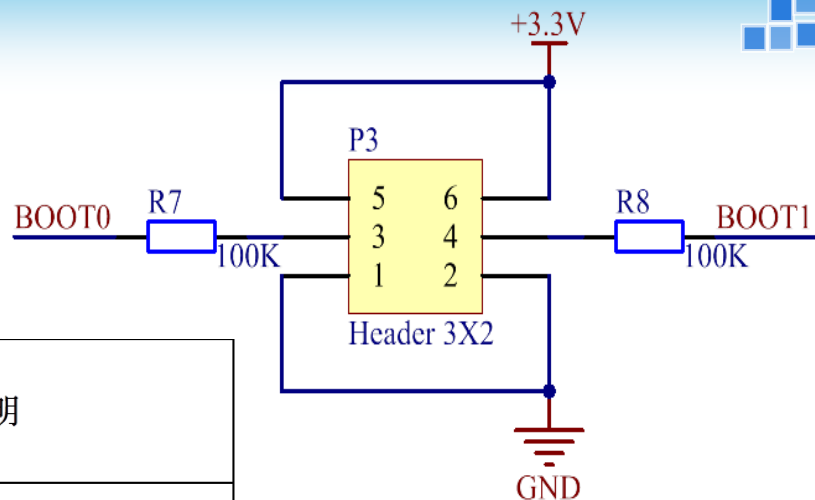
STM32低电平复位，电路其可以实现上电复位和按键复位。





复位后的启动设置电路

可以通过跳线帽来设置BOOT0引脚和BOOT1引脚的电平状态。



启动模式选择引脚		启动模式	说明
BOOT1	BOOT0		
X	0	从用户闪存启动	这是正常的工作模式
0	1	从系统存储器启动	启动的程序功能由厂家设置
1	1	从内置 SRAM 启动	这种模式可以用于调试

在系统复位后，SYSCLK的第4个上升沿，BOOT引脚的值将被锁存。用户可以通过设置BOOT1和BOOT0引脚的状态，来选择在复位后的如下3种启动模式：

- ① **片内Flash（主闪存存储器）启动：**片内Flash首地址0x0800 0000被映射到启动空间0x0000 0000，从片内Flash开始执行程序。
- ② **从系统存储器启动：**系统存储器被映射到启动空间0x0000 0000，从片内Flash开始执行程序。系统存储器首地址(互联型产品原有地址为0x1FFF B000，其它产品原有地址为0x1FFF F000)。
- ③ **从内置SRAM启动：**片内部SRAM首地址0x2000 0000被映射到启动空间0x0000 0000，从片内部SRAM首地址开始执行程序。



最小硬件系统：存储器模块

- 存储器模块为系统程序的保存和运行提供空间，在系统设计中主要根据处理器的存储器接口选择合适的存储器芯片（**存储类型、容量、速度和接口类型**）
- ARM最小系统中的存储器通常包括存放程序的NAND Flash和用于程序运行的SDRAM。
- 存储器模块通常挂接在ARM芯片的**局部总线上(外部三总线)**。

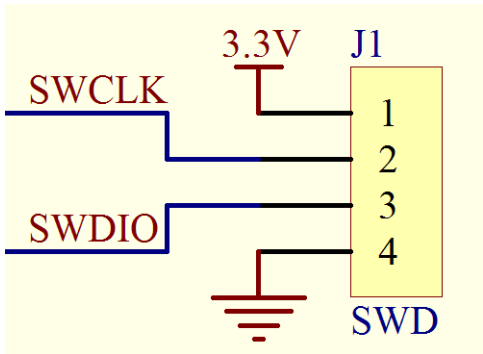
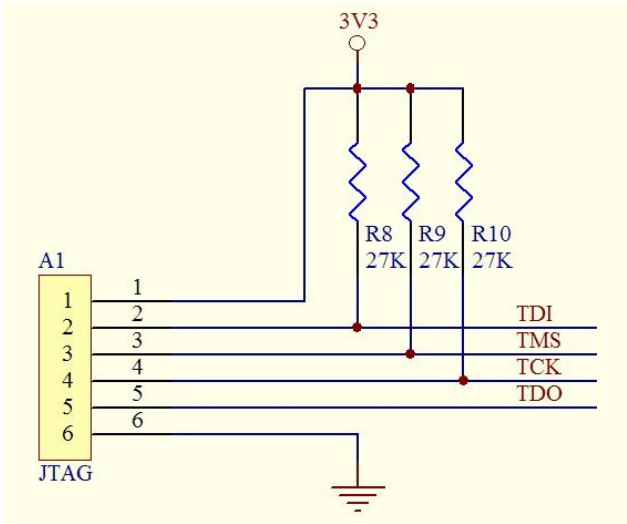


最小硬件系统：调试接口

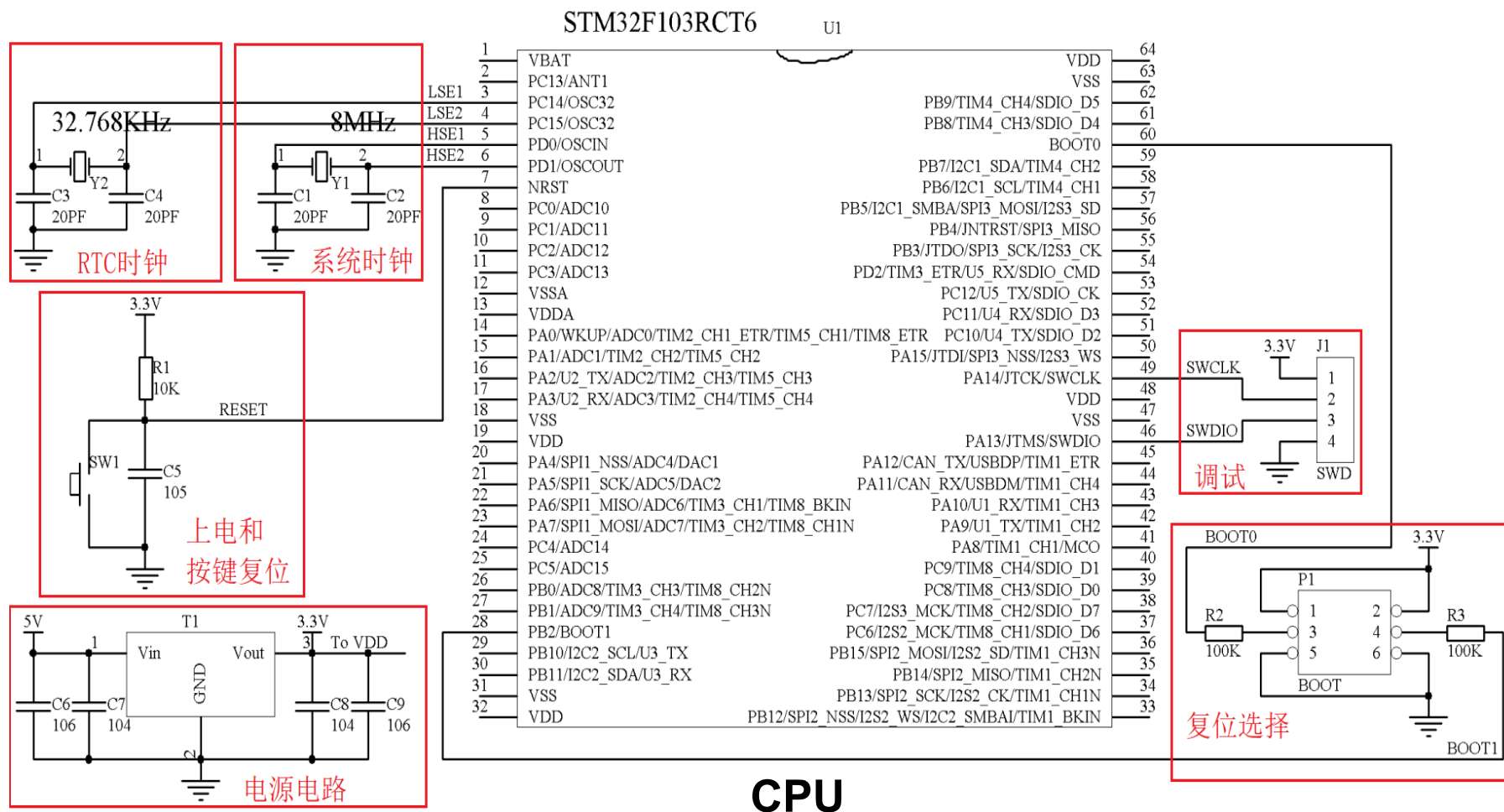
ARM微处理器一般都采用JTAG作为基本调试接口
nTRST, TMS, TCK, TDI和TDO

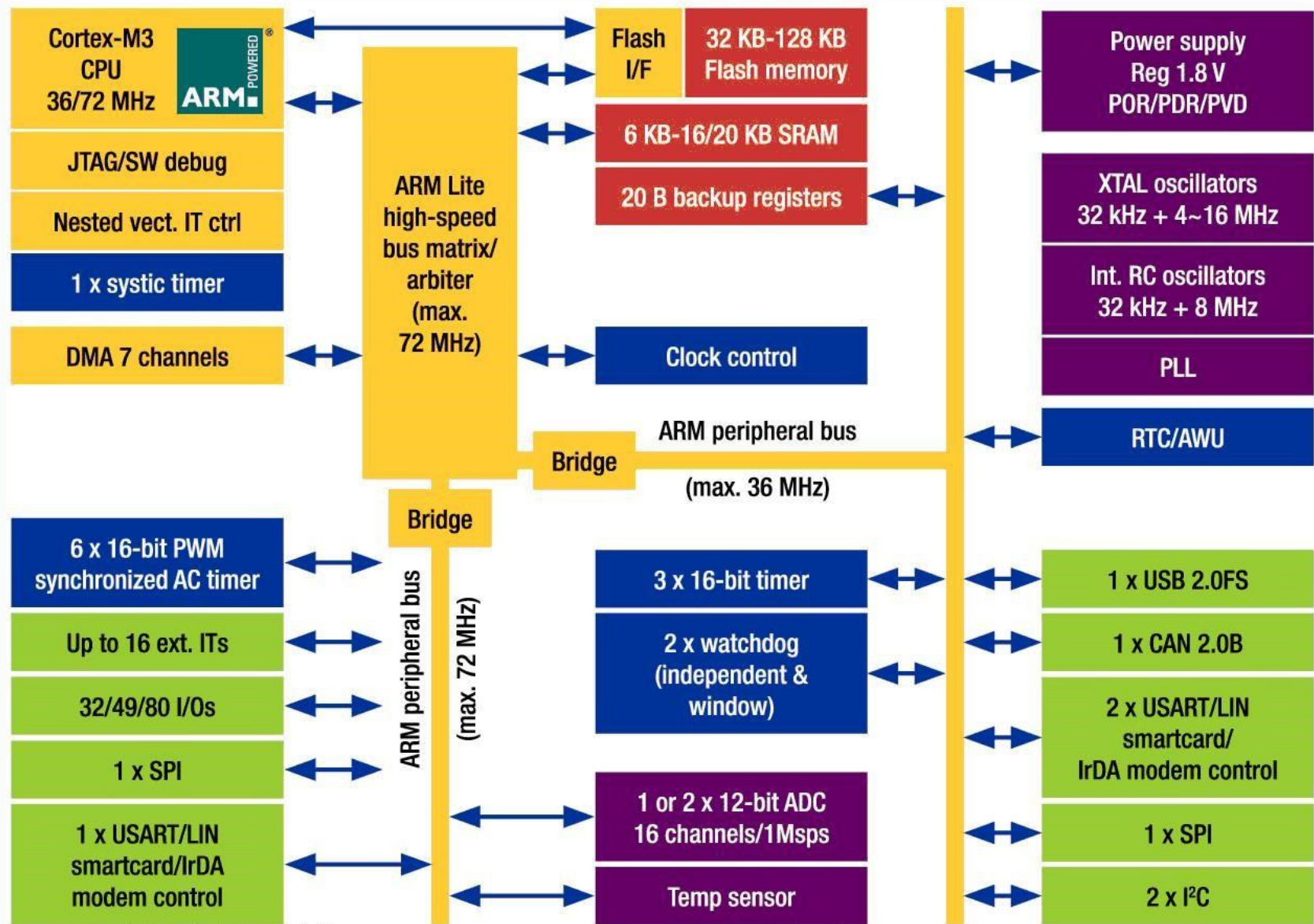
JTAG 引脚连接说明

SWJ-DP引脚名	JTAG调试端口		SWJ调试端口		引脚分配
	类型	描述	类型	调试分配	
JTMS/SWDIO	输入	JTAG测试模式选择	I/O	I/O	PA13
JTCK/SWDCLK	输入	JTAG测试时钟	输入	串行线时钟	PA14
JTDI	输入	JTAG测试数据输入			PA15
JTDO/TRACESWO	输出	JTAG测试数据输出		异步跟踪	PB3
JNTRST	输入	JTAG测试复位			PB4



实际的STM32F103最小系统





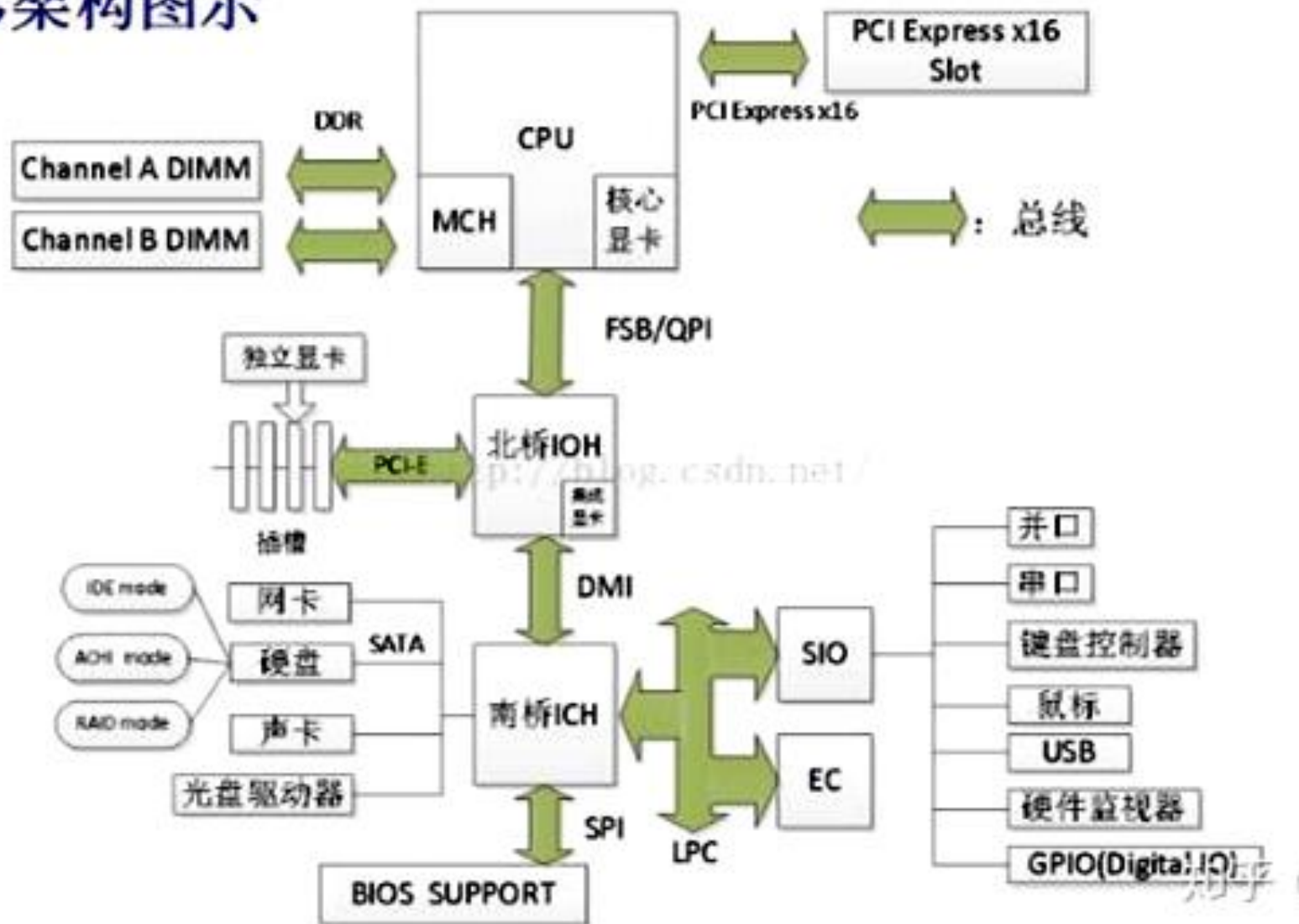
DMA:直接存储器访问
RTC:实时时钟
AWU:内置RTC报警的自动唤醒功能

POR:上电复位
PDR:掉电复位
PVD:可设置电压检测器

知乎 @大飒乔公子

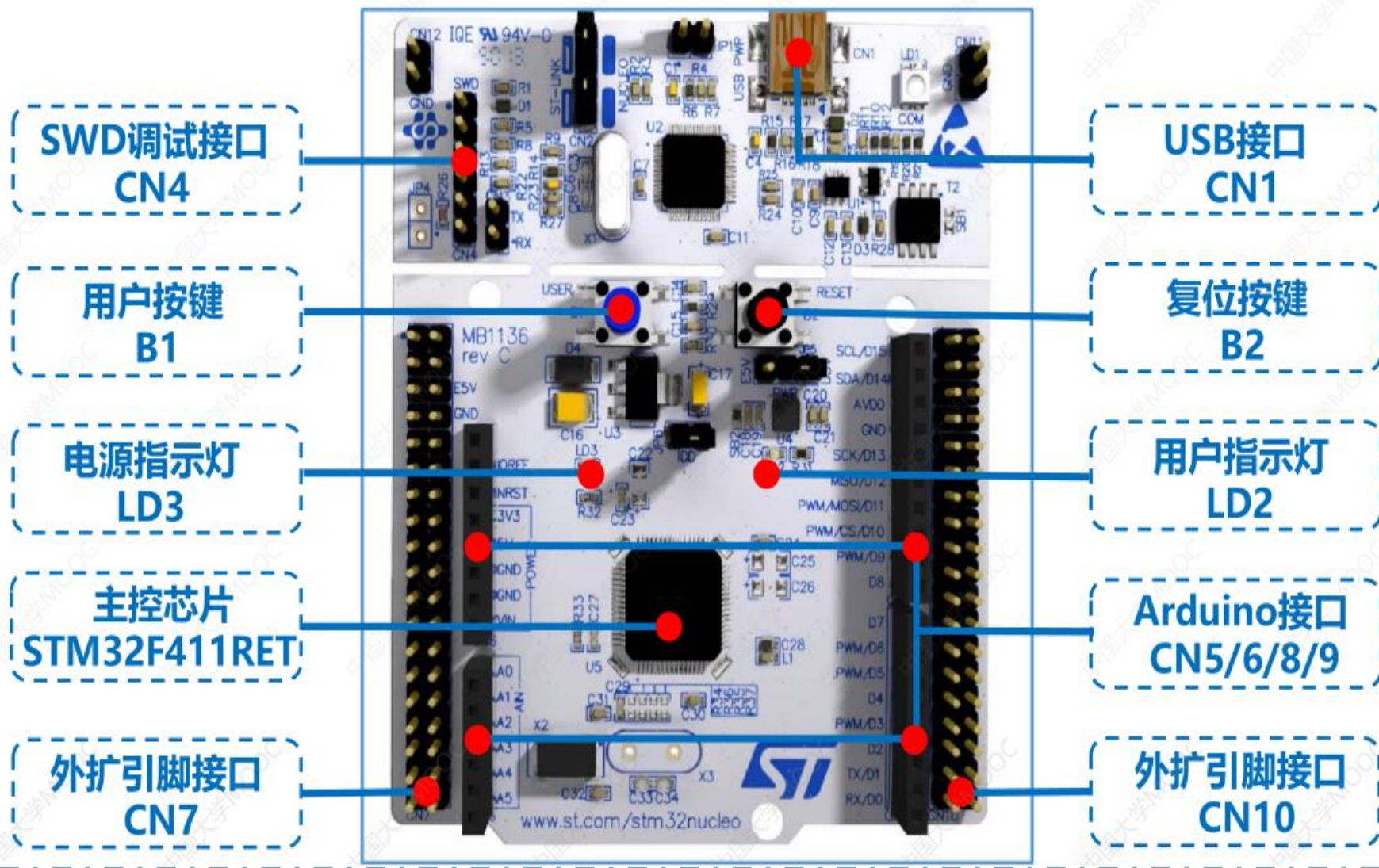


PC架构图示



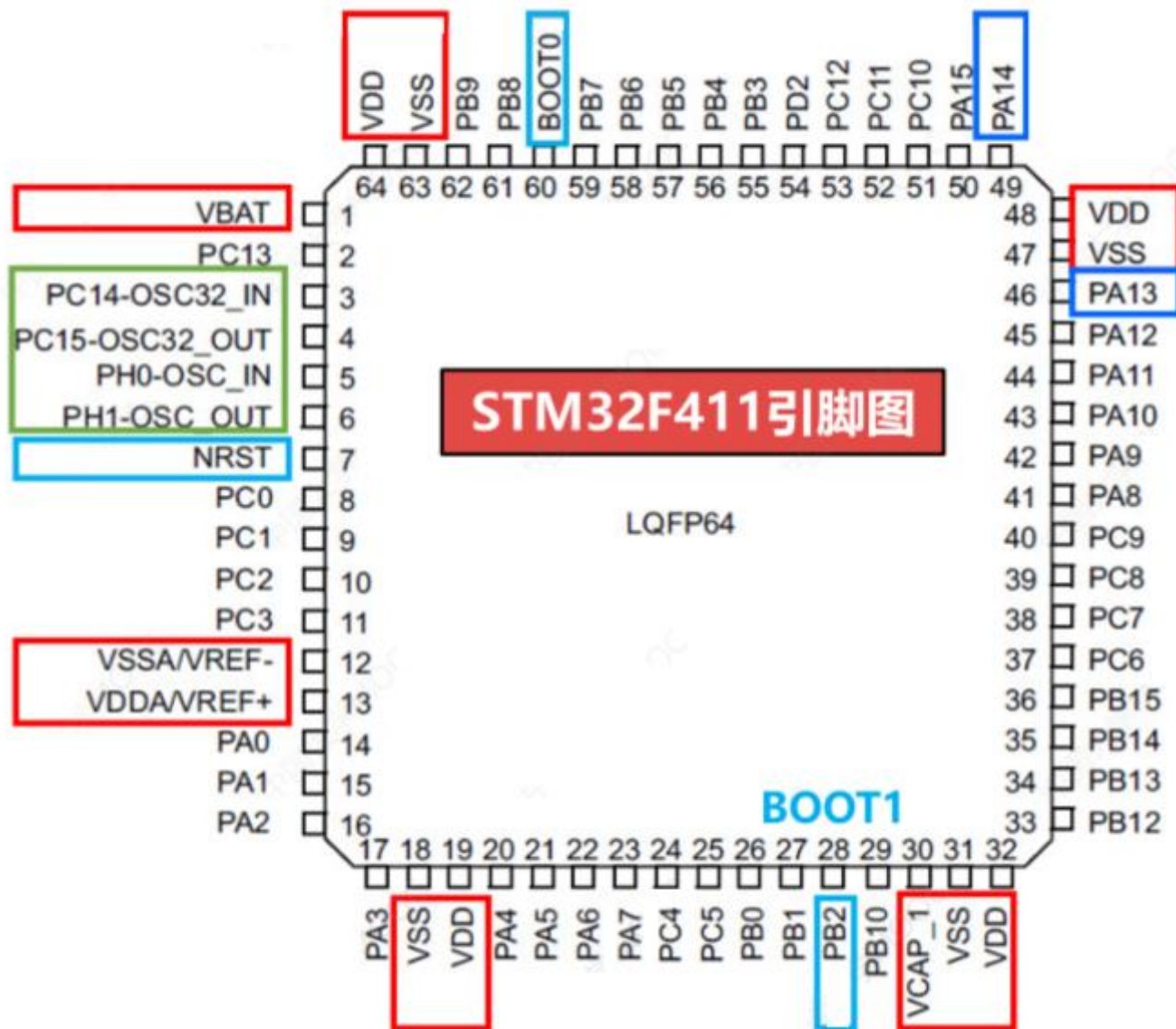


STM32 Nucleo开发板



STM32F411的64引脚

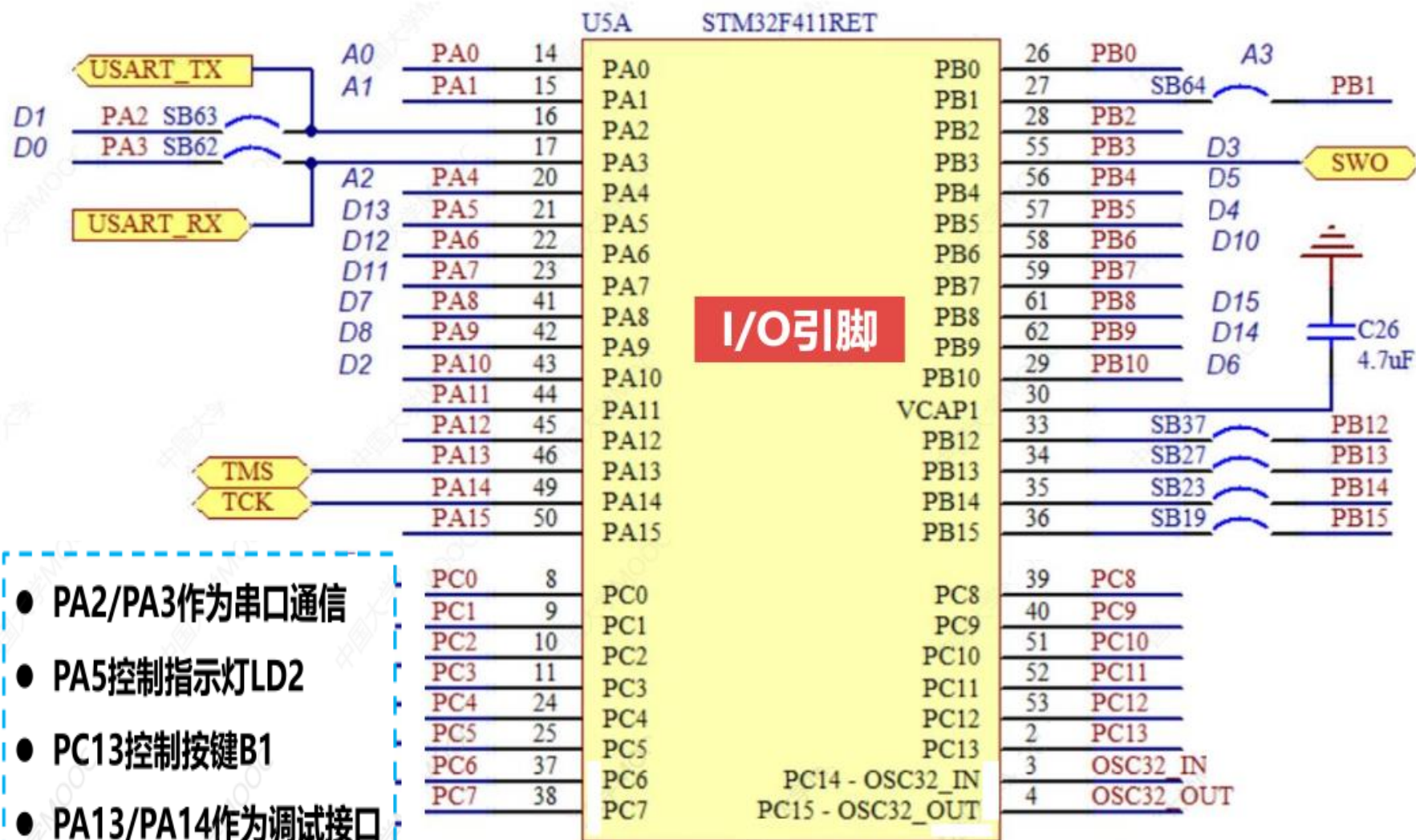
- 电源引脚 (3.3V)
- 复位及启动模式引脚
- 时钟引脚
- 仿真调试引脚
- 通用数字I/O引脚

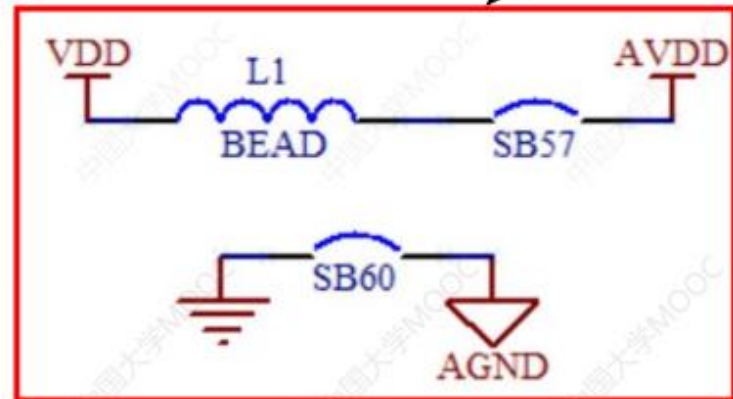
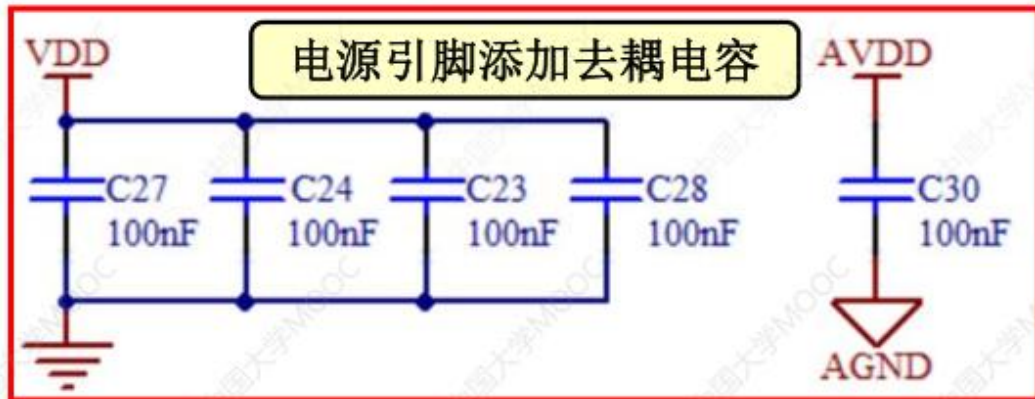




STM32F411RET6片内资源

- 512KB Flash和128KB SRAM
- 1个16位高级定时器, 2个32位通用定时器, 5个16位通用定时器
- 3路USART; 5路SPI/I2S; 3路I2C; 1路SDIO
- 1路12位16通道ADC
- 1个全速USB 2.0 OTG
- 50个通用数字I/O口



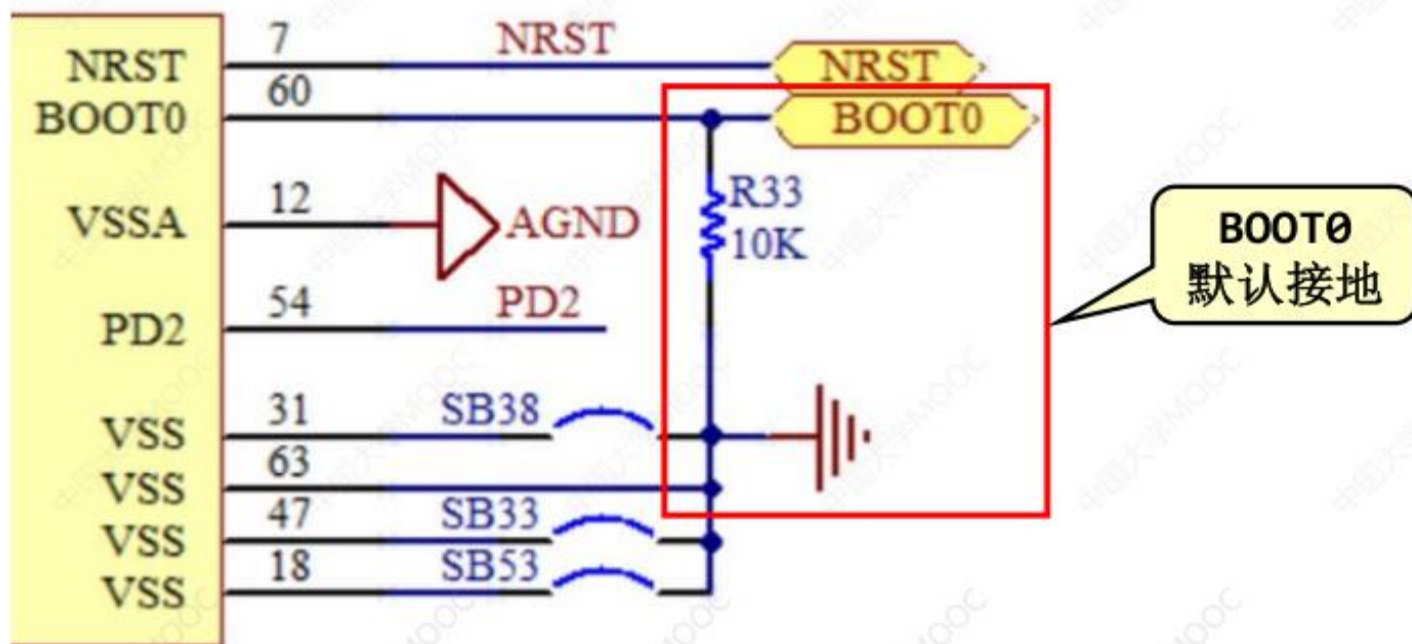


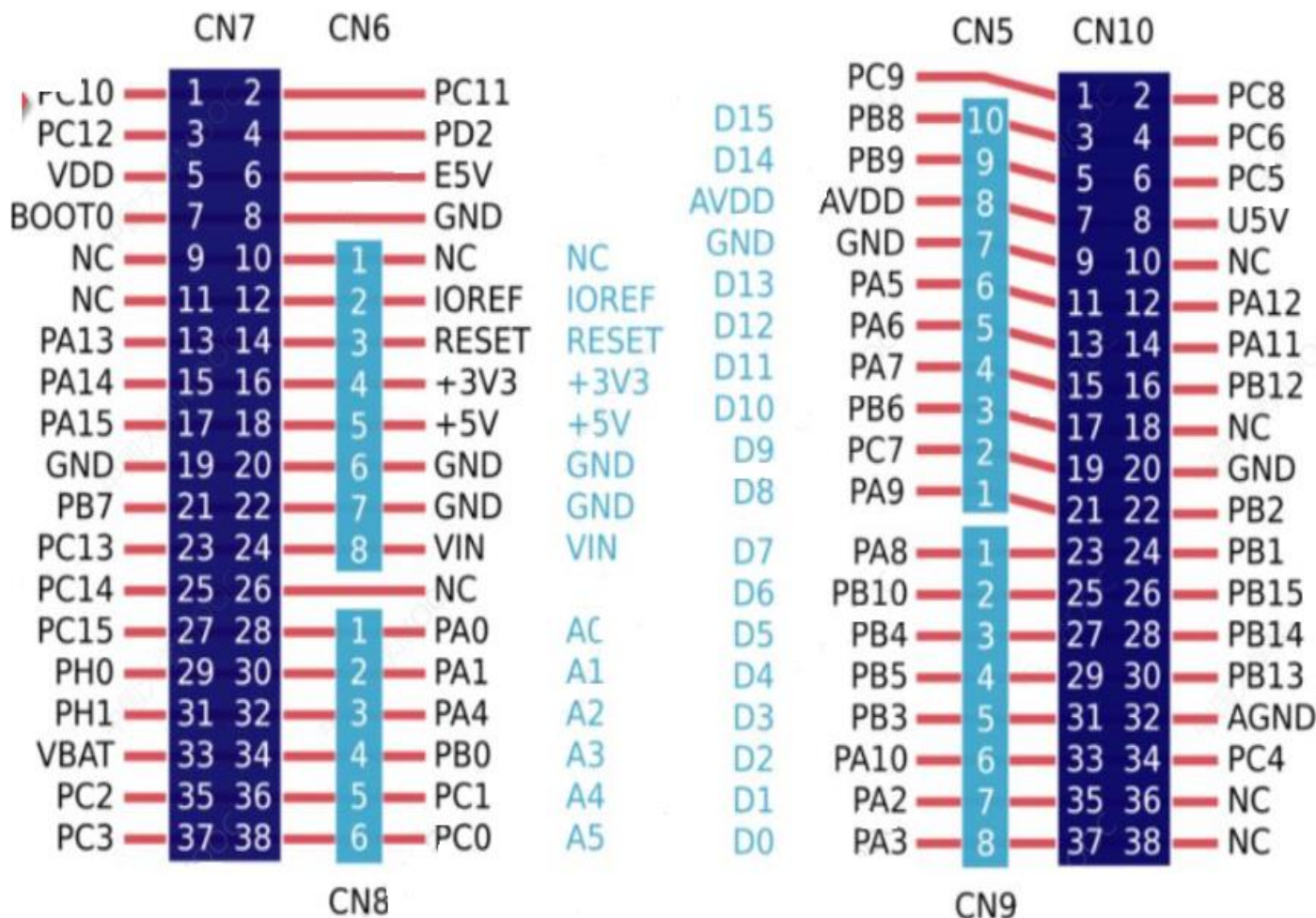
启动模式说明



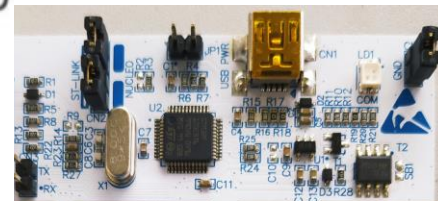
BOOT0	BOOT1	启动模式	说明
0	X	用户闪存存储器	从MCU片内Flash启动，常用启动方式
1	0	系统存储器	从系统存储器启动，主要用于串口下载 (ISP模式)
1	1	SRAM启动	从MCU片内SRAM启动，主要用于代码调试

用户闪存模式





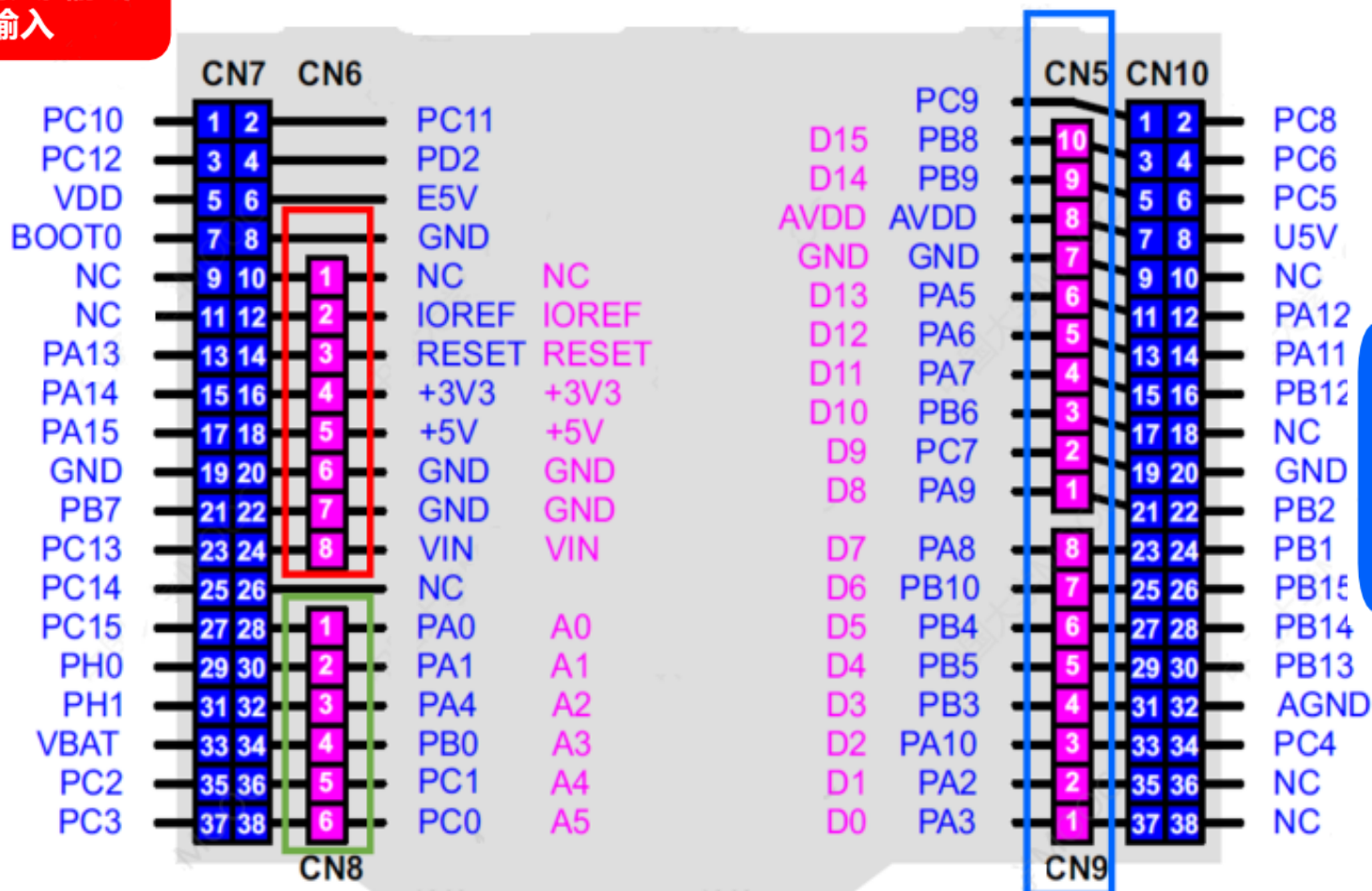
开发板的外扩接口





3.3V输入/输出
5V输入/输出
Vin输入

Arduino UNO接口



I2C
SPI
定时器
UART
GPIO

A/D接口

电源接口_CN6



序号	名称	对应STM32引脚	功能说明
1	NC	—	—
2	IOREF	—	输出3.3V直流电压
3	RESET	NRST	复位MCU
4	+3V3	—	外接3.3V直流电压输入或者输出3.3V直流电压
5	+5V	—	外接5V直流电压输入或者输出5V直流电压
6	GND	—	地
7	GND	—	地
8	VIN	—	外接7~12V直流电压输入

A/D接口_CN8

序号	名称	对应STM32引脚	功能说明
1	A0	PA0	ADC1_0 : ADC1的转换通道0
2	A1	PA1	ADC1_1 : ADC1的转换通道1
3	A2	PA4	ADC1_4 : ADC1的转换通道4
4	A3	PB0	ADC1_8 : ADC1的转换通道8
5	A4	PC1	ADC1_11: ADC1的转换通道11
6	A5	PC2	ADC1_10: ADC1的转换通道10



序号	名称	对应STM32引脚	功能说明
10	D15	PB8	I ² C1_SCL: I ² C1的时钟信号
9	D14	PB9	I ² C1_SDA: I ² C1的数据信号
8	AREF	—	模拟电压AVDD
7	GND	—	地
6	D13	PA5	SPI1_SCK: SPI1的时钟信号
5	D12	PA6	SPI1_MISO: SPI1的主机输入/从机输出信号
4	D11	PA7	TIM1_CH1N: 定时器1的捕获/比较通道1的互补输出 SPI1_MOSI : SPI1的主机输出/从机输入信号
3	D10	PB6	TIM4_CH1: 定时器4的捕获/比较通道1 SPI1_CS : SPI1的从机选择信号
2	D9	PC7	TIM3_CH2: 定时器2的捕获/比较通道2
1	D8	PA9	通用数字I/O

CN9

序号	名称	对应STM32引脚	功能说明
8	D7	PA8	通用数字I/O
7	D6	PB10	TIM2_CH3: 定时器2的捕获/比较通道3
6	D5	PB4	TIM3_CH1: 定时器3的捕获/比较通道1
5	D4	PB5	通用数字I/O
4	D3	PB3	TIM2_CH2: 定时器2的捕获/比较通道2
3	D2	PA10	通用数字I/O
2	D1	PA2	USART2_TX: 串口通信2的数据发送引脚
1	D0	PA3	USART2_RX: 串口通信2的数据接收引脚



原生引脚

Nucleo开发板原生用户引脚

序号	引脚名称	功能说明
1	PA5	控制指示灯LD2，高电平驱动方式
2	PC13	控制按键B1，上拉式驱动
3	PA2	UART2发送引脚，与ST-Link仿真器连接
4	PA3	UART2接收引脚，与ST-Link仿真器连接
5	PH0和PH1	外部低速时钟HSE
6	PC14和PC15	外部低速时钟LSI



STM32微控制器的四个时钟源

LSI



内部低速时钟。由芯片内部的RC振荡器提供，默认频率为32KHz

HSI



内部高速时钟。由芯片内部的RC振荡器提供，默认频率为16MHz

HSE



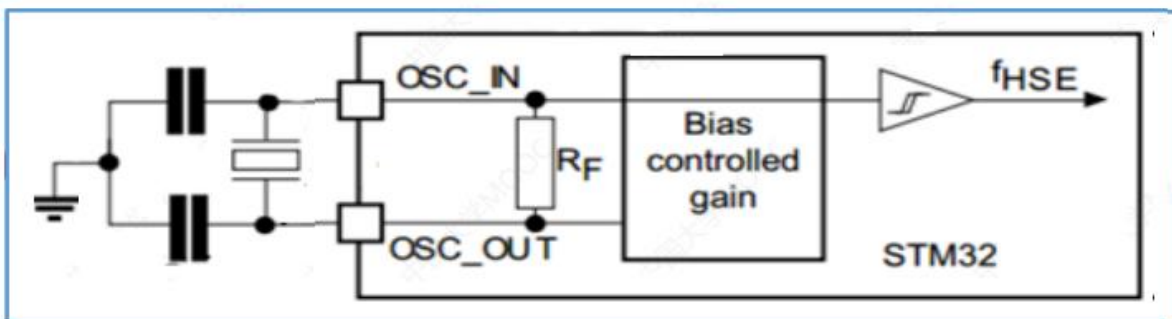
外部高速时钟。通过在OSC_IN和OSC_OUT引脚接入晶振实现，频率范围为4MHz ~ 26MHz。也可以直接接入外部时钟信号，频率范围为1MHz ~ 50MHz

LSE

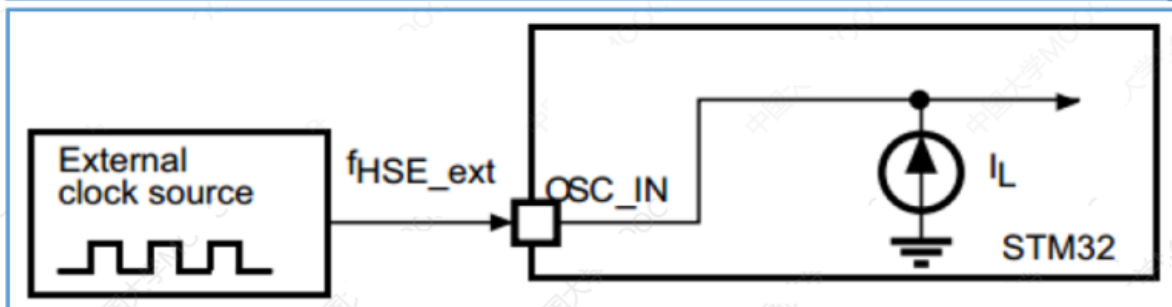


外部低速时钟。通过在OSC32_IN和OSC32_OUT引脚接入32.768KHz的晶振实现

振荡模式



旁路模式

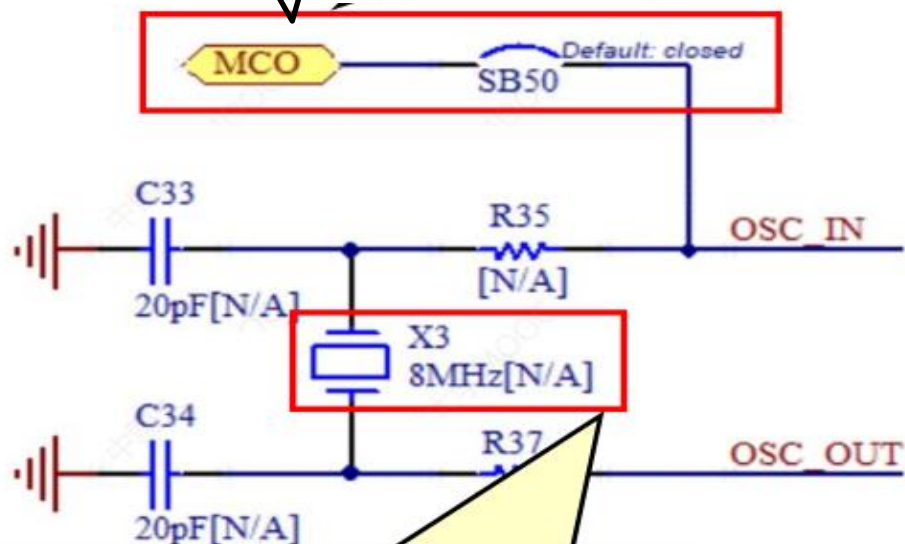


外部时钟输入

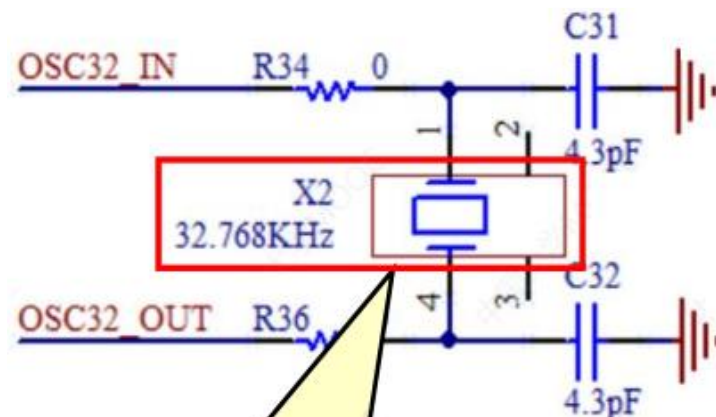
添加外部晶振

- ① 断开焊桥SB16、SB50、SB54和SB55;
- ② 焊接8MHz石英晶体X3;
- ③ 焊接2个20pF电容C33和C34;
- ④ 焊接2个0欧电阻R35和R37或者用焊锡短接

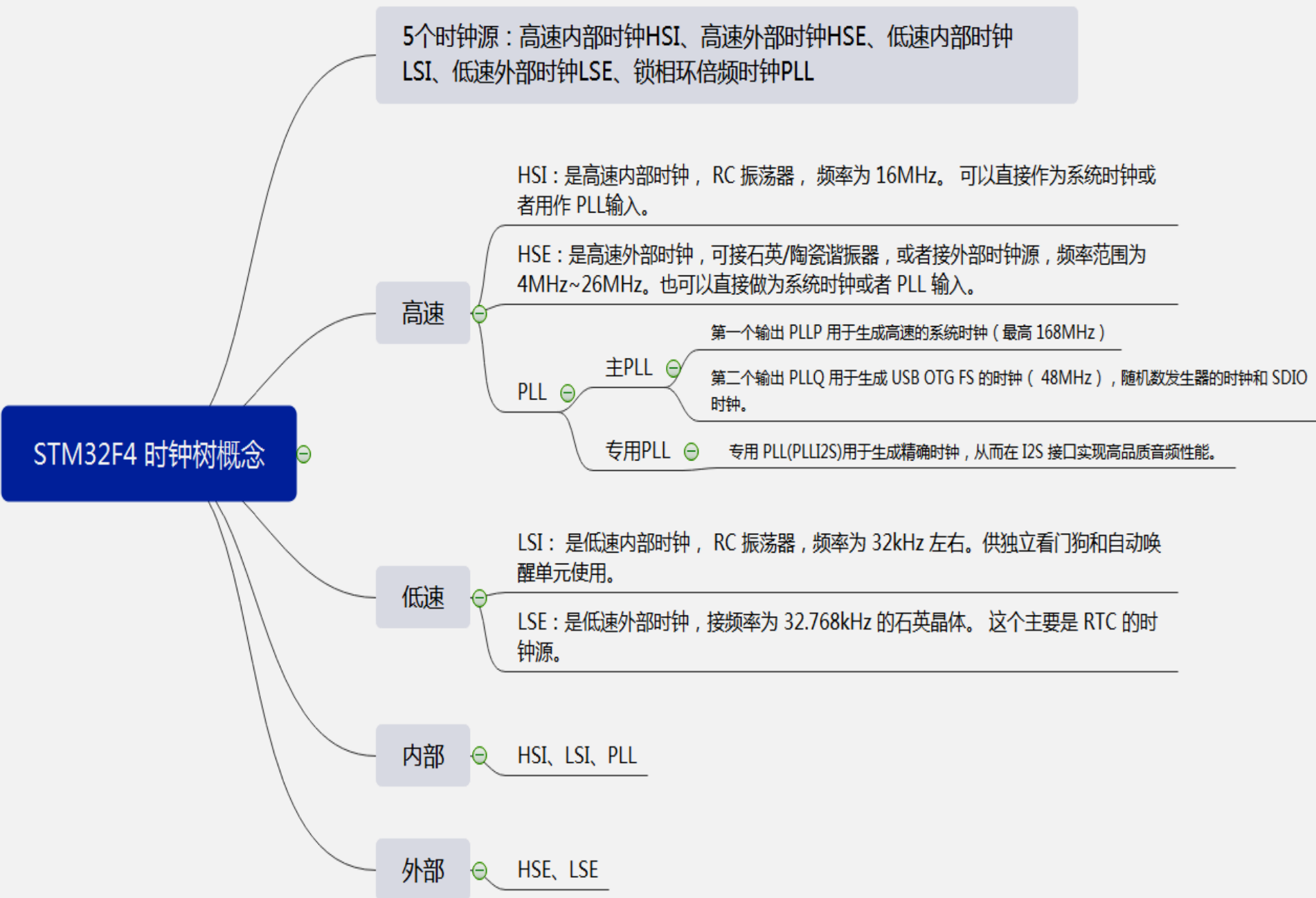
来自ST-Link仿真器的8MHz时钟



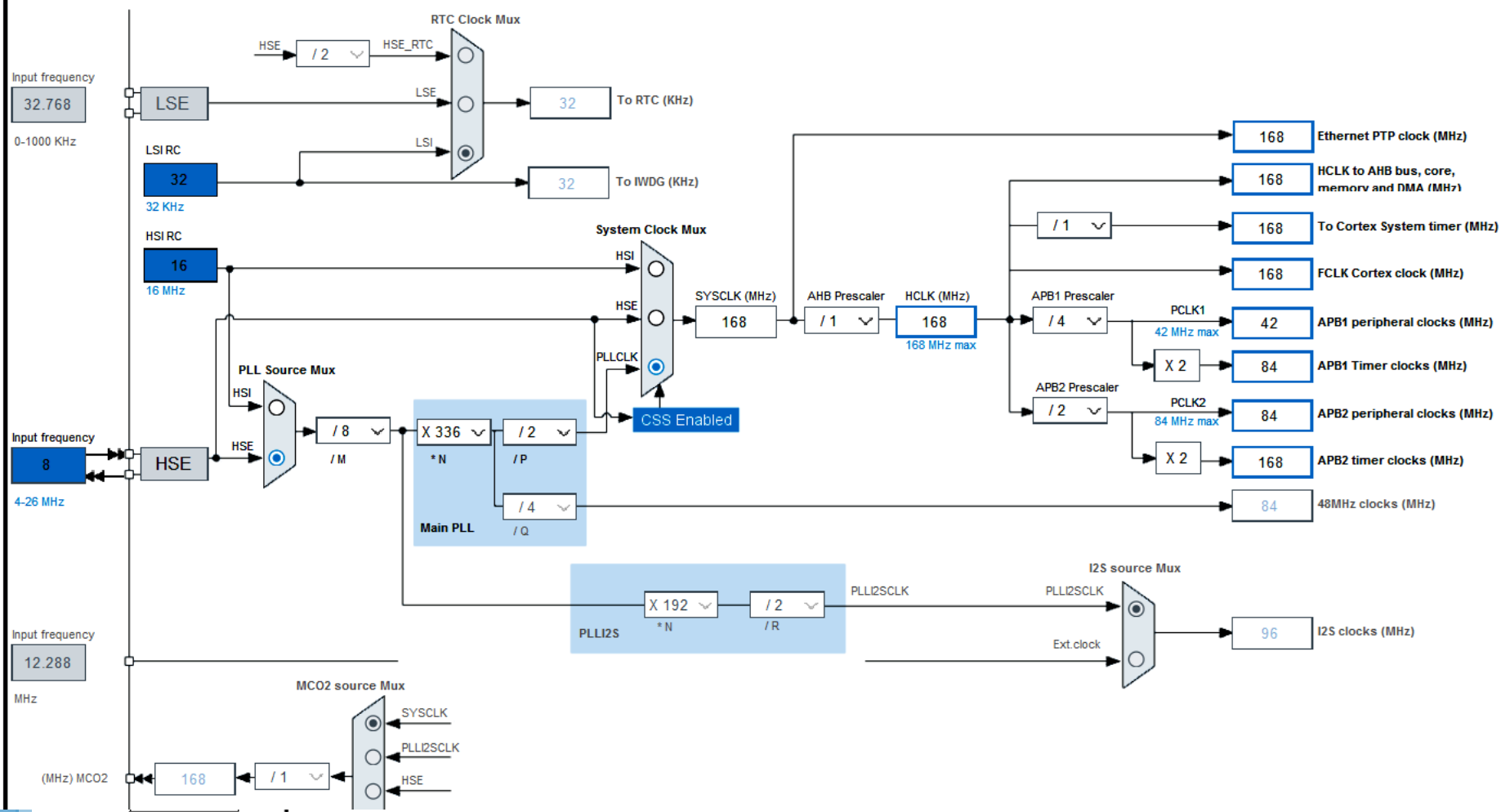
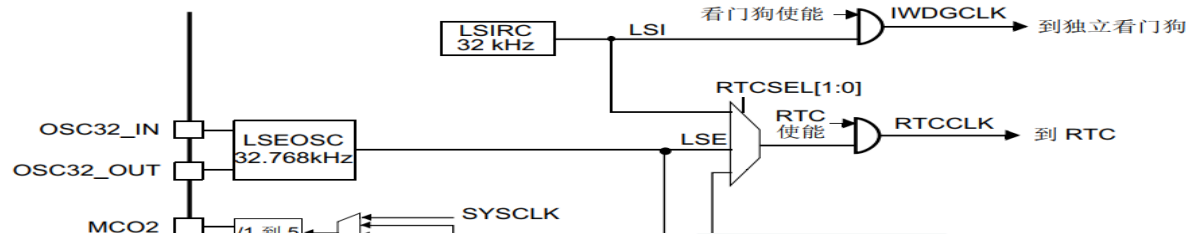
外部高速时钟HSE，默认未焊接



外部低速时钟LSE



STM32F4时钟树与时钟源





STM32F4的总线与存储器架构

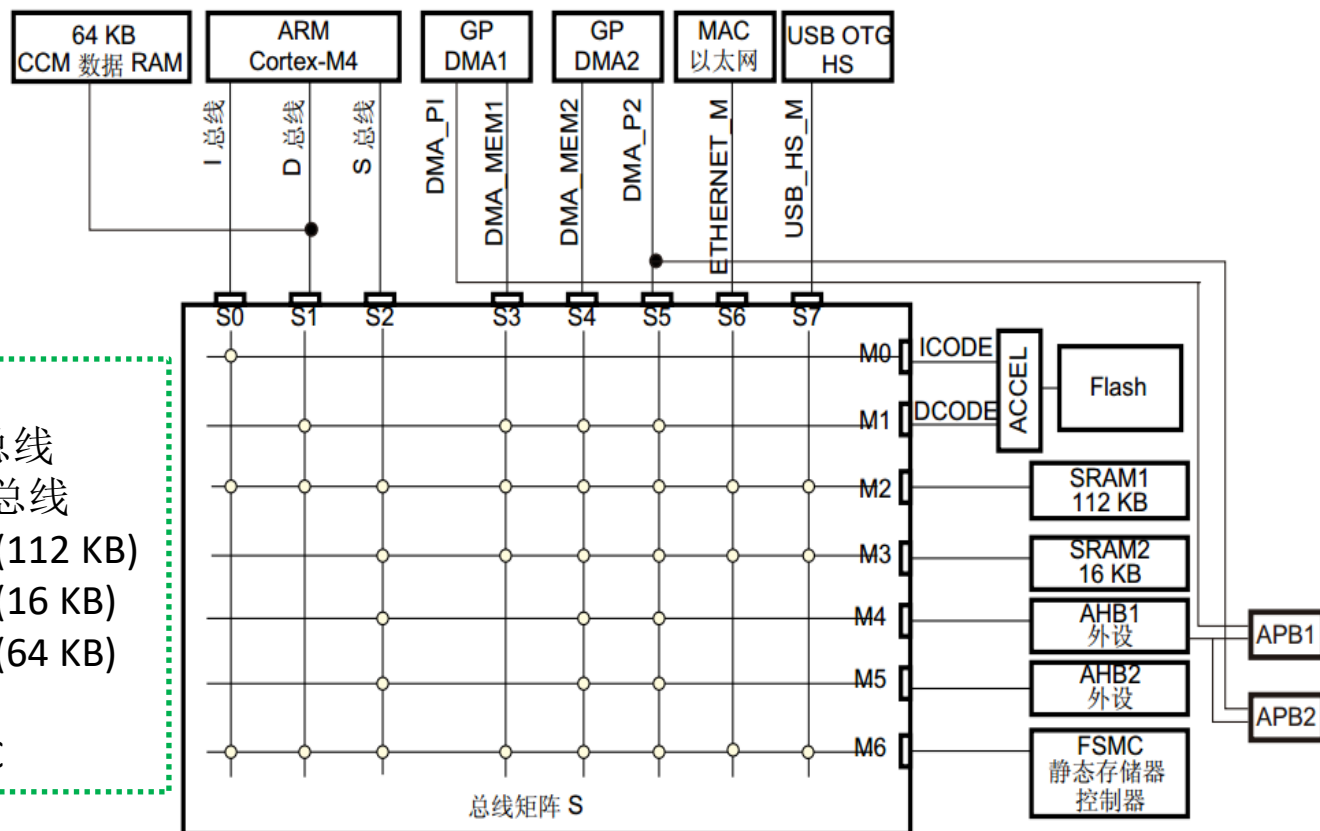
8条主控总线:

- (1) Cortex™-M4F内核I总线、D总线和S总线
- (2) DMA1存储器总线
- (3) DMA2存储器总线
- (4) DMA2外设总线
- (5) 以太网DMA 总线
- (6) USB OTG HS DMA 总线

1 总线架构

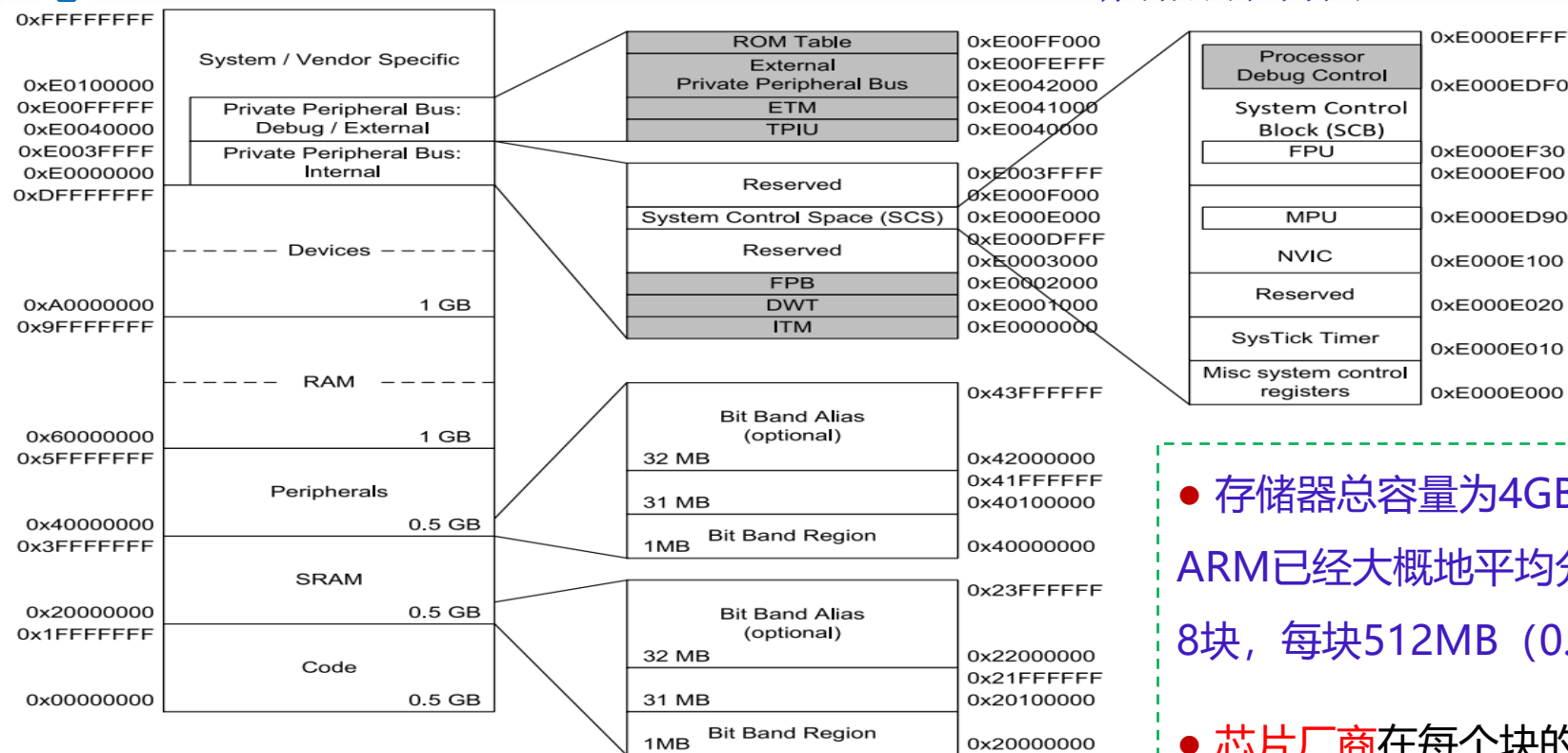
7条被控总线:

- (1) 内部Flash ICode总线
- (2) 内部Flash DCode总线
- (3) 主要内部SRAM1 (112 KB)
- (4) 辅助内部SRAM2 (16 KB)
- (5) 辅助内部SRAM3 (64 KB)
- (6) AHB1外设
- (7) AHB2外设—FSMC



2 存储器架构

存储器架构图



- 存储器总容量为4GB，ARM已经大概地平均分成了8块，每块512MB (0.5G)
- 芯片厂商在每个块的范围
- 内设计各具特色的外设，但是只用了其中的一部分而已，未分配给片上存储器和外设的存储区域均视为“保留区”。地址是由厂家规定好的，用户只能用而不能改。

序号	用途	地址范围
Block0	Code	0x0000 0000 ~ 0x1FFF FFFF(512MB)
Block1	SRAM	0x2000 0000 ~ 0x3FFF FFFF(512MB)
Block2	片上外设	0x4000 0000 ~ 0x5FFF FFFF(512MB)
Block3	FSMC的bank1 ~ bank2	0x6000 0000 ~ 0x7FFF FFFF(512MB)
Block4	FSMC的bank3 ~ bank4	0x8000 0000 ~ 0x9FFF FFFF(512MB)
Block5	FSMC寄存器	0xA000 0000 ~ 0xCFFF FFFF(512MB)
Block6	没有使用	0xD000 0000 ~ 0xDFFF FFFF(512MB)
Block7	Cortex-M4 内部外设	0xE000 0000 ~ 0xFFFF FFFF(512MB)



STM32F4xx 寄存器边界地址

边界地址	外 设	总线	边界地址	外 设	总线
0xA000 0000～ 0xA000 0FFF	FSMC 控制寄存器	AHB3	0x4002 4000～ 0x4002 4FFF	BKPSRAM	AHB1
0x5006 0800～ 0x5006 0BFF	RNG	AHB2	0x4002 3C00～ 0x4002 3FFF	Flash 接口寄存器	
0x5006 0400～ 0x5006 07FF	HASH		0x4002 3800～ 0x4002 3BFF	RCC	
0x5006 0000～ 0x5006 03FF	CRYP		0x4002 3000～ 0x4002 33FF	CRC	
0x5005 0000～ 0x5005 03FF	DCMI		0x4002 2000～ 0x4002 23FF	GPIOI	
0x5000 0000～ 0x5003 FFFF	USB OTG FS	AHB1	0x4002 1C00～ 0x4002 1FFF	GPIOH	
0x4004 0000～ 0x4007 FFFF	USB OTG HS		0x4002 1800～ 0x4002 1BFF	GPIOG	
0x4002 9000～ 0x4002 93FF	以太网 MAC		0x4002 1400～ 0x4002 17FF	GPIOF	
0x4002 8C00～ 0x4002 8FFF			0x4002 1000～ 0x4002 13FF	GPIOE	
0x4002 8800～ 0x4002 8BFF			0x4002 0C00～ 0x4002 0FFF	GPIOD	
0x4002 8400～ 0x4002 87FF			0x4002 0800～ 0x4002 0BFF	GPIOC	
0x4002 6400～ 0x4002 67FF	DMA2		0x4002 0400～ 0x4002 07FF	GPIOB	
0x4002 6000～ 0x4002 63FF	DMA1		0x4002 0000～ 0x4002 03FF	GPIOA	

边界地址	外 设	总线	边界地址	外 设	总线
0x4001 6800~ 0x4001 6BFF	LCD-TFT	APB2	0x4000 7400~ 0x4000 77FF	DAC	APB1
0x4001 5800~ 0x4001 5BFF	SAI1		0x4000 7000~ 0x4000 73FF	PWR	
0x4001 5400~ 0x4001 57FF	SPI6		0x4000 6800~ 0x4000 6BFF	CAN2	
0x4001 5000~ 0x4001 53FF	SPI5		0x4000 6400~ 0x4000 67FF	CAN1	
0x4001 4800~ 0x4001 4BFF	TIM11		0x4000 5C00~ 0x4000 5FFF	I2C3	
0x4001 4400~ 0x4001 47FF	TIM10		0x4000 5800~ 0x4000 5BFF	I2C2	
0x4001 4000~ 0x4001 43FF	TIM9		0x4000 5400~ 0x4000 57FF	I2C1	
0x4001 3C00~ 0x4001 3FFF	EXTI		0x4000 5000~ 0x4000 53FF	USART5	
0x4001 3800~ 0x4001 3BFF	SYSCFG		0x4000 4C00~ 0x4000 4FFF	USART4	
0x4001 3400~ 0x4001 37FF	SPI4		0x4000 4800~ 0x4000 4BFF	USART3	
0x4001 3000~ 0x4001 33FF	SPI1		0x4000 4400~ 0x4000 47FF	USART2	
0x4001 2C00~ 0x4001 2FFF	SDIO		0x4000 4000~ 0x4000 43FF	I2S3ext	
0x4001 2000~ 0x4001 23FF	ADC1-ADC2-ADC3		0x4000 3C00~ 0x4000 3FFF	SPI3 / I2S3	
0x4001 1400~ 0x4001 17FF	USART6		0x4000 3800~ 0x4000 3BFF	SPI2 / I2S2	
0x4001 1000~ 0x4001 13FF	USART1		0x4000 3400~ 0x4000 37FF	I2S2ext	
0x4001 0400~ 0x4001 07FF	TIM8		0x4000 3000~ 0x4000 33FF	IWDG	
0x4001 0000~ 0x4001 03FF	TIM1		0x4000 2C00~ 0x4000 2FFF	WWDG	
0x4000 7C00~ 0x4000 7FFF	USART8	APB1	0x4000 2800~ 0x4000 2BFF	RTC&BKP Registers	
0x4000 7800~ 0x4000 7BFF	USART7		0x4000 2000~ 0x4000 23FF	TIM14	

边界地址	外 设	总线	边界地址	外 设	总线
0x4000 1C00～ 0x4000 1FFF	TIM13	APB1	0x4000 0C00～ 0x4000 0FFF	TIM5	APB1
0x4000 1800～ 0x4000 1BFF	TIM12		0x4000 0800～ 0x4000 0BFF	TIM4	
0x4000 1400～ 0x4000 17FF	TIM7		0x4000 0400～ 0x4000 07FF	TIM3	
0x4000 1000～ 0x4000 13FF	TIM6		0x4000 0000～ 0x4000 03FF	TIM2	



STM32F4XX框图

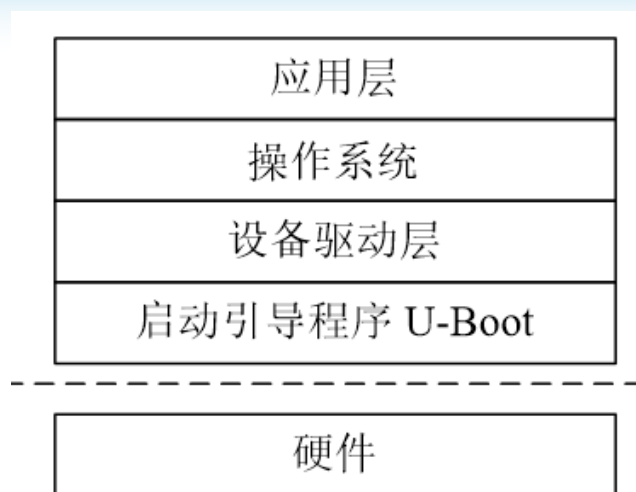


1. 高速功能需通过ULPI接口连接一个外部PHY 2. 只适用于STM32F417x和STM32F415x



嵌入式系统的软件基本组成

- **Bootloader**
- 板级支持包**BSP**
- 嵌入式操作系统
- 上层专用软件



软件层次结构示意图

Boot loader分析

- **PC机：BIOS**。完成初始化处理器配置、硬件初始化。
- 嵌入式系统自己编写程序：**Boot loader**。
- 系统加电后、操作系统内核或应用程序运行前执行。
- 完成：初始化硬件设备、建立内存空间映射图、将系统的软硬件环境设定在一个合适的状态，为最终调用操作系统内核运行用户应用程序准备好正确的环境。



引导加载程序Boot Loader是系统加电后运行的第一段软件代码

回忆一下 PC 的体系结构我们可以知道，PC 机中的引导加载程序由 BIOS(其本质就是一段固件程序)和位于硬盘 MBR 中的 OS Boot Loader（比如，Linux 中的 LILO 和 GRUB 等）一起组成。BIOS 在完成硬件检测和资源分配后，将硬盘 MBR 中的 Boot Loader 读到系统的 RAM 中，然后将控制权交给 OS Boot Loader。Boot Loader 的主要运行任务就是将内核映象从硬盘上读到 RAM 中，然后跳转到内核的入口点去运行，也即开始启动操作系统。

在嵌入式系统中，通常并没有像 BIOS 那样的固件程序（注，有的嵌入式 CPU 也会内嵌一段短小的启动程序），因此整个系统的加载启动任务就完全由 Boot Loader 来完成。比如在一个基于 ARM7TDMI core 的嵌入式系统中，系统在上电或复位时通常都从地址 0x00000000 处开始执行，而在这个地址处安排的通常就是系统的 Boot Loader 程序。

简单地说，Boot Loader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。

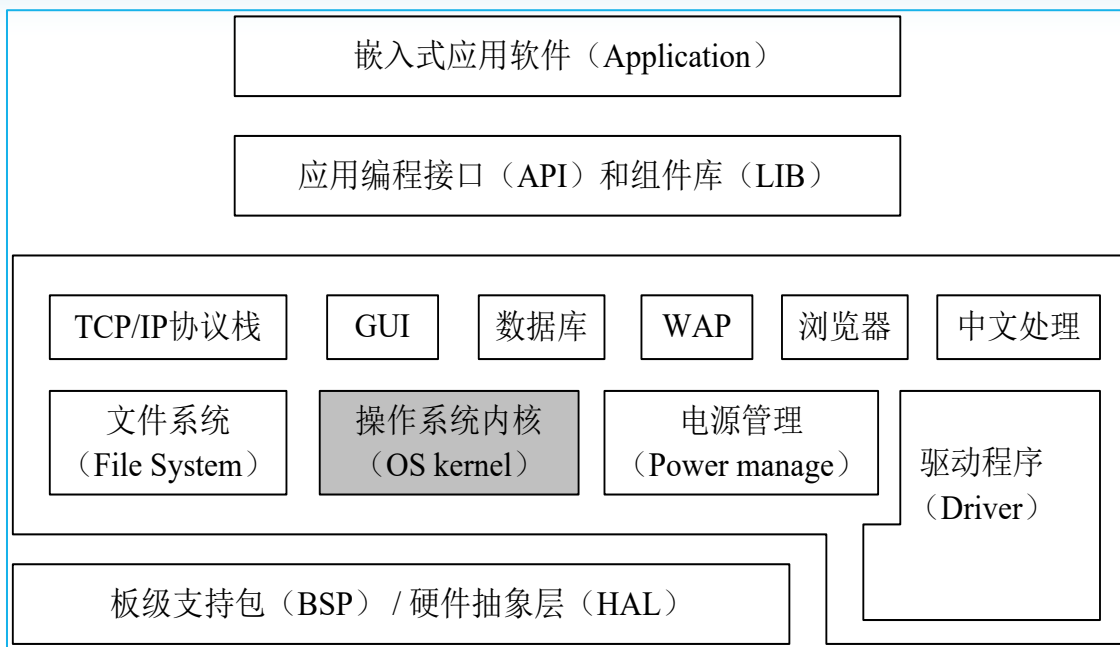


嵌入式系统的软件开发问题

- 1) 嵌入式操作系统。
- 2) 操作系统与应用软件的集成。
- 3) 软件的结构。
- 4) 嵌入式系统的软件是没有出口的，程序不能“退出”，整个程序的结构应该是无限的循环。
- 5) 嵌入式系统的软件设计需要考虑硬件的支持、操作系统的支持、程序的初始化和引导等诸多的方面。
- 6) 嵌入式系统的软件可能没有操作系统，在裸机上直接开发。



嵌入式软件从层次上看，包括板级支持包、嵌入式操作系统、中间件和应用软件。通常包括与硬件相关的底层驱动软件、系统内核、设备驱动接口、通信协议、图形界面、标准化浏览器等。



板级支持包BSP

- 它将系统中与硬件直接相关的一层软件独立出来，称之为Board Support Package即简称BSP。为嵌入式系统的硬件提供统一的**软件接口**；
- 实现应用程序的硬件无关性。它将具体的硬件设备和软件分离开，便于软件移植，是一种**硬件抽象层 (HAL)**。
- BSP的主要功能在于配置系统硬件使其工作于正常的状态，完成硬件与软件之间的信息交互，为OS及上层应用程序提供一个与硬件无关的软件平台。
- BSP位于硬件平台与操作系统之间，用于对上层软件屏蔽各种硬件相关性。



嵌入式操作系统具有通用操作系统的基本特点，如

- 能够有效管理越来越复杂的系统资源；
- 能够把硬件虚拟化，使得开发人员从繁忙的驱动程序移植和维护中解脱出来；
- 能够提供库函数、驱动程序、工具集以及应用程序。
- 与通用操作系统相比较，嵌入式操作系统在系统实时高效性、硬件的相关依赖性、软件固态化以及应用的专用性等方面具有较为突出的特点



1) 板级支持包

2) 嵌入式操作系统

3) 中间件

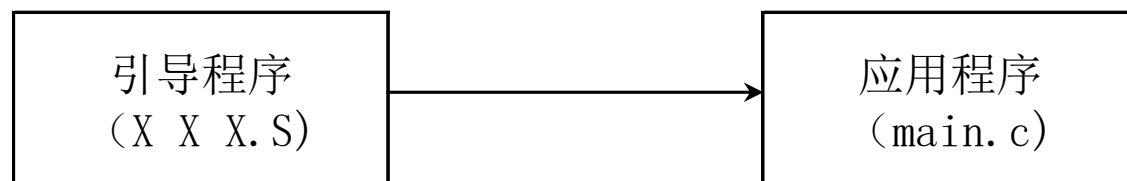
4) 应用软件



嵌入式系统的软件一般固化于嵌入式存储器中，是嵌入式系统的控制核心，控制着嵌入式系统的运行，实现嵌入式系统的功能。由此可见，嵌入式软件在很大程度上决定整个嵌入式系统的价值。

从软件结构上划分，嵌入式软件分为无操作系统和带操作系统两种。

◆ 无操作系统的嵌入式软件



➤ 引导程序

一般由汇编语言编写，在嵌入式系统上电后运行，完成自检、存储映射、时钟系统和外设接口配置等硬件初始化操作。

➤ 应用程序

一般由C语言编写，直接架构在硬件之上，在引导程序之后运行，负责实现嵌入式系统的主要功能。



◆ 带操作系统的嵌入式软件



带操作系统的嵌入式软件的体系结构如图所示，自下而上包括设备驱动层、操作系统层和应用软件层等。

相比无操作系统的嵌入式软件，带操作系统的嵌入式软件规模较大，其应用软件架构于嵌入式操作系统上，而非直接面对嵌入式硬件，可靠性高，开发周期短，易于移植和扩展，适用于功能复杂的嵌入式系统。

从理论上讲，基于操作系统的开发模式，具有快捷、高效的特点，开发的软件移植性、后期维护性、程序稳健性等都比较良好。但是，不是所有系统都要基于操作系统，因为这种模式要求开发者对操作系统的原理有比较深透的掌握，一般功能比较简单的系统，不建议使用操作系统，毕竟操作系统也占用系统资源；也不是所有系统都能使用操作系统，因为操作系统对系统的硬件有一定的要求。因此，在通常情况下，虽然STM32单片机是32位系统，但不主张嵌入操作系统。



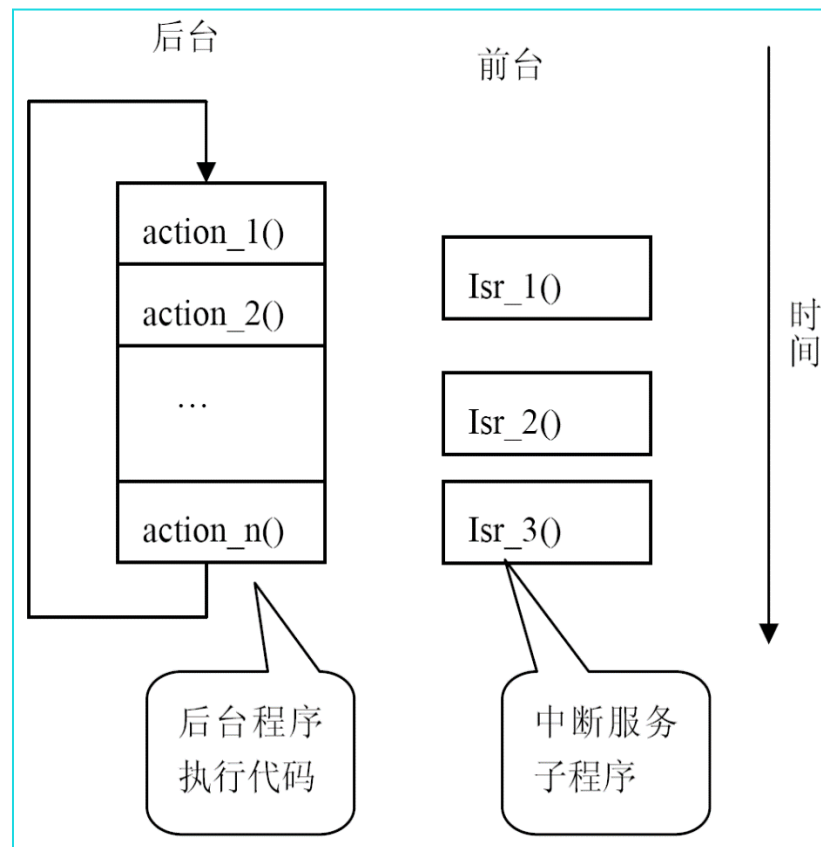
无操作系统的嵌入式系统软件设计方法

- 前后台系统
- 中断（事件）驱动系统
- 巡回服务系统
- 基于定时器的巡回服务系统

前后台系统

后台： 应用程序是一个无限循环，巡回地执行多个事件，完成相应的操作。这一部分软件称为后台。通常在主程序main()中被调用。

前台： 中断服务程序处理异步事件，这一部分可以看成是前台。



后台可以称为**任务级**，前台可以称为**中断级**。强实时性的关键操作一定要用中断来实现。中断服务程序提供数据（实时性数据）。



程序框架-后台

```
main()
{
    /* 硬件初始化 */
    while(1) /* 后台程序 */
    {
        action1();
        action2();
        action3();
        ...
    }
}
```

```
action_1()
{
    /* 执行动作n */
    ...
}

... ..
action_n()
{
    /* 执行动作n */
    ...
}
```

程序框架-前台

```
Isr_1()
{
    /* 中断1的中断服务程序 */
    ...
}

... ..
Isr_n()
{
    /* 中断2的中断服务程序 */
    ...
}
```



中断（事件）驱动系统

原理：整个嵌入式系统软件由中断服务程序构成；主程序完成系统的初始。

应用：低功耗系统设计。

构成：主程序完成系统的初始化；中断服务例程完成事务处理。

中断驱动系统-主程序

```
main() /*完成系统的硬件初始化和数据结构的初始化（如果必要的话）*/  
{  
    /* to do: 系统的初始化 */  
    while(1)  
        enter_low_power(); /*进入低功耗状态*/  
}
```

```
Isr_n() /* 其中的一个中断服务程序 */  
{  
    /* to do: 处理中断事件 */  
}
```



巡回服务系统(查询)

应用：嵌入式处理器/控制器的中断源不多

解决方案

- 增加中断源-需要硬件，成本高
- 软件方案-软件巡回服务

```
main()
{
    /* to do: 系统初始化 */
    while(1)
    {
        action_1();/*巡回检测事件1并处理事件*/
        action_2();/*巡回检测事件2并处理事件*/
        .....
        action_n();/*巡回检测事件n并处理事件*/
    }
}
```



基于定时器的巡回服务系统

- 普通巡回服务系统的缺点
 - 处理器全速运行, 开销大-功耗高-电池供电系统
 - 降低处理器的工作时间-基于定时器的巡回服务系统
- 构成
 - 主程序
 - 定时器中断服务程序

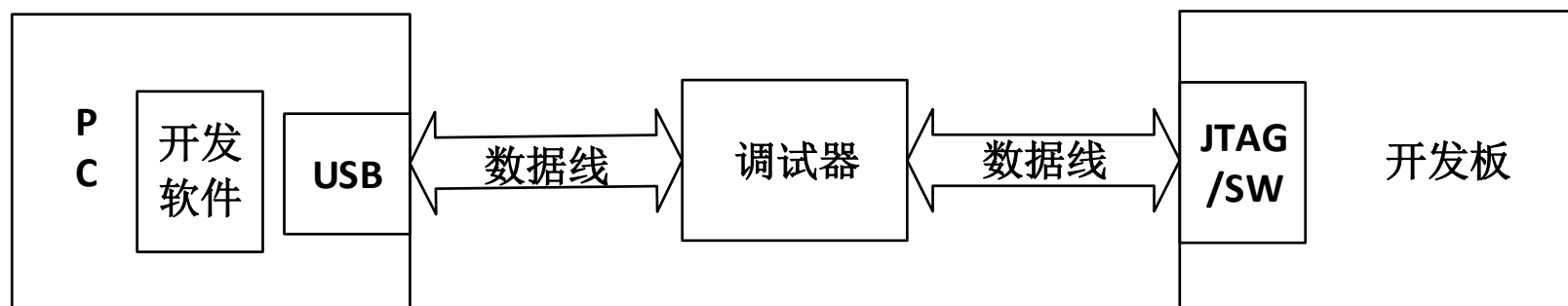
```
main()
{
    /* to do: 系统初始化 */
    /* to do: 设置定时器 */
    while(1)
    {
        enter_low_power();
    }
}
```

```
Isr_timer() /* 定时器的中断服务程序 */
{
    action_1(); /*执行事件1的处理*/
    action_2(); /*执行事件2的处理*/
    ...
    action_n(); /*执行事件n的处理*/
}
```

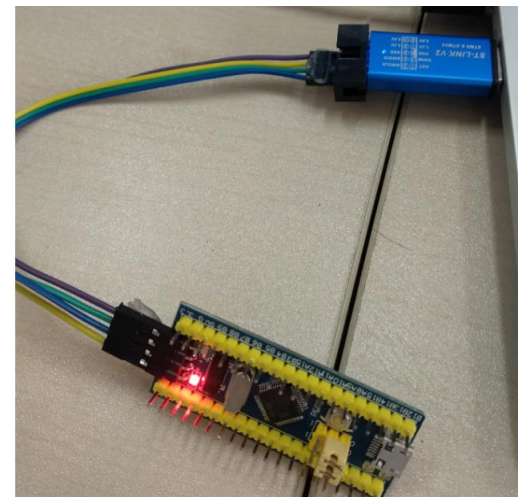


开发工具

进行STM32F系列微控制器的程序开发需要搭建一个交叉开发环境，其中包括计算机、开发软件、调试器、开发板或自己设计的电路板（包括STM32F系列微控制器）。



- 开发板；
- 软件集成开发工具；
- 调试工具；
- 其他：RTOS、开源协议栈。

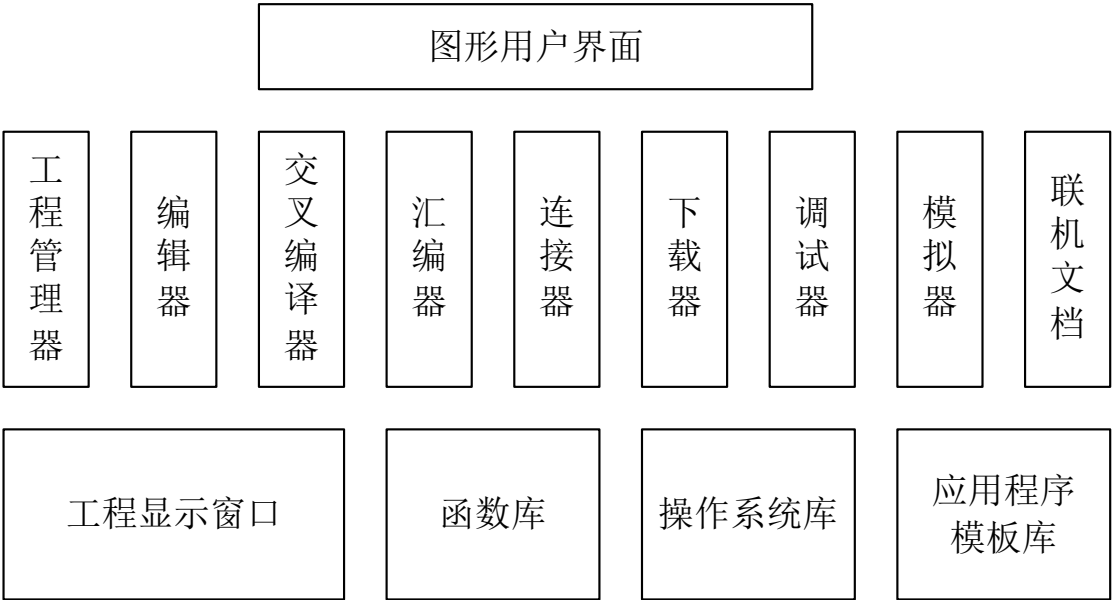




集成开发环境的一般性内部结构概略图解

IDE中的组件

- 1) 工程管理器
- 2) 操作系统库
- 3) 高级语言程序标准函数库
- 4) 函数库
- 5) 程序模板库



ARM开发工具根据功能的不同，可分为编译器、汇编器、连接器、调试器、嵌入式实时操作系统、函数库、评估板、**JTAG**仿真器、在线仿真器等。

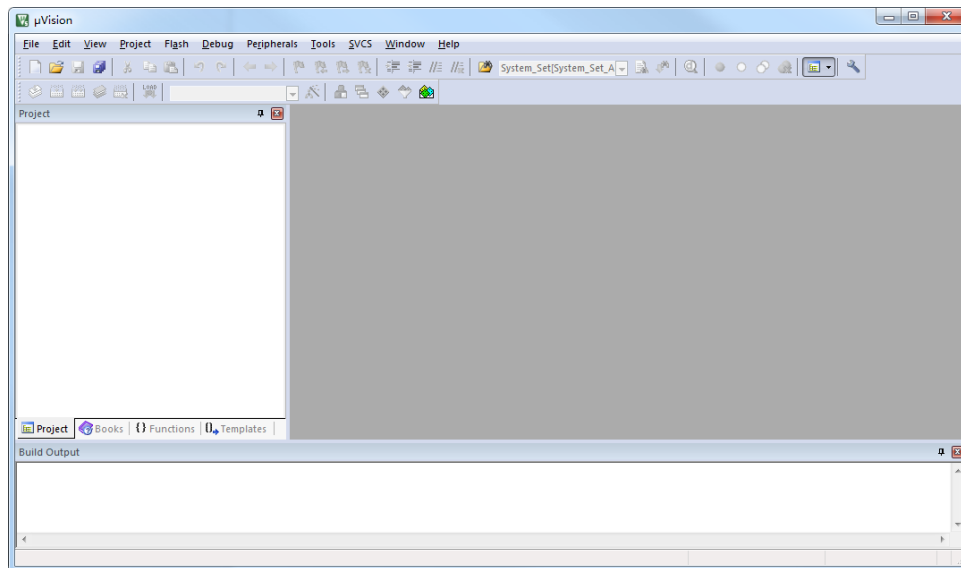


1) KEIL MDK

KEIL MDK 集成了业内最领先的技术，支持**Cortex-M, Cortex-A**等**ARM**内核处理器，集成**Flash**烧写模块等，具备针对不同调试器的在线调试功能，已经成为**ARM**软件开发工具的标准。

2) IAR for ARM

www.iar.com

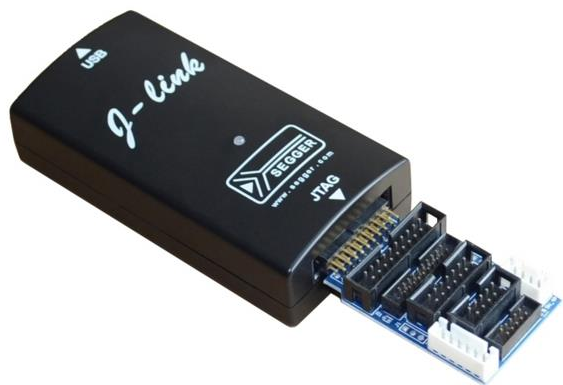


可访问www.keil.com获取更多内容。



调试工具

J-Link 是SEGGER 公司为支持仿真ARM 内核芯片推出的JTAG 仿真器。是通用的开发工具，可以用于**KEIL**，**IAR**，**ADS** 等平台 速度，效率，功能均比**ULINK**强。需要安装驱动。



ULINK是**KEIL**公司开发的仿真器，专用于**KEIL MDK**平台。在**KEIL MDK**平台下无需驱动，可直接使用。



3) STlink

STLink 是ST 公司为STM8和STM32系列MCU设计的调试器。需要安装驱动。

调试接口

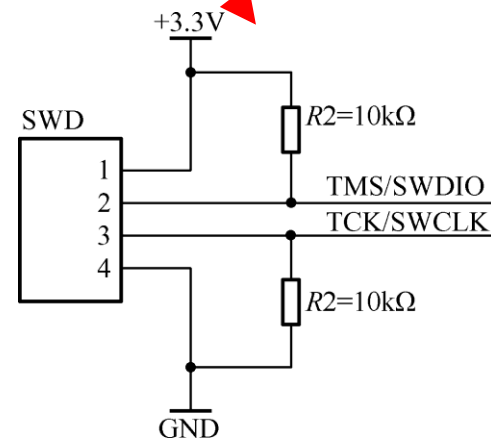
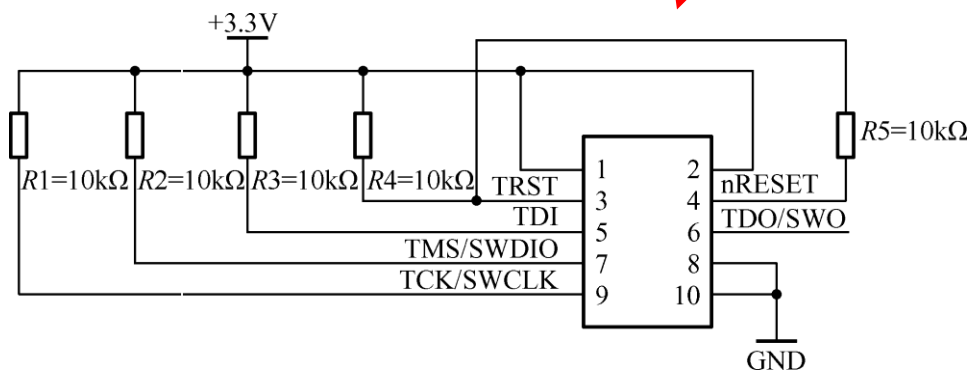


VCC	1	□	□	2	VCC
TRST	3	□	□	4	GND
TDI	5	□	□	6	GND
TMS	7	□	□	8	GND
TCLK	9	□	□	10	GND
RTCK	11	□	□	12	GND
TDO	13	□	□	14	GND
RESET	15	□	□	16	GND
N/C	17	□	□	18	GND
N/C	19	□	□	20	GND

JTAG

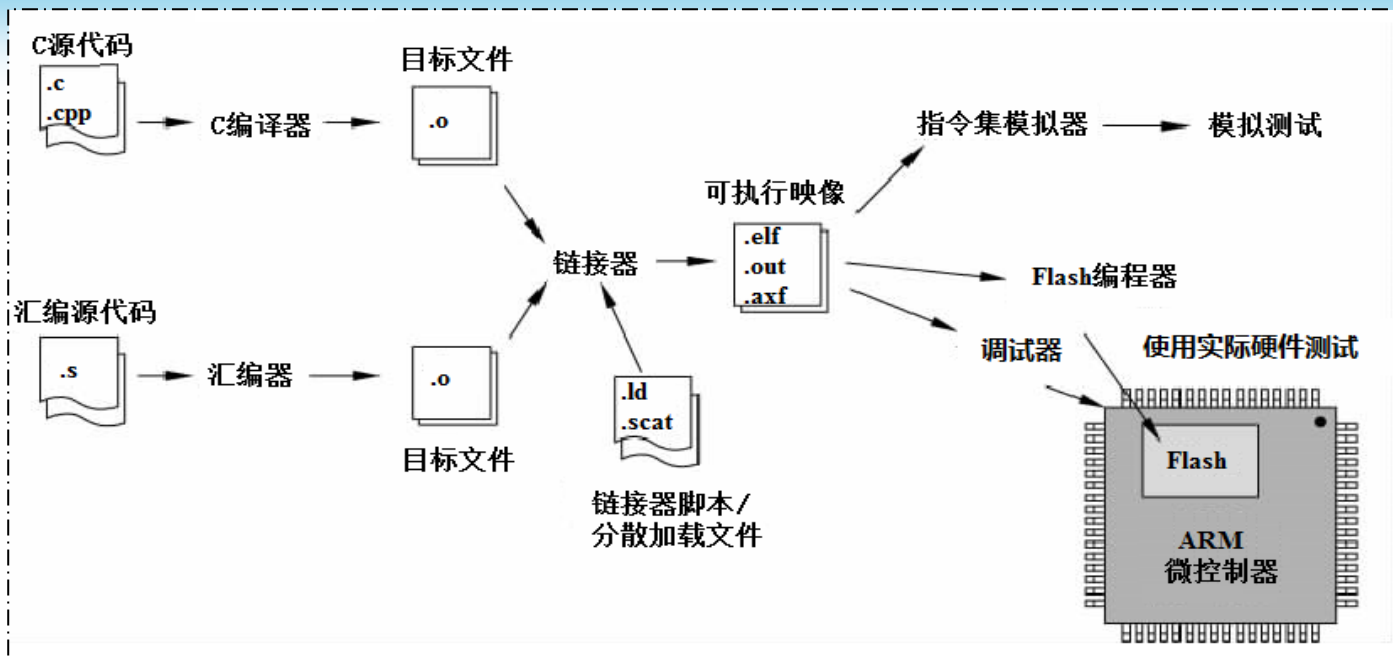
VCC	1	□	□	2	VCC
N/C	3	□	□	4	GND
N/C	5	□	□	6	GND
SWDIO	7	□	□	8	GND
SWCLK	9	□	□	10	GND
N/C	11	□	□	12	GND
SWO	13	□	□	14	GND
RESET	15	□	□	16	GND
N/C	17	□	□	18	GND
N/C	19	□	□	20	GND

SWD

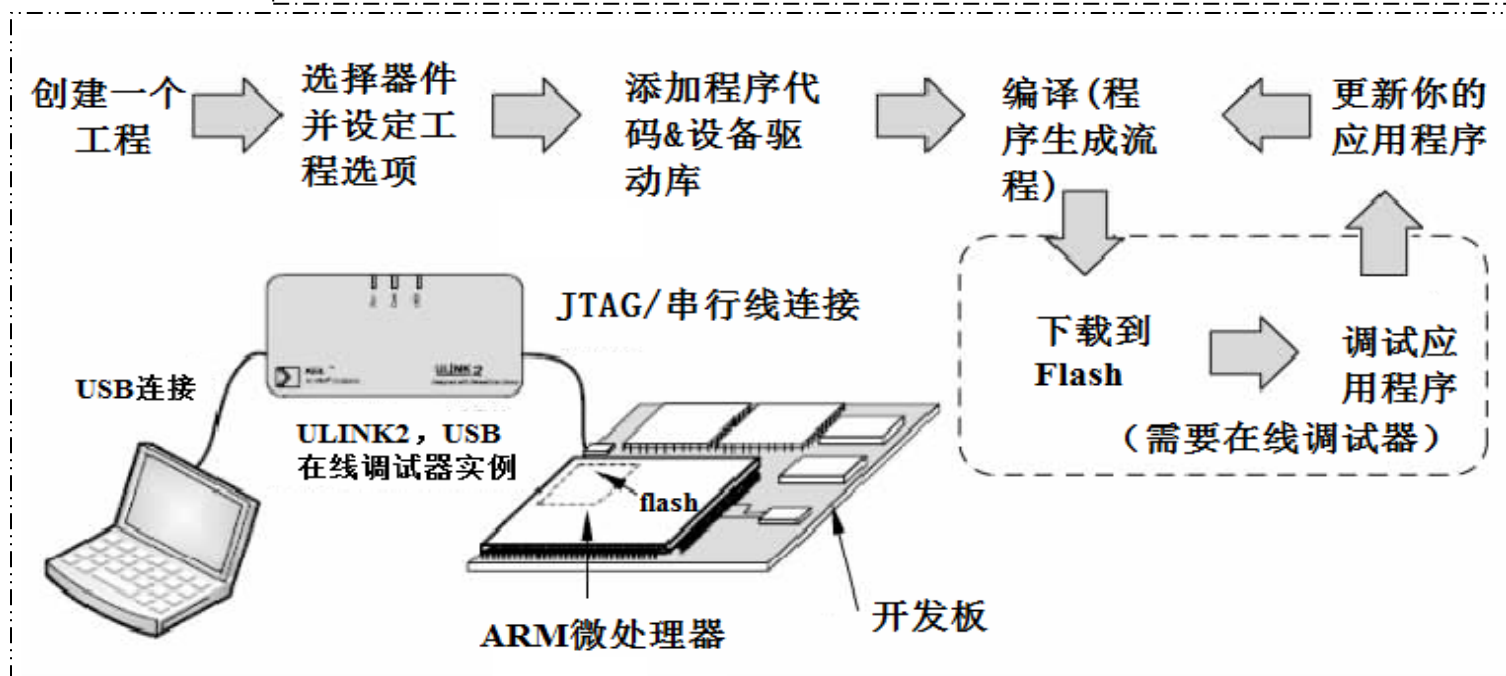


开发流程

1、程序代码的生成流程

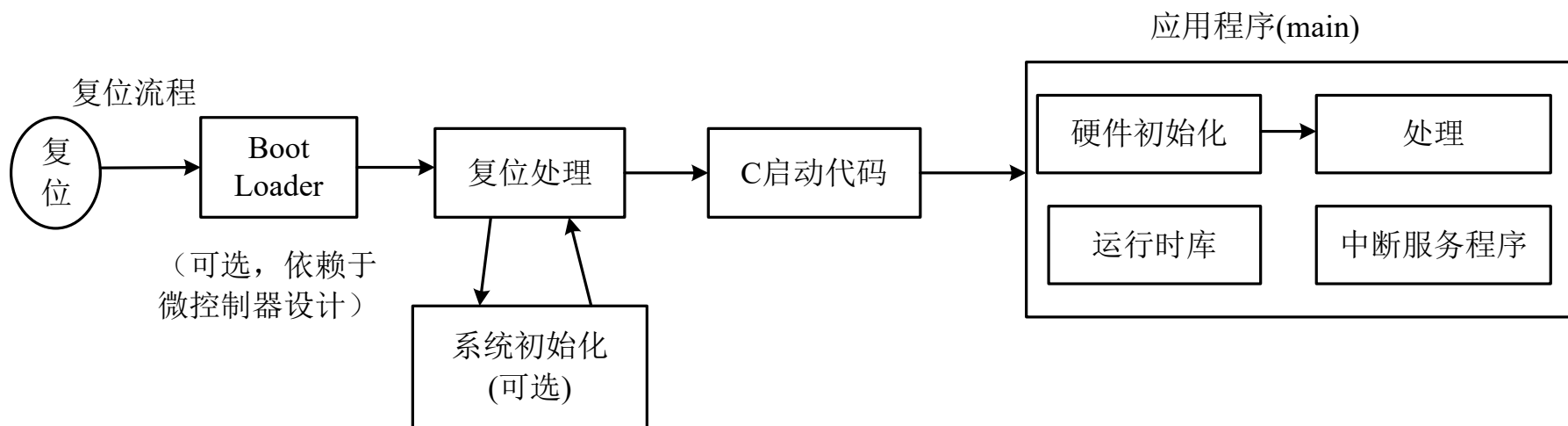


2、开发过程





复位以后，处理器将会运行复位流程。

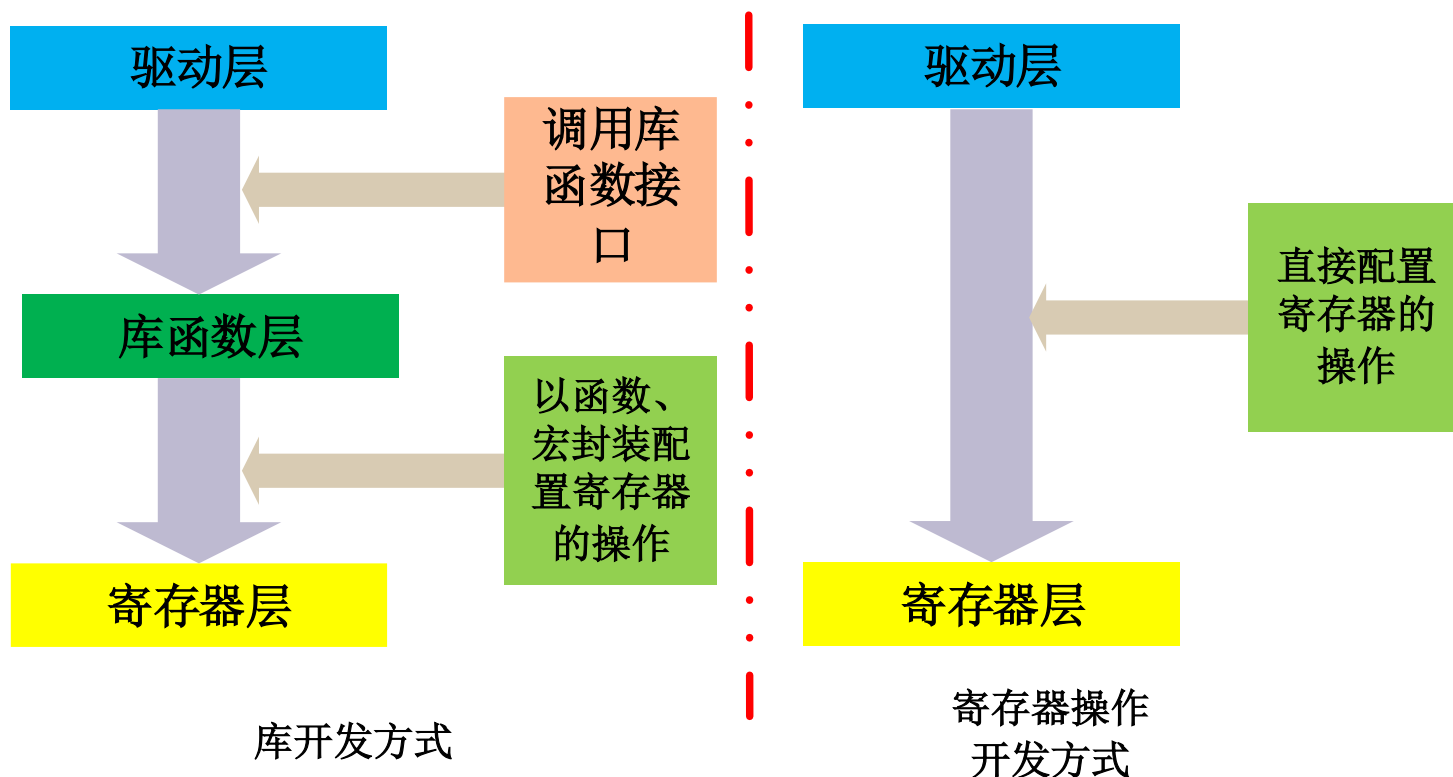


在复位流程中，处理器会取出MSP的初始值和复位向量，然后开始执行复位处理，这些所需信息都放在一个叫做**启动代码的程序文件中**。启动代码中的复位处理可能还会履行系统初始化的职责（例如，时钟控制电路和锁相环PLL的初始化），有些情况下，系统初始化是在C程序的main()函数中开始的。例如，如果在开发中使用Keil微控制器开发套件（MDK），工程创建向导可以将所选芯片对应的**默认启动代码**复制到工程中。



软件开发形式_C语言开发ARM应用

STM32库是由ST公司针对STM32提供的函数接口，即API (Application Program Interface)，开发者可调用这些函数接口来配置STM32的寄存器，使开发人员得以脱离最底层的寄存器操作，有开发快速，易于阅读，维护成本低等优点。





STM32标准函数库介绍

库是架设在寄存器与用户驱动层之间的代码，向下处理与寄存器直接相关的配置，向上为用户提供配置寄存器的接口。库开发方式与直接配置寄存器方式的区别。

特点	库开发方式	特点	直接寄存器操作开发方式
更接程序员的思维	1) 用结构体封装寄存器参数	更接近机械思维	直接针对寄存器的某些为进行置1或清0操作，能清晰看到驱动代码控制的底层对象。
	2) 用宏表示参数，意义明确		
	3) 用函数封装对寄存器的操作		
移植性好	代码的易读性好，使得驱动修改非常方便	运行效率	没有库函数层，省去代码为分层而消耗的资源

STM32的内核是ARM公司设计的处理器体系架构。ST公司或其它芯片生产厂商，负责设计的是在内核之外的部件，被称为核外外设或片上外设、设备外设。如芯片内部的模数转换外设ADC、串口UART、定时器TIM等。

为了解决不同的芯片厂商生产的Cortex微控制器软件的兼容性问题，ARM与芯片厂商建立了CMSIS标准(Cortex MicroController Software Interface Standard)。



Cortex微控制器软件接口标准(CMSIS)

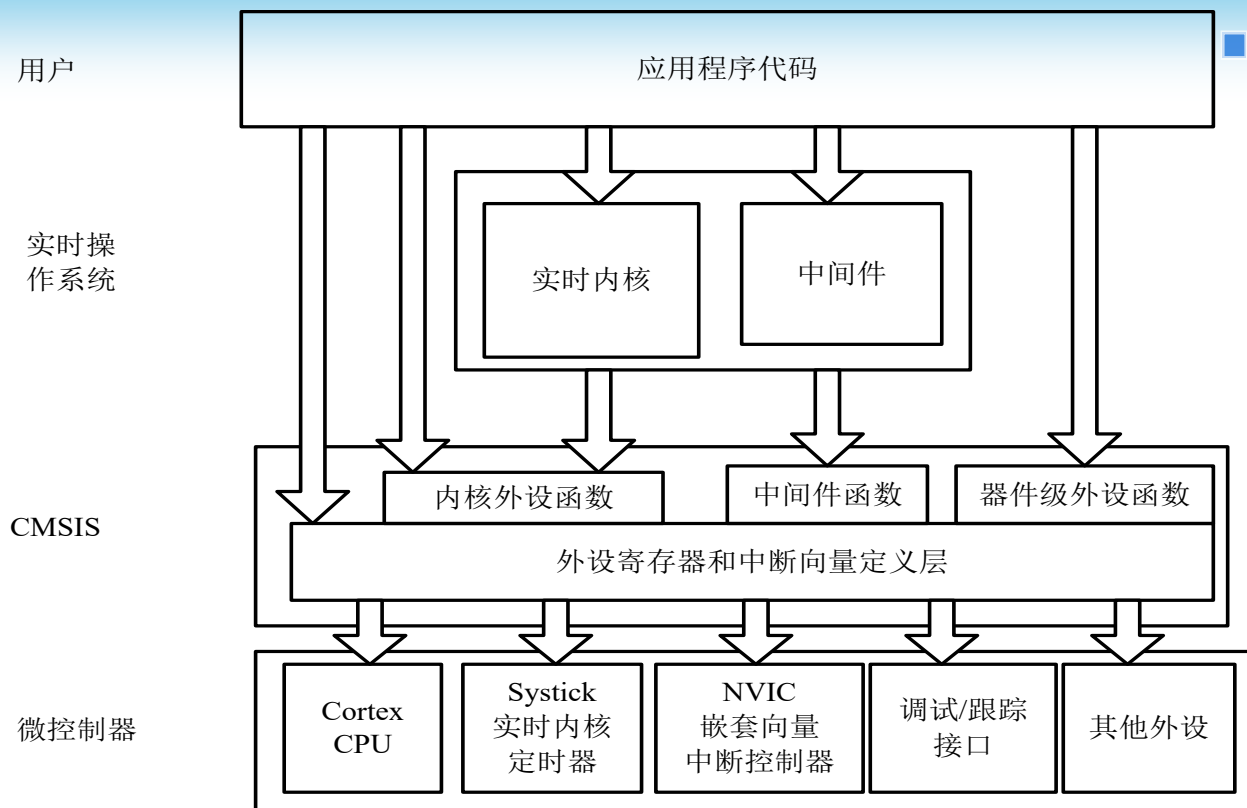
为了使软件产品具有高度的兼容性和可移植性，ARM公司与许多微控制器供应商和软件方案供应商共同努力，开发了一个通用的软件框架CMSIS，该框架适用于大多数的Cortex-M处理器以及Cortex-M微控制器产品。CMSIS针对处理器特性提供了标准化的操作函数。

CMSIS为嵌入式软件提供了以下标准化的内容：

- 标准化的操作函数，用于访问NVIC、系统控制块（SCB），SysTick的中断控制和初始化。
- NVIC、SCB和SysTick寄存器的标准化定义。
- 使用Cortex-M微控制器特殊指令的标准化函数。
- 系统异常处理的标准化命名。
- 系统初始化函数的标准化命名。通用的系统初始化函数被命名为void SystemInit(void)。
- 为时钟频率信息建立标准化的变量。这个变量为SystemCoreClock。
- 设备驱动库的通用平台。

在将来的CMSIS的发行版中可能会纳入一套通用的通信访问函数。

CMSIS的组织结构



CMSIS可以分为以下3层:

- ①**核心外设访问层:** ARM 公司提供的访问，定义处理器内部寄存器地址以及功能函数。
- ②**中间件访问层:** 定义访问中间件的通用API，由ARM公司提供
- ③**设备外设访问层(MCU相关):** 定义硬件寄存器的地址以及外设的访问函数。



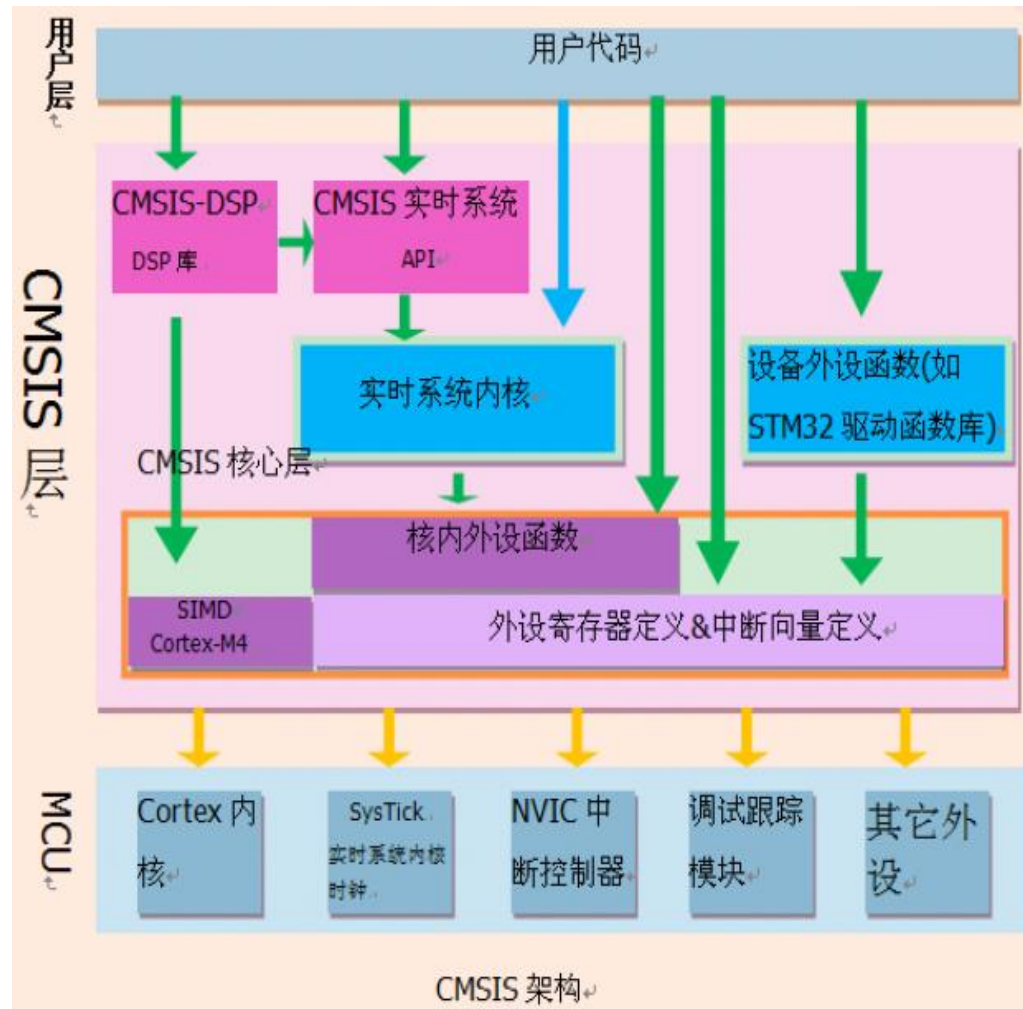
CMSIS标准，实际是新建了一个软件抽象层

CMSIS标准中最主要的是CMSIS核心层，它包括了：

1、内核函数层：其中包含用于访问内核寄存器的名称、地址定义，主要由**ARM**公司提供。

2、设备外设访问层：提供了片上的核外外设的地址和中断定义，主要由芯片生产商提供。

CMSIS层位于硬件层与操作系统或用户层之间，提供了与芯片生产商无关的**硬件抽象层**，可以为接口外设、实时操作系统提供简单的处理器软件接口，屏蔽了硬件差异，这对软件的移植是有极大的好处的。



STM32的库，就是按照CMSIS标准建立的

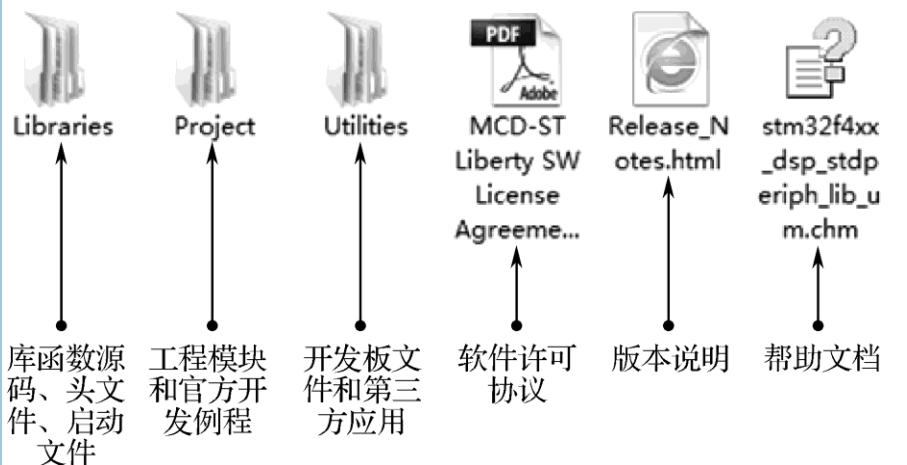


所谓 **CMSIS**，实际是新建了一个软件抽象层。它位于硬件层与操作系统或用户层之间，提供了与芯片生产商无关的硬件抽象层，可以为接口外设、实时操作系统提供简单的处理器软件接口，屏蔽了硬件差异，这对软件的移植是有极大的好处的。**STM32** 的库就是按照**CMSIS**建立的。

数据类型

C 和 C99 数据类型	位数	范围(有符号)	范围(无符号)
char, int8_t, uint8_t	8	-128~127	0~255
short, int16_t, uint16_t	16	-32 768~32 767	0~65 535
int, int32_t, uint32_t	32	-2 147 483 648~2 147 483 647	0~4 294 967 265
long	32	-2 147 483 648~2 147 483 647	0~4 294 967 265
long long, int64_t, uint64_t	64	$-2^{63} \sim 2^{63} - 1$	$0 \sim 2^{64} - 1$
float	32	$-3.402\,823\,4 \times 10^{38} \sim 3.402\,823\,4 \times 10^{38}$	
double	64	$-1.797\,693\,134\,862\,315\,7 \times 10^{308} \sim$ $1.797\,693\,134\,862\,315\,7 \times 10^{308}$	
long double	64	$-1.797\,693\,134\,862\,315\,7 \times 10^{308} \sim$ $1.797\,693\,134\,862\,315\,7 \times 10^{308}$	
pointers	32	0x0~0xFFFFFFFF	
enum	8/16/32	可能的最小数据类型	
bool(C++), _Bool(C)	8	真或假	
wchar_t	16	0~65 535	

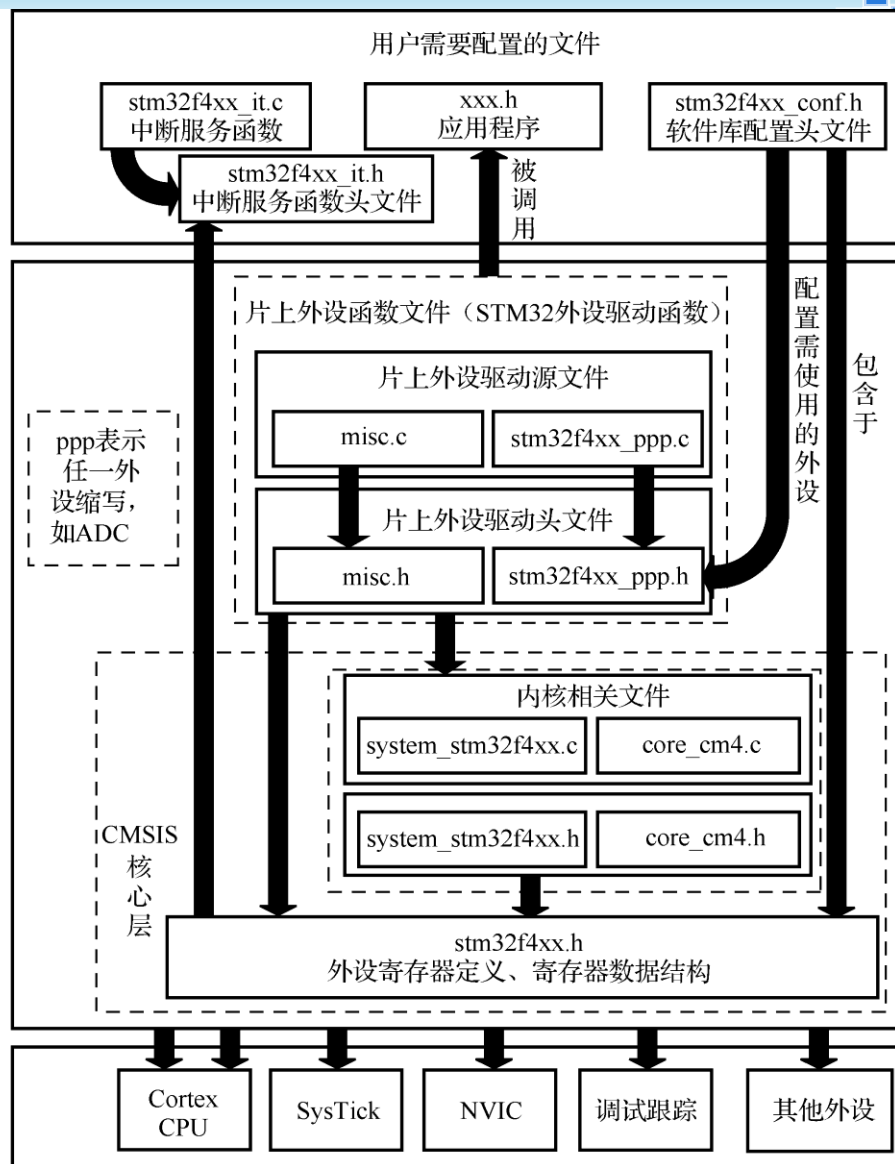
STM32标准函数库介绍



CMSIS层

在libraries/drivers文件夹下有include和source两个文件夹，这两个文件夹都属于cmsis的设备外设函数部分，在source和include文件夹里就是st公司针对每个stm32外设编写的库函数文件，每个外设对应一个.c和.h后缀的文件

硬件层



<http://www.stmcu.org/document/list/index/category-524>



STM32F4xx标准函数库文件分析

启动文件	startup_stm32f429_439xx.s	启动文件	1必须
外设相关	stm32f4xx.h	外设寄存器定义	
	system_stm32f4xx.h	用于系统初始化	
	system_stm32f4xx.c	用于配置系统时钟	2必须
	stm32f4xx_xx.h	外设标准函数库头文件	
	stm32f4xx_xx.c	外设标准函数库源件	3必须，与外设相关
	misc.h		
	misc.c	NVIC、SysTick相关函数	4必须
内核相关	core_cm4.h	内核寄存器定义	
	core_cmFunc.h	操作内核相关，不常用	
	core_cmInstr.h	定义了内核指令	
	core_cmSimd.h	定义SIMD指令	
用户相关	stm32f4xx_it.h		
	stm32f4xx_it.c	用户编写的中断服务函数	5
	main.c	用户应用程序主程序入口	6必须
	其他应用子程序	用户自定义应用功能	7



标准函数库使用

需要使用哪一个片上外设即将库中，其对应的源文件添加到工程中。

同时，在stm32f4xx_conf.h包含对应头文件。

源文件 stm32f4xx_XX.c

头文件

- misc.c
- stm32f4xx_adc.c
- stm32f4xx_can.c
- stm32f4xx_cec.c
- stm32f4xx_crc.c
- stm32f4xx_cryp.c
- stm32f4xx_cryp_aes.c
- stm32f4xx_cryp_des.c
- stm32f4xx_cryp_tdes.c
- stm32f4xx_dac.c
- stm32f4xx_dbgmcu.c
- stm32f4xx_dcmi.c
- stm32f4xx_ddsdr.c
- stm32f4xx_dma.c
- stm32f4xx_dma2d.c
- stm32f4xx_dsi.c
- stm32f4xx_exti.c
- stm32f4xx_flash.c
- stm32f4xx_flash_ramfunc.c
- stm32f4xx_fsmc.c
- stm32f4xx_fmpi2c.c
- stm32f4xx_fsmc.c
- stm32f4xx_gpio.c
- stm32f4xx_hash.c
- stm32f4xx_hash_md5.c

- misc.h
- stm32f4xx_adc.h
- stm32f4xx_can.h
- stm32f4xx_cec.h
- stm32f4xx_crc.h
- stm32f4xx_cryp.h
- stm32f4xx_dac.h
- stm32f4xx_dbgmcu.h
- stm32f4xx_dcmi.h
- stm32f4xx_ddsdr.h
- stm32f4xx_dma.h
- stm32f4xx_dma2d.h
- stm32f4xx_dsi.h
- stm32f4xx_exti.h
- stm32f4xx_flash.h
- stm32f4xx_flash_ramfunc.h
- stm32f4xx_fsmc.h
- stm32f4xx_fmpi2c.h
- stm32f4xx_fsmc.h
- stm32f4xx_gpio.h
- stm32f4xx_hash.h
- stm32f4xx_i2c.h
- stm32f4xx_iwdg.h
- stm32f4xx_lptim.h
- stm32f4xx_ltdc.h

GPIO相关的操作函数



2025/7/1



用来编写中断服务程序的一个文件，已经提供了相关片上外设的中断服务程序的框架，用户只需要填写响应中断的程序。

```
/*
 * STM32F4xx Peripherals Interrupt :
 * Add here the Interrupt Handler for the used peripheral
 * available peripheral interrupt handler's name please
 * file (startup_stm32f429_439xx.s).
 */
/**
 * @}
 */
void EXTI2_IRQHandler(void)
{
    //确保是否产生了EXTI Line中断
    if(EXTI_GetITStatus(KEY1_INT_EXTI_LINE) != RESET)
    {
        // LED 取反
        LED_TOGGLE;
        //清除中断标志位
        EXTI_ClearITPendingBit(KEY1_INT_EXTI_LINE);
    }
}

EXPORT WWDG_IRQHandler [WEAK]
EXPORT PVD_IRQHandler [WEAK]
EXPORT TAMP_STAMP_IRQHandler [WEAK]
EXPORT RTC_WKUP_IRQHandler [WEAK]
EXPORT FLASH_IRQHandler [WEAK]
EXPORT RCC_IRQHandler [WEAK]
EXPORT EXTI0_IRQHandler [WEAK]
EXPORT EXTI1_IRQHandler [WEAK]
EXPORT EXTI2_IRQHandler [WEAK]
EXPORT EXTI3_IRQHandler [WEAK]
EXPORT EXTI4_IRQHandler [WEAK]
EXPORT DMA1_Stream0_IRQHandler [WEAK]
EXPORT DMA1_Stream1_IRQHandler [WEAK]
EXPORT DMA1_Stream2_IRQHandler [WEAK]
EXPORT DMA1_Stream3_IRQHandler [WEAK]
EXPORT DMA1_Stream4_IRQHandler [WEAK]
EXPORT DMA1_Stream5_IRQHandler [WEAK]
EXPORT DMA1_Stream6_IRQHandler [WEAK]
EXPORT ADC_IRQHandler [WEAK]
EXPORT CAN1_TX_IRQHandler [WEAK]
EXPORT CAN1_RX0_IRQHandler [WEAK]
EXPORT CAN1_RX1_IRQHandler [WEAK]
EXPORT CAN1_SCE_IRQHandler [WEAK]
EXPORT EXTI9_5_IRQHandler [WEAK]
EXPORT TIM1_BRK_TIM9_IRQHandler [WEAK]
```

中断服务程序名字与启动文件中定义的要保持一致！



Keil MDK工程中保护的文件

标准函数库使用

1、**startup_stm32f429_439xx.s**

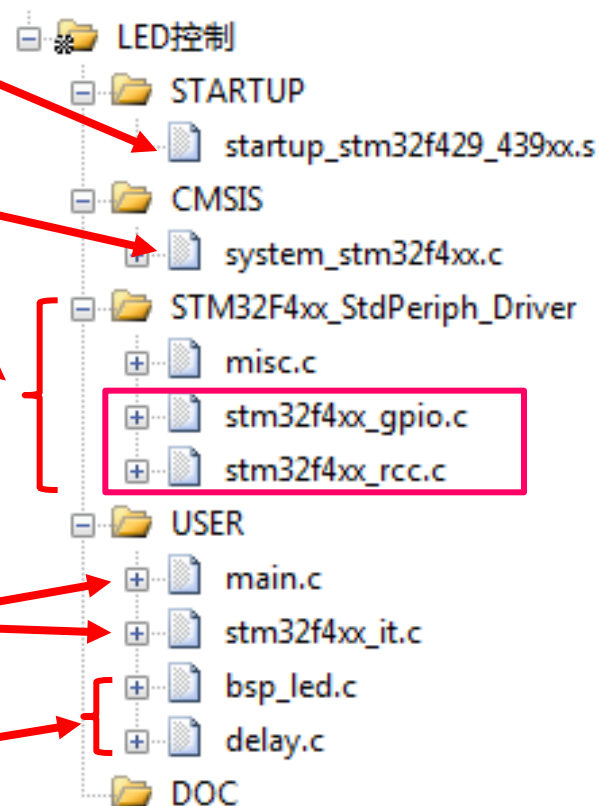
2、**system_stm32f4xx.c**

3、**stm32f4xx_xx.c** : LED控制需要使用到GPIO, 并用到了系统时钟定时生成的延时功能, 因此在工程中需要包含需要包含stm32f4xx_gpio.c和stm32f4xx_rcc.c

4、**stm32f4xx_it.c**

5、**main.c**

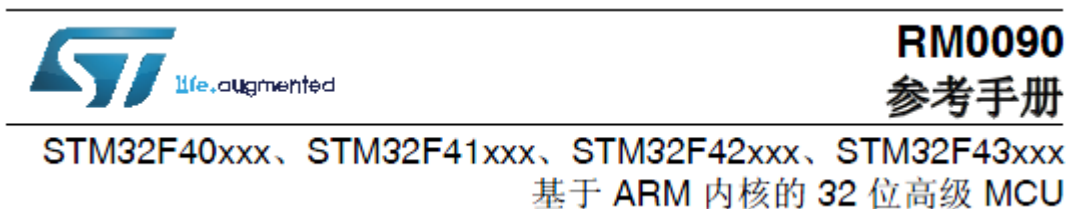
6、其他应用程序



2025/7/1



1、STM32F4XX有完整的中文编程参考手册，说明了这一系列芯片的所有功能和使用方法，一般查阅的是这一个手册。



- 1 文档约定
- 2 存储器和总线架构
- 3 嵌入式 Flash 接口
- 4 CRC 计算单元
- 5 电源控制器 (PWR)
- 6 复位和时钟控制 (RCC)
- 7 通用 I/O (GPIO)
- 8 系统配置控制器 (SYSCFG)
- 9 DMA 控制器 (DMA)
- 10 中断和事件
- 11 模数转换器 (ADC)
- 12 数模转换器 (DAC)
- 13 数字摄像头接口 (DCMI)
- 14 高级控制定时器 (TIM1 和 TIM8)
- 15 通用定时器 (TIM2 到 TIM5)
- 16 通用定时器 (TIM9 到 TIM14)
- 17 基本定时器 (TIM6 和 TIM7)
- 18 独立看门狗 (IWDG)
- 19 窗口看门狗 (WWDG)
- 20 加密处理器 (CRYP)
- 21 随机数发生器 (RNG)
- 22 散列处理器 (HASH)
- 23 实时时钟 (RTC)
- 24 控制器区域网络 (bxCAN)
- 25 内部集成电路 (I2C) 接口
- 26 通用同步异步收发器 (USART)

2、对应具体芯片的数据手册一般是英文的，一般说明具体芯片的基本描述、电气特性等。



STM32F4系列微控制器存储器映射和寄存器

- 1、存储器内部各个块的功能由芯片厂商，给存储器分配地址的过程就称为存储器映射。
- 2、STM32F4XX在存储空间上采用冯诺依曼结构，在寻址上采用哈佛结构。可寻址的存储空间为 $2^{32}=4\text{G Byte}$ ，共被分为8个块。每个存储单元都有一个地址，存储单位为字节。
- 3、编写应用程序一般固化在Flash中，程序运行过程中使用的变量主要杯定义在SRAM中（Block1），控制片上外设使用到的寄存器被配备在Block2上，当需要外部扩展存储空间时使用Block3、4。

序号	用途	地址范围
Block 0	SRAM（FLASH）	0x0000 0000 ~ 0x1FFF FFFF(512MB)
Block 1	SRAM	0x2000 0000 ~ 0x3FFF FFFF(512MB)
Block 2	片上外设	0x4000 0000 ~ 0x5FFF FFFF(512MB)
Block 3	FMC的bank1 ~ bank2	0x6000 0000 ~ 0x7FFF FFFF(512MB)
Block 4	FMC的bank3 ~ bank4	0x8000 0000 ~ 0x9FFF FFFF(512MB)
Block 5	FMC	0xA000 0000 ~ 0xCFFF FFFF(512MB)
Block 6	FMC	0xD000 0000 ~ 0xDFFF FFFF(512MB)
Block 7	Cortex-M4内部外设	0xE000 0000 ~ 0xFFFF FFFF(512MB)



存储器映射

0x2003 0000~0x3FFF FFFF
0x2002 0000~0x2002 FFFF
0x2001 C000~0x2001 FFFF
0x2000 0000~0x2001 BFFF
0x1FFF C008~0x1FFF FFFF
0x1FFF C000~0x1FFF C00F
0x1FFF 7A10~0x1FFF 7FFF
0x1FFF 0000~0x1FFF 7A0F
0x1FFE C008~0x1FFE FFFF
0x1FFE C000~0x1FFE C00F
0x1001 0000~0x1FFE BFFF
0x1000 0000~0x1000 FFFF
0x0820 0000~0x0FFF FFFF
0x0800 0000~0x081F FFFF
0x0020 0000~0x07FF FFFF
0x0000 0000~0x001F FFFF

保留
64KB SRAM3
16KB SRAM2
112KB SRAM1
保留
选项字节
保留
系统存储器
保留
选项字节
保留
64K CCM SRAM
保留
Flash
保留
根据BOOT引脚, 将 Flash、系统存储器、 SRAM映射到此处

Cortex-M4内部外设 BLOCK7	0.5GB
FMC (SDRAM区域) BLOCK6	0.5GB
保留	
FMC控制寄存器 BLOCK5	0.5GB
FMC区域3、4 BLOCK4	0.5GB
FMC区域1、2 BLOCK3	0.5GB
片上外设 BLOCK2	0.5GB
SRAM BLOCK1	0.5GB
SRAM BLOCK0	0.5GB

0xFFFFFFFF
0xE0000000
0xDFFFFFFF
0xC000 0000
0xBFFFFFFF
0xA0001000
0xA000FFFF
0xA0000000
0x9FFFFFFF
0x80000000
0x7FFFFFFF
0x60000000
0x5FFFFFFF
0x40000000
0x3FFFFFFF
0x20000000
0x1FFFFFFF
0x00000000

保留
Cortex-M4内部外设
AHB3
保留
AHB2
保留
AHB1
保留
APB2
保留
APB1

0xE010 0000~0xFFFF FFFF
0xE000 0000~0xE00F FFFF
0x6000 0000~0xDFFF FFFF
0x5006 0C00~0xFFFF FFFF
0x5006 0BFF
0x5000 0000
0x4008 0000~0x4FFF FFFF
0x4007 FFFF
0x4002 0000
0x4001 6C00~0x4001 FFFF
0x4001 6BFF
0x4001 0000
0x4000 8000~0x4000 FFFF
0x4000 7FFF
0x4000 0000



寄存器映射

把片上外设对应的寄存器在存储空间上分配地址的过程就叫寄存器映射。与存储单元一样，每个寄存器（一般都是32位的）都有一个寻址地址。

寄存器是一个很重要的概念。

- 应用程序对片上外设的初始化和控制是通过对片上外设对应一系列寄存器的修改、读写来实现的。
- STM32F429的各个片上外设通过APB1、APB2总线和AHB1、AHB2总线连接到内核上。
- 每个总线都有对应的基地址，挂载在相应总线上的外设寄存器的地址，都以相应总线的基地址进行偏移。

具体可以查看STM32F4XX参考手册（RM0090） p52-p54。

各总线基地址：

总线名称	总线基地址	相对外设基地址的偏移
APB1	0x4000 0000	0x0
APB2	0x4001 0000	0x0001 0000
AHB1	0x4002 0000	0x0002 0000
AHB2	0x5000 0000	0x1000 0000
AHB3	0x6000 0000	已不属于片上外设



GPIO(通用输入输出端口)挂载在AHB1总线上

GPIO基地址:

外设名称	外设基地址	相对AHB1总线的地址偏移
GPIOA	0x4002 0000	0x0
GPIOB	0x4002 0400	0x0000 0400
GPIOC	0x4002 0800	0x0000 0800
GPIOD	0x4002 0C00	0x0000 0C00
GPIOE	0x4002 1000	0x0000 1000
GPIOF	0x4002 1400	
GPIOG	0x4002 1800	
GPIOH	0x4002 1C00	

GPIOH端口的寄存器列表

寄存器名称	寄存器地址	相对GPIOH基址的偏移
GPIOH_MODER	0x4002 1C00	0x00
GPIOH_OTYPER	0x4002 1C04	0x04
GPIOH_OSPEEDR	0x4002 1C08	0x08
GPIOH_PUPDR	0x4002 1C0C	0x0C
GPIOH_IDR	0x4002 1C10	0x10
GPIOH_ODR	0x4002 1C14	0x14
GPIOH_BSRR	0x4002 1C18	0x18
GPIOH_LCKR	0x4002 1C1C	0x1C
GPIOH_AFRH	0x4002 1C20	0x20
GPIOH_AFRH	0x4002 1C24	0x24



GPIOx端口数据输出寄存器ODR描述

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	③
Reserved																②
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	③
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0	②
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	①

位31:16 保留，必须保持复位值。

位15:0 ODRy[15:0]: 端口输出数据 (y=0~15) 这些位可通过软件读取和写入。 } ④

①是寄存器中位段的操作权限，**r**表示只读，**w**表示只写，**rw**表示可读可写。

②是位段名。

③是位段编号，从**0**开始。

④是对寄存器各位段的使用说明，通过这一部分可以得到这一个寄存器所能实现的功能。

端口有**16**个引脚，编号对应**0-15**。

数据输出寄存器的低**16**为对应于端口的每一个引脚。



例如 GPIOH端口的16个引脚输出高电平

通过绝对地址访问内存单元

```
// GPIOH 端口全部输出 高电平  
* (unsigned int*) (0x40021C14) = 0xFFFF;
```

通过宏定义方式访问内存单元

```
1 // GPIOH 端口全部输出 高电平  
2 #define GPIOH_ODR      (unsignedint*) (0x40021C14)  
3 * GPIOH_ODR = 0xFFFF;
```

为了方便操作，我们干脆把指针操作“*”也定义到寄存器别名里面

```
1 // GPIOH 端口全部输出 高电平  
2 #define GPIOH_ODR      *(unsignedint*) (0x40021C14)  
3 GPIOH_ODR = 0xFFFF;
```



1、总线和外设基址宏定义

定义在头文件
STM32F4xx.h中

```
/* 外设基址 */
#define PERIPH_BASE ((unsigned int)0x40000000)
/* 总线基址 */
#define APB1PERIPH_BASE PERIPH_BASE
#define APB2PERIPH_BASE (PERIPH_BASE + 0x00010000)
#define AHB1PERIPH_BASE (PERIPH_BASE + 0x00020000)
#define AHB2PERIPH_BASE (PERIPH_BASE + 0x10000000)
/* GPIO外设基址 */
#define GPIOA_BASE (AHB1PERIPH_BASE + 0x0000)
#define GPIOB_BASE (AHB1PERIPH_BASE + 0x0400)
#define GPIOC_BASE (AHB1PERIPH_BASE + 0x0800)
#define GPIOD_BASE (AHB1PERIPH_BASE + 0x0C00)
#define GPIOE_BASE (AHB1PERIPH_BASE + 0x1000)
#define GPIOF_BASE (AHB1PERIPH_BASE + 0x1400)
```

2、使用结构体封装寄存器

```
typedef unsigned int uint32_t; /*无符号32位变量*/
typedef unsigned short int uint16_t; /*无符号16位变量*/

/* GPIO寄存器列表 */
typedef struct {
    uint32_t MODER; /*GPIO模式寄存器 地址偏移: 0x00 */
    uint32_t OTYPER; /*GPIO输出类型寄存器 地址偏移: 0x04 */
    uint32_t OSPEEDR; /*GPIO输出速度寄存器 地址偏移: 0x08 */
    uint32_t PUPDR; /*GPIO上拉/下拉寄存器 地址偏移: 0x0C */
    uint32_t IDR; /*GPIO输入数据寄存器 地址偏移: 0x10 */
    uint32_t ODR; /*GPIO输出数据寄存器 地址偏移: 0x14 */
    uint16_t BSRRL; /*GPIO置位/复位寄存器低16位部分 地址偏移: 0x18 */
    uint16_t BSRRH; /*GPIO置位/复位寄存器高16位部分 地址偏移: 0x1A */
    uint32_t LCKR; /*GPIO配置锁定寄存器 地址偏移: 0x1C */
    uint32_t AFR[2]; /*GPIO复用功能配置寄存器 地址偏移: 0x20-0x24 */
} GPIO_TypeDef;
```



例：将GPIOH端口的10号引脚先输出低电平，再输出高电平。

```
// PH10输出输出低电平
GPIOH->ODR  &=  ~(1<<10);

// PH10输出输出高电平
GPIOH->ODR  |=  (1<<10);
```

注意：

1) 在对寄存器操作的时候一般通过读-修改-写的方式实现。

常用操作有：位与&（清零），位或|（置位），异或^（取反）；

2) 选择合适的操作和对应的屏蔽字。



基于固件库开发和直接操作寄存器的区别

STM32为了方便用户开发程序，提供了一套丰富的STM32固件库。使用固件库开发应用系统可以大大提高用户的开发效率。

固件库就是函数的集合，固件库函数的作用是向下负责直接操作寄存器，向上提供用户函数调用的接口（API）。

固件库将寄存器底层操作都装起来，提供一整套API供开发者调用。大多数场合下，用户不需要知道操作的是哪个寄存器，只需要知道调用哪些函数即可。通过使用固件函数库，无需深入掌握细节，用户也可以轻松应用每一个外设。因此，使用固态函数库可以大大减少用户的程序编写时间，进而降低开发成本。并且，所有的驱动程序源代码都符合ANSI-C标准，不受开发环境影响。



STM32固件库函数的命名规则

重庆理工大学通信工程

缩 写	外 设 名 称	缩 写	外 设 名 称
ADC	模数转换器	IWDG	独立看门狗
BKP	备份寄存器	NVIC	嵌套中断向量列表控制器
CAN	控制器局域网模块	PWR	电源/功耗控制
CRC	CRC 计算单元	RCC	复位与时钟控制器
DAC	数模转换器	RTC	实时时钟
DMA	直接内存存取控制器	SDIO	SDIO 接口
EXTI	外部中断事件控制器	SPI	串行外设接口
FLASH	闪存存储器	SysTick	系统嘀嗒定时器
FSMC	灵活的静态存储器控制器	TIM	通用定时器
GPIO	通用输入/输出端口	TIM1	高级控制定时器
I2C	I2C 总线接口	USART	通用同步异步接收发射端
I2S	I2S 总线接口	WWDG	窗口看门狗

stm32f4xx_XX.c

源文件

- misc.c
- stm32f4xx_adc.c
- stm32f4xx_can.c
- stm32f4xx_cec.c
- stm32f4xx_crc.c
- stm32f4xx_cryp.c
- stm32f4xx_cryp_aes.c
- stm32f4xx_cryp_des.c
- stm32f4xx_cryp_tdes.c
- stm32f4xx_dac.c
- stm32f4xx_dbgmcu.c
- stm32f4xx_dcmi.c
- stm32f4xx_dfldm.c
- stm32f4xx_dma.c
- stm32f4xx_dma2d.c
- stm32f4xx_dsi.c
- stm32f4xx_exti.c
- stm32f4xx_flash.c
- stm32f4xx_flash_ramfunc.c
- stm32f4xx_fmc.c
- stm32f4xx_fmpi2c.c
- stm32f4xx_fsmc.c
- stm32f4xx_gpio.c
- stm32f4xx_hash.c
- stm32f4xx_hash_md5.c

头文件

- misc.h
- stm32f4xx_adc.h
- stm32f4xx_can.h
- stm32f4xx_cec.h
- stm32f4xx_crc.h
- stm32f4xx_cryp.h
- stm32f4xx_dac.h
- stm32f4xx_dbgmcu.h
- stm32f4xx_dcmi.h
- stm32f4xx_dfldm.h
- stm32f4xx_dma.h
- stm32f4xx_dma2d.h
- stm32f4xx_dsi.h
- stm32f4xx_exti.h
- stm32f4xx_flash.h
- stm32f4xx_flash_ramfunc.h
- stm32f4xx_fmc.h
- stm32f4xx_fmpi2c.h
- stm32f4xx_fsmc.h
- stm32f4xx_gpio.h
- stm32f4xx_hash.h
- stm32f4xx_j2c.h
- stm32f4xx_iwdg.h
- stm32f4xx_lptim.h
- stm32f4xx_ltdc.h



3、通用输入输出GPIO-人机交互接口

◆ General Purpose Input / Output-通用输入输出

GPIO（通用输入/输出）是微控制器和外部进行通信的**最基本的通道**，几乎所有微控制器上都有GPIO。

GPIO的每个引脚都能够被独立配合，在微控制器片内外设功能较多而外部引脚数量有限的情况下，**GPIO引脚都具备复用功能**。

- 借助GPIO，微控制器可以实现对外围设备（如LED和按键等）最简单、最直观的监控。
- 除此之外，当微控制器没有足够的I/O引脚或片内存储器时，GPIO还可用于串行和并行通信、存储器扩展等。

STM32F系列微控制器中有多个GPIO端口，以英文字母进行编号，每个端口有16个引脚，根据芯片型号不同端口数量不同。

ARM处理器芯片的大部分引脚都可以通过设定相应的**控制寄存器**实现基本的**GPIO**功能，并可编程设置信号方向、电平上拉/下拉等功能。



GPIO管脚命名

◆ STM32微控制器最多可以提供个多功能双向I/O引脚。这些I/O引脚依次分布在不同的**端口**中。

- **端口号**：端口号通常以大写字母命名，从A开始，依次类推。例如，GPIOA、GPIOB、GPIOC、…GPIOG等。
- **引脚号**：每个端口有16个I/O引脚，分别命名为0-15。例如，STM32F411RET6微控制器的GPIOA端口有16个引脚，分别为PA0、PA1、PA2、PA3、…、PA14和PA15。

端口 (PORT)

独立的外设子模块，包括多个引脚，通过多个硬件寄存器控制引脚

引脚 (PIN)

对应微控制器的一个管脚，归属于端口，由端口寄存器的对应位控制

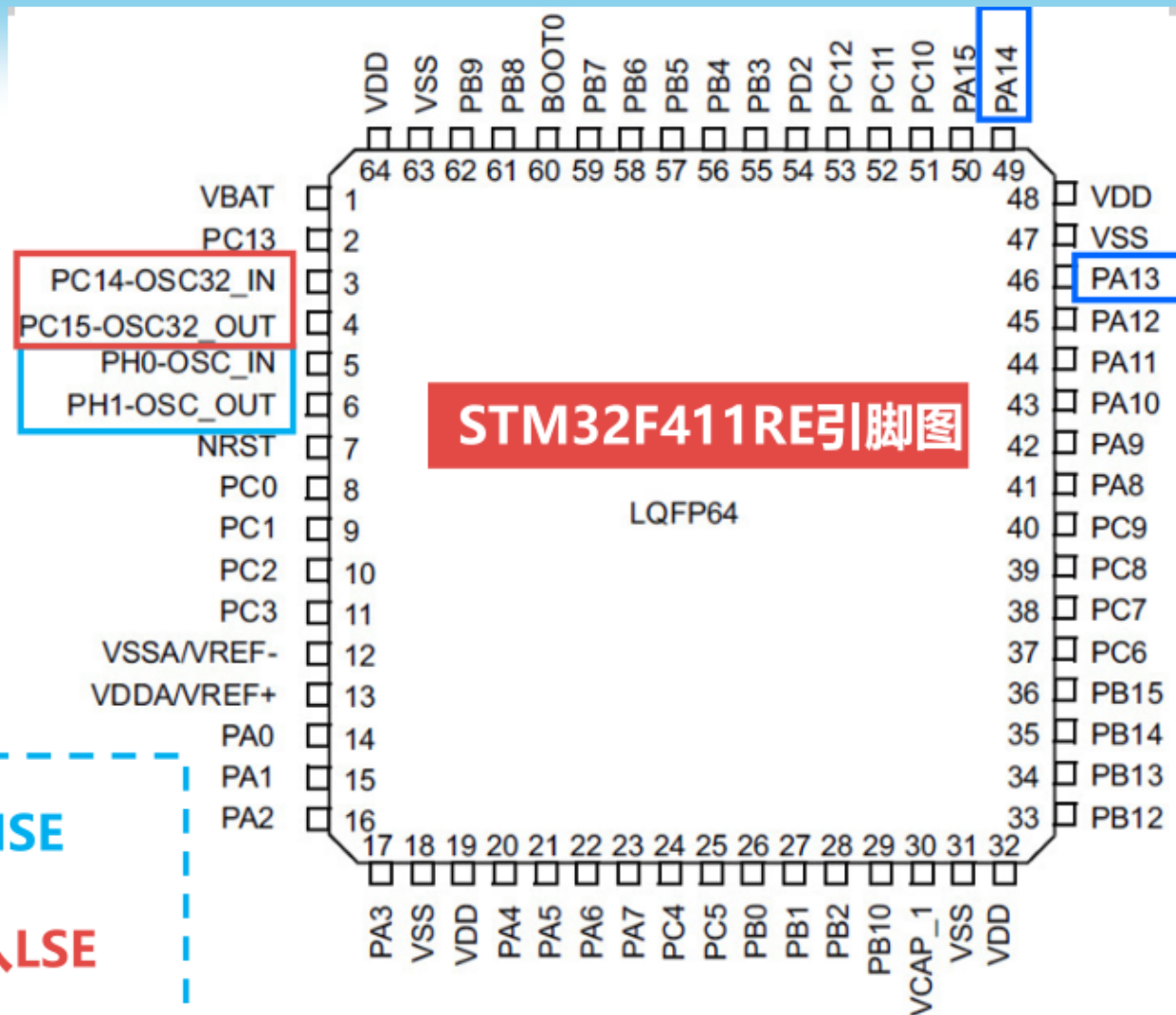


STM32F411RET6的端口和引脚

5组端口，50个I/O引脚

端口	引脚	数量	备注
GPIOA	PA0 ~ PA15	16	—
GPIOB	PB0 ~ PB15	15	PB11引脚被VCAP_1引脚取代
GPIOC	PC0 ~ PC15	16	—
GIOD	PD2	1	—
GPIOH	PH0、PH1	2	—

总结：一个端口默认包含16个引脚，但是不同型号的STM32微控制器所包含的端口数量及各端口包含的引脚数量各不相同，具体信息可以查询芯片的数据手册

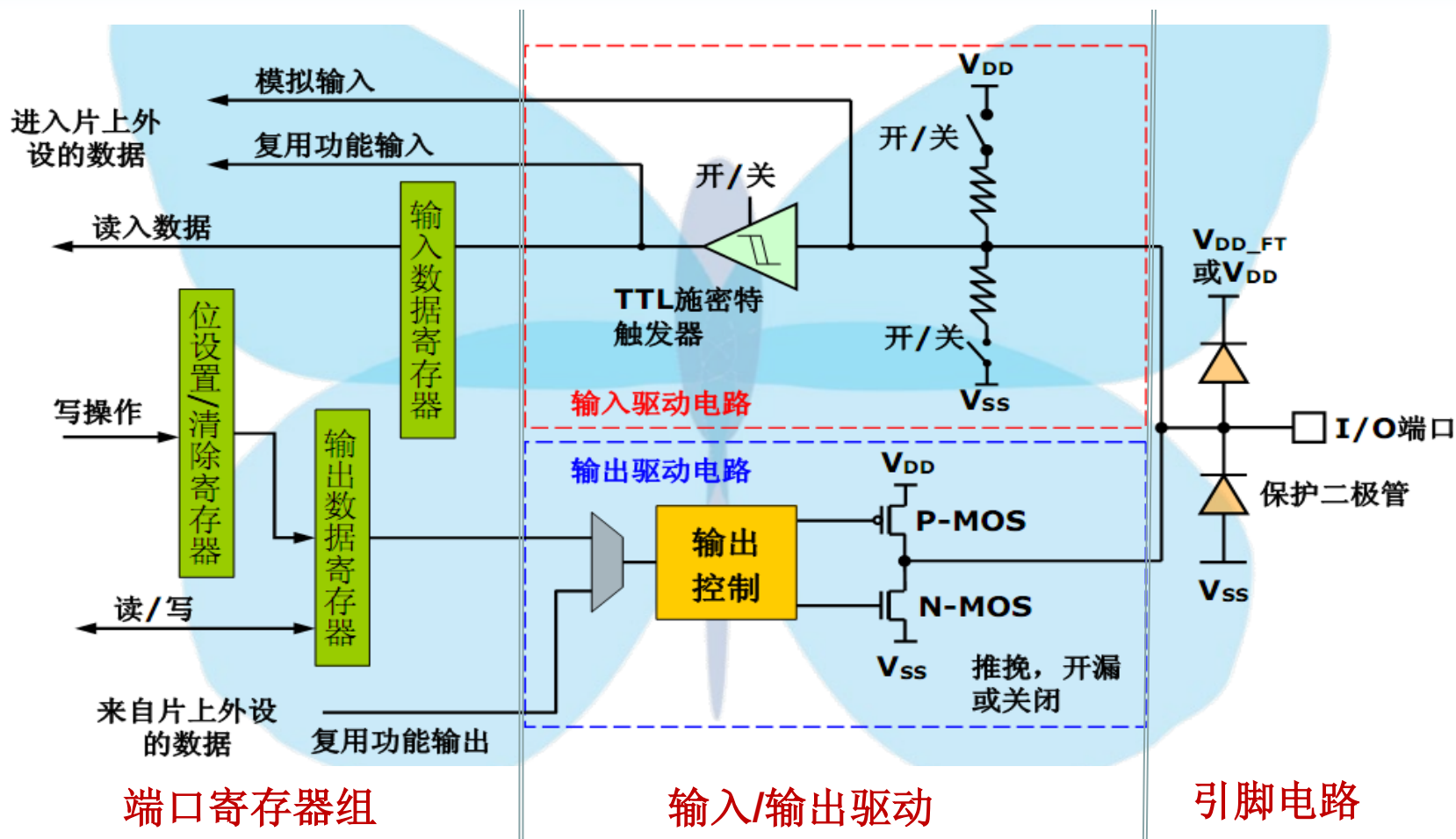


- PH0和PH1接入HSE
- PC14和PC15接入LSE
- PA13和PA14作为SWD接口

用户能使用的I/O引脚为44 (50-6)



GPIO内部结构



包括引脚电路（ESD静电保护二极管）、输入输出驱动（控制电路与驱动器）和端口寄存器组（输入输出寄存器）



GPIO的功能

- 普通I/O功能

复位期间和刚复位后，复用功能未开启，I/O端口被配置成浮空输入模式。输入数据寄存器（GPIOx_IDR）在每个APB2时钟周期捕捉I/O引脚上的数据。

当作为输出配置时，写到输出数据寄存器（GPIOx_ODR）上的值输出到相应的I/O引脚。可以以推挽模式或开漏模式（当输出0时，只有N-MOS被打开）使用输出驱动器。

所有GPIO引脚有一个内部弱上拉和弱下拉，当配置为输入时，它们可以被激活也可以被断开。

- 单独的位设置或位清除

当对GPIOx_ODR的个别位编程时，软件不需要禁止中断：在单次AHB1写操作里，可以只更改一个或多个位。

- 外部中断/唤醒线

所有端口都有外部中断能力。为了使用外部中断线，端口必须配置成输入模式。



- 复用功能 (AF)

使用默认复用功能前必须对端口位配置寄存器编程。

对于复用输入功能，端口必须配置成输入模式（浮空、上拉或下拉）且输入引脚必须由外部驱动。

对于复用输出功能，端口必须配置成复用功能输出模式（推挽或开漏）。

对于双向复用功能，端口位必须配置复用功能输出模式（推挽或开漏）。此时，输入驱动器被配置成浮空输入模式。

- 软件重新映射I/O复用功能

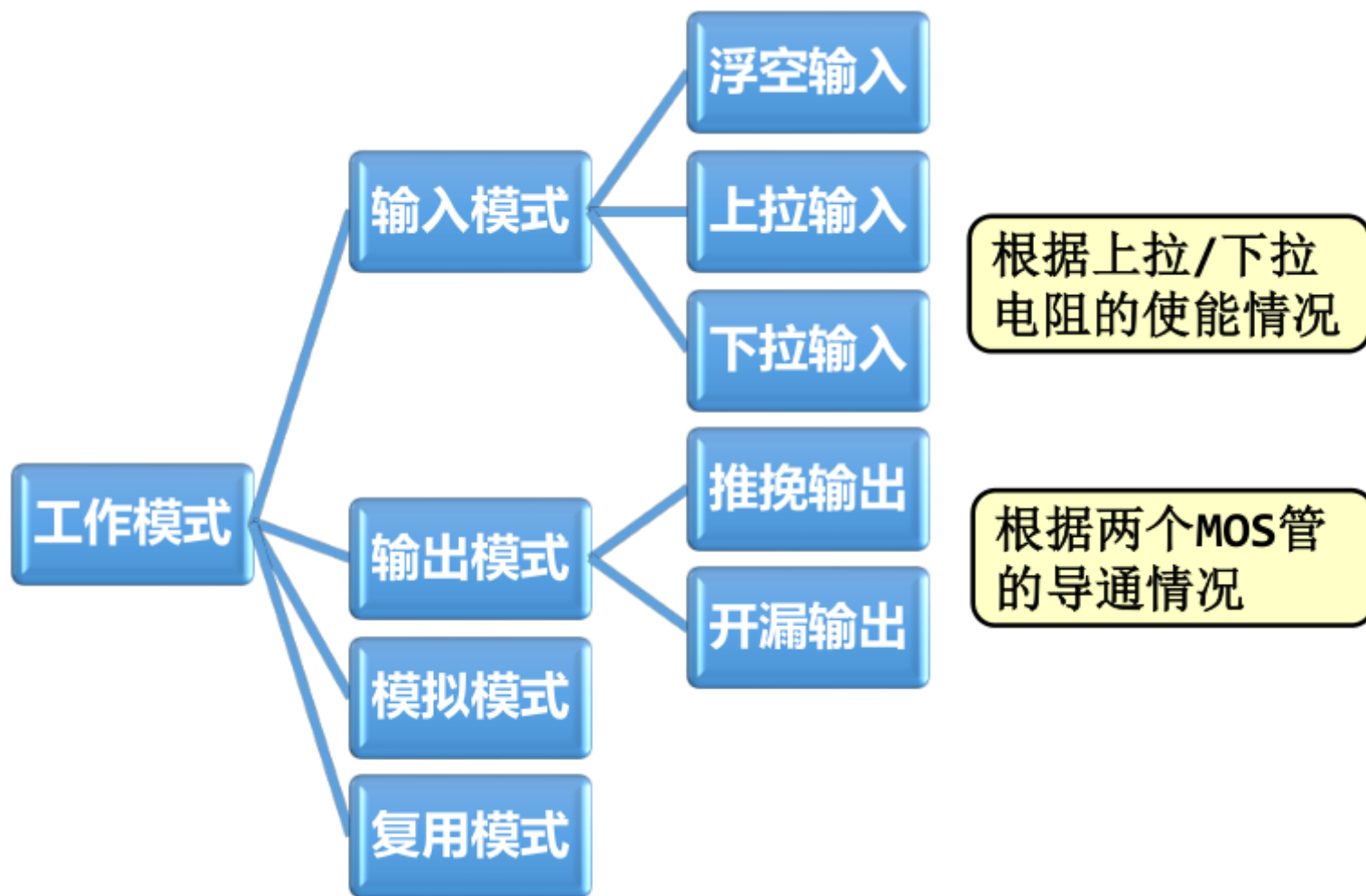
为了使不同封装器件的外设I/O功能的数量达到最优，可以把一些复用功能重新映射到其他一些引脚上。

- GPIO锁定机制

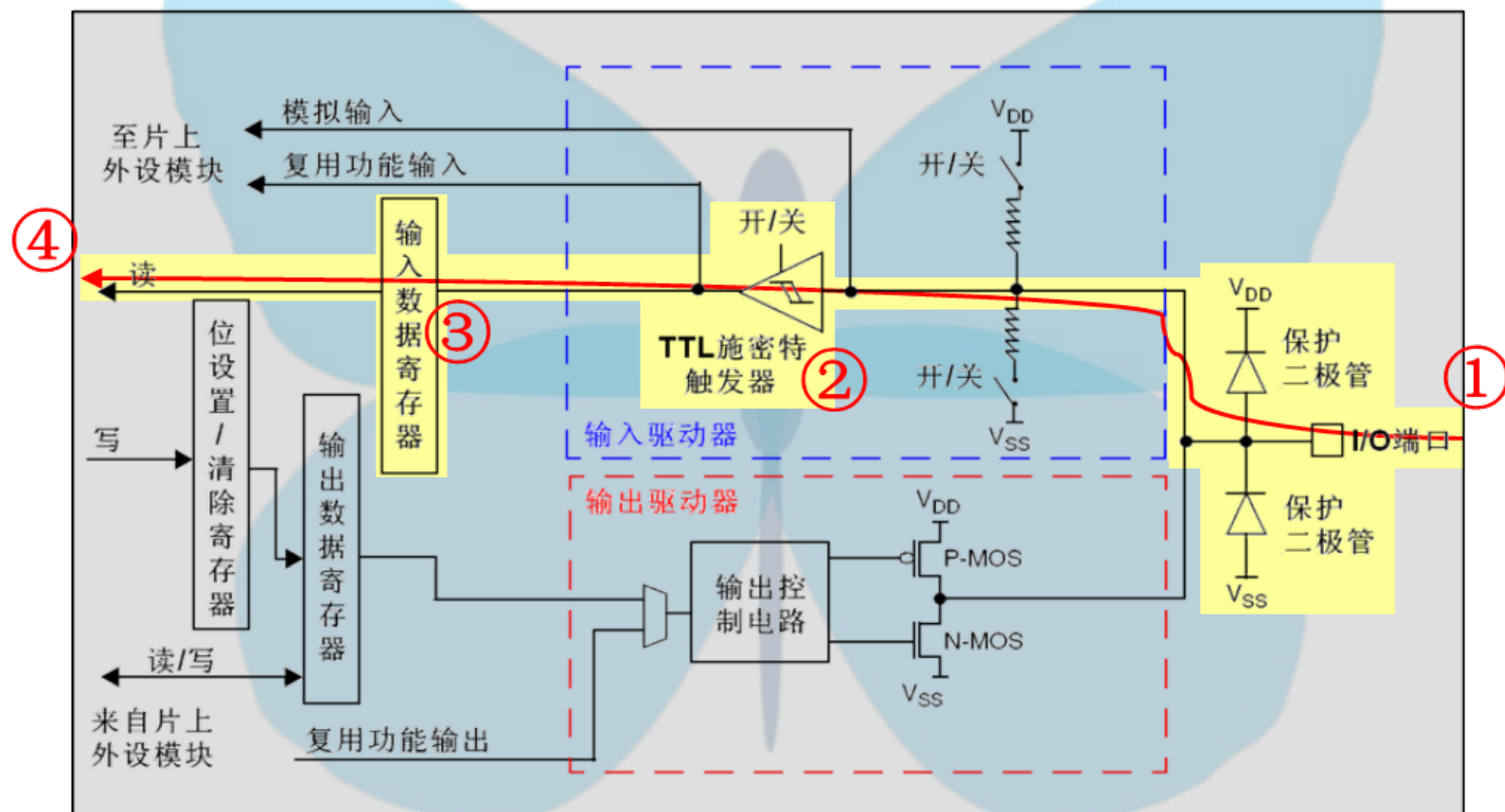
锁定机制允许冻结IO配置。



GPIO工作模式



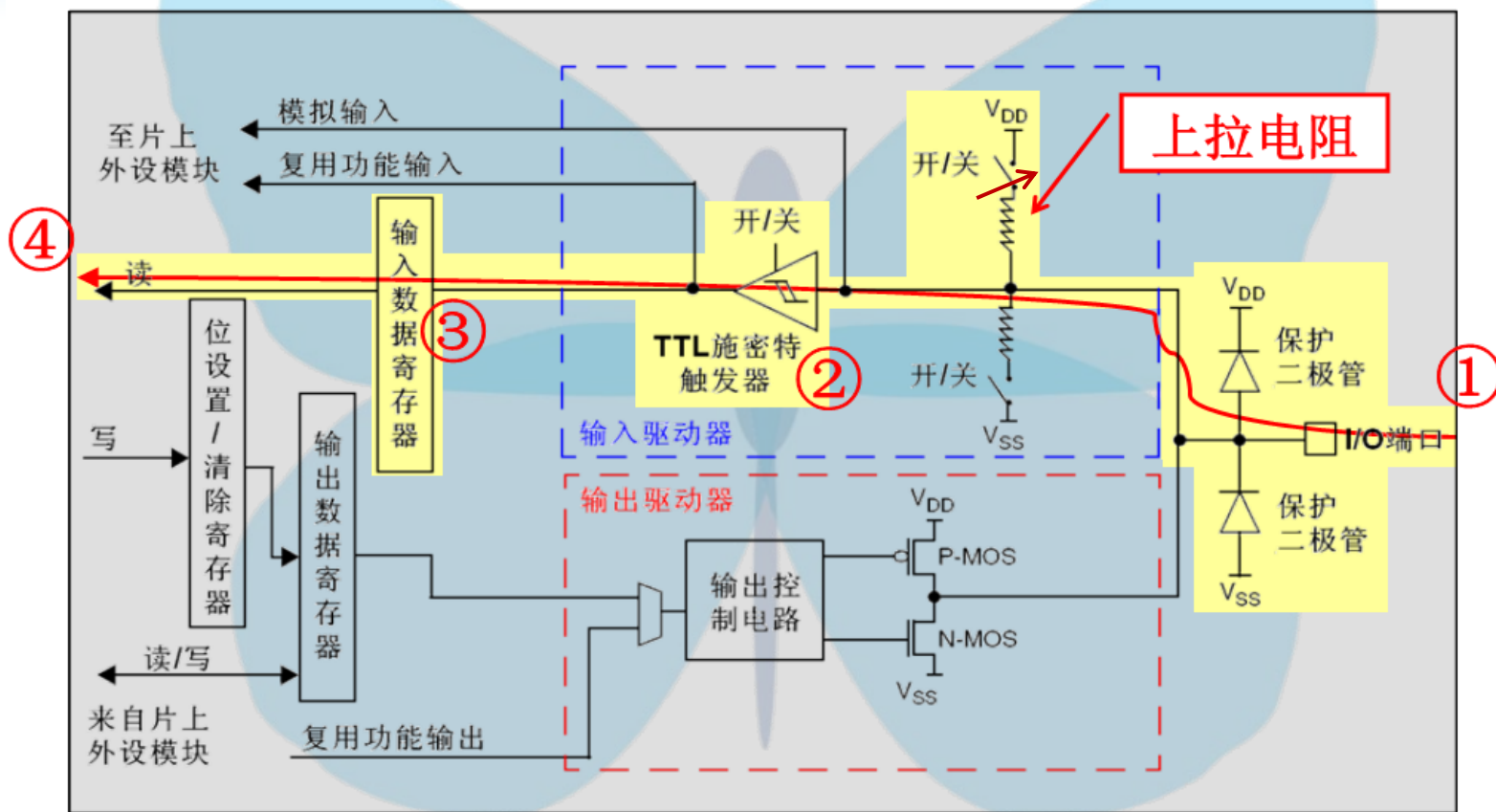
GPIO的工作模式—输入浮空模式



浮空输入模式，图中阴影的部分处于不工作状态，尤其是下半部分的输出驱动电路。黄色的高亮部分显示了数据传输通道，外部的电平信号通过左边编号①的I/O端口进入STM32，经过编号②的施密特触发器的整形送入编号③的“输入数据寄存器”，在“输入数据寄存器”的另一端(编号4)，CPU可以随时读出I/O端口的电平状态。

浮空输入模式的特点：输入电压的不确定性（当外输入悬空）

GPIO的工作模式—输入上拉模式



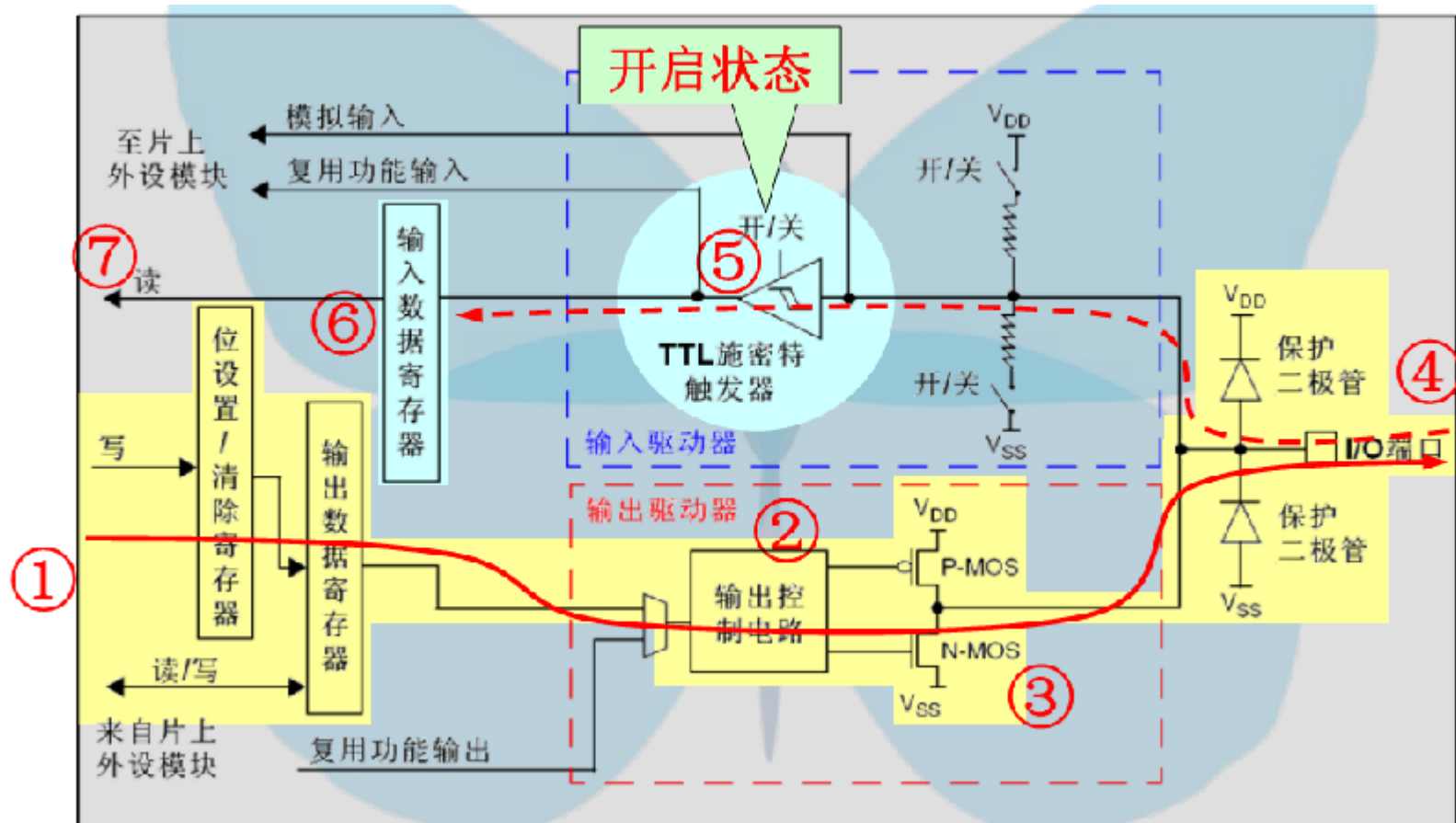
上拉输入模式的配置，与前面的浮空输入模式相比，仅仅是在数据通道上部，接入了一个上拉电阻(开关闭合)，根据STM32的数据手册，这个上拉电阻阻值介于30K~50K欧姆。同样，CPU可以随时在“输入数据寄存器”的另一端，读出I/O端口的电平状态。

上拉输入模式的特点：当输入没有信号输入（悬空）时输入数据寄存器能读出高电平状态

对于下拉输入模式的配置，数据通道的下部，接入了一个下拉电阻(开关闭合)，当输入没有信号输入（悬空）时，输入数据寄存器能读出低电平状态



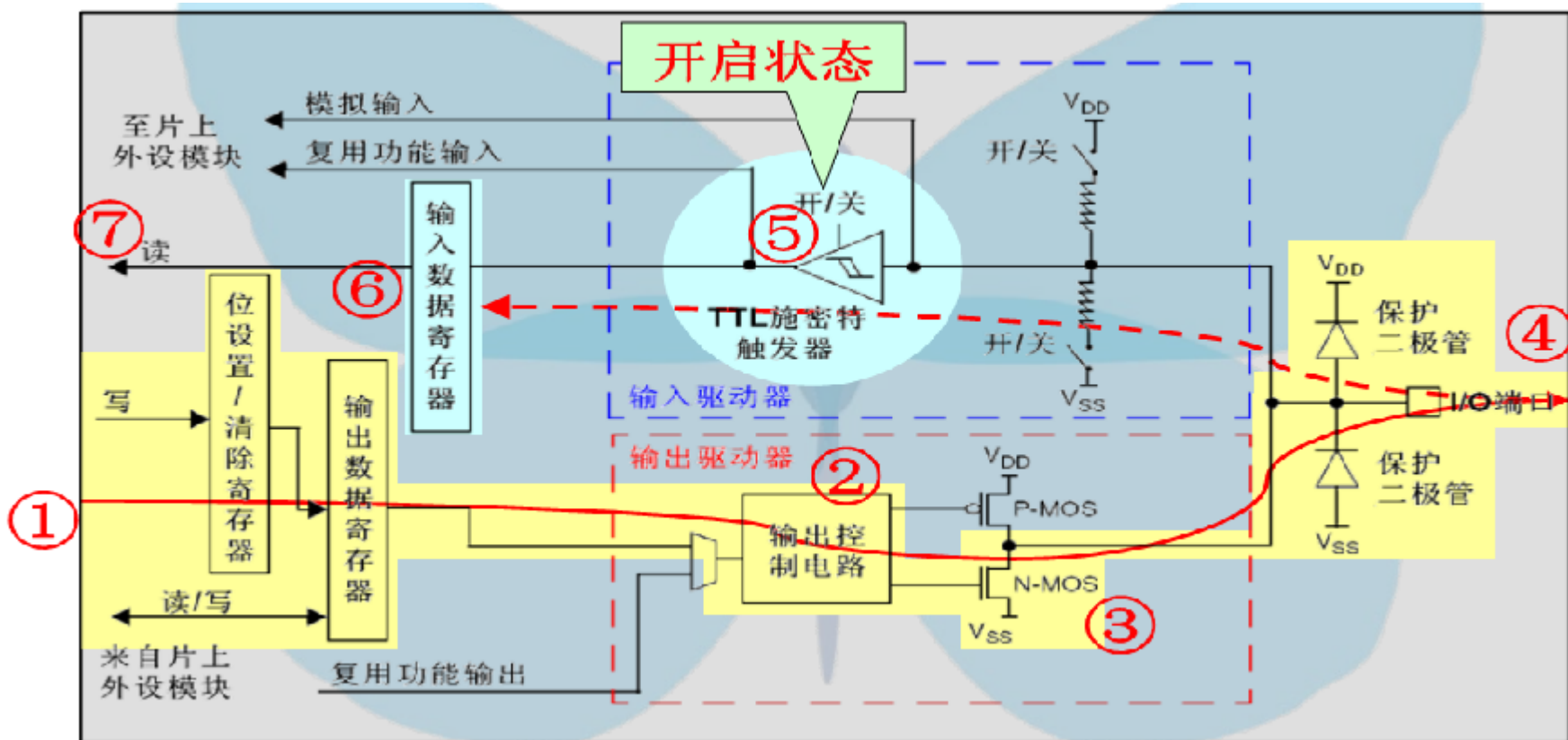
GPIO的工作模式—推挽输出模式



推挽输出模式： 在编号③的一个P-MOS管和下方的N-MOS管组成互补式。P-MOS管导通下方的N-MOS管截止，输出高电平；P-MOS管截止下方的N-MOS管导通，达到输出低电平。这个模式下CPU仍然可以从“输入数据寄存器”读到外部电路的信号。



GPIO的工作模式—开漏输出模式

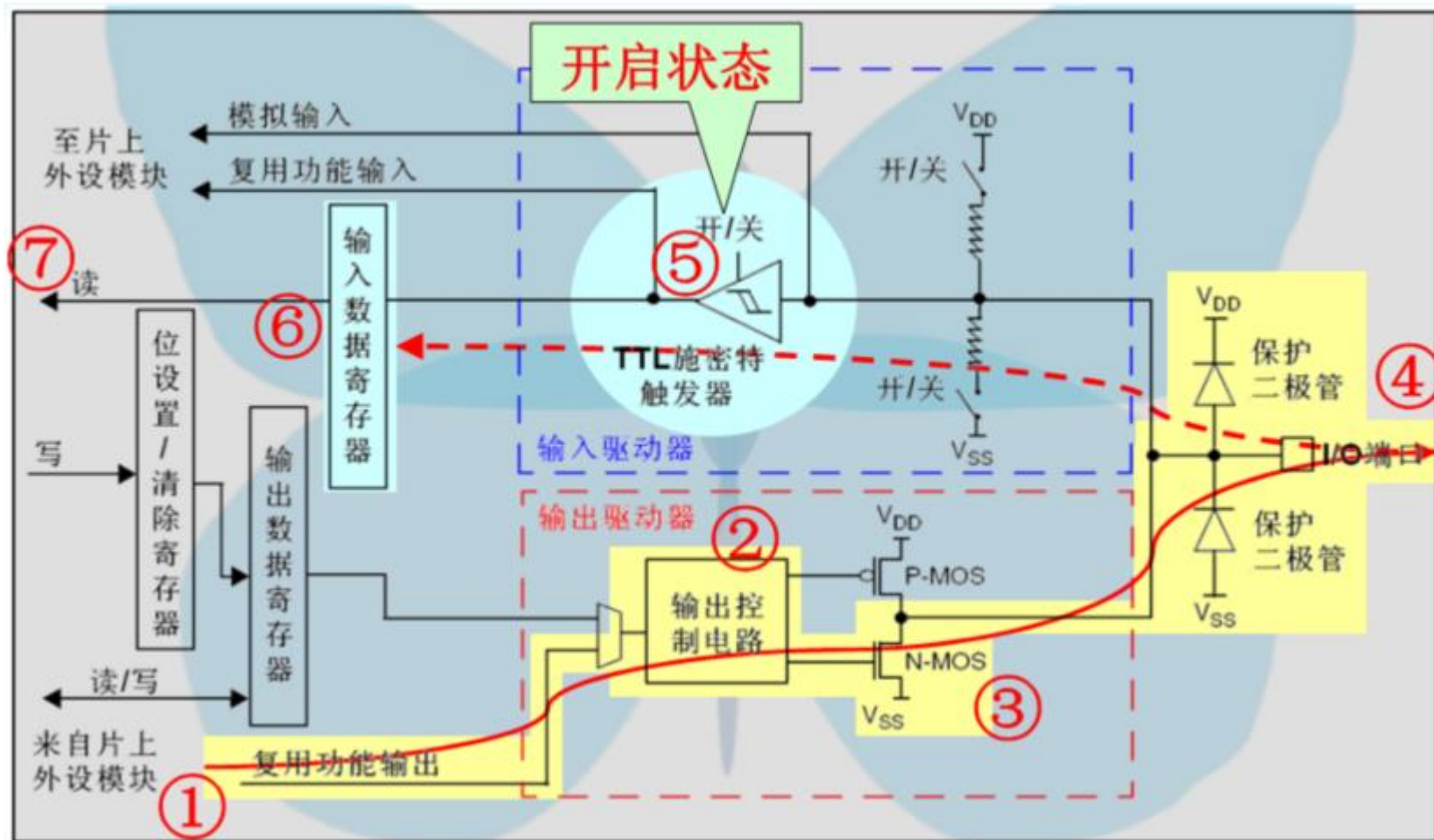


开漏输出模式：P-MOS管处于关闭（止状态），当N-MOS管为开启状态（饱和）时I/O端口的电平为低电平；当N-MOS管为关闭状态（止）时I/O端口的电平将完全由I/O端口外部的上拉电阻和电压决定（逻辑“1”）。CPU也可以在“输入数据寄存器”读到外部电路的信号，而不是它自己输出的逻辑“1”。

这里我们看到所有的上拉、下拉电阻和施密特触发器，均处于断开状态，因此“输入数据寄存器”将不能反映端口上的电平状态，也就是说，模拟输入配置下，CPU不能在“输入数据寄存器”上读到有效的数据。

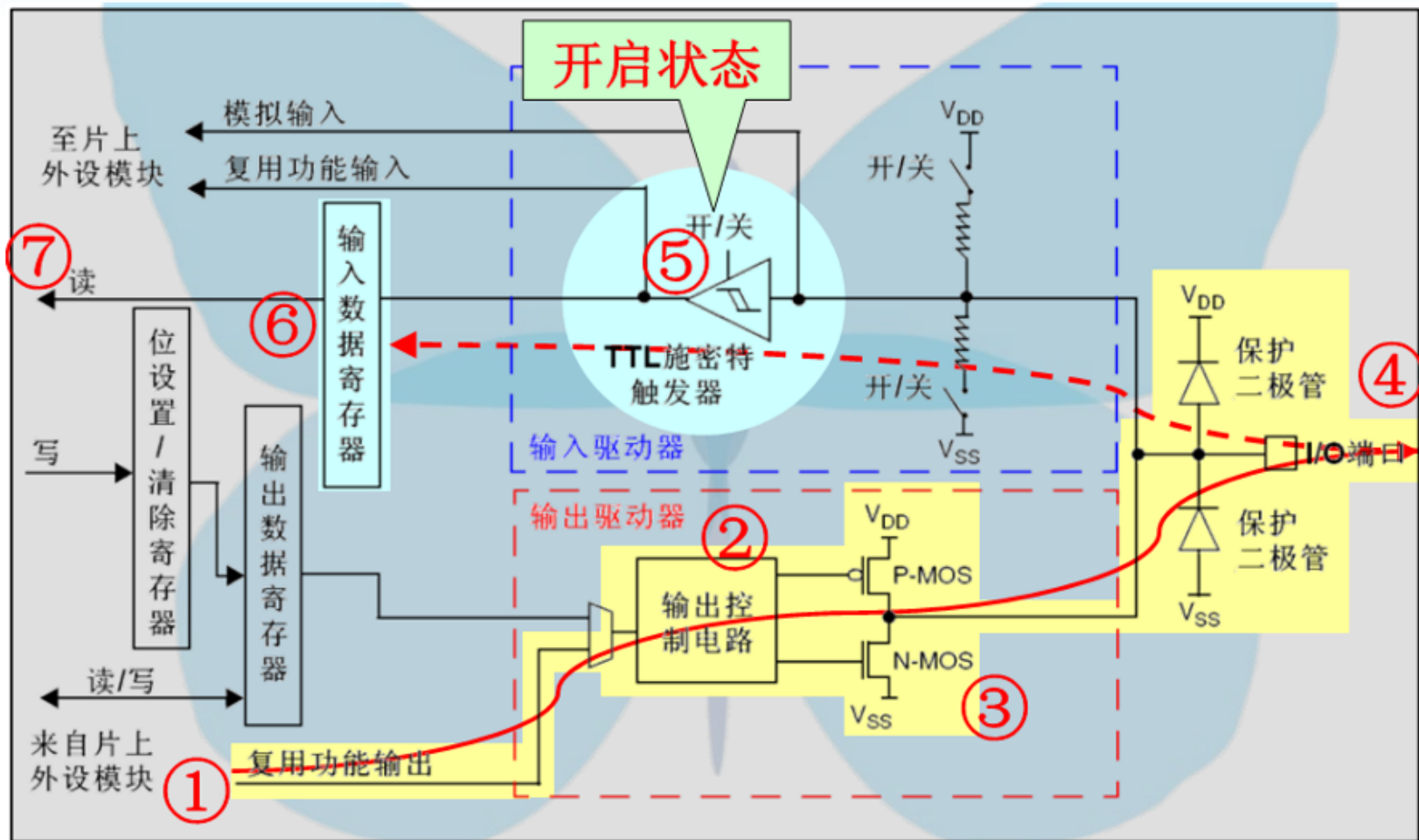


GPIO的工作模式—开漏复用输出模式



开漏复用输出模式与开漏输出模式的配置基本相同，不同的是编号2的输出控制电路的输入，与复用功能的输出端相连，此时输出数据寄存器被从输出通道断开了。同样，CPU可以从“输入数据寄存器”读到外部电路的信号。

GPIO的工作模式—推挽复用输出模式



最后是推挽复用输出模式，同样的道理，编号2的输出控制电路的输入，与复用功能的输出端相连，此时输出数据寄存器被从输出通道断开了。其它部分与前述模式一致，包括对“输入数据寄存器”的读取。



GPIO工作模式的用法总结

● 输入模式

- 浮空输入：按键识别
- 上拉输入：IO内部上拉电阻输入
- 下拉输入：IO内部下拉电阻输入

● 模拟模式

- 作为片内模拟外设的对外引脚
- 单纯作为低功耗使用

● 输出模式

- 推挽输出：可以输出高/电平，主要用于连接数字器件，如指示灯和继电器等模块；
- 开漏输出：只能输出低电平，适合于电流型驱动，也可作为电平转换。

● 复用模式

- 复用推挽：片内外设功能（URAT的TX, RX, SPI的MOSI, MISO, SCK, SS）；
- 复用开漏：片内外设功能（I2C的SCL, SDA）。



GPIO端口模式:

STM32每一个输入/输出引脚(即GPIO端口的每一位)可以配置成以下8种模式(4输入+2输出+2复用输出):

输入

- 输入浮空: **IN_FLOATING**
- 输入上拉: **IPU (In Push-Up)**
- 输入下拉: **IPD (In Push-Down)**
- 模拟输入: **AIN (Analog In)**

输出

- 开漏输出: **Out_OD (Open Drain Output)**
- 推挽式输出: **Out_PP (Push-Pull Output)**

复用输出

- 推挽式复用功能: **AF_PP (Push-Pull Output Alternate-Function)**
- 开漏复用功能: **AF_OD (Open Drain Output Alternate-Function)**



复用功能重映射

用户根据实际需要可以把某些外设的“复用功能”从“默认引脚”转移到“备用引脚”上，这就是外设复用功能的I/O引脚重映射。

GPIO重映射实例

STM32F103xx增强型
LQFP100管脚图

TIM4_CH1
TIM4_CH2
TIM4_CH3
TIM4_CH4

SPI1_NSS
SPI1_SCK
SPI1_MISO
SPI1_MOSI

USART2_CTS
USART2_RTS
USART2_TX
USART2_RX
USART2_CK

USART1_TX
USART1_RX

TIM3_CH1
TIM3_CH2
TIM3_CH3
TIM3_CH4

USART3_TX
USART3_RX
USART3_CK
USART3_CTS
USART3_RTS

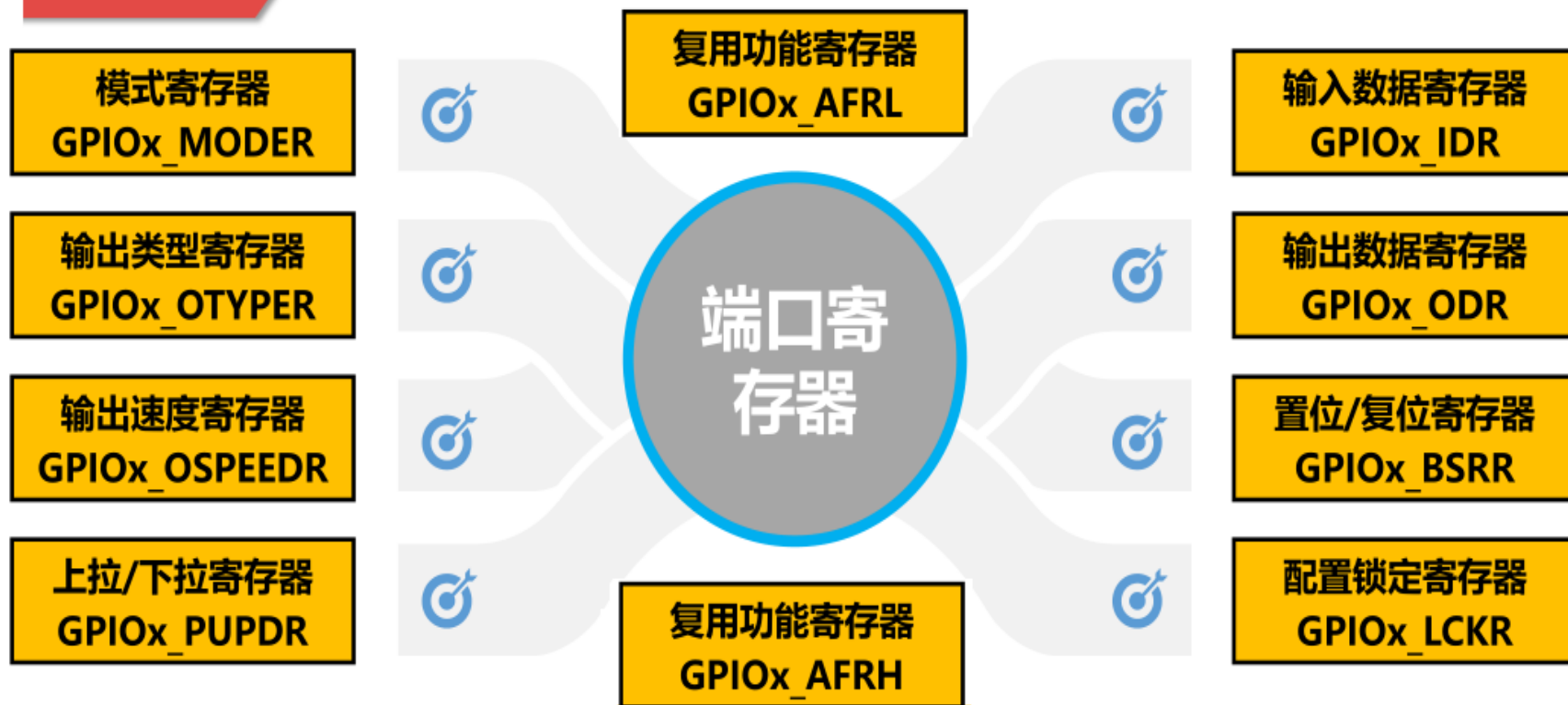
TIM2_CH3
TIM2_CH4



GPIO端口的寄存器

每一个端口都包括这10个寄存器，其中x表示端口号，取值从A~K

端口寄存器





GPIO端口的寄存器

每个GPIO端口有10个寄存器。

- 1)、一个端口模式寄存器 (GPIOx_MODER)
 - 2)、一个端口输出类型寄存器(GPIOx_OTYPER)
 - 3)、一个端口输出速度寄存器 (GPIOx_OSPEEDR)
 - 4)、一个端口上拉下拉寄存器 (GPIOx_PUPDR)
 - 5)、一个端口输入数据寄存器 (GPIOx_IDR)
 - 6)、一个端口输出数据寄存器 (GPIOx_ODR)
 - 7)、一个端口置位/复位寄存器 (GPIOx_BSRR)
 - 8)、一个端口配置锁存寄存器 (GPIOx_LCKR)
 - 9)、两个复用功能寄存器 (低位GPIOx_AFRL & GPIOx_AFRH)
- 4个32位配置寄存器
- 2个32位数据寄存器
- 每个GPIO端口的寄存器相互独立!

- ◆ 如果配置一个IO口需要2个位，那么刚好32位寄存器配置一组IO口16个IO口
- ◆ 如果配置一个IO口只需要1个位，一般高16位保留
- ◆ BSRR寄存器32位分为低16位BSRRL和高16位BSRRH，BSRRL配置一组IO口的16个IO口的置位状态（1），BSRRH配置复位状态（0）。



模式寄存器GPIOx_MODER

31 ~ 30	29 ~ 28	27 ~ 26	25 ~ 24	23 ~ 22	21 ~ 20	19 ~ 18	17 ~ 16
MODER	MODER	MODER	MODER	MODER	MODER	MODER	MODER
15[1:0]	14[1:0]	13[1:0]	12[1:0]	11[1:0]	10[1:0]	9[1:0]	8[1:0]
15 ~ 14	13 ~ 12	11 ~ 10	9 ~ 8	7 ~ 6	5 ~ 4	3 ~ 2	1 ~ 0
MODER	MODER	MODER	MODER	MODER	MODER	MODER	MODER
7[1:0]	6[1:0]	5[1:0]	4[1:0]	3[1:0]	2[1:0]	1[1:0]	0[1:0]

- 32位寄存器，每2位一组，用于设置对应引脚的工作模式，如：bit0和bit1控制Px0
- 00：输入模式（复位值） 01：输出模式 10：复用模式 11：模拟模式

输出类型寄存器GPIOx_OTYPER

31 ~ 16															
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT	OT	OT	OT	OT	OT	OT	OT	OT	OT	OT	OT	OT	OT	OT	OT
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- 32位寄存器，高16位保留，低16位的每1位用于设置对应引脚的输出类型
- 0：推挽输出（复位值） 1：开漏输出



输出速度寄存器GPIOx_OSPEEDR

31 ~ 30	29 ~ 28	27 ~ 26	25 ~ 24	23 ~ 22	21 ~ 20	19 ~ 18	17 ~ 16
OSPEED R 15[1:0]	OSPEE DR 14[1:0]	OSPEED R 13[1:0]	OSPEED R 12[1:0]	OSPEED R 11[1:0]	OSPEED R 10[1:0]	OSPEED R 9[1:0]	OSPEED R 8[1:0]
15 ~ 14	13 ~ 12	11 ~ 10	9 ~ 8	7 ~ 6	5 ~ 4	3 ~ 2	1 ~ 0
OSPEED R 7[1:0]	OSPEE DR 6[1:0]	OSPEED R 5[1:0]	OSPEED R 4[1:0]	OSPEED R 3[1:0]	OSPEED R 2[1:0]	OSPEED R 1[1:0]	OSPEED R 0[1:0]

- 32位寄存器，每2位一组，用于设置对应引脚的输出速度
- 00：低速（复位值） 01：中速 10：高速 11：超高速

STM32微控制器I/O管脚内部有多个响应速度（2MHz、25MHz、50MHz、100MHz）不同的驱动电路，用户可以根据自己的需要选择合适的驱动电路。一般推荐I/O引脚的输出速度是其输出信号速度的5-10倍。

- 对于连接LED、数码管和蜂鸣器等外部设备，一般设置为低速。
- 对于串口来说，这样只需要用中速的GPIO的引脚速度就可以了。
- 对于I2C接口，可以选用高速的GPIO引脚速度。
- 对于SPI接口，需要选择呢超高速的GPIO引脚速度
- 对于用作FSMC复用功能连接存储器的输出引脚，一般设置为超速度。



上拉/下拉寄存器GPIOx_PUPDR

31 ~ 30	29 ~ 28	27 ~ 26	25 ~ 24	23 ~ 22	21 ~ 20	19 ~ 18	17 ~ 16
PUPDR 15[1:0]	PUPDR 14[1:0]	PUPDR 13[1:0]	PUPDR 12[1:0]	PUPDR 11[1:0]	PUPDR 10[1:0]	PUPDR 9[1:0]	PUPDR 8[1:0]
15 ~ 14	13 ~ 12	11 ~ 10	9 ~ 8	7 ~ 6	5 ~ 4	3 ~ 2	1 ~ 0
PUPDR 7[1:0]	PUPDR 6[1:0]	PUPDR 5[1:0]	PUPDR 4[1:0]	PUPDR 3[1:0]	PUPDR 2[1:0]	PUPDR 1[1:0]	PUPDR 0[1:0]

- 32位寄存器，每2位一组，用于使能对应引脚的上拉/下拉电阻
- 00：无上拉和下拉电阻（复位值） 01：使能上拉电阻 10：使能下拉电阻 11：保留

输入数据寄存器GPIOx_IDR

31 ~ 16															
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID R15	ID R14	ID R13	ID R12	ID R11	ID R10	ID R9	ID R8	ID R7	ID R6	ID R5	ID R4	ID R3	ID R2	ID R1	ID R0

- 32位寄存器，高16位保留，低16位的每1位用于存放对应引脚的电平状态
- 0：对应引脚输入低电平 1：对应引脚输入高电平



输出数据寄存器GPIOx_ODR

31 ~ 16															
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OD R15	OD R14	OD R13	OD R12	OD R11	OD R10	OD R9	OD R8	OD R7	OD R6	OD R5	OD R4	OD R3	OD R2	OD R1	OD R0

- 32位寄存器，高16位保留，低16位的每1位用于控制对应引脚输出高/低电平
- **0**：控制对应引脚输出低电平 **1**：控制对应引脚输出高电平

置位/复位寄存器GPIOx_BSRR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR 15	BR 14	BR 13	BR 12	BR 11	BR 10	BR 9	BR 8	BR 7	BR 6	BR 5	BR 4	BR 3	BR 2	BR 1	BR 0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS 15	BS 14	BS 13	BS 12	BS 11	BS 10	BS 9	BS 8	BS 7	BS 6	BS 5	BS 4	BS 3	BS 2	BS 1	BS 0

- 高16位控制对应引脚输出低电平：写入**1**对应引脚输出**低电平**；写入0，没有任何作用
- 低16位控制对应引脚输出高电平：写入**1**对应引脚输出**高电平**；写入0，没有任何作用

复用功能低位寄存器 (GPIOx_AFRL)和复用功能高位寄存器 (GPIOx_AFRH)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:0 **AFRLy**: 端口 x 位 y 的复用功能选择 (Alternate function selection for port x bit y) (y = 0..7)

这些位通过软件写入，用于配置复用功能 I/O。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

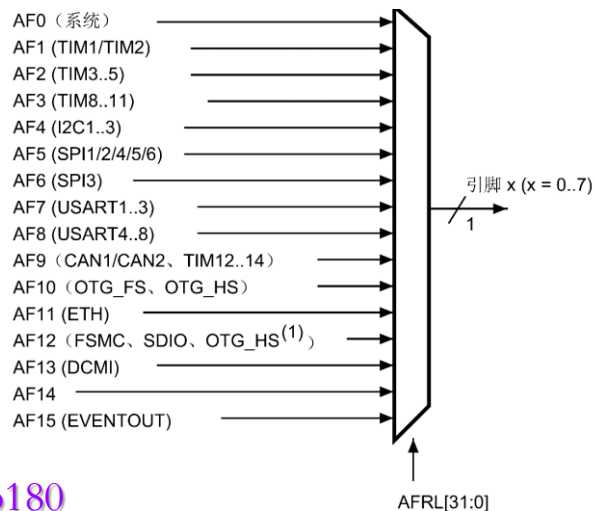
位 31:0 **AFRH_y**: 端口 x 位 y 的复用功能选择 (Alternate function selection for port x bit y) (y = 8..15)

这些位通过软件写入，用于配置复用功能 I/O。

AFRL_y 选择:

0000: AF0	1000: AF8
0001: AF1	1001: AF9
0010: AF2	1010: AF10
0011: AF3	1011: AF11
0100: AF4	1100: AF12
0101: AF5	1101: AF13
0110: AF6	1110: AF14
0111: AF7	1111: AF15

GPIOx_AFRL定义的
0-7号引脚复用功能

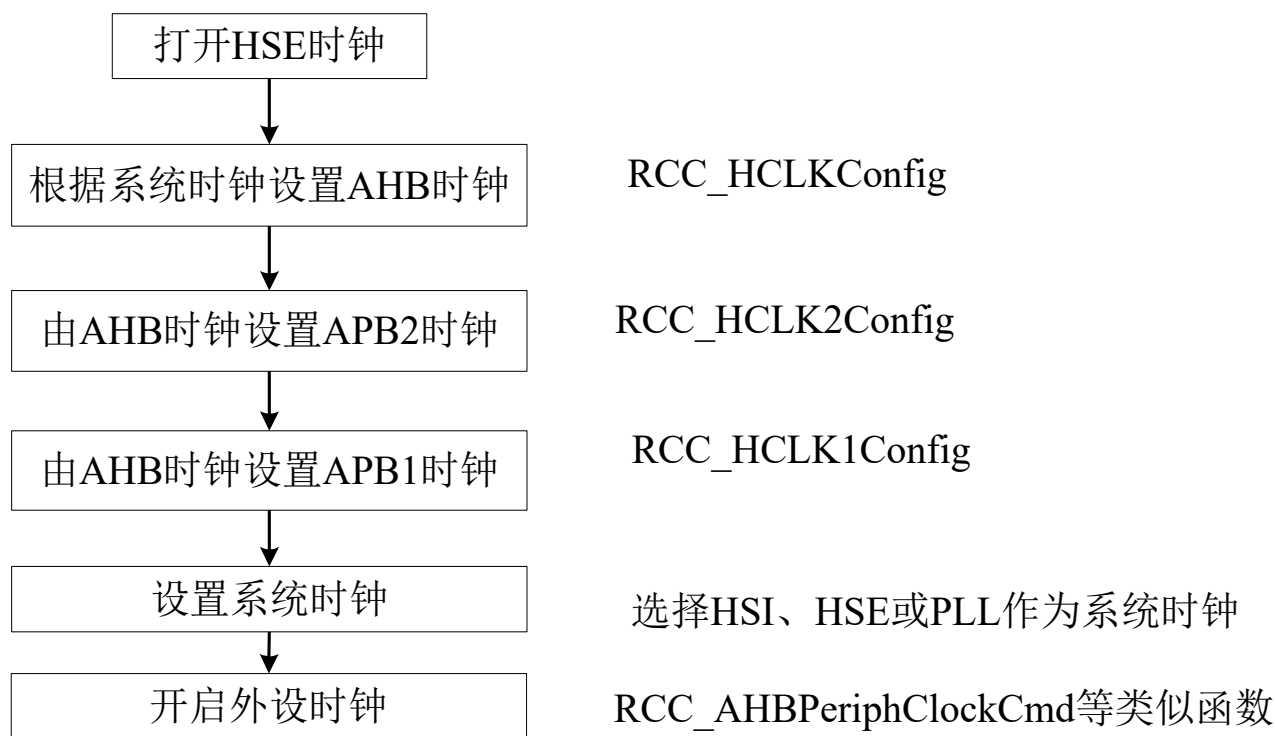




RCC时钟模块的寄存器

时钟配置是初始化的一项重要工作，**要使用某个外设，必须使能该外设的时钟**。时钟配置需要先考虑系统时钟的来源（内部时钟、外部时钟、外部振荡器），并且是否需要锁相环（PLL）。然后再考虑内部总线和外部总线，最后考虑外设的时钟信号。应遵从先倍频作为处理器的时钟，然后再由内向外分频的原则。

时钟配置主要就是进行设置RCC时钟模块的寄存器(流程)



大部分片上外设挂载在**AHB**和**APB**总线上，总线时钟**HCLK**、**PCLK1**、**PCLK2**也就提供了大多数片上外设的工作时钟。要想使用片上外设，相应编程序首先要使能对应总线到片上外设的时钟。

[illegible]



STM32F4XX参考手册
(RM0090) p135

RCC AHB1 外设时钟使能寄存器 (RCC_AHB1ENR)
RCC AHB2 外设时钟使能寄存器 (RCC_AHB2ENR)
RCC APB1 外设时钟 使能寄存器 (RCC_APB1ENR)
RCC APB2 外设时钟 使能寄存器 (RCC_APB2ENR)

```
#define RCC_AHB1Periph_GPIOD ((uint32_t)0x00000008) 1<<3
```

例: `RCC_AHB1PeriphClockCmd (RCC_AHB1Periph_GPIOD, ENABLE);`

使能GPIOD的工作时钟，操作的寄存器是RCC_AHB1ENR：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	OTGHS ULPIEN	OTGHS EN	ETHMA CPTPE N	ETHMA CRXEN	ETHMA CTXEN	ETHMA CEN	Reserved			DMA2EN	DMA1EN	CCMDATA RAMEN	Res.	BKPSR AMEN	Reserved
	rw	rw	rw	rw	rw	rw				rw	rw			rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Reserved			GPIOE N	GPIOH EN	GPIOE N	GPIOF E N	GPIOE EN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
			rw				rw	rw	rw	rw	rw	rw	rw	rw	rw

位3 **GPIODEN**: IO 端口 D 时钟使能 (IO port D clock enable)
由软件置 1 和清零。

- 0: 禁止 IO 端口 D 时钟
- 1: 使能 IO 端口 D 时钟



在STM32 中凡是使用到外设功能，都要使能对应的外设时钟，否则即使配置好端口初始化也无法正常使用。因此还需要知道时钟 RCC 外设的基地址，通过《STM32F4xx 中文参考手册》“存储器和总线架构”的“存储器映射”章节可以知道 RCC 时钟外设也是挂接在 AHB1 总线上，根据其偏移值可以得到 RCC 时钟外设的基地址。找到对应的 RCC 使能寄存器。

RCC AHB1 外设时钟使能寄存器 (RCC_AHB1ENR) 偏移地址: 0x30

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	OTGHS ULPIEN	OTGHS EN	ETHMA CPTPE N	ETHMA CRXEN	ETHMA CTXEN	ETHMA CEN	Reserved			DMA2EN	DMA1EN	CCMDATA RAMEN	Res.	BKPSR AMEN	Reserved
	rw	rw	rw	rw	rw	rw				rw	rw			rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Reserved			GPIOIE N	GPIOH EN	GPIOGE N	GPIOFE N	GPIOEEN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
			rw				rw	rw	rw	rw	rw	rw	rw	rw	rw

位 3 **GPIODEN**: IO 端口 D 时钟使能 (IO port D clock enable)

由软件置 1 和清零。

0: 禁止 IO 端口 D 时钟

1: 使能 IO 端口 D 时钟

位 2 **GPIOCEN**: IO 端口 C 时钟使能 (IO port C clock enable)

由软件置 1 和清零。

0: 禁止 IO 端口 C 时钟

1: 使能 IO 端口 C 时钟

位 1 **GPIOBEN**: IO 端口 B 时钟使能 (IO port B clock enable)

由软件置 1 和清零。

0: 禁止 IO 端口 B 时钟

1: 使能 IO 端口 B 时钟

位 0 **GPIOAEN**: IO 端口 A 时钟使能 (IO port A clock enable)

由软件置 1 和清零。

0: 禁止 IO 端口 A 时钟

1: 使能 IO 端口 A 时钟

位 6 **GPIOGEN**: IO 端口 G 时钟使能 (IO port G clock enable)

由软件置 1 和清零。

0: 禁止 IO 端口 G 时钟

1: 使能 IO 端口 G 时钟

位 5 **GPIOFEN**: IO 端口 F 时钟使能 (IO port F clock enable)

由软件置 1 和清零。

0: 禁止 IO 端口 F 时钟

1: 使能 IO 端口 F 时钟

位 4 **GPIOEEN**: IO 端口 E 时钟使能 (IO port E clock enable)

由软件置 1 和清零。

0: 禁止 IO 端口 E 时钟

1: 使能 IO 端口 E 时钟



使用举例

要求PA0输出低电平，PA15输出高电平

方法一：使用输出数据寄存器实现

```
unsigned int  value;           // 定义变量value，存放寄存器的值
value = GPIOA_ODR;           // 读出输出数据寄存器原有的值
value = value & 0xFFFFF000;  // 设置value变量的bit0为0
value = value | 0x00008000;   // 设置value变量的bit15为1
GPIOA_ODR = value;           // 将新的配置数据写入输出数据寄存器
```

方法二：使用置位/复位寄存器实现

```
GPIOA_BSRR = 0x00018000; // 设置PA0输出低，PA15输出高
```

结论：设置多个引脚输出高/低电平时，使用置位/复位寄存器更加简单



基于寄存器方式控制GPIO

利用指针完成地址转换

1. 对于C语言的编译器而言，寄存器的地址值只代表一个有符号的常数，无法代表地址；
2. 在C语言中，利用指针类型来存放变量的地址，利用指针类型定义的变量称为指针变量；
3. 利用强制类型转换可以将常数转为指针变量；
4. 指针的基类型表示指针所指向变量的类型，它决定了从该地址开始，可以访问的地址范围；
5. 指针的解引用表示从指针所指向的地址中取出存放的数据。

unsigned int *p 表示定义一个指向无符号整型的指针

p = 0x40020000UL 表示为指针变量p赋值，指向地址单元0x40020000

***p** 即指针的解引用，表示访问从地址单元0x40020000开始的4个地址单元的内容

寄存器定义： `#define GPIOA_MODER
*(volatile unsigned int *) (0x40020000UL)`

STM32F411芯片的GPIO模块各端口寄存器的地址范围

端口号	寄存器组地址范围	备注
GPIOA	0x4002 0000 ~ 0x4002 03FF	每组端口占用了 0x000 ~ 0x3FF共1K字节的存储空间，而实际每组端口只包括10个32位的寄存器，只需要40个字节的存储空间，多余的存储空间预留給芯片升级使用。
GPIOB	0x4002 0400 ~ 0x4002 07FF	
GPIOC	0x4002 0800 ~ 0x4002 0BFF	
GPIOD	0x4002 0C00 ~ 0x4002 0FFF	
GPIOE	0x4002 1000 ~ 0x4002 13FF	
GPIOH	0x4002 1C00 ~ 0x4002 1FFF	

端口GPIOA对应寄存器的起始地址及偏移量

寄存器名称	寄存器起始地址	偏移量
GPIOA_MODER	0x4002 0000	0x00
GPIOA_OTYPER	0x4002 0004	0x04
GPIOA_OSPEEDR	0x4002 0008	0x08
GPIOA_PUPDR	0x4002 000C	0x0C
GPIOA_IDR	0x4002 0010	0x10
GPIOA_ODR	0x4002 0014	0x14
GPIOA_BSRR	0x4002 0018	0x18
GPIOA_LCKR	0x4002 001C	0x1C
GPIOA_AFRH	0x4002 0020	0x20
GPIOA_AFRH	0x4002 0024	0x24

0x40020000

作为模式寄存器的起始地址，也是端口GPIOA寄存器组的起始地址

所有寄存器为32位，占用四个字节的存储空间



控制指示灯LD2连续闪烁

```
1. #define GPIOA_MODER    *(unsigned int *) (0x40020000UL) // 模式寄存器
2. #define GPIOA_ODR      *(unsigned int *) (0x40020014UL) // 输出数据寄存器
3. #define RCC_AHB1ENR    *(unsigned int *) (0x40023830UL) // 外设时钟使能寄存器
4. int main()
5. {
6.     unsigned int delay = 1000000; // 延时变量
7.     RCC_AHB1ENR |= 1<<0;           // 开启端口 GPIOA 的时钟
8.     GPIOA_MODER &= ~(3<<(5*2));    // 清除模式寄存器的 bit11:bit10 为 00
9.     GPIOA_MODER |= 1<<(5*2);       // 设置 bit11:bit10 = 01, PA5 为推挽输出类型
10.    while(1)
11.    {
12.        GPIOA_ODR |= 1<<5;          // 设置 bit5 为 1, PA5 输出高电平, 开启 LD2
13.        delay = 1000000;
14.        while(delay--);             // 延时
15.        GPIOA_ODR &= ~(1<<5);       // 设置 bit5 为 0, PA5 输出低电平, 关闭 LD2
16.        delay = 1000000;
17.        while(delay--);             // 延时
18.    }
19. }
```

芯片复位后, 引脚默认的输出类型为推挽输出

扩展任务
使用置位/复位寄存器
BSRR控制指示灯LD2

在main.c文件中添加用户代码



利用结构指针访问寄存器组

地址连续 → 利用结构体实现

为系统固有的或用户自定义的数据类型定义一个别名

```
struct GPIO
{
    volatile unsigned int MODER;
    volatile unsigned int OTYPER;
    volatile unsigned int OSPEEDR;
    volatile unsigned int PUPDR;
    volatile unsigned int IDR;
    volatile unsigned int ODR;
    volatile unsigned int BSRR;
    volatile unsigned int LCKR;
    volatile unsigned int AFR[2];
};
```

目前只定义结构体模板，
还没有定义结构体变量

```
typedef struct
{
    __IO uint32_t MODER;
    __IO uint32_t OTYPER;
    __IO uint32_t OSPEEDR;
    __IO uint32_t PUPDR;
    __IO uint32_t IDR;
    __IO uint32_t ODR;
    __IO uint32_t BSRR;
    __IO uint32_t LCKR;
    __IO uint32_t AFR[2];
} GPIO_TypeDef;
```

别名



两种方式在定义结构体变量时的区别

利用结构体模板定义结构体变量

```
struct GPIO    GPIOA;    // 定义结构体变量
struct GPIO    * pGPIOA; // 定义指向结构体变量的指针
```

利用typedef定义结构体变量

```
GPIO_TypeDef    GPIOA;    // 定义结构体变量
GPIO_TypeDef    * pGPIOA; // 定义指向结构体变量的指针
```

例如

```
#define GPIOA ((GPIO_TypeDef *) 0x40020000UL)
```

0x40020000UL 将有符号常数转换为一个无符号32位常数



GPIO_TypeDef *(0x40020000UL)表示
将常数强制转换为指向GPIO_TypeDef类型的结构体指针



利用define取别名，用GPIOA作为结构体指针的别名

结构体指针加成员变量的形式访问硬件寄存器
GPIOA -> MODER; GPIOA -> OTYPER

GPIO端口的定义

```
#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE ((GPIO_TypeDef *) GPIOE_BASE)
#define GPIOH ((GPIO_TypeDef *) GPIOH_BASE)
```

GPIOx_BASE表示该端口寄存器组的起始地址

利用结构体指针+成员变量的形式访问寄存器，如 GPIOC -> MODER

- ① 为了便于用户使用，ST公司将STM32微控制器片内所有外设的寄存器都采用上述的方法进行了定义，以 **.h** 文件的形式提供给用户；
- ② 用户在使用时，只需包含该头文件，就可以通过外设的结构体指针访问外设的相关寄存器；
- ③ 不同型号微控制器的头文件各不相同。以STM32F411系列微控制器为例，对应的.h文件为**stm32f411xe.h**；

控制指示灯LD2连续闪烁2



基于寄存器方式控制GPIO

本质：配置寄存器

```
1. #include "stm32f411xe.h" // 头文件中包含
2. int main()
3. {
4.     unsigned int delay = 1000000; // 延时变量
5.     RCC->AHB1ENR |= 1<<0; // 开启端口 GPIOA 的时钟
6.     GPIOA->MODER &= ~(3<<(5*2)); // 清除模式寄存器
7.     GPIOA->MODER |= 1<<(5*2); // 设置 bit11:10 为输出模式
8.     while(1)
9.     {
10.        GPIOA->BSRR |= 1<<5; // 设置 bit5 为 1，清除 PA5 上的低电平
11.        delay = 1000000;
12.        while(delay--); // 延时
13.        GPIOA->BSRR |= 1<<(5+16); // 设置 bit21 为 1，PA5 输出低电平，关闭 LD2
14.        delay = 1000000;
15.        while(delay--); // 延时
16.    }
17. }
```

GPIO引脚的初始化
根据引脚的编号和功能来设置相关寄存器的对应位

GPIO引脚的操作

- 设置相关寄存器的对应位控制GPIO引脚输出高/低电平（ODR和BSRR）
- 读取相关寄存器的对应位获取GPIO引脚的电平状态（IDR）

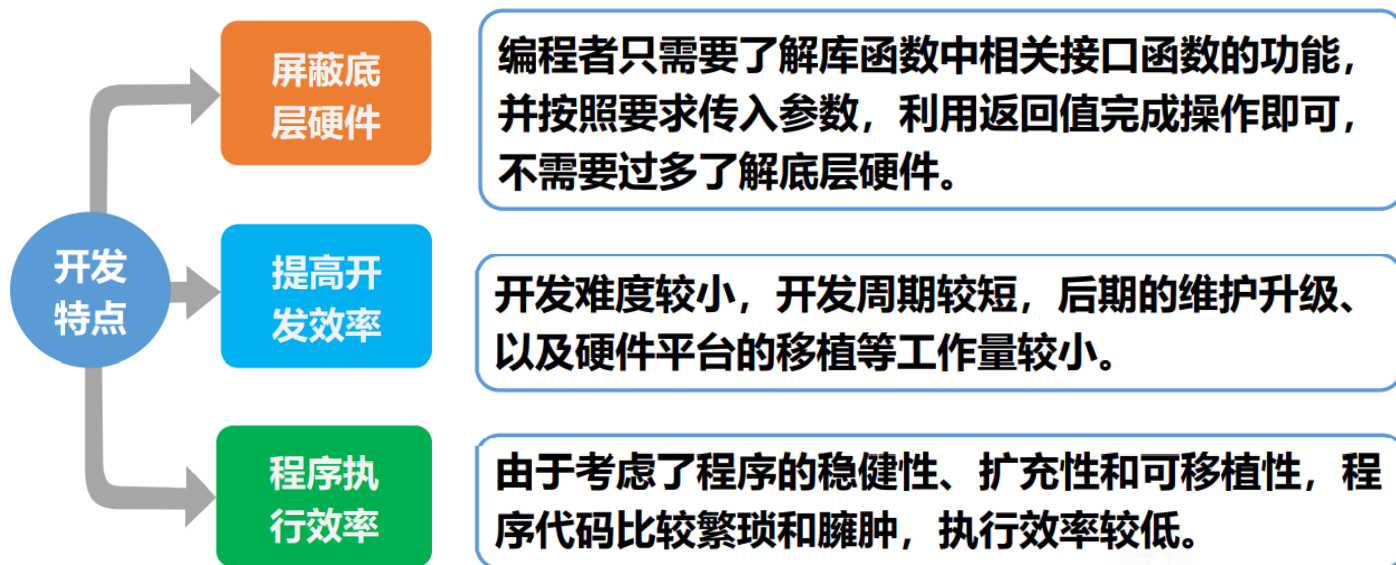


基于库函数方式的GPIO控制

汇编是面向硬件底层的语言工具，不适合编写大型、复杂的程序。C是结构化的高级语言，是嵌入式系统开发主要使用的语言工具。无论是采用汇编语言还是采用C语言，实际上所有MCU的应用开发都是靠配置寄存器来完成的。因此如何充分发挥C语言结构化的优点，上面是由C语言操作寄存器来实现GPIO控制。

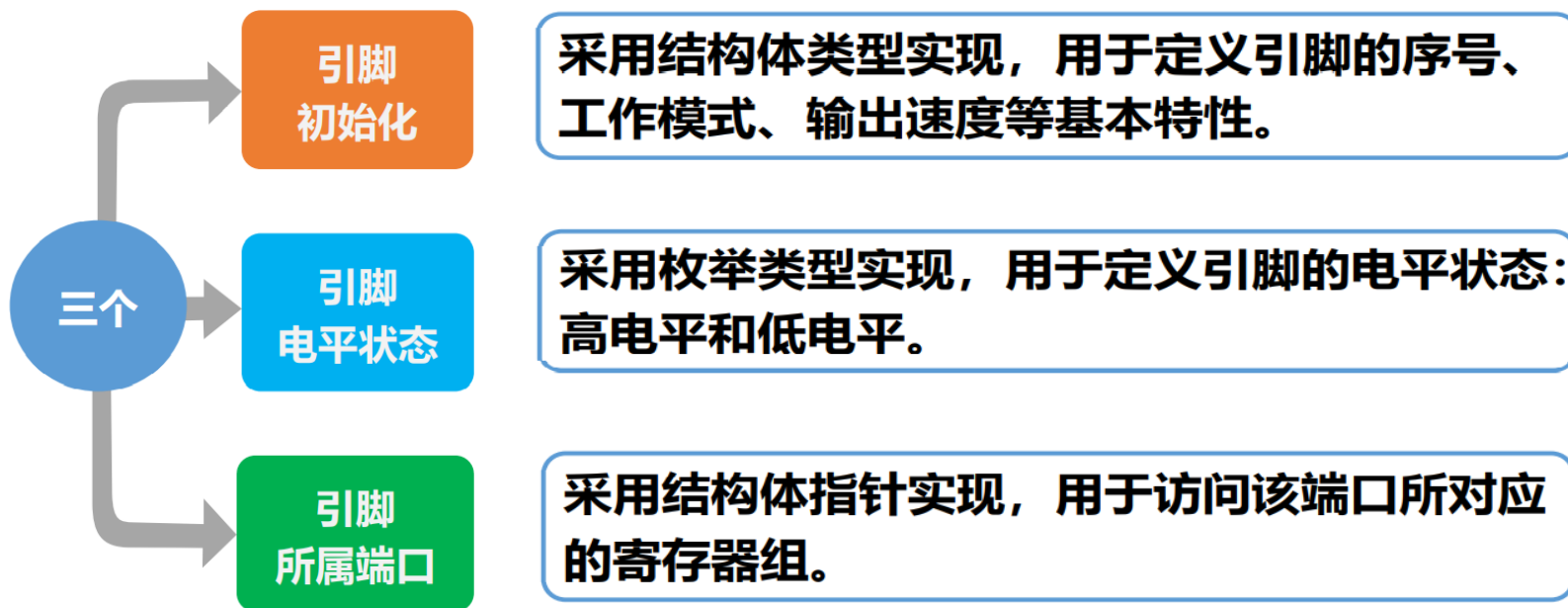
通过库函数方式调用规范化的、经过封装的库函数来间接配置寄存器开发嵌入式应用。开发者可调用这些函数接口来配置STM32的寄存器，使开发人员得以脱离最底层的寄存器操作，有开发快速，易于阅读，维护成本低等优点。

基于库函数的程序开发方式的特点

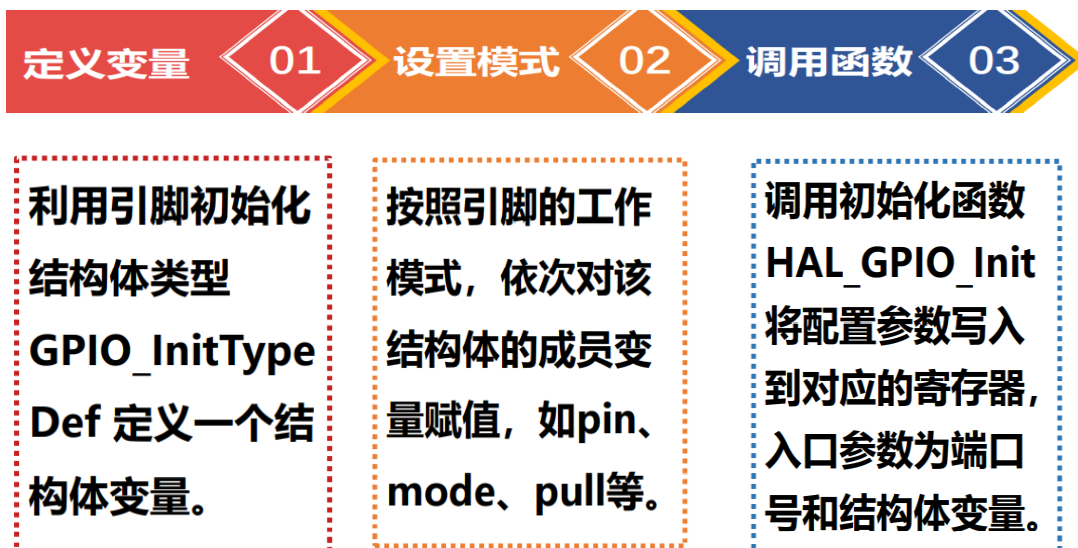




• GPIO外设数据类型



使用函数库的 引脚初始化步骤





➤ 引脚初始化数据类型

```
typedef struct
{
    uint32_t          GPIO_Pin;      //指定要初始化的端口
    GPIOMode_TypeDef  GPIO_Mode;     //端口模式
    GPIO_Speed_TypeDef GPIO_Speed;    //速度
    GPIO_OType_TypeDef GPIO_OType;    //输出类型
    GPIOPuPd_TypeDef  GPIO_PuPd;     //上拉或者下
} GPIO_InitTypeDef;
```

```
typedef enum
{
    GPIO_Mode_IN  = 0x00, /*!< GPIO Input Mode */
    GPIO_Mode_OUT = 0x01, /*!< GPIO Output Mode */
    GPIO_Mode_AF  = 0x02, /*!< GPIO Alternate function Mode */
    GPIO_Mode_AN  = 0x03 /*!< GPIO Analog Mode */
}GPIOMode_TypeDef;
```

Table 183. GPIO_Pin 值

GPIO_Pin	描述
GPIO_Pin_None	无管脚被选中
GPIO_Pin_0	选中管脚 0
GPIO_Pin_1	选中管脚 1
GPIO_Pin_2	选中管脚 2
GPIO_Pin_3	选中管脚 3
GPIO_Pin_4	选中管脚 4
GPIO_Pin_5	选中管脚 5
GPIO_Pin_6	选中管脚 6
GPIO_Pin_7	选中管脚 7
GPIO_Pin_8	选中管脚 8
GPIO_Pin_9	选中管脚 9
GPIO_Pin_10	选中管脚 10
GPIO_Pin_11	选中管脚 11
GPIO_Pin_12	选中管脚 12
GPIO_Pin_13	选中管脚 13
GPIO_Pin_14	选中管脚 14
GPIO_Pin_15	选中管脚 15
GPIO_Pin_All	选中全部管脚



Table 184. GPIO_Speed 值

GPIO_Speed	描述
GPIO_Speed_10MHz	最高输出速率 10MHz
GPIO_Speed_2MHz	最高输出速率 2MHz
GPIO_Speed_50MHz	最高输出速率 50MHz

```
typedef enum
{
    GPIO_OType_PP = 0x00,
    GPIO_OType_OD = 0x01
} GPIOOType_TypeDef;
```

```
typedef enum
{
    GPIO_PuPd_NOPULL = 0x00,
    GPIO_PuPd_UP      = 0x01,
    GPIO_PuPd_DOWN    = 0x02
} GPIOPuPd_TypeDef;
```



➤ 引脚电平状态数据类型

```
typedef enum
{
    Bit_RESET = 0,    //引脚低电平状态
    Bit_SET           //引脚高电平状态
} BitAction;
```

➤ 端口数据类型：指向端口寄存器组的结构体指针

宏常量定义	含义
GPIOA	选择端口A
GPIOB	选择端口B
GPIOC	选择端口C
GPIOD	选择端口D
GPIOE	选择端口E
GPIOH	选择端口H

注意

- ① 不同型号的STM32微控制器的端口数量各不相同；
- ② 端口数据类型的定义是在以芯片型号命名的.h文件中，如STM32F411系列MCU对应的头文件stm32f411xe.h



例：GPIO引脚初始化

- ① 设置引脚PF9和PF10为推挽输出模式，速度100MHz，还能上拉功能；
- ② 设置引脚PC13为浮空输入模式；

可以一次初始化一个IO组下的多个IO，前提是这些IO口的配置方式一样。

引脚初始化程序

```
GPIO_InitTypeDef  GPIO_InitStruct;           //定义引脚初始化变量

//配置引脚PF9和PF10初始化设置

GPIO_InitStruct.GPIO_Pin = GPIO_Pin_9|GPIO_Pin_10; //需要配置的IO口引脚
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_OUT;         //普通输出模式
GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;        //推挽输出
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_100MHz;    //100MHz
GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_UP;          //上拉

GPIO_Init(GPIOF, &GPIO_InitStruct); //调用初始化函数完成引脚PF9,F10初始化

//配置引脚PC13初始化设置

GPIO_InitStruct.GPIO_Pin = GPIO_Pin_13;           //需要配置的IO口引脚
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN;          //输入模式
GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;      //无上拉、无下拉

GPIO_Init(GPIOC, &GPIO_InitStruct); //调用初始化函数完成引脚PC13初始化
```

GPIO库函数

GPIO外设接口函数的概述

**Table 179. GPIO 库函数**

函数名	描述
GPIO_DeInit	将外设 GPIOx 寄存器重设为缺省值
GPIO_AFIODeInit	将复用功能（重映射事件控制和 EXTI 设置）重设为缺省值
GPIO_Init	根据 GPIO_InitStruct 中指定的参数初始化外设 GPIOx 寄存器
GPIO_StructInit	把 GPIO_InitStruct 中的每一个参数按缺省值填入
GPIO_ReadInputDataBit	读取指定端口管脚的输入
GPIO_ReadInputData	读取指定的 GPIO 端口输入
GPIO_ReadOutputDataBit	读取指定端口管脚的输出
GPIO_ReadOutputData	读取指定的 GPIO 端口输出
GPIO_SetBits	设置指定的数据端口位
GPIO_ResetBits	清除指定的数据端口位
GPIO_WriteBit	设置或者清除指定的数据端口位
GPIO_Write	向指定 GPIO 数据端口写入数据
GPIO_PinLockConfig	锁定 GPIO 管脚设置寄存器
GPIO_EventOutputConfig	选择 GPIO 管脚用作事件输出
GPIO_EventOutputCmd	使能或者失能事件输出
GPIO_PinRemapConfig	改变指定管脚的映射
GPIO_EXTILineConfig	选择 GPIO 管脚用作外部中断线路




GPIO外设接口常用函数

1个初始化函数:

```
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
```

2个读取输入电平函数: IDR

```
uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx,
                               uint16_t GPIO_Pin);
uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx);
```



2个读取输出电平函数: ODR

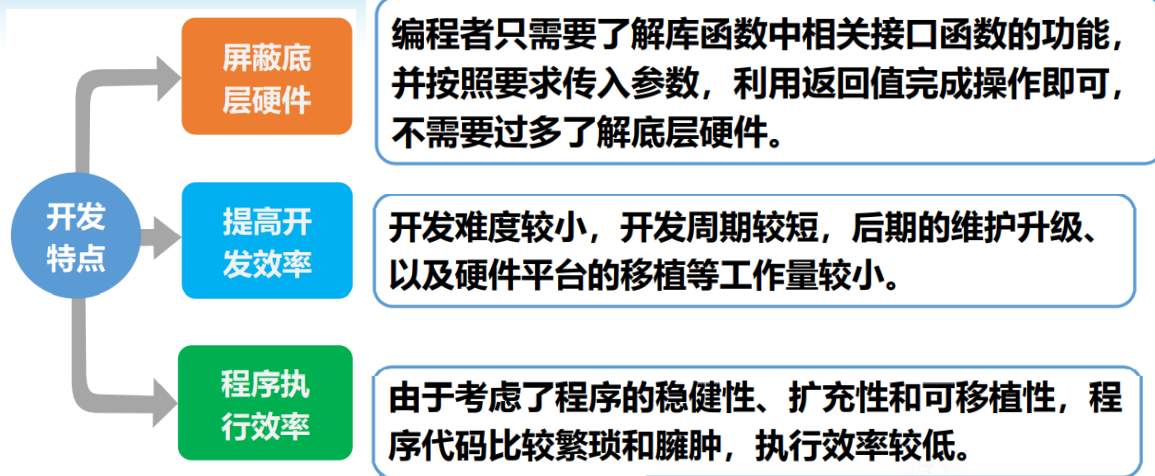
```
uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef* GPIOx,
                                uint16_t GPIO_Pin);
uint16_t GPIO_ReadOutputData(GPIO_TypeDef* GPIOx);
```

4个设置输出电平函数:

```
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin); //BSRRL
void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin); //BSRRH
void GPIO_WriteBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, BitAction
                  BitVal); //BSRR
void GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal); //ODR
```



基于库函数的程序 开发方式的特点



GPIO典型应用步骤

使用库函数实现片上外设的控制，一般需要以下步骤：

- 1) 使能相应片上外设的时钟 (非常重要)，涉及到的文件 **主要函数**：
例如：RCC_APB2PeriphClockCmd (RCC_APB2Periph_USART1, ENABLE)
RCC_AHB1PeriphClockCmd (RCC_AHB1Periph_GPIOD, ENABLE)
- 2) 设置对应于片上外设使用的GPIO引脚初始化工作。
- 3) 如果使用复用功能，需要单独设置每一个GPIO引脚的复用功能。
- 4) 在应用程序中读取引脚状态、控制引脚输出状态或使用复用功能完成特定功能。



GPIO模块的使用步骤

- 1 使能时钟**
使能引脚所属端口的系统总线时钟 (AHB) : 调用函数 `_HAL_RCC_GPIOx_CLK_ENABLE`
- 2 设置参数**
利用引脚初始化类型 `GPIO_InitTypeDef` 定义结构体变量, 根据具体需求配置成员变量: `Pin`、`Mode`、`Pull`、`Speed`、`Alternate`
- 3 配置引脚**
调用初始化函数 `HAL_GPIO_Init` 完成引脚配置, 将配置参数写入对应的硬件寄存器
- 4 控制引脚**
使用对应的控制函数完成引脚的控制: 函数 `HAL_GPIO_ReadPin` 读取引脚状态; 函数 `HAL_GPIO_WritePin` 设置引脚电平





STM32F4的GPIO的开发步骤

第1步，定义GPIO初始化结构体变量。

```
GPIO_InitTypeDef GPIO_InitStructure;
```

第2步，使能GPIO时钟。使能GPIOH

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOH, ENABLE );
```

第3步，填充GPIO初始化结构体的成员变量。

```
GPIO_InitStructure.GPIO_Pin = GPIO_PIN_12|GPIO_PIN_13; //GPIOH12\13
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;           //输出
GPIO_InitStructure.GPIO_OType = GPIO_OType_OD;          //开漏输出
GPIO_InitStructure.GPIO_Pull = GPIO_PuPd_NOPULL;         //没有上拉/下拉电阻
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;       //速度为100MHz
```

第4步，调用GPIO初始化函数。

```
GPIO_Init(GPIOH, &GPIO_InitStructure);                //调用初始化函数
```

第5步，调用GPIO的功能函数。

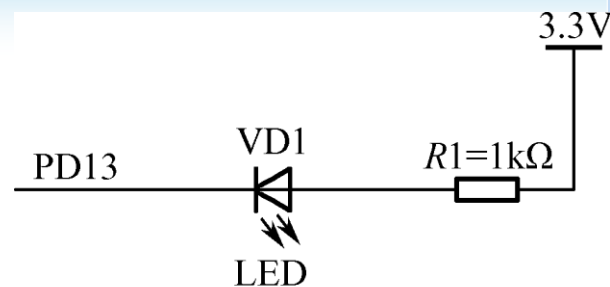
```
GPIO_SetBits(GPIOH,GPIO_PIN_12 |GPIO_PIN_13);          //同时点亮LED
GPIO_ResetBits(GPIOH,GPIO_PIN_12 |GPIO_PIN_13);         //同时熄灭LED
```

GPIO输出应用实例1

要求：控制LED灯，以1s为周期进行闪烁。

LED灯阴极连接GPIO引脚，阳极通过一个限流电阻连接到电源。在3.3V电压驱动下，限流电阻大小为330~1000Ω。

当PD13被置为低电平时，LED灯VD1亮；
当PD13被置为高电平时，LED灯VD1灭。



编程要点

- (1) 使能GPIO时钟。调用函数RCC_AHB1PeriphClockCmd()。不同的外设调用的时钟使能函数可能不一样。
- (2) 初始化GPIO模式。填充GPIO初始化结构体的成员变量,调用函数GPIO_Init()。
- (3) 操作GPIO，设置引脚输出状态。调用函数GPIO_SetBits();或GPIO_ResetBits()或GPIO_ToggleBits()。

```
#include "stm32f4xx.h"

int main( void )
{  uint32_t  i;

    /* LED 端口初始化 */
    GPIO_InitTypeDef GPIO_InitStructure;    // 定义GPIO_InitTypeDef类型的结构体
    /*开启LED相关的GPIOD外设时钟*/
    RCC_AHB1PeriphClockCmd (RCC_AHB1Periph_GPIOD, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;        //对PD13引脚进行设置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;     //选择输出
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;    //推挽输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;      //设置引脚为上拉模式*/
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;  //设置引脚速率为2MHz
    /*调用库函数，使用上面配置的GPIO_InitStructure初始化GPIO*/
    GPIO_Init(GPIOD, &GPIO_InitStructure);
    /* 控制LED灯 */

    while (1)
    {
        GPIO_ResetBits(GPIOD, GPIO_Pin_13);  // 亮
        for( i=0; i<6000000; i++ ) ;
        GPIO_SetBits(GPIOD, GPIO_Pin_13);    // 灭
        for( i=0; i<6000000; i++ ) ;
    }
}
```

```
#include "stm32f4xx.h"

int main( void )
{  uint32_t  i;

    /* LED 端口初始化 */
    GPIO_InitTypeDef GPIO_InitStructure;    // 定义GPIO_InitTypeDef类型的结构体
    /*开启LED相关的GPIOD外设时钟*/
    RCC_AHB1PeriphClockCmd (RCC_AHB1Periph_GPIOD, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;        //对PD13引脚进行设置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;     //选择输出
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;    //推挽输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;      //设置引脚为上拉模式*/
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;  //设置引脚速率为2MHz
    /*调用库函数，使用上面配置的GPIO_InitStructure初始化GPIO*/
    GPIO_Init(GPIOD, &GPIO_InitStructure);
    /* 控制LED灯 */

    while (1)
    {
        GPIO_ResetBits(GPIOD, GPIO_Pin_13);  // 亮
        delay(6000000);
        GPIO_SetBits(GPIOD, GPIO_Pin_13);    // 灭
        delay(6000000);
    }
}
```

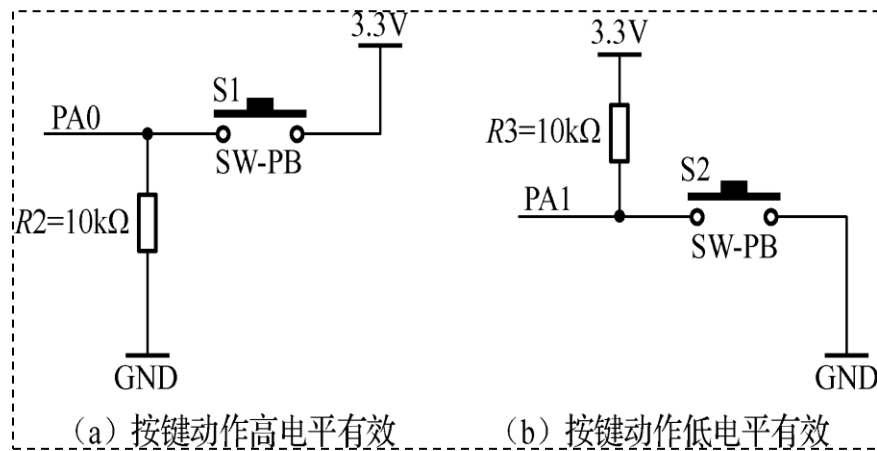


GPIO输入应用实例2

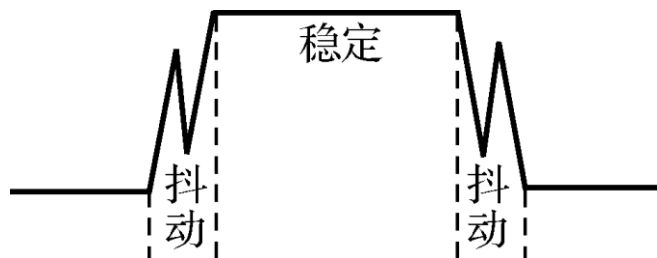
要求：使用独立按键S1和S2控制LED灯，按下S1点亮LED灯，按下S2熄灭LED灯。

图（a）中，独立按键有效按键动作检测电平是**高电平**。

图（b）中，独立按键有效按键动作检测电平是**低电平**。



为了保证按键检测的正确性，需要在程序中去抖动处理，防止抖动对检测的干扰。



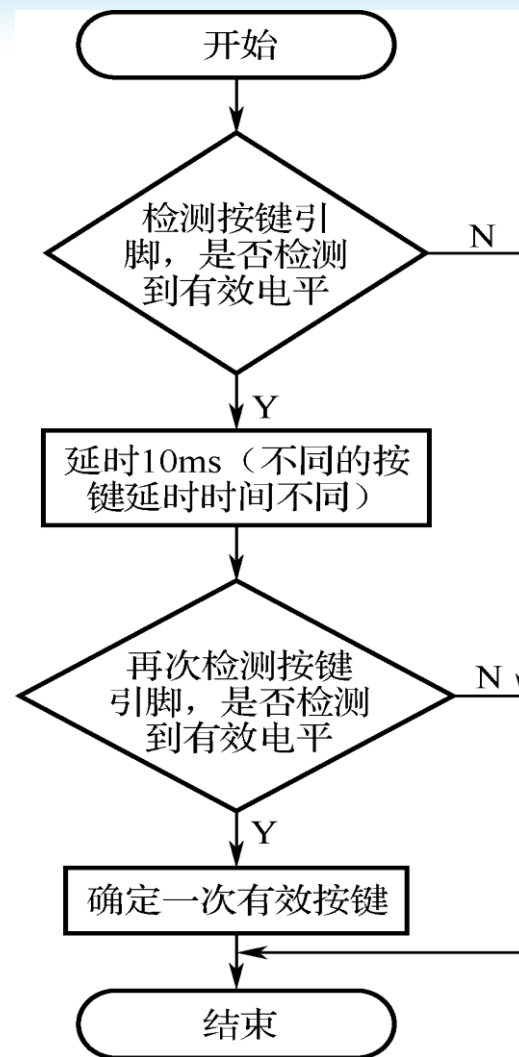
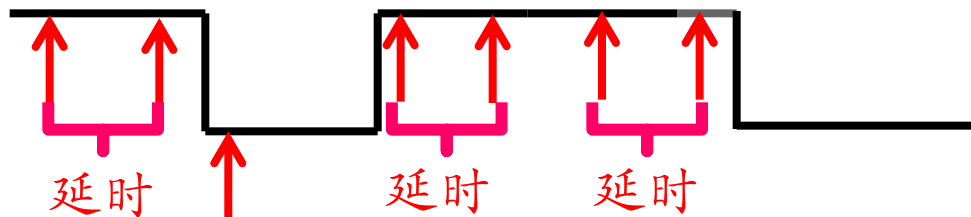
机械按键抖动示意图



独立按键软件扫描方法，需要在程序运行过程中**循环或定时检测**按键连接的引脚，

- ①在首次检测到按键有效电平，
- ②**延时10ms**（不同按键延时不同）后，
- ③再检测一次引脚电平，如能再次检测到有效电平，**则是一次有效按键动作**，**反之则认为**是误操作。

如果按键检测有效电平为高电平



一般独立按键检测程序流程图



编程要点

- (1) 使能GPIO时钟。调用函数RCC_AHB1PeriphClockCmd()。
- (2) 初始化GPIO模式。调用函数GPIO_Init()。
- (3) 操作GPIO，读取引脚状态。调用函数GPIO_ReadInputDataBit();

在main函数中完成如下功能。

- (1) 初始化SysTick。
- (2) 初始化LED灯和按键的GPIO引脚。
- (3) 在无限循环中扫描按键，并在检测到有效按键动作后，控制LED灯亮或灭。

```
int main(void)
{
    delay_init();           //初始化SysTick，用于延时
    LED_Config();           //初始化LED灯的GPIO引脚
    Key_Config();           //初始化按键的GPIO引脚
    /*按键控制LED灯*/
    while (1)               /*按键S1扫描判别，高电平有效*/
    {
        if(Key_Scan(GPIOA,GPIO_Pin_0,1) == KEY_ON)
        {
            LED_ON;          //点亮LED
        } /*按键S2扫描判别，低电平有效*/
        if(Key_Scan(GPIOA,GPIO_Pin_1,0) == KEY_ON)
        {
            LED_OFF;         //熄灭LED
        }
    }
}
```



按键的GPIO引脚初始化

按键的GPIO引脚PA0和PA1的初始化:

- (1) 输入模式;
- (2) 上拉/下拉电阻断开。

```
void Key_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    /*开启按键GPIO口的时钟*/
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE)

    /*选择按键的引脚*/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;

    /*设置引脚为输入模式*/
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;

    /*设置引脚不上拉也不下拉*/
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

    /*使用上面的结构体初始化按键*/
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```



LED灯的GPIO引脚初始化

```
void LED_Config(void)
{
    /*定义一个GPIO_InitTypeDef类型的结构体*/
    GPIO_InitTypeDef GPIO_InitStructure;
    /*开启LED相关的GPIO外设时钟*/ 第一步
    RCC_AHB1PeriphClockCmd (RCC_AHB1Periph_GPIOD, ENABLE);
    /*选择要控制的GPIO引脚*/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
    /*设置引脚模式为输出模式*/
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    /*设置引脚的输出类型为推挽输出*/
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    /*设置引脚为上拉式*/
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    /*设置引脚速率为2MHz */
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
    /*调用库函数，使用上面配置的GPIO_InitStructure初始化GPIO*/
    GPIO_Init(GPIOD, &GPIO_InitStructure); // 第二步
    /*打开LED灯 */
    LED_ON;
}
```

```
#define LED_OFF    GPIO_SetBits(GPIOD, GPIO_Pin_13);
#define LED_ON     GPIO_ResetBits(GPIOD, GPIO_Pin_13);
```



按键扫描程序

```
uint8_t Key_Scan(GPIO_TypeDef* GPIOx,uint16_t GPIO_Pin,uint8_t Key_Lvl)
{
    /*检测是否有按键按下*/
    if(GPIO_ReadInputDataBit(GPIOx,GPIO_Pin) == Key_Lvl )//第一次检测电平
    {
        delay_ms(10);    //去抖动
        if(GPIO_ReadInputDataBit(GPIOx,GPIO_Pin) == Key_Lvl)//第二次检测电平
            return KEY_ON;    //确认有效按键动作返回
        else
            return KEY_OFF;    //无有效按键动作返回
    }
    else
        return KEY_OFF;    //无有效按键动作返回
}
```

```
#define KEY_OFF    0;
#define KEY_ON    1;
```



头文件

```
#include "stm32f4xx.h"
#include "bsp_led.h"
#include "bsp_key.h"
#include "delay.h"
```

// delay.h

```
#ifndef __DELAY_H
#define __DELAY_H

#include "stm32f4xx.h"

void delay_init(void);
void delay_ms(uint32_t nTime);

#endif
```

// delay.c

```
#include "stm32f4xx.h"
#include "delay.h"
volatile uint32_t TimingDelay;
void SysTick_Handler(void)
{
    if (TimingDelay != 0)
    { TimingDelay--;
    }
}

void delay_init(void)
{ if (SysTick_Config(SystemCoreClock / 1000))
  {
    while (1);
  }
}

void delay_ms(uint32_t nTime)
{
    TimingDelay = nTime;
    while(TimingDelay != 0);
}
```



```
// bsp_key.h
#ifndef __BSP_KEY_H
#define __BSP_KEY_H
#include "stm32f4xx.h"

#define KEY_ON    1
#define KEY_OFF   0

void Key_Config(void);

uint8_t Key_Scan(GPIO_TypeDef* GPIOx,uint16_t GPIO_Pin,uint8_t Key_Lvl);

#endif /* __LED_H */
```

bsp_key.c文件中包含两个函数:

```
void Key_Config(void)
Key_Scan ( GPIO_TypeDef* GPIOx,uint16_t GPIO_Pin,uint8_t Key_Lvl );
```



```
//bsp_led.h
```

```
#ifndef __BSP_LED_H
```

```
#define __BSP_LED_H
```

```
#include "stm32f4xx.h"
```

```
#define LED_PIN GPIO_Pin_13
```

```
#define LED_GPIO_PORT GPIOC
```

```
#define LED_GPIO_CLK RCC_AHB1Periph_GPIOC
```

```
#define LED_OFF GPIO_SetBits(LED_GPIO_PORT, LED_PIN )
```

```
#define LED_ON GPIO_ResetBits(LED_GPIO_PORT, LED_PIN )
```

```
#define LED_TOGGLE GPIO_ToggleBits(LED_GPIO_PORT, LED_PIN )
```

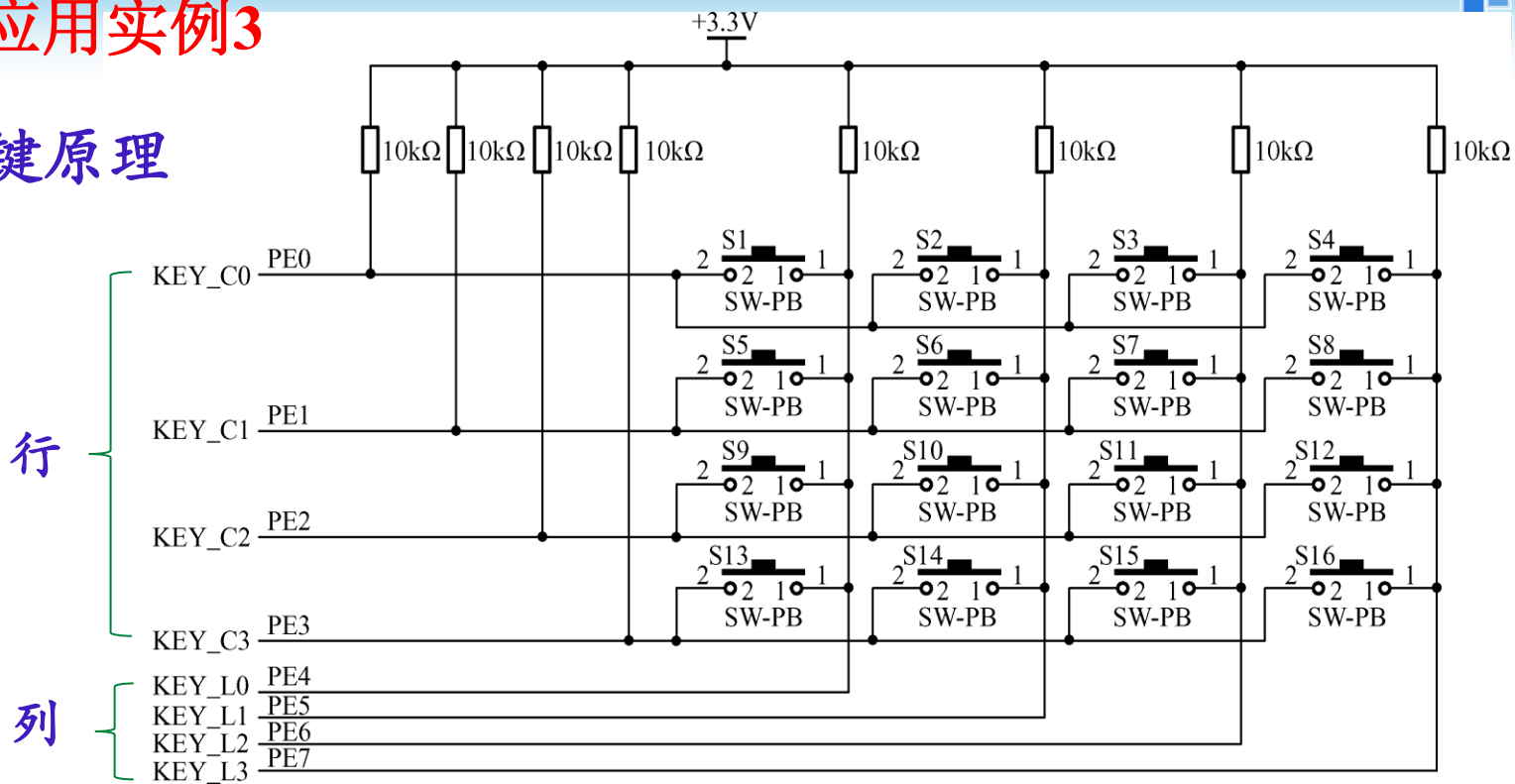
```
void LED_Config(void);
```

```
#endif /* __BSP_LED_H */
```

```
#define LED_TOGGLE GPIO_ToggleBits(LED_GPIO_PORT,LED_PIN)
```

矩阵按键应用实例3

✓ 矩阵按键原理



4×4矩阵按键电路原理图

1) 逐行扫描

通过在矩阵按键的每一条行线上轮流输出低电平，检测矩阵按键的列线，当检测到的列线不全为高电平的时候，说明有按键按下。然后，根据当前输出低电平的行号和检测到低电平的列号组合，判断是哪一个按键被按下。



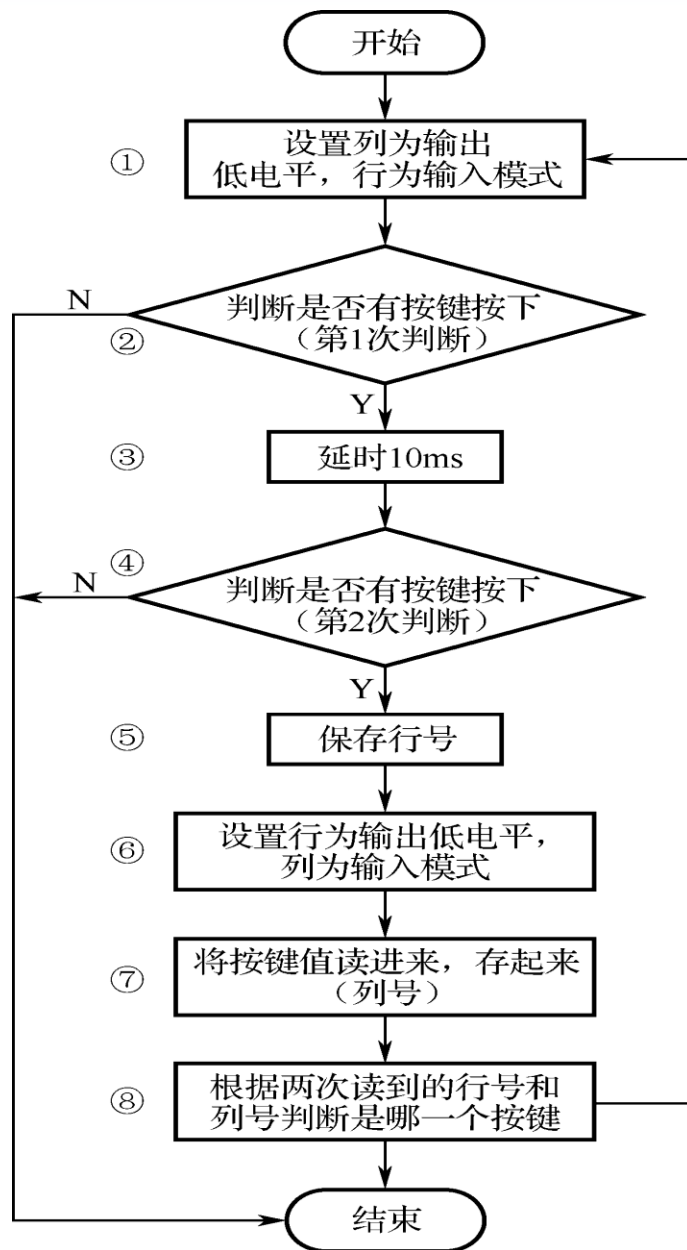
2) 行列扫描

首先，在全部**行线**上输出低电平，检测矩阵按键的列线，当检测到的列线不全为高电平的时候，说明有按键按下，并判断是哪一列有按键按下。

然后，反过来，在全部**列线**上输出低电平，检测矩阵按键的行线，当检测到的行线不全为高电平的时候，说明有按键按下，并判断是哪一行有按键按下。

最后，根据检测到的行号和检测的列号组合，以判断是哪一个按键被按下。

行列扫描法的程序流程图





✓ 矩阵按键程序实现

主程序

要求：根据矩阵原理，编写矩阵按键应用程序，轮询S1~S16按键动作，若检测到S15（15号）按键按下，则反转LED灯。

编程要点

1) 使能LED灯和矩阵按键的GPIO时钟。调用函数：

RCC_AHB1PeriphClockCmd();

2) 初始化LED灯和矩阵按键的GPIO模式。调用函数：

GPIO_Init();

3) 编写矩阵按键扫描程序。

```
#include "stm32f4xx.h"
#include "bsp_led.h"
#include "bsp_key.h"
#include "delay.h"
u8 key_value;
int main(void)
{
    delay_init();           //初始化延时函数
    LED_Config();           //初始化LED灯引脚
    Matrix_Key_Config();     //初始化矩阵按键引脚
    /*轮询矩阵按键状态，若15号（S15）
    按键按下，则反转LED灯*/
    while(1)
    {
        key_value=Matrix_Key_Scan();
        if( key_value !=0xff)
        { /*如果检测到的按键是S15，
            则将LED灯状态反转*/
            if(key_value==15
                LED_TOGGLE; //反转LED灯状态
            }
        }
    }
}
```



1) 矩阵按键GPIO初始化

```
void Matrix_Key_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE); /*使能GPIO的时钟*/

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3|
    GPIO_Pin_4|GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7; /*选择按键的引脚*/

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; /*设置引脚模式为输出模式*/

    GPIO_InitStructure.GPIO_OType = GPIO_OType_OD; /*设置引脚的输出类型为开漏输出*/

    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL; /*设置引脚模式为不上拉下拉模式*/

    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz; /*设置引脚输出速度为25MHz*/

    GPIO_Init(GPIOE, &GPIO_InitStructure); /*使用上面的结构体初始化按键*/
}
```



矩阵按键扫描程序

uint8_t **Matrix_Key_Scan(void)**

```
{ u8 hang, lie, key_value;
  GPIO_InitTypeDef GPIO_InitStructure;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4|GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_OD;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;
  GPIO_Init(GPIOE, &GPIO_InitStructure);
  GPIO_ResetBits(GPIOD,GPIO_Pin_4|GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_Init(GPIOE, &GPIO_InitStructure);
  if((GPIO_ReadInputData(GPIOE)&0x0f) != 0x0f) //第一次检测按键是否按下
  {
    delay_ms(100); //
    if((GPIO_ReadInputData(GPIOE)&0x0f) != 0x0f) //第二次检测按键是否按下
    {
      hang=GPIO_ReadInputData(GPIOE)&0x0f; //读取行号
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4|GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
      GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
      GPIO_Init(GPIOE, &GPIO_InitStructure);
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
      GPIO_InitStructure.GPIO_OType = GPIO_OType_OD;
      GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;
      GPIO_Init(GPIOE, &GPIO_InitStructure);
      GPIO_ResetBits (GPIOE,GPIO_Pin_0|GPIO_Pin_1|GPIO_Pin_2|GPIO_Pin_3);
```

列设为
开漏输出模式

行设为输入模式

列
设为
输入

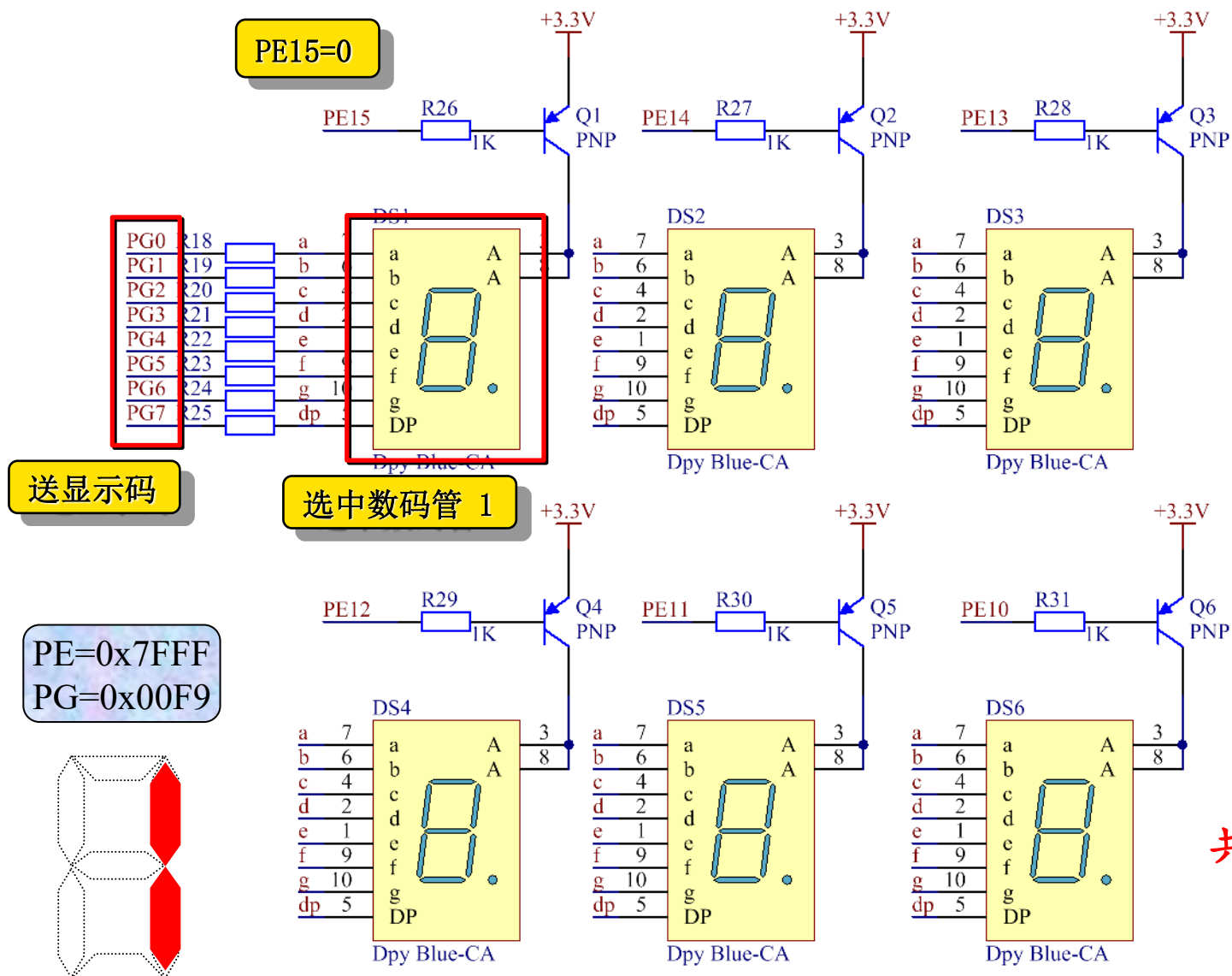
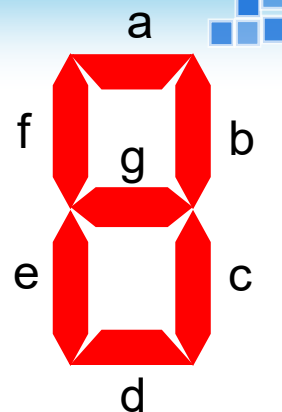
行
设为
输出



```
if((GPIO_ReadInputData(GPIOE)&0xf0)!=0xf0)
{
    lie=GPIO_ReadInputData(GPIOE)&0xf0; //读取列号
    switch(hang | lie) //行号（高4位）列号（低4位）
    {
        case 0x77: key_value= 16; break; //0111 0111
        case 0x7b: key_value= 15; break; //0111 1011
        case 0x7d: key_value= 14; break; //0111 1101
        case 0x7e: key_value= 13; break; //0111 1110
        case 0xb7: key_value= 12; break; //1011 0111
        case 0xbb: key_value= 11; break; //1011 1011
        case 0xbd: key_value= 10; break; //1011 1101
        case 0xbe: key_value= 9; break; //1011 1110
        case 0xd7: key_value= 8; break; //1101 0111
        case 0xdb: key_value= 7; break; //1101 1011
        case 0xdd: key_value= 6; break; //1101 1101
        case 0xde: key_value= 5; break; //1101 1110
        case 0xe7: key_value= 4; break; //1110 0111
        case 0xeb: key_value= 3; break; //1110 1011
        case 0xed: key_value= 2; break; //1110 1101
        case 0xee: key_value= 1; break; //1110 1110
        default: key_value= 0xff; break; }
    }
    else key_value= 0xff;
} else key_value= 0xff;
} else key_value= 0xff;
return key_value;
}
```

数码管动态显示应用实例4

STM32F103ZET6



共阳数码管



数码管动态显示其项目任务是显示“两个数码管（小时）两个数码管（分钟）两个数码管（秒）”；主程序赋值的三个变量：“**hour**”、“**minute**”、“**second**”的值显示在六位数码管上，并在小时个位和分钟个位数字下面显示一个点。

```
#include "stm32f10x.h"
#include "LED.h"
#include "beepkey.h"
#include "dsgshow.h"
#include "EXTI.H"

u8 smgduan[11]={0xc0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x82, 0xf8, 0x80, 0x90};
u16 smgwei[6]={0x7fff, 0xbfff, 0xdfff, 0xefff, 0xf7ff, 0xfbff};
u8 hour, minute, second;
int main()
{
    hour=9; minute=6; second=18;
    DsgShowInit();
    EXTIInti();
    while(1)
    {
        DsgShowTime();
    }
}
```



DsgShowInit()函数负责数码管显示初始化

```
void DsgShowInit() //配置引脚PE15~10 (选数码管), PG7~0 (小数点+数码管七段)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE | RCC_APB2Periph_GPIOG,
    ENABLE); //Cortex-M3处理器

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15 | GPIO_Pin_14 | GPIO_Pin_13 |
    GPIO_Pin_12 | GPIO_Pin_11 | GPIO_Pin_10 | GPIO_Pin_9 | GPIO_Pin_8;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOE, &GPIO_InitStructure); //配置四位引脚PE15~10输出

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOG, &GPIO_InitStructure); //配置八位引脚PG7~0输出
}
```

DsgShowTime()函数完成“时、分、秒”的显示

```
void DsgShowTime()
{
    ul6 j;
    GPIO_Write(GPIOE, smgwei[0]); //选DS1数码管（最高位）
    GPIO_Write(GPIOG, smgduan[hour/10]); //小时的十位数值
    for(j=0; j<400; j++);
    GPIO_Write(GPIOE, smgwei[1]);
    GPIO_Write(GPIOG, (smgduan[hour%10])&0xff7f); //小时的个位数值和小数点
    for(j=0; j<400; j++);
    GPIO_Write(GPIOE, smgwei[2]);
    GPIO_Write(GPIOG, smgduan[minute/10]); //分的十位数值
    for(j=0; j<400; j++);
    GPIO_Write(GPIOE, smgwei[3]);
    GPIO_Write(GPIOG, (smgduan[minute%10])&0xff7f); //分的个位数值和小数点
    for(j=0; j<400; j++);
    GPIO_Write(GPIOE, smgwei[4]);
    GPIO_Write(GPIOG, smgduan[second/10]); //秒的十位数值
    for(j=0; j<400; j++);
    GPIO_Write(GPIOE, smgwei[5]); //选DS6数码管（最低位）
    GPIO_Write(GPIOG, smgduan[second%10]); //秒的个位数值
    for(j=0; j<400; j++);
}
```



基于ARM微处理器的开发和应用

END