STM32Fxxx 驱动库软件使用手册

版权

版本

根据市场和用户的需要，本产品会作相关的功能调整以及技术改进，另外，由于编写人员各种原因都可能引起说明书内容修订，恕不另行通知。下面表格显示本产品使用手册在不同时期的修订版本及修订原因说明:

版本	修改内容	完成日期	修订部门
V1.10	正式发布驱动库(适用驱动库版本：V1.10)	2019．3．5	研发部

适用产品型号 ：

序号	类型	订货型号	备注
1	工业控制模块	STM32F103VE、STM32F103ZE(EB8612I)、STM32F107VC(EMB8610I)、STM32F407VE	
2	工业控制板	EMB8600I、EMB8612IA、EMB8616I、 EMB8618I	
3	开发板	STM32F103VE-DK、STM32F107VC-DK STM32F407VE-DK	

特别提醒：本软件只适用于以上产品硬件，并不适用于其它产品。

销售及服务网络：

北京中嵌凌云电子有限公司：

销售电话：18801080298,13220180005　　传真：010-63983650

公司地址：北京市海淀区吴家场路 1 号院 2 号楼 3 单元底 1-1　　邮编：100036

西安中嵌凌云：

销售电话：029-68888268　　传真：029-88772044

公司地址：西安市长安区东长安街 501 号运维国际大厦 8 层 B806 室　　邮编：710199

技术支持：

电　　话：18801080298,　QQ:1104849817,　微信：13399288868

技术支持 QQ 群：3170083　　电子邮件：embedarm@126.com

公司网址：http://www.embedarm.com

目　　录

第一章. 系统软件开发模版文件结构

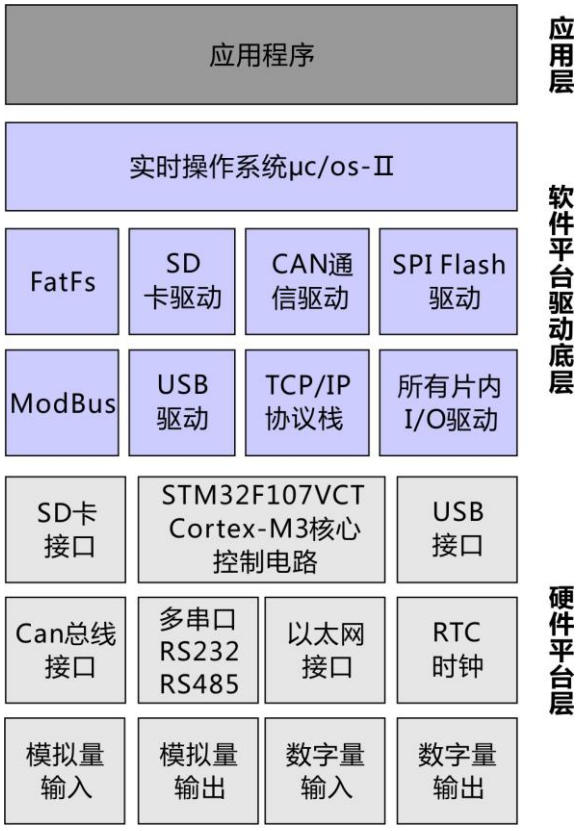
1. 软件开发模版文件结构表

序号	文件夹	文件	描述
1	source 用户程序 用户可修改本目录下程序来完成软件设计	main.c	系统主程序
		UserVar.c/UserVar.h	应用程序全局常量和变量定义文件,用户应用程序的全局常量和变量在这里定义
		ISRHook.c	系统中断处理文件,在这里处理所有中断及驱动库钩子函数
		APP_Init.c	驱动库硬件接口初始化函数:根据板子的配置文件及IO配置文件进行初始化
		TaskIO.c	DI/DO端口/按键/RTC时钟/EEPROM/SPI FLASH测试例程
		TaskUartCAN.c	Uart(RS232/RS485)/CAN通信测试例程
		TaskADC_DAC.c	ADC模拟量采集/DAC模拟量输出测试例程
		TaskPWMClk.c	PWM输出/FCLK脉冲输入/定时器测试例程
		TaskFile.c	SPI FLASH/SD卡/U盘/Nand Flash 文件读写例程(FatFS文件系统)
		TaskModbus.c	Modbus协议通信测试例程
		TaskLWIP.c	LWIP网络通信测试例程
		TaskDTU.c	DTU控制数据收发测试例程
2	config 系统配置 这个目录是整个软件结构配置目录	const.h	常量定义，用户不可更改
		vars.h,vars.c	驱动库全局变量定义，用户不可更改
		config.h	全局配置文件，用户根据自己所应用板子型号选择定义，如应用EMB8600I工控板则需要如下定义:#define PRODUCT_TYPE EMB8600I
		EMB8600I_Config.h EMB8600I_IOConfig.h	工控板EMB8600I配置文件和IO配置文件,用户根据需要自行修改
		EMB8612IA_Config.h EMB8612IA_IOConfig.h	工控板EMB8612IA配置文件和IO配置文件,用户根据需要自行修改
		EMB8616I_Config.h EMB8616I_IOConfig.h	工控板EMB8616I配置文件和IO配置文件,用户根据需要自行修改
		EMB8618I_Config.h EMB8618I_IOConfig.h	工控板EMB8618I配置文件和IO配置文件,用户根据需要自行修改
		STM32F107VC_Config.h STM32F107VC_IOConfig.h	工控模块STM32F107VC配置文件和IO配置文件,用户根据需要自行修改
		STM32F103VE_Config.h STM32F103VE_IOConfig.h	工控模块STM32F103VE配置文件和IO配置文件,用户根据需要自行修改

		STM32F103ZE_Config.h STM32F103ZE_IOConfig.h	工控模块STM32F103ZE配置文件和IO配置文件,用户根据需要自行修改
		STM32F407VE_Config.h STM32F407VE_IOConfig.h	工控模块STM32F407VE配置文件和IO配置文件,用户根据需要自行修改
3	OSConfig 操作系统配置及变量定义目录	os_cfg.h	ucos操作系统配置文件
		app_cfg.h	应用任务配置文件
		Includes.h	操作系统引用头文件
		OSHook.c、OSHook.h	操作系统应用钩子函数处理文件
		OSVar.c、 OSVar.c	操作系统全局变量定义文件
4、	libraries 基础驱动库 注：此文件目录下文件不可更改，由我公司进行维护	sysint.h	系统驱动库头文件
		gpio.h	GPIO通用IO硬件驱动程序头文件
		exit.h	外部中断硬件驱动程序头文件
		rtc.h	RTC 实时时钟硬件驱动程序头文件
		iwdg.h	独立看门狗硬件驱动程序头文件
		uart.h	UART 异步串口(RS232 或 RS485)硬件驱动程序头文件
		i2c.h	I2C 总线硬件驱动程序头文件
		spi.h	SPI 总线硬件驱动程序头文件
		can.h	CAN 总线硬件驱动程序头文件
		timer.h	定时器(PWM/FCLK)硬件驱动程序头文件
		dac.h	DAC 硬件驱动程序头文件
		adc.h	ADC 硬件驱动程序头文件
		flash.h	内部 FLASH 硬件驱动程序头文件
		crc.h	CRC 校验硬件驱动程序头文件
		sd.h	SD卡(SPI)读写硬件驱动程序头文件
		AT45DBXX.h	AT45DBXX系列FLASH读写硬件驱动程序头文件
		W25QXX.h	W25QXX系列FLASH读写硬件驱动程序头文件
		eeprom.h	EEPROM读写硬件驱动程序头文件
		USBHost.h	USB主机硬件驱动程序头文件
		USBDevice.h	USB设备硬件驱动程序头文件
		net.h	网络接口硬件驱动程序头文件
		IAP.h	IAP固件更新驱动程序头文件
		fsmc.h	Fsmc 外部总线驱动程序头文件
		NFlash.h	Nand Flash 读写硬件驱动程序头文件
		Delay.h	延时函数驱动程序头文件
		subfun.h	常用功能函数驱动程序头文件

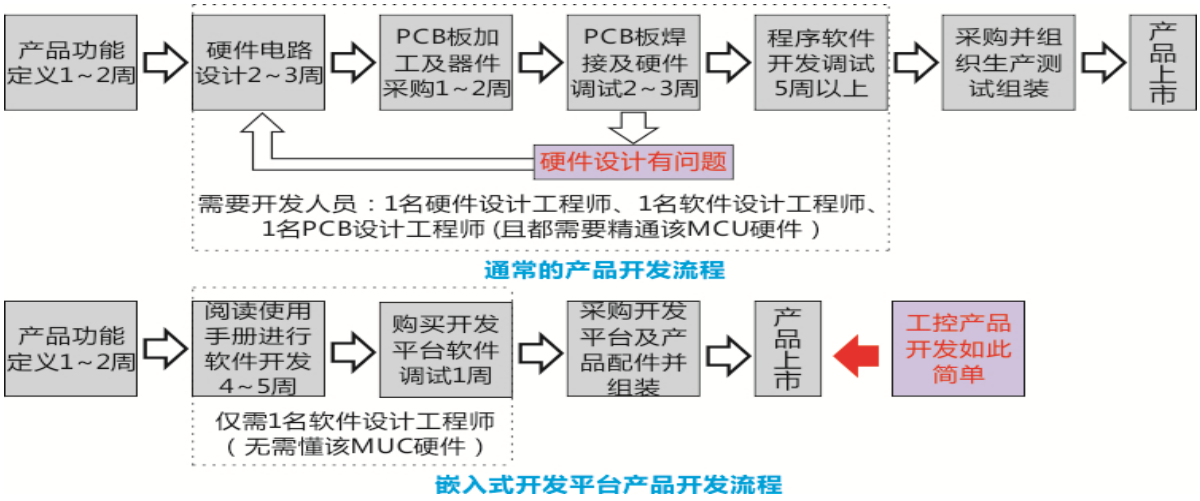
		STM32F107VC_V110.xxxx.lib	STM32F107VC模块驱动库文件
		STM32F103VE_V110.xxxx.lib	STM32F103VE模块驱动库文件
		STM32F103ZE_V110.xxxx.lib	STM32F103ZE模块驱动库文件
		STM32F407ZE_V110.xxxx.lib	STM32F407ZE模块驱动库文件
5	Fatfs 文件系统	ff.c、ff.h	Fatfs文件系统源代码，用户一般不用修改
		diskio.c、diskio.h	Fatfs硬件接口驱动文件，包括SPI FLASH/SD卡/U盘/Nand Flash的文件系统接口驱动；已经做好,客户不要修改
		integer.h	数据类型定义头文件一般不用修改
		ffconf.h	Fatfs文件系统配置文件,客户可以修改
6	init 初始化	inita.s	系统初始化汇编文件
		vectors.s	中断向量汇编文件
7	lwip TCPIP协议栈	lwip.c	网络协议栈源文件
		sys_arch.c sys_arch.h	lwip移植到ucos上接口驱动文件
		ethernetif.c	LWIP网络硬件驱动接口文件
		lwipopts.h	lwip配置文件,可以替代opt.h
		opt.h	lwip默认配置文件
8	ucos-ii 操作系统	ucos-ii.c	ucos-ii源文件
		cpu_a.asm,os_cpu_a.asm， lib_mem_a.asm	ucos-ii汇编移植汇编文件

2. 软件系统文件结构图



开发平台系统框图

3. 软件开发流程比较



第二章 系统配置文件说明

1. const.h说明

整个软件系统及驱动库常量都在const.h这个文件中定义，不允许用户更改该文件，否则出错。要求用户尽可能在编程中应用这里的定义，而保持软件的兼容性。如果用户需要定义自己的常量，请在source目录下UserVars.h中定义。

2. vars.c, vars.h说明

vars.c、vars.h是系统软件及驱动库定义的全局变量，包括配置变量，缓存buf等等。不允许用户更改该文件，否则出错。一般用户无需管理这个两个文件。如果用户需要定义自己的全局变量，请在source目录下UserVars.h，UserVars.c，中定义。

3. config.h说明

这个文件是整个软件平台最全局的配置文件,用户只需要在这里配置工控板型号即可。例如用户购买工控板型号是EMB8600I，则只需要做如下配置即可：#define PRODUCT_TYPE EMB8600I, 其它产品型号请注销掉；这样根据这个配置，文件下面会自动为该工控板匹配相应的模块型号、配置文件和IO配置文件，如下定义：

```
#if (PRODUCT_TYPE == EMB8600I)        // 如果选择产品型号: EMB8600I
#define  MODULE_TYPE  STM32F107VC     // 设置产品应用核心工控模块型号, 对应驱动库
                                          STM32F107VC_xxxx_xxxxxxxx.lib
#include "EMB8600I_Config.h"          // 工控板EMB8600I功能参数配置文件
#include "EMB8600I_IOConfig.h"        // 工控板EMB8600I IO端口配置文件
#endif
```

对于工控板EMB8600I只需修改EMB8600I_Config.h和EMB8600I_IOConfig.h 两个配置文件即可，
其它工控板定义参考此例,详细内容请查看例程的config.h文件.
另外: 请根据是否使用UCOS-II来设置UCOS_II_EN为1或0

4. FLASH及RAM分配文件说明

STM32F107VC_Flash.scat:	工控模块STM32F107VC, FLASH及RAM分配文件
STM32F107VC_FlashIAP.scat:	工控模块STM32F107VC, FLASH及RAM分配文件(编译可生成IAP功能固件)
STM32F103VE_Flash.scat:	工控模块STM32F103VE, FLASH及RAM分配文件
STM32F103VE_FlashIAP.scat:	工控模块STM32F103VE, FLASH及RAM分配文件(编译可生成IAP功能固件)
STM32F103ZE_Flash.scat:	工控模块STM32F103ZE, FLASH及RAM分配文件
STM32F103ZE_FlashIAP.scat:	工控模块STM32F103ZE, FLASH及RAM分配文件(编译可生成IAP功能固件)
STM32F407VE_Flash.scat:	工控模块STM32F107VE, FLASH及RAM分配文件
STM32F407VE_FlashIAP.scat:	工控模块STM32F107VE, FLASH及RAM分配文件(编译可生成IAP功能固件)

请按键"Alt+F7"打开配置界面, 在Linker界面的"Scatter File"下加载相应的FLASH及RAM分配文件, 注意: xxxxxxxxxxx_FlashIAP.scat, 可编译生成IAP功能固件。 用户如果不熟悉scat文件配置请不要修改，使用默认配置。

第三章 基础库函数说明

1. 简介及函数列表

基础库函数是STM32FXXX系列工控模块(板)产品公共的函数库，具有函数精简高效、使用简单、兼容性好等特点，可以使用户快速开发产品。这些函数都在Libraries文件夹下，函数列表如下：

序号	头文件	函数原型	函数说明
1	sysint.h	INT32S Syslib_Init(SYSLIB_PARA *pPara)	系统驱动库参数初始化函数
		INT32S SysLib_Ctrl(INT8U Cmd, INT32U Para)	系统驱动库控制函数
2	gpio.h	INT32S IO_Init(INT32U IOx, INT8U Mode, INT8U Speed)	IO初始化函数
		INT32U IO_Read(INT32U IOx)	读取IO输入值
		void IO_Write(IN32U IOx, INT16U val)	写入IO输出值
		INT32S IO_Ctrl(INT32U IOx, INT8U Cmd, INT32U Para)	IO命令控制
3	exit.h	INT32S EXTI_Init(EXTI_PARA *pPara)	外部IO中断初始化函数
		INT32S EXTI_Ctrl(INT8U id, INT8U Cmd)	外部IO中断使能控制函数
4	rtc.h	INT32S RTC_Init(RTC_PARA *pPara)	RTC初始化函数
		INT32S RTC_Read(RTC_TIME *rtc)	读取时间
		INT32S RTC_Write(RTC_TIME *rtc)	设置时间
		INT32U RTC_Ctrl(INT8U Cmd, INT32U Para, RTC_TIME *rtc)	RTC控制函数
5	iwdg.h	void IWDG_Init(INT32U t)	IWDG看门狗初始化函数
		void IWDG_Ctrl(INT8U Cmd)	IWDG看门狗控制函数
6	uart.h	INT32S Uart_Init(INT8U id, UART_PARA *pPara)	Uart初始化函数
		INT32S Uart_Read(INT8U id, INT8U *p, INT16U len)	读取接收数据
		INT32S Uart_Write(INT8U id, INT8U *p, INT16U len)	发送数据
		INT32S Uart_RecvChar(INT8U id, INT8U *val)	接收一个字节数据
		INT32S Uart_SendChar(INT8U id, INT8U val)	发送一个字节数据
		INT32S Uart_Ctrl(INT8U id, INT8U Cmd, INT32U Para)	UART命令控制
7	i2c.h	INT32S I2C_Init(INT8U I2Cx, I2C_PARA *pPara)	I2C初始化函数
		INT32S I2C_Read(INT8U id, INT16U I2CAddr, INT16U addr, INT8U *p, INT16U len)	I2C读数据函数
		INT32S I2C_Write(INT8U id, INT16U I2CAddr, INT16U addr, INT8U *p, INT16U len)	I2C写数据函数
		INT32S I2C_Ctrl(INT8U id, INT8U Cmd, INT32U Para)	I2C控制函数
8	spi.h	INT32S SPI_Init(INT8U id, SPI_PARA *pPara)	SPI初始化函数
		INT32S SPI_Read(INT8U id, INT8U *p, INT32U len)	SPI读数据函数
		INT32S SPI_Write(INT8U id, INT8U *p, INT32U len)	SPI写数据函数
		INT8U SPI_ReadWriteByte(INT8U id, INT8U val)	SPI读写一个字节函数
		INT32S SPI_Ctrl(INT8U id, INT8U Cmd, INT32U Para)	SPI控制函数
9	can.h	INT32S CAN_Init(INT8U id, CAN_PARA *pPara,	CAN初始化函数

		CAN_FILTER_PARA *pFilter)	
		INT32S CAN_FilterInit(CAN_FILTER_PARA *pFilter)	CAN滤波器初始化函数
		INT32S CAN_Write(INT8U id, CAN_TX_MSG *Para)	CAN写数据函数
		INT32S CAN_Read(INT8U id, CAN_RX_MSG *Para)	CAN读数据函数
		INT32S CAN_Ctrl(INT8U id, INT8U Cmd, INT32U Para)	CAN控制函数
10	adc.h	INT32S ADC_Init(INT8U id, ADC_PARA *pPara)	ADC初始化函数
		INT32S ADC_Read(INT8U id, INT16S *p, INT8U len)	ADC读取函数
		INT32S ADC_Ctrl(INT8U id, INT8U Cmd, INT32U Para)	ADC控制函数
11	dac.h	INT32S DAC_Init (INT8U id, DAC_PARA *pPara)	DAC初始化函数
		void DAC_Write(INT8U id, INT16U val)	DAC写(输出)数据函数
		INT32S DAC_Ctrl(INT8U id, INT8U Cmd, INT32U Para)	DAC控制函数
12	timer.h	INT32S Timer_Init(INT8U id, TIM_PARA *pPara)	定时器初始化函数
		INT32S Timer_Ctrl(INT8U id, INT8U Cmd, TIM_CTRL *pPara)	定时器控制函数
		INT32S PWM_Init(INT8U id, PWM_PARA *pPara)	PWM初始化函数
		INT32S PWM_Ctrl(INT8U id, INT8U Cmd, PWM_CTRL *pPara)	PWM控制函数
		INT32S FCLK_Init(INT8U id, FCLK_PARA *pPara)	FCLK初始化函数
		INT32S FCLK_Ctrl(INT8U id, INT8U Cmd, INT8U Chx)	FCLK控制函数
		INT32S FCLK_Read(INT8U id, INT8U Cmd, INT8U Chx, INT32U *p, INT16U len, INT16U TimeOut)	FCLK读取函数
13	bkp.h	void BKP_Init(void)	备份寄存器初始化函数
		INT32S BKP_Write(INT8U id, INT16U *p, INT16U len)	备份寄存器写入函数
		INT32S BKP_Read(INT8U id, INT16U *p, INT16U len)	备份寄存器读取函数
14	flash.h	INT32S Flash_Write(INT32U StartAddr, INT8U *p, INT16U len)	向FLASH写数据
		INT32S Flash_Read(INT32U StartAddr, INT8U *p, INT16U len)	读取FLASH写数据
		INT32S Flash_Ctrl(INT8U Cmd, INT32U Para)	FLASH控制函数
15	fsmc.h	INT32S FSMC_Init(FSMC_PARA *pPara)	FSMC初始化函数
		INT16U FSMC_Read(INT16U addr)	读外部总线函数
		void FSMC_Write(INT16U addr, INT16U val)	写外部总线函数
		INT32S FSMC_Ctrl(INT8U Cmd, INT32U Para)	FSMC控制函数
16	crc.h	INT32U CRC32(INT32U *p, INT16U len)	计算32bit CRC函数
		INT16U CRC16(INT8U *p, INT16U len)	CRC16计算函数
		INT16U CRC16_2(INT8U *p1, INT16U len1, INT8U *p2, INT16U len2)	2组数CRC16计算函数
		INT8U CRC8(INT8U *p, INT16U len)	CRC8计算函数
		INT32U CRC_Ctrl(INT8U Cmd, INT32U Para)	CRC控制函数
17	delay.h	void Delay_us(INT16U val)	微妙延时函数
		void Delay_ms(INT16U val)	毫秒延时函数
		void Delay_s(INT16U val)	秒延时函数
		INT32S USBD_Init(USBD_PARA *pPara)	USB设备初始化函数

18	USBDevice.h	INT32S USBD_Read(INT8U *p, INT16U len)	USB设备读数据函数
		INT32S USBD_Write(INT8U *p, INT16U len)	USB设备写数据函数
		INT32S USBD_Ctrl(INT8U Cmd, INT32U Para)	USB设备控制函数
19	USBHost.h	INT32S USBH_Init(USBH_PARA *pPara)	USB主机初始化函数
		INT32S USBH_Ctrl(INT8U Cmd, INT32U Para)	USB主机控制函数
		INT8U UDisk_Read(INT8U pdrv, INT8U *buff, INT32U sector, INT8U count)	USB主机读取U盘数据函数
		INT8U UDisk_Write(INT8U pdrv, INT8U *buff, INT32U sector, INT8U count)	USB主机写入U盘数据函数
20	net.h	INT32S NET_Init(NET_PARA *pPara)	网络初始化函数
		INT32S NET_Write(INT32U len)	发送数据函数
		INT32S NET_Read(INT32U *p, INT16U *len)	接收数据函数
		INT32S NET_Ctrl(INT8U Cmd, INT32U Para)	网络控制函数
21	IAP.h	INT32S IAP_Init(IAP_PARA *Para)	IAP初始化函数
		INT32S IAP_Write(INT8U id, INT8U *p, INT32U addr, INT32U len)	IAP固件数据写入函数
		INT32S IAP_Ctrl(INT8U id, INT8U Cmd, INT32U Para)	IAP控制函数
22	eeprom.h	INT32S EEPROM_Init(EEPROM_PARA *pPara)	EEPROM初始化函数
		INT32S EEPROM_Read(INT16U addr, INT8U *p, INT16U len)	EEPROM读数据函数
		INT32S EEPROM_Write(INT16U addr, INT8U *p, INT16U len)	EEPROM写数据函数
23	W25QXX.h	INT32S W25QXX_Init(W25QXX_PARA *pPara)	W25QXX初始化函数
		INT32S W25QXX_Write(INT8U *p, INT32U addr, INT16U len)	W25QXX任意地址写数据函数
		INT32S W25QXX_WritePage(INT8U *p, INT32U addr, INT16U len)	W25QXX按页写数据函数
		INT32U W25QXX_Read(INT8U *p, INT32U addr, INT32U len)	W25QXX任意地址读数据函数
		INT32S W25QXX_WriteSector(INT8U *p, INT32U sector, INT32U count)	W25QXX按扇区写数据函数
		INT32S W25QXX_ReadSector(INT8U *p, INT32U sector, INT32U count)	W25QXX按扇区读数据函数
		INT32S W25QXX_Ctrl(INT8U Cmd, INT32U Para)	W25QXX控制函数
24	AT45DBXX.h	INT32S AT45DBXX_Init(AT45DBXX_PARA *pPara)	AT45DBXX初始化函数
		INT32S AT45DBXX_WritePage(INT8U *p, INT32U page, INT32U count)	AT45DBXX按页写数据函数
		INT32S AT45DBXX_ReadPage(INT8U *p, INT32U page, INT32U count)	AT45DBXX按页读数据函数
		INT32S AT45DBXX_Write(INT8U *p, INT32U addr, INT32U len)	AT45DBXX任意地址写数据函数
		INT32S AT45DBXX_Read(INT8U *p, INT32U addr, INT32U len)	AT45DBXX任意地址读数据函数
		INT32S AT45DBXX_Ctrl(INT8U Cmd, INT32U Para)	AT45DBXX控制函数

25	NFlash.h	INT32S NFlash_Init(NFLASH_PARA *pPara)	Nand flash初始化函数
		INT32S NFlash_ReadSector(INT8U *p, INT32U sector, INT32U count)	Nand flash读扇区数据函数
		INT32S NFlash_WriteSector(INT8U *p, INT32U sector, INT32U count)	Nand flash写扇区数据函数
		INT32U NFlash_Ctrl(INT8U Cmd, INT32U Para)	Nand flash控制函数
26	sd.h	INT32S SD_Init(SD_PARA *pPara)	SD卡初始化函数
		INT8U SD_Read(INT8U *buff, INT32U sector, INT8U count)	SD卡读数据函数
		INT8U SD_Write(INT8U *buff, INT32U sector, INT8U count)	SD卡写数据函数
		INT32S SD_Ctrl(INT8U Cmd, INT32U Para);	SD卡控制函数
27	modbus.h	INT32S Modbus_ReadCoils(INT8U ch, INT8U id, INT16U addr, INT16U len, INT8U *p, INT16U TimeOut)	Modbus主机读取线圈函数
		INT32S Modbus_ReadDisInput(INT8U ch, INT8U id, INT16U addr, INT16U len, INT8U *p, INT16U TimeOut)	Modbus主机读取离散输入量函数
		INT32S Modbus_ReadHoldReg(INT8U ch, INT8U id, INT16U addr, INT16U len, INT16U *p, INT16U TimeOut)	Modbus主机读取保持寄存器函数
		INT32S Modbus_ReadInputReg(INT8U ch, INT8U id, INT16U addr, INT16U len, INT16U *p, INT16U TimeOut)	Modbus主机读取输入寄存器函数
		INT32S Modbus_WriteSingleCoil(INT8U ch, INT8U id, INT16U addr, INT8U val, INT16U TimeOut)	Modbus主机写单个线圈函数
		INT32S Modbus_WriteSingleReg(INT8U ch, INT8U id, INT16U addr, INT16U val, INT16U TimeOut)	Modbus主机写单个保持寄存器函数
		INT32S Modbus_WriteMulCoils(INT8U ch, INT8U id, INT16U addr, INT16U len, INT8U *p, INT16U TimeOut)	Modbus主机写多个线圈函数
		INT32S Modbus_WriteMulReg(INT8U ch, INT8U id, INT16U addr, INT16U len, INT16U *p, INT16U TimeOut)	Modbus主机写多个保持寄存器函数
		INT32S Modbus_ReadWriteMulReg(INT8U ch, INT8U id, INT16U waddr, INT16U wlen, INT16U raddr, INT16U rlen, INT16U *p, INT16U TimeOut)	Modbus主机读和写多个保持寄存器函数
		INT32S Modbus_Proc(INT8U ch, INT8U id, INT8U *p, INT16U len, MODBUS_PARA *pPara)	Modbus从机通信指令解析函数
28	subfun.h	INT16U GetStringLength(INT8U *p)	取得字符串长度
		INT32S MatchStr(INT8U *pSrc, INT8U *pDst, INT8U m, INT16U *n)	字符串匹配搜索
		INT32S StringComp(INT8U *str1,INT8U *str2, INT16U len)	字符串对比
		void IntToStr(INT8U *pDst, INT32U val)	32位整型数转换为字符串(十进制)
		INT32U StrToInt(INT8U *pSrc)	将字符串转换为32位整型数

		INT8U AsciiToHex(INT8U val)	将ASCII码转换为16进制数
		void IPToStr(INT8U *pDst, INT8U ip[])	将4字节IP转换为点间隔的IP字符串
		INT32S StrToIP(INT8U ip[], INT8U *pSrc)	将点间隔的IP字符串转换为4字节IP
		void HexToStr(INT8U val, INT8U *pDst)	将HEX数转换为2个字符串
		void StrCopy(INT8U *pDst, INT8U *pSrc)	单字符串拷贝函数
		void StrCopy2(INT8U *pDst, INT8U *pSrc1, INT8U *pSrc2)	2字符串拷贝函数
		void StrCopy3(INT8U *pDst, INT8U *pSrc1, INT8U *pSrc2, INT8U *pSrc3)	3字符串拷贝函数
		void StrCopy4(INT8U *pDst, INT8U *pSrc1, INT8U *pSrc2, INT8U *pSrc3, INT8U *pSrc4)	4 字符串拷贝函数
		void StrCopy5(INT8U *pDst, INT8U *pSrc1, INT8U *pSrc2, INT8U *pSrc3, INT8U *pSrc4, INT8U *pSrc5)	5 字符串拷贝函数

2. 驱动库硬件初始化函数

函数原型	void API_Init(void)
函数功能	驱动库硬件初始化函数
入口参数	无
返回参数	无

函数体
去掉了条件编译
 	
具体看APP_Init.c文件
 	
这个函数请用户不要修改
	void API_Init (void)
	{
	SysLib_APPInit(); // 系统驱动库应用初始化函数;
	UserVars_Init(); // 用户全局变量初始化
	Uart_APPInit(); // UART1-UART6应用初始化函数;
	Logo_Out(); // 软件logo打印输出
	EEPROM_APPInit(); // EEPROM应用初始化函数;
	APP_ParaRead((USER_VARS *)&UserVars.Head); // 读取用户参数
	IAP_APPInit(); // IAP应用初始化
	IO_APPInit(); // 所有IO初始化
	EXTI_APPInit(); // 外部中断和事件应用初始化
	BKP_Init(); // BKP应用初始化
	IWDG_Init(IWDG_TIME); // 初始化看门狗时间
	IWDG_Ctrl(CMD_IWDG_ENA); // 使能看门狗
	IWDG_Ctrl(CMD_IWDG_CLEAR); // 喂狗
	ModbusSlave_APPInit(); //Modbus从机(设备)模式应用初始化函数;
	RTC_APPInit(); // RTC应用初始化函数;
	SPI_APPInit(); // SPI总线应用初始化
	ADC_APPInit(); // ADC应用初始化函数;
	DAC_APPInit(); // DAC应用初始化

	TIM_APPInit(); // 定时器应用初始化
	FCLK_APPInit(); // FCLK输入应用初始化
	PWM_APPInit(); // PWM输出应用初始化
	CAN_APPInit(); // CAN1-2应用初始化
	SPIFlash_APPInit(); // SPI Flash(W25QXX或AT45DBXX)应用初始化
	SD_APPInit(); // SD卡应用初始化
	USBH_APPInit(); // USB主机模式应用初始化
	USBD_APPInit(); // USB设备模式应用初始化
	FSMC_APPInit(); // FSMC总线初始化
	NFlash_APPInit(); // NAND FLASH应用初始化
	NET_APPInit(); // 网络配置使能
	}

使用说明
 	
一般在主函数开头调用
	int main (void)
	{
	API_Init (); //main主程序第一个调用该函数初始化
	…… // 后面程序
	}

3. 系统及驱动库初始化函数(sysint.h)

3.1 系统驱动库初始化

函数原型	INT32S SysLib_Init(SYSLIB_PARA *pPara)
函数功能	系统驱动库初始化函数
入口参数	SYSLIB_PARA *pPara
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码；

使用说明	// SYSLIB参数结构
	typedef struct
	{
	INT32U Flag; // 标志
	INT8U ModuleType; // 模块类型
	INT8U DebugUart; // 选择printf函数输出Uart
	INT16U VectorTableAddr; // 中断向量表地址
	INT32U OscClk; // 外部或内部晶振频率
	INT32U SysClk; // 系统时钟频率
	INT16U Tick; // 系统定时器定时时间, 单位ms
	}SYSLIB_PARA;
	参考APP_Init.c文件中的函数SysLib_APPInit()中初始化例程,主要实现以下功能：
	(1) 通过标志Flag设置操作系统使能、中断向量表、系统定时器使能、选择内部时钟晶振还是外部时钟晶振、各种调试信息输出。
	(2) ModuleType：设置模块类型定义
	(3) DebugUart：设置调试信息输出串口(UART)选择
	(4) VectorTableAddr：设置中断向量表地址
	(5) OscClk：设置外部或内部时钟晶振频率
	(6) SysClk：设置系统时钟频率

	(7) Tick: 设置系统定时器定时时间, 单位ms 注意：在API_Init()中SysLib_APPInit()必须第一个被调用,来完成上述功能

3.2 系统控制

函数原型	INT32S SysLib_Ctrl(INT8U Cmd, INT32U Para)
函数功能	系统控制函数
入口参数	Cmd: 控制命令: CMD_SYSLIB_RESET/ CMD_SYSLIB_READ_VERSION/ CMD_SYSLIB_READ_DATE/ CMD_MCO1/ CMD_MCO2 Para: 控制参数
返回参数	命令CMD_SYSLIB_READ_VERSION返回：驱动库版本，返回值/100：主版本号；返回值%100：次版本号； 命令CMD_SYSLIB_READ_DATE返回：驱动库生成日期指针； 其它命令返回：ERR_FALSE: 失败; ERR_TRUE: 成功

使用说明	INT32S ver;
	INT8U *p;
	ver = SysLib_Ctrl(CMD_SYSLIB_READ_VERSION, 0); // 读取驱动库版本
	printf("Lib Version: %d.%d%d\r\n", ver/100, (ver%100)/10, ver%10);
	p = (INT8U *)SysLib_Ctrl(CMD_SYSLIB_READ_DATE, 0); // 读取驱动库生成日期
	printf("Lib Date: %s\r\n", p);
	
	使能MCO1输出HSE:
	SysLib_Ctrl(CMD_MCO1, MCO_OUT_ENA| MCO1_HSE);
	
	系统软件复位
	SysLib_Ctrl(CMD_SYSLIB_RESET, 0);

4.延时函数(delay.h)

4.1 微妙延时函数

函数原型	void Delay_us(INT16U val)
函数功能	微秒延时函数
入口参数	val: 延时时间，范围: 0~65535us
返回参数	无
注意	请不要超过延时时间范围；这个延时是靠软件延时不会十分准确； 最好us延时小于1000，大于1000请用Delay_ms延时函数
使用说明	Delay_us(100); // 延时100us

4.2 毫秒延时函数

函数原型	void Delay_ms(INT16U val)
函数功能	微秒延时函数
入口参数	val: 延时时间，范围: 0~65535ms
返回参数	无
注意	在有操作系统时且val大于10ms会利用操作系统延时函数，小于10ms利用软件延时；

	这个延时是靠软件延时不会十分准确；
使用说明	Delay_ms(100); // 延时100ms

4.3 秒延时函数

函数原型	void Delay_s(INT16U val)
函数功能	秒延时函数
入口参数	val: 延时时间，范围: 0~65535s
返回参数	无
注意	在有操作系统时会利用操作系统延时函数，这个延时是靠软件延时不会十分准确
使用说明	Delay_s(100); // 延时100s

5. 独立看门狗函数(iwdg.h)

5.1 初始化函数

函数原型	void IWDG_Init(INT32U t)
函数功能	IWDG看门狗初始化
入口参数	t，看门狗定时时间，单位：ms，范围: 10~26000ms
返回参数	无
注意	由于看门狗时钟误差，在设定值的一半(t/2)的时间内必须调用清除函数，以保证看门狗不复位；

使用说明	在配置文件中做如下配置:
	#define IWDG_EN 0 // 内部看门狗使能，1：打开使能， 0：关闭
	#define IWDG_TIME 1000 // 看门狗时间设定，设置范围:200~26000ms
	
	在API_Init()函数中初始化例程
	#if (IWDG_EN > 0)
	IWDG_Init(IWDG_TIME); // 初始化看门狗时间
	IWDG_Ctrl(CMD_IWDG_ENA); // 使能看门狗
	IWDG_Ctrl(CMD_IWDG_CLEAR); // 喂狗
	#endif

5.2 控制函数

函数原型	void IWDG_Ctrl(INT8U Cmd)
函数功能	控制操作
入口参数	Cmd： 控制参数： IWDOG_ENA: 打开看门狗； IWDOG_CLEAR: 清除看门狗定时器，即喂狗；
返回参数	无

使用说明	void main(void)
	{
	API_Init(); // 初始化
	#if (IWDG_EN > 0)
	WatchDog_Ctrl(CMD_IWDG_CLEAR); // 打开看门狗
	#endif

```
    While(1)
    {
        Delay_ms(500);                  // 间隔500ms调用清除看门狗
        #if (IWDG_EN > 0)
        WatchDog_Ctrl(IWDOG_CLEAR);     // 系统不会复位，如果超过
        #endif                          // 1000ms则系统会复位
    }
}
```

6. CRC16校验函数(crc.h)

函数原型	INT16U CRC16(INT8U *p, INT16U len)
函数功能	计算CRC16
入口参数	*p： 要计算CRC16的数据指针; Len：数据长度;
返回参数	返回：16位CRC16校验值

调用示例	INT16U crc, i;
	INT8U buf[64];
	for (i=0; i<64; i++)
	{ buf[i] = i;
	}
	crc = CRC16(buf, 64);
	printf(" crc: %d \r\n", crc);

7. RTC读写操作(rtc.h)

7.1 RTC初始化函数

函数原型	INT32S RTC_Init(RTC_PARA *pPara)
函数功能	RTC初始化函数
入口参数	pPara, 参数指针
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码;

使用说明	在配置文件中做如下配置:
	#define RTC_EN 1 // RTC使能, 1：打开使能， 0：关闭
	#define RTC_SECIT_EN 0 // RTC秒中断使能, 1：打开使能， 0：关闭
	#define RTC_ALRIT_EN 0 // RTC闹钟中断使能, 1：打开使能， 0：关闭
	
	参考在API_Init()函数中调用RTC_APPInit()应用初始化如下:
	RTC_PARA Para;
	Para.Flag = 0;
	Para.Flag |= RTC_CLK_LSE_FLAG; // 设置外部低速时钟晶振
	flag = RTC_Init((RTC_PARA *)&Para.Flag); // RTC初始化

7.2 RTC时钟设置函数

函数原型	INT32S RTC_Write(RTC_TIME *rtc)
函数功能	设置RTC时间

入口参数	*rtc, RTC时间数据指针
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;

使用说明	INT32S flag;
	RTC_TIME rtc;
	rtc.year = 12; rtc.month = 12; rtc.day = 31; rtc.hour = 23;
	rtc.minute = 59; rtc.second = 30;
	flag = RTC_Write(&rtc);
	if (flag == ERR_TRUE)
	{ //设置时间成功}

7.3 RTC时钟读取函数

函数原型	INT32S RTC_Read(RTC_TIME *rtc)
函数功能	读取RTC时间
入口参数	*rtc, RTC时间数据指针
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;

使用说明	INT32S flag;
	RTC_TIME rtc;
	flag = RTC_Read(&rtc);
	if (flag == ERR_TRUE) //读取时间成功
	{ printf("Date: %d-%d-%d, time: %d: %d: %d, 星期: %d\r\n", rtc.year+2000,rtc.month,rtc.day,rtc.hour,rtc.minute,rtc.second,rtc.week);}

7.4 RTC控制函数

函数原型	INT32S RTC_Ctrl(INT8U Cmd, INT32U Para, RTC_TIME *rtc)
函数功能	RTC控制操作
入口参数	Cmd: 控制命令: CMD_RTC_GET_COUNTER, 读取RTC计数器值命令 CMD_RTC_SET_COUNTER, 设置RTC计数器值命令 CMD_RTC_SET_ALMTIM, 设置闹钟时间 CMD_RTC_GET_ALMCOUNT 读取闹钟计数器值命令 CMD_RTC_SET_ALMCOUNT 设置闹钟计数值 CMD_RTC_SET_CAL, 设置校准值 CMD_RTC_SET_1SPLUS, 设置PC13(IO46)输出秒脉冲 CMD_RTC_SET_ALARM, 设置PC13(IO46)输出闹钟脉冲 CMD_RTC_SET_COO, 设置PC13(IO46)输出校准时钟脉冲 CMD_RTC_STOP_PC13, 停止PC13输出脉冲; CMD_RTC_SECOND_INT, Para是ENABLE设置秒中断, Para是DISABLE清除秒中断 CMD_RTC_ALARM_INT, Para是ENABLE设置报警中断, Para是DISABLE清除报警中断 Para: 命令参数 *rtc, RTC时间数据指针
返回参数	除返回数据指令外, 返回:ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
使用说明	参考在API_Init()函数中调用RTC_APPInit()应用初始化

8. IO读写操作(gpio.h)

8.1 IO初始化函数：

函数原型	INT32S IO_Init(INT32U IOx, INT8U Mode, INT8U Speed)
函数功能	IO初始化函数
入口参数	IOx, 单独IO端口, 按单个端口初始化: PA0~PA15,PB0~PB15,PC0~PC15,PD0~PD15, PE0~PE15,PF0~PF15,PG0~PG15,PH0~PH15,PI0~PI15; 成组端口, 只初始化bit0~bit15为1的端口: (PA/PB/PC/PD/PE/PF/PG/PH/PI)+(GPIO_PIN_0|GPIO_PIN_1|...|GPIO_PIN_15) Mode, IO模式设置如下: 通用输出模式: IO_OUT_PP, 通用推挽输出模式; IO_OUT_OD, 通用开漏输出模式; 输入模式: IO_IN_FLOATING, 浮空输入模式(复位后的状态) IO_IN_IPD, 内部下拉输入模式; IO_IN_IPU, 内部上拉输入模式 模拟输入模式: IO_AIN, 模拟输入模式; Speed, IO输出速度：如果IO模式是输入模式, 则该参数设置为: 0(IO_INPUT), 不区分速度;如果IO模式是输出模式, 则该参数设置为速度选择: IO_SPEED_10MHz, 最大速度10MHz IO_SPEED_2MHz, 最大速度2MHz IO_SPEED_50MHz, 最大速度50MHz IO_SPEED_25MHz, 最大速度25MHz, 只适用STM32F4xx IO_SPEED_100MHz, 最大速度100MHz, 只适用STM32F4xx
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
使用说明	参考在API_Init()函数中调用IO_APPInit()实现所有IO初始化 以PA0为例 推挽输出初始化：IO_Init(PA0, IO_OUT_PP, IO_SPEED_2MHz); 漏极开路输出： IO_Init(PA0, IO_OUT_OD, IO_SPEED_2MHz); 浮空输入： IO_Init(PA0, IO_IN_FLOATING, 0); 内部上拉输入： IO_Init(PA0, IO_IN_IPU, 0);

8.2 IO读函数

函数原型	INT32U IO_Read(INT32U IOx)
函数功能	读取IO端口电平
入口参数	IOx, 单独IO端口: PA0~PA15,PB0~PB15,PC0~PC15,PD0~PD15,PE0~PE15,PF0~PF15, PG0~PG15,PH0~PH15,PI0~PI15; 成组端口, 只读取bit0~bit15为1的端口: (PA/PB/PC/PD/PE/PF/PG/PH/PI)+(GPIO_PIN_0|GPIO_PIN_1|...|GPIO_PIN_15)
返回参数	IOx是单独IO端口：返回值是1,高电平, 是0,低电平; IOx是成组端口： 返回值bit0~bit15分别代表Px0~Px15(x为A,B,C,D,E,F,G,H,I),

	以bit7为例, 是1则Px7输入高电平, 是0则Px7输入低电平; 注意：如果返回值是0x80000000, 则表示出错

使用说明	INT32U val;
	val = IO_Read(PA0);
	if (val == 0) {// PA0输入低电平};
	if (val == 1) {// PA0输入高电平};

8.3 IO写函数

函数原型	void IO_Write(INT32U IOx, INT16U val)
函数功能	写入IO输出值
入口参数	IOx, 单独 IO 端口: PA0~PA15,PB0~PB15,PC0~PC15,PD0~PD15,PE0~PE15,PF0~PF15, PG0~PG15,PH0~PH15,PI0~PI15; 成组端口, 只输出 bit0~bit15 为 1 的端口: (PA/PB/PC/PD/PE/PF/PG/PH/PI)+(GPIO_PIN_0|GPIO_PIN_1|...|GPIO_PIN_15) IOx 是单独 IO 端口：val, 1 输出高电平, 0,输出低电平; IOx 是成组端口： val 的 bit0~bit15 分别代表 Px0~Px15(x 为 A,B,C,D,E,F,G,H,I), 以 bit7 为例, 是 1 则 Px7 输出高电平, 是 0 则 Px7 输出低电平
返回参数	无
调用示例	PA7输出高电平：IO_Write(PA7, 1); PA7输出低电平：IO_Write(PA7, 0);

8.4 IO控制函数

函数原型	INT32S IO_Ctrl(INT32U IOx, INT8U Cmd, INT32U Para)
函数功能	IO命令控制
入口参数	IOx, 单独IO端口: PA0~PA15,PB0~PB15,PC0~PC15,PD0~PD15,PE0~PE15,PF0~PF15, PG0~PG15,PH0~PH15,PI0~PI15; 成组端口, 只控制bit0~bit15为1的端口: (PA/PB/PC/PD/PE/PF/PG/PH/PI)+(GPIO_PIN_0|GPIO_PIN_1|...|GPIO_PIN_15) Cmd, IO控制命令: CMD_IO_NEG, IO取反; 参数Para为0 CMD_IO_ON_T, IO置1后并延时一段时间再置0; 参数Para为延时时间,单位ms; CMD_IO_OFF_T, IO置0后并延时一段时间再置1; 参数Para为延时时间,单位ms; CMD_IO_RST, 复位IO寄存器为初始状态; 参数Para为0; 此时IOx应该 PA/PB/PC/PD/PE/PF/PG/PH/PI CMD_IO_CLOSE, 关闭IO时钟,也就是关闭DAC功能,可以省电; 参数Para为0; 此时IOx 应该PA/PB/PC/PD/PE/PF/PG/PH/PI Para, 命令参数;
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
使用说明	PA7输出取反: IO_Ctrl(PA7, CMD_IO_NEG, 0); PA7输出100ms高电平：IO_Ctrl(PA7, CMD_IO_ON_T, 100);

9. 外部IO中断函数(exti.h)

9.1 中断初始化函数

函数原型	INT32S EXTI_Init(EXTI_PARA *pPara)
函数功能	外部IO中断初始化函数;
入口参数	*pPara, 中断初始化参数指针
返回参数	ERR_TRUE: 正确; ERR_FALSE: 失败;

使用说明	以EXIT0为例, 根据配置文件配置如下:
	#define EXTI0_EN 1 // 中断或事件使能: 0,关闭中断和事件; 1, 打开中断;
	// 2, 打开事件请求
	#define EXTI0_IO PE0 // 在PA0, PB0, PC0, PD0, PE0, PF0, PG0, PH0, PI0
	// 中选择一个IO作为中断输入口;
	#define EXTI0_MODE 1 // 触发中断和事件模式: 0,上升沿触发中断和事件; 1,
	//降沿触发中断和事件; 2,上升沿下降沿都触发中断和事件;
	
	参考在API_Init()函数中调用EXTI_APPInit()实现所有外部中断初始化
	
	注: 初始化这个函数没有启动中断, 需要调用EXTI_Ctrl()启动中断

9.2 中断控制函数

函数原型	INT32S EXTI_Ctrl(INT8U id, INT8U Cmd)
函数功能	外部IO中断使能控制函数
入口参数	id, 中断索引: EXTI0_ID~EXTI15_ID, EXTI16_PVD_ID, EXTI17_RTCAlarm_ID, EXTI18_USBWakeUp_ID, EXTI19_NETWakeUp_ID; STM32F4XX芯片新增定义: EXTI20_USBHSWakeUp_ID, EXTI21_RTCTSE_ID, EXTI22_RTCWakeUp_ID Cmd: EXTI_CMD_DIS: 关闭中断和事件请求命令 EXTI_CMD_INT_ENA: 打开中断命令 EXTI_CMD_EVENT_ENA: 打开事件请求命令
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
使用说明	打开和关闭PE0外部中断 使能中断: EXInt_Ctrl(EXTI0_ID, EXTI_CMD_INT_ENA); 关闭中断: EXInt_Ctrl(EXTI0_ID, 0);

9.3 外部IO中断索引ID,产生IO及对应中断处理函数表，用户可在中断处理函数中写入代码

中断索引	可产生该中断的IO必须在表中选择一个	ISRHook.c 中断处理函数
EXTI0_ID	PA0, PB0, PC0, PD0, PE0, PF0, PG0, PH0, PI0	void EXTI0_ISRHook(void)
EXTI1_ID	PA1, PB1, PC1, PD1, PE1, PF1, PG1, PH1, PI1	void EXTI1_ISRHook(void)
EXTI2_ID	PA2, PB2, PC2, PD2, PE2, PF2, PG2, PH2, PI2	void EXTI2_ISRHook(void)
EXTI3_ID	PA3, PB3, PC3, PD3, PE3, PF3, PG3, PH3, PI3	void EXTI3_ISRHook(void)
EXTI4_ID	PA4, PB4, PC4, PD4, PE4, PF4, PG4, PH4, PI4	void EXTI4_ISRHook(void)
EXTI5_ID	PA5, PB5, PC5, PD5, PE5, PF5, PG5, PH5, PI5	void EXTI5_ISRHook(void)
EXTI6_ID	PA6, PB6, PC6, PD6, PE6, PF6, PG6, PH6, PI6	void EXTI6_ISRHook(void)
EXTI7_ID	PA7, PB7, PC7, PD7, PE7, PF7, PG7, PH7, PI7	void EXTI7_ISRHook(void)
EXTI8_ID	PA8, PB8, PC8, PD8, PE8, PF8, PG8, PH8, PI8	void EXTI8_ISRHook(void)

EXTI9_ID	PA9, PB9, PC9, PD9, PE9, PF9, PG9, PH9, PI9	void EXTI9_ISRHook(void)
EXTI10_ID	PA10,PB10,PC10,PD10,PE10,PF10,PG10,PH10,PI10	void EXTI10_ISRHook(void)
EXTI11_ID	PA11,PB11,PC11,PD11,PE11,PF11,PG11,PH11,PI11	void EXTI11_ISRHook(void)
EXTI12_ID	PA12,PB12,PC12,PD12,PE12,PF12,PG12,PH12,PI12	void EXTI12_ISRHook(void)
EXTI13_ID	PA13,PB13,PC13,PD13,PE13,PF13,PG13,PH13,PI13	void EXTI13_ISRHook(void)
EXTI14_ID	PA14,PB14,PC14,PD14,PE14,PF14,PG14,PH14,PI14	void EXTI14_ISRHook(void)
EXTI15_ID	PA15,PB15,PC15,PD15,PE15,PF15,PG15,PH15,PI15	void EXTI15_ISRHook(void)

10. UART通信操作(uart.h)

10.1 UART初始化函数

函数原型	INT32S Uart_Init(INT8U id, UART_PARA *pPara)
函数功能	Uart初始化函数
入口参数	id：UART索引标识(UART1_ID~UART6_ID)； *pPara, UART初始化参数指针；
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码；
使用说明	以UART1为例, 根据配置文件配置如下： #define UART1_EN 1 // UART1使能, 1：打开使能， 0：关闭 #define UART1_BAUD 115200 // 定义波特率，可以设置：1200，2400，4800，9600， // 19200，38400，57600，115200 #define UART1_WORD_LENGTH 0 // 定义数据字长, 0: 8bit; 1: 9bit; #define UART1_STOP_BITS 0 // 停止位, 0:1bit;1:2bit;2:0.5bit;3: 1.5bit; #define UART1_PARITY 0 // 奇偶检验位, 0:无校验; 1:偶校验; 2: 奇校验; #define UART1_RXBUF_SIZE 256 // 定义接收缓存长度, 范围大于0,不可以太大; #define UART1_TXBUF_SIZE 1024 // 定义发送缓存长度, 范围大于0,不可以太大; 参考在API_Init()函数中调用Uart_APPInit()实现UART1初始化: 注意: 一般不需要修改Uart_APPInit()函数

10.2 接收一个字节数据函数

函数原型	INT32S Uart_RecvChar(INT8U id, INT8U *val)
函数功能	接收一个字节数据
入口参数	id：UART索引标识(UART1_ID~UART6_ID)； *val: 接收数据的指针；
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码；

使用说明	INT32S flag;
	INT8U val;
	flag = Uart_RecvChar(UART1_ID, &val);
	if (flag == ERR_TRUE)
	{ // 接收数据成功，数据在val中}

10.3 接收一块数据函数

函数原型	INT32S Uart_Read(INT8U id, INT8U *p, INT16U len)
函数功能	接收一块数据函数

入口参数	id：UART索引标识(UART1_ID~UART6_ID)； *p, 接收数据块指针； len, 接收数据块长度；
返回参数	ERR_TRUE: 接收数据成功；ERR_FALSE: 接收数据失败；

使用说明	INT32S flag;
	INT8U buf[16];
	flag = Uart_Read(UART1_ID, buf，16);
	if (flag == ERR_TRUE)
	{ // 接收数据成功，数据在buf中}

10.4 发送一个字节数据

函数原型	INT32S Uart_SendChar(INT8U id, INT8U val)
函数功能	发送一个字节数据
入口参数	id：UART索引标识(UART1_ID~UART6_ID)； val: 发送的数据；
返回参数	ERR_TRUE,操作成功； 其它值，参见const.h中错误代码；
注意	该函数只是将数据发送缓存中,UART会立即启动发送数据，但函数返回时并不意味着数据已经发送完成；

使用说明	INT32S flag;
	INT8U val;
	val = 55;
	flag = Uart_SendChar (UART1_ID, val);
	if (flag == ERR_TRUE)
	{ // 发送数据成功}

10.5 发送一块数据

函数原型	INT32S Uart_Write(INT8U id, INT8U *p, INT16U len)
函数功能	发送一块数据
入口参数	id：UART索引标识(UART1_ID~UART5_ID)； *p, 发送数据块指针； len, 发送数据块长度；
返回参数	ERR_TRUE,操作成功； 其它值，参见const.h中错误代码；
注意	该函数只是将数据发送缓存中,UART会立即启动发送数据，但函数返回时并不意味着数据已经发送完成；

使用说明	INT32S flag;
	INT8U buf[16], i;
	for (i=0; i<16; i++)
	{ buf[i] = i;
	}
	flag = Uart_Write(UART1_ID, buf，16);
	if (flag == ERR_TRUE)
	{ // 发送数据成功}

10.6 UART控制函数

函数原型	INT32S Uart_Ctrl(INT8U id, INT8U Cmd, INT32U Para)
函数功能	UART控制操作
入口参数	id：UART索引标识(UART1_ID~UART5_ID)； Cmd：UART控制命令: CMD_UART_GetCharsRxBuf: 读取接收数据缓存中数据长度 CMD_UART_GetCharsTxBuf: 读取发送数据缓存中空闲空间长度 CMD_UART_ChangeBaud: 改变波特率 CMD_UART_ClearRxBuffer: 清除接收缓存中数据. CMD_UART_ClearTxBuffer: 清除发送缓存中数据. CMD_UART_ChangeUtcf: 改变串口数据格式. CMD_UART_RXCtrl: 串口接收使能控制 CMD_UART_RST: 复位UART寄存器为初始状态 CMD_UART_CLOSE: 关闭UART时钟,也就是关闭UART功能,可以省电 Para：命令控制参数
返回参数	ERR_TRUE,操作成功；其它值, 参见const.h中错误代码； 有返回数据的返回相应数据；

使用说明
如果间隔20ms接收数据缓存中的数据长度无变化则认为该数据为一段数据；读出数据并原样返回
 	
一般在接收数据之前要调用读取接收缓存内是否有数据，再去读取相应长度数据
	void main(void)
	{
	INT32S flag;
	INT8U buf[64], i;
	INT16U len，rUart1Len;
	rUart1Len=0;
	while(1)
	{ Delay_ms(20); // 延时20ms
	// 读取接收数据长度
	len = Uart_Ctrl(UART1_ID, CMD_UART_GetCharsRxBuf,0);
	if ((len == rUart1Len)&&(len>0))
	{
	if (len>64) {len = 64;};
	Uart_Read(UART1_ID, buf, len); // 读取数据
	Uart_Write(UART1_ID, buf, len);// 原样返回数据
	rUart1Len -= len;
	}
	else
	{ rUart1Len = len; }
	}
	}

10.7 关于调试打印printf函数：

这个是系统调用函数，我们在配置文件中定义这个函数输出通道使用哪个UART, 配置如下：

```
#define DEBUG_EN     1        // DEBUG输出使能, 1：打开使能， 0：关闭
#define DEBUG_UART UART1_ID // 选择DEBUG串口输出,在这里也可以选择UART1_ID~UART6_ID
```

11. SPI读写操作(spi.h)

11.1 SPI初始化函数

函数原型	INT32S SPI_Init(INT8U id, SPI_PARA *pPara)
函数功能	SPI初始化函数
入口参数	id, SPI识别号(SPI1_ID,SPI2_ID,SPI3_ID); *pPara, SPI初始化参数指针;
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
使用说明	以SPI1为例,根据配置文件配置如下: #define SPI1_EN 1 // SPI使能, 1:打开使能, 0:关闭 #define SPI1_CKMODE SPI_CKMODE3 // 时钟相位模式, 参见spi.h中说明 #define SPI1_DIVCLK SPI_DIVCLK_8 // SPI时钟分频系数 具体应用请参考在API_Init()函数中调用SPI_APPInit()实现SPI初始化, 一般客户不需要更改该函数;

11.2 SPI读数据

函数原型	INT32S SPI_Read(INT8U id, INT8U *p, INT32U len)
函数功能	读取数据函数
入口参数	id: SPI识别:SPI1_ID、SPI2_ID、SPI3_ID *p: 读出数据存储的地址指针; len:要读出数据长度;
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;

使用说明	INT32S flag;
	INT8U buf[16];
	IO_Write (PA15,0); // 片选置低,假设PA15是片选信号
	flag = SPI_Read (SPI1_ID, buf, 16);
	IO_Write (PA15,1); // 片选置高
	if (flag == ERR_TRUE)
	{ // 读取数据成功,数据在buf中}

11.3 SPI写数据

函数原型	INT32S SPI_Write(INT8U id, INT8U *p, INT32U len)
函数功能	发送数据函数
入口参数	id: SPI识别:SPI1_ID、SPI2_ID、SPI3_ID *p: 写入数据存储的地址指针; len:写入数据长度;
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;

使用说明	INT32S flag;
	INT8U buf[16], i;
	for (i=0; i<16; i++)
	{ buf[i] = i;
	}
	IO_Write (PA15,0); // 片选置低,假设PA15是片选信号

	flag = SPI_Write (SPI1_ID, buf, 16);
	IO_Write (PA15,1); // 片选置高
	if (flag == ERR_TRUE)
	{ // 写入数据成功}

11.4 SPI读写一个字节数据

函数原型	INT8U SPI_ReadWriteByte(INT8U id, INT8U val)
函数功能	发送数据函数
入口参数	id: SPI识别：SPI1_ID、SPI2_ID、SPI3_ID val: 写入数据;
返回参数	返回：读出的数据;

使用说明	INT8U val,rst;
	val = 10;
	IO_Write (PA15,0); // 片选置低，假设PA15是片选信号
	rst = SPI_ReadWriteByte(SPI1_ID, val);
	IO_Write (PA15,1); // 片选置高
	// 返回数据在rst中

11.5 SPI命令控制

函数原型	INT32S SPI_Ctrl(INT8U id, INT8U Cmd, INT32U Para)
函数功能	SPI命令控制
入口参数	id: SPI识别：SPI1_ID、SPI2_ID、SPI3_ID Cmd: 控制命令: CMD_SPI_ENA: 使能SPI外设, Para为0 CMD_SPI_DIS: 使能SPI外设, Para为0 CMD_SPI_DIVCLK：设置时钟分频系数, Para为SPI_DIVCLK_2~SPI_DIVCLK_256 CMD_SPI_CKMODE：时钟相位模式，Para为：SPI_CKMODE0~SPI_CKMODE3 CMD_SPI_RST: 复位SPI寄存器为初始状态, Para为0 CMD_SPI_CLOSE: 关闭SPI时钟,也就是关闭SPI功能,可以省电, Para为0 Para：SPI命令控制参数;
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
注意	初始化完成，SPI已经使能开始工作，无需调用SPI_Ctrl再次使能; 这里这个函数初始化了SCK/MOSI/MISO, 对于SPI片选信号GPIO用户自行初始化

使用说明	SPI_Ctrl (SPI1_ID, MD_SPI_DIVCLK, SPI_DIVCLK_4); // 设置分频系数
	SPI_Ctrl (SPI1_ID, CMD_SPI_CKMODE, SPI_CKMODE2); // 时钟模式设定

12. I2C读写操作(i2c.h)

12.1 I2C初始化函数

函数原型	INT32S I2C_Init(INT8U I2Cx, I2C_PARA *pPara)
函数功能	I2C初始化函数
入口参数	id, I2C索引标识：I2C1_ID、I2C2_ID、I2C3_ID; *pPara, I2C参数表指针
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;

使用说明	暂无

12.2 I2C读数据

函数原型	INT32S I2C_Read(INT8U id, INT16U I2CAddr, INT16U addr, INT8U *p, INT16U len)
函数功能	I2C总线读数据函数
入口参数	id, I2C索引标识：I2C1_ID、I2C2_ID、I2C3_ID; I2CAddr, I2C器件识别地址; addr,读数据的起始地址; *p, 读出数据存储的地址指针; len,要读出数据长度;
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码;
使用说明	暂无

12.3 I2C写数据

函数原型	INT32S I2C_Write(INT8U id, INT16U I2CAddr, INT16U addr, INT8U *p, INT16U len)
函数功能	I2C总线写数据函数
入口参数	id, I2C索引标识：I2C1_ID、I2C2_ID、I2C3_ID; I2CAddr, I2C器件识别地址; addr, 写数据的起始地址; *p, 写入数据存储的地址指针; len,要写入数据长度;
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码;
使用说明	暂无

12.4 I2C命令控制

函数原型	INT32S I2C_Ctrl(INT8U id, INT8U Cmd, INT32U Para)
函数功能	I2C命令控制
入口参数	id, I2C索引标识：I2C1_ID、I2C2_ID、I2C3_ID; Cmd, I2C控制命令, 如下定义： CMD_I2C: 启用I2C外设或禁用I2C外设, Para是ENABLE或DISABLE CMD_I2C_RST: 复位I2C寄存器为初始状态, Para为0 CMD_I2C_CLOSE: 关闭I2C时钟,也就是关闭I2C功能,可以省电, Para为0 Para，I2C命令控制参数;
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码;
使用说明	暂无

13. CAN总线读写操作(can.h)

13.1 CAN初始化函数

函数原型	INT32S CAN_Init(INT8U id, CAN_PARA *pPara, CAN_FILTER_PARA *pFilter)
函数功能	CAN始化函数
入口参数	id, CAN识别号(CAN1_ID、CAN2_ID)； *pPara, CAN初始化参数指针;

	pFilter, 过滤器参数指针;
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
注意	有2个CAN接口的只需初始化一次过滤器; 如果CAN1初始化了过滤器, 则CAN2就不需要再初始化过滤器了，直接把pFilter参数设置为0就可以了;

使用说明	以CAN1为例,根据配置文件配置如下:
	#define CAN1_EN 1 // CAN1使能, 1：打开使能， 0：关闭
	#define CAN1_MODE 0 // 0,正常模式; 1, 环回模式(用于调试); 2, 静默模式、
	//(用于调试); 3, 环回/静默模式(用于调试);
	#define CAN1_IDE CAN_EXT_ID // 帧类型: 标准帧:CAN_STD_ID,
	// 扩展帧:CAN_EXT_ID
	#define CAN1_RTR CAN_RTR_DATA // 选择数据帧:CAN_RTR_DATA
	// 或远程帧:CAN_RTR_REMOTE
	#define CAN1_BAUD 50000 // CAN1波特率;
	#define CAN1_RXBUF_SIZE 16 // CAN接收缓存可接收消息个数,范围 1~256
	#define CAN1_TXBUF_SIZE 16 // CAN发送缓存可发送消息个数,范围 1~256
	
	具体应用请参考在API_Init()函数中调用CAN_APPInit()实现CAN初始化, 一般客户不需要更改该函数;

13.2 CAN滤波器初始化函数

函数原型	INT32S CAN_FilterInit(CAN_FILTER_PARA *pFilter)
函数功能	CAN过滤器初始化函数
入口参数	*pFilter, 过滤器参数指针
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;

使用说明	CAN_FILTER_PARA Filter;
	INT16U i;
	INT32S flag;
	// 滤波器初始化参数
	Filter.CAN2StartBank = CAN2_START_BANK; // CAN2开始组,
	Filter.FIFO = CAN_FILTER_FIFO; // CAN 过滤器FIFO关联配置
	Filter.Scale = CAN_FILTER_SCALE; // CAN 过滤器位宽寄存器:
	Filter.Mode = CAN_FILTER_MODE; // CAN过滤器模式
	Filter.Active = CAN_FILTER_ACTIVE; // 过滤器激活 (Filter active)
	Filter.pBuf = CAN_FilterBuf; // 设置的过滤器寄存器数据指针
	Filter.len = 14; // 设置的过滤器寄存器数据长度
	Filter.MaxLen = CAN_FILTER_MAXNUM; // 滤波器总长度
	for(i=1; i<=14; i++) // 初始化滤波器ID为1-14
	{
	CAN_FilterBuf[i] = (i<<SHIFT_BIT)|(CAN1_IDE<<2)|(CAN1_RTR<<1);
	}
	flag = CAN_FilterInit((CAN_FILTER_PARA *)&Filter.CAN2StartBank);
	if (flag == ERR_TRUE)
	{ printf("CAN初始化: OK\r\n"); }

13.3 CAN接收数据函数

函数原型	INT32S CAN_Read(INT8U id, CAN_RX_MSG *pRxMsg)
函数功能	CAN接收数据函数
入口参数	id: CAN识别号：CAN1_ID、CAN2_ID; * pRxMsg： 接收数据块指针;
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;

使用说明	INT16U num, i;
	CAN_RX_MSG CanRx;
	INT32S flag;
	num = CAN_Ctrl(CAN1_ID, CMD_CAN_GetMsgRxBuf, 0);
	for (i=0; i<num; i++)
	{
	flag = CAN_Read(CAN1_ID, (CAN_RX_MSG *)&CanRx);
	if (flag == ERR_TRUE)
	{ // 接收数据成功0}
	}

13.4 CAN发送函数

函数原型	INT32S CAN_Write(INT8U id, CAN_TX_MSG *pTxMsg)
函数功能	CAN发送数据函数
入口参数	id: CAN识别号：CAN1_ID、CAN2_ID; * pTxMsg：发送数据块指针;
返回参数	返回：ERR_TRUE,发送成功；ERR_FALSE,发送失败；
注意	该函数只是将数据发送缓存中,CAN会立即启动发送数据，但函数返回时并不意味着数据已经发送完成;

使用说明	INT16U num, i;
	CAN_TX_MSG CanTx;
	INT32S flag;
	// 初始化发送数据
	CanTx.Id = 1;
	CanTx.IDE = CAN_EXT_ID;
	CanTx.RTR = CAN_RTR_DATA;
	CanTx.DLC = 8;
	For (i=0;i<8;i++)
	{
	CanTx.Data[i] = i;
	}
	num = CAN_Ctrl(CAN1_ID, CMD_CAN_GetMsgTxBuf, 0);
	if (num>0)
	{
	flag = CAN_Write(CAN1_ID, (CAN_TX_MSG *)&CanTx);
	if (flag == ERR_TRUE)

	{ // 发送数据成功} }

13.5 CAN控制函数

函数原型	INT32S CAN_Ctrl(INT8U id, INT8U Cmd, INT32U Para)
函数功能	CAN控制函数
入口参数	id: CAN识别号：CAN1_ID、CAN2_ID; Cmd: CAN控制命令： CMD_CAN_GetMsgRxBuf: 读取接收数据消息缓存中消息数量, 默认Para为0 CMD_CAN_GetMsgTxBuf: 读取发送数据消息缓存中空闲空间数量, 默认Para为0 CMD_CAN_ClearMsgRxBuf: 清除接收缓存中数据, 默认Para为0 CMD_CAN_ClearMsgTxBuf: 清除发送缓存中数据, 默认Para为0 CMD_CAN_RST: 复位CAN寄存器为初始状态, 默认Para为0 CMD_CAN_CLOSE: 关闭CAN时钟,也就是关闭CAN功能,可以省电, 默认Para为0 Para: CAN命令控制参数;
返回参数	控制命令CMD_CAN_GetMsgRxBuf和CMD_CAN_GetMsgTxBuf返回相应数值；其它指令返回ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
使用说明	参看上面CAN发送接收函数例程

13.6 注意：CAN总线很复杂，尤其滤波器设置，请用户仔细芯片参考手册CAN这部分，能充分理解这部分。在这里想解释清楚很难！

14. ADC采集操作(adc.h)

14.1 ADC初始化函数

函数原型	INT32S ADC_Init(INT8U id, ADC_PARA *pPara)
函数功能	ADC初始化函数
入口参数	id, ADC索引标识：ADC1_ID,ADC2_ID,ADC3_ID; 默认ADC1_ID *pPara, ADC初始化参数;
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
注意	初始化完成，ADC并没有开始工作, 必须调用ADC_Ctrl(ADC1_ID, CMD_ADC_ENA, 0)使能ADC才开始工作。

使用说明	根据配置文件配置如下：
	#define ADC_EN 1 // ADC使能, 1：打开使能， 0：关闭
	#define ADC_OUT_TYPE ADC_ISROUT // 可以选择: 0：ADC_ISROUT,选择中断输出AD
	//采样值; 1：ADC_READOUT, 选择ADC_Read()函数读取采样值
	// 模块模拟量输入使能定义
	#define AI1_EN 1 // AI1使能, 1：打开使能， 0：关闭
	#define AI2_EN 1 // AI2使能, 1：打开使能， 0：关闭
	#define AI3_EN 1 // AI3使能, 1：打开使能， 0：关闭
	#define AI4_EN 1 // AI4使能, 1：打开使能， 0：关闭
	#define AI5_EN 1 // AI5使能, 1：打开使能， 0：关闭
	#define AI6_EN 1 // AI6使能, 1：打开使能， 0：关闭

	#define AI7_EN 1 // AI7使能, 1：打开使能， 0：关闭
	#define AI8_EN 1 // AI8使能, 1：打开使能， 0：关闭
	#define AI9_EN 0 // AI9使能, 1：打开使能， 0：关闭
	#define AI10_EN 0 // AI10使能, 1：打开使能， 0：关闭
	#define AI_MAX_NUM 10 // 所选择AI输仁鼓茁通道序号最大值, 比如最大使能通道是AI10_EN, 则设置为10, 默认都设置为10
	#define ADC_CHNUM (AI1_EN+AI2_EN+AI3_EN+AI4_EN+AI5_EN+AI6_EN+AI7_EN +AI8_EN+AI9_EN+AI10_EN) // 采样通道数
	#define ADC_AVGNUM 4 // 定义采样次数来计算平均值, 范围 1~256,
	// 注意：此值太大会占用很大内存空间
	#define ADC_FREQ 10 // AD采样频率, 每秒钟采样次数
	#define ADC_SAMPLE_TIME ADC_SAMP7TOUS // 采样间隔
	#define ADC_TIM5 ADC_TIM5CH1_FLAG // 选择AD采样定时器, 可在 ADC_TIM5CH1_FLAG/ADC_TIM5CH2_FLAG/ADC_TIM5CH3_FLAG /ADC_TIM5CH4_FLAG/ADC_TIM5MAIN_FLAG中选择
	
	具体应用请参考在API_Init()函数中调用ADC_APPInit()实现ADC初始化, 一般客户不需要更改该函数;

14.2 设置步骤

(1) 使用 ADC 首先设置 ADC_EN 为 1，使能 ADC

(2) 系统提供 8 路 ADC 转换，再根据实际需要设置 AI1_EN~ AI8_EN,分别使能这 8 路 ADC 转换

(3) 采样通道数 ADC_CHNUM 系统自动计算，不用用户更改。

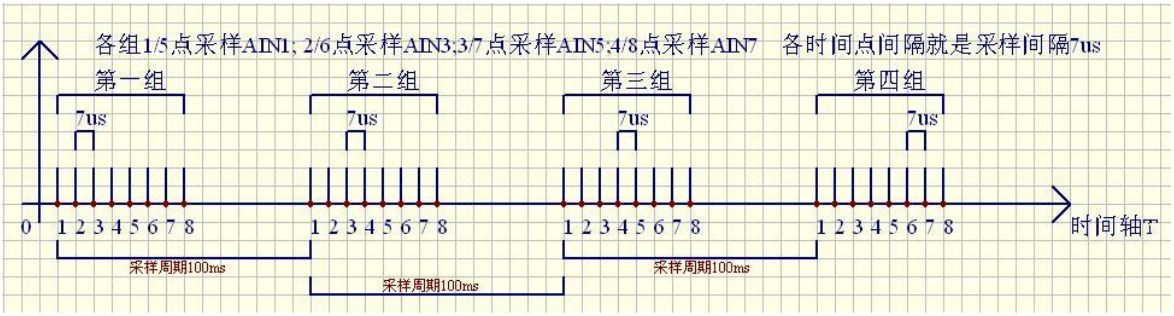
（4）定义采样次数来计算平均值 ADC_AVGNUM 即是完成 1 次计算，每一路 AD 采样次数，用于计算平均值。注意：根据实际需要设置此值，不可设置过大，否则占用很大内存。

（5）ADC_FREQ 是 AD 采样频率，每秒钟采样次数即采样周期。

（6）ADC_SAMPLE_TIME 是采样间隔。

这几个参数相互关系如下图：

假设：AI1_EN、AI3_EN、AI5_EN、AI7_EN 使能，其它关闭； ADC_AVGNUM 设置成 2， ADC_FREQ 设置成 10(HZ)即采样周期 100ms，采样间隔 ADC_SAMPLE_TIME 定义 ADC_SAMP7TOUS 即 7us



14.3 通过中断方式读取 AD 转换值：

(1) 当第一组 AD 转换完成后产生中断，处理完数据会回调 ISRHook.c 的 ADC_ISRHook(ADC_VAR *pData)函数，处理完的数据在 pData 中。

（2）通过邮箱消息发送到 TaskADC 任务中。具体处理参见 TaskADC.c，这种方式接收数据不会丢掉数据，每组采样数据都会被接收。

14.4 读取 AD 数据

函数原型	INT32S ADC_Read(INT8U id, INT16S *p, INT8U len)
函数功能	读取AD数据函数
入口参数	id: ADC 标识：ADC1_ID, ADC2_ID, ADC3_ID, 现只支持 ADC1_ID; *p: 读取返回数据指针; len：读取数据通道个数;
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;

使用说明	INT32S flag;
	INT16S buf[8];
	flag = ADC_Read(ADC1_ID, buf, 8); // 读取1-8通道AD
	if (flag == ERR_TRUE)
	{ // 读取数据成功，数据在buf中: buf[0]:AI1; buf[1]:AI2; buf[2]:AI3;
	//buf[3]:AI4; buf[4]:AI5; buf[5]:AI6; buf[6]:AI7; buf[7]:AI8; }

注意	**调用次函数的频率要大于采样频率可以保证AD采样数据不丢失**

14.5 AD 控制

函数原型	INT32S ADC_Ctrl(INT8U id, INT8U Cmd, INT32U Para)
函数功能	AD控制函数
入口参数	Id: ADC 标识：ADC1_ID, ADC2_ID, ADC3_ID, 现只支持 ADC1_ID; Cmd：ADC 控制命令： CMD_ADC_ENA:开始转换; CMD_ADC_DIS: 停止转换; Para: 控制参数
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;

使用说明	ADC_Ctrl (ADC1_ID, CMD_ADC_ENA, 0); // 使能AD转换
	ADC_Ctrl (ADC1_ID, CMD_ADC_DIS, 0); // 停止AD转换

15. DAC输出操作(dac.h)

15.1 DAC初始化函数

函数原型	INT32S DAC_Init (INT8U id, DAC_PARA *pPara)
函数功能	DAC初始化函数
入口参数	id, ADC索引(DAC1_ID~DAC2_ID); *pPara, DAC设置参数指针
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
使用说明	以DAC1为例,根据配置文件配置如下: #define DAC1_EN 1 // DAC1使能, 1：打开使能， 0：关闭 #define DAC1_MODE 0 // DAC1模式：0(DAC_MODE_MTOUT), 手动输出; // 1(DAC_MODE_ATOUT_N), 连续输出1~N个缓存中的数据后停止; // 2(DAC_MODE_ATOUT), 持续输出缓存中的数据, 不停止; #define DAC1_FREQ 1000 // DAC1自动输出频率 #if ((DAC1_EN>0)&&(DAC1_MODE>0)) #define DAC1_TXBUF_SIZE 256 // DAC1发送数据缓存长度 #endif 具体应用请参考在API_Init()函数中调用DAC_APPInit()实现DAC初始化, 一般客户

	不需要更改该函数；

15.2 手动控制DAC输出函数

函数原型	void DAC_Write(INT8U id, INT16U val)
函数功能	手动控制DAC输出函数即直接向DAC写数据转换成电压输出
入口参数	id：ADC识别号DAC1_ID、DAC2_ID； val： 写入的数据，数据数值范围：0~4095；对应输出电压：0~2.5V（或者10V）；
返回参数	无
注意	这个函数只能在DAC模式设置成手动输出模式有效

使用说明	DAC_Write(DAC1_ID, 0);// 输出0V电压
	DAC_Write(DAC1_ID, 4095*2/5);// 输出1V电压
	DAC_Write(DAC1_ID, 4095*4/5);// 输出2V电压
	DAC_Write(DAC1_ID, 4095);// 输出2.5V电压

15.3 DAC控制函数

函数原型	INT32S DAC_Ctrl(INT8U id, INT8U Cmd, INT32U Para)
函数功能	DAC控制函数
入口参数	id： ADC识别号DAC1_ID、DAC2_ID； Cmd： ADC控制命令： CMD_DAC_DIS： 停止DAC CMD_DAC_ENA： 使能DAC CMD_DAC_FREQ： 设置DAC更新频率，Para是设置的频率 CMD_DAC_STATUS： 读取DAC状态，返回0，停止；返回1，正在执行输出 CMD_DAC_RST： 复位DAC寄存器为初始状态 CMD_DAC_CLOSE： 关闭DAC时钟,也就是关闭DAC功能,可以省电 Para： ADC命令控制参数，Para默认为0
返回参数	无
使用说明	1. 手动输出模式： (1) 使能DAC输出：DAC_Ctrl(DAC1_ID, CMD_DAC_ENA, 0)； (2) 输出数据：DAC_Write(DAC1_ID, 2048)； 注意：这种模式不需要初始化缓存； 2. 自动单次或多次输出缓存中的数据模式： (1) 初始化缓存DAC1/2_Buffer (2) 设置输出频率：DAC_Ctrl(DAC1_ID, CMD_DAC_FREQ, freq)； (3) 使能DAC输出4个缓存波形：DAC_Ctrl(DAC1_ID, CMD_DAC_ENA, 4)； (4) 读取状态：flag = DAC_Ctrl(DAC1_ID, CMD_DAC_STATUS, 0)； 如果flag=1：正在执行输出；如果flag=0：输出4个波形完成； (5) 如果flag=0可以跳到（1）或（2）或（3）步开始执行 3. 自动连续输出缓存中的据据模式 (1) 初始化缓存DAC1/2_Buffer (2) 设置输出频率：DAC_Ctrl(DAC1_ID, CMD_DAC_FREQ, freq)； (3) 使能DAC输出：DAC_Ctrl(DAC1_ID, CMD_DAC_ENA, 0)； (4) 可以调用DAC_Ctrl(DAC1_ID, CMD_DAC_DIS, 0)停止输出或调用 DAC_Ctrl(DAC1_ID, CMD_DAC_FREQ, freq)改变输出波形频率；

注意	在DAC操作模式1，2输出缓存的数据产生波形的频率=DAC输出频率/缓存大小

16. BKP备份寄存器读写(bkp.h)

16.1 BKP初始化函数

函数原型	void BKP_Init(void)
函数功能	BKP初始化函数
入口参数	无
返回参数	无

使用说明	根据配置文件配置如下：
	#define BKP_EN 1 // BKP使能，1：打开使能， 0：关闭
	请参考在API_Init()函数中调用
	#if (BKP_EN_EN > 0)
	BKP_Init();
	#endif

特别说明	备份寄存器是指42(或2048)个16位寄存器，只要有RTC电池供电，在电源断电或系统复位也能保持内容，所以这个在特殊情况下可以随机保存参数，断电不消失。还可以无限次写入，没有像EEPROM那样有次数限制。

16.2备份寄存器写数据函数

函数原型	INT32S BKP_Write(INT16U addr, INT16U *p, INT16U len)
函数功能	备份寄存器写数据函数
入口参数	addr, 备份SRAM(寄存器)起始地址; *p, 写入数据存储的地址指针; len,要写入数据长度;
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
注意	该寄存器在不掉电或掉电有RTC备用电池情况下复位系统数据不会丢失; STM32F10X系列, addr/addr+len范围: 1~42; STM32F40X系列, addr/addr+len范围: 0~2047; 为增加写入速度, 本函数不再做参数范围检查, 请千万不要超过参数范围

使用说明	INT32S flag;
	INT16U buf[4] = {100,1000,10000,20000};
	flag = BKP_Write(1, buf, 4);
	if(flag == ERR_TRUE)
	{//写入成功}

16.3 备份寄存器读数据函数

函数原型	INT32S BKP_Read(INT16U addr, INT16U *p, INT16U len)
函数功能	备份寄存器读数据函数
入口参数	addr, 备份SRAM(寄存器)起始地址; *p, 读出数据存储的地址指针; len,要读出数据长度;
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
注意	该寄存器在不掉电或掉电有RTC备用电池情况下复位系统数据不会丢失;

	STM32F10X系列, addr/addr+len范围: 1~42; STM32F40X系列, addr/addr+len范围: 0~2047; 为增加写入速度, 本函数不再做参数范围检查, 请千万不要超过参数范围

使用说明	INT32S flag;
	INT16U buf[4];
	flag = BKP_Read(1, buf, 4);
	if(flag == ERR_TRUE)
	{//读数据成功,数据在buf中}

17. FLASH读写操作(flash.h)

17.1 向FLASH写数据函数

函数原型	INT32S Flash_Write(INT32U StartAddr, INT8U *p, INT32U len)
函数功能	向FLASH写数据
入口参数	StartAddr, 写入FLASH的起始地址; p, 写数据指针; len, 数据长度
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
注意	len应该是4的倍数
使用说明	无

17.2 读取FLASH数据函数

函数原型	INT32S Flash_Read(INT32U StartAddr, INT8U *p, INT16U len)
函数功能	读取FLASH数据
入口参数	StartAddr, 读FLASH的起始地址; p, 读数据指针; len, 数据长度
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
注意	len应该是4的倍数
使用说明	无

17.3 FLASH控制函数

函数原型	INT32S Flash_Ctrl(INT8U Cmd, INT32U Para)
函数功能	FLASH控制函数
入口参数	Cmd: 控制命令: CMD_FLASH_UNLOCK, FLASH解锁 CMD_FLASH_LOCK, FLASH加锁 CMD_FLASH_PAGE_ERASE, FLASH页擦除(2KB) Para: 控制参数 控制命令CMD_FLASH_UNLOCK和CMD_FLASH_LOCK, Para设置为0 控制命令CMD_FLASH_PAGE_ERASE, Para为擦除的页或扇区编号
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
注意	对于CMD_FLASH_PAGE_ERASE命令, STM32F1xx系列芯片: 按页(2KB)擦除; STM32F4xx系列芯片: 按扇区(FLASH_SECTOR_0~FLASH_SECTOR_11)擦除;
使用说明	无

18.IAP固件更新操作(IAP.h)

18.1 IAP初始化函数

函数原型	INT32S IAP_Init(IAP_PARA *pPara)
函数功能	IAP初始化函数

入口参数	*pPara, 初始化参数指针
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;

使用说明	根据配置文件配置如下:
	#define IAP_EN 0// 0,可生成用仿真器下载调试代码; 1, 用于生成IAP更新代码
	#if (IAP_EN>0)
	#define IAP_FLASH 1 // 更新固件存储器选择: 0,2: 无效; 1(IAP_B_ID), B区
	//FLASH; 3(IAP_C_ID), C区FLASH; 对于STM32F107VC: 如果固件小于120KB,
	//可以选择B区Flash, 如果固件大于120KB必须选择C区Falsh;
	//对于STM32F103VE/STM32F103ZE和EMB8612I: 如果固件小于248KB,可以选择
	//B区Flash, 如果固件大于248KB必须选择C区Flash
	#define IAP_TFTP_EN 1// 利用以太网TFTP协议更新固件使能: 1,使能; 0,关闭;
	#define IAP_UART_EN 1// 利用串口自定义协议更新固件使能: 1,使能;0, 关闭;
	#endif
	具体应用请参考在API_Init()函数中调用IAP_APPInit()实现IAP初始化, 一般客户不需要更改该函数;

18.2 IAP固件数据写入函数

函数原型	INT32S IAP_Write(INT8U id, INT8U *p, INT32U addr, INT32U len)
函数功能	IAP固件数据写入函数
入口参数	id, 操作FLASH区域标志: 只能是IAP_B_ID(B区)或IAP_C_ID(C区), 其它无效; *p, 要写入固件数据的指针; addr, 写入固件数据的起始地址; len,要写入固件数据的长度;
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
使用说明	参考软件例程

18.3 IAP控制函数

函数原型	INT32S IAP_Ctrl(INT8U id, INT8U Cmd, INT32U Para)
函数功能	IAP控制函数
入口参数	id, 操作FLASH区域标志: 只能是IAP_B_ID(B区)或IAP_C_ID(C区), 其它无效 Cmd: 控制命令: CMD_IAP_UNLOCK, FLASH解锁 CMD_IAP_LOCK, FLASH加锁 CMD_IAP_ERASE, FLASH擦除 CMD_IAP_UPDATE, 更新启动参数(id为IAP_B_ID: B区拷贝到A区并启动; id为IAP_C_ID: C区拷贝到A+B区并启动) Para: 控制参数, 当Cmd是CMD_IAP_UPDATE, 该参数为更新固件大小; Cmd是其它值设置为0;
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
使用说明	参考软件例程

19. PWM输出操作(timer.h)

19.1 PWM初始化函数

函数原型	INT32S PWM_Init(INT8U id, PWM_PARA *pPara)
函数功能	PWM初始化函数

入口参数	id, PWM索引(STM32F1XX系列芯片: PWM1_ID~PWM4_ID; STM32F4XX系列芯片: PWM1_ID~PWM8_ID)； *pPara, 初始化参数指针参数值;
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码;

使用说明	以PWM1为例,根据配置文件配置如下:
	#define PWM1_EN 1 // PWM1使能, 1：打开使能， 0：关闭
	// 定义PWM输出模式设置
	#define PWM1_MODE PWM_FREQ_N // 可以选择: 0(PWM_FREQ): 连续脉冲频率输出,
	// 持续输出； 1(PWM_FREQ_N): 多个脉冲频率输出, 输出完设定的脉冲数后停
	// 止； 2(PWM_RATE): 固定频率占空比可调连续脉冲输出: 输出PWM, 频率
	// 固定,占空比0%-100%可调, 持续输出
	#define PWM1_FREQ 1000 // 初始频率
	#define PWM1_TIM TIM2_ID // 选择定时器
	#define PWM1CH1_EN 1 // PWM1CH1：1, 使能; 0, 关闭
	#define PWM1CH2_EN 0 // PWM1CH2：1, 使能; 0, 关闭
	#define PWM1CH3_EN 0 // PWM1CH3：1, 使能; 0, 关闭
	#define PWM1CH4_EN 0 // PWM1CH4：1, 使能; 0, 关闭
	// PWM所有通道使能
	#define PWM1CH_EN (PWM1CH1_EN|(PWM1CH2_EN<<1)|(PWM1CH3_EN<<2)
	|(PWM1CH4_EN<<3)) // PWM1所有通道使能：
	//BIT0:CH1;BIT1:CH2;BIT2:CH3;BIT3:CH4;
	// PWM1CH1~ PWM1CH4初始占空比, 50%(0(0.0%)~1000(100.0%))
	#define PWM1CH1_RATE 500
	#define PWM1CH2_RATE 500
	#define PWM1CH3_RATE 500
	#define PWM1CH4_RATE 500
	// PWM1CH1~ PWM1CH4停止模式输出管脚电平: 0, 低电平; 1, 高电平
	#define PWM1CH1_PIN 0
	#define PWM1CH2_PIN 0
	#define PWM1CH3_PIN 0
	#define PWM1CH4_PIN 0
	
	具体应用请参考在API_Init()函数中调用PWM_APPInit()实现PWM初始化, 一般客户不需要更改该函数;

19.2 PWM控制函数

函数原型	INT32S PWM_Ctrl(INT8U id, INT8U Cmd, PWM_CTRL *pPara)
函数功能	PWM输出控制操作
入口参数	id: PWM索引：id, PWM索引(STM32F1XX系列芯片: PWM1_ID~PWM4_ID; STM32F4XX系列芯片: PWM1_ID~PWM8_ID) cmd：读取命令: CMD_PWM_ENA： 使能; CMD_PWM_DIS： 停止; CMD_PWM_FREQ：设置PWM频率;

	CMD_PWM_RATE： 设置PWM占空比； CMD_PWM_STATUS： 读取PWM状态； CMD_PWM_PULNUM, 读取当前发出脉冲个数； pPara, 控制参数指针；
返回参数	控制命令是CMD_PWM_STATUS时返回PWM状态;其它命令返回ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码

使用说明	PWM_CTRL Para;
	// 初始化控制参数
	Para.Freq = PWM1_FREQ; // 设置默认频率
	Para.Chx = PWM1CH_EN; // 设置使能通道
	Para.Rate[0] = PWM1CH1_RATE; // 设置每个通道输出PWM占空比
	Para.Rate[1] = PWM1CH2_RATE;
	Para.Rate[2] = PWM1CH3_RATE;
	Para.Rate[3] = PWM1CH4_RATE;
	
	#if (PWM1_MODE == PWM_FREQ) // 连续脉冲频率输出模式, 持续输出
	Para.Mode = PWM_CTRL_FREQ; // 设置控制更改PWM频率输出, 并且连续
	#endif
	#if (PWM1_MODE == PWM_FREQ_N) // 多个脉冲频率输出, 输出完设定的4
	// 冲数后停止
	Para.Mode = PWM_CTRL_FREQ_N; // 控制更改PWM频率输出, 并且输出N个
	//脉冲后停止
	Para.Num[0] = FREQ_N; // 设置每个通道输出脉冲个数
	Para.Num[1] = FREQ_N;
	Para.Num[2] = FREQ_N;
	Para.Num[3] = FREQ_N;
	#endif
	#if (PWM1_MODE == PWM_RATE) // 固定频率占空比可调连续脉冲
	// 出: 输出PWM, 频率固定, 占空比0%-100%可调, 持续输出
	Para.Mode = PWM_CTRL_RATE; // 控制更改PWM占空比输出
	#endif
	PWM_Ctrl(id, CMD_PWM_ENA, pPara); // 按参数设置使能PWM输出
	
	具体应用请参考TaskPWMFClk.c中的例程；

20. FCLK脉冲输入操作(timer.h)

20.1 FCLK初始化函数

函数原型	INT32S FCLK_Init(INT8U id, FCLK_PARA *pPara)
函数功能	FCLK初始化函数
入口参数	id, FCLK索引(STM32F1XX系列芯片: FCLK1_ID~FCLK4_ID; STM32F4XX系列芯片: FCLK1_ID~FCLK8_ID); *pPara, FCLK初始化参数指针；
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码；
使用说明	以FCLK1为例,根据配置文件配置如下：

	#define FCLK1_EN 1 // FCLK1使能, 1：打开使能， 0：关闭
	#define FCLK1_MODE 2 // 模式选择: 0(FCLK_MODE_COUNT), 计数模式(1路, CH1 // 输入有效); 1(FCLK_MODE_DECODE), 正交编码器计数(CH1接A，CH2接B); // 2(FCLK_MODE_FREQ), 测频模式(4路, CH1, CH2, CH3, CH4输入都有效); // 3(FCLK_MODE_PWMRATE), 测PWM占空比模式(1路, CH1输入有效);
	#define FCLK1_TIM TIM1_ID // 选择定时器, 这个设置不可更改
	#define FCLK1CH1_EN 1 // FCLK1：1, 使能; 0, 关闭
	#define FCLK1CH2_EN 1 // FCLK2：1, 使能; 0, 关闭
	#define FCLK1CH3_EN 0 // FCLK3：1, 使能; 0, 关闭
	#define FCLK1CH4_EN 0 // FCLK4：1, 使能; 0, 关闭
	// FCLK1所有通道使能
	#define FCLK1CH_EN (FCLK1CH1_EN|(FCLK1CH2_EN<<1)|(FCLK1CH3_EN<<2) |(FCLK1CH4_EN<<3)) // FCLK1所有通道使能
	#define FCLK1_MINFREQ 100 // 模式2，3中，测量最小频率设定, 单位hz
	具体应用请参考在API_Init()函数中调用FCLK_APPInit()实现FCLK初始化, 一般客户不需要更改该函数;

20.2 FCLK控制函数

函数原型	INT32S FCLK_Ctrl(INT8U id, INT8U Cmd, INT8U Chx)
函数功能	FCLK输入控制函数
入口参数	id, FCLK 索引(STM32F1XX 系列芯片: FCLK1_ID~FCLK4_ID; STM32F4XX 系列芯片: FCLK1_ID~FCLK8_ID) Cmd, 控制命令: CMD_FCLK_ENA, 使能; CMD_FCLK_DIS, 停止; CMD_FCLK_STATUS, 读取 FCLK 状态; Chx, 通道号(FCLK_CH1FLAG~FCLK_CH4FLAG);
返回参数	控制命令是CMD_PWM_STATUS时返回FCLK状态; 其它命令返回ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
使用说明	在调用完FCLK_Init()初始化函数后，必须调用FCLK_Ctrl()控制启动函数：例如：FCLK_Ctrl(FCLK1_ID, CMD_FCLK_ENA, FCLK1CH_EN);

20.3 FCLK读取函数

函数原型	INT32S FCLK_Read(INT8U id, INT8U Cmd, INT8U Chx, INT32U *p, INT16U len, INT16U TimeOut)
函数功能	FCLK读取函数
入口参数	id, FCLK 索引(STM32F1XX 系列芯片: FCLK1_ID~FCLK4_ID; STM32F4XX 系列芯片: FCLK1_ID~FCLK8_ID) Cmd：读取命令: CMD_FCLK_FREQ：读取 FCLK_CH1, FCLK_CH2, FCLK_CH3, FCLK_CH4,的频率，单位：0.01hz CMD_FCLK_DECODE：读取正交解码计数值; CMD_FCLK_PWMRATE： 读取 FCLK1 的占空比，单位：0.01%; CMD_FCLK_COUNT： 读取计数值; Chx： 通道号：FCLK_CH1、FCLK_CH2、FCLK_CH3、FCLK_CH4;

	*p: 数据缓存指针; len: 要读取数据长度; TimeOut: 超时返回;
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;

使用说明	根据FCLK工作模式配置选择不同命令读取
	INT32U buf[16];
	INT32S flag;
	(1) 读取频率
	flag = FCLK_Read(FCLK1_ID, CMD_FCLK_FREQ, FCLK_CH1, buf, 16, 1000);
	if (flag == ERR_TRUE)
	{// 16组频率值在buf中, 单位0.01hz}
	(2) 读取占空比
	flag = FCLK_Read(FCLK1_ID, CMD_FCLK_PWMRATE, 0, buf, 16, 1000);
	if (flag == ERR_TRUE)
	{// 占空比值在buf中, 单位0.01%}
	(3) 读取计数值
	flag = FCLK_Read(FCLK1_ID, CMD_FCLK_COUNT, 0, buf, 1, 1000);
	if (flag == ERR_TRUE)
	{// 计数值在buf[0]中}
	(4) 读取正交解码计数值
	flag = FCLK_Read(FCLK1_ID, CMD_FCLK_DECODE, 0, buf, 1, 1000);
	if (flag == ERR_TRUE)
	{// 计数值在buf[0]中}

21. 定时器操作(timer.h)

21.1 定时器初始化函数

函数原型	INT32S Timer_Init(INT8U id, TIM_PARA *pPara)
函数功能	TIMER初始化函数
入口参数	id, TIMER索引: STM32F1XX: TIM1_ID~TIM8_ID; STM32F4XX: TIM1_ID~TIM14_ID; *pPara, 初始化参数指针;
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;

使用说明	以TIM1为例,根据配置文件配置如下:
	#define TIM1_EN 0 // TIM1使能, 1: 打开使能, 0: 关闭
	#define TIM1_MODE 0 // TIM1工作模式: 0, TIM_WKMODE_INT, 定时器工作在定
	// 时中断模式, 定时时间由参数t,t1-t4设置;
	// 1, TIM_WKMODE_COUNT, 定时器工作在定时计数模式, 外部通
	// 过调用Timer_Ctrl函数参数CMD_TIM_ENA/CMD_TIM_DIS启动
	停// 止定时器, 再通过参数CMD_TIM_READ读取计数值
	#define TIM1CH1_EN 0 // TIM1子定时器1使能, 1: 打开使能, 0: 关闭
	#define TIM1CH2_EN 0 // TIM1子定时器2使能, 1: 打开使能, 0: 关闭
	#define TIM1CH3_EN 0 // TIM1子定时器3使能, 1: 打开使能, 0: 关闭
	#define TIM1CH4_EN 0 // TIM1子定时器4使能, 1: 打开使能, 0: 关闭
	// TIM1所有定时器(包含子定时器)使能

	#define TIM1CH_EN (TIM1_EN|(TIM1CH1_EN<<1)|(TIM1CH2_EN<<2)
	|(TIM1CH3_EN<<3)|(TIM1CH4_EN<<4)) // TIM1所有定时器(包含子定时器)：
	// 初始定时时间设定 (注意: 子定时器1、2、3、4定时时间必须小于主定时器定时时间)
	#define TIM1_T 1000000 // TIM1主定时器定时时间, 单位us
	#define TIM1_T1 1000000 // TIM1子定时器1定时时间, 单位us
	#define TIM1_T2 1000000 // TIM1子定时器2定时时间, 单位us
	#define TIM1_T3 1000000 // TIM1子定时器3定时时间, 单位us
	#define TIM1_T4 500000 // TIM1子定时器4定时时间, 单位us
	#define TIM1_PSC (SYSCLK/1000000) // 分频系数, 当工作模式设置为
	// TIM_WKMODE_COUNT, 请设置这个; 默认计数单位是1us
	
	具体应用请参考在API_Init()函数中调用TIM_APPInit()实现FCLK初始化, 一般客户不需要更改该函数;

21.2 定时器控制函数

函数原型	INT32S Timer_Ctrl(INT8U id, INT8U Cmd, TIM_CTRL *pPara)
函数功能	定时器控制函数
入口参数	id, TIMER索引: STM32F1XX: TIM1_ID~TIM8_ID; STM32F4XX: TIM1_ID~TIM14_ID; Cmd：控制命令: CMD_TIM_EMA: 使能定时器 CMD_TIM_DIS: 关闭定时器 CMD_TIM_STATUS: 读取定时器状态 pPara, 参数数据结构指针;
返回参数	控制命令是CMD_TIM_STATUS时返回TIMER状态;其它命令返回ERR_TRUE, 控制正确, ERR_TRUE, 控制失败；当控制命令是CMD_TIM_STATUS, 返回数值：0: 定时器停止; 1：定时器正在运行

使用说明	（1）以TIM2为例,在定时器工作在定时中断模式中,启动定时器
	TIM_CTRL TIMCtrl;
	#if ((TIM2_EN == 1)&&(TIM2_MODE == TIM_WKMODE_INT))
	TIMCtrl.Chx = TIM2CH_EN;
	TIMCtrl.t = TIM2_T; // 设置主定时器定时时间
	TIMCtrl.t1 = TIM2_T1; // 设置子定时器1定时时间
	TIMCtrl.t2 = TIM2_T2; // 设置子定时器2定时时间
	TIMCtrl.t3 = TIM2_T3; // 设置子定时器3定时时间
	TIMCtrl.t4 = TIM2_T4; // 设置子定时器4定时时间
	
	Timer_Ctrl(TIM2_ID, CMD_TIM_ENA, (TIM_CTRL *)&TIMCtrl.Chx);
	#endif
	定时中断函数在ISRHook.c中: void TIM2_ISRHook(INT32U flag)用户可以在这个函数下写自己的应用程序
	
	（2）以TIM2为例,在定时器工作在定时计数模式中,读取延时20ms的计数值，例

	程如下：
	// TIM2读取计数值
	#if ((TIM2_EN == 1)&&(TIM2_MODE == TIM_WKMODE_COUNT)) // 定时器工作在定时计数模式
	Timer_Ctrl(TIM2_ID, CMD_TIM_ENA, 0); // 使能定时器
	Delay_ms(20); // 延时20ms
	Timer_Ctrl(TIM2_ID, CMD_TIM_DIS, 0); // 关闭定时器
	cnt = Timer_Ctrl(TIM2_ID, CMD_TIM_READ, 0); // 读取20ms计数值
	#if (DEBUG_APP_EN == 1)
	printf("TIM2计数值: %d\r\n", cnt); // 打印输出计数值
	#endif
	#endif

22. 外部总线操作(fsmc.h, 只在STM32F103ZE模块中有效)

22.1 初始化函数

函数原型	INT32S FSMC_Init(FSMC_PARA *pPara)
函数功能	FSMC初始化函数
入口参数	*pPara, 初始化参数指针
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码；

使用说明	根据配置文件配置如下：
	#define EXBUS_EN 1 // 外部总线使能：1, 使能； 0, 关闭；
	
	#define EXBUS_ADDSET 3 // 外部总线地址建立时间(范围：0~15)：实际
	// 建立时间(EXBUS_ADDSET+1)个HCLK;
	#define EXBUS_DATAST 5 // 外部总线数据保持时间(范围：1~255)：实际
	// 保持时间：读(EXBUS_DATAST+3)个HCLK, 写
	// (DATAST+1)个HCLK;
	#define EXBUS_TURN 3 // 外部总线恢复时间(范围：0~15)：实际恢复
	// 时间(EXBUS_TURN+1)个HCLK：1, 使能； 0, 关闭；
	
	具体应用请参考在API_Init()函数中调用FSMC_APPInit()实现FSMC初始化, 一般客户不需要更改该函数；

22.2 总线读数据

函数原型	INT16U EXBUS_Read(INT16U addr)
函数功能	总线读数据
入口参数	addr： 读数据地址, 范围0~31(针对基地址的相对地址)；
出口参数	返回16位数据
使用说明	INT16U val; val = EXBUS_Read(2); // 读取地址2数据

22.3 总线写数据

函数原型	EXBUS_Write(INT16U addr, INT16U val)
使用说明	总线读数据
入口参数	addr： 写数据地址, 范围0~31(针对基地址的相对地址); val：写入的数据
返回参数	无
调用示例	INT16U val; val = 0x5A5A EXBUS_Write (2, val); // 向地址2写入数据

22.4 总线控制

函数原型	INT32S FSMC_Ctrl(INT8U Cmd, INT32U Para)
函数功能	总线控制函数
入口参数	Cmd: CMD_FSMC_SRAM_TEST: SRAM测试 Para: 默认
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;

使用说明	INT32S flag;
	flag = FSMC_Ctrl (CMD_FSMC_SRAM_TEST, 0);
	if (flag == ERR_TRUE)
	{ // SRAM测试成功 }

23 USB设备接口(USBDevice.h)

23.1初始化函数

函数原型	INT32S USBD_Init(USBD_PARA *pPara)
函数功能	USB设备模式初始化函数
入口参数	USBD_PARA *pPara, USB初始化参数
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码;

使用说明	根据配置文件配置如下：
	// USB设备模式设置
	// 注意: USB_HOST_EN和USB_DEVICE_EN不能同时使能
	#define USB_DEVICE_EN 0 // USB设备设备使能：1, 使能； 0, 关闭;
	// USB设备, 实现将板子的SD卡、SPI FLASH或者NAND FALSH作为存储介质实现 USB Mass Storage功能, 计算机可以通过USB口直接读取SD卡、SPI FLASH或者NAND FLASH的文件； 注: USB_VCP_EN和USB_MSC_EN不能同时使能
	#define USB_MSC_EN 1 // USB Mass Storage使能, 1：打开使能， 0：关闭
	#define USB_MSC_LUN 0 // USB Mass Storage存储介质选择: 0, SPI FLASH
	// 置为逻辑驱动器; 1, SD卡设置为逻辑驱动器; 其它值无效
	
	// USB设备 虚拟串口通信设置
	#define USB_VCP_EN 0 // USB VCP虚拟串口使能, 1：打开使能， 0：关闭
	#define USB_RXBUF_SIZE 1024 // 定义接收缓存长度
	
	具体应用请参考在API_Init()函数中调用USBD_APPInit()实现USB设备初始化, 一

	般客户不需要更改该函数;

23.2 虚拟串口读数据

函数原型	INT32S USB_Read(INT8U id, INT8U *p, INT16U len)
函数功能	虚拟串口读数据
入口参数	id: 索引标识, 暂时未用, 默认0; *p: 接收数据块指针; len: 接收数据块长度
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码
注意	如果数据缓存中数据小于读取长度len, 则该函数并不等待, 直接返回ERR_FALSE; 读取成功后, 数据缓存清空
使用说明	参考USB_Ctrl()使用说明

23.3 虚拟串口写数据

函数原型	INT32S USB_Write(INT8U id, INT8U *p, INT16U len)
函数功能	虚拟串口写数据
入口参数	id: 索引标识, 暂时未用, 默认0; *p: 发送数据块指针; len: 发送数据块长度
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码
注意	该函数发送完成后返回, 如果超时1秒还没发送完返回ERR_FALSE
使用说明	参考USB_Ctrl()使用说明

23.4 虚拟串口控制

函数原型	INT32S USB_Ctrl(INT8U id, INT8U Cmd, INT32U Para)
函数功能	虚拟串口控制
入口参数	id: 索引标识, 暂时未用, 默认0; Cmd, USB控制命令: CMD_USBD_START: USB启动运行 CMD_USBD_STOP: USB停止运行 CMD_USBD_VBUS_CONNECT: 读取USB硬件连接状态(是否有USB设备插入): // 返回ERE_TRUE, USB设备已经连接; 返回ERR_FALSE, 无USB设备连接; CMD_USBD_GetCharsRxBuf: 读取接收数据缓存中数据长度 CMD_USBD_ClearRxBuffer: 清除接收缓存 Para, 命令控制参数, 默认0
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码

调用示例
 	
如果间隔20ms接收数据缓存中的数据长度无变化则认为该数据为一段
	void USB_Test(void)
	{
	INT32S flag;
	INT8U buf[64], i;
	INT16U len, rLen;
	rLen=0;
	while(1)

数据；读出数据并原样返回
 	
一般在接收数据之前要调用读取接收缓存内是否有数据，在去读取相应长度数据
	{
	Delay_ms(20); // 延时20ms
	// 读取接收数据长度
	len = USB_Ctrl(0, CMD_USB_GetCharsRxBuf,0);
	if ((len == rLen)&&(len>0))
	{
	if (len>64) {len = 64;};
	USB_Read(0, buf, len); // 读取数据
	USB_Write(0, buf, len);// 原样返回数据
	rLen -= len;
	}
	else
	{
	rLen = len;
	}
	}
	}

24 USB主机接口(USBHost.h)

24.1初始化函数

函数原型	INT32S USBH_Init(INT8U id, USBH_PARA *pPara)
函数功能	USB设备模式初始化函数
入口参数	id, USB索引值, 默认是0 USBD_PARA *pPara, USB初始化参数
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码；

使用说明	根据配置文件配置如下：
	// 注意: USB_HOST_EN和USB_DEVICE_EN不能同时使能
	#define USB_HOST_EN 1 // USB主机模式使能：1, 使能； 0, 关闭;
	// U盘使能配置
	#define UDISK_EN 1 // U盘使能：1, 使能； 0, 关闭;
	
	具体应用请参考在API_Init()函数中调用USBH_APPInit()实现USB设备初始化, 一般客户不需要更改该函数；

24.2 USB主机控制函数

函数原型	INT32S USBH_Ctrl(INT8U id, INT8U Cmd, INT32U Para)
函数功能	USB主机控制函数
入口参数	id, USB索引值, 默认是0 Cmd, 控制命令: CMD_USBH_SYNC: USB主机同步处理 CMD_USBH_STATUS: 读取USB主机状态 CMD_UDISK_STATUS: 读取U盘状态 CMD_UDISK_SECTOR_COUNT: 读取U盘扇区数量

	CMD_UDISK_SECTOR_SIZE: 读取U盘扇区大小 Para, 命令参数;
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码

使用说明	#if (USB_HOST_EN > 0) // USB主机模式使能
	flag = USBH_Ctrl(USB_ID, CMD_USBH_SYNC, 0);
	if (flag&USBH_WORK_OK)
	{// 检测到U盘}
	#endif
	参考FatFS接口文件diskio.c中的应用

24.3 FatFS接口读取U盘扇区数据函数

函数原型	INT8U UDisk_Read(INT8U pdrv, INT8U *p, INT32U sector, INT8U count)
函数功能	FatFS接口读取U盘扇区数据函数
入口参数	pdrv: 物理驱动器序号, 默认0 *p, 读取数据缓存指针; sector, 读取扇区起始序号 count, 读取扇区数量 (1..255)
返回参数	RES_OK, Successful RES_ERROR, R/W Error RES_WRPRT, Write Protected RES_NOTRDY, Not Ready RES_PARERR, Invalid Parameter
使用说明	参考FatFS接口文件diskio.c中的应用

24.4 FatFS接口写入U盘扇区数据函数

函数原型	INT8U UDisk_Write(INT8U pdrv, INT8U *p, INT32U sector, INT8U count)
函数功能	FatFS接口写入U盘扇区数据函数
入口参数	pdrv: 物理驱动器序号, 默认0 *p, 写入数据缓存指针; sector, 写入扇区起始序号 count, 写入扇区数量 (1..255)
返回参数	RES_OK, Successful RES_ERROR, R/W Error RES_WRPRT, Write Protected RES_NOTRDY, Not Ready RES_PARERR, Invalid Parameter
使用说明	参考FatFS接口文件diskio.c中的应用

25. EEPROM读写操作(eeprom.h)

25.1 EEPROM初始化函数

函数原型	INT32S EEPROM_Init(EEPROM_PARA *pPara)

函数功能	EEPROM初始化函数
入口参数	pPara, EEPROM参数指针
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
使用说明	根据配置文件配置如下：注意：这个配置不可更改 #define EEPROM_EN 1 // EEPROM使能, 1：打开使能, 0：关闭 #define EEPROM_DEVICE AT24C64 // 定义器件型号 #define EEPROM_FREQ 100000 // 读写时钟频率 参考在API_Init()函数中调用EEPROM_APPInit()实现EEPROM初始化,一般客户不需要更改该函数

25.2 EEPROM读函数

函数原型	INT32S EEPROM_Read(INT16U addr, INT8U *p, INT16U len)
函数功能	EEPROM读函数
入口参数	addr：EEPROM读取的起始地址; *p: 读取数据存储的地址指针; len: 要读取数据长度;
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
注意	因为AT24C64的地址:0x01E00~0x01FFF(最后512字节)作为驱动库IAP参数, 所以请不做操作这个地址范围,否则出错;

使用说明	INT32S flag;
	INT8U buf[16];
	flag = EEPROM_Read(0, buf, 16); // 读取地址0开始的16个字节
	if (flag == ERR_TRUE)
	{ // 读取数据成功，数据在buf中
	}

25.3 EEPROM写函数

函数原型	INT32S EEPROM_Write(INT16U addr, INT8U *p, INT16U len)
函数功能	EEPROM写函数
入口参数	addr：EEPROM写入的起始地址; *p: 写入数据存储的地址指针; len: 要写入数据长度;
返回参数	返回：ERR_TRUE: 写入成功; ERR_FALSE: 写入失败;
注意	因为AT24C64的地址:0x01E00~0x01FFF(最后512字节)作为驱动库IAP参数, 所以请不做操作这个地址范围,否则出错;

使用说明	INT32S flag;
	INT8U buf[16], i;
	for (i=0; i<16; i++)
	{ buf[i] = i;
	}
	flag = EEPROM_Write(0, buf, 16); // 写入地址0开始的16个字节
	if (flag == ERR_TRUE)
	{ // 写入数据成功}

26. AT45DBXX读写操作(AT45DBXX.h)

26.1 AT45DBXX初始化函数

函数原型	INT32S AT45DBXX_Init(AT45DBXX_PARA *pPara)
函数功能	AT45DBXX初始化函数
入口参数	*pPara, 初始化参数指针
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;

使用说明	根据配置文件配置如下:
	#define SPIFLASH_EN 1 // SPI FLASH使能: 1, 使能; 0, 关闭;
	
	#define SPIFLASH_MODE 1 // SPI FLASH操作方式: 1, 利用FATFS文件系统进行读写; 0,用SPI FLASH读写函数进行操作; 注意:2种操作方式只能选择一种
	#define SPIFLASH_TYPE AT45DBXX // SPI FLASH类型定义: W25QXX或AT45DBXX
	#if (SPIFLASH_TYPE == AT45DBXX)
	#define AT45DBXX_MODEL AT45DB161 // SPI FLASH型号AT45DB081/161/321
	#define AT45DBXX_PAGE_NUM 4096 // SPI FLASH总页数
	#define AT45DBXX_FATFS_PAGENUM (4096-480) // 用于建立文件系统的页数(范围: 0~AT45DBXX_FATFS_PAGENUM-1)
	#define AT45DBXX_ZDY_PAGENUM (SPIFLASH_PAGE_NUM-SPIFLASH_FATFS_PAGENUM) // 用于自定义页数(范围: AT45DBXX_FATFS_PAGENUM ~ AT45DBXX_PAGE_NUM-1)
	#define AT45DBXX_PAGE_SIZE 512 // AT45DBXX 页大小
	#endif
	
	参考在API_Init()函数中调用SPIFlash_APPInit()实现AT45DBXX初始化, 一般客户不需要更改该函数

26.2 按页读取FLASH的数据

函数原型	INT32S AT45DBXX_ReadPage(INT8U *p, INT32U page, INT32U count)
函数功能	按页读取FLASH的数据
入口参数	*p, 要读取数据存储区指针; page,读取flash数据的起始页; count, 要读出数据的页数; page/count+page范围: 0~4095/8191(AT45DB161/AT45DB321);
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
注意	每页512字节

使用说明	INT32S flag;
	INT8U buf[512]; // 注意要设置为全局数组
	flag = AT45DBXX_ReadPage(buf, 0, 1); // 读取0页数据
	if (flag == ERR_TRUE)
	{ // 读取数据成功, 数据在buf中}

26.3 读取FLASH的数据

函数原型	INT32S AT45DBXX_Read(INT8U *p, INT32U addr, INT32U len)
函数功能	读取FLASH的数据
入口参数	*p, 要读取数据存储区指针; addr, 内部FLASH起始地址; len,要读取数据的长度;

	addr地址范围: 0x000000~0x001fffff/0x003fffff(AT45DB161/AT45DB321);
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
注意	本函数不做地址范围范围检查,所以不要超过范围

使用说明	INT32S flag;
	INT8U buf[16];
	flag = AT45DBXX_Read(buf, 0, 16); // 读取地址0开始的16个字节
	if (flag == ERR_TRUE)
	{ // 读取数据成功, 数据在buf中}

26.4 按页写入FLASH数据

函数原型	INT32S AT45DBXX_WritePage(INT8U *p, INT32U page, INT32U count)
函数功能	按页写入FLASH数据
入口参数	*p, 要写入数据的指针; page,写入flash数据的起始页; count, 要写入数据的页数; page范围: 0~4095/8191(AT45DB161/AT45DB321); count范围: count + page<=4095/8191;
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;失败;
注意	每页是512字节

使用说明	INT32S flag, i;
	INT8U buf[512]; // 注意要设置为全局数组
	for (i=0; i<512; i++)
	{ buf[i] = i; }
	flag = AT45DBXX_WritePage(buf, 0, 1); // 写入0页地址数据
	if (flag == ERR_TRUE)
	{// 写入数据成功}

26.5 写入FLASH一段数据

函数原型	INT32S AT45DBXX_Write(INT8U *p, INT32U addr, INT32U len)
函数功能	写入FLASH一段数据
入口参数	*p, 要写入数据的指针; addr,写入flash数据的起始地址; len,要写入数据的长度; addr地址范围: 0x000000~0x001fffff/0x003fffff(AT45DB161/AT45DB321);
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;失败;
注意	本函数不做地址范围范围检查,所以不要超过范围

使用说明	INT32S flag;
	INT8U buf[16], i;
	for (i=0; i<16; i++)
	{ buf[i] = i;
	}
	flag = AT45DBXX_Write(buf, 0, 16); // 写入地址0开始的16个字节
	if (flag == ERR_TRUE)
	{ // 写入数据成功}

26.6 SPI Flash命令控制

函数原型	INT32S AT45DBXX_Ctrl(INT8U Cmd, INT32U Para)
函数功能	AT45DBXX命令控制

入口参数	Cmd, AT45DBXX控制命令： CMD_AT45DBXX_PAGE_ERASE: 擦除页; Para为页数 CMD_AT45DBXX_BLOCK_ERASE: 擦除4KB块, 8页; Para为块数 CMD_AT45DBXX_SECTOR_ERASE: 擦除128KB扇区, 32块，256页; Para为扇区数 CMD_AT45DBXX_CHIP_ERASE: 擦除整个芯片; Para为0 CMD_AT45DBXX_RDID: 读取厂商及器件ID; Para为0 Para，SPI Flash命令控制参数, 参考上面说明
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码; 有返回的命令返回相应参数
使用说明	无

27. W25QXX读写操作(W25QXX.h)

27.1 W25QXX初始化函数

函数原型	INT32S W25QXX_Init(W25QXX_PARA *pPara)
函数功能	W25QXX初始化函数
入口参数	*pPara, 初始化参数指针
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;

使用说明	根据配置文件配置如下：
	#define SPIFLASH_EN 1 // SPI FLASH使能：1, 使能; 0, 关闭;
	#define SPIFLASH_MODE 1 // SPI FLASH操作方式：1, 利用FATFS文件系统进行读
	// 写; 0,用SPI FLASH读写函数进行操作; 注意:2种操作方式只能选择一种
	#define SPIFLASH_TYPE W25QXX // SPI FLASH类型定义: W25QXX或AT45DBXX
	#if (SPIFLASH_TYPE == W25QXX)
	// 注意: 因为W25QXX是按扇区擦除, 所以建立文件系统就按扇区计算
	#define W25QXX_MODEL W25Q64 // SPI FLASH型号：W25Q80~W25Q128
	#define W25QXX_SECTOR_SIZE 4096 // W25QXX扇区大小
	#define W25QXX_SECTOR_NUM 2048 // SPI FLASH总扇区数
	#define W25QXX_FATFS_SECTORNUM (2048-60) // 用于建立文件系统的扇区数(范围：
	// 0~W25QXX_SECTOR_NUM-1)
	#define W25QXX_ZDY_STARTSECTOR W25QXX_FATFS_SECTORNUM
	// 用于自定义扇区起始扇区
	#define W25QXX_ZDY_SECTORNUM (W25QXX_SECTOR_NUM - W25QXX_FATFS_SECTORNUM)
	// 用于自定义扇区数(范围：W25QXX_ZDY_STARTSECTOR~ W25QXX_SECTOR_NUM-1)
	#define W25QXX_PAGE_SIZE 256 // W25QXX读写页大小
	#endif
	
	参考在API_Init()函数中调用SPIFlash_APPInit()实现W25QXX初始化，一般客户不需要更改该函数

27.2 按扇区读取FLASH的数据

函数原型	INT32S W25QXX_ReadSector(INT8U *p, INT32U sector, INT32U count)
函数功能	按扇区读取FLASH的数据
入口参数	*p, 要读取数据存储区指针; sector,读取Flash数据的起始扇区; count, 要读取

	的扇区数;
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码;
注意	扇区大小为512字节, 本函数只能应用在FATFS的diskio.c中调用;

使用说明	INT32S flag;
	INT8U buf[512]; // 注意要设置为全局数组
	flag = W25QXX_ReadSector (buf, 0, 1); // 读取0扇区数据
	if (flag == ERR_TRUE)
	{ // 读取数据成功，数据在buf中}

27.3 读取FLASH的数据

函数原型	INT32U W25QXX_Read(INT8U *p, INT32U addr, INT32U len)
函数功能	读取FLASH的数据
入口参数	*p, 要读取数据存储区指针; addr, 内部FLASH起始地址； len,要读取数据的长度; addr地址范围: W25Q80, 0x000000~0x000fffff; W25Q16, 0x000000~0x001fffff; W25Q32, 0x000000~0x003fffff; W25Q64, 0x000000~0x007fffff; W25Q128, 0x000000~0x00ffffff;
返回参数	ERR_TRUE,操作成功； 其它值, 参见const.h中错误代码;
注意	本函数不做地址范围范围检查,所以不要超过范围

使用说明	INT32S flag;
	INT8U buf[16];
	flag = W25QXX_Read(buf，0, 16); // 读取地址0开始的16个字节
	if (flag == ERR_TRUE)
	{ // 读取数据成功，数据在buf中}

27.4 按页写入FLASH数据

函数原型	INT32S W25QXX_WritePage(INT8U *p, INT32U addr, INT16U len)
函数功能	按页写入FLASH数据
入口参数	*p, 要写入数据的指针; addr, 每页的起始起始地址； len, 要写入数据长度
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;失败;
注意	addr必须是256的整数倍, len范围必须是1-256;

使用说明	INT32S flag，i;
	INT8U buf[256]; // 注意要设置为全局数组
	for (i=0; i<256; i++)
	{ buf[i] = i; }
	flag = W25QXX_WritePage (buf，0, 1); // 写入0页地址数据
	if (flag == ERR_TRUE)
	{// 写入数据成功}

27.5 写入FLASH一段数据

函数原型	INT32S W25QXX_Write(INT8U *p, INT32U addr, INT16U len)
函数功能	写入FLASH一段数据
入口参数	*p, 要写入数据的指针; addr,写入flash数据的起始地址； len,要写入数据的长度 addr地址范围: W25Q80, 0x000000~0x000fffff; W25Q16, 0x000000~0x001fffff;

	W25Q32, 0x000000~0x003fffff; W25Q64, 0x000000~0x007fffff; W25Q128, 0x000000~0x00ffffff;
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;失败;
注意	本函数不做地址范围范围检查,所以不要超过范围

使用说明	INT32S flag;
	INT8U buf[16], i;
	for (i=0; i<16; i++)
	{ buf[i] = i; }
	flag = W25QXX_Write (buf, 0, 16); // 写入地址0开始的16个字节
	if (flag == ERR_TRUE)
	{ // 写入数据成功}

27.6 按扇区写入FLASH数据

函数原型	INT32S W25QXX_WriteSector(INT8U *p, INT32U sector, INT32U count)
函数功能	按扇区写入FLASH数据
入口参数	*p, 要写入数据的指针; sector,写入flash数据的起始扇区序号; count, 要写入扇区数量
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;失败;
注意	扇区大小为512字节, 本函数只能应用在FATFS的diskio.c中调用

使用说明	INT32S flag, i;
	INT8U buf[512]; // 注意要设置为全局数组
	for (i=0; i<512; i++)
	{ buf[i] = i; }
	flag = W25QXX_WriteSector(buf, 0, 1); // 写入0扇区地址数据
	if (flag == ERR_TRUE)
	{// 写入数据成功}

27.7 W25QXX命令控制

函数原型	INT32S W25QXX_Ctrl(INT8U Cmd, INT32U Para)
函数功能	W25QXX命令控制
入口参数	Cmd, W25QXX控制命令: CMD_W25QXX_SYNC: 同步处理(回写缓存), 只能在FATFS的diskio.c中调用 CMD_W25QXX_INIT: W25QXX初始化, 只能在FATFS的diskio.c中调用 CMD_W25QXX_STATUS: 读取W25QXX状态,只能在FATFS的diskio.c中调用 CMD_W25QXX_SECTOR_ERASE: 擦除扇区; Para, 扇区序号 CMD_W25QXX_BLOCK32KB_ERASE: 擦除32KB块; Para, 32KB块序号 CMD_W25QXX_BLOCK64KB_ERASE: 擦除64KB块; Para, 64KB块序号 CMD_W25QXX_CHIP_ERASE: 擦除整个芯片; Para, 为0 CMD_W25QXX_RDID: 读取厂商及器件ID; Para, 为0 CMD_W25QXX_POWERDOWN: 进入掉电模式; Para, 为0 CMD_W25QXX_WAKEUP: 唤醒; Para, 为0 Para, W25QXX命令控制参数, 参看上面说明Para, SPI Flash命令控制参数, 参考上面说明

返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码; 有返回的命令返回相应参数
使用说明	无

28. Nand Flash读写操作(NFlash.h)

28.1 NFlash初始化函数

函数原型	INT32S NFlash_Init(NFLASH_PARA *pPara)
函数功能	NFlash初始化函数
入口参数	*pPara, 初始化参数指针
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
使用说明	根据配置文件配置如下(这个应需要占用缓存128KB): #define NFLASH_EN 1 // Nand Flash使能: 1, 使能; 0, 关闭; #define NFLASH_RBIT_EN 0 // Nand Flash RB信号中断使能: 1, 使能; 0,关闭; #define NFLASH_ECCEN 1 // Nand Flash ECC校验使能: 1, 使能; 0, 关闭; #define NFLASH_ECC_SIZE 512 // Nand Flash ECC页面大小: //256/512/1024/2048/4096/8192字节可选 #define NFLASH_BLOCK_NUM 2048 // Nand Flash 总块数 #define NFLASH_BLOCK_SIZE 64 // Nand Flash 每个块包含页数 #define NFLASH_PAGE_SIZE 2048 // Nand Flash 页大小 #define NFLASH_MAX_BAD_BLOCK 64 // Nand Flash 最大坏块数, 这个部分用于 // 替换数据区坏块 参考在API_Init()函数中调用NFlash_APPInit()实现Nand Flash初始化, 一般客户不需要更改该函数

28.2按扇区读取Nand Flash的数据

函数原型	INT32S NFlash_ReadSector(INT8U *p, INT32U sector, INT32U count)
函数功能	按扇区读取Nand Flash的数据
入口参数	*p, 要读取数据存储区指针; sector,读取Flash数据的起始扇区; count, 要读取的扇区数;
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
注意	扇区大小为512字节, 本函数只能应用在FATFS的diskio.c中调用;
使用说明	参考diskio.c

28.3按扇区写入Nand Flash数据

函数原型	INT32S NFlash_WriteSector(INT8U *p, INT32U sector, INT32U count)
函数功能	按扇区写入Nand Flash数据
入口参数	*p, 要写入数据的指针; sector,写入flash数据的起始扇区序号; count, 要写入扇区数量
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;失败;
注意	扇区大小为512字节, 本函数只能应用在FATFS的diskio.c中调用
使用说明	参考diskio.c

28.4 NFlash控制函数

函数原型	INT32S W25QXX_Ctrl(INT8U Cmd, INT32U Para)
函数功能	NFlash控制函数
入口参数	Cmd, 控制命令: CMD_NFLASH_SYNC: Nand Flash同步处理 CMD_NFLASH_BADBLK: Nand Flash坏块处理, 调用完NFlash_APPInit()后, // 必须调用NFlash_Ctrl(CMD_NFLASH_BADBLK, 0)处理坏块问题 CMD_NFLASH_STATUS: 读取Nand Flash状态 CMD_NFLASH_SECTOR_COUNT: 读取Nand Flash扇区数量 CMD_NFLASH_SECTOR_SIZE: 读取Nand Flash扇区大小 CMD_NFLASH_BLOCK_SIZE: 读取块大小(包含多少扇区数量) CMD_NFLASH_RDID: 读取厂商及器件ID, 返回器件ID指针,5个字节 CMD_NFLASH_FORMAT: 格式化芯片 Para，命令控制参数;
返回参数	根据命令不同, 返回不同数据; ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
使用说明	参考diskio.c

29. SD卡读写操作(sd.h)

29.1 SD卡初始化函数

函数原型	INT32S SD_Init(SD_PARA *pPara)
函数功能	SD卡初始化函数
入口参数	*pPara, 初始化参数指针
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
使用说明	根据配置文件配置如下： #define SDCARD_EN 1 // SD卡使能：1, 使能; 0, 关闭; 参考在API_Init()函数中调用SD_APPInit()实现SD卡初始化，一般客户不需要更改该函数

29.2 按扇区读取SD卡的数据

函数原型	INT8U SD_Read(INT8U *p, INT32U sector, INT8U count)
函数功能	FatFS接口写入SD卡扇区数据函数
入口参数	*p, 写入数据缓存指针; sector, 写入扇区起始序号 count, 写入扇区数量 (1..255)
返回参数	RES_OK, Successful; RES_ERROR, R/W Error; RES_WRPRT, Write Protected RES_NOTRDY, Not Ready; RES_PARERR, Invalid Parameter;
注意	扇区大小为512字节, 本函数只能应用在FATFS的diskio.c中调用;
使用说明	参考diskio.c

29.3 按扇区写入SD卡数据

函数原型	INT8U SD_Write(INT8U *p, INT32U sector, INT8U count)
函数功能	FatFS接口写入SD卡扇区数据函数
入口参数	*p, 写入数据缓存指针; sector,写入扇区起始序号;

	count,写入扇区数量 (1..255)
返回参数	RES_OK, Successful; RES_ERROR, R/W Error; RES_WRPRT, Write Protected RES_NOTRDY, Not Ready; RES_PARERR, Invalid Parameter;
注意	扇区大小为512字节, 本函数只能应用在FATFS的diskio.c中调用
使用说明	参考diskio.c

29.4 SD卡控制函数

函数原型	INT32S SD_Ctrl(INT8U Cmd, INT32U Para)
函数功能	FatFS接口SD卡控制函数
入口参数	Cmd, 控制命令, 如下: CMD_SD_SYNC: SD检测同步处理: 检测SD是否插入或拔出, 插入则打开电源 // 并初始化, 拔出则关闭电源 CMD_SD_STATUS: 读取SD卡状态 CMD_SD_SECTOR_COUNT: 读取SD卡扇区数量 CMD_SD_BLOCK_SIZE: 读取SD卡块大小 CMD_SD_SECTOR_SIZE: 读取SD卡扇区大小 CMD_SD_TYPE: 读取SD卡类型 Para, 参数,默认0,暂时未用
返回参数	根据不同命令返回不同的值 当Cmd是CMD_SD_SYNC和CMD_SD_STATUS时: 返回SD卡状态: STA_NOINIT(bit0=1): SD卡没有初始化 STA_NODISK(bit1=1): 没有发现SD卡 STA_PROTECT(bit2=1): SD卡写保护
使用说明	参考diskio.c

30. 以太网通信读写操作(net.h)

30.1 初始化函数

函数原型	INT32S NET_Init(NET_PARA *pPara)
函数功能	网络初始化函数
入口参数	*pPara, 初始化参数指针
返回参数	ERR_TRUE,操作成功; 其它值, 参见const.h中错误代码;
使用说明	根据配置文件配置如下: #define SDCARD_EN 1 // SD卡使能: 1, 使能; 0, 关闭; 参考在API_Init()函数中调用SD_APPInit()实现SD卡初始化，一般客户不需要更改该函数

30.2 将长度len的数据发送函数

函数原型	INT32S NET_Write(INT32U len)
函数功能	将长度len的数据发送函数
入口参数	len, 发送数据长度
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
使用说明	参考ethernetif.c

30.3 按扇区写入SD卡数据

函数原型	INT32S NET_Read(INT32U *p, INT16U *len)
函数功能	接收数据函数
入口参数	*p, 接收数据起始地址; len, 接收数据长度
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
使用说明	参考ethernetif.c

30.4 网络控制函数

函数原型	INT32S NET_Ctrl(INT8U Cmd, INT32U Para)
函数功能	网络控制函数
入口参数	Cmd, 控制命令: CMD_NET_SYNC：网络同步操作：检测网络断线和连接，并做相应处理,返回网络状态 CMD_NET_STATUS：返回网络状态 CMD_NET_SETMACADDR：设置MAC地址 CMD_NET_START：启动网络工作 CMD_NET_GETTXBUF：得到发送BUF地址 Para, 命令参数, 默认0 CMD_NET_GETTXBUF
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
使用说明	参考ethernetif.c

31．常用功能函数子程序

这部分函数比较简单，这里不再详述，参见subfun.h头文件。

第四章 MODBUS总线函数应用说明

1. MODBUS主机通信配置

根据配置文件配置如下：

```
#define MODBUS_EN         1          // MODBUS通信使能：1, 使能;  0, 关闭;
#define MODBUS_MODE       0          // MODBUS通信模式：0, RTU;  1, ASCII码;
#define MODBUS_CH         UART3_ID   // MODBUS通信通道：0: UART1_ID, 1: UART2_ID, 2:
                                     // UART3_ID, 3: UART4_ID, 4: UART5_ID;
#define MODBUS_ID         1          // MODBUS操作设备的通信地址ID, 默认1
#define MODBUS_TIMEOUT    1000       // MODBUS通信超时时间, 单位ms;
```

2. MODBUS主机通信函数

2.1 主机读取线圈函数

函数原型	INT32S Modbus_ReadCoils(INT8U ch, INT8U id, INT16U addr, INT16U len, INT8U *p , INT16U TimeOut)
函数功能	主机读取线圈函数，调用该函数发送读线圈请求
入口参数	ch: 通讯管道号(UART1_ID,UART2_ID,UART3_ID,UART4_ID,UART5_ID)，如果向一个非主机的管道发送请求，将返回无效管道出错 id: 从机的地址，1~255 addr：线圈起始地址，取值范围为：0x0000~0xffff len: 读取线圈个数，取值范围为：0x001~0x07d0 *p: 保存线圈值的指针，指向的地址类型为 8 位字符型。*p 字符的第0位值为第1个地址的值，第7位为第8个地址的值，第9个线圈地址的值在*(p+1)地址的第0位； TimeOut: 执行超时时间，单位ms; 超过这个时间设备没有返回, 本函数返回超时错误代码
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码
注意	本函数Modbus协议功能代码是：01, 操作输出线圈

使用说明	#define DATA_LEN 16
	INT8U buf[(DATA_LEN+7)/8];
	INT32S flag;
	flag=Modbus_ReadCoils(MODBUS_CH,MODBUS_ID,0,DATA_LEN,buf，MODBUS_TIMEOUT);
	if(flag == ERR_TRUE)
	{ // 16个线圈状态在buf中}

2.2 主机读取离散输入量函数

函数原型	INT32S Modbus_ReadDisInput(INT8U ch, INT8U id, INT16U addr, INT16U len, INT8U *p , INT16U TimeOut)
函数功能	主机读取离散输入量函数，调用该函数发送读离散输入量请求
入口参数	ch: 通讯管道号(UART1_ID,UART2_ID,UART3_ID,UART4_ID,UART5_ID)，如果向一个非主机的管道发送请求，将返回无效管道出错 id: 从机的地址，1~255

	addr：离散输入量起始地址，取值范围为：0x0000~0xffff len： 读取离散输入量个数，取值范围为：0x001~0x07d0 *p： 保存离散输入量值的指针，指向的地址类型为 8 位字符型。*p 字符的第0位值为第1个地址的值，第7位为第8个地址的值，第9个离散输入量地址的值在*(p+1)地址的的第0位； TimeOut：执行超时时间,单位ms；超过这个时间设备没有返回，本函数返回超时错误代码
返回参数	ERR_TRUE，操作成功；其它值，参见const.h中错误代码
注意	本函数Modbus协议功能代码是：02，操作离散输入量

使用说明	#define DATA_LEN 16
	INT8U buf[(DATA_LEN+7)/8];
	INT32S flag;
	flag=Modbus_ReadDisInput(MODBUS_CH,MODBUS_ID,0,DATA_LEN,buf,MODBUS_TIMEOUT);
	if(flag == ERR_TRUE)
	{ // 16个离散输入量在buf中}

2.3 主机读取保持寄存器函数

函数原型	INT32S Modbus_ReadHoldReg(INT8U ch, INT8U id, INT16U addr, INT16U len, INT16U *p , INT16U TimeOut)
函数功能	主机读取保持寄存器函数，调用该函数发送读保持寄存器请求
入口参数	ch： 通讯管道号(UART1_ID,UART2_ID,UART3_ID,UART4_ID,UART5_ID)，如果向一个非主机的管道发送请求，将返回无效管道出错 id： 从机的地址，1~255 addr：保持寄存器起始地址，取值范围为：0x0000~0xffff len： 读取保持寄存器个数，取值范围为：0x01~0x07d *p： 保存保持寄存器值的指针，指向的地址类型为 16 位无符号整型； p 指向读出的第1个寄存器值存放的地址 TimeOut：执行超时时间,单位ms；超过这个时间设备没有返回，本函数返回超时错误代码
返回参数	ERR_TRUE，操作成功；其它值，参见const.h中错误代码
注意	本函数Modbus协议功能代码是：03，操作保持寄存器

使用说明	#define DATA_LEN 16
	INT16U buf[DATA_LEN];
	INT32S flag;
	flag =
	Modbus_ReadHoldReg(MODBUS_CH,MODBUS_ID,0,DATA_LEN,buf,MODBUS_TIMEOUT);
	if(flag == ERR_TRUE)
	{ // 16个保持寄存器值在buf中}

2.4 主机读取输入寄存器函数

函数原型	INT32S Modbus_ReadInputReg(INT8U ch, INT8U id, INT16U addr, INT16U len, INT16U *p , INT16U TimeOut)
函数功能	主机读取输入寄存器函数，调用该函数发送读输入寄存器请求
入口参数	ch: 通讯管道号(UART1_ID,UART2_ID,UART3_ID,UART4_ID,UART5_ID)，如果向一个非主机的管道发送请求，将返回无效管道出错 id: 从机的地址，1~255 addr：输入寄存器的起始地址，取值范围为：0x0000~0xffff len: 读取输入寄存器个数，取值范围为：0x01~0x07d *p: 保存输入寄存器值的指针，指向的地址类型为 16 位无符号整型; p 指向读出的第1个寄存器值存放的地址 TimeOut: 执行超时时间,单位ms; 超过这个时间设备没有返回, 本函数返回超时错误代码
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码
注意	本函数Modbus协议功能代码是：04, 操作输入寄存器

使用说明	#define DATA_LEN 16
	INT16U buf[DATA_LEN];
	INT32S flag;
	flag =
	Modbus_ReadInputReg (MODBUS_CH,MODBUS_ID,0,DATA_LEN,buf,MODBUS_TIMEOUT);
	if(flag == ERR_TRUE)
	{ // 16个输入寄存器值在buf中}

2.5 主机写单个线圈函数

函数原型	INT32S Modbus_WriteSingleCoil(INT8U ch, INT8U id, INT16U addr, INT8U val)
函数功能	主机写单个线圈函数，调用该函数发送写单个线圈请求
入口参数	ch: 通讯管道号(UART1_ID,UART2_ID,UART3_ID,UART4_ID,UART5_ID)，如果向一个非主机的管道发送请求，将返回无效管道出错 id: 从机的地址，1~255 addr：写入线圈的地址，取值范围为：0x0000~0xffff val: 写入线圈的值：0, 关闭线圈; 1, 打开线圈 TimeOut: 执行超时时间,单位ms; 超过这个时间设备没有返回, 本函数返回超时错误代码
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码
注意	本函数Modbus协议功能代码是：05, 操作输出线圈

使用说明	#define DATA_LEN 16
	INT32S flag;
	INT16U i;
	for (i=0; i<DATA_LEN; i++) // 从地址0开始设置16个线圈为1
	{ flag = Modbus_WriteSingleCoil(MODBUS_CH,MODBUS_ID, i, 1,MODBUS_TIMEOUT);
	if(flag == ERR_TRUE)
	{ // 操作成功 }
	}

2.6 主机写单个保持寄存器函数

函数原型	INT32S Modbus_WriteSingleReg(INT8U ch, INT8U id, INT16U addr, INT16U val , INT16U TimeOut)
函数功能	主机写单个保持寄存器函数，调用该函数发送写单个保持寄存器请求
入口参数	ch： 通讯管道号(UART1_ID,UART2_ID,UART3_ID,UART4_ID,UART5_ID)，如果向一个非主机的管道发送请求，将返回无效管道出错 id： 从机的地址，1~255 addr：写入保持寄存器的地址，取值范围为：0x0000~0xffff val： 写入保持寄存器的值，取值范围为：0x0000~0xffff TimeOut: 执行超时时间,单位ms; 超过这个时间设备没有返回, 本函数返回超时错误代码
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码
注意	本函数Modbus协议功能代码是：06, 操作保持寄存器

使用说明	#define DATA_LEN 16
	INT16U i;
	INT32S flag;
	for (i=0; i<DATA_LEN; i++)
	{ flag = Modbus_WriteSingleReg(MODBUS_CH,MODBUS_ID, i, i,MODBUS_TIMEOUT);
	if(flag == ERR_TRUE)
	{ // 操作成功 }
	}

2.7 主机写多个线圈函数

函数原型	INT32S Modbus_WriteMulCoils(INT8U ch, INT8U id, INT16U addr, INT16U len, INT8U *p , INT16U TimeOut)
函数功能	主机写多个线圈函数，调用该函数发送写多个线圈请求
入口参数	ch： 通讯管道号(UART1_ID,UART2_ID,UART3_ID,UART4_ID,UART5_ID)，如果向一个非主机的管道发送请求，将返回无效管道出错 id： 从机的地址，1~255 addr：线圈起始地址，取值范围为：0x0000~0xffff len： 需写入线圈的个数，取值范围为：0x01~0x07b0 *p： 写入线圈数据缓冲的指针，指向的地址类型为8位字符型。*p 字符的第0位值为第1个地址的值，第7位为第8个地址的值，第9个线圈地址的值在*(p+1)地址的第0位 TimeOut: 执行超时时间,单位ms; 超过这个时间设备没有返回, 本函数返回超时错误代码
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码
注意	本函数Modbus协议功能代码是：15, 操作输出线圈
使用说明	#define DATA_LEN 16 INT8U buf[(DATA_LEN+7)/8]; INT32S flag; INT16U i;

	for(i=0;i<((DATA_LEN+7)/8); i++)
	{ buf[i] = 0;}
	flag =
	Modbus_WriteMulCoils(MODBUS_CH,MODBUS_ID,0,DATA_LEN,buf,MODBUS_TIMEOUT);
	if(flag == ERR_TRUE)
	{ // 操作成功}

2.8 主机写多个寄存器函数

函数原型	INT32S Modbus_WriteMulReg(INT8U ch, INT8U id, INT16U addr, INT16U len, INT16U *p , INT16U TimeOut)
函数功能	主机写多个寄存器函数，调用该函数发送写多个寄存器请求
入口参数	ch: 通讯管道号(UART1_ID,UART2_ID,UART3_ID,UART4_ID,UART5_ID)，如果向一个非主机的管道发送请求，将返回无效管道出错 id: 从机的地址，1~255 addr：保持寄存器起始地址，取值范围为：0x0000~0xffff len: 需写入保持寄存器的个数，取值范围为：0x01~0x078 *p: 需写入保持寄存器的数据缓冲区的指针，指针类型为16位 TimeOut: 执行超时时间,单位ms; 超过这个时间设备没有返回, 本函数返回超时错误代码
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码
注意	本函数Modbus协议功能代码是：16, 操作保持寄存器

使用说明	#define DATA_LEN 16
	INT16U i, buf[DATA_LEN];
	INT32S flag;
	for(i=0;i<DATA_LEN; i++)
	{ buf[i] = 0;}
	flag =
	Modbus_WriteMulReg(MODBUS_CH,MODBUS_ID,0,DATA_LEN,buf,MODBUS_TIMEOUT);
	if(flag == ERR_TRUE)
	{ // 操作成功);}

2.9 主机读写多个寄存器函数

函数原型	INT32S Modbus_ReadWriteMulReg(INT8U ch, INT8U id, INT16U waddr, INT16U wlen, INT16U raddr, INT16U rlen, INT16U *p , INT16U TimeOut)
函数功能	主机读写多个寄存器函数，调用该函数读和写多寄存器请求。执行该函数时，先写入后读出
入口参数	ch: 通讯管道号(UART1_ID,UART2_ID,UART3_ID,UART4_ID,UART5_ID)，如果向一个非主机的管道发送请求，将返回无效管道出错 id: 从机的地址，1~255 waddr：写入保持寄存器的起始地址，取值范围为：0x0000~0xffff wlen: 读出保持寄存器的个数，取值范围为：1~121 raddr：写入保持寄存器的起始地址，取值范围为：0x0000~0xffff rlen: 读出保持寄存器的个数，取值范围为：1~125

	*p: 需写入保持寄存器的数据缓冲区指针，同时也是读出寄存器数据存放缓冲区的指针 TimeOut: 执行超时时间,单位ms; 超过这个时间设备没有返回, 本函数返回超时错误代码
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码
注意	本函数Modbus协议功能代码是：23, 操作保持寄存器

使用说明	#define DATA_LEN 16
	INT16U i, buf[DATA_LEN];
	INT32S flag;
	for(i=0;i<DATA_LEN; i++)
	{ buf[i] = i;}
	flag = Modbus_ReadWriteMulReg(MODBUS_CH,MODBUS_ID,0,DATA_LEN,0,DATA_LEN,buf,MODBUS_TIMEOUT);
	if(flag == ERR_TRUE)
	{ // 操作成功}

3. MODBUS从机通信配置

根据配置文件配置如下：

```
#define MODBUS_SLAVE_EN          1       // MODBUS 从机通信使能：1, 使能;  0, 关闭;
#define MODBUS_SLAVE_MODE        0       // MODBUS 从机通信模式：0, RTU;  1, ASCII 码;
#define MODBUS_SLAVE_CH    UART4_ID     // MODBUS 从机通信通道：0: UART1_ID, 1: UART2_ID,
                                        // 2: UART3_ID, 3: UART4_ID, 4: UART5_ID;
#define MODBUS_SLAVE_ID          1       // MODBUS 从机通信地址码, 范围：1~255;
#define MODBUS_COILS_BASEADDR      10000  // 线圈寄存器基地址;
#define MODBUS_DISINPUT_BASEADDR   10000  // 离散输入量寄存器基地址;
#define MODBUS_HOLDREG_BASEADDR    10000  // 保持寄存器基地址;
#define MODBUS_INPUTREG_BASEADDR   10000  // 输入寄存器基地址;
#define MODBUS_MAX_COILS      32  // MODBUS 从机最大线圈数量(读写,可用功能码:1,5,15);
#define MODBUS_MAX_DISINPUT   32  // MODBUS 从机最大离散输入量(只读,可用功能码:2);
#define MODBUS_MAX_HOLDREG    16  // MODBUS 从机最大保持寄存器(读写,可用功能
                                  // 码:3,6,16,23)数量;
#define MODBUS_MAX_INPUTREG   16  // MODBUS 从机最大输入寄存器(只读,可用功能码:4)数量;
```

4. MODBUS从机通信函数

函数原型	INT32S Modbus_Proc(INT8U ch, INT8U id, INT8U *p, INT16U len, MODBUS_PARA *pPara)
函数功能	MODBUS从机通信指令解析函数
入口参数	ch: 通信通道: UART1_ID~UART5_ID(0-4), MODBUS_TCP_ID(10); id: 本机设备ID(总线地址), 范围:1-250; *p: 数据指针; len: 数据长度 *pPara: Modbus从机模式参数;
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;

使用说明
 	
具体请查看TaskModbus.c文件
	INT32S flag;
	INT16U rUartLen, len;
	INT8U buf[64];
	len = Uart_Ctrl(MODBUS_SLAVE_CH, CMD_UART_GetCharsRxBuf,0);// 读取接收数据长度
	if ((len == rUartLen)&&(len>0))
	{ Uart_Read(MODBUS_SLAVE_CH, buf, len); // 读取接收数据到buf
	flag = Modbus_Proc(MODBUS_SLAVE_CH, MODBUS_SLAVE_ID, buf, len, (MODBUS_PARA *)&ModbusPara.Flag); // Modbus数据处理
	if (flag == ERR_TRUE)
	{// Modbus协议解析成功 }
	rUartLen -= len;
	}
	else{ rUartLen = len;}

附录：驱动库更新内容说明

序号	版本	时间	更新内容
1	V1.10	2019.3.5	正式发布。 **注：该版本驱动库是在原有V1.04版本基础上进行大幅升级而来；功能更多，适应面更广，更稳定，使用更加方便；**