

AMKN 驱动库使用手册

(2025 年 12 月 1 日修订版)

版权声明

本产品使用手册包含的所有内容均受版权法的保护，未经北京中嵌凌云电子有限公司的书面授权，任何组织和个人不得以任何形式或手段对整个手册和部分内容进行复制和转载。

免责声明

本文档并未授予任何知识产权的许可，并未以明示或暗示，或以禁止发言或其它方式授予任何知识产权许可。除在其产品的销售条款和条件声明的责任之外，我司概不承担其他责任。并且我司对本产品的销售和使用不作任何明示或暗示的担保，包括对产品特定用途的适用性，适销性或对任何专利权、版权或其他知识产权的侵权责任等均不作担保。我司对文档中包含的文字、图片及其它内容的准确性和完整性不承担任何法律或非法律责任，我司可能随时会对产品描述和相关的功能调整或技术改进，保留修改文档中任何内容的权利，恕不另行通知。

商标声明

AMKN 系北京中嵌凌云电子有限公司注册商标，未经书面授权，任何人不得以任何方式使用该商标、标记。

销售及服务网络

北京

销售电话：185 0042 1002

地 址：北京市海淀区吴家场路 1 号院 2 号楼

邮 箱：sales@embedarm.com

西安

销售电话：029-6888 8268（工作日）

手 机：189 9285 2102

地 址：西安市曲江新区旺座曲江 H 座 3003 室

邮 箱：sales@embedarm.com

技术支持：

电 话：029-8877 2044（工作日）

手 机：188 0108 0298

微 信：133 9928 8868

邮 箱：embedarm@126.com

网 址：www.zqlymcu.com

版本

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. 20	正式发布版本	2022. 4. 1	研发部
V1. 21	修改部分文档	2022. 6. 10	研发部
V1. 20. 06	修改部分文档, 文档版本和驱动库版本保持一至, 方便识别	2022. 11. 15	研发部
V1. 30. 00	增加STM32H7XX和STM32MP157系列模块, 修改部分内容; 适用于AMKN软件版本: V1. 30. XX	2025. 7. 1	研发部
V1. 31. 01	修改部分文档	2025. 12. 1	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、STM32F107VC、STM32F407VE、STM32F407ZE、GD32F303VE、GD32F303ZE、GD32F307VE、STM32H723ZG、STM32H743XI、STM32H750XB、STM32MP157	
2	工业控制板	AMKN8602、AMKN8602G、AMKN8612、AMKN8626、AMKN8628、AMKN8629、AMKN8630、AMKN8638、AMKN8639、AMKN8802、AMKN8701G、AMKN8702G、AMKN8703、RTU-BUS、RTU-6XXX系列	

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。

目录

1. AMKN 驱动库简介	5
2. 函数列表:	5
3. 系统及驱动库初始化函数 (SYSINT.H)	15
4. 延时函数 (DELAY.H)	18
5. 独立看门狗函数 (IWDG.H)	19
6. CRC16 校验函数 (CRC.H)	21
7. RTC 读写操作 (RTC.H)	21
8. IO 读写操作 (GPIO.H)	24
9. 外部 IO 中断函数 (EXTI.H)	28
10. UART 通信操作 (UART.H)	31
11. SPI 读写操作 (SPI.H)	35
12. I ² C 读写操作 (I ² C.H)	40
13. CAN 总线读写操作 (CAN.H)	42
14. ADC 采集操作 (ADC.H)	50
15. DAC 输出操作 (DAC.H)	53
16. BKP 备份寄存器读写 (BKP.H)	55
17. FLASH 读写操作 (FLASH.H)	57
18. IAP 固件更新操作 (IAP.H)	59
19. PWM 输出操作 (TIMER.H)	61
20. FCLK 脉冲输入操作 (TIMER.H)	64
21. 定时器操作 (TIMER.H)	67
23. USB 设备接口 (USBDEVICE.H)	71
24. USB 主机接口 (USBHOST.H)	74
25. EEPROM 读写操作 (EEPROM.H)	76
26. AT45DBXX 读写操作 (AT45DBXX.H)	78
27. W25QXX 读写操作 (W25QXX.H)	81
28. NAND FLASH 读写操作 (NFLASH.H)	85
29. SD 卡读写操作 (SD.H)	87
30. 以太网通信读写操作 (NET.H)	89
31. 常用功能函数子程序	90
32. DMA 初始化函数	90
33. MODBUS 主机通信函数	91
34. MODBUS 协议处理函数	102
34. LPTIMER 定时器操作 (LPTIMER.H)	103
35. SDRAM 操作 (SDRAM.H)	106
36. QSPI 读写操作 (QSPI.H)	108
37. LTDC 控制操作 (LTDC.H)	110
38. DMA2D 控制操作 (DMA2D.H)	114
39. MDMA 控制操作 (MDMA.H)	119
40. JPEG 控制操作 (JPEG.H)	120
41. IPCC 异和通信 (IPCC.H)	121
附录: 驱动库更新内容说明	123

1. AMKN驱动库简介

AMKN驱动库是北京中嵌凌云针对STM32和GD32系列MCU独立开发的第三方驱动库。相对于ST和GD官方驱动库具有如下特点：精简高效、使用简单、兼容性好等特点，非常适合快速开发软件。

2. 函数列表：

这些函数都在Libraries文件夹下，函数分类列表如下：

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
		INT32S SysLib_Register(INT8U Type, void *pPara)	系统注册函数
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(INT32U IOx, INT16U val)	写入IO输出值
		INT32S IO_Ctrl(INT32U IOx, INT8U Cmd, INT32U Para)	IO命令控制
		IO_ReadBit(IOx) 注：STM32H7XX和STM32MP15X不支持本函数	采用位带操作, 读取IO输入值；
		IO_WriteBit(IOx, val) 注：STM32H7XX和STM32MP15X不支持本函数	采用位带操作, 写入IO输出值；
3	exti.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)	RTC读取时间
		INT32S RTC_Write(RTC_TIME *rtc)	RTC设置时间
		INT32S 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
		INT32S Uart_Init(INT8U id, UART_PARA *pPara)	Uart初始化函数
		INT32S Uart_Read(INT8U id, INT8U *p, INT16U len)	接收一段数据

6	uart.h	INT32S Uart_Write(INT8U id, INT8U *p, INT16U len)	发送一段数据
		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	I ² C.h	INT32S I ² C_Init(INT8U I ² Cx, I ² C_PARA *pPara)	I ² C初始化函数
		INT32S I ² C_Read(INT8U id, INT16U I ² CAddr, INT16U addr, INT8U *p, INT16U len)	I ² C读数据函数
		INT32S I ² C_Write(INT8U id, INT16U I ² CAddr, INT16U addr, INT8U *p, INT16U len)	I ² C写数据函数
		INT32S I ² C_Ctrl(INT8U id, INT8U Cmd, INT32U Para)	I ² C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_ReadWrite(INT8U id, INT8U *pTX, INT8U *pRX, INT32U len)	SPI读写多字节函数
		INT32S SPI_DMABWrite(INT8U id, INT8U *p, INT32U len, SPI_DMA_PARA *pPara)	SPI总线利用DMA写数据函数
		INT32S SPI_DMABRead(INT8U id, INT8U *p, INT32U len, SPI_DMA_PARA *pPara)	SPI总线利用DMA读数据函数
		INT32S SPI_DMABReadWrite(INT8U id, INT8U *pTX, INT8U *pRX, INT32U len, SPI_DMA_PARA *pPara)	SPI总线利用DMA读写数据函数
		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_FILTER_PARA *pFilter)	CAN初始化函数 STM/GD32F1/3/4XX支持
		INT32S CAN_Init(INT8U id, CAN_PARA *pPara)	CAN初始化函数 STM32H7XX/MP15X支持
		INT32S CAN_FilterInit(CAN_FILTER_PARA *pFilter)	CAN滤波器初始化函数 STM/GD32F1/3/4XX支持
		INT32S CAN_FilterInit(INT8U id, CAN_FILTER_PARA *pFilter)	CAN滤波器初始化函数 STM32H7XX/MP15X支持
		INT32S CAN_Write(INT8U id, CAN_TX_MSG *pTxMsg, INT16U Num)	CAN发送多组数据函数
		INT32S CAN_Read(INT8U id, CAN_RX_MSG *pRxMsg,	CAN读多组数据函数

		INT16U Num)	
		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 STM/GD32F1/3/4XX支持
		INT32S ADC_Read(INT8U id, INT32S *p, INT8U len)	STM32H7XX/MP15X支持
		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 PWM_Write(INT8U id, PWM_WRITE_PARA *pPara);	PWM写入控制函数
		INT32S PWM_MulCtrl(INT32U IDFlag, INT8U Cmd)	多路PWM同时输出控制
		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(INT16U addr, INT16U *p, INT16U len)	备份寄存器写入函数
		INT32S BKP_Read(INT16U addr, 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(INT32U val)	毫秒延时函数
		void Delay_s(INT16U val)	秒延时函数
18	USBDevice.h	INT32S USBD_Init(INT8U id, USBD_PARA *pPara)	USB设备初始化函数
		INT32S USBD_Read(INT8U id, INT8U *p, INT16U len)	USB设备读数据函数
		INT32S USBD_Write(INT8U id, INT8U *p, INT16U len)	USB设备写数据函数
		INT32S USBD_Ctrl(INT8U id, INT8U Cmd, INT32U Para)	USB设备控制函数
19	USBHost.h	INT32S USBH_Init(INT8U id, USBH_PARA *pPara)	USB主机初始化函数
		INT32S USBH_Ctrl(INT8U id, INT8U Cmd, INT32U Para)	USB主机控制函数
		INT8U UDisk_Read(INT8U pdrv, INT8U *p, INT32U sector, INT8U count)	USB主机读取U盘数据函数
		INT8U UDisk_Write(INT8U pdrv, INT8U *p, 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 STM/GD32F1/3/4XX支持
		INT32S NET_Write(NET_TxPacket *p)	发送数据函数 STM32H7XX支持
		INT32S NET_Read(INT32U *p, INT16U *len)	接收数据函数 STM/GD32F1/3/4XX支持
		INT32S NET_Read(NET_RxPacket *pRxPacket)	接收数据函数 STM32H7XX支持
		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 *p, INT32U sector, INT8U count)	SD卡读数据函数
		INT8U SD_Write(INT8U *p, 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_Init(MODBUS_INIT_PARA *pPara)	Modbus主机初始化函数
		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_WriteSingleHoldReg(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_WriteCoils(INT8U ch, INT8U id, INT16U addr, INT16U len, INT8U *p, INT16U TimeOut)	Modbus主机写多个线圈函数
		INT32S Modbus_WriteHoldReg(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_ReadWriteHoldReg(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_Ctrl(INT8U ch, INT8U Cmd, INT32U Para)	Modbus主机接口控制函数
		void Modbus_RxProc(INT8U ch, INT8U *p, INT16U len)	Modbus主机接口接收数据处理函数
28	subfun.h	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
		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	字符串对比

		len)	
		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
		INT32U StrCopy(INT8U *pDst, INT8U *pSrc)	单字符串拷贝函数, 增加返回拷贝字符个数
		INT32U StrCopy2(INT8U *pDst, INT8U *pSrc1, INT8U *pSrc2)	2个字符串拷贝函数, 增加返回拷贝字符个数
		INT32U StrCopy3(INT8U *pDst, INT8U *pSrc1, INT8U *pSrc2, INT8U *pSrc3)	3个字符串拷贝函数, 增加返回拷贝字符个数
		INT32U StrCopy4(INT8U *pDst, INT8U *pSrc1, INT8U *pSrc2, INT8U *pSrc3, INT8U *pSrc4)	4个字符串拷贝函数, 增加返回拷贝字符个数
		INT32U StrCopy5(INT8U *pDst, INT8U *pSrc1, INT8U *pSrc2, INT8U *pSrc3, INT8U *pSrc4, INT8U *pSrc5)	5个字符串拷贝函数, 增加返回拷贝字符个数
29	dma.h	INT32S DMA_Init(INT8U id, DMA_INIT *pPara)	DMA初始化函数
30	lptimer.h	INT32S LPTimer_Init(INT8U id, LPTIM_PARA *pPara)	LPTimer初始化函数
		INT32S LPTimer_Ctrl(INT8U id, INT8U Cmd, LPTIM_CTRL *pPara)	LPTimer控制操作函数
31	sdram.h	INT32S SDRAM_Init(SDRAM_PARA *pPara)	SDRAM初始化函数
		void SDRAM_Write8b(INT32U addr, INT8U *p, INT32U len)	向SDRAM写入8位数据
		void SDRAM_Write16b(INT32U addr, INT16U *p, INT32U len)	向SDRAM写入16位数据
		void SDRAM_Write32b(INT32U addr, INT32U *p, INT32U len)	向SDRAM写入32位数据

		void SDRAM_Read8b(INT32U addr, INT8U *p, INT32U len)	从SDRAM读取8位数据
		void SDRAM_Read16b(INT32U addr, INT16U *p, INT32U len)	从SDRAM读取16位数据
		void SDRAM_Read32b(INT32U addr, INT32U *p, INT32U len)	从SDRAM读取32位数据
32	qspi.h	INT32S QSPI_Init(QSPI_PARA *pPara)	QSPI初始化函数
		INT32S QSPI_Write(INT32U addr, INT8U *p, INT32U len)	QSPI总线写数据函数
		INT32S QSPI_Write(INT32U addr, INT8U *p, INT32U len)	QSPI总线写数据函数
		INT32S QSPI_Ctrl(INT8U Cmd, INT32U Para)	QSPI命令控制函数
33	ltdc.h	INT32S LTDC_Init(LTDC_PARA *pPara)	LTDC初始化函数
		void LCD_DrawPixel(INT8U layer, INT16U x, INT16U y, INT32U Color)	画1个像素点
		void LCD_DrawLine(INT8U layer, INT16U x1, INT16U y1, INT16U x2, INT16U y2, INT32U Color)	画1条直线
		void LCD_DrawRect(INT8U layer, INT16U x, INT16U y, INT16U Width, INT16U Heigh, INT32U Color)	画水平放置的矩形
		void LCD_DrawCircle(INT8U layer, INT16U x, INT16U y, INT16U Radius, INT16U Color)	画圆
		void LCD_DrawPoints(INT8U layer, INT16U *x, INT16U *y, INT16U Size, INT32U Color)	绘制一组点,并将这些点连接起来,可用于波形显示
		void LCD_DrawImage(INT8U layer, INT16U x, INT16U y, INT16U Width, INT16U Heigh, INT32U PixelFormat, void *pColor)	画图像
		void LCD_FillRect(INT8U layer, INT16U x, INT16U y, INT16U Width, INT16U Heigh, INT32U Color)	用一个颜色值填充一个矩形
		void LCD_ClearScreen(INT8U layer, INT32U Color)	用一个颜色清屏
		void LTDC_SetEnterWindow(INT8U layer, INT16U x, INT16U y, INT16U Width, INT16U Heigh)	设置窗口模式,进入窗口显示模式
		void LTDC_QuitWindow(INT8U layer)	退出窗口显示模式
		INT32S LTDC_Ctrl(INT8U Cmd, INT8U Layer, INT32U	LTDC命令控制

		Para)	
34	dma2d. h	INT32S DMA2D_Init(DMA2D_PARA *pPara)	DMA2D初始化函数
		INT32S DMA2D_CopyBuffer(INT16U xSize, INT16U ySize, INT32U SrcAddr, INT16U SrcLineOffset, INT32U DstAddr, INT16U DstLineOffset, INT32U PixelFormat)	从前景层复制指定区域的图像数据到目标区域
		INT32S DMA2D_CopyBufferPFC(INT16U xSize, INT16U ySize, INT32U SrcAddr, INT16U SrcLineOffset, INT32U SrcPixelFormat, INT32U DstAddr, INT16U DstLineOffset, INT32U DstPixelFormat)	从前景层复制指定区域的图像数据到目标区域并进行PFC转换
		INT32S DMA2D_CopyBufferMixPFC(INT16U xSize, INT16U ySize, INT32U SrcAddr, INT16U SrcLineOffset, INT32U SrcPixelFormat, INT32U DstAddr, INT16U DstLineOffset, INT32U DstPixelFormat, INT32U Alpha)	从前景层复制指定区域的图像数据到目标区域并进行PFC转换及和目标区域颜色进行混合
		INT32S DMA2D_AlphaBlendingBulk(INT16U xSize, INT16U ySize, INT32U SrcFGAddr, INT16U SrcFGLineOffset, INT8U SrcFGPixelFormat, INT32U SrcBGAddr, INT16U SrcBGLineOffset, INT8U SrcBGPixelFormat, INT32U DstAddr, INT16U DstLineOffset, INT8U DstPixelFormat)	前景层和背景层作为DMA2D输入, 且支持颜色格式转换和颜色混合
		INT32S DMA2D_Fill(INT16U xSize, INT16U ySize, INT32U DstAddr, INT16U DstLineOffset, INT32U ColorIndex, INT32U PixelFormat)	颜色填充
		INT32S DMA2D_Ctrl(INT8U Cmd, INT32U Para)	DMA2D命令控制
35	mdma. h	INT32S MDMA_Init(MDMA_PARA *pPara)	MDMA初始化
		INT32S MDMA_ChxInit(INT8U ch, MDMA_CH_PARA *pPara)	MDMA通道初始化
		INT32S MDMA_StartTransfer(INT8U ch, INT32U SrcAddress, INT32U DstAddress, INT32U BlockDataLength, INT32U BlockCount, INT8U isWait, INT32U TimeOut);	MDMA启动传输
		INT32S MDMA_Ctrl(INT8U ch, INT8U Cmd, void *pPara)	MDMA命令控制
36	jpeg. h	INT32S JPEG_Init(JPEG_PARA *pPara)	JPEG初始化
		INT32S JPEG_SatrtDecode(INT32U SrcAddress, INT32U SrcDataLength, INT32U DestAddress, INT32U isWait,	JPEG启动解码

		INT32U TimeOut)	
		INT32S JPEG_Ctrl(INT8U Cmd, void *pPara)	JPEG命令控制
37	ipcc.h	INT32S IPCC_Init(void)	IPCC初始化函数
		INT32S IPCC_Ctrl(INT8U Cmd, INT8U ch, INT8U dir, void *pPara)	IPCC命令控制

3. 系统及驱动库初始化函数(sysint.h)

3.1 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	<pre>// SYSLIB_PARA参数结构 typedef struct { INT32U Flag; // 标志 INT8U ModuleClass; // 模块类型 INT8U ModuleType; // 模块型号 INT8U DebugUart; // 选择printf函数输出Uart INT32U VectorTableAddr; // 中断向量表地址, 改为32位 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 INT16U OSTick; // 设置操作系统定时时间, 单位ms } SYSLIB_PARA;</pre> 参考..\libapp\libapp.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)ModuleClass: 设置模块类型定义 (3)ModuleType: 设置模块型号定义 (4)DebugUart: 设置调试信息输出串口(UART)选择 (5)VectorTableAddr: 设置中断向量表地址 (6)OscClk: 设置外部或内部时钟晶振频率 (7)SysClk: 设置系统时钟频率 (8)Tick: 设置系统定时器定时时间, 单位ms (9)OSTick: 设置操作系统定时时间, 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, 系统复位, Para默认是0 CMD_SYSLIB_READ_VERSION, 读取驱动库版本号, Para默认是0 CMD_SYSLIB_READ_DATE, 读取驱动库编译生成日期, Para默认是0 CMD_MCO1/ CMD_MCO2: MCO1/2输出控制, Para参看: MCO1/2参数定义 CMD_SYSLIB_READ_MCUID: 读取MCU唯一的ID值, 96位, 12字节, Para默认是0 CMD_SYSLIB_READ_BOOT: 读取启动模式: 只有STM32MP15X支持此功能, Para默认是0 Para: 控制参数
返回参数	不同命令返回不同参数, 参考使用说明
使用说明	<pre> INT32S ver; INT8U *p; // 读取驱动库版本号 ver = SysLib_Ctrl (CMD_SYSLIB_READ_VERSION, 0); printf("*\r\n* 驱动库版本: %d.%d.%d.%d\r\n", ver/10000, (ver%10000)/1000, (ver%1000)/100, (ver%100)/10, ver%10); // 读取驱动库生成日期 p = (INT8U *)SysLib_Ctrl (CMD_SYSLIB_READ_DATE, 0); // 读取驱动库生成日期 printf("LibDate: %s\r\n", p); // 读取MCU唯一的ID值: 12字节, 返回数据指针 p = (INT8U *)SysLib_Ctrl (CMD_SYSLIB_READ_MCUID, 0); printf("MCUID: %02X %02X %02X %02X %02X %02X %02X %02X %02X %02X %02X\r\n", p[0], p[1], p[2], p[3], p[4], p[5], p[6], p[7], p[8], p[9], p[10], p[11]); // 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); </pre>

3.3 系统注册

函数原型	INT32S SysLib_Register(INT8U Type, void *pPara)
函数功能	系统注册函数，注册一些中断和初始化处理函数
入口参数	Type: 注册函数类型，定义如下: <pre> #define REGISTER_IO_TYPE 0 // IO函数类型 #define REGISTER_UART_TYPE 1 // UART函数类型 #define REGISTER_CAN_TYPE 2 // CAN函数类型 #define REGISTER_TIM_TYPE 3 // TIM函数类型 #define REGISTER_FCLK_TYPE 4 // FCLK函数类型 #define REGISTER_PWM_TYPE 5 // PWM函数类型 #define REGISTER_ADC_TYPE 7 // ADC函数类型 #define REGISTER_DAC_TYPE 8 // DAC函数类型 #define REGISTER_RTC_TYPE 9 // RTC函数类型 #define REGISTER_EXTI_TYPE 10 // EXTI函数类型 #define REGISTER_DMA_TYPE 11 // DMA函数类型 #define REGISTER_OS_TYPE 12 // OS函数类型 #define REGISTER_SPI_TYPE 13 // SPI函数类型 #define REGISTER_USBD_TYPE 14 // USBD函数类型 #define REGISTER_USBH_TYPE 15 // USBH函数类型 #define REGISTER_I2C_TYPE 16 // I2C函数类型 #define REGISTER_NET_TYPE 17 // NET函数类型 #define REGISTER_LPTIM_TYPE 18 // LPTIM函数类型 #define REGISTER_QSPI_TYPE 19 // QSPI函数类型 #define REGISTER_LTDC_TYPE 20 // LTDC函数类型 </pre> *pPara: 具体注册函数体
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UART为例，说明如下 // UART_REGISTER数据结构 <pre> typedef struct { void (*InitCallback)(INT8U id, UARTx_TypeDef *pPara); //初始化回调函数 void (*IRQHandler)(INT8U id, INT32U Flag, INT32U Para); // 中断函数 }UART_REGISTER; void Uart_InitCallback(INT8U id, UARTx_TypeDef *pPara) { // 初始化回调函数 } void Uart_IRQHandler(INT8U id, INT32U Flag, INT32U Para) { // 中断处理函数 } UART_REGISTER pPara; </pre>

```
pPara.InitCallback = Uart_InitCallback;  
pPara.IRQHandler = Uart_IRQHandler;  
SysLib_Register(REGISTER_UART_TYPE, &pPara); // 注册函数
```

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	微妙延时函数
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; 强烈建议大于1000us请用Delay_ms延时函数; 这个延时不会十分准确;
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(INT32U val)
函数功能	毫秒延时函数
入口参数	val: 延时时间, 范围: 0~0xFFFFFFFF ms
返回参数	无
注意	在有操作系统时且val大于等于1ms会利用操作系统延时函数, 小于1ms利用软件延时; 这个延时不会十分准确;
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: <pre>#define IWDG_EN 1 // 内部看门狗使能, 1: 打开使能, 0: 关闭 #define IWDG_TIME 3000 // 看门狗时间设定, 设置范围: 200~26000ms</pre> 在../libapp/API_Init() 函数中初始化例程 <pre>#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</pre>

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: CMD_IWDG_ENA: 打开看门狗; CMD_IWDG_CLEAR: 清除看门狗定时器, 即喂狗;
返回参数	无
使用说明	<pre>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(IWDG_CLEAR); // 系统不会复位, 如果超过 #endif // 1000ms则系统会复位 } }</pre>

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	<pre> 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); </pre>

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: <pre> #define RTC_EN 1 // RTC使能, 1: 打开使能, 0: 关闭 </pre> 参考在../libapp/rtc_app.c中RTC_AppInit()应用初始化如下: <pre> 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 </pre>

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	<pre> INT32S flag; RTC_TIME rtc; rtc.year = 12; rtc.month = 12; rtc.day = 31; rtc.hour = 23; rtc.minute = 59; rtc.second = 30; rtc.ss = 0; flag = RTC_Write(&rtc); if (flag == ERR_TRUE) { //设置时间成功} </pre>

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	<pre> 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);} </pre>

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(I046)输出秒脉冲 CMD_RTC_SET_ALARM, 设置PC13(I046)输出闹钟脉冲 CMD_RTC_SET_C00, 设置PC13(I046)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../libapp/rtc_app.c中RTC_Applnit()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 (支持 STM32F1XX/GD32F3XX 模块) IO_SPEED_2MHz, 最大速度 2MHz (支持 STM32F1XX/4XX/GD32F3XX 模块) IO_SPEED_50MHz, 最大速度 50MHz (支持 STM32F1XX/4XX/GD32F3XX 模块) IO_SPEED_120MHz, 最大速度 120MHz (支持 GD32F3XX 模块) IO_SPEED_25MHz, 最大速度 25MHz, (支持 STM32F4xx 模块) IO_SPEED_100MHz, 最大速度 100MHz (支持 STM32F4xx 模块) STM32H7XX/MP15X 模块支持以下速度 IO_SPEED_LOW、IO_SPEED_MEDIUM、IO_SPEED_HIGH、IO_SPEED_VERY_HIGH
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../libapp/io_app.c中IO_Applnit()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, IO_INPUT); 内部上拉输入: IO_Init(PA0, IO_IN_IPU, IO_INPUT);

8.2 IO输入函数

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, PIO~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输出函数

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, PIO~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);

8.5 采用位带操作IO输入函数（注：STM32H7XX和STM32MP15X不支持本函数）

函数原型	IO_ReadBit(IOx)
函数功能	快速读取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;
返回参数	返回值是1, 高电平, 是0, 低电平;
使用说明	以PA0端口为例 INT8U val; val = IO_ReadBit(PA0); if (val == 0) { } ; // PA0输入低电平 if (val == 1) { } ; // PA0输入高电平

8.6 采用位带操作IO输出函数（注：STM32H7XX和STM32MP15X不支持本函数）

函数原型	IO_WriteBit(IOx, val)
函数功能	快速输出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;
返回参数	无
使用说明	以PA0端口为例 IO_WriteBit(PA0, 0); // PA0输出低电平 IO_WriteBit(PA0, 1); // PA0输出高电平

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	<pre>// EXTI_PARA参数数据结构 typedef struct { INT8U id; // 中 断 ID 选 择 : EXTI0_ID~EXTI15_ID, EXTI16_XX_ID~EXTI22_XX_ID INT8U Mode; // 中 断 模 式 : EXTI_RISING, EXTI_FALLING, EXTI_RISING_FALLING INT8U IOx; // 选择IO端口: PA0~PA15, PB0~PB15, PC0~PC15, PD0~PD15, // PE0~PE15, PF0~PF15, PG0~PG15, PH0~PH15, PI0~PI15; INT8U IRQPriority; // 中断优先级, 范围: 0-15 }EXTI_PARA; // 以PE0为例初始化中断 EXTI_PARA Para; Para.id = EXTI0_ID; // 选择中断ID Para.Mode = EXTI_RISING; // 上升沿中断 Para.IOx = PE0; // 选择PE0端口 Para.IRQPriority = EXTI_IRQ_PRIORITY; EXTI_Init((EXTI_PARA *)&Para); // 初始化EXTI0 (PE0) 中断 EXTI_Ctrl(EXTI0_ID, EXTI_CMD_INT_ENA); // 启动EXTI0 (PE0) 中断 具体参考../libapp/exti_app.c中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</pre>

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, EXTI16_XX_ID~EXTI22_XX_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, EXTI_CMD_DIS);

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, PJ0, PK0	void EXTI0_ISRHook(void)
EXTI1_ID	PA1, PB1, PC1, PD1, PE1, PF1, PG1, PH1, PI1, PJ1, PK1	void EXTI1_ISRHook(void)
EXTI2_ID	PA2, PB2, PC2, PD2, PE2, PF2, PG2, PH2, PI2, PJ2, PK2	void EXTI2_ISRHook(void)
EXTI3_ID	PA3, PB3, PC3, PD3, PE3, PF3, PG3, PH3, PI3, PJ3, PK3	void EXTI3_ISRHook(void)
EXTI4_ID	PA4, PB4, PC4, PD4, PE4, PF4, PG4, PH4, PI4, PJ4, PK4	void EXTI4_ISRHook(void)
EXTI5_ID	PA5, PB5, PC5, PD5, PE5, PF5, PG5, PH5, PI5, PJ5, PK5	void EXTI5_ISRHook(void)
EXTI6_ID	PA6, PB6, PC6, PD6, PE6, PF6, PG6, PH6, PI6, PJ6, PK6	void EXTI6_ISRHook(void)
EXTI7_ID	PA7, PB7, PC7, PD7, PE7, PF7, PG7, PH7, PI7, PJ7, PK7	void EXTI7_ISRHook(void)
EXTI8_ID	PA8, PB8, PC8, PD8, PE8, PF8, PG8, PH8, PI8, PJ8, PK8	void EXTI8_ISRHook(void)
EXTI9_ID	PA9, PB9, PC9, PD9, PE9, PF9, PG9, PH9, PI9, PJ9, PK9	void EXTI9_ISRHook(void)
EXTI10_ID	PA10, PB10, PC10, PD10, PE10, PF10, PG10, PH10, PI10, PJ10, PK10	void EXTI10_ISRHook(void)
EXTI11_ID	PA11, PB11, PC11, PD11, PE11, PF11, PG11, PH11, PI11, PJ11, PK11	void EXTI11_ISRHook(void)
EXTI12_ID	PA12, PB12, PC12, PD12, PE12, PF12, PG12, PH12, PI12, PJ12, PK12	void EXTI12_ISRHook(void)
EXTI13_ID	PA13, PB13, PC13, PD13, PE13, PF13, PG13, PH13, PI13, PJ13, PK13	void EXTI13_ISRHook(void)
EXTI14_ID	PA14, PB14, PC14, PD14, PE14, PF14, PG14, PH14, PI14, PJ14, PK14	void EXTI14_ISRHook(void)
EXTI15_ID	PA15, PB15, PC15, PD15, PE15, PF15, PG15, PH15, PI15, PJ15, PK15	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~UARTx_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	<pre>// UART_PARA参数结构 typedef struct { INT32U Flag; // 工作参数标志 INT32U BaudRate; // 波特率: 参见波特率定义 INT8U Mode; // 工作模式: 参考工作模式定义 INT8U RxMode; // 接收数据工作模式: 参考接收数据工作模式 INT32U RxTimeOut; // 接收超时时间: 单位us, 请根据波特率自行计算超 // 时间, 一般建议范围: 1000us~100000us; // 用户一般根据这个来自动将接收到的数据分段处理 INT8U TXPin; // 发送数据IO管脚定义 INT8U RXPIn; // 接收数据IO管脚定义 INT8U RS485DirPin; // RS485方向控制IO管脚定义 INT8U *pTxBuf; // 发送数据缓存指针 INT16U TxBufLen; // 发送缓存长度 INT8U *pRxBuf; // 接收数据缓存指针 INT16U RxBufLen; // 接收缓存长度 INT8U IRQPriority; // 中断优先级, 范围: 0-15 } UART_PARA;</pre> 具体初始化参考在../libapp/uart_app.c中调用Uart_AppInit()实现UART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~UARTx_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 接收多字节数据函数

函数原型	INT32S Uart_Read (INT8U id, INT8U *p, INT16U len)
函数功能	接收多字节数据函数
入口参数	id: UART索引标识 (UART1_ID~UARTx_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~UARTx_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	<pre> INT32S flag; INT8U val; val = 55; flag = Uart_SendChar (UART1_ID, val); if (flag == ERR_TRUE) { // 发送数据成功} </pre>

10.5 发送多字节数据

函数原型	INT32S Uart_Write (INT8U id, INT8U *p, INT16U len)
函数功能	发送多字节数据
入口参数	id: UART索引标识 (UART1_ID~UARTx_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	<pre> 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} </pre>

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 CMD_UART_SYNC: UART同步操作 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	<pre> 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; } } } </pre>

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	<pre>// SPI参数结构 typedef struct { INT16U Flag; // 参数标志 SPI_PIN Pin; // SPI管脚 SPI_CR1_TypeDef cfg; // SPI配置参数 INT8U CRCPolynomial; // CRC多项式 } SPI_PARA;</pre> 具体应用请参考在../libapp/spi_app.c中SPI_Applnit()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~SPIx_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	<pre>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}</pre>

11.3 SPI利用DMA读数据

函数原型	INT32S SPI_DMAREad(INT8U id, INT8U *p, INT32U len, SPI_DMA_PARA *pPara)
函数功能	读取数据函数
入口参数	id: SPI识别: SPI1_ID~SPIx_ID *p: 读出数据存储的地址指针; len: 要读出数据长度; pPara, DMA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	<pre> INT32S flag; INT8U buf[16] , SPI_DMA_PARA Para; Para.Flag=0; Para.Timeout = 1000; IO_Write (PA15,0); // 片选置低, 假设PA15是片选信号 flag = SPI_DMAREad (SPI1_ID, buf, 16, &Para); IO_Write (PA15,1); // 片选置高 if (flag == ERR_TRUE) { // 读取数据成功, 数据在buf中} </pre>

11.4 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~SPIx_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	<pre> 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} </pre>

11.5 SPI利用DMA写数据

函数原型	INT32S SPI_DMAWrite(INT8U id, INT8U *p, INT32U len, SPI_DMA_PARA *pPara)
函数功能	发送数据函数
入口参数	id: SPI识别: SPI1_ID~SPIx_ID *p: 写入数据存储的地址指针; len: 写入数据长度; pPara, DMA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	<pre> INT32S flag; INT8U buf[16], i, SPI_DMA_PARA Para; for (i=0; i<16; i++) { buf[i] = i; } Para.Flag=0; Para.Timeout = 1000; IO_Write (PA15,0); // 片选置低, 假设PA15是片选信号 flag = SPI_DMAWrite (SPI1_ID, buf, 16, &Para); IO_Write (PA15,1); // 片选置高 if (flag == ERR_TRUE) { // 写入数据成功} </pre>

11.6 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~SPIx_ID val: 写入数据;
返回参数	返回: 读出的数据;
使用说明	<pre> 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 </pre>

11.7 SPI读写多字节数据

函数原型	INT8U SPI_ReadWrite(INT8U id, INT8U *pTX, INT8U *pRX, INT32U len)
函数功能	发送数据函数
入口参数	id: SPI识别: SPI1_ID~SPIx_ID pTX, 发送数据指针; pRX, 接收数据指针; len, 发送接收数据长度
返回参数	返回: 读出的数据;
使用说明	<pre> INT32S i, flag; INT8U TxBuf[16], RxBuf[16]; for (i=0; i<16; i++) { TxBuf[i] = i; } IO_Write (PA15, 0); // 片选置低, 假设PA15是片选信号 flag = SPI_ReadWrite(SPI1_ID, TxBuf, RxBuf, 16); IO_Write (PA15, 1); // 片选置高 if (flag == ERR_TRUE) { // 读写数据成功, 数据在RxBuf} </pre>

11.8 SPI利用DMA读写多字节数据

函数原型	INT8U SPI_DMAReadWrite(INT8U id, INT8U *pTX, INT8U *pRX, INT32U len, SPI_DMA_PARA *pPara)
函数功能	发送数据函数
入口参数	id: SPI识别: SPI1_ID~SPIx_ID; pTX, 发送数据指针; pRX, 接收数据指针; len, 发送接收数据长度; pPara, DMA参数指针
返回参数	返回: 读出的数据;
使用说明	<pre> INT32S i, flag; INT8U TxBuf[16], RxBuf[16]; SPI_DMA_PARA Para; Para.Flag=0; Para.TimeOut = 1000; for (i=0; i<16; i++) { TxBuf[i] = i; } IO_Write (PA15, 0); // 片选置低, 假设PA15是片选信号 flag = SPI_ReadWrite(SPI1_ID, TxBuf, RxBuf, 16, &Para); IO_Write (PA15, 1); // 片选置高 if (flag == ERR_TRUE) { // 读写数据成功, 数据在RxBuf} </pre>

11.9 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~SPIx_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 I ² C_Init(INT8U I ² Cx, I ² C_PARA *pPara)
函数功能	I ² C初始化函数
入口参数	id, I ² C索引标识: I ² C1_ID~I ² Cx_ID; *pPara, I ² C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	<pre>// I²C参数结构 typedef struct { INT16U Flag; // 参数标志 INT8U PinRemap; // I²C引脚功能重映射 INT8U SDAPin; // SDA引脚 INT8U SCLPin; // SCL引脚 INT16U Mode; // I²C工作模式: INT16U DutyCycle; // Specifies the I²C fast mode duty cycle. // This parameter can be a value of I²C_duty_cycle_in_fast_mode INT16U OwnAddress1; // Specifies the first device own address. // This parameter can be a 7-bit or 10-bit address. INT16U Ack; // Enables or disables the acknowledgement. INT16U AcknowledgedAddress; // Specifies if 7-bit or 10-bit address //is acknowledged. INT32U ClockSpeed; // I²C时钟频率, 设置值必须小于400KHz. } I²C_PARA; 具体应用请参考在../libapp/I²C_app.c中I²C_AppInit()实现I²C初始化, 一般 客户不需要更改该函数;</pre>

12.2 I²C读数据

函数原型	INT32S I ² C_Read(INT8U id, INT16U I ² CAddr, INT16U addr, INT8U *p, INT16U len)
函数功能	I ² C总线读数据函数
入口参数	id, I ² C索引标识: I ² C1_ID~I ² Cx_ID; I ² CAddr, I ² C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 I ² C_Write(INT8U id, INT16U I ² CAddr, INT16U addr, INT8U *p, INT16U len)
函数功能	I ² C总线写数据函数
入口参数	id, I ² C索引标识: I ² C1_ID~I ² Cx_ID; I ² CAddr, I ² C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 I ² C_Ctrl(INT8U id, INT8U Cmd, INT32U Para)
函数功能	I ² C命令控制
入口参数	id, I ² C索引标识: I ² C1_ID~I ² Cx_ID; Cmd, I ² C控制命令, 如下定义: CMD_I ² C: 启用I ² C外设或禁用I ² C外设, Para是ENABLE或DISABLE CMD_I ² C_RST: 复位I ² C寄存器为初始状态, Para为0 CMD_I ² C_CLOSE: 关闭I ² C时钟, 也就是关闭I ² C功能, 可以省电, Para为0 Para, I ² C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(注: 只支持STM32F10X/4XX/GD32F30X模块)

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~CANx_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	<pre>// CAN初始化参数数据结构: typedef struct { INT16U Flag; // 工作标志 INT8U Mode; // 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); INT32U Baud; // CAN波特率 INT8U IDE; // 帧类型: 标准帧:CAN_STD_ID, 扩展帧:CAN_EXT_ID INT8U RTR; // 远程发送请求: 数据帧, CAN_RTR_DATA; 远程帧, // CAN_RTR_REMOTE; INT8U PinRemap; // CAN引脚功能重映射 INT8U TXPin; // CAN发送引脚定义 INT8U RXPin; // CAN发送引脚定义 INT8U SJW; // 重新同步跳跃宽度, 范围: CAN_SJW_1tq~CAN_SJW_3tq INT8U BS1; // 时间段1, 范围: CAN_BS1_1tq~CAN_BS1_16tq INT8U BS2; // 时间段2, 范围: CAN_BS2_1tq~CAN_BS2_1tq INT32U MCR; // MCR寄存器 INT16U RxMsgNum; // CAN接收缓存可接收消息个数, 范围 1~256, 数量不要 // 太大, 会占用大量RAM INT16U TxMsgNum; // CAN发送缓存可发送消息个数, 范围 1~256, 数量不要 // 太大, 会占用大量RAM CAN_BUF_MSG *pRxMsgBuf; // CAN接收缓存指针 CAN_BUF_MSG *pTxMsgBuf; // CAN发送缓存指针 INT8U IRQPriority; // 中断优先级, 范围: 0-15 }CAN_PARA; // CAN滤波器数据帧结构定义 typedef struct {</pre>

```
INT8U  CAN2StartBank; // CAN2开始组, 它们定义了CAN2(从)接口的开始组, //范围是1~27

INT32U  FIFO; // CAN 过滤器FIFO关联配置:Bit27~Bit0有效, bit0是第0组,
//bit27是第27组 报文在通过了某过滤器的过滤后, 将被存放到其关联的FIFO中,
// 0: 过滤器被关联到FIFO0; 1: 过滤器被关联到FIFO1。

INT32U  Scale; // CAN 过滤器位宽寄存器:Bit27~Bit0有效, bit0是第0组, // bit27
是第27组, 0: 过滤器位宽为2个16位; 1: 过滤器位宽为单个32位。


INT32U  Mode; // CAN过滤器模式 (Filter mode): Bit27~Bit0有效,
//bit0是第0组, bit27是第27组, 0: 过滤器组x的2个32位寄存器工作在标识符屏蔽//位
模式; 1: 过滤器组x的2个32位寄存器工作在标识符列表模式。

INT32U  Active; // 过滤器激活 (Filter active): Bit27~Bit0有效,
//bit0是第0组, bit27是第27组; 0: 过滤器被禁用; 1: 过滤器被激活。

INT32U  *pBuf; // 设置的过滤器寄存器数据指针
INT8U  len; // 设置的过滤器寄存器数据长度
INT8U  MaxLen; // 滤波器总长度
}CAN_FILTER_PARA;
```

具体应用请参考在../libapp/can_app.c中CAN_Applnit()实现CAN初始化, 一般客户不需要更改该函数;

13.2 CAN初始化函数(注: 只支持STM32H7XX/STM32MP15X模块)

函数原型	INT32S CAN_Init(INT8U id, CAN_PARA *pPara)
函数功能	CAN始化函数
入口参数	id, CAN识别号(CAN1_ID~CANx_ID); *pPara, CAN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;
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	// CAN初始化参数数据结构: <pre> typedef struct { INT16U Flag; // 工作标志 INT8U Mode; // 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); INT32U Baud; // CAN波特率 INT8U IDE; // 帧类型: 标准帧:CAN_STD_ID, 扩展帧:CAN_EXT_ID INT8U RTR; // 远程发送请求: 数据帧, CAN_RTR_DATA; 远程帧, // CAN_RTR_REMOTE; INT8U TXPin; // CAN发送引脚定义 INT8U RXPin; // CAN发送引脚定义 INT8U FrameFormat; //CAN帧模式: CAN_FRAME_CLASSIC/ // CAN_FRAME_FD_NO_BRS/CAN_FRAME_FD_BRS INT8U TxDataSize; // 发送数据域大小: CAN_DATA_BYTES_8~CAN_DATA_BYTES_64 INT8U RxDataSize; // 接收数据域大小: CAN_DATA_BYTES_8~CAN_DATA_BYTES_64 INT32U DataBaud; // CAN数据波特率 INT8U IRQPriority; // 中断优先级, 范围: 0-15 }CAN_PARA; </pre> 具体应用请参考在../libapp/can_app.c中CAN_AppInit()实现CAN初始化, 一般客户不需要更改该函数;

13.3 CAN滤波器初始化函数(注: 只支持STM32F10X/4XX/GD32F30X模块)

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	<pre>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"); }</pre>

13.4 CAN滤波器初始化函数(注：只支持STM32H7XX/STM32MP15X模块)

函数原型	INT32S CAN_FilterInit(INT8U id, CAN_FILTER_PARA *pFilter)
函数功能	CAN过滤器初始化函数
入口参数	id, CAN识别号(CAN1_ID~CANx_ID); *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	<pre> // CAN滤波器数据帧结构定义 typedef struct { INT8U IdType; // ID 帧类型: 0, 标准帧:CAN_STD_ID; 1, 扩展 帧:CAN_EXT_ID; INT8U FilterIndex; // 滤波器索引: 0-15 INT8U FilterType; // 滤 波 器 类 型 : 0,CAN_FILTER_RANGE; //1,CAN_FILTER_DUAL; 2,CAN_FILTER_MASK; 3,CAN_FILTER_RANGE_NO_EIDM INT8U FilterConfig; //滤波器配置: 目前只支持CAN_FILTER_TO_RXFIF0 INT8U RxBufferIndex; // 接收缓冲区索引, 只有FilterConfig选择 // CAN_FILTER_TO_RXBUFFER才有效;范围: 0-31; INT8U IsCalibrationMsg; // 指定是否为配置滤波器校准信息。如果 // FilterConfig不同, 则忽略此参数来自FDCAN_FILTER_TO_RXBUFFER。 // 此参数可以是: 0, 普通消息; 1: 校准信息; 默认是0 INT32U FilterID1; // 滤波器筛选ID1: 标准帧范围:0-0x07FF; // 扩展帧范围: 0-0x1FFFFFFF; INT32U FilterID2; // 滤波器筛选ID2: 标准帧范围:0-0x07FF; // 扩展帧范围: 0-0x1FFFFFFF; } CAN_FILTER_PARA; CAN_FILTER_PARA Filter; INT32S flag; // 滤波器初始化参数 Filter.FilterConfig = CAN1_FILTER_CONFIG; Filter.RxBufferIndex = CAN1_RXBUF_ID; Filter.IsCalibrationMsg = CAN1_ISCALIB_MSG_ID; #if (CAN1_FOACT_EN > 0) // 过滤器组0设置 </pre>

```
Filter.IdType = CAN1_F0IDE;

Filter.FilterType = CAN1_F0MODE;

Filter.FilterID1 = CAN1_F0ID1;

Filter.FilterID2 = CAN1_F0ID2;

Filter.FilterIndex = 0;

flag = CAN_FilterInit(CAN1_ID, &Filter);

if (flag != ERR_TRUE)

{ printf("AT+ERROR=CAN1 Filter0 Init, ERROR\r\n");}

else { printf("CAN初始化: OK\r\n"); }
```

具体应用请参考在../libapp/can_app.c中CAN_AppInit()实现CAN初始化，一般客户不需要更改该函数；

13.5 CAN接收数据函数

函数原型	INT32S CAN_Read(INT8U id, CAN_RX_MSG *pRxMsg, INT16U Num)
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; Num, 接收CAN消息数量;
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	<pre>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, 1); if (flag == ERR_TRUE) { // 接收数据成功 } }</pre>

13.6 CAN发送函数

函数原型	INT32S CAN_Write(INT8U id, CAN_TX_MSG *pTxMsg, INT16U Num)
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; Num, 发送CAN消息数量;
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	<pre>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; } flag = CAN_Write(CAN1_ID, (CAN_TX_MSG *)&CanTx, 1); if (flag == ERR_TRUE) { // 发送数据成功 }</pre>

13.7 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

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	<pre>// ADC参数结构 typedef struct { INT32U Flag; // 参数标志 INT8U Mode; // 工作模式: 参考工作模式定义 INT8U ExTSel; // 规则转换外部触发选择 INT8U AINum; // AIX采样通道板子端口数量 INT8U *pAITab; // AI采样通道顺序转换表指针 INT16U AvgNum; // 定义采样次数来计算平均值, 范围 1~256 INT16U Freq; // AD采样频率, 每秒钟采样次数 INT16U SampleTime; // ADC采样转换时间 INT16U *pBuf; // AD采集数据缓存指针 #if ((MODULE_CLASS == STM32MP15X) (MODULE_CLASS == STM32H7XXX)) INT32S *pBuf; // AD采集数据缓存指针 #else INT16U *pBuf; // AD采集数据缓存指针 #endif INT8U IRQPriority; // 中断优先级, 范围: 0-15 #if ((MODULE_CLASS == STM32MP15X) (MODULE_CLASS == STM32H7XXX)) INT8U ExTEnMode; // 外部触发使能模式: ADC_EXTEN_DIS/ADC_EXTEN_RISING //ADC_EXTEN_FALLING/ADC_EXTEN_RISING_FALLING #endif }</pre>

	<pre> INT8U Clock; // 时钟 INT16U OVSR; // 过采样倍数 INT8U OVSS; // 过采样数据右移位数 #endif }ADC_PARA; 具体应用请参考在../libapp/adc_app.c中ADC_AppInit()实现ADC初始化，一般 客户不需要更改该函数； </pre>
--	---

14.2 读取 AD 数据

函数原型	<pre> INT32S ADC_Read(INT8U id, INT16S *p, INT8U len) </pre> INT32S ADC_Read(INT8U id, INT32S *p, INT8U len) // 只有STM32H7XX和STM32MP15X支持该函数
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	<pre> INT32S flag; INT16S buf[8]; // INT32S buf[8]; // STM32H7XX和STM32MP15X用这个定义 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]:A11; buf[1]:A12; buf[2]:A13; //buf[3]:A14; buf[4]:A15; buf[5]:A16; buf[6]:A17; buf[7]:A18; } } </pre>
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

14.3 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	<pre>// ADC初始化参数结构 typedef struct { INT16U Flag; // 工作标志 INT8U Mode; // 工作模式 INT8U pin; // DAC管脚 INT16U *pbuf; // 缓存指针 INT16U len; // 缓存长度 INT32U Freq; // 工作频率 INT8U IRQPriority; // 中断优先级, 范围: 0-15 }DAC_PARA;</pre> 具体应用请参考在../libapp/dac_app.c中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	<pre>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</pre>

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: 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： <pre>#define BKP_EN 1 // BKP使能, 1: 打开使能, 0: 关闭</pre> 请参考在API_Init() 函数中调用 <pre>#if (BKP_EN > 0) BKP_Init(); #endif</pre>
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/GD32F30X 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	<pre>INT32S flag; INT16U buf[4] = {100, 1000, 10000, 20000}; flag = BKP_Write(1, buf, 4); if(flag == ERR_TRUE) { //写入成功 }</pre>

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/GD32F30X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页或扇区擦除 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/GD32F3XX系列模块: 按页 (2KB) 擦除; STM32F4xx系列模块: 按扇区 (FLASH_SECTOR_0~FLASH_SECTOR_11) 擦除; STM32H7xx系列模块: 按扇区 (FLASH_SECTOR_0~FLASH_SECTOR_15) 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	<pre>// IAP参数数据结构 typedef struct { INT16U Flag; // 标志位 INT32U HSEClk; // 外部时钟晶振频率 INT16U Moduel; // 模块类型 INT16U UartID; // 串口ID, Debug信息输出串口选择 INT32U Baud; // 串口波特率 INT16U ABPageSize; // A/B/AB区页大小 INT16U CPageSize; // C区页大小 AREA_DEF Area[4]; // 区参数 INT16U BootPin; // 启动检测管脚 INT8U FileName[12]; // 启动文件名称, 采用8.3文件定义 INT8U RS485_DIR; // RS485的方向管脚 INT8U TX_PIN; // UART TX管脚 INT8U RX_PIN; // UART RX管脚 } IAP_PARA</pre> 具体应用请参考在../libapp/libapp_app.c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) CMD_IAP_ERASE_PAGE, FLASH 页擦除, 只支持STM32F1xx/GD32F3XX系列模块 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(PWM1_ID~PWMx_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	<pre>// PWM参数结构 typedef struct { INT32U Flag; // 工作参数标志 INT8U Mode; // PWM工作模式 INT8U TimerID; // 选择定时器 TIM_PIN TimPin; // 所选定时器管脚定义 INT32U Freq; // PWM输出初始频率 INT16U Rate[4]; // 4个通道, PWM输出占空比 INT8U IRQPriority; // 中断优先级, 范围: 0-15 } PWM_PARA;</pre> 具体应用请参考在../libapp/pwm_app.c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 DMA方式输出函数

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((PWM1_ID~PWMx_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_ENAMUL: 同时使能多通道 CMD_PWM_DISMUL: 同时停止多通道 CMD_PWM_PULNUM: 读取当前发出脉冲个数 CMD_PWM_PAUSE: 暂时停止PWM输出 pPara, 控制参数指针; // PWM控制参数结构 <pre>typedef struct { INT8U Mode; // 控制操作模式 INT8U Chx; // 选择通道 INT32U Freq; // PWM输出频率 INT16U Rate[4]; // 4个通道, PWM输出占空比 INT32U Num[4]; // 输出脉冲数量 } PWM_CTRL;</pre>
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	具体应用请参考在../libapp/pwm_app.c中相关函数;

19.3 PWM利用DMA控制函数

函数原型	INT32S PWM_Write(INT8U id, PWM_WRITE_PARA *pPara)
函数功能	PWM利用DMA输出控制操作
入口参数	id: PWM索引: id, PWM索引(PWM1_ID~PWMx_ID) pPara: 参数指针, 参数数据结构如下 <pre>typedef struct { INT32U Flag; // 控制标志位 INT8U ch; // 选择PWM输出通道: PWM_CH1~PWM_CH4 INT8U chx; // 选 择 PWM 输 出 通 道 标 志 : // PWM_CH1FLAG~PWM_CH4FLAG INT16U Rate; // PWM占空比, 暂时未用 INT32U Freq; // 脉冲频率 INT32U len; // 数据缓存数量 void *pData; // 数据缓存指针 } PWM_WRITE_PARA;</pre>
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	具体应用请参考在../libapp/pwm_app.c中相关函数;

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(FCLK1_ID~FCLKx_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	<pre>// FCLK参数结构 typedef struct { INT32U Flag; // 工作参数标志 INT8U Mode; // FCLK工作模式 INT8U TimerID; // 选择定时器 TIM_PIN TimPin; // 所选定时器管脚定义 INT32U MinFreq; // PWM输出初始频率 INT16U *pBuf[4]; // 接收数据缓存指针 INT16U BufLen[4]; // 接收缓存长度 INT8U IRQPriority; // 中断优先级, 范围: 0-15 } FCLK_PARA;</pre> 具体应用请参考在../libapp/fclk_app.c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 (FCLK1_ID~FCLKx_ID); Cmd, 控制命令: CMD_FCLK_ENA: 使能单一通道; CMD_FCLK_DIS: 停止单一通道; CMD_FCLK_STATUS: 读取 FCLK 状态; CMD_FCLK_CLEAR: 定时器值清零 CMD_FCLK_CNT: 读取测量脉冲频率或者占空比数量 CMD_FCLK_CLK: 读取 FCLK 所使用的时钟值 Chx, 通道号是 FCLK_CH1FLAG~FCLK_CH4FLAG, 可以多通道操作; 或者通道号是:FCLK_CH1~FCLK_CH4, 只能单通道操作;
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 (FCLK1_ID~FCLKx_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, 通道号是: FCLK_CH1~FCLK_CH4, 只能单通道操作; *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 <pre> 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 } </pre>

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: TIM1_ID~TIMx_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	<pre>// TIMER参数结构 typedef struct { INT32U Flag; // 工作参数参数标志 INT8U Mode; // 工作模式 INT16U Prescaler; // 分频系数, 当工作模式是TIM_WKMODE_COUNT // 时, 必须设// 置分频系数 INT16U TrgoSel; // TRGO输出选择 INT8U IRQPriority; // 中断优先级, 范围: 0-15 }TIM_PARA;</pre> 具体应用请参考在../libapp/tim_app.c中调用TIM_AppInit()实现TIM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_ENA: 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 <pre> TIM_CTRL TIMCtrl; TIMCtrl.Chx = TIM2CH_EN; TIMCtrl.t = TIM2_T; // 设置主定时器定时时间 Timer_Ctrl(TIM2_ID, CMD_TIM_ENA, (TIM_CTRL *)&TIMCtrl.Chx); </pre> 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: <pre> // TIM2读取计数值 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 </pre>

22. 外部总线操作(fsmc.h, 只在STM32F103ZE/GD32F303ZE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: <pre>#define EXBUS_EN 1 // 外部总线使能: 1, 使能; 0, 关闭;</pre> <pre>#define EXBUS_ADDSET 3 // 外部总线地址建立时间(范围: 0~15):</pre> 实际 // 建立时间 (EXBUS_ADDSET+1) 个 HCLK; <pre>#define EXBUS_DATAST 5 // 外部总线数据保持时间(范围: 1~255):</pre> 实际// 保持时间: 读 (EXBUS_DATAST+3) 个 HCLK, 写// (DATAST+1) 个HCLK; <pre>#define EXBUS_TURN 3 // 外部总线恢复时间(范围: 0~15): 实际</pre> 恢复// 时间 (EXBUS_TURN+1) 个HCLK: 1, 使能; 0, 关闭; 具体应用请参考在../libapp/libapp.c中调用FSMC_ApplInit()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(USB_D PARA *pPara)
函数功能	USB设备模式初始化函数
入口参数	USB_D 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: <pre>// 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设备使能: 1, 使能; 0, 关闭; // USB设备, 实现将板子的SD卡、SPI FLASH或者NAND FLASH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</pre> 具体应用请参考在../libapp/usb_app.c中调用USB_D ApplInit() 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	<pre>void USB_Test(void) { INT32S flag; INT8U buf[64], i; INT16U len, rLen; rLen=0; while(1) { 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; } } }</pre>

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,
在去读取相应
长度数据

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: <pre>// 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;</pre> 具体应用请参考在../libapp/usb_app.c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: <pre>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</pre> 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	<pre>#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</pre> 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 <pre>#define EEPROM_EN 1 // EEPROM使能, 1: 打开使能, 0: 关闭</pre> <pre>#define EEPROM_DEVICE AT24C64 // 定义器件型号</pre> <pre>#define EEPROM_FREQ 100000 // 读写时钟频率</pre> 具体参考在../libapp/eeprom_app.c中调用EEPROM_ApplInit()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, 所以请不要操作这个地址范围, 否则出错;
使用说明	<pre>INT32S flag;</pre> <pre>INT8U buf[16];</pre> <pre>flag = EEPROM_Read(0, buf, 16); // 读取地址0开始的16个字节</pre> <pre>if (flag == ERR_TRUE)</pre> <pre>{ // 读取数据成功, 数据在buf中}</pre>

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, 所以请不要操作这个地址范围, 否则出错;
使用说明	<pre>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}</pre>

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	//SPI Flash(AT45DBXX) 初始化参数定义 <pre>typedef struct { INT16U Flag; // 工作标志 INT8U SPI_ID; // 选择SPI ID INT8U CS_Pin; // 片选管脚 INT16U SectorSize; // Flash扇区大小(单位: 字节) INT32U SectorNum; // Flash扇区数量 INT32U FatFSSectorNum; // FatFS文件系统占用扇区数量 } AT45DBXX_PARA;</pre> 具体参考在../libapp/spiflash_app.c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	<pre>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}</pre>

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 : 0x000000~0x001ffffff/0x003ffffff (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	<pre> 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 } </pre>

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	<pre>//SPI Flash(W25QXX) 初始化参数定义 typedef struct { INT16U Flag; // 工作标志 INT8U SPI_ID; // 选择SPI ID INT8U CS_Pin; // 片选管脚 INT16U SectorSize; // Flash扇区大小(单位: 字节) INT32U SectorNum; // Flash扇区数量 INT32U FatFSSectorNum; // FatFS文件系统占用扇区数量 INT8U *pBuf; // FatFS接口函数缓存 }W25QXX_PARA; 具体参考在../libapp/spiflash_app.c中调用SPIFlash_APPInit() 实现 W25QXX初始化, 一般客户不需要更改该函数</pre>

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	<pre>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}</pre>

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	<pre> 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} </pre>

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	<pre> 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} </pre>

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, 0x000000~0x00fffff;
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	<pre> 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} </pre>

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	<pre> 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} </pre>

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	NFLASH初始化数据结构 <pre>typedef struct { INT16U Flag; // 工作标志 INT16U BlkNum; // Nand Flash 总块数 INT16U PagePerBlk; // Nand Flash 每个块包含页数 INT16U PageSize; // 页大小 INT16U PageNum; // 总页数 INT16U BadBlkNum; // 坏块总数 INT8U *pBadBlkFlag; // 坏块标记指针 INT16U *pBadTab; // 坏块替换表指针 INT8U *pBlkBuf; // 块缓存指针 INT8U *pBlkSpareBuf; // 块空闲区域缓存指针 }NFLASH_PARA;</pre> 具体参考在../libapp/libapp.c中调用NFlash_Applnit()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	具体参考在../libapp/libapp. c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	具体参考在../libapp/net_app.c中调用NET_APPInit()实现网络初始化, 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) INT32S NET_Write (NET_TxPacket *p) 只有STM32H7XX模块支持
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 读取网络数据

函数原型	INT32S NET_Read (INT32U *p, INT16U *len) INT32S NET_Read (NET_RxPacket *pRxPacket) 只有STM32H7XX模块支持
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。

32. DMA初始化函数

函数原型	INT32S DMA_Init (INT8U id, DMA_INIT *pPara)
函数功能	DMA初始化
入口参数	<pre>// DMA初始化数据结构 typedef struct { INT32U Flag; // DMA工作标志 INT16U ChPara[8]; // 通道参数 } DMA_INIT;</pre>
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	具体参考在../libapp/dma_app.c中调用DMA_AppInit()实现网络初始化, 一般客户不需要更改该函数

33. MODBUS主机通信函数

33.1 Modbus主机初始化函数

函数原型	INT32S Modbus_Init(MODBUS_INIT_PARA *pPara)
函数功能	Modbus主机初始化
入口参数	<pre>// Modbus主机模式初始化参数结构 typedef struct { INT8U Flag; // 工作标志 INT8U t; // 等待延时, 单位ms, 一般设置小于20的值 INT8U len; // 工作缓存长度, 小于256字节 INT8U *pbuf; // 工作缓存指针 INT8U max_chnum; // 最大通道数量 MODBUS_CHX_DATA *pChData; // 通道数据指针 void (*Callback)(MODBUS_CHX_DATA *pData); // 返回数据回调函数 }MODBUS_INIT_PARA;</pre>
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	具体参考在../libapp/modbus_app.c中调用Modbus_AppInit()实现初始化, 一般客户不需要更改该函数

33.2 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~UARTx_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	<pre> #define DATA_LEN 16 INT8U buf[(DATA_LEN+7)/8]; INT32S flag; // 设备ID为10，读取线圈地址为0开始的16个线圈值, 超时时间是100ms flag=Modbus_ReadCoils(UART2_ID, 10, 0, DATA_LEN, buf, 100); if(flag == ERR_TRUE) { // 16个线圈状态在buf中} </pre>

33.3 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~UARTx_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	<pre> #define DATA_LEN 16 INT8U buf[(DATA_LEN+7)/8]; INT32S flag; // 设备ID为10，读取离散量地址为0开始的16个值, 超时时间是100ms flag=Modbus_ReadDisInput (UART2_ID, 10, 0, DATA_LEN, buf, 100); if(flag == ERR_TRUE) { // 16个离散输入量在buf中} </pre>

33.4 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~UARTx_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	<pre> #define DATA_LEN 16 INT16U buf[DATA_LEN]; INT32S flag; // 设备ID为10, 读取地址为0开始的16个保持寄存器值, 超时时间是100ms flag = Modbus_ReadHoldReg(UART2_ID, 10, 0, DATA_LEN, buf, 100); if(flag == ERR_TRUE) { // 16个保持寄存器值在buf中} </pre>

33.5 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~UARTx_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	<pre> #define DATA_LEN 16 INT16U buf[DATA_LEN]; INT32S flag; // 设备ID为10, 读取地址为0开始的16个输入寄存器值, 超时时间是100ms flag = Modbus_ReadInputReg (UART2_ID, 10, 0, DATA_LEN, buf, 100); if(flag == ERR_TRUE) { // 16个输入寄存器值在buf中} </pre>

33.6 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~UARTx_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	<pre>INT32S flag; // 设备ID为10，地址8的线圈设置为1, 超时时间是100ms flag = Modbus_WriteSingleCoil(UART2_ID, 10, 8, 1, 100); if(flag == ERR_TRUE) { // 操作成功 }</pre>

33.7 主机写单个保持寄存器函数

函数原型	INT32S Modbus_WriteSingleHoldReg(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~UARTx_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	<pre>INT32S flag; // 设备ID为10，将地址8写入数据1000, 超时时间是100ms flag=Modbus_WriteSingleHoldReg (UART2_ID, 10, 8, 1000, 100); if(flag == ERR_TRUE) { // 操作成功 }</pre>

33.8 主机写多个线圈函数

函数原型	INT32S Modbus_WriteCoils(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~UARTx_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	<pre> #define DATA_LEN 16 INT8U buf[(DATA_LEN+7)/8]; INT32S flag, i; for(i=0; i<((DATA_LEN+7)/8); i++) { buf[i] = 0x55;} // 设备ID为10，将从地址0开始的16个线圈写入数据, 超时时间是100ms flag = Modbus_WriteCoils(UART2_ID, 10, 0, DATA_LEN, buf, 100); if(flag == ERR_TRUE) { // 操作成功} </pre>

33.9 主机写多个保持寄存器函数

函数原型	INT32S Modbus_WriteHoldReg(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~UARTx_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	<pre> #define DATA_LEN 16 INT16U i, buf[DATA_LEN]; INT32S flag; for (i=0; i<DATA_LEN; i++) { buf[i] = 0x1234;} //设备ID为10，向地址0开始的16个保持寄存器写入值, 超时时间是100ms flag = Modbus_WriteHoldReg (UART2_ID, 10, 0, DATA_LEN, buf, 100); if(flag == ERR_TRUE) { // 操作成功);} </pre>

33.10 主机读写多个寄存器函数

函数原型	INT32S Modbus_ReadWriteHoldReg(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~UARTx_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	<pre> #define DATA_LEN 16 INT16U i, buf[DATA_LEN]; INT32S flag; for (i=0; i<DATA_LEN; i++) { buf[i] = i;} //设备ID为10, 向地址0的16个保持寄存器写入并读取;, 超时时间是100ms flag = Modbus_ReadWriteHoldReg (UART2_ID, 10, 0, DATA_LEN, 0, DATA_LEN, buf, 100); if(flag == ERR_TRUE) { // 操作成功} </pre>

33.11 主机接口控制函数

函数原型	INT32S Modbus_Ctrl(INT8U ch, INT8U Cmd, INT32U Para)
函数功能	主机接口控制函数
入口参数	ch, UART索引标识(UART1_ID~UART6_ID); Cmd, 控制命令; 目前只支持CMD_MODBUS_SYNC Para, 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	
使用说明	Modbus_Ctrl(UART2_ID, CMD_MODBUS_SYNC, 10); // Modbus同步处理, 10是调用间隔10ms

33.12 主机接口接收数据处理函数

函数原型	void Modbus_RxProc(INT8U ch, INT8U *p, INT16U len)
函数功能	主机接口接收数据处理函数
入口参数	ch, 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	无
注意	本函数只可以在利用回调函数取得返回数据时应用, UART口接收到数据调用这个处理, 处理结果会通过回调函数输出
使用说明	Modbus_RxPro(UART2_ID, p, 20); // UART2接收到20个数据调用这个函数做处理

34. MODBUS协议处理函数

34.1 Modbus从机协议处理函数

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~UART8_ID(0-7), 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	<pre>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;}</pre>
注意	在调用处理函数之前需要初始化变量, 具体参考在../libapp/modbus_app.c中调用ModbusSlave_Applnit()实现初始化, 一般客户不需要更改该函数

34. LPTimer定时器操作(lptimer.h)

34.1 LPTimer定时器初始化函数

函数原型	INT32S LPTimer_Init(INT8U id, TIM_PARA *pPara)
函数功能	LPTimer初始化函数
入口参数	id, LPTimer索引: LPTIM1_ID ~ LPTIM5_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	<pre>// LPTimer参数结构 typedef struct { INT32U Flag; // 工作参数参数标志 INT8U Mode; // 工作模式 LPTIM_PIN Pin; INT8U ClkSel; // 时钟选择 INT16U Prescaler; // 分频系数, 当工作模式是TIM_WKMODE_COUNT 时, 必须设置分频系数 INT32U Clk; // 时钟频率 INT16U TrgoSel; // TRGO输出选择, Modify: 2021.5.18 INT8U IRQPriority; // 中断优先级, 范围: 0-15 }LPTIM_PARA;</pre> 具体应用请参考在../libapp/lptim_app.c中调用LPTIM_AppInit()实现LPTimer初始化, 一般客户不需要更改该函数;

34.2 LPTimer定时器控制函数

函数原型	INT32S LPTimer_Ctrl(INT8U id, INT8U Cmd, TIM_CTRL *pPara)
函数功能	LPTimer定时器控制函数
入口参数	id, LPTimer索引: LPTIM1_ID ~ LPTIM5_ID Cmd: 控制命令: CMD_LPTIM_ENA: 使能定时器, 当工作模式是TIM_WKMODE_INT(中断模式), 参数pPara必须设置使能的通道及定时时间; 当工作模式是TIM_WKMODE_COUNT(计数模式), 参数pPara=0; CMD_LPTIM_DIS: 关闭定时器, 当工作模式是TIM_WKMODE_INT(中断模式), 参数pPara可以设置为0, 关闭所有; 或者pPara可以设置要使能的通道, 关闭相应通道; 当工作模式是TIM_WKMODE_COUNT(计数模式), 参数pPara=0; CMD_LPTIM_STATUS: 读取定时器状态, 返回值就是状态, 参数pPara=0; CMD_LPTIM_READ: 读取定时计数值, 返回值就是计数值, 参数pPara=0; CMD_LPTIM_RST: 复位TIM寄存器为初始状态, 参数pPara=0; CMD_LPTIM_CLOSE: 关闭TIM时钟, 也就是关闭TIM功能, 可以省电, 参数pPara=0; CMD_LPTIM_SETCNT: 设置TIM计数值, pPara->val是计数值参数pPara, 参数数据结构指针;
返回参数	控制命令是CMD_LPTIM_STATUS时返回LPTIMER状态; 控制命令是CMD_LPTIM_READ时返回计数值; 其它命令返回ERR_TRUE, 控制正确, ERR_TRUE, 控制失败;
使用说明	(1) 以LPTIM2为例, 在定时器工作在定时中断模式中, 启动定时器 <pre> LPTIM_CTRL LPTIMCtrl; LPTIMCtrl.Flag = 0; LPTIMCtrl.t = LPTIM2_T; // 设置定时时间 LPTimer_Ctrl(LPTIM2_ID, CMD_LPTIM_ENA, (LPTIM_CTRL*)&LPTIMCtrl.Flag); </pre> 定时中断函数在ISRHook.c中: void LPTIM2_ISRHook(INT32U flag)用户可以在这个函数下写自己的应用程序 (2) 以LPTIM2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: <pre> // LPTIM2读取计数值 LPTimer_Ctrl(LPTIM2_ID, CMD_LPTIM_ENA, 0); // 使能定时器 </pre>

```
Delay_ms(20); // 延时20ms
LPTimer_Ctrl(LPTIM2_ID, CMD_LPTIM_DIS, 0); // 关闭定时器
cnt = LPTimer_Ctrl(LPTIM2_ID, CMD_LPTIM_READ, 0); // 读取20ms
计数值
#if (DEBUG_APP_EN == 1)
printf("LPTIM2计数值: %d\r\n", cnt); // 打印输出计数值
#endif
```

35. SDRAM操作(s dram. h)

35.1 SDRAM初始化函数

函数原型	INT32S SDRAM_Init(SDRAM_PARA *pPara)
函数功能	SDRAM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	<pre> // SDRAM初始化参数定义 typedef struct { INT32U Flag; // 操作标志 INT8U RCDDelay; // TRCD, 在发送列读写命令时必须要与行有效命令 有一个间隔, 这个间隔被定义为 tRCD, 即 RAS to CAS Delay (RAS 至 CAS 延迟) INT8U RPDelay; //TRP, Precharge command Period, 预充电有效周期 INT8U WriteRecoveryTime; // TWR INT8U RowCycleDelay; // TRC INT8U SelfRefreshTime; // TRAS INT8U ExitSelfRefreshDelay; // TXSR INT8U LoadToActiveDelay; // TMRD INT8U ColumnBitsNumber; // 数据列位数: 8, 9, 10, 11; 本模块是9 INT8U RowBitsNumber; // 数据行位数: 11, 12, 13; 本模块是13 INT8U MemoryDataWidth; // 数据带宽: 8, 16, 32, 本模块是16 INT8U InternalBankNumber; // BANK数量: 2或4, 本模块是4 INT8U CASLatency; // 可以设置1, 2, 3; 设置3比较稳定 INT8U SDClockPeriod; // 时钟分频: 2或3 INT8U ReadPipeDelay; // CAS延时后延后多少个SDRAM时钟周期读取数据: 0, 1, 2 一般设置0 INT8U RefreshCycles; //刷新周期: 要查看芯片手册, 本模块是64ms INT16U RefreshCount; // 刷新行数: 要查看芯片手册, 本模块是8192 } SDRAM_PARA; 具体应用请参考在../libapp/sdram_app.c中调用SDRAM_AppInit()实现SDRAM初始化, 一般客户不需要更改该函数; </pre>

35.2 SDRAM写函数

函数原型	按 8 位写: void SDRAM_Write8b(INT32U addr, INT8U *p, INT32U len) 按 16 位写: void SDRAM_Write16b(INT32U addr, INT16U *p, INT32U len) 按 32 位写: void SDRAM_Write32b(INT32U addr, INT32U *p, INT32U len)
函数功能	SDRAM写函数
入口参数	addr, 写数据起始地址; *p, 数据指针; len, 写入的数据长度
返回参数	无
使用说明	(1) 从地址0x1000开始写入4个8位数据 <pre>static INT8U buf[4] = {0x01, 0x02, 0x03, 0x04}; SDRAM_Write8b(0x1000, buf, 4);</pre> (2) 从地址0x1000开始写入4个16位数据 <pre>static INT16U buf[4] = {0x0101, 0x0202, 0x0303, 0x0404}; SDRAM_Write16b(0x1000, buf, 4);</pre> (3) 从地址0x1000开始写入4个32位数据 <pre>static INT32U buf[4] = {0x01010101, 0x02020202, 0x03030303, 0x04040404}; SDRAM_Write32b(0x1000, buf, 4);</pre>

35.3 SDRAM读函数

函数原型	按 8 位读: void SDRAM_Read8b(INT32U addr, INT8U *p, INT32U len) 按 16 位读: void SDRAM_Read16b(INT32U addr, INT16U *p, INT32U len) 按 32 位读: void SDRAM_Read32b(INT32U addr, INT32U *p, INT32U len)
函数功能	SDRAM读函数
入口参数	addr, 读取数据起始地址; *p, 数据指针; len, 读取的数据长度;
返回参数	无
使用说明	(2) 从地址0x1000开始读4个8位数据 <pre>INT8U buf[4]; SDRAM_Read8b(0x1000, buf, 4);</pre> (4) 从地址0x1000开始读4个16位数据 <pre>INT16U buf[4]; SDRAM_Read16b(0x1000, buf, 4);</pre> (5) 从地址0x1000开始读4个32位数据 <pre>INT32U buf[4], 0x04040404}; SDRAM_Read32b(0x1000, buf, 4);</pre>

36. QSPI读写操作(qspi.h)

36.1 QSPI初始化函数

函数原型	INT32S QSPI_Init(QSPI_PARA *pPara)
函数功能	QSPI初始化函数
入口参数	*pPara, QSPI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	<pre>// QSPI参数结构 typedef struct { INT16U Flag; // 参数标志 QSPI_PIN Pin; // QSPI管脚 INT8U Device; // 器件定义 INT8U CLKDiv; // 时钟分频, 范围: 1-255 INT8U IOMode; // IO线数定义 INT8U CKMode; // 模式0: QSPI_CKMODE0; 模式3: QSPI_CKMODE3 INT32U FlashSize; // Flash大小定义 INT8U IRQPriority; // 中断优先级, 范围: 0-15 } QSPI_PARA; 具体应用请参考在../libapp/qspi_app.c中QSPI_AppInit()实现QSPI初始化, 一般客户不需要更改该函数;</pre>

36.2 QSPI读数据

函数原型	INT32S QSPI_Read(INT32U addr, INT8U *p, INT32U len)
函数功能	读取数据函数
入口参数	addr, 读取数据的起始地址; *p, 读取数据的存储的地址指针; 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;
使用说明	<pre>#define QSPI_ADDR 0x00010000 INT32S flag; INT8U buf[16]; flag = QSPI_Read (QSPI_ADDR, buf, 16); if (flag == ERR_TRUE) { // 读取数据成功, 数据在buf中}</pre>

36.3 QSPI写数据

函数原型	INT32S QSPI_Write(INT32U addr, INT8U *p, INT32U len)
函数功能	写数据函数
入口参数	addr, 写入数据的起始地址; *p, 写入数据的存储的地址指针; 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	<pre>#define QSPI_ADDR 0x00010000 INT32S flag; INT8U buf[4] = {0x01, 0x02, 0x03, 0x04}; flag = QSPI_Write (QSPI_ADDR, buf, 4); if (flag == ERR_TRUE) { // 写入数据成功}</pre>

36.4 QSPI命令控制

函数原型	INT32S QSPI_Ctrl(INT8U Cmd, INT32U Para)
函数功能	QSPI命令控制
入口参数	Cmd: 控制命令: CMD_QSPI_READ_STATUS 0 // 读取状态, 暂时未使用 CMD_QSPI_READ_ID 1 // 读取ID CMD_QSPI_SECTOR_ERASE 2 // 擦除扇区 CMD_QSPI_RST 3 // 复位QSPI寄存器为初始状态 CMD_QSPI_CLOSE 4 // 关闭QSPI时钟, 也就是关闭SPI功能 Para, QSPI命令控制参数, 默认是0;
返回参数	ERR_TRUE, 操作成功; 其它值, 参见const.h中错误代码;
注意	无
使用说明	无

37. LTDC控制操作(ltdc.h)

37.1 初始化函数

函数原型	INT32S LTDC_Init(LTDC_PARA *pPara)
函数功能	LTDC初始化函数
入口参数	*pPara, LTDC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	LTDC参数结构LTDC_PARA, 参见ltdc.h中的定义 具体应用请参考在../libapp/ltdc_app.c中LTDC_AppInit()实现初始化, 一般客户不需要更改该函数;

37.2 画像素点

函数原型	void LCD_DrawPixel(INT8U layer, INT16U x, INT16U y, INT32U Color)
函数功能	画1个像素点
入口参数	layer, 图层: LTDC_LAYER_1, LTDC_LAYER_2; x, y: 像素点坐标; Color: 颜色代码;
返回参数	无
使用说明	在图层1的坐标(30, 100)处画一个红色的点 <pre>#define RGB(R, G, B) (((R)>>3)<<11) ((G)>>2)<<5) ((B)>>3)) // 将8位R, G, B转化为 16位RGB565格式</pre> enum //颜色定义 <pre>{COLOR_WHITE = RGB(255, 255, 255), /* 白色 */ COLOR_BLACK = RGB(0, 0, 0), /* 黑色 */ COLOR_RED = RGB(255, 0, 0), /* 红色 */ COLOR_GREEN = RGB(0, 255, 0), /* 绿色 */ COLOR_BLUE = RGB(0, 0, 255), /* 蓝色 */ COLOR_YELLOW = RGB(255, 255, 0) /* 黄色 */};</pre> <pre>LCD_DrawPixel(LTDC_LAYER_1, 30, 100, COLOR_RED);</pre>

37.3 画直线

函数原型	void LCD_DrawLine(INT8U layer, INT16U x1, INT16U y1, INT16U x2, INT16U y2, INT32U Color)
函数功能	画1条直线
入口参数	layer, 图层: LTDC_LAYER_1, LTDC_LAYER_2; x1, y1: 直线起点坐标; x2, y2: 直线终点坐标; Color: 颜色代码;
返回参数	无
使用说明	在图层1的坐标 (30, 100) 和 (200, 300) 画一条红色直线 LCD_DrawLine(LTDC_LAYER_1, 30, 100, 200, 300, COLOR_RED); 注: COLOR_RED定义见TaskLCD.c中定义

37.4 画水平放置的矩形

函数原型	void LCD_DrawRect(INT8U layer, INT16U x, INT16U y, INT16U Width, INT16U Heigh, INT32U Color)
函数功能	画水平放置的矩形
入口参数	layer, 图层: LTDC_LAYER_1, LTDC_LAYER_2; x, y: 矩形左上角坐标 Width: 矩形宽; Heigh, 矩形高; Color: 颜色代码
返回参数	无
使用说明	在图层1的起始坐标 (30, 100) 画一个宽300, 高200的红色矩形 LCD_DrawRect(LTDC_LAYER_1, 30, 100, 300, 200, COLOR_RED); 注: COLOR_RED定义见TaskLCD.c中定义

37.5 画圆

函数原型	void LCD_DrawCircle(INT8U layer, INT16U x, INT16U y, INT16U Radius, INT16U Color)
函数功能	画圆
入口参数	layer, 图层: LTDC_LAYER_1, LTDC_LAYER_2; x, y: 圆心坐标; Radius: 圆半径; Color: 颜色代码
返回参数	无
使用说明	在图层1的圆心坐标 (200, 300) 画一个半径100的红色矩形 LCD_DrawRect(LTDC_LAYER_1, 100, 300, 100, COLOR_RED); 注: COLOR_RED定义见TaskLCD.c中定义

37.6 画波形

函数原型	void LCD_DrawPoints(INT8U layer, INT16U *x, INT16U *y, INT16U Size, INT32U Color)
函数功能	绘制一组点, 并将这些点连接起来, 可用于波形显示
入口参数	layer, 图层: LTDC_LAYER_1, LTDC_LAYER_2; *x, *y: 坐标数组; Size: 坐标数组大小; Color: 颜色代码
返回参数	无
使用说明	在图层1按下面坐标数组画红色波形 INT16U xbuf[6] = { 10, 50, 80, 100, 200, 400 } ; INT16U ybuf[6] = { 20, 60, 90, 120, 240, 460 } ; LCD_DrawPoints(LTDC_LAYER_1, xbuf, yxbuf, 6, COLOR_RED) ; 注: COLOR_RED定义见TaskLCD.c中定义

37.6 画图像

函数原型	void LCD_DrawImage(INT8U layer, INT16U x, INT16U y, INT16U Width, INT16U Heigh, INT32U PixelFormat, void *pColor)
函数功能	画图像
入口参数	layer, 图层: LTDC_LAYER_1, LTDC_LAYER_2; x, y: 图像左上角的坐标 Width: 图像的宽度; Heigh: 图像的高度; PixelFormat: 图像数据格式; pColor: 图像数据指针;
返回参数	无
使用说明	无

37.8 填充矩形

函数原型	<code>void LCD_FillRect(INT8U layer, INT16U x, INT16U y, INT16U Width, INT16U Heigh, INT32U Color)</code>
函数功能	用一个颜色值填充一个矩形
入口参数	layer, 图层: LTDC_LAYER_1, LTDC_LAYER_2; x, y: 矩形左上角的坐标 Width:矩形的宽度; Heigh:矩形的高度; Color: 颜色代码;
返回参数	无
使用说明	在图层1的起始坐标 (30, 100) 画一个宽300, 高200的矩形, 并用红色填充 <code>LCD_FillRect(LTDC_LAYER_1, 30, 100, 300, 200, COLOR_RED);</code> 注: COLOR_RED定义见TaskLCD.c中定义

37.9 用一个颜色清屏

函数原型	<code>void LCD_ClearScreen(INT8U layer, INT32U Color)</code>
函数功能	用一个颜色清屏
入口参数	layer, 图层: LTDC_LAYER_1, LTDC_LAYER_2; Color: 颜色代码;
返回参数	无
使用说明	在图层1用黑色清屏 <code>LCD_ClearScreen(LTDC_LAYER_1, COLOR_BLACK);</code> 注: COLOR_BLACK定义见TaskLCD.c中定义

37. 10 LTDC命令控制

函数原型	INT32S LTDC_Ctrl (INT8U Cmd, INT8U Layer, INT32U Para)
函数功能	命令控制
入口参数	Cmd, LTDC控制命令: CMD_SET_LAYER_ALPHA: 设置图层透明度, Para范围: 0x00-0xff CMD_SET_PIXEL_FORMAT: 设置图层的像素格式, Para 可以选择: LTDC_PIXEL_FORMAT_ARGB8888, LTDC_PIXEL_FORMAT_RGB888, LTDC_PIXEL_FORMAT_RGB565, LTDC_PIXEL_FORMAT_ARGB1555, LTDC_PIXEL_FORMAT_ARGB4444, LTDC_PIXEL_FORMAT_L8, LTDC_PIXEL_FORMAT_AL44, LTDC_PIXEL_FORMAT_AL88 CMD_LTDC_RST: 复位LTDC寄存器为初始状态, Para为0 CMD_LTDC_CLOSE: 关闭LTDC时钟, 就是关闭LTDC功能, 可以省电, Para为0 Layer, LTDC层选择: LTDC_LAYER_1, LTDC_LAYER_2 Para, LTDC命令控制参数;
返回参数	无
使用说明	无

38. DMA2D控制操作(dma2d.h)

38. 1 DMA2D初始化函数

函数原型	INT32S DMA2D_Init (DMA2D_PARA *pPara)
函数功能	DMA2D初始化函数
入口参数	*pPara, DMA2D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	DMA2D参数结构DMA2D_PARA, 参见dma2d.h中的定义 具体应用请参考在../libapp/dma2d_app.c中DMA2D_AppInit()实现初始化, 一般客户不需要更改该函数;

38.2 操作函数DMA2D_CopyBuffer

函数原型	INT32S DMA2D_CopyBuffer(INT16U xSize, INT16U ySize, INT32U SrcAddr, INT16U SrcLineOffset, INT32U DstAddr, INT16U DstLineOffset, INT32U PixelFormat)
函数功能	从前景层复制指定区域的图像数据到目标区域
入口参数	xSize: 目标区域的X轴大小, 即每行像素数 ySize: 目标区域的Y轴大小, 即行数 SrcAddr: 前景层图像数据源地址 SrcLineOffset: 前景层图像的行偏移 DstAddr: 输出图像数据目的地址 DstLineOffset: 输出的行偏移 PixelFormat: 前景层图像和输出图像像素格式
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	显示RGB565位图: 显示起始坐标: (328, 20); 位图长:128; 位图高:128; 位图行偏移:LCD_WIDTH-128; 输入位图颜色: RGB565; 显示位图颜色: RGB565; DMA2D_CopyBuffer(128, 128, (INT32U) Image4_RGB565_128x128x2, 0, (LTDC_L1_START_ADDRESS+LCD_WIDTH*20*2+328*2), LCD_WIDTH-128, MA2D_FORMAT_RGB565);

38.3 操作函数DMA2D_CopyBufferPFC

函数原型	INT32S DMA2D_CopyBufferPFC(INT16U xSize, INT16U ySize, INT32U SrcAddr, INT16U SrcLineOffset, INT32U SrcPixelFormat, INT32U DstAddr, INT16U DstLineOffset, INT32U DstPixelFormat)
函数功能	从前景层复制指定区域的图像数据到目标区域并进行PFC转换
入口参数	xSize: 目标区域的X轴大小, 即每行像素数 ySize: 目标区域的Y轴大小, 即行数 SrcAddr: 前景层图像数据源地址 SrcLineOffset: 前景层图像的行偏移 SrcPixelFormat: 前景层图像像素格式 DstAddr: 输出图像数据目的地址 DstLineOffset: 输出图像的行偏移 DstPixelFormat: 输出图像像素格式
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	显示ARGB8888位图: 显示起始坐标: (176, 20); 位图长:128; 位图高:128; 位图行偏移:LCD_WIDTH-128; 输入位图颜色: ARGB8888; 显示位图颜色: RGB565;

```
DMA2D_CopyBufferPFC(128, 128, (INT32U) Image1_ARGB8888_128x128x4, 0,
DMA2D_FORMAT_ARGB8888, (LTDC_L1_START_ADDRESS+LCD_WIDTH*20*2+176*2),
LCD_WIDTH-128, DMA2D_FORMAT_RGB565);
```

38.4 操作函数DMA2D_CopyBufferMixPFC

函数原型	INT32S DMA2D_CopyBufferMixPFC(INT16U xSize, INT16U ySize, INT32U SrcAddr, INT16U SrcLineOffset, INT32U SrcPixelFormat, INT32U DstAddr, INT16U DstLineOffset, INT32U DstPixelFormat, INT32U Alpha)
函数功能	从前景层复制指定区域的图像数据到目标区域并进行PFC转换及和目标区域颜色进行混合
入口参数	xSize: 目标区域的X轴大小，即每行像素数 ySize: 目标区域的Y轴大小，即行数 SrcAddr: 前景层图像数据源地址 SrcLineOffset: 前景层图像的行偏移 SrcPixelFormat: 前景层图像像素格式 DstAddr: 输出图像数据目的地址 DstLineOffset: 输出图像的行偏移 DstPixelFormat: 输出图像像素格式 Alpha: 前景层图像透明度
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	Alpha透明度200的位图显示：位图地址：Image3_RGB565_128x128x2；位图行偏移：0；显示起始地址：(176, 168)；目标区行偏移：LCD_WIDTH-128；目标区长：128；目标区高：128；位图显示透明度：200； DMA2D_CopyBufferMixPFC(128, 128, (INT32U) Image3_RGB565_128x128x2, 0, DMA2D_FORMAT_RGB565, (LTDC_L1_START_ADDRESS+LCD_WIDTH*168*2+176*2), LCD_WIDTH-128, DMA2D_FORMAT_RGB565, 200);

38.5 操作函数DMA2D_AlphaBlendingBulk

函数原型	INT32S DMA2D_AlphaBlendingBulk(INT16U xSize, INT16U ySize, INT32U SrcFGAddr, INT16U SrcFGLineOffset, INT8U SrcFGPixelFormat, INT32U SrcBGAddr, INT16U SrcBGLineOffset, INT8U SrcBGPixelFormat, INT32U DstAddr, INT16U DstLineOffset, INT8U DstPixelFormat)
函数功能	前景层和背景层作为DMA2D输入，且支持颜色格式转换和颜色混合
入口参数	xSize: 目标区域的X轴大小，即每行像素数 ySize: 目标区域的Y轴大小，即行数 SrcFGAddr: 前景层图像数据源地址 SrcFGLineOffset: 前景层图像的行偏移 FGPixelFormat: 前景层图像像素格式 SrcBGAddr: 背景层图像数据源地址 SrcBGLineOffset: 背景层图像的行偏移 BGPixelFormat: 背景层图像像素格式 DstAddr: 输出图像数据目的地址 DstLineOffset: 输出图像的行偏移、 DstPixelFormat: 输出图像像素格式
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	两个位图混合：显示起始坐标：(24, 168)；位图长：128；位图高：128；前景层位图颜色：ARGB8888；前景层行偏移：0；背景层位图颜色：ARGB8888；背景层行偏移：0；位图行偏移：LCD_WIDTH-128；输出显示位图颜色：RGB565； DMA2D_AlphaBlendingBulk(128, 128, (INT32U) Image1_ARGB8888_128x128x4, 0, DMA2D_FORMAT_ARGB8888, (INT32U) Image2_ARGB8888_128x128x4, 0, DMA2D_FORMAT_ARGB8888, (LTDC_L1_START_ADDRESS+LCD_WIDTH*168*2+24*2), LCD_WIDTH-128, DMA2D_FORMAT_RGB565)；

38.6 操作函数DMA2D_Fill

函数原型	INT32S DMA2D_Fill(INT16U xSize, INT16U ySize, INT32U DstAddr, INT16U DstLineOffset, INT32U ColorIndex, INT32U PixelFormat)
函数功能	颜色填充
入口参数	xSize: 要填充区域的X轴大小，即每行像素数 ySize: 要填充区域的Y轴大小，即行数 DstAddr: 图像数据目的地址 DstLineOffset: 输出图像的行偏移 ColorIndex: 颜色值 PixelFormat: 图像像素格式
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	使用DMA2D刷色块：显示起始坐标：(24, 20)；色块长:128；色块高:128；色块行偏移:LCD_WIDTH-128；色块颜色：CL_GREEN DMA2D_Fill(128, 128, (LTDC_L1_START_ADDRESS+LCD_WIDTH*20*2+24*2), LCD_WIDTH-128, CL_GREEN, DMA2D_FORMAT_RGB565);

38.7 DMA2D命令控制

函数原型	INT32S DMA2D_Ctrl(INT8U Cmd, INT32U Para)
函数功能	DMA2D命令控制
入口参数	Cmd, DMA2D控制命令： CMD_DMA2D_STATUS: 读取状态，Para为0，返回ERR_TRUE表示完成传输(DMA2D已经空闲)，返回ERR_BUSY表示正在传输当中； CMD_DMA2D_RST: 复位寄存器为初始状态，Para为0 CMD_DMA2D_CLOSE: 关闭时钟，也就是关闭DMA2D功能，可以省电，Para为0 Para, DMA2D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	无

39. MDMA控制操作 (mdma. h)

39.1 MDMA初始化函数

函数原型	INT32S MDMA_Init(MDMA_PARA *pPara)
函数功能	MDMA初始化函数
入口参数	*pPara, MDMA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	MDMA参数结构MDMA_PARA, 参见mdma. h中的定义 具体应用请参考在../libapp/mdma_app. c中MDMA_AppInit() 实现初始化, 一般客户不需要更改该函数;

39.2 MDMA通道初始化函数

函数原型	INT32S MDMA_ChxInit(INT8U ch, MDMA_CH_PARA *pPara)
函数功能	MDMA通道初始化
入口参数	ch范围: MDMA_CH1-MDMA_CH16; pPara, MDMA 通道初始化参数;
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	MDMA通道参数结构MDMA_CH_PARA, 参见mdma. h中的定义 具体应用请参考在../libapp/mdma_app. c中MDMA_AppInit() 实现初始化, 一般客户不需要更改该函数;

39.3 MDMA启动传输函数

函数原型	INT32S MDMA_StartTransfer(INT8U ch, INT32U SrcAddress, INT32U DstAddress, INT32U BlockDataLength, INT32U BlockCount, INT8U isWait, INT32U TimeOut)
函数功能	MDMA启动传输
入口参数	ch范围: MDMA_CH1-MDMA_CH16 SrcAddress, 源数据地址; DstAddress, 目的数据地址; BlockDataLength, 块数据长度; BlockCount, 块数量; isWait: 1, 等待传输完成返回; 0, 不等待, 启动后直接返回; 当isWait=1时, TimeOut定义超时时间, 单位ms
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	无

39.4 MDMA命令控制

函数原型	INT32S MDMA_Ctrl(INT8U ch, INT8U Cmd, void *pPara)
函数功能	MDMA命令控制
入口参数	Cmd, MDMA控制命令: CMD_MDMA_STATUS: 读取状态, ch范围: MDMA_CH1-MDMA_CH16, Para为0, 返回ERR_TRUE表示完成传输(MDMA已经空闲), 返回ERR_BUSY表示正在传输当中; CMD_MDMA_REGISTER_IRQHandler: 注册通道中断处理回调函数, ch范围: MDMA_CH1-MDMA_CH16, Para, 中断函数指针 CMD_MDMA_RST: 复位寄存器为初始状态, ch为0, Para为0 CMD_MDMA_CLOSE: 关闭时钟, 也就是关闭MDMA功能, 可以省电, ch为0, Para为0 Para, DMA2D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	无

40. JPEG控制操作(jpeg.h)

40.1 JPEG初始化函数

函数原型	INT32S JPEG_Init(JPEG_PARA *pPara)
函数功能	JPEG初始化函数
入口参数	*pPara, JPEG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	JPEG参数结构JPEG_PARA, 参见jpeg.h中的定义 具体应用请参考在../libapp/jpeg_app.c中JPEG_AppInit()实现初始化, 一般客户不需要更改该函数;

40.2 JPEG启动解码函数

函数原型	INT32S JPEG_SatrtDecode(INT32U SrcAddress, INT32U SrcDataLength, INT32U DestAddress, INT32U isWait, INT32U TimeOut)
函数功能	JPEG启动解码
入口参数	SrcAddress, JPEG源数据地址; DestAddress, 解码后数据目的地址; SrcDataLength, 源数据长度; isWait: 1, 等待解码完成返回; 0, 不等待, 启动解码后直接返回; 当isWait=1时, TimeOut定义超时时间, 单位ms
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	无

40.3 JPEG命令控制

函数原型	INT32S JPEG_Ctrl (INT8U Cmd, void *pPara)
函数功能	JPEG命令控制
入口参数	Cmd, JPEG控制命令: CMD_JPEG_STATUS: 读取状态, pPara为0 CMD_JPEG_INFO: 读取转换后图像信息, pPara是传入的信息指针 CMD_JPEG_RST: 复位JPEG寄存器为初始状态, pPara为0 CMD_JPEG_CLOSE: 关闭时钟, 也就是关闭MDMA功能, 可以省电, pPara为0 pPara, JPEG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	无

41. IPCC异和通信(ipcc.h)

41.1 IPCC初始化函数

函数原型	INT32S IPCC_Init(void)
函数功能	IPCC初始化函数
入口参数	无
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	无

41.2 IPCC命令控制

函数原型	INT32S IPCC_Ctrl(INT8U Cmd, INT8U ch, INT8U dir, void *pPara)
函数功能	IPCC命令控制
入口参数	Cmd, 控制命令: CMD_IPCC_NotifyCPU: 通知远程处理器; ch范围: IPCC_CH1-IPCC_CH6; dir=IPCC_TX_DIR, 表示解除发送通道空闲中断屏蔽(使能发送空闲中断); dir=IPCC_RX_DIR, 表示解除接收通道被占用中断屏蔽(使能接收通道被占用中断); CMD_IPCC_GetStatus: 读取IPCC通道状态; ch范围: IPCC_CH1-IPCC_CH6; dir=IPCC_TX_DIR, 表示读取发送通道状态; dir=IPCC_RX_DIR, 表示读取接收通道状态; CMD_IPCC_Callback: 设置IPCC通道中断回调函数; dir=IPCC_CH_TX_DIR是发送通道空闲中断回调函数; dir=IPCC_CH_RX_DIR是接收通道被占用中断回调函数; Para, 参数指针; 在CMD_IPCC_NotifyCPU和CMD_IPCC_GetStatus时为0, 没有用; 在CMD_IPCC_Callback是回调函数指针
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	无

附录：驱动库更新内容说明

序号	版本	时间	更新内容
1	V1. 20	2022. 6. 17	正式发布。 注：该版本驱动库是在原有V1. 10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；
2	V1. 20. 06	2022. 11. 15	(1) 修改DAC中函数DAC_Ctrl读取状态错误问题 (2) 修改PWM中函数PWM_Ctrl读取状态错误问题 (3) 在Modbus_Init初始化函数中：增加部分参数；增加Modbus_RxProc和Modbus_Ctrl两个函数；修改原有的各个函数，可以实现发送数据和返回数据分离执行，加快函数执行效率；实现多路Modbus通道并行操作； (4) 将函数Modbus_WriteSingleReg()/Modbus_WriteMulCoils()/Modbus_WriteMulReg()/Modbus_ReadWriteHoldReg()分别更名为：Modbus_WriteSingleHoldReg()/Modbus_WriteCoils()/Modbus_WriteHoldReg()/Modbus_ReadWriteHoldReg()； 参数不变，原函数名可以继续使用(建议新名字)； (5) 修改StrCopy、StrCopy2、StrCopy3、StrCopy4、StrCopy5函数，增加返回值，返回总共拷贝的字符总数
3	V1. 30. 00	2025. 7. 1	因为增加STM32H7XX和STM32MP15X系列模块而增加和修改部分内容
4	V1. 31. 00	2025. 12. 1	增加QSPI/LPTimer/SDRAM/MDMA/DMA2D/JPEG/LTDC/IPCC功能