

AMKN 软件开发平台 使用手册

(2026 年 7 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AMKN软件开发平台(适用驱动库版本：V1. 20)	2022. 6. 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，文档版本和测试程序版本保持一至，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. 00	修改部分内容，适用于AMKN软件版本：V1. 31. XX	2025. 12. 1	研发部
V1. 40. 00	修改部分内容，适用于AMKN软件版本：V1. 40. XX	2026. 7. 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、AMKN8701G、AMKN8702G、AMKN8703、AMKN8804、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

第一章. AMKN 软件开发框架文件结构	5
1. AMKN 软件开发框架简介:	5
2. AMKN 软件开发框架基本功能:	5
3. AMKN 软件框架结构图	6
4. AMKN 软件主体结构介绍	7
5. AMKN 软件文件结构表	12
第二章 AMKN 软件配置文件说明	24
1. CONST.H 说明	24
2. CONFIG.H 说明	24
3. FLASH 及 RAM 分配文件说明:	24
第三章 AMKN 软件各模块编程使用说明	26
1. DI 输入编程说明:	26
2. DO 输出控制编程说明:	28
3. KEY 按键输入编程说明:	31
4. SW 拨码开关输入编程说明:	33
5. UART 收发数据编程说明:	35
6. CAN 收发数据编程说明:	40
7. ADC(AI) 输入编程说明:	46
8. DAC(AO) 输出编程说明:	51
9. FCLK 脉冲输入编程说明:	53
10. PWM 脉冲输出编程说明:	61
11. TIM 定时器编程说明:	65
12. EXTI 外部中断编程说明:	67
13. SPI 总线编程说明:	68
14. I ² C 总线编程说明:	69
15. RTC 时钟编程说明:	70
16. EEPROM 读写编程说明:	71
17. SPIFLASH 读写编程说明:	72
18. NET 网络通信编程说明:	74
19. USB 通信编程说明:	79
20. SD 卡读写编程说明:	83
21. NAND FLASH 读写编程说明:	86
22. MODBUS 主机通信编程说明:	88
23. MODBUS SLAVE 从机通信编程说明:	90
24. WIFI 通信编程说明:	93
25. 外部器件(传感器)编程说明:	99
26. AT 指令编程说明:	106
27. LPTIM 定时器编程说明:	107
28. SDRAM 编程说明:	108
29. DMA2D 编程说明:	109
30. LTDC 编程说明:	112
31. MDMA 编程说明:	116
32. JPEG 编程说明:	119
33. QSPI 编程说明:	120
34. LVGL 编程说明	122
35. OPENAMP 编程说明	123
附录: 文档修改记录	125

第一章. AMKN 软件开发框架文件结构

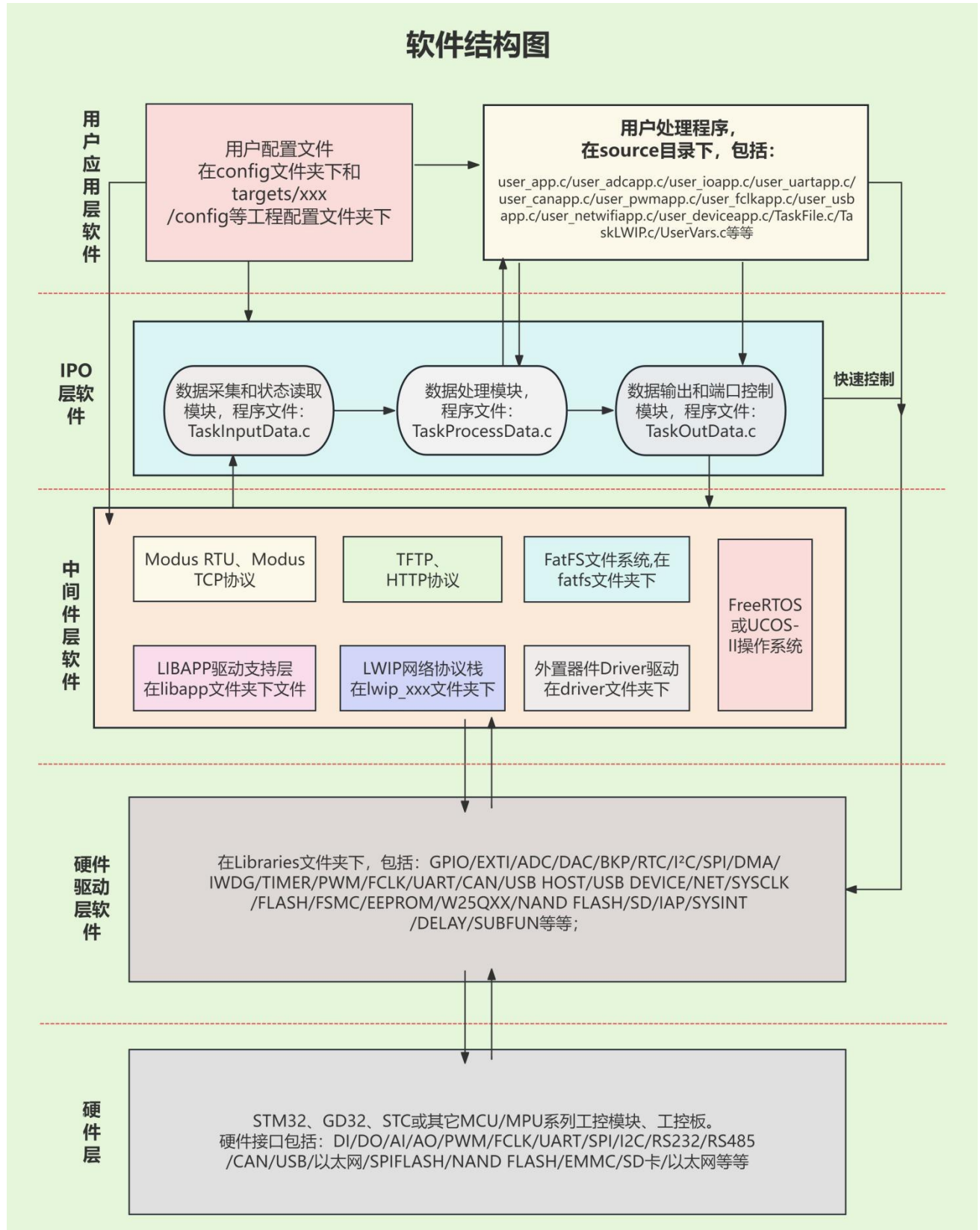
1. AMKN 软件开发框架简介:

AMKN 软件开发框架是北京中嵌凌云公司总结多年嵌入式软件开发经验，归纳总结的用户应用软件开发框架结构。这个平台以 STM32FXXX/H7XX/MP15X 系列和 GD32FXXX 系列工控模块、工控板为硬件平台，以 AMKN 驱动库为基础开发的应用软件开发框架结构。在这个平台上开发应用软件具有精简高效、使用简单、稳定可靠、兼容性好等特点，非常适合快速开发软件。在以下说明中 AMKN 软件开发框架简称：AMKN 软件。

2.AMKN 软件开发框架基本功能:

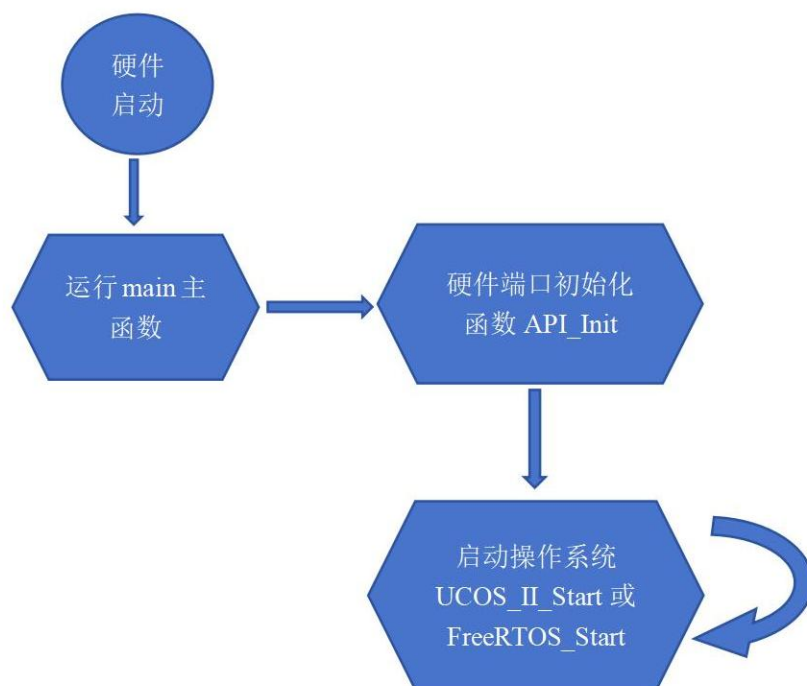
完成对硬件端口 DI/AI/FCLK 的数据采集；
完成对硬件端口 DO/AO/PWM 输出端口的控制；
完成对 UART/RS232/RS485/USB/CAN/以太网/WIFI 通信的接收和发送；
完成对 EEPROM/SD 卡/NAND FLASH/SPI FLASH/U 盘等存储设备的文件读写；
完成 IPO 数据流程控制。IPO 流程控制包括：I 是 Input：数据输入和状态输入；P 是 Process：数据处理，回调用户应用程序；O 是 Output：数据输出和控制；
完成参数配置模块；
完成用户编程接口统一标准化。

3. AMKN软件框架结构图



4. AMKN 软件主体结构介绍

(1) 主程序入口流程图（程序源码：source/main.c）



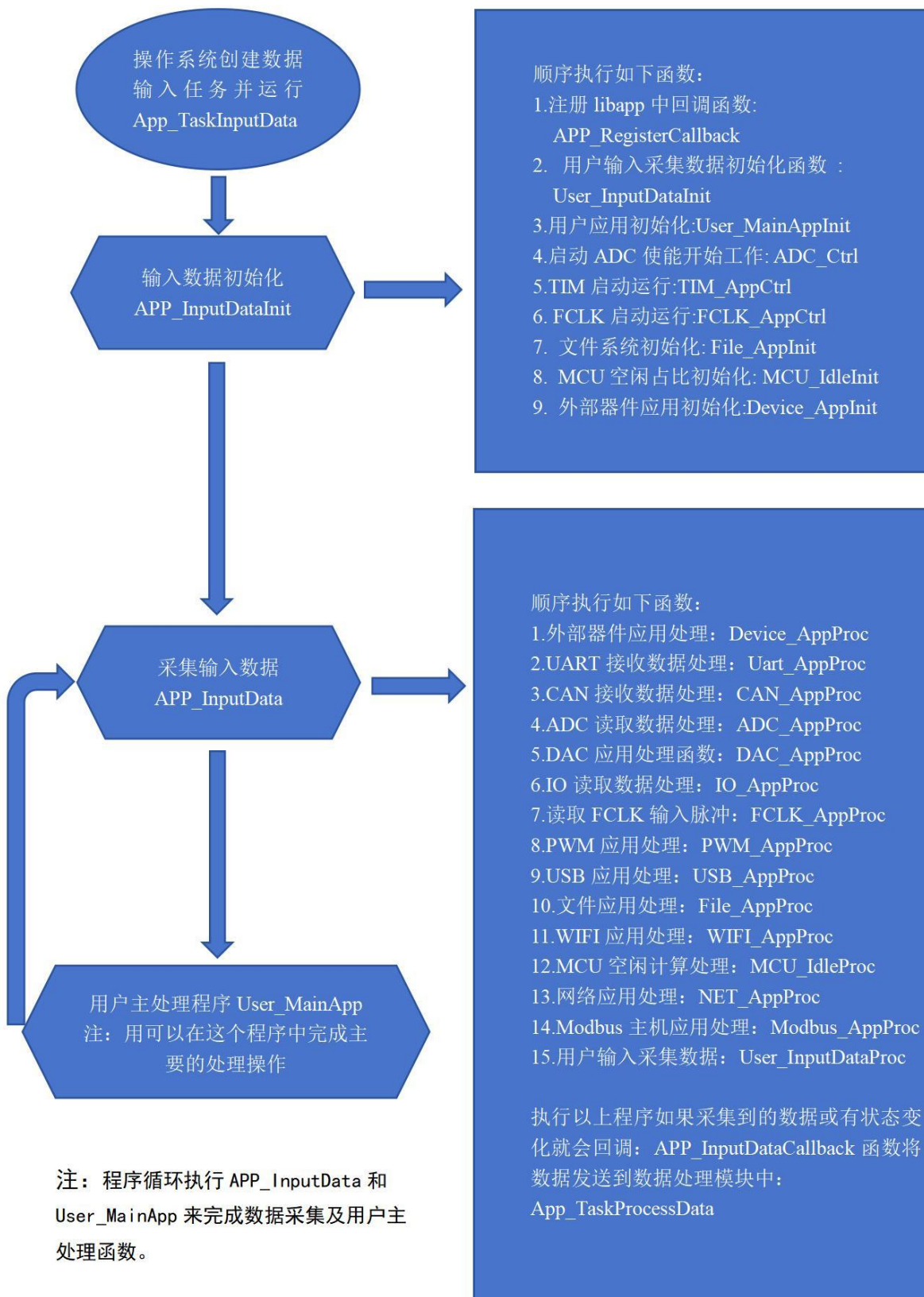
注意：UCOS_II_Start 或 FreeRTOS_Start 运行后将控制权交给操作系统，该函数永远不会返回！

(2) 硬件端口初始化流程（程序源码：labapp/labapp.c）

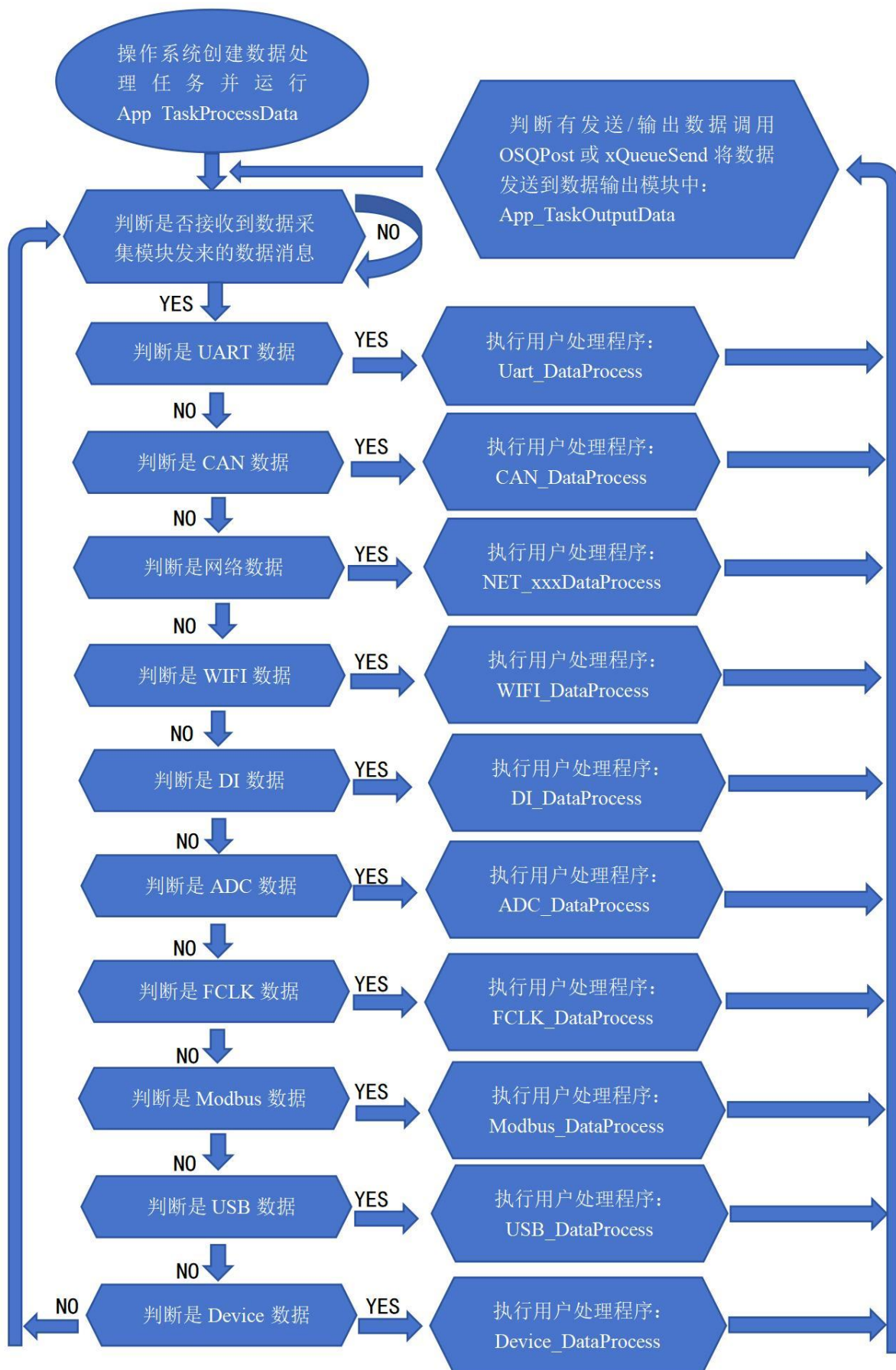
注：在启动操作系统前必须初始化所有硬件端口，只有初始化了硬件端口，后面才可以使用



(3) 数据输入和读取状态流程图：（程序源码：source/TaskInputData.c）



(4) 数据处理流程图 (程序源码: source/TaskProcessData.c)



(5) 数据输出和端口控制流程图（程序源码：source/TaskOutputData.c）



5. AMKN软件文件结构表

序号	文件夹	文件	描述
1	config 系统配置 这个目录是整个软件平台结构配置目录，用户可以根据实际需要来配置	const.h	常量定义，用户不可更改
		config.h	全局配置文件，本文件包含以下各个配置文件
		product_config	工控板选型配置文件，如应用AMKN8602G工控板则需要如下定义： #define PRODUCT_TYPE AMKN8602G
		dma_config.h	本文件是DMA部分配置文件，用户不需要修改这个文件
		can_config.h	本文件是CAN滤波器部分配置文件，用户可以修改这个文件
		debug_config.h	本文件是DEBUG调试输出配置文件，用户根据实际需要修改这个文件
		test_config.h	本文件是测试配置文件
		iap_config.h	本文件是IAP配置文件
		modbus_config.h	本文件是modbus主机应用配置文件
		modbus_slave_config.h	本文件是modbus从机应用配置文件
		irq_priority_config.h	本文件是中断优先级配置文件
		mdma_config.h	本文件是MDMA部分配置文件，用户可以修改这个文件
		middlewares_config.h	中间件配置文件
2	targets	AMKN8602G目录： config/AMKN8602G_Config.h、 config/AMKN8602G_I0Config.h	里面包含工控板AMKN8602G的工程文件/配置文件（config里）/输出文件（output里） 工控板AMKN8602G配置文件和I0配置文件
	各型号工控板MDK工程文件及配置文件，用户根据需要	AMKN8626目录： config/AMKN8626_Config.h config/AMKN8626_I0Config.h	里面包含工控板AMKN8626的工程文件/配置文件（config里）/输出文件（output里） 工控板AMKN8626配置文件和I0配置文件
		AMKN8628目录： config/AMKN8628_Config.h	里面包含工控板AMKN8628的工程文件/配置文件（config里）/输出文件（output里）

	自行修改	config/AMKN8628_I0Config.h	工控板AMKN8628配置文件和I0配置文件
		AMKN8629目录: config/AMKN8629_Config.h config/AMKN8629_I0Config.h	里面包含工控板AMKN8629的工程文件/配置文件 (config里)/输出文件 (output里) 工控板AMKN8629配置文件和I0配置文件
		AMKN8630目录: config/AMKN8630_Config.h config/AMKN8630_I0Config.h	里面包含工控板AMKN8630的工程文件/配置文件 (config里)/输出文件 (output里) 工控板AMKN8630配置文件和I0配置文件
		AMKN8638目录: config/AMKN8638_Config.h config/AMKN8638_I0Config.h	里面包含工控板AMKN8638的工程文件/配置文件 (config里)/输出文件 (output里) 工控板AMKN8638配置文件和I0配置文件
		AMKN8639目录: config/AMKN8639_Config.h config/AMKN8639_I0Config.h	里面包含工控板AMKN8639的工程文件/配置文件 (config里)/输出文件 (output里) 工控板AMKN8639配置文件和I0配置文件
		还有其它工控板这里就不再列举, 请查看例程源码	
3	source 用户程序	main.c	系统主程序
		vars.c ./inc/vars.h	全局公共变量, 由厂家维护, 不建议用户修改这里; 如果增加全局变量请在UserVars.h中定义
		uservars.c ./inc/uservars.h	应用程序全局常量和变量定义文件, 用户应用程序的全局常量和变量在这里定义
	用户可修改本目录下程序来完成软件设计	ISRHook.c	系统中断处理文件, 在这里处理所有中断及驱动库钩子函数
		TaskInputData.c 注: 一般不用修改	本文件负责所有输入数据采集并发送到TaskProcessData中处理
		TastProcessData.c 注: 一般不用修改	本文件输入输出处理任务, 所有由TaskInputData任务及其它任务发来的数据都有这个任务处理
		TastOutputData.c	本文件负责数据输出, 实现UART/CAN/NET/WIFI

	注：一般不用修改	等数据发送功能；
	TastLWIP.c 注：一般不用修改	本文件负责网络协议栈LWIP处理；
	TaskFile.c	本文件负责文件系统读写任务处理
	TastEventCtrl.c	本文件负责事件控制处理功能
	TaskZqxyCtrl.c	本文件负责中嵌凌云自定义通信协议处理
	TaskUserCtrl.c	本文件负责用户定义控制功能任务处理
	TaskModbus.c	本文件负责用户Modbus主机通信任务处理
	TaskTest.c	本文件负责用户测试功能任务处理
	TaskLCD.c	本文件负责LCD显示任务
	TaskOpenAMP.c	本文件负责OpenAMP处理任务
	TaskCanOpen.c	本文件负责CanOpen处理任务
	TaskStepMotor.c	本文件负责步进电机处理任务
	comfun.c ./inc/comfun.h	本文件是用户公共应用函数
	user_ioapp.c	本文件是用户IO类数据应用程序
	user_uartapp.c	本文件是UART数据用户处理程序
	user_canapp.c	本文件是用户处理CAN输入数据应用程序
	user_adcapp.c	本文件是用户处理ADC采集数据应用程序
	user_fclapp.c	本文件是用户处理FCLK输入数据应用程序
	user_netwifiapp.c	本文件是用户网络和WIFI数据处理应用程序
	user_usbapp.c	本文件是用户处理USB输入数据应用程序
	user_deviceapp.c	本文件是用户处理外部器件应用程序
	user_app.c ./inc/user_app.h	本文件是用户应用处理主程序
	user_testapp.c	本文件是用户测试应用程序
	./inc/modbus_holdreg_config.h	本文件是ModbusRTU寄存器定义
	./inc/configpara.h	全局存储参数配置文件
	./inc/user_config.h、 user_const.h、user_data.h	本文件是用户APP配置、常量及数据结构头文件
	user_pwmapp.c	本文件是用户处理PWM输出应用程序

		./inc/user_pwmapp.h	
		user_inputdata.c	本文件是用户输入采集数据应用程序
		user_virtuartapp.c	本文件是虚拟串口应用程序
		user_stepmotor.c	本文件是步进电机用户应用程序
		./inc/user_stepmotor.h	
		./image/image.c	本文件是测试用图像数据
		./inc/image.h	
4	libapp 驱动库应 用处理程 序 注：一般 用户不要 修改这个 目录下的 文件，由 我司维护	./fonts/asc12.c、asc16.c、 asc24.c、asc32.c、hz12.c、 hz16.c、hz24.c、hz32.c ./inc/fonts.h	各种字体文件
		libappvars.c ./inc/libappvars.h	本文件负责libapp目录下应用程序公共全局变量引用头文件，用户不要把自己的全局变量定义到这里，这部分由中嵌凌云厂家维护
		libapp.c ./inc/libapp.h	本文件是驱动库总的硬件初始化函数，参数存储及公共应用处理函数
		irqhandler.c	本文件是驱动库系统中断处理函数
		liboshook.c ./inc/liboshook.h	本文件负责驱动库调用操作系统的钩子函数
		adc_app.c	本文件是ADC驱动应用处理程序
		can_app.c	本文件是CAN驱动应用处理程序
		dac_app.c	本文件是DAC驱动应用处理程序
		eeeprom_app.c	本文件是EEPROM驱动应用处理程序
		exti_app.c	本文件是EXTI驱动应用处理程序
		fclk_app.c	本文件是FCLK驱动应用处理程序
		io_app.c	本文件是IO驱动应用处理程序
		modbus_app.c	本文件是MODBUS主机驱动应用处理程序
		modbus_slave_app.c	本文件是MODBUS从机驱动应用处理程序
		net_app.c	本文件是网络驱动库应用处理程序
		pwm_app.c	本文件是PWM驱动应用处理程序

		rtc_app.c	本文件是RTC驱动应用处理程序
		spiflash_app.c	本文件是SPIFLASH驱动应用处理程序
		tim_app.c	本文件是TIMER驱动应用处理程序
		uart_app.c	本文件是UART驱动应用处理程序
		file_app.c	本文件是文件操作应用处理程序
		fatfs_app.c	本文件是文件系统FatFS应用处理程序
		I ² C_app.c	本文件是I ² C驱动应用处理程序
		spi_app.c	本文件是SPI驱动应用处理程序
		usb_app.c	本文件是USB驱动应用处理程序
		dma_app.c	本文件是DMA驱动应用处理程序
		lptim_app.c	本文件是LPTIM驱动应用处理程序
		qspi_app.c	本文件是QSPI驱动应用处理程序
		sdram_app.c	本文件是SDRAM驱动应用处理程序
		ltdc_app.c	本文件是LTDC驱动应用处理程序
		dma2d_app.c	本文件是DMA2D驱动应用处理程序
		mdma_app.c	本文件是MDMA驱动应用处理程序
		jpeg_app.c	本文件是JPEG驱动应用处理程序
		tftp.c	本文件是TFTP协议驱动程序，参见《STM32Fxxx系列工控板(模块)AT指令_1.04_20250701.pdf》
		at.c	本文件是AT指令驱动程序
		./inc/xxxx_app.h	以上文件的头文件目录
5	libraries 基础驱动库 注：此文件目录下文件不可	sysint.h	系统驱动库头文件
		gpio.h	GPIO通用IO硬件驱动程序头文件
		exit.h	外部中断硬件驱动程序头文件
		rtc.h	RTC 实时时钟硬件驱动程序头文件
		iwdg.h	独立看门狗硬件驱动程序头文件
		uart.h	UART 异步串口 (RS232 或 RS485) 硬件驱动程序头文件
		I ² C.h	I ² C 总线硬件驱动程序头文件

更改, 由 我司进行 维护	spl.h	SPI 总线硬件驱动程序头文件
	can.h	CAN 总线硬件驱动程序头文件
	timer.h	定时器(PWM/FCLK) 硬件驱动程序头文件
	lptimer.h	lptim 定时器硬件驱动程序头文件
	dac.h	DAC 硬件驱动程序头文件
	adc.h	ADC 硬件驱动程序头文件
	flash.h	内部 FLASH 硬件驱动程序头文件
	crc.h	CRC 校验硬件驱动程序头文件
	sd.h	SD卡(SPI) 读写硬件驱动程序头文件
	At45dbxx.h	AT45DBXX系列FLASH读写硬件驱动程序头文件
	w25qxx.h	W25QXX系列FLASH读写硬件驱动程序头文件
	eprom.h	EEPROM读写硬件驱动程序头文件
	dma.h	DMA控制器硬件驱动程序头文件
	usbhost.h	USB主机硬件驱动程序头文件
	usbdevice.h	USB设备硬件驱动程序头文件
	net.h	网络接口硬件驱动程序头文件
	iap.h	IAP固件更新驱动程序头文件
	fsmc.h	Fsmc 外部总线驱动程序头文件
	ltcd.h	LTDC 控制器硬件驱动程序头文件
	sdram.h	SDRAM 硬件驱动程序头文件
	dma2d.h	DMA2D 硬件驱动程序头文件
	mdam.h	MDMA 硬件驱动程序头文件
	jpeg.h	JPEG 硬件驱动程序头文件
	ipcc.h	IPCC 硬件驱动程序头文件
	nflash.h	Nand Flash 读写硬件驱动程序头文件
	delay.h	延时函数驱动程序头文件
	subfun.h	常用功能函数驱动程序头文件
	STM32F107VC_V14x.xxxx.lib	STM32F107VC模块驱动库文件
	STM32F103VE_V14x.xxxx.lib	STM32F103VE模块驱动库文件

		STM32F103ZE_V14x. xxxx. lib	STM32F103ZE模块驱动库文件
		STM32F407XX_V14x. xxxx. lib	STM32F407XX系列模块驱动库文件
		STM32H743XX_V14x. xxxx. lib	STM32H743XX系列模块驱动库文件
		STM32H723XX_V14x. xxxx. lib	STM32H723XX系列模块驱动库文件
		STM32MP15X_V14x. xxxx. lib	STM32MP15X系列模块驱动库文件
		GD32F307XX_V14x. xxxx. lib	GD32F307XX模块驱动库文件
		GD32F303VE_V14x. xxxx. lib	GD32F303VE模块驱动库文件
		GD32F303ZE_V14x. xxxx. lib	GD32F303ZE模块驱动库文件
6	device 用户一些 外部器件 驱动程序	device_app.c/device_app.h	本文件是外部器件应用操作程序
		device.c	该文件包含外部器件驱动程序，只加载该文件就可以，其它驱动文件不必加载
		device_config.h	本文件是用户一些外部器件驱动配置文件
		device_exfun.h	引用外部函数
		ch455目录: ch455.c ch455_app.c/ch455_config.h	CH455按键控制采集芯片驱动程序
		i ² c_io目录: i ² c_io.c/i ² c_io.h	I0模拟I ² C总线驱动程序
		shtxx目录: shtxx_config.h shtxx.c/shtxx_app.c	shtxxx系列温湿度传感器驱动程序
		wifi目录: wifi_config.h wifi.c/wifi_app.c	wifi模块驱动程序
		max6675目录: max6675_config.h max6675.c/max6675_app.c	MAX6675芯片操作驱动函数文件
		sp106目录: sp106.c/sp106.h	SPL06芯片操作驱动函数文件
		dac8562目录: dac8562_config.h dac8562.c/dac8562_app.c	2通道16位DAC转换芯片驱动函数文件
		ads1232目录: ads1232_config.h ads1232.c/ads1232_app.c	2通道24位ADC转换芯片驱动函数文件
		ads1220目录: ads1220_config.h ads1220.c/ads1220_app.c	4通道24位ADC转换芯片驱动函数文件
		ch9434目录: ch9434_config.h	SPI转4个UART芯片CH9434驱动函数文件

		ch9434. c/ch9434_app. c	
		tm1640目录:tm1640_config. h tm1640. c/tm1640_app. c	LED数码管控制芯片TM1640驱动函数文件
		wtn6xxx目录:wtn6xxx_config. h wtn6xxx. c/wtn6xxx_app. c	WTN6xxx系列语音播放芯片驱动函数文件
		ad5175目录:ad5175_config. h ad5175. c/ad5175_app. c	数字电位器AD5175驱动函数文件
		ad7682目录:ad7682_config. h ad7682. c/ad7682_app. c	4通道16位ADC转换芯片驱动函数文件
		dac8554目录:dac8554_config. h dac8554. c/dac8554_app. c	4通道16位DAC转换芯片驱动函数文件
		dac128s085目录:dac128s085. c dac128s085_app. c dac128s085_config. h	8通道12位DAC转换芯片驱动函数文件
		mcp3208目录:mcp3208_config. h mcp3208. c/mcp3208_app. c	8通道12位ADC转换芯片驱动函数文件
		mcp4728目录:mcp4728_config. h mcp4728. c/mcp4728_app. c	4通道12位DAC转换芯片驱动函数文件
		w5500目录:w5500_config. h w5500. c/w5500_app. c	SPI转以太网芯片驱动函数文件
		by8302目录:by8302_config. h by8302. c/by8302_app. c	语音播放芯片驱动函数文件
		max31865目 录:max31865_config. h max31865. c/max31865_app. c	MAX31865芯片操作驱动函数文件
7	fatfs 文件系统	源码目录: ./middlewares/third_party/fatfs/source/	
		ff. c、ff. h	fatfs文件系统源代码, 用户一般不用修改
		diskio. c、diskio. h	fatfs硬件接口驱动文件, 包括SPI FLASH/SD卡/U 盘/Nand Flash的文件系统接口驱动; 已经做好,

			客户不要修改
		integer.h	数据类型定义头文件一般不用修改
		ffconf.h	Fatfs文件系统配置文件, 客户可以修改
8	init 初始化 注: 用户 不要修改 这个目录 下的任何 文件	inita.s	系统初始化汇编文件
		freertos/vectors.s	支持freertos操作系统的中断向量表 vectors.s: 适用于cortex_m3和cortex_m4内核
		freertos/vectors_m7.s	vectors_m7.s: 适用于cortex_m7内核
		freertos/vectors_m4_mp15x.s	vectors_m4_mp15x.s: 适用于 stm32mp15x 的 cortex_m4内核
		ucos_ii/vectors.s	支持ucos_ii操作系统的中断向量表 vectors.s: 适用于cortex_m3和cortex_m4内核
		ucos_ii/vectors_m7.s	vectors_m7.s: 适用于cortex_m7内核
		ucos_ii/vectors_m4_mp15x.s	vectors_m4_mp15x.s: 适用于 stm32mp15x 的 cortex_m4内核
		nos/vectors.s	支持无操作系统的中断向量表 vectors.s: 适用于cortex_m3和cortex_m4内核
9	lwip TCP/IP协 议栈	源码目录: ./middlewares/third_party/lwip/lwip-2.1.3 ./middlewares/third_party/lwip/lwip-1.4.1 UCOS-II 操作系统例程使用 lwip-1.4.1; 其它例程使用 lwip-2.1.3	
		lwip.c	网络协议栈源文件
		sys_arch.c sys_arch.h	lwip移植到ucos上接口驱动文件
		ethernetif.c	LWIP网络硬件驱动接口文件
		lwipopts.h	lwip配置文件, 可以替代opt.h
		opt.h	lwip默认配置文件
10	ucos-ii 操作系统	源码目录: ./middlewares/third_party/ucos-ii	
		ucos-ii.c	ucos-ii源文件

	注：这部分可选	cpu_a. asm, os_cpu_a. asm, lib_mem_a. asm	ucos-ii 汇编移植汇编文件
11	ucos_app	源码目录：./middlewares_app/ucos_app	
	操作系统	os_cfg. h	ucos操作系统配置文件
	配置及变	app_cfg. h	应用任务配置文件
	量定义目	Includes. h	操作系统引用头文件
	录	OSHook. c/OSHook. h	操作系统应用钩子函数处理文件
	注：这部分可选	OSVar. c/OSVar. c	操作系统全局变量定义文件
		ucos_app. c/ucos_app. h	本文是ucos应用处理程序
12	freertos 操作系统 注：这部分可选	源码目录：./middlewares/third_party/freertos/V10.5.1	
		Croutine. c、event_groups. c、 heap_4. c、list. c、queue. c、 portmacro. h、stream_buffer. c、 tasks. c、timers. c	freertos源文件
		目录include	引用有文件
		目录CM3、CM4、M7	不同内核操作系统接口文件
13	freertos	源码目录：./middlewares_app/freertos_app	
	_app操作	FreeRTOSConfig. h	freertos操作系统配置文件
	系统配置	app_cfg. h	应用任务配置文件
	及变量定	OSVar. c/OSVar. h	操作系统全局变量定义文件
	义目录	FreeRTOS_App. c	本文是freertos应用处理程序
	注：这部分可选		
14	lvgl 轻量级通用图形库源码	源码目录：./middlewares/third_party/lvgl/lvgl-8.3.10/	
		/src	源码目录
		/examples	图形实例目录
		/demos	测试demo
		lv_conf. h, lvgl. h	配置头文件和引用头文件
15	lvgl_app	源码目录：./middlewares_app/lvgl_app	
	轻量级图	lvgl_config. h	应用配置文件

	形库应用	lvgl_app.c、lvgl_app.h	应用程序及头文件
	程序	/generated、/custom	用户测试例程目录
16	CANOpen 主机通信 源码	源码目录：./middlewares/third_party/canopen	
		/src、/inc	源码目录及引用头文件目录
		/dictionary/Master.c、 Master.h	映射变量声明源文件及头文件
		/objdictedit	目标编辑软件
		/dot	用户参考文档
17	CANOpen 主机通信 应用软件	源码目录：./middlewares_app/canopen_app	
		canopen.c/canopen.h	CANOpen 核心模块源文件及头文件
		canopen_config.h	全局配置文件
		canopen_def.h	CANOpen 常量定义文件, 0x1000-0x1FFF 的词典索引, PDO 索引, PDO 传输类型
		canopen_motor.c canopen_motor.h	电机控制应用层, 包含 PDO 配置和电机控制逻辑; 电机控制 API 头文件
		canopen_motor_config.h canopen_motor_def.h	电机配置文件 电机定义文件, DS402 常用索引
		./source/TaskCanOpen.c	CANOpen 任务主循环, 包含初始化和周期性处理
18	OpenAMP 源码	异核通信源码目录：./middlewares/third_party/openamp 主要针对 STM32MP157 和 STM32MP257 间异核通信	
19	OpenAMP 应用程序	异核通信应用程序目录：./middlewares_app/openamp_app	
		openamp_conf.h openamp_config.h	OpenAMP应用配置文件
		openamp.c/openamp.h	OpenAMP应用主接口文件头文件
		openamp_app.c/openamp_app.h	OpenAMP应用处理程序及头文件
		virtuart_app.c/virtuart_app.h	OpenAMP虚拟串口应用处理程序及头文件
		mbox_ipcc.c/mbox_ipcc.h	IPCC邮箱通信程序及头文件
		rsc_table.c/rsc_table.h	OpenAMP共享内存资源表程序及头文件
		openamp_log.c/openamp_log.h	信息打印输出

20	stepmotor源码	步进电机驱动源码目录：./middlewares/amkn/stepmotor/	
		stepmotor.c/stepmotor.h	步进电机驱动源码程序及头文件
21	stepmotor控制应用程序	步进电机控制应用程序目录：./middlewares_app/stepmotor_app	
		stepmotor_app.c	步进电机应用控制程序及头文件
		stepmotor_app.h	
		stepmotor_modbus_app.c	ModbusRTU协议控制步进电机
		stepmotor_modbus_holdreg_config.h	ModbusRTU控制寄存器定义
		stepmotor_config.h	步进电机控制总配置
		amkn8626_stepmotor_config.h	各个工控板配置
		amkn8628_stepmotor_config.h	
		amkn8629_stepmotor_config.h	
		amkn8630_stepmotor_config.h	
		amkn8638_stepmotor_config.h	
		amkn8639_stepmotor_config.h	
		amkn8804_stepmotor_config.h	
		amknxxxx_stepmotor_config.h	
22	dot	readme.txt	软件说明文档

第二章 AMKN 软件配置文件说明

1. const.h 说明

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

2. config.h 说明

这个文件是整个软件框架总配置文件, 包括以下配置:

(1) UCOS-II、FreeRTOS 操作系统配置

(2) IAP 配置文件 iap_config.h

MODBUS 主机配置文件 modbus_config.h

(4) MODBUS 从机配置文件 modbus_slave_config.h

(5) CANOPEN 配置文件 canopen_config.h

(6) 产品型号配置文件 product_config.h

(7) CAN 过滤器组配置文件 can_config.h

(8) DEBUG 输出配置文件 debug_config.h

(9) DMA1/DMA2 系统配置文件 dma_config.h

(10) 外部器件配置文件 device_config.h

(11) 用户测试配置文件 test_config.h

(12) 用户中断优先级配置文件 irq_priority_config.h

(13) MDMA 配置文件 mdma_config.h

(14) 中间件配置文件 middlewares_config.h

3. FLASH 及 RAM 分配文件说明:

注: 以下文件分别在 ./target/AMKNXXX/config 目录下

STM32F107VC_Flash.scat: 工控模块 STM32F107VC, FLASH 及 RAM 分配文件

STM32F107VC_FlashIAP.scat: 工控模块 STM32F107VC, FLASH 及 RAM 分配文件(编译可生成带 IAP 功能固件)

STM32F103VE_Flash.scat: 工控模块 STM32F103VE, FLASH 及 RAM 分配文件

STM32F103VE_FlashIAP.scat: 工控模块 STM32F103VE, FLASH 及 RAM 分配文件(编译可生成带 IAP 功

能固件)

STM32F103ZE_Flash.scat: 工控模块 STM32F103ZE, FLASH 及 RAM 分配文件

STM32F103ZE_FlashIAP.scat: 工控模块 STM32F103ZE, FLASH 及 RAM 分配文件 (编译可生成带 IAP 功能固件)

STM32F407XE_Flash.scat: 工控模块 STM32F407VE/ZE, FLASH 及 RAM 分配文件

STM32F407XE_FlashIAP.scat: 工控模块 STM32F407VE/ZE, FLASH 及 RAM 分配文件 (编译可生成带 IAP 功能固件)

GD32F307XE_Flash.scat: 工控模块 GD32F307VE, FLASH 及 RAM 分配文件

GD32F307XE_FlashIAP.scat: 工控模块 GD32F307VE, FLASH 及 RAM 分配文件 (编译可生成带 IAP 功能固件)

GD32F303XE_Flash.scat: 工控模块 GD32F303VE, FLASH 及 RAM 分配文件

GD32F303XE_FlashIAP.scat: 工控模块 GD32F303VE, FLASH 及 RAM 分配文件 (编译可生成带 IAP 功能固件)

GD32F303ZE_Flash.scat: 工控模块 GD32F303ZE, FLASH 及 RAM 分配文件

GD32F303ZE_FlashIAP.scat: 工控模块 GD32F303ZE, FLASH 及 RAM 分配文件 (编译可生成带 IAP 功能固件)

STM32H723XX_Flash.scat: 工控模块 STM32H723ZG, FLASH 及 RAM 分配文件

STM32H723XX_FlashIAP.scat: 工控模块 STM32H723ZG, FLASH 及 RAM 分配文件 (编译可生成带 IAP 功能固件)

STM32H743XX_Flash.scat: 工控模块 STM32H743XI, FLASH 及 RAM 分配文件

STM32H743XX_FlashIAP.scat: 工控模块 STM32H743XI, FLASH 及 RAM 分配文件 (编译可生成带 IAP 功能固件)

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

第三章 AMKN 软件各模块编程使用说明

1. DI 输入编程说明:

(1) 关键词定义如下: DI_EN、DI_MODE、DI_SCAN_T、DI_ATOUT_T、DI_NUM、DI1~DI32、DI1_ID~DI64_ID、DI1FLAG~DI64FLAG、DI1_8SPI_EN、DI9_16SPI_EN、DI17_24SPI_EN、DI25_32SPI_EN、HC597_STB、HC597_LOAD、HC597_CS

(2) 在 IO 配置文件 AMKNXXX_IOConfig.h 中定义: DI 的端口及 DI 数量, 参考如下:

```
#define DI_NUM    2    // 定义 DI 输入数量
```

```
#define DI1       PF0
```

```
#define DI2       PF1
```

定义 2 个 DI, 分别对应硬件 PF0 和 PF1 端口

(3) 在功能配置文件 AMKNXXX_Config.h 中定义, 参考如下:

```
#define DI_EN      1    // DI 使能, 1: 打开使能, 0: 关闭; 这个参数必须为 1
```

```
#define DI_MODE    0    // 设置 DI 工作模式: 0, 实时回调函数读取+查询读取模式双模式;  
                        // 1, 查询读取模式;
```

```
#define DI_SCAN_T  10    //设置定时扫描时间间隔, 单位: ms; 如果想快速扫描 DI 端口可以将该  
                        //参数设置小一些, 但最小是 1ms
```

```
#define DI_ATOUT_T 3000  // 设置自动输出时间间隔, 单位: ms, 这个参数可以修改
```

(4) 在../libapp/io_app.c 中查看 IO_AppInit() 和 DI_LibAppVarsInit() 初始化函数, 这些函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

(5) 在../libapp/io_app.c 中查看 IO_AppProc() 处理函数:

```
#if ((DI_EN > 0) && (DI_NUM > 0))    // DI 使能条件编译  
  
tick = APP_GetSubTick(LibAppVars.DI.Scan_t); // 读取和上次采集的时间间隔, 单位 ms  
  
if (tick >= DI_SCAN_T)    // 计算和上次采集时间间隔大于等于设置值  
{  
  
    LibAppVars.DI.Scan_t += tick;    // 记录本次采集时间  
  
    DI_Scan();    // 扫描 DI 输值, 并发送
```

```
#if ((DI1_8SPI_EN > 0) || (DI9_16SPI_EN > 0) || (DI17_24SPI_EN > 0) || (DI25_32SPI_EN > 0))  
    DI_SPIRead();                // 扫描 SPI 转 DI 输值，并发送  
  
#endif  
  
}  
  
#endif
```

这部分代码是 DI 采集处理，里面增加了防抖处理，用户不需要修改该代码；当 DI_MODE 配置为 0 时 DI 采集结果通过 LibAppVars.Register.APP_InputDataCallback 回调函数发送出来。

(6) DI 采集的结果被发送到 ./source/user_ioapp.c 中的 DI_DataProcess 函数

在这个函数中将 DI 的相关数据存储到 pVars->DI 的相关变量中，并调用 User_AppDIProc 进行处理。

用户可以在 User_AppDIProc 这个函数中编写自己的处理程序。

当用户把 DI_MODE 配置为 1 (0 也可以) 时，也可以直接调用 DI_Read() 或 DI_MulRead() 读取 DI 输入值。

2. D0 输出控制编程说明:

(1) 关键词定义如下: D0_EN、D0_SCAN_T、D0_NUM、D01~D0128、D0_OUT_MODE、D01_ID~D0128_ID、D01FLAG~D0128FLAG、D01_INIT_VAL~D0128_INIT_VAL、D01_8SPI_EN、D09_16SPI_EN、D017_24SPI_EN、D025_32SPI_EN、HC595_STB、HC595_ENA

(2) 在 IO 配置文件 AMKNXXX_IOConfig.h 中定义: D0 的端口及 D0 数量, 参考如下:

```
#define D0_NUM    2    // 定义 D0 输出端口数量

#define D01      PF0

#define D02      PF1

#define D0_OUT_MODE IO_OUT_PP // 定义 D0 输出端口模式: IO_OUT_PP 为推挽输出模式; IO_OUT_OD 为开路输出模式

#define D01_INIT_VAL 0 // 定义 D01 输出初始化值

#define D02_INIT_VAL 0 // 定义 D02 输出初始化值
```

定义 2 个 D0, 分别对应硬件 PF0 和 PF1 端口

(3) 在功能配置文件 AMKNXXX_Config.h 中定义, 参考如下:

```
#define D0_EN    1    // D0 使能, 1: 打开使能, 0: 关闭; 这个参数必须为 1

#define D0_SCAN_T 1 // 设置定时扫描时间间隔, 单位: ms; 这个参数可以修改
```

(4) 在../libapp/io_app.c 中查看 IO_Applnit() 和 D0_LibAppVarsInit() 初始化函数, 这函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

(5) 在../libapp/io_app.c 中有 2 个独立函数控制 D0 输出:

控制单路 D0 输出函数: D0_Write(INT8U id, INT8U val)

id, D0 标识 D01_ID~D0128_ID; val, 输出值, 0 或者 1;

比如控制 D02 输出 1 则这样写: D0_Write(D02_ID, 1);

控制多路 D0 同时输出函数: D0_MulWrite(INT8U Group, INT32U D0Flag, INT8U val)

Group 为 0 时: D0Flag, bit0~bit31 依序代表 D01~D032, 如果是 1 表示该 D0 执行输出; val,

输出值, 0 或者 1;

比如控制 D01 D02 同时输出 1 则这样写: `DO_MulWrite(D0_GROUP1_ID, D01FLAG|D02FLAG, 1);`

注意: 这 2 个函数在任何位置和时间点都可以使用。

(6) 在 `../libapp/io_app.c` 中有 1 个 D0 事件控制函数:

```
void DO_EventCtrl(INT8U id, INT8U Cmd, INT16U t1, INT16U t2)
```

`id`, D0 标识 `D01_ID~D032_ID`;

`Cmd`, 控制命令:

```
#define LIBAPP_DO_CMD_IDLE 0x00 // 停止命令, 无命令在执行
```

```
#define LIBAPP_DO_CMD_ON_T 0x01 // 控制 D0 输出 1, 延时 t 毫秒后恢复原输出 0
```

```
#define LIBAPP_DO_CMD_OFF_T 0x02 // 控制 D0 输出 0, 延时 t 毫秒后恢复原输出 1
```

```
#define LIBAPP_DO_CMD_NEG_T1 0x03 // 控制 D0 连续翻转输出: D0 先输出 t1
```

// 毫秒的 1, 再输出 t2 毫秒 0, 如此循环

```
#define LIBAPP_DO_CMD_NEG_T2 0x04 // 控制 D0 连续翻转输出: D0 先输出 t1 // 毫秒的 0, 再输出 t2
```

毫秒 1, 如此循环

`t1, t2`: 输出延时, 根据 `Cmd` 命令不同, `t1, t2` 定义不同, 如下:

`LIBAPP_DO_CMD_IDLE`: `t1, t2` 设置为 0;

`LIBAPP_DO_CMD_ON_T`, `LIBAPP_DO_CMD_OFF_T`: `t1` 设置为延时时间, `t2` 设置为 0;

`LIBAPP_DO_CMD_NEG_T1`: `t1` 为 D0 输出 1 时间, `t2` 为 D0 输出 0

`LIBAPP_DO_CMD_NEG_T2`: `t1` 为 D0 输出 0 时间, `t2` 为 D0 输出 1

比如控制 D02 输出 1 个 50ms 高电平脉冲:

```
DO_EventCtrl(D02_ID, LIBAPP_DO_CMD_ON_T, 50, 0);
```

比如控制 D02 持续输出: 50ms 高电平, 1000ms 低电平:

```
DO_EventCtrl(D02_ID, LIBAPP_DO_CMD_NEG_T1, 50, 1000);
```

注意: 调用 `DO_EventCtrl()` 函数后立即执行操作, 但后面结束操作或翻转操作是

靠 `IO_AppProc()` 处理函数的下面代码来完成的:

```
#if (DO_EN > 0) // D0 使能条件编译
```

```
if (LibAppVars.D0.Flag > 0)           // D0 工作使能标志
{
    tick = APP_GetSubTick(LibAppVars.D0.Scan_t); // 读取和上次采集间隔时间, 单位 ms
    if (tick >= D0_SCAN_T) // 计算和上次采集时间间隔大于等于设置值
    {
        LibAppVars.D0.Scan_t += tick; // 记录本次采集时间
        D0_EventProc(tick);           // D0 输出事件处理
    }
}
else
{
    LibAppVars.D0.Scan_t = LibAppVars.Tick; // 记录本次采集时间
}
#endif
```

3. KEY 按键输入编程说明:

(1) 关键词定义如下: KEY_EN、KEY_MODE、KEY_SCAN_T、KEY_CDOWN_T、KEY_NUM、KEY1~KEY32、KEY1_ID~KEY32_ID、KEY1FLAG~KEY32FLAG

(2) 在 IO 配置文件 AMKNXXXX_IOConfig.h 中定义: KEY 的端口及 KEY 数量, 参考如下:

```
#define KEY_NUM    2    // 定义 KEY 按键数量

#define KEY1      PF0

#define KEY2      PF1
```

定义 2 个 KEY, 分别对应硬件 PF0 和 PF1 端口

(3) 在功能配置文件 AMKNAMKNXXXX_Config.h 中定义, 参考如下:

```
#define KEY_EN    1 // DI 使能, 1: 打开使能, 0: 关闭; 这个参数必须为 1

#define KEY_MODE 0 // 设置 KEY 工作模式: 0, 实时回调函数读取+查询读取双模式; 1, 查询读取模式;

#define KEY_SCAN_T 10 //设置定时扫描时间间隔, 单位: ms; 这个参数可以修改

#define KEY_CDOWN_T (KEY_SCAN_T*100) // 设置按键连续按下输出时间间隔, 单位: ms; 注意: 必须是
                                     // 定时扫描时间的整数倍, 设置为 0 表示不输出
```

(3) 在 ../libapp/io_app.c 中查看 IO_AppInit() 和 KEY_LibAppVarsInit() 初始化函数, 这函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

(4) 在 ../libapp/io_app.c 中查看 IO_AppProc() 处理函数:

```
// 按键读取处理

#if ((KEY_EN > 0) && (KEY_NUM > 0)) // KEY 使能条件编译

tick = APP_GetSubTick(LibAppVars.KEY.Scan_t); // 读取和上次采集的间隔// 时间, 单位 ms

if (tick >= KEY_SCAN_T) // 计算和上次采集时间间隔大于等于设置值
{
    LibAppVars.KEY.Scan_t += tick; // 记录本次扫描时间

    Key_Read(); // 读取按键
}

#endif
```

这部分代码是 KEY 采集处理，里面增加了防抖处理，用户不需要修改该代码；当 KEY_MODE 配置为 0 时，KEY 采集结果通过 LibAppVars.Register.APP_InputDataCallback 回调函数发送出来。

(5) KEY 采集的结果被发送到../source/user_ioapp.c 中的 KEY_DataProcess 函数

在这个函数中调用 Key_UserProcess()，用户请在这个函数根据 KEY 值做相应的处理；

当用户把 KEY_MODE 配置为 1 (0 也可以) 时，也可以直接调用 Key_Read() 返回按键值。

4. SW 拨码开关输入编程说明:

(1) 关键词定义如下: SW_EN、SW_MODE、SW_SCAN_T、SW_ATOUT_T、SW_NUM、SW1~SW32、SW1_ID~SW32_ID

(2) 在 IO 配置文件 AMKNXXXX_IOConfig.h 中定义: SW 的端口及 SW 数量, 参考如下:

```
#define SW_NUM    2    // 定义拨码开关位置
```

```
#define SW1      PF0
```

```
#define SW2      PF1
```

定义 2 个 SW, 分别对应硬件 PF0 和 PF1 端口

(3) 在功能配置文件 AMKNXXXX_Config.h 中定义, 参考如下:

```
#define SW_EN    1    // DI 使能, 1: 打开使能, 0: 关闭; 这个参数必须为 1
```

```
#define SW_MODE  0    // 设置 SW 工作模式: 0, 实时回调函数读取+查询读取双模式; 1, 查询读取模式;
```

```
#define SW_SCAN_T 10 //设置定时扫描时间间隔, 单位: ms; 这个参数可以修改
```

```
#define SW_ATOUT_T 3000 // 设置自动输出时间间隔, 单位: ms
```

(4) 在../libapp/io_app.c 中查看 IO_AppInit() 和 SW_LibAppVarsInit() 初始化函数, 这些函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

(5) 在../libapp/io_app.c 中查看 IO_AppProc() 处理函数:

```
// SW 拨码开关读取处理
```

```
#if ((SW_EN > 0)&&(SW_NUM > 0))                // SW 使能条件编译
```

```
tick = APP_GetSubTick(LibAppVars.SW.Scan_t);    // 读取和上次采集的间隔时间, // 单位 ms
```

```
if (tick >= SW_SCAN_T) // 计算和上次采集时间间隔大于等于设置值
```

```
{
```

```
    LibAppVars.SW.Scan_t += tick;                // 记录本次扫描时间
```

```
    SW_Scan();                                    // 拨码开关扫描处理
```

```
}
```

```
#endif
```

这部分代码是 SW 采集处理, 里面增加了防抖处理, 用户不需要修改该代码; 当 SW_MODE 配置为 0 时, SW 采集结果通过 LibAppVars.Register.APP_InputDataCallback 回调函数发送出来。

(6) SW 采集的结果被发送到../source/user_ioapp.c 中的 SW_DataProcess 函数

在这个函数中调用 User_AppSWProc(), 用户请在这个函数根据 SW 值做相应的处理:

当用户把 SW_MODE 配置为 1 (0 也可以) 时, 也可以直接调用 SW_Read() 返回拨码开关值。

5. UART 收发数据编程说明:

(1) 关键词定义如下: UARTx_EN、UARTx_RXMODE、UARTx_SCAN_T、UARTx_RX_TIMEOUT、UARTx_BAUD、UARTx_WORD_LENGTH、UARTx_STOP_BITS、UARTx_PARITY、UARTxTX_DMA_EN、UARTxRX_DMA_EN、UARTx_RXBUF_SIZE、UARTx_TXBUF_SIZE、UARTx_TX、UARTx_RX、UARTx_DIR、UARTx_DIR_HL

注意: x 范围是 1~10, 硬件模块不同, x 值也不同

以下说明以工控板 EMB8602 为例。

(2) 在 IO 配置文件 AMKNXXX_IOConfig.h 中定义: 参考如下:

// UART1(管脚)功能重映射设置: 转成 RS232 接口(JP8 的 TX1、RX1)

```
#define UART1_REMAP  UART_REMAP_1  // UART1 重映射 1
```

```
#define UART1_TX      PB6             // 设置 TX 管脚, 对应 JP8 的 1 脚 TX1
```

```
#define UART1_RX      PB7             // 设置 RX 管脚, 对应 JP8 的 2 脚 RX1
```

```
#define UART1_DIR     IO_NONE        // 设置 RS485 方向控制 IO, 没有转 RS485 接口则设置为 IO_NONE
```

```
#define UART1_DIR_HL  0              // 定义 RS485 通信时接收数据时方向电平, 0: 低电平接收; 1: 高电平接收
```

// UART2(管脚)功能重映射设置: 转成 RS232 接口(JP8 的 TX2、RX2)

```
#define UART2_REMAP  UART_REMAP_1  // UART2 重映射 1
```

```
#define UART2_TX      PD5             // 设置 TX 管脚, 对应 JP8 的 3 脚 TX2
```

```
#define UART2_RX      PD6             // 设置 RX 管脚, 对应 JP8 的 4 脚 RX2
```

```
#define UART2_DIR     IO_NONE        // 设置 RS485 方向控制 IO, 没有转 RS485 接口则设置为 IO_NONE
```

```
#define UART2_DIR_HL  0              // 定义 RS485 通信时接收数据时方向电平, 0: 低电平接收; 1: 高电平接收
```

// UART3(管脚)功能重映射设置: 转成 RS232 接口(JP8 的 TX3、RX3)

```
#define UART3_REMAP  UART_REMAP_2  // UART3 重映射 2
```

```
#define UART3_TX      PD8             // 设置 TX 管脚, 对应 JP8 的 5 脚 TX3
```

```
#define UART3_RX      PD9             // 设置 RX 管脚, 对应 JP8 的 6 脚 RX3
```

```
#define UART3_DIR     IO_NONE        // 设置 RS485 方向控制 IO, 没有转 RS485 接口则设置为 IO_NONE
```

```
#define UART3_DIR_HL  0              // 定义 RS485 通信时接收数据时方向电平, 0: 低电平接收; 1: 高电平接收
```

// UART4(管脚)功能重映射设置: 转成 RS485 接口(JP9 的 A+、B-)

```
#define UART4_REMAP  UART_REMAP_0  // UART4 没有重映射
```

```
#define UART4_TX      PC10            // 设置 TX 管脚
```

```
#define UART4_RX      PC11            // 设置 RX 管脚
```

```
#define UART4_DIR    PE3    // 设置 RS485 方向控制 IO, 没有转 RS485 接口则设置为 IO_NONE

#define UART4_DIR_HL  0      // 定义 RS485 通信时接收数据时方向电平, 0: 低电平接收; 1: 高电平接收

// UART5(管脚) 功能重映射设置: 转成 RS232 接口(JP8 的 TX5、RX5)

#define UART5_REMAP  UART_REMAP_0  // UART5 没有重映射

#define UART5_TX      PC12          // 设置 TX 管脚, 对应 JP8 的 7 脚 TX5

#define UART5_RX      PD2          // 设置 RX 管脚, 对应 JP8 的 8 脚 RX5

#define UART5_DIR     IO_NONE  // 设置 RS485 方向控制 IO, 没有转 RS485 接口// 则设置为 IO_NONE

#define UART5_DIR_HL  0  // 定义 RS485 通信时接收数据时方向电平, 0: //低电平接收; 1: 高电平接收

注意: 一般 UARTx_REMAP、UARTx_TX、UARTx_RX 这些设置是根据具体硬件而改变; 强烈建议按我司工控板的
硬件设计来固定这些端口, 以保证软件的一致性。
```

(3) 在功能配置文件 AMKNXXXX_Config.h 中定义, UART1 为例参考如下:

```
#define UART1_EN      1    // UART1 使能, 1: 打开使能, 0: 关闭

#if (UART1_EN > 0)

#define UART1_RXMODE  0    // 接收数据工作模式设置: 0, UART_RXMODE_SCAN;
                        // 1, UART_RXMODE_IRQ; 2, UART_RXMODE_ISRHOOK;

#define UART1_SCAN_T  10   // 设置定时扫描时间间隔, 单位: ms

#define UART1_BAUD     115200 // 设置波特率, 可以设置: 1200, 2400, // 4800, 9600, 19200,
                        // 38400, 57600, 115200

#define UART1_WORD_LENGTH 0 // 设置数据字长, 0: 8bit; 1: 9bit;

#define UART1_STOP_BITS 0 // 设置停止位, 0: 1bit; 1: 2bit; 2: 0.5bit; 3: 1.5bit;

#define UART1_PARITY   0 // 设置奇偶检验位, 0: 无校验; 1: 偶校验; // 2: 奇校验;

#define UART1_FIFO_EN  1 // 设置发送接收 FIFO 使能, 1: 打开使能, 0: 关闭;

#if (UART1_FIFO_EN > 0)

#define UART1_RX_TIMEOUT 20 // 定义接收超时时间, 单位: us;

#endif

#define UART1TX_DMA_EN 0 // 设置发送 DMA 使能, 1: 打开使能, 0: 关闭;

#define UART1RX_DMA_EN 0 // 设置接收 DMA 使能, 1: 打开使能, 0: 关闭;

#define UART1_RXBUF_SIZE 256 // 设置接收缓存长度, 范围大于 0, 根据自己实际需要设置;
```

```
#define UART1_TXBUF_SIZE 1024 // 设置发送缓存长度, 范围大于 0, 根据自己实际需要设置;  
#endif
```

用户根据实际设计需要修改以上设置参数。

(4) 在../libapp/uart_app.c 中查看 Uart_Applnit() 初始化函数, 这函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

(5) 在../libapp/uart_app.c 中查看 Uart_RxProc 处理函数:

```
void Uart_RxProc(void)  
{  
    INT32U tick;  
    //-----UART1 接收处理-----  
    #if ((UART1_EN > 0)&&(UART1_RX_EN > 0)) // 条件编译 UART1 使能  
    if ((LibAppVars.Uart[UART1_ID].Flag&UART_RX_DISABLE_FLAG) == 0) // 判断 UART1 接收没有关闭  
    {  
        #if (UART1_RXMODE==UART_RXMODE_SCAN)//条件编译扫描接收数据模式  
        tick = APP_GetSubTick(LibAppUart[UART1_ID].Scan_t); //读取和上次采集的间隔时间, 单位 ms  
        if (tick >= UART1_SCAN_T) // 计算和上次采集时间间隔大于等于设置值  
        {  
            LibAppUart[UART1_ID].Scan_t += tick; // 记录本次扫描时间  
            Uart_SCANRxProc(UART1_ID); // UART1 扫描接收数据处理  
        }  
        #endif  
        #if (UART1_RXMODE == UART_RXMODE_IRQ) // 条件编译中断接收数据模式  
        Uart_IRQRxProc(UART1_ID); // UART1 中断接收数据处理  
        #endif  
    }  
    #endif  
    ... ..  
}
```

这部分代码是 UART1 接收数据处理，用户不需要修改该代码；接收到的数据通过 LibAppVars.Register.APP_InputDataCallback 回调函数发送出来。

(6) UART1 接收数据被发送到../source/user_uartapp.c 中的 Uart_DataProcess 函数

在这个函数中调用 Uart1_UserProcess(), 用户请在这个函数做相应的处理:

```
void Uart1_UserProcess(INT8U *p, INT16U len)
{
    INT16U i;

    #if (AT_EN > 0)                // 条件编译 AT 指令使能,
    AT_Proc(p, len);              // AT 指令处理函数
    #else                          // 没有使能 AT 指令, 则按正常数据处理

        #if (APP_UART_RX_DEBUG_EN > 0) // 条件编译使能接收数据打印输出
            //打印数据输出到调试串口(注意: 按 16 进制数据打印)

            printf("AT+UART1=RX[%d]:", len);

            for(i=0; i<MIN(len-1, DEBUG_MAX_DATA_LEN); i++)
            { printf("%02x ", p[i]); }

            printf("%02x\r\n", p[i]);
        #endif

        // 复制接收数据到缓存中

        for(i=0; i<MIN(len, UART1_RXBUF_SIZE); i++)
        { pVars->Uart1.RxBuf[i] = p[i]; }

        pVars->Uart1.Rxlen = len;
        pVars->Uart1.Flag |= AMKN_UART_UPDATE_FLAG;

        // 用户在这里处理 UART1 接收到的数据: 数据缓存 pVars->Uart1.RxBuf[], 数据长度
        // pVars->Uart1.Rxlen
        //

        #if (APP_UART1_TEST_EN > 0) // 测试功能: 将数据在原样发送回去

        #if (OS_EN == 0)

            Uart_Write(UART1_ID, pVars->Uart1.RxBuf, len);

        #else
```

```
    Uart_UserSendData(UART1_ID, pVars->Uart1.RxBuf, len, AMKN_NOCOPY);  
  
    #endif  
  
    #endif  
  
#endif  
}
```

注意：Uart_UserSendData 这个函数是发送数据函数；

UART 发送数据函数在../source/comfun.c 中的 Uart_UserSendData，在有操作系统下用户可以调用这个函数发送数据。

6. CAN 收发数据编程说明:

(1) 关键词定义如下: CAN_x_EN、CAN_x_MODE、CAN_x_RXMODE、CAN_x_SCAN_T、CAN_x_IDE、CAN_x_BAUD、CAN_x_RTR、CAN_x_RXBUF_SIZE、CAN_x_TXBUF_SIZE、CAN_x_TX、CAN_x_RX

注意: x 范围是 1~3, 硬件模块不同, x 值也不同

以下说明以工控板 EMB8602 的 CAN1 为例。

(2) 在 IO 配置文件 AMKNXXXX_IOConfig.h 中定义: 参考如下:

```
#define CAN1_TX    PD1    // CAN1 设置 TX 管脚  
#define CAN1_RX    PD0    // CAN1 设置 RX 管脚
```

注意: 一般 CAN_x_REMAP、CAN_x_TX、CAN_x_RX 这些设置是根据具体硬件而改变; 强烈建议按我司工控板的硬件设计来固定这些端口, 以保证软件的一致性。

(3) 在功能配置文件 AMKNXXXX_Config.h 中定义, CAN1 为例参考如下:

```
#define CAN1_EN      1    // CAN1 使能, 1: 打开使能, 0: 关闭  
#if (CAN1_EN > 0)  
#define CAN1_MODE    0  
// 在 STM32F1XX/4XX/GD32F3XXX 模块中: 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);  
// 在 STM32H7XX/STM32MP15X 模块中: 0, 正常模式; 1, 受限操作模式; 2, 总线监控模式;  
// 3, 内部环回模式(用于调试); 4, 外部环回模式(用于调试)  
  
#define CAN1_RXMODE    0 // 接收消息工作模式设置: 0, CAN_RXMODE_SCAN; 1, CAN_RXMODE_IRQ;  
#define CAN1_SCAN_T    1 // 设置定时扫描时间间隔, 单位: ms  
#define CAN1_IDE      CAN_EXT_ID // 帧类型: 0, 标准帧: CAN_STD_ID; 1, 扩展帧: CAN_EXT_ID;  
#define CAN1_RTR      CAN_RTR_DATA // 选择数据帧: 0, CAN_RTR_DATA; 远程帧: 1, CAN_RTR_REMOTE;  
#define CAN1_BAUD      1000000 // CAN1 波特率;  
#define CAN1_RXBUF_SIZE    16 // CAN 接收缓存可接收消息个数, 范围 1~256  
#define CAN1_TXBUF_SIZE    16 // CAN 发送缓存可发送消息个数, 范围 1~256
```

注: CAN1_RXBUF_SIZE、CAN1_TXBUF_SIZE 只在 STM32F1XX/4XX/GD32F3XXX 模块中有效

(4) 在 STM32H7XX/STM32MP15X 模块新增:

```
#define CAN1_DATA_BAUD (2*CAN1_BAUD) // CAN1 数据波特率, 注意这个波特率在 CAN 帧模式是
// CAN_FRAME_FD_BRS(切换比特率)模式才有效该数据波特率可以是通常波特率 1-5 倍, 通常设置 2 倍
#define CAN1_FRAME_FORMAT 1 // CAN 帧模式: 0, CAN_FRAME_CLASSIC:CAN 标准模式
// 1, CAN_FRAME_FD_NO_BRS:FDCAN 模式, 但不切换比特率;
// 2, CAN_FRAME_FD_BRS:FDCAN 模式, 切换比特率: 注意暂时不支持这个模式
#define CAN1_TXDATA_SIZE 64 // 发送数据域大小: 8/12/16/20/24/32/48/64
#define CAN1_RXDATA_SIZE 64 // 接收数据域大小: 8/12/16/20/24/32/48/64
#endif
```

用户根据实际设计需要修改以上设置参数。

(5) 在../libapp/can_app.c 中查看 CAN_Applnit() 初始化函数, 这函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

在../config/can_config.h 配置滤波器设置

(6) 在 STM32F1XX/4XX/GD32F3XXX 模块中 CAN 过滤器组配置:

```
#define CAN2_START_BANK 14 // CAN2 开始的滤波器组, 范围是 1~27; 可以固定为 14, 表示 0-13
// 为 CAN1 滤波器组, 表示 14-27 为 CAN2 滤波器组;
#define CAN_FILTER_SCALE 0x0FFFFFFF // CAN 过滤器位宽寄存器: Bit27~Bit0 有效, bit0 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。 注意这个必须固定位 32 位, 不可更改;
#define CAN_FILTER_FIFO 0x0AAAAAAA // CAN 过滤器 FIFO 关联配置, 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; 用户可不
//用修改这个配置: 第偶数组 0/2/.../26 过滤器关联到 FIFO0, 第奇数组 1/3/.../27 过滤器关联到 FIFO1,
#if (CAN1_EN > 0) #define CAN1_FOACT_EN 1 // CAN1 过滤器 0 组配置: 1, 使能; 0, 关闭
#if (CAN1_FOACT_EN > 0) //-----过滤器组 0 设置
#define CAN1_FOMODE 1 // CAN 过滤器模式: 0: 2 个 32 位寄存器工作在标识符掩码(屏蔽位)
// 模式; 1: 2 个 32 位寄存器工作在标识符列表模式
#define CAN1_FOIDE CAN_EXT_ID // 帧类型: 0, 标准帧:CAN_STD_ID; 1, 扩展帧:CAN_EXT_ID;
#define CAN1_FORTR CAN_RTR_DATA // 选择数据帧: 0, CAN_RTR_DATA; 选择远程帧: 1, CAN_RTR_REMOTE;
// 在标识符列表模式下可以设置 2 个可识别 ID, 如下: 标识符 ID 是 1 和 2
#if (CAN1_FOMODE == 1)
```

```

#define CAN1_F0R1          0x00000001

#define CAN1_F0R2          0x00000002

#endif

// 在标识符掩码(屏蔽位)模式设置可识别 ID, 标识符 ID 范围: 0x00000000~0x000000FF 可进行以下设置
#if (CAN1_FOMODE == 0)

#define CAN1_F0R1          0x000000FF  // 标识符 ID 值
#define CAN1_F0R2          0x0FFFFFF0  // 掩码设置

#endif

#endif

//-----过滤器组 1 设置
... ..

//-----过滤器组 13 设置
... ..

#endif

```

(7) 如果 CAN 只接收小于 28 个设备数据时, 可以将 CAN1_FOMODE~CAN1_F13MODE 设置为 1 (标识符列表模式), CAN1_F0R1/2~CAN1_F13R1/2 分别设置滤波器 ID。如果 CAN 接收大于 28 个设备数据时, 必须将 CAN1_FOMODE~CAN1_F13MODE 设置为 0 (标识符掩码(屏蔽位)模式)。比如设置 ID 范围是 0x00000000~0x000000FF, 可以按上面例子设置。一般 CAN1_FxR1 设置为最大 ID, 最小 ID 就是 CAN1_FxR1&CAN1_FxR2

(8) 在 STM32H7XX/STM32MP15X 模块中 CAN 过滤器组配置: 以 CAN1 过滤器组 0 举例

```

#define CAN1_FILTER_CONFIG CAN_FILTER_TO_RXFIF00  // CAN 滤波器关联到 FIF00

#define CAN1_RXBUF_ID      0          // 接收缓冲区索引, 目前固定设置为 0

#define CAN1_ISCALIB_MSG_ID 0          // 指定是否为配置滤波器校准信息: 设置为 0 就是普通消息
                                     //过滤器组 0 设置

#define CAN1_FOACT_EN      1          // CAN1 过滤器 0 组配置: 1, 使能; 0, 关闭

#if (CAN1_FOACT_EN > 0)

#define CAN1_FOMODE        0          // CAN 过滤器模式: 0: 范围滤波模式; 1: 双固定 ID 滤波模式
                                     // 2: 经典过滤器: 过滤器 ID1=过滤器, 过滤器 ID2=掩码; 3: 范围滤波模式, EIDM 掩码未应用

```

```
#define CAN1_F0IDE      CAN_EXT_ID      // 帧类型: 0, 标准帧:CAN_STD_ID; 1, 扩展帧:CAN_EXT_ID;

// 要根据 CAN1_F0MODE 模式设置以下 ID
```

```
#define CAN1_F0ID1      0x00000001
```

```
#define CAN1_F0ID2      0x0000000F
```

```
#endif
```

举例说明:

A1. CAN1_F0MODE: 0, CAN1_F0ID1: 0x0000001, CAN1_F0ID2: 0x00000FF

这个配置可以接收 ID 是 0x0000001 到 0x00000FF 的扩展帧数据

A2. CAN1_F0MODE: 1, CAN1_F0ID1: 0x0000001, CAN1_F0ID2: 0x00000FF

这个配置可以接收 ID 是 0x0000001 和 0x00000FF 的扩展帧数据

A3. CAN1_F0MODE: 2, CAN1_F0ID1: 0x0000100, CAN1_F0ID2: 0xFFFFF00

这个配置可以接收 ID 是 0x0000100 到 0x00001FF 所有扩展帧数据, 只要满足 CAN1_F0ID1=(接收数据 ID&CAN1_F0ID2) 都会被接收

A4. CAN1_F0MODE: 3, 暂时未作验证, 可以不用

(9) 在../libapp/can_app.c 中查看 CAN_RxProc 处理函数:

```
void CAN_RxProc(void)
{
    INT32U tick;

    //----CAN1 接收处理-----

    #if (CAN1_EN > 0)                // 条件编译 CAN1 使能

    #if (CAN1_RXMODE == CAN_RXMODE_SCAN)    // 条件编译扫描接收数据模式

    tick = APP_GetSubTick(LibAppCAN[CAN1_ID].Scan_t); // 读取和上次采集的间隔时间, 单位 ms
    if (tick >= CAN1_SCAN_T)                // 计算和上次采集时间间隔大于等于设置值
    {
        LibAppCAN[CAN1_ID].Scan_t += tick;    // 记录本次扫描时间

        CAN_SCANRxProc(CAN1_ID);              // CAN1 扫描接收数据处理
    }

    #endif

    #if (CAN1_RXMODE == CAN_RXMODE_IRQ)    // 条件编译中断接收数据模式
```

```
CAN_IRQRxProc(CAN1_ID);           // CAN1 中断接收数据处理

#endif

#endif

... ..

}
```

这部分代码是 CAN1 接收数据处理，用户不需要修改该代码；接收到的数据通过 LibAppVars.Register.APP_InputDataCallback 回调函数发送出来。

(10) CAN1 接收数据被发送到../source/user_canapp.c 中的 CAN_DataProcess 函数

在这个函数中调用 CAN1_UserProcess(), 用户请在这个函数做相应的处理:

```
void CAN1_UserProcess(CAN_RX_MSG *pRxMsg, INT16U Num)
{
    // 用户在这里处理接收到的数据: pRxMsg, 接收消息指针, Num 消息个数

    INT16U i;

    for (i=0; i<Num; i++)           // 循环处理 Num 个 CAN 消息
    {
        #if (APP_CAN_DEBUG_EN > 0) // 条件编译使能接收数据打印输出, 打印数据输出到调试
                                    // 串口(注意: 按 16 进制数据打印)

        printf("AT+CAN1=RX[%d,%d]:", pRxMsg[i].ID, pRxMsg[i].DLC);

        if (pRxMsg[i].DLC > 0)
        {
            printf("%02x", pRxMsg[i].Data[0]);

            for (j=1; j<pRxMsg[i].DLC; j++)
            { printf(" %02x", pRxMsg[i].Data[j]);}

        }

        printf("\r\n");

        #endif

        // 将数据复制到 pVars->CAN1.RxMsg 中, 用户可以直接应用

        if (i < CAN1_RXBUF_SIZE)
        {

            pVars->CAN1.RxMsg[i].ID = pRxMsg[i].ID;
```

```
pVars->CAN1.RxMsg[i].IDE = pRxMsg[i].IDE;

pVars->CAN1.RxMsg[i].RTR = pRxMsg[i].RTR;

pVars->CAN1.RxMsg[i].DLC = pRxMsg[i].DLC;

pVars->CAN1.RxMsg[i].FMI = pRxMsg[i].FMI;

for (j=0; j<pRxMsg[i].DLC; j++)

{   pVars->CAN1.RxMsg[i].Data[j] = pRxMsg[i].Data[j];   }

}

// 用户在这里处理接收到的数据: pVars->CAN1.RxMsg[], 接收消息缓存
//

}

pVars->CAN1.Num = MIN(Num, CAN1_RXBUF_SIZE);

pVars->CAN1.AllNum += pVars->CAN1.Num;

pVars->CAN1.Flag |= AMKN_CAN_UPDATE_FLAG; // 设置 CAN1 收到新数据标志

#if (APP_CAN1_TEST_EN > 0) // 条件编译使能 CAN1 测试, 测试功能: 将数据在原样发送回去
#if (OS_EN == 0)

CAN_Write(CAN1_ID, (CAN_TX_MSG *)&pVars->CAN1.RxMsg[0], Num);

#else

CAN_UserSendData(CAN1_ID, (CAN_TX_MSG *)&pVars->CAN1.RxMsg[0], Num, AMKN_NOCOPY);

#endif

#endif

}

CAN 发送数据函数在../source/comfun.c 中的 CAN_UserSendData, 在有操作系统下用户可以调用这个函数
发送数据。
```

7. ADC(AI)输入编程说明:

(1) 关键词定义如下: ADC_EN、ADC_MODE、ADC_DOUBLE_BUFFER_EN、ADC_NOAVG_EN、ADC_READ_MODE、ADC_SCAN_T、AI1_EN~AIx_EN、ADC_CHNUM、ADC_AVGNUM、ADC_SAMPLE_TIME、ADC_FREQ、ADC_TIM14、ADC_TIM5、ADC_EXTSEL、AI1_RANGE~AIx_RANGE、ADC_DMA_EN 注意: x 范围是 1~10, 硬件模块不同, x 值也不同

以下说明以工控板 EMB8602 的 ADC 为例。

(2) 在 IO 配置文件 AMKNXXX_IOConfig.h 中定义: 参考如下:

```
// AI 端口定义

#define AI_NUM      8      // 定义 AI 输入端口数量

// 设置板子端口模拟量输入 AI1-AI8 与模块模拟量 AIN0-AIN15 对应关系

#define AI1          AIN0    // 设置 AI1<->AIN0 (PA0)
#define AI2          AIN3    // 设置 AI2<->AIN3 (PA3)
#define AI3          AIN6    // 设置 AI3<->AIN6 (PA6)
#define AI4          AIN8    // 设置 AI4<->AIN8 (PB0)
#define AI5          AIN9    // 设置 AI5<->AIN9 (PB1)
#define AI6          AIN10   // 设置 AI6<->AIN10 (PC0)
#define AI7          AIN12   // 设置 AI7<->AIN12 (PC2)
#define AI8          AIN13   // 设置 AI8<->AIN13 (PC3)
```

注意: 一般 AI1~AIx 这些设置是根据具体硬件而改变; 强烈建议按我司工控板的硬件设计来固定这些端口, 以保证软件的一致性。

(3) 在功能配置文件 AMKNXXX_Config.h 中定义, 参考如下:

```
#define ADC_EN      1  // ADC 使能, 1: 打开使能, 0: 关闭

#if (ADC_EN > 0)

#define ADC_MODE    0  // 0, ADC_MODE_SWSTART: 选择定时软件触发单时间点多次
                        // 采集模式; 1, ADC_MODE_EXTSEL: 选择外部触发等间隔采集模式

#define ADC_DOUBLE_BUFFER_EN 0 // 缓冲模式: 1, 使能双缓冲; 0, 选择单缓冲
                        // 使用双缓冲存储采集数据, 适合采集数据量比较大的情况, 给数据处理留有很大时间

#if (ADC_DOUBLE_BUFFER_EN > 0)

#define ADC_NOAVG_EN 0 // 是否做平均处理: 1, 内部不做数据平均处理, 原始采集数据输出; 0, 内
```

// 部做平均处理。注意：务必在双缓冲模式设置该标志，否则无效

#endif

// 设置读取 AD 转换输出值方式

#define ADC_READ_MODE // 可以选择：0：ADC_MODE_IRQ, 选择中断输出 AD 采样值；

// 1：ADC_MODE_SCAN, 选择定时扫描，用 ADC_Read() 函数读取采样值

#define ADC_SCAN_T 100 // 设置定时扫描时间间隔，单位：ms

// 模块模拟量输入使能定义

#define AI1_EN 1 // AI1 使能，1：打开使能，0：关闭

#define AI2_EN 1 // AI2 使能，1：打开使能，0：关闭

#define AI3_EN 1 // AI3 使能，1：打开使能，0：关闭

#define AI4_EN 1 // AI4 使能，1：打开使能，0：关闭

#define AI5_EN 1 // AI5 使能，1：打开使能，0：关闭

#define AI6_EN 1 // AI6 使能，1：打开使能，0：关闭

#define AI7_EN 1 // AI7 使能，1：打开使能，0：关闭

#define AI8_EN 1 // AI8 使能，1：打开使能，0：关闭

#define AI9_EN 0 // AI9 使能，1：打开使能，0：关闭

#define AI10_EN 0 // AI10 使能，1：打开使能，0：关闭

#define ADC_CHNUM (AI1_EN+AI2_EN+AI3_EN+AI4_EN+AI5_EN+AI6_EN+AI7_EN+AI8_EN+AI9_EN+AI10_EN)

// 采样通道数

#define ADC_AVGNUM 4 // 定义采样次数来计算平均值，范围 1~256，注意：此值太大会占用很大内存空间

#define ADC_SAMPLE_TIME ADC_SAMP21T0US // 采样间隔

#define ADC_FREQ 1 // 每秒钟输出采样结果次数(做完平均值后输出)

#if (ADC_MODE == ADC_MODE_SWSTART)

#define ADC_TIM5 ADC_TIM5MAIN_FLAG // 固定选择 ADC_TIM5MAIN_FLAG

#endif

#if (ADC_MODE == ADC_MODE_EXTSEL)

#define ADC_EXTSEL ADC_EXTSEL_T3TRG0 // 选择 AD 采样触发源，固定选择 ADC_EXTSEL_T3TRG0

#endif

// 板子端口输入量程：0，原始采样值(0~4095)；1，0~+10V；2，-10V~+10V；3，0~5V；

// 4，-5V~+5V；5，0~+20mA；6，-20mA~+20mA；

```
#define AI1_RANGE    1  // AI1 采样量程
#define AI2_RANGE    1  // AI2 采样量程
#define AI3_RANGE    1  // AI3 采样量程
#define AI4_RANGE    1  // AI4 采样量程
#define AI5_RANGE    5  // AI5 采样量程
#define AI6_RANGE    5  // AI6 采样量程
#define AI7_RANGE    5  // AI7 采样量程
#define AI8_RANGE    5  // AI8 采样量程
#define AI9_RANGE    0  // AI9 采样量程
#define AI10_RANGE   0  // AI10 采样量程

// 目前 ADC 采样固定用 DMA 方式存储数据

#define ADC_DMA_EN    1  // ADC DMA 使能, 1: 打开使能, 0: 关闭;

#endif
```

(4) 在../libapp/adc_app.c 中查看 ADC_Applnit() 初始化函数, 这函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

(5) 在../libapp/adc_app.c 中查看 ADC_ReadProc 处理函数:

```
void ADC_ReadProc(void)
{
    INT32U tick;

    #if (ADC_READ_MODE == ADC_MODE_SCAN)    // 条件编译定时扫描读取模式
        tick = APP_GetSubTick(LibAppADC[ADC1_ID].Scan_t); // 读取和上次采集的间隔时间, 单位 ms
        if (tick >= ADC_SCAN_T)            // 计算和上次采集时间间隔大于等于设置值
        {
            LibAppADC[ADC1_ID].Scan_t += tick;    // 记录本次扫描时间
            ADC_SCANRead(ADC1_ID);                // 扫描读取 ADC 采集值
        }

    #endif

    #if (ADC_READ_MODE == ADC_MODE_IRQ)        // 条件编译中断读取模式
```

```

ADC_IRQRead(ADC1_ID); // 中断读取 ADC 采集值

#endif

}

```

这部分代码是取数 ADC 采集读据，用户不需要修改该代码；数据通过 LibAppVars.Register.APP_InputDataCallback 回调函数发送出来。

(5) ADC 采集数据被发送到../source/user_adcapp.c 中的 ADC_DataProcess 函数, 用户请在这个函数做相应的处理:

```

void ADC_DataProcess(INT8U id, INT32S *p, INT16U len)
{
    INT16U i;
    for (i=0; i<MIN(AI_NUM, len); i++)
    {
        pVars->AI.buf[id][i] = p[i]; // 复制原始值
        pVars->AI.val[id][i] = ADC_Conv(AI_Range[i], p[i]); // 计算量程值
    }
    pVars->AI.cnt++;
    pVars->AI.Flag |= (AMKN_ADC1_UPDATE_FLAG<<id);

    #if (APP_ADC_DEBUG_EN > 0) // 条件编译使能接收数据打印输出
    if ((UserVars.ADCCnt%ADC_FREQ) == 0) //每秒打印一次采集结果
    {
        printf("AT+AI=%d", AI_NUM);
        #if (AI1_EN == 1) // 判断 AI1 使能
        printf(", %d", pVars->AI.val[0][AI1_ID]);
        #endif
        ... ..
        #if (AI16_EN == 1) // 判断 AI16 使能
        printf(", %d", pVars->AI.val[0][AI16_ID]);
        #endif
    }
}

```

```
    printf("\r\n");  
}  
  
#endif  
}
```

注意：p，ADC 采样值数据指针，内部数据按 p[A11_ID]~p[A16_ID] 顺序排列。len 是数据个数。

8. DAC(A0)输出编程说明:

(1) 关键词定义如下: DACx_EN、DACx_MODE、DACx_FREQ、DACx_OUTBUF_EN、DACx_SCAN_T、TIM6_DAC1_EN、TIM7_DAC2_EN、DACx_TXBUF_SIZE、DACx_DMA_EN、DMA1CH6_DAC1_EN、DMA1CH7_DAC2_EN

注意: x 范围是 1~2

以下说明以工控板 EMB8602 的 DAC 为例。

(2) 因为 DAC1 和 DAC2 的管脚是固定的, 所以只在 ../libraries/dac.h 文件中定义:

```
#define DAC1_PIN      PA4  // DAC1 管脚定义
#define DAC2_PIN      PA5  // DAC2 管脚定义
```

(3) 在功能配置文件 AMKNXXX_Config.h 中定义, 以 DAC1 为例参考如下:

```
#define DAC1_EN      1      // DAC1 使能, 1: 打开使能, 0: 关闭
#if (DAC1_EN>0)
#define DAC1_MODE    0      // DAC1 模式: 0(DAC_MODE_MTOUT), 手动输出;
                                // 1(DAC_MODE_ATOUT_N), 连续输出 1~N 个缓存中的数据后停止;
                                // 2(DAC_MODE_ATOUT), 持续输出缓存中的数据, 不停止;
#define DAC1_FREQ    1000   // DAC1 自动输出频率
#define DAC1_OUTBUF_EN 0     // DAC1 输出缓存使能: 1, 使能; 0, 关闭;
#define DAC1_SCAN_T  3000   // 设置定时扫描时间间隔, 单位: ms
#if (DAC1_MODE>0)
#define TIM6_DAC1_EN 1      // 设置定时器 6 用于 DAC1 功能, 这个设置不可更改
#define DAC1_TXBUF_SIZE 256 // DAC1 发送数据缓存长度
#define DAC1_DMA_EN   1     // DAC1 DMA 使能, 1: 打开使能, 0: 关闭;
#endif
#endif
#endif
```

(4) 在 ../libapp/dac_app.c 中查看 DAC_AppInit() 初始化函数, 这函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

(5) 在 ../libapp/dac_app.c 中查看 DAC_AppTest 函数, 这个函数是操作 DAC 输出的测试例程; 在 ../source/user_testapp.c 的 User_AppTest 函数中被间隔 1 秒调用。以 DAC1 为例说明如下:

```
void DAC_AppTest(void)
```

```
{  
  
    #if (DAC1_EN > 0)  
  
        #if (DAC1_MODE == DAC_MODE_MTOUT)           // DAC1 手动输出  
  
            DAC_AppMTOutTest(DAC1_ID);  
  
        #endif  
  
  
        #if (DAC1_MODE == DAC_MODE_ATOUT_N)           // 模式 1, 连续输出 N 个波形后停止  
  
            DAC_AppATOutNTest(DAC1_ID, SINE_WAVE);           // 输出正弦波测试  
  
            //DAC_AppATOutNTest(DAC1_ID, TRI_WAVE);           // 输出三角波测试  
  
            //DAC_AppATOutNTest(DAC1_ID, SAW_WARE);           // 输出锯齿三角波测试  
  
            //DAC_AppATOutNTest(DAC1_ID, SQUARE_WAVE);       // 输出方波三角波测试  
  
        #endif  
  
  
        #if (DAC1_MODE == DAC_MODE_ATOUT)           // 模式 2, 持续输出不停止  
  
            DAC_AppATOutTest(DAC1_ID, SINE_WAVE);           // 输出正弦波测试  
  
            //DAC_AppATOutTest(DAC1_ID, TRI_WAVE);           // 输出三角波测试  
  
            //DAC_AppATOutTest(DAC1_ID, SAW_WARE);           // 输出锯齿三角波测试  
  
            //DAC_AppATOutTest(DAC1_ID, SQUARE_WAVE);       // 输出方波三角波测试  
  
        #endif  
  
  
        #if (DAC1_MODE == DAC_MODE_NOISE)           // 模式 3, 生成噪声波模式;  
  
            DAC_AppNoiseOutTest(DAC1_ID);           // 输出噪声测试  
  
        #endif  
  
  
        #if (DAC1_MODE == DAC_MODE_TRIANGLE)           // 模式 3, 生成三角波模式;  
  
            DAC_AppTriangleOutTest(DAC1_ID);           // 输出三角波测试  
  
        #endif  
  
    #endif  
  
}
```

用户可查看 DAC_AppMTOutTest、DAC_AppATOutNTest、DAC_AppATOutTest 函数具体实现

9. FCLK 脉冲输入编程说明:

(1) 关键词定义如下: FCLKx_EN、FCLKx_MODE、FCLKx_SCAN_T、FCLKx_IRQREAD_EN、FCLKx_TIM、
FCLKxCH1_EN~FCLKxCH4_EN、FCLKxCH_EN、FCLKx_MINFREQ、FCLKxCH1_PIN~FCLKxCH4_PIN、
FCLKxCH1_PCS~FCLKxCH4_PCS、FCLKxCH1_BUF_SIZE~FCLKxCH4_BUF_SIZE、
FCLKxCH1_DMA_EN~FCLKxCH4_DMA_EN 注意: x 范围是 1~4

以下说明以工控板 EMB8602 的 FCLK1 为例。

(2) 在 IO 配置文件 AMKNXXX_IOConfig.h 中定义: 参考如下:

```
#define FCLK_NUM          1          // FCLK 数量定义

// FCLK1 (TIM4) 管脚功能重映射设置:

#define FCLK1_REMAP      TIM_REMAP_1  // FCLK1 (TIM4) 管脚重映射

#define FCLK1_CH1        PD12          // FCLK1 (TIM4) CH1 管脚定义
#define FCLK1_CH2        PD13          // FCLK1 (TIM4) CH2 管脚定义
#define FCLK1_CH3        PD14          // FCLK1 (TIM4) CH3 管脚定义
#define FCLK1_CH4        PD15          // FCLK1 (TIM4) CH4 管脚定义
#define FCLK1_ETR        IO_NONE      // FCLK1 (TIM4) ETR 管脚定义
```

(3) 在功能配置文件 AMKNXXX_Config.h 中定义, 参考如下:

```
#define FCLK1_EN          1 // FCLK1 使能, 1: 打开使能, 0: 关闭

#if (FCLK1_EN > 0)

#define FCLK1_MODE        2 // 模式选择:

    // 0(FCLK_MODE_COUNT), 计数模式(1 路, CH1 输入有效);
    // 1(FCLK_MODE_DECODE), 正交编码器计数(CH1 接 A, CH2 接 B);
    // 2(FCLK_MODE_FREQ), 测频模式(4 路, CH1, CH2, CH3, CH4 输入都有效);
    // 3(FCLK_MODE_PWMRATE), 测 PWM 占空比模式(1 路, CH1 输入有效);

#define FCLK1_SCAN_T      1000 // 设置定时扫描时间间隔, 单位: ms
#define FCLK1_ATOUT_T     3000 // 设置 AT 指令输出间隔, 单位: ms

#if ((FCLK1_MODE == 2) || (FCLK1_MODE == 3))

#define FCLK1_IRQREAD_EN  1 // 中断读取模式使能: 0, 采用 FCLK_Read() 函数读取; 1, 采用中断
    // 函数 FCLK_IRQHandler() 输出结果, 注意: 只有在测频模式和测 PWM 占空比模式有效
```

```
#endif

#define FCLK1_TIM    TIM4_ID    // 选择定时器, 这个设置不可更改

#define TIM4_FCLK_EN  1 // 设置定时器 4 用于 FCLK 功能, 这个设置不可更改

// 在计数模式和测 PWM 占空比模式下 FCLK1CH1_EN 必须使能; 在正交编码器计数模式下 FCLK1CH1_EN,
// FCLK1CH2_EN 必须使能; 在测频模式根据实际硬件设计使能各个通道

#define FCLK1CH1_EN    1          // FCLK1: 1, 使能; 0, 关闭
#define FCLK1CH2_EN    1          // FCLK2: 1, 使能; 0, 关闭
#define FCLK1CH3_EN    0          // FCLK3: 1, 使能; 0, 关闭
#define FCLK1CH4_EN    0          // FCLK4: 1, 使能; 0, 关闭

// FCLK1 所有通道使能: BIT0:CH1;BIT1:CH2;BIT2:CH3;BIT3:CH4;

#define FCLK1CH_EN    (FCLK1CH1_EN|(FCLK1CH2_EN<<1)|(FCLK1CH3_EN<<2)|(FCLK1CH4_EN<<3))

#define FCLK1_MINFREQ  100    // 模式 2, 3 中, 测量最小频率设定, 单位 hz

#define FCLK1CH1_PIN    0 // FCLK1 管脚输入信号触发边沿: 0, 上升沿; 1, 下降沿
#define FCLK1CH2_PIN    0 // FCLK2 管脚输入信号触发边沿: 0, 上升沿; 1, 下降沿
#define FCLK1CH3_PIN    0 // FCLK3 管脚输入信号触发边沿: 0, 上升沿; 1, 下降沿
#define FCLK1CH4_PIN    0 // FCLK4 管脚输入信号触发边沿: 0, 上升沿; 1, 下降沿

// CH1~4 管脚输入信号预分频系数: 0, 不分频; 1, 2 分频; 2, 4 分频; 3, 8 分频;

#define FCLK1CH1_PCS    0
#define FCLK1CH2_PCS    0
#define FCLK1CH3_PCS    0
#define FCLK1CH4_PCS    0

#if ((FCLK1_MODE == 2) || (FCLK1_MODE == 3))

#if (FCLK1CH1_EN > 0)

#define FCLK1CH1_BUF_SIZE  16          // FCLK1CH1 缓存长度, 范围 1~64

#endif

#if (FCLK1CH2_EN > 0)

#define FCLK1CH2_BUF_SIZE  16          // FCLK1CH2 缓存长度, 范围 1~64

#endif

#if (FCLK1CH3_EN > 0)

#define FCLK1CH3_BUF_SIZE  16          // FCLK1CH3 缓存长度, 范围 1~64
```

```
#endif

#if (FCLK1CH4_EN > 0)

#define FCLK1CH4_BUF_SIZE 16          // FCLK1CH4 缓存长度, 范围 1~64

#endif

#endif

// FCLK1 各个通道 DMA 使能, 只有测频模式和测 PWM 占空比模式才支持 DMA 功能
// 注意以下 FCLK 各通道同时只允许一个通道 DMA 使能, 使用 TIM4 定时器

#if (FCLK1_MODE > 1)

#define FCLK1CH1_DMA_EN 0           // CH1 DMA: 1, 使能; 0, 关闭;
#define FCLK1CH2_DMA_EN 0           // CH2 DMA: 1, 使能; 0, 关闭;
// #define FCLK1CH3_DMA_EN 0         // CH3 DMA: 1, 使能; 0, 关闭;

// 注意: FCLK1CH4 不支持 DMA 功能

#endif

#endif
```

(4) 在../libapp/fclk_app.c 中查看 FCLK_AppInit() 初始化函数, 这函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

(5) 在../libapp/fclk_app.c 中查看 FCLK_AppProc 处理函数:

注: 本函数由../source/user_testapp.c 中 User_AppTest 函数间隔 100ms 调用 1 次

```
void FCLK_AppProc(void)
{
    INT32U tick;

    //----FCLK1-----

    #if (FCLK1_EN > 0)          // 条件编译 FCLK1 使能

        tick = APP_GetSubTick(LibAppFCLK[FCLK1_ID].Scan_t); // 读取和上次采集的间隔时间, 单位 ms

        if (tick >= FCLK1_SCAN_T)          // 计算和上次采集时间间隔大于等于设置值
        {

            LibAppFCLK[FCLK1_ID].Scan_t += tick; // 记录本次扫描时间

            #if (FCLK1_MODE == FCLK_MODE_COUNT) // 计数模式(1 路, CH1 输入有效);
```

```

    FCLK_ReadCount(FCLK1_ID);          // 读取计数值

    #endif

    #if (FCLK1_MODE == FCLK_MODE_DECODE) //正交编码器计数(CH1 接 A, CH2 接 B);
    FCLK_ReadDeCode(FCLK1_ID);          // 读取正交编码器计数值
    #endif

    #if (FCLK1_MODE == FCLK_MODE_FREQ)   // 测频模式(4 路, CH1, CH2, CH3, CH4 输入都有效);
    FCLK_ReadFreq(FCLK1_ID, FCLK1CH_EN); // 读取测量频率
    #endif

    #if (FCLK1_MODE == FCLK_MODE_PWMRATE) // 测 PWM 占空比模式(1 路, CH1 输入有效)
    FCLK_ReadPWMRate(FCLK1_ID);          // 读取 PWM 输入占空比
    #endif
}

#endif

//-----FCLK2-----

... ..
}

```

这部分代码是读取 FCLK 输入读据，用户不需要修改该代码；数据通过 LibAppVars.Register.APP_InputDataCallback 回调函数发送出来。

(6) 读取 FCLK 输入读据被发送到../source/user_fclkapp.c 中的 FCLK_DataProcess 函数, 根据 ID 不同, 用户请在这个函数做相应的处理:

```

void FCLK1_UserProcess(LIBAPP_FCLK_VARS *pData, INT32U len)
{
    INT16U i;

    pVars->FCLK1.Flag = pData->OKFlag;

    #if (FCLK1_MODE == FCLK_MODE_COUNT) // 计数模式(1 路, CH1 输入有效);
    if (pData->Mode == FCLK_MODE_COUNT)
    {
        pVars->FCLK1.Flag = pData->OKFlag;

        if (pData->OKFlag & FCLK_CH1OK_FLAG) // 测量正确
        { // 用户自行在这里处理数据: pData->cnt 是计数值, 记录数据

```

```
pVars->FCLK1.cnt = pData->cnt;

pVars->FCLK1.Flag |= AMKN_FCLK_UPDATE_FLAG; // 设置 FCLK1 收到新数据标志

#if (APP_FCLK_DEBUG_EN == 1)

printf("AT+FCLK1=N,%d\r\n",  pData->cnt);

#endif

}

else if (pData->OKFlag&FCLK_CH1ERR_FLAG) // 测量错误

{

    pVars->FCLK1.Flag |= AMKN_FCLK_UPDATE_FLAG; // 设置 FCLK1 收到新数据标志

    #if (APP_FCLK_DEBUG_EN == 1)

    printf("AT+FCLK1=N, ERROR\r\n");

    #endif

}

}

#endif

#if (FCLK1_MODE == FCLK_MODE_DECODE) // 正交编码器计数(CH1 接 A, CH2 接 B);

if (pData->Mode == FCLK_MODE_DECODE)

{

    pVars->FCLK1.Flag = pData->OKFlag;

    if (pData->OKFlag&FCLK_CH1OK_FLAG) // 测量正确

    { // 用户自行在这里处理数据

        // pData->QEICnt 正交编码模式：本次采集计数增量值

        // pData->QEICount 正交编码模式：计算得到的正交编码器总计数值

        // 记录数据

        pVars->FCLK1.QEICnt = pData->QEICnt;

        pVars->FCLK1.QEICount = pData->QEICount;

        pVars->FCLK1.Flag |= AMKN_FCLK_UPDATE_FLAG; // 设置 FCLK1 收到新数据标志

        #if (APP_FCLK_DEBUG_EN == 1)

        printf("AT+FCLK1=E,%d,%d\r\n",  pData->QEICnt,  pData->QEICount);

        #endif

    }

}
```

```
}

else if (pData->OKFlag&FCLK_CH1ERR_FLAG) // 测量错误

{

    pVars->FCLK1.Flag |= AMKN_FCLK_UPDATE_FLAG; // 设置 FCLK1 收到新数据标志

    #if (APP_FCLK_DEBUG_EN == 1)

        printf("AT+FCLK1=E, ERROR\r\n");

    #endif

}

}

#endif

#if (FCLK1_MODE == FCLK_MODE_FREQ) // 测频模式(4 路, CH1, CH2, CH3, CH4 输入都有效);

    if (pData->Mode == FCLK_MODE_FREQ)

    { // 用户自行在这里处理数据; len: 测量频率结果的个数; pData->Freq[][] 频率值, 单位 0.01HZ

        pVars->FCLK1.Flag = pData->OKFlag;

        pVars->FCLK1.Num = MIN(AMKN_FCLK_MAX_FREQ_NUM, len);

        #if (FCLK1CH1_EN > 0)

            if (pData->OKFlag&FCLK_CH1OK_FLAG) // CH1 测量正确

            { // 记录数据

                for (i=0; i<pVars->FCLK1.Num; i++)

                { pVars->FCLK1.Freq[FCLK_CH1][i] = pData->Freq[FCLK_CH1][i];

                }

                pVars->FCLK1.Flag |= AMKN_FCLK_UPDATE_FLAG; // 设置 FCLK1 收到新数据标志

                #if (APP_FCLK_DEBUG_EN == 1)

                    printf("AT+FCLK1=F1"); // 打印输出频率值

                    for (i=0; i<len; i++)

                    {

                        printf(", %d. %02d", pData->Freq[FCLK_CH1][i]/100, pData->Freq[FCLK_CH1][i]%100);

                    }

                    printf("\r\n");

                }

            }

        }

    }

#endif

#endif
```

```
#endif

}

else if (pData->OKFlag&FCLK_CH1ERR_FLAG) //CH1 测量错误
{
    pVars->FCLK1.Flag |= AMKN_FCLK_UPDATE_FLAG; // 设置 FCLK1 收到新数据标志

    #if (APP_FCLK_DEBUG_EN == 1)
        printf("AT+FCLK1=F1, ERROR\r\n");
    #endif
}

#endif

#if (FCLK1CH2_EN > 0)
    ... ..
#endif

#if (FCLK1CH3_EN > 0)
    ... ..
#endif

#if (FCLK1CH4_EN > 0)
    ... ..
#endif

}

}

#endif

#if (FCLK1_MODE == FCLK_MODE_PWMRATE) // 测 PWM 占空比模式(1 路, CH1 输入有效)
    if (pData->Mode == FCLK_MODE_PWMRATE)
    {
        pVars->FCLK1.Flag = pData->OKFlag;

        if (pData->OKFlag&FCLK_CH1OK_FLAG) // 测量正确
        {
            // 用户自行在这里处理数据: len: 测量占空比结果的个数

            // pData->Rate[] 占空比结果, 单位 0.01; 记录数据:

            pVars->FCLK1.Num = MIN(USER_FCLK_MAX_PWMRATE_NUM, len);
        }
    }
}
```

```
for (i=0; i<pVars->FCLK1.Num; i++)
{
    pVars->FCLK1.Rate[i] = pData->Rate[i];
}

pVars->FCLK1.Flag |= AMKN_FCLK_UPDATE_FLAG; // 设置 FCLK1 收到新数据标志
#if (APP_FCLK_DEBUG_EN == 1)
printf("AT+FCLK1=D"); // 打印输出占空比值
for (i=0; i<len; i++)
{ printf(",%d.%02d", pData->Rate[i]/100, pData->Rate[i]%100);
}

printf("\r\n");
#endif
}

else if (pData->OKFlag&FCLK_CH1ERR_FLAG) // 测量错误
{
    pVars->FCLK1.Flag |= AMKN_FCLK_UPDATE_FLAG; // 设置 FCLK1 收到新数据标志
    #if (APP_FCLK_DEBUG_EN == 1)
    printf("AT+FCLK1=D, ERROR\r\n");
    #endif
}

}

#endif
}
```

10. PWM 脉冲输出编程说明:

(1) 关键词定义如下: PWMx_EN、PWMx_MODE、PWMx_FREQ、PWMx_TIM、TIMx_PWM_EN、PWMxCH1_EN~PWMxCH4_EN、PWMxCH_EN、PWMxCH1_RATE~PWMxCH4_RATE、PWMxCH1_PIN~PWMxCH4_PIN、PWMx_DMA_EN、PWM_NUM、PWMx_REMAP、PWMx_CH1~PWMx_CH4、PWMx_ETR 注意: x 范围是 1~8

以下说明以工控板 EMB8602 的 PWM1 为例。

(2) 在 IO 配置文件 AMKNXXXX_IOConfig.h 中定义: 参考如下:

```
// PWM 端口定义

#define PWM_NUM      1          // PWM 数量定义

#define PWM1_REMAP    TIM_REMAP_3 // PWM1 (TIM3) 管脚重映射

#define PWM1_CH1      PC6        // PWM1 (TIM3) CH1 管脚定义

#define PWM1_CH2      PC7        // PWM1 (TIM3) CH2 管脚定义

#define PWM1_CH3      PC8        // PWM1 (TIM3) CH3 管脚定义

#define PWM1_CH4      PC9        // PWM1 (TIM3) CH4 管脚定义

#define PWM1_ETR      IO_NONE    // PWM1 (TIM3) ETR 管脚定义
```

(3) 在功能配置文件 AMKNXXXX_Config.h 中定义, 参考如下:

```
#define PWM1_EN      1          // PWM1 使能, 1: 打开使能, 0: 关闭

#if (PWM1_EN > 0)

// 设置 PWM 输出模式设置

#define PWM1_MODE     PWM_FREQ // 可以选择: 0 (PWM_FREQ): 连续脉冲频率输出, 持续输出
// 1 (PWM_FREQ_N): 多个脉冲频率输出, 输出完设定的脉冲数后停止
// 2 (PWM_RATE): 固定频率占空比可调连续脉冲输出: 频率固定, 占空比 0%-100%可调, 持续输出
// 3 (PWM_WRITE): 新增控制模式, 利用 PWM_Write() 函数实现输出控制模式, 具体控制方式由函数参数决定, 如果使能了 DMA 就利用 DMA 方式控制 PWM, 节约 MCU 资源

#define PWM1_FREQ      1000      // 初始频率

#define PWM1_TIM        TIM3_ID   // 选择定时器, 这个设置不可更改

#define TIM3_PWM_EN     1         // 设置定时器 3 用于 PWM 功能, 这个设置不可更改通道使能

#define PWM1CH1_EN      1         // PWM1CH1: 1, 使能; 0, 关闭

#define PWM1CH2_EN      1         // PWM1CH2: 1, 使能; 0, 关闭
```

```

#define PWM1CH3_EN      0          // PWM1CH3: 1, 使能; 0, 关闭
#define PWM1CH4_EN      0          // PWM1CH4: 1, 使能; 0, 关闭

//PWM1 所有通道使能: BIT0:CH1;BIT1:CH2;BIT2:CH3;BIT3:CH4;

#define PWM1CH_EN      (PWM1CH1_EN|(PWM1CH2_EN<<1)|(PWM1CH3_EN<<2)|(PWM1CH4_EN<<3))

#define PWM1CH1_RATE    500 // PWM1CH1 初始占空比 50%(0(0.0%)~1000(100.0%))
#define PWM1CH2_RATE    500 // PWM1CH2 初始占空比 50%(0(0.0%)~1000(100.0%))
#define PWM1CH3_RATE    500 // PWM1CH3 初始占空比 50%(0(0.0%)~1000(100.0%))
#define PWM1CH4_RATE    500 // PWM1CH4 初始占空比 50%(0(0.0%)~1000(100.0%))

#define PWM1CH1_PIN 0 // PWM1CH1 停止模式输出管脚电平: 0, 低电平; 1, 高电平
#define PWM1CH2_PIN 0 // PWM1CH2 停止模式输出管脚电平: 0, 低电平; 1, 高电平
#define PWM1CH3_PIN 0 // PWM1CH3 停止模式输出管脚电平: 0, 低电平; 1, 高电平
#define PWM1CH4_PIN 0 // PWM1CH4 停止模式输出管脚电平: 0, 低电平; 1, 高电平

#if (PWM1_MODE == PWM_WRITE)

#define PWM1_DMA_EN      1 // PWM1 DMA 使能, 1: 打开使能, 0: 关闭;

#endif

// 互补通道使能

#if ((PWM1_TIM == TIM1_ID) || (PWM1_TIM == TIM8_ID))

#define PWM1CH1N_EN      0          // PWM1CH1N: 1, 使能; 0, 关闭
#define PWM1CH2N_EN      0          // PWM1CH2N: 1, 使能; 0, 关闭
#define PWM1CH3N_EN      0          // PWM1CH3N: 1, 使能; 0, 关闭

#define PWM1CH1N_PIN 0 // PWM1CH1N 停止模式管脚输出电平(无效驱动电平): 0, 低电平; 1, 高电平
#define PWM1CH2N_PIN 0 // PWM1CH2N 停止模式管脚输出电平(无效驱动电平): 0, 低电平; 1, 高电平
#define PWM1CH3N_PIN 0 // PWM1CH3N 停止模式管脚输出电平(无效驱动电平): 0, 低电平; 1, 高电平

#define PWM1_DTG          1000      // 互补模式死区时间设置, 单位: ns

#define PWM1_BKIN_EN      0          // 刹车输入使能: 0, 关闭; 1, 使能

#endif

#endif

```

(4) 在../libapp/pwm_app.c 中 PWM_AppInit 完成初始化;PWM_Start/PWM_Stop/PWM_ModifyFreq/PWM_ModifyRate/PWM_AppWrite 完成控制操作, 请自行查看程序源码。

(5) 查看../source/user_pwmapp.c 中 PWM_AppTest 函数，这个函数是操作 PWM 输出的测试例程，被在../source/user_testapp.c 中的 User_MainApp 函数间隔 100ms 调用。以 PWM1 为例说明如下：

```
void PWM_AppTest(void)
{
    INT32U tick;

    INT32U PWM_IDFlag = 0;

    /**PWM1 输出控制测试*****/
    #if (PWM1_EN > 0)                // 条件编译 PWM1 使能
        #if (PWM1_MODE == PWM_FREQ) // 条件编译连续脉冲输出
            PWM_UserFreqProcess(PWM1_ID);
        #endif

        #if (PWM1_MODE == PWM_FREQ_N) // 多个脉冲频率输出，输出完设定的脉冲数后停止
            PWM_UserFreqNProcess(PWM1_ID);
        #endif

        #if (PWM1_MODE == PWM_RATE) // 固定频率占空比可调连续脉冲输出：输出 PWM，频率固
            // 定，占空比 0%~100%可调，持续输出
            PWM_UserRateProcess(PWM1_ID);
        #endif

        #if (PWM1_MODE==PWM_WRITE) // DMA 输出控制模式
            PWM_UserWriteProcess(PWM1_ID);
        #endif
    #endif

    //PWM2 输出控制测试
    #if (PWM2_EN > 0) // PWM2 使能
        ... ..
    #endif
    ... ..
}
```

用户可以自行在 `../source/user_pwmapp.c` 查看 `PWM_UserFreqProcess`、`PWM_UserFreqNProcess`、`PWM_UserRateProcess`、`PWM_UserWriteProcess` 函数源码。

11. TIM 定时器编程说明：

(1) 关键词定义如下：TIMx_EN、TIMx_MODE、TIMx_T、TIMx_PSC

x 范围：1~24；注意：定时器是芯片内部功能，无管脚定义。

(2) 在功能配置文件 AMKNXXXX_Config.h 中定义，以 TIM1 为例参考如下：

```
#define TIM1_EN    1 // TIM1 使能, 1: 打开使能, 0: 关闭

#define TIM1_MODE  0 // TIM1 工作模式: 0, TIM_WKMODE_INT, 定时器工作在定时中断模式, 定
//时时间由参数 TIM1_T 设置; 1, TIM_WKMODE_COUNT, 定时器工作在定时计数模式, 外部通过调
//用 Timer_Ctrl 函数参数 CMD_TIM_ENA/CMD_TIM_DIS 启动/停止定时器, 再通过参数
//CMD_TIM_READ 读取计数值

// 初始定时时间设定

#define TIM1_T      1000000          // TIM1 定时器定时时间, 单位 us

#define TIM1_PSC    (SYSCLK/1000000) // 分频系数, 当工作模式设置为 TIM_WKMODE_COUNT,
// 请设置这个; 默认计数单位是 1us

#if ((TIM1_PWM_EN + TIM1_FCLK_EN + TIM1_EN)>1)

#error "ERROR: TIM1 不能同时用于 PWM, FCLK 和 TIM1 定时功能!"

#endif
```

(3) 在 ../libapp/tim_app.c 中查看 TIM_AppInit 初始化函数, 这函数根据上面配置初始化 TIM, 用户一般不需要修改该函数, 具体代码用户自行查看。

(4) 在功能配置文件 AMKNXXXX_Config.h 中做如下定义：

```
#define TIM1_EN    1 // TIM1 使能, 1: 打开使能, 0: 关闭

#define TIM1_MODE  0 // TIM1 工作模式: 定时中断模式

#define TIM1_T      1000000 // TIM1 定时器定时时间, 单位 us
```

则间隔 1 秒会产生一次中断, 中断函数在 ../source/ISRHooks.c 中的 TIM1_ISRHook() 函数: 在这个函数中用户自行编写中断处理程序, 但要尽可能短, 缩短执行时间。

(5) 如果将 TIM1_MODE 定义为 1 则定时器工作在定时计数模式, 请查看 ../libapp/tim_app.c 的 TIM_AppTest_ReadCount 函数:

```
INT32U cnt_buf[24], cnt;
```

```
// 各个定时间第一次读取计数值

    #if ((TIM1_EN == 1)&&(TIM1_MODE == TIM_WKMODE_COUNT))

        cnt_buf[TIM1_ID] = Timer_Ctrl(TIM1_ID, CMD_TIM_READ, 0);

    #endif

    // 延时 t ms

    Delay_ms(t);

    // 各个定时间第二次读取计数值

    #if ((TIM1_EN == 1)&&(TIM1_MODE == TIM_WKMODE_COUNT))

        cnt = Timer_Ctrl(TIM1_ID, CMD_TIM_READ, 0); // 读取当前计数值

        cnt = cnt - cnt_buf[TIM1_ID];                // 计算与上次测量差值

        #if (APP_TIM_DEBUG_EN == 1)

            printf("AT+TIM1=%d\r\n", cnt);           // 打印输出计数值

        #endif

    #endif
```

这个函数执行结果是测量 Delay_ms(20) 这个函数的准确执行时间, cnt 就是单位为 1us 的计数值, 就是准确执行时间。用户可以参考这个实现测量任何操作的准确执行时间。

12. EXTI 外部中断编程说明:

(1) 关键词定义如下: EXTIx_EN、EXTIx_IO、EXTIx_MODE、EXTI16_PVD_EN、EXTI16_PVD_MODE、EXTI17_RTCAAlarm_EN、EXTI17_RTCAAlarm_MODE、EXTI18_USBWakeUp_EN、EXTI18_USBWakeUp_MODE、EXTI19_NETWakeUp_EN、EXTI19_NETWakeUp_MODE、EXTI20_USBHSWakeUp_EN、EXTI20_USBHSWakeUp_MODE、EXTI21_RTCTSE_EN、EXTI21_RTCTSE_MODE、EXTI22_RTCWakeUp_EN、EXTI22_RTCWakeUp_MODE

x 范围: 0~15;

(2) 在功能配置文件 AMKNXXX_Config.h 中定义, 以 EXTI0 为例参考如下:

// EXTI0 中断配置

```
#define EXTI0_EN    1  // 中断或事件使能: 0, 关闭中断和事件; 1, 打开中断并在初始化中使能; 2, 打  
开事件请求并在初始化中使能; 3, 打开中断并在初始化中不使能; 4, 打开事件请求并在初始化中不使能;  
  
#define EXTI0_IO    PB0  // 在 PA0, PB0, PC0, PD0, PE0, PF0, PG0, PH0, PIO 中选  
                          // 择一个 IO 作为中断输入口; 这个要根据硬件设置来配置。  
  
#define EXTI0_MODE    0  // 触发中断和事件模式: 0, 上升沿触发中断和事件; 1, 下降沿触发中断  
                          // 和事件; 2, 上升沿下降沿都触发中断和事件;
```

以上配置实现: PB0 端口当出现上升沿脉冲时产生中断。

(3) 在../libapp/exti_app.c 中查看 EXTI_ApplInit 初始化函数, 这函数根据上面配置初始化 EXTI, 用户一般不需要修改该函数, 具体代码用户自行查看。

(4) 中断函数在../source/ISRHooks.c 中的 EXTI0_ISRHook() 函数: 在这个函数中用户自行编写中断处理程序, 但要尽可能短, 缩短执行时间。

13. SPI 总线编程说明：

(1) 关键词定义如下：SPIx_EN、SPIx_CKMODE、SPIx_DIVCLK、SPIxTX_DMA_EN、SPIxRX_DMA_EN、SPIx_NSS、SPIx_SCK、SPIx_MISO、SPIx_MOSI

x 范围：0~5；以下说明以 SPI1 为例

(2) 在 IO 配置文件 AMKNXXXX_IOConfig.h 中定义：参考如下：

// SPI1 引脚定义：

```
#define SPI1_REMAP SPI_REMAP_1 // SPI1 重映射 1

#define SPI1_NSS PA15 // SPI1_NSS 管脚，这个管脚就是板载 SPI FLASH(25QXX)片选

#define SPI1_SCK PB3 // SPI1_SCK 管脚

#define SPI1_MISO PB4 // SPI1_MOSI 管脚

#define SPI1_MOSI PB5 // SPI1_MISO 管脚
```

(3) 在功能配置文件 AMKNXXXX_Config.h 中定义，参考如下：

```
#define SPI1_EN 1 // SPI 使能， 1：打开使能， 0：关闭

#if (SPI1_EN > 0)

#define SPI1_CKMODE SPI_CKMODE3 // 时钟相位模式，参见 spi.h 中说明

#define SPI1_DIVCLK SPI_DIVCLK_16 // SPI 时钟分频系数

//SPI DMA 配置

#define SPI1TX_DMA_EN 0 // 设置发送 DMA 使能，1：打开使能， 0：关闭；

#define SPI1RX_DMA_EN 0 // 设置接收 DMA 使能，1：打开使能， 0：关闭；

#endif
```

(4) 在 ../libapp/spi_app.c 中查看 SPI_ApplInit 初始化函数，这函数根据上面配置初始化 SPI，用户一般不需要修改该函数，具体代码用户自行查看。

(5) 因为 SPI 总线的应用一般是外挂 SPI 总线器件，这里只举外挂 74HC595 芯片操作的例子：在 ../libapp/io_app.c 中的 D0_SPIOutput() 函数：在这个函数中用户通过 SPI1 总线发送长度为 len 的数据到 HC595，用户可以查看源码。

14. I²C 总线编程说明:

(1) 关键词定义如下: I²Cx_EN、I²Cx_FREQ、I²Cx_SCL、I²Cx_SDA

x 范围: 0~5; 以下说明以 I²C3 为例

(2) 在 IO 配置文件 AMKNXXXX_IOConfig.h 中定义: 参考如下:

```
#define I2C3_SCL    PA8    // I2C3_SCL 管脚
#define I2C3_SDA    PC9    // I2C3_SDA 管脚
```

(3) 在功能配置文件 AMKNXXXX_Config.h 中定义, 参考如下:

```
#define I2C3_EN      1      // I2C3 使能,      1: 打开使能, 0: 关闭
#define I2C3_FREQ    100000 // 读写时钟频率
```

(4) 在 ../libapp/I²C_app.c 中查看 I²C_ApplInit 初始化函数, 这函数根据上面配置初始化 I²C, 用户一般不需要修改该函数, 具体代码用户自行查看。

(5) 因为 I²C 总线的应用一般是外挂 I²C 总线器件, 这里只举外挂 CH455 芯片操作的例子: 在 ../driver/ch455/ch455.c 中的 CH455_Read() 函数: 在这个函数中用户通过 I²C 总线的 I²C_Read 读取按键值。

15. RTC 时钟编程说明：

(1) 关键词定义如下：RTC_EN、RTC_SCAN_T、RTC_SECIT_EN、RTC_ALRIT_EN

(2) 在功能配置文件 AMKNXXX_Config.h 中定义，参考如下：

```
#define RTC_EN          1          // RTC 使能, 1: 打开使能, 0: 关闭
#define RTC_SCAN_T      1000      // 设置定时扫描时间间隔, 单位: ms
#define RTC_SECIT_EN    0          // RTC 秒中断使能, 1: 打开使能, 0: 关闭
#define RTC_ALRIT_EN    0          // RTC 闹钟中断使能, 1: 打开使能, 0: 关闭

// 上电后默认初始化时间定义:

#define RTC_YEAR        22          // 初始化年: 22 表示 2022 年
#define RTC_MONTH       4           // 初始化月: 4 表示 4 月份
#define RTC_DAY         30          // 初始化天: 30 表示 30 日
#define RTC_HOUR        23          // 初始化小时: 23 表示 23 时
#define RTC_MINUTE       59          // 初始化分钟: 59 表示 59 分
#define RTC_SECOND       30          // 初始化秒: 30 表示 30 秒
```

(3) 在../libapp/rtc_app.c 中查看 RTC_AppInit 初始化函数，这函数根据上面配置初始化 RTC，用户一般不需要修改该函数。具体代码用户自行查看。

(4) 在../config/test_config.h 中做如下定义：

```
#define APP_RTC_TEST_EN    1  // RTC 测试使能: 1, 使能; 0, 关闭
```

(5) 在../source/user_testapp.c 中的 User_AppTest() 中会调用 RTC_AppTest() 函数，间隔 1 秒读取 RTC 时间并打印输出。请自行查看读取代码。

16. EEPROM 读写编程说明:

(1) 关键词定义如下: EEPROM_EN、EEPROM_DEVICE、EEPROM_FREQ

(2) 在功能配置文件 AMKNXXX_Config.h 中定义, 参考如下:

```
#define EEPROM_EN      1          // EEPROM 使能, 1: 打开使能, 0: 关闭

#define EEPROM_DEVICE  AT24C64 // 定义器件型号

#define EEPROM_FREQ    100000 // 读写时钟频率

#if (EEPROM_EN == 0)

#error "ERROR: EEPROM_EN 必需使能"

#endif
```

注意: 以上配置用户不要修改。EEPROM 应用 I²C1 总线, 而且管脚定义是固定的

(3) 在 IO 配置文件 AMKNXXX_IOConfig.h 中定义:

```
// 板载 EEPROM 管脚定义(即 I2C1 引脚定义)

#define I2C1_SCL      PB8      // I2C1 SCL 管脚

#define I2C1_SDA      PB9      // I2C1 SDA 管脚
```

(4) 在../libapp/eeprom_app.c 中查看 EEPROM_AppInit 初始化函数, 这函数根据上面配置初始化 EEPROM, 用户一般不需要修改该函数。具体代码用户自行查看。

(5) 在../config/test_config.h 中做如下定义:

```
#define APP_EEPROM_TEST_EN 1 // EEPROM 测试使能: 1, 使能; 0, 关闭

//EEPROM 读写测试配置

#define EEPROM_START_ADDR 1200 // EEPROM 读写起始地址

#define EEPROM_LEN        256 // EEPROM 读写数据长度, 与设置缓存长度一致
```

在../source/user_testapp.c 中的 User_AppTestInit() 中会调用 EEPROM_AppTest() 函数。根据上面配置的读写起始地址和长度进行读写测试。具体测试程序参见../libapp/eeprom_app.c 中 EEPROM_AppTest() 函数。

17. SPIFLASH 读写编程说明:

(1) 关键词定义如下: SPIFLASH_EN、SPIFLASH_MODE、SPIFLASH_TYPE、W25QXX_MODEL、W25QXX_SECTOR_SIZE、W25QXX_SECTOR_NUM、W25QXX_FATFS_STARTSECTOR、W25QXX_FATFS_SECTORNUM、W25QXX_SAVE_PARA_SECTOR、W25QXX_ZDY_STARTSECTOR、W25QXX_ZDY_SECTORNUM、W25QXX_PAGE_SIZE

(2) 在 IO 配置文件 AMKNXXXX_IOConfig.h 中定义: 参考如下:

// SPI1 引脚定义:

```
#define SPI1_REMAP SPI_REMAP_1 // SPI1 重映射 1

#define SPI1_NSS PA15 // SPI1_NSS 管脚, 这个管脚就是板载 SPI FLASH (25QXX) 片选

#define SPI1_SCK PB3 // SPI1_SCK 管脚

#define SPI1_MISO PB4 // SPI1_MOSI 管脚

#define SPI1_MOSI PB5 // SPI1_MISO 管脚

// 板载 SPI FLASH (W25QXX) 片选定义

#define W25QXX_CS PA15
```

注意: 这里配置不可以更改, 在工控模块内 SPIFLASH 是用 SPI1 总线控制, 而且管脚是固定的。

(3) 在功能配置文件 AMKNXXXX_Config.h 中定义, 参考如下:

```
#define SPIFLASH_EN 1 // SPI FLASH 使能: 1, 使能; 0, 关闭;

#define SPIFLASH_MODE 0 // SPI FLASH 操作方式: 1, 利用 FATFS 文件系统进行读写;
// 0, 用 SPI FLASH 读写函数进行操作; 注意: 2 种操作方式只能选择一种

#define SPIFLASH_TYPE W25QXX // SPI FLASH 类型设置: 必须是 W25QXX
// 注意: 因为 W25QXX 是按扇区擦除, 所以建立文件系统就按扇区计算

#define W25QXX_MODEL W25Q64 // SPI FLASH 型号

#define W25QXX_SECTOR_SIZE 4096 // W25QXX 扇区大小

#define W25QXX_SECTOR_NUM 2048 // SPI FLASH 总扇区数

#define W25QXX_FATFS_STARTSECTOR 0 // 用于建立文件系统的起始扇区, 必须是 0

#define W25QXX_FATFS_SECTORNUM 1920 // 用于建立文件系统的扇区数, 这个一般不要修改

#define W25QXX_SAVE_PARA_SECTOR 1920 // 用于存储系统参数扇区, 这个一般不要修改

#define W25QXX_ZDY_STARTSECTOR (W25QXX_SAVE_PARA_SECTOR+1) // 用于自定义扇区起始扇区

#define W25QXX_ZDY_SECTORNUM 120 // 用于自定义扇区数

#define W25QXX_PAGE_SIZE 256 // W25QXX 读写页大小
```

```
#if (SPIFLASH_EN == 0)

    #error "ERROR: SPIFLASH_EN 必需使能"

#endif
```

(4) 在../libapp/spiflash_app.c 中查看 SPIFlash_AppInit 初始化函数，这函数根据上面配置初始化 SPI FLASH，用户一般不需要修改该函数，具体代码用户自行查看。

(5) 在../config/test_config.h 中做如下定义：

```
#define APP_SPIFLASH_TEST_EN    1           // SPIFLASH 测试使能：1，使能；0，关闭

// W25QXX(SPI FLASH) 读写测试配置

// 注意：读写按页(256 字节)，但擦除按扇区(4096 字节)擦除，每个扇区包含 16 个页

// 读写测试全部在扇区 W25QXX_ZDY_STARTSECTOR 里进行

#if (SPIFLASH_TYPE == W25QXX)

// 按页读写配置

#define W25QXX_TEST_START_SECTOR    W25QXX_FATFS_SECTORNUM // W25QXX FLASH 测试起始扇区

#define W25QXX_TEST_SECTOR_NUM      4           // W25QXX FLASH 测试扇区个数

#define W25QXX_TEST_START_PAGE      (W25QXX_TEST_START_SECTOR*16) // W25QXX 读写起始页

#define W25QXX_TEST_PAGE_NUM        (W25QXX_TEST_SECTOR_NUM*16)   // W25QXX 读写数据页数，

                                   // 按字节随机地址读写配置，W25QXX FLASH 读写起始地址

#define W25QXX_TEST_START_ADDR      (W25QXX_TEST_START_PAGE*W25QXX_PAGE_SIZE+100)

#define W25QXX_TEST_LEN              256        // W25QXX FLASH 读写数据长度(256 字节)，小于 512 字节

#endif
```

(6) 在../source/user_testapp.c 中的 User_AppTestInit() 中会调用 W25QXX_AppTest() 函数。根据上面配置的读写起始地址和长度进行读写测试。具体测试程序参见../libapp/spiflash_app.c 中 W25QXX_AppTest() 函数。

18. NET 网络通信编程说明：

(1) 关键词定义如下: LWIP_EN、LWIP_SCAN_T、LWIP_CONFIG_EN、LOCAL_IP、LOCAL_PORT、LOCAL_SUBNET_MASK、LOCAL_GATEWAY、LWIP_TCP_SERVER_EN、MODBUS_TCP_EN、TCP_SERVER_LOCAL_PORT、LWIP_MAX_TCP_SERVER_LINK_NUM、LWIP_UDP_SERVER_EN、UDP_SERVER_LOCAL_PORT、LWIP_MAX_UDP_SERVER_LINK_NUM、LWIP_TCP_CLIENT_EN、LWIP_MAX_TCP_DSC_NUM、LWIP_TCP_DSC1_IP~LWIP_TCP_DSC4_IP、LWIP_TCP_DSC1_PORT~LWIP_TCP_DSC4_PORT、LWIP_UDP_CLIENT_EN、WIP_MAX_UDP_DSC_NUM、LWIP_UDP_DSC1_IP~LWIP_UDP_DSC4_IP、LWIP_UDP_DSC1_PORT~LWIP_UDP_DSC4_PORT、LWIP_TFTP_SERVER_EN、TFTP_SERVER_LOCAL_PORT、LWIP_MAX_TFTP_SERVER_LINK_NUM、LWIP_HTTP_EN、ETH_RXBUFNB、ETH_TXBUFNB、ETH_MAX_RX_PACKET_SIZE、ETH_MAX_TX_PACKET_SIZE、ETH_MDC、ETH_REF、ETH_MDIO、ETH_CRS_DV、ETH_RXD0、ETH_RXD1、ETH_TX_EN、ETH_TXD0、ETH_TXD1、ETH_MCO、ETH_RESET

以下说明以工控板 EMB8602 的网络为例。

(2) 在 IO 配置文件 AMKNXXXX_IOConfig.h 中定义：参考如下：

// 网络管脚定义

```
#define ETH_MDC          PC1      // 推挽复用输出，高速(50MHz)
#define ETH_REF          PA1      // 浮空输入(复位状态)
#define ETH_MDIO         PA2      // 推挽复用输出，高速(50MHz)
#define ETH_CRS_DV       PA7      // 浮空输入(复位状态)
#define ETH_RXD0         PC4      // 浮空输入(复位状态)
#define ETH_RXD1         PC5      // 浮空输入(复位状态)
#define ETH_TX_EN        PG11     // 推挽复用输出，高速(50MHz)
#define ETH_TXD0         PG13     // 推挽复用输出，高速(50MHz)
#define ETH_TXD1         PG14     // 推挽复用输出，高速(50MHz)
#define ETH_MCO          IO_NONE
#define ETH_RESET        PE4      // 推挽输出，复位引脚
```

(3) 在功能配置文件 AMKNXXXX_Config.h 中定义，参考如下：

```
#define TASK_LWIP_EN      1        // LWIP(TCPIP)操作任务： 0，关闭；1，使能运行；
// TCPIP(LWIP)协议栈配置
#define LWIP_EN           TASK_LWIP_EN // TCPIP(LWIP)协议栈使能：1，使能； 0，关闭；
```

```
#define LWIP_SCAN_T      1    // 设置定时扫描时间间隔, 单位: ms

#define LWIP_CONFIG_EN   1    // 配置选择: 1, 按配置设置网络; 0, 按 EEPROM 存储信息配置网络
```

(4) 本机参数设置

```
#define LOCAL_IP          "192.168.1.99"    // 本地 IP

#define LOCAL_PORT        5000             // 本地端口号

#define LOCAL_SUBNET_MASK "255.255.255.0"   // 本地子网掩码

#define LOCAL_GATEWAY     "192.168.1.1"    // 本地网关
```

(5) 本机作为 TCP 服务器模式, 配置

```
#define LWIP_TCP_SERVER_EN 1 // TCP_SERVER 使能: 1, 使能; 0, 关闭;

#if (LWIP_TCP_SERVER_EN > 0)

#define MODBUS_TCP_EN      0 // Modbus TCP 使能: 1, 执行 Modbus TCP 测试使能; 0, 执行服务器
                             // 模式网络测试; 注意使能 MODBUS_TCP_EN, 请使能 MODBUS_SLAVE_EN

#define TCP_SERVER_LOCAL_PORT 502 // 本地端口号

#else

#define TCP_SERVER_LOCAL_PORT 5000 // 本地端口号

#endif

#define LWIP_MAX_TCP_SERVER_LINK_NUM 4 // 最大连接 TCP 客户端个数, 最大设置 8

#endif
```

(6) 本机作为 UDP 服务器模式, 配置

```
#define LWIP_UDP_SERVER_EN 0 // UDP_SERVER 使能: 1, 使能; 0, 关闭;

#if (LWIP_UDP_SERVER_EN > 0)

#define UDP_SERVER_LOCAL_PORT 5100 // 本地端口号

#define LWIP_MAX_UDP_SERVER_LINK_NUM 4 // 最大连接 UDP 客户端个数, 最大 8 个

#endif
```

(7) 本机作为 TCP 客户端模式, 配置

```
#define LWIP_TCP_CLIENT_EN    0  // TCP_CLIENT 使能：1，使能； 0，关闭；

#if (LWIP_TCP_CLIENT_EN > 0)

#define TCP_CLIENT_LOCAL_PORT  5200  // 本地端口号

#define LWIP_TCP_DSC1_EN      1      // 远端服务器 1 使能：1，使能； 0，关闭；
#define LWIP_TCP_DSC2_EN      0      // 远端服务器 2 使能：1，使能； 0，关闭；
#define LWIP_TCP_DSC3_EN      0      // 远端服务器 3 使能：1，使能； 0，关闭；
#define LWIP_TCP_DSC4_EN      0      // 远端服务器 4 使能：1，使能； 0，关闭；
#define LWIP_MAX_TCP_DSC_NUM  2      // 远端 TCP 服务器个数，最大 4 个

#if (LWIP_TCP_DSC1_EN > 0)

#define LWIP_TCP_DSC1_IP      "192.168.1.44"//"192.168.1.200"  // 远端服务器 1 IP
#define LWIP_TCP_DSC1_PORT    5201                      // 远端服务器 1 端口号
#endif

#if (LWIP_TCP_DSC2_EN > 0)

#define LWIP_TCP_DSC2_IP      "192.168.1.44"//"192.168.1.200"  // 远端服务器 2 IP
#define LWIP_TCP_DSC2_PORT    5202                      // 远端服务器 2 端口号
#endif

#if (LWIP_TCP_DSC3_EN > 0)

#define LWIP_TCP_DSC3_IP      "192.168.1.44"//"192.168.1.200"  // 远端服务器 3 IP
#define LWIP_TCP_DSC3_PORT    5203                      // 远端服务器 3 端口号
#endif

#if (LWIP_TCP_DSC4_EN > 0)

#define LWIP_TCP_DSC4_IP      "192.168.1.44"//"192.168.1.200"  // 远端服务器 4 IP
#define LWIP_TCP_DSC4_PORT    5204                      // 远端服务器 4 端口号
#endif

#endif
```

(8) 本机作为 UDP 客户端模式，配置

```
#define LWIP_UDP_CLIENT_EN    0      // UDP_CLIENT 使能：1，使能； 0，关闭；

#if (LWIP_UDP_CLIENT_EN > 0)

#define UDP_CLIENT_LOCAL_PORT  5300  // 本地端口号
```

```
#define LWIP_MAX_UDP_DSC_NUM    2    // 远端 UDP 服务器个数, 最大 4 个

#if (LWIP_MAX_UDP_DSC_NUM > 0)

#define LWIP_UDP_DSC1_IP        "192.168.1.44"//"192.168.1.200"    // 远端服务器 1 IP
#define LWIP_UDP_DSC1_PORT      5301                            // 远端服务器 1 端口号
#endif

#if (LWIP_MAX_UDP_DSC_NUM > 1)

#define LWIP_UDP_DSC2_IP        "192.168.1.44"//"192.168.1.200"    // 远端服务器 2 IP
#define LWIP_UDP_DSC2_PORT      5302                            // 远端服务器 2 端口号
#endif

#if (LWIP_MAX_UDP_DSC_NUM > 2)

#define LWIP_UDP_DSC3_IP        "192.168.1.44"//"192.168.1.200"    // 远端服务器 3 IP
#define LWIP_UDP_DSC3_PORT      5303                            // 远端服务器 3 端口号
#endif

#if (LWIP_MAX_UDP_DSC_NUM > 3)

#define LWIP_UDP_DSC4_IP        "192.168.1.44"//"192.168.1.200"    // 远端服务器 4 IP
#define LWIP_UDP_DSC4_PORT      5304                            // 远端服务器 4 端口号
#endif

#endif
```

(9) 本机作为 TFTP 服务器模式, 配置

```
#define LWIP_TFTP_SERVER_EN    0    // TFTP_SERVER 使能: 1, 使能; 0, 关闭;

#if ((IAP_EN > 0)&&(IAP_TFTP_EN>0))    // 如果 IAP TFTP 使能则 LWIP_TFTP_SERVER_EN 必须使能
#define LWIP_TFTP_SERVER_EN    1    // TFTP_SERVER 使能: 1, 使能; 0, 关闭;
#endif

#if (LWIP_TFTP_SERVER_EN > 0)

#define TFTP_FILE_DISK 1 //TFTP 操作文件盘选择: 0, SPIFLASH_DISK, SPI FLASH 设置为逻辑驱动器 0;
                        //
                        //          1, SD_DISK, SD 卡设置为逻辑驱动器 1;
                        //
                        //          2, USB_DISK, U 盘设置为逻辑驱动器 2;
                        //
                        //          3, NFLASH_DISK, NAND FLASH 设置为逻辑驱动器 3;
```

```
#define TFTP_RRQ_FILE_EN      0    // 利用 TFTP 读取文件使能：1，使能； 0，关闭；
#define TFTP_WRQ_FILE_EN      0    // 利用 TFTP 写入文件使能：1，使能； 0，关闭；
#define TFTP_WEQ_FILE_MODE    2    // 0，如果文件存在，停止写，返回错误
                                   // 1，如果文件存在，打开文件，并在文件结尾添加数据
                                   // 2，如果文件存在，删除原有文件，新建文件

#define TFTP_SERVER_LOCAL_PORT 69  // 本地端口号
#define LWIP_MAX_TFTP_SERVER_LINK_NUM 2 // 最大连接 TFTP 客户端个数，最大 4 个
#endif
```

(10) 本机作为 HTTP 服务器模式，配置

```
#define LWIP_HTTP_EN          0    // HTTP 使能：1，使能； 0，关闭；
```

(11) 驱动中缓存设置

```
#define ETH_RXBUFNB           2
#define ETH_TXBUFNB           2
#define ETH_MAX_RX_PACKET_SIZE 1520
#define ETH_MAX_TX_PACKET_SIZE 1520
```

(12) 在../libapp/net_app.c 中查看 NET_AppProc 处理函数：

这个函数功能是网络初始化、建立通信任务、检测连接状态等功能。程序比较长，就不在这里列出代码了。

在有操作系统的例程里，各个模式的网络通信功能在../source/Tasklwip.c 中实现，具体如下：

App_TaskTCPCClient1~App_TaskTCPCClient4：TCP 客户端模式通信任务

App_TaskUDPCClient：UDP 客户端模式通信任务

App_TaskTCPSTServer：TCP 服务端模式通信任务

App_TaskUDPSTServer：UDP 服务端模式通信任务

App_TaskTFTPSTServer：TFTP 服务器模式通信任务

用户不需要修改该代码，网络接收的数据通过 NET_TCPCClientDataProcess、NET_UDPCClientDataProcess、NET_TCPSTServerDataProcess、NET_UDPSTServerDataProcess 等函数处理。用户只需在这些函数中做应用处理就行。这些函数在../source/user_netwifiapp.c，请自行查看源码。

19. USB 通信编程说明:

(1) 关键词定义如下:USB_HOST_EN、UDISK_EN、USB_DEVICE_EN、USB_MSC_EN、USB_MSC_LUN、USB_SCAN_T、USB_VCP_EN、USB_RXBUF_SIZE、USB_PWR、USB_DM、USB_DP、USB_VBUS

(2) 在 IO 配置文件 AMKNXXXX_IOConfig.h 中定义: 参考如下:

```
#define USB_PWR      PD4

#define USB_DM       PA11

#define USB_DP       PA12

#define USB_VBUS     PA9
```

注意: 以上端口定义除了 USB_PWR 外, 不可以更改。

(3) 在功能配置文件 AMKNXXXX_Config.h 中定义, 参考如下:

```
#define USB_SCAN_T    10      // 设置定时扫描时间间隔, 单位: ms

// USB 主机模式配置, 注意: USB_HOST_EN 和 USB_DEVICE_EN 不能同时使能

#define USB_HOST_EN   0      // USB 主机模式使能: 1, 使能; 0, 关闭;

#define UDISK_EN      1      // U 盘使能: 1, 使能; 0, 关闭;

// USB 设备模式设置

#define USB_DEVICE_EN  0      // USB 设备设备使能: 1, 使能; 0, 关闭;

// USB 设备, 实现将板子的 SD 卡、SPI FLASH 或者 NAND 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     0      // 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 设置为逻辑驱动器; 1, SD 卡设置为逻辑驱动器;

// USB 设备 虚拟串口通信设置

// 注: USB_VCP_EN 和 USB_MSC_EN 不能同时使能

#define USB_VCP_EN     1      // USB VCP 虚拟串口使能, 1: 打开使能, 0: 关闭

#define USB_RXBUF_SIZE 512 // 定义接收缓存长度, 这个参数决定每虚拟串口最大接收数据长度
```

(4) 在 ../libapp/usb_app.c 中查看 USBH_ApplInit 和 USBD_ApplInit 初始化函数, 这函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

(5) 在../libapp/usb_app.c 中查看 USB_AppProc 处理函数:

```
void USB_AppProc(void)
{
    INT32U tick;

    // 在 USB 主机模式下实时同步 USB 状态, 当插拔 U 盘时用这个程序进行相应处理
    #if (USB_HOST_EN > 0)                // 条件编译 USB 主机模式使能
        tick = APP_GetSubTick(LibAppVars.USBH.Scan_t); // 读取和上次采集的间隔时间, 单位 ms
        if (tick >= USB_SCAN_T)          // 计算和上次采集时间间隔大于等于设置值
        {
            LibAppVars.USBH.Scan_t += tick;        // 记录本次扫描时间
            USBH_Ctrl(USB_ID, CMD_USBH_SYNC, 0);    // USB 主机同步处理
        }
    #endif

    // USB 设备模式下, 用 USB Mass Storage 协议操作 SD 卡或 SPI FLASH。当用 USB 线连接板子和计算
    // 机时, 计算机会把板子上的 SD 卡或 SPI FLASH 当作 U 盘来操作
    #if ((USB_DEVICE_EN > 0)&&(USB_MSC_EN > 0)) // 条件编译 USB 设备模式使能及
        // USB Mass Storage 使能
        tick = APP_GetSubTick(LibAppVars.USBD.Scan_t); // 读取和上次采集的间隔时间, 单位 ms
        if (tick >= USB_SCAN_T)          // 计算和上次采集时间间隔大于等于设置值
        {
            LibAppVars.USBD.Scan_t += tick;        // 记录本次扫描时间
            USBD_Ctrl(USB_ID, CMD_USBD_SYNC, 0)     // USB 设备同步处理, 检测 USB 设
                                                    // 备插入并启动 USB, 监测 USB 设备拔出并关闭 USB 设备;
        }
    #endif

    // USB 设备模式下, 工作在虚拟串口模式
    #if ((USB_DEVICE_EN > 0)&&(USB_VCP_EN > 0)) // 条件编译虚拟串口模式
        INT16U len;
        INT32S flag;
        tick = APP_GetSubTick(LibAppVars.USBD.Scan_t); // 读取和上次采集的间隔时间, 单位 ms
```

```

if (tick >= USB_SCAN_T)                                // 计算和上次采集时间间隔大于等于设置值
{
    LibAppVars.USB.D.Scan_t += tick;                    // 记录本次扫描时间
    flag = USB_D_Ctrl(USB_ID, CMD_USBD_SYNC, 0); // USB 同步处理, 检测 USB 设备插入并启动
                                                    // USB, 监测 USB 设备拔出并关闭 USB 设备;

    if (flag&USBD_WORK_OK)                             // 判断虚拟串口正常工作
    {
        len = USB_D_Ctrl(USB_ID, CMD_USBD_GetCharsRxBuf, 0); // 读取接收数据长度
        if ((len == LibAppVars.USB.D.len)&&(len>0))
        {
            USB_D_Read(USB_ID, LibApp_USBRxBuf, len); // 读取 USB 数据
            LibAppVars.Register.APP_InputDataCallback(LIBAPP_DATA_USB_TYPE, USB_ID,
                len, LibApp_USBRxBuf); // 调用回调函数, 发送消息数据
            LibAppVars.USB.D.len -= len;
        }
        else
        {
            LibAppVars.USB.D.len = len;
        }
    }
}
#endif

```

(6) 在虚拟串口模式 USB 接收数据被发送到../source/user_usbapp.c 中的 USB_DataProcess 函数, 用户请在这个函数做相应的处理:

```

void USB_DataProcess(INT8U id, INT8U *p, INT16U len)
{
    INT16U i;

    #if (APP_USB_DEBUG_EN == 1)
        printf("AT+USB=RX[%d]:%02X", len, p[0]);
    #endif
}

```

```
for (i=1; i<MIN(len, DEBUG_MAX_DATA_LEN); i++)
{
    printf(" %02x", p[i]);
}

printf("\r\n");

#endif

// 用户在这里处理 USB 接收到的数据, 数据在缓存 p 中, 数据长度 len

#if (APP_USB_TEST_EN > 0)
// 测试功能: 将数据在原样发送回去
// USBD_Write(id, pbuf, len);
USB_UserSendData(id, p, len);
#endif
}
```

(7) 在 USB 主机模式下, 插入 U 盘进行 U 盘读写测试。

在../config/test_config.h 中查看 U 盘读写测试配置:

```
#define APP_UDISK_FILE_TEST_EN    1    // 文件读写测试: 0, 关闭; 1, 使能
#if (APP_UDISK_FILE_TEST_EN > 0)
#define APP_UDISK_FILE_TEST_MODE  1    // 文件测试模式: 0, 直接用 FileApp 中函数进行文件
                                        //读写测试; 1, 直接用 APP_FileCtrl 函数进行文件读写测试
#define APP_UDISK_FILE_FBUF_SIZE  700  // 测试数据包大小定义, 大于等于 512, 小于 1024
// 可以在下面更改测试读写数据包数量
#define APP_UDISK_FILE_PAGE_NUM    10  // 设置读写测试数据包数量
#endif
在../source/user_testapp.c 中查看 U 盘读写测试程序 User_FileAppTest
```

20. SD 卡读写编程说明：

(1) 关键词定义如下：SDCARD_EN、SD_CS、SD_INR、SD_PWR、SD_WP

(2) 在 IO 配置文件 AMKNXXX_IOConfig.h 中定义：参考如下：

// SD 卡控制 IO 定义

```
#define SD_CS      PE5  
  
#define SD_INR     PE6  
  
#define SD_PWR     PE7  
  
#define SD_WP      PC13
```

注意：以上端口定义根据硬件设计来定义。如果 SD 卡采用 SPI1 接口通信，可在本文件中查看 SPI1 端口定义。如果 SD 卡采用 SDMMC 接口通信，可在本文件中查看 SDMMC 端口定义：

// SDMMC 管脚定义

```
#define SDMMC_CMD  PD2  
  
#define SDMMC_CLK  PC12  
  
#define SDMMC_D0    PC8  
  
#define SDMMC_D1    PC9  
  
#define SDMMC_D2    PC10  
  
#define SDMMC_D3    PC11
```

(3) 在功能配置文件 AMKNXXX_Config.h 中定义，参考如下：

```
#define SDCARD_EN      1      // SD 卡使能：1，使能； 0，关闭；  
  
#if (SDCARD_EN > 0)  
  
#define SD_MODE        SD_SDMMC_MODE  // SD 卡接口模式：0(SD_SPI_MODE)，SPI 操作模式；  
                                       // 1(SD_SDMMC_MODE)，SDMMC 操作模式；  
  
#define SD_PWR_EN      0      // SD 卡自动供电使能：0，由管脚控制供电；  
                                       // 1，自动供电，无需控制，板子上电直接供电；  
  
#if (SD_MODE == SD_SDMMC_MODE)  
  
#define SD_SDMMC_ID    SDMMC1_ID  
  
#define SD_BUS_WIDE    SDMMC_BUS_4BIT // 选择：1，SDMMC_BUS_1BIT；4，SDMMC_BUS_4BIT；  
  
#endif  
  
#if (SD_MODE == SD_SPI_MODE)
```

```
#define SD_SPI_ID SPI1_ID

#endif

#endif
```

// 文件系统使能：1，使能； 0，关闭；

```
#define FATFS_EN (SDCARD_EN | (SPIFLASH_EN & SPIFLASH_MODE) | UDISK_EN)

#define FATFS_SCAN_T 10 // 设置定时扫描时间间隔，单位：ms
```

(4) 在../libapp/libapp.c 中查看 SD_AppInit 初始化函数，这函数根据上面配置初始化，用户一般不需要修改该函数，具体代码用户自行查看。

(5) 在../libapp/file_app.c 中查看 File_AppProc 处理函数：

```
void File_AppProc(void)
{
    INT32U tick;

    #if (SDCARD_EN > 0) // 条件编译 SD 卡使能
        tick = APP_GetSubTick(LibAppVars.SD.Scan_t); // 读取和上次扫描的间隔时间，单位 ms
        if (tick >= FATFS_SCAN_T) // 计算和上次扫描时间间隔大于等于设置值
        {
            LibAppVars.SD.Scan_t += tick; // 记录本次扫描时间
            File_SDProc(); // SD 卡处理：文件初始化、文件句柄创建、SD 卡状态监测
        }
    #endif

    ... ..
}
```

具体用户查看 File_SDProc 函数处理过程。

(6) 以下测试说明，在插入 SD 后，会对 SD 卡进行读写测试。

在../config/test_config.h 中查看 SD 卡读写测试配置：

```
#define APP_SD_FILE_TEST_EN 1 // 文件读写测试：0, 关闭；1, 使能

#if (APP_SD_FILE_TEST_EN > 0)
```

```
#define APP_SD_FILE_TEST_MODE 0 // 文件测试模式：0，直接用 FileApp 中函数进行文件读
                                // 写测试；1，直接用 APP_FileCtrl 函数进行文件读写测试

// 可以在下面更改测试读写数据包大小

#define APP_SD_FILE_FBUF_SIZE 700 // 测试数据包大小定义，大于等于 512, 小于 1024

// 可以在下面更改测试读写数据包数量

#define APP_SD_FILE_PAGE_NUM 16 // 设置读写测试数据包数量

#endif

请用户在 ../source/user_testapp.c 中查看 SD 卡读写测试程序 User_FileAppTest()
```

21. NAND FLASH 读写编程说明:

(1) 关键词定义如下: NFLASH_EN、NFLASH_RBIT_EN、NFLASH_ECCEN、NFLASH_ECC_SIZE、NFLASH_BLOCK_NUM、NFLASH_BLOCK_SIZE、NFLASH_PAGE_SIZE、NFLASH_MAX_BAD_BLOCK

(2) Nand Flash 只在 STM32F103ZE/GD32F303ZE 模块上应用, 管脚在驱动库内部定义, 所以在 IO 配置文件 AMKNXXX_I0Config.h 中不在做定义, 用户无需关注。

(3) 在功能配置文件 AMKNXXX_Config.h 中定义, 参考如下:

```
#define NFLASH_EN          1 // Nand Flash 使能: 1, 使能; 0, 关闭;

#define NFLASH_RBIT_EN    1 // Nand Flash RB 信号中断使能: 1, 使能; 0, 关闭; 当 Nand Flash
// 作为 USB Mass Storage 存储介质 (即 USB_DEVICE_EN=1, USB_MSC_EN=1, USB_MSC_LUN=3) 时,
// RB 信号中断必须关闭;

#define NFLASH_ECCEN      0 // Nand Flash ECC 校验使能: 1, 使能; 0, 关闭;

#define NFLASH_ECC_SIZE   512 // Nand Flash ECC 页面大小: 256/512/1024/2048/4096 字节可选

#define NFLASH_BLOCK_NUM  2048 // Nand Flash 总块数

#define NFLASH_BLOCK_SIZE 64 // Nand Flash 每个块包含页数

#define NFLASH_PAGE_SIZE  2048 // Nand Flash 页大小

#define NFLASH_MAX_BAD_BLOCK 64 // Nand Flash 最大坏块数, 这个部分用于替换数据区坏块
```

(4) 在 ../libapp/libapp.c 中查看 NFlash_Applnit 初始化函数, 这函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

(5) 在 ../libapp/file_app.c 中查看 File_AppProc 处理函数:

```
void File_AppProc(void)
{
    INT32U tick;

    #if (NFLASH_EN > 0) // 条件编译 NAND FLASH 使能

    tick = APP_GetSubTick(LibAppVars.NFlash.Scan_t); // 读取和上次扫描的间隔时间, 单位 ms
    if (tick >= FATFS_SCAN_T) // 计算和上次扫描时间间隔大于等于设置值
    {
        LibAppVars.NFlash.Scan_t += tick; // 记录本次扫描时间

        File_NFlashProc(); // NAND FLASH 处理: 文件初始化、文件句柄创建、状态监测
    }
}
```

```
    }  
    #endif  
    ... ..  
}
```

具体用户查看 File_NFlashProc 函数处理过程。

(6) 以下测试说明，板子上电后，会对 Nand Flash 进行读写测试。

在 ../config/test_config.h 中查看读写测试配置：

```
#define APP_NFLASH_FILE_TEST_EN    1    // 文件读写测试：0, 关闭；1, 使能  
  
#if (APP_NFLASH_FILE_TEST_EN > 0)  
  
#define APP_NFLASH_FILE_TEST_MODE  1    // 文件测试模式：0, 直接用 FileApp 中函数进行  
// 文件读写测试； 1, 直接用 APP_FileCtrl 函数进行文件读写测试  
// 可以在下面更改测试读写数据包大小  
  
#define APP_NFLASH_FILE_FBUF_SIZE  700  // 测试数据包大小定义，大于等于 512, 小于 1024  
// 可以在下面更改测试读写数据包数量  
  
#define APP_NFLASH_FILE_PAGE_NUM   10   // 设置读写测试数据包数量  
  
#endif
```

用户在 ../source/user_testapp.c 中查看 Nand Flash 读写测试程序 User_FileAppTest()

22. Modbus 主机通信编程说明:

(1) 关键词定义如下: MODBUS_EN、MODBUS_MODE、MODBUS_CH1~MODBUS_CH4、MODBUS_ID、MODBUS_TIMEOUT、MODBUS_RXSCAN_T、MODBUS_BUF_SIZE、MODBUS_SCAN_T、MODBUS_CH_MAX_NUM

(2) 在功能配置文件 modbus_config.h 中定义, 参考如下:

```
#define MODBUS_EN          0          // MODBUS 通信使能: 1, 使能; 0, 关闭;

#define MODBUS_MODE        0          // MODBUS 通信模式: 0, RTU; 1, ASCII 码(暂不支持);

#define MODBUS_CH_MAX_NUM  4          // MODBUS 通信通道数量: 1~4

#define MODBUS_TIMEOUT     1000       // MODBUS 通信超时时间, 单位 ms;

#define MODBUS_RXSCAN_T    10         // MODBUS 函数内部接收数据时扫描间隔

#define MODBUS_BUF_SIZE    64         // MODBUS 函数内部工作缓存长度, 范围大于 0, 根据自己实际
// 需要设置, 不可以太大;
```

```
#define MODBUS_SCAN_T      20         // 设置定时扫描时间间隔, 单位: ms
```

(3) MODBUS_CH1: 通道 1 配置说明:

```
#define MODBUS_CH1_EN      1          // MODBUS 通信通道 1 使能: 1, 使能; 0: 关闭;

#if (MODBUS_CH1_EN > 0)

#define MODBUS_CH1         UART5_ID   // MODBUS 通信通道 1: 0: UART1_ID, 1: UART2_ID, 2: UART3_ID,
// 3: UART4_ID, 4: UART5_ID;

#define MODBUS_CH1_MAX_ERROR_NUM 3    // 最大通信出错次数, 超过这个此时表示该设备通信故障, 不
// 在启动通信

// 从机设备寄存器缓存大小配置

#define MODBUS_CH1_MAX_COILS_NUM 32   // MODBUS 从机最大线圈(DO)数量(读写, 可用功能码:1, 5, 15);

#define MODBUS_CH1_MAX_DISINPUT_NUM 32 // MODBUS 从机最大离散输入量(DI)(只读, 可用功能码:2);

#define MODBUS_CH1_MAX_HOLDREG_NUM 16 // MODBUS 从机最大保持寄存器(读写, 可用功能码:3, 6, 16)数量;

#define MODBUS_CH1_MAX_INPUTREG_NUM 16 // MODBUS 从机最大输入寄存器(AI)(只读, 可用功能码:4)数量;

// 从机设备操作使能控制

#define MODBUS_CH1_DISINPUT_EN        1    // 从机设备 DI(离散)数字量输入寄存器操作使能

#define MODBUS_CH1_COILS_EN           1    // 从机设备 DO(线圈)数字量输出寄存器操作使能

#define MODBUS_CH1_INPUTREG_EN        1    // 从机设备 AI 模拟量输入寄存器操作使能

#define MODBUS_CH1_HOLDREG_EN         1    // 从机设备保持寄存器操作使能
```

```
#endif // #define MODBUS_CH1_EN
```

(4) 在../libapp/modbus_app.c 中查看 Modbus_AppInit 初始化函数，这函数根据上面配置初始化，用户一般不需要修改该函数，具体代码用户自行查看。

(5) 在../source/TaskModbus.c 中查看 Modbus_AppTest 处理函数：这个函数实现对外部 Modbus 设备寄存器的读写操作，具体用户自行查看代码。

23. Modbus Slave 从机通信编程说明:

(1) 关键词定义如下: MODBUS_SLAVE_EN、MODBUS_SLAVE_MODE、MODBUS_SLAVE_CH、MODBUS_SLAVE_ID、MODBUS_COILS_BASEADDR、MODBUS_DISINPUT_BASEADDR、MODBUS_HOLDREG_BASEADDR、MODBUS_INPUTREG_BASEADDR、MODBUS_MAX_COILS、MODBUS_MAX_DISINPUT、MODBUS_MAX_HOLDREG、MODBUS_MAX_INPUTREG

(2) 在功能配置文件 modbus_slave_config.h 中定义, 参考如下:

```
#define MODBUS_SLAVE_EN      0      // MODBUS 从机通信使能: 1, 使能; 0, 关闭;

#if (MODBUS_SLAVE_EN > 0)

#define MODBUS_SLAVE_MODE    0      // MODBUS 从机通信模式: 0, RTU; 1, ASCII 码(暂不支持);

#define MODBUS_SLAVE_CTRL_EN  1      // MODBUS 从机通信 A0/D0/PWM 控制使能: 1, 使能; 0, 关闭;

#define MODBUS_SLAVE_VIRUART_EN 1      // MODBUS 从机通信虚拟串口 VIR UART 使能: 1, 使能; 0, 关闭;

                                   // 只应用到 STM32MP15X 模块

#define MODBUS_SLAVE_UART_EN  0      // MODBUS 从机通信 UART 通道使能: 1, 使能; 0, 关闭;

// 设置多通道通信(CH1-CH4)

#if (MODBUS_SLAVE_UART_EN > 0)

#define MODBUS_SLAVE_CH      UART3_ID // MODBUS 从机通信通道 1: UART1_ID~UART10_ID;

// #define MODBUS_SLAVE_CH2    UART2_ID // MODBUS 从机通信通道 2: UART1_ID~UART10_ID;

// #define MODBUS_SLAVE_CH3    UART3_ID // MODBUS 从机通信通道 2: UART1_ID~UART10_ID;

// #define MODBUS_SLAVE_CH4    UART4_ID // MODBUS 从机通信通道 2: UART1_ID~UART10_ID;

#endif

#define MODBUS_SLAVE_ID      1        // MODBUS 从机通信地址码, 范围: 1~255;

#define MODBUS_COILS_BASEADDR 0        // 线圈寄存器基地址;

#define MODBUS_DISINPUT_BASEADDR 0     // 离散输入量寄存器基地址;

#define MODBUS_HOLDREG_BASEADDR 0      // 保持寄存器基地址;

#define MODBUS_INPUTREG_BASEADDR 0     // 输入寄存器基地址;

#define MODBUS_MAX_COILS     64        // MODBUS 从机最大线圈数量(读写, 可用功能码:1, 5, 15);

#define MODBUS_MAX_DISINPUT  64        // MODBUS 从机最大离散输入量(只读, 可用功能码:2);

#define MODBUS_MAX_HOLDREG   256      // MODBUS 从机最大保持寄存器(读写, 可用功能码:3, 6, 16, 23)数量;

#define MODBUS_MAX_INPUTREG  128      // MODBUS 从机最大输入寄存器(只读, 可用功能码:4)数量;
```

```
#define MODBUS_CONFIG_HOLDREG_BASEADDR    60000    // 配置保持寄存器基地址;

#if ((IAP_EN > 0)&&(IAP_MODBUS_EN > 0))

#define MODBUS_MAX_CONFIG_HOLDREG    2048    // MODBUS 从机最大配置保持寄存器(读写, 可用功能
                                              // 码:3, 6, 16)数量;

#else

#define MODBUS_MAX_CONFIG_HOLDREG    28    // MODBUS 从机最大配置保持寄存器(读写, 可用功能
                                              //码:3, 6, 16)数量;

#endif

#define MODBUS_TCP_LOCAL_PORT            502    // 本地端口号

#define MODBUS_SLAVE_SCAN_T            10    // 默认扫描时间
```

(3) 在../libapp/modbus_slave_app.c 中查看 ModbusSlave_AppInit 初始化函数, 这函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

(4) 在../source/user_uartapp.c 中查看 Uart_DataProcess 处理函数。这个函数判断 UART 接收的数据, 并进行 Modbus 协议解析代码如下:

```
#if (MODBUS_SLAVE_EN > 0)&&(MODBUS_SLAVE_UART_EN > 0)    // Modbus 从机使能/UART 通道使能

#ifdef MODBUS_SLAVE_CH

    if (MODBUS_SLAVE_CH == id)

    {

        ModbusRTU_Process(id, MODBUS_SLAVE_ID, p, len, 0);    // 接收数据进行 Modbus 处理

        return;

    }

#endif

#ifdef MODBUS_SLAVE_CH2

    if (MODBUS_SLAVE_CH2 == id)

    {

        ModbusRTU_Process(id, MODBUS_SLAVE_ID, p, len, 0);    // 接收数据进行 Modbus 处理

        return;

    }

}
```

```
#endif
```

```
#endif
```

ModbusRTU_Process 是 ModbusRTU 协议处理函数，在../libapp/modbus_slave_app.c 中，请自行查看代码
注意：在 ModbusRTU_Process 函数中调用 Modbus_Proc，Modbus_Proc 是解析 Modbus 协议，并自动完成本机寄存器读写。

(4) 本机寄存器在../libapp/modbus_slave_app.c 中定义：

```
MODBUS_DATA ModbusData;                // 缓存数据

__attribute__((aligned(4))) INT8U  ModbusCoils[(MODBUS_MAX_COILS-1)/8 + 1]; // 输出线圈数组
__attribute__((aligned(4))) INT8U  ModbusDisInput[(MODBUS_MAX_DISINPUT-1)/8 + 1]; // 输入离散
输入量数组

__attribute__((aligned(4))) INT16U ModbusHoldReg[MODBUS_MAX_HOLDREG];        // 保持寄存器
__attribute__((aligned(4))) INT16U ModbusInputReg[MODBUS_MAX_INPUTREG];      // 输入寄存器
__attribute__((aligned(4))) INT16U ModbusConfigHoldReg[MODBUS_MAX_CONFIG_HOLDREG]; // 配置保持
寄存器

MODBUS_PARA ModbusPara;                // 参数定义
MODBUS_PARA *pModbusPara;              // 参数指针定义
```

注意：寄存器的数据结构指针由../libapp/modbus_slave_app.h 引出：

```
extern MODBUS_PARA *pModbusPara;        // 参数指针定义
```

用户在应用程序中 pModbusPara 这个指针操作寄存器，如：

```
pModbusPara->pDisInput[0] = (pVars->DI.nval[0]>>0)&0xff;
```

24. WIFI 通信编程说明:

(1) 关键词定义如下: WIFI_EN、WIFI_UART、WIFI_SCAN_T、WIFI_CONFIG_EN、WIFI_WKMODE、WIFI_NETYPE、WIFI_TC_EN、WIFI_ROUTER_SSID、WIFI_ROUTER_PASSWORD、WIFI_SSID、WIFI_PASSWORD、WIFI_LOCAL_GATEWAY、WIFI_LOCAL_IP 、 WIFI_LOCAL_PORT 、 WIFI_SUBNET_MASK 、 WIFI_DSC_IP 、 WIFI_DSC_PORT 、 WIFI_TCP_SERVER_TIMEOUT

(2) WIFI 所有源码文件在../**device/wifi**/里, 包括: wifi.c/wifi.h, wifi_app.c/wifi_app.h, wifi_e103-w01.c, wifi_bw16_wb2.c, wifi_config.h

(3) 在功能配置文件 wifi_config.h 中定义, 参考如下:

```
#define WIFI_EN          1                // WIFI 使能: 1, 使能; 0, 关闭;

#if (WIFI_EN > 0)

#include "wifi/wifi.h"

#include "wifi/wifi_app.h"

#define APP_WIFI_TEST_EN  1  // WIFI 测试使能(远端设备发来的数据原数据发回):1, 使能; 0, 关闭
#define APP_WIFI_TX_TEST_EN 0 // WIFI 自动发送测试使能(间隔 1 秒发送"0123456789ABCDEF"):
                                // 1, 使能; 0, 关闭

#define WIFI_DEBUG_EN     1 // WIFI 打印使能: 0, 关闭; 1, 使能关键信息打印; 2, 使能全部信息打印;
#define WIFI_CONFIG_EN    1 // 配置选择: 1, 按配置设置网络; 0, 按 EEPROM 存储信息配置网络
#define WIFI_TYPE         WIFI_E103_W01 // 目前 WIFI_E103_W01/WIFI_BW16/WIFI_BW2 可以选择
#define WIFI_UART         UART10_ID      // 选择串口: UART1_ID~UART10_ID; 注意: 请务必将
                                // AMKNxxxx_Config.h 中 UARTx_RX_EN 设置为 0"
#define WIFI_SCAN_T       5              // 设置定时扫描时间间隔, 单位: ms, 范围 1-5ms
#define WIFI_WKMODE       WIFI_STATION  // 选择 WIFI 模块工作模式: WIFI_STATION
#define WIFI_ENRYPTION    WIFI_WPA_WPA2_PSK // WIFI 模块加密方式定义: WIFI_OPEN/WIFI_WPA_PSK
                                // WIFI_WPA2_PSK/WIFI_WPA_WPA2_PSK

#define WIFI_CH            5              // WIFI 通道定义, 范围: 1-64
#define WIFI_ROUTER_SSID   "ZQLY"        // 选择 WIFI 路由器接入点名称
#define WIFI_ROUTER_PASSWORD "zqly10802" // 选择 WIFI 路由器接入点密码
#define WIFI_AP_SSID       "AMKN"        // 本 WIFI 模块作为 AP 接入点名称
```

```
#define WIFI_AP_PASSWORD      "1234"          // 本 WIFI 模块作为 AP 接入点密码

#define WIFI_LOCAL_IP         "192.168.1.98"    // 选择 WIFI 模块本地 IP

#define WIFI_LOCAL_PORT       5000              // 选择 WIFI 模块本地端口

#define WIFI_LOCAL_GATEWAY     "192.168.1.1"     // 选择 WIFI 模块本地网关

#define WIFI_SUBNET_MASK       "255.255.255.0"  // 选择 WIFI 模块本地子网掩码

#define WIFI_MAX_LINK_JGTIME   3000            // WIFI 连接失败后再次连接的时间间隔

#define WIFI_TCP_SERVER_TIMEOUT 600            // 设置 TCP Server 超时时间, 超时无通信断开连接, 范围
                                              // 0-7200, 设置为 0 时永远不超时
```

(4) 通信通道配置

设置 WIFI LINK ID1 固定作为 TCP SERVER 模式通信

注意 1: 该通道必须固定设置为 TCP SERVER

注意 2: 如果想要最大连接 2 个客户端, 则下面的 WIFI_LINK_ID2 通道必须使能, 并且除了 WIFI_LINK_ID2_AUTO 设置 0 以外, 其它设置必须和本设置相同;

注意 3: 如果想要最大连接 3 个客户端, 则下面的 WIFI_LINK_ID2/WIFI_LINK_ID3 通道必须使能, 并且除了 WIFI_LINK_ID2_AUTO/WIFI_LINK_ID3_AUTO 设置 0 以外, 其它设置必须和本设置相同;

注意 4: 如果想要最大连接 4 个客户端, 则下面的 WIFI_LINK_ID2/WIFI_LINK_ID3/WIFI_LINK_ID4 通道必须使能, 并且除了 WIFI_LINK_ID2_AUTO/WIFI_LINK_ID3_AUTO/WIFI_LINK_ID4_AUTO 设置 0 以外, 其它设置必须和本设置相同;

注意 5: 如果想要最大连接 5 个客户端, 则下面的 WIFI_LINK_ID2/WIFI_LINK_ID3/WIFI_LINK_ID4

/WIFI_LINK_ID5 通道必须使能, 并且除了 WIFI_LINK_ID2_AUTO/WIFI_LINK_ID3_AUTO

/WIFI_LINK_ID4_AUTO/WIFI_LINK_ID5_AUTO 设置 0 以外, 其它设置必须和本设置相同;

```
#define WIFI_LINK_ID1_EN      1                // WIFI 连接通道 1 使能: 1, 使能; 0, 关闭;

#if (WIFI_LINK_ID1_EN > 0)

#define WIFI_LINK_ID1_AUTO    1                // 通道 1 自动启动定义: 1, 自动连接; 0, 手动连接

#define WIFI_LINK_ID1_TYPE     WIFI_TCP_SERVER // 通道 1 通信模式定义: 0, WIFI_TCP_CLIENT, 通
// 信类型 TCP 客户端; 1, WIFI_TCP_SERVER, 通信类型 TCP 服务端; 2, WIFI_UDP_CLIENT, 通信类型 UDP
// 客户端; 3, WIFI_UDP_SERVER, 通信类型 UDP 服务端;

#define WIFI_LINK_ID1_REMOTE_IP ""            // 选择远端服务器 IP

#define WIFI_LINK_ID1_REMOTE_PORT 0           // 选择远端服务器端口号

#define WIFI_MODBUS_TCP_EN     0                // Modbus TCP 使能: 1, 执行 Modbus TCP 测试使能;
```

// 0, 执行服务器模式网络测试

```
#if (WIFI_MODBUS_TCP_EN > 0)

#define WIFI_MODBUS_SLAVE_ID      1           // 配置 ModbusTCP 设备地址(ID)
#define WIFI_LINK_ID1_LOCAL_PORT  502         // 选择 WIFI 模块 MODBUS TCP 本地端口
#else

#define WIFI_LINK_ID1_LOCAL_PORT   WIFI_LOCAL_PORT // 选择本地口号,
#endif

#endif

// 设置 WIFI LINK ID2 作为 TCP SERVER 模式通信

#define WIFI_LINK_ID2_EN          1           // WIFI 连接通道 1 使能: 1, 使能; 0, 关闭;
#if (WIFI_LINK_ID2_EN > 0)

#define WIFI_LINK_ID2_AUTO        0           // 通道 2 自动启动定义: 1, 自动连接; 0, 手动连接
#define WIFI_LINK_ID2_TYPE        WIFI_TCP_SERVER // 通道 1 通信模式定义: 0, WIFI_TCP_CLIENT, 通
// 信类型 TCP 客户端; 1, WIFI_TCP_SERVER, 通信类型 TCP 服务端; 2, WIFI_UDP_CLIENT, 通信类型
// UDP 客户端; 3, WIFI_UDP_SERVER, 通信类型 UDP 服务端;
#define WIFI_LINK_ID2_REMOTE_IP    ""          // 选择远端服务器 IP
#define WIFI_LINK_ID2_REMOTE_PORT  0           // 选择远端服务器端口号
#define WIFI_LINK_ID2_LOCAL_PORT   WIFI_LINK_ID1_LOCAL_PORT // 选择本地口号, 作为 TCP SERVER
// 通信, 必须和 WIFI_LINK_ID1 本地端口相同

#endif

// 设置 WIFI LINK ID3 作为 TCP CLENT 模式通信

#define WIFI_LINK_ID3_EN          0           // WIFI 连接通道 3 使能: 1, 使能; 0, 关闭;
#if (WIFI_LINK_ID3_EN > 0)

#define WIFI_LINK_ID3_AUTO        1           // 通道 3 自动启动定义: 1, 自动连接; 0, 手动连接
#define WIFI_LINK_ID3_TYPE        WIFI_TCP_CLIENT // 通道 1 通信模式定义: 0, WIFI_TCP_CLIENT,
//通信类型 TCP 客户端; 1, WIFI_TCP_SERVER, 通信类型 TCP 服务端; 2, WIFI_UDP_CLIENT, 通信类型 UDP
//客户端; 3, WIFI_UDP_SERVER, 通信类型 UDP 服务端;
#define WIFI_LINK_ID3_REMOTE_IP    "192.168.1.44" // 选择远端服务器 IP
```

```
#define WIFI_LINK_ID3_REMOTE_PORT    6003           // 选择远端服务器端口号

#define WIFI_LINK_ID3_LOCAL_PORT     5003           // 选择本地口号

#endif


// 设置 WIFI LINK ID4 作为 UDP CLIENT 模式通信

#define WIFI_LINK_ID4_EN              0              // WIFI 连接通道 4 使能：1，使能； 0，关闭；

#if (WIFI_LINK_ID4_EN > 0)

#define WIFI_LINK_ID4_AUTO            1              // 通道 4 自动启动定义：1，自动连接；0，手动连接

#define WIFI_LINK_ID4_TYPE            WIFI_UDP_CLIENT // 通道 1 通信模式定义：0, WIFI_TCP_CLIENT, 通信
// 类型 TCP 客户端； 1,  WIFI_TCP_SERVER, 通信类型 TCP 服务端； 2, WIFI_UDP_CLIENT, 通信类型 UDP
// 客户端； 3,  WIFI_UDP_SERVER, 通信类型 UDP 服务端；

#define WIFI_LINK_ID4_REMOTE_IP       "192.168.1.44" // 选择远端服务器 IP

#define WIFI_LINK_ID4_REMOTE_PORT     6004           // 选择远端服务器端口号

#define WIFI_LINK_ID4_LOCAL_PORT      5004           // 选择本地口号

#endif


// 设置 WIFI LINK ID5 固定作为 UDP SERVER(TFTP)协议模式通信

#define WIFI_LINK_ID5_EN              0              // WIFI 连接通道 5 使能：1，使能； 0，关闭；

#if (WIFI_LINK_ID5_EN > 0)

#define WIFI_LINK_ID5_AUTO            1              // 通道 1 自动启动定义：1，自动连接；0，手动连接

#define WIFI_LINK_ID5_TYPE            WIFI_UDP_SERVER // 通道 1 通信模式定义：0, WIFI_TCP_CLIENT,
//通信类型 TCP 客户端； 1,  WIFI_TCP_SERVER, 通信类型 TCP 服务端;2, WIFI_UDP_CLIENT, 通信类型 UDP
//客户端； 3,  WIFI_UDP_SERVER, 通信类型 UDP 服务端；

#define WIFI_LINK_ID5_REMOTE_IP       ""             // 选择远端服务器 IP

#define WIFI_LINK_ID5_REMOTE_PORT     0              // 选择远端服务器端口号

#define WIFI_LINK_ID5_LOCAL_PORT      69             // 选择本地口号

#endif
```

(5) 发送和接收缓存设置

```
#define WIFI_TXBUF_EN                1 // 1: WIFI 发送缓存使能；0: WIFI 发送缓存关闭；注意在多通道通
```

// 信或者发送数据非常频繁必须使能这个发送缓存

```
#if (WIFI_TXBUF_EN > 0)

#define WIFI_TXBUF_NUM      4    // WIFI 发送缓存最大数量, 最大 32 个
#define WIFI_TXBUF_SIZE     256  // WIFI 发送缓存大小
#define WIFI_MAX_TX_TIMEOUT 200  // 最大超时 200ms
#endif

#define WIFI_RX_MALLOC_EN    0    // 1: 接收数据缓存动态申请; 0: 接收数据缓存静态定义;
#if (WIFI_RX_MALLOC_EN == 0)
#define WIFI_RXBUF_SIZE      512  // WIFI 接收缓存大小定义, 这个值必须大于等于 WIFI_UART 所配置
                                // 的 UART 接收缓存大小; 注意: WIFI_RX_MALLOC_EN 只有定义为 0 这个设置才有效
#endif
#endif
```

(6) 在../**device/wifi**/wifi_app.c 中查看 WIFI_AppInit 初始化函数, 这函数根据上面配置初始化, 用户一般不需要修改该函数, 具体代码用户自行查看。

(7) 在../**device/wifi**/wifi_app.c 中查看 WIFI_AppProc 处理函数:

```
void WIFI_AppProc(void)
{
    INT32U tick;

    tick = APP_GetSubTick(LibAppVars.WIFI.Scan_t); // 读取和上次扫描的间隔时间, 单位 ms
    if (tick >= WIFI_SCAN_T)                        // 计算和上次扫描时间间隔大于等于设置值
    {
        if (LibAppVars.WIFI.t_ms > tick)            // WIFI 操作内部定时变量操作
        {
            LibAppVars.WIFI.t_ms -= tick;
        }
        else
        {

```

```
        LibAppVars.WIFI.t_ms = 0;

    }

    LibAppVars.WIFI.Scan_t += tick;           // 记录本次扫描时间
    WIFI_SCANRxProc(&LibAppVars.WIFI);       // WIFI 扫描接收数据处理
    WIFI_Sync(&LibAppVars.WIFI);             // WIFI 状态同步操作
}

}
```

WIFI_Sync 这个函数对 WIFI 模块进行初始化、模式参数设置、状态监测等等操作。WIFI_SCANRxProc 这个函数接收 WIFI 数据并进行处理，有效数据通过 LibAppVars.Register.APP_InputDataCallback 回调函数发送出来。

(7) WIFI 接收数据被发送到../source/user_netwifiapp.c 中的 WIFI_DataProcess 函数中，具体函数请自行查看。

注意：NET_UserSendData 这个函数是 WIFI 发送数据函数；

WIFI 发送数据函数在../source/comfun.c 中的 NET_UserSendData，具体代码自行查看。用户需要通过 WIFI 发送数据请调用这个函数。

25. 外部器件(传感器)编程说明:

(1) 硬件设计经常会遇见在 MCU 外围设计一些外部器件(传感器)之类的, 在文件夹../driver 下我们一些常用外围器件驱动程序。我们后续会继续增加其它外部器件驱动程序。

(2) 在功能配置文件../device/device_config.h 中参考如下:

```
#include "ch455/ch455_config.h" // CH455 参数配置(4*7 按键驱动芯片)
#include "shtxx/shtxx_config.h" // 温湿度参数配置
#include "tm1640/tm1640_config.h" // TM1640 参数配置
#include "wtn6xxx/wtn6xxx_config.h" // WTN6XXX 参数配置
#include "dac8562/dac8562_config.h" // DAC8562 数配置
#include "ads1220/ads1220_config.h" // ADS1220 参数配置
#include "ads1232/ads1232_config.h" // ADS1232 参数配置
#include "ch9434/ch9434_config.h" // CH9434 参数配置
#include "max6675/max6675_config.h" // MAX6675 参数配置
#include "mcp4728/mcp4728_config.h" // MCP4728 参数配置
#include "dac128s085/dac128s085_config.h" // DAC128S085 参数配置
#include "ad5175/ad5175_config.h" // AD5175 参数配置
#include "by8302/by8302_config.h" // BY8302 参数配置
#include "mcp3208/mcp3208_config.h" // MCP3208 参数配置
#include "wifi/wifi_config.h" // WIFI 参数配置
#include "stepmotor/stepmotor_config.h" // STEP MOTOR 参数配置
#include "w5500/w5500_config.h" // W5500 参数配置
#include "dac8554/dac8554_config.h" // ADC8554 参数配置
#include "ad7682/ad7682_config.h" // AD7682 参数配置
#include "ft5xxx/ft5xxx_config.h" // FT5XXX 参数配置
#include "max31865/max31865_config.h" // MAX31865 参数配置

// 外部器件使能定义

#define DEVICE_EN    1    // DEVICE 使能, 1: 打开使能, 0: 关闭
```

这些是所有外置器件的参数配置文件, 具体要查看哪个, 可以打开相关配置文件。

(3) 外部器件初始化, 请打开../device/device_app.c, 查看 Device_Applnit 函数, 代码如下:

```
void Device_Applnit(void)
{
    #if (CH455_EN > 0)        // CH455 配置使能
    CH455_Applnit();          // CH455 应用初始化
    CH455_AppCtrl(CMD_CH455_APP_CTRL_CALLBACK, CH455_AppCallBack); // 注册数据回调函数
    #endif

    #if (SHTXX_EN > 0)        // SHTXX 温湿度传感器配置使能
    SHTXX_Applnit();          // SHTXX 温湿度传感器初始化
    SHTXX_AppCtrl(CMD_SHTXX_APP_CTRL_CALLBACK, SHTXX_AppCallBack); // 注册数据回调函数
    #endif

    #if (TM1640_EN > 0)        // TM1640 配置使能
    TM1640_Applnit();          // TM1640 初始化
    #endif

    #if (LCD_EN > 0)           // LCD 配置使能
    LCD_Applnit();             // LCD 初始化
    #endif

    #if (WTN6XXX_EN > 0)        // WTN6XXX 配置使能
    WTN6XXX_Applnit();          // WTN6XXX 初始化
    #endif

    #if (DAC8562_EN > 0)        // DAC8562 配置使能
    DAC8562_Applnit();          // DAC8562 初始化
    #endif

    #if (ADS1220_EN > 0)        // ADS1220 配置使能
    ADS1220_Applnit();          // ADS1220 初始化
    ADS1220_AppCtrl(CMD_ADS1220_APP_CTRL_CALLBACK, ADS1220_AppCallBack); // 注册数据回调函数
    #endif

    #if (ADS1232_EN > 0)        // ADS1232 配置使能
    ADS1232_Applnit();          // ADS1232 初始化
    ADS1232_AppCtrl(CMD_ADS1232_APP_CTRL_CALLBACK, ADS1232_AppCallBack); // 注册数据回调函数
```

```
#endif

#if (MAX6675_EN > 0)    // MAX6675 配置使能
MAX6675_AppInit();      // MAX6675 初始化
MAX6675_AppCtrl(CMD_MAX6675_APP_CTRL_CALLBACK, MAX6675_AppCallBack); // 注册数据回调函数
#endif

#if (CH9434_EN > 0)     // CH9434 配置使能
CH9434_AppInit();       // CH9434 初始化
CH9434_AppCtrl(CMD_CH9434_APP_CTRL_CALLBACK, CH9434_AppCallBack); // 注册数据回调函数
#endif

#if (MCP4728_EN > 0)    // MCP4728 配置使能
MCP4728_AppInit();     // MCP4728 初始化
#endif

#if (AD5175_EN > 0)     // AD5175 配置使能
AD5175_AppInit();      // AD5175 初始化
#endif

#if (BY8302_EN > 0)     // BY8302 配置使能
BY8302_AppInit();      // BY8302 初始化
#endif

#if (MCP3208_EN > 0)    // MCP3208 配置使能
MCP3208_AppInit();     // MCP3208 初始化
MCP3208_AppCtrl(CMD_MCP3208_APP_CTRL_CALLBACK, MCP3208_AppCallBack); // 注册数据回调函数
#endif

#if (WIFI_EN > 0)
WIFI_AppInit();        // 初始化 WIFI 模块
#endif

#if (STEP_MOTOR_EN > 0)
StepMotor_AppInit();   // 初始化 STEP_MOTOR 模块
#endif

#if (W5500_EN > 0)
W5500_AppInit();       // W5500 初始化
```

```

#endif

#if (DAC8554_EN > 0)    // DAC8554 配置使能
DAC8554_AppInit();      // DAC8554 初始化
#endif

#if (AD7682_EN > 0)     // AD7682 配置使能
AD7682_AppInit();      // AD7682 初始化
AD7682_AppCtrl(CMD_AD7682_APP_CTRL_CALLBACK, AD7682_AppCallback); // 注册数据回调函数
#endif

#if (FT5XXX_EN > 0)     // FT5XXX 配置使能
FT5XXX_AppInit();      // FT5XXX 应用初始化
FT5XXX_AppCtrl(CMD_FT5XXX_APP_CTRL_CALLBACK, FT5XXX_AppCallback); // 注册数据回调函数
#endif

#if (MAX31865_EN > 0)   // MAX31865 配置使能
MAX31865_AppInit();    // MAX31865 初始化
MAX31865_AppCtrl(CMD_MAX31865_APP_CTRL_CALLBACK, MAX31865_AppCallback); //注册数据回调函数
#endif
}

```

这个函数是所有器件初始化函数，器件 XXX_EN 只要在配置里使能就会初始化该器件。本函数由../source/TaskInputData.c 中的 APP_InputDataInit 函数调用。

(4) 外部器件操作处理，请打开../device/device_app.c，查看 Device_AppProc 函数，代码如下：

```

void Device_AppProc(void)
{
    #if (CH455_EN > 0)
    CH455_AppProc();      // CH455 应用处理
    #endif

    #if (SHTXX_EN > 0)
    SHTXX_AppProc();      // SHTXX 温湿度传感器应用处理
    #endif

    #if (TM1640_EN > 0)

```

```
TM1640_AppProc();      // TM1640 应用处理

#endif

#if (LCD_EN > 0)

LCD_AppProc();          // LCD 应用处理

#endif

#if (WTN6XXX_EN > 0)

WTN6XXX_AppProc();      // WTN6XXX 应用处理

#endif

#if (DAC8562_EN > 0)

DAC8562_AppProc();      // DAC8562 应用处理

#endif

#if (ADS1220_EN > 0)

ADS1220_AppProc();      // ADS1220 应用处理

#endif

#if (ADS1232_EN > 0)

ADS1232_AppProc();      // ADS1232 应用处理

#endif

#if (MAX6675_EN > 0)

MAX6675_AppProc();      // MAX6675 应用处理

#endif

#if (CH9434_EN > 0)      // CH9434 配置使能

CH9434_AppProc();        // CH9434 接收应用处理

#endif

#if (MCP4728_EN > 0)

MCP4728_AppProc();      // MCP4728 应用处理

#endif

#if (AD5175_EN > 0)

AD5175_AppProc();        // AD5175 应用处理

#endif

#if (BY8302_EN > 0)
```

```
BY8302_AppProc();      // BY8302 应用处理

#endif

#if (MCP3208_EN > 0)

MCP3208_AppProc();      // MCP3208 应用处理

#endif

#if (WIFI_EN > 0)

WIFI_AppProc();          // WIFI 应用处理

#endif

#if (STEP_MOTOR_EN > 0)

StepMotor_AppProc();     // STEP_MOTOR 模块处理

#endif

#if (W5500_EN > 0)

W5500_AppProc();         // W5500 应用处理

#endif

#if (DAC8554_EN > 0)

DAC8554_AppProc();       // DAC8554 应用处理

#endif

#if (AD7682_EN > 0)

AD7682_AppProc();        // AD7682 应用处理

#endif

#if (CH455_EN > 0)

CH455_AppProc();         // CH455 应用处理

#endif

#if (FT5XXX_EN > 0)      // FT5XXX 配置使能

FT5XXX_AppProc();       // FT5XXX 应用处理

#endif

#if (MAX31865_EN > 0)    // MAX31865 配置使能

MAX31865_AppProc();     // MAX31865 应用处理      // MAX31865 初始化

#endif

}
```

这个函数是所有器件应用函数，器件 XXX_EN 只要在配置里使能就会调用该函数处理。本函数由../source/TaskInputData.c 中的 APP_InputData 函数调用。

（5） 外部器件处理结果，请打开../source/user_deviceapp.c，查看各个器件执行操作结果的回调函数，用户可以在这里做进一步的应用处理。

26. AT 指令编程说明：

(1) 关键词定义如下： AT_EN

特别说明：AT 指令是我公司自定义编写的，旨在用户做一些调试配置测试之用。具体 AT 指令参见《AMKNxxx 系列工控板(模块)AT 指令_1.04_xxxxxx.pdf》。该指令通过调试串口(由 DEBUG_UART 定义，默认 UART1)进行控制。

(2) 在功能配置文件 AMKNXXX_Config.h 中定义，参考如下：

```
#define AT_EN          1          // AT 指令使能：0，关闭；1，使能；
```

(3) 在 ../source/user_uartapp.c 中的 Uart1_UserProcess 函数是处理 UART1 接收的数据

```
void Uart1_UserProcess(INT8U *p, INT16U len)
{
    INT16U i;
    #if (AT_EN > 0)                // 条件编译 AT 指令使能,
        AT_Proc(p, len);          // AT 指令处理函数
    #else                          // 没有使能 AT 指令，则按正常数据处理
        ... ..
    #endif
}
```

如果使能了 AT_EN 则首先会调用 AT_Proc 进行 AT 指令处理。

(4) 在 ../libapp/at.c 中用户可自行查看 AT 指令处理函数。

注意：用户不要修改 at.c 中的程序，如果发现 bug 请反馈给我们，由我司进行修改。

27. LPTIM 定时器编程说明:

(1) 关键词定义如下: LPTIMx_EN、LPTIMx_MODE、LPTIMx_T、LPTIMx_PSC

x 范围: 1~5; 注意: 定时器是芯片内部功能, 无管脚定义。

(2) 在功能配置文件 AMKNXXXX_Config.h 中定义, 以 LPTIM1 为例参考如下:

```
#define LPTIM1_EN      1 // LPTIM1 使能, 1: 打开使能, 0: 关闭

#define LPTIM1_MODE    0 // LPTIM 工作模式: 0, LPTIM_MODE_INT, 定时器工作在定时中断模式,
// 定时时间由参数 TIM1_T 设置; 1, LPTIM_MODE_COUNT, 定时器工作在定时计数模式, 外部通过调用
// LPTimer_Ctrl 函数参数 CMD_LPTIM_ENA 先启动定时器, 再通过参数 CMD_LPTIM_READ 读取计数值, 然后
// 通过参数 CMD_LPTIM_DIS 停止定时器; 2, LPTIM_MODE_PWM, PWM 输出模式 (OUT 输出);
// 3, LPTIM_MODE_FCLK_COUNT, 脉冲输入计数 (IN1 输入); 4, LPTIM_MODE_FCLK_DECOUNT, 正交编码器脉
// 冲输入计数 (IN1, IN2 输入); 5, LPTIM_MODE_OUT, 定时器输出工作模式
```

初始定时时间设定

```
#define LPTIM1_CLK      LPTIM_LSE_32768HZ // 定时器时钟源选择

#define LPTIM1_T        1000000 // 定时器定时时间, 单位 us

#define LPTIM1_PSC      LPTIM_PRESCALER_DIV1 分频系数: LPTIM_PRESCALER_DIV1
//LPTIM_PRESCALER_DIV2/LPTIM_PRESCALER_DIV4/LPTIM_PRESCALER_DIV8
```

(3) 在 ../libapp/lptim_app.c 中查看 LPTIM_ApplInit 初始化函数, 这函数根据上面配置初始化 LPTIM, 用户一般不需要修改该函数, 具体代码用户自行查看。

(4) 在功能配置文件 AMKNXXXX_Config.h 中做如下定义:

```
#define LPTIM1_EN      1 // LPTIM1 使能, 1: 打开使能, 0: 关闭

#define LPTIM1_MODE    0 // LPTIM1 工作模式: 定时中断模式

#define LPTIM1_T        1000000 // LPTIM1 定时器定时时间, 单位 us
```

则间隔 1 秒会产生一次中断, 中断函数在 ../source/ISRHooks.c 中的 LPTIM1_ISRHook() 函数: 在这个函数中用户自行编写中断处理程序, 但要尽可能短, 缩短执行时间。

28. SDRAM 编程说明:

(1) 关键词定义如下: SDRAM_EN、SDRAM_ADDR、SDRAM_SIZE、SDRAM_LCD_START_ADDR

(2) 在功能配置文件 AMKNXXX_Config.h 中定义:

```
#define SDRAM_EN          1                // 1, 使能; 0, 关闭;  
  
#define SDRAM_LCD_START_ADDR 0xC1C00000 // LCD SDRAM 起始地址(从 28MB 地址开始)
```

在 sdr.h 中定义:

```
#define SDRAM_ADDR        ((INT32U) 0xC0000000) // SDRAM 起始地址, 由硬件决定不可更改  
  
#define SDRAM_SIZE        ((INT32U) 32*1024*1024) // SDRAM 大小, 由硬件决定不可更改
```

(3) 在 ../libapp/sdr_app.c 中查看 SDRAM_AppInit 初始化函数, 这函数初始化 SDRAM, 用户一般不需要修改该函数, 具体代码用户自行查看。

(4) 在测试配置文件 ../config/test_config.h 中做如下定义:

```
#define APP_SDRAM_TEST_EN    1 // SDRAM 测试使能: 1, 使能; 0, 关闭
```

在 ../source/user_testapp.c 中的函数 User_AppTestInit 调用 SDRAM_AppTest 进行读写测试, 判断 SDRAM 的工作状态。

用户可以参考这个测试编写自己的读写程序。

29. DMA2D 编程说明:

(1) 关键词定义如下: DMA2D_EN、DMA2D_INT_EN、DMA2D_NO_BLOCK_EN、DMA2D_OUT_COLOR_FORMAT、DMA2D_OUT_ALPHA_INVERTED_EN、DMA2D_OUT_RB_SWAP_EN、DMA2D_FG_COLOR_FORMAT、DMA2D_FG_ALPHA_MODE、DMA2D_FG_ALPHA_INVERTED_EN、DMA2D_FG_RB_SWAP_EN、DMA2D_FG_YCbCr_CSS、DMA2D_FG_CLUT_SIZE、DMA2D_FG_CLUT_COLOR_MODE、DMA2D_BG_COLOR_FORMAT、DMA2D_BG_ALPHA_MODE、DMA2D_BG_RB_SWAP_EN、DMA2D_BG_ALPHA_INVERTED_EN、DMA2D_BG_YCbCr_CSS、DMA2D_BG_CLUT_SIZE、DMA2D_BG_CLUT_COLOR_MODE

(2) 在功能配置文件 AMKNXXX_Config.h 中定义:

```
#define DMA2D_EN          1          // 1, 使能; 0, 关闭;

#if (DMA2D_EN > 0)

#define DMA2D_TIMEOUT    100        // 设置操作超时: 单位 ms;

#define DMA2D_INT_EN     0          // 设置中断使能: 0x3F(DMA2D_INT_MASK), 使能全部中断; 0, 关闭所有中断;

#define DMA2D_NO_BLOCK_EN 0          // 设置函数操作无阻塞: 1, 使能; 0, 关闭

// 输出区参数设置

#define DMA2D_OUT_COLOR_FORMAT 2 //设置输出颜色格式: 0(DMA2D_FORMAT_ARGB8888); 1(DMA2D_
//FORMAT_RGB888); 2(DMA2D_FORMAT_RGB565); 3(DMA2D_FORMAT_ARGB1555); 4(DMA2D_FORMAT_ARGB4444);
#define DMA2D_OUT_ALPHA_INVERTED_EN 0 // 设置输出 ALPHA 取反使能: 0(DMA2D_ALPHA_REGULAR),
// 正常输出; 1(DMA2D_ALPHA_INVERTED), 反转输出;
#define DMA2D_OUT_RB_SWAP_EN 0 // 设置输出颜色格式中 R 通道和 B 通道的交换使能:
// 0(DMA2D_REGULAR_RB), 不交换(RGB or ARGB); 1(DMA2D_SWAP_RB), 交换(BGR or ABGR);

// 前景层(输入)参数设置

#define DMA2D_FG_COLOR_FORMAT 2 // 设置前景层颜色格式: 0(DMA2D_FORMAT_ARGB8888); 1(DMA2D_
// FORMAT_RGB888); 2(DMA2D_FORMAT_RGB565); 3(DMA2D_FORMAT_ARGB1555); 4(DMA2D_FORMAT_ARGB4444);
// 5(DMA2D_FORMAT_L8); 6(DMA2D_FORMAT_AL44); 7(DMA2D_FORMAT_AL88); 8(DMA2D_FORMAT_L4);
// 9(DMA2D_FORMAT_A8); 10(DMA2D_FORMAT_A4); 11(DMA2D_FORMAT_YCBCR);
#define DMA2D_FG_ALPHA_MODE 0 // 设置前景层 Alpha 模式: 0(DMA2D_ALPHA_MODE_NONE), 不修改前景
// 或背景图像的 alpha 通道值; 1(DMA2D_ALPHA_MODE_REPLACE), 原始前景或背景图像的 alpha 通道值替换
// 为 ALPHA[7:0]; 2(DMA2D_ALPHA_MODE_MIX), 原始前景或背景图像的 alpha 通道值替换为 ALPHA[7:0]与
// 原始 alpha 通道值的乘积;
#define DMA2D_FG_ALPHA_INVERTED_EN 0 // 设置前景层 ALPHA 取反使能: 0(DMA2D_ALPHA_REGULAR),
```

//正常输出; 1 (DMA2D_ALPHA_INVERTED), 反转输出;

```
#define DMA2D_FG_RB_SWAP_EN    0 // 设置前景层颜色格式中 R 通道和 B 通道的交换使能:
// 0 (DMA2D_REGULAR_RB), 不交换 (RGB or ARGB); 1 (DMA2D_SWAP_RB), 交换 (BGR or ABGR);

#define DMA2D_FG_YCbCr_CSS    0 // 设置前景层颜色 YCbCr 颜色模式的采样格式:
// 0 (DMA2D_YCbCr_CSS_444), 4:4:4; 1 (DMA2D_YCbCr_CSS_422), 4:2:2; 2 (DMA2D_YCbCr_CSS_420), 4:2:0;

#define DMA2D_FG_CLUT_SIZE      256 // 设置前景颜色 CLUT 大小: 1-256

#define DMA2D_FG_CLUT_COLOR_MODE    0 // 设置前景色中 CLUT 颜色模式: 0 (DMA2D_CCM_ARGB8888):
// ARGB8888; 1 (DMA2D_CCM_RGB888): RGB888;

// 背景层(输入)参数设置

#define DMA2D_BG_COLOR_FORMAT    2 // 设置背景层颜色格式: 0 (DMA2D_FORMAT_ARGB8888);
// 1 (DMA2D_FORMAT_RGB888); 2 (DMA2D_FORMAT_RGB565); 3 (DMA2D_FORMAT_ARGB1555);
// 4 (DMA2D_FORMAT_ARGB4444); 5 (DMA2D_FORMAT_L8); 6 (DMA2D_FORMAT_AL44); 7 (DMA2D_FORMAT_AL88);
// 8 (DMA2D_FORMAT_L4); 9 (DMA2D_FORMAT_A8); 10 (DMA2D_FORMAT_A4); 11 (DMA2D_FORMAT_YCBCR);

#define DMA2D_BG_ALPHA_MODE    0 // 设置背景层 Alpha 模式: 0 (DMA2D_ALPHA_MODE_NONE),
// 不修改前景或背景图像的 alpha 通道值; 1 (DMA2D_ALPHA_MODE_REPLACE), 原始前景或背景图像的 alpha
// 通道值替换为 ALPHA[7:0]; 2 (DMA2D_ALPHA_MODE_MIX), 原始前景或背景图像的 alpha 通道值替换为
// ALPHA[7:0]与原始 alpha 通道值的乘积;

#define DMA2D_BG_ALPHA_INVERTED_EN    0 // 设置背景层 ALPHA 取反使能: 0 (DMA2D_ALPHA_REGULAR),
// 正常输出; 1 (DMA2D_ALPHA_INVERTED), 反转输出;

#define DMA2D_BG_RB_SWAP_EN      0 // 设置背景层颜色格式中 R 通道和 B 通道的交换使能:
// 0 (DMA2D_REGULAR_RB), 不交换 (RGB or ARGB); 1 (DMA2D_SWAP_RB), 交换 (BGR or ABGR);

#define DMA2D_BG_YCbCr_CSS      0 // 设置背景层颜色 YCbCr 颜色模式的采样格式:
// 0 (DMA2D_YCbCr_CSS_444), 4:4:4; 1 (DMA2D_YCbCr_CSS_422), 4:2:2; 2 (DMA2D_YCbCr_CSS_420), 4:2:0;

#define DMA2D_BG_CLUT_SIZE      256 // 设置背景颜色 CLUT 大小: 1-256

#define DMA2D_BG_CLUT_COLOR_MODE    0 // 设置背景色中 CLUT 颜色模式: 0 (DMA2D_CCM_ARGB8888):
// ARGB8888; 1 (DMA2D_CCM_RGB888): RGB888;

#endif
```

(3) 在../libapp/dma2d_app.c 中查看 DMA2D_AppInit 初始化函数, 根据上面配置初始化 DMA2D, 用户一般不需要修改该函数, 具体代码用户自行查看。

(4) 在../source/TaskLCD.c 中的函数 DispBMP 进行操作测试，用户可以参考这个测试编写自己的读写程序。

30. LTDC 编程说明:

(1) 关键词定义如下: LTDC_EN、LCD_PRODUCT、LCD_DISP_TYPE、LCD_WIDTH、LCD_HEIGHT、LCD_FCLK、LCD_HSYNC_WIDTH、LCD_HBP_WIDTH、LCD_HFP_WIDTH、LCD_VSYNC_WIDTH、LCD_VBP_WIDTH、LCD_VFP_WIDTH、LCD_HSYNC_POLARITY、LCD_VSYNC_POLARITY、LCD_DE_POLARITY、LCD_CLK_POLARITY、LCD_BACKCOLOR_RED、LCD_BACKCOLOR_GREEN、LCD_BACKCOLOR_BLUE、LTDC_BUFFER_SIZE、LTDC_LAYER1_EN、LTDC_L1_WINDOW_X0、LTDC_L1_WINDOW_X1、LTDC_L1_WINDOW_Y0、LTDC_L1_WINDOW_Y1、LTDC_L1_PIXEL_FORMAT、LTDC_L1_ALPHA、LTDC_L1_BF1、LTDC_L1_BF2、LTDC_L1_IMAGE_WIDTH、LTDC_L1_IMAGE_HEIGHT、LTDC_L1_START_ADDRESS、LTDC_L1_ALPHA0、LTDC_L1_BACKCOLOR_RED、LTDC_L1_BACKCOLOR_GREEN、LTDC_L1_BACKCOLOR_BLUE、LTDC_LAYER2_EN、LTDC_L2_WINDOW_X0、LTDC_L2_WINDOW_X1、LTDC_L2_WINDOW_Y0、LTDC_L2_WINDOW_Y1、LTDC_L2_PIXEL_FORMAT、LTDC_L2_ALPHA、LTDC_L2_BF1、LTDC_L2_BF2、LTDC_L2_IMAGE_WIDTH、LTDC_L2_IMAGE_HEIGHT、LTDC_L2_START_ADDRESS、LTDC_L2_ALPHA0、LTDC_L2_BACKCOLOR_RED、LTDC_L2_BACKCOLOR_GREEN、LTDC_L2_BACKCOLOR_BLUE

(2) 在 IO 配置文件 AMKNXXX_IOConfig.h 中定义: LTDC 的端口定义如下:

```
// 控制信号

#define LCD_CLK          PI14

#define LCD_HSYNC        PI12

#define LCD_VSYNC        PI13

#define LCD_DE           PK7

// RED

#define LCD_R0           IO_NONE // PI15

#define LCD_R1           IO_NONE // PJ0

#define LCD_R2           IO_NONE // PJ1

#define LCD_R3           PJ2

#define LCD_R4           PJ3

#define LCD_R5           PJ4

#define LCD_R6           PJ5

#define LCD_R7           PJ6

// GREEN

#define LCD_G0           IO_NONE // PJ7
```

```
#define LCD_G1      IO_NONE  // PJ8
#define LCD_G2      PI15     // PJ9
#define LCD_G3      PJ10
#define LCD_G4      PJ11
#define LCD_G5      PK0
#define LCD_G6      PK1
#define LCD_G7      PK2

// BLUE

#define LCD_B0      IO_NONE  // PJ12
#define LCD_B1      IO_NONE  // PJ13
#define LCD_B2      IO_NONE  // PJ14
#define LCD_B3      PJ15
#define LCD_B4      PK3
#define LCD_B5      PK4
#define LCD_B6      PK5
#define LCD_B7      PK6

#define LCD_RESET   IO_NONE

#define LCD_BL      PF6
```

(3) 在功能配置文件 AMKNXXX_Config.h 中定义：

```
#define LTDC_EN      1        // 1, 使能; 0, 关闭;

#if (LTDC_EN > 0)

#define LCD_PRODUCT  LCD_ATK_MD0700R_800X480  // LCD 型号定义: 0, LCD_ATK_MD0700R_800X480;
                                              // 1, LCD_ATK_MD0700R_1024X600

// LCD 基本参数配置

#define LCD_DISP_TYPE 0 // LCD 显示方式: 0 (LCD_TYPE_HS), 横屏; 1 (LCD_TYPE_HS180), 横屏 180;
                        // 2 (LCD_TYPE_VS), 竖屏; 3 (LCD_TYPE_VS180), 竖屏 180;

#if (LCD_PRODUCT == LCD_ATK_MD0700R_800X480)

#define LCD_WIDTH      800    // LCD 分辨率宽
```

```
#define LCD_HEIGHT 480 // LCD 分辨率高

#define LCD_FCLK LTDC_CLK_32MHZ //LCD 时钟频率: LTDC_CLK_32MHZ/LTDC_CLK_48MHZ/LTDC_CLK_64MHZ
//LTDC_CLK_12MHZ, 一般分辨率和时钟频率关系: 800*480->LTDC_CLK_32MHZ;
//1024*600->LTDC_CLK_48MHZ; 480*272->LTDC_CLK_12MHZ;

#define LCD_HSYNC_WIDTH 20 // HSYNC 信号宽度
#define LCD_HBP_WIDTH 46 // HBP 信号宽度
#define LCD_HFP_WIDTH 210 // HFP 信号宽度

#define LCD_VSYNC_WIDTH 10 // VSYNC 信号宽度
#define LCD_VBP_WIDTH 23 // VBP 信号宽度
#define LCD_VFP_WIDTH 22 // VFP 信号宽度

#endif

// 配置信号极性
#define LCD_HSYNC_POLARITY 0 // HSYNC 极性配置: 0(低电平)或 1(高电平)
#define LCD_VSYNC_POLARITY 0 // VSYNC 极性配置: 0(低电平)或 1(高电平)
#define LCD_DE_POLARITY 0 // DE 极性配置: 0(低电平)或 1(高电平)
#define LCD_CLK_POLARITY 0 // Pixel Clock 极性配置: 0(低电平)或 1(高电平)

// 配置背景层颜色
#define LCD_BACKCOLOR_RED 0
#define LCD_BACKCOLOR_GREEN 0
#define LCD_BACKCOLOR_BLUE 100

// 配置每个图层缓存大小
#define LTDC_BUFFER_SIZE 0x00200000

// 图层 1 配置
#define LTDC_LAYER1_EN 1 //LTDC 图层 1 使能配置: 1, 使能; 0, 关闭;

#if (LTDC_LAYER1_EN > 0)
#define LTDC_L1_WINDOW_X0 0 // 设置图层水平起始位置, 范围 0x000 到 0xFFFF。
#define LTDC_L1_WINDOW_X1 LCD_WIDTH // 设置图层水平结束位置, 范围 0x000 到 0xFFFF。
#define LTDC_L1_WINDOW_Y0 0 // 设置图层垂直起始位置, 范围 0x000 到 0x7FF。
#define LTDC_L1_WINDOW_Y1 LCD_HEIGHT // 设置图层垂直结束位置, 范围 0x000 到 0x7FF。
```

```
#define LTDC_L1_PIXEL_FORMAT LTDC_PIXEL_FORMAT_RGB565 // 设置图层所使用的颜色格式

#define LTDC_L1_ALPHA      0xFF      // Alpha 常数, 范围 0x00 - 0xFF。

// 配置图层混合因数

#define LTDC_L1_BF1      LTDC_BF1_PAxCA // 设置混合因数 1: LTDC_BF1_CA 或 LTDC_BF1_PAxCA
#define LTDC_L1_BF2      LTDC_BF2_PAxCA // 设置混合因数 2: LTDC_BF2_CA 或 LTDC_BF2_PAxCA


#define LTDC_L1_IMAGE_WIDTH LCD_WIDTH // 设置颜色帧缓冲区行长, 即要从显存读取一行的长度,
                                        // 范围 0x0000 到 0x1FFF。

#define LTDC_L1_IMAGE_HEIGHT LCD_HEIGHT // 设置颜色帧缓冲区行数, 即要从显存读取的行数,
                                        // 范围 0x000 到 0xFFF。

#define LTDC_L1_START_ADDRESS (SDRAM_LCD_START_ADDR) // 图层的显存地址

// 设置图层默认色

#define LTDC_L1_ALPHA0      0      // 设置图层默认 Alpha 值, 范围 0x00 - 0xFF, 与结构体成员
                                        // Backcolor 一起使用, 组成 ARGB 颜色格式用于图层背景色

#define LTDC_L1_BACKCOLOR_RED 255
#define LTDC_L1_BACKCOLOR_GREEN 0
#define LTDC_L1_BACKCOLOR_BLUE 0

#endif
```

(4) 在../libapp/ltdc_app.c 中查看 LTDC_AppInit 初始化函数, 这函数初始化 LTDC, 用户一般不需要修改该函数, 具体代码用户自行查看。

(5) 在../source/TaskLCD.c 中的函数 User_LCDProcess 进行 LCD 操作测试, 用户可以参考这个测试编写自己的读写程序。

31. MDMA 编程说明:

(1) 关键词定义如下: MDMA_EN、MDMA_TEST_EN、MDMA_CHx_EN、MDMA_CHx_REQUEST、MDMA_CHx_TRANSFER_MODE、MDMA_CHx_PRIORITY、MDMA_CHx_DATAALIGN、MDMA_CHx_LITTLE_ENDIANNESS、MDMA_CHx_BUFFER_TRANSFER_LEN、MDMA_CHx_SRC_DINC、MDMA_CHx_SRC_DATASIZE、MDMA_CHx_SRC_BURST、MDMA_CHx_DEST_DINC、MDMA_CHx_DEST_DATASIZE、MDMA_CHx_DEST_BURST

其中: x 范围: 1~16

(2) 在功能配置文件../config/mdma_config.h 中定义 (以 MDMA CH9 为例):

```
#define MDMA_EN            1           // MDMA 使能: 1, 使能; 0, 关闭;

#define MDMA_TEST_EN       0           // MDMA 测试使能: 1, 使能; 0, 关闭;

// MDMA CH9 配置: 固定用于 JPEG 编解码数据输入

#define MDMA_CH9_EN        1           // MDMA CH8 使能: 1, 使能; 0, 关闭;

#if (MDMA_CH9_EN > 0)

#define MDMA_CH9_REQUEST   MDMA_REQ_JPEG_INFIFO_TH // 触发 MDMA 的请求源配置: 范围: 0~30,
                                                    // 具体参考 mdma.h 中的定义, JPEG 的 FIFO 阈值触发

#define MDMA_CH9_TRANSFER_MODE MDMA_BUFFER_TRANSFER // MDMA 的传输模式: 0 (MDMA_BUFFER
// _TRANSFER): 缓冲传输; 1 (MDMA_BLOCK_TRANSFE): 块传输; 2 (MDMA_MUL_BLOCK_TRANSFER): 多块传输;
// 3 (MDMA_FULL_TRANSFER): 链表传输;

#define MDMA_CH9_PRIORITY   MDMA_PRIORITY_HIGH // MDMA 通道的优先级设置: 0 (MDMA_PRIORITY_LOW):
低; 1 (MDMA_PRIORITY_MEDIUM): 中; 2 (MDMA_PRIORITY_HIGH): 高; 3 (MDMA_PRIORITY_VERY_HIGH): 非常高;

#define MDMA_CH9_DATAALIGN MDMA_DATAALIGN_PACKENABLE // MDMA 填充/对齐设置: 0 (MDMA_DATAALIGN
_RIGHT): 右对齐, 用 0 进行填充 (默认值); 1 (MDMA_DATAALIGN_RIGHT_SIGNED): 右对齐, 符号扩展;
2 (MDMA_DATAALIGN_LEFT): 左对齐 (用 0 行填充); 3 (MDMA_DATAALIGN_PACKENABLE): 将源数据封装/解//
封为目标数据大小。所有数据均右对齐 (采用小端模式)

#define MDMA_CH9_LITTLE_ENDIANNESS 0 // MDMA 数据传输格式: 0 (MDMA_LITTLE_ENDIANNESS_PRE):
正常的小端格式; 1 (MDMA_LITTLE_ENDIANNESS_BEX): 小端格式, 以半字为单位, 交互高低字节;
2 (MDMA_LITTLE_ENDIANNESS_HEX): 小端格式, 以字为单位, 交互半字; 4 (MDMA_LITTLE_ENDIANNESS_WEX):
```

小端格式, 以双字为单位, 交换字;

#define MDMA_CH9_BUFFER_TRANSFER_LEN 32 // 单次传输长度设置, 范围 1-128, 单位字节。每次传输 32 个字节, JPEG 的 FIFO 阈值是 32 字节

#define MDMA_CH9_SRC_DINC MDMA_INC_BYTE // MDMA 源数据增量/减量偏移量: 0 (MDMA_INC_DISABLE), 无增量, 地址指针固定; 8 (MDMA_INC_BYTE):, 单字节, 每次数据传输后, 源地址指针递增 1 个字节; 9 (MDMA_INC_HWORD): 半字即 2 字节, 每次数据传输后, 源地址指针递增 2 个字节; 10 (MDMA_INC_WORD): 字即 4 字节, 每次数据传输后, 源地址指针递增 4 个字节; 11 (MDMA_INC_DWORD): 双字即 8 字节, 每次数据传输后, 源地址指针递增 8 个字节; 12 (MDMA_DEC_BYTE): 单字节, 每次数据传输后, 源地址指针递减 1 个字节; 13 (MDMA_DEC_HWORD) 半字即 2 字节, 每次数据传输后, 源地址指针递减 2 个字节; 14 (MDMA_DEC_WORD): 字即 4 字节, 每次数据传输后, 源地址指针递减 4 个字节; 15 (MDMA_DEC_DWORD) 双字即 8 字节, 每次数据传输后, 源地址指针递减 8 个字节;

#define MDMA_CH9_SRC_DATASIZE MDMA_DATASIZE_BYTE // MDMA 源数据带宽: 0 (MDMA_DATASIZE_BYTE): 单字节; 1 (MDMA_DATASIZE_HWORD): 半字即 2 个字节; 2 (MDMA_DATASIZE_WORD): 字即 4 字节; 3 (MDMA_DATASIZE_DWORD): 双字即 8 个字节;

#define MDMA_CH9_SRC_BURST MDMA_BURST_32BEATS // 源数据端突发设置: 特别注意突发传输的数量不能超过 BufferTransferLength 参数设置的大小, 具体支持的参数如下: 0 (MDMA_BURST_SINGLE): 单次传输; 1 (MDMA_BURST_2BEATS): 2 个节拍突发; 2 (MDMA_BURST_4BEATS): 4 个节拍突发; 3 (MDMA_BURST_8BEATS): 8 个节拍突发; 4 (MDMA_BURST_16BEATS): 16 个节拍突发; 5 (MDMA_BURST_32BEATS): 32 个节拍突发; 6 (MDMA_BURST_64BEATS): 64 个节拍突发; 7 (MDMA_BURST_128BEATS): 128 个节拍突发;

#define MDMA_CH9_DEST_DINC MDMA_INC_DISABLE // MDMA 目标数据增量/减量偏移量: 0 (MDMA_INC_DISABLE), 无增量, 地址指针固定; 8 (MDMA_INC_BYTE):, 单字节, 每次数据传输后, 目标地址指针递增 1 个字节; 9 (MDMA_INC_HWORD): 半字即 2 字节, 每次数据传输后, 目标地址指针递增 2 个字节; 10 (MDMA_INC_WORD): 字即 4 字节, 每次数据传输后, 目标地址指针递增 4 个字节; 11 (MDMA_INC_DWORD): 双字即 8 字节, 每次数据传输后, 目标地址指针递增 8 个字节; 12 (MDMA_DEC_BYTE): 单字节, 每次数据传输后, 目标地址指针递减 1 个字节; 13 (MDMA_DEC_HWORD) 半字即 2 字节, 每次数据传输后, 目标地址指针递减

2 个字节；14(MDMA_DEC_WORD)：字即 4 字节，每次数据传输后，目标地址指针递减 4 个字节；
15(MDMA_DEC_DWORD) 双字即 8 字节，每次数据传输后，目标地址指针递减 8 个字节；

```
#define MDMA_CH9_DEST_DATASIZE      MDMA_DATASIZE_WORD    // MDMA 目标数据带宽：  
0(MDMA_DATASIZE_BYTE)：单字节；1(MDMA_DATASIZE_HWORD)：半字即 2 个字节；2(MDMA_DATASIZE_WORD)：  
字即 4 字节；3(MDMA_DATASIZE_DWORD)：双字即 8 个字节；
```

```
#define MDMA_CH9_DEST_BURST          MDMA_BURST_16BEATS    // 目标数据端突发设置：特别注意突发传输  
的数量不能超过 BufferTransferLength 参数设置的大小，具体支持的参数如下：0(MDMA_BURST_SINGLE)：  
单次传输；1(MDMA_BURST_2BEATS)：2 个节拍突发；2(MDMA_BURST_4BEATS)，4 个节拍突发；  
3(MDMA_BURST_8BEATS)：8 个节拍突发；4(MDMA_BURST_16BEATS)：16 个节拍突发；5(MDMA_BURST_32BEATS)：  
32 个节拍突发；6(MDMA_BURST_64BEATS)：64 个节拍突发；7(MDMA_BURST_128BEATS)：128 个节拍突发；  
#endif
```

(3) 在../libapp/mdma_app.c 中查看 MDMA_AppInit 初始化函数，根据上面配置初始化 MDMA，用户一般不需要修改该函数，具体代码用户自行查看。

(4) 在测试配置文件../config/mdma_config.h 中做如下定义：

```
#define MDMA_TEST_EN      1    // MDMA 测试使能：1，使能；0，关闭；  
  
#define MDMA_CH2_EN      1    // MDMA CH2 使能：1，使能；0，关闭；
```

在../source/usser_testapp.c 中的函数 User_AppTestInit 调用 MDMA_AppTest，用于测试 MDMA 传输，通过打印输出观察传输性能。用户可以参考此程序编写自己的 MDMA 传输数据。

32. JPEG 编程说明:

(1) 关键词定义如下: JPEG_EN

(2) 在功能配置文件 AMKNxxxx_Config.h 中定义:

```
#define JPEG_EN          1          // JPEG 使能: 1, 使能; 0, 关闭;  
  
#if (JPEG_EN > 0)  
  
#endif
```

注意: JPEG 编解码数据输入使用 MDMA_CH9, 必须使能 MDMA_CH9: #define MDMA_CH9_EN 1

JPEG 编解码数据输出使用 MDMA_CH10, 必须使能 MDMA_CH10: #define MDMA_CH10_EN 1

具体配置请查看文件 ../config/mdma_config.h

(3) 在 ../libapp/jpeg_app.c 中查看 JPEG_AppInit 初始化函数, 根据上面配置初始化 JPEG, 用户一般不需要修改该函数, 具体代码用户自行查看。

(4) 在 ../source/TaskLCD.c 中的函数 JPEG_AppDecode 进行 jpeg 文件解码并在 LCD 上显示测试, 用户可以参考这个测试编写自己的程序。

33. QSPI 编程说明：利用 QSPI 接口读写 W25Q64

(1) 关键词定义如下:QSPI_EN、QSPI_DEVICE、QSPI_DIVCLK、QSPI_IOMODE、QSPI_CKMODE、QSPI_FLASH_SIZE

(2) 在 IO 配置文件 AMKNXXX_IOConfig.h 中定义：QSPI 的端口定义如下：

// QSPI 管脚定义

```
#define QSPI_BK1_CS      PG6
#define QSPI_BK1_CLK     PF10
#define QSPI_BK1_I00     PF8
#define QSPI_BK1_I01     PF9
#define QSPI_BK1_I02     IO_NONE
#define QSPI_BK1_I03     IO_NONE
```

(3) 在功能配置文件 AMKNxxxx_Config.h 中定义：

```
#define QSPI_EN          1          // QSPI 使能,      1: 打开使能,  0: 关闭
#if (QSPI_EN > 0)
#define QSPI_DEVICE      GD25Q64    // 器件选择
#define QSPI_DIVCLK      QSPI_DIVCLK_4 // QSPI 时钟分频系数, Modify 2025.10.12 改为 4 分频比较稳定
#define QSPI_IOMODE      QSPI_IOMODE_2_LINES // IO 线数定义: 固定是这个不可改变
#define QSPI_CKMODE      QSPI_CLK_MODE0 // CLK 模式: 模式 0: QSPI_CKMODE0; 模式 3: QSPI_CKMODE3
#define QSPI_FLASH_SIZE  QSPI_FLASH_SIZE_8MB // Flash 大小定义; 固定 8MB
#endif
```

(4) 在 ../libapp/qspi_app.c 中查看 QSPI_ApplInit 初始化函数，根据上面配置初始化 QSPI，用户一般不需要修改该函数，具体代码用户自行查看。

(5) 在功能配置文件 AMKNXXX_Config.h 中定义，参考如下：

```
#define SPIFLASH_EN      1          // SPI FLASH 使能: 1, 使能;  0, 关闭;
#define SPIFLASH_MODE     0          // SPI FLASH 操作方式: 1, 利用 FATFS 文件系统进行读写;
                                   // 0, 用 SPI FLASH 读写函数进行操作; 注意: 2 种操作方式只能选择一种
#define SPIFLASH_TYPE     W25QXX    // SPI FLASH 类型设置: 必须是 W25QXX
```

// 注意：因为 W25QXX 是按扇区擦除，所以建立文件系统就按扇区计算

```
#if (SPIFLASH_TYPE == W25QXX)

#define W25QXX_MODE    W25QXX_MODE_QSPI // 读写模式：0 (W25QXX_MODE_SPI)，应用 SPI 总
                                     // 线来读写； 1 (W25QXX_MODE_QSPI)，应用 QSPI 总线来读写；

#if (W25QXX_MODE == W25QXX_MODE_SPI)

#define W25QXX_SPI_ID  SPI1_ID

#endif

#endif
```

(6) 在 ../config/test_config.h 中做如下定义：

```
#define APP_SPIFLASH_TEST_EN    1          // SPIFLASH 测试使能：1，使能；0，关闭

// W25QXX (SPI FLASH) 读写测试配置

// 注意：读写按页 (256 字节)，但擦除按扇区 (4096 字节) 擦除，每个扇区包含 16 个页

// 读写测试全部在扇区 W25QXX_ZDY_STARTSECTOR 里进行

#if (SPIFLASH_TYPE == W25QXX)

// 按页读写配置

#define W25QXX_TEST_START_SECTOR  W25QXX_FATFS_SECTORNUM // W25QXX FLASH 测试起始扇区

#define W25QXX_TEST_SECTOR_NUM    4          // W25QXX FLASH 测试扇区个数

#define W25QXX_TEST_START_PAGE    (W25QXX_TEST_START_SECTOR*16) // W25QXX 读写起始页

#define W25QXX_TEST_PAGE_NUM      (W25QXX_TEST_SECTOR_NUM*16)    // W25QXX 读写数据页数，
                                     // 按字节随机地址读写配置，W25QXX FLASH 读写起始地址

#define W25QXX_TEST_START_ADDR    (W25QXX_TEST_START_PAGE*W25QXX_PAGE_SIZE+100)

#define W25QXX_TEST_LEN           256        // W25QXX FLASH 读写数据长度 (256 字节)，小于 512 字节

#endif
```

(7) 在 ../source/user_testapp.c 中的 User_AppTestInit() 中会调用 W25QXX_AppTest() 函数。根据上面配置的读写起始地址和长度进行读写测试。具体测试程序参见 ../libapp/spiflash_app.c 中 W25QXX_AppTest() 函数。

34. LVGL 编程说明

(1) 关键词定义如下:LVGL_EN、LVGL_SCAN_T、LVGL_BUFFER_MODE、LVGL_BUFFER_LINE、LVGL_DEMO_WIDGETS_EN、LVGL_DEMO_BENCHMARK_EN、LVGL_DEBUG_EN

(2) 在功能配置文件 `./middlewares_app/lvgl_app/lvgl_Config.h` 中定义:

```
#define LVGL_EN                1    // LVGL 使能: 1, 使能; 0, 关闭;

#if (LVGL_EN > 0)

#define LVGL_SCAN_T            10    // 设置定时扫描时间间隔, 单位: ms

#define LVGL_BUFFER_MODE       1     // LVGL 缓存模式设置: 1, 单缓存模式; 2, 双缓存模式;

#define LVGL_BUFFER_LINE       20    // LVGL 缓存行数设置: 最小是 10, 设置越大性能越好, 但占用
// 内存越大, 占用内存大小计算公式(按 RGB565 计算): LCD 宽边分辨率
// *LVGL_BUFFER_LINE*LVGL_BUFFER_MODE*2; 例如 LCD 是 1024*600 则, 内存大小=1024*20*2*2=81920 字节
// 下面 2 个 demo 只能选择一个使能

#define LVGL_DEMO_WIDGETS_EN    1    // demo widgets 使能: 1, 使能; 0, 关闭;

#define LVGL_DEMO_BENCHMARK_EN 0     // demo benchmark 使能: 1, 使能; 0, 关闭;

#define LVGL_DEBUG_EN          1     // LVGL 打印使能: 1, 使能; 0, 关闭

#endif
```

(3) 在 `./middlewares_app/lvgl_app/lvgl_app.c` 中查看 LVGL_AppInit 初始化函数, 用户一般不需要修改该函数, 具体代码用户自行查看。在 `./source/TaskLCD.c` 中 App_TaskLCD 任务中调用此函数。

(4) 在 `./middlewares_app/lvgl_app/lvgl_app.c` 中查看 LVGL_AppProc 处理函数, 用户在这里编写自己的 LCD 显示程序, 具体代码用户自行查看。在 `./source/TaskLCD.c` 中 App_TaskLCD 任务中调用此函数。

35. OpenAMP 编程说明

(1) 关键词定义如下: OPENAMP_EN、LOPENAMP_DEBUG_EN、VIR_UARTx_EN、VIR_UARTx_MODE、VIR_MODBUS_SLAVE_ID、VIR_UART_MAX_NUM、VIR_UART_DEBUG_EN; 其中 x 范围: 0-8

(2) 在功能配置文件 ./middlewares_app/openamp_app/openamp_config.h 中定义:

```
#define OPENAMP_EN          1    // OPENAMP 使能: 1, 使能; 0, 关闭;

#if (OPENAMP_EN > 0)

#define OPENAMP_DEBUG_EN    1    // OPENAMP 打印使能: 1, 使能; 0, 关闭

// OpenAMP 下虚拟串口通信配置

#define VIR_UART_EN          1    // 虚拟串口使能: 1, 使能; 0, 关闭;

#if (VIR_UART_EN > 0)

#define VIR_UART_SCAN_T      1    // 设置定时扫描时间间隔, 单位: ms

#define VIR_UART0_EN          1    // 设置虚拟串口 0 使能: 1, 使能; 0, 关闭;

#define VIR_UART0_MODE        0    // 设置虚拟串口 0 工作模式: 0, Modbus RTU 工作模式;
                                   // 1, 自定义工作模式; 2, 测试工作模式: 将接收到的数据发回;

#define VIR_MODBUS_SLAVE_ID  1    // 设置本机 MODBUS ID

#define VIR_UART1_EN          1    // 设置虚拟串口 1 使能: 1, 使能; 0, 关闭;

#define VIR_UART1_MODE        2    // 设置虚拟串口 1 工作模式: 0, Modbus RTU 工作模式;
                                   // 1, 自定义工作模式; 2, 测试工作模式: 将接收到的数据发回;

#define VIR_UART2_EN          1    // 设置虚拟串口 2 使能: 1, 使能; 0, 关闭;

#define VIR_UART2_MODE        2    // 设置虚拟串口 2 工作模式: 0, Modbus RTU 工作模式;
                                   // 1, 自定义工作模式; 2, 测试工作模式: 将接收到的数据发回;

#define VIR_UART3_EN          1    // 设置虚拟串口 3 使能: 1, 使能; 0, 关闭;

#define VIR_UART3_MODE        2    // 设置虚拟串口 3 工作模式: 0, Modbus RTU 工作模式;
                                   // 1, 自定义工作模式; 2, 测试工作模式: 将接收到的数据发回;

#define VIR_UART4_EN          1    // 设置虚拟串口 4 使能: 1, 使能; 0, 关闭;

#define VIR_UART4_MODE        2    // 设置虚拟串口 4 工作模式: 0, Modbus RTU 工作模式;
                                   // 1, 自定义工作模式; 2, 测试工作模式: 将接收到的数据发回;

#define VIR_UART5_EN          1    // 设置虚拟串口 5 使能: 1, 使能; 0, 关闭;

#define VIR_UART5_MODE        2    // 设置虚拟串口 5 工作模式: 0, Modbus RTU 工作模式;
```

```

// 1, 自定义工作模式; 2, 测试工作模式: 将接收到的数据发回;

#define VIR_UART6_EN      1    // 设置虚拟串口 6 使能: 1, 使能; 0, 关闭;

#define VIR_UART6_MODE    2    // 设置虚拟串口 6 工作模式: 0, Modbus RTU 工作模式;
                                // 1, 自定义工作模式; 2, 测试工作模式: 将接收到的数据发回;

#define VIR_UART7_EN      1    // 设置虚拟串口 7 使能: 1, 使能; 0, 关闭;

#define VIR_UART7_MODE    2    // 设置虚拟串口 7 工作模式: 0, Modbus RTU 工作模式;
                                // 1, 自定义工作模式; 2, 测试工作模式: 将接收到的数据发回;

#define VIR_UART8_EN      1    // 设置虚拟串口 8 使能: 1, 使能; 0, 关闭;

#define VIR_UART8_MODE    2    // 设置虚拟串口 8 工作模式: 0, Modbus RTU 工作模式;
                                // 1, 自定义工作模式; 2, 测试工作模式: 将接收到的数据发回;

#define VIR_UART_MAX_NUM  9    // 最大虚拟串口数量

#define VIR_UART_DEBUG_EN 1    // 虚拟串口打印使能: 1, 使能; 0, 关闭

#endif

#endif

```

(3) 在 `./middlewares_app/openamp_app/openamp_app.c` 中查看 `OpenAMP_AppInit` 初始化函数, 用户一般不需要修改该函数, 具体代码用户自行查看。在 `./source/TaskInputData.c` 中 `APP_InputDataInit` 中调用此函数。

(4) 在 `./middlewares_app/openamp_app/openamp_app.c` 中查看 `OpenAMP_AppProc` 应用处理程序, 我们在这里接收其它内核发来的数据并处理, 同时调用 `VirUart_AppProc` 虚拟串口处理程序, 具体代码用户自行查看。在 `./middlewares_app/openamp_app/openamp_app.c` 和 `viruart_app.c`。

(5) 在 `./middlewares_app/openamp_app/viruart_app.c` 中 `VirUart_Write(INT8U id, INT8U *p, INT16U len)` 是虚拟串口发送函数, 其中 `id`, `VIR_UART` 索引标识 (`VIR_UART1_ID~VIR_UARTx_ID`); `*p`, 发送数据块指针; `len`, 发送数据块长度; 用户可以直接调用此函数来发送数据。

(6) 在 `./middlewares_app/openamp_app/viruart_app.c` 中 `VIRT_UARTx_RxCpltCallback` 是虚拟串口接收数据回调函数, 在这个函数中调用 `LibAppVars.Register.APP_InputDataCallback` 发送接收到数据。

在 `./source/user_viruartapp.c` 中 `VirUartx_UserProcess` 是接收数据处理函数, 用户可以在这里编写自己的应用程序。

附录：文档修改记录

序号	版本	时间	更新内容
1	V1. 20	2022. 6. 1	正式发布。
2	V1. 21	2022. 6. 10	修改部分文档
3	V1. 20. 06	2022. 11. 15	(1). 增加产品 RTU-BUS_V1.20 工程文件 / 测试例程及 config.h, const.h 文件 (2). 在 config 目录下增加 IAP 配置文件 iap_config.h, 将所有 AMKNAMKNXXXX_Config.h 中的 IAP 配置部分全部取消 (3). 将 libapp 目录下所有文件中函数都改为 XXX_AppXXX() 形式, 但为了兼容老版本, 原有函数做了定义, 也可以用。 (4). 将 libapp 目录下部分文件增加了注册函数, 以方便接收数据及产生中断 (5). 修改 dac_app.c 中 DAC_AppTest 函数, 修正波形输出错误 (6). 在 io_app.c 中: 将 DI_Read 函数改为只读取 1 路 DI 值, 读取多路 DI 值改为用 DI_MulRead 这个函数 (7). 在 io_app.c 中: 将 __DI_Read() 取消 DI1_MODE~DI32_MODE 条件编译定义, 解决未定义无法读取 DI 问题 (8). 在 config 目录下增加 MODBUS 配置文件 modbus_config.h, 将所有 AMKNAMKNXXXX_Config.h 中的 MODBUS 配置部分全部取消 (9). 将 source 目录下增加 user_modbusapp.c 文件, 实现 MODBUS 的操作例程; (10). 将 source 目录下增加 user_pwmapp.c 文件, 实现 PWM 的操作例程 (11). 在 AMKNAMKNXXXX_Config.h 配置文件中增加 PWMx_SCAN_T 定义 (12). 将原 source 文件夹下 user_app.c 中的 User_AppProc/User_AppInit/User_AppConfigInit 分别更名为 User_MainAppProc/User_MainAppInit/User_MainAppConfigInit。并将 User_MainAppProc 调用位置更改为 APP_ProcessData 或 App_TaskProcessData 函数。 User_MainAppProc/User_MainAppInit/User_MainAppConfigInit 用户编写应用程主要放到这 3 个函数当中 (13). 在 UserVars.h 中增加了用户使用变量, 用于存储采集的数据, 用户可以自由修改。 (14). 在 driver 文件夹下增加 SPI 转 4 个 UART 芯片 CH9434 驱动程序 (15). 在 driver 文件夹下增加 DAC8562 芯片驱动程序 (16). 在 driver 文件夹下增加 24 位 AD 采集芯片 ADS1232 驱动程序 (17). 在 driver 文件夹下增加 24 位 AD 采集芯片 ADS1220 驱动程序 (18). 在 driver 文件夹下增加 LED 数码管控制芯片 TM1640 驱动程序 (19). 在 driver 文件夹下增加 WTN6xxx 系列语音播放芯片驱动程序 (20). 修改 source 文件夹下 user_testapp.c 文件, 完善测试程序 (21). 修改 config 文件夹下 test_config.h 文件, 完善测试配置 (22). 修改 io_app.c 中 DI 和 SW 初始化函数, 解决初始状态错误问题
4	V1. 30. 00	2025. 7. 1	(1). 增加工控模块 STM32H723XX 和 STM32H743XX, 而增加和修改部分文档 (2). 修改 AMKN 软件框架图形化说明及结构说明
5	V1. 31. 00	2025. 12. 1	修改部分内容, 适用于 AMKN 软件版本: V1. 31. XX

6	V1.40.00	2026.7.1	基础修改： (1) 更新驱动库版本到：1.40.00，具体修改内容见：lib_readme.txt (2) 修改../device/wifi/wifi_config.h，增加根据产品类型使能WIFI和选择通信UART (3) 修改../config/debug_config.h，增加根据产品类型选择DEBUG信息输出UART (4) 修改../config/modbus_config.h，以便支持主机函数可重入。修改../libapp/modbus_app.c中的Modbus_Applnit函数，取消总体缓存，增加通道缓存，以便支持主机函数可重入 (5) 修改TaskLWIP.c中TCPClient_Init函数内容：将pTCPClient->Link[id].conn->send_timeout设置为0，解决在FreeRTOS下发送数据失败问题 (6) 修改comfun.c中NET_OutputData函数内容：将OSQTCPClien2tTxData修改为OSQTCPCClient2TxData，解决编译错误 (7) 修改modbus_slave_app.c、user_app.c、user_pwmapp.c、dac_app.c中部分函数，支持最新的Modbus保持寄存器定义(在./source/inc/modbus_holdreg_config.h) (8) 修改vars.c/vars.h增加一些参数定义 (9) 默认使能MODBUS_SLAVE_EN，支持UART2（其它工控板）或UART3（AMKN8639工控板）访问 (10) 在工控板AMKN8804中默认使能MODBUS_SLAVE_VIRUART_EN，支持虚拟串口VIR_UARR0访问 (11) 默认使能WIFI_EN，自动选择通信端口UART (12) 增加TaskStepMotor步进电机控制任务 (13) 增加TaskCanOpenCANOpen通信任务 AMKN软件框架结构变化： (1) 建立中间件文件夹：middlewares，负责中间件软件；建立中间件应用文件夹：middlewares_app，负责中间件软件用户应用； (2) 在middlewares下建立st/third_party/amkn文件夹，分别负责放入ST官方中间件、third_party第三方中间件、AMKN(中嵌凌云)中间件 (3) 将fatfs中间件移到middlewares/third_party下，目前版本是R0.13b；将file_app.c/file_app.h，fatfs_app.c/fatfs_app.h从labapp文件夹下移到middlewares_app/fatfs_app (4) 将lwip中间件移到middlewares/third_party/lwip下，目前FreeRTOS和无操作系统例程使用lwip版本是V2.1.3，UCOS-II操作系统使用的lwip版本是V1.4.1 (5) 将freertos中间件移到middlewares/third_party/freertos下，目前使用版本是V10.5.1将freertos_app整体移到middlewares_app下 (6) 将ucos-ii中间件移到middlewares/third_party/下，目前使用版本是V2.92.11将ucos_app整体移到middlewares_app下 (6) 增加canopen中间件，将canopen中间件移到middlewares/third_party/下；将canopen_app整体移到middlewares_app下
---	----------	----------	--

			(7) 将 lvgl 中间件移到 middlewares/third_party/下; 将 lvgl_app 整体移到 middlewares_app 下 (8) 将 openamp 中间件移到 middlewares/third_party/下; 将 openamp_app 整体移到 middlewares_app 下 (9) 将 stepmotor 驱动从 device 中移出; 将 stepmotor 中间件移到 middlewares/amkn/下; 将 stepmotor_app 移到 middlewares_app 下 (10) 将 source 文件夹下建立 inc 文件夹, 将 source 文件夹下所有头文件(.h)移到文件夹 inc 下 (11) 将 libapp 文件夹下建立 inc 文件夹, 将 libapp 文件夹下所有头文件(.h)移到文件夹 inc 下 (12) 将原 driver 文件夹修改为 device, 是一些外置芯片(或模块)的驱动程序; 增加 device.c 文件可以通过配置自动加载程序; 修改 w5500 驱动程序 (13) 将 FreeRTOS 和 UCOS-II 操作系统例程合并成一个, 以 AMKN8629 为例在工程目录下 ./targets/AMKN8629 分别定义 AMKN8629_FreeRTOS 和 AMKN8629_UCOS-II 将例程命名为: AMKNxxxx_MDK_OS_V1.40.XX_2026XXXX (14) 优化步进电机驱动包括 stepmotor.c/stepmotor.h, stepmotor_app.c/stepmotor_app.h, user_stepmotor.c/user_stepmotor.h, stepmotor_modbus_app.c/TaskStepMotor.c, stepmotor_config.h/stepmotor_modbus_holdreg_config.h, amkn9630_stepmotor_config.h 等等文件
--	--	--	--