AMKN 软件开发平台
使用手册

（2022 年 11 月 15 日修订版）

版权声明

免责声明

商标声明

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.embedarm.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

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	
2	工业控制板	AMKN8600、AMKN8602G、AMKN8612、AMKN8612G、AMKN8616G、AMKN8626、AMKN8628、AMKN8630、STM32F407VE-DK、GD32F307VE-DK、GD32F303VE-DK、RTU-BUS系列、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

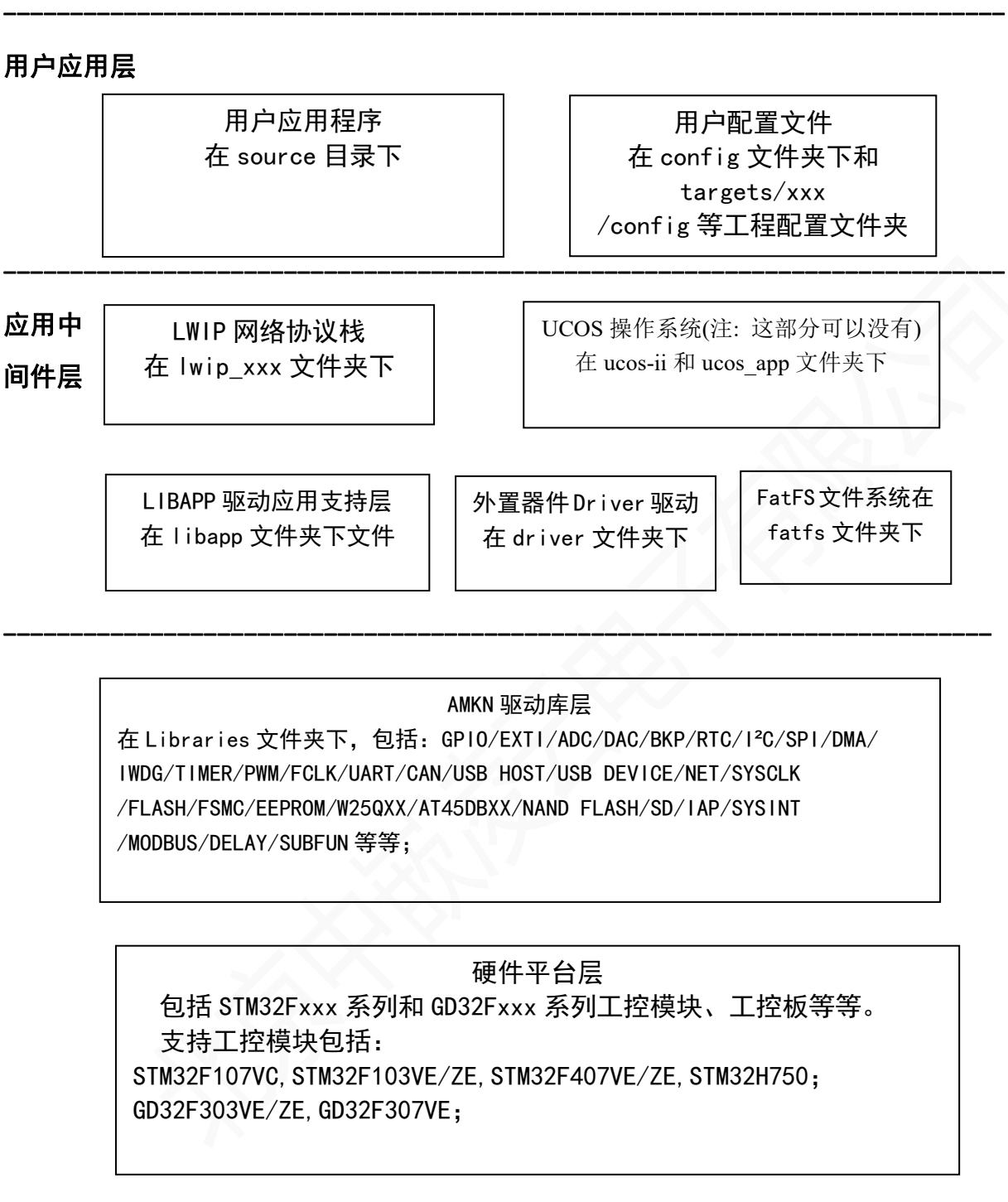
第一章. AMKN 软件开发平台文件结构 · 5
1. AMKN 软件开发平台简介 · 5
2. AMKN 软件结构框图 · 6
3. AMKN 软件主体结构介绍 · 7
4. AMKN 软件文件结构表 · 22
第二章. AMKN 软件系统配置文件说明 · 29
第三章. AMKN 软件各模块编程使用说明 · 31
1. DI 输入编程说明 · 31
2. DO 输出控制编程说明 · 33
3. KEY 按键输入编程说明 · 36
4. SW 拨码开关输入编程说明 · 38
5. UART 收发数据编程说明 · 40
6. CAN 收发数据编程说明 · 46
7. ADC(AI)输入编程说明 · 52
8. DAC(AO)输出编程说明 · 56
9. FCLK 脉冲输入编程说明 · 60
10. PWM 脉冲输出编程说明 · 66
11. TIM 定时器编程说明 · 70
12. EXTI 外部中断编程说明 · 71
13. SPI 总线编程说明 · 72
14. I²C 总线编程说明 · 73
15. RTC时钟编程说明 · 74
16. EEPROM 读写编程说明 · 74
17. SPI Flash 读写编程说明 · 75
18. NET 网络通信编程说明 · 78
19. USB 通信编程说明 · 87
20. SD 卡读写编程说明 · 92
21. Nand Flash 读写编程说明 · 95
22. Modbus 主机通信编程说明 · 98
23. Modbus 从机通信编程说明 · 99
24. WIFI 通信编程说明 · 102
25. 外部器件(传感器)编程说明 · 105
26. AT 指令编程说明 · 108
附录：文档修改记录 · 109

第一章. AMKN软件开发平台文件结构

1. AMKN软件开发平台简介

AMKN软件开发平台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软件开发平台简称：AMKN软件。

2. AMKN软件结构框图



3. AMKN软件主体结构介绍

(1) 主函数main

在../source/mian.c中

```
int  main (void)
{
    API_Init();                 // 硬件及应用功能初始化
    #if (UCOS_II_EN > 0)        // 选择UCOS-II为操作系统
    UCOS_II_Start();            // UCOS-II启动运行
    #elif (FREE_RTOS_EN > 0)    // 选择FREE_RTOS为操作系统
    #else                       // 无操作系统
    MainApp();                  // 应用主循环
    #endif
    return 0;                   // 永远运行不到这里
}
```

这个函数是硬件加电或复位后第1个运行的的函数：首先执行硬件及应用功能初始化函数API_Init。下来如果有操作系统则执行UCOS_II_Start，让操作系统接管程序运行。如果没有操作系统则执行无操作系统应用主循环MainApp函数。

(2) 硬件及应用功能初始化函数API_Init

在../libapp/libapp.c中

```
void  API_Init (void)
{
    SysLib_AppInit();           // 系统驱动库应用初始化函数;
    #if (IWDG_EN > 0)           // IWDG配置使能
    IWDG_Init(IWDG_TIME);       // 初始化看门狗时间
    IWDG_Ctrl(CMD_IWDG_ENA);    // 使能看门狗
    IWDG_Ctrl(CMD_IWDG_CLEAR);  // 喂狗
    #endif
    #if ((DMA1_EN > 0)||(DMA2_EN > 0)) // DMA1-DMA2配置使能
    DMA_AppInit();                  // DMA1-DMA2应用初始化函数;
```

```
    #endif
    EEPROM_AppInit();                   // EEPROM应用初始化函数;
    #if ((SPI1_EN > 0)||(SPI2_EN > 0)||(SPI3_EN > 0)) // SPI1~SPI3配置使能
    SPI_AppInit();                      // SPI总线应用初始化
    #endif
    #if (SPIFLASH_EN > 0)    // SPI FLASH配置使能
    SPIFlash_AppInit();      // SPI Flash(W25QXX或AT45DBXX)应用初始化
    #endif
    LibApp_ParaRead();    // 读取用户参数
    LibAppVars_Init();    // LibAppVars变量初始化函数
    #if ((UART1_EN > 0)||(UART2_EN > 0)||(UART3_EN > 0)||(UART4_EN > 0)||(UART5_EN > 0)
        ||(UART6_EN > 0)) // UART1-UART6配置使能
    Uart_AppInit();       // UART1-UART6应用初始化函数;
    #endif
    Logo_Out();           // 软件logo打印输出
    #if ((I²C2_EN > 0)||(I²C3_EN > 0)) // I²C2-I²C3配置使能
    I²C_AppInit();       // I²C2-I²C3应用初始化函数;
    #endif
     #if (IAP_EN > 0)     // IAP配置使能
    IAP_AppInit();        // IAP应用初始化
    #endif
     #if (IWDG_EN > 0)             // IWDG配置使能
    IWDG_Ctrl(CMD_IWDG_CLEAR);  // 喂狗
    #endif
    IO_AppInit();        // 所有IO初始化
    #if (EXIT_EN > 0)
   EXTI_AppInit();     // 外部中断和事件应用初始化
    #endif
    #if (BKP_EN > 0)    // BKP配置使能
```

```
    BKP_Init();          // BKP应用初始化
    #endif
    #if (MODBUS_SLAVE_EN > 0)    // Modbus设备模式配置使能
    ModbusSlave_AppInit();       //Modbus从机(设备)模式应用初始化函数;
    #endif
    #if (MODBUS_EN > 0)          // Modbus主机模式配置使能
    Modbus_AppInit();            // Modbus主机模式应用初始化函数
    #endif
    #if (IWDG_EN > 0)            // IWDG配置使能
       IWDG_Ctrl(CMD_IWDG_CLEAR);  // 喂狗
    #endif
    #if (RTC_EN > 0)             // RTC配置使能
    RTC_AppInit();               // RTC应用初始化函数;
    #endif
    #if (ADC_EN>0)               // ADC配置使能
    ADC_AppInit();               // ADC应用初始化函数;
    #endif
    #if ((DAC1_EN>0)||(DAC2_EN>0)) // DAC1-2配置使能
    DAC_AppInit();               // DAC应用初始化
    #endif
    #if ((TIM1_EN > 0)||(TIM2_EN > 0)||(TIM3_EN > 0)||(TIM4_EN > 0)||(TIM5_EN > 0)||
        (TIM6_EN > 0)||(TIM7_EN > 0)||(TIM8_EN > 0)||(TIM9_EN > 0)||(TIM10_EN > 0)||
        (TIM11_EN >0)||(TIM12_EN > 0)||(TIM13_EN > 0)||(TIM14_EN > 0)) //TIM1-TIM14配置
使能
    TIM_AppInit();       // 定时器应用初始化
    #endif
    #if (FCLK1_EN>0)||(FCLK2_EN>0)||(FCLK3_EN>0)||(FCLK4_EN>0)||(FCLK5_EN>0)
        ||(FCLK6_EN>0)||(FCLK7_EN>0)||(FCLK8_EN>0)) //FCLK1-FCLK8配置使能
    FCLK_AppInit();      // FCLK输入应用初始化
```

```
#endif
#if ((PWM1_EN>0)||(PWM2_EN>0)||(PWM3_EN>0)||(PWM4_EN>0)||(PWM5_EN>0)||(PWM6_EN>0)
    ||(PWM7_EN>0)||(PWM8_EN>0)) //PWM1-PWM8配置使能
PWM_AppInit();      // PWM输出应用初始化
#endif
#if ((CAN1_EN>0)||(CAN2_EN>0)) // CAN1-2配置使能
CAN_AppInit();                  // CAN1-2应用初始化
#endif
#if (SDCARD_EN > 0)      // SD卡配置使能
SD_AppInit();            // SD卡应用初始化
#endif
#if (IWDG_EN > 0)            // IWDG配置使能
IWDG_Ctrl(CMD_IWDG_CLEAR);  // 喂狗
#endif
 #if ((USB_HOST_EN > 0)&&(MODULE_CLASS != STM32F103XX)) // USB主机模式配置使能
USBH_AppInit();          // USB主机模式应用初始化
#endif
#if (USB_DEVICE_EN > 0) // USB设备模式配置使能
USBD_AppInit();          // USB设备模式应用初始化
#endif
#if (FSMC_EN > 0)        // FSMC总线配置使能
FSMC_AppInit();          // FSMC总线初始化
#endif
#if (NFLASH_EN > 0)      // NAND FLASH配置使能
NFlash_AppInit();        // NAND FLASH应用初始化
#endif
#if (LWIP_EN >0)
NET_AppInit();             // 网络配置使能
#endif
```

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

这个函数完成所有硬件初始化及一些应用初始化，一般用户无需修改这个函数。具体各个硬件初始化程序请查看相关初始化函数。

(3) 应用主循环函数MainApp

在../source/mian.c中

```
void MainApp(void)
{
    APP_InputDataInit();      // 输入数据初始化或启动运行
    while(1)
    {
        APP_InputData();     // 输入数据处理
        APP_ProcessData();   // 数据处理
        APP_OutputData();    // 数据输出
    }
}
```

这个函数是在无操作系统的主循环函数，基本采用IPO模式。下面分别介绍这几个函数。

(4) 输入数据初始化函数APP_InputDataInit

在../source/TaskInputData.c中

```
void APP_InputDataInit(void)
{
    INT32S flag;
    UserVars_Init();          //  全局变量初始化
    APP_RegisterCallback();  // 注册libapp中回调函数，不可取消
    User_InputDataInit();     // 用户输入采集数据初始化函数
    User_MainAppInit();       // 用户应用初始化，不可取消
```

```
    #if ((AI_NUM > 0)&&(ADC_EN > 0))
    ADC_Ctrl(ADC1_ID, CMD_ADC_ENA, 0);    // 启动ADC使能开始工作
    #endif
    #if (TIMX_EN > 0)
    TIM_AppCtrl(0, CMD_TIM_APP_CTRL_START_ALL, 0);    // TIM启动运行
    #endif
    #if (FCLK_EN > 0)
    FCLK_AppCtrl(0, CMD_FCLK_APP_CTRL_START_ALL, 0);// FCLK启动运行
    #endif
    #if (FATFS_EN > 0)          // 文件系统操作使能
    File_AppInit();             // 文件应用操作初始化
    #endif
    #if (MCU_IDLE_EN > 0)
    MCU_IdleInit();           // MCU空闲占比初始化
    #endif
    #if (DEVICE_EN > 0)
    Device_AppInit();         // 外部器件应用初始化
    #endif
    // 以下是用户测试处理
    #if (APP_TEST_EN > 0)
    User_AppTestInit();          // 用户测试初始化
    #endif
}
```

这个函数是输入数据模块初始化或启动运行函数。如果有操作系统则这个函数被App_TaskInputData任务调用。

如果没有操作系统这个函数被应用主循环函数MainApp调用。这个函数用户一般不需要修改。

(5) 输入数据函数APP_InputData

在../source/TaskInputData.c中

```
void APP_InputData(void)
{
    #if (DEBUG_ISRHOOK_EN > 0)  // 条件编译中断打印调试功能使能
    if(DebugInfo.len > 0)       // 判断是否有要打印数据
    {
        Uart_Write(DEBUG_UART, DebugInfo.buf, DebugInfo.len);  // 输出打印数据
        DebugInfo.len = 0;      // 清除数据长度
    }
    #endif
    #if (DEVICE_EN > 0)
    Device_AppProc();           // 外部器件应用处理
    #endif
    // UART1-8接收数据处理
    Uart_AppProc();               // UART接收数据处理
    // CAN1-2接收数据处理
    #if ((CAN1_EN > 0)||(CAN2_EN > 0))
    CAN_AppProc();                // CAN接收数据处理
    #endif
    // ADC读取数据处理
    #if ((AI_NUM > 0)&&(ADC_EN > 0))
    ADC_AppProc();                // ADC读取采集数据处理
    #endif
    #if ((DAC1_EN > 0)||(DAC2_EN > 0))
    DAC_AppProc();                // DAC应用处理函数
    #endif
   IO_AppProc();                  // IO读取数据处理
    // 读取FCLK输入脉冲
```

```
    #if (FCLK_EN > 0)
    FCLK_AppProc();             // FCLK读取处理
    #endif
    #if ((USB_DEVICE_EN > 0)||(USB_HOST_EN > 0))  // USB配置使能
    USB_AppProc();              // USB应用处理
    #endif
    #if (FATFS_EN > 0)          // 文件系统操作使能
    File_AppProc();             // 文件应用处理
    #endif
    #if (WIFI_EN > 0)
    WIFI_AppProc();             // WIFI应用处理
    #endif
    User_AppProc();             // 用户应用处理, 这个不要取消
    #if (MCU_IDLE_EN > 0)
    MCU_IdleProc();             // MCU空闲计算处理
    #endif
    #if (LWIP_EN > 0)
    NET_AppProc();              // 网络应用处理
    #endif
    #if (MODBUS_EN > 0)
    Modbus_AppProc();           // Modbus主机应用处理
    #endif
    User_InputDataProc();       // 用户输入采集数据
    // 以下是用户测试处理
    #if (APP_TEST_EN > 0)
    User_AppTestProc();         // 用户测试应用处理
    #endif
}
```

这个函数是所有输入数据的读取接收模块。如果有操作系统则这个函数被

App_TaskInputData任务调用。

如果没有操作系统这个函数被应用主循环函数MainApp调用。这个函数用户一般不需要修改。在各个应用处理程序里，读取和接收的数据会调用APP_InputDataCallback这个回调函数将数据发送出去。

(6) 数据处理函数APP_ProcessData

在../source/TastProcessData.c中

```
void APP_ProcessData(void)
{    INT16U i;
    for (i=0; i<LIBAPP_INPUTDATA_MSG_SIZE; i++)   // 循环判定消息缓存是否有新数据
    {
        if (LibAppVars.InputDataMsg[i].Flag > 0)  // 判断有新数据
        {    // 调用用户处理函数
         APP_UserProcessData((LIBAPP_INPUTDATA_MSG *)&LibAppVars.InputDataMsg[i].Flag);
        }
    }
}
void APP_UserProcessData(LIBAPP_INPUTDATA_MSG *pInputDataMsg)
{
    switch(pInputDataMsg->Type)
    {
        case LIBAPP_DATA_UART_TYPE:                    // UART接收的数据类型
            Uart_DataProcess(pInputDataMsg);           // 对UART接收数据做处理
            break;
        #if ((CAN1_EN > 0)||(CAN2_EN > 0))
        case LIBAPP_DATA_CAN_TYPE:                     // CAN接收的数据类型
            CAN_DataProcess(pInputDataMsg);            // 对CAN接收数据做处理
            break;
        #endif
        #if (LWIP_EN > 0)
```

```
#if (LWIP_TCP_SERVER_EN > 0)
case LIBAPP_DATA_TCP_SERVER_TYPE:              // TCP服务器模式数据类型
    NET_TCPServerDataProcess(pInputDataMsg);   // 对TCP服务器接收数据做处理
    break;
#endif
#if (LWIP_UDP_SERVER_EN > 0)
case LIBAPP_DATA_UDP_SERVER_TYPE:              // UDP服务器模式数据类型
    NET_UDPServerDataProcess(pInputDataMsg);   // 对UDP服务器接收数据做处理
    break;
#endif
#if (LWIP_TFTP_SERVER_EN > 0)
case LIBAPP_DATA_TFTP_SERVER_TYPE:             // TFTP服务器模式数据类型
    NET_TFTPServerDataProcess(pInputDataMsg); // 对TFTP服务器接收数据做处理
    break;
#endif
#if (LWIP_TCP_CLIENT_EN > 0)
case LIBAPP_DATA_TCP_CLIENT_TYPE:              // TCP客户端模式数据类型
    NET_TCPClientDataProcess(pInputDataMsg);   // 对TCP客户端接收数据做处理
    break;
#endif
#if (LWIP_UDP_CLIENT_EN > 0)
case LIBAPP_DATA_UDP_CLIENT_TYPE:              // UDP客户端模式数据类型
    NET_UDPClientDataProcess(pInputDataMsg);   // 对UDP客户端接收数据做处理
    break;
#endif
#endif
#if (WIFI_EN > 0)
case LIBAPP_DATA_WIFI_TYPE:                    // WIFI数据类型
    WIFI_DataProcess(pInputDataMsg);           // 对WIFI接收数据做处理
```

```
        break;
    #endif
    #if (KEY_EN > 0)
    case LIBAPP_DATA_KEY_TYPE:                  // 按键数据类型
        KEY_DataProcess(pInputDataMsg);         // 对按键数据做处理
        break;
    #endif
    #if (DI_EN > 0)
    case LIBAPP_DATA_DI_TYPE:                   // DI数据类型
        DI_DataProcess(pInputDataMsg);          // 对DI数据做处理
        break;
    #endif
    #if (SW_EN > 0)
    case LIBAPP_DATA_SW_TYPE:                   // SW拨码开关数据类型
        SW_DataProcess(pInputDataMsg);          // 对SW数据做处理
        break;
    #endif
    #if (ADC_EN > 0)
    case LIBAPP_DATA_ADC_TYPE:                  // ADC数据类型
        ADC_DataProcess(pInputDataMsg);         // 对ADC数据做处理
        break;
    #endif
    #if (FCLK_EN > 0)
    case LIBAPP_DATA_FCLK_TYPE:                 // FCLK数据类型
        FCLK_DataProcess(pInputDataMsg);        // 对FCLK数据做处理
        break;
    #endif
    #if (MODBUS_EN > 0)
    case LIBAPP_DATA_MODBUS_TYPE:               // MODBUS数据类型
```

```
            Modbus_DataProcess(pInputDataMsg);          // 对MODBUS数据做处理
            break;
        #endif
        #if ((USB_DEVICE_EN > 0)&&(USB_VCP_EN > 0))   // 虚拟串口使能
        case LIBAPP_DATA_USB_TYPE:                      // USB数据类型
            USB_DataProcess(pInputDataMsg);             // 对USB数据做处理
            break;
        #endif
        #if (DEVICE_EN > 0)
        case LIBAPP_DATA_DEVICE_TYPE:                   // 外部器件(传感器)类型数据
            Device_DataProcess(pInputDataMsg);          // 对外部器件(传感器)数据做处理
            break;
        #endif
        default:
            break;
    }
    pInputDataMsg->Flag = 0;    // 清除数据标志
}
```

这个APP_UserProcessData函数是处理所有输入数据，如果有操作系统则这个函数被App_TaskProcessData

任务调用。如果没有操作系统这个函数被APP_ProcessData调用(APP_ProcessData被应用主循环函数MainApp

调用)。这个函数用户一般不需要修改。

(7) 数据输出函数APP_OutputData

在../source/TastOutputData.c中

```
void APP_OutputData(void)
{
    INT16U i;
```

```
    for (i=0; i<LIBAPP_OUTPUTDATA_MSG_SIZE; i++)   // 循环判定消息缓存是否有新数据
    {
      if (LibAppVars.OutputDataMsg[i].Flag > 0)  // 判断有新数据
      {   // 调用用户处理函数
        APP_UserOutputData((LIBAPP_OUTPUTDATA_MSG *)&LibAppVars.OutputDataMsg[i].Flag);
      }
    }
}
void APP_UserOutputData(LIBAPP_OUTPUTDATA_MSG *pOutputDataMsg)
{
    switch(pOutputDataMsg->Type)
    {
        case LIBAPP_DATA_UART_TYPE:                    // UART数据类型
            Uart_UserOutputData(pOutputDataMsg);       // UART发送数据
            break;
        #if ((CAN1_EN > 0)||(CAN2_EN > 0))
        case LIBAPP_DATA_CAN_TYPE:                     // CAN数据类型
            CAN_UserOutputData(pOutputDataMsg);        // CAN发送数据
            break;
        #endif
        #if (LWIP_EN > 0)
        #if (LWIP_TCP_SERVER_EN > 0)
        case LIBAPP_DATA_TCP_SERVER_TYPE:                  // TCP服务器模式数据类型
            NET_UserTCPServerOutputData(pOutputDataMsg);   // TCP服务器发送数据
            break;
        #endif
        #if (LWIP_UDP_SERVER_EN > 0)
        case LIBAPP_DATA_UDP_SERVER_TYPE:                  // UDP服务器模式数据类型
            NET_UserUDPServerOutputData(pOutputDataMsg);   // UDP服务器发送数据
```

```
        break;
    #endif
    #if (LWIP_TFTP_SERVER_EN > 0)
    case LIBAPP_DATA_TFTP_SERVER_TYPE:                  // TFTP服务器模式数据类型
        NET_UserTFTPServerOutputData(pOutputDataMsg);   // TFTP服务器发送数据
        break;
    #endif
    #if (LWIP_TCP_CLIENT_EN > 0)
    case LIBAPP_DATA_TCP_CLIENT_TYPE:                   // TCP客户端模式数据类型
        NET_UserTCPClientOutputData(pOutputDataMsg);    // TCP客户端发送数据
        break;
    #endif
    #if (LWIP_UDP_CLIENT_EN > 0)
    case LIBAPP_DATA_UDP_CLIENT_TYPE:                   // UDP客户端模式数据类型
        NET_UserUDPClientOutputData(pOutputDataMsg);    // UDP客户端发送数据
        break;
    #endif
    #endif
    #if (WIFI_EN > 0)
    case LIBAPP_DATA_WIFI_TYPE:                         // WIFI数据类型
        WIFI_UserOutputData(pOutputDataMsg);            // WIFI发送数据
        break;
    #endif
    #if ((USB_DEVICE_EN > 0)&&(USB_VCP_EN > 0)) // 虚拟串口模式
    case LIBAPP_DATA_USB_TYPE:                          // USB数据类型
        USB_UserOutputData(pOutputDataMsg);             // USB发送数据
        break;
    #endif
    default:
```

```
            break;
        }
    }
```

这个APP_UserOutputData函数是处理输出数据，如果有操作系统则这个函数被App_TaskOutputData

任务调用。如果没有操作系统这个函数被APP_OutputData调用(APP_OutputData被应用主循环函数MainApp调用)。这个函数用户一般不需要修改。

4. AMKN软件文件结构表

序号	文件夹	文件	描述
1	source 用户程序 用户可修改本目录下程序来完成软件设计	main.c	系统主程序
		UserVar.c/UserVar.h	应用程序全局常量和变量定义文件,用户应用程序的全局常量和变量在这里定义
		ISRHook.c	系统中断处理文件,在这里处理所有中断及驱动库钩子函数
		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 注：一般不用修改	本文件负责数据输出，实现UART/CAN/NET/WIFI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
		comfun.c/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_fclkapp.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	本文件是用户应用程序
		user_testapp.c	本文件是测试应用程序
		user_config.h	本文件是用户APP配置文件
		user_pwmapp.c	本文件是用户处理PWM输出应用程序
		user_modbusapp.c	本文件是用户处理modbus应用程序
		user_inputdata.c	本文件是用户输入采集数据应用程序
2	config 系统配置 这个目录是整个软件平台结构配置目录	const.h	常量定义，用户不可更改
		vars.h,vars.c	驱动库全局变量定义，用户不可更改
		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
		config.h	全局配置文件，用户根据自己所应用板子型号选择定义，如应用AMKN8602G工控板则需要如下定义： #define PRODUCT_TYPE AMKN8602G
		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应用配置文件
3	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	LibAppVars.h/LibAppVars.c	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/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/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
		eeprom_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
		I²C_app.c	本文件是I²C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

4	ucos_app 操作系统配置及变量定义目录 注：这部分可选	os_cfg.h	ucos操作系统配置文件
		app_cfg.h	应用任务配置文件
		Includes.h	操作系统引用头文件
		OSHook.c/OSHook.h	操作系统应用钩子函数处理文件
		OSVar.c/OSVar.c	操作系统全局变量定义文件
		ucos_app.c/ucos_app.h	本文是ucos应用处理程序
5	libraries 基础驱动库 注：此文件目录下文件不可更改，由我司进行维护	sysint.h	系统驱动库头文件
		gpio.h	GPIO通用IO硬件驱动程序头文件
		exit.h	外部中断硬件驱动程序头文件
		rtc.h	RTC 实时时钟硬件驱动程序头文件
		iwdg.h	独立看门狗硬件驱动程序头文件
		uart.h	UART 异步串口(RS232 或 RS485)硬件驱动程序头文件
		I²C.h	I²C 总线硬件驱动程序头文件
		spi.h	SPI 总线硬件驱动程序头文件
		can.h	CAN 总线硬件驱动程序头文件
		timer.h	定时器(PWM/FCLK)硬件驱动程序头文件
		dac.h	DAC 硬件驱动程序头文件
		adc.h	ADC 硬件驱动程序头文件
		flash.h	内部 FLASH 硬件驱动程序头文件
		crc.h	CRC 校验硬件驱动程序头文件
		sd.h	SD卡(SPI)读写硬件驱动程序头文件
		AT45DBXX.h	AT45DBXX系列FLASH读写硬件驱动程序头文件
		W25QXX.h	W25QXX系列FLASH读写硬件驱动程序头文件
		eeprom.h	EEPROM读写硬件驱动程序头文件
		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
		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_V120.xxxx.lib	STM32F107VC模块驱动库文件
		STM32F107VC_NUSB_V120.xxxx.lib	STM32F107VC模块不包含USB部分驱动库文件
		STM32F103VE_V120.xxxx.lib	STM32F103VE模块驱动库文件
		STM32F103VE_NUSB_V120.xxxx.lib	STM32F103VE模块不包含USB部分驱动库文件
		STM32F103ZE_V120.xxxx.lib	STM32F103ZE模块驱动库文件
		STM32F407XX_V120.xxxx.lib	STM32F407XX系列模块驱动库文件
		GD32F307XX_V120.xxxx.lib	GD32F307XX模块驱动库文件
		GD32F307XX_NUSB_V120.xxxx.lib	GD32F307XX模块不包含USB部分驱动库文件
		GD32F303VE_V120.xxxx.lib	GD32F303VE模块驱动库文件
		GD32F303VE_NUSB_V120.xxxx.lib	GD32F303VE模块不包含USB部分驱动库文件
		GD32F303ZE_V120.xxxx.lib	GD32F303ZE模块驱动库文件
6	driver 用户一些外部器件驱动程序	AT.c/AT.h	本文件负责AT指令处理,参见《STM32Fxxx系列工控板(模块)AT指令_1.02_20211119.pdf》
		CH455.c/CH455.h	CH455按键控制采集芯片驱动程序
		I²C_io.c/I²C_io.h	IO模拟I²C总线驱动程序
		shtxx.c/shtxx.h	shtxxx系列温湿度传感器驱动程序
		wifi.c/wifi.h	wifi模块驱动程序
		max6675.c/max6675.h	MAX6675芯片操作驱动函数文件

		spl06.c/spl06.h	SPL06芯片操作驱动函数文件
		dac8562.c/dac8562.h	DAC8562芯片操作驱动函数文件
		dac8562_app.c/dac8562_app.h	DAC8562芯片操作应用函数文件
		ads1232.c/ads1232.h	4位AD采集芯片ADS1232驱动函数文件
		ads1220.c/ads1220.h	24位AD采集芯片ADS1220驱动函数文件
		ads1220_app.c/ads1220_app.h	24位AD采集芯片ADS1220应用处理函数文件
		ch9434.c/ch9434.h	SPI转4个UART芯片CH9434驱动函数文件
		ch9434_app.c/ch9434_app.h	SPI转4个UART芯片CH9434应用处理函数文件
		tm1640.c/tm1640.h	LED数码管控制芯片TM1640驱动函数文件
		tm1640_app.c/tm1640_app.h	LED数码管控制芯片TM1640应用处理函数文件
		wtn6xxx.c/wtn6xxx.h	WTN6xxx系列语音播放芯片驱动函数文件
		wtn6xxx_app.c/wtn6xxx_app.h	WTN6xxx系列语音播放芯片应用处理函数文件
		device_app.c/device_app.h	本文件是外部器件应用操作程序
		device_config.h	本文件是用户一些外部器件驱动配置文件
7	Fatfs 文件系统	ff.c、ff.h	Fatfs文件系统源代码，用户一般不用修改
		diskio.c、diskio.h	Fatfs硬件接口驱动文件，包括SPI FLASH/SD卡/U盘/Nand Flash的文件系统接口驱动；已经做好，客户不要修改
		integer.h	数据类型定义头文件一般不用修改
		ffconf.h	Fatfs文件系统配置文件,客户可以修改
8	init 初始化	inita.s	系统初始化汇编文件
		vectors.s	中断向量汇编文件
9	lwip TCPIP协议栈	lwip.c	网络协议栈源文件
		sys_arch.c sys_arch.h	lwip移植到ucos上接口驱动文件
		ethernetif.c	LWIP网络硬件驱动接口文件
		lwipopts.h	lwip配置文件,可以替代opt.h
		opt.h	lwip默认配置文件

10	ucos-ii 操作系统	ucos-ii.c	ucos-ii源文件
		cpu_a.asm,os_cpu_a.asm, lib_mem_a.asm	ucos-ii汇编移植汇编文件
11	targets 各型号工控板MDK工程文件及配置文件	AMKN8602G目录： Config/AMKN8602G_Config.h、 Config/AMKN8602G_IOConfig.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IO配置文件,用户根据需要自行修改
		AMKN8612G目录： Config/AMKN8612G_Config.h Config/AMKN8612G_IOConfig.h	里面包含工控板AMKN8612G的工程文件/配置文件(config里)/输出文件（output里） 工控板AMKN8612G配置文件和IO配置文件,用户根据需要自行修改
		AMKN8616G目录： Config/AMKN8616G_Config.h Config/AMKN8616G_IOConfig.h	里面包含工控板AMKN8616G的工程文件/配置文件(config里)/输出文件（output里） 工控板AMKN8616G配置文件和IO配置文件,用户根据需要自行修改
		AMKN8626目录： Config/AMKN8626_Config.h Config/AMKN8626_IOConfig.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IO配置文件,用户根据需要自行修改
		AMKN8628目录 Config/AMKN8628_Config.h Config/AMKN8628_IOConfig.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） 工控板AMKN8628配置文件和IO配置文件,用户根据需要自行修改
		STM32F407VE-DK目录： Config/STM32F407VE-DK_Config.h Config/STM32F407VE-DK_IOConfig.h	里面包含开发板STM32F407VE-DK的工程文件/配置文件(config里)/输出文件（output里） 开发板STM32F407VE-DK配置文件和IO配置文件,用户根据需要自行修改
		GD32F307VE-DK目录：	里面包含开发板GD32F307VE-DK的工程文件/配置

		Config/GD32F307VE-DK_Config.h Config/GD32F307VE-DK_IOConfig.h	文件(config里)/输出文件（output里） 开发板GD32F307VE-DK配置文件和IO配置文件，用户根据需要自行修改
		GD32F303VE-DK目录： Config/GD32F303VE-DK_Config.h Config/GD32F303VE-DK_IOConfig.h	里面包含开发板GD32F303VE-DK的工程文件/配置文件(config里)/输出文件（output里） 开发板GD32F303VE-DK配置文件和IO配置文件，用户根据需要自行修改
12	dot	readme.txt	软件说明文档

第二章 AMKN软件系统配置文件说明

1. const.h说明

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

2. vars.c, vars.h说明

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

3. config.h说明

这个文件是整个软件平台最全局的配置文件,用户只需要在这里配置工控板型号即可。例如用户购买工控板型号是AMKN8602G，则只需要做如下配置即可：#define PRODUCT_TYPE AMKN8602G, 其它产品型号请注销掉；这样根据这个配置，文件下面会自动为该工控板匹配相应的模块型号、配置文件和IO配置文件，如下定义：

```
#if (PRODUCT_TYPE == AMKN8602G)        // 如果选择产品型号: AMKN8602
#define  GD32F3_M4                     // 定义MCU的内核
#define  STM32F1M3_GD32F3M4
#define  MODULE_CLASS GD32F307XX       // 设置产品应用核心工控模块类别, 对应驱动库
                                       // GD32F307XX_xxxx_xxxxxxxx.lib
#define  MODULE_TYPE  GD32F307VE       // 设置产品应用核心工控模块型号, 对应驱动库
                                       // GD32F307XX_xxxx_xxxxxxxx.lib
#include "AMKN8602G_Config.h"          // 工控板AMKN8602G功能参数配置文件
#include "AMKN8602G_IOConfig.h"        // 工控板AMKN8602G IO端口配置文件
#endif
```

对于工控板AMKN8602G只需修改AMKN8602G_Config.h和AMKN8602G_IOConfig.h 两个配置文件即可，其它工控板定义参考此例,详细内容请查看例程的config.h文件.

另外：请根据是否使用操作系统来设置OS_EN为1或0

4. 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IAP功能固件)

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

STM32F103VE_FlashIAP.scat: 工控模块STM32F103VE, FLASH及RAM分配文件(编译可生成IAP功能固件)

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

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

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IAP功

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

第三章 AMKN软件各模块编程使用说明

1. DI输入编程说明

关键词定义如下：DI_EN、DI_MODE、DI_SCAN_T、DI_ATOUT_T、DI_NUM、DI1~DI32、DI1_ID~DI32_ID、DI1FLAG~DI32FLAG、DI1_8SPI_EN、DI9_16SPI_EN、DI17_24SPI_EN、DI25_32SPI_EN、HC597_STB、HC597_LOAD、HC597_CS

(1) 在IO配置文件XXXX_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

(2) 在功能配置文件XXXX_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；这个参数可以修改
#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
```

(3) 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。

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

```
// DI读取处理
#if ((DI_EN > 0)&&(DI_NUM > 0))                    // DI使能条件编译
tick = APP_GetSubTick(LibAppVars.DI.Scan_t);  // 读取和上次采集的间隔// 时间，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。

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

在这个函数中调用DI_UserProcess(),用户请在这个函数根据DI的变化做相应的处理：

```
void DI_UserProcess(INT8U id, INT8U val)
{   .... ...
    switch (id)
    {
        case DI1_ID:
            if (val == 1) {  } // DI1由0变为1处理
            else {  }          // DI1由1变为0处理
        break;
        case DI2_ID:
            if (val == 1) {  } // DI2由0变为1处理
            else {  }          // DI2由1变为0处理
        break;
      ... ...
      }
}
```

(6) 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. DO输出控制编程说明

关键词定义如下：DO_EN、DO_SCAN_T、DO_NUM、DO1~DO32、DO_OUT_MODE、DO1_ID~DO32_ID、DO1FLAG~DO32FLAG 、 DO1_INIT_VAL~DO32_INIT_VAL 、 DO1_8SPI_EN 、 DO9_16SPI_EN 、 DO17_24SPI_EN、DO25_32SPI_EN、HC595_STB、HC595_ENA

(1) 在IO配置文件XXXX_IOConfig.h中定义：DO的端口及DO数量，参考如下：

```
#define DO_NUM    2    // 定义DO输出端口数量
#define DO1    PF0
#define DO2    PF1
 #define DO_OUT_MODE IO_OUT_PP // 定义DO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 DO1_INIT_VAL   0 // 定义DO1输出初始化值
#define DO2_INIT_VAL   0 // 定义DO2输出初始化值
```

定义2个DO，分别对应硬件PF0和PF1端口

(2) 在功能配置文件XXXX_Config.h中定义，参考如下：

```
#define DO_EN    1    // DO使能，1：打开使能， 0：关闭；这个参数必须为1
#define DO_SCAN_T 1 //设置定时扫描时间间隔，单位：ms；这个参数可以修改
```

(3) 在../libapp/io_app.c 中查看 IO_AppInit()和 DO_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中有2个独立函数控制DO输出：

控制单路DO输出函数：DO_Write(INT8U id, INT8U val)

id，DO标识DO1_ID~DO32_ID；val，输出值，0或者1；

比如控制DO2输出1则这样写：DO_Write(DO2_ID, 1)；

控制多路DO同时输出函数：DO_MulWrite(INT32U DOFlag, INT8U val)

DOFlag, bit0~bit31依序代表DO1~DO32, 如果是1表示该DO执行输出; val, 输出值, 0或者1;

比如控制DO1 DO2同时输出1则这样写：DO_MulWrite(DO1FLAG|DO2FLAG, 1);

注意：这2个函数在任何位置和时间点都可以使用。

(5) 在../libapp/io_app.c中有1个DO事件控制函数：

```
void DO_EventCtrl(INT8U id, INT8U Cmd, INT16U t1, INT16U t2)
id, DO标识DO1_ID~DO32_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DO输出1, 延时t毫秒后恢复原输出0
 #define LIBAPP_DO_CMD_OFF_T 0x02 // 控制DO输出0, 延时t毫秒后恢复原输出1
#define LIBAPP_DO_CMD_NEG_T1   0x03 // 控制DO连续翻转输出: DO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DO连续翻转输出: DO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DO输出1时间,t2为DO输出0

LIBAPP_DO_CMD_NEG_T2: t1为DO输出0时间,t2为DO输出1

比如控制DO2输出1个50ms高电平脉冲：

```
DO_EventCtrl(DO2_ID,  LIBAPP_DO_CMD_ON_T, 50, 0);
```

比如控制DO2持续输出：50ms高电平，1000ms低电平：

```
DO_EventCtrl(DO2_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)                          // DO使能条件编译
if (LibAppVars.DO.Flag > 0)              // DO工作使能标志
{
    tick = APP_GetSubTick(LibAppVars.DO.Scan_t); // 读取和上次采集
                                                 //间隔时间，单位ms
    if (tick >= DO_SCAN_T) // 计算和上次采集时间间隔大于等于设置值
    {
        LibAppVars.DO.Scan_t += tick;  // 记录本次采集时间
        DO_EventProc(tick);            // DO输出事件处理
    }
}
else
{
    LibAppVars.DO.Scan_t = LibAppVars.Tick; // 记录本次采集时间
}
#endif
```

3. KEY按键输入编程说明

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

(1) 在IO配置文件XXXX_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

(2) 在功能配置文件XXXX_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+查询读取模式
                   // 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：

```
void Key_UserProcess(INT8U status, INT32U key, INT32U t)
{
    .... ...
    switch (id)
    {
        #ifdef KEY1
        case KEY1_ID:
          if (status == KEY_STA_DOWN) // 按键(第1次)按下
         {   }
         else if (status == KEY_STA_CDOWN) // 按键持续按下
         {   }
         else if (status == KEY_STA_UP) // 按键抬起
         {   }
        break;
        #endif
        #ifdef KEY2
        case KEY2_ID:
          if (status == KEY_STA_DOWN) // 按键(第1次)按下
         {   }
         else if (status == KEY_STA_CDOWN) // 按键持续按下
         {   }
         else if (status == KEY_STA_UP) // 按键抬起
         {   }
        break;
       #endif
         ... ...
    }
}
```

(6) 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

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

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

```
#define SW_NUM    2    // 定义KEY按键数量
#define SW1    PF0
#define SW2    PF1
```

定义2个SW，分别对应硬件PF0和PF1端口

(2) 在功能配置文件XXXX_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+查询读取模式双模式；
                  // 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
```

(3) 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。

(4) 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。

(5) 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SW_UserProcess()，用户请在这个函数根据SW值做相应的处理：

```
void SW_UserProcess(INT8U id, INT8U val)
{
    .... ...
    switch (id)
    {
        #ifdef SW1
        case SW1_ID:
            if (val == ON) { } // SW1由OFF拨到ON位置
            else { }           // SW1由ON拨到OFF位置
            break;
        #endif
        #ifdef SW2
        case SW2_ID:
            if (val == ON) { } // SW2由OFF拨到ON位置
            else { }           // SW2由ON拨到OFF位置
            break;
        #endif
         ... ...
    }
}
```

(6) 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

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1~8，硬件模块不同，x值也不同

以下说明以工控板EMB8602为例。

(1) 在IO配置文件XXXX_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。

(2) 在功能配置文件XXXX_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_RX_TIMEOUT   1000    // 定义接收超时时间，单位us;
#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 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，不可以太大;
#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，不可以太大;
#endif
```

用户根据实际设计需要修改以上设置参数。

(3) 在../libapp/uart_app.c 中查看 Uart_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../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)     // 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(LibAppVars.Uart[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
        {
            LibAppVars.Uart[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
      }
    ... ...
  }
```

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。

(5) 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 ((DEBUG_APP_EN > 0)&&(APP_UART1_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)
    #if (AT_EN > 0)
    if (LibAppVars.Para.ATFlag&AT_UART_FLAG)
                                        // 判断AT配置参数是否允许打印输出
    #endif
    {
        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
    // 用户在这里处理UART1接收到的数据：数据指针p，数据长度len
    //
    #if (APP_UART1_TEST_EN > 0)
    // 测试功能：将数据在原样发送回去
    Uart_UserSendData(UART1_ID, p, len, 1, 0);
    #endif
#endif
}
```

注意：Uart_UserSendData这个函数是发送数据函数；

(6) UART发送数据函数在../source/comfun.c中的Uart_UserSendData，具体代码如下

```
INT32S Uart_UserSendData(INT8U id, INT8U *p, INT16U len, INT8U num, INT16U t)
{
  INT8U i;
  LIBAPP_OUTPUTDATA_MSG *pTxData;  // 发送消息数据指针

  pTxData = APP_GetFreeOutputDataMsg(); // 申请发送缓存
  if (pTxData > 0)                      // 发送缓存有效
  {
      pTxData->Flag = LIBAPP_MSG_OK_FLAG;      // 设置信息有效标志
      pTxData->Type = LIBAPP_DATA_UART_TYPE;  // 设置UART数据类型
      pTxData->id = id;                        // 设置UART ID
      pTxData->len = len;                      // 设置要发送的数据长度
      pTxData->SerialNum = LibAppVars.SerialNum++;     // 设置流水号
      pTxData->num = num;                              // 设置发送次数
      pTxData->t = t;                                  // 设置发送间隔
      for (i=0; i<len; i++)                 // 拷贝发送的数据到发送缓存
      {
          pTxData->Data.pbuf[i] = p[i];
      }
      APP_UserOutputDataMsg(pTxData);                 // 发送数据消息

      return ERR_TRUE;
  }
  else
  {
      return ERR_FALSE;
  }
}
```

用户需要通过串口发送数据请调用这个函数。

6. CAN收发数据编程说明

关键词定义如下：CANx_EN、CANx_MODE、CANx_RXMODE、CANx_SCAN_T、CANx_IDE、CANx_BAUD、CANx_RTR、CANx_RXBUF_SIZE、CANx_TXBUF_SIZE、CANx_TX、CANx_RX

注意：x范围是1~2，硬件模块不同，x值也不同

以下说明以工控板EMB8602的CAN1为例。

(1) 在IO配置文件XXXX_IOConfig.h中定义：参考如下：

```
#define CAN1_REMAP   CAN_REMAP_2     // CAN1重映射2
#define CAN1_TX     PD1     // CAN1设置TX管脚
#define CAN1_RX     PD0     // CAN1设置RX管脚
```

注意：一般CANx_REMAP、CANx_TX、CANx_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。

(2) 在功能配置文件XXXX_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  // 0,正常模式; 1, 环回模式(用于调试); 2, 静默模
                       //式(用于调试); 3, 环回/静默模式(用于调试);
#define CAN1_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
#endif
```

用户根据实际设计需要修改以上设置参数。

(3) 在../libapp/uart_app.c 中查看 CAN_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../config/can_config.h配置滤波器设置

```
// 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是第0组，bit27是第27组，  0：过滤器位宽为2个16位；
//1：过滤器位宽为单个32位。 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；  用户可以不用修改这个配置：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)
//-----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         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_FOR1          0x00000001
#define CAN1_FOR2          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_FOR1          0x000000FF    // 标识符ID值
#define CAN1_FOR2          0x0FFFFF00    // 掩码设置
#endif
#endif
//-----过滤器组1设置
... ...
//-----过滤器组13设置
... ...
#endif
```

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_FOR1/2~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

(5) 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(LibAppVars.CAN[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
        {
          LibAppVars.CAN[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。

(6) 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)
{
    INT16U i;
  for (i=0; i<Num; i++)           // 循环处理Num个CAN消息
    {
    // 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
    //
    #if ((DEBUG_APP_EN > 0)&&(APP_CAN1_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)
    #if (AT_EN > 0)
    if (LibAppVars.Para.ATFlag&AT_CAN_FLAG)//判断AT配置参数是否允许打印
                                           //输出
    #endif
    {
        printf("AT+CAN1=RX[%d,%d]:%02x %02x %02x %02x %02x %02x %02x %02x
        \r\n",pRxMsg[i].ID,pRxMsg[i].DLC,pRxMsg[i].Data[0],pRxMsg[i].Data[1],
        pRxMsg[i].Data[2], pRxMsg[i].Data[3], pRxMsg[i].Data[4], pRxMsg[i].Data[5],
        pRxMsg[i].Data[6], pRxMsg[i].Data[7]);
    }
    #endif
  }

  #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
  CAN_UserSendData(CAN1_ID, (CAN_TX_MSG *)pRxMsg, Num, 1, 0);
  #endif
}
```

(7) CAN发送数据函数在../source/comfun.c中的CAN_UserSendData，具体代码如下

```
INT32S CAN_UserSendData(INT8U id, CAN_TX_MSG *pTxMsg, INT16U len,
                        INT8U num, INT16U t)
{
    INT8U i, j;
    LIBAPP_OUTPUTDATA_MSG *pTxData;  // 发送消息数据指针
    pTxData = APP_GetFreeOutputDataMsg();          // 申请发送缓存
    if ((pTxData > 0)&&(len < LIBAPP_MAX_CANTX_SIZE))  // 发送缓存有效
```

```
    {
        pTxData->Flag = LIBAPP_MSG_OK_FLAG;      // 设置信息有效标志
        pTxData->Type = LIBAPP_DATA_CAN_TYPE;    // 设置CAN数据类型
        pTxData->id = id;                        // 设置CAN ID
        pTxData->len = len;                      // 设置要发送的消息个数
        pTxData->SerialNum = LibAppVars.SerialNum++;     // 设置流水号
        pTxData->num = num;                              // 设置发送次数
        pTxData->t = t;                                  // 设置发送间隔
        for (i=0; i<len; i++)                // 拷贝发送的数据到发送缓存
        {
            pTxData->Data.CanTx[i].IDE = pTxMsg[i].IDE;
            pTxData->Data.CanTx[i].RTR = pTxMsg[i].RTR;
            pTxData->Data.CanTx[i].ID = pTxMsg[i].ID;
            pTxData->Data.CanTx[i].DLC = pTxMsg[i].DLC;
            for (j=0; j<pTxMsg[i].DLC; j++)
            {
                pTxData->Data.CanTx[i].Data[j] = pTxMsg[i].Data[j];
            }
        }
        APP_UserOutputDataMsg(pTxData);              // 发送数据消息
        return ERR_TRUE;
    }
    else
    {
        return ERR_FALSE;
    }
}
```

用户需要通过CAN发送数据请调用这个函数。

7. ADC(AI)输入编程说明

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。

(1) 在IO配置文件XXXX_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。

(2) 在功能配置文件XXXX_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_SAMP21TOUS  // 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_T3TRGO  // 选择AD采样触发源, 固定选择
                                           // ADC_EXTSEL_T3TRGO
#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; 【暂不支持: 10, 正弦波-10V~+10V; //11, 正
弦波-5V~+5V; 12, 正弦波-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
```

(3) 在../libapp/adc_app.c 中查看 ADC_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../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(LibAppVars.ADC[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
    {
        LibAppVars.ADC[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(LIBAPP_INPUTDATA_MSG *pDataMsg)
{
    INT16U i, *pData;
    pData = (INT16U *)pDataMsg->pData; // 取得数据缓存指针
    #if (MODBUS_SLAVE_EN > 0)          // Modbus设备模式使能
    for (i=0; i<pDataMsg->len; i++)
    {
        ModbusInputReg[i] = pData[i];  // 将AD转换值写入Modbus输入寄存器
    }
```

```
        #endif
         // 条件编译使能接收数据打印输出
        #if ((DEBUG_APP_EN == 1)&&(APP_ADC_DEBUG_EN > 0))
        #if (AT_EN > 0)
       if(LibAppVars.Para.ATFlag&AT_AI_FLAG)//判断AT配置参数是否允许打印输出
        #endif
        {
            printf("AT+AI=%d", ADC_CHNUM);
            #if (AI1_EN == 1)   // 判断AI1使能
            printf(",%d", pData[AI1_ID]);
            #endif
            ... ...
        }
      }
```

注意：pData，ADC采样值数据指针，内部数据按pData[AI1_ID]~pData[AI10_ID]顺序排列。pDataMsg->len是数据个数。

8. DAC(AO)输出编程说明

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。

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

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

(2) 在功能配置文件XXXX_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
```

(3) 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。

(4) 在../libapp/adc_app.c中查看DAC_AppTest函数，这个函数是操作DAC输出的测试例程，被

在../source/user_testapp.c的User_AppTestProc()中被调用。以DAC1为例说明如下：

```
void DAC_AppTest(void)
{
    INT32U  flag, tick;
    #if (DAC1_EN > 0)
    tick = APP_GetSubTick(LibAppVars.DAC[DAC1_ID].Scan_t);  // 读取和上次操
                                                            // 作间隔时间, 单位ms
    if (tick >= DAC1_SCAN_T)      // 计算和上次操作时间间隔大于等于设置值
```

```
{
    LibAppVars.DAC[DAC1_ID].Scan_t += tick;     // 记录本次扫描时间
    // 在手动输出模式, 间隔DAC1_SCAN_T时间DAC输出增加10%,
    // 当输出值达到最大值时，重新从0开始输出
    #if (DAC1_MODE == DAC_MODE_MTOUT)            // DAC1手动输出
    LibAppVars.DAC[DAC1_ID].val += 409;          // 增加10%
    if (LibAppVars.DAC[DAC1_ID].val>4095)        // 判断超过4095
    {
        LibAppVars.DAC[DAC1_ID].val = 0;         // 重新设置为0
    }
    DAC_Write(DAC1_ID, LibAppVars.DAC[DAC1_ID].val);  // 执行DAC输出

    #if (DAC1_MODE == DAC_MODE_ATOUT_N) // 模式1, 连续输出N个波形后停止
      // 读取DAC状态
      flag = DAC_AppCtrl(DAC1_ID, LIBAPP_DAC_CMD_STATUS, 0, 0);
      if (flag == ERR_TRUE)                  // 如果状态正确则执行新的操作
      {
          flag = DAC_AppConfigWave(DAC1_ID, DAC1_TEST_WARE, 4095,
                    DAC1_TXBUF_SIZE, 50);  // 设置波形数据
           if (flag == ERR_TRUE)
           {
              // 输出100个10HZ的波形
               DAC_AppCtrl(DAC1_ID, LIBAPP_DAC_CMD_START, 10, 100);
            }
      }else{}
    #endif

      #if (DAC1_MODE == DAC_MODE_ATOUT)    // 模式2, 持续输出不停止
      // 读取DAC状态
```

```
flag = DAC_AppCtrl(DAC1_ID, LIBAPP_DAC_CMD_STATUS, 0, 0);
if (flag == ERR_TRUE)                              // DAC停止操作
{
   flag = DAC_AppConfigWave(DAC1_ID, DAC1_TEST_WARE, 4095,
                            DAC1_TXBUF_SIZE, 50); // 设置波形数据
   if (flag == ERR_TRUE)
   {
        // 持续输出10HZ的波形
       DAC_AppCtrl(DAC1_ID, LIBAPP_DAC_CMD_START, 10, 0);
      }
}
else  // 正在运行
{
    if (LibAppVars.DAC[DAC1_ID].Ware.Freq == 10)
    {     // 持续输出50HZ的波形
          DAC_AppCtrl(DAC1_ID, LIBAPP_DAC_CMD_FREQ, 50, 0);
     }
     else
     {     // 持续输出10HZ的波形
           DAC_AppCtrl(DAC1_ID, LIBAPP_DAC_CMD_FREQ, 10, 0);
      }
 }
 #endif
```

9. FCLK脉冲输入编程说明

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。

(1) 在IO配置文件XXXX_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
```

(2) 在功能配置文件XXXX_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
```

(3) 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。

(4) 在../libapp/fclk_app.c中查看FCLK_ReadProc处理函数：

```
void FCLK_ReadProc(void)
{
   INT32U tick;
  //----FCLK1---------------------------------------
   #if (FCLK1_EN > 0)                  // 条件编译FCLK1使能
   // 读取和上次采集的间隔时间, 单位ms
  tick = APP_GetSubTick(LibAppVars.FCLK[FCLK1_ID].Scan_t);
  if (tick >= FCLK1_SCAN_T)      // 计算和上次采集时间间隔大于等于设置值
  {
```

```
            LibAppVars.FCLK[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。

(5) 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)
    {
        #if (FCLK1_MODE == FCLK_MODE_COUNT)  // 计数模式(1路, CH1输入有效);
        if (pData->Mode == FCLK_MODE_COUNT)
        {
```

```
    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
       printf("AT+FCLK1=N,%d\r\n",  pData->cnt);
    }
}
#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)
{
     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
   printf("AT+FCLK1=E,%d,%d\r\n", pData->QEICnt, pData->QEICount);
      }
 }
 #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
         #if (FCLK1CH1_EN > 0)
```

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


                    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
```

10. PWM脉冲输出编程说明

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。

(1) 在IO配置文件XXXX_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
```

(2) 在功能配置文件XXXX_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: 输出PWM, 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
#endif
```

(3) 在../libapp/pwm_app.c中查看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的User_AppTestProc()中被调用。以PWM1为例说明如下：

```
void PWM_AppTest(void)
{
    INT32U tick, PWM_IDFlag = 0;
    // 读取和上次采集的间隔时间, 单位ms
    tick = APP_GetSubTick(LibAppVars.PWM[PWM1_ID].Scan_t);
    if (tick >= PWM_SCAN_T)      // 计算和上次采集时间间隔大于等于设置值
   {
        LibAppVars.PWM[PWM1_ID].Scan_t += tick;  // 记录本次扫描时间
        // 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_ModifyFreq(PWM1_ID, PWM_TestFreq);  // 修改PWM脉冲频率
        #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
        // 判断输出脉冲是否完成
        if (PWM_Ctrl(PWM1_ID, CMD_PWM_STATUS, 0) == PWM1CH_EN)
```

```
        {
            PWM_Start(PWM1_ID, PWM1CH_EN, PWM_TEST_FREQ,
                    PWM_TEST_RATE, PWM_TEST_NUM); // 启动新的脉冲输出
        }
        #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, 持续输出修改
PWM脉冲占空比
          PWM_ModifyRate(PWM1_ID, PWM1CH_EN, PWM_TestRate);
            #endif

          #if (PWM1_MODE==PWM_WRITE)     // DMA输出控制模式
          PWM_Write(PWM1_ID, (PWM_WRITE_PARA*)&LibAppVars.
           PWM1WritePara.Flag);  // 按配置PWM1WritePara输出脉冲
          PWM_IDFlag |= PWM1_IDFLAG;     // 设置标志
          #endif
            PWM_DIRENA_Ctrl(PWM1_ID);  // 控制DIR1和ENA1
          #endif

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

11. TIM 定时器编程说明

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

x 范围：1~14；注意：定时器是芯片内部功能，无管脚定义。

(1) 在功能配置文件 XXXX_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
```

(2) 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。

(3) 在功能配置文件 XXXX_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。

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_ReadCount 函数：

```
Timer_Ctrl(TIM1_ID, CMD_TIM_ENA, 0);    // 使能定时器
Delay_ms(20);                            // 延时 20ms
Timer_Ctrl(TIM1_ID, CMD_TIM_DIS, 0);    // 关闭定时器
```

```
cnt = Timer_Ctrl(TIM1_ID, CMD_TIM_READ, 0); // 读取 20ms 计数值
#if (APP_TIM_DEBUG_EN == 1)
if (LibAppVars.Para.ATFlag&AT_TIM_FLAG)
{
    printf("AT+TIM1=%d\r\n", cnt);          // 打印输出计数值
}
#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

关键词定义如下: EXTIx_EN、EXTIx_IO、EXTIx_MODE、EXTI16_PVD_EN、EXTI16_PVD_MODE、EXTI17_RTCAlarm_EN 、 EXTI17_RTCAlarm_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;

(1) 在功能配置文件 XXXX_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, PI0 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。

(2) 在../libapp/exti_app.c 中查看 EXTI_AppInit 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。

(3) 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

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

(1) 在IO配置文件XXXX_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
```

(2) 在功能配置文件 XXXX_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
```

(3) 在../libapp/spi_app.c 中查看 SPI_AppInit 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。

(4) 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 DO_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

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

(1) 在IO配置文件XXXX_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
```

(2) 在功能配置文件 XXXX_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
```

(3) 在../libapp/I²C_app.c 中查看 I²C_AppInit 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。

(4)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../driverc 中的 CH455_Read()函数:在这个函数中用户通过 I²C3 总线的 I²C_Read 读取按键值。

15. RTC 时钟编程说明

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

（1）在功能配置文件 XXXX_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
```

（2）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在../source/user_testconfig.h 中

做如下定义：

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

（3）在../source/user_testapp.h 中的 User_AppTestProc()中会调用 RTC_AppTest()函数，按 RTC_SCAN_T 扫描间隔读取 RTC 时间并打印输出。RTC_AppTest()函数在../libapp/rtc_app.c 中。

16. EEPROM 读写编程说明

关键词定义如下：EEPROM_EN、EEPROM_DEVICE、EEPROM_FREQ

(1) 在功能配置文件 XXXX_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，在 IO 配置文件 XXXX_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
```

(2) 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。

(3) 在../source/user_testconfig.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     1000      // 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
```

(4) 在../source/user_testapp.h 中的 User_AppTestStart()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

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

(1) 在IO配置文件XXXX_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。

(2) 在功能配置文件 XXXX_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    1     // SPI FLASH 操作方式：1，利用 FATFS 文件系统进行读写；
                                // 0，用 SPI FLASH 读写函数进行操作；注意:2 种操作方式只能选择一种
#define SPIFLASH_TYPE     W25QXX    // SPI FLASH 类型设置：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
```

(3) 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。

(4) 在../source/user_testconfig.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   2040   // W25QXX FLASH 测试起始扇区
#define W25QXX_TEST_SECTOR_NUM     8      // 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
```

(5) 在../source/user_testapp.h 中的 User_AppTestStart()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

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。

(1) 在IO配置文件XXXX_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
```

(2) 在功能配置文件XXXX_Config.h中定义，参考如下：

```
// TCPIP(LWIP)协议栈配置
#define LWIP_EN       1    // 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
// 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
// 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，注意使能这个，请使能MODBUS_SLAVE_EN
 #define TCP_SERVER_LOCAL_PORT    5000  // 本地端口号
 #define LWIP_MAX_TCP_SERVER_LINK_NUM 8 // 最大连接TCP客户端个数,最大设置8
 #endif

// 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    5001     // 本地端口号
#define LWIP_MAX_UDP_SERVER_LINK_NUM 16 // 最大连接UDP客户端个数,最大16个
#endif

// 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 LWIP_MAX_TCP_DSC_NUM  4 // 远端TCP服务器个数

#define LWIP_TCP_DSC1_IP       "192.168.1.200"  // 远端服务器1 IP
#define LWIP_TCP_DSC1_PORT     5001             // 远端服务器1端口号
#define LWIP_TCP_DSC2_IP       "192.168.1.201"  // 远端服务器2 IP
#define LWIP_TCP_DSC2_PORT     5002             // 远端服务器2端口号
#define LWIP_TCP_DSC3_IP       "192.168.1.202"  // 远端服务器3 IP
#define LWIP_TCP_DSC3_PORT     5003               // 远端服务器3端口号
#define LWIP_TCP_DSC4_IP       "192.168.1.203"  // 远端服务器4 IP
#define LWIP_TCP_DSC4_PORT     5004             // 远端服务器4端口号
#endif

// 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 LWIP_MAX_UDP_DSC_NUM    4        // 远端UDP服务器个数

#define LWIP_UDP_DSC1_IP    "192.168.1.200" // 远端服务器1 IP
#define LWIP_UDP_DSC1_PORT   5001          // 远端服务器1端口号
#define LWIP_UDP_DSC2_IP     "192.168.1.201" // 远端服务器2 IP
#define LWIP_UDP_DSC2_PORT   5002          // 远端服务器2端口号
#define LWIP_UDP_DSC3_IP     "192.168.1.202"  // 远端服务器3 IP
#define LWIP_UDP_DSC3_PORT   5003           // 远端服务器3端口号
#define LWIP_UDP_DSC4_IP     "192.168.1.203"  // 远端服务器4 IP
#define LWIP_UDP_DSC4_PORT   5004           // 远端服务器4端口号
#endif

// 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_SERVER_LOCAL_PORT    69  // 本地端口号
#define LWIP_MAX_TFTP_SERVER_LINK_NUM    16 // 最大连接TFTP客户端个数，最大16个
#endif

// 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；
// 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
```

(3) 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、支持TFTP通信、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：

Client_TCP_Thread(): TCP客户端模式通信任务

Client_UDP_Thread(): UDP客户端模式通信任务

tcpecho_thread(): TCP服务端模式通信任务

udpecho_thread(): UDP服务端模式通信任务

用户不需要修改该代码，网络接收的数据通过

LibAppVars.Register.APP_InputDataCallback回调函数发送出来。

(4) 网络接收数据被发送到../source/user_netwifiapp.c中的各个函数中，具体函数介绍如下：

TCP客户端模式接收数据处理函数：

```
void NET_TCPClientDataProcess(LIBAPP_INPUTDATA_MSG *pDataMsg)
{
    INT16U i;
    INT8U *pbuf;
    pbuf = (INT8U *)pDataMsg->pData;
    // 用户在这里处理网络接收到的数据：接收数据在pbuf[]中,
    // 数据长度是pDataMsg->len, 请根据不同的数据来源做相应处理
    switch(pDataMsg->id)
    {
        case NET_DSC1_ID:           // 处理数据服务器1发来的数据
            //NET_UserDsc1Process(pbuf, pDataMsg->len);
            break;
        case NET_DSC2_ID:           // 处理数据服务器2发来的数据
            //NET_UserDsc2Process(pbuf, pDataMsg->len);
            break;
        case NET_DSC3_ID:           // 处理数据服务器3发来的数据
            //NET_UserDsc3Process(pbuf, pDataMsg->len);
            break;
        case NET_DSC4_ID:           // 处理数据服务器4发来的数据
            //NET_UserDsc4Process(pbuf, pDataMsg->len);
            break;
        default:
            break;
```

```
    }
    #if (APP_NET_TEST_EN > 0)
    // 测试功能: 将数据在原样发送回去
     NET_UserSendData(LIBAPP_DATA_TCP_CLIENT_TYPE, pDataMsg->id, pbuf,
                      MIN(pDataMsg->len, LIBAPP_MAX_TXBUF_SIZE), 1, 0);
    #endif
  }
```

UDP客户端模式接收数据处理函数：

```
void NET_UDPClientDataProcess(LIBAPP_INPUTDATA_MSG *pDataMsg)
{
  INT16U i;
  INT8U *pbuf;
  pbuf = (INT8U *)pDataMsg->pData;
  // 用户在这里处理网络接收到的数据，接收数据在pbuf[]中,
  // 数据长度是pDataMsg->len, 请根据不同的数据来源做相应处理
    switch(pDataMsg->id)
    {
        case NET_DSC1_ID:           // 处理数据服务器1发来的数据
          //NET_UserDsc1Process(pbuf, pDataMsg->len);
          break;
        case NET_DSC2_ID:           // 处理数据服务器2发来的数据
          //NET_UserDsc2Process(pbuf, pDataMsg->len);
          break;
        case NET_DSC3_ID:           // 处理数据服务器3发来的数据
          //NET_UserDsc3Process(pbuf, pDataMsg->len);
          break;
        case NET_DSC4_ID:           // 处理数据服务器4发来的数据
          //NET_UserDsc4Process(pbuf, pDataMsg->len);
```

```
            break;
        default:
            break;
    }
    #if (APP_NET_TEST_EN > 0)
    // 测试功能: 将数据在原样发送回去
     NET_UserSendData(LIBAPP_DATA_UDP_CLIENT_TYPE, pDataMsg->id, pbuf,
                      MIN(pDataMsg->len, LIBAPP_MAX_TXBUF_SIZE), 1, 0);
    #endif
  }
}
```

TCP服务器模式接收数据处理函数:

```
void NET_TCPServerDataProcess(LIBAPP_INPUTDATA_MSG *pDataMsg)
{
  INT16U i;
  INT8U *pbuf;
  pbuf = (INT8U *)pDataMsg->pData;
  // 用户在这里根据不同设备发来的数据做处理,接收数据在pbuf[]中,
  // 数据长度是pDataMsg->len, 请根据不同的数据来源做相应处理
  switch (pDataMsg->id)
  {
      case NET_CLIENT1_ID:  // 处理客户端1发来数据
          break;
      case NET_CLIENT2_ID:  // 处理客户端2发来数据
          break;
      case NET_CLIENT3_ID:  // 处理客户端3发来数据
```

```
            break;
        case NET_CLIENT4_ID:  // 处理客户端4发来数据
            break;
        case NET_CLIENT5_ID:  // 处理客户端5发来数据
            break;
        case NET_CLIENT6_ID:  // 处理客户端6发来数据
            break;
        case NET_CLIENT7_ID:  // 处理客户端7发来数据
            break;
        case NET_CLIENT8_ID:  // 处理客户端8发来数据
            break;
        default:
            break;
    }
    #if (APP_NET_TEST_EN > 0)
    // 测试功能: 将数据在原样发送回去
    NET_UserSendData(LIBAPP_DATA_TCP_SERVER_TYPE, pDataMsg->id, pbuf,
                     MIN(pDataMsg->len, LIBAPP_MAX_TXBUF_SIZE), 1, 0);
    #endif
   }
```

UDP服务器模式接收数据处理函数

```
  void NET_UDPServerDataProcess(LIBAPP_INPUTDATA_MSG *pDataMsg)
  {
    INT16U i;
    INT8U *pbuf;
    pbuf = (INT8U *)pDataMsg->pData;
```

```
    // 用户在这里根据不同设备发来的数据做处理,接收数据在pbuf[]中,
    // 数据长度是pDataMsg->len, 请根据不同的数据来源做相应处理
    switch (pDataMsg->id)
    {
        case NET_CLIENT1_ID:  // 处理客户端 1 发来数据
            Break;
        case NET_CLIENT2_ID:  // 处理客户端 2 发来数据
            break;
        case NET_CLIENT3_ID:  // 处理客户端 3 发来数据
            break;
        case NET_CLIENT4_ID:  // 处理客户端 4 发来数据
            break;
        case NET_CLIENT5_ID:  // 处理客户端 5 发来数据
            break;
        case NET_CLIENT6_ID:  // 处理客户端 6 发来数据
            break;
        case NET_CLIENT7_ID:  // 处理客户端 7 发来数据
            break;
        case NET_CLIENT8_ID:  // 处理客户端 8 发来数据
            break;
        default:
            break;
    }
    #if (APP_NET_TEST_EN > 0)
    // 测试功能: 将数据在原样发送回去
   NET_UserSendData(LIBAPP_DATA_UDP_SERVER_TYPE, pDataMsg->id, pbuf,
    MIN(pDataMsg->len, LIBAPP_MAX_TXBUF_SIZE), 1, 0);
     #endif
}
```

19. USB 通信编程说明

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

(1) 在IO配置文件XXXX_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。

(2) 在功能配置文件 XXXX_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 FALSH 作为存储介质实现 USB Mass Storage 功能,
// 计算机可以通过 USB 口直接读取 SD 卡、SPI FLASH 或者 NAND FLASH 的文件
// 注: USB_VCP_EN 和 USB_MSC_EN 不能同时使能
#define USB_MSC_EN        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
```

(3) 在../libapp/usb_app.c 中查看 USBH_AppInit 和 USBD_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../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.USBD.Scan_t += tick;              // 记录本次扫描时间
        flag = USBD_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 = USBD_Ctrl(USB_ID, CMD_USBD_GetCharsRxBuf,0);     // 读取接收数据长度
            if ((len == LibAppVars.USBD.len)&&(len>0))
            {
                USBD_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.USBD.len -= len;
            }
            else
```

```
            {
                LibAppVars.USBD.len = len;
            }
        }
    }
    #endif
```

(5) 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(LIBAPP_INPUTDATA_MSG *pDataMsg)
{
    INT16U i;
    INT8U *pbuf;
    pbuf = (INT8U *)pDataMsg->pData;
    // 用户在这里处理USB接收到的数据，数据在缓存pbuf中，数据长度pDataMsg->len

    #if (APP_USB_TEST_EN > 0)
    // 测试功能：将数据在原样发送回去
    USB_UserSendData(0, p, len, 1, 0);
    #endif
  }
```

(6) 在USB主机模式下，插入U盘进行U盘读写测试。

在../source/user_testconfig.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
```

```
// 可以在下面更改测试读写数据包大小
#define APP_UDISK_FILE_FBUF_SIZE  700  // 测试数据包大小定义，至少大于等于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

```
void User_FileAppTest(void)
{
    ... ...
    // U 盘文件读写测试
    #if (APP_UDISK_FILE_TEST_EN > 0)
    #if ((USB_HOST_EN > 0)&&(UDISK_EN > 0)) // U 盘使能
      if ((LibAppVars.UDisk.Status ==
          FILE_IDLE)&&((UserVars.Flag&USER_UDISK_FILE_TEST_OK_FLAG)==0))
    {
        #if (APP_UDISK_FILE_TEST_MODE == 0)
          FileApp_Test(USB_DISK, LibAppVars.buf, APP_UDISK_FILE_FBUF_SIZE,
          APP_UDISK_FILE_PAGE_NUM);
        #else
          APP_FileCtrl_Test(USB_DISK, LibAppVars.buf, APP_UDISK_FILE_FBUF_SIZE,
          APP_UDISK_FILE_PAGE_NUM);
        #endif
        UserVars.Flag |= USER_UDISK_FILE_TEST_OK_FLAG;  // 用户 U 盘文件读写测试完成标志
    }
    else
    {
```

```
            if (LibAppVars.UDisk.Status != FILE_IDLE)
            {
                UserVars.Flag &= ~USER_UDISK_FILE_TEST_OK_FLAG;  // 清除标志
            }
        }
        #endif
            #endif
        ... ...
    }
```

20. SD 卡读写编程说明

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

(1) 在IO配置文件XXXX_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。SD 卡采用 SPI1 接口控制，可在本文件中查看 SPI1 端口定义。

(2) 在功能配置文件 XXXX_Config.h 中定义，参考如下：

```
#define SDCARD_EN               1           // SD 卡使能：1, 使能;  0, 关闭;
#define                                                          FATFS_EN
(SDCARD_EN|(SPIFLASH_EN&SPIFLASH_MODE)|UDISK_EN)
                                        // 文件系统使能：1, 使能;  0, 关闭;
#define FATFS_SCAN_T            10    // 设置定时扫描时间间隔, 单位: ms
```

(3) 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。

(4) 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。

(5) 以下测试测序，在插入 SD 后，会对 SD 卡进行读写测试。

在../source/user_testconfig.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   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_SD_FILE_FBUF_SIZE   700   // 测试数据包大小定义，至少大于等于 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    10    // 设置读写测试数据包数量
#endif
```

在../source/user_testapp.c 中查看 SD 卡读写测试程序

```
void User_FileAppTest(void)
{
    ... ...
    // SD 卡文件读写测试
    #if (APP_SD_FILE_TEST_EN > 0)
    #if (SDCARD_EN > 0)             // SD 卡使能
    if                              ((LibAppVars.SD.Status                          ==
FILE_IDLE)&&((UserVars.Flag&USER_SD_FILE_TEST_OK_FLAG)==0))
    {
      #if (APP_SD_FILE_TEST_MODE == 0)
      FileApp_Test(SD_DISK,         LibAppVars.buf,         APP_SD_FILE_FBUF_SIZE,
APP_SD_FILE_PAGE_NUM);
      #else
      APP_FileCtrl_Test(SD_DISK,        LibAppVars.buf,         APP_SD_FILE_FBUF_SIZE,
APP_SD_FILE_PAGE_NUM);
      #endif
      UserVars.Flag |= USER_SD_FILE_TEST_OK_FLAG;  // 用户 SD 卡文件读写测试完成标志
    }
    else
    {
        if (LibAppVars.SD.Status != FILE_IDLE)
        {
```

```
                UserVars.Flag &= ~USER_SD_FILE_TEST_OK_FLAG;  // 清除标志
            }
        }
        #endif
        #endif
    ... ...
}
```

21. NAND FLASH 读写编程说明

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

（1）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 XXXX_IOConfig.h中不在做定义，用户无需关注。

（2）在功能配置文件 XXXX_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  ：256/512/1024/2048/4096/8192 字节可选
    #define NFLASH_BLOCK_NUM      2048  // Nand Flash 总块数
    #define NFLASH_BLOCK_SIZE     64   // Nand Flash 每个块包含页数
    #define NFLASH_PAGE_SIZE      2048  // Nand Flash 页大小
    #define NFLASH_MAX_BAD_BLOCK 64     // Nand Flash 最大坏块数，这个部分用于替换数据区坏块
```

（3）在../libapp/libapp.c 中查看 NFlash_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../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。

(5) 以下测试测序，板子上电后，会对 Nand Flash 进行读写测试。

在../source/user_testconfig.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，至少大于等于 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
```

在../source/user_testapp.c 中查看 Nand Flash 读写测试程序

```
void User_FileAppTest(void)
{
    ... ...
    // NAND FLASH 文件读写测试
    #if (APP_NFLASH_FILE_TEST_EN > 0)
    #if (NFLASH_EN>0) // NAND FLASH 使能
    if ((LibAppVars.NFlash.Status ==
FILE_IDLE)&&((UserVars.Flag&USER_NFLASH_FILE_TEST_OK_FLAG)==0))
    {
        #if (APP_NFLASH_FILE_TEST_MODE == 0)
        FileApp_Test(NFLASH_DISK, LibAppVars.buf, APP_NFLASH_FILE_FBUF_SIZE,
                     APP_NFLASH_FILE_PAGE_NUM);
        #else
        APP_FileCtrl_Test(NFLASH_DISK, LibAppVars.buf, APP_NFLASH_FILE_FBUF_SIZE,
                          APP_NFLASH_FILE_PAGE_NUM);
        #endif
        UserVars.Flag |= USER_NFLASH_FILE_TEST_OK_FLAG; //用户 NAND FLASH 文件读写测试完成标志
    }
    #endif
```

```
        #endif

        ... ...

    }
```

22. Modbus 主机通信编程说明

关键词定义如下：MODBUS_EN、MODBUS_MODE、MODBUS_CH、MODBUS_ID、MODBUS_NUM、MODBUS_TIMEOUT、MODBUS_RXSCAN_T、MODBUS_BUF_SIZE、MODBUS_SCAN_T

(1) 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_NUM          1           // MODBUS 操作设备的数量，默认 1
#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       1000        // 设置定时扫描时间间隔，单位：ms
```

(2) 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。

(3) 在../source/user_modbusapp.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

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

(1) 在功能配置文件 modbus_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;
#define MODBUS_SLAVE_MODE        0            // MODBUS 从机通信模式：0，RTU；  1，ASCII
码(暂不支持);
#define MODBUS_SLAVE_CH          UART4_ID     // MODBUS 从机通信通道：0：UART1_ID，1:
UART2_ID,
                                              // 2：UART3_ID，3：UART4_ID，4：UART5_ID;
#define MODBUS_SLAVE_ID          1            // MODBUS 从机通信地址码，范围：1~255;
#define MODBUS_COILS_BASEADDR    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         32           // MODBUS 从机最大线圈数量(读写,可用功能
码:1,5,15);
#define MODBUS_MAX_DISINPUT      32           // MODBUS 从机最大离散输入量(只读,可用功能
码:2);
#define MODBUS_MAX_HOLDREG       16   // MODBUS 从机最大保持寄存器(读写,可用功能
码:3,6,16,23)数量; #define MODBUS_MAX_INPUTREG      16          // MODBUS 从机最大输入寄存器
(只读,可用功能码:4)数量;
```

(2) 在../libapp/modbus_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。

(3) 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：

```
void Uart_DataProcess(LIBAPP_INPUTDATA_MSG *pDataMsg)
{
```

```
// 控制板作为 Modbus 设备测试，Modbus Slave 数据处理
#if (MODBUS_SLAVE_EN > 0) // Mobus Slave 模式使能
INT32S flag;
INT16U i;
INT8U *pbuf;
if (LibAppVars.Uart[pDataMsg->id].Flag&UART_MODBUS_SLAVE_FLAG)//判断该 UART 是否应用于 MobusSlave
{
    pbuf = (INT8U *)pDataMsg->pData;
    flag = Modbus_Proc(MODBUS_SLAVE_CH, LibAppVars.Para.ModbusSlaveID, pDataMsg->pData,
                       pDataMsg->len, (MODBUS_PARA *)&ModbusPara.Flag); // Modbus Slave 数据处理
}
#endif
... ...
}
```

注意：Modbus_Proc 这个是解析 Modbus 协议，并自动完成本机寄存器读写。本机寄存器在../config/vars.c

中定义：

```
#if (MODBUS_SLAVE_EN > 0)
MODBUS_DATA ModbusData;                                    // 缓存数据
__attribute__((aligned(4))) INT8U ModbusCoils[(MODBUS_MAX_COILS-1)/8 + 1]; // 输出线圈数组
// 输入离散输入量数组
__attribute__((aligned(4))) INT8U ModbusDisInput[(MODBUS_MAX_DISINPUT-1)/8 + 1];
__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
    MODBUS_PARA ModbusPara;                                    // 参数定义
    #endif
```

用户在应用程序中可以直接给 ModbusCoils、ModbusDisInput、ModbusHoldReg 和 ModbusInputReg 赋值。

24. WIFI 通信编程说明

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

（1）在功能配置文件 XXXX_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.h"
    #define WIFI_UART               UART3_ID // 选择串口：UART1_ID~UART6_ID;
    #define WIFI_SCAN_T             10      // 设置定时扫描时间间隔，单位：ms
    #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_WKMODE                 WIFI_STATION   // 选择 WIFI 模块工作模式：
WIFI_STATION 或 WIFI_AP
    #define WIFI_NETYPE             WIFI_TCP_SERVER  // 选择 WIFI 模块网络通信方式：
                                    //
                                     WIFI_TCP_CLIENT/WIFI_UDP_CLIENT/WIFI_TCP_SERVE
                                     R
    #if         ((WIFI_WKMODE        ==        WIFI_STATION)&&((WIFI_NETYPE        ==
WIFI_TCP_CLIENT)||(WIFI_NETYPE ==              WIFI_UDP_CLIENT)))     //  只 有 在
```

STATION 的 TCP Client 或 UDP Client 才支持透传模式

```
    #define WIFI_TC_EN              0                   // 透传使能: 1, 使能; 0, 关闭
    #endif
    #define WIFI_ROUTER_SSID        "G1604"         // 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_LOCAL_GATEWAY      "192.168.1.1"    // 选择 WIFI 模块本地网关
    #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_SUBNET_MASK        "255.255.255.0"  // 选择 WIFI 模块本地子网掩码

    #define WIFI_DSC_IP             "192.168.1.200"  // 选择远端服务器 IP
    #define WIFI_DSC_PORT           5001             // 选择远端服务器端口号
    #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
    #endif
```

(2) 在../driver/wifi.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。

(3) 在../driver/wifi.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。

(4) WIFI接收数据被发送到../source/user_netwifiapp.c中的WIFI_DataProcess函数中，具体函数如下：

```
void WIFI_DataProcess(LIBAPP_INPUTDATA_MSG *pDataMsg)
{
    INT16U i;
    INT8U *pbuf;
    pbuf = (INT8U *)pDataMsg->pData;
    #if (APP_WIFI_DEBUG_EN == 1)
    printf("AT+WIFI=RX[ID=%c:%d.%d.%d.%d,%d,%d]:%02X",      pDataMsg->id+'1',
```

```
        LibAppVars.WIFI.ip[0],        LibAppVars.WIFI.ip[1],LibAppVars.WIFI.ip[2],
        LibAppVars.WIFI.ip[3], LibAppVars.WIFI.port, pDataMsg->len, pbuf[0]);
        for(i=1; i<MIN(pDataMsg->len, DEBUG_MAX_DATA_LEN); i++)
        {
            printf(" %02x", pbuf[i]);
        }
        printf("\r\n");
        #endif
        // 用户在这里处理WIFI接收到的数据: 接收数据在pbuf[]中, 数据长度是
    pDataMsg->len
        //
        #if (APP_WIFI_TEST_EN > 0)
        // 测试功能: 将数据在原样发送回去
        NET_UserSendData(LIBAPP_DATA_WIFI_TYPE,      pDataMsg->id,      pbuf,
        MIN(pDataMsg->len, LIBAPP_MAX_TXBUF_SIZE), 1, 0);
        #endif
    }
    #endif
```

注意：NET_UserSendData这个函数是WIFI发送数据函数；

(5) 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

关键词定义如下: DEVICE_SHTXX_ID、DEVICE_CH455_ID、CH455_EN、CH455_MODE、CH455_I²C、CH455_SCAN_T、SHTXX_EN、SHTXX_SCAN_T

特别说明：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。

（1）在功能配置文件 device_config.h 中定义，参考如下：

```
// 传感器序号定义
#define  DEVICE_SHTXX_ID     0    // 器件 SHTXX ID 值
#define  DEVICE_CH455_ID     1    // 器件 CH455 ID 值

// CH455 参数配置(4*7 按键驱动芯片)
// 注意:如果 CH455 模式选择 0，则必须使能 I²C3_EN;如果 CH455 模式选择 1，则必须关闭 I²C3_EN;
#if (PRODUCT_TYPE == EMB8628I)
#define CH455_EN            1            // CH455 使能,       1：打开使能，  0：关闭
#else
#define CH455_EN            0            // CH455 使能,       1：打开使能，  0：关闭
#endif
#define CH455_MODE          1            // CH455 模式,       0(CH455_MODE_I²C)：用 I²C 总线控制;
                                         //  1(CH455_MODE_IO)：用 IO(模拟 I²C 总线)控制
#if (CH455_MODE == CH455_MODE_I²C)
#define CH455_I²C    I²C3_ID     // I²C 选择：0(I²C1_ID)，1(I²C2_ID)，2(I²C3_ID)
#else
#define CH455_I²C    I²C2_IO_ID  // IO 模拟 I²C 选择：0(I²C1_IO_ID)，1(I²C2_IO_ID)，2(I²C3_IO_ID)
#endif
#define CH455_SCAN_T        20            // 设置定时扫描时间间隔，单位：ms
// 温湿度参数配置
#define SHTXX_EN            0          // SHTXX 使能，1：打开使能，  0：关闭
#define SHTXX_SCAN_T        20         // 设置定时扫描时间间隔，单位：ms
                                       // 注意这个值最小是 20ms，不可小于 20ms，默认 20 就可以
```

```
// 外部器件使能定义
#define DEVICE_EN       (SHTXX_EN|CH455_EN)   // DEVICE 使能, 1：打开使能， 0：关闭
```

(2) 在../driver/device_app.c 中查看 Device_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:

```
void Device_AppInit(void)
{
    #if (CH455_EN > 0)        // CH455 配置使能
    CH455_APPInit();          // CH455 应用初始化
    #endif

    #if (SHTXX_EN > 0)        // SHTXX 温湿度传感器配置使能
    SHTXX_APPInit();          // SHTXX 温湿度传感器初始化
    #endif
}
```

注意：如果用户有其它传感器需要加入参数这个程序在这里进行初始化。

(3) 在../driver/device_app.c中查看Device_AppProc处理函数：

```
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
}
```

这个函数实现具体外部器件的应用处理，处理完的数据通过

LibAppVars.Register.APP_InputDataCallback

回调函数发送出来。

(4) 外部器件数据被发送到../source/user_deviceapp.c中的Device_DataProcess函数中，具体函数如下：

```
void Device_DataProcess(LIBAPP_INPUTDATA_MSG *pDataMsg)
{
    switch(pDataMsg->id)
    {
        #if (SHTXX_EN > 0)
        case DEVICE_SHTXX_ID:                       // SHTXX温湿度的数据类型
            SHTXX_UserProcess(pDataMsg->pData);     // 处理SHTXX温湿度数据
            break;
        #endif
        #if (CH455_EN > 0)
        case DEVICE_CH455_ID:                       // CH455按键数据类型
            CH455_UserProcess(pDataMsg->pData);     // 处理CH455按键数据
            break;
        #endif
        default:
            break;
    }
}
```

用户请自行查看SHTXX_UserProcess和CH455_UserProcess函数，在这两个函数里做相应处理。

26. AT 指令编程说明

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《AMKN 系列工控板(模块)AT 指令_1.02_xxxx.pdf》。该指令通过调试串口(由 DEBUG_UART 定义，默认 UART1)进行控制。

(1) 在功能配置文件 XXXX_Config.h 中定义，参考如下：

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

(2) 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。

(3) 在../driver/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 请返馈给我们，由我司进行修改。

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 AMKNxxxx_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 AMKNxxxx_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 MOBUS 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 AMKNxxxx_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