GoKit3(V)二次开发-程序详解

机智云

编制人	Ture Zhang	审核人	Andy Gao	批准人	
产品名称		产品型号		文档编号	
会签日期			版本	V0.1.0	

GizWits

修改记录：

修改时间	修改记录	修改人	版本	备注
20160927	初建	TureZhang	0.1.0	
20160930	二版修改	TureZhang	0.1.1	

目录

1. 通信协议详解

1.1 协议命令格式

我们首先了解具体**通讯协议的约定**，可以看到协议格式为：

header(2B)=0xFFFF, len(2B), cmd(1B), sn(1B), flags(2B), payload(xB), checksum(1B)

说明：

1) 包头(header)固定为 0xFFFF；
2) 长度(len)是指从 cmd 开始到整个数据包结束所占用的字节数；
3) 命令字（cmd）表示具体的命令含义，详见协议举例；
4) 消息序号(sn)由发送方给出,接收方响应命令时需把消息序号返回给发送方；
5) 标志位（flag），本产品填写默认 0；
6) payload（p0 数据区），详细参见 p0 数据区约定；
7) 检验和(checksum)的计算方式为从 len～DATA，按字节求和；
8) 所有发送的命令都带有确认,如在 200 毫秒内没有收到接收方的响应,发送方应重发,最多重发 3 次；
9) 多于一个字节的整型数字以大端字节序编码（网络字节序）；
10) 数字均用 16 进制表示；

1.2“p0 数据区约定”解析

“p0 数据区约定”有如下功能：

1) 模块向 MCU 发送控制命令时携带 **p0 命令**和**命令标志位**以及**可写数据区**
2) MCU 主动发送状态时或者回复模块的状态查询时携带 **p0 命令**和**完整数据区**
3) 数据区会自动合并布尔和枚举变量，且有严格的顺序，不可任意改变

怎么来理解这三个功能呢？将前序中准备的**《XX-机智云接入串口通讯协议文档》**如打开，我们会看到如下命令：

1）WiFi 模组请求设备信息；
2）WiFi 模组与设备 MCU 的心跳；
3）设备 MCU 通知 WiFi 模组进入配置模式；
4）设备 MCU 重置 WiFi 模组；
5）WiFi 模组向设备 MCU 通知 WiFi 模组工作状态的变化；
6）WiFi 模组请求重启 MCU；
7）非法消息通知；
8）WiFi 模组读取设备的当前状态；
9）设备 MCU 向 WiFi 模组主动上报当前状态；
10）WiFi 模组控制设备；
（之后非重点省略）

“**p0 数据区约定**”主要作用是完成有效数据的上传(协议 4.8、4.9)与下达(协议 4.10)，其中**上传协议**的组成形式为：**action(1B) + dev_status(11B)；下达协议**的组成形式为：**action(1B) + attr_flags(1B) + attr_vals(6B)**；其中：

p0 数据区内容	含义
action	表示”p0 命令”的传输方向，即：WiFi -> MCU 或 MCU ->Wifi
dev_status	表示上报的所有数据点的设备状态
attr_flags	表示有效的控制型数据点
attr_vals	表示有效控制数据点的数据值

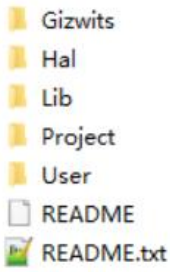
需要特别注意的是“p0 数据区约定”约定第三条，数据区会自动合并布尔和枚举变量，且有严格的顺序，不可任意改变。对应上面的“byte0”合并了“bool”和“enum”类型。

至此“**p0 数据区约定**”的解析到此结束，之后我们还会分析 MCU 的程序实现。

2. 程序详解

2.1 代码目录介绍

2.1.1 一级目录说明



文件夹	说明
Gizwits	协议相关目录
Hal	外设驱动库
Lib	STM32 驱动库
Project	工程管理文件
User	代码入口文件目录
README.txt	GoKit3(V)文档

2.1.2 代码文件说明

```
1  ├── Gizwits
2  │   ├── gizwits_product.c ---------->//产品相关处理函数，如：eventProcess()
3  │   ├── gizwits_product.h ---------->//gizwits_product.c 头文件，定义产品软硬件版本号
4  │   ├── gizwits_protocol.c --------->//gizwits协议相关的处理模块，API的封装模块
5  │   └── gizwits_protocol.h --------->//gizwits_protocol.c头文件，产品相关数据结构体
6  ├── Hal
7  │   └── Hal_key
8  │       └── Hal_key.h ------------->//Hal_key.c头文件
9  ├── Lib
10 ├── Project
11 │   ├── gokit_mcu_stm32.uvopt
12 │   └── gokit_mcu_stm32.uvproj ----->//工程管理入口文件
13 ├── README.txt
14 └── User
15     ├── delay.c ------------------->//延时函数
16     ├── delay.h
17     ├── main.c -------------------->//程序入口函数main()所在文件，包括各模块的初始化
18     ├── stm32f10x_conf.h ---------->//stm32平台相关资源管理
19     ├── stm32f10x_it.c ------------>//stm32平台相关中断函数
20     └── stm32f10x_it.h ------------>//stm32f10x_it.c头文件
```

主要文件说明：

文件	说明
gizwits_product.c	该文件为产品相关处理函数，如 ***gizEventProcess***() 平台相关硬件初始化，如串口、定时器等
gizwits_product.h	该文件为 gizwits_product.c 的头文件，存放产品相关宏定义如： HARDWARE_VERSION 、SOFTWARE_VERSION
gizwits_protocol.c	该文件为 SDK API 接口函数定义文件
gizwits_protocol.h	该文件为 gizwits_protocol.c 对应头文件，相关 API 的接口声明均在此文件中

2.1.3 协议 API 介绍

API 名称	API 功能
void gizwitsInit(void)	gizwits 协议初始化接口。 用户调用该接口可以完成 Gizwits 协议相关初始化（包括协议相关定时器、串口的初始化）。
void gizwitsSetMode(uint8_t mode)	参数 mode[in]：仅支持 0,1 和 2,其他数据无效。 参数为 0，恢复模组出厂配置接口，调用会清空所有配置参数，恢复到出厂默认配置。 参数为 1 时配置模组进入 SoftAp 模式； 参数为 2 配置模组进入 AirLink 模式。
void gizwitsHandle(dataPoint_t *dataPoint)	参数 dataPoint[in]:用户设备数据点。 该函数中完成了相应协议数据的处理即数据上报的等相关操作。
int8_t gizwitsEventProcess (eventInfo_t *info, uint8_t *data, uint32_t len)	参数 info[in]:事件队列 参数 data[in]:数据 参数 len [in]:数据长度 用户数据处理函数,包括 wifi 状态更新事件和控制事件。 a) Wifi 状态更新事件 WIFI_开头的事件为 wifi 状态更新事件，data 参数仅在 WIFI_RSSI 有效，data 值为 RSSI 值,数据类型为 uint8_t，取值范围 0~7。 b) 控制事件 与数据点相关,本版本代码会打印相关事件信息，相关数值也一并打印输出，用户只需要做命令的具体执行即可。

2.2 程序实现原理

协议实现机制：

协议解析后，将 P0 数据区的有效数据点生成对应的数据点事件，再按事件处理数据点。

数据点转换事件的说明：

根据协议 P0 数据区的 attr_flags 位判断出有效数据点，并将其转化成对应的数据点事件，然后在事件处理函数中(***gizwitsEventProcess***)完成事件的处理。

2.3 程序初始化说明

2.3.1 数据协议结构体的定义

结构体 ***dataPoint_t*** ，代码位置: **gokit_mcu_stm32_xxx\Gizwits\gizwits_protocol.h**

```
/** 用户区设备状态结构体*/
__packed typedef struct {
  bool valueLED_OnOff;                          ///< 对应数据点：LED_OnOff 读写类型：
  LED_COLOR_ENUM_T valueLED_Color;              ///< 对应数据点：LED_Color 读写类型：
  uint32_t valueLED_R;                          ///< 对应数据点：LED_R 读写类型：可写
  uint32_t valueLED_G;                          ///< 对应数据点：LED_G 读写类型：可写
  uint32_t valueLED_B;                          ///< 对应数据点：LED_B 读写类型：可写
  int32_t valueMotor_Speed;                     ///< 对应数据点：Motor_Speed 读写类型
  bool valueInfrared;                           ///< 对应数据点：Infrared 读写类型：
  int32_t valueTemperature;                     ///< 对应数据点：Temperature 读写类型
  uint32_t valueHumidity;                       ///< 对应数据点：Humidity 读写类型：
  bool valueAlert_1;                            ///< 对应数据点：Alert_1 读写类型：报
  bool valueAlert_2;                            ///< 对应数据点：Alert_1 读写类型：报
  bool valueFault_LED;                          ///< 对应数据点：Fault_LED 读写类型：
  bool valueFault_Motor;                        ///< 对应数据点：Fault_Motor 读写类型
  bool valueFault_TemHum;                       ///< 对应数据点：Fault_TemHum 读写类
  bool valueFault_IR;                           ///< 对应数据点：Fault_IR 读写类型：
} dataPoint_t;
```

说明：结构体 ***dataPoint_t*** 作用是存储用户区的设备状态信息，用户根据云端定义的数据点向其对应的数据位赋值后便不需关心数据的转换，其数据位对应“**p0 数据区约定**”中的“**4.9 设备 MCU 向 WiFi 模组主动上报当前状态**”中的：dev_status(11B) 位：

4.9 设备MCU向WiFi模组主动上报当前状态¶

设备MCU发送:

header(2B)	len(2B)	cmd(1B)	sn(1B)	flags(2B)	action(1B)	dev_status(11B)	checksum(1B)
0xFFFF	0x0011	0x05	0x##	0x0000	0x04	设备状态	0x##

attrFlags_t、attrVals_t ，代码位置: **gokit_mcu_stm32_xxx\Gizwits\gizwits_protocol.h**

```
/** 对应协议"4.10 WiFi模组控制设备"中的标志位"attr_flags(1B)" */
__packed typedef struct {
    uint8_t flagLED_OnOff:1;                    ///< 对应数据点：LED_OnOff 读写类
    uint8_t flagLED_Color:1;                    ///< 对应数据点：LED_Color 读写类
    uint8_t flagLED_R:1;                        ///< 对应数据点：LED_R 读写类型：
    uint8_t flagLED_G:1;                        ///< 对应数据点：LED_G 读写类型：
    uint8_t flagLED_B:1;                        ///< 对应数据点：LED_B 读写类型：
    uint8_t flagMotor_Speed:1;                  ///< 对应数据点：Motor_Speed 读写
} attrFlags_t;

/** 对应协议"4.10 WiFi模组控制设备"中的数据值"attr_vals(6B)" */
__packed typedef struct {
    uint8_t wBitBuf[COUNT_W_BIT];               ///< 可写型数据点 布尔和枚举变量
    uint8_t valueLED_R;                         ///< 对应数据点：LED_R 读写类型：
    uint8_t valueLED_G;                         ///< 对应数据点：LED_G 读写类型：
    uint8_t valueLED_B;                         ///< 对应数据点：LED_B 读写类型：
    uint16_t valueMotor_Speed;                  ///< 对应数据点：Motor_Speed 读写
} attrVals_t;
```

结构体 ***attrFlags_t***、***attrVals_t*** 分别对应“**p0 数据区约定**”中的“**4.10 WiFi 模组控制设备**”中的：attr_flags(1B) + attr_vals(6B)位：

4.10 WiFi模组控制设备¶

WiFi模组发送:

header(2B)	len(2B)	cmd(1B)	sn(1B)	flags(2B)	action(1B)	attr_flags(1B)	attr_vals(6B)	checksum(1B)
0xFFFF	0x000D	0x03	0x##	0x0000	0x01	是否设置标志位	设置数据值	0x##

结构体 **devStatus_t**，代码位置: **gokit_mcu_stm32_xxx\Gizwits\gizwits_protocol.h**

```
/** 对应协议"4.9 设备MCU向WiFi模组主动上报当前状态"中的设备状态"dev_status(11B)"
__packed typedef struct {
    uint8_t wBitBuf[COUNT_W_BIT];               ///< 可写型数据点 布尔和枚举变

    uint8_t valueLED_R;                         ///< 对应数据点：LED_R 读写类
    uint8_t valueLED_G;                         ///< 对应数据点：LED_G 读写类
    uint8_t valueLED_B;                         ///< 对应数据点：LED_B 读写类
    uint16_t valueMotor_Speed;                  ///< 对应数据点：Motor_Speed

    uint8_t rBitBuf[COUNT_R_BIT];               ///< 只读型数据点 布尔和枚举变

    uint8_t valueTemperature;                   ///< 对应数据点：Temperature
    uint8_t valueHumidity;                      ///< 对应数据点：Humidity 读写

    uint8_t valueAlert_1:1;                     ///< 对应数据点：Alert_1 读写
    uint8_t valueAlert_2:1;                     ///< 对应数据点：Alert_1 读写

    uint8_t valuereserve_2:6;                   ///< 数据位补齐

    uint8_t valueFault_LED:1;                   ///< 对应数据点：Fault_LED 读
    uint8_t valueFault_Motor:1;                 ///< 对应数据点：Fault_Motor
    uint8_t valueFault_TemHum:1;                ///< 对应数据点：Fault_TemHum
    uint8_t valueFault_IR:1;                    ///< 对应数据点：Fault_IR 读写

    uint8_t valuereserve_3:4;                   ///< 数据位补齐
} devStatus_t;
```

结构体 ***devStatus_t*** 对应“**p0 数据区约定**”中的“**4.9 设备 MCU 向 WiFi 模组主动上**

报当前状态”中的：dev_status(11B) 位：

4.9 设备MCU向WiFi模组主动上报当前状态¶

设备MCU发送：

header(2B)	len(2B)	cmd(1B)	sn(1B)	flags(2B)	action(1B)	dev_status(11B)	checksum(1B)
0xFFFF	0x0011	0x05	0x##	0x0000	0x04	设备状态	0x##

特别说明：

A. 数据结构说明

dataPoint_t 为应用层数据结构，开发者需要了解并会使用（具体使用方式请查看：“2.7.1 只读型数据的获取”一节）。

attrFlags_t、***attrVals_t***、***devStatus_t*** 为通信层数据结构，开发者需要结合通讯协议进行理解。

B. 位段举例说明：

uint8_t motor_switch:1; 是一种位段的使用方式。因为 uint8_t 型数据占用 8bit（8 位）的空间，协议中 **motor_switch 占用**字段 bit0（第一位）所以 ***uint8_t motor_switch:1*** 表示使用 1 位的空间。

uint8_t reserve:7; 因为程序中申请内存时的最小单位是 byte(字节)，而这里我们是按 bit(位，8bit = 1byte)进行了使用，故需补齐不足 1byte 的剩余 bit(使用 n bit 后需补齐剩余的 8-n bit)。

注：位段不能跨字节操作，否则会造成数据读写错误。

2.3.2 程序主函数

位置：**gokit_mcu_stm32_xxx\User\main.c** 中 main() 函数：

```
178  int main(void)
179  {
180      SystemInit();
181
182      userInit();
183      keyInit();
184
185      gizwitsInit();
186
187      printf("MCU Init Success \n");
188      while(1)
189      {
190          watchdogFeed();
191
192          userHandle();
193
194          gizwitsHandle((dataPoint_t *)&currentDataPoint);
195      }
196  }
```

相关说明：

函数	说明
SystemInit()	平台相关的硬件初始化 （非 API，不同的平台名称可能不同）
userInit()	用户相关的初始化，如：外设驱动初始化、打印串口初始化 （非 API，不同的平台名称可能不同）
gizwitsInit()	平台、协议处理初始化，如：用户定时器初始化、协议通信串口初始化 （协议 API）
userHandle()	用户事件回调函数，用户可以自定义事件在该函数中完成相应的协议处理。 （非 API，不同的平台名称可能不同）
gizwitsHandle()	协议相关的主函数 （协议 API）

2.3.3 用户程序初始化

接下来看用户初始化相关代码（位置：**main.c** 中 ***userInit() 函数***）：

```
/**
* 用户初始化函数

* 在该函数中完成了外设驱动初始化以及用户相关数据的初始
* @param none
* @return none
* @note 开发者可在此函数内添加新的驱动初始及状态初始化
*/
void userInit(void)
{
    delayInit(72);
    uartxInit();
    rgbLedInit();
    rgbKeyGpioInit();
    motorInit();
    dht11Init();
    irInit();
    watchdogInit(2);    //5,625看门狗复位时间2s
    memset((uint8_t*)&currentDataPoint, 0, sizeof(dataPoint_t));
    motorStatus(MOTOR_SPEED_DEFAULT);
}
```

这部分完成了 RGB LED、电机、温湿度、红外传感器的硬件驱动初始化以及电机初始状态，对应的驱动程序实现都在 **gokit_mcu_stm32_xxx\Hal** 下。

这里主要完成了配置入网的功能，作为开发者可以按照自己的需求来实现这部分代码。

下面是平台协议相关初始化 （位置：**main.c** 中 ***gizwitsInit() 函数***）：

```
/**
* @brief gizwits协议初始化接口

* 用户调用该接口可以完成Gizwits协议相关初始化（包括协议相关定时器、串口的初始）

* 用户可在在此接口内完成数据点的初始化状态设置

* @param none
* @return none
*/
void gizwitsInit(void)
{
    timerInit();
    uartInit();

    pRb.rbCapacity = RB_MAX_LEN;
    pRb.rbBuff = rbBuf;
    rbCreate(&pRb);

    memset((uint8_t *)&gizwitsProtocol, 0, sizeof(gizwitsProtocol_t));
}
```

其中完成了定时器、串口的初始化（详情查看 2.3.4、2.3.5 两节），以及一个环形缓冲区的初始化。

最后是一个通信处理模块结构体的变量的初始化，该变量为通信模块的全局变量：

```
1460        memset((uint8_t *)&gizwitsProtocol, 0, sizeof(gizwitsProtocol_t));
```

其定义的位置：**user\user_mian.c**

```
25  /** 协议全局变量 **/
26  gizwitsProtocol_t gizwitsProtocol;
```

相关结构体内容，详情查看“2.3.1 数据协议结构体的定义”一节。

2.3.4 定时器使用

相关代码：

定时器初始化，代码位置：**gokit_mcu_stm32_xxx\Gizwits\gizwits_product.c 中 timerInit()函数**

```
void timerInit(void)
{
    u16 arr = 7199;
    u16 psc = 9;
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_APB1PeriphClockCmd(TIMER_RCC, ENABLE); //时钟使能

    //定时器TIM3初始化
    TIM_TimeBaseStructure.TIM_Period = arr; //设置在下一个更新事件装入活动的自动重载寄存器周期的值
    TIM_TimeBaseStructure.TIM_Prescaler =psc; //设置用来作为TIMx时钟频率的预分频值
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1; //设置时钟分割:TDTS = Tck_tim
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;  //TIM向上计数模式
    TIM_TimeBaseInit(TIMER, &TIM_TimeBaseStructure); //根据指定的参数初始化TIMx的时间基数单位0

    TIM_ITConfig(TIMER, TIM_IT_Update,ENABLE ); //使能指定的TIM3中断,允许更新中断

    //中断优先级NVIC设置
    NVIC_InitStructure.NVIC_IRQChannel = TIMER_IRQ;  //TIM3中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;  //优占优先级0级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;  //从优先级3级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ通道被使能
    NVIC_Init(&NVIC_InitStructure);  //初始化NVIC寄存器
    TIM_Cmd(TIMER, ENABLE);  //使能TIMx
}
```

注：这里我们定义了周期为 1ms 的定时器，其定时计算公式为：

TIM3溢出时间计算（单位为us）：

Tout= ((arr+1)*(psc+1))/Tclk；

其中：

Tclk：TIM3的输入时钟频率（单位为Mhz）

arr：自动重装载寄存器（TIMx_ARR）；

psc：预分频器(TIMx_PSC);

结果：

((7199+1)*(9+1))/72 = 1000 us = 1 ms

定时器中断函数，代码位置：**gokit_mcu_stm32_xxx\Gizwits\gizwits_product.c**

```
/**
* @brief 定时器TIM3中断处理函数

* @param none
* @return none
*/
void TIMER_IRQ_FUN(void)
{
    if (TIM_GetITStatus(TIMER, TIM_IT_Update) != RESET)
    {
        TIM_ClearITPendingBit(TIMER, TIM_IT_Update  );
        gizTimerMs();
    }
}
```

注：在该中断函数内我们完成了周期为 1ms 的定时累加计数。

定时器使用说明：

代码位置：**gokit_mcu_stm32_xxx\Gizwits\gizwits_product.h**

```
/**@name TIM3相关宏定义
* @{
*/
#define TIMER TIM3
#define TIMER_IRQ TIM3_IRQn
#define TIMER_RCC RCC_APB1Periph_TIM3
#define TIMER_IRQ_FUN TIM3_IRQHandler
/**@} */
```

a. 这里我们使用定时器 TIM3(**#define TIMER TIM3**)；
b. TIM3 的中断回调函数为 **UTIM3_IRQHandler() (#define TIMER_IRQ_FUN TIM3_IRQHandler)**；

特别说明（复用 **TIMER2** 的方式，修改对应宏即可）：

```
/**@name TIM2相关宏定义
* @{
*/
#define TIMER TIM2
#define TIMER_IRQ TIM2_IRQn
#define TIMER_RCC RCC_APB1Periph_TIM2
#define TIMER_IRQ_FUN TIM2_IRQHandler
/**@} */
```

```
/**@name TIM3相关宏定义
* @{
*/
#define TIMER TIM2
#define TIMER_IRQ TIM2_IRQn
#define TIMER_RCC RCC_APB1Periph_TIM2
#define TIMER_IRQ_FUN TIM2_IRQHandler
/**@} */
```

2.3.5 串口的使用

相关代码：

串口初始化，位置：**gokit_mcu_stm32_xxx\Gizwits\gizwits_product.c 中的 uartInit()**

```
    /*使能串口中断,并设置优先级*/
    NVIC_InitStructure.NVIC_IRQChannel = UART_IRQ;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
```

串口中断函数，位置：**gokit_mcu_stm32_xxx\Gizwits\gizwits_product.c**

```
/**
* @brief USART2串口中断函数

* 接收功能，用于接收与WiFi模组间的串口协议数据
* @param none
* @return none
*/
void UART_IRQ_FUN(void)
{
    uint8_t value = 0;
    if(USART_GetITStatus(USART2, USART_IT_RXNE) != RESET)
    {
        USART_ClearITPendingBit(USART2,USART_IT_RXNE);
        value = USART_ReceiveData(USART2);

        gizPutData(&value, 1);
    }
}
```

串口使用说明：

代码位置：**gokit_mcu_stm32_xxx\Gizwits\gizwits_product.h**

```
/**@name USART相关宏定义
* @{
*/
#define UART_BAUDRATE          9600
#define UART_PORT              2
#define UART                   USART2
#define UART_IRQ               USART2_IRQn
#define UART_IRQ_FUN           USART2_IRQHandler
```

a. 这里我们使用 USART2(**#define UART USART2**)，作为数据通信的串口;
b. 设置它的波特率为 9600(**#define UART_BAUDRATE 9600**)
c. USART2 的串口中断回调函数为 **USART2_IRQHandler() (#define UART_IRQ_FUN USART2_IRQHandler)**，在该中断函数内我们完成了串口数据的接收。

特别说明（复用 **USART1** 的方式，修改对应宏即可）：

```
/**@name USART相关宏定义
* @{
*/
#define UART_BAUDRATE          9600
#define UART_PORT              1
#define UART                   USART1
#define UART_IRQ               USART1_IRQn
#define UART_IRQ_FUN           USART1_IRQHandler
```

2.4 配置模式说明

设备需要进入配置模式才能进行联网，并与云端进行通信，在本示例工程中是通过按键触发进入相应的配置模式。

Wifi 配置接口说明：

```
/**
* @brief WiFi 配置接口

* 用户可以调用该接口使 WiFi 模组进入相应的配置模式或者复位模组

* @param[in] mode 配置模式选择：0x0， 模组复位 ;0x01， SoftAp 模式 ;0x02， AirLink 模式
* @return 错误命令码
*/
```

- **int32_t gizwitsSetMode(uint8_t mode)**

程序中触发逻辑位置：**gokit_mcu_stm32_xxx\User\main.c**

A. 进入 Soft AP 模式：key2 按键短按。

```
114  void key2ShortPress(void)
115  {
116      printf("KEY2 PRESS ,Soft AP mode\n");
117
118      //Soft AP mode, RGB red
119      ledRgbControl(250, 0, 0);
120      gizwitsSetMode(WIFI_SOFTAP_MODE);
121  }
```

B. 进入 AirLink 模式：key2 按键长按。

```
128  void key2LongPress(void)
129  {
130      printf("KEY2 PRESS LONG ,AirLink mode\n");
131
132      //AirLink mode, RGB Green
133      ledRgbControl(0, 250, 0);
134      gizwitsSetMode(WIFI_AIRLINK_MODE);
135  }
```

C. 模组复位：key1 按键长。

```
103  void key1LongPress(void)
104  {
105      printf("KEY1 PRESS LONG ,Wifi Reset\n");
106      gizwitsSetMode(WIFI_RESET_MODE);
107  }
```

2.5 协议处理函数的实现

位置：**gokit_mcu_stm32_xxx\Gizwits\gizwits_protocol.c** 中 **gizwitsHandle()** 函数：

```
1511 /**
1512 * @brief 协议处理函数
1513
1514 * 该函数中完成了相应协议数据的处理及数据主动上报的相关操作
1515
1516 * @param [in] currentData 待上报的协议数据指针
1517 * @return none
1518 */
1519 int32_t gizwitsHandle(dataPoint_t *currentData)
```

以下是该协议处理函数的详细介绍：

- 首先是一些局部变量的初始化，比较重要的是：“**protocolHead_t *recvHead = NULL;**”它的作用是保存解析出来的协议包头。

```
1518 int32_t gizwitsHandle(dataPoint_t *currentData)
1519 {
1520     int8_t ret = 0;
1521     uint16_t i = 0;
1522     uint8_t ackData[RB_MAX_LEN];
1523     uint16_t protocolLen = 0;
1524     uint32_t ackLen = 0;
1525     protocolHead_t *recvHead = NULL;
```

- 然后是协议的重发机制，它的作用是对发送后的协议数据进行超时判断，超时 200ms 进行重发，重发上限为三次：

```
1533     /*重发机制*/
1534     gizProtocolAckHandle();
```

- 接下来程序会从环形缓冲区中抓取一包的数据，例如协议 4.9：

4.9 设备MCU向WiFi模组主动上报当前状态¶

设备MCU发送：

header(2B)	len(2B)	cmd(1B)	sn(1B)	flags(2B)	action(1B)	dev_status(11B)	checksum(1B)
0xFFFF	0x0011	0x05	0x##	0x0000	0x04	设备状态	0x##

程序中对应如下：

```
1535        ret = gizProtocolGetOnePacket(&pRb, gizwitsProtocol.protocolBuf, &protocolLen);
```

- 当我们获得到一整包的数据，就会进入下面的 if 判断逻辑，进行协议的解析。

```
1537        if(0 == ret)
1538        {
1539            GIZWITS_LOG("Get One Packet!\n");
1540
```

这里保存了接收到的协议包头：

```
1550            recvHead = (protocolHead_t *)gizwitsProtocol.protocolBuf;
```

- 然后是各协议命令的处理流程：

```
1551        switch (recvHead->cmd)
1552        {
1553            case CMD_GET_DEVICE_INTO:
1554                gizProtocolGetDeviceInfo(recvHead);
1555                break;
1556            case CMD_ISSUED_P0:
1557                ret = gizProtocolIssuedProcess(gizwitsProtocol.protocolB
1558                if(0 == ret)
1559                {
1560                    gizProtocolIssuedDataAck(recvHead, ackData, ackLen);
1561                }
1562                break;
1563            case CMD_HEARTBEAT:
1564                gizProtocolCommonAck(recvHead);
1565                break;
```

其中完成了《机智云 - 设备串口通讯协议》中相关的协议处理，如下：

例如协议 4.8：

4.8 WiFi模组读取设备的当前状态

WiFi模组发送：

header(2B)	len(2B)	cmd(1B)	sn(1B)	flags(2B)	action(1B)	checksum(1B)
0xFFFF	0x0006	0x03	0x##	0x0000	0x02	0x##

其“cmd”值 为“0x03”，对应程序中的的 case 为“**CMD_ISSUED_P0**”

```
1556                case CMD_ISSUED_P0:
1557                    ret = gizProtocolIssuedProcess(gizwitsProtocol.protocolBuf
1558                    if(0 == ret)
1559                    {
1560                        gizProtocolIssuedDataAck(recvHead, ackData, ackLen);
1561                    }
1562                    break;
```

同理其他协议 cmd 值对应的宏定义的位置在 **Gizwits\gizwits_protocol.h** 中:

```
263 /** 协议命令码 */
264 __packed typedef enum
265 {
266     CMD_GET_DEVICE_INTO             = 0x01,         ///< 命令字，对应协议：4.1 WiFi模组
267     ACK_GET_DEVICE_INFO             = 0x02,         ///< 命令字，对应协议：4.1 WiFi模组
268
269     CMD_ISSUED_P0                   = 0x03,         ///< 命令字，对应协议：4.8 WiFi模组
270     ACK_ISSUED_P0                   = 0x04,         ///< 命令字，对应协议：4.8 WiFi模组
271
272     CMD_REPORT_P0                   = 0x05,         ///< 命令字，对应协议：4.9 设备MCU向
273     ACK_REPORT_P0                   = 0x06,         ///< 命令字，对应协议：4.9 设备MCU向
274
275     CMD_HEARTBEAT                   = 0x07,         ///< 命令字，对应协议：4.2 WiFi模组
276     ACK_HEARTBEAT                   = 0x08,         ///< 命令字，对应协议：4.2 WiFi模组
277
```

其中与 P0 协议有关的处理都在“***gizProtocolIssuedProcess***”中完成，详情请查看“2.6 控制型协议的实现”、“2.7 上报型协议的实现”两节。

其余协议处理函数功能如下所示：

函数	说明
gizProtocolGetDeviceInfo	完成“协议 4.1 WiFi 模组请求设备信息”
gizProtocolIssuedProcess	完成“协议 4.8 WiFi 模组读取设备的当前状态”与“协议 4.10 WiFi 模组控制设备”。 当 Wifi 模组接收来自云端或 APP 端下发的相关协议数据发送到 MCU 端，经过协议报文解析后将相关协议数据传入次函数，进行下一步的协议处理。
gizProtocolCommonAck	发送通用协议报文数据
gizProtocolModuleStatus	完成“协议 4.5 WiFi 模组向设备 MCU 通知 WiFi 模组工作状态的变化”的处理
gizProtocolWaitAckCheck	完成 “协议 4.4 设备 MCU 重置 WiFi 模组 中 WiFi 模组回复”后清除 ACK 协议报文
gizProtocolReboot	完成“协议 4.4 设备 MCU 重置 WiFi 模组” 的相关操作
gizProtocolErrorCmd	完成“协议 4.7 非法消息通知” 的处理
gizwitsEventProcess()	执行用户事件回调函数，用户可以自定义事件在该函数中完成相应的协议处理。

- 协议判断完成后是一个状态机的判断，用来完成对应协议命令的处理：

```
1598    if(1 == gizwitsProtocol.issuedFlag)
1599    {
1600        gizwitsProtocol.issuedFlag = 0;
1601        gizwitsEventProcess(&gizwitsProtocol.issuedProcessEvent, (uint
1602        memset((uint8_t *)&gizwitsProtocol.issuedProcessEvent,0x0,size
1603    }
1604    else if(2 == gizwitsProtocol.issuedFlag)
1605    {
1606        gizwitsProtocol.issuedFlag = 0;
1607        gizwitsEventProcess(&gizwitsProtocol.wifiStatusEvent, (uint8_t
1608        memset((uint8_t *)&gizwitsProtocol.wifiStatusEvent,0x0,sizeof(
1609    }
1610    else if(3 == gizwitsProtocol.issuedFlag)
1611    {
1612        gizwitsProtocol.issuedFlag = 0;
1613        gizwitsEventProcess(&gizwitsProtocol.issuedProcessEvent, (uint
1614    }
```

例如在 P0 协议处理函数（***gizProtocolIssuedProcess***）中，当我们完成了控制型协议的解析，会让 issuedFlag = 1，如下：

```
1404    switch(issuedAction)
1405    {
1406        case ACTION_CONTROL_DEVICE:
1407            gizDataPoint2Event((gizwitsIssued_t *)(inData+siz
1408            gizwitsProtocol.issuedFlag = 1;
1409            outData = NULL;
1410            *outLen = 0;
1411            break;
```

然后会执行如下的处理，执行 **gizwitsEventProcess 函数**：

```
1598    if(1 == gizwitsProtocol.issuedFlag)
1599    {
1600        gizwitsProtocol.issuedFlag = 0;
1601        gizwitsEventProcess(&gizwitsProtocol.issuedProcessEvent, (uint8_t *)&gizwit
1602        memset((uint8_t *)&gizwitsProtocol.issuedProcessEvent,0x0,sizeof(gizwitsPrc
1603    }
```

在 ***gizwitsEventProcess*** 中，完成了对应控制型事件的处理，其他状态的 issuedFlag 同理。

- 之后是一个数据上报判断机制，主要执行了 ***gizCheckReport*** 函数。

```
1616    if((1 == gizCheckReport(currentData, (dataPoint_t *)&gizwitsProtocol.gizL
1617    {
1618        GIZWITS_LOG("changed, report data\n");
1619        gizDataPoints2ReportData(currentData,&gizwitsProtocol.reportData.devS
1620        gizReportData(ACTION_REPORT_DEV_STATUS, (uint8_t *)&gizwitsProtocol.r
1621        memcpy((uint8_t *)&gizwitsProtocol.gizLastDataPoint, (uint8_t *)curre
1622    }
```

gizCheckReport 函数的作用用来判断当前与上次上报数据的一致性，如果符合上报条件便上报，上报条件要符合协议“4.9 设备 MCU 向 WiFi 模组主动上报当前状态”中的描述：

2. 关于发送频率。当设备MCU收到WiFi模组控制产生的状态变化，设备MCU应立刻主动上报当前状态，发送频率不受限制。但如设备的状态的变化是由于用户触发或环境变化所产生的，其发送的频率不能快于6秒每次。建议按需上报，有特殊上报需求请联系机智云。

符合上报之后会执行数据类型的转化函数 ***gizDataPoints2ReportData***（详情查看“**2.8 机智云协议数据处理**”一节），以及数据上报函数 ***gizReportData***。

- 最后一段代码是一个数据定时上报机制：

```
1624    if(1000*60*10 <= (gizGetTimerCount() - gizwitsProtocol.lastReportTime))
1625    {
1626        GIZWITS_LOG("Info: 600S report data\n");
1627        gizDataPoints2ReportData(currentData,&gizwitsProtocol.reportData.devStatus);
1628        gizReportData(ACTION_REPORT_DEV_STATUS, (uint8_t *)&gizwitsProtocol.reportData.devStatu
1629        memcpy((uint8_t *)&gizwitsProtocol.gizLastDataPoint, (uint8_t *)currentData, sizeof(dat
1630    }
1631    return 0;
```

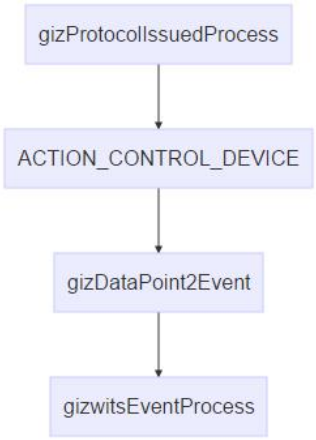
对应协议“4.9 设备 MCU 向 WiFi 模组主动上报当前状态”中的描述：

3. 设备MCU需要每隔10分钟定期主动上报当前状态。

至此我们完成了协议处理函数的详解。

2.6 控制型协议的实现

与控制型协议相关的函数调用关系如下：



函数调用说明：

函数	说明
gizProtocolIssuedProcess	该函数被 gizwitsHandle 调用，接收来自云端或 app 端下发的相关协议数据
ACTION_CONTROL_DEVICE	进行“控制型协议”的相关处理
gizDataPoint2Event	根据协议生成“控制型事件”，并完成相应数据类型的转换
gizwitsEventProcess	根据已生成的“控制型事件”进行相应事件处理（即调用相应的驱动函数）

2.6.1 控制型事件处理

相关代码位置:

gokit_mcu_stm32_xxx\Gizwits\gizwits_product.c 中 ***gizwitsEventProcess()*** 函数：

功能说明：

完成写类型外设的事件处理。

相应代码：

```
61      switch(info->event[i])
62      {
63        case EVENT_LED_ONOFF:
64          currentDataPoint.valueLED_OnOff = dataPointPtr->valueLED_OnOff;
65          GIZWITS_LOG("Evt: EVENT_LED_ONOFF %d \n", currentDataPoint.valueLED_OnOff);
66          if(0x01 == currentDataPoint.valueLED_OnOff)
67          {
68              ledRgbControl(254,0,0);
69          }
70          else
71          {
72              ledRgbControl(0,0,0);
73          }
74          break;
```

2.6.2 可写型数据类型转换

接收到来自云端的数据后，由于原始数据经过特殊处理，所以要在**gizDataPoint2Event 中**进行相应的数据的转换。

转换函数说明：

gizDecompressionValue	完成传输数据的压缩处理，详情查看“2.8.2 数据解压与压缩处理”一节。
gizX2Y	将用户区数据转化为传输数据，详情查看“2.8.1 数据点类型转换”一节。

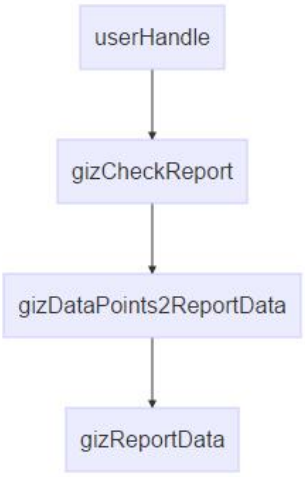
程序中对应：

```
678     if(0x01 == issuedData->attrFlags.flagLED_OnOff)
679     {
680         info->event[info->num] = EVENT_LED_ONOFF;
681         info->num++;
682         dataPoints->valueLED_OnOff = gizDecompressionValue(LED_ONOFF_BYT
683     }
```

```
699     if(0x01 == issuedData->attrFlags.flagLED_G)
700     {
701         info->event[info->num] = EVENT_LED_G;
702         info->num++;
703         dataPoints->valueLED_G = gizX2Y(LED_G_RATIO,  LED_G_ADDITION,
704     }
```

2.7 上报型协议的实现

与上报型协议相关的函数调用关系如下：



函数调用说明：

函数	说明
userHandle	获取用户区的上报型数据
gizCheckReport	判断是否上报当前状态的数据
gizDataPoints2ReportData	完成用户区数据到上报型数据的转换
gizReportData	将转换后的上报数据通过串口发送给 WiFi 模块

2.7.1 只读型数据的获取

相关代码位置:

gokit_mcu_stm32_xxx\User\main.c 中 ***userHandle()*** 函数：

使用说明：

该函数中完成了用户区上报型数据的获取。用户只需将读到的数据赋值到**用户区当前设备状态结构体**即可：

```
74      currentDataPoint.valueInfrared = irHandle();
```

注：赋值完的数据是通过 **gizwitsHandle** （详情请查看“2.5 协议处理函数的实现”一节中：数据上报判断机制 ***gizCheckReport*** 部分）上报云端的，开发者不需要关注变化上报和定时上报。

2.7.2 上报状态判断

为了让 API 接口更简化，处理更简单，机智云把更多的判断放到协议模块来处理，达到了开发者只要把状态更新到协议处理模块，不需要关心何时上报，由协议处理模块自动完处理的目的。

相关代码:

gokit_mcu_stm32_xxx\Gizwits\gizwits_protocol.c 中 ***checkReport()*** 函数：

功能说明：

根据协议判断是否上报当前状态的数据，判断逻辑如下：

1. 控制型数据发生状态变化，立刻主动上报当前状态
2. 用户触发或环境变化所产生的，其发送的频率不能快于 6 秒每次

协议中说明如下：(“4.9 设备 MCU 向 WiFi 模组主动上报当前状态”)

2. 关于发送频率。当设备MCU收到WiFi模组控制产生的状态变化，设备MCU应立刻主动上报当前状态，发送频率不受限制。但如设备的状态的变化是由于用户触发或环境变化所产生的，其发送的频率不能快于6秒每次。建议按需上报，有特殊上报需求请联系机智云。

3. 设备MCU需要每隔10分钟定期主动上报当前状态。

以红灯开关为例：

```
/**
 * @brief 对比当前数据和上次数据
 *
 * @param [in] cur          : 当前数据点数据
 * @param [in] last         : 上次数据点数据
 *
 * @return : 0,数据无变化;1, 数据有变化
 */
static int8_t gizCheckReport(dataPoint_t *cur, dataPoint_t *last)
{
    int8_t ret = 0;
    static uint32_t lastReportTime = 0;

    if((NULL == cur) || (NULL == last))
    {
        GIZWITS_LOG("gizCheckReport Error , Illegal Param\n");
        return -1;
    }
    if(last->valueLED_OnOff != cur->valueLED_OnOff)
    {
        GIZWITS_LOG("valueLED_OnOff Changed\n");
        ret = 1;
    }
```

注：这部分用户可结合“2.5 协议处理函数的实现”一节中“数据上报判断机制”的内容来理解）。

2.7.3 只读型数据类型转换

获得到用户区的原始数据后，在传输到云端前要进行相应的数据转换，所以要在 **gizDataPoints2ReportData** 中进行相应的数据的转换。

转换函数说明：

gizCompressValue	完成传输数据的压缩处理，详情查看“2.8.2 数据解压与压缩处理”一节。
gizY2X	将用户区数据转化为传输数据，详情查看“2.8.1 数据点类型转换”一节。

2.8 机智云协议数据处理

2.8.1 数据点类型转换

机智云为使设备功能定义更加简单直接，使用户输入的数值转换成设备能够识别的 uint 类型，这套算法的核心公式是：y=kx+m （y：显示值；x：传输值；k：分辨率；m：增量）

以《微信宠物屋》的温湿度传感器温度数据点为例：

取值范围：-13（Ymin） ~ 187（Ymax），分辨率：1，增量：-13 ；

其分辨率、偏移量作为宏定义定义在 **app\Gizwits\gizwits_product.h** 中：

```
51  #define TEMPERATURE_RATIO                    1
52  #define TEMPERATURE_ADDITION                 (-13)
53  #define TEMPERATURE_DEFAULT                  0
```

根据公式：y=kx＋m，k = 1 ; m = -13

实际传输的值：x = (y - m) / k

转换函数在程序中的说明：

A.X2Y 的转换：

```
359  * @brief 转化为协议中的y值及App UI界面的显示值
360  *
361  * @param [in] ratio     : 修正系数k
362  * @param [in] addition  : 增量m
363  * @param [in] preValue  : 作为协议中的x值，是实际通讯传输的值
364  *
365  * @return aftValue : 作为协议中的y值，是App UI界面的显示值
366  */
367 static int32_t gizX2Y(uint32_t ratio, int32_t addition, uint32_t preValue)
```

B. Y2X 的转换：

```
339 /**
340  * @brief 转化为协议中的x值及实际通讯传输的值
341  *
342  * @param [in] ratio     : 修正系数k
343  * @param [in] addition  : 增量m
344  * @param [in] preValue  : 作为协议中的y值，是App UI界面的显示值
345  *
346  * @return aft_value : 作为协议中的x值，是实际通讯传输的值
347  */
348 static uint32_t gizY2X(uint32_t ratio, int32_t addition, int32_t preValue)
```

2.8.2 数据解压与压缩处理

设备端与自云端的数据交互过程中，一些特殊类型（bool 和 enum 类型）的数据点原始数据只有被特殊处理后才可被云端解析，所以设备端在接收云端数据时要进行数据的**解压处理**；在向云端发送数据时进行数据的**压缩处理**。

机智云已封装出了相应的处理接口：

处理名称	接口名称
bool 和 enum 类型数据点数据解压	***gizDecompressionValue***
bool 和 enum 类型数据点数据压缩	***gizCompressValue***

以《微信宠物屋》的 RGB LED 控制为例，云端定义如下：

对应文档中数据存储格式如下：

字节序	bit序	数据内容	说明
byte0	bit7 bit6 . . . bit1 bit0	0b00000111	LED_OnOff，类型为bool，值为true：字段bit0，字段值为0b1； LED_Color，类型为enum，值为3：字段bit2 ~ bit1，字段值为0b11；

字节序与 bit 序对应代码中宏定义如下：

```
#define LED_ONOFF_BYTEOFFSET            0           ///< 数据点LED_OnOff 节序
#define LED_ONOFF_BITOFFSET             0           ///< 数据点LED_OnOff bit序
#define LED_ONOFF_LEN                   1           ///< 数据点LED_OnOff 字段值

#define LED_COLOR_BYTEOFFSET            0           ///< 数据点LED_Color 节序
#define LED_COLOR_BITOFFSET             1           ///< 数据点LED_Color bit序
#define LED_COLOR_LEN                   2           ///< 数据点LED_Color 字段值
```

对应的数据点在接收解压时处理如下(位于 ***gizDataPoint2Event*** 函数中)：位于

```
    if(0x01 == issuedData->attrFlags.flagLED_OnOff)
    {
        info->event[info->num] = EVENT_LED_ONOFF;
        info->num++;
        dataPoints->valueLED_OnOff = gizDecompressionValue(LED_ONOFF_BYTEOFFSET,LED_ONOFF_BITOFFSET,LED_ONOFF_LEN,(u
    }

    if(0x01 == issuedData->attrFlags.flagLED_Color)
    {
        info->event[info->num] = EVENT_LED_COLOR;
        info->num++;
        dataPoints->valueLED_Color = gizDecompressionValue(LED_COLOR_BYTEOFFSET,LED_COLOR_BITOFFSET,LED_COLOR_LEN,(u
    }
```

对应的数据点在发送压缩时处理如下(位于 ***gizDataPoints2ReportData*** 函数中)：

```
    gizCompressValue(LED_ONOFF_BYTEOFFSET,LED_ONOFF_BITOFFSET,LED_ONOFF_LEN,(uint8_t *)devStatusPtr,dataPoints->valueLED_OnOff);
    gizCompressValue(LED_COLOR_BYTEOFFSET,LED_COLOR_BITOFFSET,LED_COLOR_LEN,(uint8_t *)devStatusPtr,dataPoints->valueLED_Color);
```

3. 相关支持

1) 如果您是开发者

GoKit 是面向智能硬件开发者限量免费开放，注册我们的论坛或关注我们的官方微信均可发起申请即可。

开发者论坛： http://club.gizwits.com/forum.php

文档中心：http://docs.gizwits.com/hc/

2) 如果您是团体

GizWits 针对团体有很多支持计划，您可以和 GizWtis 联系，快速得到 GoKit 以及技术支持；

网站地址：http://www.gizwits.com/about-us

官方二维码：