GoKit3(S)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
20161008	初建	TureZhang	0.1.0	

目录

1. 通信协议详解

1.1 协议阅读前需知

A. SOC 版与 MCU 版的区别：

由于 SoC 方案是直接在 WiFi 模组上进行开发故没有 MCU 这一概念，无需进行串口协议传输，没有协议组包和协议解析这些步骤，所以没有串口协议，重点是“P0 数据区”解析这部分。

B. SOC 版与 MCU 版的联系：

云端生成的协议文档默认是 MCU 版的协议文档，其实 SOC 版完全可复用 MCU 版的协议，故在这里直接将《xxx-机智云接入串口通讯协议文档》中的 MCU 理解为 SOC（后文同理）。

1.2“p0 数据区约定”

“p0 数据区约定”有如下功能：

1) 模块向 SOC 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) SOC 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 SOC 的心跳；
3）设备 SOC 通知 WiFi 模组进入配置模式；
4）设备 SOC 重置 WiFi 模组；
5）WiFi 模组向设备 SOC 通知 WiFi 模组工作状态的变化；
6）WiFi 模组请求重启 SOC；
7）非法消息通知；
8）WiFi 模组读取设备的当前状态；
9）设备 SOC 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）

大部分的基础通信协议代码机智云已经为大家实现了，所以我们特别关注 8、9、10 三条命令即可。

我们先关注命令 10 如下：

4.10 WiFi模组控制设备

WiFi模组发送：

[redacted]	action(1B)	attr_flags(1B)	attr_vals(6B)	[redacted]
[redacted]	0x01	是否设置标志位	设置数据值	[redacted]

注：SOC 版代码无需关注 P0 协议区以外的协议内容，后文同理。

对应上面“p0 数据区约定”中的**功能 1）**“模块向 SOC 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**”，可知：“action(1B)”代表 p0 命令、“attr_flags(1B）”代表命令标志位、“attr_vals(6B)”代表可写数据区。

那程序中如何识别呢，往下看协议的注解：

1. 命令标志位(**attr_flags**)表示相关的数据值是否为有效值，相关的标志位为“1”表示值有效，为“0”表示值无效，从右到左的标志位依次为：

标志位	功能
bit0	设置 LED_OnOff
bit1	设置 LED_Color
bit2	设置 LED_R
bit3	设置 LED_G
bit4	设置 LED_B
bit5	设置 Motor_Speed

这里可以清楚的看到 **attr_flags** 占 1B 字节，其中 bit0 代表：设置 LED_OnOff.....bit5：设置 Motor_Speed，那么对于我们的 SOC 接收到 WIFI 发来的控制命令后，我们通过识别 **attr_flags** 的每一位即可对应出需要控制的设备。

2. 设置数据值(**attr_vals**(6B)) 即可写数据区，定义如下：

2. 设置数据值(attr_vals)存放数据值，只有相关的设置标志位为1时，数据值才有效。例如数据包为0x07 FE FE FE 00 0A 时，其格式为:

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；
byte1		0xFE	LED_R，类型为uint8，字段值为254； 实际值计算公式y=1.000000*x+(0.000000) x最小值为0，最大值为254

这里可以清楚的看到，只有相关的设置标志位（**attr_flags**）为 1 时，数据值才是有效的，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。

1.3 协议分析总结

“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

至此**“p0 数据区约定”**的解析到此结束，之后我们还会分析 SOC 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
app	用户目录（开发者主要关注）
bin	固件生成目录
include	模组驱动相关库
ld	动态链接库
lib	工程文件
tools	相关工具
readme.txt	Gokit3S 文档介绍
user guide V0.3.pdf	Gokit3S 二次开发导读

2.1.2 代码文件说明

```
├── app
│   ├── driver
│   │   ├── hal_key.c                //按键驱动程序
│   ├── gen_misc.bat
│   ├── gen_misc.sh                  //编译工具，执行./gen_misc.sh
│   ├── Gizwits
│   │   ├── gizwits_product.c        //产品相关的处理函数，如gizEventProcess()
│   │   ├── gizwits_product.h        //gizwits_product.c头文件，主要定义软硬件版本号
│   │   ├── gizwits_protocol.c       //gizwits协议相关的处理模块，API的封装等
│   │   ├── gizwits_protocol.h       //gizwits_protocol.c头文件，包括协议相关结构体，数据点相关结构体等
│   ├── include
│   │   ├── driver
│   │   │   ├── hal_key.h            //hal_key.c头文件
│   │   ├── ssl
│   │   └── user
│   │       └── user_main.c          //程序入口函数user_init()所在文件，包括各模块的初始化，task创建等
├── bin
│   ├── at
│   │   ├── 1024+1024
│   │   ├── 512+512
│   │   ├── noboot
│   ├── _temp_by_dltool
│   └── upgrade
│       ├── user1.4096.new.6.bin     //编译生成的执行文件，烧录使用
├── include
│   ├── gagent_external.h            //gagent接口头文件
├── ld
├── lib
│   ├── libgagent.a                  //gagent封装库文件
├── tools
└── 开发指南V0.1.pdf                 //使用说明
```

主要文件说明：

文件	说明
libgagent.a	该文件为机智云设备接入协议库文件,文件位于 lib 目录下
gagent_external.h	该文件为 libgagent.a 对应头文件,两个文件配合使用
gizwits_product.c	该文件为平台相关处理文件，存放事件处理 API 接口函数，即 ***gizwitsEventProcess***()
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	该文件为协议实现文件，存放 SDK API 接口函数
gizwits_protocol.h	该文件为 gizwits_protocol.c 对应头文件，协议相关宏定义以及 API 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。 该函数中完成了数据上报等相关操作。
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(gizEventProcess)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*/
#pragma pack(1)
typedef struct {
  bool valueLED_OnOff;                          ///< 对应数据点：LED_OnOff 读写类型：可写 数
  LED_COLOR_ENUM_T valueLED_Color;              ///< 对应数据点：LED_Color 读写类型：可写 数
  uint32_t valueLED_R;                          ///< 对应数据点：LED_R 读写类型：可写 数据类
  uint32_t valueLED_G;                          ///< 对应数据点：LED_G 读写类型：可写 数据类
  uint32_t valueLED_B;                          ///< 对应数据点：LED_B 读写类型：可写 数据类
  int32_t valueMotor_Speed;                     ///< 对应数据点：Motor_Speed 读写类型：可写
  bool valueInfrared;                           ///< 对应数据点：Infrared 读写类型：只读 数
  int32_t valueTemperature;                     ///< 对应数据点：Temperature 读写类型：只读
  uint32_t valueHumidity;                       ///< 对应数据点：Humidity 读写类型：只读 数
  bool valueAlert_1;                            ///< 对应数据点：Alert_1 读写类型：报警 数据
  bool valueAlert_2;                            ///< 对应数据点：Alert_1 读写类型：报警 数据
  bool valueFault_LED;                          ///< 对应数据点：Fault_LED 读写类型：故障 数
  bool valueFault_Motor;                        ///< 对应数据点：Fault_Motor 读写类型：故障
  bool valueFault_TemHum;                       ///< 对应数据点：Fault_TemHum 读写类型：故障
  bool valueFault_IR;                           ///< 对应数据点：Fault_IR 读写类型：故障 数
} 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，其数据位分别对应“**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：

[redacted]	action(1B)	dev_status(11B)	[redacted]
[redacted]	0x04	设备状态	[redacted]

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" */
typedef struct {
  uint8_t flagLED_OnOff:1;                      ///< 对应数据点：LED_OnOff 读写类型
  uint8_t flagLED_Color:1;                      ///< 对应数据点：LED_Color 读写类型
  uint8_t flagLED_R:1;                          ///< 对应数据点：LED_R 读写类型：可
  uint8_t flagLED_G:1;                          ///< 对应数据点：LED_G 读写类型：可
  uint8_t flagLED_B:1;                          ///< 对应数据点：LED_B 读写类型：可
  uint8_t flagMotor_Speed:1;                    ///< 对应数据点：Motor_Speed 读写类
} attrFlags_t;

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

[redacted]	action(1B)	attr_flags(1B)	attr_vals(6B)	[redacted]
[redacted]	0x01	是否设置标志位	设置数据值	[redacted]

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" */
typedef struct {
  uint8_t wBitBuf[COUNT_W_BIT];                        ///< 可写型数据点 布尔和枚举变量 所占
  uint8_t valueLED_R;                                  ///< 对应数据点：LED_R 读写类型：可写
  uint8_t valueLED_G;                                  ///< 对应数据点：LED_G 读写类型：可写
  uint8_t valueLED_B;                                  ///< 对应数据点：LED_B 读写类型：可写
  uint16_t valueMotor_Speed;                           ///< 对应数据点：Motor_Speed 读写类型
  uint8_t rBitBuf[COUNT_R_BIT];                        ///< 只读型数据点 布尔和枚举变量 所占
  uint8_t valueTemperature;                            ///< 对应数据点：Temperature 读写类型
  uint8_t valueHumidity;                               ///< 对应数据点：Humidity 读写类型：只
  uint8_t valueAlert_1:1;                              ///< 对应数据点：Alert_1 读写类型：报
  uint8_t valueAlert_2:1;                              ///< 对应数据点：Alert_1 读写类型：报
  uint8_t valuereserve_2:6;                            ///< 数据位补齐
  uint8_t valueFault_LED:1;                            ///< 对应数据点：Fault_LED 读写类型：
  uint8_t valueFault_Motor:1;                          ///< 对应数据点：Fault_Motor 读写类型
  uint8_t valueFault_TemHum:1;                         ///< 对应数据点：Fault_TemHum 读写类型
  uint8_t valueFault_IR:1;                             ///< 对应数据点：Fault_IR 读写类型：故
  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：

[redacted]	action(1B)	dev_status(11B)	[redacted]
[redacted]	0x04	设备状态	[redacted]

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-soc-esp8266\app\user\user_main.c** 中 user_init() 函数：

```
229 /**
230  * @brief 程序入口函数
231  *
232  * 在该函数中完成用户相关的初始化
233  * @param none
234  * @return none
235  */
236 void ICACHE_FLASH_ATTR user_init(void)
```

说明：该函数作为整个系统的程序入口初始化了 Gagent 模块和 Gizwits 协议模块这两个主要的部分，其中跟开发者有关的是函数是 ***gizwitsInit()***、***userTimerFunc()***、***gizwitsUserTask()***，相关说明：

函数	说明
gizwitsInit()	协议解析处理模块初始化函数（协议 API）
userTimerFunc()	定时器回调函数（100ms 定时周期，与时间相关的开发逻辑可以在这里实现）
gizwitsUserTask()	用户事件回调函数，用户可在该函数中完成相应任务的处理

2.3.3 用户程序初始化

位置：**user_main.c** 中 ***“//user_init 相关程序”***

```
272     //user init
273     //rgb led init
274     rgbGpioInit();
275     rgbLedInit();
276
277     //key init
278     keyInit();
279
280     //motor init
281     motorInit();
282     motorControl(MOTOR_SPEED_DEFAULT);
283
284     //temperature and humidity init
285     dh11Init();
286
287     //Infrared init
288     irInit();
289
290     //gizwits InitSIG_UPGRADE_DATA
291     gizwitsInit();
292
293     system_os_task(gagentProcessRun, USER_TASK_PRIO_1, TaskQueue, TaskQueueLen);
```

这部分完成了 RGB LED、按键、电机、温湿度、红外传感器的硬件驱动调用，对应的驱动程序实现都在 **gokit-soc-esp8266\app\driver** 下。

其中完成了定时器初始化（详情查看 2.3.4 节）：

```
314     //user timer
315     os_timer_disarm(&userTimer);
316     os_timer_setfn(&userTimer, (os_timer_func_t *)userTimerFunc, NULL);
317     os_timer_arm(&userTimer, USER_TIME_MS, 1);
```

以及系统任务初始化（详情查看 2.3.5 节）：

```
293     system_os_task(gagentProcessRun, USER_TASK_PRIO_1, TaskQueue, TaskQueueLen);
```

2.3.4 定时器使用

代码位置：**app\user\user_main.c 中的 *user_init()函数***

```
314     //user timer
315     os_timer_disarm(&userTimer);
316     os_timer_setfn(&userTimer, (os_timer_func_t *)userTimerFunc, NULL);
317     os_timer_arm(&userTimer, USER_TIME_MS, 1);
```

相关宏定义：

```
61  /**@name 用户定时器相关参数
62   * @{
63   */
64  #define USER_TIME_MS 100                                    ///< 定时时间，单位:
65  LOCAL os_timer_t userTimer;                                 ///< 用户定时器结构体
66  #define TH_TIMEOUT                  (1000 / USER_TIME_MS)   ///< Temperature and
67  #define INF_TIMEOUT                 (500 / USER_TIME_MS)    ///< Infrared detecti
```

API 说明：

os_timer_setfn

功能:
设置定时器回调函数。使用定时器，必须设置回调函数。

函数定义:
```
void os_timer_setfn(
os_timer_t *ptimer,
os_timer_func_t *pfunction,
void *parg
)
```

参数:
os_timer_t *ptimer : 定时器结构
os_timer_func_t *pfunction : 定时器回调函数
void *parg : 回调函数的参数

os_timer_arm

功能:
使能毫秒级定时器

函数定义:
```
void os_timer_arm (
os_timer_t *ptimer,
uint32_t milliseconds,
bool repeat_flag
)
```

参数:
os_timer_t *ptimer : 定时器结构
uint32_t milliseconds : 定时时间，单位：毫秒
bool repeat_flag : 定时器是否重复

回调函数说明：

```
152 /**
153  * 用户数据获取
154  *
155  * 此处需要用户实现除可写数据点之外所有传感器数据的采集,可自行定义采集频率和设计数据过滤算法
156  * @param none
157  * @return none
158  */
159 void ICACHE_FLASH_ATTR userTimerFunc(void)
```

在 **userTimerFunc()** 中完成了周期 100ms 的定时执行，开发者可以在 **user_handle()** 中实现定时读取外设数据的操作，将读取到的数据赋值到用户区的全局结构体变量：

```
171     if(INF_TIMEOUT < irCtime)
172     {
173         irCtime = 0;
174
175         curIr = irUpdateStatus();
176         currentDataPoint.valueInfrared = curIr;
177     }
```

2.3.5 系统任务的使用

代码位置：**app\user\user_main.c 中的 *user_init()函数***

```
293     system_os_task(gagentProcessRun, USER_TASK_PRIO_1, TaskQueue, TaskQueueLen);
```

API 使用说明：

ESP8266 系统任务说明

system_os_task

功能:
创建系统任务

函数定义:
```
bool system_os_task(
 os_task_t task,
 uint8 prio,
 os_event_t *queue,
 uint8 qlen
)
```

参数:
os_task_t task : 任务函数
uint8 prio : 任务优先级，当前支持 3 个优先级的任务: 0/1/2; 0 为最低优先级
os_event_t *queue : 消息队列指针
uint8 qlen : 消息队列深度

system_os_post

功能:
向任务发送消息

函数定义:
```
bool system_os_post (
 uint8 prio,
 os_signal_t sig,
 os_param_t par
)
```

参数:
uint8 prio : 任务优先级，与建立时的任务优先级对应。
os_signal_t sig : 消息类型
os_param_t par : 消息参数

回调函数说明：

```
206 void ICACHE_FLASH_ATTR gizwitsUserTask(os_event_t * events)
207 {
208     uint8_t i = 0;
209     uint8_t vchar = 0;
210
211     if(NULL == events)
212     {
213         os_printf("!!! gizwitsUserTask Error \n");
214     }
215
216     vchar = (uint8)(events->par);
217
218     switch(events->sig)
219     {
220     case SIG_UPGRADE_DATA:
221         gizwitsHandle((dataPoint_t *)&currentDataPoint);
222         break;
223     default:
224         os_printf("---error sig! ---\n");
225         break;
226     }
227 }
```

开发者可以自定义系统任务(**system_os_post** 中的消息类型)，然后在系统任务回调函数中(**gizwitsUserTask**)添加对应的任务处理(即 switch 中对应的消息类型)。

需要注意的是：**任务优先级**不可随意修改（共有三个优先级，提供给开发者的是优先级 0）：

```
312     system_os_task(gizwitsUserTask, USER_TASK_PRIO_0, userTaskQueue, userQueueLen);
```

2.4 配置模式说明

开发者只有先调用“WiFi 配置接口”API 才能使 WiFi 模组进入相应的配置模式，进而完成联网、云端通信的等功能。

“WiFi 配置接口”API 位置：**gokit_mcu_stm32_xxx\Gizwits\gizwits_protocol.h**

```
902 /**
903  * @brief WiFi配置接口
904  *
905  * 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
906  *
907  * @param[in] mode 配置模式选择：0x0， 模组复位 ;0x01， SoftAp模式 ;0x02， AirLink模式
908  * @return 错误命令码
909  */
910 void ICACHE_FLASH_ATTR gizwitsSetMode(uint8_t mode)
```

在本示例工程中是通过**按键触发**进入相应的配置模式，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。

```
115 LOCAL void ICACHE_FLASH_ATTR key2ShortPress(void)
116 {
117     os_printf("#### key2 short press, soft ap mode \n");
118
119     rgbControl(250, 0, 0);
120     gizwitsSetMode(WIFI_SOFTAP_MODE);
121 }
```

B. 进入 AirLink 模式：key2 按键长按。

```
128 LOCAL void ICACHE_FLASH_ATTR key2LongPress(void)
129 {
130     os_printf("#### key2 long press, airlink mode\n");
131
132     rgbControl(0, 250, 0);
133
134     gizwitsSetMode(WIFI_AIRLINK_MODE);
135 }
```

C. 模组复位：key1 按键长。

```
103 LOCAL void ICACHE_FLASH_ATTR key1LongPress(void)
104 {
105     os_printf("#### key1 long press, default setup\n");
106     gizMSleep();
107     gizwitsSetMode(WIFI_RESET_MODE);
108 }
```

注：开发者可以按照自己的需求来实现配置模式。

2.5 协议处理函数的实现

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

该函数被 Gagent 模块调用，处理来自云端或 APP 端的相关 p0 数据协议。

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“**gizwitsIssued_t *gizIssuedData**”它的作用是保存解析出来的协议包头：

```
770     gizwitsIssued_t *gizIssuedData= (gizwitsIssued_t *)&inData[1];
```

协议格式对应协议“4.10 WiFi 模组控制设备”中“P0 协议区”的标志位"attr_flags" + 数据值"attr_vals"

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##

- 然后是各协议命令的处理流程：

```
784     switch(inData[0])
785     {
786         case ACTION_CONTROL_DEVICE:
787             gizDataPoint2Event(gizIssuedData, &gizwitsProtocol.issuedProcessEvent,&gizwi
788
789             system_os_post(USER_TASK_PRIO_2, SIG_ISSUED_DATA, 0);
790             *outLen = 0;
791             break;
792
793         case ACTION_READ_DEV_STATUS:
794             gizDataPoints2ReportData(&gizwitsProtocol.gizLastDataPoint,&gizwitsProtocol.
795             gizwitsProtocol.reportData.action = ACTION_READ_DEV_STATUS_ACK;
```

由于 SOC 版相对 MCU 版去掉了串口协议等概念，故开发者在只需了解《xxx 机智云接入串口通信协议文档》中的 8、10 三条指令：

> 8）WiFi 模组读取设备的当前状态；
> 10）WiFi 模组控制设备；

- 下面以以协议 4.8 的处理为例：

4.8 WiFi模组读取设备的当前状态

WiFi模组发送：

[redacted]	action(1B)	[redacted]
[redacted]	0x02	[redacted]

其“action”值 为“0x02”，对应程序中的的 case 为“**ACTION_READ_DEV_STATUS**”

```
793         case ACTION_READ_DEV_STATUS:
794             gizDataPoints2ReportData(&gizwitsProtocol.gizLastDataPoint,&gizwitsProtocol.reportDat
795             gizwitsProtocol.reportData.action = ACTION_READ_DEV_STATUS_ACK;
796             os_memcpy(outData, (uint8_t *)&gizwitsProtocol.reportData, sizeof(gizwitsReport_t));
797             *outLen = sizeof(gizwitsReport_t);
798             break;
```

之后完成了上报数据的数据类型转化（转化后的数据存储在 ***gizwitsReport_t*** 中的 ***devStatus*** 数据位中）：

```
794             gizDataPoints2ReportData(&gizwitsProtocol.gizLastDataPoint,&gizwits
```

最后将待上报的数据以指针拷贝的方式进行输出：

```
795             gizwitsProtocol.reportData.action = ACTION_READ_DEV_STATUS_ACK;
796             os_memcpy(outData, (uint8_t *)&gizwitsProtocol.reportData, sizeof(gizwitsReport_t));
797             *outLen = sizeof(gizwitsReport_t);
```

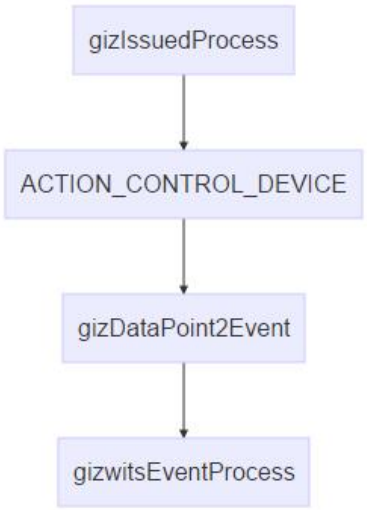
- 同理其他协议 action 值对应的宏定义的位置在 **Gizwits\gizwits_protocol.h** 中:

```
269 /** P0 command 命令码*/
270 typedef enum
271 {
272     ACTION_CONTROL_DEVICE        = 0x01,            ///< 协议4.10 WiFi模组控制设备 WiFi模组发送
273     ACTION_READ_DEV_STATUS       = 0x02,            ///< 协议4.8 WiFi模组读取设备的当前状态 WiFi
274     ACTION_READ_DEV_STATUS_ACK   = 0x03,            ///< 协议4.8 WiFi模组读取设备的当前状态 设备
275     ACTION_REPORT_DEV_STATUS     = 0X04,            ///< 协议4.9 设备MCU向WiFi模组主动上报当前状
276     ACTION_W2D_TRANSPARENT_DATA = 0x05,             ///< WiFi到设备MCU透传
277     ACTION_D2W_TRANSPARENT_DATA = 0x06,             ///< 设备MCU到WiFi透传
278 } action_type_t;
```

以上便是 p0 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
giziIssuedProcess	该函数被 gagent 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 控制型事件的生成

相关代码位置：

app\Gizwits\gizwits_protocol.c 中 gizDataPoint2Event*()* 函数：

功能说明：

在该函数中完成了写类型外设事件的生成，以“红灯开关数据点”为例：

```
596   if(0x01 == issuedData->attrFlags.flagLED_OnOff)
597   {
598       info->event[info->num] = EVENT_LED_ONOFF;
599       info->num++;
600       dataPoints->valueLED_OnOff = gizDecompressionValue(LED_ONOFF_BYTEOFFSET,LED_ONOFF_BITOFFSET,
601   }
```

这里对应协议“**4.10 WiFi 模组控制设备**”：

4.10 WiFi模组控制设备

WiFi模组发送：

[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	action(1B)	attr_flags(1B)	attr_vals(6B)	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	0x01	是否设置标志位	设置数据值	[redacted]

前面我们已经知道程序里的“***issuedData->attr_flags***”就对应《微信宠物屋-机智云接入串口通信协议文档.pdf》中的“**4.10 WiFi 模组控制设备**”中的 **attr_flags**(1B)，作用是用来控制所选位的设备，在文档中我们可以看到 attr_flags 的第 0 位是用来选择控制 LED 灯开关的，即只要设置了第 0 位为 1 就表示要控制 LED 等开关了，代码中对应如下：

```
605   if(0x01 == issuedData->attrFlags.flagLED_Color)
```

接下来便是控制型事件的生成：

```
598           info->event[info->num] = EVENT_LED_ONOFF;
599           info->num++;
```

以及完成数据的解压（详情请查看“2.8.2 数据解压与压缩处理”一节）：

```
600   dataPoints->valueLED_OnOff = gizDecompressionValue(LED_ONOFF_BYTEOFFSET,LED_ONOFF_BITO
```

注意：枚举（如 ***EVENT_LED_ONOFF***）用来直观的表示事件的含义，用户自行添加、更改（位置：**app\Gizwits\gizwits_protocol.h**）

```
181   EVENT_LED_ONOFF,                    ///< 红灯开关控制事件
182   EVENT_LED_COLOR,                    ///< LED组合颜色控制事件
183   EVENT_LED_R,                        ///< LED红色值控制事件
184   EVENT_LED_G,                        ///< LED绿色值控制事件
185   EVENT_LED_B,                        ///< LED蓝色值控制事件
186   EVENT_MOTOR_SPEED,                  ///< 电机转速控制事件
187   EVENT_TYPE_MAX                      ///< 枚举成员数量计算 (用户误删)
188 } EVENT_TYPE_T;
```

2.6.2 控制型事件处理

代码位置:

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

功能说明：

完成写类型外设事件的处理。

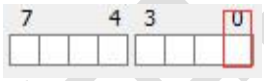
```
60      switch(info->event[i])
61      {
62        case EVENT_LED_ONOFF:
63          currentDataPoint.valueLED_OnOff = dataPointPtr->valueLED_OnOff;
64          os_printf("Evt: EVENT_LED_ONOFF %d \n", currentDataPoint.valueLED_OnOff);
65          if(0x01 == currentDataPoint.valueLED_OnOff)
66   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

这段程序功能的控制 LED 灯的开关：LED 开关控制位“***issued->attr_vals.led_onoff***”的值若是 LED_On（0x01）表示灯开，为 LED_Off（0x01）表示灯关。这对应《微信宠物屋-机智云接入串口通信协议文档.pdf》中的“**4.10 WiFi 模组控制设备**”中的 attr_vals(6B)，即“数据位”，如下所示：

2. 设置数据值(attr_vals)存放数据值，只有相关的设置标志位为1时，数据值才有效。例如数据包为 0x07 FE FE FE 00 0A 时，其格式为:

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；

第 0 位用来控制红灯亮灭，对应到在云端定义的数据点含义为：



= 0x00 = 红灯灭

= 0x01 = 红灯亮

下面的程序基本和上面一样，只要大家看懂了《xxx-机智云接入串口通信协议文档.pdf》中的“**4.10 WiFi 模组控制设备**”中的 attr_flags(1B) 、attr_vals(6B)这两位就能编写控制型协议的程序了。

2.6.3 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     }
```

特别说明：

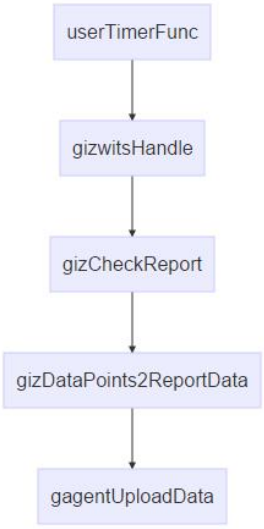
网络字节序转化

数据点为 uint16、uint32 型的数据要考虑**网络字节序转化**（uint16 即使用 ***exchangeBytes() 函数***），以电机控制为例：

```
151         case SetMotor:
152             os_printf("########## motor speed is %d\n", issued->attr_vals.motor);
153
154             valueMotor = X2Y(MOTOR_SPEED_RATIO, MOTOR_SPEED_ADDITION, exchangeBytes(issued->attr_vals.motor));
155
156             motorControl(valueMotor);
157
158             reportData.dev_status.motor = issued->attr_vals.motor;
159             break;
```

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
userTimerFunc	获取用户区的上报型数据
gizwitsHandle	用户调用该接口可以完成设备数据的变化上报
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
gagentUploadData	将上报数据发送给 WiFi 模块

2.7.1 只读型数据的获取

相关代码:

app\user\user_main.c 中 ***userTimerFunc()*** 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：

```
171     if(INF_TIMEOUT < irCtime)
172     {
173         irCtime = 0;
174 
175         curIr = irUpdateStatus();
176         currentDataPoint.valueInfrared = curIr;
177     }
```

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:

Gizwits\gizwits_protocol.c 中 ***gizCheckReport()*** 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“逻辑 1：控制型数据主动上报当前状态”为例：

```
672     if(last->valueLED_OnOff != cur->valueLED_OnOff)
673     {
674         os_printf("valueLED_OnOff Changed\n");
675         ret = 1;
676     }
```

以“逻辑 2：控制型数据主动上报当前状态”为例：

```
738     if(last->valueTemperature != cur->valueTemperature)
739     {
740         if(gizGetTimerCount()-lastReportTime >= REPORT_TIME_MAX)
741         {
742             os_printf("Temperature Changed\n");
743             lastReportTime = gizGetTimerCount();
744             ret = 1;
745         }
746     }
```

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** 中完成相应的数据的转换。

转换函数说明：

gizDataPoints2ReportData	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检测为例：

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：

```
#define TEMPERATURE_RATIO            1            ///< 环境温度分辨率
#define TEMPERATURE_ADDITION         -13          ///< 环境温度增量
#define TEMPERATURE_MIN              0            ///< 环境温度最小值
#define TEMPERATURE_MAX              200          ///< 环境温度最大值
```

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：

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

B. Y2X 的转换：

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

功能定义更加

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
}


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
}
```

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 *)devStatusPt
gizCompressValue(LED_COLOR_BYTEOFFSET,LED_COLOR_BITOFFSET,LED_COLOR_LEN,(uint8_t *)devStatusPt
gizCompressValue(INFRARED_BYTEOFFSET,INFRARED_BITOFFSET,INFRARED_LEN,(uint8_t *)devStatusPtr,d
gizByteOrderExchange((uint8_t *)devStatusPtr->wBitBuf,sizeof(devStatusPtr->wBitBuf));
```

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：