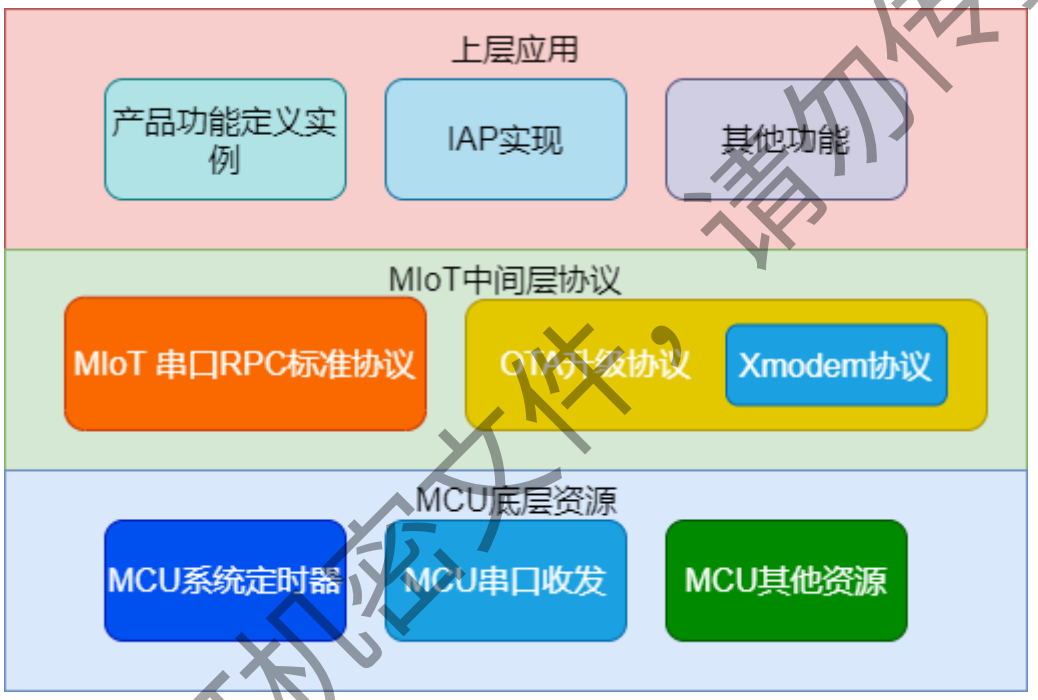
MCU_Demo二次开发手册

1.概述

MCU_Demo是基于“**通用模组+MCU**”的接入方式，开发的一款MCU端demo程序，开发者可以将此demo作为参考来开发自己的产品，加快产品上线速度。MCU_Demo实现了模组与MCU的串口通信、MCU OTA升级等产品接入所需要的最基本的功能（**注**：MCU_Demo 的OTA流程只实现了从模组端下载固件到MCU，并未实现烧写，也没有做OTA异常处理，**demo程序仅供参考**）。

2.代码架构



如上图所示，MCU_Demo的代码结构大致分为三层：

- **MCU底层资源**

主要是MCU底层相关内容，包括：UART、系统定时器、通用GPIO、看门狗等等。**MCU_Demo针对不同MCU的适配主要是在这一层。**

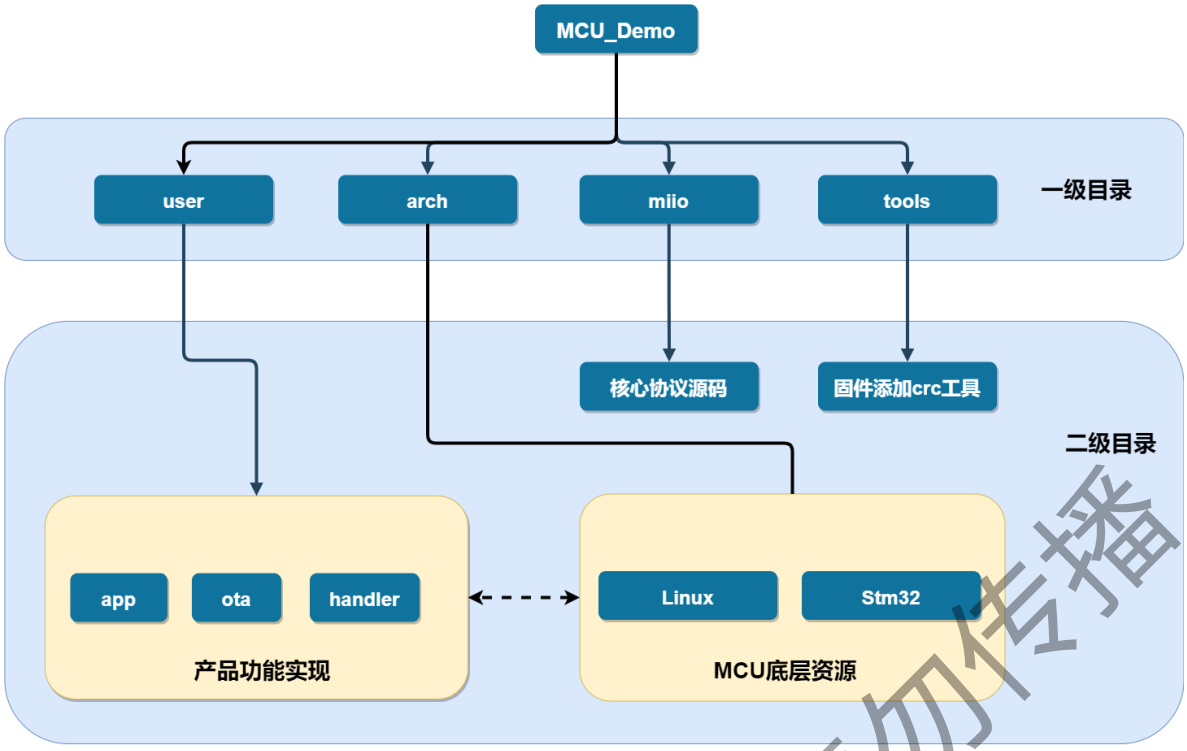
- **MIoT中间层协议**

主要实现与MIoT模组通信相关的协议，包括：与通用模组通信的串口协议、OTA升级协议、Xmodem传输协议等。

- **上层应用**

主要实现产品功能，包括：定义的相关属性、功能实例、产品其他功能等，若产品支持固件升级，则还需要实现MCU IAP功能。

3.源码目录结构



如上图所示，为MCU_Demo源码目录结构：

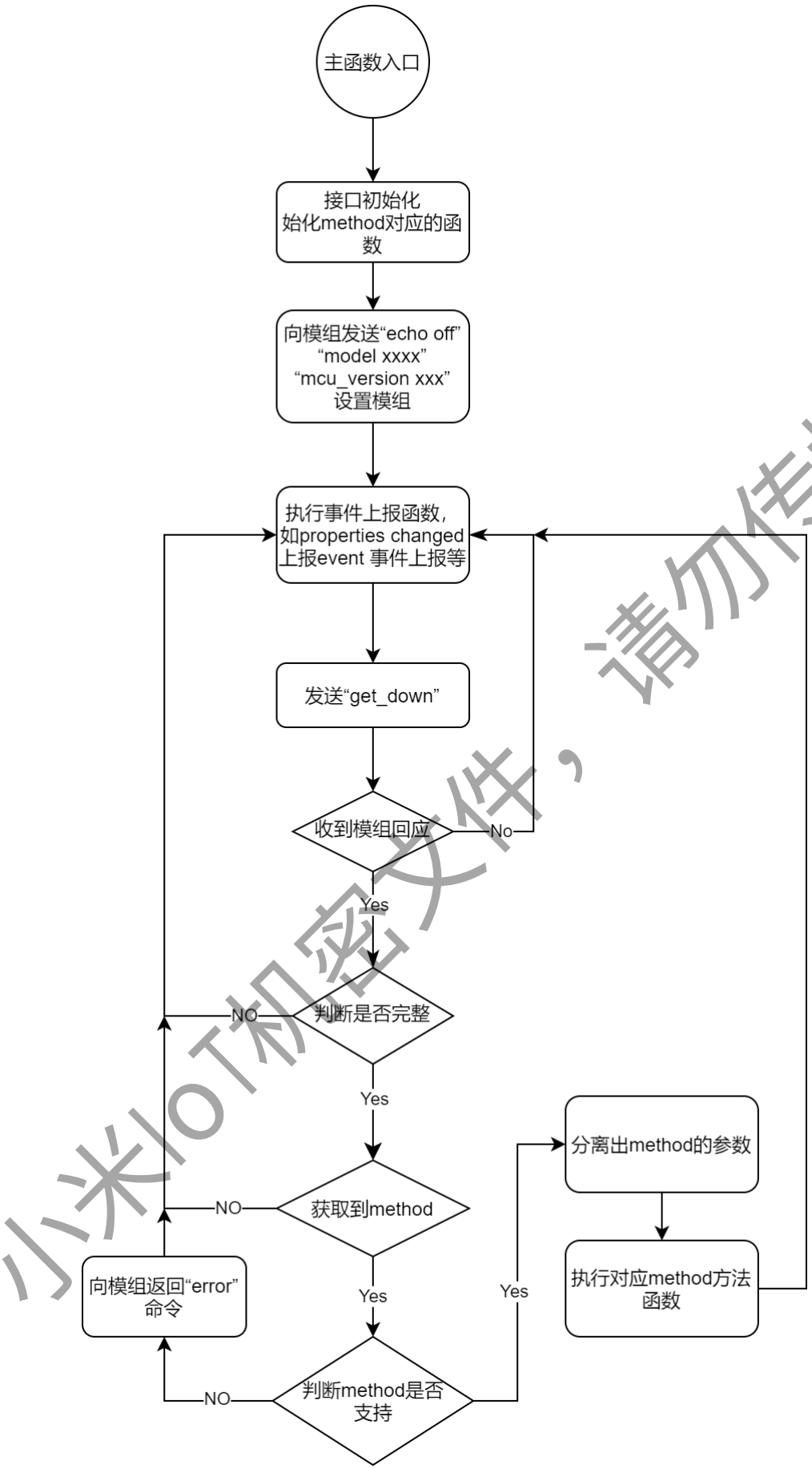
1.arch目录主要存放板级相关代码，以兼容适配不同平台，目前包括Linux和Stm32平台，适配新的MCU后会放到该目录下。

2.user目录主要存放产品功能实现代码，主要包括功能定义的相关内容和ota升级相关内容。

3.tools目录存放固件添加crc工具，只有添加了crc校验的固件，才能完成OTA流程。

4.miio目录，主要存放协议相关代码

4.程序执行流程



1. 初始话各个接口，注册method的回调函数
2. 向模组发送echo off关掉会显，发送model设置模组的model，发送mcu_verision设置MCU软件版本号
3. 进入周期循环，首先判断是否需要上报，如上报properitys_change或者event等

4. 向模组发送get_down方法，并等待模组回复
5. 若模组超时未回复则进入下一个循环
6. 若模组正常回复，检测收到的数据中是否有支持的method，若不支持则返回error命令
7. 若支持该method，则直接调用注册的回调函数，回调函数中会向模组返回执行结果
8. 执行完毕后进入下一个循环

5.在MCU_Demo上开发功能

user/app目录

user/app目录主要存放**用户自定义的功能函数**

- 用户自定义的功能函数，建议**统一放在user/app/user_app_func.c文件中，并统一在user/app/user_app_main.c中的user_app_main()接口中调用**
- 该目录下已经为用户提供了相关例程可直接调试使用，更具体的功能函数开发由用户进行
- 执行properties changed和event occured上报函数

  在user_app_main()中定义了用户执行函数，在此处开发者可以判断是否需要调用属性或者事件上报函数，**此处调用的上报函数并没有判断条件，而是周期性执行，只作为demo测试时使用；正式产品使用时需要添加条件判断**

```
int user_app_main(int argc, void *argv)
{

if(cnt++ % 20 == 0) {//此处周期性上报，只作为测试时使用,正常使用时需要加上函数执行条件判断
    /* spec functions */
    E_4_1_Testevent(argv, 1, false);
    P_3_1_On_doChange(argv, false);
#if 1
        /* add user functions here */
        // app_func_factory(argv);
        app_func_get_time(argv);
        app_func_get_mac(argv);
        app_func_get_net_state(argv);
        // app_func_reboot(argv);
        // app_func_restore(argv);
        // app_func_getwifi(argv);
        // app_func_setwifi(argv);
        app_func_get_version(argv);
        // app_func_set_mcu_version(argv);
        // app_func_get_arch_platform(argv);
#endif
    }

return MIIO_OK;
}
```

user/handler目录

user/handler目录主要存放，[用户在代码自动生成平台生成的函数]，具体内容以用户在平台上定义的功能为准（**DEMO中以小米model：miot.plug.plugv1**为例）

功能函数适配说明

- **开始适配前，用户需要了解小米model和SPEC的相关说明文档**
- 下面以通用模型**miot.plug.plugv1**中的**P_2_1_On_doSet**函数为例，说明用户在代码自动生成平台上获取代码后，如何进行适配操作：

```
static void P_2_1_On_doSet(property_operation_t *o)
{
    /* judge value format */
    if (o->value->format != PROPERTY_FORMAT_BOOLEAN)
    {
        o->code = OPERATION_ERROR_VALUE;
        return;
    }

    /* TODO : execute operation */

    /* return execution result */
    o->code = OPERATION_OK;

    return;
}
```

- 该部分代码放置在**user/handler/S_2_Switch_doSet.c**文件中，**P_2_1_On_doSet**函数主要表示对于`MCU`控制的开关进行通断操作
- 用户**只需要对于函数中标记的TODO部分进行具体实现**：在接收到小米后台下发的控制指令后，DEMO会自动解析Wi-Fi模组的串口指令**down set_properties 2 1 [true]/[false]**并进入到该函数内，用户只需做出**打开/关闭开关**的动作，并将返回结果赋值给结构体指针**property_operation_t *o**的code成员即可（code成员为枚举类型，具体定义可在**property_operation_t**结构体中查看）

user/ota目录

user/ota目录主要存放**MCU OTA升级相关代码**

- 小米已经为用户做好标准的**Xmodem**通信流程，让**MCU**能够顺利从小米**Wi-Fi**模组处获取**MCU**升级固件
- 在用户收到**Wi-Fi**模组传输的**MCU**升级固件后，**更进一步的固件升级操作**，由用户在**user/arch/arch_ota.c**文件中定义的接口函数完成
- **关于用户如何上传MCU固件到小米开发者平台，和如何通过后台指令进行MCU OTA升级，可参阅：小米开发者平台OTA文档**
- MCU_Demo并未实现OTA异常处理，需要开发者根据所开发的产品添加。

user/user_config.h文件

- **user/user_config.h**文件存放用户配置选项，其中为用户准备了**USER_OS_ENABLE**、**USER_OTA_ENABLE**等宏定义开关进行代码的适配，定义了**USER_MODEL**、**USER_MCU_VERSION**等宏定义需要用户修改为开发中采用的**model**和**MCU**版本号

IAP（In applicating Programing)

IAP就是通过软件实现在线电擦除和编程的方法。

- IAP即在应用编程，也就是用户可以使用自己的程序对MCU的中的运行程序进行更新，而无需借助于外部烧写器。

- 目前MCU_Demo中只实现了通过Xmodem协议将固件传输到MCU中，并未实现将固件烧写到MCU flash中，需要根据具体的MCU芯片实现其IAP功能。

6.将MCU_Demo移植到新平台

已有的工程例程介绍

MCU_Demo内自带Linux下和Stm32的工程例程，分别在源码目录mcu_demo/arch/linux和mcu_demo/arch/stm32。

```
gong@ubuntu:/mnt/hgfs/code/test_code/mcu_demo/mcu_demo/arch/linux$ ls -l
total 33
-rwxrwxrwx 1 root root 5300 Jun 11 00:39 app_main.c
-rwxrwxrwx 1 root root  512 Jun 11 00:39 arch_dbg.c
-rwxrwxrwx 1 root root 4475 Jun 11 00:39 arch_dbg.h
-rwxrwxrwx 1 root root 2178 Jun 11 00:39 arch_define.h
-rwxrwxrwx 1 root root  229 Jun 11 00:39 arch_init.c
-rwxrwxrwx 1 root root  921 Jun 11 00:39 arch_init.h
-rwxrwxrwx 1 root root  623 Jun 11 00:39 arch_os.c
-rwxrwxrwx 1 root root 1705 Jun 11 00:39 arch_os.h
-rwxrwxrwx 1 root root  176 Jun 11 00:39 arch_ota.c
-rwxrwxrwx 1 root root 1126 Jun 11 00:39 arch_ota.h
-rwxrwxrwx 1 root root 6599 Jun 11 00:43 arch_uart.c
-rwxrwxrwx 1 root root 3263 Jun 11 00:39 arch_uart.h
-rwxrwxrwx 1 root root 2727 Jun 11 00:39 Makefile
gong@ubuntu:/mnt/hgfs/code/test_code/mcu_demo/mcu_demo/arch/linux$
```

linux平台下直接使用make编译，即可在当前目录下生成可执行文件。

stm32的工程使用Keil集成开发环境打开即可。

新的MCU需要做的适配

MCU_Demo在运行时需要**通信串口**与模组通信，同时需要**调试串口**打印debug信息，因此，MCU至少需要具备**2个串口**资源；另外，由于MCU_Demo内部分功能依赖系统时间，建议MCU具备一个严格的**系统时钟**。根据不同的MCU品牌型号，这部分内容会有差异，所以适配工作主要针对这一块进行。下面以stm32为例说明适配新平台：

stm32的例程中，需要适配的文件全部放在mcu_demo/arch/stm32/device文件夹中。

- **系统函数的适配**

在arch_define.h文件中有如下定义，主要是usleep函数，需要MCU来实现us级系统延迟。同时需要MCU支持malloc内存分配函数、字符串操作函数strtok等。

```
#define arch_usleep(us)                 usleep(us)
#define arch_msleep(ms)                 arch_usleep(ms*1000)

#define arch_memset(str, val, len)      memset(str, val, len)
#define arch_memcpy(dst, src, len)      memcpy(dtong'shist, src, len)

#define arch_malloc(len)                malloc(len)
#define arch_calloc(num, len)           calloc(num, len)

#define arch_strtok(str, temp)          strtok(str, temp)
```

- **串口的适配**

MCU_Demo中会用到两个串口，因此需要MCU端实现通信串口和调试串口这两个串口的适配。

通信串口的适配，主要在mcu_demo/arch/stm32/device/arch_uart.c文件中。

_uart_init通信串口的初始化函数，需要配置串口**波特率115200，8数据位，无奇偶校验，1位停止位，无硬件流控制**，如下是stm32通信串口的初始化函数。

```
uart_error_t _uart_init(miio_uart_t *uart)
{
    do {
        GPIO_InitTypeDef GPIO_InitStructure;
        USART_InitTypeDef USART_InitStructure;
        NVIC_InitTypeDef NVIC_InitStructure;
        RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
        RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2,ENABLE);

        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
        GPIO_Init(GPIOA, &GPIO_InitStructure);

        GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
        GPIO_Init(GPIOA, &GPIO_InitStructure);

        NVIC_InitStructure.NVIC_IRQChannel = USART2_IRQn;
        NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 3 ;
        NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
        NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
        NVIC_Init(&NVIC_InitStructure);

        USART_InitStructure.USART_BaudRate = 115200;
        USART_InitStructure.USART_WordLength = USART_WordLength_8b; /* 8
data bits */
        USART_InitStructure.USART_StopBits = USART_StopBits_1; /* 1 stop
bit */
        USART_InitStructure.USART_Parity = USART_Parity_No; /* no parity
*/
        USART_InitStructure.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;
        USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
```

```
            USART_Init(USART2, &USART_InitStructure);
            USART_ITConfig(USART2, USART_IT_RXNE, ENABLE);
            USART_Cmd(USART2, ENABLE);
        }while(false);
        /* adjust end */
        uart->params.baud_rate = 115200;
        uart->params.data_bits = 8;
        uart->params.parity = 0;
        uart->params.stop_bits = 1;
        return UART_OK;
    }
```

_uart_send_str字符串发送函数，需要实现通过串口发送字符串，如下是stm32通信串口的字符串发送函数。

```
    int _uart_send_str(miio_uart_t *uart, const char* str)
    {
        int len = strlen(str);
        int n_send = 0;
        int t = 0;
        if (len <= 0) { return UART_OK; }

        arch_os_mutex_get(&(uart->write_mutex));
        /* the following is an example for linux platform */
        /* user should adjust below for each mcu platform */
        /* adjust start */

        for(t = 0; t < len; t++) {
            while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);

            USART_SendData(USART2, str[t]);
            n_send++;
        }
        while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);
        /* adjust end*/
        arch_os_mutex_put(&(uart->write_mutex));

        if (n_send < len) {
            LOG_INFO_TAG(MIIO_LOG_TAG, "send string failed");
            return UART_SEND_ERROR;
        }

    #if PRINT_SEND_BUFF
        LOG_INFO_TAG(MIIO_LOG_TAG, "send string : %s", str);
    #endif

        return n_send;

    }
```

_uart_send_byte字符发送函数，需要实现通过串口发送一个字符，如下是stm32通信串口的字符串送函数。

```
    int _uart_send_byte(miio_uart_t *uart, const char c)
    {
        int n_send = 0;
```

```
        arch_os_mutex_get(&(uart->write_mutex));
        /* the following is an example for linux platform */
        /* user should adjust below for each mcu platform */

        /* adjust start */

        while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);
        USART_SendData(USART2, c);
        n_send++;
        while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);
        /* adjust end */

        arch_os_mutex_put(&(uart->write_mutex));

        if (n_send < 1) {
            LOG_INFO_TAG(MIIO_LOG_TAG, "send byte failed : %x[hex]", c);
            return UART_SEND_ERROR;
        }

        LOG_INFO_TAG(MIIO_LOG_TAG, "send byte : %x[hex]", c);
        return n_send;

    }
```

_uart_send_str_wait_ack命令发送函数，需要实现通过串口发送命令，并等待模组回复ok，如下是stm32下_uart_send_str_wait_ack函数的实现。

```
    int _uart_send_str_wait_ack(miio_uart_t *uart, const char* str)
    {
        int len = strlen(str);
        int n_send = 0;
        int t = 0;
        uint8_t ack_buf[ACK_BUF_SIZE] = { 0 };
        if (len <= 0) { return UART_OK; }

        memset(ack_buf, 0, ACK_BUF_SIZE);
        arch_os_mutex_get(&(uart->write_mutex));
        /* the following is an example for linux platform */
        /* user should adjust below for each mcu platform */

        /* adjust start */

        for(t = 0; t < len; t++) {
            while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);
            USART_SendData(USART2, str[t]);
            n_send++;
        }
        while(USART_GetFlagStatus(USART2, USART_FLAG_TC) == RESET);
        /* adjust end */

        arch_os_mutex_put(&(uart->write_mutex));

        if (n_send < len) {
            LOG_INFO_TAG(MIIO_LOG_TAG, "send string wait ack failed 1");
            return UART_SEND_ERROR;
        }
```

```
        #if PRINT_SEND_BUFF
            LOG_INFO_TAG(MIIO_LOG_TAG, "send string : %s", str);
        #endif

            uart->recv_str(uart, ack_buf, ACK_BUF_SIZE, USER_UART_TIMEOUT_MS);
            if (0 != strncmp((const char*)ack_buf, "ok", strlen("ok"))) {
                LOG_INFO_TAG(MIIO_LOG_TAG, "send string wait ack failed 2
str=%s\n", ack_buf);
                return UART_RECV_ACK_ERROR;
            }
            return n_send;
        }
```

USART2_IRQHandler串口接收中断，此中断函数是stm32平台特有的，其他MCU只需实现其相应功能即可；功能是接收串口数据，保存到环形缓冲队列中_write_ringbuff，环形队列在MCU_demo中已经实现，直接调用即可。

```
        void USART2_IRQHandler(void)
        {
            if(USART_GetITStatus(USART2, USART_IT_RXNE) != RESET)
            {
                USART_ClearITPendingBit(USART2,USART_IT_RXNE);
                _write_ringbuff(USART_ReceiveData(USART2));
            }
        }
```

调试串口的适配，主要在mcu_demo\mcu_demo\arch\stm32\device\arch_init.c文件中。

arch_mcu_init函数主要初始化了系统delay函数，和调试串口。

```
        int arch_mcu_init(void)
        {
            delay_init();
            NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
            uart_init(115200);

            LOG_INFO_TAG(TAG, "<****** add mcu init func here ******>");

            return MIIO_OK;

        }
```

调试串口还需要实现printf函数，在keil开发环境中使用printf作为标准输出的方法参考：

```
        1、添加 #include "stdio.h"

        2、重定义fputc函数
            int fputc(int ch,FILE *f)
            {
                USART_SendData(UART4,ch);
                while(!(UART4->SR&USART_FLAG_TXE));
                return(ch);
            }

        3、魔术棒--Target 勾选Use Micro LIB
```

以下是在stm32下具体的代码实现：

```
#if 1
#pragma import(__use_no_semihosting)

struct __FILE
{
    int handle;

};

FILE __stdout;

void _sys_exit(int x)
{
    x = x;
}
/* redirect fputc() */
int fputc(int ch, FILE *f)
{
    while((USART1->SR & 0X40) == 0);
    USART1->DR = (u8) ch;
    return ch;
}
#endif
```

- **arch_ota_func函数ota升级函数的适配**

此函数主要实现MCU在应用升级功能（IAP），具体需要根据不同的MCU平台来实现对应的IAP功能。mcu_demo/arch/linux/arch_ota.c

```
int arch_ota_func(unsigned char *pbuf, size_t len, size_t sum)
{
    /* trans data to MCU flash here */
    return MIIO_OK;
}
```

7.对升级了错误固件的补救处理

如果MCU的应用固件已经损坏，需要一种机制对其再次升级。主要是有两种方法：

第一种：

小米WiFi模组通过判断上次升级是否成功完成，来决定本次上电后是否要再次对MCU升级。流程是小米WiFi模组上电后，检查备份分区里是否有完整的MCU固件。如果有，则再次对MCU升级。升级成功后，擦除备份分区里的MCU固件。这种方法要求小米WiFi模组与MCU必须同时上电。

第二种：

MCU重启自检发现固件损坏，用update_me命令要求小米WiFi模组对其再次升级。自检的一种推荐方式见下图。
注：无特殊情况不建议使用“update_me”再次触发升级。

推荐的MCU流程图：

左图为mcu bootloader流程，中图为mcu应用流程，右图是mcu应用固件损坏后的情况。