

ESP32 操作指南

作者:向仔州

ESP32 环境搭建	5
ESP32 IDF 环境搭建方法.....	5
有了 IDF 库的环境搭建基础，下面来搭建 MDF 库环境.....	9
MakeFile:8:/project.mk: No such file or directory.....	9
打开烧写软件.....	11
在 VScode 环境下搭建 ESP32 工程.....	13
make: ****没有规则可制作目标 “menuconfig”	13
make: ***No rule to make target “menuconfig”.....	15
编译过程中官方问题居多的情况，不好解决。如果有现成的交叉编译器压缩包，请跳过此页.....	15
有些库，比如 IDF/MDF 在 make menuconfig 的时候，报错 check_python_dependencies.....	17
ESP32 MDF 创建第 1 个工程.....	18
Error: 'TAG' undeclared	19
ESP32 错误处理宏，在调用 ESP32 API 函数的过程中大部分都使用了错误宏包含.....	19
ESP_ERROR_CHECK(...)//API 执行错误类型判断.....	20
ESP32 NVS 存储器，类似 MCU EEPROM.....	21
nvs_flash_init(); //初始化 NVS.....	21
nvs_flash_erase() //擦除 NVS 存储器.....	21
nvs_open(...)//打开 NVS 存储表.....	21
nvs_set_str(...)//写字符串存储.....	21
nvs_set_i32(...)//写 32 位有符号整数.....	21
nvs_set_blob(...)//写无固定长度(如结构体)存储.....	22
nvs_commit(...)//保存句柄的数据进 flash.....	22
nvs_close(...)//关闭打开的句柄.....	22
读取 NVS 存储器里面的值.....	23
nvs_get_str(...)//读出 NVS flash 字符串.....	23
nvs_get_i32(...)//读取 NVS 整形.....	23
nvs_get_blob(...)//读出 NVS flash 结构体参数.....	25
CPU 0:main CPU IDLE1 运行问题.....	25
ESP32 WIFI 连接热点.....	25
tcpip_adapter_init(); //使用 wifi 前一定要先初始化 tcpip 适配器.....	25
esp_event_loop_init(...);//wifi 进入 STA 模式后需要一个状态机不断的维护当前 wifi 状态，向 esp_event_loop_init 注册一个回调函数来处理当前维护的 wifi 状态..	25
esp_wifi_set_mode(...)//设置 wifi 工作模式.....	25
esp_wifi_set_config(...); //设置 wifi 在一种(STA/AP)模式下，需要的参数.....	25
esp_wifi_start(void) //启动 wifi.....	25
Wifi 初始化和启动之前要先设置好事件回调状态机.....	26
SYSTEM_EVENT_STA_START //wifi 启动第一次进入事件.....	27
SYSTEM_EVENT_STA_CONNECTED //连接路由器成功事件.....	27
SYSTEM_EVENT_STA_GOTIP // 获取到 IP 后，重启 WiFi 热点事件.....	27
SYSTEM_EVENT_STA_DISCONNECTED //路由器连接被断开事件.....	27

报错 assertion "0 && "mconfig_bluifi_init(&bluifi_config)" failed。其实就是 esp_bt_mem_release 函数问题.....	55
SNTP 服务器配置，获取网络标准时间.....	57
sntp_setserver(.....) //通过 IP 地址设置 SNTP 服务器.....	57
sntp_setservername(.....) //通过域名设置 SNTP 服务器.....	57
第 1: 先连接上 wifi 网络，外网打开.....	58
第 2: 获取 SNTP 服务器时间.....	58
sntp_setoperatingmode(.....) //设置 SNTP 工作模式.....	58
sntp_init() //SNTP 客户端初始化函数.....	58
setenv() //时区设置.....	58
tzset(); //设置时间环境变量.....	58
time(time_t *timep) //获取 ESP32 本地时间，就算网断了也能一直按照 CPU 时 钟节拍获取.....	59
struct tm *localtime_r(.....)//是可重入函数，线程安全，将 timep 为 long 形的 数据解析成年月日时分秒.....	59
size_t strftime(....)//根据 format 中定义的格式化规则，格式化结构 timeptr 表示的时间，并把它存储在 str 中。.....	60
如果你不需要把时间的年月日时分秒做成一个 long 型传递，你就可以直接用 strftime 显示，方便些.....	60
sntp_set_time_sync_notification_cb(....)//设置 NTP 同步回调函数，在函数里面 证明获取 NTP 服务时间是否正常.....	60
sntp_get_sync_status(void) //获取 SNTP 服务同步是否更新完成.....	61
sntp_setoperatingmode(....) //设置 SNTP 模式.....	61
还有一种阻塞方式同步 SNTP，程序卡死在 SNTP 位置，等待 SNTP 同步完成，程 序再运行.....	62
sntp_sync_time(....); //未注册回调可以使用时间更新函数同步时间.....	62
Wifi mesh 网络使用（IDF 使用方法，有点问题）建议看 MDF.....	62
tcpip_adapter_dhcps_stop(....) //停止 DHCP 服务器.....	63
tcpip_adapter_dhcpc_stop(....) //停止 DHCP 客户端.....	63
esp_mesh_init(void) //mesh 网络初始化.....	63
MESH_INIT_CONFIG_DEFAULT(); //mesh 网络配置需要分配结构体.....	63
esp_mesh_set_config(....) //配置 mesh 网络.....	63
esp_mesh_start(void) //启动 mesh 网络.....	63
int esp_mesh_get_layer(void) //获取当前设备属于那一层节点.....	66
获取本设备当前是不是 mesh 网络根节点.....	67
esp_mesh_is_root(void) //判断本设备是不是根节点.....	67
ESP32 MDF wifi mesh 使用.....	69
mdf_event_loop_init(....) //mesh 启动后事件回调函数.....	69
MWIFI_INIT_CONFIG_DEFAULT(); //分配 mesh 网络结构.....	69
mwifi_init(....) //mesh 网络初始化.....	69
mwifi_set_config(....) //配置 mesh 网络.....	69
mwifi_start() //启动 mesh 网络.....	69
mdf_event_loop_cb_t 设置的 mesh 事件回调函数事件值.....	69
MDF_EVENT_MWIFI_STARTED mesh 网络自组网成功会自动执行一次该事 件.....	69
MDF_EVENT_MWIFI_PARENT_CONNECTED 本设备与父节点连接上,触发该事 件.....	69
MDF_EVENT_MWIFI_PARENT_DISCONNECTED 父节点与子节点断开触发该事 件.....	69
MDF_EVENT_MWIFI_ROUTING_TABLE_ADD 新添加子接点会触发路由表更 改.....	69

MDF_EVENT_MWIFI_ROUTING_TABLE_REMOVE 子节点断开会触发路由表删除.....	69
MDF_EVENT_MWIFI_ROOT_GOT_IP.....	69
esp_mesh_get_total_node_num(void) //获取当前 mesh 网络设备总数.....	69
esp_mesh_get_parent_bssid(...) //获取父节点的 BSSID.....	71
mwifi_get_parent_rssi() //子节点获取父节点信号强度.....	62
esp_mesh_get_routing_table_size(void) //获取本设备下的子节点数量，不包含本设备上一级父节点的设备数.....	73
EPS32 获取本模块的一些网络信息.....	74
esp_efuse_mac_get_default(...)//获取本机出厂默认的 MAC 地址.....	74
tcpip_adapter_get_ip_info(...)//获取本机被路由器分配的 IP 地址，掩码，网关.....	74
ESP32 连接网络 ping 实现.....	76
esp_ping_set_target(...) //ping 功能配置函数.....	76
ping_init(void) //ping 启动函数，在执行之前一定配置好 esp_ping_set_target 函数.....	76
ping_deinit(void) //ping 停止函数.....	78
ESP32 AP 模式 与另外一台 ESP32 STA 设备或者手机 STA 设备相连进行客户端到服务器数据传输.....	79
先设定 ESP32 AP 的本地 IP 地址.....	79
tcpip_adapter_set_ip_info(...)//设置本地设备 IP 地址.....	79
tcpip_adapter_get_ip_info(...)//查询本地 IP 地址.....	79
ESP32 TCP 客户端测试.....	80
客户端向服务器单向发数据.....	80
服务器向客户端发送数据，双向发送.....	83
在客户端模式下设置客户端本地端口.....	84
AP 模式下 TCP 服务端实现.....	85
ESP32 wifi MESH 网络数据转发.....	88
esp_mesh_set_group_id(...)//设置组 ID.....	88
mwifi_write(...)//发送数据给 mesh 网络中指定 mac 地址的节点.....	88
mwifi_read(...) //读取父节点发过来的数据.....	88
mwifi_is_connected(void) //本节点是否连接上父节点.....	89
mwifi_write(...)//向根节点或者父节点发送数据，也是用 mwifi_write.....	71
根节点自动识别，子节点自动识别，自动识别目的 MAC 地址，自动识别源 MAC.....	94
esp_mesh_get_parent_bssid(...) //获取父节点 MAC 地址.....	94
多个子节点设备同时启动，那么都有成为根节点的可能，那么如何相互通信呢？	100
mwifi_root_read(...) //根节点读取子节点数据.....	100
子节点如何知道根节点与服务器已经连接成功？	109
mwifi_post_root_status(...) //根节点发送标志,其实其它节点也可以发送....	109
mwifi_get_root_status() //子节点接收根节点的标志.....	109

ESP32 环境搭建

ESP32 IDF 环境搭建方法

esp32_win32_msys2_environment_and_toolchain-20181001.zip

esp-idf-v3.3.2.zip

去乐鑫官方或者论坛下载这两个文件

esp32_win32_msys2_environment_and_toolchain-20170111 版本绝对不行的，要用 20181001
不然你会出现 VS CODE 打开 terminal 终端出现闪退。

esp32_win32_msys2_environment_and_toolchain-20181001 是交叉编译器

esp-idf-v3.3.2 是代码例程包

解压缩，一定放在全英文路径下。

esp-idf-v3.3.2 代码例程目录

msys32 交叉编译器目录

进去 msys32 目录 msys32\etc\profile.d

名称	修改日期	类型
bash_completion.sh	2018-06-07 19:02	Shell Script
esp32_toolchain.sh	2020-09-15 9:17	Shell Script
lang.sh	2018-08-08 15:49	Shell Script
perlbin.csh	2018-06-28 20:11	CSH 文件
perlbin.sh	2018-06-28 20:11	Shell Script
tzset.sh	2018-08-08 15:49	Shell Script

修改 esp32_toolchain.sh 文件

```
# This file was created by ESP-IDF windows_install_prerequisites.sh
# and will be overwritten if that script is run again.
export PATH="$PATH:/opt/xtensa-esp32-elf/bin"
export IDF_PATH="D:/xzz_ESP32/esp-idf-v3.3.2"
```

这两行代码决定了你每次打开 ESP32 命令行终端执行哪一个路径

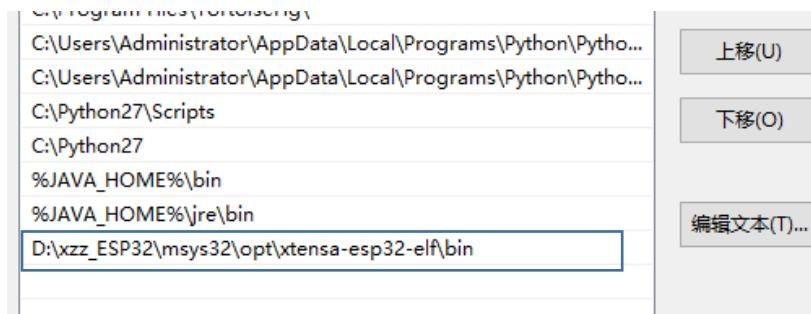
export PATH="\$PATH:/opt/xtensa-esp32-elf/bin" 这句是指定当前目录下的交叉编译器，一般不用修改。

export IDF_PATH="D:/xzz_ESP32/esp-idf-v3.3.2" 这一句必须要修改，指定你当前例程目录
esp-idf-v3.3.2\examples\mesh\manual_networking 我跑的是 idf 版本库的 mesh 网络

名称	修改日期	类型
build	2020-09-14 16:54	文件夹
main	2020-04-09 18:04	文件夹
CMakeLists.txt	2020-04-09 18:04	文本文档
Makefile	2020-04-09 18:04	文件
README.md	2020-04-09 18:04	MD 文件
sdkconfig	2020-09-14 15:54	文件

所以 IDF_PATH 改成以下路径

```
export IDF_PATH="D:/xzz_ESP32/esp-idf-v3.3.2"
```



环境变量要不要加交叉编译器路径我不确定

进入 msys32 目录，打开终端

M	mingw32.exe	2016-09-08 4:37	应用程序 50 KB
IMG	mingw32.ini	2016-09-08 4:37	配置设置 1 KB
M	mingw64.exe	2016-09-08 4:37	应用程序 50 KB
IMG	mingw64.ini	2016-09-08 4:37	配置设置 1 KB
M	msys2.exe	2016-09-08 4:37	应用程序 50 KB
M	msys2.ico	2018-08-08 15:49	ICO 图片文件 26 KB
IMG	msys2.ini	2016-09-08 4:37	配置设置 1 KB
IMG	msys2_shell.cmd	2018-08-08 15:49	Windows 命令脚本 7 KB
IMG	network.xml	2018-10-01 10:00	XML 文档 1 KB

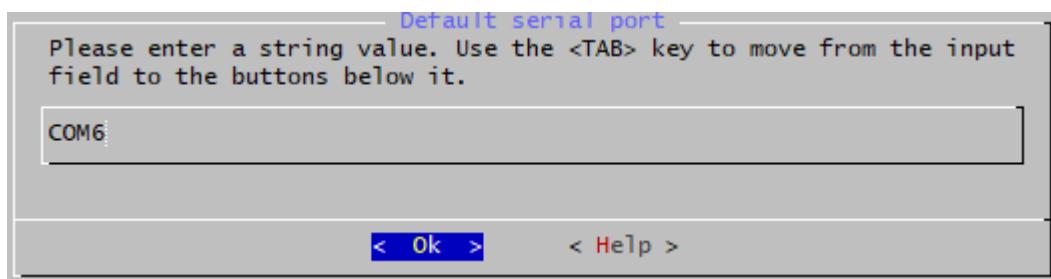
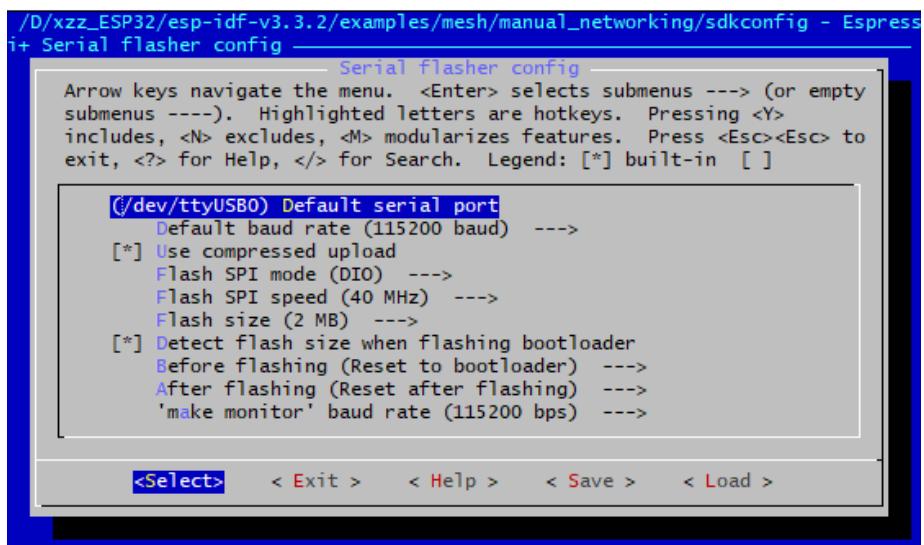
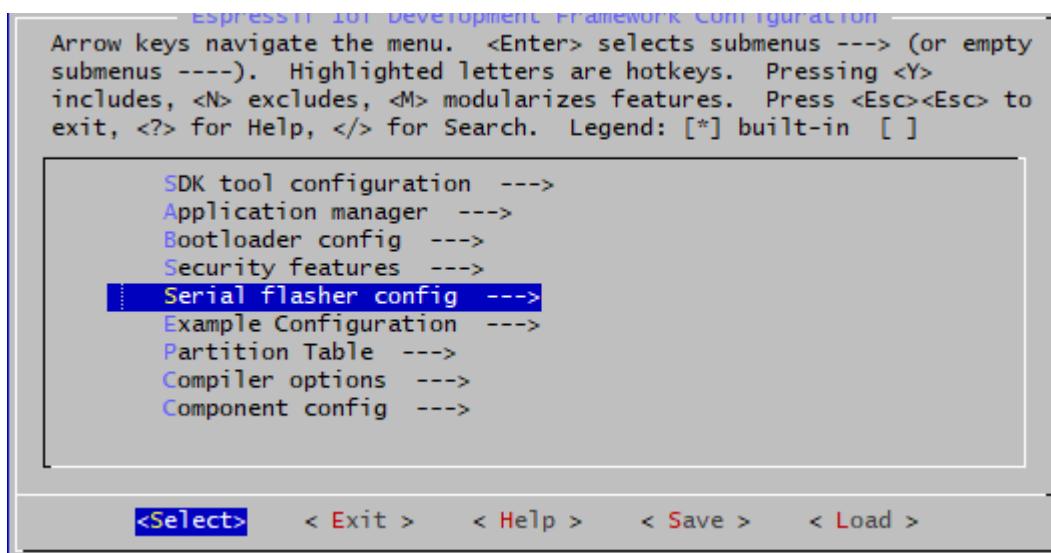
记住编译 idf 的例程用 mingw32. exe。其它 mingw64，或者 msys2 都会报/bin/sh: python: not found 问题

```
Administrator@3339DD8YRYOXD9A MINGW32 ~
$ cd /D
Administrator@3339DD8YRYOXD9A MINGW32 /D
$ cd xzz_ESP32/esp-idf-v3.3.2/examples/mesh/manual_networking/
```

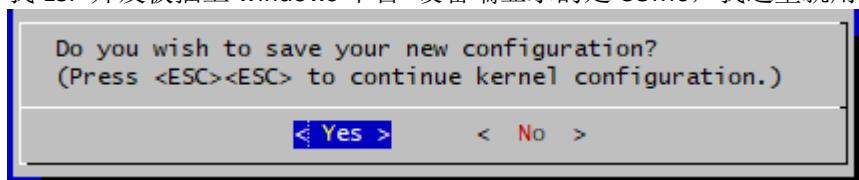
进入 manual_networking 目录

```
$ make menuconfig
make[1]: 进入目录"/d/xzz_ESP32/esp-idf-v3.3.2/tools/kconfig"
/usr/bin/gcc -c -I/usr/include/ncursesw -DCURSES_LOC=<curses.h> -DNCURSES_WIDECHAR=1 -DLOCALE -MMD -MP /d/xzz_ESP32/esp-idf-v3.3.2/tools/kconfig/mconf.c -o mconf.o
flex -L -Pzconf -ozconf.lex.c /d/xzz_ESP32/esp-idf-v3.3.2/tools/kconfig/zconf.l
bison -t -l -p zconf -o zconf.tab.c /d/xzz_ESP32/esp-idf-v3.3.2/tools/kconfig/zconf.y
sed -E "s/^\x0D$//" /d/xzz_ESP32/esp-idf-v3.3.2/tools/kconfig/zconf.gperf | gperf -t --output-file zconf.hash.c -a -C -E -g -k '1,3,$' -p -t
/usr/bin/gcc -I /d/xzz_ESP32/esp-idf-v3.3.2/tools/kconfig -c -I/usr/include/ncursesw -DCURSES_LOC=<curses.h> -DNCURSES_WIDECHAR=1 -DLOCALE -MMD -MP zconf.tab.c -o zconf.tab.o
```

执行 make menuconfig



我 ESP 开发板插上 windows 平台 设备端显示的是 COM6，我这里就用 COM6 做串口输出



直接 make

```
$ make
Toolchain path: /opt/xtensa-esp32-elf/bin/xtensa-esp32-elf-gcc
Toolchain version: crosstool-ng-1.22.0-80-g6c4433a5
Compiler version: 5.2.0
CC build/bootloader/bootloader_support/src/bootloader_random.o
CC build/bootloader/bootloader_support/src/flash_encrypt.o
CC build/bootloader/bootloader_support/src/bootloader_sha.o
CC build/bootloader/bootloader_support/src/esp_image_format.o
CC build/bootloader/bootloader_support/src/flash_partitions.o
CC build/bootloader/bootloader_support/src/bootloader_flash_config.o
CC build/bootloader/bootloader_support/src/bootloader_clock.o
CC build/bootloader/bootloader_support/src/secure_boot.o
CC build/bootloader/bootloader_support/src/bootloader_common.o
CC build/bootloader/bootloader_support/src/bootloader_utility.o
CC build/bootloader/bootloader_support/src/bootloader_init.o
CC build/bootloader/bootloader_support/src/secure_boot_signatures.o
CC build/bootloader/bootloader_support/src/flash_qio_mode.o
CC build/bootloader/bootloader_support/src/bootloader_flash.o
```

```
esptool.py v2.8
To flash all build output, run 'make flash' or:
python /d/xzz_ESP32/esp-idf-v3.3.2/components/esptool_py/esptool/esptool.py --chip esp32 --port COM6 --baud 115200 --before default_reset --after hard_reset write_flash -z --flash_mode dio --flash_freq 40m --flash_size detect 0x1000 /D/xzz_ESP32/esp-idf-v3.3.2/examples/mesh/manual_networking/build/bootloader/bootloader.bin 0x10000 /D/xzz_ESP32/esp-idf-v3.3.2/examples/mesh/manual_networking/build/manual_networking.bin 0x8000 /D/xzz_ESP32/esp-idf-v3.3.2/examples/mesh/manual_networking/build/partitions_singleapp.bin

Administrator@3339DD8YRY0XD9A MINGW32 /D/xzz_ESP32/esp-idf-v3.3.2/examples/mesh/manual_networking
```

编译成功，在 build 目录下会生成三个 bin 文件

名称	修改日期	类型	大小
bootloader_support	2020-09-15 9:40	文件夹	
efuse	2020-09-15 9:40	文件夹	
esptool_py	2020-09-15 9:40	文件夹	
log	2020-09-15 9:40	文件夹	
main	2020-09-15 9:40	文件夹	
micro-ecc	2020-09-15 9:40	文件夹	
soc	2020-09-15 9:40	文件夹	
spi_flash	2020-09-15 9:40	文件夹	
bootloader.bin	2020-09-15 9:40	BIN 文件	26 KB
bootloader.elf	2020-09-15 9:40	ELF 文件	189 KB
bootloader.map	2020-09-15 9:40	MAP 文件	306 KB
component_project_vars.mk	2020-09-15 9:39	MK 文件	1 KB

第 1 个文件是 bootloader.bin 需要下载

examples > mesh > manual_networking > build

xtensa-debug-module	2020-09-15 9:43	文件夹
ldgen.section_infos	2020-09-15 9:43	SECTION_INFOS...
manual_networking.bin	2020-09-15 9:43	BIN 文件
manual_networking.elf	2020-09-15 9:43	ELF 文件
manual_networking.map	2020-09-15 9:43	MAP 文件
partitions_singleapp.bin	2020-09-15 9:40	BIN 文件

第 2 和第 3 个是在 build 目录下的 bin 文件，需要下载。

有了 IDF 库的环境搭建基础，下面来搭建 MDF 库环境

 **esp-mdf-v1.0.zip** 去官方下载 esp-mdf-v1.0 版本，这是最稳定的版本，不建议尝试其它的 MDF 版本。

esp-idf-v3.3.2	2020-04-09 18:04	文件夹
esp-mdf-v1.0	2020-09-15 8:41	文件夹
msys32	2020-09-14 15:34	文件夹

交叉编译我们还是使用 msys32，修改 msys32 的 msys32/etc/profile.d/esp32_toolchain.sh

```
esp32_toolchain.sh
1 # This file was created by ESP-IDF windows_install_prerequisites.sh
2 # and will be overwritten if that script is run again.
3 export PATH="$PATH:/opt/xtensa-esp32-elf/bin"
4 #export IDF_PATH="D:/xzz_ESP32/esp-idf-v3.3.2"
5 export IDF_PATH="D:/xzz_ESP32/esp-mdf-v1.0"
```

屏蔽#export IDF_PATH="D:/xzz_ESP32/esp-idf-v3.3.2" IDF 库

加入 export IDF_PATH="D:/xzz_ESP32/esp-mdf-v1.0" MDF 库 保存，重新启动终端 mingw32.exe

我们编译 esp-mdf-v1.0\examples\function_demo\mcommon 例程试试

ESP32 > esp-mdf-v1.0 > examples > function_demo > mcommon

名称	修改日期	类型
main	2020-09-15 8:41	文件夹
CMakeLists.txt	2020-05-26 16:13	文本文档
Makefile	2020-05-26 16:13	文件
partitions.csv	2020-05-26 16:13	XLS 工作表
README.md	2020-05-26 16:13	MD 文件
README_cn.md	2020-05-26 16:13	MD 文件
sdkconfig.defaults	2020-05-26 16:13	DEFAULTS 文件

```
Administrator@3339DD8YRY0XD9A MINGW32 /D/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon
$ ls
CMakeLists.txt  Makefile      README.md      sdkconfig.defaults
main           partitions.csv  README_cn.md

Administrator@3339DD8YRY0XD9A MINGW32 /D/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon
$ make menuconfig
Makefile:8: /project.mk: No such file or directory
make: *** 没有规则可制作目标“/project.mk”。 停止。
```

我发现编译不过 MakeFile:8:/project.mk: No such file or directory

```
XZ666@XZ666PC MINGW32 /d/ESP32system/esp-mdf-v1.0/esp-idf/examples/bluetooth/blufi
$ make clean
Makefile:10: /make/project.mk: No such file or directory
make: *** No rule to make target '/make/project.mk'. Stop.
```

还有这种都属于 MakeFile:8 这种类似情况。

```

ESP32 > esp-mdf-v1.0 > examples > function_demo > mcommon
名称          修改日期      类型
build        2020-09-15 10:04  文件夹
main         2020-09-15 8:41   文件夹
CMakeLists.txt 2020-05-26 16:13  文本文档
Makefile      2020-05-26 16:13  文件
partitions.csv 2020-05-26 16:13  XLS 工作簿

```

```

3 # project subdirectory.
4 #
5
6 PROJECT_NAME := $(notdir $(shell pwd))
7
8 include ${MDF_PATH}/project.mk
9

```

你例程里面的 Makefile 是 MDF_PATH 变量

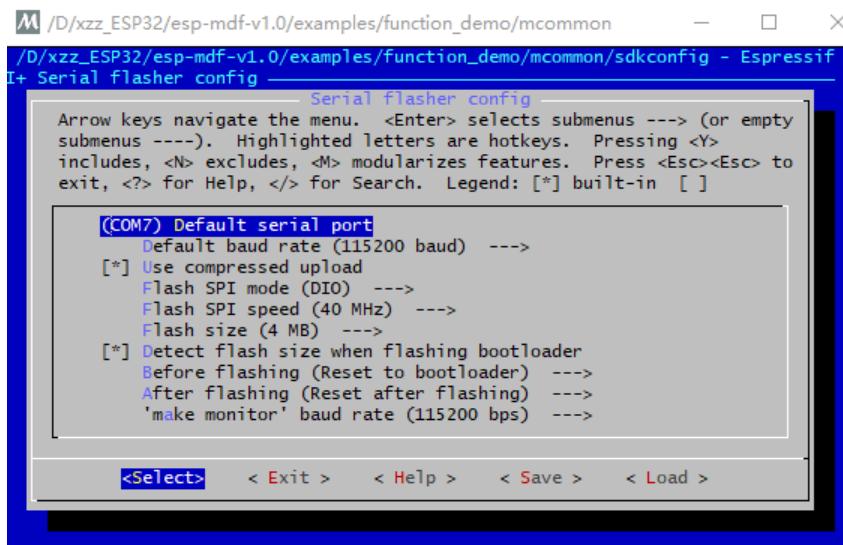
```

# This file was created by ESP-IDF windows_install_prerequisites.sh
# and will be overwritten if that script is run again.
export PATH="$PATH:/opt/xtensa-esp32-elf/bin"
#export IDF_PATH="D:/xzz_ESP32/esp-idf-v3.3.2"
export MDF_PATH="D:/xzz_ESP32/esp-mdf-v1.0"

```

那么你的 esp32_toolchain.sh 就必须是改成 MDF_PATH，这样终端打开才会设置和 Makefile 一样的变量

然后重新打开 mingw32.exe
make menuconfig



然后 make

```

Administrator@3339DD8YRY0XD9A MINGW32 /D/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon
$ MAKE
make
Toolchain path: /opt/xtensa-esp32-elf/bin/xtensa-esp32-elf-gcc
Toolchain version: crosstool-ng-1.22.0-80-g6c4433a5
Compiler version: 5.2.0
App "mcommon" version: v1.0-dirty
CC build/bootloader/bootloader_support/src/bootloader_random.o
CC build/bootloader/bootloader_support/src/flash_encrypt.o
CC build/bootloader/bootloader_support/src/bootloader_sha.o
CC build/bootloader/bootloader_support/src/esp_image_format.o
CC build/bootloader/bootloader_support/src/flash_partitions.o
CC build/bootloader/bootloader_support/src/bootloader_flash_config.o
CC build/bootloader/bootloader_support/src/bootloader_clock.o
CC build/bootloader/bootloader_support/src/secure_boot.o
CC build/bootloader/bootloader_support/src/bootloader_common.o

```

开始编译 MDF 库了

```

esptool.py v2.8
To flash all build output, run 'make flash' or:
python /d/xzz_ESP32/esp-mdf-v1.0/esp-idf/components/esptool_py/esptool/esptool.py --chip esp32 --port COM7 --baud 115200 --before default_reset --after hard_reset write_flash -z --flash_mode dio --flash_freq 40m --flash_size detect 0xd000 /D/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/ota_data_initial.bin 0x1000 /D/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/bootloader/bootloader.bin 0x10000 /D/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/mcommon.bin 0x8000 /D/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/partitions.bin

```

编译成功，生成文件

我们把路径整理出来

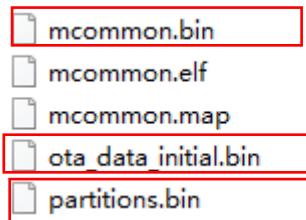
```

python /d/xzz_ESP32/esp-mdf-v1.0/esp-idf/components/esptool_py/esptool/esptool.py --chip esp32 --port COM7 --baud 115200 .....
0xd000 /D/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/ota_data_initial.bin
0x1000 /D/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/bootloader/bootloader.bin
0x10000 /D/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/mcommon.bin
0x8000 /D/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/partitions.bin

```

就是这 4 个 bin 文件，串口 COM7，波特率 115200

在 build 目录下有



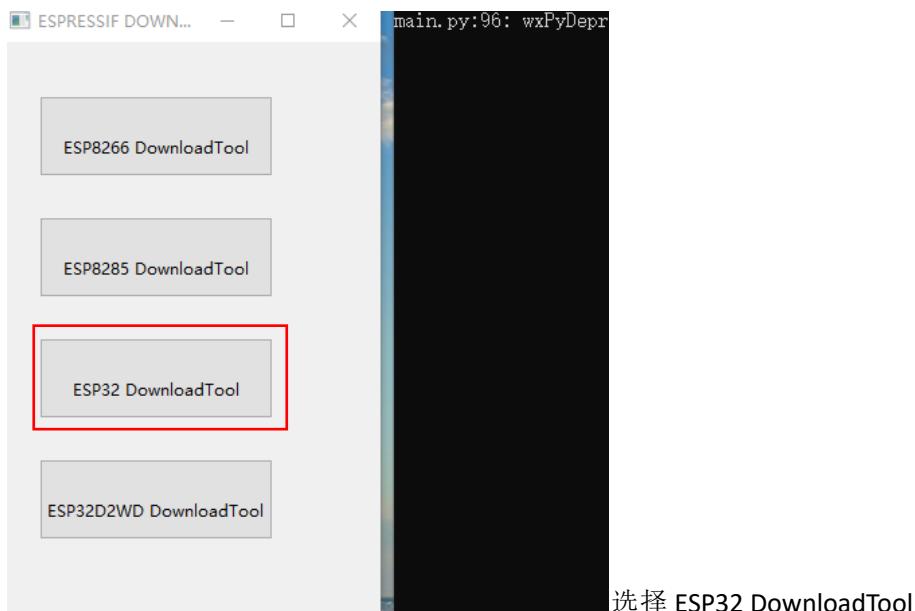
名称	修改日期	类型	大小
bootloader_support	2020-09-15 10:10	文件夹	
efuse	2020-09-15 10:10	文件夹	
esptool_py	2020-09-15 10:10	文件夹	
log	2020-09-15 10:10	文件夹	
main	2020-09-15 10:10	文件夹	
micro-ecc	2020-09-15 10:10	文件夹	
soc	2020-09-15 10:10	文件夹	
spi_flash	2020-09-15 10:10	文件夹	
bootloader.bin	2020-09-15 10:10	BIN 文件	26 KB
bootloader.elf	2020-09-15 10:10	ELF 文件	189 KB
bootloader.map	2020-09-15 10:10	MAP 文件	304 KB
component_project_vars.mk	2020-09-15 10:10	MK 文件	1 KB

在 bootloader 目录下有 bootloader.bin 文件

我们现在将这 4 个文件下载进去看看

打开烧写软件





File	Address
ota_data_initial.bin	0xd000
bootloader.bin	0x1000
mcommon.bin	0x10000
partitions.bin	0x8000

打开每一个 bin 文件都要填入它对应的地址

0xd000 ota_data_initial.bin

0x1000 bootloader.bin

0x10000 mcommon.bin

0x8000 partitions.bin

Download Panel 1

FINISH 完成

AP: FCF5C43C4A1D STA: FCF5C43C4A1C
BT: FCF5C43C4A1E ETHERNET: FCF5C43C4A1F

Download Panel 1

IDLE 等待

START STOP ERASE COM: COM6 BAUD: 921600

点击 START

在 VScode 环境下搭建 ESP32 工程

还是以 esp-mdf-v1.0/examples/function_demo/mcommon 为例

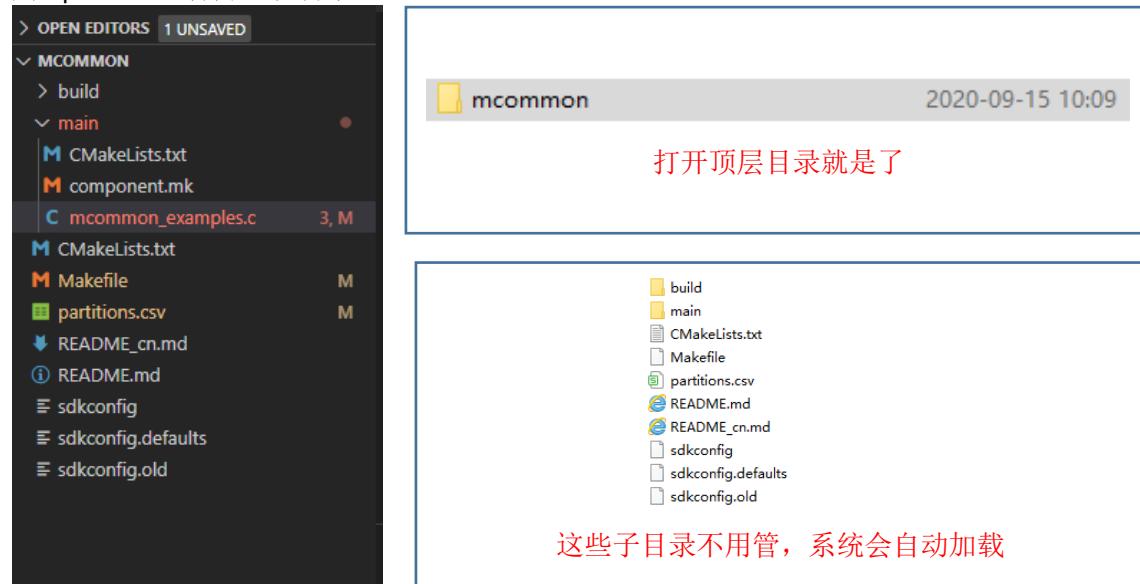
在 esp-mdf-v1.0\examples\function_demo\mcommon\main 目录下修改 main.c

```
void app_main()
{
    while(1){
        MDF_LOGI("*****");
        MDF_LOGI("          Log output      ");
        MDF_LOGI("*****");
        sleep(1);
    }
}
```

入口函数就说 app_main()

打印要加延时，不然电脑串口会死机，反正我公司的电脑是这个情况。

用 open folder 打开工程目录



修改 setting.json 文件

```
"terminal.integrated.shell.windows": "D:\\xzz_ESP32\\msys32\\msys2_shell.cmd",

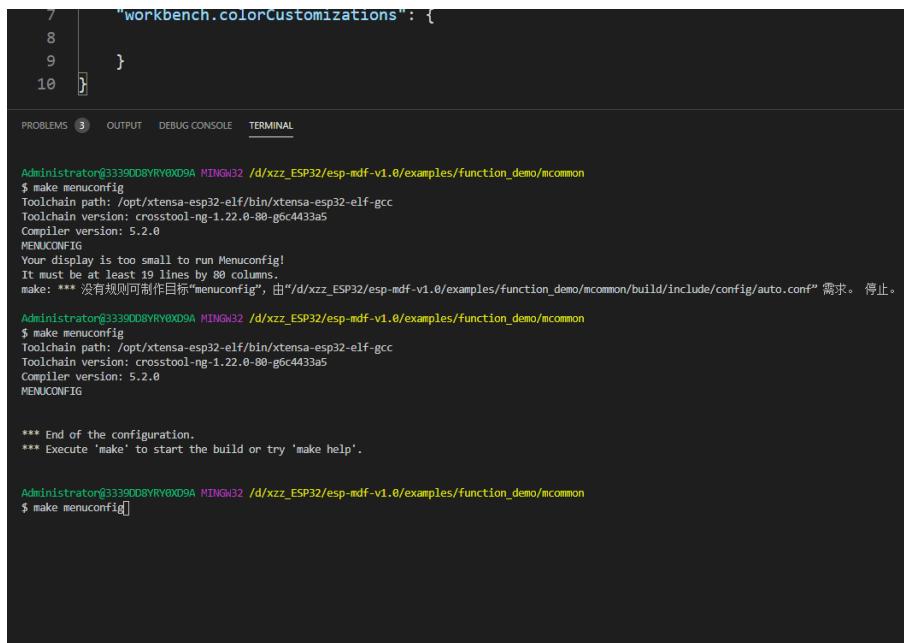
"terminal.integrated.shellArgs.windows": ["-defterm", "-mingw32", "-no-start", "-here"],
"terminal.integrated.automationShell.linux": "",
"editor.fontSize": 20,
"workbench.colorCustomizations": {

}
```

虽然你在终端用的是 mingw32.exe，但是在 vscode 你还是指定 msys2_shell 路径才可以
你在终端 make menuconfig 的时候报错

```
$ make menuconfig
toolchain path: /opt/xtensa-esp32-elf/bin/xtensa-esp32-elf-gcc
Toolchain version: crosstool-ng-1.22.0-80-g6c4433a5
Compiler version: 5.2.0
MENUCONFIG
Your display is too small to run Menuconfig!
It must be at least 19 lines by 80 columns.
make: *** 没有规则可制作目标“menuconfig”，由“/d/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/include/config/auto.conf”需求。停止。
```

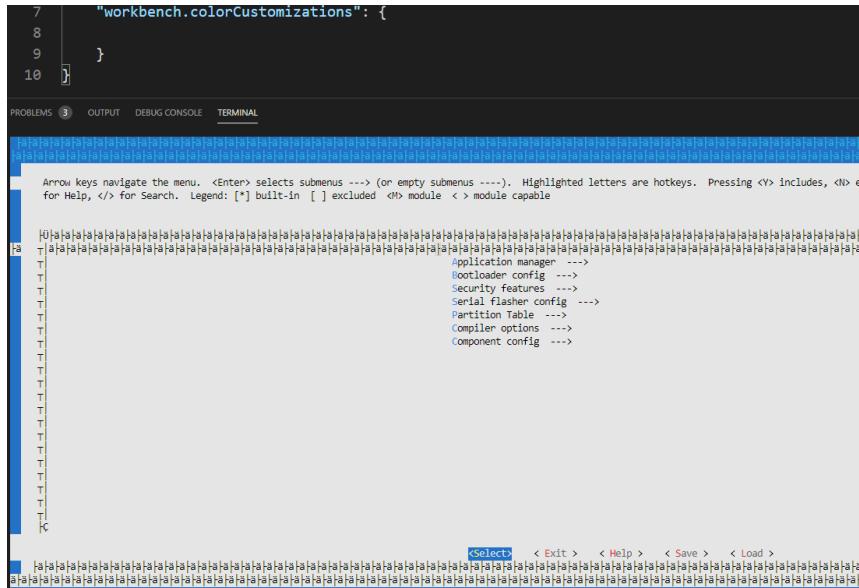
这个错误是你的 VScode 终端窗口没有拉大造成的



```
Administrator@33390DB8YRY0XD9A MINGW32 /d/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon
$ make menuconfig
Toolchain path: /opt/xtensa-esp32-elf/bin/xtensa-esp32-elf-gcc
Toolchain version: crosstool-ng-1.22.0-80-g6c4433a5
Compiler version: 5.2.0
MENUCONFIG
Your display is too small to run Menuconfig!
It must be at least 19 lines by 80 columns.
make: *** 没有规则可制作目标“menuconfig”，由“/d/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/include/config/auto.conf” 需求。 停止。
Administrator@33390DB8YRY0XD9A MINGW32 /d/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon
$ make menuconfig
Toolchain path: /opt/xtensa-esp32-elf/bin/xtensa-esp32-elf-gcc
Toolchain version: crosstool-ng-1.22.0-80-g6c4433a5
Compiler version: 5.2.0
MENUCONFIG
*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.
```

Administrator@33390DB8YRY0XD9A MINGW32 /d/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon
\$ make menuconfig[6]

你把终端窗口拉这么大就没有问题了



```
Administrator@33390DB8YRY0XD9A MINGW32 /d/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon
$ make menuconfig
Arrow keys navigate the menu. <Enter> selects submenus --> (or empty submenus ---). Highlighted letters are hotkeys. Pressing <> includes, <> excludes, <> module capable
for Help, <> for Search. Legend: [*] built-in [ ] excluded <> module
[*] Application manager -->
    [ ] Toolchain config -->
        [ ] Security features -->
        [ ] Serial flasher config -->
        Partition Table -->
        Compiler options -->
        Component config -->
```

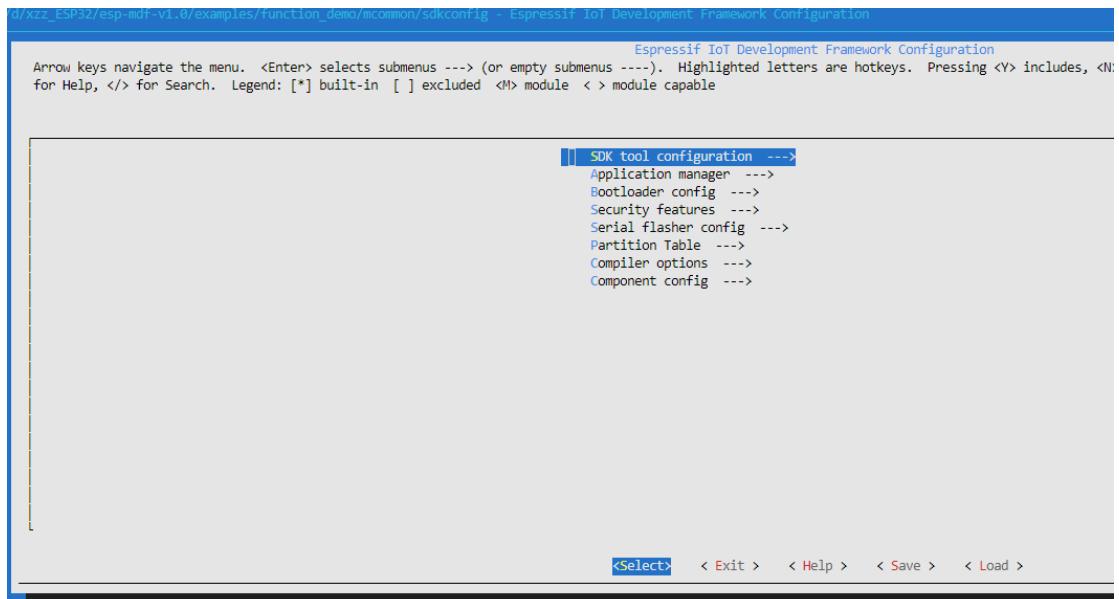
你看，menuconfig 界面出来了，但是是乱码。

这个问题还是要回到你的 msys32 编译器目录 msys32\etc\profile.d\esp32_toolchain.sh 去解决

export LANG="en_US" 将这行代码填入 esp32_toolchain.sh

```
# and will be overwritten if that script is run again
export LANG="en_US"
export PATH="$PATH:/opt/xtensa-esp32-elf/bin"
#export IDF_PATH="D:/xzz_ESP32/esp-idf-v3.3.2"
export MDF_PATH="D:/xzz_ESP32/esp-mdf-v1.0"
```

重启 VScode 搞定



你看界面没有乱码了

如果 make menuconfig 窗口拉大了还是出现 make: ***No rule to make target “menuconfig”..

```
$ make menuconfig
MENUCONFIG
make: *** No rule to make target 'menuconfig', needed by '/d/ESP32/system/xzz_project/build/include/config/auto.conf'. Stop.
```

那就是你工程拷贝其它工程的 Makefile 除了问题，因为 example 里面的工程如果是我已经编译好了的。那么拷贝过来可能有 BUG。所以你最好自己照着案例的工程重新写个 Makefile。或者将案例工程先 make clean 一下(最好在 mingw32 操作)，然后再拷贝 Makefile 到自己工程。然后直接 make 工程

```
181 void app_main()
182 {
183     while(1){
184         MDF_LOGI("*****");
185         MDF_LOGI("          Log output          ");
186         MDF_LOGI("*****");
187         sleep(1);
188     }
189     log_test();
190
191     MDF_LOGI("*****");
192     MDF_LOGI("          info store          ");
193     MDF_LOGI("*****");
194 }
```

PROBLEMS ③ OUTPUT DEBUG CONSOLE TERMINAL

```
Administrator@3339DD8YRY0X09A MINGW32 /d/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon
$ make menuconfig
Toolchain path: /opt/xtensa-esp32-elf/bin/xtensa-esp32-elf-gcc
Toolchain version: crosstool-ng-1.22.0-88-g6c4433a5
Compiler version: 5.2.0
MENUCONFIG

*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.
```

```
Administrator@3339DD8YRY0X09A MINGW32 /d/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon
$ make
Toolchain path: /opt/xtensa-esp32-elf/bin/xtensa-esp32-elf-gcc
Toolchain version: crosstool-ng-1.22.0-88-g6c4433a5
Compiler version: 5.2.0
Python requirements from D:/xzz_ESP32/esp-mdf-v1.0/esp-idf/requirements.txt are satisfied.

App "mcommon" version: v1.0-dirty
To flash all build output, run 'make flash' or:
python /d/xzz_ESP32/esp-mdf-v1.0/esp-idf/components/esptool_py/esptool/esptool.py --chip esp32 --port COM6 --baud 115200 --before default_reset --after detect 0xd000 /d/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/ota_data_initial.bin 0x1000 /d/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/mcommon.bin 0x8000 /d/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon/build/partitions.bin
```

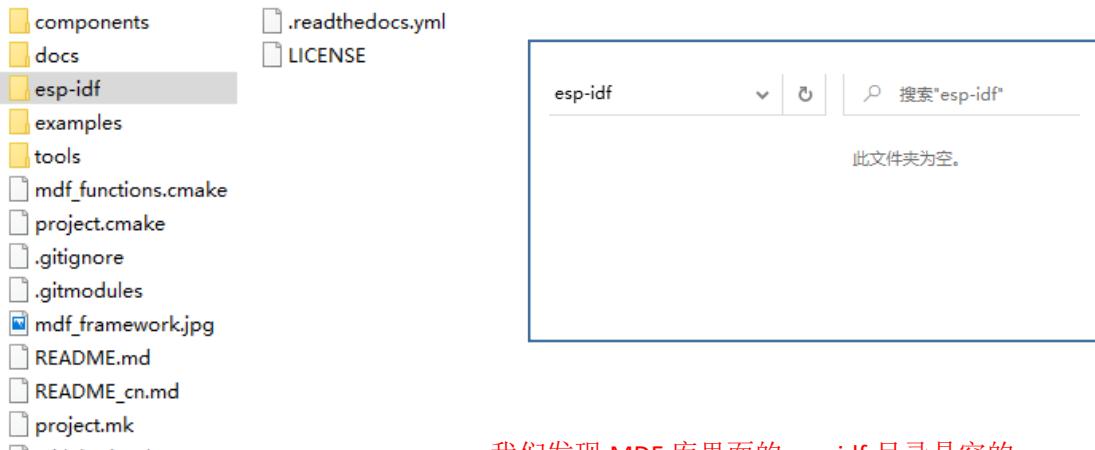
```
Administrator@3339DD8YRY0X09A MINGW32 /d/xzz_ESP32/esp-mdf-v1.0/examples/function_demo/mcommon
$
```

编译成功，文件输出成功。

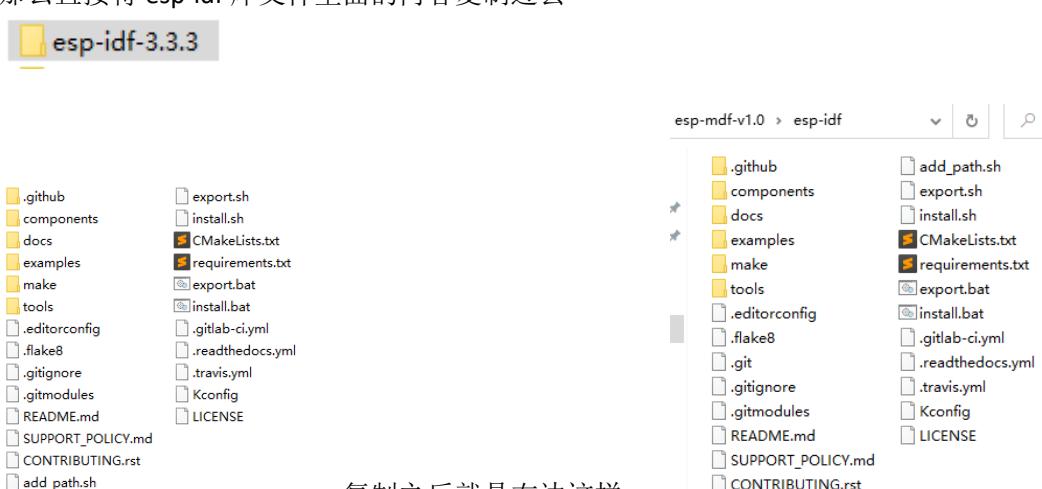
编译过程中官方问题居多的情况，不好解决。如果有现成的交叉编译器压缩包，请跳过此页
不管是 IDF 还是 MDF 库 编译过程中会出现 `esp-idf/make/project.mk: No such file or directory`,

Make: ***没有规则可制作目标.....project.mk 停止

```
$ make
fatal: 不是一个 git 仓库（或者任何父目录）: .git
D:/ESP32system/esp-mdf-1.0/project.mk:10: D:/ESP32system/esp-mdf-1.0/esp-idf/make/project.mk: No such file or directory
make: *** 没有规则可制作目标“D:/ESP32system/esp-mdf-1.0/esp-idf/make/project.mk”。停止。
```



那么直接将 esp-idf 库文件里面的内容复制过去



然后直接 make menuconfig

```
make[1]: *** [/d/ESP32system/esp-mdf-v1.0/esp-idf/make/project.mk:612: /d/ESP32system/esp-mdf-v1.0/esp-idf/components/micro-ecc/.git] 错误 1
make: *** [/d/ESP32system/esp-mdf-v1.0/esp-idf/components/bootloader/Makefile.projbuild:41: /d/ESP32system/esp-mdf-v1.0/examples/function_demo/mcommon/build/bootloader/bootloader.bin] 错误 2
```

没有报找不到 project.mk 文件，但是还是报错，这是因为你的 make menuconfig 没有进行设置，一定要随便设置下 make menuconfig，而不是打开 menuconfig 关闭就可以了。我设置 menuconfig 配置串口号。

make 直接编译，经过了一段时间编译。

```
d:\esp32system\msys32\opt\xtensa-esp32-elf\xtensa-esp32-elf\sys-include\stdlib.h:155:44: error: expected initializer before '_result_use_check'
void *reallocarray(void *, size_t, size_t) __result_use_check __alloc_size(2)
^~~~~~
d:\esp32system\msys32\opt\xtensa-esp32-elf\xtensa-esp32-elf\sys-include\stdlib.h:340:52: error: expected initializer before '_alloc_align'
void * aligned_alloc(size_t, size_t) __malloc_like __alloc_align(1)
^~~~~~
```

又报交叉编译器错误，看来交叉编译器版本也有关系。现在只有先用稳定版本的交叉编译器。

如果有稳定的交叉编译器版本，可以跳过以上内容

有些库，比如 IDF/MDF 在 make menuconfig 的时候，报错 check_python_dependencies

```
system32\WindowsPowerShell\v1.0\;D:\ESP32system\msys32\opt\xtensa-esp32-elf\bin;D:\ESP32s
\ESP32system\msys32\usr\bin\vendor_perl;D:\ESP32system\msys32\usr\bin\core_perl
make: *** No rule to make target 'check_python_dependencies', needed by 'all'. Stop.
```

这时候有两种方法

第 1 种如果你 MDF 库没有问题，你就将 IDF 里面的 examples 例程的 Makefile 改成 MDF 路径。

第 2 种：

pip install --upgrade pip

```
D:\ESP32system\esp-idf-3.3.3>pip install --upgrade pip
You are using pip version 7.0.1, however version 20.2.3 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
Collecting pip
  Downloading https://files.pythonhosted.org/packages/4e/5f/528232275f6509b1fff703c9280e58951a81abe24
/pip-20.2.3-py2.py3-none-any.whl (1.5MB)
   53% |██████████| 798kB 4.6kB/s eta 0:02:32
```

先更新 pip

```
/pip-20.2.3-py2.py3-none-any.whl (1.5MB)
  82% |██████████| 1.2MB 1.8kB/s eta 0:00:00
Traceback (most recent call last):
  File "C:\Python27\lib\site-packages\pip\basecommand.py", line 223, in main
    status = self.run(options, args)
  File "C:\Python27\lib\site-packages\pip\commands\install.py", line 37, in run
    requirement_setuptools_sdist(self.install_dir)
  File "C:\Python27\lib\site-packages\pip\req\req_setuptools.py", line 200, in requirement_setuptools_sdist
    requirement_setuptools_sdist(self.install_dir)
  File "C:\Python27\lib\site-packages\pip\req\req_setuptools.py", line 200, in requirement_setuptools_sdist
    requirement_setuptools_sdist(self.install_dir)
```

如果 pip 更新变红色，表示网速不好，翻墙后重复执行 pip install --upgrade pip
更新完成后

一定要用 mingw32 里面的 python 来下载 requirements.txt 里面的内容

```
python2.7.exe -m pip install --user -r $IDF_PATH/requirements.txt
```

```
XZ666@XZ666PC MINGW32 /D/ESP32system/esp-idf-3.3.3
$ python2.7.exe -m pip install --user -r $IDF_PATH/requirements.txt
Processing d:/esp32system/esp-idf-3.3.3/tools/kconfig_new/esp-windows-curses
Requirement already satisfied: setuptools>=21 in d:/esp32system/msys32/mingw32/lib/python2.7/site-packages (from -r D:/ESP32system/esp-idf-3.3.3/requirements.txt (line 4)) (40.4.3)
Collecting click>=5.0 (from -r D:/ESP32system/esp-idf-3.3.3/requirements.txt (line 8))
  Downloading https://files.pythonhosted.org/packages/d2/3d/fa/6db83bf75c4f8d338c2fd15c8d33fdd7ad23a9b5e57eb6c5c
e26b430e/click-7.1.2-py3-none-any.whl (82kB)
   100% |██████████| 92kB 71kB/s
Requirement already satisfied: pyserial>=3.0 in d:/esp32system/msys32/mingw32/lib/python2.7/site-packages (from -r D:/ESP32system/esp-idf-3.3.3/requirements.txt (line 9)) (3.4)
Requirement already satisfied: future>=0.15.2 in d:/esp32system/msys32/mingw32/lib/python2.7/site-packages (from -r D:/ESP32system/esp-idf-3.3.3/requirements.txt (line 10)) (0.17.1)
下载过程中遇到问题，网速不好，翻墙继续重复下载
```

```
100% |██████████| 747kB 12kB/s eta 0:00:09Exception:
Traceback (most recent call last):
  File "D:/ESP32system/msys32/mingw32/lib/python2.7/site-packages/pip/_internal/basecommand.py", line 141, in main
    status = self.run(options, args)
  File "D:/ESP32system/msys32/mingw32/lib/python2.7/site-packages/pip/_internal/commands/install.py", line 299, in run
    create_bufdir(temp.mingw277c)
    i686-w64-mingw32-gcc -fno-strict-aliasing -march=i686 -mtune=generic -O2 -pipe -fwrapv -D__USE_MINGW_ANSI_STDIO=1 -DNDEBUG -DNDEBUG -I/usr/include/ffi -I/usr/include/libffi -I:D:/ESP32system/msys32/mingw32/include/python2.7 -c c/_cffi_backend.c -o build/temp.mingw-2.7/c/_cffi_backend.o
    error: command 'i686-w64-mingw32-gcc' failed: No such file or directory

    Command "D:/ESP32system/msys32/mingw32/bin/python2.7.exe -u -c "import setuptools, tokenize;__file__='c:/users/xz666/appdata/local/temp/pip-install-sscbfh/ffi/setup.py';f=getattr(tokenize, 'open', open)(__file__);code=f.read().replace('\r\n', '\n');f.close();exec(compile(code, __file__, 'exec'))" install --record c:/users/xz666/appdata/local/temp/pip-record-qqip08/install-record.txt --single-version-externally-managed --prefix c:/users/xz666/appdata/local/temp/pip-build-env-fxyn0b --compile" failed with error code 1 in c:/users/xz666/appdata/local/temp/pip-install-sscbfh/ffi/
```

```
Command "D:/ESP32system/msys32/mingw32/bin/python2.7.exe -m pip install --ignore-installed --no-user --prefix c:/users/xz666/appdata/local/temp/pip-build-env-fxyn0b --no-warn-script-location --no-binary :none: --only-binary :none: -i https://pypi.org/simple -- "setuputils >= 40.8.0" wheel "Cython >= 0.29.14" "cffi >= 1.12.3 ; platform_python_implementation == 'CPython'" "greenlet>=0.4.14 ; platform_python_implementation == 'CPython'" failed with error code 1 in None
```

最好下载到这里出问题了，还没有解决这个问题，后续再解决。

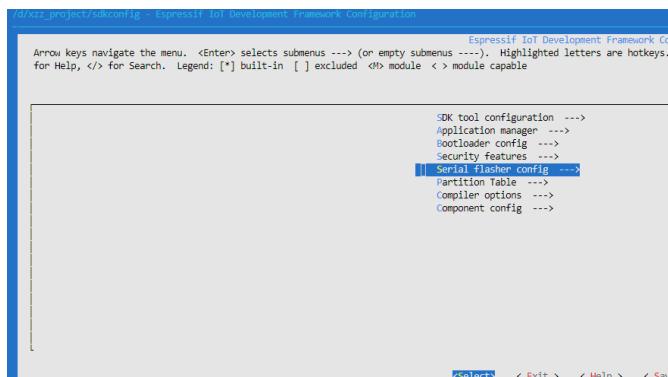
ESP32 MDF 创建第 1 个工程

名称	修改日期	类型	大小
build	2020-09-15 18:08	文件夹	
component.mk	2020-05-26 16:13	MK 文件	1 KB
main.c	2020-09-15 18:08	sourceinsight.c_f...	1 KB
Makefile	2020-09-15 18:04	文件	1 KB
sdkconfig	2020-09-15 10:36	文件	20 KB

直接 make

```
Administrator@3339DD8YRY0XD9A MINGW32 /d/xzz_project
$ make
DEFCONFIG
#
# configuration written to /d/xzz_project/sdkconfig
#
MENUCONFIG
Your display is too small to run Menuconfig!
It must be at least 19 lines by 80 columns.
make: *** No rule to make target 'menuconfig', needed by '/d/xzz_project/sdkconfig'. Stop.
```

需要你先执行 make menuconfig 这样会在根目录下生成 sdkconfig 文件



我暂时不知道要设置哪些

我直接复制官方的 sdkconfig 算了

名称	修改日期	类型	大小
build	2020-09-15 14:55	文件夹	
component.mk	2020-05-26 16:13	MK 文件	1 KB
main.c	2020-09-15 14:51	sourceinsight.c_f...	1 KB
Makefile	2020-09-15 14:52	文件	1 KB
sdkconfig	2020-09-15 14:53	文件	20 KB

这就已经生成了 sdkconfig, 我

还是复制官方的算了。

```
D:/xzz_project/build/esp32/libesp32.a(cpu_start.o):(.literal.main_task+0x18): undefined reference to `app_main'
D:/xzz_project/build/esp32/libesp32.a(cpu_start.o): In function `main_task':
D:/xzz_ESP32/esp-mdf-v1.0/esp-idf/components/esp32/cpu_start.c:537: undefined reference to `app_main'
collect2.exe: error: ld returned 1 exit status
make: *** [D:/xzz_ESP32/esp-mdf-v1.0/esp-idf/make/project.mk:483: /d/xzz_project/build/xzzmain.elf] Error 1
```

这个错误是因为你的 main.c 文件没有进入 main 目录, main 是个很特殊的目录, 必须创建 main 目录, 在 main 目录下建立你自己的 C 文件

在项目根目录下创建 main 目录

名称	修改日期	类型	大小
build	2020-09-15 18:08	文件夹	
main	2020-09-15 18:03	文件夹	
Makefile	2020-09-15 18:04	文件	1 KB
sdkconfig	2020-09-15 10:36	文件	20 KB

这就是整个目录结构

```
D:/xzz_project/main/main.c: In function 'app_main':
D:/xzz_ESP32/esp-mdf-v1.0/components/mdf_common/include/mdf_err.h:91:41: error: 'TAG' undeclared (first use in this function)
    esp_log_write(ESP_LOG_INFO, TAG, MDF_LOG_FORMAT(I, format), esp_log_timestamp(), TAG, __LINE__, ##_VA_ARGS_);
                                         ^
D:/xzz_project/main/main.c:9:9: note: in expansion of macro 'MDF_LOGI'
    MDF_LOGI("*****");
           ^
```

Error: 'TAG' undeclared 这是因为 IDF 和 MDF 库里面打印用的 MDF_LOGI 函数，函数内置 TAG 变量，所以需要给变量赋值。

```
#include "mdf_common.h"
#include "sys/unistd.h"

static const char *TAG = "xxxxxxxxxx";

void app_main()
{
    while(1){
        MDF_LOGI("*****");
        MDF_LOGI("      xxxxxxxxx Log output      ");
        MDF_LOGI("*****");
        sleep(1);
    }
}
```

整个 TAG 全局变量一定要定义，因为在 MDF_LOGI 里面是定义了整个 TAG 变量的。整个 TAG 的参数会在串口打印的时候显示。

```
to Flash all build output, run "make flash" or:
python /d/xzz_ESP32/esp-mdf-v1.0/esp-idf/components/esptool_py/esptool/esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 115200 --before default_reset --after hard_reset write_flash -z --flash_mode dio --flash_freq 40m --flash_size detect 0x1000 /d/xzz_project/build/bootloader/bootloader.bin 0x10000 /d/xzz_project/build/xzzpri.bin 0x8000 /d/xzz_project/build/partitions_singleapp.bin
```

0x1000 /d/xzz_project/build/bootloader/bootloader.bin

0x10000 /d/xzz_project/build/xzzpri.bin

0x8000 /d/xzz_project/build/partitions_singleapp.bin

其实这三个地址值是根据分区表来的，《[ESP32_peripheral_UserGuide](#)》文档会讲。本文其它章节也有介绍。

```
I (76050) [xxxxxxxx, 11]: ****
I (77060) [xxxxxxxx, 9]: ****
I (77060) [xxxxxxxx, 10]:      xxxxxxxxx Log output
I (77060) [xxxxxxxx, 11]: ****
I (78070) [xxxxxxxx, 9]: ****
I (78070) [xxxxxxxx, 10]:      xxxxxxxxx Log output
I (78070) [xxxxxxxx, 11]: ****
I (79080) [xxxxxxxx, 9]: ****
I (79080) [xxxxxxxx, 10]:      xxxxxxxxx Log output
I (79080) [xxxxxxxx, 11]: ****
I (80090) [xxxxxxxx, 9]: ****
I (80090) [xxxxxxxx, 10]:      xxxxxxxxx Log output
```

输出打印成功

ESP32 错误处理宏，在调用 ESP32 API 函数的过程中大部分都使用了错误宏包含

ESP_ERROR_CHECK 宏

esp_err_t API 执行错误类型判断
大多数 ESP32 API 采用 esp_err_t 变量来返回 API 执行的错误/正确结果
esp_err_t 是带符号类型，返回 ESP_OK 表示成功，ESP_OK = 0
esp_err_t 变量；根据变量值可以判定程序发生了什么错误，将其打印出来。

案例 1：

```
esp_err_t err;  
do {  
    err = sdio_slave_send_queue(addr, len, arg, timeout);  
    // 如果发送队列已满就不断重试  
}while (err == ESP_ERR_TIMEOUT); //可以根据 err 等于的 ESP_ERR_TIMEOUT 值来判断函数错  
误类型  
if (err != ESP_OK)  
{  
    // 处理其他错误  
}
```

对应中间组件，通常并不希望发生错误时就终止程序，有可能错误并不致命
在这种情况下使用 ESP_ERROR_CHECK 比较合适，让代码也看起更整洁

案例 2：

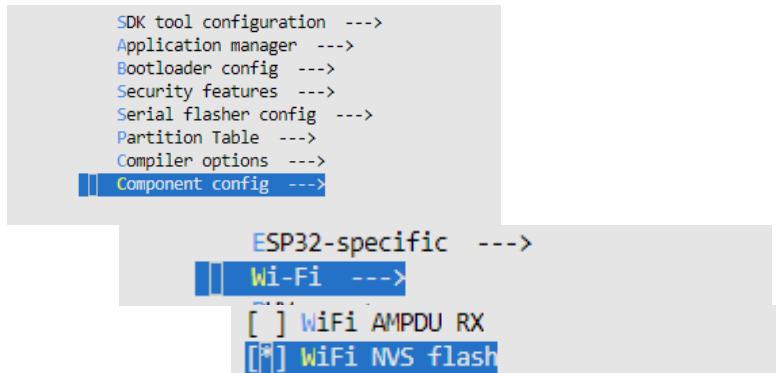
```
ESP_ERROR_CHECK(spi_bus_initialize(host, bus_config, dma_chan));  
如果执行 spi_bus_initialize(...)发生错误，ESP_ERROR_CHECK 打印如下：
```

```
ESP_ERROR_CHECK failed: esp_err_t 0x107 (ESP_ERR_TIMEOUT) at 0x400d1fdf //直接结果  
//还要打印错误位置  
file: "/Users/user/esp/example/main/main.c" line 20func: app_mainexpression:  
sdmmc_card_init(host, &card)
```

ESP32 NVS 存储器，类似 MCU EEPROM

ESP32 NVS 存放的数据其实是在内部 flash 中的，可以存储 softAP 的参数配置信息。

```
#include "nvs.h" 包含该头文件
```



打开 WiFi NVS flash 才能使用 NVS 存储功能

NVS 存储区的大小是根据分区表来的，《ESP32_peripheral_UserGuide》文档会讲

nvs_flash_init(); //初始化 NVS

nvs_flash_erase() //擦除 NVS 存储器

调用 “nvs_flash_init();”，如果失败可调用 “nvs_flash_erase()” 擦除 NVS，然后再次初始化。

```
esp_err_t nvs_open(const char* name, nvs_open_mode open_mode, nvs_handle
```

*out_handle); //打开 NVS 存储表

name: 填入字符串，定义表名称

open_mode: 打开该表后，读写模式

NVS_READWRITE 表示可读可写；

NVS_READONLY 表示只读

*out_handle 传入 nvs_handle 的句柄变量，后面的 nvs_set_str, nvs_set_i32... 等函数都是操作这个句柄来操作该 NVS 表

例如： nvs_handle my_handle;

```
nvs_open( "List", NVS_READWRITE, &my_handle );
```

```
esp_err_t nvs_set_str (nvs_handle handle, const char* key, const char* value); //写字符串存储
```

handle: 传入上面定义的 nvs_handle my_handle; 句柄

Key: 传入 K 值

value: 写入 K 值对应的 value 值，字符串

例如： static const char *DATA1 = "param 1" //自定义键值，这个要记住哦

```
ESP_ERROR_CHECK( nvs_set_str( handle, DATA1, "i am a string." ));
```

//向键值 param 1 写入对应的字符串 "i am a string." 存储。以后掉电，再启动，记得键值是多少，才能取对应的数据

```
esp_err_t nvs_set_i32 (nvs_handle handle, const char* key, int32_t value); //写 32 位有符号整数
```

handle: 传入上面定义的 nvs_handle my_handle; 句柄

Key: 传入 K 值

value: 写入 K 值对应的 value 值，32 位有符号整数

```
例如: int32_t value_store = 666; //定义有符号整型  
static const char *DATA2 = "param 2" //自定义键值, param2 键值对应有符号整型  
ESP_ERROR_CHECK( nvs_set_i32( handle, DATA2, value_store ) )
```

```
esp_err_t nvs_set_blob(nvs_handle handle, const char* key, const void* value, size_t length);  
//写无固定长度(如结构体)存储  
handle: 传入上面定义的 nvs_handle my_handle; 句柄  
*Key: 传入 K 值  
void* value: 结构体地址  
length: 用 sizeof 计算结构体长度
```

例如: static const char *DATA3 = "param 3"; //定义存放结构体键值

```
wifi_config_t wifi_config_to_store = {  
.sta = {  
.ssid = "store_ssid:hello_kitty",  
.password = "store_password:1234567890",  
},  
};
```

```
ESP_ERROR_CHECK( nvs_set_blob( handle, DATA3, &wifi_config_to_store , sizeof(wifi_config_to_store)) );
```

```
esp_err_t nvs_commit(nvs_handle handle); //保存句柄的数据进 flash  
handle: 传入上面定义的 nvs_handle my_handle; 句柄  
void nvs_close(nvs_handle handle); //关闭打开的句柄  
handle: 传入上面定义的 nvs_handle my_handle; 句柄
```

```
#include "mdf_common.h"  
#include "sys/unistd.h"  
  
#include "nvs.h"  
  
static const char *TAG = "xxxxxxxx";  
  
struct StructXZZ  
{  
    char *str;  
    int32_t num;  
};
```

```

void app_main()
{
    nvs_handle handle; //定义句柄

    esp_err_t err = nvs_flash_init(); //初始化nvs_flash
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );

    static const char *DATA1 = "param 1"; //定义键值
    static const char *DATA2 = "param 2"; //定义键值
    static const char *DATA3 = "param 3"; //定义键值

    int32_t value_store = 666;
    struct StructXZZ struxzz;
    struxzz.str = "ABCDEFG";
    struxzz.num = 100;

    ESP_ERROR_CHECK(nvs_open( "XZZ list", NVS_READWRITE, &handle ) );
    ESP_ERROR_CHECK(nvs_set_str( handle, DATA1, "i am a xzz." ) );
    ESP_ERROR_CHECK(nvs_set_i32( handle, DATA2, value_store ) );
    ESP_ERROR_CHECK(nvs_set_blob( handle, DATA3, &struxzz, sizeof(struxzz)) );

    ESP_ERROR_CHECK( nvs_commit(handle) );
    nvs_close(handle);

    while(1){
        MDF_LOGI("*****xzz nvs_flash write*****");
        sleep(1);
    }
}

I (0) cpu_start: Starting scheduler on APP CPU.
I (383) [xxxxzzzz, 51]: *****xzz nvs_flash write*****
I (1383) [xxxxzzzz, 51]: *****xzz nvs_flash write*****
I (2383) [xxxxzzzz, 51]: *****xzz nvs_flash write*****
I (3383) [xxxxzzzz, 51]: *****xzz nvs_flash write*****

```

读取 NVS 存储器里面的值

```

esp_err_t nvs_get_str (nvs_handle handle, const char* key, char* out_value, size_t* length);
//读出 NVS flash 字符串

```

handle: 传入 nvs_handle my_handle; 定义的 句柄

*Key: 传入 K 值 你上次写入的 K 值, 这次去找到这个 K 值, 把里面数据读出来

* out_value 接收字符串数组地址

* length 字符串长度, 但是这里用变量地址来传

```

esp_err_t nvs_get_i32 (nvs_handle handle, const char* key, int32_t* out_value); //读取 NVS 整形

```

handle: 传入 nvs_handle my_handle; 定义的 句柄

*Key: 传入 K 值 你上次写入的 K 值, 这次去找到这个 K 值, 把里面数据读出来

* out_value 接收变量地址

```

esp_err_t nvs_get_blob(nvs_handle handle, const char* key, void* out_value, size_t* length);
//读出 NVS flash 结构体参数
handle: 传入 nvs_handle my_handle; 定义的 句柄
*Key: 传入 K 值 你上次写入的 K 值, 这次去找到这个 K 值, 把里面数据读出来
* out_value 传入结构体地址, 或者其它参数地址, 这里是 Void* 任意参数处理
* length 传入结构体长度变量地址

```

```

void app_main()
{
    nvs_handle handle; //定义句柄

    esp_err_t err = nvs_flash_init(); //初始化nvs_flash
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {

        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );

    static const char *DATA1 = "param 1"; //定义键值
    static const char *DATA2 = "param 2"; //定义键值
    static const char *DATA3 = "param 3"; //定义键值

    char read_str[32] = {0};
    uint32_t str_length = 32;//预设接收字符串长度

    int32_t read_value = 0;//读取NVS flash里面得值

    struct StructXZZ Readstruzz = {0}; //读取NVS flash里面得值
    uint32_t structlen = sizeof(Readstruzz); //预设接收结构体长度

    ESP_ERROR_CHECK(nvs_open( "XZZ list", NVS_READWRITE, &handle ) );//读写XZZ list表
    ESP_ERROR_CHECK(nvs_get_str( handle, DATA1, read_str,&str_length)); //读出XZZ list列表里param1 K值得字符串
    ESP_ERROR_CHECK(nvs_get_i32(handle, DATA2, &read_value) );
    ESP_ERROR_CHECK(nvs_get_blob(handle, DATA3, &Readstruzz, &structlen) );

    nvs_close(handle); //关闭句柄

    while(1){

        MDF_LOGI("*****xzz nvs_flash read str %s *****",read_str);
        MDF_LOGI("*****xzz nvs_flash read value %d *****",read_value);
        MDF_LOGI("*****xzz nvs_flash read struct %s %d *****",Readstruzz.str,Readstruzz.num);

        sleep(1);
    }
}

```

app_main 里面的 while(1)
一定要插入 sleep, 不然主线程死循环时间片无法释放, CPU 会出现 IDLE 情况

```

I (56942) [xxxxzzzz, 54]: *****xzz nvs_flash read str i am a xzz. *****
I (56942) [xxxxzzzz, 55]: *****xzz nvs_flash read value 666 *****
I (56942) [xxxxzzzz, 56]: *****xzz nvs_flash read struct .@?d 0 *****

```

结构体里面字符串有乱码, 这个后面解决

CPU 0:main _ CPU IDLE1 运行问题

```

E (25203) task_wdt: Tasks currently running:
E (25203) task_wdt: CPU 0: main
E (25203) task_wdt: CPU 1: IDLE1
E (30203) task_wdt: Task watchdog got triggered. The following tasks did not reset the watchdog in time:
E (30203) task_wdt: - IDLE0 (CPU 0)
E (30203) task_wdt: Tasks currently running:
E (30203) task_wdt: CPU 0: main
E (30203) task_wdt: CPU 1: IDLE1

```

ESP32 WIFI 连接热点

```
#include "esp_wifi.h"    包含 wifi 头文件  
#include "nvs_flash.h"  这是 nvsflash 头文件
```

```
tcpip_adapter_init(); //使用 wifi 前一定要先初始化 tcpip 适配器
```

```
esp_err_t esp_event_loop_init(system_event_cb_t cb, void *ctx)  
//wifi 进入 STA 模式后需要一个状态机不断的维护当前 wifi 状态,  
向 esp_event_loop_init 注册一个回调函数来处理当前维护的 wifi 状态  
再说明白点 ESP32 WIFI 启动的时候会产生很多事件回调, 这些事件的都绑定在一个回调函数上, 在函数内通过判断传递的参数  
来判断事件类型, cb 就是这个回调函数  
cb: 传入自定义的回调函数  
ctx: 这是给回调函数传外部参数, 没有参数填 NULL
```

回调函数定义原型

```
typedef esp_err_t (*system_event_cb_t)(void *ctx, system_event_t *event);  
*ctx: 外部传入进来的参数  
*event: 回调函数被调用时, 使用 event 来查阅发生了什么事件
```

```
wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT(); //wifi 启动前必须配置  
初始化  
esp_wifi_init(&cfg) //初始化 wifi
```

wifi_config_t 用于配置 wifi 的 SSID 和密码的结构体

```
esp_err_t esp_wifi_set_mode(wifi_mode_t mode) //设置 wifi 工作模式
```

```
mode: WIFI_MODE_STA      这是 STA 模式  
      WIFI_MODE_AP       这是 AP 模式  
      WIFI_MODE_APSTA   这是 AP 加 STA 同时工作模式
```

```
esp_err_t esp_wifi_set_config(wifi_interface_t ifx, wifi_config_t  
*conf); //设置 wifi 在一种(STA/AP)模式下, 需要的参数  
ifx: WIFI_MODE_STA STA 模式, WIFI_MODE_AP AP 模式, WIFI_MODE_APSTA APSTA 双模式  
*conf: wifi_sta_config_t sta /**< 如果是 STA 的配置 */
```

```
typedef struct {  
    uint8_t ssid[32]; /**< SSID of target AP*/  
    uint8_t password[64]; /**< password of target  
AP*/  
    bool bssid_set; //是否需要检查连接 AP 热点的 MAC  
    地址, 需要检查填 1, 一般不检查填 0  
    uint8_t bssid[6]; /**< MAC address of target AP*  
    uint8_t channel; //目标 AP 通道, 从指定通道开始  
    扫描, 1~13 通道扫描, 如果 AP 通道未知, 填 0 就是了  
} wifi_sta_config_t;
```

```
wifi_ap_config_t ap //这个 AP 模式的时候讲解  
esp_err_t esp_wifi_start(void) //启动 wifi
```

Wifi 初始化和启动之前要先设置好事件回调状态机

```
Static esp_err_t event_handler(void *ctx, system_event_t *event)
{
    switch (event->event_id) {
        case SYSTEM_EVENT_STA_START:
            esp_wifi_connect(); //连接热点
            break;
        case SYSTEM_EVENT_STA_GOT_IP:
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            /* This is a workaround as ESP32 WiFi libs don't currently
               auto-reassociate. */
            esp_wifi_connect(); //连接热点
            break;
        default:
            break;
    }
    return ESP_OK;
}
```

这些状态参数后面会讲解

```
void app_main()
{
    esp_err_t err = nvs_flash_init(); //记住在做 wifi 功能时，一定要对 nvs_flash 初始化，不然板子会一直重启
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        // NVS partition was truncated and needs to be erased
        // Retry nvs_flash_init
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );
    tcpip_adapter_init(); //初始化 TCP/IP 适配层 不然启动 wifi 模式会导致模块不停重启
    ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT(); //这个函数是常规操作，用来给 wifi 初始化赋值
    ESP_ERROR_CHECK( esp_wifi_init(&cfg) ); //wifi 使用之前，必须执行这段初始化
    wifi_config_t wifi_config = {
        .sta = {
            .ssid = "OPPO A11", //要连接设备 SSID
            .password = "12345678", //要连接设备密码
        },
    };
    ESP_LOGI(TAG, "Setting WiFi configuration SSID %s...", wifi_config.sta.ssid);
    ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA) ); //wifi 启动为 STA 模式
    ESP_ERROR_CHECK( esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config) );
    //给 wifi 接口设置参数，STA 和 AP 是不一样的参数，这里 set_config 设置的是 STA 模式下传入 SSID 和密码
    ESP_ERROR_CHECK( esp_wifi_start() ); //启动 wifi
    while(1){
        sleep(1);
    }
}
```

手机或者路由设置好 AP 热点，连接成功

```
I (266) xxxzzzzz: Setting WiFi configuration SSID OPPO A1...
I (366) phy: phy_version: 4180, cb3948e, Sep 12 2019, 16:39:13, 0, 0
I (366) wifi:mode : sta (fc:f5:c4:3c:4a:1c)
I (74356) wifi:new:<4,0>, old:<1,0>, ap:<255,255>, sta:<4,0>, prof:13
I (74366) wifi:state: init -> auth (b0)
I (74366) wifi:state: auth -> assoc (0)
I (74376) wifi:state: assoc -> run (10)
I (74386) wifi:connected with OPPO A11, aid = 1, channel 4, BW20, bssid = 18:d0:c5:ca:92:01 //打印连接成功的信息
I (74386) wifi:security type: 3, phy: bgn, rssi: -17
I (74396) wifi:pm start, type: 1
I (74446) wifi:AP's beacon interval = 204800 us, DTIM period = 2
I (78196) event: sta ip: 192.168.43.238, mask: 255.255.255.0, gw: 192.168.43.1
```

下面讲下事件回调函数过程 `event_handler(void *ctx, system_event_t *event)`

```

event: SYSTEM_EVENT_STA_START //第一次进入
event: SYSTEM_EVENT_STA_CONNECTED //连接成功

event: SYSTEM_EVENT_STA_GOTIP // 获取到 IP 后, 重启 WiFi 热点

event: SYSTEM_EVENT_STA_DISCONNECTED //连接被断开

```



```

static esp_err_t event_handler(void *ctx, system_event_t *event)
{
    switch (event->event_id) [
        case SYSTEM_EVENT_STA_START:
            esp_wifi_connect(); //连接热点
            break;
        case SYSTEM_EVENT_STA_CONNECTED:
            ESP_LOGI(TAG, "connect AP success");
            break;
        case SYSTEM_EVENT_STA_GOT_IP:

            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            /* This is a workaround as ESP32 WiFi libs don't currently
             * auto-reassociate. */
            ESP_LOGI(TAG, "connect AP disconnect");
            break;
        default:
            break;
    ]
    return ESP_OK;
}

```

加入连接成功状态

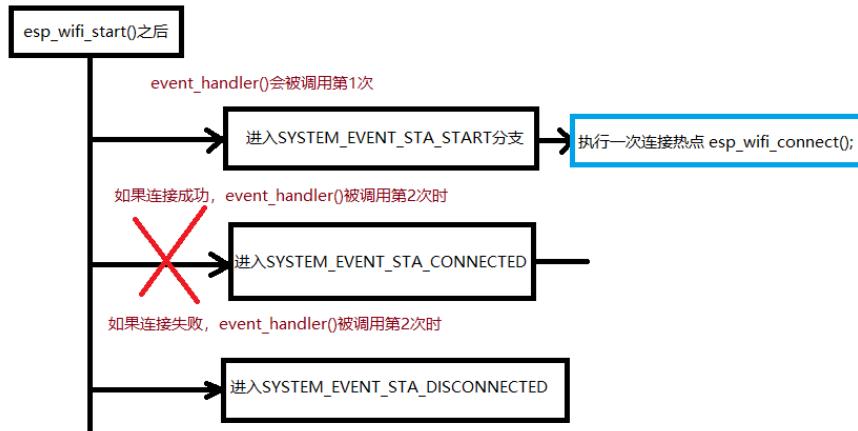
```

I (266) xxxxxxxx: Setting WiFi configuration SSID OPPO All...
I (356) phy: phy_version: 4180, cb3948e, Sep 12 2019, 16:39:13, 0, 0
I (366) wifi:mode : sta (fc:f5:c4:3c:4a:1c)
I (606) wifi:new:<1,0>, old:<1,0>, ap:<255,255>, sta:<1,0>, prof:13
I (606) wifi:state: init -> auth (b0)
I (616) wifi:state: auth -> assoc (0)
I (616) wifi:state: assoc -> run (10)
I (646) wifi:connected with OPPO All, aid = 1, channel 1, BW20, bssid = 18:d0:c5:ca:92:01
I (646) wifi:security type: 3, phy: bgn, rssi: -28
I (656) wifi:pm start, type: 1

I (656) xxxxxxxx: connect AP success
I (716) WiFi:AP's beacon interval = 204800 us, DTIM period = 2
I (1686) event: sta ip: 192.168.43.238, mask: 255.255.255.0, gw: 192.168.43.1

```

连接成功打印



```

static esp_err_t event_handler(void *ctx, system_event_t *event)
{
    switch (event->event_id) {
    case SYSTEM_EVENT_STA_START:
        esp_wifi_connect(); //连接热点
        break;
    case SYSTEM_EVENT_STA_CONNECTED:
        ESP_LOGI(TAG, "connect AP success");
        break;
    case SYSTEM_EVENT_STA_GOT_IP:
        break;
    case SYSTEM_EVENT_STA_DISCONNECTED:
        /* This is a workaround as ESP32 WiFi libs don't currently
         * auto-reassociate. */
        ESP_LOGI(TAG, "connect AP disconnect");
        break;
    default:
        break;
    }
    return ESP_OK;
}

```

比如我把热点关了

```

I (463976) wifi:new:<1,0>, old:<1,0>, ap:<255,255>, sta:<1,0>, prof:13
I (463976) xxxxxxxx: connect AP disconnect

```

但是发现，并没有去重连，理论上应该是要进行重连的，有时候热点不关，就是信号不好的情况也会出现断开现象，所以需要重连。

STA 模式重连机制



```

static esp_err_t event_handler(void *ctx, system_event_t *event)
{
    switch (event->event_id) {
        case SYSTEM_EVENT_STA_START:
            esp_wifi_connect(); //连接热点
            break;
        case SYSTEM_EVENT_STA_CONNECTED:
            ESP_LOGI(TAG, "connect AP success");
            break;
        case SYSTEM_EVENT_STA_GOT_IP:
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            /* This is a workaround as ESP32 WiFi libs don't currently
               auto-reassociate. */
            ESP_LOGI(TAG, "connect AP disconnect");
            esp_wifi_connect(); //连接热点
            break;
        default:
            break;
    }
}
  
```

这次没连接成功, 继续发送连接命令

I (286) xxxxxxxx: Setting WiFi configuration SSID OPPO All...
I (386) phy: phy_version: 4180, cb3948e, Sep 12 2019, 16:39:13, 0, 0
I (386) wifi:mode : sta (fc:f5:c4:3c:4a:1c)
I (2436) xxxxxxxx: connect AP disconnect
I (4496) xxxxxxxx: connect AP disconnect
I (6546) xxxxxxxx: connect AP disconnect
I (8596) xxxxxxxx: connect AP disconnect
I (10646) xxxxxxxx: connect AP disconnect
I (12696) xxxxxxxx: connect AP disconnect

状态机无限循环等待连接成功为止

ESP32 STA 模式下如何获取路由分配给自己 IP 地址

```
system_event_t . event_info.got_ip.ip_info.ip  
//得到 IP 地址
```

```
ip4addr_ntoa(....) //将 IP 地址转换为能识别的 ASCII 字符
```

```
static esp_err_t event_handler(void *ctx, system_event_t *event)  
{  
    switch (event->event_id) {  
        case SYSTEM_EVENT_STA_START:  
            esp_wifi_connect(); //连接热点  
            break;  
        case SYSTEM_EVENT_STA_CONNECTED:  
            ESP_LOGI(TAG, "connect AP success");  
            break;  
        case SYSTEM_EVENT_STA_GOT_IP:  
            ESP_LOGI(TAG, "xzz got ip:%s", ip4addr_ntoa(&event->event_info.got_ip.ip_info.ip));  
            //打印分配到的IP到调试窗口,转换IP地址为ASCII字符串
```

在事件循环中
加入 IP 地址获
取就是了

```
I (1796) xxxzzzzz: connect AP success  
I (1826) wifi:AP's beacon interval = 204800 us, DTIM period = 2  
T (2696) event: sta ip: 192.168.43.238, mask: 255.255.255.0, gw: 192.168.43.1  
I (2696) xxxzzzzz: xzz got ip:192.168.43.238
```

这就是分配给设备的 IP 地址

ESP32 查询连接上路由器后，路由器的 ssid，路由器与 ESP32 之间信号强度，路由 BSSID

```
esp_err_t esp_wifi_sta_get_ap_info(wifi_ap_record_t *ap_info)  
//查询连接上路由器的信息  
esp_err_t ESP_OK 表示连接成功  
ESP_ERR_WIFI_CONN, ESP_ERR_WIFI_NOT_CONNECT 都表示热点不存在
```

*ap_info : 返回连接 AP 后，AP 的信息，信息有如下：

wifi_ap_record_t uint8_t bssid[6]; 连接上路由后，路由的 BSSID 可以用
来判断设备连接的哪一个路由
uint8_t ssid[33]; 连续上路由后，查询路由名(SSID)
uint8_t primary; 连接 AP 的信道(通道)
int8_t rssi; 设备与 AP 之间的信号强度

Wifi 启动代码执行之后，只需要在循环里面执行查询函数 `esp_wifi_sta_get_ap_info`，提取 `ap_info` 里面的信息就是了。

```

while(1){

    err = esp_wifi_sta_get_ap_info(&ap_info);
    if(err == ESP_OK)
    {
        ESP_LOGI(TAG, "MAC = %x :%x :%x :%x :%x",ap_info.bssid[0],
        ap_info.bssid[1],ap_info.bssid[2],ap_info.bssid[3],ap_info.bssid[4],ap_info.bssid[5]);
        //查询路由BSSID

        ESP_LOGI(TAG, "ssid = %s",ap_info.ssid); //查询路由SSID
        ESP_LOGI(TAG, "rss = %d",ap_info.rssi); //查询设备和路由之间信号强度
        ESP_LOGI(TAG, "channel = %d",ap_info.primary); //查询设备和路由的通信信道
        ESP_LOGI(TAG, "phy_11b = %d",ap_info.phy_11b); //查询设备是否开启B模G模N模
        ESP_LOGI(TAG, "phy_11g = %d",ap_info.phy_11g);
        ESP_LOGI(TAG, "phy_11n = %d",ap_info.phy_11n);

    }
    else
    {
        ESP_LOGI(TAG, "get ap info failed");
        ESP_LOGI(TAG, "MAC = %x :%x :%x :%x :%x",ap_info.bssid[0],
        ap_info.bssid[1],ap_info.bssid[2],ap_info.bssid[3],ap_info.bssid[4],ap_info.bssid[5]);
    }

    sleep(1);
}

I (301456) xxxxxxxx: ssid = OPPO A11
I (301456) xxxxxxxx: rssi = -41
I (301456) xxxxxxxx: channel = 4
I (301456) xxxxxxxx: phy_11b = 1
I (301466) xxxxxxxx: phy_11g = 1
I (301466) xxxxxxxx: phy_11n = 1
I (302466) xxxxxxxx: MAC = 18 :d0 :c5 :ca :92 :1
I (302466) xxxxxxxx: ssid = OPPO A11
I (302466) xxxxxxxx: rssi = -39
I (302466) xxxxxxxx: channel = 4
I (302466) xxxxxxxx: phy_11b = 1
I (302476) xxxxxxxx: phy_11g = 1
I (302476) xxxxxxxx: phy_11n = 1
I (303476) xxxxxxxx: MAC = 18 :d0 :c5 :ca :92 :1
I (303476) xxxxxxxx: ssid = OPPO A11
I (303476) xxxxxxxx: rssi = -42
I (303476) xxxxxxxx: channel = 4
I (303476) xxxxxxxx: phy_11b = 1
I (303486) xxxxxxxx: phy_11g = 1
I (303486) xxxxxxxx: phy_11n = 1
I (304486) xxxxxxxx: MAC = 18 :d0 :c5 :ca :92 :1
I (304486) xxxxxxxx: ssid = OPPO A11
I (304486) xxxxxxxx: rssi = -42
I (304486) xxxxxxxx: channel = 4
I (304486) xxxxxxxx: phy_11b = 1
I (304496) xxxxxxxx: phy_11g = 1
I (304496) xxxxxxxx: phy_11n = 1
I (305496) xxxxxxxx: MAC = 18 :d0 :c5 :ca :92 :1
I (305496) xxxxxxxx: ssid = OPPO A11
I (305496) xxxxxxxx: rssi = -40
I (305496) xxxxxxxx: channel = 4
I (305496) xxxxxxxx: phy_11b = 1
I (305506) xxxxxxxx: phy_11g = 1
I (305506) xxxxxxxx: phy_11n = 1

I (428696) xxxxxxxx: get ap info failed
I (428696) xxxxxxxx: MAC = 0 :0 :0 :0 :0
I (429436) xxxxxxxx: connect AP disconnect

```

这就是获取的路由信息，`rssi` 在不停的变化，`rssi` 接近于 0 表示信号最强

如果路由断开就会执行 `else` 语句，从 BSSID 也能看出来，因为 BSSID 全部等于 0 我用 MAC 打印的 BSSID

使用 WIFI 功能 STA 模式下扫描附近的网络

实现设备周边 wifi 热点的数量扫描

```
esp_err_t esp_wifi_scan_start(const wifi_scan_config_t *config, bool block)
//每次扫描都需要执行一次 scan_start 函数
*config 可以填入 SSID, 默认也可以填入 NULL
```

```
esp_err_t esp_wifi_scan_get_ap_num(uint16_t *number) //获取周边 wifi 设备数量
*number 传入整型地址, 获取扫描后设备数量, 在 esp_wifi_scan_start 之后执行
```

```
static esp_err_t event_handler(void *ctx, system_event_t *event)
{
    switch (event->event_id) {
        case SYSTEM_EVENT_STA_START:
            // esp_wifi_connect(); //连接热点
            break;
        case SYSTEM_EVENT_STA_CONNECTED:
            ESP_LOGI(TAG, "connect AP success");
            break;
        case SYSTEM_EVENT_STA_GOT_IP:
            ESP_LOGI(TAG, "xzz got ip:%s", ip4addr_ntoa(&event->event_info.got_ip.ip_info.ip));
            // 打印分配到的IP到调试窗口, 转换IP地址为ASCII字符串
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            /* This is a workaround as ESP32 WiFi libs don't currently
             * auto-reassociate. */
            ESP_LOGI(TAG, "connect AP disconnect");
            // esp_wifi_connect(); //连接热点
            break;
        case SYSTEM_EVENT_SCAN_DONE: //加入周边热点扫描事件
            ESP_LOGI(TAG, "SCAN MODE");
            break;
        default:
            break;
    }
    return ESP_OK;
}

void app_main()
{
    esp_err_t err = nvs_flash_init(); //记住在做wifi功能时,一定要对nvs_flash初始化,不然板子会一直重启
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        // NVS partition was truncated and needs to be erased
        // Retry nvs_flash_init
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );

    tcpip_adapter_init(); //初始化TCP/IP适配层 不然启动wifi模式会导致模块不停重启
    ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT(); //这个函数是常规操作,用来给wifi初始化赋值
    ESP_ERROR_CHECK( esp_wifi_init(&cfg) ); //wifi使用之前,必须执行这段初始化

    //不需要设置SSID和密码

    ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA) ); //wifi启动为STA模式
    //在SCAN模式下取消 esp_wifi_set_config
    ESP_ERROR_CHECK( esp_wifi_start() ); //启动wifi
}
```

现在是 wifi 热点扫描模式, 那么一定要取消自动重连函数, 不然设备会不停重启

尤其是这里, 因为一直在断线重连, 所以一定要取消重连

加入扫描事件触发分支

wifi 初始化过程有少许修改

```

while(1){

    ESP_ERROR_CHECK(esp_wifi_scan_start(NULL, 1)); //启动wifi扫描功能, 这是阻塞模式, 执行比较慢

    uint16_t apCount = 0; //查询热点数量
    esp_wifi_scan_get_ap_num(&apCount); //获取周边设备数量
    ESP_LOGI(TAG, "AP number = %d", apCount);

    sleep(1);
}

```

每次都要执行扫描, 才能获取周边热点数量

```

(61356) xxxxxxxx: AP number = 9
(64406) xxxxxxxx: SCAN MODE
(64456) xxxxxxxx: AP number = 7
(67506) xxxxxxxx: SCAN MODE
(67556) xxxxxxxx: AP number = 9
(70606) xxxxxxxx: SCAN MODE
(70656) xxxxxxxx: AP number = 7
(73706) xxxxxxxx: SCAN MODE
(73756) xxxxxxxx: AP number = 8
(76806) xxxxxxxx: SCAN MODE
(76856) xxxxxxxx: AP number = 9
(79906) xxxxxxxx: SCAN MODE
(79956) xxxxxxxx: AP number = 8
(83006) xxxxxxxx: SCAN MODE

```

这就是周边热点数量

根据热点数量获取每个热点的信息

```

esp_err_t esp_wifi_scan_get_ap_records(
    uint16_t *number, wifi_ap_record_t *ap_records) //将 wifi 热点的具体信息获取出来

*number: 传入扫描到的 wifi 热点数量
* ap_records : 返回连接AP后, AP 的信息, 信息有如下: (在esp_wifi_sta_get_ap_info介绍过)
    wifi_ap_record_t  uint8_t bssid[6]; 连接上路由后, 路由的 BSSID 可以用
                                                来判断设备连接的哪一个路由
    uint8_t ssid[33]; 连接上路由后, 查询路由名(SSID)
    uint8_t primary; 连接 AP 的信道(通道)
    int8_t rssi;      设备与 AP 之间的信号强度
加入 wifi_auth_mode_t authmode: 识别热点的加密方式
    WIFI_AUTH_OPEN
    WIFI_AUTH_WEP
    WIFI_AUTH_WPA_PSK
    WIFI_AUTH_WPA2_PSK
    WIFI_AUTH_WPA_WPA2_PSK
    WIFI_AUTH_WPA2_ENTERPRISE
    WIFI_AUTH_WPA3_PSK
    WIFI_AUTH_WPA2_WPA3_PSK

```

```

while(1){

    ESP_ERROR_CHECK(esp_wifi_scan_start(NULL, 1)); //启动wifi扫描功能, 这是阻塞模式, 执行比较慢

    uint16_t apCount = 0; //查询热点数量
    esp_wifi_scan_get_ap_num(&apCount); //获取周边设备数量
    ESP_LOGI(TAG, "AP number = %d", apCount);

    wifi_ap_record_t *list = (wifi_ap_record_t *)malloc(sizeof(wifi_ap_record_t) * apCount);
    //定义一个wifi_ap_record_t的结构体的链表空间
    ESP_ERROR_CHECK(esp_wifi_scan_get_ap_records(&apCount, list)); //获取上次扫描中找到的AP列表。
    ESP_LOGI(TAG, "=====");
    ESP_LOGI(TAG, "          SSID           |   RSSI   |     AUTH     ");
    ESP_LOGI(TAG, "=====");

```

```

int i;
for (i=0; i<apCount; i++)
{
    char *authmode;
    switch(list[i].authmode) {
    case WIFI_AUTH_OPEN:
        authmode = "WIFI_AUTH_OPEN";
        break;
    case WIFI_AUTH_WEP:
        authmode = "WIFI_AUTH_WEP";
        break;
    case WIFI_AUTH_WPA_PSK:
        authmode = "WIFI_AUTH_WPA_PSK";
        break;
    case WIFI_AUTH_WPA2_PSK:
        authmode = "WIFI_AUTH_WPA2_PSK";
        break;
    case WIFI_AUTH_WPA_WPA2_PSK:
        authmode = "WIFI_AUTH_WPA_WPA2_PSK";
        break;
    default:
        authmode = "Unknown";
        break;
    }
    ESP_LOGI(TAG, "%20.26s      %11d      %22.22s\n",list[i].ssid, list[i].rssi, authmode);
}//将链表的数据信息打印出来
free(list); //释放链表
sleep(1);

```

	SSID	RSSI	AUTH
I (627836) xxxxxxxx: SCAN MODE			
I (627886) xxxxxxxx: AP number = 9			
I (627886) xxxxxxxx: =====			
I (627886) xxxxxxxx:	OPPO A11	-46	WIFI_AUTH_WPA2_PSK
I (627906) xxxxxxxx:	metasia	-65	WIFI_AUTH_WPA_WPA2_PSK
I (627916) xxxxxxxx:	TP-LINK_57B5	-68	WIFI_AUTH_WPA_WPA2_PSK
I (627926) xxxxxxxx:	ChinaNet-XQm2	-73	WIFI_AUTH_WPA_WPA2_PSK
I (627936) xxxxxxxx:	test	-75	WIFI_AUTH_WPA_WPA2_PSK
I (627946) xxxxxxxx:	ROM 7-8	-80	WIFI_AUTH_WPA_WPA2_PSK
I (627956) xxxxxxxx:	metasia	-84	WIFI_AUTH_WPA_WPA2_PSK
I (627966) xxxxxxxx:	Dayanghuahua	-88	WIFI_AUTH_WPA_WPA2_PSK
I (627976) xxxxxxxx:	CMCC-HEVt	-95	WIFI_AUTH_WPA_WPA2_PSK

这就是执行结果，我 OPPO A11 离得最近信号最好。

ESP32 AP 模式使用

wifi_config_t 结构体设置

```
typedef union {
    wifi_ap_config_t ap; /*< AP 的配置 */
    wifi_sta_config_t sta; /*< STA 的配置 */
} wifi_config_t;

wifi_ap_config_t ap //在 AP 模式下配置 ap 参数
    uint8_t ssid[32]; 设置设备本身 AP 热点名
    uint8_t password[64]; 设置设备 AP 密码，就算没有密码也要默认写 "<password>" 
    uint8_t ssid_len; 计算 ssid 字符长度填入 ssid_len
    uint8_t channel;      ESP32 AP 通道(频道)
    wifi_auth_mode_t authmode; 加密方式，如 WIFI_AUTH_OPEN
                            WIFI_AUTH_WEP
                            WIFI_AUTH_WPA_PSK
                            WIFI_AUTH_WPA2_PSK
                            WIFI_AUTH_WPA_WPA2_PSK
                            WIFI_AUTH_WPA2_ENTERPRISE
                            WIFI_AUTH_WPA3_PSK
                            WIFI_AUTH_WPA2_WPA3_PSK
    uint8_t ssid_hidden; 设置为 0 就是 SSID 广播模式
    uint8_t max_connection; 最多只能被 4 个 STA 设备同时连接
    uint16_t beacon_interval; 信标间隔 100~60000ms， 默认设置 100ms

void app_main()
{
    esp_err_t err = nvs_flash_init(); //记住在做 wifi 功能时，一定要对 nvs_flash 初始化，不然板子会一直重启
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        // NVS partition was truncated and needs to be erased
        // Retry nvs_flash_init
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );

    tcpip_adapter_init(); //初始化TCP/IP适配层 不然启动 wifi 模式会导致模块不停重启
    ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT(); //这个函数是常规操作，用来给 wifi 初始化赋值
    ESP_ERROR_CHECK( esp_wifi_init(&cfg) ); //wifi 使用之前，必须执行这段初始化

    /*设置AP模式的参数*/
    wifi_config_t ap_config = {
        .ap = {
            .ssid = "ESP32",
            .password = "<password>", //不设置 AP 密码，但是必须写入默认值
            .ssid_len = 5, //SSID 长度为 5
            .ssid_hidden=0, //广播本设备 SSID，这样其它设备都能搜索到
            .max_connection = 4, //最大连接数
            .authmode = WIFI_AUTH_OPEN, //没有密码属于 OPEN 模式
            .beacon_interval=100
        }
    }; //ap 接入点参数配置。

    ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_AP) ); //wifi 启动为 AP 模式
    ESP_ERROR_CHECK( esp_wifi_set_config(WIFI_IF_AP, &ap_config) ); //wifi 启动为 AP 模式
    ESP_ERROR_CHECK( esp_wifi_start() ); //启动 wifi
    esp_wifi_connect(); //启动连接

    while(1)[
        sleep(1);
    ]
}
```

事件处理函数要设置

设置 AP 热点名和密码等

这里改成 AP 模式

```
I (367) phy: phy version: 4180, cb3948e, Sep 12 2019, 16:39:13, 0, 0
I (369) wifi:mode : softAP (fc:f5:c4:3c:4a:1d)
I (370) wifi:Total power save buffer number: 16
I (371) wifi:Init max length of beacon: 752/752
I (376) wifi:Init max length of beacon: 752/752
```

等待外部设备 STA 连接。softAP(fc: f5: c4: 3c: 4a: 1d) 就是本 AP 设备的 MAC 地址

```
I (173299) wifi:new:<1,0>, old:<1,1>, ap:<1,1>, sta:<255,255>, prof:1
I (173301) wifi:station: 18:d0:c5:ca:92:01 join, AID=1, bgn, 20
I (173589) tcpip_adapter: softAP assign IP to station,IP is: 192.168.4.2
```

有设备接入！！，接入的 STA 设备 MAC 地址为 18:d0:c5:ca:92:01，本设备分配 IP 192.168.4.2 给 STA 设备。

现在我手机发现 ESP32 这个 AP 热点不可上网

如果有新的 STA 设备连接，需要进行统计和解析

event: SYSTEM_EVENT_AP_START	第一次启动 AP 事件宏
event: SYSTEM_EVENT_AP_STACONNECTED	STA 连入 AP 设备事件宏
event: SYSTEM_EVENT_AP_STADISCONNECTED	STA 断开 AP 设备事件宏

```
static esp_err_t event_handler(void *ctx, system_event_t *event)
{
    switch (event->event_id) [
        case SYSTEM_EVENT_STA_START:
            // esp_wifi_connect(); //连接热点
            break;
        case SYSTEM_EVENT_STA_CONNECTED:
            ESP_LOGI(TAG, "connect AP success");
            break;
        case SYSTEM_EVENT_STA_GOT_IP:
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            ESP_LOGI(TAG, "connect AP disconnect");
            //esp_wifi_connect(); //连接热点
            break;
        case SYSTEM_EVENT_AP_START: //AP启动后会先执行一次AP_START分支
            ESP_LOGI(TAG, "AP start ");
            break;
        case SYSTEM_EVENT_AP_STACONNECTED: //有STA设备连接进来会执行该分支
            ESP_LOGI(TAG, "AP connect ");
            break;
        case SYSTEM_EVENT_AP_STADISCONNECTED: //如果STA设备断开，会进来执行该分支
            ESP_LOGI(TAG, "AP disconnect ");
            break;
        default:
            break;
    ]
}
```

这是在事件回调函数加入的 AP 宏的内容

```

void app_main()
{
    esp_err_t err = nvs_flash_init(); //记住在做wifi功能时，一定要对nvs_flash初始化，不然板子会一直重启
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        // NVS partition was truncated and needs to be erased
        // Retry nvs_flash_init
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );

    tcpip_adapter_init(); //初始化TCP/IP适配层 不然启动wifi模式会导致模块不停重启
    ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT(); //这个函数是常规操作，用来给wifi初始化赋值
    ESP_ERROR_CHECK( esp_wifi_init(&cfg) ); //wifi使用之前，必须执行这段初始化

    /*设置AP模式的参数*/
    wifi_config_t ap_config = {
        .ap = {
            .ssid = "ESP32"
    }
}

```

设备初始化和上面 AP 模式启动一样不变，只在 event_handler 事件函数里面加了内容

```

I (392) [xxxxxxxx: AP start]
I (6763) wifi:new:<1,0>, old:<1,1>, ap:<1,1>, sta:<255,255>, prof:1
I (6765) wifi:station: 18:d0:c5:ca:92:01 join, AID=1, bgn, 20
I (6766) [xxxxxxxx: AP connect]
I (7145) tcpip_adapter: softAP assign IP to station,IP is: 192.168.4.2
I (9834) wifi:station: 18:d0:c5:ca:92:01 leave, AID = 1, bss_flags is 131171, bss:0x3ffc676c
I (9835) wifi:new:<1,0>, old:<1,0>, ap:<1,1>, sta:<255,255>, prof:1
I (9838) [xxxxxxxx: AP disconnect]
I (13025) wifi:new:<1,0>, old:<1,0>, ap:<1,1>, sta:<255,255>, prof:1
I (13026) wifi:station: 18:d0:c5:ca:92:01 join, AID=1, bgn, 20
I (13027) [xxxxxxxx: AP connect]
I (13336) tcpip_adapter: softAP assign IP to station,IP is: 192.168.4.2
I (14814) wifi:station: 18:d0:c5:ca:92:01 leave, AID = 1, bss_flags is 131171, bss:0x3ffc676c
I (14815) wifi:new:<1,0>, old:<1,0>, ap:<1,1>, sta:<255,255>, prof:1
I (14818) [xxxxxxxx: AP disconnect]

```

AP 设备启动之后先执行 AP start 分支

如果有 STA 连接进来执行 AP connect 分支

如果 STA 与本设备断开执行 AP disconnect 分支

可以用这种方式计算设备连接次数，也可以在 **SYSTEM_EVENT_AP_STACONNECTED** 分支里面及时获取 STA 设备的 IP 地址，设备名称，MAC 地址。

获取接入 STA 设备的 IP 地址，MAC 地址

完成该功能需要《如果有新的 STA 设备连接，需要进行统计和解析》的内容

ESP32 蓝牙配网

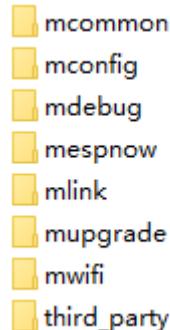
乐鑫蓝牙配网安卓 APP 下载地址

<https://github.com/EspressifApp/EspBlufiForAndroid/releases>

注意编译蓝牙库要注意的事项，使用其它有些库也需要注意。

esp-mdf-v1.0 → components

在 MDF 的 components 目录里面没有蓝牙库

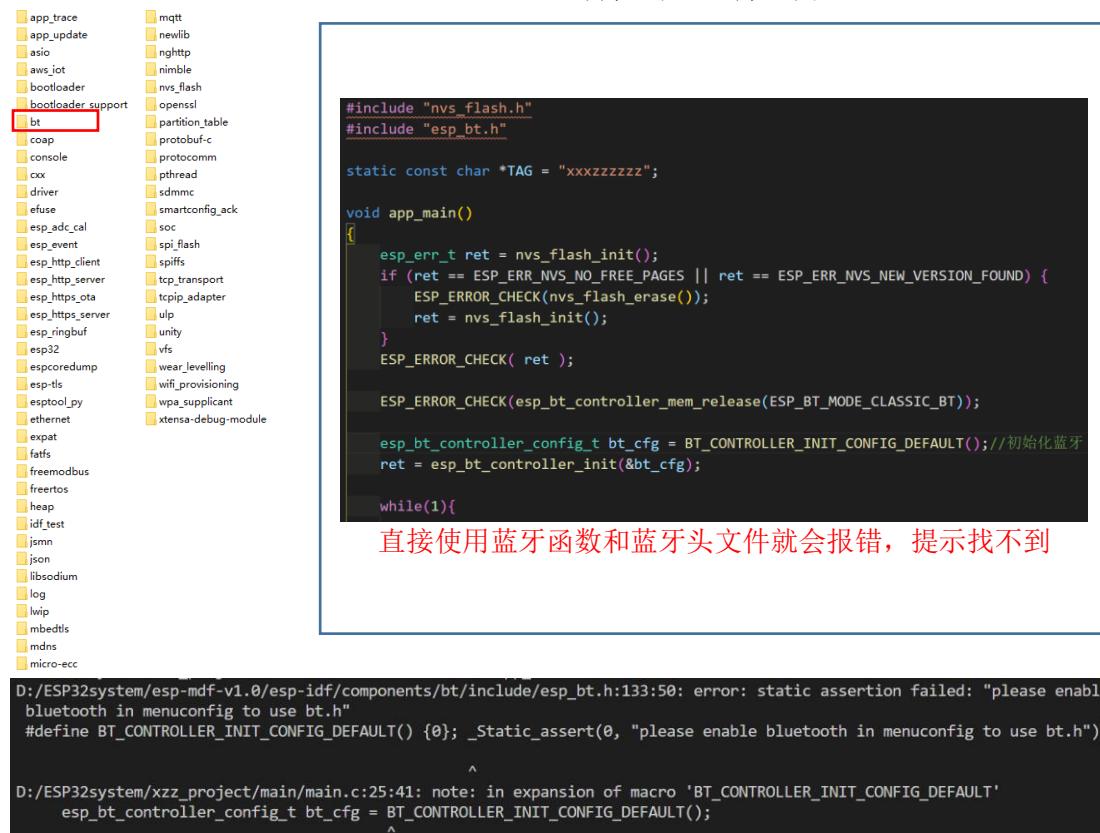


都是跟 mesh 网络相关的库

因为其它的功能库，都在 IDF 库里面

esp-mdf-v1.0 → esp-idf → components

蓝牙库也在 IDF 库里面



提示文件找不到，需要使能 `bt.h`，其实就是在去 `make menuconfig` 把蓝牙打开

```
CC build/lwip/port/esp32/debug/lwip_debug.o  
AR build/lwip/liblwip.a
```

XZ666@XZ666PC MINGW32 /d/ESP32/system/xzz project

编译完也不会再显示报错，只是没有结果。

设置 make menuconfig 打开蓝牙功能

Component config --->

进入组件，其实前面 wifi 编译没有报错是因为 wifi 功能默认打开了的。

[] Bluetooth --->

[] Bluetooth

本来蓝牙功能就没打开

[*] Bluetooth

Bluetooth controller --->

[*] Bluedroid Enable (NEW) --->

打开蓝牙功能

里面还有其它配置功能，但是我的 BLUFI 蓝牙配网功能是默认打开就有的。

```
#include "nvs_flash.h"
#include "esp_bt.h"

static const char *TAG = "xxxxxxxx";

void app_main()
{
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK( ret );

    ESP_ERROR_CHECK(esp_bt_controller_mem_release(ESP_BT_MODE_CLASSIC_BT));

    esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT(); // 初始化蓝牙
    ret = esp_bt_controller_init(&bt_cfg);

    while(1){
```

再次 make 编译

```
LD build/xzz_project.elf
esptool.py v2.8
To flash all build output, run 'make flash' or:
python /d/ESP32system/esp-mdf-v1.0/esp-idf/components/esptool_py/
0 --before default_reset --after hard_reset write_flash -z --flas
d/ESP32system/xzz_project/build/bootloader/bootloader.bin 0x10000
/d/ESP32system/xzz_project/build/partitions_singleapp.bin
```

编译通过

```
I (135) BTDM INIT: BT controller compile version [e989f20]
I (135) system_api: Base MAC address is not set, read default base MAC address
from BLK0 of EFUSE
```

根据打印结果证明 BLE 蓝牙启动起来了。

```

esp_err_t esp_bt_controller_mem_release(esp_bt_mode_t mode)
//释放蓝牙某种模式下的内存占用

mode: ESP_BT_MODE_BLE 设置蓝牙为 BLE 模式运行
ESP_BT_MODE_CLASSIC_BT 设置蓝牙为经典模式运行
ESP_BT_MODE_BTDM 设置蓝牙运行在双模式情况下

esp_bt_controller_config_t BT_CONTROLLER_INIT_CONFIG_DEFAULT()
//获取蓝牙默认参数

esp_err_t esp_bt_controller_init(esp_bt_controller_config_t *cfg) //蓝牙初始化

esp_err_t esp_bluetooth_enable(void)//使能蓝牙, 必须在 esp_bt_controller_init 之后
如果使能蓝牙之后不小心再次执行 esp_bluetooth_enable() 系统就会崩溃, 如果想再次
esp_bluetooth_enable(), 就必须先关闭蓝牙 esp_bluetooth_disable() 之后再
esp_bluetooth_enable(void)

esp_err_t esp_bluetooth_disable(void)//关闭蓝牙, 必须在 esp_bluetooth_deinit() 之前

esp_err_t esp_ble_gap_start_advertising(esp_ble_adv_params_t *adv_params)
//gap 层设置蓝牙广播参数, 启动蓝牙广播。
*adv_params: 蓝牙广播参数设置结构
    struct esp_ble_adv_params_t
        uint16_t adv_int_min //蓝牙广播最小间隔时间
        uint16_t adv_int_max //蓝牙广播最大间隔时间
        adv_type //广播模式 ADV_TYPE_IND
            ADV_TYPE_DIRECT_IND_HIGH
            ADV_TYPE_SCAN_IND
            ADV_TYPE_NONCONN_IND
            ADV_TYPE_DIRECT_IND_LOW
        own_addr_type //设备地址类型
            BLE_ADDR_TYPE_PUBLIC 使用 public 进行广播
            BLE_ADDR_TYPE_RANDOM 使用可解析地址进行广播
            BLE_ADDR_TYPE_RPA_RANDOM 使用静态随机地址进行广播
        channel_map //广播通道
            ADV_CHNL_37 或 ADV_CHNL_38 或 ADV_CHNL_39 或 ADV_CHNL_ALL 所有通道
        adv_filter_policy //从机广播过滤设置
            ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY (可被任何主机和其它从机扫描
到该设备, 建立连接, 不使用白名单)
            ADV_FILTER_ALLOW_SCAN_WLST_CON_ANY (可以建立所有连接请求和
让白名单中的设备扫描到)
            ADV_FILTER_ALLOW_SCAN_ANY_CON_WLST (可以让任何设备都找到, 但是
建立连接只按照白名单的设备来建立), 就是上一个宏反过来
            ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST (只有白名单中的设备能扫描
到本设备, 只有白名单中的设备可以连接本设备)

esp_err_t esp_ble_gap_set_device_name(const char *name)
//设置本设备的蓝牙名称, 这样其它手机扫到本设备就可以显示名称, 而不是显示 MAC 地址

esp_err_t esp_ble_gap_config_adv_data(esp_ble_adv_data_t *adv_data) //覆盖 BTv 默认的 ADV 参数
*adv_data : 定义广播用的数据结构体

```

```

#include "nvs_flash.h"

#include "esp_bt.h"
#include "esp_bt_main.h"
#include "esp_ble_gap.h"
#include "esp_bt_device.h"

static esp_ble_adv_params_t example_adv_params = {
    .adv_int_min          = 0x100,
    .adv_int_max          = 0x100,
    .adv_type              = ADV_TYPE_IND,
    .own_addr_type        = BLE_ADDR_TYPE_PUBLIC,
    //peer_addr            =
    //peer_addr_type       =
    .channel_map          = ADV_CHNL_ALL,
    .adv_filter_policy    = ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY,
};

static esp_ble_adv_data_t example_adv_data = {
    .set_scan_rsp = false,
    .include_name = true,
    .include_txpower = true,
    .min_interval = 0x0006, //slave connection min interval, Time = min_interval *
1.25 msec
    .max_interval = 0x0010, //slave connection max interval, Time = max_interval *
1.25 msec
    .appearance = 0x00,
    .manufacturer_len = 0,
    .p_manufacturer_data = NULL,
    .service_data_len = 0,
    .p_service_data = NULL,
    .service_uuid_len = 16,
    // .p_service_uuid = example_service_uuid128,
    .flag = 0x6,
};

void app_main()
{
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK( ret );

    esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_bt_controller_init(&bt_cfg)); //初始化蓝牙控制器
    ESP_ERROR_CHECK(esp_bt_controller_enable(ESP_BT_MODE_BLE)); //设置蓝牙模式
    ESP_ERROR_CHECK(esp_ble_gap_start_advertising(&example_adv_params)); //gap 层设置蓝牙广播参数，和启动蓝牙广播
    esp_ble_gap_set_device_name("XZZ_HOME"); //设置蓝牙设备名
    esp_ble_gap_config_adv_data(&example_adv_data); //执行GAP配置后 蓝牙设备名才能扫描显示出来
}

蓝牙广播成功，手机搜索到 XZZ_HOME 设备，但是连接之后就断开了，下面进行完善。

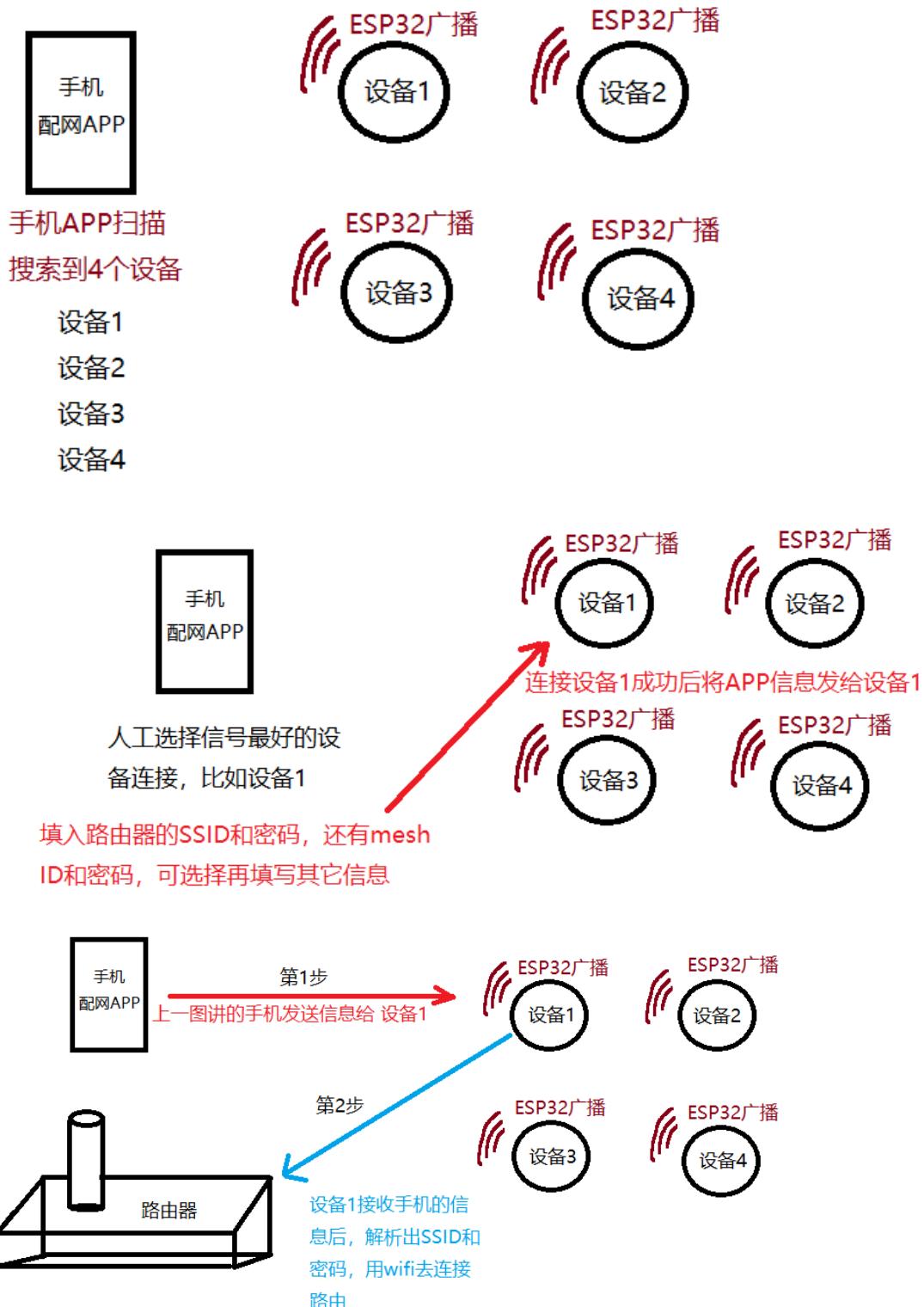
```

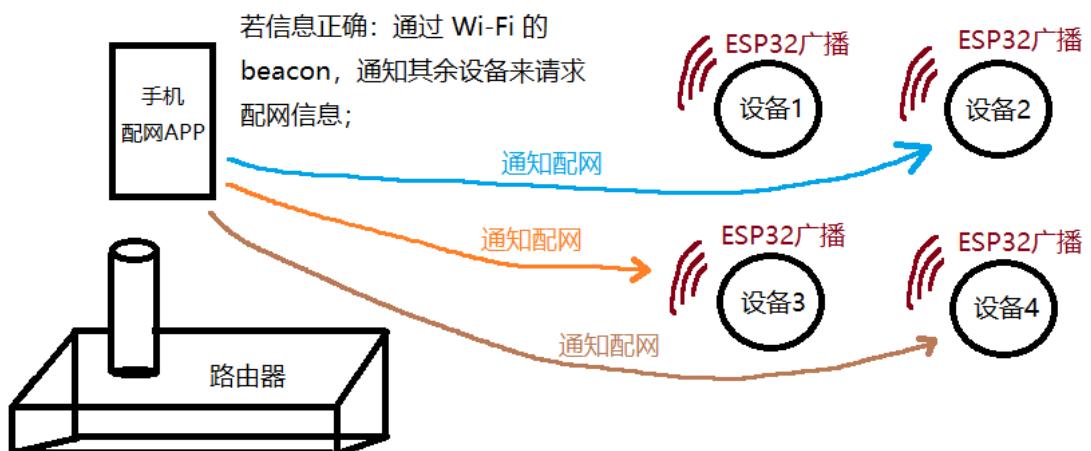
蓝牙 BLE 采用 mesh 网络进行配网，实现每个节点都可以同时配网

mesh 网络看我下面的章节，里面出现 mdf 函数的内容，也在下面 mesh 网络章节中
这里使用 MDF 库里面的 mconfig 功能对 mesh 节点进行蓝牙配网

蓝牙配置 mesh 网络分为两种模式，Mconfig-Blufi 和 Mconfig-Chain

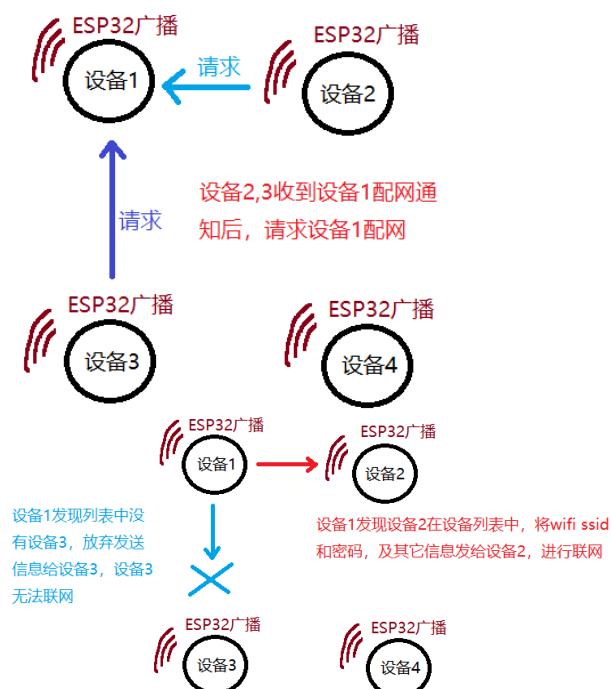
下面讲解 Mconfig-Blufi 配网方式

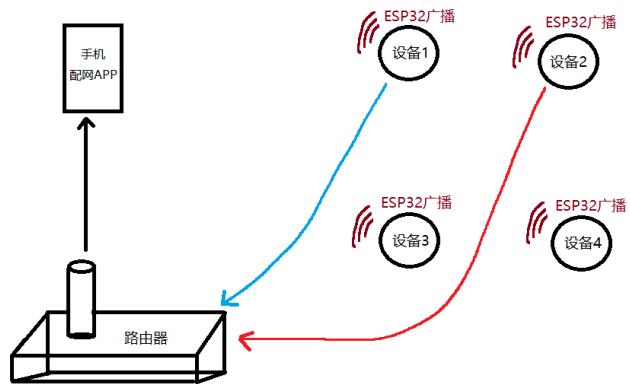




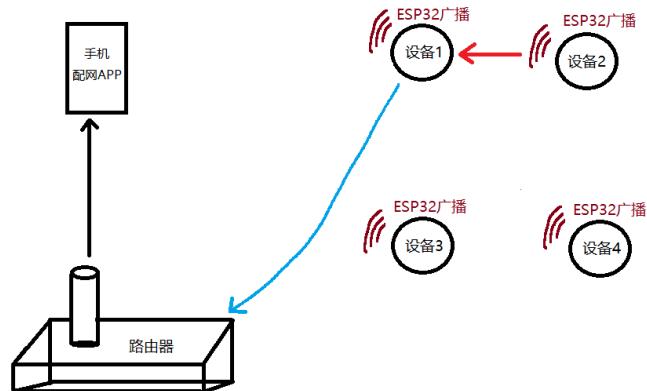
配网单个设备 **Mconfig-Blufi** 模式已经完成。, 如果不想手机一个一个设备去配网, 而是一次性将其它设备配网完, 那么接着第 4 步, 使用 **Mconfig-Chain** 继续向下进行。

Mconfig-Chain 已配网设备给未配网设备传递配网信息
设备 1, 通知其余设备



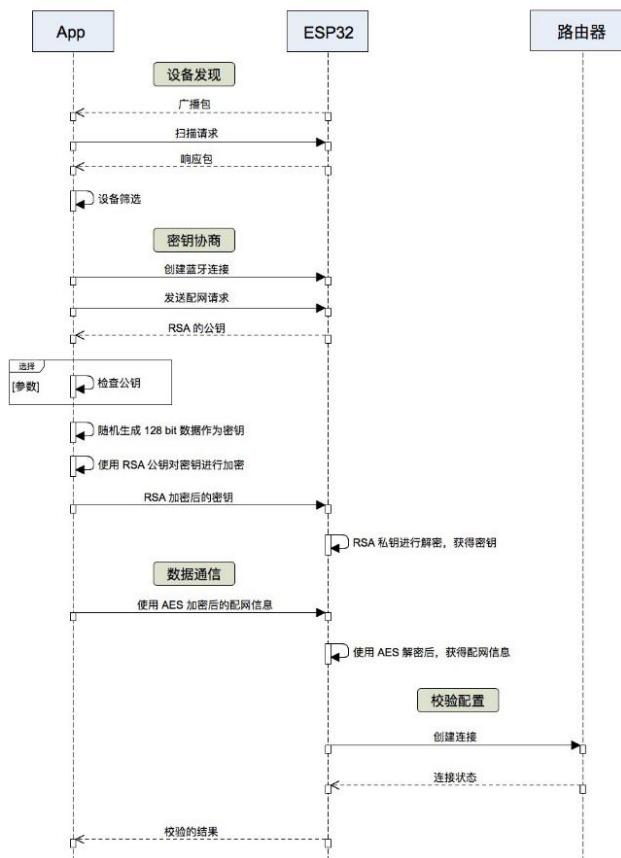


还有一种情况就是设备 2 可能是子节点。



设备 2 经过根节点设备 1 联网。配网过程完成

Mconfig-BluFi 是基于 BluFi (由 Espressif 定义的蓝牙配网协议) 的配网协议，流程如下



```
esp_err_t esp_wifi_set_ps(wifi_ps_type_t type) //设置 wifi 省电模式  
type: 填入 wifi_ps_type_t 数据类型  
    WIFI_PS_NONE //不进入省电模式  
    WIFI_PS_MIN_MODEM //在此模式下本机在每一个 DTIM 周期接收一个信标  
    WIFI_PS_MAX_MODEM //最大省电模式, 此模式下本机接收信标时间间隔由  
                        wifi_sta_config_t 结构里面的参数决定
```

mespnow_init() //初始化 ESP NOW 协议模式

```
mdf_err_t mconfig_btluifi_init(const mconfig_btluifi_config_t *config)  
//初始化蓝牙网络配置
```

*config: 配置 mconfig_btluifi_config_t 参数

```
char name[MCONFIG_BLUIFI_NAME_SIZE] //本地设备和外设名称, 如果名  
称长度大于 10 字节, 它将覆盖 custom_data 变量, 从而导致 custom_data  
不可以使用  
uint16_t company_id //公司标识标识符  
uint16_t tid //设备类型序号, 序号不同 APP 收到的蓝牙图标也不同  
uint8_t custom_size //自定义数据大小  
uint8_t custom_data[MCONFIG_BLUIFI_CUSTOM_SIZE] //自定义数据放  
在蓝牙广播包里  
bool only_beacon //只发送信标, 不支持连接
```

mdf_err_t mconfig_chain_slave_init(void) //子节点初始化获取根节点发来的网络信
息。

```
mdf_err_t mconfig_queue_read(mconfig_data_t **mconfig_data,  
TickType_t wait_ticks)  
//读取队列中的蓝牙配置信息, 因为手机发送过来的配置信息就在队列中, 所以定义个变量  
来读取
```

**mconfig_data: 填入接收队列数据的指针, 必须下 MDF_FREE 之后调用
mwifi_config_t config //APP 发过来的路由器 SSID 密码和 mesh ID
密码都在这变量里面, 可以调用获取
uint8_t custom[32 + CONFIG_MCONFIG_CUSTOM_EXTERN_LEN] //APP 为
用户自定义的蓝牙数据, 如 UUID, 用户名, 令牌
uint16_t whitelist_size //APP 发送过来, 设置的白名单大小
mconfig_whitelist_t whitelist_data[0] //白名单设备的地址
wait_ticks: 阻塞等待 APP 蓝牙发送数据过来, 可以填入阻塞时间

mdf_err_t mconfig_chain_slave_deinit(void) //停止 mconfig_chain_slave_init 子
节点接收根节点初始化网络信息, 关闭 chain slave 设备

mdf_err_t mconfig_btluifi_deinit(void) //解除蓝牙网络配置

代码范例:

```
#include "esp_bt.h"  
#include "mespnow.h"  
#include "mconfig_btluifi.h"  
#include "mconfig_chain.h"
```

```

static const char *TAG = "XZZ_mconfig";
static mdf_err_t event_loop_cb(mdf_event_loop_t event, void *ctx)
{
    MDF_LOGI("event_loop_cb, event: %d", event);
    switch (event) {
        case MDF_EVENT_MWIFI_STARTED:
            MDF_LOGI("MDF_EVENT_MWIFI_STARTED");
            break;
        case MDF_EVENT_MWIFI_PARENT_CONNECTED:
            MDF_LOGI("MDF_EVENT_PARENT_CONNECTED");
            break;
        case MDF_EVENT_MWIFI_ROUTING_TABLE_ADD:
        case MDF_EVENT_MWIFI_ROUTING_TABLE_REMOVE:
            MDF_LOGI("total_num: %d", esp_mesh_get_total_node_num());
            break;
        case MDF_EVENT_MCONFIG_BLUFI_CONNECTED:
            MDF_LOGI("MDF_EVENT_MCONFIG_BLUFI_CONNECTED");
            break;
        case MDF_EVENT_MCONFIG_BLUFI_STA_CONNECTED:
            MDF_LOGI("MDF_EVENT_MCONFIG_BLUFI_STA_CONNECTED");
            break;
        /***< Add a custom communication process */
        case MDF_EVENT_MCONFIG_BLUFI_RECV:
            {
                mconfig_bluifi_data_t *bluifi_data = (mconfig_bluifi_data_t *)ctx;
                MDF_LOGI("recv data: %.s", bluifi_data->size, bluifi_data->data);
                // ret = mconfig_bluifi_send(bluifi_data->data, bluifi_data->size);
                // MDF_ERROR_BREAK(ret != MDF_OK, "<%> mconfig_bluifi_send", mdf_err_to_name(ret));
                break;
            }
        default:
            break;
    }
    return MDF_OK;
}
void app_main()
{
    char name[28] = {0x0};
    char custom_data[32] = {0x0};
    mdf_err_t ret = nvs_flash_init(); //常规操作 NVS 初始化
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        MDF_ERROR_ASSERT(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT(); //常规操作，用来给 wifi 分配初始化值
    tcpip_adapter_init(); //这里不变，初始化 TCP IP 协议层
    MDF_ERROR_ASSERT(mdf_event_loop_init(event_loop_cb)); //MDF 库使用下，STA 模式启动之后的回调函数
    MDF_ERROR_ASSERT(esp_event_loop_init(NULL, NULL)); //STA 连接后的回调函数，这里不设置，因为用的 MDF 库
    MDF_ERROR_ASSERT(esp_wifi_init(&cfg)); //常规操作，wifi 使用之前必须执行这段初始化
    MDF_ERROR_ASSERT(esp_wifi_set_storage(WIFI_STORAGE_FLASH)); //wifi 配置可以存储进 flash，也可以不存储
    MDF_ERROR_ASSERT(esp_wifi_set_mode(WIFI_MODE_STA)); //wifi 设置为 STA 模式启动
    MDF_ERROR_ASSERT(esp_wifi_set_ps(WIFI_PS_NONE)); //设置 wifi 是否进入省电模式，我这里选择 NONE，不设置
    MDF_ERROR_ASSERT(esp_mesh_set_6m_rate(false)); //设置网络最小数据包传输速度，这里取消最小数据包传输速度
    MDF_ERROR_ASSERT(esp_wifi_start()); //启动 wifi STA 模式
    MDF_ERROR_ASSERT(mespnow_init()); //初始化 ESP NOW
    mconfig_bluifi_config_t bluifi_config = { //蓝牙信息配置变量，可以将自定义的蓝牙设备名称放入改变量
        .tid = 1, /*< 设备类型序号，用于区分不同产品，  
APP 搜索蓝牙的时候根据 tid 序号不同，显示的图标也不同。*/
        .company_id = MCOMMON_ESPRESSIF_ID, //公司标识标识符
    };
    //strncpy(bluifi_config.name, "XZZ BLE Mconfig", sizeof(bluifi_config.name) - 1); //将自定义的蓝牙设备名称放入 bluifi 配置
    char name[] = "XZZ BLE Mconfig"; //给蓝牙设备取名，便于搜索
    strncpy(bluifi_config.name, name, sizeof(bluifi_config.name) - 1); //将自定义的蓝牙设备名称放入 bluifi 配置
    MDF_LOGI("XZZ BLE Mconfig");
    MDF_ERROR_ASSERT(mconfig_bluifi_init(&bluifi_config)); //初始化蓝牙网络配置
    MDF_ERROR_ASSERT(mconfig_chain_slave_init()); //本机为从机，初始化蓝牙配置的网络，接收到网络配置信息放入队列
    mconfig_data_t *mconfig_data = NULL; //用来存放手机发过来的 SSID 密码和 mesh ID 密码
    MDF_ERROR_ASSERT(mconfig_queue_read(&mconfig_data, portMAX_DELAY)); //阻塞等待读取网络配置信息
    /*打印手机发过来的网络配置信息，看看是否正确*/
    MDF_LOGI("XZZ BLE Mconfig get router ssid: %s", mconfig_data->config.router_ssid);
    MDF_LOGI("XZZ BLE Mconfig get router password: %s", mconfig_data->config.router_password);
    MDF_LOGI("XZZ BLE Mconfig get router mesh ID: %s", mconfig_data->config.mesh_id);
    MDF_LOGI("XZZ BLE Mconfig get router mesh password: %s", mconfig_data->config.mesh_password);
    MDF_ERROR_ASSERT(mconfig_chain_slave_deinit());
    //接收到手机蓝牙发过来的数据后，必须释放 mconfig_chain_slave_init 中的资源
    MDF_ERROR_ASSERT(mconfig_bluifi_deinit()); //解除蓝牙网络配置初始化
    while(1)
    {
        sleep(1);
    }
}

```

strncpy(char *dest, char *src, ...)
 因为 src 是指针，所以必须传入地址，直接写字符串会导致设备重启

```

I (224) [XZZ_mconfig, 95]: XZZ BLE Mconfig
I (228) [XZZ_mconfig, 12]: event_loop_cb, event: 8704
I (3448) BTDM_INIT: BT controller compile version [e989f20]
I (3449) system_api: Base MAC address is not set, read default base MAC address from BLK0 of EFUSE
I (3855) [mconfig_bluifi, 539]: BLUFI init finish, set ble advertising data
I (3855) system_api: Base MAC address is not set, read default base MAC address from BLK0 of EFUSE
I (3866) [XZZ_mconfig, 12]: event_loop_cb, event: 8449
I (3869) [mconfig_bluifi, 882]: start ble advertising
I (7769) [mconfig_chain, 361]: Generate RSA public and private keys
I (7770) wifi:ic_enable_sniffer

```

蓝牙启动之后等待手机连接

手机搜索到 XZZ BLE Mconfig 蓝牙设备，设备的 MAC 地址为 fc:f5:c4:3c:4a:1c(其实这就是设备的网卡地址，被设备广播了)

```

I (43312) [XZZ_mconfig, 31]: MDF_EVENT_MCONFIG_BLUFI_STA_CONNECTED
I (43545) wifi:new:<l,1>, old:<l1,0>, ap:<255,255>, sta:<l,1> prof:11
I (43547) wifi:state: init -> auth (b0)
I (43552) wifi:state: auth -> assoc (0)
I (43569) wifi:state: assoc -> run (10)
I (43605) wifi:connected with metasia, aid = 2, channel 1, 40U, bssid = 50:fa:84:3c:b7:90
I (43606) wifi:security type: 4, phy: bgn, rssi: -79
I (43612) wifi:pm start, type: 0

I (43633) [XZZ_mconfig, 31]: MDF_EVENT_MCONFIG_BLUFI_STA_CONNECTED
I (44400) event: sta ip: 192.168.1.135, mask: 255.255.255.0, gw: 192.168.1.1
I (45759) [mconfig_bluifi, 568]: BLUFI ble disconnect
I (45759) [XZZ_mconfig, 10]: event_loop_cb, event: 8449
I (45760) [XZZ_mconfig, 10]: event_loop_cb, event: 8710
I (45765) [XZZ_mconfig, 10]: event_loop_cb, event: 8707
E (45772) wifi:STA is scanning or connecting, or AP has connected with external ST

W (45781) [mconfig_chain, 314]: esp_wifi_set_channel, channel: 2, second: 1
I (45788) [XZZ_mconfig, 100]: XZZ BLE Mconfig get router ssid: metasia
I (45795) [XZZ_mconfig, 101]: XZZ BLE Mconfig get router passwd: metasia0902
I (45803) [XZZ_mconfig, 102]: XZZ BLE Mconfig get router mesh ID: P# 123456789
I (45811) [XZZ_mconfig, 103]: XZZ BLE Mconfig get router mesh passwd: 123456789

```

收到的这些 ID 和密码可以直接放入自己定义的存储器，以便 wifi 使用或者 mesh 使用。

以上程序并不完善，只是证明证明获取手机发来的用户 SSID 和密码之类的信息，然后还要将这些信息写入 wifi，下面实现：

```

#include "esp_bt.h"
#include "mespnow.h"
#include "mconfig_bluifi.h"
#include "mconfig_chain.h"

static const char *TAG = "XZZ_mconfig";
static mdf_err_t event_loop_cb(mdf_event_loop_t event, void *ctx)
{
    MDF_LOGI("event_loop_cb, event: %d", event);
    switch (event) {
        case MDF_EVENT_MWIFI_STARTED:
            MDF_LOGI("MDF_EVENT_MWIFI_STARTED");
            break;
        case MDF_EVENT_MWIFI_PARENT_CONNECTED:
            MDF_LOGI("MDF_EVENT_PARENT_CONNECTED");
            break;
        case MDF_EVENT_MWIFI_ROUTING_TABLE_ADD:
        case MDF_EVENT_MWIFI_ROUTING_TABLE_REMOVE:
            MDF_LOGI("total_num: %d", esp_mesh_get_total_node_num());
            break;
        case MDF_EVENT_MCONFIG_BLUFI_CONNECTED:
            MDF_LOGI("MDF_EVENT_MCONFIG_BLUFI_CONNECTED");
            break;
        case MDF_EVENT_MCONFIG_BLUFI_STA_CONNECTED:
            MDF_LOGI("MDF_EVENT_MCONFIG_BLUFI_STA_CONNECTED");
            break;
        /*< Add a custom communication process */
        case MDF_EVENT_MCONFIG_BLUFI_RECV: {
            mconfig_bluifi_data_t *bluifi_data = (mconfig_bluifi_data_t *)ctx;
            MDF_LOGI("recv data: %.*s", bluifi_data->size, bluifi_data->data);
            // ret = mconfig_bluifi_send(bluifi_data->data, bluifi_data->size);
            // MDF_ERROR_BREAK(ret != MDF_OK, "<%> mconfig_bluifi_send", mdf_err_to_name(ret));
        }
    }
}

```

```

        break;
    }
    default:
        break;
}
return MDF_OK;
}

void app_main()
{
    char custom_data[32] = {0x0};
    mdf_err_t ret = nvs_flash_init(); //常规操作 NVS 初始化
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        MDF_ERROR_ASSERT(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT(); //常规操作, 用来给 wifi 分配初始化值
    tcpip_adapter_init(); //这里不变, 初始化 TCP IP 协议层
    MDF_ERROR_ASSERT(mdf_event_loop_init(event_loop_cb)); //MDF 库使用下, STA 模式启动之后的回调函数
    MDF_ERROR_ASSERT(esp_event_loop_init(NULL, NULL)); //STA 连接后的回调函数, 这里不设置, 因为用的 MDF 库
    MDF_ERROR_ASSERT(esp_wifi_init(&cfg)); //常规操作, wifi 使用之前必须执行这段初始化
    MDF_ERROR_ASSERT(esp_wifi_set_storage(WIFI_STORAGE_FLASH)); //wifi 配置可以存储进 flash, 也可以不存储
    MDF_ERROR_ASSERT(esp_wifi_set_mode(WIFI_MODE_STA)); //wifi 设置位 STA 模式启动
    MDF_ERROR_ASSERT(esp_wifi_set_ps(WIFI_PS_NONE)); //设置 wifi 是否进入省电模式, 我这里选择 NONE, 不设置
    MDF_ERROR_ASSERT(esp_mesh_set_6m_rate(false)); //设置网络最小数据包传输速度, 这里取消最小数据包传输速度
    MDF_ERROR_ASSERT(esp_wifi_start()); //启动 wifi STA 模式
    MDF_ERROR_ASSERT(mspnow_init()); //初始化 ESP NOW
/*    char name[28] = {0x0};
    uint8_t sta_mac[6] = {0};
    MDF_ERROR_ASSERT(esp_wifi_get_mac(ESP_IF_WIFI_STA, sta_mac));
    sprintf(name, "ESP-WIFI-MESH_%02x%02x", sta_mac[4], sta_mac[5]);
*/
    mconfig_bluifi_config_t bluifi_config = { //蓝牙信息配置变量, 可以将自定义的蓝牙设备名称放入改变量
        .tid = 1, /*< 设备类型序号, 用于区分不同产品,
                    APP 搜索蓝牙的时候根据 tid 序号不同, 显示的图标也不同. */
        .company_id = MCOMMON_ESPRESSIF_ID, //公司标识标识符
    };
    char name[] = "XZZ BLE Mconfig"; //给蓝牙设备取名, 便于搜索
    strncpy(bluifi_config.name, name, sizeof(bluifi_config.name) - 1); //将自定义的蓝牙设备名称放入 bluifi 配置
    MDF_LOGI("XZZ BLE Mconfig");
    MDF_ERROR_ASSERT(mconfig_bluifi_init(&bluifi_config)); //初始化蓝牙网络配置
    MDF_ERROR_ASSERT(mconfig_chain_slave_init()); //本机为从机, 初始化蓝牙配置的网络, 接收到网络配置信息放入队列
    mconfig_data_t *mconfig_data = NULL; //用来存放手机发过来的 SSID 密码和 mesh ID 密码
    MDF_ERROR_ASSERT(mconfig_queue_read(&mconfig_data, portMAX_DELAY)); //阻塞等待读取网络配置信息
    /*打印手机发过来的网络配置信息, 看看是否正确*/
    MDF_LOGI("XZZ BLE Mconfig get router ssid: %s", mconfig_data->config.router_ssid);
    MDF_LOGI("XZZ BLE Mconfig get router passwd: %s", mconfig_data->config.router_password);
    MDF_LOGI("XZZ BLE Mconfig get router mesh ID: %s", mconfig_data->config.mesh_id);
    MDF_LOGI("XZZ BLE Mconfig get router mesh passwd: %s", mconfig_data->config.mesh_password);
    MDF_ERROR_ASSERT(mconfig_chain_slave_deinit());
    //接收到手机蓝牙发过来的数据后, 必须释放 mconfig_chain_slave_init 中的资源
    MDF_ERROR_ASSERT(mconfig_bluifi_deinit()); //解除蓝牙网络配置初始化
    /*定义 wifi 配置及用户名, 密码的变量, 用来把蓝牙得到的 wifi, 密码, meshid..
     *信息转移到 mwifi_config 变量, 方便后面释放蓝牙 mconfig_data 变量 */
    mwifi_config_t mwifi_config = {0x0};
    /*从源蓝牙配置内存地址拷贝数据到目标 wifi 配置内存地址*/
    memcpy(&mwifi_config, &mconfig_data->config, sizeof(mwifi_config_t));
    mwifi_init_config_t init_config = MWIFI_INIT_CONFIG_DEFAULT(); //分配 mesh wifi 内存
    MDF_ERROR_ASSERT(mwifi_init(&init_config)); //初始化 mesh wifi
    MDF_ERROR_ASSERT(mwifi_set_config(&mwifi_config)); //传入 wifi 用户名和密码, mesh ID
    MDF_ERROR_ASSERT(mwifi_start()); //启动 mesh wifi 网络
    while(1)
    {
        sleep(1);
    }
}

```

```

I (79696) wifi:connected with metasia, aid = 4, channel 11, BSSID = 50:fa:84:3b:ce:7e
I (79700) wifi:security type: 4, phy: bgn, rssi: -62
I (79762) wifi:pm start, type: 0

I (79765) wifi:AP's beacon interval = 102400 us, DTIM period = 1
I (79769) mesh: [scan]new scanning time:600ms
E (79769) wifi:[beacon]new interval:300ms
I (79800) mesh: <nvs>write layer:1
D (79801) [mwifi, 131]: esp_mesh_event_cb event.id: 7
I (79801) [mwifi, 137]: Parent is connected
I (79803) [XZZ_mconfig, 10]: event_loop_cb, event: 7
D (79804) [mwifi, 131]: esp_mesh_event_cb event.id: 11
I (79809) [XZZ_mconfig, 18]: MDF_EVENT_PARENT_CONNECTED
I (79816) [mwifi, 234]: State represents: 0
I (79824) [XZZ_mconfig, 10]: event_loop_cb, event: 11

```

加入采用 mesh wifi 连接路由器的代码

连接成功

如果要 ble mesh 配网功能在线重复使用，记得释放内存

```
mwifi_init_config_t init_config = MWIFI_INIT_CONFIG_DEFAULT(); //分配 mesh wifi 内存
MDF_ERROR_ASSERT(mwifi_init(&init_config)); //初始化 mesh wifi
MDF_ERROR_ASSERT(mwifi_set_config(&mwifi_config)); //传入 wifi 用户名和密码, mesh ID
MDF_ERROR_ASSERT(mwifi_start()); //启动 mesh wifi 网络

MDF_FREE(mconfig_data); //拷贝完成之后就可以释放蓝牙配置信息，因为 WiFi 信息已经拷贝到 wifi 变量了，一定要启动 mesh wifi 后才能释放蓝牙配置的数据
MDF_ERROR_ASSERT(esp_bt_mem_release(ESP_BT_MODE_BLE)); //释放蓝牙 BLE 控制器和内存，一定要 MDF_FREE 之后执行，这里要注意，如果是在线重复启动蓝牙，这句要取消
while(1)
{
    sleep(1);
}
```

加入蓝牙后重点注意事项，分区表问题

```
E (48654) wifi:alloc eb len=148 type=2 fail, heap:480
E (48654) wifi:m f probe req l=0
E (49155) wifi:alloc eb len=148 type=2 fail, heap:480
E (49155) wifi:m f probe req l=0
```

突然运行到一半打印奇怪得字符

```
bort() was called at PC 0x40085d1b on core 0
ELF file SHA256: 3d9b20ec86180aa0c02944d2d7fd05eb62b2ffe0afef71f624b3a687277ed643
Backtrace: 0x4008e548:0x3ffffe810 0x4008e7e1:0x3ffffe830 0x40085d1b:0x3ffffe850 0x40085d3d:0x3ffffe870 0x40085e7d:0x3fff
bd0 0x401b772e:0x3fffec00 0x40133a4f:0x3fffec20 0x40133a57:0x3fffec40 0x40133ab3:0x3fffec70 0x40097cb6:0x3fffec90
E (12384) esp_core_dump_flash: Core dump flash config is corrupted! CRC=0x6522df69 instead of 0x0
Rebooting...
ets Jul 29 2019 12:21:46

rst:0xc (SW_CPU_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:6780
load:0x40078000,len:12072
load:0x40080400,len:6708
entry 0x40080778
```

突然 flash 报错

E (37263) lwip_arch: thread_sem_init: out of memory 内存不足
E (12384) esp_core_dump_flash: Core dump flash config is corrupted! CRC=0x6522df69 instead of 0x0

这些情况可能是因为分区表容量太小造成的。

FeedingDevice.bin	2020-10-17 17:05	BIN 文件	1,572 KB
大小:	1.53 MB		

我编译出来的应用程序超过 1.5M 太大了，可能超过分区表的范围。

xzzpri.bin	2020-10-17 10:28	BIN 文件	1,341 KB
大小:	1.30 MB	我单独测试蓝牙模块功能都占用了 1.3M	

何况项目还加入了 NVS 存储和其它功能，所以超过 1.5M 了

下面尝试修改分区表大小看看能否解决该问题

分区表的划分定义及操作方式 请查阅 [《ESP32_peripheral_UserGuide》](#) 外设文档

打开工程下的分区表文件 partitions.csv

#	Name,	Type,	SubType,	Offset,	Size,	Flags
	wifi,	data,	nvs,	0x9000,	16k	
	otadata,	data,	ota,	0xd000,	8k	
	phy_init,	data,	phy,	0xf000,	4k	
	ota_0,	app,	ota_0,	0x10000,	1920k	
	ota_1,	app,	ota_1,	,	1920k	
	nvs,	data,	nvs,	,	192K	

应用程序启动地址是从
10000 开始，
大小为 1920K 也就是
1.9M，不应该出问题呢？

#	Name,	Type,	SubType,	Offset,	Size,	Flags
	wifi,	data,	nvs,	0x9000,	16k	
	otadata,	data,	ota,	0xd000,	8k	
	phy_init,	data,	phy,	0xf000,	4k	
	ota_0,	app,	ota_0,	0x10000,	3M	
	ota_1,	app,	ota_1,	,	192k	
	nvs,	data,	nvs,	,	192K	

应用程序启动地址是从
10000 开始，
大小改为 3M

第 2 个应用程序备份位置
就得改小，不然编译的时
候容量不够

```
Partitions defined in 'E:\xzz\FeedingDevice\partitions.csv' occupy 5.1MB of flash (5373952 bytes) which does not fit in configured flash size 4MB.
make: *** [/d/xzz_ESP32/esp-mdf-v1.0/esp-idf/components/partition_table/Makefile.projbuild:61: /e/xzz/FeedingDevice/build/partitions.bin] Error 2
make: *** Waiting for unfinished jobs...
```

这就是两个应用程序分区超过本身 flash 4M 容量的大小，而报错，所以要将 ota_1 改成 192K 就可以顺利编译通过了。其实 ota_0 和 ota_1 都设置为 1920k 是没有问题的这里是我搞错了问题是出现在线程堆溢出，根项目有关。后面介绍

记住 修改 partitions.csv 之后一定要进行重新编译，因为除了下载 partitions.csv 到芯片以外，程序各模块启动区域也要遵循 partitions.csv 文件的分区。在《ESP32_peripheral_UserGuide》 外设文档有介绍。

<input checked="" type="checkbox"/> E:\xzz\FeedingDevice\build\bootloader\bootloader.bin	...	@	0x1000
<input checked="" type="checkbox"/> E:\xzz\FeedingDevice\build\FeedingDevice\firmware\ota_0\ota_0.ino.uf2	...	@	0x10000
<input checked="" type="checkbox"/> E:\xzz\FeedingDevice\build\partitions.bin	...	@	0x8000

下载软件地址设置必须和分区
表每个部分的起始地址一样，所
以这些启动程序的地址是可
以任意修改的，只要保证下载软件
地址和分区表地址一致就 OK

程序运行正常了。

注意：设备如果已经处于 STA 模式，正在与路由器通讯的情况下。如果进行蓝牙配网，执行了一些重复的代码会造成设备崩溃重启，下面列出解决方案！！

正在运行的程序

主程序已经联网，正在运行中，这时候我

去蓝牙配网，如果执行以下这些函数：



重复分配 wifi 配置

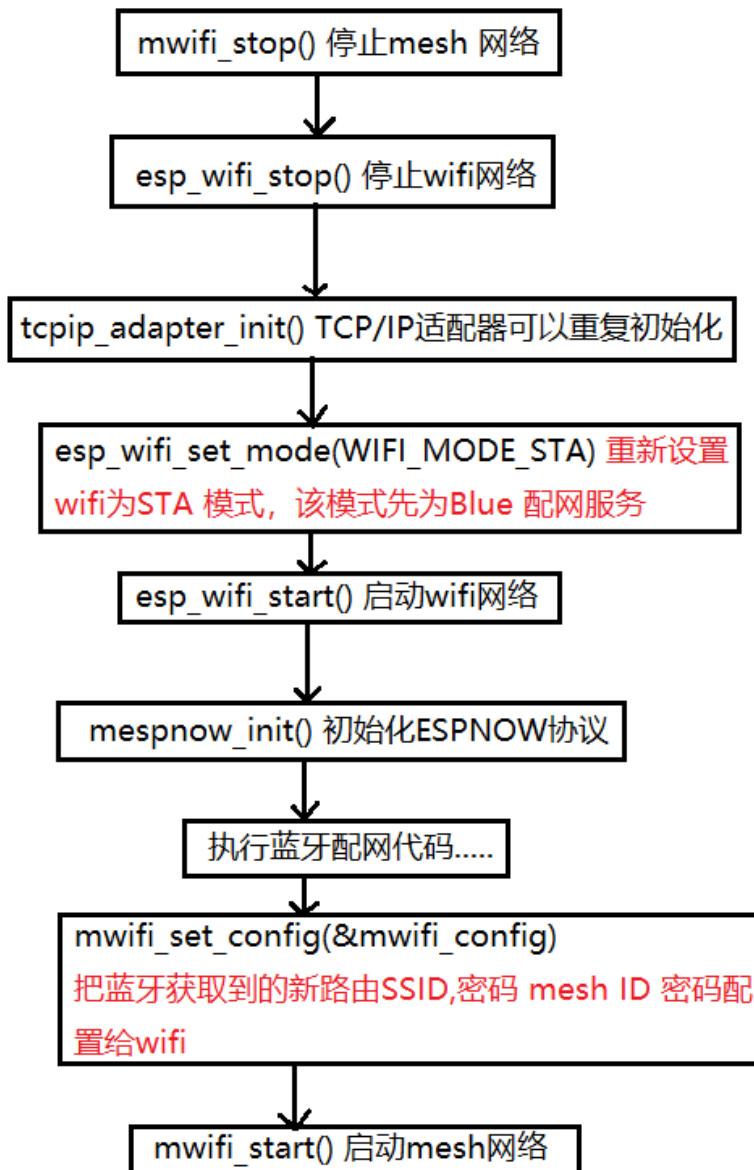
重复初始化(系统崩溃)

重复注册事件回调(系统崩溃)

重复注册 mesh 网络事件回调(系统崩溃)

所以在主程序已经连接网络运行的过程中，如果要进行蓝牙 mesh 方式配网，那么 wifi STA 模式的重启过程要简化

蓝牙 wifi mesh 配网简化流程



其余wifi配置功能之前程序运行就配置好的，就不需要去重新配置

```
MDF_ERROR_ASSERT(mwifi_stop()); //停止 wifi mesh 网络
MDF_ERROR_ASSERT(esp_wifi_stop()); //wifi STA 模式停止

tcpip_adapter_init(); //重复初始化 TCP/IP 适配器
MDF_ERROR_ASSERT(esp_wifi_set_storage(WIFI_STORAGE_FLASH));
MDF_ERROR_ASSERT(esp_wifi_set_mode(WIFI_MODE_STA)); //wifi 设置为 STA 模式启动
MDF_ERROR_ASSERT(esp_wifi_set_ps(WIFI_PS_NONE));
MDF_ERROR_ASSERT(esp_mesh_set_6m_rate(false));
MDF_ERROR_ASSERT(esp_wifi_start()); //启动 wifi STA 模式
MDF_ERROR_ASSERT(mespnow_init()); //初始化 ESPNOW
char BlueName[28] = {0x0}; //存放蓝牙设备名称
uint8_t sta_mac[6] = {0};

MDF_ERROR_ASSERT(esp_wifi_get_mac(ESP_IF_WIFI_STA, sta_mac)); //为了防止蓝牙设备重名，获取 MAC 地址，区分设备
sprintf(BlueName, "Feed-BlueMcfg-WIFI-MESH_%02x%02x", sta_mac[4], sta_mac[5]); //在蓝牙设备名尾部用 MAC 地址区分
/*蓝牙信息配置，可以自定义蓝牙设备名称*/
mconfig_bluifi_config_t bluifi_config = {
    .tid = 1,
    .company_id = MCOMMON_ESPRESSIF_ID, //公司标识
};

strncpy(bluifi_config.name, BlueName, sizeof(bluifi_config.name) - 1); //将自定义的蓝牙设备名称放入 bluifi 配置
MDF_ERROR_ASSERT(mconfig_bluifi_init(&bluifi_config)); //初始化蓝牙网络配对
MDF_ERROR_ASSERT(mconfig_chain_slave_init()); //本机为从机，初始化蓝牙配置的网络
```

```

mconfig_data_t *mconfig_data = NULL;
MDF_ERROR_ASSERT(mconfig_queue_read(&mconfig_data, portMAX_DELAY)); //等待读取网络配置信息
MDF_LOGI("BLE Mconfig data success"); //蓝牙网络配置数据接收成功
MDF_ERROR_ASSERT(mconfig_chain_slave_deinit());
MDF_ERROR_ASSERT(mconfig_bluifi_deinit());
mwifi_config_t mwifi_config = {0x0};
memcpy(&mwifi_config, &mconfig_data->config, sizeof(mwifi_config_t)); //BLE 数据内存拷贝到 wifi 内存
MDF_LOGI("Blue mconfig router SSID : %s", mwifi_config.router_ssid);
MDF_LOGI("Blue mconfig router PASSWD : %s", mwifi_config.router_password);
MDF_LOGI("Blue mconfig mesh ID : %x : %x : %x : %x : %x : %x",
        mwifi_config.mesh_id[0], mwifi_config.mesh_id[1], mwifi_config.mesh_id[2], mwifi_config.mesh_id[3],
        mwifi_config.mesh_id[4], mwifi_config.mesh_id[5]);
MDF_LOGI("Blue mconfig mesh PASSWD : %s", mwifi_config.mesh_password);
MDF_FREE(mconfig_data); //拷贝完成之后就可以释放蓝牙配置信息，因为 WiFi 信息已经拷贝到 wifi 变量
MDF_ERROR_ASSERT(esp_bt_mem_release(ESP_BT_MODE_BLE));
MDF_ERROR_ASSERT(mwifi_set_config(&mwifi_config));
MDF_ERROR_ASSERT(mwifi_start());

```

```

I (51415) mesh: 388<stop>
W (51416) mesh: [mesh.c.391] <stop>vendor ie
W (51419) mesh: [mesh.c.394] <stop>send block
W (51423) mesh: [mesh.c.397] <stop>rx task
I (51427) mesh: 1804<flush>up:0, up-be:0
W (51430) mesh: [mesh_schedule.c.2408] <break>
I (51435) mesh: 1804<flush>up:0, up-be:0
W (51438) mesh: [mesh_schedule.c.2726] <done>MESH_STOP_EVENT_BIT_XON
W (51444) mesh: [mesh_schedule.c.2727] <stop>xon
I (51448) mesh: 1785<flush>xon:0, mgmt:0, bcast:0, down:0, donw-be:0
W (51454) mesh: [mesh_schedule.c.1854] <break>
I (51459) mesh: 1785<flush>xon:0, mgmt:0, bcast:0, down:0, donw-be:0
W (51465) mesh: [mesh.c.400] <stop>tx task
I (51469) wifi:state: run -> init (0)
I (51473) wifi:pm stop, total sleep time: 0 us / 40316076 us

I (51478) wifi:new:<6,0>, old:<6,2>, ap:<6,2>, sta:<6,2>, prof:6
W (51489) mesh: [mesh.c.403] <stop>nwk task
W (51490) mesh: [mesh.c.406] <stop>wifi event
W (51491) mesh: [mesh.c.410] <stop>route
W (51495) mesh: [mesh_schedule.c.2167] <done>MESH_STOP_EVENT_BIT_TX
W (51501) mesh: [mesh_schedule.c.2168] <stop>tx
W (51505) [ThreadWifi, 203]: socket connect, ret: -1, ip: 0.0.0.0, port: 0
W (51506) mesh: [mesh.c.359] <done>MESH_STOP_EVENT_BIT_SELF
W (51518) mesh: [mesh.c.415] <stop>free mesh_myself_mbox
W (51523) mesh: [mesh.c.420] <stop>free mesh_tcpip_mbox
W (51528) mesh: [mesh.c.423] <stop>is_receiving:0
W (51532) mesh: [mesh.c.434] <stop>free mesh_xmit_state_mbox
W (51538) mesh: [mesh.c.447] <stop>nvs task
I (51544) wifi:Total power save buffer number: 16
I (51549) mesh: 489<end>
I (51550) [mwifi, 208]: MESH is stopped
I (51553) [ThreadWifi, 819]: event loop_cb, event: 1

```

关闭 wifi 成功

```

I (76906) wifi:new:<6,0>, old:<6,2>, ap:<255,255>, sta:<6,2>, prof:1
I (76914) [mconfig_bluifi, 544]: BLUIFI deinit finish
W (76920) BT_APPL: bta_dm_disable BTA_DISABLE_DELAY set to 200 ms
I (77027) ThreadMain: XZZ totalsec = 340818864
I (77028) ThreadMain: =====
I (77134) [ThreadWifi, 819]: event loop_cb, event: 8705
    (77135) [feed_blue_mconfig, 61]: Feed Blue mconfig router SSID : metasia
    (77137) [feed_blue_mconfig, 62]: Feed Blue mconfig router PASSWD : metasia0902
    (77145) [feed_blue_mconfig, 65]: Feed Blue mconfig mesh ID : 50 : fa : 84 : 3b : ce : 7e
    (77154) [feed_blue_mconfig, 66]: Feed Blue mconfig mesh PASSWD : 123456789
I (77162) mesh: [scan]new scanning time:300ms
E (77165) wifi:[beacon]interval:100ms
I (77169) wifi:mode : sta (10:52:lc:88:46:20) + softAP (10:52:lc:88:46:21)
I (77177) wifi:Total power save buffer number: 16
I (77183) mesh: <nvs>read layer:1
I (77184) mesh: <nvs>read assoc:0, err:0x1102
E (77188) wifi:[beacon]interval:100ms
I (78181) wifi:Total power save buffer number: 16
I (79187) wifi:Set ps type: 0

I (79192) wifi:mode : sta (10:52:lc:88:46:20)
I (79195) mesh: <MESH_NWK_LOOK_FOR_NETWORK>need_scan:0x1, need_scan_router:0x0, look_for_n

```

获得 wifi ssid 密码和 mesh id 密码

```

I (85023) wifi:connected with metasia, aid = 16, channel 6, BW20, bssid = 50:fa:84:3b:ce:7e
I (85026) wifi:security type: 4, phy: bgn, rss: -67
I (85031) wifi:pm start, type: 0

```

联网成功，其实这里就已经完成了。但是如果之前的程序运行了 SNTP 那么还会出现崩溃。

sntp 服务器下一章有介绍，这里只讲解怎么解决问题

```
assertion "Operating mode must not be set while SNTP client is running" failed:  
  sntp_setoperatingmode  
  abort() was called at PC 0x4014218f on core 0
```

汇报 SNTP 服务运行错误

Operating mode must not be set while SNTP client is running

Sntp 客户端运行时，不能操作 sntp。

```
I (86002) example: Connecting to WiFi and getting time over NTP.
```

主要问题就在， wifi 连接路由器成功后，系统又去重新获取 NTP 服务器的时间，但是现在 NTP 在本地运行起的。所以需要先关闭 ntp 客户端，再重新获取 NTP 服务器时间。

sntp_stop() //SNTP 客户端关闭

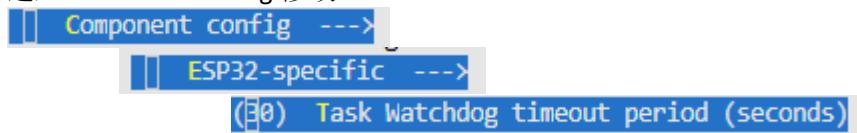
```
MDF_ERROR_ASSERT(mwifi_set_config(&mwifi_config));  
MDF_ERROR_ASSERT(mwifi_start());  
  
sntp_stop(); //重新配网连接 wifi 的时候先停止 SNTP 服务器
```

根据以上代码，再重新启动 mesh wifi 的时候关闭 SNTP 客户端

蓝牙配网错误 Task watchdog got triggered. The following tasks did not reset the watchdog in time 这是因为有程序阻塞了其它线程，导致其它进程无法喂看门狗，所以看门狗重启。

```
I (35032) [mconfig_bleif, 882]: start ble advertising  
E (36837) task_wdt: Task watchdog got triggered. The following tasks did not reset the watchdog in time:  
E (36837) task_wdt: - IDLE0 (CPU 0)  
E (36837) task_wdt: Tasks currently running:  
E (36837) task_wdt: CPU 0: mconfig_rsa_gen  
E (36837) task_wdt: Aborting.
```

进入 make menuconfig 修改



FreeRTOS 空闲任务从 10 秒改到 30 秒喂狗一次

其实这不是主要原因

mconfig_queue_read 阻塞任务函数修改

```
MDF_ERROR_ASSERT(mconfig_queue_read(&mconfig_data, portMAX_DELAY));
```

将最大阻塞时间改为可设置的时间

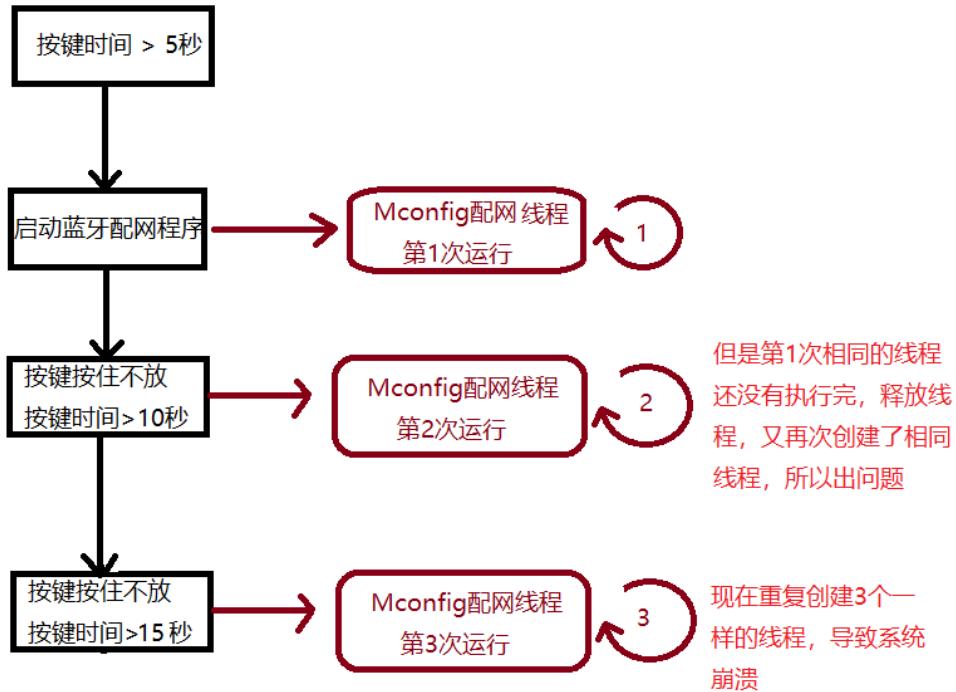
```
do  
{  
    vTaskDelay(1000 / portTICK_RATE_MS);  
    MDF_LOGI("XZZ mconfig_queue_read TIMEOUT");  
}  
while(mconfig_queue_read(&mconfig_data, 100) == MDF_ERR_TIMEOUT);
```

改成 100ms 阻塞时间，没有收到手机 APP 发来的蓝牙数据包，就跳出等待，进行延时，将时间片分配给其它任务喂看门狗。循环等待 APP 发来的蓝牙数据包。

看似有所改善，但是 mconfig_queue_read 阻塞方式也不是主要原因

esp_bt_mem_release 函数到底需不要，解决蓝牙重复配网的方法完善

第 1 个问题，根据我的项目需求，我导致同一个蓝牙配网任务在没执行完删除任务的情况下重复创建。



后面问题无限循环，按键一直再按，无限创建相同线程，系统崩溃

为了解决线程没执行完重复创建的问题，一定要给线程加标志位。

```
void Start_BlueMconfig(void)
{
    xTaskCreate(ThreadBlueMconfig, "ThreadBlueMconfig", 4 * 1024,NULL, 11, &BlueMconfig_Handle); // 创建蓝牙配网线程，优先级 11
    configASSERT( Feed_BlueMconfig_Handle );
}
```

这就是未加标志位造成的 Start_BlueMconfig 重复执行，在第 1 次 Start_BlueMconfig 线程还没执行完的情况下，再次创建 Start_BlueMconfig 线程，无限循环创建，导致系统崩溃。

```
void Start_BlueMconfig(void)
{
    if( Feed_BlueMconfig_Handle == NULL )
    {
        Feed_Blue_Mconfig_FLAG = true; // 加入线程创建标志位

        xTaskCreate(ThreadBlueMconfig, "ThreadBlueMconfig", 4 * 1024,NULL, 11, &BlueMconfig_Handle);
        configASSERT( Feed_BlueMconfig_Handle );
    }
    else if (Feed_Blue_Mconfig_FLAG == false)
    {
        Feed_Blue_Mconfig_FLAG = true; // 加入线程创建标志位

        xTaskCreate(ThreadBlueMconfig, "ThreadBlueMconfig", 4 * 1024,NULL, 11, &BlueMconfig_Handle);
        configASSERT( BlueMconfig_Handle );
    }
    else
    {
    }
}
```

加入线程创建标志位，线程问题解决。

这里再提一下线程句柄断言问题 configASSERT

如果使用 configASSERT 来判断线程句柄，那么线程也要加入句柄地址，不然会出现创建线程断言错误

```
xTaskCreate(ThreadBlueMconfig, "ThreadBlueMconfig", 4 * 1024,NULL, 11, NULL);
```

改为 xTaskCreate(ThreadBlueMconfig, "ThreadBlueMconfig", 4 * 1024,NULL, 11, &BlueMconfig_Handle);

启动蓝牙配网线程，以下代码是可以设备在线重复执行蓝牙配网功能的程序

```
void ThreadBlueMconfig(void *arg)
{
    MDF_ERROR_ASSERT(mwifi_stop()); //停止 wifi mesh 网络
    MDF_ERROR_ASSERT(esp_wifi_stop()); //wifi STA 模式停止
    和流程一样,先关闭正在执行的 wifi

    tcpip_adapter_init(); //重复初始化 TCP/IP 适配器
    MDF_ERROR_ASSERT(esp_wifi_set_storage(WIFI_STORAGE_FLASH));
    MDF_ERROR_ASSERT(esp_wifi_set_mode(WIFI_MODE_STA)); //wifi 设置为 STA 模式启动
    MDF_ERROR_ASSERT(esp_wifi_set_ps(WIFI_PS_NONE));
    MDF_ERROR_ASSERT(esp_mesh_set_6m_rate(false));
    MDF_ERROR_ASSERT(esp_wifi_start()); //启动 wifi STA 模式
    MDF_ERROR_ASSERT(mespnow_init()); //初始化 ESP NOW
    char BlueName[128] = {0x0}; //存放蓝牙设备名称
    uint8_t sta_mac[6] = {0};
    MDF_ERROR_ASSERT(esp_wifi_get_mac(ESP_IF_WIFI_STA, sta_mac)); //为了防止蓝牙设备重名, 获取 MAC 地址, 区分设备
    sprintf(BlueName, "Feed-BlueMcfg-WIFI-MESH_%02x%02x", sta_mac[4], sta_mac[5]); //在蓝牙设备名尾部用 MAC 地址区分
    /*蓝牙信息配置, 可以自定义蓝牙设备名称*/
    mconfig_bluifi_config_t bluifi_config = {
        .tid = 1,
        .company_id = MCOMMON_ESPRESSIF_ID, //公司标识
    };
    strncpy(bluifi_config.name, BlueName, sizeof(bluifi_config.name) - 1); //将自定义的蓝牙设备名称放入 bluifi 配置
    MDF_ERROR_ASSERT(mconfig_bluifi_init(&bluifi_config)); //初始化蓝牙网络配置
    MDF_ERROR_ASSERT(mconfig_chain_slave_init()); //本机为从机, 初始化蓝牙配置的网络
    mconfig_data_t *mconfig_data = NULL;
    MDF_ERROR_ASSERT(mconfig_queue_read(&mconfig_data, portMAX_DELAY)); //阻塞死等待读取网络配置信息
    MDF_ERROR_ASSERT(mconfig_chain_slave_deinit());
    MDF_ERROR_ASSERT(mconfig_bluifi_deinit()); //清除本次的蓝牙网络配置, 方便下一次再重新初始化蓝牙
    mwifi_config_t mwifi_config = {0x0};
    memcpy(&mwifi_config, &mconfig_data->config, sizeof(mwifi_config_t)); //BLE 数据内存拷贝到 wifi 内存
    MDF_LOGI("Feed Blue mconfig router SSID : %s", mwifi_config.router_ssid);
    MDF_LOGI("Feed Blue mconfig router PASSWD : %s", mwifi_config.router_password);
    MDF_LOGI("Feed Blue mconfig mesh ID : %x : %x : %x : %x : %x : %x",
            mwifi_config.mesh_id[0], mwifi_config.mesh_id[1], mwifi_config.mesh_id[2], mwifi_config.mesh_id[3],
            mwifi_config.mesh_id[4], mwifi_config.mesh_id[5]);
    MDF_LOGI("Feed Blue mconfig mesh PASSWD : %s", mwifi_config.mesh_password);
    MDF_FREE(mconfig_data); //拷贝完成之后就可以释放蓝牙配置信息, 因为 WiFi 信息已经拷贝到 wifi 变量
    // MDF_ERROR_ASSERT(esp_bt_mem_release(ESP_BT_MODE_BLE)); //屏蔽该功能很关键, 下面会讲
    // mwifi_init_config_t init_config = MWIFI_INIT_CONFIG_DEFAULT();
    // MDF_ERROR_ASSERT(mwifi_init(&init_config)); //取消 init 因为设备前期启动已经初始化 mesh 一次了
    MDF_ERROR_ASSERT(mwifi_set_config(&mwifi_config));
    MDF_ERROR_ASSERT(mwifi_start());
    sntp_stop(); //重新配网连接 wifi 的时候先停止 SNTP 服务器
    Feed_Blue_Mconfig_FLAG = false;
    MDF_LOGI("delete ThreadFeedBlueMconfig");
    vTaskDelete(Feed_BlueMconfig_Handle); //删除当前线程
}
```

你可以无限成功配网了

```
esp_err_t esp_bt_mem_release(esp_bt_mode_t mode) //蓝牙内存释放
```

```
MDF_FREE(mconfig_data); //拷贝完成之后就可以释放蓝牙配置信息, 因为 WiFi 信息已经拷贝到 wifi 变量

MDF_ERROR_ASSERT(esp_bt_mem_release(ESP_BT_MODE_BLE)); //如果我打开释放蓝牙内存功能

MDF_ERROR_ASSERT(mwifi_set_config(&mwifi_config));
MDF_ERROR_ASSERT(mwifi_start());

I (34583) system_api: Base MAC address is not set, read default base MAC address from BLK0 of EFUSE
I (34594) [ThreadWifi, 819]: event_loop_cb, event: 8449
I (34597) [mconfig_bluifi, 882]: start ble advertising
I (36895) [mconfig_chain, 361]: Generate RSA public and private keys
I (74254) mesh: [SCAN][ch:1]AP:10, other[ID:0, RD:0], MAP:0, idle:0, candidate:1, root:0, topMAP:0[c:0,i:1][50:fa:84:3b:ce:7e]router found<>
I (74256) mesh: 1212[SCAN]init rc[10:52:1c:88:46:21, -69], mine:0, voter:0
I (74263) mesh: [SCAN:2/10]rc[128][10:52:1c:88:46:21, -68], self[10:52:1c:88:46:20, -69, reason:0, votes:1, idle][mine:1, voter:1(1.00)percent:0.90][
I (74580) mesh: [SCAN][ch:1]AP:10, other[ID:0, RD:0], MAP:0, idle:0, candidate:1, root:0, topMAP:0[c:0,i:1][50:fa:84:3b:ce:7e]router found<>
I (74582) mesh: 1212[SCAN]init rc[10:52:1c:88:46:21, -66], mine:0, voter:0
I (74589) mesh: [SCAN:3/10]rc[128][10:52:1c:88:46:21, -68], self[10:52:1c:88:46:20, -66, reason:0, votes:1, idle][mine:1, voter:1(1.00)percent:0.90][
```

设备开机第 1 次蓝牙配网很成功

设备不关机的情况下进行第 2 次配网，

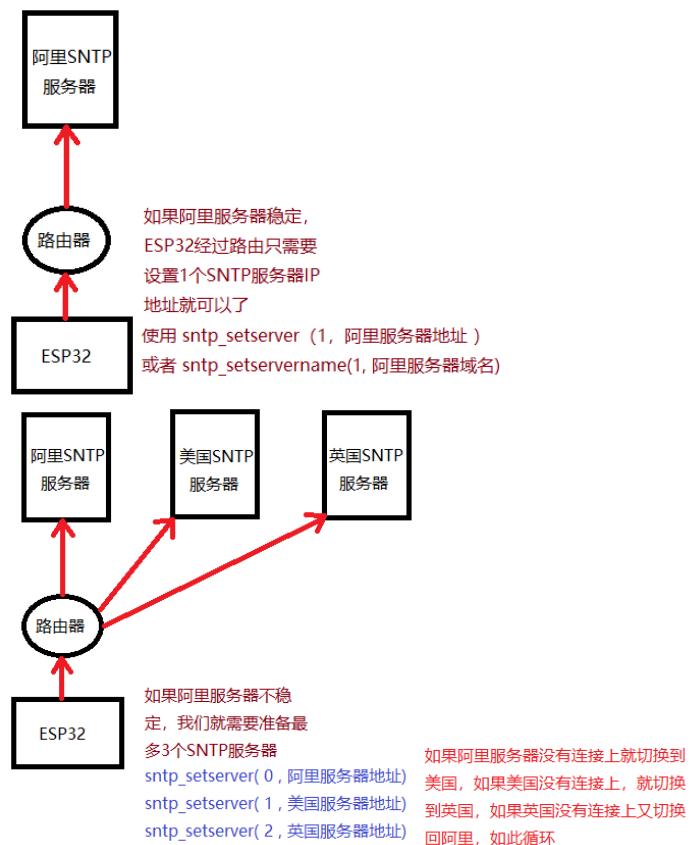
报错 assertion "0 && "mconfig_bt_if_init(&bt_if_config)"" failed

```
W (139292) [mconfig_bt_if, 918]: <ESP_ERR_INVALID_STATE> Initialize bt controller
W (139293) [feed_blue mconfig, 50]: <ESP_ERR_INVALID_STATE> MDF_ERROR_ASSERT failed
assertion "0 && "mconfig_bt_if_init(&bt_if_config)"" failed: file "E:/xzz/FeedingDe
abort() was called at PC 0x401421f3 on core 0
```

直接报初始化蓝牙错误。

这是因为 `esp_bt_mem_release` 执行之后是不可逆的，所以执行 `esp_bt_mem_release` 不止释放了蓝牙内存，而且还永久关闭了蓝牙配网功能。在线重复蓝牙配网应用中，取消使用该函数。

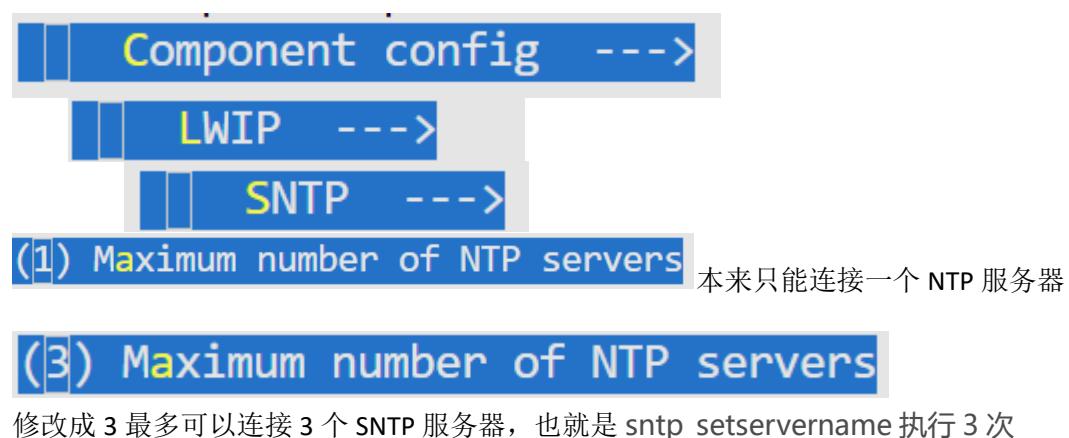
SNTP 服务器配置，获取网络标准时间



```
#include "lwip/apps/sntp.h" 包含该头文件
```

```
void sntp_setserver(unsigned char idx, ip_addr_t *addr) //通过 IP 地址设置 SNTP 服务器  
idx: SNTP 服务器编号, 最多支持 3 个 SNTP 服务器 (0~2) ; 0 号为主服务器, 1 号和 2 号为备用服务器  
*addr: SNTP 服务器 IP 地址; 用户需自行写入, 传入的是合法 SNTP 服务器
```

```
void sntp_setservername(unsigned char idx, char *server) //通过域名设置 SNTP 服务器  
idx: SNTP 服务器编号, 最多支持 3 个 SNTP 服务器 (0~2) ; 0 号为主服务器, 1 号和 2 号为备用服务器  
*addr: SNTP 服务器域名地址(使用域名容易记); 用户需自行写入, 传入的是合法 SNTP 服务器
```



第 1: 先连接上 wifi 网络，外网打开

```
static esp_err_t event_handler(void *ctx, system_event_t *event)
{
    switch (event->event_id) {
        case SYSTEM_EVENT_STA_START:
            esp_wifi_connect(); //连接热点
            break;
        case SYSTEM_EVENT_STA_GOT_IP:
            break;
        case SYSTEM_EVENT_STA_CONNECTED:
            ESP_LOGI(TAG, "wifi connect success");
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            esp_wifi_connect(); //连接热点
            break;
        default:
            break;
    }
    return ESP_OK;
}

void app_main()
{
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) +
        ESP_ERROR_CHECK(nvs_flash_erase());
    ret = nvs_flash_init();
}
ESP_ERROR_CHECK( ret );

tcpip_adapter_init(); //初始化TCP/IP适配层
ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
ESP_ERROR_CHECK( esp_wifi_init(&cfg) ); //wifi使用之前，必须执行这段初始化

wifi_config_t wifi_config = {
    .sta = {
        .ssid = "OPPO A11", //要连接设备SSID
        .password = "12345678", //要连接设备密码
    },
};

ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA) ); //wifi启动为STA模式
ESP_ERROR_CHECK( esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config) );
ESP_ERROR_CHECK( esp_wifi_start() ); //启动wifi
}
```

wifi 连接成功

第 2: 获取 SNTP 服务器时间

```
#include <lwip/apps/snntp.h> //加入 SNTP 头文件
void snntp_setoperatingmode(u8_t operating_mode) //设置 SNTP 工作模式
operating_mode: SNTP_OPMODE_POLL 单播模式
                SNTP_OPMODE_LISTENONLY 广播模式
snntp_init() //SNTP 客户端初始化函数
setenv() //时区设置
tzset(); //设置时间环境变量
    snntp_setoperatingmode(SNTP_OPMODE_POLL);
    snntp_setservername(0, "ntp1.aliyun.com"); //阿里云
    snntp_setservername(1, "210.72.145.44"); // 国家授时中心服务器 IP 地址
    snntp_setservername(2, "1.cn.pool.ntp.org");
    snntp_init();

    setenv("TZ", "CST-8", 1);
    tzset();
```

```
time(time_t *timep) //获取 ESP32 本地时间，就算网断了也能一直按照 CPU 时钟节拍获取  
*timep: 传入 long 变量地址
```

```
struct tm *localtime_r(const time_t *timep, struct tm *result)
```

```
//是可重入函数，线程安全，将 timep 为 long 形的数据解析成年月日时分秒
```

```
*timep: 传入获取当前时间的 long 形数据地址
```

```
*result: 传入 struct tm 解析成年月日时分秒的结构体
```

```
time_t now = 0;
struct tm timeinfo = { 0 };

while(1){

    time(&now);
    localtime_r(&now, &timeinfo);
    ESP_LOGI(TAG, "-----current time: %d:%d:%d:%d:%d:%d:%d",
    timeinfo.tm_yday,
    timeinfo.tm_wday, timeinfo.tm_year,
    timeinfo.tm_mon, timeinfo.tm_mday,
    timeinfo.tm_hour, timeinfo.tm_min,
    timeinfo.tm_sec);

    sleep(1);
}

I (2742) xxxzzzzz: Wifi connect success
I (2902) wifi:AP's beacon interval = 204800 us, DTIM period = 2
I (3372) xxxzzzzz: -----current time: 0:4:70:0:1:8:0:3
I (3692) event: sta ip: 192.168.43.238, mask: 255.255.255.0, gw: 192.168.43.1
I (4372) xxxzzzzz: -----current time: 0:4:70:0:1:8:0:4
I (5372) xxxzzzzz: -----current time: 0:4:70:0:1:8:0:5
I (6372) xxxzzzzz: -----current time: 0:4:70:0:1:8:0:6
I (7372) xxxzzzzz: -----current time: 261:5:120:8:18:23:12:9
I (8372) xxxzzzzz: -----current time: 261:5:120:8:18:23:12:10
I (9372) xxxzzzzz: -----current time: 261:5:120:8:18:23:12:11
I (10372) xxxzzzzz: -----current time: 261:5:120:8:18:23:12:12
I (11372) xxxzzzzz: -----current time: 261:5:120:8:18:23:12:13
I (12372) xxxzzzzz: -----current time: 261:5:120:8:18:23:12:14
I (13372) xxxzzzzz: -----current time: 261:5:120:8:18:23:12:15
I (14372) xxxzzzzz: -----current time: 261:5:120:8:18:23:12:16
I (15372) xxxzzzzz: -----current time: 261:5:120:8:18:23:12:17
I (16372) xxxzzzzz: -----current time: 261:5:120:8:18:23:12:18
I (17372) xxxzzzzz: -----current time: 261:5:120:8:18:23:12:19
```

连接网络之后不是
马上获取 SNTP 时间

而是经
过几秒
才能获
取

这里的年是 120，也就是系统计算的 1900 年
到当前时间年份的差值

月份是从 0 开始计算

所以这里的年需要自己 $120+1900 = 2020$ 才正确。

而且系统启动后不是马上就能与 SNTP 服务器同步时间，需要延时几秒再获取 SNTP 时间。
或者参考下面的 NTP 时间同步成功回调函数来解决。

```
size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr)
//根据 format 中定义的格式化规则，格式化结构 timeptr 表示的时间，并把它存储在 str 中。
如果你不需要把时间的年月日时分秒做成一个 long 型传递，你就可以直接用 strftime 显示，方便些。
```

*str: 得到时间字符后，将其放入 str 指针变量，方便后面显示

*format: 指定某种格式化方式，格式化*timeptr 里面的数据，然自动传递给 str

```
hile(1){
    time(&now);
    localtime_r(&now, &timeinfo);

    if (timeinfo.tm_year < (2016 - 1900)) {
        ESP_LOGE(TAG, "The current date/time error");
    } else {
        strftime(strftime_datebuf, sizeof(strftime_datebuf), "%F", &timeinfo); //取出日期/年月日
        strftime(strftime_timebuf, sizeof(strftime_timebuf), "%T", &timeinfo); //取出时间/时分秒
    }

    ESP_LOGI(TAG, "---LOGI---current year month day: %s ", strftime_datebuf); //LOGU 输出
    ESP_LOGI(TAG, "---LOGI---hours min sec day: %s ", strftime_timebuf);
    printf("----printf--current year month day: %s\n ", strftime_datebuf); //printf 输出也可以
    printf("----printf--current hours min sec: %s\n ", strftime_timebuf);

    sleep(1);
}
```

```
口$..-printf--current year month day: 2
----printf--current hours min sec: I
E (5432) xxxzzzzz: The current date/time error
口$.5432) xxxzzzzz: ---LOGI---current year month day: 2
I (5432) xxxzzzzz: ---LOGI---hours min sec day: I
口$..-printf--current year month day: 2
----printf--current hours min sec: I
E (6442) xxxzzzzz: The current date/time error
口$.6442) xxxzzzzz: ---LOGI---current year month day: 2
I (6442) xxxzzzzz: ---LOGI---hours min sec day: I
口$..-printf--current year month day: 2
----printf--current hours min sec: I
E (7452) xxxzzzzz: The current date/time error
口$.7452) xxxzzzzz: ---LOGI---current year month day: 2
I (7452) xxxzzzzz: ---LOGI---hours min sec day: I
口$..-printf--current year month day: 2
----printf--current hours min sec: I
I (8462) xxxzzzzz: ---LOGI---current year month day: 2020-09-18
I (8462) xxxzzzzz: ---LOGI---hours min sec day: 23:32:30
----printf--current year month day: 2020-09-18
----printf--current hours min sec: 23:32:30
I (9472) xxxzzzzz: ---LOGI---current year month day: 2020-09-18
I (9472) xxxzzzzz: ---LOGI---hours min sec day: 23:32:31
----printf--current year month day: 2020-09-18
----printf--current hours min sec: 23:32:31
I (10482) xxxzzzzz: ---LOGI---current year month day: 2020-09-18
I (10482) xxxzzzzz: ---LOGI---hours min sec day: 23:32:32
```

联网后本地没有同步到 SNTP 服务器，打印乱码，而且报错

后面同步到 SNTP 服务器，打印正常

以上都是正常获取 NTP 服务器同步时间的情况，如果获取 NTP 服务器时间出现问题我如何检测情况？

```
void sntp_set_time_sync_notification_cb(sntp_sync_time_cb_t callback)
//设置 NTP 同步回调函数，在函数里面证明获取 NTP 服务时间是否正常
```

```
typedef void (*sntp_sync_time_cb_t)(struct timeval *tv) //事件回调函数格式
```

```
sntp_sync_status_t sntp_get_sync_status(void) //获取 SNTP 服务同步是否更新完成
```

```
sntp_sync_status_t 更新完成返回 SNTP_SYNC_STATUS_COMPLETED  
如果更新尚未完成 返回 SNTP_SYNC_STATUS_RESET
```

```
void sntp_setoperatingmode(u8_t operating_mode) //设置 SNTP 模式
```

```
operating_mode: SNTP_OPMODE_POLL 单播模式
```

```
void XZZ_sntp_sync_time_cb_t(struct timeval *tv) //SNTP 服务器连接成功会执行这个回调
{
    ESP_LOGI(TAG, "SNTP server sync success");
}

void app_main()
{
    esp_err_t err = nvs_flash_init(); //记住在做 wifi 功能时，一定要对 nvs_flash 初始化，不然板子会一直重启
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        // NVS partition was truncated and needs to be erased
        // Retry nvs_flash_init
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );
    tcpip_adapter_init(); //初始化 TCP/IP 适配层 不然启动 wifi 模式会导致模块不停重启
    ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT(); //这个函数是常规操作，用来给 wifi 初始化赋值
    ESP_ERROR_CHECK( esp_wifi_init(&cfg) ); //wifi 使用之前，必须执行这段初始化
    wifi_config_t wifi_config =
    {
        .sta = {
            .ssid = "metasia",
            .password = "metasia0902",
        },
    };
    ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA) ); //wifi 启动为 STA 模式
    ESP_ERROR_CHECK( esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config) );
    ESP_ERROR_CHECK( esp_wifi_start() );//启动 wifi
    sntp_setoperatingmode(SNTP_OPMODE_POLL); //单播模式
    sntp_setservername(0, "cn.ntp.org.cn"); //设置访问服务器 中国提供商
    sntp_set_time_sync_notification_cb(&XZZ_sntp_sync_time_cb_t); //设置事件回调函数
    sntp_init();
    setenv("TZ", "CST-8", 1);
    tzset();
    time_t now = 0;
    struct tm timeinfo = {0};
    while(1){
        time(&now);
        localtime_r(&now,&timeinfo);
        ESP_LOGI(TAG,"----current Time = %d : %d : %d : %d : %d",timeinfo.tm_year,
        timeinfo.tm_mon,timeinfo.tm_mday,timeinfo.tm_hour,timeinfo.tm_min,timeinfo.tm_sec);

        sleep(1);
    }
}
```

SNTP 同步成
功会执行

增加 SNTP 服
务器回调

开机没同步成功不
会执行回调

同步成功之后执行
回调

可以在同步成功之后
的回调中，做一些时间
相关的事，比如 mesh
广播时间之类的

还有一种阻塞方式同步 SNTP，程序卡死在 SNTP 位置，等待 SNTP 同步完成，程序再运行

```
sntp_setoperatingmode(SNTP_OPMODE_POLL); //单播模式
sntp_setservername(0, "cn.ntp.org.cn"); //设置访问服务器 中国提供商
sntp_set_time_sync_notification_cb(&XZZ_sntp_sync_time_cb_t); //设置时间回调函数
sntp_init();
setenv("TZ", "CST-8", 1);
tzset();

time_t now = 0;
struct tm timeinfo = {0};
while (sntp_get_sync_status() == SNTP_SYNC_STATUS_RESET); //如果 SNTP 同步未完成就
卡在这里
while(1){
    time(&now);
    localtime_r(&now,&timeinfo);
    ESP_LOGI(TAG,"----current Time = %d : %d : %d : %d : %d",
year,
        timeinfo.tm_mon,timeinfo.tm_mday,timeinfo.tm_hour,timeinfo.tm_min,timeinfo.
tm_sec);
    sleep(1);
}
```

代码中加入阻塞同步，程序卡在这里

时间没同步，就不会进入 while(1) 循环

```
I (512) xxxzzzzz: connect AP success
I (531) wifi:AP's beacon interval = 102400 us, DTIM period = 1
I (1233) event: sta ip: 192.168.1.140, mask: 255.255.255.0, gw: 192.168.1.1
I (6378) xxxzzzzz: SNTP server sync success
I (6378) xxxzzzzz: ----current Time = 120 : 8 : 26 : 15 : 13 : 21
I (7379) xxxzzzzz: ----current Time = 120 : 8 : 26 : 15 : 13 : 22
I (8379) xxxzzzzz: ----current Time = 120 : 8 : 26 : 15 : 13 : 23
I (9379) xxxzzzzz: ----current Time = 120 : 8 : 26 : 15 : 13 : 24
I (10379) xxxzzzzz: ----current Time = 120 : 8 : 26 : 15 : 13 : 25
I (11380) xxxzzzzz: ----current Time = 120 : 8 : 26 : 15 : 13 : 26
```

时间没同步程序就在 sntp_get_sync_status 位置等待，不会让 while(1) 获取无效的时间。

```
void sntp_sync_time(struct timeval *tv); //未注册回调可以使用时间更新函数同步时间
使用该函数需要取消 sntp_set_time_sync_notification_cb(...) 函数，这是因为 sntp_set_time_sync_notification_cb 中内部执行了
settimeofday(...), 写系统时间函数,然后我又在时间同步后，回调函数中执行了 settimeofday(...), 两次时间不一致影响系统时间。
void initialize_sntp(void)
{
    ESP_LOGI(TAG, "Initializing SNTP");
    sntp_setoperatingmode(SNTP_OPMODE_POLL);
    sntp_setservername(0, "ntp1.aliyun.com");
    sntp_setservername(1, "cn.ntp.org.cn");
    sntp_setservername(2, "pool.ntp.org");
    //sntp_set_time_sync_notification_cb(time_sy
```

初始化 SNTP 执行
SNTP_SYNC_MODE_SMOOTH 模式

```
#ifdef CONFIG_SNTP_TIME_SYNC_METHOD_SMOOTH
    sntp_set_sync_mode(SNTP_SYNC_MODE_SMOOTH);
#endif
    sntp_init();
```

```
#define CONFIG_SNTP_TIME_SYNC_METHOD_CUSTOM
#ifndef CONFIG_SNTP_TIME_SYNC_METHOD_CUSTOM
void sntp_sync_time(struct timeval *tv)
{
    //settimeofday(tv, NULL); //-
    xzz_time_sync_notification(tv); //同步系统时间
    ESP_LOGI(TAG, "Time is synchronized from custom code"
    sntp_set_sync_status(SNTP_SYNC_STATUS_COMPLETED);
}
#endif
```

SNTP 服务器同步成功会自动触发
sntp_sync_time 函数，所以在自己文件
任意位置定义 sntp_sync_time 函数即
可， sntp_sync_time 函数的*tv 会自动
获取 1970 到当前的时间，所以取出 tv
的时间进行计算即可

Wifi mesh 网络使用 (IDF 使用方法, 有点问题) 建议看 MDF

但是这里面的一些获取 mesh 网络信息的 API 可以使用

```
esp_err_t tcpip_adapter_dhcps_stop(tcpip_adapter_if_t tcpip_if) //停止 DHCP 服务器  
tcpip_if : TCPPIP_ADAPTER_IF_STA 停止 WIFI STA 接口  
TCPPIP_ADAPTER_IF_AP 停止 wifi soft-ap 接口  
TCPIP_ADAPTER_IF_ETH 停止以太网接口
```

```
esp_err_t tcpip_adapter_dhcpc_stop(tcpip_adapter_if_t tcpip_if) //停止 DHCP 客户端  
tcpip_if: 停止接口选项同上
```

```
esp_err_t esp_mesh_init(void) //mesh 网络初始化
```

```
mesh_cfg_t cfg = MESH_INIT_CONFIG_DEFAULT(); //mesh 网络配置需要分配结构体
```

mesh_cfg_t: uint8_t channel mesh 预设信道号, 默认填 0 信道,

信道不一样, 发射的中心频率也不一样, 如果当前信道太拥挤(反应出来就是网速慢), 可以设置自动切换信道功能。

bool allow_channel_switch 信道切换功能, 填 false 禁止信道自动切换, true 允许信道自动切换。

mesh_event_cb_t event_cb mesh 网络发生变化调用 event_cb 设置的回调函数
mesh_addr_t mesh_id 一个 mesh 网络中, 所有设备的 mesh id 必须一样, 都是一样 mesh id 的设备才可以相互进行根节点子节点连接

mesh_router_t router 设置要连接路由器 SSID 和密码, 路由器用来连接外网, mesh 网络根节点和路由器相连, 这样根节点的数据和子节点的数据才可以经过路由器与服务器交互。

mesh_router_t	uint8_t ssid[32]	填入路由器 ssid
	uint8_t ssid_len	填入 ssid 长度
	uint8_t bssid[6]	填入路由器 BSSID
如果设置了 bssid, 那么可以再设置下 allow_router_switch		
uint8_t password[64] //填入路由器密码		
bool allow_router_switch //写 true 如果设置 bssid 的路由器挂了, 可以切换成 ssid 模式, 找其他 ssid 一样的路由器连接。		

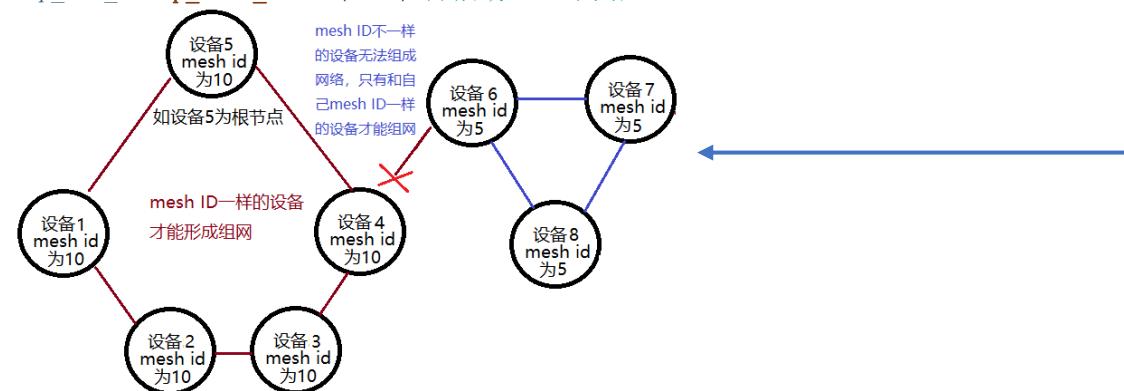
mesh_ap_cfg_t	uint8_t password[64]	自定义 mesh 网络软 AP 密码
	uint8_t max_connection	允许 mesh ap 每个子节点

最大连接设备数量, 每个节点最大连接不超过 10.

```
esp_err_t esp_mesh_set_config(const mesh_cfg_t *config) //配置 mesh 网络
```

*config: 传入我上面 mesh_cfg_t 设置的网络配置变量 cfg

```
esp_err_t esp_mesh_start(void) //启动 mesh 网络
```



代码例程

```
void mesh_event_handler(mesh_event_t event)
{
    switch (event.id) {
    case MESH_EVENT_STARTED:
        ESP_LOGI(MESH_TAG, "MESH_EVENT_STARTED");
        break;
    case MESH_EVENT_STOPPED:
        ESP_LOGI(MESH_TAG, "MESH_EVENT_STOPPED");
        break;
    case MESH_EVENT_CHILD_CONNECTED:
        ESP_LOGI(MESH_TAG, "MESH_EVENT_CHILD_CONNECTED");
        break;
    case MESH_EVENT_CHILD_DISCONNECTED:
        ESP_LOGI(MESH_TAG, "MESH_EVENT_CHILD_DISCONNECTED");
        break;
    default:
        ESP_LOGI(MESH_TAG, "default");
        break;
    }
}

void app_main(void)
{
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK( ret );

    tcpip_adapter_init(); //初始化 TCP/IP 适配层

    ESP_ERROR_CHECK(tcpip_adapter_dhcps_stop(TCPIP_ADAPTER_IF_AP)); //停止本设备 AP 模式下的 DHCP 分配 IP
    服务
    ESP_ERROR_CHECK(tcpip_adapter_dhcpc_stop(TCPIP_ADAPTER_IF_STA)); //停止本设备 STA DHCP 客户端被分配 IP
    服务
    ESP_ERROR_CHECK(esp_event_loop_init(NULL, NULL)); //wifi 启动后, 处理 wifi 事件的回调函数暂时不注册使用

    /*启动 ESP WIFI mesh 先决条件就是初始化 Lwip 和 wifi*/
    wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT(); //分配 wifi 配置内存
    ESP_ERROR_CHECK(esp_wifi_init(&config)); //初始化 wifi
    ESP_ERROR_CHECK(esp_wifi_start()); //启动 wifi
    /* mesh initialization */
    ESP_ERROR_CHECK(esp_mesh_init()); //mesh 初始化

    mesh_cfg_t cfg = MESH_INIT_CONFIG_DEFAULT(); //给 mesh 网络分配结构体内存, 用来配置 mesh 网络
    /* mesh ID */
    static const uint8_t MESH_ID[6] = { 0x77, 0x77, 0x77, 0x77, 0x77, 0x77 }; //这就是 mesh id 像 MAC 地址
    memcpy((uint8_t *) &cfg.mesh_id, MESH_ID, 6); //mesh 网络中, 每个节点设备的 mesh id 必须一样
    /* mesh event callback */
    cfg.event_cb = &mesh_event_handler; //mesh 网络连接后不管是子节点还是其它网络接入都会触发该回调函数
    /* router */
    cfg.channel = 0;
    cfg.router ssid_len = strlen("OPPO A11"); //计算连接路由器的 SSID 长度
    memcpy((uint8_t *) &cfg.router.ssid, "OPPO A11", cfg.router.ssid_len); //设置连接路由器的 SSID
    memcpy((uint8_t *) &cfg.router.password, "12345678", strlen("12345678")); //设置连接路由器的密码
    /* mesh softAP */
    ESP_ERROR_CHECK(esp_mesh_set_ap_authmode(WIFI_AUTH_WPA2_PSK)); //启动 mesh AP 设置加密方式 WPA2
    //这行代码 esp_mesh_set_ap_authmode 最好放在 esp_mesh_set_config 之后, 不然有些时候会造成系统不停重启

    cfg.mesh_ap.max_connection = 6;
    memcpy((uint8_t *) &cfg.mesh_ap.password, "87654321", strlen("87654321")); //这是 mesh AP 的密码
    ESP_ERROR_CHECK(esp_mesh_set_config(&cfg)); //配置 mesh 网络
    /* mesh start */
    ESP_ERROR_CHECK(esp_mesh_start()); //启动 mesh 网络
}
```

这些代码复制, 注意编码方式, 建议手写。

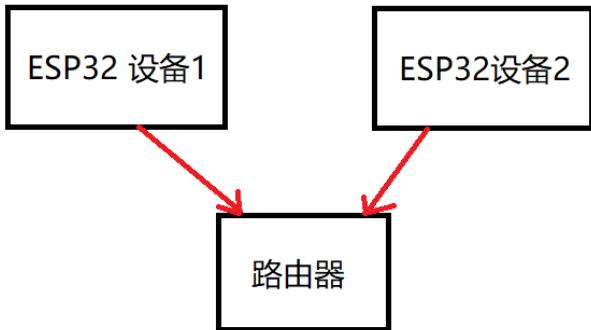
运行结果

注意: 为了方便, 路由器 SSID 我从 OPPO A11 改成了 metasia , 所以代码的 cfg.router.ssid 和密码都要改

```
I (1425) XXXXXXXX: MESH_EVENT_STARTED
```

ESP32 启动 wifi mesh 组网模式成功会自动执行一次 MESH_EVENT_STARTED 事件

现在有两个设备



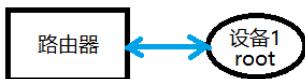
如果ESP32 设备1 先连接上路由器

会打印 wifi:connected with metasia, aid = 2, channel 1, 40U, bssid = 50:fa:84:3c:b7:90

bssid = 50:fa:84:3c:b7:90 路由器的BSSID

metasia 路由器名称

还要打印 mesh: <nvs>write layer:1 这表示ESP32 设备1在 1层节点, 也就是root 根节点

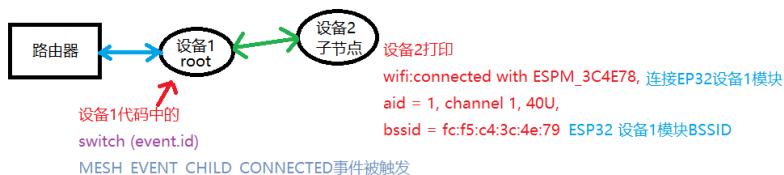
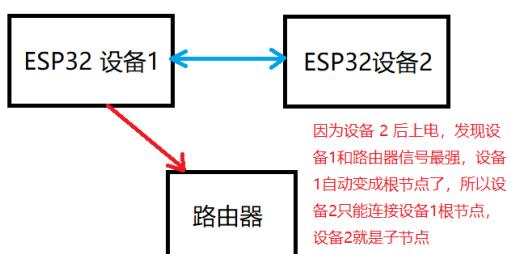


```
I (7488) wifi:connected with metasia, aid = 2, channel 1, BW20, bssid = 50:fa:84:3c:b7:47  
I (7492) wifi:security type: 4, phy: bgn, rss: -84  
I (7500) wifi:pm start, type: 0
```

```
E (7502) event: invalid static ip  
I (7504) mesh: [scan]new scanning time:600ms  
E (7508) wifi:[beacon]new interval:300ms  
I (7514) mesh: <nvs>write layer:1  
I (7537) wifi:AP's beacon interval = 102400 us, DTIM period = 1
```

这表示我设备 1 连接上 SSID 为 metasia 路由器, 设备处于 layer1 层 1 根节点

现在我给 ESP 设备 2 上电



```

I (4087) wifi:connected with ESPM_3C4E78, aid = 1, channel 1, 40U, bssid = fc:f5:c4:3c:4e:79
I (4089) wifi:security type: 0, phy: bgn, rssi: -29
I (4097) wifi:pm start, type: 0

E (4099) event: invalid static ip
I (4101) mesh: [scan]new scanning time:600ms
E (4105) wifi:[beacon]new interval:300ms
I (4109) mesh: 1891<am>parent monitor, my layer:2(cap:25)(node), interval:10555ms, retries:1<normal connected>
I (4235) wifi:AP's beacon interval = 307200 us, DTIM period = 2
I (14928) mesh: 4448<active>parent layer:1(node), channel:, rssi:-29, assoc:0(cnx rssи threshold:-120)my assoc:0
I (16432) mesh: 5193<scan>parent layer:1, rssi:-30, assoc:1(cnx rssи threshold:-120)

```

连接设备 1

设备 2 在层 2，也就是设备 1 root 的子节点

设备 1 与设备 2 之间的信道号，相互之间的信号强度。

`int esp_mesh_get_layer(void) //获取当前设备属于那一层节点`

返回 int 作为节点变量

该 API 一定在 `MESH_EVENT_PARENT_CONNECTED` 事件里面调用才行

```

void mesh_event_handler(mesh_event_t event)
{
    switch (event.id)
    {
        case MESH_EVENT_STARTED:
            ESP_LOGI(TAG, "MESH_EVENT_STARTED");
            break;

        case MESH_EVENT_STOPPED:
            ESP_LOGI(TAG, "MESH_EVENT_STOPPED");
            break;

        case MESH_EVENT_CHILD_CONNECTED:
            ESP_LOGI(TAG, "MESH_EVENT_CHILD_CONNECTED");
            break;

        case MESH_EVENT_CHILD_DISCONNECTED:
            ESP_LOGI(TAG, "MESH_EVENT_CHILD_DISCONNECTED");
            break;

        case MESH_EVENT_PARENT_CONNECTED:
            ESP_LOGI(TAG, "MESH_EVENT_PARENT_CONNECTED");
            mesh_layer = esp_mesh_get_layer();
            break;
    }
}

```

`esp_mesh_get_layer` 只有在 `MESH_EVENT_PARENT_CONNECTED` 事件调用，才能获取到设备层

```

ESP_ERROR_CHECK(esp_mesh_start()); //启动mesh网络

while (1)
{
    ESP_LOGI(TAG, "mesh_layer = %d", mesh_layer); //获取当前设备属于那一层
    sleep(1);
}

I (503170) xxxxxxxxxxx: mesh_layer = 1
I (504170) xxxxxxxxxxx: mesh_layer = 1
I (505170) xxxxxxxxxxx: mesh_layer = 1
I (506170) xxxxxxxxxxx: mesh_layer = 1
I (507170) xxxxxxxxxxx: mesh_layer = 1

```

主循环打印当前设备在 1 层，root 根节点

获取本设备当前是不是 mesh 网络根节点。

bool esp_mesh_is_root(void) //判断本设备是不是根节点
返回 true 表示设备处于根节点，返回 false 表示设备处于子节点

```
bool Rootflag = false;
while (1)
{
    ESP_LOGI(TAG, "mesh_layer = %d", mesh_layer); //获取当前设备属于哪一层
    Rootflag = esp_mesh_is_root();
    if(Rootflag == true)
    {
        ESP_LOGI(TAG, "current device root node");
    }
    else
    {
        ESP_LOGI(TAG, "current device child node");
    }
    sleep(1);
}
```

将该程序下载进两个设备

```
I (142674) xxxxxxxx: mesh_layer = -1
I (142674) xxxxxxxx: current device child node
I (142715) mesh: [SCAN][ch:1]AP:3, other(ID:0, RD:0), MAP:0, idle
I (142717) mesh: 1212[SCAN]init rc[fc:f5:c4:3c:4e:79,-85], mine:0
I (142724) mesh: [SCAN:13/13+x+]rc[128][fc:f5:c4:3c:4e:79,-89], s
1,fc:f5:c4:3c:4e:79]

I (142740) mesh: <nvs>write layer:0
E (142742) wifi:[beacon]interval:100ms
I (142746) mesh: [DONE]connect to router:metasia, channel:1, rssi
:79/-89/1]
I (143674) xxxxxxxx: mesh_layer = -1
I (143674) xxxxxxxx: current device root node
```

设备 A 在搜索连接路由的时候，没有执行 MESH_EVENT_PARENT_CONNECTED 事件，所以现在不管是判断当前设备属于哪一层，或者当前设备是不是根节点不重要。

```
I (145122) wifi:state: assoc -> run (10)
I (145140) wpa: <EAPOL>state:6
I (145141) wpa: <EAPOL>receiving the 1/4 EAPOL-Key, state:6
I (145158) wpa: <EAPOL>state:7
I (145159) wpa: <EAPOL>receiving the 3/4 EAPOL-Key, state:7
I (145159) wifi:connected with metasia, aid = 1, channel 1, 40U, bssid = 50:fa:84:3c:b7:47
I (145163) wifi:security type: 4, phy: bgn, rssi: -84
I (145219) wifi:pm start, type: 0

E (145222) event: invalid static ip
I (145222) mesh: [scan]new scanning time:600ms
E (145223) wifi:[beacon]new interval:300ms
I (145226) mesh: <nvs>write layer:1
I (145227) xxxxxxxx: MESH_EVENT_PARENT_CONNECTED
I (145616) wifi:AP's beacon interval = 102400 us, DTIM period = 1
I (146111) xxxxxxxx: mesh_layer = 1
I (146111) xxxxxxxx: current device root node
I (147111) xxxxxxxx: mesh_layer = 1
I (147111) xxxxxxxx: current device root node
I (147846) wifi:new:<1,1>, old:<1,1>, ap:<1,1>, sta:<1,1>, prof:1
I (147847) wifi:station: fc:f5:c4:3c:4a:1c join, AID=1, bgn, 40U
I (147851) mesh: <nvs>write assoc1
I (147851) xxxxxxxx: MESH_EVENT_CHILD_CONNECTED
I (148111) xxxxxxxx: mesh_layer = 1
I (148111) xxxxxxxx: current device root node
I (149111) xxxxxxxx: mesh_layer = 1
```

当设备 A 连接上路由器之后，MESH_EVENT_PARENT_CONNECTED 事件执行，本设备所在层显示获取到了，本设备是不是根节点也获取了。

下面看看设备 B

```
I (82585) XXXXXXXX: mesh_layer = -1
I (82585) XXXXXXXX: current device child node
I (83585) XXXXXXXX: mesh_layer = -1
I (83585) XXXXXXXX: current device child node
I (83925) wifi:new:<1,l>, old:<1,l>, ap:<1,l>, sta:<1,l>, prof:l
I (83926) wifi:state: init -> auth (b0)
I (83930) wifi:state: auth -> assoc (0)
I (83936) wifi:state: assoc -> run (10)
I (83937) wifi:connected with ESPM_3C4E78, aid = 1, channel 1, 40U, bssid = fc:f5:c4:3c:4e:79
I (83939) wifi:security type: 0, phy: bgn, rssi: -35
I (83948) wifi:pm start, type: 0

E (83949) event: invalid static ip
I (83952) mesh: [scan]new scanning time:600ms
E (83955) wifi:[beacon]new interval:300ms
I (83960) mesh: 1891<arp>parent monitor, my layer:2(cap:25)(node), interval:7435ms, retries:1<normal connected>
I (83969) XXXXXXXX: MESH_EVENT_PARENT_CONNECTED
I (84070) wifi:AP's beacon interval = 307200 us, DTIM period = 2
I (84585) XXXXXXXX: mesh_layer = 2
I (84585) XXXXXXXX: current device child node
I (85585) XXXXXXXX: mesh_layer = 2
I (85585) XXXXXXXX: current device child node
I (86585) XXXXXXXX: mesh_layer = 2
```

设备 B 在没有连接上路由器或者根节点之前，也是输出-1 层和 false。

当设备连接上根节点之后，`MESH_EVENT_PARENT_CONNECTED` 也会被触发，可以获取设备 B 在第 2 层，设备 B 属于子节点

使用 IDF 的 mesh 网络有可能会出现一会能连上路由，一会无法连上的情况，所以 mesh 网络最好用 MDF 库实现，可能是两设备 mesh ID 重了造成的，正好根节点代码有问题

```
I (7739) mesh: [SCAN][ch:6]AP:7, other(ID:0, RD:0), MAP:0, idle:0, candidate:1, root:0, topMAP:0[c:0,i:1][50:fa:84:3b:ce:7e]router found<-
I (7742) mesh: 1212[SCAN]init rc[fc:f5:c4:3c:4a:1d,-64], mine:0, voter:0
I (7748) mesh: [SCAN://10]rc[128][fc:f5:c4:3c:4a:1d,-64], self[fc:f5:c4:3c:4a:1c,-64,reason:0,votes:1,idle][mine:1,voter:1(1.00)percent:0.90][128,1,fc:f5:c4:3c:4a:1d]
I (8065) mesh: [SCAN][ch:6]AP:7, other(ID:0, RD:0), MAP:0, idle:0, candidate:1, root:0, topMAP:0[c:0,i:1][50:fa:84:3b:ce:7e]router found<-
I (8067) mesh: 1212[SCAN]init rc[fc:f5:c4:3c:4a:1d,-65], mine:0, voter:0
I (8073) mesh: [SCAN:8/10]rc[128][fc:f5:c4:3c:4a:1d,-64], self[fc:f5:c4:3c:4a:1c,-65,reason:0,votes:1,idle][mine:1,voter:1(1.00)percent:0.90][128,1,fc:f5:c4:3c:4a:1d]
I (8390) mesh: [SCAN][ch:6]AP:6, other(ID:0, RD:0), MAP:0, idle:0, candidate:1, root:0, topMAP:0[c:0,i:1][50:fa:84:3b:ce:7e]router found<-
I (8393) mesh: 1212[SCAN]init rc[fc:f5:c4:3c:4a:1d,-66], mine:0, voter:0
I (8399) mesh: [SCAN:9/10]rc[128][fc:f5:c4:3c:4a:1d,-64], self[fc:f5:c4:3c:4a:1c,-66,reason:0,votes:1,idle][mine:1,voter:1(1.00)percent:0.90][128,1,fc:f5:c4:3c:4a:1d]
```

一直不停的连接路由

ESP32 MDF wifi mesh 使用

```
mdf_err_t mdf_event_loop_init(mdf_event_loop_cb_t cb) //mesh 启动后事件回调函数  
cb: 填入回调函数
```

```
typedef mdf_err_t (*mdf_event_loop_cb_t)(mdf_event_loop_t event, void *ctx);  
//回调函数原型
```

```
mwifi_init_config_t cfg = MWIFI_INIT_CONFIG_DEFAULT(); //分配 mesh 网络结构
```

```
mdf_err_t mwifi_init(const mwifi_init_config_t *config) //mesh 网络初始化  
*config : 填入分配 mesh 网络的 mwifi_init_config_t 结构体
```

```
mdf_err_t mwifi_set_config(const mwifi_config_t *config) //配置 mesh 网络
```

```
*config: mwifi_config_t
```

```
char router_ssid[32]; 本设备连接路由器的 SSID
```

```
char router_password[64]; //本设备连接路由器的密码
```

```
uint8_t router_bssid[6]; 本设备连接路由器的 BSSID
```

```
uint8_t mesh_id[6]; 本设备 mesh 网络的 id 号, 其余设备如果都是这个 id 那么就是这个网络的子节点
```

```
char mesh_password[64]; 本设备 mesh 网络密码, 记住密码长度一定要大于 8 位, 不然系统会不停重启报错
```

```
uint8_t channel; 预设网络信道
```

```
uint8_t channel_switch_disable 如果这个值没设置, 设备全通道扫描
```

```
uint8_t router_switch_disable
```

```
mwifi_start() //启动 mesh 网络
```

```
mdf_event_loop_cb_t 设置的 mesh 事件回调函数事件值,
```

```
MDF_EVENT_MWIFI_STARTED mesh 网络自组网成功会自动执行一次该事件
```

```
MDF_EVENT_MWIFI_PARENT_CONNECTED 本设备与父节点连接上,触发该事件
```

```
MDF_EVENT_MWIFI_PARENT_DISCONNECTED 父节点与子节点断开触发该事件
```

```
MDF_EVENT_MWIFI_ROUTING_TABLE_ADD 新添加子接点会触发路由表更改
```

```
MDF_EVENT_MWIFI_ROUTING_TABLE_REMOVE 子节点断开会触发路由表删除
```

```
MDF_EVENT_MWIFI_ROOT_GOT_IP
```

```
int esp_mesh_get_total_node_num(void) //获取当前 mesh 网络设备总数
```

```

#include "mdf_common.h"
#include "sys/unistd.h"
#include "nvs.h"
#include "mwifi.h" //在 MDF 库使用中 该头文件包含了 esp wifi 和 mdf wifi
static const char *TAG = "xxxxxxxx";
static mdf_err_t event_loop_cb(mdf_event_loop_t event, void *ctx)
{
    bool RootFlag = false;
    switch (event) {
        case MDF_EVENT_MWIFI_STARTED:
            MDF_LOGI("MESH is started");
            break;
        case MDF_EVENT_MWIFI_PARENT_CONNECTED: //与父节点连接上事件触发
            MDF_LOGI("Parent is connected success");
            if((RootFlag = esp_mesh_is_root())) //判断本设备是不是根节点
            {
                MDF_LOGI("device root node");
            }
            else
            {
                MDF_LOGI("device sub node");
            }
            break;
        case MDF_EVENT_MWIFI_PARENT_DISCONNECTED: //与父节点断开，事件触发，当然父节点发现子节点断开也会触发
            MDF_LOGI("Parent is disconnected");
            break;
        case MDF_EVENT_MWIFI_ROUTING_TABLE_ADD: //如果本设备为父节点，子节点连接上该设备会触发该路由表事件
            MDF_LOGI("Subnode add");
        case MDF_EVENT_MWIFI_ROUTING_TABLE_REMOVE:
            MDF_LOGI("total_num: %d", esp_mesh_get_total_node_num()); //返回连接本设备父节点的子节点数，算上父节点
            break;
        case MDF_EVENT_MWIFI_ROOT_GOT_IP:
            MDF_LOGI("Root obtains the IP address. It is posted by LWIP stack automatically");
            break;
        default:
            break;
    }
    return MDF_OK;
}
void app_main()
{
    esp_err_t err = nvs_flash_init(); //记住在做 wifi 功能时，一定要对 nvs_flash 初始化，不然板子会一直重启
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );
    /*配置 esp IDF 库的 wifi*/
    tcpip_adapter_init();
    wifi_init_config_t IDFCfg = WIFI_INIT_CONFIG_DEFAULT(); //分配 wifi 初始化结构体
    MDF_ERROR_ASSERT(esp_event_loop_init(NULL, NULL)); //wifi 事件回调函数不用
    MDF_ERROR_ASSERT(esp_wifi_init(&IDFCfg)); //wifi 初始化
    MDF_ERROR_ASSERT(esp_wifi_set_mode(WIFI_MODE_STA)); //wifi 先启动为 STA 模式，如果 mesh 网络有根节点自动换成节点模式
    MDF_ERROR_ASSERT(esp_wifi_start());//启动 wifi
    /***** ****
    mwifi_init_config_t cfg = MWIFI_INIT_CONFIG_DEFAULT(); //分配 wifi mesh 结构体
    mwifi_config_t config = {
        .router_ssid = "metasia",
        .router_password = "metasia0902",
        .mesh_id = "12345",
        .mesh_password = "87654321", //mesh 密码一定要超过 8 位，不然系统会不断重启报 mwifi_set_config 错误
    };
    MDF_ERROR_ASSERT(mdf_event_loop_init(event_loop_cb)); //mesh 网络事件回调
    MDF_ERROR_ASSERT(mwifi_init(&cfg)); //初始化 wifi mesh
    MDF_ERROR_ASSERT(mwifi_set_config(&config)); //配置 wifi mesh 网络
    MDF_ERROR_ASSERT(mwifi_start());//启动 wifi mesh 网络
    while (1)
    {
        sleep(1);
    }
}

```

```

I (9956) [mwifi, 137]: Parent is connected
I (9958) [xxxxxxxx, 18]: Parent is connected success
D (9959) [mwifi, 131]: esp_mesh_event_cb event.id: 11
I (9965) [xxxxxxxx, 22]: device root node
I (9970) [mwifi, 234]: State represents: 0
D (9979) [mwifi, 131]: esp_mesh_event_cb.event.id: 14
D (10805) [mwifi, 131]: esp_mesh_event_cb.event.id: 17
I (10805) event: sta ip: 192.168.1.108, mask: 255.255.255.0, gw: 192.168.1.1
I (10807) [xxxxxxxx, 38]: Root obtains the IP address. It is posted by LwIP stack automatically

```

A 设备启动结果

就算本设备连接上路由器成为根节点，也会认为路由器是父节点，所以连接路由器成功，父节点事件触发

本设备为根节点

```

I (7424) [mwifi, 137]: Parent is connected
I (7429) [xxxxxxxx, 18]: Parent is connected success
D (7430) [mwifi, 131]: esp_mesh_event_cb event.id: 11
I (7435) [xxxxxxxx, 26]: device sub node
I (7440) [mwifi, 234]: State represents: 0

```

B 设备启动结果

B 设备启动后，连接 A 设备成功，父节点事件触发

本设备为子节点状态

```

I (79049) [mwifi, 222]: Routing table is changed by adding newly joined children add_num: 1, total_num: 2
I (79054) [xxxxxxxx, 33]: Subnode add
I (79058) mesh: <nvs>write assoc:1
I (79059) [xxxxxxxx, 35]: total_num: 2

```

A 设备状态发生变化

A 设备发现有子节点接入，执行路由表事件

这时候获取网络设备数量就算 2 个

如何获取子设备连接的到底是哪一个父节点，如何在父节点获取连接上来的子节点设备 ID 号？

```
esp_err_t esp_mesh_get_parent_bssid(mesh_addr_t *bssid) //获取父亲节点的 BSSID
```

```

mesh_addr_t ParentSSID; //子节点数量
while (1)
{
    esp_mesh_get_parent_bssid(&ParentSSID); //获取父节点BSSID
    MDF_LOGI("ParentSSID = %x : %x : %x : %x : %x : %x", ParentSSID.addr[0], ParentSSID.addr[1],
    ParentSSID.addr[2], ParentSSID.addr[3], ParentSSID.addr[4], ParentSSID.addr[5]);
    sleep(1);
}

```

```

I (8725) mesh: [SCAN:10/10]rc[128][fc:f5:c4:3c:4e:79,-62], self[fc:f5:c4:3c:4e:78,-61,reason:0,votes:2,f5:c4:3c:4e:79]
I (8740) mesh: <nvs>write layer:0
I (8743) mesh: <nvs>write assoc:0
E (8745) wifi:[beacon]interval:100ms
I (8749) mesh: [DONE]connect to router:metasia, channel:6, rssi:-61, 50:fa:84:3b:ce:7e[layer:0, assoc:0]
9/-62/1]
I (8885) wifi:new:<6,1>, old:<6,1>, ap:<6,1>, sta:<6,0>, prof:6
I (8886) wifi:state: init -> auth (b0)
I (8893) wifi:state: auth -> assoc (0)
I (8909) wifi:state: assoc -> run (10)
I (8921) wpa: <EAPOL>state:6
I (8922) wpa: <EAPOL>receiving the 1/4 EAPOL-Key, state:6
I (8939) wpa: <EAPOL>state:7
I (8939) wpa: <EAPOL>receiving the 3/4 EAPOL-Key, state:7
I (8940) wifi:connected with metasia, aid = 12, channel 6, BW20, bssid = 50:fa:84:3b:ce:7e

```

A 本设备 BSSID

连接的是路由器

```
I (11472) [xxxxxxxx, 84]: ParentSSID = 50 : fa : 84 : 3b : ce : 7e  
I (12472) [xxxxxxxx, 84]: ParentSSID = 50 : fa : 84 : 3b : ce : 7e  
I (13472) [xxxxxxxx, 84]: ParentSSID = 50 : fa : 84 : 3b : ce : 7e  
I (14472) [xxxxxxxx, 84]: ParentSSID = 50 : fa : 84 : 3b : ce : 7e  
I (15472) [xxxxxxxx, 84]: ParentSSID = 50 : fa : 84 : 3b : ce : 7e
```

这是 A 设备获取路由器的 BSSID

```
I (14473) [xxxxxxxx, 84]: ParentSSID = fc : f5 : c4 : 3c : 4e : 79  
I (15473) [xxxxxxxx, 84]: ParentSSID = fc : f5 : c4 : 3c : 4e : 79  
I (16473) [xxxxxxxx, 84]: ParentSSID = fc : f5 : c4 : 3c : 4e : 79  
I (17473) [xxxxxxxx, 84]: ParentSSID = fc : f5 : c4 : 3c : 4e : 79
```

这是 B 设备子节点获取到父节点 A 设备的 BSSID

这个 API `esp_mesh_get_parent_bssid(mesh_addr_t *bssid)` 最好是在 `MESH_EVENT_PARENT_CONNECTED` 父节点连接成功事件下使用

`int8_t mwifi_get_parent_rssi() //子节点获取父节点信号强度
返回父节点与本节点的 RSSI 值`

```
MDF_ERROR_ASSERT(mdf_event_loop_init(event_loop_cb)); //mesh 网络事件回调  
MDF_ERROR_ASSERT(mwifi_init(&cfg)); //初始化 wifi mesh  
MDF_ERROR_ASSERT(mwifi_set_config(&config)); //配置 wifi mesh 网络  
MDF_ERROR_ASSERT(mwifi_start()); //启动 wifi mesh 网络

int8_t NodeRSSI;
while (1)
{
    NodeRSSI = mwifi_get_parent_rssi();
    MDF_LOGI("parent RSSI = %d", NodeRSSI);
    sleep(1);
}
```

```
D (12948) [mwifi, 131]: esp_mesh_event_cb event.id: 3
I (13524) [xxxxxxxx, 83]: parent RSSI = -69
I (14524) [xxxxxxxx, 83]: parent RSSI = -68
I (15524) [xxxxxxxx, 83]: parent RSSI = -65
I (16524) [xxxxxxxx, 83]: parent RSSI = -67
I (17524) [xxxxxxxx, 83]: parent RSSI = -67
I (18524) [xxxxxxxx, 83]: parent RSSI = -62
```

这是 A 设备与路由器之间的信号强度关系

```
I (18475) [xxxxxxxx, 83]: parent RSSI = -23
I (19475) [xxxxxxxx, 83]: parent RSSI = -24
I (20475) [xxxxxxxx, 83]: parent RSSI = -24
I (21475) [xxxxxxxx, 83]: parent RSSI = -24
I (22475) [xxxxxxxx, 83]: parent RSSI = -23
I (23475) [xxxxxxxx, 83]: parent RSSI = -24
```

这是**B**设备里父节点**A**设备很近的信号强度

```
I (53475) [xxxxxxxx, 83]: parent RSSI = -41
I (54475) [xxxxxxxx, 83]: parent RSSI = -37
I (55475) [xxxxxxxx, 83]: parent RSSI = -37
I (56475) [xxxxxxxx, 83]: parent RSSI = -43
I (57475) [xxxxxxxx, 83]: parent RSSI = -43
I (58475) [xxxxxxxx, 83]: parent RSSI = -57
```

这是**B**设备离**A**设备比较远的信号强度

```
int esp_mesh_get_routing_table_size(void) //获取本设备下的子节点数量，不包含本设备上一级父节点的设备数
```

```
MDF_ERROR_ASSERT(mdf_event_loop_init(event_loop_cb)); //mesh网络事件回调
MDF_ERROR_ASSERT(mwifi_init(&cfg)); //初始化wifi mesh
MDF_ERROR_ASSERT(mwifi_set_config(&config)); //配置 wifi mesh 网络
MDF_ERROR_ASSERT(mwifi_start()); //启动wifi mesh网络

int8_t nodenum; //子节点数量
while (1)
{
    nodenum = esp_mesh_get_routing_table_size(); //获取本设备下子节点数量
    MDF_LOGI("nodenum = %d", nodenum);
    sleep(1);
}
```

```
I (8963) [xxxxxxxx, 22]: device root node
D (8968) [mwifi, 131]: esp_mesh_event_cb event.id: 14
I (9466) [xxxxxxxx, 83]: nodenum = 1
D (9801) [mwifi, 131]: esp_mesh_event_cb event.id: 17
I (9803) event: sta ip: 192.168.1.110, mask: 255.255.255.0, gw: 192.168.1.1
I (9804) [xxxxxxxx, 38]: Root obtains the IP address. It is posted by LwIP stack automatically
I (10466) [xxxxxxxx, 83]: nodenum = 1
I (11054) wifi:new:<6,1>, old:<6,1>, ap:<6,1>, sta:<6,0>, prof:6
I (11055) wifi:station: fc:f5:c4:3c:4a:1c join, AID=1, bgn, 40U
D (11075) [mwifi, 131]: esp_mesh_event_cb event.id: 5
I (11076) [mwifi, 222]: Routing table is changed by adding newly joined children add_num: 1, total_num: 2
I (11080) [xxxxxxxx, 33]: Subnode add
I (11087) mesh: <nvs>write assoc:1
I (11087) [xxxxxxxx, 35]: total_num: 2
D (11089) [mwifi, 131]: esp_mesh_event_cb event.id: 3
I (11466) [xxxxxxxx, 83]: nodenum = 2
I (12466) [xxxxxxxx, 83]: nodenum = 2
I (13466) [xxxxxxxx, 83]: nodenum = 2
I (14466) [xxxxxxxx, 83]: nodenum = 2
I (15466) [xxxxxxxx, 83]: nodenum = 2
I (16466) [xxxxxxxx, 83]: nodenum = 2
I (17466) [xxxxxxxx, 83]: nodenum = 2
```

A设备是根节点，加上一个**B**设备，在加上自身设备，一共为2节点

```
I (25753) mesh: 5193<scan>parent layer:1, rssi:-38, assoc:1(cnx rssи threshold:-120)
I (25754) mesh: [SCAN][ch:6]AP:1, other(ID:0, RD:0), MAP:1, idle:0, candidate:1, root:1, topMAP:0
I (25762) mesh: 6579[weak]try rssи threshold:-120, backoff times:0, max:5<-78,-82,-85>
I (25769) mesh: 616[monitor]no change, parent:fc:f5:c4:3c:4e:79, rssи:-38
I (25776) mesh: 1891<arm>parent monitor, my layer:2(cap:16)(node), interval:7733ms, retries:2<>
I (26466) [xxxxxxxx, 83]: nodenum = 1
I (27466) [xxxxxxxx, 83]: nodenum = 1
I (28466) [xxxxxxxx, 83]: nodenum = 1
I (29466) [xxxxxxxx, 83]: nodenum = 1
I (30466) [xxxxxxxx, 83]: nodenum = 1
```

B设备为子节点，**B**设备下没有网络连接进来，所以**B**设备只有自己节点，数量1。

ESP32 获取本模块的一些网络信息

```
esp_err_t esp_efuse_mac_get_default(uint8_t *mac)//获取本机出厂默认的 MAC 地址  
*mac : 传入 6 个 uint8_t 的数组  
获取 MAC 地址成功返回 ESP_OK
```

头文件位置 esp32/include/esp_system.h

下面我在本机 STA 模式连接路由器情况下实验

```
ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA) ); // WiFi 模式为 STA 模式  
ESP_ERROR_CHECK( esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config) );  
ESP_ERROR_CHECK( esp_wifi_start() );//启动wifi  
  
uint8_t MAC[6]; //获取ESP32出厂固化的MAC地址  
  
while (1)  
{  
    ESP_ERROR_CHECK(esp_efuse_mac_get_default(MAC));  
    ESP_LOGI(TAG, "MAC = %x:%x:%x:%x:%x:%x ", MAC[0], MAC[1], MAC[2], MAC[3], MAC[4], MAC[5]);  
    sleep(1);  
}
```

```
I (327) wifi:mode : sta (fc:f5:c4:3c:4a:lc)  
I (331) XXXXXXXXXXXSTA: MAC = fc:f5:c4:3c:4a:lc  
I (1056) wifi:new:<6,0>, old:<1,0>, ap:<255,255>, sta:<6,0>, prof:1  
I (2048) wifi:state: init -> auth (b0)  
I (2053) XXXXXXXXXXXSTA: MAC = fc:f5:c4:3c:4a:lc  
I (2056) wifi:state: auth -> assoc (0)  
I (2072) wifi:state: assoc -> run (10)  
I (2106) wifi:connected with metasia, aid = 6, channel 6, BW20, bssid = 50:fa:84:3b:ce:7e  
I (2107) wifi:security type: 4, phy: bgn, rssi: -55  
I (2112) wifi:pm start, type: 1  
  
I (2194) wifi:AP's beacon interval = 102400 us, DTIM period = 1  
I (3053) XXXXXXXXXXXSTA: MAC = fc:f5:c4:3c:4a:lc  
I (3562) event: sta ip: 192.168.1.128, mask: 255.255.255.0, gw: 192.168.1.1  
I (4053) XXXXXXXXXXXSTA: MAC = fc:f5:c4:3c:4a:lc  
I (5053) XXXXXXXXXXXSTA: MAC = fc:f5:c4:3c:4a:lc
```

这是本机默认打印的 MAC 地址

这是连接路由器，返回的路由器 BSSID

这是我自己的 MAC 地址和本机一样

```
esp_err_t tcpip_adapter_get_ip_info(tcpip_adapter_if_t tcpip_if,  
tcpip_adapter_ip_info_t *ip_info)//获取本机被路由器分配的 IP 地址, 掩码, 网关
```

tcpip_if: 获取某路接口的 IP 地址

TCPIP_ADAPTER_IF_STA STA 接口
TCPIP_ADAPTER_IF_AP AP 接口
TCPIP_ADAPTER_IF_ETH 以太网接口

```
typedef struct {  
    ip4_addr_t ip; /* Interface IPV4 address */  
    ip4_addr_t netmask; /* Interface IPV4 netmask */  
    ip4_addr_t gw; /* Interface IPV4 gateway address */  
} tcpip_adapter_ip_info_t;
```

```

    uint8_t MAC[6]; //获取ESP32出厂固化的MAC地址
    tcpip_adapter_ip_info_t local_ip;

    while (1)
    {
        ESP_ERROR_CHECK(esp_efuse_mac_get_default(MAC));
        ESP_LOGI(TAG, "MAC = %x:%x:%x:%x:%x:%x", MAC[0], MAC[1], MAC[2], MAC[3], MAC[4], MAC[5]);
        tcpip_adapter_get_ip_info(TCPIP_ADAPTER_IF_STA, &local_ip);
        printf("self:IPSTR\n", IP2STR(&local_ip.ip));
        printf("self:IPSTR\n", IP2STR(&local_ip.netmask));
        printf("self:IPSTR\n", IP2STR(&local_ip.gw));
        sleep(1);
    }
}

```

```

I (336) xxxxxxxxSTA: MAC = fc:f5:c4:3c:4a:1c
self:0.0.0.0
self:0.0.0.0
self:0.0.0.0
I (456) wifi:new:<6,0>, old:<1,0>, ap:<255,255>, sta:<6,0>, prof:1
I (457) wifi:state: init -> auth (b0)
I (473) wifi:state: auth -> assoc (0)
I (488) wifi:state: assoc -> run (10)
I (526) wifi:connected with metasia, aid = 6, channel 6, BW20, bssid = 50:fa:84:3b:ce:7e
I (527) wifi:security type: 4, phy: bgn, rssi: -54
I (531) wifi:pm start, type: 1
I (601) wifi:AP's beacon interval = 102400 us, DTIM period = 1
I (1238) event: sta ip: 192.168.1.128, mask: 255.255.255.0, gw: 192.168.1.1
I (1338) xxxxxxxxSTA: MAC = fc:f5:c4:3c:4a:1c
self:192.168.1.128
self:255.255.255.0
self:192.168.1.1

```

在没联网成功时，死循环情况下也能获取到 IP 地址

联网成功，路由器给本机分配的 IP 地址和网关

在联网成功后获取的 IP 地址

改进方法，为了防止死循环获取到无用的 IP 地址，建议把 IP 地址获取函数放入事件

```

tcpip_adapter_ip_info_t local_ip;

static esp_err_t event_handler(void *ctx, system_event_t *event)
{
    switch (event->event_id)
    {
        case SYSTEM_EVENT_STA_START:
            esp_wifi_connect(); //连接热点
            break;
        case SYSTEM_EVENT_STA_GOT_IP:
            break;
        case SYSTEM_EVENT_STA_CONNECTED: //联网成功
            ESP_LOGI(TAG, "SYSTEM_EVENT_STA_CONNECTED ");
            sleep(1); //联网成功后要等待1秒才能成功获取本机IP，如果不等待就会出现获取全部为0
            ESP_ERROR_CHECK(tcpip_adapter_get_ip_info(TCPIP_ADAPTER_IF_STA, &local_ip));
            printf("self:IPSTR\n", IP2STR(&local_ip.ip));
            printf("self:IPSTR\n", IP2STR(&local_ip.netmask));
            printf("self:IPSTR\n", IP2STR(&local_ip.gw));
            break;
    }
}

I (451) wifi:new:<6,0>, old:<1,0>, ap:<255,255>, sta:<6,0>, prof:1
I (452) wifi:state: init -> auth (b0)
I (461) wifi:state: auth -> assoc (0)
I (478) wifi:state: assoc -> run (10)
I (526) wifi:connected with metasia, aid = 8, channel 6, BW20, bssid = 50:fa:84:3b:ce:7e
I (527) wifi:security type: 4, phy: bgn, rssi: -65
I (532) wifi:pm start, type: 1

I (535) xxxxxxxxSTA: SYSTEM_EVENT_STA_CONNECTED
I (538) wifi:AP's beacon interval = 102400 us, DTIM period = 1
I (1331) xxxxxxxxSTA: MAC = fc:f5:c4:3c:4a:1c
self:192.168.1.128
self:255.255.255.0
self:192.168.1.1

```

把 IP 获取地址放在联网成功事件函数里面处理，但是必须延时 1 秒才能正常获取，这点必须得注意优化下

获取 IP 成功

ESP32 连接网络 ping 实现

```
#include "ping/ping.h" //ping_init 在该头文件
#include "esp_ping.h" //设置 esp_ping_set_target 函数在该头文件
```

```
esp_err_t esp_ping_set_target(ping_target_id_t opt_id, void *opt_val, uint32_t
opt_len); //ping 功能配置函数
```

opt_id: //要配置什么功能

PING_TARGET_IP_ADDRESS 设置 ip 地址 传入 uint32_t 变量

PING_TARGET_IP_ADDRESS_COUNT 设置 ping 多少次 传入 uint32_t 变量

PING_TARGET_RCV_TIMEOUT 每次 ping 等待服务器回传超时时间 传入 uint32_t 变量

PING_TARGET_DELAY_TIME 每次 ping 发送间隔时间 传入 uint32_t 变量

PING_TARGET_RES_FN 每次 ping 发送和接收回调函数，传入函数地址

opt_val: 传入 opt_id 设置参数对应的变量

opt_len: 计算对应变量的大小

```
esp_err_t pingResults(ping_target_id_t msgType, esp_ping_found * pf)
{
    printf("=====\\r\\n");
    printf("ping Send total_count %d \\r\\n", pf->send_count); //发送 ping 的个数
    printf("ping Recv total_count %d \\r\\n", pf->recv_count); //如果连接指定 IP 服务器成功，这里会接收回传
    printf("ping timeout_count %d \\r\\n", pf->timeout_count); //超时时间
    printf("ping min_time %d \\r\\n", pf->min_time); //最小时间
    printf("ping max_time %d \\r\\n", pf->max_time); //最大时间
    printf("ping resp_time %d\\r\\n", pf->resp_time); 响应时间
    return ESP_OK;
}
```

int ping_init(void); //ping 启动函数，在执行之前一定配置好 esp_ping_set_target 函数

```
void Start_Ping_ESP32(void){
    uint32_t ip_pinG = ipaddr_addr("163.177.151.109");//百度 IP 地址
    uint32_t ping_count = 2000;//ping 的次数
    uint32_t ping_timeout = 1000; //超时时间
    uint32_t ping_delay = 1000; //ping 的间隔
    ping_deinit();
    esp_ping_set_target(PING_TARGET_IP_ADDRESS_COUNT, &ping_count, sizeof(uint32_t)); //设置 ping 执行
    多少次
    esp_ping_set_target(PING_TARGET_RCV_TIMEOUT, &ping_timeout, sizeof(uint32_t)); //每次 ping 之后等待服
    务器回传时间最多等多久
    esp_ping_set_target(PING_TARGET_DELAY_TIME, &ping_delay, sizeof(uint32_t)); //每次 ping 发送间隔时间
    esp_ping_set_target(PING_TARGET_IP_ADDRESS, &ip_pinG, sizeof(uint32_t)); //ping 服务器的 IP 地址
    esp_ping_set_target(PING_TARGET_RES_FN, &pingResults, sizeof(pingResults)); //回调函数

    uint8_t res = 0;
    res = ping_init(); //初始化 ping，也就启动 ping
    if(res == 0){
        printf("PING set success!\\n");
    }else{
        printf("PING error:%d\\n",res);
    }
}
```

测试结论

```
PING set success!
I (475) wifi:new:<8,0>, old:<1,0>, ap:<255,255>, sta:<8,0>, prof:6
I (477) wifi:state: init -> auth (b0)
=====
ping Send total_count 1
ping Recv total_count 0
ping timeout_count 1
ping min_time 0
ping max_time 0
ping resp_time 1000
I (1478) wifi:state: auth -> init (200)
I (1478) wifi:new:<8,0>, old:<8,0>, ap:<255,255>, sta:<8,0>, prof:6
=====
ping Send total_count 2
ping Recv total_count 0
ping timeout_count 2
ping min_time 0
ping max_time 0
ping resp_time 1000
I (3653) wifi:new:<8,0>, old:<8,0>, ap:<255,255>, sta:<8,0>, prof:6
I (3654) wifi:state: init -> auth (b0)
I (4655) wifi:state: auth -> init (200)
I (4656) wifi:new:<8,0>, old:<8,0>, ap:<255,255>, sta:<8,0>, prof:6
```

在没有连接上路由器，你 ping 百度
收不到服务器数据包，所以接收数
据包为 0

```
I (10123) event: sta ip: 192.168.43.221, mask: 255.255.255.0, gw: 192.168.43.1
=====
ping Send total_count 6
ping Recv total_count 0
ping timeout_count 6
ping min_time 0
ping max_time 0
ping resp_time 1026
=====
ping Send total_count 7
ping Recv total_count 0
ping timeout_count 7
ping min_time 0
ping max_time 0
ping resp_time 1002
```

在连接上 wifi 路由器之后，你发现
路由器没有连接外网，那么只有发
送计数，接收服务器数据还是为 0

```

ping Send total_count 12
ping Recv total_count 1
ping timeout_count 11
ping min_time 491
ping max_time 491
ping resp_time 491
=====
ping Send total_count 13
ping Recv total_count 2
ping timeout_count 11
ping min_time 223
ping max_time 491
ping resp_time 223
=====
ping Send total_count 14
ping Recv total_count 3
ping timeout_count 11
=====
ping Send total_count 26
ping Recv total_count 9
ping timeout_count 17
ping min_time 218
ping max_time 491
ping resp_time 1008
=====
ping Send total_count 27
ping Recv total_count 9
ping timeout_count 18
ping min_time 218
ping max_time 491
ping resp_time 1004

```

在 wifi 路由器连上外网之后，我发给百度服务器的数据有了回应，证明外网连接成功，接收计数器计数

如果接收了 9 次服务器数据，我把外网断开了。那么接收数据量就停留在外网断开的最后一次值。而不会清 0.

所以为了判定网络是否断开，只有计算每一次 **Recv total_count** 的值是否大于上一次，如果等于上一次，证明外网断开。

void ping_deinit(void); //ping 停止函数

```

while (1)
{
    sleep(10);
    ping_deinit(); //ping 停止

    sleep(10);
    Start_Ping_ESP32(); //ping 启动
}


```

ping 5 次之后会停
一段时间，然后重
新开始 ping

```

ping Send total_count 4
ping Recv total_count 0
ping timeout_count 4
ping min_time 0
ping max_time 0
ping resp_time 1000
=====
ping Send total_count 5
ping Recv total_count 0
ping timeout_count 5
ping min_time 0
ping max_time 0
ping resp_time 1000
=====
ping Send total_count 5
ping Recv total_count 0
ping timeout_count 5
ping min_time 0
ping max_time 0
ping resp_time 1000
PING set success!
=====
ping Send total_count 1
ping Recv total_count 0
ping timeout_count 1
ping min_time 0
ping max_time 0
ping resp_time 1000
=====
ping Send total_count 2
ping Recv total_count 0
ping timeout_count 2
ping min_time 0
ping max_time 0
ping resp_time 1000

```

ESP32 AP 模式 与另外一台 ESP32 STA 设备或者手机 STA 设备相连进行客户端到服务器数据传输

先设定 **ESP32 AP** 的本地 IP 地址

```
esp_err_t tcpip_adapter_set_ip_info(tcpip_adapter_if_t tcpip_if, const  
tcpip_adapter_ip_info_t *ip_info) //设置本地设备 IP 地址
```

*tcpip_if: 设置接口 前面介绍过, TCPIP_ADAPTER_IF_STA STA 模式
TCPPIP_ADAPTER_IF_AP AP 模式*

**ip_info: 设置 IP 地址, 网关, 子网掩码的参数*

```
Struct tcpip_adapter_ip_info_t {  
    ip4_addr_t ip (uint32_t) //ip 地址  
    ip4_addr_t netmask (uint32_t) //子网掩码  
    ip4_addr_t gw (uint32_t) //网关  
}
```

```
tcpip_adapter_ip_info_t ip_info = {  
    .ip.addr = ipaddr_addr("192.168.1.99"),  
    .netmask.addr = ipaddr_addr("255.255.255.0"),  
    .gw.addr = ipaddr_addr("192.168.1.1"),  
};
```

这就是设置 IP 地址的方法

设置 IP 地址之后还必须写入芯片才行, 这时候要用 `tcpip_adapter_set_ip_info` 函数来写

```
ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_AP) ); //wifi 启动 AP 模式  
ESP_ERROR_CHECK( esp_wifi_set_config(ESP_IF_WIFI_AP, &ap_config) );  
ESP_ERROR_CHECK( esp_wifi_start() ); //启动 wifi  
  
tcpip_adapter_ip_info_t ip_info = {  
    .ip.addr = ipaddr_addr("192.168.1.99"),  
    .netmask.addr = ipaddr_addr("255.255.255.0"),  
    .gw.addr = ipaddr_addr("192.168.1.1"),  
};  
ESP_ERROR_CHECK(tcpip_adapter_dhcps_stop(TCPIP_ADAPTER_IF_AP)); //一定要先关掉  
DHCP 服务  
ESP_ERROR_CHECK(tcpip_adapter_set_ip_info(TCPIP_ADAPTER_IF_AP, &ip_info)); //设置  
本地 IP  
ESP_ERROR_CHECK(tcpip_adapter_dhcps_start(TCPIP_ADAPTER_IF_AP)); //重启 DHCP 服务
```

记住设置 IP 地址一定要在启动 AP 或者 STA 之后才能去设置, 尤其是 AP 模式, 如果启动之前去设置, AP 启动之后又会被 DHCP 改掉 IP 地址, 而且设备还搜索不到 AP

本地 IP 设置成功之后不能用<查询路由器给本机分配 IP 地址>章节的方法来获取本地 IP
`system_event_t . event_info.got_ip.ip_info.ip`

因为这个 `event_info.got_ip.ip_info.ip` 是在分配 IP 后的 IP 池查询的, 不是查询本地 IP 的

```
esp_err_t tcpip_adapter_get_ip_info(tcpip_adapter_if_t tcpip_if, tcpip_adapter_ip_info_t  
*ip_info) //查询本地 IP 地址  
tcpip_if: 接口TCPIP_ADAPTER_IF_STA STA 模式  
TCPPIP_ADAPTER_IF_AP AP 模式 我这里用 AP 模式  
*ip_info: 查询后信息存放地址
```

```
tcpip_adapter_ip_info_t local_ip; //获取本地 IP
ESP_ERROR_CHECK(tcpip_adapter_get_ip_info(TCPIP_ADAPTER_IF_AP, &local_ip));

printf("self:IPSTR\n", IP2STR(&local_ip.ip));
printf("self:IPSTR\n", IP2STR(&local_ip.netmask));
printf("self:IPSTR\n", IP2STR(&local_ip.gw));
```

```
self:192.168.1.99
self:255.255.255.0
self:192.168.1.1
```

这就打印出了你设置在本地的 IP 地址和网关,本地 IP 设置成功。

ESP32 TCP 客户端测试

客户端向服务器单向发数据

Socket 编程采用 linux 的 socket 编程方法

```
bool ConnectFlag = false; //什么时候创建 socket 标志位

static esp_err_t event_handler(void *ctx, system_event_t *event)
{
    switch (event->event_id)
    {
        case SYSTEM_EVENT_STA_START:
            esp_wifi_connect(); //连接热点
            break;
        case SYSTEM_EVENT_STA_GOT_IP:
            ESP_LOGI(TAG, "server ip : %s ", ip4addr_ntoa(&event->event_info.got_ip.ip_info.ip));
            break;
        case SYSTEM_EVENT_STA_CONNECTED: //联网成功
            ESP_LOGI(TAG, "connect AP success ..");
            ConnectFlag = true; //一定要连接上路由了才去创建 socket
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            esp_wifi_connect(); //连接热点
            break;
        default:
            break;
    }
    return ESP_OK;
}
```

```

void app_main()
{
    esp_err_t err = nvs_flash_init(); //记住在做 wifi 功能时，一定要对 nvs_flash 初始化，不然板子会一直重启
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );

    tcpip_adapter_init(); //初始化 TCP/IP 适配层
    ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT(); //分配 wifi 初始化结构体
    ESP_ERROR_CHECK( esp_wifi_init(&cfg) ); //wifi 初始化
    wifi_config_t wifi_config =
    {
        .sta = {
            .ssid = "metasia",
            .password = "metasia0902",
        },
    };
    ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA) ); //wifi 启动为 STA 模式
    ESP_ERROR_CHECK( esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config) );
    ESP_ERROR_CHECK( esp_wifi_start() ); //启动 wifi
    while(ConnectFlag == false) //等待连接上路由器才退出死循环，执行下面 socket 创建
    {
        sleep(4);
    }
    int client_socket = socket(AF_INET, SOCK_STREAM, 0); //创建 socket
    if(client_socket < 0)
    {
        ESP_LOGI(TAG,"socket Creat failed ..");
        close(client_socket);
    }
    else
        ESP_LOGI(TAG,"socket Creat success ..");
    struct sockaddr_in xserver_addr = {0};
    xserver_addr.sin_addr.s_addr = inet_addr("192.168.1.69"); //写入连接的服务端 IP 地址
    xserver_addr.sin_family = AF_INET; //IPV4 协议
    xserver_addr.sin_port = htons(50395); //服务端端口号
    int ret = connect(client_socket, (struct sockaddr *)&xserver_addr, sizeof(xserver_addr)); //连接服务器
    if (ret != 0)
    {
        ESP_LOGI(TAG,"TCP server connect failed ..");
        close(client_socket);
        return;
    }
    else
        ESP_LOGI(TAG,"socket connect success ..");
    while(1)
    {
        err = send(client_socket, "1234567", 6, 0); //发送数据到服务器
        if (err < 0) {
            ESP_LOGI(TAG,"send failed ..");
        }
        sleep(1);
    }
}

```

网络调试助手设置，接收客户端程序



协议类型选择 **TCP server** 服务器

主机 IP 地址选择电脑的 IP 地址，因为是 windows 电脑做的服务端，用 cmd 查询 ip 端口号选择电脑开放出来的端口，用 cmd 查询，然后点击打开，等待板子连接。

根据输出 “123456” 的内容，证明板子客户端发送数据到电脑成功。

在实际开发中，**socket** 创建和收发数据代码应该创建个线程去处理，该线程在 **SYSTEM_EVENT_STA_CONNECTED** 连接路由器成功事件中触发。

```
int client_socket = socket(AF_INET, SOCK_STREAM, 0); //创建 socket
if(client_socket < 0)
{
    ESP_LOGI(TAG,"socket Creat failed ..");
    close(client_socket);
}
else
    ESP_LOGI(TAG,"socket Creat success ..");
struct sockaddr_in xserver_addr = {0};
xserver_addr.sin_addr.s_addr = inet_addr("192.168.1.69");//写入连接的服务端 IP 地址
xserver_addr.sin_family = AF_INET; //IPV4 协议
xserver_addr.sin_port = htons(50395); //服务端端口号
int ret = connect(client_socket, (struct sockaddr *)&xserver_addr, sizeof(xserver_addr)); //连接服务器
if (ret != 0)
{
    ESP_LOGI(TAG,"TCP server connect failed ..");
    close(client_socket);
    return;
}
else
    ESP_LOGI(TAG,"socket connect success ..");
while(1)
{
    err = send(client_socket, "1234567", 6, 0); //发送数据到服务器
    if (err < 0) {
        ESP_LOGI(TAG,"send failed ..");
    }
    sleep(1);
}
```

就是这段代码应该放入线程来执行

服务器向客户端发送数据，双向发送

```
int client_socket = socket(AF_INET, SOCK_STREAM, 0); //创建 socket
if(client_socket < 0)
{
    ESP_LOGI(TAG,"socket Creat failed ..");
    close(client_socket);
}
else
    ESP_LOGI(TAG,"socket Creat success ..");

struct sockaddr_in xserver_addr = {0};
xserver_addr.sin_addr.s_addr = inet_addr("192.168.1.69");//写入连接的服务端 IP 地址
xserver_addr.sin_family = AF_INET; //IPV4 协议
xserver_addr.sin_port = htons(50395); //服务端端口号
int ret = connect(client_socket, (struct sockaddr *)&xserver_addr, sizeof(xserver_addr)); //连接服务器
if (ret != 0)
{
    ESP_LOGI(TAG,"TCP server connect failed ..");
    close(client_socket);
    return;
}
else
    ESP_LOGI(TAG,"socket connect success ..");

uint8_t rx_buffer[1024] = {0}; //创建一个接收数据缓冲区

while(1)
{
    err = send(client_socket, "1234567", 6, 0); //发送数据
    if (err < 0) {
        ESP_LOGI(TAG,"send failed ..");
    }
    int len = recv(client_socket, rx_buffer, sizeof(rx_buffer) - 1, 0); //接收服务端数据, 返回接收的数据长度
    if(len > 0){
        rx_buffer[len] = 0; // 数组最后一个等于 0
        printf("Received %d bytes from %s:\n", len, rx_buffer);
    }
    sleep(1);
}
```

连接服务器成功后
创建个接收数组

第一次发送数据到客户
端之后，程序会卡在
recv，因为这是阻塞式
接收，所以必须手动让
网络调试助手发数据





在客户端模式下设置客户端本地端口

```

int client_socket = socket(AF_INET, SOCK_STREAM, 0); //创建socket
if(client_socket < 0)
{
    ESP_LOGI(TAG,"socket Creat failed ..");
    close(client_socket);
}
else
    ESP_LOGI(TAG,"socket Creat success ..");

struct sockaddr_in xserver_addr = {0};
xserver_addr.sin_addr.s_addr = inet_addr("192.168.1.69");//写入连接的服务端IP地址
xserver_addr.sin_family = AF_INET; //IPV4协议
xserver_addr.sin_port = htons(50395); //服务端口号

struct sockaddr_in Loacl_addr; //设置本地端口
Loacl_addr.sin_addr.s_addr = htonl(INADDR_ANY);
Loacl_addr.sin_family = AF_INET;
Loacl_addr.sin_port = htons(54321);
uint8_t res = 0;
res = bind(client_socket,(struct sockaddr *)&Loacl_addr,sizeof(Loacl_addr));
if(res != 0){
    printf("bind error\n");
}
else
{
    printf("bind success\n");
}

```

加入 bind 设置本地客户端端口号

AP 模式下 TCP 服务端实现

可以用服务端的方式接收手机或者其它客户端发来的 wifi ssid 和密码，也可以和客户端进行数据通信

```
#include "mdf_common.h"
#include "sys/unistd.h"

#include "nvs.h"
#include "mwifi.h" //在 MDF 库使用中 该头文件包含了 esp wifi 和 mdf wifi
static const char *TAG = "xzzSTA";
bool ConnectFlag = false; //什么时候创建 socket 标志位
static esp_err_t event_handler(void *ctx, system_event_t *event)
{
    switch (event->event_id)
    {
        case SYSTEM_EVENT_STA_START:
            esp_wifi_connect(); //连接热点
            break;
        case SYSTEM_EVENT_STA_GOT_IP:
            ESP_LOGI(TAG, "server ip : %s ", ip4addr_ntoa(&event->event_info.got_ip.ip_info.ip));
            break;
        case SYSTEM_EVENT_STA_CONNECTED: //联网成功
            ESP_LOGI(TAG, "connect AP success ..");
            ConnectFlag = true; //一定要连接上路由器了才去创建 socket
            break;
        case SYSTEM_EVENT_STA_DISCONNECTED:
            esp_wifi_connect(); //连接热点
            break;
        default:
            break;
    }
    return ESP_OK;
}
void app_main()
{
    esp_err_t err = nvs_flash_init(); //记住在做 wifi 功能时，一定要对 nvs_flash 初始化，不然板子会一直重启
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );
    tcpip_adapter_init(); //初始化 TCP/IP 适配层
    ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT(); //分配 wifi 初始化结构体
    ESP_ERROR_CHECK( esp_wifi_init(&cfg) ); //wifi 初始化
    wifi_config_t wifi_config =
    {
        .sta = {
            .ssid = "ESP32AP",
            .password = "12345678",
        },
    };
    ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA) ); //wifi 启动为 STA 模式
    ESP_ERROR_CHECK( esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config) );
    ESP_ERROR_CHECK( esp_wifi_start() ); //启动 wifi
    while(ConnectFlag == false) //等待连接上路由器才退出死循环，执行下面 socket 创建
    {
        sleep(4);
    }
    int client_socket = socket(AF_INET, SOCK_STREAM, 0); //创建 socket
    if(client_socket < 0)
    {
        ESP_LOGI(TAG, "socket Creat failed ..");
        close(client_socket);
    }
    else
        ESP_LOGI(TAG, "socket Creat success ..");
    struct sockaddr_in xserver_addr = {0};
    xserver_addr.sin_addr.s_addr = inet_addr("192.168.1.99"); //写入连接的服务端 IP 地址
    xserver_addr.sin_family = AF_INET; //IPV4 协议
    xserver_addr.sin_port = htons(9999); //服务端端口号
    struct sockaddr_in Loacl_addr; //设置本地端口
    Loacl_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```

Loacl_addr.sin_family = AF_INET;
Loacl_addr.sin_port = htons(54321);
uint8_t res = 0;
res = bind(client_socket,(struct sockaddr *)&Loacl_addr,sizeof(Loacl_addr));
if(res != 0){
    printf("bind error\n");
}
else
{
    printf("bind success\n");
}
int ret = connect(client_socket, (struct sockaddr *)&xserver_addr, sizeof(xserver_addr)); //连接
服务器
if (ret != 0)
{
    ESP_LOGI(TAG,"TCP server connect failed ..");
    close(client_socket);
    return;
}
else
    ESP_LOGI(TAG,"socket connect success ..");

uint8_t rx_buffer[1024] = {0}; //创建一个接收数据缓冲区
while(1)
{
    err = send(client_socket, "1234567", 6, 0); //发送数据到服务器
    if (err < 0) {
        ESP_LOGI(TAG,"send failed ..");
    }
    sleep(1);
}
}

```

客户端连接服务器的 IP 地址和端口

```

int client_socket = socket(AF_INET, SOCK_STREAM, 0); //创建socket
if(client_socket < 0)
{
    ESP_LOGI(TAG,"socket Creat failed ..");
    close(client_socket);
}
else
    ESP_LOGI(TAG,"socket Creat success ..");

struct sockaddr_in xserver_addr = {0};
xserver_addr.sin_addr.s_addr = inet_addr("192.168.1.99");//写入连接的服务端IP地址
xserver_addr.sin_family = AF_INET; //IPv4协议
xserver_addr.sin_port = htons(9999); //服务端口号

int ret = connect(client_socket, (struct sockaddr *)&xserver_addr, sizeof(xserver_addr));
if (ret != 0)
{
    ESP_LOGI(TAG,"TCP server connect failed ..");
    close(client_socket);
    return;
}
else
    ESP_LOGI(TAG,"socket connect success ..");

uint8_t rx_buffer[1024] = {0}; //创建一个接收数据缓冲区
while(1)
{
    err = send(client_socket, "1234567", 6, 0); //发送数据到服务器
    if (err < 0) {
        ESP_LOGI(TAG,"send failed ..");
    }
    sleep(1);
}

```

每秒中给服务端发数据

```
I (564) xzzAP: sta connect AP ..
I (594) tcpip_adapter: softAP assign IP to station,IP is: 192.168.1.100
I (1414) xzzAP: server socket Creat success ..
I (1414) xzzAP: server bind success ..
I (1414) xzzAP: server listen success ..
I (7134) wifi:new:<1,1>, old:<1,1>, ap:<1,1>, sta:<255,255>, prof:1
I (7134) wifi:station: fc:f5:c4:3c:4e:78 join, AID=1, bgn, 40U
I (8074) wifi:new:<1,1>, old:<1,1>, ap:<1,1>, sta:<255,255>, prof:1
I (8074) wifi:station: fc:f5:c4:3c:4e:78 join, AID=1, bgn, 40U
I (8094) xzzAP: sta connect AP ..
I (8214) tcpip_adapter: softAP assign IP to station,IP is: 192.168.1.100
E (11954) xzzAP: recv : 123456
E (12954) xzzAP: recv : 123456
E (13954) xzzAP: recv : 123456
E (14954) xzzAP: recv : 123456
E (15954) xzzAP: recv : 123456
```

服务端收到客户端的数据。

ESP32 wifi MESH 网络数据转发

需要有前面< ESP32 MDF wifi mesh 使用>的知识



```
esp_err_t esp_mesh_set_group_id(const mesh_addr_t *addr, int num) //设置组 ID
```

```
mdf_err_t mwifi_write(const uint8_t *dest_addrs, const  
                      mwifi_data_type_t *data_type, const void *data, size_t size, bool block) //发送数据给  
mesh 网络中指定 mac 地址的节点, (注意这个 mwifi_write 不是向服务器发送数据, 而是  
向网络子节点发送数据, 而且向网络子节点发数据必须用 mwifi_write, 不能用 socket)
```

***dest_addrs**: 数据要发送到哪一个父节点设备, 就填入父节点设备的 MAC 地址。父节点
的 MAC 地址可以用 mwifi_read 或者在事件中执行获取父节点 MAC 函数。

***data_type**: 如果使用默认配置, 参数填 NULL

***data**: 要发送给父节点的数据包

size: 数据包长度, 用 strlen 计算出来

block: 是否阻塞等待数据发送结果, true 为阻塞等待, false 不等待

```
mdf_err_t mwifi_read(uint8_t *src_addr, mwifi_data_type_t *data_type,  
                     void *data, size_t *size, TickType_t wait_ticks) //读取父节点发过来的数据。
```

mwifi_read 只能接收 mesh 网络节点之间的数据, 和 **mwifi_write** 一样。不能直接接收服
务端的数据, 如果要接收服务端的数据, 必须向根节点读取。

***src_addr**: 接收数据后, 父节点的 MAC 地址会存放在 *src_addr 中, 这样我就知道是哪
个节点传输给我的数据。所以如果不在事件中获取父节点地址, 可以在这里 *src_addr 中获
取。

***data_type**: 如果使用默认配置, 参数填 NULL

***data**: 接收父节点发来的数据包

***size**: 定义个变量, 来计算父节点发过来的数据包长度。

wait_ticks: 如果没有收到数据包, 要不要程序阻塞等待, 填入需要阻塞等待的时间

```
bool mwifi_is_connected(void) //本节点是否连接上父节点  
连接上返回 true, 未连接上返回 false
```

代码示例 ESP32 节点 1 root 节点代码

```
#include "mdf_common.h"  
#include "sys/unistd.h"  
#include "nvs.h"  
#include "mwifi.h" //在 MDF 库使用中 该头文件包含了 esp wifi 和 mdf wifi  
  
static const char *TAG = "xxxxxxxx";  
bool ConnectFlag = false; //什么时候创建 socket 标志位  
static mdf_err_t event_loop_cb(mdf_event_loop_t event, void *ctx)  
{  
    bool RootFlag = false;  
    switch (event) {  
        case MDF_EVENT_MWIFI_STARTED:  
            MDF_LOGI("MESH is started");  
            break;  
        case MDF_EVENT_MWIFI_PARENT_CONNECTED: //与父节点连接上事件触发  
            MDF_LOGI("Parent is connected success");  
            if((RootFlag = esp_mesh_is_root())) //判断本设备是不是根节点  
            {  
                MDF_LOGI("device root node");  
                ConnectFlag = true;  
            }  
            else  
            {  
                MDF_LOGI("device sub node");  
            }  
            break;  
        case MDF_EVENT_MWIFI_PARENT_DISCONNECTED: //与父节点断开, 事件触发, 当然父节点发现子节点断开也会触发  
            MDF_LOGI("Parent is disconnected");  
            break;  
        case MDF_EVENT_MWIFI_ROUTING_TABLE_ADD: //如果本设备为父节点, 子节点连接上该设备会触发该路由表事件  
            MDF_LOGI("Subnode add");  
        case MDF_EVENT_MWIFI_ROUTING_TABLE_REMOVE:  
            MDF_LOGI("total_num: %d", esp_mesh_get_total_node_num()); //返回连接本设备父节点的子节点数, 算上父节点  
            break;  
        case MDF_EVENT_MWIFI_ROOT_GOT_IP: {  
            MDF_LOGI("Root obtains the IP address. It is posted by LwIP stack automatically");  
            break;  
        }  
        default:  
            break;  
    }  
    return MDF_OK;  
}
```

这是 mesh 网络事件触发函数, 和<[ESP32 MDF wifi mesh](#)

```
void app_main()  
{  
    esp_err_t err = nvs_flash_init(); //记住在做 wifi 功能时, 一定要对 nvs_flash 初始化,  
不然板子会一直重启  
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {  
        ESP_ERROR_CHECK(nvs_flash_erase());  
        err = nvs_flash_init();
```

```

    }
    ESP_ERROR_CHECK( err );
    /*配置 esp IDF 库 的 wifi*/
    tcpip_adapter_init();
    wifi_init_config_t IDFCfg = WIFI_INIT_CONFIG_DEFAULT(); //分配 wifi 初始化结构体
    MDF_ERROR_ASSERT(esp_event_loop_init(NULL, NULL)); //wifi 事件回调函数不用
    MDF_ERROR_ASSERT(esp_wifi_init(&IDFCfg)); //wifi 初始化
    MDF_ERROR_ASSERT(esp_wifi_set_mode(WIFI_MODE_STA)); //wifi 先启动为 STA 模式, 如果
mesh 网络有根节点自动换成节点模式
    MDF_ERROR_ASSERT(esp_wifi_start()); //启动 wifi
    /*****
    mwifi_init_config_t cfg = MWIFI_INIT_CONFIG_DEFAULT(); //分配 wifi mesh 结构体
    mwifi_config_t config = {
        .router_ssid = "metasia",
        .router_password = "metasia0902",
        .mesh_id = "12345",
        .mesh_password = "87654321", //mesh 密码一定要超过 8 位, 不然系统会不断重启报
    mwifi_set_config 错误
    };
    MDF_ERROR_ASSERT(mdf_event_loop_init(event_loop_cb)); //mesh 网络事件回调
    MDF_ERROR_ASSERT(mwifi_init(&cfg)); //初始化 wifi mesh
    MDF_ERROR_ASSERT(mwifi_set_config(&config)); //配置 wifi mesh 网络
    MDF_ERROR_ASSERT(mwifi_start()); //启动 wifi mesh 网络
    while(ConnectFlag == false) //等待连接上路由器才退出死循环, 执行下面 socket 创建
    {
        sleep(4);
    }
    int sockfd = -1;
    struct sockaddr_in server_addr = { //连接服务器的 IP 和端口号
        .sin_family = AF_INET,
        .sin_port = htons(50395),
        .sin_addr.s_addr = inet_addr("192.168.1.69"),
    };
    sockfd = socket(AF_INET, SOCK_STREAM, 0); //如果是 root 根节点就必须用 socket 连接服
务器
    if(sockfd < 0)
    {
        ESP_LOGI(TAG,"socket Creat failed ..");
        close(sockfd);
    }
    else
        ESP_LOGI(TAG,"socket Creat success ..");
    int ret = connect(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockad
dr_in)); //连接服务器
    if (ret != 0)
    {
        ESP_LOGI(TAG,"TCP server connect failed ..");
        close(sockfd);
        return;
    }
    mwifi_data_type_t data_type = {0x0};
    char *data = MDF_CALLOC(1, MWIFI_PAYLOAD_LEN); //动态分配缓
存, MWIFI_PAYLOAD_LEN 最大有效载荷 1456
    size_t size = MWIFI_PAYLOAD_LEN - 1; //最大有效载荷-1
    uint8_t src_addr[6] = {0x0}; //源地址
    while (1)
    {
        while (mwifi_is_connected()) //查询本根节点是否连接上路由器
        {
            memset(data, 0, MWIFI_PAYLOAD_LEN);
            ret = mwifi_read(src_addr, &data_type, data, &size, portMAX_DELAY); //接
收子节点数据
            if (ret < 0)

```



```

        ESP_LOGI(TAG, "mwifi_root_read failed ..");
    else
        ESP_LOGI(TAG, "mwifi_root_read success ..");
    ret = write(sockfd, data, size); //将子节点数据发送给服务器
    if (ret < 0)
        ESP_LOGI(TAG, "write failed ..");
    else
        ESP_LOGI(TAG, "write success ..");
    }
    sleep(1);
}

```

将data数据用socket的write发送给服务器

下面我们编写 节点 2 的发送代码

```
mdf_err_t mwifi_write(const uint8_t *dest_addrs, const
                      mwifi_data_type_t *data_type, const void *data, size_t size, bool block) //向根节点或者父节点发送数据，也是用 mwifi_write
```

dest_addrs: 发数据包要到达的目的设备的 **MAC 地址**，如果是发给根节点可以直接填 **NULL**，如果是发给 **mesh** 网络中的某一个节点，就需要填入 **MAC 地址**，包括发送给中间父节点，也需要填入 **MAC 地址**

我们现在先测试填入 **MAC 地址**的方法，记住根节点设备的 **MAC 地址**

节点 2 发送给目标节点的代码(**目标节点可能是根节点，也可能是父节点，也可能是其它子节点**)

```
#include "mdf_common.h"
#include "sys/unistd.h"
#include "nvs.h"
#include "mwifi.h" //在 MDF 库使用中 该头文件包含了 esp wifi 和 mdf wifi
static const char *TAG = "xzzMeshNode2";
bool ConnectFlag = false; //什么时候创建 socket 标志位
static mdf_err_t event_loop_cb(mdf_event_loop_t event, void *ctx)
{
    bool RootFlag = false;
    switch (event) {
        case MDF_EVENT_MWIFI_STARTED:
            MDF_LOGI("MESH is started");
            break;
        case MDF_EVENT_MWIFI_PARENT_CONNECTED: //与父节点连接上事件触发
            MDF_LOGI("Parent is connected success");
            if((RootFlag = esp_mesh_is_root())) //判断本设备是不是根节点
            {
                MDF_LOGI("device root node");
                ConnectFlag = true;
            }
            else
            {
                MDF_LOGI("device sub node");
                ConnectFlag = true;
            }
            break;
        case MDF_EVENT_MWIFI_PARENT_DISCONNECTED: //与父节点断开，事件触发，当然父节点发现子节点断开也会触发
            MDF_LOGI("Parent is disconnected");
            break;
        case MDF_EVENT_MWIFI_ROUTING_TABLE_ADD: //如果本设备为父节点，子节点连接上该设备会触发该路由表事件
            MDF_LOGI("Subnode add");
        case MDF_EVENT_MWIFI_ROUTING_TABLE_REMOVE:
            MDF_LOGI("total_num: %d", esp_mesh_get_total_node_num()); //返回连接本设备父节点的子节点数，算上父节点
    }
}
```

```

        break;
    case MDF_EVENT_MWIFI_ROOT_GOT_IP:
        MDF_LOGI("Root obtains the IP address. It is posted by LwIP stack automatically");
        break;
    default:
        break;
    }
    return MDF_OK;
}
void app_main()
{
    esp_err_t err = nvs_flash_init(); //记住在做 wifi 功能时，一定要对 nvs_flash 初始化，不然板子会一直重启
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );
    /*配置 esp IDF 库的 wifi*/
    tcpip_adapter_init();
    wifi_init_config_t IDFCfg = WIFI_INIT_CONFIG_DEFAULT(); //分配 wifi 初始化结构体
    MDF_ERROR_ASSERT(esp_event_loop_init(NULL, NULL)); //wifi 事件回调函数不用
    MDF_ERROR_ASSERT(esp_wifi_init(&IDFCfg)); //wifi 初始化
    MDF_ERROR_ASSERT(esp_wifi_set_mode(WIFI_MODE_STA)); //wifi 先启动为 STA 模式，如果
mesh 网络有根节点自动换成节点模式
    MDF_ERROR_ASSERT(esp_wifi_start()); //启动 wifi
    /*****
    mwifi_init_config_t cfg = MWIFI_INIT_CONFIG_DEFAULT(); //分配 wifi mesh 结构体
    mwifi_config_t config = {
        .router_ssid = "metasia",
        .router_password = "metasia0902",
        .mesh_id = "12345",
        .mesh_password = "87654321", //mesh 密码一定要超过 8 位，不然系统会不断重启报
mwifi_set_config 错误
    };
    MDF_ERROR_ASSERT(mdf_event_loop_init(event_loop_cb)); //mesh 网络事件回调
    MDF_ERROR_ASSERT(mwifi_init(&cfg)); //初始化 wifi mesh
    MDF_ERROR_ASSERT(mwifi_set_config(&config)); //配置 wifi mesh 网络
    MDF_ERROR_ASSERT(mwifi_start()); //启动 wifi mesh 网络
    while(ConnectFlag == false) //等待连接上路由器才退出死循环，执行下面 socket 创建
    {
        sleep(4);
    }
    //root fc:f5:c4:3c:4e:78
    uint8_t dest_addr[6] = {0xfc,0xf5,0xc4,0x3c,0x4e,0x78};
    mwifi_data_type_t data_type = {0x0};
    int ret = -1;
    while (1)
    {
        while (mwifi_is_connected())
        {
            ret = mwifi_write(dest_addr, &data_type, "MeshNode2", strlen("MeshNode2"),
"), true); //发送数据到指定节点
            if (ret < 0) {
                ESP_LOGI(TAG, "mwifi_write failed ..");
            }
            sleep(1);
        }
        sleep(1);
    }
}

```

目标地址很重要，除了发送给根节点可以 NULL，发送给其它节点必须填 MAC 地址，mesh 网络是靠 MAC 地址找设备的



因为我是单向传输，所以必须先启动根节点，再启动子节点去连接根节点。

```
I (412) wifi:mode : sta [fc:f5:c4:3c:4e:78] + softAP (fc:f5:c4:3c:4e:79)
I (420) wifi:Total power save buffer number: 16
I (421) wifi:Init max length of beacon: 752/752
I (426) wifi:Init max length of beacon: 752/752
I (433) mesh: <nvs>read layer:0
I (433) mesh: <nvs>read assoc:0
E (436) wifi:[beacon]interval:100ms
I (447) wifi:Total power save buffer number: 16
I (447) wifi:Set ps type: 0
```

根节点启动

根节点 mac 地址是 fc:f5:c4:3c:4e:78

```
I (9954) mesh: [scan]new scanning time:600ms
E (9954) wifi:[beacon]new interval:300ms
I (9959) mesh: <nvs>write layer:1
D (9959) [mwifi, 131]: esp_mesh_event_cb event.id: 7
I (9964) [mwifi, 137]: Parent is connected
I (9969) [xxxxxxxx, 17]: Parent is connected success
D (9970) [mwifi, 131]: esp_mesh_event_cb event.id: 11
I (9975) [xxxxxxxx, 21]: device root node
J (9980) [mwifi, 234]: State represents: 0
D (9989) [mwifi, 131]: esp_mesh_event_cb event.id: 14
I (10002) wifi:AP's beacon interval = 102400 us, DTIM period = 1
D (11308) [mwifi, 131]: esp_mesh_event_cb event.id: 17
I (11309) event: sta ip: 192.168.1.119, mask: 255.255.255.0, gw: 192.168.1.1
I (11311) [xxxxxxxx, 38]: Root obtains the IP address. It is posted by LwIP stack automatically
```

根节点等待子节点连接

```
I (13654) mesh: 4448<active>parent layer:1[7node], channel:1, rssi:-42, assoc:0(cnx rssi threshold:-120)my_assoc:0
D (14505) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4e:78, mesh_data.size: 9, data: MeshNode2
I (15158) mesh: 5193<scan>parent layer:1, rssi:-45, assoc:1(cnx rssi threshold:-120)
I (15159) mesh: [SCAN][ch.1]AP 1, other{ID:0, RD:0}, MAP 1, idle:0, candidate:1, root:1, topMAP.0fc.2,i.2}[50:fa:84:3b:ce:7e]<weak>
I (15167) mesh: 6579[weaktry rssi_threshold:-120, backoff times:0, max:-78,-82,-85>
I (15174) mesh: 616[monitor]n change, parent:fc:f5:c4:3c:4e:79, rssi:-45
I (15181) mesh: 1891<arm>parent monitor, my layer:2(cap:16)(node), interval:11823ms, retries:2<>
```

子节点再 my layer2 层 2 发送的目标 mac 地址确实是根节点 fc:f5:c4:3c:4e:78

```
D (15508) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4e:78, mesh_data.size: 9, data: MeshNode2
D (16515) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4e:78, mesh_data.size: 9, data: MeshNode2
D (17521) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4e:78, mesh_data.size: 9, data: MeshNode2
D (18527) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4e:78, mesh_data.size: 9, data: MeshNode2
D (19530) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4e:78, mesh_data.size: 9, data: MeshNode2
D (20539) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4e:78, mesh_data.size: 9, data: MeshNode2
D (21543) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4e:78, mesh_data.size: 9, data: MeshNode2
```

子节点发送的数据

```
D (14909) [mwifi, 1091]: esp_mesh_recv, src_addr: fc:f5:c4:3c:4a:lc, size: 9, data: MeshNode2
I (14910) xxxxxxxx: mwifi_root_read success ..
I (14917) xxxxxxxx: write success ..
D (15912) [mwifi, 1091]: esp_mesh_recv, src_addr: fc:f5:c4:3c:4a:lc, size: 9, data: MeshNode2
I (15912) xxxxxxxx: mwifi_root_read success ..
I (15917) xxxxxxxx: write success ..
D (16922) [mwifi, 1091]: esp_mesh_recv, src_addr: fc:f5:c4:3c:4a:lc, size: 9, data: MeshNode2
I (16923) xxxxxxxx: mwifi_root_read success ..
I (16928) xxxxxxxx: write success ..
```

根节点收到子节点的数据，还有每一帧包含的子节点 MAC 地址

根节点再转发到服务器或者 PC 电脑

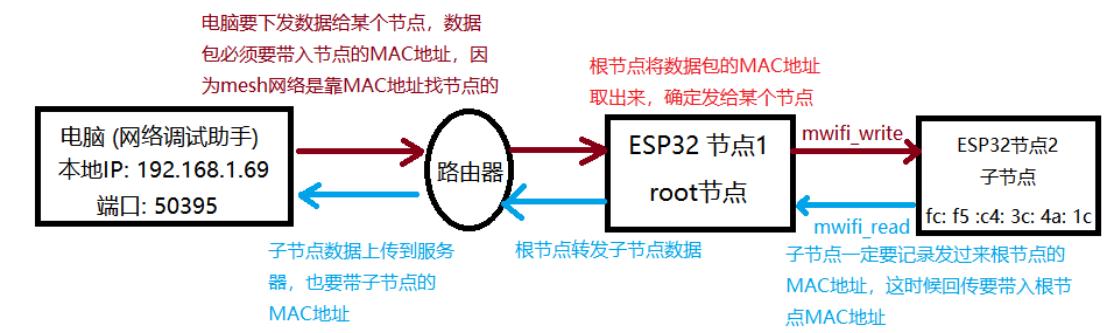


mesh 网络初步功能调试成功

根节点自动识别，子节点自动识别，自动识别目的 MAC 地址，自动识别源 MAC

以上代码是单向传输，我们是人为让根节点先启动来解决这个问题，实际使用过程中必须让设备有自动判别根节点的能力，还有自动识别 MAC 地址目的地的能力。

判断根节点我们必须使用<Wifi mesh 网络使用（IDF 使用方法，有点问题）>章节根节点判定函数
esp_mesh_is_root(void)



`esp_err_t esp_mesh_get_parent_bssid(mesh_addr_t *bssid) //获取父节点 MAC 地址
*bssid: 定义 mesh_addr_t 变量，变量里面包含 addr[6]参数`

根节点例程

```
#include "mdf_common.h"
#include "sys/unistd.h"
#include "nvs.h"
#include "mwifi.h" //在 MDF 库使用中 该头文件包含了 esp wifi 和 mdf wifi

static const char *TAG = "xxxxxxxx";
bool ConnectFlag = false; //什么时候创建 socket 标志位
static mdf_err_t event_loop_cb(mdf_event_loop_t event, void *ctx)
{
    bool RootFlag = false;
    switch (event) {
        case MDF_EVENT_MWIFI_STARTED:
            MDF_LOGI("MESH is started");
            break;
        case MDF_EVENT_MWIFI_PARENT_CONNECTED: //与父节点连接上事件触发
            MDF_LOGI("Parent is connected success");
            if((RootFlag = esp_mesh_is_root())) //判断本设备是不是根节点
            {
                MDF_LOGI("device root node");
                ConnectFlag = true;
            }
            else
            {
                MDF_LOGI("device sub node");
            }
            break;
        case MDF_EVENT_MWIFI_PARENT_DISCONNECTED: //与父节点断开，事件触发，当然父节点发现子节点断开也会触发
            MDF_LOGI("Parent is disconnected");
            break;
        case MDF_EVENT_MWIFI_ROUTING_TABLE_ADD: //如果本设备为父节点，子节点连接上该设备会触发该路由表事件
            MDF_LOGI("Subnode add");
        case MDF_EVENT_MWIFI_ROUTING_TABLE_REMOVE:
            MDF_LOGI("total_num: %d", esp_mesh_get_total_node_num()); //返回连接本设备父节点的子节点数，算上父节点
            break;
        case MDF_EVENT_MWIFI_ROOT_GOT_IP:
            MDF_LOGI("Root obtains the IP address. It is posted by LwIP stack automatically");
            break;
        default:
            break;
    }
    return MDF_OK;
}
void app_main()
{
    esp_err_t err = nvs_flash_init(); //记住在做 wifi 功能时，一定要对 nvs_flash 初始化，不然板子会一直重启
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );
    /*配置 esp IDF 库的 wifi*/
    tcpip_adapter_init();
    wifi_init_config_t IDFCfg = WIFI_INIT_CONFIG_DEFAULT(); //分配 wifi 初始化结构体
    MDF_ERROR_ASSERT(esp_event_loop_init(NULL, NULL)); //wifi 事件回调函数不用
    MDF_ERROR_ASSERT(esp_wifi_init(&IDFCfg)); //wifi 初始化
    MDF_ERROR_ASSERT(esp_wifi_set_mode(WIFI_MODE_STA)); //wifi 先启动为 STA 模式，如果 mesh 网络有根节点自动换成节点模式
    MDF_ERROR_ASSERT(esp_wifi_start());//启动 wifi
    /*****mwifi_init_config_t cfg = MWIFI_INIT_CONFIG_DEFAULT(); //分配 wifi mesh 结构体*****
    mwifi_init_config_t cfg = MWIFI_INIT_CONFIG_DEFAULT(); //分配 wifi mesh 结构体
    mwifi_config_t config = {
        .router_ssid = "metasia",
        .router_password = "metasia0902",
        .mesh_id = "12345",
        .mesh_password = "87654321", //mesh 密码一定要超过 8 位，不然系统会不断重启报 mwifi_set_config 错误
    };
    MDF_ERROR_ASSERT(mdf_event_loop_init(event_loop_cb)); //mesh 网络事件回调
    MDF_ERROR_ASSERT(mwifi_init(&cfg)); //初始化 wifi mesh
    MDF_ERROR_ASSERT(mwifi_set_config(&config)); //配置 wifi mesh 网络
    MDF_ERROR_ASSERT(mwifi_start()); //启动 wifi mesh 网络
}
```

判断该设备是不是根节点，判断后要做相应处理，我还没有做相应处理，下一节再做

发

算上父节点

误

```

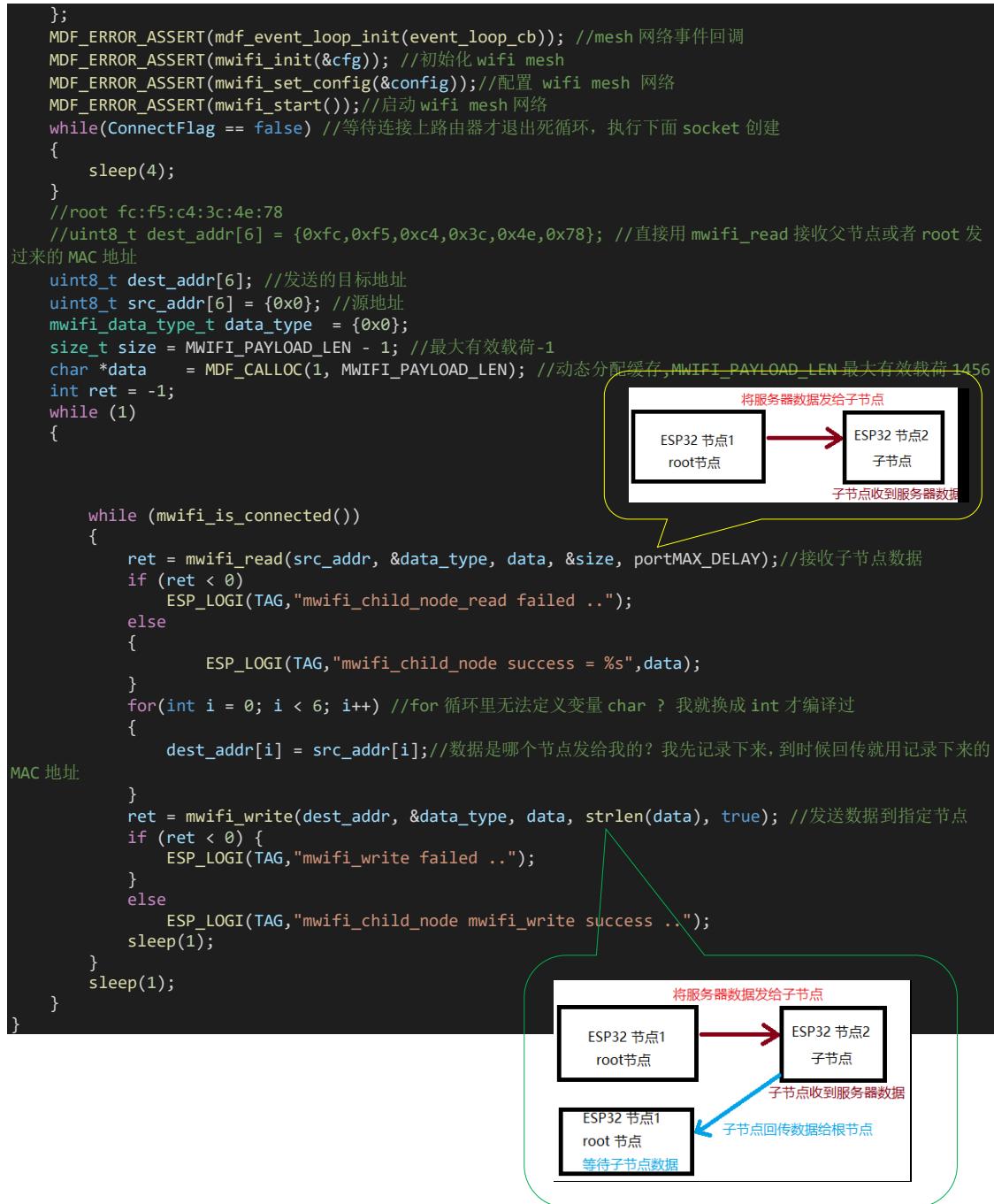
while(ConnectFlag == false) //等待连接上路由器才退出死循环，执行下面 socket 创建
{
    sleep(4);
}
int sockfd = -1;
struct sockaddr_in server_addr = { //连接服务器的 IP 和端口号
    .sin_family = AF_INET,
    .sin_port = htons(50395),
    .sin_addr.s_addr = inet_addr("192.168.1.69"),
};
sockfd = socket(AF_INET, SOCK_STREAM, 0); //如果是 root 根节点就必须用 socket 连接服务器
if(sockfd < 0)
{
    ESP_LOGI(TAG,"socket Creat failed ..");
    close(sockfd);
}
else
    ESP_LOGI(TAG,"socket Creat success ..");
int ret = connect(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr_in)); //连接服务
器
if (ret != 0)
{
    ESP_LOGI(TAG,"TCP server connect failed ..");
    close(sockfd);
    return;
}
mwifi_data_type_t data_type = {0x0};
char *data = MDF_CALLOC(1, MWIFI_PAYLOAD_LEN); //动态分配缓存,MWIFI_PAYLOAD_LEN 最大有效载荷 1456
size_t size = MWIFI_PAYLOAD_LEN - 1; //最大有效载荷-1
uint8_t src_addr[6] = {0x0}; //源地址
char rx_buffer[128]; //接收服务器发来的数据
uint8_t dest_addr[6] = {0xfc,0xf5,0xc4,0x3c,0x4a,0x1c}; //子节点 MAC 地址，也可以叫服务器在数据包里面嵌
入子节点 MAC 地址，我在根节点将 MAC 地址解析出来，放入 dest_addr[6]
while (1)
{
    while (_m wifi_is_connected()) //查询本根节点是否连接上路由器
    {
        memset(rx_buffer, 0, 128); //清除上一次服务器数据，不然因为数组太大，字符后面跟着乱码
        ret = recv(sockfd, rx_buffer, sizeof(rx_buffer) - 1, 0); //接收这一次服务器数据
        if (ret < 0) {
            ESP_LOGI(TAG,"root recv server failed ..");
        }
        else
            ESP_LOGI(TAG,"root recv server success = %s",rx_buffer);
        ret = mwifi_write(dest_addr, &data_type, rx_buffer, strlen(rx_buffer), true); //发送数据
到指定节点
        if (ret < 0) {
            ESP_LOGI(TAG,"mwifi_write failed ..");
        }
    }
    memset(data, 0, MWIFI_PAYLOAD_LEN);
    ret = mwifi_read(src_addr, &data_type, data, &size, portMAX_DELAY); //接收子节点数据
    if (ret < 0)
        ESP_LOGI(TAG,"mwifi_root_read failed ..");
    else
        ESP_LOGI(TAG,"mwifi_root_read success ..");
    ret = write(sockfd, data, size); //将子节点数据发送给服务器
    if (ret < 0)
        ESP_LOGI(TAG,"write server failed ..");
    else
        ESP_LOGI(TAG,"write server success ..");
    }
    sleep(1);
}

```

子节点例程

```
#include "mdf_common.h"
#include "sys/unistd.h"
#include "nvs.h"
#include "mwifi.h" //在 MDF 库使用中 该头文件包含了 esp wifi 和 mdf wifi

static const char *TAG = "xzzMeshNode2";
bool ConnectFlag = false; //什么时候创建 socket 标志位
static mdf_err_t event_loop_cb(mdf_event_loop_t event, void *ctx)
{
    mesh_addr_t MAC;
    bool RootFlag = false;
    switch (event) {
        case MDF_EVENT_MWIFI_STARTED:
            MDF_LOGI("MESH is started");
            break;
        case MDF_EVENT_MWIFI_PARENT_CONNECTED: //与父节点连接上事件触发
            MDF_LOGI("Parent is connected success");
            MDF_ERROR_ASSERT(esp_mesh_get_parent_bssid(&MAC)); //获取父节点的 BSSID 也就是父节点 MAC
            ESP_LOGI(TAG, "parent MAC = %x:%x:%x:%x:%x", MAC.addr[0], MAC.addr[1], MAC.addr[2], MAC.addr[3], MAC.addr[4], MAC.addr[5]);
            if((RootFlag = esp_mesh_is_root())) //判断本设备是不是根节点
            {
                MDF_LOGI("device root node");
                ConnectFlag = true;
            }
            else
            {
                MDF_LOGI("device sub node");
                ConnectFlag = true;
            }
            break;
        case MDF_EVENT_MWIFI_PARENT_DISCONNECTED: //与父节点断开，事件触发，当然父节点发现子节点断开也会触发
            MDF_LOGI("Parent is disconnected");
            break;
        case MDF_EVENT_MWIFI_ROUTING_TABLE_ADD: //如果本设备为父节点，子节点连接上该设备会触发该路由表事件
            MDF_LOGI("Subnode add");
        case MDF_EVENT_MWIFI_ROUTING_TABLE_REMOVE:
            MDF_LOGI("total_num: %d", esp_mesh_get_total_node_num()); //返回连接本设备父节点的子节点数，算上父节点
            break;
        case MDF_EVENT_MWIFI_ROOT_GOT_IP:
            MDF_LOGI("Root obtains the IP address. It is posted by LwIP stack automatically");
            break;
        default:
            break;
    }
    return MDF_OK;
}
void app_main()
{
    esp_err_t err = nvs_flash_init(); //记住在做 wifi 功能时，一定要对 nvs_flash 初始化，不然板子会一直重启
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );
    /*配置 esp IDF 库的 wifi*/
    tcpip_adapter_init();
    wifi_init_config_t IDFCfg = WIFI_INIT_CONFIG_DEFAULT(); //分配 wifi 初始化结构体
    MDF_ERROR_ASSERT(esp_event_loop_init(NULL, NULL)); //wifi 事件回调函数不用
    MDF_ERROR_ASSERT(esp_wifi_init(&IDFCfg)); //wifi 初始化
    MDF_ERROR_ASSERT(esp_wifi_set_mode(WIFI_MODE_STA)); //wifi 先启动为 STA 模式，如果 mesh 网络有根节点自动换成节点模式
    MDF_ERROR_ASSERT(esp_wifi_start()); //启动 wifi
    //*****mwifi_set_config()*****//分配 wifi mesh 结构体
    mwifi_init_config_t cfg = MWIFI_INIT_CONFIG_DEFAULT(); //分配 wifi mesh 结构体
    mwifi_config_t config = {
        .router_ssid = "metasia",
        .router_password = "metasia0902",
        .mesh_id = "12345",
        .mesh_password = "87654321", //mesh 密码一定要超过 8 位，不然系统会不断重启报 mwifi_set_config 错误
    }
}
```



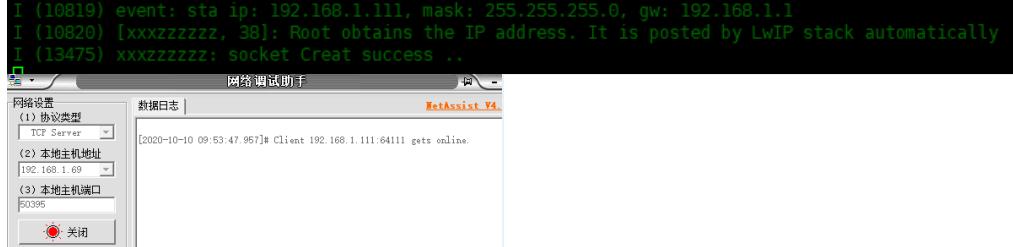
也就是根节点代码中如下一段：

```

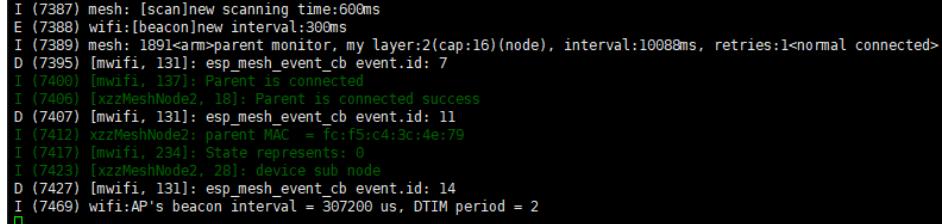
ret = write(sockfd, data, size); //将子节点数据发送给服务器
if (ret < 0)
    ESP_LOGI(TAG, "write server failed ..");
else
    ESP_LOGI(TAG, "write server success ..");

```

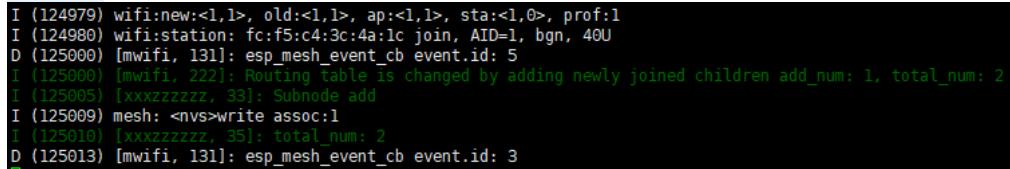
因为是根节点，所以根节点先启动，连接上网络调试助手，等待子节点连接。

```
I (10036) [mwifi, 137]: Parent is connected
I (10039) [xxxxxxxx, 17]: Parent is connected success
D (10040) [mwifi, 131]: esp_mesh_event_cb event.id: 11
I (10046) [xxxxxxxx, 21]: device root node
I (10051) [mwifi, 234]: State represents: 0
D (10061) [mwifi, 131]: esp_mesh_event_cb event.id: 14
D (10081) [mwifi, 131]: esp_mesh_event_cb event.id: 17
I (10819) event: sta ip: 192.168.1.111, mask: 255.255.255.0, gw: 192.168.1.1
I (10820) [xxxxxxxx, 38]: Root obtains the IP address. It is posted by LwIP stack automatically
I (13475) xxxxxxxx: socket Creat success ..

```

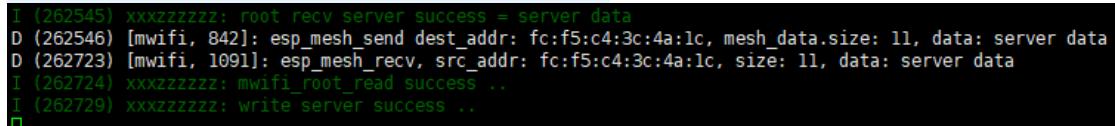
子节点连接根节点

```
I (7387) mesh: [scan]new scanning time:600ms
E (7388) wifi:[beacon]new interval:300ms
I (7389) mesh: 1891<-->parent monitor, my layer:2(cap:16)(node), interval:10088ms, retries:1<normal connected>
D (7395) [mwifi, 131]: esp_mesh_event_cb event.id: 7
I (7400) [mwifi, 137]: Parent is connected
I (7406) [xzzMeshNode2, 18]: Parent is connected success
D (7407) [mwifi, 131]: esp_mesh_event_cb event.id: 11
I (7412) xzzMeshNode2: parent MAC = fc:f5:c4:3c:4e:79
I (7417) [mwifi, 234]: State represents: 0
I (7423) [xzzMeshNode2, 28]: device sub node
D (7427) [mwifi, 131]: esp_mesh_event_cb event.id: 14
I (7469) wifi:AP's beacon interval = 307200 us, DTIM period = 2

```

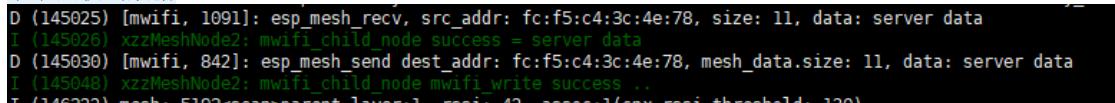
根节点会输出子节点的 MAC 地址，记住是子节点 STA 的 MAC 地址，STA 模式的 MAC 地址就算本模块的地址。

```
I (124979) wifi:new:<1,1>, old:<1,1>, ap:<1,1>, sta:<1,0>, prof:1
I (124980) wifi:station: fc:f5:c4:3c:4a:lc join, AID=1, bgn, 40U
D (125000) [mwifi, 131]: esp_mesh_event_cb event.id: 5
I (125000) [mwifi, 222]: Routing table is changed by adding newly joined children add_num: 1, total_num: 2
I (125005) [xxxxxxxx, 33]: Subnode add
I (125009) mesh: <nvs>write assoc1
T (125010) [xxxxxxxx, 35]: total_num: 2
D (125013) [mwifi, 131]: esp_mesh_event_cb event.id: 3

```

服务器向根节点发数据，要求根节点把数据传给子节点。

```
I (262545) xxxxxxxx: root recv server success = server data
D (262546) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4a:lc, mesh_data.size: 11, data: server data
D (262723) [mwifi, 1091]: esp_mesh_recv, src_addr: fc:f5:c4:3c:4a:lc, size: 11, data: server data
I (262724) xxxxxxxx: mwifi_root_read success ..
I (262729) xxxxxxxx: write server success ..

```

子节点收到数据

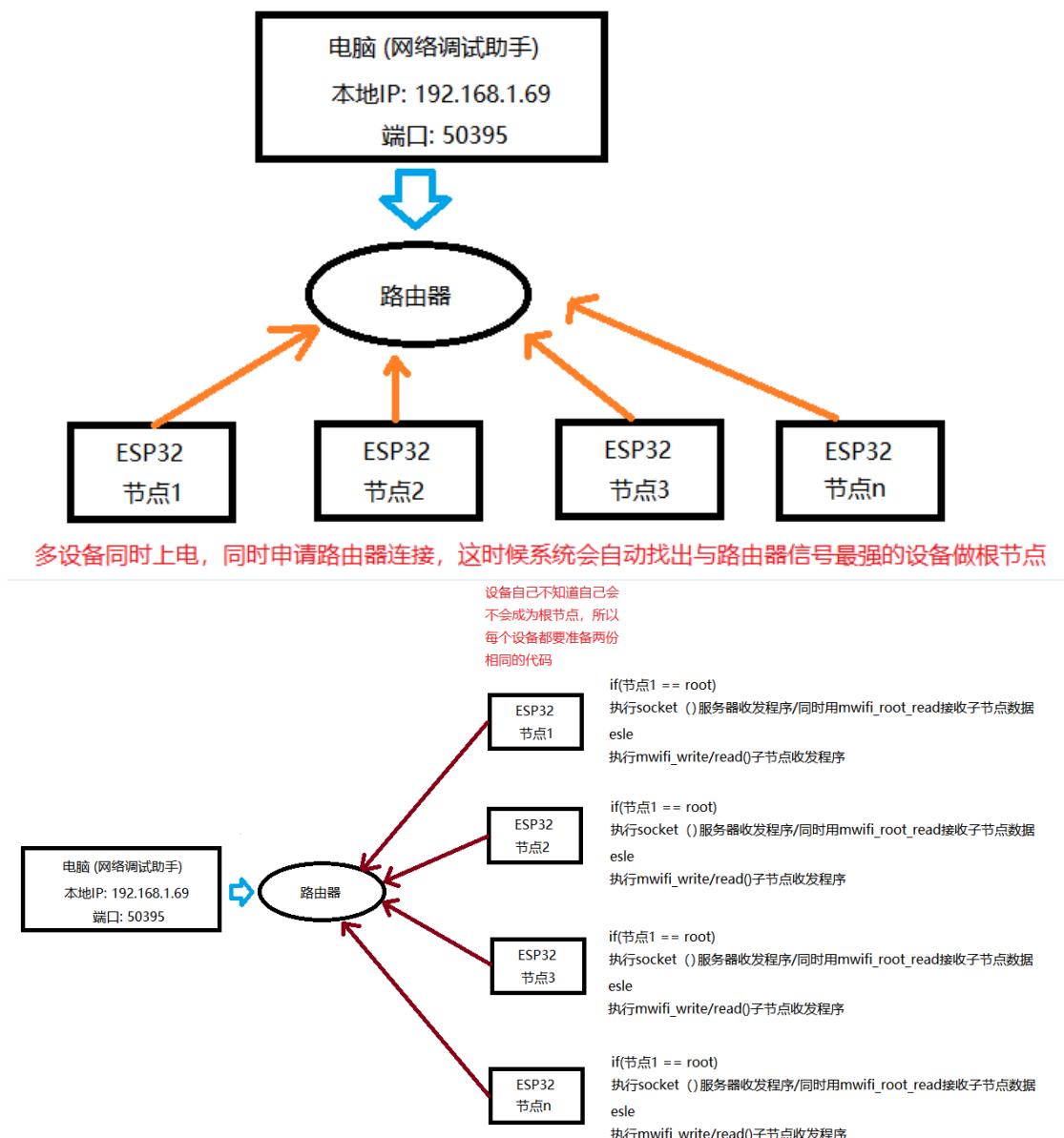
```
D (145025) [mwifi, 1091]: esp_mesh_recv, src_addr: fc:f5:c4:3c:4e:78, size: 11, data: server data
I (145026) xzzMeshNode2: mwifi_child_node success = server data
D (145030) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4e:78, mesh_data.size: 11, data: server data
I (145048) xzzMeshNode2: mwifi_child_node mwifi_write success ..
I (146222) mesh: E102 receive parent layer1_recv(42) success(say reci threshold: 120)

```

子节点将数据经过根节点回传到服务器



这就是整个过程

多个子节点设备同时启动，那么都有成为根节点的可能，那么如何相互通信呢？



`mwifi_root_read(src_addr, data_type, data, size, wait_ticks) //根节点读取子节点数据`

这是我上一节得代码片段，这是根节点

```
while (1)
{
    while (mwifi_is_connected()) //查询本根节点是否连接上路由器
    {
        memset(rx_buffer, 0, 128); //清除上一次服务器数据，不然因为数组太大，字符后面跟着乱码
        ret = recv(sockfd, rx_buffer, sizeof(rx_buffer) - 1, 0); //接收这一次服务器数据
        if (ret < 0) {
            ESP_LOGI(TAG, "root recv server failed ..");
        }
        else
            ESP_LOGI(TAG, "root recv server success = %s", rx_buffer);
        ret = mwifi_write(dest_addr, &data_type, rx_buffer, strlen(rx_buffer), true); //发送数据到指定节点
        if (ret < 0) {
            ESP_LOGI(TAG, "mwifi_write failed ..");
        }
        memset(data, 0, MWIFI_PAYLOAD_LEN);
        ret = mwifi_read(src_addr, &data_type, data, &size, portMAX_DELAY); //接收子节点数据
        if (ret < 0)
            ESP_LOGI(TAG, "mwifi_root_read failed ..");
        else
            ESP_LOGI(TAG, "mwifi_root_read success ..");
        ret = write(sockfd, data, size); //将子节点数据发送给服务器
        if (ret < 0)
            ESP_LOGI(TAG, "write server failed ..");
        else
            ESP_LOGI(TAG, "write server success ..");
    }
    sleep(1);
}
```

如果子节点得 mwifi_write(NULL, ...) 目的地地址填写得 NULL，那么 mwifi_read 就无法使用，因为 NULL 是直接要求传根节点

子节点发送 mesh 数据包，填入得目的地址是 NULL

```
ret = mwifi_write(NULL, &data_type, data, strlen(data), true); //发送数据到指定节点
if (ret < 0) {
    ESP_LOGI(TAG, "mwifi_write failed ..");
}
else
    ESP_LOGI(TAG, "mwifi_child_node mwifi_write success ..");
```

```
D (17871) [mwifi, 1091]: esp_mesh_recv, src_addr: fc:f5:c4:3c:4e:78, size: 11, data: server data
I (17872) xzzMeshNode2: mwifi_child_node success = server data
D (17876) [mwifi, 842]: esp_mesh_send dest_addr: ff:00:00:01:00:00, mesh_data.size: 11, data: server data
I (17891) xzzMeshNode2: mwifi_child_node mwifi_write success ..
```

当子节点收到网络数据包后

```
I (13517) XXXXXXXX: socket Create success ..
I (20671) XXXXXXXX: root recv server success = server data
D (20672) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4a:lc, mesh_data.size: 11, data: server data
```

根节点没有收到子节点得回传



将根节点的 mesh 网络接收函数换成 mwifi_root_read，那么子节点就算用 mwifi_write(NULL,...)，根节点也能收到数据转发给服务器

```
ret = mwifi_root_read(src_addr, &data_type, data, &size, portMAX_DELAY); //接收子节点  
数据  
if (ret < 0)  
    ESP_LOGI(TAG, "mwifi_root_read failed ..");  
else  
    ESP_LOGI(TAG, "mwifi_root_read success ..");
```

```
I (35416) xzzMeshNode2: mwifi_child_node mwifi_write success ..  
D (36417) [mwifi, 1091]: esp_mesh_recv, src_addr: fc:f5:c4:3c:4e:78, size: 11, data: server data  
I (36418) xzzMeshNode2: mwifi_child_node success = server data  
D (36422) [mwifi, 842]: esp_mesh_send dest_addr: ff:00:00:01:00:00, mesh_data.size: 11, data: server data  
I (36436) xzzMeshNode2: mwifi_child_node mwifi_write success ..  
D (37438) [mwifi, 1091]: esp_mesh_recv, src_addr: fc:f5:c4:3c:4e:78, size: 11, data: server data  
T (37439) xzzMeshNode2: mwifi_child_node success = server data  
D (37443) [mwifi, 842]: esp_mesh_send dest_addr: ff:00:00:01:00:00, mesh_data.size: 11, data: server data  
I (37459) xzzMeshNode2: mwifi_child_node mwifi_write success ..  
D (38459) [mwifi, 1091]: esp_mesh_recv, src_addr: fc:f5:c4:3c:4e:78, size: 11, data: server data  
T (38460) xzzMeshNode2: mwifi_child_node success = server data
```

子节点流畅的发数据

```
D (50168) [mwifi, 1351]: esp_mesh_recv_toDS, src_addr: fc:f5:c4:3c:4e:1c, size: 11, data: server data  
I (50169) XXXXXXXX: mwifi_root_read success ..  
I (50174) XXXXXXXX: write server success ..  
I (50178) XXXXXXXX: root recv server success = server data  
D (50183) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4a:1c, mesh_data.size: 11, data: server data  
D (51188) [mwifi, 1351]: esp_mesh_recv_toDS, src_addr: fc:f5:c4:3c:4a:1c, size: 11, data: server data  
I (51189) XXXXXXXX: mwifi_root_read success ..  
T (51194) XXXXXXXX: write server success ..  
I (51198) XXXXXXXX: root recv server success = server data  
D (51203) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4a:1c, mesh_data.size: 11, data: server data  
D (52211) [mwifi, 1351]: esp_mesh_recv_toDS, src_addr: fc:f5:c4:3c:4a:1c, size: 11, data: server data  
I (52211) XXXXXXXX: mwifi_root_read success ..  
I (52217) XXXXXXXX: write server success ..  
T (52220) XXXXXXXX: root recv server success = server data  
D (52226) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4a:1c, mesh_data.size: 11, data: server data  
D (53228) [mwifi, 1351]: esp_mesh_recv_toDS, src_addr: fc:f5:c4:3c:4a:1c, size: 11, data: server data  
I (53229) XXXXXXXX: mwifi_root_read success ..  
I (53234) XXXXXXXX: write server success ..  
I (53238) XXXXXXXX: root recv server success = server data
```

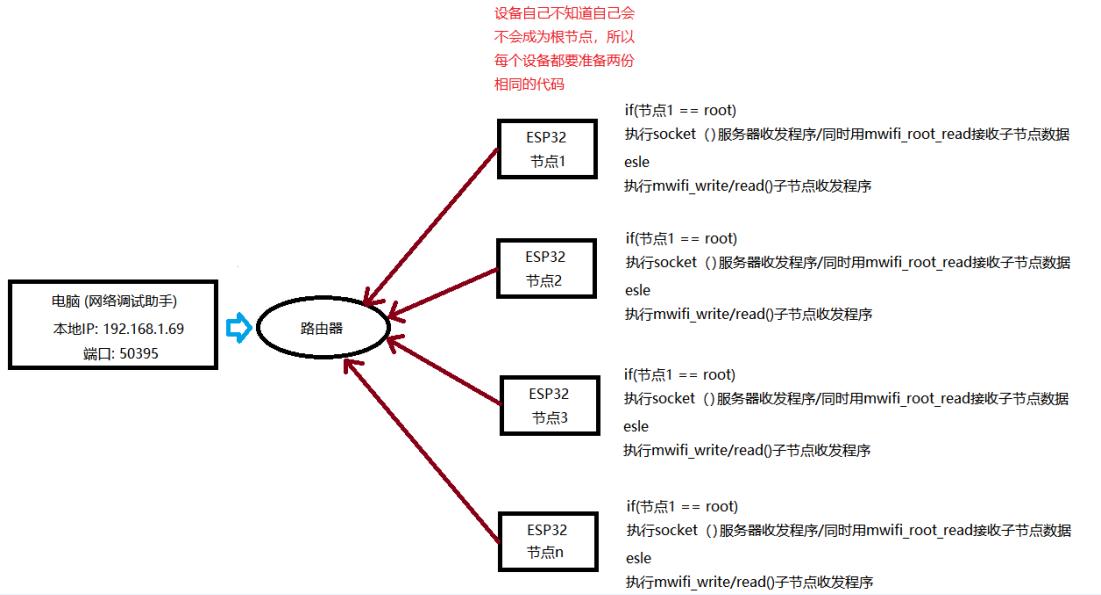
根节点流畅的转发

```
server data  
[2020-10-10 16:06:37.911]# RECV ASCII FROM 192.168.1.128 :55310>  
server data  
  
[2020-10-10 16:06:39.624]# SEND ASCII TO ALL>  
server data  
  
[2020-10-10 16:06:39.683]# RECV ASCII FROM 192.168.1.128 :55310>  
server data  
  
[2020-10-10 16:06:40.504]# SEND ASCII TO ALL>  
server data  
  
[2020-10-10 16:06:40.696]# RECV ASCII FROM 192.168.1.128 :55310>  
server data  
  
[2020-10-10 16:06:41.591]# SEND ASCII TO ALL>  
server data  
  
[2020-10-10 16:06:41.717]# RECV ASCII FROM 192.168.1.128 :55310>
```

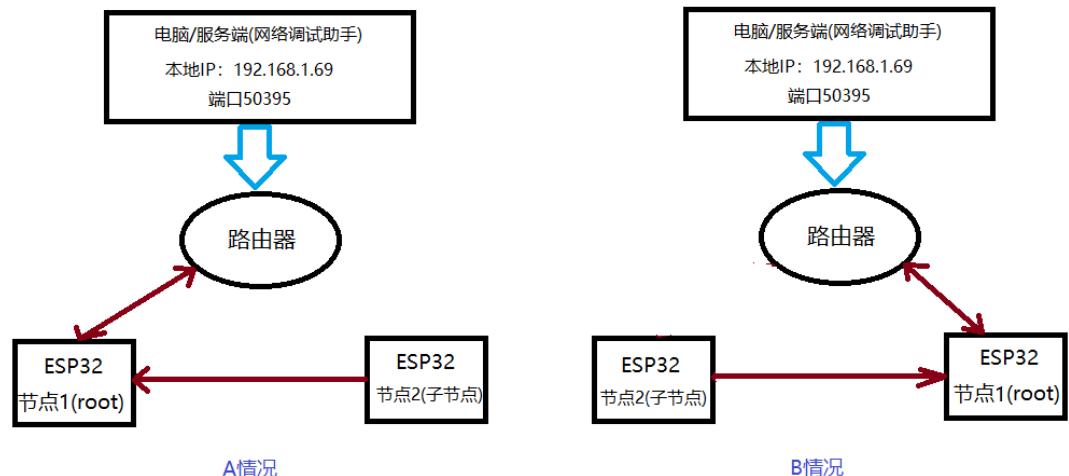
网络调试助手发一次，就能收到一次

所以子节点向给服务器发数据的时候，根节点地址可以填写 NULL，让根节点帮你转发。

有了以上根节点自动识别的知识之后，现在可以完成多个设备同时联网，自动变成根节点的网络转包应用了。



我手上只有两块设备，下面进行试验



所有不知道某块板子是root的情况下，我就写一份带主节点和子节点的代码就行了，然后用MAC地址去寻找设备

代码例程：一份代码两个程序使用

```
#include "mdf_common.h"
#include "sys/unistd.h"
#include "nvs.h"
#include "mwifi.h" //在 MDF 库使用中 该头文件包含了 esp wifi 和 mdf wifi

static const char *TAG = "xxxxxxxx";
bool RootFlag = false; //根节点标志
bool childFlag = false; //子节点标志
bool NodeConnectFlag = false; //节点连接标志
static mdf_err_t event_loop_cb(mdf_event_loop_t event, void *ctx)
{
    //RootFlag = false; 
    switch (event) {
        case MDF_EVENT_MWIFI_STARTED:
            MDF_LOGI("MESH is started");
            break;
        case MDF_EVENT_MWIFI_PARENT_CONNECTED: //与父节点连接上事件触发
            MDF_LOGI("Parent is connected success");
            RootFlag = esp_mesh_is_root(); //获取设备是不是根节点
            if(RootFlag == true) //判断本设备是不是根节点
            {
                MDF_LOGI("device root node");
                RootFlag = true; //是根节点
            }
            else
            {
                childFlag = true; //非根节点，子节点
                MDF_LOGI("device sub node");
            }
            NodeConnectFlag = true; //连接成功
            break;
        case MDF_EVENT_MWIFI_PARENT_DISCONNECTED: //与父节点断开，事件触发，当然父节点发现子节点断开也会触发
            MDF_LOGI("Parent is disconnected");
            break;
        case MDF_EVENT_MWIFI_ROUTING_TABLE_ADD: //如果本设备为父节点，子节点连接上该设备会触发该路由表事件
            MDF_LOGI("Subnode add");
        case MDF_EVENT_MWIFI_ROUTING_TABLE_REMOVE:
            MDF_LOGI("total_num: %d", esp_mesh_get_total_node_num()); //返回连接本设备父节点的子节点数，算上父节点
            break;
        case MDF_EVENT_MWIFI_ROOT_GOT_IP:
            MDF_LOGI("Root obtains the IP address. It is posted by LwIP stack automatically");
            break;
        default:
            break;
    }
    return MDF_OK;
}
void app_main()
{
    esp_err_t err = nvs_flash_init(); //记住在做 wifi 功能时，一定要对 nvs_flash 初始化，不然板子会一直重启
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );
    /*配置 esp IDF 库 的 wifi*/
}
```

事件回调函数中尽量不要定义变量，因为系统会回调很多次，不知不觉得把你变量初始值改成定义时候的默认值

不管哪个设备开机后都会进入 PARENT_CONNECTED 事件然后获取是不是根节点，是根节点将标识置 1。

```

tcpip_adapter_init();
wifi_init_config_t IDFcfg = WIFI_INIT_CONFIG_DEFAULT(); //分配 wifi 初始化结构体
MDF_ERROR_ASSERT(esp_event_loop_init(NULL, NULL)); //wifi 事件回调函数不用
MDF_ERROR_ASSERT(esp_wifi_init(&IDFcfg)); //wifi 初始化
MDF_ERROR_ASSERT(esp_wifi_set_mode(WIFI_MODE_STA)); //wifi 先启动为 STA 模式, 如果
mesh 网络有根节点自动换成节点模式
MDF_ERROR_ASSERT(esp_wifi_start()); //启动 wifi
/****************************************/
mwifi_init_config_t cfg = MWIFI_INIT_CONFIG_DEFAULT(); //分配 wifi mesh 结构体
mwifi_config_t config = {
    .router_ssid = "metasia",
    .router_password = "metasia0902",
    .mesh_id = "12345",
    .mesh_password = "87654321", //mesh 密码一定要超过 8 位, 不然系统会不断重启报
mwifi_set_config 错误
};
MDF_ERROR_ASSERT(mdf_event_loop_init(event_loop_cb)); //mesh 网络事件回调
MDF_ERROR_ASSERT(mwifi_init(&cfg)); //初始化 wifi mesh
MDF_ERROR_ASSERT(mwifi_set_config(&config)); //配置 wifi mesh 网络
MDF_ERROR_ASSERT(mwifi_start()); //启动 wifi mesh 网络
while(NodeConnectFlag == false) //如果连接不成功, 不是根节点或者子节点, 死循环
{
    sleep(4);
}

ESP_LOGI(TAG, "break while .. RootFlag = %d", RootFlag);
mwifi_data_type_t data_type = {0x0};
char *data = MDF_CALLOC(1, MWIFI_PAYLOAD_LEN); //动态分配缓
存, MWIFI_PAYLOAD_LEN 最大有效载荷 1456
size_t size = MWIFI_PAYLOAD_LEN - 1; //最大有效载荷-1
uint8_t src_addr[6] = {0x0}; //源地址
char rx_buffer[128]; //接收服务器发来的数据
//uint8_t dest_addr[6] = {0xfc,0xf5,0xc4,0x3c,0x4a,0x1c}; //子节点 MAC 地址
uint8_t dest_addr[6] = {0xfc,0xf5,0xc4,0x3c,0x4e,0x78}; //子节点 MAC 地址
int sockfd = -1, ret = -1;
if(RootFlag == true) //如果该设备是根节点
{
    ESP_LOGI(TAG, "root root root root");
    struct sockaddr_in server_addr = { //连接服务器的 IP 和端口号
        .sin_family = AF_INET,
        .sin_port = htons(50395),
        .sin_addr.s_addr = inet_addr("192.168.1.69"),
    };
    sockfd = socket(AF_INET, SOCK_STREAM, 0); //如果是 root 根节点就必须用 socket 连
接服务器
    if(sockfd < 0)
    {
        ESP_LOGI(TAG, "socket Creat failed ..");
        close(sockfd);
    }
    else
        ESP_LOGI(TAG, "socket Creat success ..");
    ret = connect(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr_in)); //连接服务器
    if (ret != 0)
    {
        ESP_LOGI(TAG, "TCP server connect failed ..");
        close(sockfd);
        return;
    }
    while (1)
    {
}

```

如果是 A 设备为根节点, 那么你就要填入 B 设备的 MAC 地址
项目中设备 MAC 由服务器下发

如果是 B 设备为根节点, 那么你就要填入 A 设备的 MAC 地址, 项目中设备 MAC 由服务器下发

根节点执行 if 为真这段程序

```

        while (mwifi_is_connected()) //查询本根节点是否连接上路由器
    {
        memset(rx_buffer, 0, 128); //清除上一次服务器数据,不然因为数组太大,字符后面跟着乱码
        ret = recv(sockfd, rx_buffer, sizeof(rx_buffer) - 1, 0); //接收这一次服务器数据
        if (ret < 0) {
            ESP_LOGI(TAG,"root recv server failed ..");
        }
        else
            ESP_LOGI(TAG,"root recv server success = %s",rx_buffer);
        ret = mwifi_write(dest_addr, &data_type, rx_buffer, strlen(rx_buffer), true); //发送数据到指定节点
        if (ret < 0)
            ESP_LOGI(TAG,"mwifi_write failed ..");
        }
        memset(data, 0, MWIFI_PAYLOAD_LEN);
        ret = mwifi_root_read(src_addr, &data_type, data, &size, portMAX_DELAY); //接收子节点数据
        if (ret < 0)
            ESP_LOGI(TAG,"mwifi_root_read failed ..");
        else
            ESP_LOGI(TAG,"mwifi_root_read success ..");
        ret = write(sockfd, data, size); //将子节点数据发送给服务器
        if (ret < 0)
            ESP_LOGI(TAG,"write server failed ..");
        else
            ESP_LOGI(TAG,"write server success ..");
        }
        sleep(1);
    }
else //如果该设备是子节点
{
    ESP_LOGI(TAG,"chiled chiled chiled");
    while (1)
    {

        while (mwifi_is_connected())
    {
        ret = mwifi_read(src_addr, &data_type, data, &size, portMAX_DELAY); //接收子节点数据
        if (ret < 0)
            ESP_LOGI(TAG,"mwifi_child_node_read failed ..");
        else
        {
            ESP_LOGI(TAG,"mwifi_child_node success = %s",data);
        }
        /* 直接取消根节点的 MAC 地址, 因为用的 NULL*/
        ret = mwifi_write(NULL, &data_type, data, strlen(data), true); //发送数据到指定节点
        if (ret < 0) {
            ESP_LOGI(TAG,"mwifi_write failed ..");
        }
        else
            ESP_LOGI(TAG,"mwifi_child_node mwifi_write success ..");
        sleep(1);
    }
}
}

```

子节点执行 else 程序

节点 1 为 root

```
I (10041) [xxxxxxxx, 22]: device root node
I (10046) [mwifi, 234]: State represents: 0
D (10055) [mwifi, 131]: esp_mesh_event_cb event.id: 14
D (10819) [mwifi, 131]: esp_mesh_event_cb event.id: 17
I (10821) event: sta ip: 192.168.1.124, mask: 255.255.255.0, gw: 192.168.1.1
I (10822) [xxxxxxxx, 41]: Root obtains the IP address. It is posted by LwIP stack automatically
I (12045) wifi:new:<6,1>, old:<6,1>, ap:<6,0>, sta:<6,0>, prof:6
I (12046) wifi:station: fc:f5:c4:3c:4e:78 join, AID=1, bgn, 40U
D (12067) [mwifi, 131]: esp_mesh_event_cb event.id: 5
I (12067) [mwifi, 222]: Routing table is changed by adding newly joined children add_num: 1, total_num: 2
I (12072) [xxxxxxxx, 36]: Subnode add
I (12075) mesh: <nvs>write assoc:1
I (12077) [xxxxxxxx, 38]: total_num: 2
D (12080) [mwifi, 131]: esp_mesh_event_cb event.id: 3
I (13480) xxxxxxxx: break while .. RootFlag = 1
I (13480) xxxxxxxx: root root root root
I (13481) xxxxxxxx: socket Creat success ..
I (17613) xxxxxxxx: root recv server success = server data
D (17614) [mwifi, 842]: esp_mesh_send dest_addr: fc:f5:c4:3c:4e:78, mesh_data.size: 11, data: server data
D (17731) [mwifi, 1351]: esp_mesh_recv_toS, src_addr: fc:f5:c4:3c:4e:78, size: 11, data: server data
I (17732) xxxxxxxx: mwifi_root_read success ..
I (17736) xxxxxxxx: write server success ..
I (22427) xxxxxxxx: root recv server success = server data
```

节点 2 为子节点

```
I (8701) [mwifi, 137]: Parent is connected
I (8706) [xxxxxxxx, 18]: Parent is connected success
D (8707) [mwifi, 131]: esp_mesh_event_cb event.id: 11
I (8712) [xxxxzzzz, 28]: device sub node
I (8717) [mwifi, 234]: State represents: 0
D (8726) [mwifi, 131]: esp_mesh_event_cb event.id: 14
I (8740) wifi:AP's beacon interval = 307200 us, DTIM period = 2
I (9483) xxxxxxxx: break while .. RootFlag = 0
I (9483) xxxxxxxx: chiled chiled chiled
I (13989) mesh: 4448<active>parent layer:1(node), channel:6, rssi:-44, assoc:0(cnxx rsssi threshold:-120)my_i
D (14257) [mwifi, 1091]: esp_mesh_recv, src_addr: fc:f5:c4:3c:4a:1c, size: 11, data: server data
I (14258) xxxxxxxx: mwifi_child_node success = server data
D (14262) [mwifi, 842]: esp_mesh_send dest_addr: ff:00:00:01:00:00, mesh_data.size: 11, data: server data
I (14286) xxxxxxxx: mwifi_child_node mwifi_write success ..
I (15492) mesh: 5193<scan>parent layer:1, rsssi:-45, assoc:1(cnxx rsssi threshold:-120)
I (15493) mesh: [SCAN][ch:6]AP:1, other(ID:0, RD:0), MAP:1, idle:0, candidate:1, root:1, topMAP:0[c:2,i:2]
I (15501) mesh: 6579[weak]try rsssi_threshold:-120, backoff times:0, max:5<-78,-82,-85>
I (15509) mesh: 616[monitor]no change, parent:fc:f5:c4:3c:4a:1d, rsssi:-45
I (15515) mesh: 1891<arm>parent monitor, my layer:2(cap:16)(node), interval:9492ms, retries:2<>
D (18993) [mwifi, 1091]: esp_mesh_recv, src_addr: fc:f5:c4:3c:4a:1c, size: 11, data: server data
I (18994) xxxxxxxx: mwifi_child_node success = server data
D (18998) [mwifi, 842]: esp_mesh_send dest_addr: ff:00:00:01:00:00, mesh_data.size: 11, data: server data
I (19014) xxxxxxxx: mwifi_child_node mwifi_write success ..
D (21520) [mwifi, 1091]: esp_mesh_recv, src_addr: fc:f5:c4:3c:4a:1c, size: 11, data: server data
I (21521) xxxxxxxx: mwifi_child_node success = server data
D (21525) [mwifi, 842]: esp_mesh_send dest_addr: ff:00:00:01:00:00, mesh_data.size: 11, data: server data
I (21548) xxxxxxxx: mwifi_child_node mwifi_write success ..
```



节点1为子节点

```
I (7420) [mwifi, 137]: Parent is connected
I (7426) [xxxxzzzz, 18]: Parent is connected success
D (7427) [mwifi, 131]: esp_mesh_event_cb event.id: 11
I (7432) [xxxxzzzz, 28]: device sub node
I (7437) [mwifi, 234]: State represents: 0
D (7445) [mwifi, 131]: esp_mesh_event_cb event.id: 14
I (9477) xxxxzzzz: break while .. RootFlag = 0
I (9477) xxxxzzzz: chiled chiled chiled
```

节点2为根节点

```
I (10822) event: sta ip: 192.168.1.125, mask: 255.255.255.0, gw: 192.168.1.1
I (10823) [xxxxzzzz, 41]: Root obtains the IP address. It is posted by LwIP stack automatically
I (13480) xxxxzzzz: break while .. RootFlag = 1
I (13480) xxxxzzzz: root root root root
I (13481) xxxxzzzz: socket Creat success ..
```



数据正常收发

子节点如何知道根节点与服务器已经连接成功？

`mdf_err_t mwifi_post_root_status(bool status)` //根节点发送标志,其实其它节点也可以发送
`status:` 可以写 True, 或者 false

`bool mwifi_get_root_status()` //子节点接收根节点的标志
返回, `mwifi_post_root_status` 发送的 True 或者 false

在上一章节的代码基础上更改

```
while(NodeConnectFlag == false) //如果连接不成功, 不是根节点或者子节点, 死循环
{
    sleep(4);
}
ESP_LOGI(TAG, "break while .. RootFlag = %d", RootFlag);
mwifi_data_type_t data_type = {0x0};
char *data = MDF_CALLOC(1, MWIFI_PAYLOAD_LEN); //动态分配缓存,MWIFI_PAYLOAD_LEN 最大有效载荷 1456
size_t size = MWIFI_PAYLOAD_LEN - 1; //最大有效载荷-1
uint8_t src_addr[6] = {0x0}; //源地址
char rx_buffer[128]; //接收服务器发来的数据
//uint8_t dest_addr[6] = {0xfc,0xf5,0xc4,0x3c,0x4a,0x1c}; //子节点 MAC 地址
uint8_t dest_addr[6] = {0xfc,0xf5,0xc4,0x3c,0x4e,0x78}; //子节点 MAC 地址
int sockfd = -1, ret = -1;
if(RootFlag == true) //如果该设备是根节点
{
    ESP_LOGI(TAG, "root root root root");
    struct sockaddr_in server_addr = { //连接服务器的 IP 和端口号
        .sin_family = AF_INET,
        .sin_port = htons(50395),
        .sin_addr.s_addr = inet_addr("192.168.1.69"),
    };
    sockfd = socket(AF_INET, SOCK_STREAM, 0); //如果是 root 根节点就必须用 socket 连接服务器
    if(sockfd < 0)
    {
        ESP_LOGI(TAG, "socket Creat failed ..");
        close(sockfd);
    }
    else
        ESP_LOGI(TAG, "socket Creat success ..");
    ret = connect(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr_in)); //连接服务器
    if (ret != 0)
    {
        ESP_LOGI(TAG, "TCP server connect failed ..");
        close(sockfd);
        return;
    }
    MDF_ERROR_ASSERT(mwifi_post_root_status(true)); //向所有子节点发送标志, 子节点根据这个标志判断我是不是连接上服务器
    while (1)
    {
        while (mwifi_is_connected()) //查询本根节点是否连接上路由器
        {
            memset(rx_buffer, 0, 128); //清除上次服务器数据, 不然因为数组太大, 字符后面跟着乱码
            ret = recv(sockfd, rx_buffer, sizeof(rx_buffer) - 1, 0); //接收这一次服务器数据
            if (ret < 0)
                ESP_LOGI(TAG, "root recv server failed ..");
            else
                ESP_LOGI(TAG, "root recv server success = %s", rx_buffer);
            ret = mwifi_write(dest_addr, &data_type, rx_buffer, strlen(rx_buffer), true); //发送数据到指定节点
            if (ret != MDF_OK)
                ESP_LOGI(TAG, "mwifi_write failed ..");
            memset(data, 0, MWIFI_PAYLOAD_LEN);
            ret = mwifi_root_read(src_addr, &data_type, data, &size, portMAX_DELAY); //接收子节点数据
            if (ret < 0)
                ESP_LOGI(TAG, "mwifi_root_read failed ..");
            else
                ESP_LOGI(TAG, "mwifi_root_read success ..");
            ret = write(sockfd, data, size); //将子节点数据发送给服务器
            if (ret < 0)
                ESP_LOGI(TAG, "write server failed ..");
            else
                ESP_LOGI(TAG, "write server success ..");
        }
        sleep(1);
    }
} else //如果该设备是子节点
{
    ESP_LOGI(TAG, "chiled chiled chiled");
    while (1)
```

如果根节点和服务器连接上了，就向所有子节点发送 true

```

{
    while (mwifi_is_connected())
    {
        bool serverFLAG = mwifi_get_root_status();
        while(serverFLAG == false);
        ESP_LOGI(TAG, "node get root connect server success");

        ret = mwifi_read(src_addr, &data_type, data, &size, portMAX_DELAY); //接收子节点数据
        if (ret < 0)
            ESP_LOGI(TAG, "mwifi_child_node_read failed ..");
        else
        {
            ESP_LOGI(TAG, "mwifi_child_node success = %s", data);
        }
        /* 直接取消根节点的 MAC 地址，因为用的 NULL */
        ret = mwifi_write(NULL, &data_type, data, strlen(data), true); //发送数据到指定节点
        if (ret < 0) {
            ESP_LOGI(TAG, "mwifi_write failed ..");
        }
        else
            ESP_LOGI(TAG, "mwifi_child_node mwifi_write success ..");
        sleep(1);
    }
}

```

子节点就用 mwifi_get_root_status 函数去
获取 mwifi_post_root_status 发送的标志位
来判断是不是根节点连接上服务器了，如果
连接上，跳出死循环

```

(13480) xxxzzzzz: root root root root
(13481) xxxzzzzz: socket Creat success ..
(13620) mesh: <MESH_NWK_ROOT_TO_DS_STATE>toDS:1
(13621) [mwifi, 131]: esp_mesh_event_cb event.id: 11
(13621) [mwifi, 234]: State represents: 1
(30571) wifi:new:<11,2>, old:<11,2>, ap:<11,0>, prof:11
(30572) wifi:station: fc:f5:c4:3c:4e:78 join, AID=1, bgn, 40D
(30590) [mwifi, 131]: esp_mesh_event_cb event.id: 5
(30591) [mwifi, 222]: Routing table is changed by adding newly joined children add_num: 1, total_num: 2
(30595) [xxxzzzzz, 36]: Subnode add
(30600) [xxxzzzzz, 38]: total_num: 2
(30652) mesh: <nvs>write assoc:1
(30653) [mwifi, 131]: esp_mesh_event_cb event.id: 3

```

根节点已经连接上服务器

```

I (7404) mesh: 1891<arm>parent monitor, my layer:2(cap:16)(node), interval: 1000ms
D (7413) [mwifi, 131]: esp_mesh_event_cb event.id: 7
I (7418) [mwifi, 137]: Parent is connected
I (7423) [xxxzzzzz, 18]: Parent is connected success
D (7424) [mwifi, 131]: esp_mesh_event_cb event.id: 11
I (7429) [xxxzzzzz, 28]: device sub node
I (7435) [mwifi, 234]: State represents: 1
D (7443) [mwifi, 131]: esp_mesh_event_cb event.id: 14
I (7564) wifi:AP's beacon interval = 307200 us, DTIM period = 2
I (9514) xxxzzzzz: break while .. RootFlag = 0
I (9514) xxxzzzzz: chiled chiled chiled
I (9514) xxxzzzzz: node get root connect server success

```

子节点得到根节点连接服务器标志，跳出死循环。

如果根节点没有连上服务器呢？

```

I (10819) event: sta ip: 192.168.1.141, mask: 255.255.255.0, gw: 192.168.1.1
I (10821) [xxxzzzzz, 41]: Root obtains the IP address. It is posted by LwIP stack automatically
I (13484) xxxzzzzz: break while .. RootFlag = 1
I (13484) xxxzzzzz: root root root root
I (13485) xxxzzzzz: socket Create success
I (13579) xxxzzzzz: TCP server connect failed ..

```

根节点未连接上服务器

```

I (7433) [mwifi, 137]: Parent is connected
I (7438) [xxxzzzzz, 18]: Parent is connected success
D (7439) [mwifi, 131]: esp_mesh_event_cb event.id: 11
I (7444) [xxxzzzzz, 28]: device sub node
I (7449) [mwifi, 234]: State represents: 0
D (7458) [mwifi, 131]: esp_mesh_event_cb event.id: 14
I (7535) wifi:AP's beacon interval = 307200 us, DTIM period = 2
I (9532) xxxzzzzz: break while .. RootFlag = 0
I (9532) xxxzzzzz: chiled chiled chiled

```

子节点无法跳过死循环，无法执行 node get root connect server success 打印。