

HarmonyOS 设备开发入门

版本： 1.2

作者： 连志安

时间： 2020 年 11 月

更新： 鸿蒙小车及其语音控制等 6 个子章节

目录

目录	1
第 1 章 HarmonyOS 介绍.....	2
1.1 鸿蒙系统与 Linux、Android 的不同.....	2
1.2 LiteOS 内核.....	2
1.3 相关资料.....	3
第 2 章 开发环境搭建.....	3
2.1 Linux 环境搭建.....	3
2.2 Windows 访问 ubuntu 文件.....	5
2.3 Windows 环境搭建.....	8
2.4 烧录.....	8
第 3 章 Hi3861 开发.....	8
3.1 编写一个简单的 hello world 程序.....	8
3.2 Hi3861 相关代码结构.....	10
3.3 Hi3861 启动流程.....	11
3.4 Hi3861 AT 指令源码分析，如何添加一条自己的 AT 指令	14
3.5 Hi3861 WiFi 操作，热点连接.....	17
3.6 Hi3861 OLED 驱动.....	21
3.7 Hi3861 实现 APP 配网功能.....	24
3.8 如何往鸿蒙系统源码中添加第三方软件包.....	31
3.9 移植 paho mqtt 软件包到鸿蒙系统.....	36
3.10 ADC 按键的使用.....	51
3.11 使用鸿蒙开发板实现第一个物联网项目	56
3.12 分析 helloworld 程序是如何被调用，SYS_RUN 做什么事情	60
3.13 基于鸿蒙系统 + Hi3861 的 WiFi 小车.....	66
3.13 Hi3861 NV 操作——如何保存数据到开发板，断电不丢失	72
3.13 MQTT 编程.....	78
3.13 语音控制小车.....	79

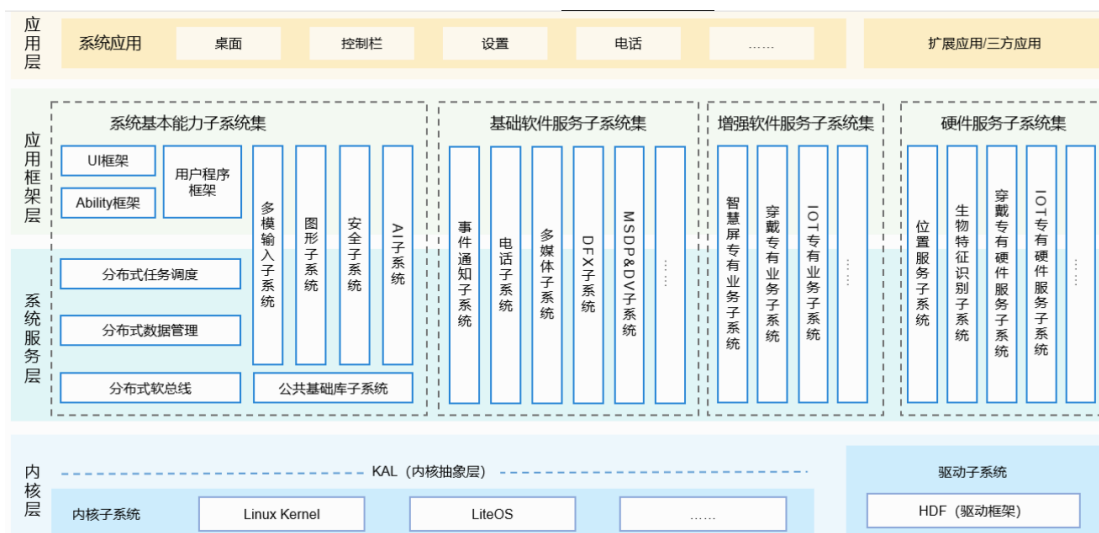
第 1 章 HarmonyOS 介绍

1.1 鸿蒙系统与 Linux、Android 的不同

HarmonyOS 是一款“面向未来”、面向全场景（移动办公、运动健康、社交通信、媒体娱乐等）的分布式操作系统。在传统的单设备系统能力的基础上，HarmonyOS 提出了基于同一套系统能力、适配多种终端形态的分布式理念，能够支持多种终端设备。

HarmonyOS 整体遵从分层设计，从下向上依次为：内核层、系统服务层、框架层和应用层。系统功能按照“系统 > 子系统 > 功能/模块”逐级展开，在多设备部署场景下，支持根据实际需求裁剪某些非必要的子系统或功能/模块。

HarmonyOS 技术架构如图所示。



我们可以看到，鸿蒙系统不单单是一个内核，它还包含了整个操作系统的所有框架，更像是 Windows 和 Android。

而鸿蒙系统的内核支持 Linux 和 LiteOS。

1.2 LiteOS 内核

LiteOS 是一个内核，相比其 Linux 来说，它更精简，启动时间更快。同时 liteOS 内核有 liteOS-a 和 liteOS-m 。

liteOS-a 通常运行支持 MMU 的芯片上，支持内核/APP 空间隔离。ARM cotex -A 系列

liteOS-m 运行在没有 MMU 的芯片上,也就是 MCU,例如我们常见的 STM32 芯片。所以鸿蒙 OS 也是支持 STM32 系列单片机的,但是目前还没有完成移植工作。

1.3 相关资料

鸿蒙官方文档: <https://www.harmonyos.com/cn/develop>

鸿蒙 gitee: <https://openharmony.gitee.com/openharmony>

鸿蒙 OS 代码下载:

https://device.harmonyos.com/cn/docs/start/get-code/oem_sourcecode_guide-0000001050769927

第 2 章 开发环境搭建

关于开发环境的搭建,可以参考华为官网说明。

https://device.harmonyos.com/cn/docs/start/introduce/oem_quickstart_3861_build-0000001054781998。

目前鸿蒙系统的开发方式是在 Linux 系统上面编译源码,Windows 系统上编写、烧录。

故而需要搭建两个开发环境。

2.1 Linux 环境搭建

关于 Linux 系统的环境搭建,个人建议使用 ubuntu 20.04。当然我们也提供了搭建好环境的 ubuntu 20.04 镜像,大家可以直接下载,直接编译代码,不需要再按官网的操作再重新搭建环境。

目测个人第一次搭建至少需要几个小时的时间,还可能会出错。

由于百度网盘经常封链接,如果发现链接失效,可以联系我, VX 13510979604

腾讯云盘

链接: <https://share.weiyun.com/6suCAhNN>


```

[1152/1164] SOLINK ./libaudio_api.so
[1153/1164] SOLINK ./libui.so
[1154/1164] LLVM LINK ./bin/abilityMain
[1155/1164] STAMP obj/foundation/aafwk/frameworks/ability_lite/aafwk_abilityMain_lite.stamp
[1156/1164] LLVM LINK dev_tools/bin/aa
[1157/1164] STAMP obj/foundation/graphic/lite/frameworks/ui/liteui.stamp
[1158/1164] STAMP obj/foundation/aafwk/services/abilitymgr_lite/tools/tools_lite.stamp
[1159/1164] STAMP obj/foundation/aafwk/services/abilitymgr_lite/aafwk_services_lite.stamp
[1160/1164] SOLINK ./libace_lite.so
[1161/1164] STAMP obj/foundation/ace/frameworks/lite/jsfwk.stamp
[1162/1164] STAMP obj/build/lite/ohos.stamp
[1163/1164] ACTION //build/lite:gen_rootfs(//build/lite/toolchain:linux_x86_64_clang)
[1164/1164] STAMP obj/build/lite/gen_rootfs.stamp
ohos ipcamera_hi3516dv300 build success!
harmony@harmony-virtual-machine:~/harmony/code/code-hi3516$
harmony@harmony-virtual-machine:~/harmony/code/code-hi3516$

```

2.2 Windows 访问 ubuntu 文件

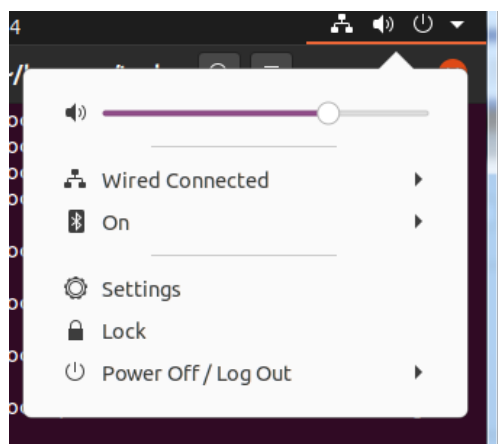
由于我们后面需要在 Windows 上直接编辑 ubuntu 系统里面的鸿蒙源码，故而我们需要使用 samba 服务，让 Windows 能访问到 ubuntu。

操作如下：

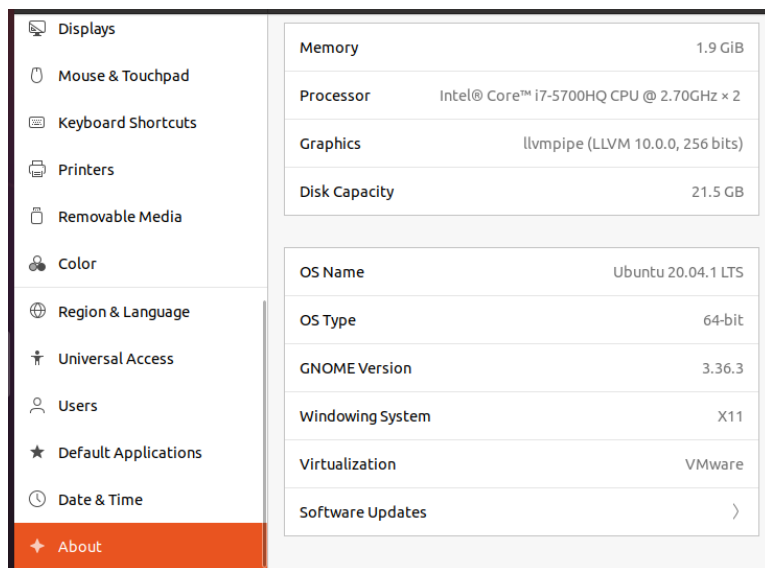
1.设置 apt-get 源

可以更快地下载 samba。设置如下

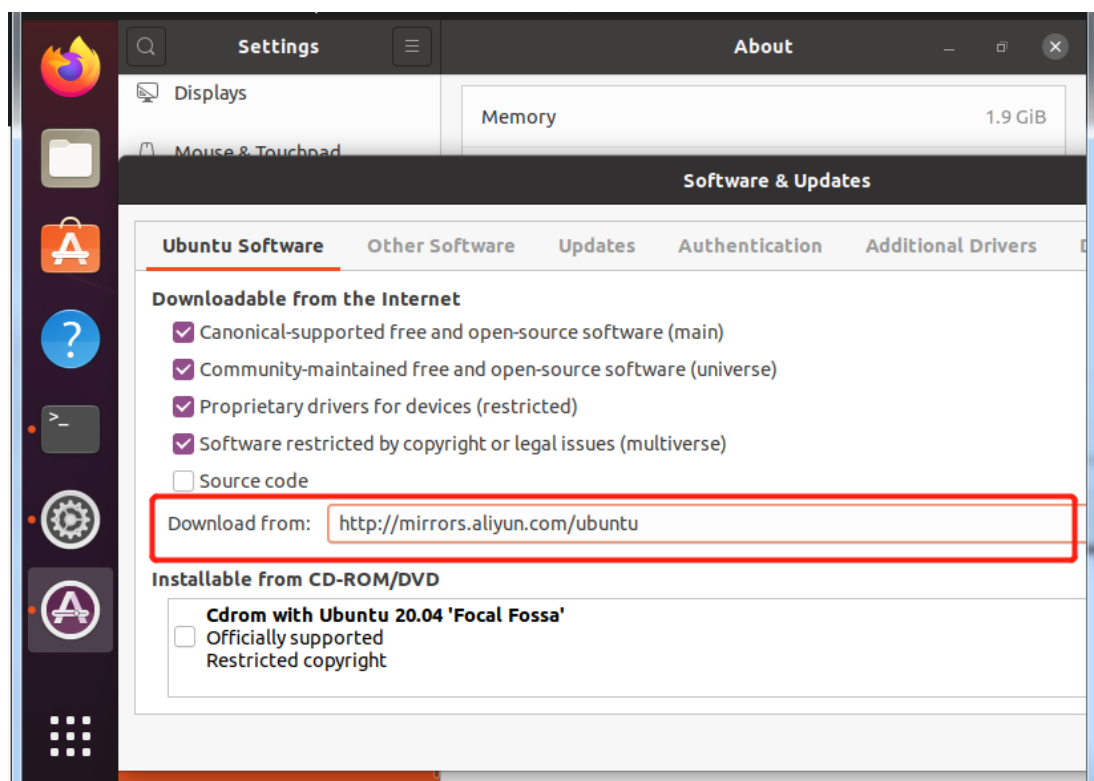
(1) 在桌面右上角点击打开菜单，点击 setting 选项。



(2) 在设置选项右侧下拉找到“关于”，点击 Software Updates。



(3) 在软件和更新界面里可以看到“下载自”，我们可以进行修改。



(4) 推荐选择 [mirros.aliyun.com](http://mirrors.aliyun.com) 或者 mirrors.tuna.tsinghua.edu.cn，你也可以点击选择最佳服务器，测

(5) 试连接最快的软件源（测试时间较长）。

(6) 最后，退出软件与更新界面时，会提示更新软件列表信息，点击重新载入即可。

2. 安装 samba

输入如下命令：

```
sudo apt-get install samba
```

```
sudo apt-get install samba-common
```

修改 samba 配置文件

```
sudo vim /etc/samba/smb.conf
```

在最后加入如下内容：

```
[work]
```

```
comment = samba home directory  
path = /home/harmony/  
public = yes  
browseable = yes  
public = yes  
writeable = yes  
read only = no  
valid users = harmony  
create mask = 0777  
directory mask = 0777  
#force user = nobody  
#force group = nogroup  
available = yes
```

保存退出后，输入如下命令，设置 samba 密码，建议 123456 即可

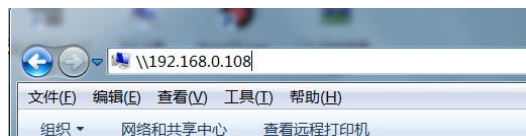
```
sudo smbpasswd -a harmony
```

重启 samba 服务

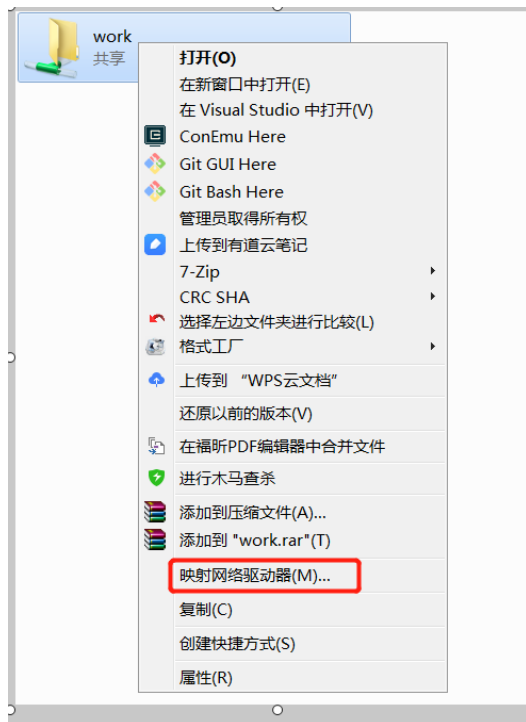
```
sudo service smbd restart
```

3.windows 映射

在文件夹路径输入虚拟机的 IP 地址



最后映射成网络驱动器即可



2.3 Windows 环境搭建

Windows 的环境搭建，官网已经有了，这里就不赘述。

https://device.harmonyos.com/cn/docs/ide/user-guides/tool_install-000000105164976

2.4 烧录

烧录也可以参考官方文档：

https://device.harmonyos.com/cn/docs/ide/user-guides/riscv_upload-0000001051668683

第3章 Hi3861 开发

3.1 编写一个简单的 hello world 程序

编写一个 hello world 程序比较简单，可以参考官网：

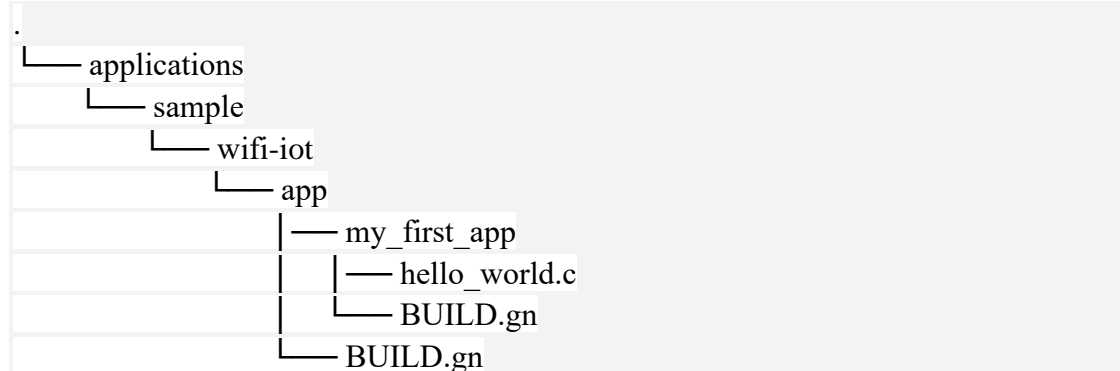
0168544

本文在这里做下总结：

（1）确定目录结构。

开发者编写业务时，务必先在 `./applications/sample/wifi-iot/app` 路径下新建一个目录（或一套目录结构），用于存放业务源码文件。

例如：在 `app` 下新增业务 `my_first_app`，其中 `hello_world.c` 为业务代码，`BUILD.gn` 为编译脚本，具体规划目录结构如下：



（2）编写业务代码。

在 `hello_world.c` 中新建业务入口函数 `HelloWorld`，并实现业务逻辑。并在代码最下方，使用 HarmonyOS 启动恢复模块接口 `SYS_RUN()` 启动业务。（`SYS_RUN` 定义在 `ohos_init.h` 文件中）

```
#include "ohos_init.h"
#include "ohos_types.h"

void HelloWorld(void)
{
    printf("[DEMO] Hello world.\n");
}

SYS_RUN(HelloWorld);
```

（3）编写用于将业务构建成静态库的 `BUILD.gn` 文件。

如步骤 1 所述，`BUILD.gn` 文件由三部分内容（目标、源文件、头文件路径）构成，需由开发者完成填写。以 `my_first_app` 为例，需要创建 `./applications/sample/wifi-iot/app/my_first_app/BUILD.gn`，并完如下配置。

```
static_library("myapp") {
    sources = [
        "hello_world.c"
    ]
}
```

```

    ]
    include_dirs = [
        "../utils/native/liteos/include"
    ]
}

```

`static_library` 中指定业务模块的编译结果，为静态库文件 `libmyapp.a`，开发者根据实际情况完成填写。

`sources` 中指定静态库.a 所依赖的.c 文件及其路径，若路径中包含 `"/"` 则表示绝对路径（此处为代码根路径），若不包含 `"/"` 则表示相对路径。

`include_dirs` 中指定 `source` 所需要依赖的.h 文件路径。

（4）编写模块 `BUILD.gn` 文件，指定需参与构建的特性模块。

配置 `./applications/sample/wifi-iot/app/BUILD.gn` 文件，在 `features` 字段中增加索引，使目标模块参与编译。`features` 字段指定业务模块的路径和目标，以 `my_first_app` 举例，`features` 字段配置如下。

```

import("../build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "my_first_app:myapp",
    ]
}

```

`my_first_app` 是相对路径，指向 `./applications/sample/wifi-iot/app/my_first_app/BUILD.gn`。

`myapp` 是目标，指向 `./applications/sample/wifi-iot/app/my_first_app/BUILD.gn` 中的 `static_library("myapp")`。

3.2 Hi3861 相关代码结构

目前 hi3861 用的是 `liteos-m` 内核，但是目前 hi3681 的 `liteos-m` 被芯片 rom 化了，固化在芯片内部了。所以在 `harmonyOS` 代码是找不到 hi3861 的内核部分。

但是这样不妨碍我们去理清 hi3861 的其他代码结构。

hi3861 平台配置文件

`build\lite\platform\hi3861v100_liteos_riscv\platform.json`

该文件描述了 hi3681 平台相关的代码路径，例如 `application`、`startup` 等。

```
{
  "subsystems": [
    {
      "subsystem": "applications",
      "optional": "true",
      "components": [
        {
          "component": "wifi_iot_sample_app",
          "optional": "true",
          "targets": [
            "//applications/sample/wifi-iot/app"
          ],
          "features": [],
          "deps": {}
        }
      ]
    },
    {
      "subsystem": "startup",
      "optional": "true",
      "components": [
        {
          "component": "syspara",
          "optional": "false",
          "targets": [
            "//base/startup/frameworks/syspara_lite/parameter:parameter"
          ]
        }
      ]
    }
  ]
}
```

这里我列举出来几个比较重要的：

子系统：applications：

路径：applications/sample/wifi-iot/app

作用：这个路径下存放了 hi3681 编写的应用程序代码，例如我们刚刚写得 hello world 代码就放在这个路径下。

子系统：iot_hardware：

路径：base/iot_hardware/frameworks/wifiot_lite

作用：存放了 hi3681 芯片相关的驱动、例如 spi、gpio、uart 等。

子系统：vendor

路径：vendor/hisi/hi3861/hi3861

作用：存放了 hi3681 相关的厂商 SDK 之类的文件。其中最重要的是

vendor\hisi\hi3861\hi3861\app\wifiot_app\init\app_io_init.c

vendor\hisi\hi3861\hi3861\app\wifiot_app\src\app_main.c

其中，app_io_init.c 是 hi3681 内核启动后的 io 口相关设置，用户需根据应用场景，合理选择各外设的 IO 复用配置。

app_main.c 是内核启动进入的应用程序入口。

3.3 Hi3861 启动流程

由于 hi3681 的 liteos-m 被芯片 rom 化了，固化在芯片内部了。所以我们主要看内核启动后的第一个入口函数。

代码路径：

vendor\hisi\hi3861\hi3861\app\wifiot_app\src\app_main.c

```
hi_void app_main(hi_void)
{
#ifdef CONFIG_FACTORY_TEST_MODE
    printf("factory test mode!\r\n");
#endif

    const hi_char* sdk_ver = hi_get_sdk_version();
    printf("sdk ver:%s\r\n", sdk_ver);
    hi_flash_partition_table *ptable = HI_NULL;

    peripheral_init();

    .....中间省略代码

    HOS_SystemInit();
}
```

app_main 一开始打印了 SDK 版本号，最后一行会调用 HOS_SystemInit(); 函数进行鸿蒙系统的初始化。我们进去看下初始化做了哪些动作。

路径：base/startup/services/bootstrap_lite/source/system_init.c

```
void HOS_SystemInit(void)
{
    MODULE_INIT(bsp);
    MODULE_INIT(device);
    MODULE_INIT(core);
    SYS_INIT(service);
    SYS_INIT(feature);
}
```

```

MODULE_INIT(run);
SAMGR_Bootstrap();
}

```

我们可以看到主要是初始化了一些相关模块、系统，包括有 bsp、device（设备）。其中最终的是 MODULE_INIT(run);

它负责调用了，所有 run 段的代码，那么 run 段的代码是哪些呢？

事实上就是我们前面 application 中使用 SYS_RUN() 宏设置的函数名。

还记得我们前面写的 hello world 应用程序吗？

```

#include "ohos_init.h"
#include "ohos_types.h"

void HelloWorld(void)
{
    printf("[DEMO] Hello world.\n");
}
SYS_RUN(HelloWorld);

```

也就是说所有用 SYS_RUN() 宏设置的函数都会在使用 MODULE_INIT(run); 的时候被调用。

为了验证这一点，我们可以加一些打印信息，如下：

```

00021: void HOS_SystemInit(void)
00022: {
00023:     printf("____>>>> %s %d \r\n", __FILE__, __LINE__);
00024:     MODULE_INIT(bsp);
00025:
00026:     printf("____>>>> %s %d \r\n", __FILE__, __LINE__);
00027:     MODULE_INIT(device);
00028:     printf("____>>>> %s %d \r\n", __FILE__, __LINE__);
00029:     MODULE_INIT(core);
00030:     printf("____>>>> %s %d \r\n", __FILE__, __LINE__);
00031:     SYS_INIT(service);
00032:     printf("____>>>> %s %d \r\n", __FILE__, __LINE__);
00033:     SYS_INIT(feature);
00034:
00035:     printf("____>>>> %s %d \r\n", __FILE__, __LINE__);
00036:     MODULE_INIT(run);
00037:     printf("____>>>> %s %d \r\n", __FILE__, __LINE__);
00038:     SAMGR_Bootstrap();
00039:
00040:     printf("____>>>> %s %d \r\n", __FILE__, __LINE__);
00041: } ? end HOS_SystemInit ?
00042:

```

我们重新编译后烧录。打开串口查看打印信息，如下：

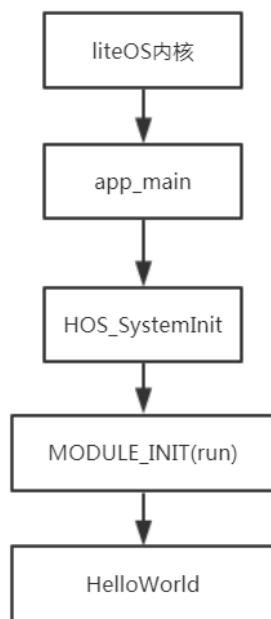
```

>>>>> app/wifiot_app/src/app_main.c 505
>>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 23
>>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 26
>>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 28
>>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 30
>>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 32
>>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 35
[1za][DEMO] Hello world.
>>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 37
>>>>> ../../base/startup/services/bootstrap_lite/source/system_init.c 40
>>>>> app/wifiot_app/src/app_main.c 508

```

可以看到在 35 行之后，就打印 `hello world` 的信息。符合预期。

我们可以大致列出整个流程：



3.4 Hi3861 AT 指令源码分析，如何添加一条自己的 AT 指令

这节主要讲下 hi3861 的 AT 指令相关。先看下 AT 指令在源码中的位置。
上一节已经说到，hi3861 内核启动后的第一个入口函数。

代码路径：

`vendor\hisi\hi3861\hi3861\app\wifiot_app\src\app_main.c`

`hi_void app_main(hi_void)`

在 `app_main` 函数中，会调用 `hi_at_init` 进行 AT 指令的相关初始化。如果初始化成功，则开始注册各类 AT 指令，代码如下：

```

#if defined(CONFIG_AT_COMMAND) || defined(CONFIG_FACTORY_TEST_MODE)
    ret = hi_at_init();
    if (ret == HI_ERR_SUCCESS) {
        hi_at_sys_cmd_register();
    }
#endif

```

初始化部分暂时先不看，主要是底层相关的。我们重点看下 hi_at_sys_cmd_register 注册 AT 指令的函数。

```

hi_void hi_at_sys_cmd_register(hi_void)
{
    printf("____>>>> %s %d \r\n", __FILE__, __LINE__);

    hi_at_general_cmd_register();
#ifndef CONFIG_FACTORY_TEST_MODE
        hi_at_sta_cmd_register();
        hi_at_softap_cmd_register();
#endif
        hi_at_hipriv_cmd_register();
#ifndef CONFIG_FACTORY_TEST_MODE
#ifdef LOSCFG_APP_MESH
        hi_at_mesh_cmd_register();
#endif
        hi_at_lowpower_cmd_register();
#endif
        hi_at_general_factory_test_cmd_register();
        hi_at_sta_factory_test_cmd_register();
        hi_at_hipriv_factory_test_cmd_register();
        hi_at_io_cmd_register();
}

```

其中，hi_at_general_cmd_register 是注册通用指令。代码如下：

```

void hi_at_general_cmd_register(void)
{
    hi_at_register_cmd(g_at_general_func_tbl, AT_GENERAL_FUNC_NUM);
}

```

其实就是把 g_at_general_func_tbl 数组的 AT 指令都注册进来。我们可以看到这个数组的内容：

```

const at_cmd_func g_at_general_func_tbl[] = {
    {"", 0, HI_NULL, HI_NULL, HI_NULL, (at_call_back_func)at_exe_at_cmd},
    {"+RST", 4, HI_NULL, HI_NULL, (at_call_back_func)at_setup_reset_cmd, (at_call_back_func)at_exe_reset_cmd},
    {"+MAC", 4, HI_NULL, (at_call_back_func)cmd_get_macaddr, (at_call_back_func)cmd_set_macaddr, HI_NULL},
    {"+HELP", 5, HI_NULL, HI_NULL, HI_NULL, (at_call_back_func)at_exe_help_cmd},

#ifdef CONFIG_FACTORY_TEST_MODE
    {"+SYSINFO", 8, HI_NULL, HI_NULL, HI_NULL, (at_call_back_func)at_query_sysinfo_cmd},

```

g_at_general_func_tbl 的结构体原型如下：

```

typedef struct {
    //AT 指令命。前面省略 AT

    hi_char *at_cmd_name;

    //指令的长度

    hi_s8    at_cmd_len;

    //at 测试时调用的回调函数

    at_call_back_func at_test_cmd;

    //at 查询时调用的回调函数

    at_call_back_func at_query_cmd;

    //at 设置时调用的回调函数

    at_call_back_func at_setup_cmd;

    //at 运行时调用的回调函数

    at_call_back_func at_exe_cmd;

} at_cmd_func;

```

看到这个数组，聪明的朋友应该知道怎么增加第一条属于自己的指令了吧~~~~

（1）增加 AT 指令

```

const at_cmd_func g_at_general_func_tbl[] = {
    {"", 0, HI_NULL, HI_NULL, HI_NULL, (at_call_back_func)at_exe_at_cmd},
    {"+RST", 4, HI_NULL, HI_NULL, (at_call_back_func)at_setup_reset_cmd, (at_call_back_func)at_exe_reset_cmd},
    {"+MYTEST", 7, at_test_mytest_cmd, at_query_mytest_cmd, at_setup_mytest_cmd, at_exe_mytest_cmd},
    {"+MAC", 4, HI_NULL, (at_call_back_func)cmd_get_macaddr, (at_call_back_func)cmd_set_macaddr, HI_NULL},
    {"+HELP", 5, HI_NULL, HI_NULL, HI_NULL, (at_call_back_func)at_exe_help_cmd},

```

（2）完善相关函数：

```
hi_u32 at_setup_mytest_cmd(hi_s32 argc, const hi_char *argv[])
```

```

{
    hi_at_printf("at_setup_mytest_cmd \r\n");

    return HI_ERR_SUCCESS;
}

```

```
hi_void at_exe_mytest_cmd(hi_s32 argc, const hi_char *argv[])
```

```

{

```



```

        hi_at_printf("at_exe_mytest_cmd \r\n");
        return HI_ERR_SUCCESS;
    }

hi_u32 at_query_mytest_cmd(hi_s32 argc, const hi_char* argv[])
{
    hi_at_printf("at_query_mytest_cmd \r\n");
    return HI_ERR_SUCCESS;
}

hi_u32 at_test_mytest_cmd(hi_s32 argc, const hi_char* argv[])
{
    hi_at_printf("at_test_mytest_cmd \r\n");
    return HI_ERR_SUCCESS;
}

```

编译后我们开始测试：

发送：AT+MYTEST

接收：at_exe_mytest_cmd

ERROR

发送：AT+MYTEST?

接收：at_query_mytest_cmd

发送：AT+MYTEST=1

接收：at_setup_mytest_cmd

3.5 Hi3861 WiFi 操作，热点连接

之前我们使用 Hi3861 的时候，是使用 AT 指令连接到 WiFi 热点的。例如：

- | | |
|---------------------------------|--------------------------------------|
| 1. AT+STARTSTA | - 启动STA模式 |
| 2. AT+SCAN | - 扫描周边AP |
| 3. AT+SCANRESULT | - 显示扫描结果 |
| 4. AT+CONN="SSID",,2,"PASSWORD" | - 连接指定AP，其中SSID/PASSWORD为待连接的热点名称和密码 |
| 5. AT+STASTAT | - 查看连接结果 |
| 6. AT+DHCP=wlan0,1 | - 通过DHCP向AP请求wlan0的IP地址 |

查看WLAN模组与网关联通是否正常，如下图所示。

- | | |
|--------------------|------------------------------------|
| 1. AT+IFCFG | - 查看模组接口IP |
| 2. AT+PING=X.X.X.X | - 检查模组与网关的联通性，其中X.X.X.X需替换为实际的网关地址 |

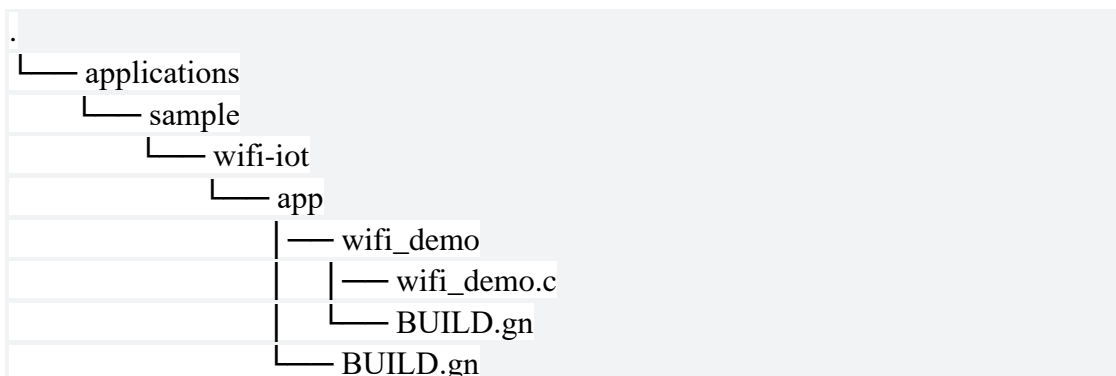
但是很多时候，我们需要实现开机后自动连接到某个热点，光靠 AT 指令不行。

Hi3861 为我们提供了 WiFi 操作的相关 API，方便我们编写代码，实现热点连接。

1.代码实现

先直接上代码和操作演示。

跟我们最早的 hello world 代码一样，在 app 下新增业务 wifi_demo，其中 hello_world.c 为业务代码，BUILD.gn 为编译脚本，具体规划目录结构如下：



Wifi_demo.c 代码如下：

见附件 doc\05 WiFi 操作\sta_demo\sta_demo.c

Wifi_demo 目录下的 BUILD.gn 文件内容如下：

```

static_library("wifi_demo") {
    sources = [
        "wifi_demo.c"
    ]

    include_dirs = [
        "../utils/native/lite/include",
    ]
}

```

```

        "/kernel/liteos_m/components/cmsis/2.0",
        "/base/iot_hardware/interfaces/kits/wifiot_lite",
        "/vendor/hisi/hi3861/hi3861/third_party/lwip_sack/include",
        "/foundation/communication/interfaces/kits/wifi_lite/wifiservice",
    ]
}

```

app 目录下的 BUILD.gn 文件内容修改如下：

```
import("//build/lite/config/component/lite_component.gni")
```

```

lite_component("app") {
    features = [
        "wifi_demo:wifi_demo",
    ]
}

```

编译烧录，查看串口数据：

```

+NOTICE:SCANFINISH
WiFi: Scan results available
SSID: 15919500
SSID: Netcore_FD55A7
../../../../applications/sample/wifi-iot/app/wifi_demo/wifi_demo.c 94
../../../../base/startup/services/bootstrap_lite/source/system_init.c 40
../../../../base/startup/services/bootstrap_lite/source/system_init.c 43
app/wifiot_app/src/app_main.c 502
00 00:00:00 0 132 D 0/HIVIEW: hilog init success.
00 00:00:00 0 132 D 0/HIVIEW: log limit init success.
00 00:00:00 0 132 I 1/SAMGR: Bootstrap core services(count:3).
00 00:00:00 0 132 I 1/SAMGR: Init service:0x4ae97c TaskPool:0xf9e0c
00 00:00:00 0 132 I 1/SAMGR: Init service:0x4ae9ec TaskPool:0xf9e0c
00 00:00:00 0 132 I 1/SAMGR: Init service:0x4aeafc TaskPool:+NOTICE:CONNECTED
0xf9f0c
00 00:00:00 0 164 I 1/SAMGR: Init service 0x4ae9ec <time: 5570ms> success!
00 00:00:00 0 220 I 1/SAMGR: Init service 0x4ae97c <time: 5570ms> success!
00 00:00:00 0 8 D 0/HIVIEW: hiview init success.
00 00:00:00 0 8 I 1/SAMGR: Init service 0x4aeafc <time: 5570ms> success!
00 00:00:00 0 8 I 1/SAMGR: Initialized all core system services!
00 00:00:00 0 220 I 1/SAMGR: Bootstrap system and application services(count:0).
00 00:00:00 0 220 I 1/SAMGR: Initialized all system and application services!
00 00:00:00 0 220 I 1/SAMGR: Bootstrap dynamic registered services(count:0).
WiFi: Connected

```

可以看到有打印扫描到的热点名称：

SSID: 15919500

SSID: Netcore_FD55A7

同时最后打印：WiFi: Connected 成功连接上热点。

2.wifi api 接口说明

Hi3861 提供了非常多的 wifi 相关 API，主要文件是 hi_wifi_api.h

我们这里只列举最重要的几个 API

(1) 开启 STA

```
int hi_wifi_sta_start(char *ifname, int *len);
```

(2) 停止 STA

```
int hi_wifi_sta_stop(void);
```

(3) 扫描附件的热点

```
int hi_wifi_sta_scan(void);
```

(4) 连接热点

```
int hi_wifi_sta_connect(hi_wifi_assoc_request *req);
```

其中 hi_wifi_assoc_request *req 结构的定义如下：

```
typedef struct {
    char ssid[HI_WIFI_MAX_SSID_LEN + 1];    /**< SSID. CNcomment: SSID 只支持ASCII字符.CNend */
    hi_wifi_auth_mode auth;                  /**< Authentication mode. CNcomment: 认证类型.CNend */
    char key[HI_WIFI_MAX_KEY_LEN + 1];       /**< Secret key. CNcomment: 秘钥.CNend */
    unsigned char bssid[HI_WIFI_MAC_LEN];    /**< BSSID. CNcomment: BSSID.CNend */
    hi_wifi_pairwise pairwise;               /**< Encryption type. CNcomment: 加密方式,不需指定时置0.CNend */
} hi_wifi_assoc_request;
```

这里需要注意的是，通常加密方式是：HI_WIFI_SECURITY_WPA2PSK

例如我家的热点的连接方式的代码实现如下：

```

/* copy SSID to assoc_req */
//热点名称
rc = memcpy_s(assoc_req.ssid, HI_WIFI_MAX_SSID_LEN + 1, "15919500", 8); /* 9:ssid length */
if (rc != EOK) {
    printf("%s %d \r\n", __FILE__, __LINE__);
    return -1;
}

/*
 * OPEN mode
 * for WPA2-PSK mode:
 * set assoc_req.auth as HI_WIFI_SECURITY_WPA2PSK,
 * then memcpy(assoc_req.key, "12345678", 8).
 */
//热点加密方式
assoc_req.auth = HI_WIFI_SECURITY_WPA2PSK;

/* 热点密码 */
memcpy(assoc_req.key, "11206582488", 11);

```

3.6 Hi3861 OLED 驱动

Hispark WiFi 开发套件又提供一个 OLED 屏幕，但是鸿蒙源码中没有这个屏幕的驱动，我们需要自己去移植。



经过一晚上的调试，现在终于在鸿蒙系统上实现 OLED 屏幕的显示了，效果如下：



这里记录一下移植的过程

(1) 编写驱动代码

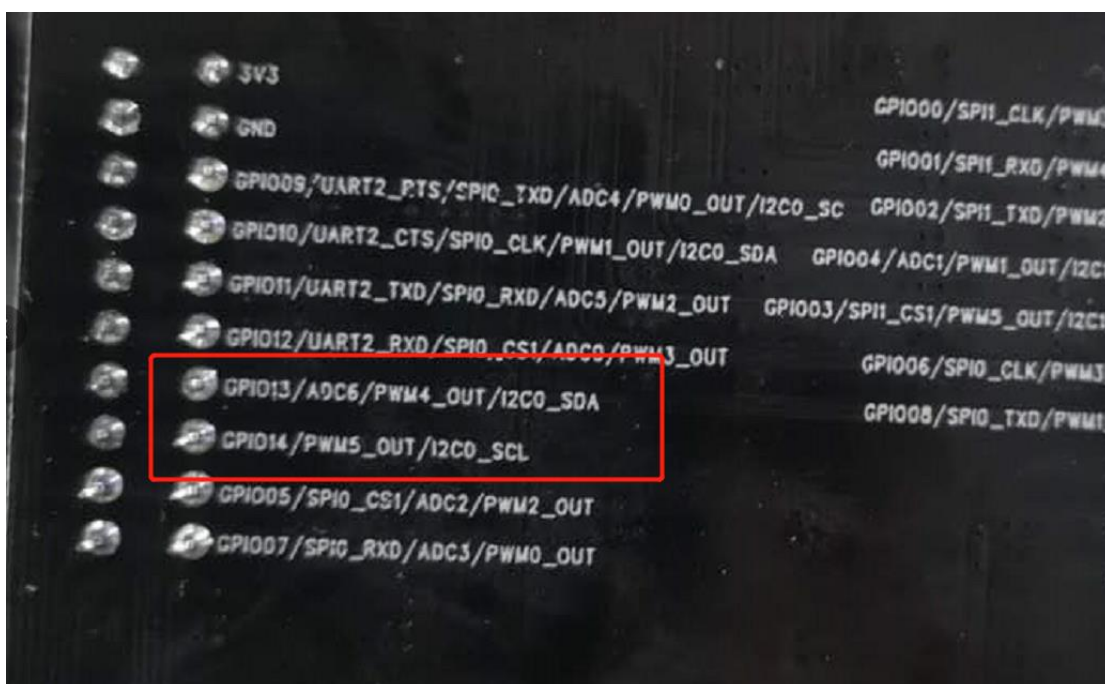
首先在

```
└─ applications
    └─ sample
        └─ wifi-iot
            └─ app
```

新增应用：oled_demo，源码已经放在附件，大家自己下载。

(2) 设置 I2C 引脚复用

确定 i2c 引脚，查看原理图，可以看到 OLED 屏幕使用的是 I2C0，引脚是 GPIO13、GPIO14



所以我们需要修改源码，在

vendor\hisi\hi3861\hi3861\app\wifiot_app\init\app_io_init.c 文件中，初始化 I2C 引脚的代码修改成如下：

```
#ifdef CONFIG_I2C_SUPPORT
/* I2C IO 复用也可以选择 3/4; 9/10, 根据产品设计选择 */
hi_io_set_func(HI_IO_NAME_GPIO_13, HI_IO_FUNC_GPIO_13_I2C0_SDA);
hi_io_set_func(HI_IO_NAME_GPIO_14, HI_IO_FUNC_GPIO_14_I2C0_SCL);
#endif
```

(3) 开启 I2C 功能

修改文件：vendor\hisi\hi3861\hi3861\build\config\usr_config.mk

增加 CONFIG_I2C_SUPPORT=y

以上修改变完成了，重新编译即可看到 OLED 能成功驱动。

(4) OLED 屏幕驱动讲解

入口函数：

```
void my_oled_demo(void)
```

```
{
```

```
    //初始化，我们使用的是 I2C0
```

```
    hi_i2c_init(HI_I2C_IDX_0, 100000); /* baudrate: 100000 */
```

```

led_init();

OLED_ColorTurn(0);//0 正常显示, 1 反色显示
OLED_DisplayTurn(0);//0 正常显示 1 屏幕翻转显示

OLED_ShowString(8,16,"hello world",16);

OLED_Refresh();
}

```

I2C 写函数:

```

hi_u32 my_i2c_write(hi_i2c_idx id, hi_u16 device_addr, hi_u32 send_len)
{
    hi_u32 status;
    hi_i2c_data es8311_i2c_data = { 0 };

    es8311_i2c_data.send_buf = g_send_data;
    es8311_i2c_data.send_len = send_len;
    status = hi_i2c_write(id, device_addr, &es8311_i2c_data);
    if (status != HI_ERR_SUCCESS) {
        printf("==== Error: I2C write status = 0x%x! =====\r\n",
status);

        return status;
    }

    return HI_ERR_SUCCESS;
}

```

3.7 Hi3861 实现 APP 配网功能

本节主要讲如何去实现 Hi3861 配网功能。本节知识有点多，包括 Hi3861 的 WiFi 操作，AP 模式、STA 模式、按键功能、网络编程、JSON 数据格式、手机 APP。

所有源码，还有手机 APP 均提供下载，大家自领。

也可以直接观看视频。

先上原理：

目前主流的 WIFI 配置模式有以下 2 种：

1、智能硬件处于 AP 模式（类似路由器，组成局域网），手机用于 STA 模式

手机连接到处于 AP 模式的智能硬件后组成局域网,手机发送需要连接路由的 SSID 及密码至智能硬件，智能硬件主动去连接指定路由后,完成配网

2、一键配网(smartConfig)模式

智能硬件处于混杂模式下,监听网络中的所有报文;手机 APP 将 SSID 和密码编码到 UDP 报文中,通过广播包或组播报发送,智能硬件接收到 UDP 报文后解码,得到正确的 SSID 和密码,然后主动连接指定 SSID 的路由完成连接。

本文主要讲如何实现第一种 AP 方式。

AP 是 (Wireless) Access Point 的缩写，即 (无线) 访问接入点。简单来讲就像是无线路由器一样，设备打开后进入 AP 模式，在手机的网络列表里面，可以搜索到类似 TPLINK_XXX 的名字（SSID）。

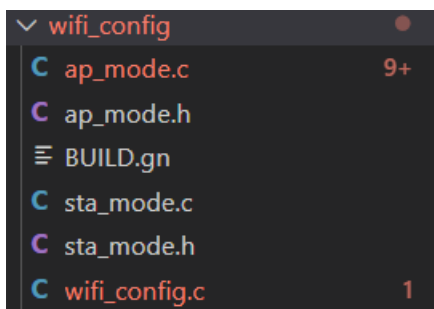
连接步骤：

- 1、Hi3861 上面有一个 user 按键，用户可以按下这个按钮，Hi386 会进入 AP 模式
- 2、手机扫描 WIFI 列表：扫描到 Hi3861 的 SSID（目前是“Hispark-WiFi-IoT”）连接该智能硬件设备，通过手机 APP 发送我们要连接的热点的 ssid 和密码
- 3、智能硬件设备通过 UDP 包获取配置信息，切换网络模式连接 WIFI 后配网完成

代码实现

（1）代码结构

代码主要由 3 个文件组成



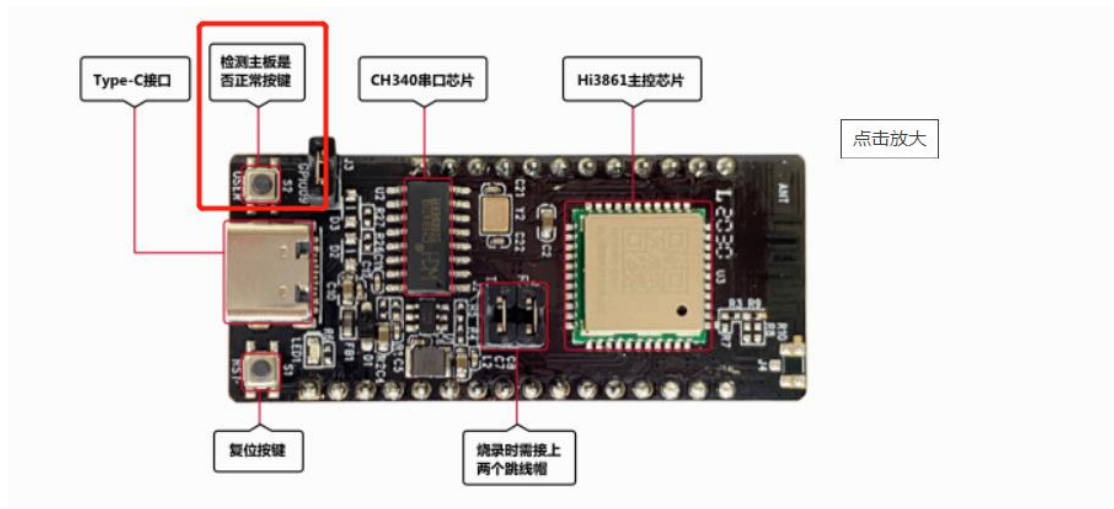
ap_mode.c: 主要实现 AP 模式，并实现一个简单的 UDP 服务器，获取手机 APP 传输过来的热点账号和密码。

sta_mode.c: 实现连接配网的功能。

wifi_config.c: 入口函数，实现按下按键后开始配网的功能。

（2）按键功能实现

通过查阅原理图，我们可以看到 Hi3861 在 type-C 口附近有一个 user 按钮，如图，主要不要和复位按钮搞错了。user 按钮对应的是 GPIO5 引脚。



于是我们可以使用按键中断编程的方式去实现，代码如下：

```
/* 设置 按键中断响应 */
hi_void my_gpio_isr_demo(hi_void)
{
    hi_u32 ret;

    printf("----- gpio isr demo -----r\n");

    (hi_void)hi_gpio_init();

    hi_io_set_func(HI_IO_NAME_GPIO_5, HI_IO_FUNC_GPIO_5_GPIO); /* uart1 rx */

    ret = hi_gpio_set_dir(HI_GPIO_IDX_5, HI_GPIO_DIR_IN);
    if (ret != HI_ERR_SUCCESS) {
        printf("===== ERROR =====gpio -> hi_gpio_set_dir1 ret:%d\r\n", ret);
        return;
    }

    ret = hi_gpio_register_isr_function(HI_GPIO_IDX_5, HI_INT_TYPE_EDGE,
                                       HI_GPIO_EDGE_RISE_LEVEL_HIGH, my_gpio_isr_func, HI_NULL);
    if (ret != HI_ERR_SUCCESS) {
        printf("===== ERROR =====gpio -> hi_gpio_register_isr_function ret:%d\r\n", ret);
    }
}
```

其中需要主要的是需要使用 hi_io_set_func(HI_IO_NAME_GPIO_5, HI_IO_FUNC_GPIO_5_GPIO); 修改 GPIO5 为普通引脚，否则 GPIO5 默认会被初始化为 串口引脚，导致无法使用。

GPIO5 中断回调函数如下：

```

/* gpio callback func */
hi_void my_gpio_isr_func(hi_void *arg)
{
    hi_unref_param(arg);
    printf("----- gpio isr success -----\\r\\n");

    //启动配网功能
    start_wifi_config_flg = 1;
}

```

其实很简单，就是置某个变量为 1 而已。

(3) 接下来进入 AP 模式

代码如下，一旦发现 start_wifi_config_flg 不为 0，也就是说发生了按键被按下的事件，那就会调用 wifi_start_softap 函数进入 AP 模式

```

void *wifi_config_thread(const char *arg)
{
    arg = arg;

    my_gpio_isr_demo();

    while(start_wifi_config_flg == 0)
    {
        usleep(300000);
    }

    printf("wifi_start_softap \\r\\n");
    wifi_start_softap();

    osThreadExit();
    return NULL;
}

```

(4) AP 模式

AP 模式的代码部分也很简单，首先我们要先设置好 Hi3861 AP 模式下的 SSID，然后开放网络，不加密。对应的函数是 wifi_start_softap

```

rc = memcpy_s(hapd_conf.ssid, HI_WIFI_MAX_SSID_LEN + 1, "Hispark-WiFi-IoT", 16); /* 9:ssid length */
if (rc != EOK) {
    return -1;
}

hapd_conf.authmode = HI_WIFI_SECURITY_OPEN;
hapd_conf.channel_num = 1;

ret = hi_wifi_softap_start(&hapd_conf, ifname, &len);
if (ret != HISI_OK) {
    printf("hi_wifi_softap_start\n");
    return -1;
}

```

接下来设置好 Hi3861 的网段、IP 等，并开启 UDP 服务：

```

IP4_ADDR(&st_gw, 192, 168, 10, 1); /* input your IP for example: 192.168.1.1 */
IP4_ADDR(&st_ipaddr, 192, 168, 10, 1); /* input your netmask for example: 192.168.1.1 */
IP4_ADDR(&st_netmask, 255, 255, 255, 0); /* input your gateway for example: 255.255.255.0 */
netifapi_netif_set_addr(g_lwip_netif, &st_ipaddr, &st_netmask, &st_gw);

netifapi_dhcp_start(g_lwip_netif, 0, 0);

udp_thread();

```

(5) UDP 服务器

UDP 服务器绑定的端口号是 50001，使用 socket 通信接口

```

void udp_thread(void)
{
    int ret;
    struct sockaddr_in servaddr;
    cJSON *recvjson;

    int sockfd = socket(PF_INET, SOCK_DGRAM, 0);

    //服务器 ip port
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(50001);

    bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
}

```

绑定完端口号后，进入接收数据

```

memset(recvline, sizeof(recvline), 0);
ret = recvfrom(sockfd, recvline, 1024, 0, (struct sockaddr*)&addrClient, (socklen_t*)&sizeClientAddr);

if(ret>0)
{
    char *pClientIP =inet_ntoa(addrClient.sin_addr);

    printf("%s-%d(%d) says:%s\n",pClientIP,ntohs(addrClient.sin_port),addrClient.sin_port, recvline);

    //进行json解析
    recvjson = cJSON_Parse(recvline);

    printf("ssid : %s\r\n", cJSON_GetObjectItem(recvjson, "ssid")->valuestring);
    printf("passwd : %s\r\n", cJSON_GetObjectItem(recvjson, "passwd")->valuestring);

    memset(ssid, sizeof(ssid), 0);
    memset(passwd, sizeof(passwd), 0);

    strcpy(ssid, cJSON_GetObjectItem(recvjson, "ssid")->valuestring);
    strcpy(passwd, cJSON_GetObjectItem(recvjson, "passwd")->valuestring);

    cJSON_Delete(recvjson);

    //先停止AP模式
    wifi_stop_softap();

    //启动STA模式
    start_sta_connect(ssid, strlen(ssid), passwd, strlen(passwd));
}

```

数据这里我使用 json 格式，由于鸿蒙系统代码中已经自带 cJSON 库，可以直接使用，这一部分的代码也比较简单，大家可以看看。

（6）开启 STA 模式

启动 STA 模式的代码部分也比较简单，我之前有一篇文章有讲，具体代码如下：

```

113 //wifi模块初始化，刚刚我们推出了AP模式。需要重新初始化
114 ret = hi_wifi_init(wifi_vap_res_num, wifi_user_res_num);
115 if (ret != HISI_OK) {
116     printf("%s %d \r\n", __FILE__, __LINE__);
117     //return -1;
118 }
119
120 //启动STA模式
121 ret = hi_wifi_sta_start(iffname, &len);
122 if (ret != HISI_OK) {
123     printf("%s %d \r\n", __FILE__, __LINE__);
124     return;
125 }
126
127 /* 注册wifi事件回调函数，如果成功连接上热点，会有打印信息
128 */
129 ret = hi_wifi_register_event_callback(wifi_wpa_event_cb);
130 if (ret != HISI_OK) {
131     printf("register wifi event callback failed\n");
132 }
133
134 /* acquire netif for IP operation */
135 g_lwip_netif = netifapi_netif_find(iffname);
136 if (g_lwip_netif == NULL) {
137     printf("%s: get netif failed\n", __FUNCTION__);
138     return ;
139 }
140
141 /* 开始进行热点连接 */
142 ret = hi_wifi_start_connect(ssid, ssid_len, passwd, passwd_len);
143 if (ret != 0) {
144     printf("%s %d \r\n", __FILE__, __LINE__);
145     return ;
146 }

```

关键代码已经做了注释。

(7) 连接热点

连接热点时，只需要传入 ssid、加密方式和密码即可。

需要主要的地方是我们通常的 wifi 加密都是 HI_WIFI_SECURITY_WPA2PSK

```

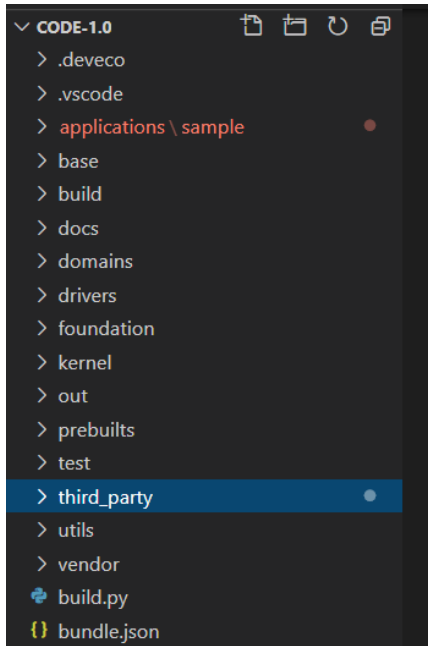
66 int hi_wifi_start_connect(char *ssid, int ssid_len, char *passwd, int passwd_len)
67 {
68     int ret;
69     errno_t rc;
70     hi_wifi_assoc_request assoc_req = {0};
71
72     /* copy SSID to assoc_req */
73     //热点名称
74     rc = memcpy_s(assoc_req.ssid, HI_WIFI_MAX_SSID_LEN + 1, ssid, ssid_len); /* 9:ssid length */
75     if (rc != EOK) {
76         printf("%s %d \r\n", __FILE__, __LINE__);
77         return -1;
78     }
79
80     /*
81      * OPEN mode
82      * for WPA2-PSK mode:
83      * set assoc_req.auth as HI_WIFI_SECURITY_WPA2PSK,
84      * then memcpy(assoc_req.key, "12345678", 8).
85      */
86     //热点加密方式
87     assoc_req.auth = HI_WIFI_SECURITY_WPA2PSK;
88
89     /* 热点密码 */
90     memcpy(assoc_req.key, passwd, passwd_len);
91
92
93     ret = hi_wifi_sta_connect(&assoc_req);
94     if (ret != HISI_OK) {
95         printf("%s %d \r\n", __FILE__, __LINE__);
96         return -1;
97     }
98     printf("%s %d \r\n", __FILE__, __LINE__);
99     return 0;
100 }

```

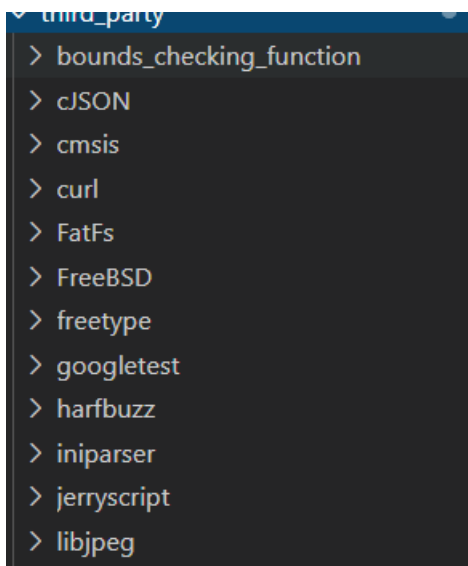
作者简介: 连志安, 旗点云科技创始人, 广东长虹技术研究所(国企)Android 南美 Android 软件负责人。之前在康佳集团(国企)、CVTE(上市公司)等公司任职。负责过 Android TV、智能网关、路由器、智能家居、安防报警器等项目。熟悉单片机、嵌入式 linux、服务器、Android 系统、手机 APP 开发等。

3.8 如何往鸿蒙系统源码中添加第三方软件包

打开鸿蒙系统的源码, 可以看到有这么一个文件夹: `third_party`。里面存放的是第三方的代码。



点开我们可以看到有很多第三方代码：



后续我们如果需要往系统中添加、移植任何开源代码，都可以添加到这个文件夹中。接下来，教大家如何添加一个自己的软件包，名字为 `a_myparty`。

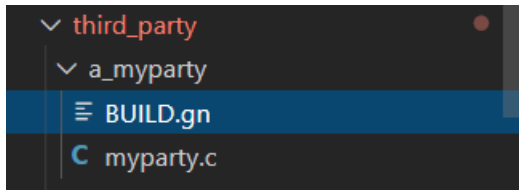
1. 新建一个文件夹 `a_myparty`

2. 往文件中放置软件包源码

这里我放在的是 `myparty.c` 文件

3. 新建 `BUILD.gn` 文件

整个代码目录如下：



4. myparty.c 文件内容如下：

其实，我这个只是为了演示的，所以里面代码没什么作用

```
#include <stdio.h>
```

```
void myparty_test(void)
{
    printf("first myparty \r\n");
}
```

5. BUILD.gn 文件内容如下：

BUILD.gn 文件主要是描述了软件包的相关信息，包括编译哪些源文件，头文件路径、编译方式（目前 Hi3861 只支持静态加载）

```
import("//build/lite/config/component/lite_component.gni")
import("//build/lite/ndk/ndk.gni")
```

#这里是配置头文件路径

```
config("a_myparty_config") {
    include_dirs = [
        ".",
    ]
}
```

#这里是配置要编译哪些源码

```
a_myparty_sources = [
    "myparty.c",
]
```

#这里是静态链接，类似于 Linux 系统的 .a 文件

```

lite_library("a_myparty_static") {
    target_type = "static_library"

    sources = a_myparty_sources

    public_configs = [ ":a_myparty_config" ]
}

#这里是动态加载，类似于 Linux 系统的 .so 文件
lite_library("a_myparty_shared") {
    target_type = "shared_library"

    sources = a_myparty_sources

    public_configs = [ ":a_myparty_config" ]
}

#这里是入口，选择是静态还是动态
ndk_lib("a_myparty_ndk") {
    if (board_name != "hi3861v100") {
        lib_extension = ".so"

        deps = [
            ":a_myparty_shared"
        ]
    } else {
        deps = [
            ":a_myparty_static"
        ]
    }

    head_files = [
        "../third_party/a_myparty"
    ]
}

```

到了这里我们基本上就写完了。

最后我们要让这个第 3 方软件包编译到我们固件中。

6. 打开第 3 方软件包功能，使其参与编译：

打开 vendor\hisi\hi3861\hi3861\BUILD.gn 文件

在下图部分添加 "//third_party/a_myparty:a_myparty_static"

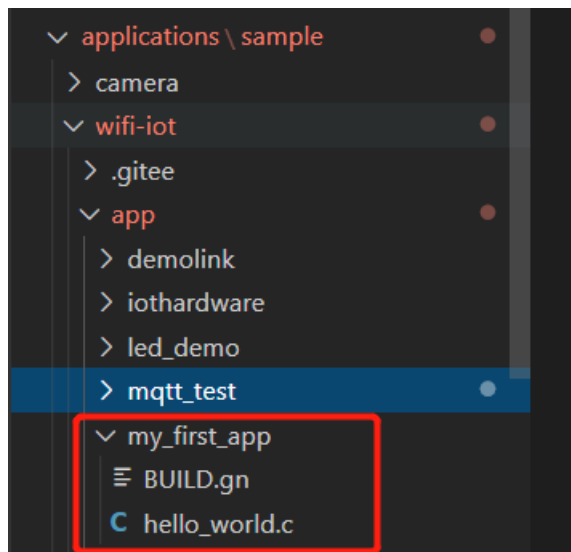
别忘了分号。。。

```
17
18 lite_component("sdk") {
19     features = [ ]
20
21     deps = [
22         "//kernel/liteos_m/components/cmsis",
23         "//kernel/liteos_m/components/kal",
24         "//third_party/cJSON:cjson_static",
25         "//third_party/pahomqtt:pahomqtt_static",
26         "//third_party/a_myparty:a_myparty_static"
27     ]
28 }
```

7. 使用

到了这里我们的第 3 方软件包就添加完成了，接下来我们要在 app 代码中使用它

打开 applications\sample\wifi-iot\app\my_first_app\BUILD.gn 文件，没有的同学请自己先完成 hello world 入门例程先。



添加 "//third_party/a_myparty" 头文件路径，BUILD.gn 文件内容如下：

```
static_library("my_first_app") {
    sources = [
        "hello_world.c"
    ]
    include_dirs = [
        "//utils/native/liteos/include",
        "//third_party/a_myparty"
    ]
}
```

```
    ]  
}
```

打开 hello_world.c 文件，内容如下：

```
#include "ohos_init.h"  
  
#include "ohos_types.h"  
  
#include "stdio.h"  
  
//导入头文件  
  
#include "myparty.h"  
  
void HelloWorld(void)  
{  
    printf("%s %d \r\n", __FILE__, __LINE__);  
    printf("[DEMO] Hello world.\n");  
    //调用第 3 方软件包 的函数 myparty_test()  
    myparty_test();  
}  
  
SYS_RUN(HelloWorld);
```

最后编译测试即可看到打印信息：

```
[DEMO] Hello world.  
first myparty
```

说明添加成功。

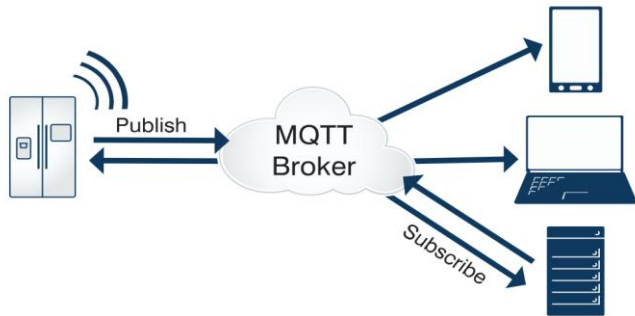
3.9 移植 paho mqtt 软件包到鸿蒙系统

MQTT 是当前最主流的物联网通信协议，需要物联网云平台，例如华为云、阿里云、移动 OneNET 都支持 mqtt。而 Hi3861 则是一款专为 IoT 应用场景打造的芯片。本节主要讲如何在鸿蒙系统中通过移植第 3 方软件包 paho mqtt 去实现 MQTT 协议功能，最后会给出测试验证。为后续的物联网项目打好基础。

3.9.1 MQTT 介绍

MQTT 全称为 Message Queuing Telemetry Transport（消息队列遥测传输）

是一种基于发布/订阅范式的二进制“轻量级”消息协议，由 IB 公司发布。针对于网络受限和嵌入式设备而设计的一种数据传输协议。MQTT 最大优点在于，可以以极少的代码和有限的带宽，为连接远程设备提供实时可靠的消息服务。作为一种低开销、低带宽占用的即时通讯协议，使其在物联网、小型设备、移动应用等方面有较广泛的应用。MQTT 模型如图所示。



3.9.2 移植 paho mqtt 软件包

1. 下载 paho mqtt 软件包，添加到鸿蒙代码中

paho mqtt-c 是基于 C 语言实现的 MQTT 客户端，非常适合用在嵌入式设备上。首先下载源码：<https://github.com/eclipse/paho.mqtt.embedded-c>

下载之后解压，会得到这么一个文件夹：

.settings	2020/10/22 19:42	文件夹	
Debug	2020/10/22 19:42	文件夹	
doc	2020/10/22 19:42	文件夹	
MQTTClient	2020/10/22 19:42	文件夹	
MQTTClient-C	2020/10/22 19:42	文件夹	
MQTTPacket	2020/10/22 19:42	文件夹	
si	2020/10/22 19:42	文件夹	
test	2020/10/22 19:42	文件夹	
.cproject	2018/3/5 23:56	CPROJECT 文件	18 KB
.gitignore	2018/3/5 23:56	文本文档	1 KB
.project	2018/3/5 23:56	PROJECT 文件	1 KB
.travis.yml	2018/3/5 23:56	YML 文件	1 KB
about.html	2018/3/5 23:56	Chrome HTML D...	2 KB
BUILD.gn	2020/10/22 19:44	GN 文件	2 KB
CMakeLists.txt	2018/3/5 23:56	文本文档	2 KB
CONTRIBUTING.md	2018/3/5 23:56	MD 文件	4 KB
edl-v10	2018/3/5 23:56	文件	2 KB
epl-v10	2018/3/5 23:56	文件	11 KB
library.properties	2018/3/5 23:56	PROPERTIES 文件	1 KB
Makefile	2018/3/5 23:56	文件	6 KB
notice.html	2018/3/5 23:56	Chrome HTML D...	10 KB
README.md	2018/3/5 23:56	MD 文件	4 KB
travis-build.sh	2018/3/5 23:56	Shell Script	1 KB
travis-env-vars	2018/3/5 23:56	文件	1 KB
travis-install.sh	2018/3/5 23:56	Shell Script	1 KB

我们在鸿蒙系统源码的 third_party 文件夹下创建一个 pahomqtt 文件夹，然后把解压

后的所有文件都拷贝到 pahomqtt 文件夹下，目录结构大致如下：

```
third_party
  pahomqtt
    .settings
    Debug
    doc
    MQTTClient
    MQTTClient-C
    MQTTPacket
```

下一步，我们在 pahomqtt 文件夹下面新建 BUILD.gn 文件，用来构建编译。其内容如下：

```
# Copyright (c) 2020 Huawei Device Co., Ltd.

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import("//build/lite/config/component/lite_component.gni")
import("//build/lite/ndk/ndk.gni")

config("pahomqtt_config") {
  include_dirs = [
    "MQTTPacket/src",
    "MQTTPacket/samples",
    "../vendor/hisi/hi3861/hi3861/third_party/lwip_sack/include",
    "../kernel/liteos_m/components/cmsis/2.0",
```

```

    ]
}

pahomqtt_sources = [
    "MQTTPacket/samples/transport.c",
    "MQTTPacket/src/MQTTConnectClient.c",
    "MQTTPacket/src/MQTTConnectServer.c",
    "MQTTPacket/src/MQTTDeserializePublish.c",
    "MQTTPacket/src/MQTTFormat.c",
    "MQTTPacket/src/MQTTPacket.c",
    "MQTTPacket/src/MQTTSerializePublish.c",
    "MQTTPacket/src/MQTTSubscribeClient.c",
    "MQTTPacket/src/MQTTSubscribeServer.c",
    "MQTTPacket/src/MQTTUnsubscribeClient.c",
    "MQTTPacket/src/MQTTUnsubscribeServer.c",
]

```

```

lite_library("pahomqtt_static") {
    target_type = "static_library"
    sources = pahomqtt_sources
    public_configs = [ ":pahomqtt_config" ]
}

```

```

lite_library("pahomqtt_shared") {
    target_type = "shared_library"
    sources = pahomqtt_sources
    public_configs = [ ":pahomqtt_config" ]
}

```

```

ndk_lib("pahomqtt_ndk") {
    if (board_name != "hi3861v100") {
        lib_extension = ".so"
        deps = [
            ":pahomqtt_shared"

```

```

    ]
} else {
    deps = [
        ":pahomqtt_static"
    ]
}

head_files = [
    "../third_party/pahomqtt"
]
}

```

2. 让 hi3861 编译的时候，编译 paho mqtt 软件包

打开 vendor\hisi\hi3861\hi3861\BUILD.gn 文件，在 lite_component("sdk") 中增加
 "../third_party/pahomqtt:pahomqtt_static",
 修改后文件内容如下：

```

17
18 lite_component("sdk") {
19     features = [ ]
20
21     deps = [
22         ["//kernel/liteos_m/components/cmsis",
23          "//kernel/liteos_m/components/kal",
24          "//third_party/cJSON:cjson_static",
25          "//third_party/pahomqtt:pahomqtt_static",
26          "//third_party/a_myparty:a_myparty_static"
27         ]
28     ]
29 }

```

完成以上修改后，就可以开始编译了，然而很不幸的。。。你会发现好多编译报错。

不过没事，我们来一个一个解决。

3. 移植，修改编译报错

打开 third_party\pahomqtt\MQTTPacket\samples\transport.c 文件，这个文件也是我们主要移植的文件，我们需要实现 socket 相关的操作，包括发送、接收数据。其实移植就 3 步。

(1) 首先我们导入几个头文件

```

#include "lwip/ip_addr.h"
#include "lwip/netifapi.h"

```



```
#include "lwip/sockets.h"
```

(2) 其次修改 `transport_sendPacketBuffer` 函数，内容修改后如下：

```
int transport_sendPacketBuffer(int sock, unsigned char* buf, int buflen)
{
    int rc = 0;

    rc = send(sock, buf, buflen, 0);

    return rc;
}
```

(3) 后面编译的时候会报错说 `close` 函数不存在，我们修改 `transport_close` 函数，修改后内容如下：

```
int transport_close(int sock)
{
    int rc;

    rc = shutdown(sock, SHUT_WR);

    rc = recv(sock, NULL, (size_t)0, 0);

    rc = lwip_close(sock);

    return rc;
}
```

(4) 修改完 `transport.c` 文件后，大家编译的时候估计会遇到很多编译错误，都是某个局部变量未使用那种，大家可以修改就行。

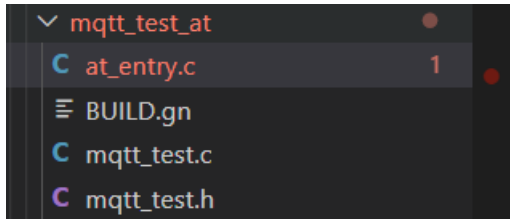
类似于这样的，提示 `buflen` 未使用的错误，大家只需要在代码中随便写个 `buflen = buflen`；即可。

```
int MQTTDeserialize_connack(unsigned char* sessionPresent, unsigned char* connack_rc, unsigned char* buf, int buflen)
{
    MQTTHeader header = {0};
    unsigned char* curdata = buf;
    unsigned char* enddata = NULL;
    int rc = 0;
    int mylen;
    MQTTConnackFlags flags = {0};

    buflen = buflen;
```

3.9.3 编写测试代码

测试代码比较好写。主要是 3 个文件，内容我都贴出来了：



(1) BUILD.gn 文件内容：

```
static_library("mqtt_test") {  
    sources = [  
        "mqtt_test.c",  
        "at_entry.c"  
    ]  
  
    include_dirs = [  
        "../utils/native/lite/include",  
        "../kernel/liteos_m/components/cmsis/2.0",  
        "../base/iot_hardware/interfaces/kits/wifiot_lite",  
        "../vendor/hisi/hi3861/hi3861/third_party/lwip_sack/include",  
        "../foundation/communication/interfaces/kits/wifi_lite/wifiservice",  
        "../third_party/pahomqtt/MQTTPacket/src",  
        "../third_party/pahomqtt/MQTTPacket/samples",  
        "../vendor/hisi/hi3861/hi3861/components/at/src"  
    ]  
}
```

(2) at_entry.c 文件主要是注册了一个 AT 指令，后面大家可以使用 AT+MQTTTEST 指令来测试 MQTT 功能。代码如下：

```
#include <stdio.h>  
#include <unistd.h>  
#include "ohos_init.h"  
#include "cmsis_os2.h"  
#include <unistd.h>  
#include <at.h>
```

```

#include <hi_at.h>

#include "hi_wifi_api.h"
#include "mqtt_test.h"

void mqtt_test_thread(void * argv)
{
    argv = argv;
    mqtt_test();
}

hi_u32 at_exe_mqtt_test_cmd(void)
{
    osThreadAttr_t attr;

    attr.name = "wifi_config_thread";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = 4096;
    attr.priority = 36;

    if (osThreadNew((osThreadFunc_t)mqtt_test_thread, NULL, &attr) == NULL) {
        printf("[LedExample] Falied to create LedTask!\n");
    }

    AT_RESPONSE_OK;
    return HI_ERR_SUCCESS;
}

const at_cmd_func g_at_mqtt_func_tbl[] = {
    {"+MQTTTEST", 9, HI_NULL, HI_NULL, HI_NULL,

```

```

(at_call_back_func)at_exe_mqtt_test_cmd},

};

void AtExampleEntry(void)
{
    hi_at_register_cmd(g_at_mqtt_func_tbl,
sizeof(g_at_mqtt_func_tbl)/sizeof(g_at_mqtt_func_tbl[0]));
}

SYS_RUN(AtExampleEntry);

```

(3) mqtt_test.c 文件则是编写了一个简单的 MQTT 测试代码，具体代码讲解，后面会重新开一篇。其中测试用的 mqtt 服务器是我自己的服务器：**106.13.62.194**

大家也可以改成自己的，也可以直接用我个人的 mqtt 服务器。

```

#include <stdio.h>

#include <unistd.h>

#include "ohos_init.h"
#include "cmsis_os2.h"

#include <unistd.h>
#include "hi_wifi_api.h"
// #include "wifi_sta.h"
#include "lwip/ip_addr.h"
#include "lwip/netifapi.h"

#include "lwip/sockets.h"

#include "MQTTPacket.h"
#include "transport.h"

int toStop = 0;

```

```

int mqtt_connect(void)
{
    MQTTPacket_connectData data = MQTTPacket_connectData_initializer;

    int rc = 0;

    int mysock = 0;

    unsigned char buf[200];

    int buflen = sizeof(buf);

    int msgid = 1;

    MQTTString topicString = MQTTString_initializer;

    int req_qos = 0;

    char* payload = "hello HarmonyOS";

    int payloadlen = strlen(payload);

    int len = 0;

    char *host = "106.13.62.194";
    //char *host = "192.168.1.102";

    int port = 1883;


    mysock = transport_open(host, port);

    if(mysock < 0)
        return mysock;


    printf("Sending to hostname %s port %d\n", host, port);


    data.clientID.cstring = "me";

    data.keepAliveInterval = 20;

    data.cleansession = 1;

    data.username.cstring = "testuser";

    data.password.cstring = "testpassword";


    len = MQTTSerialize_connect(buf, buflen, &data);

    printf("%s %d \r\n", __FILE__, __LINE__);

    rc = transport_sendPacketBuffer(mysock, buf, len);

```

```

printf("%s %d \r\n", __FILE__, __LINE__);

/* wait for connack */

if (MQTTPacket_read(buf, buflen, transport_getdata) == CONNACK)
{
    printf("%s %d \r\n", __FILE__, __LINE__);

    unsigned char sessionPresent, connack_rc;

    if (MQTTDeserialize_connack(&sessionPresent, &connack_rc, buf, buflen) != 1 ||
connack_rc != 0)
    {
        printf("%s %d \r\n", __FILE__, __LINE__);

        printf("Unable to connect, return code %d\n", connack_rc);

        goto exit;
    }
}

else
    goto exit;

/* subscribe */

topicString.cstring = "subtopic";

printf("%s %d \r\n", __FILE__, __LINE__);

len = MQTTSerialize_subscribe(buf, buflen, 0, msgid, 1, &topicString, &req_qos);

printf("%s %d \r\n", __FILE__, __LINE__);

rc = transport_sendPacketBuffer(mysock, buf, len);

printf("%s %d \r\n", __FILE__, __LINE__);

if (MQTTPacket_read(buf, buflen, transport_getdata) == SUBACK) /* wait for suback */
{
    unsigned short submsgid;

    int subcount;

    int granted_qos;

    printf("%s %d \r\n", __FILE__, __LINE__);

    rc = MQTTDeserialize_suback(&submsgid, 1, &subcount, &granted_qos, buf, buflen);

```

```

    if (granted_qos != 0)
    {
        printf("%s %d \r\n", __FILE__, __LINE__);

        printf("granted qos != 0, %d\n", granted_qos);

        goto exit;
    }
}

else

    goto exit;

/* loop getting msgs on subscribed topic */
topicString.cstring = "pubtopic";
while (!toStop)
{
    /* transport_getdata() has a built-in 1 second timeout,
    your mileage will vary */
    printf("%s %d \r\n", __FILE__, __LINE__);

    if (MQTTPacket_read(buf, buflen, transport_getdata) == PUBLISH)
    {
        printf("%s %d \r\n", __FILE__, __LINE__);

        unsigned char dup;

        int qos;

        unsigned char retained;

        unsigned short msgid;

        int payloadlen_in;

        unsigned char* payload_in;

        int rc;

        MQTTString receivedTopic;

        printf("%s %d \r\n", __FILE__, __LINE__);

        rc = MQTTDeserialize_publish(&dup, &qos, &retained, &msgid, &receivedTopic,
                                    &payload_in, &payloadlen_in, buf, buflen);

        printf("message arrived %.*s\n", payloadlen_in, payload_in);

        rc = rc;
    }
}

```

```

    }

    printf("publishing reading\n");

    len = MQTTSerialize_publish(buf, buflen, 0, 0, 0, 0, topicString, (unsigned char*)payload,
payloadlen);

    printf("%s %d \r\n", __FILE__, __LINE__);

    rc = transport_sendPacketBuffer(mysock, buf, len);

}

printf("disconnecting\n");

len = MQTTSerialize_disconnect(buf, buflen);

printf("%s %d \r\n", __FILE__, __LINE__);

rc = transport_sendPacketBuffer(mysock, buf, len);

printf("%s %d \r\n", __FILE__, __LINE__);

exit:

transport_close(mysock);

printf("%s %d \r\n", __FILE__, __LINE__);

rc = rc;

return 0;

}

void mqtt_test(void)
{
    mqtt_connect();
}

```

mqtt_test.h 文件内容:

```

#ifndef __MQTT_TEST_H__
#define __MQTT_TEST_H__

```



```
void mqtt_test(void);
```

```
#endif /* __MQTT_TEST_H__ */
```

到这里就完成了代码部分，可以开始编译了。

3.9.4 MQTT 实验

这里我们需要先下载一个 Windows 电脑端的 MQTT 客户端，这样我们就可以用电脑订阅开发板的 MQTT 主题信息了。

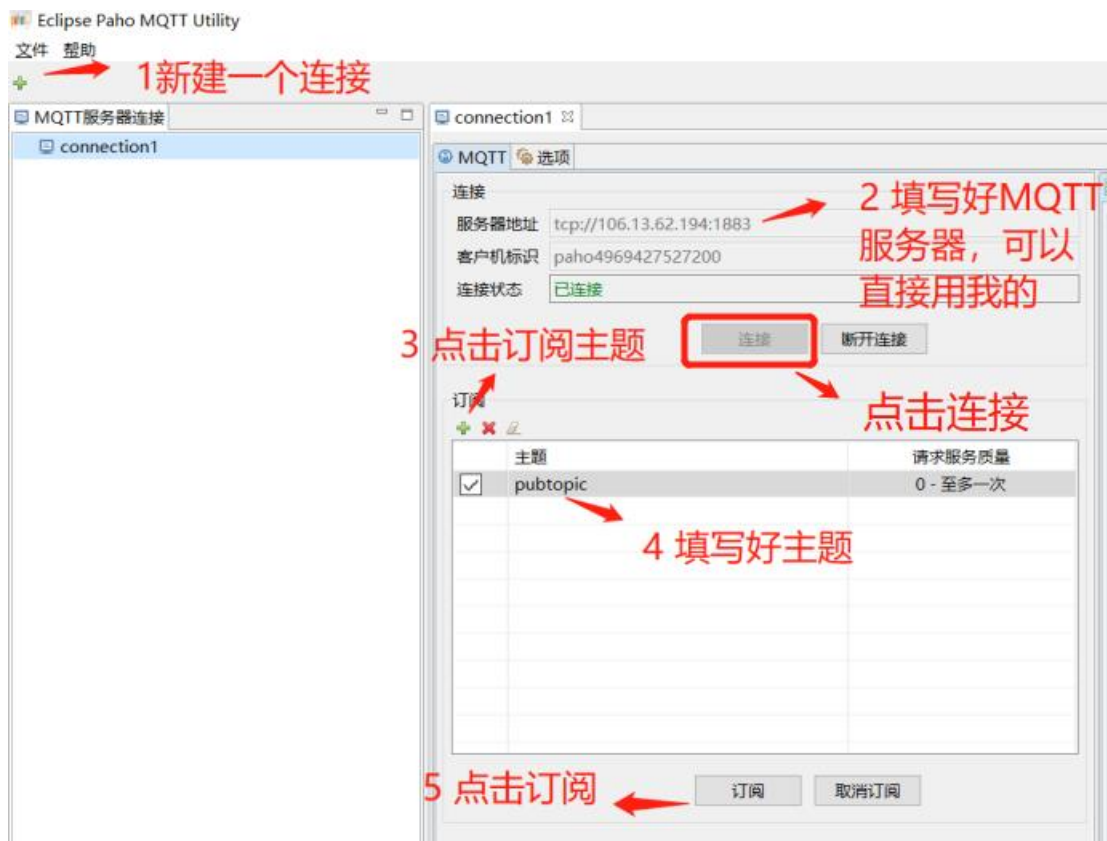
电脑版的 mqtt 客户端下载链接：

<https://repo.eclipse.org/content/repositories/paho-releases/org/eclipse/paho/org.eclipse.paho.ui.app/1.1.1/>

我们选择这一个：



弄完后打开软件，按图操作：



操作完后，我们把编译后程序烧写到开发板，输入如下串口指令，让开发板连接上网络，因为 MQTT 功能需要网络支持。输入如下串口指令：

AT+STARTSTA 开启 STA 模式

AT+CONN="12-203",,2,"07686582488" 连接到路由器，注意 wifi 热点名和密码

用自己的

AT+DHCP=wlan0,1 自带获取 IP 地址

AT+IFCFG 打印查看 IP 地址

串口指令的应答应该如下：

```
AT+STARTSTA
OK
AT+CONN="12-203",,2,"07686582488"
OK
+NOTICE:SCANFINISH
+NOTICE:CONNECTED
AT+DHCP=wlan0,1
OK
AT+IFCFG
+IFCFG:wlan0 ip=192.168.1.103, netmask=255.255.255.0, gateway=192.168.1.1, ip6=FE80::B6C9:B9FF:FEAF:6A8F, HWaddr=b4:c9:b9:af:6a:8f, MTU=1500, LinkStatus=1, RunStatus=1
+IFCFG:lo, ip=127.0.0.1, netmask=255.0.0.0, gateway=127.0.0.1, ip6=::1, HWaddr=00:00:00:00:00:00, MTU=16436, LinkStatus=1, RunStatus=1
OK
```

成功连接上路由器后，请确保路由器是可以上网的。

然后我们输入我们的 MQTT 测试的 AT 指令： AT+MQTTTEST

应该可以看到如下打印：

```
Sending to hostname 106.13.62.194 port 1883
publishing reading
publishing reading
publishing reading
publishing reading
publishing reading
```

此时我们去查看 我们电脑端的 MQTT 客户端软件，可以看到右边已经有接收 MQTT 信息了，主题未 pubtopic，消息内容为 hello HarmonyOS！，说明实验成功。

历史记录 最新主题消息					
事件	主题	消息	服务质量	已保留	时间
已订阅	pubtopic		0		2020-10-23 12:43:04
已接收	pubtopic	hello HarmonyOS	0	否	2020-10-23 12:45:08
已接收	pubtopic	hello HarmonyOS	0	否	2020-10-23 12:45:09
已接收	pubtopic	hello HarmonyOS	0	否	2020-10-23 12:45:10
已接收	pubtopic	hello HarmonyOS	0	否	2020-10-23 12:45:11
已接收	pubtopic	hello HarmonyOS	0	否	2020-10-23 12:45:12

3.9.5 总结

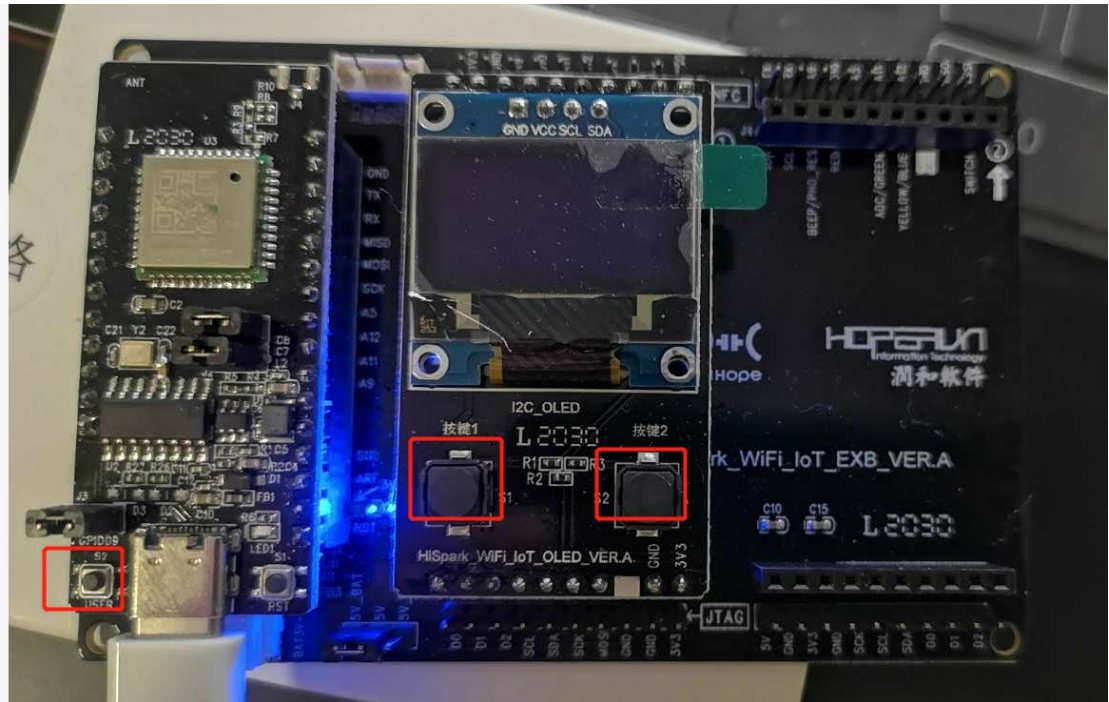
这一次的内容比较多，其中总结起来就 4 步：

- (1) 添加第三方软件包 paho mqtt，关于如何添加第 3 方软件包，我之前有一篇文章已经讲了。
- (2) 移植 paho mqtt
- (3) 编写测试代码，这里我们用的是注册 AT 指令的方式，方便大家使用 AT 指令测试。
- (4) 测试，这里用电脑装 mqtt 客户端程序，去验证。

3.10 ADC 按键的使用

本节主要介绍 Hi3861 的 ADC 功能，顺便实现 ADC 按键检测。这里先看效果吧。

查看开发板，可以看到除了复位按键之外，还有 3 个按键。而查看原理，我们可以看到这个 3 个按键其实都是接的 GPIO5 引脚，而 GPIO5 引脚又可复用为 ADC2 引脚。



故而，我们可以猜测出来我们可以使用 ADC 检测电压，判断出来是哪个引脚被按下了。

看下效果：

当我按下 按键 1 的时候，串口会打印：

```
vlt_min:0.563, vlt_max:0.577  
KEY_EVENT_S1
```

当我按下按键 2 的时候串口会打印：

```
vlt_min:0.963, vlt_max:0.970  
KEY_EVENT_S2
```

当我按下 USER 按键的时候串口会打印

```
vlt_min:0.197, vlt_max:0.204  
KEY_EVENT_S3
```

其中 vlt_min 表示读取到 ADC 值的最小值，

vlt_max 表示读取到 ADC 值的最大值。

由此我们可以看到，按键 1 被按下的时候，ADC 值得范围在 0.563 ~ 0.577

按键 2 按下后，ADC 值在 0.963 ~ 0.970

USER 按键按下后 ADC 值 在 0.197 ~ 0.204

如果没有按键按下，则 ADC 值在 3.227 ~ 3.241

vlt_min:3.227, vlt_max:3.241

代码实现其实很简单。

(1) 引脚初始化

这里由于 GPIO5 默认被复用为串口引脚，这里我们重新修改为普通 GPIO 引脚。初始化代码如下：

```
(hi_void)hi_gpio_init();

hi_io_set_func(HI_IO_NAME_GPIO_5, HI_IO_FUNC_GPIO_5_GPIO); /* uart1 rx */

ret = hi_gpio_set_dir(HI_GPIO_IDX_5, HI_GPIO_DIR_IN);
if (ret != HI_ERR_SUCCESS) {
    printf("==== ERROR =====gpio -> hi_gpio_set_dir1 ret:%d\r\n", ret);
    return;
}
```

(2) 读取 ADC 值

读取 ADC 值的代码页相对简单，这里，我是重复读取 64 次，减少误判。

```
memset_s(g_adc_buf, sizeof(g_adc_buf), 0x0, sizeof(g_adc_buf));

for (i = 0; i < ADC_TEST_LENGTH; i++) {
    ret = hi_adc_read((hi_adc_channel_index)HI_ADC_CHANNEL_2, &data,
HI_ADC_EQU_MODEL_1, HI_ADC_CUR_BAIS_DEFAULT, 0);
    if (ret != HI_ERR_SUCCESS) {
        printf("ADC Read Fail\n");
        return;
    }
    g_adc_buf[i] = data;
}
```

(3) 对读出来的 ADC 值进行判断处理

S1 对应的是按键 1 、 S2 对应的是按键 2 、 S3 对应的是 USER 按键

```
hi_void convert_to_voltage(hi_u32 data_len)
{
    hi_u32 i;
    float vlt_max = 0;
```

```

float vlt_min = VLT_MIN;

float vlt_val = 0;

hi_u16 vlt;

for (i = 0; i < data_len; i++) {
    vlt = g_adc_buff[i];

    float voltage = (float)vlt * 1.8 * 4 / 4096.0;  /* vlt * 1.8 * 4 / 4096.0: Convert code into voltage
*/

    vlt_max = (voltage > vlt_max) ? voltage : vlt_max;

    vlt_min = (voltage < vlt_min) ? voltage : vlt_min;
}

printf("vlt_min: %.3f, vlt_max: %.3f\n", vlt_min, vlt_max);

vlt_val = (vlt_min + vlt_max)/2.0;

if((vlt_val > 0.4) && (vlt_val < 0.6))
{
    if(key_flg == 0)
    {
        key_flg = 1;

        key_status = KEY_EVENT_S1;
    }
}

if((vlt_val > 0.8) && (vlt_val < 1.1))
{
    if(key_flg == 0)
    {
        key_flg = 1;

        key_status = KEY_EVENT_S2;
    }
}

if((vlt_val > 0.01) && (vlt_val < 0.3))

```

```

{
    if(key_flg == 0)
    {
        key_flg = 1;
        key_status = KEY_EVENT_S3;
    }
}

if(vlt_val > 3.0)
{
    key_flg = 0;
    key_status = KEY_EVENT_NONE;
}
}

```

（4）使用

编写好上面代码后，就可以直接在 while 循环中判断按键值了：

```

while(1)
{
    //读取 ADC 值
    app_demo_adc_test();

    switch(get_key_event())
    {
        case KEY_EVENT_NONE:
        {

        }
        break;

        case KEY_EVENT_S1:
        {
            printf("KEY_EVENT_S1 \r\n");

```

```

    }

    break;

    case KEY_EVENT_S2:
    {
        printf("KEY_EVENT_S2 \r\n");
    }
    break;

    case KEY_EVENT_S3:
    {
        printf("KEY_EVENT_S3 \r\n");
    }
    break;

}

usleep(30000);
}

```

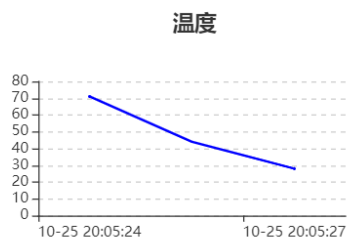
3.11 使用鸿蒙开发板实现第一个物联网项目

通常来说，一个物联网产品应当包括设备、云平台、手机 APP。我将在鸿蒙系统上移植 MQTT 协议、OneNET 接入协议，实现手机 APP、网页两者都可以远程（跨网络，不是局域网的）访问开发板数据，并控制开发板的功能。

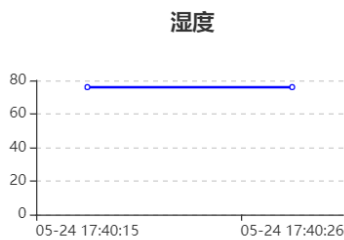
理论上来说，任何以 MQTT 协议为基础的物联网云平台都可以支持接入。

3.11.1 效果

先看下效果，我这边使用的是 OneNET 物联云平台，进入应用，可以看到如下网页界面。该网页的温度、湿度数据由 鸿蒙开发板（Hi3861）上传，同时有一个开关按钮，可以控制开发板的 LED 灯。



Led控制



另外，也提供一个手机 APP，



以上界面比较简陋，但不妨碍我们使用，另外选择 OneNET 云平台的主要原因是接入方式比较简单方便，易于学习，另外一个 OneNET 提供了物联网云平台、手机 APP，不需要大家自己再去实现，可以更多地注意力放在鸿蒙系统开发上。

当我们按下开关按钮时，可以看到开发板打印信息如下：

```
recv data is {"ledSwitch":1}
recv data is {"ledSwitch":0}
recv data is {"ledSwitch":0}
```

云平台发送过来的时一串 json 字符串，key 为 “ledSwitch”，值为 1 时，可以看到开发板的 LED 灯亮，值为 0 时，开发板 LED 灯灭。




3. 11. 2 软件包

欢迎访问 HarmonyOS 技术社区

<https://harmonyos.51cto.com/>

我这边已经将 mqtt 和 onenet 以软件包的形式发布，两个软件包分别是

- (1) onenet——实现 onenet 接入能力
- (2) pahomqtt——实现 MQTT 协议功能

 onenet	2020/10/25 20:17	文件夹
 openssl	2020/9/9 22:57	文件夹
 pahomqtt	2020/10/23 11:37	文件夹





只需要将这两个软件包放到 third_party 文件夹下即可。然后修改

code-1.0\vendor\hisi\hi3861\hi3861\BUILD.gn 文件，将 pahomqtt 和 onenet 加入到编译中

```
lite_component("sdk") {
    features = [ ]

    deps = [ "//kernel/liteos_m/components/cmsis",
              "//kernel/liteos_m/components/kal",
              "//third_party/cJSON:cjson_static",
              "//third_party/pahomqtt:pahomqtt_static",
              "//third_party/a_myparty:a_myparty_static",
              "//third_party/onenet:onenet_static"
            ]
}
```

我们来看下 onenet 文件夹：

 samples	2020/10/25 20:59	文件夹	
 BUILD.gn	2020/10/25 20:34	GN 文件	2 KB
 onenet.h	2020/10/25 20:56	C/C++ Header	6 KB
 onenet_mqtt.c	2020/10/25 20:55	C 文件	15 KB

其中 onenet.h 是头文件

onenet_mqtt.c 是全部源码，它基于 paho mqtt 的 MQTTClient 编程模型。

另外 samples 文件夹下是一个示例代码，代码内容如下：

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
#include "MQTTClient.h"
```

```
#include "onenet.h"
```

```

#define ONENET_INFO_DEVID "597952816"

#define ONENET_INFO_AUTH "202005160951"

#define ONENET_INFO_APIKEY "zgQdlB5y3Bi9pNd2bUYmS8TJHIY="

#define ONENET_INFO_PROID "345377"

#define ONENET_MASTER_APIKEY "gwaK2wJT5wgnSbJYz67CVRGvwkI="


extern int rand(void);


void onenet_cmd_rsp_cb(uint8_t *recv_data, size_t recv_size, uint8_t **resp_data, size_t *resp_size)
{

    printf("recv data is %.*s\n", recv_size, recv_data);

    *resp_data = NULL;

    *resp_size = 0;

}


int mqtt_test(void)
{

    device_info_init(ONENET_INFO_DEVID, ONENET_INFO_PROID, ONENET_INFO_AUTH,
ONENET_INFO_APIKEY, ONENET_MASTER_APIKEY);

    onenet_mqtt_init();

    onenet_set_cmd_rsp_cb(onenet_cmd_rsp_cb);

    while (1)
    {

        int value = 0;

```

```

        value = rand() % 100;

        if (onenet_mqtt_upload_digit("temperature", value) < 0)
        {
            printf("upload has an error, stop uploading");
            //break;
        }
        else
        {
            printf("buffer : {\"temperature\":%d} \r\n", value);
        }
        sleep(1);
    }

    return 0;
}

```

相关源码已经上传到 gitee。

手机 APP 下载：<https://open.iot.10086.cn/doc/art656.html#118>

由于本节内容较多，将分成多个文章，陆续放出，目前规划如下：

- （1）paho mqtt client 移植。其实我之前已经有一篇文章讲了 paho mqtt 的移植，但是那篇文章只是简单的移植，并不支持多任务，这一次 mqtt 移植将支持多任务。
- （2）onenet 协议的移植与实现。主要讲如何在 mqtt 的基础上实现 onenet 接入。
- （3）如何使用 onenet 云平台
- （4）设备（鸿蒙开发板）如何接入到 onenet，实现数据互传。

3.12 分析 helloworld 程序是如何被调用，SYS_RUN 做什么事情

相信大家都已经在鸿蒙系统上实现了自己的第一个 helloworld 程序了。

```

#include "ohos_init.h"
#include "ohos_types.h"
#include "stdio.h"

#include "myparty.h"

void HelloWorld(void)
{
    printf("%s %d \r\n", __FILE__, __LINE__);
    printf("[DEMO] Hello world.\n");

    //myparty_test();
}

SYS_RUN(HelloWorld);

```

代码很简单，编译烧录后，我们就可以看到串口有打印 [DEMO] Hello world.

但是 HelloWorld 函数是在何时被调用的呢？SYS_RUN 又是干嘛的呢？

我们来看下。

1. 启动流程

首先，我们需要分析一下 Hi3861 的启动流程。目前 Hi3861 使用的是 liteOS-M 内核，相关源码厂家没有提供，不过也不妨碍我们。经过我一番查找，可以知道 hi3861 启动内后，第一个入口函数是 app_main 函数。

(vendor\hisi\hi3861\hi3861\app\wifiot_app\src\app_main.c)

大家可以打开，看到 app_main 函数的内容，如下，当然我这里只是简版的，我删除了很多初始化的函数，只保留最终要的。

```

hi_void app_main(hi_void)
{
    //打印 sdk 版本

    const hi_char* sdk_ver = hi_get_sdk_version();

    printf("sdk ver:%s\r\n", sdk_ver);


    //串口、IO 初始化等
    peripheral_init();


    //wifi 初始化

    ret = hi_wifi_init(APP_INIT_VAP_NUM, APP_INIT_USR_NUM);
}

```

```
//鸿蒙系统初始化  
HOS_SystemInit();  
}
```

我们可以看到其实 `app_main` 启动后做了很多工作，包括 `io` 初始化、`wifi` 初始，最后调用了 `HOS_SystemInit()`；进行鸿蒙系统最后的初始化。

那我们看下 `HOS_SystemInit()`；做了什么动作吧。

打开源码 `base\startup\services\bootstrap_lite\source\system_init.c`

可以看到函数内容如下：

```
void HOS_SystemInit(void)  
{  
    MODULE_INIT(bsp);  
    MODULE_INIT(device);  
    MODULE_INIT(core);  
    SYS_INIT(service);  
    SYS_INIT(feature);  
    MODULE_INIT(run);  
    SAMGR_Bootstrap();  
}
```

看起来好像在调用某些模块，仔细看，其中有一个是 `MODULE_INIT(run)`；。顾名思义，好像在初始化或者调用 一个 `run` 模块。那 `run` 模块又是什么呢？我们看下标题的 `SYS_RUN>HelloWorld`）。是不是可以猜测其实 `MODULE_INIT(run)`；就是调用了 `HelloWorld` 呢？

哈哈~其实还真是。大家如果加打印信息，可以看到如下打印。

```
../../base/startup/services/bootstrap_lite/source/system_init.c 38
```

```
../../applications/sample/wifi-iot/app/my_first_app/hello_world.c 9
```

```
[DEMO] Hello world.
```

```
../../base/startup/services/bootstrap_lite/source/system_init.c 40
```

仔细看我加的打印语句，确实是在 38 行执行 `MODULE_INIT(run)`；后才打印 `[DEMO] Hello world.`

所以跟我们猜测的一样。当然没完，我们得分析为啥 是这样。

```

20 void HOS_SystemInit(void)
21 {
22     printf("%s %d \r\n", __FILE__, __LINE__);
23     MODULE_INIT(bsp);
24
25     printf("%s %d \r\n", __FILE__, __LINE__);
26     MODULE_INIT(device);
27
28     printf("%s %d \r\n", __FILE__, __LINE__);
29     MODULE_INIT(core);
30
31     printf("%s %d \r\n", __FILE__, __LINE__);
32     SYS_INIT(service);
33
34     printf("%s %d \r\n", __FILE__, __LINE__);
35     SYS_INIT(feature);
36
37     printf("%s %d \r\n", __FILE__, __LINE__);
38     MODULE_INIT(run);
39
40     printf("%s %d \r\n", __FILE__, __LINE__);
41     SAMGR_Bootstrap();
42
43     printf("%s %d \r\n", __FILE__, __LINE__);

```

2. 链接

我们看下 MODULE_INIT(run); 做了什么。事实上，它只是一个宏。

```

#define MODULE_INIT(name) \
do { \
    MODULE_CALL(name, 0); \
} while (0)

```

而 MODULE_CALL(name, 0); 又可以展开：当然里面的 if 语句的打印是我后面加的

```

#define MODULE_CALL(name, step) \
do { \
    InitCall *initcall = (InitCall *) (MODULE_BEGIN(name, step)); \
    InitCall *initend = (InitCall *) (MODULE_END(name, step)); \
    if (strcmp(#name, "run") == 0) \
    { \
        printf("%s %d \r\n", __FILE__, __LINE__); \
        printf("initcall addr %x , initcall val %x\r\n", initcall, *initcall); \
    } \
    for (; initcall < initend; initcall++) { \
        (*initcall)(); \
    } \
} while (0)

```

我们可以看到 它其实是定义了一个 InitCall 指针，然后指针是这个：

(MODULE_BEGIN(name, step))

而 MODULE_BEGIN 宏其实展开后如下：

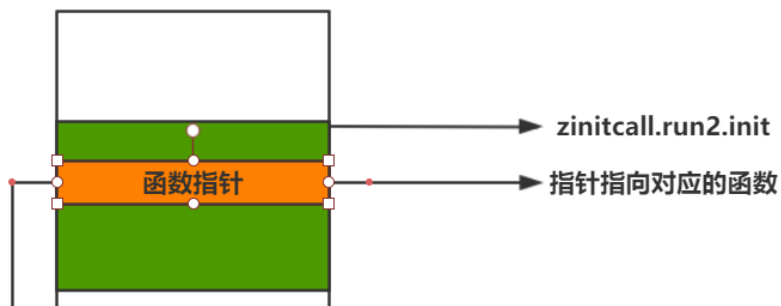
#define MODULE_NAME(name, step) ".zinitcall." #name #step ".init"

我这里再帮大家展开，其实".zinitcall." #name #step ".init" 最后 就是 **.zinitcall.run2.init**

它其实是一种写法，就是说我们代码编译的时候，代码里面有一段地址比较特殊，它的名字是 .zinitcall.run2.init ，也就是说 InitCall 指针 指向的是 .zinitcall.run2.init 代码段的地址。

画个图：

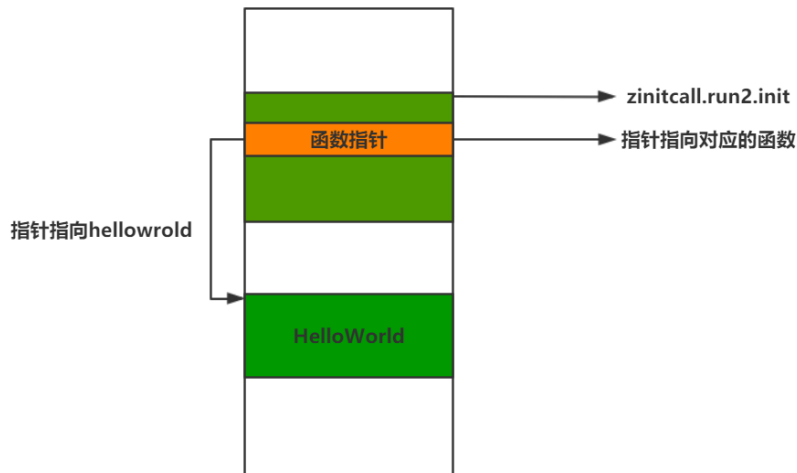
绿色的是.zinitcall.run2.init 代码段，里面存放着函数指针。



好了，到这里大家应该都明白了吧，继续看这个图，其实 这里只不过是把这个代码段里面的所有函数指针都取出来，然后执行一下函数指针指向的函数。

```
#define MODULE_CALL(name, step)
do {
    InitCall *initcall = (InitCall *) (MODULE_BEGIN(name, step)); \
    InitCall *initend = (InitCall *) (MODULE_END(name, step)); \
    if (strcmp(#name, "run") == 0) \
    { \
        printf("%s %d \r\n", __FILE__, __LINE__); \
        printf("initcall addr %x, initcall val %x\r\n", initcall, *initcall); \
    } \
    for (; initcall < initend; initcall++) { \
        (*initcall)(); \
    } \
} while (0)
```

聪明的你应该就猜到了，.zinitcall.run2.init 代码段里面的函数指针，指向的就是 HelloWorld 函数了~~~



到了这里就剩下最后一个问题了：怎么让它指向 HelloWorld 函数。

这里其实就是 SYS_RUN 的功劳了。

我们也来看 SYS_RUN 做了什么，其它也是一个宏，展开过程如下：

```
#define SYS_RUN(func) LAYER_INITCALL_DEF(func, run, "run")

#define LAYER_INITCALL_DEF(func, layer, clayer) \
    LAYER_INITCALL(func, layer, clayer, 2)

#define LAYER_INITCALL(func, layer, clayer, priority) \
    static const InitCall USED_ATTR __zinitcall_##layer##_##func \
        __attribute__((section(".zinitcall." clayer #priority ".init"))) = func
```

我们可以看到，其实 SYS_RUN(HelloWorld) 其实最终结果就是：

```
static const InitCall USED_ATTR __zinitcall_##layer##_##func \
    __attribute__((section(".zinitcall." clayer #priority ".init"))) = HelloWorld
```

看起来很复杂，我们不妨拆解一下：

```
#define LAYER_INITCALL(func, layer, clayer, priority) \
    static const InitCall USED_ATTR __zinitcall_##layer##_##func \
        __attribute__((section(".zinitcall." clayer #priority ".init"))) = func
```

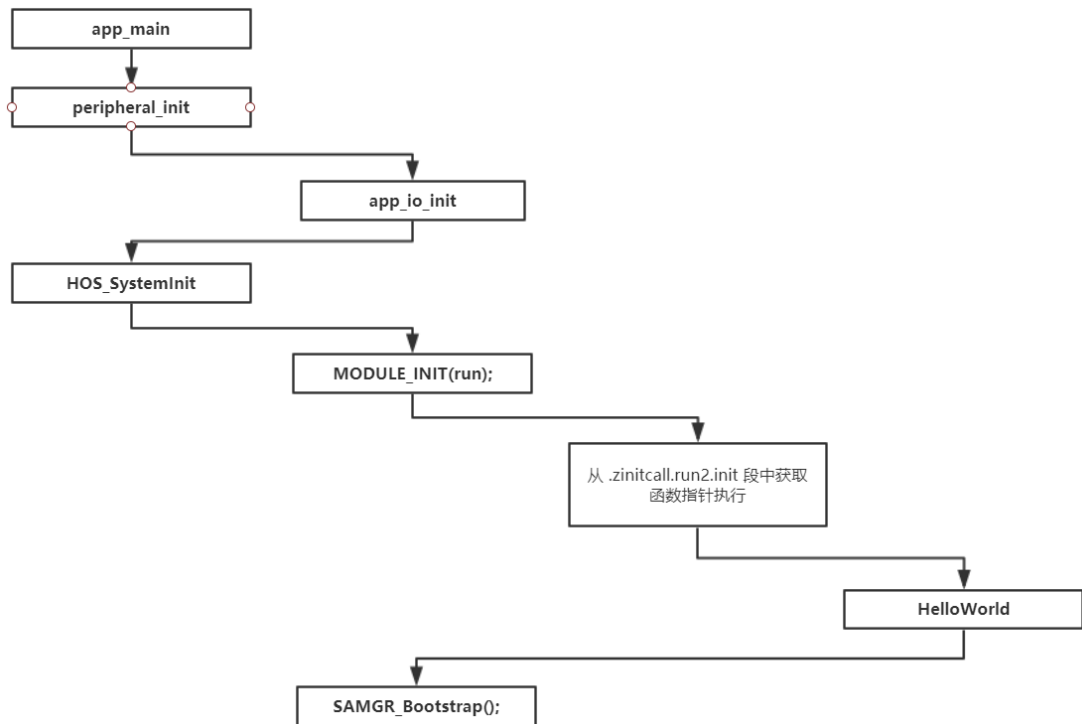
我们先不看红色字体部分，那么结果就是：

```
static const InitCall = HelloWorld
```

是不是很简单，其实就是定义了一个全局变量（函数指针），它指向 HelloWorld。

那红色字体是做什么用呢？它其实就是告诉编译，我这个变量（static const InitCall 变量），很特殊，编译的时候给我编译在 .zinitcall.run2.init 段。

到了，一切都明了了，最后来一张启动流程图总结一下：



3. 忠告

这里有两个忠告：

1、请不要直接在 `SYS_RUN()` 定义的入口函数直接写 `while(1)`

——这个很简单理解了，因为系统启动后，`app_main` 会调用到我们定义的 `SYS_RUN()` 定义的入口函数，比如 `HelloWorld`。如果我们在 `HelloWorld` 函数中写了 `while(1)` 就会导致 `app_main` 后续的代码得不到执行，肯定有问题。

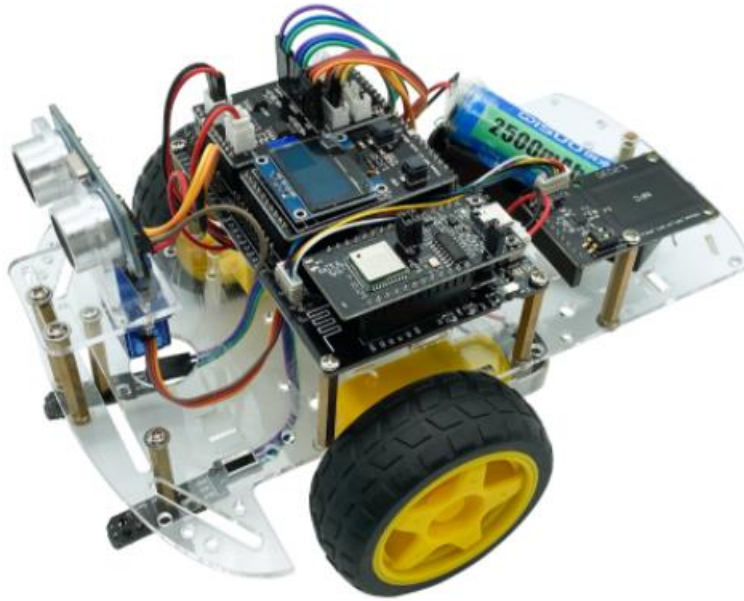
2、`SYS_RUN()` 定义的入口函数创建的线程，请一定要有 `sleep` 动作。

为了解决第一个问题，我们很自然地想到，可以在 `SYS_RUN()` 定义的入口函数 创建线程，这样就可以 `while(1)` 了。哈哈，其实也是有问题，因为 `app_main` 本身也是一个任务，如果我们自己创建的任务优先级特别高，就会导致 `app_main` 任务不会被执行，还是有问题。所以要有 `sleep`，确保 `app_main` 后续地代码能顺利执行下去。

3.13 基于鸿蒙系统 + Hi3861 的 WiFi 小车

首先，我们得有一套 WiFi 小车套件，其实也是 Hi3861 加上电机、循迹模块、超声波等模块。

小车安装完大概是这样：



HiSpark智能小车IoT开发板套

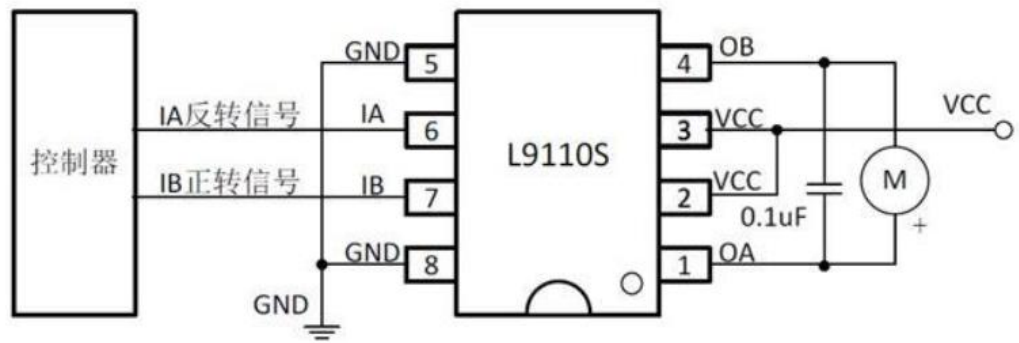
3.13.1 电机驱动

我们这里先只做最简单的，驱动小车的电机，让小车跑起来。

电机的驱动板如下图，目前电机驱动芯片用的是 L9110S 芯片。



典型的应用电路如下图：



我们可以看到，如果要控制电机，我们芯片至少需要 2 路 PWM 信号，一路用于控制正转，一路用于控制反转。

然后我们小车有两个轮子，需要两个电机，所以我们需要 4 路 PWM 信号。

查阅小车资料，可以知道，目前 Hi3861 芯片用来控制电机的 4 路 PWM 分别是：

电机	GPIO 口	PWM 通道
电机 1	GPIO 0	PWM 3
	GPIO 1	PWM 4
电机 2	GPIO 9	PWM 0
	GPIO 10	PWM 1

知道了 PWM 通道和对应的 GPIO 口，我们就可以开始编程了。

首先 PWM 初始化部分：

```
void pwm_init(void)
{
    GpioInit();
    //引脚复用
    IoSetFunc(WIFI_IOT_IO_NAME_GPIO_0, WIFI_IOT_IO_FUNC_GPIO_0_PWM3_OUT);
    IoSetFunc(WIFI_IOT_IO_NAME_GPIO_1, WIFI_IOT_IO_FUNC_GPIO_1_PWM4_OUT);
    IoSetFunc(WIFI_IOT_IO_NAME_GPIO_9, WIFI_IOT_IO_FUNC_GPIO_9_PWM0_OUT);
    IoSetFunc(WIFI_IOT_IO_NAME_GPIO_10, WIFI_IOT_IO_FUNC_GPIO_10_PWM1_OUT);

    //初始化 pwm
    PwmInit(WIFI_IOT_PWM_PORT_PWM3);
    PwmInit(WIFI_IOT_PWM_PORT_PWM4);
    PwmInit(WIFI_IOT_PWM_PORT_PWM0);
    PwmInit(WIFI_IOT_PWM_PORT_PWM1);
}
```

控制小车前进、后退、左转、右转、停止的函数：

```
void pwm_stop(void)
{
    //先停止 PWM

    PwmStop(WIFI_IOT_PWM_PORT_PWM3);

    PwmStop(WIFI_IOT_PWM_PORT_PWM4);

    PwmStop(WIFI_IOT_PWM_PORT_PWM0);

    PwmStop(WIFI_IOT_PWM_PORT_PWM1);
}

//前进
void pwm_forward(void)
{
    //先停止 PWM

    PwmStop(WIFI_IOT_PWM_PORT_PWM3);

    PwmStop(WIFI_IOT_PWM_PORT_PWM4);

    PwmStop(WIFI_IOT_PWM_PORT_PWM0);

    PwmStop(WIFI_IOT_PWM_PORT_PWM1);


    //启动 A 路 PWM

    PwmStart(WIFI_IOT_PWM_PORT_PWM3, 750, 1500);

    PwmStart(WIFI_IOT_PWM_PORT_PWM0, 750, 1500);
}


//后退
void pwm_backward(void)
{
    //先停止 PWM

    PwmStop(WIFI_IOT_PWM_PORT_PWM3);

    PwmStop(WIFI_IOT_PWM_PORT_PWM4);

    PwmStop(WIFI_IOT_PWM_PORT_PWM0);

    PwmStop(WIFI_IOT_PWM_PORT_PWM1);


    //启动 A 路 PWM

    PwmStart(WIFI_IOT_PWM_PORT_PWM4, 750, 1500);
```

```

        PwmStart(WIFI_IOT_PWM_PORT_PWM1, 750, 1500);
    }
    //左转
    void pwm_left(void)
    {
        //先停止 PWM

        PwmStop(WIFI_IOT_PWM_PORT_PWM3);
        PwmStop(WIFI_IOT_PWM_PORT_PWM4);
        PwmStop(WIFI_IOT_PWM_PORT_PWM0);
        PwmStop(WIFI_IOT_PWM_PORT_PWM1);

        //启动 A 路 PWM

        PwmStart(WIFI_IOT_PWM_PORT_PWM3, 750, 1500);
    }
    //右转
    void pwm_right(void)
    {
        //先停止 PWM

        PwmStop(WIFI_IOT_PWM_PORT_PWM3);
        PwmStop(WIFI_IOT_PWM_PORT_PWM4);
        PwmStop(WIFI_IOT_PWM_PORT_PWM0);
        PwmStop(WIFI_IOT_PWM_PORT_PWM1);

        //启动 A 路 PWM

        PwmStart(WIFI_IOT_PWM_PORT_PWM0, 750, 1500);
    }
}

```

最后，要使用 pwm 功能，我们需要修改 vendor\hisi\hi3861\hi3861\build\config\usr_config.mk 文件，把 PWM 功能打开，增加一行 CONFIG_PWM_SUPPORT=y 。如图：

```
CONFIG_I2C_SUPPORT=y
CONFIG_AT_SUPPORT=y
CONFIG_FILE_SYSTEM_SUPPORT=y
CONFIG_UART0_SUPPORT=y
CONFIG_UART1_SUPPORT=y
CONFIG_PWM_SUPPORT=y
# CONFIG_UART2_SUPPORT is not set
# end of BSP Settings
```

3.13.2 WiFi 控制部分

我们在小车上面简单编写一个 UDP 程序，监听 50001 端口号。这里使用的通信格式是 json，小车收到 UDP 数据后，解析 json，并根据命令执行相应的操作，例如前进、后退、左转、右转等，代码如下：

```
if(strcmp("forward", cJSON_GetObjectItem(recvjson, "cmd")->valuestring) == 0)
{
    set_car_status(CAR_STATUS_FORWARD);
    printf("forward\r\n");
}

if(strcmp("backward", cJSON_GetObjectItem(recvjson, "cmd")->valuestring) == 0)
{
    set_car_status(CAR_STATUS_BACKWARD);
    printf("backward\r\n");
}

if(strcmp("left", cJSON_GetObjectItem(recvjson, "cmd")->valuestring) == 0)
{
    set_car_status(CAR_STATUS_LEFT);
    printf("left\r\n");
}

if(strcmp("right", cJSON_GetObjectItem(recvjson, "cmd")->valuestring) == 0)
{
    set_car_status(CAR_STATUS_RIGHT);
    printf("right\r\n");
}

if(strcmp("stop", cJSON_GetObjectItem(recvjson, "cmd")->valuestring) == 0)
{
    set_car_status(CAR_STATUS_STOP);
    printf("stop\r\n");
}
```

电脑端，使用 C#编写一个测试程序，可以手动输入小车的 IP 地址，也可以不输入 IP 地址，这样，电脑端程序会发送广播包给小车，也可以起到控制的功能。



小车的源码，C#控制端的代码均开源，大家可以自由修改，发挥自己的想象，创造出更厉害炫酷的 DIY 产品。

3.13 Hi3861 NV 操作——如何保存数据到开发板，断电不丢失

实际产品开发过程中，我们肯定需要保存一些数据，并且掉电不丢失。由于 hi3861 并没有提供 外置 flash 之类的，也没有所谓的文件系统，无法讲数据保存到文件中。例如很多人在使用我之前写的一篇 WiFi 配网功能后，都会遇到一个问题：我配置了 WiFi 账户密码，但是下次我又得重新配网，能不能把 WiFi 账户密码保存起来？

好，接下来我们来实现这个功能：保存数据到开发板，断电不丢失。

首先我们要使用到 hi3861 的 nv 操作，它支持我们自定义一些数据保存到工厂参数分区，其实就是写入到 hi3861 的 flash 中。

不过这个功能使用挺复杂的，我们以保存 wifi 账户密码为例。

1、修改 mss_nvi_db.xml 文件

打开 vendor\hisi\hi3861\hi3861\tools\nvtool\xml_file\mss_nvi_db.xml 文件，在 Factory 中增加我们的参数：ID 为 0x0B 。


```

memcpy_s(&nv.passwd[0], sizeof(wal_cfg_ssid_my), passwd, passwd_len);

ret = hi_factory_nv_write(NV_ID, &nv, sizeof(wal_cfg_ssid_my), 0);
if (ret != HISI_OK) {
    printf("%x\n", ret);
}
/* 再次读取写入的 NV 值 */
ret = hi_factory_nv_read(NV_ID, &nv, sizeof(wal_cfg_ssid_my), 0);
if (ret != HISI_OK) {
    printf("%x\n", ret);
}
printf("nv read : %d,  ssid  :[%s]  psswd [%s]\n",ret, nv.ssid, nv.passwd);

```

附件我提供了一个 wifi 配网的升级版功能的源码，支持保存 wifi 账号密码。

完成以上操作后，我们就可以发现 wifi 账户密码可以写入到 nv 中了，可以永久保存数据了。查看开机打印：

```

.../.../applications/sample/wifi-iot/app/wifi_config/sta_mode.c 103
nv read : 0,  ssid  :[15919500]  psswd [11206582488]
----- gpio isr demo -----
+NOTICE:SCANFINISH
+NOTICE:CONNECTED
WiFi: Connected

```

可以看到开机后读取到 ssid 和密码正确，并且成功连接到 wifi 热点了。

我们再来看这个 nv 的一些内容吧：

mss_nvi_db.xml 文件记录了所有系统参数的默认值，而且这个文件其实还分组的：

```

br > hisi > hi3861 > hi3861 > tools > nvtool > xml_file > mss_nvi_db.xml
<?xml version="1.0" encoding="utf-8"?>
<HISTUDIO>
  <GROUP NAME="Factory" ID="0x3" FEATURE="1&lt;&lt;0,1&lt;&lt;5" USE
    <NV ID="1" NAME="INIT_CONFIG_XTAL_COMEPESATION" PARAM_NAME="rf_c
    <NV ID="2" NAME="HI_NV_FTM_FLASH_PARTIRION_TABLE_ID" PARAM_NAME
    <NV ID="0x03" NAME="HI_NV_FTM_STARTUP_CFG_ID" PARAM_NAME="hi_nv
    <NV ID="0x04" NAME="HI_NV_FTM_KERNELA_WORK_ID" PARAM_NAME="hi_f
    <NV ID="0x05" NAME="HI_NV_FTM_BACKUP_KERNELA_WORK_ID" PARAM_NAM
    <NV ID="0x06" NAME="HI_NV_FTM_KERNELB_WORK_ID" PARAM_NAME="hi_f
    <NV ID="0x07" NAME="HI_NV_FTM_BACKUP_KERNELB_WORK_ID" PARAM_NAM
    <NV ID="0x08" NAME="HI_NV_FLASH_CRYPT_CNT_ID" PARAM_NAME="hi_fl
    <NV ID="0x09" NAME="HI_NV_FTM_FACTORY_MODE" PARAM_NAME="hi_nv_f
    <NV ID="0x0A" NAME="HI_NV_FTM_UPG_WAIT_MODE" PARAM_NAME="hi_nv_
    <NV ID="0x0B" NAME="INIT_CONFIG_SSID_MY" PARAM_NAME="wal_cfg_ss
  </GROUP>
  <GROUP NAME="Modem" ID="0x1" FEATURE="1&lt;&lt;0,1&lt;&lt;4,1&lt;&
    <!--Range:[0x0001-0x400)-->
    <!--Range:物理层 [0x0001-0x0100) SAL层 [0x0100-0x0150) MAC层 [0x0

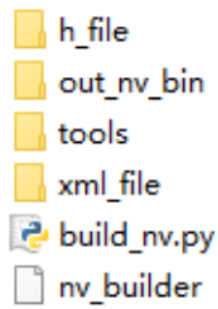
```

可以看到分为 Factory 和 Modem。

NV 模块用于管理系统关键配置信息。NV 存储于 Flash 上，分为以下 2 个区：

- 工厂区 Factory：仅在工厂时使用。
- 非工厂区 Modem：分为以下 2 个区：
 - Keep 区：NV 项在升级后保留原值
 - Modem 区：NV 项在升级后被新版本值替换。

图 2-1 NV 文件目录



NV文件目录说明如下：

- h_file：NV结构体存放位置
- out_nv_bin：生成的NV bin文件
- tools：NV工具文件
- xml_file：NV项和工厂区NV项配置文件
- build_nv.py：NV的编译脚本
- nv_builder：build_nv.py生成的可执行文件，make编译时使用

关于

<GROUP NAME="Factory" ID="0x3" FEATURE="1<<0,1<<5"

USEDMODE="0" PARAM_DEF_FILE=" ../nv/nv_factory_struct_def.txt">

每一项的说明如下：

名称	含义	说明
NAME	NV组	可配置Factory、Keep、Modem
ID	NV组ID	各组ID不能重复
FEATURE	NV组特征	暂未配置使用
USEDMODE	NV组使用模式	暂未配置使用
PARAM_DEF_FILE	NV组参数文件	配置NV参数结构体路径

关于

<NV ID="0x0B" NAME="INIT_CONFIG_SSID_MY"

3.13 MQTT 编程

我们使用的是 paho mqtt 软件包，这里介绍一下怎么使用 mqtt 协议编程。关于鸿蒙系统的 mqtt 移植好的软件包，相关 github 链接如下：

https://gitee.com/qidiyun/harmony_mqtt

这里提供一个简单的编程示例：

这里我们使用 MQTTClient 编程模型，他支持多任务多线程，非常适合用在鸿蒙系统上。

1. 网络初始化

这里定义一个 Network 结构体，然后指定我们的 MQTT 服务器的 IP 和端口号。

```
Network n;  
//初始化结构体  
NetworkInit(&n);  
//连接到指定的 MQTT 服务器 IP、端口号  
NetworkConnect(&n, "XXX.XXX.XXX.XXX", XXXX);
```

2. 设置 MQTT 缓存和启动 MQTT 线程

我们这里使用的是 MQTT 线程功能。

```
MQTTClientInit(&c, &n, 1000, buf, 100, readbuf, 100);  
MQTTStartTask(&c);
```

3. 设置 MQTT 相关参数

接下来我们设置 MQTT 的相关参数，包括版本号、客户端 ID、账户密码等

```
MQTTPacket_connectData data = MQTTPacket_connectData_initializer;  
  
data.willFlag = 0;  
//MQTT 版本为 v3  
data.MQTTVersion = 3;  
//设置客户端 ID  
data.clientID.cstring = opts.clientid;  
//设置客户端账户  
data.username.cstring = opts.username;  
//设置客户端密码  
data.password.cstring = opts.password;  
  
data.keepAliveInterval = 10;  
data.cleansession = 1;  
  
//连接到 MQTT 服务器  
rc = MQTTConnect(&c, &data);
```

4. 订阅主题和接收消息

订阅主题可以使用如下函数

```
MQTTSubscribe(&c, topic, opts.qos, messageArrived);
```

它的函数原型如下:

```
DLLEXP int MQTTSubscribe(MQTTClient* client, const char* topicFilter, enum QoS, messageHandler);
```

其中:

MQTTClient* c : 我们前面定义的 MQTTClient 结构体

const char* topicFilter: 订阅的主题

messageHandler messageHandler : 接收到主题信息后的回调处理函数。

例如上面我们的回调函数是 **messageArrived** , 它的原型如下:

```
void messageArrived(MessageData* md)
{
    MQTTMessage* message = md->message;

    //打印接收到的消息的长度、和消息内容
    printf("%.s", (int)message->payloadlen, (char*)message->payload);
    //fflush(stdout);
}
```

5. 发送消息

发送消息也比较简单, 我们只需要设置好我们的主题和消息内容即可

```
memset(&pubmsg, '\0', sizeof(pubmsg));
//消息内容为 hello harmonyOS !
pubmsg.payload = (void*)"hello harmonyOS !";
//消息长度
pubmsg.payloadlen = strlen((char*)pubmsg.payload);
pubmsg.qos = QOS0;
pubmsg.retained = 0;
pubmsg.dup = 0;

//推送消息, 主题为 pubtest
MQTTPublish(&c, "pubtest", &pubmsg);
```

完整源码如下:

3.13 语音控制小车

之前我们已经有一篇文章讲了如何驱动鸿蒙小车, 通过网络控制小车的运

行。这一篇我们来试点不一样的：使用语音控制鸿蒙小车。

这里我们使用到的是讯飞的语音识别功能，大家可以打开这个网站，申请一个测试账户：

https://www.xfyun.cn/services/lfasr?ch=bd01-b&b_scene_zt=1&renqun_youhua=648371

一般来说我们申请体验包即可，（新用户礼包需要实名认证）：



{ 产品价格 }

以下套餐针对开发者用户调用接口时使用。如果您是个人用户

套餐	体验包	新用户礼包
时长量	5小时	最高50小时
有效期	30天	一年
单价 (元/小时)	免费	免费
总价 (元)	免费	免费
使用服务	立即领取	立即领取

领取完免费使用后，我们创建新应用。



应用名称这些自己根据需求填写

* 应用名称

harmony语音识别测试

* 应用分类

应用-通讯社交-其他


* 应用功能描述

鸿蒙开发板控制识别

提交

[返回我的应用](#)

提交后，我们单击应用，查看详情

应用名称	APPID	分类
 harmony语音识别测试	5fbc8a3b	应用-通讯社交-其他

我们下载 Android SDK 包。

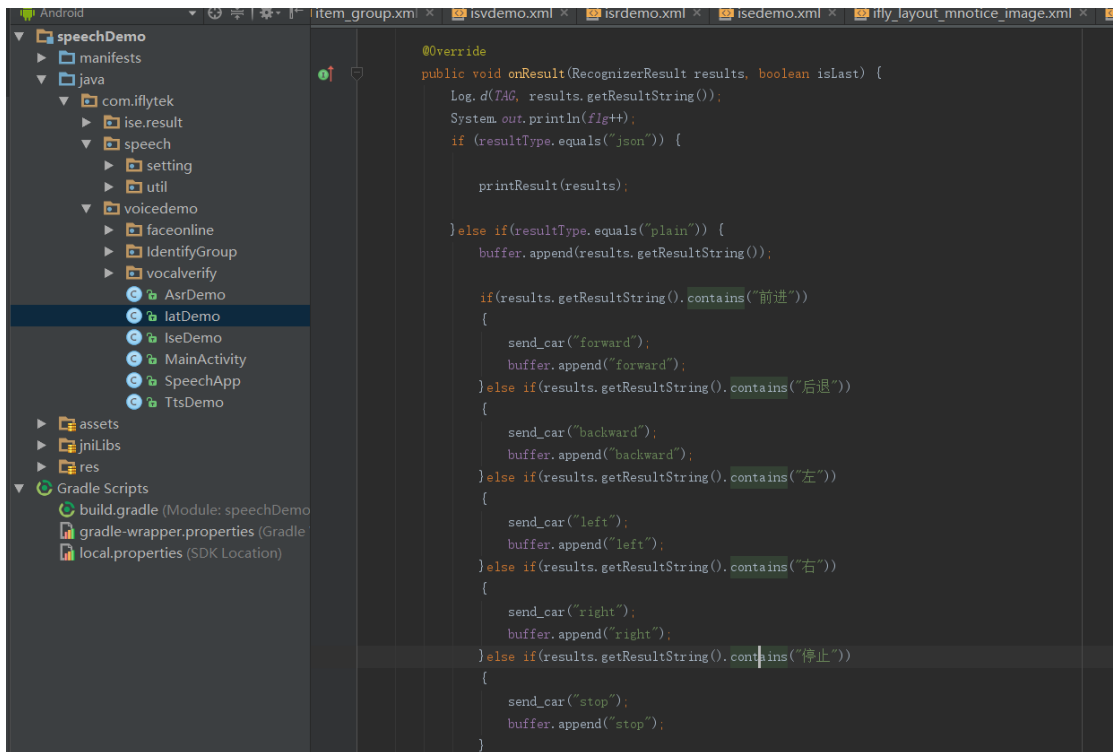
语音听写（流式版） SDK

SDK名称	版本	操作
Android MSC	1140	下载 文档
iOS MSC	1180	下载 文档
Linux MSC	1227	下载 文档
Windows MSC	1126	下载 文档
Java MSC	1021	下载 文档

Android SDK 包的使用可以查看文档。

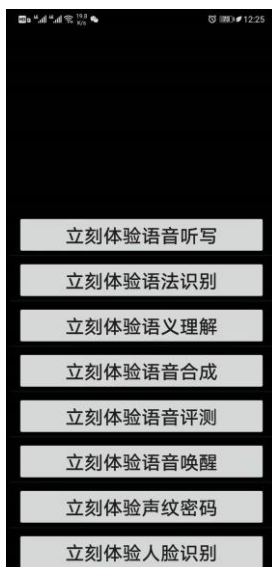
下载完后，我们在 IatDemo.java 文件的 `public void onResult(RecognizerResult results, boolean isLast)` 函数中添加我们控制小车的代码，如图：

我这边会提供我修改后的 IatDemo.java 文件，大家替换即可。



编译 app，然后得到安装包：speechDemo-debug.apk 。安装到手机。

安装后，我们选择“立即体验语音听写”，然后单击开始，说出关键字“前进”“后退”“向左”“向右”，即可看到小车做出相应的动作





代码解析:

其中比较重要的是发送小车控制指令，指令我们采用的是 json 格式，大家也可以根据自己需求，修改其它指令。

```
void send_car(final String msg)
{
    clientThread = new Thread(new Runnable() {
        @Override
        public void run() {
            JSONObject address = new JSONObject();
```

```
        try {
            address.put("cmd", msg);
            address.put("mode", "step");
        } catch (JSONException e) {
            e.printStackTrace();
        }

        try {
            InetAddress targetAddress =
InetAddress.getByName("192.168.1.103");

            DatagramPacket packet = new
DatagramPacket(address.toString().getBytes(),
address.toString().length(), targetAddress, 50001);
            client.send(packet);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
});
clientThread.start();
}
```



欢迎关注 HarmonyOS 技术社区公众号