Hi3861_WiFiIoT 工程的一点理解

liangkz 2021.04.28 v1.6

目录

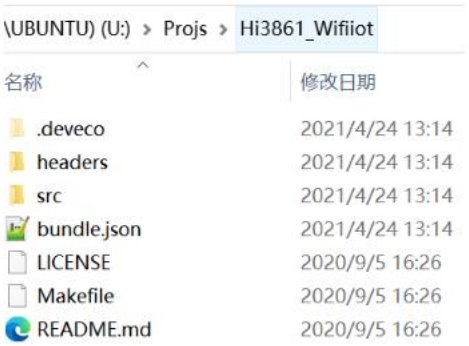
更新记录：

2021.04.23	v1.0	初始版本，前 3 节。
2021.04.25	v1.5	增加第 4 节，理解 IoT 外设控制模块。
2021.04.28	v1.6	增加第 5 节，理解启动恢复子系统.

1.关于工程本身

老规矩，从 0 开始。

在 Linux 环境下的 DevEco IDE 下创建新工程“Hi3861_Wifiiot”，设置如下图，点击“创建”，会在 Projs 目录生成默认的工程。



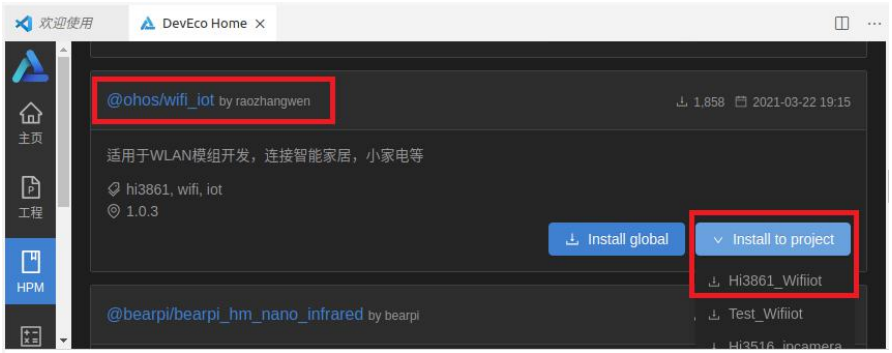
全部文件都查看一遍，看上去只有 **bundle.json** 有点有用信息：


```
{
    "name": "default",
    "version": "1.0.0",
    "description": "This is a default bundle",
    "publishAs": "source",
  "scripts": {
      "build": "make"
  },
  "dirs": {
      "headers": [
          "headers/*.h"
      ],
      "src": [
          "src/*.c"
      ],
      ".": "Makefile"
  },
    .....[省略]
    "dependencies": {},
    "devDependencies": {}
}
```

按照 README.md 的提示，执行“hpm build”，生成了 bin/hello 和 bundle-lock.json，执行“./bin/hello”打印“Hello world”，而 bundle-lock.json 则是空的。

至此，看上去工程跟鸿蒙系统/工程没多少关系，其他文件都可以删掉，唯独“bundle.json”不能删除，要是删除这个文件的话，下面这步就会 install 失败。

在 DevEco IDE 的 HPM 标签下找到“@ohos/wifi_iot”，选择“Install to project”/“Hi3861_Wifiiot”。



安装完成后，就在 Hi3861_Wifiiot 目录下得到了

看上去很干净的目录，暂不用 IDE 一键编译，先试试命令行下的“hpm build”

```
lkz@ubuntu:~/Work/Projs/Hi3861_Wifiiot$ hpm build
[WARN] -    The license of @ohos/gn is gn LICENSE. Notice open-source risks.
[WARN] -    The license of @ohos/gcc_riscv32 is GPL V2. Notice open-source risks.
[WARN] -    The license of @ohos/wifi_iot is NA. Notice open-source risks.
Building: default
make: *** No targets specified and no makefile found.    Stop.    //可能是我删掉了 makefile 的缘故
Build error: Worker stopped with exit code 2
Check error details by "/home/lkz/.hpm/log/debug/debug.2021-04-24-15-40-57.log"

lkz@ubuntu:~/Work/Projs/Hi3861_Wifiiot$ ln -s build/lite/build.py build.py
lkz@ubuntu:~/Work/Projs/Hi3861_Wifiiot$ python build.py wifiiot
[197/197] STAMP obj/vendor/hisi/hi3861/hi3861/run_wifiiot_scons.stamp
ohos wifiiot build success!
```

out 目录下也有正常的输出。

2.ohos_bundles

Hi3861_Wifiiot 项目下，很明显比鸿蒙系统完整代码的目录多了一个 ohos_bundles 文件夹和三个 json 文件，我也注意到在上一步的“Install to project”/“Hi3861_Wifiiot”时，工程目录下最先生成 ohos_bundles 目录。

下面分别看看三个 json 文件和 ohos_bundles 目录都有什么东西。

- **bundle.json**

  看上去比“Install to project”前，多了一点东西：

```
"base": {
    "name": "@ohos/wifi_iot",
    "version": "^1.0.3"
},
```

- **bundle-lock.json**

  看上去列出了本工程所有的组件共计 24 个压缩包的下载地址和 checksum，最后一个"@ohos/wifi_iot"还列出了这个组件依赖于上面的所有组件。

- **product.template.json**

```
"ohos_version": "OpenHarmony 1.0",
"board": "hi3861v100",
"kernel": "liteos_riscv",
"compiler": "gcc",
"subsystem": [],
"vendor_adapter_dir": "//vendor/hisi/hi3861/hi3861_adapter",
"third_party_dir": "//vendor/hisi/hi3861/hi3861/third_party",
```

  很明显的信息。不过为什么要特别列出 "vendor_adapter_dir"？有什么特别的作用吗？还不清楚。

- **ohos_bundles/@ohos/目录**

  很明显这是全工程 24 个组件的独立目录。

  随便进入 **build** 看一下，熟悉的就不说了，看一下 bundle.json：

```
{
    "name": "@ohos/build",
    "version": "1.0.1",
    "publishAs": "code-segment",
    "description": "编译构建提供了一个在 GN 与 ninja 基础上的编译构建框架。
        支持以下功能：构建不同芯片平台的产品。如：Hi3518EV300 平台的 ipcamera 产品，
        Hi3516DV300 平台的 ipcamera 产品，Hi3861 平台的 wifi 模组产品。
        构建 HPM 包管理配置生成的自定义产品。",
    "scripts": {
        "install": "DEST_PATH=${DEP_BUNDLE_BASE}/build &&mkdir -p $DEST_PATH && cp -r ./* $DEST_PATH"
    },
    "keywords": [
        "build"
    ],
    "license": "Apache V2",
    "repository": "",
    "homepage": "",
```

```
        "tags": [
                "build"
        ],
        "ohos": {
                "os": "1.0.0",
                "kernel": "liteos-a,liteos-m",
                "board": "hi3516,hi3518,hi3861"
        }
}
```

看上去都是很直白的，就“scripts”这个，看上去就是要执行脚本命令。

DEP_BUNDLE_BASE 应该是部署 bundle 的 base 目录，也就是项目 Hi3861_Wifiiot 目录本身。

在 Hi3861_Wifiiot/build 目录下递归创建子目录，把当前目录下的所有东西全部递归拷贝到 Hi3861_Wifiiot/build 目录下。

所以 Hi3861_Wifiiot/build 目录就是 Hi3861_Wifiiot/ohos_bundles/@ohos/build 目录的拷贝。

类似的，其他组件基本上也都是这么个情况，至于它们分别拷贝到代码根目录下的什么地方，请自己去仔细查看 bundle.json 进行梳理。

不过三个组件有点例外：**gcc_riscv32**、**gn**、**ninja**。这三个是属于构建编译系统的，他们的 bundle.json 的共同点都是去执行 scripts 目录下的 install.sh 脚本，先去仓库地址下载压缩包，然后解压到同目录下。

前面提到“**@ohos/wifi_iot**”是依赖于其余 23 个组件的，就必须要仔细看一下它的 bundle.json，果然：

```
  "scripts": {
    "dist": "export PATH=$PATH:${DEP_OHOS_gcc_riscv32}/gcc_riscv32/bin:
                          ${DEP_OHOS_gn}/gn:${DEP_OHOS_ninja}/ninja
           && hpm run parse && hpm run select && hpm run connect && hpm run compile",
           "parse": "node ./dist_scripts/parse_platform_hpm.js hi3861v100_liteos_riscv",
           "select": "node ./dist_scripts/select_product.js",
           "connect": "node ./dist_scripts/connect_subsystem.js wifiiot",
           "compile": "ln -sf ${DEP_BUNDLE_BASE}/build/lite/build.py ${DEP_BUNDLE_BASE}/build.py &&
                    cd ${DEP_BUNDLE_BASE} &&python ${DEP_BUNDLE_BASE}/build.py wifiiot",
           "install": "cp product.template.json ${DEP_BUNDLE_BASE}",
           "eco": "echo $target"
  },
```


先把三个构建编译工具所在目录的 bin 添加到环境变量中，再执行 parse、select、connect、compile 命令，前三个命令的脚本都在当前目录的 dist_scripts 内，而 compile 命令则是在代码根目录下先创建 build.py 的软链接，再切换到根目录下执行 python build.py wifiiot 开始构建和编译。根据《鸿蒙系统的编译流程及分析》一文中提到的 Gn+Ninja 的工作原理和步骤，会先去把它所依赖的 23 个组件都编译好，最终生成用于烧录开发板的 bin 文件。

这就很明白了。

3.工程的目录结构

我在《鸿蒙系统的编译流程及分析》(Link: https://harmonyos.51cto.com/posts/4070)一文中大致整理了一下鸿蒙系统的 build、out 目录结构，整个鸿蒙系统的目录结构太复杂了，我的理解还不到位，没法整理出来。不过这个 Hi3861_Wifiiot 工程，是经过 hpm 裁剪了的，总共才 24 个组件，内核也简单了很多，再加上这段时间我调试 Hi3861 的开发板，对工程内文件/代码有了一点点了解，也到了做一次整理的时候了，所以我又整理出了下面这个表格。粗浅的理解，希望能对大家有所帮助，更详细的信息，还是需要各位自己去看 README 和读代码，能亲自在开发板上调试效果会更好。

Hi3861_Wifiiot工程目录			
build.py	build/lite/build.py 编译脚本的软链接		
applications/	示例应用		
sample/wifi-iot/app	BUILD.gn	文件内的描述features只有一个"startup"，其目录也是空的。在此文件添加自己的项目进去编译。	
	startup/	空	
	samgr/	samgr的测试示例程序。可以照着这里的示例程序开发自己的server和client，或者去使用samgr。	
	iothardware/	iot板载硬件控制的示例应用。	
	demolink/	helloworld	
base/	hiviedfx/	Hilog子系统介绍.该仓库用于存放DFX框架的代码，见README.md	
		framwworks/hilog_lite[/mini/featured/]	组件设计模式和代码框架的构建部分，这里也有README.md. featured框架流水日志接口实现/mini框架流水日志接口实现
		interfaces/(innerkits/**kits**)/hilog_lite/	存放所有对外接口定义头文件,即组件被别人调用的接口， innerkits是被鸿蒙系统内部各个组件互相调用的接口， kits是鸿蒙系统外部接口，也就是给鸿蒙应用调用的接口
		services/hiview_lite+hilogcat_lite/	DFX框架服务化注册+流水日志相关服务和命令
		utils/lite	公共基础操作定义实现。包含了mini框架的config配置
	security/	HiChain安全子系统，见README.md	
		framwworks/hichainsdk_lite/	
		interfaces/(innerkits/kits)/hichainsdk_lite/	
	startup/	系统属性模块，根据鸿蒙CDD文档提供获取设备信息的接口，如：产品名、品牌名、厂家名等，同时提供设置/读取系统属性的接口。	
		frameworks/syspara_lite/	系统属性模块源码文件
		hals/syspara_lite/	系统属性模块硬件抽象层头文件
		interfaces/kits/syspara_lite/	系统属性模块对外接口,由main函数来调用，启动服务框架。
		services/bootstrap_lite/	启动恢复子系统，system_init.c
	iot_hardware/	IoT外设控制模块提供对外围设备的操作能力：ADC，AT，FLASH，GPIO，I2C，I2S，PARTITION，PWM，SDIO，UART，WATCHDOG等。使用C语言编写，目前仅支持Hi3861开发板。	
		interfaces/kits/wifiiot_lite/	IoT外设控制模块接口【提供API给上层或其他模块调用】
		frameworks/wifiiot_lite/src/	IoT外设控制模块实现【调用下层提供的HAL接口，来实现功能，同时隐藏下层的实现细节】
		hals/wifiiot_lite/	HAL适配层接口
		//vendor/hisi/hi3861/hi3861_adapter/hals/iot_hardware/wifiiot_lite/	HAL适配层接口的实现【这里再次调用 //vendor/hisi/hi3861/hi3861/latform/drivers/ 对应模块的API去操作硬件/寄存器实现最终的功能】
foundation/			
	communication/	近场设备间统一的分布式通信能力管理	
		frameworks/wifi_lite/	分布式通信子系统 README.md
		interfaces/kits/softbus_lite+wifi_lite/	
		services/softbus_lite/	
	distributedsch edule/	系统服务框架子系统，README.md	
		interfaces/**innerkits+kits**/samgr_lite/	
		services/samgr_lite/	
kernel/	liteos_m/	内核子系统	
ohos_bundles/	本工程所有组件的集合，分别分布在工程的各个角落。		
test/	XTS认证子系统		
third_party/	移植过来的三方组件/库/模块等等		
	cJSON/		
	cmsis/	定义了cmsis_os2.h，具体实现在：//kernel/liteos_m/components/cmsis/2.0/	
	unity/		
utils/	公共基础库存放鸿蒙通用的基础组件。这些基础组件可被**鸿蒙各业务子系统**及**上层应用**所使用。		
	native/lite/	公共基础库根目录,LiteOS-M平台：KV存储、文件操作、定时器，提供统一的操作接口，屏蔽对底层不同芯片组件	
		file/	文件接口实现
		hals/file/	文件操作硬件抽象层头文件
		include/	公共基础库对外接口文件[applications上面的开发会include这里的头文件]
		js/builtin/ deviceinfokit/ + filekit/ + kvstorekit/	设备信息Kit+文件Kit+KV存储Kit
		kal/timer/	Timer的KAL实现
		kv_store/innerkits + src	KV存储内部接口+KV存储源文件【为**应用程序**提供KV存储机制】
		timer_task/	Timer实现
vendor/huawei/	设备厂商 huawei 提供的组件列表		
	wifi-iot	wifi-iot 组件的详细信息 bundle.json	
		hals/utils/sys_param/	系统属性参数，包括产品类型、品牌、制造厂商、产品序列号等等，//base/startup/frameworks/syspara_lite/ 最终会调用这里的接口
		hals/utils/token/	提供符号的相关读写操作，需要OEM实现，//base/startup/frameworks/ypara_lite/ 最终会调用这里的接口
vendor/hisi/hi3861/	芯片厂商 hisi 提供的对 hi3861 芯片/平台的支持		
	hi3861_adapter/hals/	communication/wifi_lite/	wifiservice 的实现代码
		iot_hardware/wifiiot_lite/	wifiiot_lite的相关hals接口的实现 HALxxx()，里面会调用 //vendor/hisi/hi3861/hi3861/platform/drivers/ 目录下对应模块的hi_xxx()接口，而hi_xxx()里面则是直接操作芯片寄存器了。
		utils/file/	公共基础功能的文件操作接口的实现，HALxxx()，里面调用hi_xxx函数实现功能。
	hi3861/	hi3861 的SDK	
		3rd_sdk/demolink/libs/	空
		app/demo/	不编译 demo 的 init/app_io_init.c + src/app_main.c
		app/wifiiot_app/	LiteOS_M 应用层源代码 init/app_io_init.c + src/**app_main.c**
		boot/	boot相关的C源代码，包括：公共部分+flashboot+loaderboot
		build/	编译LiteOS_M内核的入口
		*/basebin/burn_for_erase_4k.bin	
		*/build_tmp	编译的临时文件？？
		*/config/**usr_config.mk**	**编译内核的用户配置，比如支持I2C/SPI/PWM等等**
		*/libs/	预编译的库？
		*/link/	boot 相关的汇编源代码，.lds+.ld.S
		*/make_scripts	make脚本
		*/scripts	python脚本
		components/	liteos-m操作系统层面的功能组件
		*/at	at 源代码
		*/hilink+iperf2+wifi	这三个都只提供.h文件
		config/	
		*/diag + nv	
		*/system_config.h	
		include/	
		license/	
		ohos/libs/	编译工程时生成的组件库文件都在这里
		output/bin/	**Hi3861_wifiiot_app_allinone.bin 等**
		third_party/	lwip_sack + mbedtls + u-boot-v2019.07 头文件+源文件
		tools/	可执行软件和python脚本
		platform/	平台提供的功能组件的具体实现
		*/drivers/**	具体模块的实现，直接写寄存器了。包括adc/flash/i2c/uart等
		*/include/**	头文件
		*/os/Huawei_LiteOS/**	都是头文件
		*/system/**	包括 cfg/cpup/partition_table/upg 的源码
		BUILD.gn	见文件内容。//vendor/hisi/hi3861/hi3861/ 编译生成“wifiiot_sdk”子系统，包含组件“sdk”，而sdk组件依赖于三个部分：”//kernel/liteos_m/components/cmsis”, ”//kernel/liteos_m/components/kal”, ”//third_party/cJSON:cjson_static”
build/	编译构建子系统，见下图，或更详细见**README.md**，里面有添加一个自定义的组件的示例。		

```
build/lite                          # 编译构建主目录
├── config/                         # 编译相关的配置项
│   ├── boards/                     # 开发板相关的变量定义。包括：开发板名、目标架构、目标CPU等
│   ├── component/                  # OpenHarmony组件相关的模板定义。包括：静态库、动态库、扩展组件、模拟器库等
│   ├── kernel/                     # OpenHarmony内核的编译变量定义与配置参数
│   └── subsystem/                  # OpenHarmony子系统列表
├── ndk/                            # NDK相关编译脚本与配置参数
├── platform                        # 平台相关的配置文件
│   ├── hi3516dv300_liteos_a/       # hi3516dv300, liteos_a平台。包括：平台配置参数、平台已有产品模板等。
│   ├── hi3518ev300_liteos_a/       # hi3518ev300, liteos_a平台。包括：平台配置参数、平台已有产品模板等。
│   └── hi3861v100_liteos_riscv/ # hi3861v100, liteos_riscv平台。包括：平台配置参数、平台已有产品模板等。
├── product/                        # 产品全量配置表。包括：配置单元、子系统列表、编译器等。
├── toolchain/                      # 编译工具链相关。包括：编译器路径、编译选项、链接选项等。
└── tools/                          # 编译构建依赖的工具。包括：mkfs等。
```

```
out/
└── wifiiot                                  # 产品名
    ├── gen/                                 # 空
    ├── libs/                                # 静态库文件夹
    ├── obj/                                 # 编译过程产生的.o文件、ninja文件和时间戳文件
    ├── suites/                              #
    ├── build.log                            # 编译日志
    ├── .ninja_log
    ├── .ninja_deps
    ├── args.gn                              # gn编译，用户自定义变量
    ├── build.ninja
    ├── build.ninja.d
    ├── toolchain.ninja
    ├── Hi3861_boot_signed_B.bin             # 带签名的bootloader备份文件
    ├── Hi3861_boot_signed.bin               # 带签名的bootloader文件
    ├── Hi3861_loader_signed.bin             # 烧写工具使用的加载文件，烧写建议使用 “Hi3861_demo_allinone.bin
    ├── Hi3861_wifiiot_app_allinone.bin      # 产线工装烧写文件(已经包含独立烧写程序 和loader程序)
    ├── Hi3861_wifiiot_app.asm               # Kernel asm文件
    ├── Hi3861_wifiiot_app_burn.bin          # 烧写文件，烧写程序建议使用 “Hi3861_wifiiot_app_allinone.bin”
    ├── Hi3861_wifiiot_app_flash_boot_ota.bin # Flash Boot升级文件
    ├── Hi3861_wifiiot_app.map               # Kernel map文件
    ├── Hi3861_wifiiot_app_ota.bin           # Kernel 升级文件
    ├── Hi3861_wifiiot_app.out               # Kernel 输出文件
    └── Hi3861_wifiiot_app_vercfg.bin        # 安全启动开启的时候，配置boot和kernel版本号，防版本回滚
```

4.理解 IoT 外设控制模块

Hi3861 开发板，最主要的功能，就是利用 IoT 外设控制模块提供对外围设备的操作能力，对外围设备操作接口包括了 GPIO， I2C， I2S 等等，详情见 README。

这一节我们就从上到下看一下是怎么实现这些控制的。

我们先看一下官方提供的应用示例程序：

applications\sample\wifi-iot\app\iothardware\ **BUILD.gn** + **led_example.c**

4.1 BUILD.gn 的展开

.c 文件等下再看，先看 BUILD.gn：

```
include_dirs = [
    "//utils/native/lite/include",                          # A
    "//kernel/liteos_m/components/cmsis/2.0",               # B
    "//base/iot_hardware/interfaces/kits/wifiiot_lite",     # C
]
```

- #A：进到 //utils/native/lite 目录，先看 readme。

  公共基础库存放 OpenHarmony 通用的基础组件。这些基础组件可被 OpenHarmony 各业务子系统及上层应用所使用。

  公共基础库在不同平台上提供的能力：

  LiteOS-M 内核(Hi3861 平台)：KV 存储、文件操作、IoT 外设控制、Dump 系统属性。

  LiteOS-A 内核(Hi3516、Hi3518 平台)：KV 存储、定时器、数据和文件存储的 JS API、Dump 系统属性。

```
utils/native/lite/              # 公共基础库根目录
├── file/                       # 文件接口实现，UtilsFileXxx()
├── hals                        # HAL目录
│   └── file                    # 文件操作硬件抽象层头文件，声明HalFileXxx()给上面UtilsFileXxx()调用
├── include                     # 公共基础库对外接口的头文件
│   ├── hos_errno.h/ohos_errno.h    # The error codes are applicable to both the application and kernel
│   │   hos_init.h/ohos_init.h      # (*InitCall) 以及 SYS_RUN()一组宏的定义
│   │   hos_types.h/ohos_types.h    # The data types are applicable to both the application and kernel.
│   ├── kv_store.h              # Provides functions for obtaining, setting, and deleting a key-value pair.
│   ├── utils_config.h          # Represents the configuration file of the utils subsystem.
│   ├── utils_file.h            # 上面文件接口的声明和部分宏的定义，UtilsFileXxx()
│   └── utils_list.h            # Doubly linked list，双向链表的实现和部分宏定义
│
├── js                          # JS API目录
│   └── builtin
│       ├── common
│       ├── deviceinfokit       # 设备信息Kit
│       ├── filekit             # 文件Kit
│       └── kvstorekit          # KV存储Kit
├── kal                         # KAL目录
│   └── timer                   # Timer的KAL实现
├── kv_store                    # KV存储实现
│   ├── innerkits               # KV存储内部接口
│   └── src                     # KV存储源文件
└── timer_task                  # Timer实现
[鸿蒙完整系统下还有更多子目录]
```

include 目录包含了很重要的头文件，应用开发或者鸿蒙系统内部其他模块，要调用这个公用基础库提供的功能时，都需要包含这个路径的头文件，其中：

1. hos_init.h/ohos_init.h 就定义了 SYS_RUN() 这一组宏，也就是下面 led_example.c 中使用到的 SYS_RUN(LedExampleEntry)；按这里的定义一路展开，最终会在通过.zinitcall.run2.init 段中的 __zinitcall_run_app_entry 去执行 LedExampleEntry()。

唐佐林老师的《SYS_RUN()和 MODULE_INIT()之间的那些事》(Link: https://harmonyos.51cto.com/posts/2017)有非常详细的分析，请去看原文。

2. utils_file.h 定义了经过 Utils 封装的文件操作接口，UtilsFileXxx() 的实现，就在上一级的 file/ 目录下，

```
UtilsFileXxx()
{
    return HalFileXxx();
}
```

而这个 HalFileXxx() 硬件抽象层的接口，就是下图的 KAL 这个位置，也见 #B 的截图：

HalFileXxx() 再下去就到了 LiteOS_M 内核提供的文件操作接口 hi_xxx()了见 #B 的截图。

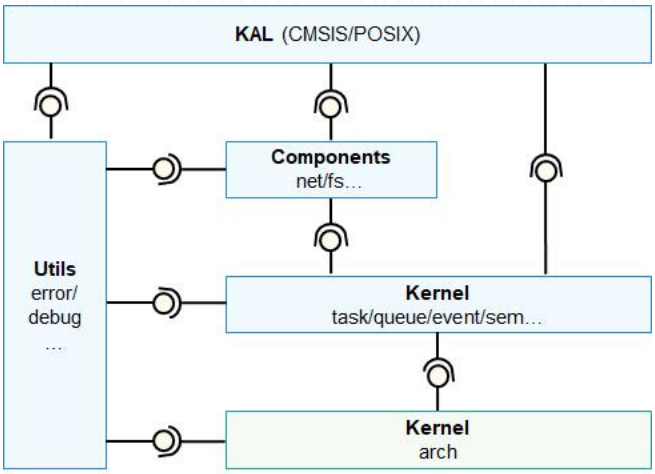
3. utils_list.h 定义和实现了一个双向链表结构，这个结构非常重要。

刚好我这两天看到《v01.10 鸿蒙内核源码分析(双向链表篇)》（Link：https://harmonyos.51cto.com/posts/3925）也推荐去看原文。

公用基础库的目录结构如上图，细节就不继续展开了，请自行阅读理解。

- #B：进入//kernel/liteos_m/目录，先看 readme。

下面这张“LiteOS-M 核内核架构图”，结合 #A 上面的截图（或者完整的鸿蒙系统架构图），要深入理解一下：



KAL(Kernel Abstract Layer，内核抽象层)，是鸿蒙系统框架层(Framework)与内核(LiteOS_M、LiteOS_A、Linux 内核)之间的接口，鸿蒙系统框架层与内核层是通过 KAL 接口进行隔离和解耦的。

KAL 可以按照 cmsis 标准或者 posix 标准来实现 Framework 和 kernel 的对接，目前代码看到的是按 cmsis-rtos v2 标准来实现的。

【这里要注意，鸿蒙系统完整代码下的 kernel/liteos_m/ 与本项目的 kernel/liteos_m/ 目录，结构上存在一些差异，但基本上不影响理解，我是两者同时对比着看的，**鸿蒙系统完整代码的目录结构**(如下)明显更加合理：

```
/kernel/liteos_m
├── components          # 可选组件
│   ├── cppsupport      # C++支持
│   └── cpup            # CPUP功能
├── kal                 # 内核抽象层
│   ├── cmsis           # cmsis标准接口支持
│   └── posix           # posix标准接口支持
├── kernel              # 内核最小功能集支持
│   ├── arch            # 内核指令架构层代码
│   │   ├── arm         # arm32架构的代码
│   │   └── include     # 对外接口存放目录
│   ├── include         # 对外接口存放目录
│   └── src             # 内核最小功能集源码
├── targets             # 板级工程目录
├── utils               # 通用公共目录
```

详见 README。

但在本工程 Hi3861_Wifiiot 里，还是按照工程的实际目录来分析。】

进入 components 目录：

kal 子目录，看上去实现了一组 KalXxx()接口，主要是 timer 相关的，都是调用了内核的 LOS_Xxx()来实现的。

cmsis 子目录，这就是按照 cmsis-rtos v2 标准来实现的一组接口，进去看一下，主要是获取内核信息、线程管理、timer 管理的。我们在 led_example.c 中调用的创建线程的接口 osThreadNew()就是在这里实现的。

关于 cmsis-rtos v2 标准及相关接口，建议看官网的 Reference：

https://www.keil.com/pack/doc/cmsis/rtos2/html/group__CMSIS__RTOS.html

CSDN 上 **XinLiBK** 将其翻译成中文了：

https://blog.csdn.net/u012325601/category_9274156.html

我在《鸿蒙系统的启动流程 v3.0》一文中提到，我验证确认了 Hi3861_Wifiiot\kernel\liteos_m\目录下的 **kernel** 虽然没有编译，但是 **components** 是有编译的，可以在里面加 log，跑起来可以打印 log。

- #C：进入//base/iot_hardware/目录，先看 readme。

  IoT 外设控制模块提供对外围设备的操作能力。

  本模块提供如下外围设备操作接口：ADC, AT, FLASH, GPIO, I2C, I2S, PARTITION, PWM, SDIO, UART, WATCHDOG 等。

  IoT 外设控制模块使用 C 语言编写，目前仅支持 Hi3861 开发板。

源代码目录结构不够详细，看我再来个稍微完整的表格，再理一下他们之间的调用关系：

//base/iot_hardware/ 目录结构：IoT外设控制模块提供对外围设备的操作能力：ADC, AT, FLASH, GPIO, I2C, I2S, PARTITION, PWM, SDIO, UART, WATCHDOG等。使用C语言编写，目前仅支持Hi3861开发板。				应用层 APP 调用 B
interfaces/	IoT外设控制模块接口，即被别人调用的接口(kits+innerkits)，有些组件只有其中之一。（如 foundation/distributedschedule/ 就有两个，具体有什么差别，我还没开始研究）			
	kits/	IoT外设控制模块接口,kits是对外接口，给上层鸿蒙应用调用。		
		wifiiot_lite/ wifiiot_*.h	具体某类外设控制模块的接口声明，如 GpioXxx()	B 的声明
	innerkits/	innerkits是对内接口，给系统内部各个组件互相调用的接口。【本例无】		
frameworks/	wifiiot_lite/	IoT外设控制模块实现，即具体的c/cpp代码就在这里。		
		src/wifiiot_*.c	具体某类外设控制模块的实现【调用下层提供的HAL接口，来实现功能，同时对上层隐藏下层的实现细节,如 GpioXxx()，将会调用 HalGpioXxx()来实现。】	B 的实现，调用 C
hals/	wifiiot_lite/	HAL适配层接口		
		hal_wifiiot_*.h	具体某类外设控制模块的HAL层接口的声明，如 HalGpioXxx()	C 的声明
//vendor/hisi/hi3861/hi3861_adapter/hals/iot_hardware/wifiiot_lite/			HAL适配层接口的实现，如 HalGpioXxx()。它将再次调用内核提供的 hi_gpio_xxx() 来实现。	C 的实现，调用 D
//vendor/hisi/hi3861/hi3861/include/			内核API hi_gpio_xxx()的声明	D 的声明
//vendor/hisi/hi3861/hi3861/platform/drivers/			hi_gpio_xxx() 操作硬件/寄存器实现最终的功能	D 的实现，操作硬件

这里 include 的 //base/iot_hardware/interfaces/kits/wifiiot_lite 就是上表中“B 的声明”，上下层之间的调用关系见最右边一列。

4.2 led_example.c 的展开

好像把上面 4.1 小节理解透了，led_example.c 也就自然理解了，这里就一笔带过。

开始：

#include 公用基础库头文件

#include KAL 层提供的 cmsis 线程管理相关头文件

#include 框架层封装的 IoT 控制模块头文件

1. 通过公用基础库提供的宏 SYS_RUN(LedExampleEntry)引导进入 LedExampleEntry；
2. LedExampleEntry 不能做堵塞类事情，因为会影响其他应用的启动，
   调用 cmsis 接口创建一个线程 LedTask，专门处理控制 Led 灯开关的事情。
3. LedTask 调用框架层 IoT 控制相关接口（上图中最右列的**调用 B** 这一步），然后逐层向下调用，
   最终实现 LED 灯的开关控制。

结束。

4.3 IoT 外设控制模块的整体理解

官方提供的上述示例程序，仅仅展示了如何通过 GPIO 去控制 Hi3861 WLAN 主板上的一颗 LED 灯。

整套开发板还有其他的扩展板，包括通用底板、显示屏板、NFC 板、智能三色灯板等等（官方资料包中还提供了更多的扩展硬件功能的指导说明），板子上不同的硬件分别可以通过不同的接口去进行控制。

要调试某个板子的硬件，需要先去 //vendor/hisi/hi3861/hi3861/build/config/usr_config.mk 打开对应的 SUPPORT 宏：

```
# BSP Settings
#
# CONFIG_I2C_SUPPORT is not set
# CONFIG_I2S_SUPPORT is not set
# CONFIG_SPI_SUPPORT is not set
# CONFIG_DMA_SUPPORT is not set
# CONFIG_SDIO_SUPPORT is not set
# CONFIG_SPI_DMA_SUPPORT is not set
# CONFIG_UART_DMA_SUPPORT is not set
# CONFIG_PWM_SUPPORT is not set
# CONFIG_PWM_HOLD_AFTER_REBOOT is not set
CONFIG_AT_SUPPORT=y
CONFIG_FILE_SYSTEM_SUPPORT=y
CONFIG_UART0_SUPPORT=y
CONFIG_UART1_SUPPORT=y
# CONFIG_UART2_SUPPORT is not set
# end of BSP Settings
```

这些宏会在系统启动 app_main()的 peripheral_init()外围设备初始化阶段，对相关控制接口和数据接口做初始化，之后就可以进行调试了，调试套路和相关控制流程，与上面 LED 灯的控制大同小异。

整套开发板的的详细资料，可以去润和官网去下载：

http://www.hihope.org/download/download.aspx?mtt=8

资料中包含了硬件的数据手册、原理图、demo code 以及更多的扩展说明，看起来可玩性还是蛮高的。

5.理解启动恢复子系统

这是一个非常重要的子系统，我在之前《鸿蒙系统的启动流程》一文中做过一些简单的分析，建议先去看一下《鸿蒙系统的启动流程 v3.0》Part 1/2 的 3/4 章节。这里就先到鸿蒙系统来整体看一下它具体是怎么回事，然后再回到本工程来对比看 hpm 的裁剪给我们留下了什么。

仍然是先看官方 readme 和重新整理目录结构。

启动恢复子系统负责从内核启动之后到应用启动之前的系统关键服务进程的启动过程以及设备恢复出厂设置的功能。涉及以下组件：

- init 启动引导组件

  init 启动引导组件对应的进程为 init 进程，是内核完成初始化后启动的第一个用户态进程。init 进程启动之后，读取 init.cfg 配置文件，根据解析结果，执行相应命令并依次启动各关键系统服务进程，在启动系统服务进程的同时设置其对应权限。

- appspawn 应用孵化组件

  负责接收用户程序框架的命令孵化应用进程，设置新进程的权限，并调用应用程序框架的入口函数。

- bootstrap 服务启动组件

  提供了各服务和功能的启动入口标识。在 SAMGR 启动时，会调用 boostrap 标识的入口函数，并启动系统服务。

- syspara 系统属性组件

  系统属性组件，根据 HarmonyOS 产品兼容性规范提供获取设备信息的接口，如：产品名、品牌名、厂家名等，同时提供设置/读取系统属性的接口。

- startup 启动组件

  负责提供大型系统（参考内存≥1GB）获取与设置操作系统相关的系统属性。

  大型系统支持的系统属性包括：设备信息如设备类型、产品名称等，系统信息如系统版本、API 版本等默认系统属性。

鸿蒙系统 //base/startup/ 目录下：				APP调用 B
startup + systemrestore	官方文档提到会有这两个组件，但是要大型系统设备（参考内存≥1GB）平台才会适配。			
appspawn_lite	appspawn应用孵化组件。目前仅支持LiteOS-A内核平台。应用孵化器，负责接受应用程序框架的命令孵化应用进程，设置其对应权限，并调用应用程序框架的入口。			启动APP
	services/			
		include/*.h	头文件	
		src/main.c + *.c	main() 调用 HOS_SystemInit()	
		src/appspawn_service.c	SYSEX_SERVICE_INIT(AppSpawnInit)	
		test/	测试用例,liteos_a内核平台的debug版本才会编译	
bootstrap_lite	bootstrap服务启动组件。目前支持LiteOS-M/LiteOS-A内核平台。提供了各服务和功能的启动入口标识。在SAMGR启动时，会调用boostrap标识的入口函数，并启动系统服务。			
	services/			
		core_main.h	定义了SYS_INIT、MODULE_INIT等宏	
		system_init.c	**OHOS_SystemInit()** 调用SYS_INIT、MODULE_INIT启动服务，并启动 **SAMGR_Bootstrap**	
		bootstrap_service.c+.h	SYS_SERVICE_INIT(Init)	
syspara_lite	syspara系统属性组件。支持LiteOS-M/LiteOS-A内核平台。系统属性各字段由OEM厂商负责定义，当前方案仅提供框架及默认值。具体值需产品方按需进行调整。			
	interface/	kits/*.h	给鸿蒙上层应用调用的接口，通过 SetXxx()/GetXxx()来设置和获取属性	B的声明
	frameworks/			
		parameter/*	liteos_m内核平台，编译成静态库，liteos_a内核平台，编译成动态库。	B的实现，调用C
		token/*	liteos_m内核平台，编译成静态库，liteos_a内核平台，编译成动态库。	B的实现，调用C
		unittest/	测试用例,liteos_a内核平台的debug版本才会编译	
	hals/	*.h	HAL层接口的头文件	C的声明
	simulator/	[空]	模拟器相关？？	
init_lite/	init启动引导组件。支持LiteOS-A内核平台。配置文件init.cfg仅支持json格式，烧写到单板之后变成只读模式，修改时必须重新打包和烧写rootfs镜像。			
	services/			
		include/*.h	头文件	
		src/main.c + *.c	实现文件 main(): read configuration file[**/etc/init.cfg**] and do jobs to start all services	读取/分析/运行 /etc/init.cfg
		test/	测试用例,liteos_a内核平台的debug版本才会编译	
//vendor/hisilicon/hispark_taurus/	HiSpark_taurus(Hi3516DV300)参考开发板，编译框架适配、解决方案参考代码和脚本。init_lite 会用到。【见上一级的README_zh.md文档】			
	BUILD.gn	定义生成 meta-targets"hispark_taurus" 的依赖关系:init_configs		
	config.json	产品全量配置表:子系统、组件列表等等		
	init_configs/ BUILD.gn + **init_liteos_a_3516dv300.cfg**	编译生成//out/hispark_taurus/ipcamera_hispark_taurus/**config/init.cfg**，生成rootfs时打包在**/etc/init.cfg**,由用户态根进程调用，据此配置启动鸿蒙系统框架层。BUILD.gn定义 target:init_configs 的生成规则		init.cfg 源文件，copy到 /etc/init.cfg
	config/	板级设备树的描述文件hcs，见官方文档驱动子系统。		
	hals/	安全子系统权限相关+系统属性组件的相关接口在 HAL层的实现		C的实现
Hi3861_Wifiiot/base/startup/	该目录下相比鸿蒙完整系统，裁减掉了 appspawn_lite、init_lite 两个组件			
	frameworks/	这三个合在一起就基本上等同于上面的 syspara_lite 组件了		
	hals/			
	interfaces/			
	services/	同上面的 bootstrap_lite/services/		

两相比较就可以看到 Hi3861 工程相对于完整系统裁剪掉了 appspawn_lite 和 init_lite 两个组件(先灰化掉了)，因为启动方式/流程上有比较大的差别，裁掉 init_lite 其实很容易理解，但为什么裁掉 appspawn_lite 我还没仔细研究。

这里就只分析 Hi3861 的 bootstrap_lite，至于 syspara_lite 比较简单，看官方文档照着上表右边的调用顺序就可以获取属性信息了。appspawn_lite 和 init_lite 两个组件，待我把相关细节搞清楚了，再完善到《鸿蒙系统的启动流程》的更新版本中，或者单独写一个理解总结出来。

下面的文字，其实也算是《鸿蒙系统的启动流程 v3.0》Part 2 的“4.第四阶段：鸿蒙系统框架层的启动”的完整分析版本。

官方 readme 对 bootstrap 服务启动组件的描述就两句话，该怎么理解：

“**提供了各服务和功能的启动入口标识。**” 就是指在//base/startup/services/bootstrap_lite/source/core_main.h 头文件中定义的宏：SYS_INIT(name)和 MODULE_INIT(name)。

【在这里又要强烈推荐去看:

连志安老师的《分析 helloworld 程序是如何被调用，SYS_RUN 做什么事情》

唐佐林老师的《SYS_RUN()和 MODULE_INIT()之间的那些事》
这两篇文章了。】

“**在 SAMGR 启动时，会调用 boostrap 标识的入口函数，并启动系统服务。**”就是指在 system_init.c 文件中的 HOS_SystemInit()函数调用 SAMGR_Bootstrap(); 去启动系统服务了。什么是“**boostrap 标识的入口函数**”，我们在下面会解释。

```
void HOS_SystemInit(void)
{
    MODULE_INIT(bsp);
    MODULE_INIT(device);
    MODULE_INIT(core);
    SYS_INIT(service);        //#A
    SYS_INIT(feature);
    MODULE_INIT(run);     //#B
    SAMGR_Bootstrap();     //#C
}
```

灰掉部分目前我还未涉足，先跳过，但要是理解了下面的内容，灰掉部分也就基本上理解了。

5.1 #A 分析 SYS_INIT(service)

打开//base/startup/services/bootstrap_lite/source/core_main.h 文件，工程编译是使用 gcc 编译器的，所以__GNUC__是有定义的。

把 **service** 代进去展开一下：

```
#define SYS_INIT(service)       \
    do {                        \
        SYS_CALL(service, 0); \
    } while (0)

#define SYS_CALL(service, 0)                                    \
    do {                                                        \
        InitCall *initcall = (InitCall *)(SYS_BEGIN(service, 0));    \
        InitCall *initend = (InitCall *)(SYS_END(service, 0));       \
        for (; initcall < initend; initcall++) {                     \
            (*initcall)();                                           \
        }                                                           \
    } while (0)
```

SYS_BEGIN 和 SYS_END 就不展开了，重点在 for 循环，可能还是不太好理解，我再把它翻译成大白话：

//for 循环就是从 initcall 地址开始，到 initend 地址(不含)结束，

//依次调用*initcall 内的地址所指向的函数，执行函数内的指令。

//initcall 是一个指针，其内容 *initcall 是符号__zinitcall_sys_service_start 的地址，即 &__zinitcall_sys_service_start，

//initend 是一个指针，其内容 *initend 是符号__zinitcall_sys_service_end 的地址，即 &__zinitcall_sys_service_end

Hi3516/Hi3518 平台工程，打开 build\lite\platform\.....\link.ld

Hi3861 工程则是 vendor\hisi\hi3861\hi3861\build\link\link.ld.S

可以看到上面的 start/end 符号，但估计你打开文件，看到里面的东西，心里还是会有很大的问号。

那就直接去看编译后输出的 map 文件：out\wifiiot\Hi3861_wifiiot_app.map，文本编辑器打开该文件，搜索一下：

```
35831                0x00000000004b00ec                __zinitcall_sys_service_start = .
35832  *(.zinitcall.sys.service0.init)
35833  *(.zinitcall.sys.service1.init)
35834  *(.zinitcall.sys.service2.init)
35835  .zinitcall.sys.service2.init
35836                0x00000000004b00ec        0x4 ohos/libs/libbootstrap.a(bootstrap_service.o)
35837  .zinitcall.sys.service2.init
35838                0x00000000004b00f0        0x4 ohos/libs/libbroadcast.a(broadcast_service.o)
35839  .zinitcall.sys.service2.init
35840                0x00000000004b00f4        0x4 ohos/libs/libhiview_lite.a(hiview_service.o)
35841  *(.zinitcall.sys.service3.init)
35842  *(.zinitcall.sys.service4.init)
35843                0x00000000004b00f8                __zinitcall_sys_service_end = .
```

这里有三个 service 要 init，从抓回来的 log 看，确实如此：

```
[system_init] SYS_INIT(service)============================
[bootstrap_service] SYS_SERVICE_INIT(Init).
[samgr_lite] SAMGR_GetInstance g_samgrImpl.mutex[NULL]->call Init()
[samgr_lite] Init.
[samgr_lite] RegisterService(name: Bootstrap)
[broadcast_service] SYS_SERVICE_INIT(Init).
[samgr_lite] RegisterService(name: Broadcast)
[hiview_service] SYS_SERVICE_INIT(Init).
[samgr_lite] RegisterService(name: hiview)
[hiview_service] Init.InitHiviewComponent.
[system_init] SYS_INIT(feature)============================
[pub_sub_feature] Init. SYS_FEATURE_INIT(Init)
[system_init] MODULE_INIT(run)============================
[helloworld] SYS_RUN(helloworld) Begin
[helloworld] Hello World. creating a HmosTask
[helloworld] SYS_RUN(helloworld) End
[LedExample] SYS_RUN(LedExampleEntry)
[system_init] SAMGR_Bootstrap()============================
```

这下应该够清楚了吧？

SYS_INIT(**service**) 是调用端的宏，对应的，定义端也有一个对应的宏 SYS_SERVICE_INIT(xxx)。

工程代码全局搜索一下“SYS_SERVICE_INIT”，把没什么用的 sample 和 .h 中的先去掉，就得到四个：

```
---- SYS_SERVICE_INIT Matches (10 in 6 files) ----
bootstrap_service.c (base\startup\services\bootstrap_lite\source) line 43 :
broadcast_service.c (foundation\distributedschedule\services\samgr_lite\com
hiview_service.c (base\hiviewdfx\services\hiview_lite) line 54 : SYS_SERVIC
samgr_server.c (foundation\distributedschedule\services\samgr_lite\samgr_se
```

前三个就是上面的三个 service。

第四个“SYS_SERVICE_INIT(InitializeRegistry);”在 foundation\distributedschedule\services\samgr_lite\samgr_server\source\samgr_server.c 文件中，

看它的 BUILD.gn 文件“shared_library("server")”，再到上级目录查看 BUILD.gn，

```
    if (ohos_kernel_type == "liteos_a" || ohos_kernel_type == "linux"){
        features += [
            "samgr_server:server",
            "samgr_client:client",
        ]
    }
```

这是 LiteOS_A 或 Linux 内核的平台才会有的，所以 Hi3861 平台的 log 上看不到这个 server 的 log。

上面的三个 service 使用的宏 **SYS_SERVICE_INIT**(Init)，我们也一步一步展开，

```
#define SYS_SERVICE_INIT(func)    LAYER_INITCALL_DEF(func, sys_service, "sys.service")
#define LAYER_INITCALL_DEF(func, layer, clayer) \
    LAYER_INITCALL(func, layer, clayer, 2)              //默认优先级 2
#define LAYER_INITCALL(func, layer, clayer, priority)                \
    static const InitCall USED_ATTR __zinitcall_##layer##_##func \
        __attribute__((section(".zinitcall." clayer #priority ".init"))) = func
```

__attribute__((section(“section_name”))) 其作用是将函数或数据放入指定名为"section_name"对应的段中。
最后分别得到：
//.zinitcall.sys.service2.init = Init //bootstrap_service 的 Init
//.zinitcall.sys.service2.init = Init //broadcast_service 的 Init
//.zinitcall.sys.service2.init = Init //hiview_service 的 Init

编译器根据 attribute+section 关键字，就把三个 Init 函数全都编译链接到了 .zinitcall.sys.service2.init 对应的段中，结果就是上面 Hi3861_wifiiot_app.map 文件截图的样子。

5.2 #B 分析 MODULE_INIT(run)

对于 MODULE_INIT(Xxx) 和对应的 SYS_RUN(Xxx)的分析，和上面没什么差别，只是编译链接的时候，将函数放到不同的段中去而已。
我在应用层放了两个 APP，分别是 helloworld 和 led_example，log 中也看到对应的 log 了。

```
35853           0x00000000004b00fc                __zinitcall_run_start = .
35854  *(.zinitcall.run0.init)
35855  *(.zinitcall.run1.init)
35856  *(.zinitcall.run2.init)
35857  .zinitcall.run2.init
35858           0x00000000004b00fc        0x4 ohos/libs/libhelloworld.a(helloworld.o)
35859  .zinitcall.run2.init
35860           0x00000000004b0100        0x4 ohos/libs/libled_example.a(led_example.o)
35861  *(.zinitcall.run3.init)
35862  *(.zinitcall.run4.init)
35863           0x00000000004b0104                __zinitcall_run_end = .
```


5.3 #C 分析 SAMGR_Bootstrap()

SAMGER：system ability manager。系统服务框架子系统，这是一个非常核心的子系统了，我会另开一章来分析。不过这里还是先简单了解一下它在这一步，做了些什么事情。

先看官方文档：

由于平台资源有限，且硬件平台多样，因此需要屏蔽不同硬件架构和平台资源的不同、以及运行形态的不同，提供统一化的系统服务开发框架[system ability (SA) framework]。根据 RISC-V、Cortex-M、Cortex-A 不同硬件平台，分为两种硬件平台，以下简称 M 核、A 核。

- **M 核：处理器架构为 Cortex-M 或同等处理能力的硬件平台，系统内存一般低于 512KB，无文件系统或者仅提供一个可有限使用的轻量级文件系统，遵循 CMSIS 接口规范。**
- A 核：处理器架构为 Cortex-A 或同等处理能力的硬件平台，内存资源大于 512KB，文件系统完善，可存储大量数据，遵循 POSIX 接口规范。

系统服务框架基于面向服务的架构，提供了服务开发、服务的子功能开发、对外接口的开发、以及多服务共进程、进程间服务调用等开发能力。其中：

- **M 核：包含服务开发、服务的子功能开发、对外接口的开发以及多服务共进程的开发框架。**
- A 核：在 M 核能力基础之上，包含了进程间服务调用、进程间服务调用权限控制、进程间服务接口的开发等能力。

约束

- 系统服务开发框架统一使用 C 开发。
- 同进程内服务间调用统一使用 IUnknown 接口对外象，消息接口统一由 IUnknown 接口传递给本服务。
- 服务名和功能名必需使用常量字符串且长度小于 16 个字节。
- **M 核：系统依赖上 bootstrap 服务，在系统启动函数中调用 OHOS_SystemInit()函数。**
- A 核：系统依赖 samgr 库，在 main 函数中调用 SAMGR_Bootstrap()函数。

再看一个完整一点的 log：

```
[system_init] SYS_INIT(service)============================
[bootstrap_service] SYS_SERVICE_INIT(Init).
[samgr_lite] SAMGR_GetInstance g_samgrImpl.mutex[NULL]->call Init()
[samgr_lite] Init.
[samgr_lite] RegisterService(name: Bootstrap)
[broadcast_service] SYS_SERVICE_INIT(Init).
[samgr_lite] RegisterService(name: Broadcast)
[hiview_service] SYS_SERVICE_INIT(Init).
[samgr_lite] RegisterService(name: hiview)
[hiview_service] Init.InitHiviewComponent.
[system_init] SYS_INIT(feature)============================
[pub_sub_feature] Init. SYS_FEATURE_INIT(Init)
[system_init] MODULE_INIT(run)============================
[helloworld] SYS_RUN(helloworld) Begin
[helloworld] Hello World. creating a HmosTask
[helloworld] SYS_RUN(helloworld) End
[LedExample] SYS_RUN(LedExampleEntry)
[system_init] SAMGR_Bootstrap()============================
[samgr_lite] SAMGR_Bootstrap. Begin:    size=3
        InitializeAllServices: size=3
        Add service:Bootstrap    to TaskPool:0xfa448...
        Add service:Broadcast    to TaskPool:0xfaab8...
        Add service:hiview       to TaskPool:0xfac78...
[task_manager] SAMGR_StartTaskPool:
        CreateTask[Bootstrap, 2048, 25]-OK!
[task_manager] SAMGR_StartTaskPool:
        CreateTask[Broadcast, 2048, 32]-OK!
[task_manager] SAMGR_StartTaskPool:
        CreateTask[hiview, 2048, 24]-OK!
[samgr_lite] SAMGR_Bootstrap. End.
[system_init] HOS_SystemInit end. %%%%%%%%%%%%%
[app_main] app_main End!
########################################################
[samgr_lite] HandleInitRequest. to Init service:id[1][Broadcast]
[broadcast_service] Initialize.

[samgr_lite] HandleInitRequest. to Init service:id[0][Bootstrap]
[bootstrap_service] Initialize.
[samgr_lite] HandleInitRequest. to Init service:id[2][hiview]
[hiview_service] Initialize.
[samgr_lite] InitCompleted: manager->status[1]
[samgr_lite] InitCompleted[1->2]: SendBootRequest[BOOT_SYS_COMPLETED]: Initialized all core system services!
[bootstrap_service] MessageHandle(Bootstrap, request->msgId=0)
[samgr_lite] SAMGR_Bootstrap. Begin:    size=3
[samgr_lite] InitCompleted: manager->status[3]
[samgr_lite] InitCompleted[3->4]: SendBootRequest[BOOT_APP_COMPLETED]: Initialized all system and application services!
[samgr_lite] SAMGR_Bootstrap. End.
[bootstrap_service] MessageHandle(Bootstrap, request->msgId=1)
[samgr_lite] SAMGR_Bootstrap. Begin:    size=3
[samgr_lite] InitCompleted: manager->status[5]
[samgr_lite] SAMGR_Bootstrap. End.
```

看 log，在 **SYS_INIT(service)这一步**，bootstrap_service 首先 init，

打开 base\startup\services\bootstrap_lite\source\bootstrap_service.c 查看它的 Init 函数，

```
static void Init(void)
{
    static Bootstrap bootstrap;
    bootstrap.GetName          = GetName;
    bootstrap.Initialize       = Initialize;
    bootstrap.MessageHandle = MessageHandle;
    bootstrap.GetTaskConfig    = GetTaskConfig;
    bootstrap.flag = FALSE;
    printf("[bootstrap_service] SYS_SERVICE_INIT(Init).\n");

    SAMGR_GetInstance()->RegisterService((Service *)&bootstrap);
}
```

它会先去 get 一个 SAMGR 的 instance 实例，向这个实例注册 bootstrap 服务。

进入 SAMGR_GetInstance()，在 foundation\distributedschedule\services\samgr_lite\samgr\source\samgr_lite.c 文件中：

```
/* ************************************************************************************************
 * Samgr Lite location structure and local variable
 * ************************************************************************************************ */
static SamgrLiteImpl g_samgrImpl;

#define TO_NEXT_STATUS(status) (BootStatus)((uint8)(status) | 0x1)

SamgrLite *SAMGR_GetInstance(void)
{
    if (g_samgrImpl.mutex == NULL) {
        printf("[samgr_lite] SAMGR_GetInstance g_samgrImpl.mutex[NULL]->call Init()\n");
        Init();
    }
    return &(GetImplement()->vtbl);
}

static SamgrLiteImpl *GetImplement(void)
{
    return &g_samgrImpl;
}

static void Init(void)
{
    printf("[samgr_lite] Init.\n");

    WDT_Start(WDG_SAMGR_INIT_TIME);
    g_samgrImpl.vtbl.RegisterService = RegisterService;
    g_samgrImpl.vtbl.UnregisterService = UnregisterService;
    g_samgrImpl.vtbl.RegisterFeature = RegisterFeature;
    g_samgrImpl.vtbl.UnregisterFeature = UnregisterFeature;
    g_samgrImpl.vtbl.RegisterFeatureApi = RegisterFeatureApi;
    g_samgrImpl.vtbl.UnregisterFeatureApi = UnregisterFeatureApi;
    g_samgrImpl.vtbl.RegisterDefaultFeatureApi = RegisterDefaultFeatureApi;
    g_samgrImpl.vtbl.UnregisterDefaultFeatureApi = UnregisterDefaultFeatureApi;
    g_samgrImpl.vtbl.GetDefaultFeatureApi = GetDefaultFeatureApi;
    g_samgrImpl.vtbl.GetFeatureApi = GetFeatureApi;
    g_samgrImpl.status = BOOT_SYS;
    g_samgrImpl.services = VECTOR_Make((VECTOR_Key)GetServiceName, (VECTOR_Compare)strcmp);
    g_samgrImpl.mutex = MUTEX_InitValue();
    (void)memset_s(g_samgrImpl.sharedPool, sizeof(TaskPool *) * MAX_POOL_NUM, 0,
                   sizeof(TaskPool *) * MAX_POOL_NUM);
    WDT_Reset(WDG_SVC_REG_TIME);
} « end Init »
```

bootstrap_service 是第一个调用 SAMGR_GetInstance() 的服务，这时候全局变量 g_samgrImpl 还没有初始化，所以就要先 init，然后就可以返回 instance 给 Bootstrap 注册服务用了，后面的 broadcast_service、hiview_service 在 init 时，直接就可以拿到 instance 去注册了。

全局变量 g_samgrImpl 记录了向它注册的所有服务的信息，包括了一组四个函数：GetName/Initialize/MessageHandle/GetTaskConfig，这就是上面提到的“**boostrap 标识的入口函数**”。

bootstrap_service、broadcast_service、hiview_service 在 SYS_INIT(service)这一步只能做很简单的注册服务的事情，否则会导致后面的 INIT 受阻。

在 SAMGR_Bootstrap(); 这一步时，SAMGR 才会真正根据注册在 g_samgrImpl 的信息，逐一为已注册的服务创建和分配资源，InitializeAllServices，AddTaskPool，SAMGR_StartTaskPool，SAMGR_SendSharedDirectRequest 等待系统调度，然后在 HandleInitRequest 中才真正调用各自 serveice 注册的 Initialize 接口去完成服务的启动，为系统提供服务。

X.总结：

总的来说，Hi3861_WiFiIot 开发板+工程项目，还是非常适合新手入门学习鸿蒙系统的设备开发的，从简单的东西入手，可以逐步渐进，把系统架构图中的：上下层次关系、模块组件关系等各种流程都理一遍，不至于一步踏进完整鸿蒙系统的汪洋大海中，举足难进。

下一步的学习，还是先以这个工程为主，结合完整鸿蒙的代码，其他还没有涉足的模块/组件都去了解一下，把板子玩熟，把设备开发的整体通路打通，形成自己的理解体系，多做总结进行分享，为鸿蒙生态贡献微薄之力。

以上，也算是我对前一阶段自己学习的所得的一点总结吧。

写到这里，我想喊一句口号，类似“迈出第一小步，梦想是星辰大海”之类的，突然想起 hb set 的产品类别名称：wifiiot_hispark_pegasus，说的不就是这个意思吗，从 spark 到 pegasus，从星星之火到星辰大海。

【本文还未结束，未来会继续添加对工程的新的理解。】