AliOS Things 用户态多应用

版本	修改记录	日期
v0.1	完成初版，在 stm32f429zi-nucleo 上体验用户态多应用。	2019.9.23
v0.2	增加在 developerkit 上体验用户态多应用;	2019.9.23
v0.3	增加添加新的应用一节	2019.9.24
v0.4	添加用户态可以使用的 API 一节	2019.9.24

1 目录

1. 快速体验

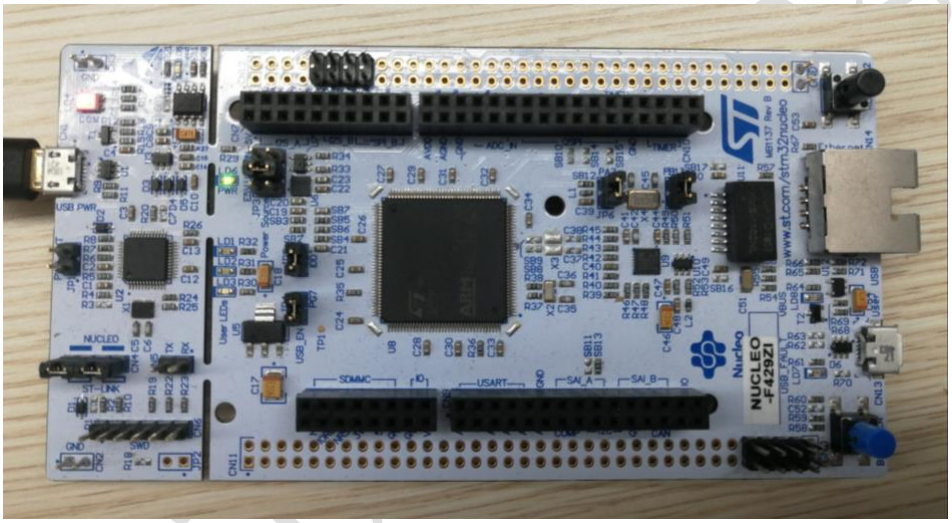
1.1 说明

本章指导用户在 stm32f429zi-nucleo 开发板和 developerkit 开发板上体验 AliOS Things 3.0.0 推出的新特性-用户态多应用。顾名思义，用户态多应用说明应用运行在用户态，相对而言，kernel 运行在内核态（保护态），并且支持多个应用同时运行。

实验所用的代码可以在 AliOS Things 官方 github 下载，地址：

https://github.com/alibaba/AliOS-Things

开发板需要用户自己准备，大概长这个样子：



stm32f429zi-nucleo

developerkit

1.2 安装编译工具

如果已经安装了 AliOS Things 编译环境，可以跳过本节。

参考 https://github.com/alibaba/AliOS-Things/wiki/Quick-Start 安装 Linux 开发下编译所需的工具和库文件。

1.2.1 安装 stm32 的烧录工具 stlink

如果已经安装了 stlink 工具，请跳过本节。

- 先安装编译 stlink 所需的 cmake 和 Libusb 库，如果已安装，请跳过

```
$ sudo apt-get install libusb-1.0-0-dev cmake
```

从 github 上拉取 stlink

```
$ git clone https://github.com/texane/stlink
```

- 编译

```
$ make
```

- 安装

```
$ cd build/Release && sudo make install
```

1.3 拉取 AliOS Things 3.0.0

使用 git 命令从 AliOS Things 官网拉去最新发布的 AliOS Things 3.0.0

```
$ git clone https://github.com/alibaba/AliOS-Things -b rel_3.0.0
```

1.4 编译、烧录（基于 stm32f429zi-nucleo）

这个 demo 需要编译一个内核镜像，两个应用 app 镜像，分别是 uapp1 和 uapp2。对 flash 和 ram 的空间划分如下表

image	flash	size	ram	size
kernel	0x8000000	512k	0x20000000	128k
uapp1	0x8080000	128k	0x20020000	64k
uapp2	0x80a0000	128k	0x10000000	32k

kernel 和两个 app 的地址在对应的链接文件中有描述，链接时按照上表分配的地址进行链接。在 kernel 的链接文件中定义了 app 镜像起始地址符号，以便 kernel 能够从对应的地址上加载 app。

1.4.1 编译内核

- 生成配置文件

```
$ aos make uapp1@stm32f429zi-mk -c config
```

- 编译

```
$ aos make MBINS=kernel
```

编译完成后，将在 out 目录下生成 uapp1@stm32f429zi-mkkernel 的文件夹，内核镜像在 out/uapp1@stm32f429zi-mkkernel/binary 里面。链接 kernel 镜像时使用的链接文件为 stm32f429zi-mk/STM32F429ZITx_FLASH_kernel.ld。

解释一下，“MBINS=kernel”告诉编译系统，编译 kernel 镜像，那么凡是要编译的组件的 aos.mk 文件中“$(NAME)_MBINS_TYPE”等于“kernel”的组件都会被编译进 kernel 镜像。

1.4.2 编译 uapp1

- 生成配置文件

```
$ aos make uapp1@stm32f429zi-mk -c config
```

- 编译 uapp1 镜像

```
$ aos make MBINS=app MBINS_APP=app1
```

编译完成后，生成的 app 镜像在 out/uapp1@stm32f429zi-mkapp/binary/目录

下。

这里"MBINS=app"告诉编译系统编译 app 镜像，那么凡是要编译的组件的 aos.mk 文件中"$(NAME)_MBINS_TYPE"等于"app"的组件都会被编译进 app 镜像。第二个字段"MBINS_APP=app1"用来指定链接文件的名字。在本例中，可以在 board/stm32f429zi-mk/目录下看到名为"app1.ld"的链接文件。

1.4.3 编译 uapp2

- 生成配置文件

```
$ aos make uapp2@stm32f429zi-mk -c config
```

- 编译 uapp2 镜像

```
$ aos make MBINS=app MBINS_APP=app2
```

与编译 uapp1 同理，在此不再赘述。

1.4.4 烧录镜像

将编译生成好的 kernel 镜像和两个 app 镜像烧录到 flash，其中为 kernel 预留的 flash 地址为[0x8000000 ~ 0x80080000)，app1 为[0x8080000 ~ 0x80a0000)，app2 为[0x80a0000 ~ 0x80c0000)。

烧写镜像的时候，先烧写 app，再烧写 kernel。有可能一次性烧写不成功，多烧写几次。

- 烧录 app1

```
$ st-flash write out/uapp1@stm32f429zi-mkapp/binary/uapp1@stm32f429zi-mk.app.bin 0x08080000
```

- 烧录 app2

```
$ st-flash write out/uapp2@stm32f429zi-mkapp/binary/uapp2@stm32f429zi-mk.app.bin 0x080a0000
```

- 烧录内核

```
$ st-flash write out/uapp1@stm32f429zi-mkkernel/binary/uapp1@stm32f429zi-mk.kernel.bin 0x08000000
```

1.5 编译、烧录（基于 developerkit）

这个 demo 需要编译一个内核镜像，两个应用 app 镜像，分别是 uapp1 和 uapp2。对 flash 和 ram 的空间划分如下表：

image	flash	size	ram	size
kernel	0x8000000	512k	0x20000000	128k
uapp1	0x8080000	128k	0x20020000	64k
uapp2	0x80a0000	128k	0x20030000	64k

kernel 和两个 app 的地址在对应的链接文件中有描述，链接时按照上表分配的地址进行链接。在 kernel 的链接文件中定义了 app 镜像起始地址符号，以便 kernel 能够从对应的地址上加载 app。

1.5.1 编译内核

- 生成配置文件

```
$ aos make uapp1@developerkit-mk -c config
```

- 编译

```
$ aos make MBINS=kernel
```

编译完成后，将在 out 目录下生成 uapp1@developerkit-mkkernel 的文件夹，内核镜像在 out/uapp1@developerkit-mkkernel/binary 里面。链接 kernel 镜像时使用的链接文件为 board/ STM32L496VGTx_FLASH_kernel.ld。

解释一下，“MBINS=kernel”告诉编译系统，编译 kernel 镜像，那么凡是要编译的组件的 aos.mk 文件中“$(NAME)_MBINS_TYPE”等于“kernel”的组件都会被编译进 kernel 镜像。

1.5.2 编译 uapp1

- 生成配置文件

```
$ aos make uapp1@developerkit-mk -c config
```

- 编译 uapp1 镜像

```
$ aos make MBINS=app MBINS_APP=app1
```

编译完成后，生成的 app 镜像在 out/uapp1@developerkit-mkapp/binary/目录下。

这里”MBINS=app”告诉编译系统编译 app 镜像，那么凡是要编译的组件的 aos.mk 文件中“$(NAME)_MBINS_TYPE”等于“app”的组件都会被编译进 app 镜像。第二个字段”MBINS_APP=app1”用来指定链接文件的名字。在本例中，可以在 board/developerkit-mk/目录下看到名为”app1.ld”的链接文件。

1.5.3 编译 uapp2

- 生成配置文件

```
$ aos make uapp2@developerkit-mk -c config
```

- 编译 uapp2 镜像

```
$ aos make MBINS=app MBINS_APP=app2
```

与编译 uapp1 同理，在此不再赘述。链接 app2 使用的链接文件为：board/developerkit-mk/app2.ld

1.5.4 烧录镜像

将编译生成好的 kernel 镜像和两个 app 镜像烧录到 flash，其中为 kernel 预留的 flash 地址为[0x8000000 ~ 0x80080000)，app1 为[0x8080000 ~ 0x80a0000)，app2 为[0x80a0000 ~ 0x80c0000)。

烧写镜像的时候，先烧写 app，再烧写 kernel。有可能一次性烧写不成功，多烧写几次。

- 烧录 app1

```
$ st-flash write out/uapp1@developerkit-mkapp/binary/uapp1@developerkit-mk.app.bin 0x08080000
```

- 烧录 app2

```
$ st-flash write out/uapp2@developerkit-mkapp/binary/uapp2@developerkit-mk.app.bin 0x080a0000
```

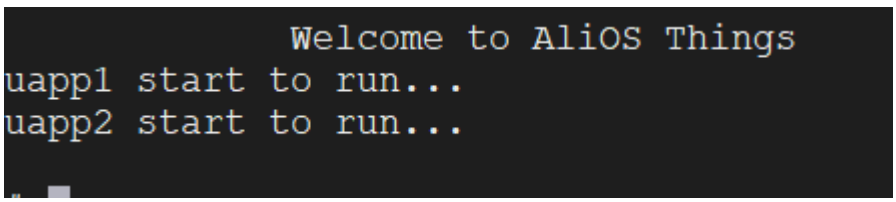
- 烧录内核

```
$ st-flash write out/uapp1@developerkit-mkkernel/binary/uapp1@developerkit-mk.kernel.bin 0x08000000
```

1.6 运行

烧录完成后，连接串口（stlink USB 口自带串口）到串口工具（secureCRT，minicom 等），波特率为 115200，**复位**运行程序。

内核在运行后自动加载已经烧录的两个 app 镜像。



1.7 常用命令

- 查看系统的应用 lsapp

```
$ lsapp
```

```
# lsapp
========== app info ==========
name                        state
uapp1                       run
uapp2                       run
```

- 查看应用进程号 process

```
$ process
```

```
# process
============ process info ============
Name                pid         tasks
uapp2               2           4
uapp1               1           4
```

- 卸载应用 kill + 进程号

```
$ kill 1
```

```
# kill 2
p
r# ocess uapp2  is unloaded, pid 2
```

BUG:进程卸载打印会出现断行

- 加载应用 start +应用名

```
$ start uapp1
```

```
# start uapp1
uapp1 start to run...
```

- 更多命令，使用 help 命令获取帮助

2 增加新的应用

上一节说明了如何基于 stm32f429zi-nucleo 和 developerkit 板子编译烧录应用态应用，以及相关的一些命令行命令，本节带领大家如何添加一个新的应用。

第一，我们都要对内核程序稍微做出一点改动，无非就是告诉内核，从什么地方加载我们新添加的应用。

第二，编写新的应用程序，将其烧录到对应的地址。

2.1 内核支持新的应用

对于 armv7m 架构，内核镜像和应用镜像都被固化在 flash 上，内核加载新的应用时，需要从应用所在的 flash 地址上读取应用程序。要添加一个新的应用，对内核的修改有两处。

2.1.1 修改内核链接文件

我们需要在内核的链接文件中定义一个符号，这个符号的值是应用程序在 flash 上放置的地址。内核程序通过读取该符号，就可以知道从什么地方加载该应用。

以 developerkit 为例，打开 developerkit 的内核链接文件 board/developerkit-mk/STM32L496VGTx_FLASH_kernel.ld。

```
PROVIDE(app1_info_addr = 0x8080000);
PROVIDE(app2_info_addr = 0x80a0000);
PROVIDE(app3_info_addr = 0x80c0000);
```

上图中在内核的链接文件中定义了三个应用程序地址符号，说明这个三个应用程序在 flash 上的地址分别是 0x8080000, 0x80a0000, 0x80c0000。如果需要添加更多的应用，就依葫芦画瓢，添加新的应用程序地址符号。

特别注意：为内核和应用程序划分 flash 区间，1) 这些区间必须要是合法的 flash 区间，不要超出芯片的 flash 范围。2）在 armv7m 架构上，flash 区间划分必须要符合 MPU 对区域划分的要求。简单来说，就是 flash 区间的起始地址必须对其到区间的的大小，或者说区间的起始地址必须是区间大小的整数倍，同时区间的大小是 2 的幂次方，最小不能小于 32 字节。用公式表示出来，(addr % size == 0) && (size == 2^n) && (size >= 32)。

2.1.2 内核应用列表中注册新的应用

打开 platform/arch/arm/armv7m-mk/common/k_bin.c, 在 g_app_info 数组中注册应用。

```
/* if more app should be supported, define them in kernel linker script */
extern char app1_info_addr;
extern char app2_info_addr;
extern char app3_info_addr;

const preamble_t *g_app_info[MAX_APP_BINS] =  {
    (preamble_t*) &app1_info_addr,
    (preamble_t*) &app2_info_addr,
    (preamble_t*) &app3_info_addr,
};
```

其中，宏 MAX_APP_BINS 定义在

platform/arch/arm/armv7m-mk/common/include/k_bin.h 中，默认值为 3，即内核最多支持 3 个应用。如果要支持更多的应用，在 k_bin.h 中修改该宏。

先声明应用符号，然后在将符号的值写入 g_app_info 数组。**注意**，一定要保持声明的符号与在内核链接脚本中定义应用符号一致。

2.2 编写新的应用

编写用户态应用程序，需要遵循一定的规范：

- 创建应用的 preamble 结构体
- 初始化应用堆
- 创建几个服务任务
- 应用进程退出
- 编写应用的链接文件

为 cortex-m 架构图处理器添加新的应用，参考 app/example/uapp1/src/entry_bin.c 和 app/example/uapp1/src/app.c 文件。链接文件参考 developerkit 的 board/app1.ld 文件。

2.2.1 创建应用的 preamble 结构体

必须为每个应用创建一个 preamble 结构体，该结构体描述应用的一些基本信息，并放置在应用 binary 的开始处。内核正是读取该结构体，初始化应用的数据段，为应用创建第一个任务，并初始化应用所使用的 MPU 区域。另外，该结构体包含其他一些预留给用户的配置的字段。

preamble_t 结构体如定义在 platform/arch/arm/armv7m-mk/include/preamble.h 里面，以 app/example/uapp1/src/entry_bin.c 为例说明。

```
#if defined (__CC_ARM) || defined (__GNUC__)
const static preamble_t preamble __attribute__((used, section(".preamble"))) =
#elif defined (__ICCARM__)
#pragma location=".preamble"
const static __root preamble_t preamble =
#endif
{
    PREAMBLE_MAGIC,
    RHINO_CONFIG_APP_NAME,
    (void*)APP_FLASH_START,
    (void*)APP_FLASH_END,
    (void*)APP_RAM_START,
    (void*)APP_RAM_END,
    (void*)APP_DATA_RAM_START,
    (void*)APP_DATA_RAM_END,
    (void*)APP_DATA_FLASH_START,
    (void*)APP_BSS_START,
    (void*)APP_BSS_END,
    &utask,
    app_entry,
    ustack,
    RHINO_CONFIG_APP_ENTRY_TASK_USTACK_SIZE,
    RHINO_CONFIG_APP_ENTRY_TASK_KSTACK_SIZE,
    RHINO_CONFIG_APP_ENTRY_TASK_PRIO,
    RHINO_CONFIG_APP_VERSION,
    {0},
};
```

- PREAMBLE_MAGIC 是应用 preamble 结构体的魔数，内核需要验证该魔数确认 app 的合法性
- RHINO_CONFIG_APP_NAME 是应用的名字字符串。该字符串的长度不要超过 16 个字符
- APP_FLASH_START 是为应用划分的 flash 的起始地址，应用必须下载到该 flash 区间的起始地址处
- APP_FLASH_END 是为应用划分的 flash 结束地址。
- APP_RAM_START 是为应用划分的 RAM 的起始地址
- APP_RAM_END 是为应用划分的 RAM 的结束地址
- APP_DATA_RAM_START 是应用 data 段在 RAM 中的起始地址
- APP_DATA_RAM_END 是应用 data 段在 RAM 中的结束地址
- APP_DATA_FLASH_START 是应用的 data 段在 flash 中的起始地址。通过 APP_DATA_RAM_START、APP_DATA_RAM_END 可以计算出应用的 data 段大小，内核初始化应用时，将应用的 data 段从 flash 复制到 RAM 中
- APP_BSS_START 是应用的 bss 段的起始地址
- APP_BSS_END 是应用 bss 段的结束地址。内核初始化应用时，将 bss 段清零
- utask 指针用于获取应用第一个任务的结构体指针。用户态任务的 task 结构体存在于内核空间，在用户态无法访问该结构体成员，但是可以作为用户态任务的操作句柄使用。
- app_entry 为应用的第一个任务的入口函数，应用开始运行时，就开始执行该函数
- ustack 是应用第一个任务的栈指针。应用第一个任务的栈必须使用静态数组
- RHINO_CONFIG_APP_ENTRY_TASK_USTACK_SIZE 是应用第一个任务的用户态栈大小

- RHINO_CONFIG_APP_ENTRY_TASK_KSTACK_SIZE 是应用第一个任务的内核栈大小
- RHINO_CONFIG_APP_ENTRY_TASK_PRIO 是应用第一个任务的优先级
- RHINO_CONFIG_APP_VERSION 是应用的版本号，占用 4 字节
- 其它，为用户预留的配置字段

2.2.2 初始化应用堆

每个应用程序都有自己的堆，用于应用动态分配内存。应用程序进入第一个任务的入口函数后的第一件事就是初始化堆。调用 umm_init()函数完成初始化用户堆，这个函数的第一个参数是堆的起始地址，第二个参数是堆的大小。

```
/* init app private heap */
mm_start = (void*)APP_HEAP_START;
mm_size  = (size_t)APP_HEAP_END - (size_t)APP_HEAP_START;
if (NULL == umm_init(mm_start, mm_size)) {
    return;
}
```

2.2.3 创建服务任务

在用户态应用中，为了支持某些特定功能，比如删除任务、命令行、timer 定时、callback 等，需要创建服务任务。这些服务都可以通过配置宏来控制。

- res 任务

  该任务服务内核申请、释放应用内存，特别地，删除应用任务时，该任务负责释放任务的用户栈到堆中。如果应用中需要支持删除任务，那么必须创建 res 任务

```
#if (RHINO_CONFIG_URES_TASK > 0)
    res_task_start(RHINO_CONFIG_URES_TASK_KSTACK_SIZE,
                   RHINO_CONFIG_URES_TASK_USTACK_SIZE,
                   RHINO_CONFIG_URES_TASK_PRIO);
#endif
```

- timer 任务

  在应用中创建 timer 任务，应用中就可以使用软件 timer 定时服务。

```
#if (RHINO_CONFIG_UTIMER_TASK > 0)
    timer_task_start(RHINO_CONFIG_UTIMER_TASK_KSTACK_SIZE,
                     RHINO_CONFIG_UTIMER_TASK_USTACK_SIZE,
                     RHINO_CONFIG_UTIMER_TASK_PRIO,
                     RHINO_CONFIG_UTIMER_MSG_NUM);
#endif
```

- cli 任务

  该任务用于执行内核发送到应用的 cli 命令。如果应用中需要注册 cli 命令，那么就必须创建 cli 任务。

```
#if (RHINO_CONFIG_UCLI_TASK > 0)
    ucli_task_start(RHINO_CONFIG_UCLI_TASK_KSTACK_SIZE,
                    RHINO_CONFIG_UCLI_TASK_USTACK_SIZE,
                    RHINO_CONFIG_UCLI_TASK_PRIO);

    cli_register_commands(uapp_builtin_cmds,
                          sizeof(uapp_builtin_cmds)/sizeof(uapp_builtin_cmds[0]));
#endif
```

- callback 任务

  该任务用于执行应用向内核注册 callback 函数。由于权限限制，内核不能直接执行应用注册的 callback，所以在应用中创建一个 callback 任务，在用户态执行应用注册到内核的回调函数。

  callback 功能需要内核支持，rel3.0.0 版本发布的内核没有使能该功能，用户应用程序不能使用该功能。

```
#if (RHINO_CONFIG_UCALLBACK_TASK > 0)
    cb_task_start(RHINO_CONFIG_UCALLBACK_TASK_KSTACK_SIZE,
                  RHINO_CONFIG_UCALLBACK_TASK_USTACK_SIZE,
                  RHINO_CONFIG_UCALLBACK_TASK_PRIO);
#endif
```

2.2.4 应用进程退出

应用的第一个任务完成堆初始化、创建服务任务后，进入 application_start()函数。原则上 application_start()函数不能返回，如果返回，则会调用进程退出函数 krhino_uprocess_exit()，内核会卸载掉该应用。

```
application_start(0, NULL);

/* Note: return from application_start(), the process will exit */

krhino_uprocess_exit();
```

在应用的其他地方，如果调用 krhino_uprocess_exit()函数，也会导致应用进程退出，内核卸载掉应用。

2.2.5 编写应用的链接文件

应用被编译链接成为一个独立的镜像，需要链接文件。对于 cortex-m 处理器，参考 developerkit 板子的应用链接文件，位于 board/developerkit/目录下的 app1.ld 或者 app2.ld。

链接文件需要为应用分配 flash 区间和 ram 区间，区间的起始地址和大小必须满足 MPU 区域的要求。简单来说，就是 flash 区间的起始地址必须对其到区间的的大小，或者说区间的起始地址必须是区间大小的整数倍，同时区间的大小是 2 的幂次方，最小不能小于 32 字节。用公式表示出来，(addr % size == 0) && (size == 2^n) && (size >= 32)。

在链接文件中，需要指定将应用的 preamble 放置在镜像的开始处，如下图所示，将 preamble 段放置在 flash 的起始地址处。

```
/* Define output sections */
SECTIONS
{
    . = ORIGIN(FLASH);

    .appinfo : {
        . = ALIGN(8);
        KEEP(*(.preamble))
    } > FLASH

    /* The program code and other data goes into FLASH */
    .text : {
        . = ALIGN(8);
        *(.text)           /* .text sections (code) */
        *(.text*)          /* .text* sections (code) */
```

链接文件还必须定义一下符号:

```
PROVIDE(_app_flash_start = ORIGIN(FLASH));
PROVIDE(_app_flash_end = ORIGIN(FLASH)+LENGTH(FLASH));

PROVIDE(_app_ram_start = ORIGIN(RAM));
PROVIDE(_app_ram_end = ORIGIN(RAM)+LENGTH(RAM));

PROVIDE(_app_data_flash_start = _sidata);
PROVIDE(_app_data_ram_start = _sdata);
PROVIDE(_app_data_ram_end = _edata);

PROVIDE(_app_bss_start = _sbss);
PROVIDE(_app_bss_end = _ebss);

PROVIDE(_app_heap_start = _ebss);
PROVIDE(_app_heap_end = ORIGIN(RAM) + LENGTH(RAM));
```

- _app_flash_start 应用 flash 区间起始地址
- _app_flash_end 应用 flash 区间结束地址
- _app_ram_start 应用 ram 区间起始地址
- _app_ram_end 应用 ram 区间结束地址
- _app_data_flash_start 应用 data 段在 flash 中的起始地址
- _app_data_ram_start 应用 data 段在 ram 中的起始地址
- _app_data_ram_end 应用 data 段在 ram 中的结束地址
- _app_bss_start 应用 bss 段的起始地址
- _app_bss_end 应用 bss 段的结束地址
- _app_heap_start 应用 heap 起始地址
- _app_heap_end 应用 heap 的结束地址

前 9 个符号在 app 的 preamble 中会用到，最后两个用于在初始化应用堆时指定堆起始地址和堆大小。

3 用户态 API

本章介绍 AliOS Things 为用户态应用提供的 API 接口，其中部分是内核提供给应用的系统调用 API，另一部分是为方便应用编程，基于系统调用包装的 API 或者用户态库中提供的 API，我们称作其他 API。

3.1 系统调用 API

系统调用提供了丰富的系统调用 API，这些 API 和内核中同等功能 API 保持一致，使用相同的内核头文件。

注意：内核头文件中的函数，并不是所有的都可以在用户态应用中使用，只有下面提供的系统调用 API 才可以在应用中使用。

3.1.1 任务

头文件 k_api.h

函数名	功能简介
krhino_cur_task_get	获取当前任务 task 结构体指正。 **注意**：不能在用户态应用中访问 task 结构体成员
krhino_task_sleep	让任务睡眠 ticks
krhino_utask_create	创建用户态任任务 **注意**：使用该函数创建用户态任务，如果用户栈是静态数组，那么不能使用 krhino_utask_del()删除该任务，推荐使用 krhino_utask_dyn_create()函数创建用户态任务。
krhino_utask_del	删除用户态任务 注意：要支持用户态任务删除，需要在应用中创建 res task，参考2.2.3 节

3.1.2 进程

头文件 utask.h

函数名	功能简介
rhino_uprocess_exit	进程退出
krhino_uprocess_res_get	获取进程相关的服务句柄，应用服务任务中会用到

3.1.3 Time

头文件 k_api.h

函数名	功能简介

krhino_sys_time_get	获取系统时间
krhino_sys_tick_get	获取系统 tick 数
krhino_ms_to_ticks	将 ms 转换为 tick 数
krhino_ticks_to_ms	将 tick 转换为 ms

3.1.4 Mutex

头文件 k_api.h

函数名	功能简介
krhino_mutex_dyn_create	创建 mutex
krhino_mutex_dyn_del	删除 mutex
krhino_mutex_lock	获取 mutex
krhino_mutex_unlock	释放 mutex

3.1.5 Semaphore

在用户态应用中要使用 Mutex，内核中需要使能 Mutex。

头文件 k_api.h

函数名	功能简介
krhino_sem_dyn_create	创建 semaphore
krhino_sem_dyn_del	删除 semaphore
krhino_sem_take	获取 semaphore
krhino_sem_give	释放 semaphore

3.1.6 queue

在用户态应用中要使用 queue，内核中需要使能 queue。

头文件 k_api.h

函数名	功能简介
krhino_queue_dyn_create	创建 queue
krhino_queue_dyn_del	删除 queue
krhino_queue_back_send	发送消息给等第一个等待者
krhino_queue_all_send	发送消息给所有等待者
krhino_queue_recv	接收消息
krhino_queue_flush	flush queue

3.1.7 buf queue

在用户态应用中要使用 buf queue，内核中需要使能 buf queue。

头文件 k_api.h

函数名	功能简介

krhino_buf_queue_dyn_create	创建 buf queue
krhino_fix_buf_queue_dyn_create	创建定长消息的 buf queue
krhino_buf_queue_dyn_de	删除 buf queue
krhino_buf_queue_send	发送消息
krhino_buf_queue_recv	接收消息
krhino_buf_queue_flush	flush buf queue

3.1.8 event

在用户态应用中要使用 event，内核中需要使能 event。

头文件 k_api.h

函数名	功能简介
krhino_event_dyn_create	创建 event
krhino_event_dyn_del	删除 event
krhino_event_get	获取 event
krhino_event_se	设置 event

3.1.9 IPC msg

在用户态应用中要使用 IPC msg，内核中需要使能宏 RHINO_CONFIG_UIPC。

头文件 ipc.h

函数名	功能简介
krhino_msg_get	创建 IPC msg
krhino_msg_del	删除 IPC msg
krhino_msg_snd	IPC 发送消息
krhino_msg_recv	IPC 接收消息

3.1.10 HAL UART

在用户态应用中要使用 HAL uart，内核中需要支持 HAL uart API，并且使能宏 RHINO_CONFIG_HAL_UART_SYSCALL。

头文件 aos/hal/uart.h

函数名	功能简介
hal_uart_init	初始化 uart
hal_uart_send	Uart 发送
hal_uart_recv_II	Uart 接收
hal_uart_finalize	结束 uart

3.1.11 HAL ADC

在用户态应用中要使用 HAL ADC，内核中需支持 HAL ADC API，并且使能宏 RHINO_CONFIG_HAL_ADC_SYSCALL。

头文件 aos/hal/adc.h

函数名	功能简介
hal_adc_init	初始化 ADC
hal_adc_value_get	获取 ADC 的值
hal_adc_finalize	结束 ADC

3.1.12 HAL FLASH

在用户态应用中要使用 HAL FLASH，内核中需要支持 HAL FLASH API，并且使能宏 RHINO_CONFIG_HAL_FLASH_SYSCALL。

头文件 aos/hal/flash.h

函数名	功能简介
hal_flash_info_get	获取 flash 分区信息
hal_flash_erase	擦除 flash
hal_flash_write	写入 flash 数据
hal_flash_erase_write	先擦除 flash，然后写入数据
hal_flash_read	从 flash 读取数据
hal_flash_enable_secure	使能 flash 安全
hal_flash_dis_secure	禁止 flash 安全

3.1.13 HAL GPIO

在用户态应用中要使用 HAL GPIO，内核中需要支持 HAL GPIO API，并且使能宏 RHINO_CONFIG_HAL_GPIO_SYSCALL。

头文件 aos/hal/gpio.h

函数名	功能简介
hal_gpio_init	初始化 GPIO
hal_gpio_output_high	GPIO 引脚输出高电平
hal_gpio_output_low	GPIO 引脚输出低电平
hal_gpio_output_toggle	Toggle GPIO 引脚电平
hal_gpio_input_get	读取 PGIO 引脚电平
hal_gpio_enable_irq	使能 GPIO 引脚中断
hal_gpio_disable_irq	禁止 GPIO 引脚中断
hal_gpio_clear_irq	清除 GPIO 引脚中断
hal_gpio_finialize	结束 GPIO

3.1.14 HAL I2C

在用户态应用中要使用 HAL I2C，内核中需要支持 HAL I2C API，并且使能宏 RHINO_CONFIG_HAL_I2C_SYSCALL。

头文件 aos/hal/i2c.h

函数名	功能简介
hal_i2c_init	初始化 I2C
hal_i2c_master_send	I2C 在主模式发送数据
hal_i2c_master_recv	I2C 在主模式接收数据
hal_i2c_slave_send	I2C 在从模式发送数据
hal_i2c_slave_recv	I2C 在从模式接收数据
hal_i2c_mem_write	I2C 写入数据
hal_i2c_mem_read	I2C 读取数据
hal_i2c_finalize	结束 I2C

3.1.15 HAL NAND

在用户态应用中要使用 HAL NAND，内核中需要支持 HAL NAND API，并且使能宏 RHINO_CONFIG_HAL_NAND_SYSCALL。

头文件 aos/hal/nand.h

函数名	功能简介
hal_nand_init	初始化 NAND flash
hal_nand_read_page	从 NAND flash 读取一页数据
hal_nand_write_page	写入 NAND flash 一页数据
hal_nand_read_spare	读取 NAND spare 区域数据
hal_nand_write_spare	写入数据到 NAND spare 区域
hal_nand_erase_block	擦除 NAND flash 块
hal_nand_finalize	结束 NAND flash

3.1.16 HAL NOR

在用户态应用中要使用 HAL NOR，内核中需要支持 HAL NOR API，并且使能宏 RHINO_CONFIG_HAL_NOR_SYSCALL。

头文件 aos/hal/nor.h

函数名	功能简介
hal_nor_init	初始化 NOR flash
hal_nor_read	从 NOR flash 读取数据
hal_nor_write	写入数据到 NOR flash
hal_nor_erase_block	擦除 NOR flash 块
hal_nor_erase_chip	擦除 NOR flash
hal_nor_finalize	结束 NOR flash

3.1.17 HAL DAC

在用户态应用中要使用 HAL DAC，内核中需要支持 HAL DAC API，并且使能宏 RHINO_CONFIG_HAL_DAC_SYSCALL。

头文件 aos/hal/dac.h

函数名	功能简介
hal_dac_init	初始化 DAC
hal_dac_start	开始 DAC 转化
hal_dac_stop	结束 DAC 转化
hal_dac_set_value	设置 DAC 的值
hal_dac_get_value	获取 DAC 的值

3.1.18 HAL RTC

在用户态应用中要使用 HAL RTC，内核中需要支持 HAL RTC API，并且使能宏 RHINO_CONFIG_HAL_RTC_SYSCALL。

头文件 aos/hal/rtc.h

函数名	功能简介
hal_rtc_init	初始化 RTC
hal_rtc_get_time	获取 RTC 时间
hal_rtc_set_time	设置 RTC 时间
hal_rtc_finalize	结束 RTC

3.1.19 HAL SD

在用户态应用中要使用 HAL SD，内核中需要支持 HAL SD API，并且使能宏 RHINO_CONFIG_HAL_SD_SYSCALL。

头文件 aos/hal/sd.h

函数名	功能简介
hal_sd_init	初始化 SD
hal_sd_blks_read	从 SD 读取若干 blk 数据
hal_sd_blks_write	写入若干 blk 数据到 SD
hal_sd_erase	擦除 SD
hal_sd_stat_get	获取 SD 状态信息
hal_sd_info_get	获取 SD 信息
hal_sd_finalize	结束 SD

3.1.20 HAL SPI

在用户态应用中要使用 HAL SPI，内核中需要支持 HAL SPI API，并且使能宏 RHINO_CONFIG_HAL_SPI_SYSCALL。

头文件 aos/hal/spi.h

函数名	功能简介
hal_spi_init	初始化 SPI
hal_spi_send	SPI 发送数据
hal_spi_recv	SPI 接收数据

hal_spi_send_recv	SPI 收发数据
hal_spi_finalize	结束 SPI

3.1.21 UCLI

在用户态应用中要使用 CLI 命令，内核中需要支持 CLI，并且使能宏 RHINO_CONFIG_UCLI_SYSCALL。

头文件 cli/cli_api.h

函数名	功能简介
cli_register_command	注册 CLI 命令
cli_unregister_command	撤销 CLI 命令
cli_register_commands	注册若干 CLI 命令
cli_unregister_commands	撤销若干 CLI 命令

3.1.22 LWIP

在用户态要使用 LWIP 网络 API，需要内核支持 LWIP，并使能宏 RHINO_CONFIG_LWIP_SYSCALL。

注意: AliOS Things rel_3.0.0 版本提供的库不支持用户态应用中使用 lwip，即使内核支持 lwip 和使能宏 RHINO_CONFIG_LWIP_SYSCALL。需要完整功能的用户态 LWIP,请联系 AliOS Things 官方。

头文件，lwip 对应的头文件

函数名	功能简介
lwip_accept	查看 lwip 官方对该 API 的描述
lwip_bind	查看 lwip 官方对该 API 的描述
lwip_shutdown	查看 lwip 官方对该 API 的描述
lwip_getpeername	查看 lwip 官方对该 API 的描述
lwip_getsockname	查看 lwip 官方对该 API 的描述
lwip_getsockopt	查看 lwip 官方对该 API 的描述
lwip_setsockopt	查看 lwip 官方对该 API 的描述
lwip_close	查看 lwip 官方对该 API 的描述
lwip_connect	查看 lwip 官方对该 API 的描述
lwip_listen	查看 lwip 官方对该 API 的描述
lwip_recv	查看 lwip 官方对该 API 的描述
lwip_read	查看 lwip 官方对该 API 的描述
lwip_recvfrom	查看 lwip 官方对该 API 的描述
lwip_send	查看 lwip 官方对该 API 的描述
lwip_sendmsg	查看 lwip 官方对该 API 的描述
lwip_sendto	查看 lwip 官方对该 API 的描述
lwip_socket	查看 lwip 官方对该 API 的描述
lwip_write	查看 lwip 官方对该 API 的描述

lwip_writev	查看 lwip 官方对该 API 的描述
lwip_select	查看 lwip 官方对该 API 的描述
lwip_ioctl	查看 lwip 官方对该 API 的描述
lwip_fcntl	查看 lwip 官方对该 API 的描述
lwip_eventfd	查看 lwip 官方对该 API 的描述
lwip_try_wakeup	查看 lwip 官方对该 API 的描述
lwip_gethostbyname	查看 lwip 官方对该 API 的描述
lwip_gethostbyname_r	查看 lwip 官方对该 API 的描述
lwip_freeaddrinfo	查看 lwip 官方对该 API 的描述
lwip_getaddrinfo	查看 lwip 官方对该 API 的描述

3.2 其他 API

3.2.1 用户态 timer

头文件 utimer.h

用户态 timer API 满足应用中定时需求，API 用法和 kernel 中 timer API 一致。使用 timer 前，需要创建 timer 任务。

```
#if (RHINO_CONFIG_UTIMER_TASK > 0)
    timer_task_start(RHINO_CONFIG_UTIMER_TASK_KSTACK_SIZE,
                     RHINO_CONFIG_UTIMER_TASK_USTACK_SIZE,
                     RHINO_CONFIG_UTIMER_TASK_PRIO,
                     RHINO_CONFIG_UTIMER_MSG_NUM);
#endif
```

函数名	功能简介
krhino_timer_create	创建软件 timer
krhino_timer_del	删除软件 timer
krhino_timer_dyn_create	动态创建软件 timer
krhino_timer_dyn_del	动态删除软件 timer
krhino_timer_start	启动 timer 开始计时
krhino_timer_stop	停止 timer
krhino_timer_change	改变 timer 属性
krhino_timer_arg_change_auto	自动改变 timer 属性，相比 krhino_timer_change，无需手动停止 timer 和启动 timer
krhino_timer_arg_change	改变 timer 回调函数的参数

3.2.2 用户态 aos API

用户态 aos API 并不是系统调用函数，而是基于系统调用函数派生出来函数 aos API 保持一致，目的是方便用户态应用编程。

头文件 aos/kernel.h

函数名	功能简介
aos_task_new	动态创建用户态任务。默认任务的内核栈大小为 4KB，使用 aos_task_exit()删除任务。前提需要创建用户态 res task
aos_task_new_ext	与 aos_task_new()类似，相比 aos_task_new()支持更多的配置参数。
aos_task_exit	删除用户态任务。注意任务只能自己删除自己，不能删除别的任务
aos_mutex_new	创建用户态 mutex
aos_mutex_free	删除 mutex
aos_mutex_lock	获取 mutex
aos_mutex_unlock	释放 mutex
aos_mutex_is_valid	判断 mutex 是否合法
aos_sem_new	创建用户态 semaphore。注意，需要内核支持 sem
aos_sem_free	删除 semaphore
aos_sem_wait	获取 semaphore
aos_sem_signal	释放 semaphore
aos_sem_is_valid	判断 semaphore 是否合法
aos_queue_new	创建用户态 queue。注意，需要内核支持 queue
aos_queue_free	删除 queue
aos_queue_send	向 queue 发送消息
aos_queue_recv	从 queue 接收消息
aos_queue_is_valid	判断 queue 是否合法
aos_timer_new	创建用户态 timer
aos_timer_new_ext	创建用户态 timer。相比 aos_timer_new()函数支持更多的配置参数
aos_timer_free	删除 timer
aos_timer_start	启动 timer
aos_timer_stop	停止 timer
aos_timer_change	改变 timer 属性
aos_zalloc	分配内从，并初始化为 0。参考 zalloc 函数
aos_malloc	分配内存。参考 malloc 函数
aos_calloc	分配多组内存。参考 calloc 函数
aos_realloc	重新分配内存。参考 realloc 函数
aos_free	释放内存。参考 free 函数
aos_now	获取系统启动后到现在的时间,单位 ns
aos_now_ms	获取系统启动后到现在的时间，单位 ms
aos_now_time_str	获取系统启动后到现在的时间，单位 ms，并转换成字符串
aos_msleep	任务睡眠 ms