**alkaid 基础架构介绍**

**{Document Name + Version}**

© 2017 MStar Semiconductor, Inc. All rights reserved.

MStar Semiconductor makes no representations or warranties including, for example but not limited to, warranties of merchantability, fitness for a particular purpose, infringement of any intellectual property right or the accuracy or completeness of this document, and reserves the right to make changes without further notice to any products herein to improve reliability, function or design. No responsibility is assumed by MStar Semiconductor arising out of the application or user of any product or circuit described herein; neither does it convey any license under its patent rights, nor the rights of others.

MStar is a trademark of MStar Semiconductor, Inc. Other trademarks or names herein are only for identification purposes only and owned by their respective owners.

REVISION HISTORY

| Revision No. | Description | Date |
| --- | --- | --- |
| {Version} | * {Initial release} | {MM/DD/YYYY} |

TABLE OF CONTENTS

[1. {Heading 1} 1](#_Toc325127903)

[1.1. {Heading 2} 1](#_Toc325127904)

[1.1.1. {Heading 3} 1](#_Toc325127905)

[2. {Heading 1} 2](#_Toc325127906)

[2.1. {Heading 2} 2](#_Toc325127907)

[2.1.1. {Heading 3} 2](#_Toc325127908)

LIST OF TABLES

[Table 1: {Table Title} 3](#_Toc325127909)

LIST OF FIGURES

[Figure 1: {Figure Name} 3](#_Toc325127910)

# 目录结构

Project configs #系统配置文件，这里面每个CHIP一个子目录

Board #开发板相关配置，不分CHIP的放在CHIP目录外面，分CHIP的放在CHIP子目录里面

Kbuild # linux kernel编译环境，每个kernel版本一个子目录

$(KERNEL\_VERSION) # kernel版本

Arch/arm/include

Include/linux

Scripts

Configs #make menuconfig产生的配置文件

Image #产生image的工厂

Build #产生image的辅助工具

Rootfs #rootfs模板

Busybox #busybox的代码/config/bin

Etc #linux配置文件

Configs #产生image的配置文件

Release #所有预编译给客户的二进制文件和对应的编译配置文件

Include #对外头文件

$(CHIP) #特定chip的编译文件

customer\_options #特性配置文件，配置的是编译宏

customer\_tailor #裁剪配置文件，配置的是模块开关

Sdk mhal #与chip相关的代码

$(MHAL) #代码文件夹，至少包含一个文件夹名为$(CHIP)，其余文件夹可选

Impl #与chip无关的代码

$(MODULE) #每个模块一个子目录

Interface #user/kernel和kernel/kernel的界面文件

Include #头文件，头文件模式为mi\_xxx.h和mi\_xxx\_datatype.h为public

Include/internal/kernel #kernel mode的内部公共头文件

Include/internal/user #user mode的内部公共头文件

Src/$(MODULE) #每个模块一个实现文件夹

Verify #验证/测试/演示代码

Module #测试模块API专用文件夹，配置文件超简单，适合懒人

Verify #测试多个模块协作专用文件夹，可以接入busybox前端

Misc #存放与alkaid关系不是很大或者虽然有关系但是后面可能会删除的代码

# 模块框架

## Linux module的基本框架

如果使用linux的基本框架，会面临开源的问题，这是因为创建设备的函数是GPL的。

但是规避GPL校验也不难，所以alkaid使用了函数指针来搭建整个框架，简而言之，所有模块依赖common：

Module3

Module2

Module1

Mi\_common

### Dev

Dev创建和注册以及操作这部分，所有模块都是一样的，因此全部封装起来，防止出现差异

Dev又分为user和kernel两个子部分，user会负责打开设备，传输数据，kernel则负责响应user

#### User mode

User mode主要操作只有一个，ioctl，open和close作为辅助动作也一并被封装起来了。

Open被封装的原因有：

* 有很多人力，希望一次跑起来的客户，可以选择不init直接上
* 希望优化开机时间的客户可能会选择lazy初始化
* Open需要的文件名需要与kernel同步

Close被封装的原因有：

* 将异常退出和正常退出同质化，也就是说，强制正常退出和异常退出走一样的流程
* Close没有参数

Ioctl本来就被封装，其流程也是固定的，但是这里会有一个版本管控的问题，user mode需要将参数size传递给内核，这样内核才能知道两边size是不是匹配，所以这部分也是被强制封装起来了

封装的位置是interface/include/internal/user/mi\_syscall.h

封装ioctl的宏是MI\_SYSCALL，表示其逻辑与syscall类似

#### Kernel mode

Kernel mode主要操作是创建节点，调用ioctl回调，因此封装的是固定流程，开放出去的是回调

封装位置是interface/include/internal/kernel/mi\_syscall.h

Kernel mode还有一些固定流程是通过代码实现的，位於interface/src/common/device.c

在device.c里面的主要对象是MI\_DEVICE\_Object\_t，每个模块使用MI\_DEVICE\_DEFINE定义一个MI\_DEVICE\_Object\_t，而后在insmod的时候自动注册到common/device.c

在kernel mode可以区分进程，因此每一个进程使用module之前都是先调用open，於是，可以借机捕获进程信息，并保存在MI\_COMMON\_Client\_t，这个对象可以用作保存per-process数据。

### Operator

每个dev都会关联一组文件操作，open/close/ioctl分别对应客户的“延迟到第一次调用API连接内核”，“函数退出”，“调用API”，其余的文件操作都不响应

延迟到第一次调用API连接内核是指无需指定哪个API为第一次调用的API，任何API调用的时候会自动触发连接到内核的动作，防止出现低级错误

函数退出是指函数正常/异常退出的时候，内核会负责清理资源，关闭文件

调用API指的是封装user/kernel和kernel/kernel的边界，主要是ioctl的封装，因为内核里面栈的大小有限，而user mode栈的大小则相对大很多，对於user/kernel这样的边界来说，kernel里面如果要接user的栈对象，就需要分配内存以确保安全

### Ioctl

因为在user/kernel和kernel/kernel两种边界都执行ioctl，因此ioctl提供了上下文信息，上下文可以区分当前是user/kernel还是kernel/kernel。

### 1.1.3 对象关系

下图描述了整个过程，user mode和kernel mode透过宏MI\_SYSCALL，经过不同的路径获得一个ioctl\_table，最终调用到底层注册上来的impl functions，这个ioctl\_table保存在device object里面。

每个process可以在open的时候产生一个Client，从Client里面拿到device object，内核里面调用MI\_SYSCALL则直接获得device object，因此两种路径对底层的区别就在於：kernel mode的client指针为NULL

Process A

Process B

MI\_SYSCALL

MI\_SYSCALL

MI\_SYSCALL

MI\_DEVICE\_Ioctl

Device.ioctl\_table

Impl\_0

Impl\_1

Impl\_2

MI\_DEVICE\_Object\_t

MI\_COMMON\_Client\_t

MI\_COMMON\_Client\_t

User mode

kernel mode

# 调试系统

## debug\_level

调试系统主要封装在这个文件里面：

interface/src/common/linux.c

interface/include/internal/kernel/mi\_print.h

interface/include/internal/kernel/mi\_device.h

### 2.1.1. debug\_level

在MODULE\_DEFILE里面会为每一个模块定义一个debug\_level，同时提供一个MI\_DEVICE\_GetDebugLevel实现。

### 2.1.3 module\_param

Module\_param是内核模块自带的参数系统，这个系统的特点是可以在insmod的时候带参数下去，这样就不会出现初始化的时候无法控制debug\_level的问题

## debugfile

debugfile封装在下面这两个文件里面：

interface/include/internel/kernel/mi\_common\_internal.h

interface/src/common/linux.c

User mode

/sys/kernel/{module}/xxx

Echo x1 x2 x3

kernel mode

Sys\_write(“x1 x2 x3”)

DebugRawFile.raw\_write(“x1 x2 x3”)

DebugFile.write(“x1”,“x2”,”x3”, 3)

User mode

Cat /sys/kernel/{module}/xxx

kernel mode

Sys\_read(“x1 x2 x3”)

DebugFile.read(“x1 x2 x3”)

DebugRawFile.raw\_read(“x1 x2 x3”)

debugfile借助sysfs将调试文件导出到文件系统，位置位於/sys/kernel/{module}/xxx

MI\_COMMON\_AddDebugFile这个API提供分词服务的文件，执行这个文件相当於在kernel里面开了applet，比如说echo x1 x2 x3 > /sys/kernel/{module}/xxx，那xxx的read函数就读到”x1” “x2” “x3”三个参数

MI\_COMMON\_AddDebugRawFile这个API提供原始数据的文件，比如上面的例子就是”x1 x2 x3”这样的参数，RawFile还需要自己制定文件size

## 2.3 perf

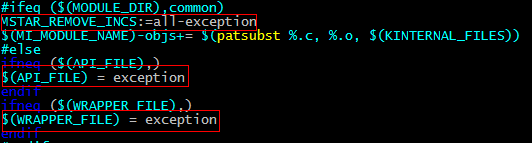
Perf是内核附带的调试工具，这个工具已经整合进alkaid，如果想知道如何使用参考这份文档：

### 

# 跨平台

## Makefile的限制

在kernel的makefile里面添加了机制，使得可以单独对某一些文件夹进行包含路径过滤，如果某文件不希望被过滤，则需要加上例外标记：



## Os wrapper

Alkaid挑选了一组OS WRAPPER，在POSIX/Linux Kernel/RTOS的表现完全一样的API，实现并且提供给上层。

User mode的位於：interface/src/common/posix\_wrapper.c

Kernel mode的位於：impl/common/mi\_os\_linux.c

# config\_tool

Config tool集成了所有与配置有关的工具，每个工具通过命令名区分，当前集成了mmap和systeminfo两个工具。

最终在insmod mi\_common.ko的时候，会触发impl/common/mi\_syscfg\_mmap.c里面的MI\_SYSCFG\_SetupMmapLoader和impl/common/mi\_syscfg\_config.c里面的MI\_SYSCFG\_SetupIniLoader。

* 1. 逻辑框架

Kernel

call\_usermodehelper

Config\_tool cmd

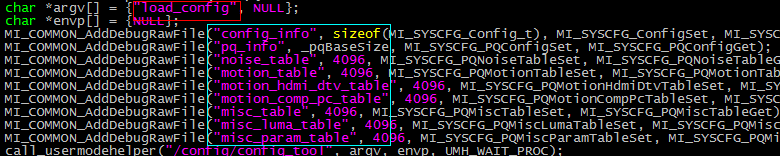
Read information

API FILE

Write config value

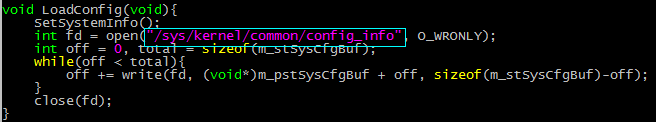
* 1. config的实现

config会比较多一些，主要是PQ因为使用二维数组的缘故多了几个特殊的入口：



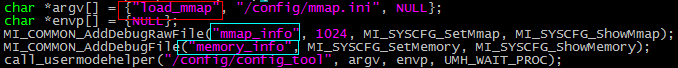
对应的config\_tool实现如下图所示：





* 1. mmap的实现

mmap的cmd为下图红色所示，API\_FILE为下图兰色所示：



从图中可以看到，confi\_tool会通过MI\_SYSCFG\_SetMmap将parser到的数据写进来，通过MI\_SYSCFG\_ShowMmap把保存好的数据回写给config\_tool，config\_tool的实现如下图所示：

