**KalinoteOS 系统支持文档**

一个普通的人在这里创造了一个有趣的系统

第三版 2021年8月16日最后更新

By:Kalinote

目录

[1. 说明 1](#_Toc80307538)

[1.1 简介 1](#_Toc80307539)

[1.2 名词解释 1](#_Toc80307540)

[KalinoteOS 1](#_Toc80307541)

[kal 1](#_Toc80307542)

[2. 系统源代码目录结构及对应功能 1](#_Toc80307543)

[/： 1](#_Toc80307544)

[文件夹： 1](#_Toc80307545)

[文件： 2](#_Toc80307546)

[/CoreFiles: 2](#_Toc80307547)

[文件夹： 2](#_Toc80307548)

[文件： 2](#_Toc80307549)

[/CoreFiles/kalinote: 2](#_Toc80307550)

[文件夹： 2](#_Toc80307551)

[文件： 2](#_Toc80307552)

[/z\_tools: 3](#_Toc80307553)

[文件夹： 3](#_Toc80307554)

[文件： 3](#_Toc80307555)

[3. 系统启动时的工作 3](#_Toc80307556)

[引导扇区 3](#_Toc80307557)

[进入系统循环前 3](#_Toc80307558)

[4. 系统kernel工作方式 4](#_Toc80307559)

[4.1 系统内存管理方式 4](#_Toc80307560)

[内存管理程序(memman)数据结构 5](#_Toc80307561)

[系统内存分配原理 5](#_Toc80307562)

[系统内存释放原理 6](#_Toc80307563)

[启动时内存初始化过程 6](#_Toc80307564)

[4.2 系统数据结构 6](#_Toc80307565)

[队列(FIFO)数据缓冲区 6](#_Toc80307566)

[双链表(List) 7](#_Toc80307567)

[4.3 系统进程管理方式 7](#_Toc80307568)

[TASK数据结构 7](#_Toc80307569)

[TSS32数据结构 7](#_Toc80307570)

[TASKLEVEL数据结构 8](#_Toc80307571)

[TASKCTL数据结构 8](#_Toc80307572)

[任务等级(TASKLEVEL) 8](#_Toc80307573)

[4.4 系统图层(sheet)管理方式 9](#_Toc80307574)

[图层(sheet)数据结构 9](#_Toc80307575)

[图层控制(SHTCTL)数据结构 9](#_Toc80307576)

[图层高度及其调整 9](#_Toc80307577)

[图层flags定义 10](#_Toc80307578)

[子图层(subsheet)的创建与子图层管理器(sht->subctl) 10](#_Toc80307579)

[图层与任务(task)的关系 11](#_Toc80307580)

[4.5 系统窗口(window)及图形(graphic)绘制方式 11](#_Toc80307581)

[系统字符绘制方式 11](#_Toc80307582)

[系统字符串的多种绘制方式 11](#_Toc80307583)

[系统方块(box)的绘制方式 12](#_Toc80307584)

[系统窗口的绘制方式 12](#_Toc80307585)

[系统图标(icon)的绘制方式 12](#_Toc80307586)

[系统任务栏(taskbar)的管理方式 12](#_Toc80307587)

[4.6 系统段号记录表及中断向量号管理方式 12](#_Toc80307588)

[系统段号记录表及中断向量号数据结构 12](#_Toc80307589)

[系统段号记录表及中断向量号初始化 13](#_Toc80307590)

[系统中断机制及其向量号分配说明 13](#_Toc80307591)

[4.7 驱动程序 14](#_Toc80307592)

[4.8 命令行(console) 14](#_Toc80307593)

[4.9 文件系统 14](#_Toc80307594)

[4.10 定时器系统 14](#_Toc80307595)

[4.11 系统语言支持 14](#_Toc80307596)

[4.12 系统全局快捷键 14](#_Toc80307597)

[5. 系统内置CMD指令 14](#_Toc80307598)

[5.1 查询类 15](#_Toc80307599)

[mem指令 15](#_Toc80307600)

[dir&ls指令 15](#_Toc80307601)

[5.2 功能类 15](#_Toc80307602)

[cls&clear指令 15](#_Toc80307603)

[type [Filename]指令 15](#_Toc80307604)

[exit 指令 15](#_Toc80307605)

[start指令 15](#_Toc80307606)

[run指令 15](#_Toc80307607)

[langmode指令 16](#_Toc80307608)

[shutdown指令 16](#_Toc80307609)

[sysmode指令 16](#_Toc80307610)

[echo指令 16](#_Toc80307611)

[6. KalinoteOS系统内部API 16](#_Toc80307612)

[6.1 数据及任务处理 16](#_Toc80307613)

[6.2 显示画面处理 16](#_Toc80307614)

[6.3 外部设备驱动 16](#_Toc80307615)

[7. KalinoteOS系统调用 16](#_Toc80307616)

[\_api\_putchar: 17](#_Toc80307617)

[\_api\_putstr0: 17](#_Toc80307618)

[cons\_putstr1: 17](#_Toc80307619)

[api\_end: 17](#_Toc80307620)

[api\_openwin: 18](#_Toc80307621)

[\_api\_putstrwin: 18](#_Toc80307622)

[\_api\_boxfilwin: 18](#_Toc80307623)

[\_api\_initmalloc: 19](#_Toc80307624)

[\_api\_malloc: 19](#_Toc80307625)

[\_api\_free: 19](#_Toc80307626)

[\_api\_point: 19](#_Toc80307627)

[\_api\_refreshwin: 20](#_Toc80307628)

[\_api\_linewin: 20](#_Toc80307629)

[\_api\_closewin: 20](#_Toc80307630)

[\_api\_getkey: 21](#_Toc80307631)

[\_api\_alloctimer: 21](#_Toc80307632)

[\_api\_inittimer: 21](#_Toc80307633)

[\_api\_settimer: 21](#_Toc80307634)

[\_api\_freetimer: 22](#_Toc80307635)

[\_api\_beep: 22](#_Toc80307636)

[\_api\_fopen: 22](#_Toc80307637)

[\_api\_fclose: 22](#_Toc80307638)

[\_api\_fseek: 23](#_Toc80307639)

[\_api\_fsize: 23](#_Toc80307640)

[\_api\_fread: 23](#_Toc80307641)

[\_api\_cmdline: 24](#_Toc80307642)

[\_api\_getlang: 24](#_Toc80307643)

[\_api\_cls: 24](#_Toc80307644)

[\_api\_get\_year; 24](#_Toc80307645)

[\_api\_get\_mon\_hex; 25](#_Toc80307646)

[\_api\_get\_day\_of\_week; 25](#_Toc80307647)

[\_api\_get\_day\_of\_month; 25](#_Toc80307648)

[\_api\_get\_hour\_hex; 25](#_Toc80307649)

[\_api\_get\_min\_hex; 26](#_Toc80307650)

[\_api\_get\_sec\_hex; 26](#_Toc80307651)

[8. 标准函数API 26](#_Toc80307652)

[string.h 26](#_Toc80307653)

[malloc.h(stdlib.h) 26](#_Toc80307654)

[stdio.h 26](#_Toc80307655)

[ctype.h 27](#_Toc80307656)

[stdlib.h 27](#_Toc80307657)

[9. 系统及kal应用软件的编译过程 27](#_Toc80307658)

[10. 参考资料 27](#_Toc80307659)

# 说明

## 1.1 简介

本文档是KalinoteOS的系统支持文档，在本文档中提到的KalinoteOS、系统、本系统、该系统等皆指KalinoteOS，在本文档中提到的程序、系统程序、软件等除非特殊指明，皆指在KalinoteOS上运行的kal或其他支持在KalinoteOS上运行的任何软件。

其实在写这个操作系统的时候，基本上就是想到什么写什么，所以有很多地方的代码考虑并不是很全面，效率可能也不高，但是如果要重构就太麻烦了，所以暂时先这样吧。同样，在写这篇支持文档的时候也是想到什么写什么，所以读起来可能也不那么“通畅”，敬请理解。

## 1.2 名词解释

### KalinoteOS

全称为Kalinote Operating System(卡铱诺特操作系统)，是一个多任务，图形化，支持中文的桌面操作系统。

### kal

全程为KalinoteOS Application，指通过指定编译器编译后能在KalinoteOS上运行的应用程序后缀名，详见《系统及kal应用软件的编译过程》一章。

# 系统源代码目录结构及对应功能

在KalinoteOS系统源代码文件夹(KalinoteOS1.0)中，有如下文件夹和文件(仅对部分重要的文件和文件夹进行描述)：

## /：

### 文件夹：

.git：github的相关数据文件夹(不用管)。

CoreFile：KalinoteOS的相关文件。

doc：系统源代码相关说明文档以及系统制作相关参考资料。

z\_tools：编译系统所用到的程序以及部分源代码，还有用于运行系统的qemu虚拟机、编译kal软件的库和编译器。

### 文件：

LICENSE：KalinoteOS源代码开放许可(MIT许可)。

README.md：系统更新日志。

## /CoreFiles:

### 文件夹：

kalinote：KalinoteOS系统源代码。

apilib：系统外部API库。

stdlib：系统标准API库。

golibc：C语言标准函数库。

resources：系统多媒体资源。

langs：系统中文和日文字体资源。

其他：KalinoteOS系统内置kal软件。

### 文件：

apilib.h：KalinoteOS系统外部API头文件(详见“KalinoteOS系统外部API”一章)。

app\_make.txt：kal应用程序批量编译makefile。

kalinote.img：编译完成的KalinoteOS系统镜像文件。

kalinote.url：KalinoteOS编译规则。

make.bat：启动z\_tools下的make.exe。

Makefile：用于批量编译的makefile文件。

stdlib.h：标准函数API文件(详见”标准函数API”一章)。

## /CoreFiles/kalinote:

### 文件夹：

console：命令台相关源代码。

driver：外部设备驱动相关源代码。

filesys：文件系统相关源代码。

gui：用户图形界面相关源代码。

kernel：系统内核相关源代码。

### 文件：

asmhead.nas：系统进入循环前的硬件配置以及引导CPU读取bootpack的汇编代码。

bootpack.h：系统源代码头文件，所有系统相关的源代码都会引用该头文件，该头文件也包含系统内部大部分API和常量以及结构体(详见” KalinoteOS系统内部API”一章)。

fonts.txt：系统半角字符字体文件。

ipl40.nas：系统读盘代码，用于启动前将KalinoteOS需要的部分读入内存。

kalinote.sys：编译后的KalinoteOS系统文件。

make.bat：启动z\_tools下的make.exe。

Makefile：用于批量编译的makefile文件。

naskfunc.nas：部分无法使用C语言实现的功能的汇编代码。

## /z\_tools:

### 文件夹：

Bz162：二进制文件查看工具。

guigui00：部分用于编译系统的标准函数头文件。

kalinote：KalinoteOS系统支持的标准函数。

qemu：方便调试系统的qemu虚拟机。

src：部分工具源代码。

win32：Windows系统支持的标准函数，用于编译windows平台工具(如bim2kal.exe等)。

### 文件：

这个目录下的文件多为编译系统或kal应用程序的软件，详细使用见”系统及kal应用软件的编译过程”一章。

# 系统启动时的工作

## 引导扇区

img镜像的前512字节(引导扇区)，作用是将系统必要文件读入内存当中，并跳转到asmhead部分，然后将处理器的控制权交给操作系统。该扇区的代码保存在/CoreFiles/kalinote/ipl40.nas中，并在编译后生成ipl40.bin，并写入到镜像第一个扇区。

在ipl40.nas中，定义了img镜像的部分属性，其中，启动区名称为”KALINIPL”，磁盘名称为”KALINOTE-OS”，磁盘格式为FAT12。

如果读入系统出现错误，程序会重复尝试，直到将系统完整读出。如果失败超过5次，则会跳转到错误程序(flag为error)，并在屏幕上打印”KalinoteOS load error!”。

## 进入系统循环前

系统程序主体文件为CoreFiles/kalinote/kernel/bootpack.c，主函数为void KaliMain(void)，该函数由/z\_tools/src/kallib0a(remake)/startup.c中的void KaliStartup(void)调用执行，并在该函数中执行进入系统循环前的代码。

在bootpack.c的KaliMain函数中，在进入系统循环前会进行一些初始化设置，流程如下(其中，带\*的为因发现重大bug或暂未实现等原因暂时取消的步骤)：

1. 初始化sysinfo，sysinfo是一个结构体，用于保存系统基本信息，该结构体位于内存0x10000处(详见《系统占用内存分布》中的注释8)，该结构体保存了系统模式、系统主版本号，系统版本号，计算机IDE硬盘数量，系统剩余内存容量，系统总内存等信息。
2. 初始化GDT和IDT(全局段号表和中断记录表)。
3. 初始化PIC(中断控制器)。
4. 中断记录表和中断控制器初始化完成后，接触CPU的禁止中断(STI)。
5. 初始化FIFO缓冲区(详见“队列(FIFO)数据缓冲区”一节)。
6. 初始化PIT(定时器)。
7. 初始化鼠标、键盘FIFO缓冲区。
8. 设置中断控制器允许PIT、PIC1、PS/2鼠标键盘。
9. 初始化ACPI
10. 保存FIFO缓冲区到内存0x0fec处(详见《系统占用内存分布》中的注释8)。
11. 测试并初始化内存(0x00000000到0xffffffff)。
12. \*初始化硬盘。
13. \*初始化调色板。
14. 初始化图层管理器。
15. 初始化多任务管理器。
16. 启动系统底层任务(task\_a，没有为什么，它就叫task\_a)。
17. 保存shtctl到内存0xfe4处(详见《系统占用内存分布》中的注释8)。
18. 设置语言模式为ASCII英文(对应langmode 0)。
19. 初始化DEBUG cons。
20. 初始化系统图形界面背景(sht\_back)。
21. \*初始化桌面右键菜单。
22. 初始化底部任务栏。
23. 初始化鼠标指针。
24. 刷新画面以显示图形化内容。
25. 载入字库(日文、英文、中文)。
26. 执行startup.kbs脚本。
27. 设置定时器，每秒更新一次时间

# 系统kernel工作方式

## 系统内存管理方式

KalinoteOS使用分段式内存管理方式，使用列表管理方法，原理是将可用内存信息以一个表的方式储存，储存的内容大概是类似”在XX号地址开始有XX字节空间可用”。为了应对不连续空间，系统中使用了一个MEMMAN\_FREES常量(该常量在bootpack.h中的memory.c部分)来建立多个表(系统设定了4090个表，约占32KB空间)用于储存可用内存段信息，每个表都可以保存一段空余内存信息，所以理论最多支持4090个内存碎片段。

例如：

Free[0]:地址0x00400000开始，有0x0A0F0510字节可用

Free[1]:地址0xAC105140开始，有0x0000D100字节可用

…

在内存碎片段占满MEMMAN\_FREES后，系统会先舍弃其他碎片空间，等到有碎片段释放后，重新进行内存检查，然后重新标记空内存段。

比如，某个程序需要分配512KB内存空间，则先遍历可用内存表，然后找到可用空间大于512KB的内存表，并将指向地址分配给程序。然后将可用地址表的指向地址更改为分配后的末尾，可用空间减少512KB。如果可用空间为0，则删除此表，MEMMAN->frees-1。

同样，在释放内存时，增加一条可用内存表，MEMMAN->frees +1。在释放内存时，还有可能会遇到两段内存表相邻(中间无不可用内存)，如：

Free[0]:地址0x00400000开始，有0x00019000字节可用

Free[1]:地址0x00419000开始，有0x07BE7000字节可用

…

以上两段内存可归纳为:

Free[0]:地址0x00400000开始，有0x07C00000字节可用

如果不将如上情况的两个表合并，则会不必要地消耗可用内存表，还有就是有程序在寻找可用内存时，有足够的可用空间却无法正常找到。

### 内存管理程序(memman)数据结构

为了进行内存管理，系统设定了一个全局的内存管理结构体struct MEMMAN，具体结构如下：

struct MEMMAN {

int frees, maxfrees, lostsize, losts;

struct FREEINFO free[MEMMAN\_FREES];

};

在结构体中，int frees表示可用信息数目(见”系统内存管理方式”一节)，最大4090条；int maxfree用于记录最大可用信息数目；int lostsize表示在系统释放失败时(详见”系统内存释放”一节)损失的内存大小；int losts表示系统释放内存失败的次数；struct FREEINFO free[MEMMAN\_FREES]用于储存可用内存信息，struct FREEINFO结构如下：

struct FREEINFO {

unsigned int addr, size;

};

其中，unsigned int addr表示可用内存地址，unsigned int size表示该地址可用的内存大小。

### 系统内存分配原理

使用memman\_alloc进行内存分配，该函数会返回一段可用内存的地址，详细使用方法见”KalinoteOS系统内部API”一章。

系统分配内存的基本原理是：通过遍历可用内存信息表(struct FREEINFO)找到一段可用且足够大的空内存，并将其标记为已占用(删除这个struct FREEINFO)，并在占用内存的末尾新建一个struct FREEINFO，标记该段的剩余空内存。

### 系统内存释放原理

使用memman\_free进行内存释放该函数会返回释放成功或失败的信息，详细使用方法见”KalinoteOS系统内部API”一章。

系统内存释放的基本原理是：系统为了方便管理内存，所以free[]是按照内存地址顺序进行排列的，所以memman\_free会先判断释放的内存在哪个位置，找到存放位置后会判断该地址的前后是否有可合并的空地址(详见”系统内存管理方式”一节)，如果有，就与前后的可用内存表合并，不单独建表，如果没有，就单独建表。在释放完成后更新最大可用内存等信息。

如果释放失败，lost+1，并对lostsize进行统计。

### 启动时内存初始化过程

首先对0x00400000-0xffffffff内存进行测试，以判断系统可用最大内存总量(32位系统理论最大值3972MB)；然后初始化内存管理结构(详见”内存管理程序(memman)数据结构”一节)；然后将0x00001000 - 0x0009efff的内存设置为可用，这一部分是是模式读取软盘的内容，在进入保护模式后，这部分内容被移动到0x00100000以后，这一段内存在启动完成后有其他用途；然后清空除系统占用外的所有保护模式扩展内存(0x00400000以后)；最后将总可用内存信息写入sysinfo结构体。

## 系统数据结构

### 队列(FIFO)数据缓冲区

在KalinoteOS中，会有多处设备(或程序)涉及到数据交换，每个设备进行数据交换的时间和数值都会有所不同。在本系统中有几种数据管理方式，其中一种是队列(FIFO,First in first out)数据管理方式，在fifo结构体中有以下几个变量：

int \*buf，一个地址指针，用来指向缓冲区地址

int p，表示下一次数据的写入地址

int q，表示下一次数据的读入地址

int size，表示缓冲区大小

int free，表示缓冲区空闲字节数

int flags，表示溢出标志

struct TASK \*task 有数据写入时需要唤醒的任务，TASK是任务结构体，详细可以参考后面的系统进程管理方式一节

在系统主进程(task\_a)中，有一个专用的fifo缓冲区，用于处理各项事务(bootpack.c中的struct FIFO32 fifo)，系统中包括鼠标、键盘、时间、命令行相关操作等需要在系统主循环中处理的数据皆使用这个缓冲区。对于这个缓冲区，不同控制会使用不同的控制区间，详细如下：

1 系统时间

2 打开新的命令台

256-511 键盘数据(键盘控制器读入值+256)

512-767 鼠标数据(鼠标控制器读入值+512)

768-1023 命令行：窗口关闭处理

1024-2023 命令行：结束任务

2024-2279 命令行：关闭命令窗口，不结束任务

### 双链表(List)

在/CoreFile/kalinote/kernel/list.c中定义了双链表结构，但是目前在系统中暂时还没有使用到。

## 系统多任务管理方式

KalinoteOS系统进程(任务，在系统中称为task，下同)以结构体方式进行储存和管理，管理系统所有task和每个task的信息储存分别为TASKCTL结构体和TASK结构体；管理每个进程的任务状态段(TSS)的结构体为TSS32结构体；管理任务级(TASK LEVEL)的为TASKLEVEL结构体。下面对这几个结构体及其管理方式进行介绍。

### TASK数据结构

struct TASK结构如下：

struct TASK {

int sel, flags;

int level, priority;

struct FIFO32 fifo;

struct TSS32 tss;

struct SEGMENT\_DESCRIPTOR ldt[2];

struct CONSOLE \*cons;

int ds\_base, cons\_stack;

struct FILEHANDLE \*fhandle;

int \*fat;

char \*cmdline;

unsigned char langmode, langbyte1;

// char \*task\_name;

};

其中，sel用于存储task的GDT编号；flags用于存储任务运行状态，flags为0时表示任务未启用，flags为1时表示任务未运行或休眠状态，flags为2时表示任务正在运行；level表示任务等级，在KalinoteOS中一共有11个任务等级(详见”任务等级(TASKLEVEL)”一节)，分别是从0到9，任务等级数字越小越优先；priority为任务优先级，优先级越高任务越优先；struct FIFO32 fifo;是task的fifo缓冲区(详见”队列(FIFO)数据缓冲区”一节)；struct TSS32 tss;用于保存任务状态段(详见”任务状态段(TSS)”一节)；struct SEGMENT\_DESCRIPTOR ldt[2];用于保存task对应的idt表(详见”系统段号记录表及中断向量号管理方式”一节)；struct CONSOLE \*cons;用于储存task对应的console(详见”命令行(console)”一节)；ds\_base和cons\_stack分别用于储存段基址和栈；struct FILEHANDLE \*fhandle;用于文件处理；fat用于储存解码的FAT表(用于FAT12)；cmdline用于储存输入的命令；langmode和langbytel用于储存语言模式(详见”系统语言支持”一节)；task\_name用于储存任务名称字符串。

### TSS32数据结构

TSS32结构体如下：

struct TSS32 {

int backlink, esp0, ss0, esp1, ss1, esp2, ss2, cr3;

int eip, eflags, eax, ecx, edx, ebx, esp, ebp, esi, edi;

int es, cs, ss, ds, fs, gs;

int ldtr, iomap;

};

该结构体负责储存应用程序在运行时所有寄存器的状态，以此来达到多任务切换。在该结构体的第一行(int backlink, esp0, ss0, esp1, ss1, esp2, ss2, cr3;)为各种段寄存器，第二行(int eip, eflags, eax, ecx, edx, ebx, esp, ebp, esi, edi;)为所有32位的寄存器，第三行(int es, cs, ss, ds, fs, gs; )为所有16位寄存器，最后的ldtr和iomap用来储存段表相关信息。

### TASKLEVEL数据结构

TASKLEVEL结构体如下：

struct TASKLEVEL {

int running;

int now;

struct TASK \*tasks[MAX\_TASKS\_LV];

};

其中，running表示正在运行的任务数量，now表示当前运行的任务，struct TASK \*tasks[MAX\_TASKS\_LV];表示该任务级的所有任务；每个任务级最多可运行250个任务。

### TASKCTL数据结构

TASKCTL结构体如下：

struct TASKCTL {

int now\_lv;

char lv\_change;

struct TASKLEVEL level[MAX\_TASKLEVELS];

struct TASK tasks0[MAX\_TASKS];

};

其中，now\_lv表示当前正在活跃的任务级，lv\_change表示下次切换任务时是否需要改变level，struct TASKLEVEL level[MAX\_TASKLEVELS];用于储存任务级数据结构体，总共10个任务级；struct TASK tasks0[MAX\_TASKS];用于储存所有任务结构体，理论最大支持2500个任务。

### 任务等级(TASKLEVEL)

为了使任务优先级不冲突，系统使用了任务等级的结构，将系统所有任务分为了10个等级(MAX\_TASKLEVELS常量)。任务等级从0到9优先级依次降低，LEVEL0为最高优先级。每个任务等级最多可支持250个任务，目前还没有做任务数量溢出的操作，task\_alloc会在溢出时返回0。

## 系统图层(sheet)管理方式

KalinoteOS图层(在系统程序中称为sheet，下同)以结构体方式管理和储存，管理系统所有图层和图层信息储存分别为SHTCTL和SHEET结构体。首先对两个结构体进行介绍。

### 图层(sheet)数据结构

首先是图层结构体struct SHEET，其结构如下代码：

struct SHEET {

unsigned int \*buf;

int bxsize, bysize, vx0, vy0;

int col\_inv, height, flags;

struct SHTCTL \*ctl;

struct TASK \*task;

};

其中，\*buf为图层缓冲区的指针，这个指针指向的位置为该层的图形数据；bxsize, bysize, vx0, vy0分别表示图层图像大小和图层在画面上的坐标位置；col\_inv是透明色色号，用于表示图层透明程度；height表示图层的高度(详细见”图层高度及其调整”一节)；flags表示图层设定信息(详细见”图层flags定义”一节)；\*ctl是图层控制结构体(详细见”图层控制(SHTCTL)数据结构”一节)；\*task是当前运行的任务(详细见”图层与任务(task)的关系”一节)。

### 图层控制(SHTCTL)数据结构

为了方便管理图层，本系统使用了一个全局的图层管理结构体，该结构体在系统启动时被创建，并且伴随系统整个生命周期。系统在启动时会创建一个名为shtctl的struct SHTCTL变量，并在内存检查(详见”系统内存管理方式”一节)完成后永久储存到内存地址0x00000FE4处(详见”系统占用内存分布”表)。图层控制结构体结构代码如下：

struct SHTCTL {

unsigned int \*vram, \*map;

int xsize, ysize, top;

struct SHEET \*sheets[MAX\_SHEETS];

struct SHEET sheets0[MAX\_SHEETS];

};

在该结构体中，变量\*vram为硬件图像缓冲区(0x00000FF8开始，详见”系统占用内存分布”表)的地址；\*map是内存管理程序(详细见” 内存管理程序(memman)数据结构”一节)给shtctl分配的内存空间，其大小等于图像显示所需内存大小，其中储存的是预处理的需要显示的信息；xsize和ysize是图层大小，和系统显示分辨率相同；top表示顶层图层(鼠标指针)图层高度，也可以理解为正在显示的图层层数；\*sheets[MAX\_SHEETS]是所有图层的图像缓冲区地址，MAX\_SHEETS是一个常量，保存的是最大图层数量，系统设定为256；sheets0[MAX\_SHEETS]存放所有图层的结构体(详见”图层(sheet)数据结构”一节)信息。

### 图层高度及其调整

由于系统在使用的时候，可能会打开许多窗口，所有的这些窗口(包括桌面、任务栏、鼠标指针等)有可能会同时在屏幕上显示，而如果要同时在屏幕上显示就需要指定一个先后顺序，所有在图层的数据结构中有高度的概念，从底层到顶层的高度从0开始，其中0号高度固定为桌面(壁纸)，1号高度固定为任务栏，最顶层高度固定为鼠标指针。如果高度为-1则是不显示，常用于需要临时隐藏的页面。

在调用sheet\_alloc函数(详见”KalinoteOS内部API函数”一章)申请内存时，系统会分配一个图层，但是图层默认为-1(不显示)，所以在申请图层后需要手动指定一个高度(使用sheet\_updown函数指定图层高度)。

在使用sheet\_updown函数调整图层高度时会先对目标数进行判断，如果没有问题，该函数会对相关的图层进行高度调整，以保持高度连续，避免出现空高度的情况，该函数详细工作原理见”KalinoteOS内部API函数”一章。

### 图层flags定义

在图层(SHEET)的结构体中有一个int flags;的变量，该变量指定了该图层的图层属性。详细的flags及其对应属性如下：

flags为SHEET\_NO\_USE常数(值为0)表示该图层没有被使用。

flags为SHEET\_USE常数(值为1)表示该图层(作为窗口)正常使用。

flags为SHEET\_APIWIN常数(值为2)表示该图层(作为外部API窗口)正常使用。

flags为SHEET\_CONS常数(值为3)表示该图层(作为命令行窗口)正常使用。

flags为SHEET\_NO\_TITLE常数(值为4)表示该图层(作为其他无标题栏窗口)正常使用。

flags为SHEET\_BACK常数(值为101)表示该图层(作为背景层)正常使用。

flags为SHEET\_TASKBAR常数(值为102)表示该图层(作为任务栏)正常使用。

flags为SHEET\_MOUSE常数(值为103)表示该图层(作为鼠标指针)正常使用。

flags为SHEET\_MENU常数(值为104)表示该图层(作为菜单栏)正常使用。

在系统处理图层事件(比如鼠标点击或拖动等)时，会先判断是哪种图层，并对相应的图层做出相应的操作(详见”系统对各个种类图层的处理”一节)。

### 子图层(subsheet)的创建与子图层管理器(sht->subctl)

在SHEET结构体中有一个成员为struct SHTCTL \*subctl;这个成员用于保存该图层的子图层的图层管理器(子图层管理器)，子图层管理器管理该图层下的所有子图层。

图层在创建的时候不会先创建图层管理器(一个原因是为了节省内存还有一个原因是要先手动分配图形缓冲区地址sheet\_setbuf，然后才能指定子图层管理器的地址)，如果需要使用子图层管理器需要手动使用shtctl\_init进行初始化和指定。

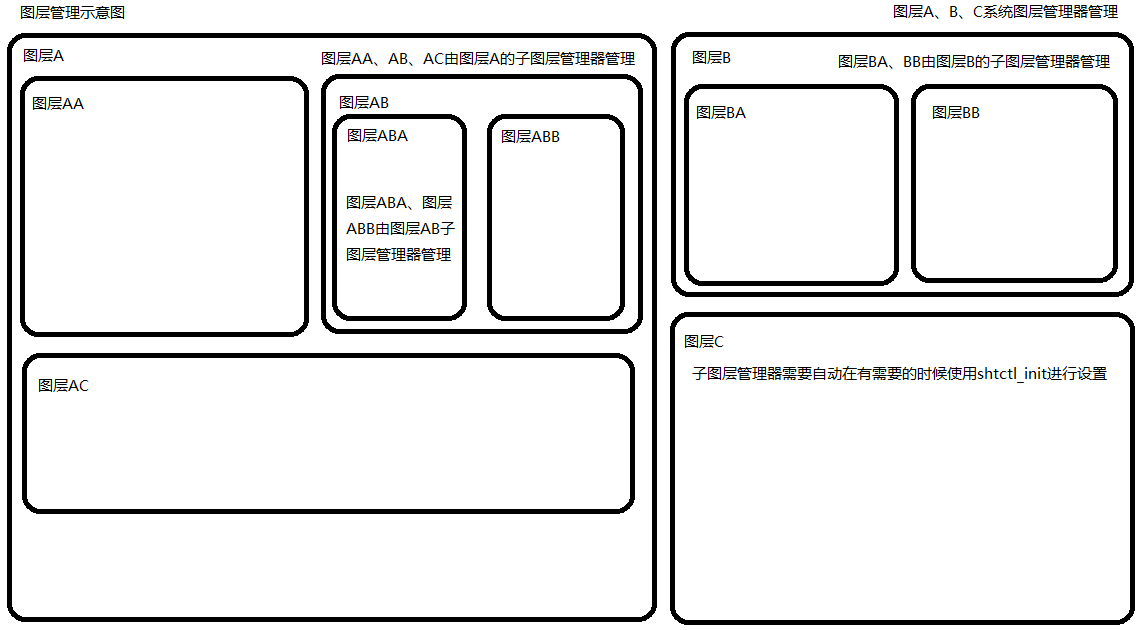
图层系统关系如下图：

图 3.4-1 图层管理示意图

在子图层更新后，必须从内到外依次进行刷新，如上图的图层ABA更新内容后，必须依次刷新图层ABA，刷新图层AB，刷新图层A，这个以后可能会增加一个刷新子图层的函数。

### 图层与任务(task)的关系

在图层初始化以后，可以指定sheet->task来绑定图层和任务，在绑定图层和任务以后，当任务结束时，系统会自动寻找残留图层并关闭。

## 系统窗口(window)及图形(graphic)绘制方式

### 系统字符绘制方式

系统内部有个API为putfont8，可以用来绘制半角字符(字符大小为8x16)，系统在绘制字体时，先读入字符数据，然后逐点判断是否应该绘制数据，如果是，则向指定的图形缓存区地址写入相应的颜色数据。所以绘制一个字符，CPU需要运行128次if判断。

对于全角字符(中文，日文等)使用putfont32进行绘制，中文字库和日文字库的渲染方式不同，中文为上下组合，日文为左右组合，所以这个函数对于中文和日文使用了不用的渲染模式。其余部分基本原理和putfont8一样，绘制一个全角字符，CPU需要运行256次if判断。

### 系统字符串的多种绘制方式

系统内部API有几种字符串绘制方式，但都是基于putfonts8\_asc，这个函数是通过字符ascii码进行绘制。在window.c中有一种方法为putfonts8\_asc\_sht，这个函数是经过putfonts8\_asc进行改进的函数，在绘制字符串之前先涂一层背景色，简化了字符更新的方法。目前还没有用于保留背景仅更新字符的函数，所有字符更新都是先画背景层再写字符。

### 系统方块(box)的绘制方式

系统使用内部API boxfill8来绘制方块，这也是使用的比较多的绘制函数之一。因为内存是一段连续的(没有换行的概念)，所以需要使用程序处理换行，所以在boxfill8中需要传入一个xsize来表示方块x的长度，在内存写到一行的末尾时y++。

### 系统窗口的绘制方式

系统在绘制窗口时，标题栏和窗体是分开渲染的，窗口的相关主题色有一个结构体来储存。在绘制窗口时使用make\_wtitle8绘制标题栏，窗口的相对0坐标是从标题栏的左上角开始，所以要注意如果绘制子图层需要加上标题栏的高度。默认标题栏颜色是粉色，debug窗口的标题栏颜色是红色。窗口的句柄参数为int型的sheet结构体位置

### 系统图标(icon)的绘制方式

使用内部API make\_icon来绘制icon，但是现在暂时还没有自定义icon的办法，目前只有console有icon绘制，这个功能可能将在后面追加更新(也可能不会，谁知道呢)。

### 系统任务栏(taskbar)的管理方式

任务栏的初始化和鼠标指针绘制以及背景层绘制在同时进行。在分配了任务栏图层后，对任务栏进行子图层的初始化，同时初始化任务栏的菜单栏(function菜单栏)并添加菜单栏选项，最后初始化任务栏结构体。在开开启一个窗口时，会在任务栏添加相应的窗口按钮，当一排窗口按钮达到一定数量时系统会自动换排，窗口按钮数量上限为256(MAX\_TSKWINBTN 常量)。目前关闭窗口的任务栏按钮功能还有严重漏洞，但是暂时还没办法定位到，所以建议不要一次打开太多窗口。

## 系统段号记录表及中断向量号管理方式

### 系统段号记录表及中断向量号数据结构

SEGMENT\_DESCRIPTOR的结构体结构如下：

struct SEGMENT\_DESCRIPTOR {

short limit\_low, base\_low;

char base\_mid, access\_right;

char limit\_high, base\_high;

};

在这个结构体中，储存了三个信息：段大小(limit)，段基址(base)，段属性(access)，为了兼容不同CPU，结构体对这些数据进行了拆分处理。

GATE\_DESCRIPTOR的结构体结构数据如下：

struct GATE\_DESCRIPTOR {

short offset\_low, selector;

char dw\_count, access\_right;

short offset\_high;

};

这两个表的大小都是8字节，因为CPU资料规定就是这样。

### 系统段号记录表及中断向量号初始化

系统段号记录表和中断向量号相关功能在/CoreFile/kalinote/kernel/dsctbl.c中实现。系统段号记录表和中断向量号分别为struct SEGMENT\_DESCRIPTOR \*gdt和struct GATE\_DESCRIPTOR \*idt结构体，存在内存的0x26f800-0x27ffff中。其中，0x270000~0x27ffff是GDT，一共保存了8192条记录，共64KB。0x26f800~0x26ffff是IDT，一共保存了255条记录，共2KB。

在gdt和idt进行初始化的时候，首先会将每个表的上限，基址，访问权限都设为0。初始化还会设置两个段，段号为1的段，表示CPU管理内存大小，范围是从0x00000000到0xffffffff，共4GB，也就是操作系统能管理的最大内存；段号为2的段，包括了整个bootpack.kal，一共512KB，地址是0x280000，因为bootpack.kal是以ORG 0为前提翻译成的机器语言(与应用程序为同一编译方式)，所以在该段可以直接执行bootpack.kal；应用程序的段号从3开始。

IDT在初始化的时候还会注册系统用到的所有中断。

在GDT和IDT初始化完成以后，分别使用load\_gdtr();和load\_idtr();将其位置(Address)和大小(Limit)注册到GDTR寄存器和IDTR寄存器中。

其他相关内容可参考源代码注释。

### 系统中断机制及其向量号分配说明

在初始化gdtidt(名为 void init\_gdtidt(void); 的函数)的过程中，会使用 void set\_segmdesc(struct SEGMENT\_DESCRIPTOR \*sd, unsigned int limit, int base, int ar); 对中断进行注册，在CPU触发中断时，调用注册的函数。通常来说，在set\_segmdesc中注册的是由nask语言写的函数，在naskfunc.nas中，然后由naskfunc.nas中的函数调用由C语言编写的中断处理函数。

目前系统注册的中断号如下：

0x00: 除0异常中断，该中断是CPU异常中的一种，在CPU执行错误的除法指令(DIV等)时触发，如0做除数时。

0x07: FPU异常中断，该中断是CPU异常中的一种，FPU(Float Point Unit)是CPU中负责处理浮点数的浮点数处理器，该处理器发生异常计算时CPU会触发0x07号中断。

0x0c: 栈异常中断，该中断是CPU异常中的一种，该中断通常发生在应用程序变量溢出时 (如应用程序数组溢出，或是变量分配栈空间不足等)。

0x0d: 一般保护异常中断，该中断是CPU异常中的一种，该中断通常发生在应用程序非法越权操作时(如非法指令，越权访问内存等)。

0x20: 时钟中断，该中断由PIC发出，由时钟模块定期触发。

0x21: P/S2键盘中断，该中断由PIC发出，由键盘发送信号时触发。

0x27: 没用的中断，不用管。

0x2c: P/S2鼠标中断，该中断由PIC发出，由鼠标发送信号时触发。

0x40: 外部应用程序系统调用中断，该中断由外部应用程序触发，用作外部应用程序的系统调用。

0x41(制作中，尚未实现): 外部驱动程序系统调用中断，该中断由外部驱动程序触发，用作外部驱动程序的系统调用。

## 驱动程序

待补充

## 命令行(console)

console的代码在/CoreFile/kalinote/console中，主要功能在console.c中，每个cmd指令都单独在一个文件中实现。

有一种特殊的console是debug console，这个console只负责输出，不能输入但是可以运行(一般情况下用不上)，这个console可以用于系统和应用程序的调试，这个console使用sysmode 1启动。

更多命令行细节后续补充。

## 文件系统

待补充

## 定时器系统

待补充

## 系统语言支持

系统支持ascii英文，GB2312简体中文和日文三种语言模式，目前日文模式有严重漏洞，所以暂时无法使用(不过平时不怎么用到，所以可能也不会修复)。

更多语言支持细节后续补充。

## 系统全局快捷键

系统指定了几个快捷键，如下：

Shift+F1：强制停止应用程序。

Shift+F2：打开新的命令窗口。

F11：切换窗口。

TAB：切换聚焦窗口。

更多系统快捷键细节后续补充。

# 系统内置CMD指令

这些是系统内部编写的指令，源代码放在/CoreFiles/kalinote/console中。

## 查询类

### mem指令

该指令为memory的缩写，作用是查询系统内存使用情况，具体方法是通过调用memman\_total内部API查询 可用内存状况，详细参考“KalinoteOS系统内部API”一章。

### dir&ls指令

该指令用于查询系统中的文件。dir和ls两个指令实现的效果相同，具体实现原理是循环搜索FILEINFO结构体中的文件名，并判断其状态(普通文件、被删除、系统目录、只读文件等)并依次输出，详细FILEINFO结构体参考“文件系统”一节。

## 功能类

### cls&clear指令

cls为clear screen的缩写，该指令的作用是清空命令行窗口。这两个指令实现方法相同，大概原理是通过黑色填充命令行窗口，然后将光标归位。

### type [Filename]指令

type指令用于输出某个文件的内容。基本实现方法是先调用file\_search来查找文件是否存在，如果存在则使用file\_loadfile读取文件内容，并打印到控制台。具体文件详细参考“KalinoteOS系统内部API”一章；如果没有找到文件，则报错。

### exit 指令

该指令用于关闭命令窗口。实现原理是先调用timer\_cancel取消掉命令窗口的timer(详细参考“KalinoteOS系统内部API”一章以及“定时器系统”一节)，然后使用taskbar\_removewin去掉任务栏按钮，然后释放程序占用内存，最后向系统循环中的FIFO发送窗口关闭指令(详细参考“队列(FIFO)数据缓冲区”一节)。

### start指令

该指令用于在新的命令窗口启动一个应用程序。

### run指令

该指令用于在当前命令窗口执行一个应用程序，并不占用当前命令窗口。

### langmode指令

该指令用于切换系统显示语言模式。

### shutdown指令

该指令用于关闭计算机。

### sysmode指令

该指令用于切换系统模式。.

### echo指令

该指令用于系统输出。

# KalinoteOS应用程序(KAL)

待补充

# KalinoteOS系统内部API

除了C语言自带的函数以外，系统还编写了大量API供系统开发内部使用，本章将对系统内部API分为数据及任务处理、显示画面处理、外部设备驱动三个大类分类讲解。

## 数据及任务处理

待补充

## 显示画面处理

待补充

## 外部设备驱动

待补充

# KalinoteOS系统调用

这一章介绍的是KalinoteOS中给外部应用程序提供的API，在KalinoteOS中，使用EDX寄存器储存API编号，所以API编号范围为1-4,294,967,295(32位)。

### \_api\_putchar:

寄存器数据：

EDX = 1

AL = 字符

描述：

在命令窗口打印单个字符。

调用方法：

**void api\_putchar(int c);**

参数int c 是需要打印的char类型数据，返回类型是void，无返回数据。

### \_api\_putstr0:

寄存器数据：

EDX = 2

EBX = 字符串

描述：

在命令窗口打印字符串，并在结尾为字符编码0时结束。

调用方法：

**void api\_putstr0(char \*s);**

参数char \*s 是需要打印的字符串指针，返回类型是void，无返回数据。

### cons\_putstr1:

寄存器数据：

EDX = 3

EBX = 字符串

描述：

在命令窗口打印字符串，并提前指定字符串长度。

调用方法：

**void api\_putstr1(char \*s, int l);**

参数char \*s 是需要打印的字符串指针，int l是打印字符串长度，返回类型是void，无返回数据。

### api\_end:

寄存器数据：

EDX = 4

描述：

用于结束运行应用程序。

调用方法：

**void api\_end(void);**

### api\_openwin:

寄存器数据：

EDX = 5

EBX = 窗口缓冲区

ESI = 窗口X(高度)

EDI = 窗口Y(宽度)

EAX = 透明色

ECX = 窗口名称

描述：

用于生成一个基本窗口。

调用方法：

**int api\_openwin(char \*buf, int xsiz, int ysiz, int col\_inv, char \*title);**

### \_api\_putstrwin:

寄存器数据：

EDX = 6

EBX = 窗口句柄

ESI = 显示字符串X坐标

EDI = 显示字符串Y坐标

EAX = 色号

ECX = 字符串长度

EBP = 字符串

描述：

用于在指定窗口的指定位置上打印一个字符串。

调用方法：

**void api\_putstrwin(int win, int x, int y, int col, int len, char \*str);**

### \_api\_boxfilwin:

寄存器数据：

EDX = 7

EBX = 窗口句柄

EAX =

ECX =

ESI =

EDI =

EBP = 色号

描述：

用于在指定窗口上打印一个方块。

调用方法：

**void api\_boxfilwin(int win, int x0, int y0, int x1, int y1, int col);**

### \_api\_initmalloc:

寄存器数据：

EDX = 8

EBX = memman地址

EAX = memman所管理的内存空间的起始地址

ECX = memman所管理的内存空间的字节数

描述：

用于初始化内存分配程序。

调用方法：

**void api\_initmalloc(void);**

### \_api\_malloc:

寄存器数据：

EDX = 9

EBX = memman的地址

ECX = 需要请求的字节数

EAX = 分配到的内存空间地址

描述：

用于分配内存空间。

调用方法：

**char \*api\_malloc(int size);**

### \_api\_free:

寄存器数据：

EDX = 10

EBX = memman地址

EAX = 需要释放的内存空间地址

ECX = 需要释放的字节数

描述：

用于释放内存空间。

调用方法：

**void api\_end(void);**

### \_api\_point:

寄存器数据：

EDX = 11

EBX = 窗口句柄

ESI = 显示位置的X坐标

EDI = 显示位置的Y坐标

EAX = 色号

描述：

在指定窗口的某个位置描绘一个像素点。

调用方法：

**void api\_point(int win, int x, int y, int col);**

### \_api\_refreshwin:

寄存器数据：

EDX = 12

EBX = 窗口句柄

EAX =

ECX =

ESI =

EDI =

描述：

用于窗口的区域渲染，在窗口绘制完成后刷新以显示。

使用方法：

**void api\_refreshwin(int win, int x0, int y0, int x1, int y1);**

### \_api\_linewin:

寄存器数据：

EDX = 13

EBX = 窗口句柄

EAX =

ECX =

ESI =

EDI =

EBP = 色号

描述：

用于窗口直线渲染。

使用方法：

**void api\_linewin(int win, int x0, int y0, int x1, int y1, int col);**

### \_api\_closewin:

寄存器数据：

EDX = 14

EBX = 窗口句柄

描述：

用于关闭窗口。

使用方法：

**void api\_closewin(int win);**

### \_api\_getkey:

寄存器数据：

EDX = 15

EAX = 获取模式

EAX = 键盘输入的字符编码

描述：

用于获取键盘数据，关于获取模式：参数为0的话该函数在没有键盘输入时返回 -1，参数为1的话该函数在没有键盘输入时休眠。

使用方法：

**int api\_getkey(int mode);**

### \_api\_alloctimer:

寄存器数据：

EDX = 16

EAX = 定时器句柄(由系统返回)

描述：

从系统申请一个定时器。

使用方法：

**int api\_alloctimer(void);**

### \_api\_inittimer:

寄存器数据：

EDX = 17

EBX = 定时器句柄

EAX = 数据

描述：

设定一个数据，在定时器超时后会返回这个数据。

使用方法：

**void api\_inittimer(int timer, int data);**

### \_api\_settimer:

寄存器数据：

EDX = 18

EBX = 定时器句柄

EAX = 时间(ms，毫秒)

描述：

设定一个时间，产生延时。

使用方法：

**void api\_settimer(int timer, int time);**

### \_api\_freetimer:

寄存器数据：

EDX = 19

EBX = 定时器句柄

描述：

释放定时器。

使用方法：

**void api\_freetimer(int timer);**

### \_api\_beep:

寄存器数据：

EDX = 20

EAX = 声音频率(0为停止)

描述：

控制蜂鸣器发声。

使用方法：

**void api\_beep(int tone);**

### \_api\_fopen:

寄存器数据：

EDX = 21

EBX = 文件名

EAX = 文件句柄(为0时表示打开失败，由操作系统返回)

描述：

用于打开一个文件。

使用方法：

**int api\_fopen(char \*fname);**

### \_api\_fclose:

寄存器数据：

EDX = 22

EAX = 文件句柄

描述：

用于关闭一个文件。

使用方法：

**void api\_fclose(int fhandle);**

### \_api\_fseek:

寄存器数据：

EDX = 23

EAX = 文件句柄

ECX = 定位模式

EBX = 定位偏移量

描述：

用于寻找文件时的文件定位，其中定位模式有以下几种：

ECX = 0 定位的起点为文件开头

ECX = 1 定位的起点为当前访问的位置

ECX = 2 定位的起点为文件末尾

使用方法：

**void api\_fseek(int fhandle, int offset, int mode);**

### \_api\_fsize:

寄存器数据：

EDX = 24

EAX = 文件句柄

ECX = 文件大小获取模式

描述：

用于获取文件大小，其中获取模式有以下几种：

ECX = 0 普通文件大小

ECX = 1 当前读取位置从文件开头起算的偏移量

ECX = 2 当前读取位置从文件末尾起算的偏移量

使用方法：

**int api\_fsize(int fhandle, int mode);**

### \_api\_fread:

寄存器数据：

EDX = 25

EAX = 文件句柄

EBX = 缓冲区地址

ECX = 最大读取字节数

EAX = 本次读取到的字节数(由系统返回)

描述：

用于读取文件。

使用方法：

**int api\_fread(char \*buf, int maxsize, int fhandle);**

### \_api\_cmdline:

寄存器数据：

EDX = 26

EBX = 存放命令行内容的地址

ECX = 最多可以放多少字节

EAX = 实际存放了多少字节(操作系统返回)

描述：

用于获取用户在命令行输入的参数

使用方法：

**int api\_cmdline(char \*buf, int maxsize);**

### \_api\_getlang:

寄存器数据：

EDX = 27

EAX = langmode(由系统返回)

描述：

查询当前系统语言模式。

使用方法：

**int api\_getlang(void);**

### \_api\_cls:

寄存器数据：

EDX = 29

描述：

清空cmd内容。

使用方法：

**void api\_cls(void);**

### \_api\_get\_year;

寄存器数据：

EDX = 30

EAX = 操作系统当前年(操作系统返回)

描述：

获取系统时间：年。

使用方法：

**unsigned int api\_get\_year(void);**

### \_api\_get\_mon\_hex;

寄存器数据：

EDX = 31

EAX = 操作系统当前月(操作系统返回)

描述：

获取系统时间：月。

使用方法：

**unsigned int api\_get\_mon\_hex(void);**

### \_api\_get\_day\_of\_week;

寄存器数据：

EDX = 32

EAX = 操作系统当前星期(操作系统返回)

描述：

获取系统时间：星期。

使用方法：

**unsigned int api\_get\_day\_of\_week(void);**

### \_api\_get\_day\_of\_month;

寄存器数据：

EDX = 33

EAX = 操作系统当前日(操作系统返回)

描述：

获取系统时间：日。

使用方法：

**unsigned int api\_get\_day\_of\_month(void);**

### \_api\_get\_hour\_hex;

寄存器数据：

EDX = 34

EAX = 操作系统当前小时(操作系统返回)

描述：

获取系统时间：时。

使用方法：

**unsigned int api\_get\_hour\_hex(void);**

### \_api\_get\_min\_hex;

寄存器数据：

EDX = 35

EAX = 操作系统当前分(操作系统返回)

描述：

获取系统时间：分。

使用方法：

**unsigned int api\_get\_min\_hex(void);**

### \_api\_get\_sec\_hex;

寄存器数据：

EDX = 36

EAX = 操作系统当前秒(操作系统返回)

描述：

获取系统时间：秒。

使用方法：

**unsigned int api\_get\_sec\_hex(void);**

# 标准函数API

下面是KalinoteOS支持的C语言标准API，具体使用方法可以查询C语言支持手册。

### string.h

int strchr(const char \*str, char c);

### malloc.h(stdlib.h)

void \*malloc(int size);

void free(void \*p);

### stdio.h

int putchar(int c);

int printf(char \*format, ...);

int scanf(const char \*format, ...);

int getchar();

int puts(const char \*str);

char \*gets(char \*str);

### ctype.h

int isspace(char c);

int isdigit(char c);

### stdlib.h

void exit(int status);

# 系统及kal应用软件的编译过程

待补充

# 参考资料

1. I/O端口地址分配表：

<https://blog.csdn.net/zeflove/article/details/7337012>

1. IDE硬盘端口操作：

<https://blog.csdn.net/loafertb/article/details/5262814>

1. BIOS数据区：

<https://blog.csdn.net/x86ipc/article/details/5303760>

1. FAT12文件系统

<https://blog.csdn.net/m0_37329910/article/details/88927673>