**KalinoteOS 系统支持文档**

目录

[1. 说明 1](#_Toc62324203)

[2. 系统源代码目录结构及对应功能 1](#_Toc62324204)

[3. 系统kernel工作方式 1](#_Toc62324205)

[3.1 系统内存管理方式 1](#_Toc62324206)

[内存管理程序(memman)数据结构 2](#_Toc62324207)

[系统内存分配原理 2](#_Toc62324208)

[系统内存释放原理 2](#_Toc62324209)

[3.2 系统数据结构 3](#_Toc62324210)

[队列(FIFO)数据缓冲区 3](#_Toc62324211)

[双链表(List) 3](#_Toc62324212)

[3.3 系统进程管理方式 3](#_Toc62324213)

[TASK数据结构 3](#_Toc62324214)

[任务等级(TASKLEVEL) 4](#_Toc62324215)

[任务状态段(TSS) 4](#_Toc62324216)

[3.4 系统图层(sheet)管理方式 4](#_Toc62324217)

[图层(sheet)数据结构 4](#_Toc62324218)

[图层控制(SHTCTL)数据结构 5](#_Toc62324219)

[图层高度及其调整 5](#_Toc62324220)

[图层flags定义 6](#_Toc62324221)

[子图层(subsheet)的创建与子图层管理器(sht->subctl) 7](#_Toc62324222)

[系统对各个种类图层的处理 7](#_Toc62324223)

[图层与任务(task)的关系 7](#_Toc62324224)

[3.5 系统窗口(window)绘制方式 7](#_Toc62324225)

[系统字符绘制方式 7](#_Toc62324226)

[系统字符串的多种绘制方式 7](#_Toc62324227)

[系统方块(box)的绘制方式 7](#_Toc62324228)

[系统窗口的绘制方式 7](#_Toc62324229)

[系统图标(icon)的绘制方式 7](#_Toc62324230)

[系统菜单栏的管理方式 7](#_Toc62324231)

[3.6 系统段号记录表及中断向量号管理方式 7](#_Toc62324232)

[3.7 命令行(console) 7](#_Toc62324233)

[3.8 系统全局快捷键 7](#_Toc62324234)

[4. 系统内置CMD指令 7](#_Toc62324235)

[4.1 查询类 7](#_Toc62324236)

[mem指令 8](#_Toc62324237)

[dir&ls指令 8](#_Toc62324238)

[4.2 功能类 8](#_Toc62324239)

[cls&clear指令 8](#_Toc62324240)

[type [Filename]指令 8](#_Toc62324241)

[exit 指令 8](#_Toc62324242)

[start指令 8](#_Toc62324243)

[run指令 8](#_Toc62324244)

[langmode指令 8](#_Toc62324245)

[shutdown指令 9](#_Toc62324246)

[sysmode指令 9](#_Toc62324247)

[echo指令 9](#_Toc62324248)

[5. KalinoteOS系统内部API 9](#_Toc62324249)

[5.1 数据及任务处理 9](#_Toc62324250)

[5.2 显示画面处理 9](#_Toc62324251)

[5.3 外部设备驱动 9](#_Toc62324252)

[6. KalinoteOS系统外部API 9](#_Toc62324253)

[\_api\_putchar: 9](#_Toc62324254)

[\_api\_putstr0: 10](#_Toc62324255)

[cons\_putstr1: 10](#_Toc62324256)

[api\_end: 10](#_Toc62324257)

[api\_openwin: 10](#_Toc62324258)

[\_api\_putstrwin: 11](#_Toc62324259)

[\_api\_boxfilwin: 11](#_Toc62324260)

[\_api\_initmalloc: 11](#_Toc62324261)

[\_api\_malloc: 12](#_Toc62324262)

[\_api\_free: 12](#_Toc62324263)

[\_api\_point: 12](#_Toc62324264)

[\_api\_refreshwin: 13](#_Toc62324265)

[\_api\_linewin: 13](#_Toc62324266)

[\_api\_closewin: 13](#_Toc62324267)

[\_api\_getkey: 14](#_Toc62324268)

[\_api\_alloctimer: 14](#_Toc62324269)

[\_api\_inittimer: 14](#_Toc62324270)

[\_api\_settimer: 14](#_Toc62324271)

[\_api\_freetimer: 15](#_Toc62324272)

[\_api\_beep: 15](#_Toc62324273)

[\_api\_fopen: 15](#_Toc62324274)

[\_api\_fclose: 15](#_Toc62324275)

[\_api\_fseek: 16](#_Toc62324276)

[\_api\_fsize: 16](#_Toc62324277)

[\_api\_fread: 16](#_Toc62324278)

[\_api\_cmdline: 17](#_Toc62324279)

[\_api\_getlang: 17](#_Toc62324280)

[\_api\_cls: 17](#_Toc62324281)

[7. 标准函数API 17](#_Toc62324282)

[string.h 18](#_Toc62324283)

[malloc.h(stdlib.h) 18](#_Toc62324284)

[stdio.h 18](#_Toc62324285)

[ctype.h 18](#_Toc62324286)

[stdlib.h 18](#_Toc62324287)

[8. 后面要做的事 18](#_Toc62324288)

[压缩算法 18](#_Toc62324289)

[API函数标准化(标准函数) 18](#_Toc62324290)

# 说明

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、软件等皆指在KalinoteOS上运行的kal或其他支持在KalinoteOS上运行的任何软件。

# 系统源代码目录结构及对应功能

# 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。

## 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)

## 系统进程管理方式

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)

### 任务状态段(TSS)

## 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)

### 系统对各个种类图层的处理

### 图层与任务(task)的关系

## 系统窗口(window)绘制方式

### 系统字符绘制方式

### 系统字符串的多种绘制方式

### 系统方块(box)的绘制方式

### 系统窗口的绘制方式

### 系统图标(icon)的绘制方式

### 系统菜单栏的管理方式

## 系统段号记录表及中断向量号管理方式

## 命令行(console)

## 系统语言支持

## 系统全局快捷键

# 系统内置CMD指令

## 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。

### dir&ls指令

该指令用于查询系统中的文件。

## 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。

### type [Filename]指令

type指令用于输出某个文件的内容。

### exit 指令

该指令用于关闭命令窗口。

### 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

## 显示画面处理

## 外部设备驱动

# KalinoteOS系统外部API

### \_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);**

### \_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);**

### 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：

停止使用

### 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字符串。

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（这个api暂时不能使用）。

使用方法：

void api\_cls(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);

# 后面要做的事

### 压缩算法

### API函数标准化(标准函数)