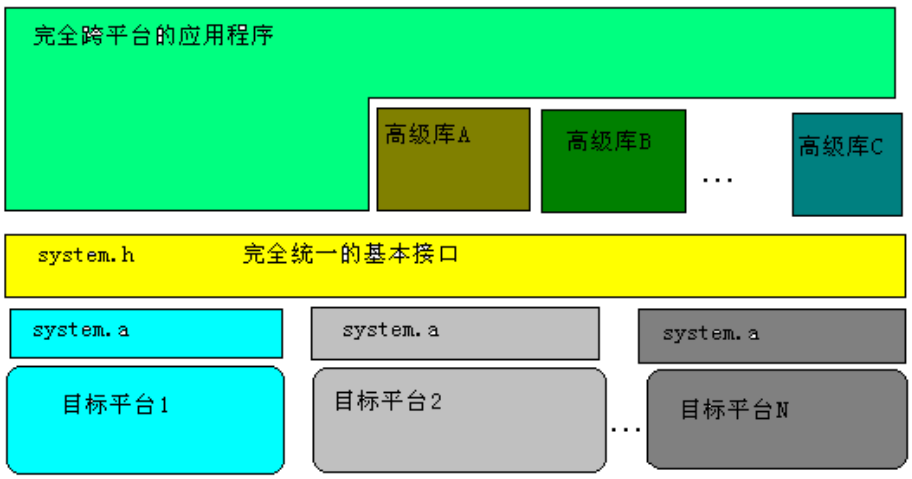
# 嵌入式可移植性开发平台

该开发平台具有高度的可移植性.所有核心代码均不需要改动,甚至不需要宏开关就能正确在各个平台上生成效果完全一致的目标程序.

## 平台架构

平台结构示意图




该平台分为 3 个部分,系统接口层,可移植函数库,和应用代码层.

系统接口层负责抽象目标平台接口,由平台设计者定义,平台移植开发者提供.

可移植函数库是一套可多平台编译的常用函数代码库,它仅使用标准 ANSI C 函数和本平台系统接口层抽象出来的函数,实现高级的功能.由平台库开发者提供.

应用代码层则是应用程序的主体,用户开发将在这里完成.由应用程序开发者提供.

简单的说,这套开发平台是定义了一组跨平台的接口和代码组织方法,使得符合规范的应用程序能实现源代码级别的跨平台能力.

## 特点和优势

1. 平台定义了一系列的抽象接口,完全地隔离各个系统的差异,使得代码能可靠地跨平台编译.
2. 平台使用 lib 来封装所有和系统相关的接口,便于各个系统移植者发布系统的编译环境时,不必提供其相应的 source 文件,保证了保密性要求.
3. 平台提供了跨平台的 lib 集合,并且提供源代码,便于移植到不同的系统中.
4. 每个层次的模块都相互独立,由各部分开发者保证其完整可靠性,并且提供了完整的自检程序.便于分工专注地开发,并且将错误模块隔离,调试方便.
5. 程序在发布的时候,可以同时提供 Win32 版本供用户预览,非常方便.
6. 大部分开发工作都是借助 Win32 的 VC 来 debug,这比直接在嵌入式系统上调试直观且方便.开发效率和机动性(你大多数时间不再花费在无数次的烧录-调试的循环中,而且不需要背着一堆的嵌入式工具到处跑)大大提高.
7. 平台并没有定义任何的编译规则,这意味着任何的编译环境都可以整合到这个平

台上.

8. 在平台升级的过程中,我们保证 100%的兼容性,就是说,代码总是能在更新的平台上正确编译.

## 目录介绍

Lib 目录

Lib 是系统库目录,存放有对应各个平台的库文件.其中的 source 目录是库文件代码.

Inc 目录

Inc 目录是平台头文件目录,分别对应与 lib 目录里的每一个文件.其中的 global.h 包含所有头文件,供各个模块统一调用.

Source 目录

Source 目录是用户程序开发目录,包括 source 的编译子目录等等都放在这里.这个目录并不一定起名为 Source,可由用户自由定义.例如我写的 selftest 目录就是 source 类型的.

Doc 目录

相关文档.

## Lib 简介

### 已经存在的 Lib

Shell.a 和 shell.lib 等等 shell 库是系统接口库,负责抽象所有的接口.

Font 是内置的一个 16 点阵字体库,可以驱动显示出 GB2312 编码字符.

Miniio 是微型标准 IO 库,实现一些常用的 shdio 风格的接口,如 dprintf(类似 printf 函数).

advlib 是完全可移植的高级功能库, 就是对平台接口的加工封装.

*在 JZ4740 平台上,默认库(libc.a)的文件函数并不可用,所以我已经手动去除了相关函数,并将这些函数在 shell.a(system.c)中重新实现.

## Lib 添加方法

由于 win32 调试方便,所以平台设计为主要在 win32 开发调试代码,最后再编译到目标平台,做优化等后续工作.

Win32 使用 VC6 开发,selftest 目录有顶层工作空间(*.dsw),内包含了下属的所有的代码(包括 lib 代码)的编译环境(*.dsp).

Lib 编译的方式:

1.使用 VC6 的新建向导创建一个静态库编译环境,把所有的 Lib 代码添加到 Source.

2.在工程设置里,在自定义组建里添加几行代码

Copy 目标.lib ..\..\..\lib\环境库目录

Copy 目标头文件.h ..\..\..\..\inc

*上面的目标目录(文件)需要你自己改成实际的目录位置.输出部分随便写几个字

符.

这样就实现了在编译 lib 的过程中自动更新了大环境的库和头文件.

Lib 每个源文件都可以包含 global.h,实现对其他库的调用.

*在 Win32.dsw 里,最好要添加对所有需要的 lib 的依赖.这样才能让编译器根据 lib 的更新情况来 rebuild.

## 优化与实现

既然本平台主要目标是嵌入式系统,速度也是最重要的因素之一.

本平台提供了兼容性极强的接口,然而内部实现必然因平台硬件不同而不同,同样的参数,效率也可能相差千百倍.

解决方案是,对于每一个平台,每一个接口设置参数,都有一个默认值,这个默认值就是该平台最优化的输入形式.应用程序可以在初始化的时候就查询这些最优化输出参数,尽可能地进行相应的优化输出.

而对于性能要求不高的部分,应用程序也可以直接按最喜欢的方式输出,因为接口标准保证:对于任何有效的参数,接口的功能不应平台而异.也就是说,即使你在只有 320*240 的设备上使用 640*480 的输出,程序也能正常输出,只不过你看到了裁剪或者缩小的画面,速度略慢而已.

## 平台移植开发者测试

对于不同的平台,需要平台移植开发者重新实现一套基本接口.

该接口都在 system.h 中有定义,这是实现一个移植到新平台上的最基本接口,实现了这个接口,所有该平台代码就可以正确编译到新平台上了.

为了减轻调试负担,我在开发平台时,写了一个 selftest 应用级程序,它是一个自我测试程序,测试了平台接口的每一个方面,只要这个程序在新平台上运行的效果和 Win32 上运行的效果一致,则说明这个新平台的 system 接口的实现方法已经基本正确了.

selftest 的注意事项:

1. fild_test 的路径必须是根据平台而修改.
2. sound_test 用到了 sound1.wav,必须放在机器的默认打开目录下.

# 平台基本接口说明

## Waveout 接口

相关函数

声音设备初始化

s32 sys_sound_init( u32 freq , u32 bit , u32 channel );

打开声音设备,如果打开成功,返回 0,错误返回 1

| | | |
|---|---|---|
| 输入参数 | u32 freq | 采样率的值,支持的频率值是 8000,16000,22050,44100 |
| | u32 Bit | 声音的采样位数,支持 8 和 16 两个值. |
| | u32 channel | 声音的通道数,支持 1 和 2,分别代表单声道和立体声. |

| | | |
|---|---|---|
| 返回值 | s32 | 0:打开成功;其他值:打开失败 |

声音设备卸载

void sys_sound_deinit( void );

默认不返回值,因为无论在哪种情况下,函数都要保证卸载都是成功的.

写声音数据

u32 sys_sound_write( void *data , u32 length );

该函数将往系统声音缓存区填充 pcm 数据,如果 pcm 缓存已满(或者大于 length 长度),则函数将挂起直到能够正常写入为止.函数将复制数据到内部缓存,因此返回后,PCM 数据区已经可以自由修改.

| | | | |
|---|---|---|---|
| 输入参数 | void *data | PCM 数据地址 | |
| | u32 length | PCM 数据长度(字节单位) | |
| 返回值 | u32 | >0 | 实际写入字节数 |
| | | =0 | 写入失败 |

检查声音设备是否可写

s32 sys_sound_canwrite( void );

这个函数一般用来检查内部声音缓存是否空闲,返回值为 0 时,调用 sys_sound_write 将可能会挂起一段时间.

返回值为 1 时, sys_sound_write 将可以快速的返回.

| | | | |
|---|---|---|---|
| 返回值 | s32 | 0 | 不可写 |
| | | 1 | 可写 |

调整系统音量

void sys_volume( int volume );

| | | |
|---|---|---|
| 输入参数 | int volume | 音量值 |
| 返回值 | 无 | |

音量值最大值是 100,最小值是 0(静音).

## 文件查找函数

文件查找函数集,实现文件的遍历,属性的获取等等功能.

相关数据结构:

```
typedef struct FILE_FIND_BLOCK{
    u32 attrib;
    u8* name;
    u32 size;
    time_t time_create;
    time_t time_write;
    time_t time_access;
    void *usrdat1;
    void *usrdat2;
```

```
    void *usrdat3;
    void *usrdat4;
} FILE_FIND_BLOCK;
```

FILE_FIND_BLOCK 是一个文件查找的关键数据结构,文件查找过程需要使用这个结构,它由 find_open 创建, find_close 释放.

| | |
|---|---|
| u32 attrib | 文件属性,具体含义请查阅文件属性部分 |
| u8* name | 文件名 |
| u32 size | 文件大小,目前最大支持 4G 文件 |
| time_t time_create | 文件创建时间 |
| time_t time_write | 文件修改时间 |
| time_t time_access | 文件访问时间 |
| void *usrdat1 | 为各个内部实现存储用的保留数据区,对应用程序无意义,请不要在应用程序里修改或者读取它们. |
| void *usrdat2 | |
| void *usrdat3 | |
| void *usrdat4 | |

```
enum{
    FFD_READONLY = 0x1,
    FFD_WRITE = 0x2,
    FFD_SYSTEM = 0x4,
    FFD_HIDDEN = 0x8,
    FFD_VOLUME = 0x10,
    FFD_DIR = 0x20,
    FFD_ARCHIVE = 0x40,
    FFD_DIR_OR = 0x80,
    FFD_UNKNOW = 0x80000000,
};
```

文件属性类型定义

| | |
|---|---|
| FFD_READONLY | 只读 |
| FFD_WRITE | 只写 |
| FFD_SYSTEM | 系统 |
| FFD_HIDDEN | 隐藏 |
| FFD_VOLUME | 卷标(盘符) |
| FFD_DIR | 目录 |
| FFD_ARCHIVE | 存档 |
| FFD_DIR_OR | 未使用 |
| FFD_UNKNOW | 未知类型 |

函数部分:

打开一个查找文件的过程

```
FILE_FIND_BLOCK *find_open( u8 *path );
```

| | | | |
|---|---|---|---|
| 参数 | u8 *path | 需要查找文件的路径 | |
| 返回值 | FILE_FIND_BLOCK * | ≠NULL | 指向一个查找结构的指针 |

| | | | |
|---|---|---|---|
| | | =NULL | 查找过程打开失败,可能是路径不存在,内存不足等等因素. |

注意:

path 参数可以设为:

1.空字串(””),此时隐含的意义为列出整个存储系统的根目录.例如:在 Win32 上列出的是盘符,在 linux 上列出”/”下的所有目录.

2.绝对路径

3.相对路径”/”或”\”,指相对于当前目录的目录.支持使用”..\”,”.\”,”../”,”./”作为相对路径名.

4.路径名可以以’\’.’/’结尾.

关闭查找文件过程

void find_close( FILE_FIND_BLOCK *block );

| | | |
|---|---|---|
| 参数 | FILE_FIND_BLOCK *block | 一个查找过程的结构指针,一般来说就是 find_open 得到的指针. |
| 返回值 | 无 | 函数保证正常使用下都关闭成功.但是也不可以对一个结构做两次关闭,或者关闭一个无效的指针,可能会引起不可预知的错误. |

查找下一个文件

int find_next( FILE_FIND_BLOCK *block );

| | | |
|---|---|---|
| 参数 | FILE_FIND_BLOCK *block | 一个查找过程的结构指针,一般来说就是 find_open 得到的指针. |
| 返回值 | 0 | 本次文件枚举失败 |
| | 1 | 本次文件枚举成功 |

注意:

1.返回 0 的时候,你应当停止继续枚举文件了,这种情况通常都是因为确实枚举完所有的目录了.

2.返回 1 的时候, block 的值才是有效的.

3.本次枚举的文件数据均写入 block 中,具体请参阅 FILE_FIND_BLOCK 的内容.

4.查找函数会找到指定目录下的所有文件(包括隐藏文件).

# 屏幕与显示函数

设置屏幕模式

s32 screen_set_mode( u32 width, u32 height, u32 colormode );

| | | | |
|---|---|---|---|
| 参数 | u32 width | 屏幕宽度 | |
| | u32 height | 屏幕高度 | |
| | u32 colormode | 色彩模式(见下表) | |
| 返回值 | s32 | 0 | 设置失败 |
| | | 1 | 设置成功 |

色彩模式表

| | |
|---|---|
| LCD_A8R8G8B8 | LCD 32 位色彩,从高到低分别为 Alpha,Red,Green,Blue,每个色彩占 8 位. |

1.程序可以自由设置色彩和屏幕模式,但是必须判断返回值.设置失败时,原有的模式不会改变.
2.调用这个程序可能引起部分程序显示失效,原因是程序没有即时得到或处理显示模式更新的消息.因此请最好在初始化的时候才调用本函数.
3.显示模式的改变将立即反映到系统中的 screen_get_width 和 screen_get_height 函数中.因此,在涉及获取显示模式的地方使用这两个函数将让你的程序获得动态改变显示模式的能力.

获取屏幕宽度

u32 screen_get_width( void );

| | | |
|---|---|---|
| 参数 | 无 | |
| 返回值 | u32 | 屏幕宽度(像素为单位) |

获取屏幕高度

u32 screen_get_height( void );

| | | |
|---|---|---|
| 参数 | 无 | |
| 返回值 | u32 | 屏幕高度(像素为单位) |

更新屏幕

void lcd_updateui( void );

| | | |
|---|---|---|
| 参数 | 无 | |
| 返回值 | 无 | |

当你把图像数据送入屏幕缓存时,你还需要调用这个函数才能让图像刷新到屏幕上.不要经常调用它,因为这是一个相对耗费时间的操作,尤其是进行软件色彩转换和缩放的时候.最好的情况是保证你想绘制的东西都完成后,最后才刷新到屏幕上.

清空屏幕缓存

void lcd_clearui( u32 color );

| 参数 | u32 color | 32bit 色彩数据(因设置的色彩模式而定) |
|---|---|---|
| 返回值 | 无 | |

这个函数的作用就是把屏幕用同一种颜色填满,就像 dos 下的 cls 命令一样.当然,你必须调用 lcd_updateui 才能看到最终效果.

返回屏幕缓存
u32* lcd_bufferui( void );

| 参数 | 无 | |
|---|---|---|
| 返回值 | u32* | 屏幕缓存地址 |

返回值是 32bit 指针的屏幕缓存地址,只要把色彩数据写到这个缓存,调用 lcd_updateui 之后,就可以在屏幕上看到图案了.

设置屏幕亮度(返回 0--成功(范围 0~100))
int screen_set_backlight( int value );

| 参数 | int value | 亮度值(0~100) |
|---|---|---|
| 返回值 | int | 0 成功,其他失败 |

亮度值 0 最暗,100 最亮
在 windows 系统中,这个函数没有效果,但是仍可调用.

## 系统调度及其他函数

系统延时(毫秒)
void sys_delay( u32 time );

| 参数 | u32 time | 延时时间(以毫秒为单位) |
|---|---|---|
| 返回值 | 无 | |

注意这个函数的实现方法,最好不要使用忙等待,使用系统挂起函数或者省电函数实现.
精度保证在 10ms 之内即可.

执行系统任务
void sys_event( void );
这个函数的作用在多线程函数中起作用,将控制权主动交给系统,如果想让系统响应事件,请在程序主循环调用它.

获取系统周期(毫秒)
u32 sys_ticks( void );

| 参数 | 无 | |
|---|---|---|
| 返回值 | u32 | ticks 值(毫秒为单位) |

ticks 值在计数溢出后,将转回 0 继续累加.

## 输入控制器函数

读取按键

u32 sys_get_key( void );

| | | |
|---|---|---|
| 参数 | 无 | |
| 返回值 | u32 | 按键键值 |

这个按键读取函数是读取 32 位的常用按键键值.这个按键不包括 A~Z,1~9 等数据按键.主要设计给嵌入式系统的按键,如方向键,电源键,功能键等等.

可用的按键说明:

| | |
|---|---|
| SYSKEY_UP | 方向键上 |
| SYSKEY_DOWN | 方向键下 |
| SYSKEY_LEFT | 方向键左 |
| SYSKEY_RIGHT | 方向键右 |
| SYSKEY_SELECT | 选择键 |
| SYSKEY_START | 开始键 |
| SYSKEY_MENU | 菜单键 |
| SYSKEY_VOLUP | 音量增大键 |
| SYSKEY_VOLDOWN | 音量减小键 |
| SYSKEY_POWER | 电源键 |
| SYSKEY_ESCAPE | 退出键 |

读取定位设备

读取直接定位设备(如鼠标,触摸板,触摸屏等等)

int sys_get_pointer( int *x, int *y, int *press );

| | | | |
|---|---|---|---|
| 参数 | int *x | 将接受的 X 坐标 | |
| | int *y | 将接受的 Y 坐标 | |
| | int *press | 将接受的压力值 | |
| 返回值 | int | 0 | 成功 |
| | | 非 0 | 失败 |

压力值的最大值是 POINTER_PRESS_MAX

压力值的最小时是 POINTER_PRESS_MIN(完全松开的状态)

在 Win32 用鼠标模拟的时候,压力值只有两个(最大和最小).

## 路径与参数接口

获取命令行参数

void sys_get_arg( int *argc,char *argv[] , int maxargc );

| | | |
|---|---|---|
| 参数 | int *argc | 将接收的到参数个数 |
| | char *argv[] | 参数指针数组 |
| | int maxargc | 最大接收参数个数 |
| 返回值 | 无 | |

利用这个函数可以获取程序命令行参数,当 argc 被设为 0 时,则证明没有获取参数,参数格式和常见的 main 函数参数格式一致.

获取当前目录

u8 *sys_get_dir( u8 *buf , int maxlen );

| 参数 | u8 *buf | 接受缓存 |
|---|---|---|
| | int maxlen | 最大长度 |
| 返回值 | u8 * | 返回 buf 值 |

成功获取则返回 buf 值,失败则返回 NULL.

获取应用程序路径

u8 *sys_get_path( u8 *buf , int maxlen );

| 参数 | u8 *buf | 接受缓存 |
|---|---|---|
| | int maxlen | 最大长度 |
| 返回值 | u8 * | 返回 buf 值 |

成功获取则返回 buf 值,失败则返回 NULL.

程序路径就是该应用程序所在的目录.

注意了:如果你在程序中使用了 sys_get_arg 函数获取参数的话,你必须在你的程序中定义下面这个回调函数,用来给系统提供文件类型信息.

(回调)注册关联文件类型

```
u8 * sys_reg_filetype( void );
```

| 参数 | void | |
|---|---|---|
| 返回值 | u8 * | 文件类型过滤字符串 |

返回 0 – 成功, 其他—失败

注册文件类型是回调函数,由系统在执行程序之前调用,获取程序可以关联的文件类型.

文件类型这样描述:[文件扩展名 1]| [文件扩展名 2]|...

例如下面的例子,关联了 BMP 和 JPG 文件.

```
u8 * sys_reg_filetype( void )
{
    return “BMP|JPG”
}
u8 * sys_reg_filetype( void )
{
    return “*.BMP”
}
```

# 多线程接口

*使用多线程接口的话,在单 cpu 系统中,请注意你的线程在空闲的时候要调用 sys_delay,以让出 cpu 资源,否则其他线程可能极少得到执行的机会.

目前在接口中,明确可以实现线程出让 cpu 资源的函数有,sys_delay,sys_event,sys_sound_write(缓存满,或者设备忙时会阻塞等待),
fwrite,fread 等 IO 函数依据不同的实现,不一定具有线程阻塞状态.

线程状态枚举变量 SYS_THREAD_STATUS

| | |
|---|---|
| SYS_THREAD_STATUS_STANDBY | 挂起状态 |
| SYS_THREAD_STATUS_RUN | 运行状态 |

线程对象 SYS_THREAD

线程原型
int sys_thread_proc( void * param, SYS_THREAD * thread );

创建线程
SYS_THREAD * sys_create_thread( int stack_size, int (*sys_thread_proc)( void * param, SYS_THREAD * thread ), void * param, SYS_THREAD_STATUS status );

| | | |
|---|---|---|
| 参数 | int stack_size | 线程的栈容量 |
| | sys_thread_proc | 线程主函数名(指针) |
| | void * param | 线程主函数自定义参数 |
| | SYS_THREAD_STATUS status | 线程初始化状态 |
| 返回值 | SYS_THREAD * | 返回的线程对象句柄 |

1. 线程函数的栈内存是由内部函数自动分配的.
2. 线程不使用之后必须及时关闭,以释放资源
3. 线程初始化状态是指线程在创建完成之后的状态,如果设为 SYS_THREAD_STATUS_RUN, 则线程在创建完成之后立即运行, SYS_THREAD_STATUS_STANDBY 状态是创建的线程挂起,直到调用 sys_resume_thread 函数才能运行.

挂起线程

int sys_suspend_thread( SYS_THREAD * pThread );

| 参数 | SYS_THREAD * pThread | 线程对象句柄 |
|---|---|---|
| 返回值 | int | 0 为成功,其他值失败 |

1. 程序返回值是 0 的话,目标线程已经成功挂起
2. 非 0 值的返回一般是发生不可预知的错误,一般不会失败,除非挂起了一个不存在的线程.

恢复线程

int sys_resume_thread( SYS_THREAD * pThread );

| 参数 | SYS_THREAD * pThread | 线程对象句柄 |
|---|---|---|
| 返回值 | int | 0 为成功,其他值失败 |

1 程序返回值是 0 的话,目标线程已经成功恢复运行

2. 非 0 值的返回一般是发生不可预知的错误,一般不会失败,除非恢复了一个不存在的线程.

关闭线程

int sys_delete_thread( SYS_THREAD * pThread );

| 参数 | SYS_THREAD * pThread | 线程对象句柄 |
|---|---|---|
| 返回值 | int | 0 为成功,其他值失败 |

1 程序返回值是 0 的话,目标线程已经成功关闭

2. 非 0 值的返回一般是发生不可预知的错误,一般不会失败,除非关闭了一个不存在的线程.

设置线程优先级

int sys_priority_thread( SYS_THREAD * pThread, PRIO_TYPE prio );

```
typedef enum{
    PRIO_TYPE_IDLE,
    PRIO_TYPE_LOWEST,
    PRIO_TYPE_BELOW_NORMAL,
    PRIO_TYPE_NORMAL,
    PRIO_TYPE_ABOVE_NORMAL,
    PRIO_TYPE_HIGHEST,
    PRIO_TYPE_REALTIME,
    PRIO_TYPE_MAX
} PRIO_TYPE;
```

| 参数 | SYS_THREAD * pThread | 线程对象句柄 |
|---|---|---|
| | PRIO_TYPE prio | 优先级 |
| 返回值 | int | 0 为成功,其他值失败 |

优先级列表

| PRIO_TYPE_IDLE | 最低级别,除了 cpu 完全空闲时执行 |
|---|---|
| PRIO_TYPE_LOWEST | 低优先级 |
| PRIO_TYPE_BELOW_NORMAL | 低于标准 |
| PRIO_TYPE_NORMAL | 标准 |
| PRIO_TYPE_ABOVE_NORMAL | 高于标准 |

| | |
|---|---|
| PRIO_TYPE_HIGHEST | 高优先级 |
| PRIO_TYPE_REALTIME | 实时(慎用) |

1. app_main 执行时得到的优先级为 PRIO_TYPE_NORMAL,自己创建的线程优先级默认也为 PRIO_TYPE_NORMAL
2. 最高优先级慎用,如果高优先级线程不主动释放 cpu 资源,整个多线程环境都可能阻塞.