一、文件分析

1、ial.h 输入抽象层相关的数据结构和宏定义

（1）对各类鼠标击键事件进行宏定义。

```
#define IAL_MOUSE_LEFTBUTTON        4
#define IAL_MOUSE_MIDDLEBUTTON      2
#define IAL_MOUSE_RIGHTBUTTON       1
#define IAL_MOUSE_FOURTHBUTTON      8
#define IAL_MOUSE_FIFTHBUTTON       16
#define IAL_MOUSE_SIXTHBUTTON       32
#define IAL_MOUSE_RESETBUTTON  64
```

（2）对鼠标事件和键盘事件进行宏定义。

```
#define IAL_MOUSEEVENT              1
#define IAL_KEYEVENT          2
```

（3）定义处理输入事件的结构体 INPUT

```
typedef struct tagINPUT
{
    char*    id;

    // Initialization and termination
    BOOL (* init_input ) (struct tagINPUT *input,  const char * mdev, const char * mtype);
    void (* term_input ) (void);

    // Mouse operations
    int   (* update_mouse) (void);
    void (* get_mouse_xy ) (int* x, int * y);
    void (* set_mouse_xy) (int x, int y);
    int   (* get_mouse_button ) (void);
    void (* set_mouse_range) (int minx, int miny, int maxx, int maxy);
    void (* suspend_mouse) (void);
    int (* resume_mouse) (void);

    // Keyboard operations
    int   (* update_keyboard ) (void);
    const char* (* get_keyboard_state ) (void);
    void (* suspend_keyboard) (void);
    int (* resume_keyboard ) (void);
    void (* set_leds) (unsigned int leds);
    int (* wait_event ) (int which, int maxfd, fd_set *in, fd_set *out, fd_set *except, struct timeval
*timeout);
    char mdev [MAX_PATH + 1];
} INPUT ;
```

（**4**）定义处理输入事件的函数。

```
#define IAL_InitInput                    (*__mg_cur_input->init_input)
#define IAL_TermInput                      (*__mg_cur_input->term_input)
#define IAL_UpdateMouse                     (*__mg_cur_input->update_mouse)
……
```

（**5**）声明输入抽象层初始化函数和结束函数。

```
int InitIAL  (void);
void TerminateIAL  (void);
```

2、 ial.c 输入抽象层相关函数

（**1**）根据定义的输入引擎 **include**对应的头文件。

```
#ifdef _QVFB_IAL
    #include "qvfbial.h"
#endif
……
```

（**2**）宏定义输入引擎的名称和类型字符长度。

```
#define LEN_ENGINE_NAME              16
#define LEN_MTYPE_NAME               16
```

（**3**）定义 **INPUT** 数组 **inputs**, 存放支持的输入引擎的名称、 初始化和终止函数。

```
static INPUT  inputs [] =
{
#ifdef _DUMMY_IAL
    {"dummy" , InitDummyInput  , TermDummyInput},
#endif
#ifdef _AUTO_IAL
    {"auto" , InitAutoInput, TermAutoInput},
#ifdef _QVFB_IAL
    {"qvfb" , InitQVFBInput, TermQVFBInput},
#endif
……
#ifdef _TFSTB_IAL
    {"TF-STB"  , InitTFSTBInput, TermTFSTBInput},
#endif
#ifdef _HI3610_IAL
    {"hi3610"  , InitHI3610Input, TermHI3610Input},
#endif
};
```

（**4**）定义当前输入。

```
INPUT  * __mg_cur_input;
```

（**5**）获得支持的输入引擎的个数即 **inputs**数组包含的元素个数。

```
#define NR_INPUTS  (sizeof(inputs) /  sizeof(INPUT ))
```

（6）int InitIAL (void)：实现输入抽象层的初始化。

函数的主要作用：

1 获得当前输入引擎的参数，包括名称、路径、类型。

2 选择输入引擎：将 __mg_cur_input 指向当前使用的输入引擎的参数存放的地址：__mg_cur_input = inputs + i ，或者设置为第一个输入引擎。

3 根据获得的参数进行初始化： IAL_InitInput (__mg_cur_input, mdev, mtype)

（7）void TerminateIAL(void)

函数的主要作用：终止输入抽象层

IAL_TermInput ()

（8）定义了一系列鼠标校正函数

当定义了 _MISC_MOUSECALIBRATE 时启用这些函数

3、 event.h 输入事件相关的数据结构和宏定义

（1）定义鼠标移动事件结构体：

```
typedef struct _MOUSEEVENT {
    int event;     //指示事件类型
    int x;         //鼠标的位置
    int y;
    DWORD status;  //状态
} MOUSEEVENT ;
typedef MOUSEEVENT * PMOUSEEVENT ;
```

（2）宏定义不同的鼠标事件类型对应的数值：

```
#define ME_MOVED            0x0000
#define ME_LEFTMASK         0x000F
#define ME_LEFTDOWN         0x0001
#define ME_LEFTUP           0x0002
#define ME_LEFTDBLCLICK     0x0003
#define ME_RIGHTMASK        0x00F0
#define ME_RIGHTDOWN        0x0010
#define ME_RIGHTUP          0x0020
#define ME_RIGHTDBLCLICK    0x0030
#define ME_MIDDLEMASK       0x0F00
#define ME_MIDDLEDOWN       0x0100
#define ME_MIDDLEUP         0x0200
#define ME_MIDDLEDBLCLICK   0x0300
#define ME_REPEATED         0xF000
```

（3）定义键盘事件的结构体

```
typedef struct _KEYEVENT {
    int event;     //事件类型
    int scancode; //按键扫描码
    DWORD status; // 状态信息
} KEYEVENT  ;
typedef KEYEVENT  * PKEYEVENT  ;
```

（4）宏定义不同的键盘事件类型对应的数值：

```
#define KE_KEYMASK                    0x000F
#define KE_KEYDOWN                    0x0001
#define KE_KEYUP                      0x0002
#define KE_KEYLONGPRESS               0x0004
#define KE_KEYALWAYSPRESS             0x0008
#define KE_SYSKEYMASK                 0x00F0
#define KE_SYSKEYDOWN                 0x0010
#define KE_SYSKEYUP                   0x0020
```

（5）定义底层事件数据类型的联合体：表示底层事件的类型可以为鼠标事件也可以为键盘事件。

```
typedef union _LWEVENTDATA {
    MOUSEEVENT   me;
    KEYEVENT   ke;
} LWEVENTDATA   ;
```

（6）定义底层事件结构体：

```
typedef struct _LWEVENT
{
    int type;   //底层事件类型
    int count; // 发生时间
    DWORD  status;   //状态
    LWEVENTDATA    data; // 数据
} LWEVENT  ;
typedef LWEVENT  * PLWEVENT;
```

（7）宏定义三种底层事件的类型：

```
#define LWETYPE_TIMEOUT                  0
#define LWETYPE_KEY                      1
#define LWETYPE_MOUSE                    2
```

（8）声明函数初始化顶层事件函数、终止底层事件函数和获得底层事件函数：

```
BOOL  InitLWEvent   (void);
void TerminateLWEvent   (void);
BOOL  GetLWEvent  (int event, PLWEVENT   lwe);
```

（9）宏定义鼠标事件的相关参数

```
#define MOUSEPARA                              "mouse"
#define MOUSEPARA_DBLCLICKTIME                 "dblclicktime"
#define DEF_MSEC_DBLCLICK                      300
#define EVENTPARA                              "event"
#define EVENTPARA_REPEATUSEC                   "repeatusec"
#define EVENTPARA_TIMEOUTUSEC                  "timeoutusec"
```

```
#define DEF_USEC_TIMEOUT                    300000
#define DEF_REPEAT_TIME                     50000
#define DEF_LPRESS_TIME                     500
#define DEF_APRESS_TIME                     1000
#define DEF_INTERVAL_TIME                   200
```

4、 event.c 输入事件相关函数

（1） static void GetDblclickTime(void)

函数作用：获得鼠标双击最大时间间隔 dblclicktime

（2） static void GetTimeout (void)

函数作用：获得超时时间 timeoutusec 和 repeatusec

（3）定义并初始化鼠标事件参数

```
static int oldbutton = 0;
static unsigned int time1;
static unsigned int time2;
```

（4）定义并初始化键盘事件参数

```
static unsigned int ke_time;
static unsigned char oldkeystate [MGUI_NR_KEYS + 1];
static unsigned char olddownkey = 0;
static DWORD  status;
static int alt1 = 0;                  /* left alt key state */
static int alt2 = 0;                  /* right alt key state */
static int capslock = 0;              /* caps lock key state */
static int esc = 0;                   /* escape scan code detected? */
static int caps_off = 1;              /* 1 = normal position, 0 = depressed */
static int numlock = 0;               /* number lock key state */
static int num_off = 1;               /* 1 = normal position, 0 = depressed */
static int slock = 0;                 /* scroll lock key state */
static int slock_off = 1;             /* 1 = normal position, 0 = depressed */
static int control1 = 0;              /* left control key state */
static int control2 = 0;              /* right control key state */
static int shift1 = 0;                /* left shift key state */
static int shift2 = 0;                /* left shift key state */
```

（5） static void ResetMouseEvent(void)reset 鼠标事件

```
static void ResetMouseEvent(void) {
    oldbutton = 0;
      time1 = 0;
      time2 = 0;
}
```

（6） static void ResetKeyEvent(void)重置键盘事件

```
static void ResetKeyEvent（void）
```

```
{
        memset (oldkeystate, 0, MGUI_NR_KEYS + 1);
        olddownkey     = 0;
        status         = 0;
        alt1           = 0;
        alt2           = 0;
        esc            = 0;
        control1       = 0;
        control2       = 0;
        shift1         = 0;
        shift2         = 0;
        capslock       = 0;
        caps_off       = 1;
        numlock        = 0;
        num_off        = 1;
        slock          = 0;
        slock_off      = 1;
        IAL_SetLeds (slock | (numlock << 1) | (capslock << 2));

        __mg_event_timeout. tv_sec = 0;
        __mg_event_timeout. tv_usec = timeoutusec;
        timeout_threshold = timeoutusec / 10000;
        repeat_threshold = repeatusec / 10000;
        timeout_count = timeout_threshold;
}
```

（**7**）定义并初始化键盘相关的全局变量：长按、一直按和两个按键时间间隔

```
unsigned int __mg_key_longpress_time = 0;
unsigned int __mg_key_alwayspress_time = DEF_APRESS_TIME;
unsigned int __mg_interval_time = DEF_INTERVAL_TIME;
```

（**8**）**static void treat_longpress (PKEYEVENT ke, unsigned int interval)**

函数作用：确定键盘的按键类型，包括：

0

KE_KEYDOWN

KE_KEYLONGPRESS

KE_KEYALWAYSPRESS

（**9**）**BOOL GetLWEvent (int event, PLWEVENT lwe)**

函数作用：根据事件 event 获取底层事件 lwe 的相关参数。将 IAL_** 类型的事件 event 转换为 LWE_**类型的底层事件 lwe。

（**10**）**BOOL GUIAPI GetKeyStatus (UINT uKey)**

函数作用：获取按键状态信息，包括鼠标按键和键盘按键

（**11**）**DWORD GUIAPI GetShiftKeyStatus (void)**

函数作用：获得是 shift 键的状态信息： **return** status;

（12）BOOL InitLWEvent (void)

函数作用：获得鼠标双击时间间隔，获得超时时间，初始化输入抽象层，充值鼠标和键盘

```
    GetDblclickTime ();
    GetTimeout ();
    if (InitIAL ())
        return  FALSE;
    ResetMouseEvent();
    ResetKeyEvent();
    return  TRUE;
```

（13）void TerminateLWEvent (void)

函数作用：结束底层事件

```
    TerminateIAL ();
```

5、 qvfb.h 输入引擎的相关数据结构和宏定义

（1）定义鼠标和键盘的管道信息

```
#define QT_VFB_MOUSE_PIPE       "/tmp/.qtvfb_mouse-%d"
#define QT_VFB_KEYBOARD_PIPE        "/tmp/.qtvfb_keyboard-%d"
```

（2）数据结构 **QVFbHeader**

```
struct QVFbHeader
{
    int width ;
    int height;
    int depth;
    int linestep;
    int dataoffset;
    RECT  update;
    BYTE  dirty ;
    int    numcols ;
    unsigned int clut [256];
};
```

（3）定义 **qvfb** 的键盘事件数据结构

```
struct QVFbKeyData
{
    unsigned int unicode ;
    unsigned int modifiers ;
    BOOL  press;
    BOOL  repeat;
};
```

（4）定义显示的数据结构

```
/* Private display data */
struct GAL_PrivateVideoData {
    unsigned char* shmrgn;
    struct QVFbHeader* hdr;
};
```

6、 qvfb.c 输入引擎相关的函数

（1）声明并定义以下函数：

```
static int QVFB_VideoInit (_THIS, GAL_PixelFormat *vformat);
static GAL_Rect ** QVFB_ListModes (_THIS, GAL_PixelFormat *format, Uint32 flags);
static GAL_Surface *QVFB_SetVideoMode (_THIS, GAL_Surface *current, int width, int height, int bpp, Uint32 flags);
static int QVFB_SetColors (_THIS, int firstcolor, int ncolors, GAL_Color *colors);
static void QVFB_VideoQuit (_THIS);

/* Hardware surface functions */
static int QVFB_AllocHWSurface (_THIS, GAL_Surface *surface);
static void QVFB_FreeHWSurface (_THIS, GAL_Surface *surface);
```

（2）其他函数的定义：

```
static GAL_VideoDevice *QVFB_CreateDevice (int devindex)
static void QVFB_UpdateRects (_THIS, int numrects, GAL_Rect *rects)
static void QVFB_DeleteDevice (GAL_VideoDevice *device)
static int QVFB_Available (void)
```

7、qvfbial.h 输入事件的相关数据结构和宏定义等

（1）定义鼠标和键盘的管道信息

```
#define QT_VFB_MOUSE_PIPE          "/tmp/.qtvfb_mouse-%d"
#define QT_VFB_KEYBOARD_PIPE        "/tmp/.qtvfb_keyboard-%d"
```

（2）定义 qvfb的键盘事件数据结构

```
struct QVFbKeyData
{
    unsigned int unicode ;
    unsigned int modifiers ;
    BYTE press;
    BYTE repeat;
};
```

（3）声明输入引擎初始化和终止函数

```
BOOL InitQVFBInput (INPUT * input, const char* mdev, const char* mtype);
void TermQVFBInput (void);
```

8、 qvfbial.c 输入事件的相关函数

（1）宏定义鼠标和键盘按键的数值

```
#define NOBUTTON          0x0000
```

```
#define LEFTBUTTON            0x0001
#define RIGHTBUTTON           0x0002
#define MIDBUTTON             0x0004
#define MOUSEBUTTONMASK 0x00FF

#define SHIFTBUTTON           0x0100
#define CONTROLBUTTON         0x0200
#define ALTBUTTON             0x0400
#define METABUTTON            0x0800
#define KEYBUTTONMASK         0x0FFF
#define KEYPAD                0x4000
```

（2）定义并初始化与输入事件相关的参数

```
static int mouse_fd = -1;     //鼠标文件描述符
static int kbd_fd = -1;       //键盘文件描述符
static POINT  mouse_pt;    //鼠标位置
static int mouse_buttons;    //鼠标按键
static struct QVFbKeyData kbd_data;      //键盘数据
static unsigned char kbd_state [NR_KEYS]; //   键盘状态
static unsigned char keycode_scancode [256];//  键盘扫描码
static unsigned char nr_changed_keys = 0;
```

（3） **static void init_code_map (void)**

函数作用：初始化键盘扫描码

（4） **static unsigned char keycode_to_scancode (unsigned char keycode, BOOL asscii)**

函数作用：将键盘扫描码转化为 ASCII 码

（5） **static int mouse_update (void)**

函数作用：更新鼠标信息 mouse_pt 和 mouse_buttons

（6） **static void mouse_getxy (int *x, int* y)**

函数作用：获得鼠标的位置信息

```
*x = mouse_pt. x;
*y = mouse_pt. y;
```

（7） **static int mouse_getbutton (void)**

函数作用：获取鼠标的 button 信息

（8） **static int keyboard_update (void)**

函数作用：更新鼠标信息，返回 nr_changed_keys;

（9） **static int read_key (void)**

函数作用：读取键盘按键

（10） **static const char* keyboard_getstate (void)**

函数作用：获取键盘状态， **return** (**char***)kbd_state;

（11）staticint wait_event (int which,int maxfd,fd_set *in, fd_set *out, fd_set *except, struct timeval*timeout)

函数作用：从文件描述符获得输入事件，返回输入事件的类型

（12）BOOLInitQVFBInput (INPUT* input, constchar* mdev, constchar* mtype)

函数作用： 初始化输入引擎， 设置键盘和鼠标的文件描述符， 为确定输入引擎的对输入事件的处理函数。

（13）void TermQVFBInput(void)

函数作用：关闭鼠标和键盘描述符。

```
    if (mouse_fd >= 0)
        close(mouse_fd);
    if (kbd_fd >= 0)
        close(kbd_fd);
```

9、window.h 与窗口和消息相关的宏定义和数据结构

（1）宏定义各种消息类型

（2）消息的数据结构 MSG

```
typedef struct _MSG
{
    /** The handle to the window which receives this message. */
    HWND            hwnd;
    /** The message identifier. */
    int             message;
    /** The first parameter of the message (32-bit integer). */
    WPARAM          wParam;
    /** The second parameter of the message (32-bit integer). */
    LPARAM          lParam;
    /** Time*/
    unsigned int    time;
#ifndef _LITE_VERSION
    /** Addtional data*/
    void*           pAdd;
#endif
} MSG;
typedef MSG* PMSG;
```

（3）宏定义消息队列的相关参数

```
#define QS_NOTIFYMSG        0x10000000
#ifndef _LITE_VERSION
  #define QS_SYNCMSG        0x20000000
#else
  #define QS_DESKTIMER      0x20000000
#endif
```

```
#define QS_POSTMSG          0x40000000
#define QS_QUIT             0x80000000
#define QS_INPUT            0x01000000
#define QS_PAINT            0x02000000
#define QS_TIMER            0x0000FFFF
#define QS_EMPTY            0x00000000
```

（**4**）获得消息时用到的宏定义

```
#define PM_NOREMOVE     0x0000
#define PM_REMOVE       0x0001
#define PM_NOYIELD      0x0002
```

（**5**）声明各种消息处理函数，如获得消息、等待消息等

（**6**）宏定义键盘分布的相关参数和键盘信息处理函数

```
#define KBD_LAYOUT_DEFAULT      "default"
#define KBD_LAYOUT_FRPC         "frpc"
#define KBD_LAYOUT_FR           "fr"
#define KBD_LAYOUT_DE           "de"
#define KBD_LAYOUT_DELATIN1     "delatin1"
#define KBD_LAYOUT_IT           "it"
#define KBD_LAYOUT_ES           "es"
#define KBD_LAYOUT_ESCP850      "escp850"
```

（**7**）宏定义 **hook** 相关的参数和函数

（**8**）宏定义窗口相关参数

（**9**）定义主窗口数据结构

```
typedef struct _MAINWINCREATE
{
    /** The style of the main window */
    DWORD  dwStyle ;
    /** The extended style of the main window */
    DWORD  dwExStyle ;
    /** The caption of the main window */
    const char*  spCaption ;
    /** The handle to the menu of the main window */
    HMENU  hMenu ;
    /** The handle to the cursor of the main window */
    HCURSOR  hCursor;
    /** The handle to the icon of the main window */
    HICON  hIcon ;
    /** The hosting main window */
    HWND     hHosting ;
    /** The window callback procedure */
    int (* MainWindowProc  )(HWND  , int, WPARAM  , LPARAM  );
    /** The position of the main window in the screen coordinates */
    int lx , ty, rx, by;
    /** The pixel value of background color of the main window */
    int iBkColor  ;
    /** The first private data associated with the main window */
    DWORD   dwAddData ;

    /** Reserved, do not use */
    DWORD   dwReserved;
} MAINWINCREATE    ;
typedef MAINWINCREATE    * PMAINWINCREATE    ;
```

（**10**）声明窗口创建、显示、获取窗口信息等等与窗口相关的函数

（**11**）宏定义主窗口的限制参数并声明相关函数

```
#define MWM_MINWIDTH                  0
#define MWM_MINHEIGHT                 1
#define MWM_BORDER                    2
#define MWM_THICKFRAME                3
。。。。。。
#define MWM_ITEM_NUMBER               31
```

（**12**）宏定义颜色信息并声明相关函数

```
#define BKC_CAPTION_NORMAL            0
#define FGC_CAPTION_NORMAL            1
#define BKC_CAPTION_ACTIVED           2
#define FGC_CAPTION_ACTIVED           3
。。。。。。
#define BKC_DESKTOP                   34
#define BKC_DIALOG                    35
#define BKC_TIP                       36
#define WEC_ITEM_NUMBER               37

/* back compitabilty defines */
#define BKC_BUTTON_DEF                WEC_3DBOX_NORMAL
#define BKC_BUTTON_PUSHED             WEC_3DBOX_DARK
#define FGC_BUTTON_NORMAL             WED_3DBOX_REVERSE
#define FGC_BUTTON_PUSHED             WED_3DBOX_REVERSE
#define BKC_EDIT_DEF                  WEC_3DBOX_LIGHT
#define BKC_EDIT_DISABLED             WEC_3DBOX_NORMAL
#define WEC_3DFRAME_LEFT_OUTER        WED_3DBOX_REVERSE
#define WEC_3DFRAME_LEFT_INNER        WEC_3DBOX_DARK
#define WEC_3DFRAME_TOP_OUTER         WED_3DBOX_REVERSE
#define WEC_3DFRAME_TOP_INNER         WEC_3DBOX_DARK
#define WEC_3DFRAME_RIGHT_OUTER       WEC_3DBOX_LIGHT
#define WEC_3DFRAME_RIGHT_INNER       WEC_3DBOX_NORMAL
```

```
#define WEC_3DFRAME_BOTTOM_OUTER            WEC_3DBOX_LIGHT
#define WEC_3DFRAME_BOTTOM_INNER            WEC_3DBOX_NORMAL
#define WEC_3DFRAME_LEFT                    WEC_3DBOX_LIGHT
#define WEC_3DFRAME_TOP                     WEC_3DBOX_LIGHT
#define WEC_3DFRAME_RIGHT                   WEC_3DBOX_DARK
#define WEC_3DFRAME_BOTTOM                  WEC_3DBOX_DARK
```

（**13**）宏定义位图相关参数并声明相关函数

（**14**）宏定义系统图标参数并声明相关函数

（**15**）滚动条相关的宏定义、数据结构和函数声明

（**16**）菜单相关的宏定义和函数声明

（**17**）对话框相关的宏定义、数据结构和函数声明

```
typedef struct _CTRLDATA
{
    /** Class name of the control */
    const char* class_name;
    /** Control style */
    DWORD           dwStyle ;
    /** Control position in dialog */
    int             x, y, w, h;
    /** Control identifier */
    int             id;
    /** Control caption */
    const char* caption;
    /** Additional data */
    DWORD           dwAddData ;
    /** Control extended style */
    DWORD           dwExStyle ;
} CTRLDATA ;
typedef CTRLDATA * PCTRLDATA ;

typedef struct _DLGTEMPLATE
{
    /** Dialog box style */
    DWORD           dwStyle ;
    /** Dialog box extended style */
    DWORD           dwExStyle ;
    /** Dialog box position */
    int             x, y, w, h;
    /** Dialog box caption */
    const char* caption;
    /** Dialog box icon */
    HICON           hIcon;
    /** Dialog box menu */
    HMENU          hMenu;
    /** Number of controls */
    int            controlnr ;
    /** Poiter to control array */
    PCTRLDATA      controls;
    /** Addtional data, must be zero */
    DWORD          dwAddData ;
} DLGTEMPLATE  ;
typedef DLGTEMPLATE  * PDLGTEMPLATE  ;
```

10、 internals.h 窗口、消息和消息队列相关的宏定义和数据结构及相关函数声明

（1）内部数据包括一些默认参数的宏定义

```
#define DEF_NR_TIMERS          32
  #define DEF_MSGQUEUE_LEN       8
  #define SIZE_CLIPRECTHEAP      16
  #define SIZE_INVRECTHEAP       16
  #define SIZE_QMSG_HEAP         8
  #define MAX_LEN_FIXSTR         64
  #define NR_HEAP                5
  #define LEN_BITMAP             (1+2+4+8+16)
```

（2）句柄类型和子类型的宏定义

```
#define TYPE_HWND              0x01
    #define TYPE_MAINWIN       0x11
    #define TYPE_CONTROL       0x12
    #define TYPE_ROOTWIN       0x13
#define TYPE_HMENU             0x02
    #define TYPE_MENUBAR       0x21
    #define TYPE_PPPMENU       0x22
    #define TYPE_NMLMENU       0x23
#define TYPE_HACCEL            0x03
#define TYPE_HCURSOR           0x05
#define TYPE_HICON             0x07
#define TYPE_HDC               0x08
    #define TYPE_SCRDC         0x81
    #define TYPE_GENDC         0x82
    #define TYPE_MEMDC         0x83

#define TYPE_WINTODEL          0xF1
#define TYPE_UNDEFINED         0xFF
```

（3）Z 序相关的数据结构

```
typedef struct _ZORDERNODE
{
     HWND   hWnd;                          /* Handle of window */
     struct _ZORDERNODE*    pNext;         /* Next window */
} ZORDERNODE ;
typedef ZORDERNODE  * PZORDERNODE ;

typedef struct _ZORDERINFO
{
    int nNumber;                           /* Number of windows */
    HWND  hWnd ;                           /* Handle of host window */
    PZORDERNODE   pTopMost ;               /* the top most Z order node */
}  ZORDERINFO ;
typedef ZORDERINFO  * PZORDERINFO ;
```

（4）滚动条和滚动窗口相关的数据结构

```
typedef struct _SCROLLWINDOWINFO
{
    int iOffx ;
    int iOffy ;
    const RECT *  rc1;
    const RECT *  rc2;
} SCROLLWINDOWINFO    ;
typedef SCROLLWINDOWINFO    * PSCROLLWINDOWINFO    ;

#define SBS_NORMAL           0x00
#define SBS_DISABLED         0x01
#define SBS_HIDE             0x02
typedef struct _SCROLLBARINFO {
    int    minPos;          // min value of scroll range.
    int    maxPos;          // max value of scroll range.
    int    curPos;          // current scroll pos.
    int    pageStep;        // steps per page.
    int    barStart;        // start pixel of bar.
    int    barLen;          // length of bar.
    BYTE   status;          // status of scroll bar.
} SCROLLBARINFO   ;
typedef SCROLLBARINFO   * PSCROLLBARINFO   ;
```

（5）符号信息相关的数据结构

```
typedef struct _CARETINFO {
    int        x;                 // position of caret
    int        y;
    void*      pNormal ;          // normal bitmap.
```

```
        void*      pXored ;              // bit-Xored bitmap.
        PBITMAP   pBitmap ;              // user defined caret bitmap.
        int        nWidth ;              // original size of caret
        int        nHeight ;
        int        nBytesNr ;            // number of bitmap bytes.
        BITMAP     caret_bmp;            // bitmap of caret.
        BOOL       fBlink ;              // does blink?
        BOOL       fShow;                // show or hide currently.
        HWND       hOwner ;              // the window owns the caret.
        UINT       uTime ;               // the blink time.
} CARETINFO ;
typedef CARETINFO * PCARETINFO ;
```

（6）消息和消息队列相关的数据结构及函数声明

```
typedef struct _QMSG
{
        MSG                      Msg;
        struct _QMSG*            next;
} QMSG ;
typedef QMSG * PQMSG ;
typedef struct _MSGQUEUE MSGQUEUE ;
typedef MSGQUEUE * PMSGQUEUE ;
typedef BOOL (* IDLEHANDLER ) (PMSGQUEUE msg_que);
struct _MSGQUEUE
{
        DWORD  dwState;                   // message queue states
        PQMSG  pFirstNotifyMsg ;          // head of the notify message queue
        PQMSG  pLastNotifyMsg ;           // tail of the notify message queue
        IDLEHANDLER  OnIdle ;             // Idle handler
        MSG * msg;                        /* post message buffer */
        int len;                          /* buffer len */
        int readpos, writepos ;           /* positions for reading and writing */
        int FirstTimerSlot ;              /* the first timer slot to be checked */
        DWORD  TimerMask ;                /* timer slots mask */
        int loop_depth;                   /* message loop depth, for dialog boxes. */
};
BOOL InitFreeQMSGList (void);
void DestroyFreeQMSGList (void);
BOOL InitMsgQueue (PMSGQUEUE pMsgQueue, int iBufferLen);
void DestroyMsgQueue (PMSGQUEUE pMsgQueue);
BOOL QueueMessage (PMSGQUEUE msg_que, PMSG msg);
extern PMSGQUEUE __mg_dsk_msg_queue;
```

（7）主窗口相关的数据结构

```
typedef struct _MAINWIN
```

```
{
        /*
         * These fields are similiar with CONTROL struct.
         */
        short DataType ;          // the data type.
        short WinType ;            // the window type.
        int left , top;           // the position and size of main window.
        int right , bottom ;
        int cl , ct;                  // the position and size of client area.
        int cr , cb;
        DWORD  dwStyle ;           // the styles of main window.
        DWORD  dwExStyle ;       // the extended styles of main window.
        int iBkColor ;              // the background color.
        HMENU  hMenu ;               // handle of menu.
        HACCEL  hAccel ;           // handle of accelerator table.
        HCURSOR  hCursor;        // handle of cursor.
        HICON  hIcon ;               // handle of icon.
        HMENU  hSysMenu ;          // handle of system menu.
        PLOGFONT  pLogFont ;   // pointer to logical font.
        HDC      privCDC ;           // the private client DC.
        INVRGN  InvRgn ;            // the invalid region of this main window.
        PGCRINFO  pGCRInfo ;   // pointer to global clip region info struct.
        PZORDERNODE  pZOrderNode ;
        PCARETINFO  pCaretInfo ;// pointer to system caret info struct.
        DWORD  dwAddData ;        // the additional data.
        DWORD  dwAddData2 ;      // the second addtional data.
        int (* MainWindowProc  )( HWND , int , WPARAM  , LPARAM  );
                                            // the address of main window procedure.
        char * spCaption ;        // the caption of main window.
        int     id;                      // the identifier of main window.
        SCROLLBARINFO   vscroll ;// the vertical scroll bar information.
        SCROLLBARINFO   hscroll ;// the horizital scroll bar information.
        struct _MAINWIN*    pMainWin ;
                                       // the main window that contains this window.
                                       // for main window, always be itself.
        HWND  hParent;              // the parent of this window.
                                       // for main window, always be HWND_DESKTOP.
        /*
         * Child windows.
         */
        HWND  hFirstChild ;        // the handle of first child window.
        HWND  hActiveChild  ;    // the currently active child window.
        HWND  hOldUnderPointer ;    // the old child window under pointer.
        HWND  hPrimitive ;        // the premitive child of mouse event.
```

```
    NOTIFPROC  NotifProc ;      // the notification callback procedure.
    /*
     * window element data.
     */
    struct _wnd_element_data*  wed;
    /*
     * Main Window hosting.
     * The following members are only implemented for main window.
     */
    struct _MAINWIN*    pHosting ; // the hosting main window.
    struct _MAINWIN*    pFirstHosted ;// the first hosted main window.
    struct _MAINWIN*    pNextHosted ;// the next hosted main window.
    PMSGQUEUE  pMessages;
                              // the message queue.
    GCRINFO  GCRInfo ;
                              // the global clip region info struct.
                              // put here to avoid invoking malloc function.
} MAINWIN  ;
struct _MAINWIN;
typedef struct _MAINWIN*    PMAINWIN  ;
```

（8）窗口相关的初始化和终止函数声明

11、 message.c 消息与消息队列相关函数

（1） BOOL InitFreeQMSGList(void)

函数作用：初始化消息列表

（2） void DestroyFreeQMSGList(void)

函数作用：销毁消息列表

（3） inline static PQMSG QMSGAlloc(void)

函数作用：

（4） inline static void FreeQMSG(PQMSG pqmsg)

函数作用：

（5） BOOL InitMsgQueue(PMSGQUEUE pMsgQueue, int iBufferLen)

函数作用：初始化消息队列，包括消息队列的类型、长度、时间槽、时间掩码、并为消息队列分配空间。

（6） void DestroyMsgQueue(PMSGQUEUE pMsgQueue)

函数作用：释放为消息队列分配的存储空间，包括 notify 类型的消息队列和 message 消息队列。

（7） PMSGQUEUE GetMsgQueue(HWND hWnd)

函数作用：获得指向消息队列的指针。

（8） BOOL QueueMessage(PMSGQUEUE msg_que, PMSG msg)

函数作用：向消息队列 msg_que 发送消息 msg。

（9） static inline WNDPROC GetWndProc (HWND hWnd)

函数作用：获得窗口处理函数。 **return** ((PMAINWIN)hWnd)-> MainWindowProc 。

（10） static HWND msgCheckInvalidRegion (PMAINWIN pWin)

函数作用：检查无效区域。函数通过递归调用自身来判断无效区域。

（11） static HWND msgCheckHostedTree (PMAINWIN pHosting)

函数作用：检索主窗口树，返回区域无效的窗口，即返回需要重新绘制的窗口。

（12） BOOL GUIAPI HavePendingMessageEx (HWND hWnd, BOOL bNoDeskTimer)

函数作用：判断是否有挂起的消息，若有，返回 TRUE，若无，返回（ IdleHandler4StandAlone （NULL ））。第二个参数为 TRUE 时，不考虑 QS_DESKTIMER 的挂起状态，为 FALSE 时才将 QS_DESKTIMER 消息的挂起状态考虑在内。

（13） BOOL GUIAPI HavePendingMessage (HWND hWnd)

函数作用：同（ 12）。**return** HavePendingMessageEx (hWnd, FALSE) ；要将 QS_DESKTIMER 消息将的挂起状态考虑在内。

（14） int GUIAPI BroadcastMessage (int iMsg, WPARAM wParam, LPARAM lParam)

参数： **int** iMsg：消息类型， WPARAM wParam, LPARAM lParam ：消息参数。

函数作用： **return** SendMessage (HWND_DESKTOP, MSG_BROADCASTMSG, 0, (LPARAM)(&msg));

（15） static inline void CheckCapturedMouseMessage (PMSG pMsg)

函数作用：

（16） IS_MSG_WANTED(message)

```
#define IS_MSG_WANTED(message) \
         ( (iMsgFilterMin <= 0 && iMsgFilterMax <= 0) || \
           (iMsgFilterMin > 0 && iMsgFilterMax >= iMsgFilterMin && \
            message >= iMsgFilterMin && message <= iMsgFilterMax) )
```

作用：判断消息是否在可识别的消息范围内。

（17） BOOL PeekMessageEx (PMSG pMsg, HWND hWnd, int iMsgFilterMin, int iMsgFilterMax, BOOL bWait, UINT uRemoveMsg)

函数作用：从窗口 hWnd 的消息队列中获得消息 pMsg，并根据 uRemoveMsg 的值对消息队列进行处理。如果消息队列中没有消息，则：当 bWait 为 TRUE 时，进入空闲处理函数获得等待被读取的输入事件， 再次从消息队列中读取消息； 当 bWait 为 FALSE 时，函数执行结束， 返回 FALSE 。

（18） int getIdle()

函数作用： **return** idle。

（19） BOOL GUIAPI WaitMessage (PMSG pMsg, HWND hWnd)

函数作用：将 pMsg 指向的内存空间清零。若消息队列中有消息，则返回 TRUE，若没有消息，则进入空闲处理函数 pMsgQueue->OnIdle (pMsgQueue) ，继续判断队列中是否有消息。

（20） BOOL GUIAPI PeekPostMessage (PMSG pMsg, HWND hWnd, int iMsgFilterMin, int iMsgFilterMax, UINT uRemoveMsg)

函数作用：获得一个 QS_POSTMSG 类型的消息并根据 uRemoveMsg 参数对消息队列进行处理。

（21）int GUIAPI SendMessage(HWND hWnd, int iMsg, WPARAM wParam, LPARAM lParam)

函数作用：向句柄为 hWnd 的窗口发送 iMsg 消息和相关参数并对消息进行处理。

（1）首先获得窗口处理函数： WndProc = GetWndProc(hWnd)

（2）再用该函数对消息进行处理： **return** (*WndProc)(hWnd, iMsg, wParam, lParam)

（22）int GUIAPI SendNotifyMessage(HWND hWnd, int iMsg, WPARAM wParam, LPARAM lParam)

函数作用：将 iMsg 消息加入窗口 hWnd 的 notify 消息队列。

（23）int GUIAPI PostMessage(HWND hWnd, int iMsg, WPARAM wParam, LPARAM lParam)

函数作用：向 hWnd 的窗口队列发送一个识别码为 iMsg，参数为 wParam 和 lParam 的消息后立即返回。

（24）int GUIAPI PostQuitMessage(HWND hWnd)

函数作用：向主窗口的消息队列中加入一个退出消息， MSG_QUIT 。

（25）int GUIAPI DispatchMessage(PMSG pMsg)

函数作用：分发并处理消息。利用 pMsg 所在的窗口的窗口处理函数对消息进行处理并返回窗口处理函数的结果。

（26）int GUIAPI ThrowAwayMessages(HWND hWnd)

函数作用：删除消息队列中所有与 hWnd 相关的消息。

（27）BOOL GUIAPI EmptyMessageQueue(HWND hWnd)

函数作用：清空一个消息队列。

12、 ourhdr.h

（1）宏定义最大行数

```
#define MAXLINE                    4096          /* max line length */
```

（2）声明函数

```
void err_dump (const char *, ...);       /* {App misc_source} */
void err_msg(const char *, ...);
void err_quit (const char *, ...);
void err_ret (const char *, ...);
void err_sys(const char *, ...);
```

（3）定义监听文件描述符的数据结构

```
typedef struct listen_fd {
    int fd;   //文件描述符标识符
    int hwnd;   //句柄
    int type;    //类型
    void* context;   //
} LISTEN_FD ;
```

```
#define mgfd_set    int
extern mgfd_set          mg_rfdset;
extern mgfd_set*         mg_wfdset;
extern mgfd_set*         mg_efdset;
extern int               mg_maxfd;
extern LISTEN_FD         mg_listen_fds [];
```

13、 listenfd.c

（1）定义文件描述符的相关变量

```
LISTEN_FD   mg_listen_fds [MAX_NR_LISTEN_FD];
static mgfd_set _wfdset, _efdset;
mgfd_set mg_rfdset;
mgfd_set* mg_wfdset = NULL;
mgfd_set* mg_efdset = NULL;
int mg_maxfd;
```

（2） BOOL GUIAPI RegisterListenFD (int fd, int type, HWND hwnd, void* context)

函数作用：注册文件描述符

```
BOOL GUIAPI RegisterListenFD (int fd, int type, HWND hwnd, void* context)
{
    return TRUE;
}
```

（3） BOOL GUIAPI RegisterListenFD (int fd, int type, HWND hwnd, void* context)

函数作用：注销文件描述符

```
BOOL GUIAPI UnregisterListenFD (int fd)
{
   return TRUE;
}
```

14、 standalone.c

（1） SRVEVTHOOK GUIAPI SetServerEventHook (SRVEVTHOOK SrvEvtHook)

\brief Sets an event hook in the server of MiniGUI-Processes.

（2） static void ParseEvent (PMSGQUEUE msg_que, int event)

函数作用：解析事件，将 IAL 类型的事件最终解析为对应的消息并加入消息队列中。

（3） BOOL GUIAPI StandAloneStartup (void)

函数作用： mg_maxfd = 0;

```
#ifndef _NEWGAL_ENGINE_BF533
    InstallIntervalTimer ();
#endif
```

（4） void StandAloneCleanup (void)

函数作用：

```
#ifndef _NEWGAL_ENGINE_BF533
```

```
    UninstallIntervalTimer ();
#endif
```

（5）BOOL minigui_idle (void)

函数作用： **return** IdleHandler4StandAlone (__mg_dsk_msg_queue);

（6）BOOL IdleHandler4StandAlone (PMSGQUEUE msg_queue)

函数作用： standalone 模式下的空闲处理函数

（1）获得文件描述符中输入事件类型： n = IAL_WaitEvent (IAL_MOUSEEVENT | IAL_KEYEVENT,mg_maxfd, &rset, wsetptr, esetptr, NULL);

（2）对输入事件进行解析：

```
    if (n & IAL_MOUSEEVENT) ParseEvent (msg_queue, IAL_MOUSEEVENT);
    if (n & IAL_KEYEVENT) ParseEvent (msg_queue, IAL_KEYEVENT);
    if (n == 0) ParseEvent (msg_queue, 0)
```

（3）函数返回值为 TRUE，表示有输入事件，返回值为 FALSE，表示超时或者有错误发生。

15、keyboard.h

（1）宏定义

```
#define VC_XLATE          0x0000    /* translate keycodes using keymap */
#define VC_MEDIUMRAW      0x0001    /* medium raw (keycode) mode */
#define VC_RAW            0x0002    /* raw (scancode) mode */
#define VC_UNICODE        0x0004    /* Unicode mode */
#define VC_APPLIC         0x0010    /* application key mode */
#define VC_CKMODE         0x0020    /* cursor key mode */
#define VC_REPEAT         0x0040    /* keyboard repeat */
#define VC_CRLF           0x0080    /* 0 - enter sends CR, 1 - enter sends CRLF */
#define VC_META           0x0100    /* 0 - meta, 1 - meta=prefix with ESC */
```

（2）定义键盘信息数据结构

```
typedef struct _key_info
{
    DWORD kbd_mode;
    DWORD shiftstate;
    DWORD oldstate;
    int npadch;
    unsigned char diacr;
    int dead_key_next;
    unsigned char type;
    unsigned char buff [50];
    int   pos;
} key_info;
```

（3）键盘分布信息数据结构及初始化数据结构：

```
typedef void (* INIT_KBD_LAYOUT ) (ushort*** key_maps_p, struct kbdiacr** accent_table_p,
                unsigned int* accent_table_size_p, char*** func_table_p);
typedef struct kbd_layout_info
```

```
{
        char* name;
        INIT_KBD_LAYOUT     init;
} kbd_layout_info ;
```

（4）声明初始化默认键盘分布

16、keyboard.c(待分析。。。)

（1）static inline void put_queue (char ch, key_info* kinfo)

函数作用：将字符放入键盘缓存中。

```
    kinfo-> buff [kinfo-> pos] = ch;
    kinfo-> pos ++;
```

（2）static inline void puts_queue (char* cp, key_info* kinfo)

函数作用：将以 cp为首地址的一串字符放入缓存中。

（3）static void applkey (int key, char mode, key_info* kinfo)

（4）static void to_utf8 (ushort c, key_info* kinfo)

函数作用：将字符经过转化后放入缓存中。

（5）宏定义连接符

```
#define A_GRAVE     '`'
#define A_ACUTE     '\''
#define A_CFLEX     '^'
#define A_TILDE     '~'
#define A_DIAER     '"'
#define A_CEDIL     ','
```

（6）定义连接符数组

```
static unsigned char ret_diacr[NR_DEAD] =       {A_GRAVE, A_ACUTE, A_CFLEX, A_TILDE,
A_DIAER, A_CEDIL };
```

（7）static unsigned char handle_diacr (unsigned char ch, key_info* kinfo)

函数作用：对连接符进行处理。

二、数据结构解析

1、鼠标移动事件数据结构

```
typedef struct _MOUSEEVENT {
        int event;    //事件标识符，代表不同的鼠标事件     ME_*
        int x;          //鼠标的位置
        int y;
        DWORD  status;   //状态信息
} MOUSEEVENT ;
typedef MOUSEEVENT  * PMOUSEEVENT ;
```

2、键盘事件的数据结构

```
typedef struct _KEYEVENT {
    int event;    //事件标识符，代表不同的键盘事件    KE_*
    int scancode; //按键扫描码
    DWORD  status; //状态信息
} KEYEVENT  ;
typedef KEYEVENT  * PKEYEVENT  ;
```

3、底层事件的数据结构

（1）底层事件的数据：

```
typedef union _LWEVENTDATA {
    MOUSEEVENT   me;
    KEYEVENT   ke;
} LWEVENTDATA   ;
```

（2）底层事件：

```
typedef struct _LWEVENT
{
    int type;   //底层事件标识符，    KE_*| ME_*
    int count; //发生时间
    DWORD  status;   //状态
    LWEVENTDATA    data; //数据，当 event为KE_* 时， data为 KEYEVENT  ，当 event为ME_* 时，
                                  //data为 MOUSEEVENT
} LWEVENT  ;
typedef LWEVENT  * PLWEVENT;
```

4、输入引擎相关的数据结构：

（1）输入引擎处理函数：

```
typedef struct tagINPUT
{
    char*    id;   //标识符，输入引擎的名称

    //  输入初始化和终止函数
    BOOL  (* init_input  ) (struct tagINPUT *input,   const char * mdev,  const char * mtype);
    void (* term_input ) (void);

    //  鼠标操作
    int    (* update_mouse) (void);
    void (* get_mouse_xy ) (int* x,  int * y);
    void (* set_mouse_xy) (int x,  int  y);
    int    (* get_mouse_button ) (void);
    void (* set_mouse_range) (int  minx,  int miny,  int  maxx,  int  maxy);
    void (* suspend_mouse) (void);
    int (* resume_mouse) (void);

    // 键盘操作
    int  (* update_keyboard ) (void);
    const char* (* get_keyboard_state ) (void);
    void (* suspend_keyboard) (void);
    int (* resume_keyboard ) (void);
    void (* set_leds) (unsigned int leds);
    int (* wait_event ) (int which, int maxfd, fd_set *in, fd_set *out, fd_set *except, struct timeval
*timeout);
    char mdev [MAX_PATH + 1];
} INPUT ;
```

（2）输入引擎的键盘数据结构：

```
struct QVFbKeyData
{
    unsigned int unicode ;
    unsigned int modifiers ;
    BYTE  press;
    BYTE  repeat;
};
```

5、消息相关数据结构：

（1）消息数据结构：

```
typedef struct _MSG
{
    HWND                hwnd;   //消息所属的窗口句柄
    int              message;   //消息标识符  MS_*
    WPARAM                wParam; //消息的第一个参数,    32位整型
    LPARAM              lParam;// 消息的第二个参数,    32位整型
    unsigned int       time;// 消息发生时间
#ifndef  _LITE_VERSION
    void*               pAdd;   // 附加数据
#endif
}  MSG;
typedef MSG *  PMSG ;
```

（2）消息链表数据结构：

```
typedef struct _QMSG
{
    MSG                    Msg;  // 当前消息
    struct _QMSG*          next;    // 下一条消息
} QMSG ;
typedef QMSG *  PQMSG ;
```

（3）消息队列数据结构

```
struct _MSGQUEUE
{
    DWORD  dwState;                 // message queue states, 消息队列的状态,   包含的消息类型
    PQMSG   pFirstNotifyMsg  ;      // notify 消息队列的第一个消息
    PQMSG   pLastNotifyMsg  ;       // notify 消息队列的最后一个消息
    IDLEHANDLER   OnIdle ;          //  空闲处理函数
    MSG * msg;                      // post  类型的消息的缓存地址
    int len;                        //缓存长度
    int readpos, writepos ;         //读写位置
    int FirstTimerSlot  ;           /* the first timer slot to be checked */
    DWORD  TimerMask ;              /* timer slots mask */
    int loop_depth ;                /* message loop depth, for dialog boxes. */
};
typedef struct _MSGQUEUE   MSGQUEUE  ;
typedef MSGQUEUE  *  PMSGQUEUE  ;
typedef BOOL  (*  IDLEHANDLER   ) (PMSGQUEUE   msg_que);
```

6、定义主窗口数据结构

（1）创建主窗口时需要的数据结构

```
typedef struct _MAINWINCREATE
{
    DWORD  dwStyle ;   //主窗口的类型
    DWORD  dwExStyle ;   //主窗口的扩展类型
    const char* spCaption ; //主窗口的标题
    HMENU  hMenu ;//主窗口的菜单句柄
    HCURSOR  hCursor; //主窗口的光标句柄
    HICON  hIcon ; //主窗口的图标句柄
    HWND     hHosting ;// 托管窗口
    int (* MainWindowProc  )(HWND , int, WPARAM  , LPARAM  );//窗口处理函数
    int lx , ty, rx, by;//窗口位置大小
    int iBkColor  ;//窗口颜色
    DWORD  dwAddData ;// 窗口附加数据
    DWORD  dwReserved;// 保留项
} MAINWINCREATE   ;
typedef MAINWINCREATE   *  PMAINWINCREATE   ;
```

（2）主窗口数据结构：

```
typedef struct _MAINWIN
{
    /*
     * These fields are similiar with CONTROL struct.
     */
```

```
short DataType ;          //  数据类型
short WinType ;           //  窗口类型
int left , top;           //  窗口位置大小
int right , bottom ;
int cl , ct;              //  客户区域的位置大小
int cr , cb;
DWORD  dwStyle ;          //  主窗口风格
DWORD  dwExStyle ;        //主窗口扩展风格
int iBkColor ;            //  背景色
HMENU  hMenu ;            //菜单句柄
HACCEL  hAccel ;          //  加速表句柄
HCURSOR  hCursor;         //光标句柄
HICON  hIcon ;            //  图标句柄
HMENU  hSysMenu ;         //  系统菜单句柄
PLOGFONT  pLogFont ;  //  指向逻辑字体的指针
HDC      privCDC ;        // the private client DC.
INVRGN  InvRgn ;          //  主窗口无效客户区域
PGCRINFO  pGCRInfo ;  // pointer to global clip region info struct.
PZORDERNODE  pZOrderNode ;
PCARETINFO  pCaretInfo ;// pointer to system caret info struct.
DWORD  dwAddData ;        //  附加数据
DWORD  dwAddData2 ;       //  附加数据
int (* MainWindowProc )(HWND , int, WPARAM , LPARAM );//  窗口处理函数
char* spCaption ;         //主窗口标题
int     id;               //  窗口标识符
SCROLLBARINFO   vscroll ;//  垂直滚动条信息
SCROLLBARINFO   hscroll ;//  水平滚动条信息
struct _MAINWIN*   pMainWin ; // 包含该窗口的主窗口，经常是自身
HWND  hParent;            //父窗口，主窗口的父窗口一般是      HWND_DESKTOP
/*
 * Child windows.
 */
HWND  hFirstChild ;       //  第一个子窗口句柄
HWND  hActiveChild  ;  //  当前处于活跃状态的子窗口
HWND  hOldUnderPointer ;   // the old child window under pointer.
HWND  hPrimitive ;        // the premitive child of mouse event.
NOTIFPROC  NotifProc ;     // the notification callback procedure.
/*
 * window element data.
 */
struct _wnd_element_data*  wed;
/*
 * Main Window hosting.
 * The following members are only implemented for main window.
```

```
     */
    struct _MAINWIN*    pHosting ; // the hosting main window.
    struct _MAINWIN*    pFirstHosted ;// the first hosted main window.
    struct _MAINWIN*    pNextHosted ;// the next hosted main window.
    PMSGQUEUE  pMessages; //  消息队列
    GCRINFO  GCRInfo ;
                                   // the global clip region info struct.
                                   // put here to avoid invoking malloc function.
} MAINWIN  ;
struct _MAINWIN;
typedef struct _MAINWIN*    PMAINWIN  ;
```

三、重要函数

1、底层事件初始化 InitLWEvent （）：

```
BOOL  InitLWEvent  (void)
{
    GetDblclickTime ();      //获得鼠标双击时间间隔
    GetTimeout ();       //获得超时时间
    if (InitIAL ())      //初始化输入抽象层
        return  FALSE;
    ResetMouseEvent();    //重置鼠标事件
    ResetKeyEvent();      //重置键盘事件
    return  TRUE;
}
```

2、初始化输入抽象层 InitIAL () ：

```
int InitIAL  (void)
{
    int    i;
    char engine [LEN_ENGINE_NAME + 1];       //输入引擎名称
    char mdev [MAX_PATH + 1];                //路径
    char mtype[LEN_MTYPE_NAME + 1];          //类型

    if (NR_INPUTS == 0)      //如果没有输入引擎，返回错误信息
        return  ERR_NO_ENGINE;
    //将system的ial_engine 段的值复制给  engine指向的地址
    if (GetMgEtcValue (  "system" , "ial_engine"  , engine, LEN_ENGINE_NAME) < 0)
        return  ERR_CONFIG_FILE;
    if (GetMgEtcValue (  "system" , "mdev" , mdev, MAX_PATH) < 0)
        return  ERR_CONFIG_FILE;
    if (GetMgEtcValue (  "system" , "mtype"  , mtype, LEN_MTYPE_NAME) < 0)
        return  ERR_CONFIG_FILE;
//找到与 engine相等的字符串，由此来确定当前的输入引擎        __mg_cur_input
```

```
    for (i = 0; i < NR_INPUTS; i++) {
        if (strncmp (engine, inputs[i]. id, LEN_ENGINE_NAME) == 0) {
            __mg_cur_input = inputs + i;
            break;
        }
    }
   //如果当前输入引擎为空
    if (__mg_cur_input == NULL) {
        fprintf (stderr, "IAL: Does not find the request engine: %s.\n"    , engine);
        if (NR_INPUTS) {      //输入引擎数组不为空
            __mg_cur_input = inputs;//  当前输入引擎为输入引擎数组的第一个
            fprintf (stderr, "IAL: Use the first engine: %s\n"    , __mg_cur_input-> id);
        }
        else
            return ERR_NO_MATCH;
    }
//将mdev存储的路径字符串复制给     __mg_cur_input-> mdev
    strcpy (__mg_cur_input-> mdev, mdev);
//根据输入引擎选择对应的输入初始化函数对输入进行初始化
    if (!IAL_InitInput (__mg_cur_input, mdev, mtype)) {
        fprintf (stderr, "IAL: Init IAL engine failure.\n"      );
        return ERR_INPUT_ENGINE;
    }
    return 0;
}
```

3、standalone 模式下的空闲操作函数：**IdleHandler4StandAlone**

函数作用：将输入事件转换成对应的消息并放入消息队列中

```
BOOL IdleHandler4StandAlone (PMSGQUEUE msg_queue)
{
    int      i, n;
    int rset, wset, eset;
    int * wsetptr = NULL;
    int * esetptr = NULL;

    if (old_timer_counter != __mg_timer_counter) {
        old_timer_counter = __mg_timer_counter;
        SetDesktopTimerFlag ();
    }

    rset = mg_rfdset;               /* rset gets modified each time around */
    if (mg_wfdset) {
        wset = *mg_wfdset;
        wsetptr = &wset;
    }
    if (mg_efdset) {
        eset = *mg_efdset;
        esetptr = &eset;
    }
    n = IAL_WaitEvent (IAL_MOUSEEVENT | IAL_KEYEVENT,
                        mg_maxfd, &rset, wsetptr, esetptr,
                        NULL);

    if (msg_queue == NULL) msg_queue = __mg_dsk_msg_queue;

     if (n < 0) {//  表示等待事件函数   IAL_WaitEvent   中出错

         /* It is time to check event again. */
         if (errno == EINTR) {//   出错类型为中断，则解析消息队列
             //if (msg_queue)
                 ParseEvent (msg_queue, 0);
         }
         return  FALSE;
     }
/*
     else if (msg_queue == NULL)
         return (n > 0);
*/



    /* handle intput event (mouse/touch-screen or keyboard) */
    //等待事件为鼠标事件，解析消息队列
    if (n & IAL_MOUSEEVENT) ParseEvent (msg_queue, IAL_MOUSEEVENT);
  //等待事件为键盘事件，解析消息队列
    if (n & IAL_KEYEVENT) ParseEvent (msg_queue, IAL_KEYEVENT);
    //等待事件为超时事件，解析消息队列
    if (n == 0) ParseEvent (msg_queue, 0);

    /* go through registered listen fds */
    for (i = 0; i < MAX_NR_LISTEN_FD; i++) {
        MSG  Msg;

        Msg.message = MSG_FDEVENT;

        if (mg_listen_fds [i].  fd ) {
            fd_set* temp = NULL;
            int type = mg_listen_fds [i].   type ;
```

```
                switch (type) {
                casePOLLIN:
                        temp = &rset;
                        break ;
                casePOLLOUT:
                        temp = wsetptr;
                        break ;
                casePOLLERR:
                        temp = esetptr;
                        break ;
                }
        }
    }
    return  (n > 0);
}
```

4、输入等待事件： IAL_WaitEvent

```
static int wait_event (int which,  int maxfd,  fd_set *in,  fd_set *out,  fd_set *except,
                        struct timeval *timeout)
{
    fprintf (stderr,"init qvfb event\n"   );
    fd_set rfds;
    int       retvalue = 0;
    int       fd, e;

    if (!in) {     //如果可读文件描述符集合为空
        in = &rfds;     //设置可读文件描述符指向     rfds
        FD_ZERO (in);//  将 rfds 对应的内存空间的文件描述符清空
    }
//如果事件类型为鼠标事件且鼠标的文件描述符          mouse_fd >= 0
    if (which & IAL_MOUSEEVENT && mouse_fd >= 0) {
        fd = mouse_fd;     // 文件描述符等于鼠标文件描述符
        FD_SET (fd, in); //  将文件描述符   fd加入到可读文件描述符中
#ifdef _LITE_VERSION //  如果为非多线程模式,
        if (fd > maxfd) maxfd = fd;       //如果文件描述符大于最大文件描述符大于         0,
#endif
    }
//如果事件类型为键盘事件且键盘的文件描述符          kbd_fd >= 0
    if (which & IAL_KEYEVENT && kbd_fd >= 0) {
        fd = kbd_fd; //  文件描述符等于鼠标文件描述符
        FD_SET (kbd_fd, in); //   将文件描述符   fd加入到可读文件描述符中
#ifdef _LITE_VERSION
        if (fd > maxfd) maxfd = fd; //    如果文件描述符大于最大文件描述符
#endif
```

```
    }
    //将处于就绪状态并且已经包含在 fd_set 结构中的描述符总数赋给 e
    e = select(maxfd + 1, in, out, except, timeout) ;
    if (e > 0) {// 如果 e大于 0
        fd = mouse_fd; // 设置文件描述符为鼠标描述符
        /* If data is present on the mouse fd, service it: */
        if (fd >= 0 && FD_ISSET (fd, in)) { // 如果 fd大于 0且位于可读文件描述符集中
            FD_CLR (fd, in); //从可读文件描述符集中删除该描述符
            retvalue |= IAL_MOUSEEVENT; //返回值设为包含 IAL_MOUSEEVENT
        }

        fd = kbd_fd; // 设置文件描述符为鼠标描述符
        /* If data is present on the keyboard fd, service it: */
        if (fd >= 0 && FD_ISSET (fd, in)) {// 如果 fd大于 0且位于可读文件描述符集中
            FD_CLR (fd, in); // 从可读文件描述符集中删除该描述符
            if (read_key ()) //读取键盘按键信息成功
                retvalue |= IAL_KEYEVENT; // 返回值设为包含 IAL_KEYEVENT
            else{ /* play at a timeout event */ //读取键盘按键信息失败
                if (timeout) { //timeout 不为 0
                    timeout-> tv_sec = 0; //设置 timeout 为0
                    timeout-> tv_usec = 0;
                }
            }
        }

    } else if (e < 0) {// 如果 e<0 ， 表示出现错误，返回 -1
        return -1;
    }
    return retvalue;// 返回 retvalue
}
```

5、读取键盘按键信息

```
static int read_key (void)
{
    static unsigned char last;
    struct QVFbKeyData l_kbd_data;
    int ret;
    unsigned char scancode;

    ret = read (kbd_fd, &l_kbd_data, sizeof (struct QVFbKeyData));

    if (ret == sizeof(struct QVFbKeyData)) {
        kbd_data = l_kbd_data;
    }
    else
        return 0;

    if (kbd_data.repeat) {
        return 0;
    }

    if (kbd_data.unicode == 0 && !kbd_data.press) {
        kbd_state [last] = 0;
    }
    else{
//          scancode = keycode_to_scancode (HIWORD (kbd_data.unicode) & 0x00FF,
    //                          LOWORD (kbd_data.unicode));
    scancode = keycode_to_scancode (LOWORD (kbd_data.unicode),HIWORD (kbd_data.unicode )
& 0x00FF);
        kbd_state [scancode] = kbd_data.press ? 1 : 0;
        last = scancode;
    }

    nr_changed_keys = last + 1;
    return 1;
}
```

6 、解析事件 ParseEvent

函数作用：将 **event** 时间转换成 MSG 类型的事件并加入到消息队列中

```
static void ParseEvent (PMSGQUEUE msg_que, int event)
{
    LWEVENT lwe;
    PMOUSEEVENT me;
    PKEYEVENT ke;
    MSG Msg;

    ke = &(lwe.data.ke);
    me = &(lwe.data.me);
    me->x = 0; me-> y = 0;
    Msg.hwnd = HWND_DESKTOP;
    Msg.wParam = 0;
    Msg.lParam = 0;

    lwe.status = 0L;
//根据 event事件获得底层事件 lwe 的值
    if (!GetLWEvent (event, &lwe))
        return ;
```

```
Msg. time = __mg_timer_counter;
//若底层事件为超时事件
if (lwe. type == LWETYPE_TIMEOUT) {
    Msg. message = MSG_TIMEOUT;//   消息类型为   MSG_TIMEOUT
    Msg. wParam = ( WPARAM  )lwe. count;
    Msg. lParam = 0;

    QueueMessage (msg_que, &Msg);//  将消息加入消息队列
}
elseif (lwe. type == LWETYPE_KEY) {//    若底层事件为键盘事件
    Msg. wParam = ke-> scancode;// 消息的 wParam成员值为键盘事件的扫描码成员
    Msg. lParam = ke-> status;// 消息的 wParam成员值为键盘事件的状态成员
    if (ke-> event == KE_KEYDOWN){//    如果键盘事件为击键事件
        Msg. message = MSG_KEYDOWN;//   消息类型为   MSG_KEYDOWN
    }
    else if (ke-> event == KE_KEYUP) {//    如果键盘事件为按键释放事件
        Msg. message = MSG_KEYUP;//   消息类型为   MSG_KEYUP
    }
    else if (ke-> event == KE_KEYLONGPRESS) {
        Msg. message = MSG_KEYLONGPRESS;
    }
    else if (ke-> event == KE_KEYALWAYSPRESS) {
        Msg. message = MSG_KEYALWAYSPRESS;
    }

    if (!(srv_evt_hook && srv_evt_hook (&Msg))) {
        QueueMessage (msg_que, &Msg);
    }
}
elseif (lwe. type == LWETYPE_MOUSE) {//    若底层事件为鼠标事件
    Msg. wParam = me-> status;// 消息的 wParam参数为鼠标的状态信息
    switch (me-> event) {// 判断鼠标事件的类型，并依此确定消息的类型
    case ME_MOVED:        //鼠标移动事件
        Msg. message = MSG_MOUSEMOVE;
        SetCursor (GetSystemCursor (IDC_ARROW));
        break ;
    case ME_LEFTDOWN:
        Msg. message = MSG_LBUTTONDOWN;
        break ;
    case ME_LEFTUP:
        Msg. message = MSG_LBUTTONUP;
        break ;
    case ME_LEFTDBLCLICK:
        Msg. message = MSG_LBUTTONDBLCLK;
```

```
            break;
        case ME_RIGHTDOWN:
            Msg.message = MSG_RBUTTONDOWN;
            break;
        case ME_RIGHTUP:
            Msg.message = MSG_RBUTTONUP;
            break;
        case ME_RIGHTDBLCLICK:
            Msg.message = MSG_RBUTTONDBLCLK;
            break;
        }

        Msg.lParam = MAKELONG (me-> x, me-> y);// 将鼠标信息转换为消息的 lParam
        if (!(srv_evt_hook && srv_evt_hook (&Msg))) {
            QueueMessage (msg_que, &Msg);// 将消息加入消息队列
        }
    }
}
```

7、获取底层事件： GetLWEvent

根据 event 的值确定 lwe 的成员值

```
BOOL GetLWEvent (int event, PLWEVENT lwe)
{
    static LWEVENT old_lwe = {0, 0};
    unsigned int interval;
    int button;
    PMOUSEEVENT me = &(lwe-> data.me);
    PKEYEVENT ke = &(lwe-> data.ke);
    const char* keystate;
    int i;
    int make;          /* 0 = release, 1 = presse */
//如果事件类型为 0, 表示超时事件或者错误事件
    if (event == 0) {
  /*#define DEF_USEC_TIMEOUT                    300000
   timeoutusec = DEF_USEC_TIMEOUT      |     timeoutusec = mytimeoutusec;
   timeout_threshold = timeoutusec / 10000;
   timeout_count = timeout_threshold;

   #define DEF_REPEAT_TIME                    50000
   repeatusec = DEF_REPEAT_TIME;     | repeatusec = myrepeatusec
   repeat_threshold = repeatusec / 10000;
   */
   // 如果超时
        if (__mg_timer_counter >= timeout_count) {
                timeout_count = __mg_timer_counter + repeat_threshold;

                // repeat last event
                if (old_lwe. type == LWETYPE_KEY        //如果旧底层事件类型为     LWETYPE_KEY
                        && old_lwe.   data.ke.event == KE_KEYDOWN) {//      且为 KEYDOWN   类型
                    memcpy (lwe, &old_lwe,    sizeof(LWEVENT   )); // 将旧事件复制给    lwe
                    lwe-> data.ke.status |= KS_REPEATED;//   将 lwe 的键盘状态设置为重复
                    return  1;
                }

                if (!(old_lwe.  type == LWETYPE_MOUSE       //如果旧底层事件类型为鼠标事件
                        && (old_lwe.   data.me.event == ME_LEFTDOWN ||
                            old_lwe. data.me.event == ME_RIGHTDOWN ||
                            old_lwe. data.me.event == ME_MIDDLEDOWN))) {
                     //且为鼠标左键或右键或者中间键的击键事件
                    // reset delay time
                    timeout_count = __mg_timer_counter + timeout_threshold;
                }

                // reset delay time
                lwe->type  = LWETYPE_TIMEOUT;     设置事件类型为超时事件
                lwe->count  = __mg_timer_counter;//  计时器等与系统计时器事件
                return  1;
            }
            return  0; // 如果是错误事件返回     0
        }
//event不等于  0，此时表示有事件发生
        timeout_count = __mg_timer_counter + timeout_threshold;
        // There was a event occurred.
        if (event & IAL_MOUSEEVENT) { //      如果事件类型包含    IAL_MOUSEEVENT
            if (!IAL_UpdateMouse ())//    更新鼠标信息
                return  0;

            lwe-> type = LWETYPE_MOUSE;//    设置底层事件类型为     LWETYPE_MOUSE
            if (RefreshCursor(&me->  x, &me-> y, &button)) {//   刷新光标信息
                me->event  = ME_MOVED;//   鼠标事件为   ME_MOVED
                time1 = 0;
                time2 = 0;

                if (oldbutton == button)      //如果前一个鼠标事件的类型与当前鼠标事件的类型相同
                    return  1;        //返回 1，表示获得事件成功
            }
      //如果前一个鼠标事件不是鼠标左键击键事件并且当前的鼠标事件是左键击键事件
```

```
        if ( !(oldbutton & IAL_MOUSE_LEFTBUTTON) &&
              (button & IAL_MOUSE_LEFTBUTTON) )
        {
            if (time1) {//  如果 time1 不为 0
                interval = __mg_timer_counter - time1;//    时间间隔等于当前时钟计数减去        time1
                if (interval <= dblclicktime)//    如果时间间隔小于双击事件时间间隔
                    me->event = ME_LEFTDBLCLICK;//      鼠标事件为左键双击事件
                else
                    me->event = ME_LEFTDOWN;//    否则为单击事件
                time1 = 0; //time1  设为  0
            }
            else{
                time1 = __mg_timer_counter; //   如果 time1 为 0,  设置  time1 为当前时钟计数
                me->event  = ME_LEFTDOWN; //    鼠标事件为单击事件
            }
            goto mouseret;
        }

        if ( (oldbutton & IAL_MOUSE_LEFTBUTTON) && //          如果前一个鼠标事件为单击
             !(button & IAL_MOUSE_LEFTBUTTON) )//        且当前事件不为左键单击事件
        {
            me->event  = ME_LEFTUP;      //鼠标事件类型为左键释放
            goto mouseret;
        }
//如果前一个鼠标事件不是右击事件且当前事件为右击事件
        if ( !(oldbutton & IAL_MOUSE_RIGHTBUTTON) &&
              (button & IAL_MOUSE_RIGHTBUTTON) )
        {
            if (time2) {     //如果 time2 不为 0
                interval = __mg_timer_counter - time2;       //两次击键的时间间隔
                if (interval <= dblclicktime)//    如果两次击键的时间间隔小于双击的时间间隔
                    me->event = ME_RIGHTDBLCLICK; //      鼠标事件为右键双击事件
                else                               //两次击键的时间间隔大于双击的时间间隔
                    me->event = ME_RIGHTDOWN;       //鼠标事件为右键单击事件
                time2 = 0;// 将 time2 置为 0
            }
            else{
                time2 = __mg_timer_counter;      //如果 time2 为 0，将 time2 设置为当前的时间
                me->event  = ME_RIGHTDOWN; //     鼠标事件为右键单击事件
            }
            goto mouseret;
        }
//如果前一个事件为鼠标右键单击事件且当前事件不为鼠标右键单击事件
        if ( (oldbutton & IAL_MOUSE_RIGHTBUTTON) &&
            !(button & IAL_MOUSE_RIGHTBUTTON) )
        {
            me->event = ME_RIGHTUP;//    则鼠标事件为鼠标右键释放事件
            goto mouseret;
        }
    }
//如果事件类型包含键盘事件
    if (event & IAL_KEYEVENT) {//
        int nr_keys = IAL_UpdateKeyboard ();//    更新键盘信息

        if (nr_keys == 0)//  如果 nr_keys 为 0，表示获取事件信息失败返，回        0
            return  0;

        lwe-> type = LWETYPE_KEY;        //设置底层事件类型为     LWETYPE_KEY
        keystate = IAL_GetKeyboardState ();       //获取键盘状态信息
//该循环的作用是判断哪个键按下或者弹起，记录相应事件的发生事件和扫描码等信息
        for (i = 1; i < nr_keys; i++) {
            if (!oldkeystate[i] && keystate[i]) {
                 ke-> event = KE_KEYDOWN;
                 ke_time =__mg_timer_counter;
                 ke-> scancode = i;
                 olddownkey = i;
                 break;
            }
            if (oldkeystate[i] && !keystate[i]) {
                 ke-> event = KE_KEYUP;
                 ke-> scancode = i;
                 break;
            }
        }
        if (i == nr_keys) { //   如果将所有的键都扫描完
            if (olddownkey == 0)      //且没有键按下时，返回      0，表示获取事件失败
                return  0;
            ke->scancode = olddownkey;      //若有键按下，则键盘事件的扫描码就为         olddownkey
            interval = __mg_timer_counter - ke_time; //    计算时间间隔
            treat_longpress (ke, interval);//  判断按键类型
            if (ke-> event == 0)     //表示获取事件失败
                return  0;                //返回 0
        }

        make = (ke-> event == KE_KEYDOWN)?1:0;

        if (i != nr_keys) {//   没有将所有的键都扫描完
            unsigned leds;   //用于点亮见哦按的三个      LED 灯
        switch (ke->scancode) {
            case SCANCODE_CAPSLOCK:        //如果按键扫描码为大小写锁定键
                if (make && caps_off) {       //如果是按键消息且    caps_off为1
                    capslock = 1 - capslock; //  转换锁定状态
                    leds = slock | (numlock << 1) | (capslock << 2);//    设置 leds的值
                    IAL_SetLeds (leds);//   用leds的值点亮对应的    LED 等
                    status = (DWORD  )leds << 16; 将leds加入状态信息
                }
//如果是键释放消息,       caps_off ,  如果是键按下消息,       caps_off 置0
                caps_off = 1 - make;//  设置 caps_off 的值
            break ;

            case SCANCODE_NUMLOCK:
                if (make && num_off) {
                    numlock = 1 - numlock;
                    leds = slock | (numlock << 1) | (capslock << 2);
                    IAL_SetLeds (leds);
                    status = (DWORD  )leds << 16;
                }
                num_off = 1 - make;
            break ;

            case SCANCODE_SCROLLLOCK:
                if (make & slock_off) {
                    slock = 1 - slock;
                    leds = slock | (numlock << 1) | (capslock << 2);
                    IAL_SetLeds (leds);
                    status = (DWORD  )leds << 16;
                }
                slock_off = 1 - make;
                break ;
//设置系统按键状态
            case SCANCODE_LEFTCONTROL:
                control1 = make;
                break ;

            case SCANCODE_RIGHTCONTROL:
                control2 = make;
                break ;

            case SCANCODE_LEFTSHIFT:
                shift1 = make;
                break ;
```

```
                    case SCANCODE_RIGHTSHIFT:
                        shift2 = make;
                        break;

                    case SCANCODE_LEFTALT:
                        alt1 = make;
                        break;

                    case SCANCODE_RIGHTALT:
                        alt2 = make;
                        break;

                }
//根据按键信息设置状态信息
                status &= ~(MASK_KS_SHIFTKEYS);

                status |= (DWORD)((capslock << 8) |
                                  (numlock << 7)    |
                                  (slock << 6)      |
                                  (control1 << 5)   |
                                  (control2 << 4)   |
                                  (alt1 << 3)       |
                                  (alt2 << 2)       |
                                  (shift1 << 1)     |
                                  (shift2));

                // Mouse button status
                if (oldbutton & IAL_MOUSE_LEFTBUTTON)
                    status |= KS_LEFTBUTTON;
                else if (oldbutton & IAL_MOUSE_RIGHTBUTTON)
                    status |= KS_RIGHTBUTTON;
            }
            ke->status = status;
            memcpy(oldkeystate, keystate, nr_keys);
            memcpy(&old_lwe, lwe, sizeof(LWEVENT));
            return 1;
        }

        old_lwe.type = 0;
        return 0;

mouseret:
        status &= ~(MASK_KS_BUTTONS);       //鼠标按钮状态的掩码
```

```
    oldbutton = button;
    if (oldbutton & IAL_MOUSE_LEFTBUTTON)
        status |= KS_LEFTBUTTON;
    if (oldbutton & IAL_MOUSE_RIGHTBUTTON)
        status |= KS_RIGHTBUTTON;
    me->status = status;
    memcpy (&old_lwe, lwe,  sizeof (LWEVENT  ));
    return 1;
}
```

8、将 post 消息加入到消息队列中

```
int GUIAPI  PostMessage (HWND  hWnd, int iMsg, WPARAM  wParam, LPARAM  lParam)
{
    PMSGQUEUE  pMsgQueue;
    MSG msg;

    if (!(pMsgQueue = GetMsgQueue(hWnd)))
        return ERR_INV_HWND;

    if (iMsg == MSG_PAINT) {
        LOCK_MSGQ (pMsgQueue);
        pMsgQueue-> dwState |= QS_PAINT;
        UNLOCK_MSGQ (pMsgQueue);
        return ERR_OK;
    }

    msg.hwnd  = hWnd;
    msg.message = iMsg;
    msg.wParam = wParam;
    msg.lParam = lParam;

    if (!QueueMessage(pMsgQueue, &msg))
        return ERR_QUEUE_FULL;

    return ERR_OK;
}
```

9、将消息加入 post 消息队列

```
/* post a message to a message queue */
BOOL QueueMessage (PMSGQUEUE  msg_que, PMSG msg)
{
    LOCK_MSGQ(msg_que);

    /* check whether the last message is MSG_MOUSEMOVE */
    if (msg-> message== MSG_MOUSEMOVE && msg->      hwnd == HWND_DESKTOP
                         && msg_que->  readpos != msg_que-> writepos ) {
        PMSG  last_msg;

        if (msg_que-> writepos  == 0)
            last_msg = msg_que-> msg + msg_que-> len - 1;
        else
            last_msg = msg_que-> msg + msg_que-> writepos  - 1;

        if (last_msg-> message == MSG_MOUSEMOVE
                             && last_msg->  wParam == msg-> wParam
                             && last_msg->  hwnd  == msg-> hwnd ) {
            last_msg-> lParam  = msg-> lParam ;
            last_msg-> time  = msg-> time ;
            goto ret;
        }
    }

    if ((msg_que-> writepos  + 1) % msg_que-> len == msg_que-> readpos) {
        UNLOCK_MSGQ(msg_que);
        return  FALSE;
    }

    /* Write the data and advance write pointer */
    msg_que->msg [msg_que-> writepos ] = *msg;    //将消息加入消息队列

    msg_que->writepos ++;
    if (msg_que-> writepos  >= msg_que-> len) msg_que-> writepos  = 0;

ret:
    msg_que->dwState |= QS_POSTMSG;
    UNLOCK_MSGQ (msg_que);
    return  TRUE;
}
```

10、 GetMessage 从消息队列总获取消息

第一步部分，是获取消息的部分；
第二部分，是等待消息循环的部分。

第二部分，请看函数最后部分， if(bWait) 的代码。对于线程版， 它就是通过 wait 信号量，让自己进入休眠。 对于进程版和 standalone 版，它调用 OnIdle 回调。OnIdle 回调在进程版中和线程版转化为对端口的 select 方法调用，从而导致一个较短时间的休眠。

重点看第一部分，它按照优先级，依次取 MSG_QUIT 消息，同步消息， notify 消息， post 消息，和 MSG_PAINT 消息和 MSG_TIMER 消息。

```
static inline BOOL  GUIAPI  GetMessage (PMSG pMsg,  HWND  hWnd)
{
        return  PeekMessageEx (pMsg, hWnd, 0, 0, TRUE, PM_REMOVE);
}
BOOL  PeekMessageEx (PMSG pMsg,  HWND  hWnd,  int iMsgFilterMin,   int iMsgFilterMax,   BOOL
bWait,  UINT  uRemoveMsg)
{
                PMSGQUEUE  pMsgQueue;
                PQMSG phead;
        //若pMsg 不为空或者当前窗口不是桌面窗口且不是主窗口返回            FALSE
                if (!pMsg || (hWnd != HWND_DESKTOP && !MG_IS_MAIN_WINDOW(hWnd)))
                        return  FALSE;
                pMsgQueue = __mg_dsk_msg_queue;     //设置消息队列为默认消息队列
                memset (pMsg, 0,  sizeof(MSG ));      //将pMsg指向的内存空间置为     0
        checkagain:
                LOCK_MSGQ (pMsgQueue);             //将消息队列上锁
                if (pMsgQueue-> dwState  & QS_QUIT) {       //如果消息队列的队首的消息类型为        QS_QUIT
                        pMsg-> hwnd  = hWnd;               //设置 pMsg 的窗口句柄为当前窗口
                        pMsg-> message = MSG_QUIT; //   设置消息类型为    MSG_QUIT
                        pMsg-> wParam  = 0;       //设置参数
                        pMsg-> lParam  = 0;
                        SET_PADD (NULL);

                        if (uRemoveMsg == PM_REMOVE) { //     如果参数  uRemoveMsg 为PM_REMOVE  ,
                                pMsgQueue-> loop_depth  --;    //消息队列的循环深度减一
                                if (pMsgQueue-> loop_depth  == 0)    //如果消息队列的循环深度为      0
                                        pMsgQueue-> dwState  &= ~QS_QUIT;      //设置消息队列的状态
                        }

                        UNLOCK_MSGQ (pMsgQueue);//    解锁消息队列
                        return  FALSE;     //返回错误，表示获取消息失败
                }
                if (pMsgQueue-> dwState  & QS_NOTIFYMSG) {//    如果消息队列类型为     QS_NOTIFYMSG
                        if (pMsgQueue-> pFirstNotifyMsg   ) { // 如果当前消息队列不为空
                                phead = pMsgQueue-> pFirstNotifyMsg   ;//phead指向当前消息队列的队首

                                *pMsg = phead->  Msg ;    //队首的消息信息赋给     pMsg指向的消息
                                SET_PADD (NULL);

                                if (IS_MSG_WANTED(pMsg->     message)) {// 该消息是否是合适的消息
                                    if (uRemoveMsg == PM_REMOVE) { // uRemoveMsg      为 PM_REMOVE
                                        pMsgQueue-> pFirstNotifyMsg    = phead-> next ; // 消息头指向下一个消息
                                        FreeQMSG (phead);     //释放刚刚取出的消息所占的内存空间
                                    }
                UNLOCK_MSGQ (pMsgQueue);
                return  TRUE;
            }
        }
        else    //如果当前消息队列为空
            pMsgQueue-> dwState  &= ~QS_NOTIFYMSG;
    }

    if (pMsgQueue-> dwState  & QS_POSTMSG) {//   如果消息队列类型为     QS_ POSTMSG
        if (pMsgQueue-> readpos != pMsgQueue->  writepos) { //  读消息位置！  =写消息位置
            *pMsg = pMsgQueue->   msg[pMsgQueue->  readpos];// 读第 readpos条消息
            SET_PADD (NULL);
            if (IS_MSG_WANTED(pMsg->     message)) {
                CheckCapturedMouseMessage (pMsg);
                if (uRemoveMsg == PM_REMOVE) {
                    pMsgQueue-> readpos++;// 读消息的位置指向下一条消息
                    if (pMsgQueue-> readpos >= pMsgQueue->  len)
                        pMsgQueue-> readpos = 0;
                }

                UNLOCK_MSGQ (pMsgQueue);
                return  TRUE;
            }
        }
        else
            pMsgQueue-> dwState  &= ~QS_POSTMSG;
    }

    /*
     * check invalidate region of the windows
     */
```

// MSG_PAINT   消息，其重点是检查了   QS_PAINT 标志。当有   QS_PAINT 标志的时候，它实际上通过   msgCheckHostedTree 函数，来检查那些窗口是需要重绘的。那些需要重绘的窗口，就会产生 MSG_PAINT   消息。

```
    if (pMsgQueue-> dwState  & QS_PAINT && IS_MSG_WANTED(MSG_PAINT)) {
        PMAINWIN    pHostingRoot;
        HWND  hNeedPaint;
        PMAINWIN    pWin;

        pMsg-> message = MSG_PAINT;      //设置消息类型
        pMsg-> wParam  = 0;
        pMsg-> lParam  = 0;
        SET_PADD (NULL);
        pHostingRoot = __mg_dsk_win;      // 设置根窗口
        if ( (hNeedPaint = msgCheckHostedTree (pHostingRoot)) ) {//    获得无效区域的句柄
            pMsg-> hwnd  = hNeedPaint;
            pWin = ( PMAINWIN   ) hNeedPaint;
            pMsg-> lParam  = (LPARAM  )(&pWin->  InvRgn .rgn);
            UNLOCK_MSGQ (pMsgQueue);
            return  TRUE;
        }

        /* no paint message */
        pMsgQueue-> dwState  &= ~QS_PAINT;
    }
    if (pMsgQueue-> dwState  & QS_DESKTIMER) {
        pMsg-> hwnd  = HWND_DESKTOP;
        pMsg-> message = MSG_TIMER;
        pMsg-> wParam  = 0;
        pMsg-> lParam  = 0;

        if  (uRemoveMsg == PM_REMOVE) {
            pMsgQueue-> dwState  &= ~QS_DESKTIMER;
        }
        return  TRUE;
    }
    if (pMsgQueue-> TimerMask   && IS_MSG_WANTED(MSG_TIMER)) {
        int slot;
        TIMER  * timer;
        /* get the first expired timer slot */
        slot = pMsgQueue->  FirstTimerSlot  ;
        do {
            if  (pMsgQueue-> TimerMask   & (0x01 << slot))
                break ;

            slot ++;
            slot %= DEF_NR_TIMERS;
            if  (slot == pMsgQueue->  FirstTimerSlot  ) {
                slot = -1;
                break ;
            }
        }  while  (TRUE);

        pMsgQueue-> FirstTimerSlot   ++;
        pMsgQueue-> FirstTimerSlot   %= DEF_NR_TIMERS;
        if ((timer = __mg_get_timer (slot))) {

            unsigned int tick_count = timer-> tick_count ;

            timer-> tick_count = 0;
            pMsgQueue-> TimerMask &= ~(0x01 << slot);

            if (timer-> proc) {
                BOOL ret_timer_proc;

                /* unlock the message queue when calling timer proc */
                UNLOCK_MSGQ (pMsgQueue);

                /* calling the timer callback procedure */
                ret_timer_proc = timer-> proc (timer-> hWnd ,
                        timer-> id, tick_count);

                /* lock the message queue again */
                LOCK_MSGQ (pMsgQueue);

                if (!ret_timer_proc) {
                    /* remove the timer */
                    __mg_remove_timer (timer, slot);
                }
            }
            else{
                pMsg-> message= MSG_TIMER;
                pMsg-> hwnd = timer-> hWnd;
                pMsg-> wParam = timer-> id;
                pMsg-> lParam = tick_count;
                SET_PADD (NULL);

                UNLOCK_MSGQ (pMsgQueue);
                return TRUE;
            }
        }
    }

    UNLOCK_MSGQ (pMsgQueue);
    /* no message, idle */
    if (bWait) {
         int id=pMsgQueue-> OnIdle (pMsgQueue);
         if(id==5)
         {
                idle=5;
                return  TRUE;
               }
               goto checkagain;
          }
          /* no message */
          return  FALSE;
}
```

11、 TranslateMessage （）处理键盘的按键信息

TranslateMessage 函数把按键按下和弹起消息转化为 MSG_CHAR消息放到消息队列中。

```
BOOL GUIAPI TranslateMessage (PMSG pMsg)
{
     int i;

     __mg_kinfo.pos = 0;

     if ((pMsg->hwnd != HWND_DESKTOP)) {
          if  ((pMsg->message  ==  MSG_KEYDOWN    || pMsg->message  ==  MSG_SYSKEYDOWN)
&&
                    pMsg->wParam < SCANCODE_USER) {
               __mg_kinfo.shiftstate = pMsg->lParam;
               handle_scancode_on_keydown (pMsg->wParam, &__mg_kinfo);
               __mg_kinfo.oldstate = pMsg->lParam;
          }
          else if ((pMsg->message == MSG_KEYLONGPRESS
                              || pMsg->message == MSG_KEYALWAYSPRESS
                              || pMsg->message == MSG_KEYUP
                              || pMsg->message == MSG_SYSKEYUP)
                         && pMsg->wParam < SCANCODE_USER) {
               __mg_kinfo.shiftstate = pMsg->lParam;
               handle_scancode_on_keyup (pMsg->wParam, &__mg_kinfo);
               __mg_kinfo.oldstate = pMsg->lParam;
          }
     }

     if (__mg_kinfo.pos == 1) {
          SendNotifyMessage (pMsg->hwnd, MSG_CHAR, __mg_kinfo.buff[0], pMsg->lParam);
     }
     else {
          for (i = 0; i < __mg_kinfo.pos; i++)
               SendNotifyMessage (pMsg->hwnd, MSG_KEYSYM,
                         MAKEWORD (__mg_kinfo.buff[i], i), pMsg->lParam);
     }
```

```
        return FALSE;
}
```

12、 DispatchMessage 分发并处理消息

```
int GUIAPI DispatchMessage (PMSG pMsg)
{
        WNDPROC WndProc;
        int iRet;

#ifdef _TRACE_MSG
        fprintf (stderr, "Message, %s: hWnd: %x, wP: %x, lP: %lx. %s\n" ,
                Message2Str (pMsg->message),
                pMsg->hwnd,
                pMsg->wParam,
                pMsg->lParam,
                SYNMSGNAME);
#endif

        if (pMsg-> hwnd == HWND_INVALID) {

#ifdef _TRACE_MSG
                fprintf (stderr, "Message have been thrown away: %s\n" ,
                        Message2Str (pMsg->message));
#endif
                return -1;
        }

        if (pMsg-> hwnd == 0)
                return -1;

        if (!(WndProc = GetWndProc (pMsg-> hwnd)))
                return -1;

        iRet = (*WndProc)(pMsg-> hwnd , pMsg-> message, pMsg-> wParam, pMsg-> lParam );
#ifdef _TRACE_MSG
        fprintf (stderr, "Message, %s done, return value: %x\n" ,
                Message2Str (pMsg->message), iRet);
#endif
        return iRet;
}
```

四、运行过程解析

1、在 InitGUI( ) 函数中与输入相关的函数

（1）初始化输入引擎

InitGUI ( )   InitLWEvent ( )   InitIAL( )

（2）设置空闲处理函数

InitGUI ( )   SetDskIdleHandler (IdleHandler4StandAlone);

（3）启动 **standalone**模式

InitGUI ( )   StandAloneStartup ( )

2、CreateMainWindow 函数中消息队列中消息的产生

（1）**MSG_CSIZECHANGED**

#define MSG_CSIZECHANGED 0x0027

 * MSG_CSIZECHANGED

 * int client_width = (int)wParam;   The width of the client area.

 * int client_height = (int)lParam;   The height of the client area.

该消息表示窗口的客户区域被改变， 当客户区域被改变时， 这个消息以 notify 消息的类型发送给消息队列。

SendNotifyMessage (hWnd, MSG_CSIZECHANGED, pWin-> cr - pWin-> cl, pWin-> cb - pWin-> ct);

消息调用路径：

CreateMainWindow (&CreateInfo)

  SendMessage((HWND) pWin, MSG_CHANGESIZE, (WPARAM)&pWin->left, 0)

   WelcomeWinProc(hWnd, iMsg, wParam, lParam)

   PreDefMainWinProc(hWnd, message, wParam, lParam)

   DefaultPostMsgHandler(pWin, message, wParam, lParam) (case MSG_CHANGESIZE)

  RecalcClientArea ((HWND) pWin)

  SendNotifyMessage (hWnd, MSG_CSIZECHANGED, pWin->cr - pWin->cl, pWin->cb - pWin->ct);

（2）**MSG_SHOWWINDOW**

#define MSG_SHOWWINDOW 0x00A0

该消息表示窗口被显示或隐藏 。

SendNotifyMessage ((HWND) pWin, MSG_SHOWWINDOW , SW_SHOWNORMAL, 0)

（第三个参数可以为 SW_NORMAL,SW_HIDE,SW_SHOWNORMAL）

调用路径：

CreateMainWindow (&CreateInfo)

  SendMessage (HWND_DESKTOP ,MSG_ADDNEWMAINWIN , (WPARAM) pWin, (LPARAM) pWin-> pZOrderNode);

   DesktopWinProc (hWnd, iMsg, wParam, lParam)

   WindowMessageHandler (message, (PMAINWIN)wParam, lParam) (case MSG_ADDNEWMAINWIN)

   dskAddNewMainWindow(pWin, (PZORDERNODE)lParam)

SendNotifyMessage ((HWND)pWin, MSG_SHOWWINDOW, SW_SHOWNORMAL, 0);

（3）MSG_PAINT

#define MSG_PAINT 0x00B1

当窗口包含无效区域时该消息被发送给窗口。

调用路径：

CreateMainWindow (&CreateInfo)

SendMessage (HWND_DESKTOP ,MSG_ADDNEWMAINWIN , (WPARAM) pWin, (LPARAM) pWin-> pZOrderNode);

DesktopWinProc (hWnd, iMsg, wParam, lParam)

WindowMessageHandler (message, (PMAINWIN)wParam, lParam) (case MSG_ADDNEWMAINWIN)

dskAddNewMainWindow(pWin, (PZORDERNODE)lParam)

InvalidateRect ((HWND)pWin, NULL, TRUE)

PostMessage (hWnd, MSG_PAINT, 0, 0)（只改变消息队列的状态 pMsgQueue->dwState |= QS_PAINT，没有消息加入消息队列）

（4）MSG_ACTIVE

#define MSG_ACTIVE 0x0033

该消息表明由于用户点击窗口而使窗口获得了输入焦点。 当用户点击窗口使窗口获得输入焦点后该消息会被发送给窗口的消息队列。

SendNotifyMessage ((HWND)pWin, MSG_ACTIVE, TRUE, 0)

调用路径：

CreateMainWindow (&CreateInfo)

SendMessage (HWND_DESKTOP ,MSG_ADDNEWMAINWIN , (WPARAM) pWin, (LPARAM) pWin-> pZOrderNode);

DesktopWinProc (hWnd, iMsg, wParam, lParam)

WindowMessageHandler (message, (PMAINWIN)wParam, lParam) (case MSG_ADDNEWMAINWIN)

dskAddNewMainWindow(pWin, (PZORDERNODE)lParam)

dskChangActiveWindow (pWin)

SendNotifyMessage ((HWND)pWin, MSG_ACTIVE, TRUE, 0)

（5）MSG_SETFOCUS

#define MSG_SETFOCUS 0x0030

该消息表示当前窗口获得输入焦点， 当窗口获得输入焦点时该消息被发送给窗口的消息队列。

SendNotifyMessage ((HWND)pWin, MSG_SETFOCUS, (HWND)pOldActive, 0)

调用路径：

CreateMainWindow (&CreateInfo)

SendMessage (HWND_DESKTOP ,MSG_ADDNEWMAINWIN , (WPARAM) pWin, (LPARAM) pWin-> pZOrderNode);

DesktopWinProc (hWnd, iMsg, wParam, lParam)

WindowMessageHandler (message, (PMAINWIN)wParam, lParam) (case MSG_ADDNEWMAINWIN)

dskAddNewMainWindow(pWin, (PZORDERNODE)lParam)

dskChangActiveWindow (pWin)

SendNotifyMessage ((HWND)pWin, MSG_SETFOCUS, (HWND)pOldActive, 0)

（**6**）**CreateMainWindow** 函数执行过程中，共产生了四个 **notify** 消息和一个 **post** 消息，四个 **notify** 消息被依次放入消息队列，**post** 消息只改变了消息队列的状态信息，并没有加入到 **post** 消息队列中。

3、ShowWindow 函数中消息的产生

（**1**）**MSG_SHOWWINDOW**

#define MSG_SHOWWINDOW 0x00A0

调用路径：

ShowWindow(hMainWnd, SW_SHOWNORMAL)

SendNotifyMessage (hWnd, MSG_SHOWWINDOW, (WPARAM)iCmdShow, 0)

4、进入消息循环之前消息队列的状态

（**1**）依次产生了 **5** 条 **notify** 消息，依次为：

MSG_CSIZECHANGED MSG_SHOWWINDOW MSG_ACTIVE MSG_SETFOCUS MSG_SHOWWINDOW

（**2**）消息对列的状态发生改变：

pMsgQueue->dwState |= QS_NOTIFYMSG

pMsgQueue->dwState |= QS_PAINT

5、消息循环中三个函数的作用

（1）GetMessage (PMSG pMsg,HWND hWnd)

return PeekMessageEx (pMsg, hWnd, 0, 0, TRUE, PM_REMOVE);

调用 **GetMessage** 实际上调用的就是 PeekMessageEx。

1）PeekMessageEx (pMsg, hWnd, 0, 0, TRUE, PM_REMOVE) 参数含义：

pMsg：获得的消息的存储位置。

hWnd：从句柄为 hWnd 的窗口的消息队列获得消息。

TRUE：消息队列中没有消息时进入空闲处理函数。

PM_REMOVE：获得消息之后将该消息从消息队列中删除。

2）PeekMessageEx (pMsg, hWnd, 0, 0, TRUE, PM_REMOVE) 函数流程：

消息处理的优先级从高到低： QS_QUIT QS_NOTIFYMSG QS_POSTMSG QS_PAINT QS_DESKTIMER

若没有以上种类的消息处理： pMsgQueue->TimerMask

若没有以上种类的消息处理且 pMsgQueue->TimerMask 不符合要求（pMsgQueue->TimerMask && IS_MSG_WANTED(MSG_TIMER)）：根据 bWait 参数的真假决定是否进入空闲处理函数，若 bWait 为真，则进入空闲函数处理流程，之后继续获取消息。若 bWait 为假，则返回获取消息失败。

（2）TranslateMessage (PMSG pMsg)

处理键盘的按键信息。 TranslateMessage 函数把按键按下和弹起消息转化为 MSG_CHAR消息

放到消息队列中。

（3）DispatchMessage(PMSG pMsg)

调用相应的窗口处理函数对消息进行处理。

6、消息循环执行过程

（1）第 1~5 次循环：处理 notify 消息队列

依次获得 CreateMainWindow 和 ShowWindow 函数产生的 5 条 notify 消息并进行处理，没有任何实际的操作。

TranslateMessage:do nothing

DispatchMessage:do nothing

（2）第 6 次循环：处理 QS_PAINT 消息

若消息队列包含该状态会产生一个 MSG_PAINT消息。

#define MSG_PAINT 0x00B1

当窗口中包含无效区域时即窗口需要重绘时该消息会被发送到相应的窗口进行处理。

消息信息： {hwnd = 135271848, message = 177, wParam = 0, lParam = 135271924, time = 0}

Translate :do nothing

Diapatch:winproc MSG_PAINT

此时消息队列中的所有消息已经被处理完，再次调用 **GetMessage** 函数时会进入空闲处理函数，从而从输入事件中获得消息。

（3）第 7 次循环：进入空闲处理函数，并获取消息

空闲处理函数获取输入事件，产生了一条 post 消息 MSG_MOUSEMOVE 。

IdleHandler4StandAlone( ) ParseEvent( ) QueueMessage (msg_que, &Msg) MSG_MOUSEMOVE

GetMessage 获取到该消息，消息信息为：

{hwnd = 135192832, message =4, wParam = 1, lParam = 10289351, time = 0}

Translate ：donothing

DispatchMessage DesktopWinProc MouseMessageHandler PostMessage (HWND_DESKTOP, MSG_DT_MOUSEMOVE , flags, MAKELONG (x, y)) QueueMessage (msg_que, &Msg)