目录

一、 **minigui**运行模式

1、线程模式： MiniGui-Threads

定义： _MGRM_THREADS

运行在 MiniGui-Threads 上的程序可以在不同的线程中建立多个窗口，但所有的窗口在一个进程或地址空间中运行，传统意义上的嵌入式操作系统。

2、进程模式： MiniGui-Processes

定义： _MGRM_Processes或者定义 _LITE_VERSION

MiniGui-Processes 上每个程序是单独的进程，每个进程也可以建立多个窗口，并且实现了多进程窗口系统。适用于具有完整 UNIX 特性的嵌入式式系统。

3、独立应用模式： MiniGui-Standalone

定义： _MGRM_STANDALONE或者定义 _LITE_VERSION和 _STAND_ALONE

通过独立任务的方式运行，既不需要多进程支持也不需要多线程支持。

二、创建主窗口 **CreateMainWindow**

1、数据结构

1、CreateMainWindow 函数参数： **PMAINWINCREATE pCreateInfo**

结构体 MAINWINCREATE 定义了被创建的窗口的位置、 标题、类型等基本参数。 实际上包含了创建窗口的 UI 风格和窗口处理函数两方面的内容。

PMAINWINCREATE为指向该结构体的指针。

```
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;       // 主窗口菜单句柄
    HCURSOR hCursor;  // 主窗口光标句柄
    HICON hIcon;  // 主窗口图标句柄
    HWND   hHosting;   // 主窗口的托管窗口   The hosting main window
    int (*MainWindowProc)(HWND, int, WPARAM, LPARAM);    //    窗口回调函数
    int lx, ty, rx, by;    // 主窗口在屏幕坐标中的位置
    int iBkColor;      // 主窗口颜色的像素值
    DWORD dwAddData;      // 私有数据  The first private data associated with the main window
    DWORD dwReserved;   // 没有用到
}MAINWINCREATE;
typedef    MAINWINCREATE*   PMAINWINCREATE;
```

2、MAINWIN 结构体：主窗口的详细信息由该结构体给出

```
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;
```

3、 **MSGQUEUE**消息队列

```
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. */
};
```

2、CreateMainWindow 函数流程

1、判断传入的参数 **pCreateInfo** 是否为空

Case NULL: 若参数为空，返回 HWND_INVALID

Case NOT NULL: 若参数不为空，继续执行 2

2、为 **PMAINWIN** 类型的 **pWin** 分配内存空间，并判断 **pWin** 是否为空

Case NULL：分配空间失败，返回 HWND_INVALID

Case NOT NULL: 分配空间成功，继续执行 3

3、是否定义 **_LITE_VERSION**

3.a 没有定义 _LITE_VERSION, 代表 minigui 的运行模式为 MiniGui-Threads

设置 pWin 的成员 pWin->pMessages 和 pWin->pHosting

3.b 定义了 _LITE_VERSION, 代表 minigui 的运行模式为非 MiniGui-Threads

设置 pWin 的成员 pWin->pMessages 和 pWin->pHosting

4、设置 **pWin** 的成员：

```
    pWin-> pMainWin         = pWin;
    pWin-> hParent          = 0;
    pWin-> pFirstHosted     = NULL;
    pWin-> pNextHosted      = NULL;
    pWin-> DataType         = TYPE_HWND;
    pWin-> WinType          = TYPE_MAINWIN;

#ifndef _LITE_VERSION
    pWin->th                = pthread_self();
```

```
#endif
    pWin-> hFirstChild       = 0;
    pWin-> hActiveChild      = 0;
    pWin-> hOldUnderPointer  = 0;
    pWin-> hPrimitive        = 0;

    pWin-> NotifProc         = NULL;

    pWin-> dwStyle           = pCreateInfo-> dwStyle ;
    pWin-> dwExStyle         = pCreateInfo-> dwExStyle ;

    pWin-> hMenu             = pCreateInfo-> hMenu ;
    pWin-> hCursor           = pCreateInfo-> hCursor ;
    pWin-> hIcon             = pCreateInfo-> hIcon ;
    if ((pWin-> dwStyle & WS_CAPTION) && (pWin-> dwStyle & WS_SYSMENU))
        pWin-> hSysMenu = CreateSystemMenu (( HWND )pWin, pWin-> dwStyle );
    else
        pWin-> hSysMenu = 0;

    pWin-> pLogFont          = GetSystemFont (SYSLOGFONT_WCHAR_DEF);

    pWin-> spCaption         = FixStrAlloc ( strlen (pCreateInfo-> spCaption));
    if (pCreateInfo-> spCaption [0])
        strcpy (pWin-> spCaption, pCreateInfo-> spCaption);

    pWin-> MainWindowProc    = pCreateInfo-> MainWindowProc ;
    pWin-> iBkColor          = pCreateInfo-> iBkColor ;

    pWin-> pCaretInfo  = NULL;

    pWin-> dwAddData  = pCreateInfo-> dwAddData ;
    pWin-> dwAddData2  = 0;

#if !defined (_LITE_VERSION) || defined (_STAND_ALONE)
    if ( !( pWin-> pZOrderNode = malloc (sizeof(ZORDERNODE ))) )
        goto err;
#endif

    /* Scroll bar */
    if (pWin-> dwStyle & WS_VSCROLL) {
        pWin-> vscroll .minPos = 0;
        pWin-> vscroll .maxPos = 100;
        pWin-> vscroll .curPos = 0;
        pWin-> vscroll .pageStep = 0;
        pWin-> vscroll .barStart = 0;
        pWin-> vscroll .barLen = 10;
        pWin-> vscroll .status = SBS_NORMAL;
    }
    else
        pWin-> vscroll .status = SBS_HIDE | SBS_DISABLED;

    if (pWin-> dwStyle  & WS_HSCROLL) {
        pWin-> hscroll .minPos = 0;
        pWin-> hscroll .maxPos = 100;
        pWin-> hscroll .curPos = 0;
        pWin-> hscroll .pageStep = 0;
        pWin-> hscroll .barStart = 0;
        pWin-> hscroll .barLen = 10;
        pWin-> hscroll .status = SBS_NORMAL;
    }
    else
        pWin-> hscroll .status = SBS_HIDE | SBS_DISABLED;
```

5、 SendMessage ((HWND)pWin, MSG_NCCREATE, 0, (LPARAM)pCreateInfo)

```
#define MSG_NCCREATE            0x0061
```

作用：表示该窗口已经创建但是还没有向系统进行注册， 当收到这种类型的消息时可以对自己创建的对象进行初始化， 但不能创建子窗口， 也不能进行绘图。 如果函数返回值为非零值，创建的窗口将被销毁。

```
* \code
 * MSG_NCCREATE for main windows:
 * PMAINWINCREATE create_info = (PMAINWINCREATE)lParam;
 *
 * MSG_NCCREATE for controls:
 * DWORD add_data = (DWORD)lParam;
 * \endcode
 *
 * \param create_info The pointer to the MAINWINCREATE structure which is
 *          passed to CreateMainWindow function.
 * \param add_data The first additional data passed to CreateWindowEx function.
 *
 * \sa CreateMainWindow, CreateWindowEx, MAINWINCREATE
```

```
main    CreateMainWindow(&CreateInfo)
   SendMessage ((HWND)pWin, MSG_NCCREATE, 0, (LPARAM)pCreateInfo)
   HelloWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
   DefaultMainWinProc(hWnd, message, wParam, lParam)
(message >= MSG_FIRSTCREATEMSG && message <= MSG_LASTCREATEMSG)
   DefaultCreateMsgHandler(pWin, message, wParam, lParam)
```

返回 0，什么也没做

6 、 SendMessage ((HWND)pWin, MSG_SIZECHANGING,(WPARAM)&pCreateInfo->lx, (LPARAM)&pWin->left);

#define MSG_SIZECHANGING 0x0025

作用：指示了将要被更改的窗口的大小，当窗口大小将要发生改变时，该消息会发送给窗口。如果你想要控制窗口改变后的实际位置和大小（窗口改变可能是 MoveWindow 或者其他函数引起的），你需要使用 MSG_SIZECHANGING 作为 SendMessage 函数的第二个参数，并且通过第二个参数返回位置和大小信息。

```
* \code
 * MSG_SIZECHANGING
 * const RECT* rcExpect = (const RECT*)wParam;
 * RECT* rcResult = (RECT*)lParam;
 * \endcode
 *
 * \param rcExpect The expected size of the window after changing.
 * \param rcResult The actual size of the window after changing.
 *
```

main CreateMainWindow(&CreateInfo)

SendMessage((HWND)pWin,MSG_SIZECHANGING,(WPARAM)&pCreateInfo->lx,(LPARAM)&pWin->left)

HelloWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)

DefaultMainWinProc(hWnd, message, wParam, lParam)

(message >= MSG_FIRSTPOSTMSG && message <= MSG_LASTPOSTMSG)

DefaultPostMsgHandler(pWin, message, wParam, lParam)

memcpy ((PRECT)lParam, (PRECT)wParam, sizeof (RECT))

将 wParam 的信息复制给 lParam，返回 0

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

#define MSG_CHANGESIZE 0x0022

作用：确定改变后的窗口大小

main CreateMainWindow(&CreateInfo)

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

HelloWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)

DefaultMainWinProc(hWnd, message, wParam, lParam)

(message >= MSG_FIRSTPOSTMSG && message <= MSG_LASTPOSTMSG)

(1) OnChangeSize (pWin, (PRECT)wParam, (PRECT)lParam) 确定边界、标题、滚动条等的大小

(2) RecalcClientArea ((HWND)pWin) 确定客户区域的坐标和大小

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

#define MSG_ADDNEWMAINWIN 0x00F0

作用：绘制窗口

main CreateMainWindow(&CreateInfo)

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

DesktopWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)

(message >= MSG_FIRSTWINDOWMSG && message <= MSG_LASTWINDOWMSG)

WindowMessageHandler (message, (PMAINWIN)wParam, lParam)

case MSG_ADDNEWMAINWIN

dskAddNewMainWindow(pWin, (PZORDERNODE)lParam)

1 dskUpdateGCRInfoOnShowNewMainWin (pWin)

2 SendAsyncMessage ((HWND)pWin, MSG_NCPAINT, 0, 0)(相当于 SendMessage)

* MSG_NCPAINT :绘制非客户区域 brief Indicates that paints non-client area.

#define MSG_NCPAINT 0x00B2

2 HelloWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)

2 DefaultMainWinProc(hWnd, message, wParam, lParam)

(message >= MSG_FIRSTPAINTMSG && message <= MSG_LASTPAINTMSG)

2 DefaultPaintMsgHandler(pWin, message, wParam, lParam)

2 wndDrawNCFrame (pWin, (HDC)wParam, (const RECT*)lParam)

(1) wndDrawNCArea (pWin, hdc)(绘制边框)

(2) wndDrawScrollBar (pWin, hdc)(绘制滚动条)

(3) wndDrawCaption (pWin, hdc, !(pWin-> dwStyle & WS_DISABLED) && (GetActiveWindow() == (HWND)pWin));(绘制标题)

(4) DrawMenuBarHelper (pWin, hdc, prcInvalid)

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

4 InvalidateRect ((HWND)pWin, NULL, TRUE)

5 dskChangActiveWindow (pWin)

9、 SendMessage ((HWND)pWin, MSG_CREATE, 0, (LPARAM)pCreateInfo)

#define MSG_CREATE 0x0060

作用：表示窗口已经创建成功

四、 **ShowWindow**函数流程

该函数的作用：根据窗口的显示状态信息对窗口进行显示

1、MG_CHECK_RET (MG_IS_NORMAL_WINDOW(hWnd), FALSE)

作用：如果不是一个正规窗口，返回 FALSE; 结束 ShowWindow

#define MG_CHECK_RET(condition, ret) **if** (!(condition)) **return** ret

#define MG_IS_NORMAL_WINDOW(hWnd) (hWnd != HWND_DESKTOP && MG_IS_WINDOW(hWnd))

/* hWnd is a normal window, not including HWND_DESKTOP*/

2、根据窗口类型和窗口的显示类型对窗口的显示状态进行调整

SW_SHOWNORMAL：激活并显示一个窗口，会将窗口显示在最上层，通过在 DesktopWinProc 中调用 dskMoveToTopMost() 实现。如果窗口被最小化或最大化， 系统将其恢复到原来的尺寸和大小。应用程序在第一次显示窗口的时候应该指定此标志。

SW_SHOW ：在窗口原来的位置以原来的尺寸激活和显示窗口 。在 DesktopWinProc 中调用 dskShowMainWindow 如果窗口原来被覆盖，使用该消息后依然维持原样。

SW_HIDE ：隐藏窗口并激活其他窗口。

如果 **hWnd** 指示的窗口是主窗口：

```
    switch (iCmdShow)
            {
                caseSW_SHOWNORMAL:
                    SendMessage (HWND_DESKTOP,
                        MSG_MOVETOTOPMOST, (    WPARAM   )hWnd, 0);
                break ;

                caseSW_SHOW:
                    SendMessage (HWND_DESKTOP,
                        MSG_SHOWMAINWIN, (     WPARAM   )hWnd, 0);
                break ;

                caseSW_HIDE:
                    SendMessage (HWND_DESKTOP,
                        MSG_HIDEMAINWIN, (     WPARAM   )hWnd, 0);
                break ;
            }
```

如果 **hWnd** 指示的窗口是控件窗口：

```
    #define WS_EX_CTRLASMAINWIN            0x40000000L
    * \brief The control can be displayed out of the main window which contains the control.
    //WS_EX_CTRLASMAINWIN      便是空间窗口可以被显示在主窗口之外
            if (pControl->dwExStyle & WS_EX_CTRLASMAINWIN){// 控件可以显示在主窗口之
外
                if (iCmdShow == SW_SHOW)
                    SendMessage (HWND_DESKTOP, MSG_SHOWGLOBALCTRL,
(WPARAM  )hWnd, iCmdShow);
                elseif (iCmdShow == SW_HIDE)
                    SendMessage (HWND_DESKTOP, MSG_HIDEGLOBALCTRL,
(WPARAM  )hWnd, iCmdShow);
                else
                    return  FALSE;
            }
            else{    //控件不可以显示在主窗口之外
                switch (iCmdShow) {
                caseSW_SHOWNORMAL:       //正常显示窗口
                caseSW_SHOW:
                    if (!(pControl->  dwStyle  & WS_VISIBLE)) {
                        pControl-> dwStyle  |= WS_VISIBLE;

                        SendAsyncMessage (hWnd, MSG_NCPAINT, 0, 0);
                        InvalidateRect (hWnd, NULL, TRUE);
                    }
                break ;

                case SW_HIDE:     //隐藏窗口
                    if (pControl-> dwStyle  & WS_VISIBLE) {

                        pControl-> dwStyle  &= ~WS_VISIBLE;
                        InvalidateRect (( HWND )(pControl-> pParent),
                            (RECT *)(&pControl->  left ), TRUE);
                    }
                break ;
                }
            }
```

3、根据 **iCmdShow** 等信息确定当前窗口是否失去输入焦点

```
#define MSG_KILLFOCUS               0x0031
* \brief Indicates that the window has lost the input focus.
//MSG_KILLFOCUS  ，指示当前控制窗口失去输入焦点
//如果窗口的显示状态为    SW_HIDE  且控制窗口的父窗口的活动窗口为当前的控制窗口
        if (iCmdShow == SW_HIDE && pControl->     pParent->active  == pControl) {
            SendNotifyMessage (hWnd, MSG_KILLFOCUS, 0, 0);
            pControl-> pParent->active  = NULL;
        }
    }
```

4、向消息队列发送消息 **MSG_SHOWWINDOW** 根据 **iCmdShow** 指示当前窗口的显示状态

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

五、获取消息 GetMessage()

1、函数作用：将消息队列中最先发生的消息参数传递给   pMsg

```
static inline BOOL  GUIAPI  GetMessage (PMSG  pMsg,  HWND  hWnd)
{
return  PeekMessageEx (pMsg, hWnd, 0, 0, TRUE, PM_REMOVE);
}
```

2、数据结构

```
struct _MSGQUEUE
{
    DWORD  dwState;                     //  消息队列的状态
    PQMSG    pFirstNotifyMsg  ;       // head of the notify message queue , notify 消息队列的队首
    PQMSG    pLastNotifyMsg  ;        // tail of the notify message queue  , notify 消息队列的队尾
    IDLEHANDLER    OnIdle ;               // Idle handler 空闲的处理程序
    MSG *  msg;                          /* post message buffer */  消息的地址
    int len;                             /* buffer len */  消息队列占用缓存大小
    int readpos, writepos ;          /* positions for reading and writing */    读写位置
```

```
        int FirstTimerSlot ;            /* the first timer slot to be checked */   第一个时间槽
        DWORD  TimerMask ;                  /* timer slots mask */  时间槽标记
        int loop_depth ;                /* message loop depth, for dialog boxes. */   消息循环深度,
    };
    DWORD  dwState 可选的取值：
    #define QS_NOTIFYMSG              0x10000000    //标志表示消息队列中有待处理的  notify  消息
    #ifndef _LITE_VERSION
      #define QS_SYNCMSG              0x20000000    //表示有待处理的同步消息
    #else
      #define QS_DESKTIMER            0x20000000
    #endif
    #define QS_POSTMSG                0x40000000    //表示有待处理的  post  消息
    #define QS_QUIT                 0x80000000      //对应 MSG_QUIT
    #define QS_INPUT                0x01000000
    #define QS_PAINT                0x02000000      //对应 MSG_PAINT
    #define QS_TIMER                0x0000FFFF      //对应 MSG_TIMER
    #define QS_EMPTY                0x00000000
```

3、函数流程

```
 *
 *                          BOOL bWait, UINT uRemoveMsg)
 * \brief Peeks a message from the message queue of a main window.
 *
 * This functions peek a message from the message queue of the window \a hWnd;
 * if \a bWait is TRUE, it will wait for the message, else return immediatly.
 *
 * \param pMsg Pointer to the result message.
 * \param hWnd The handle to the window.
 * \param iMsgFilterMin The min identifier of the message that should be peeked.
 * \param iMsgFilterMax The max identifier of the message that should be peeked.
 * \param bWait Whether to wait for a message.
 * \param uRemoveMsg Whether remove the message from the message queue.
 *         Should be the following values:
 *         - PM_NOREMOVE\n
 *           Leave it in the message queue.
 *         - PM_REMOVE
 *           Remove it from the message queue.
 *         - PM_NOYIELD
 *           Nouse now.
 *
 * \return TRUE if there is a message peeked, or FALSE.
 *
 * \sa GetMessage, PeekPostMessage, HavePendingMessage, PostMessage
    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;
    }
```

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。

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

msgCheckInvalidRegion 函数判断窗口是否存在无效区域，如果存在，则是需要重绘，否则，它会继续查找下个 hosted 窗口。

3、相关函数

1、msgCheckHostedTree

```
static HWND  msgCheckHostedTree (PMAINWIN  pHosting)
{
    HWND  hNeedPaint;
    PMAINWIN  pHosted;

    if ( (hNeedPaint = msgCheckInvalidRegion (pHosting)) )
        return  hNeedPaint;

    pHosted = pHosting-> pFirstHosted ;
    while (pHosted) {
        if ( (hNeedPaint = msgCheckHostedTree (pHosted)) )
            return  hNeedPaint;
        pHosted = pHosted-> pNextHosted ;
    }
    return  0;
}
```

2、msgCheckInvalidRegion

```
static HWND msgCheckInvalidRegion (PMAINWIN pWin)
{
    PCONTROL  pCtrl = ( PCONTROL  )pWin;
    HWND  hwnd;

    if (pCtrl-> InvRgn .rgn.head)
        return  (HWND  )pCtrl;
    pCtrl = pCtrl-> children ;
    while (pCtrl) {

        if ((hwnd = msgCheckInvalidRegion (( PMAINWIN ) pCtrl)))
            return hwnd;

        pCtrl = pCtrl-> next;
    }

    return 0;
}
```

这里面涉及很多点：

1. MiniGUI 窗口的管理方式，是主窗口和普通窗口分离的。主窗口采用 hosting 的链表树方式管理，可以遍历到所有的主窗口。而每个主窗口则是一个树的根节点，通过 parent-child 关系管理所有的子窗口。所以，msgCheckHostedTree 是遍历主窗口用的，而 msgCheckInvalidRegion 是遍历子窗口用的。注意这两个函数的返回值，都是需要重绘的窗口句柄。
2. 当找到一个需要重绘的窗口句柄的时候，它就返回该窗口句柄。返回后，消息循环接下来回调用 DispatchMessage ，该函数会调用到窗口过程。在窗口过程中，处理 MSG_PAINT 消息时，我们必须用 BeginPaint 和 EndPaint 来获取绘制的 DC。这一点非常重要，因为在 BeginPaint 中，会清除无效区域。当无效区域被清除后，下次在此调用 msgCheckHostedTree 和 msgCheckInvalidRegion 的时候，该窗口的子窗口或者兄弟窗口就会被检测。一直等到所有窗口被检测完毕后，整个绘制过程才算完成。在此过程中，MiniGUI 会多次遍历所有的窗口。
3. msgCheckHostedTree 和 msgCheckInvalidRegion 相结合使用，是通过递归的方法进行的一个先根遍历法，这个方法，总是能够保证父窗口先于子窗口被绘制，从而保证了窗口之间以正确的次序被绘制。

六、输入事件响应流程

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 ()：

1、数据结构 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 ;
```

2、输入引擎的参数信息：

```
static INPUT  inputs [] =
{
/* General IAL engines ... */
#ifdef _DUMMY_IAL
    {"dummy" , InitDummyInput, TermDummyInput},
#endif
#ifdef _AUTO_IAL
    {"auto" , InitAutoInput, TermAutoInput},
#endif
#ifdef _RANDOM_IAL
```

```
    {"random" , InitRandomInput, TermRandomInput},
#endif
#ifdef _CUSTOM_IAL
    {"custom" , InitCustomInput, TermCustomInput},
#endif
#ifdef _COMM_IAL
    {"comm" , InitCOMMInput, TermCOMMInput},
#endif
#ifdef _QVFB_IAL
    {"qvfb"  , InitQVFBInput, TermQVFBInput},
#endif
#ifdef _WVFB_IAL
    {"wvfb"  , InitWVFBInput, TermWVFBInput},
#endif
#ifdef _NATIVE_IAL_ENGINE
    {"console" , InitNativeInput, TermNativeInput},
#endif
#ifdef _DFB_IAL
    {"dfb" , InitDFBInput, TermDFBInput},
#endif
/* ... end of general IAL engines */

/* Board-specific IAL engines... */
#ifdef _ADS_IAL
    {"ADS"  , InitADSInput, TermADSInput},
#endif
#ifdef _VR4181_IAL
    {"VR4181"  , InitVR4181Input, TermVR4181Input},
#endif
#ifdef _HELIO_IAL
    {"Helio"  , InitHelioInput, TermHelioInput},
#endif
#ifdef _EP7211_IAL
    {"EP7211" , InitEP7211Input, TermEP7211Input},
#endif
#ifdef _TFSTB_IAL
    {"TF-STB"  , InitTFSTBInput, TermTFSTBInput},
#endif
#ifdef _HH5249KBDIR_IAL
    {"hh5249kbdir"  , InitHH5249KbdIrInput, TermHH5249KbdIrInput},
#endif
#ifdef _IPAQ_IAL
    {"ipaq" , InitIPAQInput, TermIPAQInput},
#endif
```

```
#ifdef _T800_IAL
        {"T800" , InitT800Input, TermT800Input},
#endif
#ifdef _MPC823_IAL
        {"MPC823"  , InitMPC823Input, TermMPC823Input},
#endif
#ifdef _UCB1X00_IAL
        {"UCB1X00"   , InitUCB1X00Input, TermUCB1X00Input},
#endif
#ifdef _PX255B_IAL
        {"PX255B"  , InitPX255BInput, TermPX255BInput},
#endif
#ifdef _MC68X328_IAL
        {"MC68X328"   , InitMC68X328Input, TermMC68X328Input},
#endif
#ifdef _SMDK2410_IAL
        {"SMDK2410"   , Init2410Input, Term2410Input},
#endif
#ifdef _DMGSTB_IAL
        {"dmg-stb" , InitDMGSTBInput, TermDMGSTBInput},
#endif
#ifdef _FIP_IAL
        {"fip"  , InitFIPInput, TermFIPInput},
#endif
#ifdef _PALMII_IAL
        {"palm2" , InitPALMIIInput, TermPALMIIInput},
#endif
#ifdef _HH2410R3_IAL
        {"hh2410r3"  , InitHH2410R3Input, TermHH2410R3Input},
#endif
#ifdef _C33L05_IAL
        {"c33l05"  , InitC33L05Input, TermC33L05Input},
#endif
#ifdef _HH2440_IAL
        {"hh2440" , InitHH2440Input, TermHH2440Input},
#endif
#ifdef _EMBEST44B0_IAL
        {"embest44b0" , InitEMBEST44b0Input, TermEMBEST44b0Input},
#endif
#ifdef _SVPXX_IAL
        {"svpxx"  , InitSvpxxInput, TermSvpxxInput},
#endif
#ifdef _ADS7846_IAL
        {"ads7846" , InitAds7846Input, TermAds7846Input},
#endif
#ifdef _L7200_IAL
    {"l7200" , InitL7200Input, TermL7200Input},
#endif
#ifdef _ARM3000_IAL
    {"arm3000" , InitARM3000Input, TermARM3000Input},
#endif
#ifdef _EMBEST2410_IAL
    {"embest2410" , InitEMBEST2410Input, TermEMBEST2410Input},
#endif
#ifdef _EM8620_IAL
    {"em8620" , InitEm8620Input, TermEm8620Input},
#endif
#ifdef _EM86_IAL
    {"em86" , InitEm86Input, TermEm86Input},
#endif
#ifdef _EM85_IAL
    {"em85" , InitEm85Input, TermEm85Input},
#endif
#ifdef _FFT7202_IAL
    {"fft7202"  , InitFFTInput, TermFFTInput},
#endif
#ifdef _UTPMC_IAL
    {"utpmc" , InitUTPMCInput, TermUTPMCInput},
#endif
#ifdef _DM270_IAL
    {"dm270"  , InitDM270Input, TermDM270Input},
#endif
#ifdef _EVMV10_IAL
    {"evmv10" , InitXscaleEVMV10Input, TermXscaleEVMV10Input},
#endif
#ifdef _FXRM9200_IAL
    {"fxrm9200"  , InitRm9200Input, TermRm9200Input},
#endif
#ifdef _ABSSIG_IAL
    {"abssig" , InitABSSIGInput, TermABSSIGInput},
#endif
#ifdef _SKYEYE_EP7312_IAL
    {"SkyEyeEP7312"  , InitSkyEyeEP7312Input, TermSkyEyeEP7312Input},
#endif
#ifdef _FIGUEROA_IAL
    {"figueroa"  , InitFiguerOAInput, TermFiguerOAInput},
#endif
```

```
#ifdef _HI3510_IAL
    {"hi3510" , InitHI3510Input, TermHI3510Input},
#endif
#ifdef _HI3610_IAL
    {"hi3610" , InitHI3610Input, TermHI3610Input},
#endif

/* ... end of board-specific IAL engines */
};
```

3、函数流程

```
INPUT * __mg_cur_input;
#define NR_INPUTS  (sizeof (inputs) / sizeof (INPUT))

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;
}
```

4、 QVFB输入引擎初始化   InitQVFBInput

```
BOOL  InitQVFBInput   (INPUT * input,  const char* mdev,  const char* mtype)
{
    char file [50];
    int display;
    fprintf (stderr,"init qvfb ial\n"   );
#ifdef _INCORE_RES
 /* Define if build MiniGUI for no file I/O system */
/* #undef _INCORE_RES */
//#define _INCORE_RES 1
    display = 0;
#endif
    /* open mouse pipe */
    sprintf  (file, QT_VFB_MOUSE_PIPE, display);
    if ((mouse_fd =  open (file, O_RDONLY)) < 0) {
        fprintf  (stderr, "QVFB IAL engine: can not open mouse pipe.\n"     );
        return  FALSE;
    }
    /* open keyboard pipe */
    sprintf  (file, QT_VFB_KEYBOARD_PIPE, display);
    if ((kbd_fd =  open (file, O_RDONLY)) < 0) {
        fprintf  (stderr, "QVFB IAL engine: can not open keyboard pipe.\n"     );
        return  FALSE;
    }
    input-> update_mouse = mouse_update;
    input-> get_mouse_xy  = mouse_getxy;
    input-> set_mouse_xy = NULL;
    input-> get_mouse_button  = mouse_getbutton;
    input-> set_mouse_range = NULL;
    input-> suspend_mouse= NULL;
    input-> resume_mouse = NULL;
    input-> update_keyboard = keyboard_update;
    input-> get_keyboard_state = keyboard_getstate;
    input-> suspend_keyboard = NULL;
    input-> resume_keyboard = NULL;
    input-> set_leds = NULL;

    input-> wait_event = wait_event;

    init_code_map ();

    return TRUE;
}
```

5、 GetValueFromEtc

```
typedef struct _ETC_S
{
    /** Allocated number of sections */
    int sect_nr_alloc ;
    /** Number of sections */
    int section_nr ;
    /** Pointer to section arrays */
    PETCSECTION   sections ;
} ETC_S ;
typedef struct _ETCSECTION
{
    /** Allocated number of keys */
    int key_nr_alloc ;
    /** Key number in the section */
    int key_nr ;
    /** Name of the section */
    char *name;
    /** Array of keys */
    char**  keys ;
    /** Array of values */
    char**  values ;
} ETCSECTION ;
typedef ETCSECTION * PETCSECTION ;
typedef unsigned int GHANDLE  ;
* Parameter:
 *        pEtcFile: etc file path name.
 *        pSection: Section name.
 *        pKey:          Key name.
 *        pValue:      The buffer will store the value of the key.
 *        iLen:          The max length of value string.
 * Return:
```

```
 *      int                  meaning
 *      ETC_FILENOTFOUND             The etc file not found.
 *      ETC_SECTIONNOTFOUND            The section is not found.
 *      ETC_EKYNOTFOUND           The Key is not found.
 *      ETC_OK               OK.
```

在pSection指向的分区的 pKey段找到 iLen长度的字符串赋给 pValue指向的字符串

```
int GUIAPI GetValueFromEtc (GHANDLE hEtc, const char* pSection,
                            const char* pKey, char* pValue, int iLen)
{
    int i, empty_section = -1;
    ETC_S *petc = ( ETC_S*) hEtc;
    PETCSECTION psect = NULL;

    if (!petc || !pValue)
        return -1;

    for (i=0; i<petc-> section_nr; i++) {
        psect = petc->sections + i;
        if (!psect-> name) {
           empty_section = i;
           continue;
        }

        if (strcmp (psect->name, pSection) == 0) {
            break;
        }
    }

    if (i >= petc-> section_nr) {
        if (iLen > 0)
            return ETC_SECTIONNOTFOUND;
        else{
            if (petc->sect_nr_alloc <= 0)
                return ETC_READONLYOBJ;

            if (empty_section >= 0)
                psect = petc->sections + empty_section;
            else{
                psect = etc_NewSection (petc);
            }

            if (psect->name == NULL) {
                psect->key_nr = 0;
                psect->name = FixStrDup (pSection);
```

```
                    psect->key_nr_alloc  = NR_KEYS_INIT_ALLOC;
                    psect->keys = malloc (sizeof (char* ) * NR_KEYS_INIT_ALLOC);
                    psect->values = malloc (sizeof (char* ) * NR_KEYS_INIT_ALLOC);
                }
            }
        }

    return etc_GetSectionValue (psect, pKey, pValue, iLen);
}
```

3、在 InitGUI （）函数中设置空闲时段的运行函数： SetDskIdleHandler

```
static inline void SetDskIdleHandler (IDLEHANDLER  idle_handler)
{
    __mg_dsk_msg_queue-> OnIdle  = idle_handler;
}
SetDskIdleHandler (IdleHandler4StandAlone);
```

此处将其设置为 **IdleHandler4StandAlone** 。

4、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) {
            case POLLIN:
                temp = &rset;
                break;
            case POLLOUT:
                temp = wsetptr;
                break;
            case POLLERR:
                temp = esetptr;
```

```
                    break;
                }
            }
        }
        return (n > 0);
}
```

5、输入等待事件： IAL_WaitEvent

1、输入引擎的选用

```
#define   IAL_WaitEvent              (*__mg_cur_input->wait_event)
```

当前为 qvfb 输入引擎的 wait_event，位于 qvfbial.c 中

2、相关数据结构

```
typedef struct
  {
    /* XPG4.2 requires this member name.    Otherwise avoid the name
       from the global namespace.    */
#ifdef __USE_XOPEN
    __fd_mask fds_bits[__FD_SETSIZE / __NFDBITS];
# define __FDS_BITS(set) ((set)->fds_bits)
#else
    __fd_mask __fds_bits[__FD_SETSIZE / __NFDBITS];
# define __FDS_BITS(set) ((set)->__fds_bits)
#endif
  } fd_set;
```

FD_ZERO,FD_SET ,FD_CLR,FD_ISSET:

FD_ZERO(fd_set *fdset); 将指定的文件描述符集清空，在对文件描述符集合进行设置前，必须对其进行初始化，如果不清空，由于在系统分配内存空间后，通常并不作清空处理，所以结果是不可知的。

FD_SET(fd_set *fdset); 用于在文件描述符集合中增加一个新的文件描述符。

FD_CLR(fd_set *fdset); 用于在文件描述符集合中删除一个文件描述符。

FD_ISSET(int fd,fd_set *fdset); 用于测试指定的文件描述符是否在该集合中。

（'fd_set') 是一组文件描述符 (fd) 的集合。由于 fd_set 类型的长度在不同平台上不同，因此应该用一组标准的宏定义来处理此类变量：

```
fd_set set;
FD_ZERO(&set); /*        将set 清零  */
FD_SET(fd, &set); /*       将fd 加入 set */
FD_CLR(fd, &set); /*       将fd 从set 中清除  */
FD_ISSET(fd, &set); /*       如果 fd 在set 中则真   */
```

struct timeval 结构：

```
struct timeval{
        long tv_sec;//second
        long tv_usec;//minisecond
```

```
 }

Static __inline__ void __set_bit(int nr , volatile void * addr)
{
  __asm__(
  "btsl %1,%0"
  :"=m" (ADDR)
  :"Ir" (nr));
}
```

btsl 功能是将 (*addr) 的第 nr 位设为 1。第一个占位符 %0 与 C 语言变量 ADDR 对应，第二个占位符 %1 与 C 语言变量 nr 对应。因此上面的汇编语句代码与下面的伪代码等价：btsl nr , ADDR , 该指令的两个操作数不能全是内存变量，因此将 nr 的限定字符串指定为 “ Ir ”，将 nr 与立即数或者寄存器相关联，这样两个操作数中只有 ADDR 为内存变量。汇编语句 btrl 的作用是对操作数的一个位进行清除。 btsl 的作用是对操作数的一位置 1 。

3、相关函数： **select**

int select (int nfds, fd_set *read-fds, fd_set *write-fds, fd_set *except-fds, struct timeval *timeout)

本函数用于确定一个或多个套接口的状态。 对每一个套接口， 调用者可查询它的可读性、可写性及错误状态信息。用 fd_set 结构来表示一组等待检查的套接口。在调用返回时，这个结构存有满足一定条件的套接口组的子集，并且 select() 返回满足条件的套接口的数目。有一组宏可用于对 fd_set 的操作。

timeout 参数控制 select() 完成的时间。若 timeout 参数为空指针，则 select() 将一直阻塞到有一个描述字满足条件。否则的话， timeout 指向一个 timeval 结构，其中指定了 select() 调用在返回前等待多长时间。如果 timeval 为{0,0} ，则 select() 立即返回，这可用于探询所选套接口的状态。如果处于这种状态，则 select() 调用可认为是非阻塞的，且一切适用于非阻塞调用的假设都适用于它。

使用 select 函数的过程一般是：

先调用宏 FD_ZERO 将指定的 fd_set 清零，然后调用宏 FD_SET 将需要测试的 fd 加入 fd_set ，接着调用函数 select 测试 fd_set 中的所有 fd，最后用宏 FD_ISSET 检查某个 fd 在函数 select 调用后，相应位是否仍然为 1。

select() 调用的返回值：

（1）正常情况下返回处于就绪状态并且已经包含在 fd_set 结构中的描述字总数；

（2）如果超时则返回 0；

（3）否则的话，返回 -1 。有以下几种错误情况：

EBADF One of the file descriptor sets specified an invalid file descriptor.

EINTR The operation was interrupted by a signal. .

EINVAL The timeout argument is invalid; one of the components is negative or too large.

4、**wait_event** 函数分析

函数参数：

（1）第一个参数表示等待事件的类型，在此为 IAL_MOUSEEVENT|IAL_KEYEVENT，表示等待的事件类型为鼠标事件或者键盘事件。

（2）第二个参数表示需要监视的最大的文件描述符值 +1 。

（3）第三个参数表示需要检测的可读文件描述符的集合。

（4）第四个参数表示可写文件描述符的集合

（5）第五个参数表示异常文件描述符的集合

（6）第六个参数 struct timeval 结构用于描述一段时间长度，如果在这个时间内，需要监视的描述符没有事件发生则函数返回，返回值为 0。

函数返回值：

（**1**）如果处于就绪状态并且已经包含在 fd_set 结构中的描述符总数大于 0，返回 retvalue= IAL_MOUSEEVENT| IAL_KEYEVENT ，此值大于 0

（2）e<0，表示出现错误，返回 -1

```
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
}
```

InitGUI () InitLWEvent () InitIAL (void)(确定输入输出引擎，当前为 QVFB) IAL_InitInput (__mg_cur_input, mdev, mtype)(当前为 InitQVFBInput)

6 、解析事件 ParseEvent

1、数据结构：

```
typedef struct _LWEVENT
{
        int type;
        int count;
        DWORD  status;
        LWEVENTDATA    data;
} LWEVENT  ;
typedef LWEVENT  *  PLWEVENT  ;

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

typedef struct _KEYEVENT {
        int event;
        int scancode;
        DWORD  status;
} KEYEVENT ;
typedef KEYEVENT * PKEYEVENT ;

typedef struct _MOUSEEVENT {
        int event;
        int x;
        int y;
        DWORD status;
} MOUSEEVENT ;
typedef MOUSEEVENT * PMOUSEEVENT ;
```

2、函数分析

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
        }
        else if (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);
        }
    }
    else if (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) {
                        caseSCANCODE_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;

                        caseSCANCODE_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 ;

                        caseSCANCODE_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
                        caseSCANCODE_LEFTCONTROL:
                            control1 = make;
                            break ;

                        caseSCANCODE_RIGHTCONTROL:
                            control2 = make;
                            break ;

                        caseSCANCODE_LEFTSHIFT:
                            shift1 = make;
                            break ;

                        caseSCANCODE_RIGHTSHIFT:
                            shift2 = make;
                            break ;

                        caseSCANCODE_LEFTALT:
                            alt1 = make;
                            break ;

                        caseSCANCODE_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、将消息加入到消息队列中

```
/* 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;
}
```