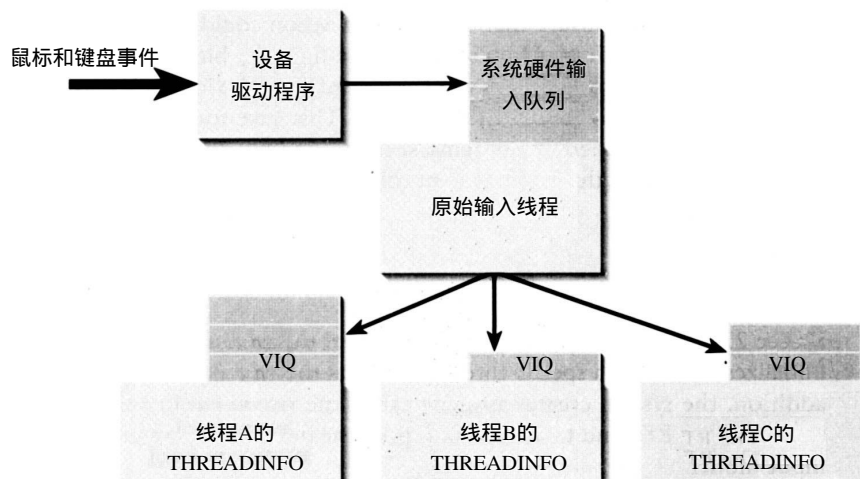


第27章 硬件输入模型和局部输入状态

本章将讨论系统的硬件输入模型。重点将考察按键和鼠标事件是如何进入系统并发送给适当的窗口过程的。微软设计输入模型的一个主要目标就是为了保证一个线程的动作不要对其他线程的动作产生不好的影响。这里是一个 16 位 Windows 中的例子：如果一个任务引起一个死循环，所有的任务都被挂起，不能再响应用户。用户只能重新启动计算机。这样就给一个单个的任务太多的控制。强壮的操作系统，例如 Windows 2000 和 Windows 98，不会使一个挂起的线程妨碍系统中其他线程接收硬件输入。

27.1 原始输入线程

图 27-1 概括描述了系统的硬件输入模型。当系统初始化时，要建立一个特殊的线程，即原始输入线程（raw input thread, RIT）。此外，系统还要建立一个队列，称为系统硬件输入队列（System hardware input queue, SHIQ）。RIT 和 SHIQ 构成系统硬件输入模型的核心。



件输入事件的响应。

那么RIT怎么才能知道要向哪一个线程的虚拟输入队列里增加硬件输入消息？对鼠标消息，RIT只是确定是哪一个窗口在鼠标光标之下。利用这个窗口，RIT调用GetWindowThreadProcessId来确定是哪个线程建立了这个窗口。返回的线程ID指出哪一个线程应该得到这个鼠标消息。

对按键硬件事件的处理稍有不同。在任何给定的时刻，只有一个线程同RIT“连接”。这个线程称为前景线程（foreground thread），因为它建立了正在与用户交互的窗口，并且这个线程的窗口相对于其他线程所建立的窗口来说处在画面中的前景。

当一个用户在系统上登录时，Windows Explorer进程让一个线程建立相应的任务栏（taskbar）和桌面。这个线程连接到RIT。如果你又要产生Calculator，那么就又有有一个线程来建立一个窗口，并且这个线程变成连接到RIT的线程。注意现在Windows Explorer的线程不再与RIT连接，因为在一个时刻只能有一个线程同RIT连接。当一个按键消息进入SHIQ时，RIT就被唤醒，将这个事件转换成适当的按键消息，并将消息放入与RIT连接的线程的虚拟输入队列。

不同的线程是如何连接到RIT的呢？我们已经说过，当产生一个进程时，这个进程的线程可以建立一个窗口。这个窗口处于前景，其建立窗口的线程同RIT相连接。另外，RIT还要负责处理特殊的键组合，如Alt+Tab、Alt+Esc和Ctrl+Alt+Del等。因为RIT在内部处理这些键组合，就可以保证用户总能用键盘激活窗口。应用程序不能够拦截和废弃这些键组合。当用户按动了某个特殊的键组合时，RIT激活选定的窗口，并将窗口的线程连接到RIT。Windows也提供激活窗口的功能，使窗口的线程连接到RIT。这些功能在本章后面讨论。

从图27-1中可以看到如何保护线程，避免相互影响的。如果RIT向窗口 B_1 或窗口 B_2 发送一个消息，消息到达线程B的虚拟输入队列。在处理消息时，线程B在与五个内核对象同步时可能会进入死循环或死锁。如果发生这种情况，线程仍然同RIT连接在一起，并且可能有更多的消息要增加到线程的虚拟输入队列中。

这种情况下，用户会发现窗口 B_1 和 B_2 都没有反应，可能想切换到窗口 A_1 。为了做这种切换，用户按Alt+Tab。因为是RIT处理Alt+Tab按键组合，所以用户总能切换到另外的窗口，不会有什么问题。在选定窗口 A_1 之后，线程A就连接到RIT。这个时候，用户就可以对窗口 A_1 进入输入，尽管线程及其窗口都没有响应。

27.2 局部输入状态

- 哪一个窗口有鼠标捕获。
- 鼠标光标的形状。
- 鼠标光标的可见性。

由于每个线程都有自己的输入状态变量，每个线程都有不同的焦点窗口、鼠标捕获窗口等概念。从一个线程的角度来看，或者它的某个窗口拥有键盘焦点，或者系统中没有窗口拥有键盘焦点；或者它的某个窗口拥有鼠标捕获，或者系统中没有窗口拥有鼠标捕获，等等。读者会想到，这种隔离应该有一些细节，对此我们将在后面讨论。

27.2.1 键盘输入与焦点

我们已经知道，RIT使用户的键盘输入流向一个线程的虚拟输入队列，而不是流向一个窗口。RIT将键盘事件放入线程的虚拟输入队列时不用涉及具体的窗口。当这个线程调用 GetMessage 时，键盘事件从队列中移出并分派给当前有输入焦点的窗口。（由该线程所建立）。图27-2说明了这个处理过程。

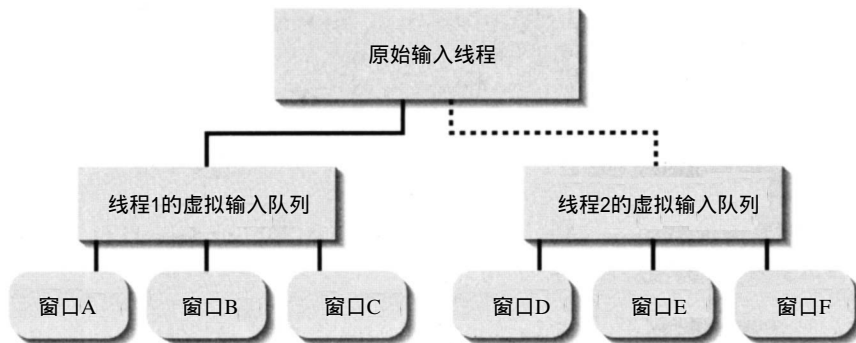


图27-2 RIT将用户的键盘输入导向一个线程的虚拟输入队列

线程1当前正在从RIT接收输入，用窗口A、窗口B或窗口C的句柄作参数调用 SetFocus 会引起焦点改变。失去焦点的窗口除去它的焦点矩形或隐藏它的插入符号，获得焦点的窗口画出焦点矩形或显示它的插入符号。

假定线程1仍然从RIT接收输入，并用窗口E的句柄作为参数调用 SetFocus。这种情况下，系统仍然将输入事件放入线程1的虚拟输入队列，而不是放入线程2的虚拟输入队列。在线程1调用 GetMessage 时，系统会从线程1的虚拟输入队列中移出事件并分派给窗口E。

的线程不一样，那么，对于建立失去焦点窗口的线程，要更新它的局部输入状态变量，说明它没有窗口拥有焦点。这时调用GetFocus将返回NULL，这会使线程知道当前没有窗口拥有焦点。

函数SetActiveWindow激活系统中一个最高层（top-level）的窗口，并对这个窗口设定焦点：

```
HWND SetActiveWindow(HWND hwnd);
```

同SetFocus函数一样，如果调用线程没有创建作为函数参数的窗口，则这个函数什么也不做。

与SetActiveWindow相配合的函数是GetActiveWindow函数：

```
HWND GetActiveWindow();
```

这个函数的功能同GetFocus函数差不多，不同之处是它返回由调用线程的局部输入状态变量所指出的活动窗口的句柄。当活动窗口属于另外的线程时，GetActiveWindow返回NULL。

其他可以改变窗口的Z序（Z-order）、活动状态和焦点状态的函数还包括BringWindowToTop和SetWindowPos：

```
BOOL BringWindowToTop(HWND hwnd);
```

```
BOOL SetWindowPos(  
    HWND hwnd,  
    HWND hwndInsertAfter,  
    int x,  
    int y,  
    int cx,  
    int cy,  
    UINT fuFlags);
```

这两个函数功能相同（实际上，BringWindowToTop函数在内部调用SetWindowPos，以HWND_TOP作为第二个参数）。如果调用这两个函数的线程没有连接到RIT，则函数什么也不做。如果调用这些函数的线程同RIT相连接，系统就会激活相应的窗口。注意即使调用线程不是建立这个窗口的线程，也同样有效。这意味着，这个窗口变成活动的，并且建立这个窗口的线程被连接到RIT。这也引起调用线程和新连接到RIT的线程的局部输入状态变量被更新。

有时候，一个线程想让它的窗口成为屏幕的前景。例如，有可能会利用Microsoft Outlook安排一个会议。在会议开始前的半小时，Outlook弹出一个对话框提醒用户会议将要开始。如果Outlook的线程没有连接到RIT，这个对话框就会藏在其他窗口的后面，有可能看不见它。因

为了制止这种现象，微软对 `SetForegroundWindow` 函数增加了更多的智能。特别规定，仅当调用一个函数的线程已经连接到 RIT 或者当前与 RIT 相连接的线程在一定的时间内（这个时间量由 `SystemParametersInfo` 函数和 `SPI_SETFOREGROUND_LOCKTIMEOUT` 值来控制）没有收到任何输入，这个函数才有效。另外，如果有一个菜单是活动的，这个函数就失效。

如果不允许 `SetForegroundWindow` 将窗口移到前景，它会闪烁该窗口的标题栏和任务条上该窗口的按钮。用户看到任务条按钮闪烁，就知道该窗口想得到用户的注意。用户应该手工激活这个窗口，看一看要报告什么信息。还可以用 `SystemParametersInfo` 函数和 `SPI_SETFOREGROUND_FLASHCOUNT` 值来控制闪烁。

由于这些新的内容，系统又提供了另外一些函数。如果调用 `AllowSetForegroundWindow` 的线程能够成功调用 `SetForegroundWindow`，第一个函数（见下面所列）可使指定进程的一个线程成功调用 `SetForegroundWindow`。为了使任何进程都可以在你的线程的窗口上弹出一个窗口，指定 `ASFW_ANY` (定义为 -1) 作为 `dwProcessId` 参数：

```
BOOL AllowSetForegroundWindow(DWORD dwProcessId);
```

此外，线程可以锁定 `SetForegroundWindow` 函数，使它总是失效的。方法是调用 `LockSetForegroundWindow`。

```
BOOL LockSetForegroundWindow(UINT uLockCode);
```

对 `uLockCode` 参数可以指定 `LSFW_LOCK` 或者 `LSFW_UNLOCK`。当一个菜单被激活时，系统在内部调用这个函数，这样一个试图跳到前景的窗口就不能关闭这个菜单。Windows Explorer 在显示 Start 菜单时，需要明确地调用这些函数，因为 Start 菜单不是一个内置菜单。当用户按了 Alt 键或者将一个窗口拉到前景时，系统自动解锁 `SetForegroundWindow` 函数。这可以防止一个程序一直对 `SetForegroundWindow` 函数封锁。

关于键盘管理和局部输入状态，其他的内容是同步键状态数组。每个线程的局部输入状态变量都包含一个同步键状态数组，但所有的线程要共享一个同步键状态数组。这些数组反映了在任何给定时刻键盘所有键的状态。利用 `GetAsyncKeyState` 函数可以确定用户当前是否按下了键盘上的一个键：

```
SHORT GetAsyncKeyState(int nVirtKey);
```

参数 `nVirtKey` 指出要检查键的虚键代码。结果的高位指出该键当前是否被按下（是为 1，否为 0）。笔者在处理一个消息时，常用这个函数来检查用户是否释放了鼠标主按钮。为函数参数赋一个虚键值 `VK_LBUTTON` 并等待返回值的高位成为 0。注意 如果调用函数的线程不是

建立的窗口上，鼠标光标就可见了。

鼠标光标管理的另一个方面是使用 ClipCursor 函数将鼠标光标剪贴到一个矩形区域。

```
BOOL ClipCursor(CONST RECT *prc);
```

这个函数使鼠标被限制在一个由 prc 参数指定的矩形区域内。当一个程序调用 ClipCursor 函数时，系统该做些什么呢？允许剪贴鼠标光标可能会对其他线程产生不利影响，而不允许剪贴鼠标光标又会影响调用线程。微软实现了一种折衷的方案。当一个线程调用这个函数时，系统将鼠标光标剪贴到指定的矩形区域。但是，如果同步激活事件发生（当用户点击了其他程序的窗口，调用了 SetForegroundWindow，或按了 Ctrl+Esc 组合键），系统停止剪贴鼠标光标的移动，允许鼠标光标在整个屏幕上自由移动。

现在我们再讨论鼠标捕获。当一个窗口“捕获”鼠标（通过调用 SetCapture）时，它要求所有的鼠标消息从 RIT 发到调用线程的虚拟输入队列，并且所有的鼠标消息从虚拟输入队列发到设置捕获的窗口。在程序调用 ReleaseCapture 之前，要一直继续这种鼠标消息的捕获。

鼠标的捕获必须同系统的强壮性折衷，也只能是一种折衷。当一个程序调用 SetCapture 时，RIT 将所有鼠标消息放入线程的虚拟输入队列。SetCapture 还要为调用 SetCapture 的线程设置局部输入状态变量。

通常一个程序在用户按一个鼠标按钮时调用 SetCapture。但是即使鼠标按钮没有被按下，也没有理由说一个线程不能调用 SetCapture。如果当一个鼠标按下时调用 SetCapture，捕获在全系统范围内执行。但当系统检测出没有鼠标按钮按下时，RIT 不再将鼠标消息只发往线程的虚拟输入队列，而是将鼠标消息发往与鼠标光标所在的窗口相联系的输入队列。这是不做鼠标捕获时的正常行为。

但是，最初调用 SetCapture 的线程仍然认为鼠标捕获有效。因此，每当鼠标处于有捕获设置的线程所建立的窗口时，鼠标消息将发往这个线程的捕获窗口。换言之，当用户释放了所有的鼠标按钮时，鼠标捕获不再在全系统范围内执行，而是在一个线程的局部范围内执行。

此外，如果用户想激活一个其他线程所建立的窗口，系统自动向设置捕获的线程发送鼠标按钮按下和鼠标按钮放开的消息。然后系统更新线程的局部输入状态变量，指出该线程不再具有鼠标捕获。很明显，通过这种实现方式，微软希望鼠标点击和拖动是使用鼠标捕获的最常见理由

27.3 将虚拟输入队列同局部输入状态挂接在一起

从上面的讨论我们可以看出这个输入模型是强壮的，因为每个线程都有自己的局部输入状态环境，并且在必要时每个线程可以连接到 RIT或从RIT断开。有时候，我们可能想让两个或多个线程共享一组局部输入状态变量及一个虚拟输入队列。

可以利用 AttachThreadInput函数来强制两个或多个线程共享同一个虚拟输入队列和一组局部输入状态变量：

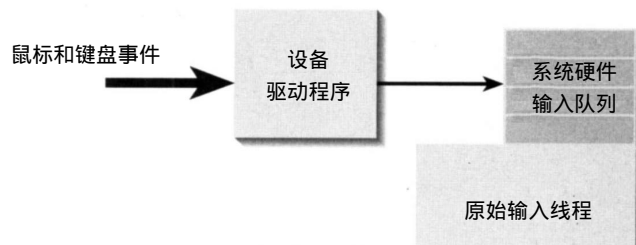
```
BOOL AttachThreadInput(  
    DWORD idAttach,  
    DWORD idAttachTo,  
    BOOL fAttach);
```

函数的第一个参数idAttach，是一个线程的ID，该线程所包含的虚拟输入队列（以及局部输入状态变量）是你不想再使用的。第二个参数 idAttachTo，是另一个线程的ID，这个线程所包含的虚拟输入队列（和局部输入状态变量）是想让两个线程共享的。第三个参数 fAttach，当想让共享发生时，被设置为 TRUE，当想把两个线程的虚拟输入队列和局部输入状态变量分开时，设定为FALSE。可以通过多次调用 AttachThreadInput函数让多个线程共享同一个虚拟输入队列和局部输入状态变量。

我们再考虑前面的例子，假定线程 A调用AttachThreadInput，传递线程 A的ID作为第一个参数，线程B的ID作为第二个参数，TRUE作为最后一个参数：

```
SHORT GetKeyState(int nVirtKey);
```

如图27-3所示，现在每个发往窗口 A1、窗口 B1或窗口 B2的硬件输入事件都将添加到线程B的虚拟输入队列中。线程 A的虚拟输入队列将不再接收输入事件，除非再一次调用 AttachThreadInput并传递FALSE作为最后一个参数，将两个线程的输入队列分开。



当将两个线程的输入都挂接在一起时，就使线程共享单一的虚拟输入队列和同一组局部输入状态变量。但线程仍然使用自己的登记消息队列、发送消息队列、应答消息队列和唤醒标志（见第26章的讨论）。

如果让所有的线程都共享一个输入队列，就会严重削弱系统的强壮性。如果某一个线程接收一个按键消息并且挂起，其他的线程就不能接收任何输入了。所以应该尽量避免使用 `AttachThreadInput` 函数。

在某些情况下，系统隐式地将两个线程挂接在一起。第一种情况是当一个线程安装一个日志记录挂钩（journal record hook）或日志播放挂钩（journal playback hook）的时候。当挂钩被卸载时，系统自动恢复所有线程，这样线程就可以使用挂钩安装前它们所使用的相同输入队列。

当一个线程安装一个日志记录挂钩时，它是让系统将用户输入的所有硬件事件都通知它。这个线程通常将这些信息保存或记录在一个文件上。因用户的输入必须按进入的次序来记录，所以系统中每个线程要共享一个虚拟输入队列，使所有的输入处理同步。

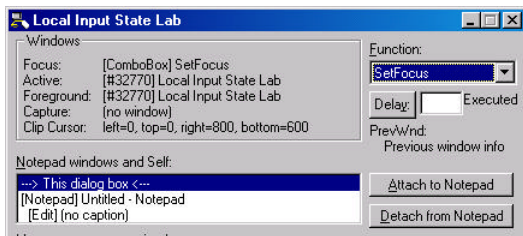
还有一些情况，系统会代替你隐式地调用 `AttachThreadInput`。假定你的程序建立了两个线程。第一个线程建立了一个对话框。在这个对话框建立之后，第二个线程调用 `GreatWindow`，使用 `WS_CHILD` 风格，并向这个子窗口的双亲传递对话框的句柄。系统用子窗口的线程调用 `AttachThreadInput`，让子窗口的线程使用对话框线程所使用的输入队列。这样就使对话框的所有子窗口之间对输入强制同步。

27.3.1 LISLab 示例程序

LISLab 程序（“27LISLab.exe”）清单列在清单 27-1 上。这是一个实验室，可以用它来实验局部输入状态。这个程序的源代码和资源文件在本书所附光盘的 27-LISLab 目录下。

为了用局部输入状态做实验，需要两个线程作为实验品。LISLab 进程有一个线程，这里选择 Notepad 的线程作为另一个。如果当 LISLab 启动时 Notepad 没有在运行，LISLab 将启动 Notepad。在 LISLab 初始化之后，就会见到图 27-4 所示的对话框。

这个窗口的左上角是 Windows 编辑器




```

Module: LISLab.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/
#define TIMER_DELAY (500) // Half a second

UINT_PTR g_uTimerId = 1;
int g_nEventId = 0;
DWORD g_dwEventTime = 0;
HWND g_hwndSubject = NULL;
HWND g_hwndNotepad = NULL;

////////////////////////////////////

void CalcWndText(HWND hwnd, PTSTR szBuf, int nLen) {

    TCHAR szClass[50], szCaption[50], szBufT[150];

    if (hwnd == (HWND) NULL) {
        _tcscpy(szBuf, TEXT("(no window)"));
        return;
    }

    if (!IsWindow(hwnd)) {
        _tcscpy(szBuf, TEXT("(invalid window)"));
        return;
    }

    GetClassName(hwnd, szClass, chDIMOF(szClass));
    GetWindowText(hwnd, szCaption, chDIMOF(szCaption));

    wprintf(szBufT, TEXT("[%s] %s"), (PTSTR) szClass,
        (*szCaption == 0) ? (PTSTR) TEXT("(no caption)") : (PTSTR) szCaption);
    _tcsncpy(szBuf, szBufT, nLen - 1);
    szBuf[nLen - 1] = 0; // Force zero-terminated string
}

```

```
void WalkWindowTreeRecurse(PWALKWINDOWTREEDATA pWWT) {

    if (!IsWindow(pWWT->hwndParent))
        return;

    pWWT->nLevel++;
    const int nIndexAmount = 2;

    for (pWWT->iBuf = 0; pWWT->iBuf < pWWT->nLevel * nIndexAmount; pWWT->iBuf++)
        pWWT->szBuf[pWWT->iBuf] = TEXT(' ');

    CalcWndText(pWWT->hwndParent, &pWWT->szBuf[pWWT->iBuf],
        chDIMOF(pWWT->szBuf) - pWWT->iBuf);
    pWWT->nIndex = ListBox_AddString(pWWT->hwndLB, pWWT->szBuf);
    ListBox_SetItemData(pWWT->hwndLB, pWWT->nIndex, pWWT->hwndParent);

    HWND hwndChild = GetFirstChild(pWWT->hwndParent);
    while (hwndChild != NULL) {
        pWWT->hwndParent = hwndChild;
        WalkWindowTreeRecurse(pWWT);
        hwndChild = GetNextSibling(hwndChild);
    }

    pWWT->nLevel--;
}
```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```
void WalkWindowTree(HWND hwndLB, HWND hwndParent) {

    WALKWINDOWTREEDATA WWT;
    WWT.hwndLB = hwndLB;
    WWT.hwndParent = hwndParent;
    WWT.nLevel = -1;

    WalkWindowTreeRecurse(&WWT);
}
```

```
ComboBox_AddString(hwndT, TEXT("SetFocus"));
ComboBox_AddString(hwndT, TEXT("SetActiveWindow"));
ComboBox_AddString(hwndT, TEXT("SetForegroundWnd"));
ComboBox_AddString(hwndT, TEXT("BringWindowToTop"));
ComboBox_AddString(hwndT, TEXT("SetWindowPos-TOP"));
ComboBox_AddString(hwndT, TEXT("SetWindowPos-BTM"));
ComboBox_SetCurSel(hwndT, 0);

// Fill the windows list box with our window
hwndT = GetDlgItem(hwnd, IDC_WNDS);
ListBox_AddString(hwndT, TEXT("---> This dialog box <---"));

ListBox_SetItemData(hwndT, 0, hwnd);
ListBox_SetCurSel(hwndT, 0);

// Now add Notepad's windows
g_hwndNotepad = FindWindow(TEXT("Notepad"), NULL);
if (g_hwndNotepad == NULL) {

    // Notepad isn't running; run it.
    STARTUPINFO si = { sizeof(si) };
    PROCESS_INFORMATION pi;
    TCHAR szCommandLine[] = TEXT("Notepad");
    if (CreateProcess(NULL, szCommandLine, NULL, NULL, FALSE, 0,
        NULL, NULL, &si, &pi)) {

        // Wait for Notepad to create all its windows.
        WaitForInputIdle(pi.hProcess, INFINITE);
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
        g_hwndNotepad = FindWindow(TEXT("Notepad"), NULL);
    }
}

WalkWindowTree(hwndT, g_hwndNotepad);

return(TRUE);
}
```

```

g_dwEventTime = GetTickCount() + 1000 *
    GetDlgItemInt(hwnd, IDC_DELAY, NULL, FALSE);
hwndT = GetDlgItem(hwnd, IDC_WNDS);
g_hwndSubject = (HWND)
    ListBox_GetItemData(hwndT, ListBox_GetCurSel(hwndT));
g_nEventId = ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_WNDFUNC));
SetWindowText(GetDlgItem(hwnd, IDC_EVENTPENDING), TEXT("Pending"));
break;
case IDC_THREADATTACH:
    AttachThreadInput(GetWindowThreadProcessId(g_hwndNotepad, NULL),
        GetCurrentThreadId(), TRUE);
    break;

case IDC_THREADDETACH:
    AttachThreadInput(GetWindowThreadProcessId(g_hwndNotepad, NULL),
        GetCurrentThreadId(), FALSE);
    break;

case IDC_REMOVECLIPRECT:
    ClipCursor(NULL);
    break;

case IDC_HIDECURSOR:
    ShowCursor(FALSE);
    break;

case IDC_SHOWCURSOR:
    ShowCursor(TRUE);
    break;

case IDC_INFINITELOOP:
    SetCursor(LoadCursor(NULL, IDC_NO));
    for (;;)
        ;
    break;

case IDC_SETCLIPRECT:
    RECT rc;
    SetRect(&rc, 0, 0, GetSystemMetrics(SM_CXSCREEN) / 2

```



```
void Dlg_OnLButtonUp(HWND hwnd, int x, int y, UINT keyFlags) {
```

```
    TCHAR szBuf[100];
```

```
    wsprintf(szBuf,
```

```
        TEXT("Capture=%-3s, Msg=LButtonUp,  x=%5d, y=%5d"),
```

```
        (GetCapture() == NULL) ? TEXT("No") : TEXT("Yes"), x, y);
```

```
    AddStr(GetDlgItem(hwnd, IDC_MOUSEMSG), szBuf);
```

```
}
```

```
////////////////////////////////////
```

```
void Dlg_OnMouseMove(HWND hwnd, int x, int y, UINT keyFlags) {
```

```
    TCHAR szBuf[100];
```

```
    wsprintf(szBuf, TEXT("Capture=%-3s, Msg=MouseMove,  x=%5d, y=%5d"),
```

```
        (GetCapture() == NULL) ? TEXT("No") : TEXT("Yes"), x, y);
```

```
    AddStr(GetDlgItem(hwnd, IDC_MOUSEMSG), szBuf);
```

```
}
```

```
////////////////////////////////////
```

```
void Dlg_OnTimer(HWND hwnd, UINT id) {
```

```
    TCHAR szBuf[100];
```

```
    CalcWndText(GetFocus(), szBuf, chDIMOF(szBuf));
```

```
    SetWindowText(GetDlgItem(hwnd, IDC_WNDFOCUS), szBuf);
```

```
    CalcWndText(GetCapture(), szBuf, chDIMOF(szBuf));
```

```
    SetWindowText(GetDlgItem(hwnd, IDC_WNDCAPTURE), szBuf);
```

```
    CalcWndText(GetActiveWindow(), szBuf, chDIMOF(szBuf));
```

```
    SetWindowText(GetDlgItem(hwnd, IDC_WNDACTIVE), szBuf);
```



```
        break;

    case 1: // SetActiveWindow
        g_hwndSubject = SetActiveWindow(g_hwndSubject);
        break;

    case 2: // SetForegroundWindow
        hwndT = GetForegroundWindow();
        SetForegroundWindow(g_hwndSubject);
        g_hwndSubject = hwndT;
        break;

    case 3: // BringWindowToTop
        BringWindowToTop(g_hwndSubject);
        break;

    case 4: // SetWindowPos w/HWND_TOP
        SetWindowPos(g_hwndSubject, HWND_TOP, 0, 0, 0, 0,
            SWP_NOMOVE | SWP_NOSIZE);
        g_hwndSubject = (HWND) 1;
        break;

    case 5: // SetWindowPos w/HWND_BOTTOM
        SetWindowPos(g_hwndSubject, HWND_BOTTOM, 0, 0, 0, 0,
            SWP_NOMOVE | SWP_NOSIZE);
        g_hwndSubject = (HWND) 1;
        break;
}

if (g_hwndSubject == (HWND) 1) {
    SetWindowText(GetDlgItem(hwnd, IDC_PREVWND), TEXT("Can't tell.));
} else {
    CalcWndText(g_hwndSubject, szBuf, chDIMOF(szBuf));
    SetWindowText(GetDlgItem(hwnd, IDC_PREVWND), szBuf);
}

g_hwndSubject = NULL; g_nEventId = 0; g_dwEventTime = 0;
SetWindowText(GetDlgItem(hwnd, IDC_EVENTPENDING), TEXT("Executed"));
}
```

```

        chHANDLE_DLGMSG(hwnd, WM_RBUTTONDOWN, Dlg_OnRButtonDown);
        chHANDLE_DLGMSG(hwnd, WM_RBUTTONDBLCLK, Dlg_OnRButtonDown);
        chHANDLE_DLGMSG(hwnd, WM_RBUTTONUP, Dlg_OnRButtonUp);

        chHANDLE_DLGMSG(hwnd, WM_TIMER, Dlg_OnTimer);
    }
    return(FALSE);
}

```

```

////////////////////////////////////

```

```

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_LISLAB), NULL, Dlg_Proc);
    return(0);
}

```

```

//////////////////////////////////// End of File //////////////////////////////////

```

LISLab.rc

```

//Microsoft Developer Studio generated-resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

```

```
// remains consistent on all systems.
IDI_LISLAB          ICON    DISCARDABLE    "LISLab.Ico"

////////////////////////////////////
//
// Dialog
//

IDD_LISLAB DIALOG DISCARDABLE 12, 38, 286, 178
STYLE WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Local Input State Lab"
FONT 8, "MS Sans Serif"
BEGIN
    GROUPBOX        "Windows", IDC_STATIC, 4, 0, 192, 56
    LTEXT            "Focus:", IDC_STATIC, 8, 12, 23, 8
    LTEXT            "Focus window info", IDC_WNDFOCUS, 52, 12, 140, 8
    LTEXT            "Active:", IDC_STATIC, 8, 20, 24, 8
    LTEXT            "Active window info", IDC_WNDACTIVE, 52, 20, 140, 8
    LTEXT            "Foreground:", IDC_STATIC, 8, 28, 40, 8
    LTEXT            "Foreground window info", IDC_WNDFOREGROUND, 52, 28, 140, 8
    LTEXT            "Capture:", IDC_STATIC, 8, 36, 29, 8
    LTEXT            "Capture window info", IDC_WNDCAPTURE, 52, 36, 140, 8
    LTEXT            "Clip Cursor:", IDC_STATIC, 8, 44, 39, 8
    LTEXT            "Cursor clipping info", IDC_CLIPCURSOR, 52, 44, 140, 8
    LTEXT            "&Function:", IDC_STATIC, 200, 4, 32, 8
    COMBOBOX         IDC_WNDFUNC, 200, 14, 82, 54, CBS_DROPDOWNLIST | WS_VSCROLL |
        WS_TABSTOP
    PUSHBUTTON       "De&lay:", IDC_FUNCSTART, 200, 30, 26, 14
    EDITTEXT         IDC_DELAY, 228, 30, 24, 12, ES_AUTOHSCROLL
    LTEXT            "Executed", IDC_EVENTPENDING, 252, 30, 32, 10
    LTEXT            "PrevWnd:", IDC_STATIC, 200, 46, 34, 8
    LTEXT            "Previous window info", IDC_PREVWND, 208, 54, 76, 18
    LTEXT            "&Notepad windows and Self:", IDC_STATIC, 4, 62, 90, 8
    LISTBOX          IDC_WNDS, 4, 72, 192, 32, WS_VSCROLL | WS_TABSTOP
    PUSHBUTTON       "&Attach to Notepad", IDC_THREADATTACH, 200, 72, 80, 12
    PUSHBUTTON       "&Detach from Notepad", IDC_THREADETACH, 200, 88, 80, 12
    LTEXT            "&Mouse messages received:", IDC_STATIC, 4, 102, 89, 8
    LISTBOX          IDC_MOUSEMSG, 4, 112, 192, 32, WS_VSCROLL | WS_TABSTOP
    LTEXT            "Click right mouse button to get context menu (Right-click)"
```

```
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#ifdef    // APSTUDIO_INVOKED
////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_LISLAB, DIALOG
    BEGIN
        RIGHTMARGIN, 283
        BOTTOMMARGIN, 170
    END
END
#endif

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
```

有窗口拥有焦点并且没有窗口是活动的。

然后可以通过改变窗口的焦点来进行实验。首先在 Local Input State Lab对话框右上角的 Function组合框内选择SetFocus。然后键入延迟时间（以秒计），即在调用SetFocus之前你想让LISLab等待的时间。对这个实验，你很可能会指定延迟为 0s。后面将简单介绍如何使用 Delay 字段。

下一步选择一个窗口，作为调用SetFocus时的参数。用 Local Input State Lab对话框左边的 Notepad Windows And Self列表框选择一个窗口。对这个实验，选择列表框中的 [Notepad] Untitled-Notepad。现在已经为调用SetFocus做好准备。只需点击Delay按钮，观察 Windows编组框会发生什么变化。什么也没发生。系统没有执行改变焦点的动作。

如果真想让SetFocus将焦点改变到Notepad，就点击Attach To Notepad按钮。点击这个按钮使LISLab调用下面的函数：

这个调用告诉LISLab的线程去使用Note- pad所使用的虚拟输入队列。另外， LISLab的线程也要与Notepad共享局部输入状态变量。

```
AttachThreadInput(GetWindowThreadProcessId(g_hwndNotepad, NULL),  
GetCurrentThreadId(), TRUE);
```

如果在点击了Attach To Notepad按钮之后，点击Notepad窗口，LISLab的对话框变成图27-5所示的样子。

现在注意，由于两个线程的输入队列是挂接在一起的，LISLab可以服从Notepad所做的窗口焦点改变。图 27-5所示的对话框显示Edit控制框当前具有焦点。如果我们显示Notepad中的File Open对话框，LISLab将继续更新它的显示屏内容，告诉我们哪一个Note-pad窗口具有焦点，哪个窗口是活动的等等。

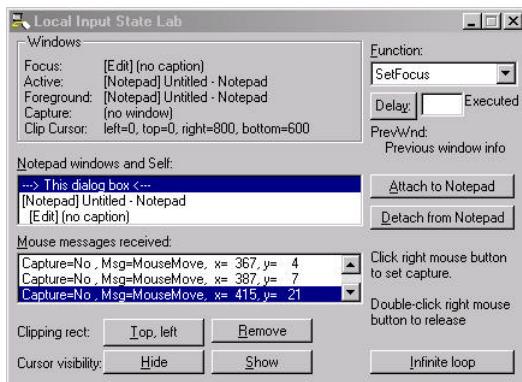


图27-5 点击Notepad窗口后的LISLab对话框

现在再回到 LISLab，点击 Delay按钮，让 SetFocus给Notepad焦点。这一次，对SetFocus的调用成功，因两个线程的输入队列是接在一起的。

读者可以继续实验，通过在 Function组合框中选择不同的函数，分别对SetActiveWindow

不过，RIT仍然同Notepad的线程相“连接”。

关于窗口和焦点还要说明一点：SetFocus函数和SetActiveWindow函数都返回原来拥有焦点或原来活动的窗口的句柄。有关这个窗口的信息显示在LISLab对话框的PrevWnd字段里。而且，LISLab在调用SetForegroundWindow之前，要先调用GetForegroundWindow来取得原来处于前景的窗口的句柄。这些信息也显示在PrevWnd字段。

现在我们对鼠标光标的内容进行实验。每当你在LISLab的对话框上移动鼠标（但没有在它的任何子窗口上移动），鼠标被显示成一个垂直箭头。当鼠标消息发送到这个对话框，消息要添加到Mouse Message Received列表框中。这样你就可以知道何时对话框在接收鼠标消息。如果你将鼠标移出对话框或移到某个子窗口之上，就会发现鼠标消息不再添加到 Mouse Message Received列表框中。

现在将鼠标移往对话框的右部，移在文本Click Right Mouse Button To Set Capture之上，然后点击并按住鼠标右键。这时，LISLab调用SetCapture并传递LISLab对话框的句柄作为参数。注意LISLab更新Windows编组框来反映它拥有鼠标捕获。

不要释放鼠标右键，在LISLab的子窗口上移动鼠标，并观察鼠标消息被添加到列表框里。注意，如果你将鼠标移出LISLab的对话框，LISLab会继续得知鼠标消息。不论你在屏幕上什么位置移动鼠标，鼠标光标都保持垂直箭头形状。

现在我们看一看系统的其他表现。释放鼠标右键，看会发生什么。在LISLab对话框的上部所反映的捕获窗口继续显示LISLab依然认为自己拥有鼠标捕获。如果你将鼠标移出LISLab的对话框，鼠标光标就不再保持垂直箭头的形状，鼠标消息也不再向Mouse Messages Received列表框中添加，你会看到鼠标捕获依然有效，因为所有窗口都使用同一组局部输入状态变量。

当完成对鼠标捕获的实验时，可以使用下面两种办法将其关闭：

- 在Local Input State Lab对话框的任何地方双击鼠标右键，让LISLab安排ReleaseCapture的调用。
- 点击一个由LISLab的线程之外的线程所建立的窗口。这样系统会自动向LISLab的对话框发送鼠标按钮弹起和鼠标按钮按下的消息。

不论使用哪种办法，要观察Windows编组框中的Capture字段是如何变化以反映没有窗口拥有鼠标捕获。

与鼠标有关的实验还有两个：其中之一涉及在一个矩形区域剪贴鼠标光标的移动，另一个涉及鼠标光标的可见性。当点击Top Left按钮时，LISLab执行下面的代码：

点击Hide或Show Cursor按钮，使LISLab执行下面的代码：

```
ShowCursor(FALSE);
```

或

```
ShowCursor(TRUE);
```

当你隐藏了鼠标光标时，如果在LISLab的对话框上移动鼠标时，不会出现鼠标光标。但在这个对话框之外移动鼠标时，鼠标光标又会出现。使用 Show按钮来抵消Hide按钮的效果。注意隐藏光标的效果是要积累的，也就是，如果 5次点击Hide按钮，必须也5次点击Show按钮，才能使光标可见。

最后一个实验是使用Infinite Loop按钮。当点击这个按钮时，LISLab执行下面的代码：

```
SetCursor(LoadCursor(NULL, IDC_NO));  
for (;;) ;
```

第一行代码将鼠标光标改变成一个缺口圆 (slashed circle)，第二行代码执行一个死循环。在点击Infinite Loop按钮之后，LISLab停止响应任何输入。如果在LISLab的对话框上移动鼠标，鼠标光标依然是缺口圆。如果将鼠标移出对话框，光标要改变，以反映它所处窗口的光标。可以用鼠标操作这些其他的窗口。

如果将鼠标移回到LISLab的对话框，系统看到LISLab没有响应，就自动将光标改回最近的形状——缺口圆。可以看到执行死循环的线程对用户是不方便的，但可以因此使用其他窗口。

注意，如果把一个窗口移到挂起的Local Input State Lab对话框，然后再把它移开，系统要发送一个WM_PAINT消息。但系统发现这个线程没有响应。系统为这个没有响应的程序重画窗口。当然，系统不能正确地重画这个窗口，因为系统不知道这个程序是干什么的。所以系统只是抹掉窗口的背景，并重画框架。

现在还有一个问题，如果屏幕上有一个窗口对我们做的任何事情（按键或击鼠标钮）都没有响应。我们如何清除这个窗口？在Windows 98中，必须按Ctrl+Alt+Del来显示图27-6的Close Program窗口。

在Windows 2000里，可以用鼠标右击Taskbar上的程序按钮，或显示图27-7的TaskManager窗口。

然后只要在窗口里选择我们想要结束的程序，在这里是 Local Input State Lab，再点击End

Task按钮。系统将试图用一种温和的方式（通过发送一个 WM_CLOSE消息）来结束LISLab，但发现该程序没有反应。在Windows 98里，这会引起系统显示图27-8的对话框。

在Windows 2000中会显示图27-9的对话框。

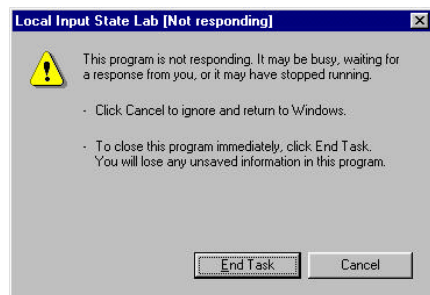


图27-8 Local Input State Lab 对话框

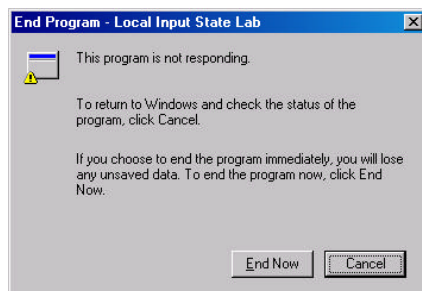


图27-9 End Program 对话框

选择End Task（在Windows 98里）或End Now（在Windows 2000里）使系统强制性地从系统中清除LISLab。Cancel按钮是告诉系统你改变了主意，不再想结束这个程序。这里，我们选择End Task或End Now，从系统中清除LISLab。

这个实验的主要目的是为了说明系统的强壮性。一个程序不可能使操作系统处于这样一个状态——使其他程序不可用。还要注意在结束处理中，Windows 98和Windows 2000都可以自动释放线程所分配的资源，不会造成内存遗漏。

27.3.2 LISWatch示例程序

LISWatch程序（“27LISWatch.exe”）的源程序清单列在清单27-2中。这是一个有用的实用程序，用来监控活动窗口、焦点窗口和鼠标捕获窗口。在本书所附光盘的27-LISWatch目录下是这个程序的源代码和资源文件。

当运行LISWatch，会显示图27-10的对话框。

当这个对话框接收到一个 WM_INITDIALOG消息时，它调用 SetTimer来设置一个计时器，每秒钟激发两次。当收到 WM_TIMER消息，对话框的内容就更新以反映哪个窗口是活动的、哪个窗口拥有焦点、哪个窗口捕获了鼠标。在这个对话框



图27-10 运行LISWatch时显示的对话框

以调用 `GetFocus`、`GetActiveWindow` 和 `GetCapture`，所有这些函数都返回有效的窗口句柄。帮助函数 `CalcWndText` 构造一个字符串，包含每个窗口的类名和窗口标题。然后每个窗口的串在 `LIS Watch` 的对话框中被更新。最后，`Dlg_OnTimer` 在返回之前，再一次调用 `AttachThreadInput`，但这次是将最后一个参数设定为 `FALSE`，这样两个线程的局部输入状态就彼此断开。

前面解释了 `LIS Watch` 的基本内容。然而，我们对 `LIS Watch` 增加了其他一些特性，在这里解释一下。当启动 `LIS Watch` 之后，它要监控系统中任何地方发生的窗口活动的变化。这就是对话框顶部的“全系统范围 (System-wide)”的意思。`LIS Watch` 还可以让你只限于观察一个线程的局部输入状态的变化。利用这种特性，`LIS Watch` 可以向你报告一个线程的确切情况。

为了让 `LIS Watch` 监控一个线程的局部输入状态，所要做的只是在 `LIS Watch` 的窗口上按下鼠标左键，在另外一个线程建立的窗口上拖动鼠标光标，然后释放鼠标按钮。在释放鼠标按钮之后，`LIS Watch` 将全局变量 `g-dwThreadIdAttachTo` 设置成所选线程的 ID。这个线程 ID 将替换 `LIS Watch` 的对话框顶部的“System-wide”。当这个全局变量不是零，`Dlg_OnTimer` 会略微改变它的行为。不是总将它的局部输入状态同前景线程的局部输入挂接在一起，而是将 `LIS Watch` 本身同所选择的线程挂接在一起。用这种方式，`LIS Watch` 调用 `GetActiveWindow`、`GetFocus` 和 `GetCapture` 来反映所选择线程的局部输入状态的情况。

我们来做一个实验。运行 `Calculator`，再用 `LIS Watch` 来选择它的窗口。当激活 `Calculator` 的窗口时，`LIS Watch` 更新它的显示内容，见图 27-11 的对话框。

这里，`Calculator` 的线程 ID 是 `0x000004ec`。当前设定 `LIS Watch` 来监控这一个线程的局部输入状态变化。如果点击 `Calculator` 的任何单选按钮或复选框，`LIS Watch` 都可以显示焦点的变化，因为所有这些窗口都是由线程 `0x000004ec` 建立的。

如果现在再激活一个由另外的程序（这里的例子是 `Notepad`）建立的窗口，`LIS Watch` 的对话框是下面的样子（见图 27-12）。



图27-11 激活Calculator窗口时的
LIS Watch 对话框

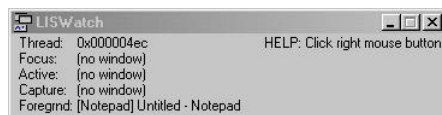


图27-12 由另一个程序激活窗口时显示的
LIS Watch 对话框

```

#include "..\CmnHdr.h"      /* See Appendix A. */
#include <tchar.h>
#include <windowsx.h>
#include "Resource.h"

////////////////////////////////////

#define TIMER_DELAY (500)      // Half a second

UINT_PTR g_uTimerId = 1;
DWORD g_dwThreadIdAttachTo = 0; // 0=System-wide; Non-zero=specific thread

////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_LISWATCH);

    // Update our contents periodically
    g_uTimerId = SetTimer(hwnd, g_uTimerId, TIMER_DELAY, NULL);

    // Make our window on top of all others
    SetWindowPos(hwnd, HWND_TOPMOST, 0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE);
    return(TRUE);
}

////////////////////////////////////

void Dlg_OnRButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y,
    UINT keyFlags) {
    chMB("To monitor a specific thread, click the left mouse button in "
        "the main window and release it in the desired window.\n"
        "--

```

```
SetCursor(LoadCursor(GetModuleHandle(NULL), MAKEINTRESOURCE(IDC_EYES)));
}

////////////////////////////////////

void Dlg_OnLButtonUp(HWND hwnd, int x, int y, UINT keyFlags) {

    if (GetCapture() == hwnd) {

        // If we had mouse capture set, get the ID of the thread that
        // created the window that is under the mouse cursor.
        POINT pt;
        pt.x = LOWORD(GetMessagePos());
        pt.y = HIWORD(GetMessagePos());
        ReleaseCapture();
        g_dwThreadIdAttachTo = GetWindowThreadProcessId(
            ChildWindowFromPointEx(GetDesktopWindow(), pt, CWP_SKIPINVISIBLE),
            NULL);

        if (g_dwThreadIdAttachTo == GetCurrentThreadId()) {
            // The mouse button is released on one of our windows;
            // monitor local-input state on a system-wide basis.
            g_dwThreadIdAttachTo = 0;

        } else {

            // The mouse button is released on a window that our thread didn't
            // create; monitor local input state for that thread only.
            AttachThreadInput(GetCurrentThreadId(), g_dwThreadIdAttachTo, TRUE);

        }
    }
}
```

```
////////////////////////////////////
```

```
_tcsncpy(szBuf, szBufT, nLen - 1);
szBuf[nLen - 1] = 0; // Force zero-terminated string
}

////////////////////////////////////

void Dlg_OnTimer(HWND hwnd, UINT id) {
TCHAR szBuf[100] = TEXT("System-wide");
HWND hwndForeground = GetForegroundWindow();
DWORD dwThreadIdAttachTo = g_dwThreadIdAttachTo;

if (dwThreadIdAttachTo == 0) {

    // If monitoring local input state system-wide, attach our input
    // state to the thread that created the current foreground window.
    dwThreadIdAttachTo =
        GetWindowThreadProcessId(hwndForeground, NULL);
    AttachThreadInput(GetCurrentThreadId(), dwThreadIdAttachTo, TRUE);

} else {

    wsprintf(szBuf, TEXT("0x%08x"), dwThreadIdAttachTo);
}

SetWindowText(GetDlgItem(hwnd, IDC_THREADID), szBuf);

CalcWndText(GetFocus(), szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDFOCUS), szBuf);

CalcWndText(GetActiveWindow(), szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDACTIVE), szBuf);

CalcWndText(GetCapture(), szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDCAPTURE), szBuf);

CalcWndText(hwndForeground, szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDFOREGRND), szBuf);
```



```
    }  
}  
  
////////////////////////////////////  
  
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {  
  
    switch (uMsg) {  
        chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);  
        chHANDLE_DLGMSG(hwnd, WM_COMMAND, Dlg_OnCommand);  
        chHANDLE_DLGMSG(hwnd, WM_TIMER, Dlg_OnTimer);  
        chHANDLE_DLGMSG(hwnd, WM_RBUTTONDOWN, Dlg_OnRButtonDown);  
        chHANDLE_DLGMSG(hwnd, WM_LBUTTONDOWN, Dlg_OnLButtonDown);  
        chHANDLE_DLGMSG(hwnd, WM_LBUTTONUP, Dlg_OnLButtonUp);  
    }  
    return(FALSE);  
}  
  
////////////////////////////////////  
  
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {  
  
    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_LISWATCH), NULL, Dlg_Proc);  
    return(0);  
}  
  
//////////////////////////////////// End of File //////////////////////////////////////
```

LISWatch.rc

```
//Microsoft Developer Studio generated resource script.  
//
```

```

#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif // APSTUDIO_INVOKED
////////////////////////////////////
//
// Dialog
//

IDD_LISWATCH_DIALOG DISCARDABLE 32768, 5, 240, 41
STYLE WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "LISWatch"
FONT 8, "MS Sans Serif"
-----

```

```
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_LISWATCH          ICON      DISCARDABLE      "LISWatch.ico"

////////////////////////////////////
//
// Cursor
//

IDC_EYES                CURSOR  DISCARDABLE      "Eyes.cur"

////////////////////////////////////
//
// DESIGNINFO
//
#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_LISWATCH, DIALOG
    BEGIN
        RIGHTMARGIN, 190
        BOTTOMMARGIN, 10
    END
END
#endif // APSTUDIO_INVOKED

#endif // English (U.S.) resources
////////////////////////////////////

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
..
```