

第12章 纤 程

Microsoft公司给Windows添加了一种纤程，以便能够非常容易地将现有的 UNIX服务器应用程序移植到Windows中。UNIX服务器应用程序属于单线程应用程序（由Windows定义），但是它能够为多个客户程序提供服务。换句话说，UNIX应用程序的开发人员已经创建了他们自己的线程结构库，他们能够使用这种线程结构库来仿真纯线程。该线程包能够创建多个堆栈，保存某些CPU寄存器，并且在它们之间进行切换，以便为客户机请求提供服务。

显然，若要取得最佳的性能，这些 UNIX应用程序必须重新设计，仿真的线程库应该用Windows提供的纯线程来替代。然而，这种重新设计需要花费数月甚至更长的时间才能完成，因此许多公司首先将它们现有的 UNIX代码移植到Windows中，这样就能够将某些应用软件推向Windows市场。

当你将UNIX代码移植到Windows中时，一些问题就会因此而产生。尤其是Windows管理线程的内存栈的方法要比简单地分配内存复杂得多。Windows内存栈开始时的物理存储器的容量比较小，然后根据需要逐步扩大。这个过程在第16章“线程的堆栈”中详细介绍。由于结构化异常处理机制的原因，代码的移植就更加复杂了。

为了能够更快和更正确地将它们的代码移植到Windows中，Microsoft公司在操作系统中添加了纤程。本章将要介绍纤程的概念、负责操作纤程的函数以及如何利用纤程的特性。要记住，如果有设计得更好的使用Windows自身线程的应用程序，那么应该避免使用纤程。

12.1 纤程的操作

首先要注意的一个问题是，实现线程的是Windows内核。操作系统清楚地知道线程的情况，并且根据Microsoft定义的算法对线程进行调度。纤程是以用户方式代码来实现的，内核并不知道纤程，并且它们是根据用户定义的算法来调度的。由于你定义了纤程的调度算法，因此，就内核而言，纤程采用非抢占式调度方式。

需要了解的下一个问题是，单线程可以包含一个或多个纤程。就内核而言，线程是抢占调度的，是正在执行的代码。然而，线程每次执行一个纤程的代码——你决定究竟执行哪个纤程（随着我们讲解的深入，这些概念将会越来越清楚）。

当使用纤程时，你必须执行的第一步操作是将现有的线程转换成一个纤程。可以通过调用ConvertThreadToFiber函数来执行这项操作：

```
PVOID ConvertThreadToFiber(PVOID pvParam);
```

该函数为纤程的执行环境分配相应的内存（约为200字节）。该执行环境由下列元素组成：

- 一个用户定义的值，它被初始化为传递给ConvertThreadToFiber的pvParam参数的值。
- 结构化异常处理链的头。
- 纤程内存栈的最高和最低地址（当将线程转换成纤程时，这也是线程的内存栈）。
- CPU寄存器，包括堆栈指针、指令指针和其他。

当对纤程的执行环境进行分配和初始化后，就可以将执行环境的地址与线程关联起来。该线程被转换成一个纤程，而纤程则在该线程上运行。ConvertThreadToFiber函数实际上返回纤程的执行环境的内存地址。虽然必须在晚些时候使用该地址，但是决不应该自己对该执行环境

数据进行读写操作，因为必要时纤程函数会为你对该结构的内容进行操作。现在，如果你的纤程（线程）返回或调用ExitThread函数，那么纤程和线程都会终止运行。

除非打算创建更多的纤程以便在同一个线程上运行，否则没有理由将线程转换成纤程。若要创建另一个纤程，该线程（当前正在运行纤程的线程）可以调用 CreateFiber函数：

```
PVOID CreateFiber(  
    DWORD dwStackSize,  
    PFIBER_START_ROUTINE pfnStartAddress,  
    PVOID pvParam);
```

CreateFiber首先设法创建一个新内存栈，它的大小由 dwStackSize参数来指明。通常传递的参数是0，按照默认设置，它创建一个内存栈，其大小可以扩展为 1MB，不过开始时有两个存储器页面用于该内存栈。如果设定一个非0值，那么就用设定的大小来保存和使用内存栈。

接着，CreateFiber函数分配一个新的纤程执行环境结构，并对它进行初始化。该用户定义的值被设置为传递给 CreateFiber的pvParam参数的值，新内存栈的最高和最低地址被保存，同时，纤程函数的内存地址（作为 pfnStartAddress参数来传递）也被保存。

PfnStartAddress参数用于设定必须实现的纤程例程的地址，它必须采用下面的原型：

```
VOID WINAPI FiberFunc(PVOID pvParam);
```

当纤程被初次调度时，该函数就开始运行，并且将原先传递给 CreateFiber的pvParam的值传递给它。可以在这个纤程函数中执行想执行的任何操作。但是该函数的原型规定返回值是VOID，这并不是因为返回值没有任何意义，而是因为该函数根本不应该返回。如果纤程确实返回了，那么线程和该线程创建的所有纤程将立即被撤消。

与ConvertThreadToFiber函数一样，CreateFiber函数也返回纤程运行环境的内存地址。但是，与ConvertThreadToFiber不同的是，这个新纤程并不执行，因为当前运行的纤程仍然在执行。在单个线程上，每次只能运行一个纤程。若要使新纤程能够运行，可以调用 Switch To Fiber函数：

```
VOID SwitchToFiber(PVOID pvFiberExecutionContext);
```

Switch To Fiber函数只有一个参数，即 pvFiberExecutionContext，它是上次调用 ConvertThreadToFiber或CreateFiber函数时返回的纤程的执行环境的内存地址。该内存地址告诉该函数要对哪个纤程进行调度。SwitchToFiber函数在内部执行下列操作步骤：

- 1) 它负责将某些当前的CPU寄存器保存在当前运行的纤程执行环境中，包括指令指针寄存器和堆栈指针寄存器。
- 2) 它将上一次保存在即将运行的纤程的执行环境中的寄存器装入CPU寄存器。这些寄存器包括堆栈指针寄存器。这样，当线程继续执行时，就可以使用该纤程的内存栈。
- 3) 它将纤程的执行环境与线程关联起来，线程运行特定的纤程。
- 4) 它将线程的指令指针设置为已保存的指令指针。线程（纤程）从该纤程上次执行的地方开始继续执行。

SwitchToFiber函数是纤程获得CPU时间的唯一途径。由于你的代码必须在相应的时间显式调用SwitchToFiber函数，因此你对纤程的调度可以实施全面的控制。记住，纤程的调度与线程调度毫不相干。纤程运行所依赖的线程始终都可以由操作系统终止其运行。当线程被调度时，当前选定的纤程开始运行，而其他纤程则不能运行，除非显式调用 SwitchToFiber函数。若要撤消纤程，可以调用DeleteFiber函数：

```
VOID DeleteFiber(PVOID pvFiberExecutionContext);
```

该函数用于删除pvFiberExecutionContext参数指明的纤程，当然这是纤程的执行环境的地址。该函数能够释放纤程栈使用的内存，然后撤消纤程的执行环境。但是，如果传递了当前与线程相关联的纤程地址，那么该函数就在内部调用 ExitThread函数，该线程及其创建的所有纤程全部被撤消。

DeleteFiber函数通常由一个纤程调用，以便删除另一个纤程。已经删除的纤程的内存栈将被撤消，纤程的执行环境被释放。注意，纤程与线程之间的差别在于，线程通常通过调用 ExitThread函数将自己撤消。实际上，用一个线程调用 TerminateThread函数来终止另一个线程的运行，是一种不好的方法。如果你确实调用了 TerminateThread函数，系统并不撤消已经终止运行的线程的内存栈。可以利用纤程的这种能力来删除另一个纤程，后面介绍示例应用程序时将说明这是如何实现的。

为了使操作更加方便，还可以使用另外两个纤程函数。一个线程每次可以执行一个纤程，操作系统始终都知道当前哪个纤程与该线程相关联。如果想要获得当前运行的纤程的执行环境的地址，可以调用 GetCurrentFiber函数：

```
PVOID GetCurrentFiber();
```

另一个使用非常方便的函数是GetFiberData：

```
PVOID GetFiberData();
```

前面讲过，每个纤程的执行环境包含一个用户定义的值。这个值使用作为 ConvertThreadToFiber或CreateFiber的pvParam参数而传递的值进行初始化。该值也可以作为纤程函数的参数来传递。GetFiberData只是查看当前执行的纤程的执行环境，并返回保存的值。

无论GetCurrentFiber还是GetFiberData，运行速度都很快，并且通常是作为内蕴函数(intrinsic function)来实现的，这意味着编译器能够为这些函数生成内联代码。

12.2 Counter示例应用程序

后面清单 12-1 中的Counter应用程序（“ 12 Counter.exe ”）使用纤程来实现后台处理。当运行该应用程序时，便出现图 12-1 所示的对话框（建议通过运行该应用程序来了解阅读下面的内容时将会出现什么情况并观察它的行为特性）。

可以将该应用程序视为包含两个单元格的超小型电子表格。第一个单元格是个可写入的单元格，它是作为编辑控件（标注为 Count To ）来实现的。第二个单元格是个只读单元格，它是作为一个静态控件（标注为 Answer ）来实现的。当改变编辑控件中的数字时，Answer单元格就会自动重新计算。对于这个简单的应用程序来说，重新计算是由计数器来进行的，它的起始数字是 0，然后慢慢递增，直到 Answer 单元格中的值与输入的数字相同为止。为了演示的需要，对话框底部的静态控件中的数字不断更新，以显示当前执行的是哪个纤程。该纤程既可以是用户界面纤程，也可以是重新计算的纤程。

为了测试该应用程序的运行情况，可以将 5 键入编辑控件。Currently Running Fiber（当前运行的纤程）域改为 Recalculation（重新计算），Answer 域中的数字慢慢地从 0 递增至 5。当计数完成时，Currently Running Fiber 域重新改为 User Interface（用户界面），线程转入睡眠状态。这时，在编辑框中，在 5 后面键入 0（使数字变为 50），然后观察计数从 0 开始，逐步递增至 50。不过这次在 Answer 域中的数字递增的同时，你移动屏幕上的窗口。你将会发现，重新计算的纤程暂停运行，而用户界面纤程重新被调度，使应用程序的用户界面保持对用户的响应状态。当

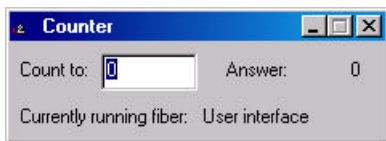


图12-1 Counter 对话框

你停止移动窗口时，重新计算的线程又被重新调度，而 Answer域则从上次暂停的地方继续开始计数。

最后一个测试项是，当重新计算线程正在计数时，改变编辑框中的数字。同样，你会看到用户界面对你的输入作出响应，但是你也会看到当你停止键入时，重新计算线程开始从头计数。这就是你在功能完善的电子表格应用程序中需要的那种行为特性。

记住，这个应用程序中没有使用任何关键代码段或其他线程同步对象，一切都是使用由两个线程组成的单线程来完成的。

下面让我们来说明这个应用程序是如何实现的。当进程的主线程通过执行 `_tWinMain`（在程序清单的结尾处）开始运行时，`ConvertThreadToFiber`函数被调用，以便将线程转换成线程，并且允许我们在以后创建另一个线程。然后，创建一个无模式对话框，它是应用程序的主对话框，接着，一个状态变量被初始化，指明后台处理的状态（`BPS`）。该状态变量是全局变量 `g_FiberInfo`中包含的 `bps`成员。如表 12-1 所示，共有 3 个状态。

表 12-1 全局变量 `g_FiberInfo` 中 `DPS` 成员的状态

状 态	描 述
<code>BPS_DONE</code>	重新计算即将完成，用户没有修改需要重新计算的任何东西
<code>BPS_STARTOVER</code>	用户修改了某些东西，因此需要从头开始重新计算
<code>BPS_CONTINUE</code>	重新计算已经开始，但是尚未完成。另外，用户没有修改需要从头开始重新计算的任何东西

后台处理的状态变量是在线程的消息循环中进行观察的，该消息循环比普通的消息循环更加复杂。下面是消息循环的作用：

- 如果存在一个窗口消息（用户界面处于活动状态）。那么它就负责处理该消息。保持用户界面的响应特性的优先级总是高于重新计算的优先级。
- 如果用户界面无事可做，它可以查看是否需要任何重新计算操作（后台处理状态是 `BPS_STARTOVER` 或 `BPS_CONTINUE`）。
- 如果没有任何重新计算操作需要执行（`BPS_DONE`），它就通过调用 `WaitMessage` 函数暂停线程的运行；只有用户界面事件能够导致需要进行的重新计算。

如果用户界面线程无事可做，同时，用户刚刚修改了编辑框中的值，那么需要从头开始重新计算（`BPS_STARTOVER`）。首先必须了解，我们可能已经有一个重新计算线程正在运行。如果是这种情况，必须删除该线程，并且创建一个从头开始计数的新线程。用户界面线程调用 `DeleteFiber` 函数来撤消现有的重新计算线程。这正是线程（比线程更加）便于使用的一个表现。删除重新计算的线程是完全没有问题的，该线程的内存栈和执行环境将被完全彻底地释放。如果使用线程而不是线程，那么用户界面线程将不会完全彻底地删除重新计算的线程，必须使用某种形式的线程之间的通信手段，并且等待重新计算的线程自行终止运行。一旦知道不再存在重新计算的线程，就可以创建新的重新计算线程，并将后台处理状态设置为 `BPS_CONTINUE`。

当用户界面空闲而重新计算的线程有事可做时，可以通过调用 `SwitchToFiber` 函数为它调度所需的时间。`SwitchToFiber` 函数要等到重新计算的线程再次调用 `SwitchToFiber` 函数并传递用户界面线程的执行环境的地址时才会返回。

`FiberFunc` 函数包含了重新计算线程执行的代码。该线程函数将得到全局结构 `g_FiberInfo` 的地址，因此它知道对话框窗口的句柄、用户界面线程的执行环境的地址，以及当前后台处理的状态。该结构的地址不需要传递，因为它是位于一个全局变量之中，但是我想要展示如何将参数传递给线程函数。此外，传递该地址可以较少地依赖代码，这总是一种好的做法。

纤程函数首先更新对话框中的状态控件，以便指明重新计算纤程正在运行。然后它要获取编辑框中的数字，并进入一个循环，从 0 开始计数，直到计算到编辑框中的这个数字。每当递增到接近这个数字的时候，便调用 GetQueueStatus 函数，以了解线程的消息队列中是否显示有任何消息（单个线程上运行的所有纤程均共享线程的消息队列）。当显示一条消息时，用户界面线程就有事可做了。由于我们希望它拥有高于重新计算的优先级，因此立即调用 SwitchToFiber 函数，使用户界面纤程能够处理该消息。当消息处理完毕后，用户界面纤程便重新调度该重新计算的纤程（如前面介绍的那样），同时，后台处理继续进行。

当没有消息需要处理时，重新计算纤程便更新对话框中的 Answer 域的数字，然后睡眠 200ms。在实际的代码中，应该删除对 Sleep 的调用，我在这里加上了对 Sleep 的调用，是为了强调展示进行重新计算所需要的时间。

当重新计算纤程完成 Answer 域数字的计算时，后台处理状态变量被设置为 BPS_DONE，同时，对 SwitchToFiber 函数的调用将对用户界面纤程进行重新调度。这时，如果用户界面纤程无事可做，它将调用 WaitMessage，暂停线程的运行，使得 CPU 时间不会浪费。

清单12-1 Counter 示例应用程序

1,2,3...

Counter.cpp

```

/*****
Module: Counter.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

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

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

// The possible state of the background processing
typedef enum {
    BPS_STARTOVER, // Start the background processing from the beginning.
    BPS_CONTINUE,  // Continue the background processing.
    BPS_DONE       // There is no background processing to do.
} BKGNDPROCSTATE;

typedef struct {
    PVOID pFiberUI; // User interface fiber execution context
    HWND hwnd;      // Handle of main UI window
    BKGNDPROCSTATE bps; // State of background processing
} FIBERINFO, *PFIBERINFO;

// A global that contains application state information. This
// global is accessed directly by the UI fiber and indirectly

```

```

// by the background processing fiber.
FIBERINFO g_FiberInfo;

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

void WINAPI FiberFunc(PVOID pvParam) {
PFIBERINFO pFiberInfo = (PFIBERINFO) pvParam;

// Update the window showing which fiber is executing.
SetDlgItemText(pFiberInfo->hwnd, IDC_FIBER, TEXT("Recalculation"));

// Get the current count in the EDIT control.
int nCount = GetDlgItemInt(pFiberInfo->hwnd, IDC_COUNT, NULL, FALSE);

// Count from 0 to nCount, updating the STATIC control.
for (int x = 0; x <= nCount; x++) {

    // UI events have higher priority than counting.
    // If there are any UI events, handle them ASAP.
    if (HIWORD(GetQueueStatus(QS_ALLEVENTS)) != 0) {

        // The UI fiber has something to do; temporarily
        // pause counting and handle the UI events.
        SwitchToFiber(pFiberInfo->pFiberUI);

        // The UI has no more events; continue counting.
        SetDlgItemText(pFiberInfo->hwnd, IDC_FIBER, TEXT("Recalculation"));
    }

    // Update the STATIC control with the most recent count.
    SetDlgItemInt(pFiberInfo->hwnd, IDC_ANSWER, x, FALSE);

    // Sleep for a while to exaggerate the effect; remove
    // the call to Sleep in production code.
    Sleep(200);
}

// Indicate that counting is complete.
pFiberInfo->bps = BPS_DONE;

// Reschedule the UI thread. When the UI thread is running
// and has no events to process, the thread is put to sleep.
// NOTE: If we just allow the fiber function to return,
// the thread and the UI fiber die -- we don't want this!
SwitchToFiber(pFiberInfo->pFiberUI);
}

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

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

    chSETDLGICONS(hwnd, IDI_COUNTER);
}

```

```

SetDlgItemInt(hwnd, IDC_COUNT, 0, FALSE);
return(TRUE);
}

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

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {
        case IDCANCEL:
            PostQuitMessage(0);
            break;

        case IDC_COUNT:
            if (codeNotify == EN_CHANGE) {

                // When the user changes the count, start the
                // background processing over from the beginning.
                g_FiberInfo.bps = BPS_STARTOVER;
            }
            break;
    }
}

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

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        case WM_INITDIALOG: Dlg_OnInitDialog(hwnd, wParam, lParam);
        case WM_COMMAND:    Dlg_OnCommand(hwnd, wParam, lParam);
    }
    return(FALSE);
}

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

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

    // Counter fiber execution context
    PVOID pFiberCounter = NULL;

    // Convert this thread to a fiber.
    g_FiberInfo.pFiberUI = ConvertThreadToFiber(NULL);

    // Create the application's UI window.
    g_FiberInfo.hwnd = CreateDialog(hinstExe, MAKEINTRESOURCE(IDD_COUNTER),
        NULL, Dlg_Proc);

    // Update the window showing which fiber is executing.
    SetDlgItemText(g_FiberInfo.hwnd, IDC_FIBER, TEXT("User interface"));
}

```

```

// Initially, there is no background processing to be done.
g_FiberInfo.bps = BPS_DONE;

// While the UI window still exists...
BOOL fQuit = FALSE;
while (!fQuit) {

    // UI messages are higher priority than background processing.
    MSG msg;
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {

        // If a message exists in the queue, process it.
        fQuit = (msg.message == WM_QUIT);
        if (!IsDialogMessage(g_FiberInfo.hwnd, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

    } else {

        // No UI msgs exist; check the state of the background processing.
        switch (g_FiberInfo.bps) {
            case BPS_DONE:
                // No background processing to do; wait for a UI event.
                WaitMessage();
                break;

            case BPS_STARTOVER:
                // User changed the count; restart the background processing.
                if (pFiberCounter != NULL) {
                    // A recalculation fiber exists; delete it so that
                    // background processing starts over from the beginning.
                    DeleteFiber(pFiberCounter);
                    pFiberCounter = NULL;
                }

                // Create a new recalc fiber that starts from the beginning.
                pFiberCounter = CreateFiber(0, FiberFunc, &g_FiberInfo);

                // The background processing started; it should continue.
                g_FiberInfo.bps = BPS_CONTINUE;

                // Fall through to BPS_CONTINUE case...

            case BPS_CONTINUE:
                // Allow the background processing to execute...
                SwitchToFiber(pFiberCounter);

                // The background processing has been paused
                // (because a UI message showed up) or has been
                // stopped (because the counting has completed).

                // Update the window showing which fiber is executing.
                SetDlgItemText(g_FiberInfo.hwnd, IDC_FIBER,
                    TEXT("User interface"));
        }
    }
}

```



```

        if (g_FiberInfo.bps == BPS_DONE) {
            // The background processing ran to completion. Delete the
            // fiber so that processing will restart next time.
            DeleteFiber(pFiberCounter);
            pFiberCounter = NULL;
        }
        break;
    } // switch on background processing state

    } // No UI messages exist
} // while the window still exists
DestroyWindow(g_FiberInfo.hwnd);

return(0); // End the application.
}

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

```

Counter.rc

```

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

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

////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

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

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

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

IDD_COUNTER_DIALOG DISCARDABLE 0, 0, 156, 37
STYLE DS_3DLOOK | DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION |
    WS_SYSMENU
CAPTION "Counter"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT        "Count to:", IDC_STATIC, 4, 6, 34, 8
    EDITTEXT     IDC_COUNT, 38, 4, 40, 14, ES_AUTOHSCROLL | ES_NUMBER

```

```

LTEXT          "Answer:", IDC_STATIC, 90, 6, 25, 8
RTEXT          "0", IDC_ANSWER, 122, 6, 23, 8
LTEXT          "Currently running fiber:", IDC_STATIC, 4, 24, 75, 8
LTEXT          "Fiber", IDC_FIBER, 80, 24, 72, 8

END

////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_COUNTER, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 149
        TOPMARGIN, 7
        BOTTOMMARGIN, 30
    END
END
#endif // APSTUDIO_INVOKED

#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

////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon

```

```
// remains consistent on all systems.
IDI_COUNTER          ICON    DISCARDABLE    "Counter.ico"
#endif              // English (U.S.) resources
////////////////////////////////////

#ifndef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

////////////////////////////////////
#endif              // not APSTUDIO_INVOKED
```
