

第16章 线程的堆栈

有时系统会在你自己进程的地址空间中保留一些区域。第 3 章讲过，对于进程和线程环境块来说，就会出现这种情况。另外，系统也可以在你自己进程的地址空间中为线程的堆栈保留一些区域。

每当创建一个线程时，系统就会为线程的堆栈（每个线程有它自己的堆栈）保留一个堆栈空间区域，并将一些物理存储器提交给这个已保留的区域。按照默认设置，系统保留 1 MB 的地址空间并提交两个页面的内存。但是，这些默认值是可以修改的，方法是在你链接应用程序时设定 Microsoft 的链接程序的 /STACK 选项：

```
/STACK:reserve[,commit]
```

当创建一个线程的堆栈时，系统将会保留一个链接程序的 /STACK 开关指明的地址空间区域。但是，当调用 CreateThread 或 _beginthreadex 函数时，可以重载原先提交的内存数量。这两个函数都有一个参数，可以用来重载原先提交给堆栈的地址空间的内存数量。如果设定这个参数为 0，那么系统将使用 /STACK 开关指明的已提交的堆栈大小值。后面将假定我们使用默认的堆栈大小值，即 1 MB 的保留区域，每次提交一个页面的内存。

图 16-1 显示了在页面大小为 4 KB 的计算机上的一个堆栈区域的样子（保留的起始地址是 0x08000000）。该堆栈区域和提交给它的所有物理存储器均拥有页面保护属性 PAGE_READWRITE。

内存地址	页面状态
0x080FF000	堆栈顶部：已提示的页面
0x080FD000	带有保护属性标志的已提交页面
0x080FD000	保留页面
0x08003000	保留页面
0x08002000	保留页面
0x08001000	保留页面
0x08000000	堆栈底部：保留页面

图 16-1 线程的堆栈区域刚刚创建时的样子

当保留了这个区域后，系统将物理存储器提交给区域的顶部的两个页面。在允许线程启动运行之前，系统将线程的堆栈指针寄存器设置为指向堆栈区域的最高页面的结尾处（一个非常接近 0x08100000 的地址）。这个页面就是线程开始使用它的堆栈的位置。从顶部向下的第二个页面称为保护页面。当线程调用更多的函数来扩展它的调用树状结构时，线程将需要更多的堆栈空间。

每当线程试图访问保护页面中的存储器时，系统就会得到关于这个情况的通知。作为响应，系统将提交紧靠保护页面下面的另一个存储器页面。然后，系统从当前保护页面中删除保护页面的保护标志，并将它赋予新提交的存储器页面。这种方法使得堆栈存储器只有在线程需要时才会增加。最终，如果线程的调用树继续扩展，堆栈区域就会变成图 16-2 所示的样子。

如图 16-2 所示，假定线程的调用树非常深，堆栈指针 CPU 寄存器指向堆栈内存地址 0x08003004。这时，当线程调用另一个函数时，系统必须提交更多的物理存储器。但是，当系统将物理存储器提交给 0x08001000 地址上的页面时，系统执行的操作与它给堆栈的其他内存区域提交物理存储器时的操作并不完全一样。图 16-3 显示了堆栈的保留内存区域的样子。

如你预计的那样，从地址 0x08002000 开始的页面的保护属性已经被删除，物理存储器被提交给从 0x08001000 地址开始的页面。它们的差别是，系统并不将保护属性应用于新的物理存储器页面（0x08001000）。这意味着该堆栈已保留的地址空间区域包含了它能够包含的全部物理存储器。最底下的页面总是被保留的，从来不会被提交。下面将要说明它的原因。

当系统将物理存储器提交给 0x08001000 地址上的页面时，它必须再执行一个操作，即它要引发一个 EXCEPTION_STACK_OVERFLOW 异常处理（在 WinNT.h 文件中定义为 0xC00000FD）。通过使用结构化异常处理（SEH），你的程序将能得到关于这个异常处理条件的通知，并且能够实现适度恢复。关于 SEH 的详细说明，请参见第 23、24 和 25 章的内容。本章结尾处的 Summation 示例应用程序将展示如何对堆栈溢出进行适度恢复。

内存地址	页面状态
0x080FF000	堆栈顶部：已提交的页面
0x080FD000	已提交的页面
0x080FD000	已提交的页面
0x08003000	已提交的页面
0x08002000	带有保护属性标志的已提交页面
0x08001000	保留页面
0x08000000	堆栈底部：保留页面

图16-2 几乎完整的线程堆栈区域

内存地址	页面状态
0x080FF000	堆栈顶部：已提交的页面
0x080FE000	已提交的页面
0x080FD000	已提交的页面
0x08003000	已提交的页面
0x08002000	已提交的页面
0x08001000	已提交的页面
0x08000000	堆栈底部：保留页面

图16-3 完整的线程堆栈区域

如果在出现堆栈溢出异常条件之后，线程继续使用该堆栈，那么在 0x08001000 地址上的页面中的全部内存均将被使用，同时，该线程将试图访问从 0x08000000 开始的页面中的内存。当该线程试图访问这个保留的（未提交的）内存时，系统就会引发一个访问违规异常条件。如果在线程试图访问该堆栈时引发了这个访问违规异常条件，线程就会陷入很大的麻烦之中。这时，系统就会接管控制权，并终止进程的运行——不仅终止线程的运行，而终止整个进程的运行。系统甚至不向用户显示一个消息框，整个进程都消失了！

下面要说明为什么堆栈区域的最后一个页面始终被保留着。这样做的目的是为了防止不小心改写进程使用的其他数据。可以看到，在 0x07FF000 这个地址上（0x08000000 下面的一个页面），另一个地址空间区域已经提交了物理存储器。如果 0x08000000 地址上的页面包含物理存储器，系统将无法抓住线程访问已保留堆栈区域的尝试。如果堆栈深入到已保留堆栈区域的下面，那么线程中的代码就会改写进程的地址空间中的其他数据，这是个非常难以抓住的错误。

16.1 Windows 98 下的线程堆栈

在 Windows 98 下，堆栈的行为特性与 Windows 2000 下的堆栈非常相似。但是它们之间存在某些重大的差别。

图16-4显示了 Windows 98 下 1 MB 的堆栈的各个区域的样子（从 0x00530000 地址上开始保留）。

首先请注意，尽管我们想要创建的堆栈大小最大只有 1 MB，但是堆栈区域的大小实际上是 1 MB 加 128 KB。在 Windows 98 中，每当为一个堆栈保留一个区域时，系统保留的区域实际上比要求的尺寸要大 128 KB。该堆栈位于该区域的中间，堆栈的前面有一个 64 KB 的块，堆栈的后面是另一个 64 KB 的块。

内存地址	大小	页面状态
0x00640000	16页面 (65 536字节)	堆栈顶部：保留供堆栈下溢时使用
0x0063F000	1页面 (4096字节)	带有PAGE_READWRITE保护属性的已提交页面，堆栈在用
0x0063E000	1页面 (4096字节)	仿真PAGE_GUARD标志的PAGE_NO_ACCESS页面
0x00638000	6页面 (24 576字节)	保留供堆栈溢出时使用的页面
0x00637000	1页 (4096字节)	带有PAGE_READ_WRITE保护属性的已提交页面，用于与16位组件相兼容
0x006540000	247页面 (1 011 712字节)	保留页面，供堆栈扩展时使用
0x00530000	16页面 (65 536字节)	堆栈底部：保留供堆栈溢出时使用

图16-4 Windows 98下线程的堆栈区域刚刚创建时的样子

堆栈开始处的 64 KB 用于抓取堆栈的溢出条件，而堆栈后面的 64 KB 则用于抓取堆栈的下溢条件。若要了解为什么需要检测堆栈下溢条件，请看下面这个代码段：

```
int WINAPI WinMain (HINSTANCE hinstExe, HINSTANCE,
    PSTR pszCmdLine, int nCmdShow) {

    char szBuf[100];
    szBuf[10000] = 0; // Stack underflow

    return(0);
}
```

当该函数的赋值语句执行时，便尝试访问线程堆栈结尾处之外的内存。当然，编译器和链接程序不会抓住上面代码中的错误，但是，如果应用程序是在 Windows 98 下运行，那么当该语句执行时，就会引发访问违规。这是 Windows 98 的一个出色特性，而 Windows 2000 是没有的。在 Windows 2000 中，可以在紧跟线程堆栈的后面建立另一个区域。如果出现这种情况，并且你试图访问你的堆栈外面的内存，那么你将破坏与进程的另一个部分相关的内存，而系统将不会发现这个情况。

需要指出的第二个重要差别是，没有一个页面具有 PAGE_GUARD 保护属性标志。由于 Windows 98 不支持这个标志，所以它使用一个不同的方法来扩展线程的堆栈。Windows 98 将紧靠堆栈下面的已提交页面标记为 PAGE_NOACCESS 保护属性（图 16-4 中的地址 0x0063E000）。然后，当线程接触读/写页面下面的页面时，将会发生访问违规。系统抓住这个访问违规，将不能访问的页面改为读写页面，并提交前一个保护页面下面的一个新保护页面。

第三个应该注意的差别是图 16-4 中的 0x00637000 地址上的单个 PAGE_READWRITE 内存页面。这个页面是为了实现与 16 位 Windows 相兼容而存在的。虽然 Microsoft 从未将它纳入文档，但是开发人员发现 16 位应用程序的堆栈段（SS）开始处的 16 个字节包含了关于 16 位应用程序的堆栈、本地堆栈和本地原子表的信息。由于在 Windows 98 上运行的 Win32 应用程序常常调用 16 位 DLL 组件，有些 16 位组件认为这些信息可以在堆栈段的开始处得到，因此 Microsoft 不得不在 Windows 98 中仿真这些字节的设置。当 32 位代码转换为 16 位代码时，Windows 98 将把一个 16

位CPU选择器映射到32位堆栈，并且将堆栈段寄存器设置为指向0x00637000地址上的页面。这时该16位代码就可以访问堆栈段的开始处的16个字节，并且可以继续运行而不会出任何问题。

现在，当Windows 98扩大它的线程堆栈时，它将继续扩大0x0063F000地址上的内存块。它也会不断地将保护页面向下移，直到1 MB的堆栈内存被提交为止。然后保护页面消失，就像在Windows 2000下运行的情况一样。系统还继续为了16位Windows组件的兼容性而将页面向下移，最后该页面将进入堆栈区域开始处的64 KB的内存块中。因此，Windows 98中一个完全提交的堆栈将类似图16-5所示的样子。

内存地址	大小	页面状态
0x00640000	16页面 (65 536字节)	堆栈顶部：保留供堆栈下溢时使用
0x00540000	256页面 (1MB)	带有PAGE_READWRITE保护属性的已提交页面，堆栈在用
0x00539000	7页面 (28 672字节)	供堆栈溢出时使用的保留页面
0x00538000	1页面 (4096字节)	带有PAGE_READWRITE保护属性的已提交页面，用于与16位组件相兼容
0x00538000	8页面 (32 768字节)	堆栈底部：保留供堆栈溢出时使用

图16-5 Windows 98下的一个完整的线程堆栈区域

16.2 C/C++运行期库的堆栈检查函数

C/C++运行期库包含一个堆栈检查函数。当编译源代码时，编译器将在必要时自动生成对该函数的调用。堆栈检查函数的作用是确保页面被适当地提交给线程的堆栈。下面让我们来看一个例子。

这是一个小型函数，它需要相当多的内存用于它的局部变量：

```
void SomeFunction () {  
    int nValues[4000];  
  
    // Do some processing with the array.  
    nValues[0] = 0; // Some assignment  
}
```

该函数至少需要16 000个字节（4000 x sizeof(int),每个整数是4个字节）的堆栈空间，以便放置整数数组。通常情况下，编译器生成的用于分配该堆栈空间的代码只是将CPU的堆栈指针递减16 000个字节。但是，在程序试图访问内存地址之前，系统并不将物理存储器分配给堆栈区域的这个较低区域。

在使用4 KB或8 KB页面的系统上，这个局限性可能导致一个问题出现。如果初次访问堆栈是在低于保护页面的一个地址上进行的（如上面这个代码中的赋值行所示），那么线程将访问已经保留的内存并且引发访问违规。为了确保能够成功地编写上面所示的函数，编译器将插入对C运行期库的堆栈检查函数的调用。

当编译程序时，编译器知道你针对的CPU系统的页面大小。x86编译器知道页面大小是4 KB，Alpha编译器知道页面大小是8 KB。当编译器遇到程序中的每个函数时，它能确定该函数

需要的堆栈空间的数量。如果该函数需要的堆栈空间大于目标系统的页面大小，编译器将自动插入对堆栈检查函数的调用。

下面这个伪代码显示了堆栈检查函数执行什么操作。之所以称它是伪代码，是因为这个函数通常是由编译器供应商用汇编语言来实现的：

```
// The C run-time library knows the page size for the target system.
#ifdef _M_ALPHA
#define PAGESIZE (8 * 1024) // 8-KB page
#else
#define PAGESIZE (4 * 1024) // 4-KB page
#endif

void StackCheck(int nBytesNeededFromStack) {
    // Get the stack pointer position.
    // At this point, the stack pointer has NOT been decremented
    // to account for the function's local variables.
    PBYTE pbStackPtr = (CPU's stack pointer);
    while (nBytesNeededFromStack >= PAGESIZE) {
        // Move down a page on the stack--should be a guard page.
        pbStackPtr -= PAGESIZE;

        // Access a byte on the guard page--forces new page to be
        // committed and guard page to move down a page.
        pbStackPtr[0] = 0;

        // Reduce the number of bytes needed from the stack.
        nBytesNeededFromStack -= PAGESIZE;
    }

    // Before returning, the StackCheck function sets the CPU's
    // stack pointer to the address below the function's
    // local variables.
}
```

Microsoft 的 Visual C++ 确实提供了一个编译器开关，使你能够控制一个页面大小的阈值，这个阈值可供编译器用来确定何时添加对 StackCheck 函数的自动调用。只有当确切地知道究竟在进行什么操作并且有着特殊需要时，才能使用这个编译器开关。对于绝大多数应用程序和 DLL 来说，都不应该使用这个开关。

16.3 Summation 示例应用程序

本章后面清单 16-1 中的 Summation (“ 16 Summation.exe ”) 示例应用程序展示了如何使用异常过滤器和异常处理程序以便对堆栈溢出进行适度恢复的方法。该应用程序的源代码和资源文件均位于本书所附光盘上的 16-Summation 目录下。若要全面了解该应用程序是如何运行的，可以参见关于 SEH 的有关章节。

Summation 应用程序用于计算从 0 到 x 的全部数字的总和，其中 x 是用户输入的一个数字。当然，进行这项操作的最简单的方法是创建一个称为 Sum 的函数，它只是进行下面的计算：

```
Sum = (x * (x + 1)) / 2;
```

然而对于这个例子来说，我将 Sum 编写为一个递归函数，这样，如果输入较大的数字，它将使用大量的堆栈空间。

当程序启动运行时，它将显示图 16-6 所示的对话框。

在这个对话框中, 你将一个数字输入编辑控件, 然后单击 Calculate 按钮。这使程序创建一个新线程, 该线程的唯一作用是将 0 到 x 的全部数字进行相加。当这个新线程运行时, 程序的主线程通过调用 WaitForSingleObject 函数并传递新线程的句柄, 等待线程运行的结果。当新线程运行终止时, 系统就唤醒主线程。主线程取出合计的总数, 方法是调用 GetExitCodeThread 函数来获得新线程的退出代码。最后, 最重要的一点是, 主线程要关闭新线程的句柄, 这样, 系统就能完全撤消线程对象, 并且使应用程序不会出现资源的泄漏。

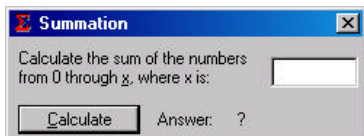


图16-6 Summation 对话框

这时, 主线程要查看合计线程的退出代码。退出代码 UNIT_MAX 指明出现了一个错误, 即合计线程在计算数字的总数时产生了堆栈溢出, 为此, 主线程显示一个消息框, 以说明这个情况。如果退出代码不是 UNIT_MAX, 那么合计线程将成功地结束其运行, 而退出代码则是合计。在这种情况下, 主线程只是将合计的结果放入对话框。

下面让我们转入合计线程的介绍。该线程的线程函数称为 SumThreadFunc。当主线程创建该线程时, 它将应该合计的各个整数的数量作为唯一的参数来传递, 这个参数就是 pvParam。然后, 函数将 uSum 变量初始化为 UNIX_MAX, 这意味着该函数认为它将不会成功地完成运行。接着, SumThreadFunc 建立 SEH, 这样, 它就能够抓住线程运行时出现的任何异常条件。然后调用递归函数 Sum 来计算总数。

如果成功地计算出总数, SumThreadFunc 函数返回 uSum 变量的值, 这是线程的退出代码。但是, 如果在 sum 函数运行时引发了一个异常条件, 系统将立即对 SEH 过滤器表达式进行计算。换句话说, 系统将调用 FilterFunc 函数, 并为它传递用于标识引发的异常条件的代码。如果是堆栈溢出异常, 那么该代码是 EXCEPTION_STACK_OVERFLOW。如果想要观察程序适度处理堆栈溢出的异常条件, 那么请告诉程序计算前面的 44 000 个数字的总数。

我的 FilterFunc 函数非常简单。它查看是否出现了堆栈溢出异常条件。如果没有出现, 它返回 EXCEPTION_CONTINUE_SEARCH。否则, 该过滤器返回 EXCEPTION_EXECUTE_HANDLER。它向系统指明过滤器预计到了这个异常条件, 同时, Except 块中包含的代码应该执行。对于这个示例应用程序来说, 异常处理程序没有什么特殊的操作需要执行, 而是让线程恰当地退出并返回代码 UNIT_MAX (这是 uSumNum 中的值)。父线程将会看到这个特殊的返回值, 并向用户显示一条警告消息。

要说明的最后一点是, 为什么要在 Sum 函数自己的线程中运行 Sum 函数, 而不是在主线程中建立一个 SEH 块, 并从 try 块中调用 Sum 函数。创建这个独立线程的理由有三:

首先, 每次创建一个线程时, 它会得到它自己的 1 MB 堆栈区域。如果我从主线程中调用 Sum 函数, 那么有些堆栈空间就已经在使用了, 因此 Sum 函数将无法使用完整的 1 MB 堆栈空间。然而, 我的示例应用程序是个简单的程序, 也许它不需要使用那么多完整的堆栈空间, 不过其他程序可能要复杂得多。我能够非常容易地想像到这样一种情况, 即 Sum 函数能够成功地计算出从 1 到 1000 的所有整数的总和。然后, 当 Sum 在以后被再次调用时, 堆栈可能变得更深, 从而在 Sum 试图只计算从 0 到 750 之间的整数的总和时, 导致堆栈溢出的发生。因此, 为了使 Sum 函数的运行具备更好的一致性, 我设法使它拥有一个尚未被其他代码使用过的完整的堆栈。

使用独立线程的第二个原因是, 关于堆栈溢出的异常条件, 线程只能得到一次通知。如果我调用主线程中的 Sum 函数, 并且发生了堆栈溢出, 那么就可以跟踪和恰当地处理该异常条件。但是, 这时已经向堆栈的所有已保留地址空间提交了物理存储器, 并且没有更多的带有已打开的保护属性标志的页面。如果用户执行另一个总数的计算, Sum 函数就会使堆栈溢出, 但是不会引发堆栈溢

出异常条件。相反，一个访问违规异常将会发生，而这时恰当地处理这个异常条件就太晚了。

使用独立的线程的最后一个原因是，该堆栈的物理存储器可以释放。请看下面这个例子。用户要求Sum函数计算从0到30 000之间的整数的总和。这需要将相当数量的物理存储器提交给堆栈区域。然后，用户要进行若干个合计操作，其中最高的数字是 5000。在这种情况下，大量的内存被提交给堆栈区域，但是却不再被使用。该物理存储器是从页文件那里分配来的。不应该使该物理存储器保持提交状态，最好是释放该内存，重新将它交给系统和其他进程。通过使SumThreadFunc的线程终止运行，系统将自动收回已经提交给堆栈区域的物理存储器。

清单16-1 Summation示例应用程序



Summation.cpp

```

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

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

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

// An example of calling Sum for uNum = 0 through 9
// uNum: 0 1 2 3 4 5 6 7 8 9 ...
// Sum:  0 1 3 6 10 15 21 28 36 45 ...
UINT Sum(UINT uNum) {

    // Call Sum recursively.
    return((uNum == 0) ? 0 : (uNum + Sum(uNum - 1)));
}

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

LONG WINAPI FilterFunc(DWORD dwExceptionCode) {

    return((dwExceptionCode == STATUS_STACK_OVERFLOW)
        ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);
}

////////////////////////////////////
// The separate thread that is responsible for calculating the sum.
// I use a separate thread for the following reasons:
// 1. A separate thread gets its own 1 MB of stack space.
// 2. A thread can be notified of a stack overflow only once.
// 3. The stack's storage is freed when the thread exits.
DWORD WINAPI SumThreadFunc(PVOID pvParam) {

    // The parameter pvParam, contains the number of integers to sum.
    UINT uSumNum = PtrToUlong(pvParam);

```



```

// uSum contains the summation of the numbers from 0 through uSumNum.
// If the sum cannot be calculated, a sum of UINT_MAX is returned.
UINT uSum = UINT_MAX;

__try {
    // To catch the stack overflow exception, we must
    // execute the Sum function while inside an SEH block.
    uSum = Sum(uSumNum);
}
__except (FilterFunc(GetExceptionCode())) {
    // If we get in here, it's because we have trapped a stack overflow.
    // We can now do whatever is necessary to gracefully continue execution.
    // This sample application has nothing to do, so no code is placed
    // in this exception handler block.
}

// The thread's exit code is the sum of the first uSumNum
// numbers, or UINT_MAX if a stack overflow occurred.
return(uSum);
}

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

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

    chSETDLGICONS(hwnd, IDI_SUMMATION);

    // Don't accept integers more than 9 digits long.
    Edit_LimitText(GetDlgItem(hwnd, IDC_SUMNUM), 9);

    return(TRUE);
}

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

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

    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case IDC_CALC:
            // Get the number of integers the user wants to sum.
            UINT uSum = GetDlgItemInt(hwnd, IDC_SUMNUM, NULL, FALSE);
            // Create a thread (with its own stack) that is
            // responsible for performing the summation.
            DWORD dwThreadId;
            HANDLE hThread = chBEGINTHREADEX(NULL, 0,
                SumThreadFunc, (PVOID) (UINT_PTR) uSum, 0, &dwThreadId);

            // Wait for the thread to terminate.
            WaitForSingleObject(hThread, INFINITE);

            // The thread's exit code is the resulting summation.
            GetExitCodeThread(hThread, (PDWORD) &uSum);
    }
}

```

```

        // Allow the system to destroy the thread kernel object
        CloseHandle(hThread);

        // Update the dialog box to show the result.
        if (uSum == UINT_MAX) {
            // If result is UINT_MAX, a stack overflow occurred.
            SetDlgItemText(hwnd, IDC_ANSWER, TEXT("Error"));
            chMB("The number is too big, please enter a smaller number");
        } else {
            // The sum was calculated successfully;
            SetDlgItemInt(hwnd, IDC_ANSWER, uSum, FALSE);
        }
        break;
    }
}

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

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);
    }
    return(FALSE);
}

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

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

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

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

```

Summation.rc

```

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

#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.

```

```
//
#define APSTUDIO_HIDDEN_SYMBOLS
#include "windows.h"
#undef APSTUDIO_HIDDEN_SYMBOLS

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

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

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

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

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

IDD_SUMMATION_DIALOG DISCARDABLE 18, 18, 162, 41
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Summation"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT          "Calculate the sum of the numbers from 0 through &x, where x is: ",
        IDC_STATIC,4,4,112,20
    EDITTEXT       IDC_SUMNUM,120,8,40,13,ES_AUTOHSCROLL
    DEFPUSHBUTTON  "&Calculate",IDC_CALC,4,28,56,12
    LTEXT          "Answer:",IDC_STATIC,68,30,30,8
    LTEXT          "?",IDC_ANSWER,104,30,56,8
END

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

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

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#define APSTUDIO_HIDDEN_SYMBOLS\r\n"
```

```
"#include ""windows.h""\r\n"
"#undef APSTUDIO_HIDDEN_SYMBOLS\r\n"
"\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END
#endif    // APSTUDIO_INVOKED

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

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

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